CS / EE 320
Computer Organization and Assembly Language
Lecture 19
Spring 2025

Shahid Masud

Topics: Examples of Pipelining Hazards and Performance, Branch Prediction in Pipelining

Topics

- Removal of Data Hazards - Examples

- Control Hazards in Pipelined MIPS

- Branch Predictors
  - Static – Compiler level
    - Branch Always Taken, Never Taken, Forward Taken, Backward Taken, etc.
    - Some improvement over no prediction

- Dynamic Branch Predictors

- Efficient Pipeline utilization in Superscalar Architectures

- Pipeline Improvements through Loop Unrolling

QUIZ
after Eid
Break

Question 1 Eliminate Data Hazard using NOP

Given Code

lw $t0, 0($s0)

lw $t1, 0($t0)

lw $t2, 0($t1)

addi $t2, $t2, 5

sw $t2, 0($t1)

addi $s0, $s0, 4

add $t6, $t4, $t5

Show Dependence

lw $t0, 0($s0)

lw $t1, 0($t0)

lw $t2, 0($t1)

addi $t2, $t2, 5

sw $t2, 0($t1)

addi $s0, $s0, 4

add $t6, $t4, $t5

With NOP for hazard removal

lw $t0, 0($s0)

NOP

NOP

lw $t1, 0($t0)

NOP

NOP

lw $t2, 0($t1)

NOP

NOP

addi $t2, $t2, 5

NOP

NOP

sw $t2, 0($t1)

addi $s0, $s0, 4

add $t6, $t4, $t5

Question 2 Eliminate Data Hazard using NOP

Given Code

addi $sp, $sp-8

sw $s0, 0($sp)

sw $ra, 4($sp)

add $s0, $t0, $t7

sub $a0, $s0, $t3

lw $s0, 0($sp)

lw $ra, 4($sp)

addi $sp, $sp, 8

Jr $ra

Show Dependence

addi $sp, $sp-8

sw $s0, 0($sp)

sw $ra, 4($sp)

add $s0, $t0, $t7

sub $a0, $s0, $t3

lw $s0, 0($sp)

lw $ra, 4($sp)

addi $sp, $sp, 8

Jr $ra

With NOP

addi $sp, $sp-8

NOP

NOP

sw $s0, 0($sp)

sw $ra, 4($sp)

add $s0, $t0, $t7

NOP

NOP

sub $a0, $s0, $t3

lw $s0, 0($sp)

lw $ra, 4($sp)

addi $sp, $sp, 8

NOP

Jr $ra

Given Code

```
# a=b+e; c=b+f

lw $t1, 0($t0)
lw $t2, 4($t0)
add $t3, $t1, $t2
sw $t3, 12($t0)
lw $t4, 8($t0)
add $t5, $t1, $t4
sw $t5, 16($t0)
```

Show Dependence

```
lw $t1, 0($t0)
lw $t2, 4($t0)          ──→ Stall
add $t3, $t1, $t2
sw $t3, 12($t0)
lw $t4, 8($t0)          ──→ Stall
add $t5, $t1, $t4
sw $t5, 16($t0)
```
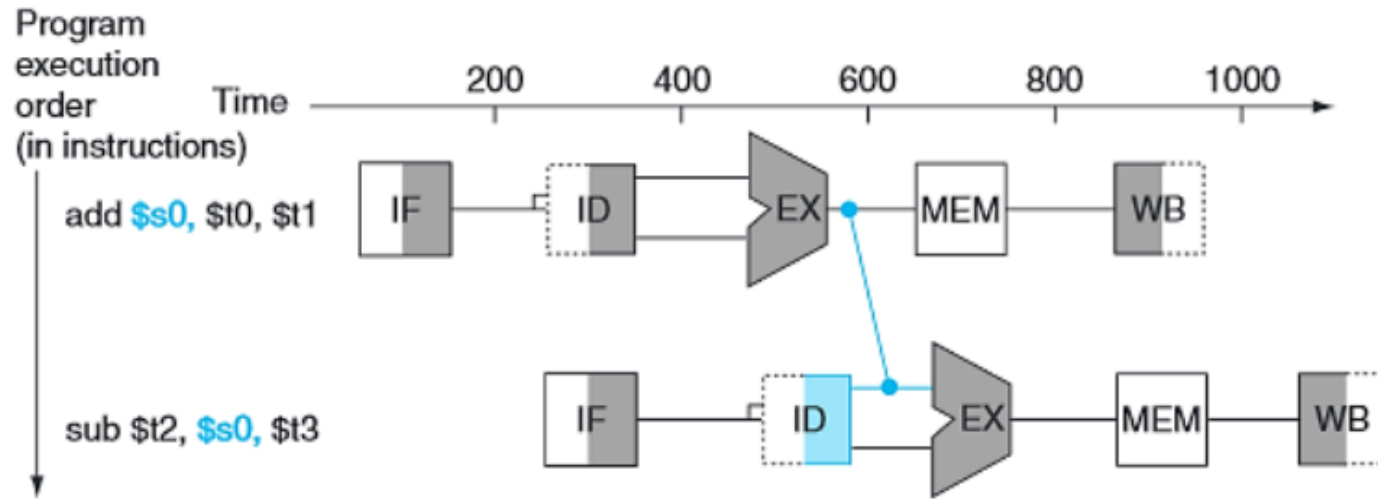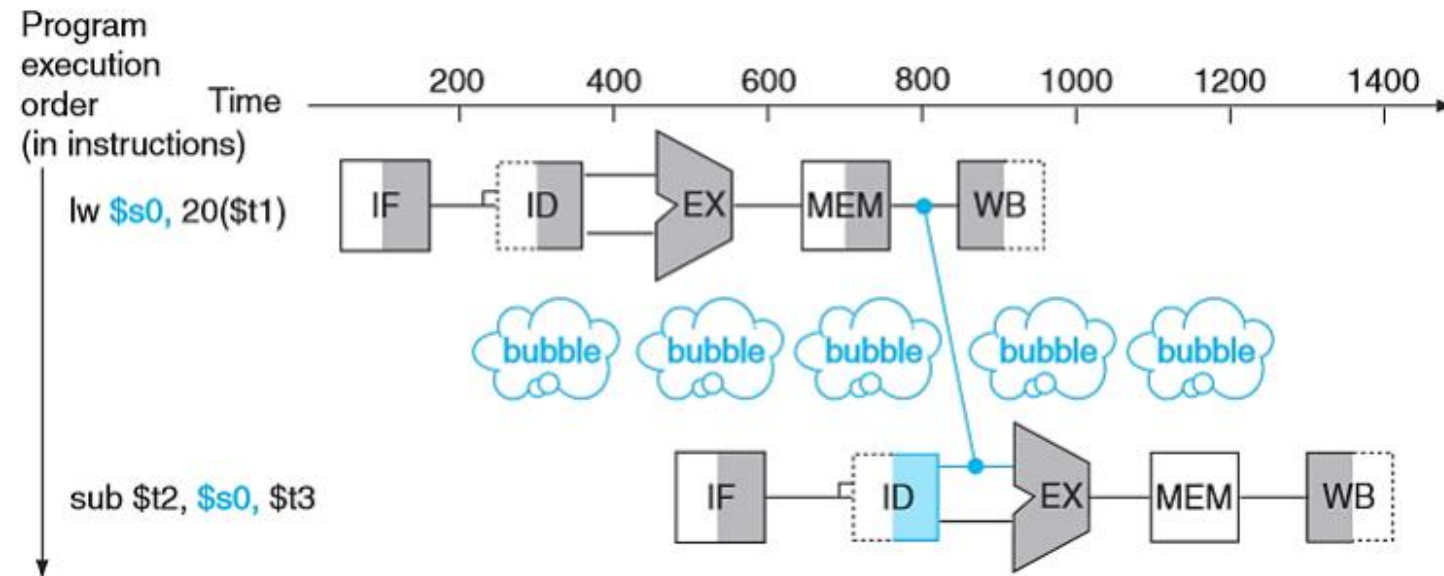
With Reordering

```
lw $t1, 0($t0)
lw $t2, 4($t0)
lw $t4, 8($t0)
add $t3, $t1, $t2
sw $t3, 12($t0)
# lw $t4, 8($t0)
add $t5, $t1, $t4
sw $t5, 16($t0)
```
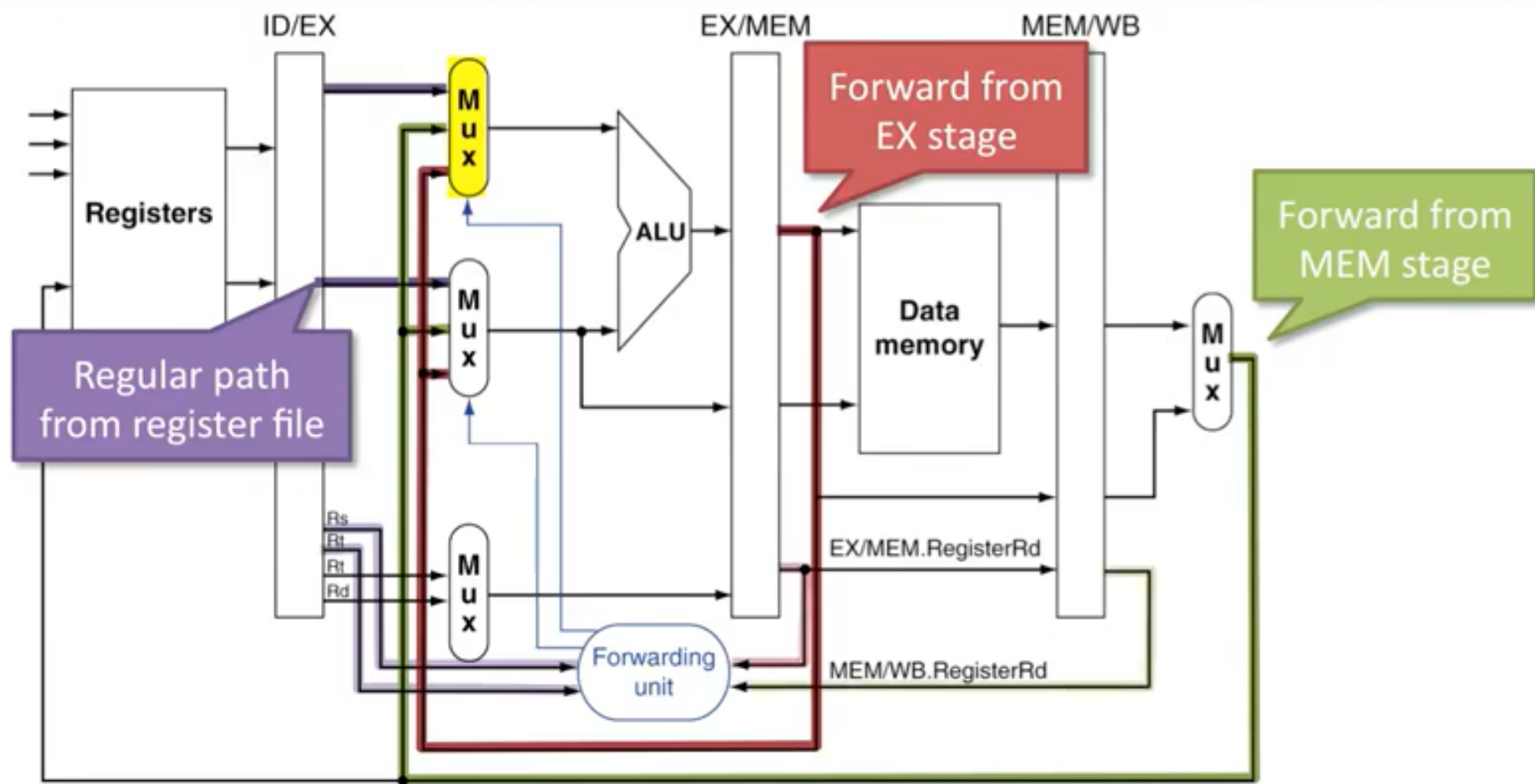
Forwarding Types

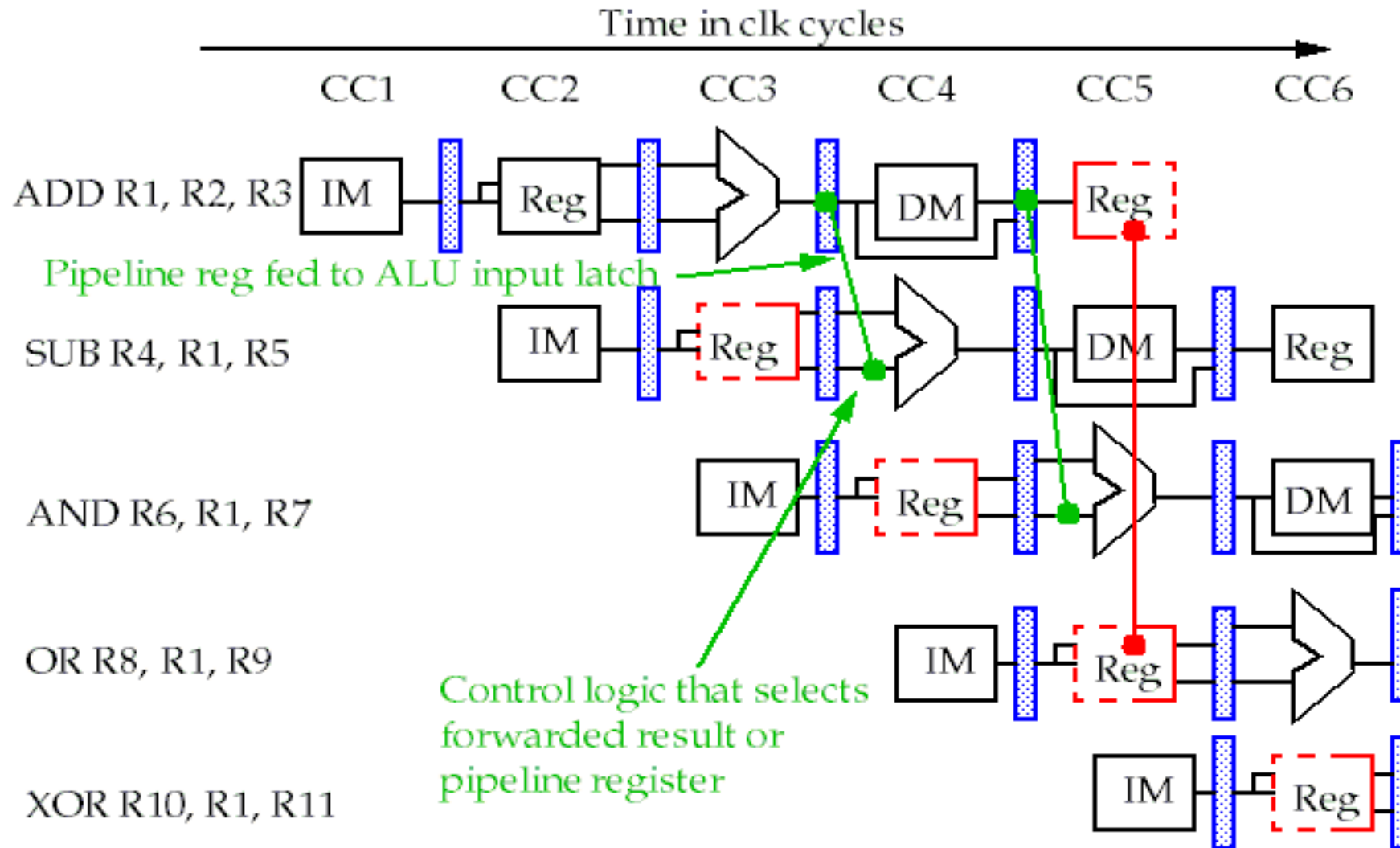**Program execution order (in instructions)**

add $s0, $t0, $t1

sub $t2, $s0, $t3

ALU Forwarding

Memory Forwarding

Program execution order (in instructions)

lw $s0, 20($t1)

sub $t2, $s0, $t3

Forwarding Paths in MIPS Pipeline

Forwarding Possibility

Time in clk cycles

CC1   CC2   CC3   CC4   CC5   CC6

ADD R1, R2, R3

Pipeline reg fed to ALU input latch

SUB R4, R1, R5

AND R6, R1, R7

OR R8, R1, R9

Control logic that selects forwarded result or pipeline register

XOR R10, R1, R11

Forwarding:
1. **After Mem**
2. **After ALU**
3. **RF R/W**

Given Code	Show Dependence	With NOP
lw $t0, 0($s0)	lw $t0, 0($s0)	lw $t0, 0($s0)
lw $t1, 0($t0)	lw $t1, 0($t0)	NOP
lw $t2, 0($t1)	lw $t2, 0($t1)	lw $t1, 0($t0)
addi $t2, $t2, 5	addi $t2, $t2, 5	NOP
sw $t2, 0($t1)	sw $t2, 0($t1)	lw $t2, 0($t1)
addi $s0, $s0, 4	addi $s0, $s0, 4	NOP
add $t6, $t4, $t5	add $t6, $t4, $t5	addi $t2, $t2, 5
		sw $t2, 0($t1)
		addi $s0, $s0, 4
		add $t6, $t4, $t5

Instruction Execution on Pipeline Diagram / Table

If we are using a pipelined processor with forwarding, we have the following stages executing in each cycle:

cycle	1	2	3	4	5	6	7	8	9	10	11	12	13
lw $t1,0($t0)	IF	ID	EX	MEM	WB								
lw $t2,4($t0)		IF	ID	EX	MEM	WB							
add $t3,$t1,$t2				IF	ID	EX	MEM	WB					
sw $t3,12($t0)					IF	ID	EX	MEM	WB				
lw $t4,8($t0)						IF	ID	EX	MEM	WB			
add $t5,$t1,$t4								IF	ID	EX	MEM	WB	
sw $t5,16($t0)									IF	ID	EX	MEM	WB

Instruction Execution with Forwarding & Reordering

The reordering allows us to execute the program in two fewer cycles than before.

cycle	1	2	3	4	5	6	7	8	9	10	11	12	13
lw $t1,0($t0)	IF	ID	EX	MEM	WB								
lw $t2,4($t0)		IF	ID	EX	MEM	WB							
lw $t4,8($t0)			IF	ID	EX	MEM	WB						
add $t3,$t1,$t2				IF	ID	EX	MEM	WB					
sw $t3,12($t0)					IF	ID	EX	MEM	WB				
add $t5,$t1,$t4						IF	ID	EX	MEM	WB			
sw $t5,16($t0)							IF	ID	EX	MEM	WB		

Question 5 Eliminate Data Hazard with **Forwarding & NOP**

Forwarding:

1. **After Mem**
2. **After ALU**
3. **RF R/W**

Given Code

```
lw $t0, 0($a0)
add $t2, $t0, $a1
addi $t2, $zero, 4
addi $t1, $zero, 2
sll $t1, $t1, 2
add $t1, $a0, $t1
sw $t2, 0($t1)
jr $ra
```

Show Dependence

```
lw $t0, 0($a0)
add $t2, $t0, $a1
addi $t2, $zero, 4
addi $t1, $zero, 2
sll $t1, $t1, 2
add $t1, $a0, $t1
sw $t2, 0($t1)
jr $ra
```

With NOP

```
lw $t0, 0($a0)
NOP
add $t2, $t0, $a1
addi $t2, $zero, 4
addi $t1, $zero, 2
sll $t1, $t1, 2
add $t1, $a0, $t1
sw $t2, 0($t1)
jr $ra
```

Only One NOP, With Forwarding

4.16 In this exercise, we examine how pipelining affects the clock cycle time of the processor. Problems in this exercise assume that individual stages of the datapath have the following latencies:

IF	ID	EX	MEM	WB
250ps	350ps	150ps	300ps	200ps

Also, assume that instructions executed by the processor are broken down as follows:

ALU/Logic	Jump/Branch	Load	Store
45%	20%	20%	15%

4.16.1 [5] <§4.6> What is the clock cycle time in a pipelined and non-pipelined processor?

4.16.2 [10] <§4.6> What is the total latency of an lw instruction in a pipelined and non-pipelined processor?

4.16.3 [10] <§4.6> If we can split one stage of the pipelined datapath into two new stages, each with half the latency of the original stage, which stage would you split and what is the new clock cycle time of the processor?

4.16.4 [10] <§4.6> Assuming there are no stalls or hazards, what is the utilization of the data memory?

4.16.5 [10] <§4.6> Assuming there are no stalls or hazards, what is the utilization of the write-register port of the "Registers" unit?

4.9 In this exercise, we examine how data dependences affect execution in the basic 5-stage pipeline described in Section 4.5. Problems in this exercise refer to the following sequence of instructions:

```
or r1,r2,r3
or r2,r1,r4
or r1,r1,r2
```

Also, assume the following cycle times for each of the options related to forwarding:

Without Forwarding	With Full Forwarding	With ALU-ALU Forwarding Only
250ps	300ps	290ps

4.9.1 [10] <§4.5> Indicate dependences and their type.

4.9.2 [10] <§4.5> Assume there is no forwarding in this pipelined processor. Indicate hazards and add nop instructions to eliminate them.

4.9.3 [10] <§4.5> Assume there is full forwarding. Indicate hazards and add NOP instructions to eliminate them.

4.9.4 [10] <§4.5> What is the total execution time of this instruction sequence without forwarding and with full forwarding? What is the speedup achieved by adding full forwarding to a pipeline that had no forwarding?

4.9.5 [10] <§4.5> Add nop instructions to this code to eliminate hazards if there is ALU-ALU forwarding only (no forwarding from the MEM to the EX stage).

4.9.6 [10] <§4.5> What is the total execution time of this instruction sequence with only ALU-ALU forwarding? What is the speedup over a no-forwarding pipeline?

Template of Pipelining Timing Diagram

Cycles →	Cyc 1	Cyc 2	Cyc 3	Cyc 4	Cyc 5	Cyc 6	Cyc 7	Cyc 8	Cyc 9	Cyc 10	Cyc 11	Cyc 12	Cyc 13	Cyc 14	Cyc 15
Instructions ↓															
INSTR 1	IF	ID	EX	MEM	WB										

Pipelining and Branch Instructions

Branch Penalty

When predicting that a branch is not taken, we proceed as normal. If the branch is not taken, there is no issue.
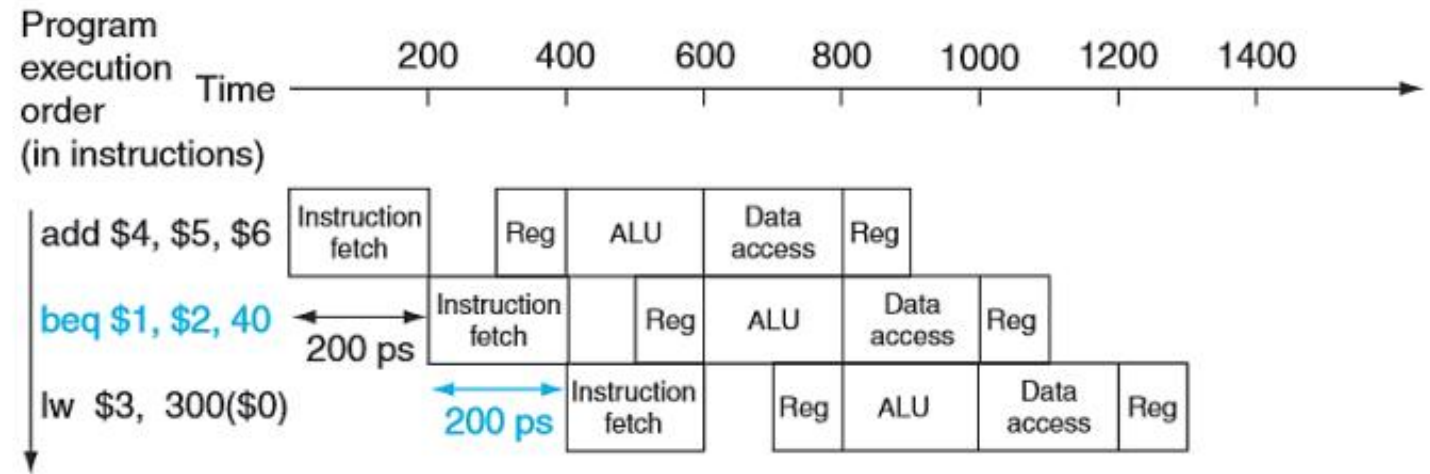
If the branch is taken, however, we incur a stalling penalty. This penalty can vary but even in a highly optimized pipeline we will have to essentially "stall" for a cycle.
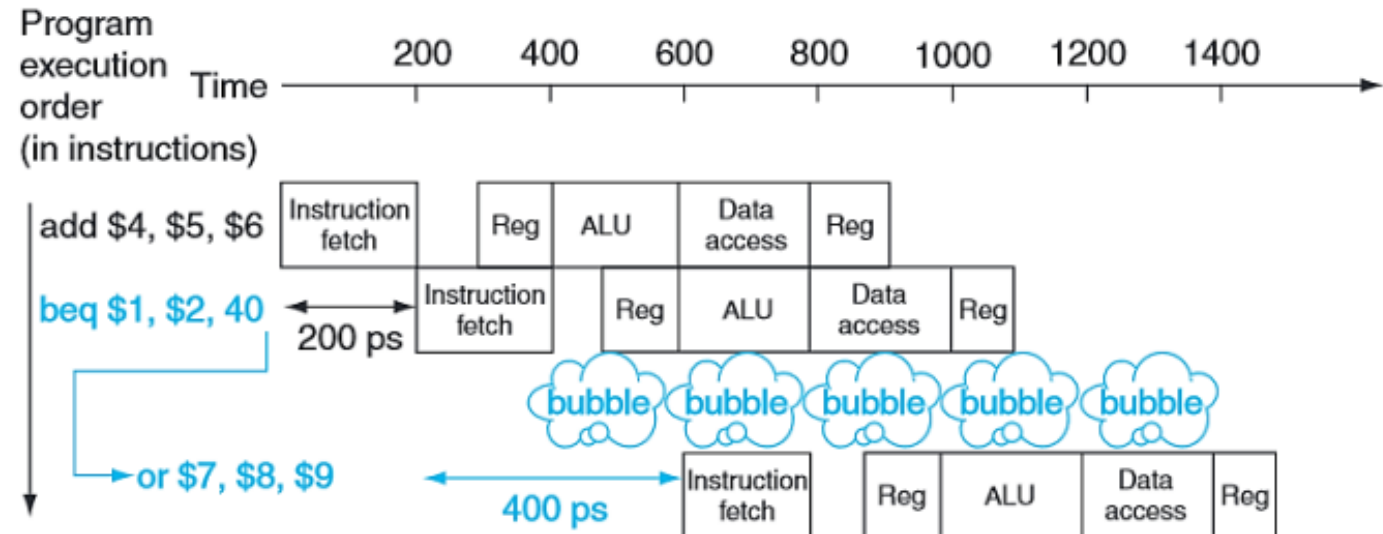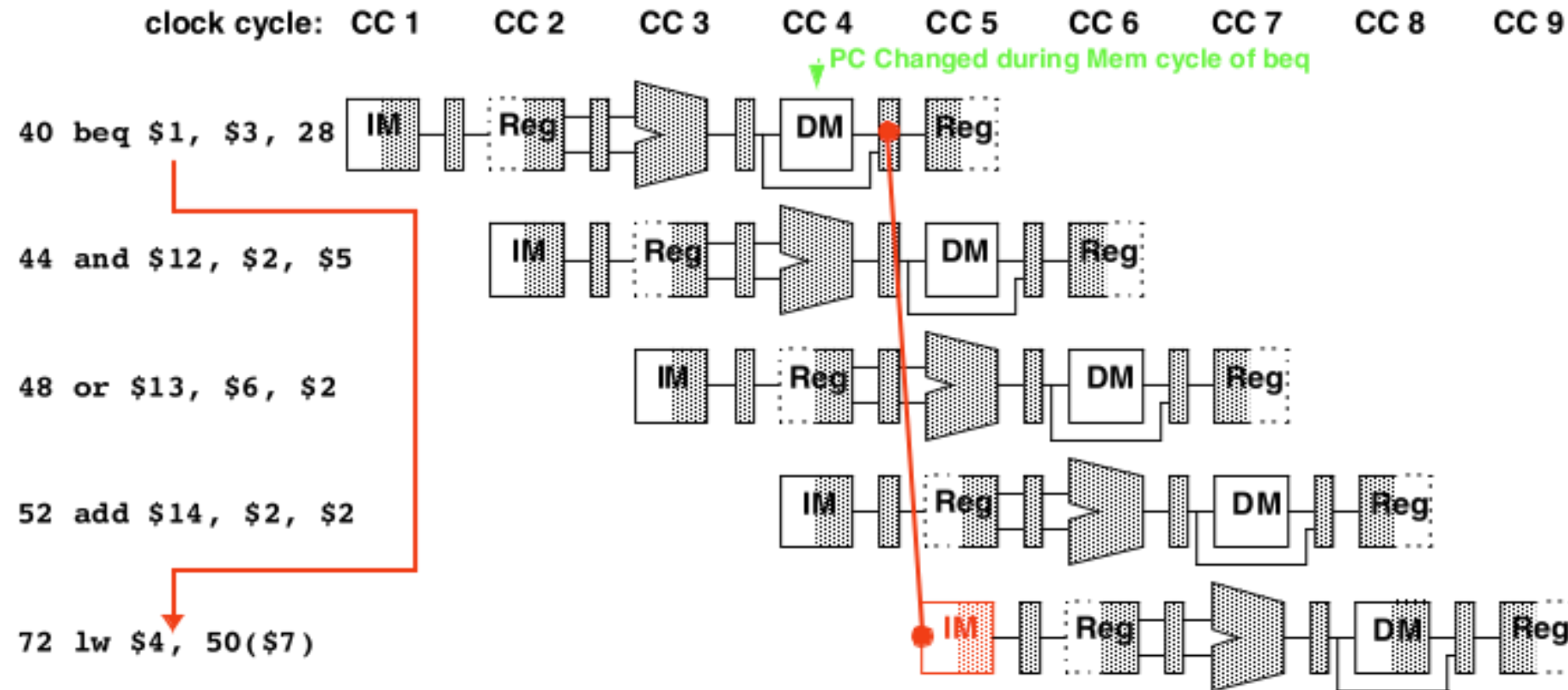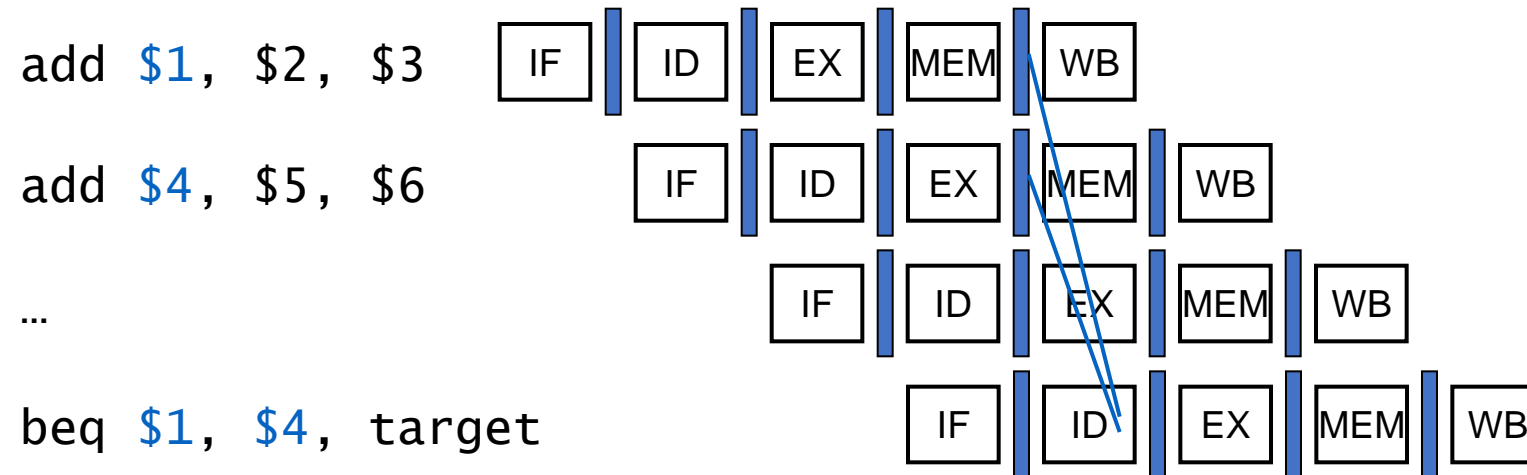
- **If branch condition true, must skip 44, 48, 52**
  - **But, these have already started down the pipeline**
  - **They will complete unless we do something about it**
- **How do we deal with this?**
  - **We'll consider 2 possibilities**

- If a comparison register is a destination of 2nd or 3rd preceding ALU instruction

add $1, $2, $3

add $4, $5, $6

…

beq $1, $4, target

- Can resolve using forwarding

- If a comparison register is a destination of preceding ALU instruction or 2nd preceding load instruction
  - Need 1 stall cycle
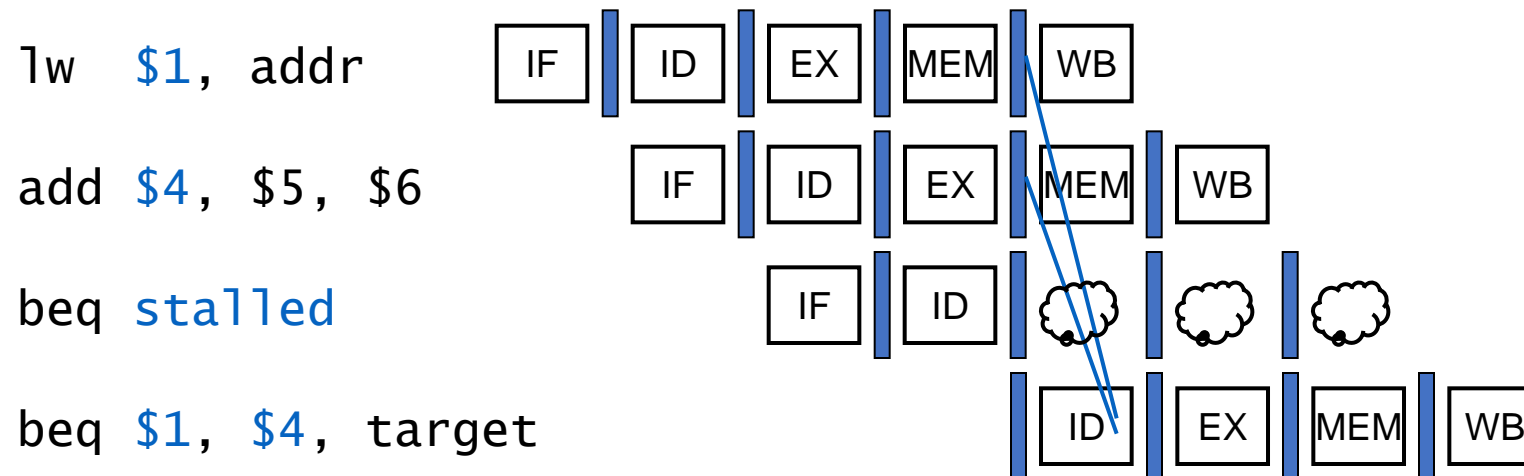
# Data Hazards for Branches

- If a comparison register is a destination of immediately preceding load instruction
  - Need 2 stall cycles


```
lw  $1, addr        IF    ID    EX   MEM   WB

beq stalled               IF    ID

beq stalled                     ID

beq $1, $0, target                    ID    EX   MEM   WB
```
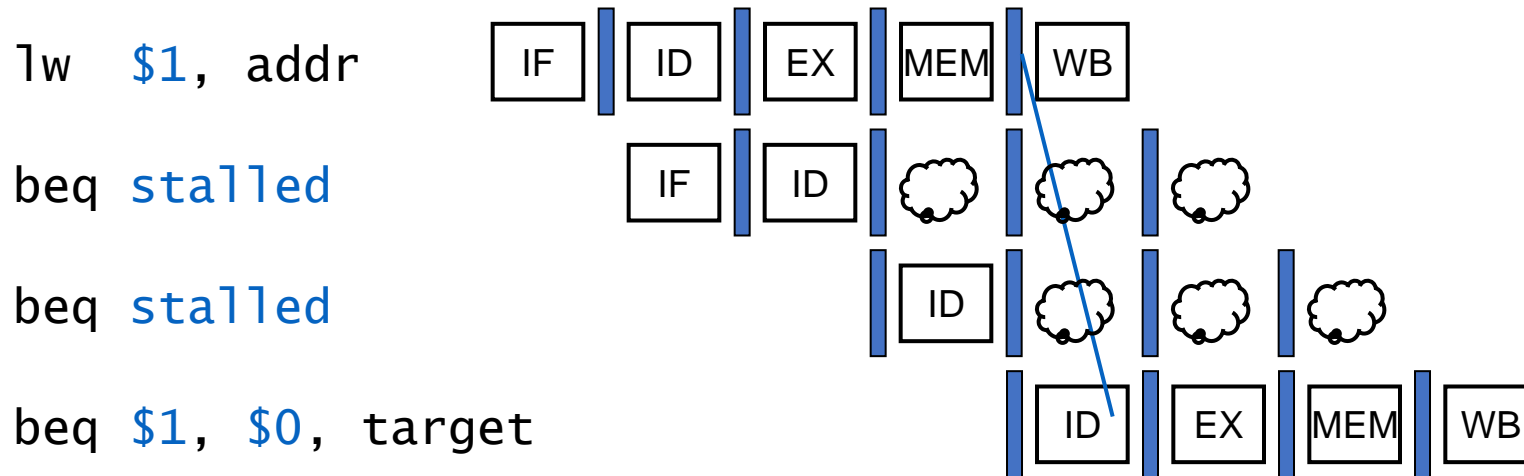
- **Branch not taken**
  - **Still must wait 3 cycles**
  - **Time lost**
  - **Could have spent CCs fetching, decoding next instructions**


Branch Penalty

Dealing with branch hazards – Some Prediction
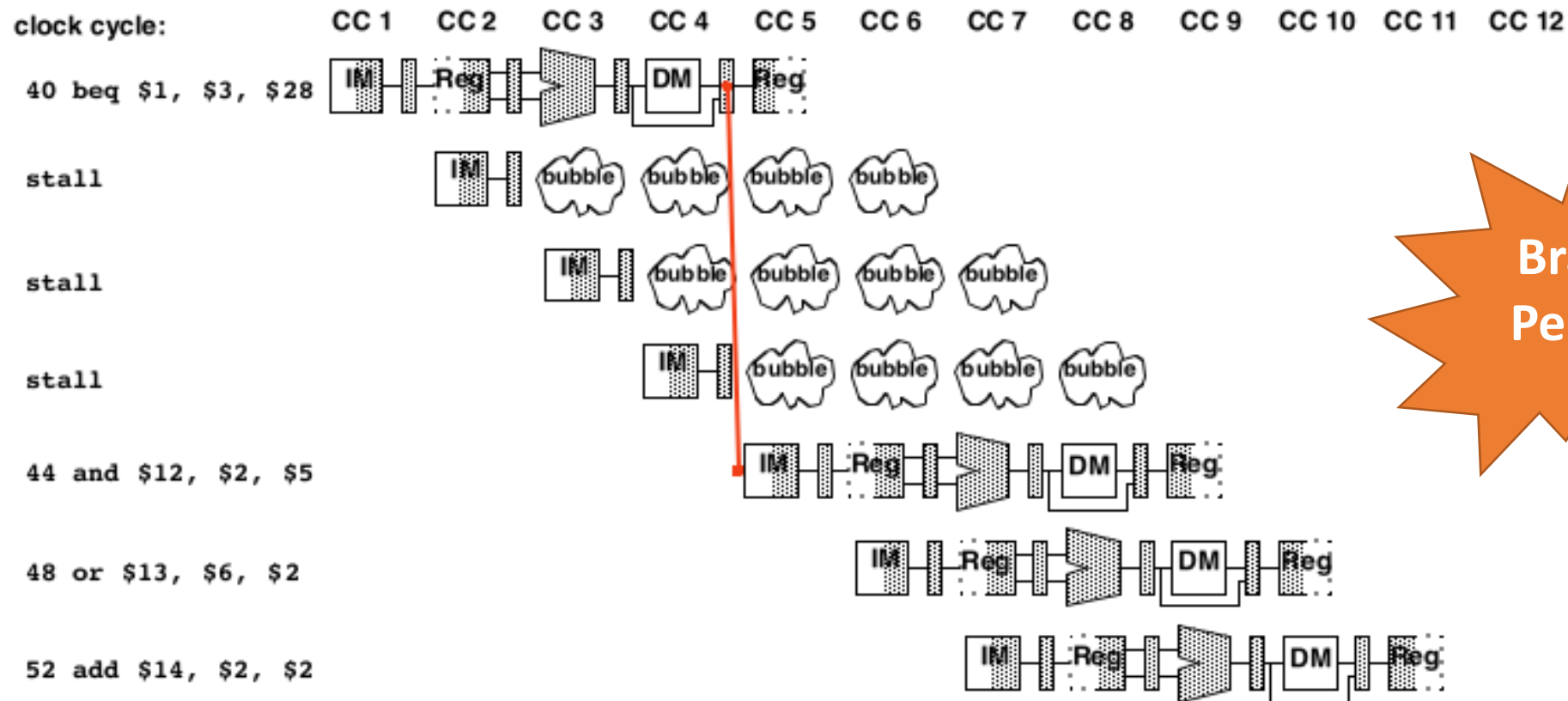
- **On average, branches are taken ½ the time**
  - **If branch not taken…**
    - **Continue normal processing**
  - **Else, if branch is taken…**
    - **Need to flush improper instruction from pipeline**

- **One approach:**
  - **Always assume branch will NOT be taken**
    - **Cuts overall time for branch processing in ½**
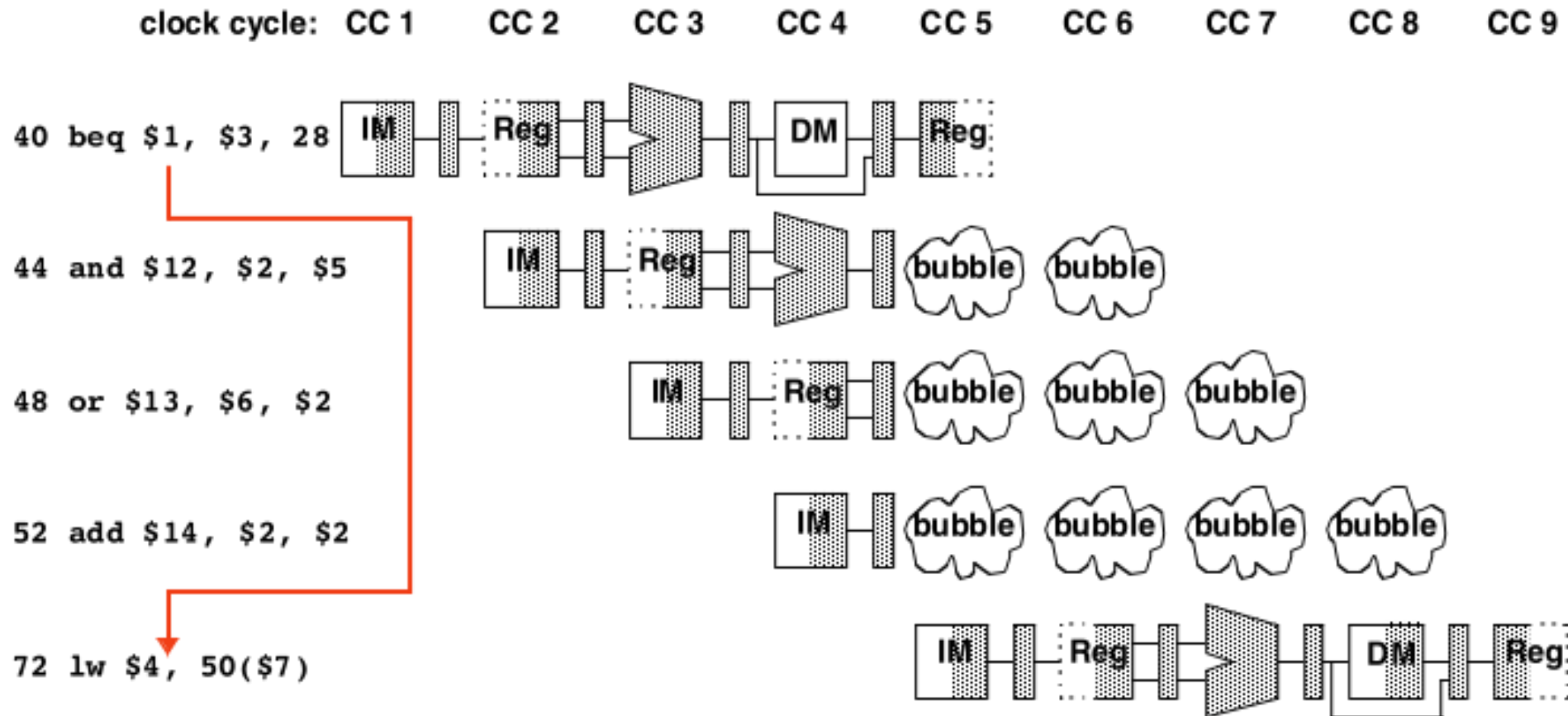 - **If prediction is incorrect, just flush the pipeline**

- **Execution proceeds normally – no penalty**
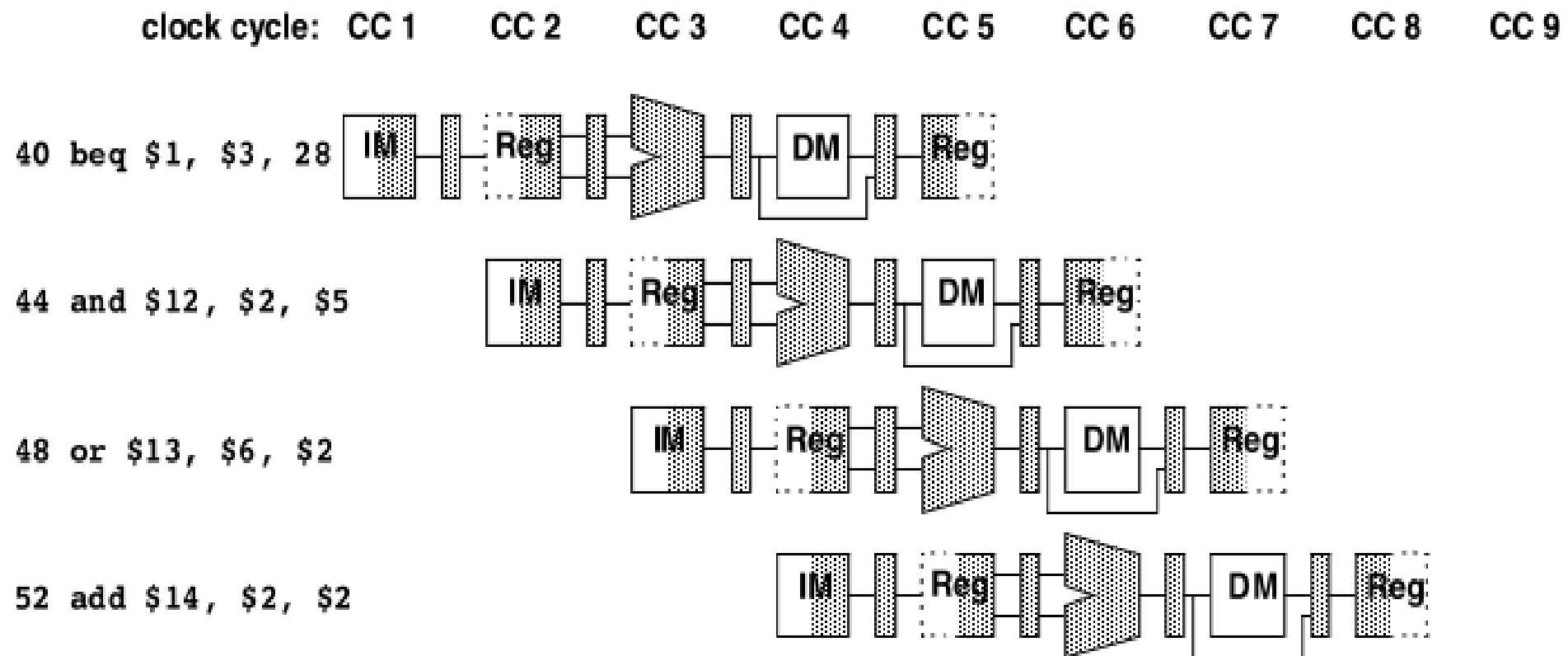
- **Bubbles injected into 3 stages during cycle 5**

Fundamental lesson(s)

- Pipelining changes the timing as to when the result(s) of an instruction are produced

- Additional HW is needed to ensure that the correct program results are produced while maintaining the speedups offered from the introduction of pipelining

- We must also account for the efficient pipelining of control instructions (e.g. beq) to preserve performance gains *and* program correctness

Branch Prediction

Introduction to Branch Prediction

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*.

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.

Speculation vs Branch Prediction

- **Definition:** Instead of adding NOP bubble, try running potential instructions instead

- **Speculation:** Run an instruction that may not be the right one to be executing

- **Branch Prediction - Choose which possible instruction to run:**
  - Static Branch Prediction – Fixed prediction scheme
  - Dynamic Branch Prediction – Prediction depends upon recent Branch decisions

Branch Prediction

- **Guess possible outcome of Branch Instruction**
  - **Technique is called "branch predicting"; needs 2 parts:**
    - **"Predictor" to guess where / if instruction will branch**
      - **(and to where)**
    - **"Recovery Mechanism":**
      - **i.e. a way to fix your mistake**
  - **Prior strategy: (Static)**
    - **Predictor: always guess branch never taken**
    - **Recovery: flush instructions if branch taken**
  - **Alternative: accumulate info. in IF stage as to… (Dynamic)**
    - **Whether or not for any particular PC value a branch was taken next**
    - **To where it is taken**
    - **How to update with information from later stages**

Static Branch Prediction

- Choose a method of Branch Prediction independent of the code running "Static Branch Prediction":
  - Assume Branches are Never Taken
  - Assume Branches are Always Taken
  - Assume Forward Branches are Taken and Backward Branches are Not Taken

- **Speculatively load next instruction** as per above scheme
- Properties of overall code and program need to be studied to come up with a good Static Prediction

Dynamic Branch Prediction

- Choose a method for speculative instruction based on **evaluating the code** that is running:
  - **(m,n) Branch Predictors**
  - **Tournament Branch Predictors**
  - **Percepton Branch Predictors**
- (m,n) Predictor looks at some past Branch Decisions
- Tournament Predictor can select between (m,n) and Static Predictor based on some algorithm, also looking at code
- Perceptron Predictor in modern hardware looks at surrounding code (before and after branch) and how it could impact Branch Decisions

Dynamic Branch Predictors Requirement

- 1-Bit and 2-Bit Branch Predictors

- (m,n) branch predictors that consider multiple loops in programs

- Branch History Table

- Cache memory for Branch Target Addresses

- State Machines of 1-Bit and 2-Bit Predictors

Branch Prediction Buffer

A branch prediction buffer is a small memory indexed by the lower portion of the address of the branch instruction. The memory simply contains one bit indicating whether the branch was taken last time or not.
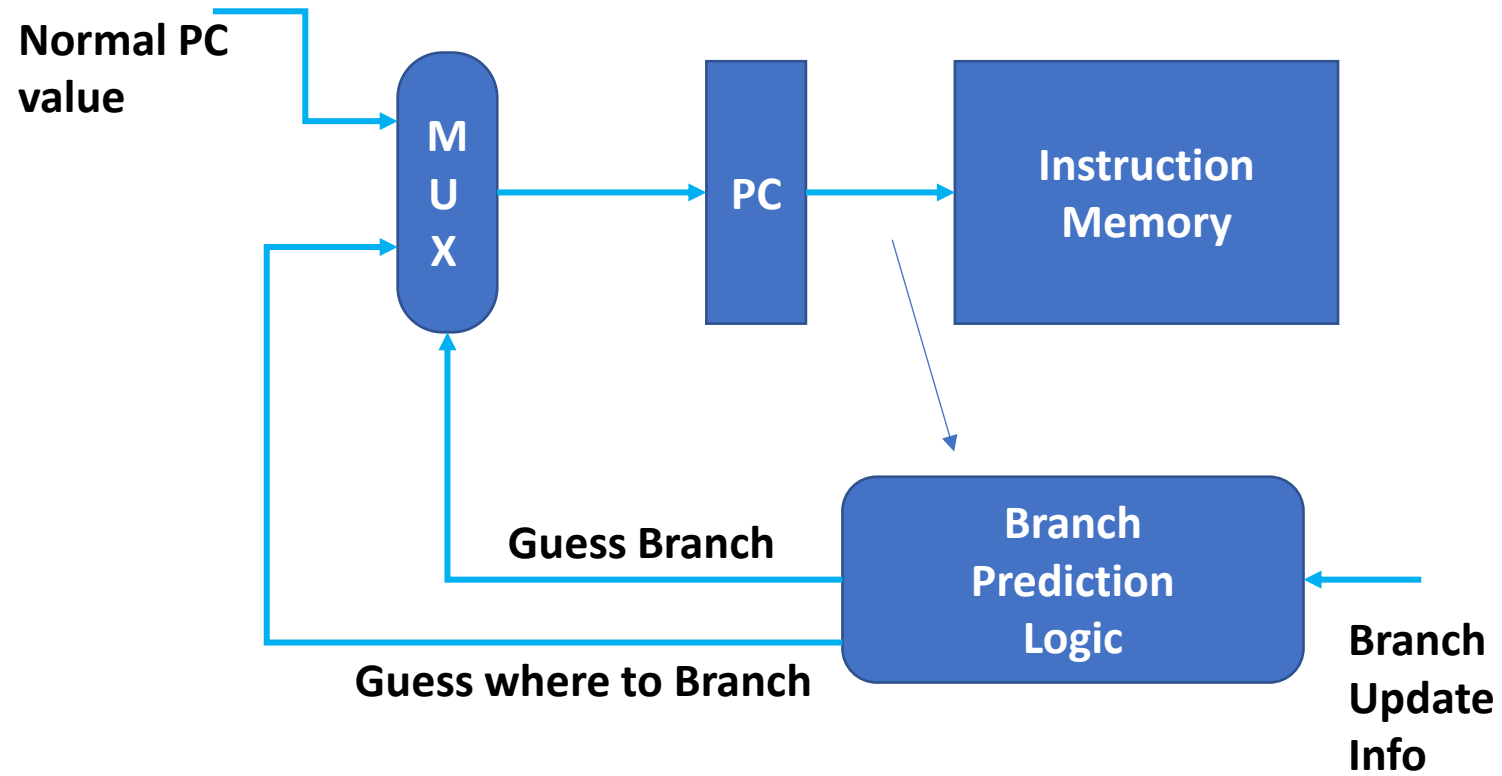
It's just a simple mechanism that might give us a hint as to what the right decision might be.

Note:

- The buffer is shared between branches with the same lower addresses. A buffer value may reflect another branch instruction.

- The branch instruction may simply make a different decision than it did before.
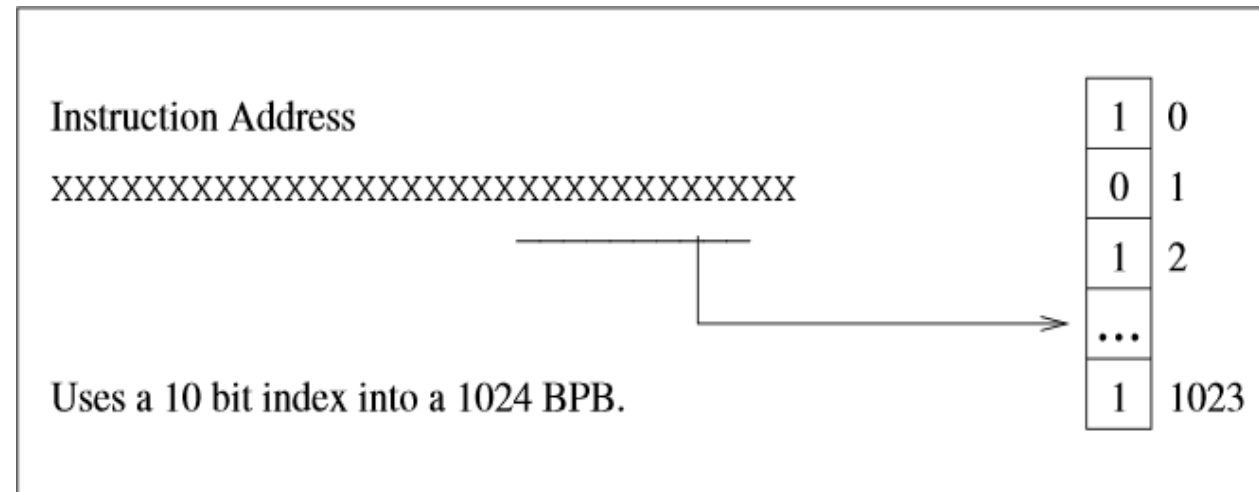
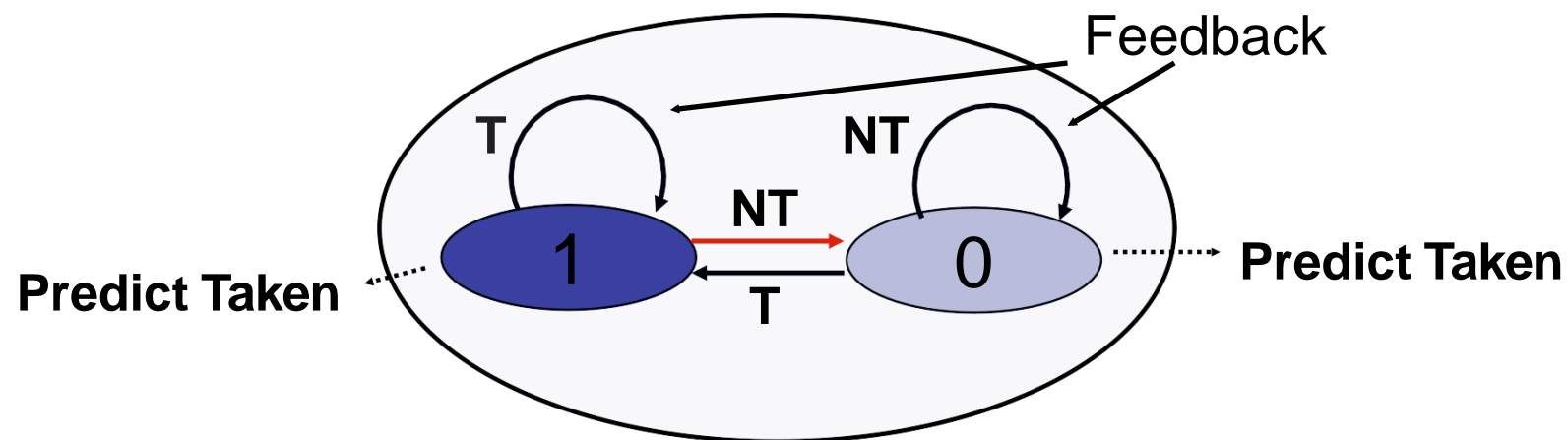# 1 Bit Branch Prediction

A Branch Predictor

Normal PC value

M U X

PC

Instruction Memory

Branch Prediction Logic

Guess Branch

Guess where to Branch

Branch Update Info

Here's how the 1-bit branch prediction buffer works:

- Each element in the buffer contains a single bit indicating whether the branch prediction was taken last time.

- We make our prediction based on the bit found in the buffer.

- If the prediction turns out to be incorrect, then we flip the bit in the buffer and correct the pipeline.

Instruction Address

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

1	0
0	1
1	2
...	
1	1023

Uses a 10 bit index into a 1024 BPB.
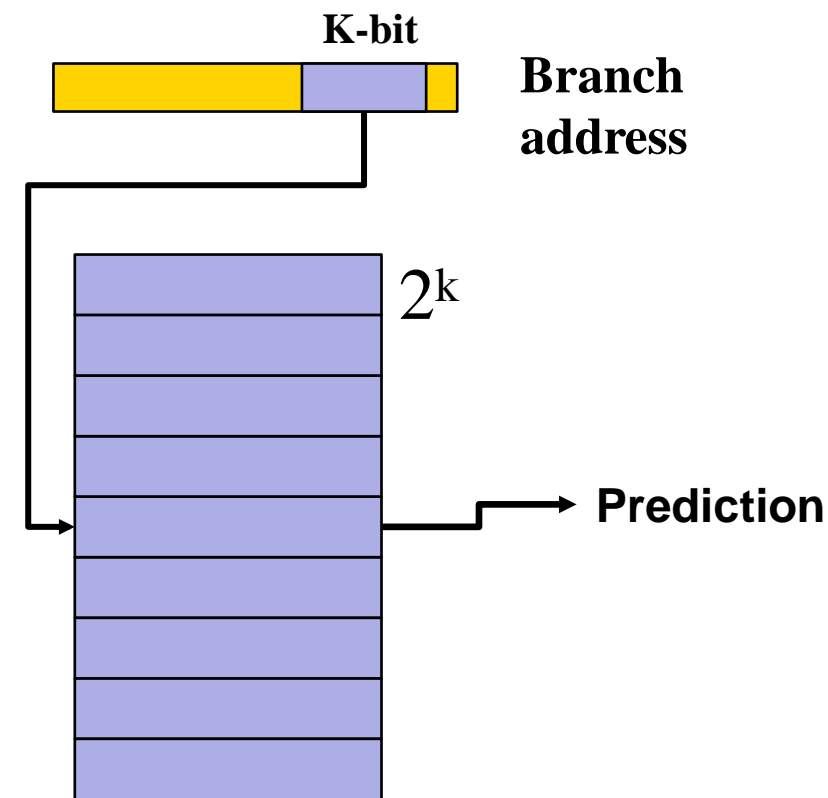
1-bit prediction

T = Taken
NT = Not Taken

BHT also called branch prediction buffer

- Can use only one 1-bit predictor, but accuracy is low

- BHT: use a table of simple predictors, indexed by bits from PC

- More entries, more cost, but less conflicts, higher accuracy

- BHT can contain complex predictors

K-bit

Branch address

2^k

Prediction

Consider a loop that branches nine times in a row, then is not taken once. What is the prediction accuracy for this branch, assuming the prediction bit for this branch remains in the prediction buffer?

```
int i = 0;
do{
    /* loop body */
    i = i + 1
}while(i < 10);
```

```
        add    $t0, $0, $0
L1:     /* loop body*/
        addi   $t0, $t0, 1
        slti   $t1, $t0, 10
        bne    $t1, $0, L1
```

- The prediction behavior will mispredict on both the first and last loop iterations.
- The last loop iteration prediction happens because we've already taken the branch nine-times so far.
- The first loop iteration happens because the bit was flipped on the last iteration of the previous execution.
- So, **the prediction accuracy is 80%.**

- **Example:**
  - **in a loop, 1-bit BHT will cause 2 mispredictions**

- **Consider a loop of 9 iterations before exit:**

  ```
  for (…){
    for (i=0; i<9; i++)
      a[i] = a[i] * 2.0;
  }
  ```

  - **End of loop case, when it exits instead of looping as before**
  - **First time through loop on *next* time through code, when it predicts *exit* instead of looping**
 - **Only 80% accuracy even if loop 90% of the time**

2 Bit Branch Prediction

2 Bit Branch Prediction

To increase this prediction accuracy, we can use a 2-bit prediction buffer. In the 2-bit scheme, a prediction must be wrong twice before the bit is flipped.
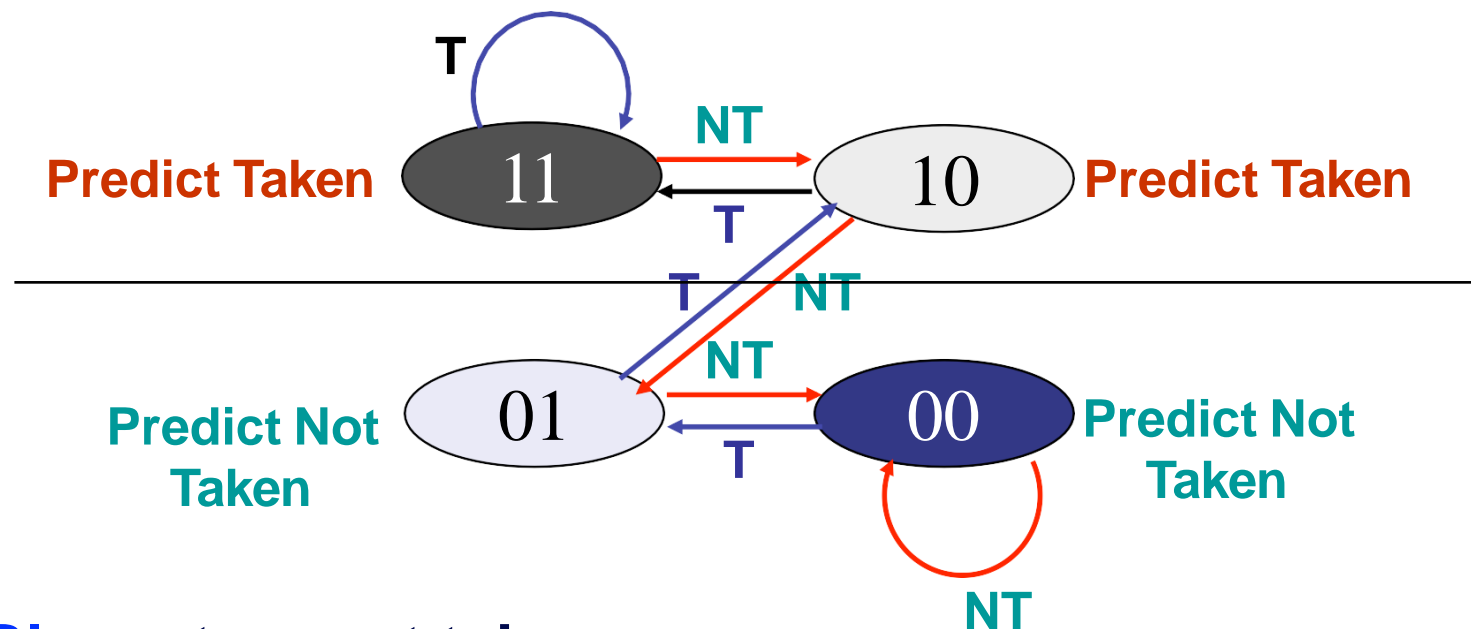
This way, a branch that strongly favors a particular decision will be wrong only once.

We can access the buffer during the IF stage to determine whether the next instruction needs to be calculated or we can continue with sequential execution.
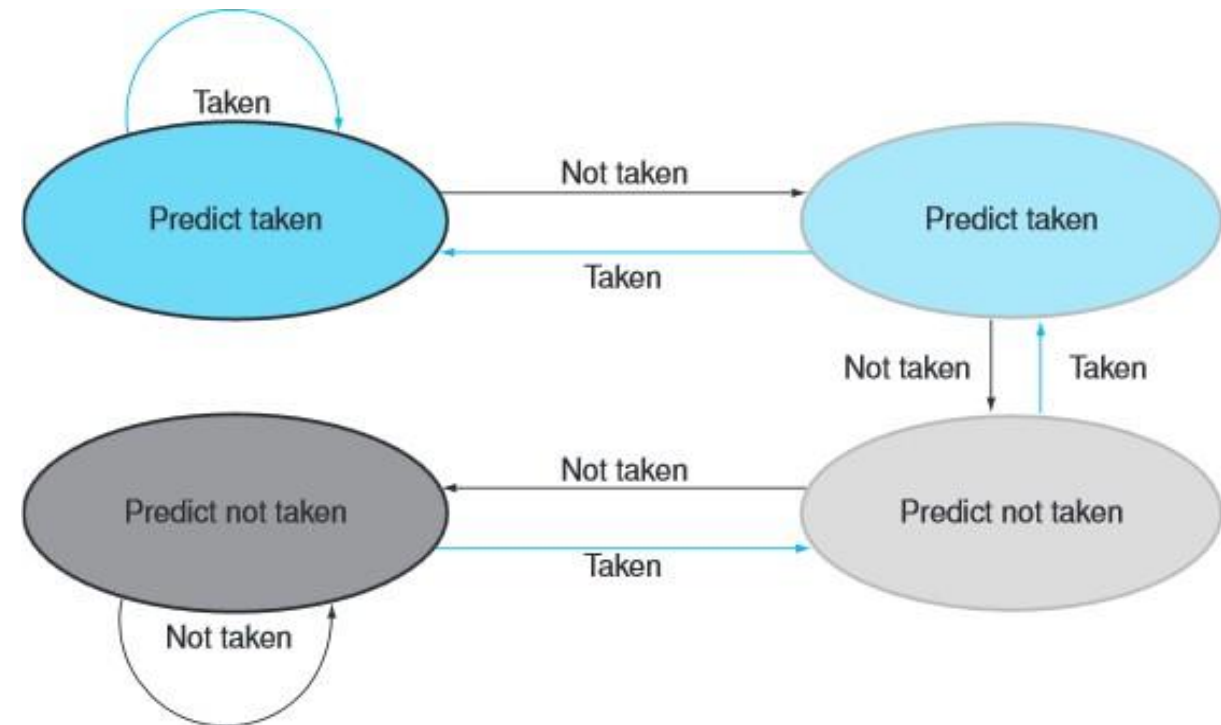
- **Solution: 2-bit scheme where change prediction only if get misprediction *twice*:**

- **Blue: stop, not taken**
- **Gray: go, taken**
- **Adds *hysteresis* to decision making process**

- Chap 4, P&H Textbook