

## Conditional Branches

- A conditional branch instruction introduces the hazard caused by the dependency of the branch condition on the result of a preceding instruction.
- The decision to branch cannot be made until the execution of that instruction has been completed.
- Branch instructions represent about 20% of the dynamic instruction count of most programs.

# Branch Prediction

- Static Branch Prediction
- Dynamic Branch Prediction

# Static Branch Prediction

- Compiler determines whether branch is likely to be taken or likely to be not taken.
  - How?

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 determines 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 not taken
  - BTFN: “backwards taken, forwards not-taken”
  - Used in ARC 600 and ARM 11
- 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
  - Speculatively issued instructions must not change processor state

# Dynamic Branch Prediction

- Monitor branch behavior and learn
  - Key assumption: past behavior indicative of future behavior
- Predict the present (current branch) using learned 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

# Simplest dynamic predictor: 1-bit Prediction

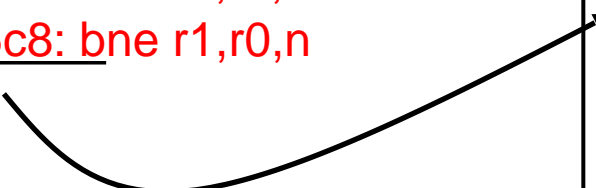
- 1 bit indicating Taken (1) or Not Taken (0)
- Branch prediction buffers:
  - Match branch PC during IF or ID stages

...

0x135c4: add r1,r2,r3

0x135c8: bne r1,r0,n

...



Branch PC	Outcome
0x135c8	0
0x147e0	1
...	...

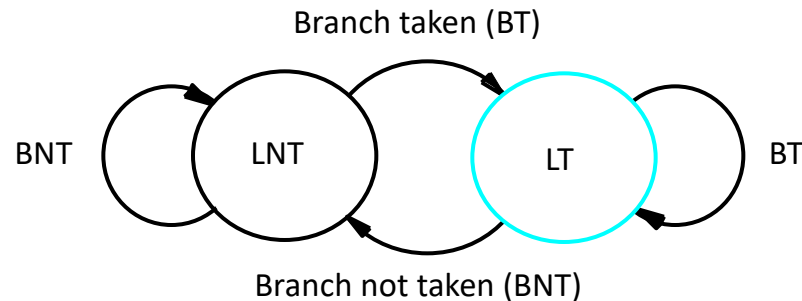
- Incurs at least 2 mis-predictions per loop

**Problem: “unstable” behavior**

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

# Dynamic Branch Prediction

- The processor hardware assesses the likelihood of a given branch being taken by keeping track of branch decisions every time that instruction is executed.
- In its simplest form, the execution history used in predicting the outcome of a given branch instruction is the result of the most recent execution of that instruction. The processor assumes that the next time the instruction is executed, the result is likely to be the same – two-state machine.



(a) A 2-state algorithm

## 2-bit (Bimodal) Branch Prediction

- Idea: add **hysteresis**
  - Prevent spurious events from affecting the most likely branch outcome
- 2-bit saturating counter:
  - 00: do not take
  - 01: do not take
  - 10: take
  - 11: take

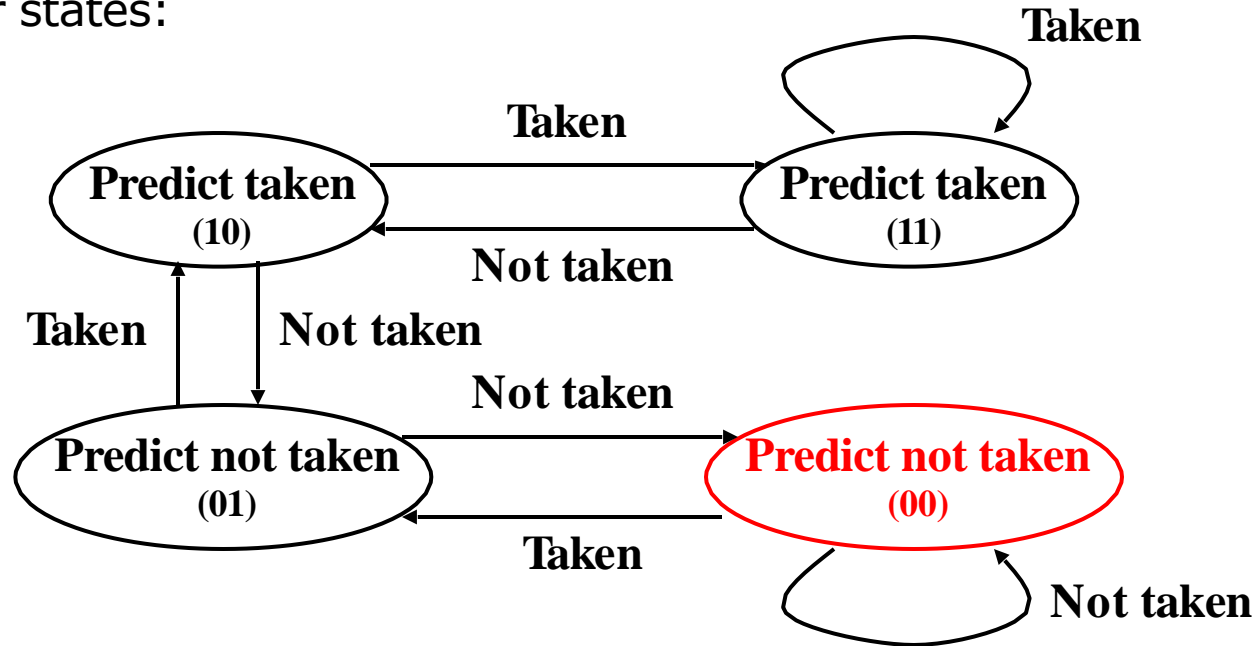
Branch PC	Outcome
0x135c8	10
0x147e0	11
...	...

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



# 2-bit (Bimodal) Branch Prediction

- Predictor states:

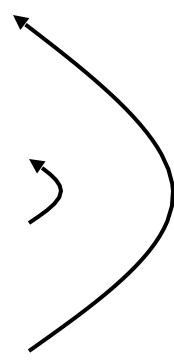


- Learns biased branches
- N-bit predictor:
  - Increment on Taken outcome and decrement on Not Taken outcome
  - If  $\text{counter} > (2^n - 1)/2$  then take, otherwise do not take
  - Takes longer to learn, but sticks longer to the prediction

# Example of 2-bit (Bimodal) Branch Prediction

- Nested loop:  

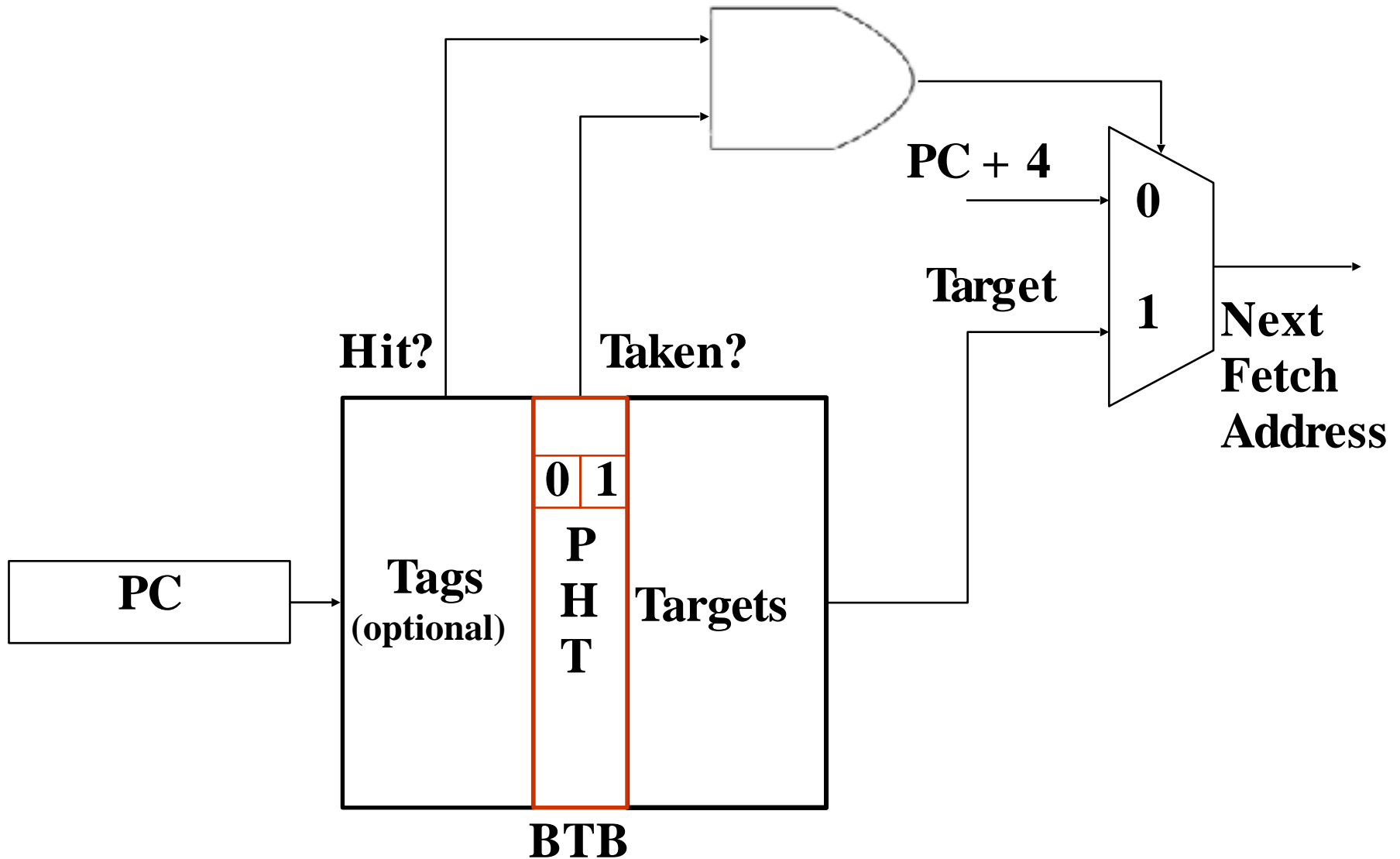
```
Loop1: ...  
    ...  
    Loop2: ...  
        bne r1,r0,loop2  
    ...  
    bne r2,r0,loop1
```

A diagram illustrating nested loops. It consists of two concentric, right-facing curved arrows. The outer arrow represents the first loop (Loop1), and the inner arrow represents the second loop (Loop2).
  - 1<sup>st</sup> 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)

# Example Continued

- 2<sup>nd</sup> 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

# Bimodal (2-bit) predictor logic



- Only if the prediction is incorrect twice in a row will the state change.
- Use static prediction alg with this alg.
- Branch prediction will be correct all the time, except for the final pass through the loop.
- Misprediction in this latter case is unavoidable.

# Correlating Branch Predictors/ two-level predictors

- Branch predictors that use the behavior of other branches to make a prediction are called correlating predictors or two-level predictors.
- The 2-bit predictor schemes use only the recent behavior of a single branch to predict the future behavior of that branch.

## 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 (multiple outcomes or history, not just the most recent) of the same branch
  - Global: A branch outcome maybe correlated with past outcomes of other branches

## Correlating Predictors (Local)

- 1- and 2-bit predictors exploit most recent history of the current branch
- Realization: outcomes of same branch correlated!
- 

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

- Branch outcomes: 1,1,1,0, 1,1,1,0 1,1,1,0....

**Idea: exploit recent history of same branch in prediction**



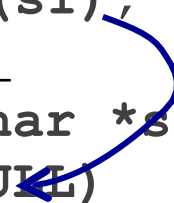
## Correlating Predictors (Global)

- 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;
    ...
```



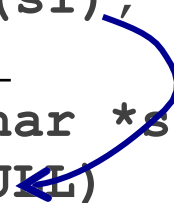
**s1 definitely not Null  
in this calling context**

## 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;
    ...
```



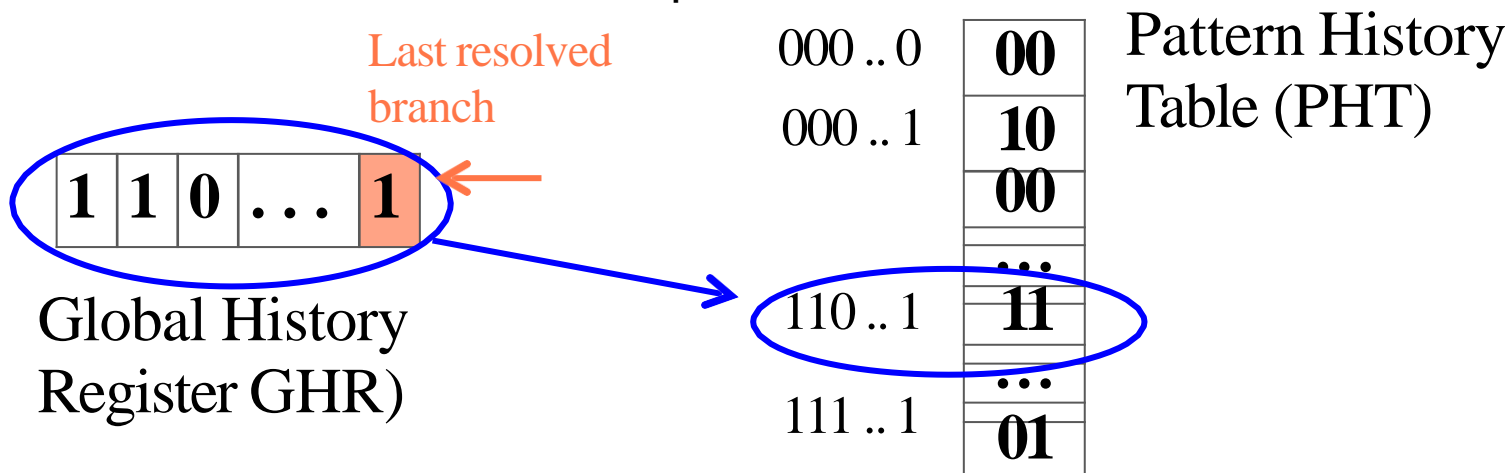
**Idea: exploit recent history of other branches in prediction**

# Correlating Branch Predictor

- 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 of that recorded the outcome that was seen for that GHR value in the recent past → Pattern History Table .
- Global history/branch predictor
- Uses two levels of history (GHR + history at that GHR)

# Global Two-Level (or 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





# General Performance Impact of Hazards

---

$$\text{Speedup from pipelining: } S = \frac{\text{CPI}_{\text{unpipelined}}}{\text{CPI}_{\text{pipelined}}} \times \frac{\text{clock}_{\text{unpipelined}}}{\text{clock}_{\text{pipelined}}}$$

$$\text{CPI}_{\text{pipelined}} = \text{ideal CPI} + \text{stall cycles per instruction} = 1 + \text{stall cycles per instruction}$$

$$\text{CPI}_{\text{unpipelined}} \sim \text{pipeline depth}$$

$$\frac{\text{clock}_{\text{unpipelined}}}{\text{clock}_{\text{pipelined}}} \sim 1$$

$$S = \frac{\text{pipeline depth}}{1 + \text{stall cycles per instruction}}$$

You are given a non-pipelined processor design which has a cycle time of 10ns and average CPI of 1.4. Calculate the latency speedup in the following questions.

Note: The solutions given assume the base CPI = 1.4 throughput. Since the question is ambiguous, you could assume pipelining changes the CPI to 1. The method for computing the answers still apply.

1. What is the best speedup you can get by pipelining it into 5 stages?

5x speedup.

The new latency would be  $10\text{ns}/5 = 2\text{ns}$ .

2. If the 5 stages are 1ns, 1.5ns, 4ns, 3ns, and 0.5ns, what is the best speedup you can get compared to the original processor?

The cycle time is limited by the slowest stage, so  $\text{CT} = 4\text{ns}$ .

$\text{Speedup} = \text{old CT} / \text{new CT} = 10\text{ns}/4\text{ns} = 2.5\text{x}$

3. If each pipeline stage added also adds 20ps due to register setup delay, what is the best speedup you can get compared to the original processor?

Adding the register delay, the new  $\text{CT} = 4.02\text{ns}$ .

$\text{Speedup} = 10\text{ns}/4.02\text{ns} = 2.488\text{x}$

# The speed up achieved in Pipelined processor.

$$\text{Speedup} = \text{Execution time}_{(\text{Non Pipeline})} / \text{Execution time}_{(\text{Pipeline})}$$

$$\text{Execution Time} = \text{CPI} * \text{Cycle time}$$

( CPI Cycles per Instruction)

$$\text{Cycle time} = 1 / \text{Clock Rate}.$$

# Does 4% accuracy improvement matter?

- Assume branches resolve in stage 10
  - Reasonable for a modern high-frequency processor
- 20% of instructions are branches
- Correctly-predicted branches have a 0-cycle penalty (CPI=1)
- 2-bit predictor: 92% accuracy
- 2-level predictor: 96% accuracy

## 2-bit predictor:

$$\text{CPI} = 0.8 + 0.2 * (10 * 0.08 + 1 * 0.92) = 1.114$$

## 2-level predictor

$$\text{CPI} = 0.8 + 0.2 * (10 * 0.04 + 1 * 0.96) = 1.072$$

**Speedup(2-level over 2-bit): 4%**