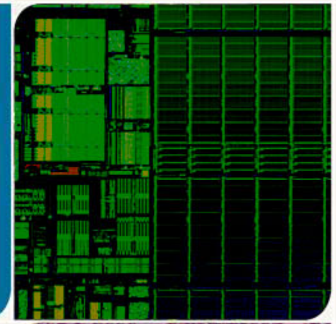
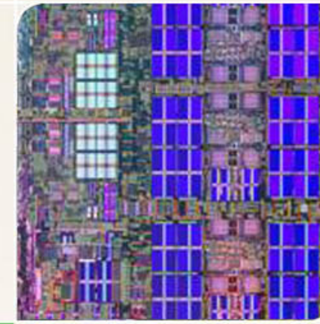
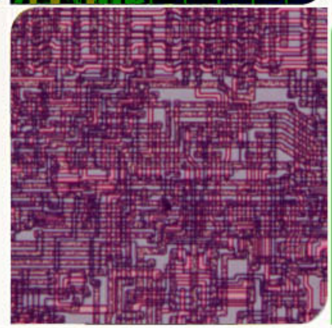
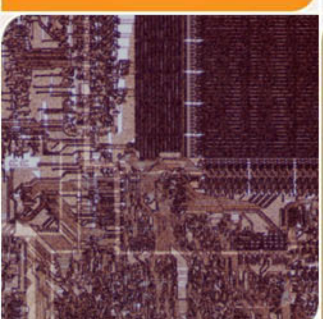






CS4290/CS6290

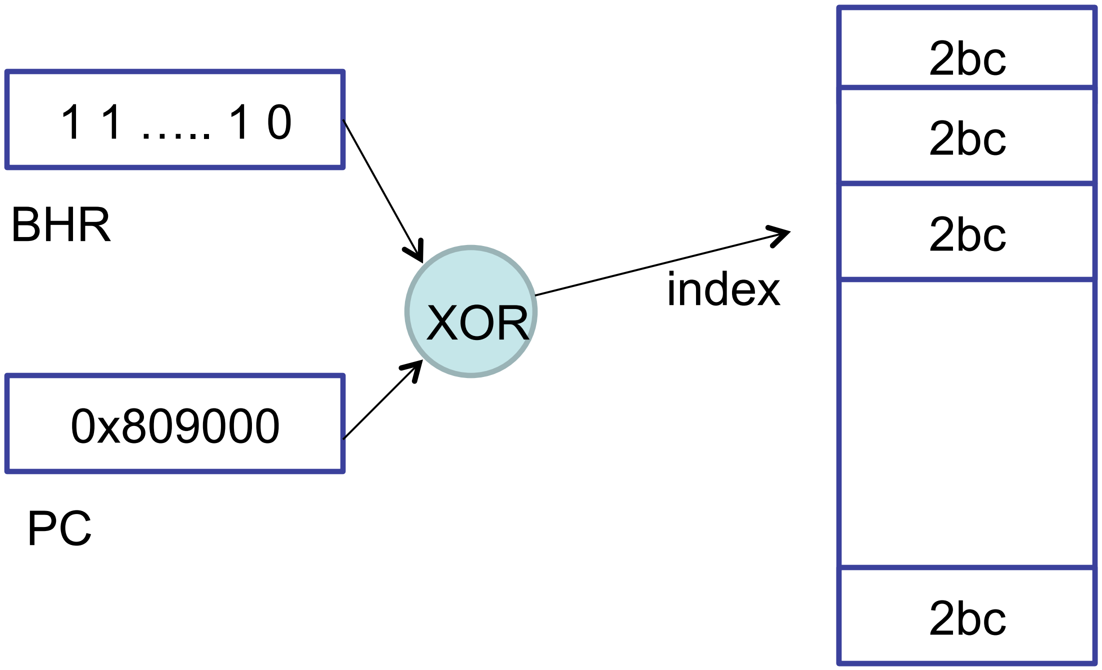
Fall 2011

Prof. Hyesoon Kim



Georgia Tech College of Computing

Gshare Branch Predictor



McFarling'93

Predictor size: 2^(history length)*2bit

G-SHARE Algorithms

```
predict_func(pc, actual_dir)
{
  index = pc xor BHR
   taken = 2bit_counters[index] > =2 ? 1 : 0
   correctly_predictied = (actual_dir == taken) ? 1 : 0  // stats
}

updated_func(pc, actual_dir)
{
  index = PC xor BHR
  if (actual_dir) SAT_INC( 2bit_counter[index] )
  else SAT_DEC ( 2bit_counter[index] )
  BHR = BHR << 1 | actual_dir
}
```

Exercise

There are three static branches, br1, br2, br3. dynamic branch trace is T,N,T,T,T,T,N,N,T (br1,br2, br3 is repeated three times). 3 bit BHR (start from 000).

PC addresses of the branches are 1,2,3 respectively. Calculate g-share branch predictor accuracy.

Init value of 2-bit counter is 2 (weakly taken)

Please turn in your solutions after the class.

When do we call branch update()?

- When do we know the branch outcome?
- Two options:
  – (1) After we know the actual branch outcome
  – (2) Speculatively update
- Pros: & Cons:
  – Think about deeper pipelines

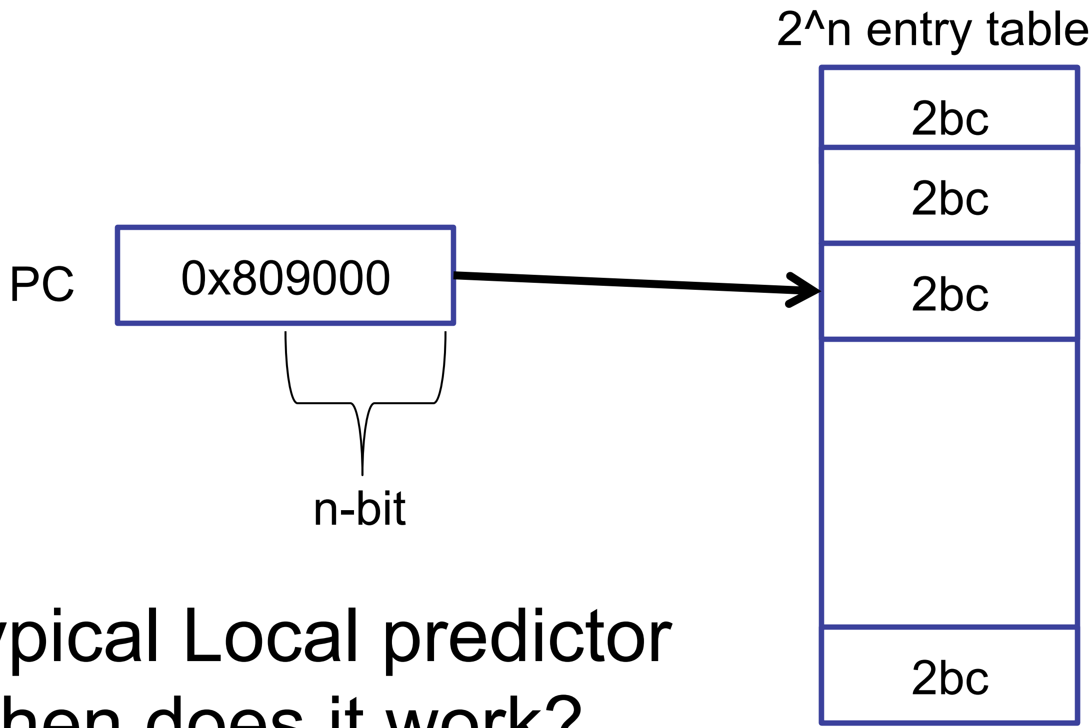
- How about prog assignment #2?

Global vs. Local Branch History

- Local Behavior
  - What is the predicted direction of Branch A given the outcomes of previous instances of Branch A?
- Global Behavior
  - What is the predicted direction of Branch Z given the outcomes of *all** previous branches A, B, …, X and Y?

* number of previous branches tracked limited by the history length

Biomodal Branch Predictor



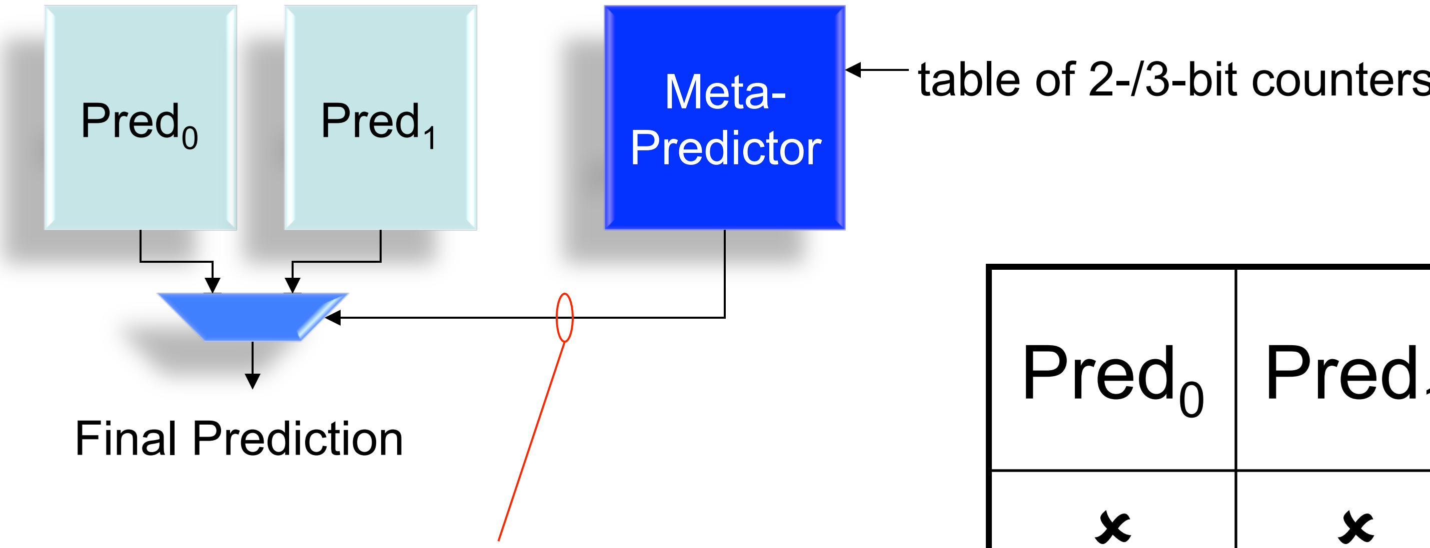
Typical Local predictor
When does it work?
-Loop,
-Repeat pattern

a++;
if (!(a%3)) { ..}

Tournament Predictors (Hybrid predictor)

- No predictor is clearly the best
  - Different branches exhibit different behaviors
    - Some "constant", some global, some local
- Idea:
  Let's have a predictor to predict
  which predictor will predict better ☺

Tournament Hybrid Predictors



If meta-counter MSB = 0, use $pred_0$ else use $pred_1$

$Pred_0$	$Pred_1$	Meta Update
✗	✗	
✗	✓	
✓	✗	
✓	✓	

Common Combinations

- Global history + Local history
- “easy” branches + global history
  - 2bC and gshare
- short history + long history

- Many types of behaviors, many combinations

Making Branches more Predictable

```
if ( t1 == 0 && t2 == 0 && t3 == 0) {
}
```

Hard to predict branches.

Anything can we do?

```
        if ( (t1 | t2 | t3)) == 0) {
        ….
        }
```

From “the software optimization cookbook” Intel

Branch Predictor Accuracy vs. Size

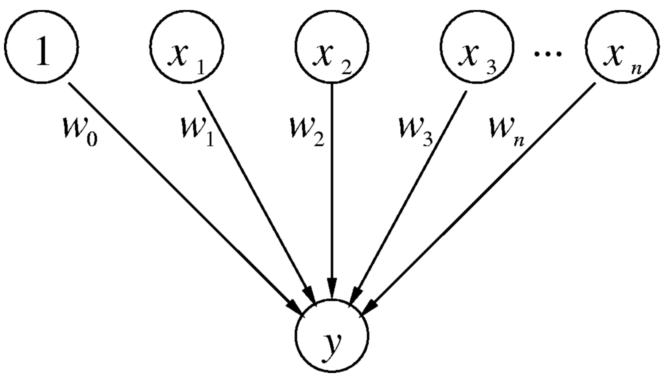
- Size of branch predictor:
  - Typically the size of PHT (Pattern History Table) (aka 2-bit counter table)
  - G-share: 2^n (n is the history length)
  - As n increases, accuracy?
  - Why?
- Downside of large size tables:
  - Longer to train
  - Long access time

Perceptron Predictor

- Use machine learning to train a branch predictor
- Outcome is not always taken or not-taken
- Train weight factors
- Requires much smaller storage
- Negative: complex calculation (solution: pipelining), linearly inseparable (solution: piece-wise linear predictor)

Branch-Predicting Perceptron

- Inputs (*x*'s) are from branch history and are -1 or +1
- *n* + 1 small integer weights (*w*'s) learned by on-line training
- Output (*y*) is dot product of *x*'s and *w*'s; predict taken if *y* ≥ 0
- Training finds correlations between history and outcome



$$y = w_0 + \sum_{i=1}^{n} x_i w_i$$

from D. Jimenez's slides

Training Algorithm

$x_{1..n}$ is the n-bit history register, x_0 is 1.
$w_{0..n}$ is the weights vector.
t is the Boolean branch outcome.
θ is the training threshold.

```
if |y| ≤ θ or ((y ≥ 0) ≠ t) then
    for each 0 ≤ i ≤ n in parallel
        if t = x_i then
            w_i := w_i + 1
        else
            w_i := w_i − 1
        end if
    end for
end if
```


Georgia Tech College of Computing

from D. Jimenez's slides

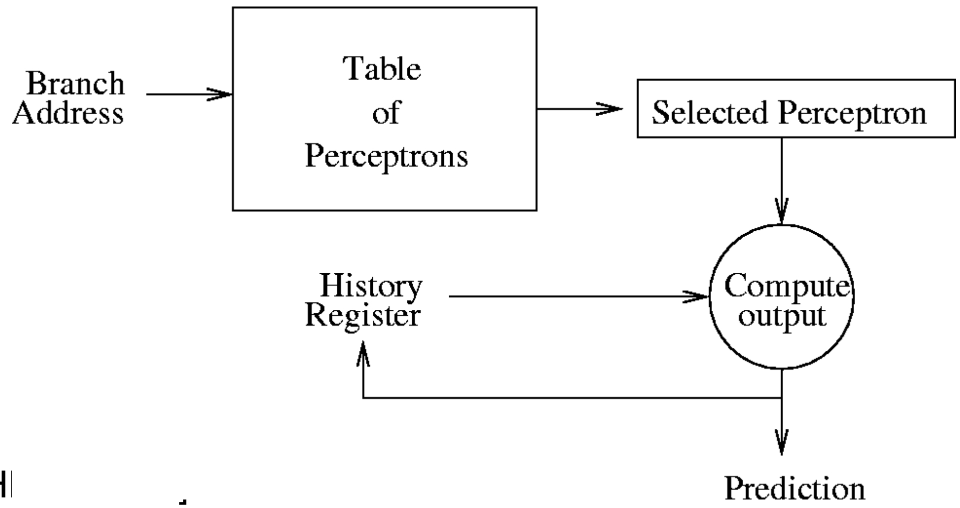
What Do The Weights Mean?

- The bias weight, w_0:
  - Proportional to the probability that the branch is taken
  - Doesn't take into account other branches; just like a Smith predictor
- The correlating weights, w_1 through w_n:
  - w_i is proportional to the probability that the predicted branch agrees with the i^{th} branch in the history
- The dot product of the *w*'s and *x*'s
  - $w_i \times x_i$ is proportional to the probability that the predicted branch is taken based on the correlation between this branch and the i^{th} branch
  - Sum takes into account all estimated probabilities
- What's θ?
  - Keeps from overtraining; adapt quickly to changing behavior

from D. Jimenez's slides

Organization of the Perceptron Predictor

- Keeps a table of *m* perceptron weights vectors
- Table is indexed by branch address modulo *m*



[Jiménez & Lin, H

from D. Jimenez's slides

Question

- How do we know when to access a branch predictor?

Target Address Prediction

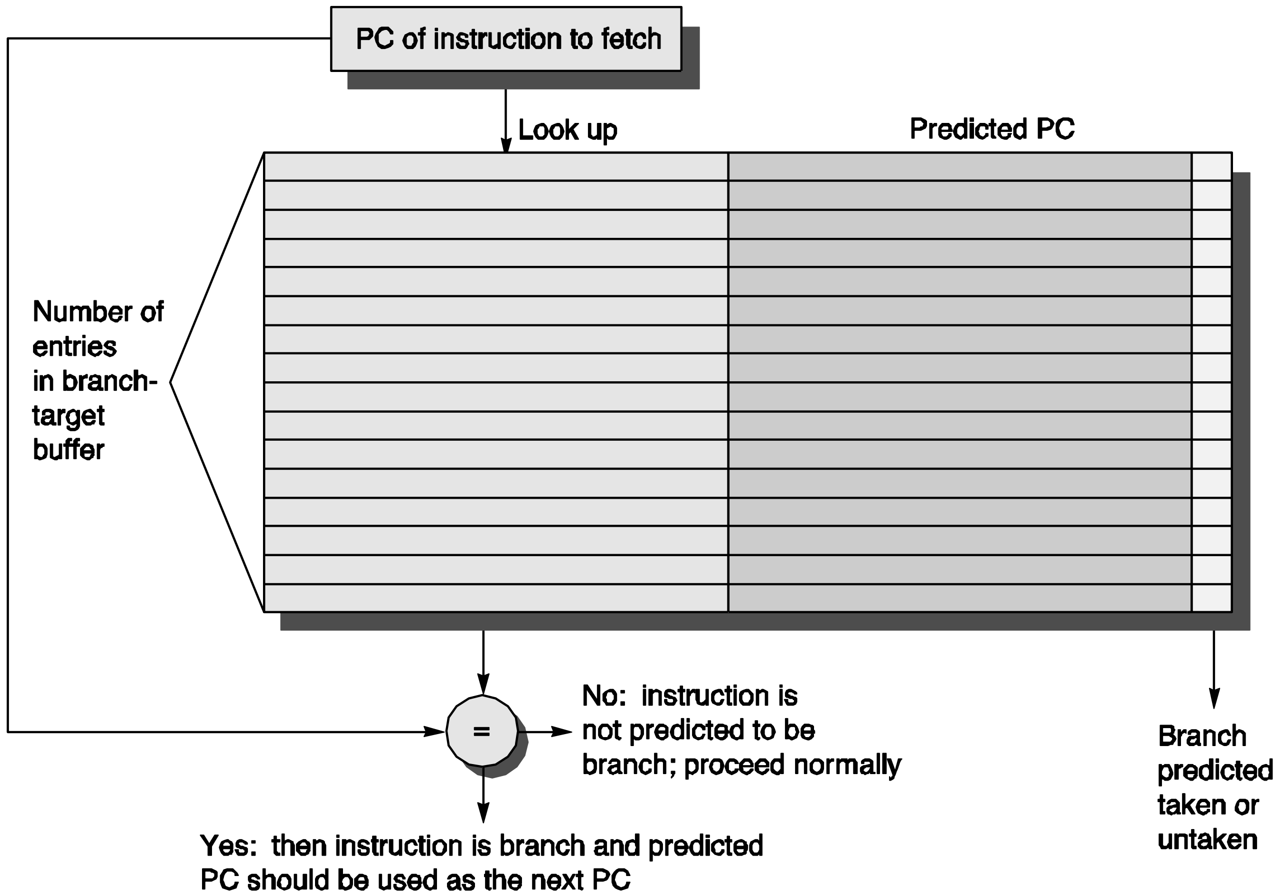
- Branch Target Buffer
  - IF stage: need to know fetch addr every cycle
  - Need target address one cycle after fetching a branch
  - For some branches (e.g., indirect) target known only after EX stage, which is way too late
  - Even easily-computed branch targets need to wait until instruction decoded and direction predicted in ID stage (still at least one cycle too late)
  - So, we have a fast predictor for the target that only needs the address of the branch instruction

Branch Target Buffer

- BTB indexed by instruction address (or fetch address)
- We don't even know if it is a branch!
- If address matches a BTB entry, it is *predicted to be a branch*
- BTB entry tells whether it is taken (direction) and where it goes if taken
- BTB takes only the instruction address, so while we fetch one instruction in the IF stage we are predicting where to fetch the next one from

Direction prediction can be factored out into separate table

Branch Target Buffer



Two Ways of Using BTB

- Target address != next PC address
  - (at least in this course and in the lab assignments)
  - Cond. Br TARGET
  - Br is taken next PC = TARGET
  - Br is not-taken next PC = current PC + Inst size
- (1) BTB stores target address:
  - Direction prediction?
- (2) BTB stores next PC addresses

BTB entry

- When do we have more than one target address for one BTB entry?
  - Return
  - Indirect branches
  - BTB is indexed with fetch address
  - Fetch address ?
    - When a processor fetches more than one instruction, it fetches a cache block. BTB is often indexed with the cache block address.
    - X86 software optimization manual: Do not put branches too nearby

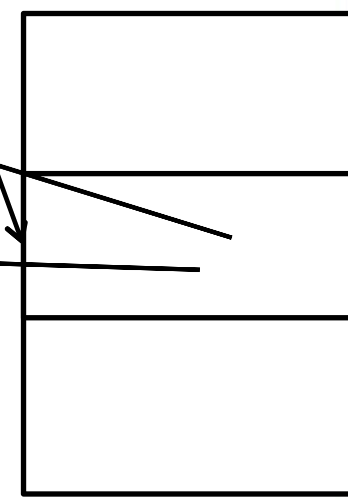
Function Calls

```
main()
{
    foo();
    printf("still hungry\n");
    ....
    foo();
    printf("full\n");
}
```

```
foo(){
  .....
  return
}
```

??

BTB



Return Address Stack (RAS)

- Function returns are frequent, yet
  - Address is difficult to compute
    (have to wait until EX stage done to know it)
  - Address difficult to predict with BTB
    (function can be called from multiple places)

Function Calls

```
         main()
         {
0x800      foo();
0x804      printf("still hungry\n");
           ....
0x900      foo();
0x904      printf("full\n");
         }
```


```
foo(){
.....
   return
   }
```

Return Address Stack (RAS)

- But return address is actually easy to predict
  - It is the address after the last call instruction that we haven't returned from yet
  - Hence the Return Address Stack

Function Calls

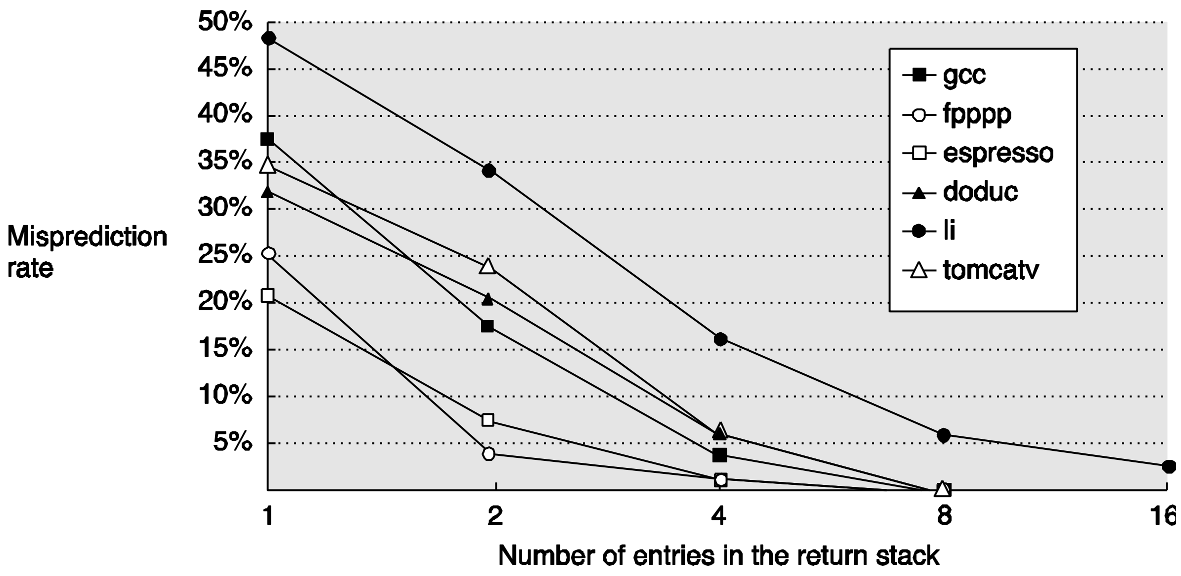
```
main()
{
0x800   foo();
0x804   printf("still hungry\n");
        ....
0x900   foo();
0x904   printf("full\n");
}
```

```
foo(){
   .....
   return
}
```

0x804 / 0x904

Return Address Stack (RAS)

- Call pushes return address into the RAS
- When a return instruction decoded,
  pop the predicted return address from RAS
- Accurate prediction even w/ small RAS



Code Optimization vs. RAS

- Now you learned RAS, what do you do to write a program to improve performance?
  - Match function calls & returns
  - Do not overflow return address stack (depth is limited.)

Review

- G-share predictor
- RAS (Return Address Stack)
- Updating branch predictor

LOOP Branches

```
for (ii =0; ii < 10; ii++)
{
 …
}
```

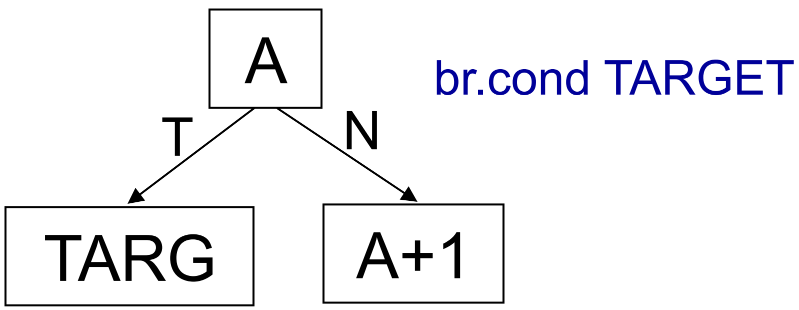
Loop branch is iterated 10 times all the time

- Special treatment for loop branches
- Why do we want loops specially?
  - Easy to predict if we know N
  - Easy to know in advance if we know N
  - Pollute branch predictor

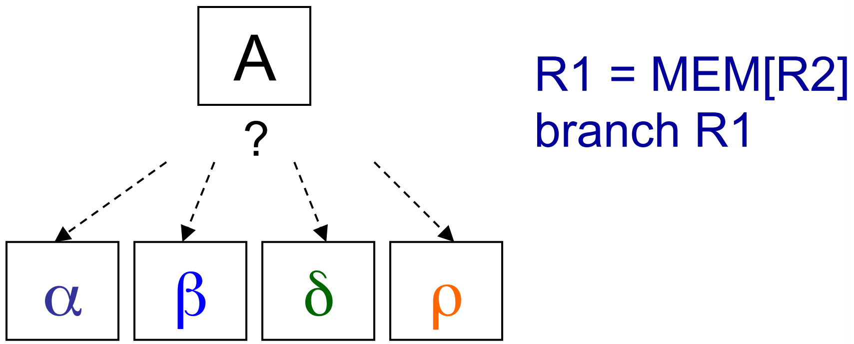
Other Options

- Prepare to branch (HPL-PD)
  - Software gives hints to the hardware about what the branch target will be. It saves us the target prediction since it has already been written into one of the target registers.
  - Works when?
- Special Loop predictor (Intel's Pentium M)
  - Detect a loop branch
  - Train the max iteration counter value

Direct vs. Indirect Branch



Conditional (Direct) Branch



Indirect Branch

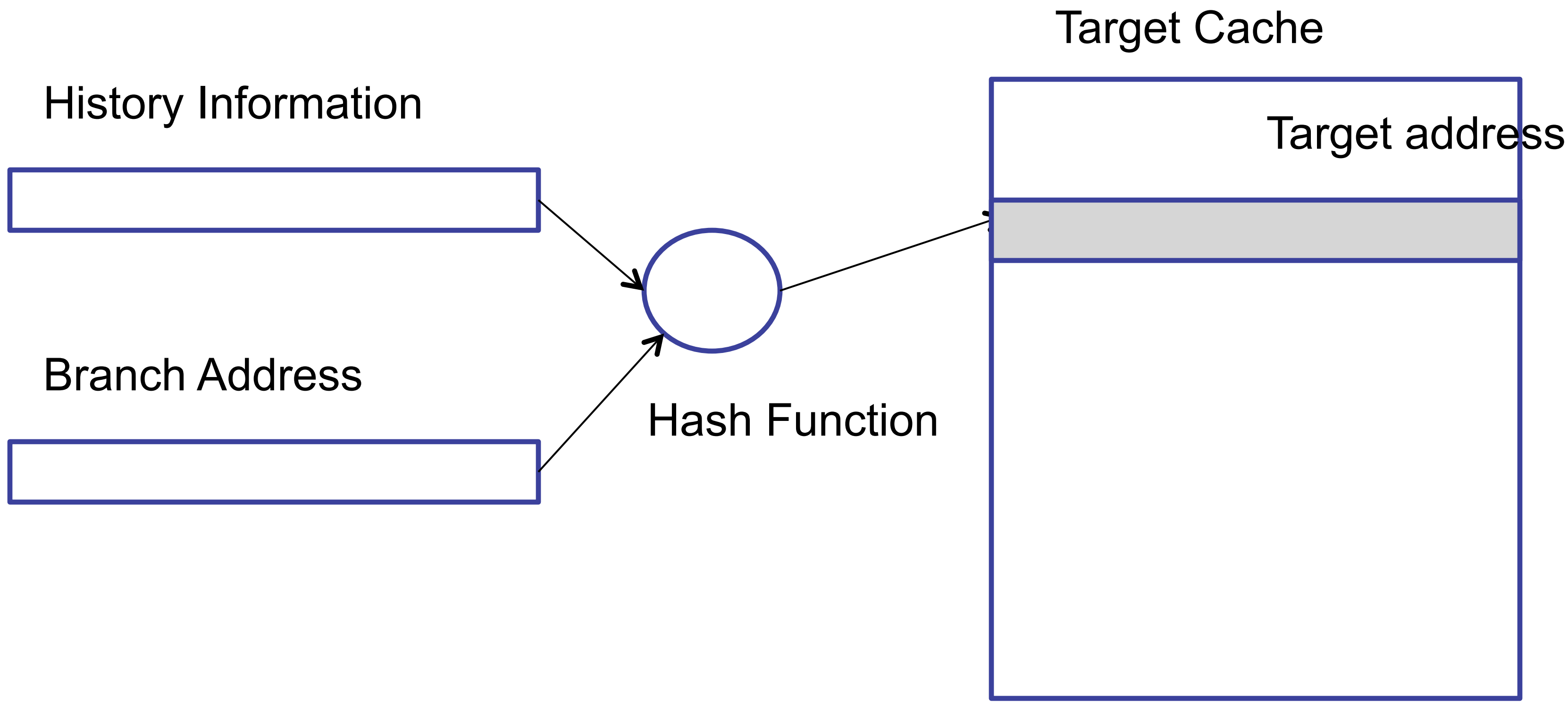
- Use the BTB
- A special indirect branch predictor (Intel's Core-2)

Indirect Branch Code Examples

- Switch statements
  - -few cases: a chain of conditional branches
- Virtual functions

Indirect Branch Predictors

- Tagged Target Cache (Chang'97)



Predicting Different Types of Branches

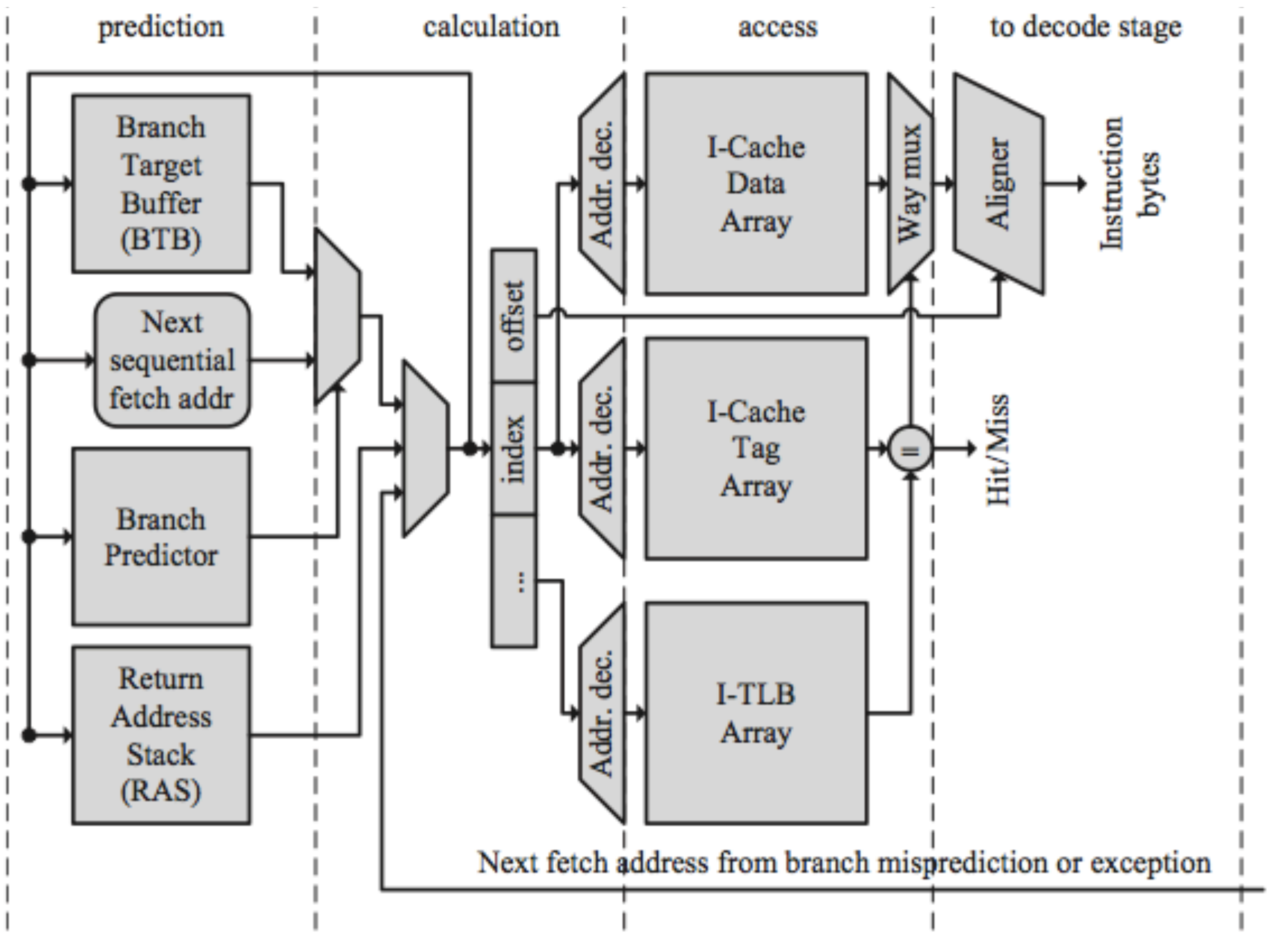
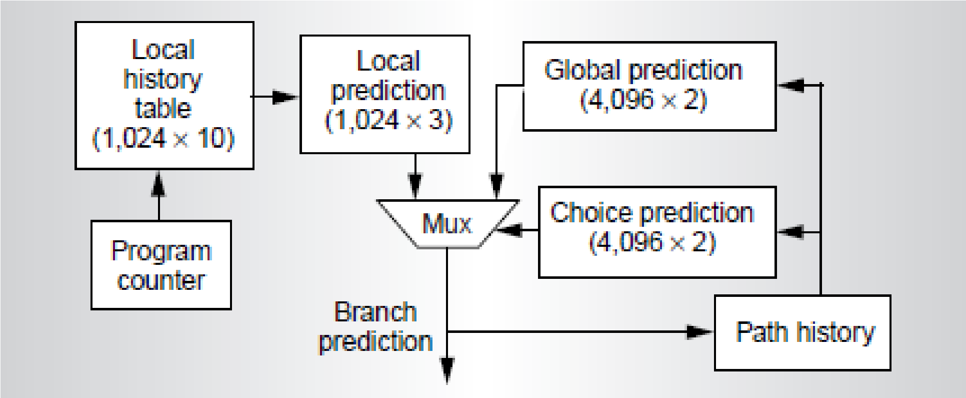


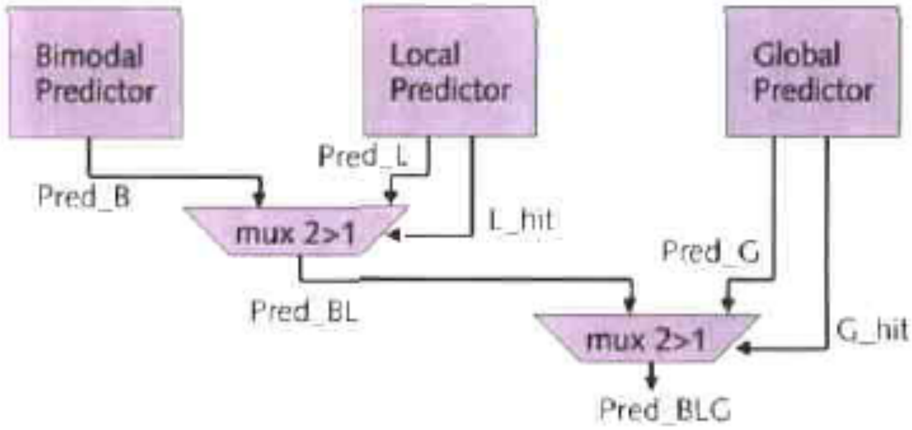
FIGURE 3.1: Example fetch pipeline

Example 1: Alpha 21264



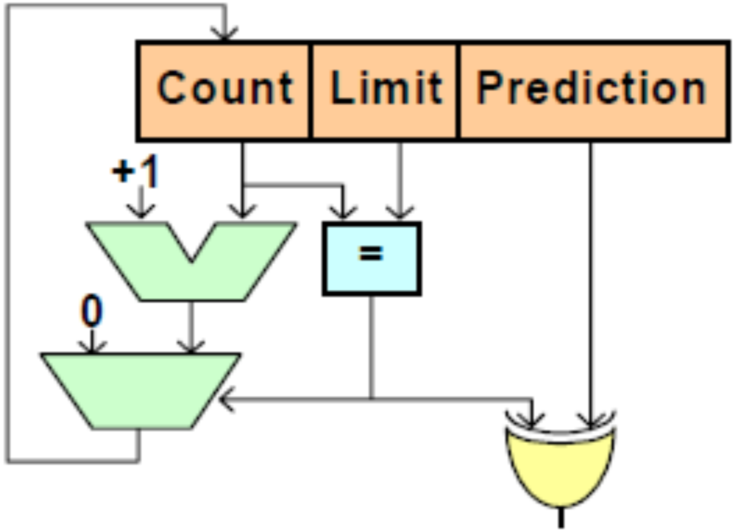
- Hybrid predictor
  - combines local history and global history components with a meta-predictor

Example 2: Pentium-M



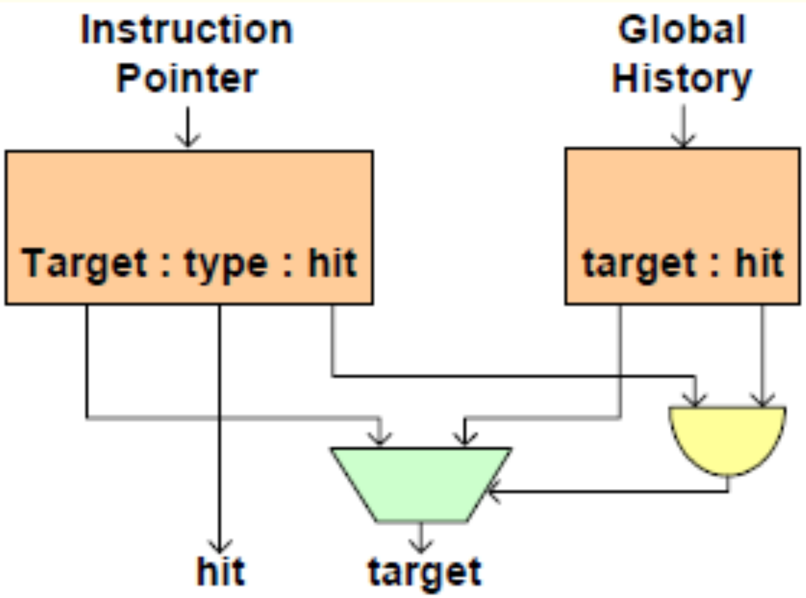
- Also hybrid, but uses tag-based selection mechanism

Pentium-M (cont'd)



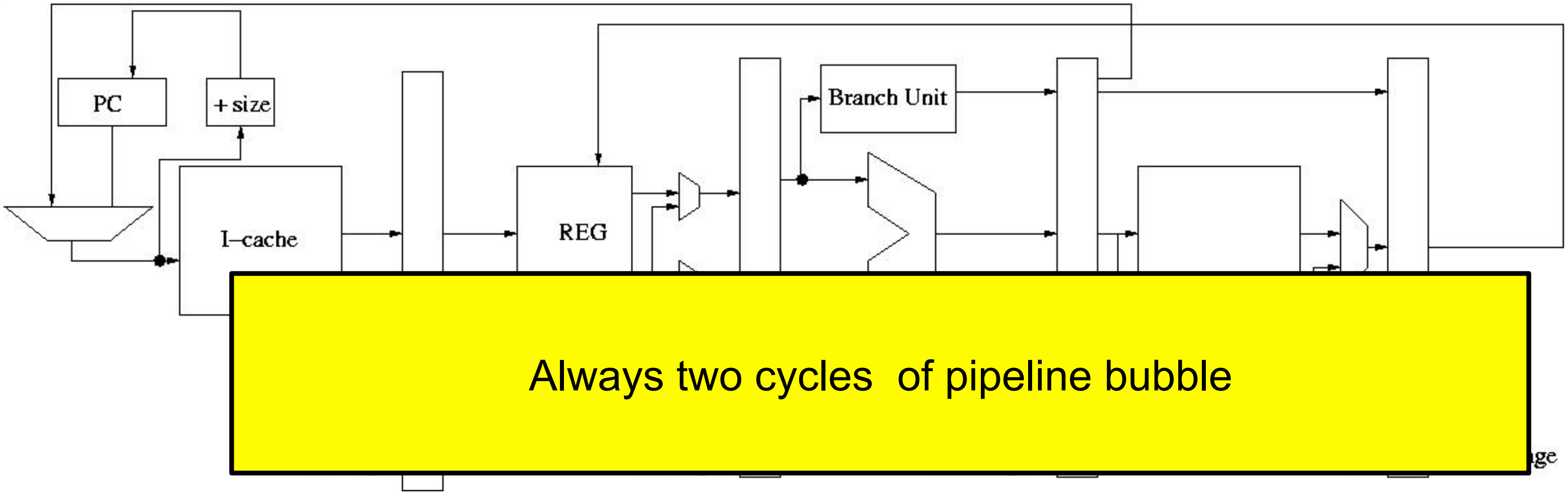
- Local component also has support for loops
  - accurately predict branches of the form $(T^kN)^*$

Pentium-M (cont'd)



- Special target prediction for indirect branches
  - common in object-oriented code (vtables)
  - assumes correlation with global history

Handling Branches (from the last lecture)



Always two cycles of pipeline bubble

cycle	PC (latch)	FE	ID	EX	MEM	WB
1	0x800	**br**				
2	0x804	add	**br**			
3	0x804	add		**br**		
4	0x900	sub			**br**	
5	0x904	add	sub			**br**
6	0x908	mul	add	sub		

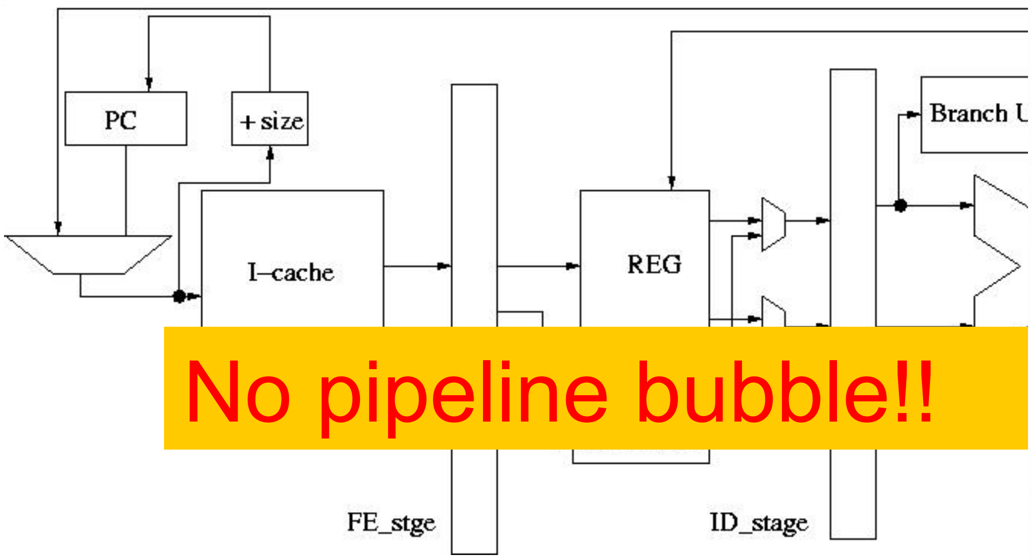
What if we

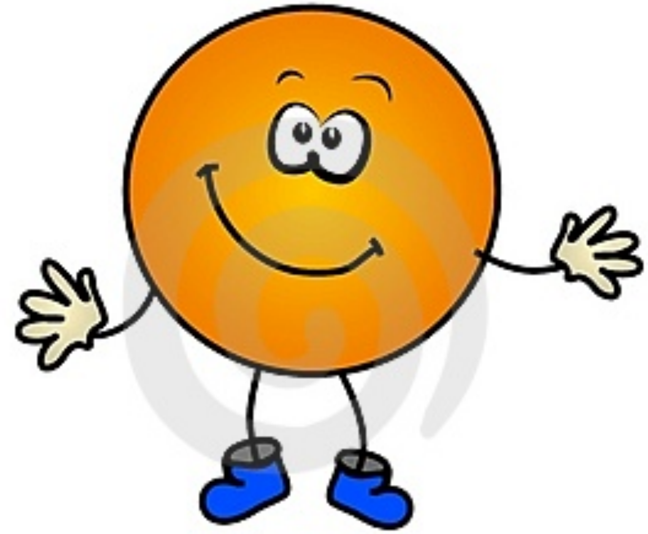
```
0x800          sub r1, r2,r3
0x804          add r4, r2,r3
0x808          br     target
0x80b
0x810
0x900 target mul r2,  r3,r4
```

```
0x900 target mul r2,  r3,r4
```

Change the rule!
Always execute the next two instructions after a branch

Rule: Always execute the next two instructions after a branch





cycle	Fetch addr	FE	ID	E		
1	0x800	br				
2	0x804	sub	br			
3	0x808	add	sub	br		
4	0x900	mul	add	sub	br	
5	0x904	div	mul	add	sub	br
6	0x908	add	div	mul	add	sub
7	0x90b	sub	add	div	mul	add

Delayed branch

- N-cycle delay slot
- The compiler fills out useful instructions inside the delay slot
- Different options:
  - Fill the slots instructions fro either taken or not-taken: When a branch is executed in other way, flush!

Remark

- Many DSP architecture, older RISC, MIPS, PA-RISC, SPARC.
- Delayed branches are architecturally ~~invisible~~ visible
  - Advantage:
    - better performance
  - Disadvantage:
    - what if implementation changes?
    - Deeper pipeline-> more branch delays?
- Interrupt/exceptions?
  - Where to go back?
- Combining with a branch predictor?

High bandwidth Instruction Supply: Trace Cache

I1

I2 br T1

I3

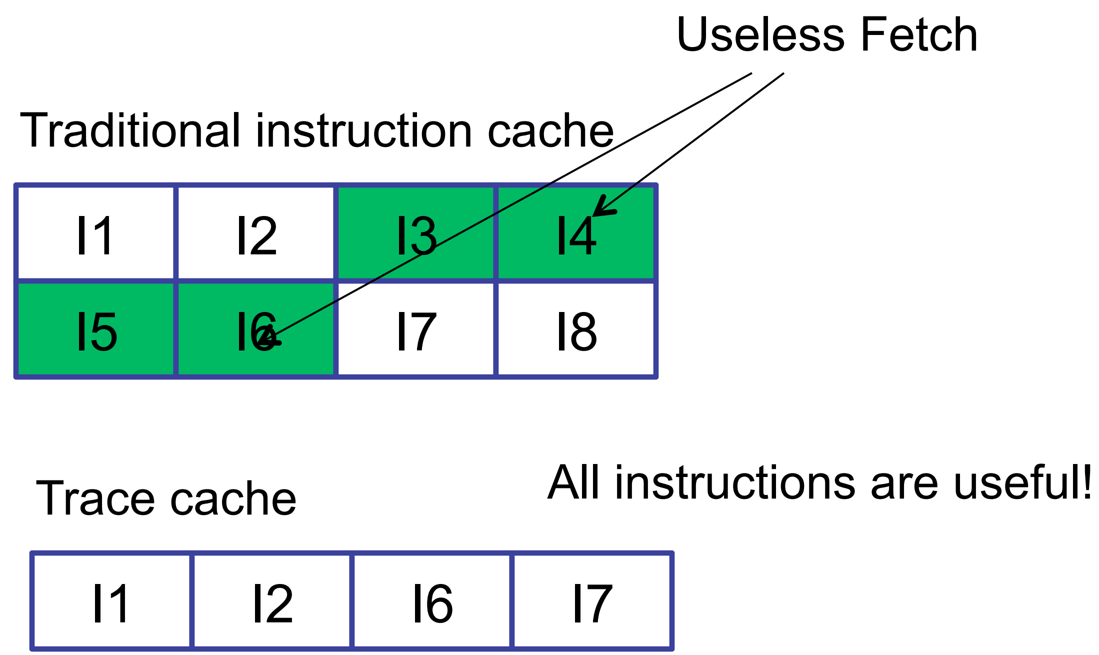
I4

I5

T1: I6

I7 br T2

T2: I13



Trace Cache

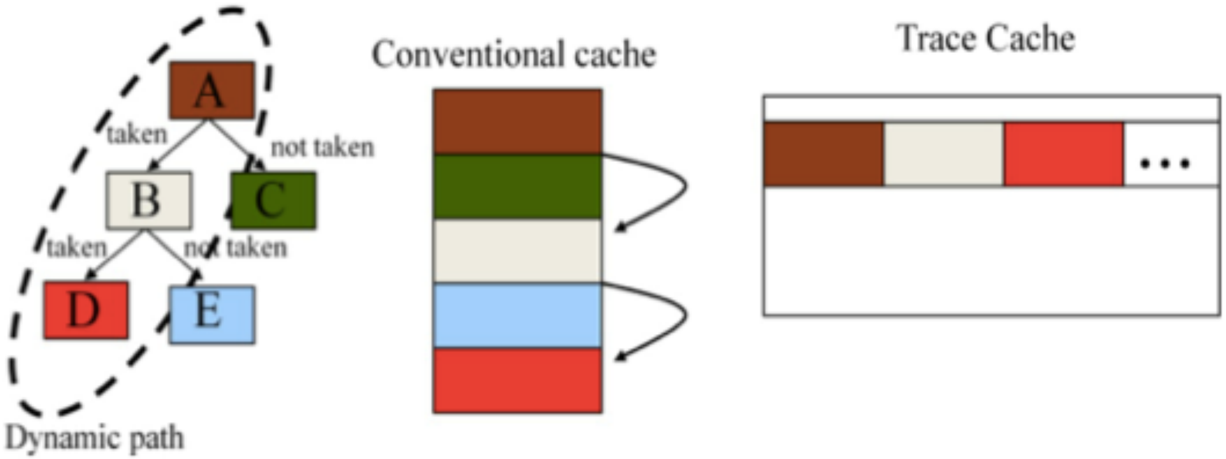
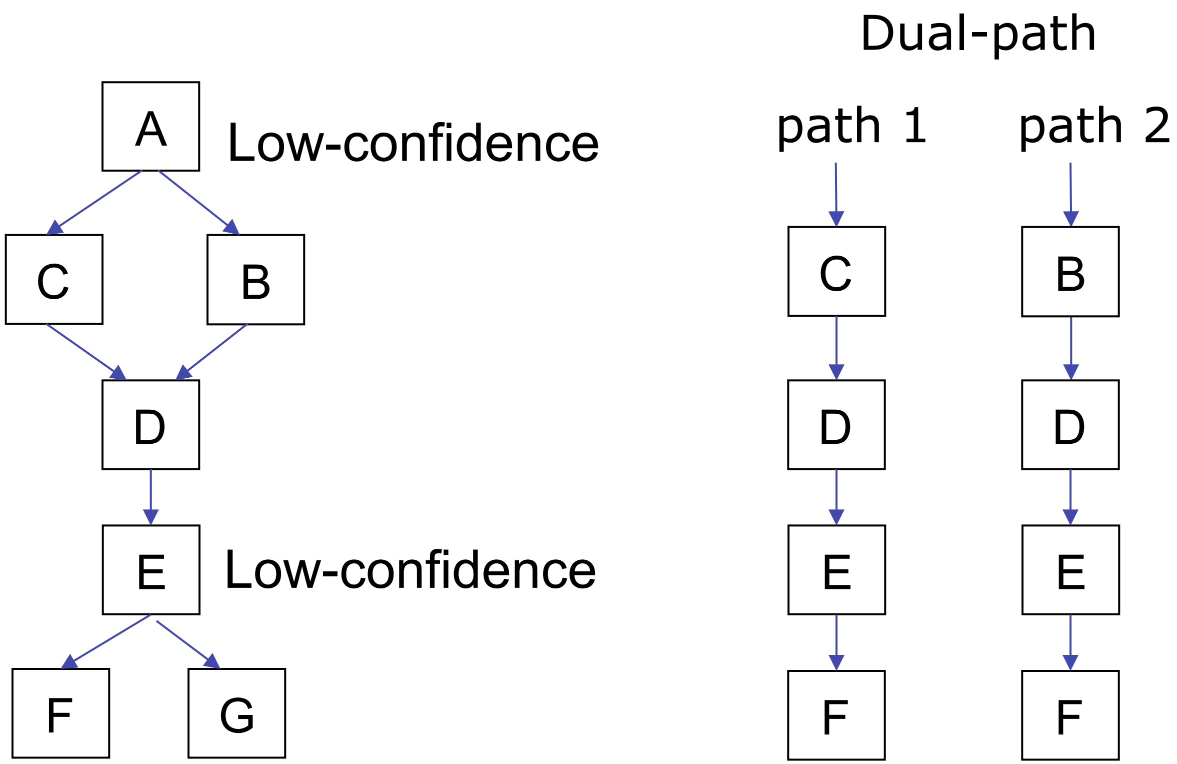


FIGURE 3.2: Conventional instruction cache and trace cache overview.

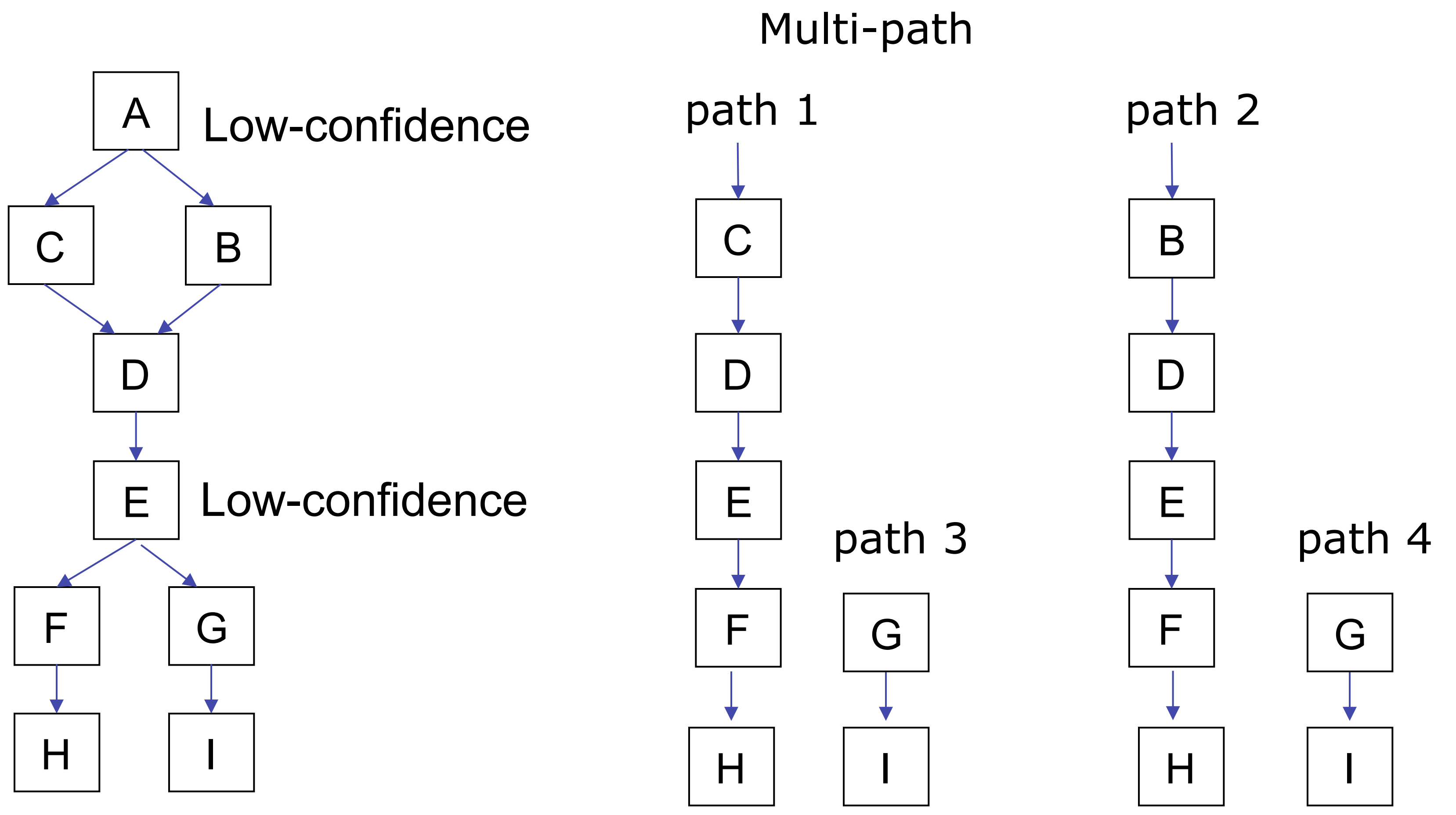
Avoiding Branch Prediction

- Dual-path execution
  - When you see a low-confidence branch, start to fetch from only two paths
  - See another low-confidence branch?
    - Ignore and just keep only two paths
- Multi-path execution
  - Whenever it sees a low-confidence branch, forks

Dual-path Execution



Multi-path Execution



Prog #2

- G-share branch predictor
- Deeper pipeline
- No test case will be provided. Solve the report questions: That will help you debug.

Loop Invariant Branches

```
for (i=0;i<10;i++) {
  if (cond1) stat1
  else if (cond 2) stat2
  else stat3
}
```

Can we optimize this code?