Two-level Branch Predictor of Aquila SoC

Abstract

In modern microprocessor design, to minimize cycle time and maximize processor throughput, an increasing number of pipeline stages are being integrated into processors. This allows instructions without dependencies to execute simultaneously in different pipeline stages, significantly enhancing circuit resource utilization and overall processor performance. However, pipelines are not without their drawbacks, the most common being two types of hazards. The first is the data hazard, which can be resolved by adding additional bypassing circuits. The second significant hazard is the control hazard, which arises when the processor encounters a branch instruction and must determine how to proceed. Without speculative execution, each branch instruction would require waiting for the execute stage to produce a result before continuing, causing a delay of at least two cycles. Therefore, an effective branch prediction mechanism must be incorporated into the processor to minimize the stall cycles or flushes caused by branches.

Introduction

Aquila is an open-source 32-bit RISC-V RV32IMA compliant processor core designed by EISL.NYCU, which contains a five stages pipeline and support UART-communication, so benchmark program can be sent to it for testing the performance.

In this project, I will implement several different branch prediction models and compare their effects on Aquila efficiency. These models range from having no branch prediction, to the simplest models such as always taken or always not taken, to more practical implementations like the 2-bit saturation counter, and finally the two-level branch predictor. I will also modify various parameters within these models to observe their impacts. Ultimately, the goal is to identify the most suitable branch predictor model.

Analysis

1. No branch predictor:

   Without any branch predictor functionality, each branch instruction must wait until the execute stage computes the branch result before continuing execution, resulting in at least two cycles of stall. Undoubtedly, the absence of branch prediction leads to the worst execution efficiency. Even if every branch prediction is incorrect, only two instructions in the decode stage and fetch stage need to be flushed, making the efficiency of having a branch predictor unquestionably better.

   Advantages:

   - Simple implementation
   - Low resource usage

   Disadvantages:

   - The stall cycles caused by branch instructions significantly impact processor performance

2. Always taken / Always not taken:

   This is the simplest method to reduce the negative impact of branches on the processor, and its implementation is straightforward, requiring minimal additional hardware resources. In assembly language, there are two primary reasons for branch instructions: if-else statements and loop structures. Among these, the branch outcomes of if-else statements are generally more evenly distributed. However, the branch outcomes of loop structures are predominantly taken since the purpose of a loop is to execute repeatedly. Therefore, considering the overall branch outcomes of a program, the proportion of branch taken is much higher than branch not taken.

   I added some registers in the Branch Prediction Unit (BPU) to record the branch status for analysis. The experimental data clearly demonstrated this characteristic. During the execution of Dhrystone for 1,000,000 cycles, the branch taken/not taken ratio was 67%/33%. Evidently, if we predict that each branch outcome is taken, we can reduce the time wasted on waiting for branches by 67% (assuming a branch penalty of two cycles). Compared to having no branch prediction, this result is significantly better.

   Advantages:

   - Minimal changes to the processor architecture
   - Low resource usage

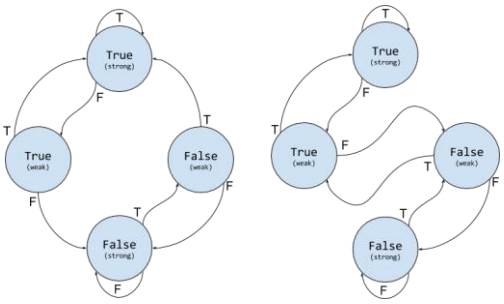
   Disadvantages:

   - Low accuracy
   - Inability to adapt to different workloads

3. Saturating-counter:

   Although always taken can achieve a reasonably high prediction accuracy, it is far from sufficient for processor designs that demand extreme efficiency. Moreover, an effective branch prediction mechanism should predict different outcomes for different branches. Therefore, we employ saturating counters to aid in branch prediction.

   In simple terms, for each branch instruction, we should consider its previous branch history to predict whether the next branch will be taken. Saturating counters provide a straightforward and convenient model for estimating branch outcomes based on past history. When the processor encounters a branch instruction at a certain address for the first time, the saturating counter is initialized to the state of that branch (taken or not taken). Each time this branch instruction is subsequently executed, the most significant bit (MSB) of the counter is used as the prediction for the branch, and the counter is updated based on the actual outcome determined in the execute stage. Thus, each branch's prediction is based solely on its own history.



Number of BHT entries	Hit rate of FSM1	Hit rate of FSM2
4	96%	96%
8	96.2%	96.3%
16	96.2%	96.2%
32	96.2%	96.2%
64	96.2%	96.2%

   In the design of saturating counters, two key factors can impact efficiency. The first is the finite state machine (FSM) of the saturating counter. Different FSM designs can affect prediction results for various workloads. I tested two different FSM designs, but they did not produce significant differences in our benchmark, so we will proceed with one of them for further discussion. The second factor is the number of Branch History Table (BHT) entries. Each unique branch instruction requires a separate BHT entry for tracking. If there are fewer BHT entries than branch instructions, some entries will need to be replaced, resulting in some branches being unpredicted. However, BHT entries consume considerable hardware resources, so a balance must be struck between resource usage and efficiency.

Number of BHT entries	BHT hit rate	branch hit rate
4	62.9%	96%
8	74.7%	96.3%
16	75.4%	96.2%
32	98.4%	96.2%
64	98.4%	96.2%

According to the experimental results, we can observe that the number of BHT entries has a significant impact on branch prediction. Notably, the prediction accuracy with 8 BHT entries is very similar to that with 16 BHT entries, and the accuracy with 32 BHT entries is very similar to that with 64 BHT entries. To understand this, we can examine the structure of the Dhrystone code, which consists of inner and outer loops. The branch instructions frequently executed in the inner loops do not exceed eight, hence the similar performance between 8 and 16 entries. Additionally, since the total number of branch instructions in the entire Dhrystone code does not exceed 32, the results for BHT entries exceeding 32 are identical to those with 32 entries. This is an important finding, indicating that the optimal number of BHT entries is determined by the number of branch instructions.

Number of BHT entries	LUTs usage	Slice registers usage	Dhrystone per second
4	136	138	52523.6
8	257	275	54042.3
16	486	548	54324.2
32	1050	1093	57155.9
64	2232	2196	57155.9

In this chart(I have re-synthesize BPU with 16 BHT entries after presentation), I compared the additional hardware resources required for different numbers of BHT entries and the corresponding performance improvements. We can easily observe that the number of BHT entries is roughly proportional to the hardware resources required by the branch predictor. Therefore, blindly increasing the number of BHT entries is not advisable. The key is to balance hardware costs and processor performance.

Advantages:

- Easy to implement
- High prediction accuracy

Disadvantages:

- Only considers local history (its own branch record) and does not account for the global branch history (records of all branch instructions)
- Unable to reflect spatial correlation

4. Two-level branch prediction:

   To incorporate the results of global branches into branch prediction, we can store the outcome of each branch in a shift register as a pattern. This pattern is then used as an index to select the corresponding BHR (Branch History Register) entry. Theoretically, this method should increase the branch hit rate, but actual data is needed to verify this. Below, I will implement several different Two-level predictors and compare their advantages and disadvantages:

   Assume the length of the shift register is nnn bits.

   Each BHR entry has 2^n saturating counters:

   Originally, each BHR entry corresponds to one saturating counter. In this implementation, each BHR entry will correspond to 2^n saturating counters, with each counter representing the branch outcome under a different global branch pattern.

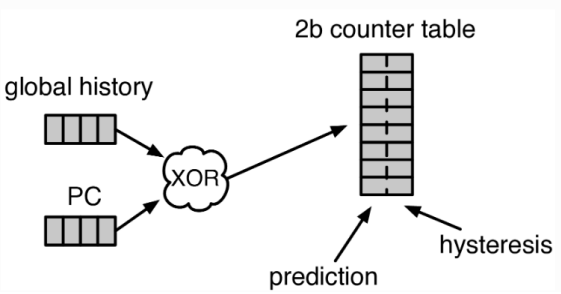
   Advantages:

   - Can record all scenarios, avoiding conflicts

   Disadvantages:

   - If some patterns do not occur, it wastes resources significantly

   Use gshare to index BHT:



   The previous method clearly uses a substantial amount of hardware resources, and some of these resources might be underutilized (for patterns that do not occur). To improve resource efficiency and enhance the correlation between branch prediction and global branch history without significantly increasing hardware usage, a simpler method is to use gshare.

   Gshare uses bits from the program counter (PC) and global branch history, performing an XOR operation to form the index for the BHT entries.

   Example:

   Suppose there are n bits in global branch history and we select n bits starting from x in PC.

   wire BHT_index;

   assign BHT_index = global_branch_history[n-1:0] ^ pc[x+n-1:x];

   The advantage of this method is evident: different BHT entries inherently contain information

about the global branch history, which should theoretically lead to more accurate predictions. However, upon implementation, we observed the following results:

predictor	LUTs usage	Slice registers usage	branch hit rate	BHT hit rate	Dhrystone per second
2-bit saturating counter	2032	2196	98.4	96.2	57155.9
Two-level	2618	2569	98.4	96	56689.3
gshare	2628	2153	97.6	96.5	56844.0

The results do not align with our expectations. Theoretically, incorporating global history should improve prediction accuracy. However, in our experimental results, the efficiency is evidently not as good as that of the simple 2-bit saturating counter predictor. The main reason for this is that the original saturating counter design maps each PC to a single BHR entry, whereas gshare might result in different addresses XORing to the same entry, causing conflicts and thus reducing efficiency.

Additionally, this could be because our benchmark is overly focused on executing a few fixed instructions. In a real-world scenario with more complex system operations and a larger branch predictor size, a two-level predictor would likely yield better prediction accuracy.

For a simple two-level predictor, the increase in hardware usage is not directly proportional to the number of entry bits. This is because such an implementation mainly increases the number of saturating counters. However, in our Branch Prediction Unit (BPU), a significant portion of resources is allocated to storing the target address corresponding to each branch instruction's Program Counter (PC), a module referred to as pc_history. These entries do not consume additional resources when employing a two-level implementation, hence the increase in area is not as substantial as might be expected.

selected part	Dhrystone per second
[7:2]	56586.5
[8:3]	55311.3
[9:4]	56689.9

This chart illustrates the impact of selecting different parts for gshare on performance. Assuming the length of the shift register is fixed, different selections can result in significant efficiency

differences. Therefore, determining which bits to select for optimal prediction performance requires further experimentation.

Although in our experiments, due to the relatively simple workload, the two-level predictor's performance did not meet expectations, data from other more complex experiments indicate that two-level predictors achieve higher prediction accuracy than saturating counters. Future work can explore different two-level models (such as TAGE, or even machine learning-based models) and experiment with various parameters to determine the optimal branch predictor.

Related work

https://people.cs.pitt.edu/~childers/CS2410/slides/lect-branch-prediction.pdf

https://docs.boom-core.org/en/latest/sections/branch-prediction/backing-predictor.html

https://course.ece.cmu.edu/~ece740/f15/lib/exe/fetch.php?media=18-740-fall15-lecture05-branch-prediction-afterlecture.pdf