

Computer Architecture

Pipeline: Hazards

Pipeline Hazards

- Where one instruction cannot immediately follow another
- Types of hazards
 - Structural hazards - attempt to use the same resource by two or more instructions
 - Control hazards - attempt to make branching decisions before branch condition is evaluated
 - Data hazards - attempt to use data before it is ready
- Can always resolve hazards by waiting (stalling the pipeline)

Structural Hazards

- **Attempt to use the same resource by two or more instructions at the same time**
- **Example: Single Memory for instructions and data**
 - Accessed by IF stage
 - Accessed at same time by MEM stage
- **Solutions**
 - Delay the second access by one clock cycle, OR
 - Provide separate memories for instructions & data
 - » This is what the book does
 - » This is called a “**Harvard Architecture**”
 - » Real pipelined processors have separate **caches**

Pipelined Example - Executing Multiple Instructions

- Consider the following instruction sequence:

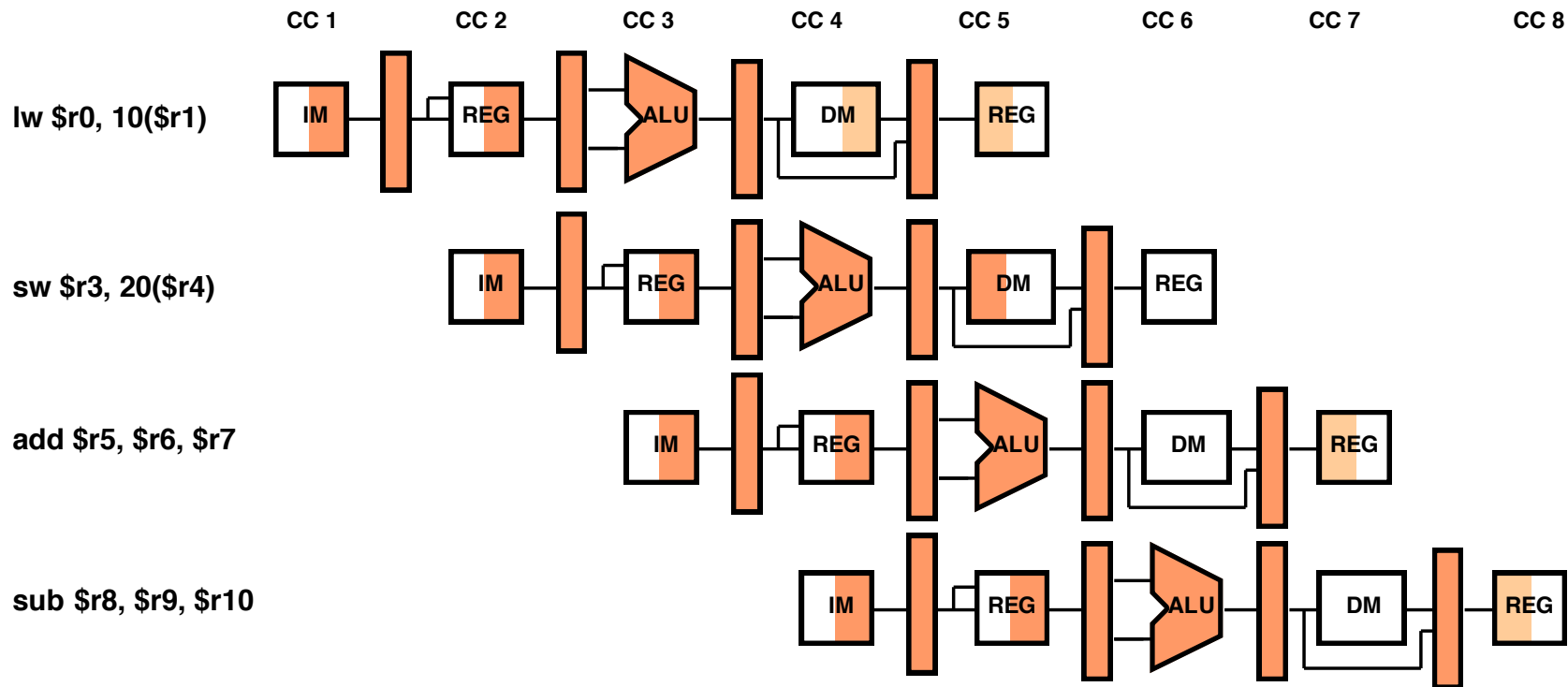
```
lw $r2, 12($r1)
```

```
sw $r3, 20($r4)
```

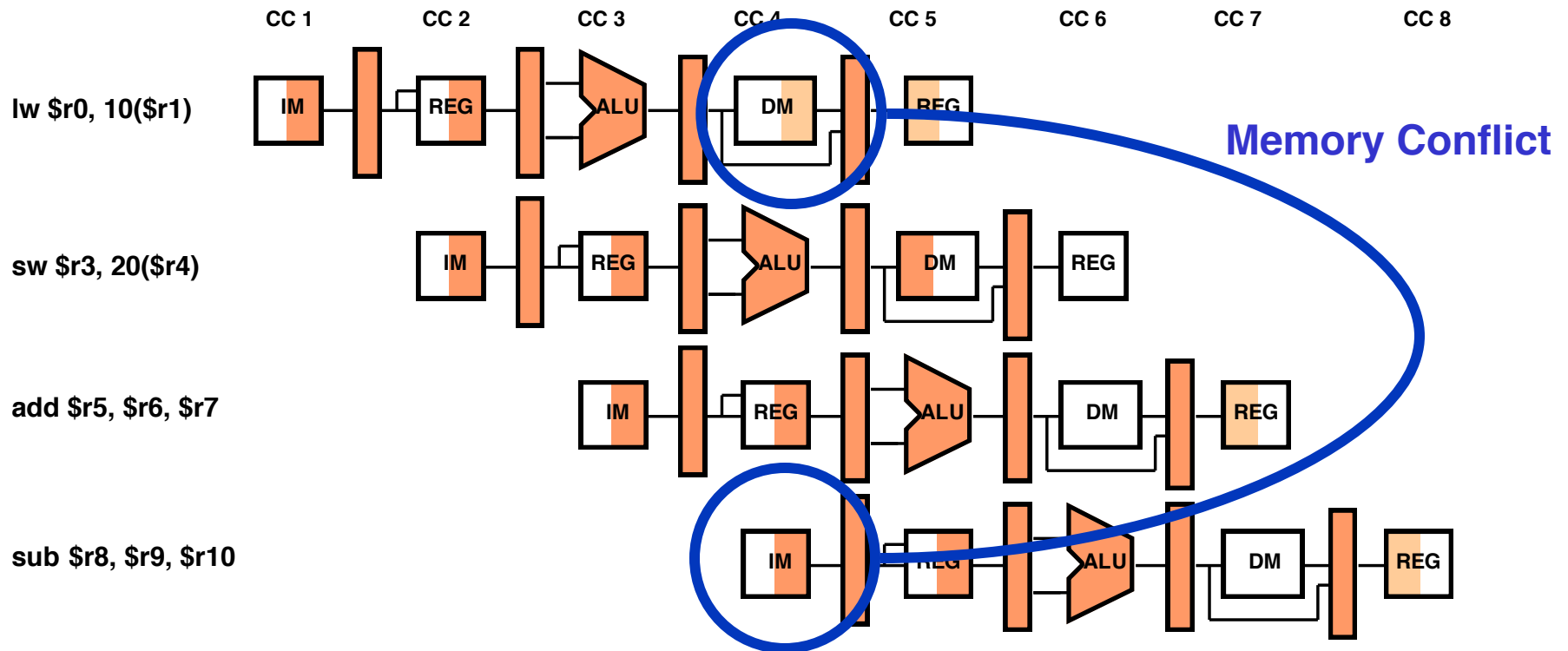
```
add $r5, $r6, $r7
```

```
sub $r8, $r9, $r10
```

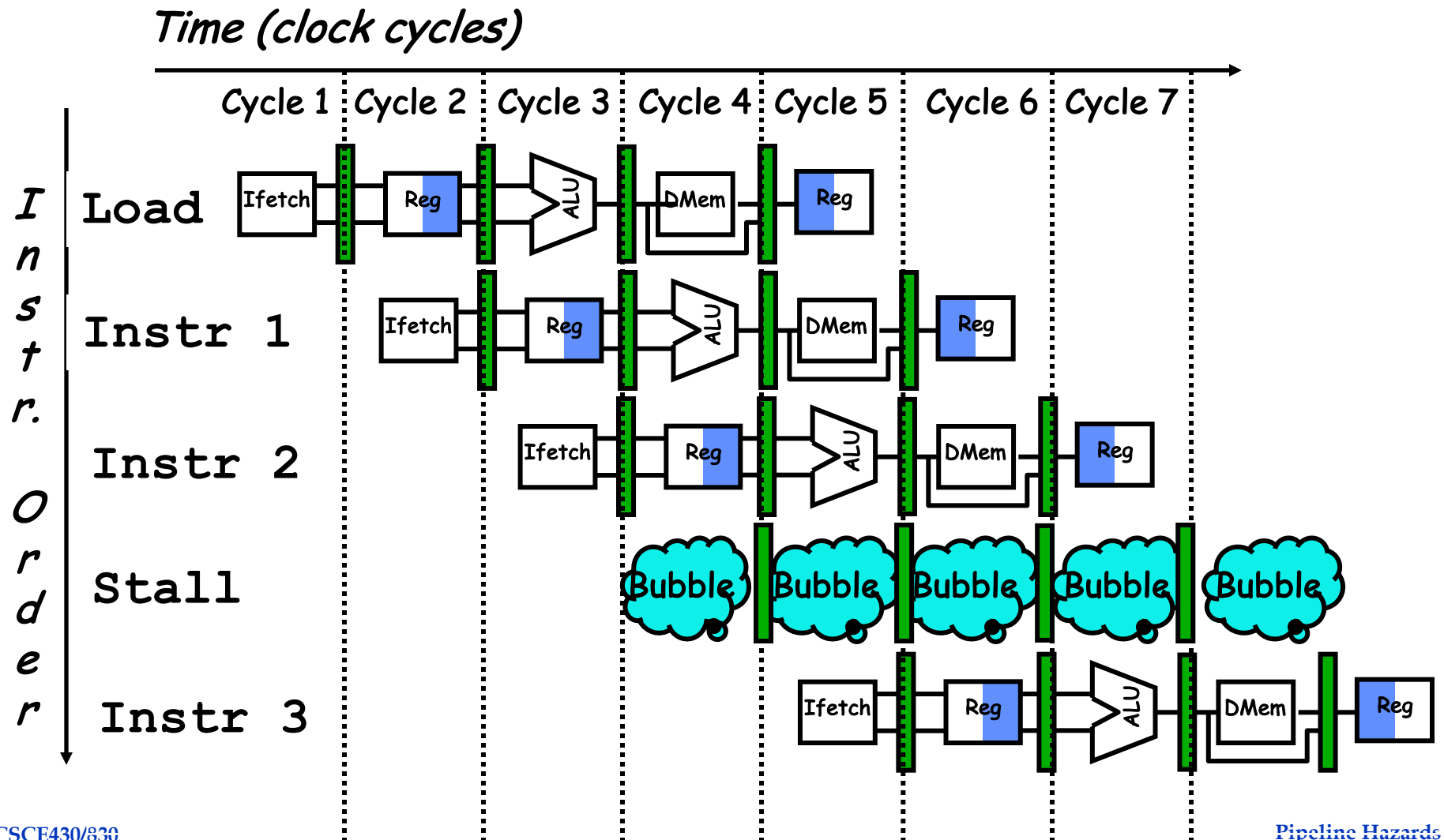
Alternative View - Multicycle Diagram



Alternative View - Multicycle Diagram



One Memory Port Structural Hazards



Dealing with Structural Hazards

Stall

- low cost, simple
- Increases CPI
- use for rare case since stalling has performance effect

Pipeline hardware resource

- useful for multi-cycle resources
- good performance
- sometimes complex e.g., RAM

Replicate resource

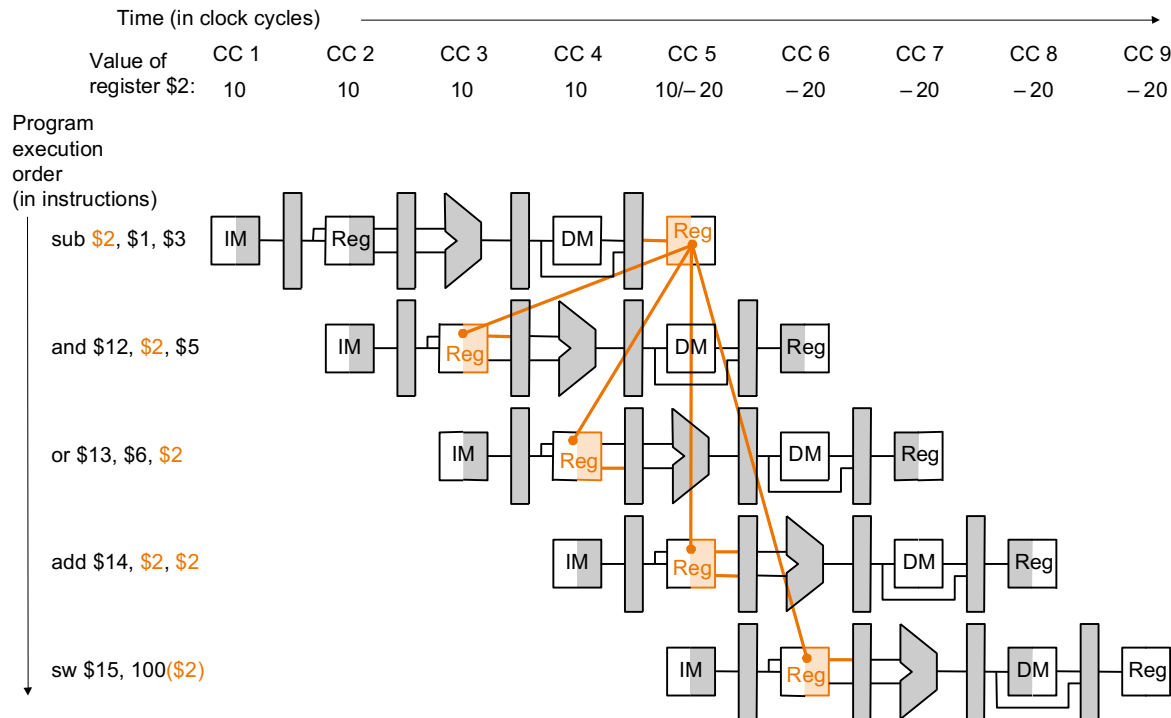
- good performance
- increases cost (+ maybe interconnect delay)
- useful for cheap or divisible resources

Structural Hazards Avoidance

- Structural hazards are reduced with these rules:
 - Each instruction uses a resource at most once
 - Always use the resource in the same pipeline stage
 - Use the resource for one cycle only
- Many RISC ISAs are designed with this in mind
- Sometimes very difficult to do this.
 - For example, memory of necessity is used in the IF and MEM stages.

Data Hazards

- Data hazards occur when data is used before it is ready



The use of the result of the SUB instruction in the next three instructions causes a data hazard, since the register \$2 is not written until after those instructions read it.

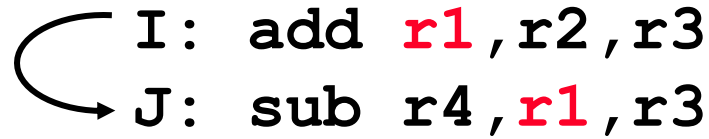
Data Hazards

Execution Order is:

Instr_I
Instr_J

Read After Write (RAW)

Instr_J tries to read operand before Instr_I writes it



I: add **r1**, r2, r3
J: sub r4, **r1**, r3

- Caused by a “**Dependence**”. This hazard results from an actual need for communication.

Data Hazards

Execution Order is:

Instr_i
Instr_j

Write After Read (WAR)

Instr_j tries to write operand before Instr_i reads i

- Gets wrong operand

```
I:  sub  r4, r1, r3
J:  add  r1, r2, r3
K:  mul  r6, r1, r7
```

- Called an “**anti-dependence**” by compiler writers. This results from reuse of the name “**r1**”.
- Can’t happen in MIPS 5 stage pipeline because:
 - All instructions take 5 stages, and
 - Reads are always in stage 2, and
 - Writes are always in stage 5

Data Hazards

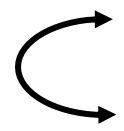
Execution Order is:

Instr_I
Instr_J

Write After Write (WAW)

Instr_J tries to write operand before Instr_I writes it

- Leaves wrong result (Instr_I not Instr_J)

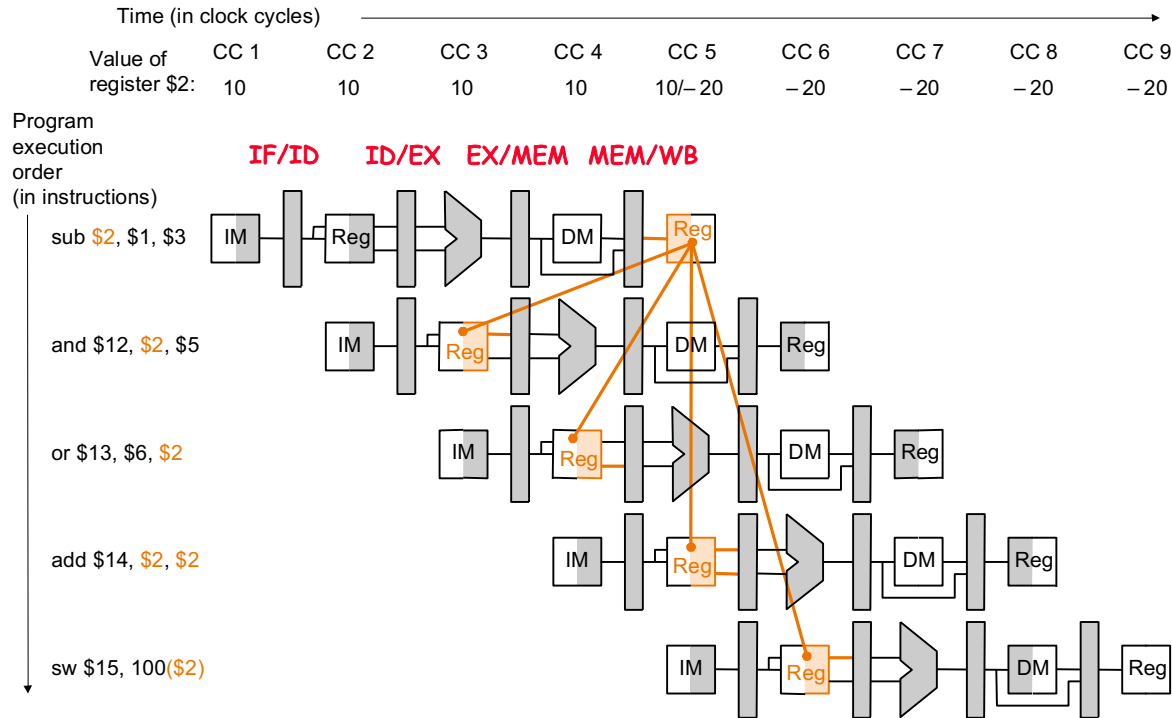


```
I:  sub  r1, r4, r3
J:  add  r1, r2, r3
K:  mul  r6, r1, r7
```

- Called an “**output dependence**” by compiler writers
This also results from the reuse of name “**r1**”.
- Can’t happen in MIPS 5 stage pipeline because:
 - All instructions take 5 stages, and
 - Writes are always in stage 5
- Will see WAR and WAW later in more complicated pipes

Data Hazard Detection in MIPS (1)

Read after Write



1a: $EX/MEM.RegisterRd = ID/EX.RegisterRs$

1b: $EX/MEM.RegisterRd = ID/EX.RegisterRt$

2a: $MEM/WB.RegisterRd = ID/EX.RegisterRs$

2b: $MEM/WB.RegisterRd = ID/EX.RegisterRt$

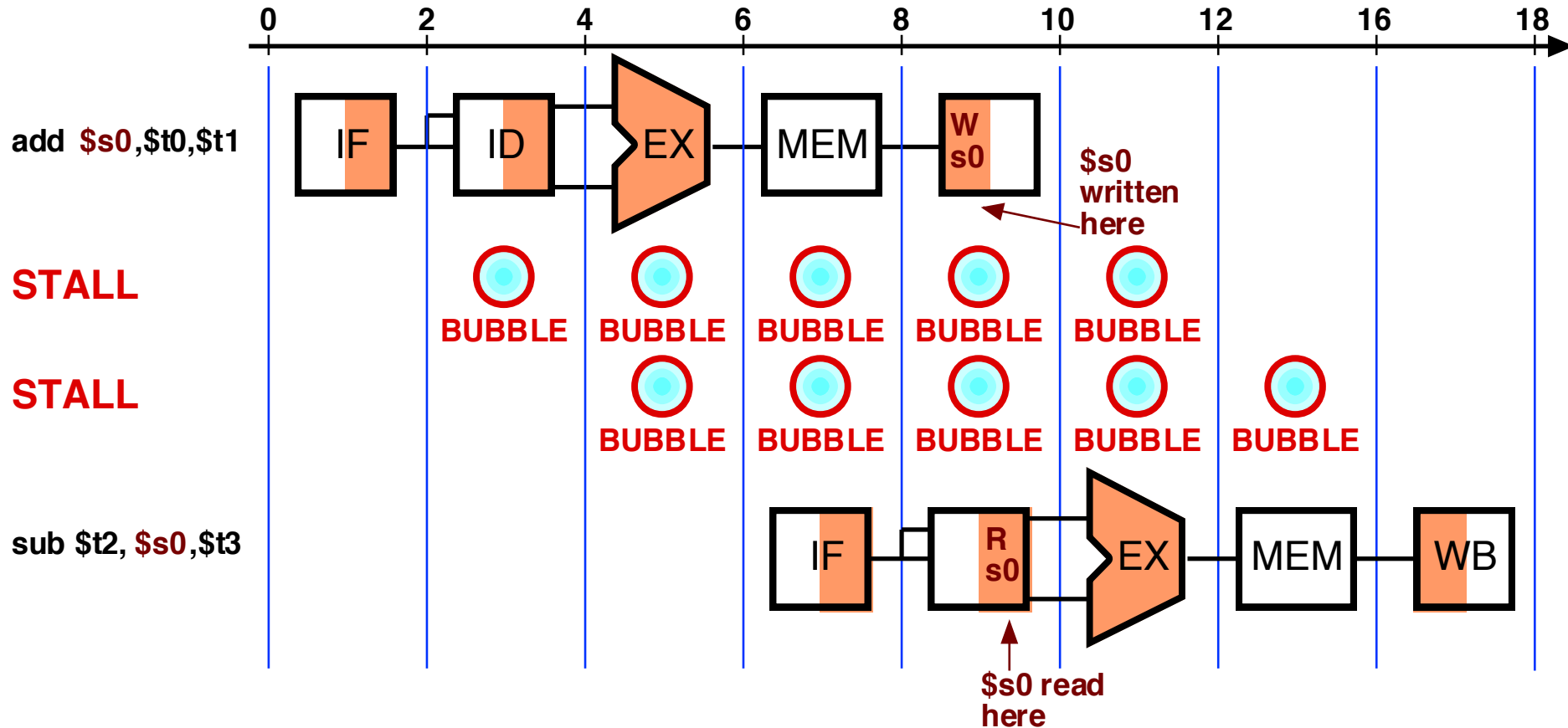
} EX hazard

} MEM hazard

Data Hazards

- **Solutions for Data Hazards**
 - **Stalling**
 - **Forwarding:**
 - » connect new value directly to next stage
 - **Reordering**

Data Hazard - Stalling



Data Hazards - Stalling

Simple Solution to RAW

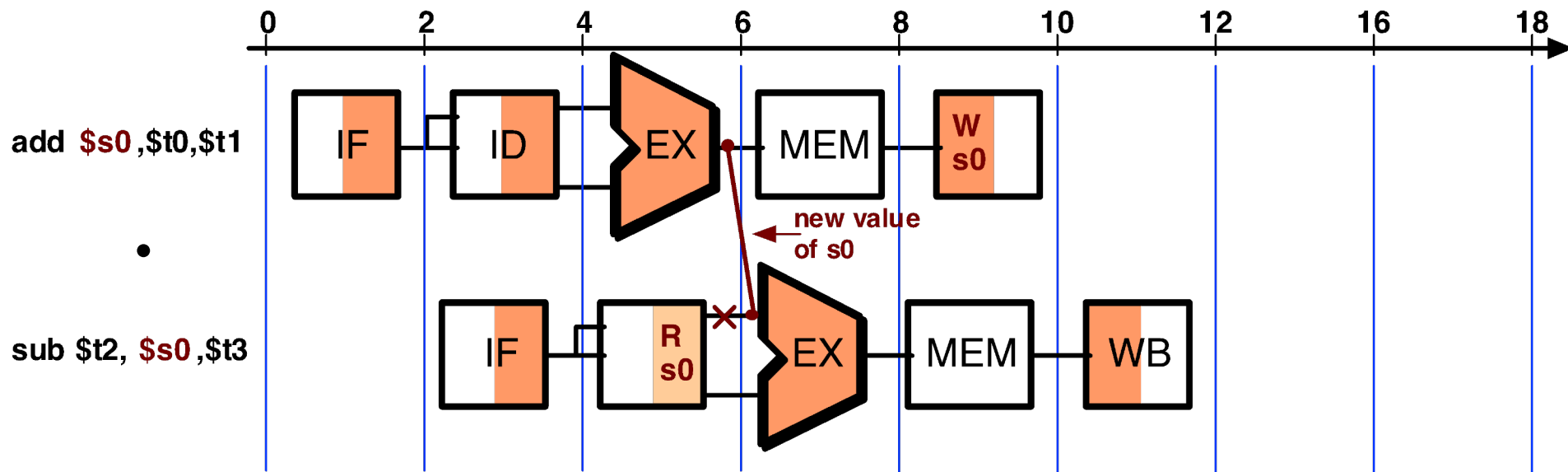
- Hardware detects RAW and stalls
- Assumes register written then read each cycle
 - + low cost to implement, simple
 - reduces IPC (increases CPI)
- Try to minimize stalls

Minimizing RAW stalls

- Bypass/forward/shortcircuit (We will use the word “forward”)
- Use data before it is in the register
 - + reduces/avoids stalls
 - complex
- Crucial for common RAW hazards

Data Hazards - Forwarding

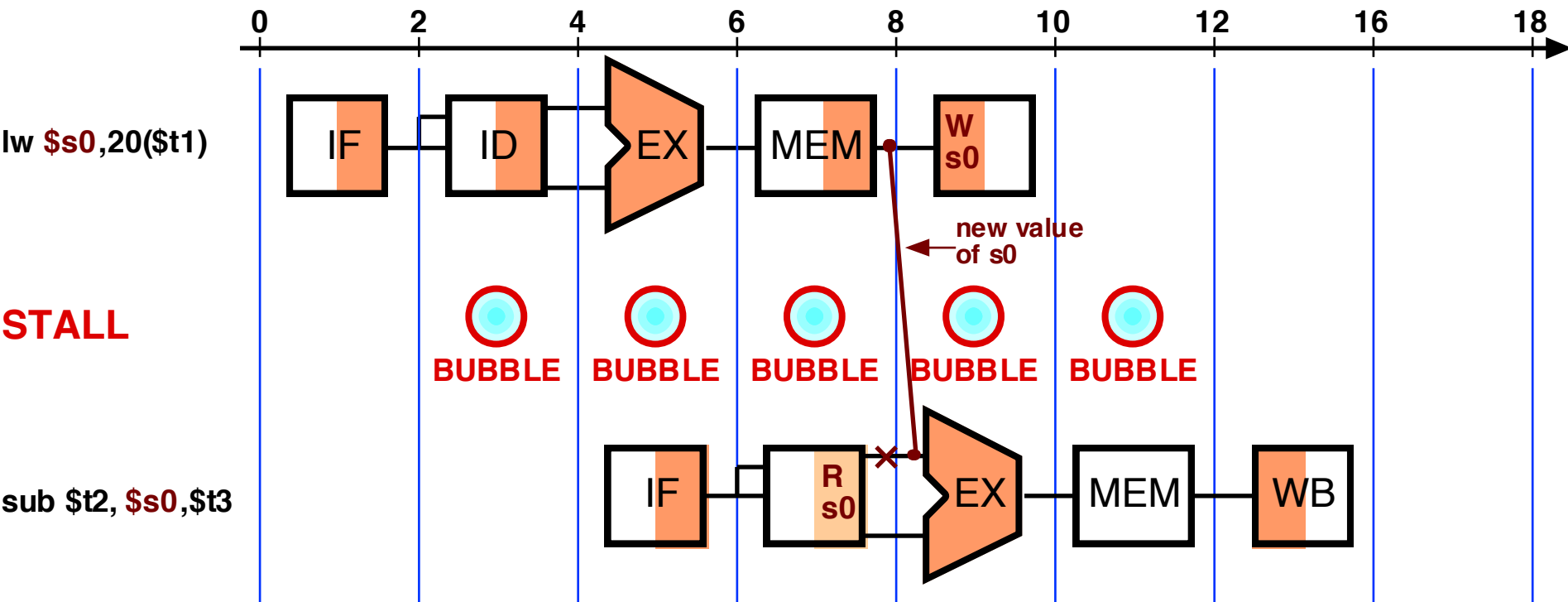
- Key idea: connect new value directly to next stage
- Still read s0, but ignore in favor of new result



- Problem: what about load instructions?

Data Hazards - Forwarding

- **STALL** still required for load - data avail. after MEM
- **MIPS** architecture calls this delayed load, initial implementations required compiler to deal with this



Data Hazards

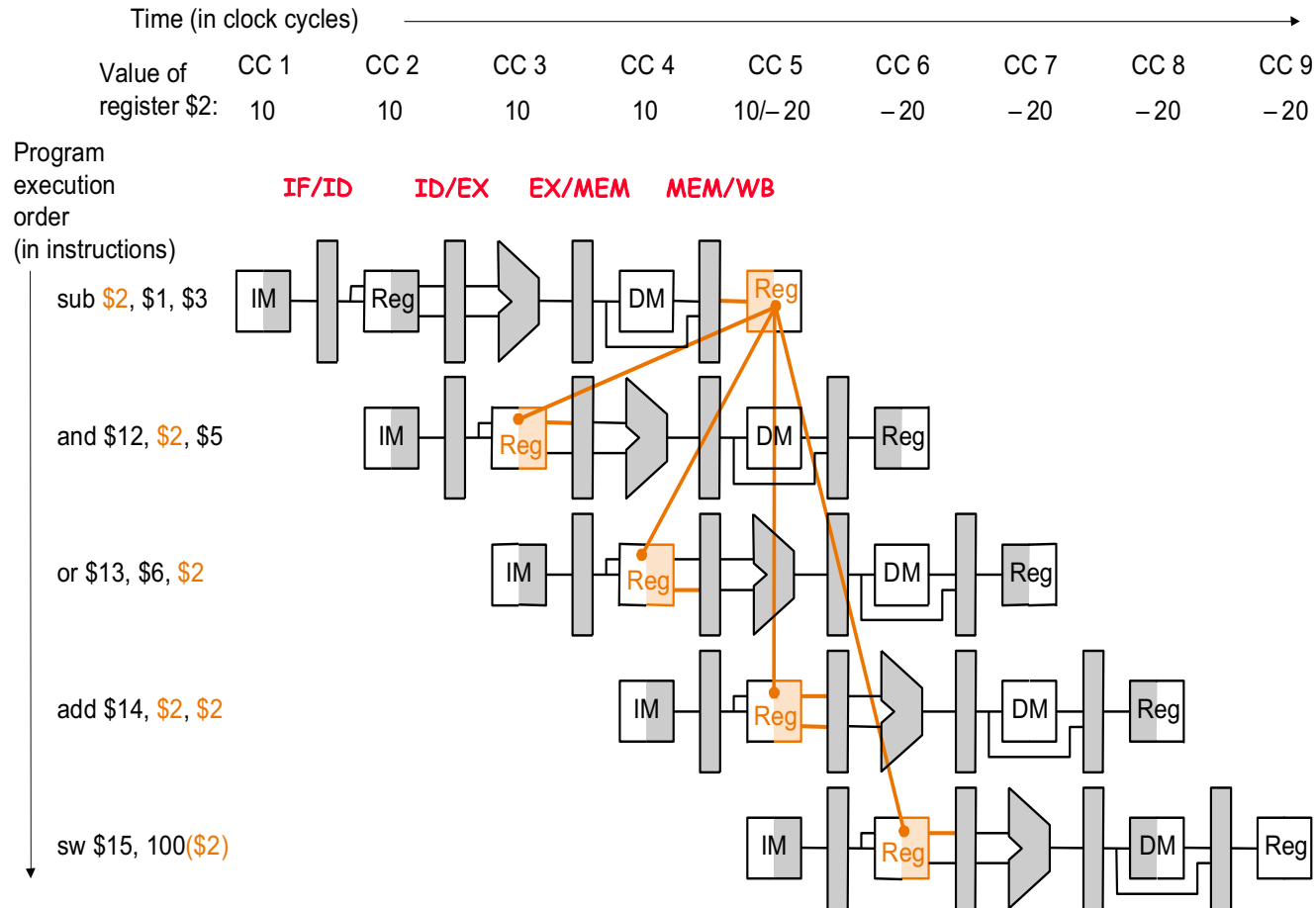
This is another representation of the stall.

LW R1, 0(R2)	IF	ID	EX	MEM	WB			
SUB R4, R1, R5		IF	ID	EX	MEM	WB		
AND R6, R1, R7			IF	ID	EX	MEM	WB	
OR R8, R1, R9				IF	ID	EX	MEM	WB

LW R1, 0(R2)	IF	ID	EX	MEM	WB				
SUB R4, R1, R5		IF	ID	stall	EX	MEM	WB		
AND R6, R1, R7			IF	stall	ID	EX	MEM	WB	
OR R8, R1, R9				stall	IF	ID	EX	MEM	WB

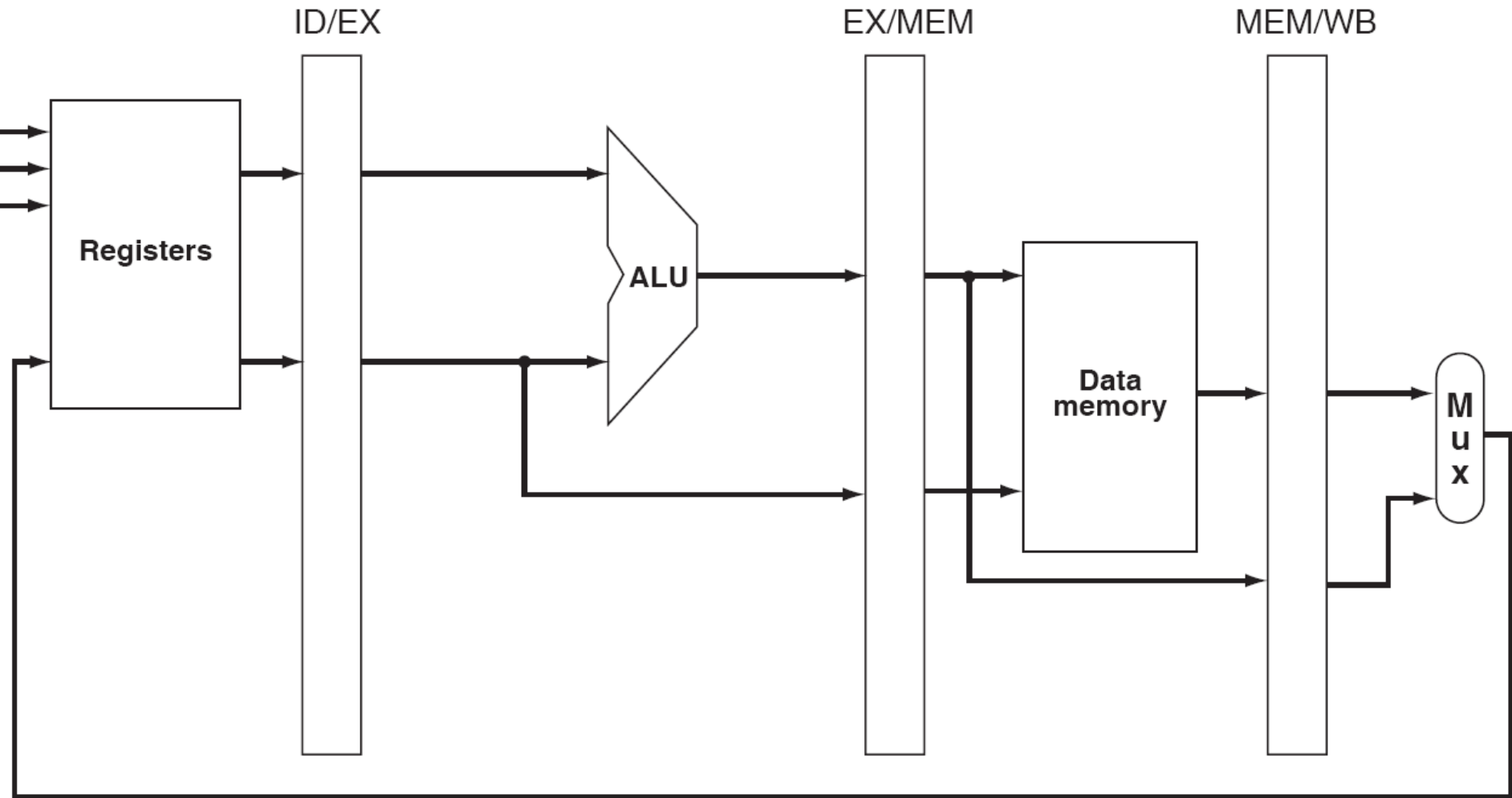
Forwarding

Key idea: connect data internally before it's stored



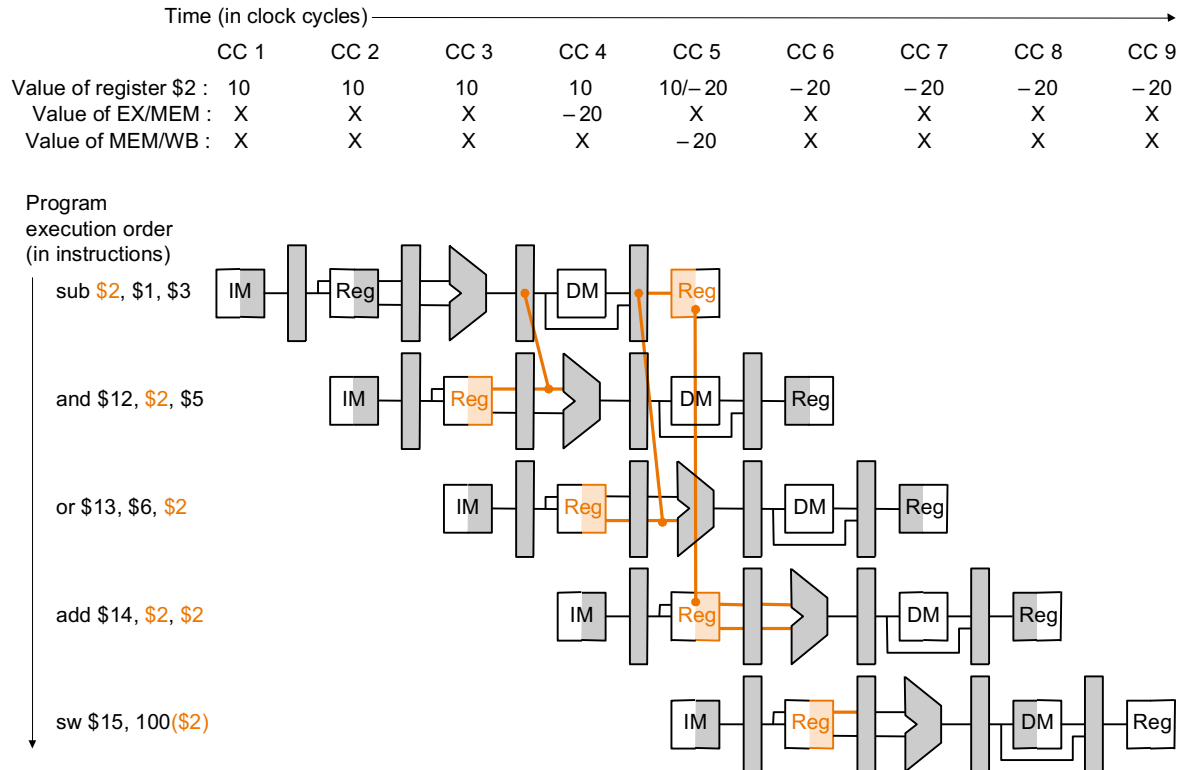
How would you design the forwarding?

No Forwarding



Data Hazard Solution: Forwarding

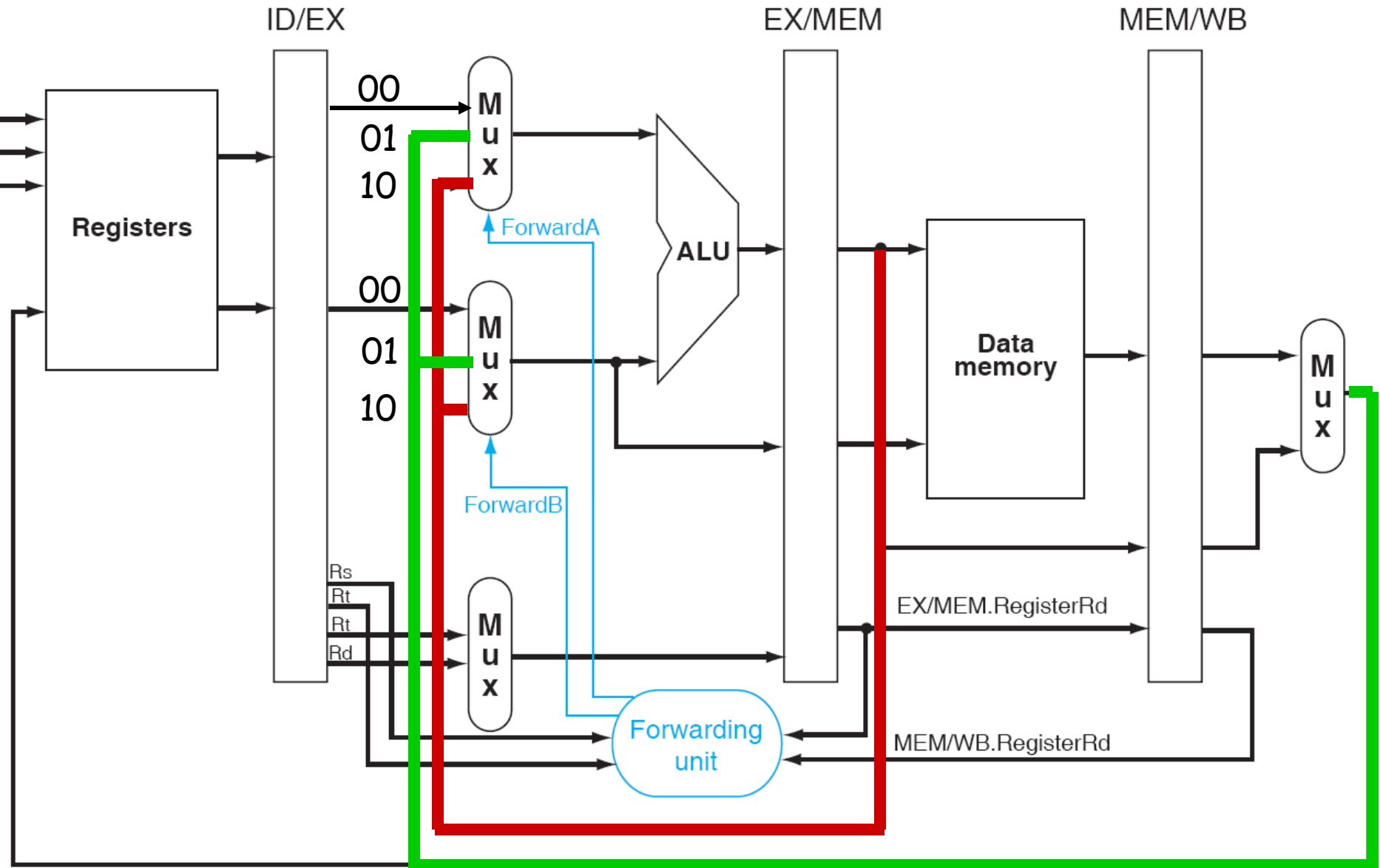
- **Key idea: connect data internally before it's stored**



Assumption:

- The register file forwards values that are read and written during the same cycle.

Forwarding



Controlling Forwarding

- Need to test when register numbers match in rs, rt, and rd fields stored in pipeline registers
- "EX" hazard:
 - EX/MEM - test whether instruction writes register file and examine **rd** register
 - ID/EX - test whether instruction reads **rs** or **rt** register and **matches rd** register in EX/MEM
- "MEM" hazard:
 - MEM/WB - test whether instruction writes register file and examine **rd (rt)** register
 - ID/EX - test whether instruction reads **rs** or **rt** register and **matches rd (rt)** register in EX/MEM

Forwarding Unit Detail - EX Hazard

if (EX/MEM.RegWrite)
and (EX/MEM.RegisterRd \neq 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRs))
 ForwardA = 10

if (EX/MEM.RegWrite)
and (EX/MEM.RegisterRd \neq 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRt))
 ForwardB = 10

Forwarding Unit Detail - MEM Hazard

if (MEM/WB.RegWrite)
and (MEM/WB.RegisterRd \neq 0)
and (MEM/WB.RegisterRd = ID/EX.RegisterRs))
ForwardA = 01

if (MEM/WB.RegWrite)
and (MEM/WB.RegisterRd \neq 0)
and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
ForwardB = 01

Data Hazards and Stalls

- So far, we've only addressed “**potential**” **data hazards**, where the **forwarding unit** was able to detect and resolve them without affecting the performance of the pipeline.
- There are also “**unavoidable**” **data hazards**, which the forwarding unit cannot resolve, and whose resolution does affect pipeline performance.
- We thus add a (unavoidable) **hazard detection unit**, which detects them and introduces **stalls** to resolve them.

Data Hazards & Stalls

- Identify the true data hazard in this sequence:

LW \$s0, 100(\$t0) ;\$s0 = memory value

ADD \$t2, \$s0, \$t3 ;\$t2 = \$s0 + \$t3

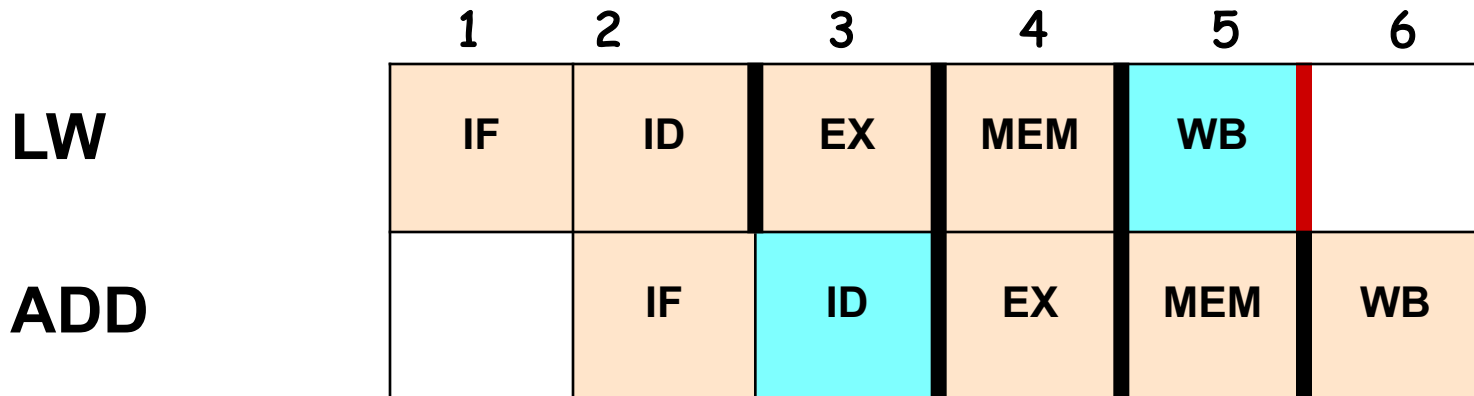
	1	2	3	4	5	6
LW	IF	ID	EX	MEM	WB	
ADD		IF	ID	EX	MEM	WB

Data Hazards & Stalls

- Identify the true data hazard in this sequence:

LW **\$s0**, 100(\$t0) ;\$s0 = memory value

ADD \$t2, **\$s0**, \$t3 ;\$t2 = \$s0 + \$t3

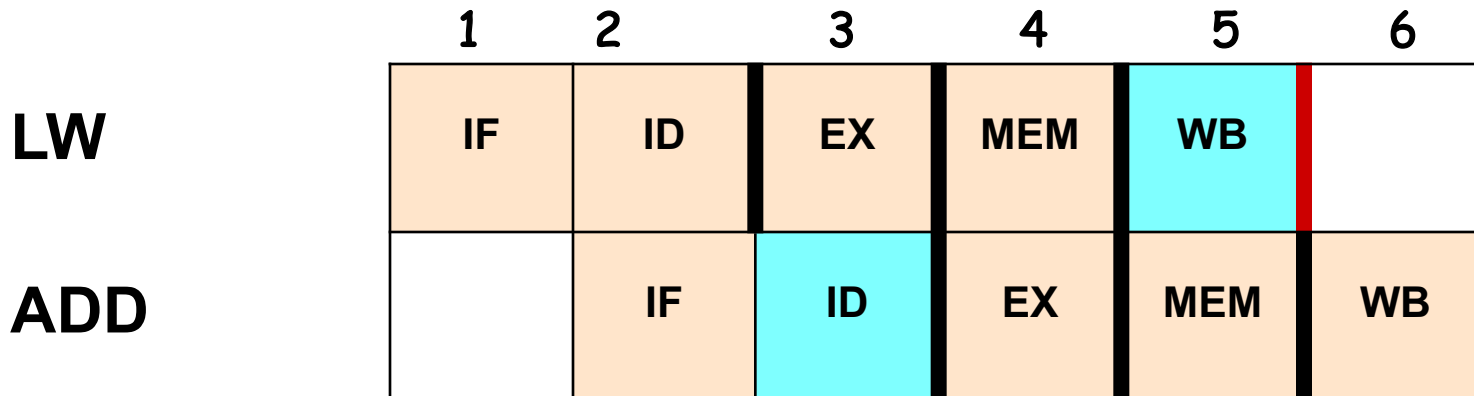


- LW doesn't write \$s0 to **Reg File** until the end of CC5, but ADD reads \$s0 from Reg File in CC3

Data Hazards & Stalls

LW **\$s0**, 100(\$t0) ;\$s0 = memory value

ADD \$t2, **\$s0**, \$t3 ;\$t2 = \$s0 + \$t3

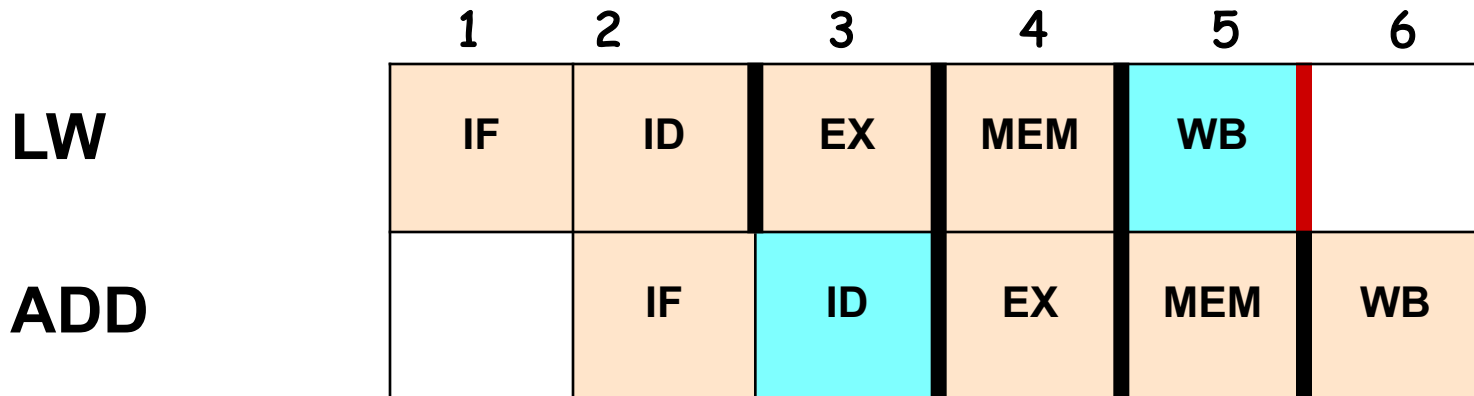


- EX/MEM forwarding won't work, because the data isn't loaded from memory until CC4 (so it's not in EX/MEM register)

Data Hazards & Stalls

LW **\$s0**, 100(\$t0) ;\$s0 = memory value

ADD \$t2, **\$s0**, \$t3 ;\$t2 = \$s0 + \$t3

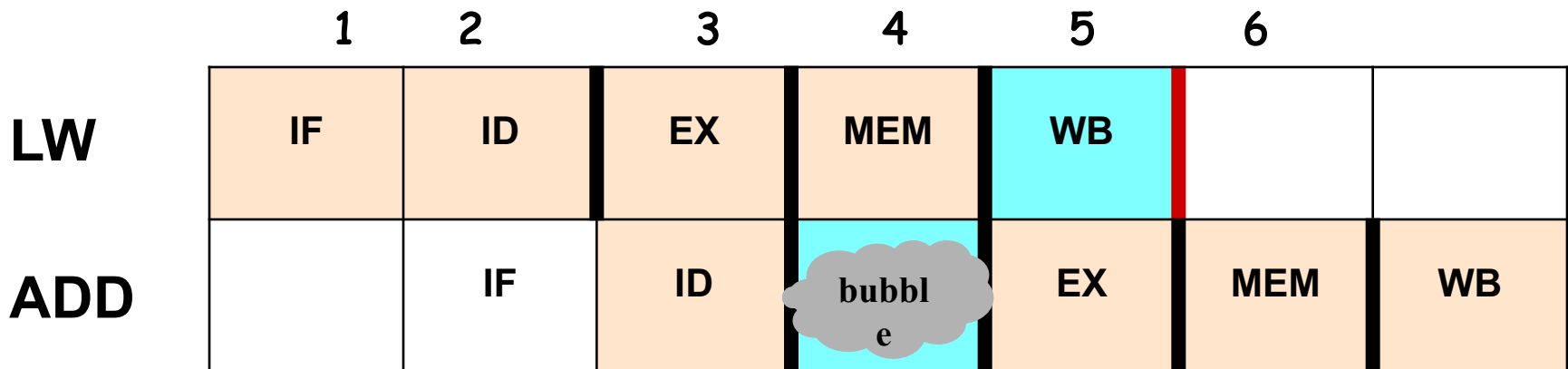


- MEM/WB forwarding won't work either, because ADD executes in CC4

Data Hazards & Stalls: implementation

LW **\$s0**, 100(\$t0) ;\$s0 = memory value

ADD \$t2, **\$s0**, \$t3 ;\$t2 = \$s0 + \$t3

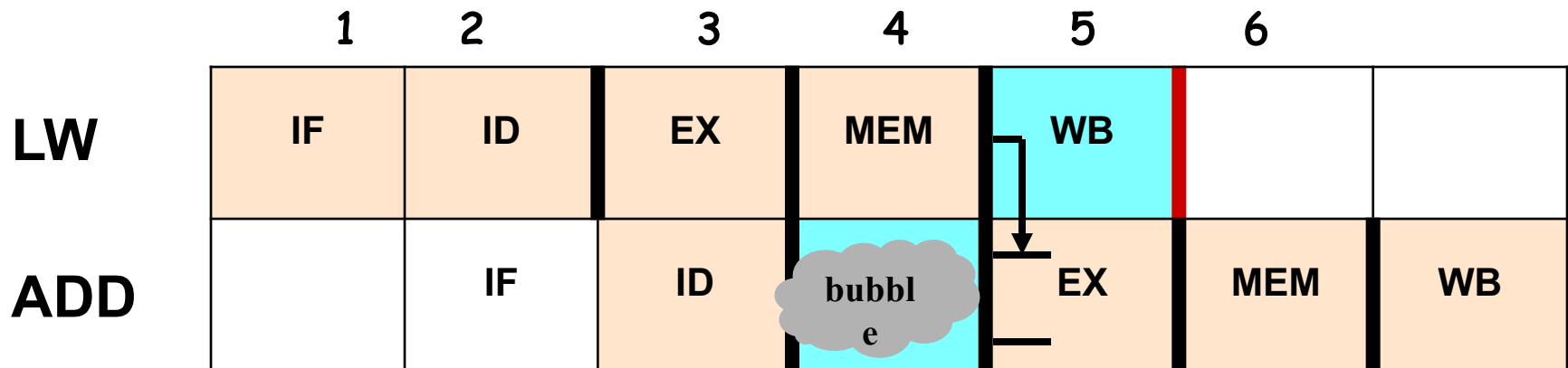


- We must handle this hazard by “**stalling**” the pipeline for 1 Clock Cycle (**bubble**)

Data Hazards & Stalls: implementation

LW **\$s0**, 100(\$t0) ;\$s0 = memory value

ADD \$t2, **\$s0**, \$t3 ;\$t2 = \$s0 + \$t3



- We can then use MEM/WB forwarding, but of course there is still a performance loss

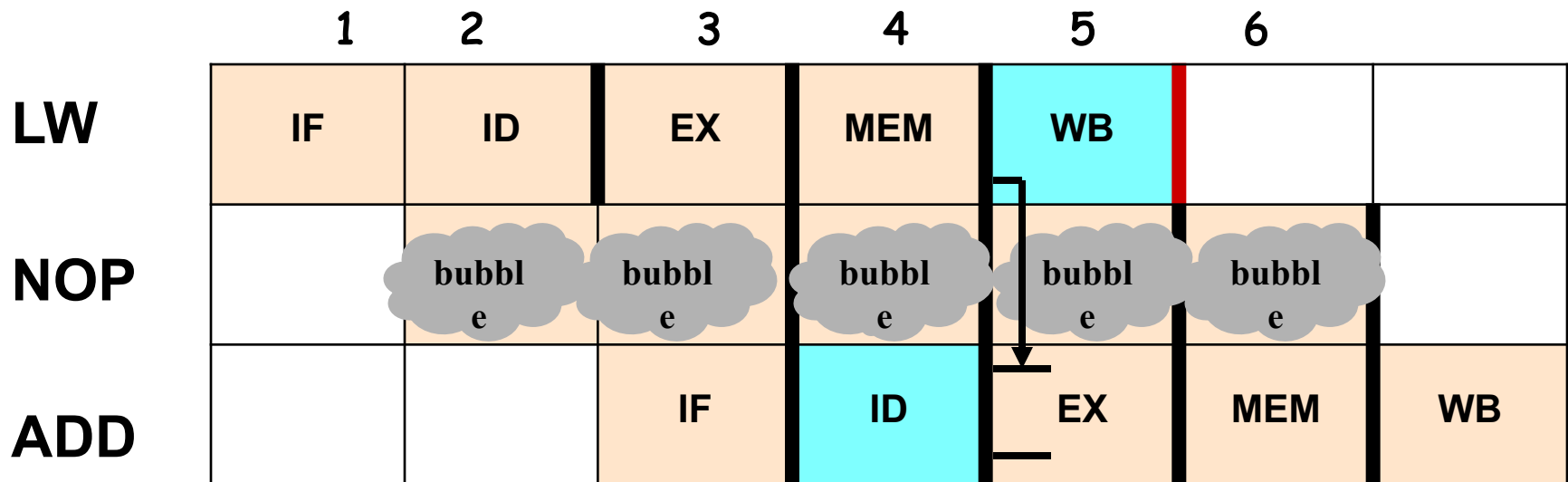
Data Hazards & Stalls: implementation

- Stall Implementation #1:** Compiler detects hazard and inserts a NOP (no reg changes (SLL \$0, \$0, 0))

LW **\$s0**, 100(\$t0) ;\$s0 = memory value

NOP ;dummy instruction

ADD \$t2, **\$s0**, \$t3 ;\$t2 = \$s0 + \$t3



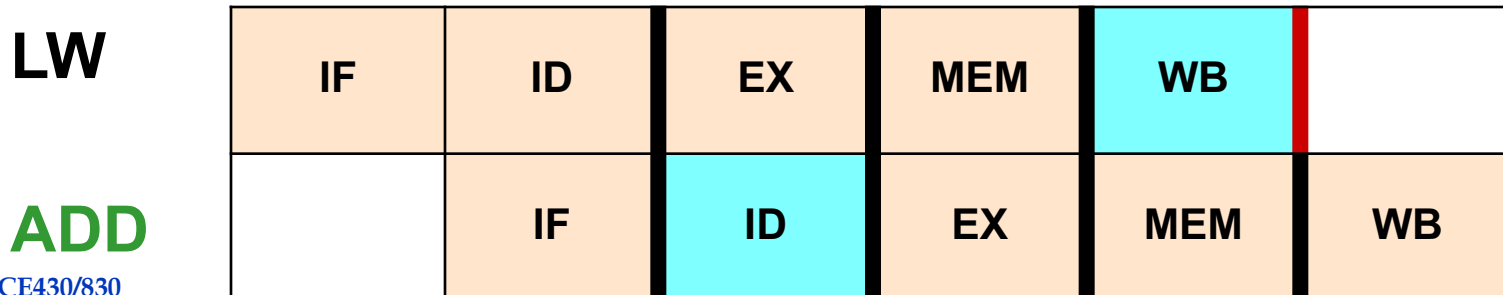
• Problem: we have to rely on the compiler

Data Hazards & Stalls: implementation

- **Stall Implementation #2:** Add a “hazard detection unit” to stall **current instruction** for 1 CC if:
- **ID-Stage Hazard Detection and Stall Condition:**

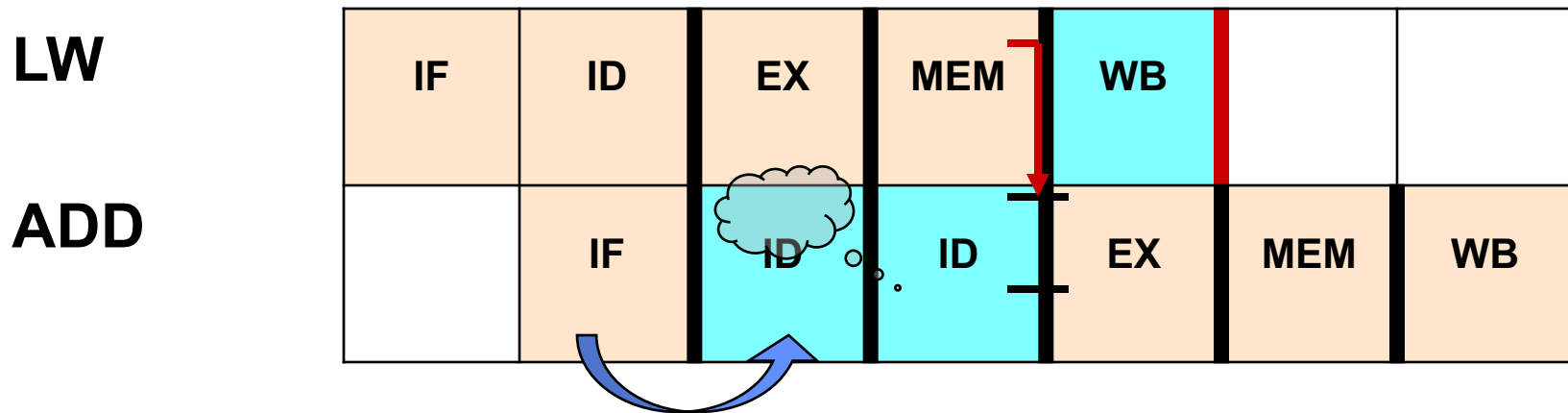
If ((ID/EX.MemRead = 1) & ;only a LW reads mem
((ID/EX.RegRT = IF/ID.RegRS) || ;RS will read load dest (RT)
(ID/EX.RegRT = IF/ID.RegRT))) ;RT will read load dest

LW **\$s0**, 100(\$t0) ;\$s0 = memory value
ADD \$t2, **\$s0**, \$t3 ;\$t2 = \$s0 + \$t3



Data Hazards & Stalls: implementation

- The effect of this stall will be to repeat the ID Stage of the current instruction. Then we do the MEM/WB forwarding on the next Clock Cycle



- We do this by preserving the current values in IF/ID for use on the next Clock Cycle

Data Hazards: A Classic Example

- Identify the data dependencies in the following code. Which of them can be resolved through forwarding?

SUB \$2, \$1, \$3

OR \$12, \$2, \$5

SW \$13, 100(\$2)

ADD \$14, \$2, \$2

LW \$15, 100(\$2)

ADD \$4, \$7, \$15

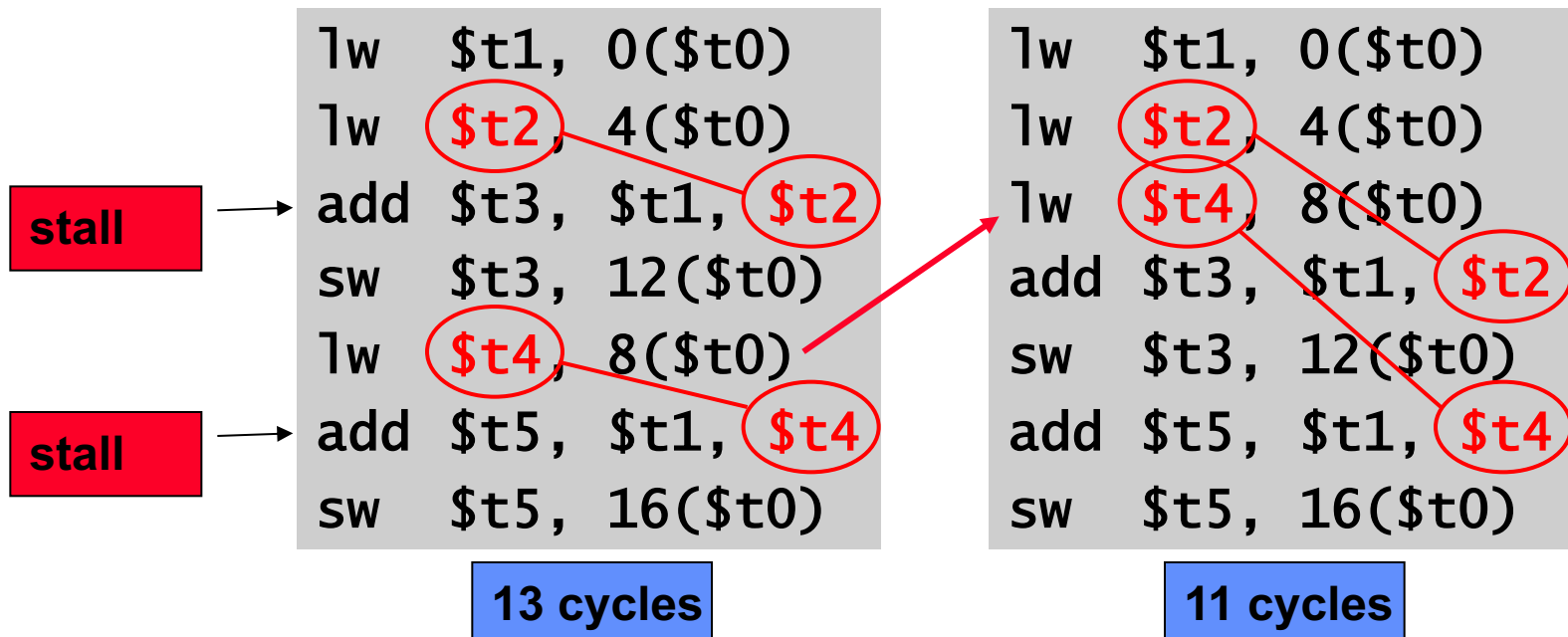
Reordering example

- **LW \$10, 12(\$2)**
- **ADD \$11,\$10,\$12**
- **AND \$14, \$14,\$15**

- **Reorder**
- **LW \$10, 12(\$2)**
- **AND \$14, \$14,\$15**
- **ADD \$11,\$10,\$12**

Code Scheduling to Avoid Stalls

- Reorder code to avoid use of load result in the next instruction
- C code for $A = B + E$; $C = B + F$;



Data Hazard Summary

- Three types of data hazards
 - **RAW (MIPS)**
 - WAW (not in MIPS)
 - WAR (not in MIPS)
- Solution to RAW in MIPS
 - Stall
 - Forwarding
 - » Detection & Control
 - EX hazard
 - MEM hazard
 - » **A stall is needed if read a register after a load instruction that writes the same register.**
 - Reordering

Control Hazards

A **control hazard** is when we need to find the destination of a branch, and can't fetch any new instructions until we know that destination.

A branch is either

- **Taken**: $PC \leq PC + 4 + \text{Immediate}$
- **Not Taken**: $PC \leq PC + 4$

Control Hazards

Control Hazard on Branches
Three Stage Stall

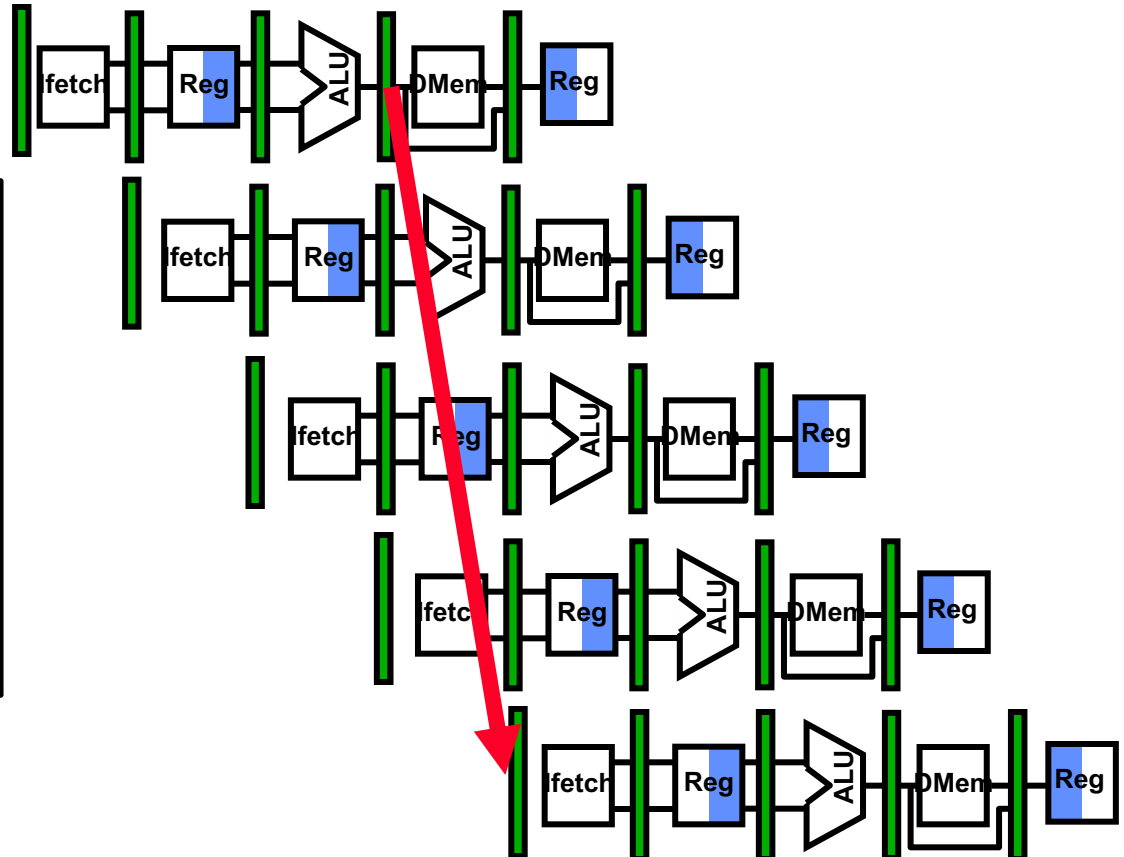
10: beq r1,r3,36

14: and r2,r3,r5

18: or r6,r1,r7

22: add r8,r1,r9

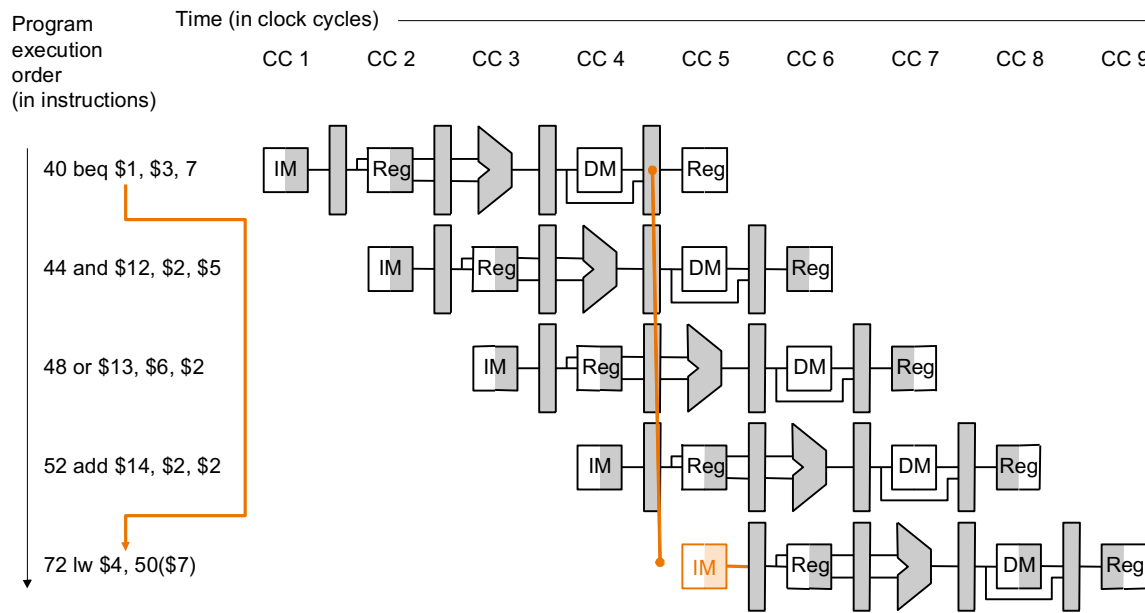
36: xor r10,r1,r11



The penalty when branch taken is 3 cycles!

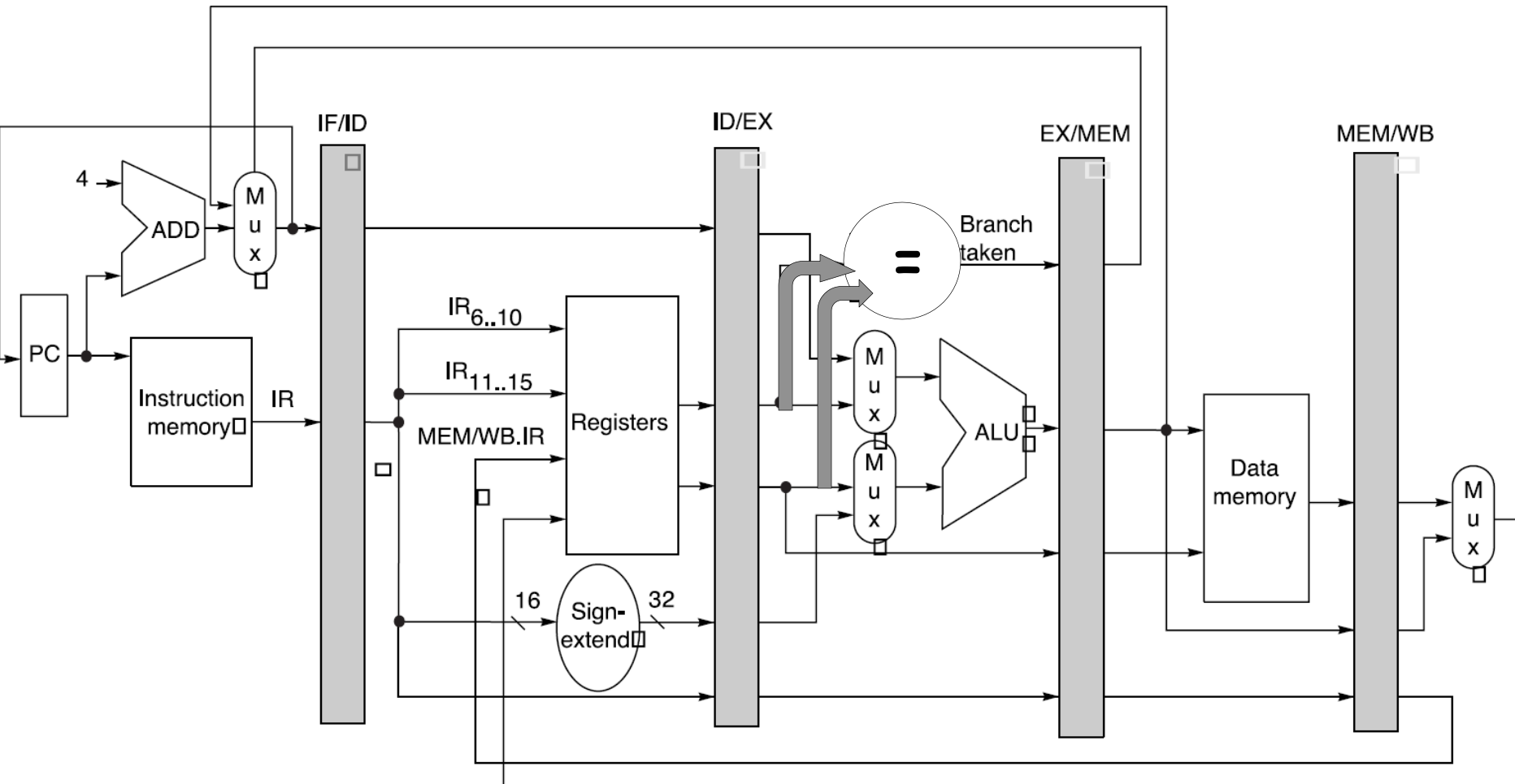
Branch Hazards

- Just stalling for each branch is not practical
- Common assumption: branch not taken
- When assumption fails: flush three instructions



(Fig. 6.37)

Basic Pipelined Processor



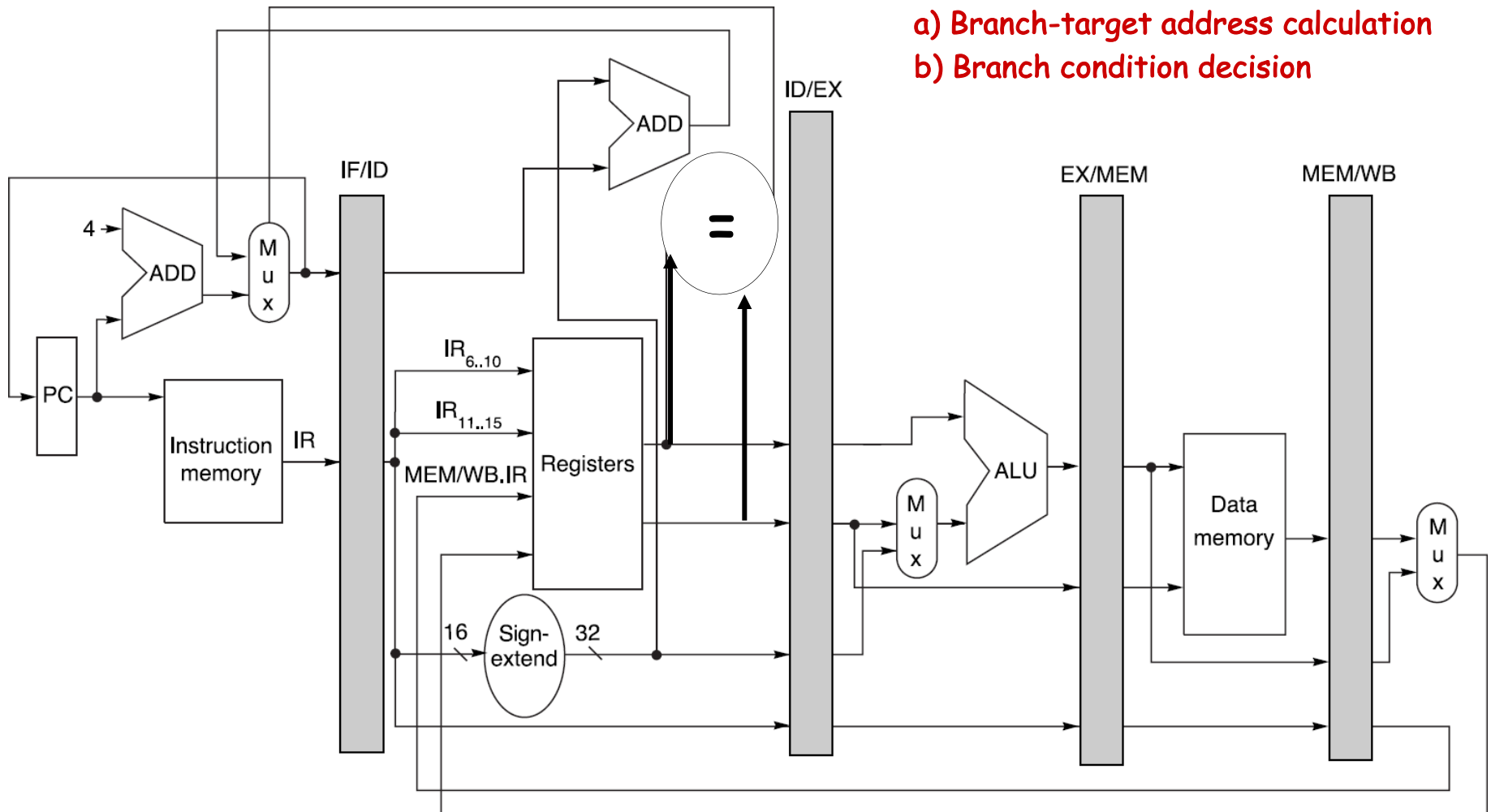
In our original Design, branches have a penalty of 3 cycles

Reducing Branch Delay

Move following to ID stage

a) Branch-target address calculation

b) Branch condition decision



Reduced penalty (1 cycle) when branch taken!

Reducing Branch Delay

- **Key idea: move branch logic to ID stage of pipeline**
 - New adder calculates branch target
 $(PC + 4 + \text{extend}(\text{IMM}))$
 - New hardware tests $rs == rt$ after register read
- **Reduced penalty (1 cycle) when branch taken**

Control Hazard Solutions

- **Stall** (software/hardware)
 - software: insert NOP by compiler
 - Hardware: stop loading instructions until result is available
- **Predict** (Hardware)
 - assume an outcome and continue fetching (undo if prediction is wrong)
 - loose cycles only on *mis-prediction*
 - » static branch prediction: base guess on instruction type
 - » dynamic branch prediction: base guess on execution history
- **Delayed branch** (software)
 - specify in architecture that the instruction immediately following branch is always executed
Compiler puts something useful (or a no-op) there.

Branch Behavior in Programs

- **Based on SPEC benchmarks on DLX**
 - Branches occur with a frequency of 14% to 16% in integer programs and 3% to 12% in floating point programs.
 - About 75% of the branches are forward branches
 - 60% of forward branches are taken
 - 80% of backward branches are taken
 - 67% of all branches are taken
- **Why are branches (especially backward branches) more likely to be taken than not taken?**

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 in-line 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

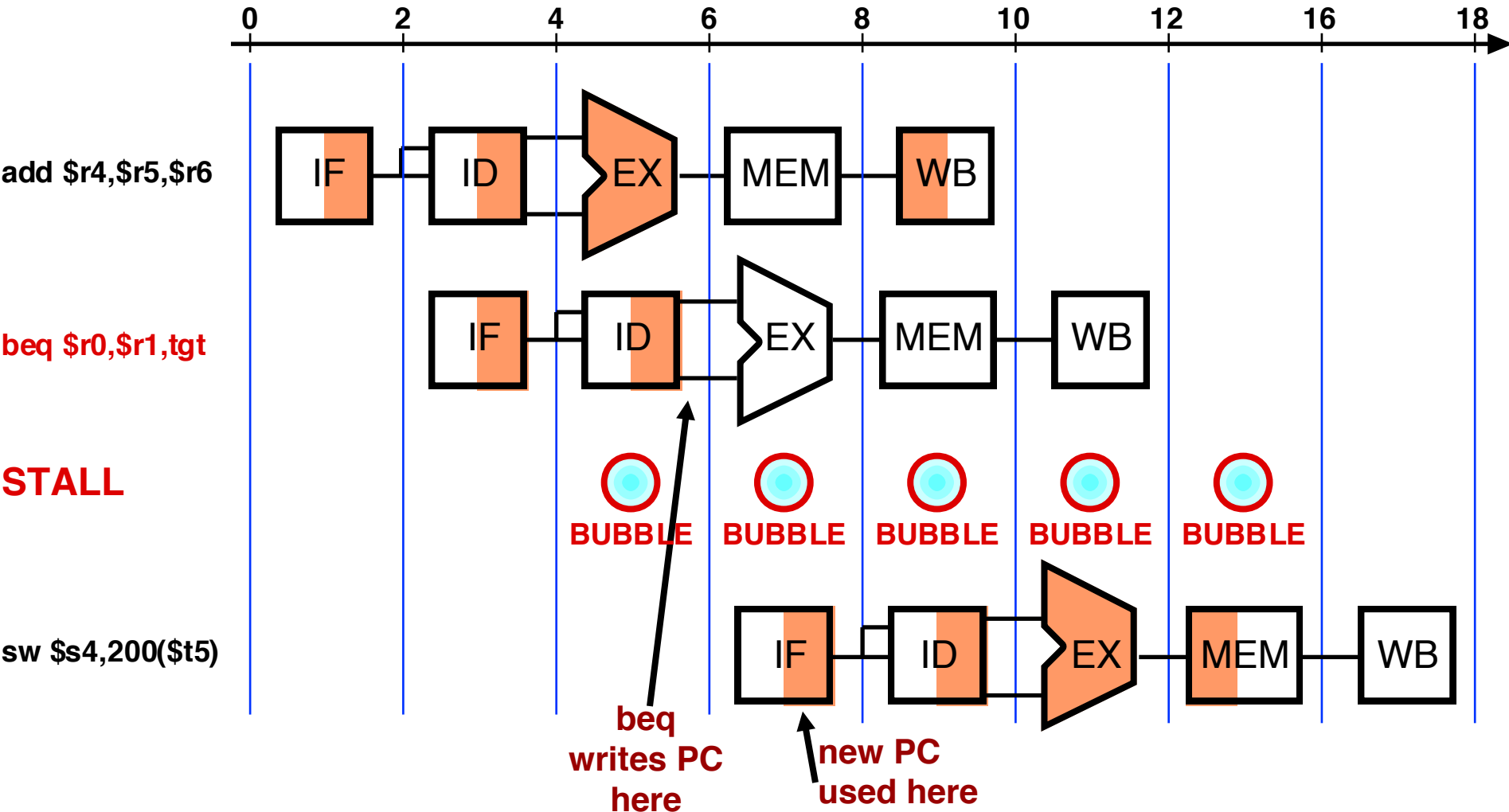
*Predicting branch **taken**:*

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/register

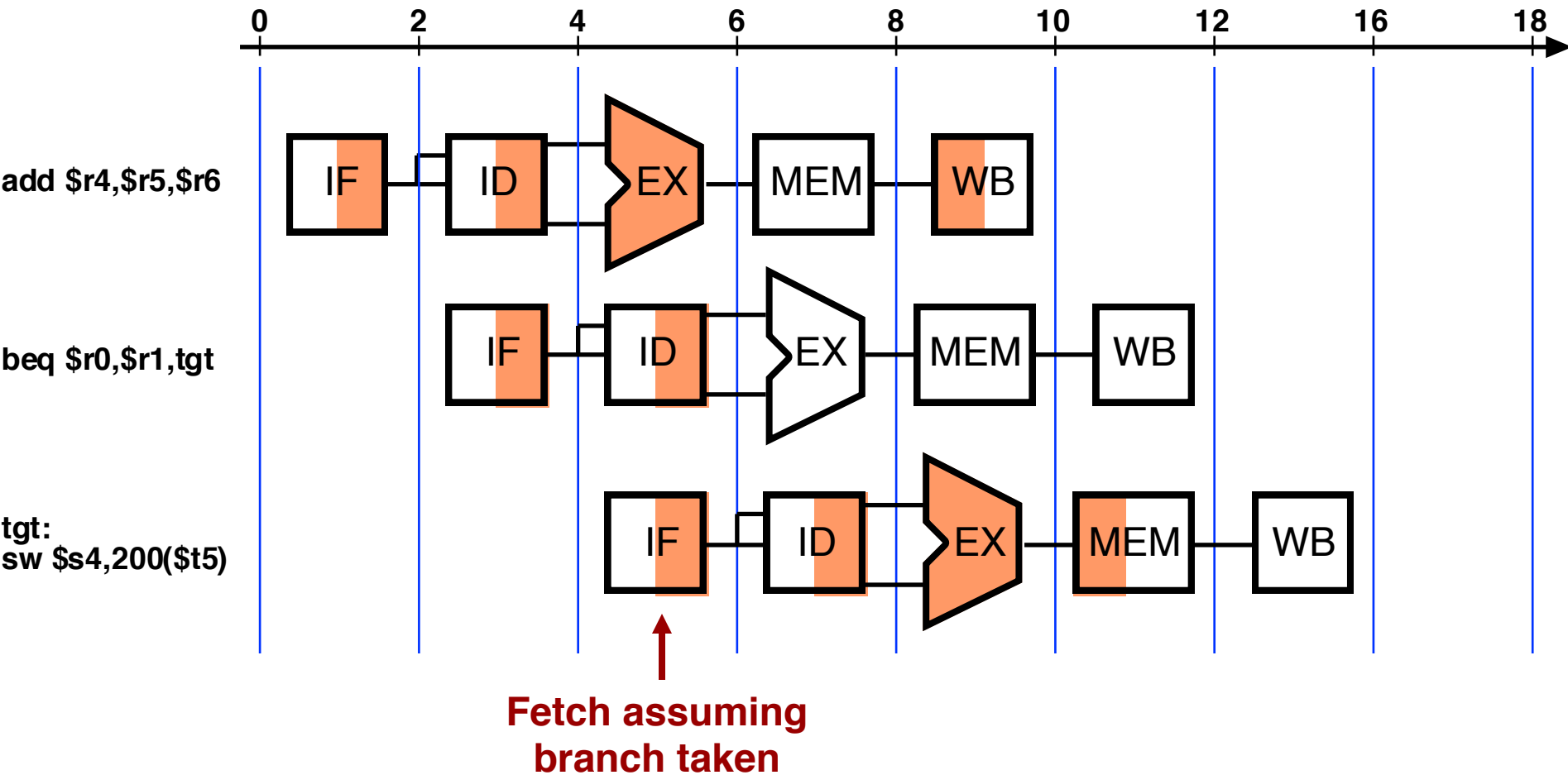
Predicting backward taken, forward not taken (BT FN)

Predict backward (loop) branches as taken, others not-taken

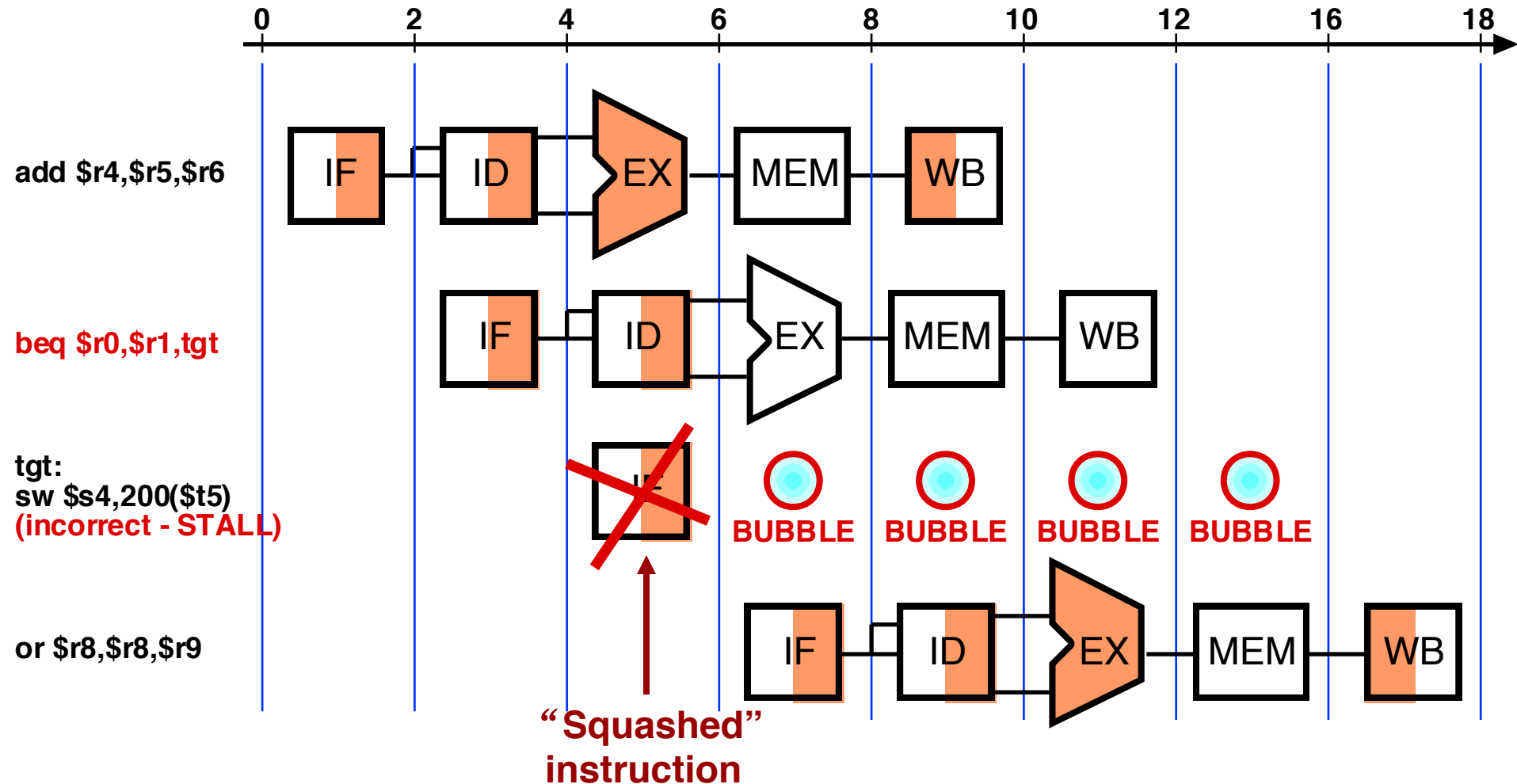
Control Hazard - Stall



Control Hazard - Correct Prediction



Control Hazard - Incorrect Prediction

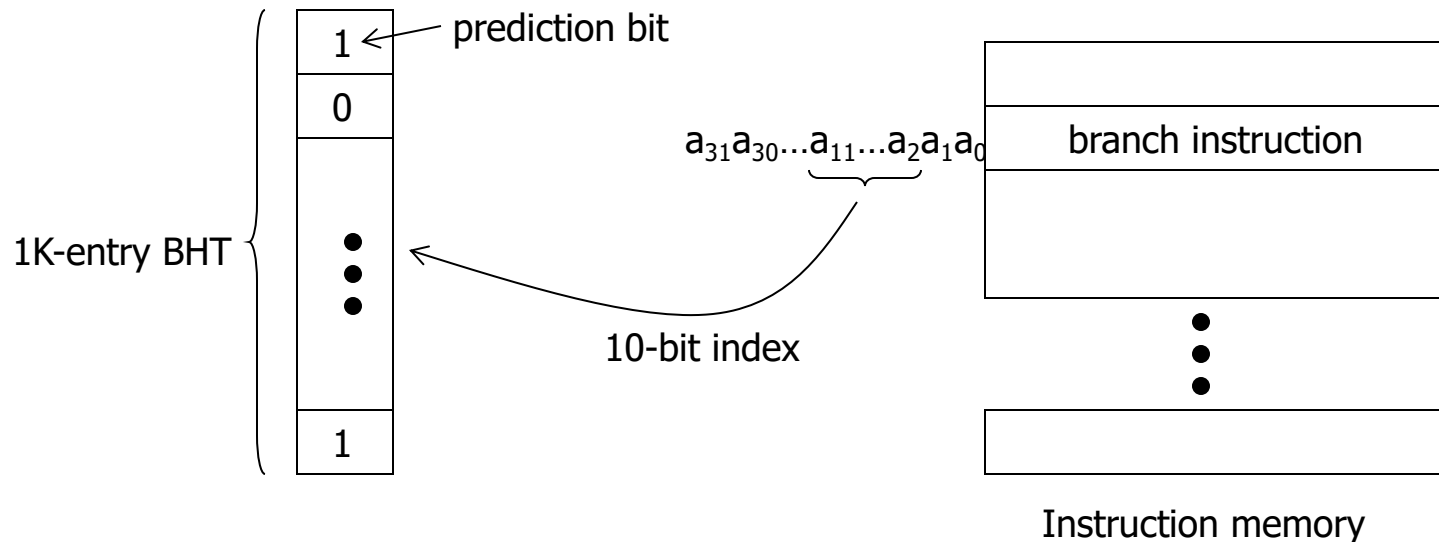


1-Bit Branch Prediction

- **Branch History Table (BHT): Lower bits of PC address index table of 1-bit values**
 - Says whether or not the branch was taken last time
 - No address check (saves HW, but may not be the 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 the 1-bit predictor for this branch assuming only this branch ever changes its corresponding prediction bit?

- **Answer: 80%.** Because there are two mispredictions – one on the first iteration and one on the last iteration. *Is this good enough and Why?*

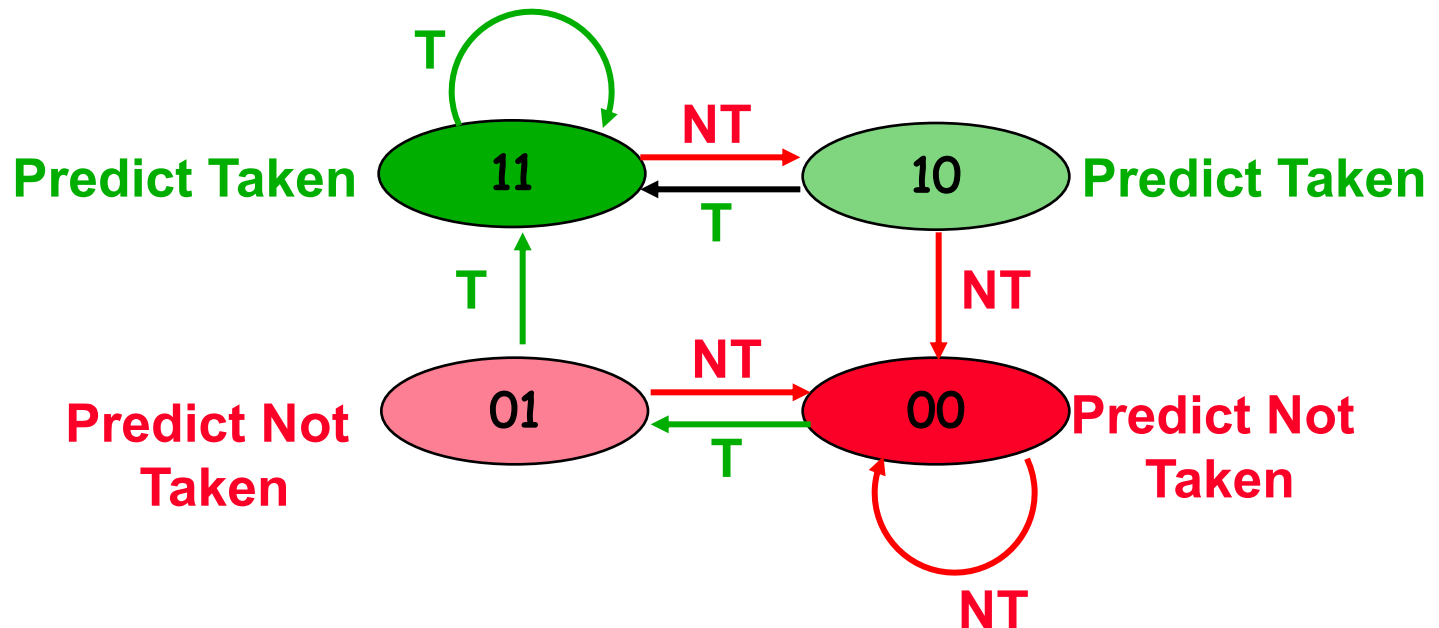
2-Bit Branch Prediction

(Jim Smith, 1981)

- Solution: a 2-bit scheme where prediction is changed only if mispredicted *twice*

Red: not taken

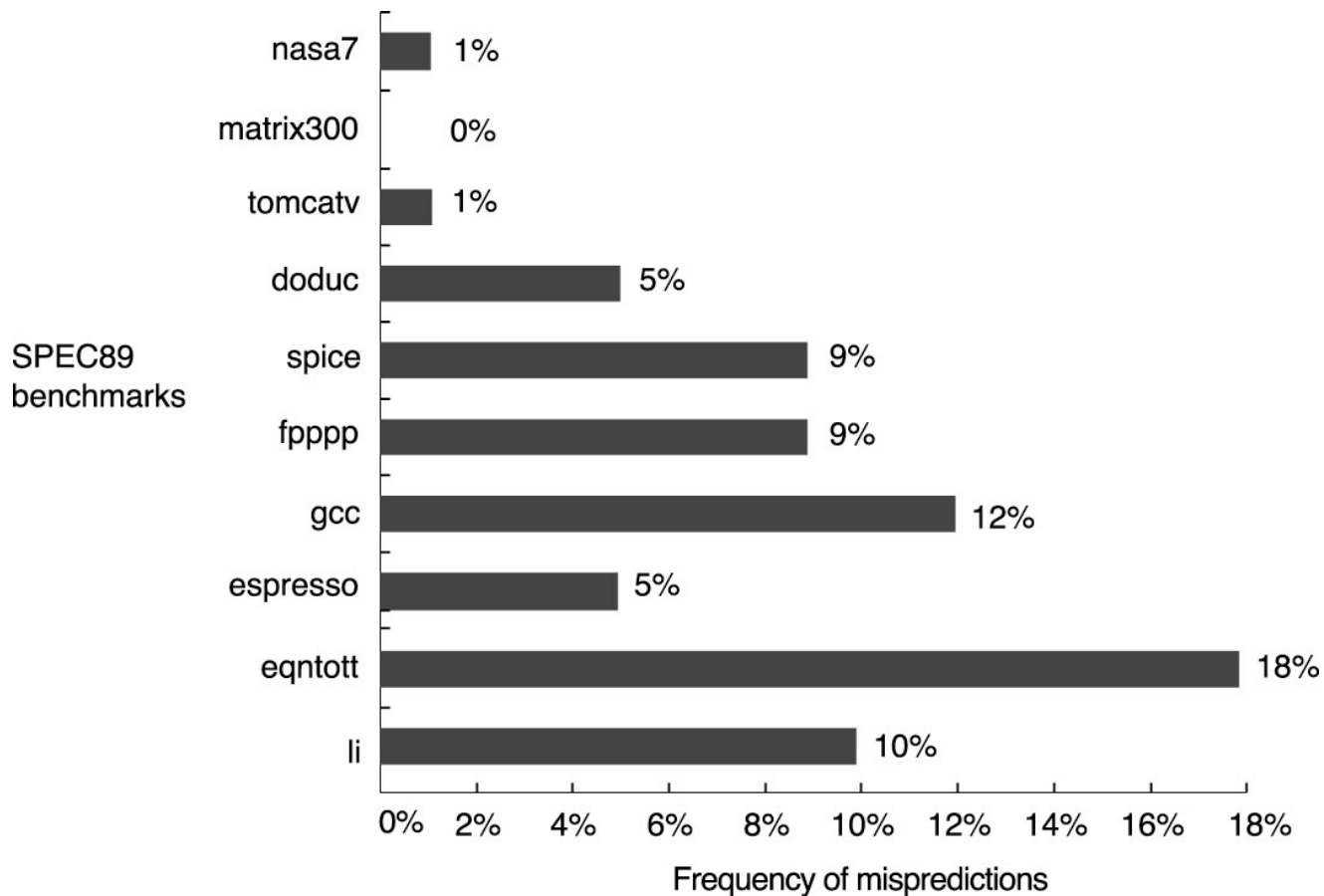
Green: taken



n-bit Saturating Counter

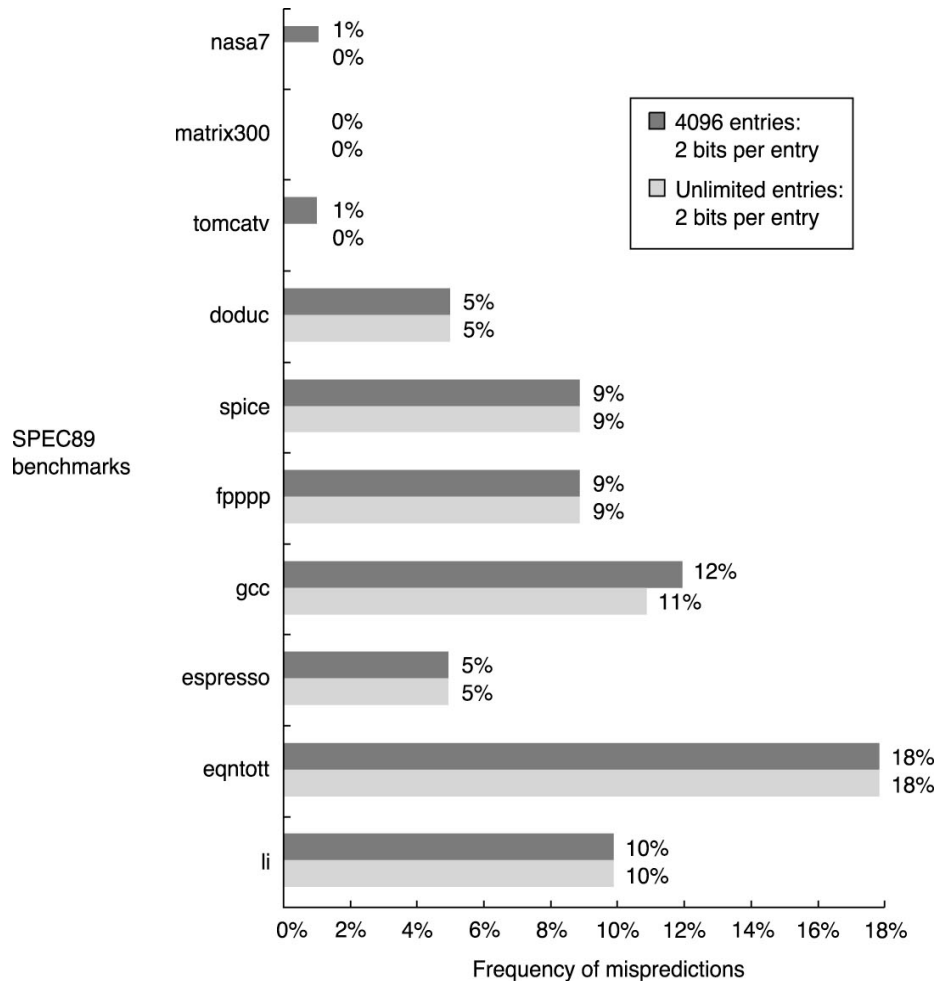
- Values: $0 \sim 2^n - 1$**
- counter can hold values between 0 and $2^n - 1$**
- predict taken when value is greater than or equal to half of maximum value:**
- The counter is incremented on each taken branch**
- and decremented on each not taken branch**
- Studies have shown that the 2-bit predictors do almost as 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

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

Correlated Predictor

The rationale:

- having the prediction depend on the outcome of only 1 branch might produce bad predictions
- some branch outcomes are correlated

example: same condition variable

```
if (d==0)
```

```
...
```

```
if (d!=0)
```

example: related condition variable

```
if (d==0)
```

```
    b=1;
```

```
if (b==1)
```

Correlated Predictor

another example: related condition variables

```
if (x==2)                /* branch 1 */
    x=0;
if (y==2)                /* branch 2 */
    y=0;
if (x!=y)                /* branch 3 */
    do this; else do that;
```

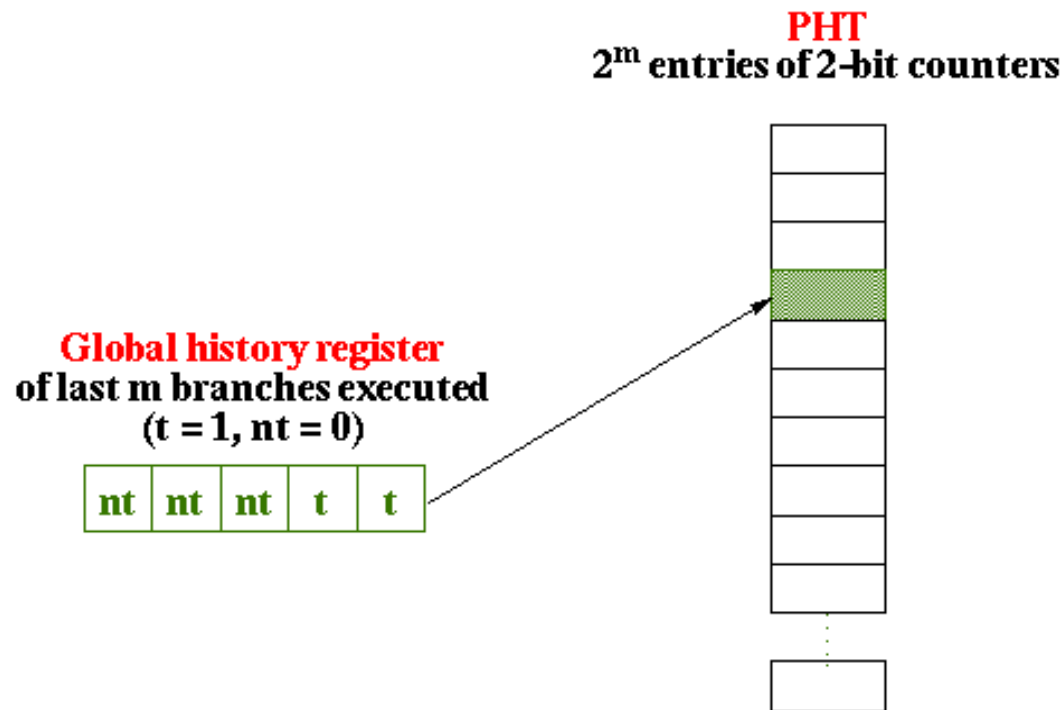
» if branches 1 & 2 are taken, branch 3 is not taken

⇒ use a **history of the past m branches**
represents a path through the program
(but still n bits of prediction)

Correlated Predictor

General idea of correlated branch prediction:

- put the global branch history in a **global history register**
 - » global history is a **shift register**: shift left in the new branch outcome
- use its value to access a **pattern history table (PHT)** of 2-bit saturating counters



Tournament Predictors

- A local predictor might work well for some branches or programs, while a global predictor might work well for others
- Provide one of each and maintain another predictor to identify which predictor is best for each branch

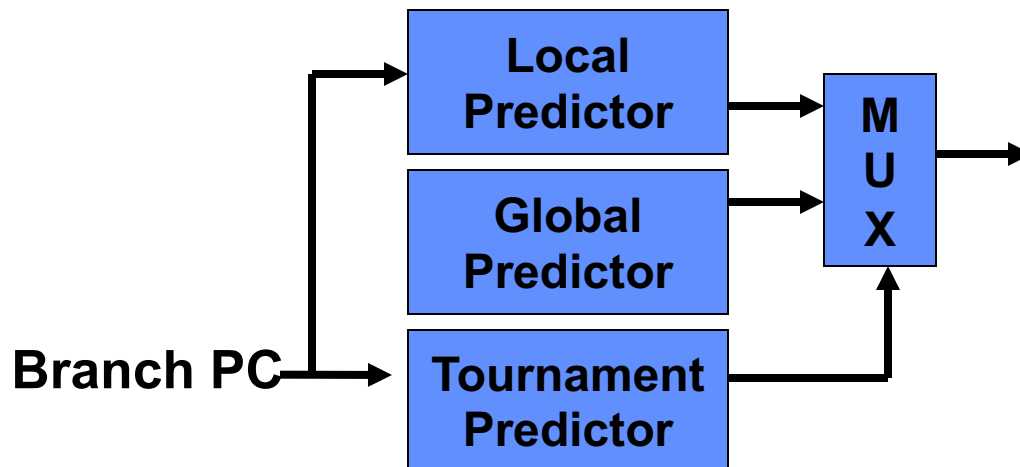
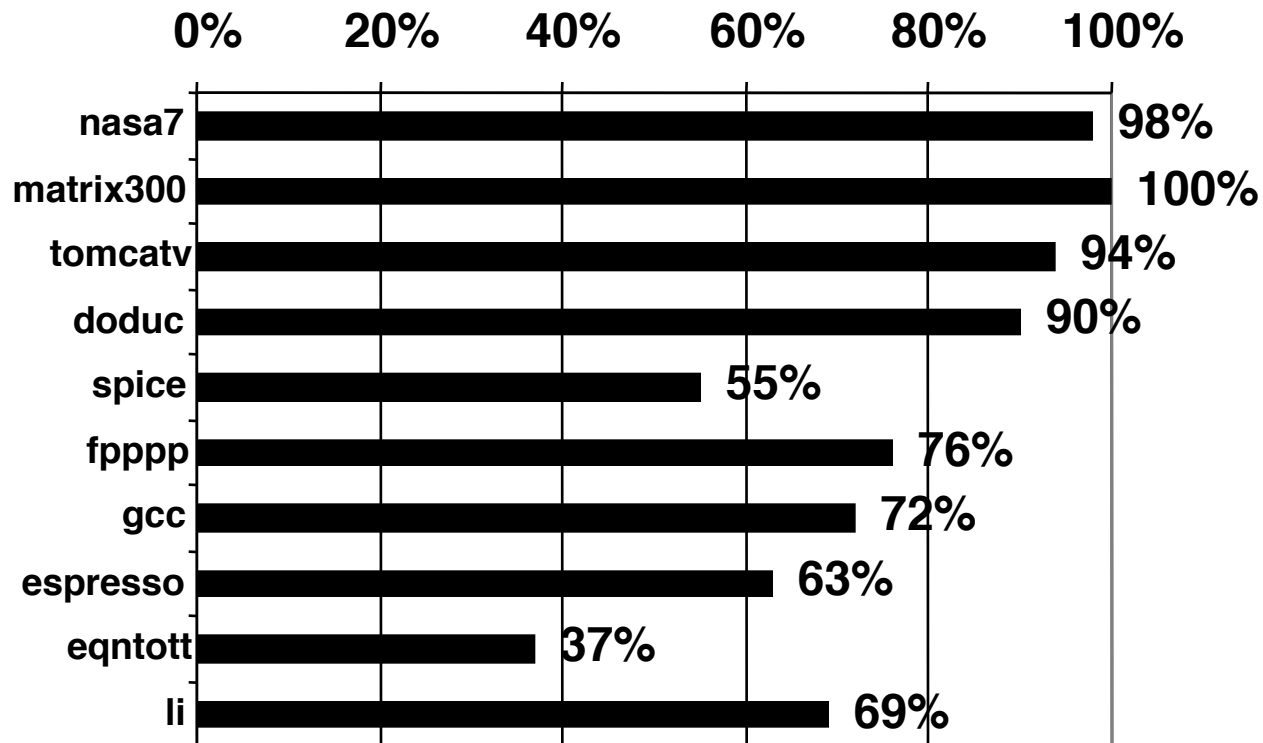
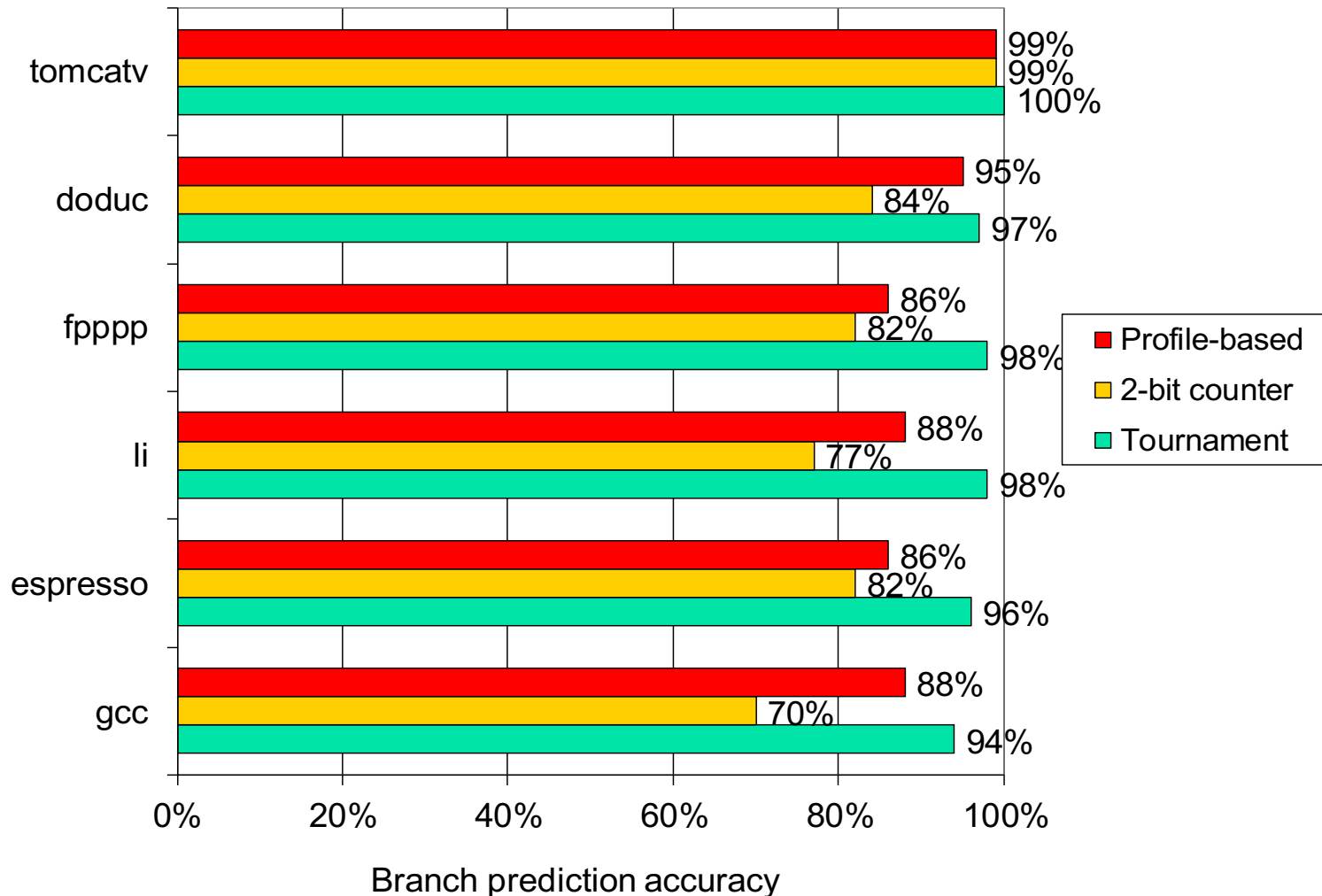


Table of 2-bit
saturating counters

% of predictions from local predictor in Tournament Prediction Scheme



Accuracy of Branch Prediction



Profile: branch profile from last execution (static in that the prediction is in encoded in the instruction, but derived from the real execution profile) • A good dynamic predictor can outperform profile-driven static prediction by a large margin

Tournament Predictor

Combine branch predictors

- local, per-branch prediction, accessed by the PC
- correlated prediction based on the last m branches, assessed by the global history
- indicator of which had been the best predictor for this branch
 - » 2-bit counter: increase for one, decrease for the other
- Compaq Alpha 21264
- ~5% misprediction on SPEC95
- 2% of die

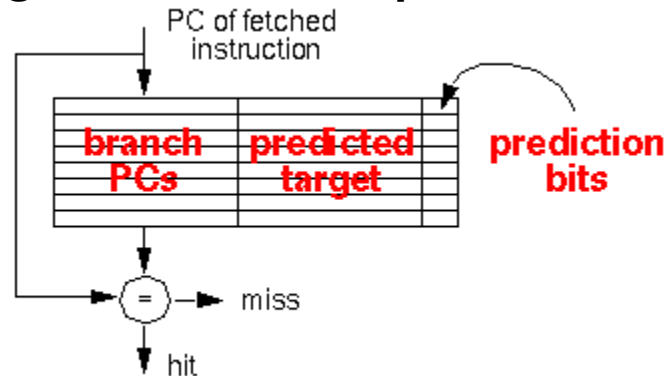
Branch Target Buffer (BTB)

Cache that stores: the PCs of branches (*tag*)
the predicted target address (*data*)
branch prediction bits (*data*)

Accessed by PC address in fetch stage

if hit: address was for *this* branch instruction

fetch the target instruction if prediction bits say taken



No branch delay if: branch found in BTB

prediction is correct

(assume BTB update is done in the next cycles)

Fetch Both Targets

Fetch target & fall-through code

- reduces the misprediction penalty
- but requires lots of I-cache bandwidth
 - » a dual-ported instruction cache
 - » requires independent bank accessing
 - » wide cache-to-pipeline buses

Calculating the Cost of Branches

Factors to consider:

- **branch frequency (every 4-6 instructions)**
- **correct prediction rate**
 - » **1 bit: ~ 80% to 85%**
 - » **2 bit: ~ high 80s to 90%**
 - » **correlated branch prediction: ~ 95%**
- **misprediction penalty**
 - Alpha 21164: 5 cycles; 21264: 7 cycles**
 - UltraSPARC 4 cycles**
 - Pentium Pro: at least 9 cycles, 15 on average**
- **or misfetch penalty**
 - have the correct prediction but not know the target address yet**
 - (may also apply to unconditional branches)**

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