Lab2, Return Address Predictor Design

JUN-KAI CHEN, 111550001

***Abstract*—In the first part of the report, we test the performance of the branch prediction unit (BPU) in the Aquila core in different situations. In the second part, we design a return address stack (RAS) for the original BPU used in the Aquila core and test the improvement of the performance.**

***Index Terms*—Aquila, Return Address Stack, Branch Predictor, CoreMark**

I. Branch Analysis

The CoreMark program used in this report is compiled by gcc compiler 13.2.0 version, with the number of iteration is 0.

A. Without the BPU

If we choose not to use the BPU in the Aquila core, it will simply increment the program counter and fetch the next instruction. Once it finds that a branch instruction is taken, correspoding signal will be asserted, and the pipeline register will be flushed.

To clearly understand the execution of branch instructions , I recorded all kinds of branch instructions and their jump times. The statistics are in table I.

If a conditional a branch instruction has lower taken ratio, it will be more efficient on Aquila without the BPU. The average taken ratio is 0.5799, which means assuming branch is not taken is less efficient in the CoreMark program. Instead of assuming branch is not taken, an alternative approach is to design a module that tries to decode and calculate the target PC at the fetch stage.

TABLE I
CoreMark score without the BPU

type	total times	taken times	taken ratio
beq	25698394	2989014	0.1163
bge	2667979	1192725	0.4471
bgeu	3195135	2188085	0.6848
blt	1040904	585272	0.5623
bltu	1684779	142988	0.0849
bne	36475951	33939898	0.9305
jal	6731133		
jalr	3553204		

B. With the BPU

For any instruction in the CoreMark program, the BPU either provides a prediction or not, based on whether the instruction is recorded by the branch history table (BHT). Thus, the only exception is a BHT miss. More precisely, the BPU always treats a recieved instruction as normal instruction that doesn't require a jump. Hence, whenever there is a branch instruction with no record, the BPU will always make a mistake and then record it. This situation mainly arises from BHT miss, and secondly from cold start.

On the other hand, if the address of an instruction is found in the BHT, the BPU provides a prediction based on the content of the entry. There will be four situations depends on prediction and execution result.

- true positive: prediction and result are "taken".
- true negative: prediction and result are "not taken".
- false positive: prediction says "taken", but the result is "not taken".
- false negative: prediction says "not taken", but the result is "taken".

The statistics for the BPU is listed in table II below. The "count" field of type "miss" indicates the total number of BHT cache miss. Additionally, flush ratio caused by BHT cache misses is 0.8057.

In all cases, instructions that cause cache misses and require jumps, as well as false positives and false negatives, cause the pipeline being flushed.

According to the statistics, the proportion of pipeline flush caused by BHT cache miss is 0.5681. Thus, Aquila actually spends more clock cycles on cache misses rather than mispredictions. However, Aquila achieves 0.9185 of average accuracy if it predicts a branch instruction.

Another conclusion is that if a branch jump is more likely to occur midway, this branch will be more difficult to predict. But in the case of CoreMark program, these higher failure rates are infrequent. Most of branch instructions are `beq` and `bne`. Although BPU performs well on these two instructions, they are still the second and third leading causes of pipeline flushes.

II. Improving Aquila

With the original BPU design, we can slightly modify the BPU structure and measuring the performance. We tried two modified approaches, but the detailed datas are not shown in the table due to space issues.

A. Different BHT Size

One of the simple way to improve the BPU in Aquila is to increase the BHT size to decrease the number of cache misses. The method doesn't require any new modules. Here, we tried to use a BHT which contains 64 entries. According to the information from the CoreMark program, the time taken is only reduced by about 0.07 seconds, and the performance is improved by about 0.5%.

TABLE II
CoreMark score with the BPU

type	count	tp	ratio	tn	ratio	fp	ratio	fn	ratio
beq	25698396	677754	0.0333	17487217	0.8600	626778	0.0308	1541252	0.0758
bge	2667979	949322	0.3722	1210483	0.4746	153118	0.0600	237480	0.0931
bgeu	3195133	1950643	0.7893	145418	0.0588	297681	0.1205	77460	0.0313
blt	1040904	496861	0.4819	374837	0.3635	74291	0.0721	85101	0.0825
bltu	1684771	101729	0.0610	1527115	0.9157	9743	0.0058	29085	0.0174
bne	36658352	32866256	0.9272	124990	0.0035	2368430	0.0668	86680	0.0024
jal	6731133	5056656	1.0000	0	0.0000	0	0.0000	0	0.0000
miss	9120288								

The design reduces BHT miss by 42.5%. It seems to be a great progress. However, the BHT hit rate decreases slightly. The conflict of the two results resulted in only a reduction of 1477627 cycles spent on pipeline flushing. By calculation, this would reduce the spent time by approximately 0.072 seconds, which exactly matches the result of the CoreMark program.

B. Different Size of Saturating Counter

Another approach is to modify the length of saturating counters in BPU. In order to compare to the original BPU, we set the size of BHT to 32 entries.

The original design uses two bits to implement the saturating counter. We tried 3-bit version and recorded the performance. An important design detail is that in Aquila, the counter will set the status to strong type after crossing the threshold. We also adapt this concept in the 3-bit version BPU.

According to the information from the CoreMark program, this design spends 0.007 more seconds. Since the size of BHT is the same, there is no significant difference on the miss rate. But the BPU hit rate of 3-bit version is slightly lower than that of 2-bit version. This problem results in the worse performance.

C. Return Address Stack

A return address stack (RAS) uses a LIFO buffer to store the return address whenever a `jal` instruction is executed. In RISC-V, a pseudo instruction `ret` is used to represent function return. Since `ret` has the fixed pattern, it's possible to check whether an instruction is `ret` before execute stage and reduce the waste cycle.

III. Return Address Stack Design

A. Stack

The block diagram of return address stack is shown in Fig. 1. This RAS module uses a cirular stack to store the value of program counter. If some optimizations that performed by the compiler push more than one address into stack but use only the last of them, the circular stack could prevent the overflow issue. To implement this type of stack, we could use a counter with a buffer whose capacity is a power of 2.

The RAS module also maintains a return address table (RAT) which is similar to the BHT to record the address of `ret` instructions. The program counter should always check the table and get the prediction of the instruction type. Since a `ret` instruction must jump to other address. We don't need to keep the saturating counters as we did in the BPU module.

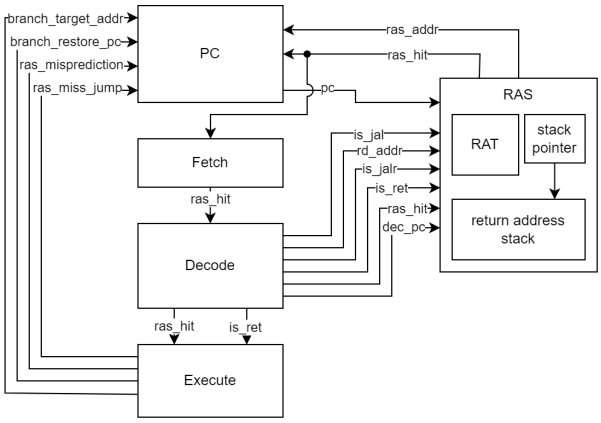

Fig. 1. The block diagram of the RAS module

In our design, the capacity of stack is 16 entries, while that of RAT is 32 entries.

B. Fetch Stage

At fetch stage, the fetch module gets the address of the PC module and the hit signal from the RAS module, passing it to the decode module by the pipeline register.

We need to remember that Aquila uses a dual port TCM as instruction and data memory. At fetch stage, a request of new instruction is sent to the TCM, and at the next clock the requested instruction will be returned. Aquila uses this delay to offset the delay of pipeline register. The impact of this design is that we will know the instruction one cycle later. Thus, we can't complete a decode pattern of the fetched instruction. Regardless of whether this operation causes a time violation, we will waste a cycle. If we adopt this design, the performance improvement we get may be only half of the potential.

C. Decode Stage

The work at decode stage is to check the pattern of the instruction. We only concern about `ret` but not `jalr`. Hence, the decode module needs to pass the other signal `is_ret` through the piepline register.

D. Execute Stage

The work at execute stage consists of three parts. First, the RAS moudle maintains the return address stack. It first check

TABLE III
RAS HIT, MISS, AND RETURN

return count	ras hit	ras miss
3197997	2767199	389546

`is_jalr` and `rd_addr` signals from the pipeline registers. If the instruction type is a call, it needs to push the return address, which is `branch_restore_pc` into the stack. On the other hand, the `is_ret` signal is used to send a pop signal to the RAS module.

Second, the RAS module need to maintain the RAT. When a new `ret` instruction comes, the module replaces the old address with it.

The last part is debugging. The prediction results of the RAS module can be divided into three types: correct, jumping to the wrong address, and not jumping. When the second situation occurs, the `ras_miss_jump` signal is asserted. This situation includes not only the address provided by the RAS module is wrong, but also the instruction type is `jal` or branch types.

IV. Return Address Stack Result

In the CoreMark program we used, the processor with the RAS module enabled spent 13.053444 seconds to complete the test. Compared to the version without RAS, which spent 13.172792 seconds, the performance has improved by 0.9% .

To understand how much RAS improves predictions, we record RAS hits and misses, as well as the number of return instructions in Table 3. In the collected data, about 98.71% of the instructions were recorded, but the hit rate of RAS was only 87.66%, which was slightly lower than the hit rate of BPU.

In addition, we also tried changing the size of the stack or RAT, such as reducing the stack capacity to 8, or setting the RAT size to 64, and the results were similar to the original data. We do not believe that the effectiveness of RAS can be improved by increasing the use of resources. As long as it exceeds a certain amount, continuing to increase resources will not cause significant changes.