Pipeline: Branch Prediction

Fall, 2017

These slides are adapted from notes by Dr. David Patterson (UCB)

Static Branch Prediction

For every branch encountered during execution predict whether the branch will be taken or not taken.

Predicting branch not taken:

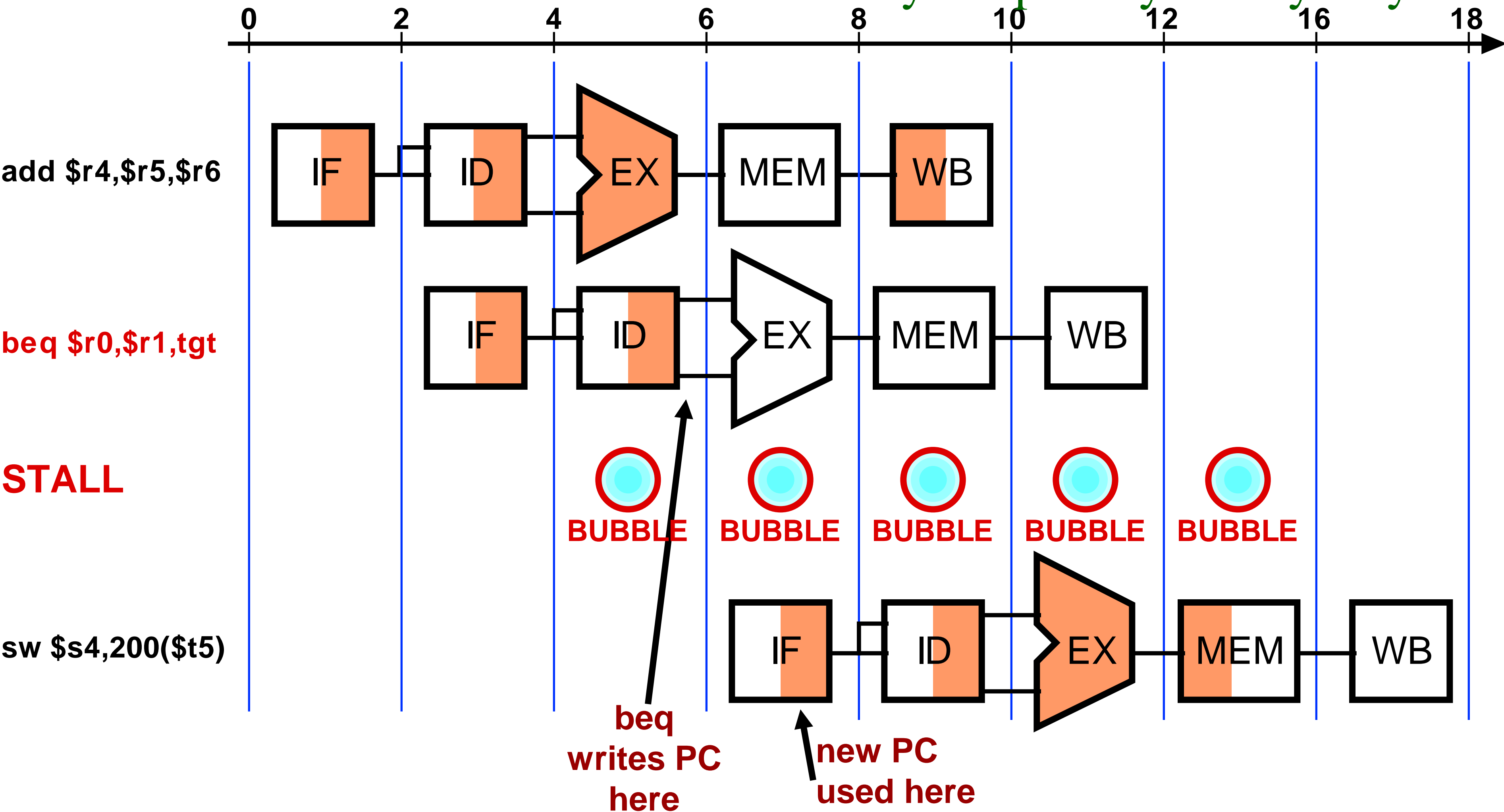
1. Speculatively fetch and execute instructions following the branch
2. If prediction incorrect flush pipeline of speculated instructions
   - Convert these instructions to NOPs by clearing pipeline registers
   - These have not updated memory or registers at time of flush

Predicting branch taken:

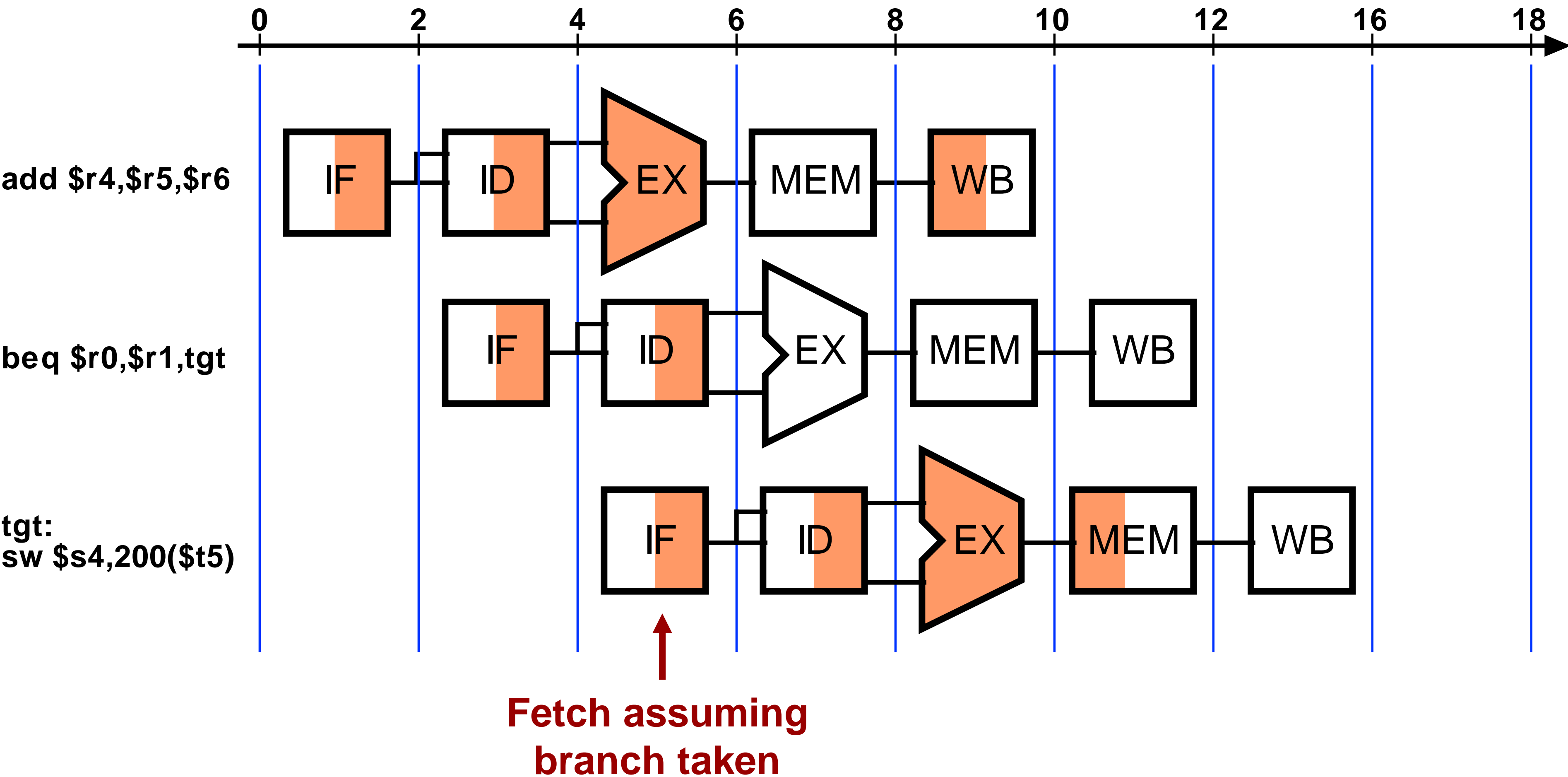
1. Speculatively fetch and execute instructions at the branch target address
2. Useful only if target address known earlier than branch outcome
   - May require stall cycles till target address known
   - Flush pipeline if prediction is incorrect
   - Must ensure that flushed instructions do not update memory/registers

Control Hazard - Stall

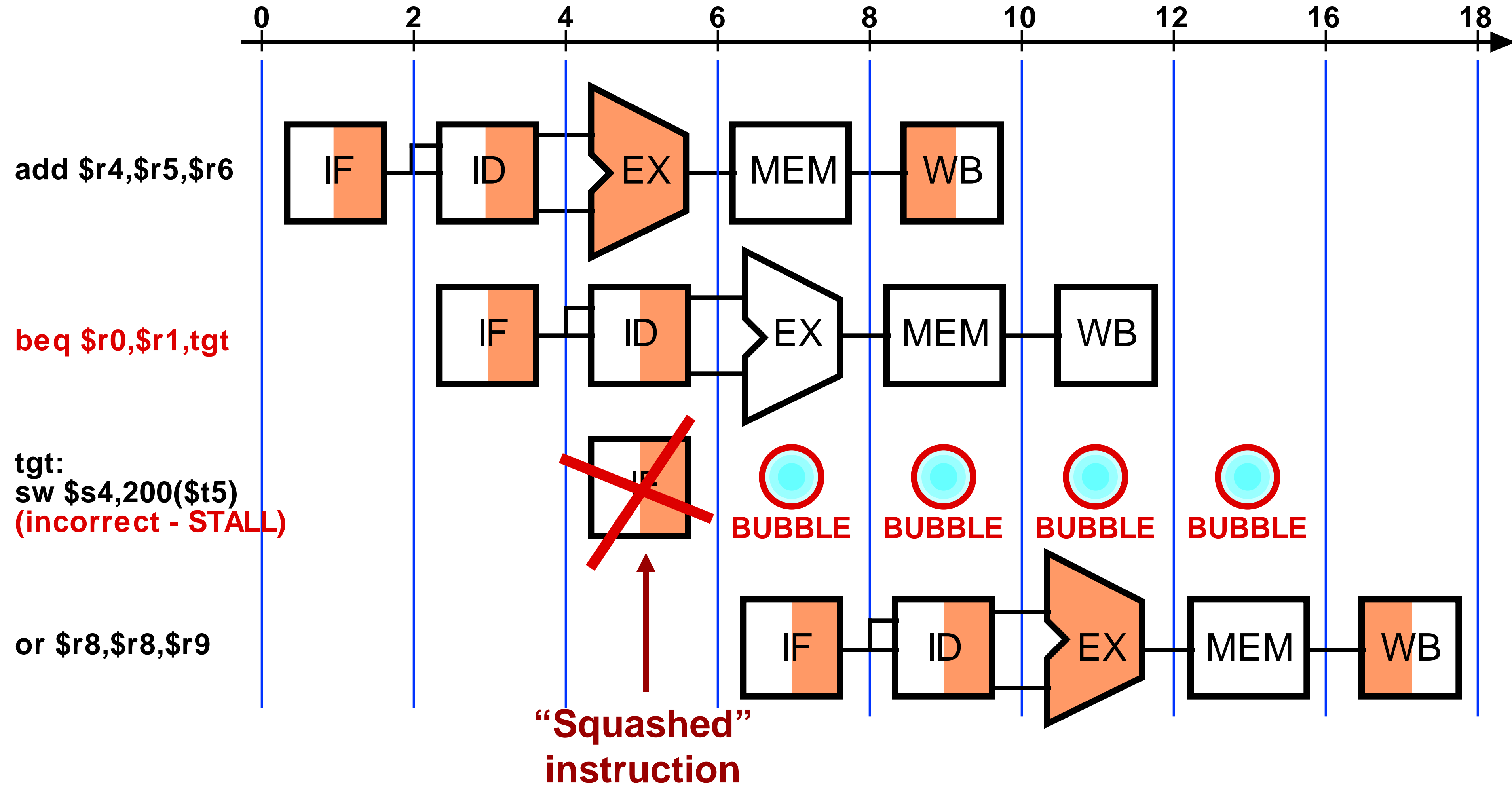
Why the penalty is only 1 cycle?



Control Hazard - Correct Prediction



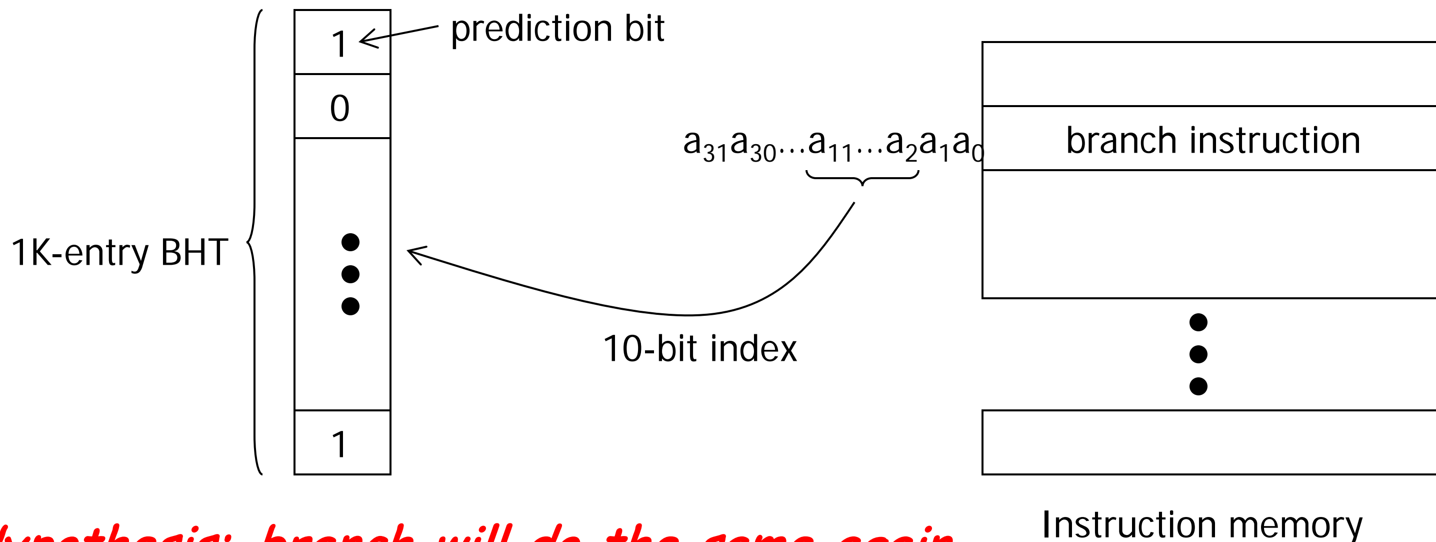
Control Hazard - Incorrect Prediction



1-Bit Branch Prediction

- Branch History Table (BHT): Lower bits of PC address index table of 1-bit values
  - Says whether or not branch taken last time
  - No address check (saves HW, but may not be right branch)
  - If prediction is wrong, invert prediction bit

1 = branch was last taken
0 = branch was last not taken



Hypothesis: branch will do the same again.

1-Bit Branch Prediction

- Example:

  Consider a loop branch that is taken 9 times in a row and then not taken once. What is the prediction accuracy of 1-bit predictor for this branch assuming only this branch ever changes its corresponding prediction bit? (assume the first is a misprediction)

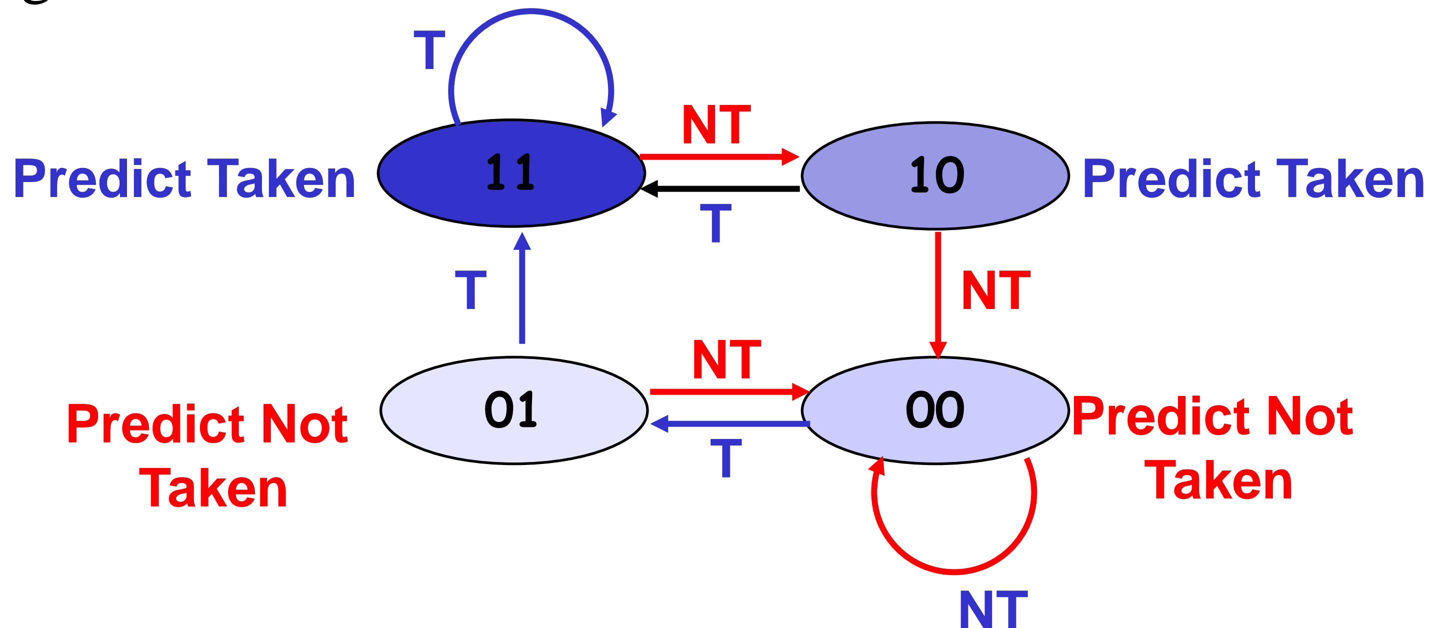
  – Answer: *80%.* Because there are two mispredictions – one on the first iteration and one on the last iteration.

2-Bit Branch Prediction *(Jim Smith, 1981)*

- Solution: 2-bit scheme where change prediction only if get misprediction *twice*

  Red: stop, not taken

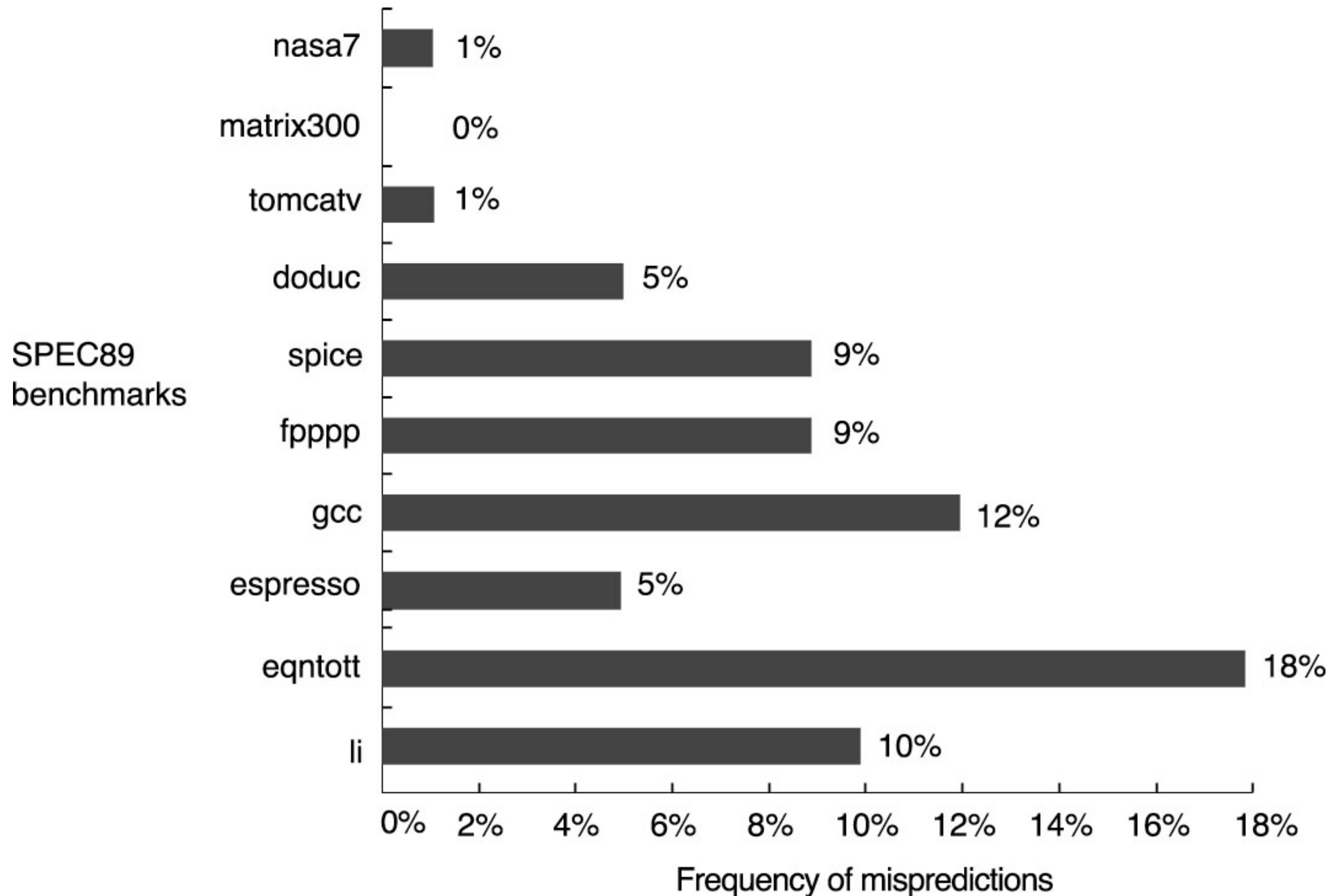
  Blue: go, taken



n-bit Saturating Counter

- Values: $0 \sim 2^n-1$
- When the counter is greater than or equal to one-half of its maximum value, the branch is predicted as taken. Otherwise, not taken.
- Studies have shown that the 2-bit predictors do almost very well, and thus most systems rely on 2-bit branch predictors.

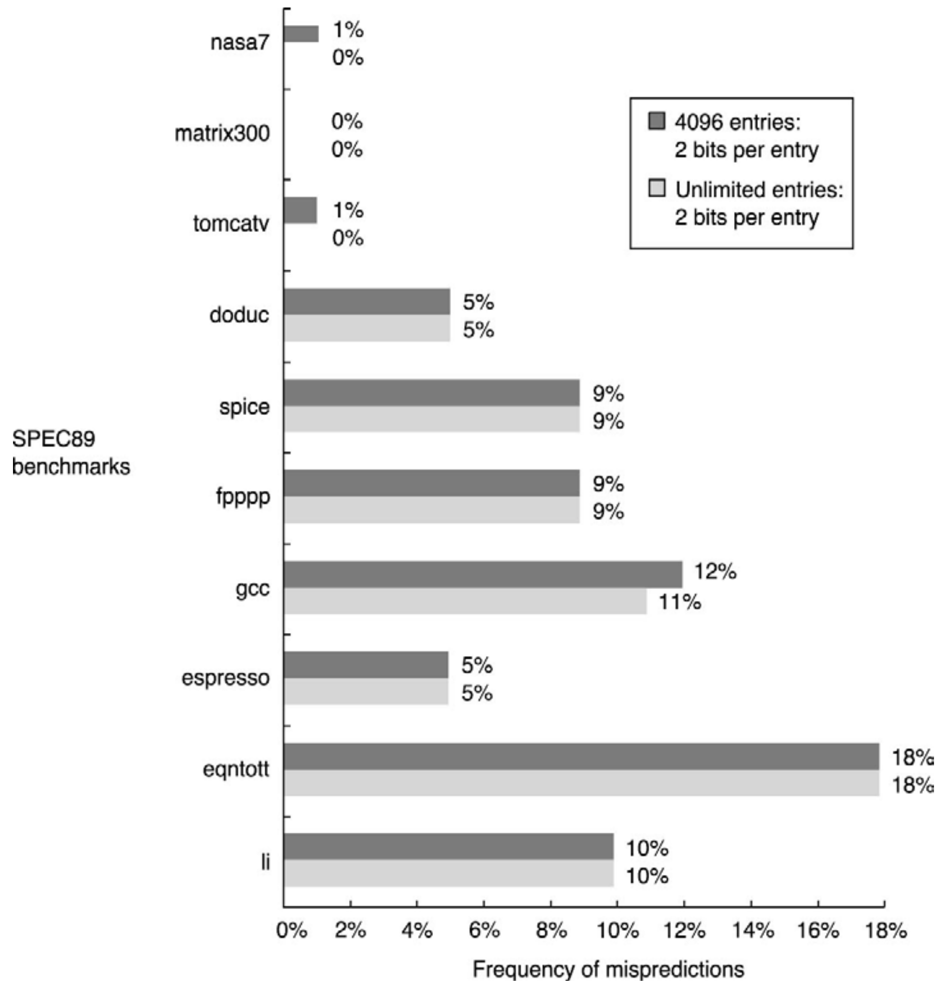
2-bit Predictor Statistics



Prediction accuracy of 4K-entry 2-bit prediction buffer on SPEC89 benchmarks: accuracy is lower for integer programs (gcc, espresso, eqntott, li) than for FP

© 2003 Elsevier Science (USA). All rights reserved.

2-bit Predictor Statistics



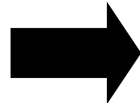
Prediction accuracy of 4K-entry 2-bit prediction buffer vs. "infinite" 2-bit buffer: increasing buffer size from 4K does not significantly improve performance

© 2003 Elsevier Science (USA). All rights reserved.

Control Hazards - Solutions

- Delayed branches – code rearranged by compiler to place independent instruction after every branch (in delay slot).

```
add  $R4,$R5,$R6
beq  $R1,$R2,20
lw $R3,400($R0)
```



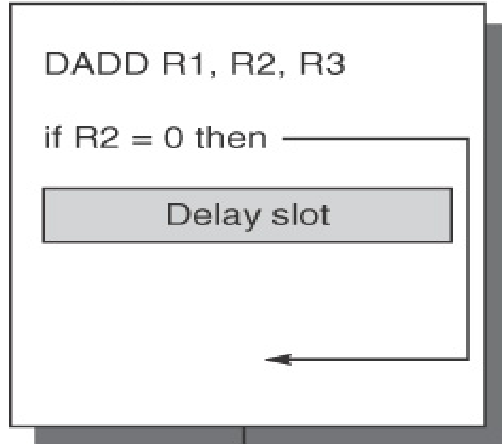
```
beq  $R1,$R2,20
add  $R4,$R5,$R6
lw $R3,400($R0)
```

Basic Idea of Delayed Branch

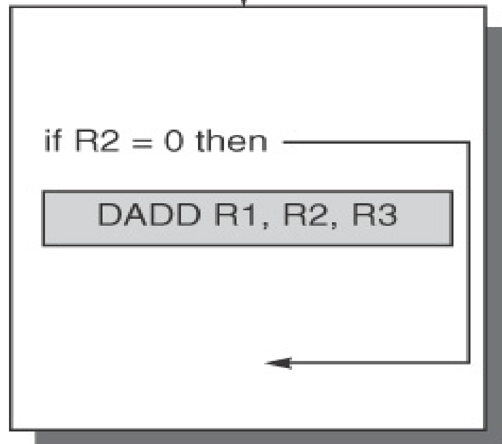
- Find an instruction that can be safely executed whether the branch is taken or not, and execute that instruction.
- When a branch instruction is encountered, the hardware puts the instruction following the branch into the pipe and begins executing it.
- We do not need to worry about whether the branch is taken or not
- we do not need to clear the pipe because no matter whether the branch is taken or not, we know the instruction is safe to execute.
- The compiler promised it would be safe.

Scheduling the Delay Slot

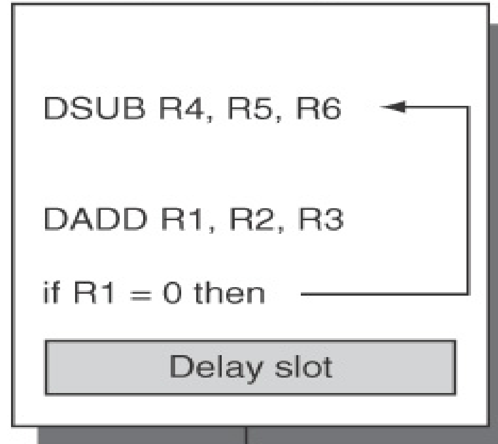
(a) From before



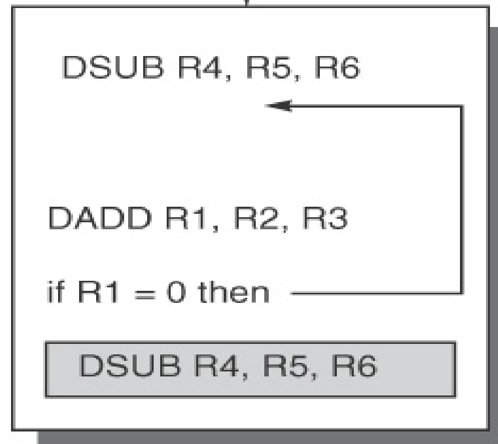
becomes



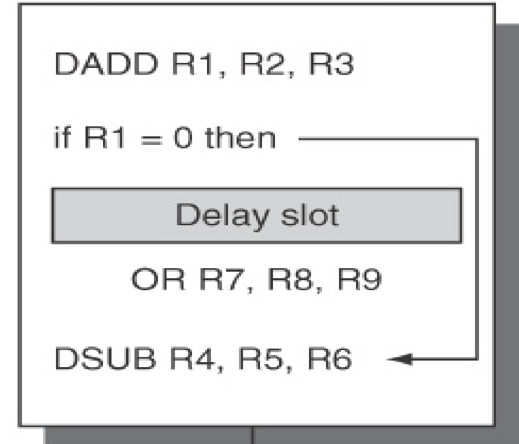
(b) From target



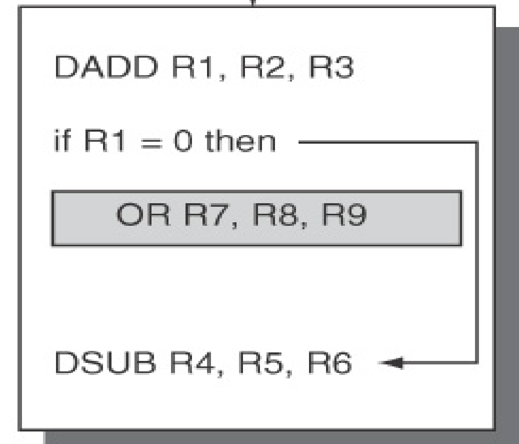
becomes



(c) From fall-through



becomes



© 2007 Elsevier, Inc. All rights reserved.

Understanding Delayed Branch

- **Instruction from before the branch**

Branch must not depend on moved instruction
Always improves performance

- **From branch target**

Must be OK to execute moved instruction when the branch is not taken
Improves performance when branch is taken

- **From fall through**

Must be OK to execute moved instruction when branch is taken
Improves performance when branch is not taken

By OK we mean that the work is wasted, but the program will still execute correctly.

Example Questions (1)

- Limitations on Delayed branch scheduling come from two things. What are they?

- restrictions on the instructions that are scheduled into delay slots.

- ability to predict at compile time whether branch is likely to be taken or not.

Example Question (2)

Does it matter to the instruction in the branch delay slot whether or not the branch is taken?

No. Whether the branch is taken or not, if the instruction is chosen to go along with the requirements set forth, it will execute independent of the branch. It should be chosen to be valid and useful.

Example Question (3)

Whose job is it to make successor instructions valid and useful?

It is the compiler's job.

Example Question (4)

List the three types of instructions that would best fill the branch delay slot and explain how they improve pipeline performance.

-Instruction from before the branch always improves performance.

-Instruction from branch target improves performance when branch taken.

- Instruction from fall through improves performance when branch not taken.

Example Code

```
LOOP:
SUB R3, R3, R1
SW R3, 0(R5)
BNEZ R3 LOOP
SW R4, 0(R6)
```

1. We could safely execute the SW R4, 0(R6) each time through the loop, but that would provide marginal improvement. Why?
2. What is a better solution?

Example Code (Cont.)

- A better solution is to move **SW R3, 0(R5)** right after BNEZ.
- We will still execute it each time through the loop, because it is in the branch-delay slot, the final result will not be altered.
- We get much better overall performance, whether we take the loop or not. Why?
- See next slide for an example run of this code fragment (twice running).

Example Code (Cont.)

```
LOOP:    SUB R3, R3, R1
         BNEZ R3 LOOP
         SW R3, 0(R5)
         SW R4, 0(R6)
```

	1	2	3	4	5	6	7	8	9	10	11
SUB R3,R3,R1	IF	ID	EX	Mem	WB						
BNEZ R3,Loop		IF	ID	EX	Mem	WB					
SW R3, 0(R5)			IF	ID	EX	Mem	WB				
SUB R3,R3,R1				IF	ID	EX	Mem	WB			
BNEZ R3,Loop					IF	ID	EX	Mem	WB		
SW R3, 0(R5)						IF	ID	EX	Mem	WB	
SW R4, 0(R6)							IF	ID	EX	Mem	WB

Example Question (5)

Consider two different 5stage pipeline machines (IF ID EXMEMWB). The first machine resolves branches in the ID stage, uses one branch delay slot, and can fill 40% of the delay slots with useful instructions. The second machine resolves branches in the EX stage and uses a predictnottaken scheme. Assume that the cycle times of the machines are identical. Assume that 25% of the instructions are branches, 30% of branches are taken, and that stalls are due to branches alone.

Which machine is faster? Justify your answer.

Answer to Question 5

1. For the first machine, for 60% of the branches, a cycle is wasted due to the inability to fill the delay slot. The CPI of this machine is 1 + 25% *60%* 1= 1.15.
2. For the second machine, for 30% of the branches, two cycles are wasted due to the unknown target address. The CPI of the second machine is 1 + 25% * 30%* 2 = 1.15.
3. Therefore, the two machines have equal performance.

Summary of Delayed Branch

- This strategy offers improvements regardless of whether we take or do not take the branch.
- The problem is trying to find an instruction that can both be safely executed whether the branch is taken or not, and will still improve performance.
- This is the compiler's job, and so using a branch delay slot makes compilers more complex to program.
- Using this option does cause one shortcoming, if the hardware is changed so that a delay-branch slot is no longer used, all the old programs will no longer work.

Summary - Control Hazard Solutions

- Stall - stop fetching instr. until result is available
  - Significant performance penalty
  - Hardware required to stall
- Predict - assume an outcome and continue fetching (undo if prediction is wrong)
  - Performance penalty only when guess wrong
  - Hardware required to "squash" instructions
- Delayed branch - specify in architecture that following instruction is always executed
  - Compiler re-orders instructions into delay slot
  - Insert "NOP" (no-op) operations when can't use (~50%)
  - This is how original MIPS worked

Summary - Pipelining Overview

- Pipelining increase <u>throughput</u> (but not latency)
- Hazards limit performance
  - Structural hazards
  - Control hazards
  - Data hazards