Branch Prediction

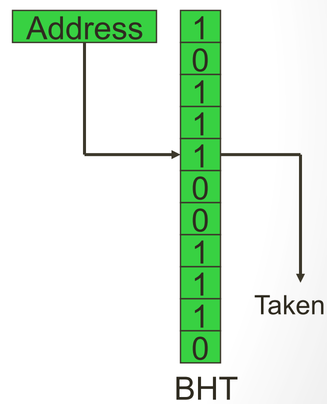
- Tackles problem of stalls from control dependencies
- Vital for multiple issue architectures
  - Branches arrive up to N times faster when issuing up to N instructions per clock cycle
  - Relative impact increases with lower potential CPI (from Amdahl's Law)
- Hardware based branch prediction
  - Dynamically predict outcome and target of branches
  - Uses run-time knowledge of branch behavior history

Branch Prediction

- Effectiveness dependent on
  - Prediction accuracy (how many predictions were correct)
  - Latency of correct predictions
  - Penalty of incorrect predictions
- Prediction accuracy and latencies depend on
  - Structure of pipeline
  - Type of predictor
  - Misprediction recovery strategies
- Local and global schemes
  - Local: predicts based on the current branch
  - Global: predicts based on previous related branches

Branch History Table (BHT)

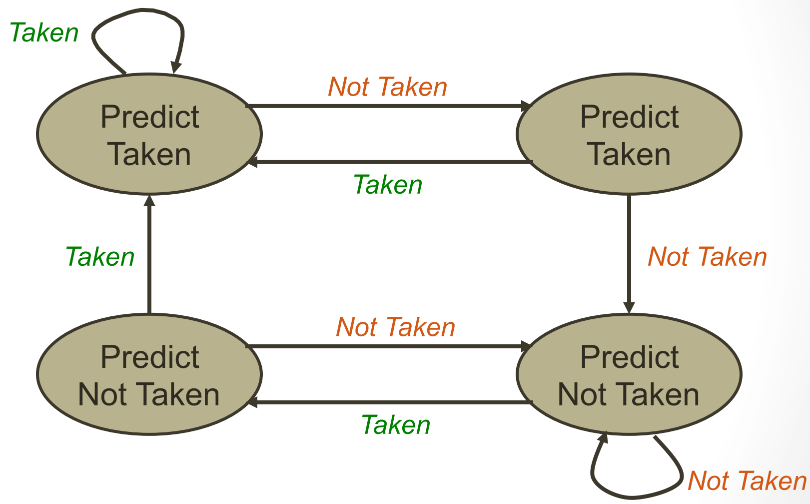
- Memory indexed by lower portion of address of branch instructions (a local scheme)
- A single bit indicates direction
  - Previously: 1=taken, 0=not taken
  - Previous direction is current prediction
- On a branch, record the correct outcome of the branch
- Multiple branches may map to the same table entry



Two-Bit Prediction

- Previous scheme - one-bit prediction
  - Consider a loop: even with all branches taken, there will be two mispredictions (one at the beginning and one when exiting the loop)
- Extend to two-bit scheme
  - A prediction must be inaccurate twice before it's changed

Two-Bit Prediction



State is recorded as two bits in the BHT

Two-Bit Saturating Counters

- Two-bit scheme may be implemented as a saturating counter
  - MSB indicates branch prediction
  - Increment on a taken branch
  - Decrement on a not-taken branch

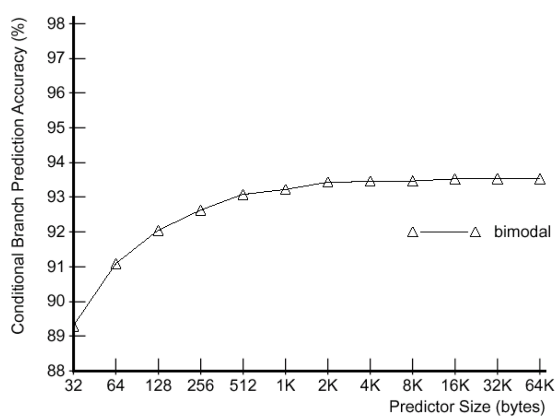
State	Description
00	No taken branches, initial
01	One taken branch
10	Two taken branches
11	Three taken branches

- Specialized case of *n*-bit saturating counter
  - Values 0 to 2^n-1,
  - Don't increment/decrement past maximum/minimum value
  - Predict taken when counter > one half maximum value
  - Two-bit scheme works nearly as well as larger number of bits

BHT Implementation

- A small cache accessed during IF
- Counter (two bits) attached to each cache line
- If branch predicted taken, fetch begins from target *as soon as target PC known*
- In DLX, the branch outcome and target are known at same time - no advantage for such a simple pipeline

Two-Bit Prediction Accuracy



Prediction accuracy for SPEC’89. Accuracy approaches that of an infinite table size.

BHT Performance

- "Bimodal prediction" works well - branches fall into one of two camps: taken or not taken
- Accuracy isn't enough - frequency also important
  - More frequent branches, the better accuracy required
- Integer codes (e.g., gcc, eqntott, espresso) may have very frequent branches
- With more ILP, accuracy (with frequency) becomes vitally important.

Improving on BHT

- Even with infinite table size - accuracy is not much improved over 4096 entries
  - Conflicts in the table isn't the problem
- Increasing bits per entry also does not help.
- Problem: *BHT uses only recent local history of a branch to predict future (not pattern based)*
- Solution: *Look at global history of other branches in making a prediction about the current one.*

Correlating Branches

- Branch history can lead to better decisions

```
if (aa==2)                       SUBUI R3,R1,2
    aa=0;                        BNEZ  R3,L1      B1
if (bb==2)                       ADD   R1,R0,R0
    bb=0;                  L1:   SUBUI R3,R2,2
if (aa!=bb) { ... }              BNEZ  R3,L2      B2
                                 ADD   R2,R0,R0
                           L2:   SUBU  R3,R1,R2
                                 BEQZ  R3,L3      B3
```

If B1 and B2 both taken, then B3 is probably not taken (110)

If B1 and B2 both not taken, then B3 is taken (001)

Correlating Branches

```
if (d == 0)                      BNEZ  R1,L1      B1
    d=1;                         ADDI  R1,R0,1
if (d == 1) { ... }        L1:   SUBUI R3,R1,1
                                 BNEZ  R3,L2      B2
                                 ...
                           L2:   ...
```

d	d==0?	B1	d before B2	d==1?	B2
0	Yes	Not taken	1	Yes	Not taken
1	No	Taken	1	Yes	Not taken
2	No	Taken	2	No	Taken

If B1 is not taken, then B2 is not taken (00).

One-Bit Predictor

d	B1 predict	B1 actual	New B1 predict	B2 predict	B2 action	New B2 predict
2	NT			NT		
0						
2						
0						

d alternates between 2 and 0

Predictors for B1 and B2 are initialized to not taken (NT)

What happens with the branch predictions???

One-Bit Predictor

d	B1 predict	B1 actual	New B1 predict	B2 predict	B2 action	New B2 predict
2	NT	T	T	NT	T	T
0	T	NT	NT	T	NT	NT
2	NT	T	T	NT	T	T
0	T	NT	NT	T	NT	NT

d alternates between 2 and 0

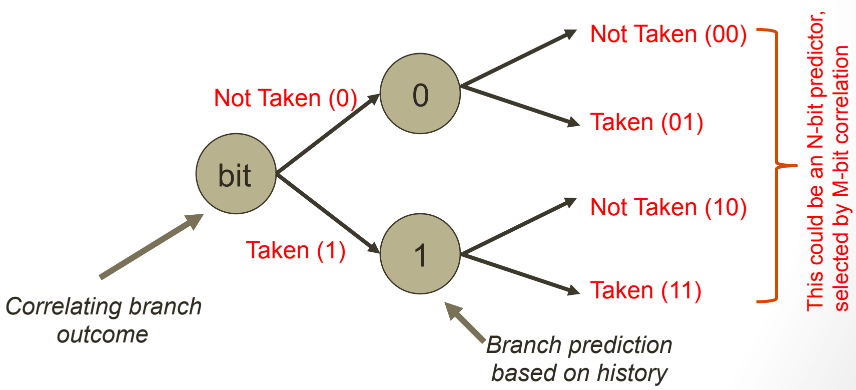
Predictors for B1 and B2 are initialized to not taken (NT)

What happens with the branch predictions???

All branches are mispredicted!

Prediction with Correlation

- With 1-bit of correlation, each branch predictor has a prediction for:
  - previous branch taken
  - previous branch not taken



1-Bit Pred., 1-Branch Correlation

d	B1 predict	B1 actual	New B1 predict	B2 predict	B2 action	New B2 predict
2						
0						
2						
0						

d alternates between 2 and 0

Predictors for B1 and B2 are initialized to not taken (NT/NT)

T/NT
- NT → If last branch is taken, use this prediction
- T → If last branch is not taken, use this prediction

What happens with the branch predictions???

1-Bit Pred., 1-Branch Correlation

d	B1 predict	B1 actual	New B1 predict	B2 predict	B2 action	New B2 predict
2	**NT**/NT	T	T/NT	NT/**NT**	T	NT/T
0	T/**NT**	NT	T/NT	**NT**/T	NT	NT/T
2	**T**/NT	T	T/NT	NT/**T**	T	NT/T
0	T/**NT**	NT	T/NT	**NT**/T	NT	NT/T

d alternates between 2 and 0

Predictors for B1 and B2 are initialized to not taken (NT/NT)

What happens with the branch predictions???

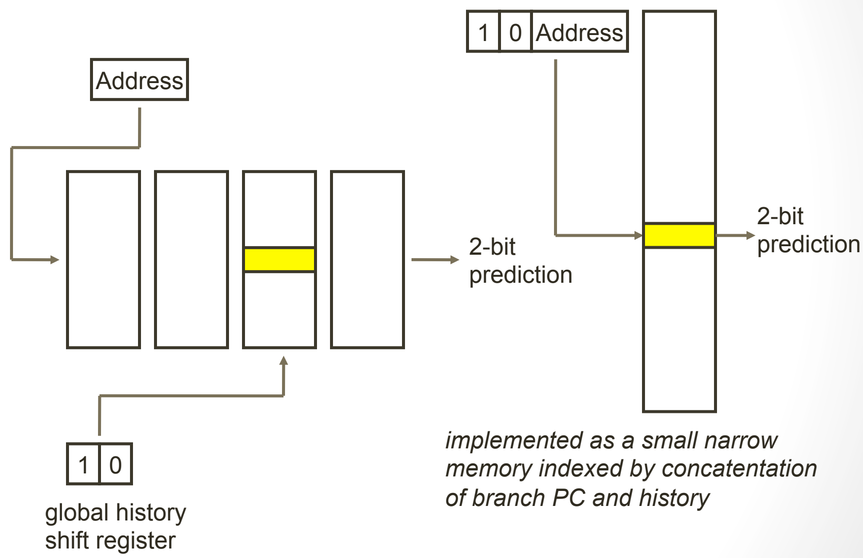
Notation: *prediction if last branch not taken/prediction if last branch taken*

Only the first iteration is mispredicted!

Prediction with Correlation

- *(m,n)* predictor
  - *m* bits of correlation
  - *n*-bit predictor for branch
  - last *m* branches (2^m) each with an *n*-bit predictor
- Implementation: Global history with selected address bits (so called "gselect")
  - *m*-bit shift register holds outcome of last *m* branches
  - BHT indexed by *m:low(PC)*
  - BHT can also be indexed just by *m* (global history prediction)

(2,2) Implementation



Trade-off in (m,n) Predictor

- *m* bits used to select predictor entry
- *m* = *a* + *b* bits
  - *a* is number of address bits
  - *b* is number of history bits

- We want enough address bits that each branch is reasonably well identified, along with an increasing number of history bits.

- Bimodal is *b=0, a=m*
- Global history is *b=m, a=0*

Local Branch Prediction

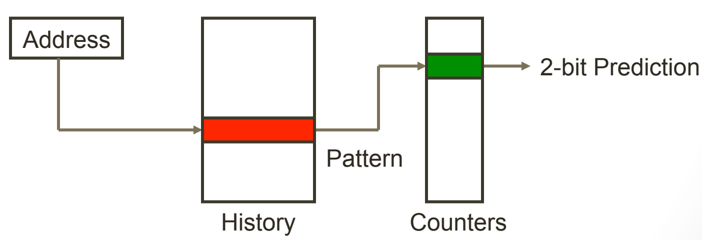
- Consider the loop

```
for (i=1; i<=4; i++) {...}
```

- Loop branch executes with pattern $(1110)^n$
- If we know how the branch has behaved previously, we can predict it.
- Local predictors use the past history of a *particular* branch (unlike the previous scheme - a global predictor)

Local Branch Prediction

- A two-level history table
- Table 1: history of recent branches indexed by the low address bits of branch instruction PC
- Table 2: two-bit branch predictors indexed by the history from table 1



Local Branch Prediction

- Assume some branch executed repeatedly.
- With 3 bits of history and 2^3 counters, the predictor can always predict the branch.
- Each execution has unique history (to index into prediction table)

Shift in 1 on a taken branch to the history

History	History
000 - iteration 0	100
001 - iteration 1	101
010	110 - iteration 4
011 - iteration 2	111 - iteration 3

Contention in Local Predictors

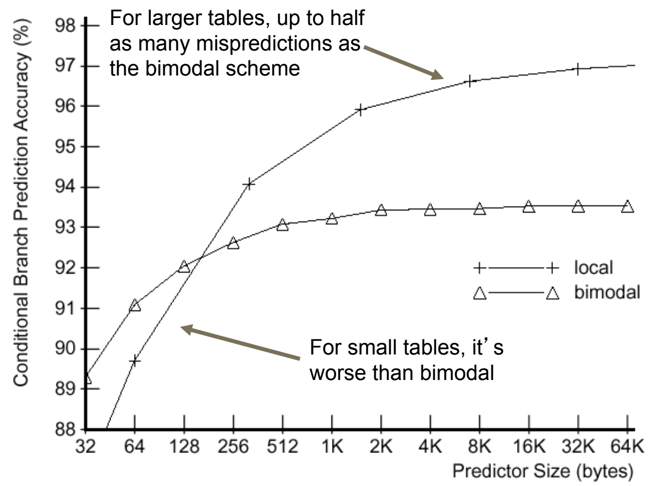
Local predictors can suffer from contention

(1) History may be a mix of histories from different branches that map to the same history entry

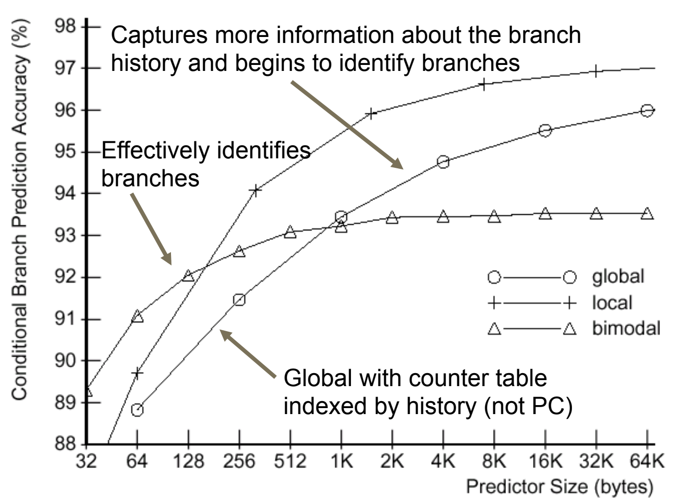
(2) Conflicts on similar history patterns

- E.g., $(0110)^n$ and $(1110)^n$ map to same entry when branch history entry says "110".

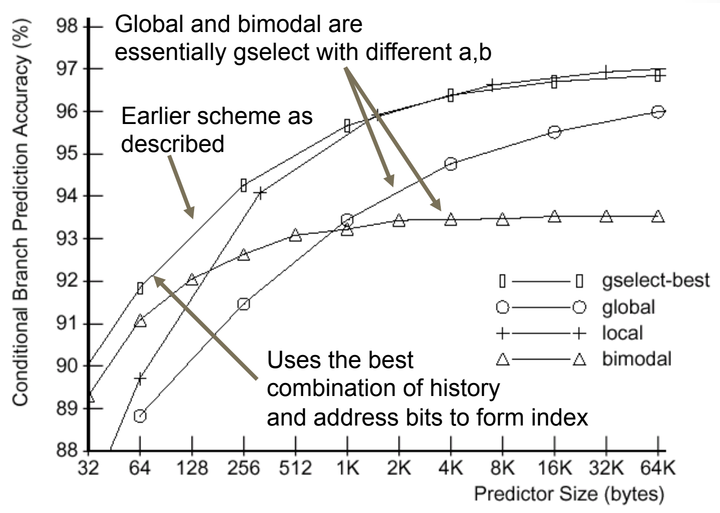
Local Branch Prediction Accuracy



Local vs. Global Accuracy



Local vs. Global Select



Local vs. Gselect

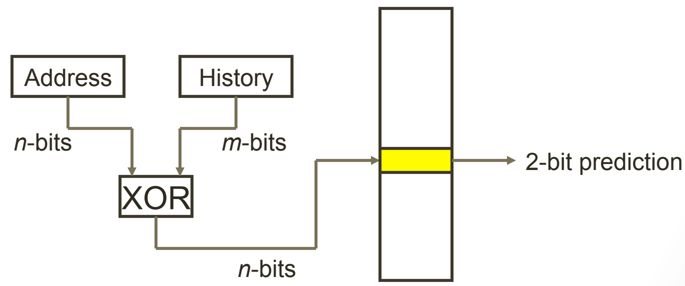
- Gselect better for < 1KB tables
- Local better for > 1KB tables (but gselect is close)

- gselect - storage space for global history is small

- gselect - a single array access
- local - two array accesses
- Thus, gselect potentially faster

Global with Index Sharing

- So called "gshare" predictor
- Similar to "gselect" predictor, except the branch address and global history are combined by doing an XOR



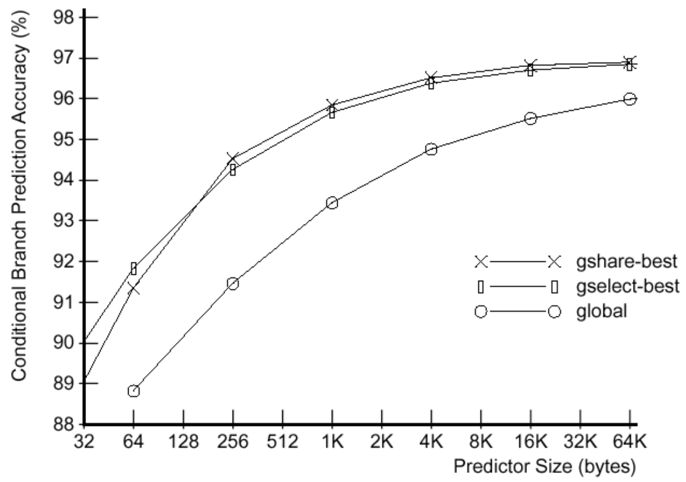
Global History w/Index Sharing

- Hash on the address + global history
- Better able to identify branches

Branch Address	Global History	gselect	gshare
00000000	00000001	00000001	00000001
00000000	00000000	00000000	00000000
11111111	00000000	11110000	11111111
11111111	10000000	11110000	01111111

- Gselect lost the history in the upper four bits

Gshare vs Gselect



Tournament Branch Predictors[1]

- Combine previous schemes into a scheme that has advantages of both
- Select among predictors P1 and P2
- A separate counter array picks among P1 and P2 - i.e., which prediction to use.
- 2-bit saturating counter - counters keep track of which predictor is more accurate

[1]also known as "combining predictors"

Keeping Track of Predictor Accuracy

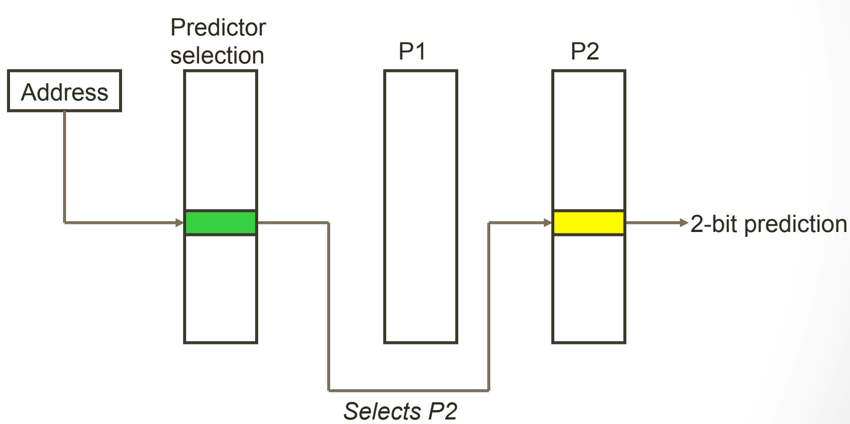
- 2-bit counter incremented/decremented

P1-correct	P2-correct	P1-correct - P2-correct	Action
0	0	0-0 = 0	None
0	1	0-1 = -1	Decrement
1	0	1-0 = 1	Increment
1	1	1-1 = 0	None

Counter value	Use predictor
00	P2
01	P2
10	P1
11	P1

Selects which predictor table to use for the prediction

Tournament Predictor Implementation



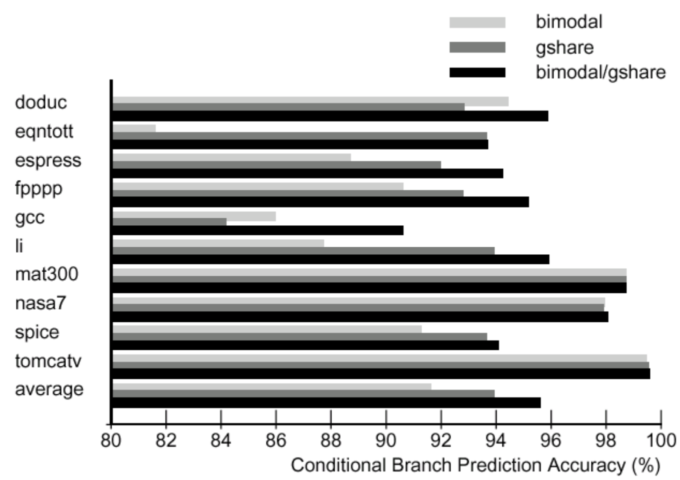
Bimodal/gshare Tournament Predictor

- Branches tend to show either local or global history

- Bimodal - use when local history is beneficial
- Gshare - use when global history is beneficial

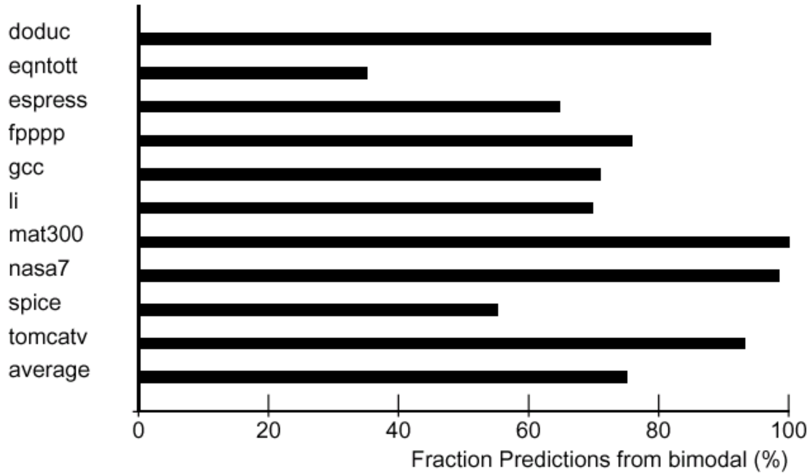
- Adapts to the particular branch by way of the predictor selection mechanism

Tournament Predictor Performance



Tournament predictor always better than either predictor alone

Which Contributes the Most?



Usually bimodal used more, but gshare helps and the predictor is chosen on a per branch basis

Branch Target Buffer (BTB)

- In DLX, we need the fetch address at end of IF
- Need to know: Undecoded instruction is a branch and what the next PC should be.

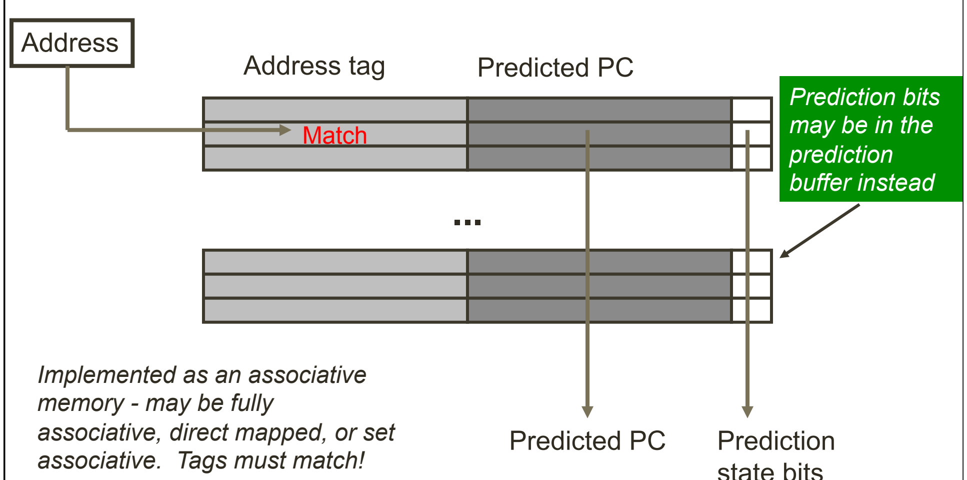
- Buffer to hold next predicted branch target address - "branch target buffer"

- Essentially, with the branch direction prediction, we can also buffer the predicted target address.

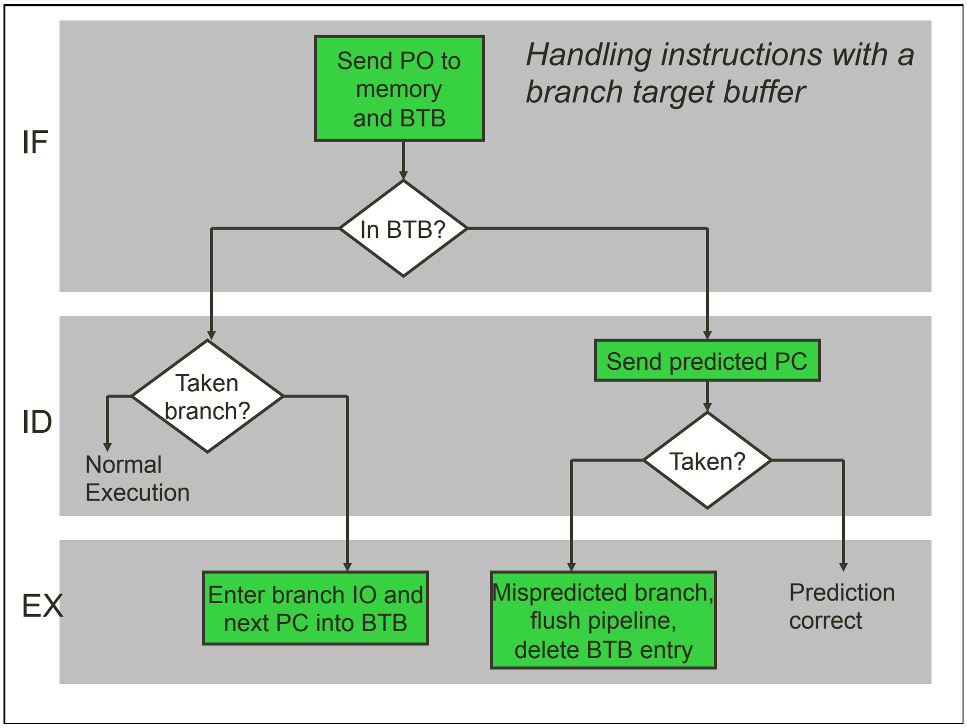
Branch Target Buffer

- Need to know whether the fetched instruction is predicted as a taken branch.
- Unlike BHT, we must tag all entries to ensure the entry corresponds to an actual branch.
- We don't even know if the instruction is a branch since it's not decoded!
- Store only predicted taken branches in BTB
  - May require two tables: One for predicted branch targets and one for the branch predictor.

BTB Implementation



Implemented as an associative memory - may be fully associative, direct mapped, or set associative. Tags must match!



Branch Prediction Summary

- Local - history of a single branch (pattern)
- Global - correlating branches
- Combined - some branches better predicted with global than local and vice versa. Combined can select among both.