上海科技大学
ShanghaiTech University

CS211
Advanced Computer Architecture

L11 Branch Prediction

Chundong Wang

October 29th, 2025

Previously in CS211

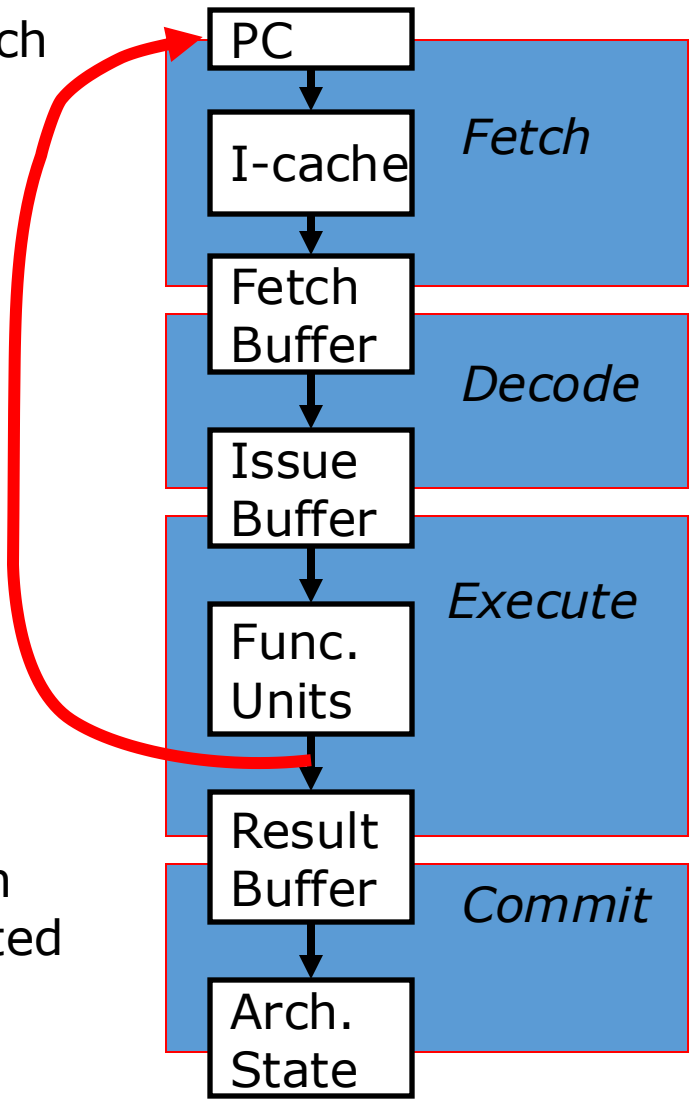
- Out-of-order Execution
  - Scoreboarding algorithm
  - Register renaming
- Issue, completion, commit
  - In-order or out-of-order

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 between branches
  - 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
  - Bypass – usually results are used immediately
  - Architectural change – 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:

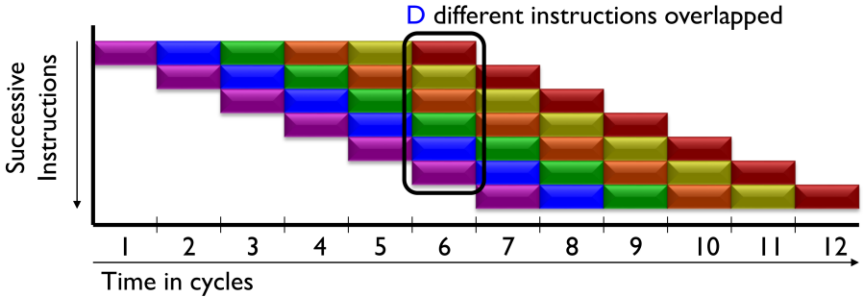
- Branch history tables, branch target buffers, etc.

Mispredict recovery mechanisms:

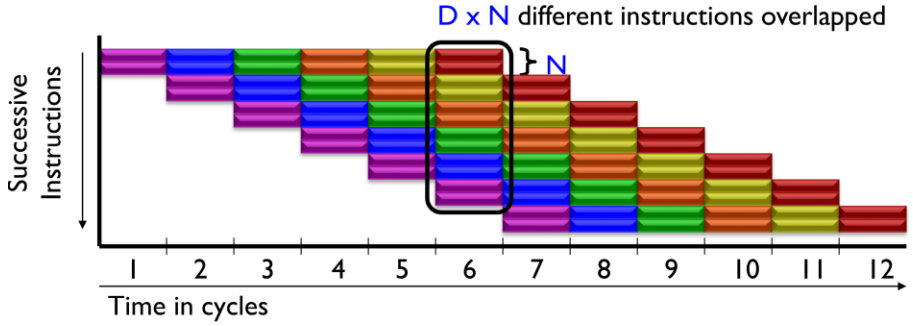
- *Keep result computation separate from commit*
- Kill instructions following branch in pipeline
- Restore state to that following branch

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 misprediction, could throw away 8*4+(80-1)=111 instructions



Scalar (never run more than 1 insn per cycle)



Superscalar (executing multiple insns in parallel)

Static Branch Prediction (I)

- Always not-taken
  - Simple to implement: no need for data structures, no direction prediction
  - Low accuracy: ~30-40%
  - Compiler can layout code such that the likely path is the “not-taken” path
- Always taken
  - No direction prediction
  - Better accuracy: ~60-70%
    - Backward branches (i.e. loop branches) are usually taken
    - Backward branch: target address lower than branch PC
- Backward taken, forward not taken (BTFN)
  - Predict backward (loop) branches as taken, others not-taken

Static Branch Prediction (II)

- Profile-based
  - Idea: Compiler determines likely direction for each branch using profile run. Encodes that direction as a hint bit in the branch instruction format.

+ Per branch prediction (more accurate than schemes in previous slide)
  → accurate if profile is representative!

-- Requires hint bits in the branch instruction format

-- Accuracy depends on dynamic branch behavior:

TTTTTTTTTTNNNNNNNNNN → 50% accuracy
TNTNTNTNTNTNTNTNTNTN → 50% accuracy

-- Accuracy depends on the representativeness of profile input set

Static Branch Prediction (III)

- Program-based (or, program analysis based)
  - Idea: Use heuristics based on program analysis to determine statically-predicted direction
  - Opcode heuristic: Predict BLEZ as NT (negative integers used as error values in many programs)
  - Loop heuristic: Predict a branch guarding a loop execution as taken (i.e., execute the loop)
  - Pointer and FP comparisons: Predict not equal

+ Does not require profiling

-- Heuristics might be not representative or good

-- Requires compiler analysis and ISA support

- Ball and Larus, "Branch prediction for free," PLDI 1993.
  - 20% misprediction rate

Static Branch Prediction (III)

- Programmer-based
  - Idea: Programmer provides the statically-predicted direction
  - Via pragmas in the programming language that qualify a branch as likely-taken versus likely-not-taken

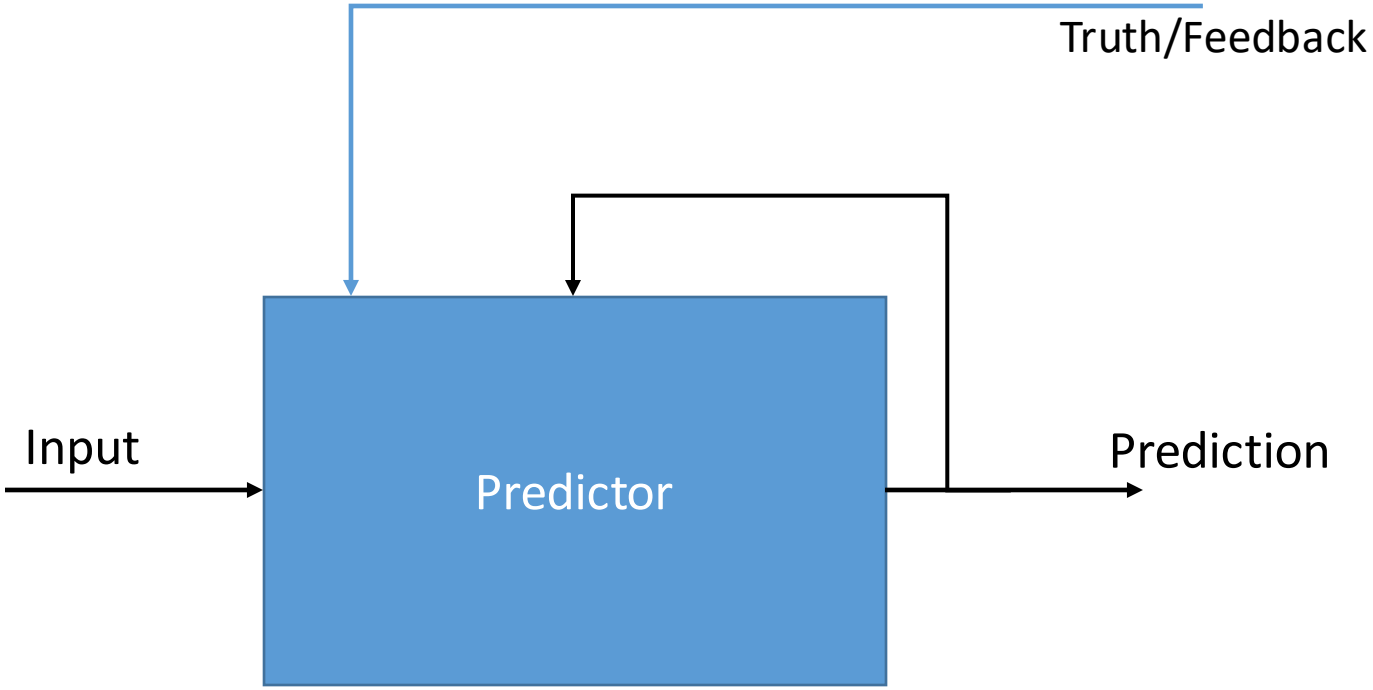
+ Does not require profiling or program analysis

+ Programmer may know some branches and their program better than other analysis techniques

-- Requires programming language, compiler, ISA support

-- Burdens the programmer?

Dynamic Prediction



Prediction as a feedback control process

Operations
- Predict
- Update

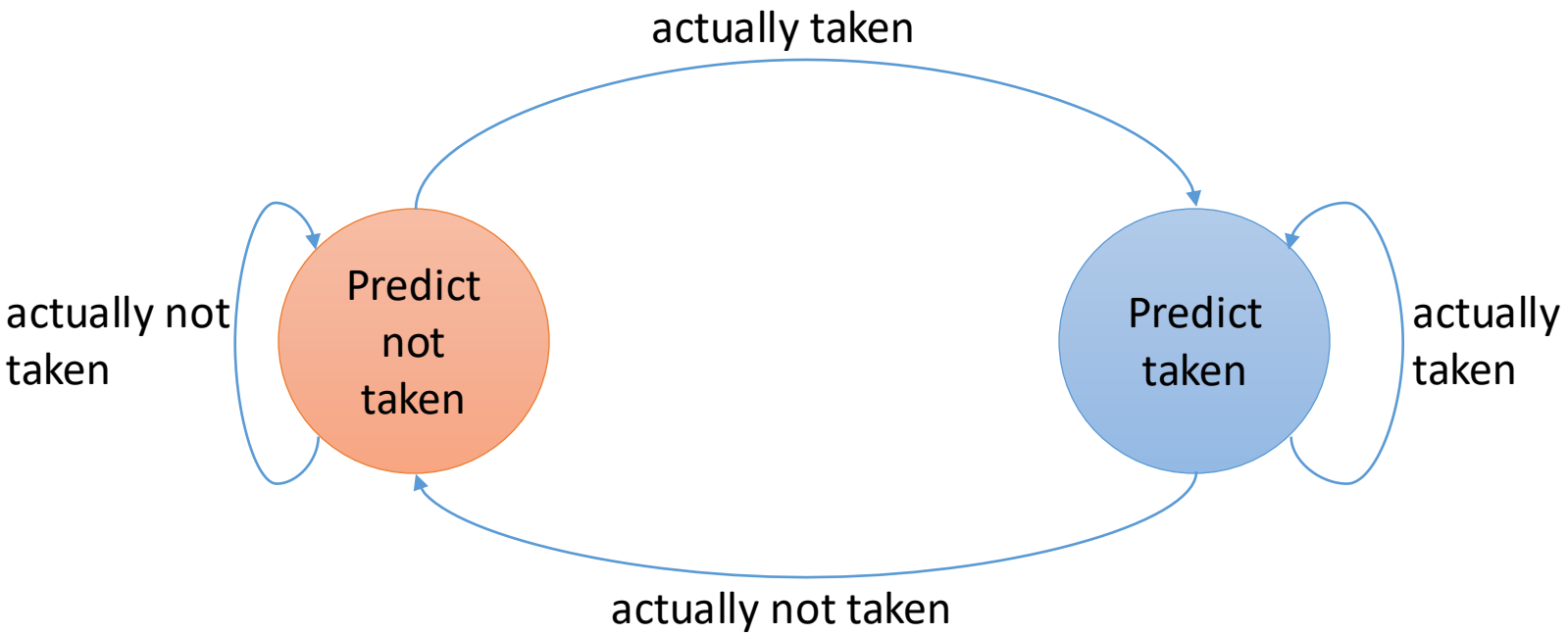
Dynamic Branch Prediction
learning based on past behavior

- 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

- Also known as Branch History Table (BHT)
- For each branch, remember last way branch went
- Has problem with loop-closing backward branches, as two mispredictions 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)

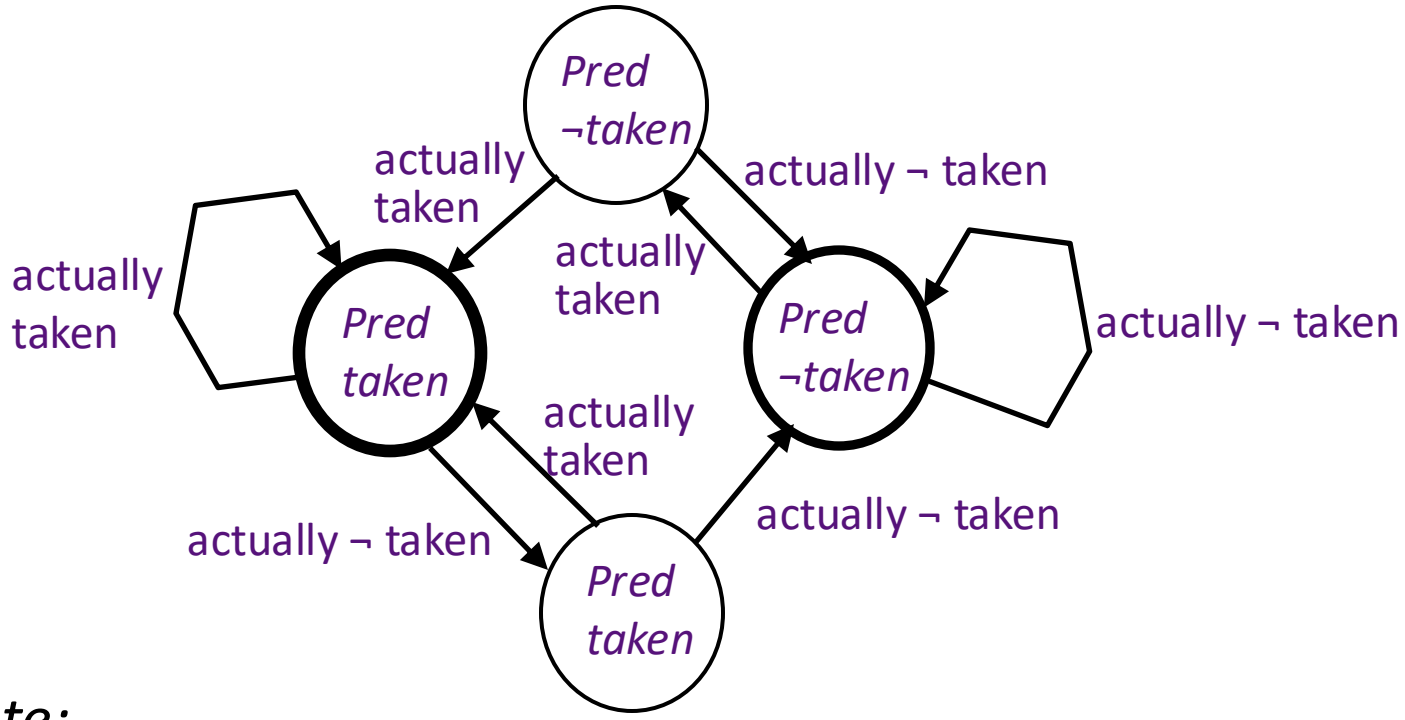
One-Bit Branch History Predictor



Predict same as last outcome

Branch Prediction Bits

- Assume 2 BP bits per instruction, 2-Bit Saturating Counter
- Change the prediction after two consecutive mistakes!

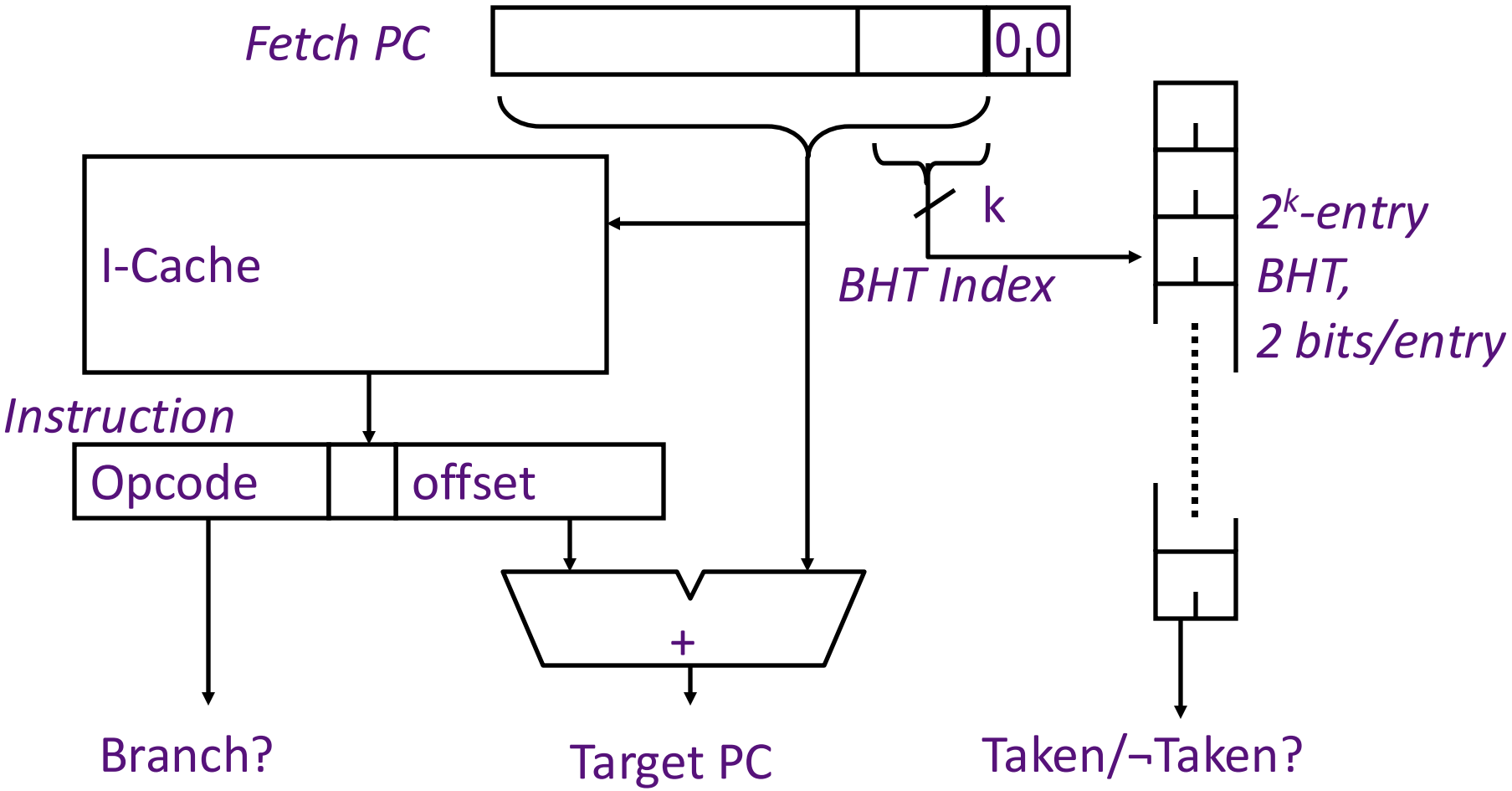


BP state:

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

Widely used in practice, e.g., MIPS R10000, four states named as *strongly taken, weakly taken, weakly not taken, strongly not taken*.

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

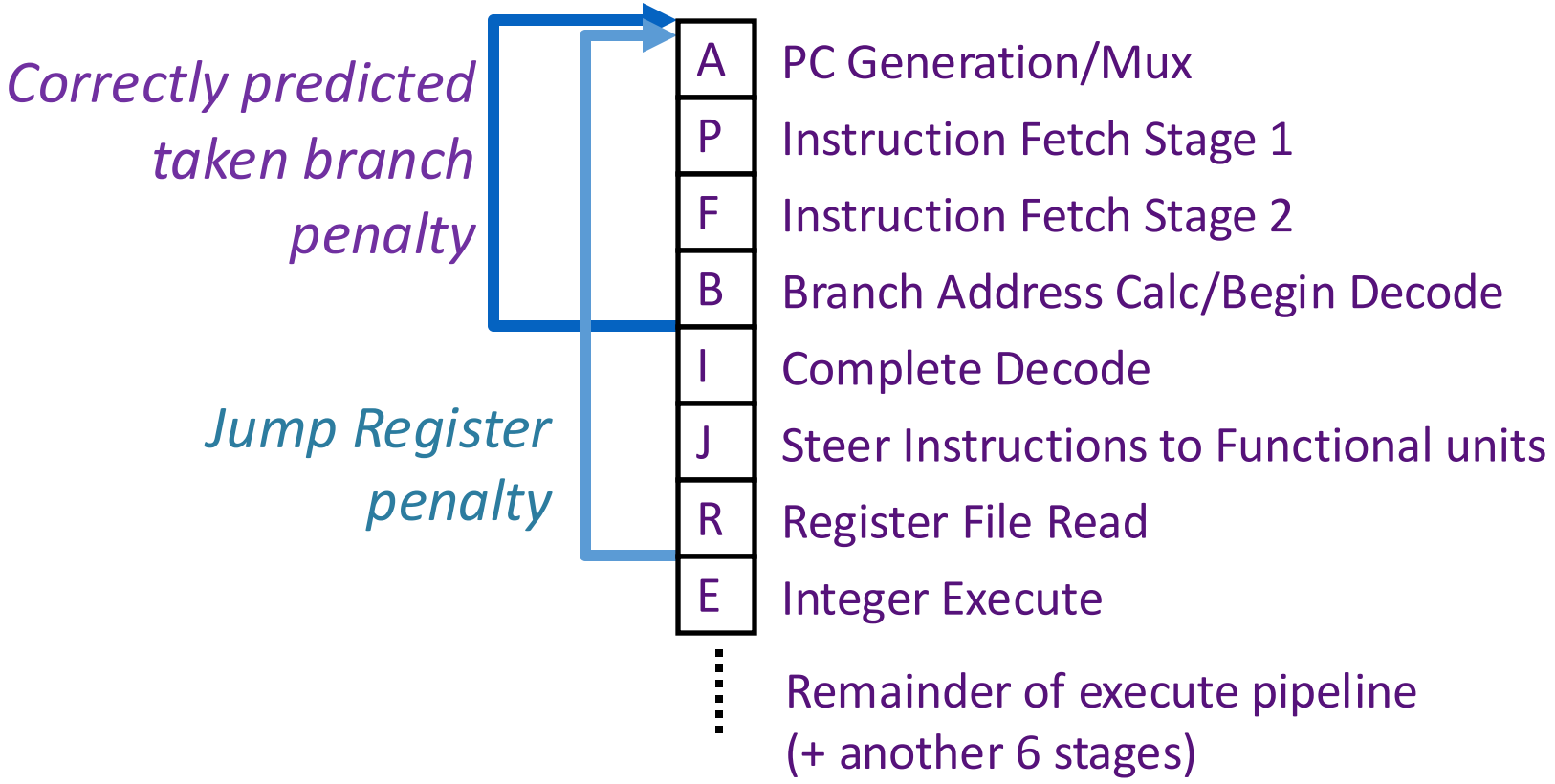
History register, H, records the direction of the last N branches executed by the processor

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!

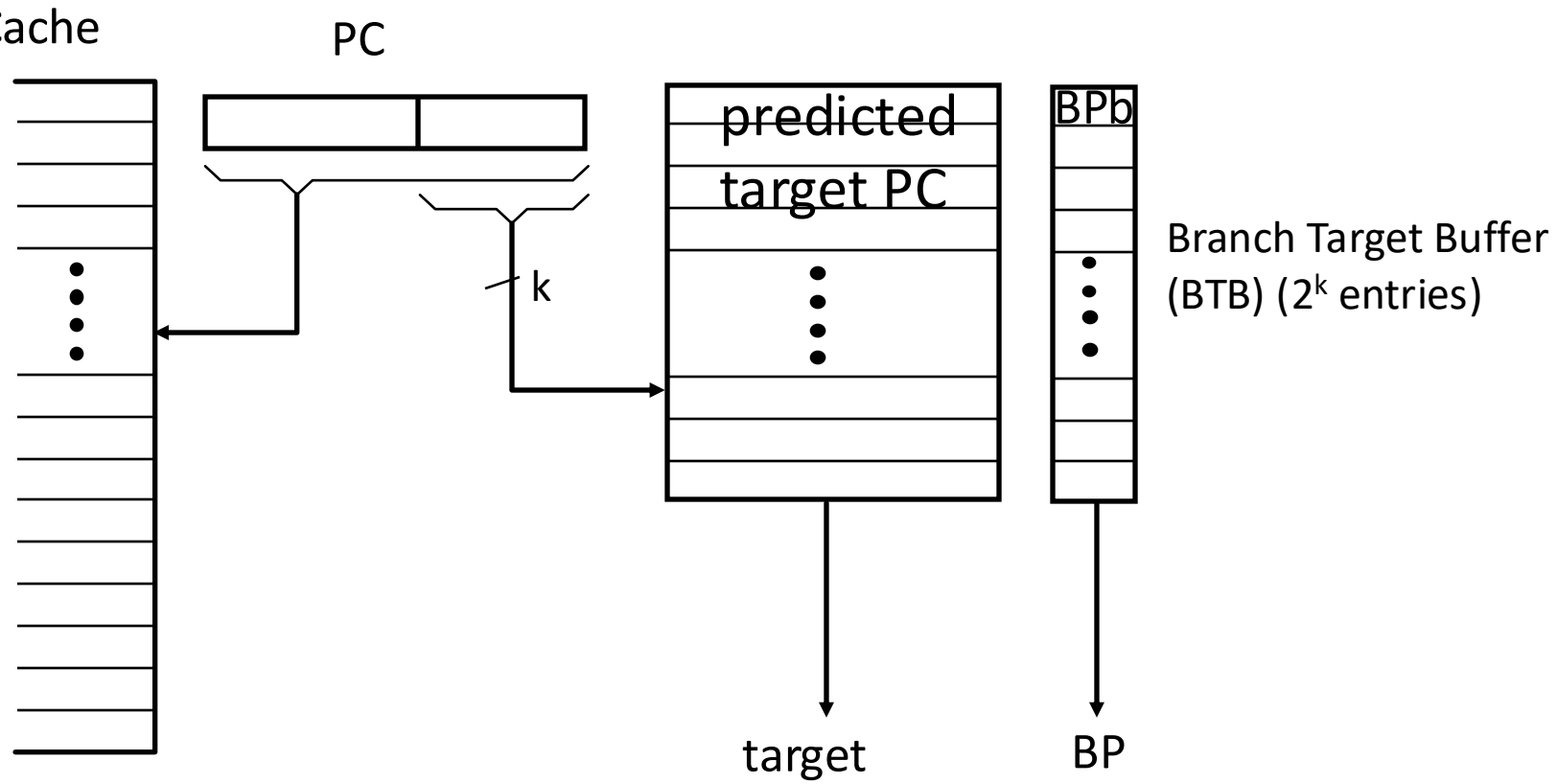
Limitations of BHTs

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



UltraSPARC-III fetch pipeline

Branch Target Buffer (untagged)



BP bits are stored with the predicted target address.

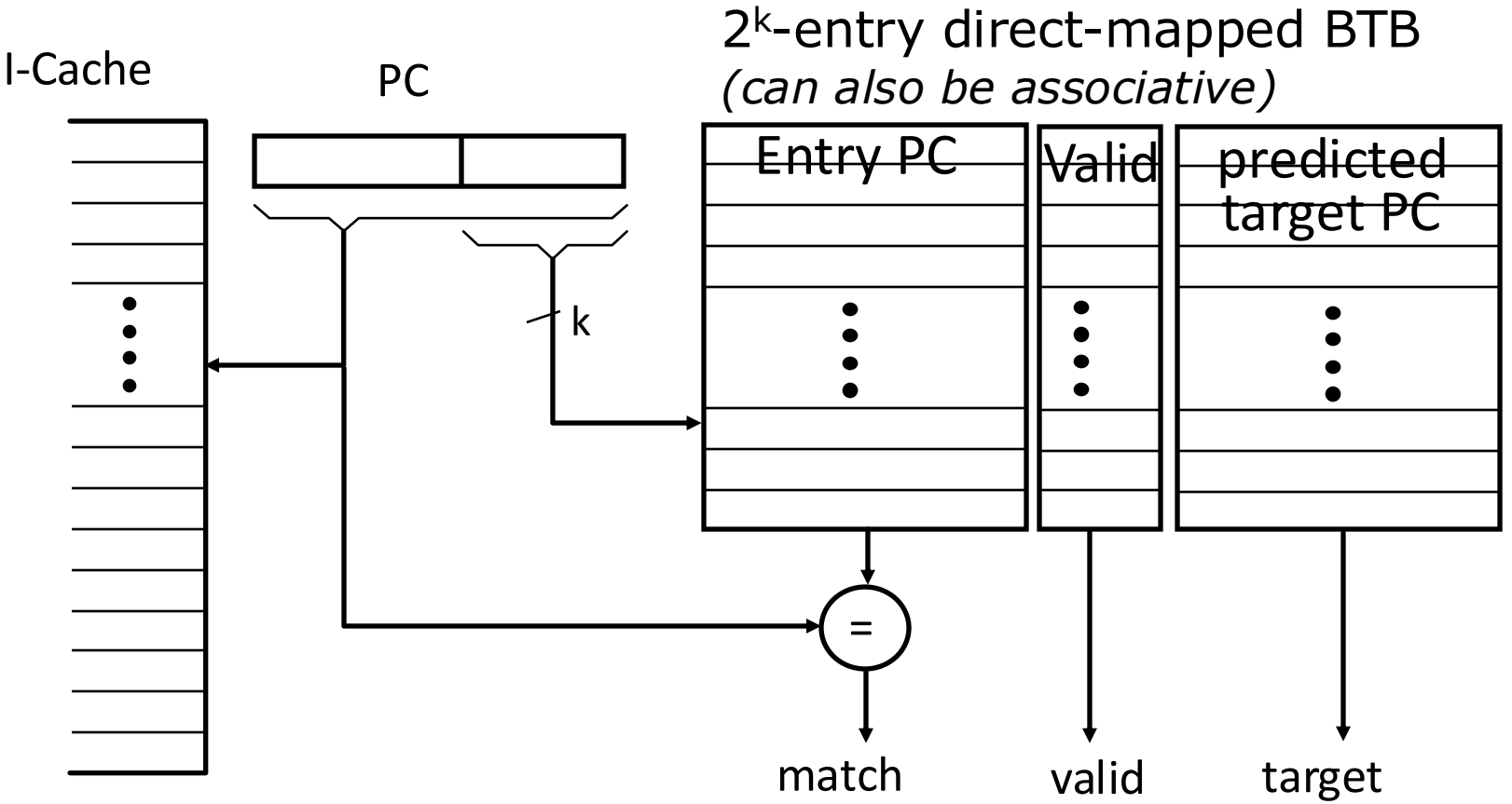
IF Stage: If (BP=taken) then next_PC=target; else next_PC=PC+4

Later: check prediction, if wrong then kill the instruction
and update BTB & BPb, else update BPb

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?*

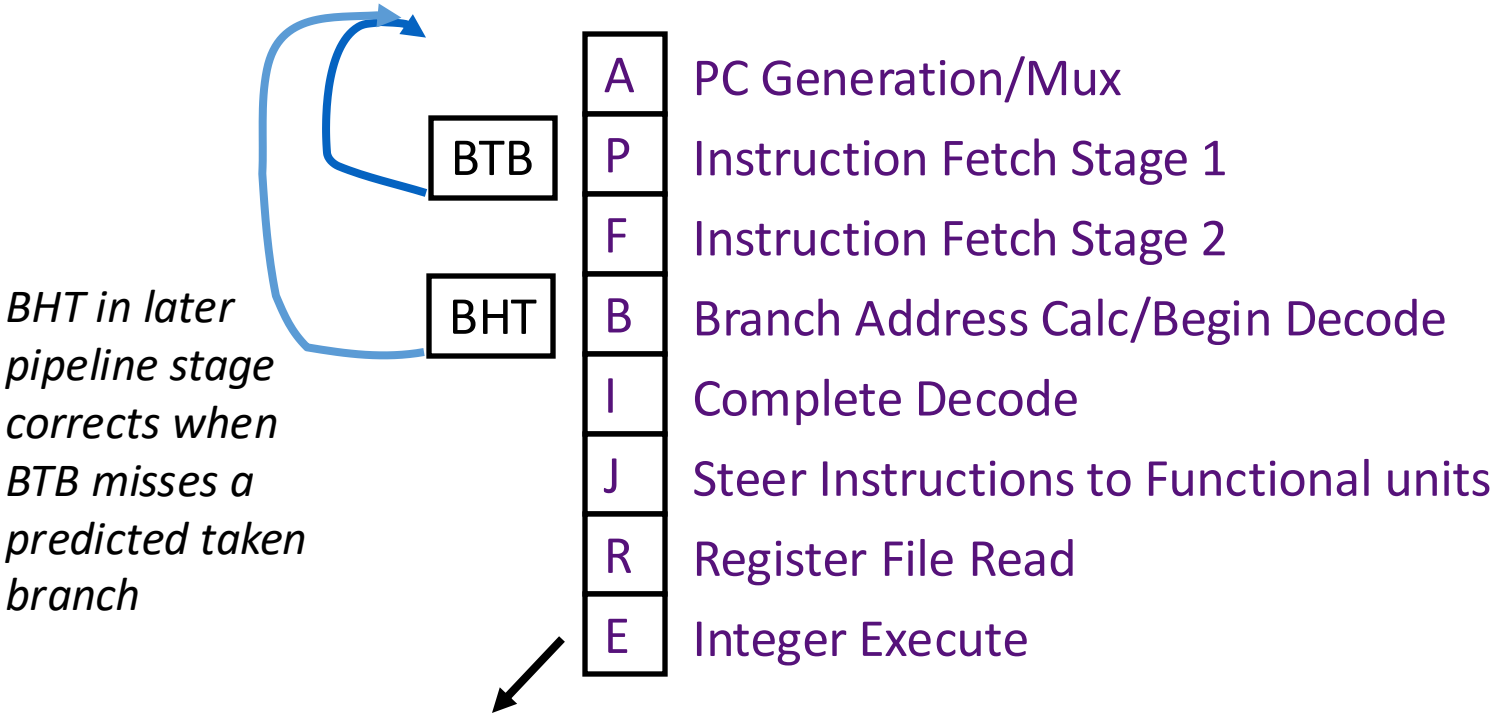
Branch Target Buffer (tagged)



- 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

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



BTB/BHT only updated after branch resolves in E stage

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

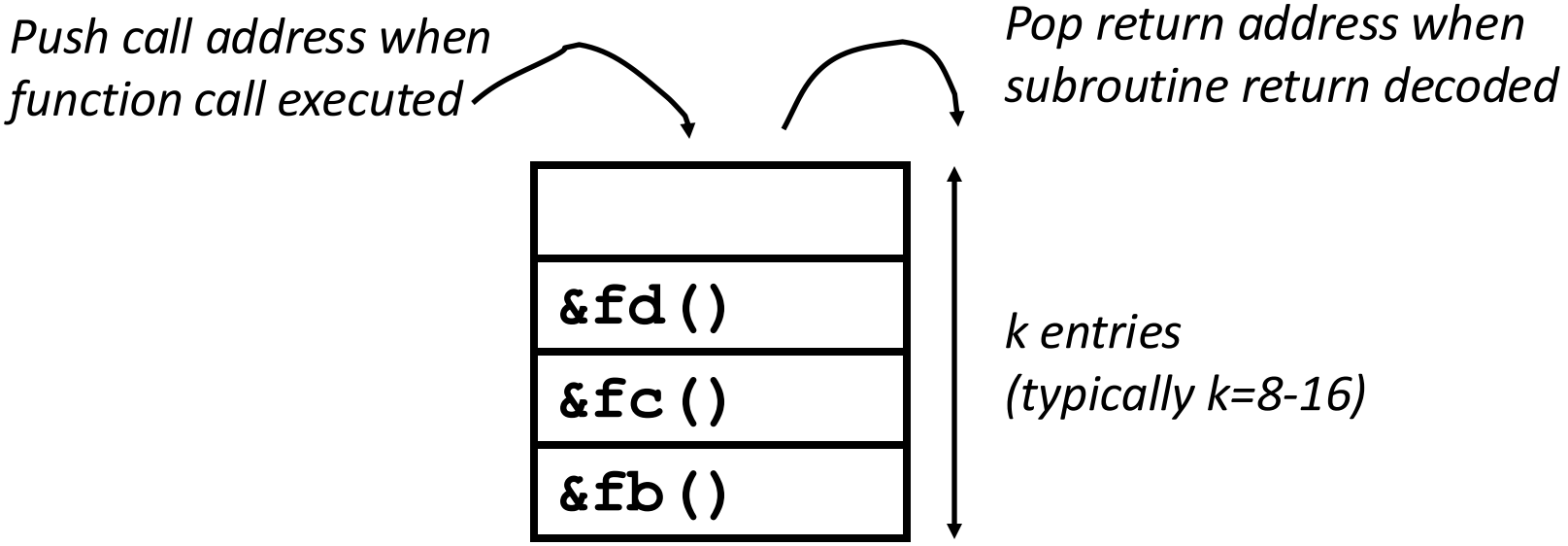
  ⇒ *Often one function called from many distinct call sites!*

  How well does BTB work for each of these cases?

Subroutine Return Stack

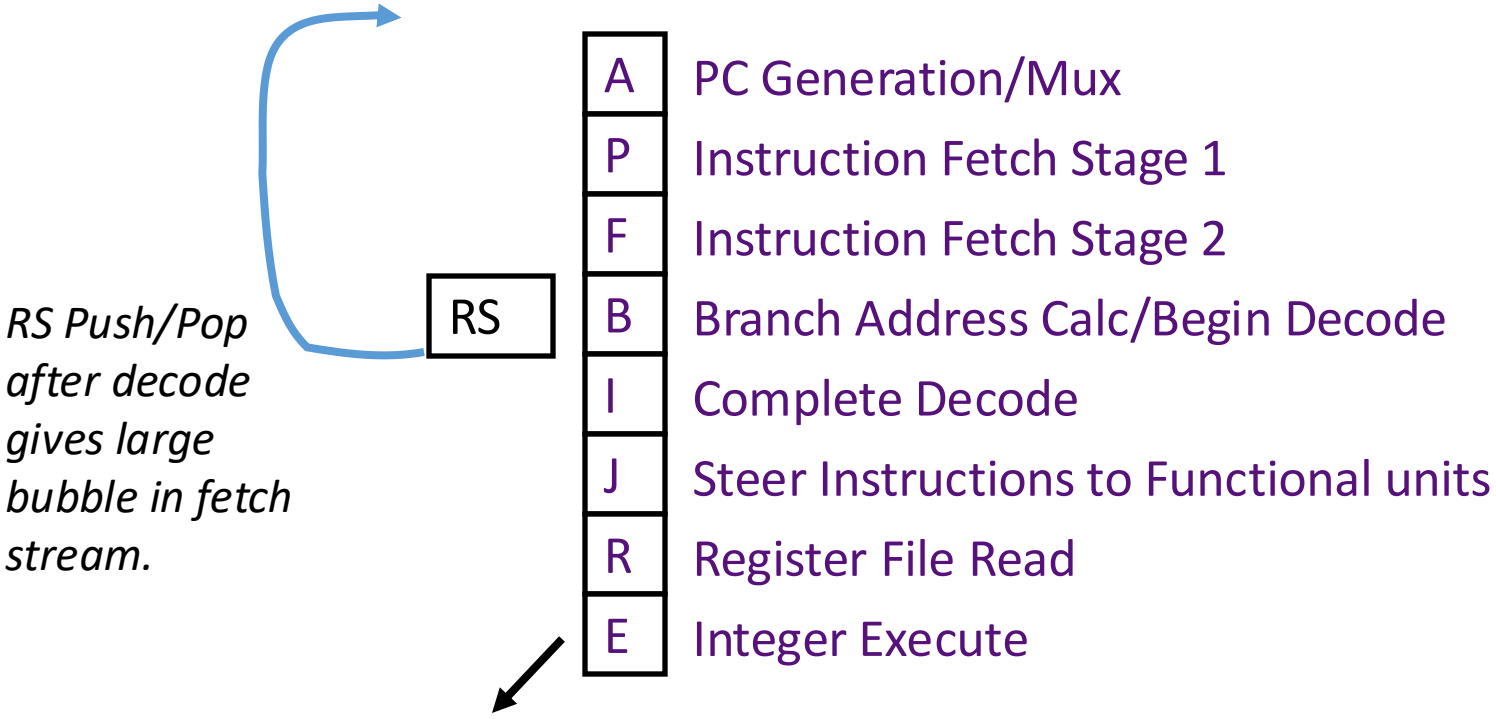
Small structure to accelerate JR for subroutine returns, typically much more accurate than BTBs.

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



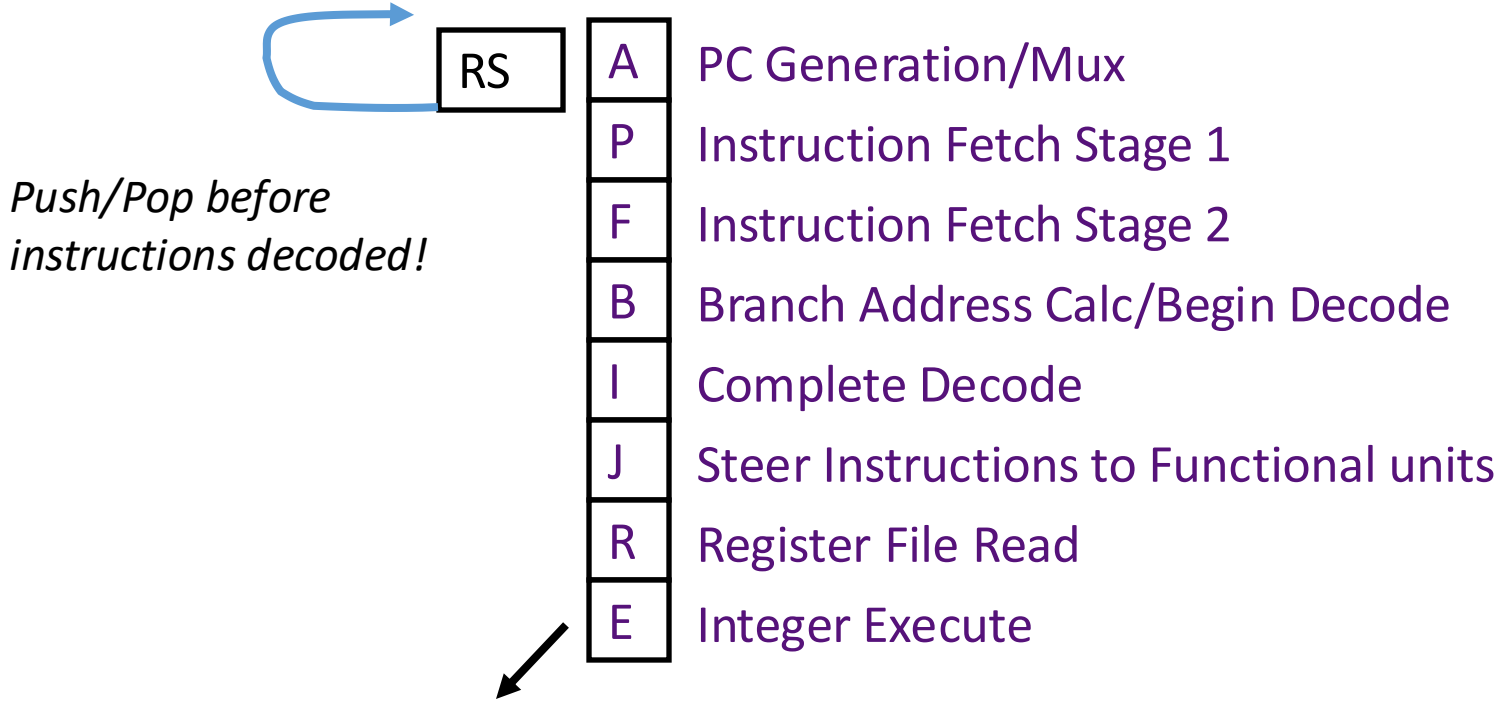
Return Stack in Pipeline

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

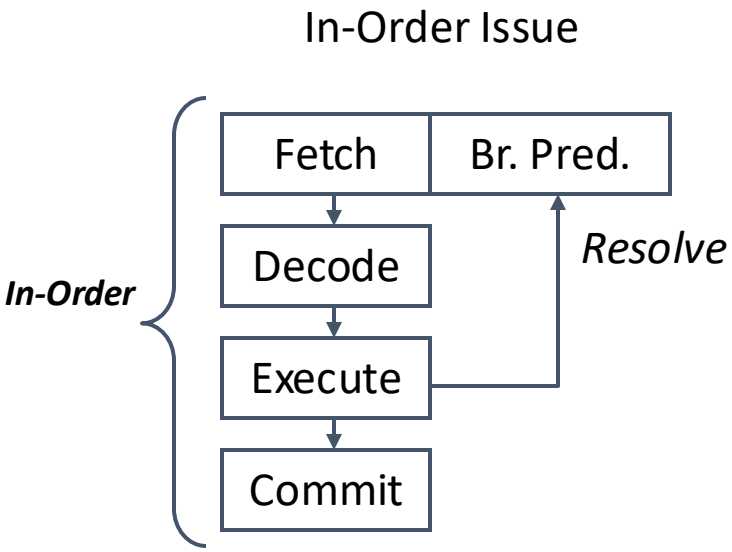


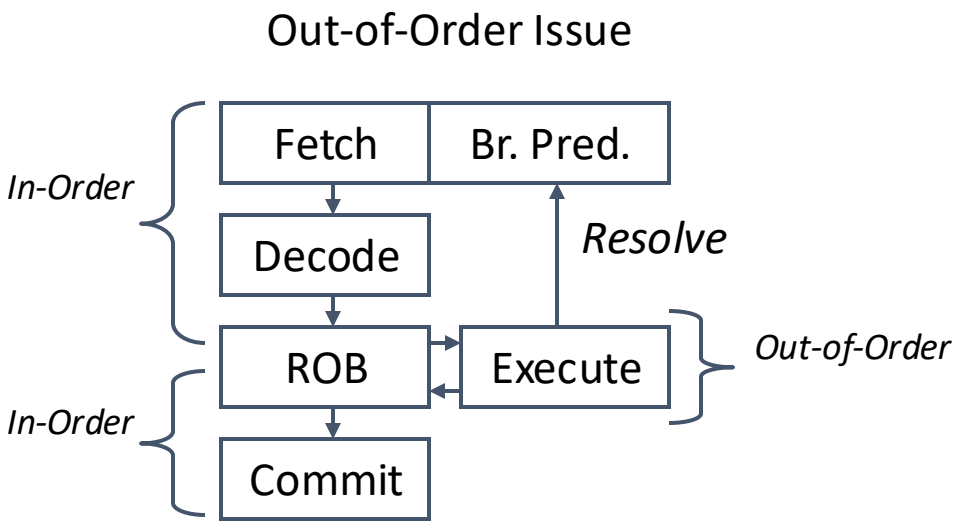
Return Stack in Pipeline

- 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

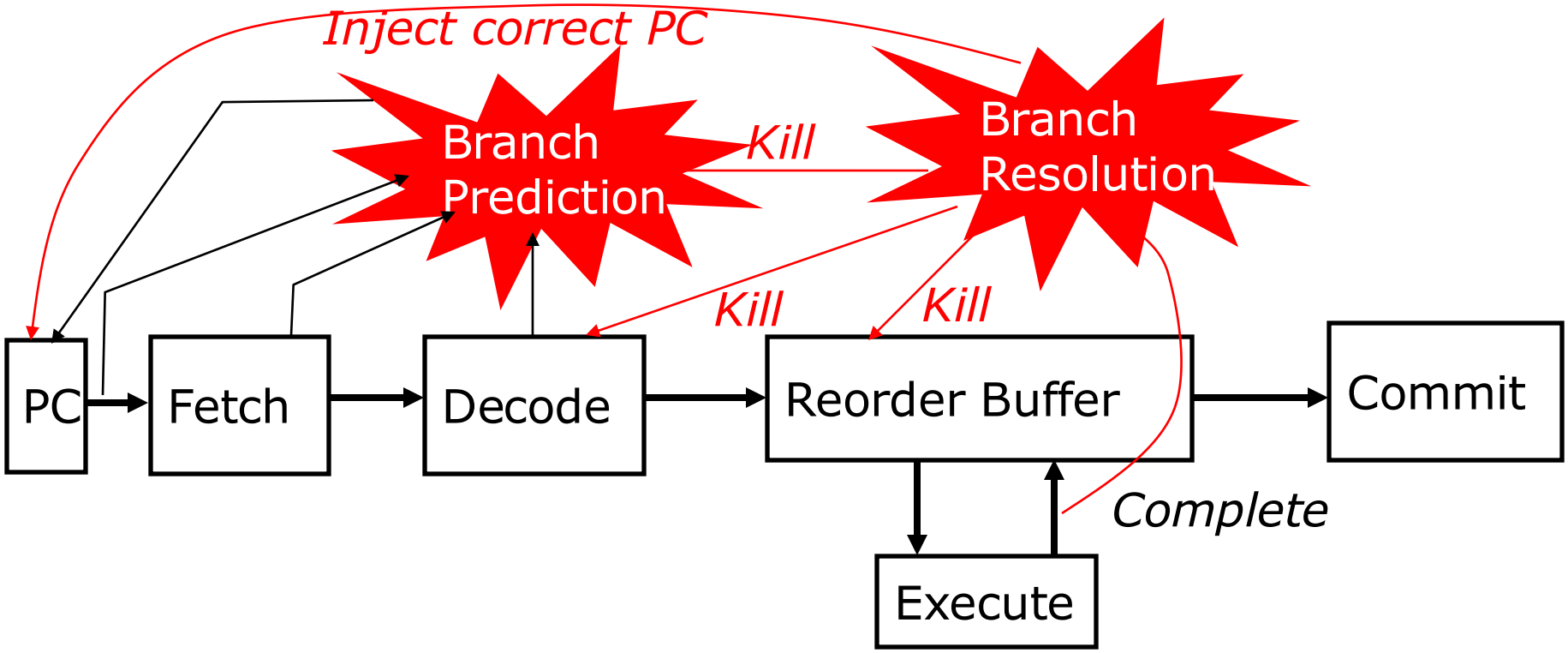
- Speculative execution, with branches resolved after later instructions complete
- Completed values held in rename registers in ROB or unified physical register file until commit

- 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 mispredictions
- MIPS R10K only has four snapshots for each of four outstanding speculative branches
- Alpha 21264 has 80 snapshots, one per ROB instruction

The R10000 processor can speculate up to four branches deep. *Shadow copies of the mapping tables* are kept every time a prediction is made, allowing the R10000 processor to recover from a mispredicted branch in a single cycle.
(MIPS R10000 Microprocessor User's Manual)

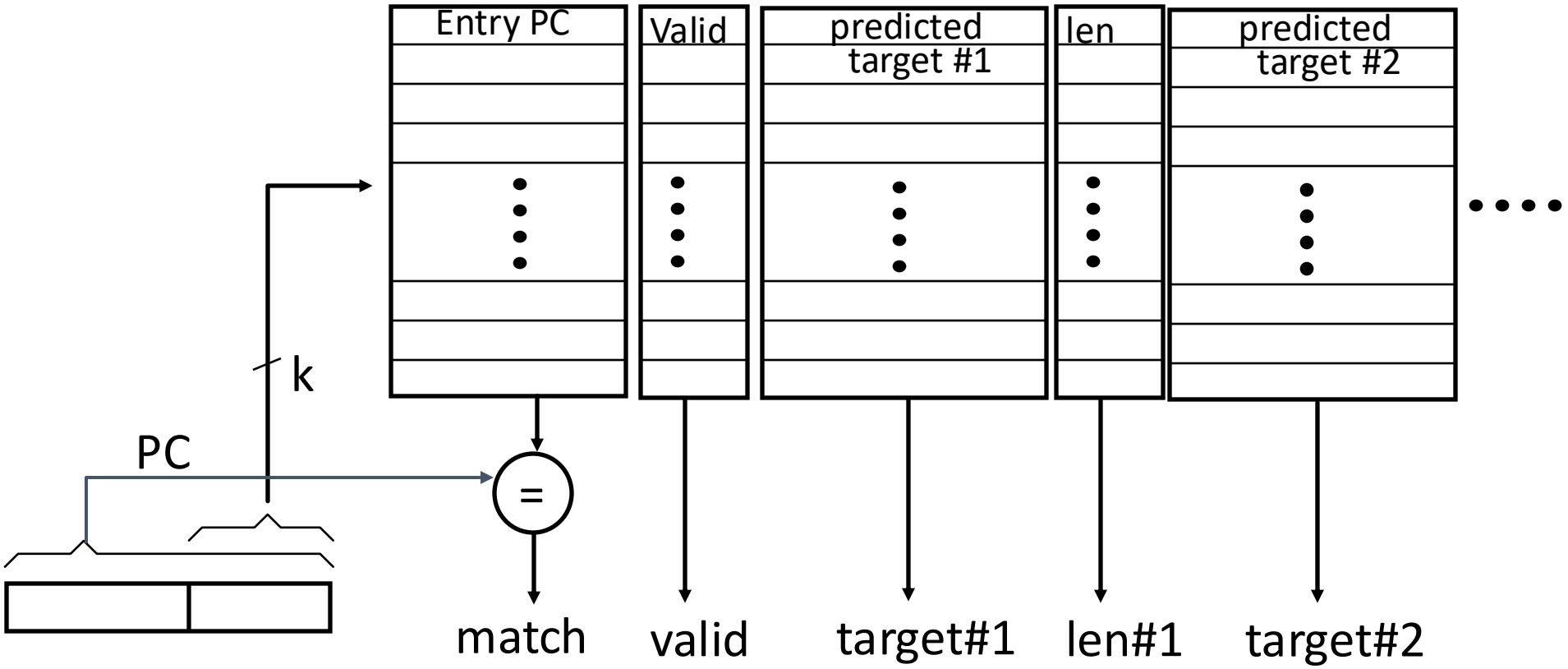
Improving Instruction Fetch

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

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

A mechanism for predicting multiple branches and fetching multiple non-consecutive basic blocks each cycle

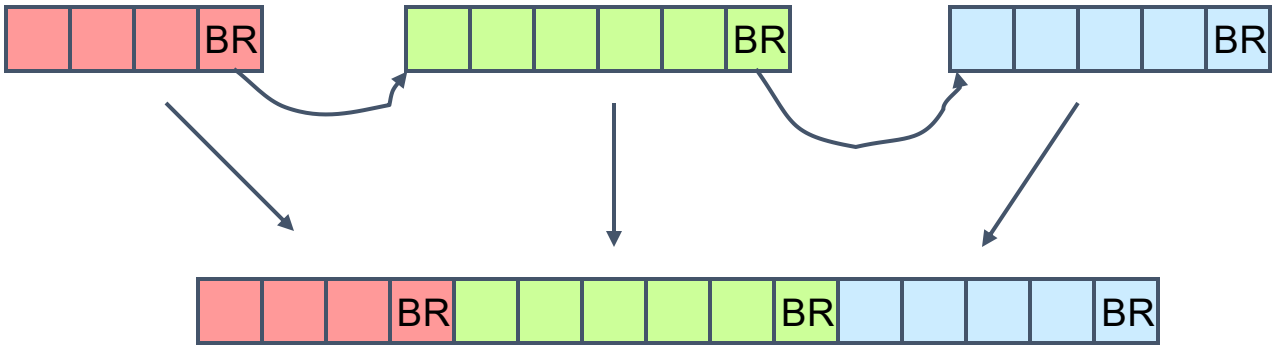
Source: https://hps.ece.utexas.edu/pub/yeh_ics7.pdf

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

Conclusion

- Static Branch prediction
- Dynamic Branch prediction

Acknowledgements

- These slides contain materials developed and copyright by:
  - Prof. Krste Asanovic (UC Berkeley)
  - Prof. James C. Hoe (CMU)
  - Prof. Daniel Sanchez (MIT)
  - Prof. Nima Honarmand (SUNY)
  - Prof. Onur Mutlu (ETHZ)