Lecture 10: Branch Prediction

Jangwoo Kim (Seoul National University)
jangwoo@snu.ac.kr

Announcement

- HW#3
  - To be posted.
  - **Due: 4/19**
- Mid-term **4/20 (6:00PM)**.

Final PC Hazard Analysis (without branch prediction)

	R/I-Type	LW	SW	Br	J	Jr
IF	use (produce)	use (produce)	use (produce)	use	use	use
ID				produce	produce	produce
EX						
MEM						
WB						

- Hazard distance on a taken branch is 1
- Again, branch delay slot semantics ensures a dependent instruction to be at least distance 2

Hazard distance is greater in modern CPUs. Why?

Simplest Branch Prediction → Predict not-taken or PC+4

- Rather than waiting for true-dependence on PC to resolve, just **guess** nextPC = PC+4 to keep fetching every cycle

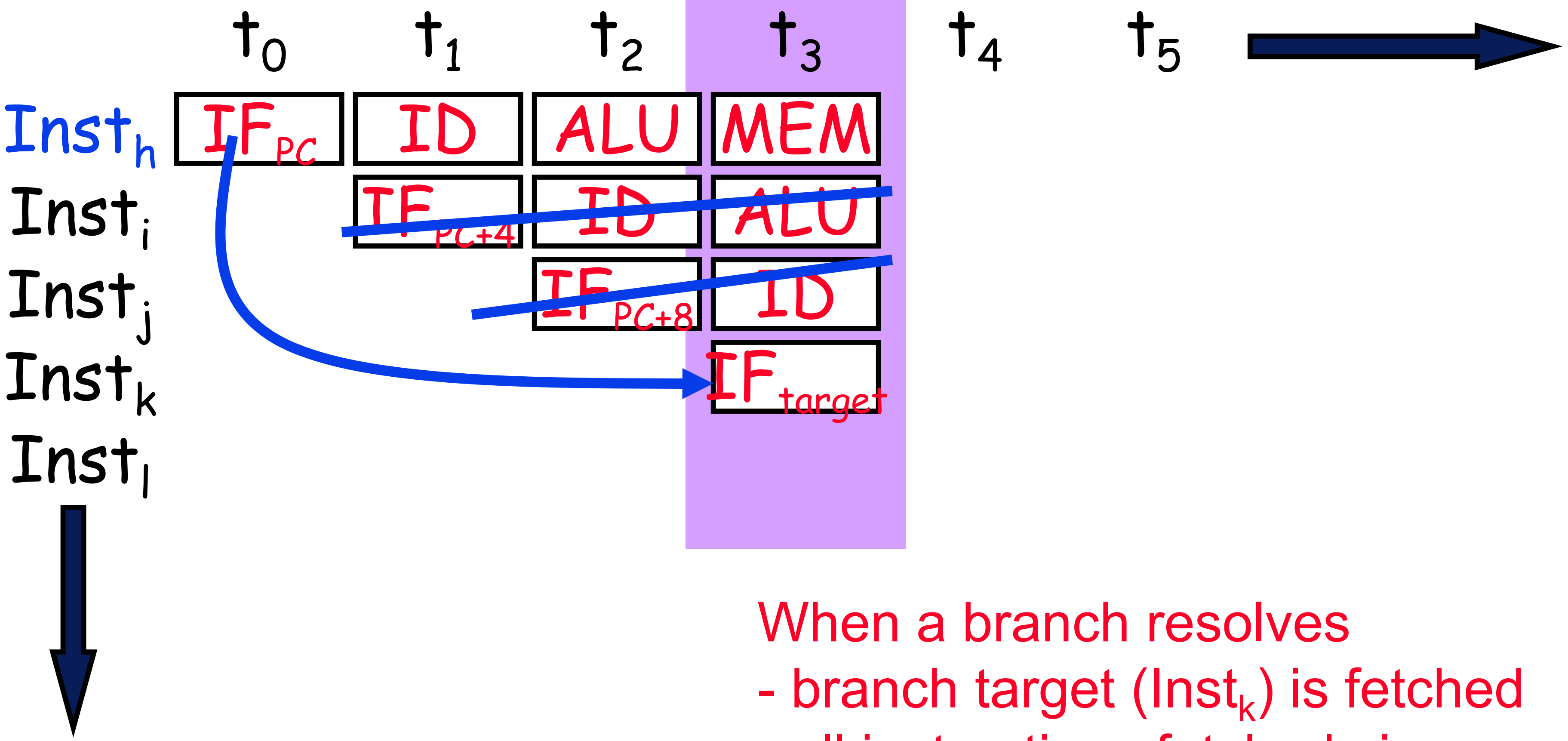
Is this a good guess? What do you lose if you guessed incorrectly?

- Only ~20% of the instruction mix is control flow
  - ~50 % of "forward" control flow (i.e., if-then-else) is taken
  - ~90% of "backward" control flow (i.e., loop back) is taken

Over all, typically ~70% taken and ~30% not taken
[Lee and Smith, 1984]

- Expect "nextPC = PC+4" ~86% of the time, but what about the remaining 14%?

Control Speculation: PC+4 (when you mispredict)



Inst_h is a branch which goes to Inst_{target}

When a branch resolves
- branch target (Inst_k) is fetched
- all instructions fetched since inst_h (so called "wrong-path" instructions) must be flushed

Performance Impact

- Correct guess ⇒ no penalty ~86% of the time
- Incorrect guess ⇒ 2 bubbles (or 3 bubbles in textbook)
- Assume
  - no data hazards
  - 20% control flow instructions
  - 70% of control flow instructions are taken
  - IPC = 1 / [1 + (0.2*0.7) * 2] =

$$= 1 / [\, 1 + \underbrace{0.14}_{\text{Probability of a wrong guess}} * \underbrace{2}_{\text{Penalty for a wrong guess}} \,] = 1 / 1.28 = 0.78$$

Can we reduce either of the two penalty terms?

Making a Better Guess

- 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

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

  (2) Then, where to jump? (or what is a branch target?)

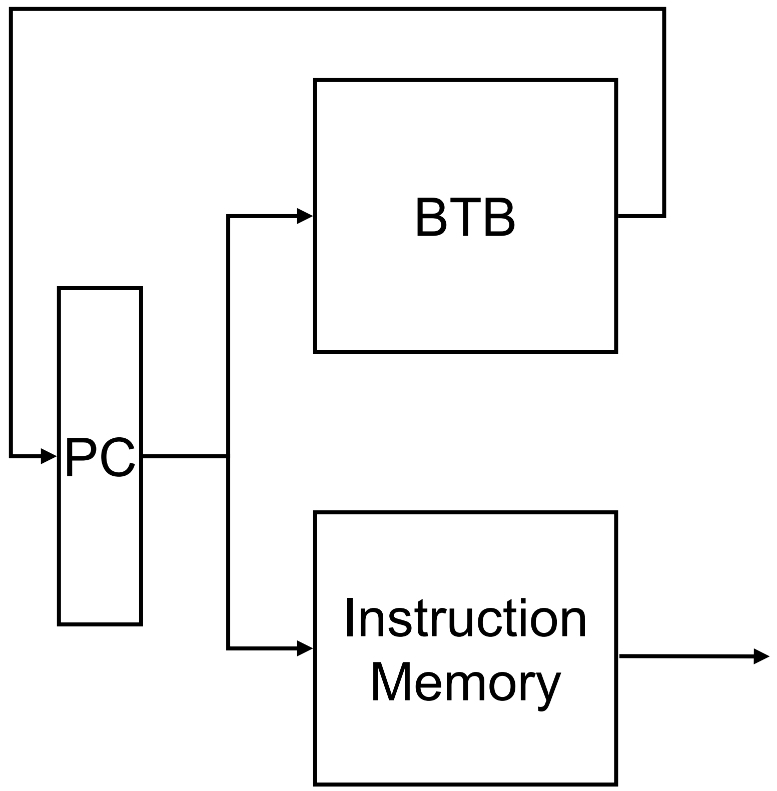
  - Again, must guess nextPC before the branch instruction is fetched (but branch target is encoded in the instruction)

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

  Fortunately,

  - PC-offset branch/jump target is static
  - We are allowed to be wrong some of the time

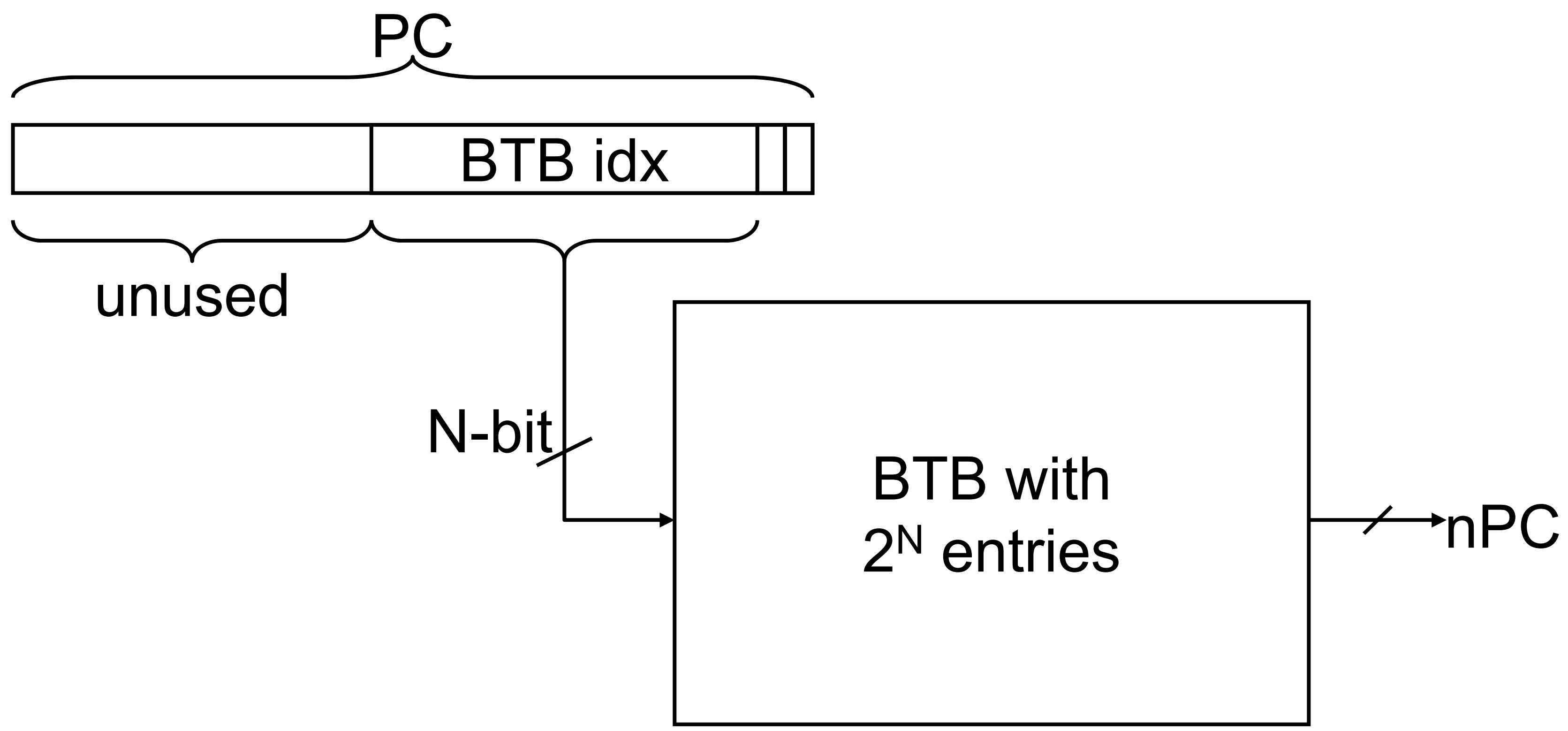
Branch Target Buffer (Oracle)



- 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 Jump
  - ?? if JR
- Effectively guessing branches are **always taken**

$$\text{IPC} = 1 / [\, 1 + (0.2 * \underline{0.3}) * 2 \,]$$
$$= 0.89$$

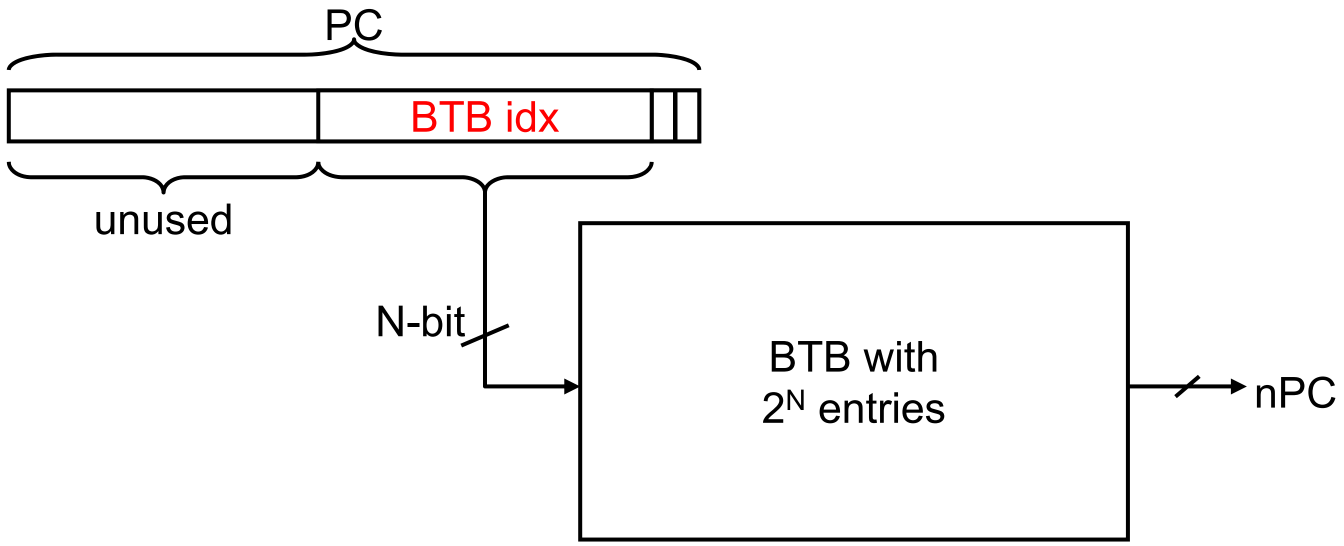
Branch Target Buffer (Reality)



- "Hash" PC into a 2N entry table
- On collision, BTB returns something meaningless and possibly wrong (since 80% of entries all hold PC+4)

How big should this table be?

When a collision happens?



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

PC: 0000000000000000000000 0011001100 : ADD → PC+4?

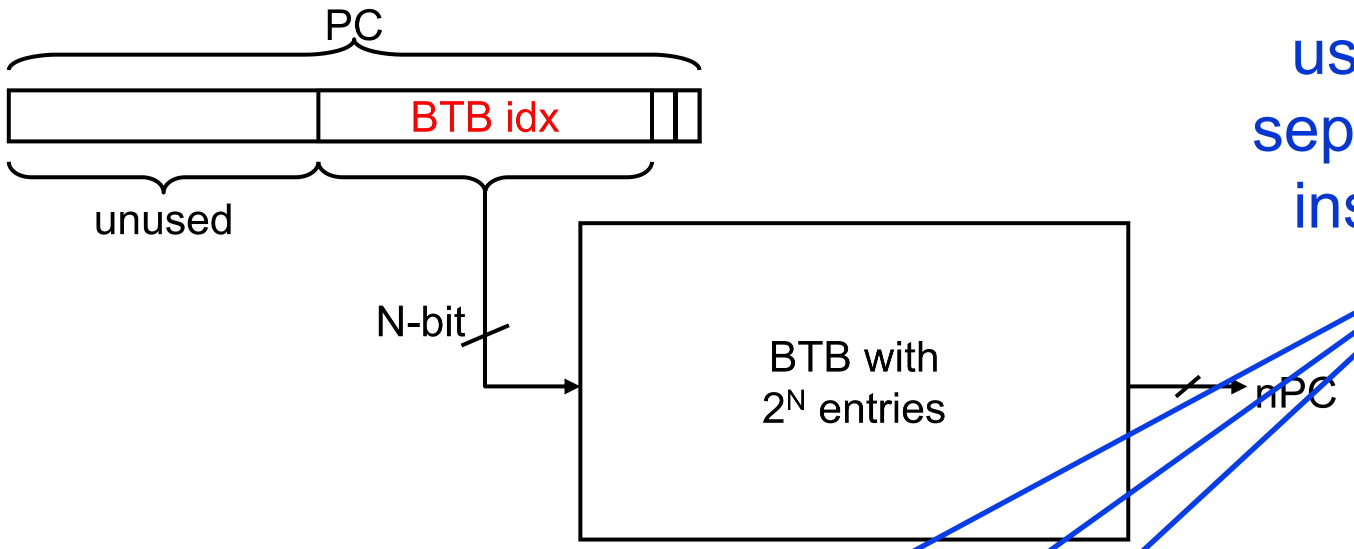
VS

PC: 11111.....11111111111111 0011001100 : JUMP → A target?

VS

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

When a collision happens?



use 'TAG' to separate these instructions!

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

PC: 0000000000000000000000 0011001100 : ADD → PC+4?

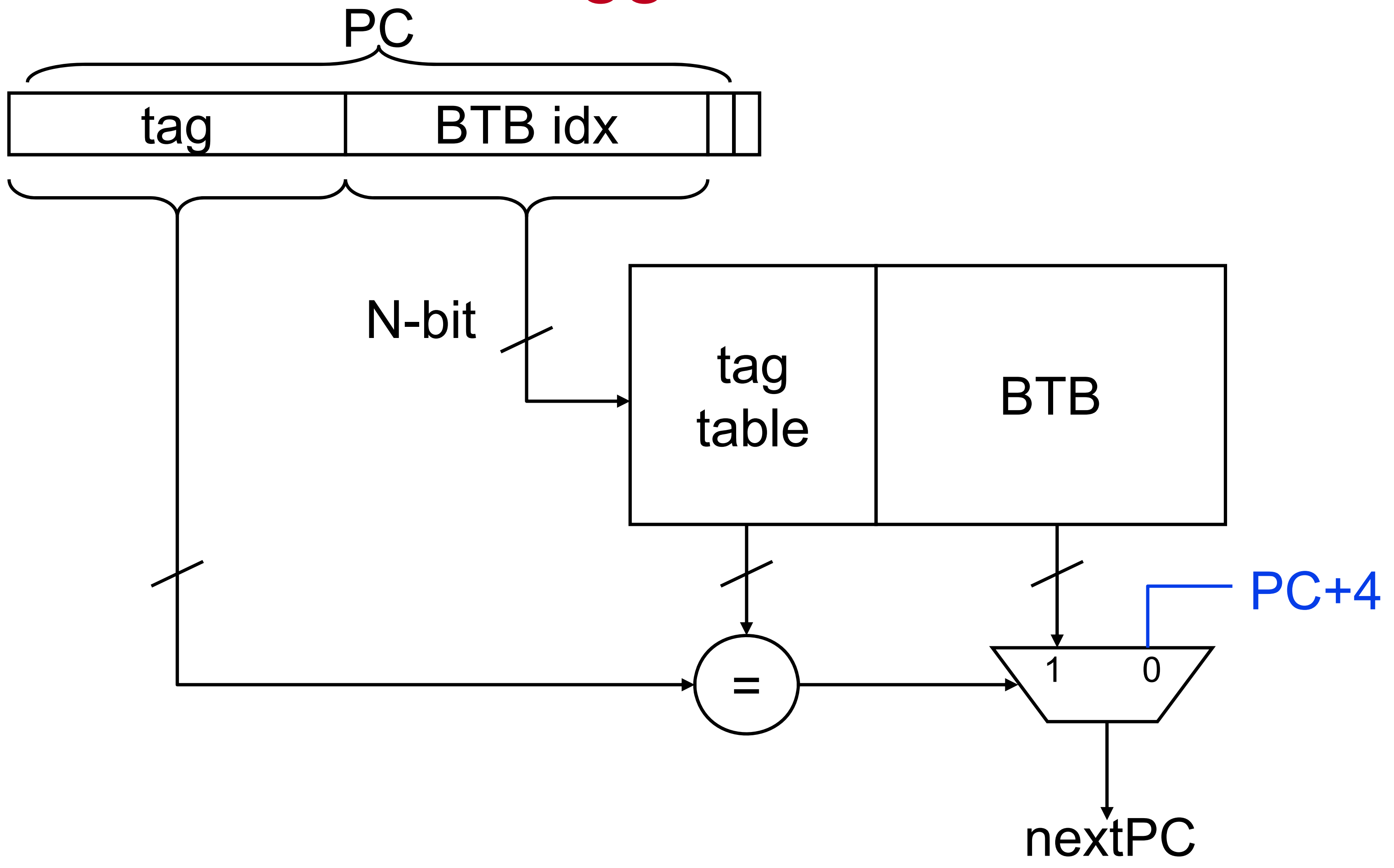
VS

PC: 11111.....11111111111111 0011001100 : JUMP → A target?

VS

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

Tagged BTB



Let's store branch instructions (to save 80% storage)!

Update tag and BTB for the new branch after each collision

More advanced prediction?

- We can get 100% correct on non-branch instructions

  Why?

- Can we do better than 70% on branch instructions?
  - We get 90% right on backward branch (dynamic)
  - We only get 50% right on forward branch (dynamic)

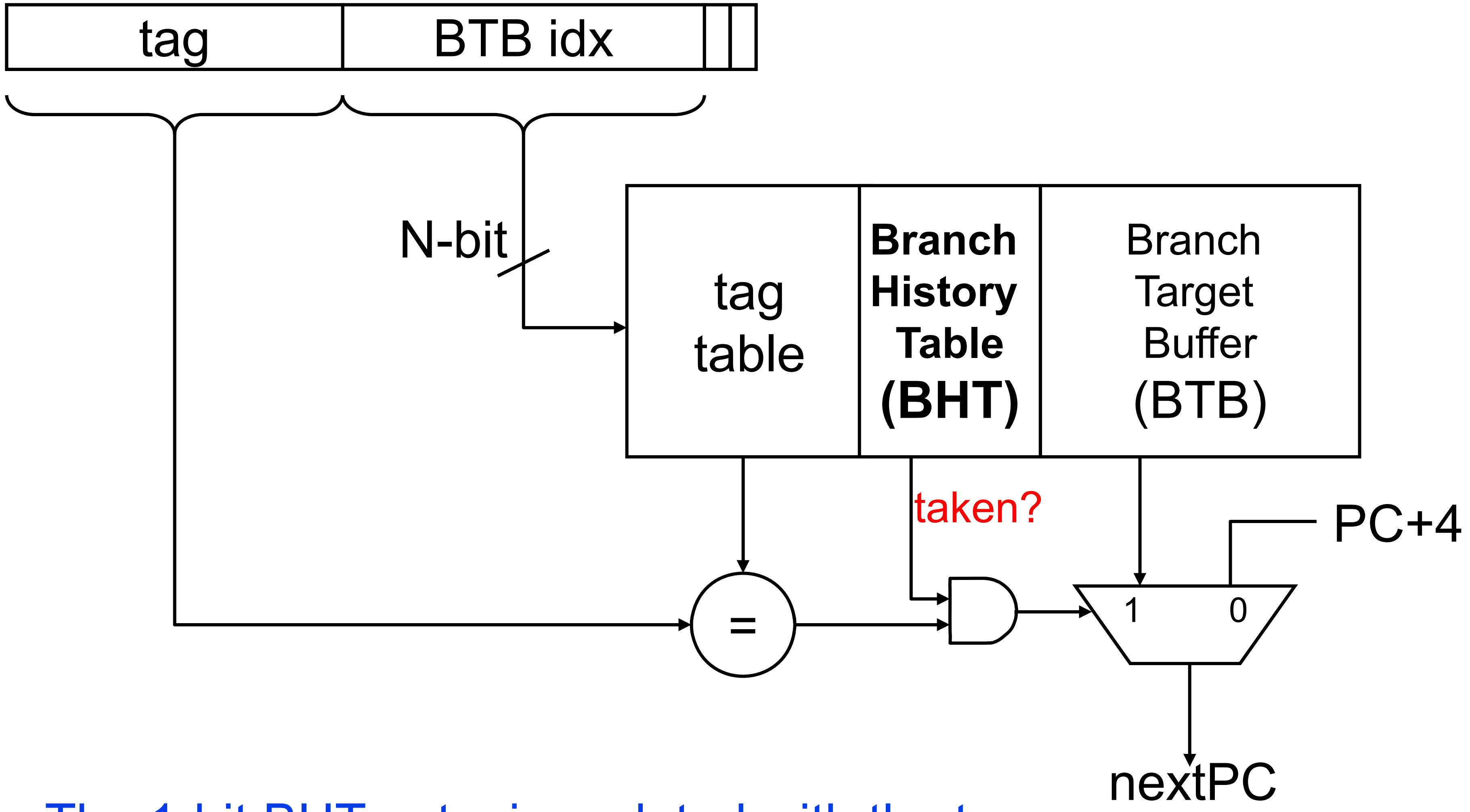
    What patterns can we leverage on forward branches?

- 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)

    → **~85%** correct prediction if we always guessed **the same outcome as the last time** the same branch was executed

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

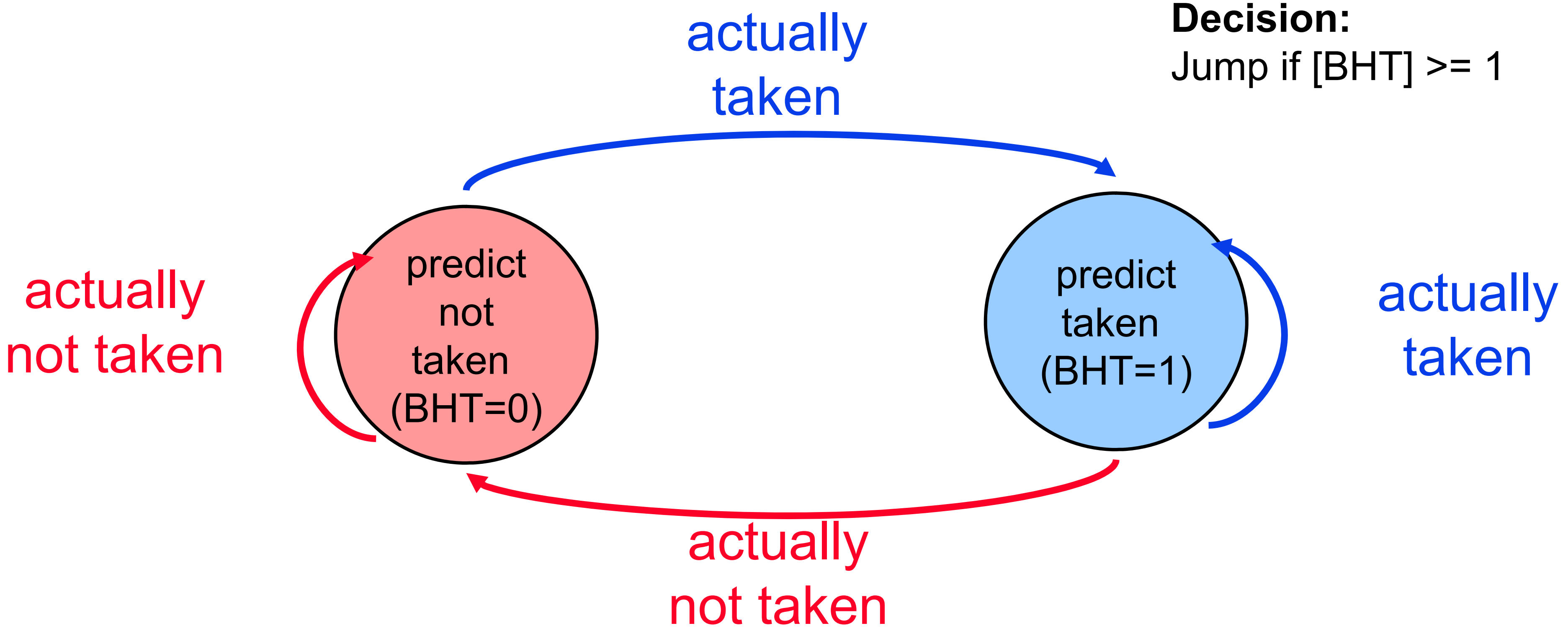
Branch History Table and Target Buffer



The 1-bit BHT entry is updated with the true outcome after each execution of a branch

Branch Prediction State Machine (1-bit)

Decision:
Jump if [BHT] >= 1

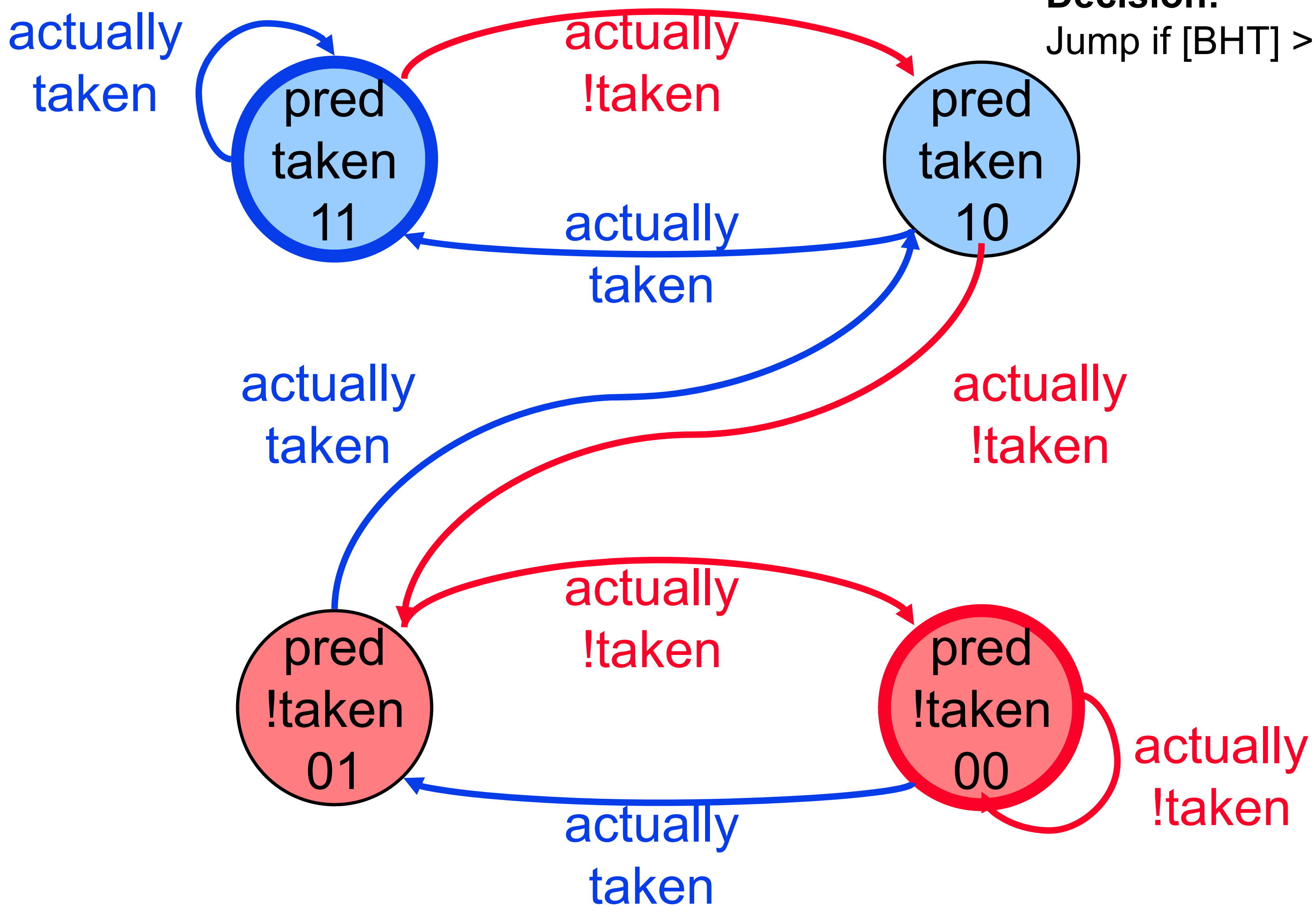


What is prediction accuracy for

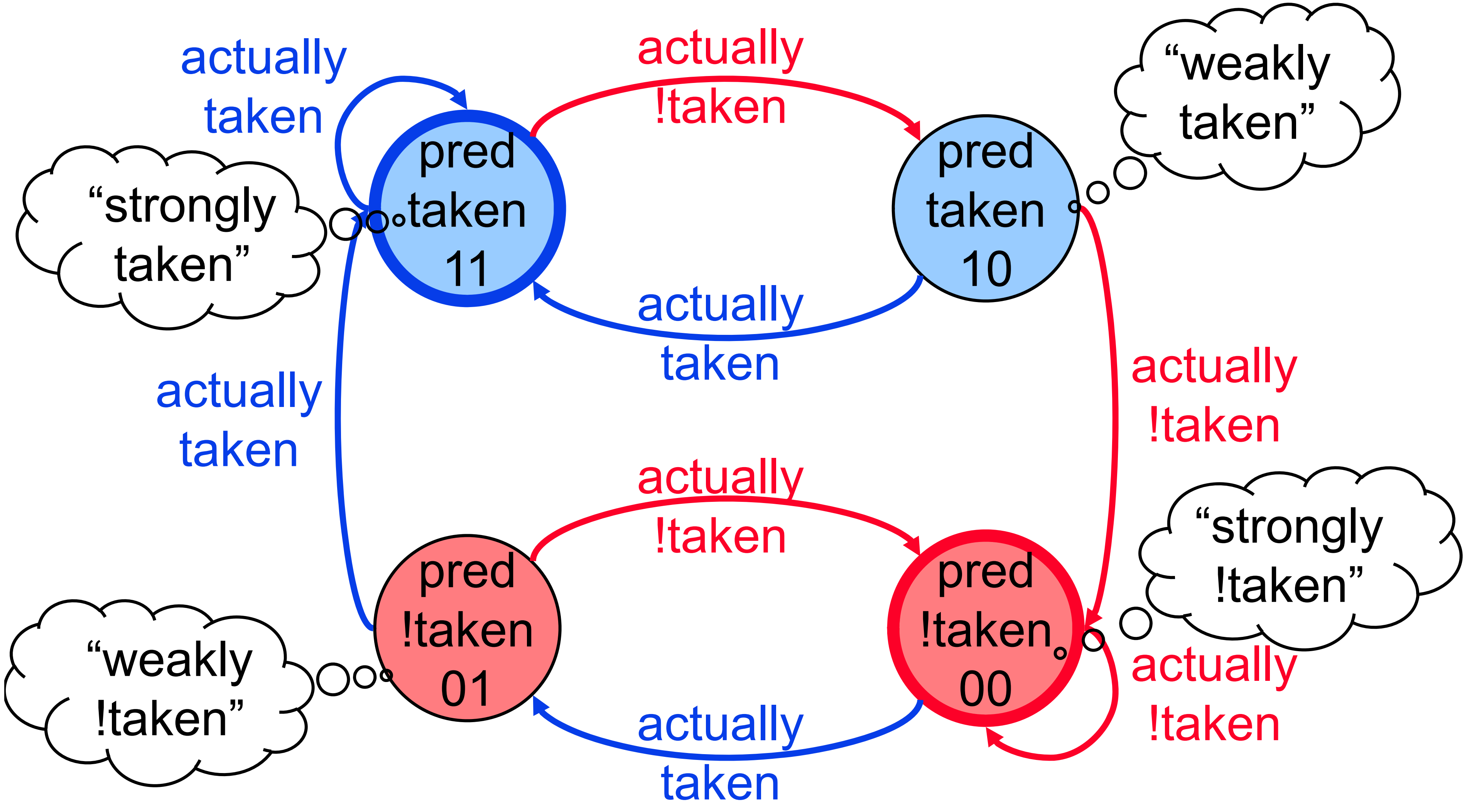
1) T → NT → T → NT → T → NT → T → NT → ..
2) T → T → NT → T → T → NT → T → T → NT → ..

2-Bit "Saturation" Counter

Decision:
Jump if [BHT] >= 10



2-Bit "Hysteresis" Counter



Change prediction after 2 consecutive mistakes

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?)	-	X	X	X	O	X	X	X	O	X	X	X	O
current counter)	0	1	2	1	0	1	2	1	0	1	2	1	0
prediction)	N	N	T	N	N	N	T	N	N	N	T	N	N

  - 2-bit hysteresis

outcome)		T	T	N	N	T	T	N	N	T	T	N	N
correct?)	-	X	X	X	X	X	X	X	X	X	X	X	X
current counter)	0	1	3	2	0	1	3	2	0	1	3	2	0
prediction)	N	N	T	T	N	N	T	T	N	N	T	T	N

Example #2

- 2-bit saturation VS 2-bit hysteresis
  - 2-bit saturation

outcome)		T	T	N	T	N	T	N	T	N	T	N	T
correct?)	-	X	X	X	X	X	X	X	X	X	X	X	X
current counter)	0	1	2	1	2	1	2	1	2	1	2	1	2
prediction)	N	N	T	N	T	N	T	N	T	N	T	N	T

  - 2-bit hysteresis

outcome)		T	T	N	T	N	T	N	T	N	T	N	T
correct?)	-	X	X	X	O	X	O	X	O	X	O	X	O
current counter)	0	1	3	2	3	2	3	2	3	2	3	2	3
prediction)	N	N	T	T	T	T	T	T	T	T	T	T	T

Different predictors can predict different patterns!

State-Machine-Based Predictors

- 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
  - Almost always do the same thing again and again (>80%)
    - 1-bit and 2-bit predictors equally effective
  - Occasionally do the opposite once (5~10%)
    - 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?**

      e.g., 2-level branch predictor

"Global" Path History

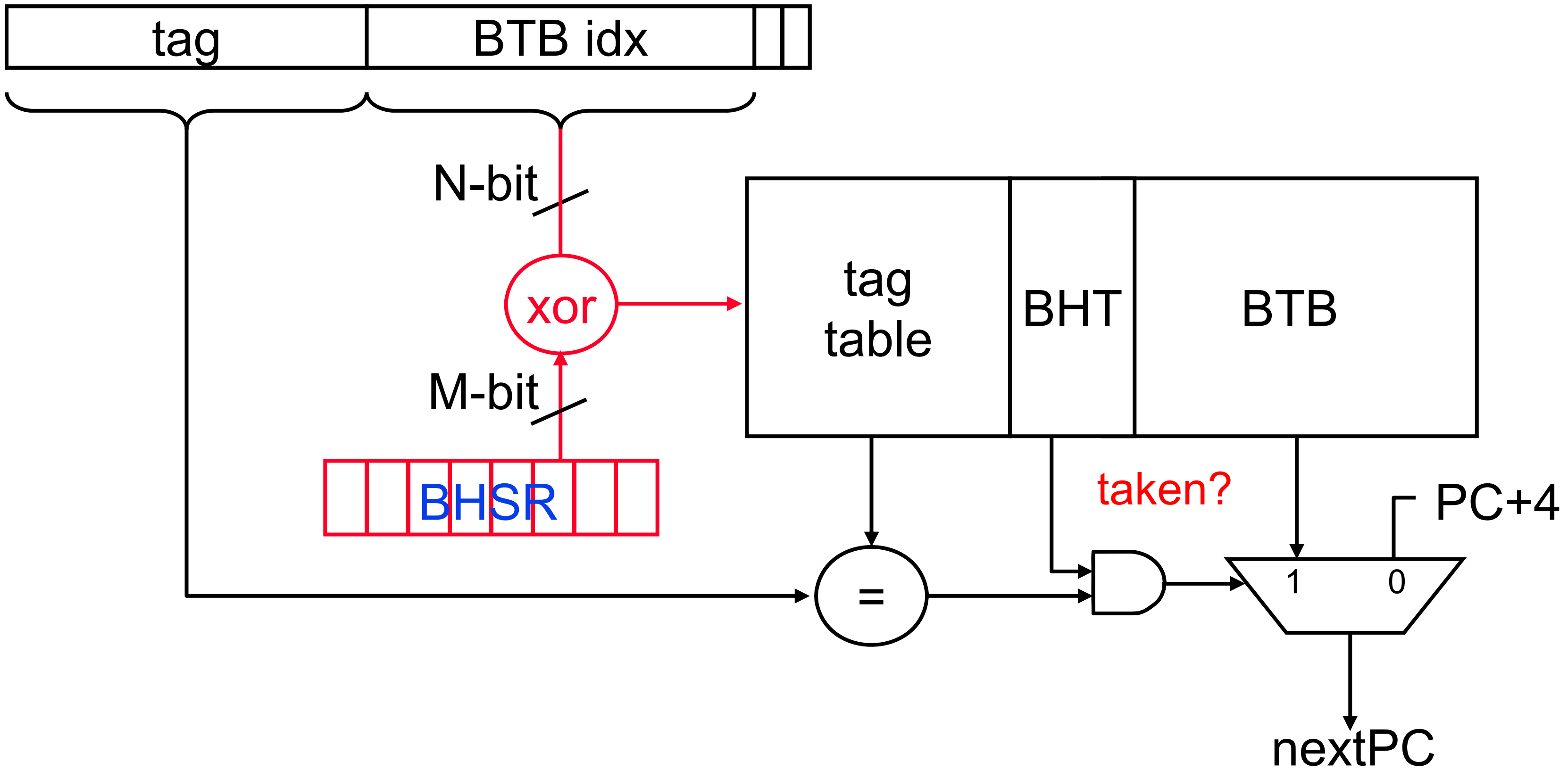
- So far we focused on each branch's behavior
- But branch outcome often correlated to other branches
- Example

```
aa = bb = 1;
if (a > 1)                        ;; B1
        aa=0;
if (b > 1)                        ;; B2
        bb=0;
if (aa == 1 && bb == 0) {   ;; B3
        ….
}
```

→ If B1 is not-taken (aa=1@B3) and B2 is taken (bb=0@B3), then B3 is certainly taken. (NT)→(T) → ?

How do you capture this information?

"Gshare" Branch Prediction [McFarling]



Global BHSR (Branch History Shift Register) tracks the outcomes of the last M branch instructions (NT→ T …)

Is this a local predictor or a global predictor?

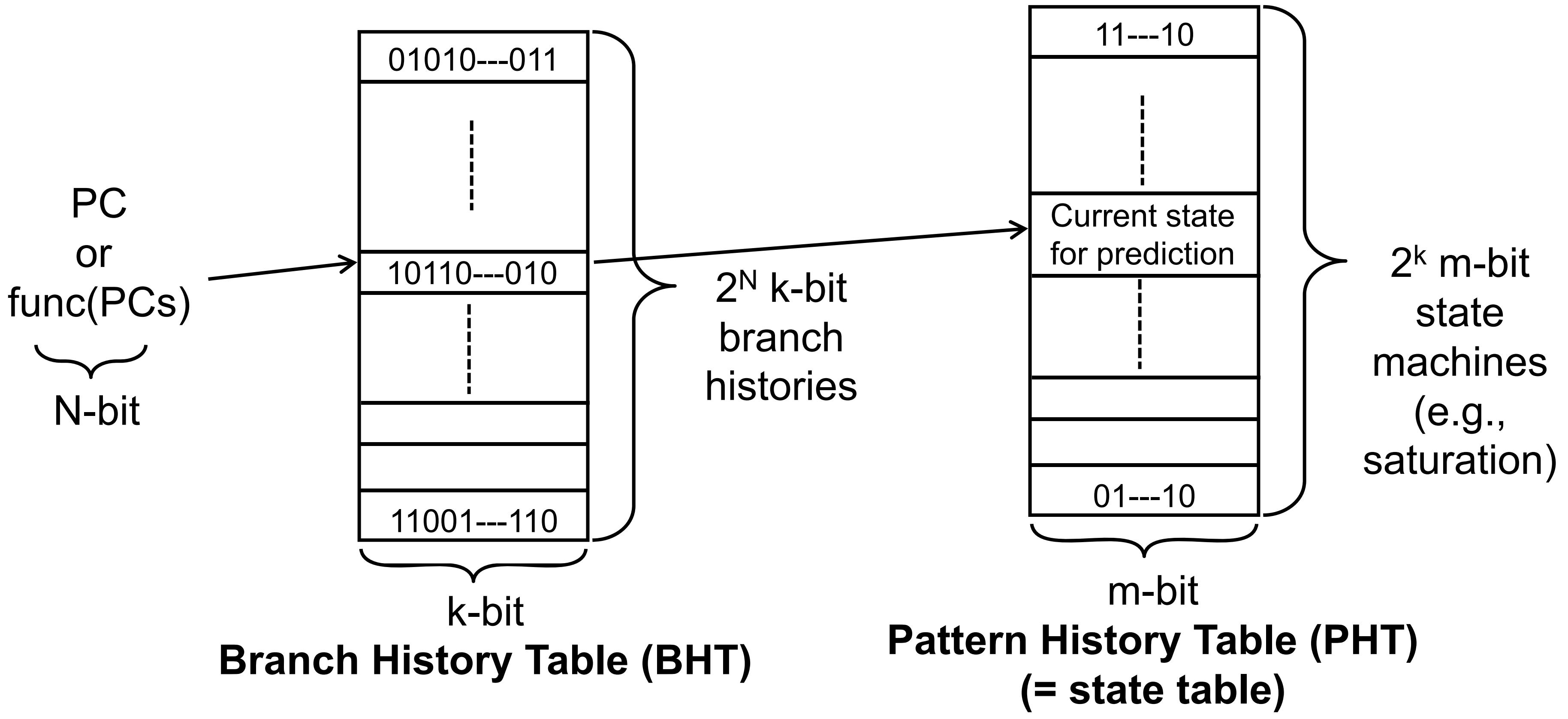
Return Address Stack

- The targets of register-indirect jumps 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., JR) 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?

Two-level branch "direction" predictor


PC
or
func(PCs)
N-bit
01010---011
10110---010
11001---110
k-bit
Branch History Table (BHT)
2^N k-bit branch histories
11---10
Current state for prediction
01---10
m-bit
Pattern History Table (PHT)
(= state table)
2^k m-bit state machines (e.g., saturation)

This predictor can predict many different patterns
(local patterns & global patterns)

Two-level branch "direction" predictors

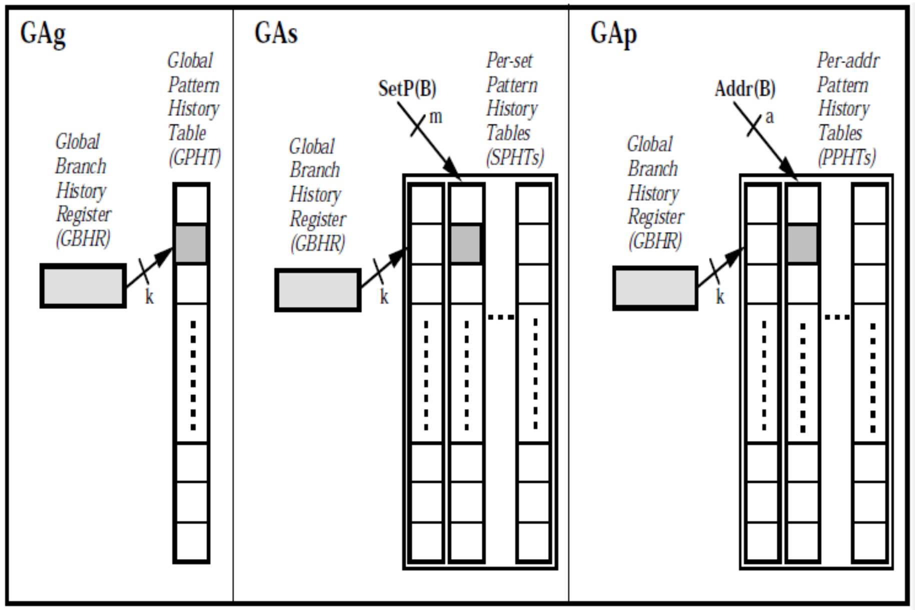
- BHT (branch history register) + PHT (pattern history table)
  - BHT
    - G : global
    - S : per-set (of addresses)
    - P : per-address
  - PHT
    - g : global
    - s : per-set (of addresses)
    - p : per-address

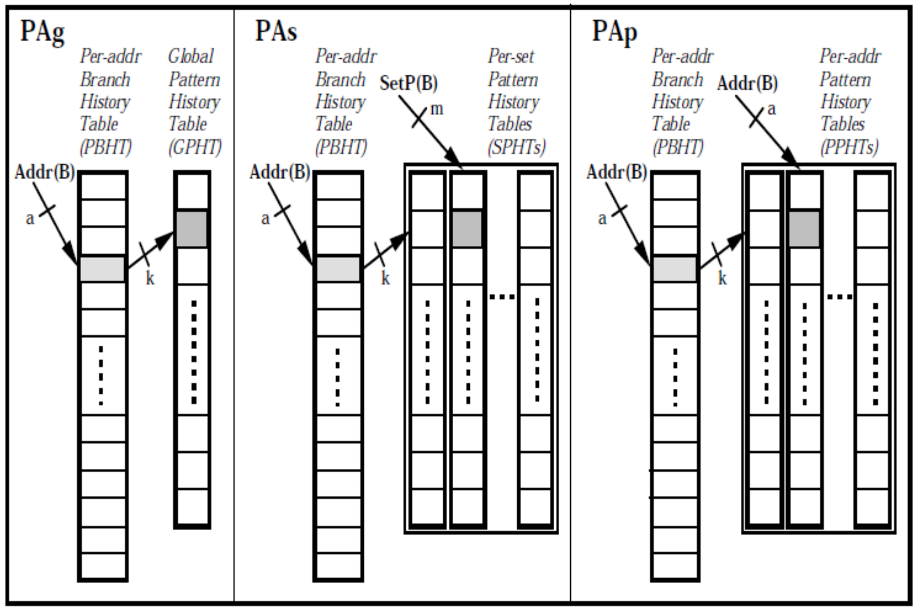
9 combinations possible
- GAg, GAs, GAp,
  SAg, SAs, SAp,
  PAg, PAs, PAp,

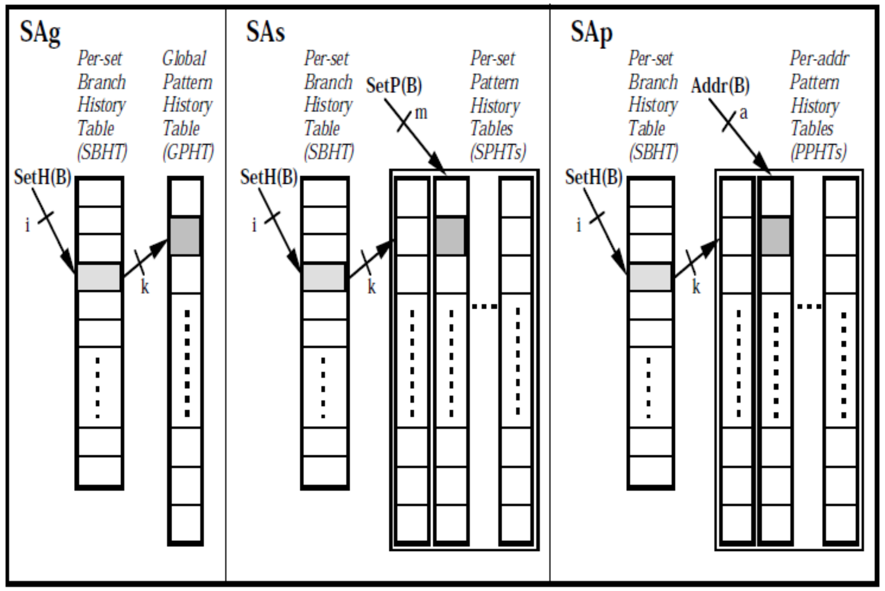
Reference

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.

Two-level branch “direction” predictors

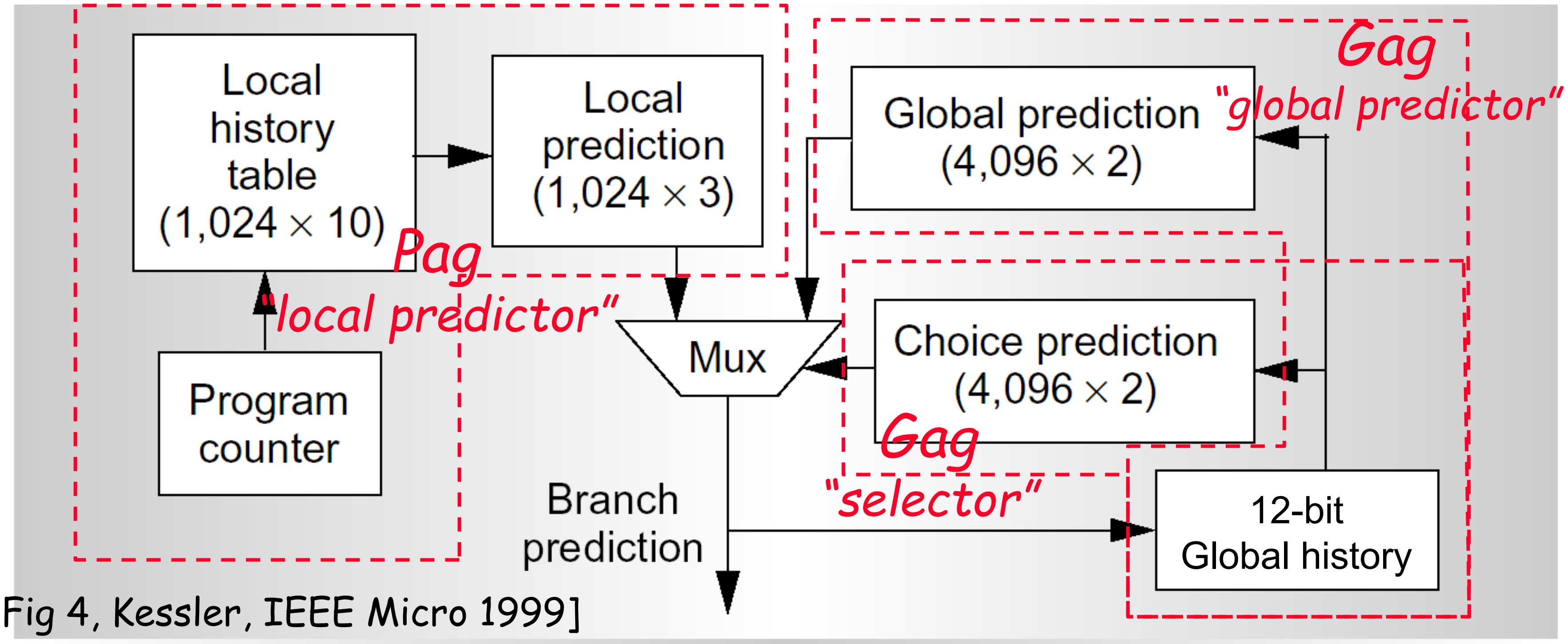

GAg
Global Branch History Register (GBHR)
Global Pattern History Table (GPHT)
k
GAs
Global Branch History Register (GBHR)
SetP(B)
m
Per-set Pattern History Tables (SPHTs)
k
GAp
Global Branch History Register (GBHR)
Addr(B)
a
Per-addr Pattern History Tables (PPHTs)
k


PAg
Per-addr Branch History Table (PBHT)
Global Pattern History Table (GPHT)
Addr(B)
a
k
PAs
Per-addr Branch History Table (PBHT)
SetP(B)
m
Per-set Pattern History Tables (SPHTs)
Addr(B)
a
k
PAp
Per-addr Branch History Table (PBHT)
Addr(B)
a
Per-addr Pattern History Tables (PPHTs)
Addr(B)
a
k


SAg
Per-set Branch History Table (SBHT)
Global Pattern History Table (GPHT)
SetH(B)
i
k
SAs
Per-set Branch History Table (SBHT)
SetP(B)
m
Per-set Pattern History Tables (SPHTs)
SetH(B)
i
k
SAp
Per-set Branch History Table (SBHT)
Addr(B)
a
Per-addr Pattern History Tables (PPHTs)
SetH(B)
i
k

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

Alpha 21264 Tournament Predictor [DEC1996]

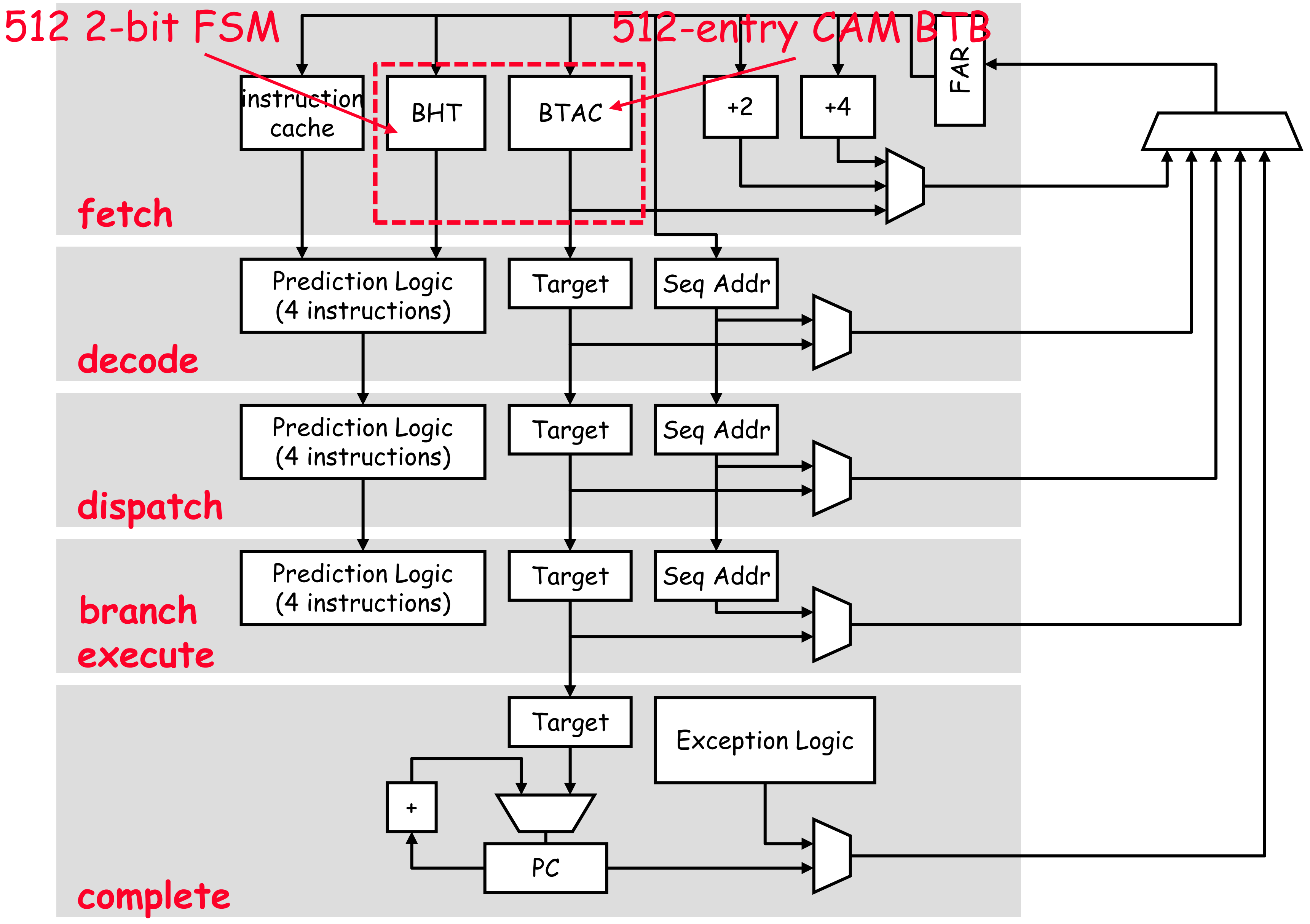

Local history table (1,024 × 10)
Local prediction (1,024 × 3)
Global prediction (4,096 × 2)
Choice prediction (4,096 × 2)
Program counter
Mux
Branch prediction
12-bit Global history
Pag
"local predictor"
Gag
"global predictor"
Gag
"selector"

[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

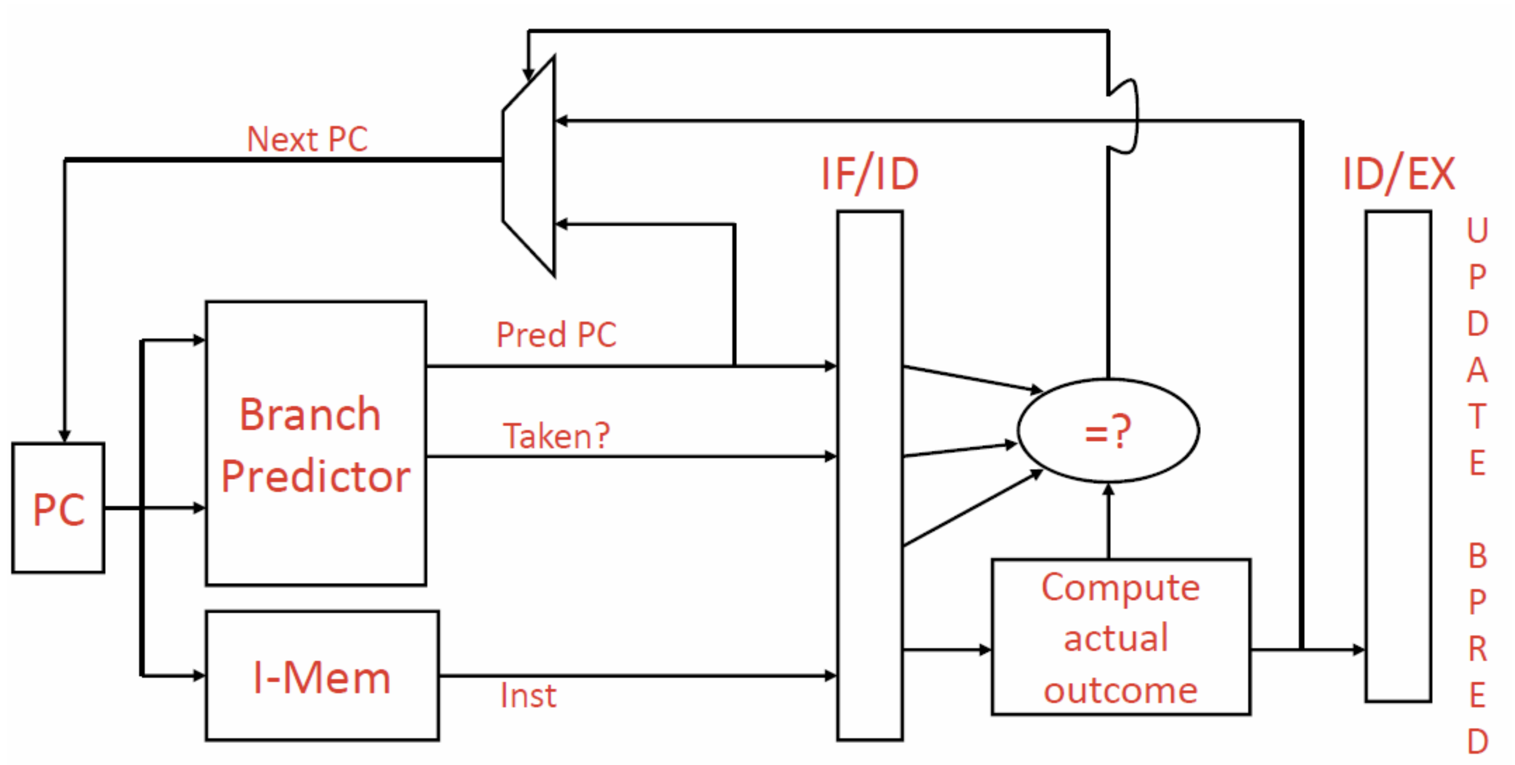
Better than 97% correct!!

Multiple Predictors: PowerPC 604 [IBM 1994]



Branch Predictor in a Pipeline

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



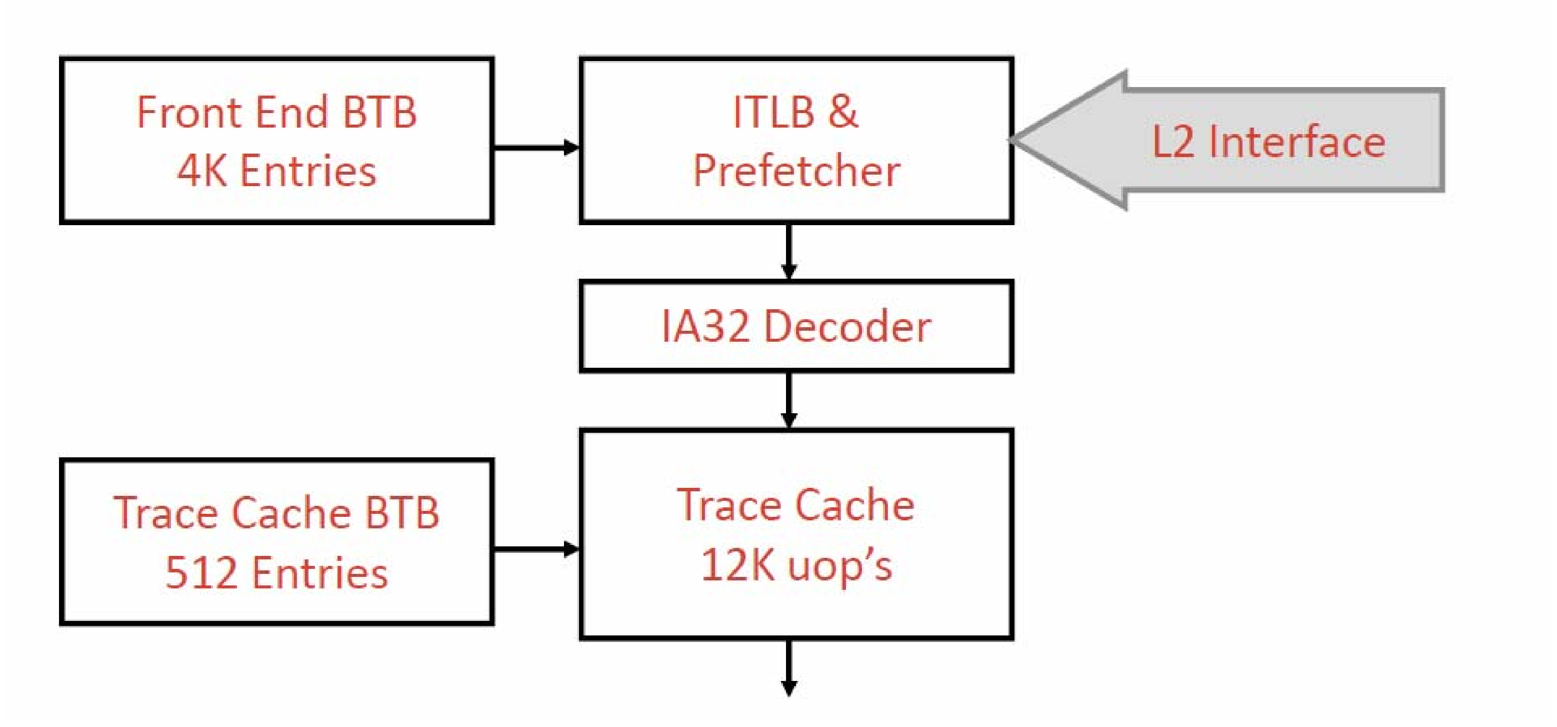
Speculative Execution Summary

- 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 BHT (=prediction state logic → reinforce prediction)
  - Do nothing more
- if nextPC was predicted incorrectly
  - Update BHT and/or BTB
  - Flush all "younger" instructions in the pipeline
  - Restart fetching at the correct target

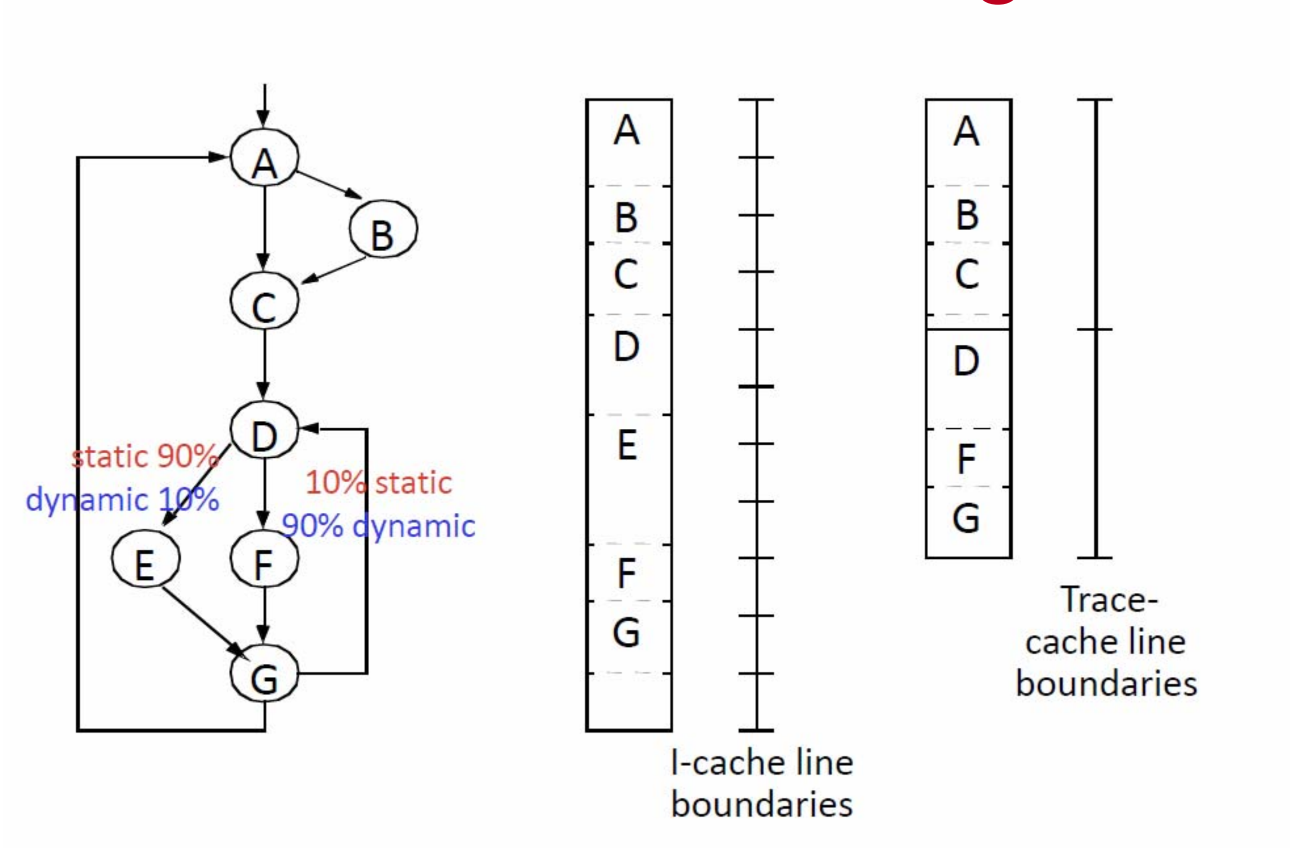
Relatively easy for in-order pipelines like 5-stage MIPS as you know. However, it can become **extremely difficult** in deeply pipelined, superscalar, out-of-order pipelines (e.g., Intel i7)

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 simper and slower



Trace Caching

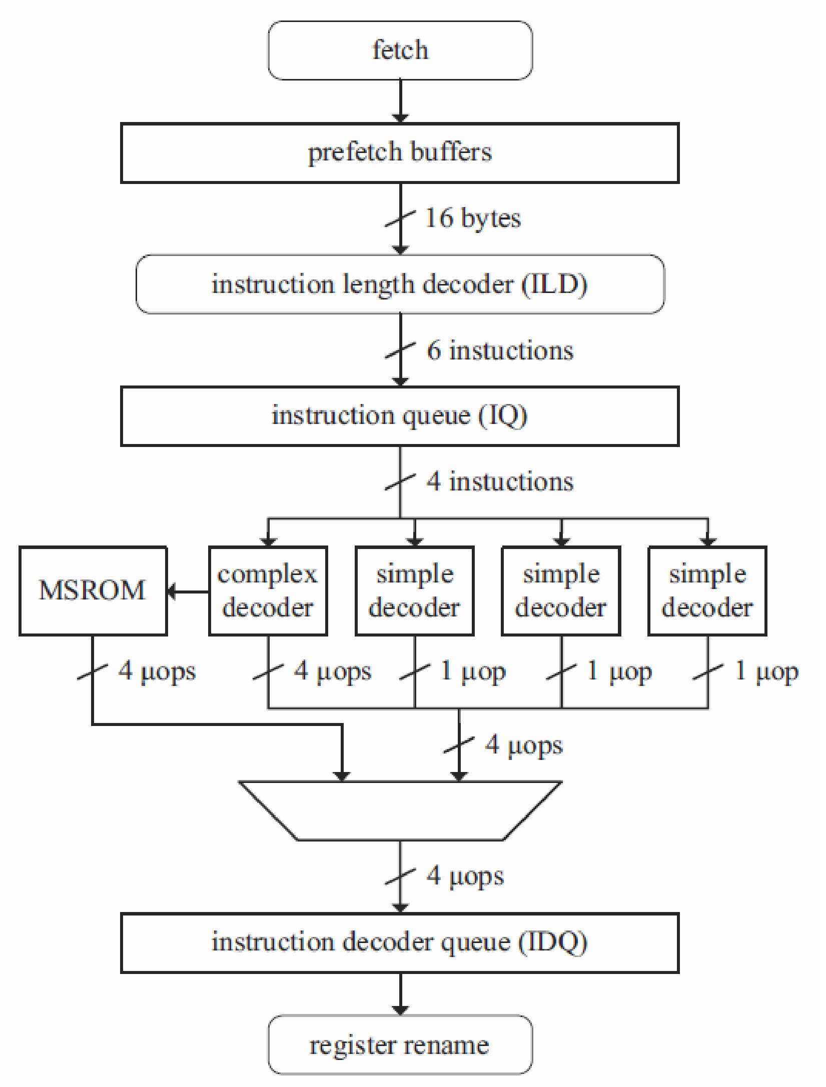


A "trace" is a sequence of instructions starting at any point in a dynamic instruction stream

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

VS. multiple versions of traces for similar instruction paths?

High-performance x86 decoding



Intel "Nehalem" decoding pipeline

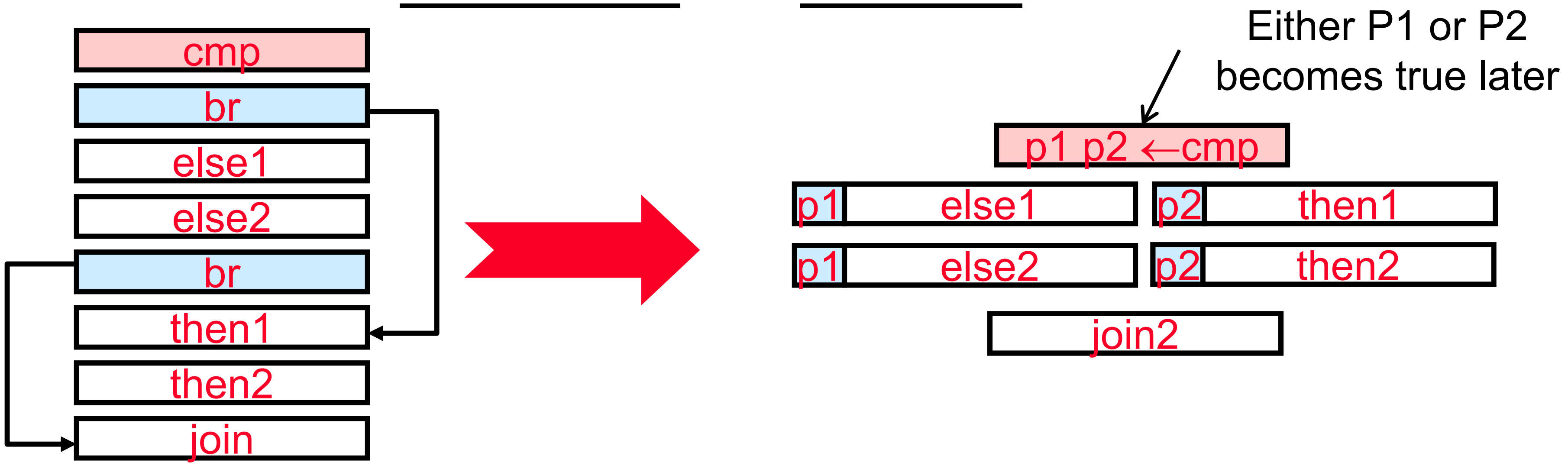
- Two phases
  - (1) Instruction Length Decoder (ILD)
    - Stream bytes → x86 instructions
    - Mostly 1 instruction per cycle
    - Sometimes 6-cycle path
  - (2) Dynamic translation
    - x86 instructions → uops
    - Mostly one x86 inst. → one uop

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

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 <u>control flow</u> into <u>data flow</u>



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 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 is always the same

    Keeping a target table can get these nearly100% right
- The not-so-easy part
  - Taken versus 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:	Homework #3 will be collected (due: 4/19)
	Mid-term on 4/20 (6:00PM--)
Reading:	Finish P&H Ch. 4 & Review P&H Ch.1~4
Handouts:	None