Chapter 3 (CONT)

Instructor: Josep Torrellas

CS433

Dynamic Hardware Branch Prediction

- Control hazards are sources of losses , especially for processors that want to issue > 1 instr / cycle
- Proposed approach : use HW to dynamically predict the outcome of a branch (may change with time)

1 Branch prediction buffer (a.k.a. branch history table)
   - Small memory indexed by lower bits of addr of branch instruction
   - Contains 1 bit that says if branch was recently taken

Note : the prediction may refer to another branch that has some low-order address bits

- If hint is wrong : prediction bit is inverted
- Problem : if branch is taken 9 times in a row in a loop we have two mispredictions

2 Two-bit prediction schemes

- Need to mispredict twice before changing the predict

See Figure C.18

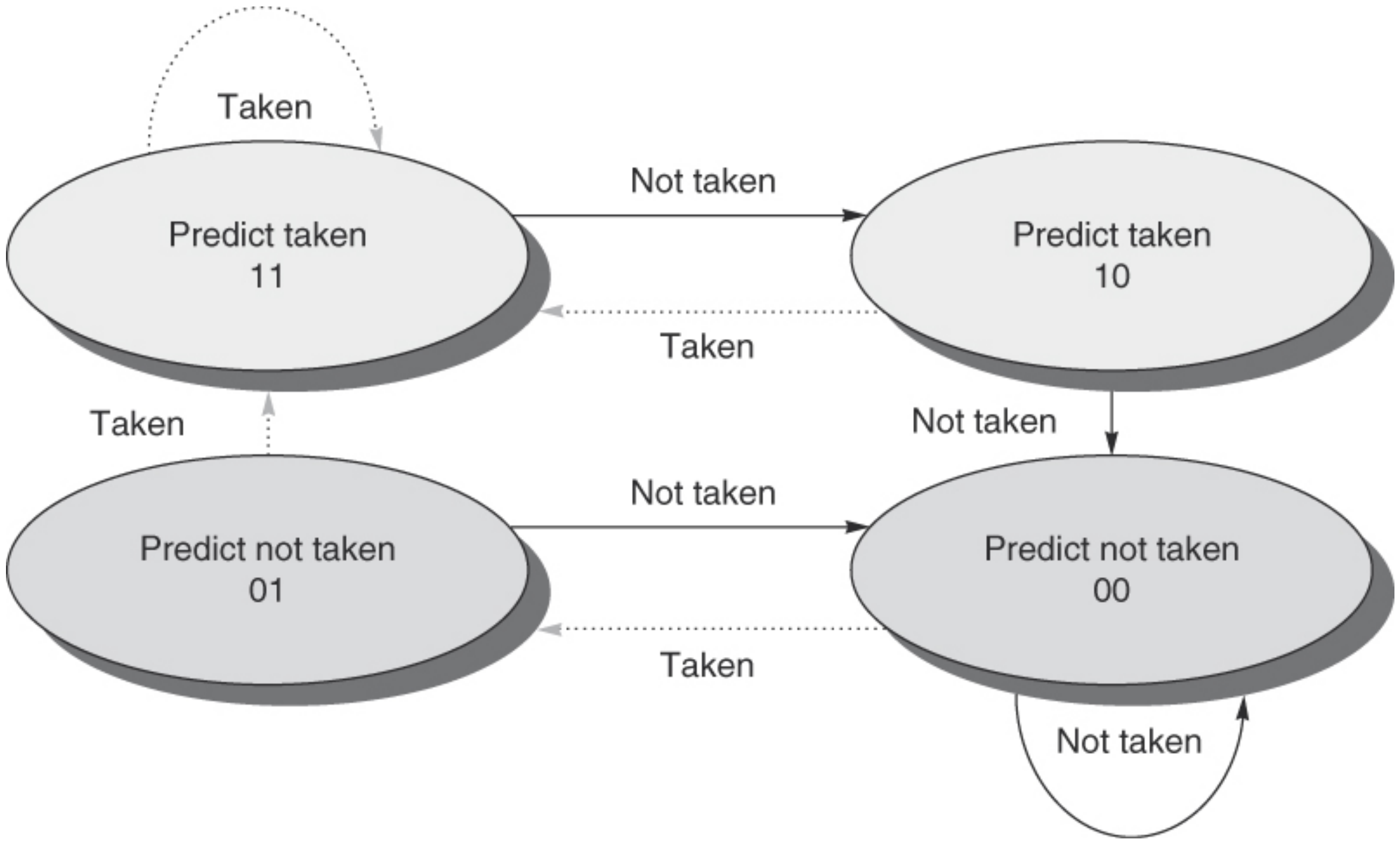

Taken
Not taken
Predict taken
11
Predict taken
10
Taken
Taken
Not taken
Not taken
Predict not taken
01
Predict not taken
00
Taken
Not taken

Figure C.18

3. N-bit saturating counter

- n bit counter can take from 0 to $2^n - 1$
- if count $\geq 2^{n-1}$ branch predict taken
  else predict untaken
- if taken, increment ; if untaken , decrement

Accuracy : 4096 entries in table, 2 bits

$\rightarrow$ misprediction rate $\approx$1 - 18% in spec89

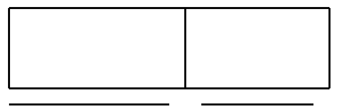
$\rightarrow$ usually better at FP programs (more loops)

4 Look at recent behavior of other branches too

“correlating predictors or two-level predictors”

```
 if ( d == 0)
         d =1 ;
if (d == 1)
```

e.g scheme where each branch has 2 separate bits


prediction used if the last branch not taken
prediction used if the last branch taken

example $\rightarrow$ works very well for b2

→ correct prediction for b1 is by chance

b2 will always work (every execution) except if d=1

This is called a (1 , 1) predictor

last ↓ branch   choose among ↓ 2 states (1 bit)

Can be (m , n)

look at ↓ last m branches   use n bits ↓ to predict

} Easy hardware to remember outcome of last m branches

- A two bit predictor with no global history is a (0,2) predictor

# bits in an (m,n) predictor ?

2^m * n * #entries

e.g (2,2)

Tournament Predictors

- Use multiple predictors, usually
  - One based on global info
  - One based on local info
  - Combining them with a selector
- They do very well
- They are a popular form of Multilevel branch predictors (use several levels of branch prediction tables together with an algorithm for choosing among them)
- Existing ones: use a 2-bit saturating counter per branch to choose among two different predictors (the four states of the counter dictate whether to use predictor 1 or 2)

Tournament Predictors

- The counter is incremented whenever the predicted predictor is correct and the other is incorrect, and is decremented in the reverse situation
- Advantage: ability to select the right predictor for right branch
- Figure 3.3: fraction of predictions that use the local predictor
- Figure 3.4: comparing local/global/tournament

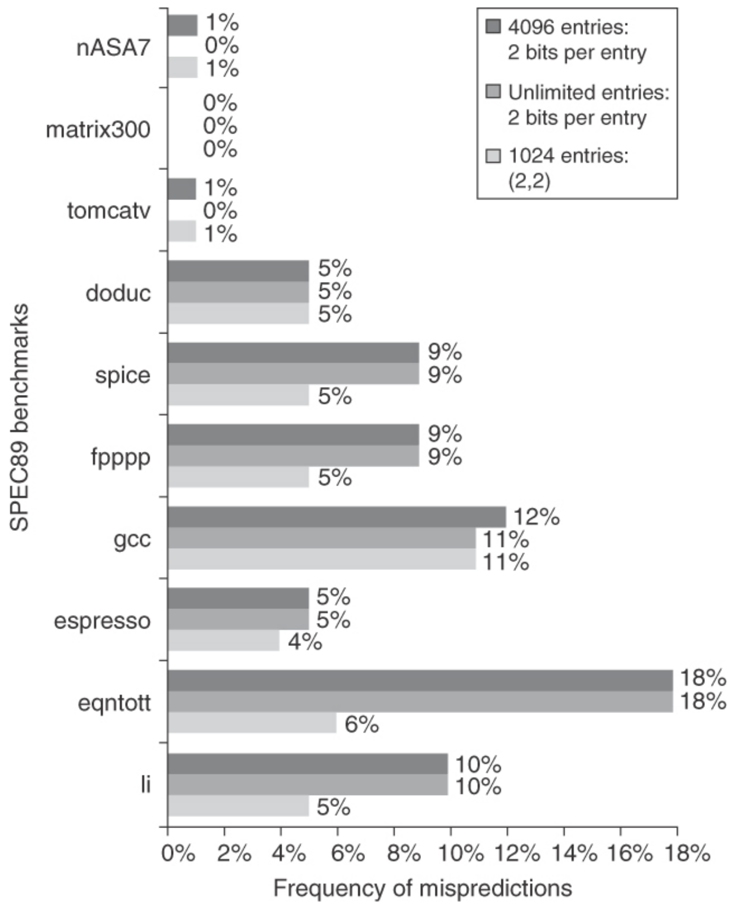


Figure 3.3 Comparison of 2-bit predictors. A noncorrelating predictor for 4096 bits is first, followed by a noncorrelating 2-bit predictor with unlimited entries and a 2-bit predictor with 2 bits of global history and a total of 1024 entries. Although these data are for an older version of SPEC, data for more recent SPEC benchmarks would show similar differences in accuracy.

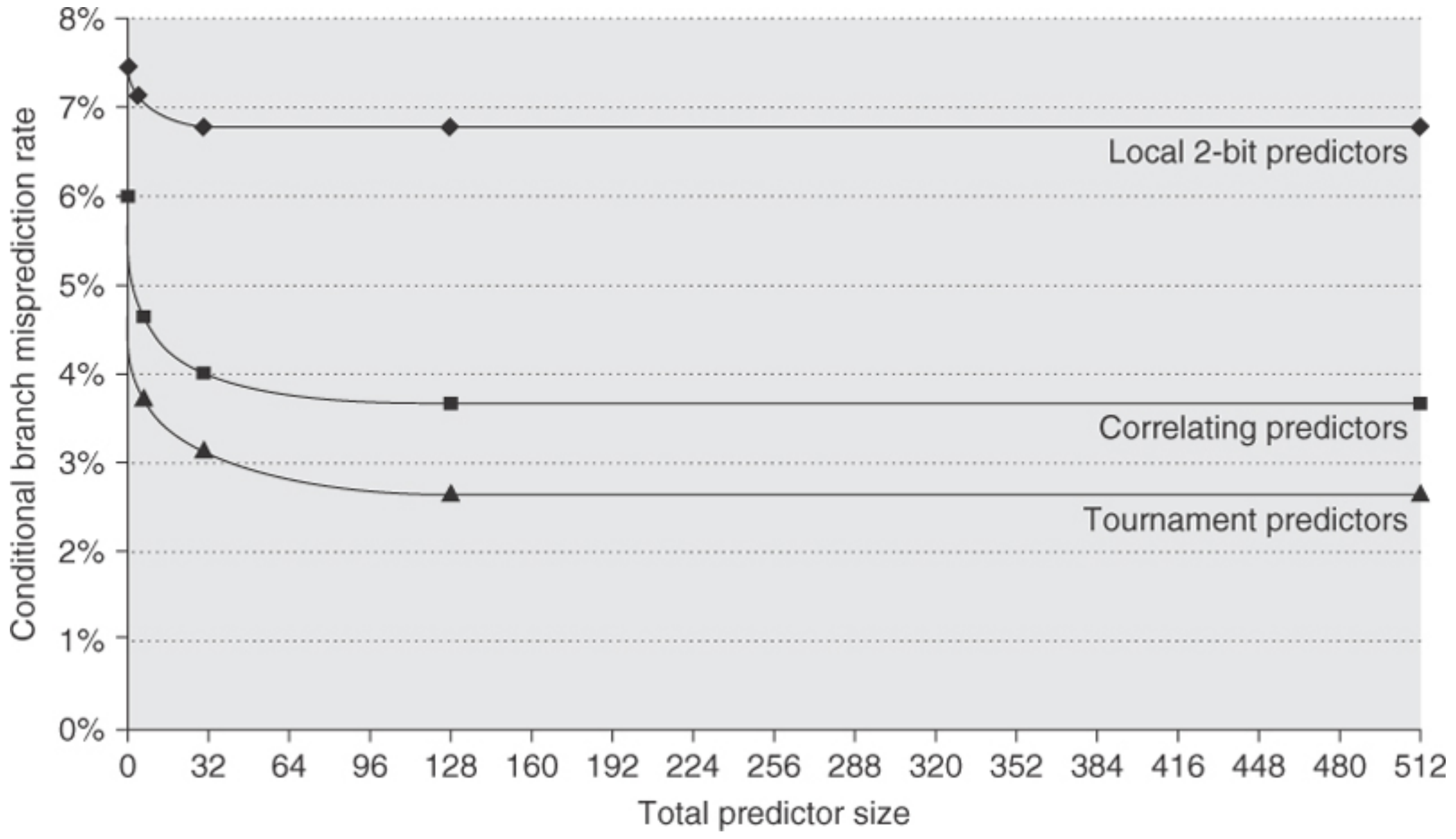


Figure 3.4 The misprediction rate for three different predictors on SPEC89 as the total number of bits is increased. The predictors are a local 2-bit predictor, a correlating predictor that is optimally structured in its use of global and local information at each point in the graph, and a tournament predictor. Although these data are for an older version of SPEC, data for more recent SPEC benchmarks would show similar behavior, perhaps converging to the asymptotic limit at slightly larger predictor sizes.

Branch Target Buffers (BTB)

- In addition to predicting the branch,we need to guess the target address
- If the target address can be determined by end of IF

  → zero branch penalty
- BTB : Small cache that stores the predicted address for the next instruction after a branch

Note:

- If we use a branch prediction table → accessed during ID

  → branches have 1 cycle penalty
- If we use a BTB → accessed during IF

  → branches have 0 cycle penalty

How a BTB Works

- During the IF, we use the addr to access the BTB
- If hit, we extract the address of the next inst. to fetch
- Note : unlike the branch prediction buffer , the entry must be for this instruction, else would fetch something wrong
- Note : Only need to store predicted - taken branches
- Easiest scheme : Store in the BTB only PC-relative conditional branches $\rightarrow$ target address is a constant

- No branch delay if entry found in BTB <u>and</u> it is correct
- There is some cost in updating the BTB in case of misprediction <u>or</u> wrong target
- We do not try to update BTB while fetching instr

  $\rightarrow$ could not access it

  $\rightarrow$ best to stall 1 or 2 cycles
- Can be combined with a branch prediction table to decide when to put entries in BTB

See Figure 3.21 and Figure 3.22

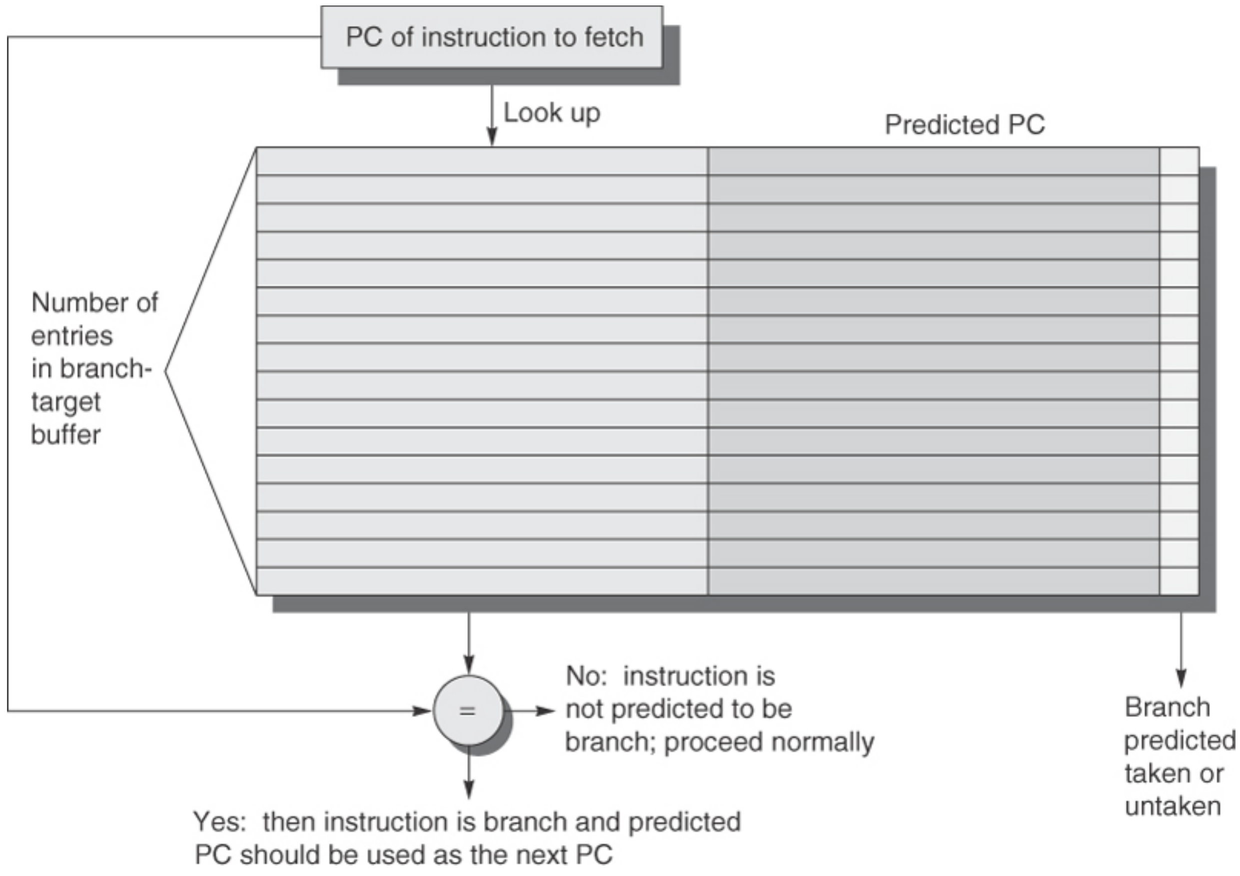

PC of instruction to fetch
Look up
Predicted PC
Number of entries in branch-target buffer
=
No: instruction is not predicted to be branch; proceed normally
Yes: then instruction is branch and predicted PC should be used as the next PC
Branch predicted taken or untaken

Figure 3.21 A branch-target buffer. The PC of the instruction being fetched is matched against a set of instruction addresses stored in the first column; these represent the addresses of known branches. If the PC matches one of these entries, then the instruction being fetched is a taken branch, and the second field, predicted PC, contains the prediction for the next PC after the branch. Fetching begins immediately at that address. The third field, which is optional, may be used for extra prediction state bits.

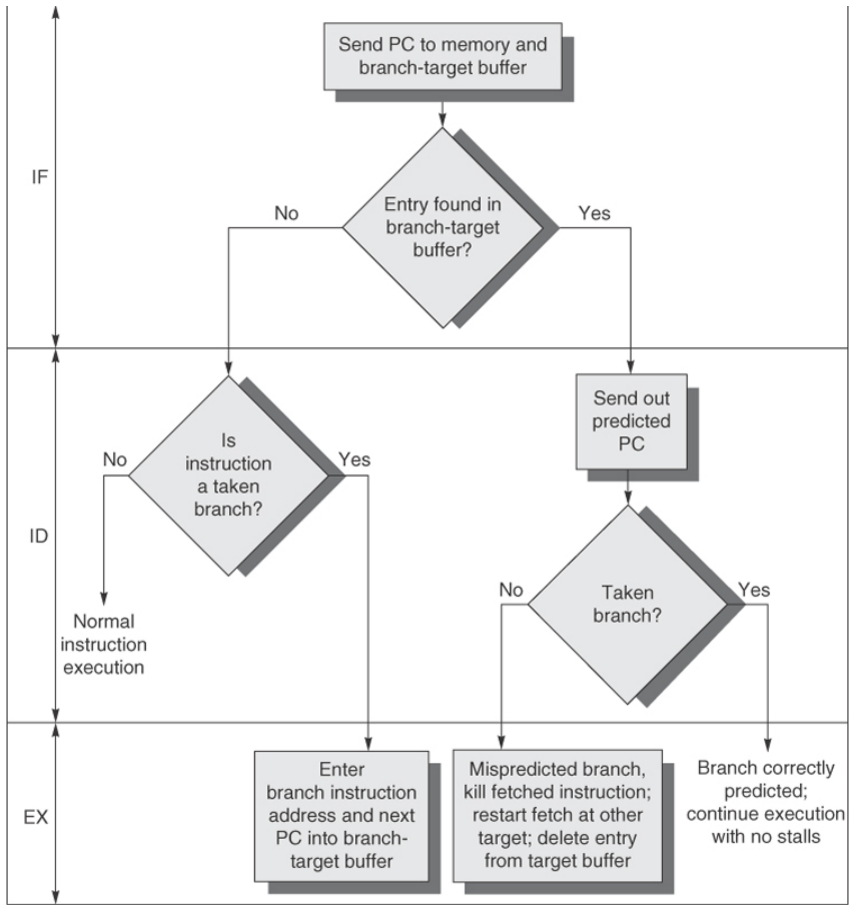

IF
ID
EX
Send PC to memory and branch-target buffer
Entry found in branch-target buffer?
No
Yes
Is instruction a taken branch?
No
Yes
Normal instruction execution
Send out predicted PC
Taken branch?
No
Yes
Enter branch instruction address and next PC into branch-target buffer
Mispredicted branch, kill fetched instruction; restart fetch at other target; delete entry from target buffer
Branch correctly predicted; continue execution with no stalls

Figure 3.22 The steps involved in handling an instruction with a branch-target buffer.

- If branch not correctly predicted (not taken or taken to wrong target) : 1 cycle update BTB

  1 cycle to restart fetching
- If branch not found & taken: 2 cycles update BTB

See figure 3.23

Example : Find branch penalty if

BTB hit rate = 90%

Prediction accuracy = 90% (for instructions in the buffer)

Branch taken frequency (for I not in buffer) = 60%

Branch Penalty = Hit in BTB and wong prediction * 2 + Miss in BTB and taken *2

= 0.90 * 0.1 * 2 + 0.1 * 0.6 * 2

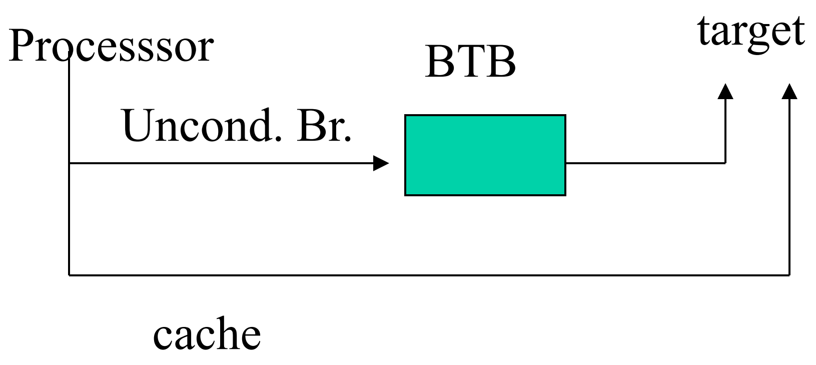
Fancier BTBs

- Store target instruction instead of target address

→ BTB access can take longer now (≡ larger BTB)

→ can do "branch folding "

(achieves zero cycle unconditional BR and sometimes zero cycle cond. BR.)



Zero cost unconditional Br

Integrated Instruction Fetch Units

- Have a fancy module that implements IF in multiple cycles (since it has to provide multiple I)
- IFU performs the following functions:
  - Branch prediction
  - Instruction prefetch
  - Buffering of instructions (may come from multiple cache lines, etc)
- IFU provides I to the issue stage

Return Address Predictors

- Handling indirect jumps (from procedure returns)
  - cannot use traditional BTB's (target changes)
  - use a stack : push the return addr on a call; pop it at return. For example 1 - 16 entries

Other

- Fetch from both the predicted and unpredicted direction:
  - Some processors have used
  - costly