National Yang Ming Chiao Tung University
Computer Architecture & System Lab

Lecture 9: Branch Predictor
CS10014 Computer Organization

Tsung Tai Yeh
Department of Computer Science
National Yang Ming Chiao University

Acknowledgements and Disclaimer

- Slides were developed in the reference with
  - CS 61C at UC Berkeley
    - https://inst.eecs.berkeley.edu/~cs61c/sp23/
  - CIS510 at Upenn
    - https://www.cis.upenn.edu/~cis5710/spring2019/
  - IF3 at the University of Edinburgh
    - https://www.inf.ed.ac.uk/teaching/courses/car/Notes/2017-18/lecture05-handling_hazards.pdf

Outline

- Branch Predictor
- Branch Target Buffer
- Bimodal Branch Predictor
- Gshare History-Based Predictor
- Tournament Predictor
- Prediction

Static Branch Prediction

- Compiler determine whether branch is likely to be taken or likely to be not taken
  - When is a branch likely to be taken?
  - When is a branch likely to be NOT taken?

```
int gtz=0;
int i = 0;

while (i < 100) {
  x = a[i];
  if (x == 0)
    continue;
  gtz++;
}
```

Static Branch Prediction

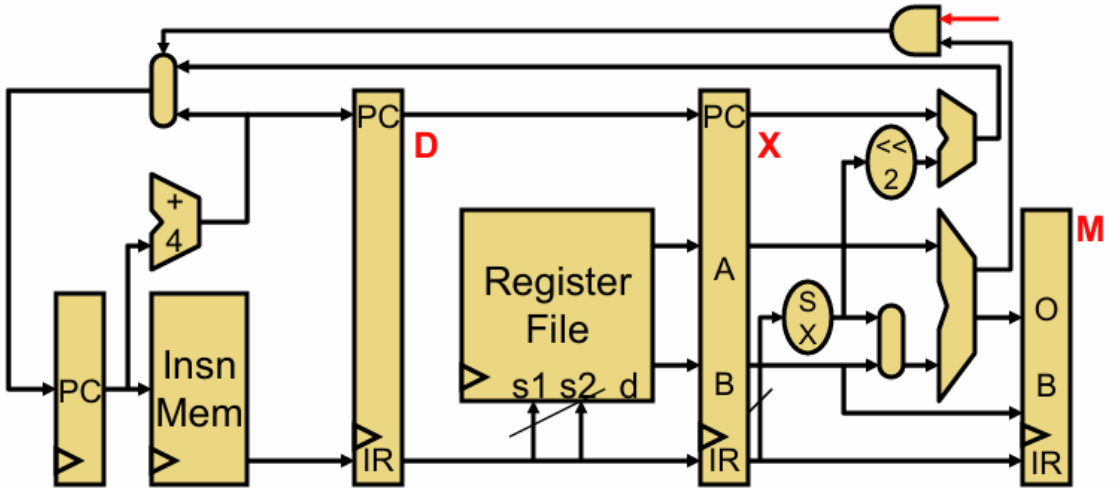
- Compiler determine whether branch is likely to be taken or likely to be not taken
- Decision is based on analysis or profile information
  - 90% of backward-going branches are taken
  - 50% of forward-going branches are no taken
- Decision is encoded in the branch instructions themselves
  - Uses 1 bit: 0=> not likely to branch, 1=> likely to branch
- Prediction may be wrong
  - Must kill instructions in the pipeline when a bad decision is made

Branch Prediction

- Idea: Predict the next fetch address (to be used in the next cycle)
- Requires three things to be predicted at fetch stage:
  - Whether the fetched instruction is a branch
  - (Conditional) branch direction
  - Branch target address (if taken)
- Observation: Target address remains the same for a conditional direct branch across dynamic instances
  - Idea: Store the target address from previous instance and access it with the PC
  - Called Branch Target Buffer (BTB) or Branch Target Address Cache

Branch Prediction

- **Branch speculation**
  - Could just stall to wait for branch outcome (two-cycle penalty)
  - Fetch past branch instructions before branch outcome is known
    - Default: assume "not-taken" (at fetch stage, can't tell it's a branch)



Branch Prediction

- **Speculative execution**
  - Execute before all parameters known with certainty
  - If speculation is **“correct”**
    - + Avoid stall, improve performance
  - If speculation is **“incorrect”** (mis-speculation)
    - Must abort/flush/squash incorrect instructions
    - Must undo incorrect changes (recover pre-speculation state)

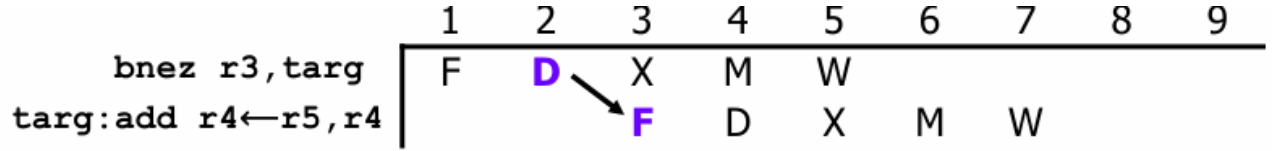
Branch Prediction

- **Control speculation mechanics**
  - Guess branch target, start fetching at guessed position
    - Doing nothing is implicitly guessing target is PC+4
    - Can actively guess other targets: dynamic branch prediction
  - Execute branch to verify guess
    - Correct speculation? Keep going
    - Mis-speculation? Flush mis-speculated instructions
      - Hopefully haven't modified permanent state (Regfile, DMem)

      + Happens naturally in in-order 5-stage pipeline

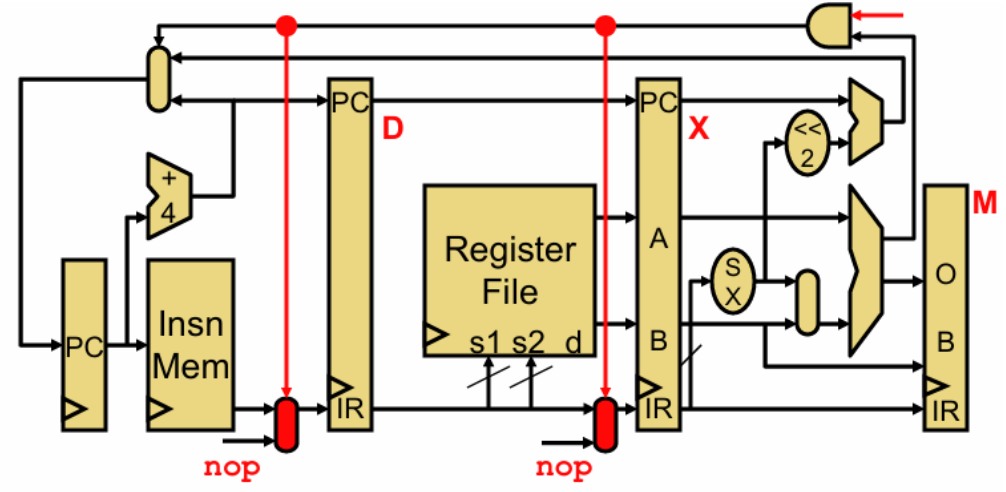
Branch Prediction

- **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
  - Option #2: During fetch?
    - How do we do that?
      - Branch predictor



Branch Recovery

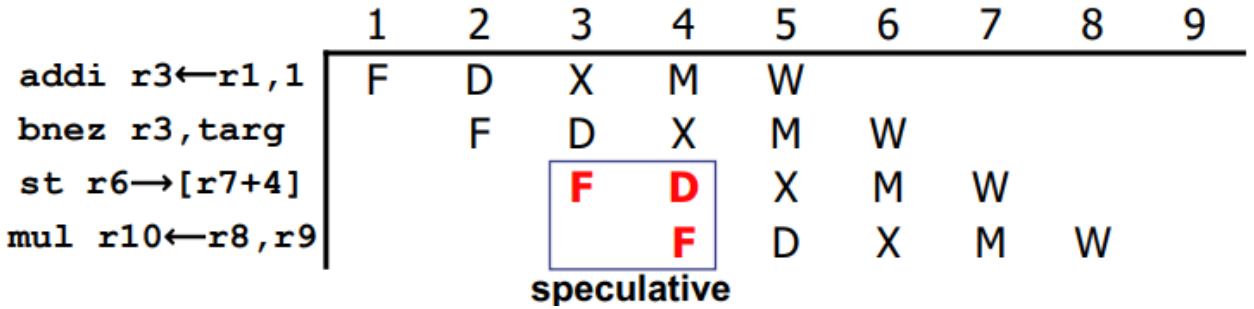
- **Branch recovery**
  - What to do when branch is actually taken?
    - Instruction that are in F and D are wrong
    - Flush them, i.e., replace them with nops



Branch Recovery

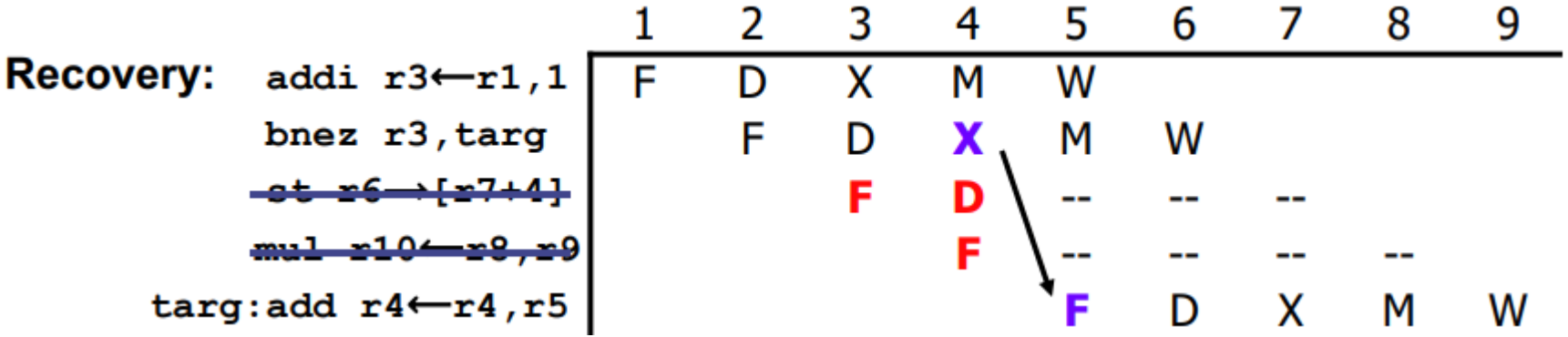
- **Branch recovery**
  - What to do when branch is actually taken
    - Flush them, i.e., replace them with nops
    - + They haven't written permanent state yet (regfile, DMem)
    - - Two cycle penalty for taken branches

Correct:



Branch Recovery

- **Mis-speculation recovery**
  - What to do on wrong guess
    - Branch resolves in **X (EXEC.) stage**
    - + Younger insts (in F, D) haven't changed permanent state
    - Flush instructions currently in D and X (i.e, replace with nops)



Takeaway Questions

- Assume that
  - **Branch: 20%,** load: 20%, store: 10%, other: 50%
  - Say, 75% of branches are taken
  - What is the CPI?

Takeaway Questions

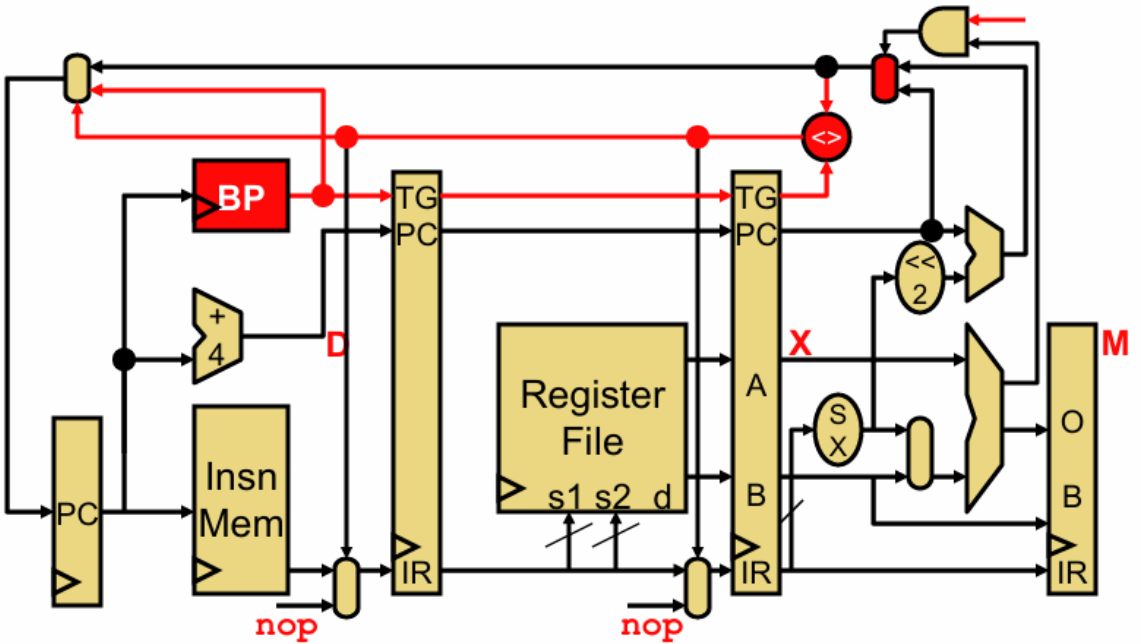
- Assume that
  - **Branch: 20%,** load: 20%, store: 10%, other: 50%
  - Say, 75% of branches are taken
  - What is the CPI?
    - CPI = 1 + 20% * 75% *2 = 1.3
    - Branches cause 30% slowdown
      - Worse with deeper pipelines, why?
      - Can we do better than assuming branch is not taken?

Dynamic Branch Prediction

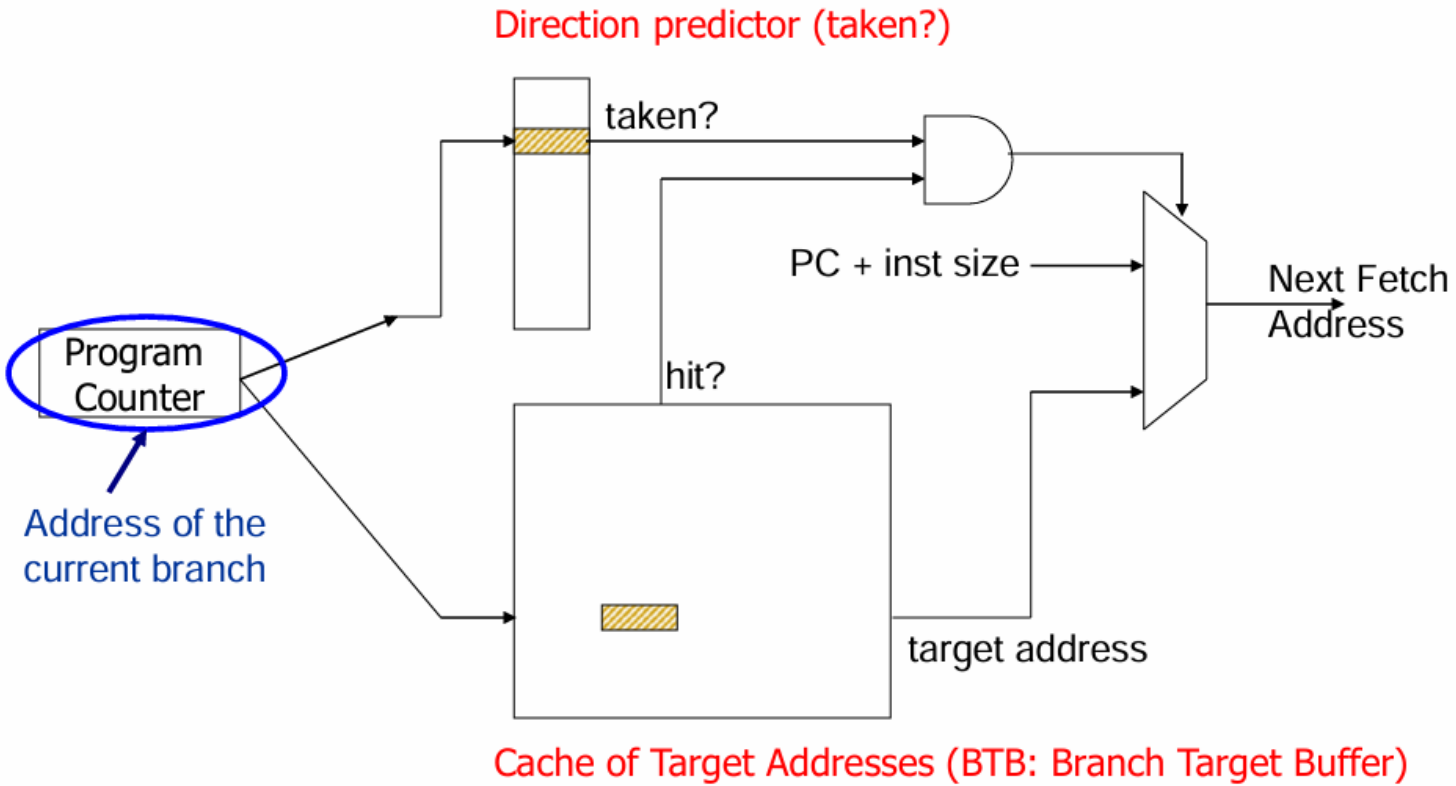
- Monitor branch behavior and learn
  - Key assumption: past behavior indicative of future behavior
- Predict the present branch using learning history
  - Identify individual branches by their PC or dynamic branch history
- Predict
  - Outcome: taken or not taken
  - Target: address of instruction to branch to
  - Check actual outcome and update the history
  - Squash incorrectly fetched instructions

Dynamic Branch Prediction

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

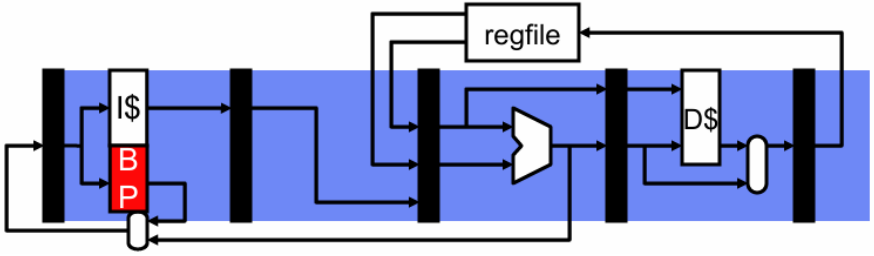


Dynamic Branch Prediction



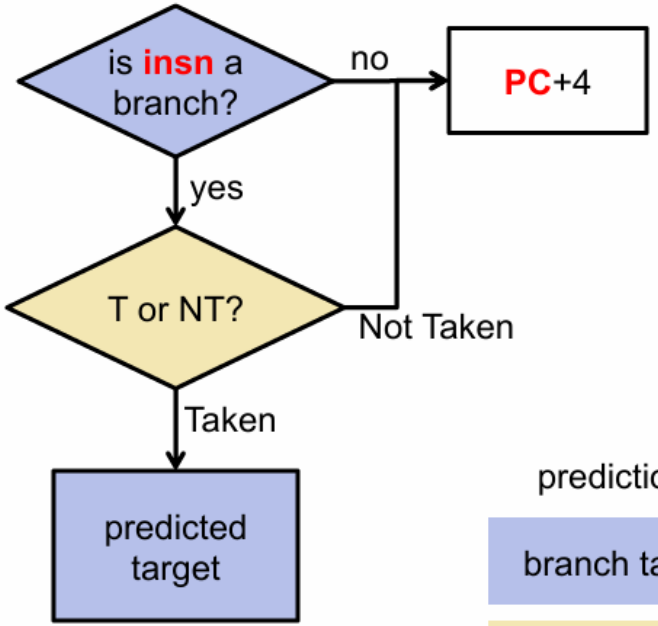
Dynamic Branch Prediction

- **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 branch only)
    - Predict taken/not-taken
  - Step #3: if the branch is taken, where does it go?



Dynamic Branch Prediction

- **Branch Prediction steps**



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

prediction source:

branch target buffer

direction predictor

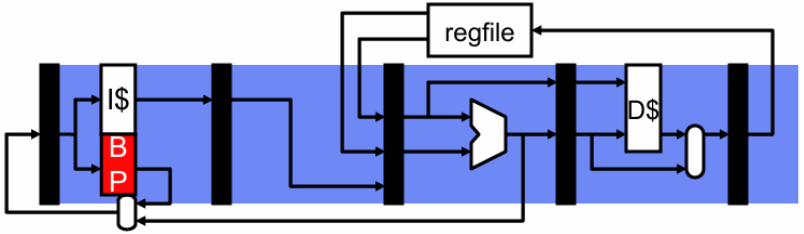
Dynamic Branch Prediction

- **Branch prediction performance**
  - 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

Branch Target Buffer

- Branch predictors tell whether the branch will be taken or not, but they say nothing about the target of the branch
- To resolve a branch early we need to know both the outcome and the target
- Solution: store the likely target of the branch in a table (cache) indexed by the branch PC → BTB
- Usually BTB is accessed in the IF stage and the branch predictor is accessed later in the ID stage

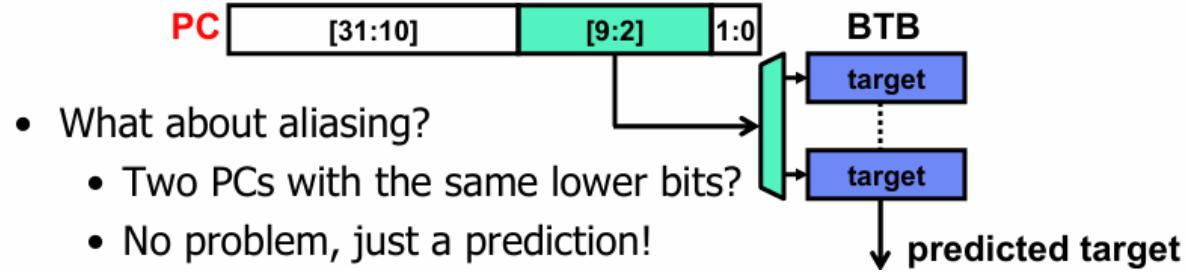
Branch Target Buffer



- **Revisit branch prediction**
  - Step #1: is it a branch?
    - Easy after decode …during fetch: predictor
  - Step #2: is the branch taken or not taken?
    - Direction predictor (applies to conditional branch only)
  - 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

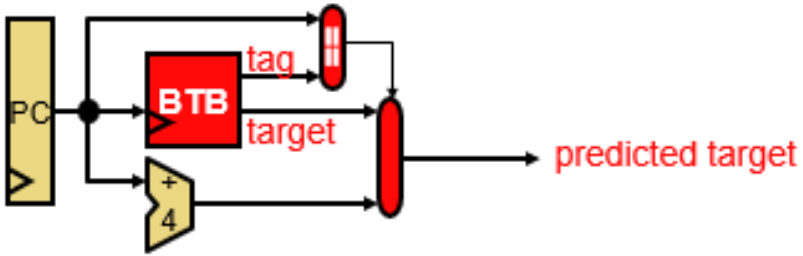
- **Branch Target Buffer (BTB)**
  - Learn from past, predict the future, record the past in a hardware
  - Guess the future PC based on past behavior
  - E.g. “Last time the branch X was taken, it went to address Y”
    - So, in the future, if address X is fetched, fetch address Y next
    - PC indexes table of bits target addresses
    - Branch will go to same place it went last time



Branches	Branch =	SB	BEQ	rs1,rs2,imm
	Branch ≠	SB	BNE	rs1,rs2,imm
	Branch <	SB	BLT	rs1,rs2,imm
	Branch ≥	SB	BGE	rs1,rs2,imm
	Branch < Unsigned	SB	BLTU	rs1,rs2,imm
	Branch ≥ Unsigned	SB	BGEU	rs1,rs2,imm

Branch Target Buffer

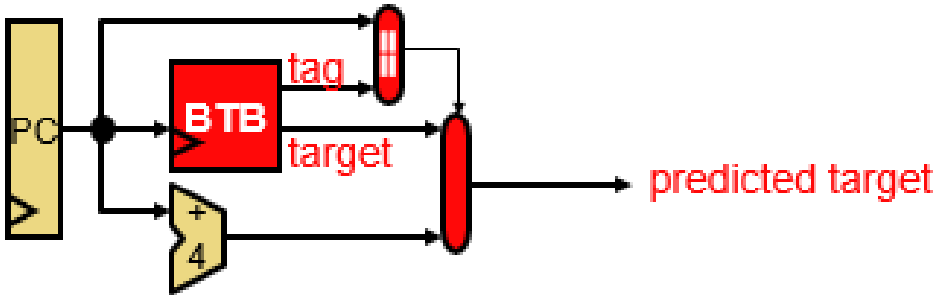
- **Branch Target Buffer (BTB)**
  - At Fetch, how does insn know it's a branch and read BTB?
    - All insns access BTB in parallel with Imm Fetch
  - BTB is used to predict which insn are branches
    - Implement by "tagging" each entry with its corresponding PC
    - Update BTB on every taken branch insn, record target PC
      - BTB[PC].tag = PC, BTB[PC].target = target of branch



Branches	Branch =	SB	BEQ	rs1,rs2,imm
	Branch ≠	SB	BNE	rs1,rs2,imm
	Branch <	SB	BLT	rs1,rs2,imm
	Branch ≥	SB	BGE	rs1,rs2,imm
	Branch < Unsigned	SB	BLTU	rs1,rs2,imm
	Branch ≥ Unsigned	SB	BGEU	rs1,rs2,imm

Branch Target Buffer

- **Branch Target Buffer (BTB)**
  - All insns access at Fetch in parallel with Imm
    - Check for tag match, signifies insn at that PC is a branch
    - Predict PC = (BTB[PC].tag == PC) ? BTB[PC].target: PC + 4

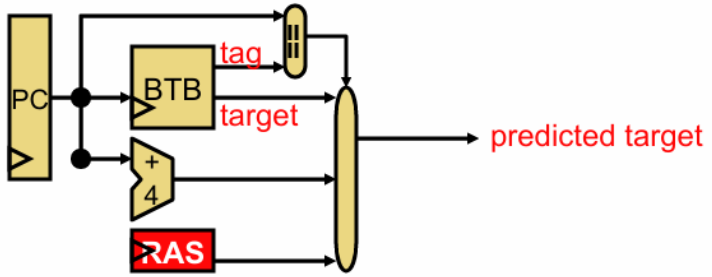


Branch Target Buffer

- **Why does a BTB work?**
  - Because most control instructions 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
      - Dynamic linked functions (DLLs): target always the same
      - Dynamic dispatched (virtual) functions: hard but uncommon
    - Two indirect unconditional jump idioms
      - Switches: hard but uncommon
      - Function returns: hard and common but …

Branch Target Buffer

- **Return Address Stack (RAS)**
  - Call instruction? RAS[TopOfStack++] = PC+4
  - Return instruction? Predicted-target = RAS[--TopOfStack]
  - How can we tell if an insn is a call/return before decoding it?
    - Access RAS on every insn BTB-style doesn't work
    - Another predictor (or put them in BTB marked as "return")
    - Or, pre-decode bits in insn mem, written when first executed

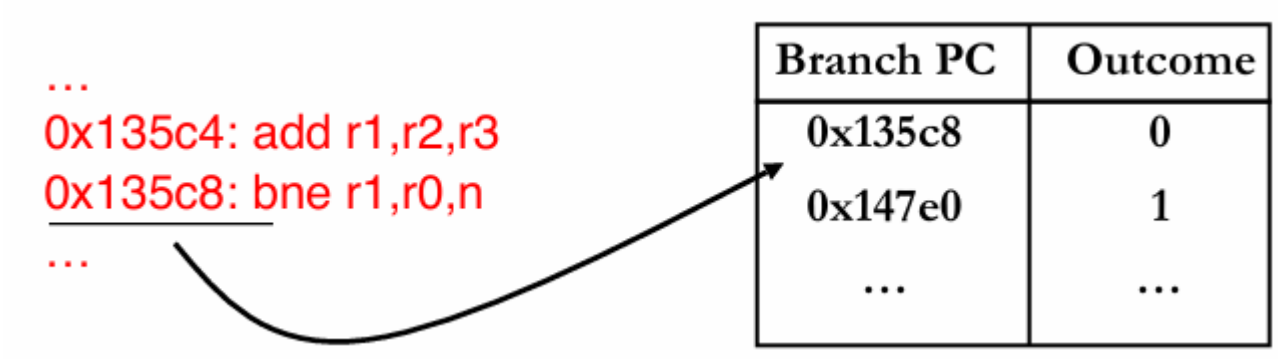


Bimodal Branch Predictor

- **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? Look back edges, checking for uncommon conditions

1-bit Branch Predictor

- 1 bit indicating Taken (1) or Not Taken (0)
- Branch prediction buffers
  - Match branch PC during IF or ID stage



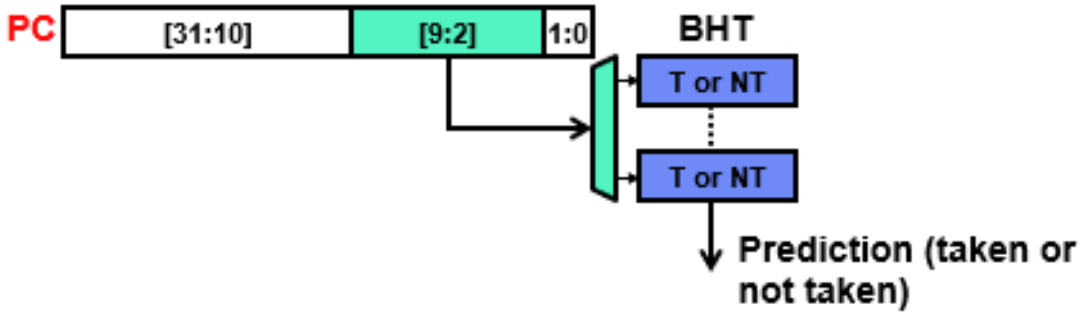
1-bit Branch Predictor

- 1 bit indicating Taken (1) or Not Taken (0)
- Incur at least 2 mis-predictions per loop
  - Problem: “unstable” behavior

```
while (i < 100) {
  x = a[i];
  if (x == 0)
    continue;
  gtz++;
}
```

1-bit Branch Predictor

- **1-bit 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!

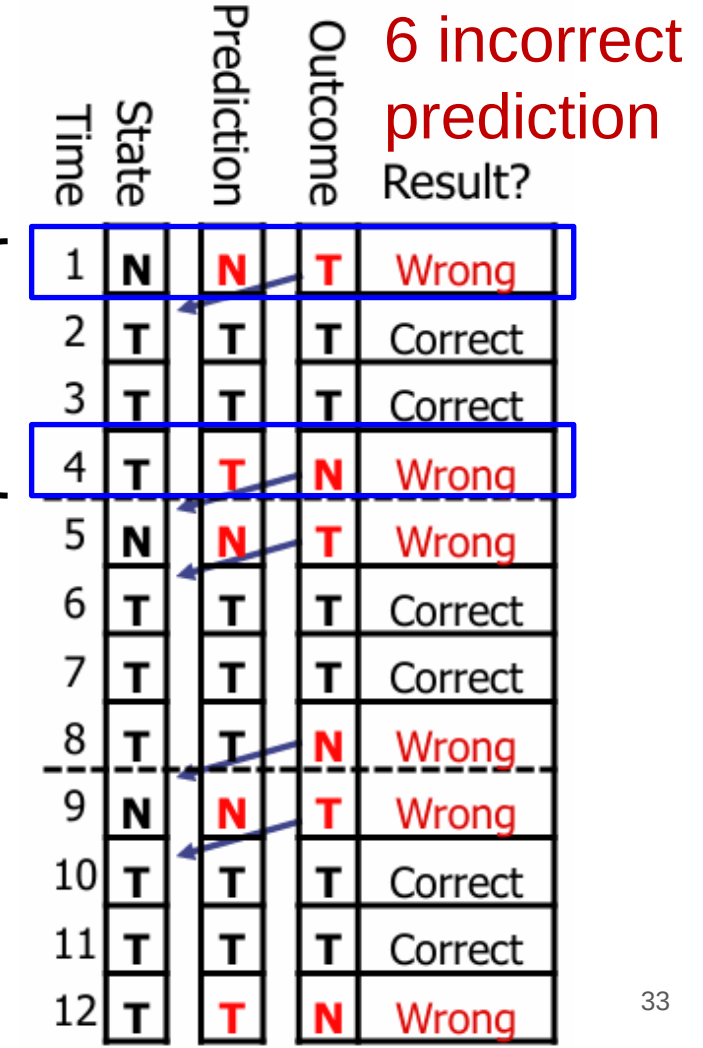


1-bit Branch Predictor

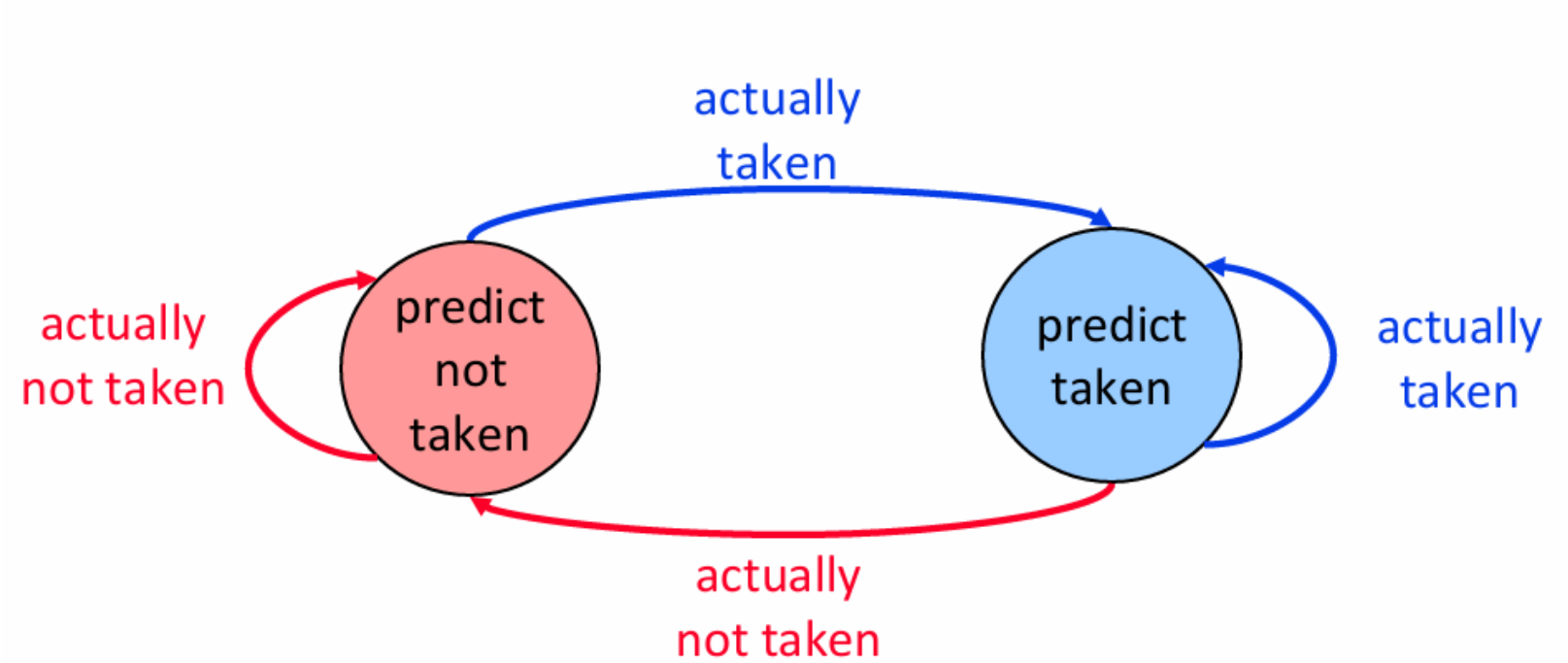
- **1-bit direction predictor**
  - Problem: inner loop branch below

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

  - The outcome is based on previous correct prediction
  - Two “built-in” mis-predictions per inner loop iteration
  - Branch predictor “changes its mind too quickly”



1-bit Branch Predictor FSM

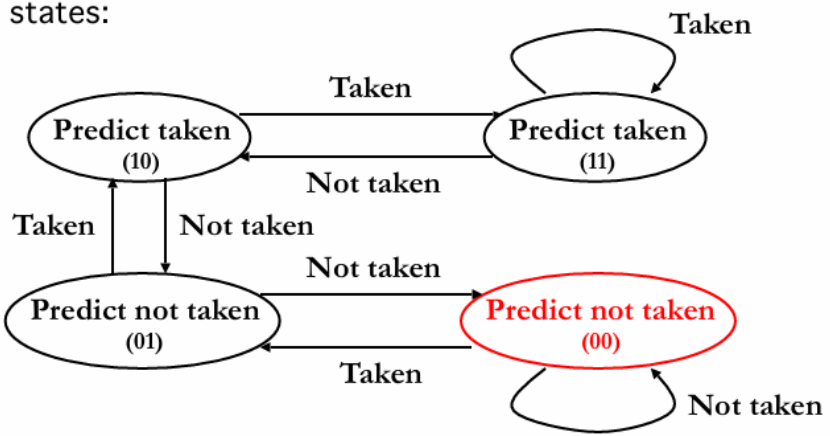


1-bit Branch Predictor

- 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.

2-bit (Bimodal) Branch Predictor

- Predictor states:



- Learns biased branches
- N-bit predictor
  - Increment on Taken outcome and decrement on Not Taken outcome
  - If counter > $(2^{n-1}) / 2$ then taken, otherwise do not take
  - Takes longer to learn, but sticks longer to the prediction

2-bit (Bimodal) Branch Predictor

- Nested loop:

```
Loop1: …
       …
Loop2: …
       bne r1,r0,loop2
       …
       bne r2,r0,loop1
```

- 1st outer loop execution:
  - 00 → predict not taken; actually taken → update to 01 (misprediction)
  - 01 → predict not taken; actually taken → update to 10 (misprediction)
  - 10 → predict taken; actually taken → update to 11
  - 11 → predict taken; actually taken
  - …
  - 11 → predict taken; actually not taken → update to 10 (misprediction)

2-bit (Bimodal) Branch Predictor

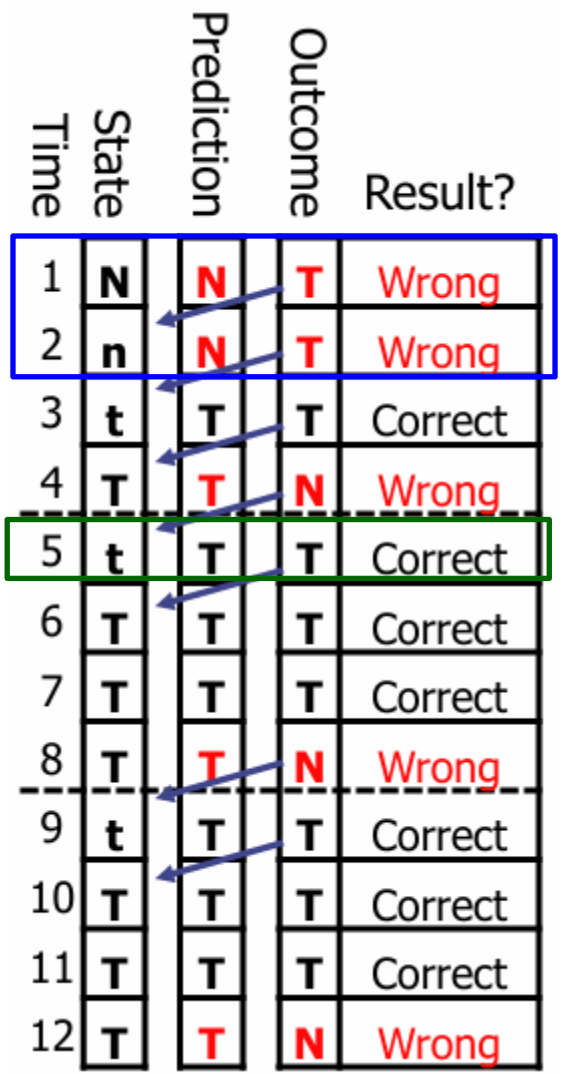
- 2nd outer loop execution onwards:
  - 10 → predict taken; actually taken → update to 11
  - 11 → predict taken; actually taken
  - ...
  - 11 → predict taken; actually not taken → update to 10 (misprediction)
- In practice misprediction rates for 2-bit predictors with 4096 entries in the buffer range from 1% to 18% (higher for integer applications than for fp applications)

- Bottom-line: 2-bit branch predictors work very well for loop-intensive applications
  - n-bit predictors (n>2) are not much better
  - Larger buffer sizes do not perform much better

2-bit (Bimodal) Branch Predictor

- **Two-bit saturating counters (2bc)**
  - 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 mis-predict each loop execution (rather than two)
      - + Fixes this pathology (which is not contrived)

5 incorrect prediction



2-bit (Bimodal) Branch Predictor

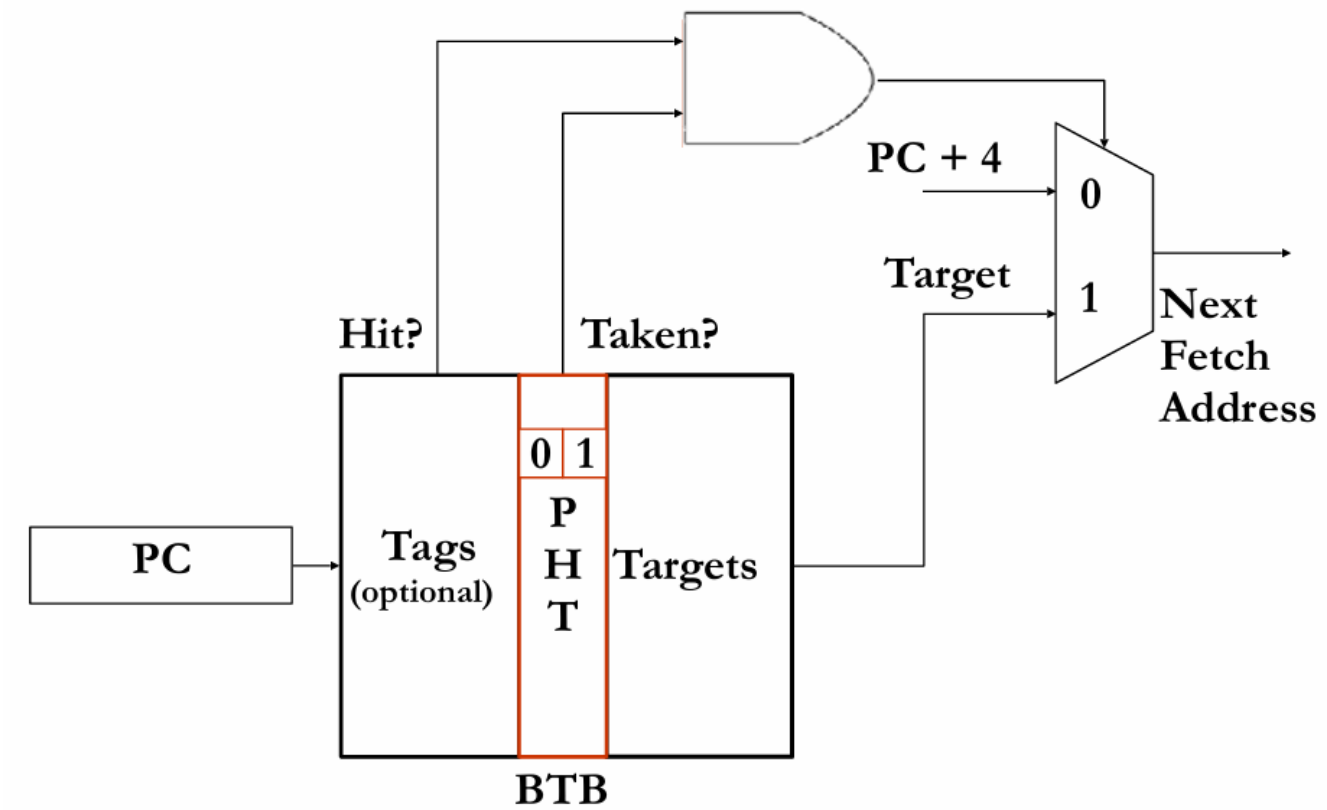
- Each branch associated with a two-bit counter
- One more bit provides hysteresis
- A strong prediction does not change with one single different outcome

- 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)

2-bit (Bimodal) Branch Predictor

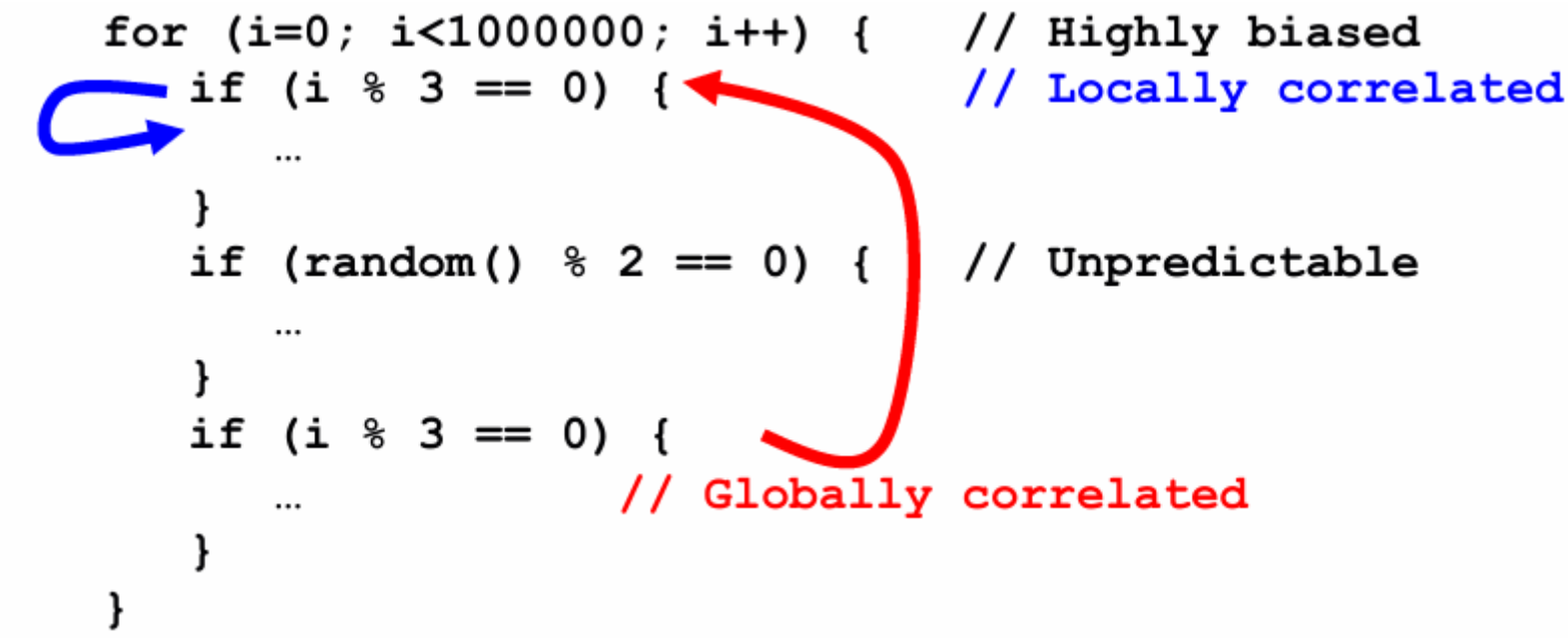


Correlating Predictors

- 1- and 2-bit predictors exploit most recent history of the current branch
- Realization: branches are correlated!
  - Local:
    - A branch outcome maybe correlated with past outcomes of the same branch
  - Global:
    - A branch outcome maybe correlated with past outcomes of other branches

Correlating Predictors

- **Branches may be correlated**
  - Consider:



Correlating Predictors (Local)

- 1- and 2-bit predictors exploit most recent history of the current branch
- Realization: outcomes of same branch correlated!
  - Branch outcomes:
    - 1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 0 …
  - Idea: exploit recent history of same branch in prediction

```
while (i < 4) {
  x = a[i];
  if (x == 0)
    continue;
  gtz++;
}
```

Correlating Predictors (Global)

- 1- and 2-bit predictors exploit most recent history of the current branch
- Realization: Different branches maybe correlated!

```
if (a == 2)
    a = 0;
if (b == 2)
    b = 0;
if (a != b) {
    …}
```

If both branches are taken,
the last branch definitely not taken

```
char s1 = "Bob"
...
if (s1 != NULL)
  reverse_str(s1);
      ____________
reverse_str(char *s) {
  if (s1 == NULL)
    return;
 ...
```

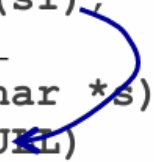
s1 definitely not Null
in this calling context

Correlating Predictors (Global)

- 1- and 2-bit predictors exploit most recent history of the current branch
- Realization: Different branches maybe correlated!

```
if (a == 2)
    a = 0;
if (b == 2)
    b = 0;
if (a != b) {
    …}
```

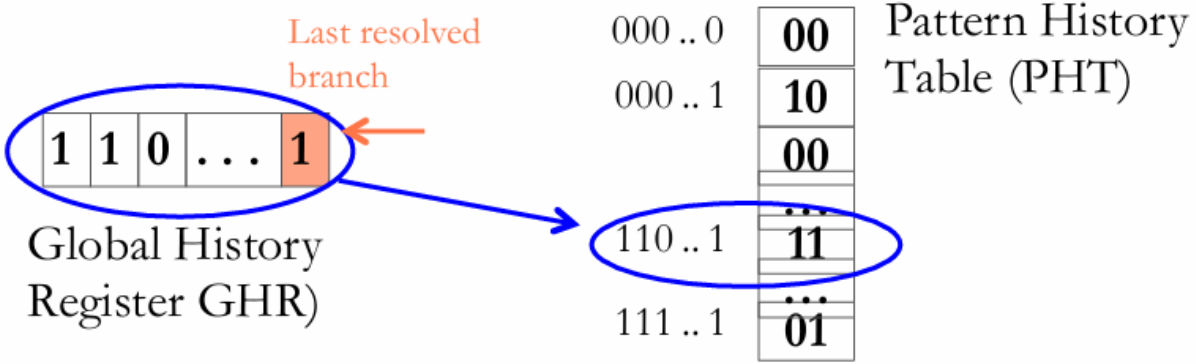
```
char s1 = "Bob"
...
if (s1 != NULL)
  reverse_str(s1);
    ___________
reverse_str(char *s) {
  if (s1 == NULL)
    return;
  ...
```



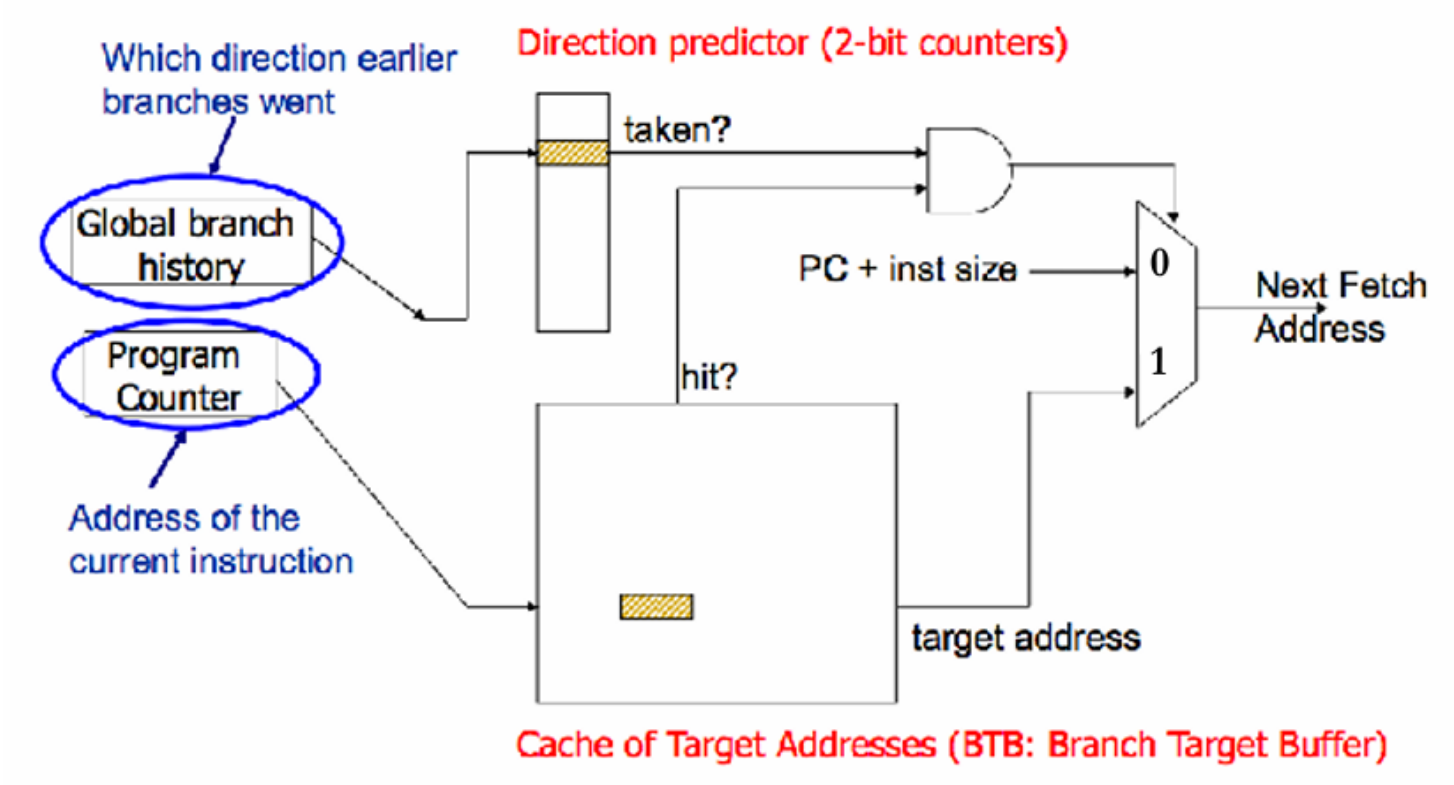
Idea: exploit recent history of other branches in prediction

Global Two-level (Correlating) Predictor

- Prediction depends on the context of the branch
- Context: history (T/NT) of the last N branches
  - First level of the predictor
  - Implemented as a shift register
- Prediction: 2-bit saturating counters
  - Indexed with the "global" history
  - Second level of the predictor

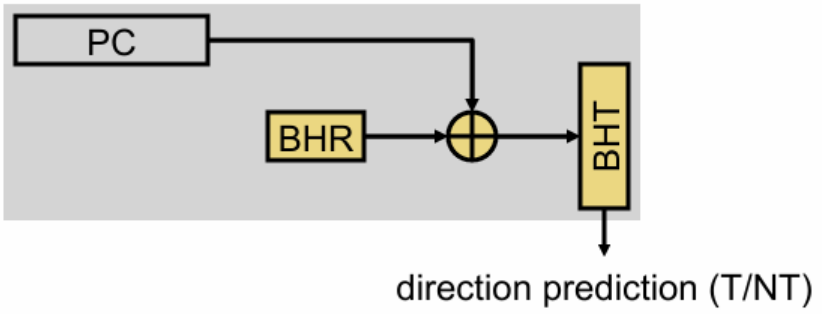


Global Two-level (Correlating) Predictor



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



Gshare History-Based Predictor

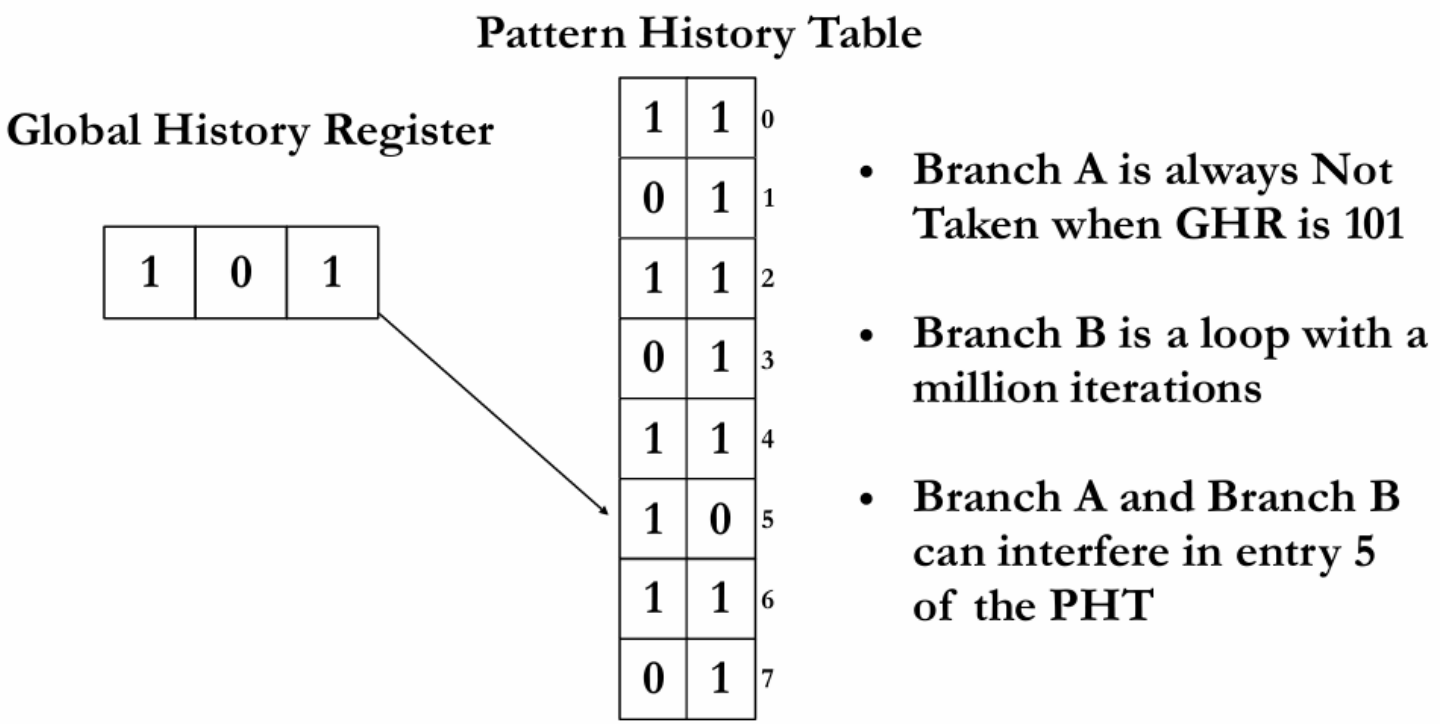
- Gshare working example
  - Assume program has one branch
  - BHT: one 1-bit DRIP entry
  - 3BHR: last 3 branch outcomes
  - Train counter, and update BHR after each branch

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

Bimodal and Global 2-level

- Bimodal (2-bit) branch predictor
  - + Good for biased branches
  - + No interference
  - - Cannot discern patterns
- Global 2-level Branch Predictor
  - + Leverages correlated branches
  - + Identifies patterns
  - - Cannot always take advantage of biased branches
  - - Interference

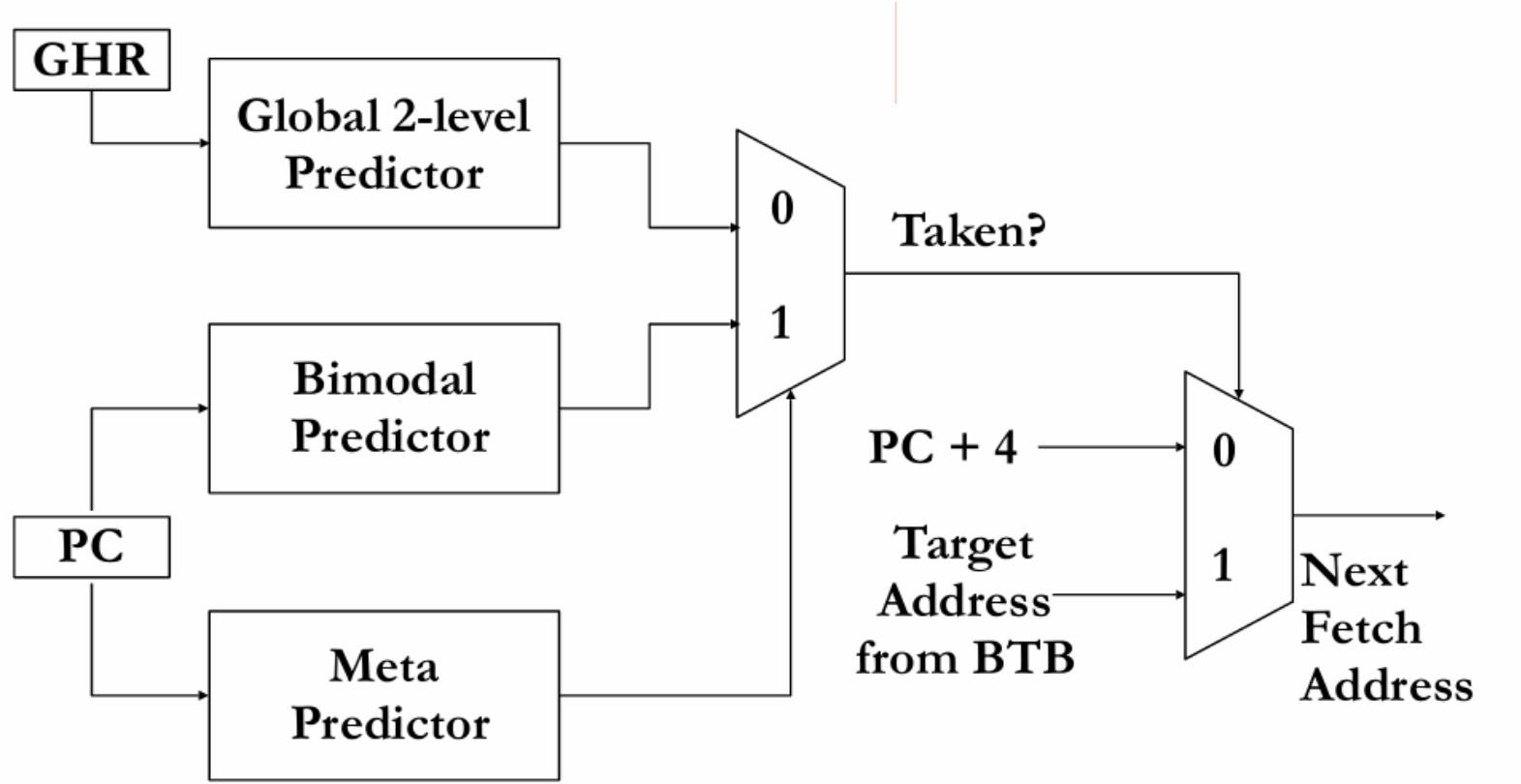
Interference in Global 2-level Predictor



Tournament Predictor

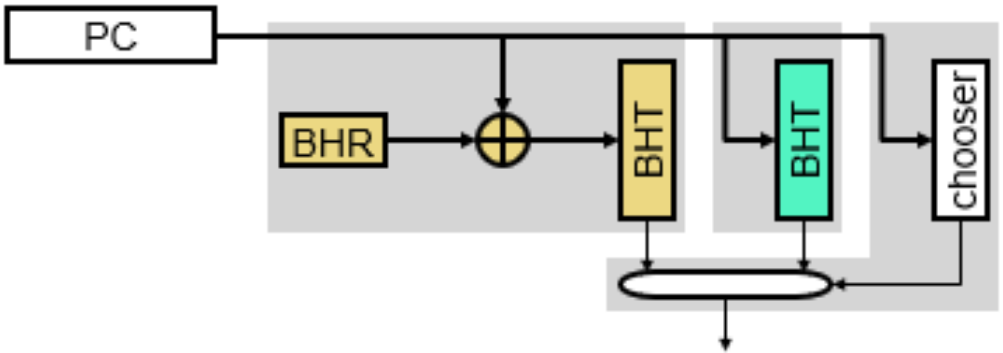
- Most branches are biased (e.g. 99% Taken)
- Filter the biased branches with a simple predictor (e.g. Bimodal)
- Predict the hard branches with the Global 2-level predictor
- Use a meta-predictor to chose a different predictor
- The meta-predictor is a PHT of 2-bit saturating counters

Tournament Predictor



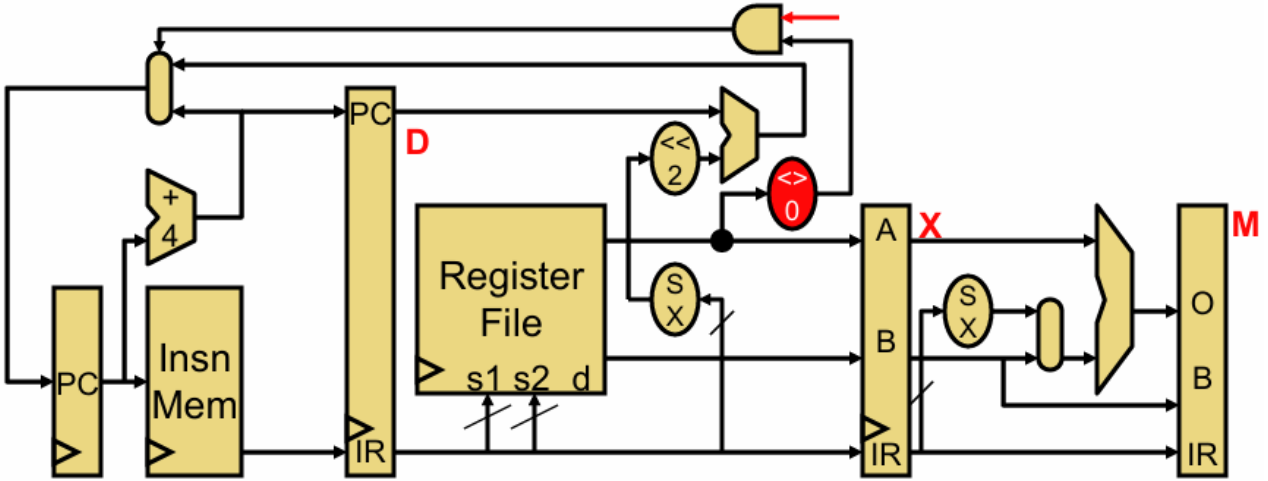
Tournament Predictor

- Hybrid (tournament) predictor
  - + Correlated predictor can be made smaller, handles fewer branches
  - + 90 – 95% accuracy



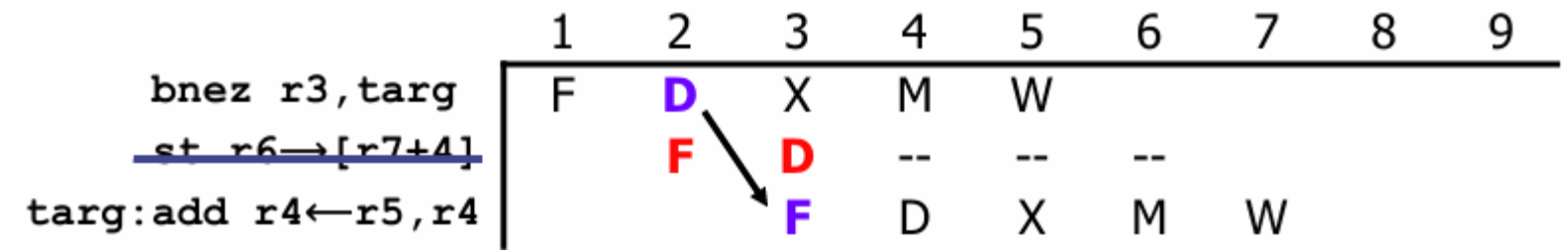
Reduce Branch Penalty

- **Fast branch: can decide at Decode, not Exec. Stage**
  - Test must be comparison to zero or equality
  - + New taken branch penalty is 1
  - - Additional instructions (slt) for more complex tests, must bypass to Decode



Reduce Branch Penalty

- **Fast branch: targets control-hazard penalty**
  - Basically, branch instructions that can resolve at D, not X
  - - Must bypass into decode stage



Reduce Branch Penalty

- **Fast branch performance**
  - Assume: Branch: 20%, 75% of branches are taken
    - CPI = 1 + 20% * 75% *1 = 1.15
    - 15% slowdown (better than the 30% from before)
  - Fast branches assume only simple comparisons
    - Not fine for ISAs with “branch if $1 > $2” operations
    - In such cases, say 25% of branches require an extra instruction
      - CPI = 1 + (20% * 75% * 1) + 20% * 25% *1 (extra insn) = 1.2
  - Another options
    - Delayed branch or branch delay slot

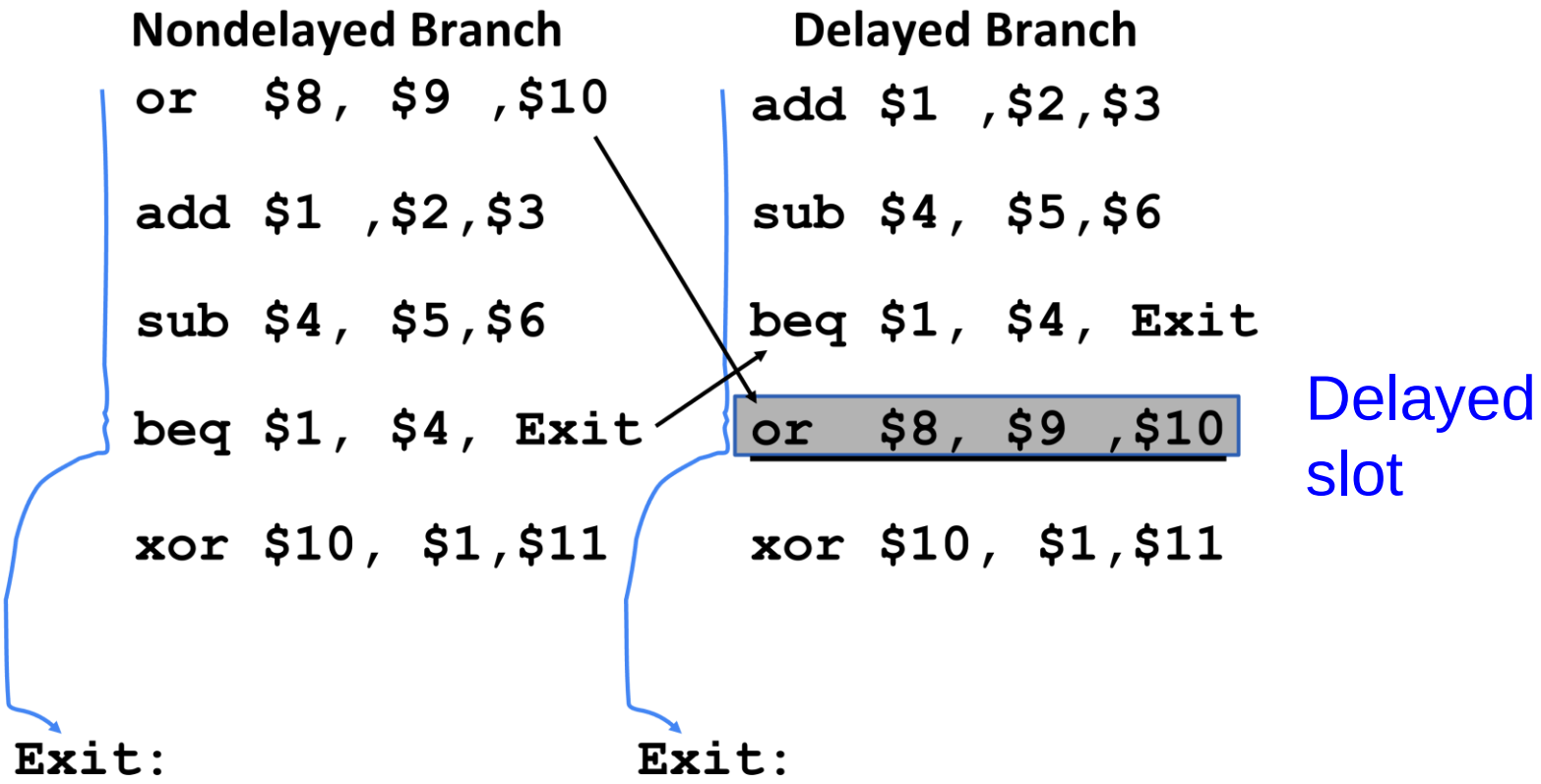
Delayed Branch Slot

- **Delayed Branch Slot**
  - **Old definition:**
    - if we take the branch, none of the instructions after the branch get execute by accident
  - **New definition:**
    - Whether or not we take the branch, the single instruction immediately following the branch gets executed (called the branch-delay slot)

Delayed Branch Slot

- **Delayed Branch Slot**
  - We always execution instruction after branch
  - Worst-case:
    - Can always put a no-op in the branch-delay slot
  - Better case:
    - Can find an instruction before the branch which can be placed in the branch-delay slot without affecting flow of the program
    - The compiler must be smart to find instructions to do this

Delayed Branch Slot



Takeaway Questions

- Dynamic branch prediction
  - 20% of instruction branches
  - Simple predictor: branches predicted with 75% accuracy
    - What is the CPI when using such a simple predictor?
  - More advanced predictor: 95% accuracy
    - What is the CPI when using such an advanced predictor?

Takeaway Questions

- Dynamic branch prediction
  - 20% of instruction branches
  - Simple predictor: branches predicted with 75% accuracy
    - What is the CPI when using such a simple predictor?
    - CPI = 1 + (20% * 25% * 2) = 1.1
  - More advanced predictor: 95% accuracy
    - What is the CPI when using such an advanced predictor?
    - CPI = 1 + (20% * 5% * 2) = 1.02

Predication

- Complex predicates are converted into multiple branches
  - If ((a == b) && (c < d) && (a > 5000)) {…}
  - 3 conditional branches
- Problem:
  - This increases the number of control dependencies
- Idea
  - Combine predicate operations to feed a single branch instruction
  - Predicates stored and operated on using condition register
  - A single branch checks the value of the combined predicate
  - + Fewer branches in code -> fewer mispredictions/stalls

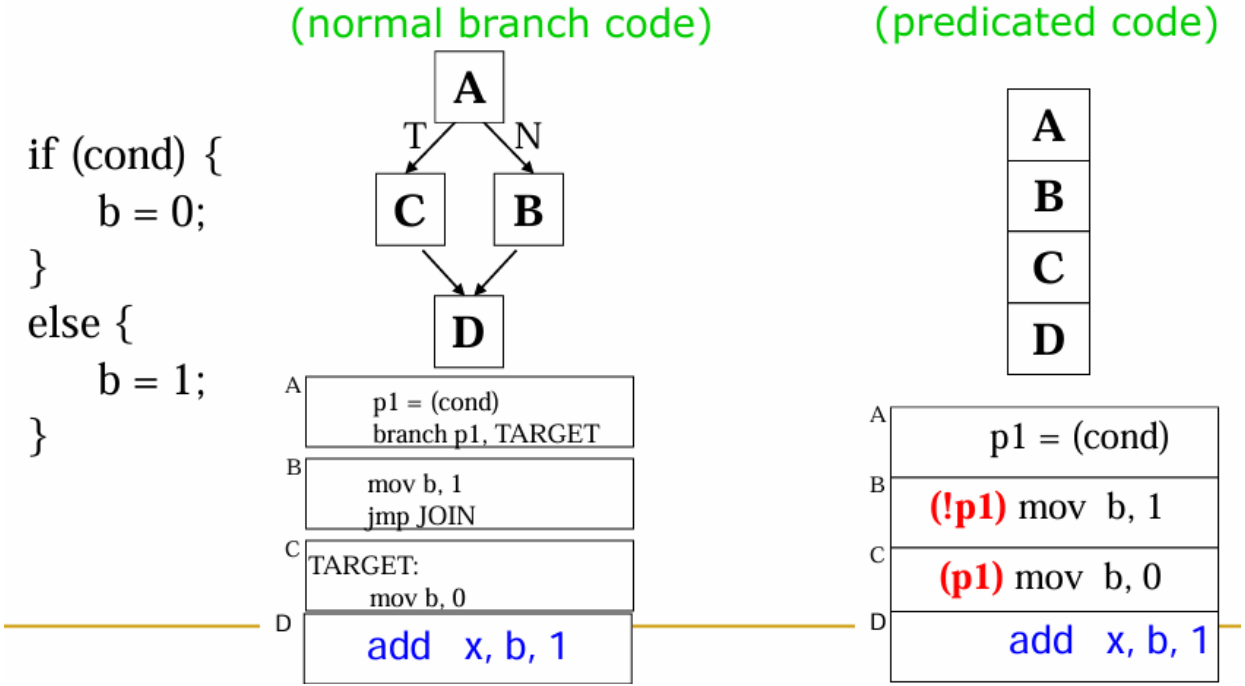
Predication

- Idea: compiler converts control dependence into data dependence -> branch is eliminated
  - Each instruction has a predicate bit set based on the predicate computation
  - Only instructions with TRUE predicates are committed (Other turned into NOPs)
  - - Possible unnecessary work
    - If the first predicate is false, no need to compute other predicates

Predication

- Idea: compiler converts control dependence into data dependence -> branch is eliminated

https://course.ece.cmu.edu/~ece740/f13/lib/exe/fetch.php?media=onur-740-fall13-module7.4.2-predicated-execution.pdf



Predication

- Instead of predicting which way we're going, why not go both ways?

```
// C code          ; original RV      ; imaginary predicated RV
if (a >= b) {      blt x1,x2,else     slt p1,x1,x2
  x += y;          add x3,x3,x4       add.!p1 x3,x3,x4
} else {           j after            sub.p1 x3,x3,x5
  x -= z;          else:
}                  sub x3,x3,x5
                   after:
```

Predication

- Predication overhead is additional instructions
  - 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
- For a given branch, predicate
  - Average number of additional insns < overall mis-predication penalty

Predication

- Advantages
  - + Eliminates mispredications for hard-to-predict branches
  - + No need for branch prediction for some branches
  - + Good if misprediction cost > useless work due to predictation
  - + Enable code optimizations hindered by the control dependency
  - + Can move instructions more freely within predictated code

Predication

- Disadvantages
  - - Cause useless work for branches that are easy to predict
  - - Reduce performance if misprediction cost < useless work
  - - Additional hardware and ISA support
  - - Cannot eliminate all hard to predict branches
    - Loop branches?

Conclusion

- Pipeline challenge is hazards
  - Forwarding helps with many data hazards
  - Delayed branch helps with control hazard in 5 stage pipeline
  - Load delay slot / interlock necessary
- More aggressive performance
  - Superscalar
  - Out-of-order execution