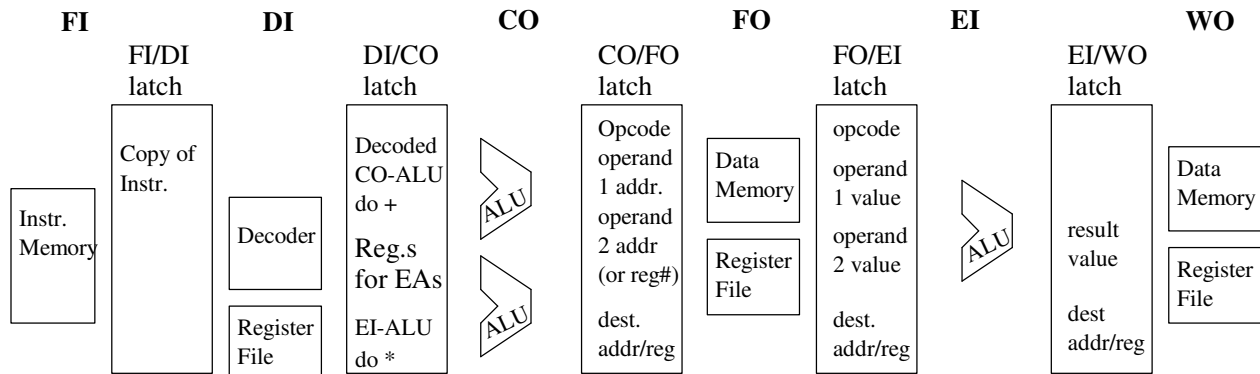


Control Hazards - branching causes problems since the pipeline can be filled with the wrong instructions.

Example: Two possible “streams” of instruction

```

BEQ R3, R8, ELSE
ADD R4, R5, R6          /* ADD should not be executed if the branch is taken */
SUB R8, R5, R6
.
.
ELSE: OR R3, R3, R2
    
```



During which stage is the target address (address of the instruction to branch to) likely to be calculated?

During which stage is the comparison between registers (R3 and R8) likely to be performed?

Stage	Abbreviation	Actions
Fetch Instruction	FI	Read next instruction into CPU
Decode Instruction	DI	Determine opcode and operand specifiers
Calculate Operands	CO	Calculate the effective addresses of all operands (on branch, calculate target address of branch)
Fetch Operands	FO	Fetch operands from memory or register file
Execute Instruction	EI	Perform the indicated operation (on conditional branch, compare register values and set PC to target address if comparison true)
Write Operand	WO	Write operand to memory or register file

Assume the branch is taken:

	Time →											
Instructions	1	2	3	4	5	6	7	8	9	10	11	12
BEQ R3, R8, ELSE		FI	DI	CO	FO	EI	WO					
ADD R4, R5, R6			FI	DI	CO	FO						
SUB R8, R5, R6				FI	DI	CO						
					FI	DI						
						FI						
ELSE: OR R3, R3, R2							FI	DI	CO	FO	EI	WO

If the branch is taken, then there is a *branch penalty* of ____ cycles.

If the branch is not taken and we continue to fetch instructions sequentially, then there is a branch penalty of ____ cycles.

Possible approaches to reducing the branch penalty:

1) Multiple streams - replicate initial pipeline stages and fetch from both streams when a branch is detected

Problems:

- causes contention for memory, registers, etc. since duplicate stages need the same hardware
- additional branch instructions enter the pipeline before the original branch decision is resolved
e.g., BEQ R3, R8, ELSE

ELSE: BNEZ R5, ELSE_2

2) Prefetch Branch of Target - (a limited form of multiple streams) save the instruction(s) from the target until the outcome of the branch is known; This saves on fetching if the branch is taken.

3) Loop buffer - small high-speed memory to store the last n most recently fetched instructions in sequence. If the branch is taken, then the loop buffer is checked to see if it already contains the target instruction.

4) Branch Prediction - predict whether the branch will be taken and fetch accordingly

Static Techniques:

a) Predict never taken - continue to fetch sequentially. If the branch is not taken, then there is no wasted fetches.

b) Predict always taken - fetch from branch target as soon as possible
(From analyzing program behavior, > 50% of branches are taken.)

c) Predict by opcode - compiler helps by having different opcodes based on likely outcome of the branch

Consider the HLL constructs:

```

HLL                                AL
While (x > 0) do                       CMP x, #0
                                        BR_LE_PREDICT_NOT_TAKEN  END_WHILE
    {loop body}
end while                               END_WHILE:

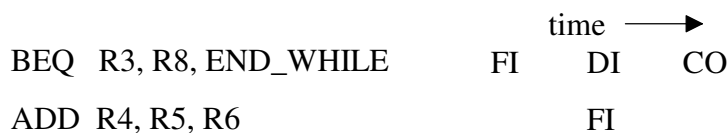
```

Studies have found about a 75% successful prediction rate using this technique.

Dynamic Techniques: try to improve prediction by recording history of conditional branch

d) Taken / Not Taken switch - one or more history bits to reflect whether the most recent executions of an instruction were taken or not.

Problem: How do we avoid always fetching the instruction after the branch?



Need target of branch, but its not calculated yet!
Plus, how do we know that we have just fetched a branch since it has not been decoded yet?

Solution:

e) Branch-History Table - small, fully-associative cache to store information about most recently executed branch instructions. (Figure 12.20)

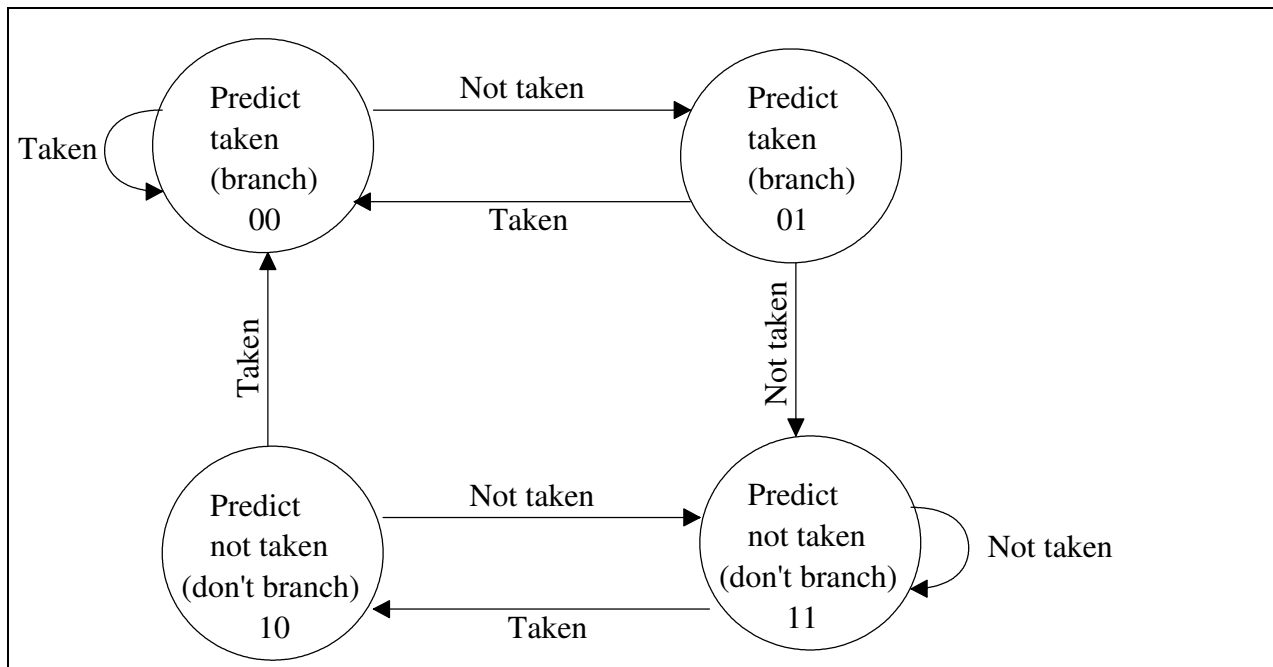
n	Type of mix		
	Compiler	Business	Scientific
0	64.1	64.4	70.4
1	91.9	95.2	86.6
2	93.3	96.5	90.8
3	93.7	96.6	91.0
4	94.5	96.8	91.8
5	94.7	97.0	92.0

Table below shows the advantage of using a Branch-history table to improve accuracy of the branch prediction. It shows the impact of past n branches on prediction accuracy.

Notice:

- 1) the big jump in using the knowledge of just 1 past branch to predict the branch
- 2) notice the big jump in going from using 1 to 2 past branches to predict the branch for scientific applications. What types of data do scientific applications spend most of their time processing? What would be true about the code for processing this type of data?

Typically, two prediction bits are use so that two wrong predictions in a row are need to change the prediction -- see Figure 12.19



How does this help for nested loops?

```

Consider the nested loops:  for (i = 1; i <= 100; i++) {
                             for (j = 1; j <= 100; j++) {
                                 <do something>
                             }
                         }

```

		<u>PREDICTION</u>	<u>ACTUAL</u>	<u>NEXT PREDICTION</u>	<u>PENALTY</u>	
i = 1		miss in BHT	NOT TAKEN	NOT TAKEN		
	j = 1	miss in BHT	NOT TAKEN	NOT TAKEN		
	j = 2	NOT TAKEN	NOT TAKEN	NOT TAKEN		
		•				
		•				
		•				
	j = 100	NOT TAKEN	NOT TAKEN	NOT TAKEN		
	j = 101	NOT TAKEN	TAKEN	TAKEN	1	
	i = 2		NOT TAKEN	NOT TAKEN	NOT TAKEN	
		j = 1	TAKEN	NOT TAKEN	NOT TAKEN	1
j = 2		NOT TAKEN	NOT TAKEN	NOT TAKEN		
		•				
		•				
		•				
j = 100		NOT TAKEN	NOT TAKEN	NOT TAKEN		
j = 101		NOT TAKEN	TAKEN	TAKEN	1	
		•				
		•				

Penalties associated with conditional branches = $1 + 2 * 99 + 1$
└──┬──┘ └┘
inner outer
loop loop

```

Consider the nested loops:  for (i = 1; i <= 100; i++) {
                             for (j = 1; j <= 100; j++) {
                                 <do something>
                             }
                         }

```

Nested Loop Unconditional Branch Penalty With 2 History Bit

	<u>PREDICTION</u>	<u>ACTUAL</u>	<u>NEXT PREDICTION</u>	<u>PENALTY</u>
i = 1	miss in BHT	NOT TAKEN	NOT TAKEN	
j = 1	miss in BHT	NOT TAKEN	NOT TAKEN	
j = 2	NOT TAKEN	NOT TAKEN	NOT TAKEN	
•				
•				
•				
j = 100	NOT TAKEN	NOT TAKEN	NOT TAKEN	
j = 101	NOT TAKEN	TAKEN	NOT TAKEN	1
i = 2	NOT TAKEN	NOT TAKEN	NOT TAKEN	
j = 1	NOT TAKEN	NOT TAKEN	NOT TAKEN	
j = 2	NOT TAKEN	NOT TAKEN	NOT TAKEN	
•				
•				
•				
j = 100	NOT TAKEN	NOT TAKEN	NOT TAKEN	
j = 101	NOT TAKEN	TAKEN	NOT TAKEN	1
•				
•				
•				

Penalties associated with conditional branches = $\underbrace{1 * 100}_{\text{inner}} + \underbrace{1}_{\text{outer}}$

5) **Delayed Branching** - redefine the branch such that one (or two) instruction(s) after the branch will always be executed.

Compiler automatically rearranges code to fill the delayed-branch slot(s) with instructions that can always be executed. Instructions in the delayed-branch slot(s) do not need to be flushed after branching. If no instruction can be found to fill the delayed-branch slot(s), then a NOOP instruction is inserted.

Without Delayed Branching	With Delayed Branching
SUB R8, R2, R1 BEQZ R3, ELSE ADD R4, R5, R6 . . ELSE: ADD R3, R3, R2	BEQZ R3, ELSE SUB R8, R2, R1 # delay slot always done ADD R4, R5, R6 . . ELSE: ADD R3, R3, R2

Due to data dependences, the instruction before the branch cannot always be moved into the branch-delay slot. Other alternative to consider are:

The Instruction at the Target of the Branch	
Without Delayed Branching	With Delayed Branching
LOOP: ADD R7, R8, R9 . . . SUB R3, R2, R1 BEQZ R3, LOOP MUL R4, R5, R6	ADD R7, R8, R9 LOOP: . . . SUB R3, R2, R1 BEQZ R3, LOOP ADD R7, R8, R9 #delay slot MUL R4, R5, R6

Can this technique always be used?

The Instruction From the Fall-Through of the Branch	
Without Delayed Branching	With Delayed Branching
SUB R3, R2, R1 BEQZ R3, ELSE ADD R8, R5, R6 . . ELSE: ADD R3, R3, R2	SUB R3, R2, R1 BEQZ R3, ELSE ADD R8, R5, R6 # delay slot . . ELSE: ADD R3, R3, R2

Can this technique always be used?