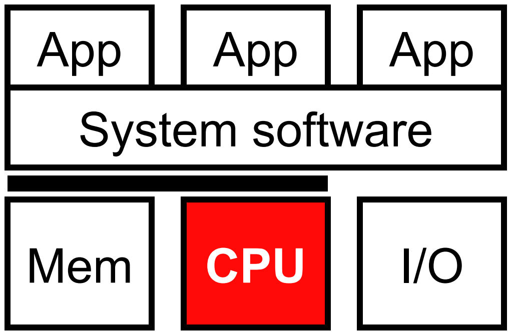
CIS 5710
Computer Organization and Design

Unit 7: Branch Prediction

Based on slides by Profs. Amir Roth, Milo Martin & C.J. Taylor

This Unit: Branch Prediction



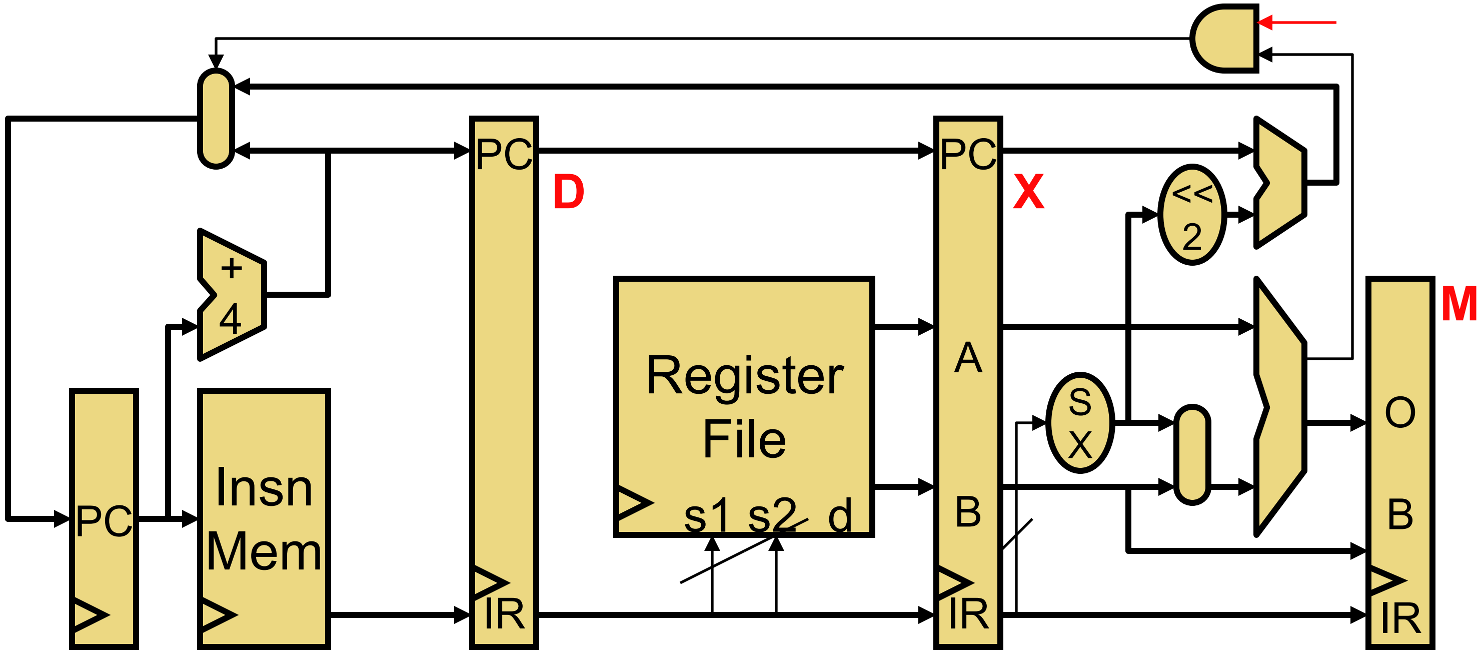
- Control hazards
  - Branch prediction

Readings

- P&H
  - Chapter 4

Control Dependences and Branch Prediction

What About Branches?



- Wait for branch outcome (two-cycle penalty)
- **Fetch past BR insn before BR outcome is known**
  - Default: assume "**not-taken**" (at fetch, can't tell it's a branch)

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)
- **Control speculation**: speculation aimed at control hazards
  - Unknown parameter: are these the correct insns to execute next?

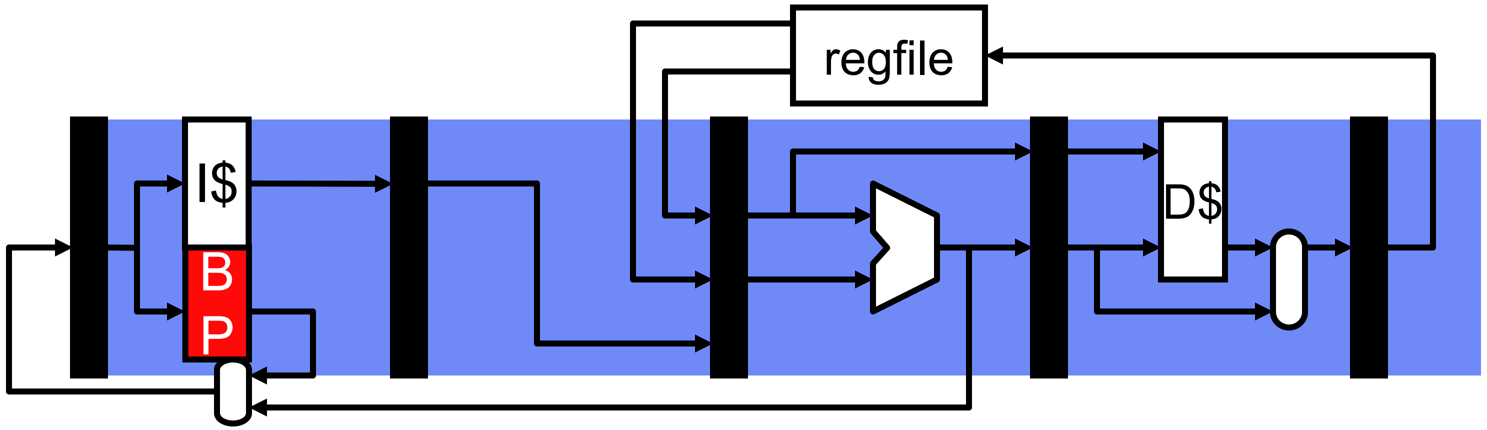
Control Speculation Mechanics

- Guess branch target, start fetching at guessed position
  - Doing nothing is implicitly guessing target is the next sequential PC
    - We were **already** speculating before!
  - Can actively guess other targets: **dynamic branch prediction**

- Execute branch to verify (check) guess
  - Correct speculation? keep going
  - Mis-speculation? Flush mis-speculated insns
    - Hopefully haven't modified permanent state (Regfile, DMem)

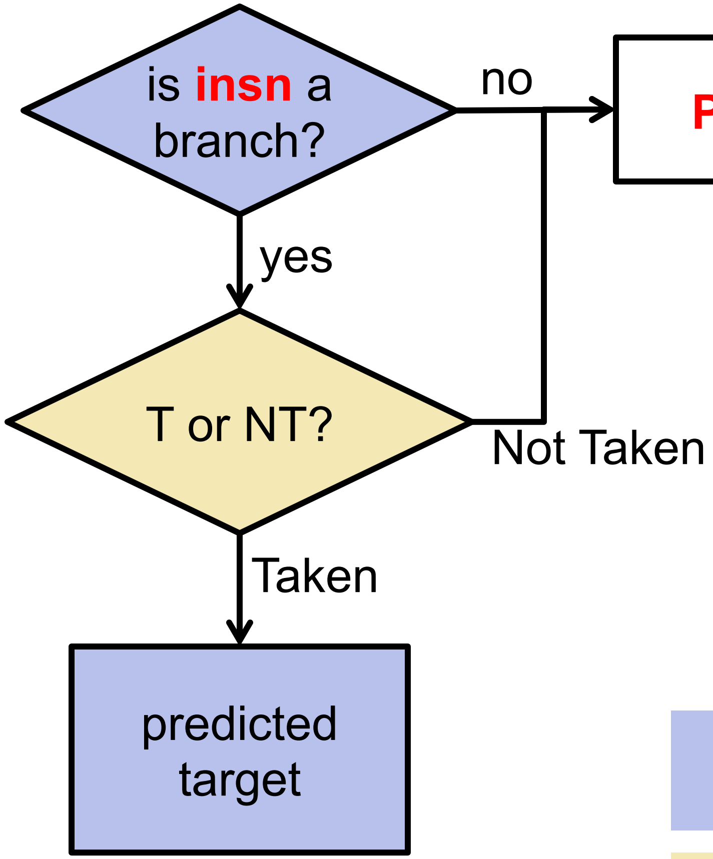
    + Happens naturally in in-order 5-stage pipeline

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 Prediction Steps



- Which insn's behavior are we trying to predict?
- Where does **PC** come from?

hardware structure:

branch target buffer

direction predictor

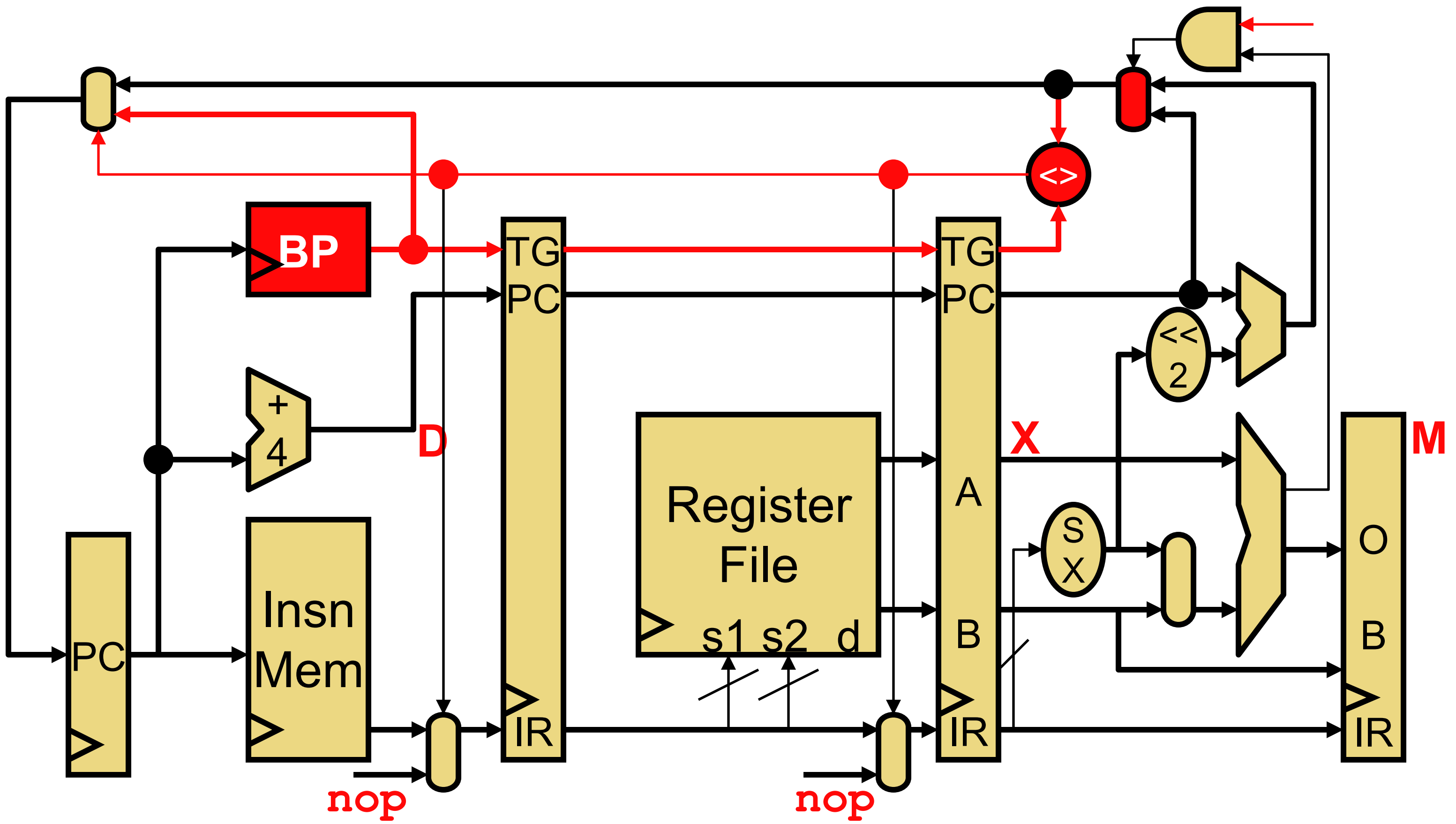
When to Perform Branch Prediction?

- 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?

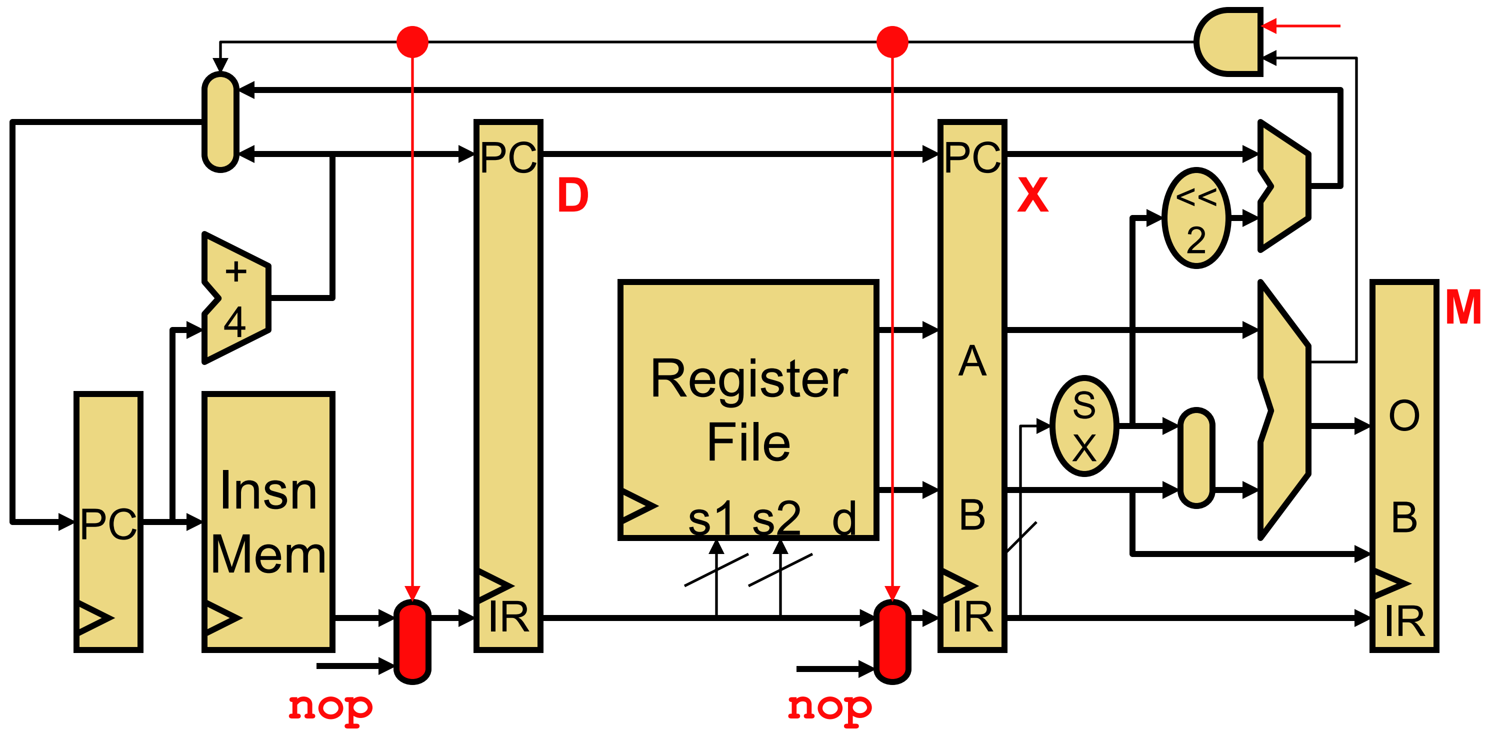
Dynamic Branch Prediction



- **Dynamic branch prediction**: hardware guesses outcome
  - Start fetching from guessed address
  - Flush on **mis-prediction**

MISPREDICTION RECOVERY

Branch Recovery



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

Branch Speculation and Recovery

Correct:

	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

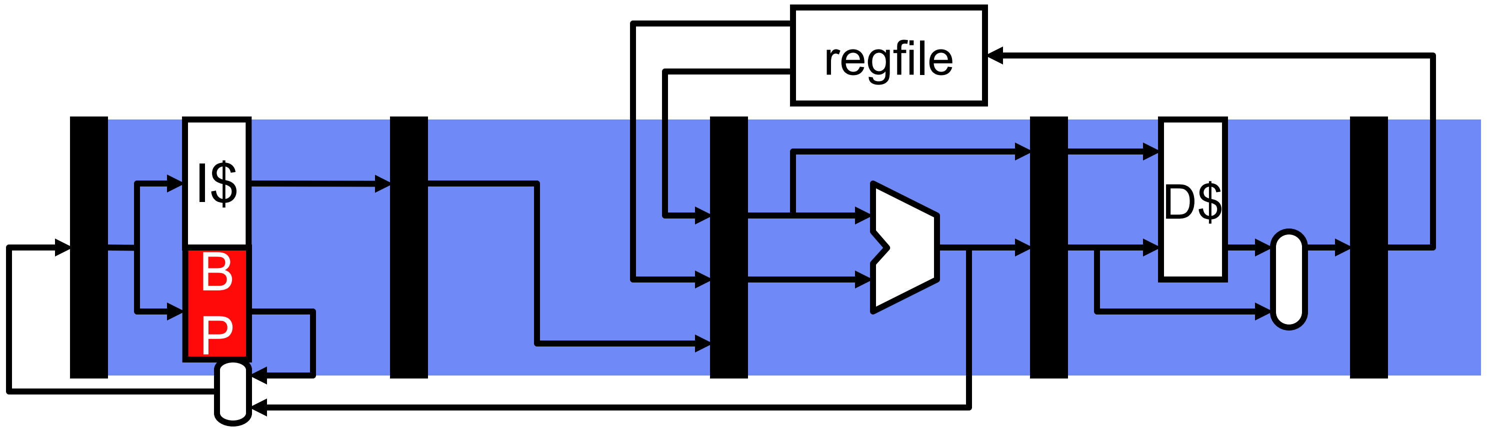
- **Mis-speculation recovery**: what to do on wrong guess
  - Not too painful in a short, in-order pipeline
  - Branch resolves in X
  - + Younger insns (in F, D) haven't changed permanent state
  - **Flush** insns currently in D and X (i.e., replace with **`nops`**)

Recovery:

	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**	--	--	--		
~~`mul r10←r8,r9`~~				**F**	--	--	--	--	
`targ:add r4←r4,r5`					**F**	D	X	M	W

IS IT A BRANCH?

Revisiting Branch Prediction Components

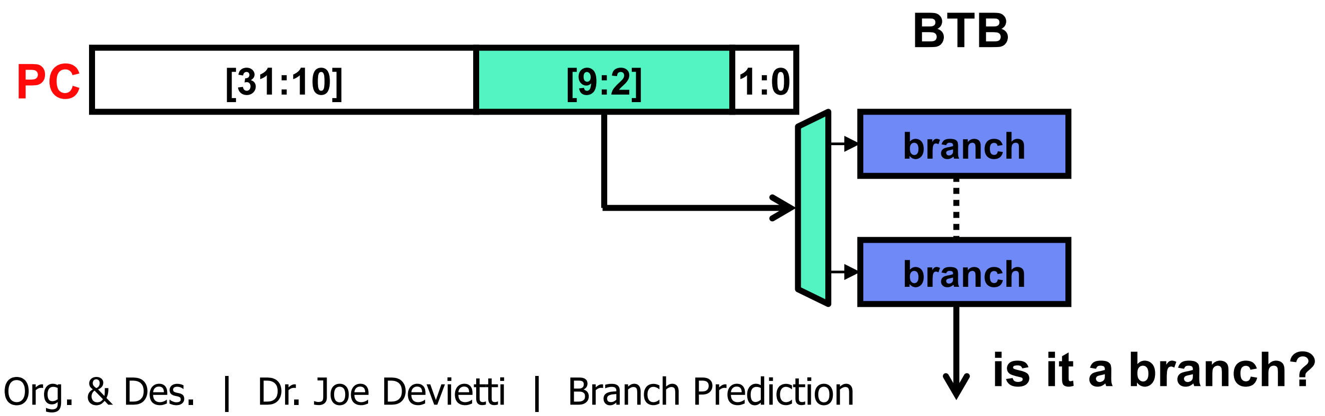


- Step #1: is it a branch?
  - Easy after decode... during fetch: **predictor**
- Step #2: is the branch taken or not taken?
  - Direction predictor (later)
- Step #3: if the branch is taken, where does it go?
  - Branch target predictor (BTB)
  - Supplies target PC if branch is taken

Branch Target Buffer

- **Learn from the past to predict the future**
  - Record the past in a hardware structure

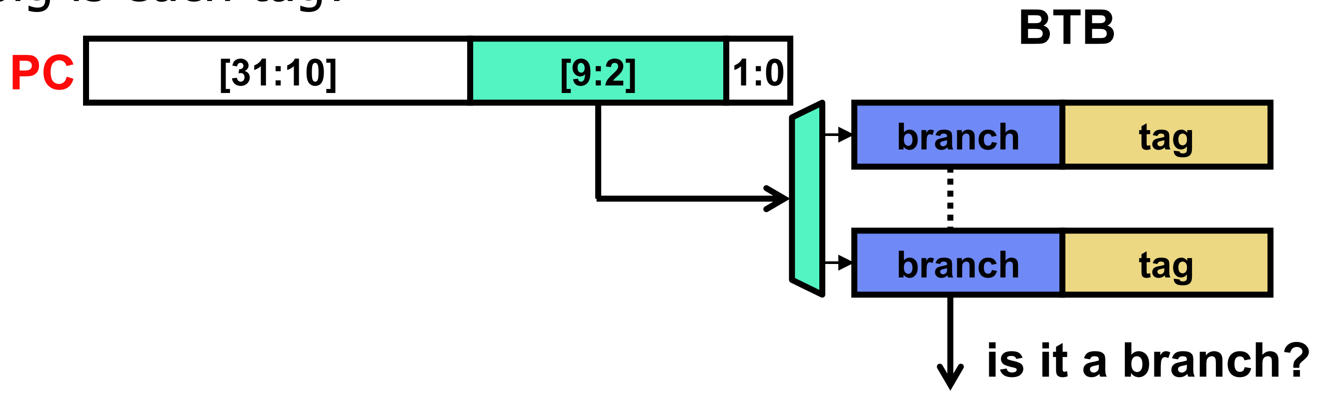
- **Branch target buffer (BTB)**:
  - Record a list of branches we have seen
    - + code doesn't change
  - PC indexes table of bits
    - each entry is 1 bit: is there a branch here?
    - set the bit if we see a branch at that index
    - What about two PCs with the same lower bits…?



Branch Target Buffer

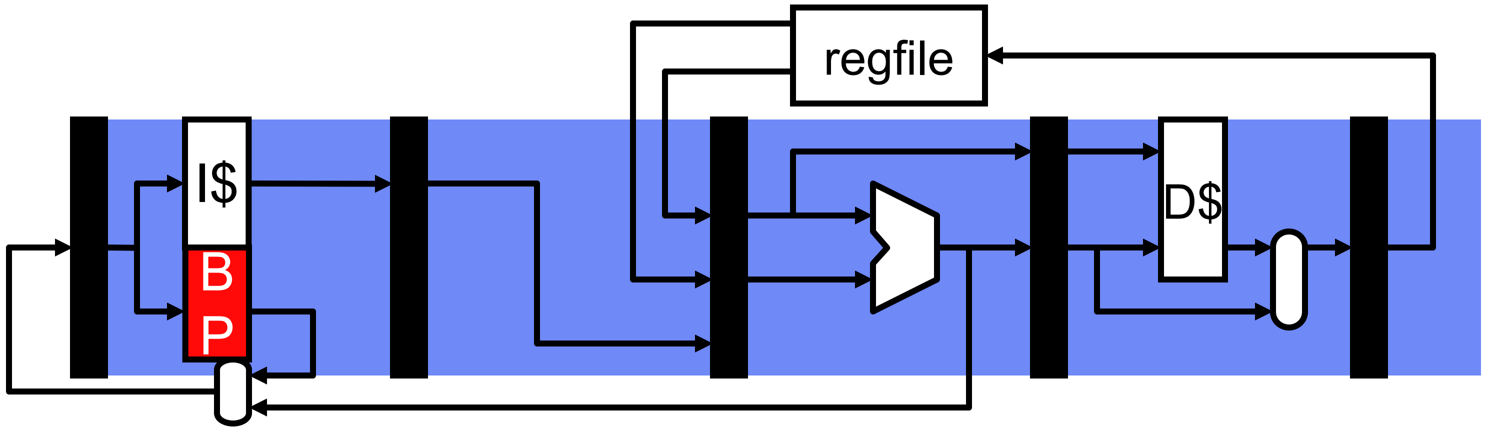
- BTB entries are too coarse-grained

+ 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

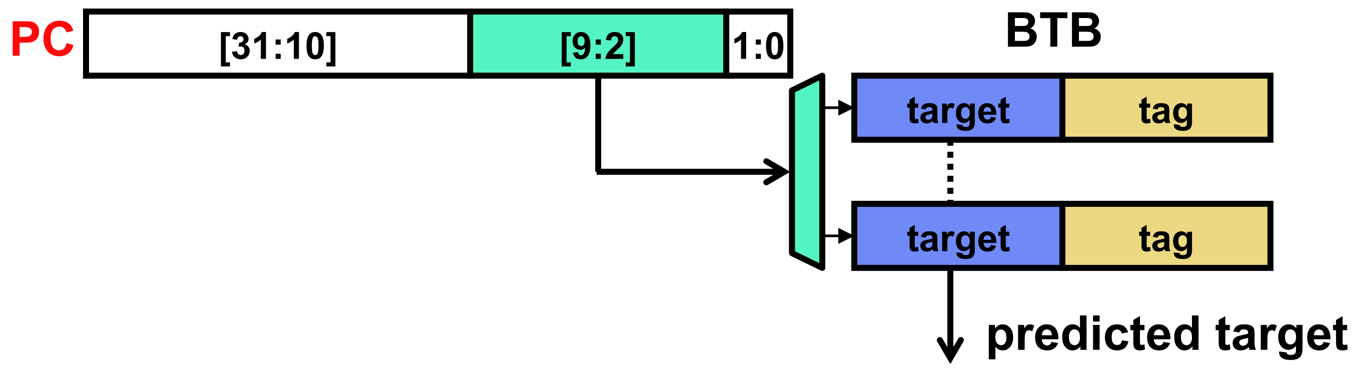
Revisiting Branch Prediction Components



- 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

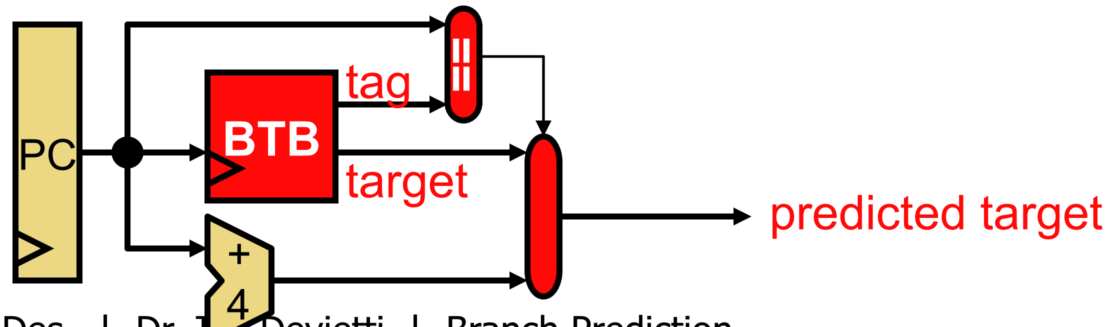
Branch Target Buffer, Again

- **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?
    - No problem, just a prediction!



Branch Target Buffer (continued)

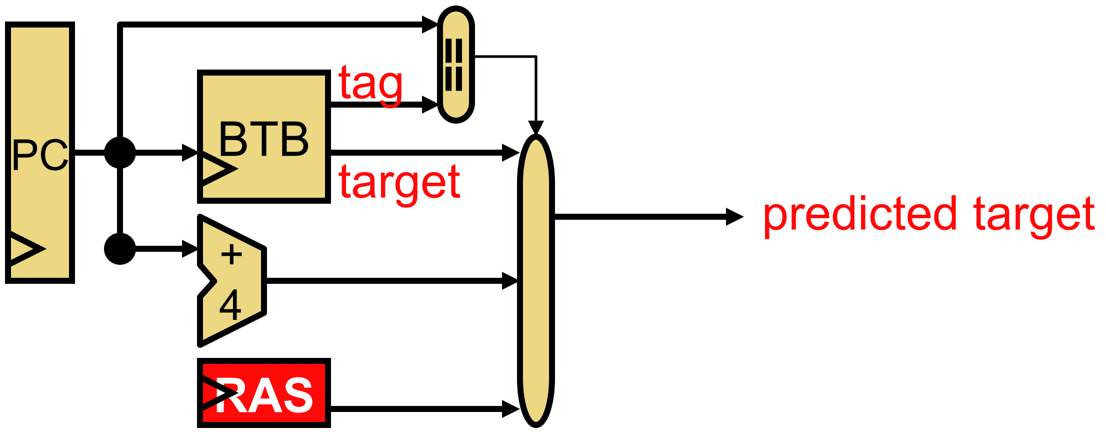
- At Fetch, how do we know we have a branch? We don't…
  - **…all insns access BTB in parallel with Imem Fetch**
- **BTB predicts which insn are branches, and targets**
  - tag each entry with its corresponding PC
  - Update BTB on every taken branch insn, record target PC:
    - BTB[PC].tag = PC, BTB[PC].target = 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



Why Does a BTB Work?

- Because most control insns use **direct targets**
  - Target encoded in insn itself → same "taken" target every time
- What about **indirect targets**?
  - Target held in a register → can be different each time
  - 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**

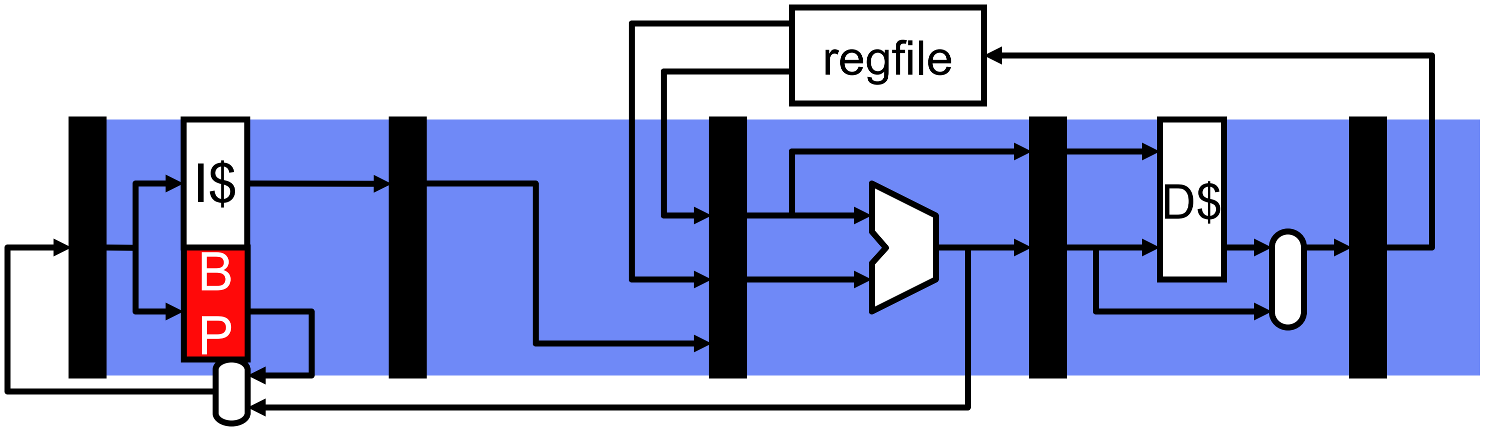
Return Address Stack (RAS)



- **Return address stack (RAS)**
  - Call instruction? RAS[TopOfStack++] = PC+4
  - Return instruction? Predicted-target = RAS[--TopOfStack]
  - Q: how can you tell if an insn is a call/return before decoding it?
    - mark some BTB entries as "returns", or use another table

BRANCH DIRECTION PREDICTION

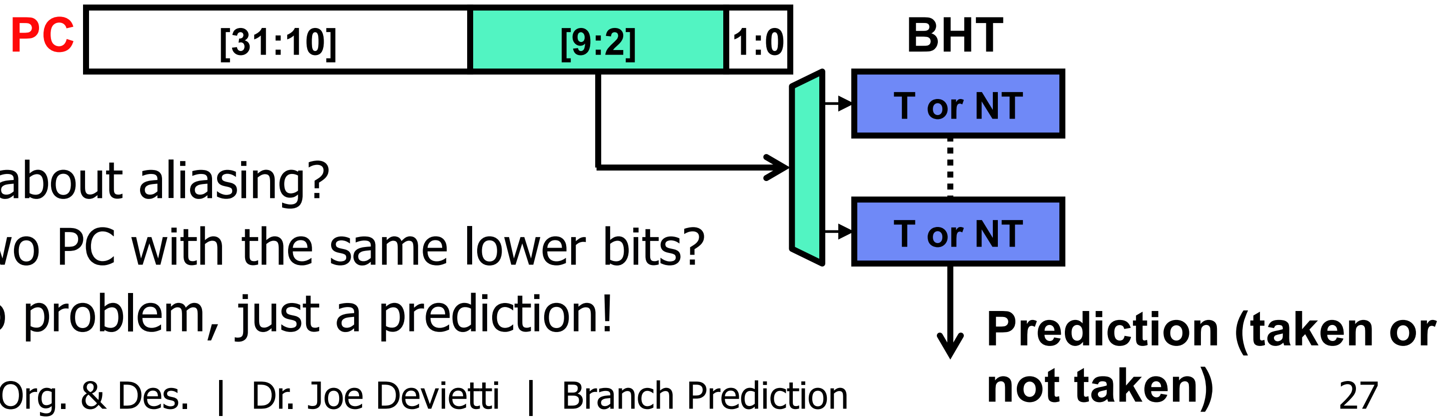
Revisiting Branch Prediction Components



- 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

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 biased or weakly biased
    - 90%+ one way or the other considered **"biased"**
    - Why? Loop back edges, checking for uncommon conditions
- **Bimodal predictor**: simplest predictor
  - PC indexes Branch History 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?
  - No problem, just a prediction!

Bimodal Branch Predictor

- 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"

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

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?

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

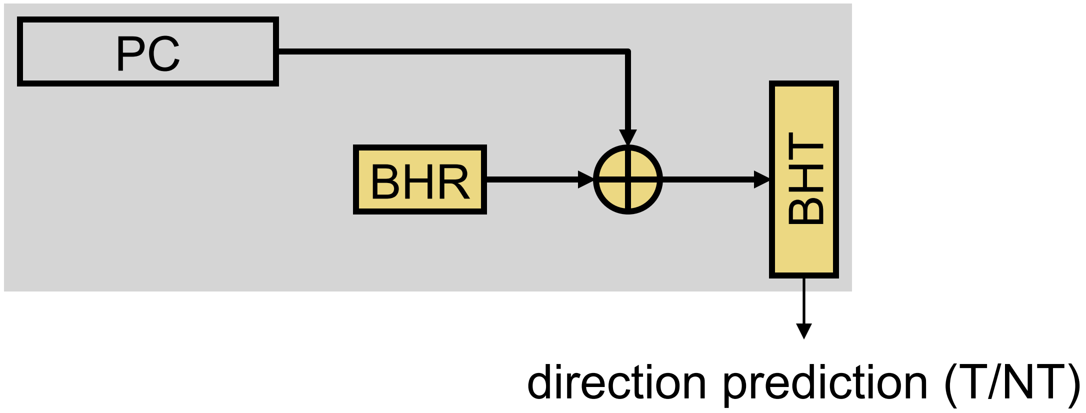
Branches may be correlated

- Consider:

```
for (i=0; i<1000000; i++) {   // Highly biased
    if (i % 3 == 0) {          // Locally correlated
        …
    }
    if (random() % 2 == 0) {   // Unpredictable
        …
    }
    if (i % 3 == 0) {
        …                // Globally correlated
    }
}
```

Gshare History-Based Predictor

- Exploits observation that branch outcomes are correlated
- Maintains recent branch outcomes in **Branch History Register (BHR)**
    - In addition to BHT of counters (typically 2-bit sat. counters)
- How do we incorporate history into our predictions?
    - Use PC xor BHR to index into BHT. Why?



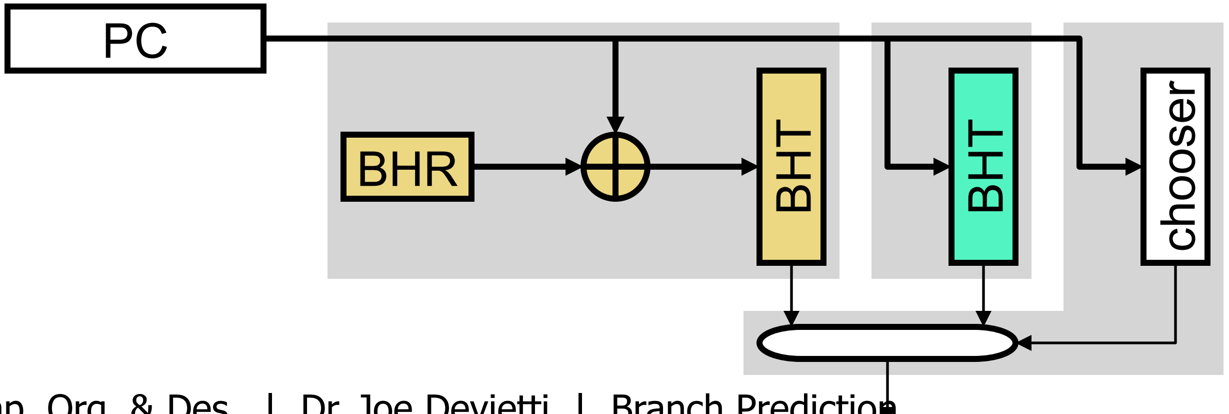
Gshare History-based Predictor

- Gshare working example
  - assume program has one branch
  - BHT: one 1-bit DIRP entry
  - **3BHR**: last 3 branch outcomes
  - train counter, **and** update BHR after each branch

Time	State	BHR	Prediction	Outcome	Result?
1	**N**	**NNN**	N	T	wrong
2	**N**	**NNT**	N	T	wrong
3	**N**	**NTT**	N	T	wrong
4	**N**	**TTT**	N	N	correct
5	**N**	**TTN**	N	T	wrong
6	**N**	**TNT**	N	T	wrong
7	**T**	**NTT**	T	T	correct
8	**N**	**TTT**	N	N	correct
9	**T**	**TTN**	T	T	correct
10	**T**	**TNT**	T	T	correct
11	**T**	**NTT**	T	T	correct
12	**N**	**TTT**	N	N	correct

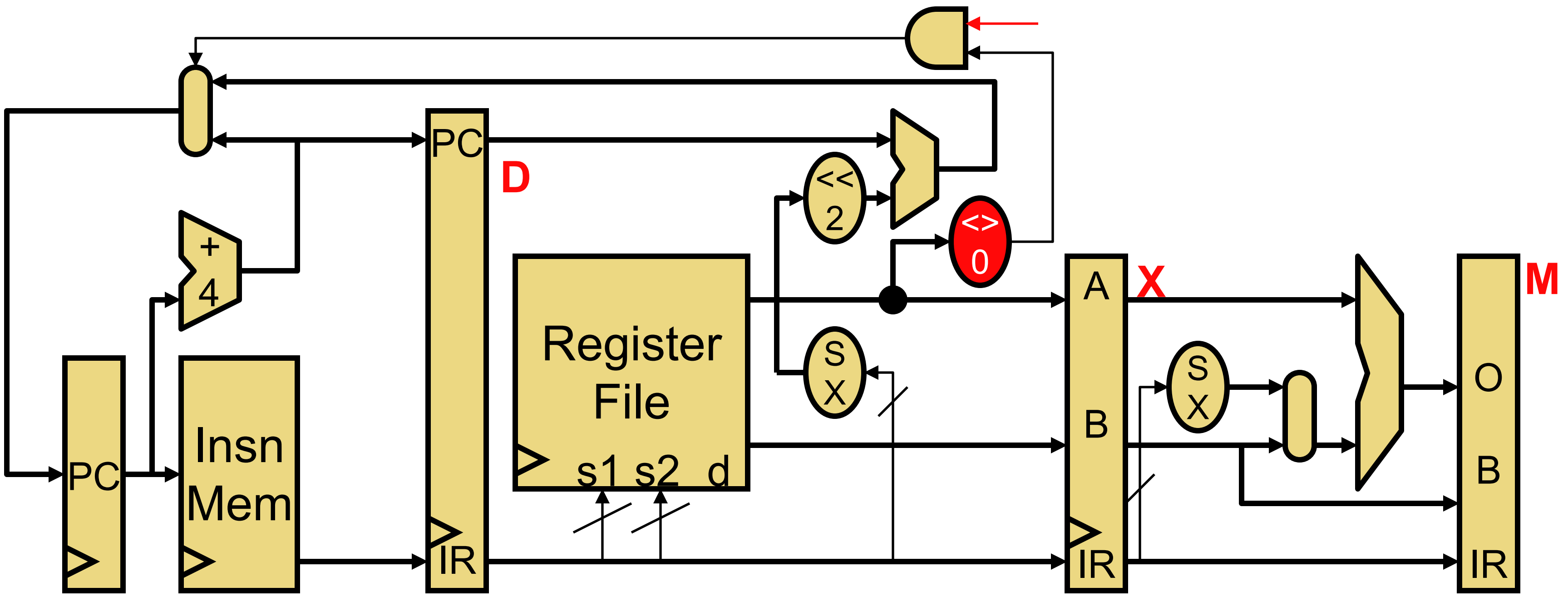
Hybrid Predictor

- **Hybrid (tournament) predictor** [McFarling 1993]
  - Attacks correlated predictor BHT capacity problem
  - Idea: combine two predictors
    - **Simple bimodal predictor** for history-independent branches
    - **Correlated predictor** for 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



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**

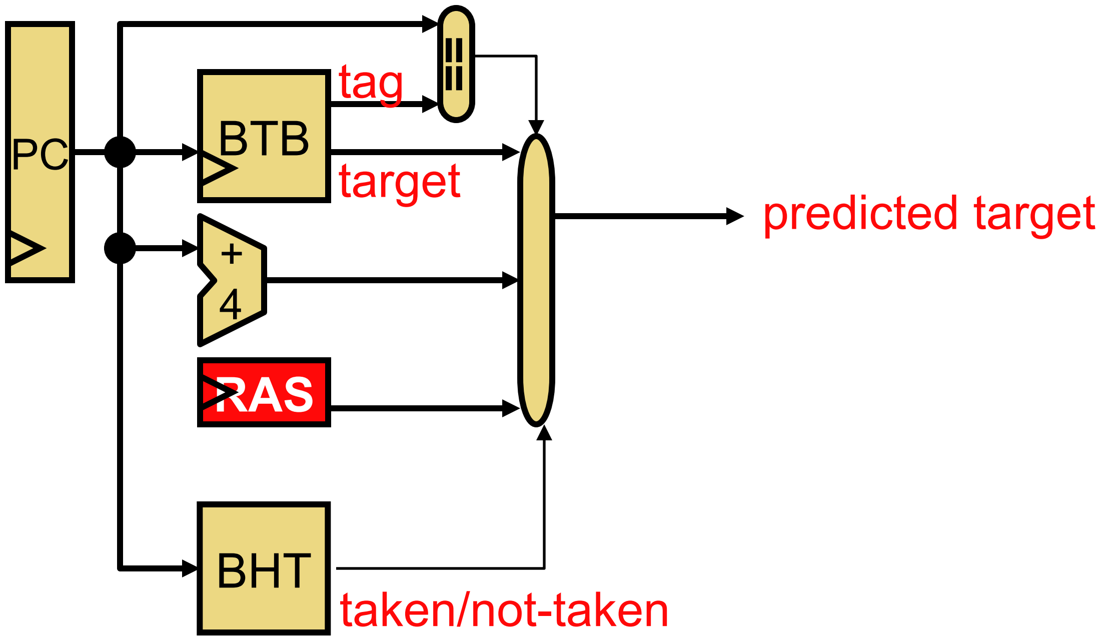
- But wait, fast branches assume only simple comparisons
  - Fine for MIPS
  - But not fine for ISAs with “branch if $1 > $2” operations

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

- Example of ISA and micro-architecture interaction
  - Type of branch instructions
  - Another option: “Delayed branch” or “branch delay slot”
  - What about condition codes?

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)

PREDICATION

Predication

- Instead of predicting which way we're going, why not go **both ways**?
  - compute a **predicate bit** indicating a condition
  - ISA includes **predicated instructions**
    - predicated insns either execute as normal or as NOPs, depending on the predicate bit
- Examples
  - x86 cmov performs conditional load/store
  - 32b ARM allows almost all insns to be predicated
    - 64b ARM has predicated reg-reg move, inc, dec, not
  - Nvidia GPU ISA supports predication on most insns
  - predicate bits are like LC4 NZP bits
    - x86 FLAGS, ARM condition codes

Predication Example

- Instead of predicting which way we're going, why not go both ways?
  - compute a predicate bit indicating a condition
  - ISA includes predicated instructions
    - predicated insns either execute as normal or as NOPs, depending on the predicate bits

```
// C code
if (a <= b) {
  x += y;
} else {
  x -= z;
}
```

```
; original LC4
CMP R1 R2
BRn else
ADD R3 <- R3 R4
JMP after
else:
SUB R3 <- R3 R5
after:
```

```
; predicated LC4
CMP R1 R2
ADDzp R3 <- R3 R4
SUBn R3 <- R3 R5
```

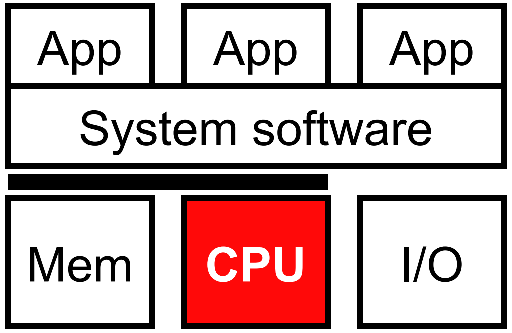
Predication Performance

- Predication overhead is additional insns
  - Sometimes overhead is zero
    - for if-then statement where condition is true
  - – Most of the times it isn't
    - if-then-else statement, only one of the paths is useful
- 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 scalar pipeline = 2
    - Mis-prediction rate is <50%, and often <20%
    - Overall mis-prediction penalty <1 and often <0.4
  - So when is predication ever 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)?
    - Use when mis-predictions >10% and insn overhead <6
  - In a 4-way out-of-order superscalar (penalty ~ 150)
    - potentially useful in more situations
  - typically only desirable for branches that mis-predict frequently
- Other predication advantages
  - Low-power: eliminates the need for a large branch predictor
  - Real-time: predicated code performs consistently
- Predication disadvantages
  - wasted time/energy compared to correct prediction
  - doesn't nest well

Summary



- Control hazards
  - Branch target prediction
  - Branch direction prediction