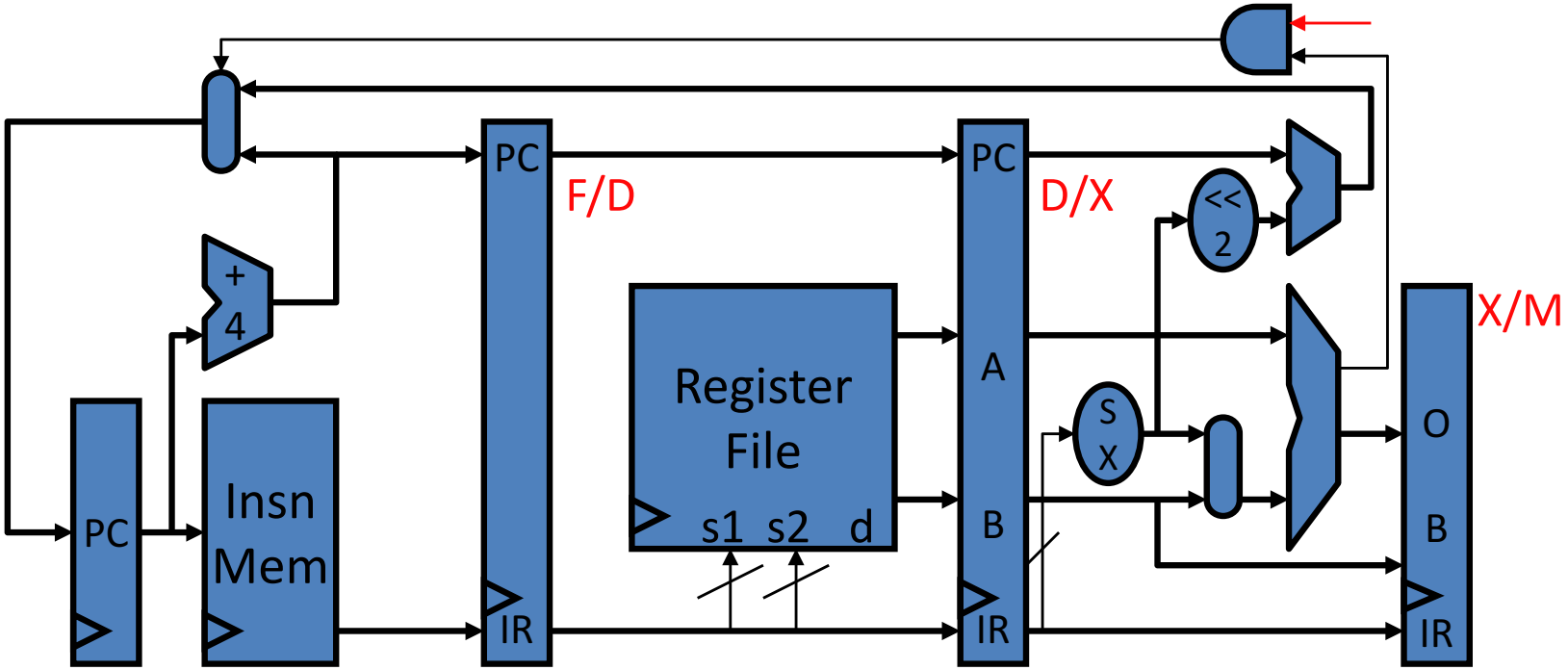
CSE 548: Computer Systems Architecture

Branch Prediction, Predication

Luis Ceze, Spring 2017

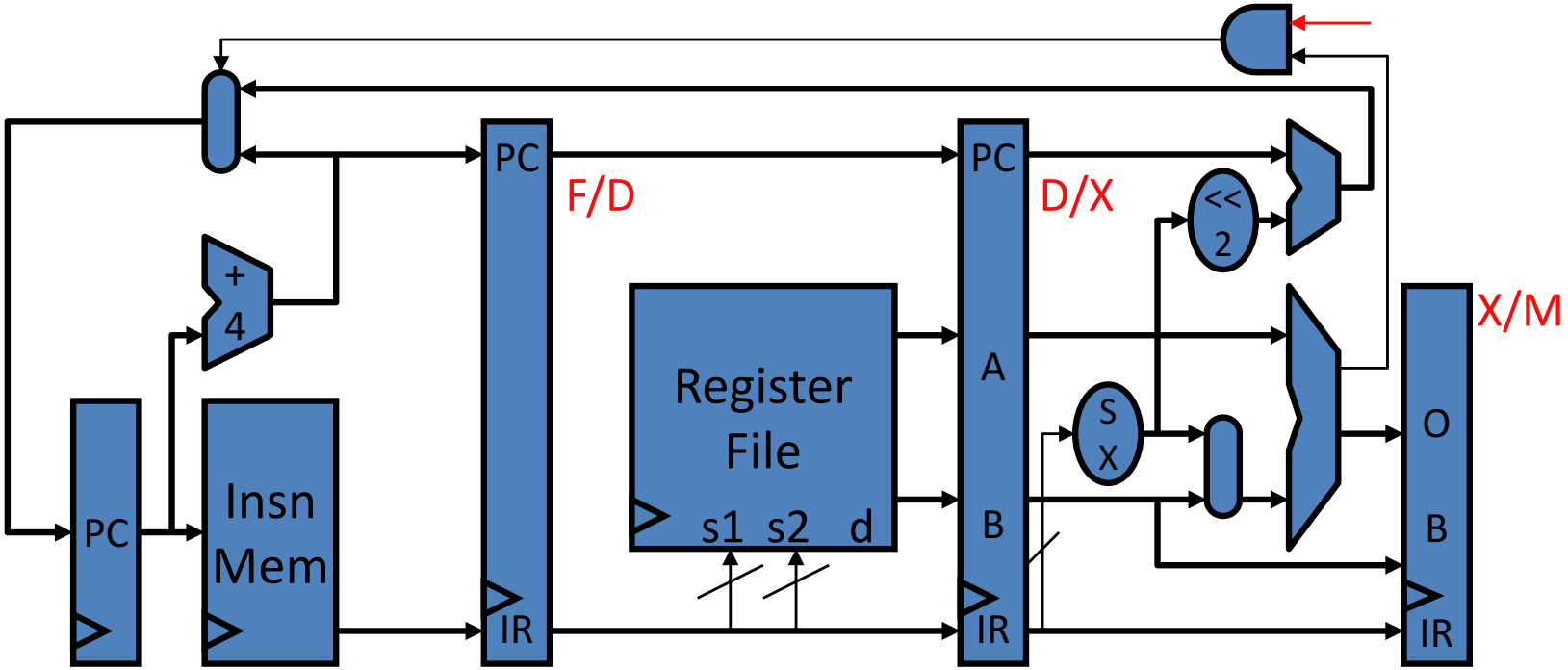
(based on slides lifted from friends at UPenn, UIUC, UW, MIT.

What About Branches?



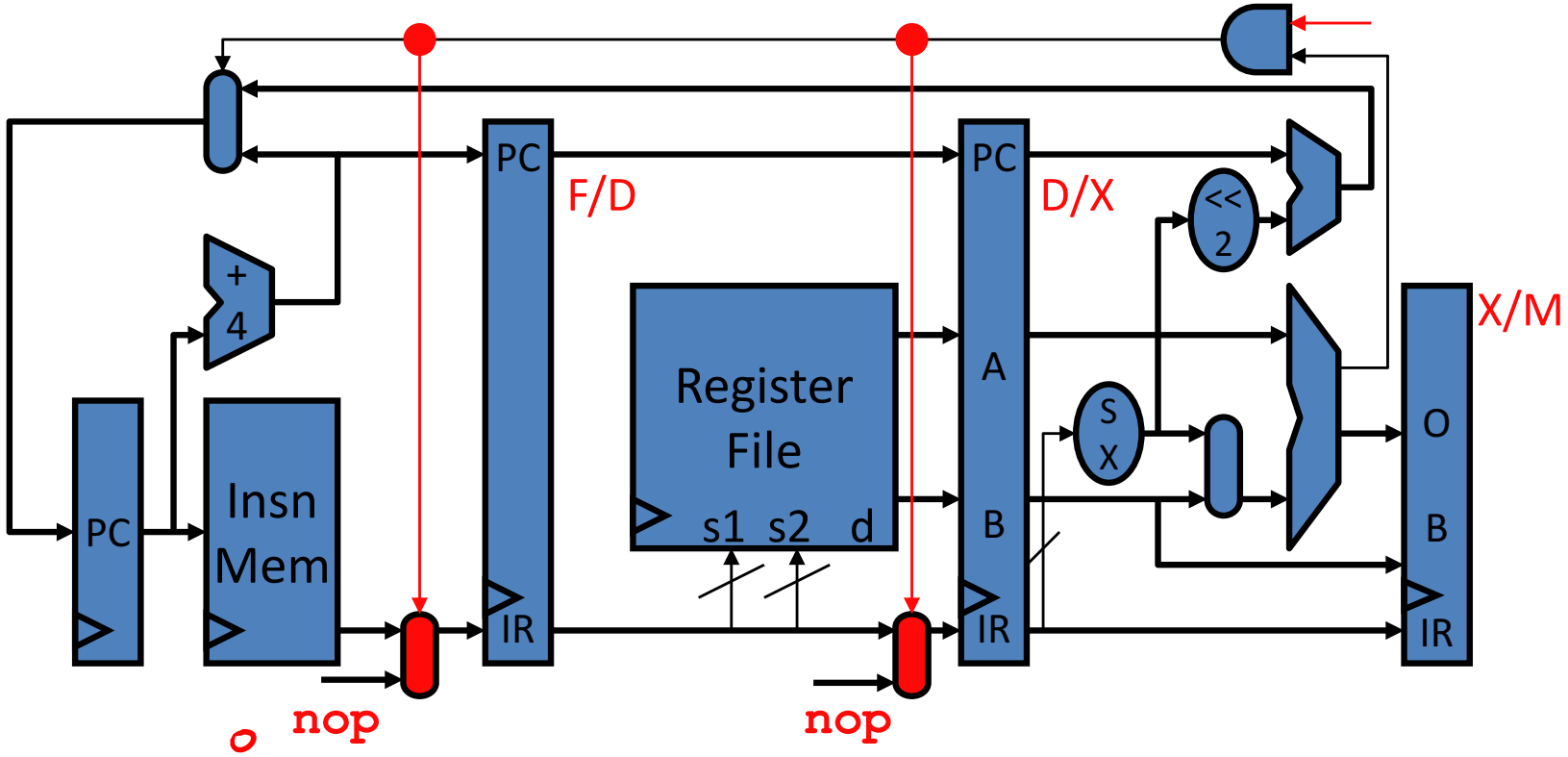
How do you think branches work?
When do you know if the branch is taken?
When do you continue fetching?

What About Branches?



- **Control hazards options**
  - Could just stall to wait for branch outcome (two-cycle penalty)
  - **Fetch past branch insns before branch outcome is known (prediction!)**
    - Default: assume "**not-taken**" (at fetch, can't tell it's a branch)
    - What if it was a **taken** branch?

Branch Recovery



- **Branch recovery**: what to do when branch is actually taken
  - Insns that will be written into F/D and D/X are wrong
  - **Flush them**, i.e., replace them with **nops**
  + They haven't had written permanent state yet (regfile, DMem)
  - Two cycle penalty for taken branches

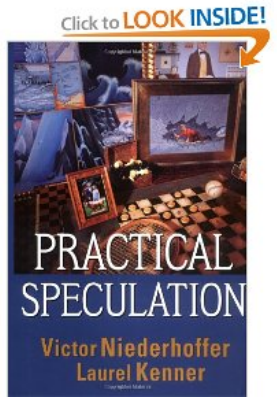
Branch Performance

- Back of the envelope calculation
  - **Branch: 20%**, load: 20%, store: 10%, other: 50%
  - Say, **75% of branches are taken**

- CPI = 1 + 20% * 75% * 2 =

  1 + **0.20 * 0.75 * 2** = 1.3
  - **Branches cause 30% slowdown**
    - Even worse with deeper pipelines
  - How do we reduce this penalty?

Big Idea: Speculative Execution

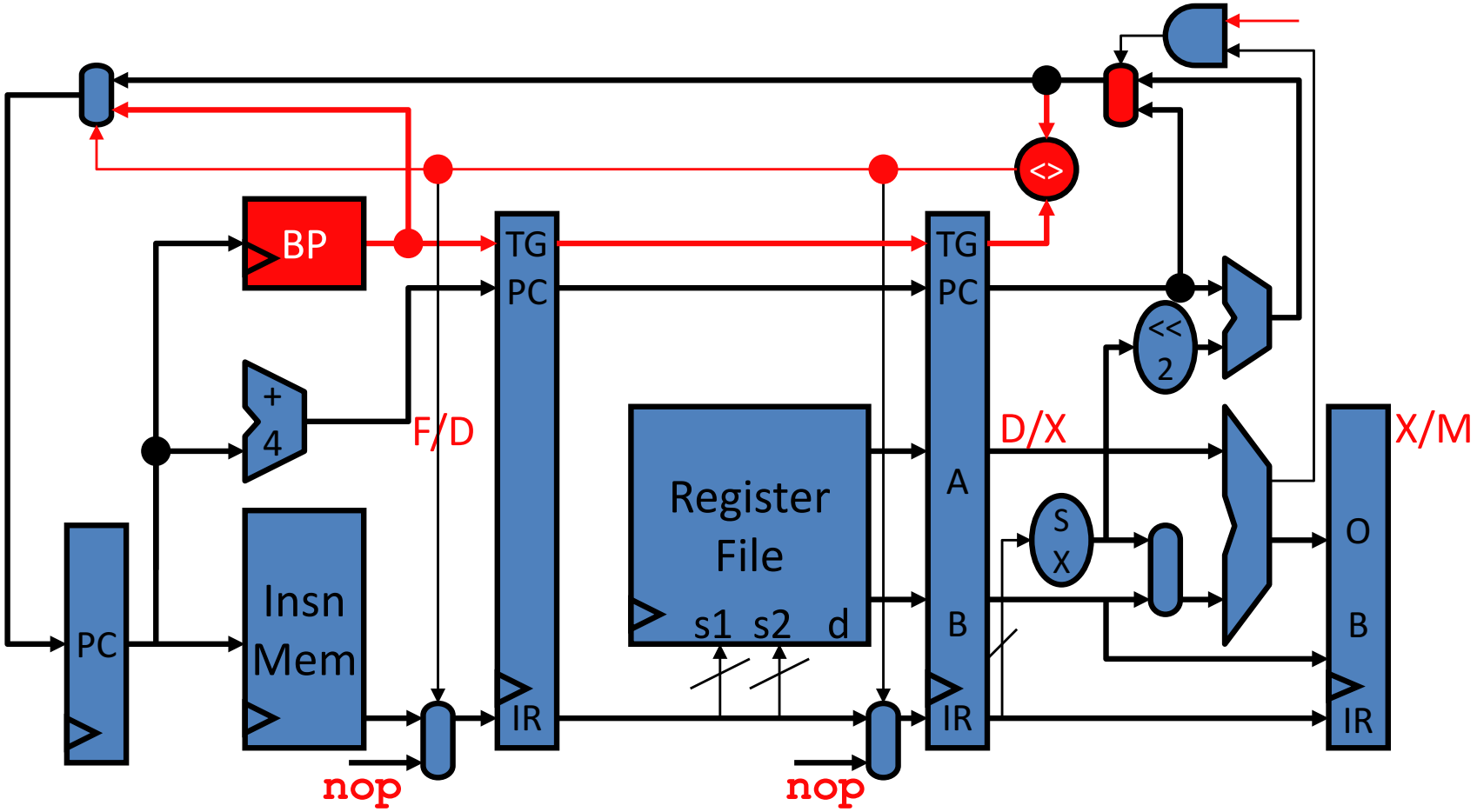


- Speculation: "risky transactions on chance of profit"

- **Speculative execution**
  - Execute before all parameters known with certainty
  - **Correct speculation**
    - + Avoid stall, improve performance
  - **Incorrect speculation (mis-speculation)**
    - Must abort/flush/squash incorrect insns
    - Must undo incorrect changes (recover pre-speculation state)
  - The "game": **[$\%_{correct}$ * gain] – [(1–$\%_{correct}$) * penalty]**

- **Control speculation**: speculation aimed at control hazards
  - Unknown parameter: are these the correct insns to execute next?
- We will see lots of other forms of speculation in computer systems design!

Branch Prediction



- **Dynamic branch prediction**:
  - Hardware guesses outcome
  - Start fetching from guessed address

Branch Prediction Performance

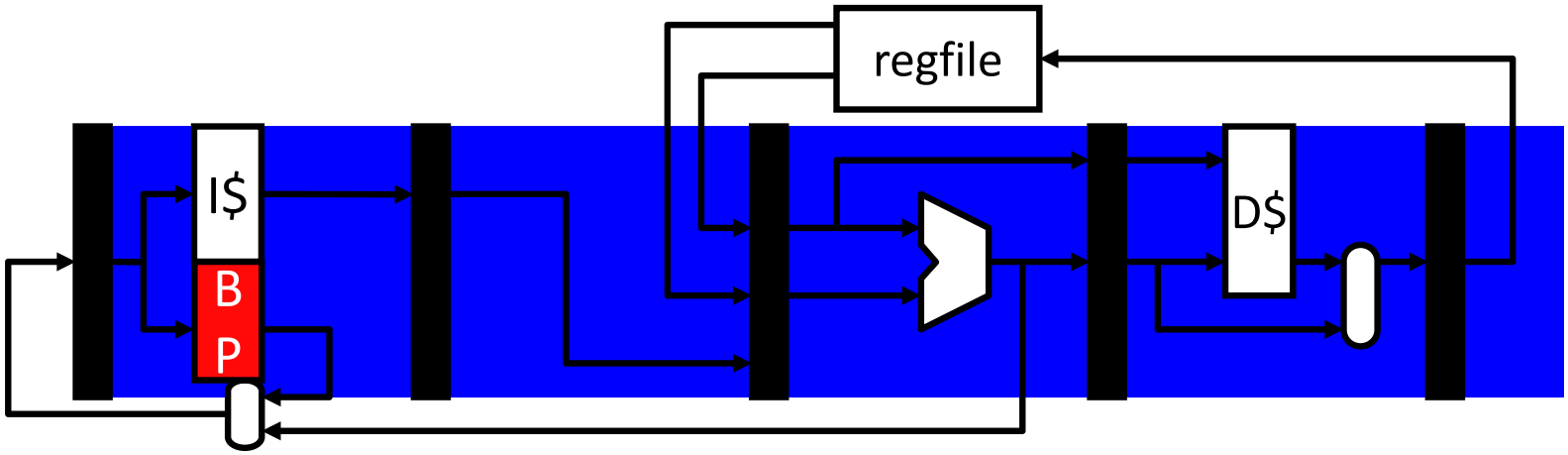
- Parameters
  - **Branch: 20%**, load: 20%, store: 10%, other: 50%
  - 75% of branches are taken
- Dynamic branch prediction
  - Branches predicted with 95% accuracy
  - CPI = 1 + 20% * 5% * 2 = **1.02**

Why are branches predictable?

Why are branches predictable?

```
for (i=0; i<1000000; i++) {    // Highly biased
   if (i % 3 == 0) {           // and this one?
      // whatever
   }
   if (random() % 2 == 0) {    // how about this one?
      …
  }
}
```

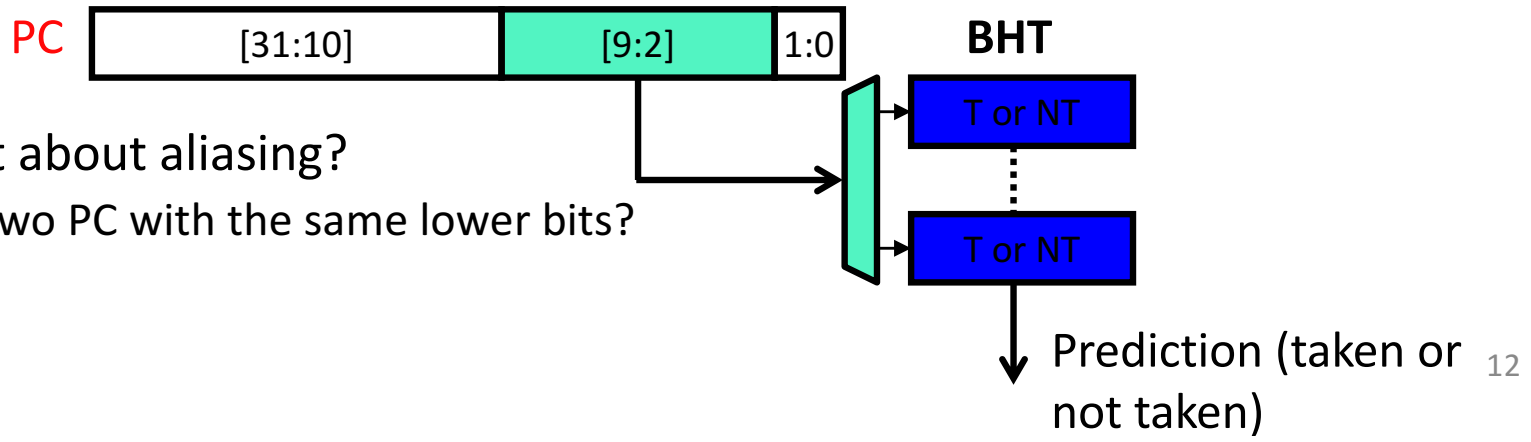
Dynamic Branch Prediction Components



- 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…

Branch Direction Prediction

- Learn from past, predict the future
  - Record the past in a hardware structure
- **Direction predictor (DIRP)**
  - Map conditional-branch PC to taken/not-taken (T/N) decision
  - Individual conditional branches often unbiased or weakly biased
    - 90%+ one way or the other considered **"biased"**
    - Why? Loop back edges, checking for uncommon conditions
- **Branch history table (BHT)**: simplest predictor
  - PC indexes table of bits (0 = N, 1 = T), no tags
  - Essentially: branch will go same way it went last time
  - What about aliasing?
    - Two PC with the same lower bits?

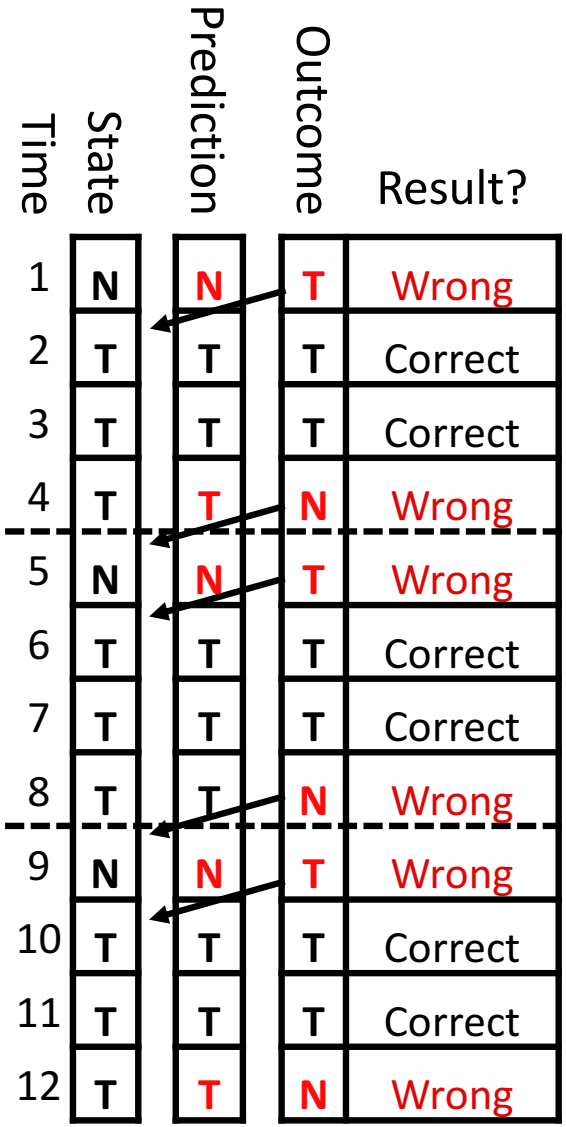


Branch History Table (BHT)

- **Branch history table (BHT)**: simplest direction predictor
  - PC indexes table of bits (0 = N, 1 = T), no tags
  - Essentially: branch will go same way it went last time
  - Problem: **inner loop branch** below

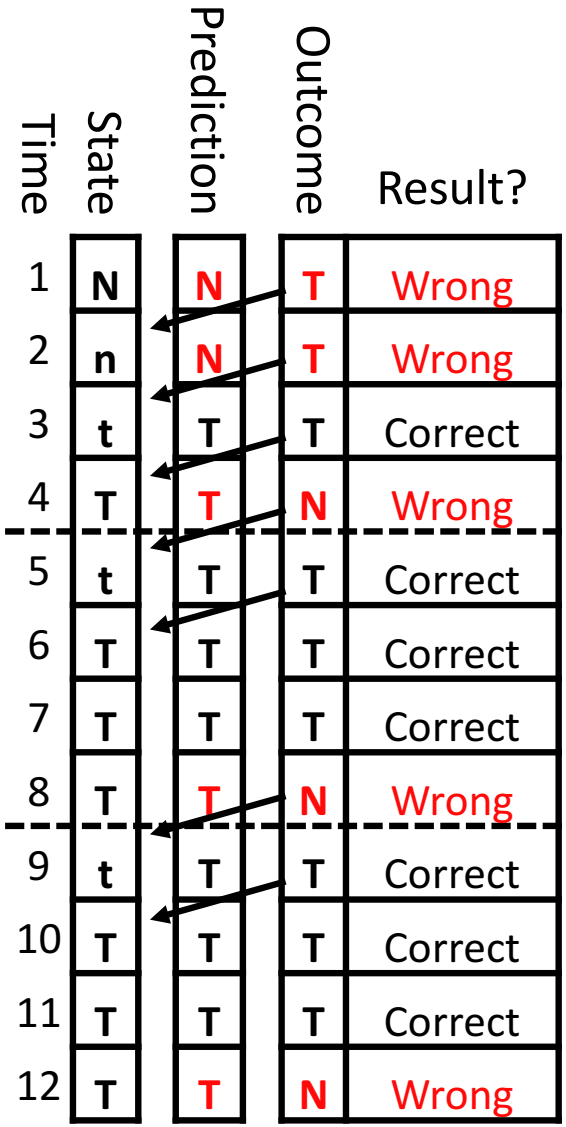
```
for (i=0;i<100;i++)
    for (j=0;j<3;j++)
        // whatever
```

    - Two "built-in" mis-predictions per inner loop iteration
    - Branch predictor "changes its mind too quickly"
  - How can we do better?



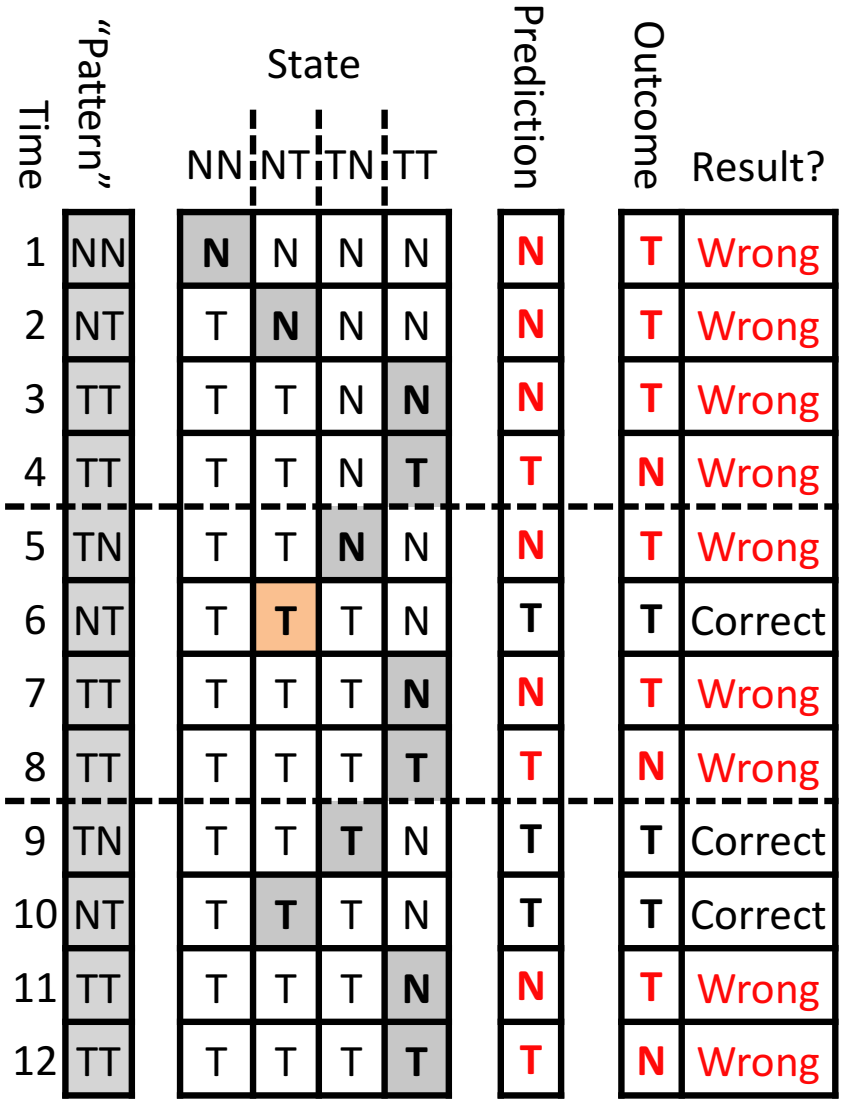
Two-Bit Saturating Counters (2bc)

- **Two-bit saturating counters (2bc)** [Smith 1981]
  - Replace each single-bit prediction
    - (0,1,2,3) = (N,n,t,T)
  - Adds "hysteresis"
    - Force predictor to mis-predict twice before "changing its mind"
  - One mispredict each loop execution (rather than two)
    - \+ Fixes this pathology (which is not contrived, by the way)
    - Can we do even better?



Correlated Predictor

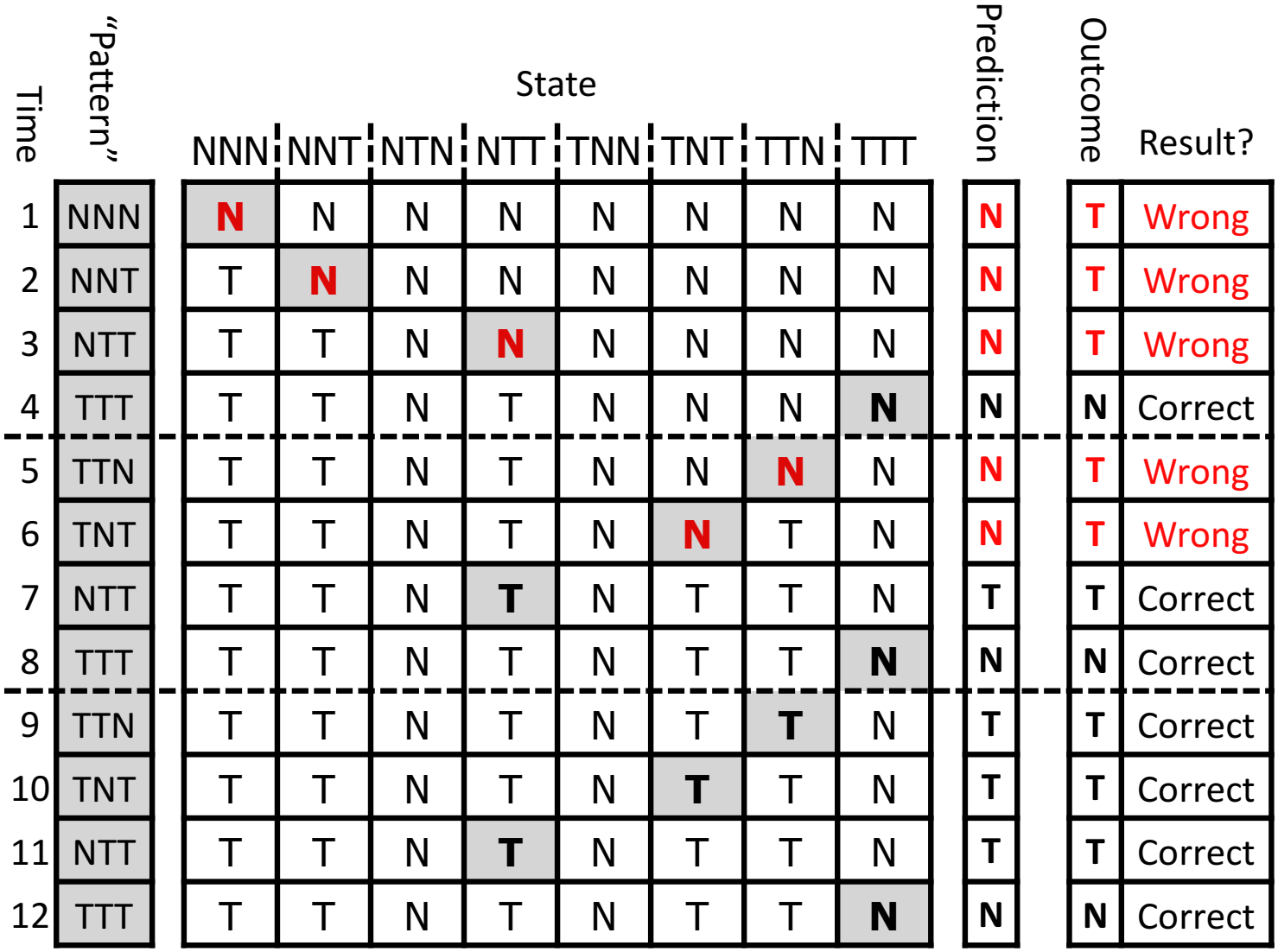
- **Correlated (two-level) predictor** [Patt 1991]
  - Exploits observation that branch outcomes are correlated
  - Maintains separate prediction per (PC, BHR) pairs
    - **Branch history register (BHR)**: recent branch outcomes
  - Simple working example: assume program has one branch
    - BHT: one 1-bit DIRP entry
    - BHT+**2BHR**: 2^2 = **4** 1-bit DIRP entries
  - Why didn't we do better?
    - BHT not long enough to capture pattern



Time	"Pattern"	State NN	State NT	State TN	State TT	Prediction	Outcome	Result?
1	NN	**N**	N	N	N	N	T	Wrong
2	NT	T	**N**	N	N	N	T	Wrong
3	TT	T	T	N	**N**	N	T	Wrong
4	TT	T	T	N	**T**	T	N	Wrong
5	TN	T	T	**N**	N	N	T	Wrong
6	NT	T	**T**	T	N	T	T	Correct
7	TT	T	T	T	**N**	N	T	Wrong
8	TT	T	T	T	**T**	T	N	Wrong
9	TN	T	T	**T**	N	T	T	Correct
10	NT	T	**T**	T	N	T	T	Correct
11	TT	T	T	T	**N**	N	T	Wrong
12	TT	T	T	T	**T**	T	N	Wrong

Correlated Predictor – 3 Bit Pattern

- **Try 3 bits of history**
- 2^3 DIRP entries per pattern



Time	"Pattern"	State: NNN	NNT	NTN	NTT	TNN	TNT	TTN	TTT	Prediction	Outcome	Result?
1	NNN	**N**	N	N	N	N	N	N	N	N	T	Wrong
2	NNT	T	**N**	N	N	N	N	N	N	N	T	Wrong
3	NTT	T	T	N	**N**	N	N	N	N	N	T	Wrong
4	TTT	T	T	N	T	N	N	N	**N**	N	N	Correct
5	TTN	T	T	N	T	N	N	**N**	N	N	T	Wrong
6	TNT	T	T	N	T	N	**N**	T	N	N	T	Wrong
7	NTT	T	T	N	**T**	N	T	T	N	T	T	Correct
8	TTT	T	T	N	T	N	T	T	**N**	N	N	Correct
9	TTN	T	T	N	T	N	T	**T**	N	T	T	Correct
10	TNT	T	T	N	T	N	**T**	T	N	T	T	Correct
11	NTT	T	T	N	**T**	N	T	T	N	T	T	Correct
12	TTT	T	T	N	T	N	T	T	**N**	N	N	Correct

+ No mis-predictions after predictor learns all the relevant patterns!

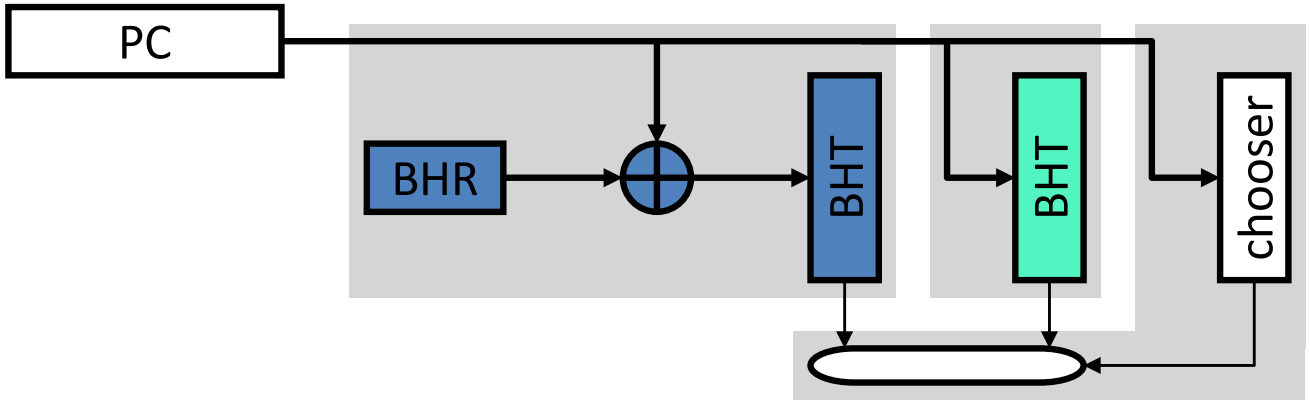
Correlated Predictor Design options

- Design choice I: one **global** BHR or one per PC (**local**)?
  - Each one captures different kinds of patterns
  - Global is better, *why?*

- Design choice II: how many history bits (BHR size)?
  - Tricky one
  + Given unlimited resources, longer BHRs are better, but…
  - BHT utilization decreases
    - Many history patterns are never seen
    - Many branches are history independent (don't care)
    - PC xor BHR allows multiple PCs to dynamically share BHT
    - BHR length < $\log_2$(BHT size)
  - Predictor takes longer to train
  - Typical length: 8–12

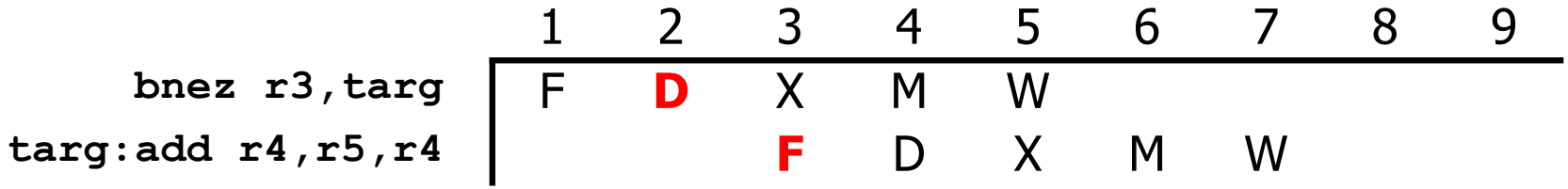
Hybrid Predictor

- **Hybrid (tournament) predictor** [McFarling]
  - Attacks correlated predictor BHT utilization problem
  - Idea: combine two predictors
    - **Simple BHT** predicts history independent branches
    - **Correlated predictor** predicts only branches that need history
    - **Chooser** assigns branches to one predictor or the other
    - Branches start in simple BHT, move mis-prediction threshold
  + Correlated predictor can be made smaller, handles fewer branches
  + 90–95% accuracy



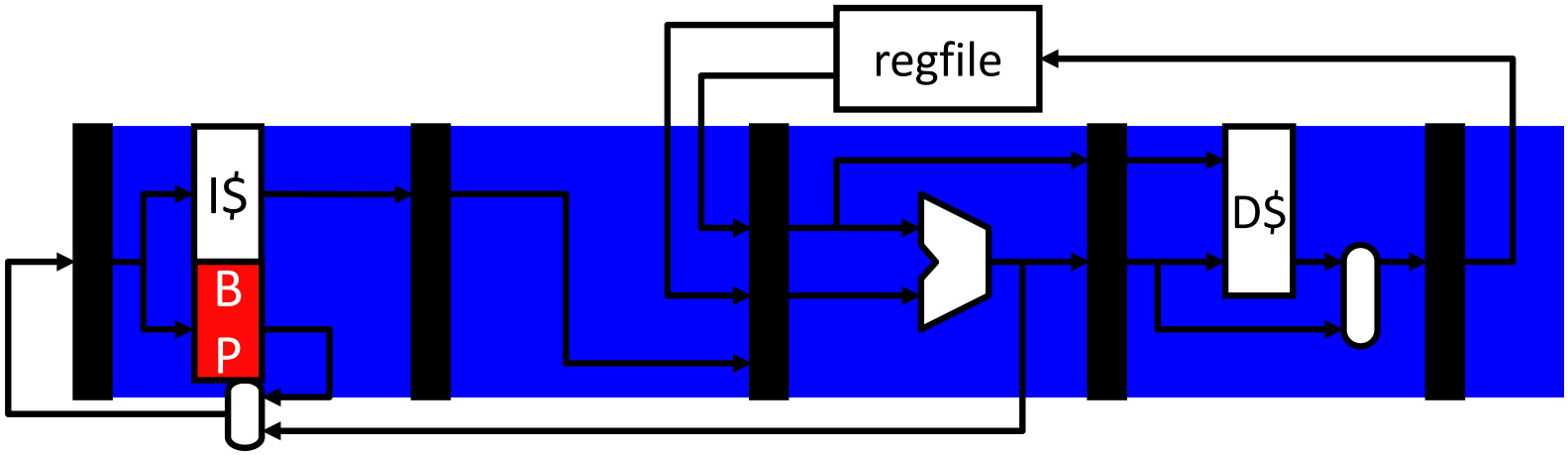
When to Perform Branch Prediction?

- 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



- During Fetch?
  - How do we do that?

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** (as before)
- 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 (BTB)

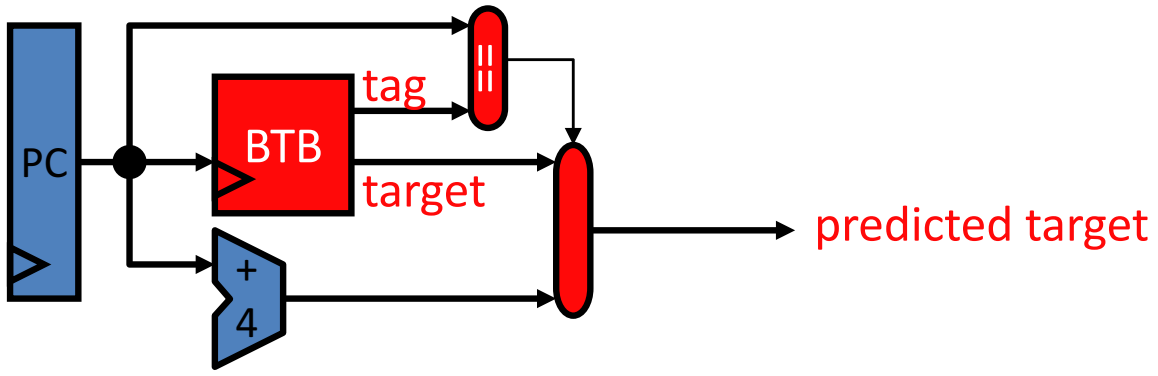
- As before: learn from past, predict the future
  - Record the past branch targets in a hardware structure

- **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"

- Operation
  - Like a cache: address = PC, data = target-PC
  - Access at Fetch *in parallel* with instruction memory
    - predicted-target = BTB[PC]
  - Updated at X whenever target != predicted-target
    - BTB[PC] = target
  - Aliasing? No problem, this is only a prediction

Branch Target Buffer (continued)

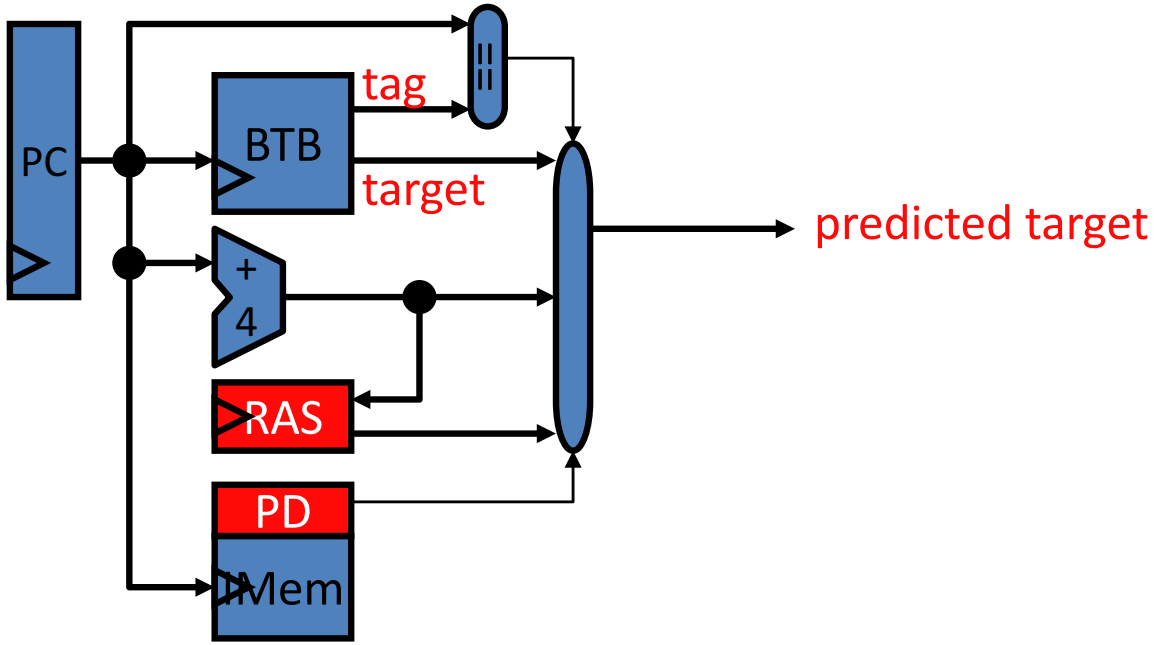
- At Fetch, how does insn know that it's a branch & should read BTB?
  - Answer: it doesn't have to, all insns read BTB
- Key idea: use BTB to predict which insn are branches
  - Tag each entry (with bits of the PC)
    - Just like a cache
  - Tag hit signifies instruction at the PC is a branch
  - Update only on taken branches (thus only taken branches in table)
- Access BTB at Fetch in parallel with instruction memory



Why Does a BTB Work?

- Because most control insns use **direct targets**
  - Target encoded in insn itself → same target every time
- What about **indirect targets**?
  - Target held in a register → can be different each time
  - Indirect conditional jumps are not widely supported
  - Two indirect call idioms
    - + Dynamically linked functions (DLLs): target always the same
    - • Dynamically dispatched (virtual) functions: hard but uncommon
  - Also two indirect unconditional jump idioms
    - • Switches: hard but uncommon
    - – Function returns: hard and common but…

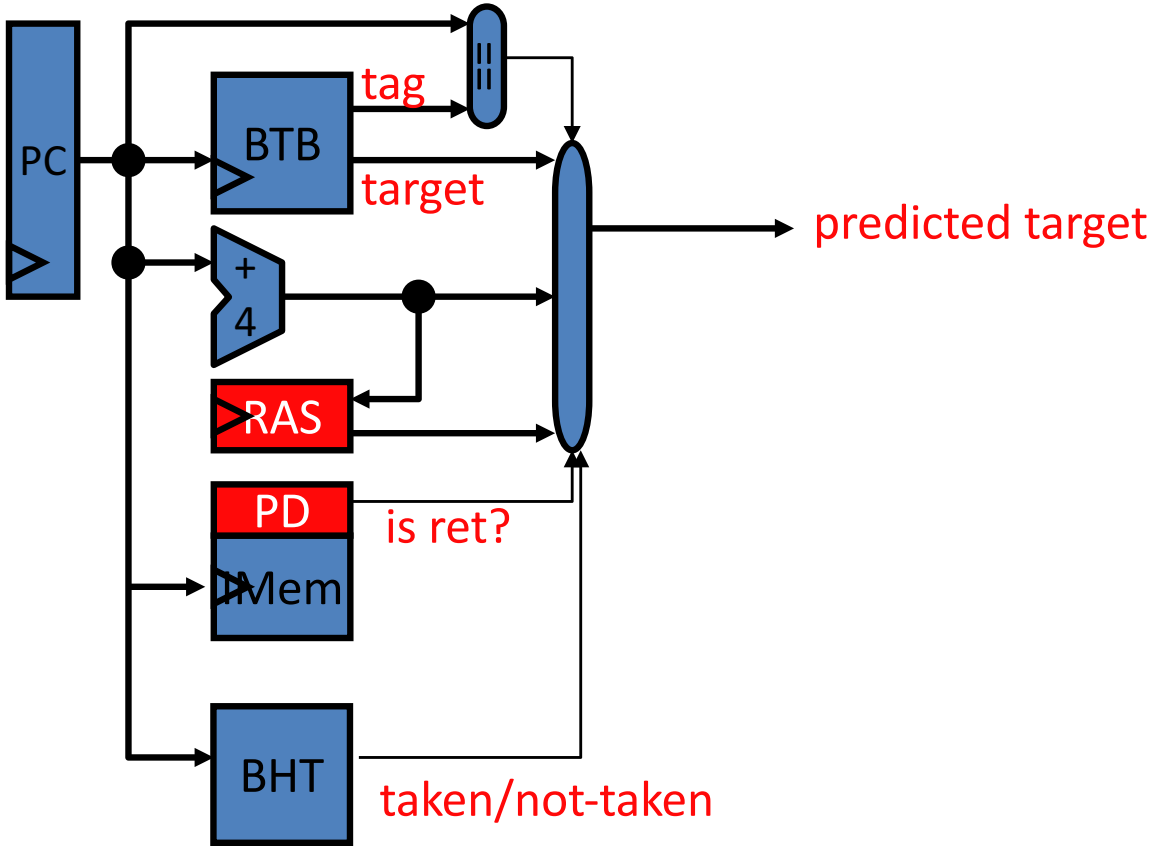
Return Address Stack (RAS)



- **Return address stack (RAS)**
  - Call instruction? RAS[TOS++] = PC+4
  - Return instruction? Predicted-target = RAS[--TOS]
  - Q: how can you tell if an insn is a call/return before decoding it?
    - Accessing RAS on every insn BTB-style doesn't work
  - Answer: **pre-decode bits** in Imem, written when first executed
    - Can also be used to signify branches

Putting It All Together

- BTB & branch direction predictor during fetch



- If branch prediction correct, no taken branch penalty

Branch Prediction Performance

- Dynamic branch prediction
  - Simple predictor at fetch; branches predicted with 75% accuracy
    - CPI = 1 + (20% * **25%** * 2)= **1.1**
  - More advanced predictor at fetch: 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
  - Pipelines are superscalar (later)

Can we get rid of (many) branches?

```
A = Y[i];
if (A == 0)
   A = W[i];
else
   Y[i] = 0;
Z[i] = A*X[i];
```

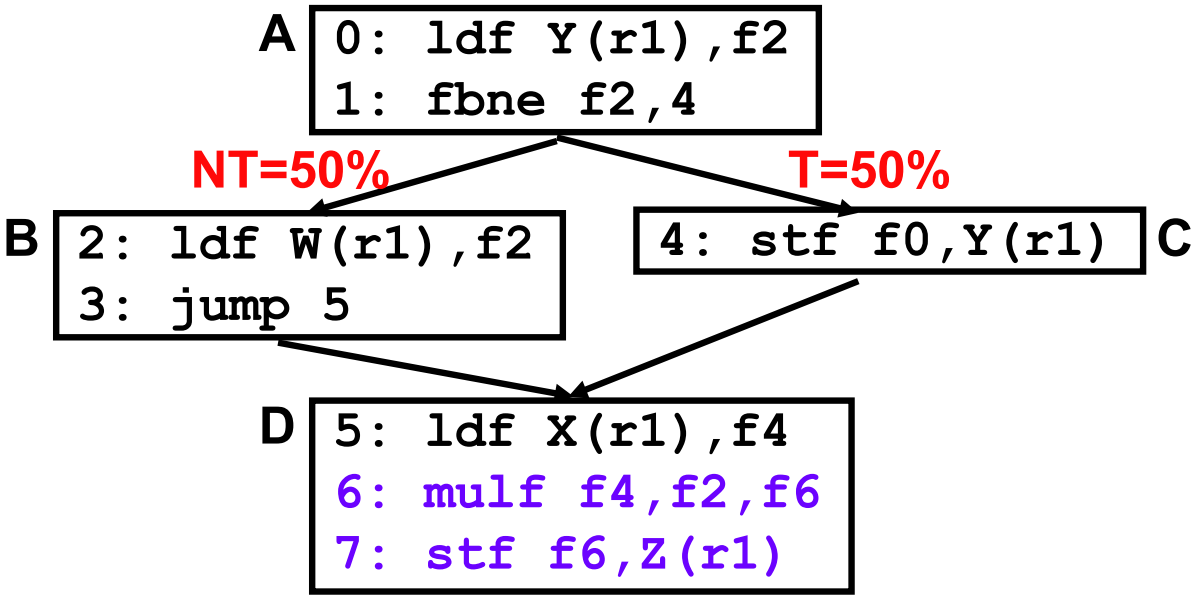
Predication If-Conversion Example

Source code

```
A = Y[i];
if (A == 0)
    A = W[i];
else
    Y[i] = 0;
Z[i] = A*X[i];
```

Machine code

```
0: ldf Y(r1),f2
1: fbne f2,4
2: ldf W(r1),f2
3: jump 5
4: stf f0,Y(r1)
5: ldf X(r1),f4
6: mulf f4,f2,f6
7: stf f6,Z(r1)
```



Using Predication

```
0: ldf Y(r1),f2
1: fspne f2,p1
2: ldf.p p1,W(r1),f2
4: stf.np p1,f0,Y(r1)
5: ldf X(r1),f4
6: mulf f4,f2,f6
7: stf f6,Z(r1)
```

ISA Support for Predication

```
0: ldf Y(r1),f2
1: fspne f2,p1
2: ldf.p p1,W(r1),f2
4: stf.np p1,f0,Y(r1)
5: ldf X(r1),f4
6: mulf f4,f2,f6
7: stf f6,Z(r1)
```

- Itanium: change branch 1 to **set-predicate insn `fspne`**
- Change insns 2 and 4 to **predicated insns**
  - `ldf.p` performs `ldf` if predicate `p1` is true
  - `stf.np` performs `stf` if predicate `p1` is false

Predication Performance

- Cost/benefit analysis
  - Benefit: predication avoids branches
    - Thus avoiding mis-predictions
    - Also reduces pressure on predictor table (fewer branches to track)
  - Cost: extra instructions (fetched, but not actually executed)
- As branch predictors are highly accurate…
  - Might not help:
    - 5-stage pipeline, two instruction on each path of if-then-else
    - No performance gain, likely slower if branch predictable
  - Or even hurt!
  - But can help:
    - Deeper pipelines, hard-to-predict branches, and few added insn
- Thus, prediction is useful, but not a panacea

Avoiding Branches via ISA: Predication

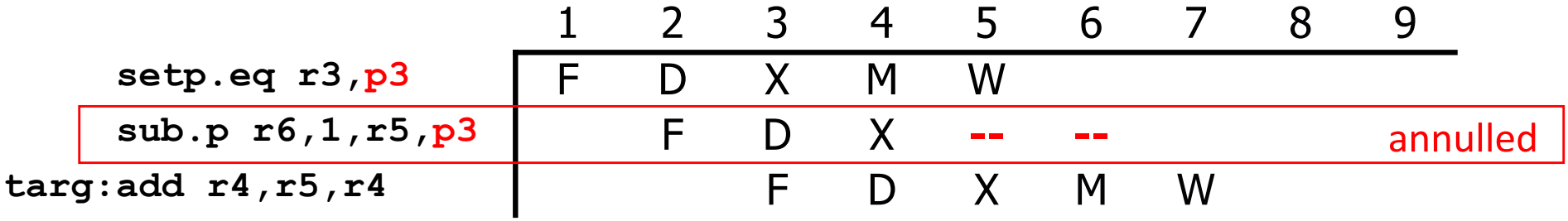
- Conventional control
  - Conditionally executed insns also conditionally fetched

	1	2	3	4	5	6	7	8	9	
`beq r3,targ`	F	D	X	M	W					
`sub r6,1,r5`		F	D	--	--	--				flushed: wrong path
`targ:add r4,r5,r4`			F	--	--	--	--			flushed: why?
`targ:add r4,r5,r4`				F	D	X	M	W		

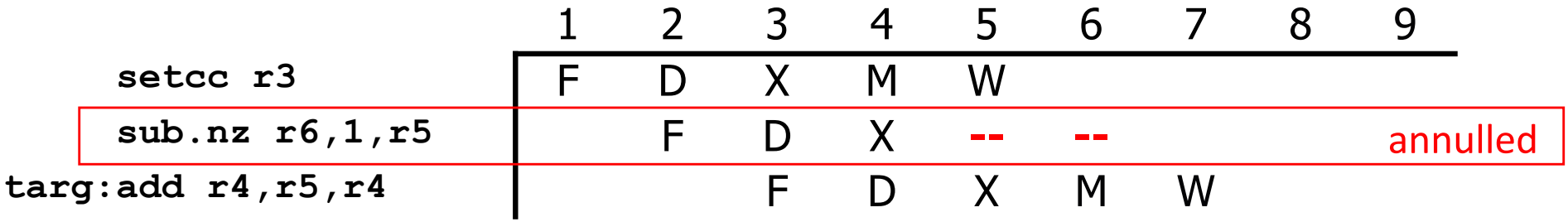
  - If **`beq`** mis-predicts, both **`sub`** and **`add`** must be flushed
  - Waste: **`add`** is independent of mis-prediction
- **Predication**: not prediction, predication
  - ISA support for conditionally-executed unconditionally-fetched insns
  - If **`beq`** mis-predicts, annul **`sub`** in place, preserve **`add`**
    - Example is if-then, but if-then-else can be predicated too
  - How is this done? How does **`add`** get correct value for **`r5`**

Full Predication

- **Full predication**
  - Every insn can be annulled, annulment controlled by…
  - Predicate registers: additional register in each insn (e.g., IA64)



  - Predicate codes: condition bits in each insn (e.g., ARM)



  - Only ALU insn shown (**sub**), but this applies to all insns, even stores
  - Branches replaced with "set-predicate" insns

Conditional Register Moves (CMOVs)

- **Conditional (register) moves**
  - Construct appearance of full predication from one primitive

```
cmoveq r1,r2,r3          // if (r1==0) r3=r2;
```

  - May require some code duplication to achieve desired effect
  - Painful, potentially impossible for some insn sequences
  - Requires more registers
  - Only good way of retro-fitting predication onto ISA (e.g., IA32, Alpha)

```
      sub r6,1,r9
      cmovne r3,r9,r5
targ:add r4,r5,r4
```

	1	2	3	4	5	6	7	8	9
sub r6,1,r9		D	X	M	W				
cmovne r3,r9,r5		F	D	X	M	W			
targ:add r4,r5,r4			F	D	X	M	W		

Predication Performance

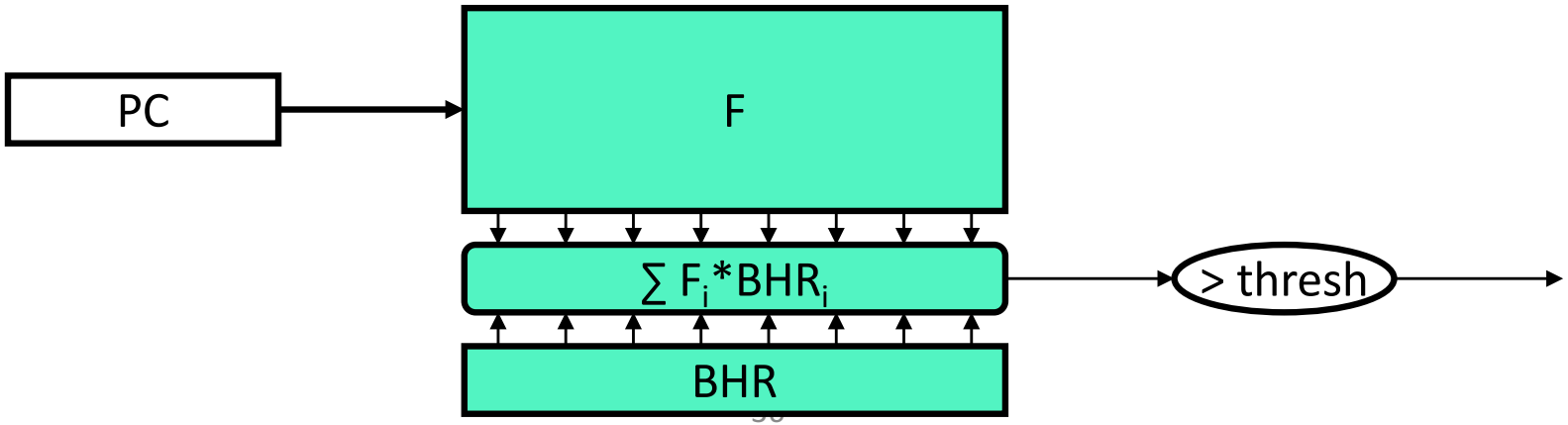
- Predication overhead is additional insns
  - Sometimes overhead is zero
    - Not-taken if-then branch: predicated insns executed
  - Most of the times it isn't
    - Taken if-then branch: all predicated insns annulled
    - Any if-then-else branch: half of predicated insns annulled
    - Almost all cases if using conditional moves
- Calculation for a given branch, predicate (vs speculate) if…
  - Average number of additional insns > overall mis-prediction penalty
  - For an individual branch
    - Mis-prediction penalty in a 5-stage pipeline = 2
    - Mis-prediction rate is <50%, and often <20%
    - Overall mis-prediction penalty <1 and often <0.4
  - So when is predication worth it?

Predication Performance

- What does predication actually accomplish?
  - In a scalar 5-stage pipeline (penalty = 2): nothing
  - In a 4-way superscalar 15-stage pipeline (penalty = 60): something
    - Use when mis-predictions >10% and insn overhead <6
  - In a 4-way out-of-order superscalar (penalty ~ 150)
    - Should be used in more situations
  - Still: only useful for branches that mis-predict frequently
- Strange: ARM typically uses scalar 5-9 stage pipelines
  - Why is the ARM ISA predicated then?
  - Low-power: eliminates the need for a large branch predictor
  - Real-time: predicated code performs consistently
  - Loop scheduling: effective software pipelining requires predication

Research: Perceptron Predictor

- **Perceptron predictor** [Jimenez]
  - – Attacks BHR size problem using machine learning approach
  - – BHT replaced by table of function coefficients F_i (signed)
  - – Predict taken if $\sum(BHR_i * F_i) >$ threshold
  - \+ Table size #PC*|BHR|*|F| (can use long BHR: ~60 bits)
    - – Equivalent correlated predictor would be $\#PC*2^{|BHR|}$
  - – How does it learn? Update F_i when branch is taken
    - $BHR_i == 1 \; ? \; F_i{+}{+} : F_i{-}{-};$
    - "don't care" F_i bits stay near 0, important F_i bits saturate
  - \+ Hybrid BHT/perceptron accuracy: 95–98%



More Research: GEHL Predictor

- Problem with both correlated predictor and perceptron
  - Same BHT real-estate dedicated to 1st history bit (1 column) …
  - … as to 2nd, 3rd, 10th, 60th…
  - Not a good use of space: 1st bit much more important than 60th

- **GEometric History-Length predictor** [Seznec, ISCA'05]
  - Multiple BHTs, indexed by geometrically longer BHRs (0, 4, 16, 32)
    - BHTs are (partially) tagged, not separate "chooser"
    - Predict: use matching entry from BHT with longest BHR
    - Mis-predict: create entry in BHT with longer BHR
  - \+ Only 25% of BHT used for bits 16-32 (not 50%)
    - Helps amortize cost of tagging
  - \+ Trains quickly
  - – 95-97% accurate

Championship Branch Prediction

- **CBP**
  - Workshop held in conjunction with MICRO
  - Submitted code is tested on standard branch traces
  - Highest prediction accuracy wins

- Two tracks
  - Idealistic: predictor simulator must run in under 2 hours
  - Realistic: predictor must synthesize into 32KB + 256 bits or less