Branching

Uses of branches

- Divert execution to new address
- Unconditional
  - Subroutines & returns
  - Jumps
- Conditional
  - Conditional execution (if-then-else)
  - Loops
- Exceptions
  - System calls, error conditions
- Branches are expensive

Subroutine calls

- RISC-like
  - `JAL dest src literal`
  - No special return instruction
  - No special stack instructions

- CISC-like
  - `CALL` [dest]
  - `RET`
  - Implicit call stack

Conditional Branches

- Program flow diverted only under computed condition
- Variation 1:

```
SUB  R2, R5, R11 /* sets CC flags */
BZ   #428        /* tests Z flag */
```

- Variation 2:

```
CMP  R2, R1, R4     /* compare R1, R4; result in R2 t
BZ   R2, #428
```

- Variation 3:

```
CMPBZ  R1, R4, #428
```

Branch Facts

- Usually use PC-relative addressing
- Some architectures have multiple PSWs
- One out of every 5 to 6 instructions
- 60% to 75% of branches are taken

Branch Handling: Terminology and Stats

Taken branch: a branch instruction that causes an instruction from a non–consecutive location to be fetched

Branch target: the address to which a taken branch transfers control to

Branch penalty: number of pipeline bubbles resulting from a taken branch

Branch Resolution: resolving the direction in which a branch transfers control to

Fall–through part: instruction sequence starting with the instruction immediately following the branch; these instructions are executed if the branch is not taken (hence, fall–through)

Branch Challenges

- Flow dep from compare to branch
- Delay in decoding branch
- Delay in resolving branch direction
- Delay in computing target effective address
- Delay in fetching branch target instructions

Branch Solutions

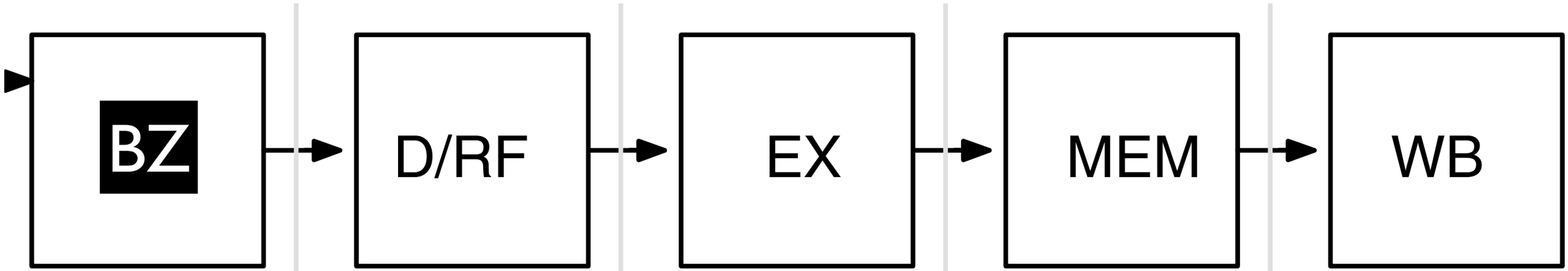
- Delay slots
- Predication
- Speculative execution

Branch Example

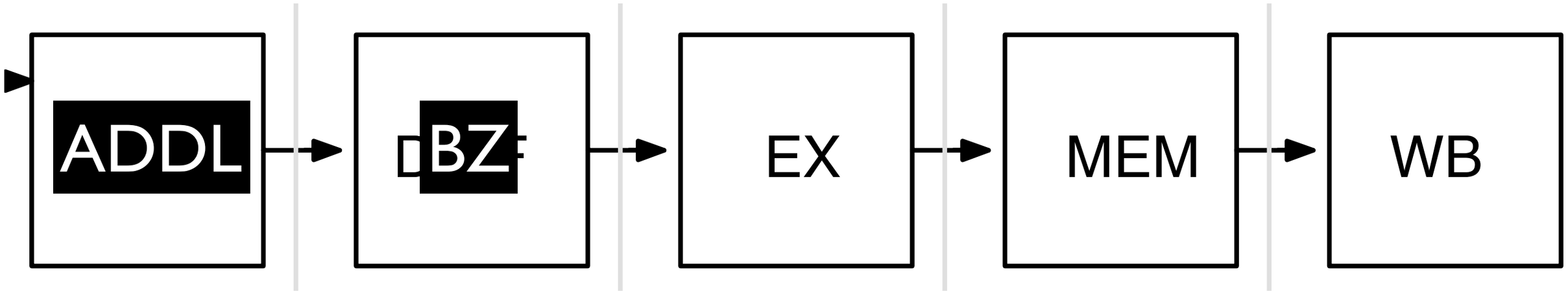
```
        SUB     R1, R2, R4
        CMP     R1, R6
        BZ      target
        ADDL    R1, R1, #4
        LOAD    R2, R4, #0
                :
                :
target: STORE   R6, R2, #0
```

Fall-through

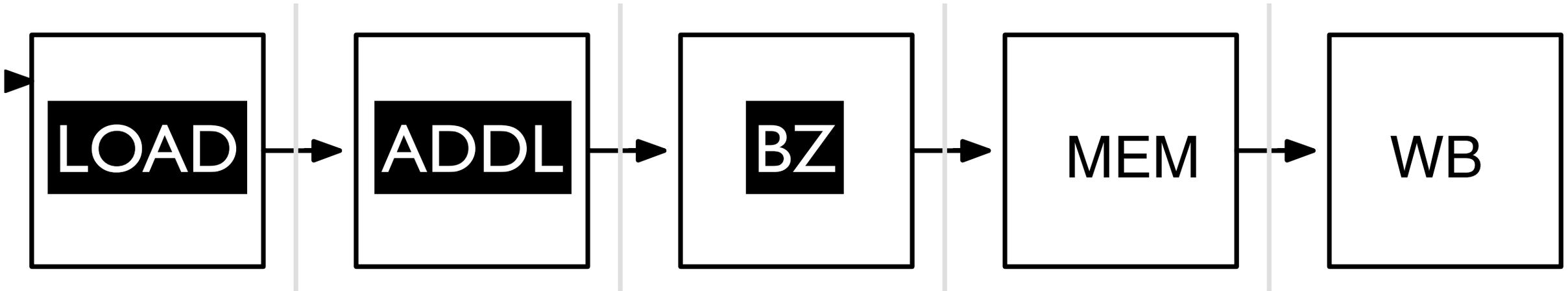
```
BZ     target
ADDL   R1, R1, #4
LOAD   R2, R4, #0
```



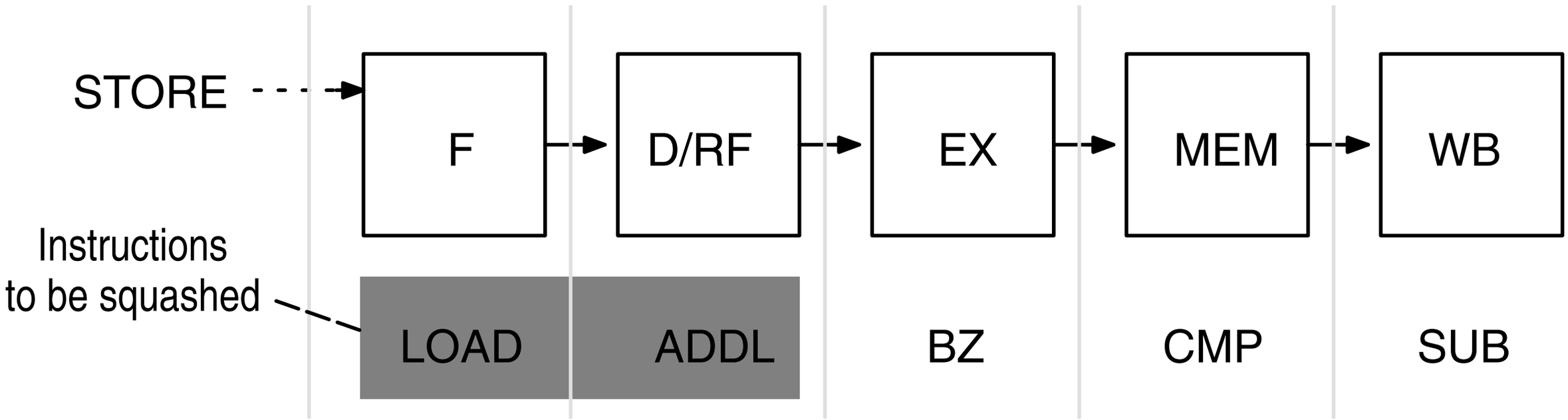
```
BZ     target
ADDL   R1, R1, #4
LOAD   R2, R4, #0
```



```
BZ     target
ADDL   R1, R1, #4
LOAD   R2, R4, #0
```



```
                SUB    R1, R2, R4
                CMP    R1, R6
                BZ     target
                ADDL   R1, R1, #4
                LOAD   R2, R4, #0
                       :
                       :
target:         STORE  R6, R2, #0
```



squashing or **flushing** or **annulment**.

An analysis:

Execution time of N instructions without branching:

$$T_{exec} = k * T + (N - 1) * T$$

Execution time with branching:

b = probability that an instruction is a branch

s = probability that the branch instruction is taken

P = length of bubbles introduced in the pipeline on a taken branch

- Each taken branch effectively prolongs the execution time by P cycles

- Total execution time with branching is thus:

$$T_{exec,branch} = T_{exec} + N.b.s.P$$

Example values: for APEX: P = 2, k = 5; from stats: b = 0.2, s = 0.75

When N is large,

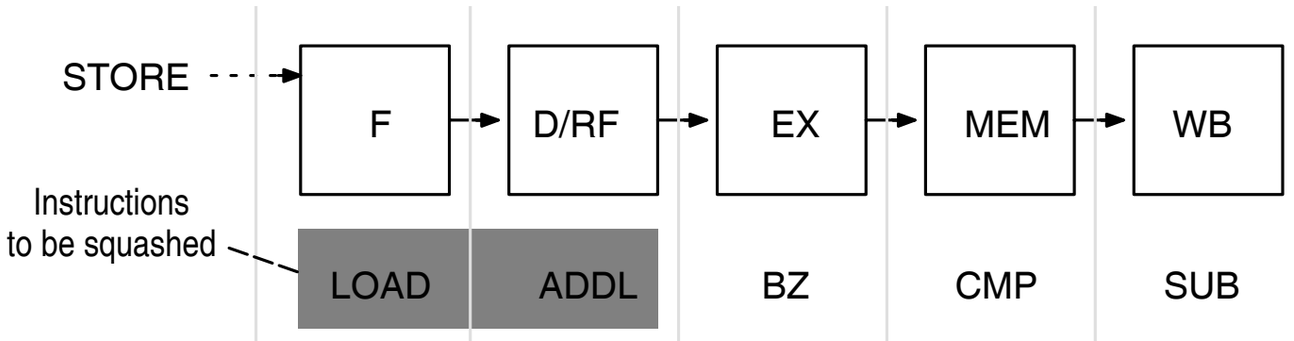
$$T_{exec,branch}/T_{exec} = 1.3$$

$\Longrightarrow$ execution time is prolonged by 30% due to branching in scalar pipelines

(Branching penalty is *more severe* in superscalar pipelines, as we will see later.)

Instruction Squashing

- Updating processor state
  - OOO: Rely on RoB to undo
  - In-order: Drop subsequent instructions
- Cache miss
  - Only affects performance, not correctness
- Page fault
  - Not processed until earlier instructions commit



Flow dependency over PSW

- OOO: Must enforce explicitly
- Store PSW in GPR: Use existing flow dependency mechanisms
  - Multiple PSW: Explicitly coded dependency
  - Combined eval and branch -- not applicable
  - Explicit setCC flag allows selective update to PSW

Techniques for Avoiding or Reducing the Penalty of Branching

- Early Compares
- Unconditionally Fetching Instruction Stream Starting at the Target
- Static Prediction
- Delayed Branching and Delayed Branching with Squashing

- Dynamic Branch Prediction:

– Branch History Table

– Branch Target Buffers

– Alternate Stream Prefetching Based on Prediction

- Separate Branch Unit

- Predicated Execution (aka conditional assignments/guarded execution)

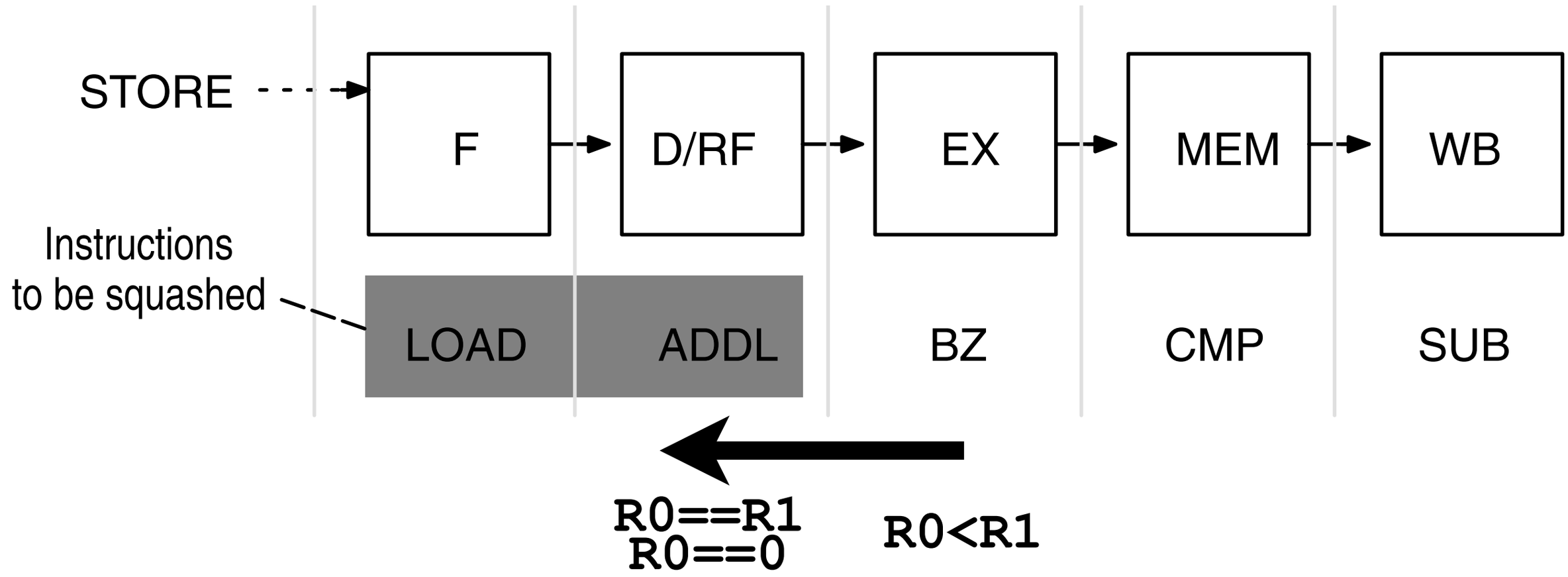
- Branch Folding

- Two–level Branch Prediction

- Hybrid Branch Handling Techniques & Others

Early Compares

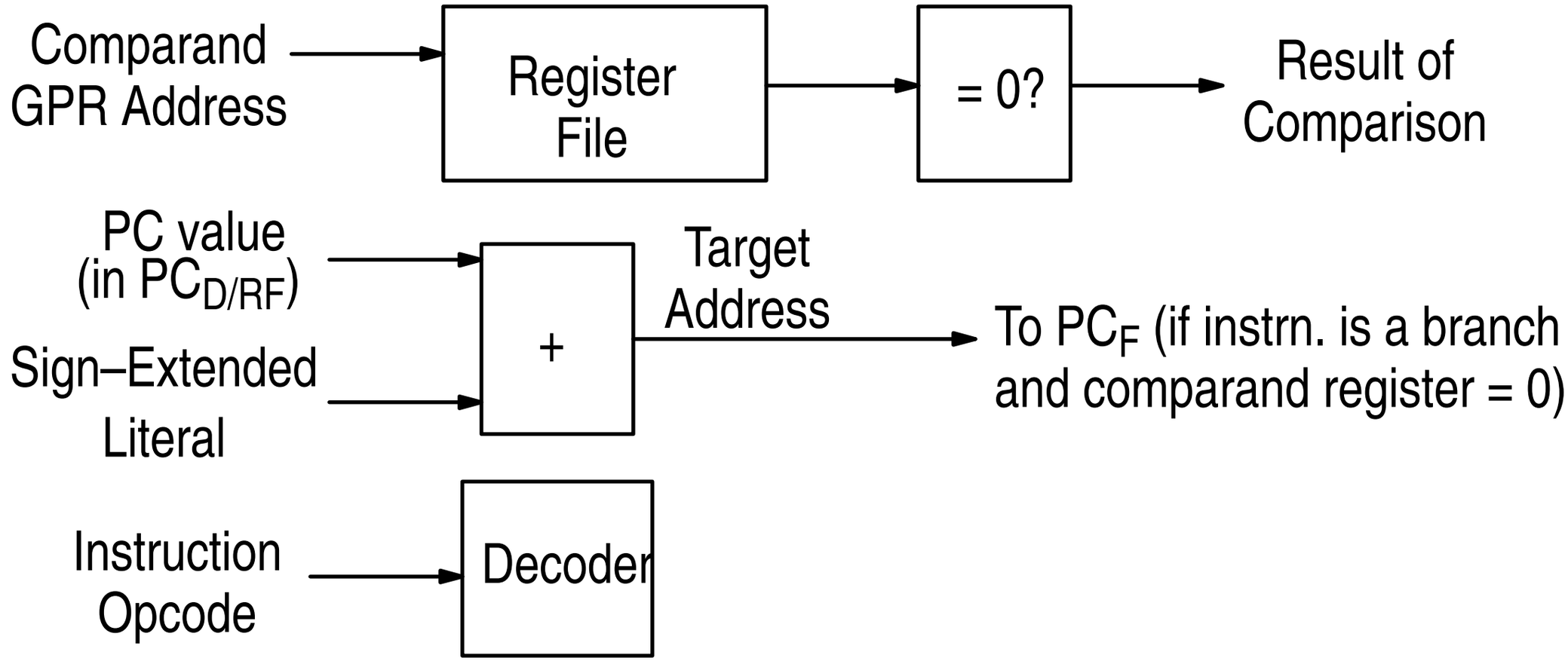
- Resolve some types of branches earlier in pipeline




```
for (i = 0; i < N; i++) { }
```

```
for (i = 0; i != N; i++) { }
```

- The decode stage also includes a dedicated adder to compute the branch target address – this address computation also proceeds speculatively in parallel with the decoding



Parallel activities within the D/RF stage for Early Compares

Control Dependency

- Execution is dependent on branch or other flow control.
- Example1:

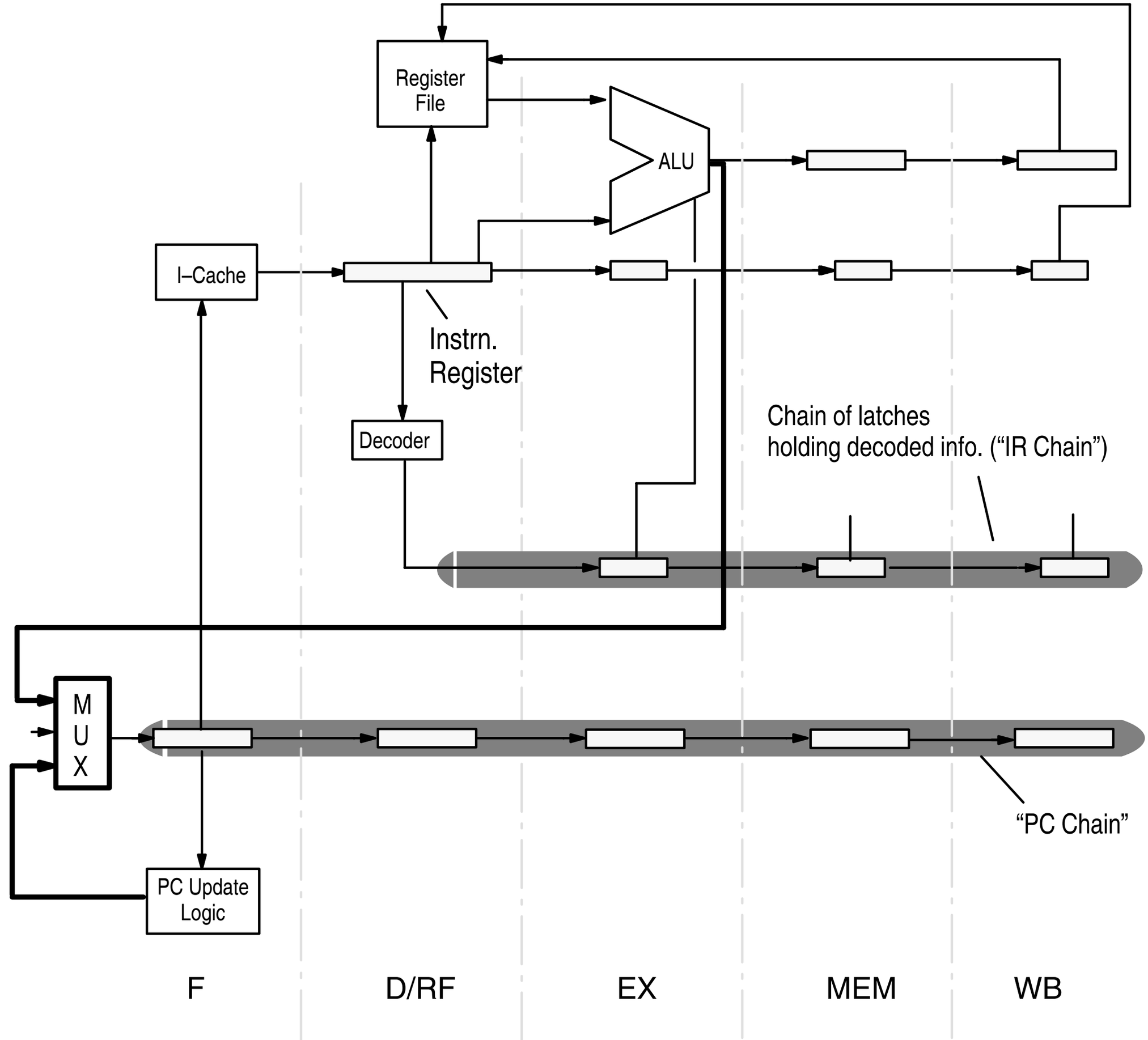
```
if  <condition>  {<statement_sequence_1>};
    <statement_sequence_2>;
```

- Example 2:

```
if  <condition>
        {<statement_sequence_1>}
    else
        {<statement_sequence_2>};
```

Delayed Branching

- Delay slot: N instructions following branch are not control dependent on branch.
- Update to PC occurs "late."
- Compiler post-processing moves independent instructions from before branch into delay slots.


Register File
ALU
I–Cache
Instrn. Register
Decoder
Chain of latches holding decoded info. (“IR Chain”)
M U X
PC Update Logic
“PC Chain”
F
D/RF
EX
MEM
WB

Making use of the delayed branching mechanism:

- Lets first consider what the delayed branching mechanism does by looking at the processing of the following code fragment on the simple version of APEX with delayed branching hardware and a delay slot of 2 cycles:

```
        I1
        I2,
        BZ target
        I3
        I4
        I5
        :

target: I20
```

  – When the branch is not taken the instruction sequence processed is:
    I1, I2, **BZ, I3, I4,** I5,....

  – The instruction sequence processed when the branch is taken is:
    I1, I2, **BZ, I3, I4,** I20, I21,....

- ► Note that the two instructions in the delay slot (I3 & I4) are never squashed – they are processed no matter which way the branch goes.

How to fill delay slots

- Preserve original data dependencies
- Move instructions independent of branch
- Move instructions that do useful work

Using an interlocking algorithm

- Consider both data and control dependencies
  - Cannot move control-dependent instructions
- Schedule conditional branch early
- Last resort: Fill with NOPs

An Example:

– Suppose the original code fragment to be reorganized to exploit the delayed branching hardware for APEX is:

```
            LOAD   R2, R5, #2
            ADDL   R5, R5, #1
            SUB    R1, R2, R4
            CMP    R1, R6
            BZ     target
            ADDL   R1, R1, #4
            LOAD   R2, R4, #0
                   :
                   :
target:     STORE  R6, R2, #0
```

– The reorganized version is:

```
LOAD   R2, R5, #2
ADDL   R5, R5, #1
SUB    R1, R2, R4
CMP    R1, R6
BZ     target
ADDL   R1, R1, #4
LOAD   R2, R4, #0
       :
       :
target: STORE  R6, R2, #0
```

```
LOAD   R2, R5, #2
SUB    R1, R2, R4
CMP    R1, R6
BZ     target
ADDL   R5, R5, #1
NOP            /* filler */
ADDL   R1, R1, #4
LOAD   R2, R4, #0
       :
       :
target: STORE  R6, R2, #0
```

Facts:

- One useful instruction to be moved into the delay slot of a branch can be found about 70% of the time
- The average probability of finding two useful instructions to be moved into the delay slot is 25%
- The average probability of finding three or more useful instructions that can be moved into the delay slot is very small

▶ It is thus clear that the delayed branching mechanism is of little use in pipelines where the delay slot is 2 cycles long or more

▶ In modern superscalar pipelines, a single cycle delay slot can encompass 2 to 6 instructions (if the machine uses 2–way to 6–way dispatching per cycle). Delayed branching is also fairly useless in these CPUs.

– Delayed branching was used extensively in many early RISC *scalar* CPUs

Delayed Branch with Squashing

- Predict direction at compile time
- Can squash instructions in delay slot if actual branch != prediction
- Move instructions from predicted path to delay slot (target or fall-through)
- Benefits from good predictions
- Example: Loop: Fill delay slot with top of loop

Implementation:

- Two bits can be associated with each conditional branch instruction to encode the prediction and the squashing information – both of these bits can be set by the compiler. The interpretation of these two bits are as follows:

  00: No squashing, irrespective of branch direction
  – This is the pure delayed branching scheme

  01: Squash if the branch is not taken (this implies that the static prediction is that branch is taken most of the time)

  10: Squash if the branch is taken (this is used if the static prediction is that the branch is not taken most of the time)

  11: Squash, irrespective of the branch direction: this is the primitive scheme, with no special handling for branches.

Squashing heuristics

- 01 or 10 if prediction can be made
- 00 with no definite prediction
- 11 in all other cases

00 -- no squashing,
01 -- squash if not taken,
10 -- squash if taken,
11 -- squash irrespective.

Branch Prediction

- Make a good guess as to whether or not a branch will be taken as early as possible
- Needs to be reasonably accurate
- Defer branch resolution while allowing execution to continue …
- … hopefully on the right path.

Static Branch Prediction

- Compiler can often make reasonable prediction:
  - Loop branch often taken
  - Error condition check usually not taken
- Prediction specified as hint in instruction.
- Use profiling to determine typical behavior

Static Prediction as Fall-back

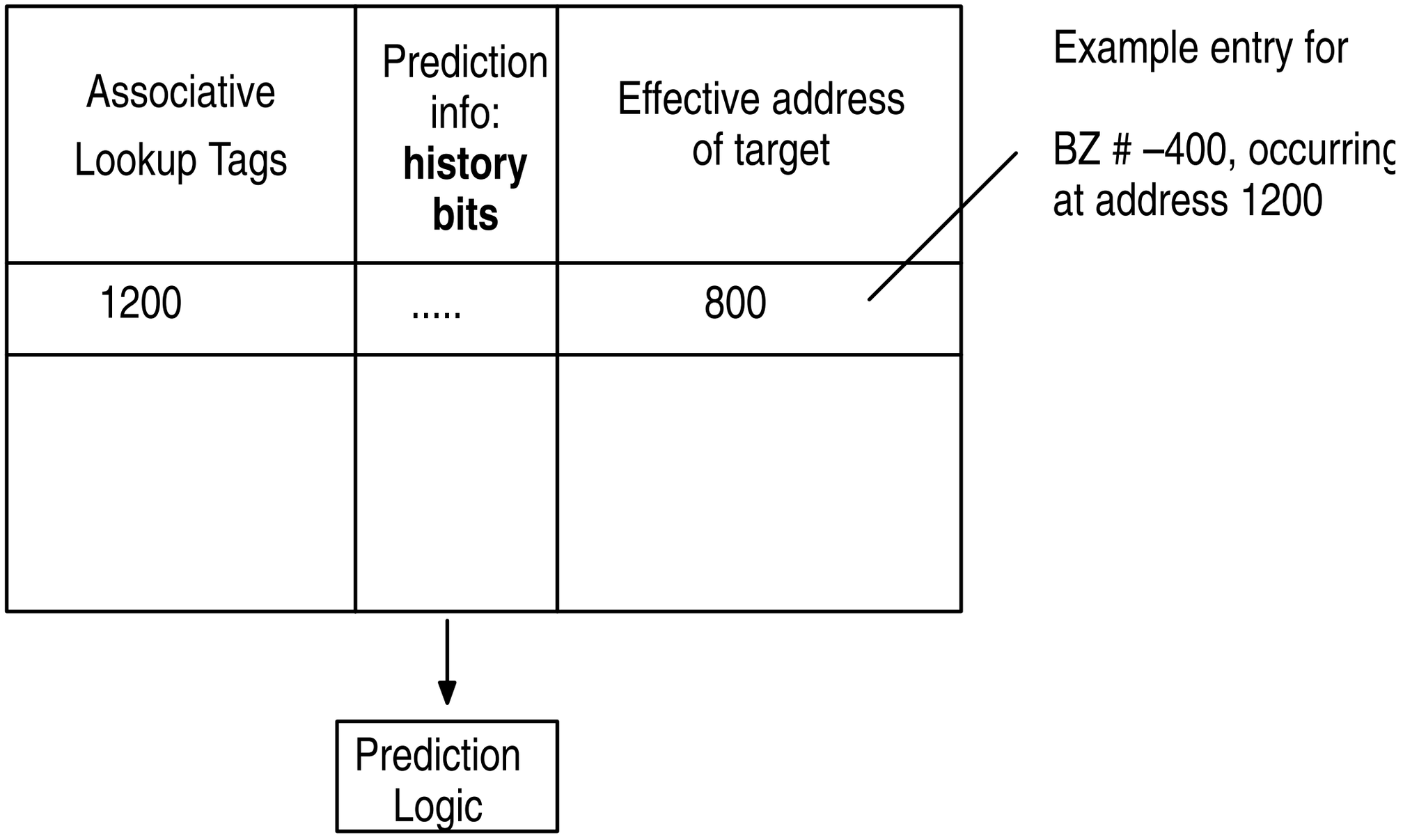
- Predict based on sign of branch offset
  - Positive: Not taken
  - Negative: Taken (e.g. loop)
- Option to switch to static prediction if dynamic performs poorly.

Dynamic Branch Prediction

- Past behavior is indicative of the future
- Keep track of branch history in Branch Target Buffer (BTB)
- BTB maintains statistics and is like a cache
- Uses address of branch as key, probed in parallel with I-cache

Example:

Fully–associative branch target buffer with explicit prediction information:



Possible outcomes/conclusions:

BTB hit, I–cache hit $\Longrightarrow$ instruction fetched is a branch instruction

BTB hit, I–cache miss $\Longrightarrow$ instruction to be fetched is a branch

BTB miss, I–cache hit $\Longrightarrow$ instruction may or may not be a branch

BTB miss, I–cache miss $\Longrightarrow$ instruction may or may not be a branch

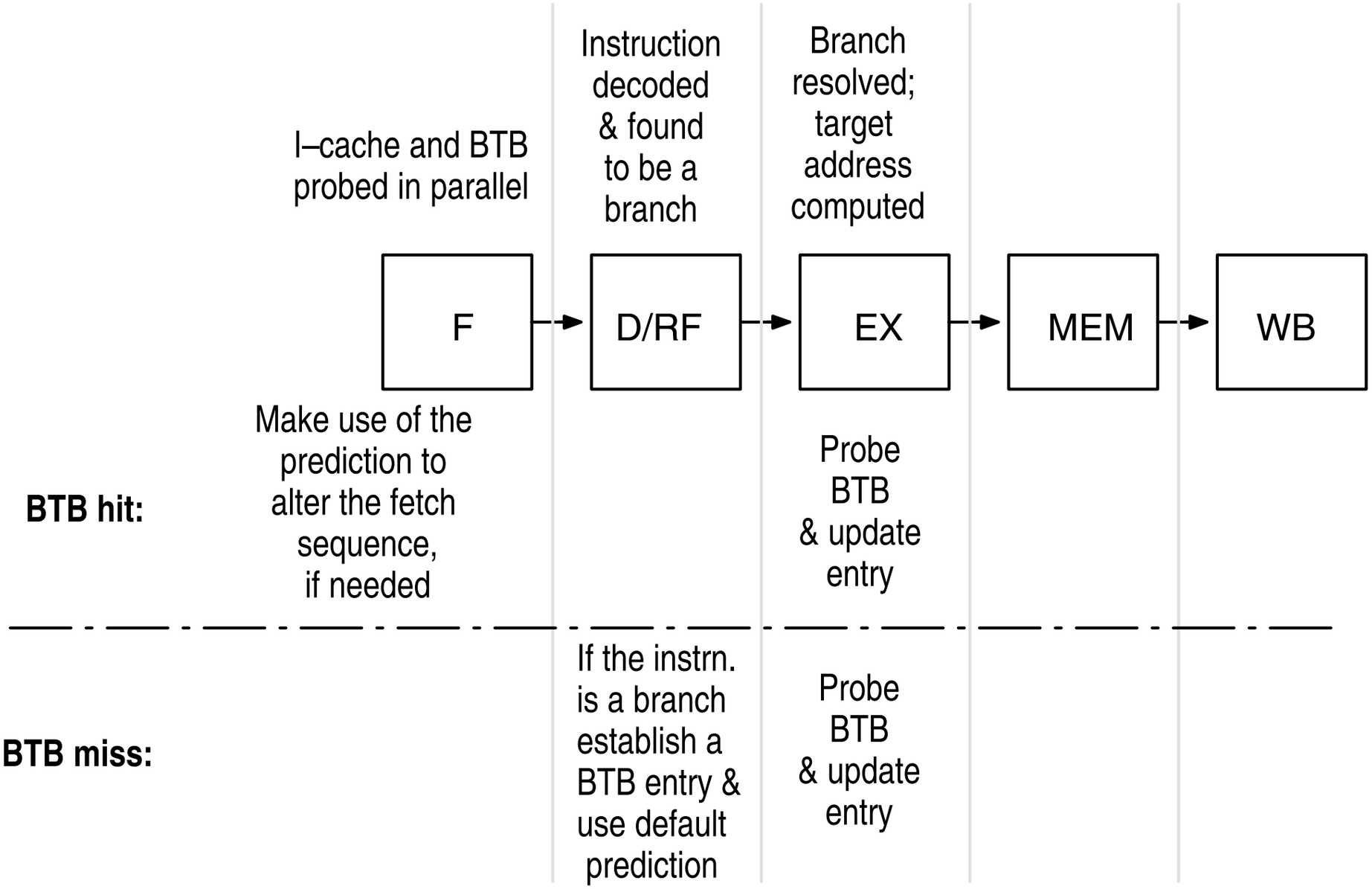
BTB Miss

- If fetched instruction turns out to be a branch:
  - Allocate BTB entry
  - Use default prediction
  - Update stats after branch resolved

BTB Hit

- Use stats in BTB entry to make prediction
- Update stats after branch resolved

- If a BTB is incorporated in the simple APEX pipeline, these actions are carried out as shown:



- <u>Penalties</u>: BTB hit and prediction correct: 0 cycles
  BTB hit and incorrect prediction: 2 cycles
  BTB miss, correct default prediction: 1 cycle (only on a taken branch, to squash instruction following branch; 0 cycles, otherwise)

BTB Hardware

- Multi-ported memory:
  - Probe from Fetch at the same time as Update from EX
- Associatively addressed or hash-based

What does BTB cache?

- Branch statistics
- Branch target address (PC-rel)
- Branch target instruction (folding)

- BTB may be merged with I-cache
- BTAC - stats in I-cache, target in BTAC

Constraints on BTB

- Can maintain finite history
- Requires quick decoding and prediction
  - Status maintained as FSM state
  - Prediction trivially computed from state

Simple Predictors

- One-bit predictor
- Two-bit predictor
  - Saturating counter

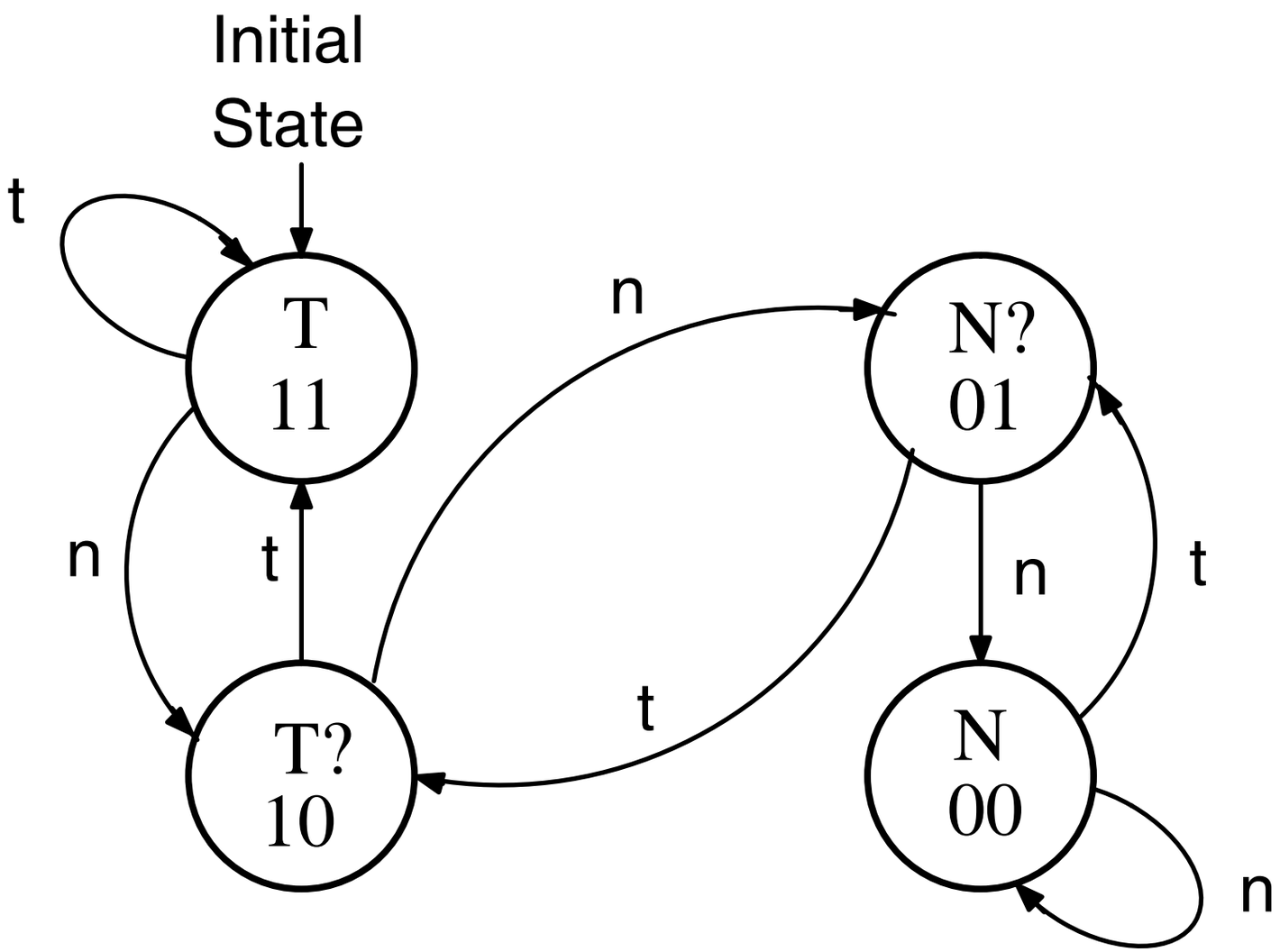
Example

Branch dir
t
t
t
t
n
t
t
t

1-bit	Prediction
1	t
1	t
1	t
1	t
1	t
0	n
1	t
1	t

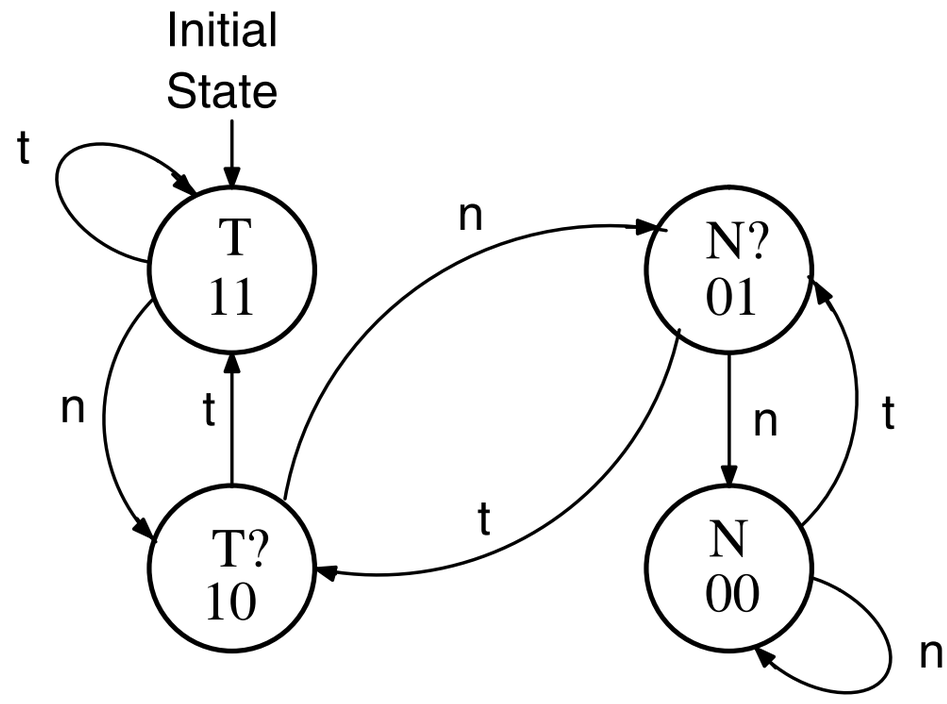
2-bit	Prediction
3	t
3	t
3	t
3	t
3	t
2	t
3	t
3	t

– The two bit saturating counter is implemented with the following FSM:



– N (predict not–taken, higher confidence), N? (predict not taken, lower confidence), T? (predict taken, lower confidence), T(predict taken, higher confidence)

– n, t: actual behavior of branch (t = taken, n = not taken)



– msb of the state label is the prediction

– Two bad guesses in a row changes prediction.

– Note need to start from the initial state (if we start in T? and alternately go through n and t, a correct prediction is never made).

- Accuracy of branch prediction:

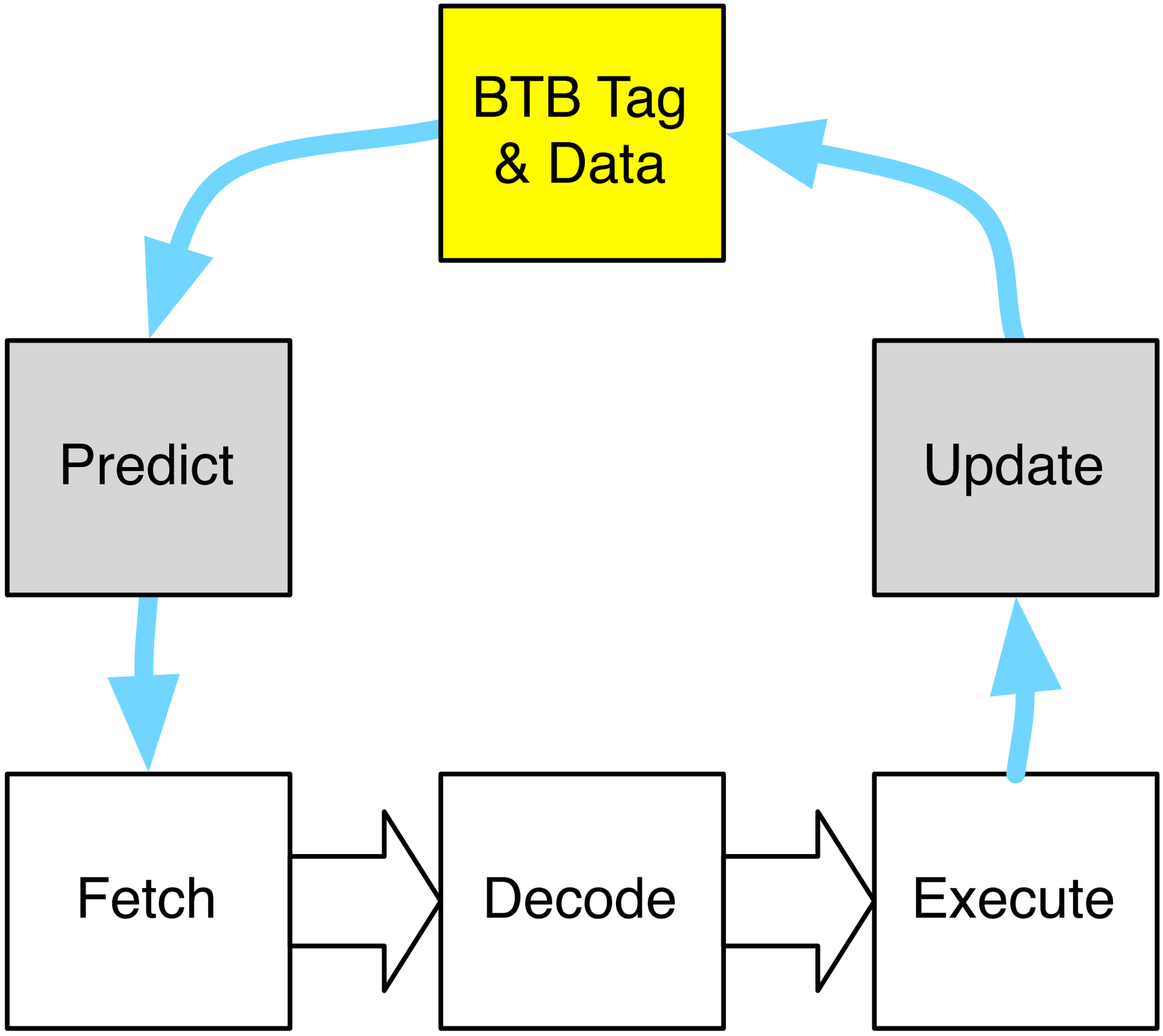
- (1984 data): Based on n bits of history per branch (CISCy + reg–to–reg ISAs)

# of history bits	Prediction accuracy (average)
0 (default prediction)	64.1% to 77.8 %
1	79.7% to 96.5%
2	83.4% to 96.5%
:	:
5	83.7% to 97.1%

– This *old* data shows that the accuracy of the predictor saturates after 2 to 3 bits of history.

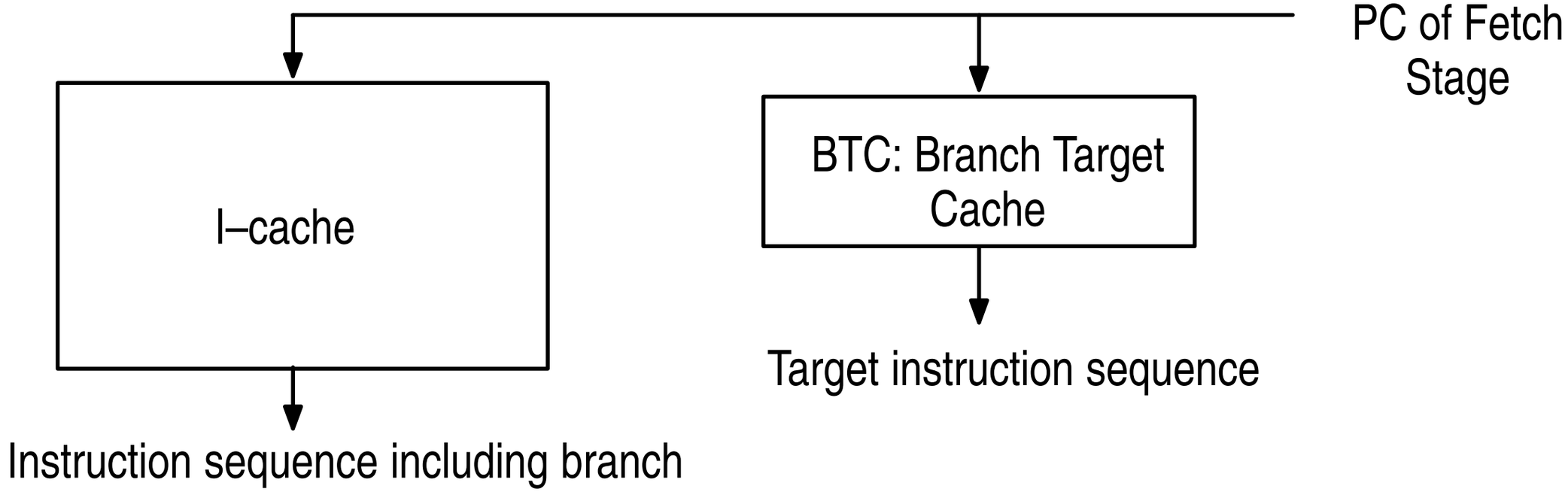
- Accuracy of prediction of 2–bit saturating counter (1992, SPEC92 benchmarks):

Prediction Technique	Accuracy (geometric mean/(range)) (over most SPEC 92 int+fp)
Static, always taken	62.5%
Static, based on sign of branch offset	68.5%/
Dynamic (1–bit)	89% (int: 76.5% to 87.5%, fp: 88% to 97.8%)
Dynamic (2–bit saturating counter)	93% (int: 86.5% to 93%, fp: 91.5% to 98.5%)


BTB Tag
& Data
Predict
Update
Fetch
Decode
Execute

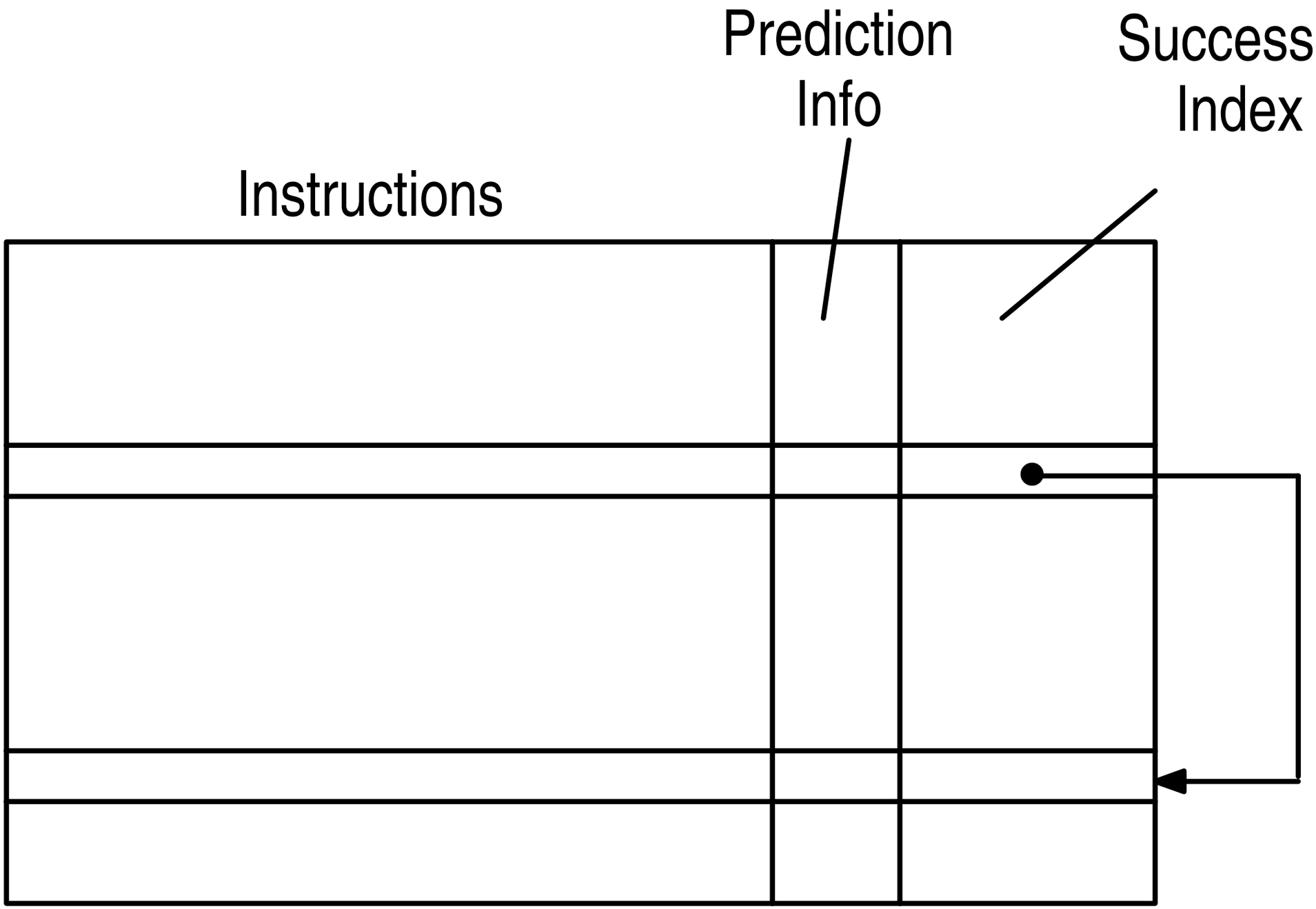
- BTB variations: examples

Separate I–cache and target instruction cache (BTC):



– I–cache and BTC are probed in parallel

Integrated I-Cache with Successor Index

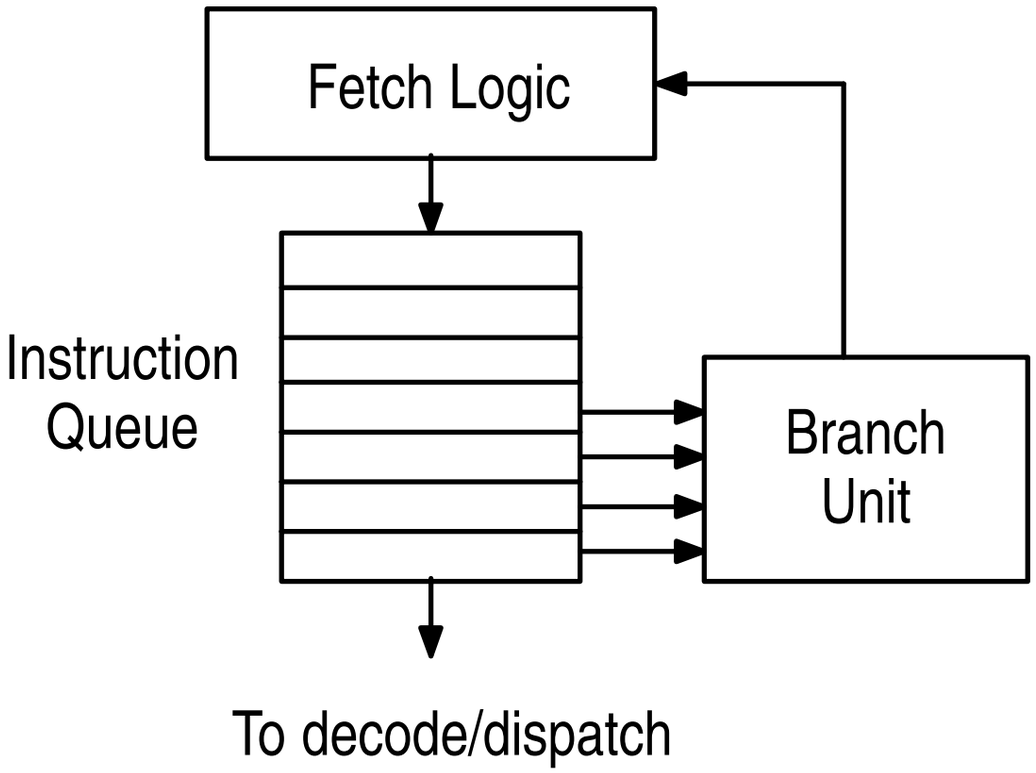


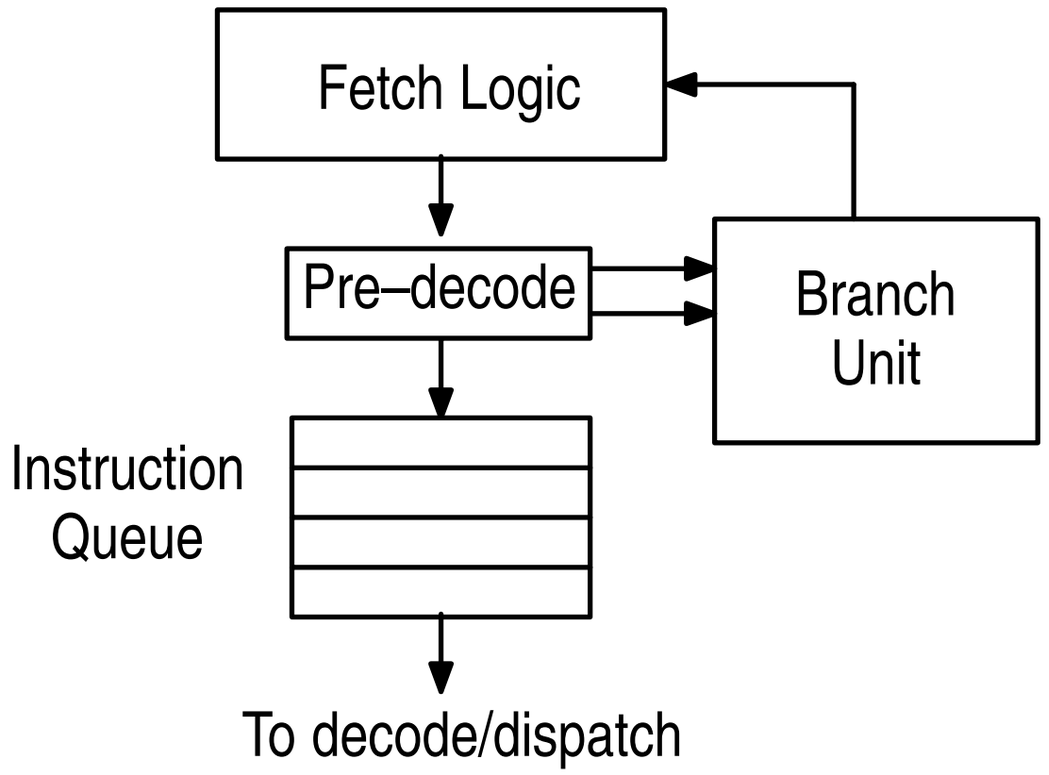
Integrated I–cache

Successor Field Benefits

- Saves 2-step I-Cache lookup (tag & data) for branch target.
- Reduces target lookup to 1 step.

Early Branch Resolution





- Dedicated Branch Unit
  - Has access to register file
  - Looks ahead in instruction sequence and pre-computes branches

Predicated Execution

- Expands *conditionals* to include non-branch instructions
- Instruction is *skipped* (NOP) if conditional evaluates to *false*
- Avoids branch overhead for short conditionally executed code sequences
- CMOV is a common predicated instruction

- Coding the following source code using CMOVs:

```
if (x == y) {
   a = c;
   y = x + 4;}
else
   a++;
```

```
CMP    R1, R2, R3
/* R2, R3 holds x, y; result of CMP in R1 */
CMOVEQ  R4, R1, R5   /* R4 holds a; R5 holds c */
ADDL   R6, R2, #4
CMOVEQ  R3, R1, R6
ADDL   R7, R4, #1
CMOVNEQ       R4, R1, R7
```

Branch History

- Use the past to predict the future
- Local history
  - What this branch did in the past
- Global history
  - Sequence of branches that led to this one
- **Two-level predictor**
  - Combine global and local history

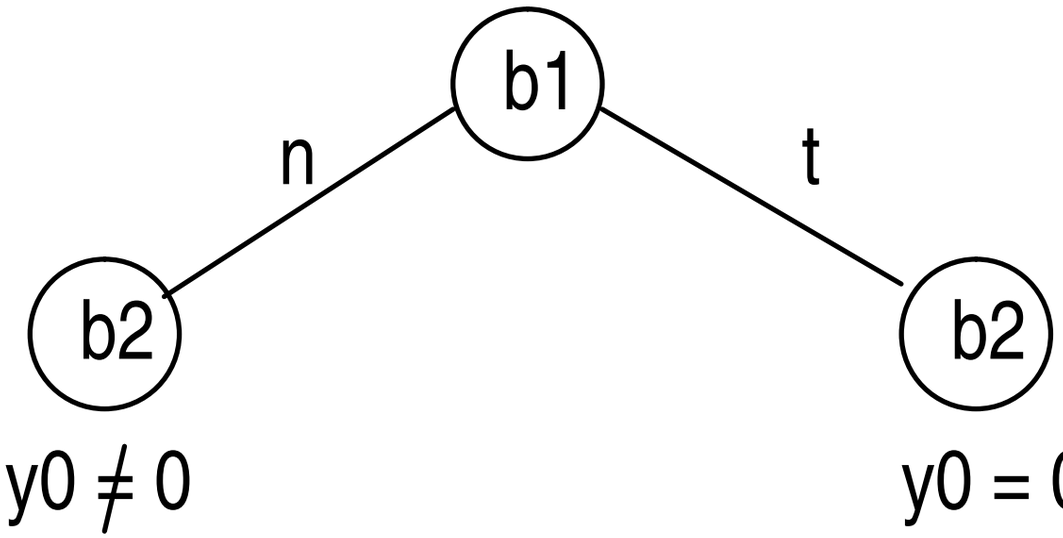
Example 1: This is from the gcc benchmark of the SPEC 92 (also SPEC 95) benchmark suite:

```
if (tem != 0)                    /* branch b1 */
    y0 = tem;
if (y0 == 0)                     /* branch b2 */
    return 0;
```

– Assume that the two "if"s are implemented using the branch instructions b1 and b2 as shown, and that the "then" clauses of the two "if"s are implemented as the fall–through part of these branches.

– This is really control dependencies that lead to data dependencies within a subsequent branch condition!

– The possible control flow paths leading from b1 to b2 are:



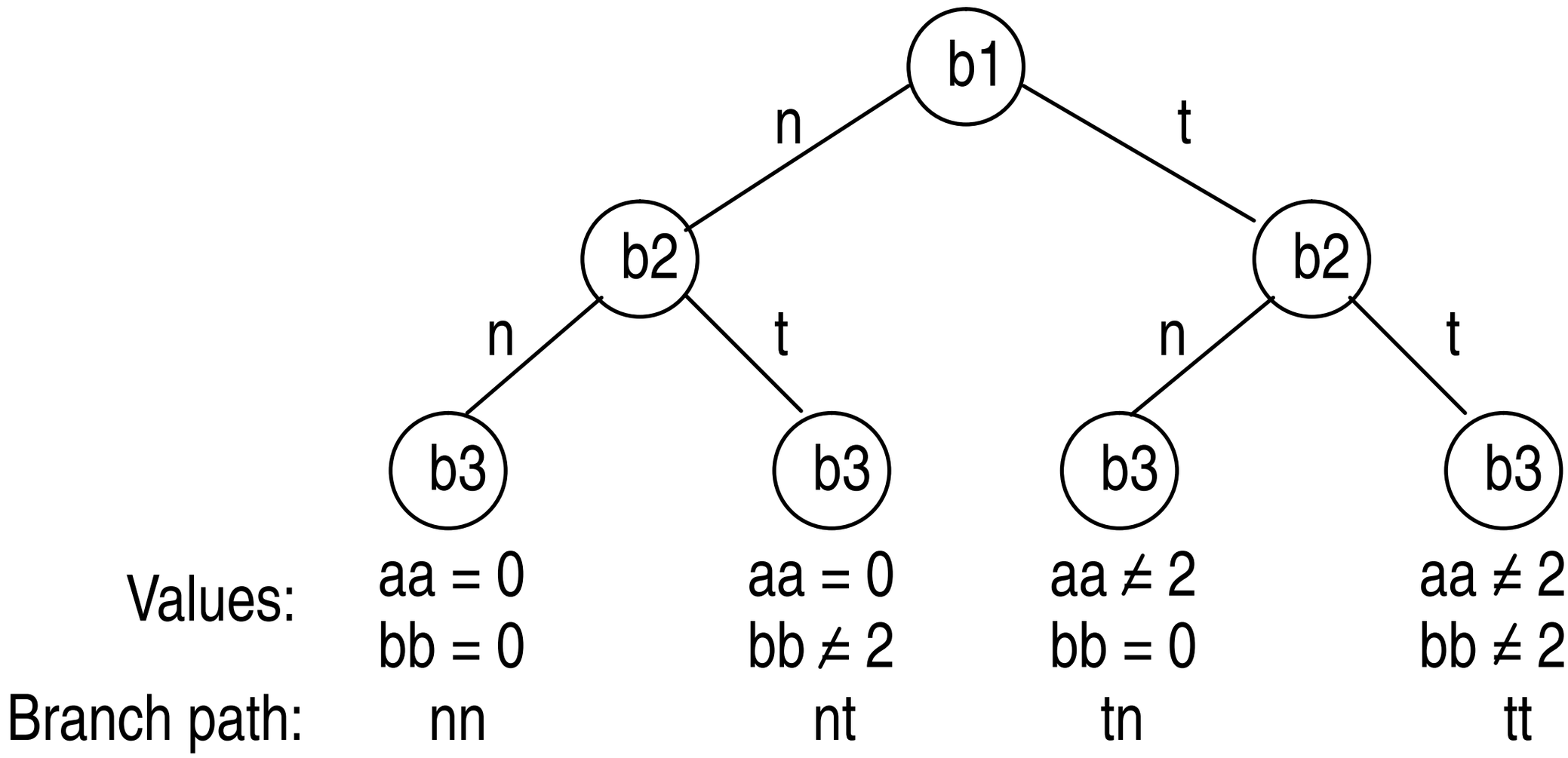
– In this case, the behavior of b2 can be predicted from the way which branch b1 actually behaved: if b1 is not taken then b2 is taken, indicating that the behaviors of b1 and b2 are correlated

Example2: This is from the eqntott benchmark (which is part of the SPEC 92 and SPEC 95 integer suites)

```
if (aa == 2)                /* branch b1 */
    aa = 0;                 /* not taken part */
if (bb == 2)                /* branch b2 */
    bb = 0;                 /* not taken part */
if (aa != bb) {             /* branch b3 */
    :                       /* not taken part */
}
```

– Assume that the three "if"s are implemented using three branch instructions b1, b2 and b3 as shown, and that the "then" clauses of the three "if"s are implemented in the fall–through part of the branches b1, b2 and b3.

– The possible control flow paths leading into the branch b3 and the values of the variables *aa* and *bb* just prior to processing b3 are:



Iter.	At start of iteration:			Path	Prediction	Actual	Counter after	Correct
#	aa	bb	counter	to b3	for b3	behavior	b3 is resolved	prediction?
0	2	2	0	nn	n	t	1	No
1	0	2	1	tn	n	t	2	No
2	2	1	2	nt	t	n	1	No
3	1	0	1	tt	n	n	0	Yes
4	2	2	0	nn	n	t	1	No
5	0	2	1	tn	n	t	2	No
6	2	1	2	nt	t	n	1	No
7	0	2	1	tn	n	t	2	No
8	1	1	2	tt	t	t	3	Yes
9	1	2	3	tn	t	n	2	No
10	0	2	2	tn	t	t	3	Yes
11	2	2	3	nn	t	t	3	Yes
12	1	2	3	tn	t	n	2	No
13	2	1	2	nt	t	n	1	No
14	0	1	1	tt	n	n	0	Yes
15	2	0	0	nt	n	t	1	No
16	2	1	1	nt	n	n	0	Yes
17	2	2	0	nn	n	t	1	No
18	2	2	1	nn	n	t	2	No
19	1	1	2	tt	t	t	3	Yes
20	0	2	3	tn	t	t	3	Yes
21	2	0	3	nt	t	t	3	Yes
22	1	2	3	tn	t	n	2	No
23	0	1	2	tt	t	n	1	No
24	2	2	1	nn	n	t	2	No

- Notice, however, the branch patterns – i.e., actual behavior of b3 (in execution order) along individual branch paths to b3:

  path nn: ttttt
  path nt: nnntnt
  path tn: tttntntn
  path tt: ntntn

- There is a more predictable branch behavior for b3 if we separate out the behavior of b3 based on the path leading to b3, as shown above

- Four different predictors for b3, one for each path into b3, can be used for this purpose. If four 2–bit saturating counters are used for each path, all initialized to 0, the number of correct predictions we get are:

  predictor for path nn: 4 correct predictions (counter values before execution of b3 on this path are: 0–1–2–3–3–3)

  predictor for path nt: 4 correct predictions (counter values before execution of b3 on this path are: 0–0–0–0–1–0)

  predictor for path tn: 3 correct predictions (counter values before execution of b3 on this path are: 0–1–2–3–2–3–2–3)

  predictor for path tt: 3 correct predictions (counter values before execution of b3 on this path are: 0–0–1–0–1)

  path nn: ttttt
  path nt: nnntnt
  path tn: tttntntn
  path tt: ntntn

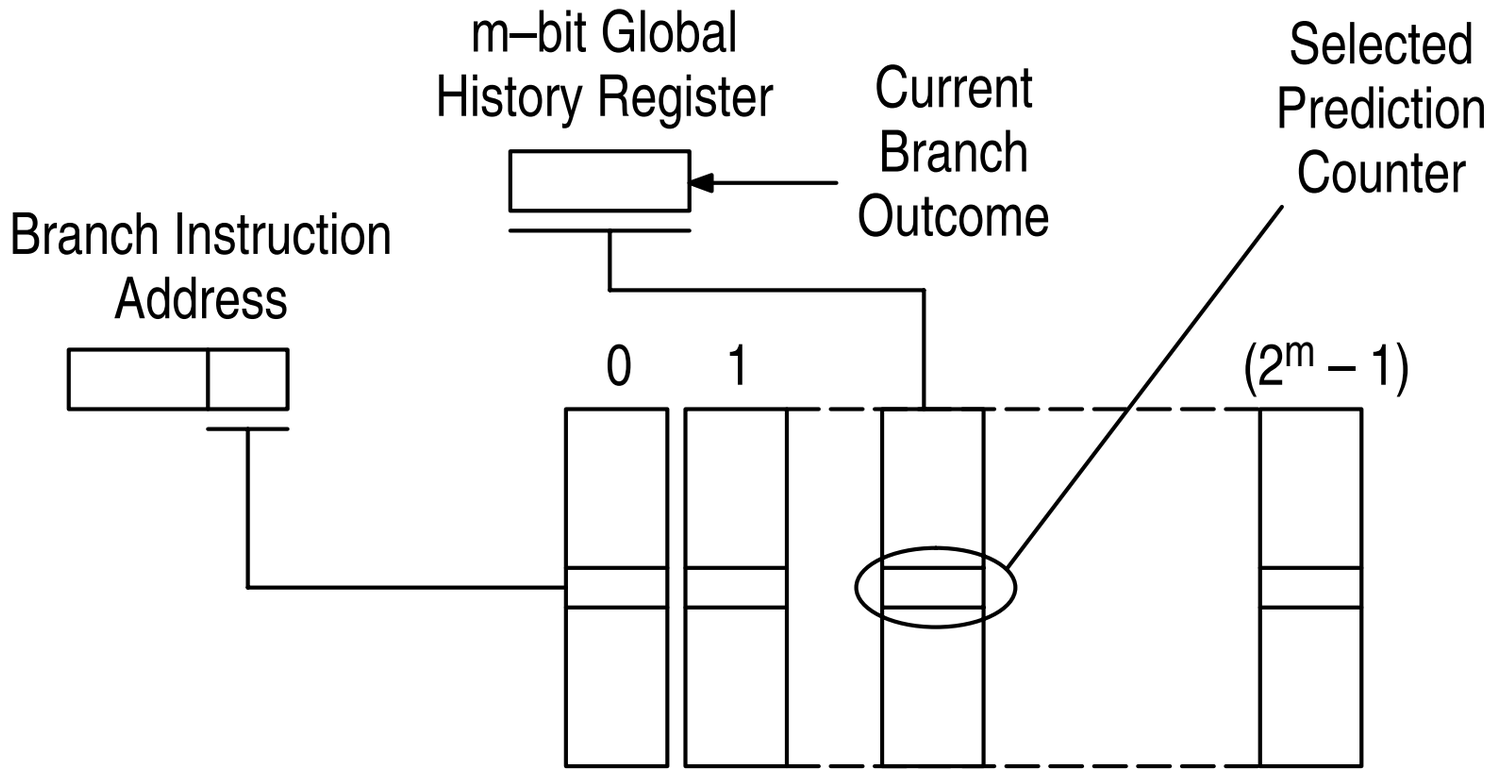
- This leads to a total of 14 correct predictions, improving the overall prediction accuracy to 56% (vs. 36% obtained from a single 2–bit saturating counter predictor for b3): this improvement comes about from taking into account the behavior of other branches on the way to b3.

- This leads to the main idea behind two–level branch predictors:

  – choose a per–branch instruction predictor based on how control arrived a branch predictor via other branches on the way

  – the latter information is global in nature and called the global branch history: it can be implemented as a shift register

- Implementation:



- m–bit global shift register can be used to select one of 2^m counter sub arrays.

- Lower order k–bits of branch instruction address can be used to select the relevant counter within the selected counter subarray.

- Alternatively, bits of the branch instruction's address and bits of the history register can be EX–ORed to locate the branch's predictor: **common in contemporary implementations**.

Storage requirements

- **m** global history bits
- **k** bits from address of branch
- **n** bits per local history counter
- **Memory size = $2^m * 2^k * n$**
- k=0 -- Global only, heavy sharing
- m=0 -- Local only, classic BTB

- Reported performance: For (8,2) configuration, 2–bit saturating counter (i.e., n = 2), with a total *unified* array size of 1 K Byte, running SPEC benchmarks on IBM RISC 6000–like system:

  – Prediction accuracy improves from about 82% for eqntott to 95% (classical, i.e., (0, 2) vs. (8, 2))

  – Improvement – but less dramatic – for other int benchmarks (up to 5.4%)

  – Little improvement for scientific benchmarks, as expected

- When only the global shift register is used to select the counter (i.e., k = 0), significant performance gains are obtained only when m is large (m = 15 boosts prediction accuracy for eqntott by 14.3% to nearly 97%)

- Other two–level predictors – see paper by Yeh & Patt: more complex design; no intuitive reasons for design

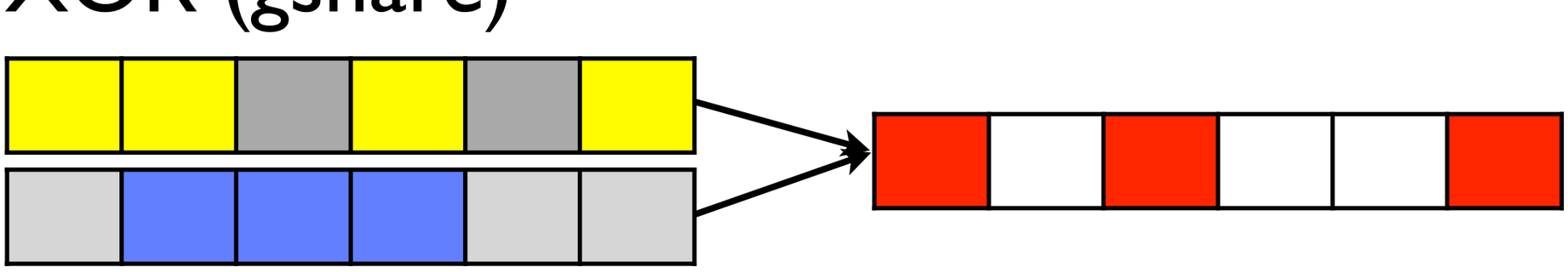
- Who uses 2–level branch prediction: Intel P6, Pentium IIs – but no details have been published!

Aliasing

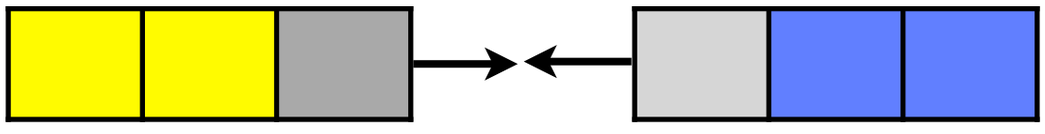
- Multiple branches use same table entry
  - Ex. Cause: Use lower-order bits of branch instruction address
  - Solution 1: Use more address bits

Aliasing

- Solution 2: Combine address and global history bits
  - XOR (gshare)

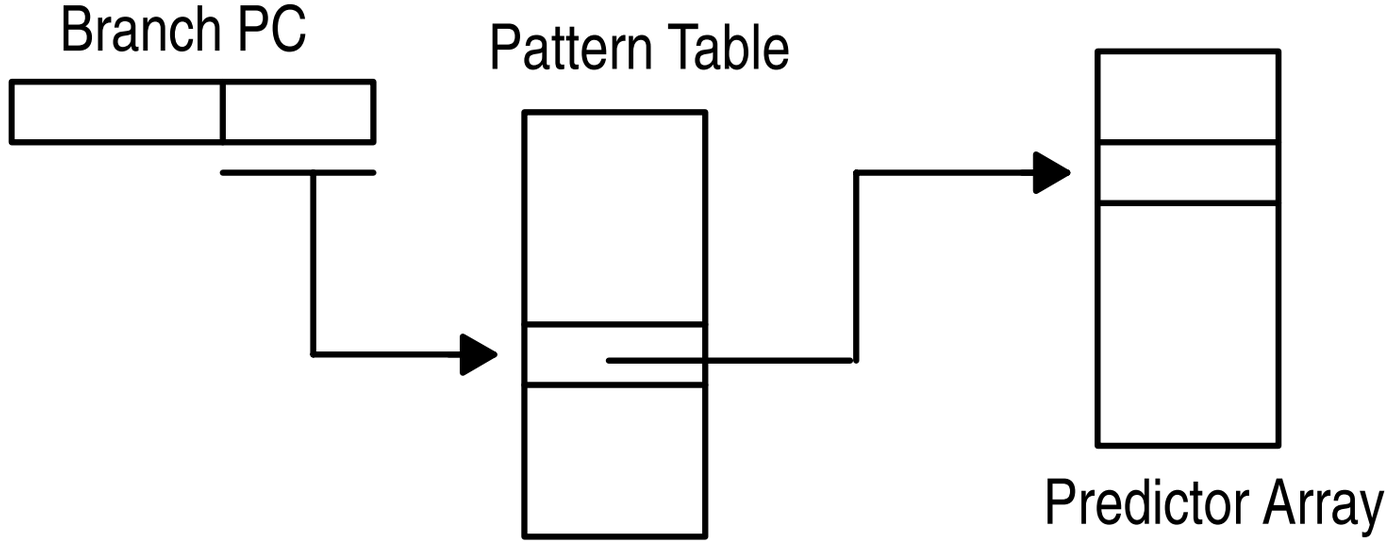


  - Concatenate (gselect)



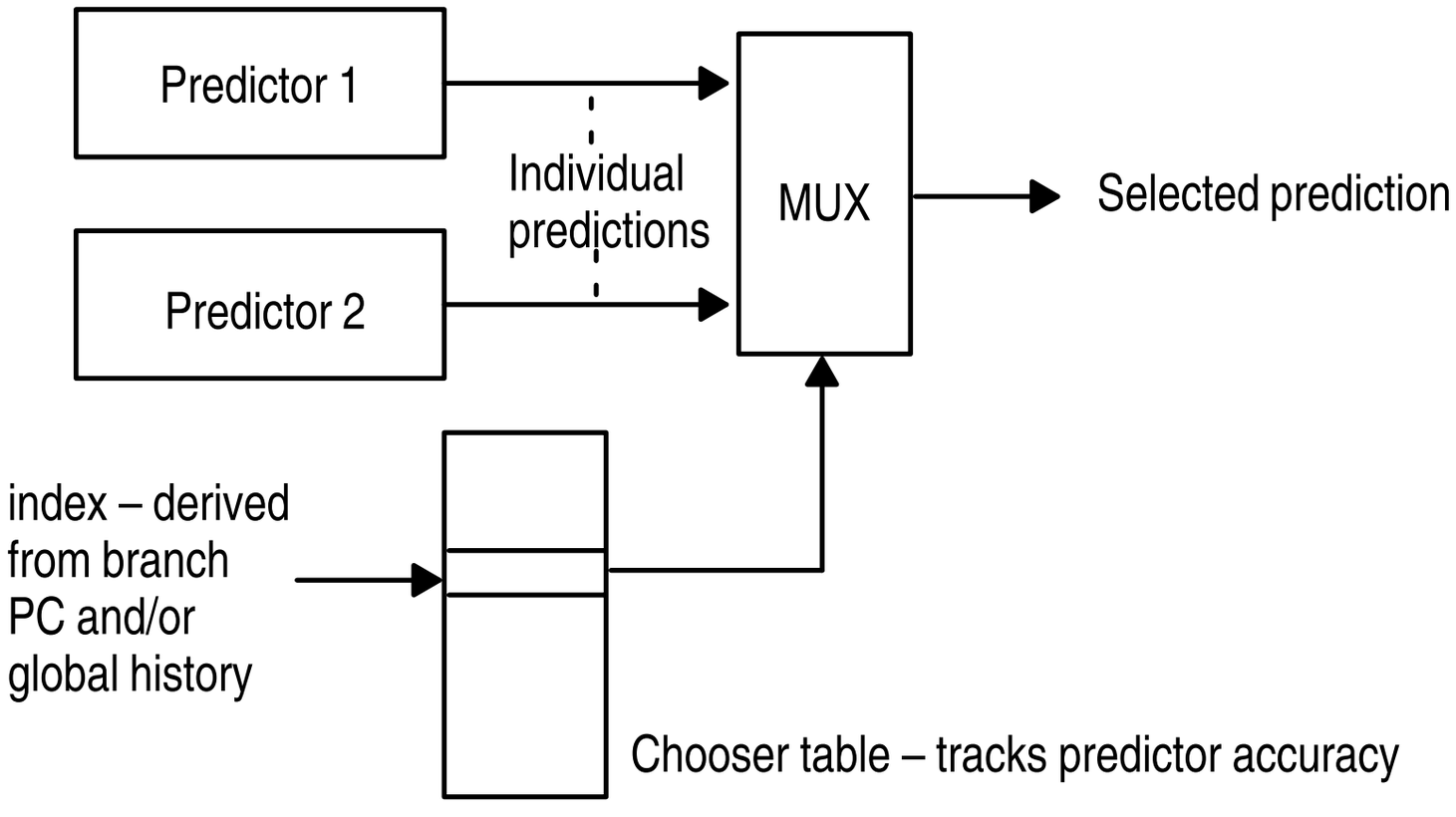
Aliasing

- Solution 3: Pattern table



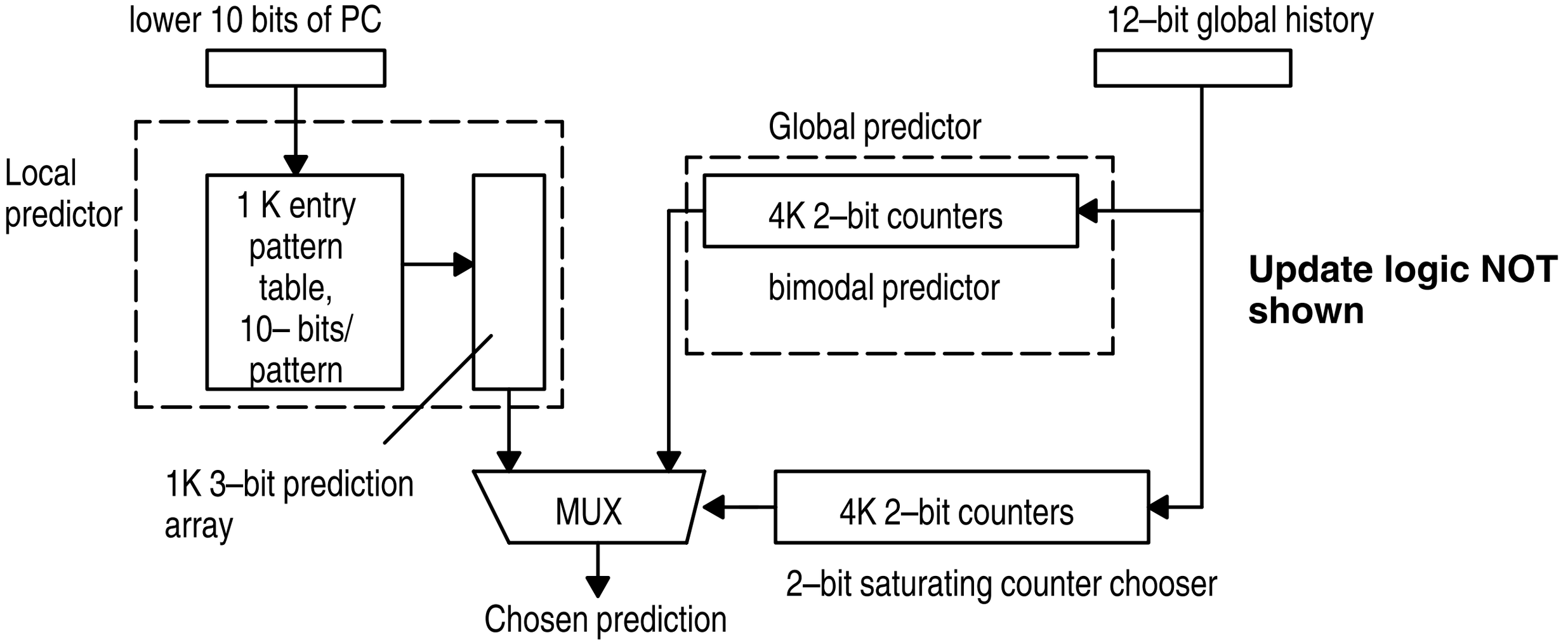
Tournament Predictor

- "Selective" or "Combined" predictor
- More than one predictor competes to more accurately predict each branch



- Chooser table could be an array of 2–bit saturating counter; recall that such counters are good at tracking consistency in behavior and forgiving momentary inconsistencies.
- ► Update a counter as follows:
  - if predictor 1 predicts correctly, and predictor 2 predicts incorrectly, decrement saturating counter
  - if predictor 2 predicts correctly, and predictor 1 predicts incorrectly, increment saturating counter
  - if both predict correctly or both predict incorrectly, do not update

Tournament Predictor



Tournament Predictor