

Branch Prediction

15-740 FALL'18

NATHAN BECKMANN

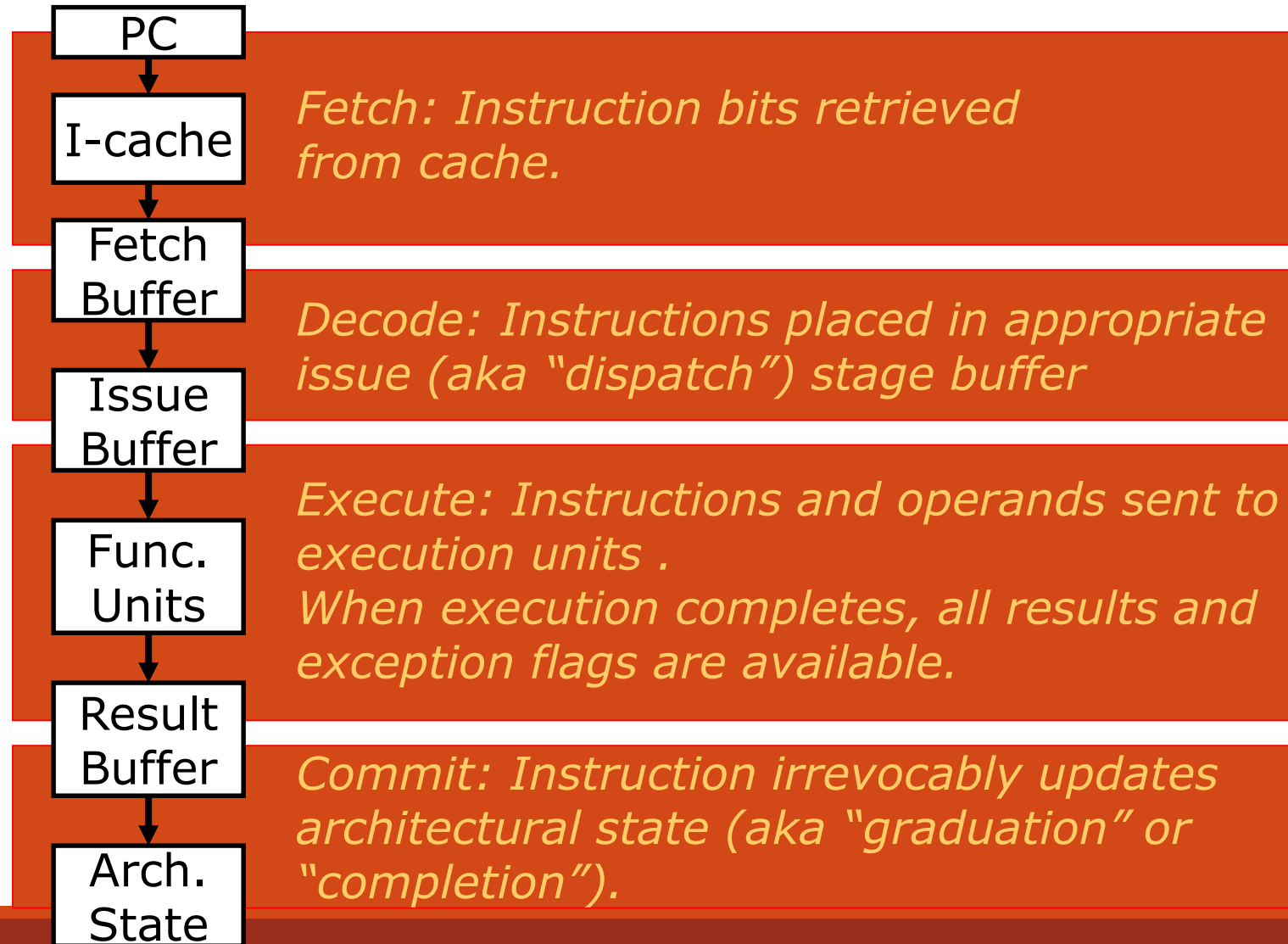
BASED ON SLIDES BY JOEL EMER, MIT

Today: Branch Prediction

Reasons to care about branch prediction:

1. Big performance impact in high-performance (von Neumann) cores
 - Big research emphasis in late '90s - early '00s
 - Still active & industry still interested
2. Example of prediction mechanisms used throughout comp arch
 - E.g., cache replacement, prefetching

Phases of Instruction Execution

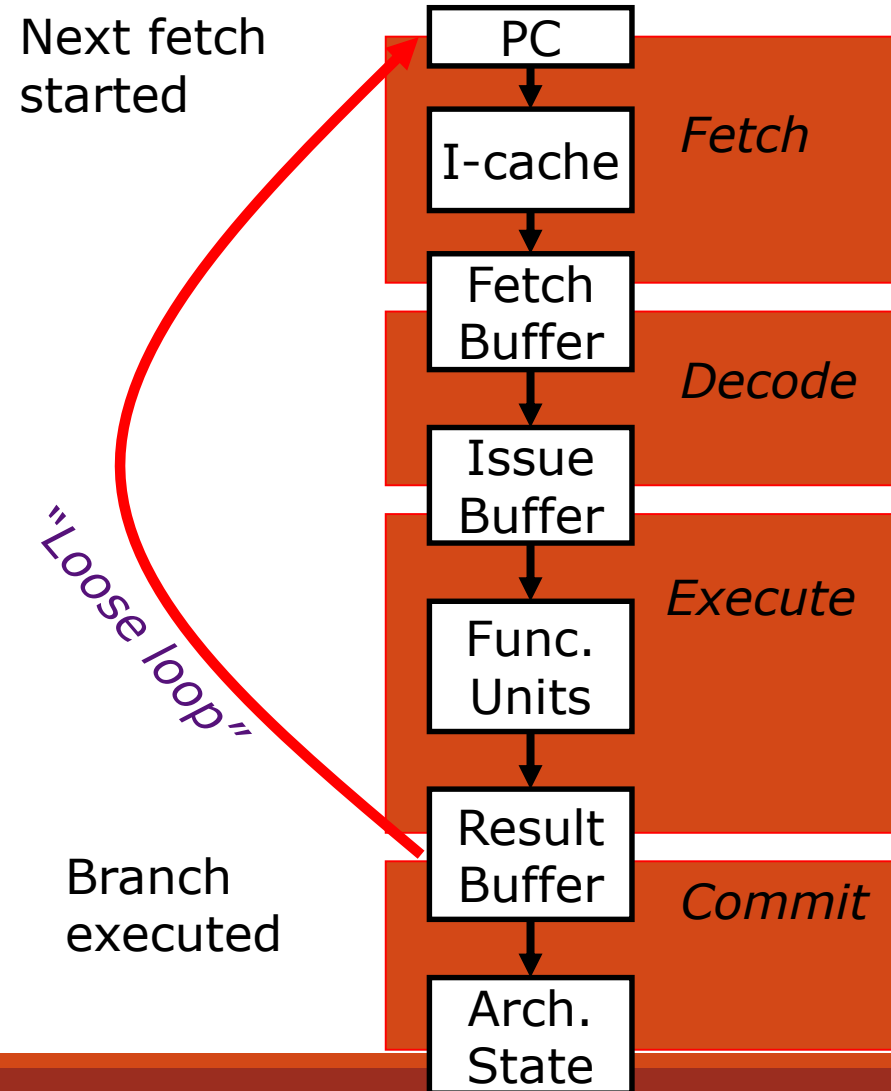


Control Flow Penalty

*Modern processors may have
> 10 pipeline stages between
next PC calculation and branch
resolution !*

*How much work is lost if
pipeline doesn't follow
correct instruction flow?*

~ Loop length x pipeline width



Average Run-Length between Branches

Average dynamic instruction mix from SPEC92:

	SPECint92	SPECfp92
ALU	39 %	13 %
FPU Add		20 %
FPU Mult		13 %
load	26 %	23 %
store	9 %	9 %
branch	16 %	8 %
other	10 %	12 %

SPECint92: *compress, eqntott, espresso, gcc , li*

SPECfp92: *doduc, ear, hydro2d, mdijdp2, su2cor*

What is the average *run length* between branches?

RISC Branches and Jumps

Each instruction fetch depends on one or two pieces of information from the preceding instruction:

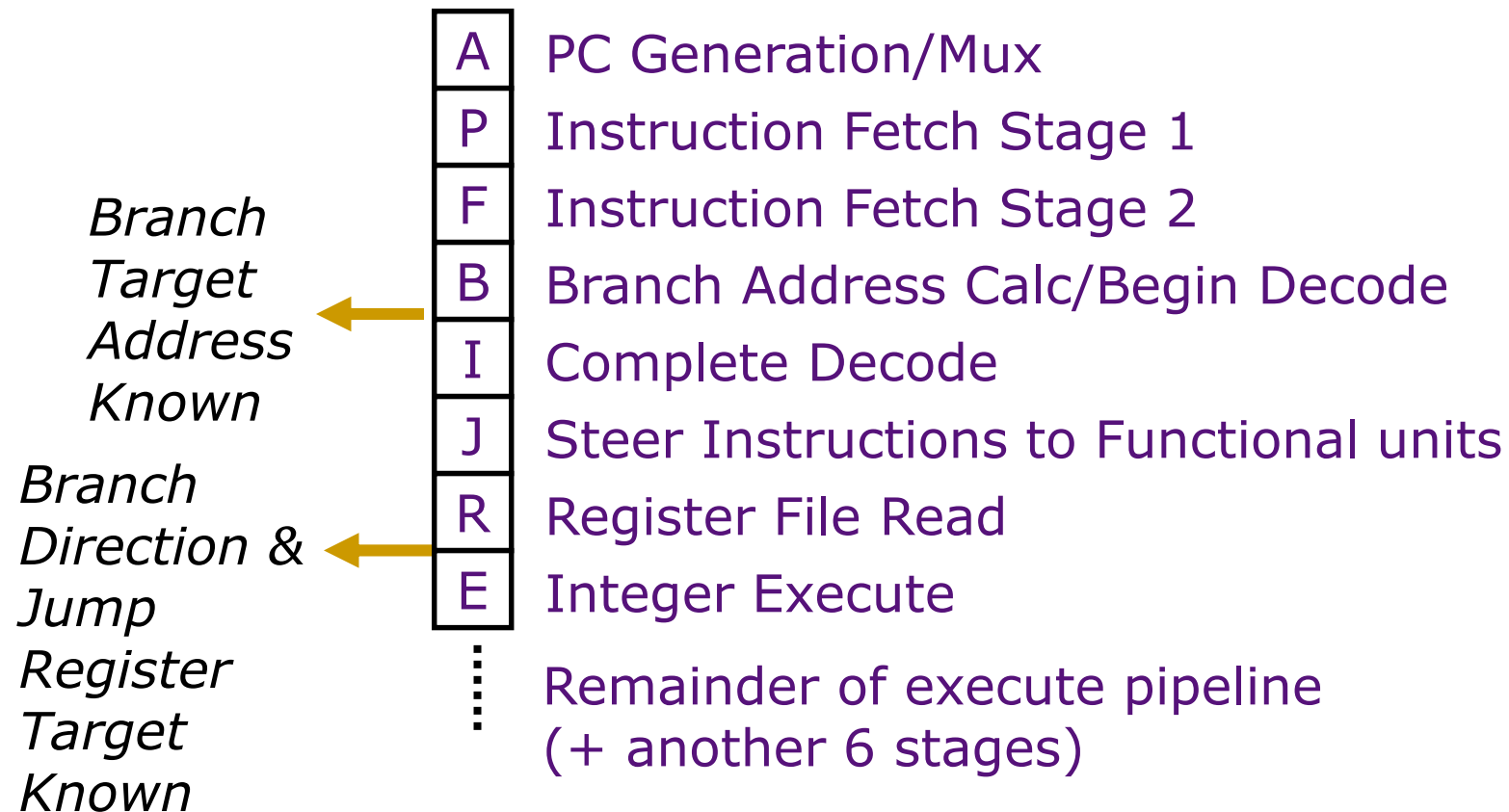
- 1) Is the preceding instruction a taken branch?
- 2) If so, what is the target address?

<i>Instruction</i>	<i>Taken known?</i>	<i>Target known?</i>
J	After Inst. Decode	After Inst. Decode
JR	After Inst. Decode	After Reg. Fetch
BEQZ/BNEZ	After Execute/ Reg. Fetch*	After Inst. Decode

*Assuming zero detect on register read

Realistic Branch Penalties

UltraSPARC-III instruction fetch pipeline stages
(in-order issue, 4-way superscalar, 750MHz, 2000)



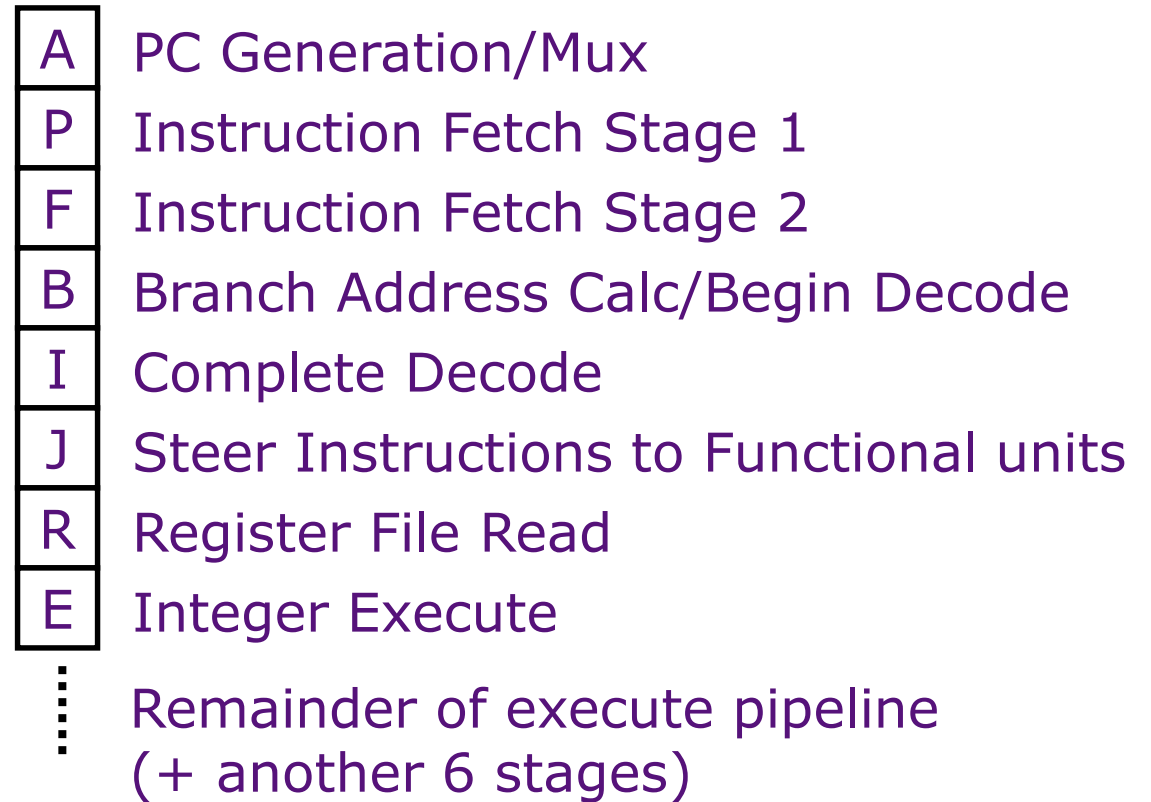
Even longer branch delays

When can we resolve the branch in this example?

```
DIV R1 ← R2, R3
BEQZ R3, 0x1000
```

What about this example?

```
LD R1 ← 0(R2)
BEQZ R1, 0x1000
```



Reducing Control Flow Penalty

Software solutions

- *Eliminate branches - loop unrolling*
Increases the run length
- *Reduce resolution time - instruction scheduling*
Compute the branch condition as early as possible (of limited value – why?)

Hardware solutions

- Find something else to do architecturally
 - *delay slots* - replace pipeline bubbles with useful work
(requires software cooperation, exposes uarch somewhat)
- *Speculate - branch prediction*
Speculative execution of instructions beyond the branch
Useful in both in- and out-of-order processors

Branch Prediction

Motivation:

Branch penalties limit performance of deeply pipelined processors

Modern branch predictors have high accuracy (>95%) and can reduce branch penalties significantly

Required hardware support:

Prediction structures:

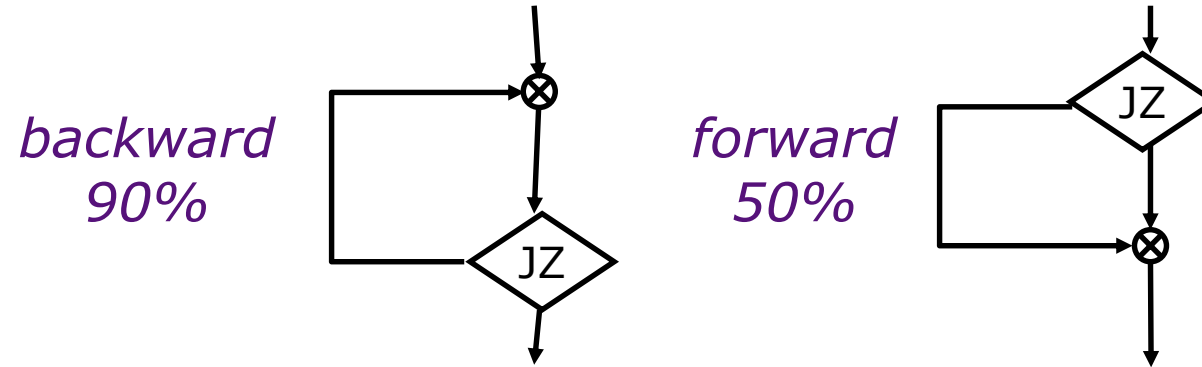
- Branch history tables, branch target buffers, etc.

Mispredict recovery mechanisms:

- *Keep result computation separate from commit*
- Kill instructions following branch in pipeline
- Restore state to state following branch

Static Branch Prediction

Overall probability a branch is taken is $\sim 60\text{-}70\%$ but:



ISA can attach preferred direction semantics to branches,
e.g., Motorola MC88110

bne0 (preferred taken) beq0 (not taken)

ISA can allow arbitrary choice of statically predicted direction,
e.g., HP PA-RISC, Intel IA-64

typically reported as $\sim 80\%$ accurate

Is Static Prediction Enough?

UltraSPARC-III

- Mispredicted branches have penalty of 6 cycles
- 4-wide issue

Wasted work per branch @ 80% accuracy:

20% misprediction rate

× 6 stages

× 4-way issue

= 4.8 wasted instructions / branch

Branches are 15-20% of instructions!

Dynamic Branch Prediction

learning based on past behavior

Temporal correlation

The way a branch resolves may be a good predictor of the way it will resolve at the next execution

Example? Loops

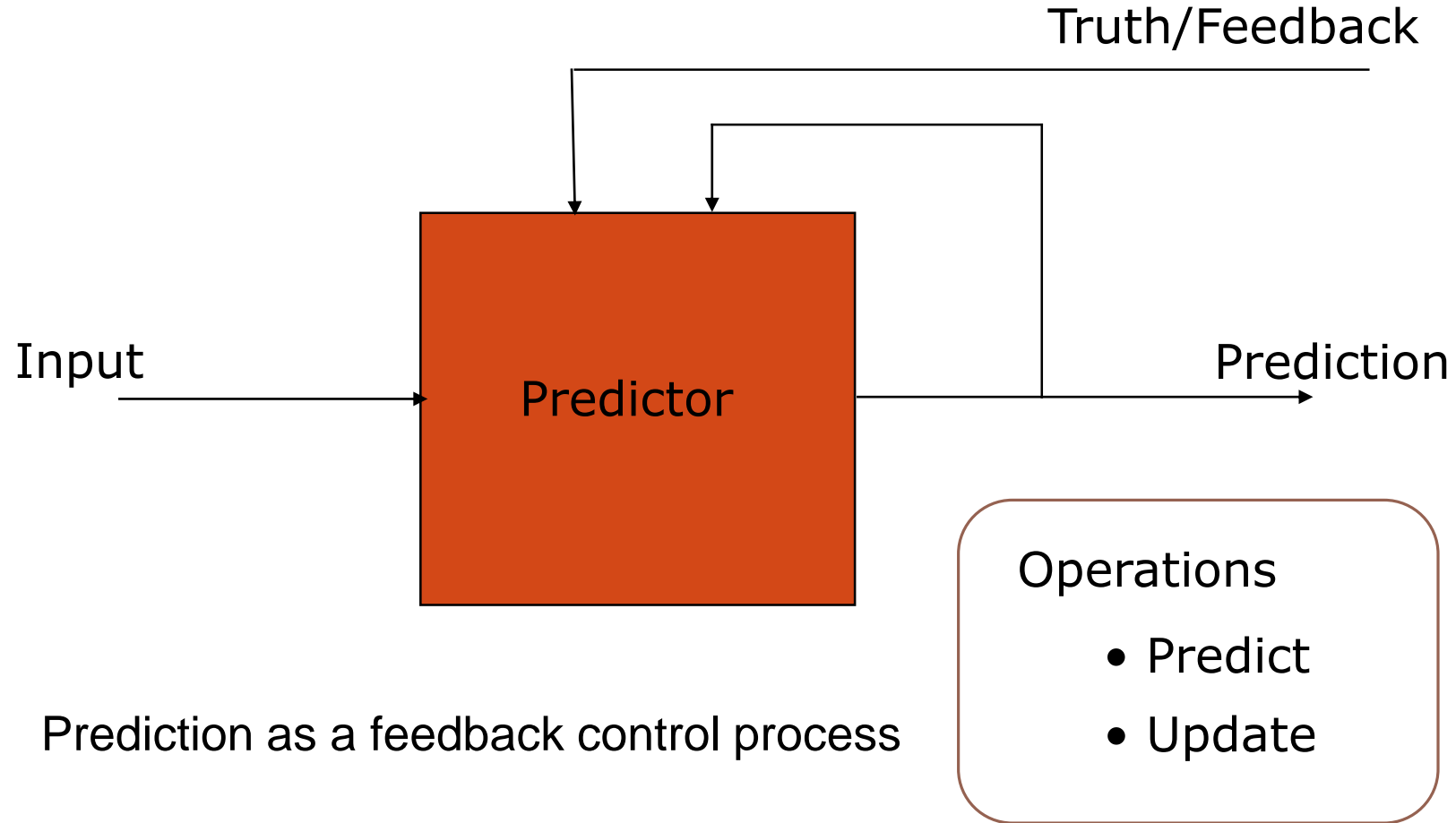
Spatial correlation

Several branches may resolve in a highly correlated manner (*a preferred path of execution*)

Example? Nest if-blocks, exception conditions, etc

Echoes of temporal/spatial locality in caches...

Dynamic Prediction



Predictor Primitive

Emer & Gloy, 1997

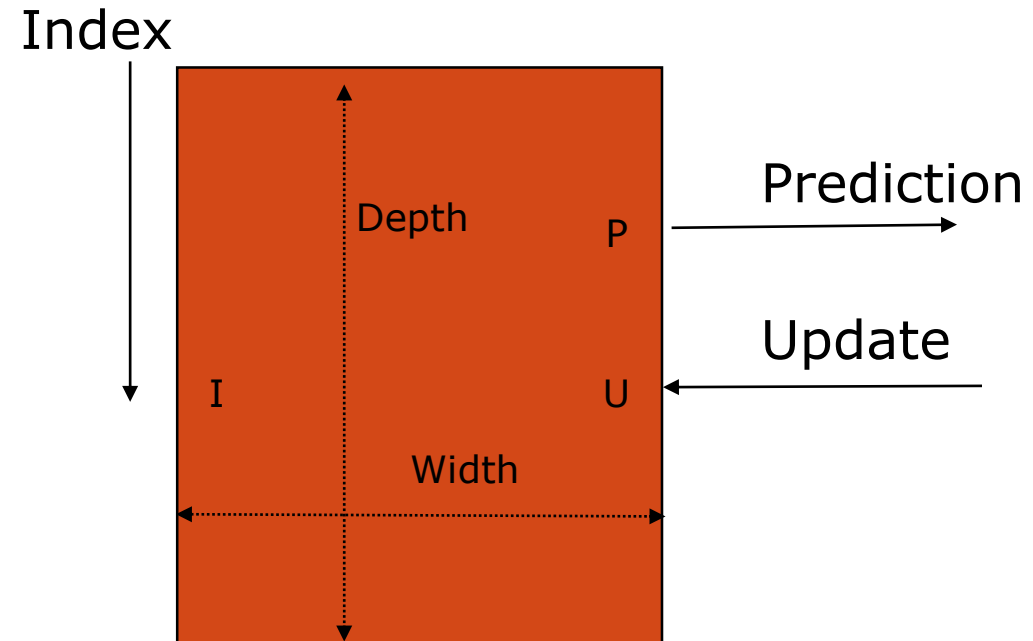
Indexed table holding values

Operations

- Predict
- Update

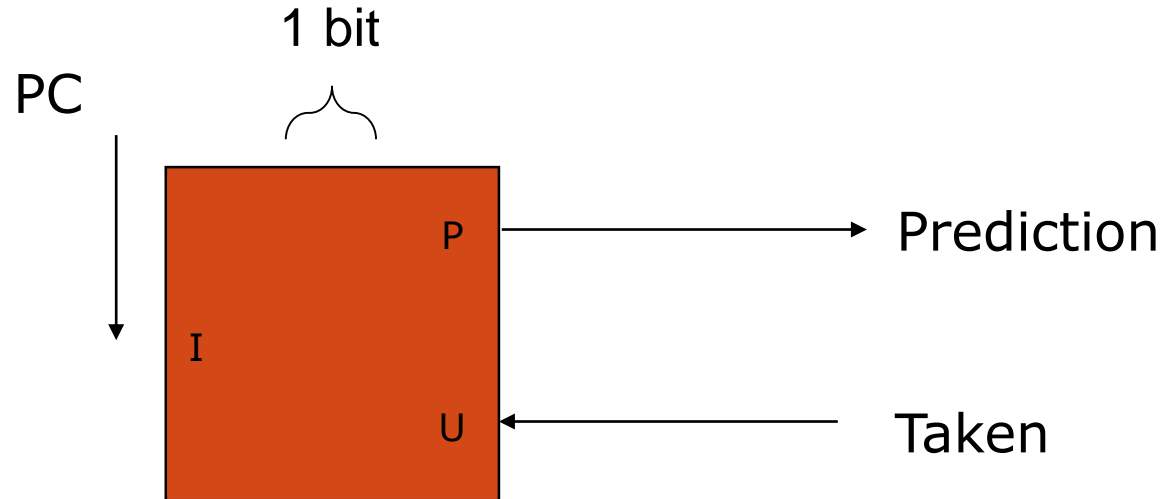
Algebraic notation

$$\text{Prediction} = P[\text{Width}, \text{Depth}](\text{Index}; \text{Update})$$



One-bit Predictor

Simple temporal prediction



$$A_{21064}(PC; T) = P[1, 2K](PC; T)$$

What happens on loop branches?

At best, mispredicts twice for every use of loop.

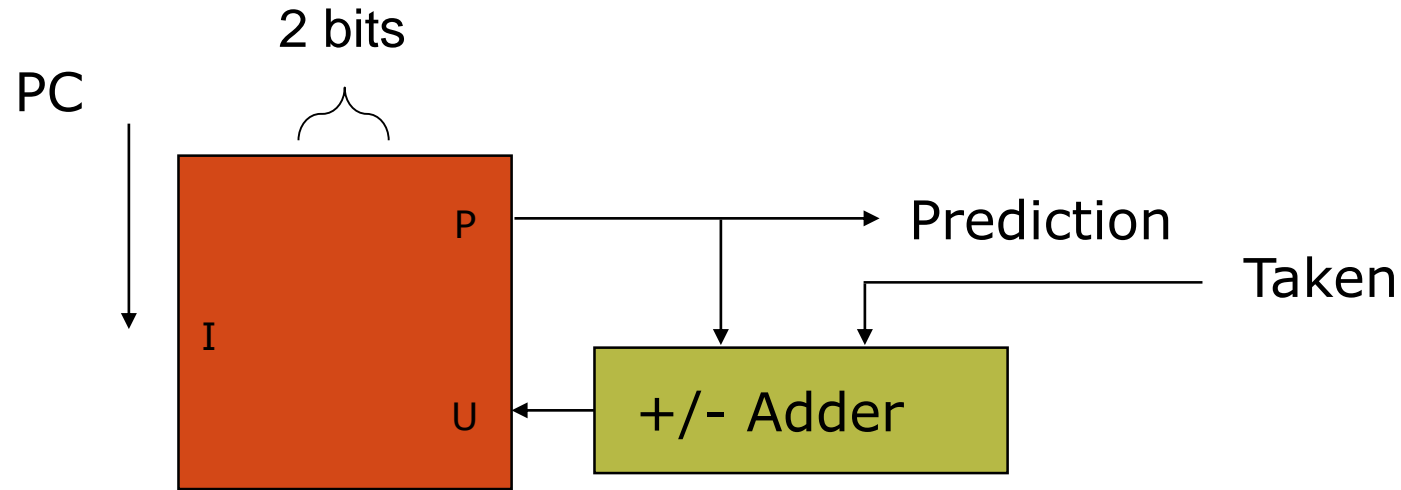
Branch Prediction Bits

- Assume 2 BP bits per instruction
- Use saturating counter

On \neg taken \rightarrow	\rightarrow On taken	1	1	Strongly taken
		1	0	Weakly taken
		0	1	Weakly \neg taken
		0	0	Strongly \neg taken

Two-bit Predictor

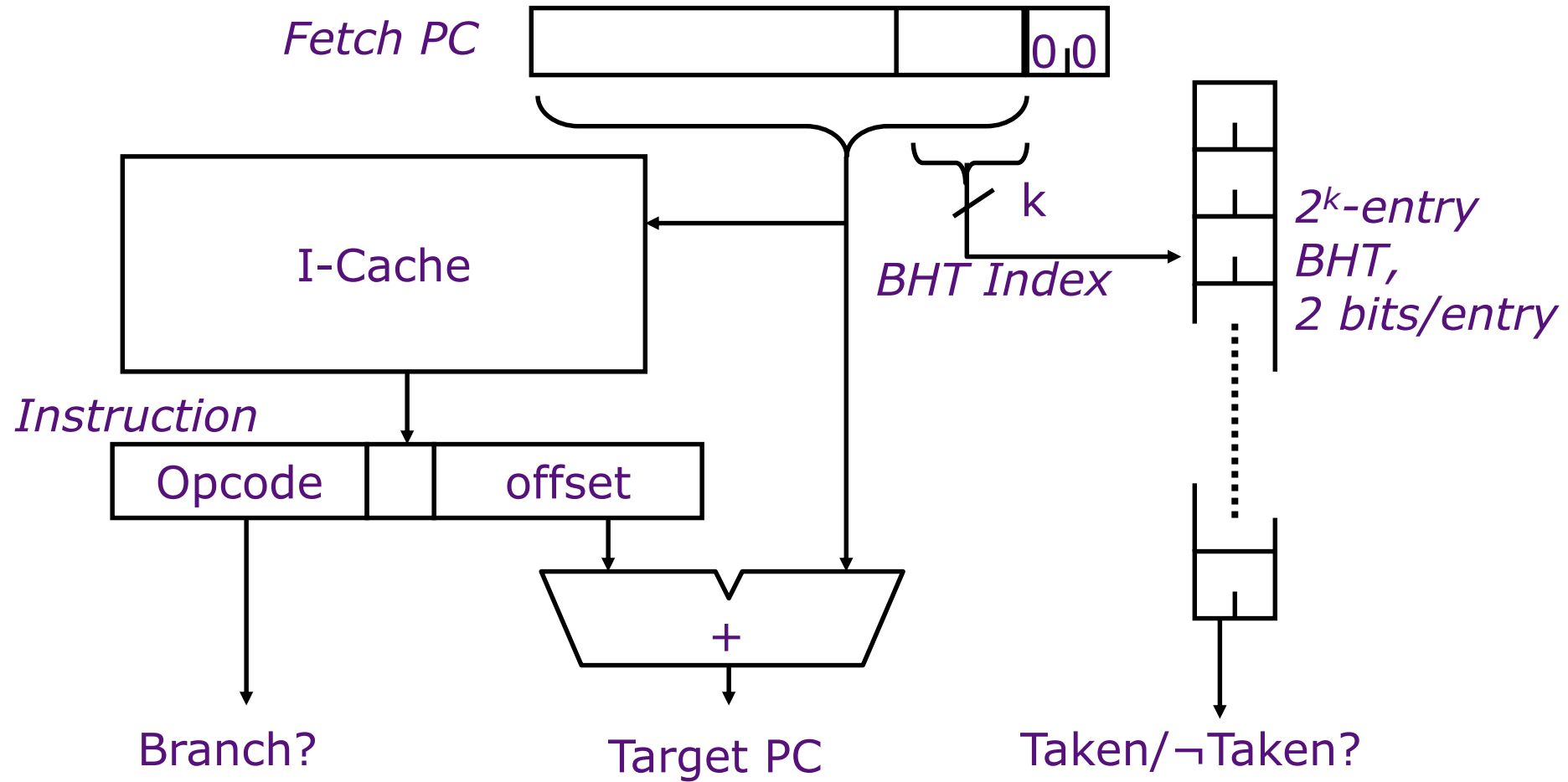
Smith, 1981



$\text{Counter}[W,D](I; T) = P[W, D](I; \text{if } T \text{ then } P+1 \text{ else } P-1)$

$A21164(PC; T) = \text{MSB}(\text{Counter}[2, 2K](PC; T))$

Branch History Table



4K-entry BHT, 2 bits/entry, ~80-90% correct predictions

Exploiting Spatial Correlation

Yeh and Patt, 1992

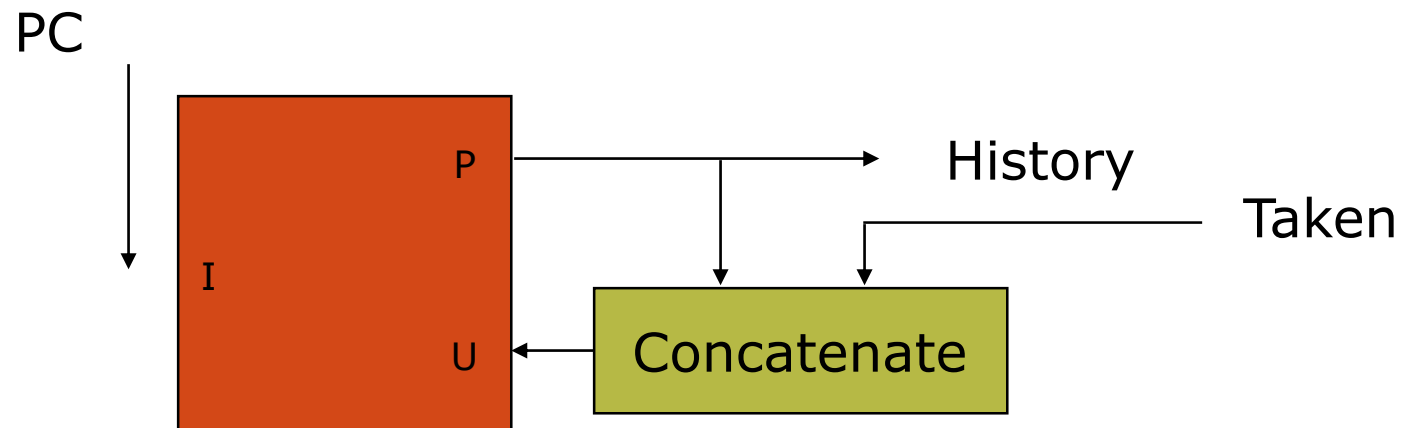
```
if (x[i] < 7) then  
    y += 1;  
if (x[i] < 5) then  
    c -= 4;
```

If first condition false, second condition also false

History register, H, records the direction of the last N branches executed by the processor

History Register

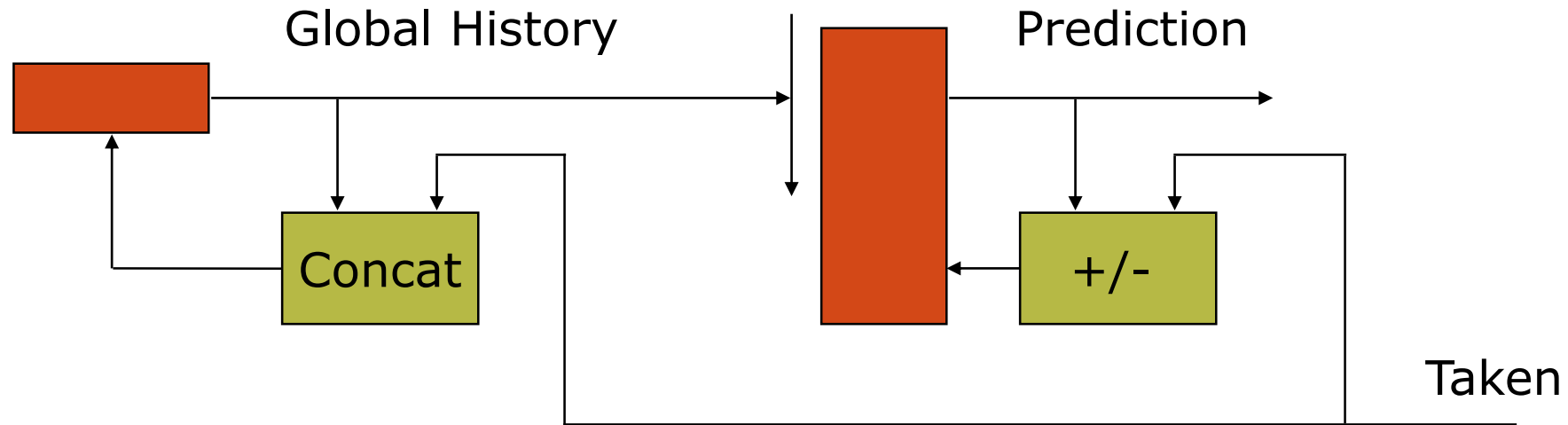
An array that stores the recent history (T/NT) of branches at different branch PCs.



$$\text{History}(\text{PC}, T) = P(\text{PC}; P \parallel T)$$

Global History

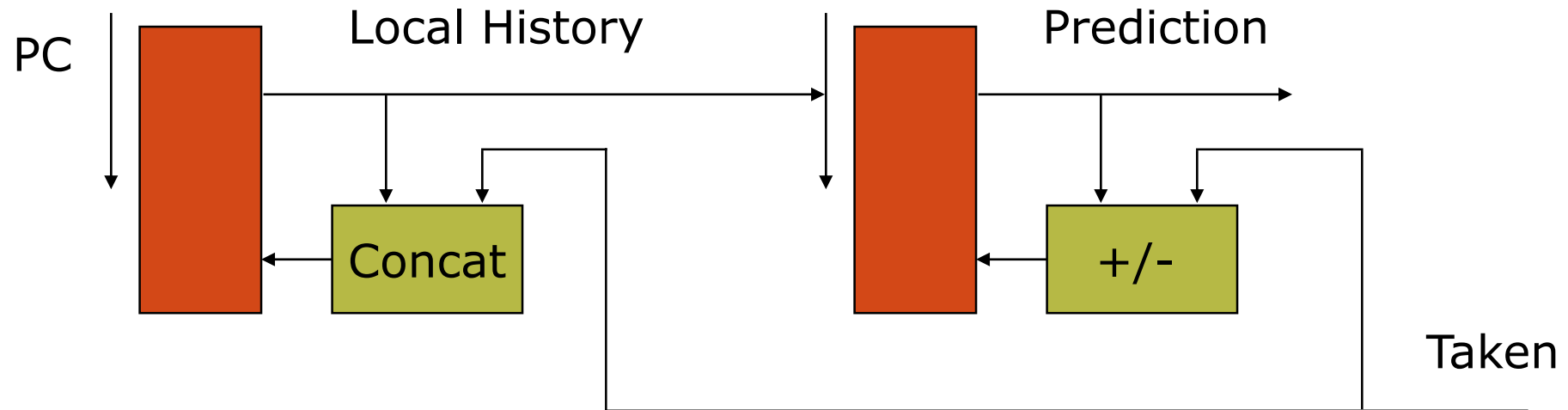
Use the global branch history (not per-PC) to predict subsequent branches.



$$\text{GHist}(\text{;T}) = \text{MSB}(\text{Counter}(\text{History}(0, \text{T}); \text{T}))$$

Can we take advantage of pattern at a particular PC?

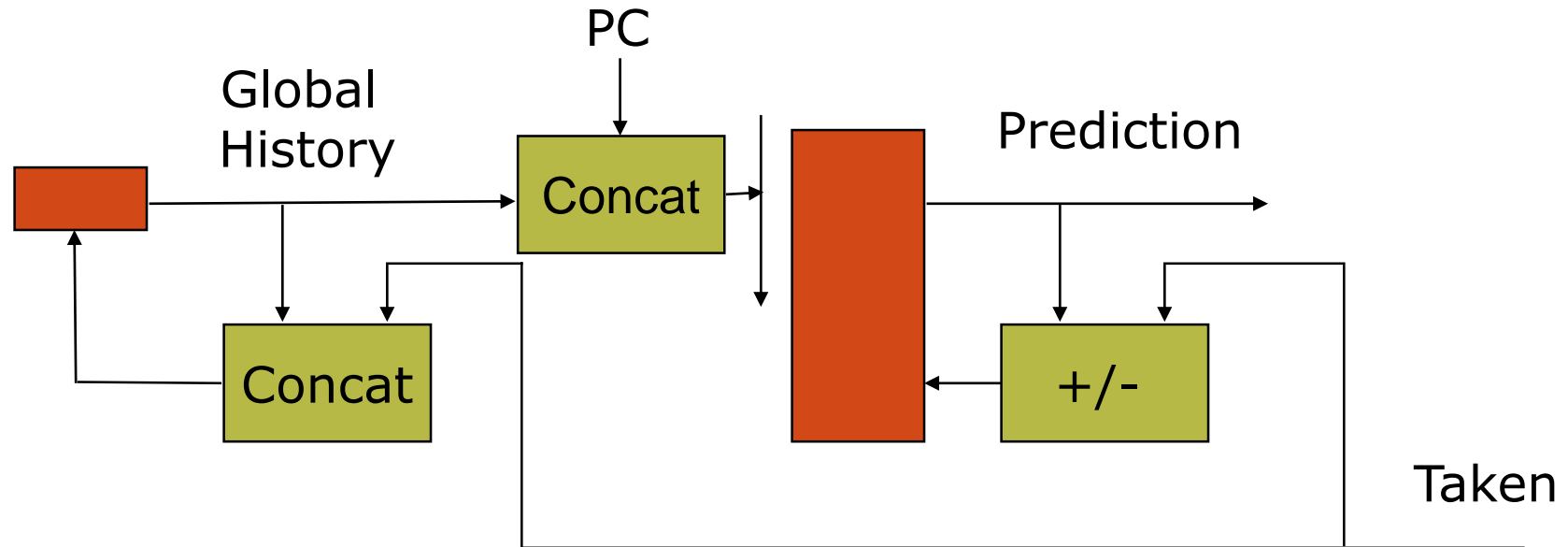
Local History



$$\text{LHist}(\text{PC}, T) = \text{MSB}(\text{Counter}(\text{History}(\text{PC}; T); T))$$

*Can we take advantage of the **global** pattern at a particular PC?*

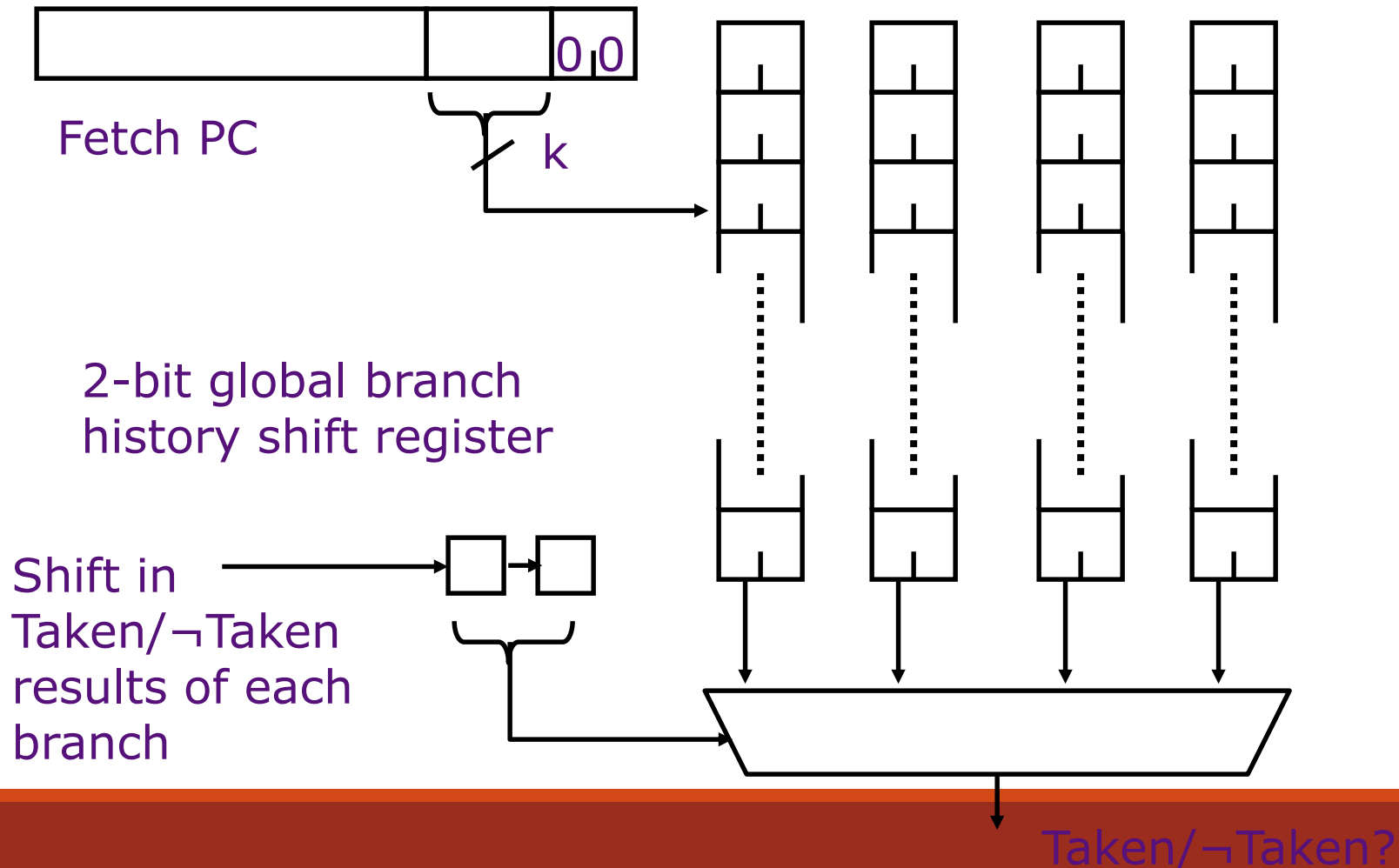
Two-level Predictor



$$2Level(PC, T) = MSB(Counter(History(0; T) || PC; T))$$

Two-Level Branch Predictor

Pentium Pro uses the result from the last two branches to select one of the four sets of BHT bits (~95% correct)



Which predictor is best?

Many different predictors were proposed

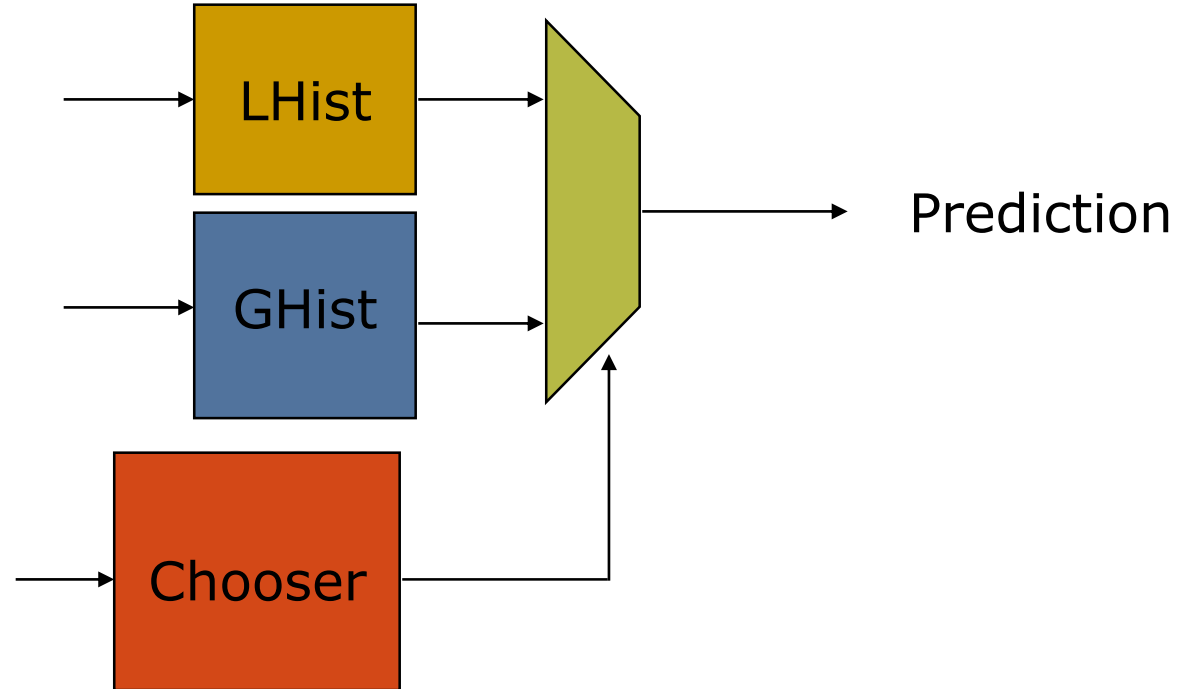
Common principles & ingredients

- Temporal / spatial correlation
- Saturating counters, concatenation

Each handles particular patterns well, but none universal

What to do?

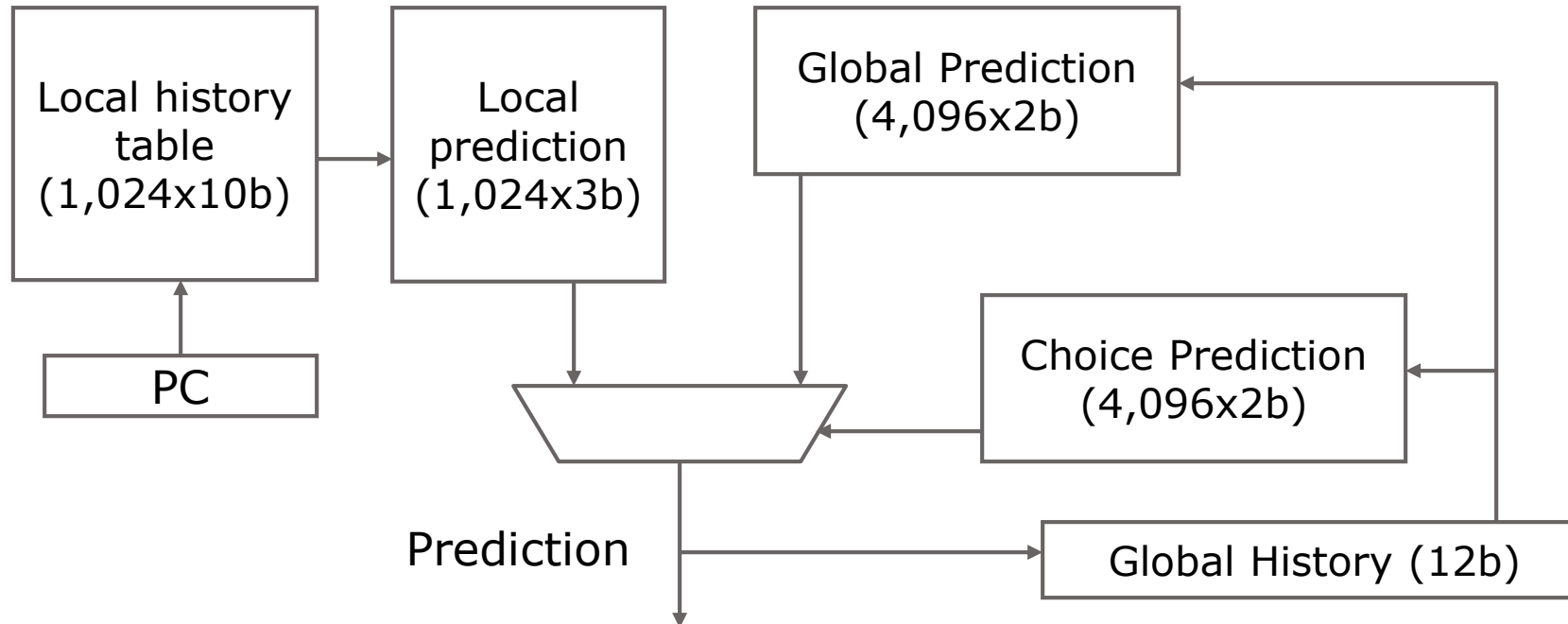
Predict the Predictor!



$$\begin{aligned}\text{Chooser} &= \text{MSB}(P(\text{PC}; P + (A==T) - (B==T))) \\ &\text{or} \\ \text{Chooser} &= \text{MSB}(P(\text{GHist}(\text{PC}; T); P + (A==T) - (B==T)))\end{aligned}$$

Tournament Branch Predictor

(Alpha 21264)



Choice predictor learns whether best to use local or global branch history in predicting next branch

Global history is speculatively updated but restored on mispredict

Claim 90-100% success on range of applications

Further down the rabbit hole...

Neural-network-based, “perceptron” branch predictors

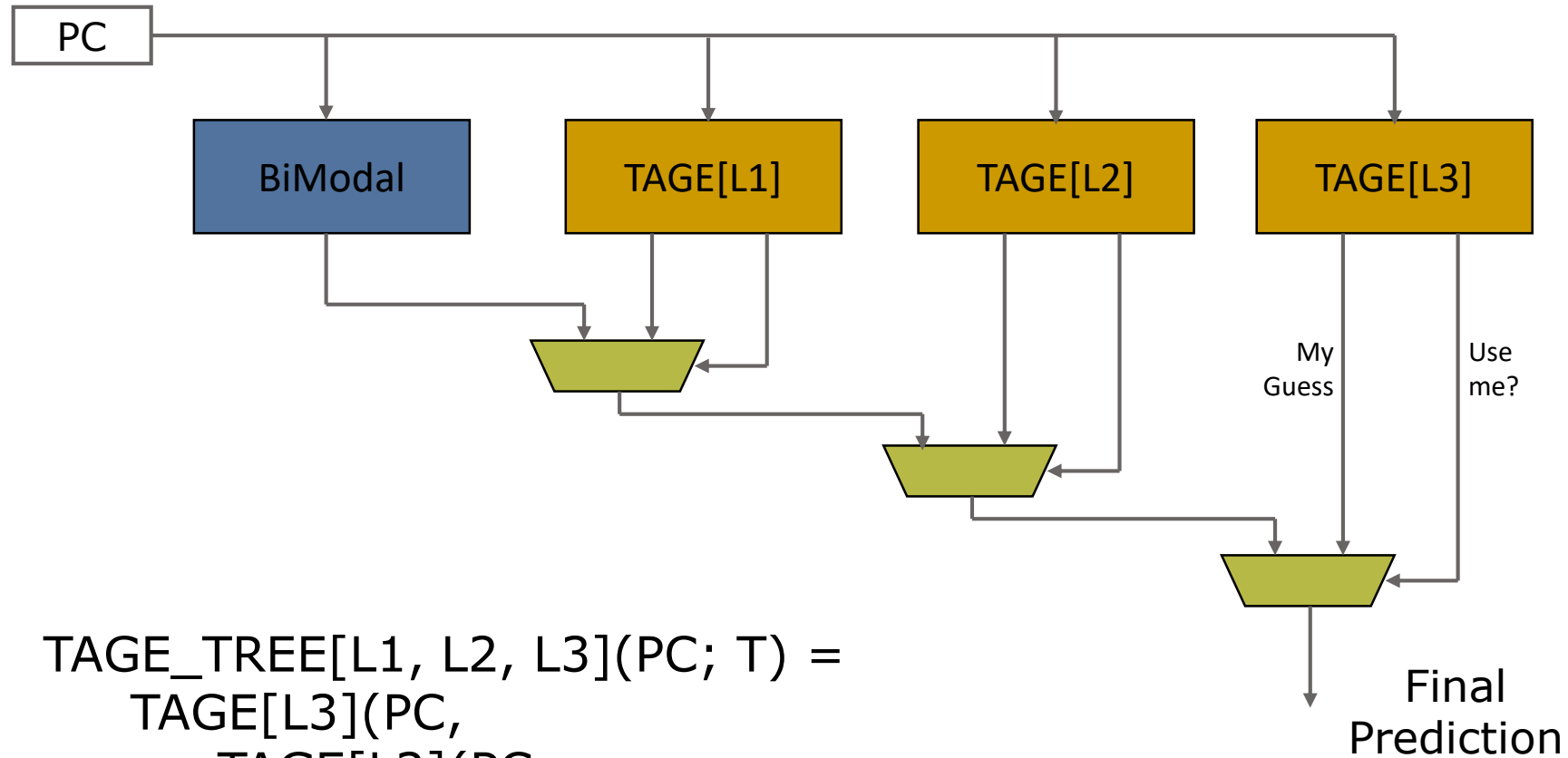
- [Vintan, IJCNN '99][Jiminez, HPCA '01]
- High prediction accuracy, but more computation
- Implemented in AMD Piledriver, AMD Ryzen

TAGE predictor

- TAgged GEometric predictor [Seznac, JILP '06]
- Keep multiple predictions with different history lengths
- Partially tag predictions to avoid false matches
- Only provide prediction on tag match
- Rumored to be what Intel uses

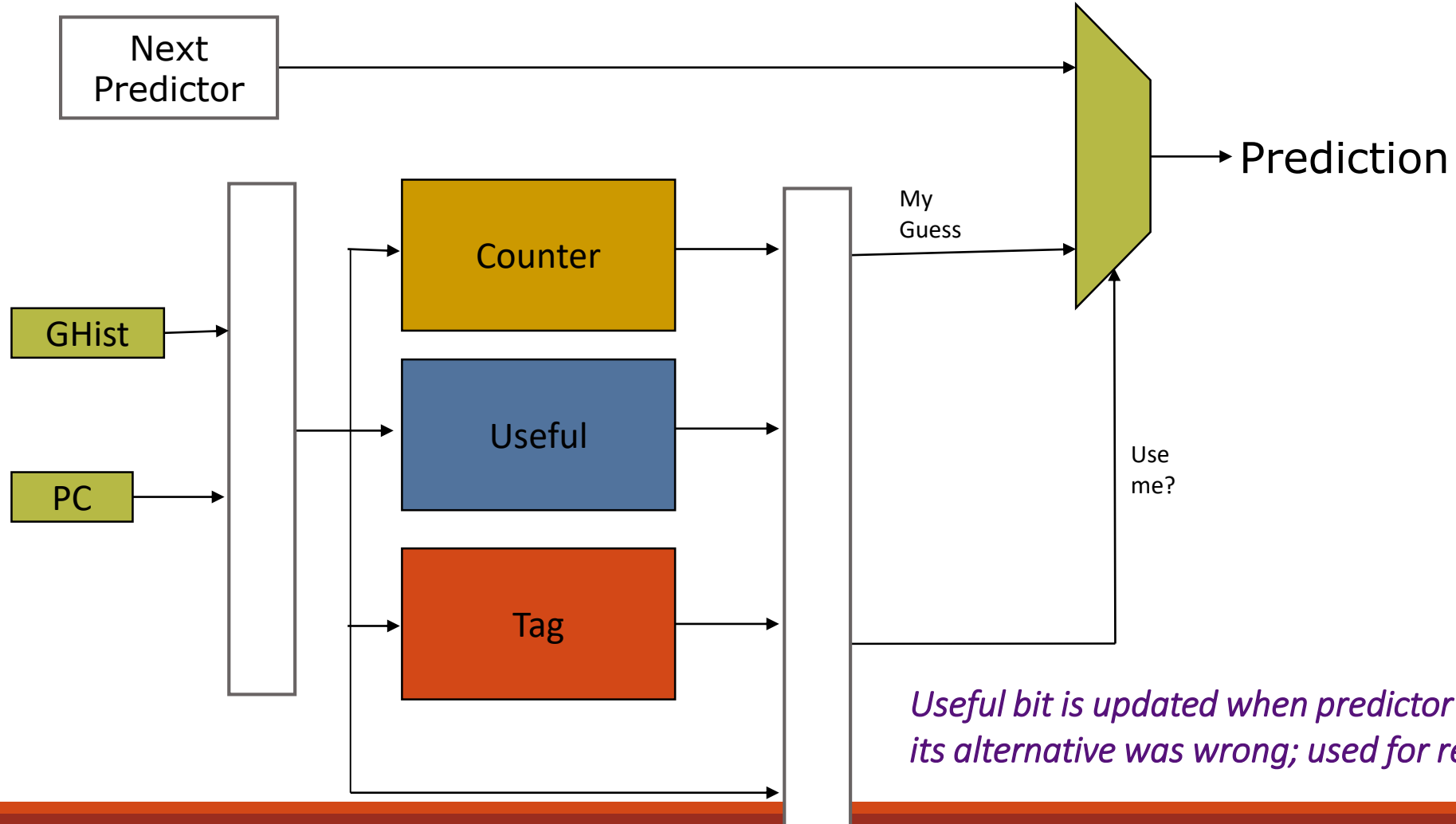
TAGE predictor

Seznec & Michaud, 2006



```
TAGE_TREE[L1, L2, L3](PC; T) =  
  TAGE[L3](PC,  
    TAGE[L2](PC,  
      TAGE[L1](PC, Bimodal(PC;T)  
;T) ;T ;T)
```

TAGE component



TAGE predictor component

TAGE[L](PC, NEXT; T) =

idx = hash(PC, GHIST[L](;T))

tag = hash(PC, GHIST[L](;T))

TAGE.U = SA(idx, tag; ((TAGE == T) && (NEXT != T)) ? 1 : SA)

TAGE.Counter = SA(idx, tag; T ? SA+1 : SA-1)

use_me = TAGE.U && isStrong(TAGE.Counter)

TAGE = use_me ? MSB(TAGE.Counter) : NEXT

Notes:

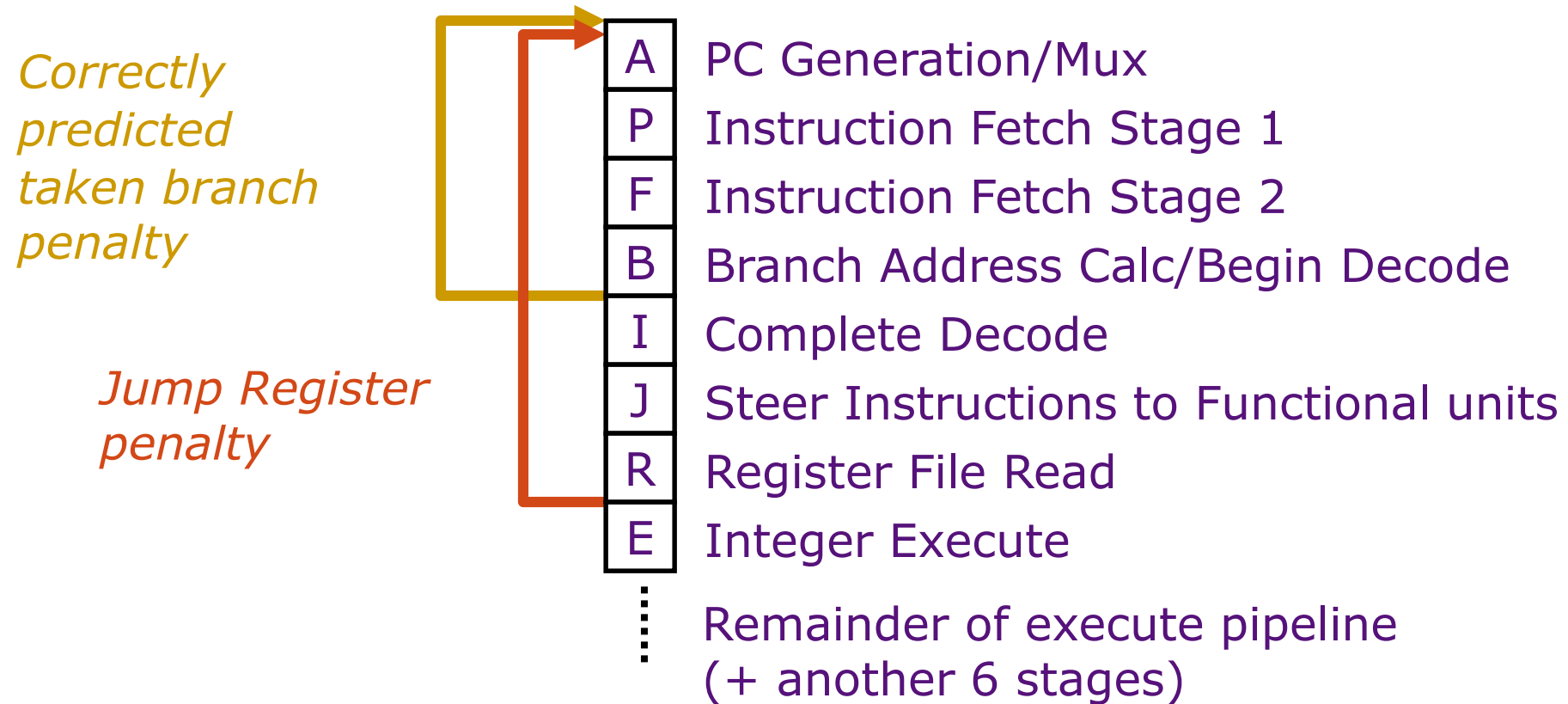
- SA is a 'set associative' structure

- SA allocation occurs on mispredict (not shown)

- TAGE.U cleared on global counter saturation

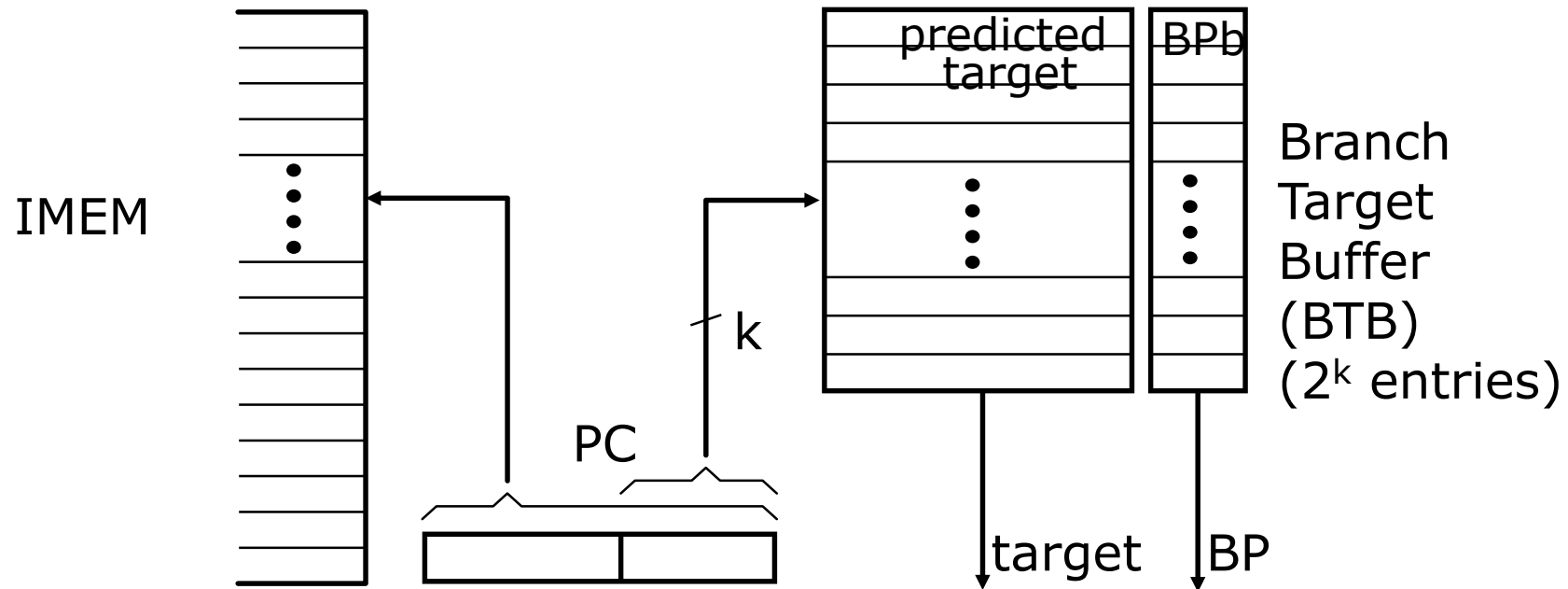
Limitations of branch predictors

Only predicts branch direction. Therefore, cannot redirect fetch stream until after branch target is determined.



UltraSPARC-III fetch pipeline

Branch Target Buffer (untagged)

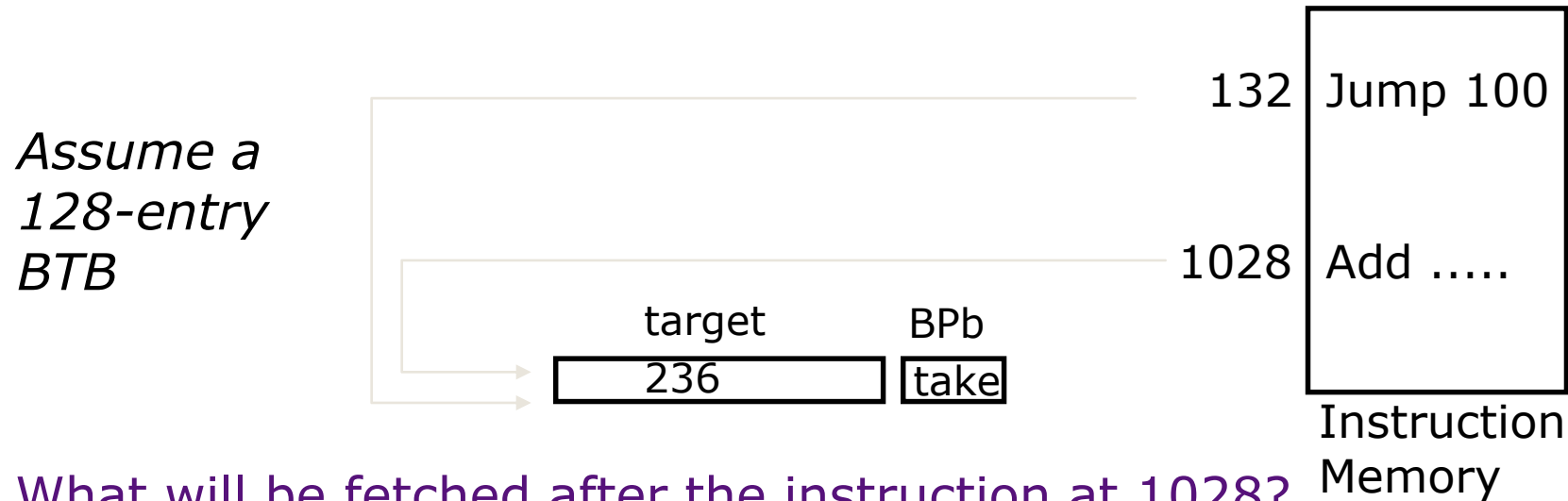


BP bits are stored with the predicted target address.

Fetch: *If (BP=taken) then NextPC=target else NextPC=PC+4*

Later: *Check prediction; if wrong then flush pipe, update BTB & BP
else update BP only`*

Address Collisions



What will be fetched after the instruction at 1028?

BTB prediction = 236

Correct target = 1032

⇒ *kill* PC=236 and *fetch* PC=1032

Is this a common occurrence?
Can we avoid these bubbles?

BTB is only for Control Instructions

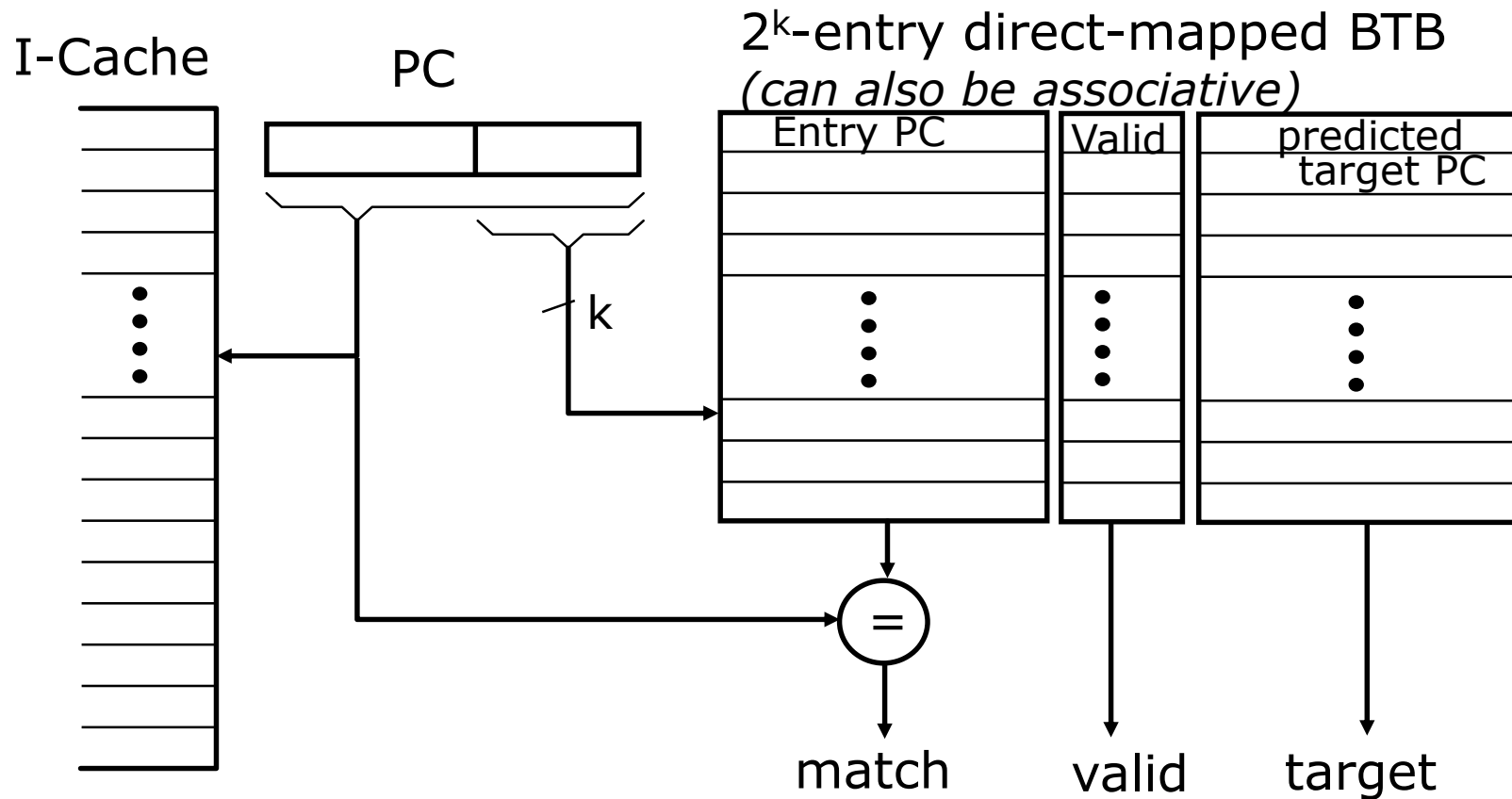
BTB contains useful information for branch and jump instructions only

⇒ Do not update it for other instructions

For all other instructions the next PC is $(PC)+4$!

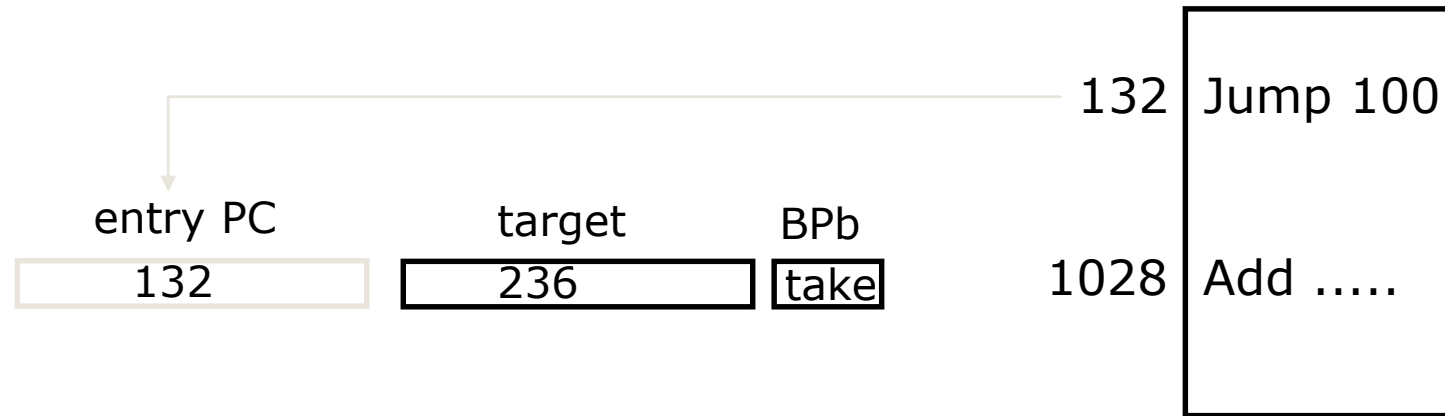
How to achieve this effect without decoding the instruction?

Branch Target Buffer (tagged)



- Keep both the branch PC and target PC in the BTB
- PC+4 is fetched if match fails
- Only ***taken*** branches and jumps held in BTB
- Next PC determined *before* branch fetched and decoded

Consulting BTB Before Decoding

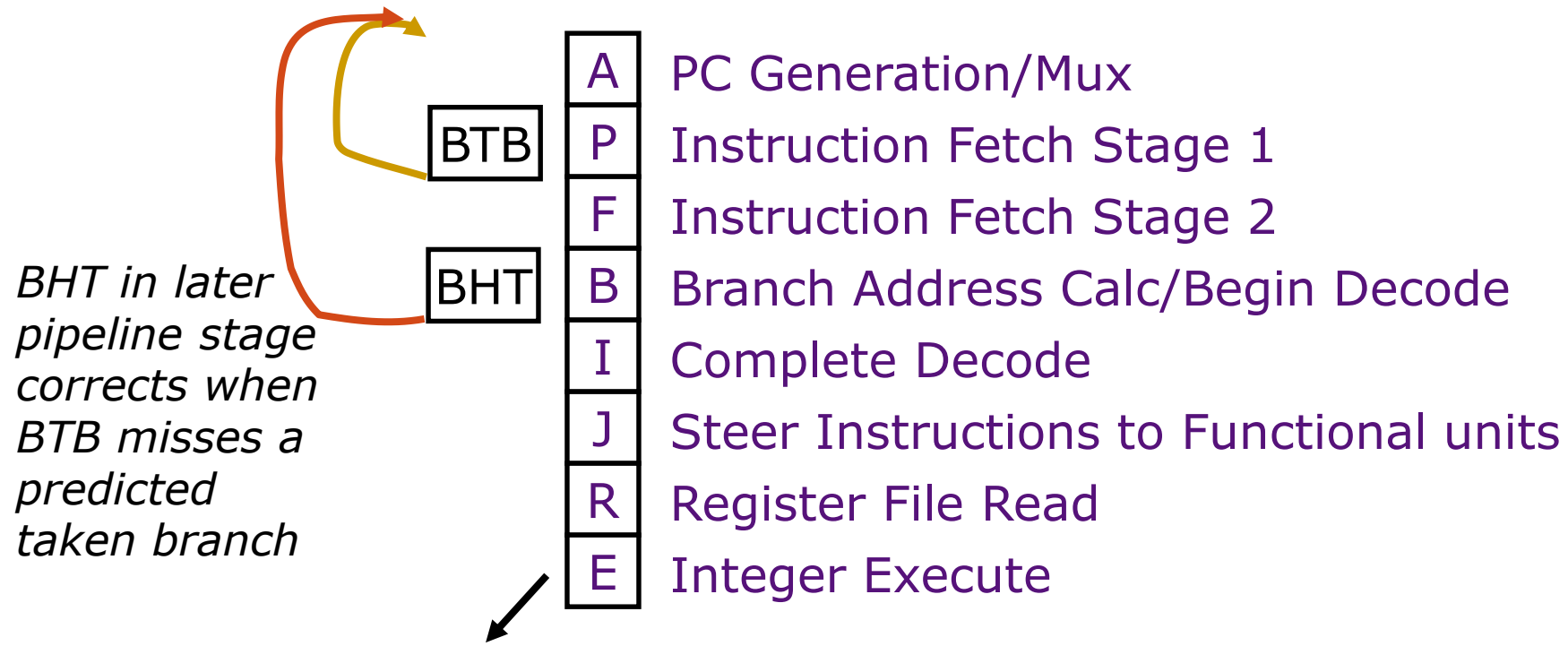


- The match for PC=1028 fails and 1028+4 is fetched
⇒ *eliminates false predictions after ALU instructions*
- BTB contains entries only for control transfer instructions
⇒ *more room to store branch targets*
(but each entry is larger)

Combining BTB and BHT

BTB entries are considerably more expensive than BHT, but can redirect fetches at earlier stage in pipeline and can accelerate indirect branches (JR)

BHT can hold many more entries and is more accurate



Uses of Jump Register (JR)

Switch statements (jump to address of matching case)

BTB works well if same case used repeatedly

Dynamic function call (jump to run-time function address)

BTB works well if same function usually called, (e.g., in C++ programming, when objects have same type in virtual function call)

Subroutine returns (jump to return address)

BTB works well if usually return to the same place

⇒ *Often one function called from many distinct call sites!*

How well does BTB work for each of these cases?

Subroutine Return Stack

Small structure to accelerate JR for subroutine returns, typically much more accurate than BTBs.

```
fa() { fb(); }
```

```
fb() { fc(); }
```

```
fc() { fd(); }
```

*Push call address when
function call executed*

*Pop return address
when subroutine
return decoded*



Overview of branch prediction

