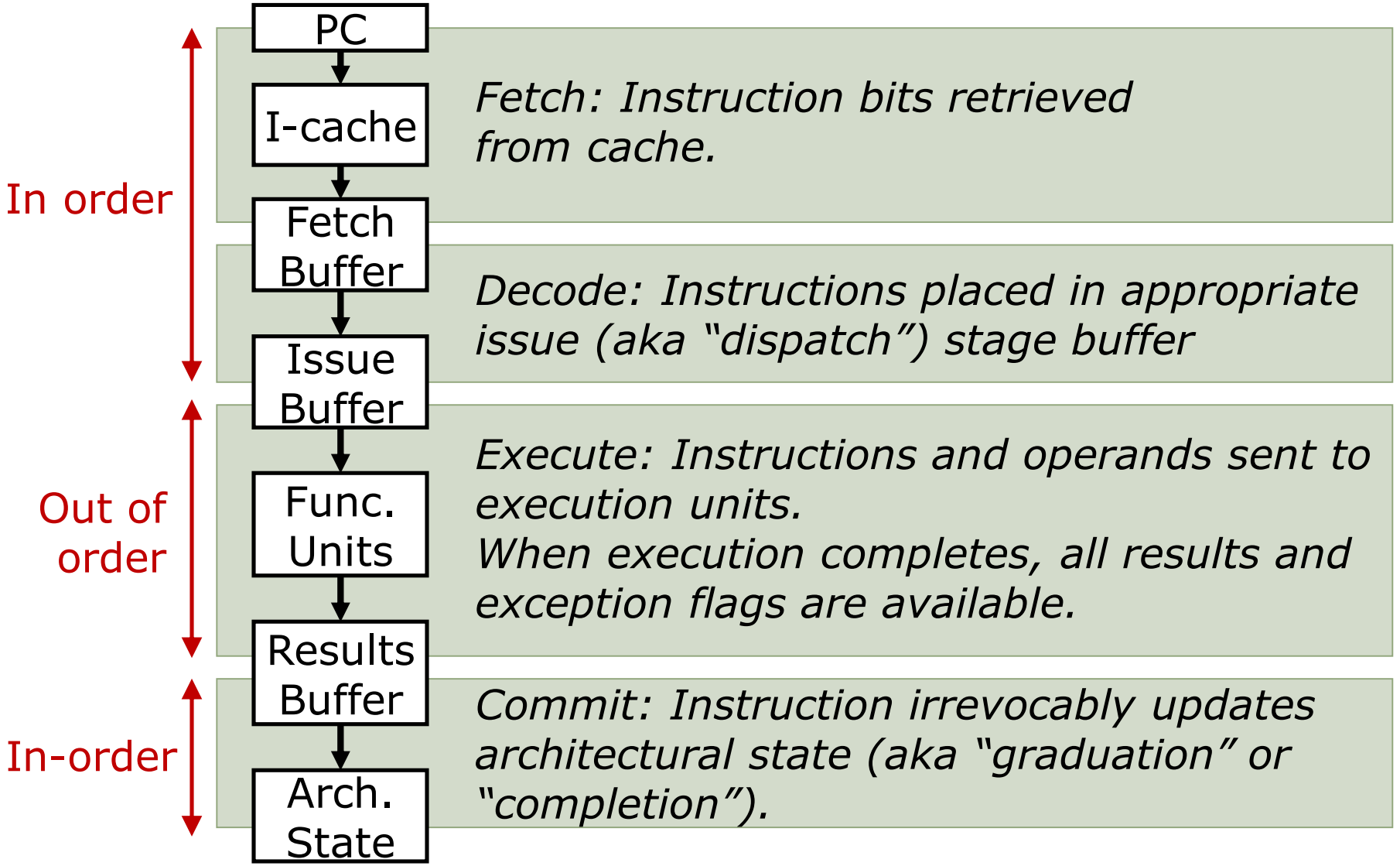
Branch Prediction

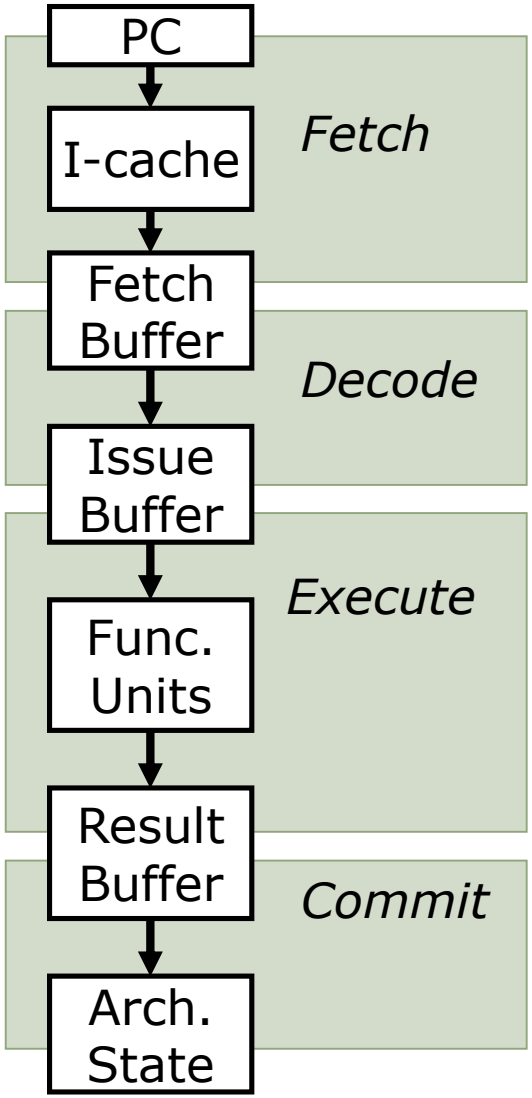
Daniel Sanchez
Computer Science and Artificial Intelligence Laboratory
M.I.T.

Reminder: Phases of Instruction Execution



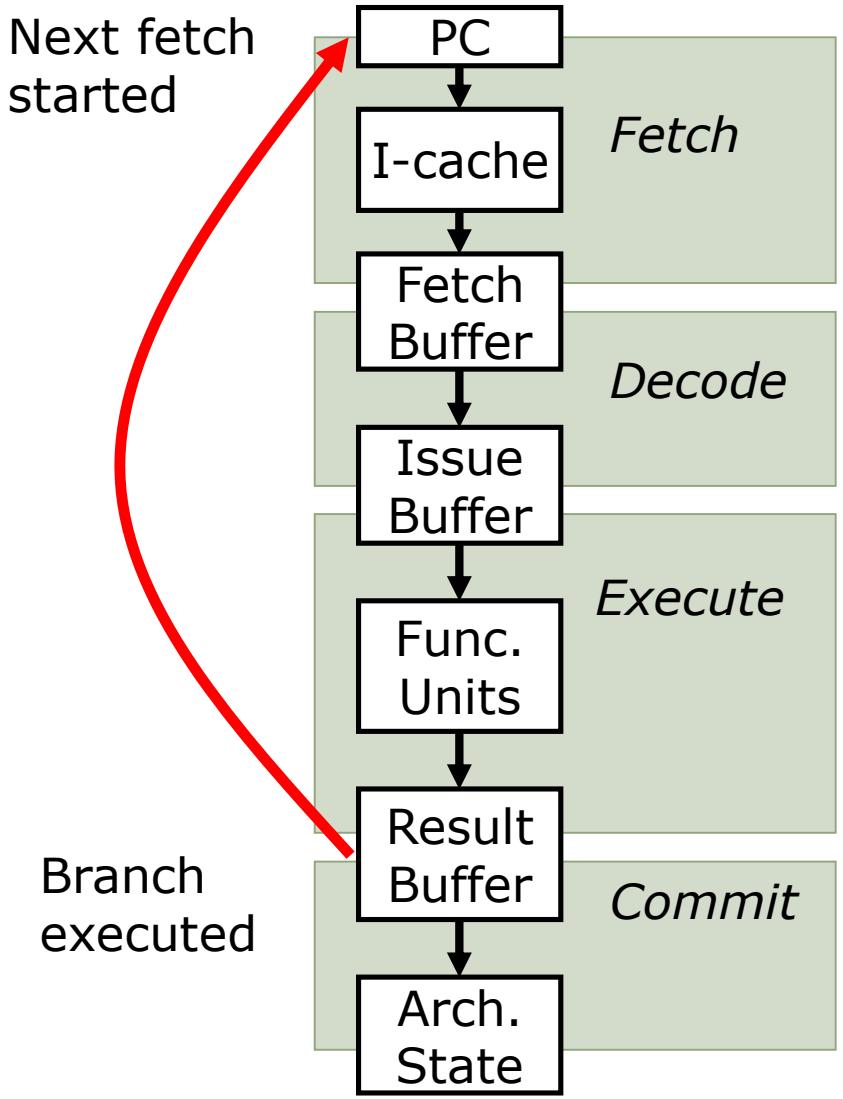
Control Flow Penalty

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



Control Flow Penalty

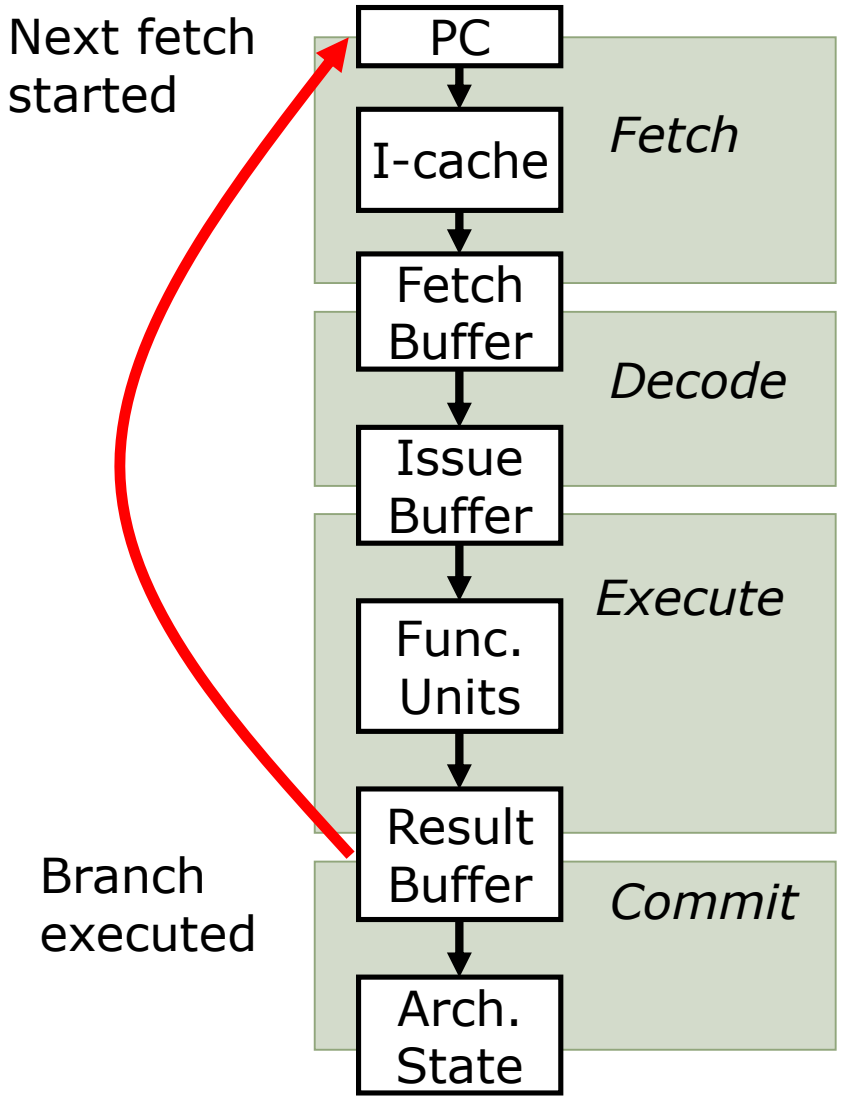
Modern processors may have > 10 pipeline stages between next PC calculation and branch resolution!



Control Flow Penalty

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

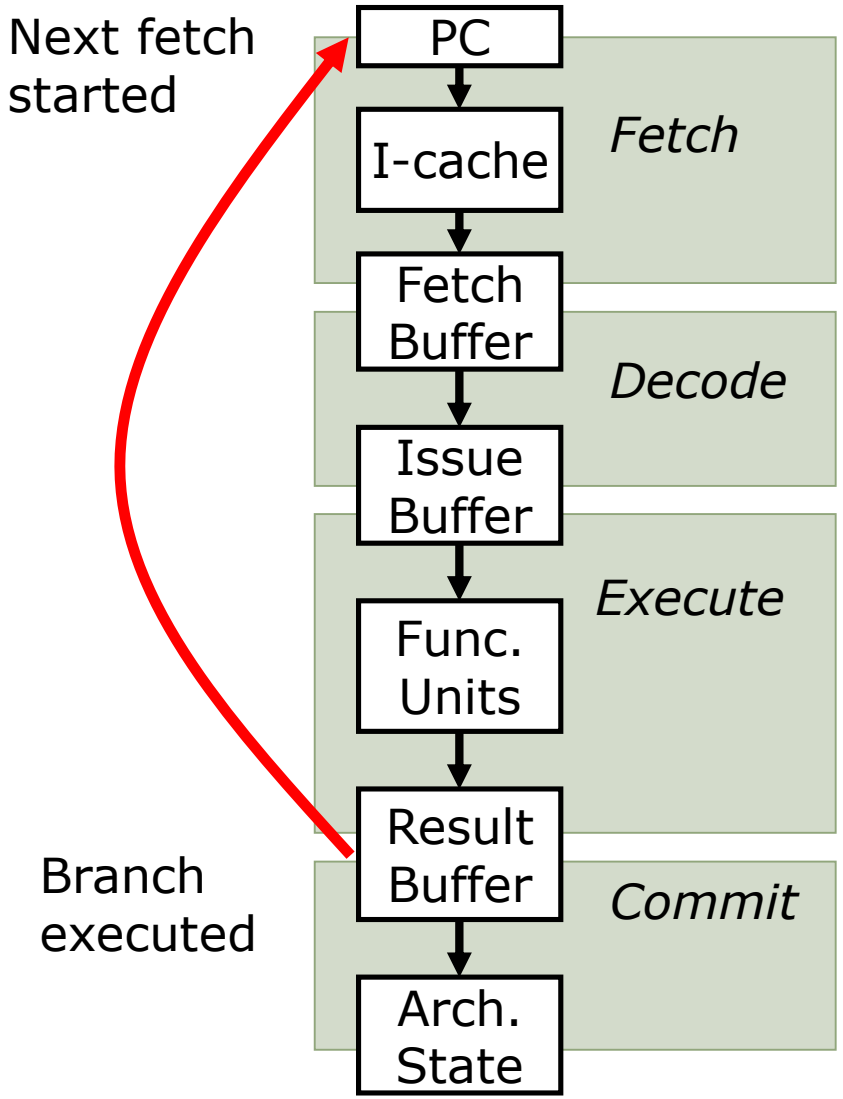
How much work is lost if pipeline doesn't follow correct instruction flow?



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?

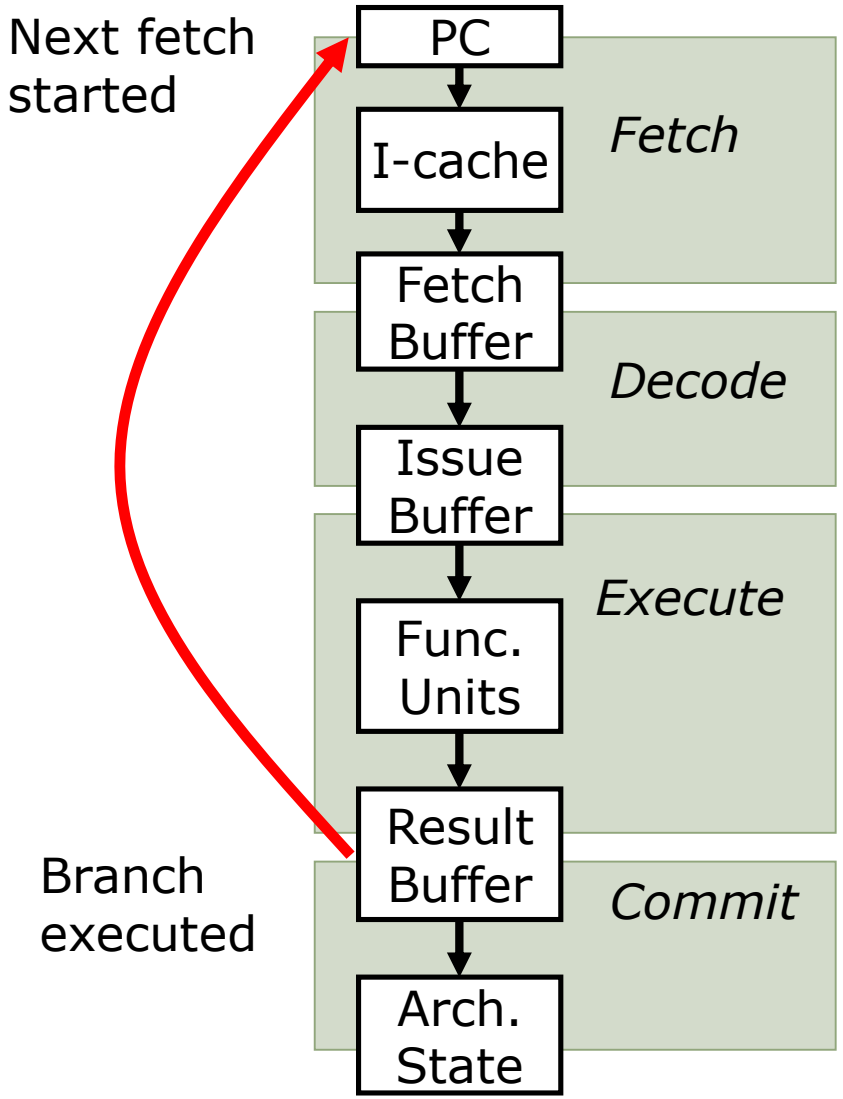


Control Flow Penalty

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

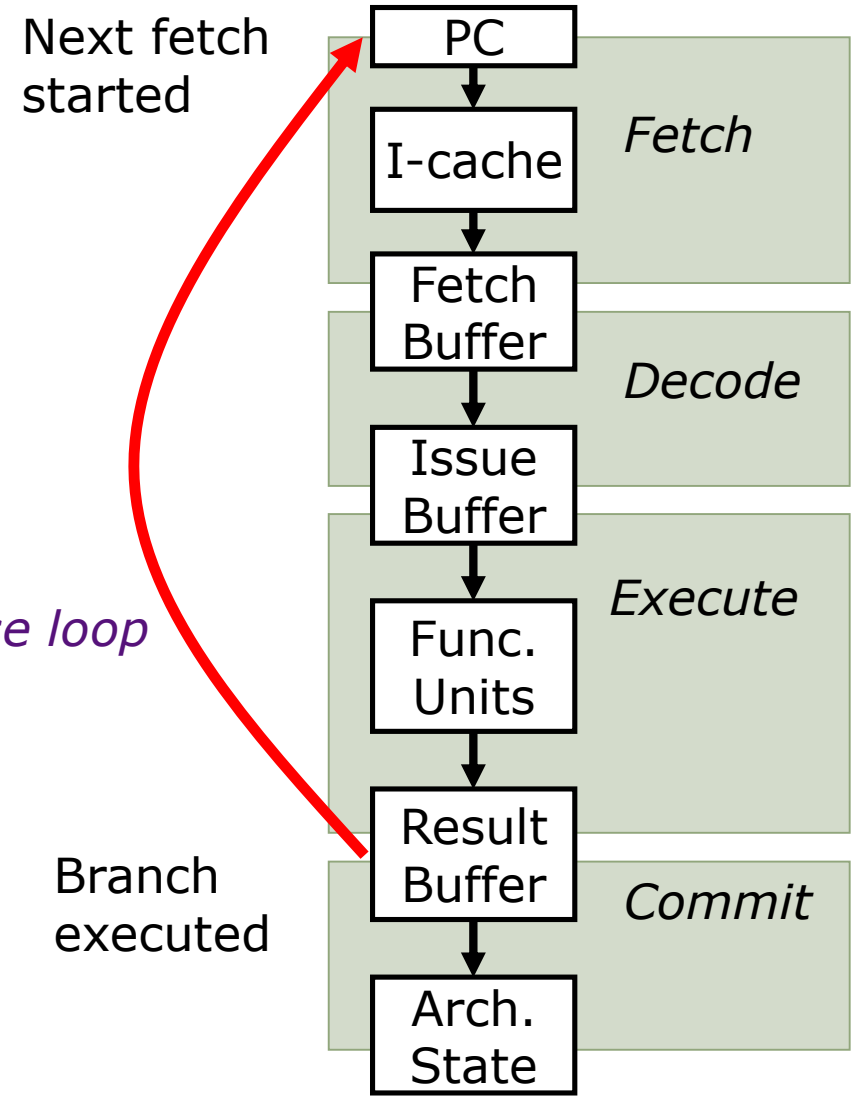


Control Flow Penalty

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 of SPEC CPU 2017 [Limaye and Adegbiya, ISPASS'18]:

	SPECint	SPECfp
Branches	19 %	11 %
Loads	24 %	26 %
Stores	10 %	7 %
Other	47 %	56 %

SPECint17: *perlbench, gcc, mcf, omnetpp, xalancbmk, x264, deepsjeng, leela, exchange2, xz*

SPECfp17: *bwaves, cactus, lbm, wrf, pop2, imagick, nab, fotonik3d, roms*

What is the average run length *between branches?*

Average Run-Length between Branches

Average dynamic instruction mix of SPEC CPU 2017 [Limaye and Adegbiya, ISPASS'18]:

	SPECint	SPECfp
Branches	19 %	11 %
Loads	24 %	26 %
Stores	10 %	7 %
Other	47 %	56 %

SPECint17: *perlbench, gcc, mcf, omnetpp, xalancbmk, x264, deepsjeng, leela, exchange2, xz*

SPECfp17: *bwaves, cactus, lbm, wrf, pop2, imagick, nab, fotonik3d, roms*

What is the average run length *between branches?*

Roughly 5-10 instructions

MIPS 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?

Instruction	*Taken known?*	*Target known?*
J		
JR		
BEQZ/BNEZ		

MIPS 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?

Instruction	*Taken known?*	*Target known?*
J	After Inst. Decode	
JR		
BEQZ/BNEZ		

MIPS 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?

Instruction	*Taken known?*	*Target known?*
J	After Inst. Decode	After Inst. Decode
JR		
BEQZ/BNEZ		

MIPS 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?

Instruction	*Taken known?*	*Target known?*
J	After Inst. Decode	After Inst. Decode
JR	After Inst. Decode	
BEQZ/BNEZ		

MIPS 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?

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

MIPS 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?

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

*Assuming zero detect on register read

MIPS 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?

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

*Assuming zero detect on register read

MIPS 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?

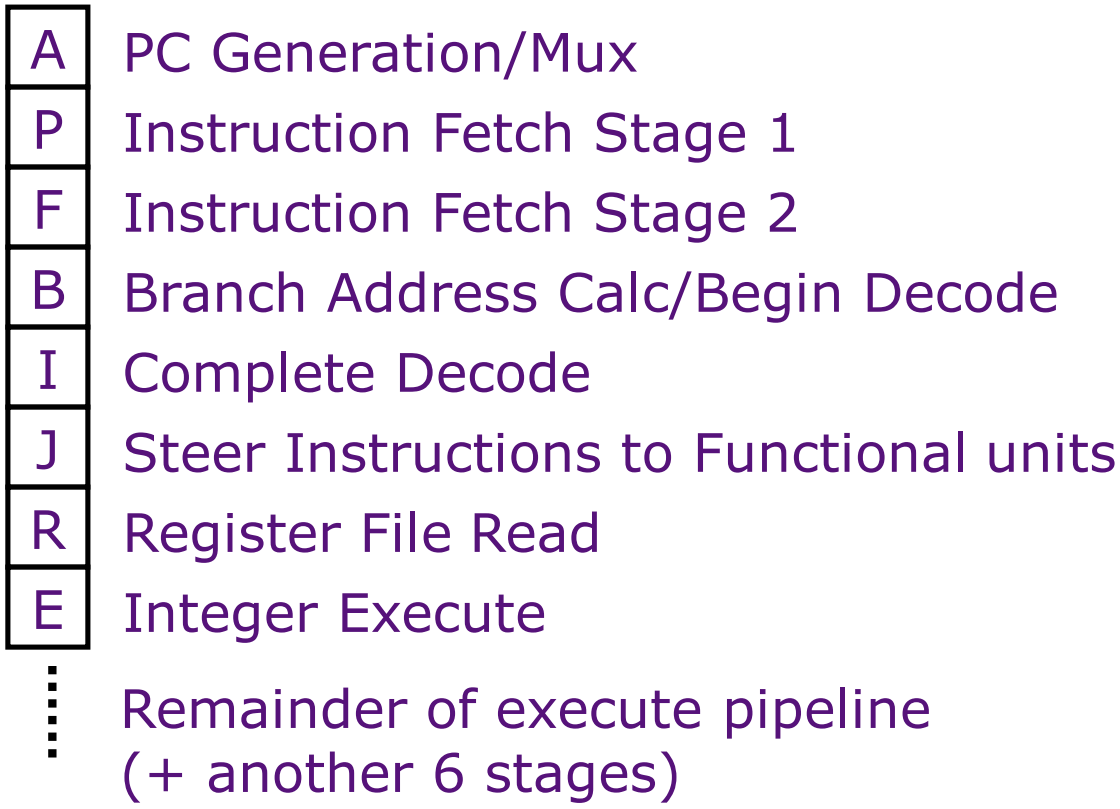
2) If so, what is the target address?

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

*Assuming zero detect on register read

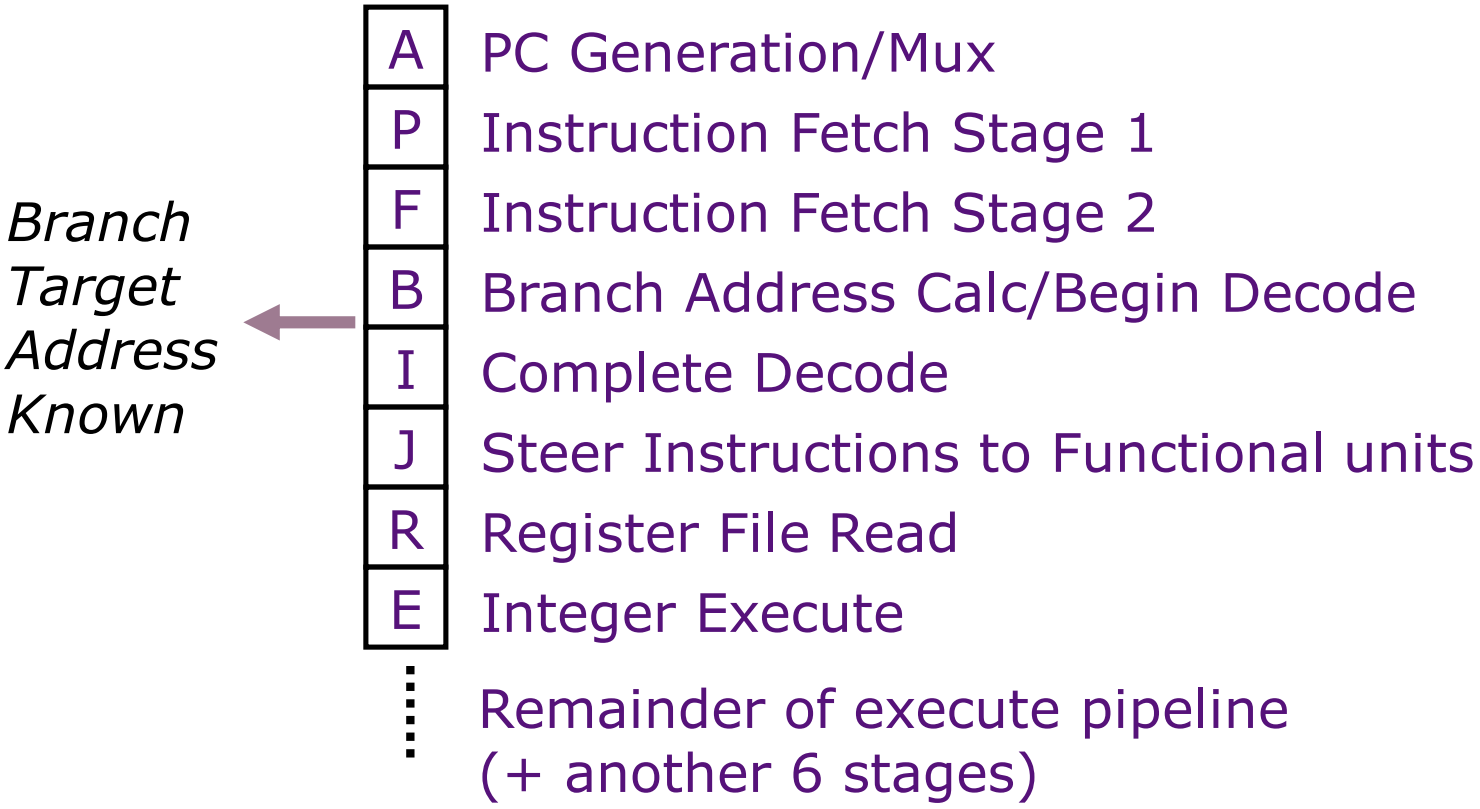
Example Branch Penalties

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



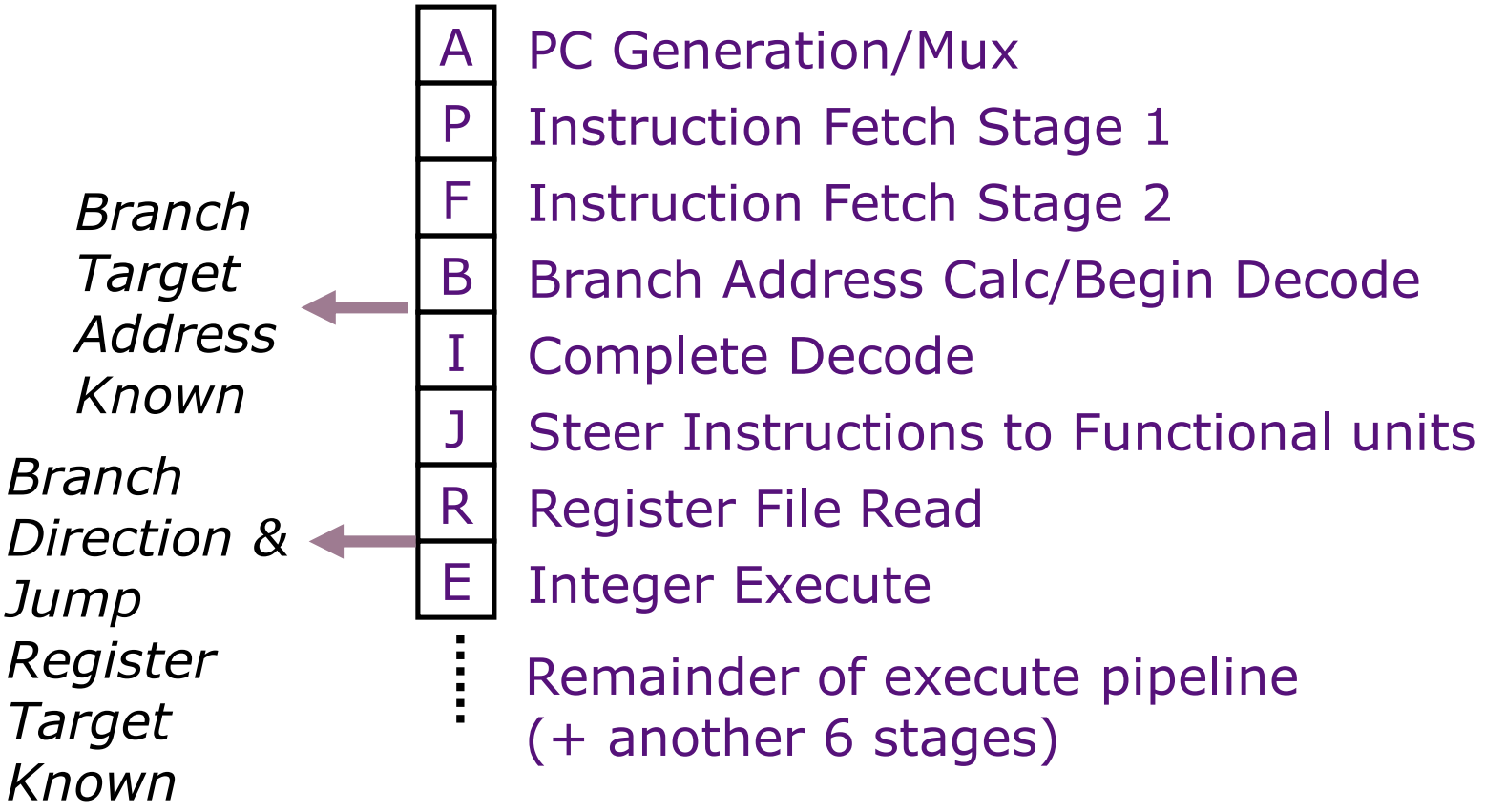
Example Branch Penalties

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



Example Branch Penalties

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



Reducing Control Flow Penalty

- Software solutions

- Hardware solutions

Reducing Control Flow Penalty

- Software solutions
  - *Eliminate branches – loop unrolling*

    Increases run length between branches

- Hardware solutions

Reducing Control Flow Penalty

- Software solutions
  - *Eliminate branches – loop unrolling*

    Increases run length between branches

- Hardware solutions

Reducing Control Flow Penalty

- Software solutions
  - *Eliminate branches – loop unrolling*

    Increases run length between branches
  - *Reduce resolution time – instruction scheduling*

    Compute the branch condition as early as possible (of limited value)
- Hardware solutions

Reducing Control Flow Penalty

- Software solutions
  - *Eliminate branches – loop unrolling*

    Increases run length between branches
  - *Reduce resolution time – instruction scheduling*

    Compute the branch condition as early as possible (of limited value)
- Hardware solutions
  - Bypass – usually results are used immediately

Reducing Control Flow Penalty

- Software solutions
  - *Eliminate branches – loop unrolling*

    Increases run length between branches
  - *Reduce resolution time – instruction scheduling*

    Compute the branch condition as early as possible (of limited value)

- Hardware solutions
  - Bypass – usually results are used immediately
  - Change architecture – find something else to do

    Delay slots – replace pipeline bubbles with useful work (requires software cooperation)

Reducing Control Flow Penalty

- Software solutions
  - *Eliminate branches – loop unrolling*

    Increases run length between branches
  - *Reduce resolution time – instruction scheduling*

    Compute the branch condition as early as possible (of limited value)

- Hardware solutions
  - Bypass – usually results are used immediately
  - Change architecture – find something else to do

    Delay slots – replace pipeline bubbles with useful work (requires software cooperation)
  - *Speculate – branch prediction*

    Speculative execution of instructions beyond the branch

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.

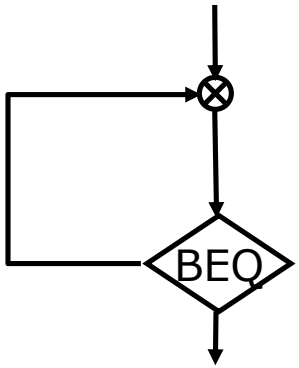
Mispredict recovery mechanisms:

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

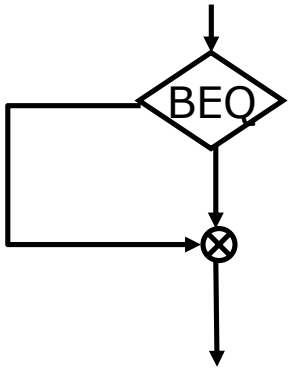
Static Branch Prediction

Overall probability a branch is taken is ~60-70% but:

backward 90%



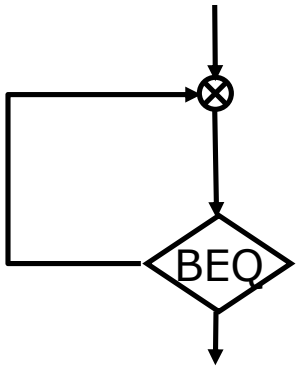
forward 50%



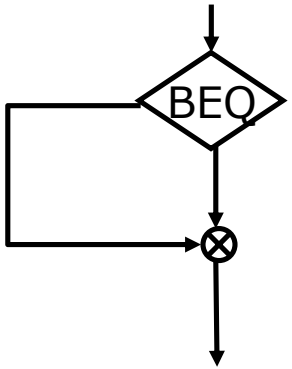
Static Branch Prediction

Overall probability a branch is taken is ~60-70% but:

backward 90%



forward 50%

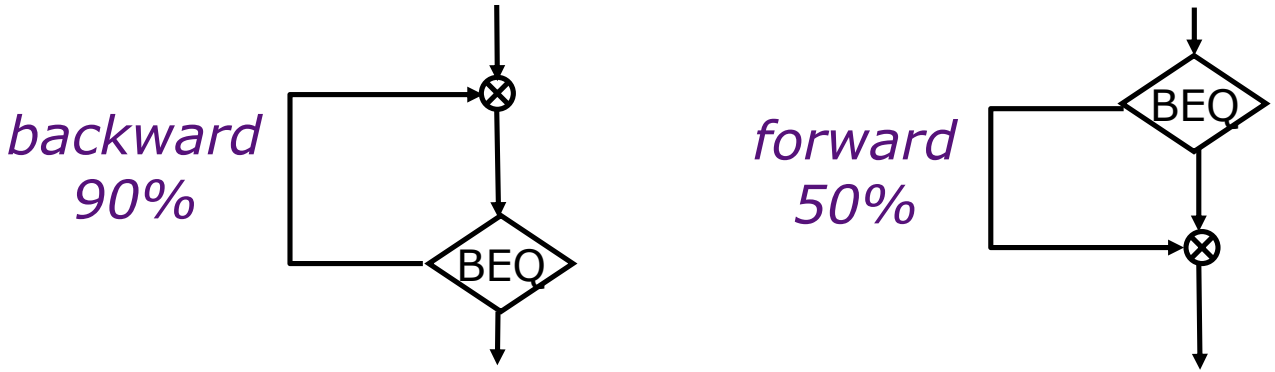


ISA can attach preferred direction semantics to branches, e.g., Motorola MC88110

bne0 *(preferred taken)* beq0 *(not taken)*

Static Branch Prediction

Overall probability a branch is taken is ~60-70% but:



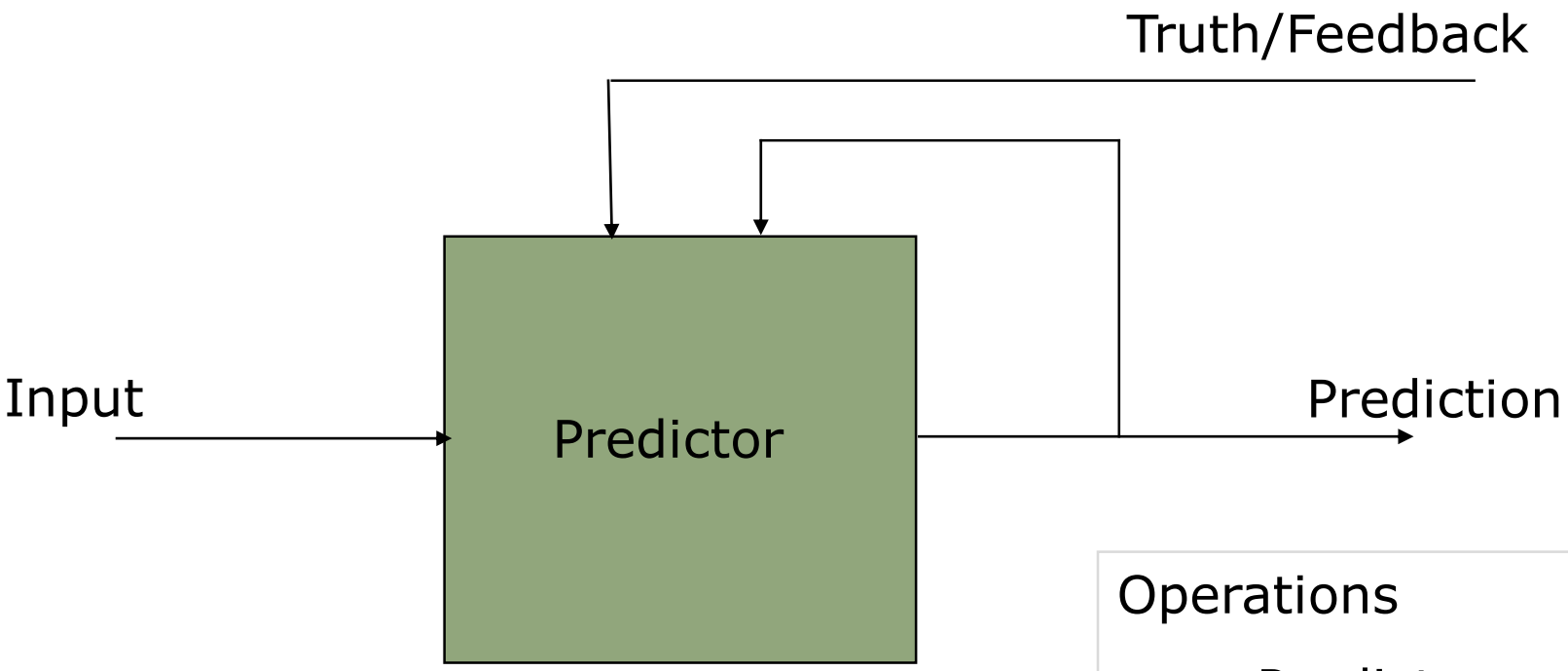
ISA can attach preferred direction semantics to branches, e.g., Motorola MC88110

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 Prediction



Operations

- Predict
- Update

Prediction as a feedback control process

Dynamic Branch Prediction
Learning based on past behavior

Temporal correlation

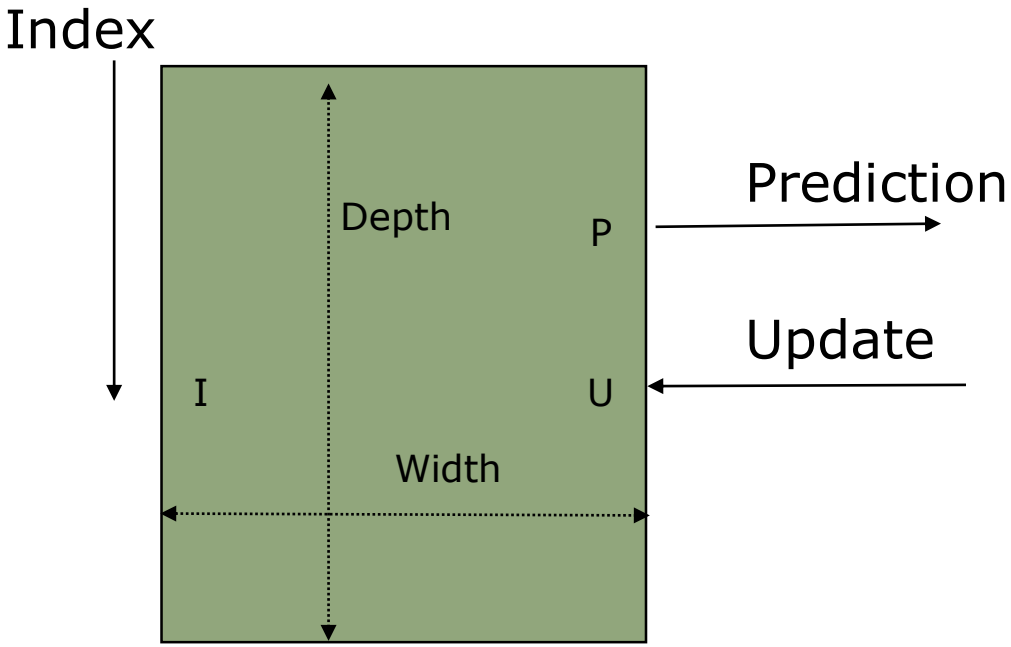
The way a branch resolves may be a good predictor of the way it will resolve at the next execution

Spatial correlation

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

Predictor Primitive
Emer & Gloy, 1997

- Indexed table holding values
- Operations
  - Predict
  - Update

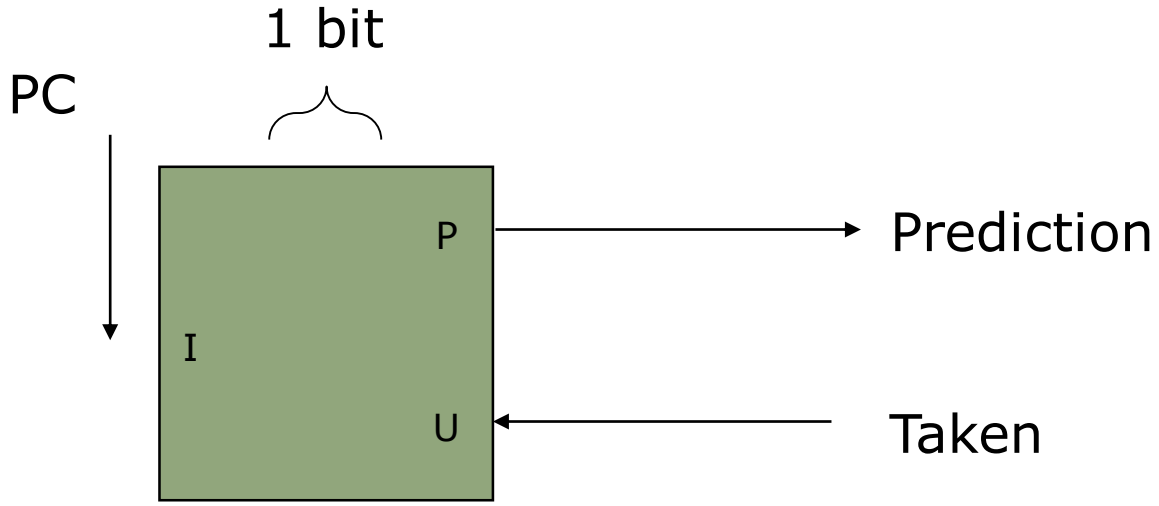


- Algebraic notation

Prediction = P[Width, Depth](Index; Update)

One-bit Predictor
aka Branch History Table (BHT)

Simple temporal prediction

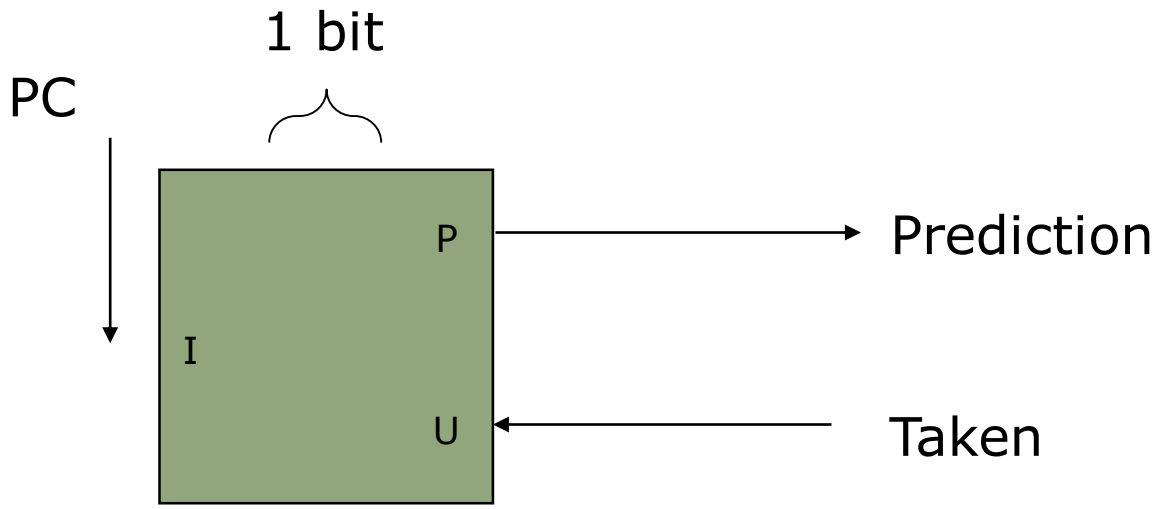


A21064(PC; T) = P[1, 2K](PC; T)

What happens on loop branches?

One-bit Predictor
aka Branch History Table (BHT)

Simple temporal prediction



A21064(PC; T) = P[1, 2K](PC; T)

What happens on loop branches?

At best, mispredicts twice for every use of loop

Two-bit Predictor
Smith, 1981

- Use two bits per entry instead of one bit
- Manage them as a saturating counter:

On not-taken ↓	↑ On taken			
		1	1	Strongly taken
		1	0	Weakly taken
		0	1	Weakly not-taken
		0	0	Strongly not-taken

  - Direction prediction changes only after two wrong predictions

Two-bit Predictor
Smith, 1981

- Use two bits per entry instead of one bit
- Manage them as a saturating counter:

On not-taken ↓	↑ On taken			
		1	1	Strongly taken
		1	0	Weakly taken
		0	1	Weakly not-taken
		0	0	Strongly not-taken

  - Direction prediction changes only after two wrong predictions

How many mispredictions per loop? ___

Two-bit Predictor
Smith, 1981

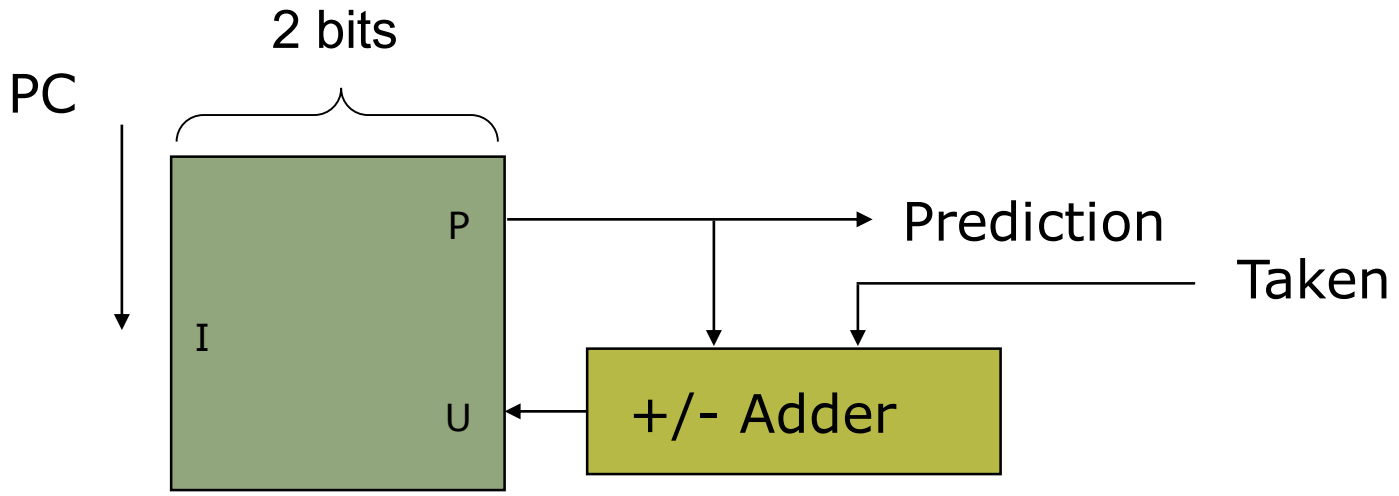
- Use two bits per entry instead of one bit
- Manage them as a saturating counter:

On not-taken ↓	↑ On taken			
		1	1	Strongly taken
		1	0	Weakly taken
		0	1	Weakly not-taken
		0	0	Strongly not-taken

  - Direction prediction changes only after two wrong predictions

How many mispredictions per loop? 1

Two-bit Predictor
Smith, 1981

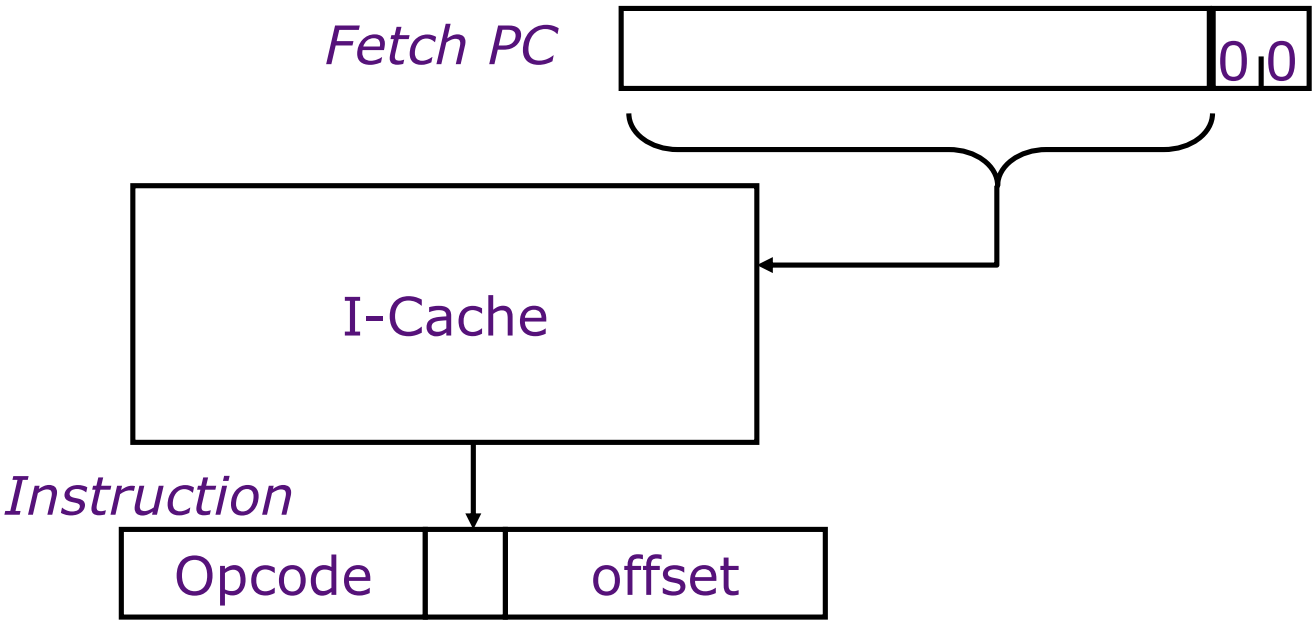


Counter[W,D](I; T) = P[W, D](I; if T then P+1 else P-1)

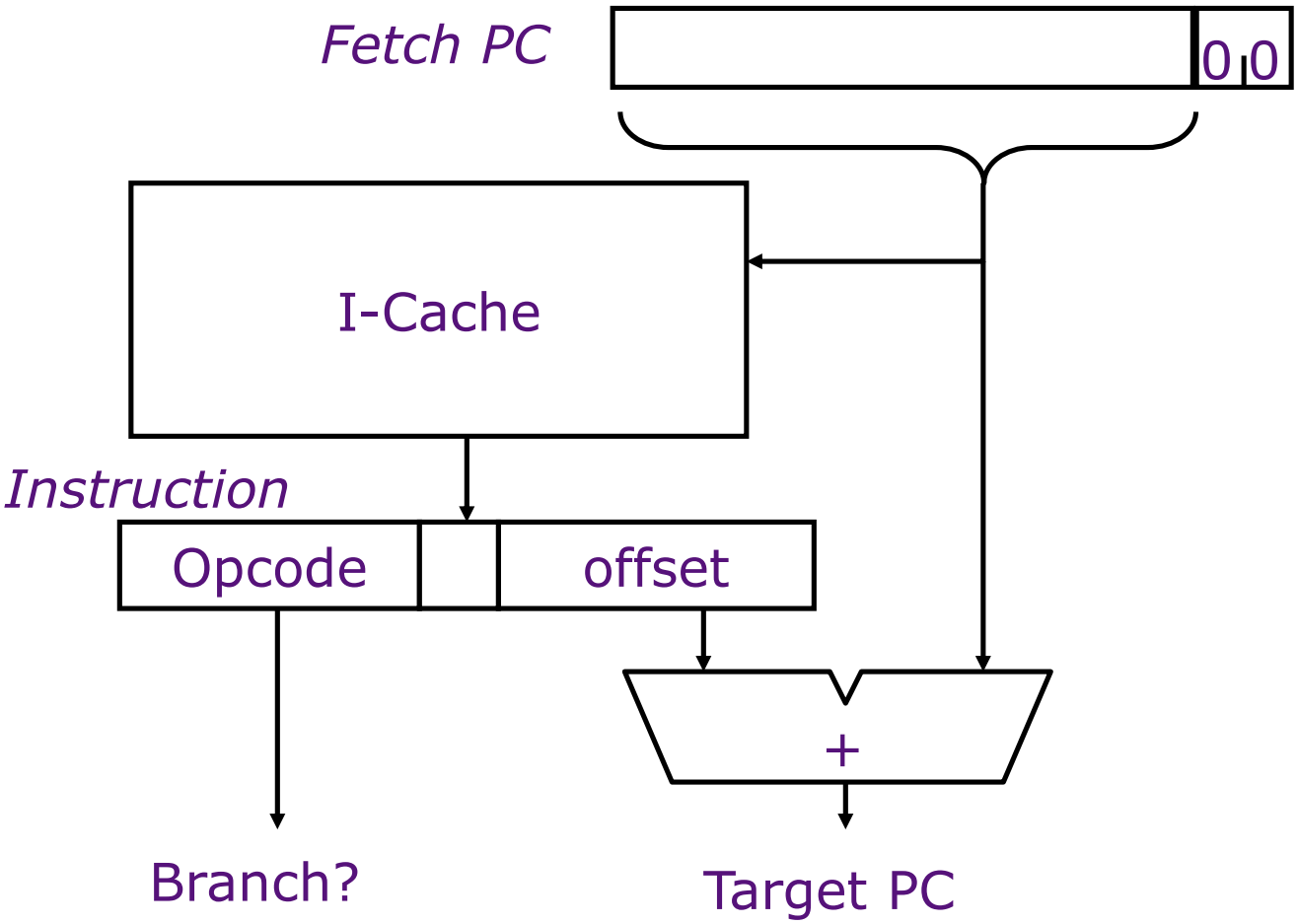
A21164(PC; T) = MSB(Counter[2, 2K](PC; T))

Branch History Table

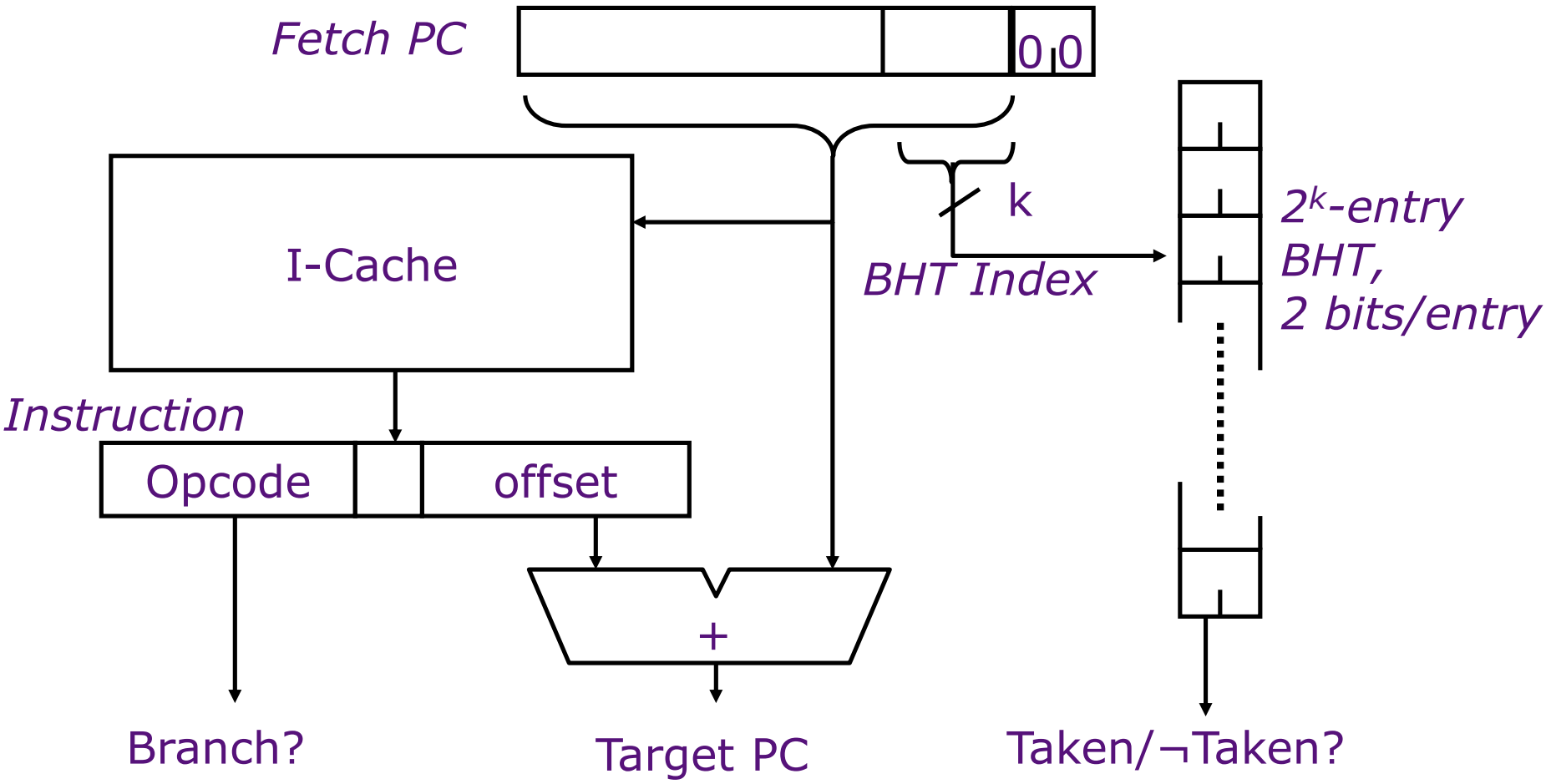
Branch History Table



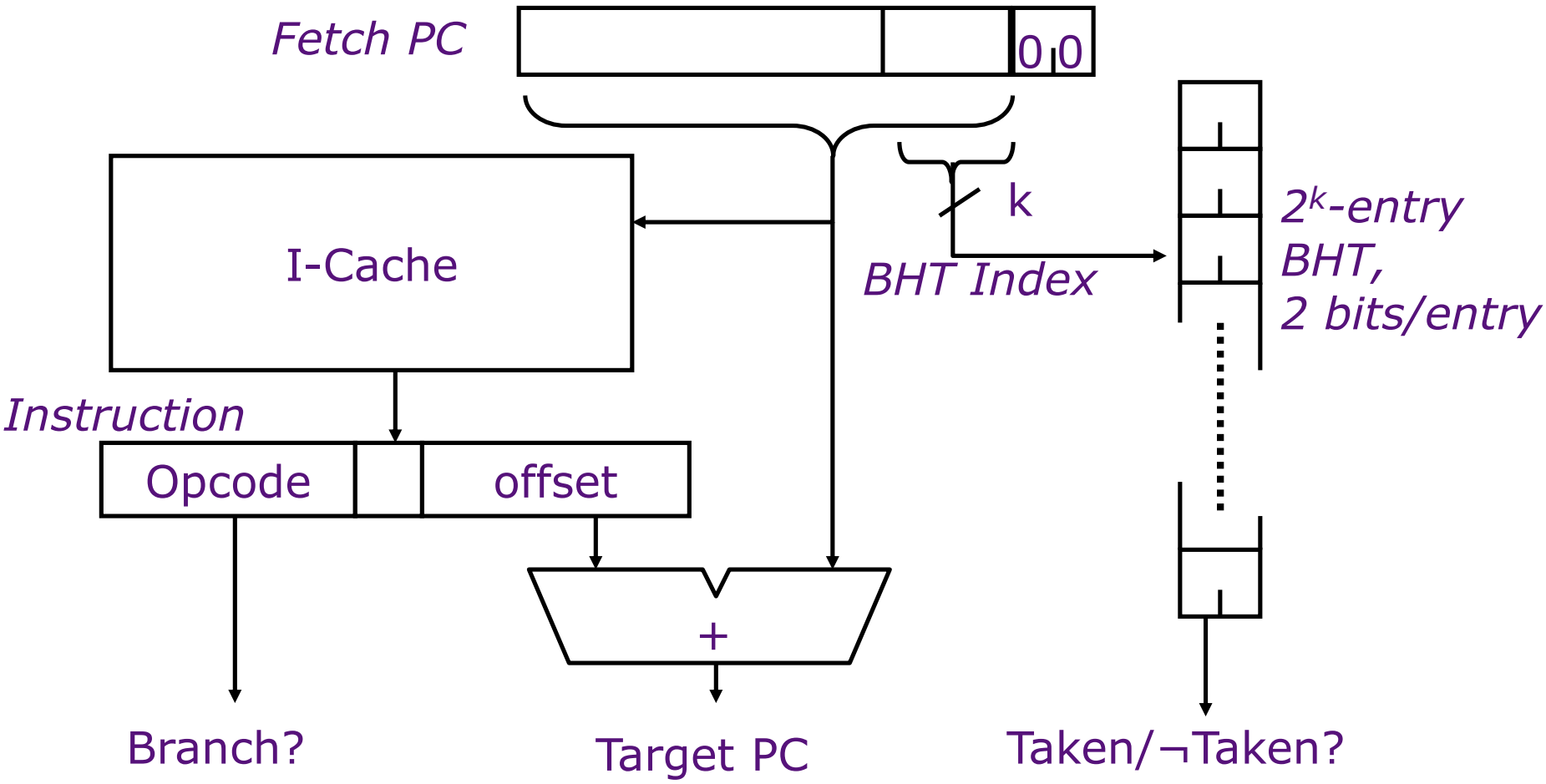
Branch History Table



Branch History Table



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

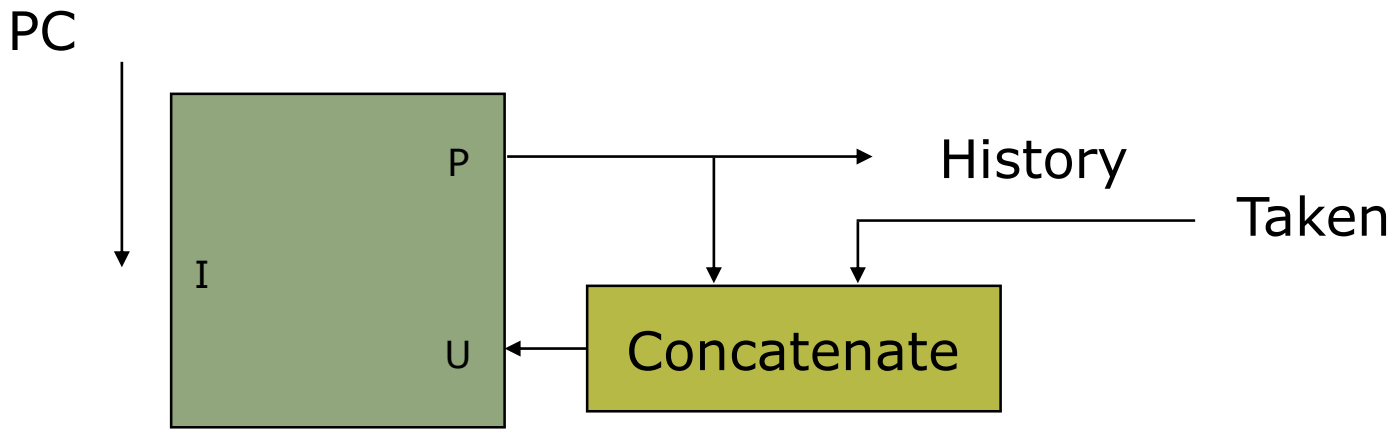
Exploiting Spatial Correlation
Yeh and Patt, 1992

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

If first condition false, second condition also false

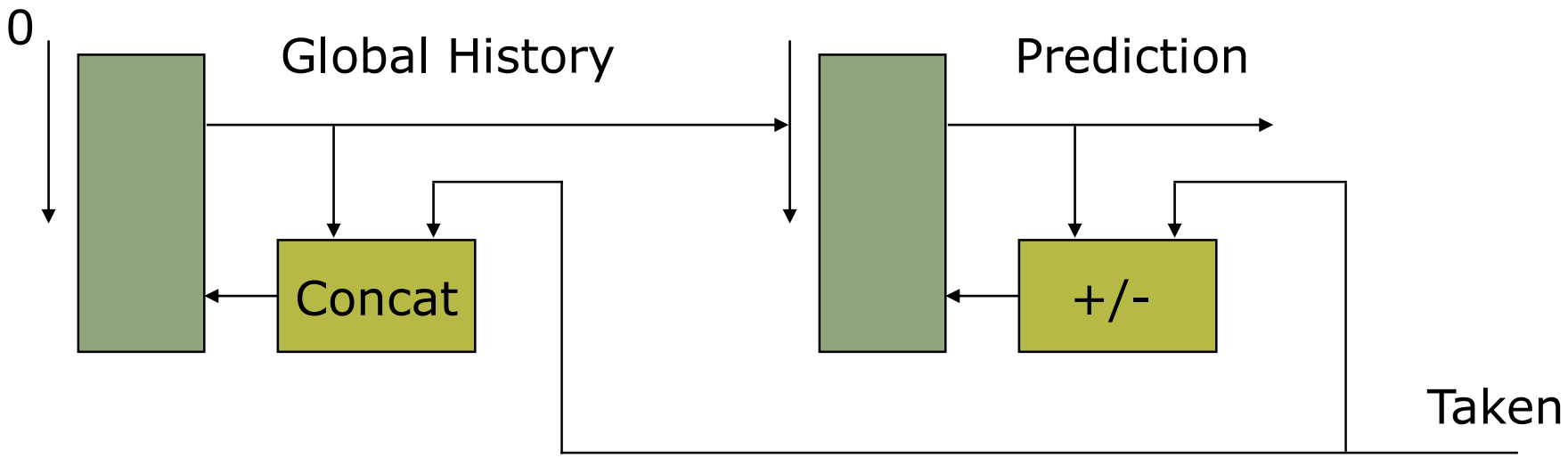
History register records the direction of the last N branches executed by the processor

History Registers
aka Pattern History Table (PHT)



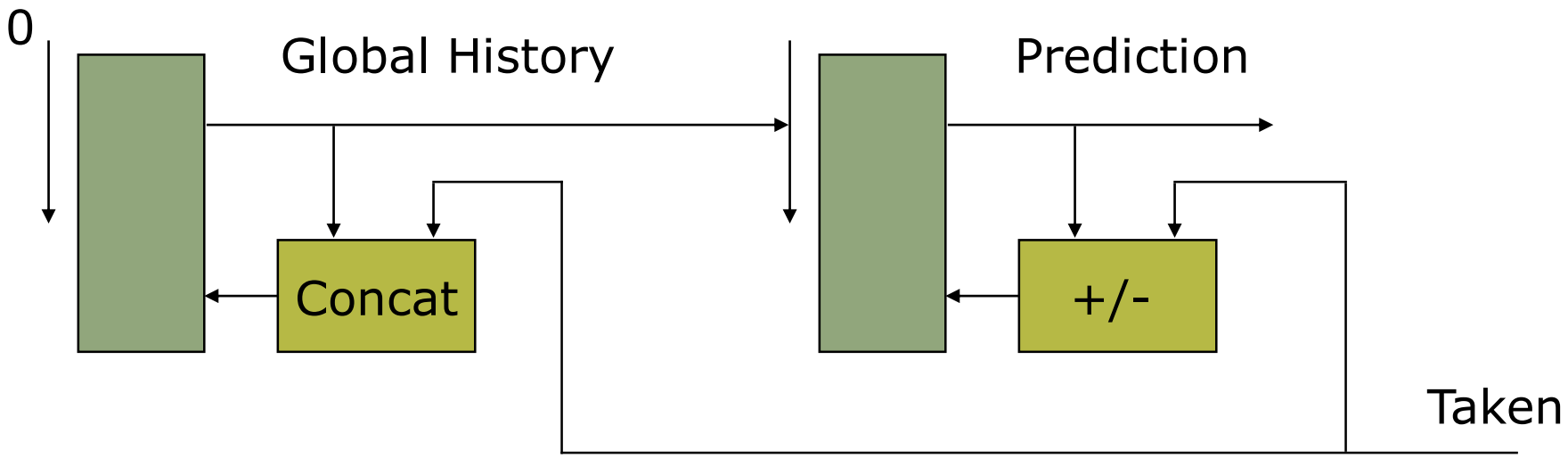
History(PC; T) = P(PC; P || T)

Global-History Predictor



GHist(;T) = MSB(Counter(History(0, T); T))

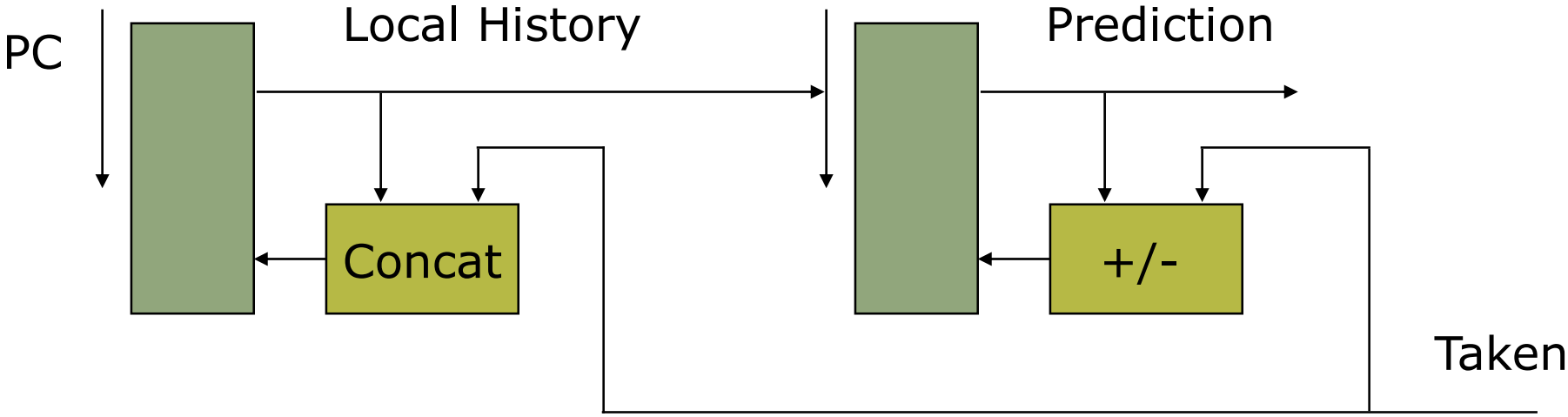
Global-History Predictor



GHist(;T) = MSB(Counter(History(0, T); T))

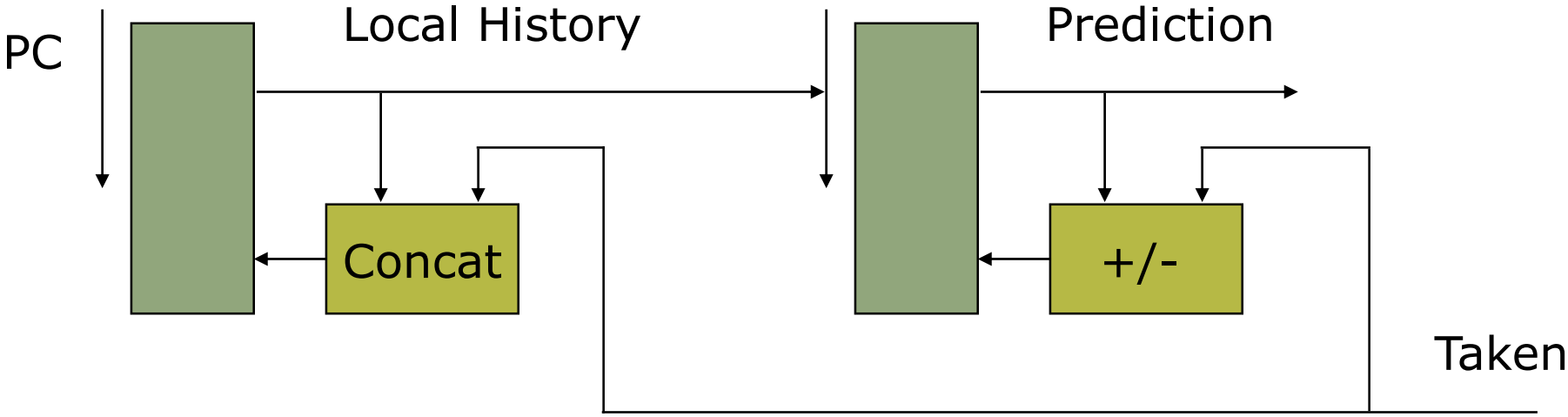
Can we take advantage of a pattern at a particular PC?

Local-History Predictor



LHist(PC; T) = MSB(Counter(History(PC; T); T))

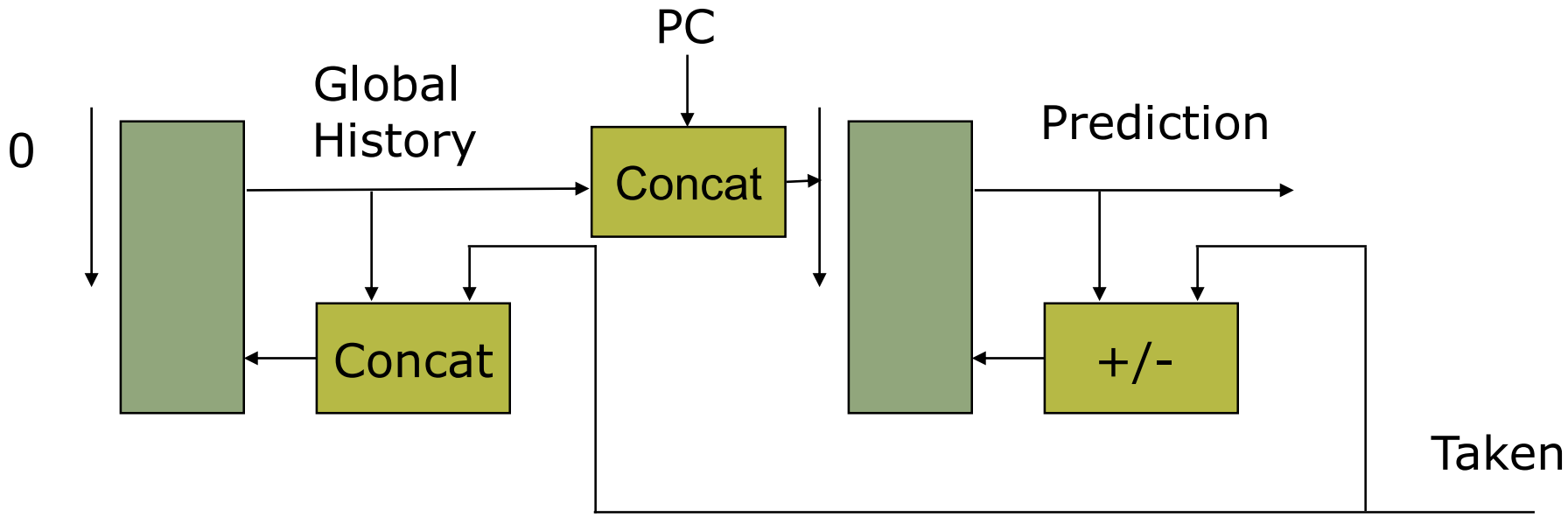
Local-History Predictor



LHist(PC; T) = MSB(Counter(History(PC; T); T))

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

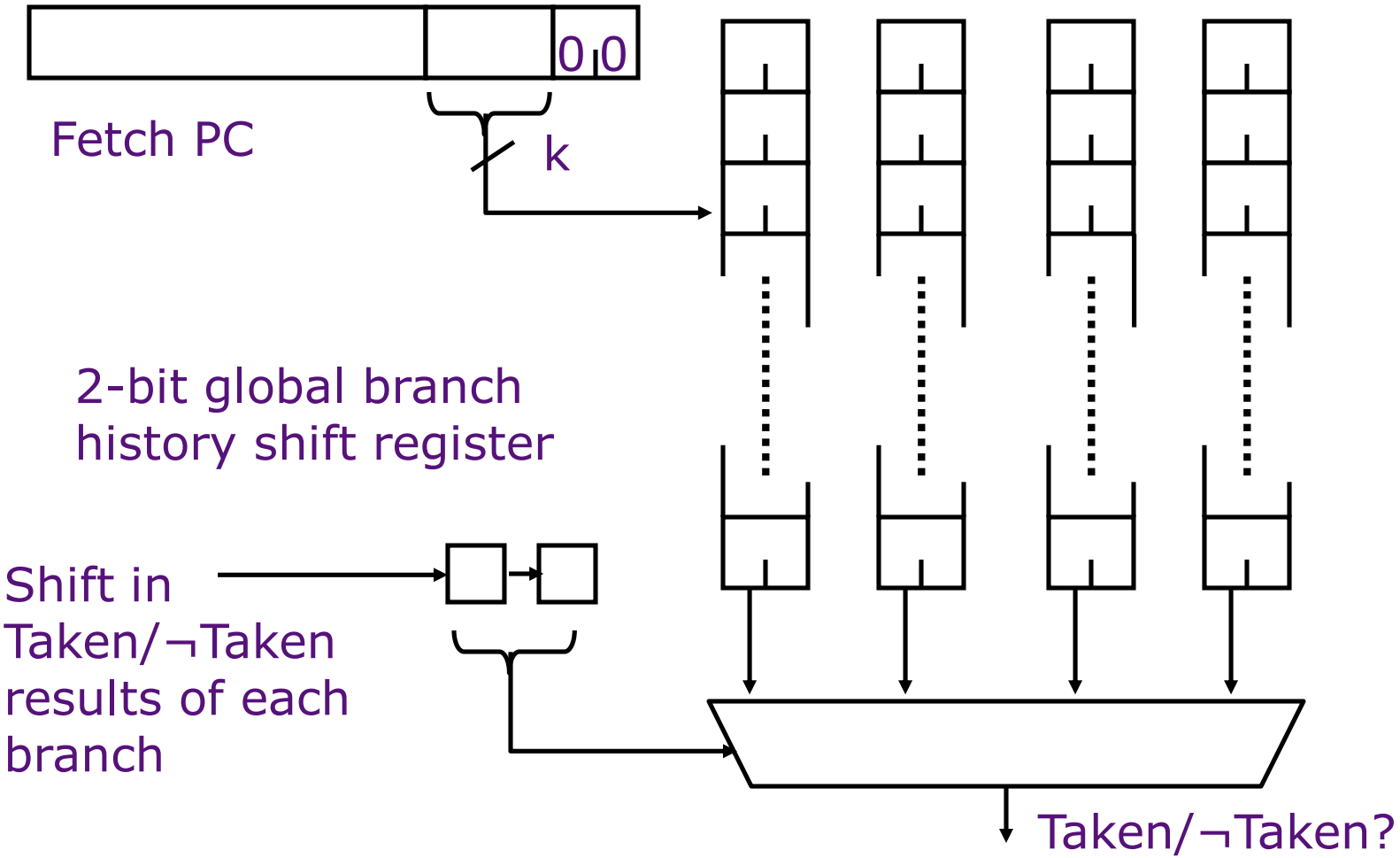
Global-History Predictor with Per-PC Counters



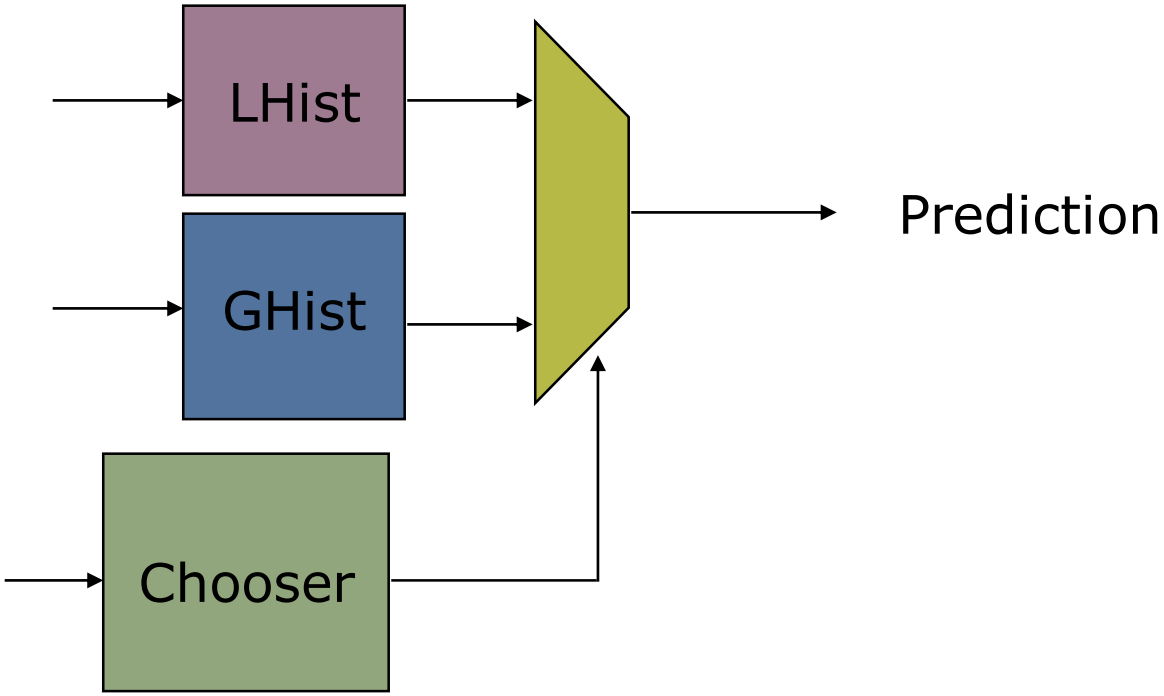
GHistPA(PC; T) = MSB(Counter(History(0; T)||PC; T))

Two-Level Branch Predictor
(Pentium Pro, 1995)

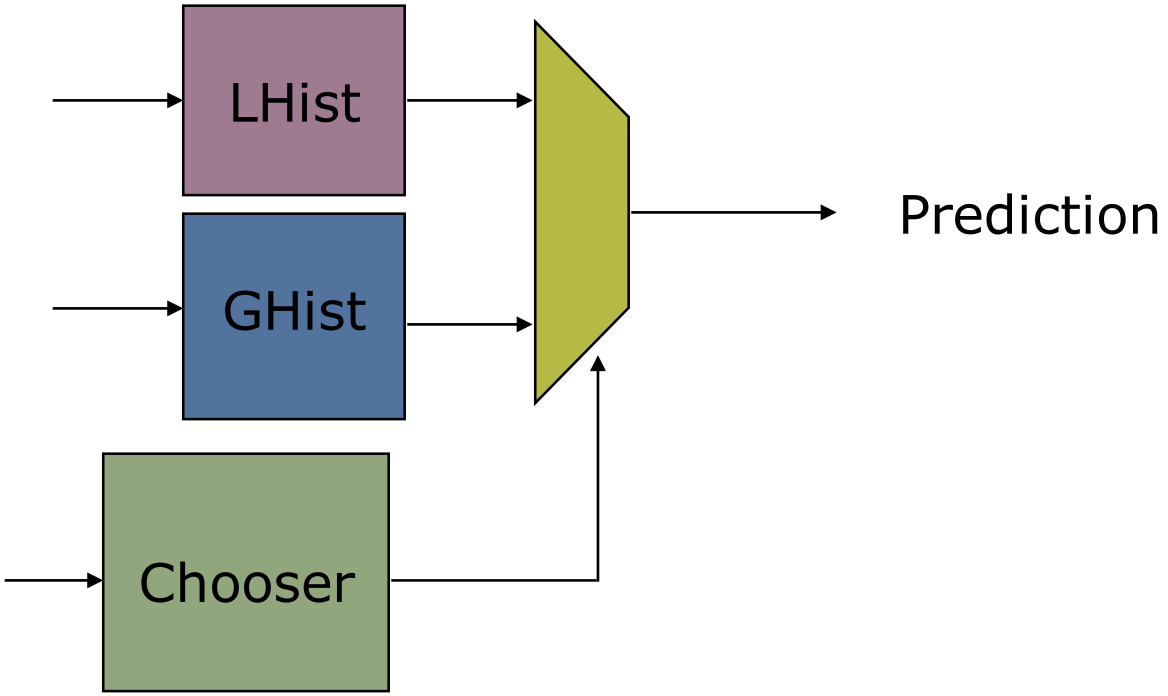
Pentium Pro uses the result from the last two branches to select one of the four sets of BHT bits (~95% correct)



Choosing Predictors

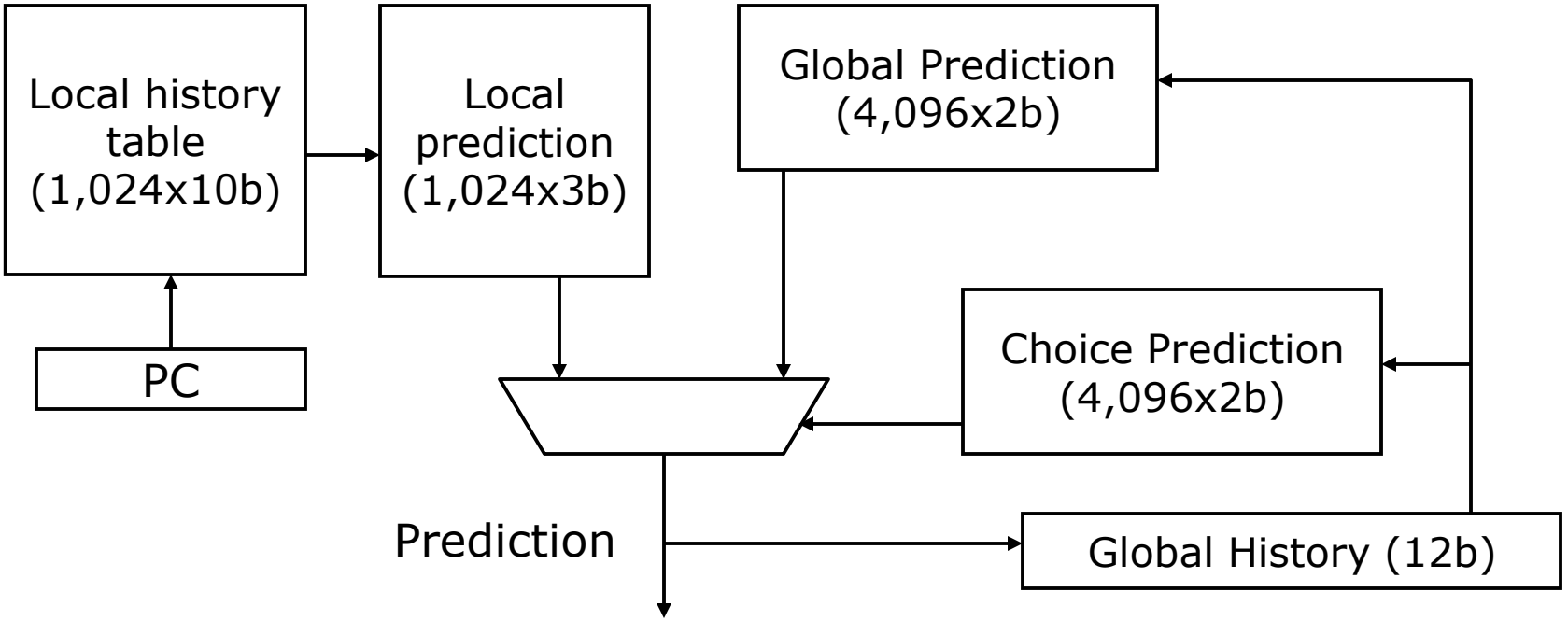


Choosing Predictors



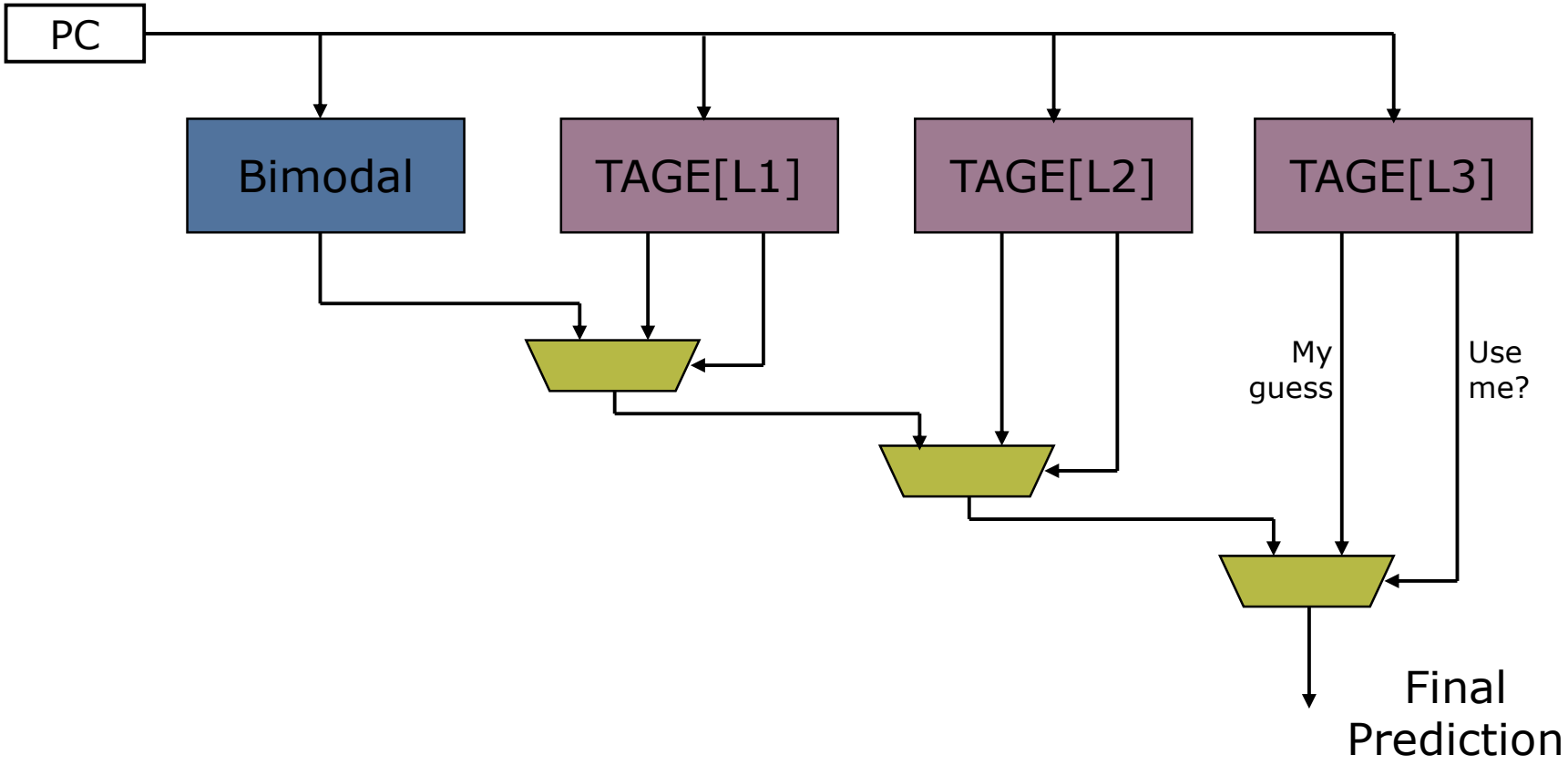
Chooser = MSB(P(PC; P + (A==T) - (B==T))
or
Chooser = MSB(P(GHist(PC; T); P + (A==T) - (B==T))

Tournament Branch Predictor
(Alpha 21264, 1996)

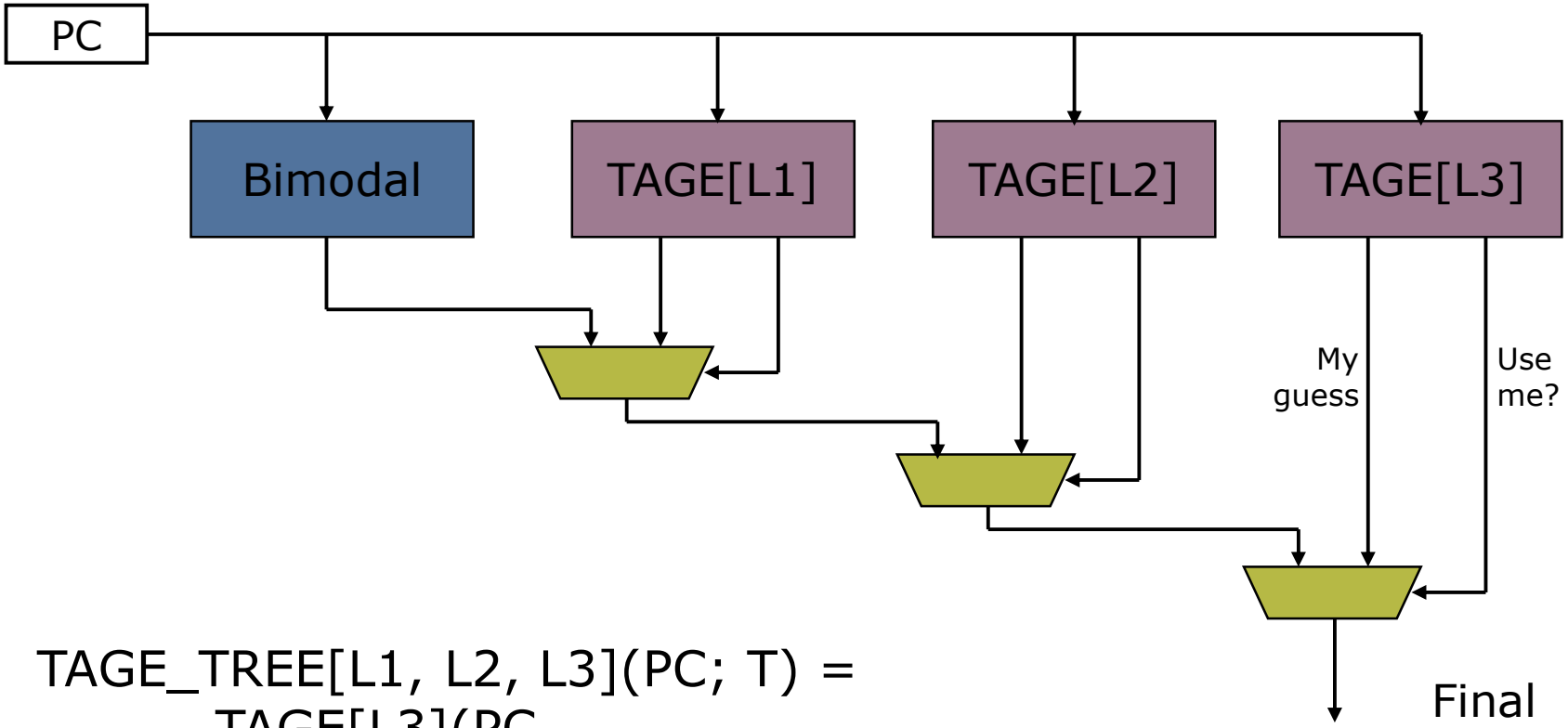


- Choice predictor learns whether best to use local or global branch history in predicting next branch
- Global history is speculatively updated but restored on mispredict
- Claim 90-100% success on range of applications

TAGE predictor
Seznec & Michaud, 2006

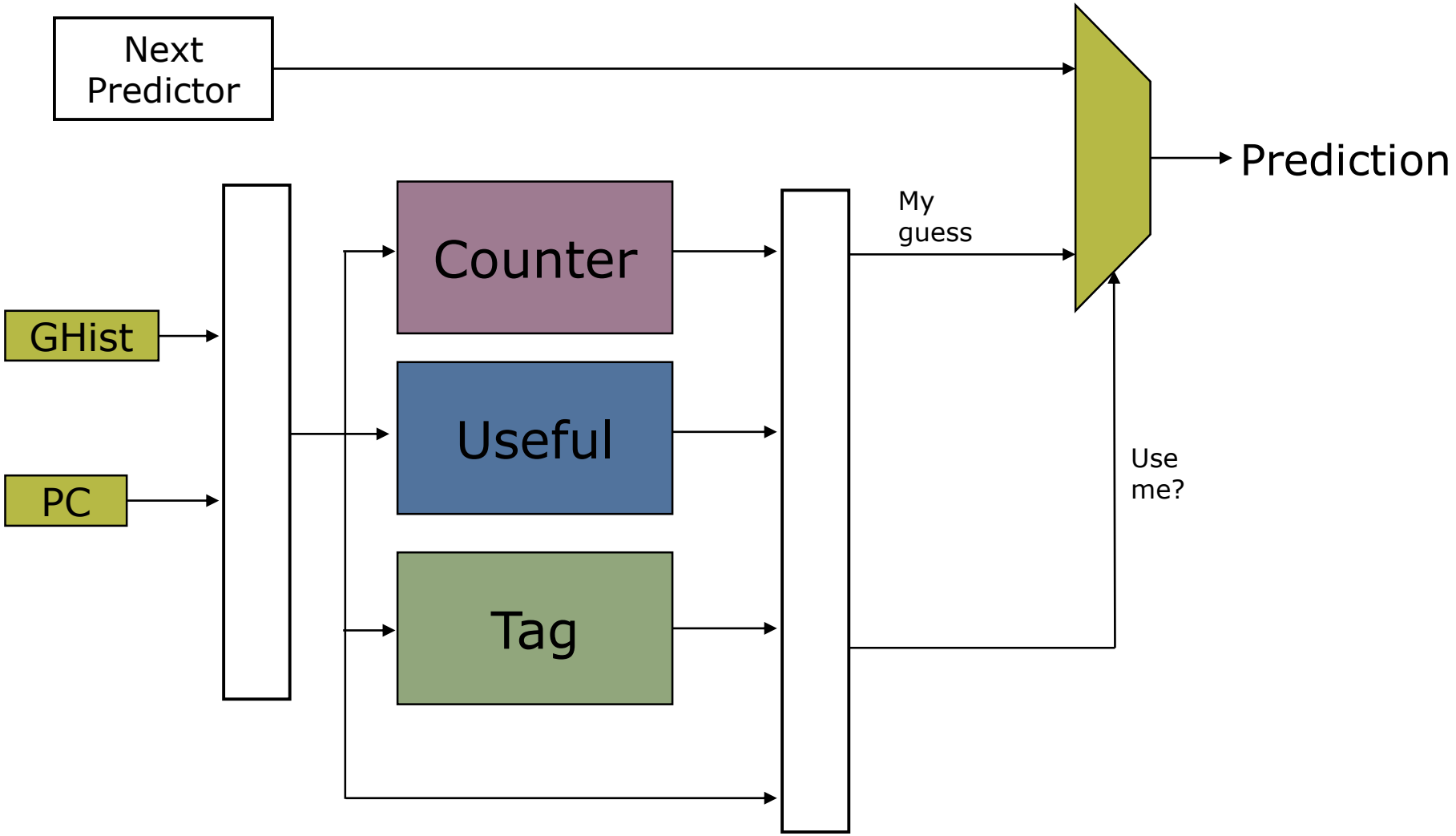


TAGE predictor
Seznec & Michaud, 2006




```
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 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)

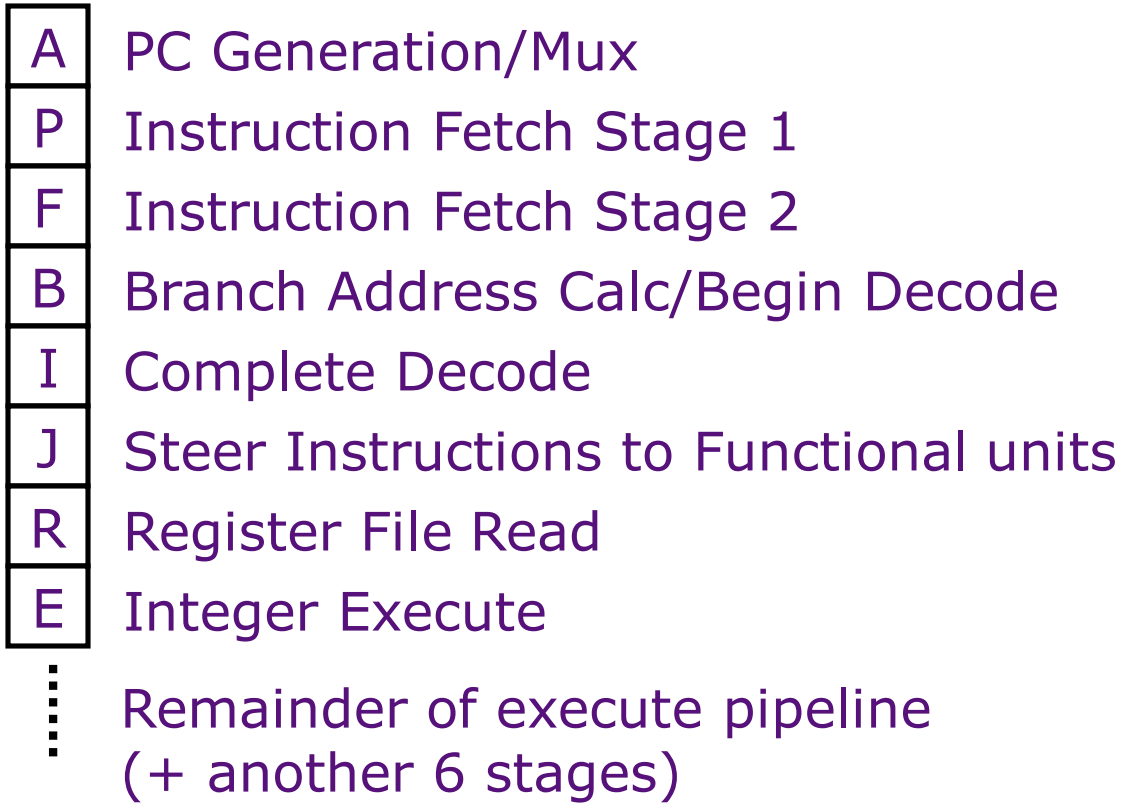
use_me = TAGE.U && isStrong(TAGE.Counter)
TAGE = use_me?MSB(TAGE.Counter):NEXT

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.

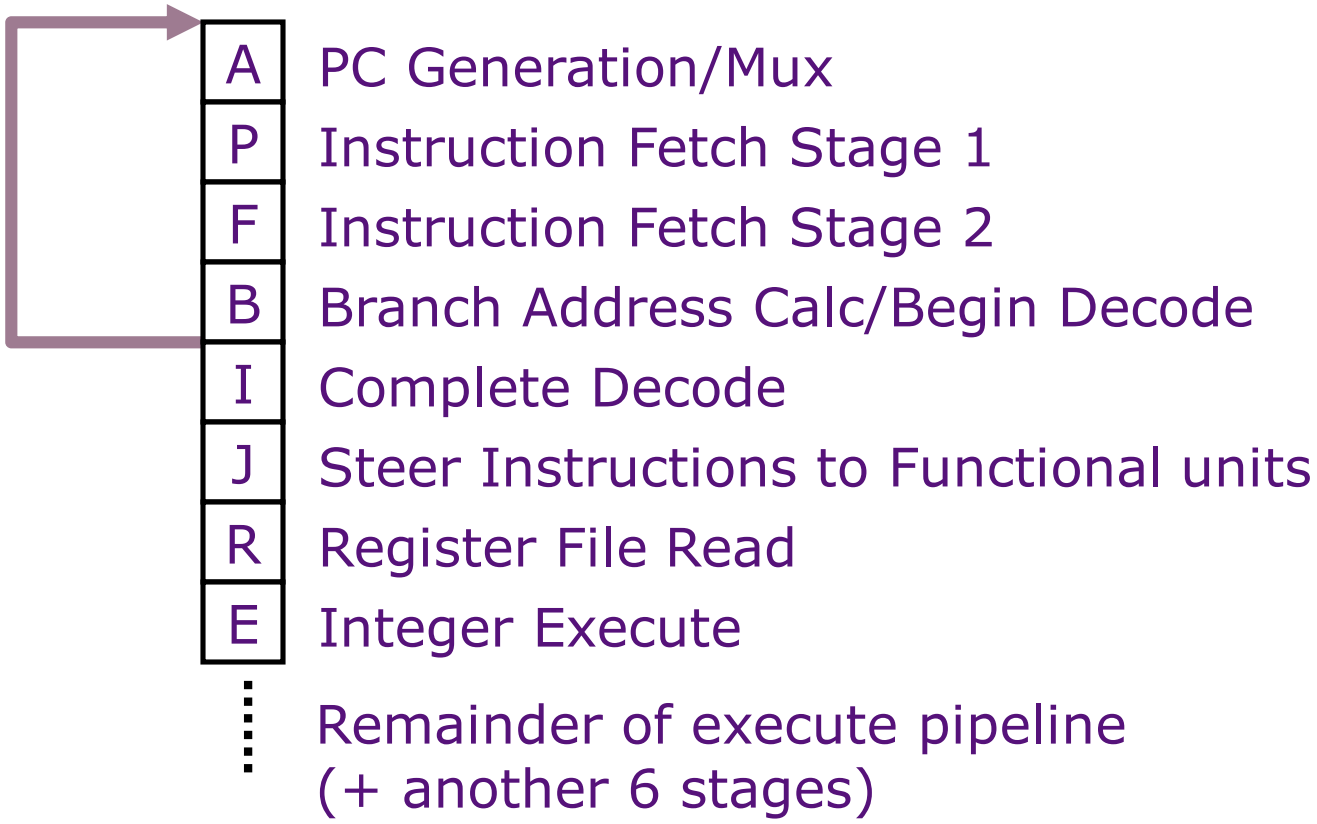


UltraSPARC-III fetch pipeline

Limitations of branch predictors

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

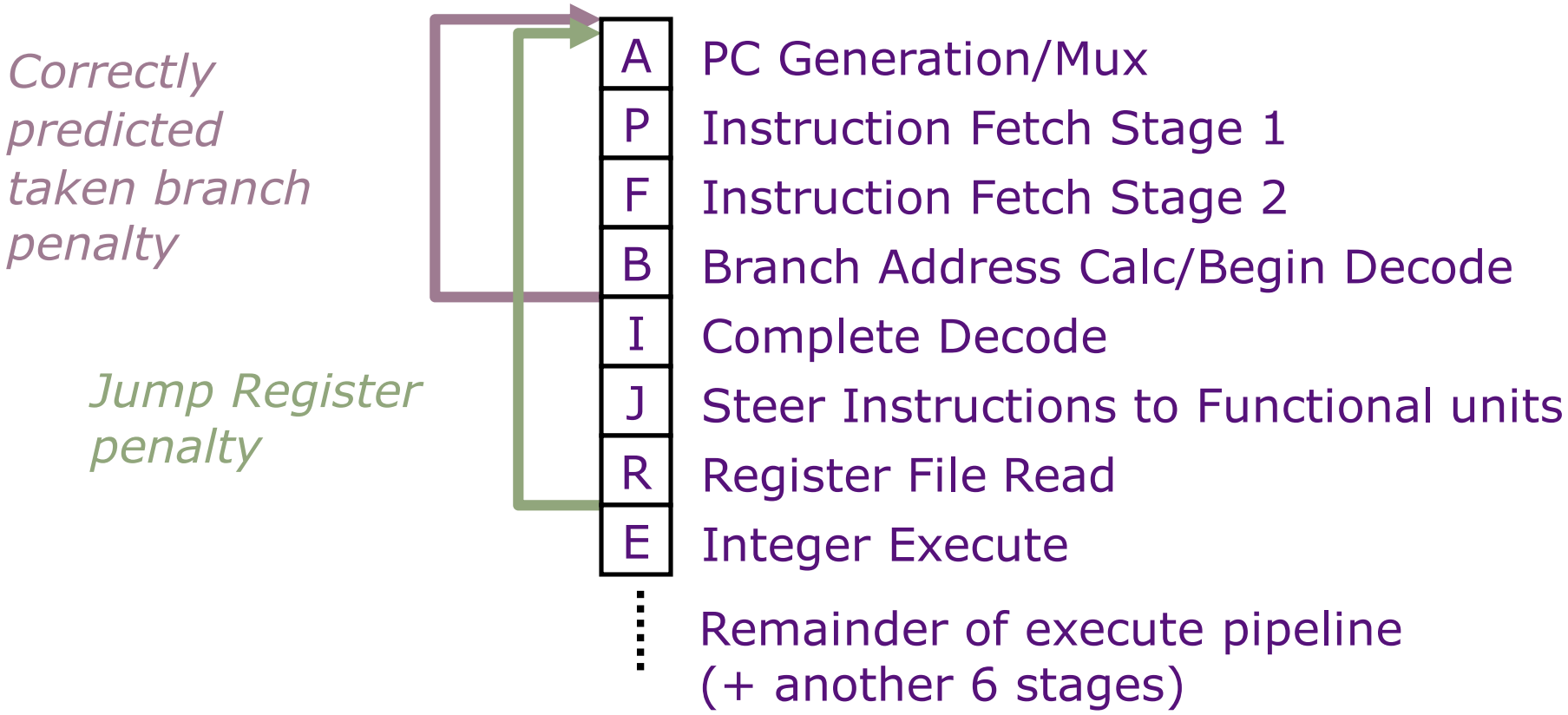
Correctly predicted taken branch penalty



UltraSPARC-III fetch pipeline

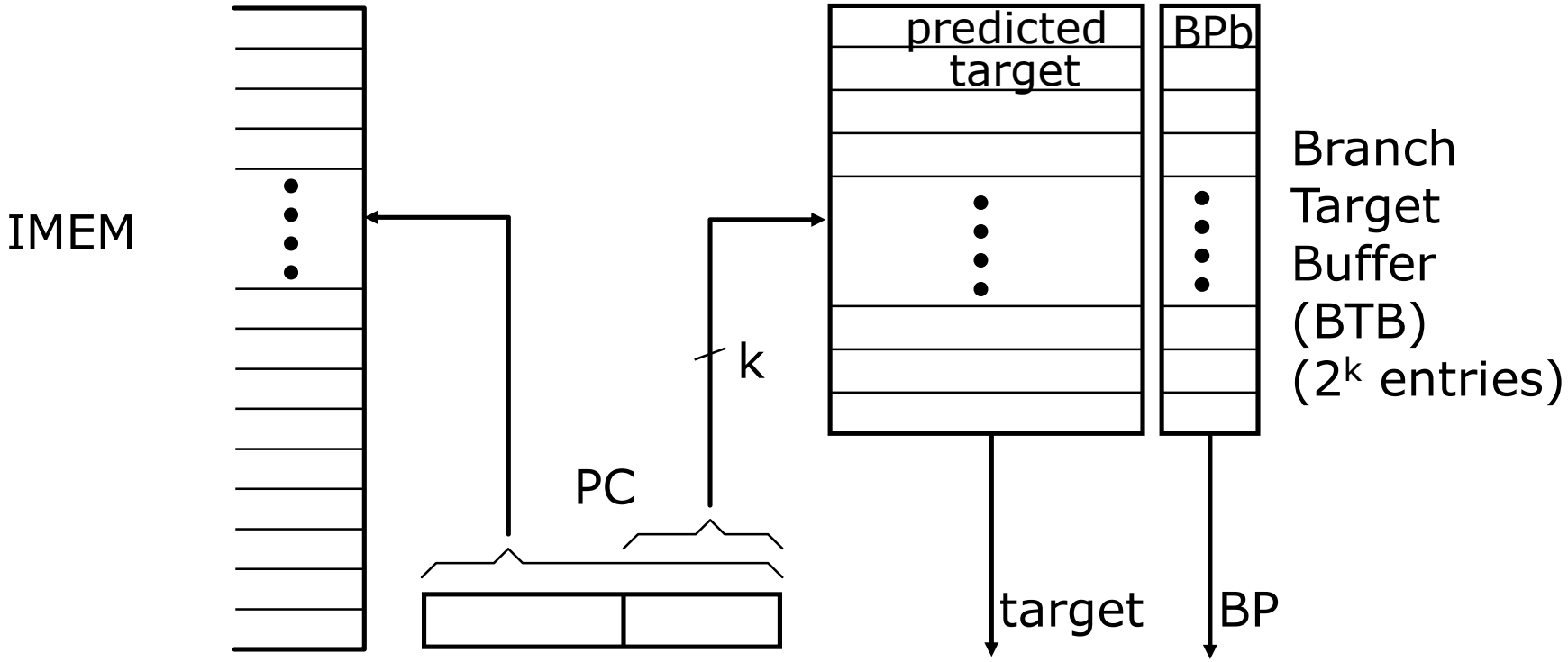
Limitations of branch predictors

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



UltraSPARC-III fetch pipeline

Branch Target Buffer (untagged)

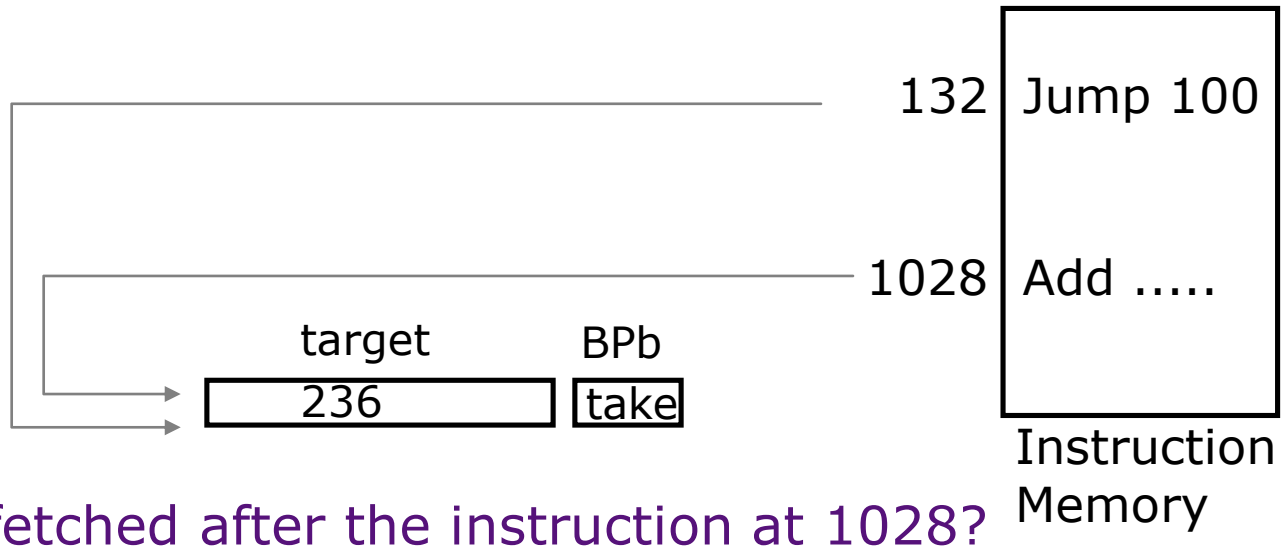


BP bits are stored with the predicted target address.

IF stage: *If (BP=taken) then nPC=target else nPC=PC+4*
later: *check prediction, if wrong then kill the instruction and update BTB & BPb, else update BPb*

Address Collisions

Assume a 128-entry BTB



What will be fetched after the instruction at 1028?

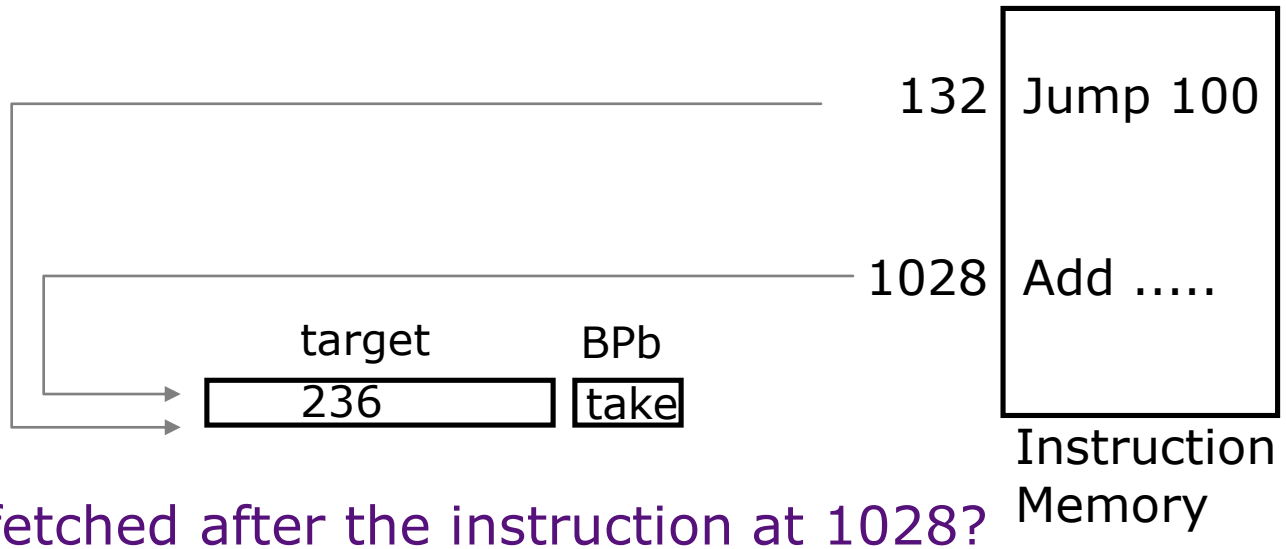
BTB prediction =

Correct target =

⇒

Address Collisions

Assume a 128-entry BTB



What will be fetched after the instruction at 1028?

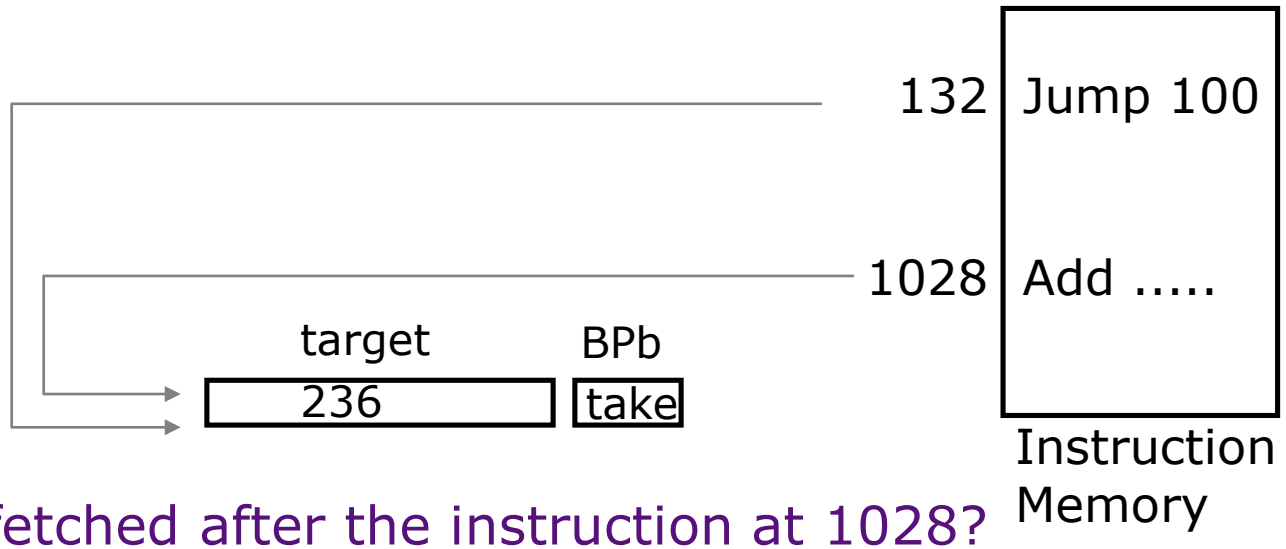
BTB prediction = 236

Correct target =

⇒

Address Collisions

Assume a 128-entry BTB



What will be fetched after the instruction at 1028?

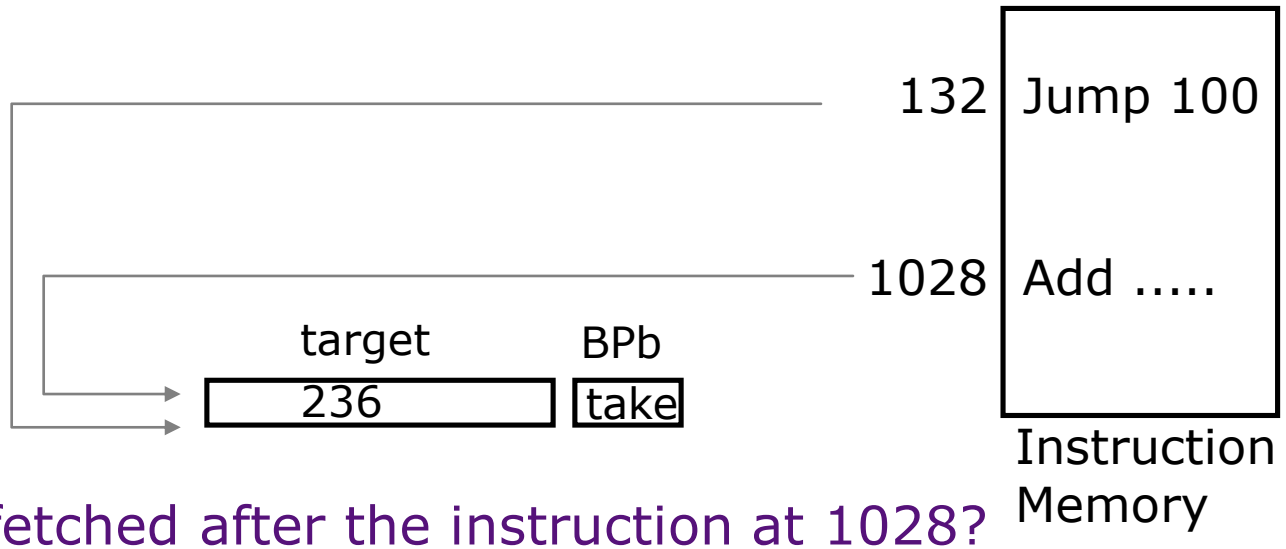
BTB prediction = 236

Correct target = 1032

⇒

Address Collisions

Assume a 128-entry BTB



What will be fetched after the instruction at 1028?

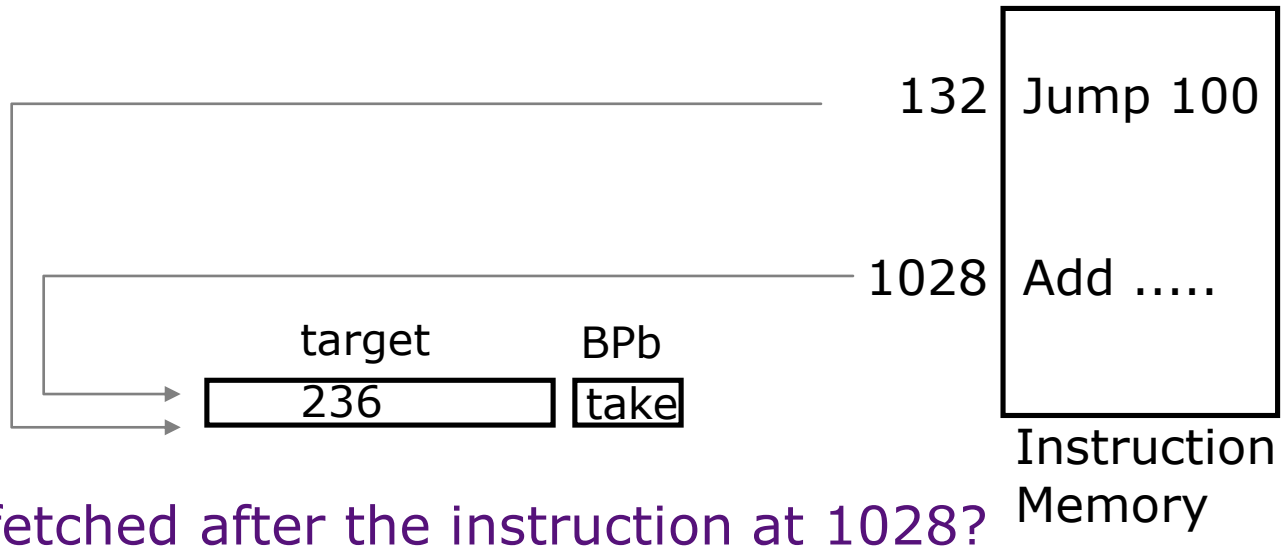
BTB prediction = 236

Correct target = 1032

⇒ *kill* PC=236 and *fetch* PC=1032

Address Collisions

Assume a 128-entry BTB



What will be fetched after the instruction at 1028?

BTB prediction = 236

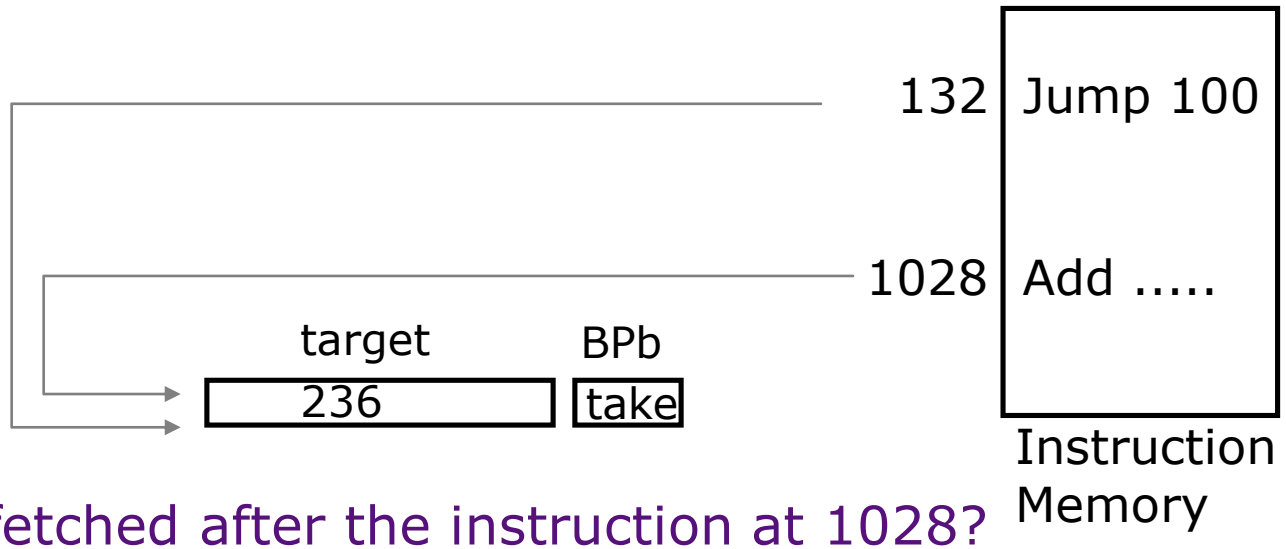
Correct target = 1032

⇒ *kill* PC=236 and *fetch* PC=1032

Is this a common occurrence?

Address Collisions

Assume a 128-entry BTB



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 mispredictions?

BTB is only for Control Instructions

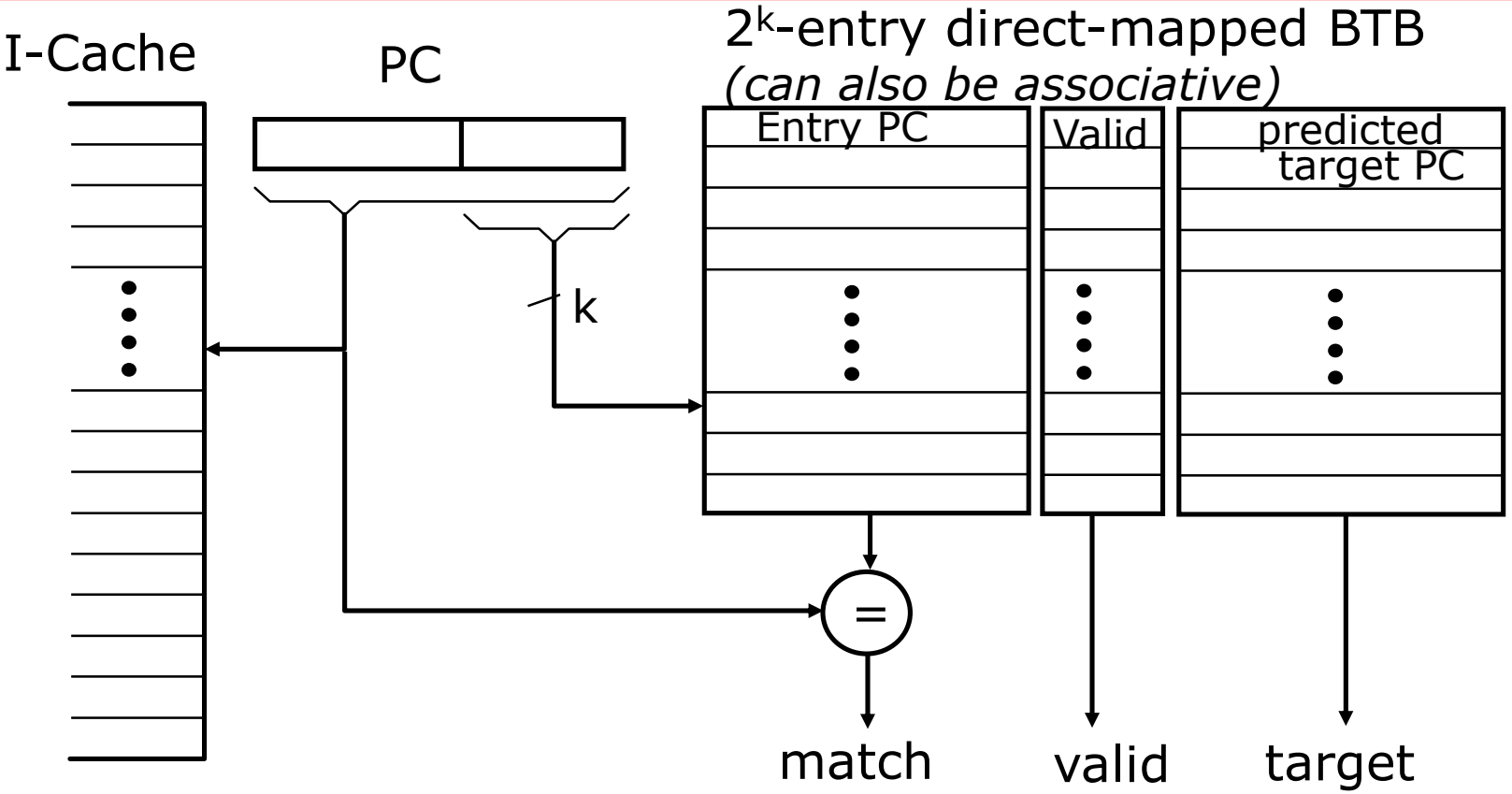
BTB contains useful information for branch and jump instructions only

$\Rightarrow$ Do not update it for other instructions

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

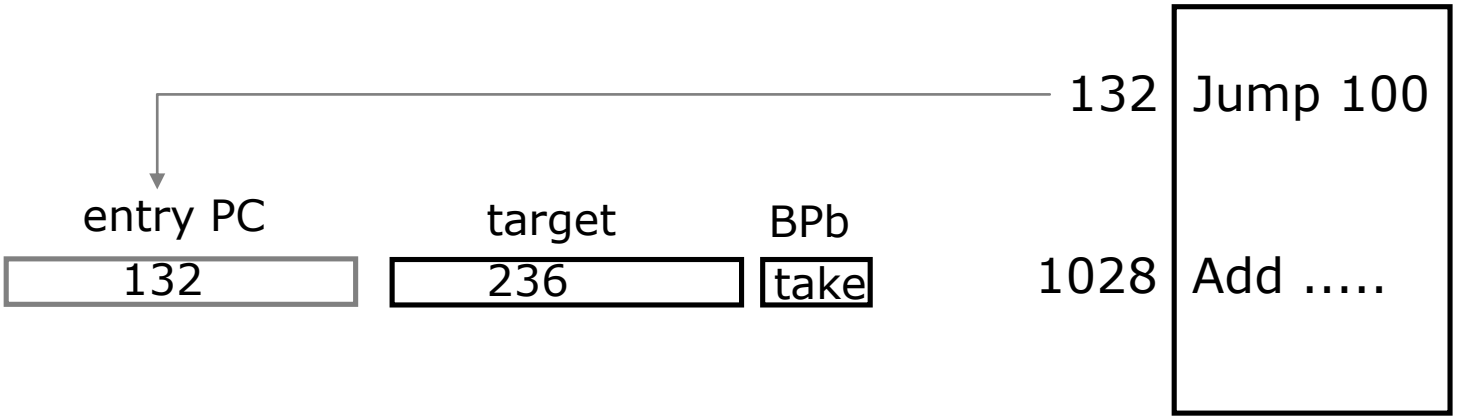
How to achieve this effect without decoding the instruction?

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*

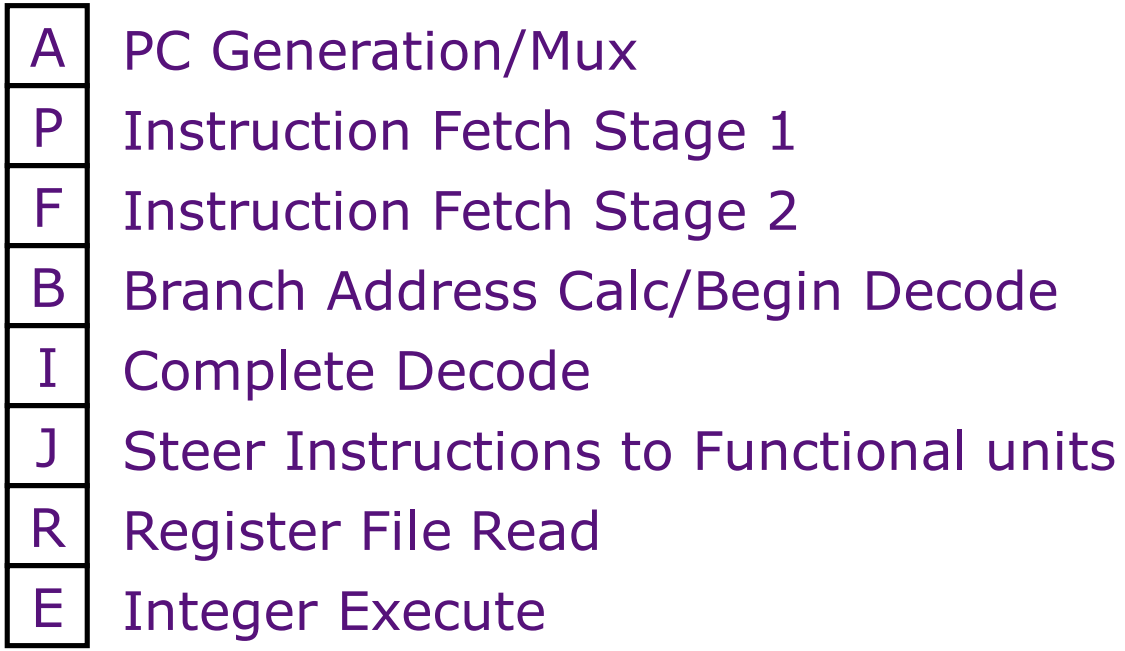
- BTB contains entries only for control transfer instructions
  ⇒ *more room to store branch targets*

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

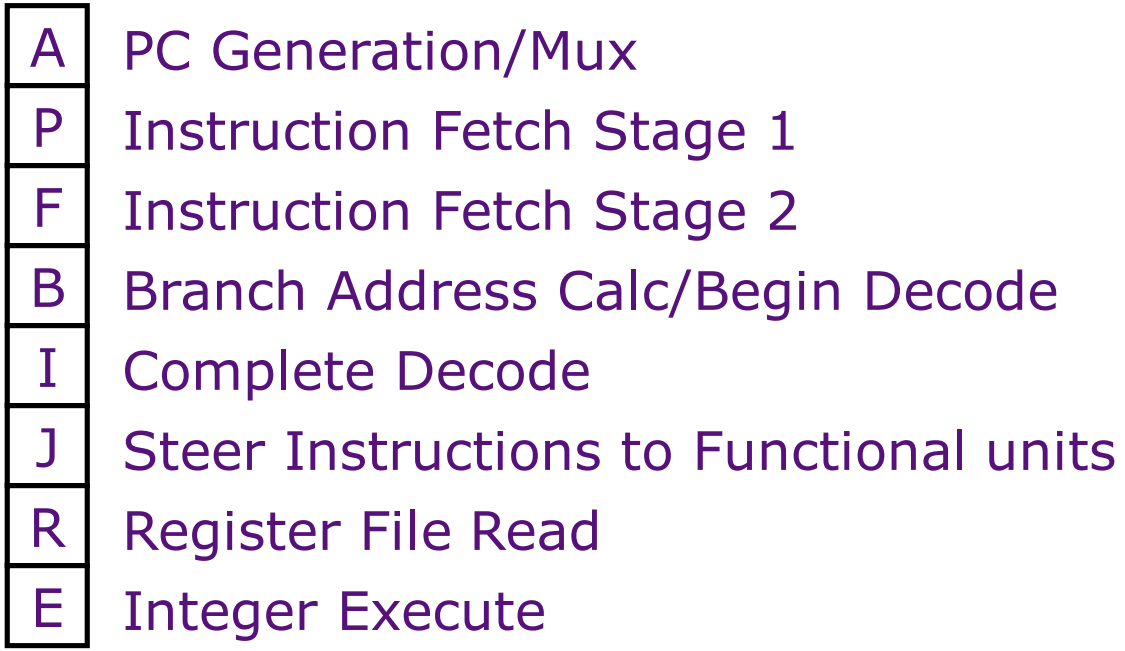
Combining BTB and BHT

- 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



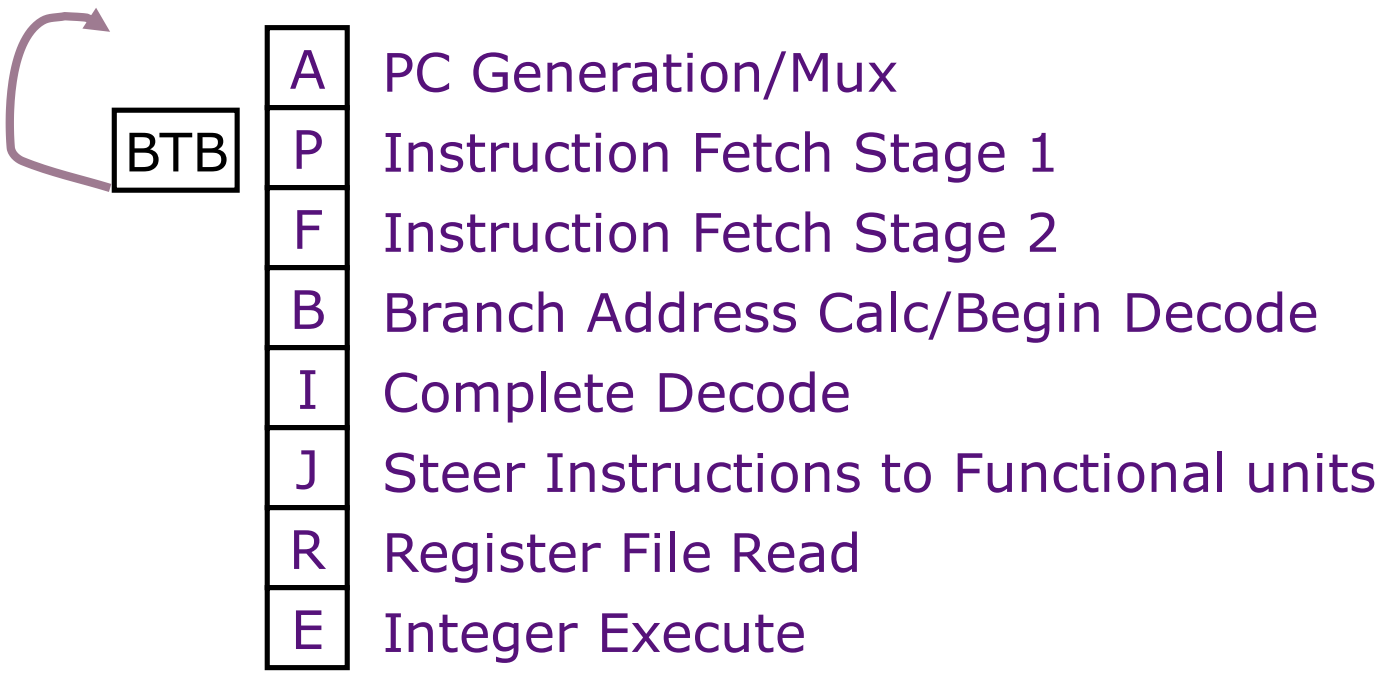
Combining BTB and BHT

- 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



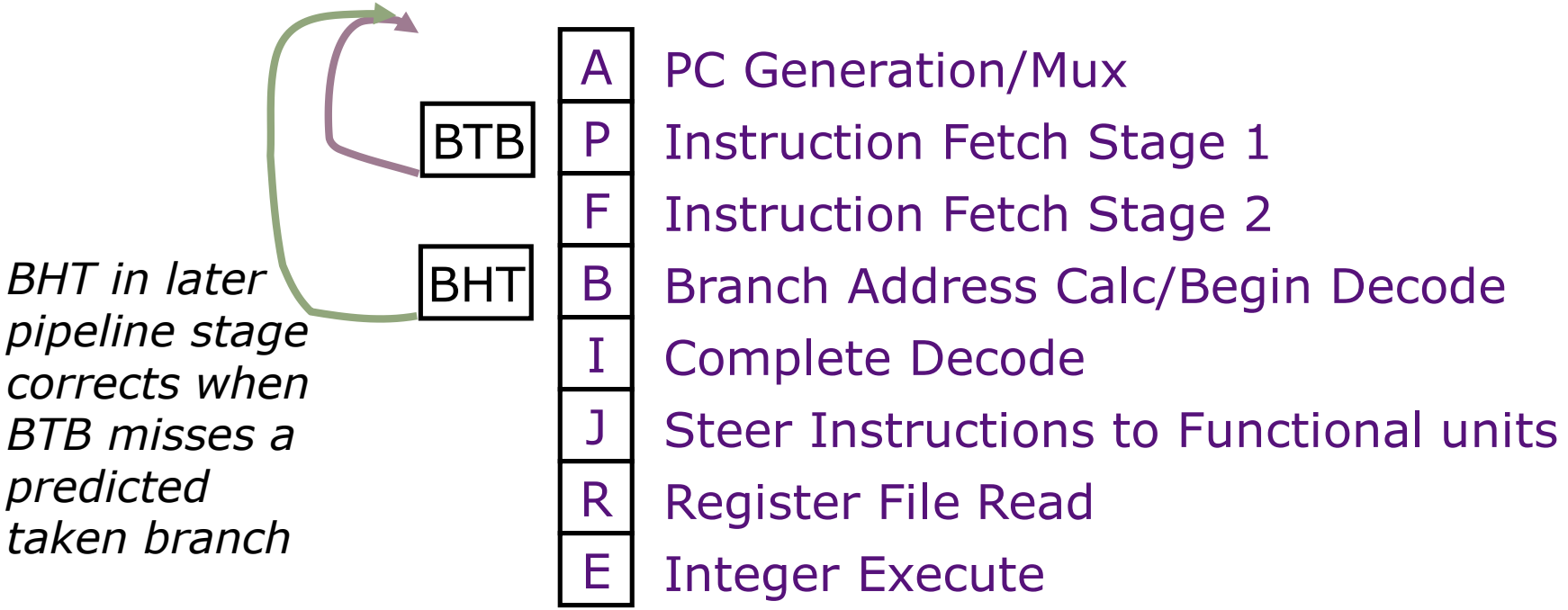
Combining BTB and BHT

- 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



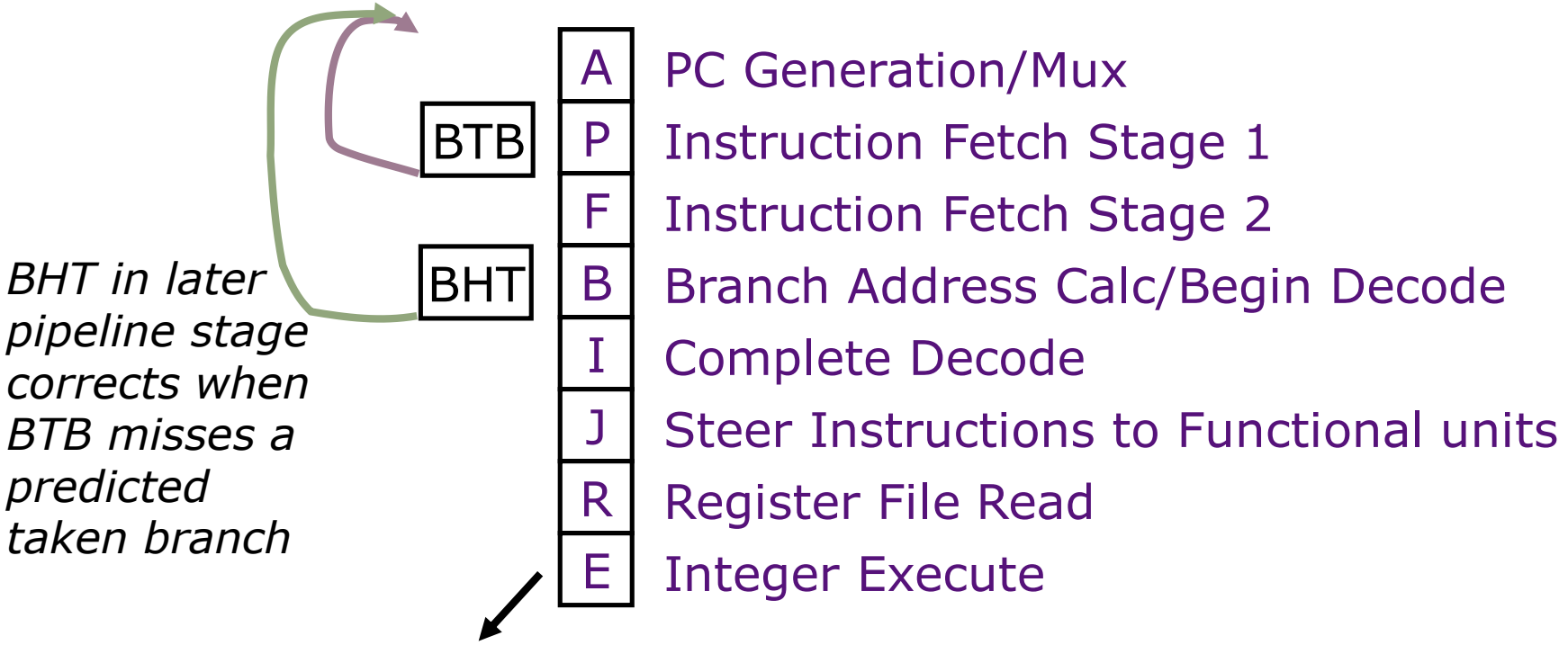
Combining BTB and BHT

- 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



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)
- Dynamic function call (jump to run-time function address)
- Subroutine returns (jump to return address)

Uses of Jump Register (JR)

- Switch statements (jump to address of matching case)
- Dynamic function call (jump to run-time function address)
- Subroutine returns (jump to return address)

How well does BTB work for each of these cases?

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)

- Subroutine returns (jump to return address)

How well does BTB work for each of these cases?

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)

How well does BTB work for each of these cases?

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

How well does BTB work for each of these cases?

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.

Subroutine Return Stack

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

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

fb() { fc(); }

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

Subroutine Return Stack

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

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

fb() { fc(); }

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

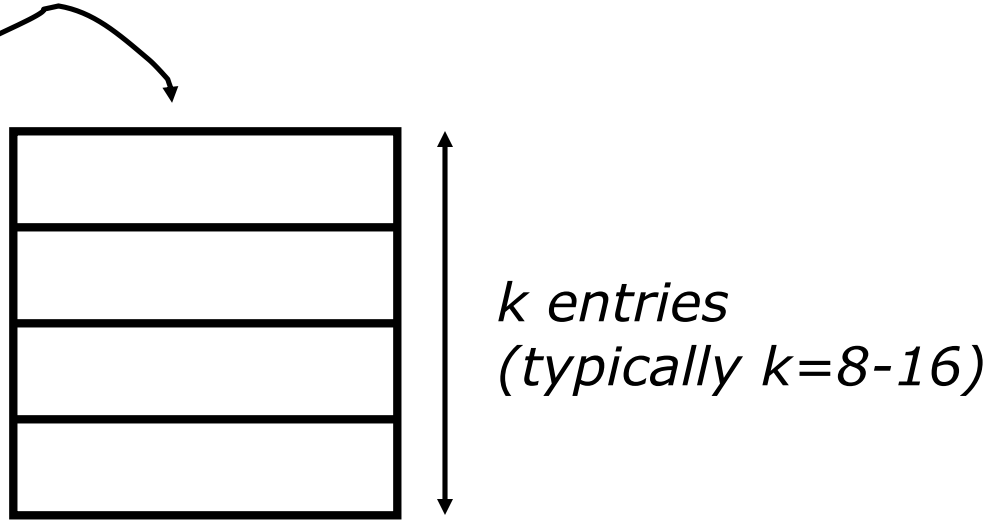
k entries
(typically k=8-16)

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

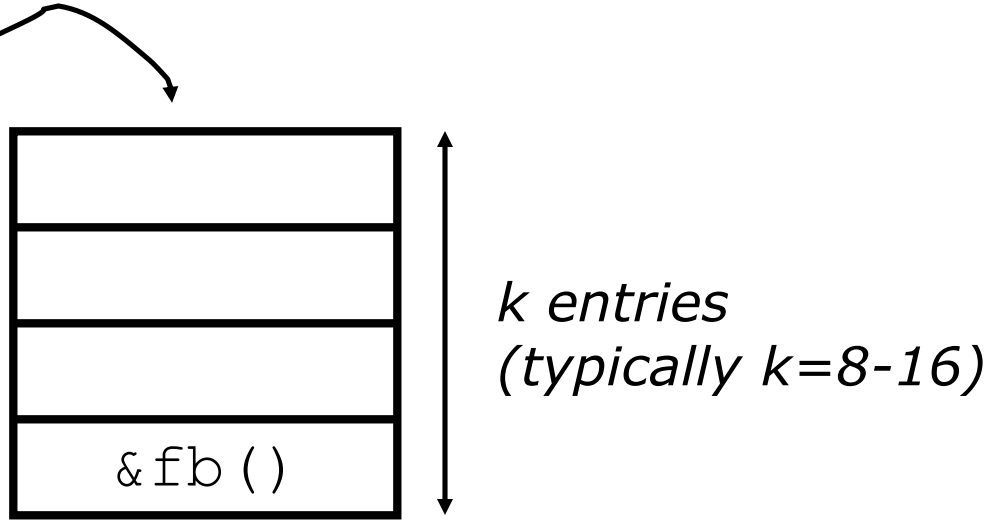


Subroutine Return Stack

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

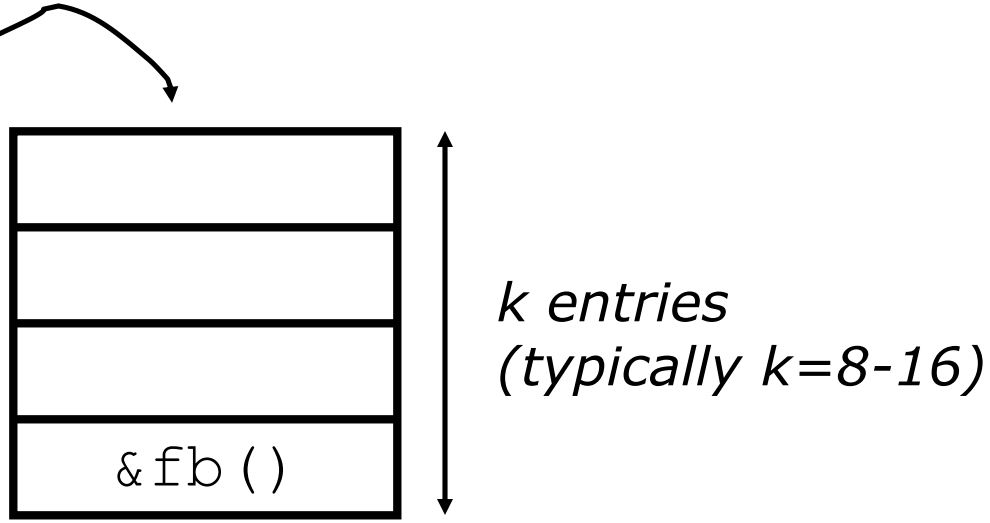


Subroutine Return Stack

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

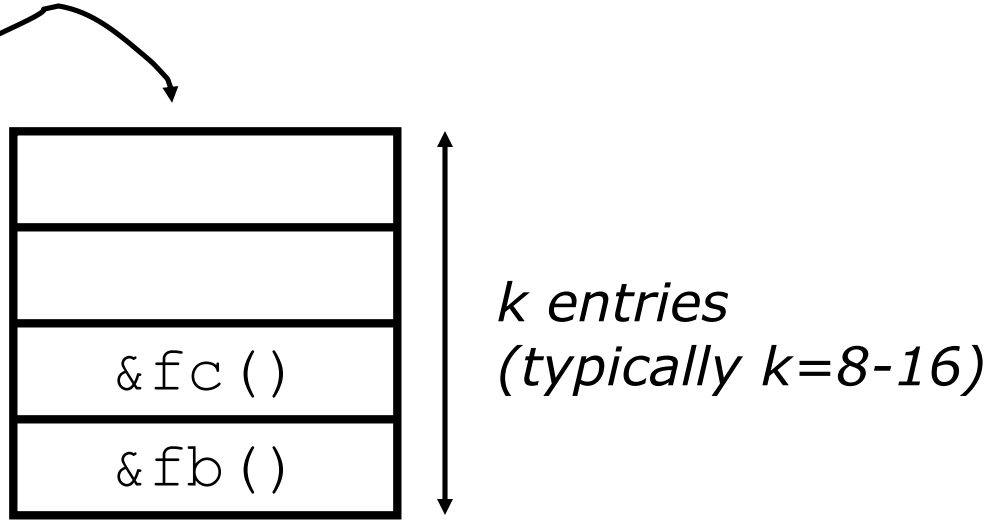


Subroutine Return Stack

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

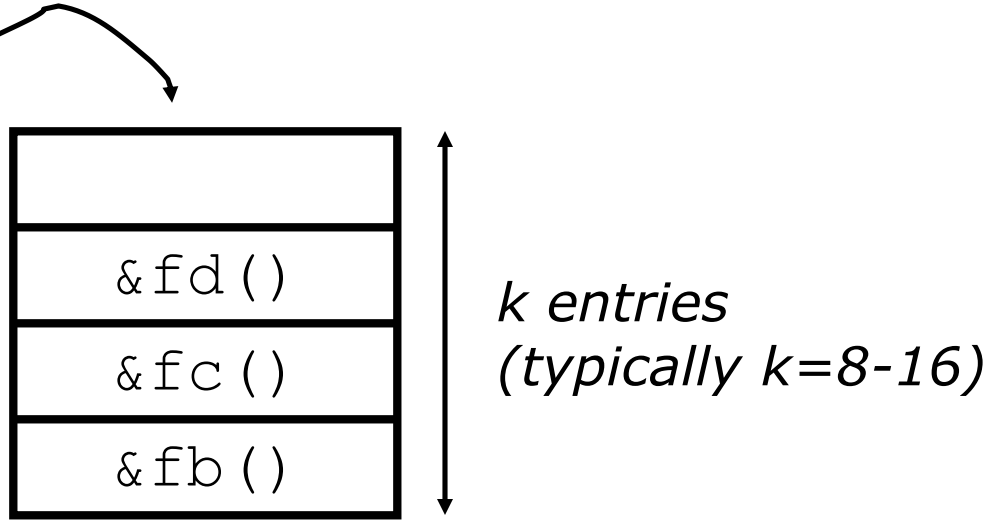


Subroutine Return Stack

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

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

fb() { fc(); }

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

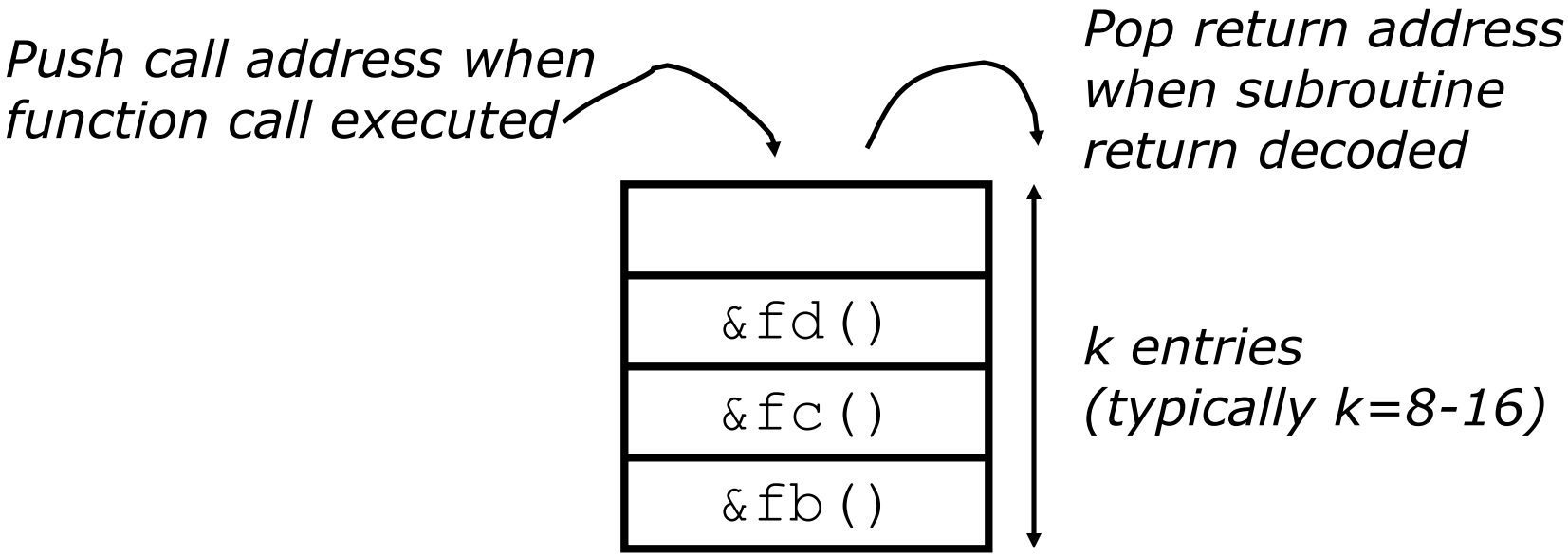
Push call address when function call executed



Subroutine Return Stack

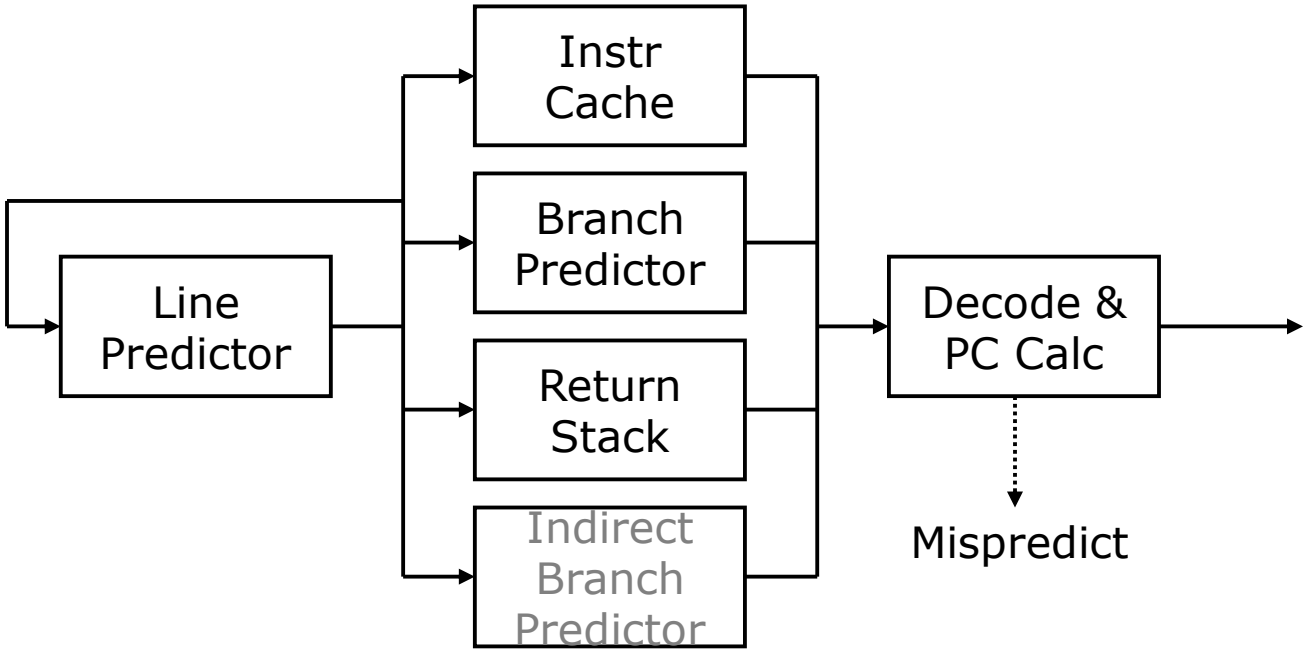
Small structure to accelerate JR for subroutine returns, typically much more accurate than BTBs.

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

fb() { fc(); }

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



Line Prediction
(*Alpha 21[234]64*)

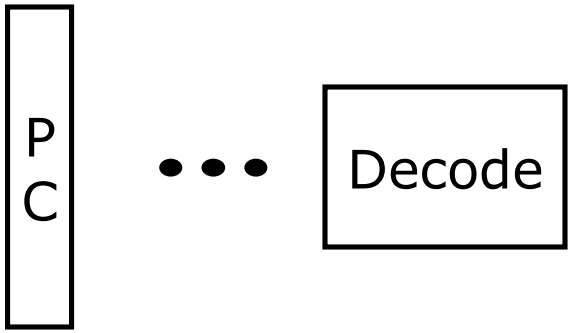
- For superscalar, useful to predict next cache line(s) to fetch



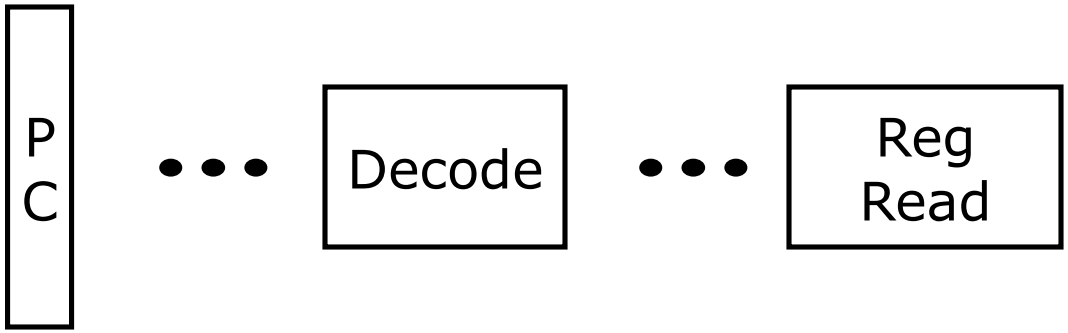
- Line Predictor predicts line to fetch each cycle (tight loop)
  - Untagged BTB structure – Why?
  - 21464 was to predict 2 lines per cycle
- Icache fetches block, and predictors improve target prediction
- PC Calc checks accuracy of line prediction(s)

Overview of Branch Prediction

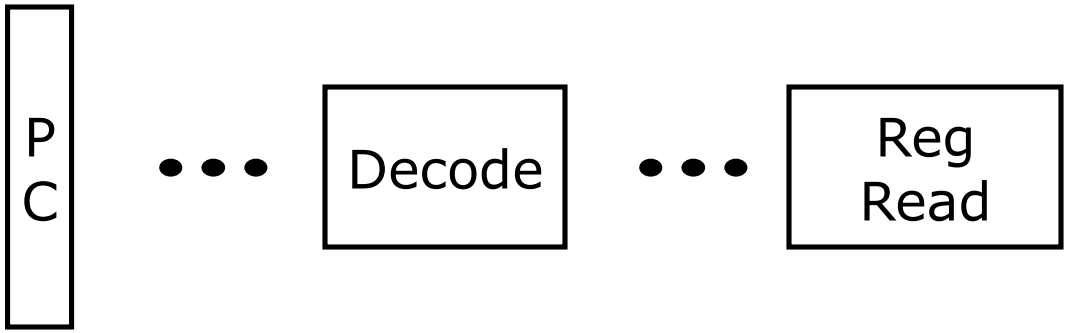
Overview of Branch Prediction



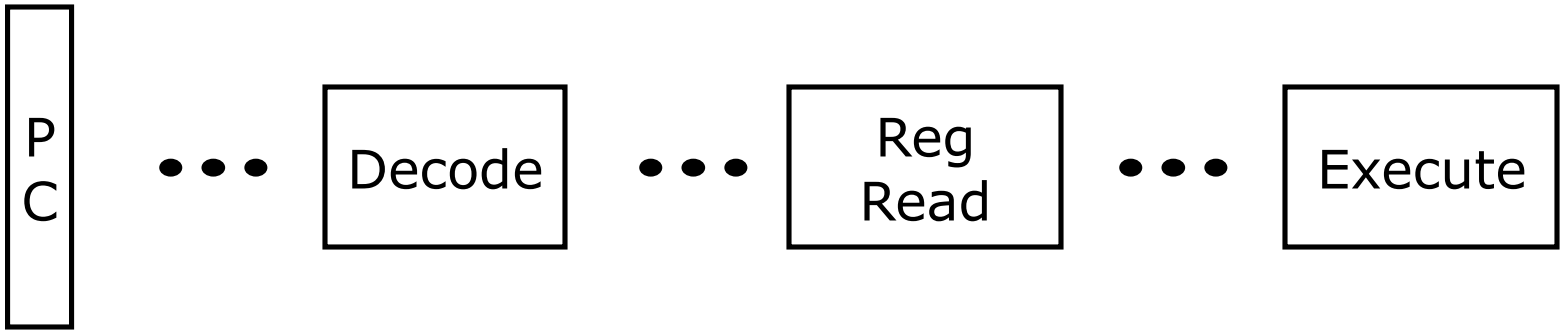
Overview of Branch Prediction



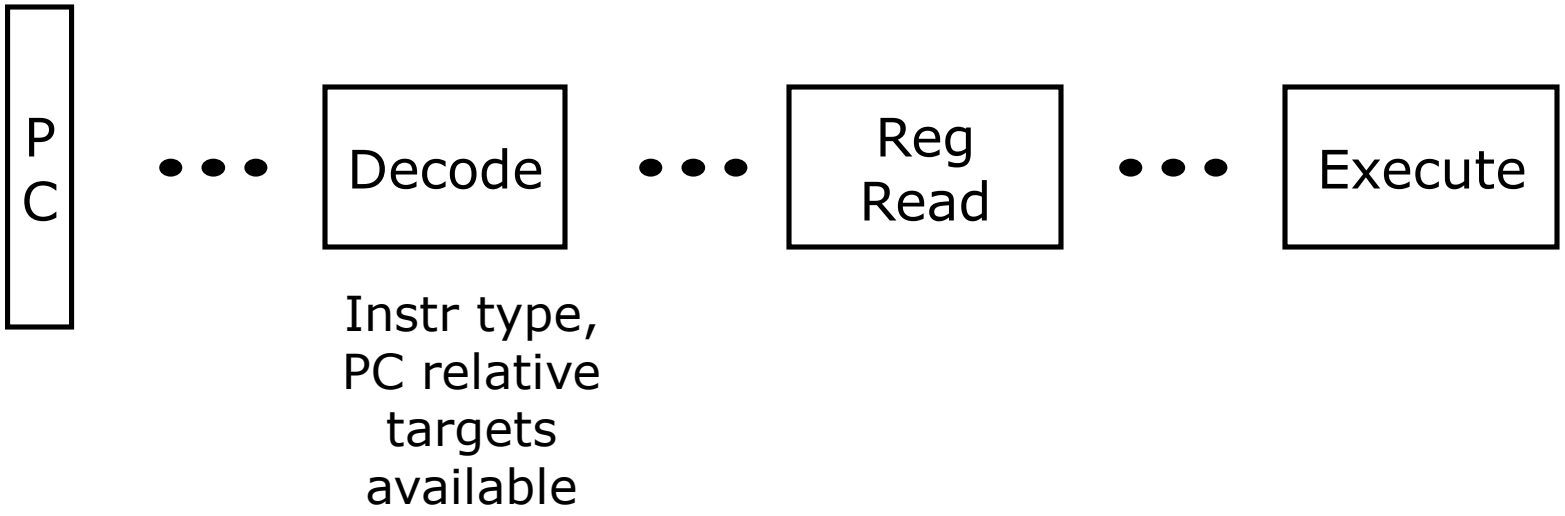
Overview of Branch Prediction



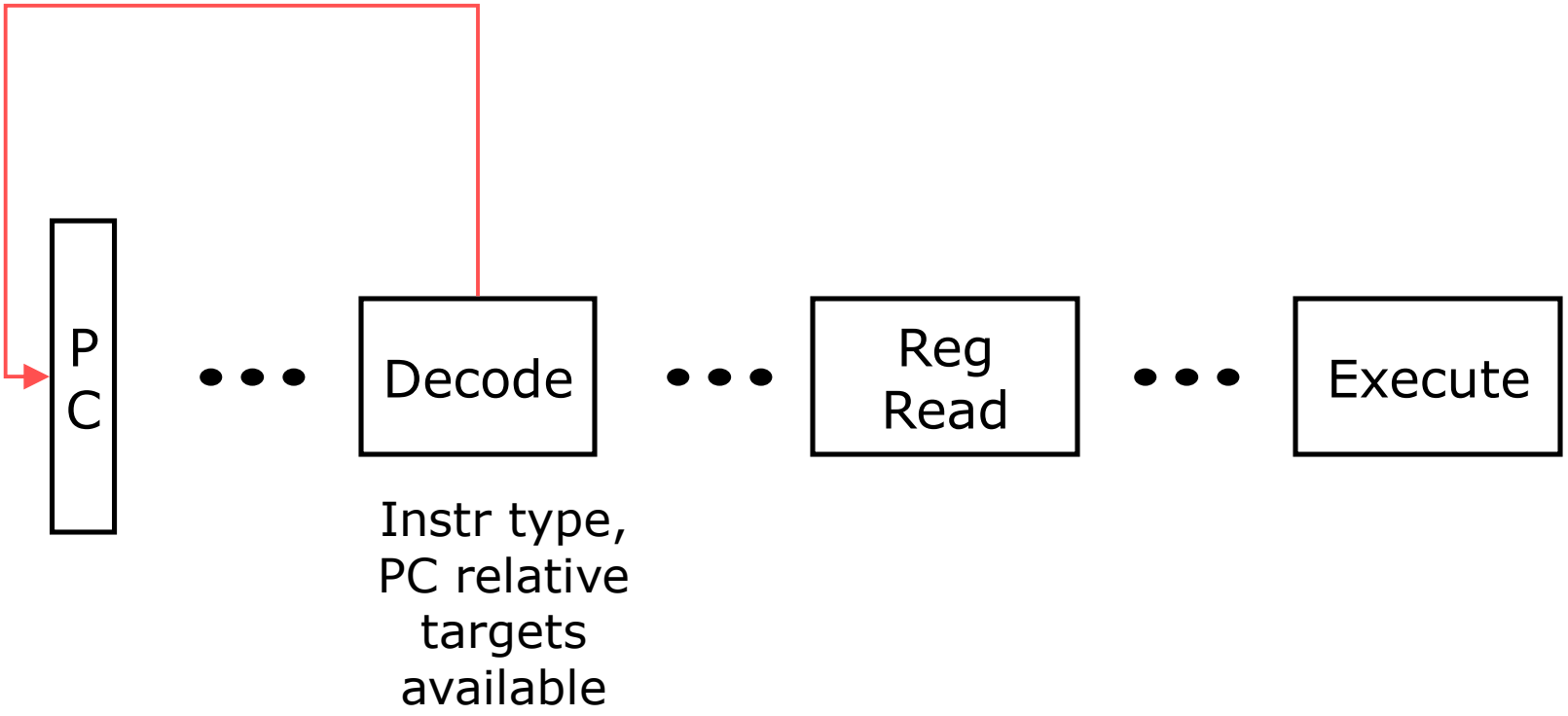
Overview of Branch Prediction



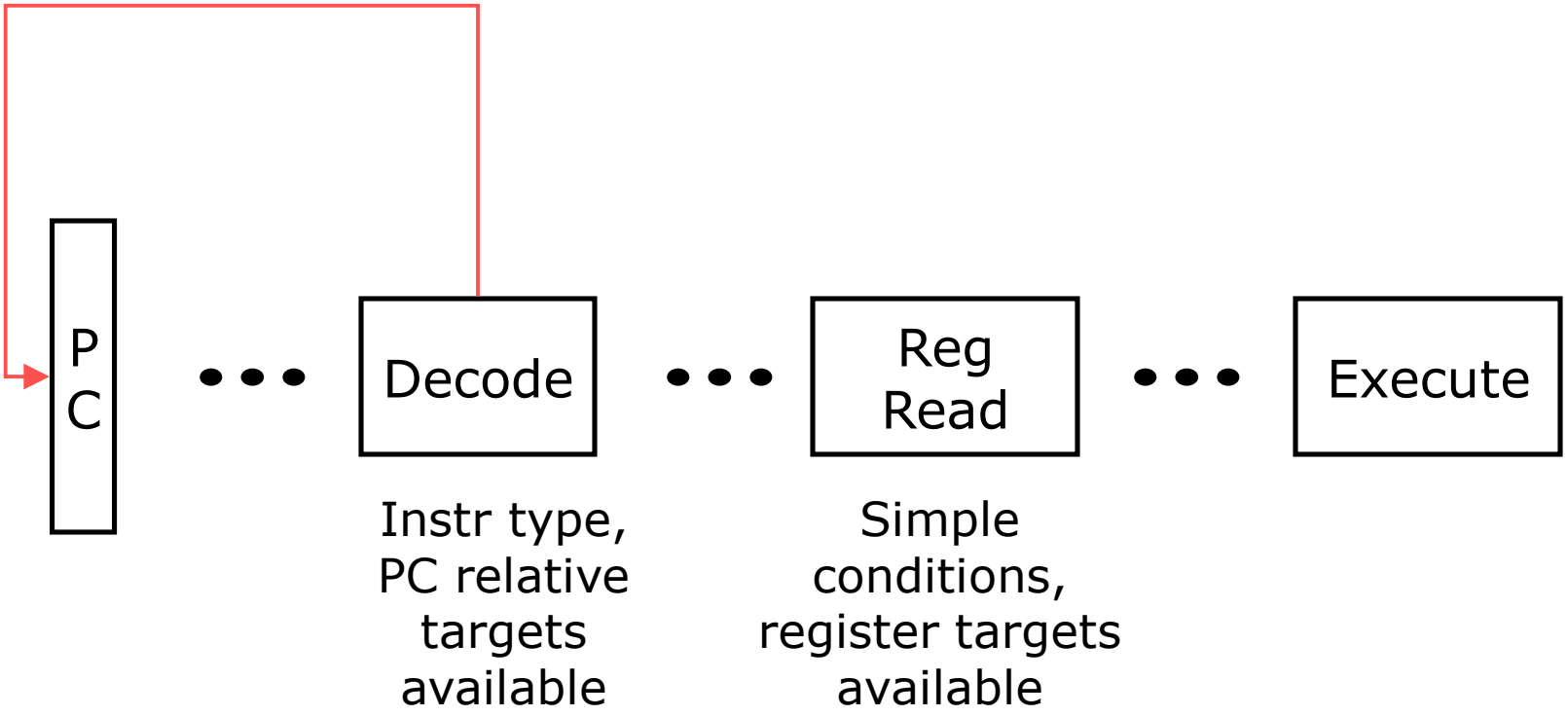
Overview of Branch Prediction



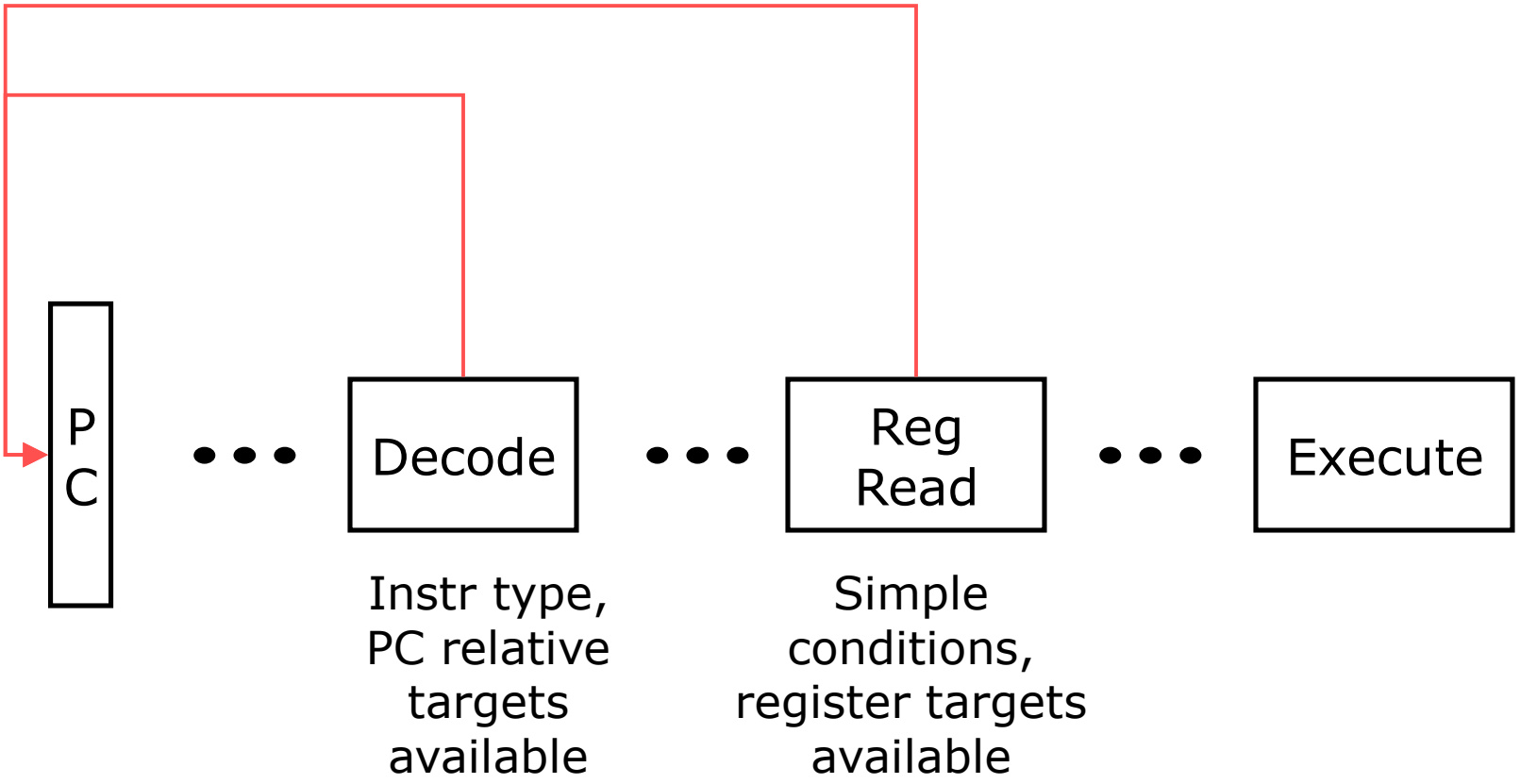
Overview of Branch Prediction



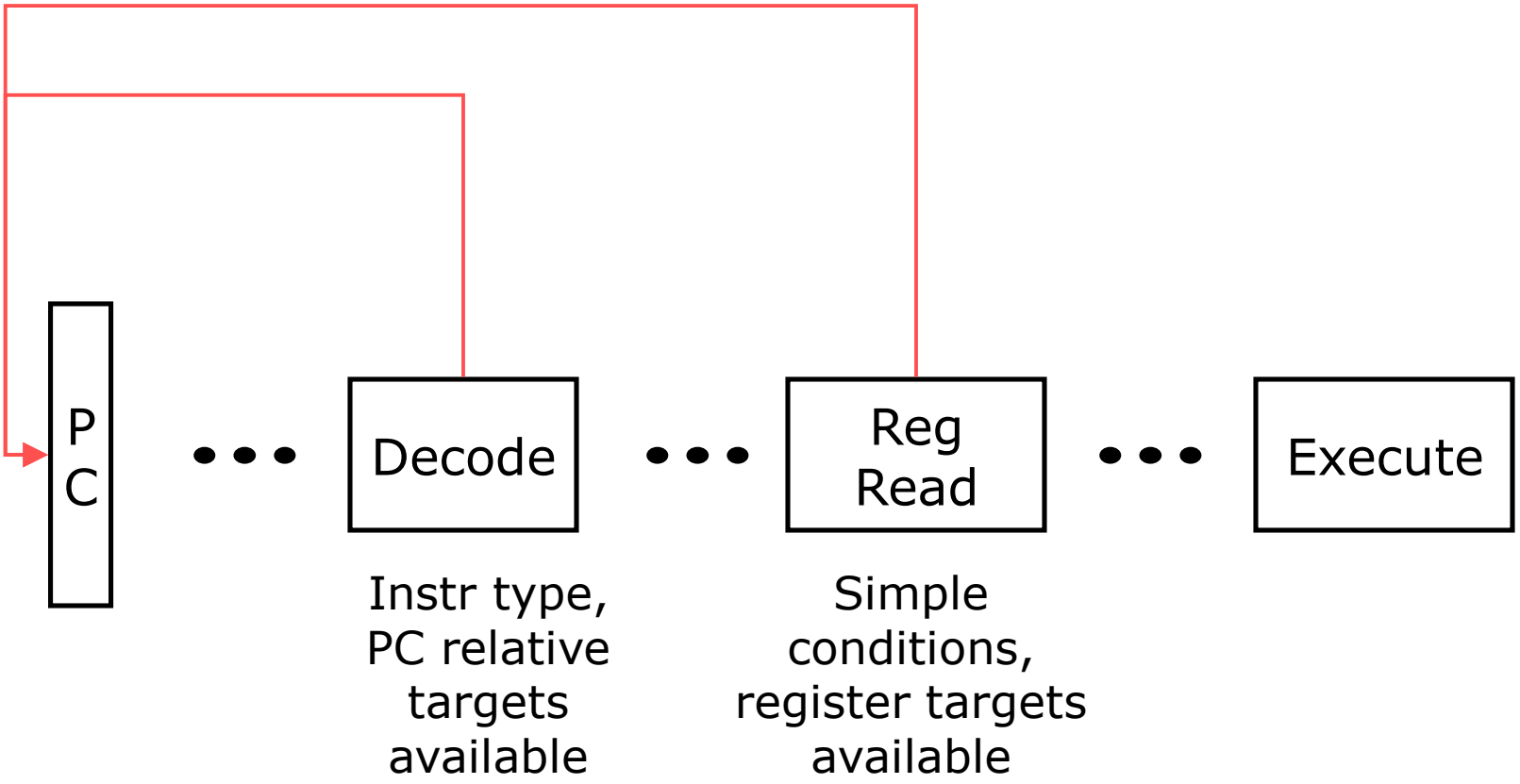
Overview of Branch Prediction



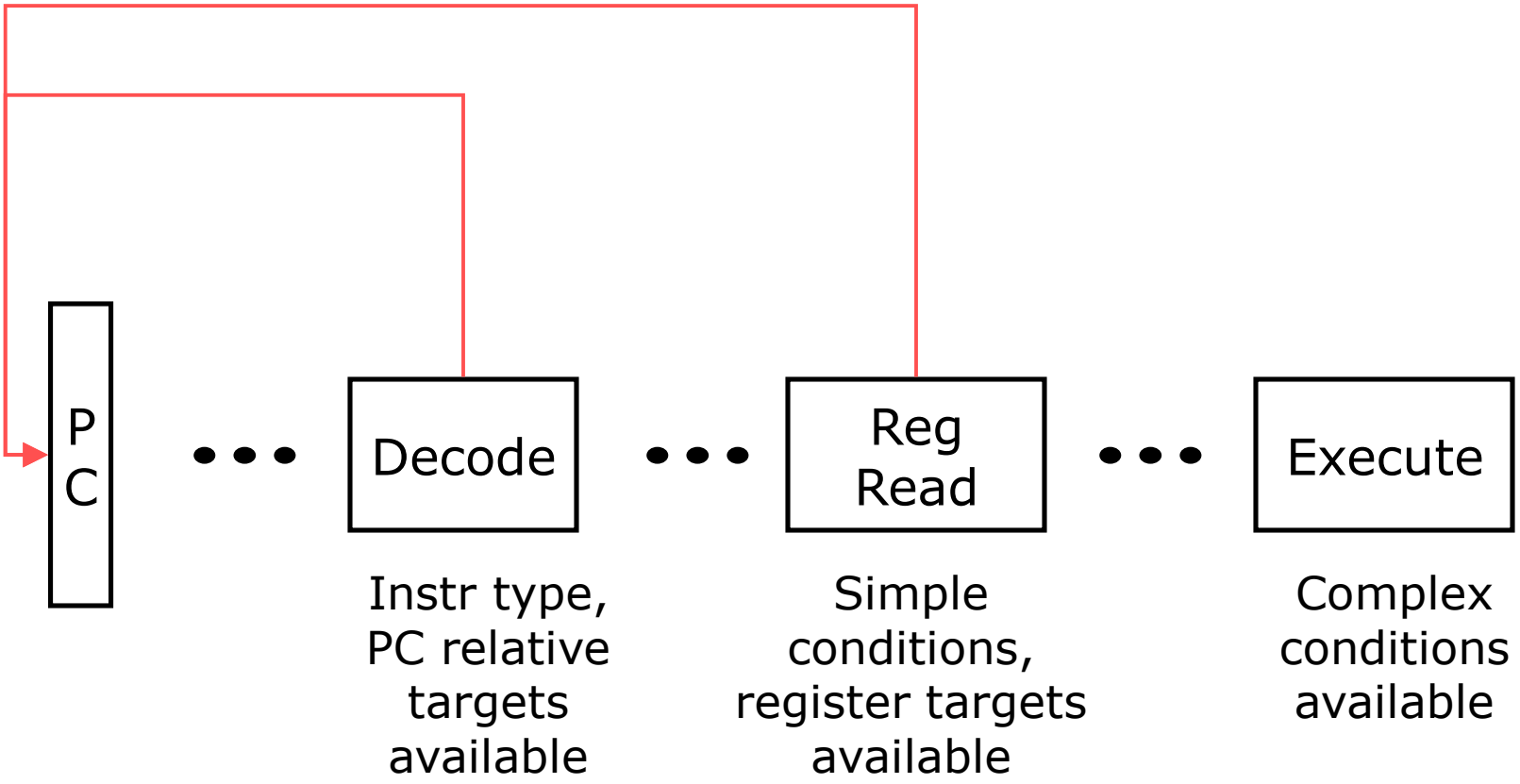
Overview of Branch Prediction



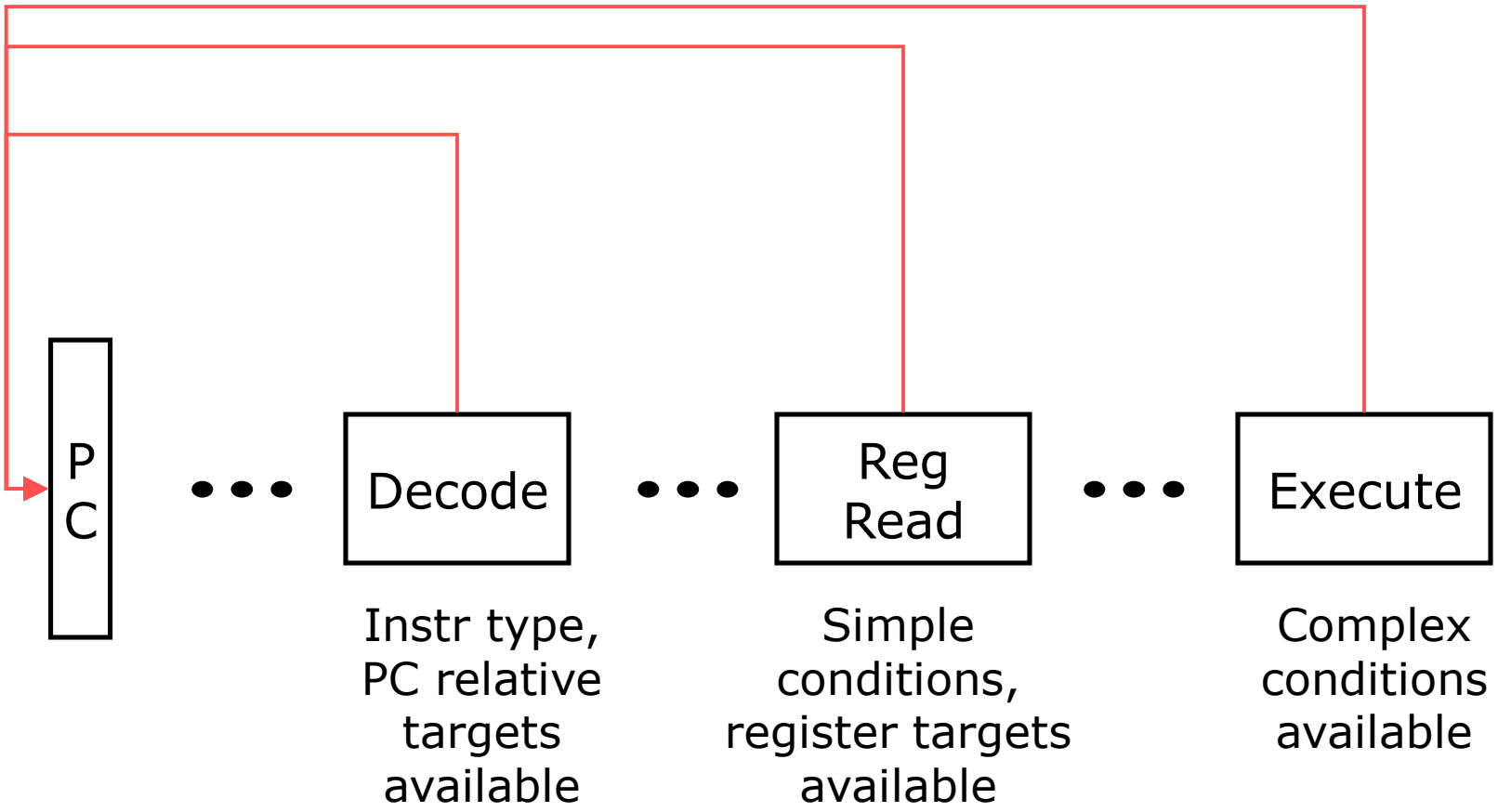
Overview of Branch Prediction



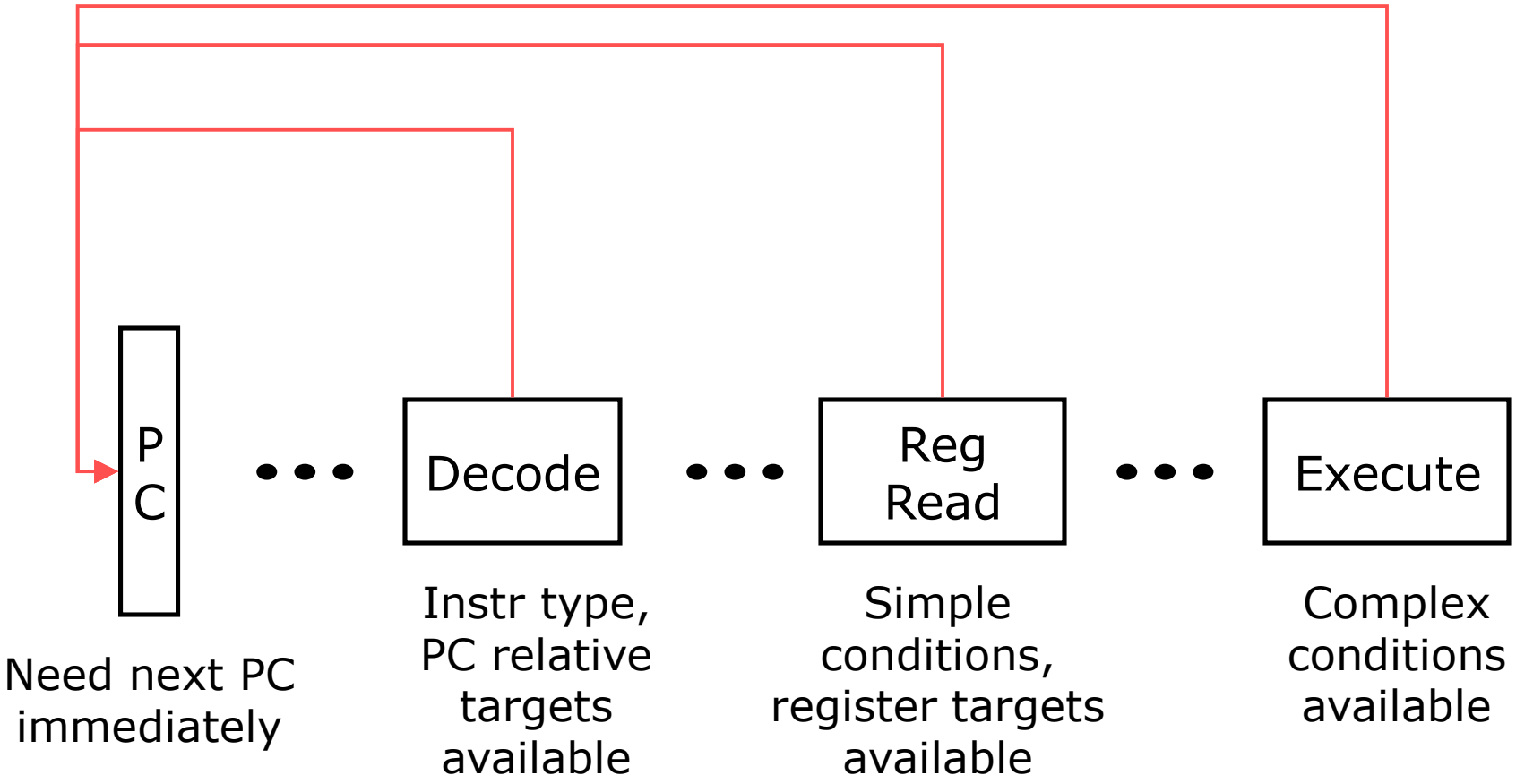
Overview of Branch Prediction



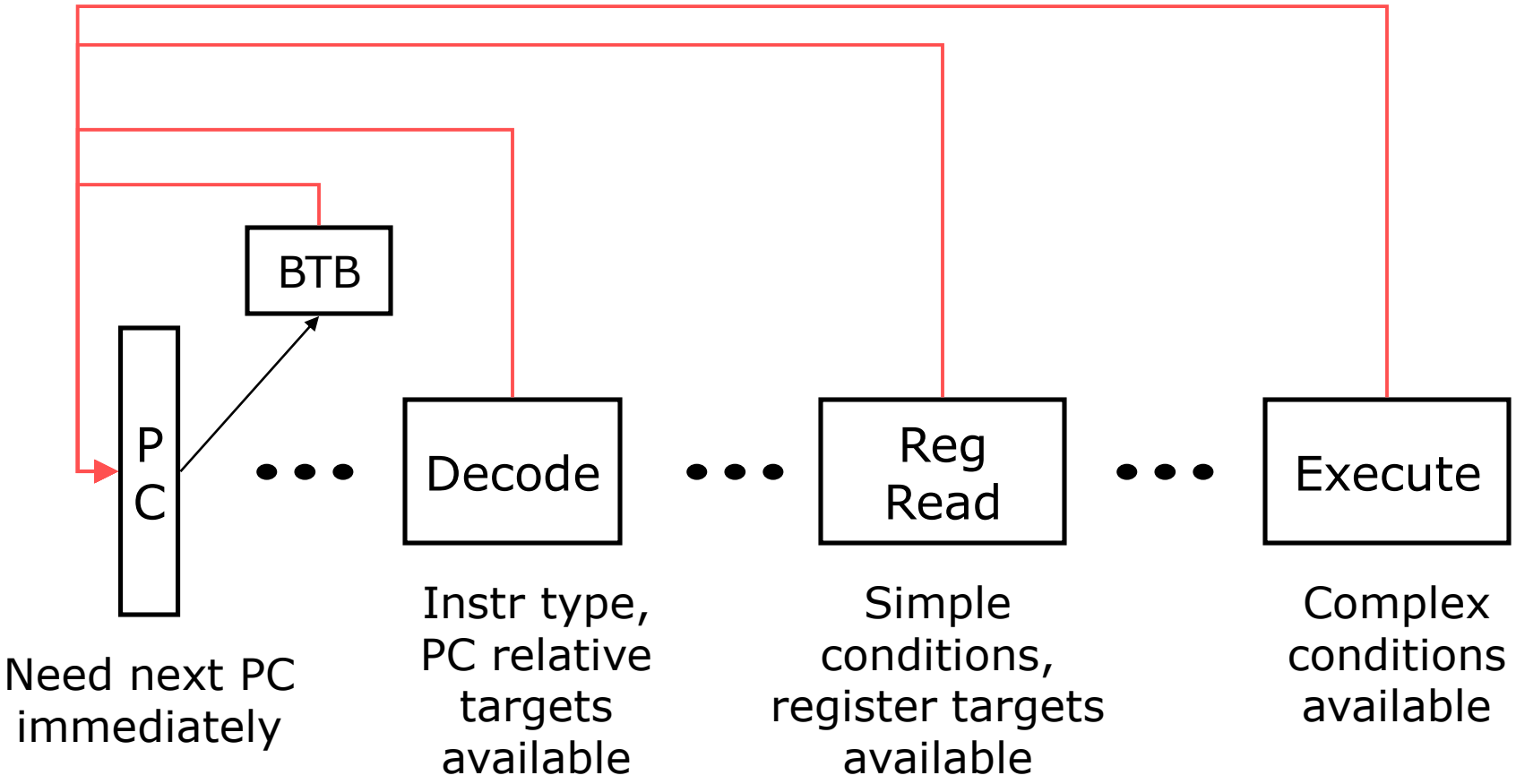
Overview of Branch Prediction



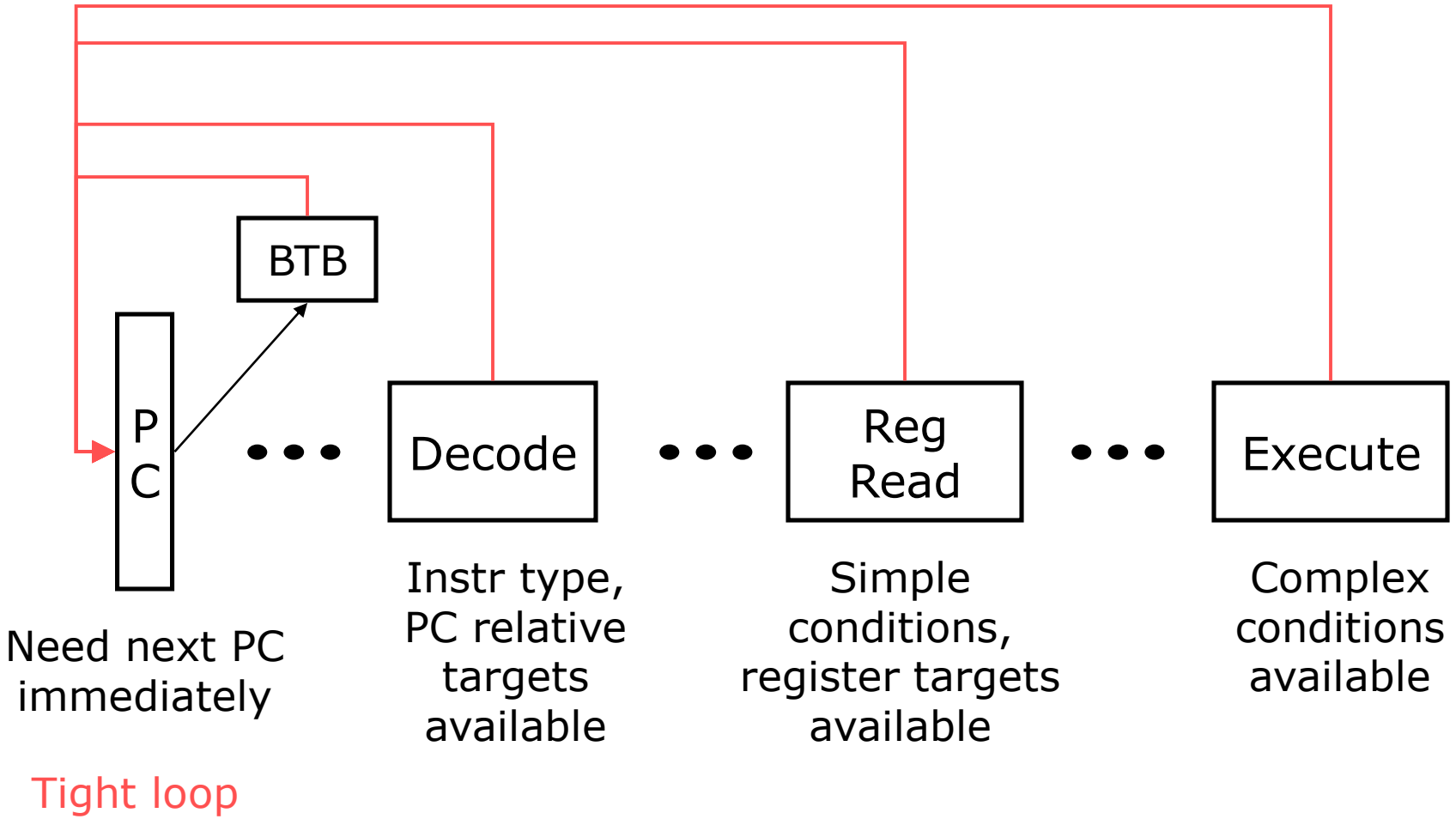
Overview of Branch Prediction



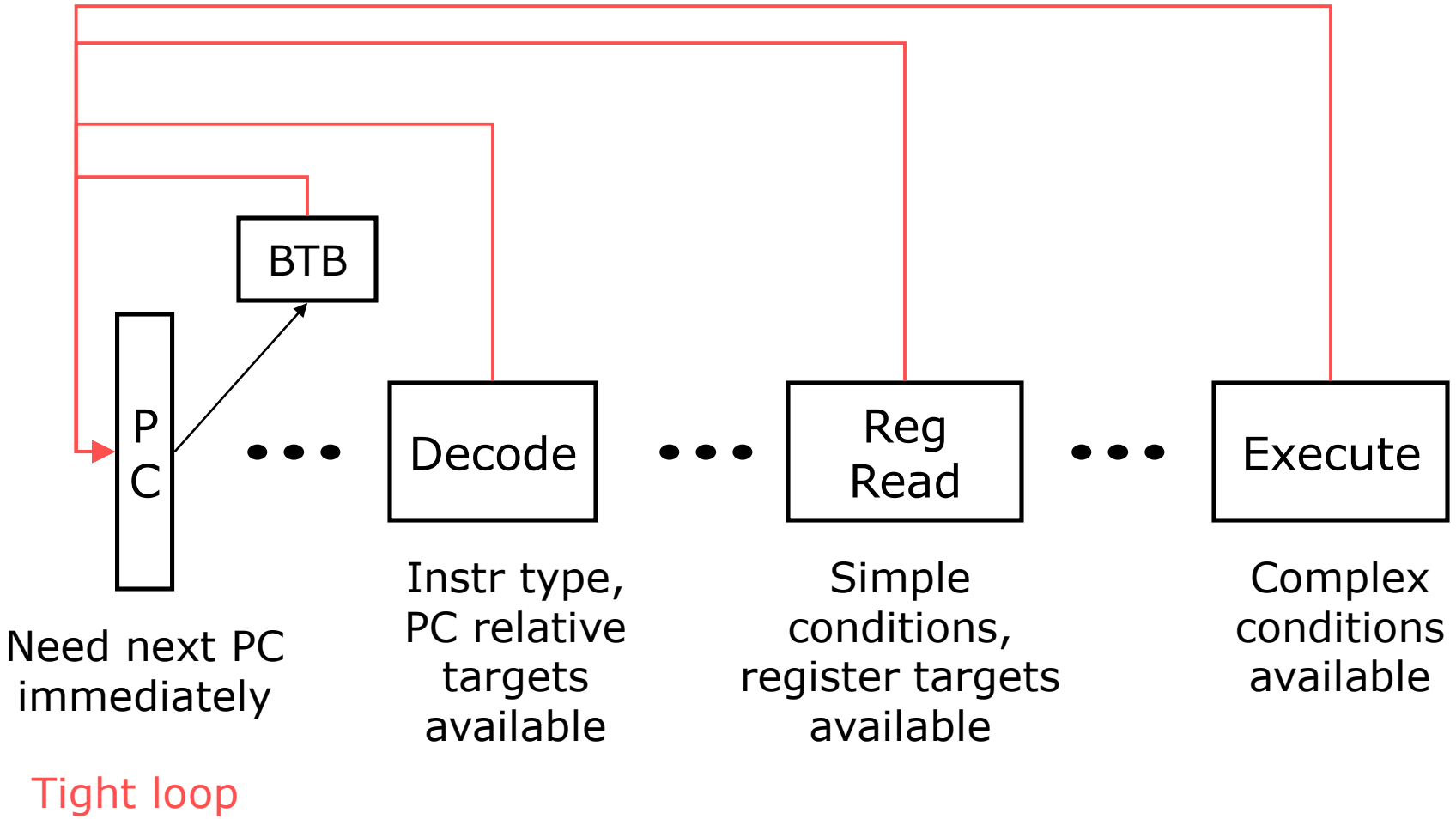
Overview of Branch Prediction



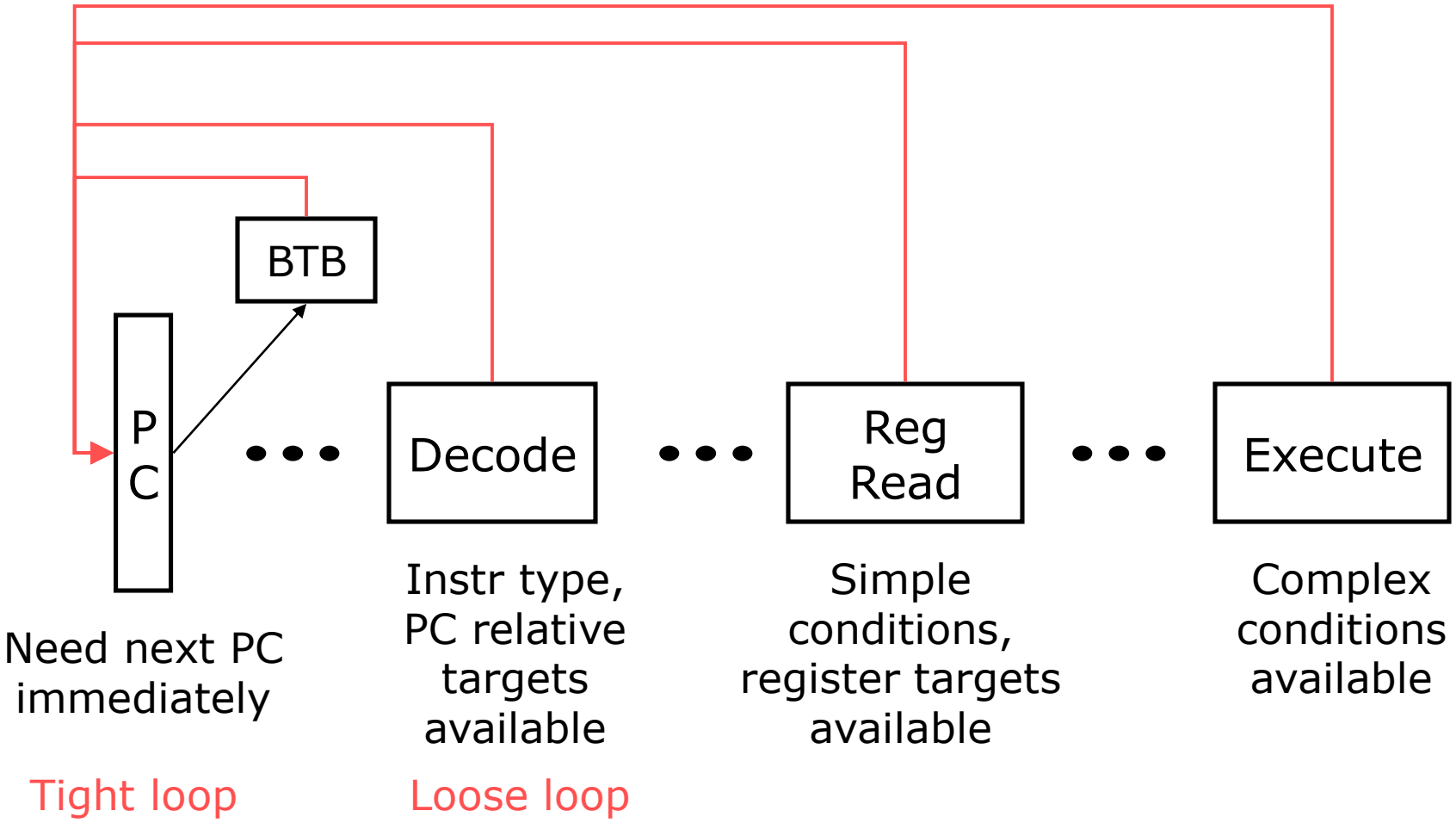
Overview of Branch Prediction



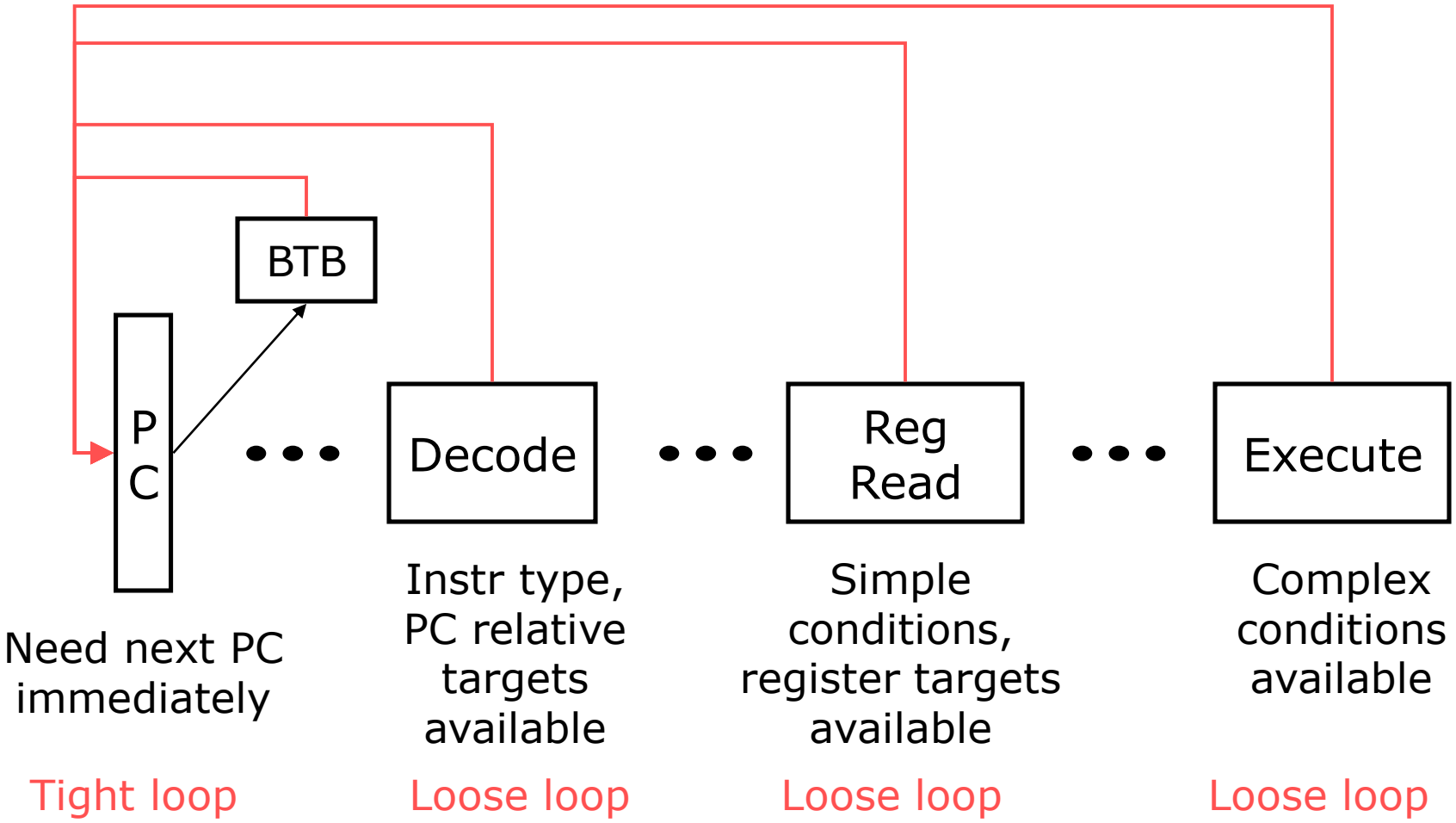
Overview of Branch Prediction



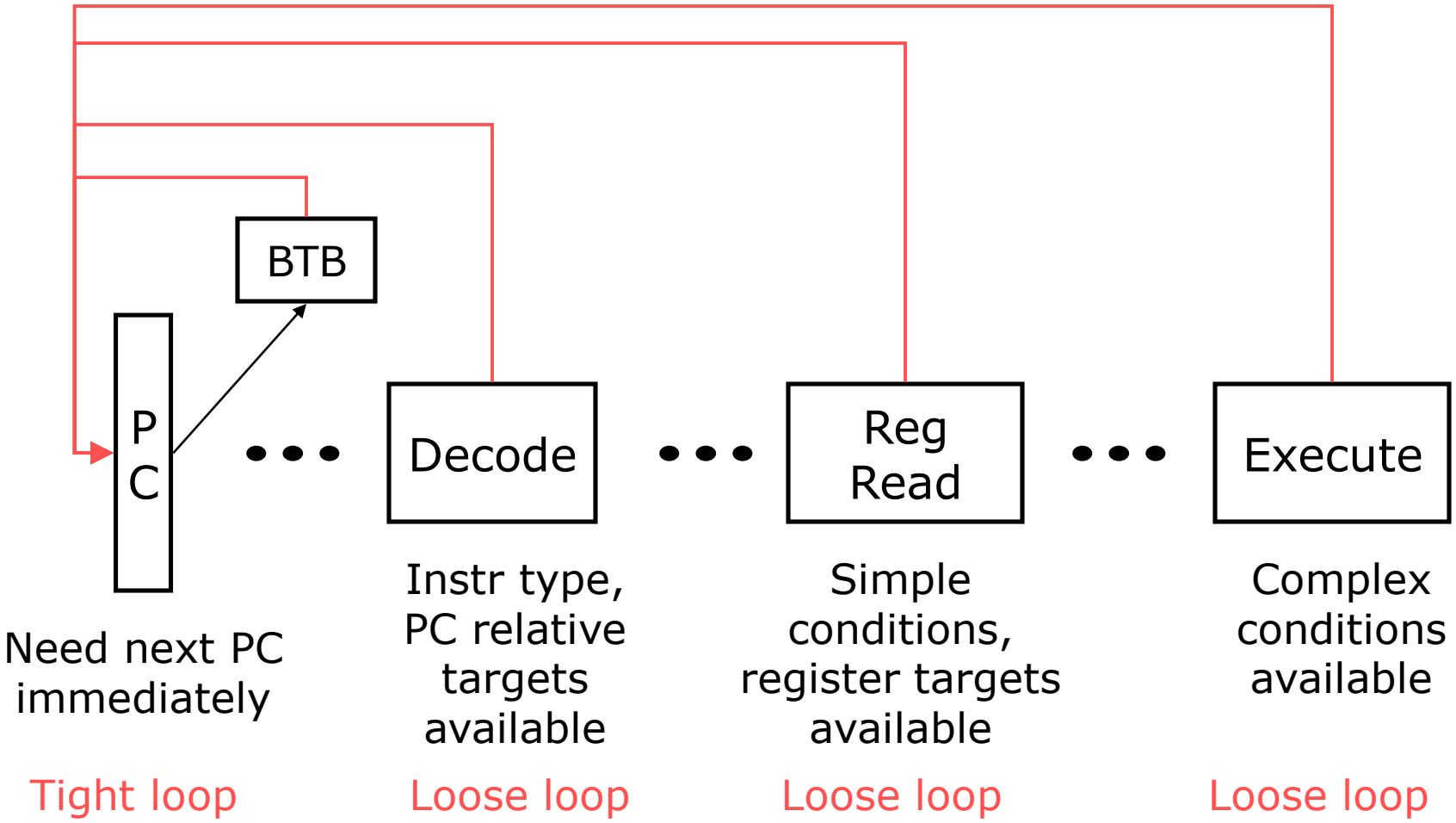
Overview of Branch Prediction



Overview of Branch Prediction

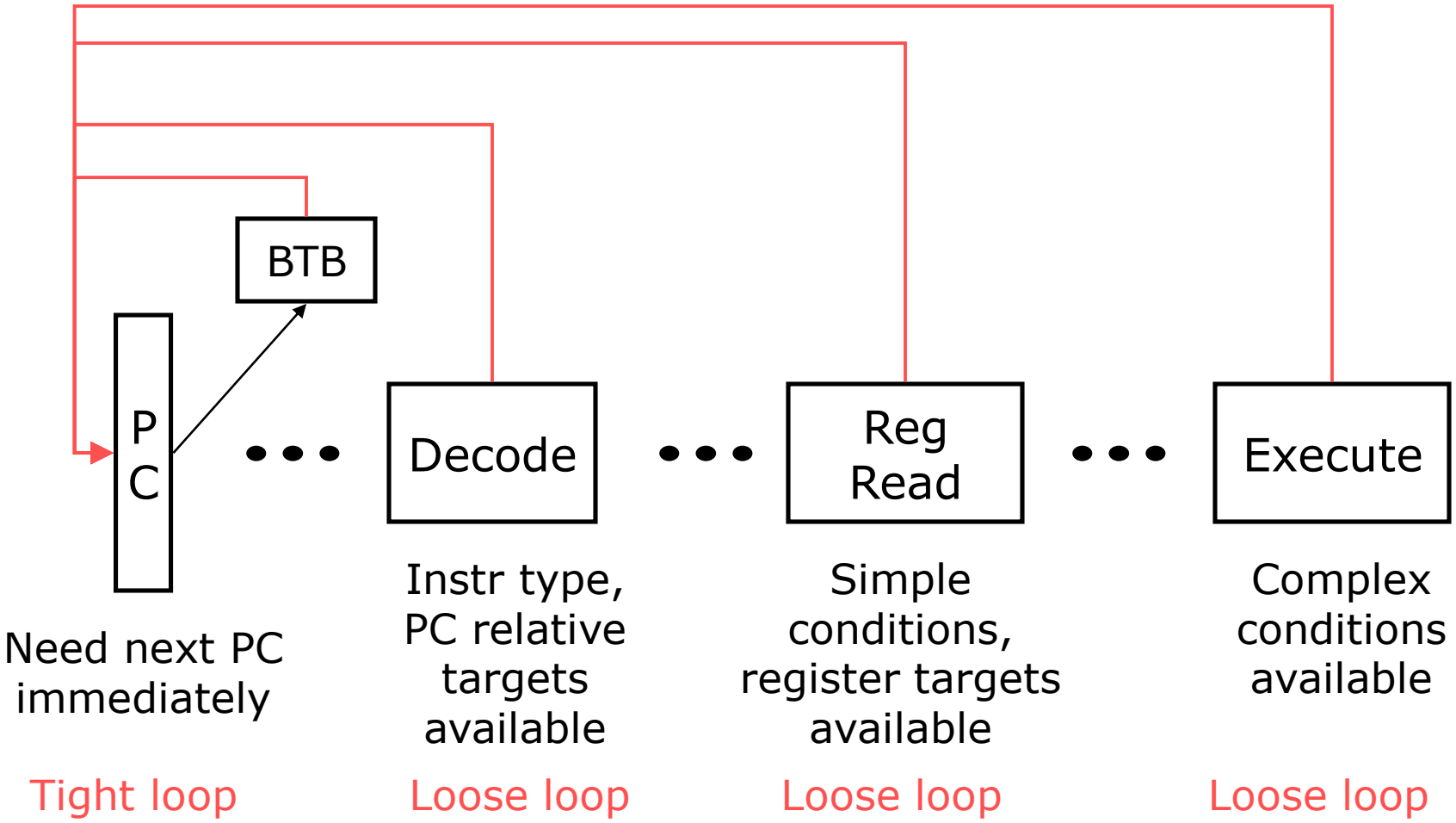


Overview of Branch Prediction



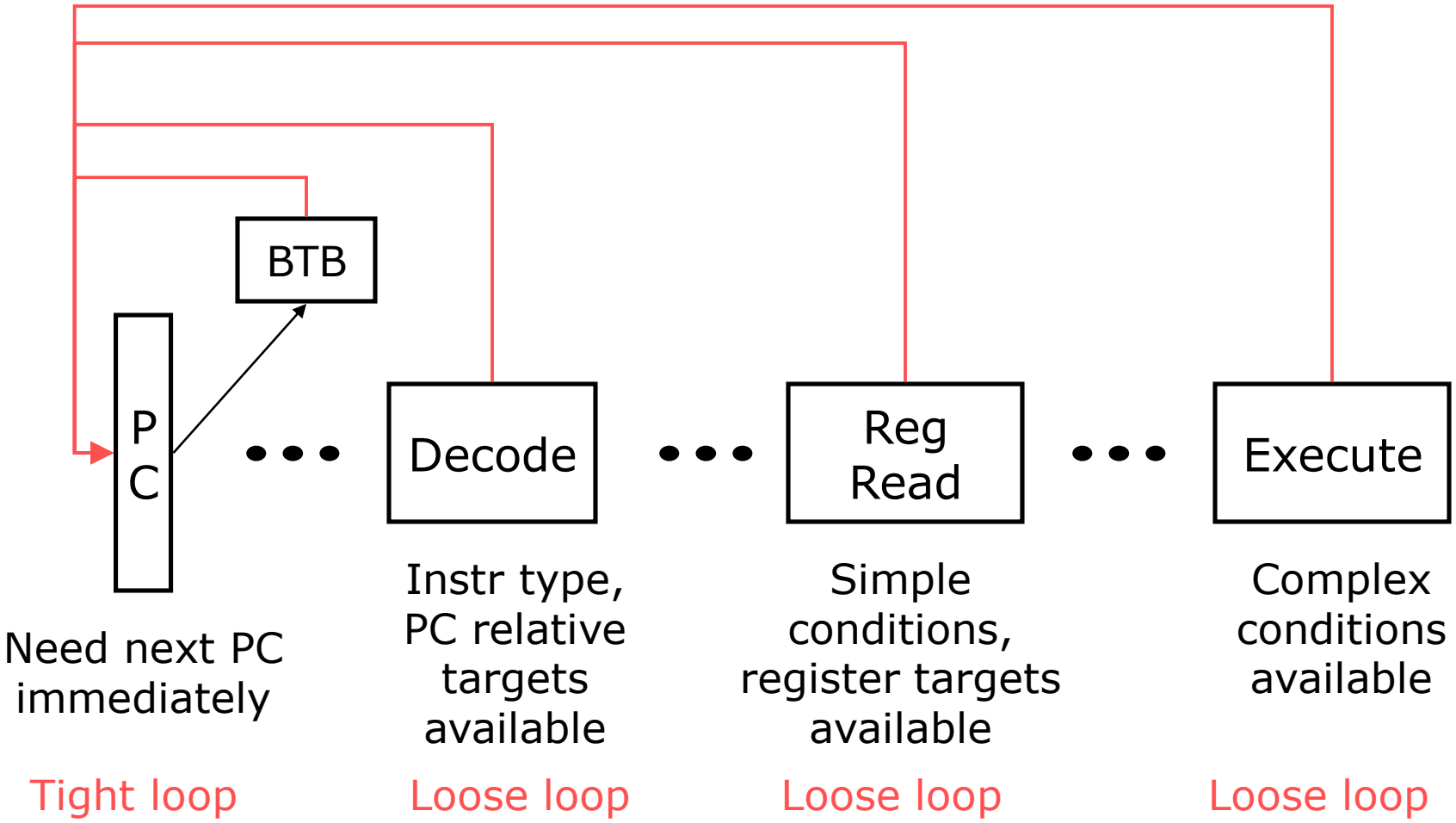
Must speculation check always be correct?

Overview of Branch Prediction



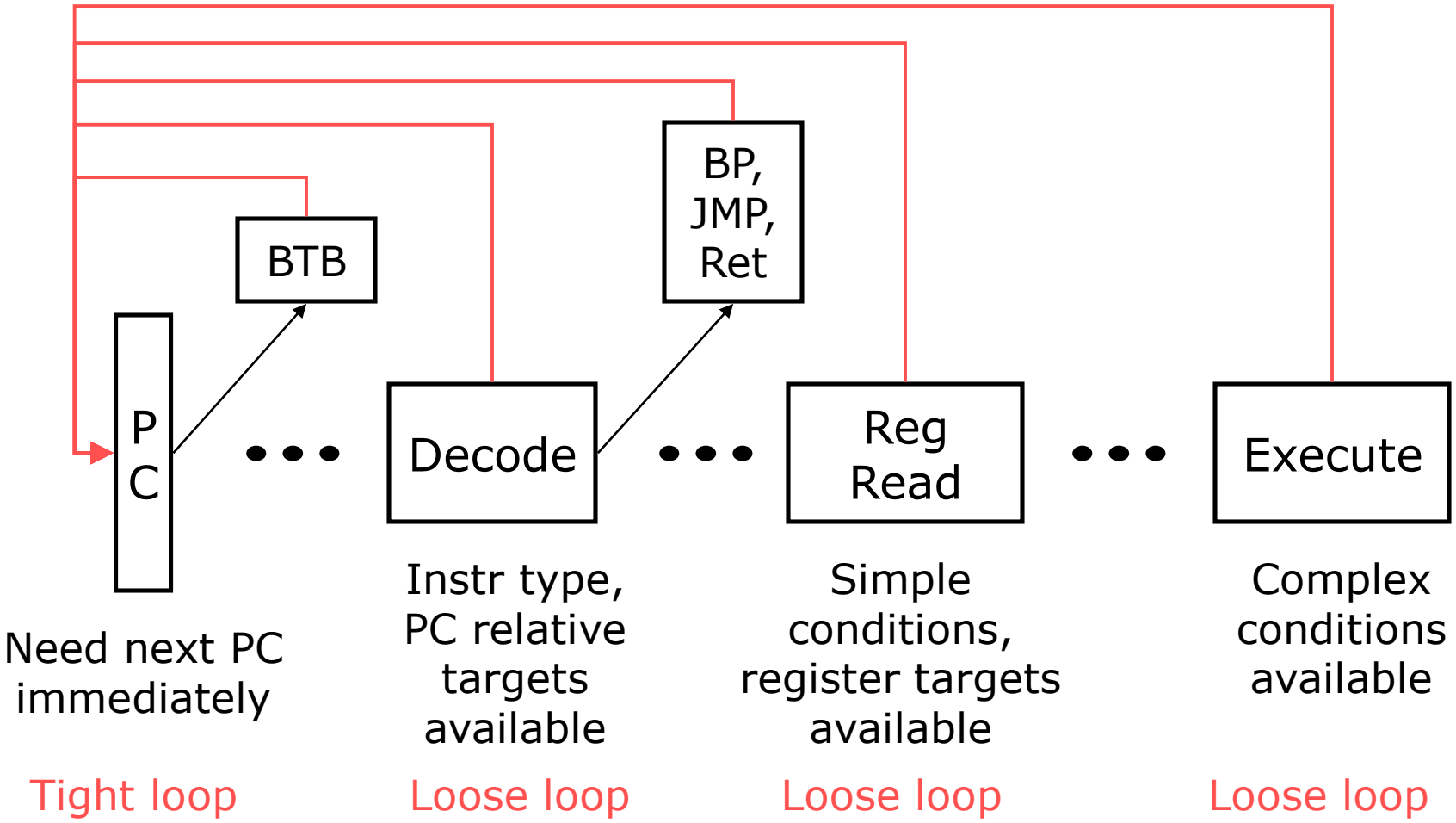
Must speculation check always be correct? No…

Overview of Branch Prediction



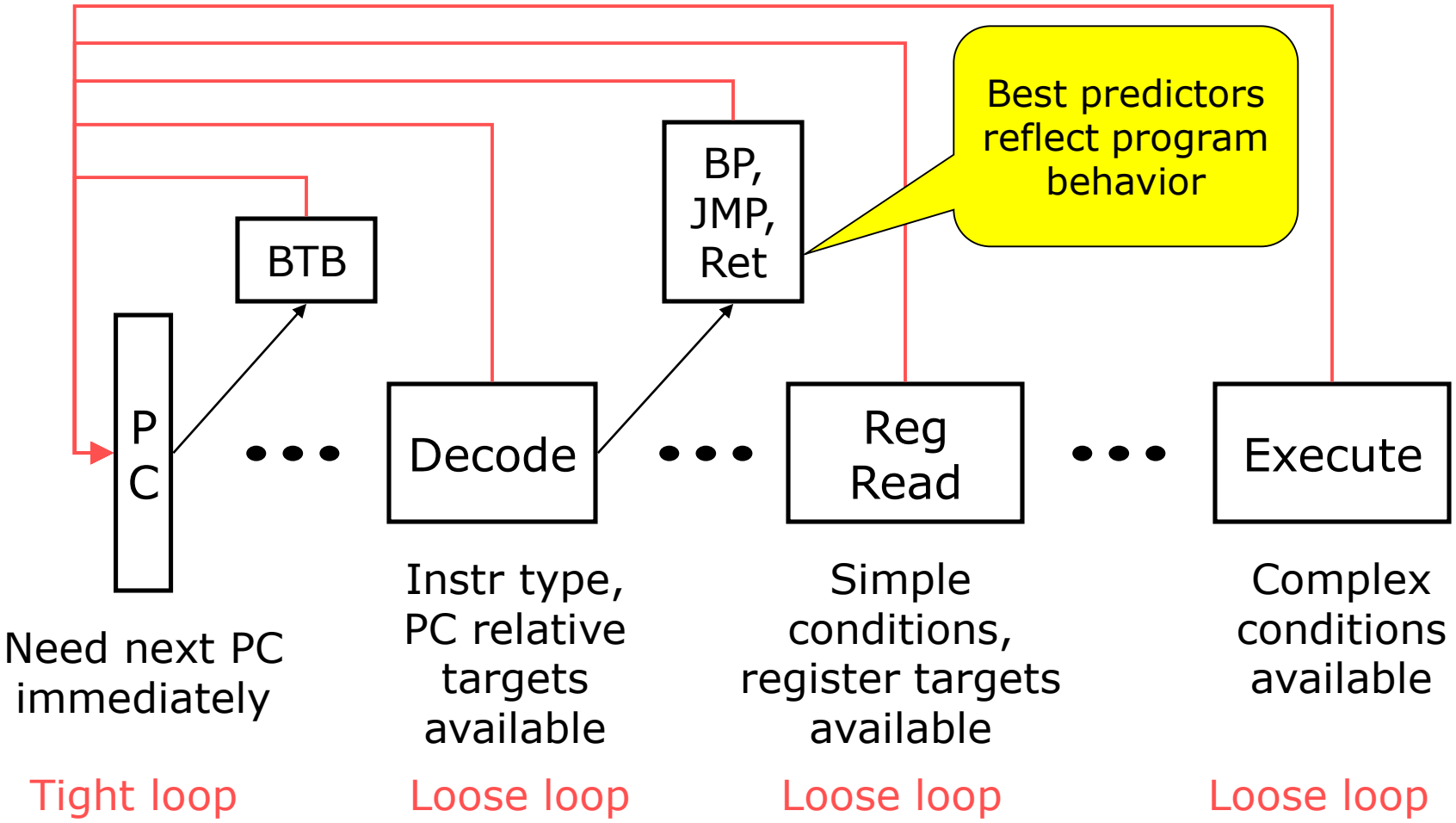
Must speculation check always be correct? No…

Overview of Branch Prediction



Must speculation check always be correct? No…

Overview of Branch Prediction



Must speculation check always be correct? No…

Next Lecture: Speculative Execution & Value Management