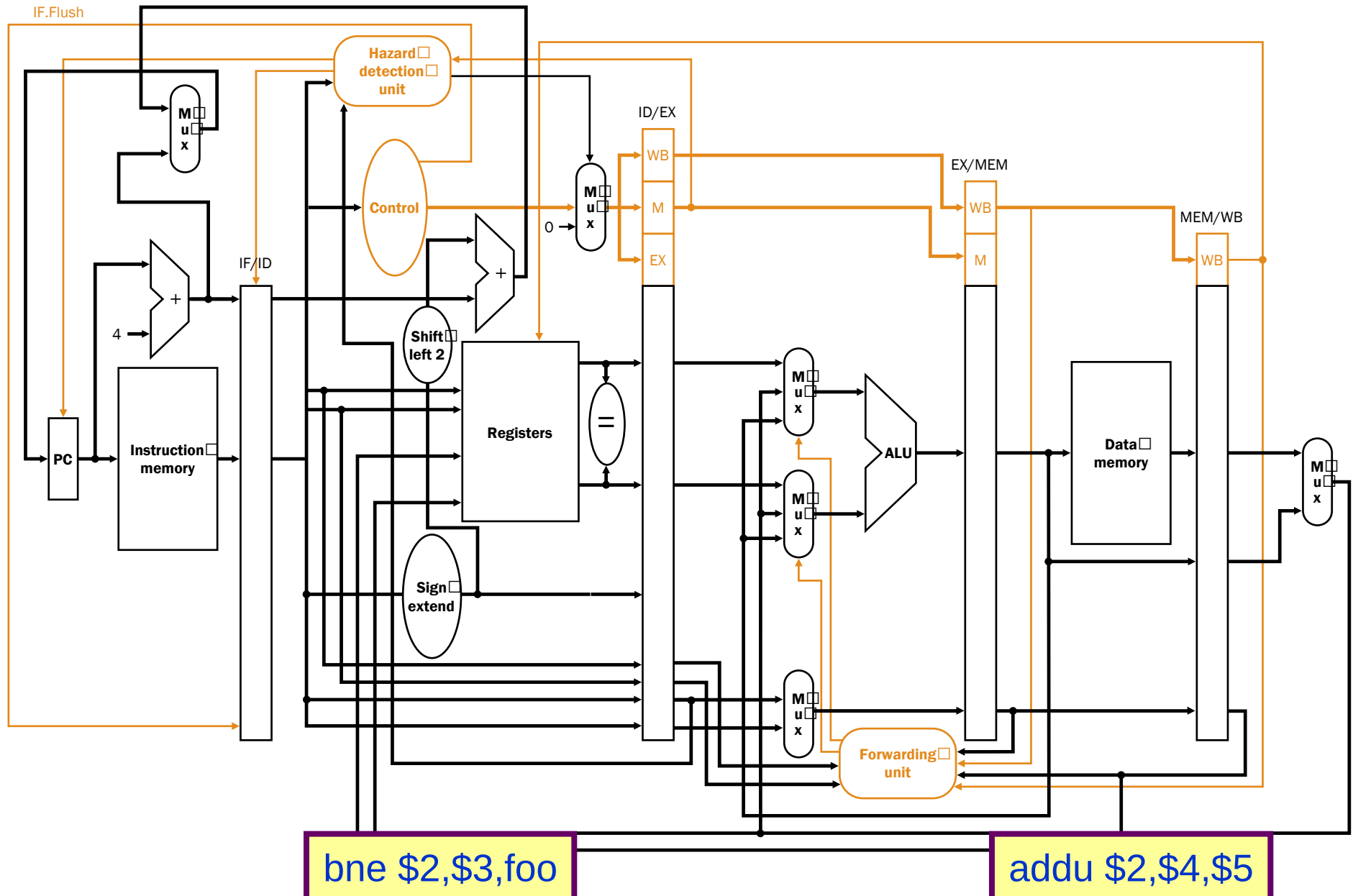
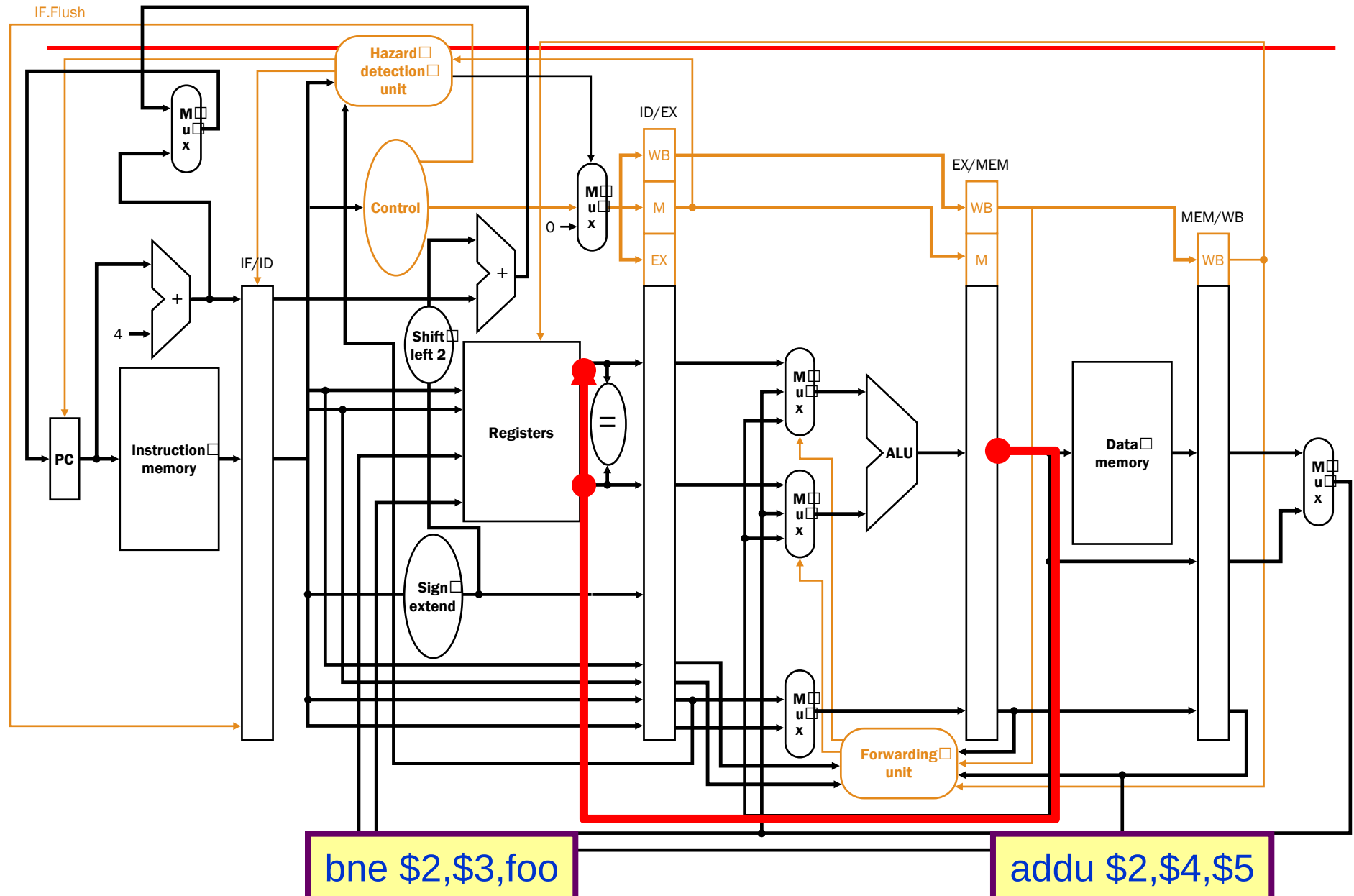


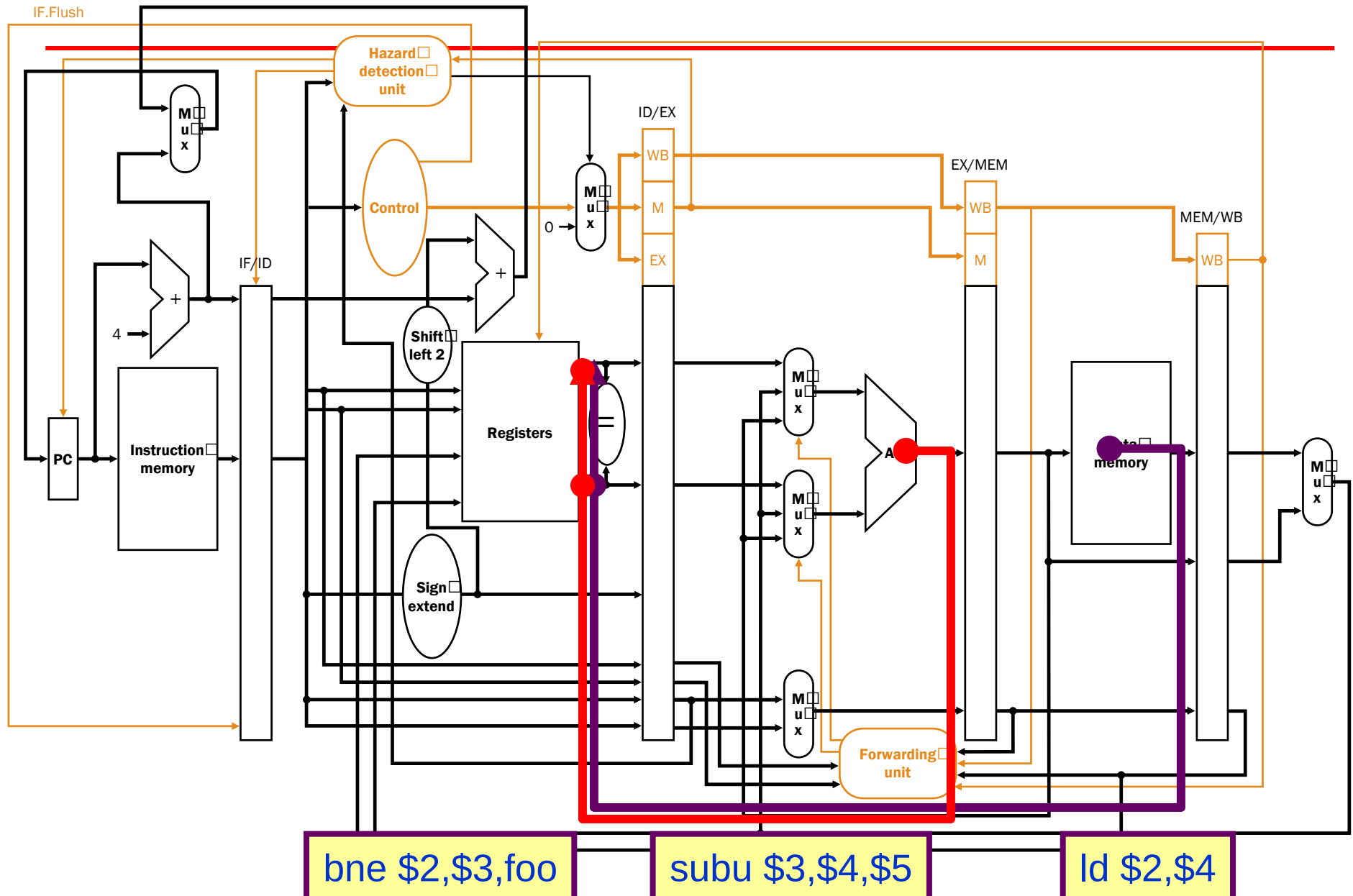
# Reducing the branch delay



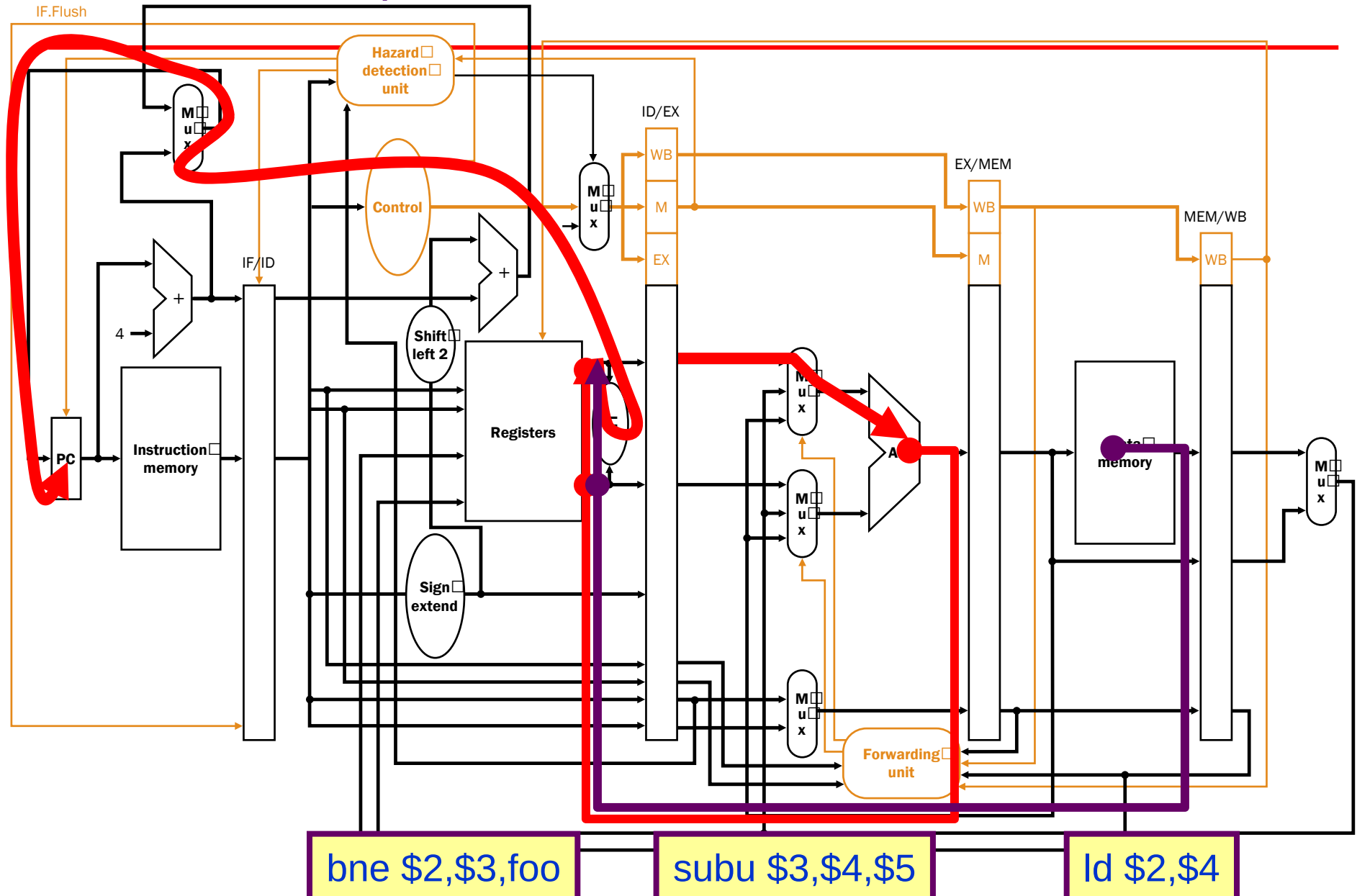
# Branch bypassing – easy case shaves one cycle off branch penalty



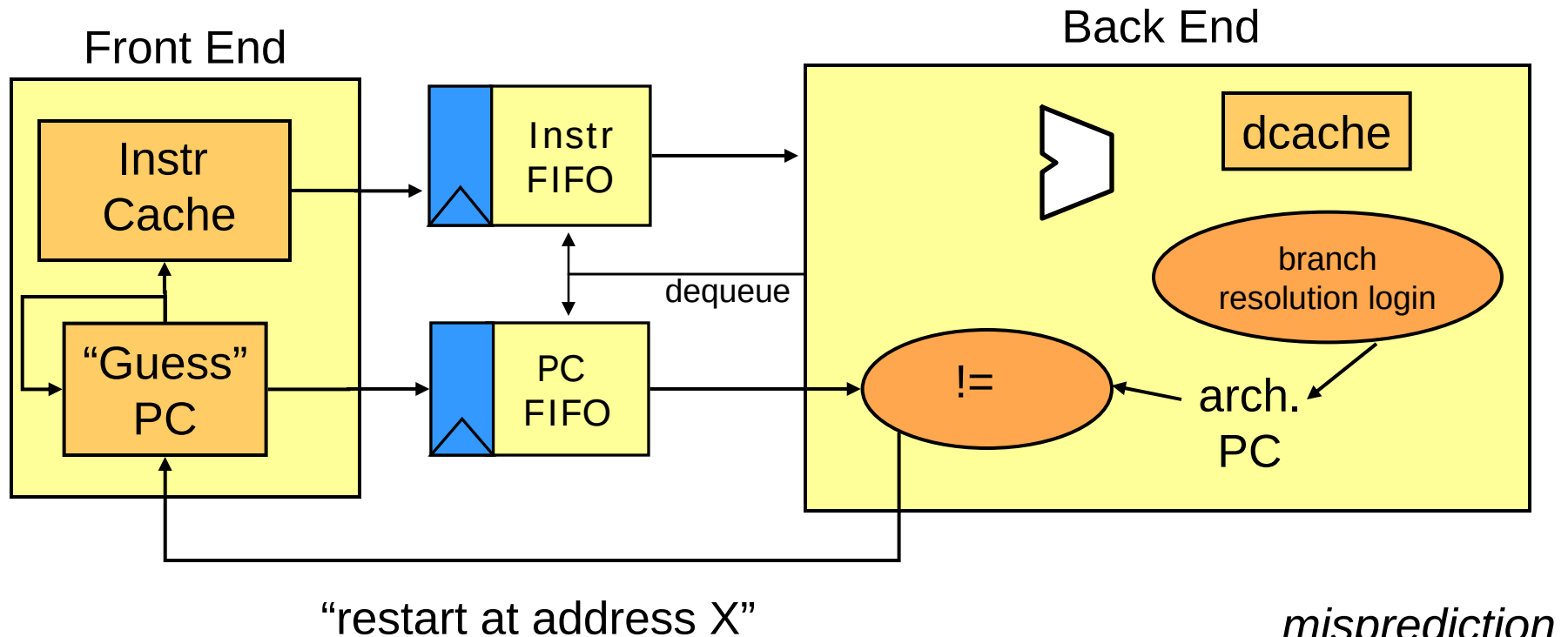
# Branch bypassing – back-to-back deps



Branch handling in decode → lots of ugly paths  
might as well execute branch in EXE stage + use  
better branch prediction



# Branch Prediction from 10,000 ft



**Invariant:  $\text{is\_correct}(\text{PC}) \rightarrow \text{is\_correct}(\text{Instr}[\text{PC}])$**

On restart (*branch misprediction*) must –

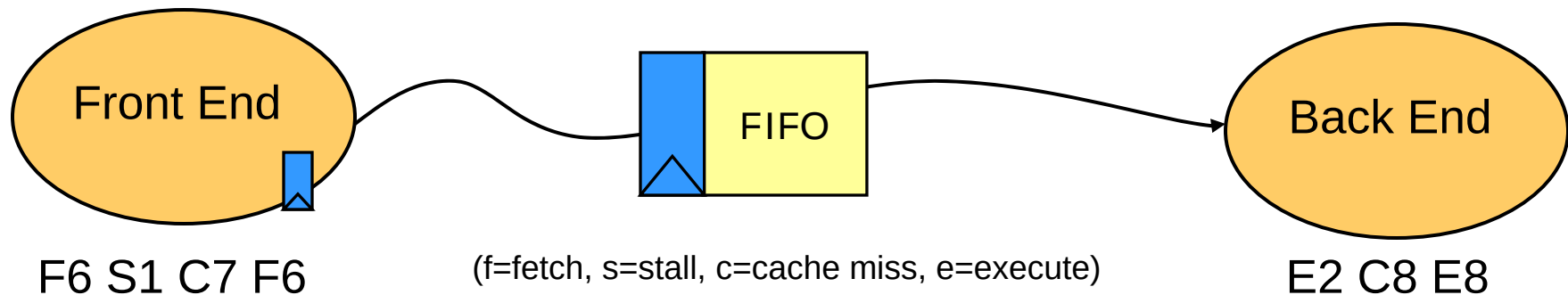
a. kill all incorrectly fetched instructions  
(to ensure correct execution)

b. refill pipeline (takes # cycles == latency of pipeline up to execute stage)

*misprediction  
penalty*

# Aside: Decoupled Execution

*Buffering Smooths Execution and Improves Cycle time by Reducing Stall Propagation*



Cycle →

|      |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|------|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| FE   | F | F | F | F | F | F | S | C | C | C | C | C | C | C | F | F | F | F | F | F |
| FIFO |   | 1 | 1 | 1 | 2 | 3 | 4 | 4 | 4 | 4 | 4 | 3 | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| BE   |   |   | E | E | C | C | C | C | C | C | C | C | E | E | E | E | E | E | E | E |

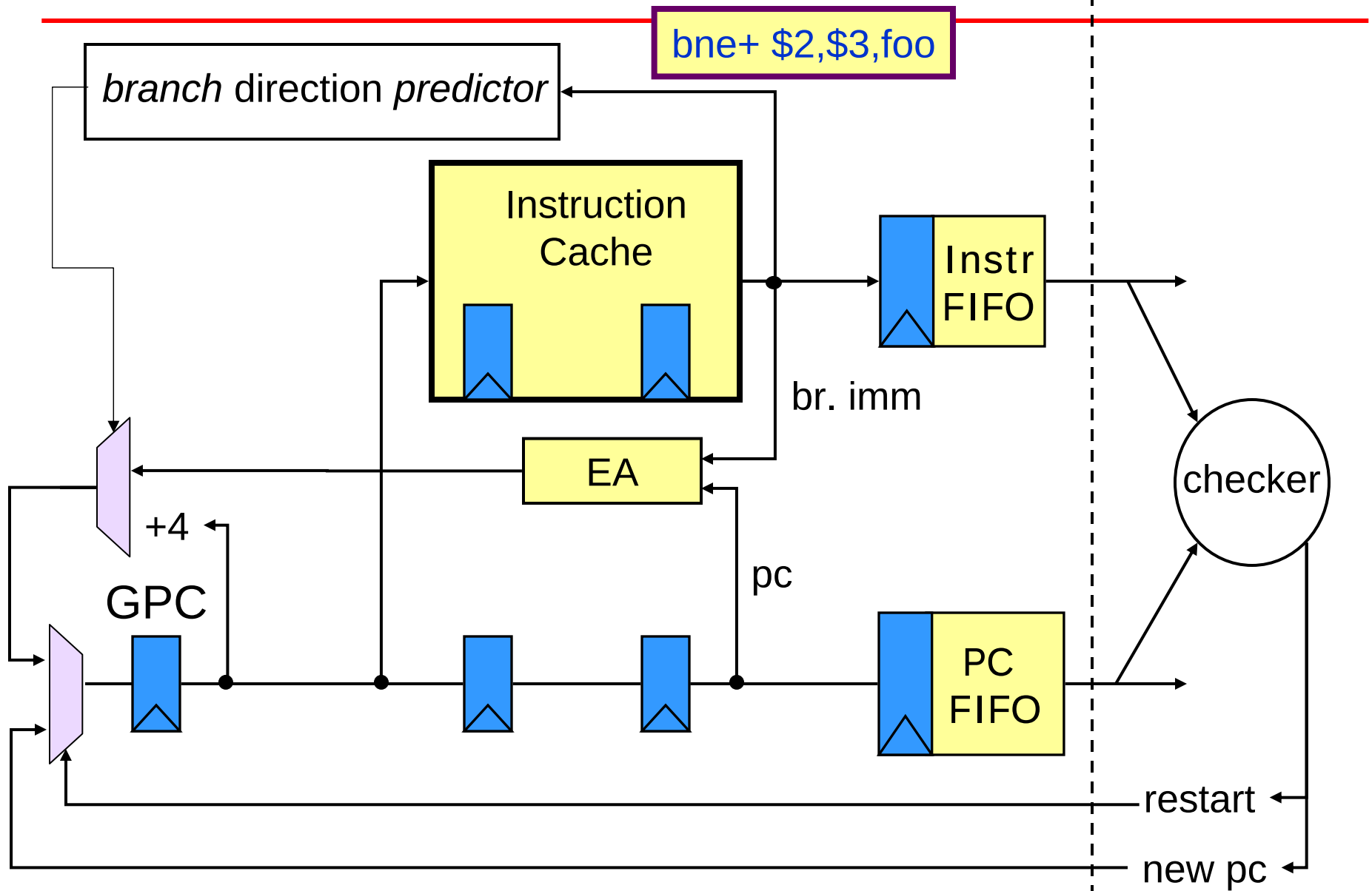
The front end runs ahead .. stalls + cache misses are overlapped.

|    |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|----|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| FE | F | F | F | S | S | S | S | S | S | S | S | F | F | F | S | C | C | C | C | C |
| BE |   | E | E | C | C | C | C | C | C | C | C | E | E | E | S | S | S | S | S | S |

without decoupling .. stalls + cache misses are not overlapped.

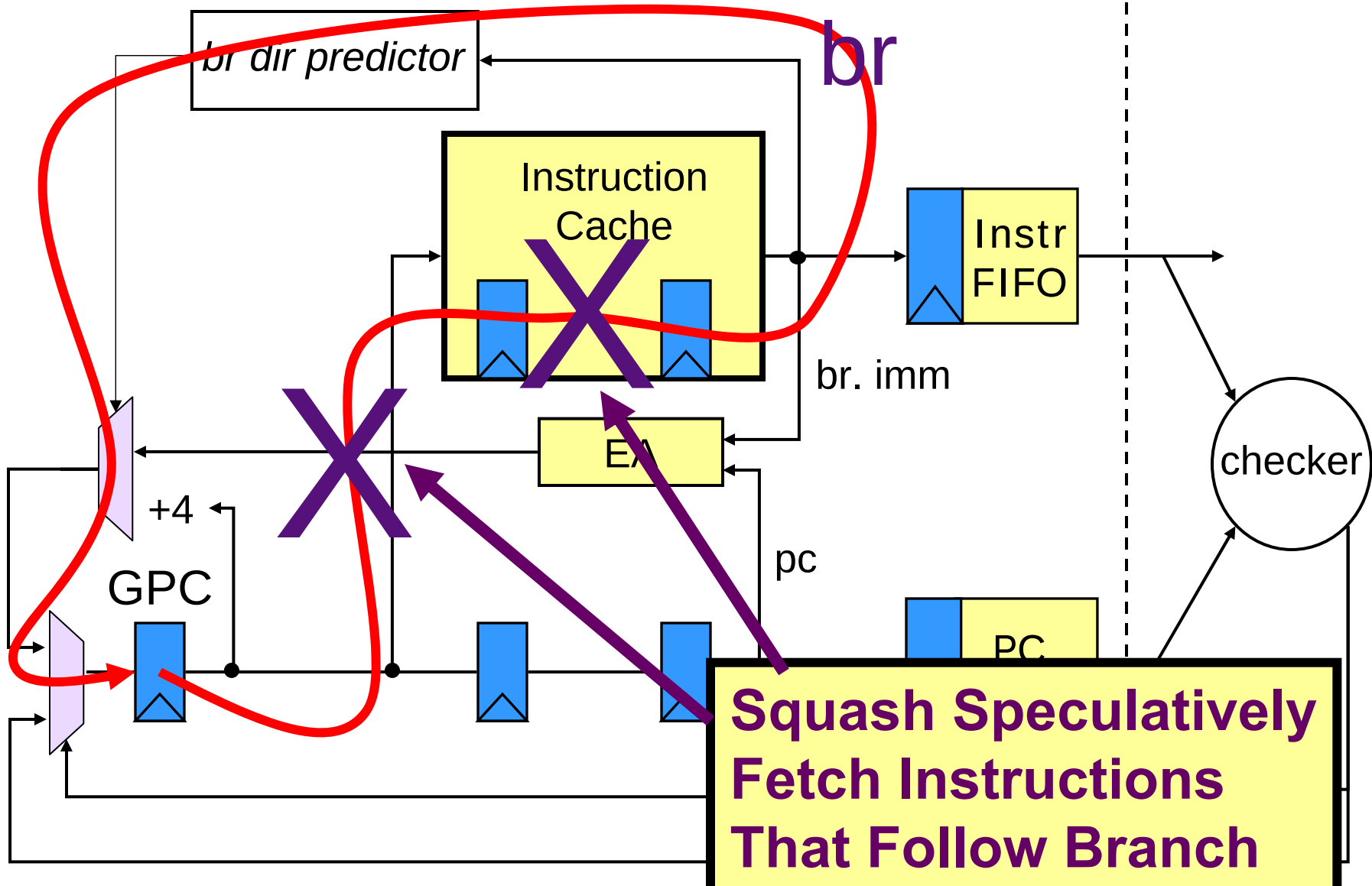
# Pipelined Front End

Back End

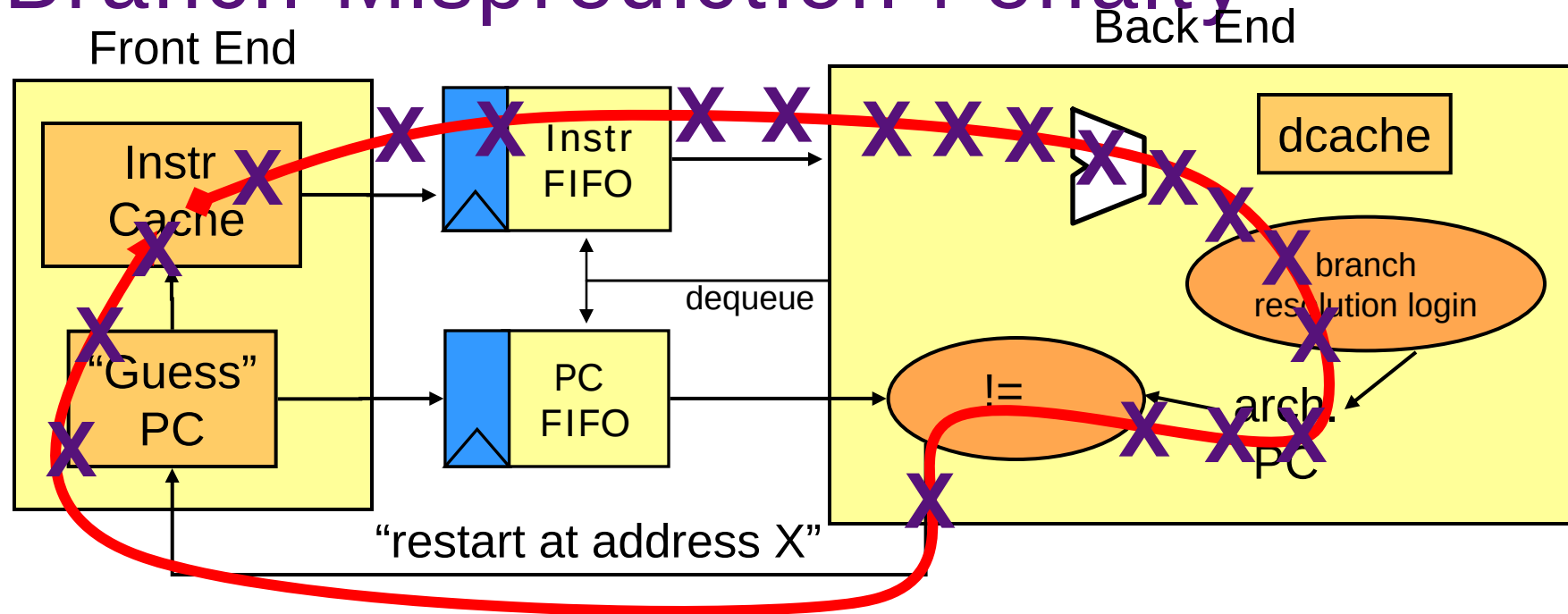


# Branch Predicted-Taken Penalty

## Back End



# Branch Misprediction Penalty



## Basic Pentium III Processor Misprediction Pipeline

| 1     | 2     | 3      | 4      | 5      | 6      | 7      | 8       | 9        | 10   |
|-------|-------|--------|--------|--------|--------|--------|---------|----------|------|
| Fetch | Fetch | Decode | Decode | Decode | Rename | ROB Rd | Rdy/Sch | Dispatch | Exec |

*Pentium 3 – ~10 cycles*

## Basic Pentium 4 Processor Misprediction Pipeline

*Pentium 4 – ~20 cycles*

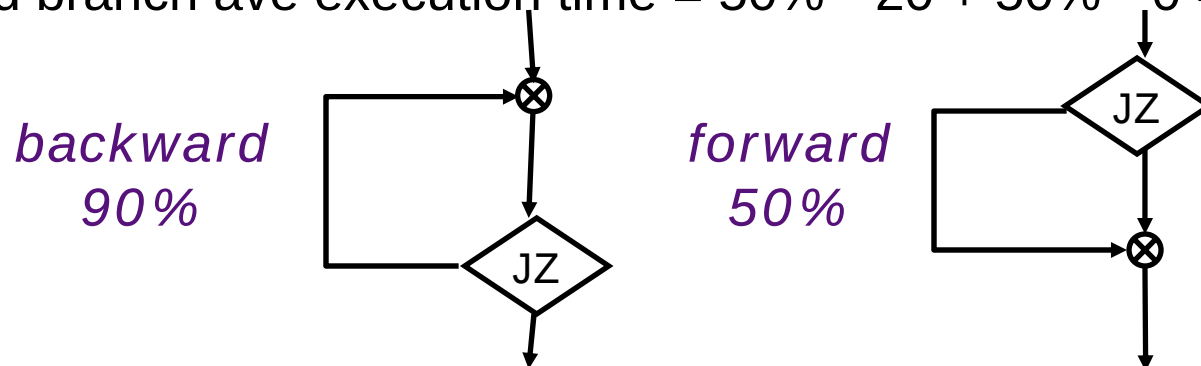
| 1  | 2      | 3  | 4     | 5     | 6     | 7      | 8   | 9   | 10  | 11  | 12   | 13   | 14 | 15 | 16 | 17   | 18    | 19    | 20 |
|----|--------|----|-------|-------|-------|--------|-----|-----|-----|-----|------|------|----|----|----|------|-------|-------|----|
| TC | Nxt IP | TC | Fetch | Drive | Alloc | Rename | Que | Sch | Sch | Sch | Disp | Disp | RF | RF | Ex | Flgs | Br Ck | Drive |    |

# Since misprediction penalty is larger, we first focus on branch (direction) prediction

- Static Strategies:
    - #1 predict taken (34% mispredict rate)
    - #2 predict (backwards taken, forwards not) (10%, 50%) mispredict rate
      - same backwards behavior as #1
      - better forwards behavior (50%-50% branches)
- penalty: #1 taken 2 cycle ~taken 20 cycle  
#2 taken 20 cycle ~taken 0 cycle

#1 forward branch ave execution time =  $50\% * 2 + 50\% * 20 = 11$  cycles

#2 forward branch ave execution time =  $50\% * 20 + 50\% * 0 = 10$  cycles



Since misprediction penalty is larger, we first focus on branch (direction) prediction

- Static Strategies:

- #3 profile (see next slide for misprediction %'s)

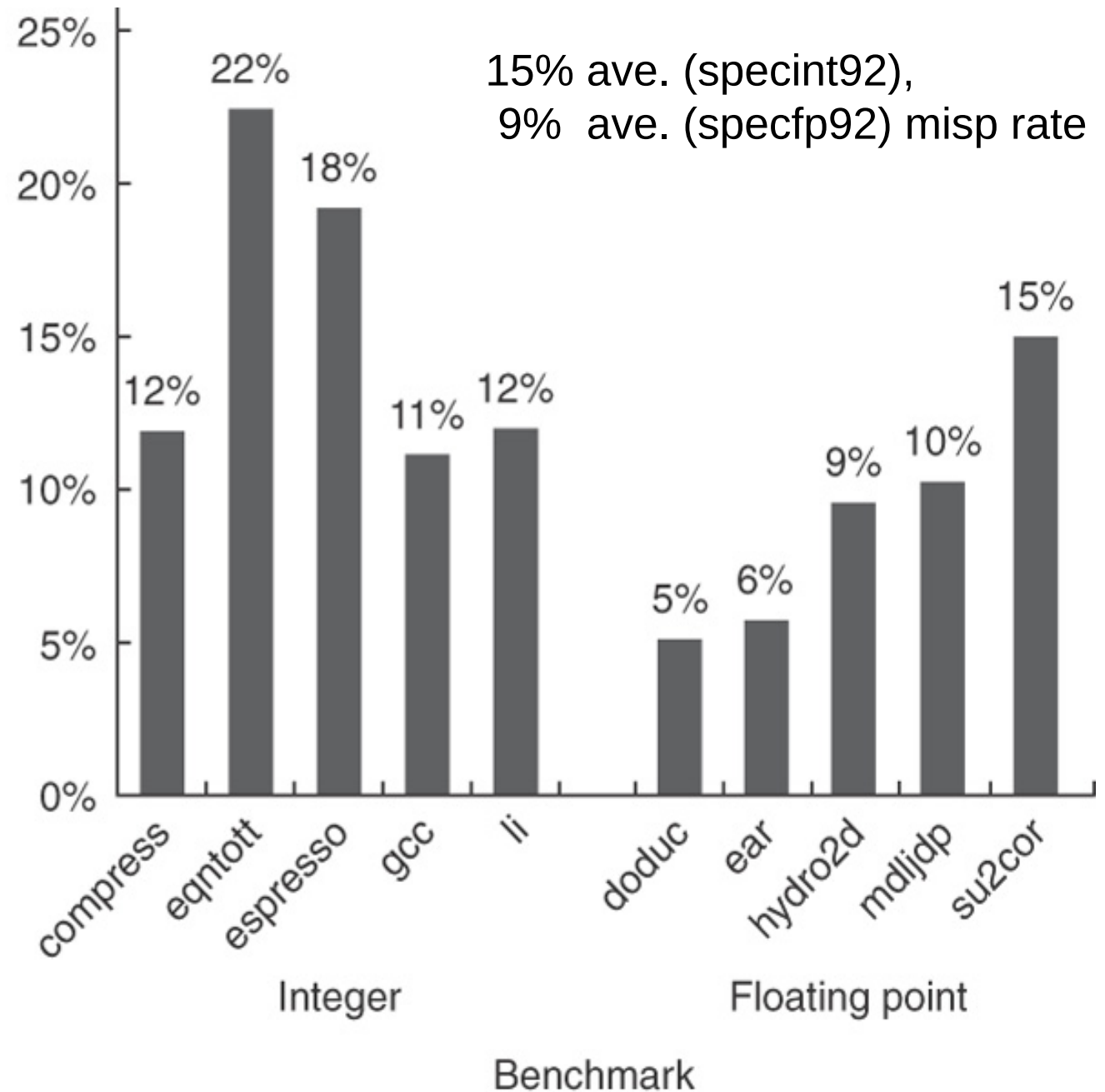
- choose a single prediction for each branch and encode in instruction
    - some studies show that sample runs are fairly representative of inputs in general
    - negative: extra programmer burden

See next slide for misprediction rates

# Profiling Based Static Prediction

*Each branch is permanently assigned a probable direction.*

*To do better we would need to change the prediction as the program runs!*



## A note on prediction/misprediction rates

Qualitatively, ratio of misprediction rates  
is better indicator of predictor improvement.

15% ave. (specint92),  
9% ave. (specfp92) misp rate

*(assumes misprediction probability independent between branches)*

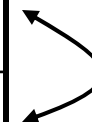
| Prediction Rate (p) | Misprediction Rate | # Consecutive Branches Predicted Correctly (w/ 50% prob) |
|---------------------|--------------------|----------------------------------------------------------|
| 50%                 | 50%                | 1                                                        |
| 78%                 | 22%                | 2.78                                                     |
| 85%                 | 15%                | 4.26                                                     |
| 91%                 | 9%                 | 7.34                                                     |
| 95%                 | 5%                 | 13.5                                                     |
| 96%                 | 4%                 | 16.98                                                    |
| 98%                 | 2%                 | 34.3                                                     |

Bernoulli Process:

$$p^k = .5$$

$$k = \lg(.5)/\lg(p)$$

2% makes a huge  
difference here

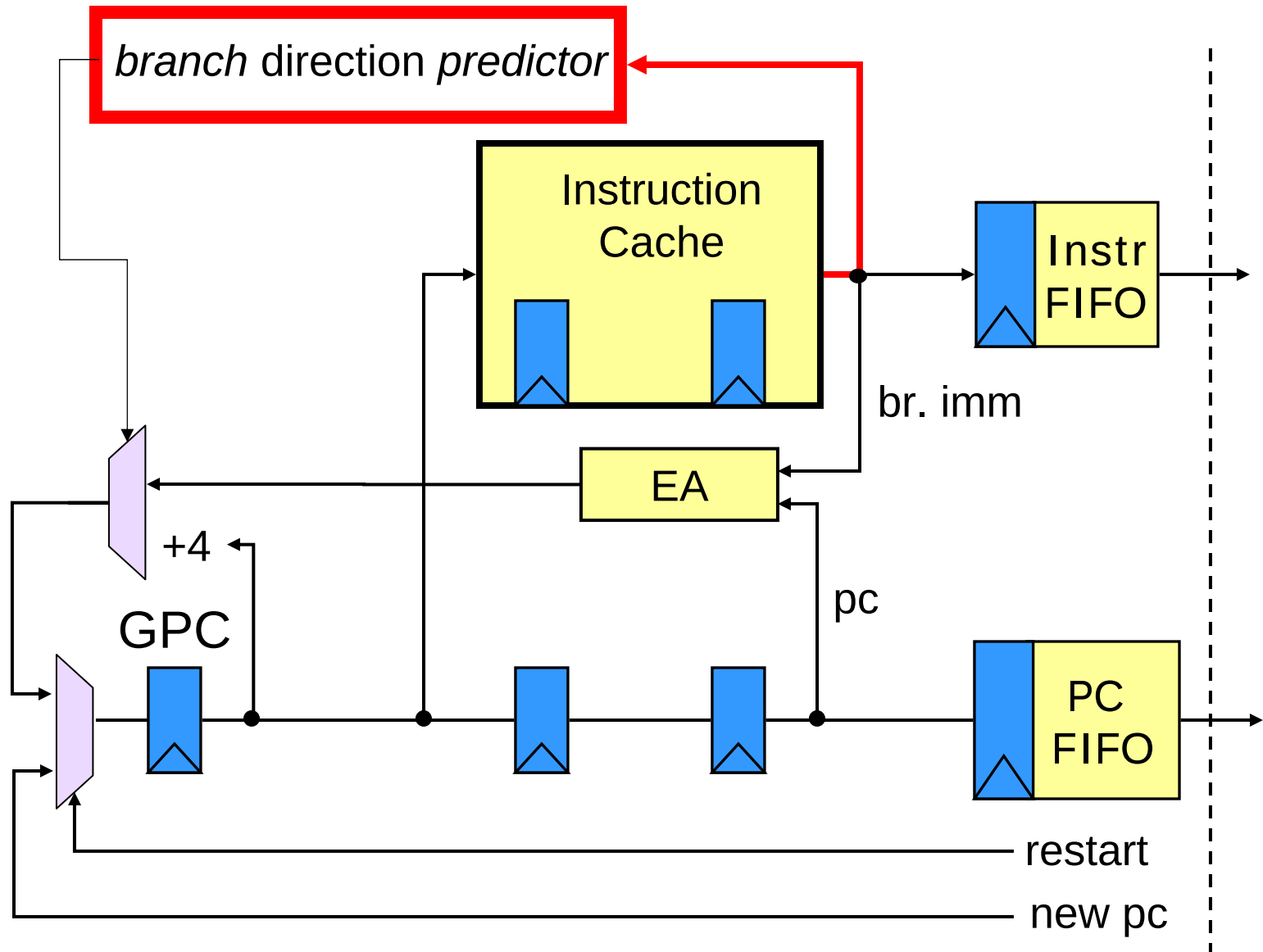


# Compiler also can take advantage of Static Prediction / Profiling / Knowledge

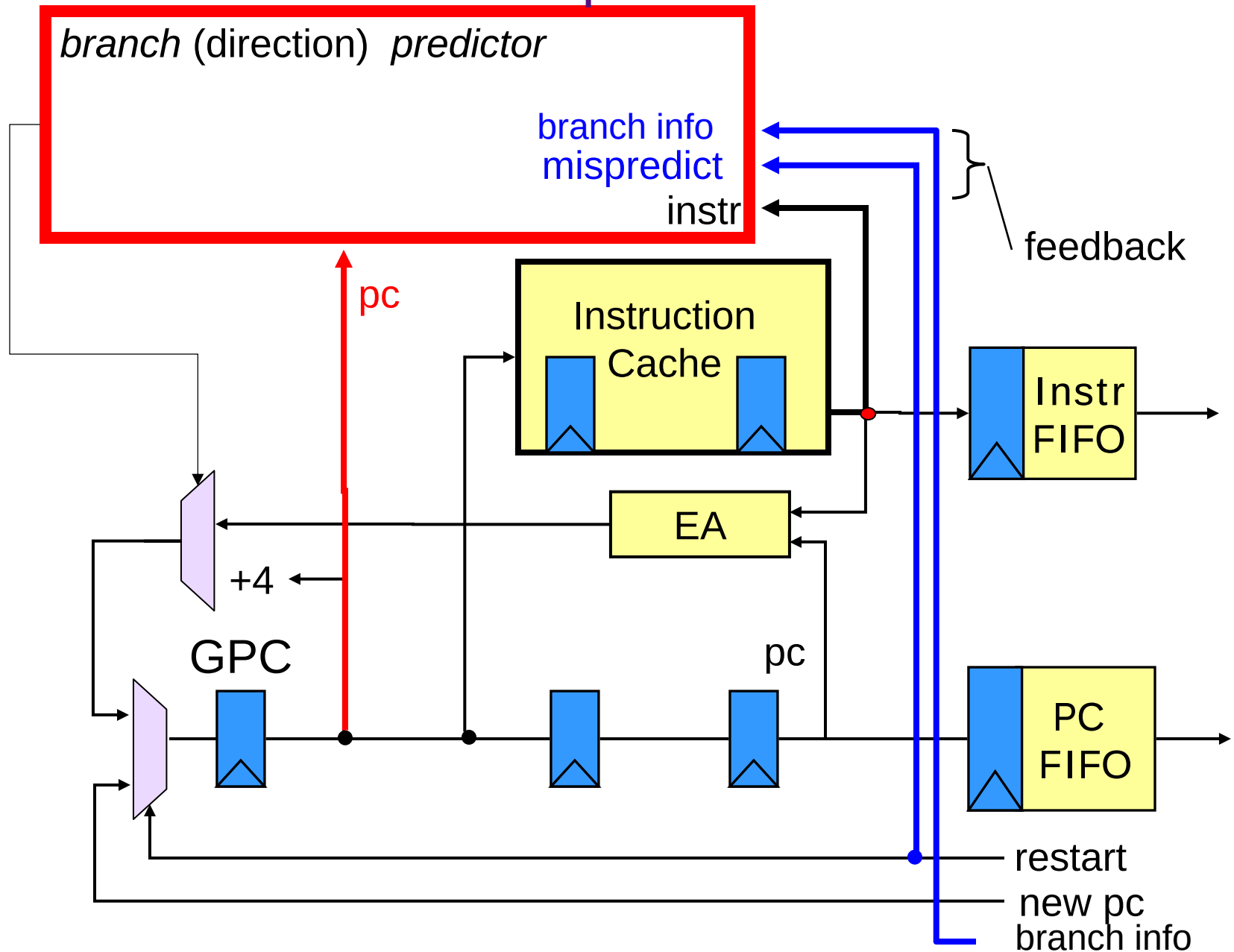
---

- Static Strategies:
  - #1 predict taken (34% mispredict rate)
  - #2 predict backwards taken, forwards not (10%, 50% mispredict rate)
  - #3 profile (see previous slide)
  - #4 delayed branches
    - always execute instructions after branches
    - avoids need to flush pipeline after branch

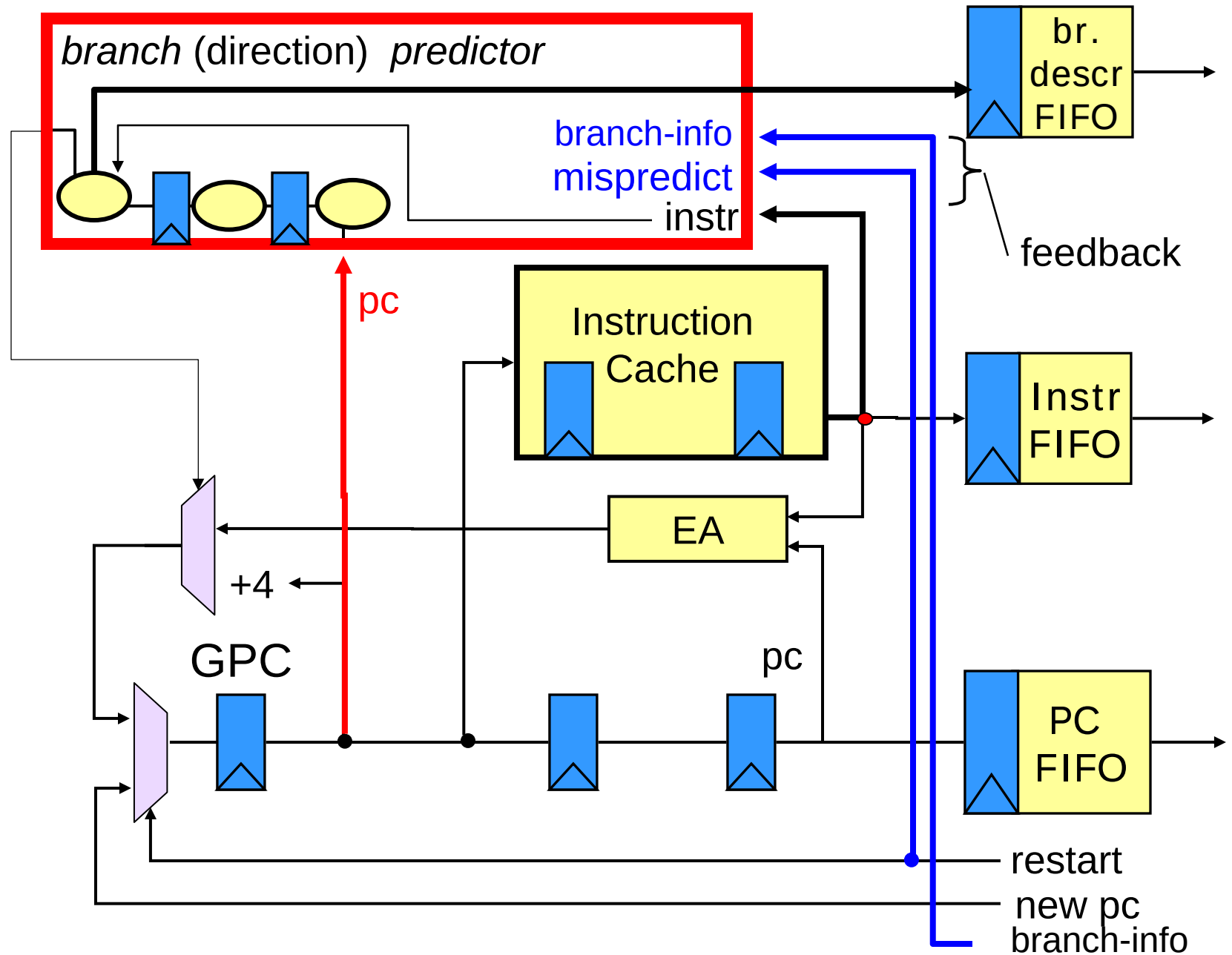
Observation: Static Prediction is limited because it only uses instructions as input + has a fixed prediction



Dynamic Prediction: More inputs allow it to adjust the brand direction prediction over time

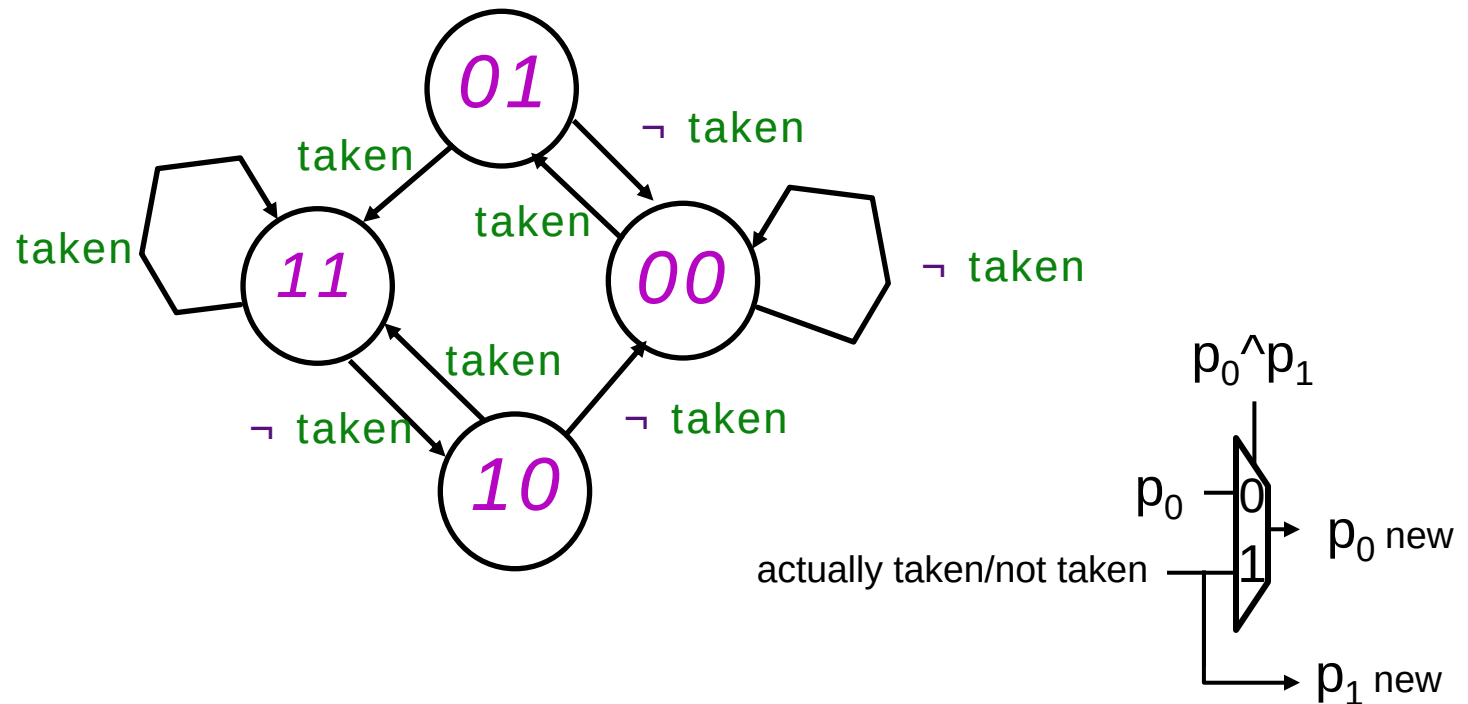


# Dynamic Prediction: More detailed



# Dynamic Branch Prediction – Track Changing Per-Branch Behavior

- Store 2 bits per branch
- Change the prediction after two consecutive mistakes!



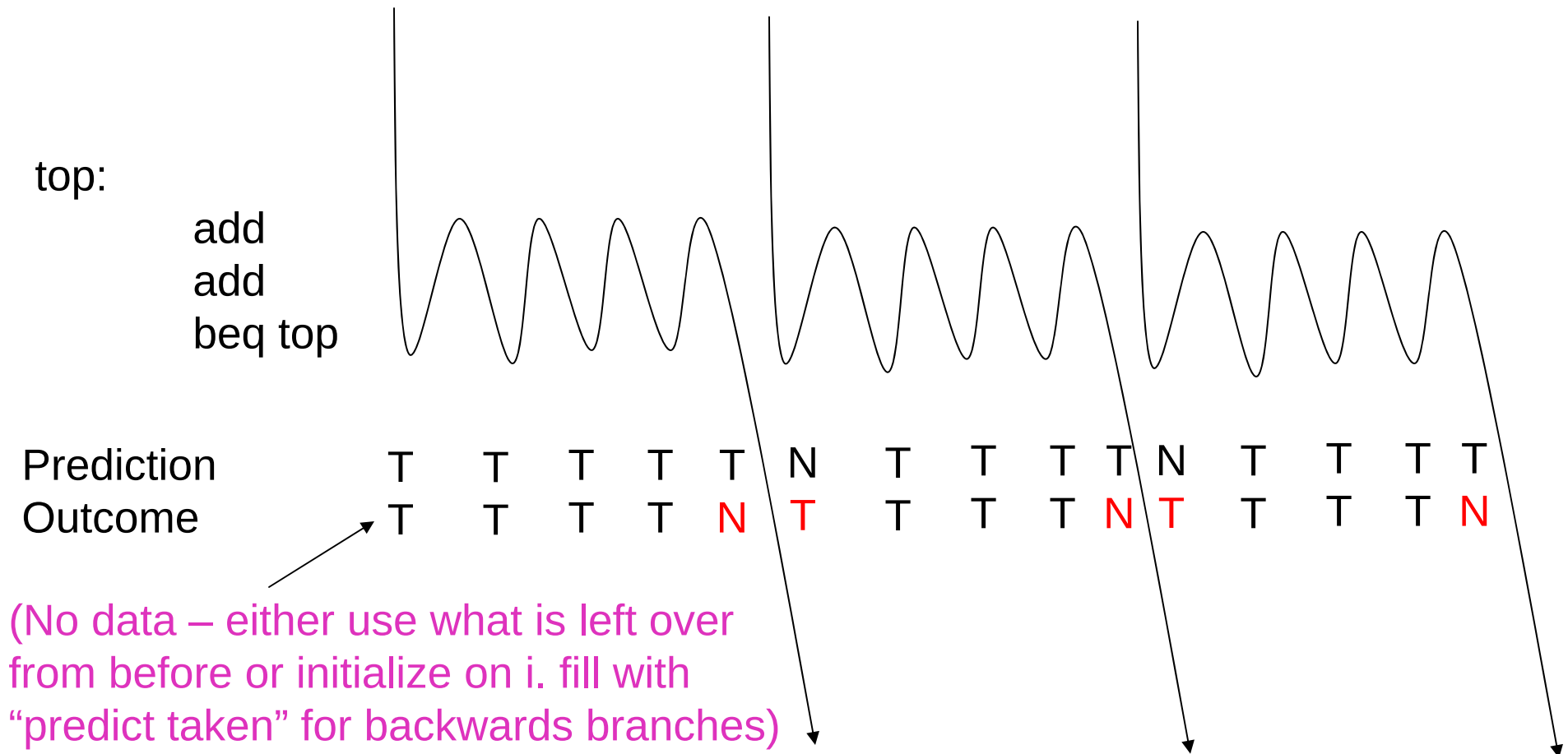
(next prediction **taken/¬taken**) x (last branch **taken/¬taken**)

**note: this is not strictly a saturating up-down counter**

# Why two bits?

- One bit is wishy-washy on loops

Two mispredictions per loop execution with single-bit prediction



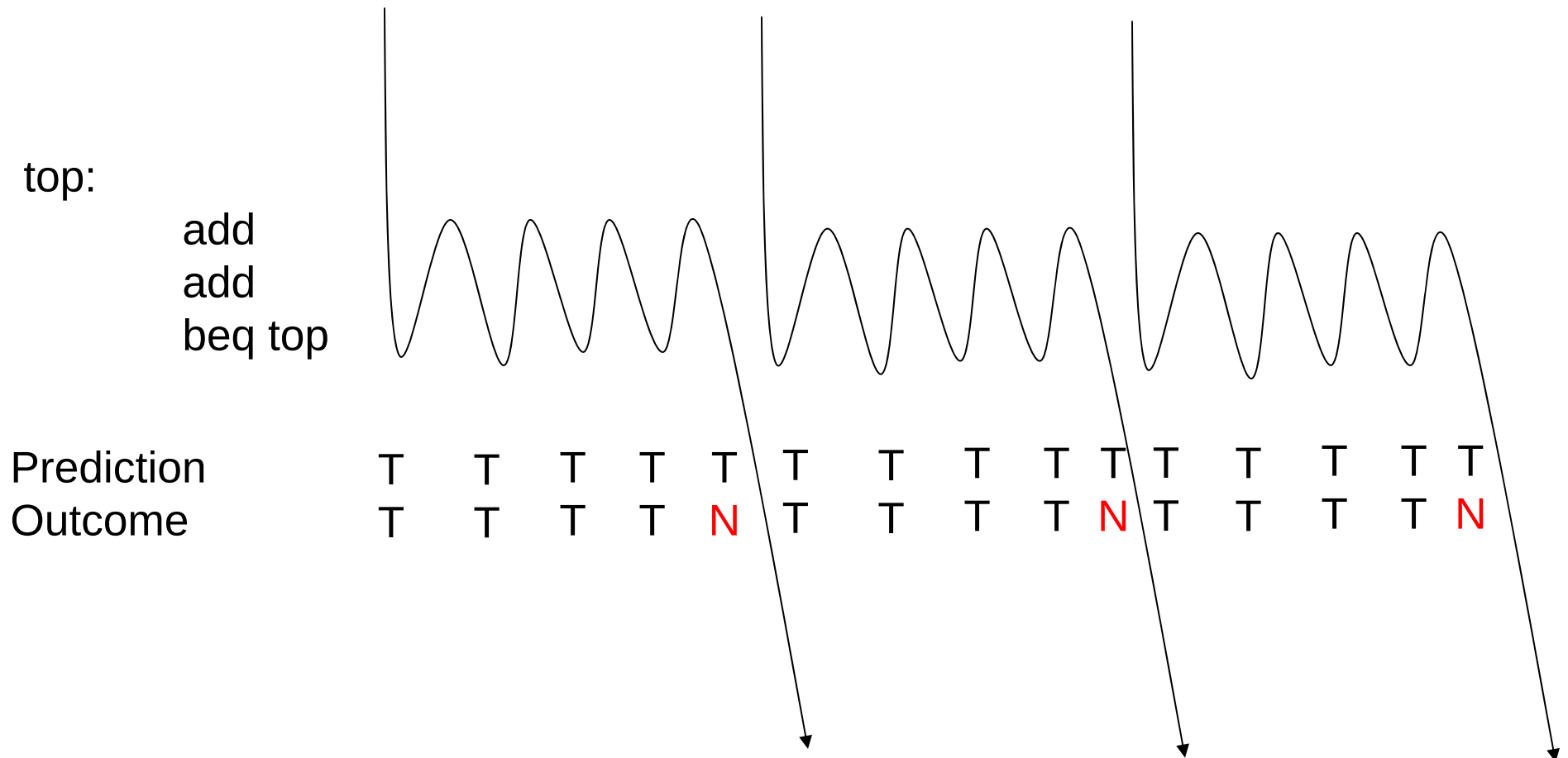
Single Bit Predictor Analysis

# Why two bits?

- One bit is wishy-washy on loops

---

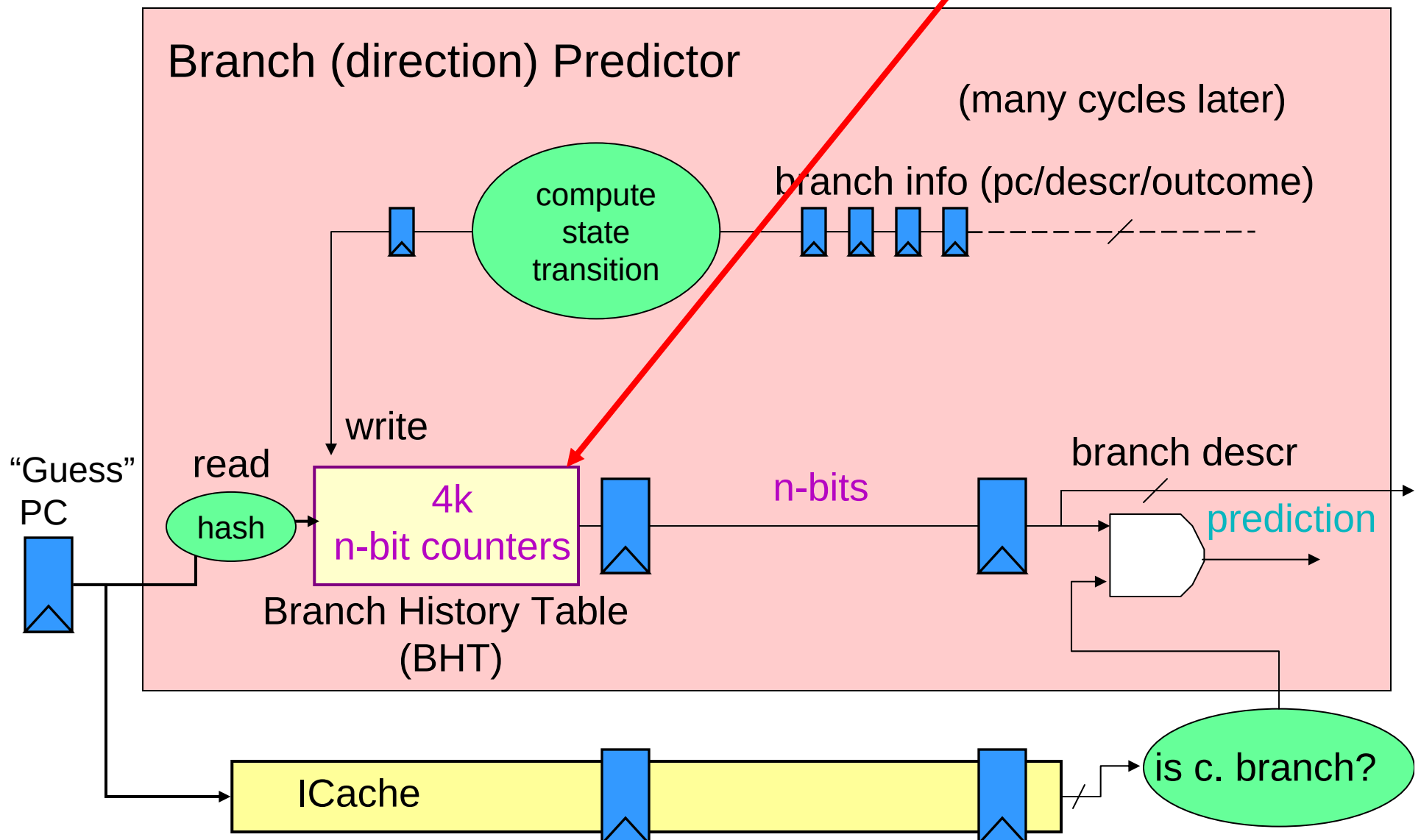
One misprediction per loop execution with two-bit prediction



Two Bit Predictor Analysis

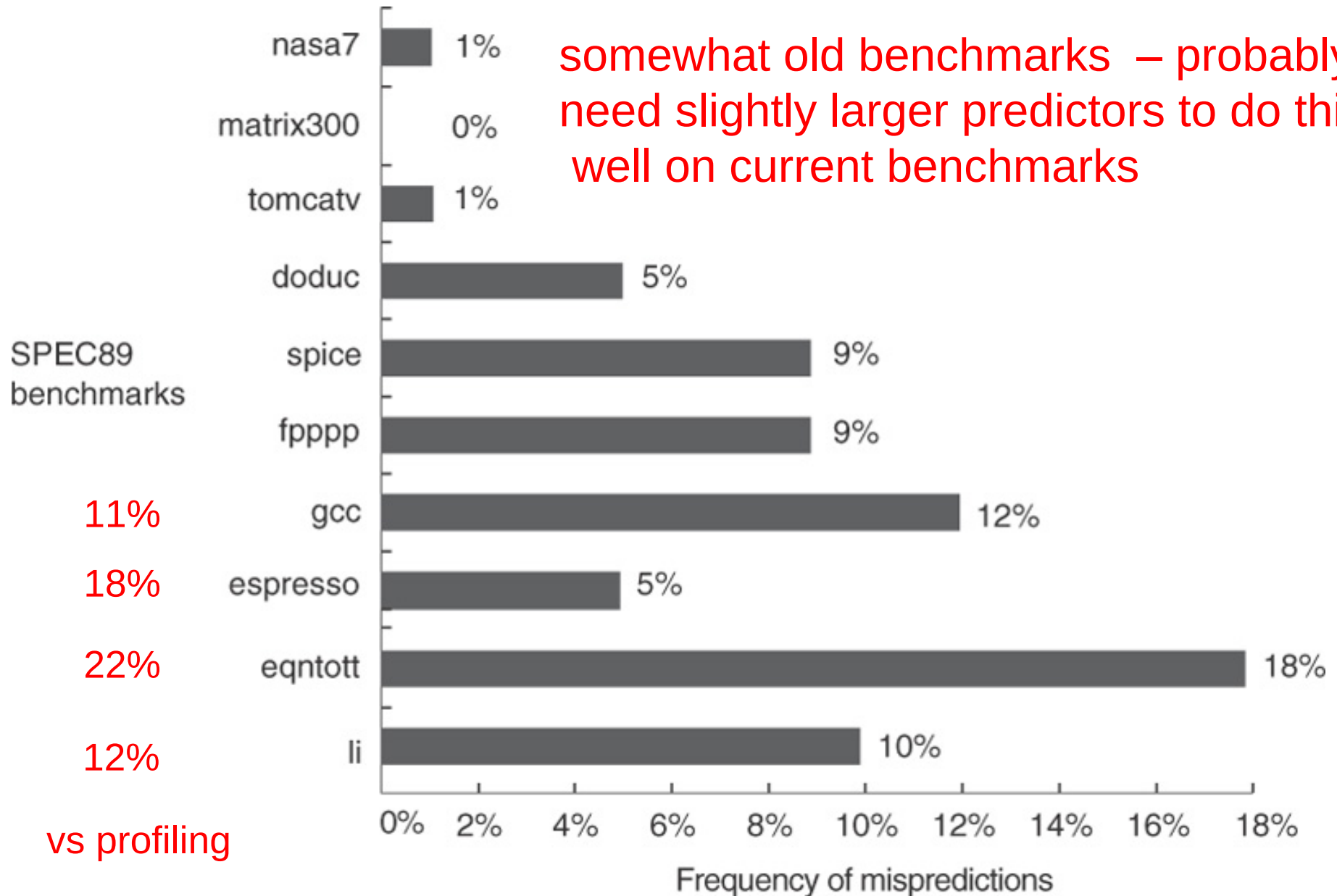
# n-bit implementation

blindly write into  
this hash table; branches  
may alias but that's "ok"



## Accuracy of simple dynamic branch predictor: 4096-entry 2-bit predictor on Spec89

somewhat old benchmarks – probably  
need slightly larger predictors to do this  
well on current benchmarks

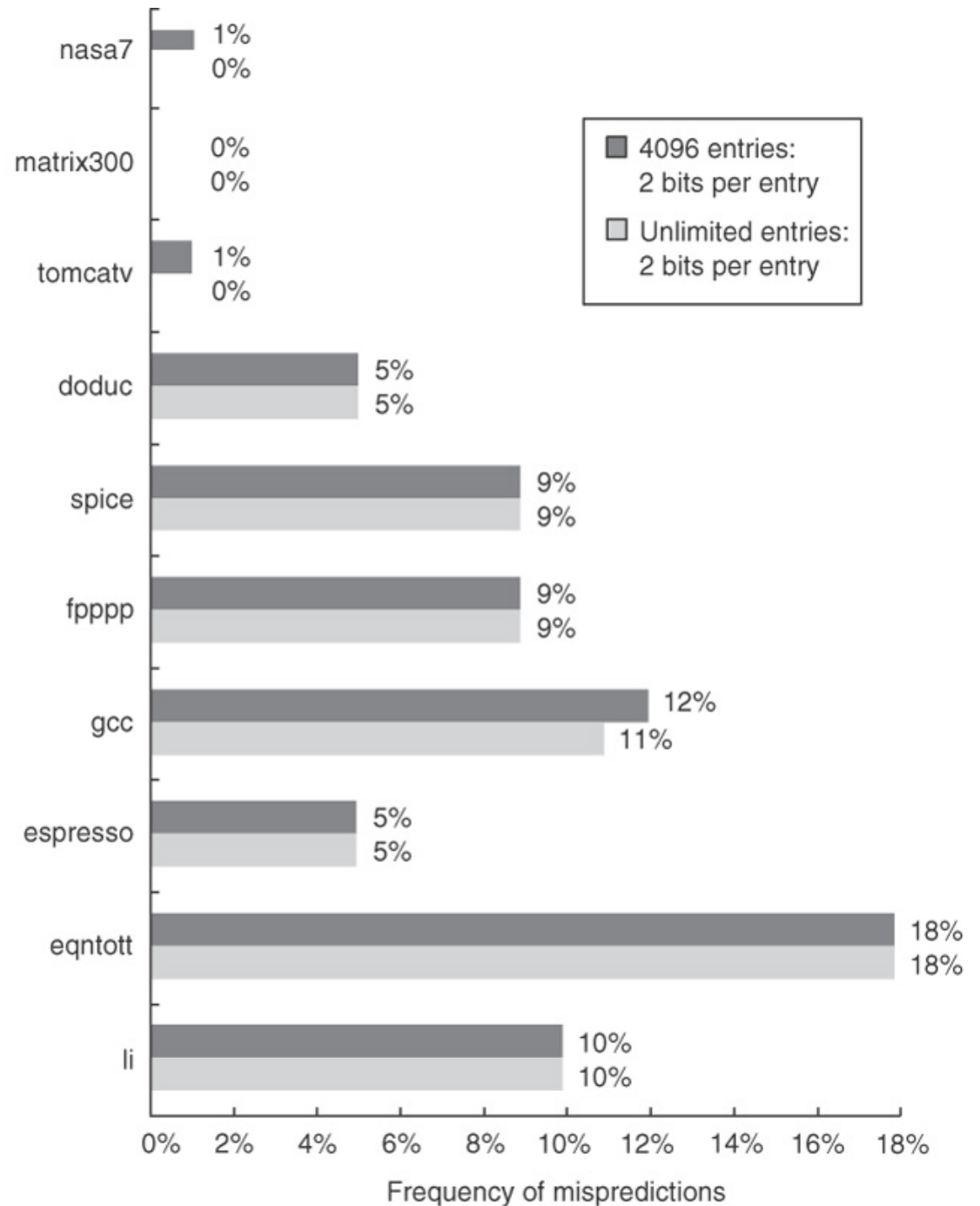


# Limits of 2-bit prediction

- $\infty$  table does not help much on spec89

SPEC89  
benchmarks

- reportedly, more bits does not help significantly either.



# 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

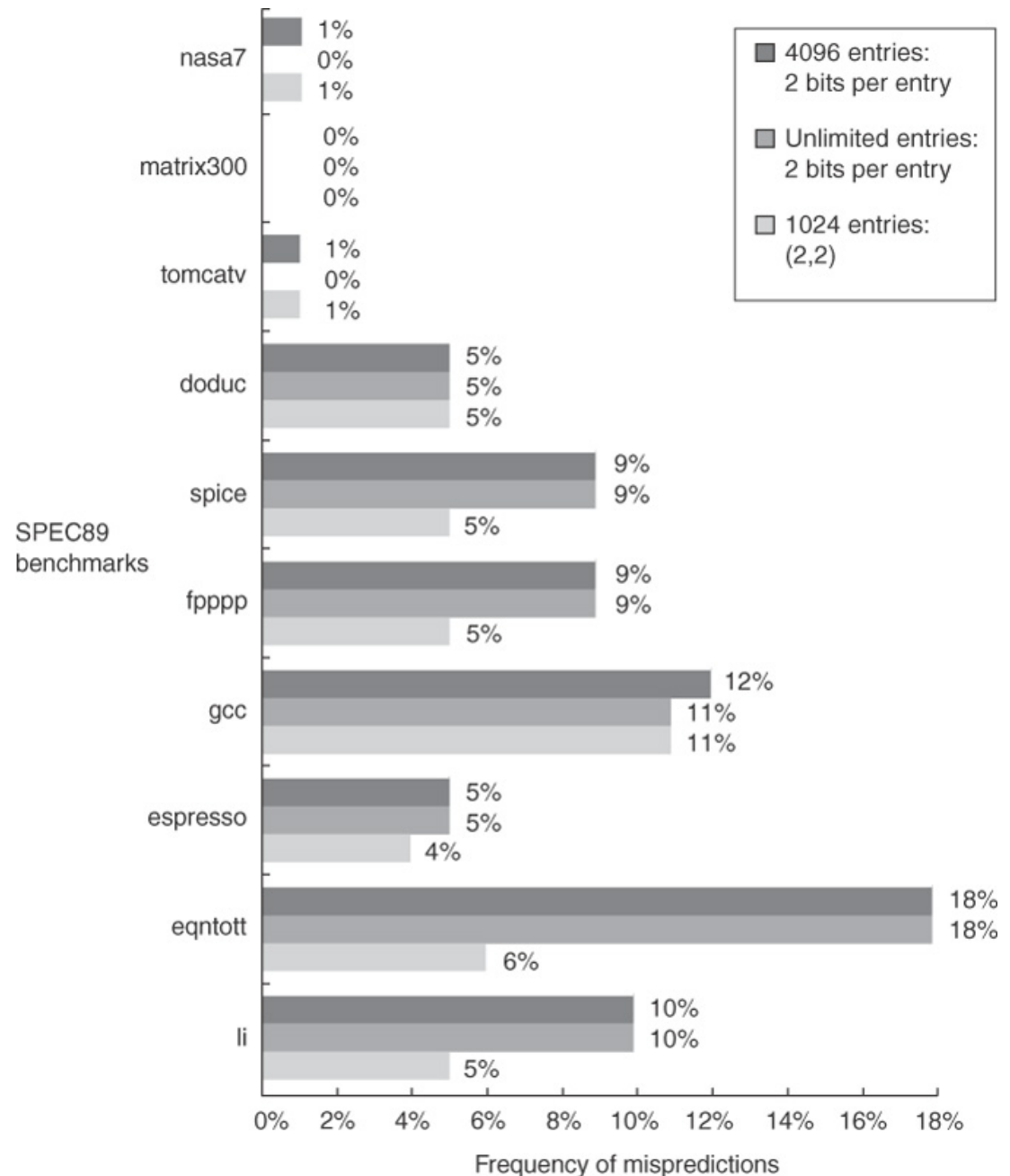
*History bit:* H records the direction of the last branch executed by the processor

Two sets of BHT bits (BHT0 & BHT1) per branch instruction

|                   |   |              |
|-------------------|---|--------------|
| H = 0 (not taken) | ⇒ | consult BHT0 |
| H = 1 (taken)     | ⇒ | consult BHT1 |

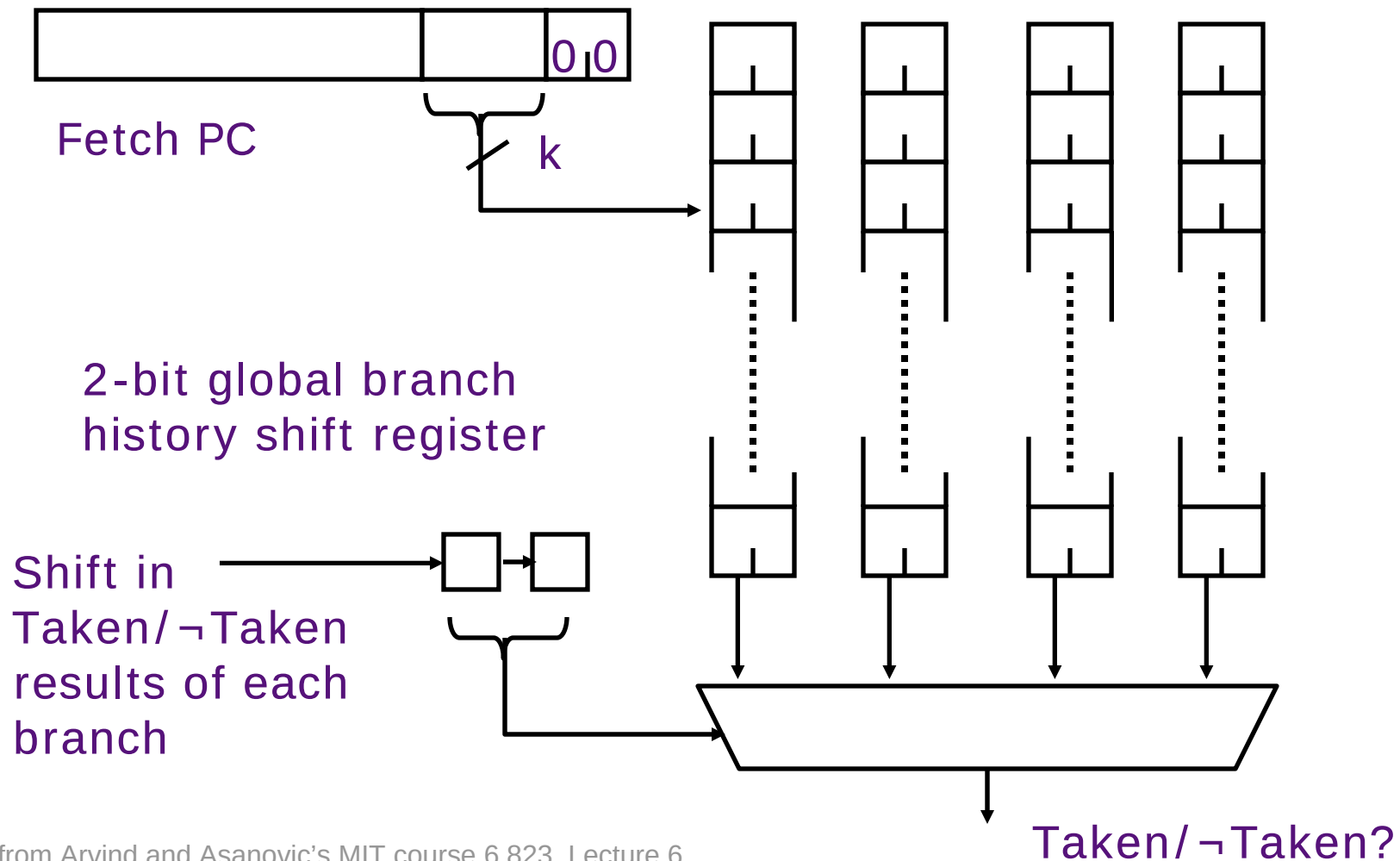
# Accuracy with 2 bits of global history

Less storage than  
4k x 2bit  
but better accuracy  
(for these benchmarks)



# Two-Level Branch Predictor

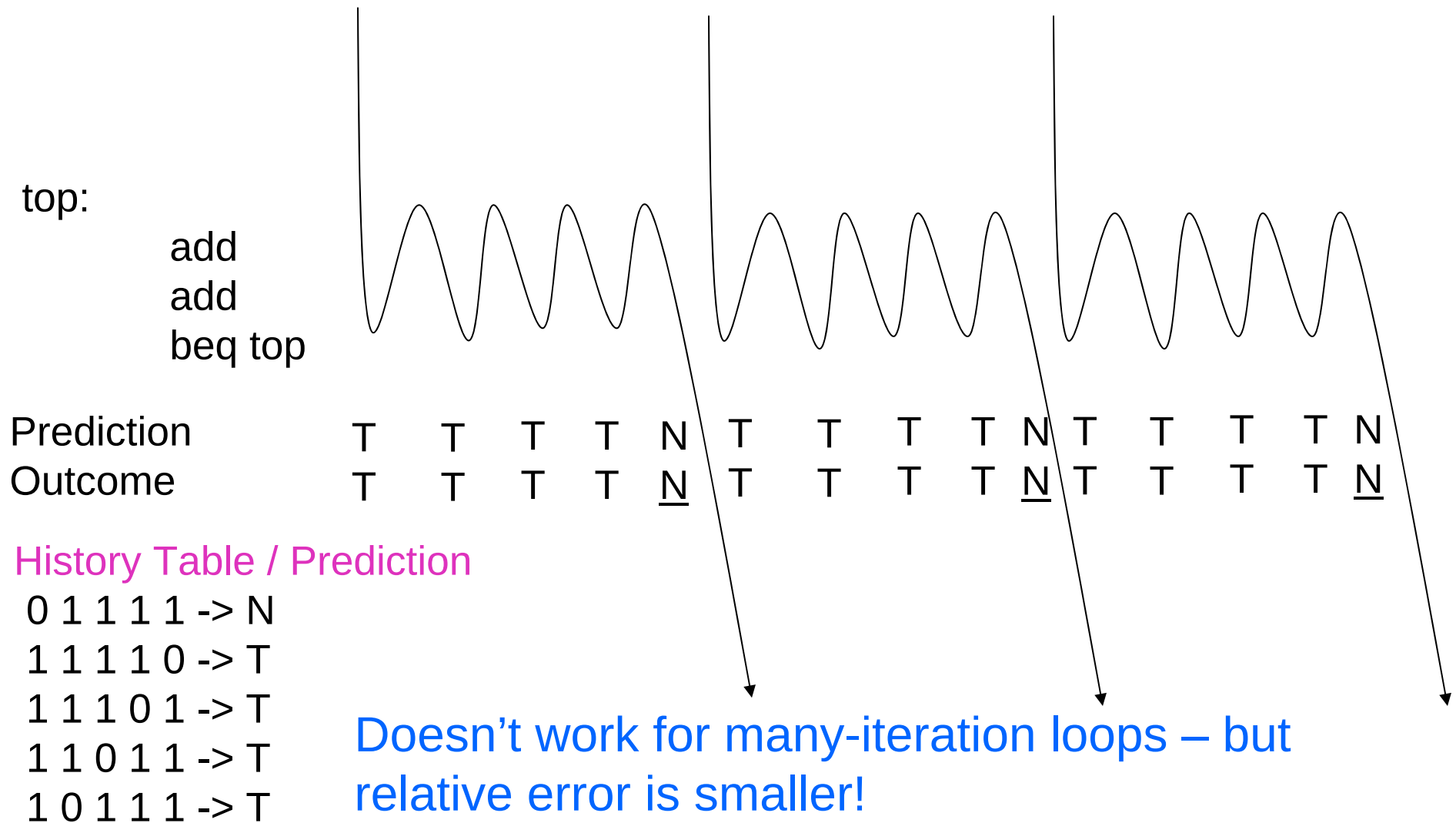
*Pentium Pro (1995) uses the result from the last two branches to select one of the four sets of BHT bits (~95% correct)*



# Benefit of longer histories for fixed-iteration loops with small iteration counts

- Unary encoding of branch patterns

No mispredictions per (5-iter) loop execution with  $\geq 5$ -bits of history



# Alpha 21264 Tournament Predictor

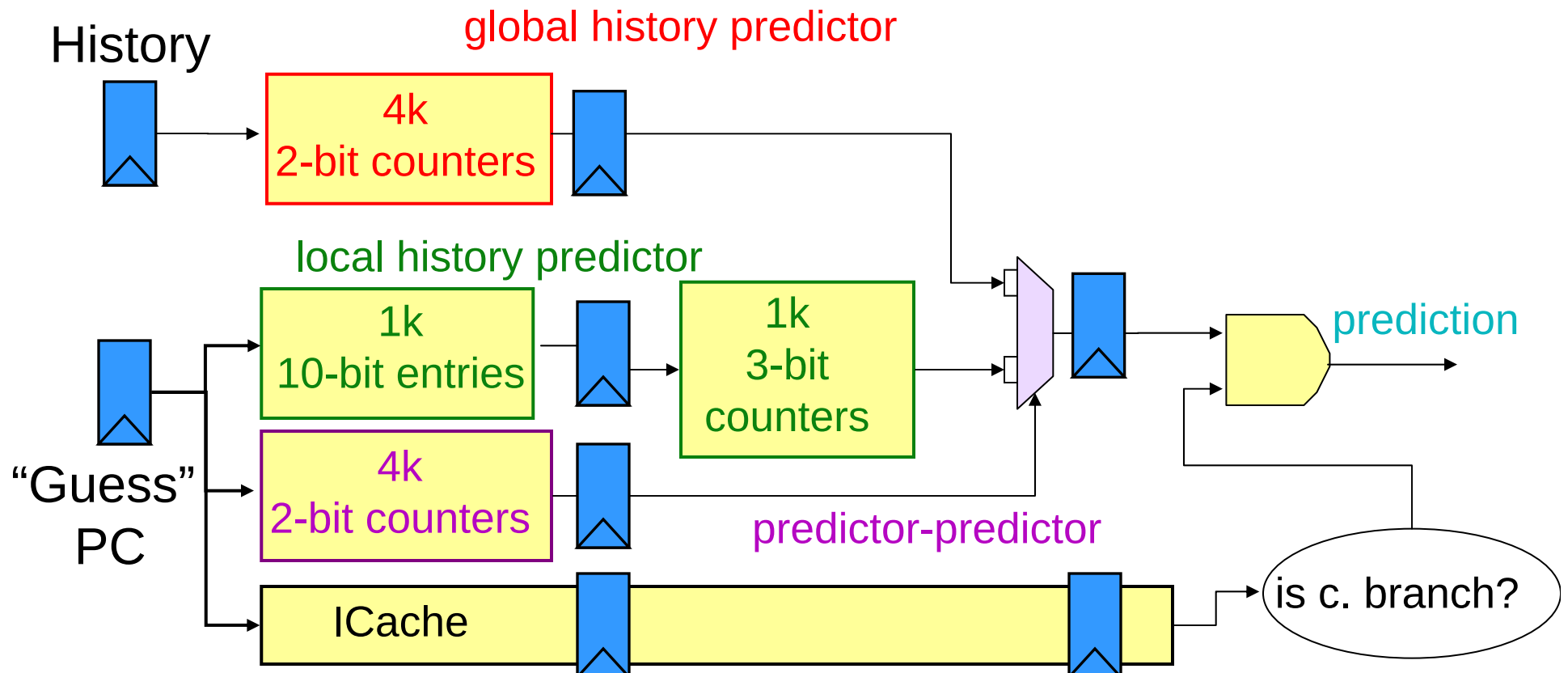
“Predictor-Predictor”: 4K 2-bit counters indexed by branch address  
- chooses between two predictors:

A. **global predictor**: 4K 2-bit counters indexed by 12-bit global history

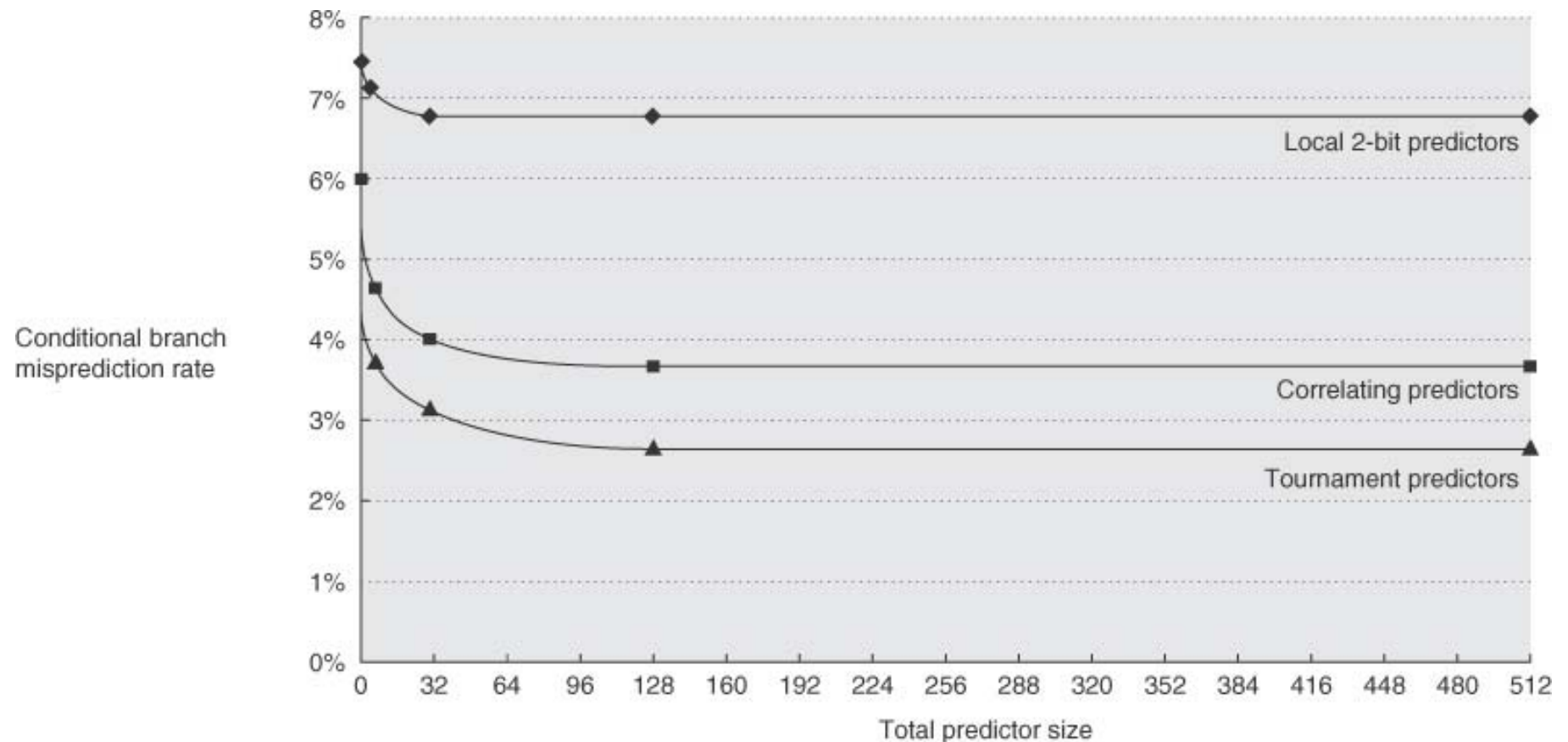
B. **local predictor**

1024 10-bit entries containing history for that entry

This history indexes into 1K 3-bit saturating counters



# Tournament, Correlating, Local Predictor Performance



© 2007 Elsevier, Inc. All rights reserved.

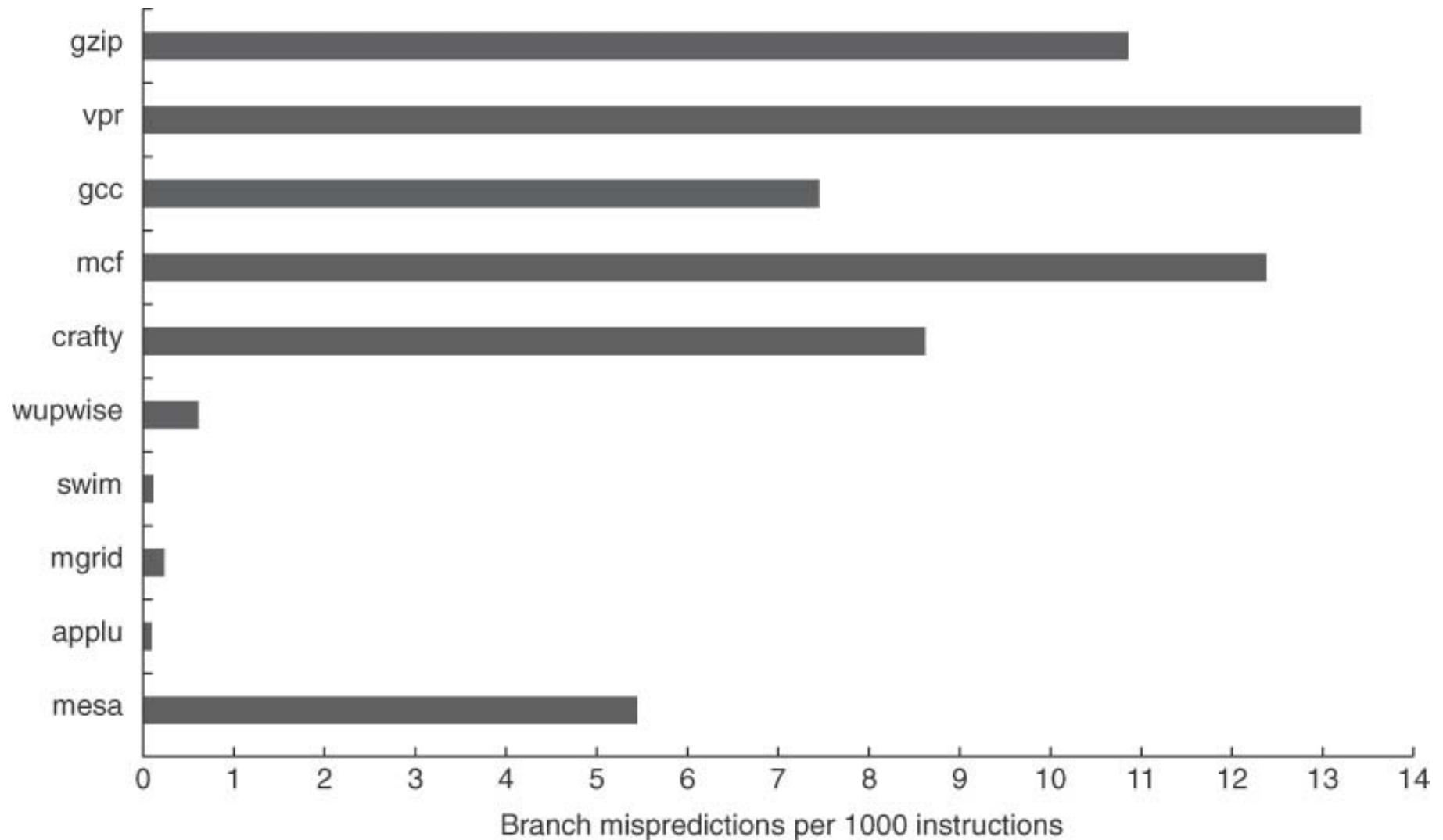
Consecutively Correctly  
Predicted Branches  
w/ 50% probability:

2-bit ~10  
correlating ~18  
tournament ~25

Spec89 (size presumably in Kbit)

Pentium 4 3.2 GHz

Spec2000 Misprediction Rates



© 2007 Elsevier, Inc. All rights reserved.

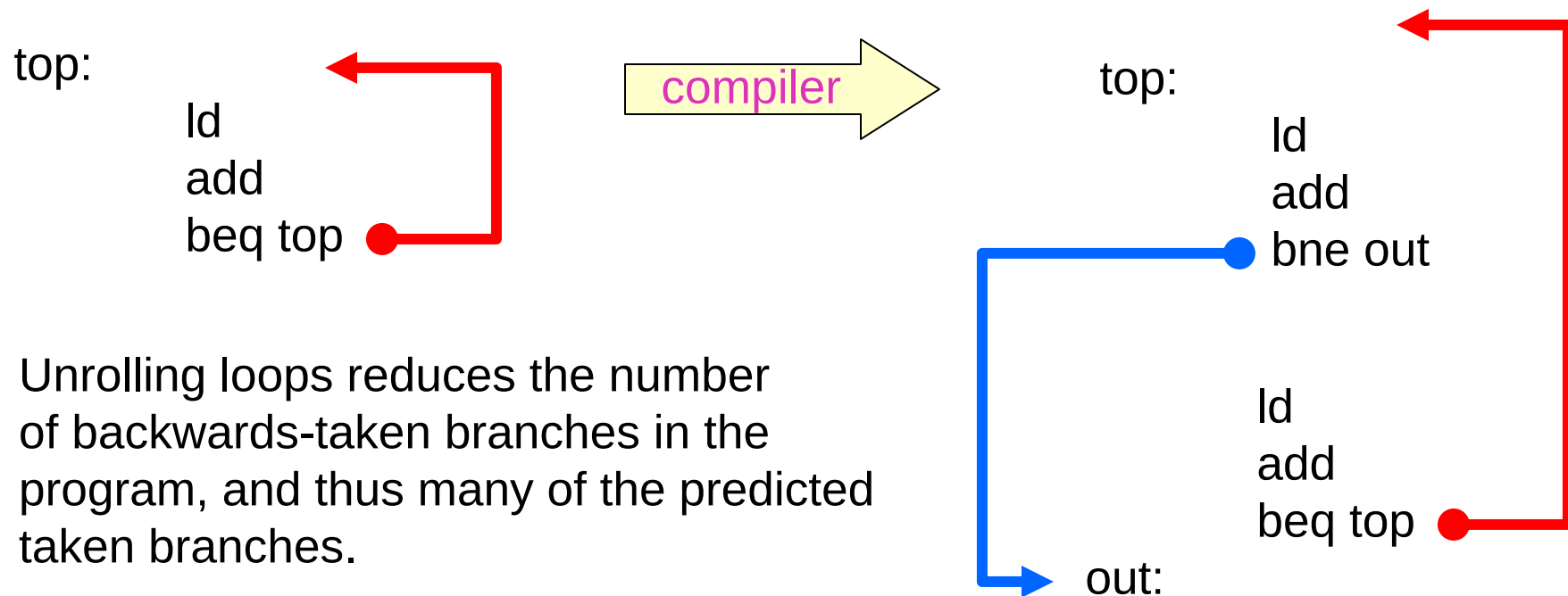
note: the metric is slightly different here, but P4 has some of the best branch prediction because it needs it – extremely long pipeline



# Top N List of Ways to Avoid Branch-Taken Penalties

---

## 1. Unroll thy loops



Unrolling loops reduces the number of backwards-taken branches in the program, and thus many of the predicted taken branches.

Matters most when loop bodies are small.

*Positive/Negatives?*

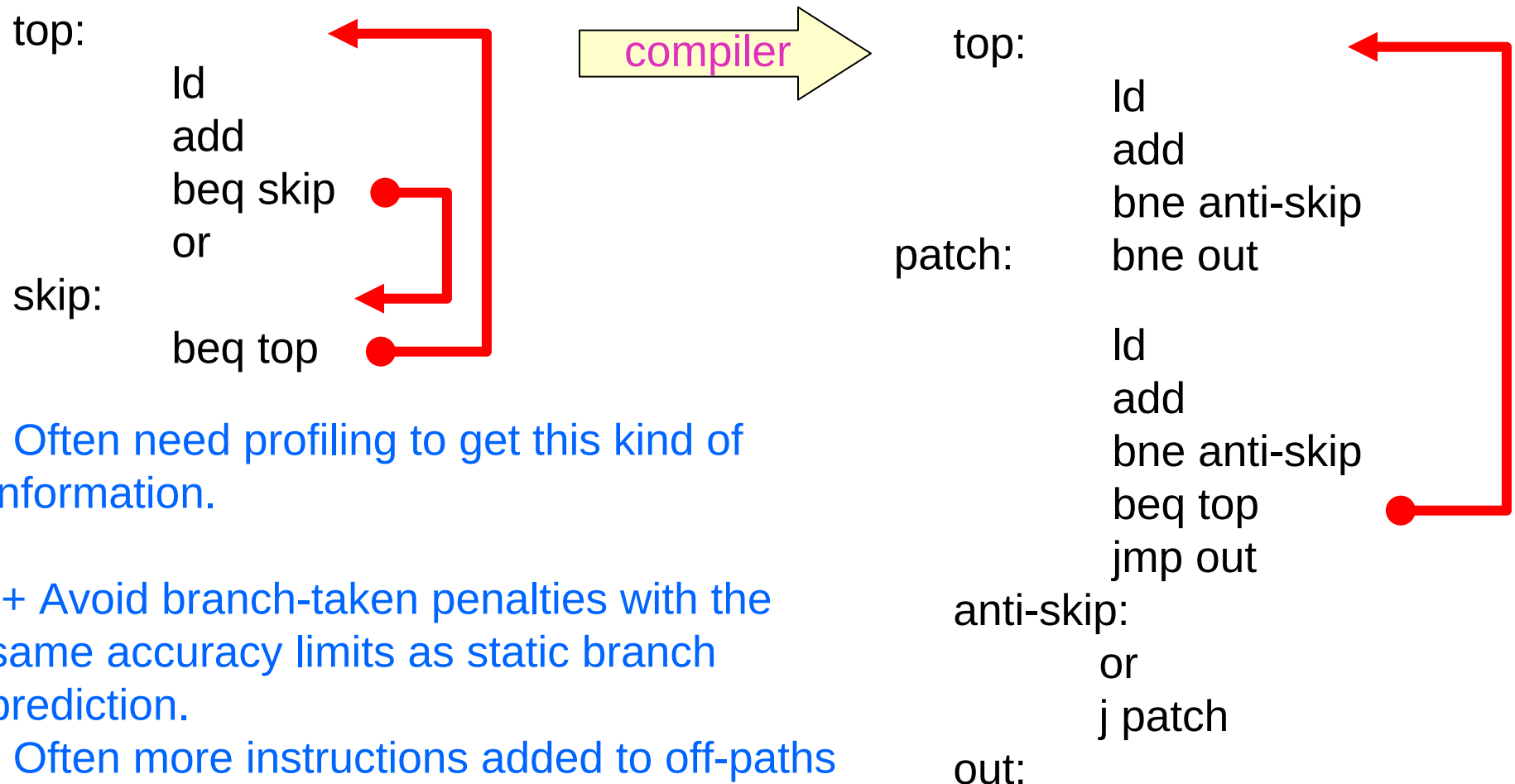
red arcs = common case in this example

# Top N List of Ways to Avoid Branch-Taken Penalties

*Positive/Negatives?*

## 2. Unroll+

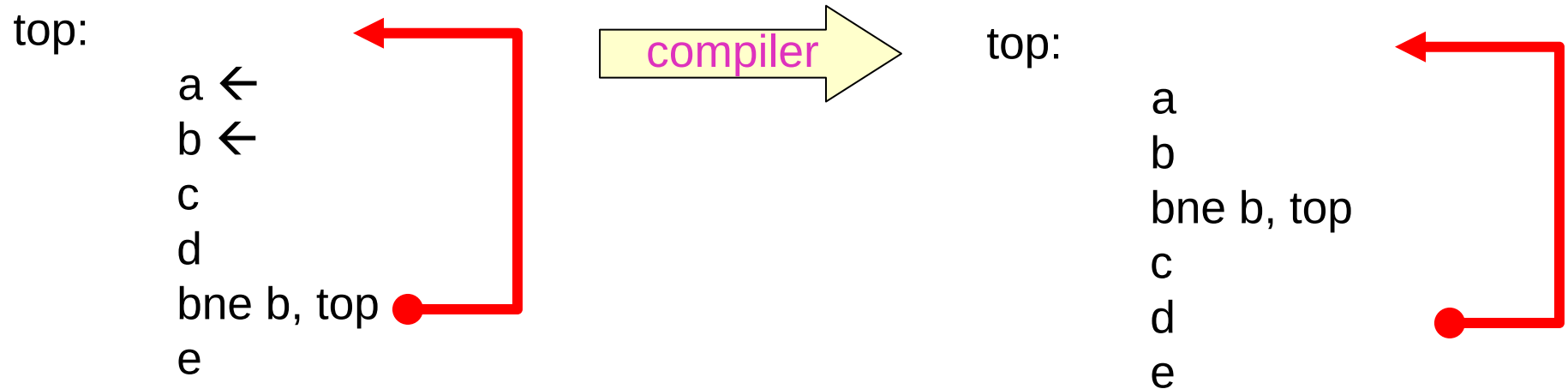
Reorder code into common paths and off-paths.



# Top N List of Ways to Avoid Branch-Taken Penalties

---

## 3. Delayed Branches

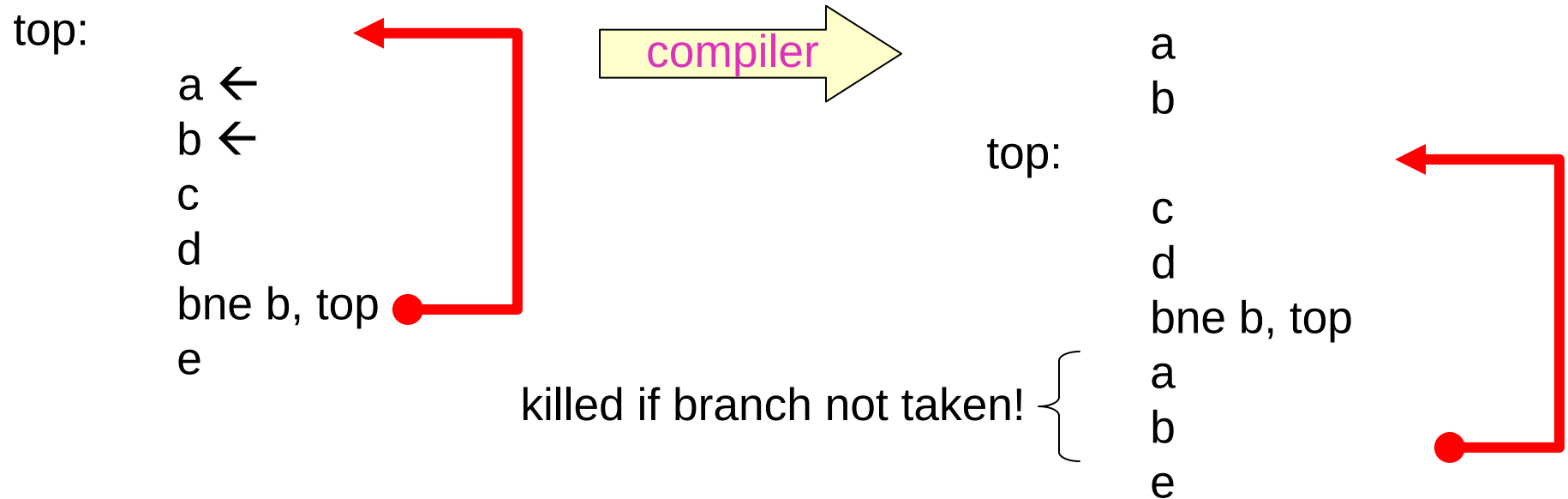


- Requires extra work that is independent of the branch that can be scheduled often not available.
- Architecturally fixed number of delay slots.
- Messy semantics – branches within branch delay slots? Exceptions?

*Positive/Negatives?*

# Top N List of Ways to Avoid Branch-Taken Penalties

## 4. Anulled Branches



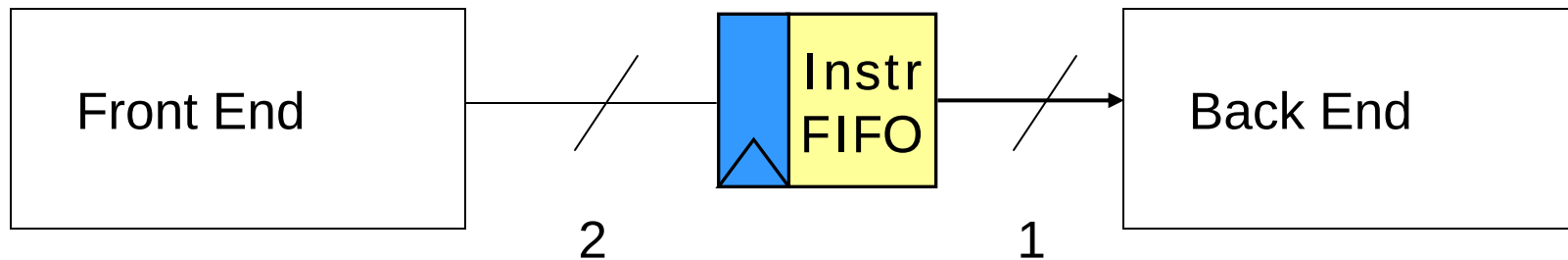
+ Filler instructions are automatically independent of branch because they come from the next iteration of the loop. It is easier to fill these than standard delayed branches.

- Architecturally fixed number of delay slots.
- Messy semantics – branches within branch delay slots? Exceptions?

# Top N List of Ways to Avoid Branch-Taken Penalties

---

## 5. Fetch Ahead (So as Not to Fall Behind)



+ Fetch unit can fetch more instructions per cycle than the backend can consume, filling the FIFO more quickly. Then, the front end can afford to spend a few cycles on each taken branch.

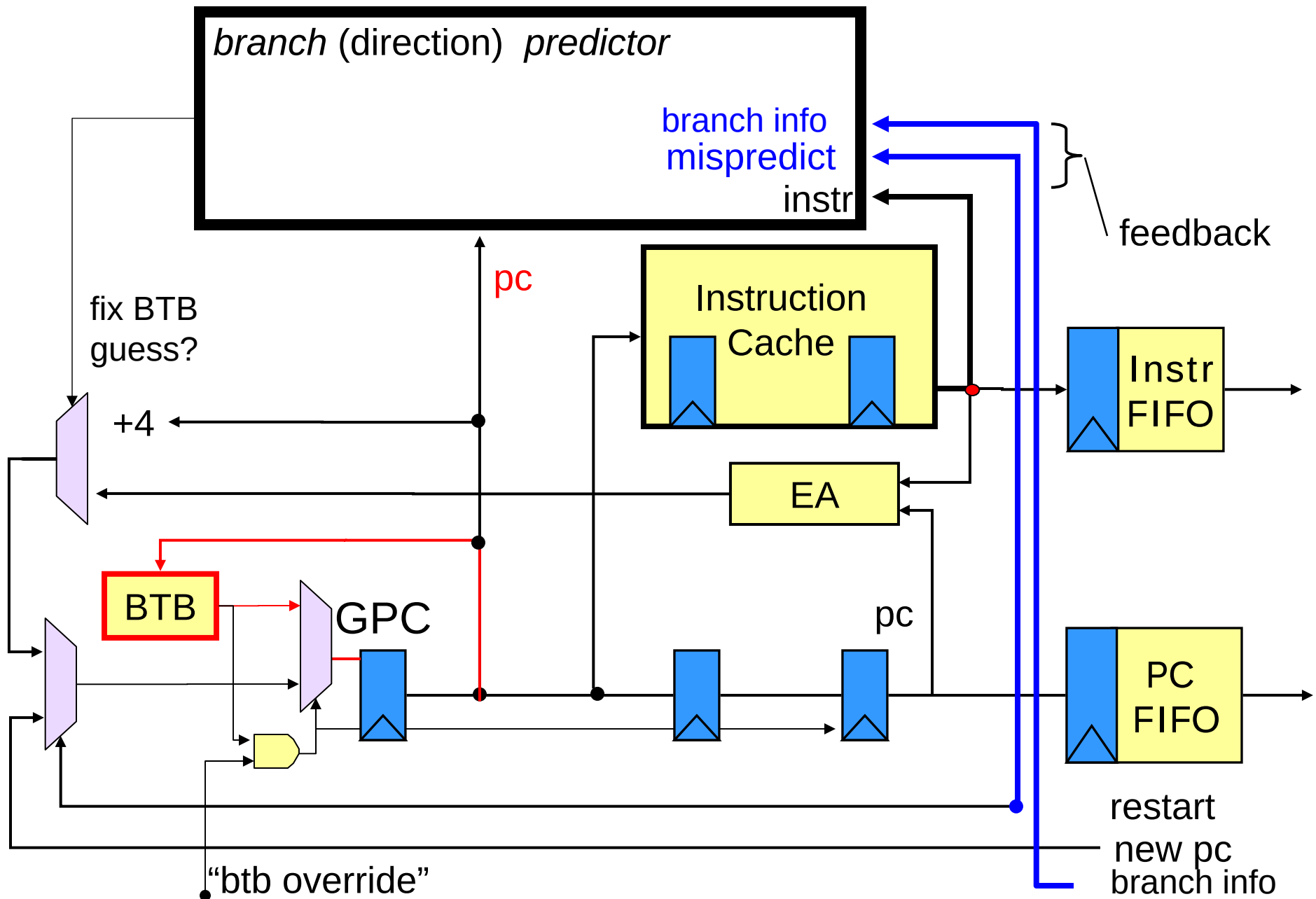
# Top N List of Ways to Avoid Branch-Taken Penalties

---

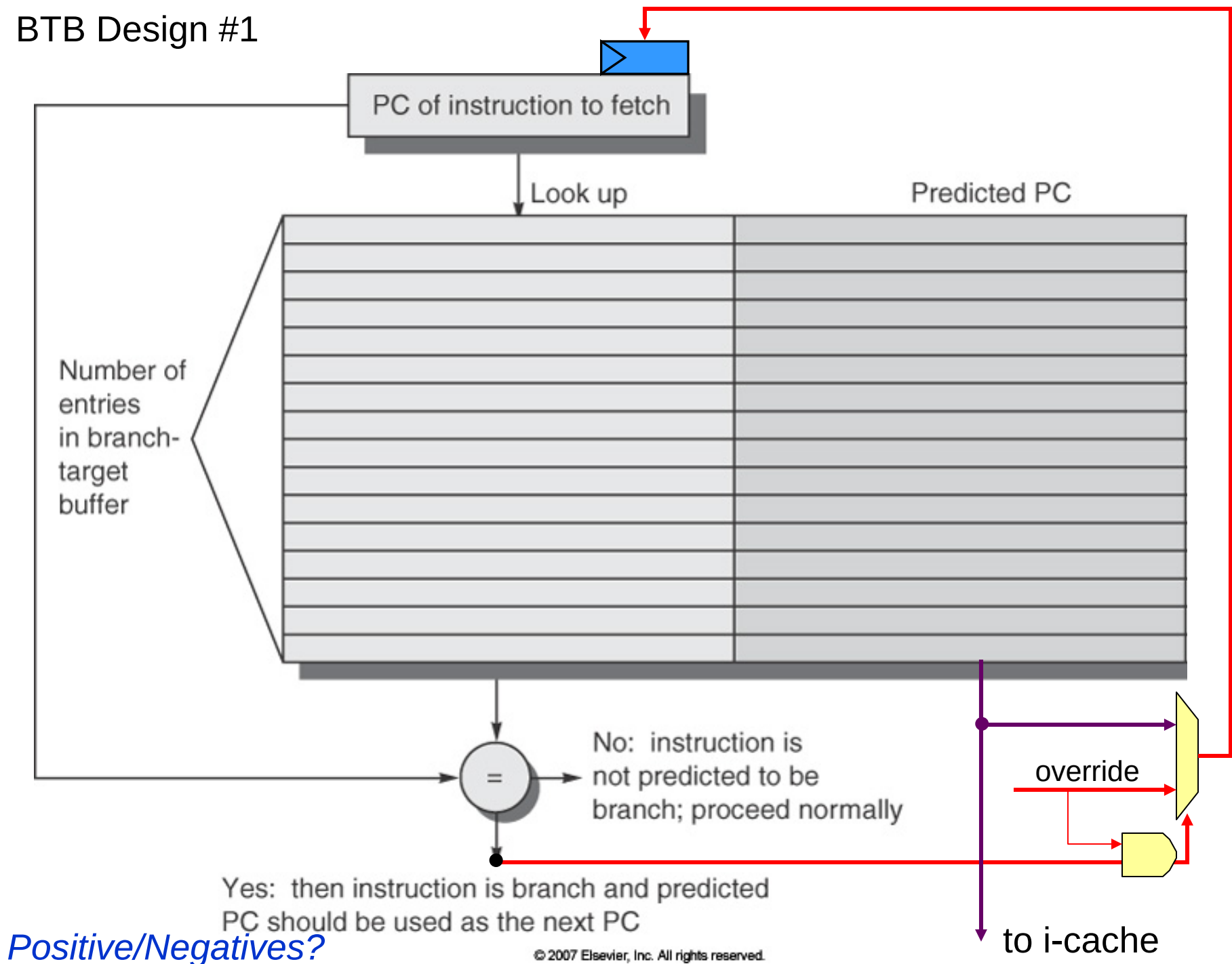
## 6. Branch Target Buffer

# Branch Target Buffer

Positive/Negatives?



## BTB Design #1

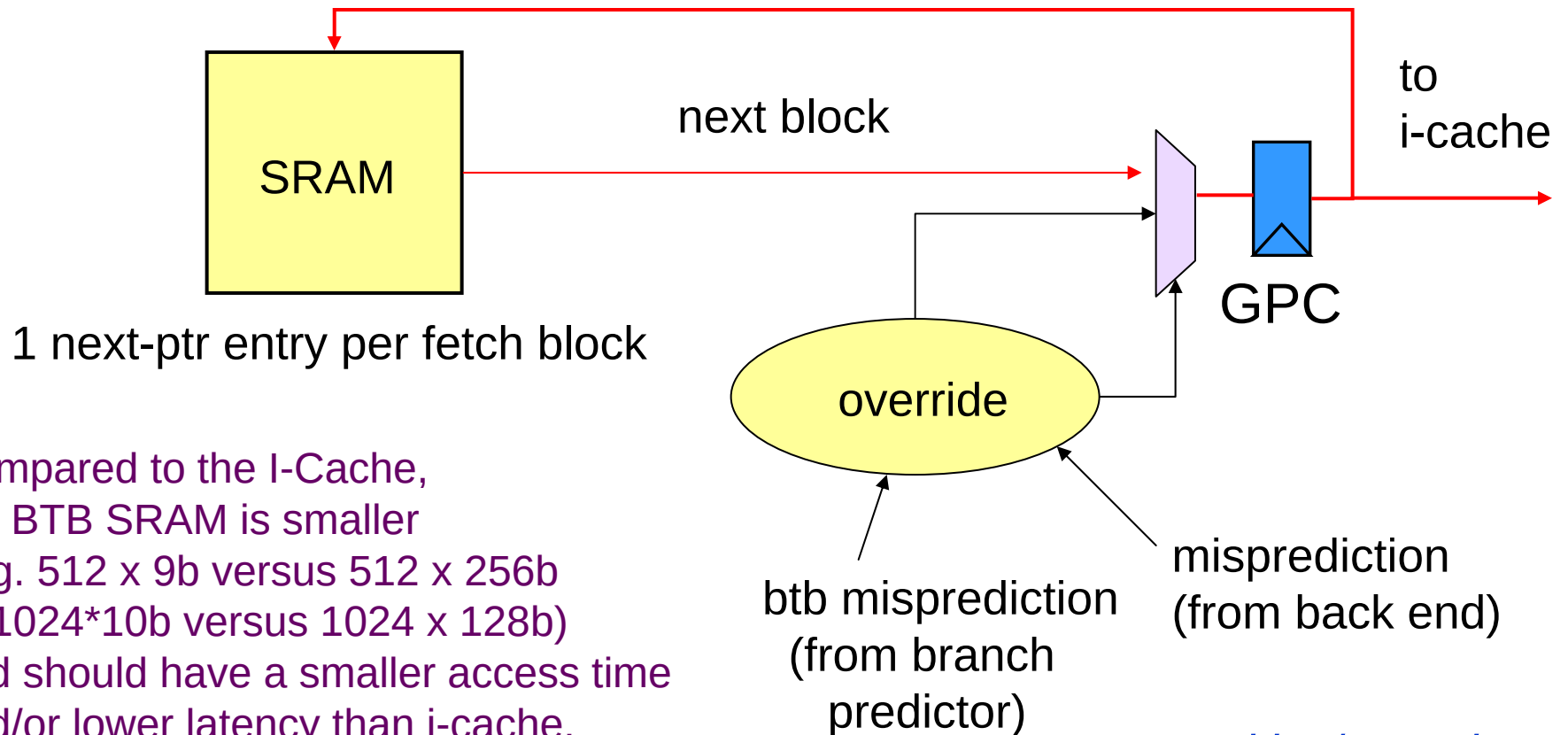


*Positive/Negatives?*

# Simple, Fast “Next-Ptr” BTB design – a la Alpha 21264

BTB selects next *fetch block* to access. Update mechanism (not shown) may include some hysteresis ala 2-bit predictor, and does not need to be on the critical path.

(The red line is the critical path – [in the Raw tile, this was the critical path of the design] - which can be optimized down to the latency through the SRAM, a Mux, and a latch.)



Compared to the I-Cache, the BTB SRAM is smaller (e.g. 512 x 9b versus 512 x 256b or 1024\*10b versus 1024 x 128b) and should have a smaller access time and/or lower latency than i-cache.

*Positive/Negatives?*