HW2_ Return Address Predictor Design

郭宗信, 111550105

Abstract*—This report presents the design and implementation of a Return Address Predictor (RAP) for the Aquila RISC-V processor to improve its CoreMark performance. We first analyze the existing Branch Prediction Unit (BPU) implementation in Aquila and its impact on CoreMark scores through experimental evaluation. Based on the analysis of branch statistics in CoreMark benchmarks, we propose a Return Address Stack (RAS) based predictor to specifically handle function return predictions. The implementation focuses on accurately predicting the target addresses of jalr instructions used for function returns. Our design incorporates a circular LIFO buffer to prevent stack overflow issues caused by tail-call optimization. Experimental results demonstrate the effectiveness of the proposed RAP design in improving branch prediction accuracy and overall processor performance.

Keywords*—Return Address Predictor (RAP), Aquila Processor, RISC-V, Branch Prediction Unit (BPU), CoreMark Performance, Return Address Stack (RAS), Function Return Prediction, Circular LIFO Buffer, Tail-call Optimization, Branch Prediction Accuracy

I. Introduction

Branch prediction plays a crucial role in modern pipelined processors by reducing the performance impact of control hazards. In a 5-stage pipeline architecture like Aquila, branch instructions can introduce delays of up to two cycles while waiting for the Execute stage to determine the next Program Counter (PC) value. While traditional Branch Prediction Units (BPUs) handle conditional branches and direct jumps effectively, they often struggle with indirect jumps, particularly function returns implemented using the jalr instruction in RISC-V.

The current Aquila implementation uses a simple 2-bit saturating counter-based predictor that works well for regular branch patterns but does not specifically address function returns. Function returns present a unique challenge because their target addresses depend on the call context rather than local or global branch history. A specialized Return Address Predictor can significantly improve prediction accuracy for these cases by maintaining a stack of return addresses that mirrors the program's call stack.

II. Methodologies

This project is divided into two primary parts: analysis of the Branch Prediction Unit (BPU) and implementation of the Return Address Predictor (RAP).

A. Analysis of the Branch Prediction Unit (BPU)

The first part focuses on BPU analysis. Initially, we tested both enabling and disabling the BPU to observe CoreMark performance. We further analyzed the effect of varying Branch History Table (BHT) sizes on CoreMark scores, exploring hit and miss rates under different conditions. The analysis included several configurations:

Setting the prediction results to zero, representing always not taken,treating both outcomes as not taken,combining different logical settings to disable the BPU selectively.

Additionally, we examined the impact of using small, medium, and large BHT sizes to observe hit and miss rate differences. For BHT size variation, three configurations were tested:

- **Small**: 32, 64 entries,
- **Medium**: 128, 256, 512 entries,
- **Large**: 1024, 8192 entries.

B. Implementation of the Return Address Predictor (RAP)

The second part involves implementing the RAP. The RAP design references the BPU structure but customizes the table and prediction mechanisms specifically for return address prediction. Unlike branch predictions, RAP does not determine if a return address should be taken or not taken, as function returns (via `jalr`) are inherently always taken. The design thus assumes an initial miss for the return prediction, creating a predictive table to communicate the predicted return address to the program counter (PC) and execution stage. In case of misprediction, the execution unit is notified to trigger a restore, prompting the pipeline to flush and recover. Architecturally, RAP operates as a stage parallel to the execution stage, as in the BPU design.

III. Implementation

This experiment used 50 iterations to stabilize CoreMark results and avoid performance disturbances.

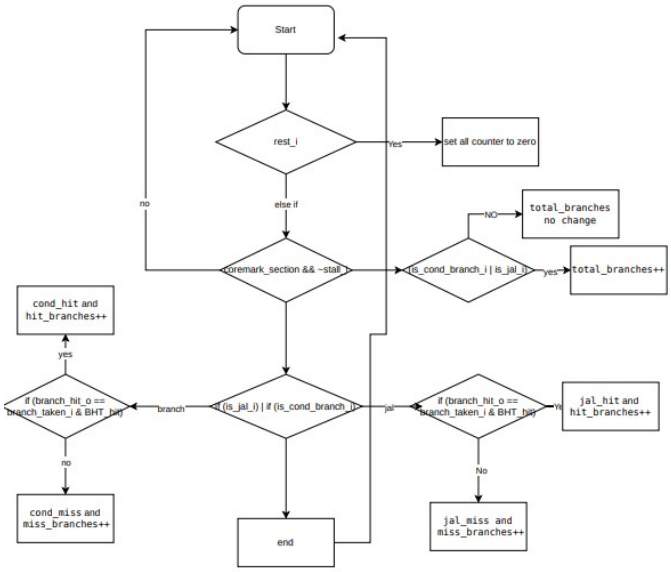
A. BPU Analysis

In this part, we analyzed the Branch Prediction Unit (BPU) by exploring the effects of enabling and disabling its core functionalities and adjusting the Branch History Table (BHT) size. This analysis aimed to observe the impact on CoreMark scores and examine hit and miss rates across different branch types.

Disabling BPU Configurations Multiple configurations were tested to disable or modify BPU operation: **Set branch_hit_o to zero**: This configuration simulates a scenario where all branches are assumed to miss, reflecting a pessimistic approach to branch prediction. Set branch_decision_o to zero: Here, all branch decisions default to not taken, which allows observation of non-branching impacts on the pipeline. Disable both **branch_hit_o and branch_decision_o**: Both predictions are set to zero, simulating a BPU that does not participate in branch handling, allowing a control measurement for CoreMark score. **Set we (write enable) to zero**: This halts updates to the BHT, effectively testing CoreMark performance

with a static, non-updating prediction table. Set both branch_hit_o and branch_decision_o to one: In this configuration, the BPU assumes all branches are correctly predicted, representing an ideal scenario with zero mispredictions.

Fig.1. *flowchart of hit rate analysis*



Adjusting BHT Size To understand the effect of BHT size on prediction accuracy and pipeline performance, we tested the BPU with small, medium, and large BHT configurations: **Small BHT sizes**: 32 and 64 entries, representing minimal allocation for branch prediction history. **Medium BHT sizes**: 128, 256, and 512 entries, providing a balanced allocation of resources. **Large BHT sizes**: 1024 and 8192 entries, representing extensive allocation for high-performance prediction data storage.

B. Return Address Predictor (RAP) Implementation

The RAP module functions as a specialized Return Address Predictor for accurately predicting function return addresses, especially for `jalr` instructions.

System Parameters and Signals: The RAP module has configurable parameters: BHT_ENTRY_NUM defines the number of entries in the Branch History Table (BHT), RAS_ENTRY_NUM specifies the Return Address Stack (RAS) depth, XLEN sets the data width. Core input signals include clk_i (system clock), rst_i (reset), stall_i (stall signal), pc_i (program counter address), and branch control signals is_ret_i and is_jal_i, allowing the predictor to differentiate between function calls and returns.

Branch History Table (BHT): The BHT is a direct-mapped table that stores branch history by associating each entry with a PC tag and target address. The we_BHT signal enables writing when a branch instruction is decoded and missing from the BHT, updating dynamically.

Return Address Stack (RAS): The RAS uses a circular stack to store return addresses for function calls. The stack pointer (sp) tracks the top of the stack, with sp_plus_1 and sp_minus_1 supporting circular indexing. When a branch flush (flush_i) occurs, the recover signal restores the stack top from backup storage.

Backup Storage (BS): The BS acts as a circular buffer for backup addresses, providing quick recovery if mispredictions occur. On a successful return prediction (ret_hit_o), the RAS top is pushed to the BS. If exe_ret_executed_i signals successful execution, the BS clears the top address.

Output Signals and Counters: ret_hit_o indicates a successful return prediction, and ret_target_addr_o provides the target address. Performance counters ret_hit_count and ret_miss_count track return prediction accuracy, supporting further RAP optimizations.

This design efficiently integrates RAS and BS mechanisms, enhancing CoreMark performance through accurate return predictions and effective misprediction recovery.

IV. RESULTS

A. BPU Analysis

This table and figure presents the performance data gathered using CoreMark.

TABLE I. COREMARK SORE IN DIFFERENT SITUZTION

index	***operation***	***score***
1	original	100.735673
2	branch_hit_o and branch_decision_o set 0	89.146502
3	branch_hit_o and branch_decision_o set 1	wrong
4	branch_hit_o set 0	89.146582
5	branch_decision_o set 0	89.146582

TABLE II. COREMARK SORE IN DIFFERENT BPU BHT SIZE

index	***size***	***score***
1	32	99.881341
2	64	100.735673
3	128	100.922431
4	256	101.157546
5	512	101.186411
6	1024	101.191633
7	8192	101.191633

Fig.. *figure of rate of the CoreMark score versus BHT size*

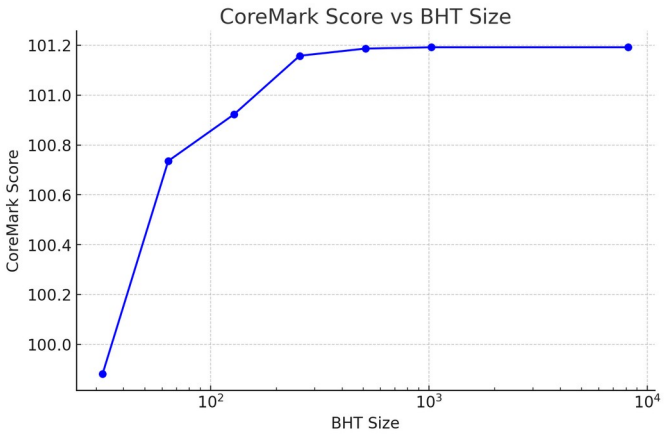


TABLE III. HIT RATE ANALYSIS

index	***name***	***value***	***rate***
1	total branch	330049	100%
2	hit branches	79711	24.15%
3	miss branches	250338	75.84%
4	jal brnach	34170	10.35%
5	cond branch	295879	89.64%
6	jal hit	1068	3.12%
7	jal miss	33102	96.87%
8	cond hit	78643	26..58%
9	cond miss	217236	73.42%

B. Return Address Predictor (RAP)

TABLE IV. BEFORE RAP AND AFTER RAP

index	***operation***	***coremark score***
1	original	100.735673
2	with RAP	100.749427

TABLE V. HIT RATE

index	***stage***	***hit rate***
1	BPU	24.15%
2	RAP	23.44%

V. DISCUSSION

A. BPU Analysis

Table I provides an analysis of various configurations for disabling core BPU signals, such as `branch_hit_o`, `branch_decision_o`, and `we`. Results indicate that disabling these signals leads to a significant drop in CoreMark scores, confirming the critical role of accurate branch prediction in performance. These signals not only control the BPU's functionality but also directly impact prediction accuracy, as seen in the lower scores when they are disabled. This suggests that even simple BPU controls can significantly affect CoreMark performance.

In Table II, we examine the impact of BHT size on performance. Increasing the BHT size improves CoreMark scores substantially, with the most notable gains observed between 32 and 512 entries. Beyond 512 entries, the score stabilizes, suggesting that further increases provide diminishing returns. This plateau in performance could be due to the lack of optimization in the current BHT structure. A potential improvement might involve introducing hierarchical BHT layers, similar to cache designs, to enable more efficient lookup and larger table sizes without sacrificing speed.

B. Hit Rate Analysis

Table III provides a detailed hit rate analysis. Results indicate that the BPU has a higher success rate for conditional branches than for `jal` return instructions, where the miss rate is particularly high. This discrepancy highlights the BPU's limitations in handling return addresses. The low hit rate for `jal` instructions suggests that a general-purpose BPU design is insufficient for accurately predicting function returns. Implementing the RAP helps alleviate some of these limitations by focusing on `jalr` return predictions. However, for broader applicability, a specialized mechanism that tracks frequently called functions in a table might enhance prediction accuracy, especially for high-frequency `jalr` targets.

C. RAP Integration and Performance Impact

Tables IV and V compare CoreMark scores and hit rates before and after RAP integration. With the RAP enabled, the CoreMark score shows a modest increase. Although the improvement is limited, this may be due to a low proportion of predictable `jalr` instructions in the benchmark or possible connection errors in the RAP implementation. Additionally, the RAP's hit rate is comparable to that of the BPU, but it specifically reduces mispredictions for `jalr` instructions, thus mitigating performance losses associated with these returns. By maintaining accurate return addresses through the RAS and BS, the RAP reduces penalties from incorrect return predictions.

D. Hit Rate Analysis

The findings indicate areas for further enhancement in both the BPU and RAP designs:

1. Optimizing the BPU Prediction Model: A potential improvement for the BPU could involve separating conditional branch handling from `jal` predictions, using tailored algorithms for each. Moreover, rather than relying solely on a 2-bit predictor, more sophisticated logic could be introduced, such as loop detection mechanisms that reduce misses in repetitive structures. This approach would ensure accurate predictions beyond simple branch conditions.

2. Enhancing RAP Functionality: While the RAP effectively targets `jalr` instructions, there is room to expand its scope. Addressing potential issues in push and pop mechanisms could improve prediction efficiency. Additionally, a more comprehensive RAP design could consider frequently called function addresses, maintaining them in a dedicated table for quick access. This approach would preemptively predict return addresses for functions with high jump frequencies, reducing lookup time and enhancing overall prediction accuracy.