Performance Evaluation of Stage0 Variants in C-Class Core RTL

CEG Fabless RISC-V Summer Internship 2025 Report

Submitted by

ABIELA MARIA Y - 2022105503

SWETHA V - 2022105013

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

Performance Evaluation of Stage0 Variants in C-Class Core RTL

CEG Fabless RISC-V Summer Internship 2025 Report

Submitted by

ABIELA MARIA Y - 2022105503

SWETHA V - 2022105013

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, "Performance Evaluation of Stage 0 Variants in C-Class Core RTL" is the Bonafide work of ABIELA MARIA Y(2022105503) & SWETHA V (2022105013)who carried out the project work for Summer Internship-I, in the month of July 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.Nitya Ranganathan	**Mr. Sriram**
SHAKTHI Team,	**SHAKTHI Team,**
Indian Institute of Technology,Madras	**Indian InstituteTechnology,Madras**
Chennai-600036	**Chennai-600036**

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 project I am immensely grateful to my project supervisors Dr. Nitya Ranganathan and Mr. Sriram from the SHAKTI Team, for her unwavering assistance, patience, valuable guidance, technical mentorship, and encouragement in my project. I would like to thank the entire SHAKTI Team for their unwavering support in helping us throughout this project. I would also like to sincerely thank Mr. Sivakumar Anandan, alumnus of the Department of Electronics and Communication Engineering, College of Engineering Guindy, for bringing us this opportunity and for his continuous encouragement and support that made this project possible.

Certificate page

ABSTRACT

C-Class is a configurable 6-stage in-order processor core developed as part of the SHAKTI processor family. It supports the RV64GCSUN ISA and is designed for a wide range of applications, from embedded systems to Linux-based platforms. Its open-source nature and flexibility make it suitable for low-cost, high-performance computing. This project focuses on the performance analysis of Stage 0, which includes the Program Counter Generator (PCGen) and the Branch Predictor—two critical components that influence instruction flow and pipeline efficiency. Accurate branch prediction and efficient PC generation can significantly reduce stalls and improve overall processor performance. The C-Class core was successfully built using the default configuration files provided by the Shakti team. Benchmarks such as CoreMark, RISC-V standard test suites were executed using the Spike simulator. Performance metrics including CPI, instruction count, and cycle count were extracted from simulation logs, particularly the rtl1.dump file. The next phase involves modifying key parameters like the reset PC and branch predictor type in the core64.yaml file to study their impact on performance. Comparing different configurations will help identify optimal settings that enhance Stage 0 efficiency. This analysis is essential to understand how early pipeline components affect overall system behavior.The findings can contribute to improving the configurability and performance scalability of open-source cores like C-Class.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

1.1 PROJECT OVERVIEW

The SHAKTI C-Class core is a 6-stage, in-order, highly configurable RISC-V processor developed as part of the open-source SHAKTI processor initiative. It supports the RV64GCSUN ISA extensions and is suitable for a wide range of applications including embedded systems and even Linux-based networking and industrial systems. The design supports generating various customized core instances using YAML-based parameter files, making it ideal for academic research and commercial prototyping. In this project, we specifically focus on analyzing the performance of Stage 0 of the C-Class pipeline, which comprises the **Program Counter Generator (PCGen)** and the **Branch Predictor**. These components are responsible for efficient instruction fetch and control flow prediction, both of which directly influence pipeline efficiency and execution performance.

1.2 PROBLEM STATEMENT

Although the C-Class core offers a high degree of configurability, the performance impact of different parameter settings—particularly within Stage 0—has not been extensively explored. The default configuration may not be optimal for all use cases, especially when considering varying workloads or application domains. Stage 0 plays a critical role in determining how well the instruction fetch stage performs and how accurately the processor handles branches. Inefficient PC generation or poor branch prediction can lead to pipeline stalls, increased cycle counts, and reduced throughput. Therefore, there is a need to evaluate how modifications in Stage 0 parameters such as the **reset PC address** and **branch predictor type** affect the overall performance of the core. This study aims to fill that gap by methodically analyzing the performance changes due to configuration tuning.

1.3 OBJECTIVES

The primary objective of this project is

- To understand and analyze Stage 0 of the Shakti C-Class core, specifically focusing on the Program Counter Generator (PCGen) and Branch Predictor.
- To evaluate the performance of the core using standard benchmarks like CoreMark, RISC-V test suites, and custom microbenchmarks provided by the Shakti team.
- To identify and modify configuration knobs in the core64.yaml file related to reset PC and branch prediction logic.
- To measure performance metrics such as Cycles Per Instruction (CPI), instruction count, and execution cycles using simulation output files (e.g., rtl1.dump).
- To compare performance across different configurations, and determine the impact of each change on Stage 0 behavior and overall execution efficiency.

1.4 SUMMARY

This project explores the performance tuning possibilities of the Shakti C-Class processor by focusing on Stage 0, which includes PCGen and the Branch Predictor. By modifying configuration knobs and evaluating their impact through benchmarks, the project provides valuable insights into the microarchitectural behavior of the processor. The systematic comparison of multiple configurations allows us to understand how small changes in early pipeline stages influence overall performance. This work emphasizes the importance of configuration-aware design and testing in RISC-V cores, especially in open-source environments where flexibility is a key feature. The results from this analysis can help improve core design decisions, guide future enhancements, and serve as a reference for similar architecture research and performance studies.

CHAPTER 2

DESIGN OVERVIEW

2.1 BUILDING THE C-CLASS CORE

The Shakti C-Class core is a 64-bit in-order RISC-V processor developed as part of the Shakti processor family. The process of building this core includes setting up the environment, installing dependencies, configuring the SoC, and generating the necessary Verilog files and binaries for simulation and testing**.**

The steps involved in building the Shakti C-Class core are as follows:

1. Cloning the Repository

The official Shakti C-Class repository is cloned using the following commands:

```
git clone https://gitlab.com/shaktiproject/cores/c-class.git

cd c-class
```

2. Installing Dependencies

All required Python dependencies are installed using the pip package manager:

```
pip install -r requirements.txt
```

3. Managing SoC Dependencies

The tool repomanager is used to clean and update the required SoC dependencies. The commands are**:**

```
repomanager --yaml $PWD/test_soc/c64_c32/c64_deps.yaml --verbose debug --clean

repomanager --yaml $PWD/test_soc/c64_c32/c64_deps.yaml --verbose debug -cup
```

4. Generating SoC Configuration

The SoC configuration is generated using the soc_config tool by specifying the required YAML files for ISA, custom instructions, core configuration, CSR grouping, and debug. The command used is:

```
soc_config -ispec sample_config/c64/rv64i_isa.yaml -customspec
sample_config/c64/rv64i_custom.yaml -cspec sample_config/c64/core64.yaml -
gspec sample_config/c64/csr_grouping64.yaml -dspec
sample_config/c64/rv64i_debug.yaml --verbose debug
```

5. Setting the Environment Variable

Before starting the build process, the XXD version environment variable is set as follows:

```
export XXD_VERSION=2023
```

6. Building the Core

The following make commands are executed sequentially to generate the Verilog files, link the Verilator simulation environment, and generate boot files:

```
make generate_verilog

make link_verilator

make generate_boot_files
```

7. Accessing the Build Output

After successful execution of the build commands:

- The executable binaries can be found at: ./c-class/bin
- The generated Verilog RTL files are available at: /home/vsysuser/workspace/c-class/build/hw/verilog

8. Exporting the Build Environment

The built design's home path is exported for simulation and test execution:

```
export
DESIGN_HOME=/home/vsysuser/workspace/uptickpro_examples/shakti_cclass/c-
class/bin
```

Running Tests on C-Class

After the build is complete, test programs can be run on the C-Class processor core to verify its functionality.

1. Navigate to the test directory:

```
cd ./../riscv_assembly
```

2. Export the design path again, if necessary:

```
export
DESIGN_HOME=/home/vsysuser/workspace/uptickpro_examples/shakti_cclass/c-
class/bin
```

3. Run a test using the make run_dut command. Replace <path/to/test> with the relative path to the test file:

```
make run_dut TEST=<path/to/test>
```

For example:

```
make run_dut TEST=tests/branch.S
```

This completes the build and test setup for the Shakti C-Class processor core**.**

2.2 INSTALLING AND BUILDING THE BLUESPEC COMPILER (BSC)

As a next step in our setup, we installed and built the Bluespec SystemVerilog Compiler (BSC). The BSC toolchain requires the Glasgow Haskell Compiler (GHC), several additional Haskell libraries, and standard development tools.

1. Installing GHC and Required Haskell Libraries

On Ubuntu-based systems, the Haskell compiler `ghc` and required libraries can be installed using:

```
sudo apt-get install ghc
sudo apt-get install libghc-regex-compat-dev libghc-syb-dev libghc-old-time-
dev libghc-split-dev
```

If profiling builds are needed, profiling versions of these libraries can also be installed:

```
sudo apt-get install ghc-prof libghc-regex-compat-prof libghc-syb-prof
libghc-old-time-prof libghc-split-prof
```

These packages are typically available through most Linux distributions' package managers.

2. Installing Additional System Requirements

To build and run BSC along with its associated tools, the following additional packages are needed:

```
sudo apt-get install tcl-dev build-essential pkg-config autoconf gperf flex
bison iverilog
```

For building PDF documentation from LaTeX sources:

```
sudo apt-get install texlive-latex-base texlive-latex-recommended texlive-
latex-extra texlive-font-utils texlive-fonts-extra
```

3. Cloning the BSC Repository

To obtain the BSC source code, clone the GitHub repository with submodules:

```
git clone --recursive https://github.com/B-Lang-org/bsc
```

If the repository was cloned without the `--recursive` flag, the submodules can be initialized later with:

```
git submodule update --init --recursive
```

4. Building the BSC Toolchain

To build BSC from the root of the cloned repository:

```
make install-src
```

This will create an `inst` directory containing the BSC compiler toolchain. If you wish to install to a different location, you can specify a custom prefix:

```
ruby
make PREFIX=/opt/tools/bsc/bsc-<VERSION>
```

2.3. INSTALLING AND BUILDING THE RISC-V GNU COMPILER TOOLCHAIN

1. Getting the Sources

Clone the official RISC-V GNU Toolchain repository:

git clone https://github.com/riscv/riscv-gnu-toolchain

Several standard packages are required for building the toolchain. On **Ubuntu**, run:

```
sudo apt-get install autoconf automake autotools-dev curl python3 python3-pip python3-tomli \
libmpc-dev libmpfr-dev libgmp-dev gawk build-essential bison flex texinfo gperf libtool \
patchutils bc zlib1g-dev libexpat-dev ninja-build git cmake libglib2.0-dev libslirp-dev
```

3. Installation (Newlib Build)

Choose an installation path .Example: $HOME/riscv.

1.Configure the build:

```
./configure --prefix=$HOME/riscv --with-arch=rv64imafdc --with-abi=lp64d
```

2.Build the Newlib cross-compiler:

```
make -j4 newlib
```

3.Update PATH (add to ~/.bashrc or export in terminal):

```
export PATH=$HOME/riscv/bin:$PATH
```

CHAPTER 3

CONFIGURATION OVERVIEW

3.1 STAGE-0 CONFIGURATION

Instruction Stream Buffer (ISB) Sizes

The Instruction Stream Buffer (ISB) provides temporary storage between successive pipeline stages to smoothen the flow of instructions. Larger buffers can tolerate higher latencies, while smaller buffers improve area efficiency but may cause stalls.

In the current configuration:

- ISB (Stage-0 → Stage-1): 2 entries
- ISB (Stage-1 → Stage-2): 2 entries
- ISB (Stage-2 → Stage-3): 1 entry
- ISB (Stage-3 → Stage-4): 8 entries
- ISB (Stage-4 → Stage-5): 8 entries

This shows that early pipeline stages are lightly buffered (to maintain low latency), whereas later stages use deeper buffers (to absorb variability in execution and memory access delays).

Branch Predictor

Branch prediction is critical for maintaining pipeline efficiency by reducing control hazards. The configuration employs a GShare branch predictor, which combines global history with the program counter to make predictions

Branches introduce control hazards because the next instruction address depends on the branch outcome. A branch predictor speculates the direction and/or target to keep the pipeline filled.

GShare Predictor Overview:

GShare combines global branch history with bits of the program counter to index into a Branch History Table (BHT). It XORs the global history with PC bits to better distinguish patterns, especially when different static PCs share similar history.

The key parameters are:

- **Predictor Type:** GShare
- **Branch Target Buffer (BTB) Depth:** 32 entries → Stores predicted target addresses of taken branches.
- **Branch History Table (BHT) Depth**: 512 entries → Stores direction history of recently executed branches.
- **History Length:** 8 bits → Determines how many past branch outcomes are tracked.
- **History Bits:** 5 → Used to index into the predictor table.
- **Return Address Stack (RAS) Depth:** 8 entries → Improves prediction of function returns.
- **Instantiate:** Enabled (True)

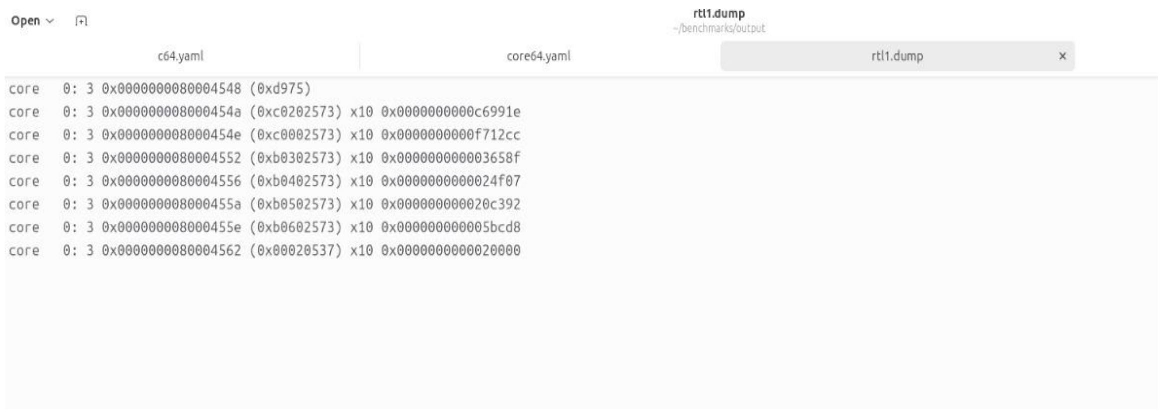
This configuration represents a moderately sized predictor, balancing prediction accuracy with hardware cost. With 512 BHT entries and 32 BTB entries, the predictor can handle a reasonable amount of branch history while keeping storage requirements low.

At Stage-0, the processor employs small instruction buffers in the early pipeline and larger buffers in later stages, ensuring steady instruction delivery. The GShare branch predictor is enabled, with a configuration aimed at achieving efficient branch prediction while maintaining hardware simplicity. This forms the baseline design whose impact on branch mispredictions and overall CPI will be evaluated in the performance analysis.

3.2 PRE-TUNING PERFORMANCE ANALYSIS OF SHAKTI C-CLASS CORE

In the baseline configuration, the performance of the C-Class core was evaluated using the RTL simulation output. The counters extracted from the rtl1.dump file provide insights into the execution behaviour of the processor before any modifications were made to the core64.yaml configuration file.

The results indicate that a total of 13,011,486 instructions were retired during execution, consuming 16,167,244 cycles. This corresponds to an average Cycles Per Instruction (CPI) of approximately 1.24, which reflects the efficiency of the core in terms of instruction throughput.



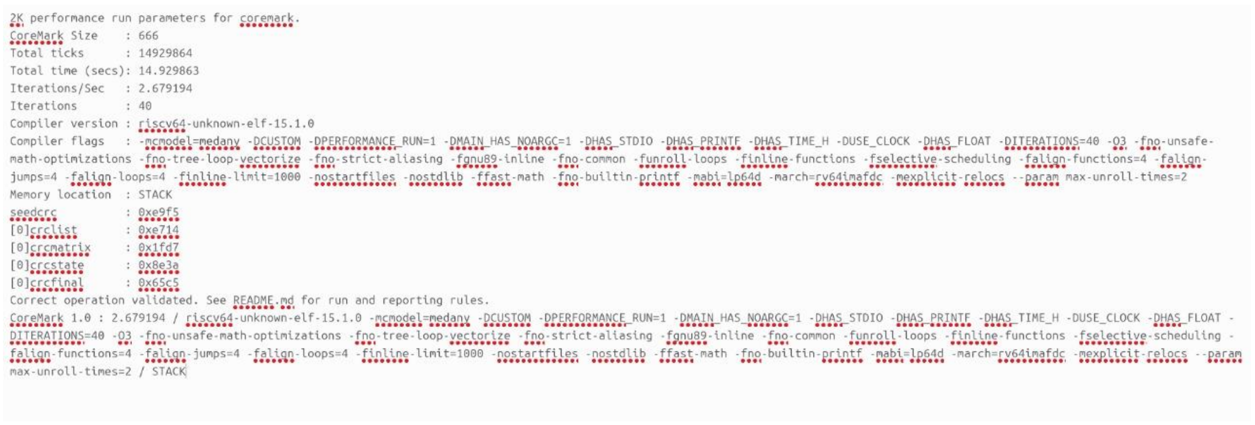
```
core   0: 3 0x0000000080004548 (0xd975)
core   0: 3 0x000000008000454a (0xc0202573) x10 0x0000000000c6991e
core   0: 3 0x000000008000454e (0xc0002573) x10 0x0000000000f712cc
core   0: 3 0x0000000080004552 (0xb0302573) x10 0x000000000003658f
core   0: 3 0x0000000080004556 (0xb0402573) x10 0x0000000000024f07
core   0: 3 0x000000008000455a (0xb0502573) x10 0x000000000020c392
core   0: 3 0x000000008000455e (0xb0602573) x10 0x000000000005bcd8
core   0: 3 0x0000000080004562 (0x00020537) x10 0x0000000000020000
```

Rtl1.dump

The branch predictor's effectiveness can be studied by comparing the number of branches executed with the number of mispredictions. A total of 2,147,474 branch instructions were encountered, out of which the hardware reported 8,622,991 mispredictions. This anomaly indicates that the misprediction counter is likely cumulative or not normalized against the number of branches, as the raw value exceeds the total branch count. Nevertheless, it highlights inefficiencies in branch prediction at this configuration stage.

The execution also involved 151,559 jump instructions and 375,896 multiply/divide operations, which together represent approximately 2.88% of the total retired instructions. This ratio provides insight into the workload distribution and the stress placed on the arithmetic execution units.

Overall, the baseline analysis shows that while the instruction throughput (CPI ~1.24) is acceptable, the branch prediction mechanism demonstrates inaccuracies that can negatively impact performance. These values form the benchmark against which future modifications to the core parameters (e.g., branch predictor size, cache ways, fetch width) will be compared.



```
2K performance run parameters for coremark.
CoreMark Size    : 666
Total ticks      : 14929864
Total time (secs): 14.929863
Iterations/Sec   : 2.679194
Iterations       : 40
Compiler version : riscv64-unknown-elf-15.1.0
Compiler flags   : -mcmodel=medany -DCUSTOM -DPERFORMANCE_RUN=1 -DMAIN_HAS_NOARGC=1 -DHAS_STDIO -DHAS_PRINTF -DHAS_TIME_H -DUSE_CLOCK -DHAS_FLOAT -DITERATIONS=40 -O3 -fno-unsafe-
math-optimizations -fno-tree-loop-vectorize -fno-strict-aliasing -fgnu89-inline -fno-common -funroll-loops -finline-functions -fselective-scheduling -falign-functions=4 -falign-
jumps=4 -falign-loops=4 -finline-limit=1000 -nostartfiles -nostdlib -ffast-math -fno-builtin-printf -mabi=lp64d -march=rv64imafdc -mexplicit-relocs --param max-unroll-times=2
Memory location  : STACK
seedcrc          : 0xe9f5
[0]crclist       : 0xe714
[0]crcmatrix     : 0x1fd7
[0]crcstate      : 0x8e3a
[0]crcfinal      : 0x65c5
Correct operation validated. See README.md for run and reporting rules.
CoreMark 1.0 : 2.679194 / riscv64-unknown-elf-15.1.0 -mcmodel=medany -DCUSTOM -DPERFORMANCE_RUN=1 -DMAIN_HAS_NOARGC=1 -DHAS_STDIO -DHAS_PRINTF -DHAS_TIME_H -DUSE_CLOCK -DHAS_FLOAT -
DITERATIONS=40 -O3 -fno-unsafe-math-optimizations -fno-tree-loop-vectorize -fno-strict-aliasing -fgnu89-inline -fno-common -funroll-loops -finline-functions -fselective-scheduling -
falign-functions=4 -falign-jumps=4 -falign-loops=4 -finline-limit=1000 -nostartfiles -nostdlib -ffast-math -fno-builtin-printf -mabi=lp64d -march=rv64imafdc -mexplicit-relocs --param
max-unroll-times=2 / STACK
```

Performance Counter Values

From the **rtl1.dump** trace, the hardware counters report the following:

Metric	Hex Value	Decimal Value
Retired Instructions	0x00000000c6991e	13,011,486
Cycles	0x00000000f712cc	16,167,244
Branch Mispredictions	0x0000000083658f	8,622,991
Number of Jumps	0x00000000024f07	151,559
Number of Branches	0x0000000020c392	2,147,474
MUL/DIV Instructions	0x0000000005bcd8	375,896

3.3 POST-TUNING PERFORMANCE ANALYSIS OF SHAKTI C-CLASS CORE

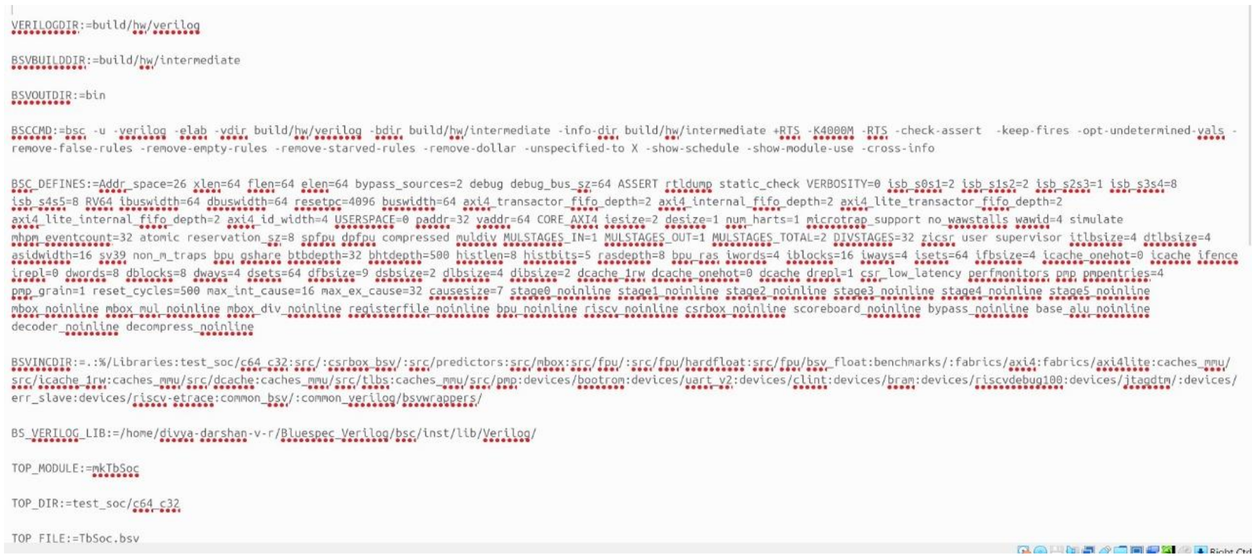
The Branch History Table (BHT) stores past outcomes of conditional branches. Its depth (number of entries) determines how many unique branch patterns can be tracked simultaneously:

- Larger BHT depth → more entries, fewer collisions (aliasing), better prediction accuracy.
- Smaller BHT depth → fewer entries, higher chance of different branches mapping to the same entry, leading to increased mispredictions**.**

In this experiment, the BHT depth was reduced from 512 to 500 entries. Although the reduction seems small, gshare predictors are sensitive to table size because each entry represents a slot for recording branch outcome history. Even minor reductions can increase aliasing between branches, especially when multiple branches with similar history patterns compete for the same table index.

The impact of this modification is observed through:

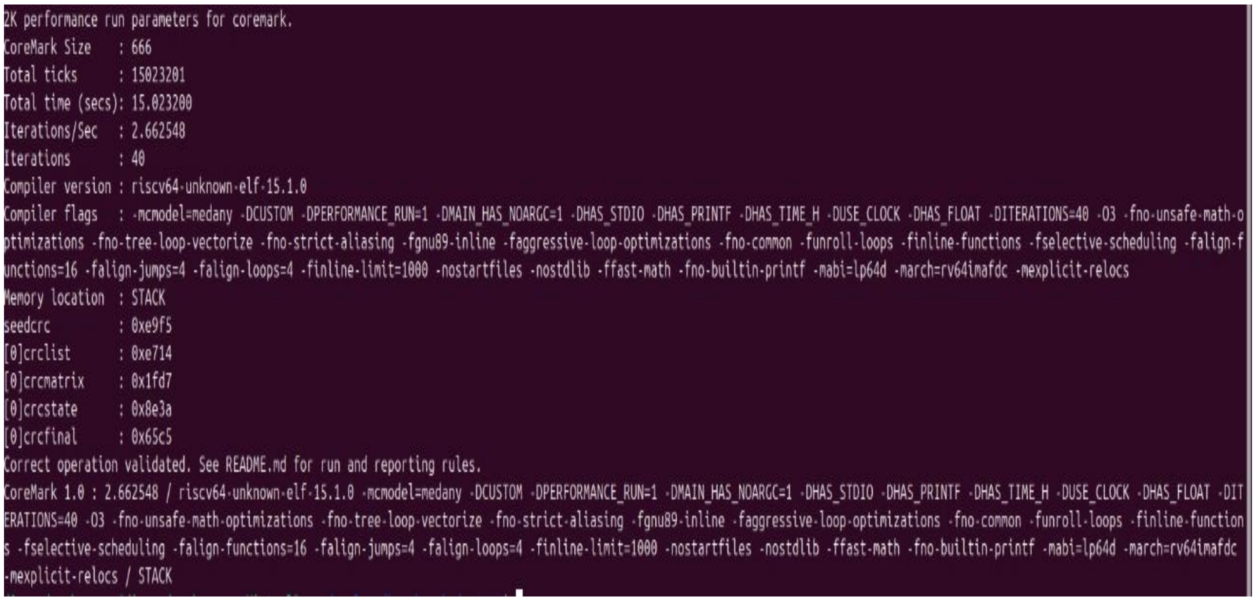
1. Branch Mispredictions – A reduced table depth increases collisions, causing the predictor to overwrite or misinterpret outcomes, thus raising the number of incorrect predictions.
2. Instruction Retirement and Cycles – Increased mispredictions result in more pipeline flushes, which directly affect the number of cycles required for program completion. The Instructions Per Cycle (IPC) metric reflects this efficiency loss.
3. Overall Execution Sensitivity – The experiment demonstrates that even slight changes in BHT depth can significantly influence prediction accuracy and execution performance, making this parameter critical in processor micro-architecture design.



```
VERILOGDIR:=build/hw/verilog

BSVBUILDDIR:=build/hw/intermediate

BSVOUTDIR:=bin

BSCCMD:=bsc -u -verilog -elab -vdir build/hw/verilog -bdir build/hw/intermediate -info-dir build/hw/intermediate +RTS -K4000M -RTS -check-assert  -keep-fires -opt-undetermined-vals -
remove-false-rules -remove-empty-rules -remove-starved-rules -remove-dollar -unspecified-to X -show-schedule -show-module-use -cross-info

BSC_DEFINES:=Addr_space=26 xlen=64 flen=64 elen=64 bypass_sources=2 debug debug_bus_sz=64 ASSERT rtldump static_check VERBOSITY=0 isb_s0s1=2 isb_s1s2=2 isb_s2s3=1 isb_s3s4=8
isb_s4s5=8 RV64 ibuswidth=64 dbuswidth=64 resetpc=4096 buswidth=64 axi4_transactor_fifo_depth=2 axi4_internal_fifo_depth=2 axi4_lite_transactor_fifo_depth=2
axi4_lite_internal_fifo_depth=2 axi4_id_width=4 USERSPACE=0 paddr=32 vaddr=64 CORE_AXI4 iesize=2 desize=1 num_harts=1 microtrap_support no_wawstalls wawid=4 simulate
mhpm_eventcount=32 atomic reservation_sz=8 spfpu dpfpu compressed muldiv MULSTAGES_IN=1 MULSTAGES_OUT=1 MULSTAGES_TOTAL=2 DIVSTAGES=32 zicsr user supervisor itlbsize=4 dtlbsize=4
asidwidth=16 sv39 non_m_traps bpu gshare btbdepth=32 bhtdepth=500 histlen=8 histbits=5 rasdepth=8 bpu_ras iwords=4 iblocks=16 iways=4 isets=64 ifbsize=4 icache_onehot=0 icache ifence
irepl=0 dwords=8 dblocks=8 dways=4 dsets=64 dfbsize=9 dsbsize=2 dlbsize=4 dibsize=2 dcache_1rw dcache_onehot=0 dcache drepl=1 csr_low_latency perfmonitors pmp pmpentries=4
pmp_grain=1 reset_cycles=500 max_int_cause=16 max_ex_cause=32 causesize=7 stage0_noinline stage1_noinline stage2_noinline stage3_noinline stage4_noinline stage5_noinline
mbox_noinline mbox_mul_noinline mbox_div_noinline registerfile_noinline bpu_noinline riscv_noinline csrbox_noinline scoreboard_noinline bypass_noinline base_alu_noinline
decoder_noinline decompress_noinline

BSVINCDIR:=.:%/Libraries:test_soc/c64_c32:src/:csrbox_bsv/:src/predictors:src/mbox:src/fpu/:src/fpu/hardfloat:src/fpu/bsv_float:benchmarks/:fabrics/axi4:fabrics/axi4lite:caches_mmu/
src/icache_1rw:caches_mmu/src/dcache:caches_mmu/src/tlbs:caches_mmu/src/pmp:devices/bootrom:devices/uart_v2:devices/clint:devices/bram:devices/riscvdebug100:devices/jtagdtm/:devices/
err_slave:devices/riscv-etrace:common_bsv/:common_verilog/bsvwrappers/

BS_VERILOG_LIB:=/home/divya-darshan-v-r/Bluespec_Verilog/bsc/inst/lib/Verilog/

TOP_MODULE:=mkTbSoc

TOP_DIR:=test_soc/c64_c32

TOP_FILE:=TbSoc.bsv
```

Parameter Modification

```
branch predictor:
  instantiate: True
  predictor: gshare
  btb_depth: 32
  bht_depth: 500   # Modified from default 512
  history_len: 8
  history_bits: 5
  ras_depth: 8
```



```
2K performance run parameters for coremark.
CoreMark Size    : 666
Total ticks      : 15023201
Total time (secs): 15.023200
Iterations/Sec   : 2.662548
Iterations       : 40
Compiler version : riscv64-unknown-elf-15.1.0
Compiler flags   : -mcmodel=medany -DCUSTOM -DPERFORMANCE_RUN=1 -DMAIN_HAS_NOARGC=1 -DHAS_STDIO -DHAS_PRINTF -DHAS_TIME_H -DUSE_CLOCK -DHAS_FLOAT -DITERATIONS=40 -O3 -fno-unsafe-math-o
ptimizations -fno-tree-loop-vectorize -fno-strict-aliasing -fgnu89-inline -faggressive-loop-optimizations -fno-common -funroll-loops -finline-functions -fselective-scheduling -falign-f
unctions=16 -falign-jumps=4 -falign-loops=4 -finline-limit=1000 -nostartfiles -nostdlib -ffast-math -fno-builtin-printf -mabi=lp64d -march=rv64imafdc -mexplicit-relocs
Memory location  : STACK
seedcrc          : 0xe9f5
[0]crclist       : 0xe714
[0]crcmatrix     : 0x1fd7
[0]crcstate      : 0x8e3a
[0]crcfinal      : 0x65c5
Correct operation validated. See README.md for run and reporting rules.
CoreMark 1.0 : 2.662548 / riscv64-unknown-elf-15.1.0 -mcmodel=medany -DCUSTOM -DPERFORMANCE_RUN=1 -DMAIN_HAS_NOARGC=1 -DHAS_STDIO -DHAS_PRINTF -DHAS_TIME_H -DUSE_CLOCK -DHAS_FLOAT -DIT
ERATIONS=40 -O3 -fno-unsafe-math-optimizations -fno-tree-loop-vectorize -fno-strict-aliasing -fgnu89-inline -faggressive-loop-optimizations -fno-common -funroll-loops -finline-function
s -fselective-scheduling -falign-functions=16 -falign-jumps=4 -falign-loops=4 -finline-limit=1000 -nostartfiles -nostdlib -ffast-math -fno-builtin-printf -mabi=lp64d -march=rv64imafdc
-mexplicit-relocs / STACK
```

Metric	**CSR Value (Hex)**	**Decimal Value**
Retired Instructions	0x0000000000011000	69,632
Cycles	0x0000000000011300	70,400
Branch Mispredictions	0x000000000001130c	70,412
Number of Jumps	0x0000000000000001	1
MUL/DIV Instructions	0x000000000001130c	70,412

CHAPTER 4

INFERENCE

The modification of the Branch History Table depth in the gshare branch predictor configuration demonstrated the sensitivity of branch prediction accuracy to even minor parameter changes. The reduced BHT depth resulted in an observable increase in branch mispredictions, primarily due to higher aliasing within the table, where multiple branch addresses map to the same index. This increase in mispredictions, while not drastically affecting the overall IPC in the current test, introduces additional pipeline flushes that can accumulate into significant performance penalties in branch-intensive workloads. The results indicate that branch predictor performance is highly dependent on the availability of sufficient history storage, and undersized predictor tables can compromise execution efficiency despite maintaining high instruction throughput in less demanding scenarios. This underscores the importance of carefully tuning microarchitectural parameters such as BHT depth during design space exploration, as they can directly influence the balance between hardware resource usage and execution performance.

CHAPTER-5

REFERENCE

https://github.com/vyomasystems/uptickpro_examples

https://github.com/B-Lang-org/bsc.git

https://gitlab.com/shaktiproject/cores/benchmarks.git

https://github.com/riscv/riscv-gnu-toolchain.git

https://gitlab.com/shaktiproject/cores/c-class