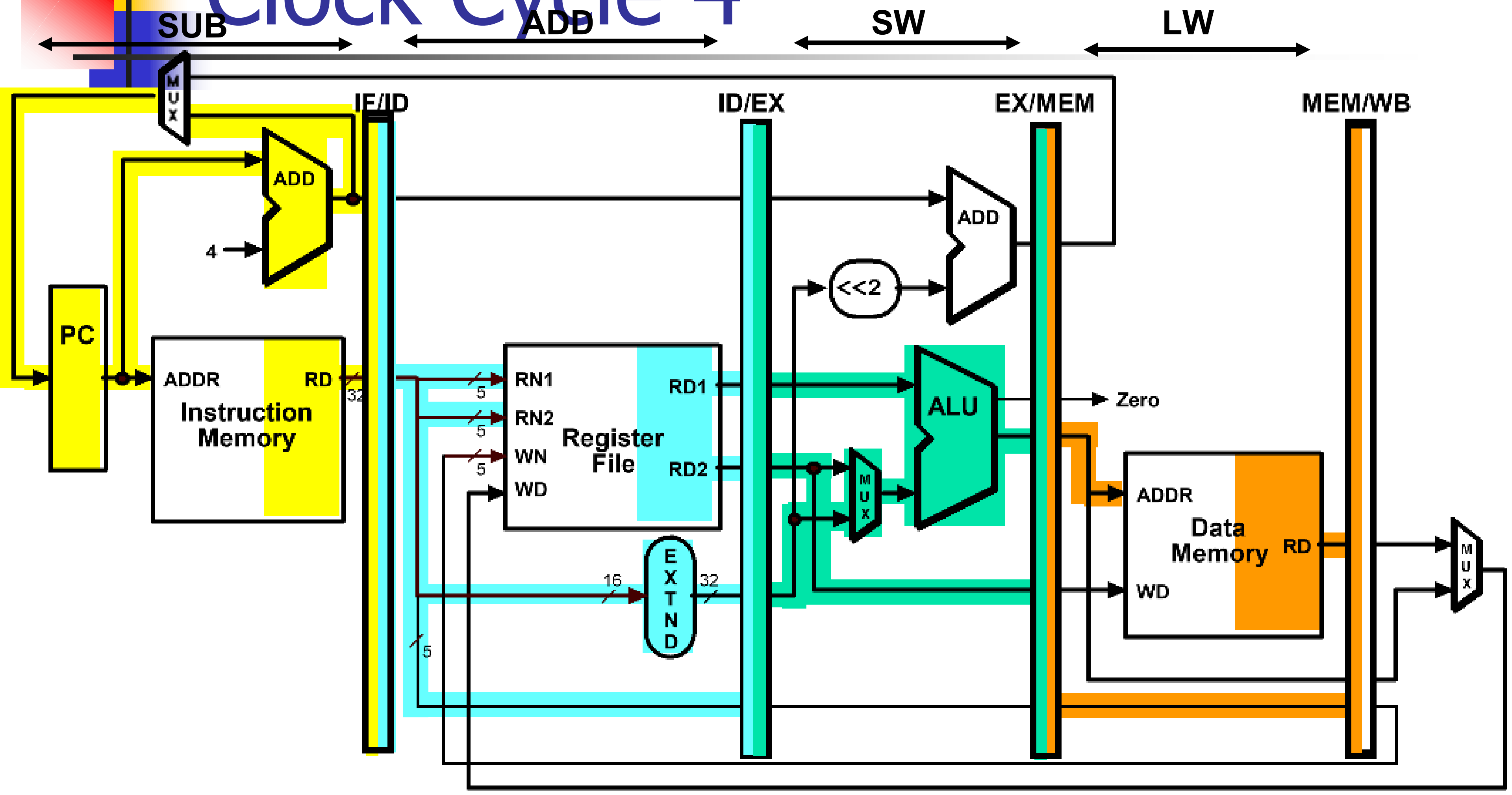
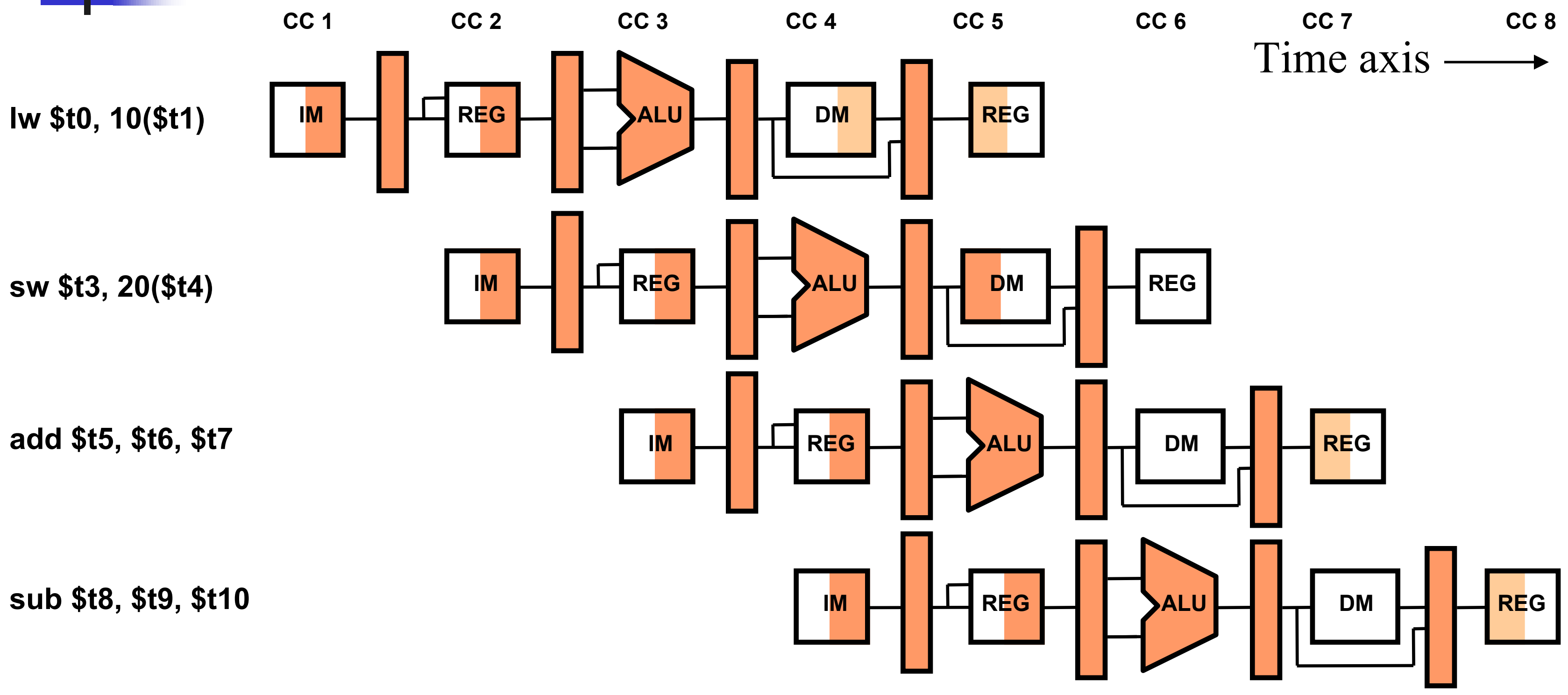
Computer Architecture
Lecture 11

Single-Clock-Cycle Diagram: Clock Cycle 4



Alternative View – Multiple-Clock-Cycle Diagram



MIPS operands

Name	Example	Comments
32 registers	$s0-$s7, $t0-$t9, $zero, $a0-$a3, $v0-$v1, $gp, $fp, $sp, $ra, $at	Fast locations for data. In MIPS, data must be in registers to perform arithmetic. MIPS register $zero always equals 0. Register $at is reserved for the assembler to handle large constants.
2^{30} memory words	Memory[0], Memory[4], ..., Memory[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential words differ by 4. Memory holds data structures, such as arrays, and spilled registers, such as those saved on procedure calls.

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add $s1,$s2,$s3	$s1 = $s2 + $s3	Three register operands
	subtract	sub $s1,$s2,$s3	$s1 = $s2 – $s3	Three register operands
	add immediate	addi $s1,$s2,100	$s1 = $s2 + 100	Used to add constants
Data transfer	load word	lw $s1,100($s2)	$s1 = Memory[$s2 + 100]	Word from memory to register
	store word	sw $s1,100($s2)	Memory[$s2 + 100] = $s1	Word from register to memory
	load half	lh $s1,100($s2)	$s1 = Memory[$s2 + 100]	Halfword memory to register
	store half	sh $s1,100($s2)	Memory[$s2 + 100] = $s1	Halfword register to memory
	load byte	lb $s1,100($s2)	$s1 = Memory[$s2 + 100]	Byte from memory to register
	store byte	sb $s1,100($s2)	Memory[$s2 + 100] = $s1	Byte from register to memory
	load upper immed.	lui $s1,100	$s1 = 100 * 2^{16}	Loads constant in upper 16 bits
Logical	and	and $s1,$s2,$s3	$s1 = $s2 & $s3	Three reg. operands; bit-by-bit AND
	or	or $s1,$s2,$s3	$s1 = $s2 | $s3	Three reg. operands; bit-by-bit OR
	nor	nor $s1,$s2,$s3	$s1 = ~ ($s2 |$s3)	Three reg. operands; bit-by-bit NOR
	and immediate	andi $s1,$s2,100	$s1 = $s2 & 100	Bit-by-bit AND reg with constant
	or immediate	ori $s1,$s2,100	$s1 = $s2 | 100	Bit-by-bit OR reg with constant
	shift left logical	sll $s1,$s2,10	$s1 = $s2 << 10	Shift left by constant
	shift right logical	srl $s1,$s2,10	$s1 = $s2 >> 10	Shift right by constant
Conditional branch	branch on equal	beq $s1,$s2,25	if ($s1 == $s2) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	bne $s1,$s2,25	if ($s1 != $s2) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	slt $s1,$s2,$s3	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than; for beq, bne
	set less than immediate	slti $s1,$s2,100	if ($s2 < 100) $s1 = 1; else $s1 = 0	Compare less than constant
Unconditional jump	jump	j 2500	go to 10000	Jump to target address
	jump register	jr $ra	go to $ra	For switch, procedure return
	jump and link	jal 2500	$ra = PC + 4; go to 10000	For procedure call

CONTROL HAZARDS

So, we've looked at two possible solutions:

- Assuming branch not taken.
  - Easy to implement.
  - High cost – three stalls if wrong.
- Performing branching in the ID stage.
  - Harder to implement – must add forwarding and hazard control earlier.
  - Lower cost – one stall if branch is taken.

We have another solution we can try: branch prediction.

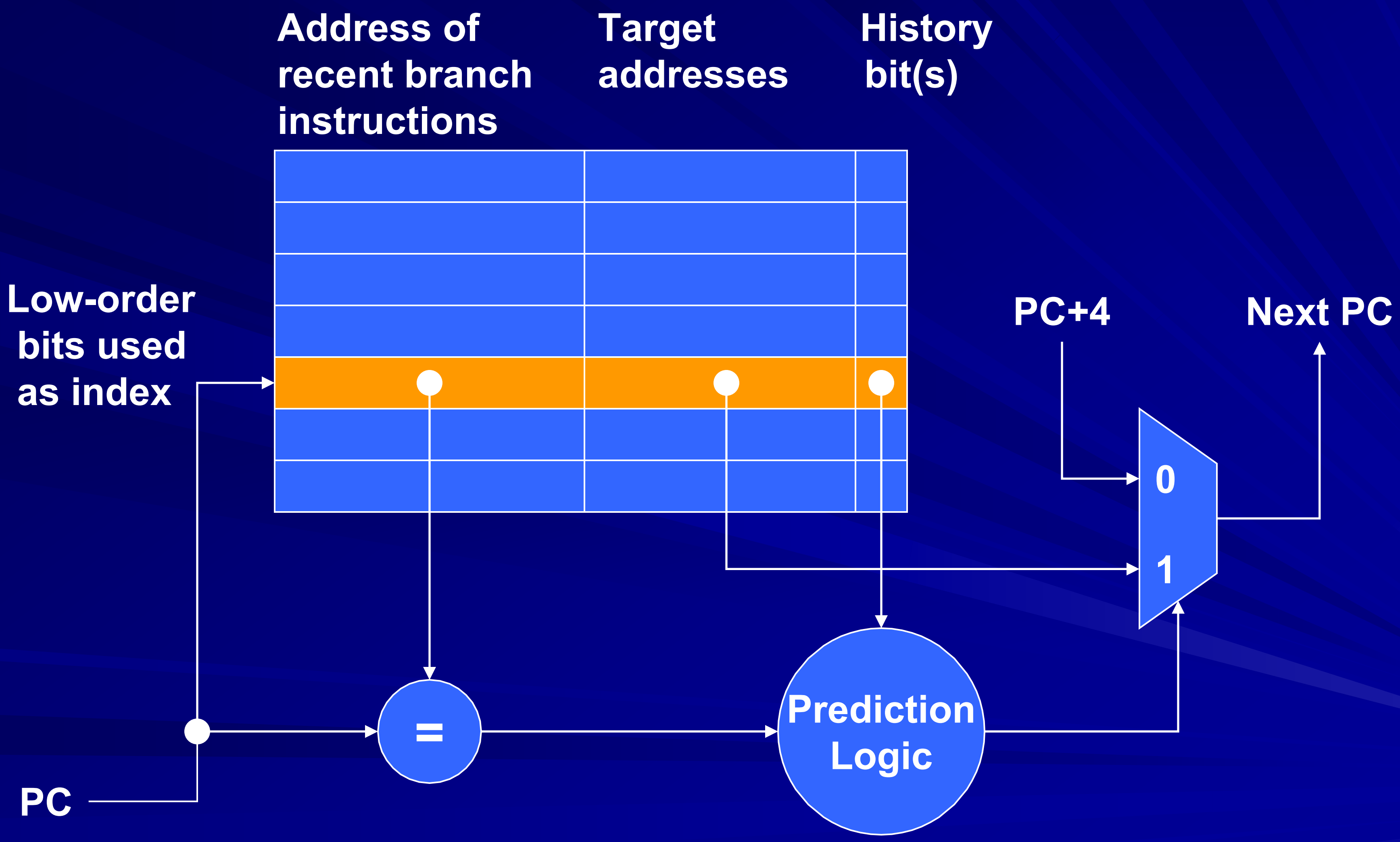
CONTROL HAZARDS

In *branch prediction*, we attempt to predict the branching decisions and act accordingly.

When we assumed the branch wasn't taken, we were making a simple static prediction. Luckily, the performance cost on a 5-stage pipeline is low but on a *deeper* pipeline with many more stages, that could be a huge performance cost!

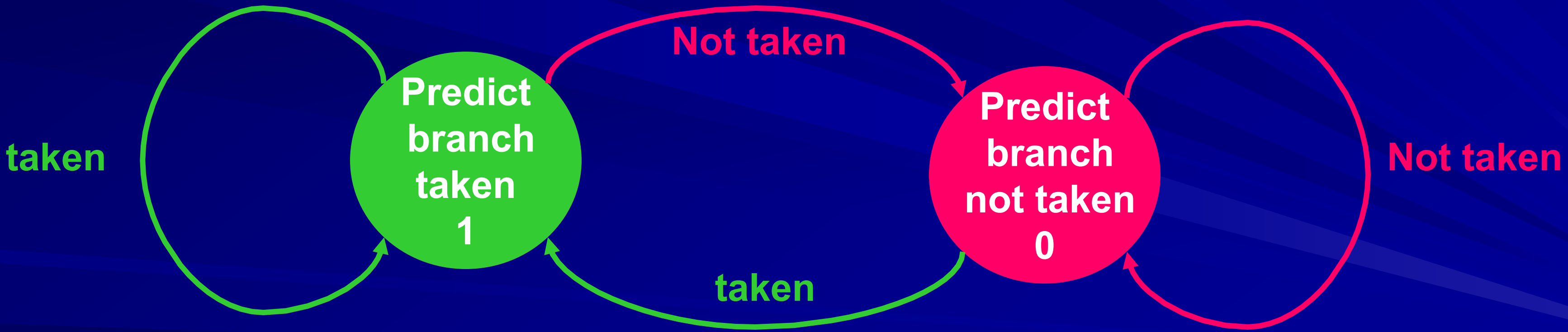
In *dynamic branch prediction*, we look up the address of the instruction to see if the branch was taken last time. If so, we will predict that the branch will be taken again and optimistically fetch the instructions from the branch target rather than the subsequent instructions.

Branch Prediction

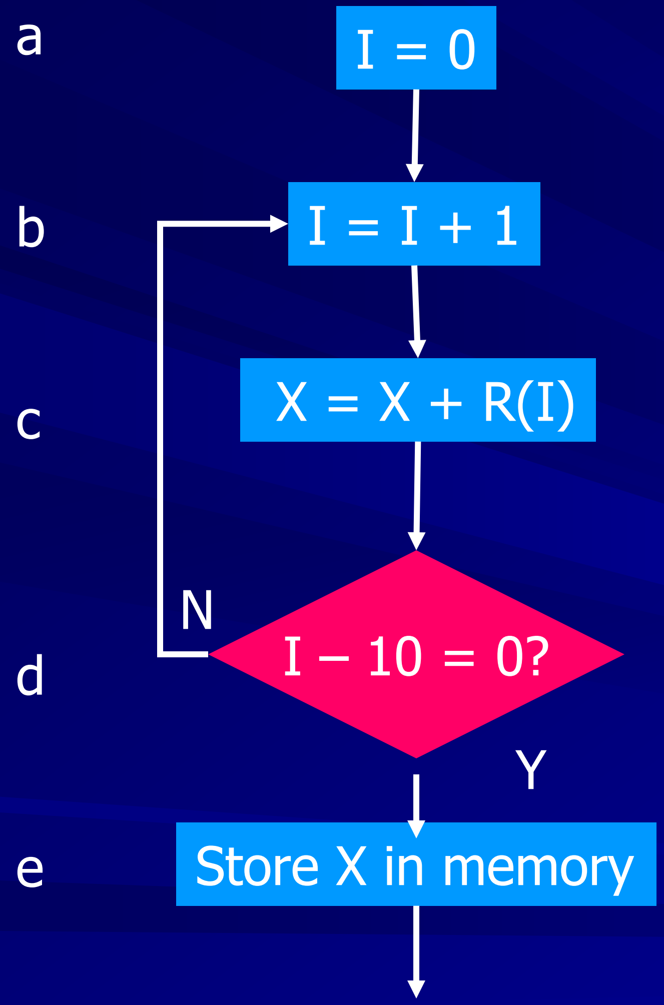

Address of recent branch instructions
Target addresses
History bit(s)
Low-order bits used as index
PC+4
Next PC
0
1
=
Prediction Logic
PC

Branch Prediction

- Useful for program loops.
- A one-bit prediction scheme: a one-bit buffer carries a "history bit" that tells what happened on the last branch instruction
    - History bit = 1, branch was taken
    - History bit = 0, branch was not taken



Branch Prediction for a Loop



Execution of Instruction d

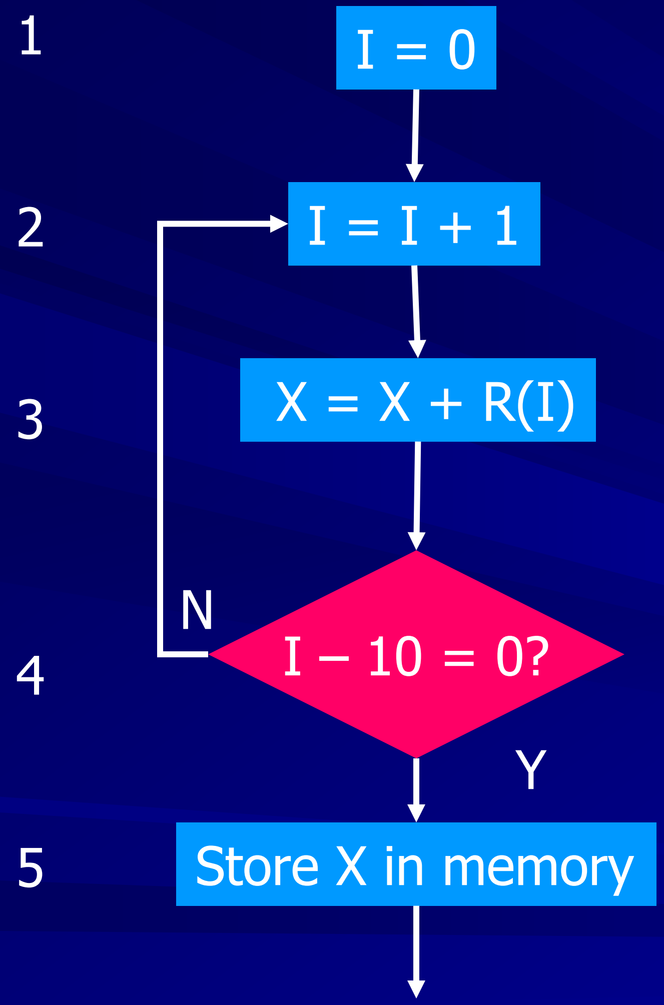
Execu-tion seq.	Old hist. bit	Next instr.			New hist. bit	Predi ction
		Pred.	I	Act.		
1	0	e	1	b	1	Bad
2	1	b	2	b	1	Good
3	1	b	3	b	1	Good
4	1	b	4	b	1	Good
5	1	b	5	b	1	Good
6	1	b	6	b	1	Good
7	1	b	7	b	1	Good
8	1	b	8	b	1	Good
9	1	b	9	b	1	Good
10	1	b	10	e	0	Bad

h.bit = 0 *branch not taken*, h.bit = 1 *branch taken*.

Prediction Accuracy

- One-bit predictor:
  - 2 errors out of 10 predictions
  - Prediction accuracy = 80%
- To improve prediction accuracy, use two-bit predictor:
  - A prediction must be wrong twice before it is changed

Branch Prediction for a Loop



Execution of Instruction 4

Execu-tion seq.	Old Pred. Buf	Next instr.			New pred. Buf	Prediction
		Pred.	I	Act.		
1	10	2	1	2	11	Good
2	11	2	2	2	11	Good
3	11	2	3	2	11	Good
4	11	2	4	2	11	Good
5	11	2	5	2	11	Good
6	11	2	6	2	11	Good
7	11	2	7	2	11	Good
8	11	2	8	2	11	Good
9	11	2	9	2	11	Good
10	11	2	10	5	10	Bad