CSED311 Computer Architecture – Lecture 10

Pipelined CPU – Branch Prediction

Eunhyeok Park

Department of Computer Science and Engineering
POSTECH

Disclaimer: Slides developed in part by Profs. Austin, Brehob, Falsafi, Hill, Hoe, Lipasti, Martin, Roth, Shen, Smith, Sohi, Tyson, Vijaykumar, and Wenisch @ Carnegie Mellon University, University of Michigan, Purdue University, University of Pennsylvania, University of Wisconsin and POSTECH.

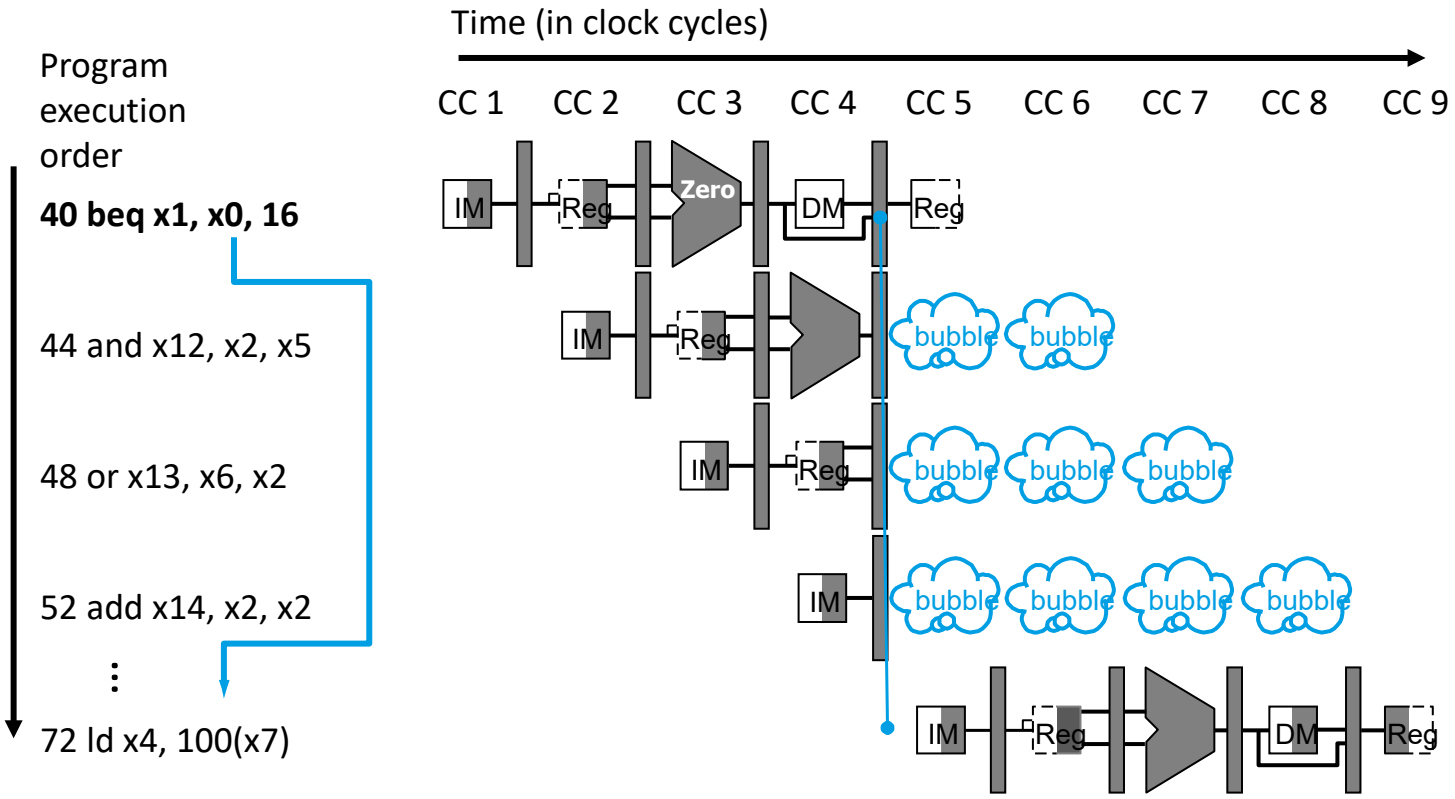
Review: PC Hazard w/o Branch Prediction

	R/I-Type	LD	SD	Bxx	JAL	JALR
IF	use	use	use	use	use	use
ID	produce	produce	produce			
EX						
MEM				produce	produce	produce
WB						

- All instructions read and modify PC
- PC hazard distance is at least 1
  - The next PC value cannot be determined until the current instruction is decoded
- Does that mean we must stall after every instruction?

Review: Simplest Branch Prediction: PC+4

- Expect “nextPC = PC+4” ~86% of the time
- Wrong predictions 14% of the time → misprediction penalty (pipeline flush → bubbles)



Review: PC Hazard – Always Predict Not Taken

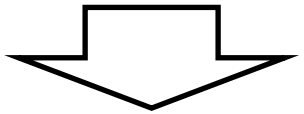
	R/I-Type	LD	SD	Bxx	JAL	JALR
IF	use (produce)	use (produce)	use (produce)	use	use	use
ID				produce	produce	
EX						produce
MEM						
WB						

- Still, hazard distance on a taken branch is at least 1. Why?
- Hazard distance is greater in modern CPUs. Why?

Review: Performance Impact (Predict PC+4)

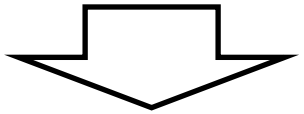
- Correct guess → No penalty ~**86%** of the time
- Incorrect guess → bubbles (branch/jump resolved in MEM stage in the baseline)
- Assume
  - No data hazards
  - 20% are control flow instructions
  - 70% of control flow instructions are taken
  - IPC = 1 / [1 + (0.2 * 0.7) * 3] = 0.70

Guess always NOT taken



Branch resolved in EX stage

  - IPC = 1 / [1 + (0.2 * 0.7) * 2] = 0.78 (+11%)



Branch resolved in ID stage

  - IPC = 1 / [1 + (0.2 * 0.7) * 1] = 0.88 (+13%)

How to reduce this?
(probability of a wrong guess)

Making a Better Guess

Guess always taken

- For ALU instructions
  - Can't do better than guessing nextPC=PC+4
  - Still tricky since we must guess nextPC before the current instruction is fetched
- For Branch/Jump instructions
  - Need to guess two things – branch "direction" and branch "target" address

(1) Why not always guess in the taken direction since 70% correct?

(2) Then, where to jump? (i.e., what is the branch target?)

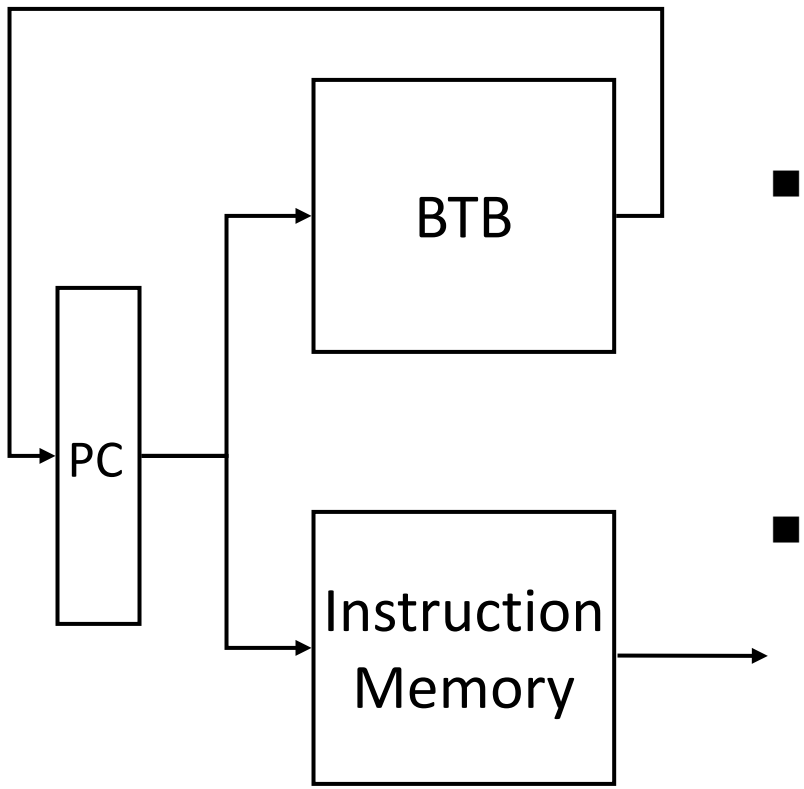
  - The branch target is encoded in the branch instruction, which is not fetched yet

→ Must make a guess based only on the current PC !!!

Fortunately,

- The same instruction will be executed repeatedly in a program
- PC-offset branch/jump target is static (except for JALR)
- We are allowed to be wrong some of the time
  (wrong prediction → pipeline flush → performance loss but still *functionally* correct)

Branch Target Buffer (Oracle)



- BTB (Oracle)
  - A giant table indexed by PC
  - Returns the guess for nextPC
- When encountering a PC for the first time, store in BTB
  - PC + 4 if ALU/LD/ST
  - PC + offset if Branch or JAL
  - ?? if JALR
- Effectively guessing branches are **always taken**

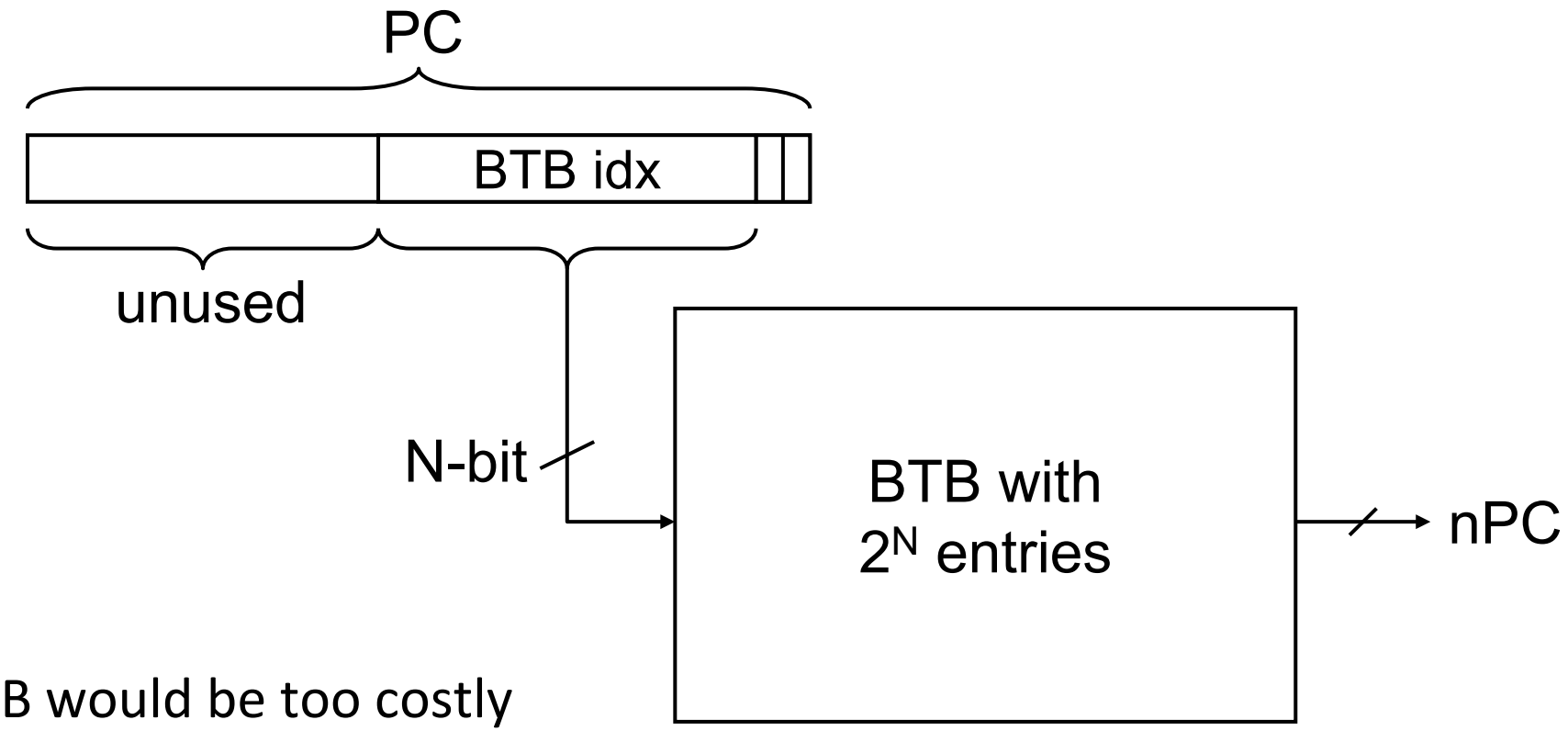
IPC = 1 / [1 + (0.2*0.3)*2]

= 0.89

Assume misprediction penalty of 2 cycles

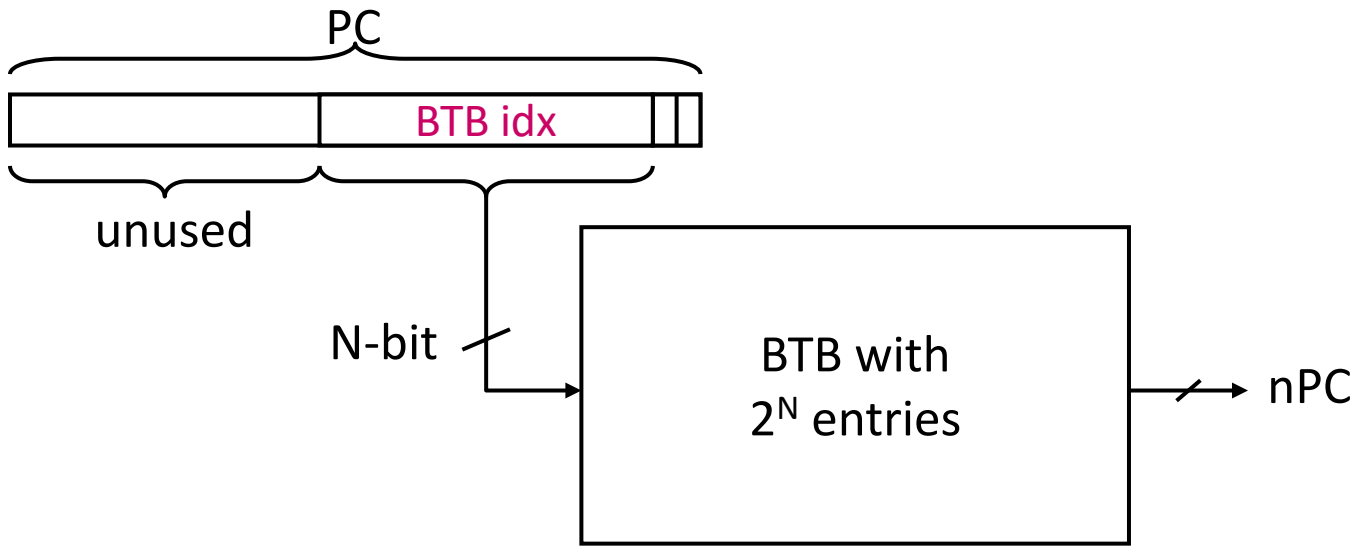
Guessed taken but not actually taken

Branch Target Buffer (Reality)



- Oracle BTB would be too costly
- "Hash" PC into a 2^N entry table
- On collision, BTB returns something meaningless and possibly wrong
  (since 80% of entries all hold PC+4)
- Large BTB reduces collision, but a large BTB has drawbacks, too (what drawbacks?)

When a Collision Happens?



- Assume BTB index is 10 bit → 1024-entry BTB

PC: 00000000000000000000 0011001100 : ADD → PC+4?

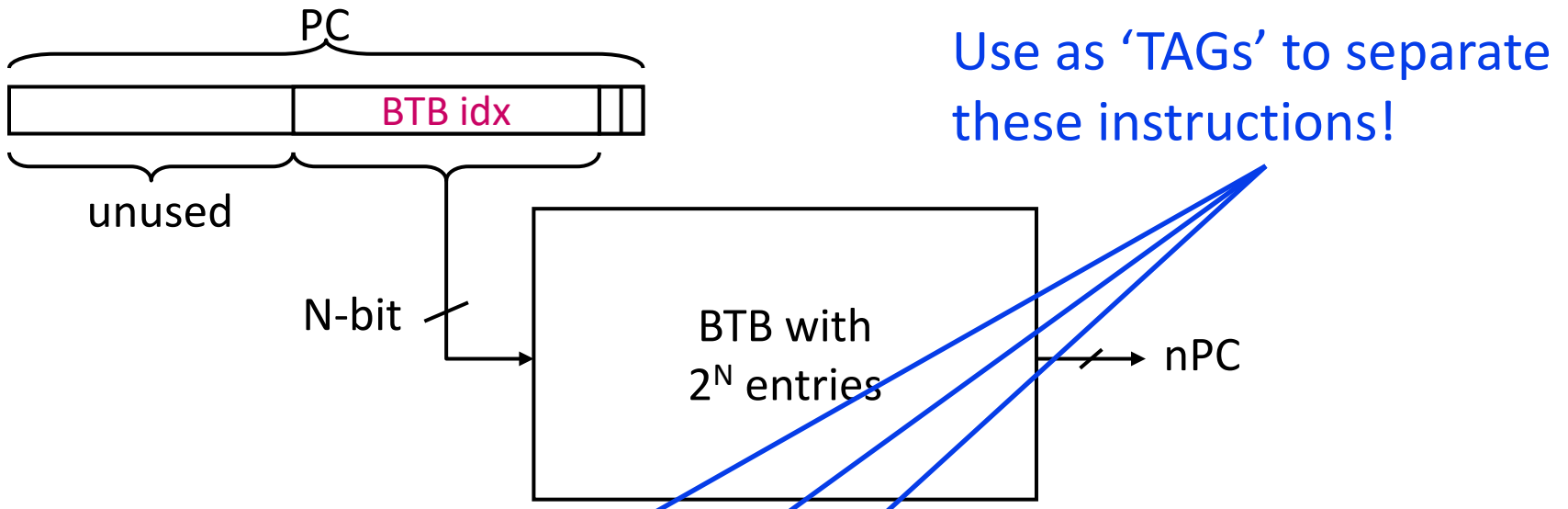
VS

PC: 11111……11111111111111 0011001100 : JAL → A target?

VS

PC: 11100……0011…..000….110 0011001100 : SUB → PC+4?

What to do about Collisions?



- Assume BTB index is 10 bit → 1024-entry BTB

PC: 00000000000000000000 0011001100 : ADD → PC+4?

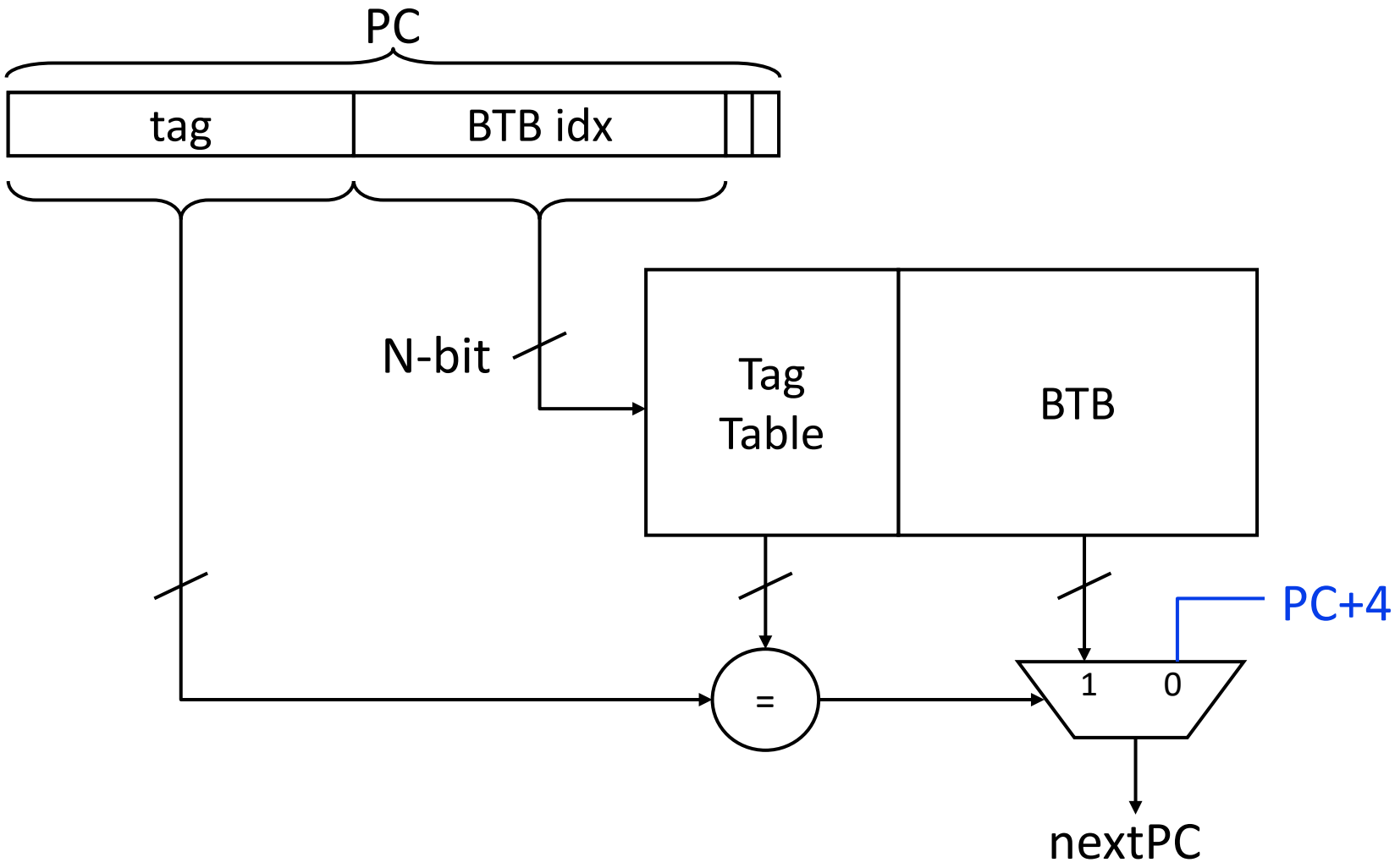
VS

PC: 11111……1111111111111 0011001100 : JAL → A target?

VS

PC: 11100……0011…..000….110 0011001100 : SUB → PC+4?

Tagged BTB



Let’s store branch/jump instructions only (to save ~80% storage)!

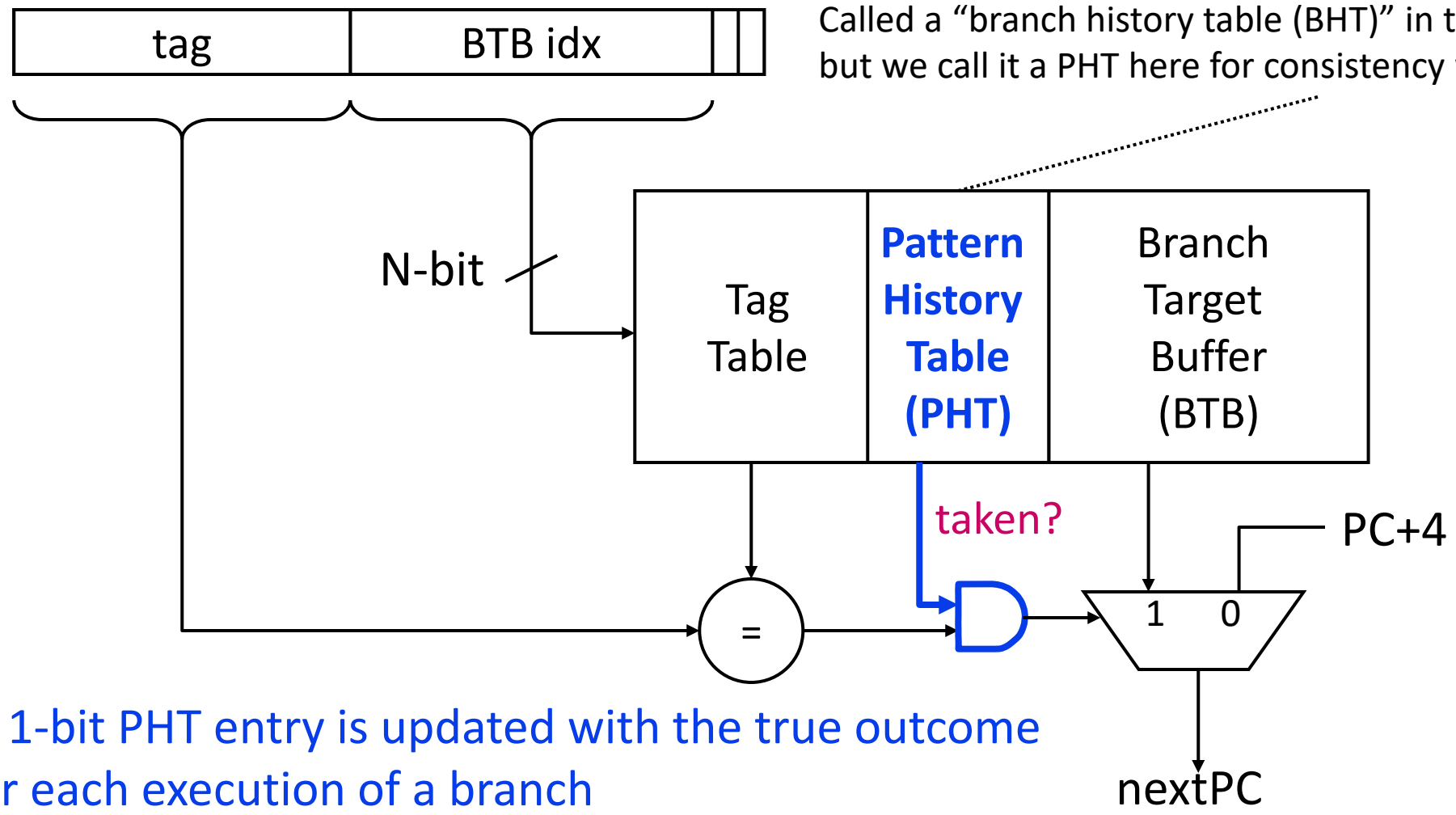
Update tag and BTB for the new branch after each collision

More Advanced (Direction) Prediction?

- We can get 100% correct on non-branch instructions
- For branch instrs, so far, we assumed either “always not taken” or “always taken”
  - Always not taken → ~30% correct prediction
  - Always taken → ~70% correct prediction
- Can we do better? What patterns can we leverage on forward branches?
- A given static branch instruction is likely to be highly biased in one direction (either taken or not taken)
  - **If it was taken last time, maybe will take this time as well** (or vice versa)
    (e.g., backward branch to the beginning of a loop)
  - Typically, ~**85%** correct prediction if we always guessed **the same outcome as the last time** the same branch was executed

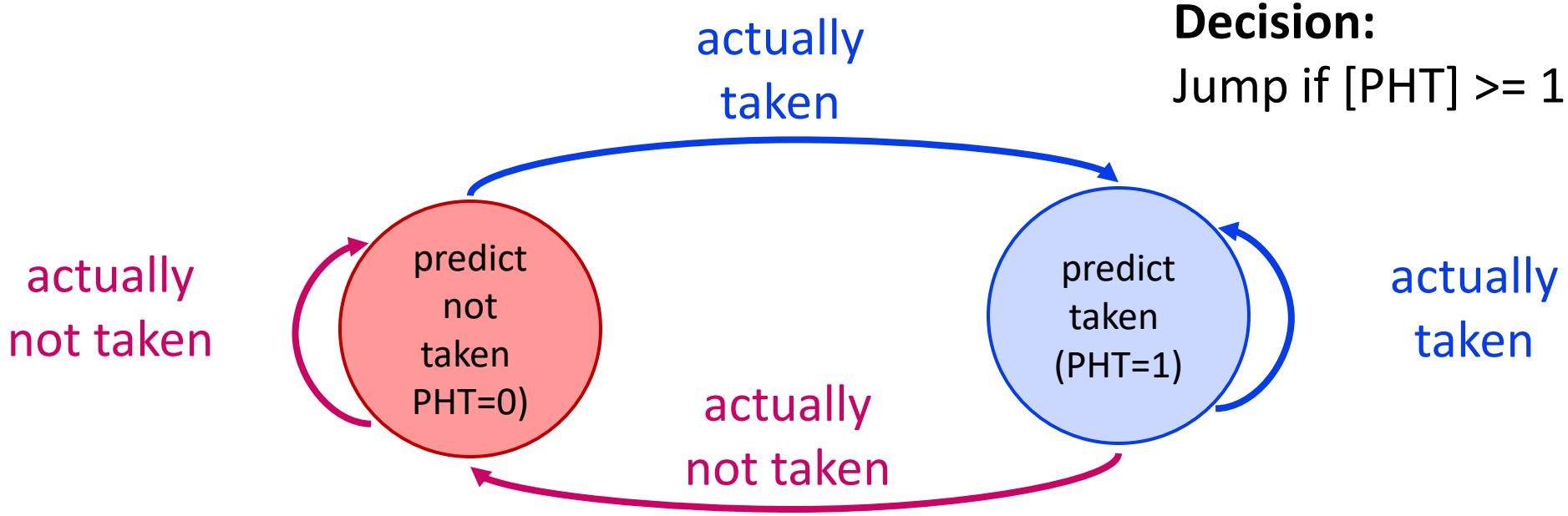
    → IPC = 1 / [1 + (0.2*<u>**0.15**</u>) * 2] = 0.94

Pattern History Table and Target Buffer



- The 1-bit PHT entry is updated with the true outcome after each execution of a branch
- Only taken branches and jumps are held in BTB
- Large or sophisticated PHT can be designed separately from the BTB

Branch Prediction State Machine (1-bit)



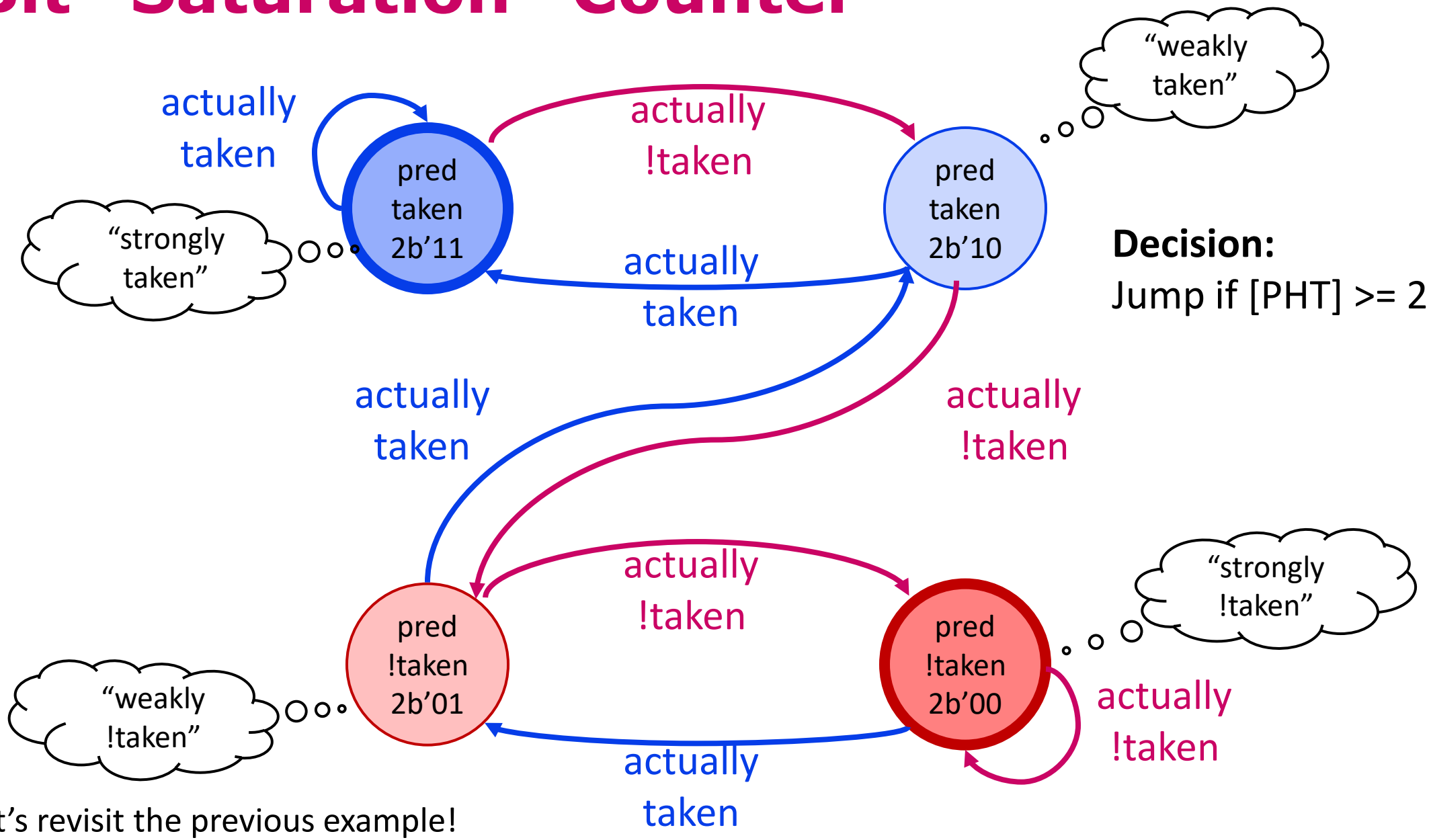
What is prediction accuracy for

1) T → NT → T → NT → T → NT → T → NT → ..

2) T → T → NT → T → T → NT → T → T → NT → ..

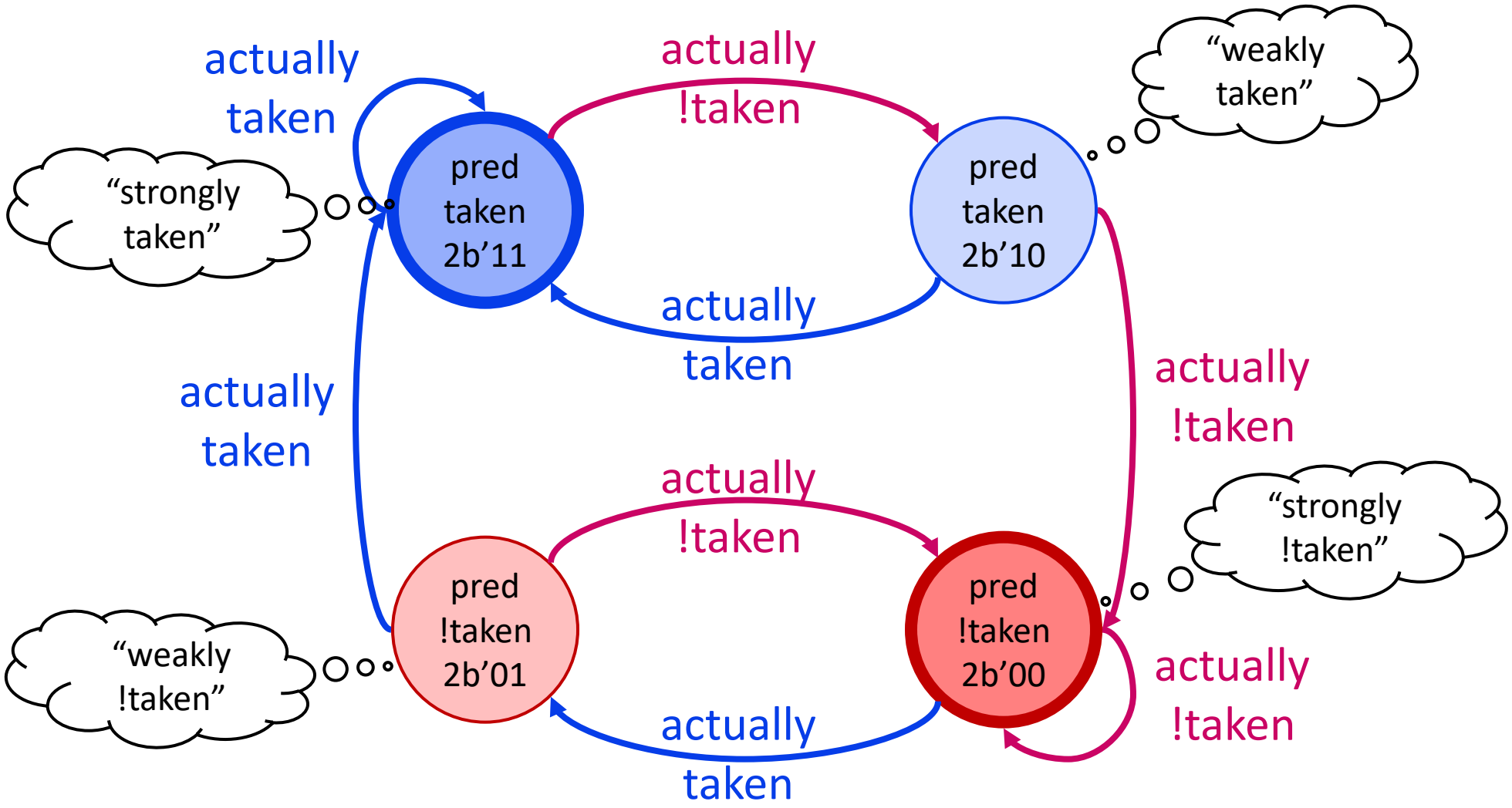
It's even worse than "guess always taken" in these cases

2-Bit "Saturation" Counter



Let's revisit the previous example!

2-Bit "Hysteresis" Counter

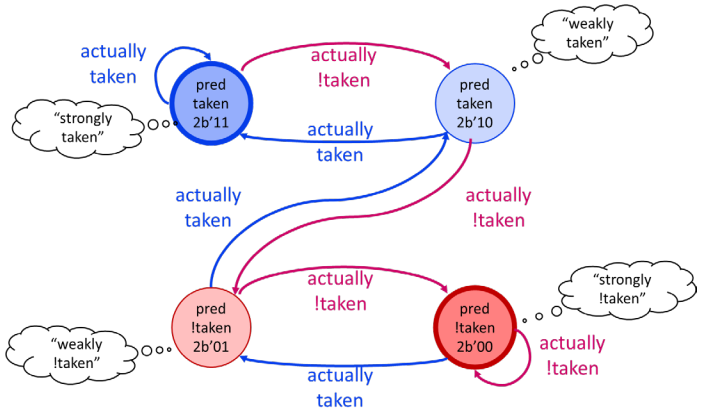


Once you change your position, you don't change it back easily.

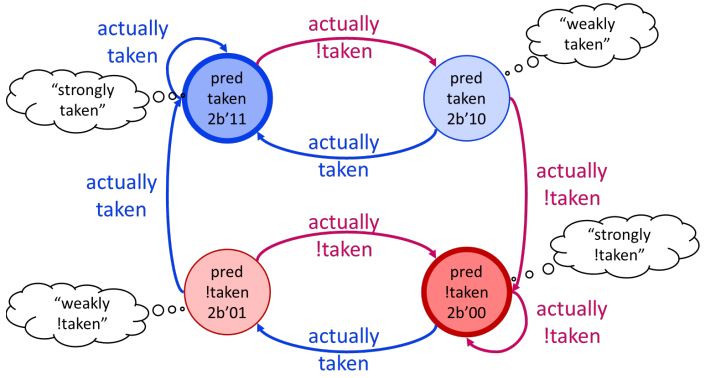
Exercise: revisit the earlier example with this predictor.

Example #1

- 2-bit saturation vs. 2-bit hysteresis
  - 2-bit saturation
    - outcome) T T N N T T N N T T N N
    - correct?) -
    - current counter) 0
    - prediction) N

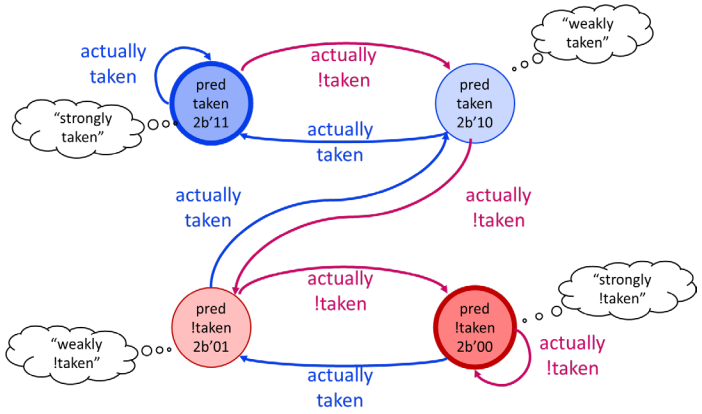


  - 2-bit hysteresis
    - outcome) T T N N T T N N T T N N
    - correct?) -
    - current counter) 0
    - prediction) N

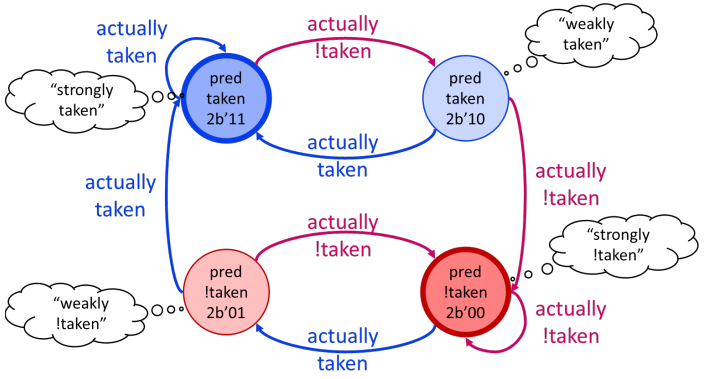


Example #2 (on your own)

- 2-bit saturation vs. 2-bit hysteresis
  - 2-bit saturation
    - outcome) T T N T N T N T N T N T
    - correct?) -
    - current counter) 0
    - prediction) N



  - 2-bit hysteresis
    - outcome) T T N T N T N T N T N T
    - correct?) -
    - current counter) 0
    - prediction) N



Different predictors can predict different patterns!

State-Machine-Based Predictors

- A 2-bit predictor can get >90% correct
  - IPC = 1 / [1 + (0.20*0.10) * 2] = 0.96
  - Any “reasonable” 2-bit predictor does about the same
- Major branch behaviors exploited
  - Mostly do the same thing repeatedly (>80%)
    - 1-bit and 2-bit predictors equally effective
  - Occasionally do the opposite once (5~10%) – example?
    - 2 mispredictions with a 1-bit predictor
    - 1 misprediction with a 2-bit predictor
  - Miscellaneous (<10%)
    - Some could be captured with more advanced predictors
    - What does Amdahl’s law say about this?
    - However, modern processors have advanced predictors. **Why?**

"Global" Path History

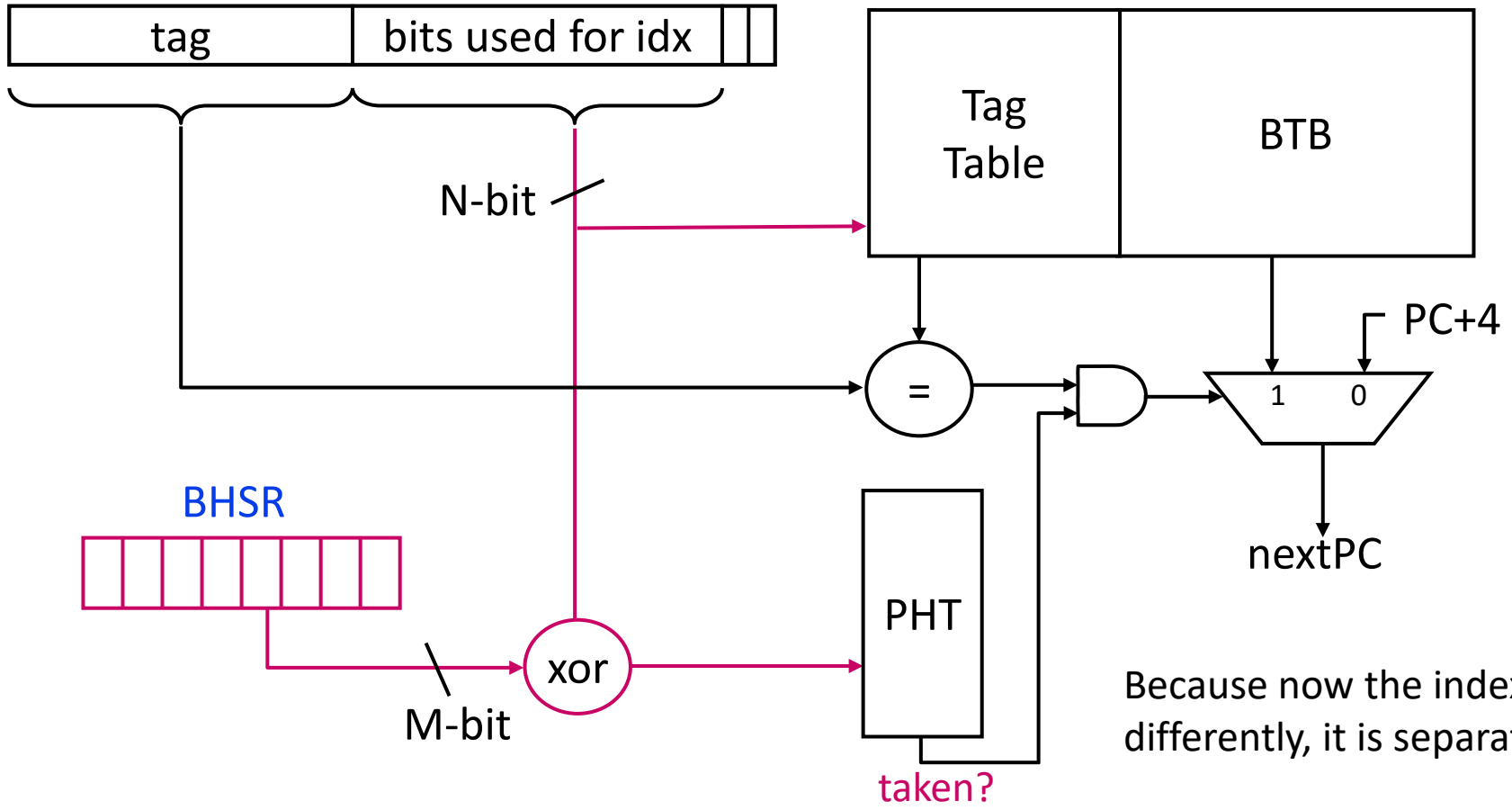
- So far, we focused on each branch's behavior
- But branch outcome often correlated to other branches
- Example

```
aa = bb = 1;
if (x > 1)                    ;; C1
        aa=0;
if (y > 1)                    ;; C2
        bb=0;
if (aa == 1 && bb == 0) {     ;; C3
        ....
}
```

→If C1 is FALSE (aa=1@C3) and C2 is TRUE (bb=0@C3), then C3 is certainly TRUE.

How do you capture this information?

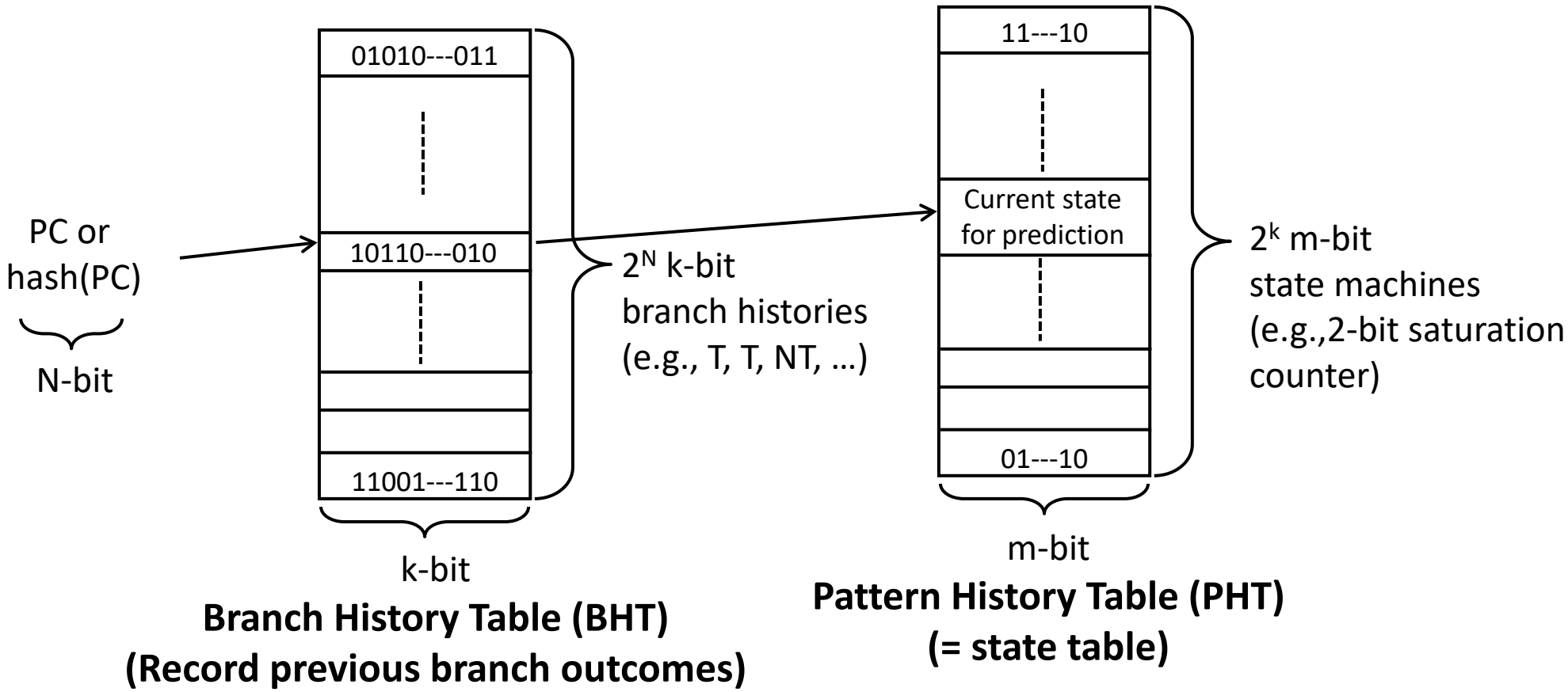
"Gshare" Branch Prediction [McFarling]



Because now the index for PHT is calculated differently, it is separated from BTB.

- Global BHSR (Branch History Shift Register) tracks the outcomes of the last M branch instructions (e.g., NT→ T …, where T is 1 and NT is 0)
- Would a global BHSR be sufficient?

Two-level Branch "Direction" Predictors



Variations of this predictor can predict many different patterns (local & global patterns)

Two-level Branch "Direction" Predictors

- BHT (branch history table) + PHT (pattern history table)
  - BHT
    - G : global
    - S : per-set (of addresses)
    - P : per-address
  - PHT
    - g : global
    - s : per-set (of addresses)
    - p : per-address

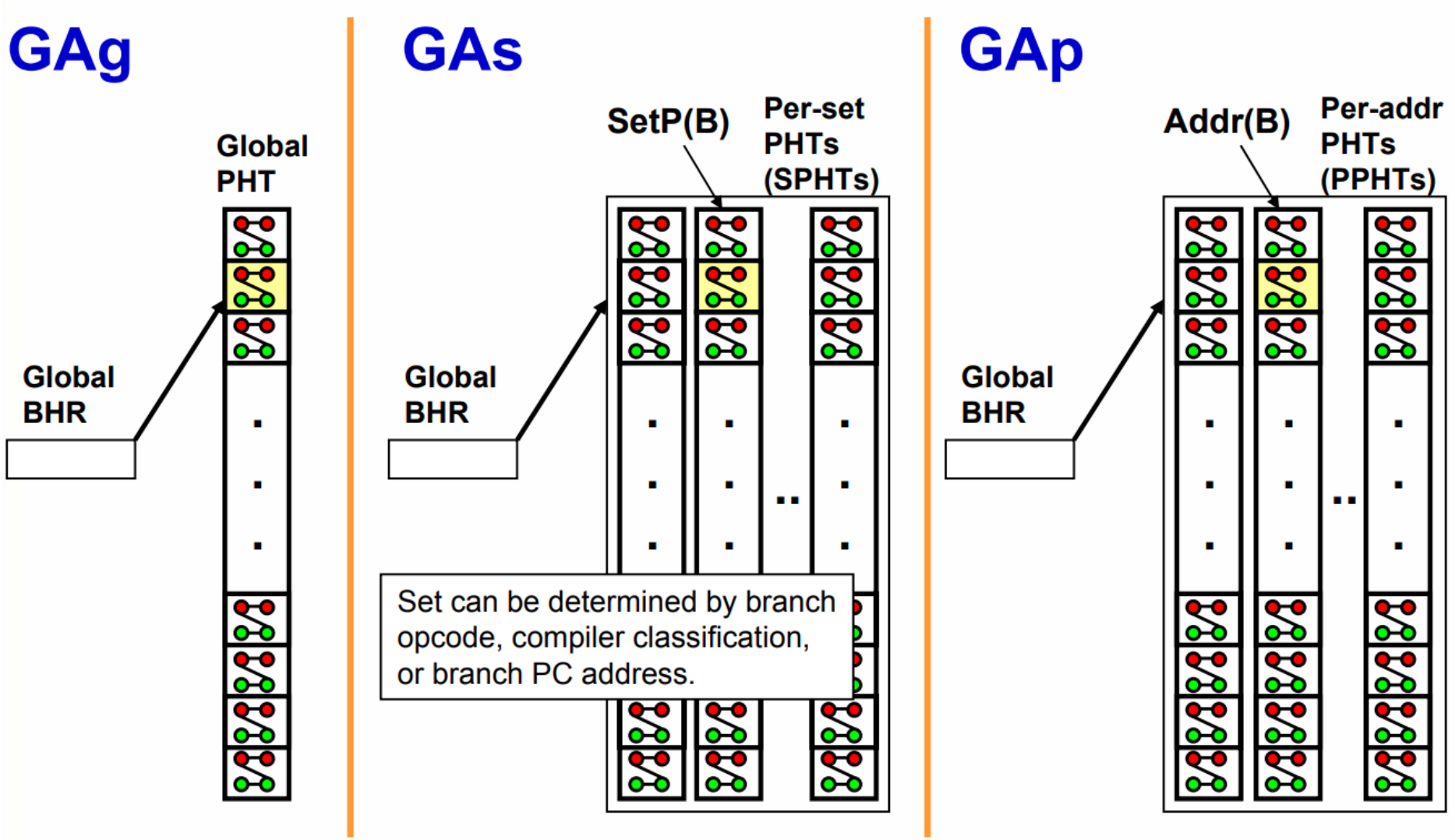
9 combinations possible ("A": Adaptive):
GAg, GAs, GAp,
SAg, SAs, SAp,
PAg, PAs, PAp,

- Make sure you understand GAg, GAp, PAg, and PAp!
  (S is just a group of addresses)

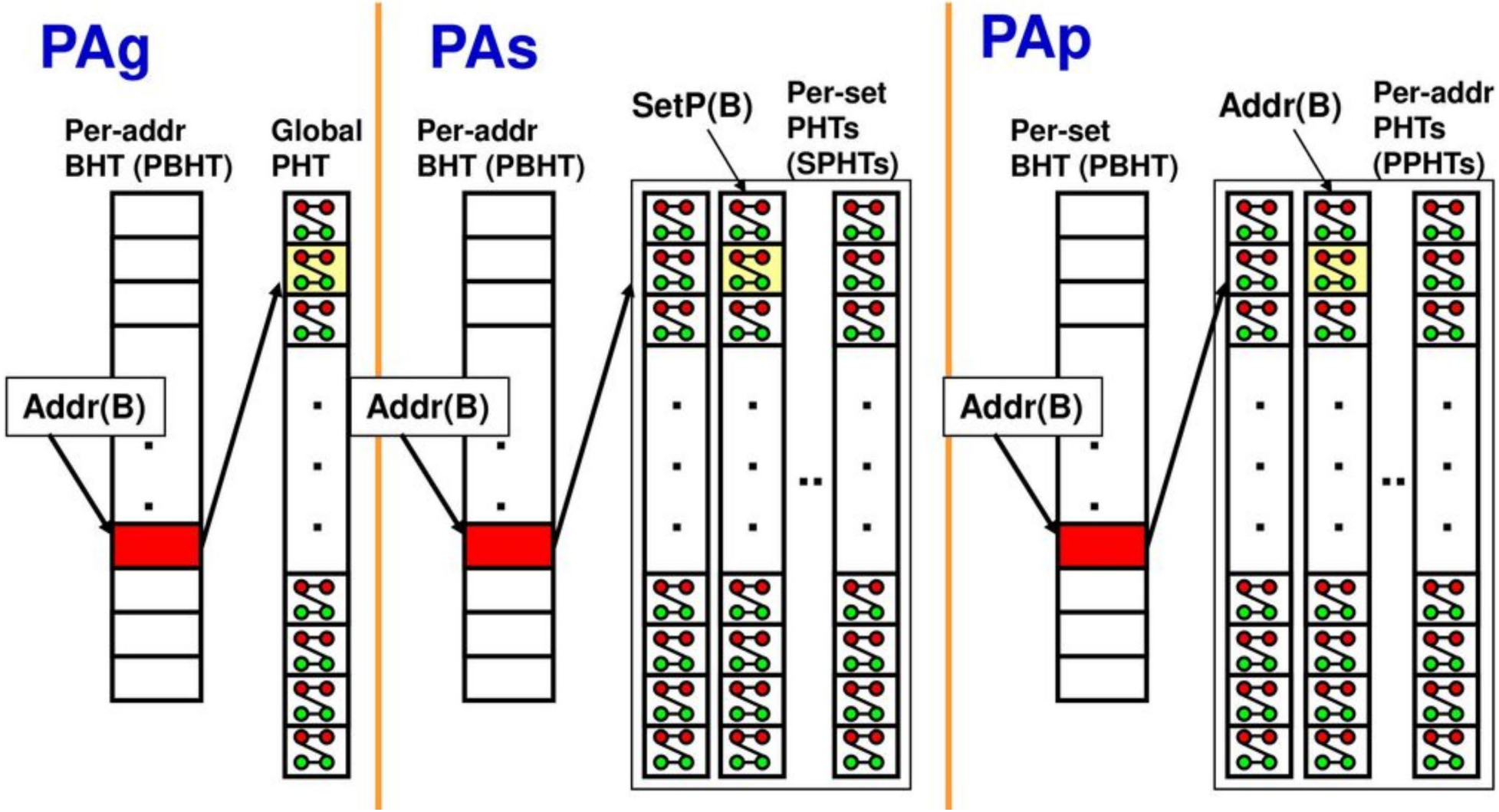
Reference

T.Y. Yeh, and Y.N. Patt. A Comparison of Dynamic Branch Predictors that use Two Levels of Branch History. In Proceedings of the International Symposium on Computer Architecture (ISCA), pp.257-266, 1993.

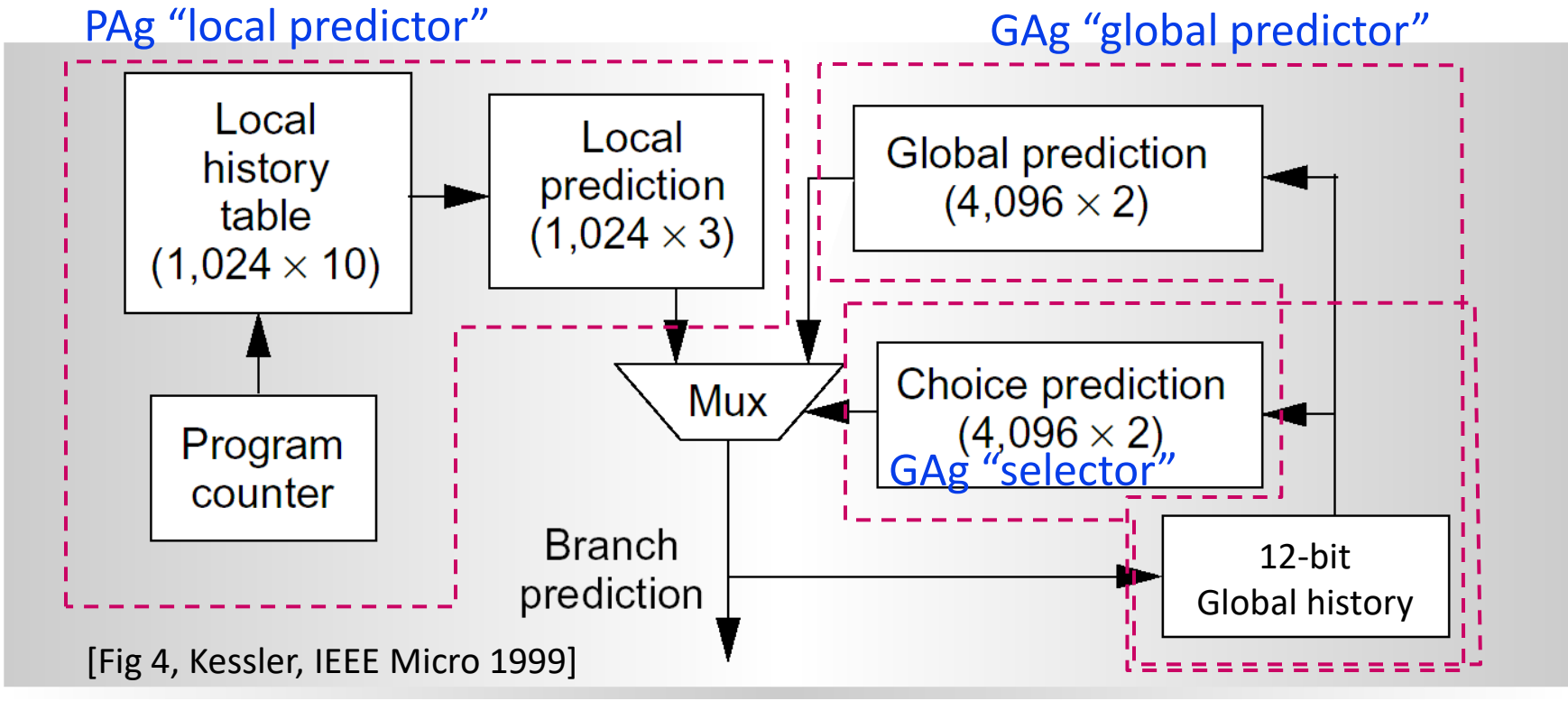
Global History Schemes



Per-Address History Schemes



Alpha 21264 Tournament Predictor [DEC1996]



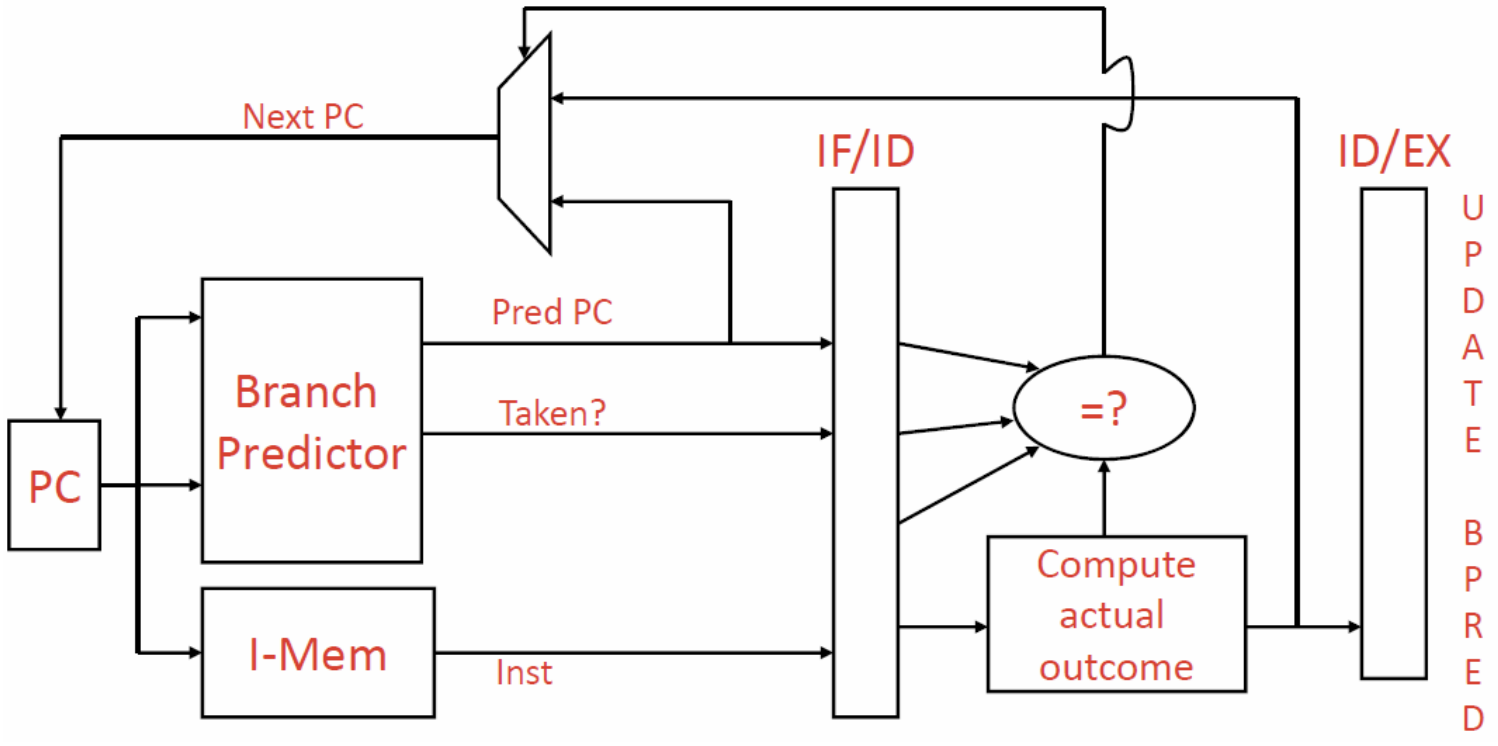
[Fig 4, Kessler, IEEE Micro 1999]

- Make separate predictions using local history (per branch) and global history (correlating all branches) to capture different branch behaviors
- A meta-predictor decides which predictor to believe

Better than 97% correct!!

Branch Predictor in a Pipeline

- "Trust, but verify"
  - Fetch with nextPC, but compute actual branch outcome
- Update branch predictor (PHT and BTB)
  - What if you have branches in wrong-path streams (for which you don't know actual outcomes yet)?



Speculative Execution Summary

- Each control flow instruction must carry **the predicted nextPC** down the pipeline
- When the control flow outcome of an instruction known (=branch resolution), the predicted nextPC is checked
- If nextPC was predicted correctly
  - Update PHT (=prediction state logic → reinforce prediction)
  - Do nothing more
- if nextPC was predicted incorrectly
  - Update PHT and/or BTB
  - Flush all “younger” instructions in the pipeline
  - Restart fetching at the correct target

Relatively easy for in-order pipelines like 5-stage RISC-V as you know. However, it can become **very difficult** in deeply pipelined, superscalar, out-of-order pipelines (e.g., Intel i7)

Return Address Stack

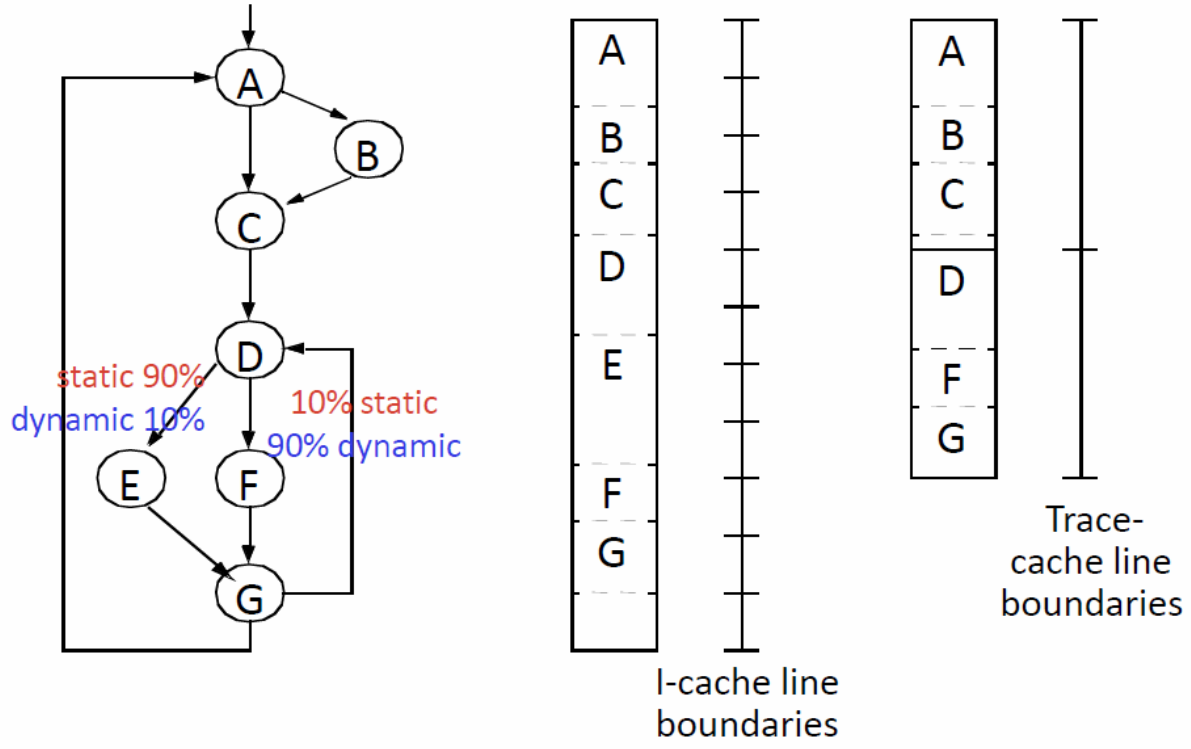
- The targets of register-indirect jumps (i.e., JALR) have little locality
  - History-based predictors don't work. Why?
  - But a simple "stack" can capture the usage pattern of function call and return very well
- Return Address Stack (RAS)
  - The return address is pushed when a link instruction (e.g., JAL) is executed, because we already know where to come back when we take a call
  - When the PC of a return instruction (e.g., JALR) is encountered, we can predict its next PC by popping the top of the stack

  How do you know when to follow RAS vs BTB?

  What would you do if return-address stack overflows?

Trace Caching

Move to after slide 41 of Lec12?
(this one and some following ones)



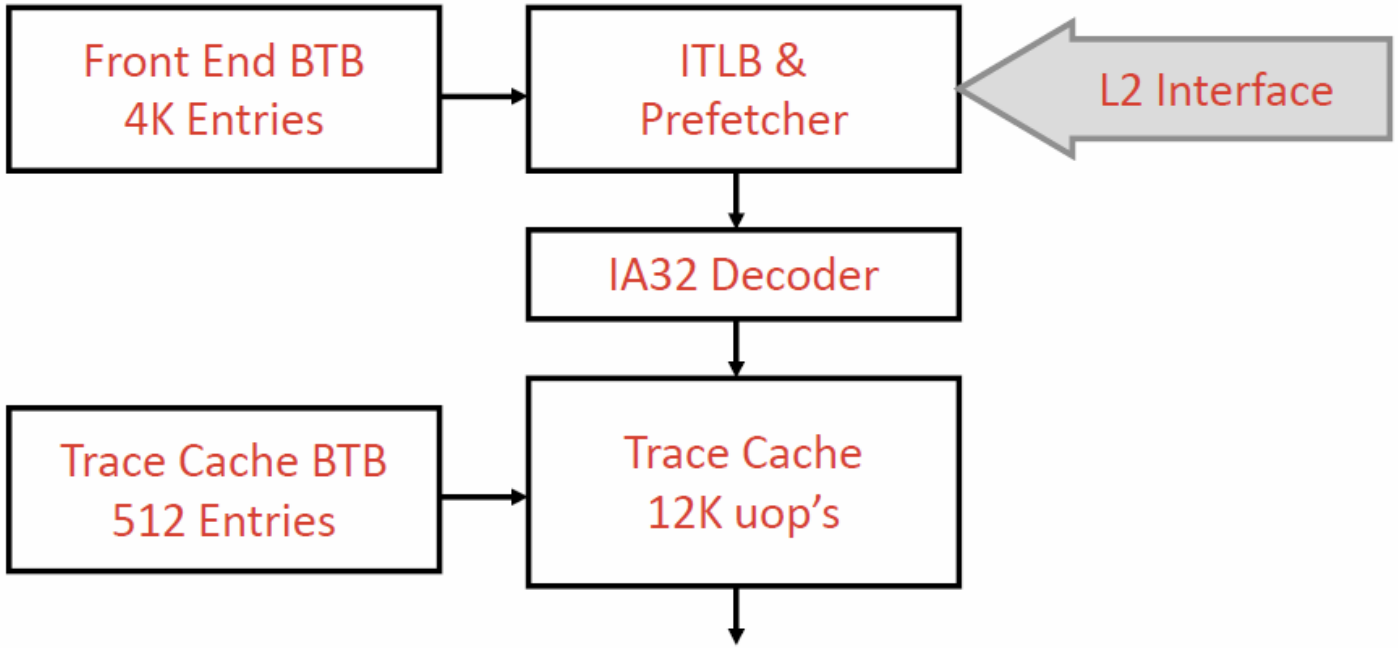
A “trace” is a sequence of instructions starting at any point in a dynamic instruction stream

Reduced flow controls (e.g., reduced # of taken branches, memory accesses)

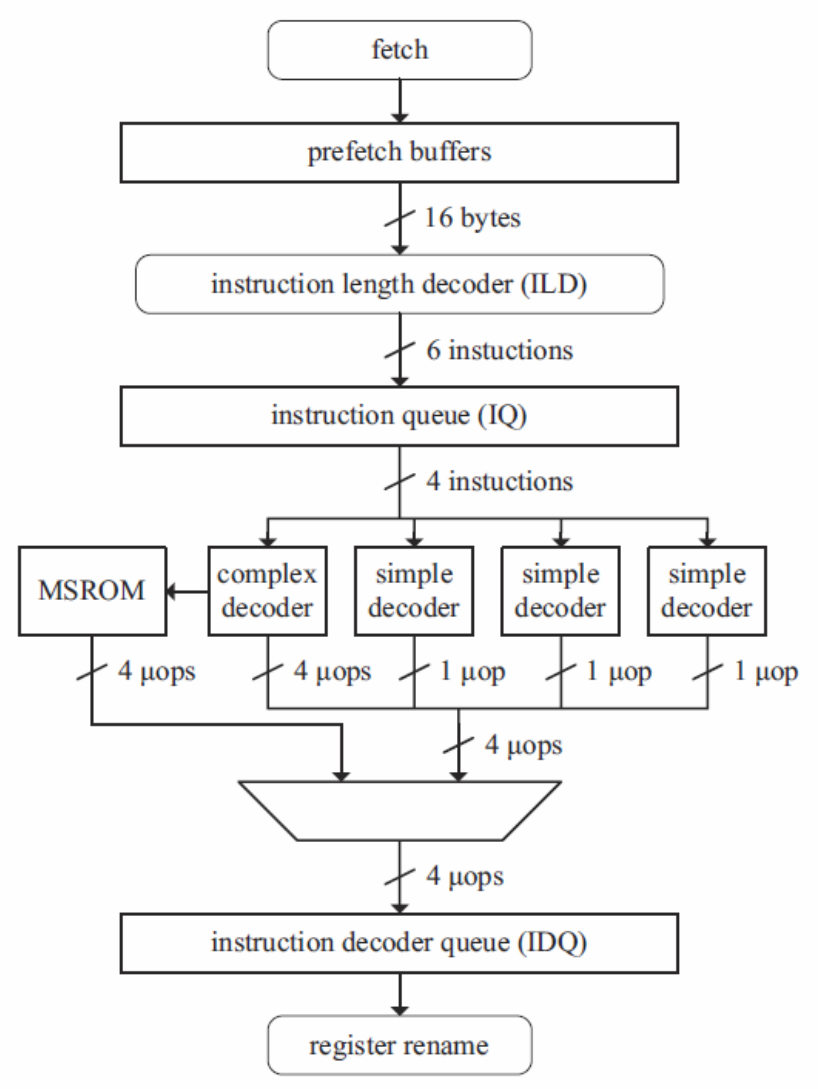
VS. multiple versions of traces for similar instruction paths?

Intel Pentium4 Cache

- A 12K-uop trace cache replaces L1 I-cache
- 6-uop per trace line, can include branches
- Trace cache returns 3-uop per cycle
- IA-32 decoder can be simpler and slower (if trace cache lookups are mostly hits)



High-performance x86 Decoding



Intel "Nehalem" decoding pipeline

- Two phases

  (1) Instruction Length Decoder (ILD)
  - Stream bytes → x86 instructions
  - Mostly 1 instruction per cycle
  - Sometimes 6-cycle path

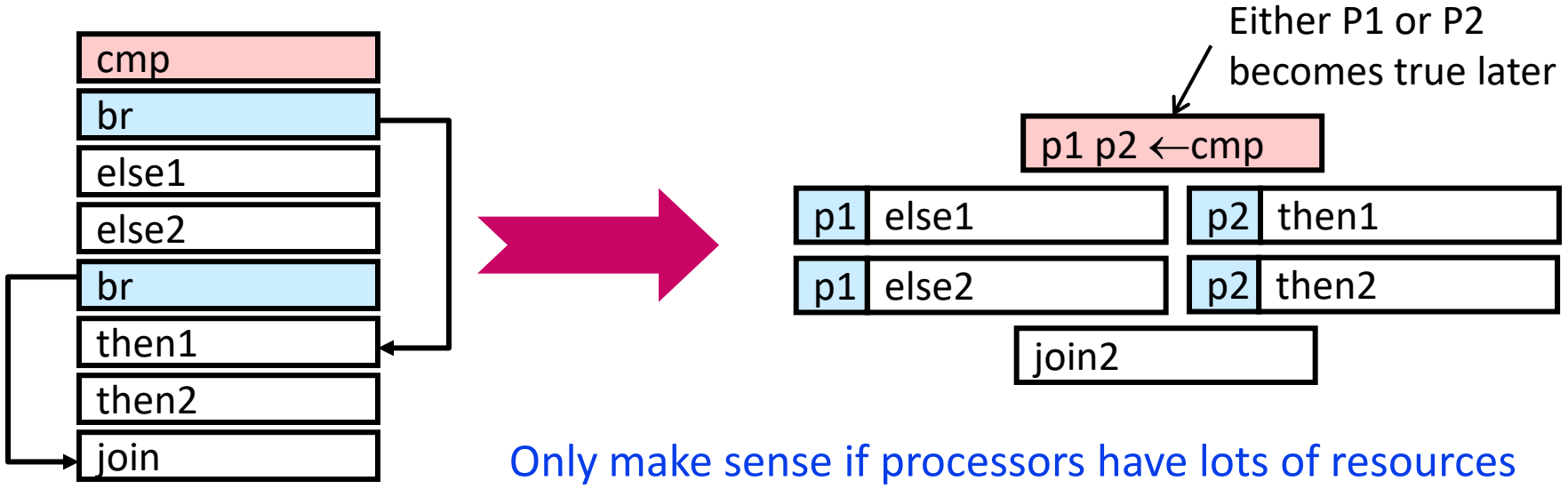
  (2) Dynamic translation
  - x86 instructions → uops
  - Mostly one x86 inst. → one uop

    e.g., reg-to-reg ops
  - 4 uop or more
    - Complex addressing
    - Can use microcontroller

Predicated Execution: If-conversion

- Example: “predication” in Intel Itanium
  - Each instruction can be separately **predicated** (= marked)
  - 64 one-bit predicate registers

    Each instruction carries a 6-bit predicate field
  - An instruction becomes effectively a NOP if its predicate is false
- Converts control flow into data flow



Only make sense if processors have lots of resources and existing BP does not work well

Involving SW in Branch Prediction

- Static branch “hints” can be encoded with every branch
  - Taken vs. Not-taken
  - Whether to allocate an entry in the dynamic BP hardware

- SW and HW can have joint control of BP hardware
  - Intel Itanium has a “**brp**” (branch prediction) instruction that can be issued ahead of the actual branch to preset the state of the BTB

- TAR (Target Address Register)
  - A small, fully-associative BTB
  - Controlled entirely by “**prepare-to-branch**” instructions
  - A hit in TAR overrides all other predictions

Branch Prediction: The Bottom Line

- Given current PC, how to determine the next PC
  - Waiting for correct information (later available) causes stalls
- The easy part
  - The same PC always points to the same instruction (except self-modifying code)
  - NextPC is always PC+4 for non-control-flow instructions,
  - The target of a PC-offset control-flow (e.g., Bxx and JAL) is always the same

    Keeping a target table can get these nearly 100% right
- The not-so-easy part
  - Taken vs. not-taken decision is not static
    - 90% of backward branches are taken (loops)
    - 50% of forward branches are taken (if-then-else)
  - A given branch almost always repeats itself

Question?

Announcements

- Reading: P&H (RISC-V ed.) Ch 4.9-4.15
- Paper reading (optional): "Dynamic Branch Prediction with Perceptrons", Jimenez and Lin, HPCA'01

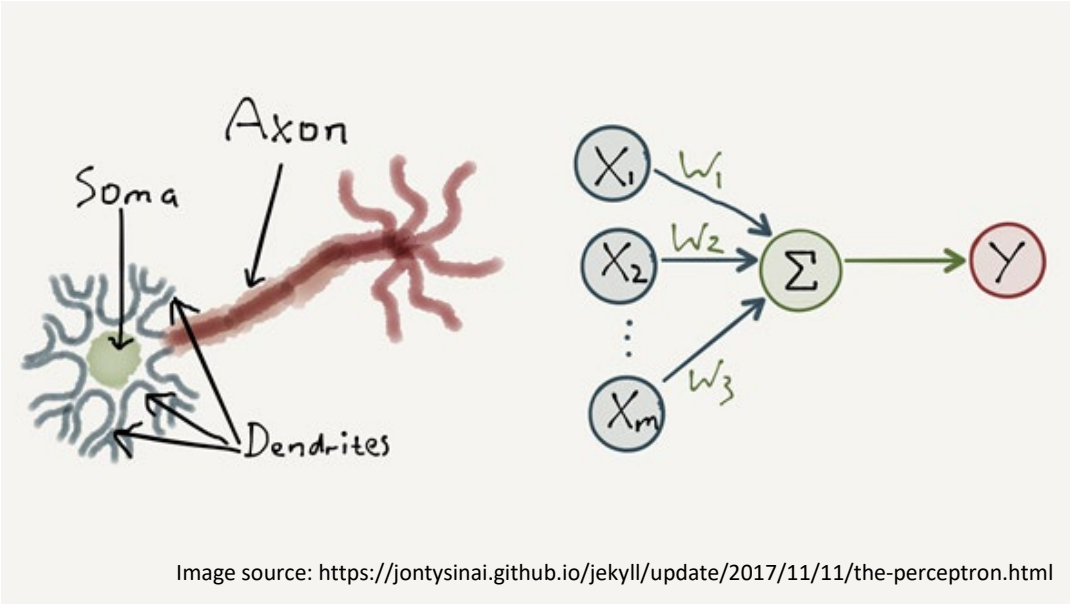


Image source: https://jontysinai.github.io/jekyll/update/2017/11/11/the-perceptron.html

Neuron Perceptron