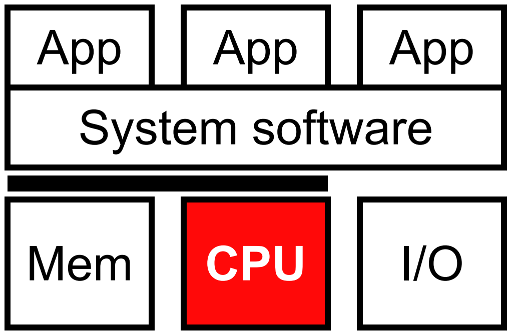
Computer Architecture

Lec 6: Branch Prediction

This Lecture



- Control Dependences
- Branch Prediction

Control Dependences

Control Dependence

- Question: What should the fetch PC be in the next cycle?
- Answer: The address of the next instruction
  - ❑ All instructions are control dependent on previous ones. Why?

- If the fetched instruction is a non-control-flow instruction:
  - ❑ Next Fetch PC is the address of the next-sequential instruction
  - ❑ Easy to determine if we know the size of the fetched instruction

- If the instruction that is fetched is a control-flow instruction:
  - ❑ How do we determine the next Fetch PC?

- In fact, how do we even know whether or not the fetched instruction is a control-flow instruction?

Branch Types

Type	**Direction at fetch time**	**Number of possible next fetch addresses?**	**When is next fetch address resolved?**
Conditional (beq)	Unknown	2	Execution (register dependent)
Unconditional (j)	Always taken	1	Decode (PC + offset)
Call (jal X)	Always taken	1	Decode (PC + offset)
Return (jr $ra)	Always taken	Many	Execution (register dependent)
Indirect (jr $r1)	Always taken	Many	Execution (register dependent)

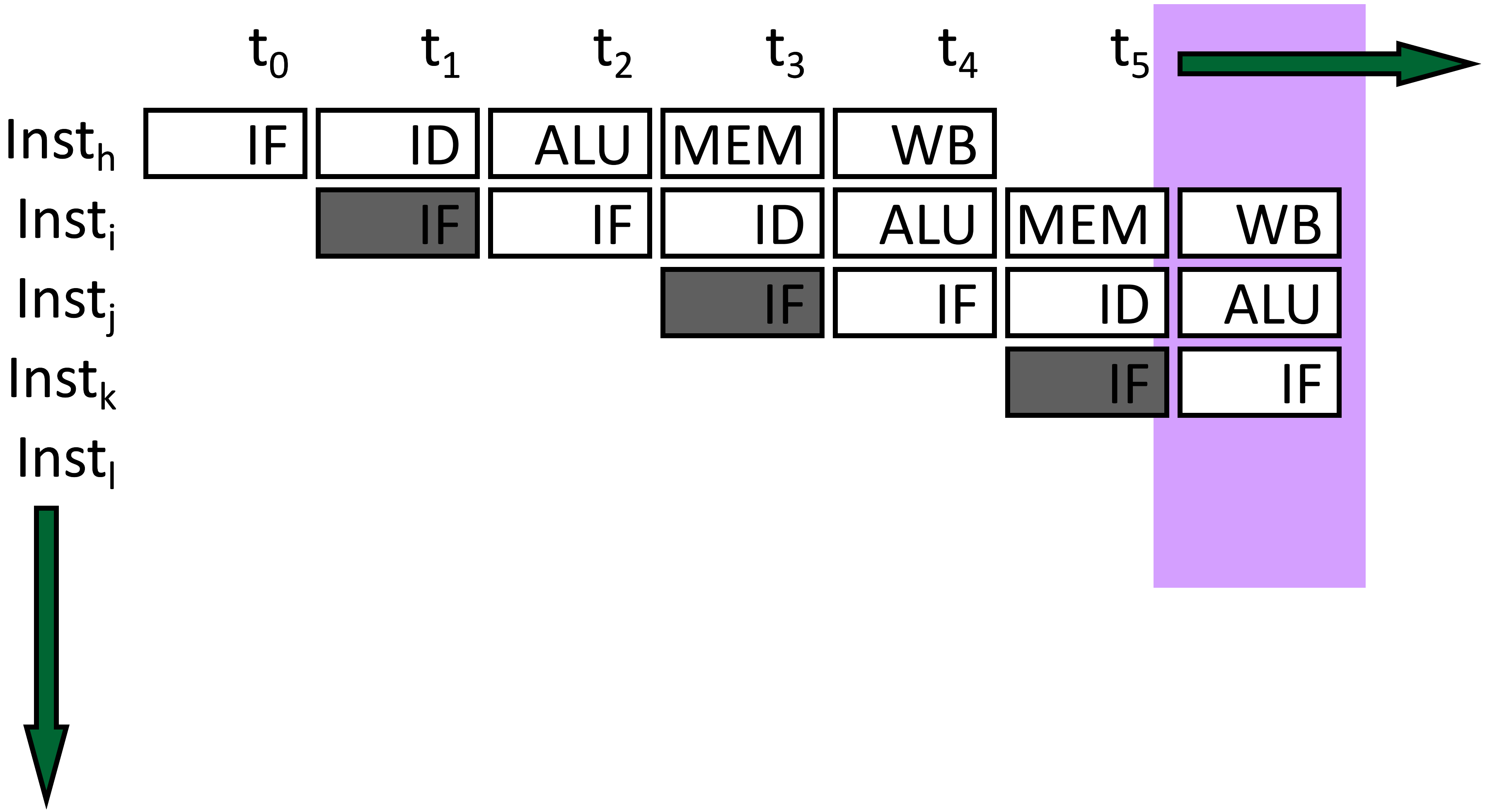
How to Handle Control Dependences

- Critical to keep the pipeline full with correct sequence of dynamic instructions.

- Potential solutions if the instruction is a control-flow instruction:

- Stall the pipeline until we know the next fetch address
- Guess the next fetch address (branch prediction)
- Employ delayed branching (branch delay slot)
- Do something else (fine-grained multithreading)
- Eliminate control-flow instructions (predicated execution)
- Fetch from both possible paths (if you know the addresses of both possible paths) (multipath execution)

Stall Fetch Until Next PC is Known: Good Idea?



t_0 t_1 t_2 t_3 t_4 t_5
$Inst_h$ IF ID ALU MEM WB
$Inst_i$ IF IF ID ALU MEM WB
$Inst_j$ IF IF ID ALU
$Inst_k$ IF IF
$Inst_l$

This is the case with non-control-flow and unconditional br instructions!

The Branch Problem

- Control flow instructions (branches) are frequent
  - 15-25% of all instructions

- Problem: Next fetch address after a control-flow instruction is not determined after N cycles in a pipelined processor
  - N cycles: (minimum) branch resolution latency

- If we are fetching W instructions per cycle (i.e., if the pipeline is W wide)
  - A branch misprediction leads to **N x W** wasted instruction slots

Importance of The Branch Problem

- Assume N = 20 (20 pipe stages), W = 5 (5 wide fetch)
  - Assume: 1 out of 5 instructions is a branch

- How long does it take to fetch 500 instructions?
  - 100% accuracy
    - 100 cycles (all instructions fetched on the correct path)
    - No wasted work; IPC = 500/100
  - 99% accuracy
    - 100 (correct path) + 20 * 1 (wrong path) = 120 cycles
    - 20% extra instructions fetched; IPC = 500/120
  - 90% accuracy
    - 100 (correct path) + 20 * 10 (wrong path) = 300 cycles
    - 200% extra instructions fetched; IPC = 500/300
  - 60% accuracy
    - 100 (correct path) + 20 * 40 (wrong path) = 900 cycles
    - 800% extra instructions fetched; IPC = 500/900

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

Branch Prediction

How to Handle Control Dependences

- Critical to keep the pipeline full with correct sequence of dynamic instructions.

- Potential solutions if the instruction is a control-flow instruction:

- Stall the pipeline until we know the next fetch address
- Guess the next fetch address (branch prediction)
- Employ delayed branching (branch delay slot)
- Do something else (fine-grained multithreading)
- Eliminate control-flow instructions (predicated execution)
- Fetch from both possible paths (if you know the addresses of both possible paths) (multipath execution)

Branch Speculation

Without Perdiction:

	1	2	3	4	5	6	7	8	9
`addi r3←r1,1`	F	D	X	M	W				
`bnez r3,targ`		F	D	**X**	M	W			
`stall`			--	--	--	--	--		
`stall`				--	--	--	--	--	
`st r6→[r7+4]`					**F**	D	X	M	W
`mul r10←r8,r9`									
`targ:add r4←r4,r5`									

Correct Prediction:

	1	2	3	4	5	6	7	8	9
`addi r3←r1,1`	F	D	X	M	W				
`bnez r3,targ`		F	D	X	M	W			
`st r6→[r7+4]`			**F**	**D**	X	M	W		
`mul r10←r8,r9`				**F**	D	X	M	W	

speculative

Correct Prediction Saves 2 Cycles!

Mis-prediction Penalty

		1	2	3	4	5	6	7	8	9
Recovery:	`addi r3←r1,1`	F	D	X	M	W				
	`bnez r3,targ`		F	D	**X**	M	W			
	~~`st r6→[r7+4]`~~			**F**	**D**	--	--	--		
	~~`mul r10←r8,r9`~~				**F**	--	--	--	--	
	`targ:add r4←r4,r5`					**F**	D	X	M	W

- **Mis-speculation recovery**: what to do on wrong guess
  - Not too painful in a short, in-order pipeline
  - Branch resolves in X
  - + Younger insns (in F, D) haven't changed permanent state
  - **Flush** insns currently in D and X (i.e., replace with **`nops`**)

Performance Analysis

- correct guess ⇒ no penalty ~86% of the time
- incorrect guess ⇒ 2 bubbles
- Assume
  - no data dependency related stalls
  - 20% control flow instructions
  - 70% of control flow instructions are taken
  - CPI = [1 + (0.20*<u>0.7</u>) * 2] =

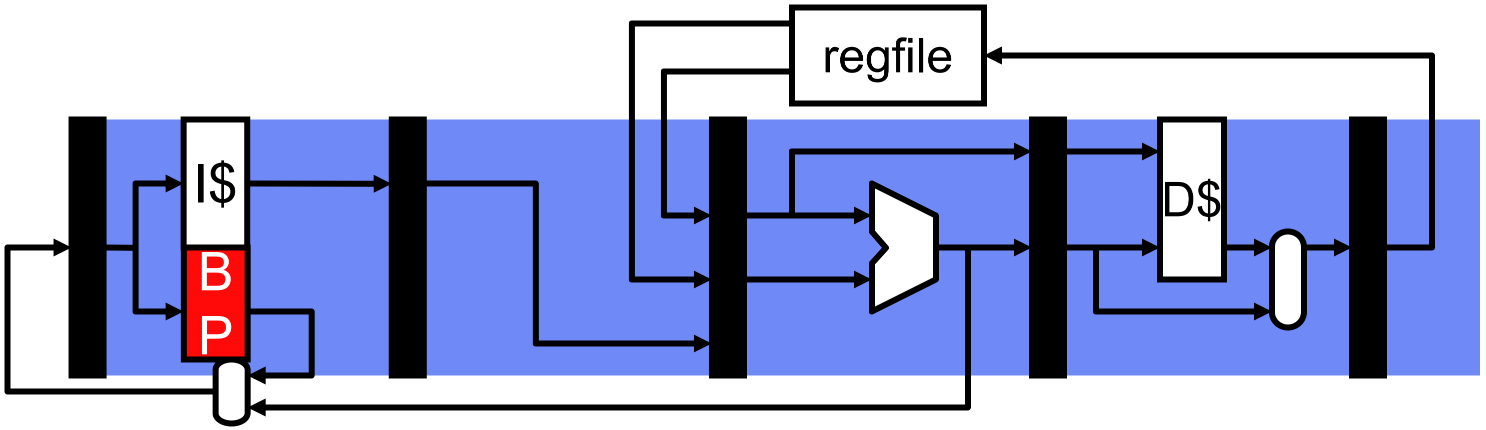
    = [1 + 0.14 * 2] = 1.28

probability of a wrong guess

penalty for a wrong guess

Can we reduce either of the two penalty terms?

Branch Prediction Components



- Step #1: is it a branch?
    - Easy after decode...
- Step #2: is the branch taken or not taken?
    - Direction predictor (applies to conditional branches only)
    - Predicts taken/not-taken
- Step #3: if the branch is taken, where does it go?
    - Easy after decode…

When to Perform Branch Prediction?

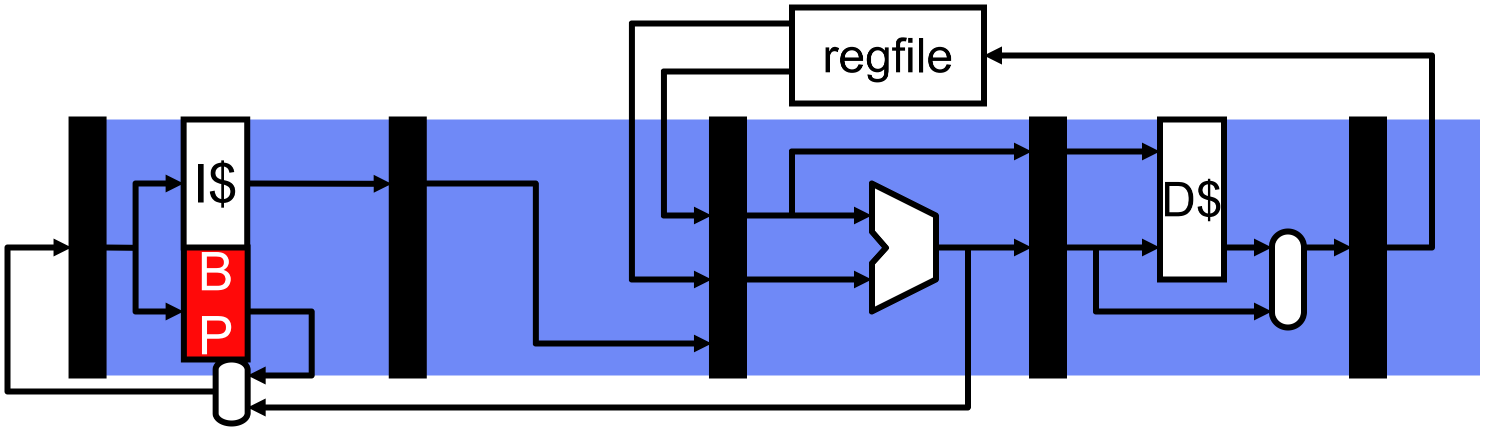
- Option #1: During Decode
  - Look at instruction opcode to determine branch instructions
  - Can calculate next PC from instruction (for PC-relative branches)
  - One cycle “mis-fetch” penalty **even if branch predictor is correct**

	1	2	3	4	5	6	7	8	9
`bnez r3,targ`	F	**D**	X	M	W				
`targ:add r4←r5,r4`			**F**	D	X	M	W		

- Option #2: During Fetch?
  - How do we do that?

Is It a Branch?

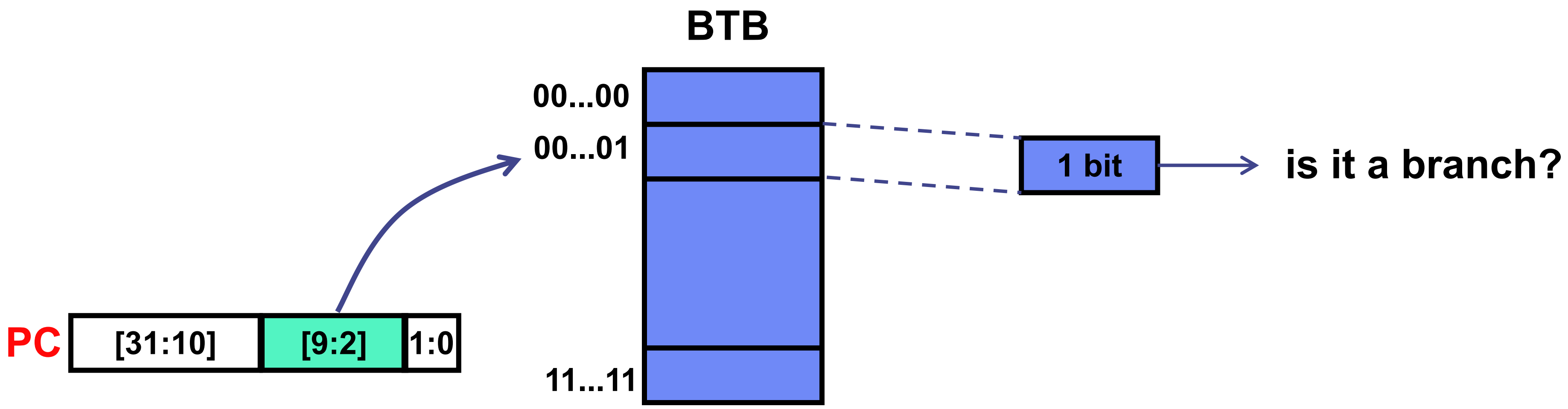
Revisiting Branch Prediction Components



- Step #1: is it a branch?
  - Easy after decode... during fetch: **predictor**
- Step #2: is the branch taken or not taken?
  - Direction predictor (later)
- Step #3: if the branch is taken, where does it go?
  - Branch target predictor (BTB)
  - Supplies target PC if branch is taken

Branch Target Buffer

- **Branch target buffer (BTB)**:
  - Record a list of branches we have seen
    + code doesn't change
  - PC indexes table of bits
    - each entry is 1 bit: is there a branch here?
  - What about aliasing?
    - Two PCs with the same lower bits?



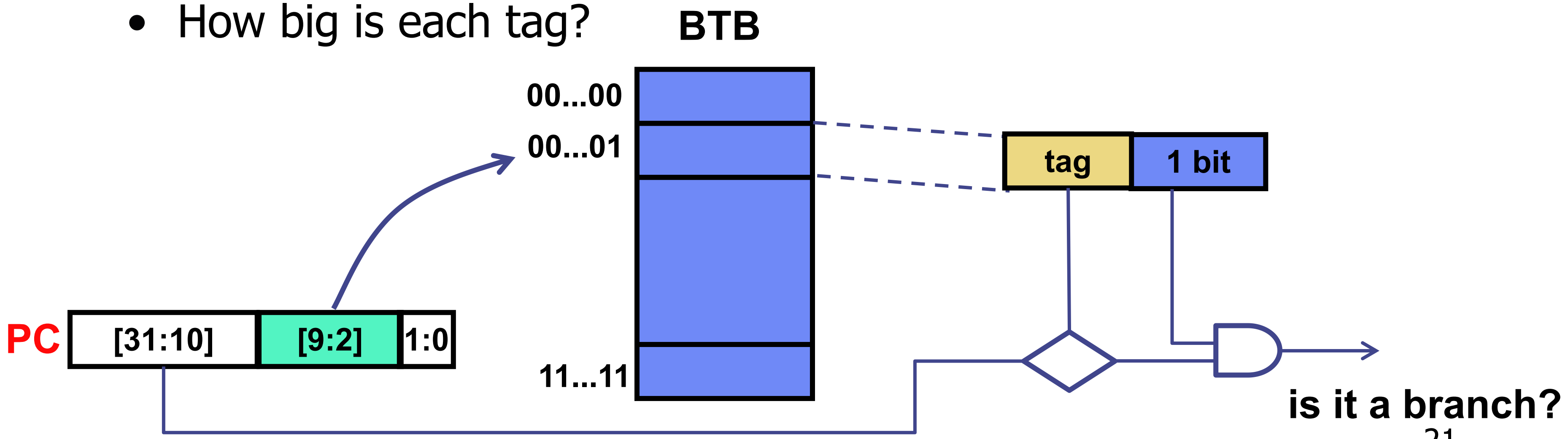
Branch Target Buffer

- BTB entries are too coarse-grained

+ Record only branches that were taken at least once
  - a never-taken branch might as well be a NOP

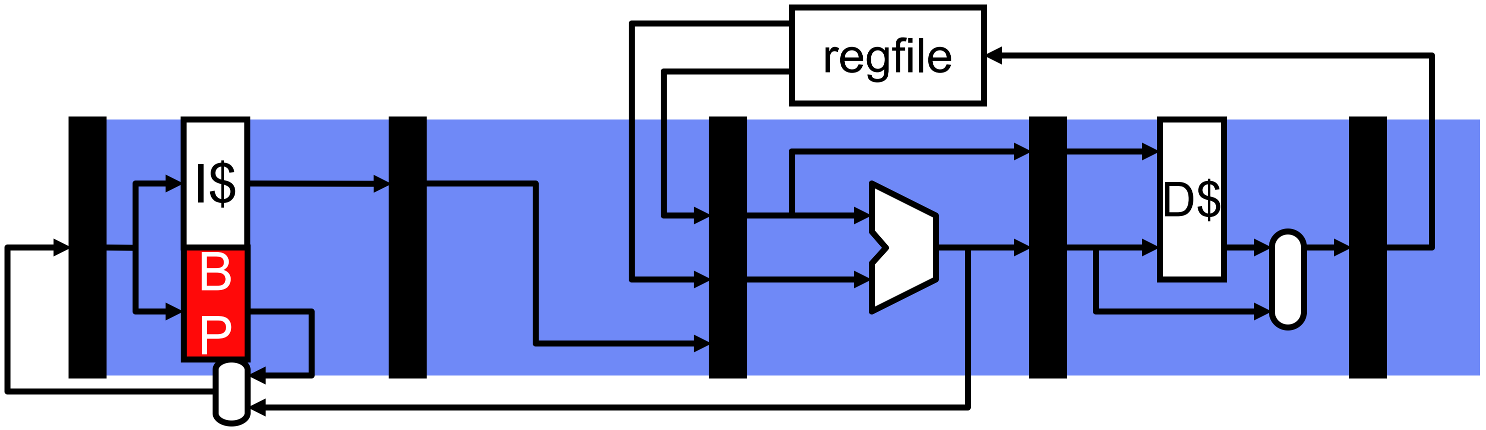
  – doesn't help enough

- better idea: **Tag each BTB entry**
  - remember **some** things precisely, rather than everything imprecisely
  - record a subset of actual taken branches
  - is_a_branch = (BTB[PC].branch && BTB[PC].tag == PC)
  - How big is each tag?



Branch Target Prediction

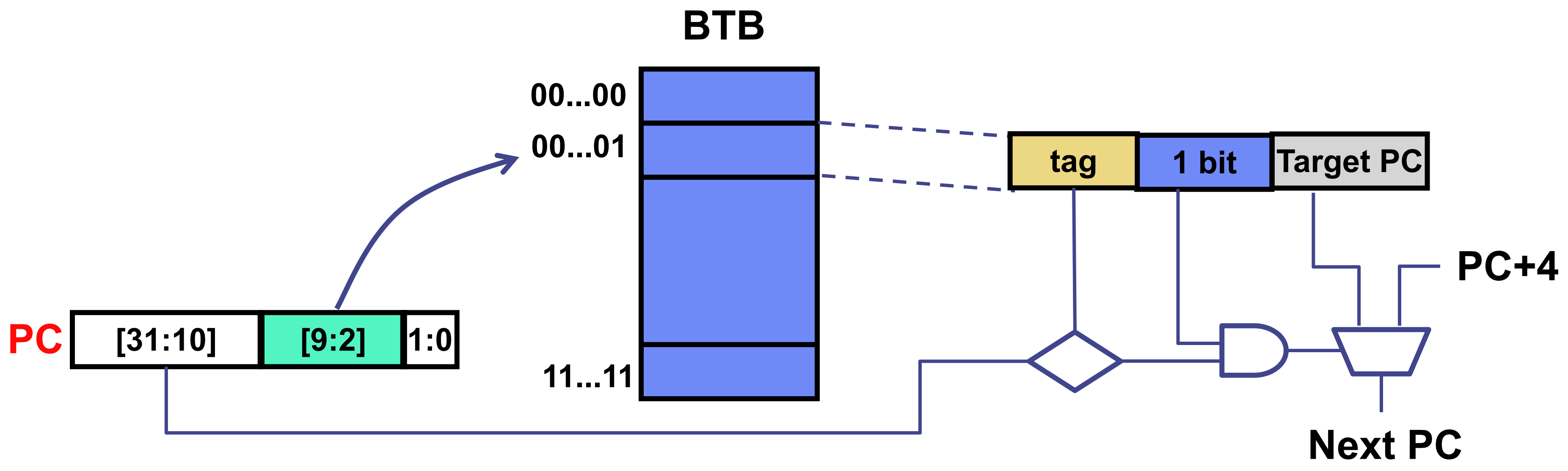
Revisiting Branch Prediction Components



- Step #1: is it a branch?
  - Easy after decode... during fetch: predictor
- Step #2: is the branch taken or not taken?
  - Direction predictor
- Step #3: if the branch is taken, where does it go?
  - **Branch target predictor (BTB)**
  - Supplies target PC if branch is taken

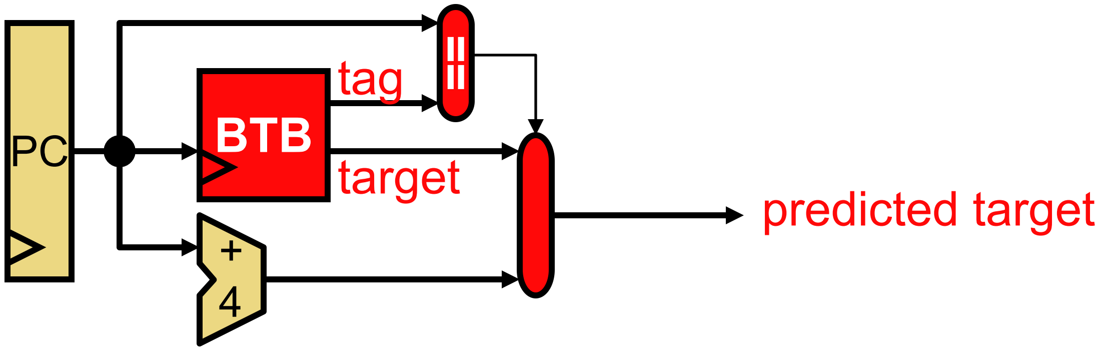
Branch Target Buffer, Again

- **Branch target buffer (BTB)**:
  - "guess" the future PC based on past behavior
  - "Last time the branch X was taken, it went to address Y"
    - "So, in the future, if address X is fetched, fetch address Y next"
  - Essentially: branch will go to same place it went last time
  - PC indexes table of **target addresses**
    - use tags to precisely remember a subset of branch targets
  - What about aliasing?
    - Two PCs with the same lower bits? (just a prediction)



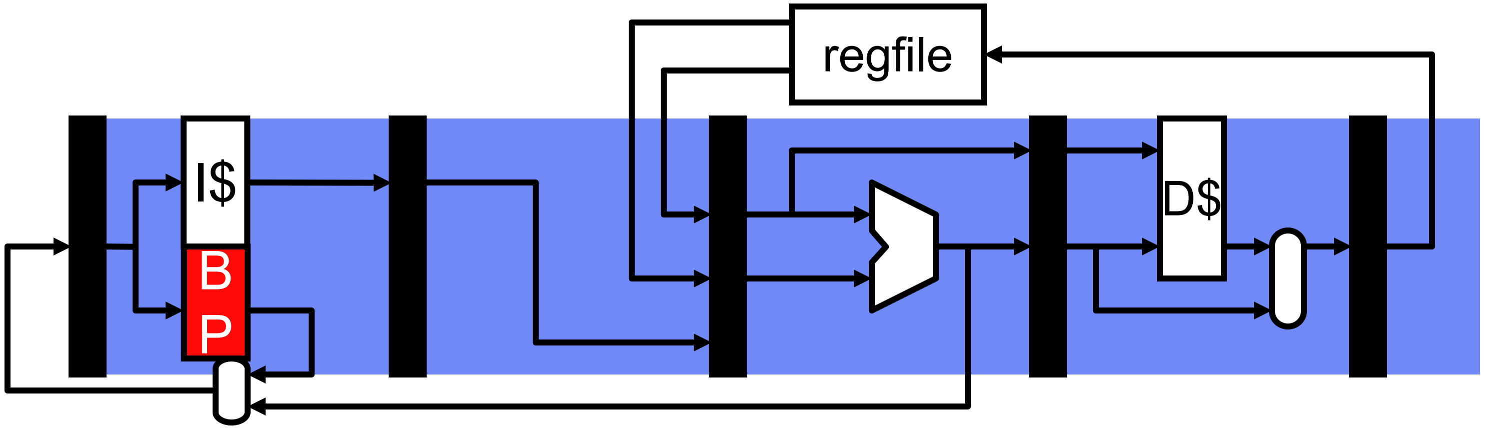
Branch Target Buffer (continued)

- At Fetch, how do we know we have a branch? We don't…
  - **…all insns access BTB in parallel with Imem Fetch**
- **BTB predicts which insn are branches, and targets**
  - **tag each entry with its corresponding PC**
  - Update BTB on every taken branch insn, record target PC:
    - BTB[PC].tag = PC, BTB[PC].target = target of branch
  - All insns access at Fetch *in parallel* with Imem
    - Check for tag match, indicates insn at that PC is a branch
      - otherwise, assume insn is **not** a branch
    - Predicted PC = (BTB[PC].tag == PC) ? BTB[PC].target : PC+4



Branch Direction Prediction

Revisiting Branch Prediction Components



- Step #1: is it a branch?
  - Easy after decode... during fetch: predictor
- Step #2: is the branch taken or not taken?
  - **Direction predictor**
- Step #3: if the branch is taken, where does it go?
  - Branch target predictor (BTB)
  - Supplies target PC if branch is taken

Simple Branch Direction Prediction

- Compile time (static)
  - Always not taken
  - Always taken
  - BTFN (Backward taken, forward not taken)
  - Profile based (likely direction)
  - Program analysis based (likely direction)

- Run time (dynamic)
  - Last time prediction (single-bit)
  - Two-bit counter based prediction
  - Two-level prediction (global vs. local)
  - Hybrid
  - Advanced algorithms (e.g., using perceptrons)

Static Branch Prediction (I)

- Always not-taken
  - Simple to implement: no need for BTB, no direction prediction
  - Low accuracy: ~30-40% (for conditional branches)
- Always taken
  - No direction prediction
  - Better accuracy: ~60-70% (for conditional branches)
    - 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
- Profile-based
  - Idea: Compiler determines likely direction for each branch using a profile run. Encodes that direction as a hint bit in the branch instruction format.

Static Branch Prediction (II)

- Program-based (or, program analysis based)
  - Idea: Use heuristics based on program analysis to determine statically-predicted direction
  - Example opcode heuristic: Predict BLEZ as NT (negative integers used as error values in many programs)
  - Example 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 (ditto for other static methods)

- 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?

Pragmas

- Idea: Keywords that enable a programmer to convey hints to lower levels of the transformation hierarchy

- if (likely(x)) { ... }
- if (unlikely(error)) { … }

- Many other hints and optimizations can be enabled with pragmas
  - E.g., whether a loop can be parallelized
  - **#pragma omp parallel**
  - **Description**
    - The omp parallel directive explicitly instructs the compiler to parallelize the chosen segment of code.

Static Branch Prediction

- All previous techniques can be combined
  - Profile based
  - Program based
  - Programmer based

- What is the common disadvantage of all three techniques?
  - Cannot adapt to dynamic changes in branch behavior
    - This can be mitigated by a dynamic compiler, but not at a fine granularity (and a dynamic compiler has its overheads…)
    - What is a Dynamic Compiler?
      - A compiler that generates code at runtime
      - Java JIT (just in time) compiler, Microsoft CLR (common lang. runtime)

More Sophisticated Direction Prediction

- Compile time (static)
  - Always not taken
  - Always taken
  - BTFN (Backward taken, forward not taken)
  - Profile based (likely direction)
  - Program analysis based (likely direction)

- Run time (dynamic)
  - Last time prediction (single-bit)
  - Two-bit counter based prediction
  - Two-level prediction (global vs. local)
  - Hybrid
  - Advanced algorithms (e.g., using perceptrons)

Dynamic Branch Prediction

- Idea: Predict branches based on dynamic information (collected at run-time)

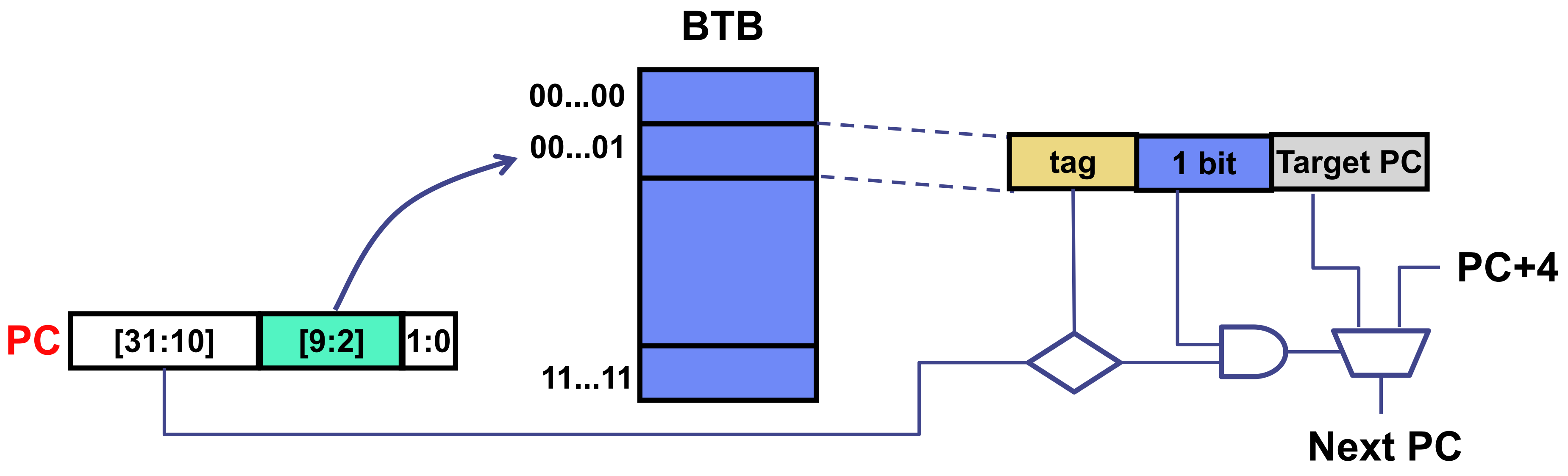
- Advantages
  - + Prediction based on history of the execution of branches
    - + It can adapt to dynamic changes in branch behavior
  - + No need for static profiling: input set representativeness problem goes away

- Disadvantages
  - -- More complex (requires additional hardware)

Last Time Predictor

- Last time predictor
  - Single bit per branch (stored in BTB)
  - 1 bit for prediction (0 = N, 1 = T)
  - Indicates which direction branch went last time it executed

  TTTTTTTTTTNNNNNNNNNN → 90% accuracy



Last Time Predictor

- Problem: **inner loop branch** below

```
for (i=0;i<100;i++)
    for (j=0;j<3;j++)
        // whatever
```

  - Two "built-in" mis-predictions per inner loop iteration
  - Branch predictor "changes its mind too quickly"

Time	State	Prediction	Outcome	Result?
1	N	N	T	Wrong
2	T	T	T	Correct
3	T	T	T	Correct
4	T	T	N	Wrong
5	N	N	T	Wrong
6	T	T	T	Correct
7	T	T	T	Correct
8	T	T	N	Wrong
9	N	N	T	Wrong
10	T	T	T	Correct
11	T	T	T	Correct
12	T	T	N	Wrong

Improving the Last Time Predictor

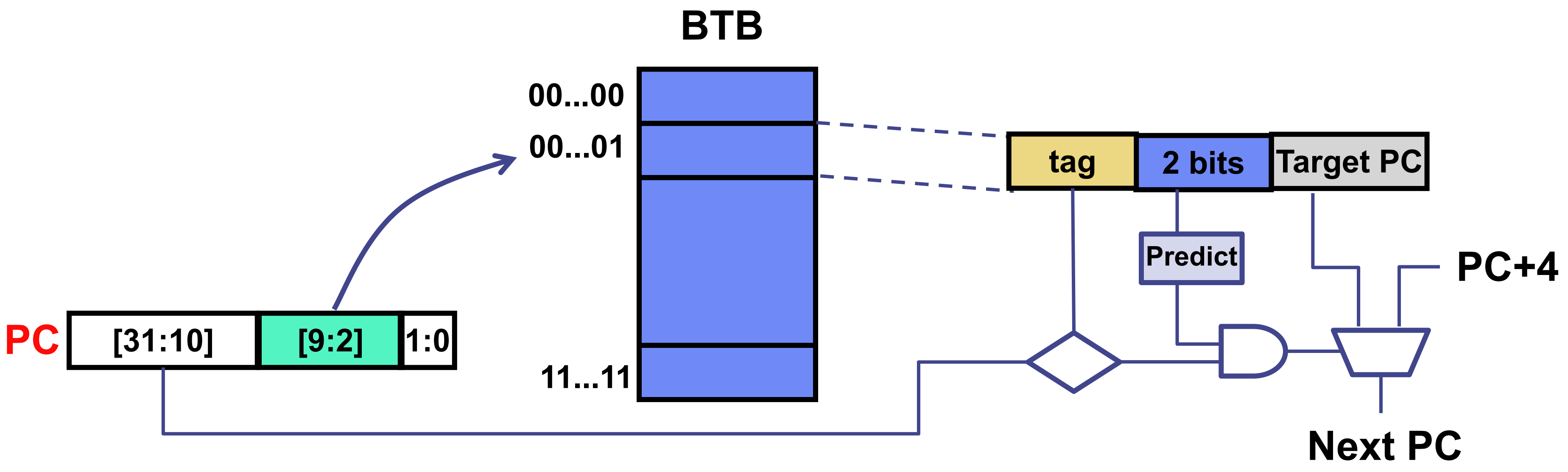
- Problem: A last-time predictor changes its prediction from T→NT or NT→T too quickly
  - even though the branch may be mostly taken or mostly not taken

- Solution Idea: Add hysteresis to the predictor so that prediction does not change on a single different outcome
  - Use two bits to track the history of predictions for a branch instead of a single bit
  - Can have 2 states for T or NT instead of 1 state for each

- Smith, "A Study of Branch Prediction Strategies," ISCA 1981.

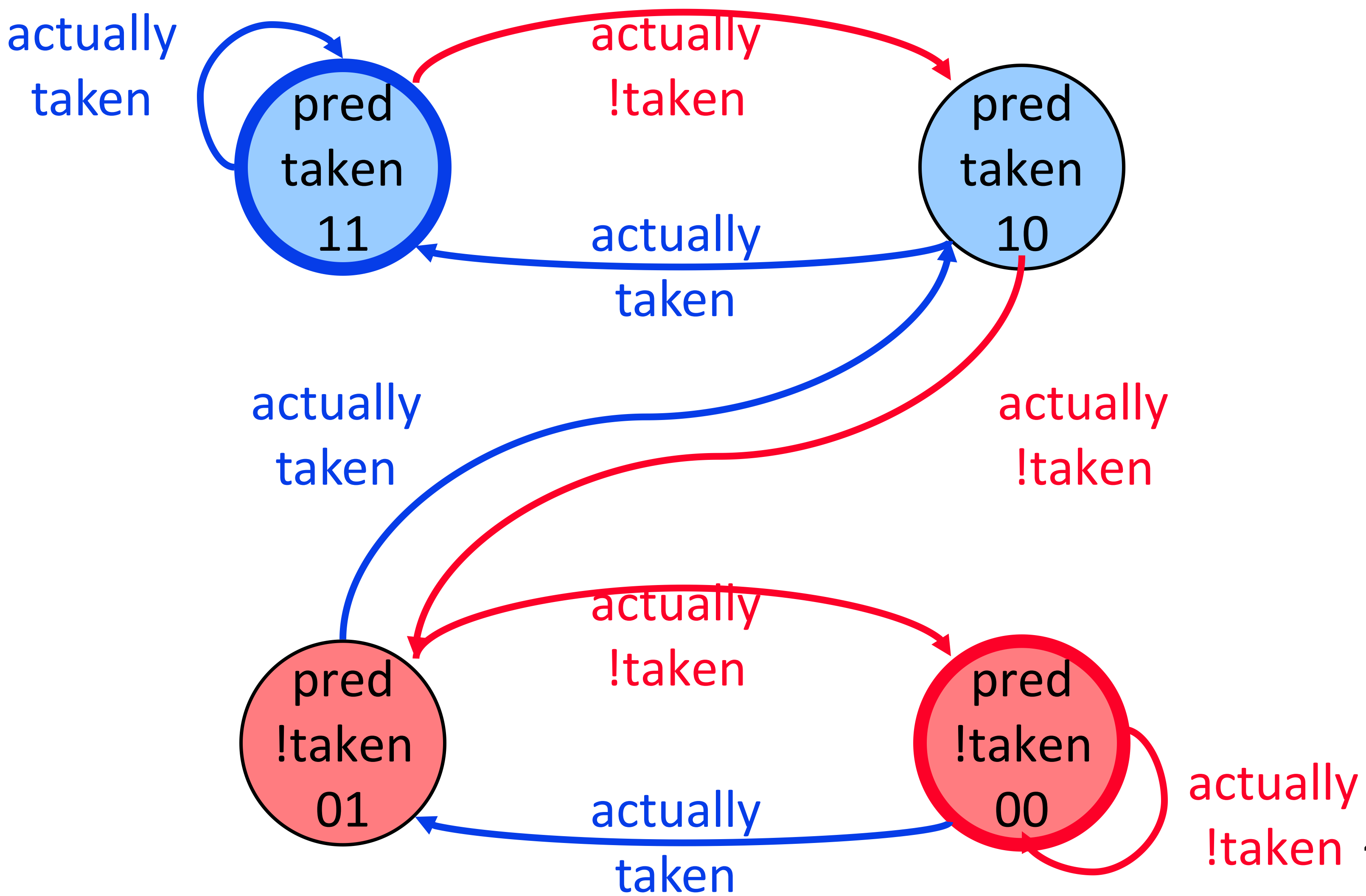
Two-Bit Counter Based Prediction

- Each branch associated with a two-bit counter
- One more bit provides hysteresis
- 2 bits: (00->N, 01->n, 10->t, 11->T)



State Machine for 2-bit Saturating Counter

- Counter using *saturating arithmetic*
  - Arithmetic with maximum and minimum values



Two-Bit Counter Based Prediction

- A strong prediction does not change with one single different outcome
- Accuracy for a loop with N iterations = (N-1)/N

TNTNTNTNTNTNTNTNTNTN → 50% accuracy

(assuming counter initialized to weakly taken)

+ Better prediction accuracy

-- More hardware cost (but counter can be part of a BTB entry)

Time	State	Prediction	Outcome	Result?
1	N	N	T	Wrong
2	n	N	T	Wrong
3	t	T	T	Correct
4	T	T	N	Wrong
5	t	T	T	Correct
6	T	T	T	Correct
7	T	T	T	Correct
8	T	T	N	Wrong
9	t	T	T	Correct
10	T	T	T	Correct
11	T	T	T	Correct
12	T	T	N	Wrong

Is This Good Enough?

- ~85-90% accuracy for **many** programs with 2-bit counter based prediction (also called bimodal prediction)

- Is this good enough?

- How big is the branch problem?

Importance of The Branch Problem

- Assume N = 20 (20 pipe stages), W = 5 (5 wide fetch)
- Assume: 1 out of 5 instructions is a branch
- Assume: Each 5 instruction-block ends with a branch

- How long does it take to fetch 500 instructions?
  - 100% accuracy
    - 100 cycles (all instructions fetched on the correct path)
    - No wasted work; IPC = 500/100
  - 90% accuracy
    - 100 (correct path) + 20 * 10 (wrong path) = 300 cycles
    - 200% extra instructions fetched; IPC = 500/300
  - 85% accuracy
    - 100 (correct path) + 20 * 15 (wrong path) = 400 cycles
    - 300% extra instructions fetched; IPC = 500/400
  - 80% accuracy
    - 100 (correct path) + 20 * 20 (wrong path) = 500 cycles
    - 400% extra instructions fetched; IPC = 500/500

Can We Do Better: Two-Level Prediction

- Last-time and 2BC predictors exploit "last-time" predictability

- Realization 1: A branch's outcome can be correlated with other branches' outcomes
  - Global branch correlation

- Realization 2: A branch's outcome can be correlated with past outcomes of the same branch (other than the outcome of the branch "last-time" it was executed)
  - Local branch correlation

Yeh and Patt, "Two-Level Adaptive Training Branch Prediction," MICRO 1991.

Global Branch Correlation (I)

- Recently executed branch outcomes in the execution path are correlated with the outcome of the next branch

- If first branch not taken, second also not taken

```
if (cond1)
...
if(cond1 AND cond2)
```

- If first branch taken, second definitely not taken

```
branch Y: if (cond1) a = 2;
...
branch X: if(a == 0)
```

Global Branch Correlation (II)

- If Y and Z both taken, then X also taken
- If Y or Z not taken, then X also not taken

```
branch Y: if (cond1)
...
branch Z: if (cond2)
...
branch X: if(cond1 AND
cond2)
```

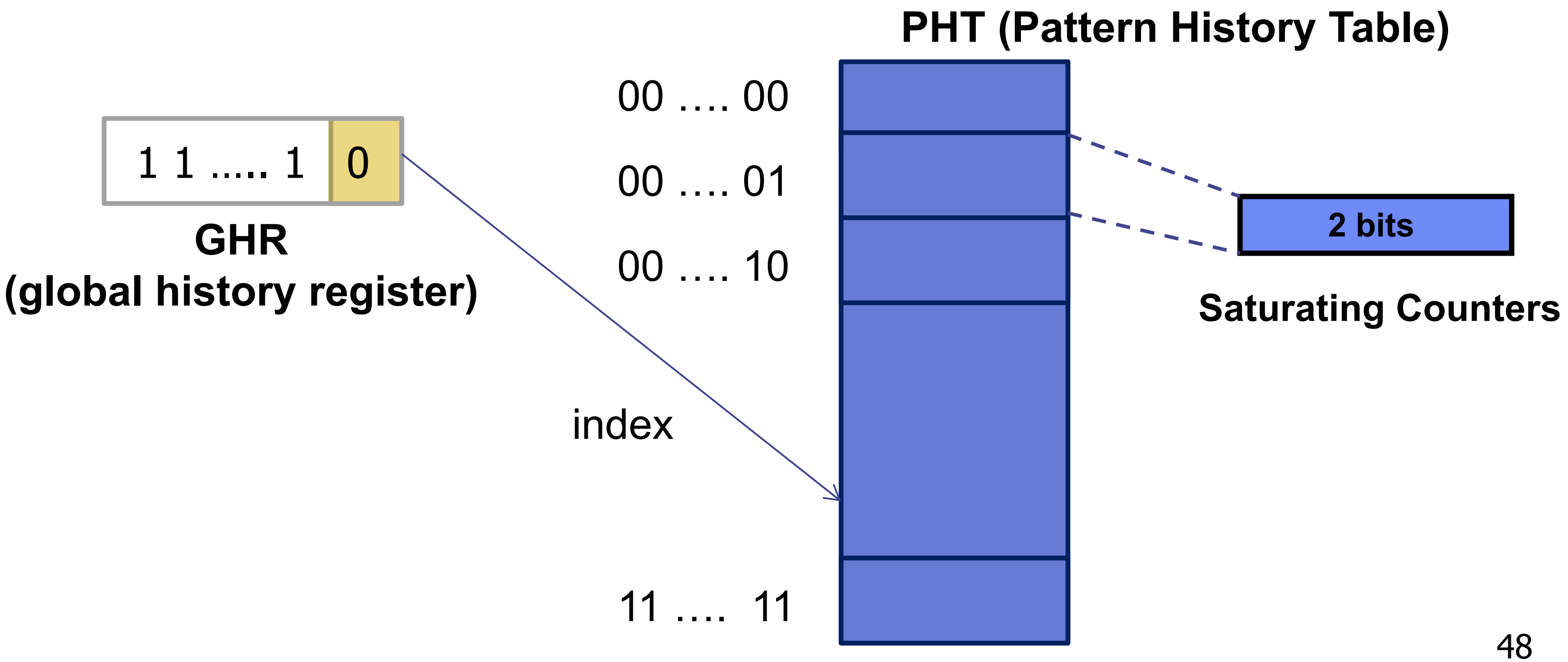
Capturing Global Branch Correlation

- Idea: Associate branch outcomes with "global T/NT history" of all branches
- Make a prediction based on the outcome of the branch the last time the same global branch history was encountered

- Implementation:
  - Keep track of the "global T/NT history" of all branches in a register → Global History Register (GHR)
  - Use GHR to index into a table that recorded the outcome that was seen for each GHR value in the recent past → Pattern History Table (table of 2-bit counters)

Two Level Global Branch Prediction

- First level: Global branch history register (N bits)
  - The direction of last N branches (1 bit per branch)
  - 1->tacken, 0->not taken
- Second level: Table of saturating counters for each history entry
  - The direction the branch took the last time the same history was seen



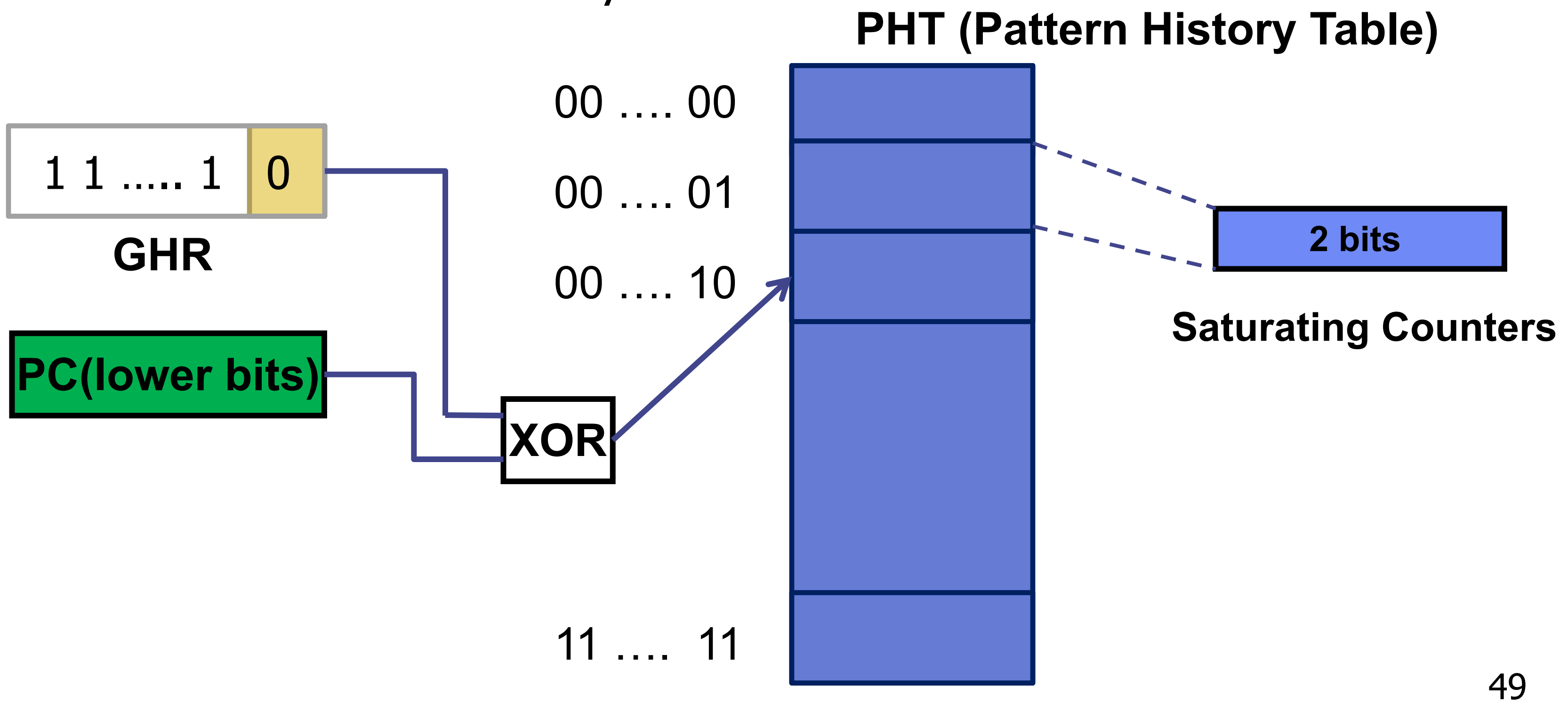
Improving Global Predictor Accuracy (I)

- Idea: Add more context information to the global predictor to take into account which branch is being predicted
  - Gshare predictor: GHR hashed with the Branch PC

  + More context information used for prediction

  + Better utilization of the two-bit counter array

  -- Increases access latency

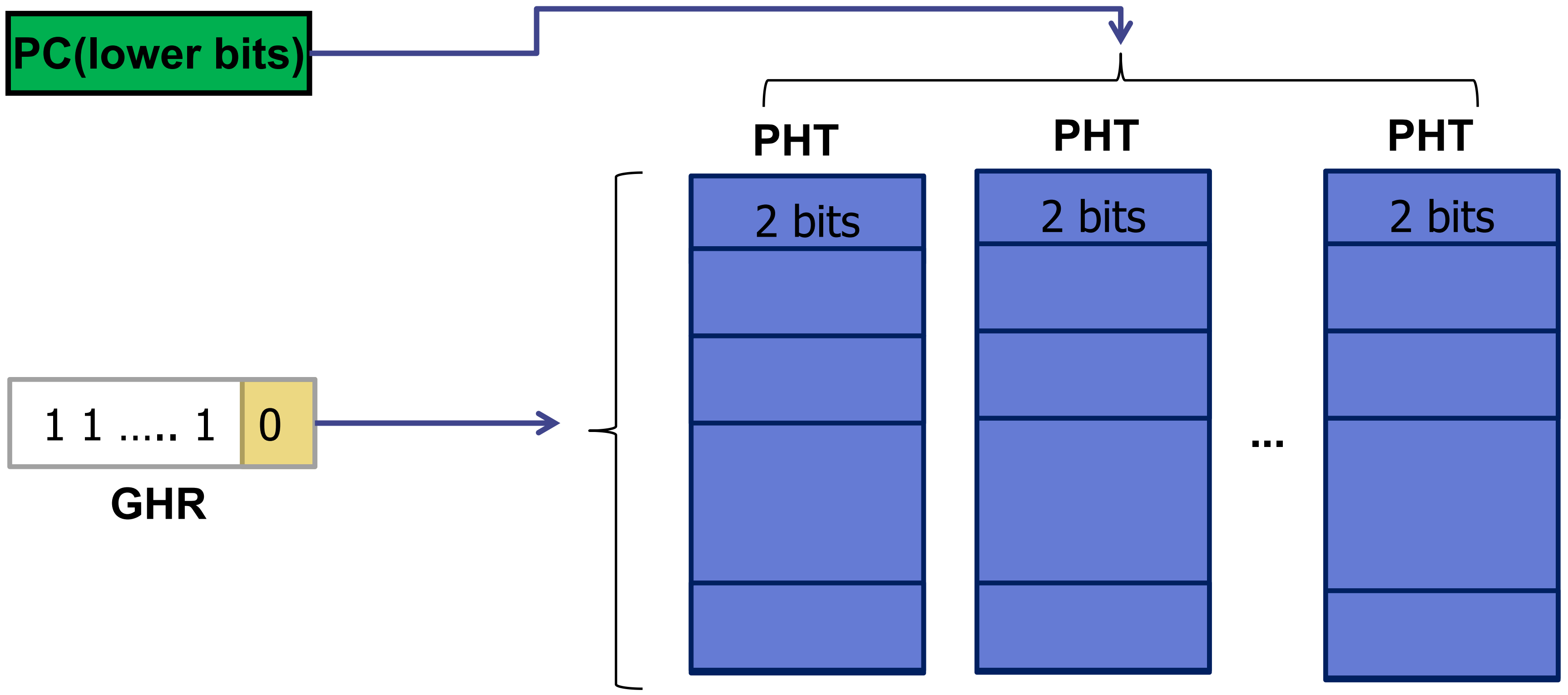


Improving Global Predictor Accuracy (II)

- Idea: Main separate PHT for different branches
  - use PC to determine the PHT
  - use GHR to determine the entry of the PHT

  + More context information used for prediction

  -- Increases hardware complexity



Intel Pentium Pro Branch Predictor

- Two level global branch predictor
- 4-bit global history register
- Multiple pattern history tables (of 2 bit counters)
  - Which pattern history table to use is determined by lower order bits of the branch address

Can We Do Better: Two-Level Prediction

- Last-time and 2BC predictors exploit "last-time" predictability

- Realization 1: A branch's outcome can be correlated with other branches' outcomes
  - Global branch correlation

- Realization 2: A branch's outcome can be correlated with past outcomes of the same branch (other than the outcome of the branch "last-time" it was executed)
  - Local branch correlation

Yeh and Patt, "Two-Level Adaptive Training Branch Prediction," MICRO 1991.

Local Branch Correlation

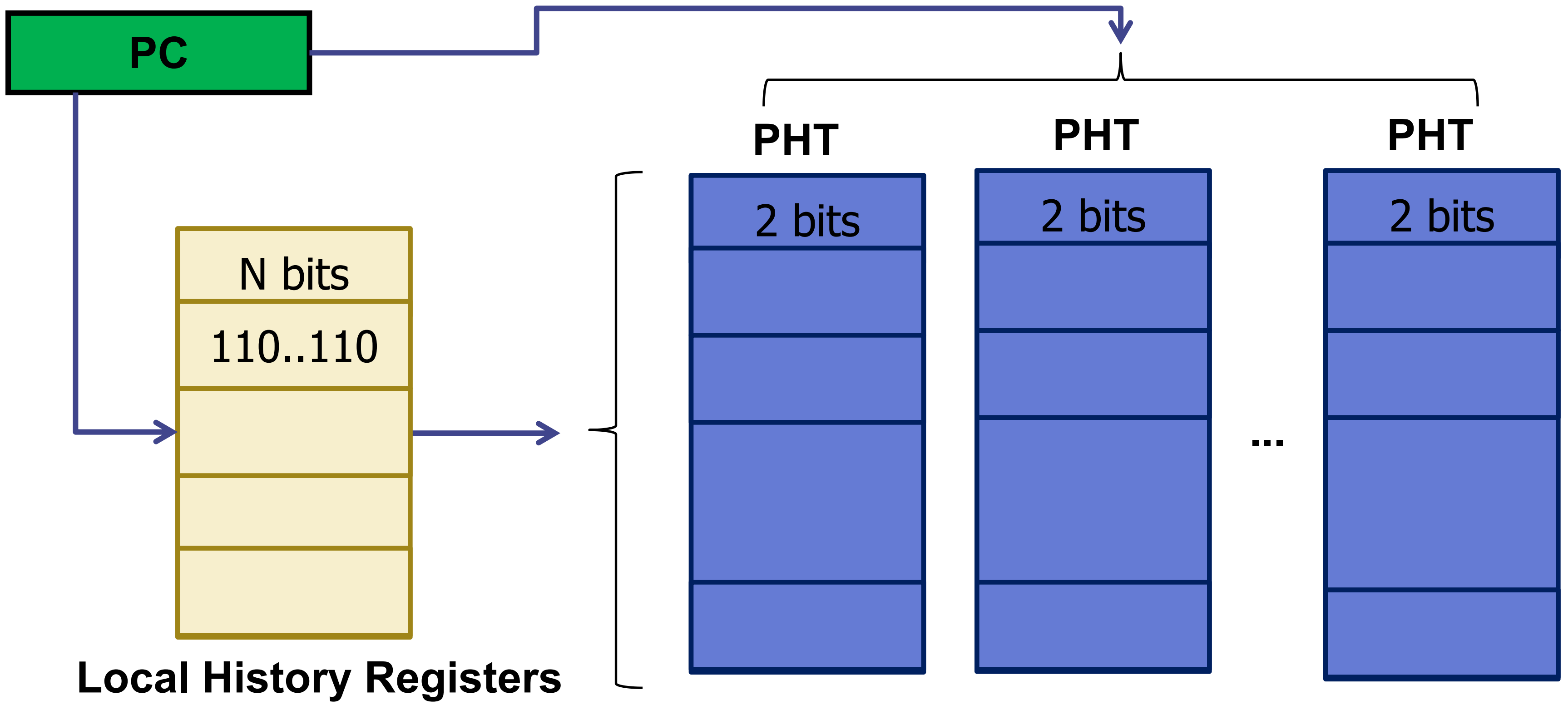
```
for(int i=1; i<4; i++)
```

Pattern: **TTTNTTTNTTTNTTTNTTTNTTTN**

Clearly, if we know the direction this branch had gone on the previous three executions, then we could always be able to predict the next branch direction

Two Level Local Branch Prediction

- First level: A set of local history registers (N bits each)
  - Select the history register based on the PC of the branch
- Second level: Table of saturating counters for each history entry
  - PC determines the PHT of this branch
  - the local history register determines the PHT entry



Can We Do Even Better?

- Predictability of branches varies

- Some branches are more predictable using local history
- Some using global
- For others, a simple two-bit counter is enough
- Yet for others, a single bit is enough

- Observation: There is heterogeneity in predictability behavior of branches
    - No one-size fits all branch prediction algorithm for all branches

- Idea: Exploit that heterogeneity by designing heterogeneous branch predictors

Hybrid Branch Predictors

- Idea: Use more than one type of predictor (i.e., multiple algorithms) and select the "best" prediction
  - E.g., hybrid of 2-bit counters and global predictor

- Advantages:

  + Better accuracy: different predictors are better for different branches

  + Reduced warmup time (faster-warmup predictor used until the slower-warmup predictor warms up)

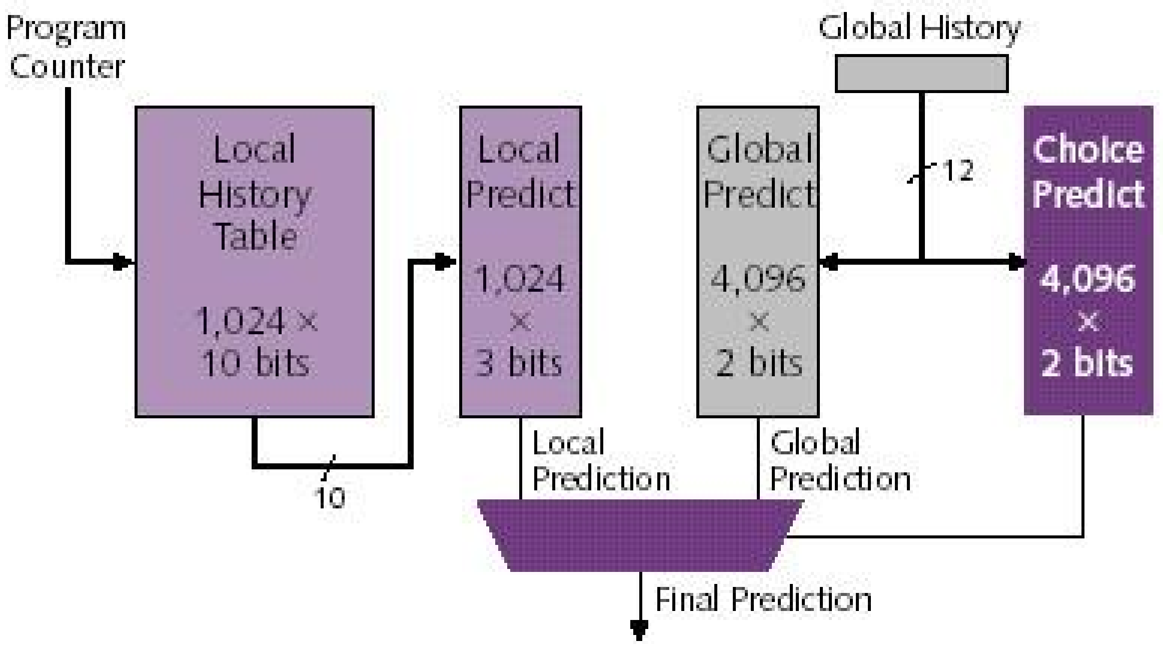
- Disadvantages:

  -- Need "meta-predictor" or "selector"

  -- Longer access latency

  - McFarling, "Combining Branch Predictors," DEC WRL Tech Report, 1993.

Alpha 21264 Tournament Predictor

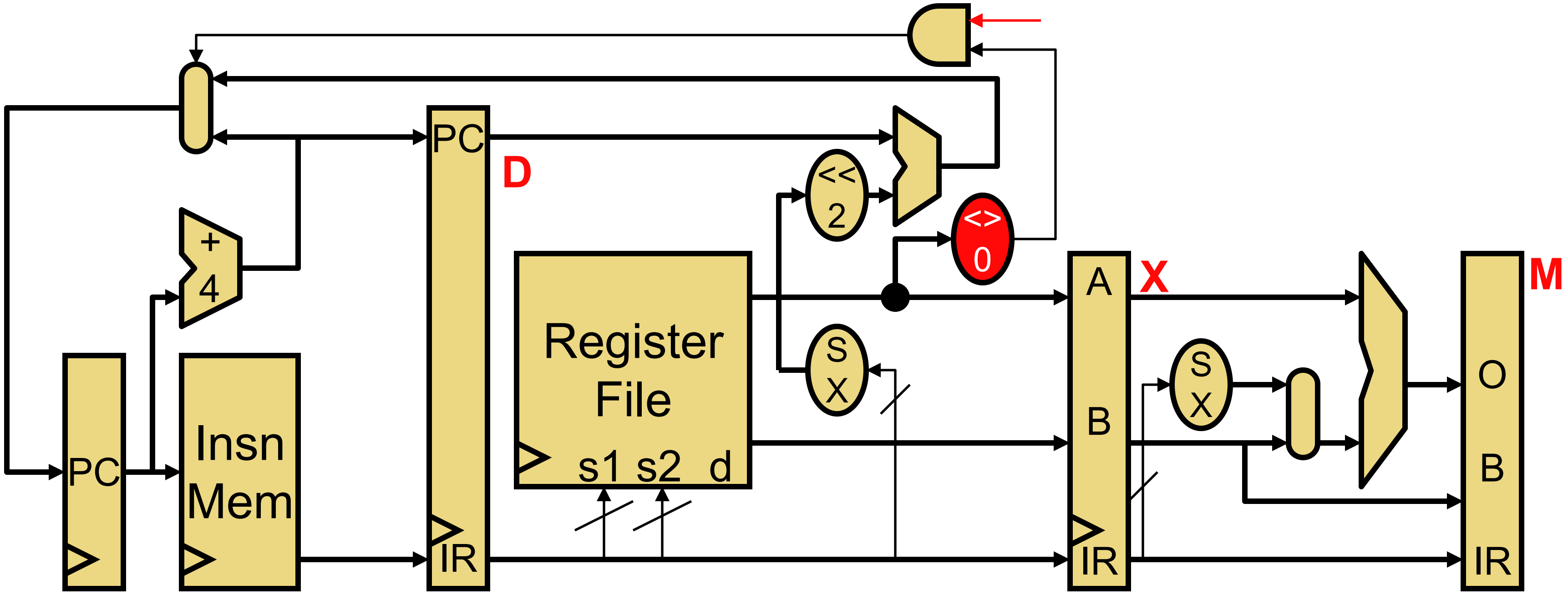


- Minimum branch penalty: 7 cycles
- Typical branch penalty: 11+ cycles
- 48K bits of target addresses stored in I-cache
- Predictor tables are reset on a context switch

- Kessler, "The Alpha 21264 Microprocessor," IEEE Micro 1999.

Reducing Branch Penalty

Reducing Penalty: Fast Branches



- **Fast branch**: can decide at D, not X
  - Test must be comparison to zero or equality, no time for ALU
  - + New taken branch penalty is 1
  - – Additional insns (`slt`) for more complex tests, must bypass to D too

Reducing Penalty: Fast Branches

- **Fast branch**: targets control-hazard penalty
  - Basically, branch insns that can resolve at D, not X
    - Test must be comparison to zero or equality, **no time for ALU**

  + New taken branch penalty is 1

  – Additional comparison insns (e.g., `cmplt`, `slt`) for complex tests

  – Must bypass into decode stage now, too

	1	2	3	4	5	6	7	8	9
`bnez r3,targ`	F	D	X	M	W				
~~`st r6→[r7+4]`~~		F	D	--	--	--			
`targ:add r4←r5,r4`			F	D	X	M	W		

Fast Branch Performance

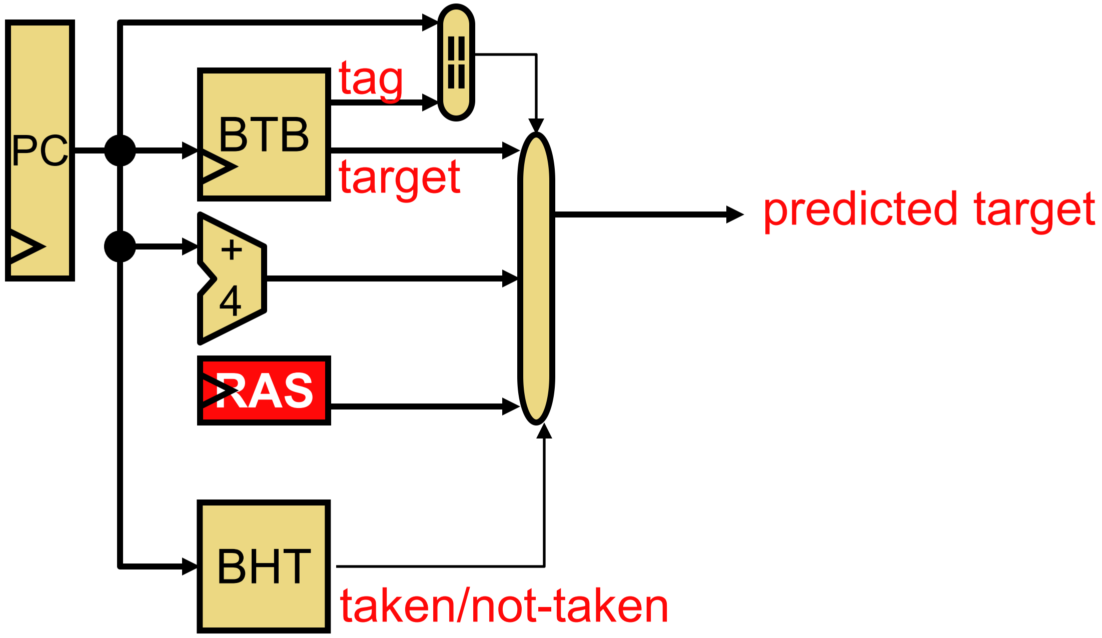
- Assume: Branch: 20%, 75% of branches are taken
  - CPI = 1 + 20% * 75% * **1** = 1 + **0.20*0.75*1** = **1.15**
    - **15% slowdown**

- But wait, fast branches assume only simple comparisons
  - Fine for MIPS
  - But not fine for ISAs with "branch if $1 > $2" operations

- In such cases, say 25% of branches require an extra insn
  - CPI = 1 + (20% * 75% * 1) + 20%*25%*1(extra insn) = **1.2**

Putting It All Together

- BTB & branch direction predictor during fetch



- If branch prediction correct, no taken branch penalty

Branch Prediction Performance

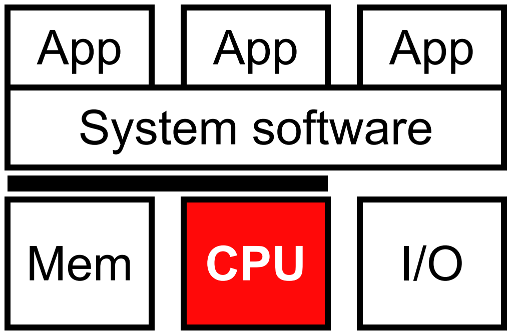
- Dynamic branch prediction
  - 20% of instruction branches
  - Simple predictor: branches predicted with 75% accuracy
    - CPI = 1 + (20% * **25%** * 2) = **1.1**
  - More advanced predictor: 95% accuracy
    - CPI = 1 + (20% * **5%** * 2) = **1.02**

- Branch mis-predictions still a big problem though
  - Pipelines are long: typical mis-prediction penalty is 10+ cycles
  - For cores that do more per cycle, predictions more costly (later)

Branch Prediction in the Future

- Further improvement is difficult
  - 95%~98%
  - Branch predictor is getting larger, Alpha21464 has 48KB
- Improvement is significant
  - 5% precision improvement can reduce 50 pipeline flush per 10000 insts
- New applications and new architecture need new branch prediction approach
  - power aware
  - SMT
  - neural network
- Branch Prediction Competetion
  - supported by industry

Summary



- Control hazards
- Branch prediction