Instituto Tecnológico y de Estudios Superiores de Occidente

Reconocimiento de validez oficial de estudios de nivel superior según acuerdo secretarial 15018, publicado en el Diario Oficial de la Federación del 29 de noviembre de 1976.

Departamento de Electrónica, Sistemas e Informática
Maestría en Diseño Electrónico

Implementation and evaluation of Branch Predictors on RISC-V

TRABAJO RECEPCIONAL que para obtener el **GRADO** de
MAESTRO EN DISEÑO ELECTRÓNICO

Presenta: Braulio Isaac Martínez Aceves

Director: Mtro. Leonel Haro Ruiz

Tlaquepaque, Jalisco 10 de Agosto de 2025

Dedicatoria

Con mucho cariño y eterno agradecimiento dedico este trabajo a todas aquellas personas que me han acompañado en este camino. En cada paso y decisión que me llevó a este momento, he tenido siempre el apoyo incondicional de mi familia, amistades, mentores y colegas. Sería imposible haber logrado esto sin ustedes y no hay palabras ni páginas suficientes para expresarles mi gratitud. Soy la suma de todo lo que comparten conmigo.

Gracias mamá, papá, hermana, hermano y abuelos.
Gracias Mtro. Leonel Haro Ruiz, Dr. Omar Humberto Longoria Gándara y profesorado del ITESO.
Gracias Itzury.
Gracias Ale, Oliver y Ricky.
Gracias Roberto, Carlos, Maik, Miguel y Chuyito.
Gracias equipo tapatío de Intel y AheadComputing.
Gracias Cassandra, Jorge y Diego.

Gracias a quienes estuvieron, a quienes siguen pero más que nada, gracias a quienes ya no están. Me encantaría que siguieras aquí; este trabajo fue escrito pensando en ti.

En memoria de Francisco Aceves y Carolina Longoria.

> *Así nomás quedó.*
> — Lagunas (2017)

Agradecimientos

Se agradece a Intel Tecnología de México, S. de R.L. de C.V. por el apoyo financiero y de infraestructura para la realización parcial de este trabajo.

Implementation and evaluation of Branch Predictors on RISC-V

by

Braulio Isaac Martínez Aceves

Abstract

This work studies three branch prediction schemes for a pipelined RV32I core with a Reorder Buffer: an N-bit counter, a Global History Register (GHR) indexed counter array, and Gshare. The design integrates a Branch Target Buffer split into unconditional and conditional structures. Each predictor is modeled in C++ using the CBP 2016 framework to measure MPKI under an 8 KB storage budget. The algorithms are implemented in SystemVerilog, integrated into the RTL, and synthesized on an Intel MAX 10 FPGA. We also document a reproducible bare-metal build and load flow for C code: cross-compilation, linker script setup, memory image generation, and RTL loading. RTL simulation evaluates IPC and total cycles on Dhrystone, Qsort, and CoreMark workloads.

C++ results show that Gshare reduces MPKI by 85.25% relative to a No BPU baseline with the same storage and outperforms GHR using the same counter and index settings. RTL results with a 15-bit index for simulation (7-bit index in synthesis due to resources) confirm the trend. Averaged across the three workloads, Gshare improves MPKI by 62.77%, IPC by 17.44%, and total cycles by 14.23% versus No BPU. Synthesis reports show modest hardware cost compared to baseline (about 38–40% more logic elements) and a small reduction in Fmax (about 8–10%). The results indicate that simple global-history predictors, and in particular Gshare, provide large accuracy and performance gains with minimal architectural complexity.

List of Figures

2.1 Abstract Branch Prediction diagram 5
2.2 5-Stage processor diagram with branch outcome resolution 5
2.3 5-Stage processor diagram with Execute misprediction recovery 5
2.4 Abstract Branch Prediction diagram with ROB 6
2.5 History Register as a Shift Register 8
2.6 Representation of Global History Register 8
2.7 Representation of Per-Branch (Local) History Register 8
2.8 Representation of a Directly-Mapped Branch Target Buffer (BTB) 9

3.1 5-Stage processor diagram with BPU mechanisms 13
3.2 Abstract Branch Prediction diagram with BRANCH_PACKET 14
3.3 Baseline Branch Prediction diagram with BTBs 17
3.4 FSM for 1-bit counter 20
3.5 FSM for 2-bit counter 20
3.6 N-bit Counter Branch Prediction diagram with BTBs 21
3.7 5-bit Global History Register as a Shift Register with update 31
3.8 Representation of an *N*-bit Counter Array indexed by the GHR 32
3.9 GHR with N-bit Counters Array Branch Prediction diagram 33
3.10 5-Stage processor diagram with GHR recovery mechanism 38
3.11 Representation of an *N*-bit Counter Array indexed by PC XOR GHR 43
3.12 Gshare BPU diagram with the *N*-bit counter array and connections to the ROB 45

List of Tables

2.1 Classification of Branch Types 3
2.2 Example of Context-Based Speculation Using Previous PCs and Branch Outcomes 4
2.3 Example of History Register Update 8

3.1 C++ CBP 2016 MPKI results of 1, 2, 3, 4, and 5 bit counter algorithms with memory requirements using a 7-bit index for the CBTB and n-bit counters array 28
3.2 Synthesis results for 1, 2, 3, and 4-bit counter algorithms showing $Fmax$ and resource usage using a 7-bit index for the CBTB, UBTB and n-bit counters array 29
3.3 C++ CBP 2016 MPKI results of 2 bit counter algorithm with memory requirements using a 15-bit index for the n-bit counters array 29
3.4 2bit counter Dhrystone 500 runs results (15-bit index, 8KB Global Counter Array) 30
3.5 2-bit counter Qsort 2048 numbers results (15-bit index, 8KB Global Counter Array) 30
3.6 2-bit counter CoreMark single run results (15-bit index, 8KB Global Counter Array) 30
3.7 2-bit counter arithmetic mean improvements across all workloads (15-bit index, 8KB Global Counter Array, No BPU baseline) 30
3.8 C++ CBP 2016 MPKI results of 1, 2, 3, 4, and 5 bit counter algorithms with GHR and memory requirements using 7-bits for CBTB index and GHR size . 39
3.9 C++ CBP 2016 MPKI results of 2 bit counter algorithm with GHR and memory requirements using 6 to 15 bits for GHR size 40
3.10 C++ CBP 2016 MPKI results comparison between 2-bit counter and GHR 2-bit counter algorithms with the same memory budget (8KB) 40
3.11 Synthesis results for 1, 2, 3, and 4-bit counter algorithms showing $Fmax$ and resource usage using a 7-bit index for the CBTB, UBTB, GHR and n-bit counters array 41
3.12 GHR Dhrystone 500 runs results (15-bit index, 8KB Global Counter Array) . 42
3.13 GHR Qsort 2048 numbers results (15-bit index, 8KB Global Counter Array) . 42
3.14 GHR CoreMark single run results (15-bit index, 8KB Global Counter Array) 42
3.15 GHR arithmetic mean improvements across all workloads (15-bit index, 8KB Global Counter Array, No BPU baseline) 42
3.16 Gshare MPKI Results for Different Counter Sizes with 7-bit GHR 48
3.17 Gshare MPKI Results for Different GHR Sizes with 2-bit Counter 49

3.18 C++ CBP 2016 MPKI results comparison between 2-bit counter, GHR 2-bit counter and Gshare 2-bit counter algorithms with the same memory budget (8KB) 49
3.19 Synthesis results for 1, 2, 3, and 4-bit counter algorithms showing $Fmax$ and resource usage using a 7-bit index for the CBTB, UBTB and n-bit counters array with Gshare 50
3.20 Gshare Dhrystone 500 runs results (15-bit index, 8KB Global Counter Array) 50
3.21 Gshare Qsort 2048 numbers results (15-bit index, 8KB Global Counter Array) 50
3.22 Gshare CoreMark single run results (15-bit index, 8KB Global Counter Array) 51
3.23 Gshare arithmetic mean improvements across all workloads (15-bit index, 8KB Global Counter Array with and without GHR, No BPU baseline) 51

List of Listings

1 C++ Predictor class methods . . . 10
2 C++ always-not-taken GetPrediction method . . . 11
3 C++ OpType enum definition . . . 11
4 SV BPU basic interface . . . 12
5 SV `BRANCH_PACKET_t` and `BRANCH_TYPE_t` basic definition . . . 13
6 SV BTB Parameters and data structures . . . 15
7 SV CBTB Interface . . . 15
8 SV CBTB Flip-Flop array generation . . . 16
9 SV BTB Speculation and Context . . . 16
10 SV CBTB Training Write Data logic . . . 18
11 SV CBTB Training Write Enable logic . . . 19
12 SV *N*-bits array declarations . . . 22
13 C++ *N*-bits array declarations . . . 23
14 C++ *N*-bits Speculation functions . . . 24
15 SV *N*-bits Speculation logic . . . 25
16 C++ *N*-bits Training logic . . . 26
17 SV *N*-bits Training logic . . . 27
18 SV `BRANCH_PACKET_t` with `BPU_HISTORY_t` definition and assignment . . . 34
19 SV GHR update code . . . 34
20 C++ GHR update code . . . 35
21 SV GHR Speculation logic . . . 36
22 C++ GHR speculation code . . . 37
23 SV GHR Recovery logic . . . 38
24 SV Gshare Speculation logic . . . 46
25 C++ Gshare speculation code . . . 47
26 Baremetal C startup code . . . 54
27 `_enter` function disassembly . . . 55
28 `_start` function disassembly . . . 56
29 `main` function disassembly . . . 56
30 `_Exit` function disassembly . . . 57
31 Linker `MEMORY` original definitions . . . 58
32 SV memory_pkg definitions . . . 58
33 Linker `MEMORY` modified definitions for RV32I RTL simulation . . . 59
34 ROM `objcopy` command . . . 60
35 RAM `objcopy` command . . . 60
36 xxd command to generate hex dump files for ROM and RAM . . . 60

37 SV `$readmemh` function 61
38 SV PC Check simulation finish logic 62
39 SV metrics collection logic 63
40 Example `CMakeLists.txt` file for a bare-metal C project 64

Contents

Abstract **3**

List of Figures **4**

List of Tables **6**

List of Listings **8**

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Problem Statement and Context . . . 1
1.3 Research question . . . 2
1.4 Solution Overview . . . 2

2 Theoretical Framework **3**
2.1 Speculation and context . . . 4
2.2 Recovery . . . 5
2.3 Training . . . 6
2.4 Direction Prediction . . . 7
2.4.1 History-Based Prediction . . . 7
2.5 Target Prediction . . . 9
2.5.1 Branch Target Buffer (BTB) . . . 9

3 Algorithm modeling and hardware implementation **10**
3.1 C++ CBP Framework concepts . . . 10
3.2 SystemVerilog implementation concepts . . . 12
3.2.1 Target prediction and BTB . . . 14
3.3 *N*-bits counter . . . 19
3.3.1 *N*-bit counters array . . . 21
3.3.2 Speculation and context . . . 23
3.3.3 Recovery . . . 25
3.3.4 Training . . . 25
3.3.5 C++ CBP 2016 MPKI Simulation results . . . 28
3.3.6 SystemVerilog FPGA Synthesis results . . . 28
3.3.7 SystemVerilog IPC and MPKI Simulation results . . . 29
3.3.8 Conclusion . . . 31

3.4 Global History Register (GHR) . . . 31
3.4.1 Speculation and Context . . . 32
3.4.2 Recovery . . . 37
3.4.3 Training . . . 39
3.4.4 C++ CBP 2016 MPKI Simulation results . . . 39
3.4.5 SystemVerilog FPGA Synthesis results . . . 41
3.4.6 SystemVerilog IPC and MPKI results . . . 41
3.4.7 Conclusion . . . 41
3.5 Gshare . . . 42
3.5.1 Speculation and Context . . . 43
3.5.2 Recovery . . . 47
3.5.3 Training . . . 48
3.5.4 C++ CBP 2016 MPKI Simulation results . . . 48
3.5.5 SystemVerilog FPGA Synthesis results . . . 50
3.5.6 SystemVerilog IPC and MPKI results . . . 50
3.5.7 Conclusion . . . 51

4 Compiling and running Baremetal C on RV32I RTL **52**
4.1 Introduction . . . 52
4.2 Startup code (`startup.c`) . . . 53
4.2.1 `_enter` function . . . 54
4.2.2 `_start` function . . . 55
4.2.3 `main` function . . . 56
4.2.4 `_Exit` function . . . 56
4.2.5 Conclusion . . . 57
4.3 Linker script (`linker.lds`) . . . 57
4.4 Compiling Baremetal C workloads and generating memory dumps . . . 59
4.4.1 Compilation . . . 59
4.4.2 Memory dump generation . . . 59
4.5 Running Baremetal C workloads on the RV32I RTL . . . 61
4.5.1 Workloads selected . . . 61
4.5.2 Loading memory dumps into the RTL simulator . . . 61
4.5.3 Stopping the simulation when the workload finishes . . . 61
4.5.4 Measuring MPKI, IPC and cycles . . . 62

A Appendix **64**
A.1 Baremetal `CMakeLists.txt` example . . . 64

Bibliography **68**

Chapter 1

Introduction

1.1 Motivation

High-performance *Central Processing Units* (*CPUs*) critically depend on their ability to extract *Instruction Level Parallelism* (*ILP*). Over the past few decades, increases in processing power and the number of in-flight instructions have enabled the development of wider, more parallel machine designs. However, extracting *ILP* is significantly constrained by the challenge of accurately handling control-flow dependencies, such as branches. As machines have grown wider with more execution resources and larger fetch windows, the number of pipeline stages has also increased. This deeper pipeline architecture amplifies the performance penalty when incorrect instructions are fetched, leading to wasted in-flight work[1].

These challenges show the importance of developing accurate branch predictors that can mitigate misprediction stalls and flushes, which are among the most common causes of pipeline waste in modern *CPUs*[2]. Although academic literature often dedicates entire sections to discussing branch predictors at a high level, concrete examples of their implementation in *Hardware Description Languages* (*HDLs*) are less common[3]. For instance, state-of-the-art predictors like *TAGE*[4] (used by AMD on their modern *Zen* architecture[5]) and *perceptron-based* predictors[6] show results and details of the algorithm on their publications based on high-level abstraction languages like C++. This leaves a gap between theoretical design and practical, hardware-level implementation, which might prove a significant challenge for anyone looking into implementing a branch predictor on hardware.

1.2 Problem Statement and Context

Branch predictors play a crucial role in *CPU* performance, and a thorough understanding of their design is essential for anyone studying modern *CPU* architectures. Despite extensive high-level academic material on the subject, there's an apparent gap when it comes to concrete *HDL* implementations of branch predictors. Due to this, documenting the detailed process of implementing a branch predictor in *HDL* and evaluating its performance becomes a relevant problem to solve. In doing so, it is possible to compare the theoretical performance predicted by high-level models with the practical hardware costs. By highlighting the trade-offs between abstract design and real-world hardware implementation, valuable insights can be achieved.

1.3 Research question

How can branch prediction algorithms be implemented on a *RISC-V RV32I* core using *SystemVerilog*, and what trade-offs in prediction accuracy and *FPGA* synthesis metrics (area and frequency) emerge when comparing these implementations to their ideal high-level C++ models and across different branch predictor designs?

1.4 Solution Overview

The goal of this work is to implement and evaluate various branch prediction mechanisms for a *RISC-V RV32I CPU*, leveraging both software-based simulation and hardware-based implementation. This work explores the effectiveness of 3 different branch prediction algorithms through high-level modeling in *C++* and low-level implementation in *SystemVerilog*, followed by performance evaluation using IPC and synthesis metrics.

- Theoretical Framework
  - Summarize the elements, concepts and challenges of branch prediction.
- Branch Prediction Algorithm Analysis and Modelling
  - Describe the working principles of different branch prediction algorithms.
  - Explain their modeling using the **CBP 2016 C++ Framework** [7].
  - Evaluate the effectiveness of each algorithm based on **Mispredicts Per Kilo Instructions (MPKI)**.
- Hardware Implementation in *SystemVerilog*
  - Describe the implementation for branch predictors in a *RISC-V CPU* pipeline.
  - Explain details for each function of the algorithm in SystemVerilog.
  - Measure the physical implications of the implemented branch predictors in terms of *FPGA* synthesis metrics: Logic Elements and Frequency.
  - Measure the performance of the implemented branch predictors in terms of *IPC* (Instructions Per Cycle), *MPKI* and *Cycles*.
- Compiling and Running **Baremetal C** on RV32I
  - Outline the requirements and challenges of compiling and executing baremetal C code.
  - Explain the toolchain used for *RV32I* compilation.
  - Describe the process of running compiled binaries in a *SystemVerilog* simulator.

Chapter 2

Theoretical Framework

Branch prediction is a fundamental technique used in modern superscalar processors to mitigate performance bottlenecks caused by control flow dependencies, particularly conditional branches[8]. By predicting the outcome and target of a branch instruction before execution, processors can speculatively fetch and execute subsequent instructions along the predicted path, improving instruction-level parallelism (ILP) and overall performance[9], [10].

Predicting a branch requires two components:

- **Direction Prediction**: Determines whether a branch will be taken or not taken.
- **Target Prediction**: Determines the target address of a taken branch.

Not all branches are equal, differentiation between **branch types** is relevant because each type might have specific needs that can be addressed to increase prediction accuracy. Table 2.1 shows and compares branch types.

Table 2.1: Classification of Branch Types

Type	**Mnemonic Examples**	**Direction**	**Branch outcome computation stage**
Unconditional	JAL, J	Always Taken (T)	Decode (PC + offset)
Unconditional Indirect	JALR	Always Taken (T)	Execute
Conditional	BEQ, BNE, BLT, BGE, BLTU, BGEU	Depends on Condition	Execute

Each prediction component requires mechanisms for **speculation and context**, **recovery** and **training** [9], [10]. These mechanisms will be explained before the main direction and target prediction components.

2.1 Speculation and context

On a high-level, branch predictors "guess" (predict) the next PC to fetch. Since they lack any decoding or executing capabilities, their predictions must be checked to ensure their correctness. The behavior of predicting multiple PC values until confirmed by decode/execute is called **speculation**, which is one of the main activities of a branch predictor and can be initially abstracted with the following formula:

$$PC_{N+1} = f(PC_N)$$

Where the next program counter PC_{N+1} is a function of the current program counter PC_N. Expanding on this concept, **speculation** can use more information besides the PC in order to make predictions. The general idea follows this principle:

*"The program **context** seen up to this point can help predict the next PC"*

The **context** can include information like previous taken branches and their targets helping identify patterns. The context at cycle N $Context_N$ is built from the context values and predictions from previous cycles.

$$Context_N = f[(Context_{N-1}, Prediction_{N-1}) \ldots, (Context_m, Prediction_m)]$$

$$Prediction_N = PC_{N+1}$$

$$Prediction_N = f(Context_N)$$

A larger m implies more context, which translates in larger storage requirements. It is also important to understand the prediction at cycle N $Prediction_N$ is a function of previous predictions.

Context information can also represent the *program path* seen so far, and it can be helpful for pattern recognition. As an example, Table 2.2 demonstrates how a $Context$ with $m = 2$ is built with historical PC values and branch information.

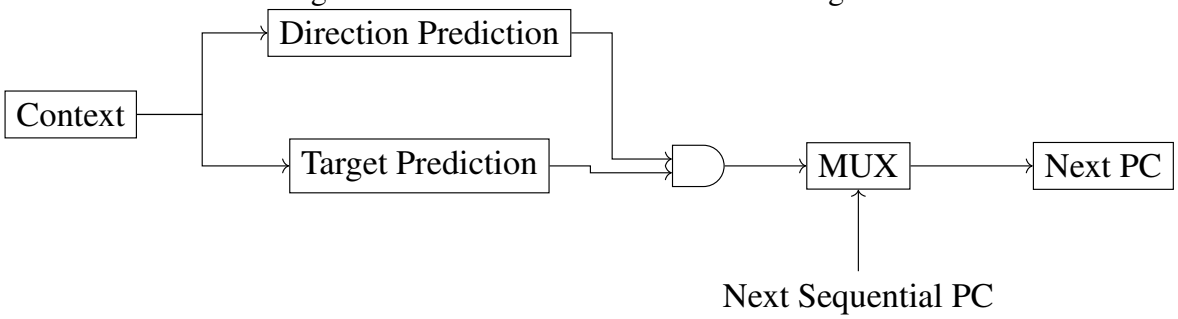
Table 2.2: Example of Context-Based Speculation Using Previous PCs and Branch Outcomes

Cycle	**N-2**			**N-1**			**N**		
	PC	Branch	PC_{n+1}	PC	Branch	PC_{n+1}	PC	Branch	PC_{n+1}
1	0	0	4	4	1	100	100	0	104
2	4	1	100	100	0	104	104	0	108
3	100	0	104	104	0	108	108	1	200
4	104	0	108	108	1	200	200	0	204

The way $Context_N$ is stored and updated varies depending on the prediction algorithm. It will be examined how different predictors (e.g., Bimodal, Gshare) structure and modify their context for improved accuracy in subsequent sections.

Figure 2.1 shows a high-level diagram of a branch predictor consisting of a **Context** used by the **Direction Prediction** and **Target Prediction** to compute a full prediction.

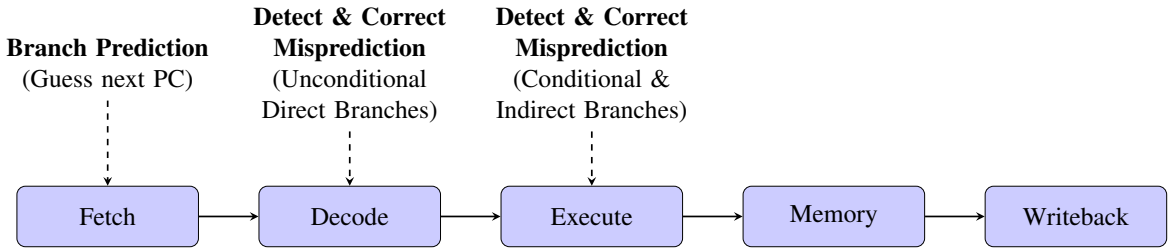
Figure 2.1: Abstract Branch Prediction diagram



2.2 Recovery

A **misprediction** occurs when a prediction does not match the decoded/executed information. In pipelined processors the branch outcome can only be confirmed in the *Decode* and *Execute* stages, which are found multiple cycles after the *Fetch* stage, where branch prediction occurs. Table 2.1 shows at which stage certain branch type outcomes are computed.

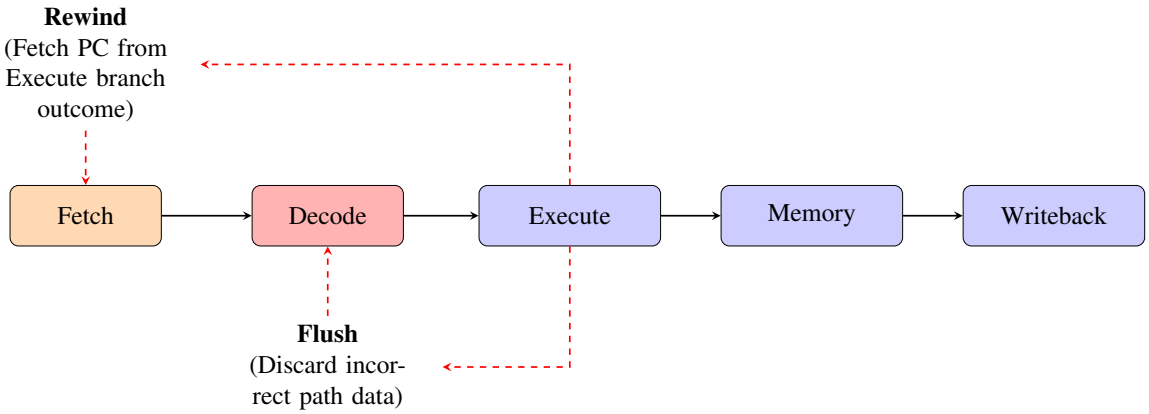
Figure 2.2: 5-Stage processor diagram with branch outcome resolution



Recovering the branch predictor implies the following items:

- **Flush:** Discard any work after the detected misprediction.
- **Rewind:** Correct the misprediction by fetching from the correct path and continue predictions from that point.

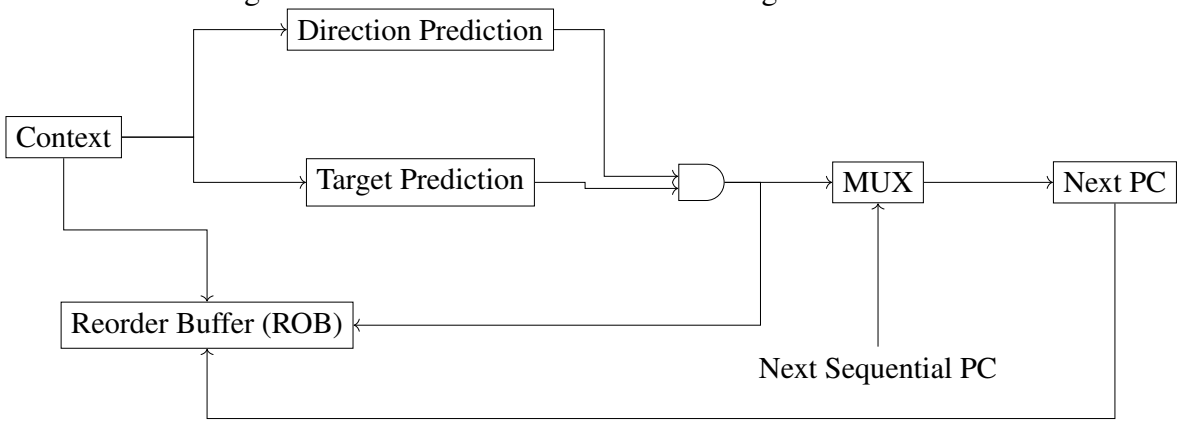
Figure 2.3: 5-Stage processor diagram with Execute misprediction recovery



Multiple predictions can happen between **Speculation** and **Recovery**. With each prediction changing the context thanks to the fact that the current prediction depends on the previous one, there is a need to store all the speculative context values in order to properly **Rewind** the processor to the misprediction point.

For this work the **ROB** (Re-order Buffer) will be leveraged to store the speculative prediction context in each entry, but a standalone *Prediction Buffer* can be implemented to store and keep track of the speculative predictions. Figure 2.4 shows a high-level diagram of a branch predictor using the ROB for speculative context and prediction storage.

Figure 2.4: Abstract Branch Prediction diagram with ROB



2.3 Training

Branch predictors can only predict branches that were seen before. Assuming an out-of-reset, empty predictor, all predictions will be *Not-Taken*. Once a branch is resolved during the *Decode/Execute* stages and the correct program path is identified, the predictor can be trained to correlate a specific context with a branch outcome. After training, the predictor will attempt to predict the branch anytime the same context is detected.

- **When** training occurs can be classified in two categories:
  - **Speculative:** Before a branch instruction is retired (i.e. During *Decode/Execute* stages)
  - **Non-Speculative:** After a branch instruction is retired (i.e. During *Writeback* stage)
- **Why** training occurs can be classified in two categories:
  - **Correction:** A misprediction needs to be corrected.
  - **Reinforcement:** A correct prediction needs to be enhanced or kept in the predictor since its consider useful.
- **What** information is trained on the predictor heavily depends on the prediction algorithm and if it's a correction or reinforcement.

The **speculation and context**, **recovery** and **training** mechanisms are the foundation of branch prediction and are required by all the algorithms, with each one implementing specific functions inside the mechanisms.

2.4 Direction Prediction

The strategies to determine whether a branch is taken or not can be classified in two ways[11]:

- **Static Branch Prediction**: Using compile-time analysis or simple heuristics (e.g., backwards taken/forwards not-taken) to predict branch behavior.
- **Dynamic Branch Prediction**: Making predictions based on the runtime behavior of branches.

This work will focus on **Dynamic Branch Prediction**.

2.4.1 History-Based Prediction

A fundamental approach to branch direction prediction is utilizing past behavior to predict future outcomes. This method, known as **history-based prediction**, relies on storing and analyzing previous branch executions to identify patterns and improve prediction accuracy.

History-based predictors are an essential part of modern speculative execution strategies. As previously introduced, speculation allows the processor to continue executing instructions beyond control dependencies, assuming the outcome of unresolved branches. The **context** of previous branches, either from a single branch (local history) or multiple branches (global history), provides information that can enhance the accuracy of predictions.

Global vs. Local History

History-based predictors generally fall into two categories:

- **Global History Prediction:** Uses a *Global History Register* (GHR) that tracks the outcomes of multiple recent branches to predict the direction of the current branch.
- **Local History Prediction:** Uses *Per-Branch History Registers* that track the past behavior of a specific branch instruction, allowing more specialized predictions based on its unique execution pattern.

History Register

This structure can be used in both global and local history schemes by tracking if a branch was taken or not with a one-bit encoding. Global history tracks all branch outcomes (hence the global name), while local history tracks the outcome on a per-branch basis. Figure 2.5 illustrates how a history register can be mapped to a 5-bit shift register while table 2.3 illustrates how a history register is updated.

Figure 2.5: History Register as a Shift Register

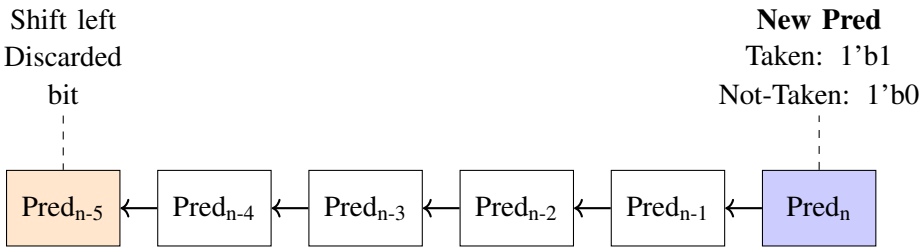


Table 2.3: Example of History Register Update

Cycle	HR_N	Branch Prediction	HR_{N+1}
1	10101	Taken	01011
2	01011	Not Taken	10110
3	10110	Taken	01101

The history register can be used to recognize patterns. Following example on table 2.3, the 5-bit register can be used to correlate 16 different taken/not-taken patterns to a taken/not-taken outcome.

Figures 2.6 and 2.7 show a simplified diagram of global and local history respectively.

Figure 2.6: Representation of Global History Register

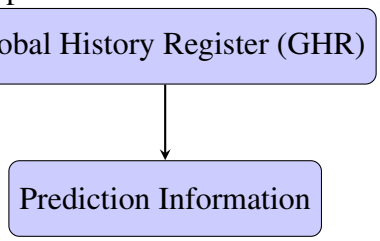
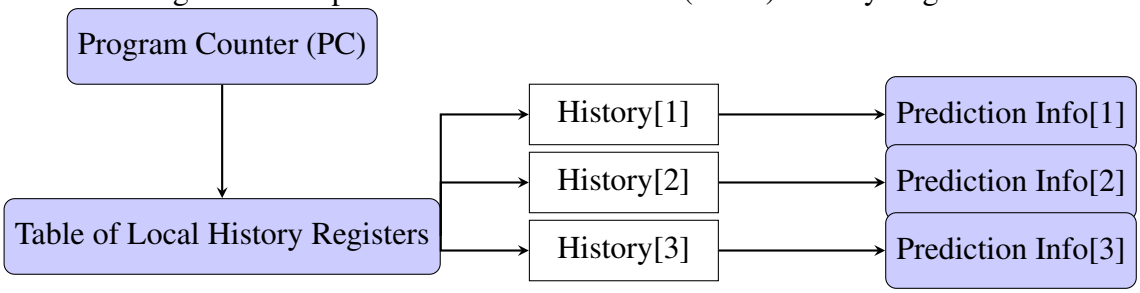


Figure 2.7: Representation of Per-Branch (Local) History Register



History-Based Prediction is a method to encode the **context** for prediction purposes. Each prediction algorithm specifically dictates what information affects the context and how it is used in the **speculation, recovery** and **training** mechanisms of the predictor.

2.5 Target Prediction

Any taken branch requires a **taken target** which indicates the next PC address to fetch from. The relationship of the address and the branch can be classified in two categories:

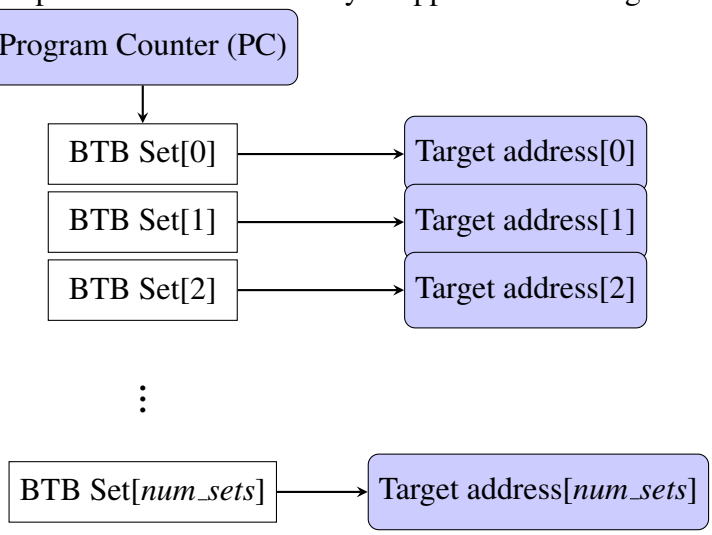
- **Direct:** Only one possible taken target address value, resolved at decode by computing $PC + offset$.
- **Indirect:** More than one possible taken target address value, resolved at execute time.

All **Direct** branch targets are static and can be predicted with a simple cache mechanism while **Indirect** branches could require complex algorithms to correctly predict the taken target value.

2.5.1 Branch Target Buffer (BTB)

Target prediction follows the same **speculation and context** principles. In order to predict a previously-taken branch its taken target must be stored in a structure to be later accessed by the context. The *BTB* implements a cache storing the target address of previously executed branch instructions[12]. Whenever a taken branch is detected by the direction predictor, the *BTB* is also accessed to provide the speculative taken target address. Figure 2.8 shows the diagram of a Directly-Mapped BTB, which is what this work will use for target prediction.

Figure 2.8: Representation of a Directly-Mapped Branch Target Buffer (BTB)



Since a PC address value can use up to 64 bits, it is not always possible for a *BTB* to store a target address for all possible PC values ($BTB_{sets} = 2^{64}$). A subset of the PC can be used to compute the BTB index address with bit slicing. This allows the structure to be physically implementable but introduces aliasing, where different PCs might use the same *BTB* set.

$$Index_{BTB} = PC[end_{bit} : start_{bit}]$$

Chapter 3

Algorithm modeling and hardware implementation

This chapter will present different branch prediction algorithms with their respective models based on the C++ CBP framework[7] and their hardware implementation in SystemVerilog. Key concepts for both C++ and SystemVerilog implementations are presented first and then expanded on later sections that will review the algorithm implications in the **speculation and context**, **recovery** and **training** mechanisms.

3.1 C++ CBP Framework concepts

The C++ CBP framework requires a `PREDICTOR` class that implements 2 methods shown in Listing 1. These functions can be used to implement any algorithm and the class can define its own data structures. The simulator will call these functions and assumes all context handling, prediction selection and training logic is implemented inside these methods. It is important to note that this framework is solely focused on **Direction Prediction**.

```
1 class PREDICTOR {
2 public:
3     // Main Prediction function
4     bool GetPrediction(UINT64 PC);
5
6     // Retire-Time Update
7     void UpdatePredictor(UINT64 PC, OpType OPTYPE,bool resolveDir,
      ↪  bool predDir, UINT64 branchTarget);}
```

Listing 1: C++ Predictor class methods

The `GetPrediction` relates to the **speculation and context** mechanism. This function provides the current `PC` as an input and returns a `bool` that indicates if the current `PC` is a taken branch or not. Listing 2 shows a basic, Always-Not-taken prediction function.

```cpp
// Main Prediction function
bool PREDICTOR::GetPrediction(UINT64 PC) { return false };
```

Listing 2: C++ always-not-taken GetPrediction method

The `UpdatePredictor` method relates to both **recovery** and **training** mechanisms. This function is called after every prediction and provides the resolved branch information which can be used for both mechanisms. Thanks to this, is it not required to model a buffer of predictions but proper recovery of speculative context might still be required depending on the algorithm, even when just one prediction was made. The following arguments are passed to this method:

- `UINT64 PC`: Program Counter address of the prediction.
- `OpType OPTYPE`: Branch type, can also indicate if the predicted `PC` was not a branch. Listing 3 shows all the different values.
- **`bool`** `resolveDir`: Resolved branch direction.
- **`bool`** `predDir`: Predicted branch direction.
- `UINT64 branchTarget`: Resolved branch target.

`resolveDir` and `predDir` can be compared to determine if a prediction was correct or not, while the remaining values can be used for context improvements or other changes depending on the algorithm.

```cpp
typedef enum {
  OPTYPE_OP                =2,

  OPTYPE_RET_UNCOND,
  OPTYPE_JMP_DIRECT_UNCOND,
  OPTYPE_JMP_INDIRECT_UNCOND,
  OPTYPE_CALL_DIRECT_UNCOND,
  OPTYPE_CALL_INDIRECT_UNCOND,

  OPTYPE_RET_COND,
  OPTYPE_JMP_DIRECT_COND,
  OPTYPE_JMP_INDIRECT_COND,
  OPTYPE_CALL_DIRECT_COND,
  OPTYPE_CALL_INDIRECT_COND,

  OPTYPE_ERROR,

  OPTYPE_MAX
}OpType;
```

Listing 3: C++ OpType enum definition

3.2 SystemVerilog implementation concepts

A common *BPU* interface shown in Listing 4 is defined which will be used for all the presented algorithms. Similar to the *PREDICTOR* class, the hardware BPU has the current *PC* as an input and produces a *branch_prediction_packet* as an output.

```
module BPU
(
    input   logic             clk,
    input   logic             rst,

    input   RV32I_OPERAND_t   current_program_counter,

    // ROB Interface
    input   logic             ROB_retire,
    input   logic             ROB_JE_CLEAR,
    input   BRANCH_PACKET_t   ROB_branch_execution_packet,

    // Output Prediction
    output  BRANCH_PACKET_t   branch_prediction_packet
);
```

Listing 4: SV BPU basic interface

Since this is a predictor on a pipelined *CPU*, the *BPU* will always produce a prediction on an active clock. The contents of the prediction will be affected depending on the inputs described in the following list:

- `rst`: Reset signal, once present the *PC* will be forced to the reset address.
- `current_program_counter`: Current *PC* address.
- `ROB_retire`: Signal from Reorder-Buffer (ROB) that indicates an instruction has been fully executed, and will be retired.
- `ROB_JE_CLEAR`: Signal from Reorder-Buffer (ROB) that indicates a Jump Execution (JE) clear. Whenever the BPU mispredicts this signal goes high after executing the mispredicting instruction and having the correct branch data.
- `ROB_branch_execution_packet`: ROB branch packet with resolved information from decode and execute stages. It provides the true program path and is used for the **recovery** and **train** mechanisms.
- `branch_prediction_packet`: *BPU* branch packet with speculative branch information.

The `BRANCH_PACKET_t` contains all the branch information and is defined in Listing 5. This data structure can include more data depending on the algorithm being implemented.

```
typedef struct packed {
    BRANCH_TYPE_t    branch_type;
    logic            is_branch;
    logic            is_taken;
    RV32I_OPERAND_t  target;
    RV32I_OPERAND_t  program_counter;
} BRANCH_PACKET_t;

typedef enum logic[1:0]{
    NO_BR                = 2'd0,
    UNCONDITIONAL        = 2'd1,
    CONDITIONAL_TAKEN    = 2'd2,
    CONDITIONAL_NTAKEN   = 2'd3
} BRANCH_TYPE_t;
```

Listing 5: SV `BRANCH_PACKET_t` and `BRANCH_TYPE_t` basic definition

For this work all algorithms will perform the **speculation and context**, **recovery** and **training** in the same stages as shown in Figure 3.1. Branches are resolved at execute and trained in the same stage if a mispredict is detected while correct branches can be trained once they're retired from the *ROB*.

Figure 3.1: 5-Stage processor diagram with BPU mechanisms

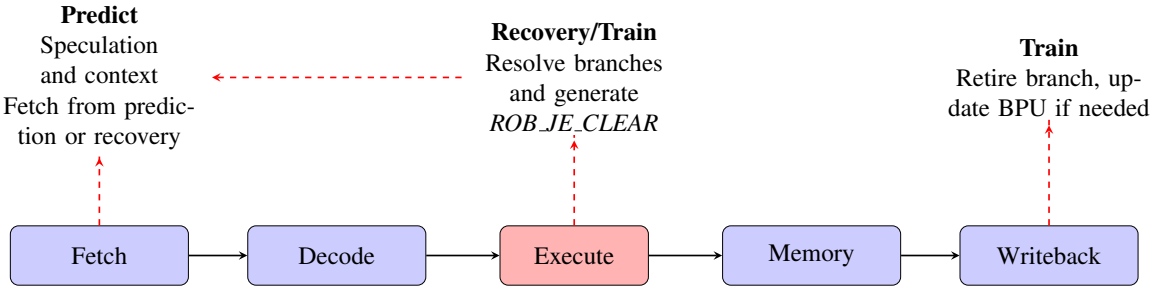
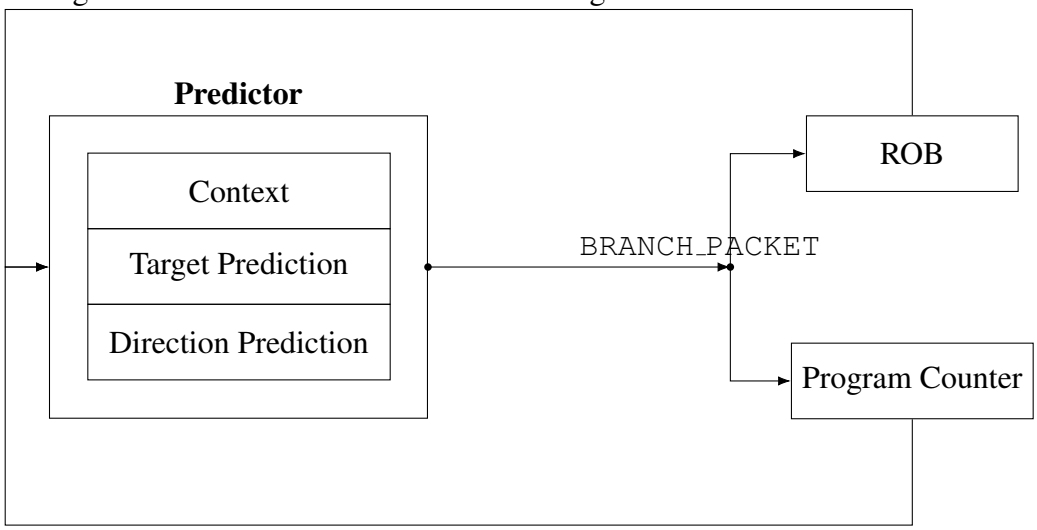


Figure 3.2 shows a block diagram of the *PREDICTOR* containing its components and passing a `BRANCH_PACKET_t` to the *PC* and the *ROB*.

Figure 3.2: Abstract Branch Prediction diagram with BRANCH_PACKET



3.2.1 Target prediction and BTB

Since this work will focus on conditional prediction algorithms, a direct-mapping BTB strategy will be implemented as the target prediction mechanism for all examples. The BTB assumes an array with each set holding the data structures defined in Listing 6, this code snippet also shows the BTB sizing define-based parameters and read address datatype *BTB_IDX_t*. By using parameters and defines the size of the BTB can be easily adjusted without major code changes. Two structures are implemented: A **Conditional BTB** (CBTB) and an **Unconditional BTB** (UBTB) that store Conditional and Unconditional branch targets respectively.

```
`define BPU_BTB_IDX_NUM_BITS 10

parameter BPU_BTB_NUM_SETS = 2 ** `BPU_BTB_IDX_NUM_BITS;

// 2-bit offset due to fixed-size RISC-V ISA
parameter BPU_BTB_IDX_MSB = (`BPU_BTB_IDX_NUM_BITS + 2) - 1;
parameter BPU_BTB_IDX_LSB = 2;

typedef logic [`BPU_BTB_IDX_NUM_BITS-1:0] BTB_IDX_t;

typedef struct packed {
    RV32I_OPERAND_t target;
    logic valid;
} CBTB_ENTRY_t;

typedef struct packed {
    RV32I_OPERAND_t target;
    logic valid;
} UBTB_ENTRY_t;

```

Listing 6: SV BTB Parameters and data structures

Both BTBs can be implemented with an almost identical interface and a *generate* statement instantiating flip-flops. Listings 7 and 8 show the interface and array generation code examples for the CBTB respectively.

```
module cbtb
(
    input   logic           clk,
    input   logic           rst,

    input   logic           ROB_JE_CLEAR,
    input   logic           ROB_retire,
    input   BRANCH_PACKET_t ROB_branch_execution_packet,

    input   BTB_IDX_t       btb_read_idx,
    output  CBTB_ENTRY_t    cbtb_predict_read_data
);
```

Listing 7: SV CBTB Interface

```
CBTB_ENTRY_t [BPU_BTB_NUM_SETS-1:0] cbtb_sets;
CBTB_ENTRY_t                        cbtb_write_data;
BTB_IDX_t                           cbtb_write_idx;
logic                               cbtb_write_en

generate
    genvar i;
    for (i=0; i<BPU_BTB_NUM_SETS; i++) begin : cbtb_flops
        `FF_D_RST_EN(
            clk,
            rst,
            (cbtb_write_en && (i == btb_write_idx)),
            cbtb_write_data,
            cbtb_sets[i]
        )
    end : cbtb_flops
endgenerate
```

Listing 8: SV CBTB Flip-Flop array generation

Speculation and context

For prediction, both *CBTB* and *UBTB* will use a subset of the PC as the indexing function based on the parameters defined in Listing 6. This address is used as the read address for the BTB read ports. The *btb_hit* signals are used to determine if there's useful data read from the BTBs by checking the *valid* bit of the data structure as shown in Listing 9. It is important to note that since BTBs are only using the PC as context, it is not necessary to implement context handling logic.

```
BTB_IDX_t     btb_read_idx;
UBTB_ENTRY_t  ubtb_read_data;
CBTB_ENTRY_t  cbtb_read_data;
logic         cbtb_hit;
logic         ubtb_hit;

assign btb_read_idx   =
 ↪  current_program_counter[BPU_BTB_IDX_MSB:BPU_BTB_IDX_LSB];
assign cbtb_read_data = cbtb_sets[btb_read_idx];
assign ubtb_read_data = ubtb_sets[btb_read_idx];

assign cbtb_hit       = cbtb_read_data.valid;
assign ubtb_hit       = ubtb_read_data.valid;
```

Listing 9: SV BTB Speculation and Context

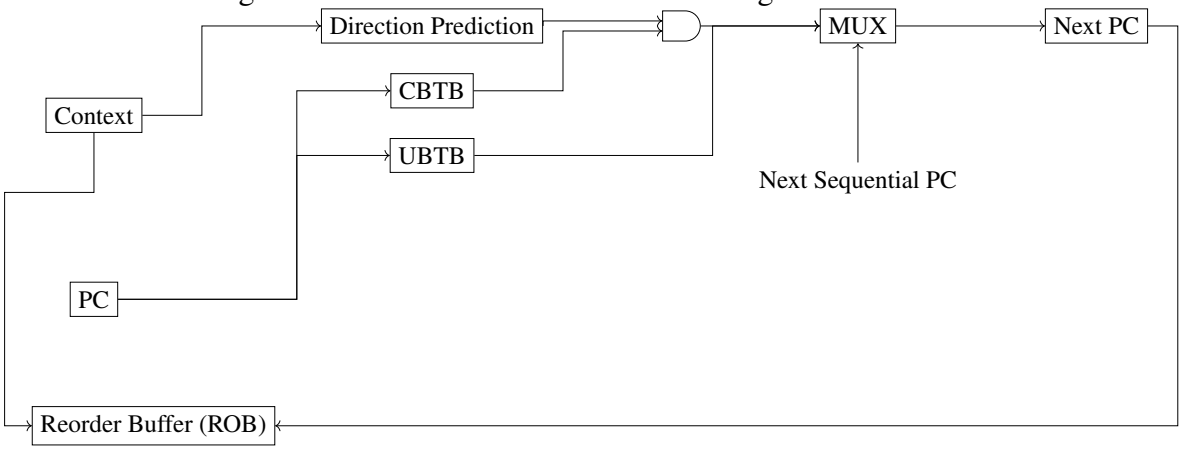
Recovery

Due to a lack of speculative changes in the BTB data, no extra logic is needed in the BTB to handle recoveries.

Training

For this work training is triggered during a `ROB_JE_CLEAR` misprediction to store the correct target address in the corresponding BTB set. Since a simple training prediction algorithm is used, it is not needed to update any other data. An example of the CBTB training write data and write enable logic is shown in Listing 10 and 11 respectively. Figure 3.3 shows the baseline predictor with BTBs.

Figure 3.3: Baseline Branch Prediction diagram with BTBs



```
// Generate btb_write_idx
assign training_program_counter =
 ↪  branch_execution_packet.program_counter;
assign btb_write_idx            =
 ↪  training_program_counter[BPU_BTB_IDX_MSB:BPU_BTB_IDX_LSB];
// Get training read_data
assign cbtb_training_read_data  = cbtb_sets[btb_write_idx];

assign rob_target = ROB_branch_execution_packet.target;

assign cbtb_diff_target =  cbtb_training_read_data.valid
                         & (   cbtb_training_read_data.target
                            != rob_target);

// Write data
always_comb
  case (branch_execution_packet.branch_type)
    CONDITIONAL_TAKEN: begin
      if (ROB_JE_CLEAR) begin
        if (~cbtb_training_read_data.valid) begin
          // Missed CONDITIONAL not present in CBTB
          cbtb_write_data.valid  = '1;
          cbtb_write_data.target = rob_target;
        end else begin
          // Incorrect target CONDITIONAL
          cbtb_write_data.valid  =  cbtb_diff_target;
          if (cbtb_diff_target) begin
            cbtb_write_data.valid = '1;
            cbtb_write_data.target = rob_target;
          end else begin
            cbtb_write_data = '0;
        end
      end else
        cbtb_write_data = '0;
    end

    // Always zeroes for other cases
    default:
      cbtb_write_data = '0;
  endcase
```

Listing 10: SV CBTB Training Write Data logic

```
// Generate btb_write_idx
assign training_program_counter =
  ↪ branch_execution_packet.program_counter;
assign btb_write_idx            =
  ↪ training_program_counter[BPU_BTB_IDX_MSB:BPU_BTB_IDX_LSB];
// Get training read_data
assign cbtb_training_read_data  = cbtb_sets[btb_write_idx];

assign cbtb_different_target =   cbtb_training_read_data.valid
                               & (cbtb_training_read_data.target
                                  !=
                                   ↪ ROB_branch_execution_packet.target);

// Write Enable
always_comb
  case (branch_execution_packet.branch_type)
    // Missed CONDITIONAL not present in CBTB
    // or incorrect target CONDITIONAL
    CONDITIONAL_TAKEN:
      cbtb_write_en  =  ROB_JE_CLEAR
                      & (  cbtb_different_target
                         | ~cbtb_training_read_data.valid);

    // BOGUS branch case, aliasing/branch that's not really there
    NO_BR:
      cbtb_write_en =  ROB_JE_CLEAR
                     & cbtb_training_read_data.valid;

    default:
      cbtb_write_en = '0;
  endcase

```

Listing 11: SV CBTB Training Write Enable logic

3.3 *N*-bits counter

This algorithm implies a history-based approach where historical information on the direction that a branch takes is used to predict its future direction. An *N*-bits counter is used to encode the direction taken by the last *n* executions of a branch. The prediction is generated depending on the value/state of the history. The algorithm can be represented with an FSM as shown in Figure 3.4 and Figure 3.5 for 1-bit and 2-bit counters respectively. This counter is only updated once the branch outcome is known, which means no speculative changes are made and the taken/not-taken information comes from execution or retirement exclusively. This method follows one of the earliest predictive strategies proposed by Smith[9] where saturating counters are used to predict the direction of branches.

Figure 3.4: FSM for 1-bit counter

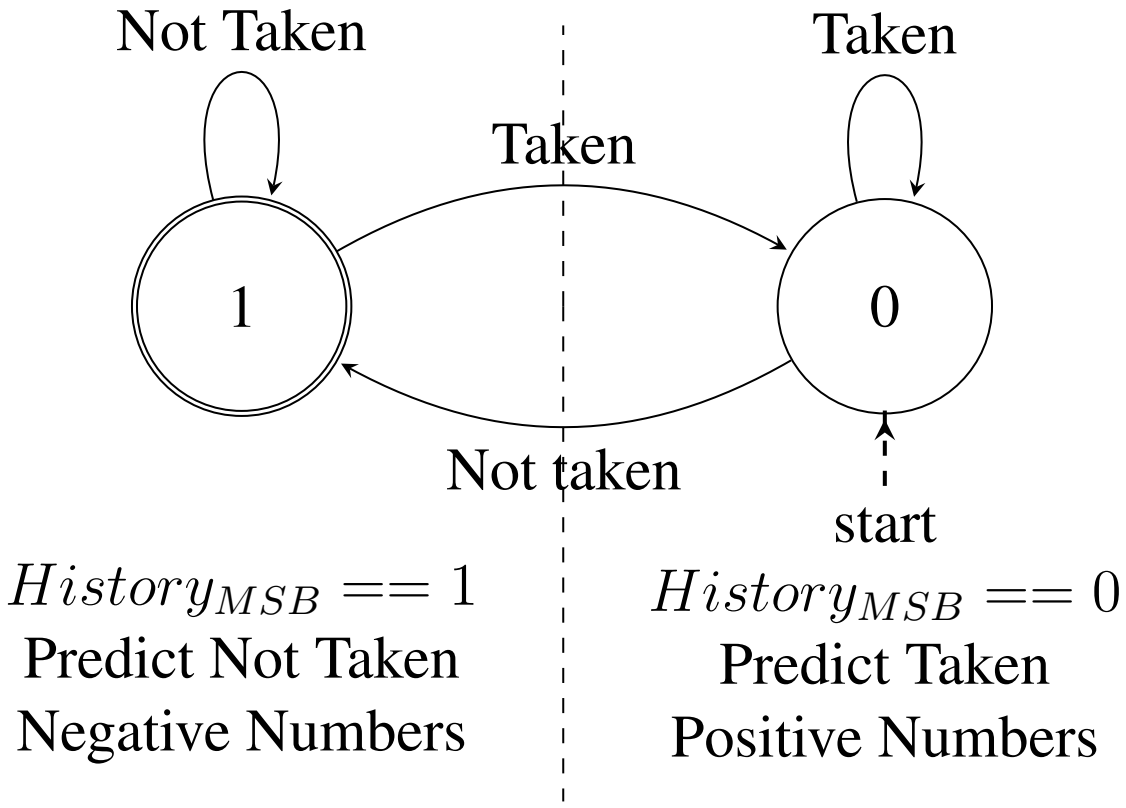
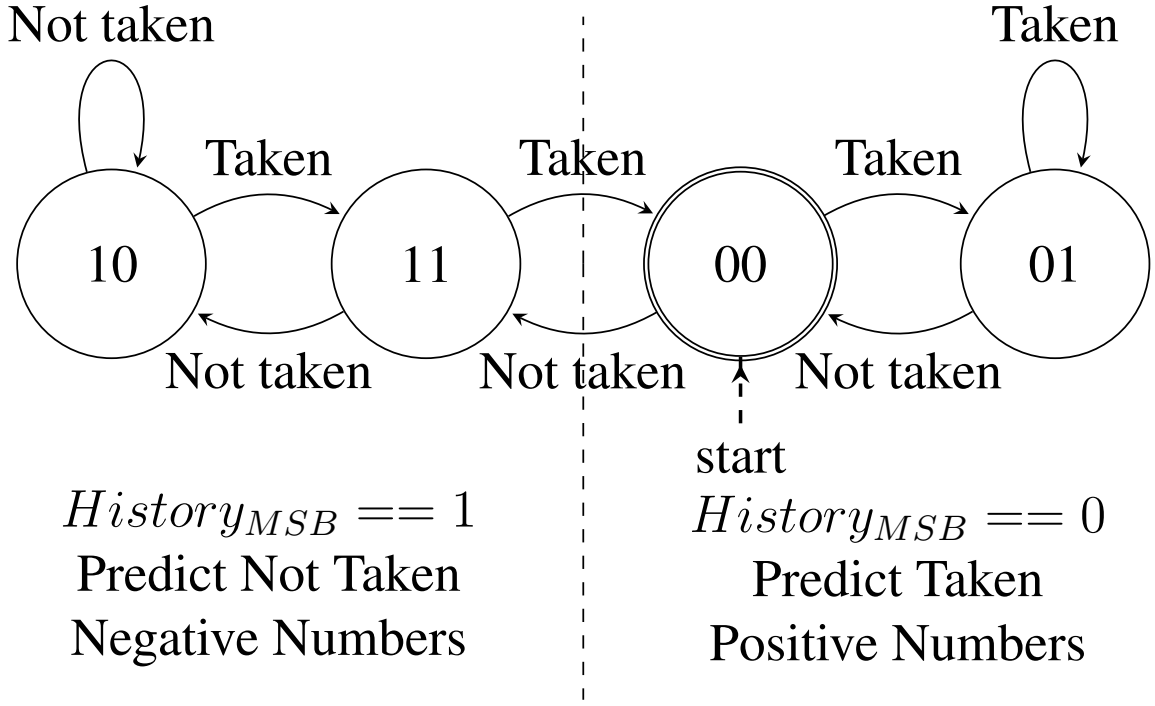


Figure 3.5: FSM for 2-bit counter



The following considerations are made for the implementation of the *N*-bit counter algorithm:

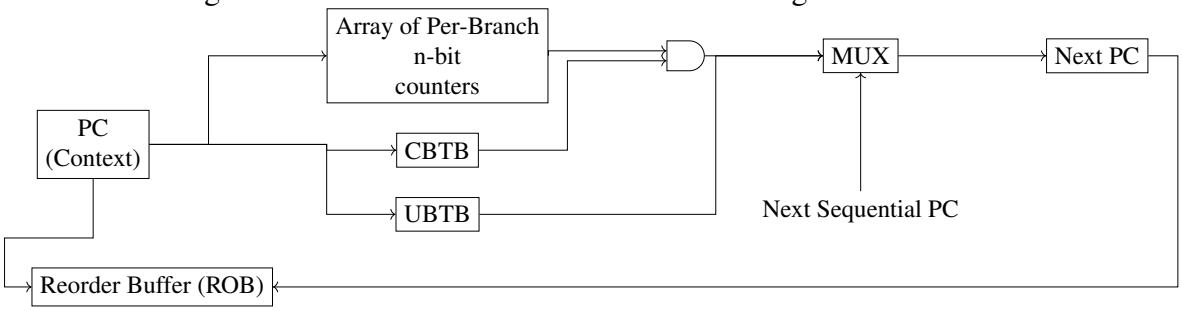
- Taken direction is predicted if the counter's MSB is set to 0 (positive numbers).
- Not-taken direction is predicted if the counter's MSB is set to 1 (negative numbers).
- The counter is incremented if the branch is taken.
- The counter is updated in a saturating manner following the next equation:

$$-2^{n-1} \leq counter \leq (2^{n-1} - 1), \quad \text{where } n \text{ is the number of bits in the counter}$$

- The counter is decremented if the branch is not-taken.
- The maximum value of the counter is set to $2^{n-1} - 1$
- The minimum value of the counter is set to -2^{n-1}.

Figure 3.6 shows a diagram of a BPU with the algorithm elements, it can be observed that the *Direction Predictor* is now replaced by the *n*-bit counters array and its information is combined with the *CBTB* in order to produce a complete prediction.

Figure 3.6: N-bit Counter Branch Prediction diagram with BTBs



3.3.1 *N*-bit counters array

It is necessary to implement a memory structure that stores the different per-branch counters that this algorithm uses. Similar to the *BTBs* a direct-mapping strategy is used in this work where a subset of the *PC* is used as the index. Listings 12 and 13 show the SystemVerilog and C++ array declarations respectively.

```
`define BPU_COUNTER_IDX_NUM_BITS 10
`define BPU_COUNTER_NUM_BITS 2
`define BPU_COUNTER_NUM_SETS 2 ** `BPU_COUNTER_IDX_NUM_BITS

// No valid bit, use valid from CBTB entry instead
typedef logic [BPU_COUNTER_NUM_BITS-1:0] COUNTER_t;
typedef logic [BPU_COUNTER_IDX_NUM_BITS-1:0] COUNTER_IDX_t;
COUNTER_t sets [`BPU_COUNTER_NUM_SETS];

generate
  genvar i;
  for (i=0; i<BPU_COUNTER_NUM_SETS; i++) begin : counters
    `FF_D_RST_EN(
      clk,
      rst,
      (write_enable && (i == idx)),
      write_data,
      sets[i])
  end : counters
endgenerate
```

Listing 12: SV *N*-bits array declarations

```cpp
#define COUNTER_ARRAY_IDX_NUM_BITS 10
#define COUNTER_NUM_BITS 2

// History Register class definition
class NbitsCounter {
  public:
  int32_t max_value = (1 << (COUNTER_NUM_BITS-1)) - 1;
  int32_t min_value = -(1 << (COUNTER_NUM_BITS-1));
  int32_t weak_taken = 0;
  int32_t weak_not_taken = -1;

  int32_t counter;
  bool valid;

  // Constructor/Initialization
  NbitsCounter() {
    counter = weak_taken;
    valid = false;
  }
}

class PREDICTOR {
  private:
    vector<NbitsCounter> counter_array;

  // Constructor/Initialization
  PREDICTOR(void) {
  counter_array.resize(1 << COUNTER_ARRAY_IDX_NUM_BITS);

  for (auto& counter : counter_array)
    counter = NbitsCounter();
  }
}
```

Listing 13: C++ *N*-bits array declarations

3.3.2 Speculation and context

Similar to the *BTB* in this algorithm the context is composed of a subset of the *PC* which is used to index the counters array. Each unique index will point to a specific counter which will be used to compute the prediction function which can be simplified to simply check the MSB of the counter as shown in the FSM diagrams (Listings 3.4 and 3.5). This structure is similar to the two-level adaptive predictor model described by Yeh and Patt [13]. In this case a Program Counter is used to index a pattern table that stores directional information. An abstract formula for the prediction function:

$$Index = PC[MSB : LSB]$$

$$Counter = CounterArray[Index]$$

$$Taken = CBTB_{valid}\&!Counter_{MSB}$$

Since the CBP Framework has a focus on prediction direction it is not required to implement a BTB, which is why the `NbitsCounter` implements a valid bit. The SystemVerilog implementation leverages the *CBTB* valid bit. Despite the difference both implementations of valid bits aim to prevent the counter to be used on no-branch predictions. The code snippets on Listing 14 and 15 show the prediction logic for C++ and SystemVerilog respectively.

```
bool NbitsCounter::GetPrediction() {
  bool taken = (counter >= weak_taken);
  if (valid && taken)
    return true;
  else
    return false;
}

uint64_t PREDICTOR::GetCounterArrayIDX(UINT64 pc) {
    uint64_t mask = (1 <<  COUNTER_ARRAY_IDX_NUM_BITS) - 1;
    return pc & mask;
}

bool PREDICTOR::GetPrediction(UINT64 PC) {
  auto counter_idx = PREDICTOR::GetCounterArrayIDX(PC);
  auto& counter = counter_array[counter_idx];

  return counter.GetPrediction();
}
```

Listing 14: C++ *N*-bits Speculation functions

```
COUNTER_IDX_t read_idx;
assign read_idx = current_program_counter[MSB:LSB];

COUNTER_t counter_read_data;
assign counter_read_data = counters[read_idx];

CBTB_ENTRY_t cbtb_read_data;

logic taken;
BRANCH_PACKET_t branch_packet;

always_comb begin
  branch_packet = '0;
  // Check for positive number with MSB==0 and CBTB valid
  taken =  cbtb_read_data.valid
         & !counter_read_data[BPU_COUNTER_NUM_BITS-1];

  if (taken) begin
    branch_packet.branch_type = CONDITIONAL_TAKEN;
    branch_packet.is_branch = '1;
    branch_packet.is_taken = '1;
    branch_packet.target = cbtb_read_data.target;
  end else begin
    branch_packet.target = next_sequential_pc;
  end
end
```

Listing 15: SV *N*-bits Speculation logic

3.3.3 Recovery

Due to the lack of speculative context and simplicity of context information this algorithm doesn't require any additional recovery mechanisms besides the general flush and resteer.

3.3.4 Training

Thanks to the counter-based approach, the training of this algorithm is done by simply updating the counter value based on the outcome of the branch. The counter is incremented or decremented depending on whether the branch was taken or not. This is done in a saturating manner, meaning that if the counter reaches its maximum or minimum value, it will not be incremented or decremented further. This allows for a more stable prediction over time, as the counter will not oscillate between values too quickly. Listings 16 and 17 show the training logic for C++ and SystemVerilog respectively.

```
1
2
3  void PREDICTOR::Update(UINT64 idx, OpType optype, bool resolveDir)
   ↪  {
4    auto& counter = counter_array[idx];
5
6    switch (optype) {
7      case OPTYPE_RET_COND:
8      case OPTYPE_JMP_DIRECT_COND:
9      case OPTYPE_JMP_INDIRECT_COND:
10     case OPTYPE_CALL_DIRECT_COND:
11     case OPTYPE_CALL_INDIRECT_COND:
12       counter.Update(resolveDir);
13       break;
14
15     // Invalidate counter if valid but branch was not found
16     default:
17       if (counter.valid)
18           counter.valid = false;
19       break;
20   }
21 }
22
23 void NbitsCounter::Update(bool resolved_direction) {
24   bool allocate = !valid & resolved_direction;
25
26   // Initialize counter and mark as valid
27   if (allocate) {
28       valid = true;
29       return;
30   }
31
32   // Update counter based on the resolved direction
33   if (valid) {
34       if (resolved_direction)
35         counter++;
36       else
37         counter--;
38       counter = std::clamp(counter, min_value, max_value);
39   }
40 }
```

Listing 16: C++ *N*-bits Training logic

```
assign counter_saturated  =  (counter_read_data == MIN_VALUE)
                           | (counter_read_data == MAX_VALUE);

// Write enable logic
always_comb
  case (branch_execution_packet.branch_type)
    CONDITIONAL_TAKEN, CONDITIONAL_NTAKEN:
      write_en =  cbtb_read_data.valid
                & (  ROB_JE_CLEAR
                   | ROB_retire);

    // BOGUS branch case, aliasing/branch that's not really there
    NO_BR:
      write_en =  ROB_JE_CLEAR
                & cbtb_read_data.valid

    default: write_en = '0;
  endcase

// Write data
always_comb
  case (branch_execution_packet.branch_type)
    CONDITIONAL_TAKEN, CONDITIONAL_NTAKEN: begin
      write_data = counter_read_data;
      if (ROB_JE_CLEAR & cbtb_read_data.valid)
        write_data =  training_is_taken
                    ? BPU_COUNTER_NUM_BITS'(counter_read_data +
                    ↪  1'b1)
                    : BPU_COUNTER_NUM_BITS'(counter_read_data -
                    ↪  1'b1);
      else
        write_data =  (~counter_saturated & cbtb_read_data.valid)
                    ? (  training_is_taken
                       ? BPU_COUNTER_NUM_BITS'(counter_read_data +
                       ↪  1'b1)
                       : BPU_COUNTER_NUM_BITS'(counter_read_data -
                       ↪  1'b1))
                    : counter_read_data;
      end

    // Always zeroes for other cases
    default: write_data = '0;
  endcase
```

Listing 17: SV *N*-bits Training logic

3.3.5 C++ CBP 2016 MPKI Simulation results

The simulation results for the C++ implementation of the *N*-bits counter algorithm with the CBP 2016 Framework are shown in Table 3.1. The results show that the algorithm achieves a significant improvement in prediction accuracy compared to the always-not-taken, no branch-prediction baseline even with a reduced number of sets. The metric used to measure the performance of the algorithm is the number of mispredictions per 1000 (kilo) instructions (MPKI). A considerable improvement is observed even with a 1-bit counter, which is expected to be less accurate than the 2-bit counter. The results also show that the algorithm is able to achieve a low MPKI even with a small number of bits in the counter (2). This indicates that the algorithm is able to effectively learn the behavior of branches over time and make accurate predictions based on this information. Improvement diminishes or even regresses as the number of bits in the counter increases. The following points are important to highlight:

- CBP 2016 framework only simulates conditional branches and does not include target prediction. The CBTB size values are only representative and were not used in the simulation but are still included for completeness due to its relevance in an actual implementation.
- Both training and testing workloads are used, which are a mix of mobile and server with short and long variations.
- The No BPU configuration is used as a baseline for the MPKI results, which is a configuration with no branch prediction (always not-taken).
- MPKI results represent the arithmetic mean of the workloads in the simulation.
- Table 3.1 uses a 7-bit index for counters array in order to align with synthesis results which are resource constrained.

Table 3.1: C++ CBP 2016 MPKI results of 1, 2, 3, 4, and 5 bit counter algorithms with memory requirements using a 7-bit index for the CBTB and n-bit counters array

Algorithm	MPKI	Improvement (%)	Global Counter size (bytes)	CBTB size (bytes)	Total Memory (bytes)
No BPU	53.043	-	-	-	-
1-bit counter	22.856	56.91	16	528	544
2-bit counter	17.819	66.41	32	528	560
3-bit counter	17.068	67.82	48	528	576
4-bit counter	17.230	67.52	64	528	592
5-bit counter	17.625	66.77	80	528	608

3.3.6 SystemVerilog FPGA Synthesis results

The synthesis results for the SystemVerilog implementation of the *N*-bits counter algorithm are shown in Table 3.2. The baseline is a pipelined, super-scalar, 1-issue RV32I architecture

with a Reorder Buffer (ROB) and no branch prediction. The results show that including a branch predictor causes decrease in Fmax and increase in resource usage compared to the baseline. The results also show that the algorithm is able to achieve a low Fmax even with a small number of bits in the counter (2). The following points are important to highlight:

- The FPGA synthesis was done using the Intel Quartus Prime software with a MAX 10 FPGA.
- The synthesis results show a significant increase in resource usage and a decrease in Fmax as the number of bits in the counter increases.
- The $Fmax$ is significantly lower than the baseline, which indicates that the branch predictor is more complex and requires more resources to implement.
- Table 3.2 uses a 7-bit index for counters array due to the resource constraints of the FPGA. The index size is a trade-off between performance and resource usage.

Table 3.2: Synthesis results for 1, 2, 3, and 4-bit counter algorithms showing $Fmax$ and resource usage using a 7-bit index for the CBTB, UBTB and n-bit counters array

Configuration	$Fmax$ (MHz)	$Fmax$ Change (%)	Logic Elements	Resource Change (%)
No BPU	55.66	0.00	27,676	0.00
1-bit counter	51.92	-6.72	38,040	+37.45
2-bit counter	50.71	-8.89	38,386	+38.70
3-bit counter	50.10	-9.99	38,726	+39.93
4-bit counter	49.83	-10.47	38,947	+40.73

3.3.7 SystemVerilog IPC and MPKI Simulation results

In order to compensate the FPGA resource limitations the synthesis results on this work were done with a 7-bit index for the CBTB, but the simulation results used a 15-bit index and a 2-bit counter. This decision is driven by one of the CBP 2016 storage budgets (8KB)[7]. With these considerations, results aim to be aligned with more realistic implementations and constraints. C++ MPKI results are shown in table 3.3.

Table 3.3: C++ CBP 2016 MPKI results of 2 bit counter algorithm with memory requirements using a 15-bit index for the n-bit counters array

Algorithm	MPKI	Improvement (%)	Global Counter size (bytes)
No BPU	53.043	-	-
2-bit counter	11.982	77.41	8,192

For the SV simulations a combination of workloads were used, the details of these workloads are described in Chapter 4. There are some key metrics to measure the performance of the algorithm:

- Instructions Per Cycle (IPC): This metric measures the number of instructions executed per cycle. A higher IPC indicates better performance.
- MPKI: This metric measures the number of mispredictions per 1000 instructions. A lower MPKI indicates better performance.
- Cycles: This metric measures the number of cycles taken to execute the workload. A lower cycle count indicates better performance.

The RV32I architecture used for the simulation has a theoretical IPC of 1.0, which means that it can execute one instruction per cycle. Due to the speculative nature of a pipelined architecture, this theoretical IPC can be achieved if there are no stalls or hazards in the pipeline. The results of the simulations for the Dhrystone, Qsort, and CoreMark workloads are shown in Tables 3.4, 3.5, and 3.6 respectively. Table 3.7 shows the average improvements across all workloads.

Table 3.4: 2bit counter Dhrystone 500 runs results (15-bit index, 8KB Global Counter Array)

Predictor	MPKI	Improvement	IPC	Improvement	Cycles	Improvement
No BPU	110.40	-	0.43	-	389348	-
2-bit Counter	27.08	75.47%	0.50	16.28%	332527	14.59%

Table 3.5: 2-bit counter Qsort 2048 numbers results (15-bit index, 8KB Global Counter Array)

Predictor	MPKI	Improvement	IPC	Improvement	Cycles	Improvement
No BPU	180.89	-	0.37	-	600824	-
2-bit Counter	50.11	72.30%	0.46	24.32%	488542	18.69%

Table 3.6: 2-bit counter CoreMark single run results (15-bit index, 8KB Global Counter Array)

Predictor	MPKI	Improvement	IPC	Improvement	Cycles	Improvement
No BPU	188.62	-	0.41	-	3429394	-
2-bit Counter	131.28	30.40%	0.44	7.32%	3182450	7.20%

Table 3.7: 2-bit counter arithmetic mean improvements across all workloads (15-bit index, 8KB Global Counter Array, No BPU baseline)

Algorithm	MPKI Improvement	IPC Improvement	Cycles Improvement
2-bit Counter	59.39%	15.97%	13.49%

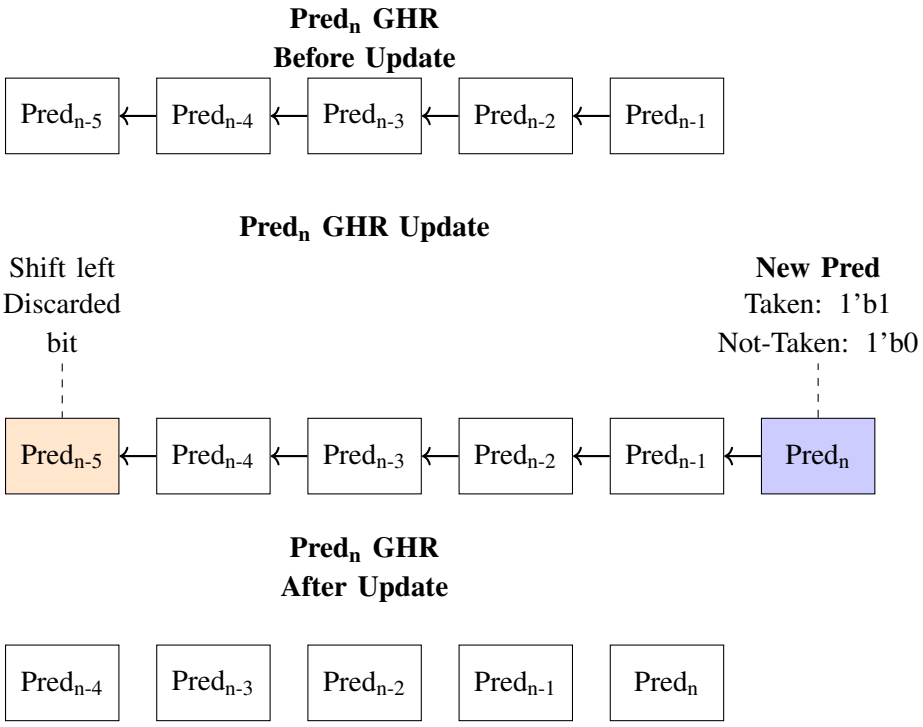
3.3.8 Conclusion

The *N*-bits counter algorithm is a simple and effective branch prediction algorithm that uses a history-based approach to predict the direction of branches. The algorithm achieves significant improvements in prediction accuracy compared to the always-not-taken baseline, even with a reduced number of bits in the counter. The results show that the algorithm is able to effectively learn the behavior of branches over time and make accurate predictions based on this information. C++ MPKI results 3.3 have some similarity to the actual SV MPKI simulated workloads results 3.7 with the differences being expected due to the different workloads used and the inclusion of non-conditional branches in the RTL simulations.

3.4 Global History Register (GHR)

This algorithm uses a global history register (GHR) to keep track of the outcomes of the last N branches. The GHR is a shift register that shifts in the outcome of the most recent branch instruction. The GHR is used to index into a pattern history table (PHT) that contains the prediction information for each possible global history, in this work the PHT is an *N*-bit counter array as described in section 3.3. The theoretical concepts of the GHR are described in the History-Based Prediction section 2.4.1. Figure 2.5 shows an example of a 5-bit GHR shift register with the most recent prediction bit ($Pred_n$) being shifted in from the left. The GHR is used to index into the counters array, which contains the prediction information for each possible global history.

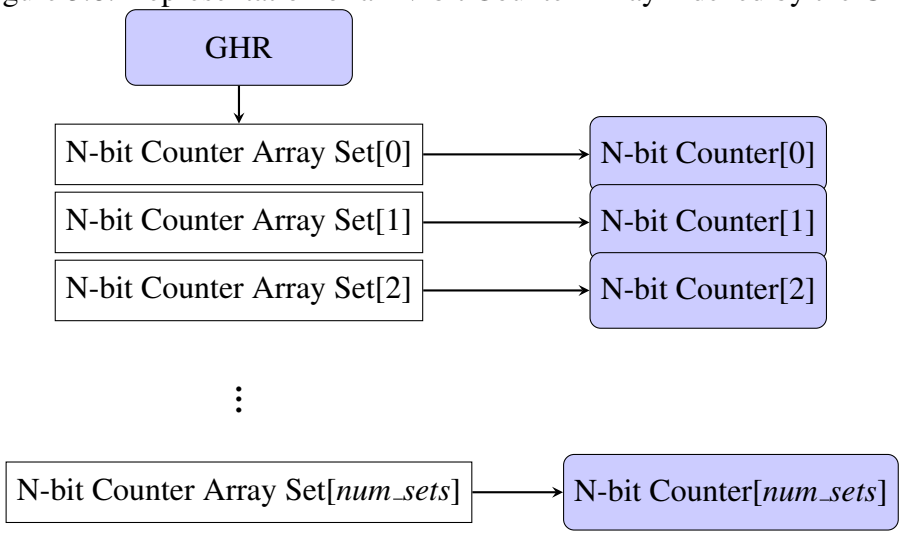
Figure 3.7: 5-bit Global History Register as a Shift Register with update



A diagram showing the GHR indexing the *N*-bit counter array is shown in Figure 3.8.

The GHR is used to index into the counters array, which contains the prediction information for each possible global history. The GHR is updated with the outcome of the most recent branch instruction, and the prediction information is retrieved from the PHT using the GHR as an index.

Figure 3.8: Representation of an *N*-bit Counter Array indexed by the GHR



The following general points are important to note about the GHR algorithm.

- The GHR is a shift register that shifts in the outcome of the most recent branch instruction.
- The complete GHR is used to index into the N-bit counter array that contains the prediction information for each possible global history.
- The GHR is speculatively updated with the outcome of the most recent branch instruction.
- The GHR must be repaired in case of a misprediction, which means that the GHR must be restored to a previous state before the misprediction occurred.

3.4.1 Speculation and Context

As mentioned in the previous section, the GHR is used as the context for the branch direction predictor. Implementation is similar to the *N*-bit counter array, where the GHR is used as the index to the *N*-bit counter array. This structure maps directly to the two-level adaptive predictor model described by Yeh and Patt [13], where a branch history or context is used to index a pattern table that stores directional information. An abstract formula for the prediction function:

$$Counter = CounterArray[GHR_{Before\_Update}]$$

$$Taken = CBTB_{valid}\&!Counter_{MSB}$$

During the prediction process there are 2 values of the GHR:

- GHR_{Before_Update}:The GHR value before being updated with the current prediction outcome. This is what is used to index the *N*-bit counter array and was updated by the last prediction outcome.
- GHR_{After_Update}: The GHR value after being updated with the current predition outcome. This will be used for the next prediction.

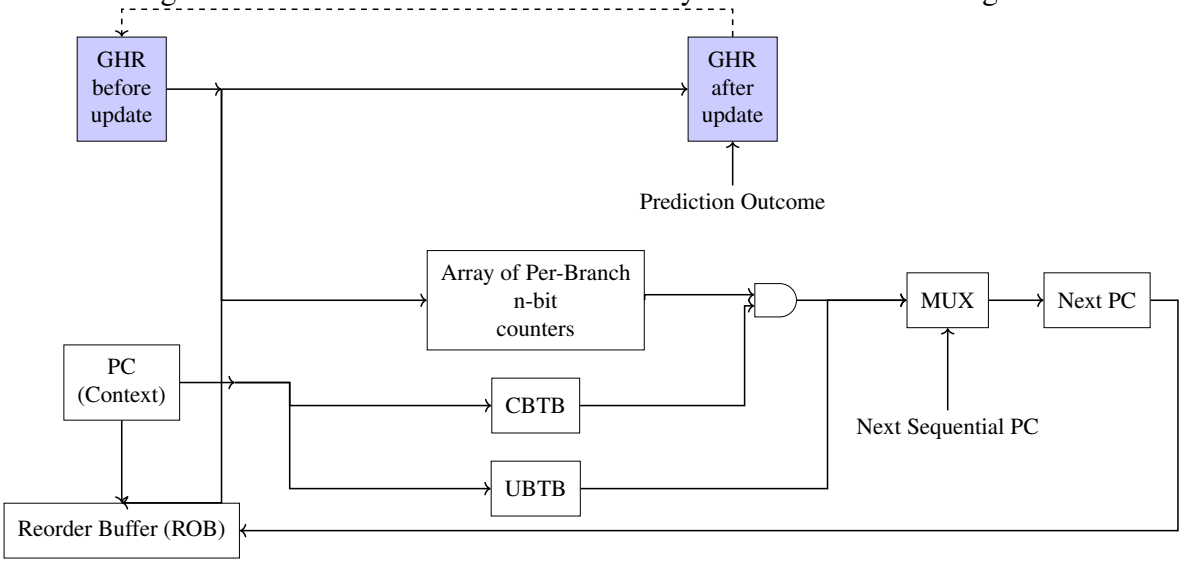
Since the GHR is updated speculatively, the GHR_{After_Update} value may not be the same as the GHR_{Before_Update} value. This means that the GHR must be repaired in case of a misprediction and it is required to store the context of each prediction in the ROB. The GHR_{Before_Update} value is stored in the ROB along with the prediction outcome. This allows the GHR to be restored to its previous state in case of a misprediction.

In summary, the following functions are performed during the speculation process:

- GHR_{Before_Update} is used to index the *N*-bit counter array and retrieve the prediction information.
- GHR_{Before_Update} is stored in the ROB along with the prediction outcome.
- GHR_{After_Update} is generated by shifting in the outcome of the most recent branch instruction.

Figure 3.9 shows a GHR BPU diagram with the *N*-bit counter array and connections to the ROB.

Figure 3.9: GHR with N-bit Counters Array Branch Prediction diagram



In order to support storing the GHR in the ROB, the `BRANCH_PACKET_t` must be modified to include the GHR value. Listing 18 shows the modified definition with the GHR value included.

```
typedef logic [BPU_HISTORY_NUM_BITS-1:0] BPU_HISTORY_t;

typedef struct packed {
    BRANCH_TYPE_t   branch_type;
    logic           is_branch;
    logic           is_taken;
    RV32I_OPERAND_t target;
    RV32I_OPERAND_t program_counter;
    BPU_HISTORY_t   history;
} BRANCH_PACKET_t;
```

Listing 18: SV `BRANCH_PACKET_t` with `BPU_HISTORY_t` definition and assignment

GHR Updates

As mentioned in the previous section, the GHR is updated speculatively with the outcome of the most recent branch instruction. The GHR is a shift register that shifts in the outcome of the most recent branch instruction. Listing 19 shows the SystemVerilog GHR update code. The GHR_{Before_Update} flops the value of GHR_{After_Update}, which is what will allow the next prediction to consume the updated GHR value. It can also be seen that the branch packet is assigned the value of GHR_{Before_Update} in order to store the GHR value in the ROB.

```
BRANCH_PACKET_t branch_packet;
BPU_HISTORY_t ghr_before_update, ghr_after_update;
logic is_taken;

assign ghr_after_update =
 ↪  {ghr_before_update[BPU_HISTORY_NUM_BITS-1:1], is_taken};

always_ff @(posedge clk)
    ghr_before_update <= ghr_after_update;

assign branch_packet.history = ghr_before_update;
```

Listing 19: SV GHR update code

As a comparison, we can see the GHR update code in C++ in Listing 20. The GHR is updated by shifting in the outcome of the most recent branch instruction.

```
uint64_t ghr = 0;

// Update the GHR with the latest prediction
void GHR::update(bool taken) {
  ghr = ((ghr << 1) | taken);
}
```

Listing 20: C++ GHR update code

Counter Array Indexing

The GHR_{Before_Update} is used to index into the *N*-bit counter array. This implies the following:

- SystemVerilog: The flopped GHR value is used to index into the *N*-bit counter array.
- C++: The GHR value **before updating** is used to index into the *N*-bit counter array.

Both implications achieve the same functional result. Speculation functions for SV and C++ are shown in Listing 21 and Listing 22 respectively. It can be noted that the speculation functions are very similar to the *N*-bit counter array speculation functions. The only difference is that the GHR value now provides the index for the *N*-bit counter array instead of the program counter. Besides that, all the logic is the same. In the SV code, the GHR_{Before_Update} is also stored in the branch packet for the ROB. The C++ code does not show this due to the C++ CBP-2016 framework not requiring a recovery mechanism. The C++ code is only used for simulation purposes and does not require the GHR to be stored in the ROB.

```
COUNTER_IDX_t read_idx;
assign read_idx = ghr_before_update;

COUNTER_t counter_read_data;
assign counter_read_data = counters[read_idx];

CBTB_ENTRY_t cbtb_read_data;

logic taken;
BRANCH_PACKET_t branch_packet;

always_comb begin
  branch_packet = '0;
  branch_packet.history = ghr_before_update;
  // Check for positive number with MSB==0 and CBTB valid
  taken =  cbtb_read_data.valid
         & !counter_read_data[BPU_COUNTER_NUM_BITS-1];

  if (taken) begin
    branch_packet.branch_type = CONDITIONAL_TAKEN;
    branch_packet.is_branch = '1;
    branch_packet.is_taken = '1;
    branch_packet.target = cbtb_read_data.target;
  end else begin
    branch_packet.target = next_sequential_pc;
  end
end
```

Listing 21: SV GHR Speculation logic

```
uint64_t ghr_mask = (1ULL << GHR_NUM_BITS) - 1;

// Get the current GHR value
uint64_t GHR::get() const {
  return ghr & ghr_mask;
}

// Predictor indexing function
UINT64 PREDICTOR::GetCounterArrayIDX(UINT64 pc) {
  return GHR.get();
}

UINT64 PREDICTOR::GetBTBIDX(UINT64 pc) {
  UINT64 mask = (1 << BTB_IDX_NUM_BITS) - 1;
  return pc & mask;
}

bool CBTB::GetPrediction(UINT64 btb_idx) {
  return cbtb[btb_idx].valid;
}

// Predictor main prediction function
bool PREDICTOR::GetPrediction(UINT64 PC) {
  UINT64 counter_idx = PREDICTOR::GetCounterArrayIDX(PC);
  UINT64 btb_idx     = PREDICTOR::GetBTBIDX(PC);

  bool counter_pred = counters.GetPrediction(counter_idx);
  bool btb_pred     = cbtb.GetPrediction(btb_idx);
  bool final_pred = counter_pred && btb_pred;
  return final_pred;
}
```

Listing 22: C++ GHR speculation code

3.4.2 Recovery

The GHR must be repaired in case of a misprediction. This means that the GHR must be restored to a previous state before the misprediction occurred. To achieve this, the GHR is stored in the ROB inside the prediction packet. This allows the GHR to be restored to its previous state in case of a misprediction by reading the mispredicting ROB entry and performing the correct update. A diagram showing in which stages the GHR is updated and repaired is shown in Figure 3.10. The following steps are performed during the GHR recovery process:

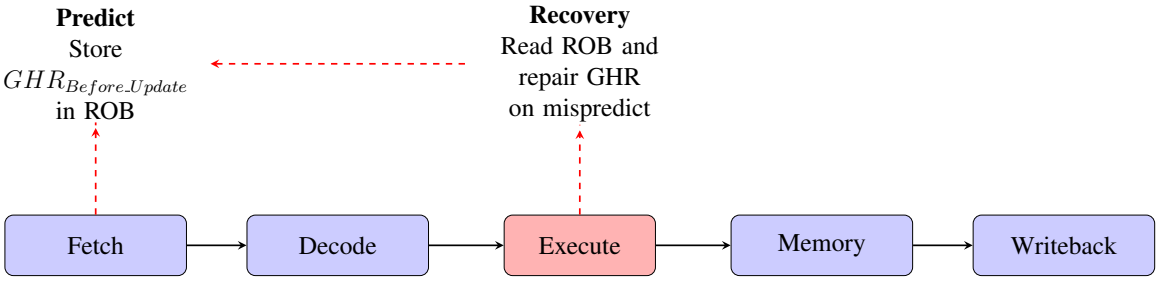
1. GHR_{Before_Update} is stored in the ROB along with the prediction outcome using the branch packet at prediction time.

2. GHR_{After_Update} is generated and used for the next prediction at prediction time.

3. In case of a misprediction, the GHR is restored to its previous state by reading the mispredicting ROB entry.

4. The GHR is updated with the correct outcome of the branch instruction using the ROB's branch_packet GHR_{Before_Update} value and the resolution of the branch instruction. This uses the same logic as the GHR update process.

Figure 3.10: 5-Stage processor diagram with GHR recovery mechanism



Once the GHR is repaired subsequent predictions will use the updated GHR value, which is now correct. The overall speculation process can move forward with its corresponding GHR updates and repairs as needed. Code for the GHR SV recovery process is shown in Listing 23.

```
1  BPU_HISTORY_t ghr_after_update;
2  BPU_HISTORY_t ghr_from_recovery;
3
4  logic  is_taken;
5  assign is_taken =  ROB_JE_CLEAR
6                   ? rob[rob_id].branch_packet.is_taken
7                   : is_prediction_taken;
8
9  assign ghr_from_recovery = rob[rob_id].branch_packet.history;
10
11 always_comb
12   if (ROB_JE_CLEAR)
13     ghr_after_update =
       ↪  {ghr_from_recovery[BPU_HISTORY_NUM_BITS-1:1], is_taken};
14   else
15     ghr_after_update =
       ↪  {ghr_before_update[BPU_HISTORY_NUM_BITS-1:1], is_taken};
```

Listing 23: SV GHR Recovery logic

C++ GHR Recovery

Due to the C++ CBP-2016 framework not requiring a recovery mechanism, the GHR recovery process is not shown in the C++ code. The C++ code is only used for simulation purposes

and does not require the GHR to be stored in the ROB. The GHR recovery process is only shown in the SystemVerilog code. Since the C++ code updates the BPU after each prediction, the GHR updates are always correct and performed at update time. This is not the case for the SystemVerilog code, where the GHR updates are performed speculatively and the branch resolution is not known until the branch instruction is executed. This means that the GHR must be repaired in case of a misprediction, which is not required in the C++ code.

3.4.3 Training

The GHR only provides the context for indexing the *N*-bit counter array and does not require any training. All updates to the GHR are performed during speculation or recovery. For details on the training process for the *N*-bit counter array, please refer to section 3.3.4.

3.4.4 C++ CBP 2016 MPKI Simulation results

The simulation results for the C++ implementation of the GHR are shown in Table 3.8.he results show that the GHR implementation achieves a significant reduction in MPKI compared to the baseline, no-bpu implementation. However, the GHR implementation does not achieve the same level of performance as the *N*-bit counter array implementation in this 7-bit GHR case. This is due to the fact that the GHR implementation only uses the GHR value to index into the *N*-bit counter array, while the *N*-bit counter array implementation uses the program counter which can provide more information about the branch instruction.

Table 3.8: C++ CBP 2016 MPKI results of 1, 2, 3, 4, and 5 bit counter algorithms with GHR and memory requirements using 7-bits for CBTB index and GHR size

Algorithm	GHR Size	MPKI	Improvement (%)	Global Counter size (bytes)	CBTB size (bytes)	Total Memory (bytes)
No BPU	-	53.043	-	-	-	-
GHR 1-bit counter	7	26.777	49.52	16	528	544
GHR 2-bit counter	7	20.569	61.22	32	528	560
GHR 3-bit counter	7	19.130	63.93	48	528	576
GHR 4-bit counter	7	19.064	64.06	64	528	592
GHR 5-bit counter	7	19.277	63.66	80	528	608

Impact of GHR Size

The GHR size is an important factor in the performance of the GHR algorithm. A larger GHR size allows for more history to be stored, which can improve the accuracy of the predictions. However, a larger GHR size also increases the memory requirements. In order to analyze the impact of GHR size on the performance of the GHR algorithm, a sweep of GHR sizes was performed using a fixed 2-bit counter size. The results of this sweep are shown in Table 3.9. Some findings from the results are:

- The GHR size has a significant impact on the performance of the GHR algorithm. A larger GHR size results in a lower MPKI and a higher improvement percentage.

- The memory requirements increase with the GHR size. This is due to the fact that a larger GHR size requires more memory to index the *N*-bit counter arrays.
- The GHR algorithm achieves a significant reduction in MPKI compared to the baseline, no-bpu implementation and even outperforms the 2-bit counter implementation in some cases.

Table 3.9: C++ CBP 2016 MPKI results of 2 bit counter algorithm with GHR and memory requirements using 6 to 15 bits for GHR size

Algorithm	GHR Size	MPKI	Improvement (%)	Global Counter size (bytes)
No BPU	-	53.043	-	-
GHR 2-bit counter	6	22.356	57.85	16
GHR 2-bit counter	7	20.569	61.22	32
GHR 2-bit counter	8	17.995	66.07	64
GHR 2-bit counter	9	16.383	69.11	128
GHR 2-bit counter	10	14.890	71.93	256
GHR 2-bit counter	11	13.647	74.27	512
GHR 2-bit counter	12	12.529	76.38	1024
GHR 2-bit counter	13	11.565	78.20	2048
GHR 2-bit counter	14	10.716	79.80	4096
GHR 2-bit counter	15	9.828	81.47	8192

GHR vs 2-bit Counter

The GHR algorithm was also compared to the 2-bit counter implementation using a fixed GHR size of 15 bits. The results of this comparison are shown in Table 3.10. The results show that the GHR algorithm achieves a significant reduction in MPKI compared to the baseline, no-bpu implementation and even outperforms the 2-bit counter implementation by 17.98%. Considering that both GHR and 2-bit counter implementations use the same 8KB memory size, the GHR algorithm is able to achieve better performance with the same memory requirements. This is due to the fact that the GHR algorithm uses the GHR value to index into the *N*-bit counter array, which can provide more information about the branch execution history compared to the program counter used in the 2-bit counter implementation.

Table 3.10: C++ CBP 2016 MPKI results comparison between 2-bit counter and GHR 2-bit counter algorithms with the same memory budget (8KB)

Algorithm	MPKI	Improvement vs No BPU (%)	Improvement vs 2-bit counter (%)	Global Counter size (bytes)
No BPU	53.043	-	-	-
2-bit counter	11.982	77.41	-	8,192
GHR 2-bit counter	9.828	81.47	17.98	8,192

3.4.5 SystemVerilog FPGA Synthesis results

The GHR algorithm was implemented in SystemVerilog and synthesized for an FPGA. The synthesis results are shown in Table 3.11. The results show that the GHR implementation achieves a significant reduction in MPKI compared to the baseline, no-bpu implementation. However, the GHR implementation does not achieve the same level of performance as the *N*-bit counter array implementation in this 7-bit GHR case. This is due to the fact that the GHR implementation only uses the GHR value to index into the *N*-bit counter array, while the *N*-bit counter array implementation uses the program counter which can provide more information about the branch instruction.

Table 3.11: Synthesis results for 1, 2, 3, and 4-bit counter algorithms showing $Fmax$ and resource usage using a 7-bit index for the CBTB, UBTB, GHR and n-bit counters array

Configuration	**Fmax (MHz)**	**Fmax Change (%)**	**Logic Elements**	**Resource Change (%)**
no bpu	55.66	0.00	27,676	0.00
1-bit counter 7-bit GHR	51.88	-6.79	38,488	+39.07
2-bit counter 7-bit GHR	50.23	-9.76	38,633	+39.59
3-bit counter 7-bit GHR	49.73	-10.65	39,032	+41.03
4-bit counter 7-bit GHR	49.15	-11.70	39,164	+41.51

3.4.6 SystemVerilog IPC and MPKI results

Similar to the previous algorithm in 3.3.7, the GHR FPGA implementation was limited to a 7-bit GHR due to hardware constraints. Simulations, on the other hand, were performed with a 15-bit GHR. **IPC, MPKI** and **Cycle** metrics are the results of the GHR algorithm measured for the Dhrystone, Qsort and CoreMark workloads, shown in tables 3.12, 3.13 and 3.14 respectively. An arithmetic mean of the improvements across all workloads is shown in Table 3.15. The results show that the GHR implementation achieves a significant reduction in MPKI compared to the baseline, no-bpu implementation. It also achieves the following improvements over the 2-bit counter algorithm given the same memory size:

- **1.65%** improvement in MPKI
- **6.07%** improvement in IPC
- **0.67%** improvement in Cycles

3.4.7 Conclusion

The GHR algorithm is a powerful branch prediction algorithm that uses a global history register to keep track of the outcomes of the last N branches. The GHR is used to index into a *N*-bit

counters array that contains the prediction information for each possible global history. The GHR algorithm achieves a significant reduction in MPKI compared to the baseline, no-bpu implementation and even outperforms the 2-bit counter implementation in some cases. The GHR algorithm is able to achieve better performance with similar memory requirements as the 2-bit counter implementation. Similar to the previous algorithm results, the SV and C++ simulation differences are expected due to the different workloads used and the inclusion of non-conditional branches in RTL simulations.

Table 3.12: GHR Dhrystone 500 runs results (15-bit index, 8KB Global Counter Array)

Predictor	MPKI	Improvement	IPC	Improvement	Cycles	Improvement
No BPU	110.40	-	0.43	-	389348	-
GHR	21.13	80.85%	0.51	18.81%	325541	16.39%

Table 3.13: GHR Qsort 2048 numbers results (15-bit index, 8KB Global Counter Array)

Predictor	MPKI	Improvement	IPC	Improvement	Cycles	Improvement
No BPU	180.89	-	0.37	-	600824	-
GHR	51.98	71.26%	0.46	24.22%	492892	17.96%

Table 3.14: GHR CoreMark single run results (15-bit index, 8KB Global Counter Array)

Predictor	MPKI	Improvement	IPC	Improvement	Cycles	Improvement
No BPU	188.62	-	0.41	-	3429394	-
GHR	133.91	29.01%	0.44	7.78%	3210365	6.39%

Table 3.15: GHR arithmetic mean improvements across all workloads (15-bit index, 8KB Global Counter Array, No BPU baseline)

Algorithm	MPKI Improvement	IPC Improvement	Cycles Improvement
2-bit Counter	59.39%	15.97%	13.49%
2-bit Counter GHR	60.37%	16.94%	13.58%

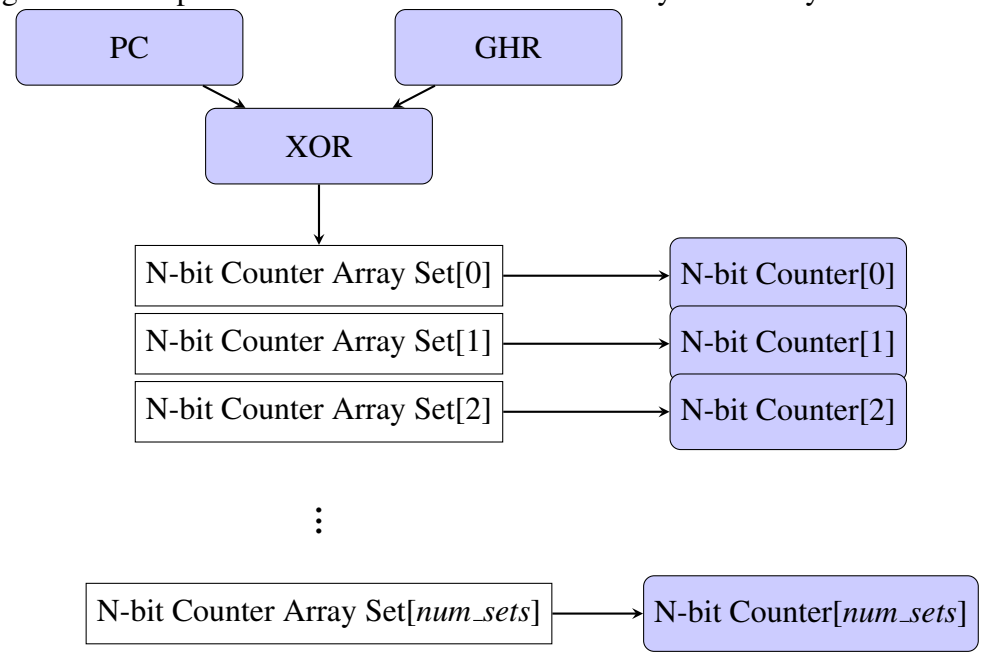
3.5 Gshare

The Gshare algorithm is an extension of the Global History Register (GHR) algorithm described in the previous section. The Gshare algorithm was first proposed by McFarling [10] as a way to improve the prediction accuracy of branch predictors. Similar to the GHR algorithm, Gshare uses a global history register to keep track of the outcomes of the last N branches. However, Gshare introduces a key improvement: it combines (hashes) the global history with the branch address (PC) to index into the pattern history table (PHT).

The primary difference between Gshare and GHR is the indexing mechanism used to access the N-bit counter array. In GHR, the GHR value is used directly as the index, whereas

in Gshare, the GHR value is XORed with the PC to generate the index. This XOR operation helps reduce aliasing by distributing different branch patterns more evenly across the PHT and reducing interference between different branches that might have the same GHR value but different behaviors. Figure 3.11 shows a diagram of the Gshare algorithm with the PC XORed with the GHR to index into the N-bit counter array.

Figure 3.11: Representation of an N-bit Counter Array indexed by PC XOR GHR



The following general points are important to note about the Gshare algorithm:

- The GHR is a shift register that shifts in the outcome of the most recent branch instruction, identical to the GHR algorithm.
- The GHR is XORed with the PC to create an index for the N-bit counter array that contains the prediction information.
- The GHR is speculatively updated with the outcome of the most recent branch instruction.
- The GHR must be repaired in case of a misprediction, which means that the GHR must be restored to a previous state before the misprediction occurred, identical to the GHR algorithm.
- The XOR result between the GHR and PC is not stored in the GHR, it is only used to index the N-bit counter array.

3.5.1 Speculation and Context

As mentioned in the previous algorithm, the GHR is used as context for the branch direction predictor, but Gshare adds the PC information to this context through the XOR operation.

Implementation is similar to the GHR algorithm with the only difference being that the GHR value is XORed with the PC to create an index to the *N*-bit counter array. This structure follows the two-level adaptive predictor model described by Yeh and Patt [13]. An abstract formula for the prediction function in Gshare:

$$Counter = CounterArray[GHR_{Before\_Update} \oplus PC]$$

$$Taken = CBTB_{valid}\&!Counter_{MSB}$$

During the prediction process, there are still 2 values of the GHR, identical to the GHR algorithm:

- GHR_{Before_Update}: The GHR value before being updated with the current prediction outcome. This is what is used (after XORing with PC) to index the *N*-bit counter array and was updated by the last prediction outcome.
- GHR_{After_Update}: The GHR value after being updated with the current prediction outcome. This will be used for the next prediction.

Since the GHR is updated speculatively, the GHR_{After_Update} value may not be the same as the GHR_{Before_Update} value. This means that the GHR must be repaired in case of a misprediction and it is required to store the context of each prediction in the ROB. The GHR_{Before_Update} value is stored in the ROB along with the prediction outcome, identical to the GHR algorithm. This allows the GHR to be restored to its previous state in case of a misprediction.

In summary, the following functions are performed during the speculation process:

- GHR_{Before_Update} is XORed with the PC to create an index for the *N*-bit counter array and retrieve the prediction information.
- GHR_{Before_Update} is stored in the ROB along with the prediction outcome.
- GHR_{After_Update} is generated by shifting in the outcome of the most recent branch instruction.

Figure 3.12 shows a Gshare BPU diagram with the *N*-bit counter array and connections to the ROB, highlighting the XOR operation between the PC and GHR.

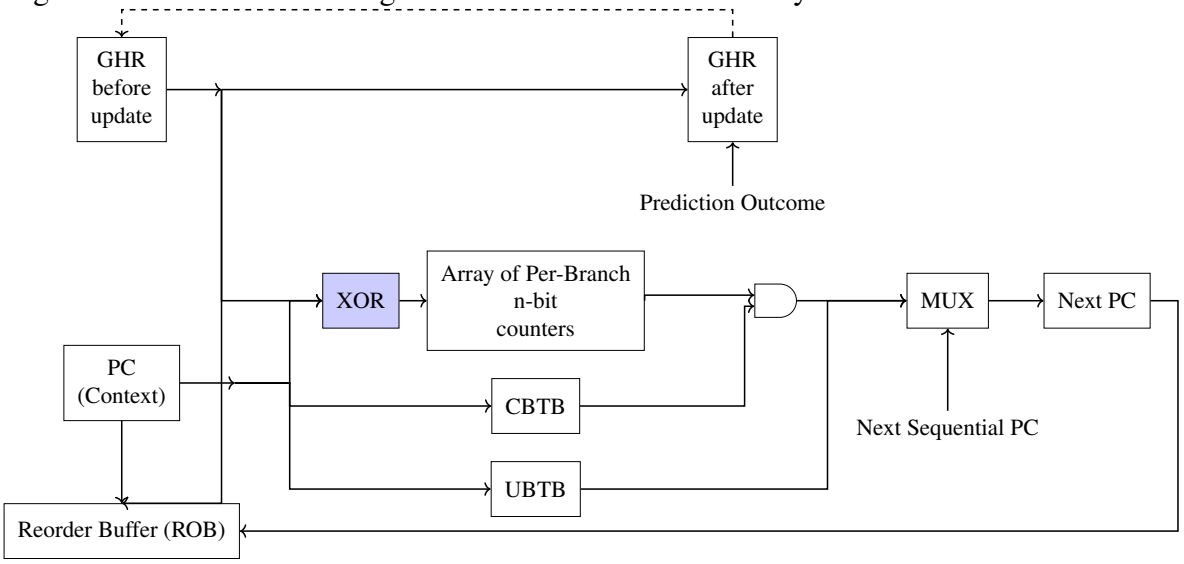
The definition is the same as for the GHR algorithm since the GHR storage requirements are identical. The only difference is in how the GHR value is used to index the counter array.

Counter Array Indexing

The key difference between Gshare and GHR is in how the counter array is indexed. In Gshare, the GHR_{Before_Update} is XORed with the PC to create an index for the *N*-bit counter array. This implies the following:

- SystemVerilog: The flopped GHR value is XORed with the PC to index into the *N*-bit counter array.

Figure 3.12: Gshare BPU diagram with the *N*-bit counter array and connections to the ROB



- C++: The GHR value **before updating** is XORed with the PC to index into the *N*-bit counter array.

Both implications achieve the same functional result. Speculation functions for SV and C++ are shown in Listing 24 and Listing 25 respectively. It can be noted that the speculation functions are very similar to the GHR speculation functions. The only difference is that the GHR value is now XORed with the PC to provide the index for the *N*-bit counter array instead of using the GHR value directly.

```
COUNTER_IDX_t read_idx;
RV32I_OPERAND_t program_counter;
assign read_idx =  ghr_before_update
                 ^ program_counter[BPU_HISTORY_NUM_BITS-1:0];

COUNTER_t counter_read_data;
assign counter_read_data = counters[read_idx];

CBTB_ENTRY_t cbtb_read_data;

logic taken;
BRANCH_PACKET_t branch_packet;

always_comb begin
  branch_packet = '0;
  branch_packet.history = ghr_before_update;
  // Check for positive number with MSB==0 and CBTB valid
  taken =  cbtb_read_data.valid
         & !counter_read_data[BPU_COUNTER_NUM_BITS-1];

  if (taken) begin
    branch_packet.branch_type = CONDITIONAL_TAKEN;
    branch_packet.is_branch = '1;
    branch_packet.is_taken = '1;
    branch_packet.target = cbtb_read_data.target;
  end else begin
    branch_packet.target = next_sequential_pc;
  end
end
```

Listing 24: SV Gshare Speculation logic

```cpp
uint64_t ghr_mask = (1ULL << GHR_NUM_BITS) - 1;

// Get the current GHR value
uint64_t GHR::get() const {
  return ghr & ghr_mask;
}

// Predictor indexing function
UINT64 PREDICTOR::GetCounterArrayIDX(UINT64 pc) {
  uint64_t gshare = pc ^ GHR.get();
  gshare &= ghr_mask;
  return gshare;
}

UINT64 PREDICTOR::GetBTBIDX(UINT64 pc) {
  UINT64 mask = (1 << BTB_IDX_NUM_BITS) - 1;
  return pc & mask;
}

bool CBTB::GetPrediction(UINT64 btb_idx) {
  return cbtb[btb_idx].valid;
}

// Predictor main prediction function
bool PREDICTOR::GetPrediction(UINT64 PC) {
  UINT64 counter_idx = PREDICTOR::GetCounterArrayIDX(PC);
  UINT64 btb_idx     = PREDICTOR::GetBTBIDX(PC);

  bool counter_pred = counters.GetPrediction(counter_idx);
  bool btb_pred     = cbtb.GetPrediction(btb_idx);
  bool final_pred = counter_pred && btb_pred;
  return final_pred;
}
```

Listing 25: C++ Gshare speculation code

3.5.2 Recovery

The GHR recovery process in Gshare is identical to the GHR algorithm. The GHR must be repaired in case of a misprediction by restoring it to a previous state before the misprediction occurred. This is accomplished by storing the GHR in the ROB inside the prediction packet, just as in the GHR algorithm. The recovery process follows the same steps:

1. GHR_{Before_Update} is stored in the ROB along with the prediction outcome using the branch packet at prediction time.

2. GHR_{After_Update} is generated and used for the next prediction at prediction time.

3. In case of a misprediction, the GHR is restored to its previous state by reading the mispredicting ROB entry.

4. The GHR is updated with the correct outcome of the branch instruction using the ROB's branch_packet GHR_{Before_Update} value and the resolution of the branch instruction.

Please refer to section 3.4.2 for a detailed description of the recovery process for the GHR algorithm.

3.5.3 Training

The training process for Gshare is identical to the GHR algorithm. The GHR only provides the context for indexing the *N*-bit counter array (after being XORed with the PC) and does not require any training itself. All updates to the GHR are performed during speculation or recovery. For details on the training process for the *N*-bit counter array, please refer to section 3.3.4.

3.5.4 C++ CBP 2016 MPKI Simulation results

The simulation results for the C++ implementation of Gshare are shown in Table 3.16. The results show that the Gshare implementation achieves a significant reduction in MPKI compared to both the baseline no-bpu implementation and the GHR implementation. This improvement is due to the XOR operation between the PC and GHR, which helps reduce aliasing in the pattern history table.

Table 3.16: Gshare MPKI Results for Different Counter Sizes with 7-bit GHR

Algorithm	GHR Size	MPKI	Improvement (%)	Global Counter size (bytes)	CBTB size (bytes)	Total Memory (bytes)
No BPU	-	53.043	-	-	-	-
Gshare 1-bit counter	7	23.488	55.72	16	528	544
Gshare 2-bit counter	7	18.824	64.51	32	528	560
Gshare 3-bit counter	7	17.700	66.63	48	528	576
Gshare 4-bit counter	7	17.708	66.62	64	528	592
Gshare 5-bit counter	7	17.992	66.08	80	528	608

Impact of GHR Size in Gshare

Similar to the GHR algorithm, the GHR size is an important factor in the performance of the Gshare algorithm. A sweep of GHR sizes was performed using a fixed 2-bit counter size to analyze the impact of GHR size on the performance of the Gshare algorithm. The results of this sweep are shown in Table 3.17.

Some key findings from the results are:

- The GHR size has a significant impact on the performance of the Gshare algorithm, with larger GHR sizes generally resulting in lower MPKI and higher improvement percentages.

- The memory requirements increase with the GHR size, similar to the GHR algorithm.
- The Gshare algorithm consistently outperforms both the baseline no-bpu implementation and the GHR algorithm across different GHR sizes due to the improved context information.

Table 3.17: Gshare MPKI Results for Different GHR Sizes with 2-bit Counter

Algorithm	GHR Size	MPKI	Improvement (%)	Global Counter size (bytes)
No BPU	-	53.043	-	-
Gshare 2-bit counter	6	22.356	57.85	16
Gshare 2-bit counter	7	20.569	61.22	32
Gshare 2-bit counter	8	17.995	66.07	64
Gshare 2-bit counter	9	16.383	69.11	128
Gshare 2-bit counter	10	14.890	71.93	256
Gshare 2-bit counter	11	13.647	74.27	512
Gshare 2-bit counter	12	12.529	76.38	1024
Gshare 2-bit counter	13	11.565	78.20	2048
Gshare 2-bit counter	14	10.716	79.80	4096
Gshare 2-bit counter	15	9.828	81.47	8192

Gshare vs 2-bit Counter

The Gshare algorithm was compared to the 2-bit counter implementations using a fixed GHR size and counter array index of 15 bits. The results of this comparison are shown in Table 3.18. The results demonstrate that the Gshare algorithm achieves a significant reduction in MPKI compared to the baseline no-bpu implementation and the 2-bit counter implementations with and without GHR. Considering that Gshare and 2-bit counter implementations use the same memory size, the Gshare algorithm achieves better performance without additional memory requirements. This improvement is due to the combination of global history and PC information in the Gshare algorithm, which provides more accurate predictions.

Table 3.18: C++ CBP 2016 MPKI results comparison between 2-bit counter, GHR 2-bit counter and Gshare 2-bit counter algorithms with the same memory budget (8KB)

Algorithm	MPKI	Improvement vs No BPU (%)	Improvement vs 2-bit counter (%)	Improvement vs GHR (%)	Global Counter size (bytes)
No BPU	53.043	-	-	-	-
2-bit counter	11.982	77.41	-	-	8,192
GHR 2-bit counter	9.828	81.47	17.98	-	8,192
Gshare 2-bit counter	7.825	85.25	34.69	20.38	8,192

3.5.5 SystemVerilog FPGA Synthesis results

The Gshare algorithm was implemented in SystemVerilog and synthesized for an FPGA. The synthesis results are shown in Table 3.19. The results indicate that the Gshare implementation achieves a significant reduction in MPKI compared to the baseline no-bpu implementation. The hardware overhead of Gshare compared to the GHR implementation is minimal, consisting mainly of the additional XOR gate needed to combine the PC and GHR values.

Table 3.19: Synthesis results for 1, 2, 3, and 4-bit counter algorithms showing $Fmax$ and resource usage using a 7-bit index for the CBTB, UBTB and n-bit counters array with Gshare

Configuration	**Fmax (MHz)**	**Fmax Change (%)**	**Logic Elements**	**Resource Change (%)**
No BPU	55.66	0.00	27,676	0.00
1-bit counter 7-bit Gshare	51.43	-7.60	38,337	+38.52
2-bit counter 7-bit Gshare	51.11	-8.17	38,741	+40.00
3-bit counter 7-bit Gshare	50.78	-8.77	38,800	+40.19
4-bit counter 7-bit Gshare	50.40	-9.45	38,839	+40.33

3.5.6 SystemVerilog IPC and MPKI results

Similar to the GHR algorithm, the Gshare FPGA implementation was limited to a 7-bit GHR due to hardware constraints, while simulations were performed with a 15-bit GHR. **IPC, MPKI** and **Cycle** metrics for the Gshare algorithm were measured for the Dhrystone, Qsort, and CoreMark workloads, shown in tables 3.20, 3.21, and 3.22 respectively. An arithmetic mean of the improvements across all workloads is shown in Table 3.23. The results show that the Gshare implementation achieves a significant reduction in MPKI compared to both the baseline no-bpu implementation and the GHR algorithm.

Table 3.20: Gshare Dhrystone 500 runs results (15-bit index, 8KB Global Counter Array)

Predictor	MPKI	Improvement	IPC	Improvement	Cycles	Improvement
No BPU	110.40	-	0.43	-	389348	-
Gshare	15.20	86.23%	0.52	20.23%	321050	17.54%

Table 3.21: Gshare Qsort 2048 numbers results (15-bit index, 8KB Global Counter Array)

Predictor	MPKI	Improvement	IPC	Improvement	Cycles	Improvement
No BPU	180.89	-	0.37	-	600824	-
Gshare	51.01	71.80%	0.45	21.62%	491783	18.15%

Table 3.22: Gshare CoreMark single run results (15-bit index, 8KB Global Counter Array)

Predictor	MPKI	Improvement	IPC	Improvement	Cycles	Improvement
No BPU	188.62	-	0.41	-	3429394	-
Gshare	131.49	30.29%	0.45	9.76%	3189479	7.00%

Table 3.23: Gshare arithmetic mean improvements across all workloads (15-bit index, 8KB Global Counter Array with and without GHR, No BPU baseline)

Algorithm	MPKI Improvement	IPC Improvement	Cycles Improvement
2-bit Counter	59.39%	15.97%	13.49%
2-bit Counter GHR	60.37%	16.94%	13.58%
2-bit Counter Gshare	62.77%	17.44%	14.23%

3.5.7 Conclusion

The Gshare algorithm is an advanced branch prediction algorithm that builds upon the GHR algorithm by combining the global history register with the branch address (PC) through an XOR operation. This XOR operation helps reduce aliasing in the pattern history table (N-bit counters array), resulting in improved prediction accuracy. The Gshare algorithm achieves a reduction in MPKI compared to both the baseline no-bpu implementation and the GHR algorithm without requiring additional memory resources. The hardware overhead of implementing Gshare compared to GHR is minimal, consisting mainly of an additional XOR gate. The Gshare algorithm demonstrates the power of combining different sources of information (branch history and branch address) to improve branch prediction accuracy.

Chapter 4

Compiling and running Baremetal C on RV32I RTL

4.1 Introduction

This chapter describes the process of compiling and running Baremetal C workloads on the RV32I RTL. Baremetal workloads are programs that run directly on the hardware without an operating system. They are typically used in embedded systems and real-time applications where low-level control over the hardware is required[14]. For this work, the workloads are written in C and compiled for the RISC-V RV32I architecture with the intent of running the binaries on an RTL simulator. The following tools and repositories were used:

- **RISC-V GCC toolchain**[15]: A set of compilers and tools for RISC-V architecture.
- **riscv-scratchpad**[16]: A public repository containing example Baremetal C workloads for RISC-V.
- **riscv_tests**[17]: A public repository containing benchmark programs for RISC-V architecture.

In order to run Baremetal C workloads on the RV32I RTL, the following items are required:

- **Startup code** that initializes the stack pointer and global pointer. It handles the entry point of the program, sets up the initial state of the program and runs the main function.
- **Linker script** that defines the memory layout of the program. It specifies the locations of the stack pointer and global pointer in memory, as well as the sections for code and data. This allows the compiler to generate the correct addresses for the program's variables and functions using the correct virtual addresses for the RISC-V implementation.
- **Memory Dump** that contains the compiled Baremetal C workload and other RAM data. The memory dump is generated after compiling and it contains the binary representation of the program's code and data. This memory dump is used to load the program into the RV32I RTL simulator, it includes both ROM and RAM data.

4.2 Startup code (`startup.c`)

Due to the lack of an operating system or runtime, Baremetal code must include appropiate stack pointer (SP) and global pointer (GP) initialization. The stack pointer is used to manage function calls and local variables, while the global pointer is used to access global variables and data structures. For this work, there are four critical functions to implement:

- `_enter`: Initializes the the stack pointer and global pointer. It is placed on the reset Program Counter address of the CPU and will be executed when the CPU is reset.
- `_start`: Sets up the initial state of the program, including initializing global variables and calling the main function. It is called by the `_enter` function.
- `main`: The main function of the Baremetal C workload, where the program execution begins. Called by the `_start` function.
- `_exit`: Called after the `main` function has completed its execution.

The existing startup code in **riscv-scratchpad**[16] shows implementations of the functions mentioned above and is used as a reference for this work. It includes proper initialization of the RAM memory regions, which are not applicable for the RTL simulation since the memory is directly loaded into the simulator. For a real hardware implementation, the memory regions would be initialized with the appropriate values using the startup code routines. For this work, a more basic implementation of the startup code is used removing the memory initialization. This code is shown in Listing 26.

The following key points are important to note about the startup code:

- The `_enter` function is marked as `naked`. This means that the function does not have a prologue or epilogue, and it is also placed in a specific section of the memory. The `naked` attribute is used to indicate that the function does not need to save or restore registers, which is important for low-level initialization code.
- The `main` function is where the actual program execution begins. It is called by the `_start` function after the stack pointer and global pointer have been initialized. For this example workload, the `main` function simply returns a value of 42.
- The `_exit` function is called after the main function has completed its execution. It enters an infinite loop to prevent returning to any other code.
- The `SP` and `GP` are initialized using the assembly `la` instruction, which loads the address of the specified label into the register. The labels `_sp` and `__global_pointer$` are defined in the linker script and represent the locations of the stack pointer and global pointer in memory. Section 4.3 describes the linker script used for this work.

```c
#include <stdint.h>
#include <string.h>

// Generic C function pointer.
typedef void(*function_t)(void) ;
extern void _enter(void)  __attribute__ ((naked,
 ↪  section(".text.metal.init.enter")));

// Entry and exit points as C functions.
extern void _start(void) __attribute__ ((noreturn));
void _Exit(int exit_code) __attribute__ ((noreturn,noinline));

// Standard entry point, no arguments.
extern int main(void);

void _enter(void) {
    // Setup SP and GP, locations defined in linker
    __asm__ volatile  (
        ".option push;"
        // Without 'norelax' the global pointer will
        // be loaded relative to the global pointer!
         ".option norelax;"
        "la    gp, __global_pointer$;"
        ".option pop;"
        "la    sp, _sp;"
        "jal   zero, _start;");
    // _start() will be called with no return.
}

void _start(void) {
    int return_code = main();
    _Exit(return_code);
}

int main(void) {
    int number_of_life = 42;
    return number_of_life;
}

void _Exit(int exit_code) {
    (void)exit_code;
    while (1) {
        __asm__ volatile (
            "li zero, 42"
        );}
}
```

Listing 26: Baremetal C startup code

4.2.1 `_enter` function

Once compiled, we can observe the disassembly of the `_enter` function using the `objdump` tool. Listing 27 shows this information. The `la` instruction loads the address of the global

pointer into the `gp` register, and the address of the stack pointer into the `sp` register. The `jal zero, _start` instruction jumps to the `_start` function without returning.

```
00400000 <_enter>:
void _enter(void) {
    // Setup SP and GP
    // The locations are defined in the linker script
    __asm__ volatile  (
  400000:       0fc11197                auipc   gp,0xfc11
  400004:       80018193                addi    gp,gp,-2048 # 10010800 <__global_pointer$>
  400008:       0fc10117                auipc   sp,0xfc10
  40000c:       7f810113                addi    sp,sp,2040 # 10010800 <__global_pointer$>
  400010:       0080006f                j       400018 <_start>
        "jal   zero, _start;"
        :  /* output: none %0 */
        : /* input: none */
        : /* clobbers: none */);
    // This point will not be executed, _start() will be called with no return.
}
  400014:       00000013                nop
```

Listing 27: `_enter` function disassembly

It can be observed that the first instruction is a `auipc` instruction, which starts the initialization of the `gp` register. This instruction is exactly at the `0x400000` address, which is the reset program counter of the CPU, indicating that the `_enter` function is placed at the reset address and will be the first code executed by the CPU.

4.2.2 `_start` function

The `_start` function is also disassembled using the `objdump` tool. Listing 28 shows the disassembly of the `_start` function. The `jal` instruction jumps to the `main` function, and the return value is stored in the `a0` register. The `_Exit` function is then called with the return value as an argument.

```
00400018 <_start>:

void _start(void)
  400018:        fe010113                addi        sp,sp,-32
  40001c:        00112e23                sw        ra,28(sp)
  400020:        00812c23                sw        s0,24(sp)
  400024:        02010413                addi        s0,sp,32
    int return_code = main();
  400028:        010000ef                jal        400038 <main>
  40002c:        fea42623                sw        a0,-20(s0)
    _Exit(return_code);
  400030:        fec42503                lw        a0,-20(s0)
  400034:        034000ef                jal        400068 <_Exit>
```

Listing 28: `_start` function disassembly

4.2.3 `main` function

The `main` function is also disassembled using the `objdump` tool. Listing 29 shows the disassembly of the `main` function. The `li` instruction loads the immediate value 42 into the `a5` register, and the value is stored in memory using the `sw` instruction. The return value is then loaded from memory and returned to the caller using the same `-20(s0)` memory address.

```
  00400038 <main>:

int main(void) {
  400038:        fe010113                addi        sp,sp,-32
  40003c:        00112e23                sw        ra,28(sp)
  400040:        00812c23                sw        s0,24(sp)
  400044:        02010413                addi        s0,sp,32
    int number_of_life = 42;
  400048:        02a00793                li        a5,42
  40004c:        fef42623                sw        a5,-20(s0)
    return number_of_life;
  400050:        fec42783                lw        a5,-20(s0)
}
  400054:        00078513                mv        a0,a5
  400058:        01c12083                lw        ra,28(sp)
  40005c:        01812403                lw        s0,24(sp)
  400060:        02010113                addi        sp,sp,32
  400064:        00008067                ret
```

Listing 29: `main` function disassembly

4.2.4 `_Exit` function

Finally, the `_Exit` function disassembly is shown in Listing 30. The `li` instruction loads the immediate value 42 into the `zero` register, and the `j` instruction jumps to the same address, creating an infinite loop. This can be used for the RTL simulation to detect when the program

has finished executing. By using the linear address of the `_Exit` function as the qualifier, our testbench can be configured to stop the simulation when the program reaches this point.

```
00400068 <_Exit>:

void _Exit(int exit_code) {
  400068:       fe010113                addi    sp,sp,-32
  40006c:       00112e23                sw      ra,28(sp)
  400070:       00812c23                sw      s0,24(sp)
  400074:       02010413                addi    s0,sp,32
  400078:       fea42623                sw      a0,-20(s0)
    (void)exit_code;
    // Halt
    while (1)
    __asm__ volatile  (
  40007c:       02a00013                li      zero,42
  400080:       ffdff06f                j       40007c <_Exit+0x14>
```

Listing 30: `_Exit` function disassembly

4.2.5 Conclusion

An appropiate startup code is essential for Baremetal C workloads to run on the RV32I RTL. With the correct `SP` and `GP` initialization, the program can run without an operating system allowing functions to be called and local variables to be managed. While the example shows a basic `main` function that simply returns a value, this function can be extended to include any logic required for the workload. This work keeps the same startup code and simply changes the definition of the `main` function to include the workload.

4.3 Linker script (`linker.lds`)

The linker script is a file that defines the memory layout of the program. It specifies the locations of the stack pointer and global pointer in memory, as well as the sections for code and data. It is also used by the compiler to generate the correct addresses for the program's variables and functions. The existing linker script in **riscv-scratchpad**[16] is used as a reference and is an excelent example that requires only minor modifications for this work. There are three memory regions defined in the original linker script:

- `itim`: Instruction tightly-integrated memory. This is not used in this work and is removed from the linker script.
- `ram`: RAM memory region. This is used for the program's data, stack, heap, and other variables.
- `rom`: ROM memory region. This is used for the program's code and constants.

The original linker `MEMORY` definitions are shown in Listing 31. There are three important parameters to configure[18]:

- `ORIGIN`: The starting address of the memory region. This will be used by the compiler to generate the correct addresses for the program's variables and functions. The address values must be aligned to the RTL implementation.
- `LENGTH`: The size of the memory region in bytes. This will be used by the compiler to allocate memory for the program's variables and functions, ensuring that they fit within the specified memory region.
- `((attributes))`: The attributes for the memory region. This specifies whether the memory region is readable, writable, executable, or not accessible.

```
MEMORY
{
    itim (airwx) : ORIGIN = 0x8000000, LENGTH = 0x2000
    ram (arw!xi) : ORIGIN = 0x80000000, LENGTH = 0x4000
    rom (irx!wa) : ORIGIN = 0x20010000, LENGTH = 0x6a120
}
```

Listing 31: Linker `MEMORY` original definitions

For this work, the memory regions are defined in Listing 32. It is important to note that in the RTL implementation both ROM and RAM used 8-bit memory sets. This simplifies the mapping since each memory set is a byte. Code shown assumes an 8KB ROM and 32KB RAM. The `START_ADDR` values are also defined, which will align with the modified linker definitions in Listing 33.

```
package mem_pkg;

  import RV32I_common_pkg::*;

  `define ROM_NUM_SETS 8000
  `define RAM_NUM_SETS 32000

  parameter ROM_NUM_SETS = `ROM_NUM_SETS;
  parameter ROM_START_ADDR = 32'h400000;
  parameter ROM_END_ADDR = ROM_START_ADDR + ((`RV32I_INSTRUCTION_WIDTH /
   ↪  8) * (ROM_NUM_SETS - 1));

  parameter RAM_NUM_SETS = `RAM_NUM_SETS;
  parameter RAM_START_ADDR = 32'h10010000;
  parameter RAM_END_ADDR = RAM_START_ADDR + ((`RV32I_INSTRUCTION_WIDTH /
   ↪  8) * (RAM_NUM_SETS - 1));

endpackage : mem_pkg;
```

Listing 32: SV memory_pkg definitions

```
MEMORY
{
    rom (irx!wa) : ORIGIN = 0x400000,   LENGTH = 8000
    ram (arw!xi) : ORIGIN = 0x10010000, LENGTH = 32000
}
```

Listing 33: Linker `MEMORY` modified definitions for RV32I RTL simulation

4.4 Compiling Baremetal C workloads and generating memory dumps

4.4.1 Compilation

With a proper startup code and linker script, Baremetal C workloads can be compiled for the RV32I architecture. The `riscv32-unknown-elf-gcc` compiler is used to compile the Baremetal C workloads. Only the three following `gcc` options are required in this work to ensure `RISC-V RV32I` compatibility[19] and linker script usage:

- `-mabi=ilp32`: Specifies the ABI (Application Binary Interface) to use for the compiled program. The `ilp32` ABI is used for 32-bit RISC-V architectures and specifies that integers, long integers, and pointers are all 32 bits in size.
- `-march=rv32i`: Specifies the architecture to use for the compiled program. The `rv32i` architecture is the base integer instruction set for 32-bit RISC-V architectures.
- `-T linker.lds`: Specifies the linker script to use for the compiled program. The `linker.lds` file defines the memory layout of the program and is used by the compiler to generate the correct addresses for the program's variables and functions.

4.4.2 Memory dump generation

In order to run the compiled workloads on the RV32I RTL a memory dump is required. For this work, it is necessary to generate two memory dumps: one for the ROM and another for the RAM. The `objcopy` tool, which is part of the RISC-V GCC toolchain is used to extract the contents of the ROM and RAM sections from the compiled binary. The different sections in the linker must be specified to the `objcopy` command to determine which binaries to dump. Listings 34 and 35 show the commands used to generate the memory dumps for the ROM and RAM respectively. Generating separate files allows the testbench to load the ROM and RAM contents separately.

```
riscv32-unknown-elf-objcopy
    --only-section=.init
    --only-section=.fini
    --only-section=.preinit_array
    --only-section=.init_array
    --only-section=.fini_array
    --only-section=.ctors
    --only-section=.dtors
    --only-section=.rodata
    --only-section=.text
    -O binary "<.elf>" "rom.bin"
```

Listing 34: ROM `objcopy` command

```
riscv32-unknown-elf-objcopy
    --only-section=.data
    --only-section=.tdata
    --only-section=.tbss
    --only-section=.tbss_space
    --only-section=.bss
    --only-section=.stack
    --only-section=.heap
    -O binary "$<.elf>" "ram.bin"
```

Listing 35: RAM `objcopy` command

Once the binary files are generated, they can be used to generate a file that contains the memory values for the ROM and RAM in a format that can be loaded in the RTL simulator testbench. For this work we require a **1-byte per line** format. The `xxd` tool is used to convert the binary files into a hex dump format. The `-p` option specifies the output format, and the `-c 1` option specifies that one byte should be printed per line[20]. The `xxd` command is shown in Listing 36. These resulting files can be loaded into the RTL simulator testbench.

```
xxd -p -c 1 rom.bin > rom.hex
xxd -p -c 1 ram.bin > ram.hex
```

Listing 36: xxd command to generate hex dump files for ROM and RAM

An example `CMakeLists.txt` file is shown in Appendix A.1. This file performs the following tasks:

- Compiles and links the Baremetal C workload using the `riscv32-unknown-elf-gcc` compiler.
- Executes the `objcopy` and `xxd` commands to generate the memory dumps.
- Executes the `objdump` command to generate disassembly files for the compiled binary.

4.5 Running Baremetal C workloads on the RV32I RTL

4.5.1 Workloads selected

The following workloads were selected to be compiled and run on the RV32I RTL. The `main` function of the startup code was modified to include the workload.

- **Dhrystone**[17]: A synthetic benchmark program that measures the performance of a system by simulating a small business application.
- **Qsort**[17]: A benchmark program that implements the quicksort algorithm to sort an array of integers.
- **CoreMark**[21]: An industry-standard benchmark that measures the performance of central processing units (CPUs) and embedded microcontrollers (MCU).

4.5.2 Loading memory dumps into the RTL simulator

With a supporting simulator the `$readmemh` system function can be used to load the memory dumps into the simulator. This function reads a hex file and loads the values into a memory array. The syntax for the `$readmemh` function is shown in Listing 37.

```
1 $readmemh("<memory dump file>", <array>);
2
3 initial begin
4   $readmemh("rom.hex", rom);
5   $readmemh("ram.hex", ram);
6 end
```

Listing 37: SV `$readmemh` function

4.5.3 Stopping the simulation when the workload finishes

The simulation can be stopped when the workload has finished executing. This is done by checking the program counter (PC) value against the linear address of the `_Exit` function. Listing 38 shows the code used to check the PC value. The `pc_exit` variable is defined in the testbench and contains the linear address of the `_Exit` function. The simulation will stop when the PC value matches the `pc_exit` value. This value can be passed as a command line argument to the simulator allowing the user to specify the exit address without modifying the testbench code.

```
int exit_pc;

// Capture the exit PC from the command line plusargs
initial
  $value$plusargs("EXIT_PC=%h", exit_pc);

// Check if the retired PC matches the exit PC
always @(negedge clk) begin
  RV32I_OPERAND_t retired_pc;
  logic retired;
  logic retired_pc_match_exit_pc;

  retired    = core.ROB_retire;
  retired_pc = core.ROB_packet.retired_pc;

  retired_pc_match_exit_pc = (retired_pc == exit_pc);

  // Exiting simulation if the retired PC matches the exit PC
  if ( retired_pc_match_exit_pc )
    $finish();
```

Listing 38: SV PC Check simulation finish logic

4.5.4 Measuring MPKI, IPC and cycles

Once the Baremetal C workloads are loaded into the RTL simulator, the simulation can be run to measure the performance of the workloads. The following metrics are measured:

- **MPKI (Mispredictions Per Kilo Instructions)**: The number of mispredicted branches per 1000 instructions executed. This metric is used to measure the performance of the branch predictor.

$$MPKI = \frac{\text{Mispredicted branches}}{\frac{\text{Instructions executed}}{1000}}$$

- **IPC (Instructions Per Cycle)**: The number of instructions executed per clock cycle. This metric is used to measure the performance of the CPU.

$$IPC = \frac{\text{Instructions executed}}{\text{Cycles}}$$

- **Cycles**: The total number of clock cycles taken to execute the workload.

In order to measure these metrics, the testbench must be configured to count the number of instructions executed, the mispredictions and the number of cycles taken. The following items can be used to measure these metrics. Listing 39 shows the code used to measure these metrics.

- **Instruction counter**: A counter that increments every time an instruction is retired.
- **Misprediction counter**: A counter that increments every time a branch is mispredicted.

- **Cycle counter**: A counter that increments every clock cycle.

```
real cycles = 0;
real instructions = 0;
real mispredicts = 0;

always @(negedge clk) begin
  RV32I_OPERAND_t retired_pc;
  logic retired;
  logic mispredict;

  retired    = core.ROB_retire;
  mispredict = core.ROB_JE_CLEAR;

  if (rst) begin
    cycles = 0;
    instructions = 0;
    mispredicts = 0;
  end else begin
    cycles++;
    instructions += retired;
    mispredicts  += mispredict;
  end
end

// Once simulation finishes we print the metrics
final begin
  real ipc;
  real mpki;

  $display("Cycles: %0.0f", cycles);
  $display("Instructions: %0.0f", instructions);
  $display("Mispredicts: %0.0f", mispredicts);

  ipc = instructions / cycles;
  mpki = mispredicts / (instructions / 1000);
  $display("IPC: %0.2f", ipc);
  $display("MPKI: %0.2f", mpki);
end
```

Listing 39: SV metrics collection logic

Appendix A

Appendix

A.1 Baremetal `CMakeLists.txt` example

Listing 40: Example `CMakeLists.txt` file for a bare-metal C project

```
cmake_minimum_required(VERSION 3.10)

# set the project name
project(baremetal_startup_c C)

# specify the C standard
set(CMAKE_C_FLAGS "\
  -march=rv32i \
  -mabi=ilp32 \
  -std=c99 \
  -O3 \
  -g \
  -Wall \
  -Wextra \
  -Wstrict-prototypes \
  -Wno-error=implicit-function-declaration \
  -Wno-error=implicit-int \
  -Wno-missing-prototypes \
  -Wno-unused-parameter \
  -Wno-unused-function \
  -ffunction-sections \
")

set ( STACK_SIZE 0x800 )
set ( HEAP_SIZE  0x800 )
set ( TARGET example_main )

# add the executable

add_executable(${TARGET}.elf startup.c ${TARGET}.c)
SET(LINKER_SCRIPT "${CMAKE_CURRENT_SOURCE_DIR}/linker.lds")

```

```
33 set_target_properties(${TARGET}.elf PROPERTIES LINK_DEPENDS
   ↪  "${LINKER_SCRIPT}")
34 target_include_directories(${TARGET}.elf PRIVATE ../include/ common/)
35
36 # Linker control
37 SET(CMAKE_EXE_LINKER_FLAGS  "${CMAKE_EXE_LINKER_FLAGS} -nostartfiles
   ↪  -Xlinker --defsym=__stack_size=${STACK_SIZE} -Xlinker
   ↪  --defsym=__heap_size=${HEAP_SIZE} -T ${LINKER_SCRIPT}
   ↪  -Wl,-Map=${TARGET}.map")
38
39 # Post processing command to create a disassembly file
40 add_custom_command(TARGET ${TARGET}.elf POST_BUILD
41         COMMAND ${CMAKE_OBJDUMP} -h -S  ${TARGET}.elf > ${TARGET}.disasm
42         COMMENT "Invoking: Disassemble")
43
44 # Post processing command to create a hex file
45 add_custom_command(TARGET ${TARGET}.elf POST_BUILD
46         COMMAND ${CMAKE_OBJCOPY} -O ihex  ${TARGET}.elf  ${TARGET}.hex
47         COMMENT "Invoking: Hexdump")
48
49 # Pre-processing command to create disassembly for each source file
50 foreach (SRC_MODULE ${TARGET} startup)
51   add_custom_command(TARGET ${TARGET}.elf
52                      PRE_LINK
53                      COMMAND ${CMAKE_OBJDUMP} -S
                        ↪  CMakeFiles/${TARGET}.elf.dir/${SRC_MODULE}.c.obj
                        ↪  > ${SRC_MODULE}.s
54                      COMMENT "Invoking: Disassemble (
                        ↪  CMakeFiles/${TARGET}.elf.dir/${SRC_MODULE}.c.obj)")
55 endforeach()
56
57 # Additional post-processing steps
58 add_custom_command(TARGET ${TARGET}.elf POST_BUILD
59         # Dump ROM sections
60         COMMAND riscv32-unknown-elf-objcopy
61             --only-section=.init
62             --only-section=.fini
63             --only-section=.preinit_array
64             --only-section=.init_array
65             --only-section=.fini_array
66             --only-section=.ctors
67             --only-section=.dtors
68             --only-section=.rodata
69             --only-section=.text
70             -O binary "${TARGET}.elf" "${TARGET}_rom.bin"
71
72         # Dump RAM sections
73         COMMAND riscv32-unknown-elf-objcopy
74             --only-section=.data
75             --only-section=.tdata
76             --only-section=.tbss
77             --only-section=.tbss_space
78             --only-section=.bss
79             --only-section=.stack
80             --only-section=.heap
```

```
81              -O binary "${TARGET}.elf" "${TARGET}_ram.bin"
82
83          COMMAND xxd -p -c 1 "${TARGET}_rom.bin" > "${TARGET}_rom.bintxt"
84          COMMAND xxd -p -c 1 "${TARGET}_ram.bin" > "${TARGET}_ram.bintxt"
85          COMMENT "Dumping ROM and RAM sections for ${TARGET}.elf")
86
87  SET(DCMAKE_EXPORT_COMPILE_COMMANDS ON)
```

Bibliography

[1] J. P. Shen and M. H. Lipasti, *Modern Processor Design: Fundamentals of Superscalar Processors*. McGraw-Hill, 2003, [Online]. [Online]. Available: https://books.google.com.mx/books?id=VIWLAAAACAAJ.

[2] S. Eyerman, J. E. Smith, and L. Eeckhout, "Characterizing the branch miss prediction penalty," in *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, Apr. 2006, pp. 48–58. DOI: 10.1109/ISPASS.2006.1620789.

[3] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 5th. Amsterdam: Morgan Kaufmann, 2012.

[4] A. Seznec and P. Michaud, "A case for (partially) TAgged GEometric history length branch prediction," *Journal of Instruction-Level Parallelism (JILP)*, vol. 8, Feb. 2006.

[5] AMD. "AMD 'Zen' Core Architecture," Accessed: Oct. 15, 2023. [Online]. Available: https://www.amd.com/en/technologies/zen-core.

[6] D. A. Jimenez and C. Lin, "Dynamic branch prediction with perceptrons," in *Proceedings of the HPCA Seventh International Symposium on High-Performance Computer Architecture*, 2001, pp. 197–206. DOI: 10.1109/HPCA.2001.903263.

[7] "Championship branch prediction," The Journal of Instruction-Level Parallelism, Accessed: Sep. 10, 2023. [Online]. Available: https://jilp.org/cbp2016/.

[8] T. Yeh and Y. Patt, "Alternative implementations of two-level adaptive branch prediction," *ISCA*, pp. 124–134, 1993.

[9] J. E. Smith, "A study of branch prediction strategies," *ISCA*, pp. 135–148, 1981.

[10] S. McFarling, "Combining branch predictors," 1993.

[11] T. Y. Y. Patt and M. Hwu, "Dynamic branch prediction strategies and mechanisms," *IEEE Transactions on Computers*, vol. 42, no. 4, pp. 396–408, 1993.

[12] J. Lee and A. J. Smith, "Branch prediction strategies and performance of pipelined architectures," *Computer*, pp. 6–22, 1984.

[13] T. Yeh and Y. Patt, "Two-level adaptive training branch prediction," *ISCA*, pp. 51–61, 1991.

[14] D. Patterson and J. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, 3rd. 2007.

[15] RISC-V Collaboration. “Risc-v gnu toolchain.” GNU toolchain for RISC-V, including GCC compiler and binutils, Accessed: May 17, 2023. [Online]. Available: https://github.com/riscv-collab/riscv-gnu-toolchain.

[16] Five Embed Dev. “Risc-v scratchpad.” GitHub repository with RISC-V development resources, Accessed: May 17, 2023. [Online]. Available: https://github.com/five-embeddev/riscv-scratchpad.

[17] RISC-V Software Sources. “Risc-v tests.” Test suite for RISC-V processors, Accessed: May 18, 2023. [Online]. Available: https://github.com/riscv-software-src/riscv-tests.

[18] GNU Project. “Gnu ld: Symbol versioning.” GNU linker documentation on symbol versioning, Accessed: Oct. 31, 2023. [Online]. Available: https://ftp.gnu.org/pub/old-gnu/Manuals/ld-2.9.1/html_node/ld_16.html#SEC16.

[19] J. Wilson. “All aboard, part 1: The -march, -mabi, and -mtune arguments to risc-v compilers.” SiFive blog post on RISC-V compiler arguments, Accessed: Oct. 31, 2023. [Online]. Available: https://www.sifive.com/blog/all-aboard-part-1-compiler-args.

[20] GNU Linux. “Xxd - make a hexdump or do the reverse.” Linux manual page for xxd command, Accessed: Oct. 31, 2023. [Online]. Available: https://linux.die.net/man/1/xxd.

[21] SiFive. “Coremark benchmark for risc-v.” GitHub repository with CoreMark benchmark implementation for RISC-V, Accessed: Nov. 15, 2023. [Online]. Available: https://github.com/sifive/benchmark-coremark.