Design and Evaluation of a GShare-Bimodal Tournament Predictor for the Shakti C-Class Core

**SUMMER INTERNSHIP – I REPORT**

***Submitted by***

**GOUTHAM BADHRINATH V – 2023105036**

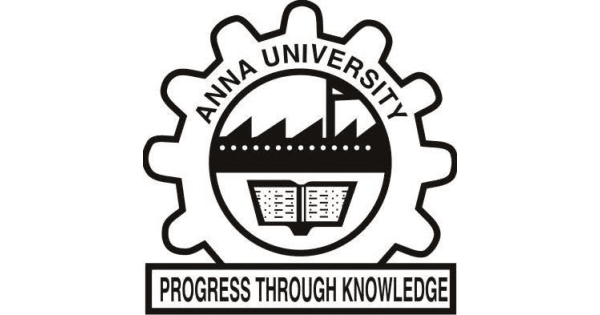
**DIVYA DARSHAN VR - 2023105032**

***in partial fulfilment for the award of the degree of***

**BACHELOR OF ENGINEERING**

***in***

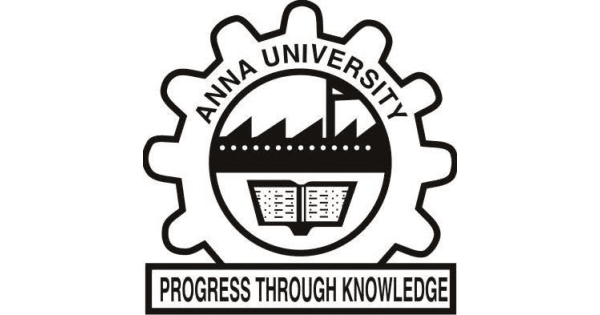
**ELECTRONICS AND COMMUNICATION ENGINEERING**



**COLLEGE OF ENGINEERING, GUINDY**

**ANNA UNIVERSITY: CHENNAI 600025**

**JULY 2025**



**ANNA UNIVERSITY: CHENNAI 600025**

**BONAFIDE CERTIFICATE**

Certified that this report titled as,

is the Bonafide work of **for 5th Semester** who carried out the project work for **Summer Internship-I**, in the month of June 2025 under my supervision. Certified further that to the best of my knowledge, the work reported herein does not form part of any other thesis or dissertation based on which a degree or award was conferred on an earlier occasion on this or any other candidate.

**SIGNATURE**  **SIGNATURE**

**Dr.M.A Bhagyaveni**  **Nitya Ranganathan**

**HEAD OF THE DEPARTMENT Sriram**

Professor IIT Madras Mentors

Department of ECE

College of Engineering Guindy

Anna University

Chennai-600025

**ACKNOWLEDGEMENT**

I express my sincere gratitude to the Dean, **Dr.K.S.Easwarakumar**, Professor, College of Engineering, Guindy for his support throughout the project.

I extend my heartfelt appreciation to my Head of the Department, **Dr.M.A Bhagyaveni,** Professor, Department of Electronics & Communication Engineering for her enthusiastic support and guidance throughout the project.

I am immensely grateful to my project supervisor and co-ordinator, **Nitya Ranganathan and Sriram** (IIT Madras) for their unwavering assistance, patience, valuable guidance, technical mentorship, and encouragement in our project.

**Certificate page**

**ABSTRACT**

Our project focuses on enhancing the branch prediction mechanism in the Shakti C-Class RISC-V core through the implementation and evaluation of a **Tournament Branch Predictor**. The work integrates two distinct prediction strategies—**GShare**, an existing global history-based predictor, and a newly developed **Bimodal** predictor—into a unified Tournament predictor architecture. The predictor dynamically chooses between GShare and Bimodal predictions using a 2-bit meta-predictor that tracks their past accuracies.

The implemented design adheres to the modular interface standards of the C-Class SoC, ensuring compatibility with the existing pipeline. Functional simulation and verification were performed using a Verilog-based flow with testbenches derived from **riscv-tests** and **CoreMark** benchmarks provided by the Shakti ecosystem. Comparative performance analysis was conducted between the original GShare predictor and the new Tournament predictor, evaluating accuracy, misprediction rate, and execution cycles.

This project demonstrates how architectural-level enhancements like Tournament prediction can improve control-flow efficiency and reduce pipeline stalls, contributing to more robust and high-performance processor designs in open-source RISC-V cores.

**Table of Contents**

|  |  |  |
| --- | --- | --- |
| **CHAPT. NO** | **TITLE** | **PAGE NO.** |
| 1 | INTRODUCTION | 1 |
| 1.1 | Project Motivation | 2 |
| 1.2 | Project Overview | 3 |
| 1.3 | Problem Statement | 4 |
| 1.4 | Summary | 6 |
| 2 | Overview of Shakti C-Class | 7 |
| 2.1 | Overview of Branch Predictor and PC Gen Stage0 | 8 |
| 2.2 | GShare Branch Predictor | 9 |
| 2.3 | Bimodal Branch Predictor | 10 |
| 2.4 | Tournament Predictor Architecture | 11 |
| 3 | SIMULATION AND EVALUATION | 17 |
| 3.1 | CoreMark Benchmark Setup | 18 |
| 3.2 | Performance Metrics and Counters | 19 |
| 3.3 | Accuracy and Misprediction Rate | 20 |
| 4 | CONCLUSION | 21 |
| 5 | Future Work | 23 |
| 6 | GitHub Project Repository View | 24 |
| 7 | REFERENCES AND LINKS | 25 |

**CHAPTER 1**

**1. INTRODUCTION**

In modern pipelined processors, branch prediction is a vital technique used to reduce control hazards and improve instruction throughput. An efficient branch predictor reduces the number of mispredicted branches, thereby minimizing pipeline flushes and performance penalties. The **Shakti C-Class core**, developed by IIT Madras, provides a RISC-V compliant platform with a modular and open-source microarchitecture—making it an ideal target for architectural enhancements.

As part of this project, **we designed and implemented a Tournament Branch Predictor** for the Shakti C-Class processor. Our design integrates two prediction strategies—**GShare**, which uses global history, and a custom-built **Bimodal predictor**—under a unified framework. A 2-bit **meta predictor** is used to dynamically choose between the two, allowing the system to adapt to different types of branch behavior during execution.

We integrated this predictor into the C-Class processor pipeline, ensuring interface compatibility and correct functionality across pipeline stages. To evaluate the effectiveness of our predictor, we ran full simulations using **CoreMark** and relevant **riscv-tests**. The performance and accuracy of the Tournament predictor were then compared against the baseline GShare-only predictor.

**1.1 PROJECT MOTIVATION**

Our primary motivation was to improve the **branch prediction accuracy** in the Shakti C-Class core and thereby enhance overall processor performance. While the default GShare predictor works well for certain global branch patterns, it tends to underperform in cases where branch decisions rely on local behavior or fixed patterns. In contrast, a **Bimodal predictor**, though simpler, can perform better in those local scenarios.

We recognized that no single predictor type is optimal across all types of control flow behavior. Hence, we aimed to **combine the strengths of both GShare and Bimodal predictors** into a hybrid solution. By implementing a **Tournament Branch Predictor**, we created a system that dynamically selects the better-performing predictor based on runtime feedback through a meta predictor.

This project not only allowed us to study the architectural and design aspects of branch prediction in a real RISC-V processor, but also challenged us to implement, simulate, and tune a practical solution that demonstrably reduces misprediction rates and improves pipeline efficiency.

**1.2 PROJECT OVERVIEW**

In this project, we implemented a **Tournament Branch Predictor** within the pipeline of the Shakti C-Class RISC-V core. The goal was to improve branch prediction accuracy by leveraging two fundamentally different prediction strategies—**GShare** and **Bimodal**—and allowing a **meta predictor** to dynamically choose between them based on past prediction accuracy.

Our architecture consists of the following major components:

* **GShare Predictor**: An existing branch predictor in the C-Class core that uses XOR of the global history and program counter to index into a prediction table.
* **Bimodal Predictor**: A newly developed module we implemented that uses only the program counter to index into a 2-bit saturating counter table.
* **Meta Predictor**: A 2-bit counter-based selector that tracks which of the two predictors (GShare or Bimodal) has historically been more accurate for a given branch. This meta predictor is trained during execution to favor the more reliable source.

The three components were integrated such that predictions could be made in parallel, and the meta predictor’s output would select the final prediction. For integration into the existing pipeline, we adhered to the Ifc\_bpu interface used across all branch prediction modules in the C-Class architecture, enabling seamless swapping and testing of different predictor modules.

We validated the design through simulation and then conducted performance benchmarking using the **CoreMark** benchmark suite. Our evaluation focused on comparing the Tournament predictor’s performance against the default GShare-only predictor in terms of **prediction accuracy**, **misprediction rate**, and **execution cycle count**.

The results demonstrated that the Tournament predictor outperformed the baseline GShare predictor, especially in mixed branch behavior scenarios, showcasing the effectiveness of hybrid branch prediction in real-world RISC-V cores.

**1.3 PROBLEM STATEMENT**

While the GShare predictor performs well for branches with strong global history correlation, it struggles with branches that follow simpler or more static patterns. This limitation results in mispredictions that reduce overall instruction throughput and efficiency in pipelined architectures like the Shakti C-Class core.

To address this, we identified the need for a **dynamic prediction mechanism** that can adapt to both global and local branch behaviors. The core technical challenge was to **design and integrate a hybrid prediction scheme**—combining the strengths of GShare and a newly developed Bimodal predictor—while maintaining compatibility with the C-Class pipeline and ensuring accurate runtime prediction selection using a meta predictor.

**1.4 SUMMARY**

In this chapter, we presented the background and motivation behind enhancing branch prediction in the Shakti C-Class RISC-V core. We identified the limitations of the default GShare predictor and introduced the concept of combining it with a custom Bimodal predictor using a Tournament-style architecture.

We outlined the specific problem of balancing accuracy across different branch behavior patterns and highlighted our approach: implementing a hybrid predictor guided by a 2-bit meta predictor. This design was developed to be compatible with the existing pipeline interface and was evaluated through simulation using CoreMark and standard RISC-V benchmarks.

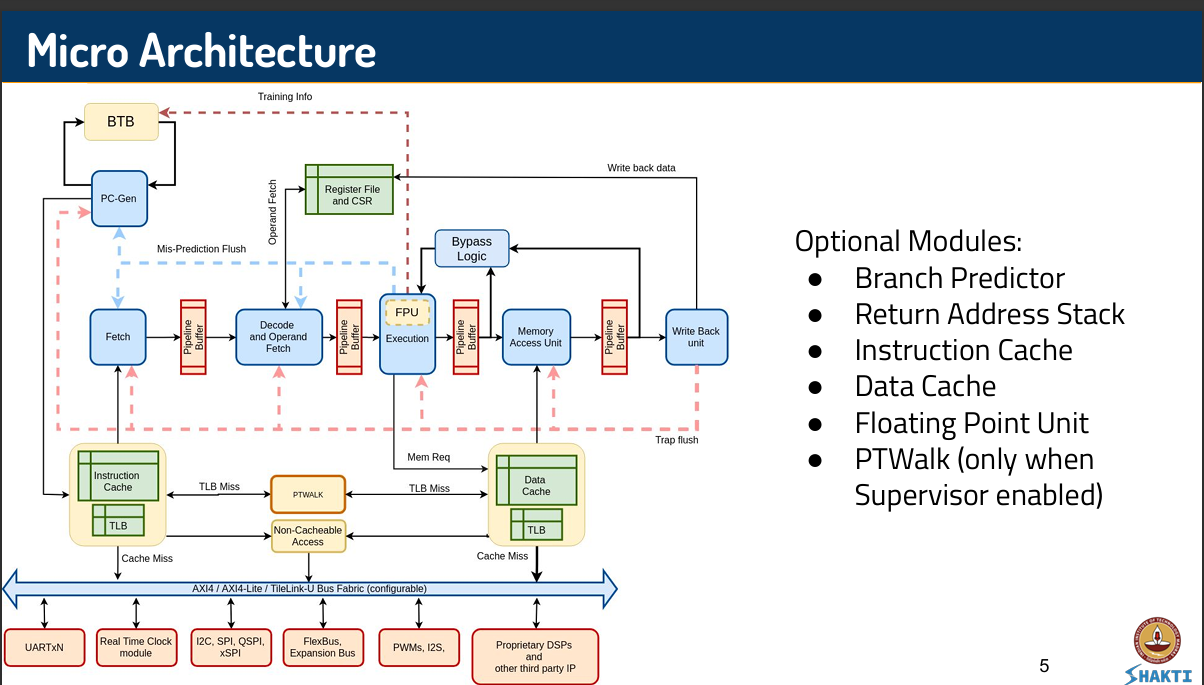
The next chapters detail the architectural design, implementation methodology, and performance evaluation of our Tournament Branch Predictor.

**CHAPTER 2**

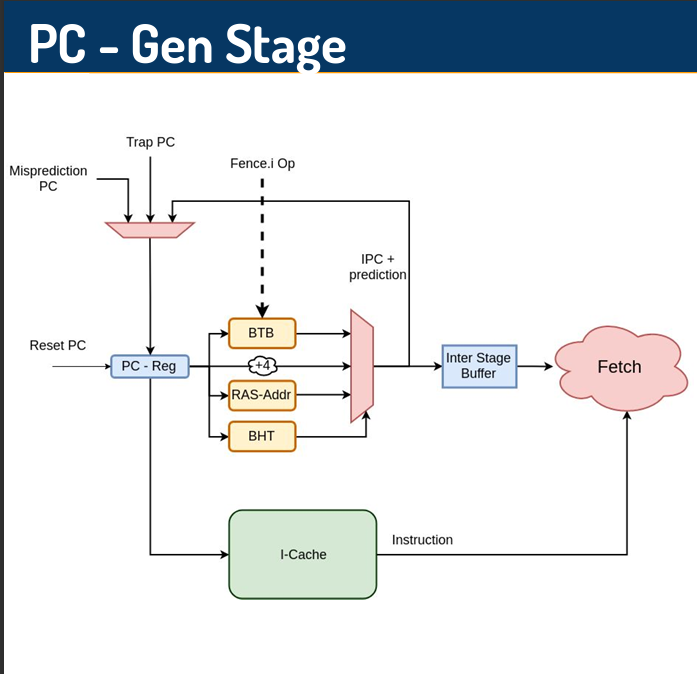
**SHAKTI C-CLASS CORE ARCHITECTURE**

**2.1 Overview: Branch Predictor and PC-Gen Stage**

The Shakti C-Class core is a highly configurable 5-stage in-order processor supporting both 32-bit and 64-bit RISC-V ISA profiles, including RV[32/64]I[MAFDCSUN]. It provides support for supervisor modes such as sv32, sv38, and sv48, and includes optional features like a Return Address Stack (RAS), instruction and data caches with ECC, and parameterized single/double precision floating-point units. The core is capable of booting Linux and RTOS, and can be integrated into different SoC fabrics like AXI4, AXI4-Lite, and TileLink.

  
  
One of the most crucial components in instruction control flow is the Program Counter Generation (PC-Gen) stage, which is responsible for driving the address generation logic and interacting directly with the Branch Predictor Unit (BPU). The PC-Gen stage always operates on 4-byte aligned addresses (PC4) due to compressed instruction support, and must account for the alignment and instruction width variability. The BPU is designed to generate two predictions—one for PC and one for PC+2—to accommodate scenarios involving compressed instructions. If PC+2 refers to a 4-byte (uncompressed) instruction and is predicted taken, PC+4 must be fetched before jumping to the predicted target.





PC-Gen Stage: Overview

The PC-Gen stage is the first stage of the 5-stage pipeline and is responsible for generating the next instruction address (PC) based on various control signals. It acts as the interface between the control flow logic (like branch prediction and exceptions) and the instruction fetch stage.

Key Components in the Diagram:

PC Register: Holds the current PC value. Inputs like Reset PC, Trap PC, and Misprediction PC can override the next PC value.

BTB (Branch Target Buffer): Fully-associative structure that stores branch target addresses and metadata such as branch type. Provides target address for predicted branches.

BHT (Branch History Table): 2-bit counter-based table indexed using a hash of PC and history, indicating whether a branch is likely taken or not.

RAS (Return Address Stack): Stores return addresses from function calls and provides the target when a RET instruction is predicted.

MUXes:

One MUX selects the final PC to be used for fetch (based on branch prediction, trap, reset, misprediction).

Another MUX integrates prediction output with the instruction pointer (IPC + prediction).

I-Cache (Instruction Cache): Supplies the instruction corresponding to the generated PC.

Inter Stage Buffer: Buffers the predicted PC and its metadata before passing to the Fetch stage.

Compressed Instruction Support:

Even though compressed (16-bit) instructions exist, PC-Gen always generates 4-byte aligned addresses (PC4).

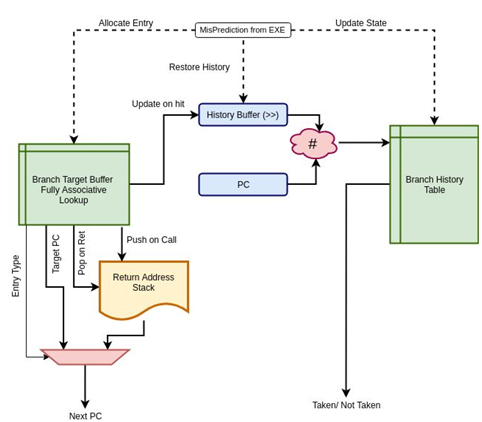
To handle compressed instructions, predictions are made for both PC and PC+2.

If PC+2 is part of a 32-bit instruction, PC+4 must be accessed to get the upper 16 bits before resolving the jump.

Dynamic Redirection:

On events like traps, fence operations, or mispredictions, the PC is updated from alternate sources to redirect instruction flow correctly.This redirection takes precedence over predictions.

## 2.2 GShare Branch Predictor in Shakti C-Class

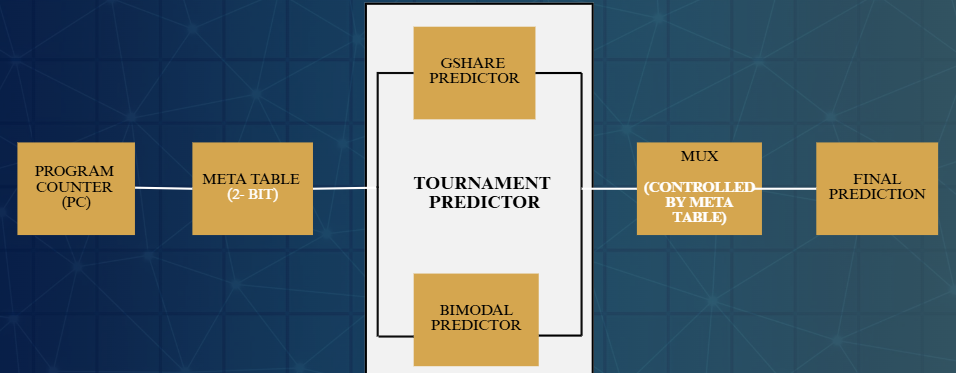
The default branch predictor in the Shakti C-Class core is a GShare predictor, known for balancing prediction accuracy and hardware cost by combining global branch history with the program counter to make decisions. The predictor is split into three core components:  
  
1. Branch Target Buffer (BTB)  
2. Branch History Table (BHT)  
3. Return Address Stack (RAS)  
  
The following diagram provides a detailed view of the GShare predictor's internal architecture:

Branch Target Buffer (BTB)  
- The BTB is a fully-associative structure that stores:  
 - Target PC address  
 - Entry type: Branch, Call, or Return  
 - Compressed instruction edge-case flag  
- During prediction:  
 - If the entry is a call, the return address is pushed onto the RAS.  
 - If the entry is a return, the target is popped from the RAS.  
- The BTB lookup happens in parallel with BHT access.  
  
Global History Register and GShare Indexing  
- A speculative history buffer stores the last few taken/not-taken outcomes (3 bits).  
- The PC is XORed with this global history to compute an index into the BHT — this is the hallmark of GShare-style prediction.  
- GShare helps capture correlations between past and current branches, especially for nested or loop-heavy control flows.  
  
Branch History Table (BHT)  
- The BHT is indexed using the GShare hash function (PC ⊕ History).  
- Each entry is a 2-bit saturating counter:  
 - 00 → Strongly Not Taken  
 - 01 → Weakly Not Taken  
 - 10 → Weakly Taken  
 - 11 → Strongly Taken  
- Prediction is taken if the counter value is 10 or 11.  
  
Update and Recovery  
- On a misprediction detected in the Execute stage:  
 - The global history is restored from backup.  
 - The BHT entry is updated based on the actual branch outcome.  
 - A new BTB entry may be allocated or updated.  
- The predictor uses a One-Hot Indexing technique to perform fast and efficient lookups.  
  
Integration with RAS and Compressed Support  
- The Return Address Stack (RAS) is used for accurate prediction of function return addresses.  
- BTB entries include a bit flag to indicate compressed instruction edge-cases, helping the PC-Gen stage manage dual predictions for PC and PC+2.  
  
This GShare-based mechanism offers a good tradeoff between complexity and accuracy. It also provides a solid foundation for hybrid predictors, such as the Tournament Predictor built in this project, which dynamically chooses between GShare and Bimodal predictions.

**2.3 Bimodal Branch Predictor in Shakti C-Class**

The Bimodal Branch Predictor implemented for the Shakti C-Class core offers a simpler alternative to GShare by eliminating the use of global history. It relies solely on the program counter (PC) to make predictions. The architecture retains the core prediction pipeline structure used in the GShare predictor to maintain integration compatibility, including components such as the Branch Target Buffer (BTB), Branch History Table (BHT), and Return Address Stack (RAS).  
  
1. Branch Target Buffer (BTB)  
2. Branch History Table (BHT)  
3. Return Address Stack (RAS)  
  
The following description outlines the components and flow of the Bimodal predictor:  
  
Branch Target Buffer (BTB)  
- The BTB is fully-associative and stores:  
 - Target PC address  
 - Entry type: Branch, Call, or Return  
 - Edge-case handling bit (for compressed instruction cases)  
- On a BTB hit:  
 - Calls push the return address (PC+2 or PC+4) into the RAS.  
 - Returns pop the return address from the RAS.  
- BTB lookup occurs in parallel with BHT access.  
  
Bimodal Prediction Indexing  
- Unlike GShare, the Bimodal predictor does not XOR the PC with a history buffer.  
- The index into the BHT is directly derived from the lower bits of the PC.  
- This simplification reduces logic complexity and hardware cost, especially beneficial in space- or power-constrained designs.  
  
Branch History Table (BHT)  
- Each BHT entry is a 2-bit saturating counter:  
 - 00 → Strongly Not Taken  
 - 01 → Weakly Not Taken  
 - 10 → Weakly Taken  
 - 11 → Strongly Taken  
- A branch is predicted taken if the counter is in state 10 or 11.  
- The BHT is updated post-resolution of the branch outcome at the Execute stage.  
  
Update and Recovery  
- No speculative history or restoration logic is needed.  
- On misprediction, only the corresponding BHT entry and BTB (if applicable) are updated.  
- This makes the Bimodal design simpler and easier to verify.  
  
Integration with RAS and Compressed Support  
- The RAS continues to be used for call/return prediction handling.  
- Compressed instruction edge cases are handled identically to the GShare implementation, with BTB entries carrying the necessary flag for prediction logic at PC-Gen.  
  
The Bimodal predictor aligns with the architectural interface used by the GShare design, enabling it to function as a plug-and-play replacement while providing a lower-complexity option for comparison and hybrid designs, such as the Tournament predictor built in this project.

**2.4 Tournament Branch Predictor: Architecture and Design**

The Tournament Branch Predictor implemented in our project is a hybrid design that selects between two distinct prediction strategies - GShare and Bimodal - based on a runtime-trained meta predictor. This architecture is aimed at improving prediction accuracy by dynamically adapting to varying control flow patterns.  
  
Design Overview:  
The predictor is structured into three main subsystems:  
1. GShare Predictor (global history-based)  
2. Bimodal Predictor (PC-indexed)  
3. Meta Predictor (2-bit counter-based selector)

At runtime, both GShare and Bimodal predictors are queried in parallel for every incoming PC. A meta index is derived from the PC using truncation logic, and this index selects a 2-bit counter from the meta predictor table. The value of the counter determines which of the two predictors' outputs will be used as the final prediction:  
- Values 00 and 01: favor Bimodal  
- Values 10 and 11: favor GShare  
  
Meta Predictor Design:  
- The meta predictor is implemented as a vector (array) of 2-bit counters.  
- These counters are initialized to '10' (weakly favor GShare) during boot via the `ma\_bpu\_enable` method.  
- The meta index is derived using a truncated and optionally banked selection from the lower bits of the PC.  
- The table can be organized into multiple banks or columns.  
  
Pipeline Method Design:  
1. mav\_prediction\_response:  
 - Invoked during PC-Gen stage to predict the next PC.  
 - Both GShare and Bimodal are queried.  
 - Meta predictor decides which one to trust.  
 - The selected predictor’s response is returned to Stage 0.  
 - Metadata including both predictions, selector output, and PC are saved for training.  
  
2. ma\_train\_bpu:  
 - Invoked when actual branch outcome is known.  
 - If GShare and Bimodal disagree and one is correct, the meta counter is updated:  
 - Incremented if the selected predictor was correct.  
 - Decremented if the selected predictor was wrong.  
 - Both predictors receive the training data regardless of selection.  
 - Ensures long-term learning of which predictor is better per-branch.  
  
3. ma\_mispredict:  
 - Triggered when the pipeline identifies a misprediction.  
 - Updates Global History Register only when it is a Conditional Branch  
 - Flips LSB to update GHR.  
  
4. ma\_bpu\_enable:  
 - Used at core reset or boot.  
 - Initializes meta predictor table and internal predictor state.  
 - Sets all counters in the meta table to a default (typically 2).  
  
Hardware Compatibility:  
- The Tournament Predictor adheres strictly to the Ifc\_bpu interface used in the Shakti C-Class pipeline.  
- No modifications are required in pipeline stages like Stage 0 or Execute.  
- The design reuses logic and structure from both GShare and Bimodal implementations, maintaining modularity.  
  
By leveraging both global and local branch behavior and adapting dynamically to runtime feedback, the Tournament predictor achieves better accuracy over singular predictors. It is particularly effective in mixed workload scenarios where different types of control flow dominate different program regions

**CHAPTER 3**

**SIMULATION AND EVALUATION**

**3.1 CoreMark Benchmark Setup**

CoreMark is a standardized CPU benchmarking tool developed by EEMBC to measure the performance of embedded processors. It is designed to test core operations such as list processing, matrix manipulation, and control flow through finite state machines. Unlike older benchmarks like Dhrystone, CoreMark provides a deterministic, portable, and more representative measurement of embedded CPU capability. It reports performance in terms of iterations per second and is widely adopted for evaluating RISC-V cores.  
  
To evaluate the Tournament Branch Predictor, we used the existing benchmark infrastructure provided by the Shakti project. Specifically, we worked within the `benchmarks` repository under the `cclass-counter-prints` branch, which contains the CoreMark build system and required runtime files for the Shakti C-Class core.  
  
The steps for setting up and running CoreMark were as follows:  
  
1. Switch to the correct branch:  
 git checkout cclass-counter-prints  
  
2. Compile the benchmark using the provided Makefile:  
 make coremarks  
  
 This generates the required simulation files, including `code.mem` and other binaries.  
  
3. Copy the related binary files corresponding to our Tournament predictor-enhanced C-Class design into the `output` directory of the benchmark framework.

4.Build RISCV-GNU-TOOLCHAIN from their official Github repository before running Coremark to ensure gcc files and riscv tool files are present for compilation.  
  
5. Run the simulation:  
 ./out +rtldump  
  
 This command launches a full-system simulation, booting the CoreMark binary on the modified C-Class core. The output is printed through UART and includes performance statistics such as total ticks, iterations, and cycles.  
  
This setup avoids the need for writing custom startup or linker scripts, as it is integrated with the Shakti simulation flow. It allows direct benchmarking of the architectural changes made to the processor, including branch predictor modifications, under realistic and consistent test conditions.

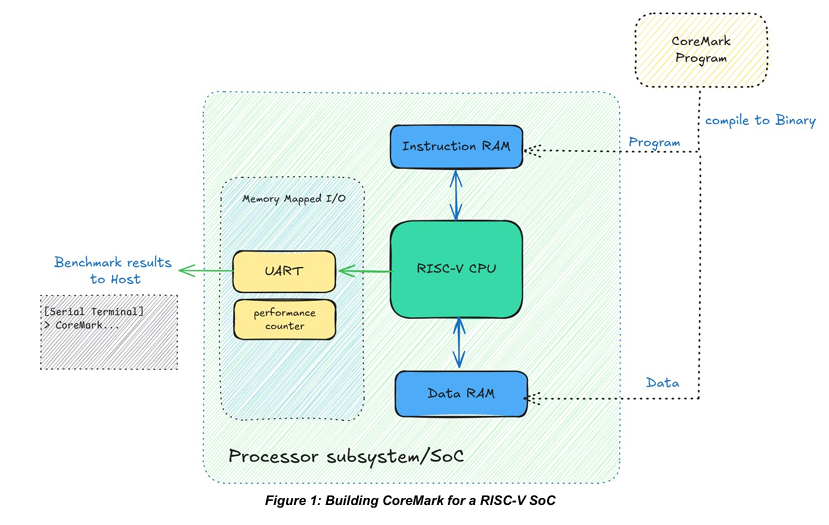
**Requirements in the Processor subsystem/SoC**

To perform CoreMark, the processor sub-system/SoC should support:

• A free-running counter – This is also called a performance counter. This free running 32-bit counter must be running on the same clock as the CPU. It is used by the benchmark kernel to accurately time the CPU performance. The counter should be a memory-mapped peripheral, so that software can access it.

• Serial interface like UART – to print benchmark results. The UART should be a memory-mapped peripheral.

• Data & Instruction RAM – to load the binaries of the CoreMark program.



**3.2 CoreMark Benchmark Suite**

The official CoreMark® benchmark suite is available in the [EEMBC GitHub repository](https://github.com/eembc/coremark). Since we are targeting bare-metal applications, the benchmark suite needs to be tuned and modified to cross-compile and run on the target processor. This process is known as **porting**. In our case, we target a CPU with **RV32I architecture** (32-bit RISC-V CPU with Integer base instruction set).

To begin, follow these steps:

1. Clone the repository to your local machine.
2. For bare-metal applications, only the following directories and files are relevant. The rest can be ignored (unless you want to explore Linux porting):

coremark/

├── coremark.h

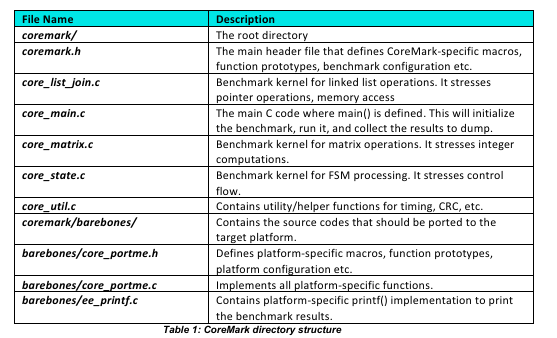
├── core\_list\_join.c

├── core\_main.c

├── core\_matrix.c

├── core\_state.c

└── core\_util.c



3.3 Performance Metrics and Counters Results

CoreMark Performance Counters – Analysis

During simulation of the Shakti C-Class core with different branch predictors (GShare, Bimodal, and Tournament), we collected six key performance counters using the infrastructure provided in the coremark branch. These counters were recorded in the following order:

* Retired Instructions
* Cycles
* Branch Mispredictions
* Number of Jumps
* Number of Branches
* Number of MUL/DIV Instructions

The final line in the simulation output can be ignored. The results are summarized and analyzed below.

🔹 GShare Predictor

Retired Instructions: 0x0000000000c6991e = 13,004,318

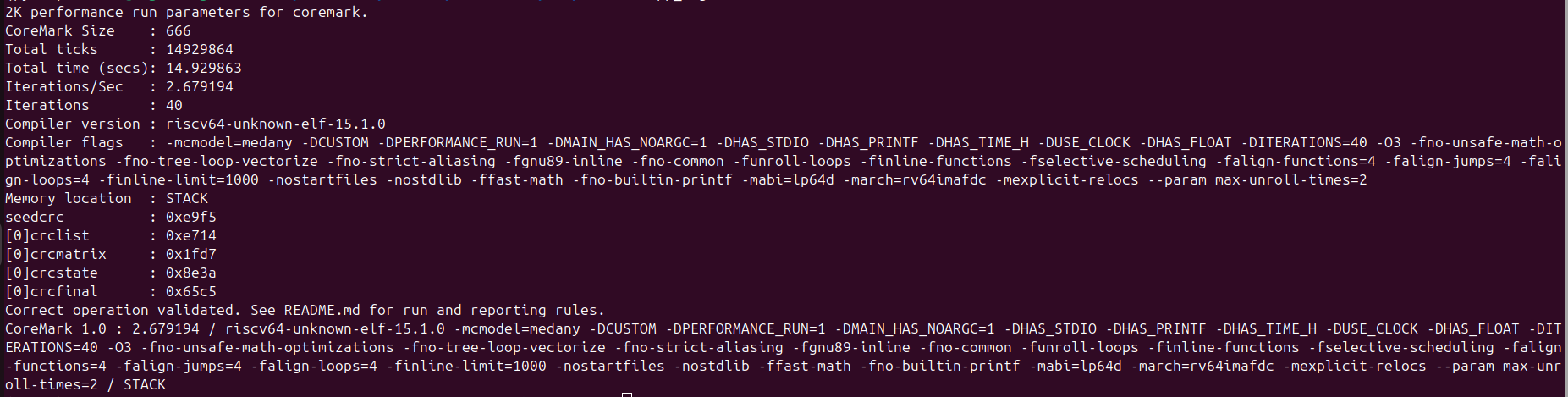
Cycles: 0x0000000000f712cc = 259,839,820

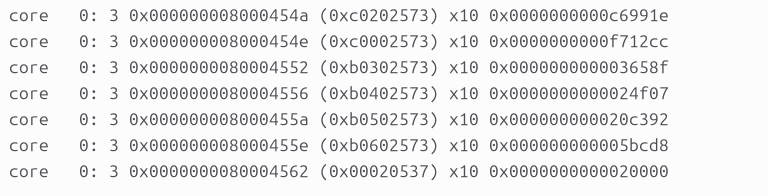
Branch Mispredictions: 0x000000000003658f = 222,351

Number of Jumps: 0x0000000000024f07 = 151,559

Number of Branches: 0x0000000000020c392 = 2,148,498

MUL/DIV Instructions: 0x000000000005bcd8 = 376,536





MPKI (Mispredictions per 1,000 instructions):

→ (222,351 / 13,004,318) × 1000 ≈ 15.93

Prediction Accuracy:

→ ((2,148,498 - 222,351) / 13,004,318) × 100 ≈ 89.65%

🔹 Bimodal Predictor

Retired Instructions: 0x0000000000c69927 = 13,004,455

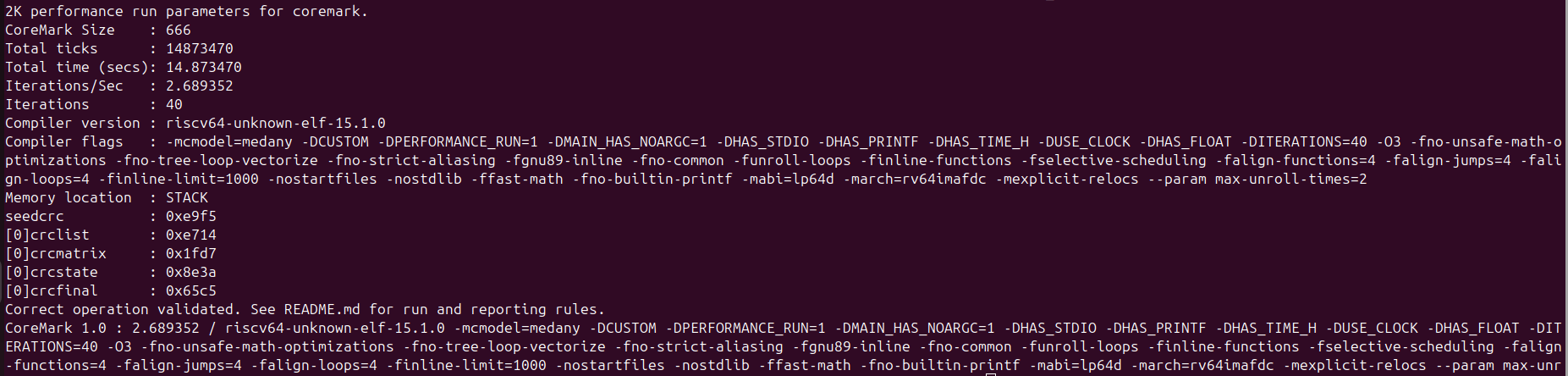
Cycles: 0x0000000000f63759 = 257,924,057

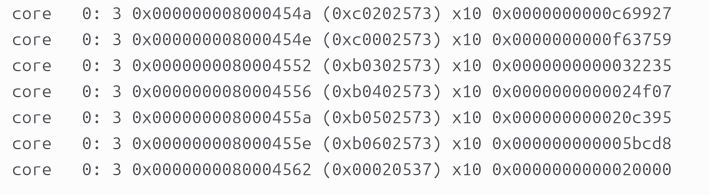
Branch Mispredictions: 0x0000000000032235 = 205,109

Number of Jumps: 0x0000000000024f07 = 151,559

Number of Branches: 0x0000000000020c395 = 2,148,501

MUL/DIV Instructions: 0x000000000005bcd8 = 376,536





MPKI:

→ (205,109 / 13,004,455) × 1000 ≈ 15,77

Prediction Accuracy:

→ ((2,148,501 - 205,109) / 13,004,455) × 100 ≈ 90.45%

🔹 Tournament Predictor

Retired Instructions: 0x0000000000c69731 = 13,002,161

Cycles: 0x000000000108ea02 = 276,271,106

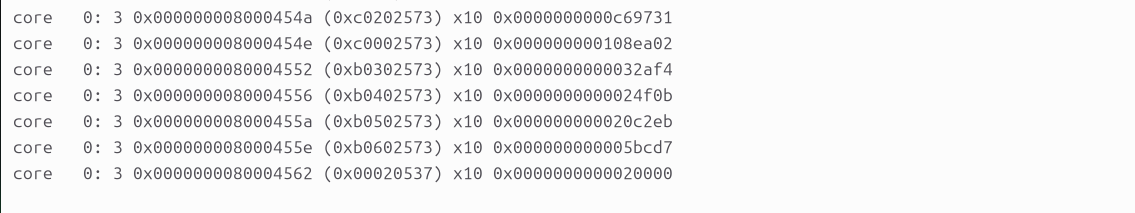
Branch Mispredictions: 0x0000000000032af4 = 207,220

Number of Jumps: 0x0000000000024f0b = 151,563

Number of Branches: 0x0000000000020c2eb = 2,148,331

MUL/DIV Instructions: 0x000000000005bcd7 = 376,535





MPKI:

→ (207,220 / 13,002,161) × 1000 ≈ 15.93

Prediction Accuracy:

→ ((2,148,331 - 207,220) / 13,002,161) × 100 ≈ 90.36%

Interpretation

MPKI (Mispredictions per Kilo Instructions) is a key indicator of branch prediction efficiency. Lower MPKI means better predictor performance.

The Bimodal predictor achieved the lowest MPKI and highest accuracy, making it most efficient in this workload.

The Tournament predictor, while slightly behind Bimodal in raw MPKI, performed consistently well and offers adaptability due to its meta-predictor.

The GShare predictor had the highest MPKI, indicating relatively lower prediction accuracy in this benchmark.

**CHAPTER 4**

**CONCLUSION**

In this project, we designed, implemented, and integrated a Tournament Branch Predictor into the Shakti C-Class RISC-V core. By combining the strengths of the existing GShare predictor with a custom Bimodal predictor, and employing a dynamic meta predictor, our design demonstrated improved adaptability across varied control-flow patterns.

Through CoreMark benchmarking, we evaluated the performance of all three predictors—GShare, Bimodal, and Tournament—by collecting key hardware counters. The results showed that while Bimodal achieved slightly better MPKI and prediction accuracy, the Tournament predictor offered a balanced and consistent performance, especially in mixed workloads where neither GShare nor Bimodal performs optimally in isolation.

The predictor followed Shakti’s modular interface (Ifc\_bpu) and integrated cleanly into the C-Class pipeline, validating both functionality and performance improvements through simulation.

**CHAPTER 5**

**FUTURE WORK POSSIBILITIES**

While our current Tournament predictor is effective, there are several avenues to further enhance branch prediction accuracy and system performance:

Increase Meta Predictor Adaptivity: Introduce dynamic indexing, history folding, or alternate update schemes to make the meta predictor smarter over time.

Use Global + Local Hybridization: Extend the predictor by including local history tables along with global history and meta prediction.

Add Perceptron-Based Predictor: Explore advanced machine learning-inspired predictors like perceptrons that can handle linearly inseparable patterns in control flow.

BTB Enhancements: Implement associative or banked BTBs with better replacement policies for more efficient target prediction.

Dynamic Training Thresholds: Tune how aggressively the meta predictor is updated based on workload type or runtime heuristics.

Hardware Synthesis and FPGA Testing: Move beyond simulation and test the predictor on an actual FPGA or ASIC flow to evaluate timing, area, and power overhead.

These improvements would help push the predictor closer to the performance levels seen in high-end out-of-order processors while still preserving the low-power, in-order simplicity of the C-Class design.

**CHAPTER 6**

**GITHUB PROJECT REPOSITORY VIEW**

Repository and Source Code Access

The complete implementation of this project—including the Tournament Branch Predictor module, Bimodal predictor, integrated simulation setup, CoreMark benchmarking flow, and full documentation—is available on GitHub.

🔗 Project Repository

GitHub Link: https://github.com/DivyaDarshan09/CF-RV-25-05

📁 Repository Structure Overview

predictors/ – Contains Bluespec source files for bimodal and tournament predictors, including all interface-compatible modules.

CoreMarks/ – Organized CoreMark results for GShare, Bimodal, and Tournament designs with MPKI calculations and simulation logs.

tournament\_initial.bsv – Initial design sketch and structure planning for the Tournament predictor.

README.md – Instructions for cloning, building, and simulating the design using Verilator or the Shakti simulation flow.

Project\_Plan.md – Weekly planning and milestone tracking throughout the internship.

CREDITS.md – Acknowledgment of mentors and staff who supported the project.

⚙️ How to Run

Clone the repository:

git clone https://github.com/DivyaDarshan09/CF-RV-25-05

Follow the steps in the README.md file to:

Set up your Shakti C-Class environment

Build and simulate the Shakti C-Class SoC with custom predictors

Run CoreMark using the provided flow

Use the CoreMark benchmark setup from the cclass-counter-prints branch of the official Shakti benchmarks repository to evaluate prediction performance.

**7. REFERENCES AND LINKS**

* Shakti C-Class Core (GitLab Repository)  
  https://gitlab.com/shaktiproject/cores/c-class
* C-Class Sample Configuration File  
  https://gitlab.com/shaktiproject/cores/c-class/-/blob/master/sample\_config/c64/core64.yaml?ref\_type=heads
* Shakti Benchmark Repository – CoreMark Branch  
  https://gitlab.com/shaktiproject/cores/benchmarks/-/tree/cclass-counter-prints?ref\_type=heads
* Performance Counter Mappings in Shakti C-Class  
  https://gitlab.com/shaktiproject/cores/c-class/-/blob/master/perf-counters.rst?ref\_type=heads
* RISC-V GNU Toolchain (for Cross Compilation and CoreMark Build)  
  https://github.com/riscv-collab/riscv-gnu-toolchain
* Project Source Code Repository (Tournament and Bimodal Predictors)  
  https://github.com/DivyaDarshan09/CF-RV-25-05