8 Processing of control transfer instructions

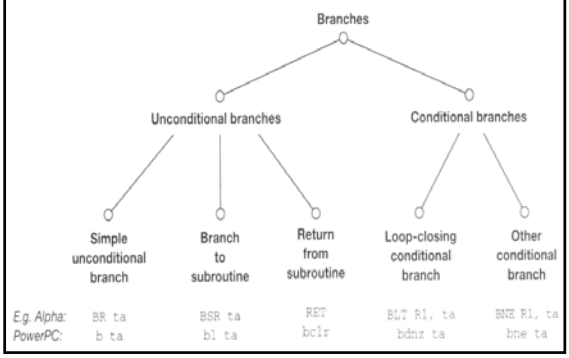
- 8.1 Introduction
- 8.2 Basic approaches to branch handling
- 8.3 Delayed branching
- 8.4 Branch processing
- 8.5 Multiway branching
- 8.6 Guarded execution

TECH
Computer Science

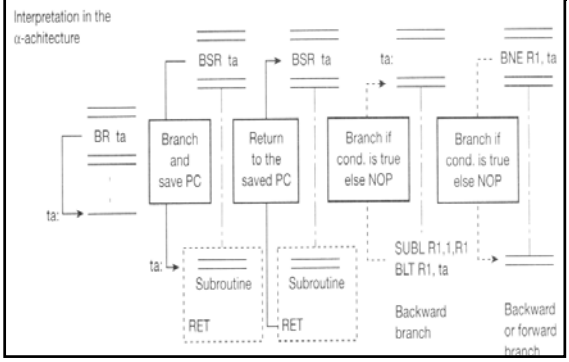
8.1 Intro to Branch

- Branches modify, conditionally or unconditionally, the value of the PC.
- To transfer control
- To alter the sequence of instructions

Major types of branches



Branch: To transfer control

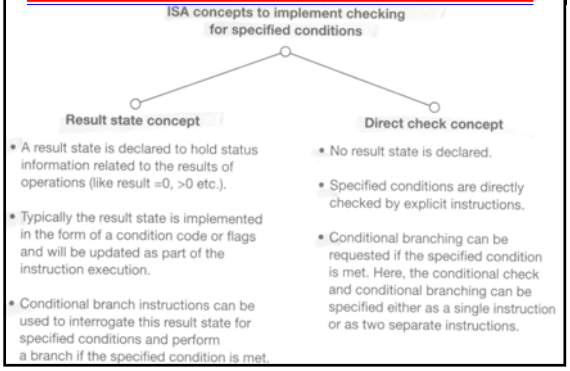


Branch: e.g.

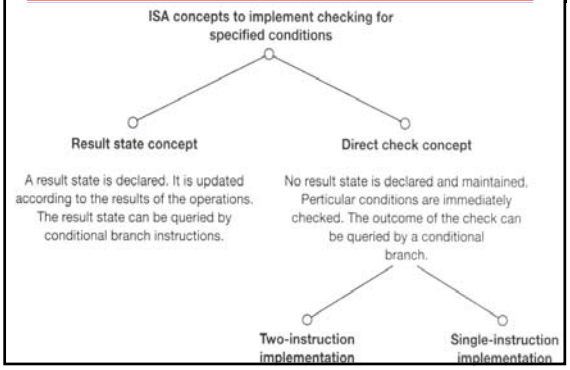
Semantics of the non-self-explanatory instructions:

```
BLT    R1, ta   // Branch if (R1) ≤ 0
SUBL   R1,1,R1  // Decrement R1 by 1,
BNE    R1, ta   // Branch to 'ta:' if (R1) ≠ 0
bl     ta       // Branch to 'ta:' and store next PC into the Link
                   Register
bclr            // Branch to the address stored in the Link Register
bdnz   ta       // Decrement Count Register, branch to 'ta:' if Count
                   Register ≠ 0
```

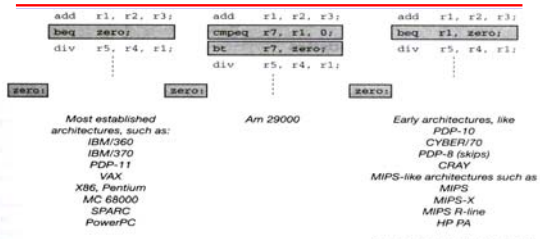
8.1.2 How to check the results of operations for specified conditions {branch} (e.g. equals 0, negative, and so on)



Alternatives for checking the operation results



Result State vs. Direct Check, e.g.



Semantics of the non self-explanatory instructions used in the example:

```
cmpeq    r7,r1,r2;    // r7 ← true if (r1) = (r2), else r7 ← false
bt       r7,null;     // branch to 'null': if (r7) = true, else NOP
cmpbeq   r1,r2,null;  // branch to 'null': if (r1) = (r2), else NOP
```

Result state approach: Disadvantage

- The generation of the result state is not straightforward
  - **It requires an irregular structure and occupies additional chip area**
- The result state is a sequential concept.
  - **It cannot be applied without modification in architectures which have multiple execution units.**

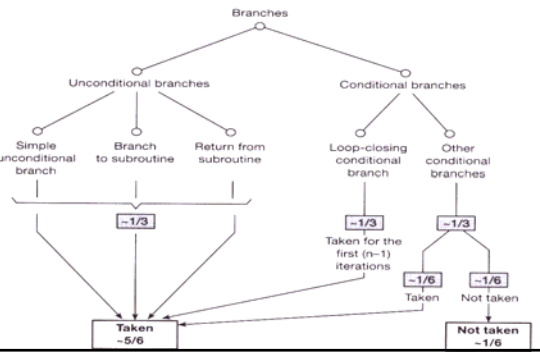
Retaining sequential consistency for condition checking (in VLIW or Superscalar)

- Use multiple sets of condition codes or flags
  - **It relies on programmer or compiler to use different sets condition codes or flags for different outcome generated by different EUs.**
- Use Direct Check approach.

Branch Statistics

- 20% of general-purpose code are branch
  - **on average, each fifth instruction is a branch**
- 5-10% of scientific code are branch
- The Majority of branches are conditional (80%)
- 75-80% of all branches are taken

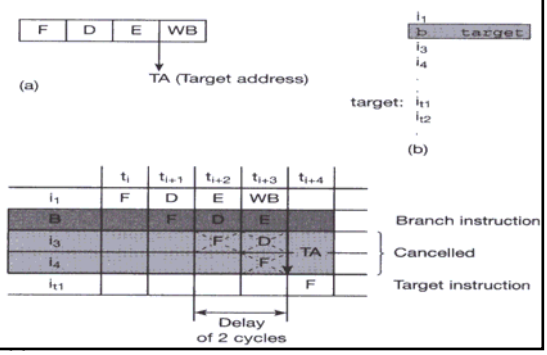
Branch statistics: Taken or not Taken



Frequency of taken and not-taken branches

Reference	Frequency of taken branches	Frequency of untaken branches
Lee and Smith, 1984	57–99 %	1–43 %
Edenfield et al., 1990	75 %	25 %
Grohoski, 1990	~ 5/6	~ 1/6

8.1.4 The branch problem: The delay caused in pipelining



More branch problems

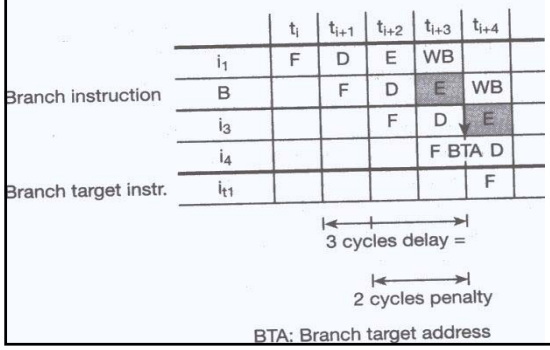
- Conditional branch could cause an even longer penalty
  - **evaluation of the specified condition needs an extra cycle**
  - **waiting for unresolved condition (the result is not yet ready)**
    - e.g. wait for the result of FDIV may take 10-50 cycles
- Pipelines became more stages than 4
  - **each branch would result in a yet larger number of wasted cycles (called bubbles)**

8.1.5 Performance measures of branch processing

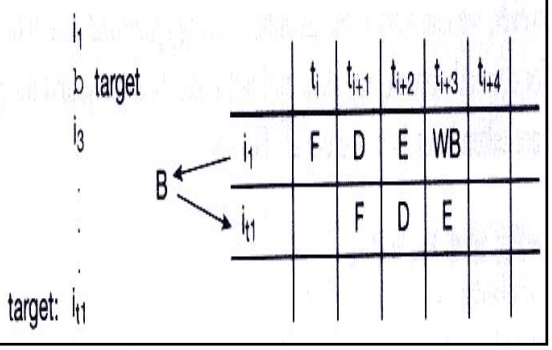
- **Pt : branch penalties for taken**
- **Pnt : branch penalties for not-taken**
- **ft : frequencies of taken**
- **fnt : frequencies for not-taken**
- **P : effective penalty of branch processing**

- P = ft * Pt + fnt * Pnt
  - **e.g. 80386:: P = 0.75 * 8 + 0.25 * 2 = 6.5 cycles**
  - **e.g. i486:: P = 0.75 * 2 + 0.25 * 0 = 1.5 cycles**
- Branch prediction correctly or mispredicted
- P = fc * Pc + fm * Pm
  - **e.g. Pentium:: P = 0.9 * 0 + 0.1 * 3.5 = 0.35 cycles**

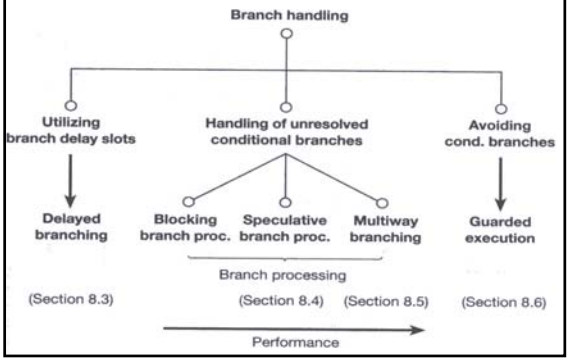
Interpretation of the concept of branch penalty



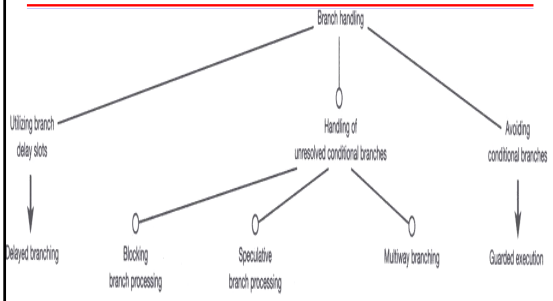
Zero-cycle branching {in no time}



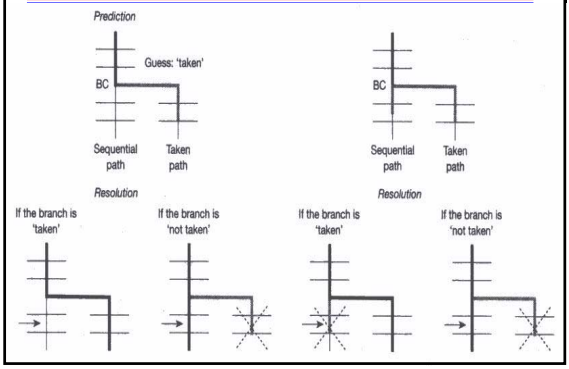
8.2 Basic approaches to branch handling



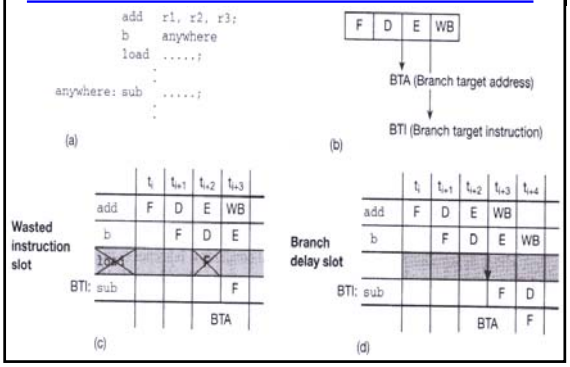
Review of the basic approaches to branch handling



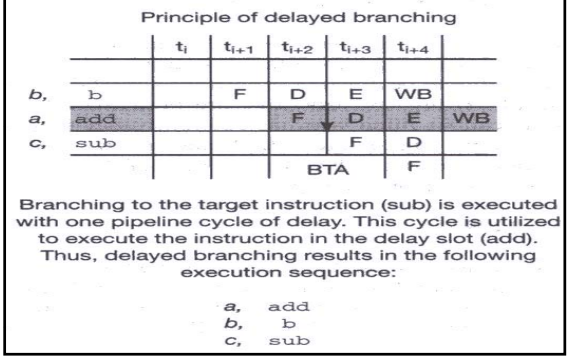
Speculative vs. Multiway branching



-Delayed Branching: Occurrence of an unused instruction slot (unconditional branch)



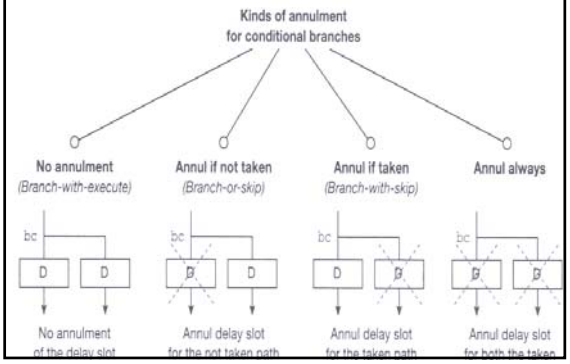
Basic scheme of delayed branching



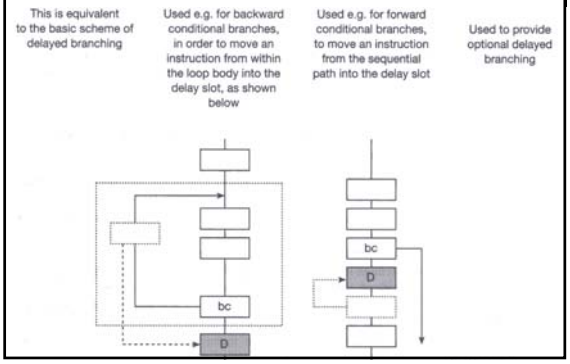
Delayed branching: Performance Gain

- Ratio of the delay slots that can be filled with useful instructions:: f_f
  - **60-70% of the delay slot can be fill with useful instruction**
    - fill only with: instruction that can be put in the delay slot but does not violate data dependency
    - fill only with: instruction that can be executed in single pipeline cycle
- Frequency of branches:: f_b
  - **20-30% for general-propose program**
  - **5-10% for scientific program**
- 100 instructions have 100* f_b delay slots,
- 100*f_b * f_f can be utilized.
- Performance Gain = (100*f_b * f_f)/100 = f_b * f_f

Delayed branching: for conditional branches {Can be cancel or not}



Where to find the instruction to fill delay slot

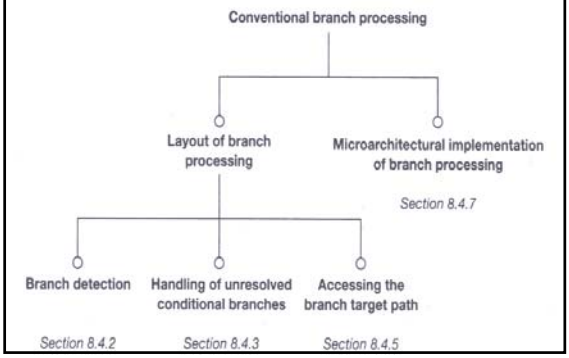


Possible annulment options provided by architectures (use special instructions) with delayed branching {Scalar only}//

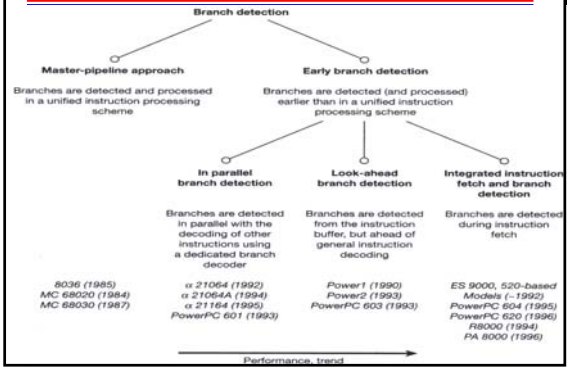
	Annulment of an instruction in a delay slot				
	Branch-with-execute	*Branch-or-skip*	*Branch-with-skip*	*Annul always*	*Multiplicity of delay slot*
IBM 801 (1978)	X			X	1
MIPS-X (1986)	X	X			2
HP PA (1986)	X	X[1]	X[2]		1
SPARC (1987)	X	X			1
MC 88100 (1988)	X			X	1
i860 (1988)		X		X	1

1: Backward branches
2: Forward branches

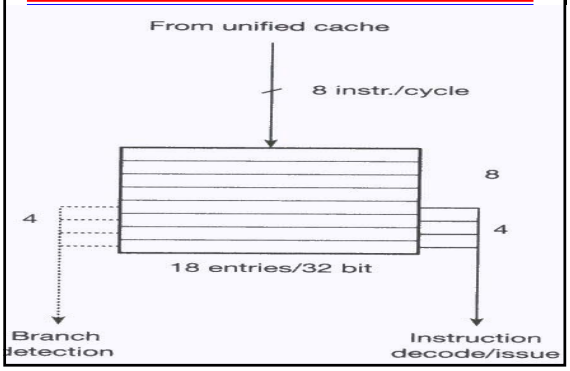
8.4 Branch Processing: design space



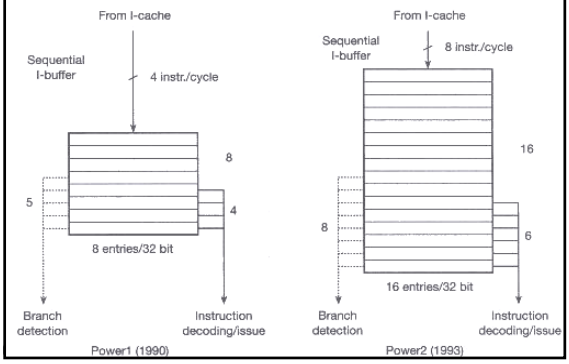
Branch detection schemes {early detection, better handling}



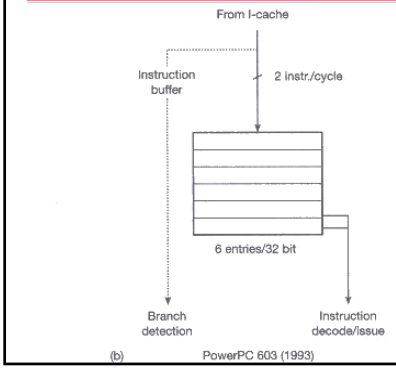
Branch detection in parallel with decoding/issuing of other instructions (in I-Buffer)



Early detection of branches by Looking ahead



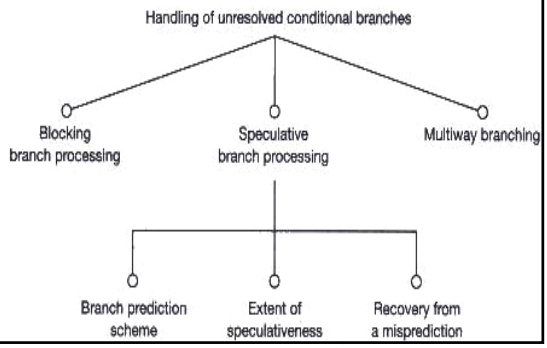
Early detection of branches by inspection of instruction that inputs to I-buffer



Early branch detection: {for scalar Processor} Integrated instruction fetch and branch detection

- Detect branch instruction during fetching
- Guess taken or not taken
- Fetch next sequential instruction or target instruction

Handling of unresolved conditional branches



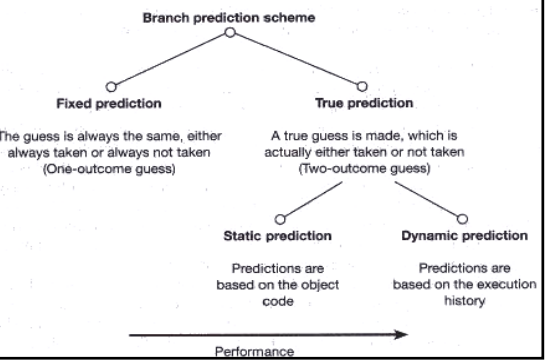
-Blocking branch processing

- Simply stalled (stopped and waited) until the specified condition can be resolved

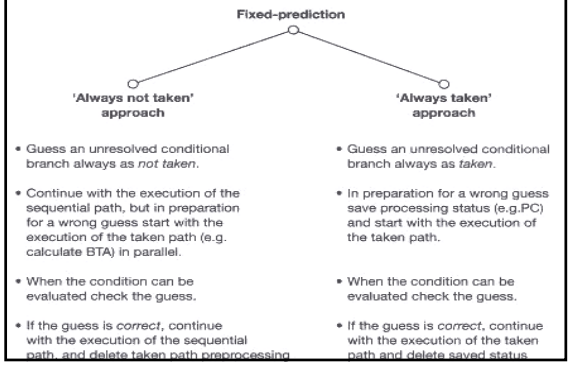
Table 8.3 Branch penalties in blocking branch processing.

Processor type	*Taken penalty cycles*	*Not-taken penalty cycles*
MC 68020 (1984)	5	3
MC 68030 (1987)	5	3
80386 (1985)	8	2

-Basic kinds of branch predictions



-The fixed prediction approach



Always not taken vs. Always Taken

- If the guess is *incorrect*, delete the speculative processing along the sequential path and continue with the processing of the taken path.

  ↓

  TP is higher than NTP.
  It is easier to implement than the 'Always taken' approach.

  E.g.Z80 000 (1984)
  80486 (1989)
  R4000 (1992)
  SuperSparc (1992)
  Power1 (1990)
  Power2 (1993)
  α 21064 (1992)
  α 21064A (1994)
  (In case of the α processors as a selectable option)

- If the guess is *incorrect*, delete the speculative processing along the taken path and continue executing the sequential path using the saved processing status.

  ↓

  TP is usually less than NTP.
  It requires a more complex implementation than the 'Always not taken' approach

  MC68040 (1990)

  TP: Taken penalty
  NTP: Not-taken penalty
  BTA: Branch target address

Performance, complexity

Always not taken: Penalty figures

Table 8.4 Penalty figures for processors employing the 'always not taken' prediction approach.

Processor type	*Taken penalty cycles*	*Not-taken penalty cycles*
Z 80000 (1984p)	3	0
80486 (1989p)	2	0
Power1 (1990)	3	0
R 4000 (1992p)	3 (D)	0
SuperSparc (1992p)	1 (D)	0
Power2 (1993)	1	0
MicroSparc (1992)	1 (D)	1 (D)

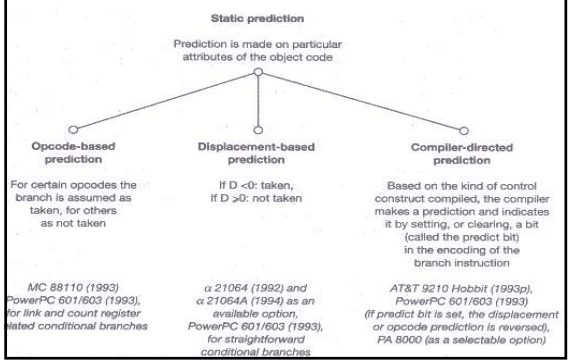
D: Delayed branching

Penalty figures for the always taken prediction approach

Table 8.5 Penalty figures for the 'always taken' prediction approach.

Processor type	*Taken penalty cycles*	*Not-taken penalty cycles*
MC 68040 (1990)	1	2

-Static branch prediction

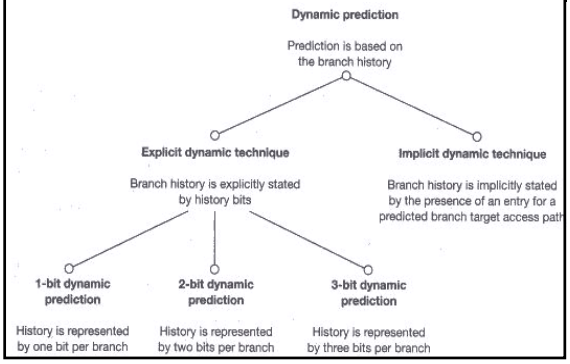


Static prediction: opcode based e.g. implemented in the MC88110

Table 8.6 Static prediction as implemented in the MC 88110 (1993).

	Instruction		
	Condition specified	*Bit 21 of the instr. code*	*Prediction*
bcnd (Branch conditional)	=0	0	Not Taken
	≠0	1	Taken
	>0	1	Taken
	<0	0	Not Taken
	≥0	1	Taken
	≤0	0	Not Taken
bb1 (Branch on bit set)			Taken
bb0 (Branch on bit clear)			Not Taken

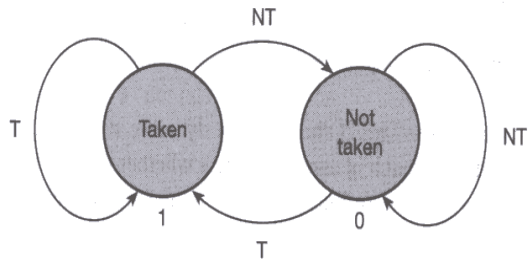
-Dynamic branch prediction: branch taken in the last n occurrences is likely to be taken next



Dynamic branch prediction: e.g.

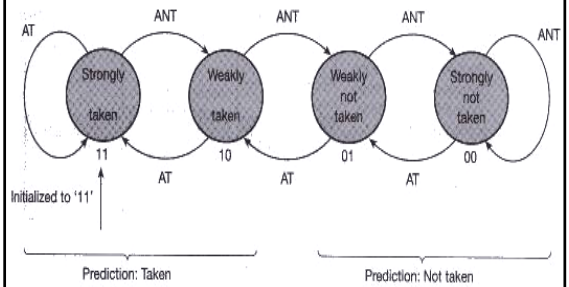
History is represented by one bit per branch		History is represented by two bits per branch		History is represented by three bits per branch			
Gmicro 100	*(1991p)*	*MC 68060*	*(1993)*	*PA 8000*	*(1996)*	*ES/9000*	*(1992p)*
α21064	*(1992)*	*Pentium*	*(1993)*			*PowerPC 604*	*(1995)*
(selectable)		*α 21064A*	*(1994)*			*PowerPC 620*	*(1996)*
R 8000	*(1994)*	*(selectable)*				*PA 8000*	*(1996)*
		α 21164	*(1995)*				
		(selectable)					
		R 8000	*(1994)*				
		PowerPC 604	*(1995)*				
		PowerPC 620	*(1996)*				
		Nx586	*(1995)*				
		M1	*(1995)*				
		UltraSparc	*(1995)*				
		R 10000	*(1996)*				

1-bit dynamic prediction: state transition diagram



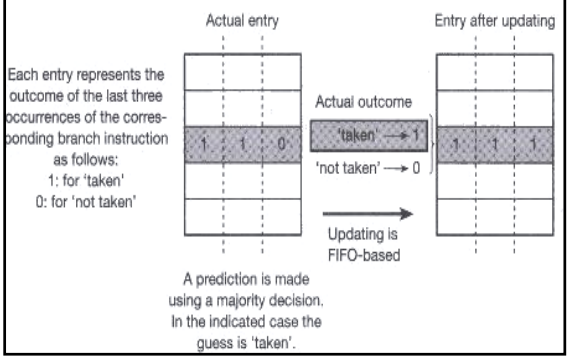
T: Branch has been taken
NT: Branch has not been taken

2-bit dynamic prediction: state transition diagram



AT: Branch has actually been taken
ANT: Branch has actually not been taken

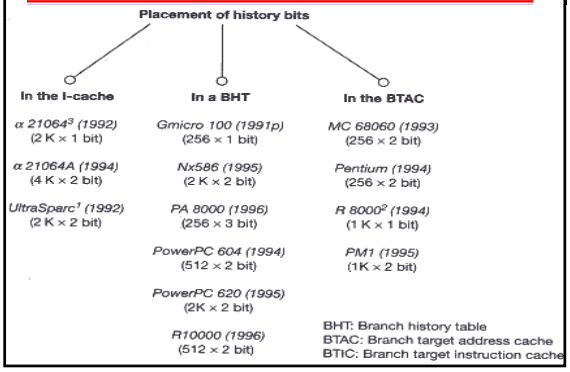
3-bit prediction



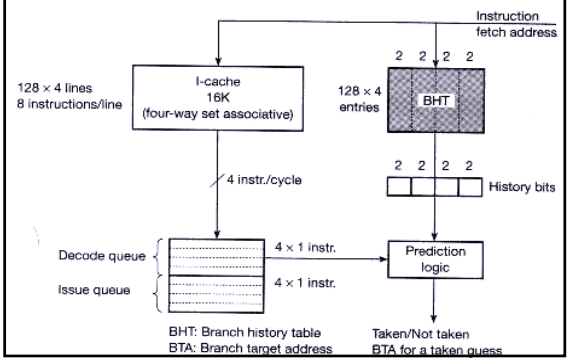
Implicit dynamic technique

- Schemes for accessing the branch target path also used for branch prediction
- Branch Target Access Cache (BTAC)
  - **holds the most recently used branch addresses**
- Branch Target Instruction Cache (BTIC)
  - **holds the most recently used target instructions**
- BTAC or BTIC holds entries only for the taken branches
- The existence of an entry means that
  - **the corresponding branch was taken at its last occurrence**
  - **so its next occurrence is also guessed as taken**

=Implementation alternatives of history bits



Example of the implementation of the BHT



=Combining implicit and 2bit prediction

Table 8.7 Combining implicit and 2-bit prediction, as implemented in the PowerPC 604 (1995) and 620 (1996) processors.

BTAC	*Outcome of the 2-bit prediction*	*Overall prediction*
Hit	Don't care	Taken
Miss	Taken	Taken
Miss	Not taken	Not taken

Combining implicit and 2bit prediction..

Table 8.8 Overall prediction by combining implicit and 2-bit prediction, as implemented in the Pentium (1993) and MC 68060 (1993) processors.

BTAC	*Outcome of the 2-bit prediction*	*Overall prediction*
Hit	Taken	Taken
Hit	Not Taken	Not taken
Miss	Don't care	Not taken

=The effect of branch accuracy on branch penalty

Table 8.9 The effect of branch accuracy on branch penalty (for $P_c = 0$ and $P_m = 4$).

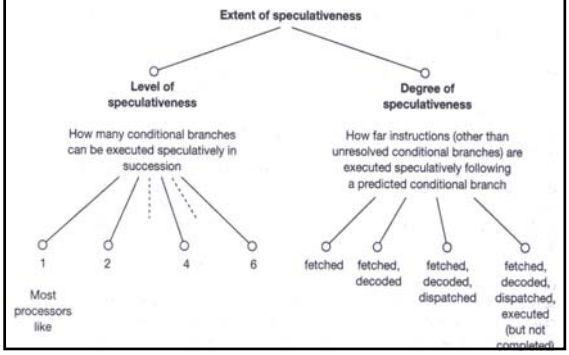
Prediction accuracy (f_c)	*Branch penalty (P_0) cycles*
0.6	1.6
0.8	0.8
0.9	0.4
0.95	0.2

Simulation results of prediction accuracy on the SPEC

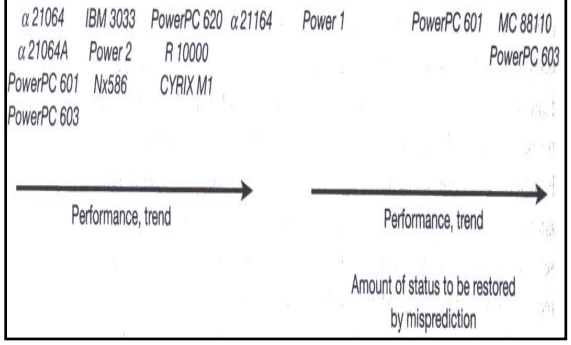
Table 8.10 Simulation results of prediction accuracy on the SPEC benchmark suite (Yeh and Patt, 1992). © 1992 ACM

Prediction method	*Prediction accuracy (%)*
Fixed, always taken	62.5
Static, displacement based	68.5
Dynamic, 1-bit	89
Dynamic, 2-bit	93

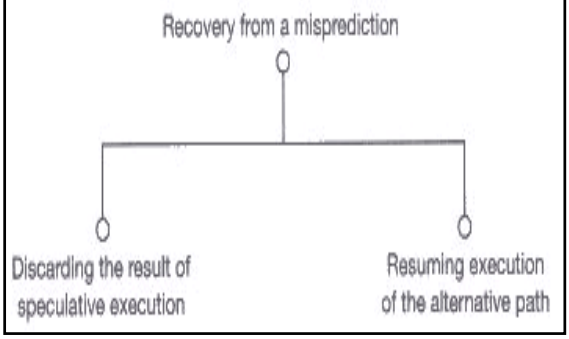
=Extent of speculative processing



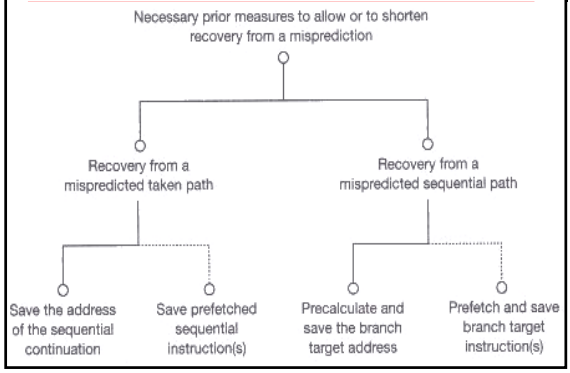
Extent of speculative processing: e.g.



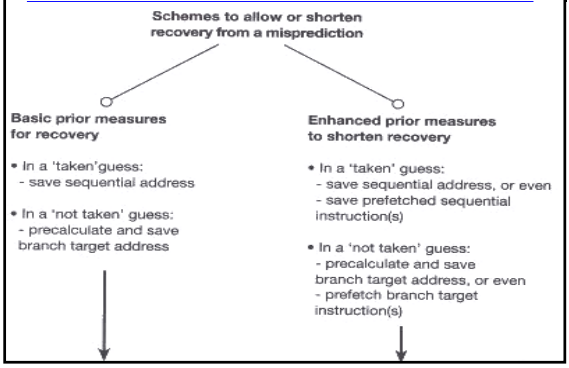
=Recovery from a misprediction: Basic Tasks



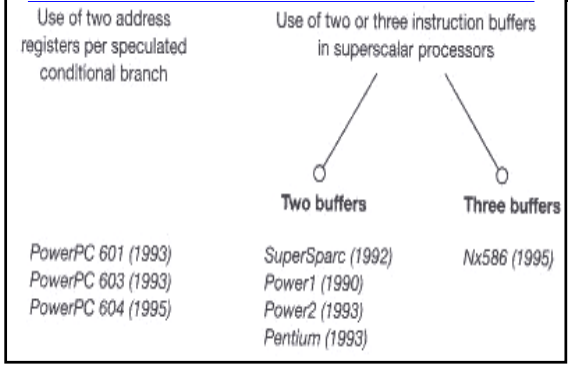
Necessary activities to allow of to shorten recovery from a misprediction



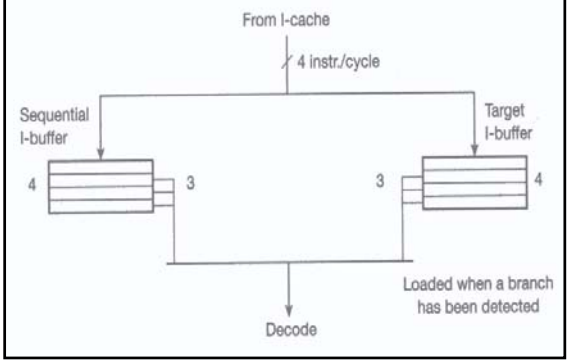
Frequently employed schemes for shortening recovery from a misprediction



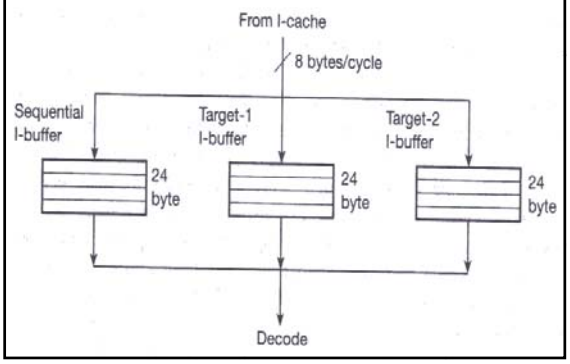
shortening recovery from a misprediction: needs



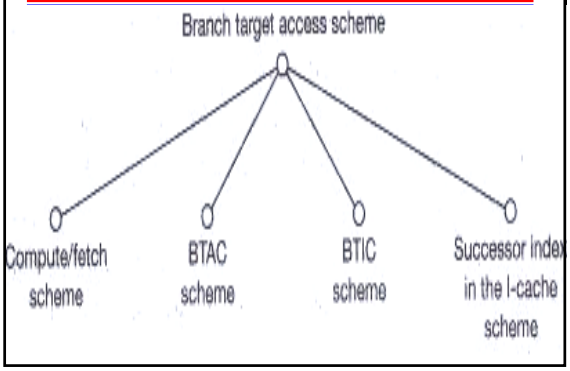
Using two instruction buffers in the supersparc to shorten recovery from a misprediction: e.g.



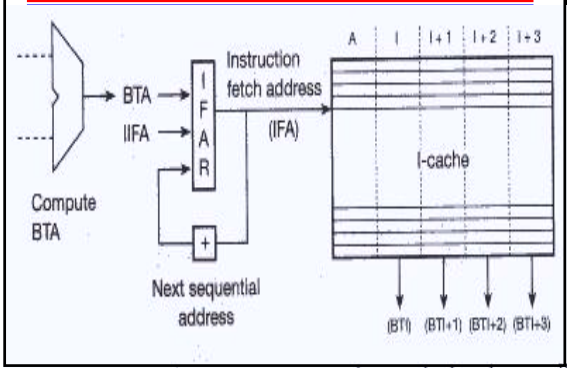
Using three instruction buffers in the Nx586 to shorten recovery from a misprediction: e.g.



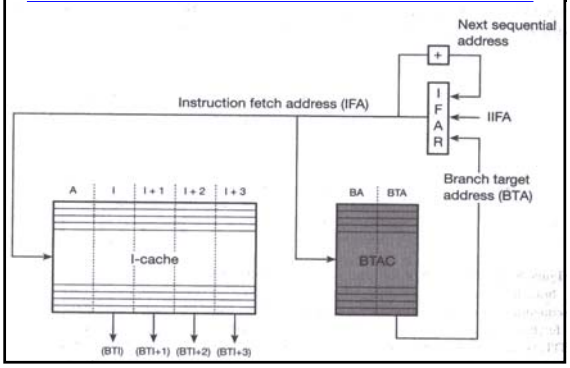
8.4.5 Branch penalty for taken guesses depends on branch target accessing schemes



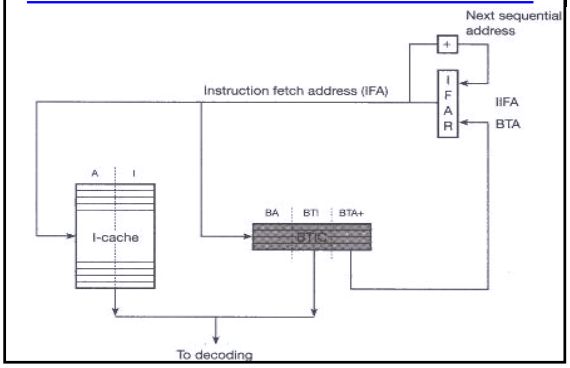
-Compute/fetch scheme for accessing branch targets {IFAR vs. PC}



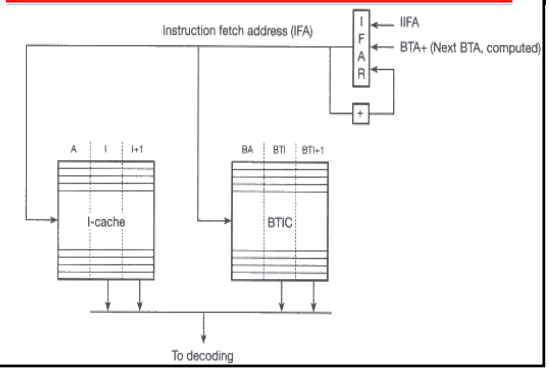
-BTAC scheme for accessing branch targets {associative search for BA, if found get BTA} {0-cycle branch: BA=BA-4}



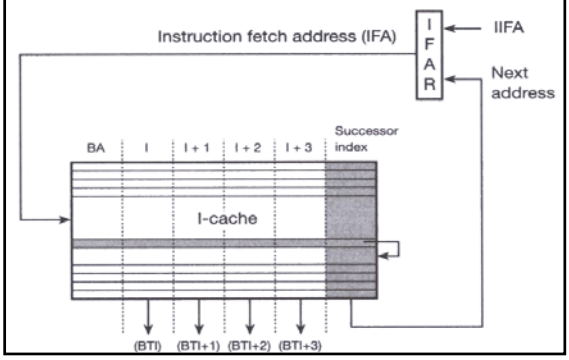
-BTIC scheme: store next BTA



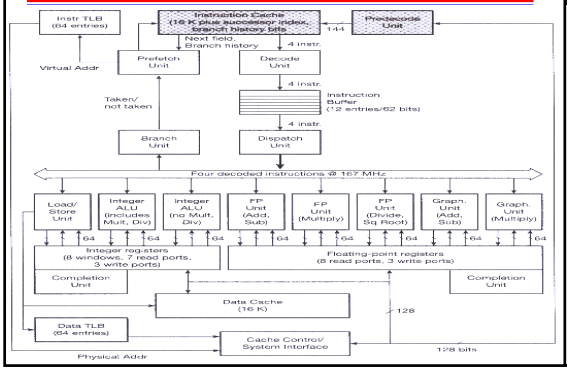
-BTIC scheme: calculate next BTA



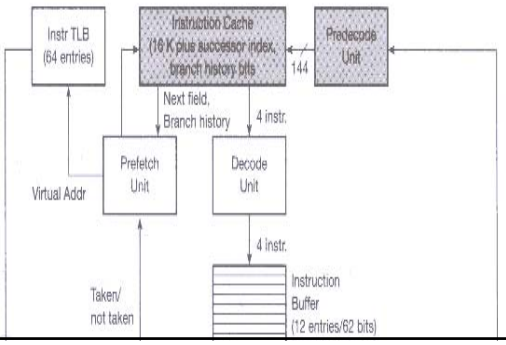
-Successor index in the I-cache scheme
to access the branch target path {index: next I, or target I}



Successor index in the I-cache scheme: e.g.
The microachitecture of the UltraSparc



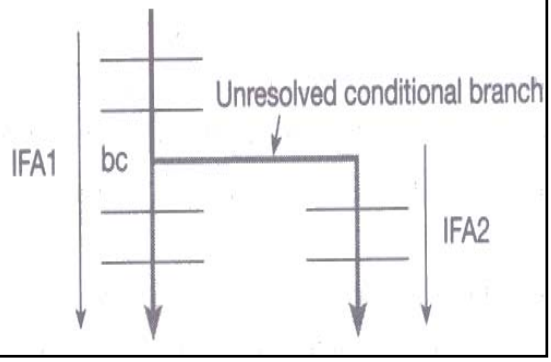
Predecode unit: detects branches, BTA, make predictions
(based on compiler's hint bit), set up I-cache Next address



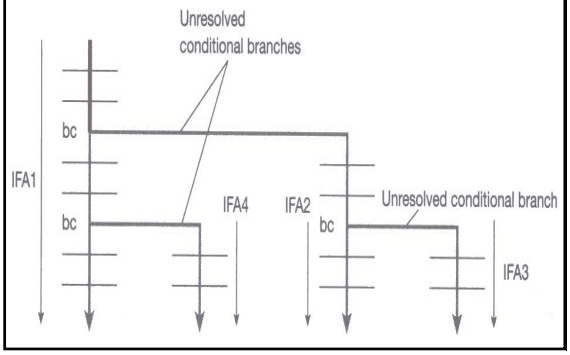
=Branch target accessing trends //

i486 (1989)	→	Pentium (1993)	
MC 68040 (1990)	→	MC 68060 (1993)	
	Am 29000 (1988) →		Am 29000 superscalar (1995)
Sparc CYC 600 (1992) SuperSparc (1992)	→		UltraSparc (1995)
R4000 (1992) R10000 (1996)	→		R8000 (1994)
PowerPC 601 (1993) PowerPC 603 (1993)	→	PowerPC 604 (1995) PowerPC 620 (1996)	

8.5 Multiway branching: {two IFA's or PC's}



Threefold multiway branching: only one correct path!



8.6 Guarded Execution

- a means to eliminate branches
- by conditional operate instructions
  - **IF the condition associated with the instruction is met,**
  - **THEN perform the specified operation**
  - **ELSE do not perform the operation**
- e.g. original
  - **beg r1, label // if (r1) = 0 branch to label**
  - **move r2, r3 // move (r2) into r3**
  - **label: …**
- e.g. guarded
  - **cmovne r1, r2, r3 // if (r1) != 0, move (r2) into r3**
  - **…**
- Convert control dependencies into data dependencies

Eliminated branches by full and restricted guarding
{full: all instruction guarded, restricted: ALU inst guarded}

Program	*Percentage of loop branches (%)*	*Percentage of eliminated branches (%)*			
		Full guarding		*Restricted guarding*	
		Cond.	*Uncond.*	*Cond.*	*Uncond.*
Compress	26.48	24.86	84.29	18.24	0.00
Eqntott	29.07	44.55	54.98	40.04	1.02
Espresso	38.08	16.76	29.03	10.17	1.17
Gcc-cc1	24.84	31.92	17.04	9.64	0.37
Sc	24.63	43.07	17.74	9.83	0.18
Sunbench	15.79	35.65	47.10	11.35	0.03
Supermips	5.03	50.69	19.36	17.15	0.60
Tektronix	16.83	37.53	41.60	17.08	7.48
TeX	25.09	12.80	24.03	5.99	1.00
Thissim	11.52	62.31	33.70	23.26	1.43
Tycho	18.28	15.64	33.84	7.10	1.31
Xlisp	27.03	13.64	14.33	13.87	14.14
Yacc	38.64	19.53	38.95	8.18	1.71
Arithmetic Mean	**23.17**	**31.15**	**35.07**	**14.76**	**2.34**

Guarded Execution: Disadvantages

- guarding transforms instructions from both the taken and the not-taken paths into guard instruction
  - **increase number of instructions**
  - **by 33% for full guarding**
  - **by 8% for restricted guarding**
  - **{more instructions more time and space}**
- guarding requires additional hardware resources if an increase in processing time is to be avoided
  - **VLIW**