

HW#2 Return Address Predictor Design



Chun-Jen Tsai
NYCU
10/07/2024

Homework Goal

- ❑ This HW ask you to design a return address predictor for Aquila to improve CoreMark score. You must:
 - Understand the current BPU first
 - Design a return address stack (RAS) to predict the target address of a `jalr` instruction

- ❑ You have 3+ weeks to implement this homework
 - You must upload your initial report by 10/22, 17:00.
 - Students who have finished the design report get extra week to prepare the final report. You should upload the final report and your code by 11/1, 17:00.

Types of Branches

- ❑ There are forward types of branches
 - Conditional forward jumps: for if-then-else statements
 - Conditional backward jumps: used in looping statements
 - Unconditional jumps: for function calls, or from bad coding
 - Fixed target: `jal` always jumps to the same address
 - Moving target: `jalr` for function return or jump-table
- ❑ The problem of branches:

```
318: li    a0, 48
31c: jal   ra, 102c
320: addi   s1, s1, -1
324: bne   s1, s3, 318
328: li     s1, -1
32c: srli   a5, s0, 0x1f
330: add     s3, s0, a5
```

The next PC should be 0x102c, but
the Fetch unit does not know that until
two clocks later (after the Execute stage)

The Fetch unit does not even know what the
next PC should be until two clocks later (after
the Execute stage)

Dependencies of Fetch on Execute

- ❑ In a 5-stage pipeline, a branch instruction, after Fetch, may take up to two cycles to determine the next PC
 - The Execute is often responsible to determine the target PC
 - Patterson's textbook did this in the Decode stage!
 - Must we stall the Fetch stage by two cycles?
- ❑ A branch predictor predicts the PC before Execute determines the target
 - If the prediction is wrong, the pipeline has to be flushed before the Memory stage!

Static Branch Prediction

- ❑ Static branch prediction always make the same decision (forward/backward × taken/not taken)
- ❑ Implementation can be done by one of three methods
 - Hardwired into the processor pipeline
 - Assuming branch always taken, the Fetch must do a quick decode of the target PC
 - Assuming branch always not taken, then the $PC \leftarrow PC + 4$
 - Compilers generate the hint bit if the ISA supports it
 - Cooperation between the processor and the compiler, by following some register usage convention. For example,
 - “bne s1, s3, 318” suggests taken
 - “bne s1, s4, 318” suggests not taken

Dynamic Branch Prediction

- ❑ The processor collects statistics **at runtime** of whether every branch instructions are taken or not
- ❑ The fetch unit fetches the **predicted** next instruction
- ❑ In the case of a misprediction, the pipeline has to be flushed to re-fetch the correct instruction
 - The penalty is high for a misprediction
 - The CPU states has not been changed upon misprediction

Stage #Cycles	Fetch	Decode	Execute	Memory	Writeback
1	BR	?	?	?	?
2	ADDI	BR	?	?	?
3	SLL	ADDI	BR	?	?
4	?	NOP	NOP	NOP	?

→ Next PC determined!

Branch Prediction Schemes

❑ Local-history Predictor

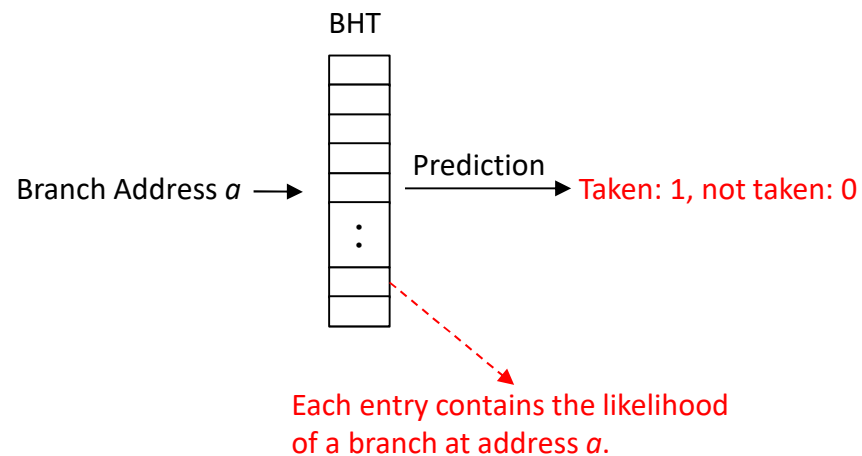
- Uses a Branch History Table (BHT) indexed by the recent branch addresses
- When the fetch unit reaches a branch location, it gets the PC for the next instruction to fetch based on the BHT

❑ Advanced Predictors

- Gshare, 1993: simplest predictor with global history
- TAGE, 2011: the best-performing predictor class
- Return address predictor for function call returns

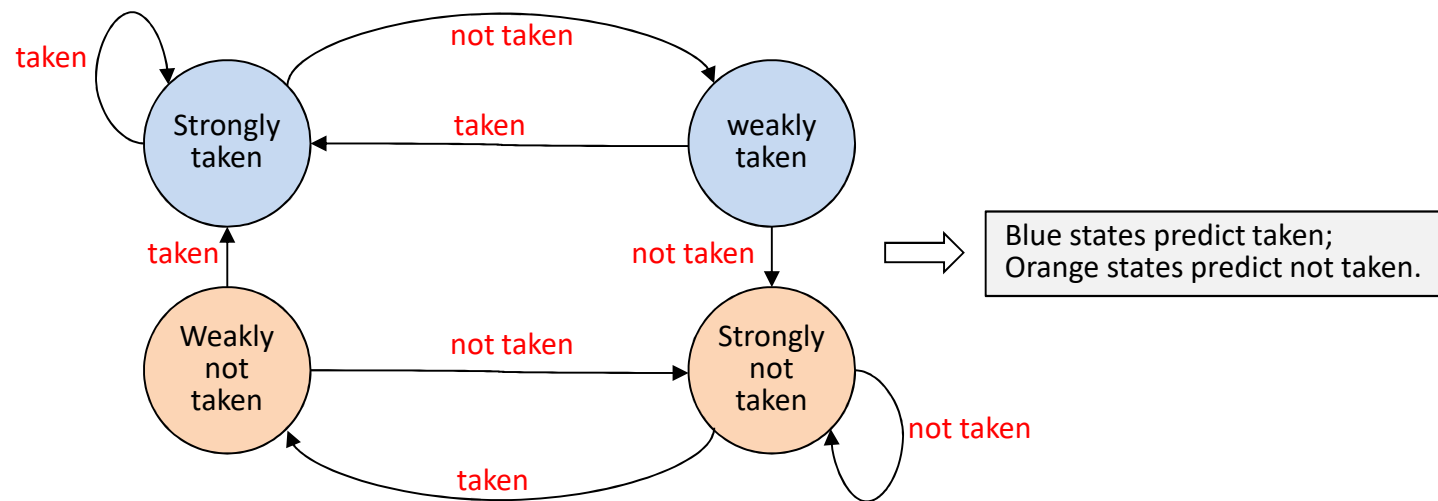
One-level Branch Predictor (1/2)

- ❑ One-level predictor is based on bimodal assumption:
 - The branch at a specific PC is either taken or non-taken most of the time
- ❑ For one-level branch predictor, we must determine:
 - How many address bits are used to index the BHT?
 - How many bits are used for the branch statistics (likelihood)?



One-level Branch Predictor (2/2)

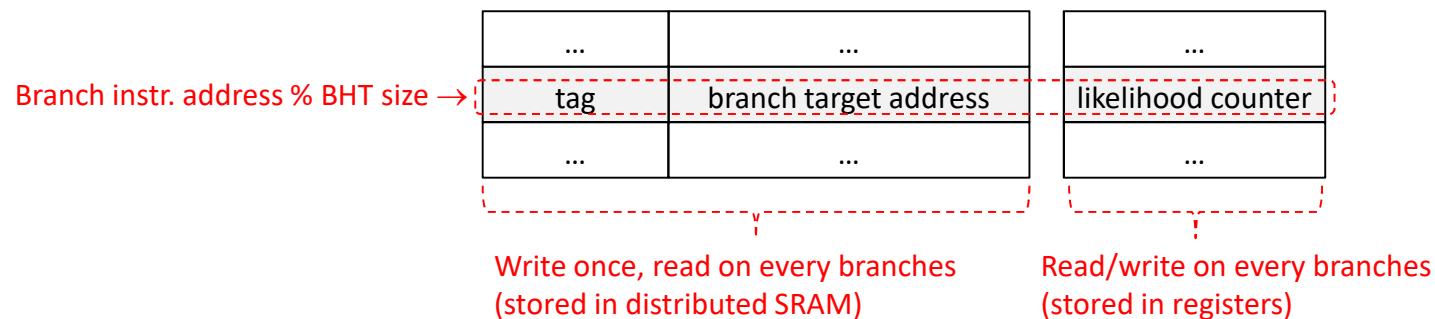
- ❑ Aquila implements the simple 2-bit predictor
 - For each branch instruction, we record its branch likelihood with one of four possible states:



- The state is updated after the execute stage determines whether the branch is taken or not.

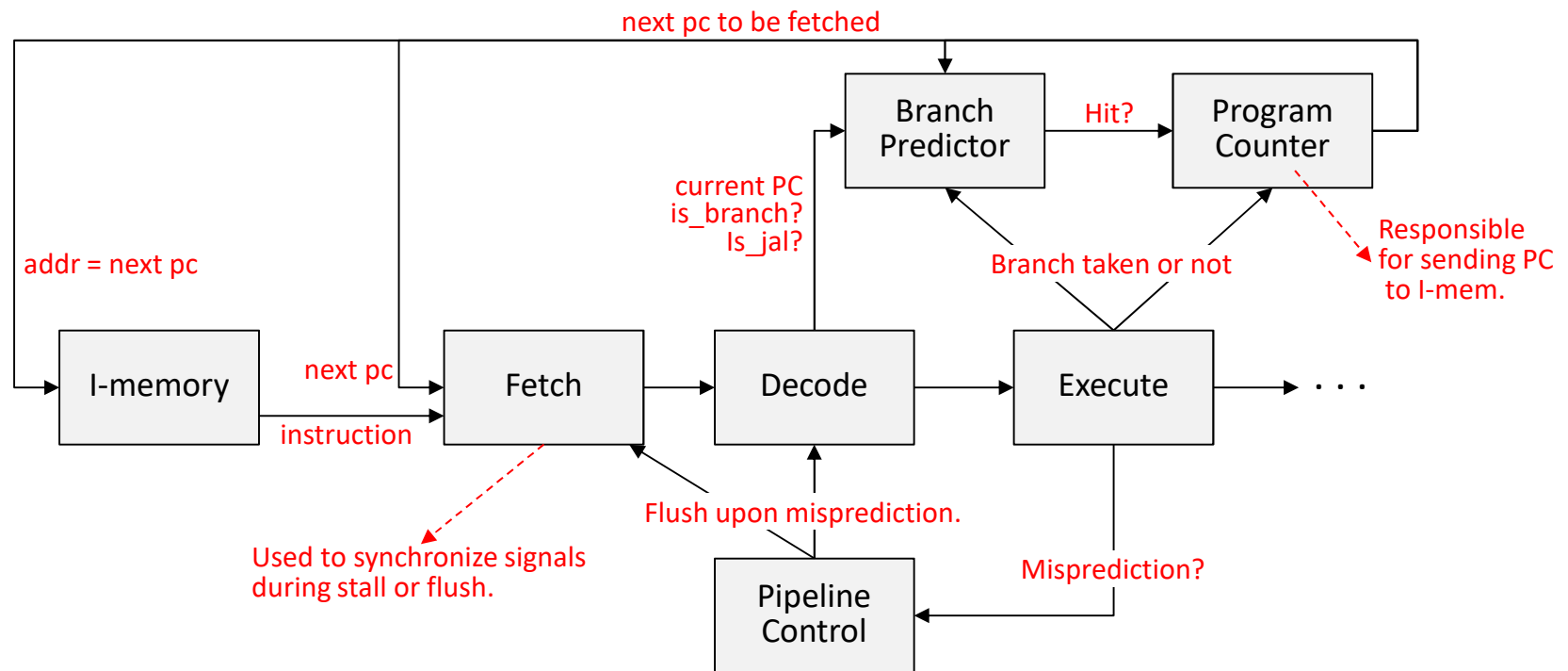
BHT Implementation

- ❑ The BHT is a tagged memory similar to caches
 - Typically, direct-mapping by the branch instruction address
 - The least significant bits is used as the index to the BHT (e.g. 5 bits for 32 table entries)
 - The full address is used as the tag to determine hit or miss
 - As an alternative, expensive fully-associative BHT can be used for better performance at higher circuit cost
- ❑ BHT table entry format:

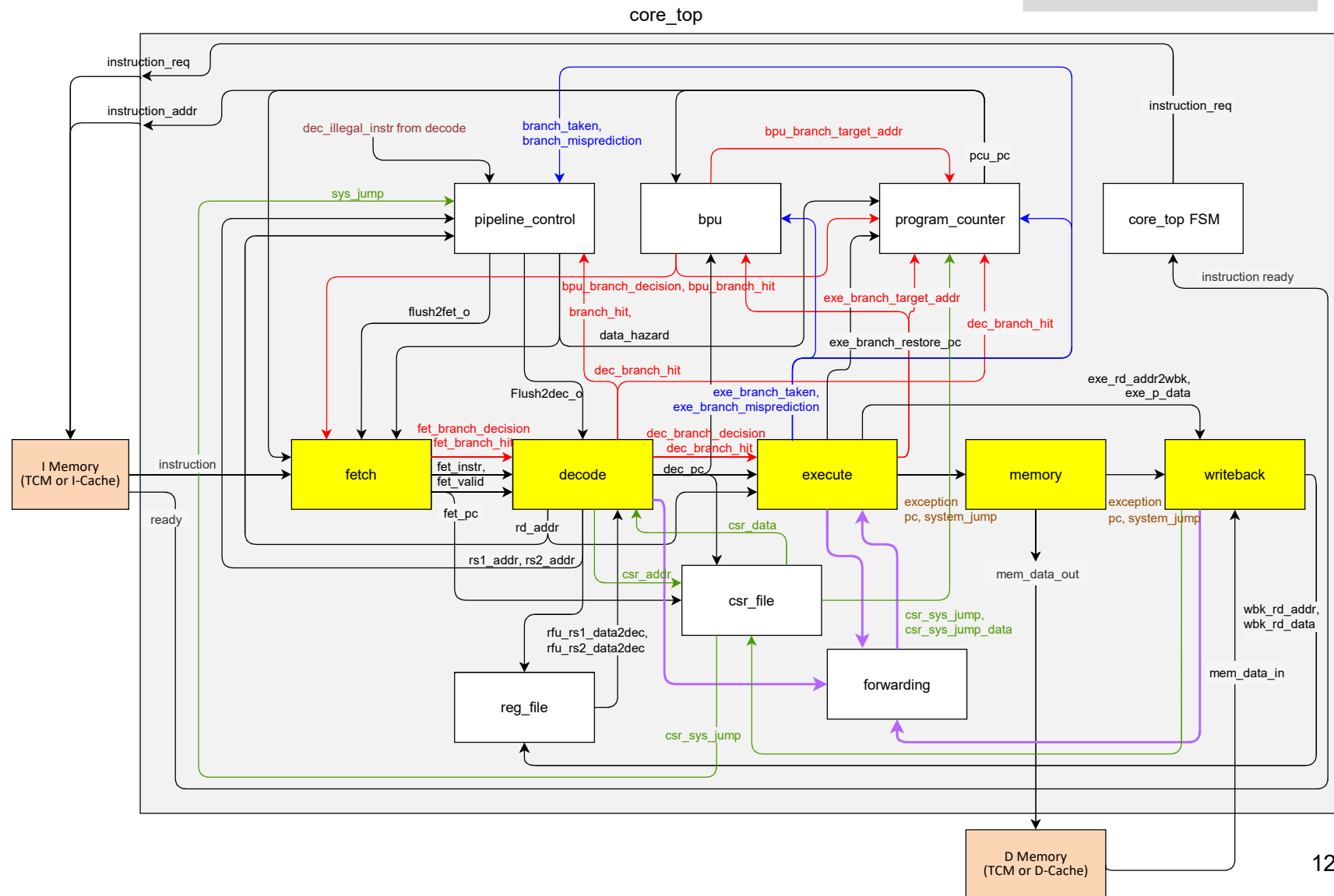


Branch Prediction Flow in Aquila

- ❑ A branch predictor tells the fetch unit which instruction to fetch before the branch has been executed
 - Aquila only predicts the target for branches and `jal`, not `jalr`

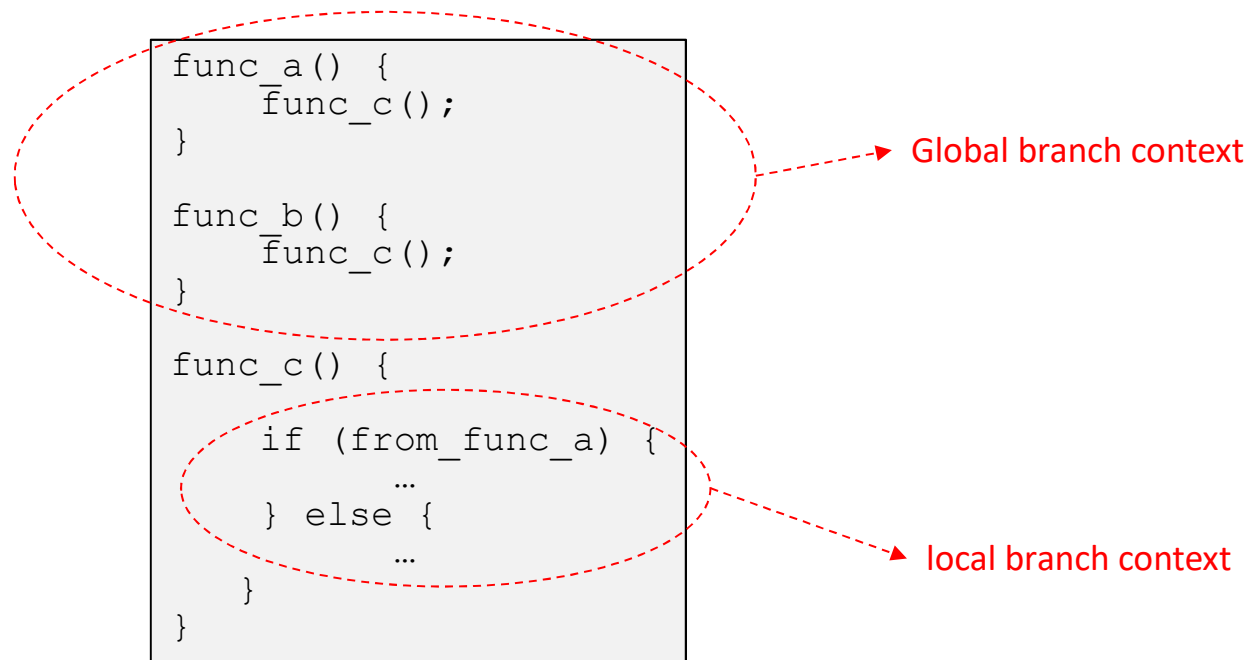


Aquila BPU Related Signals



Basic Ideas on Two-Level BPU

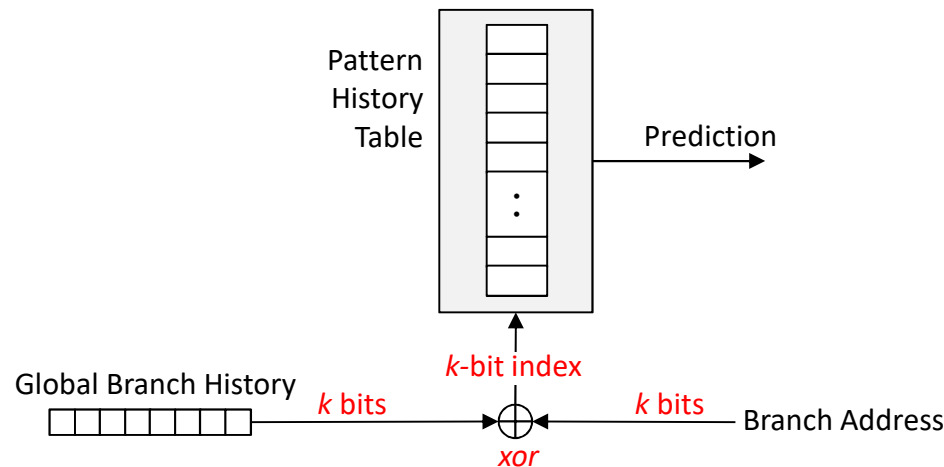
- ❑ One-level predictors lack global context information



- ❑ 2-level predictors consider both global & local contexts

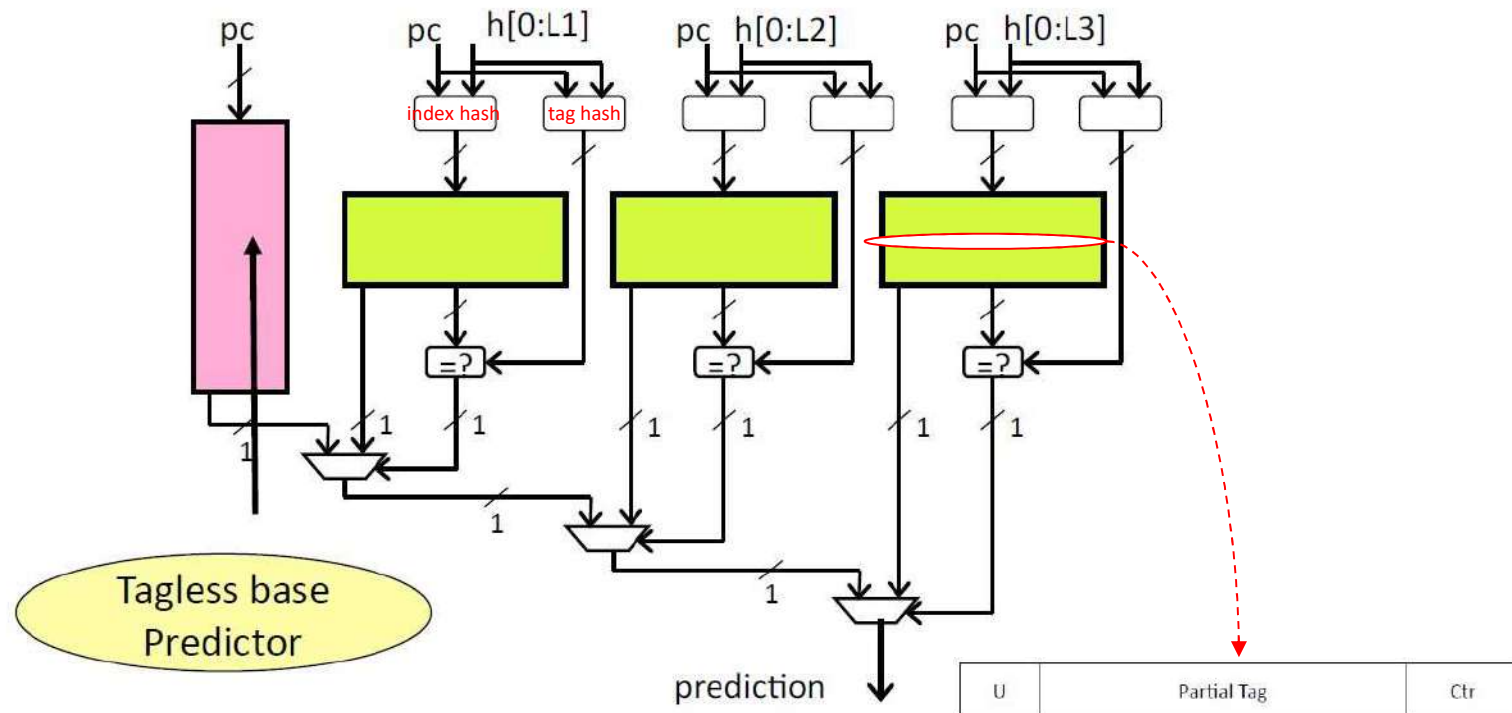
Two-level Branch Predictor

- ❑ A branch depends on:
 - History of the same branch – Local Branch History
 - Nearby branches – recorded using Global Branch History
- ❑ gshare: XOR address & history into the index bits
 - k-bit for global history would be a waste of registers
 - Global history and local history can share the index bits:



TAGE Branch Predictors (Multi-Level)

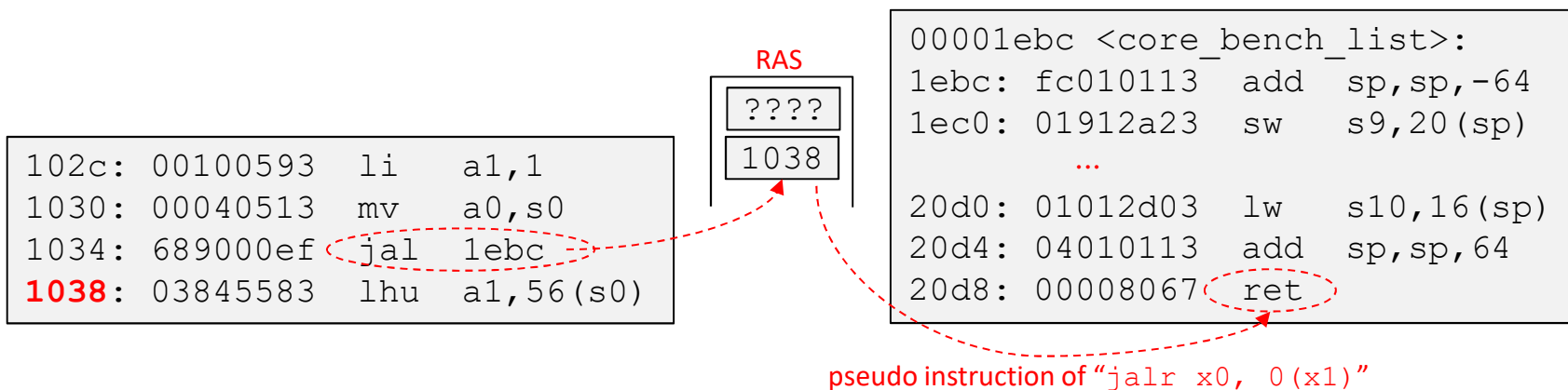
- The state-of-the-art branch predictor is TAGE:



André Seznec, "A New Case for the TAGE Branch Predictor," *MICRO 2011 : The 44th Annual IEEE/ACM Int. Symp. on Microarchitecture*, Dec. 2011.

The Return Address Predictor (RAP)

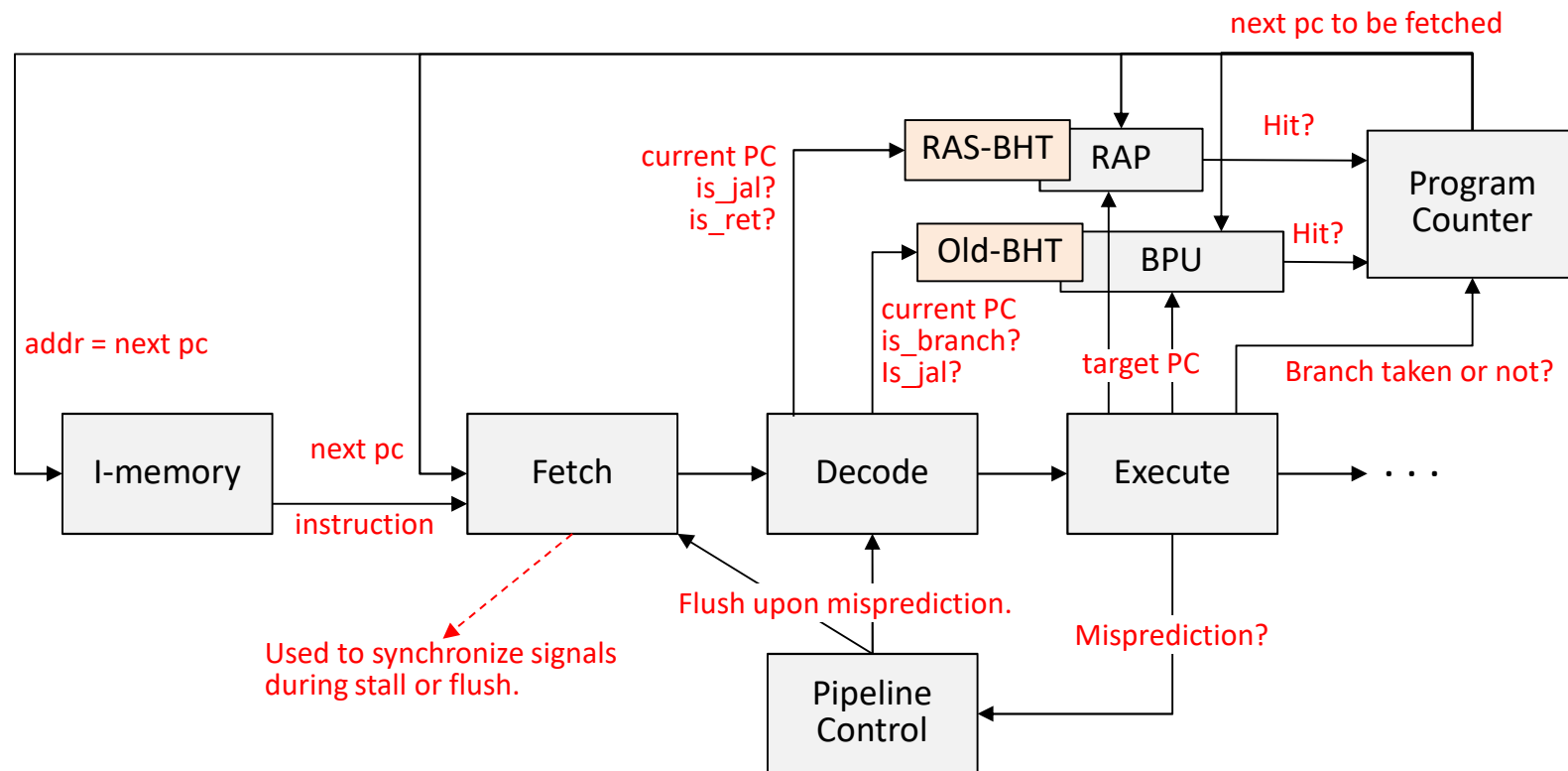
- ❑ To handle function return properly, we must maintain a stack of return addresses:



- ❑ Note: the return address stack (RAS) does not have to be too deep, a small circular buffer is preferred
 - Stack structure can be overflowed due to tail-call optimization
 - A circular LIFO buffer fixes overflow problem

Minimal Implementation of RAP

- ❑ A minimal implementation can be done as follows:



Your Homework (1/2)

❑ Part 1: analysis & design report:

- Study the branch predictor in Aquila
 - Disable BPU or change BHT size see what happens to CoreMark
- Analyze the branch statistics (e.g. hit rate and miss rate for different types of branches) of CoreMark
- Describe your architecture for return address prediction
- Your report on this part should be no more than two pages.

Your Homework (2/2)

- ❑ Part 2: Return address predictor
 - You shall at least implement the minimal RAP in page 17
 - Try other ways to improve the minimal implementation
 - Conduct experiments on different RAP techniques you have tried and discuss the results
 - The overall report size should be no more than three pages