

# **CS 152 Computer Architecture and Engineering**

## **CS252 Graduate Computer Architecture**

### **Lecture 12 – Branch Prediction and Advanced Out-of-Order Superscalars**

Krste Asanovic

Electrical Engineering and Computer Sciences  
University of California at Berkeley

**`http://www.eecs.berkeley.edu/~krste`**  
**`http://inst.eecs.berkeley.edu/~cs152`**

# Last Time in Lecture 11

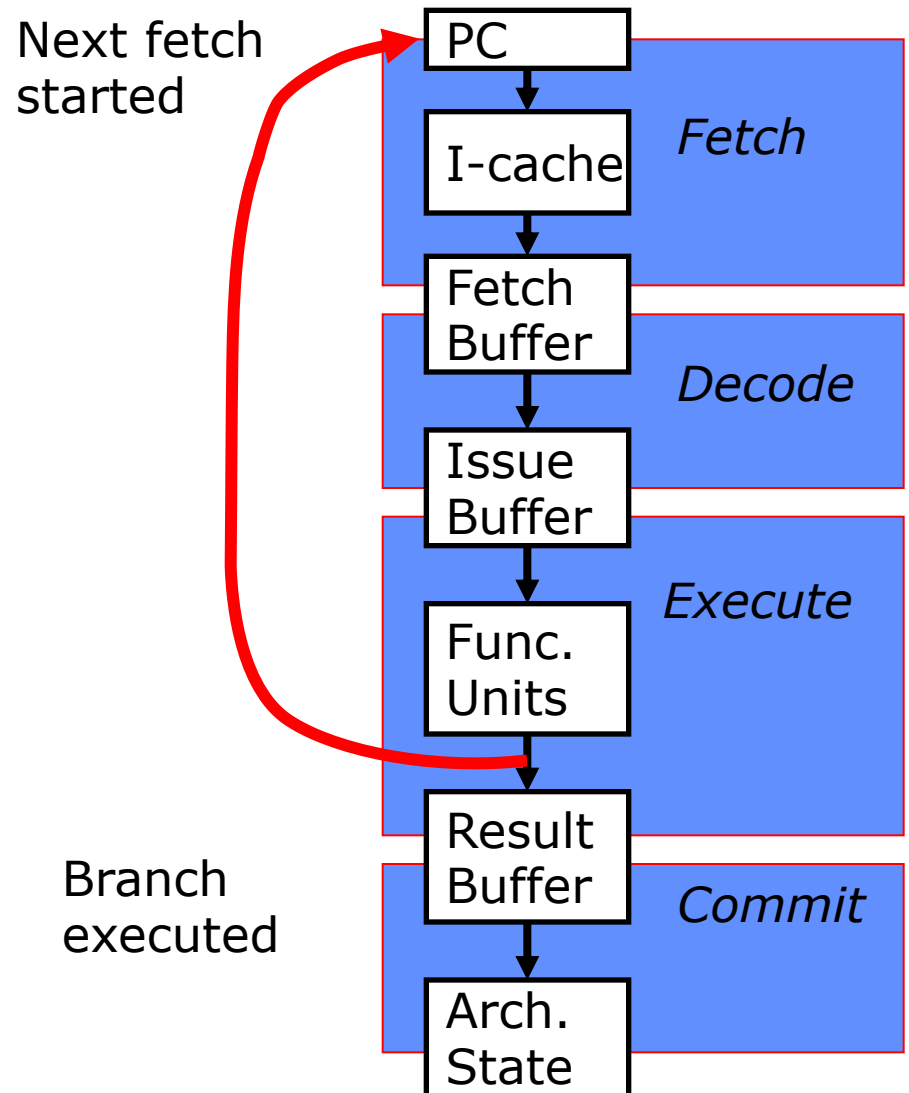
- Phases of instruction execution:
  - Fetch/decode/rename/dispatch/issue/execute/complete/commit
- Data-in-ROB design versus unified physical register design
- Superscalar register renaming

# 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 + buffers**



# 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 because branches often in critical path through code)

## ■ Hardware solutions

- Find something else to do (delay slots)
  - Replaces pipeline bubbles with useful work (requires software cooperation) – quickly see diminishing returns
- Speculate, i.e., branch prediction
  - Speculative execution of instructions beyond the branch
  - Many advances in accuracy, widely used

# 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:*

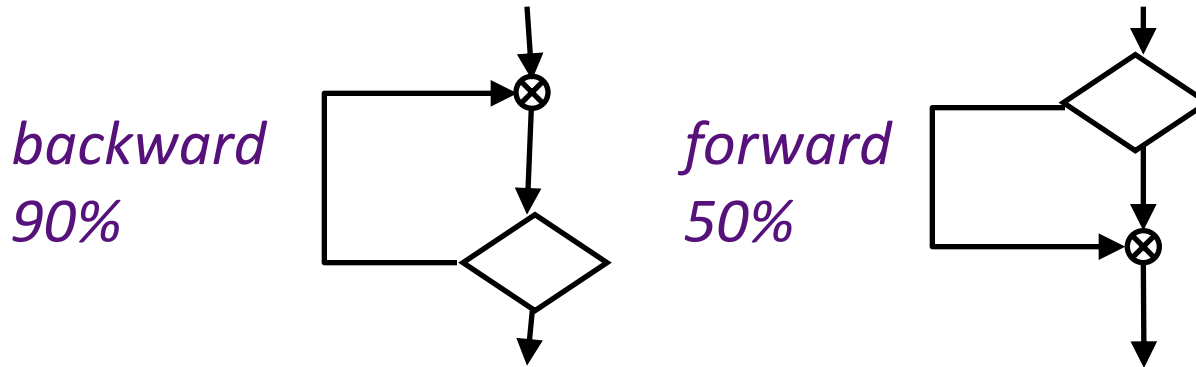
- *Keep result computation separate from commit*
- Kill instructions following branch in pipeline
- Restore state to that following branch

# Importance of Branch Prediction

- Consider 4-way superscalar with 8 pipeline stages from fetch to dispatch, and 80-entry ROB, and 3 cycles from issue to branch resolution
- On a mispredict, could throw away  $8 * 4 + (80 - 1) = 111$  instructions
- Improving from 90% to 95% prediction accuracy, removes 50% of branch mispredicts
  - If 1/6 instructions are branches, then move from 60 instructions between mispredicts, to 120 instructions between mispredicts

# Static Branch Prediction

Overall probability a branch is taken is ~60-70% 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 ~80% accurate

# 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
- Spatial correlation
  - Several branches may resolve in a highly correlated manner (a preferred path of execution)

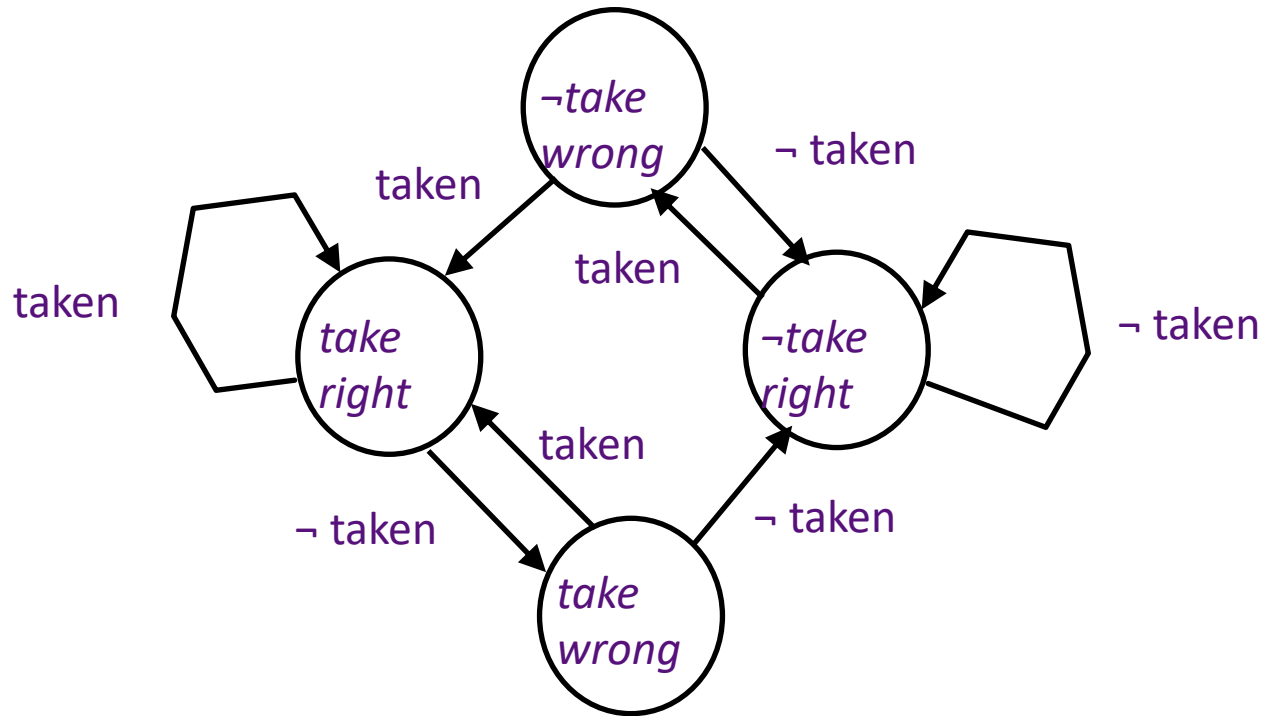


# One-Bit Branch History Predictor

- For each branch, remember last way branch went
- Has problem with loop-closing backward branches, as two mispredicts occur on every loop execution
  1. first iteration predicts loop backwards branch not-taken (loop was exited last time)
  2. last iteration predicts loop backwards branch taken (loop continued last time)

# Branch Prediction Bits

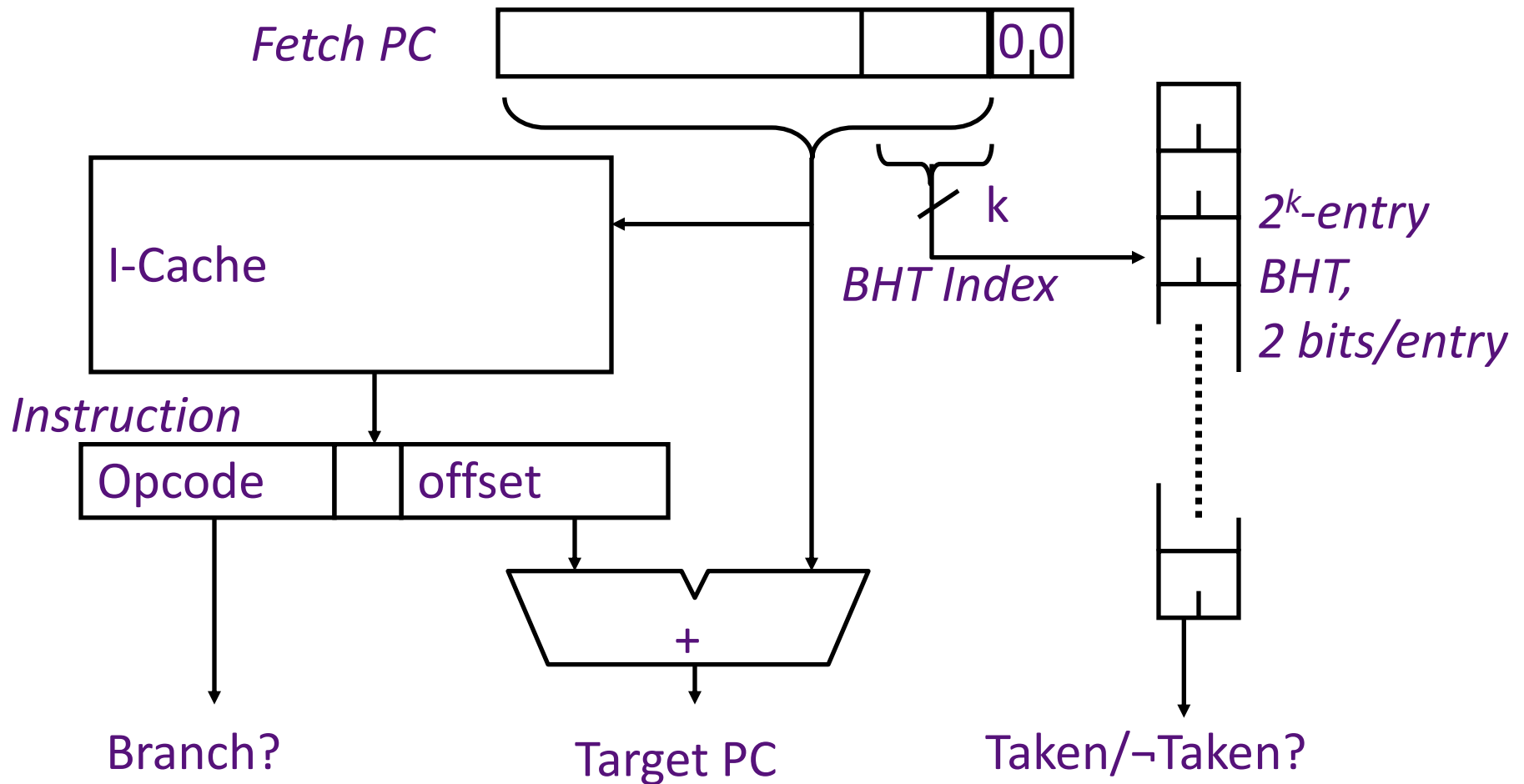
- Assume 2 BP bits per instruction
- Change the prediction after two consecutive mistakes!



*BP state:*

*(predict take/¬take) x (last prediction right/wrong)*

# Branch History Table (BHT)



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

# Exploiting Spatial Correlation

*Yeh and Patt, 1992*

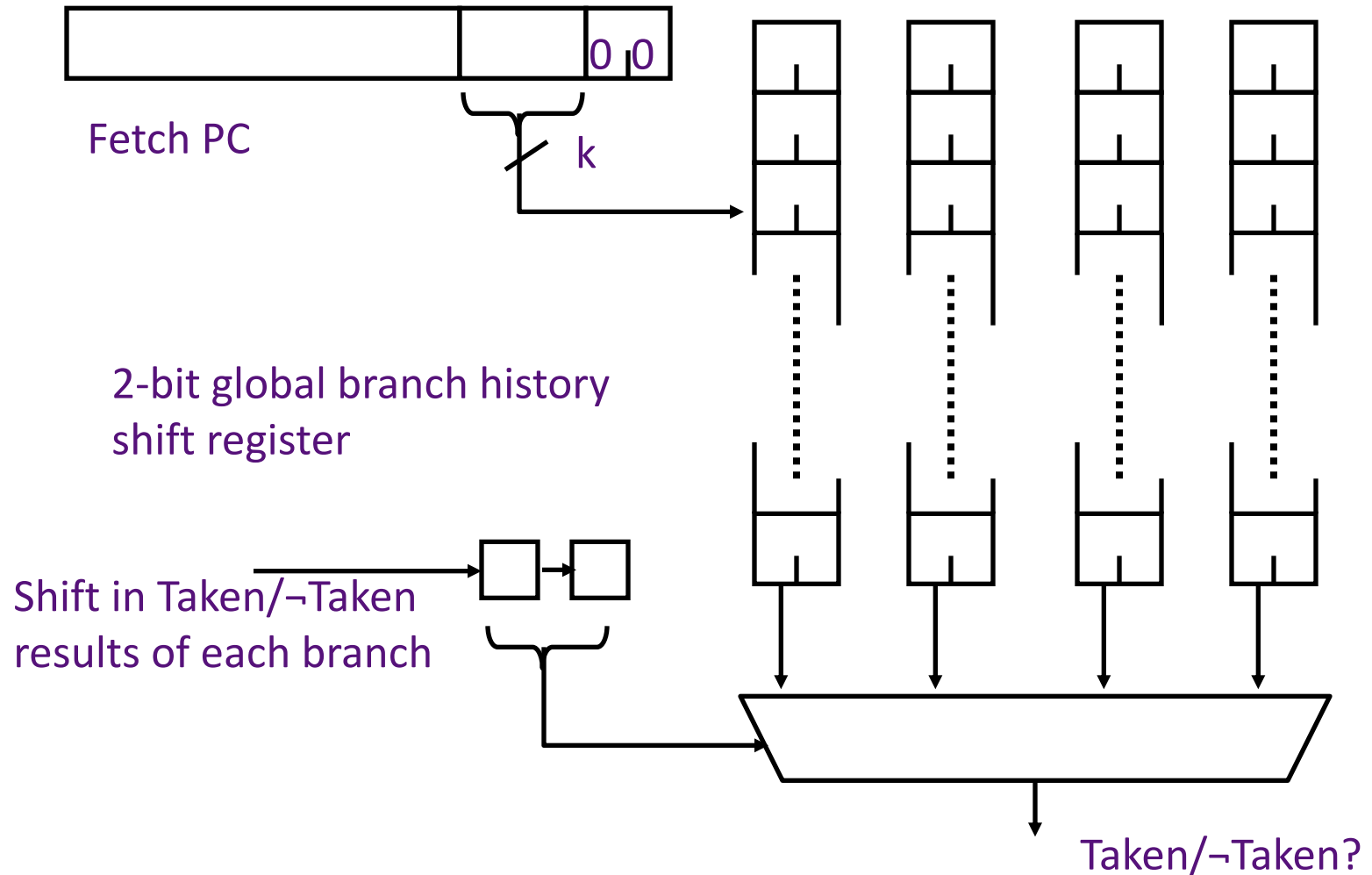
```
if (x[i] < 7) then  
    y += 1;  
if (x[i] < 5) then  
    c -= 4;
```

If first condition false, second condition also false

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

# 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)*

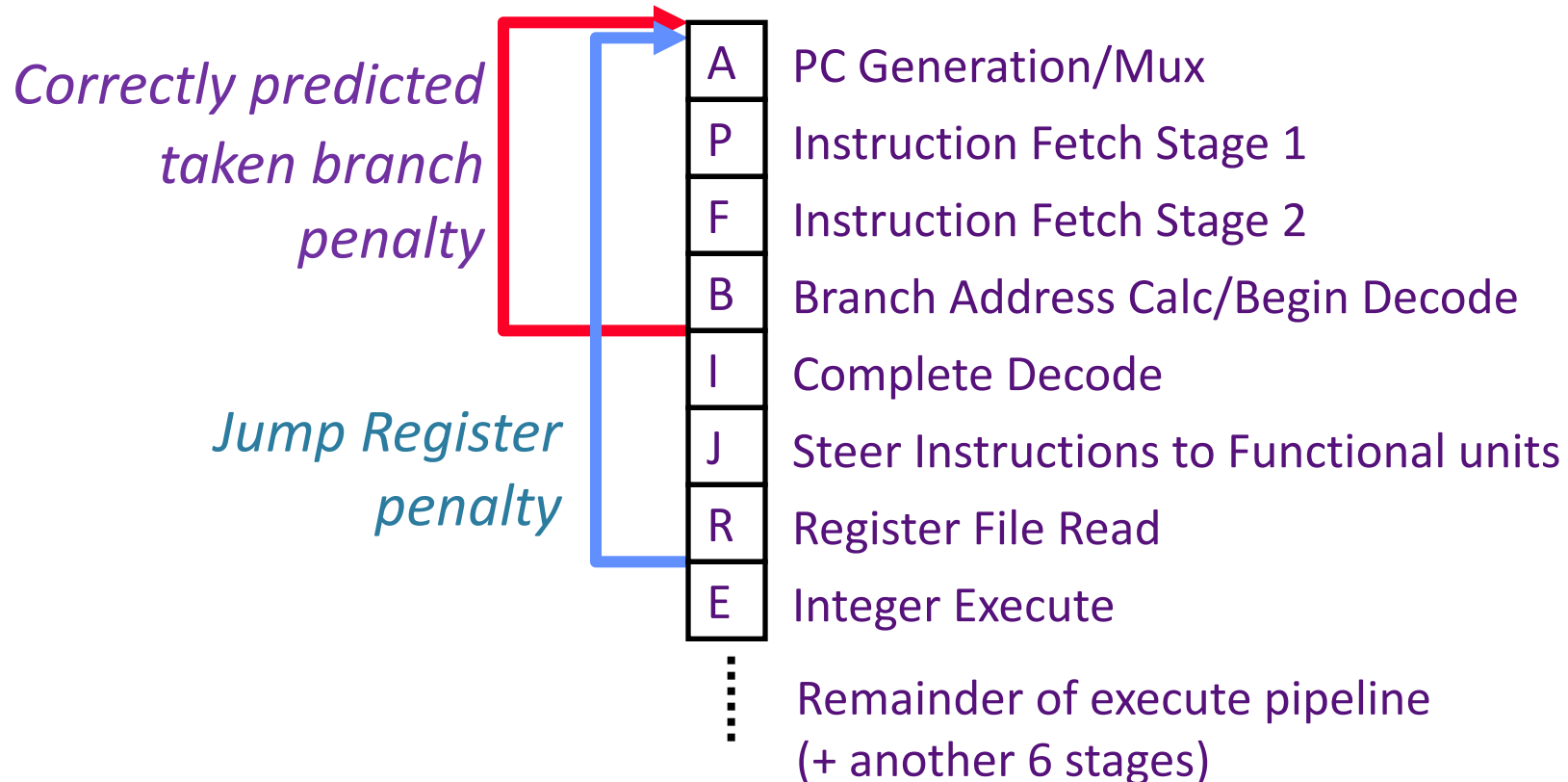


# Speculating Both Directions?

- An alternative to branch prediction is to execute both directions of a branch speculatively
  - resource requirement is proportional to the number of concurrent speculative executions
  - only half the resources engage in useful work when both directions of a branch are executed speculatively
  - branch prediction takes less resources than speculative execution of both paths
- With accurate branch prediction, it is more cost effective to dedicate all resources to the predicted direction!

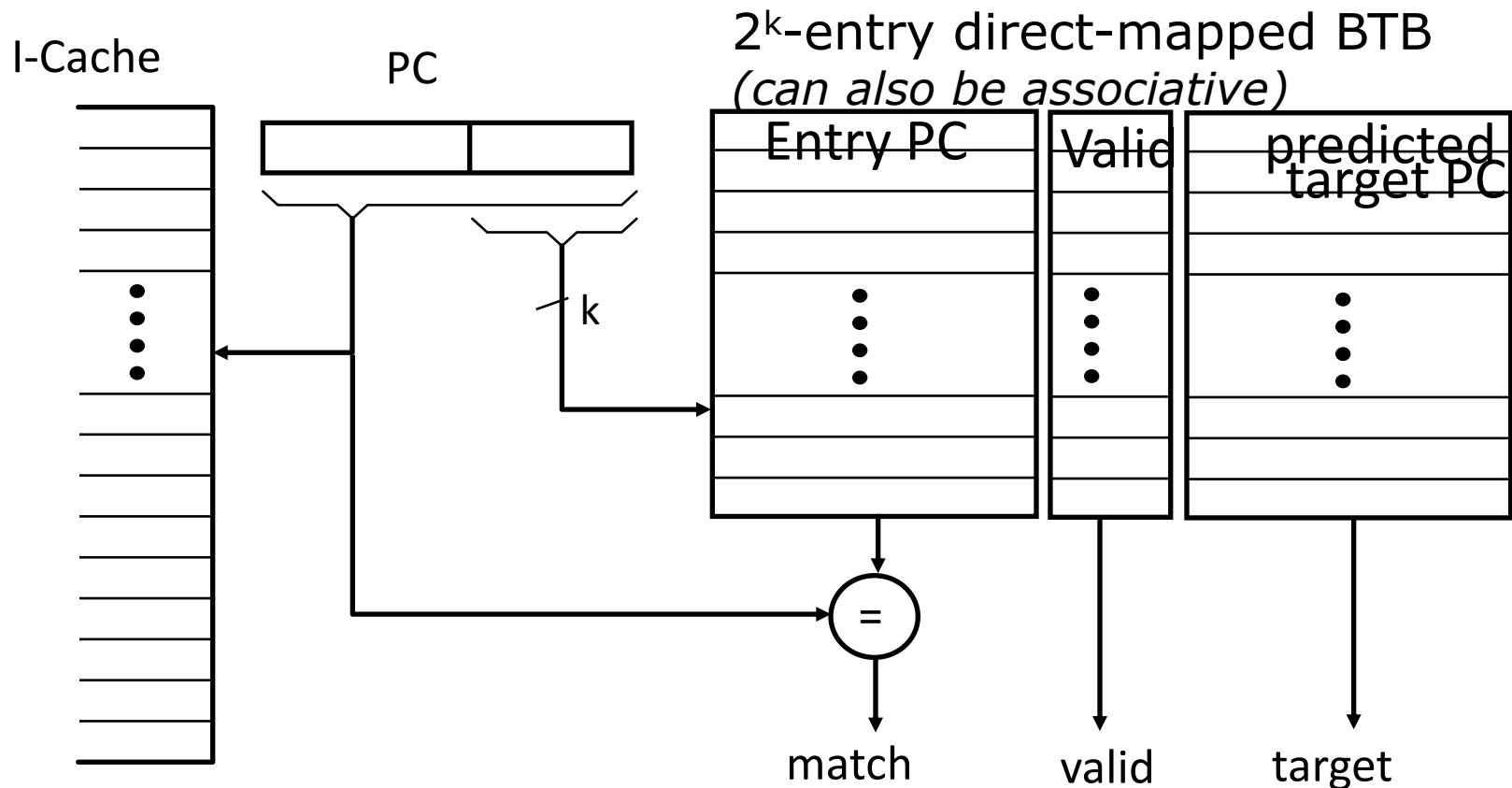
# Limitations of BHTs

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



*UltraSPARC-III fetch pipeline*

# Branch Target Buffer (BTB)

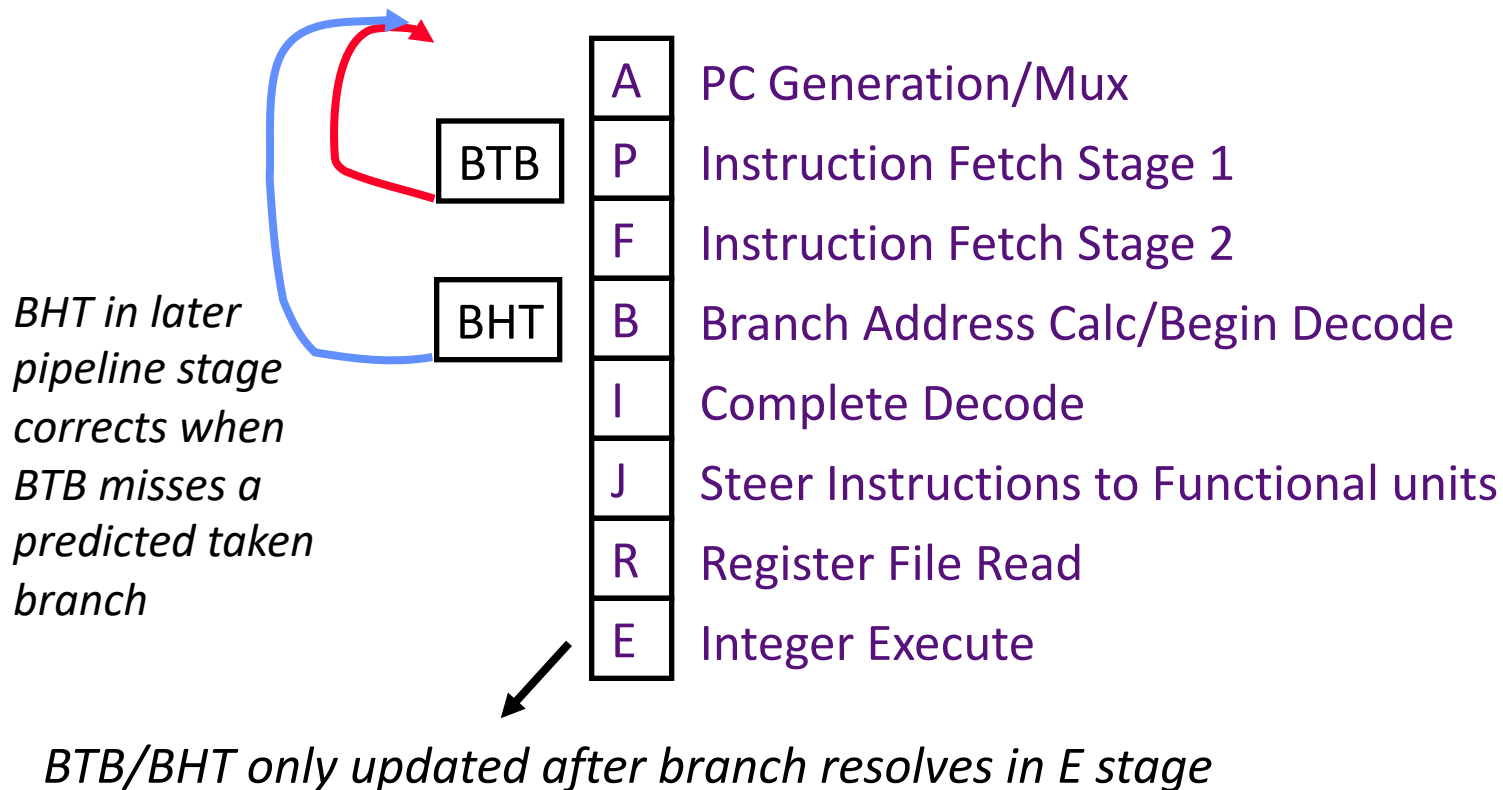


- Keep both the branch PC and target PC in the BTB
- PC+4 is fetched if match fails
- Only *taken* branches and jumps held in BTB
- Next PC determined *before* branch fetched and decoded



# 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?

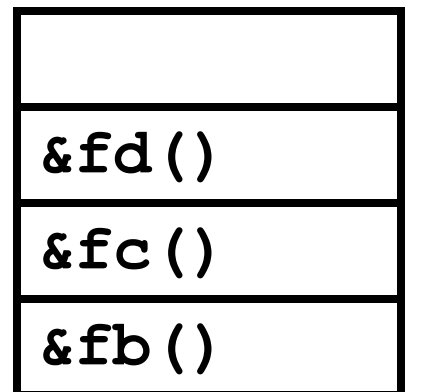
# Subroutine Return Stack

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*

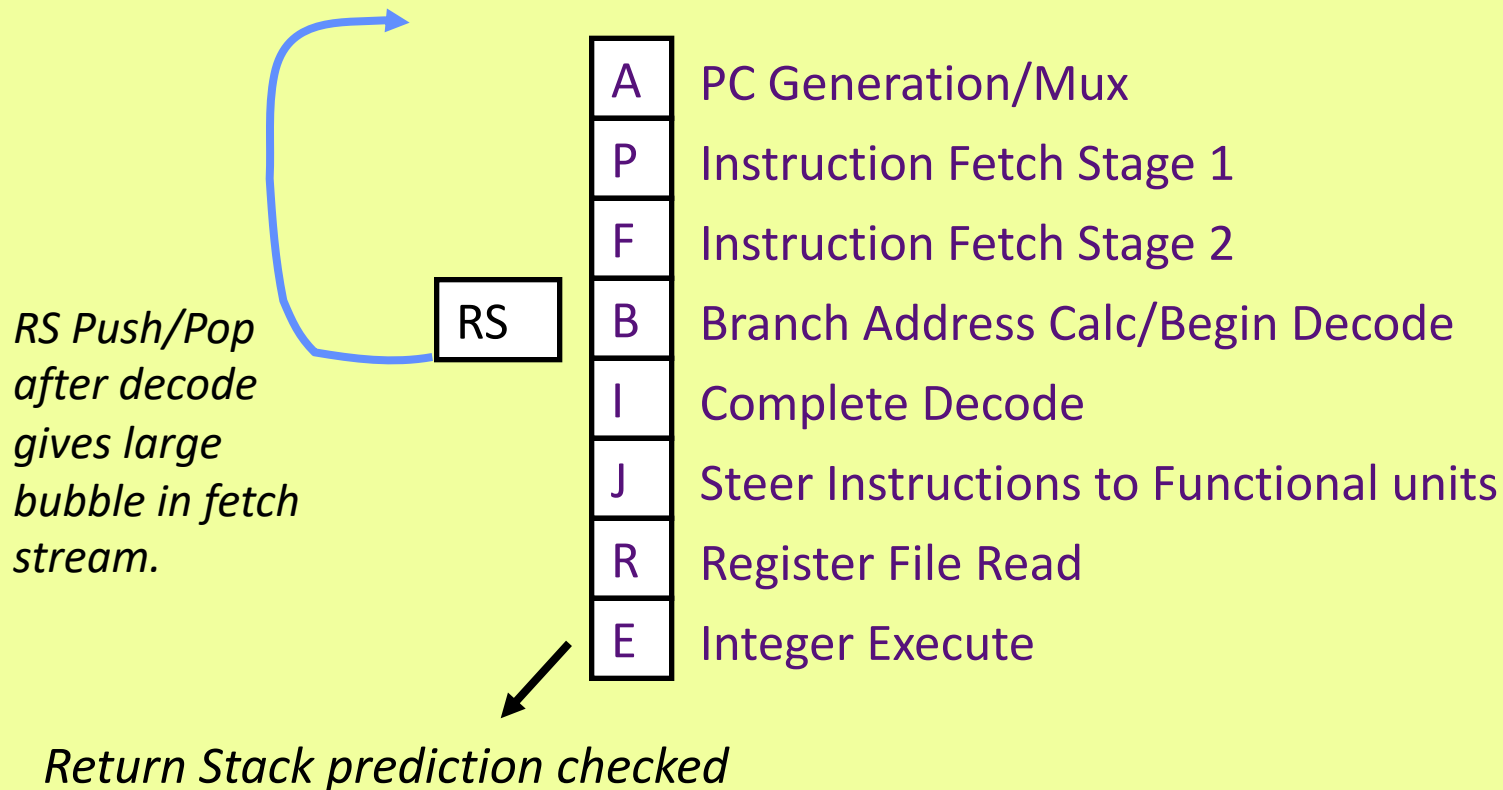
*Pop return address when  
subroutine return decoded*



*k entries  
(typically k=8-16)*

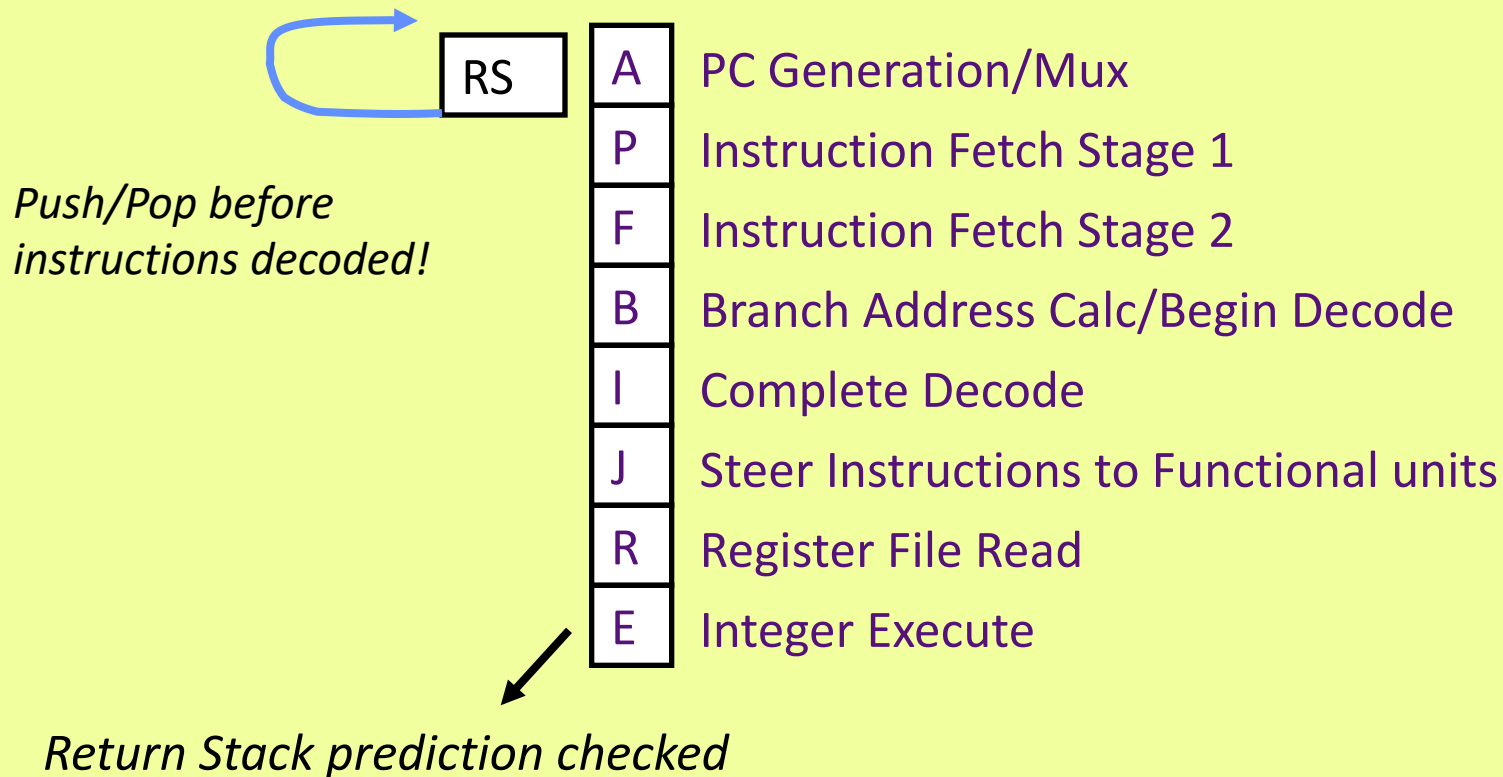
# Return Stack in Pipeline

- How to use return stack (RS) in deep fetch pipeline?
- Only know if subroutine call/return at decode

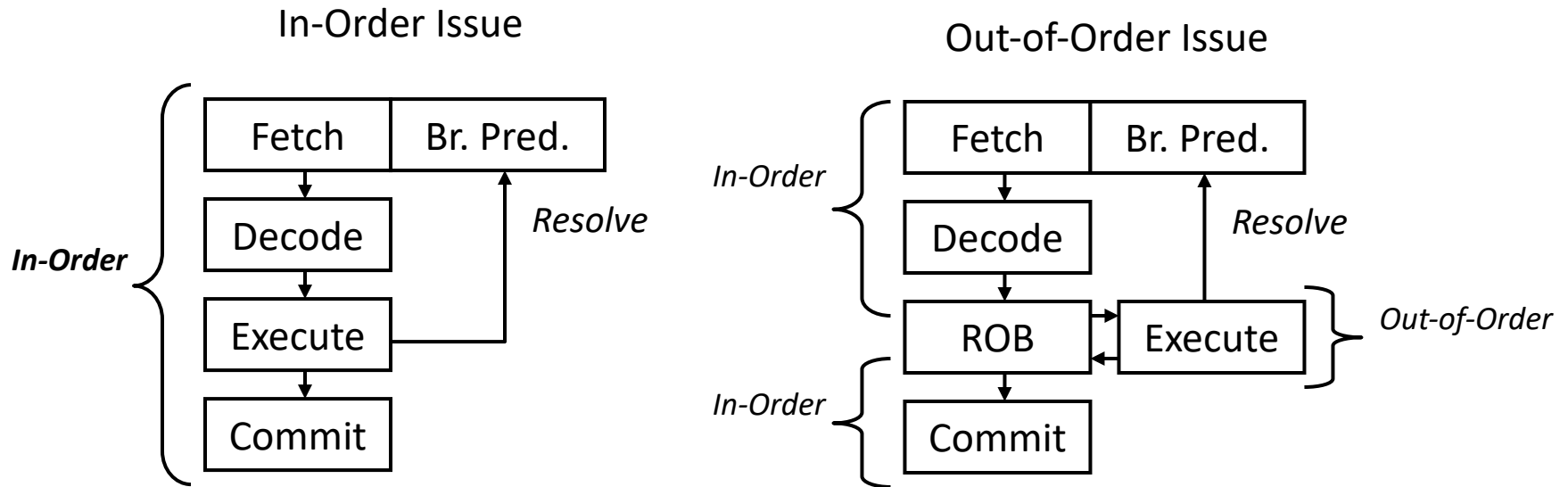


# Return Stack in Pipeline

- Can remember whether PC is subroutine call/return using BTB-like structure
- Instead of target-PC, just store push/pop bit



# In-Order vs. Out-of-Order Branch Prediction



- Speculative fetch but not speculative execution - branch resolves before later instructions complete
  - Completed values held in bypass network until commit
  - Speculative execution, with branches resolved after later instructions complete
  - Completed values held in rename registers in ROB or unified physical register file until commit
- 
- Both styles of machine can use same branch predictors in front-end fetch pipeline, and both can execute multiple instructions per cycle
  - Common to have 10-30 pipeline stages in either style of design

# InO vs. OoO Mispredict Recovery

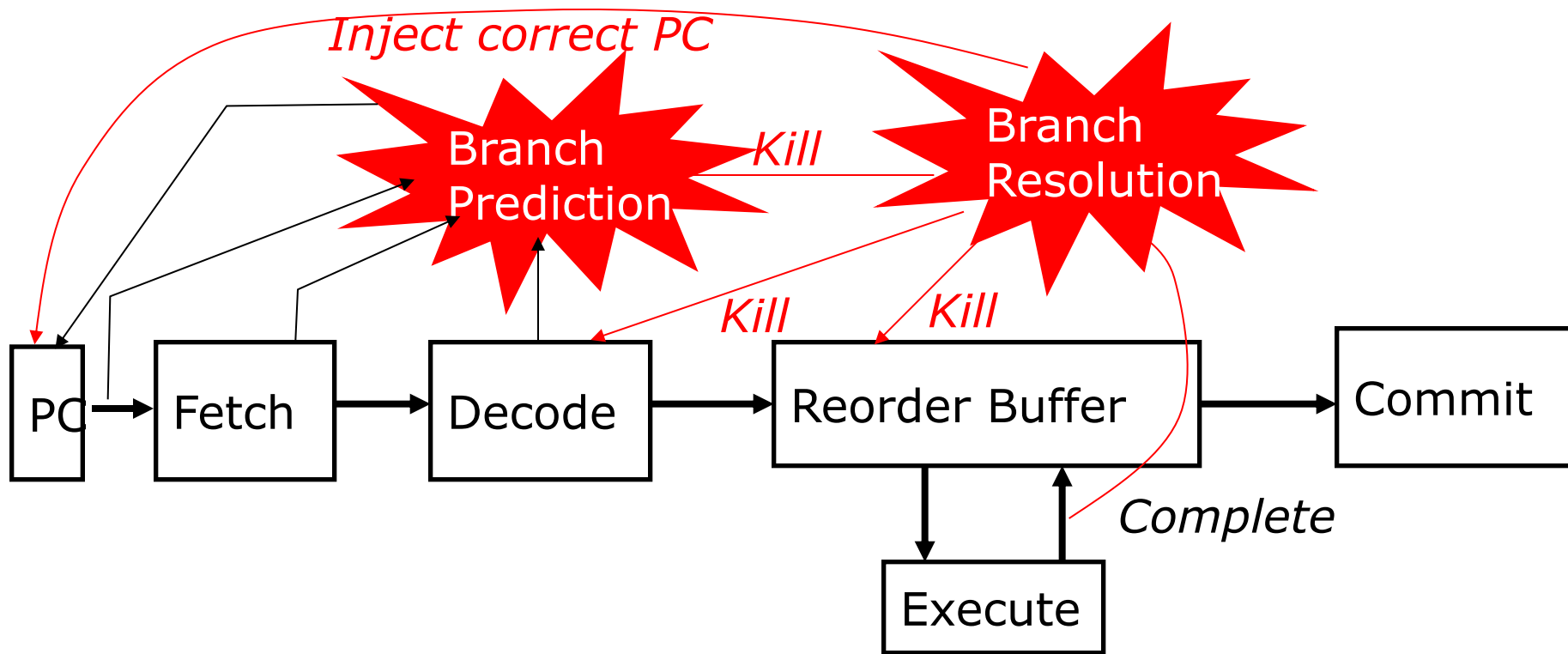
## ■ In-order execution?

- Design so no instruction issued after branch can write-back before branch resolves
- Kill all instructions in pipeline behind mispredicted branch

## ■ Out-of-order execution?

- Multiple instructions following branch in program order can complete before branch resolves
- A simple solution would be to handle like precise traps
  - Problem?

# Branch Misprediction in Pipeline



- Can have multiple unresolved branches in ROB
- Can resolve branches out-of-order by killing all the instructions in ROB that follow a mispredicted branch
- MIPS R10K uses four mask bits to tag instructions that are dependent on up to four speculative branches
- Mask bits cleared as branch resolves, and reused for next branch



# Rename Table Recovery

- Have to quickly recover rename table on branch mispredicts
- MIPS R10K only has four snapshots for each of four outstanding speculative branches
- Alpha 21264 has 80 snapshots, one per ROB instruction

# CS152 Administrivia

- Lab 2 extension, due Friday March 15
- PS 3 due Monday March 18

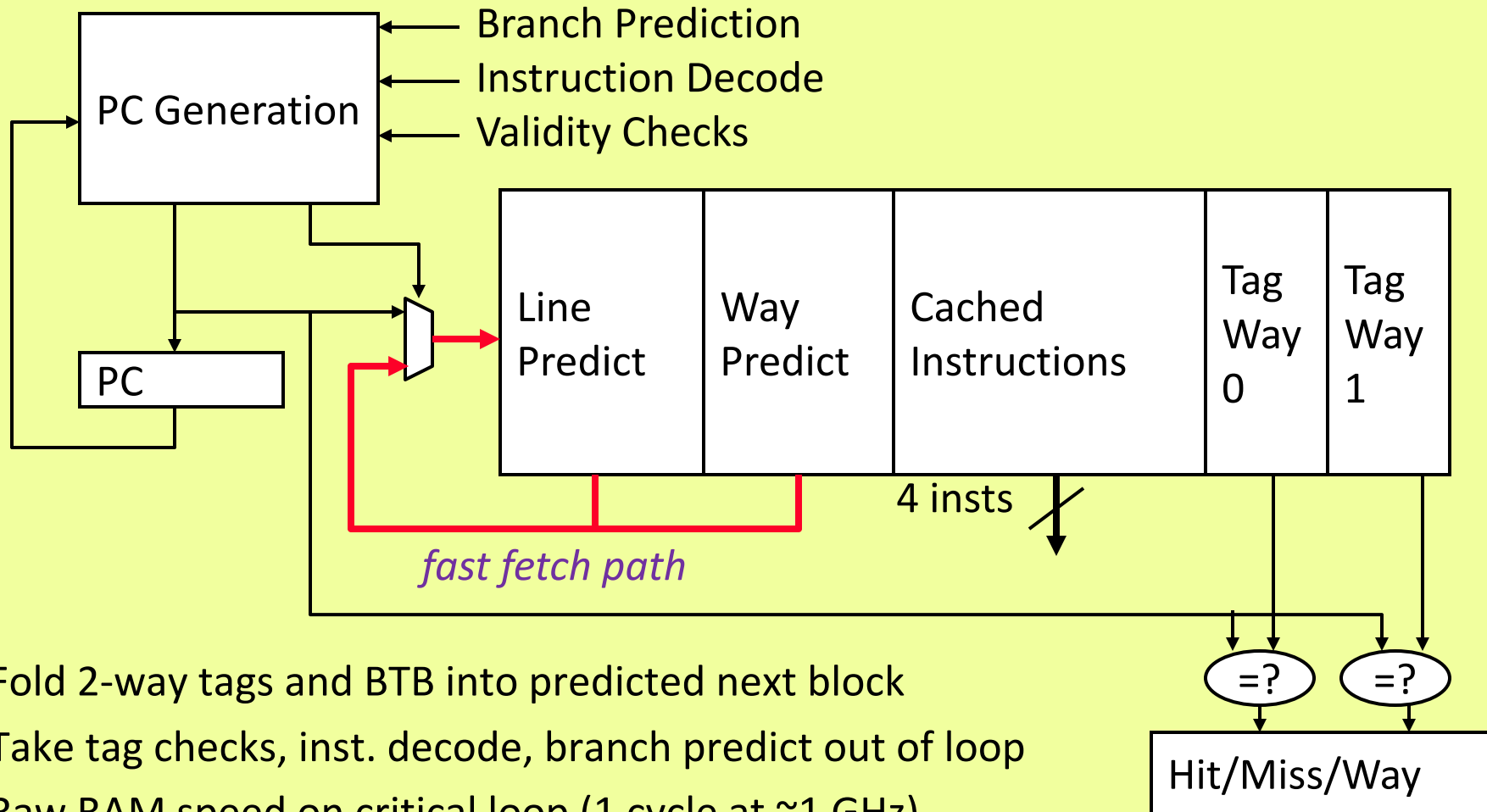
# CS252 Administrivia

- Readings next week on OoO superscalar microprocessors

# Improving Instruction Fetch

- Performance of speculative out-of-order machines often limited by instruction fetch bandwidth
  - speculative execution can fetch 2-3x more instructions than are committed
  - mispredict penalties dominated by time to refill instruction window
  - taken branches are particularly troublesome

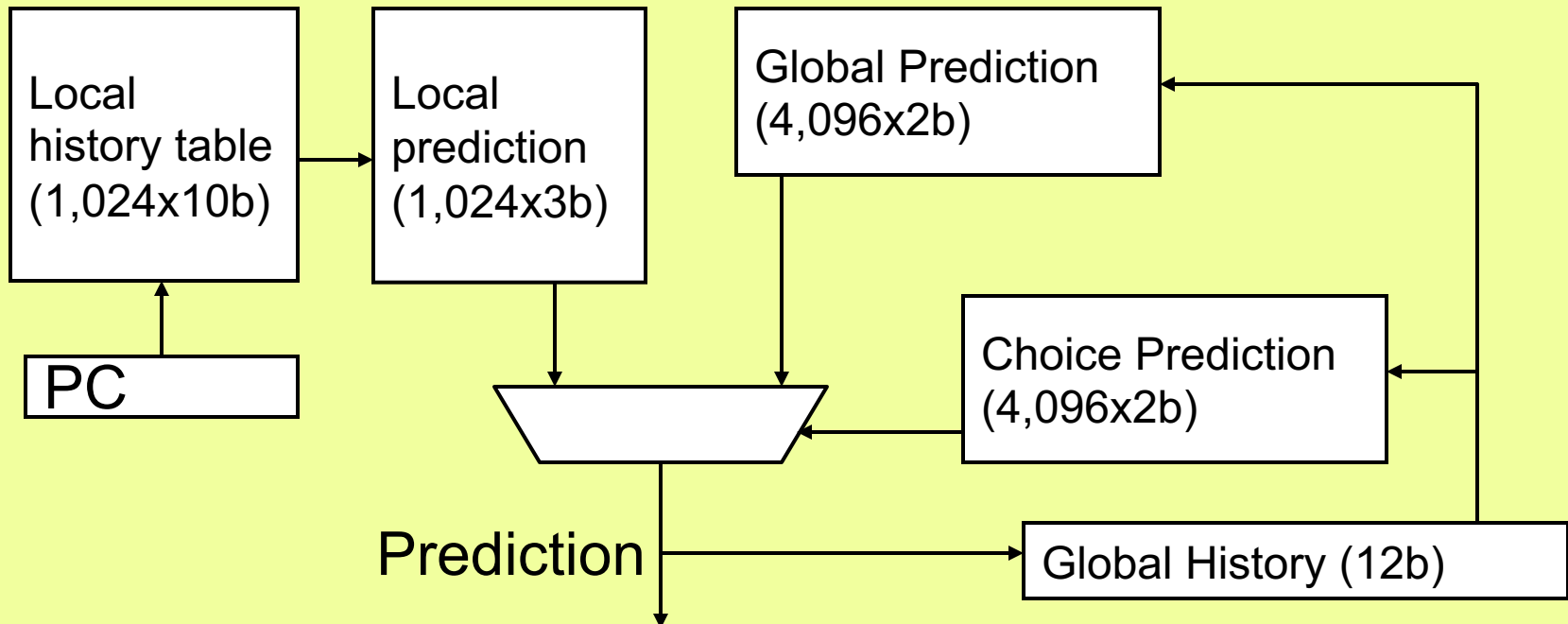
# Increasing Taken Branch Bandwidth (Alpha 21264 I-Cache)



- Fold 2-way tags and BTB into predicted next block
- Take tag checks, inst. decode, branch predict out of loop
- Raw RAM speed on critical loop (1 cycle at ~1 GHz)
- 2-bit hysteresis counter per block prevents overtraining

# Tournament Branch Predictor (Alpha 21264)

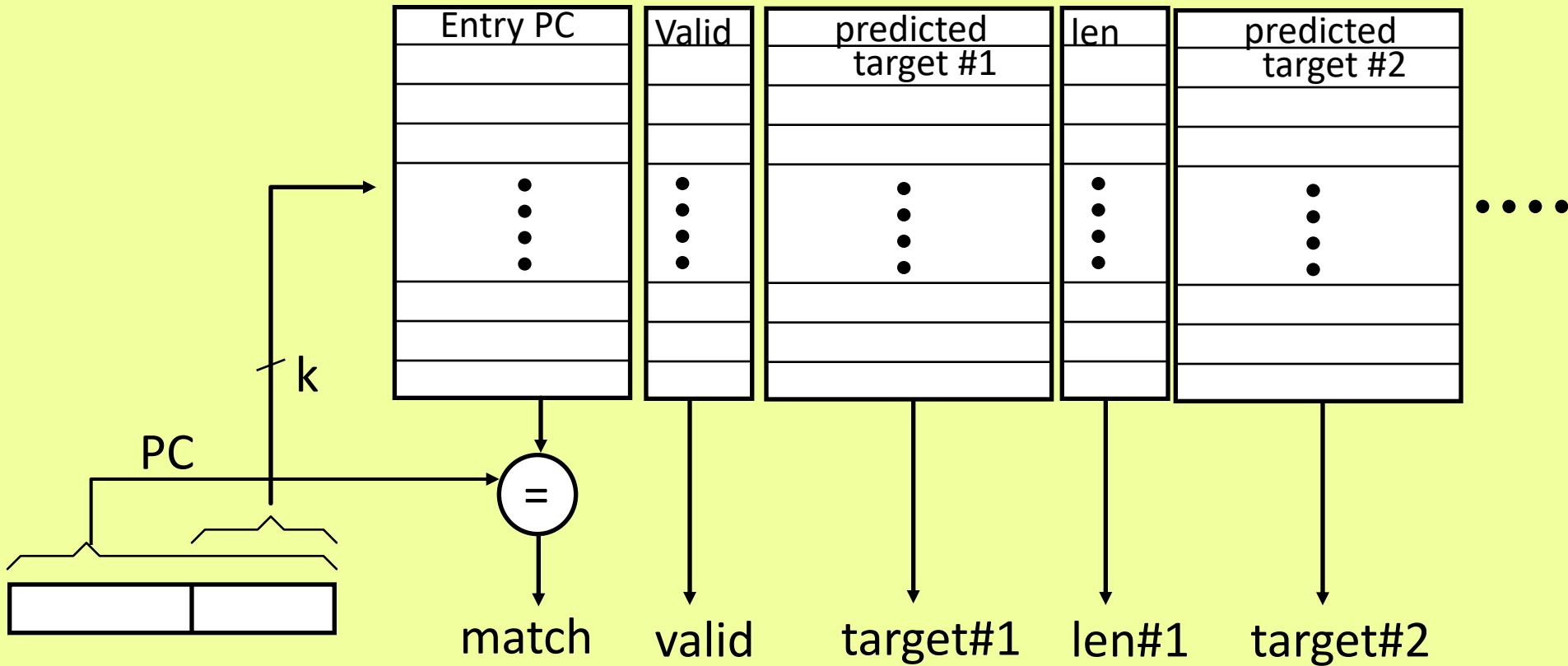
- Choice predictor learns whether best to use local or global branch history in predicting next branch
- Global history is speculatively updated but restored on mispredict
- Claim 90-100% success on range of applications



# Taken Branch Limit

- Integer codes have a taken branch every 6-9 instructions
- To avoid fetch bottleneck, must execute multiple taken branches per cycle when increasing performance
- This implies:
  - predicting multiple branches per cycle
  - fetching multiple non-contiguous blocks per cycle

# Branch Address Cache (Yeh, Marr, Patt)



Extend BTB to return multiple branch predictions per cycle

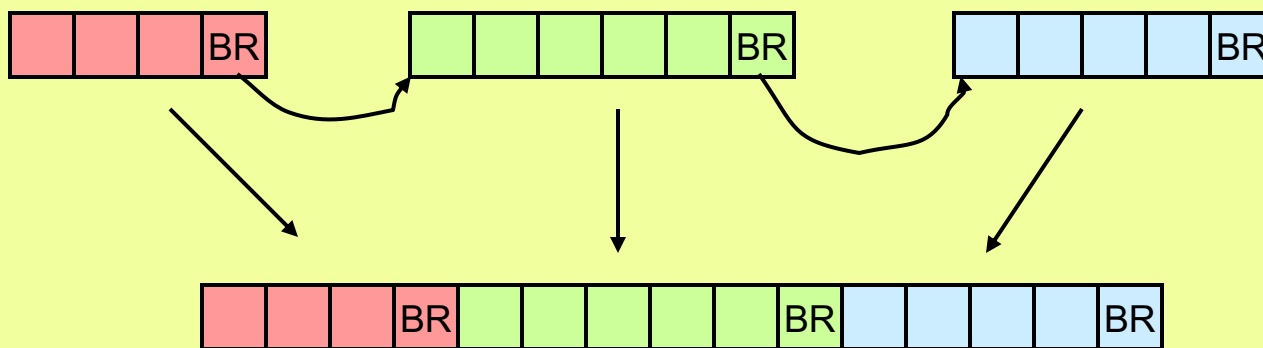


# Fetching Multiple Basic Blocks

- Requires either
  - multiported cache: expensive
  - interleaving: bank conflicts will occur
- Merging multiple blocks to feed to decoders adds latency, increasing mispredict penalty and reducing branch throughput

# Trace Cache

- Key Idea: Pack multiple non-contiguous basic blocks into one contiguous trace cache line

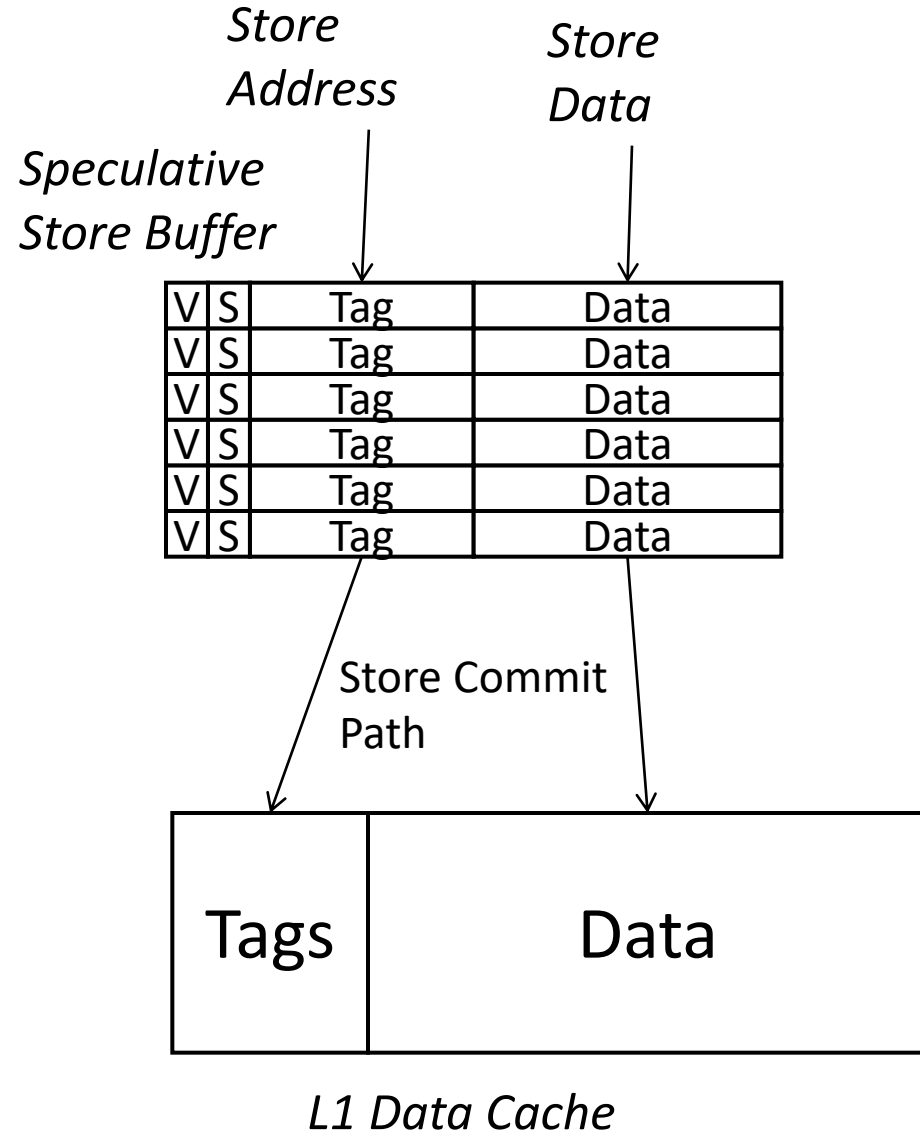


- Single fetch brings in multiple basic blocks
- Trace cache indexed by start address *and* next  $n$  branch predictions
- Used in Intel Pentium-4 processor to hold decoded uops

# Load-Store Queue Design

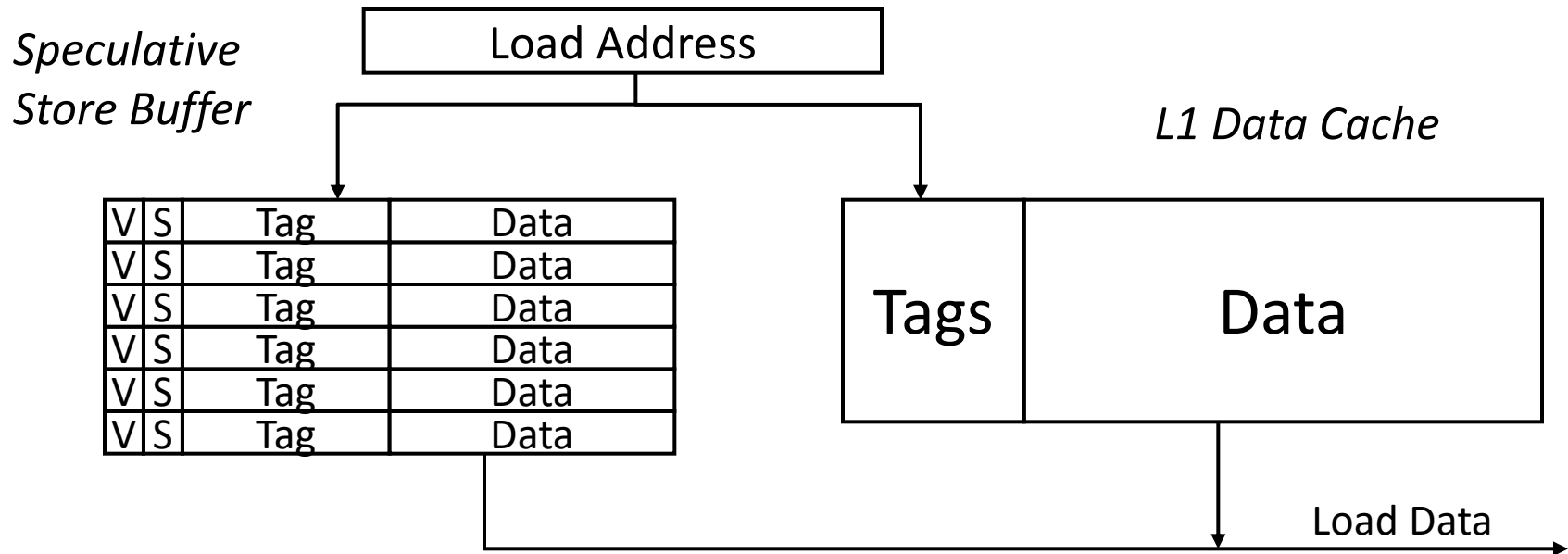
- After control hazards, data hazards through memory are probably next most important bottleneck to superscalar performance
- Modern superscalars use very sophisticated load-store reordering techniques to reduce effective memory latency by allowing loads to be speculatively issued

# Speculative Store Buffer



- Just like register updates, stores should not modify the memory until after the instruction is committed. A speculative store buffer is a structure introduced to hold speculative store data.
- During decode, store buffer slot allocated in program order
- Stores split into “store address” and “store data” micro-operations
- “Store address” execution writes tag
- “Store data” execution writes data
- Store commits when oldest instruction and both address and data available:
  - clear speculative bit and eventually move data to cache
- On store abort:
  - clear valid bit

# Load bypass from speculative store buffer



- If data in both store buffer and cache, which should we use?

Speculative store buffer

- If same address in store buffer twice, which should we use?

Youngest store older than load

# Memory Dependencies

```
sd x1, (x2)  
ld x3, (x4)
```

- When can we execute the load?

# In-Order Memory Queue

- Execute all loads and stores in program order

=> Load and store cannot leave ROB for execution until all previous loads and stores have completed execution

- Can still execute loads and stores speculatively, and out-of-order with respect to other instructions
- Need a structure to handle memory ordering...

# Conservative O-o-O Load Execution

**sd x1, (x2)**

**ld x3, (x4)**

- Can execute load before store, if addresses known and **x4** **!= x2**
- Each load address compared with addresses of all previous uncommitted stores
  - can use partial conservative check i.e., bottom 12 bits of address, to save hardware
- Don't execute load if any previous store address not known
- (MIPS R10K, 16-entry address queue)



# Address Speculation

```
sd x1, (x2)
```

```
ld x3, (x4)
```

- Guess that **x4** != **x2**
- Execute load before store address known
- Need to hold all completed but uncommitted load/store addresses in program order
- If subsequently find **x4==x2**, squash load and all following instructions
- => Large penalty for inaccurate address speculation

# Memory Dependence Prediction (Alpha 21264)

```
sd x1, (x2)
ld x3, (x4)
```

- Guess that **x4** != **x2** and execute load before store
- If later find **x4**==**x2**, squash load and all following instructions, but mark load instruction as store-wait
- Subsequent executions of the same load instruction will wait for all previous stores to complete
- Periodically clear store-wait bits

# Acknowledgements

- This course is partly inspired by previous MIT 6.823 and Berkeley CS252 computer architecture courses created by my collaborators and colleagues:
  - Arvind (MIT)
  - Joel Emer (Intel/MIT)
  - James Hoe (CMU)
  - John Kubiatowicz (UCB)
  - David Patterson (UCB)