HW 2 – Return Address Predictor

沈昱宏, 110705013

***Abstract*—In this homework assignment, I analyze the branch prediction unit (BPU) in the Aquila RISC-V core and its effects on CoreMark benchmark performance. I explored the impact of disabling the BPU and modifying the Branch History Table (BHT) size, examining changes in CoreMark scores. Additionally, branch prediction statistics, such as hit and miss rates for different branch types, are evaluated. Finally, a return address prediction architecture is implemented to reduce stall cycles.**

Keywords—Aquila, CoreMark, BPU

I. DISABLE BPU

A. Disabling BPU

I disabled the BPU by fixing the branch_hit_o (the output signal of BPU) to 0. Before disabling the BPU, the total cycle count was 2558316, and after disabling the BPU, the total cycle count increased to 2887697, showing a 12% increase in execution time.

B. Changing Entry Num

I changed the EntryNum from 64 to 256. The result shows that the total cycle count further reduced to 2547614, which means that using more memory did help the total cycle count to reduce.

II. BRAHCH STATISTICS

A. Statistics

I count the total number on hit and miss of different functions under the original setting. The hit rate is calculated by number of hit / total amount. The result shows that using a 4x memory can improve hit rate. Since the jal command is not often, the BHT is more likely to store address of branch commands, causing a lower hit rate.

TABLE I. STATISTICS

Function		**Metric**		
		Hit	***Miss***	***Hit Rate***
EntryNum = 64	jal	1200	39127	3 %
	branch	58586	253512	18.8 %
EntryNum = 256	jal	3216	37111	8 %
	branch	63420	248675	20.3 %

B. Counting mispredictions

I counted the number of mispredictions, which is the output from the execution stage. Under EntryNum = 64, the total number of mispredictions is 23882, which is 40% of table total hit, indicating that the branch prediction mechanism can be improved (the one implemented now is simply one-level branch predictor).

III. RAP DESIGN

In this section, I will briefly introduce how I design my return address prediction unit. The following graph is the architecture of my RAP design, note that there is more detail when implementing. I will introduce the five parts marked with the red numbers separately.

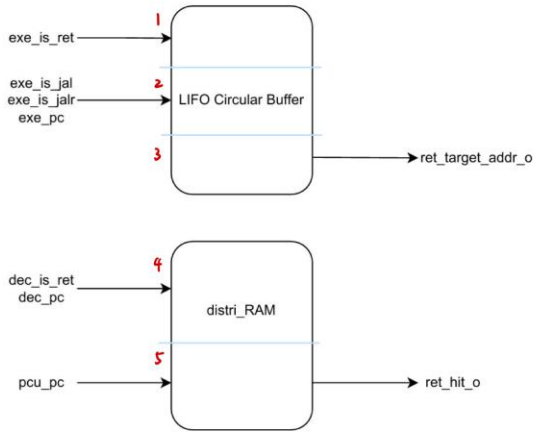

Fig 1. RAP Design

A. Overall design

I implement my return address prediction unit with a LIFO circular buffer, which stores the return addresses when jal instruction is decoded. Additionally, I store the address of ret instructions into a distri_RAM so that I can predict which instruction address contains ret instruction.

B. Popping the Circular Buffer (marked 1)

I use the signal in execution stage when pushing and popping the buffer because I found out that there are branch instructions right before ret or jal. If the branch is taken, the buffer will be accidentally pushed or popped, making the process executed in a wrong way (I do not check whether the predicted return address is correct or not).

C. Pushing the Circular Buffer (marked 2)

As described in the last paragraph, I used the signals in execution stage to decide whether I should push to the LIFO circular buffer. When I was testing my implementation, I found out that there are some jump and jr instructions, which should not affect the buffer when met. I added several conditions to when decoding so that they do not push the buffer by checking whether the rd_addr equals to 0. When the buffer should be pushed, I will push exe_pc + 4 to the buffer as the target return address.

D. Return Address (marked 3) and Buffer Design

For convenience, the ouput signal ret_target_addr is always pointing to the top element of the buffer. I have the top index stored in a register. When pushing to the buffer, the new address will be added to the buffer[top index], and the top index will be incremented by one. To make the buffer circular, I do mod operation when top index will be modified. I also add the item count to know when the buffer is empty.

E. Distri_RAM for Return Address (makred 4 & 5)

When an instruction is found to be ret in decode stage, I will cache the address into the RAM. This part is the same implementation as the one in the branch prediction unit.

Given an address from pcu_pc, the address will be checked whether it is a ret instruction by looking at the table. If hit, I will send the signal ret_hit_o to the program counter to update the pcu_pc for next iteration.

F. Other details

Following the BPU implementation, I delay the signal of branch hit and send it to the program counter so that it will not update when ret (which is included in jalr) is hit. Additionally, I also send it to pipeline control so that it will not be flushed unexpectedly.

IV. Result

With BUFFER_SIZE = 32 and ENTRY_NUM of Distri_RAM = 64, the coremark score increases from 100.739427 to 101.797748 (using the string library given by TAs and with BPU enabled).

V. Anaysis

With buffer size = 32, I did some experiment on the entry num of the distr_RAM. When setting entry_num to 2, the increase pertage of the score of CoreMark reached 0.93 percent. I speculate that the CoreMark program tends to use the same return address consecutively, making the RAP able to hit the return address even if there are only two addresses being stored in the table. I computed the hit rate when entry_num = 2, and the result shows that the hit rate is 87.7%, which shows that my guess is correct.

TABLE II. Score of Different EntryNums

ENTRY_NUM	**Metrics**	
	CoreMark Score	**Increase Percentage**
0	100.739427	0 %
2	101.677682	0.93 %
4	101.719025	0.97 %
8	101.729400	0.98 %
16	101.768741	1.02 %
32	101.797748	1.05 %
64	101.797748	1.05 %
128	101.808112	1.06 %

VI. Discusion

I implemented the RAP to get the address deterministicly instead of predicting the target address. Since I assume the LIFO buffer will always give the correct return address, I do not include any check on the misprediction (like the one in BPU). This is a little dangerous when implementing because if the push and pop of the stack is not controlled properly, the whole process will crash (this happened a lot during debugging). While it might cause a lot of errors during implementation, it will increase a lot compared to predicting the target address and verifying after the execution stage (like the one in BPU). If RAP is done in the similar implementation of BPU, there will be some mispredictions becaues the same function might be called by different functions, making the predictions to be wrong.