Computer Architecture

An Introduction

Virendra Singh

Associate Professor

Computer Architecture and Dependable Systems Lab

Department of Electrical Engineering

Indian Institute of Technology Bombay

http://www.ee.iitb.ac.in/~viren/

E-mail: viren@ee.iitb.ac.in

CS-683: Advanced Computer Architecture

Lecture 5 (09 Aug 2013)

CADSL

Advantages of Pipeline

- One instruction is completed each cycle; CPI ≈ 1, neglecting the initial pipeline latency of n cycles.
  - *Pipeline latency is defined as the number of stages in the pipeline, or*
  - *The number of clock cycles after which the first instruction is completed.*
- The clock cycle time is about four times shorter than that of single-cycle datapath and about the same as that of multicycle datapath.
- So, pipelined execution is faster, but . . .

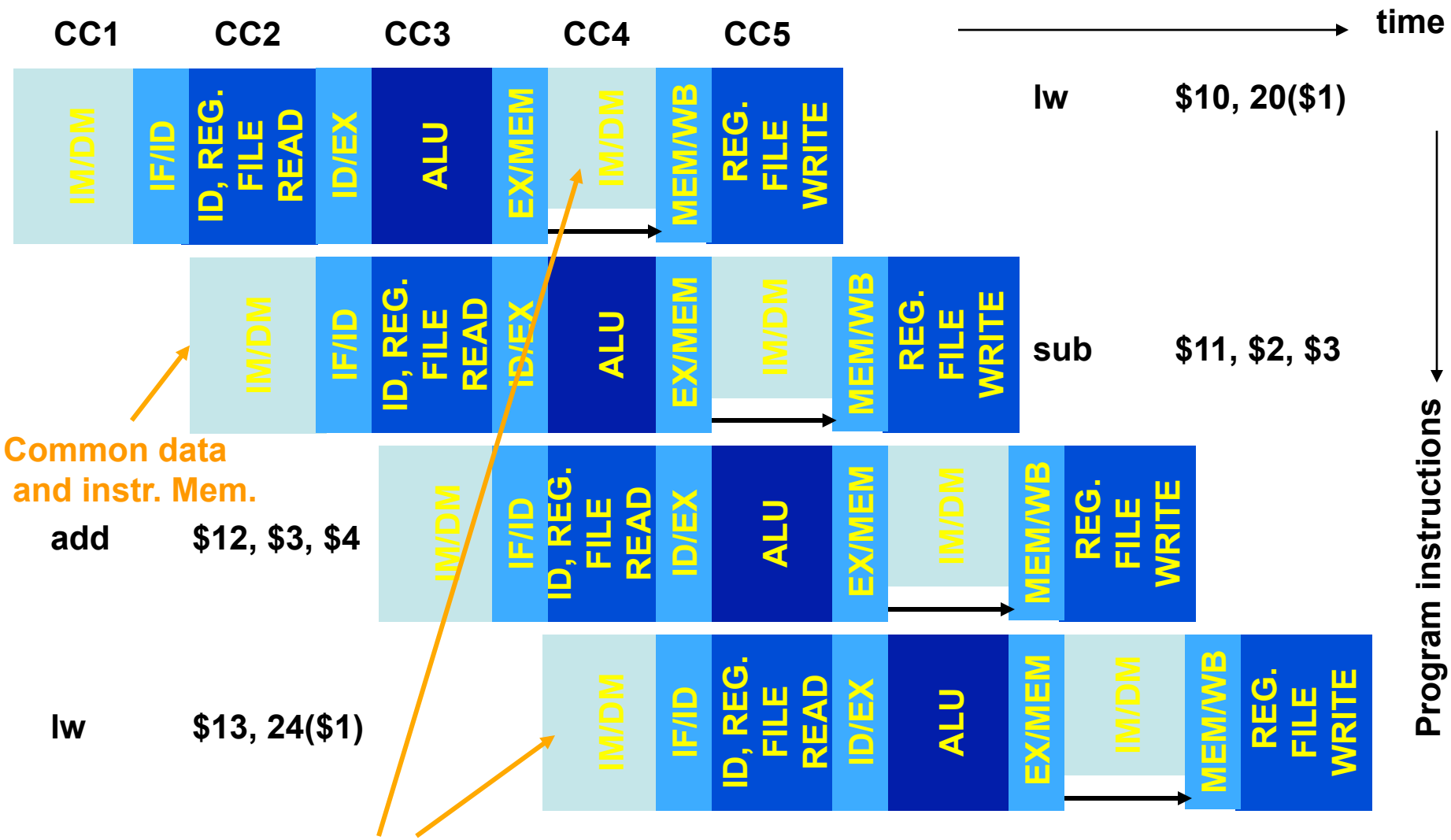
Pipeline Hazards

- Definition: *Hazard in a pipeline is a situation in which the next instruction cannot complete execution one clock cycle after completion of the present instruction.*
- Three types of hazards:
  - Structural hazard (resource conflict)
  - Data hazard
  - Control hazard

Structural Hazard

- Two instructions cannot execute due to a resource conflict.
- Example: Consider a computer with a common data and instruction memory. The fourth cycle of a *lw* instruction requires memory access (memory read) and at the same time the first cycle of the fourth instruction requires instruction fetch (memory read). This will cause a memory resource conflict.

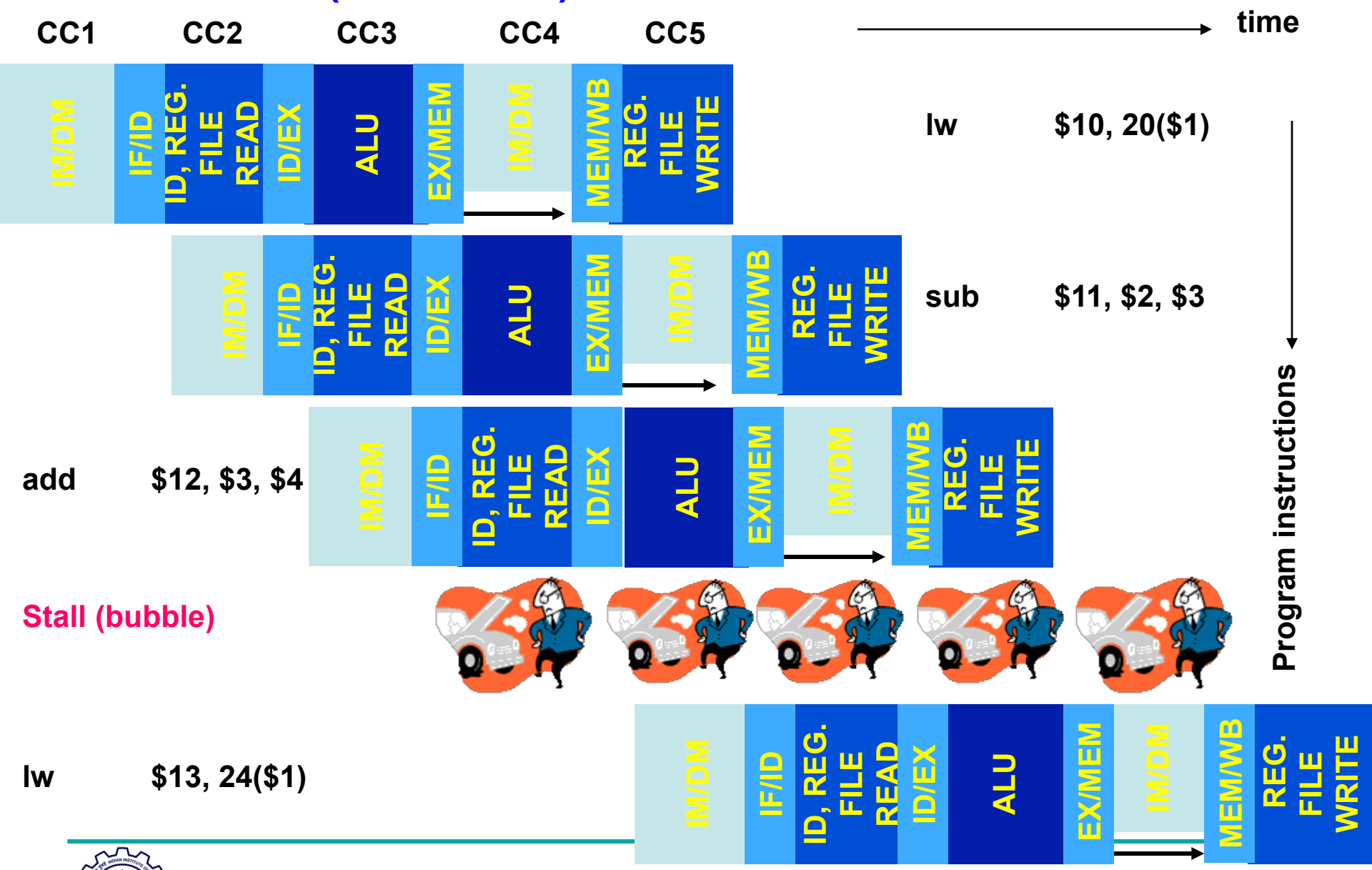
Example of Structural Hazard



Possible Remedies for Structural Hazards

- Provide duplicate hardware resources in datapath.
- Control unit or compiler can insert delays (no-op cycles) between instructions. This is known as pipeline *stall* or *bubble*.

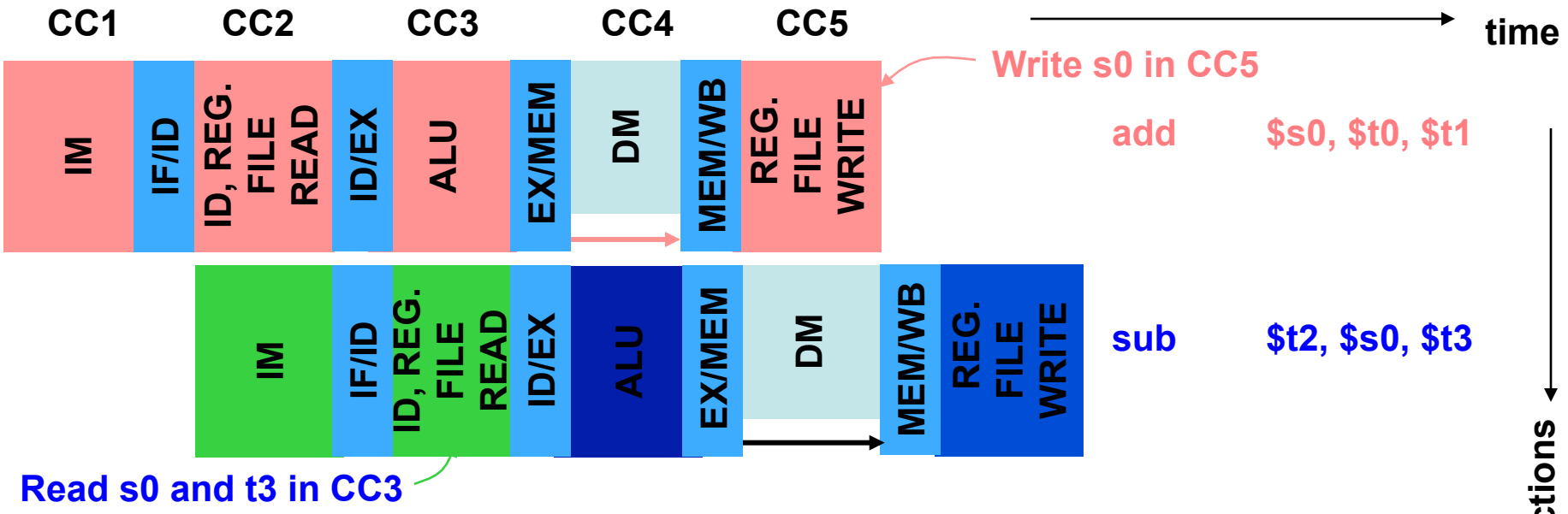
Stall (Bubble) for Structural Hazard



Data Hazard

- Data hazard means that an instruction cannot be completed because the needed data, to be generated by another instruction in the pipeline, is not available.
- Example: consider two instructions:
  - ✧ add $s0, $t0, $t1
  - ✧ sub $t2, $s0, $t3 # needs $s0

Example of Data Hazard



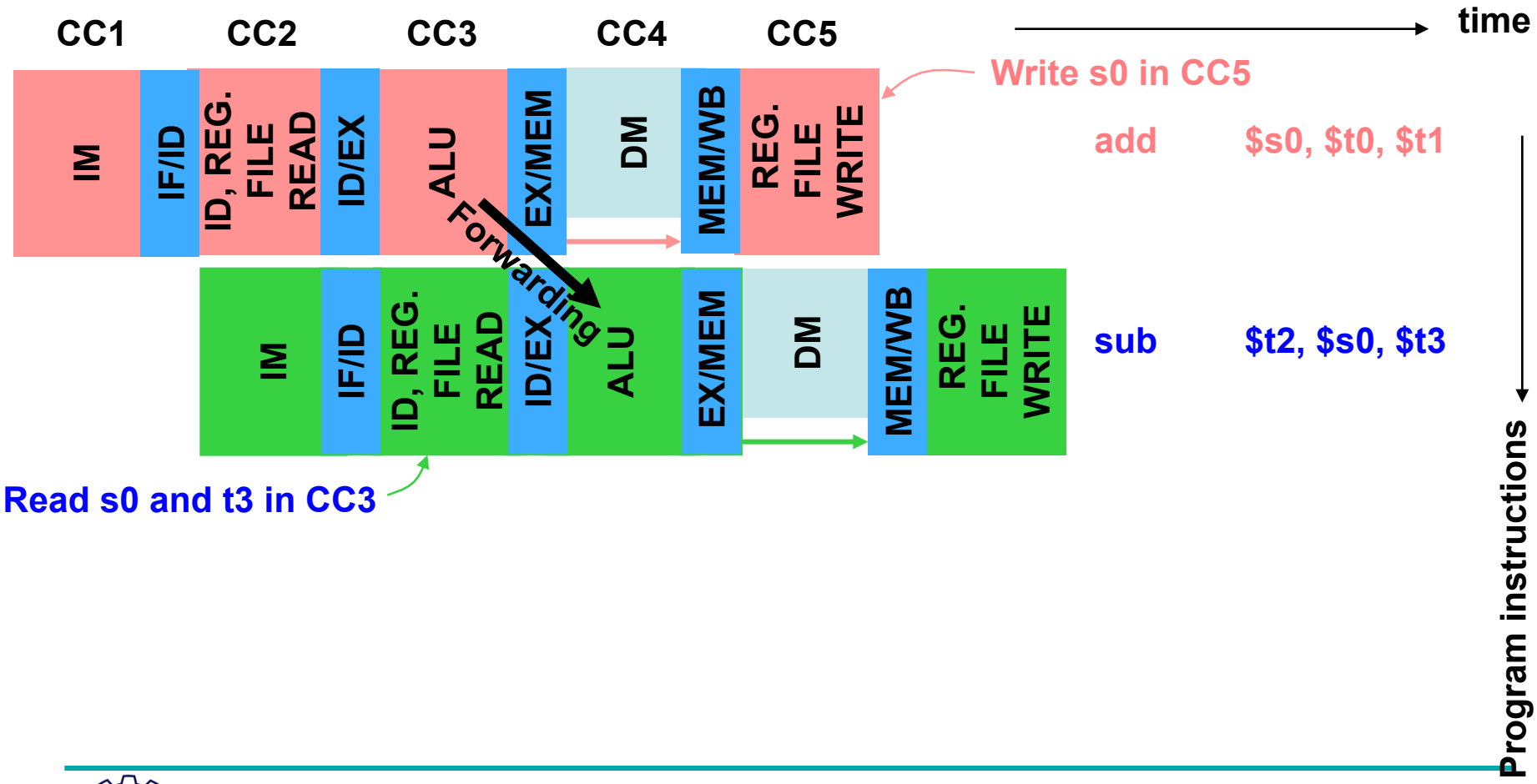
We need to read s0 from reg file in cycle 3
But s0 will not be written in reg file until cycle 5

However, s0 will only be used in cycle 4
And it is available at the end of cycle 3

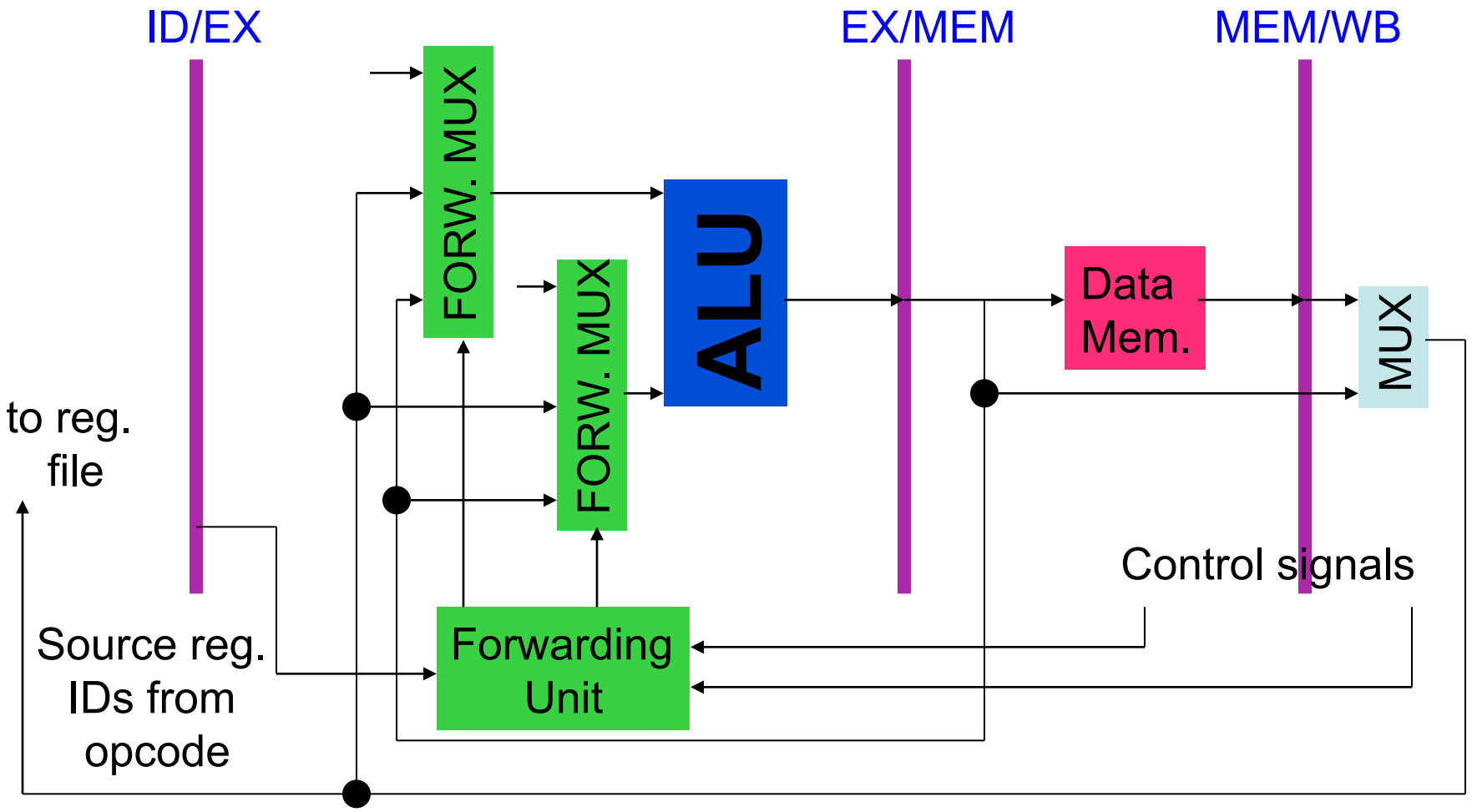
Forwarding or Bypassing

- Output of a resource used by an instruction is forwarded to the input of some resource being used by another instruction.
- Forwarding can eliminate some, but not all, data hazards.

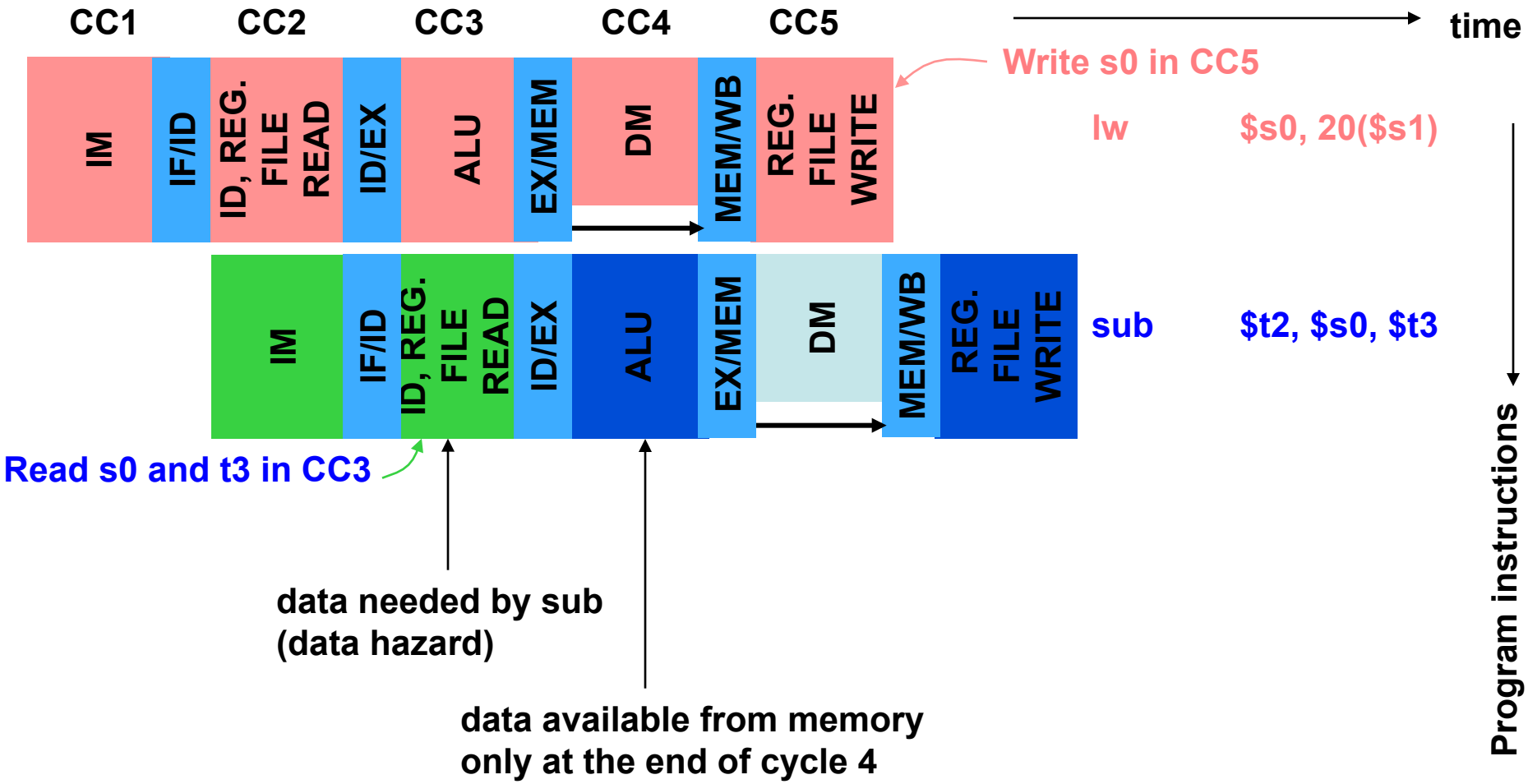
Forwarding for Data Hazard



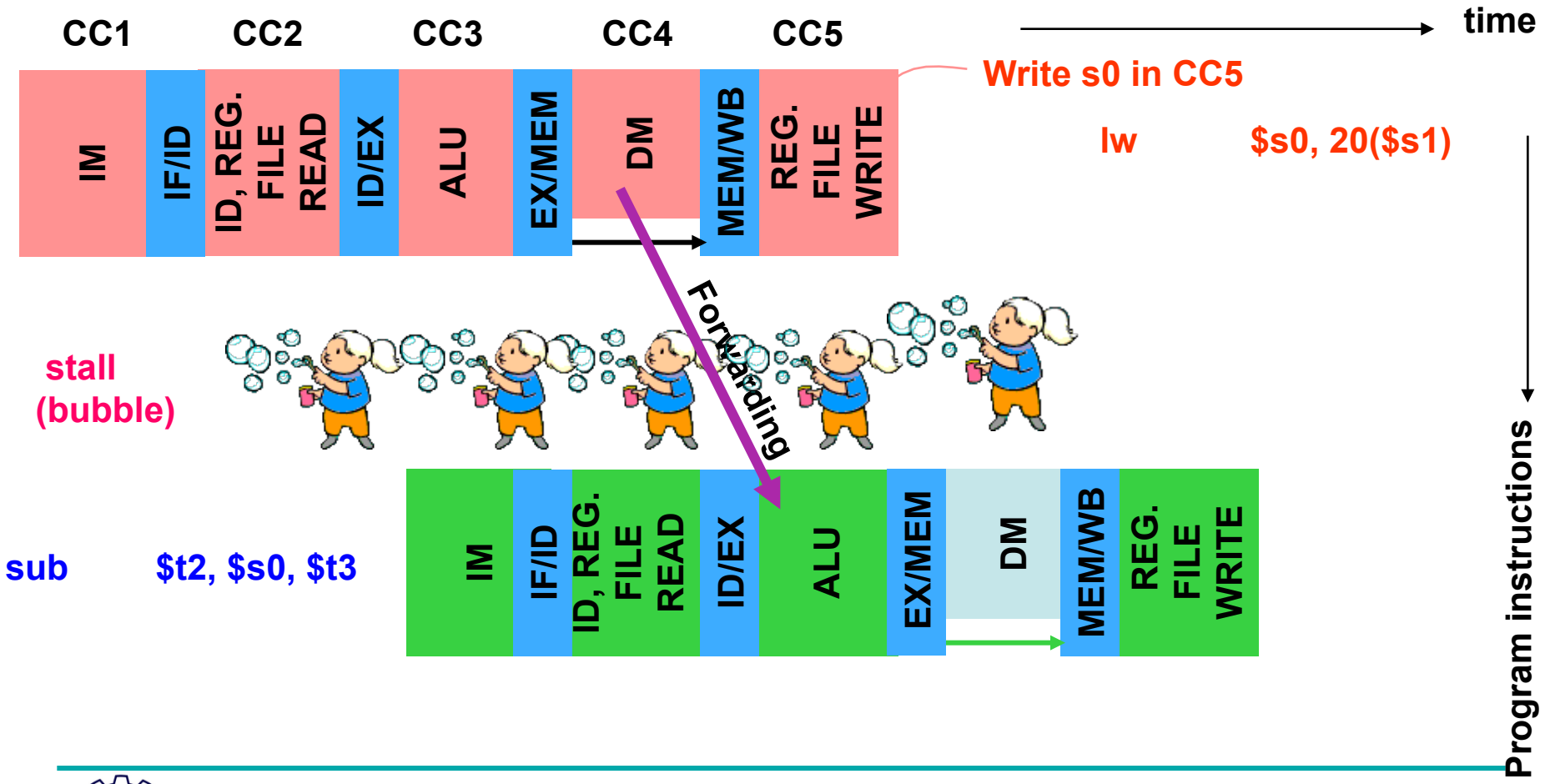
Forwarding Unit Hardware



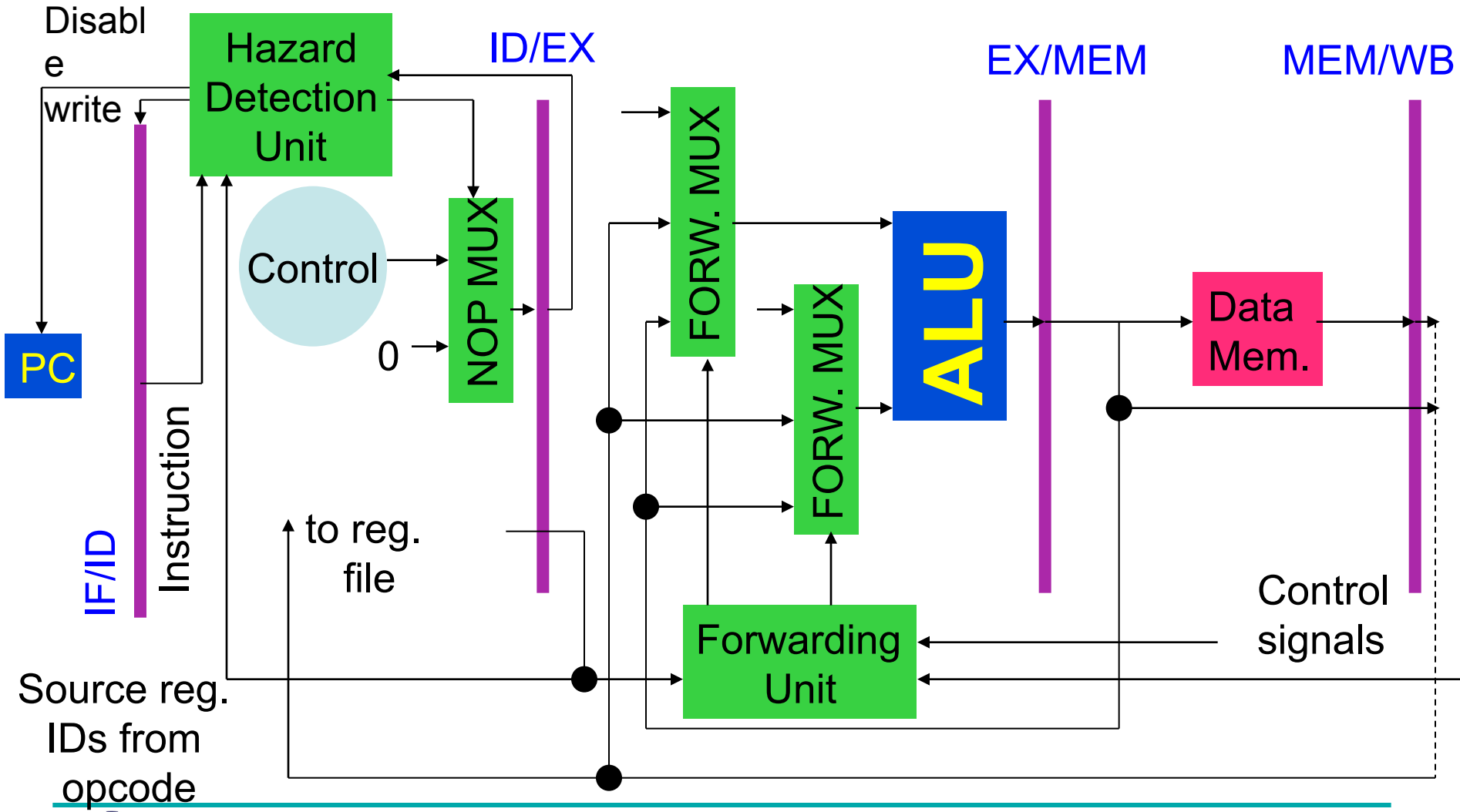
Forwarding Alone May Not Work



Use Bubble and Forwarding



Hazard Detection Unit Hardware



Resolving Hazards

- Hazards are resolved by Hazard detection and forwarding units.
- Compiler's understanding of how these units work can improve performance.

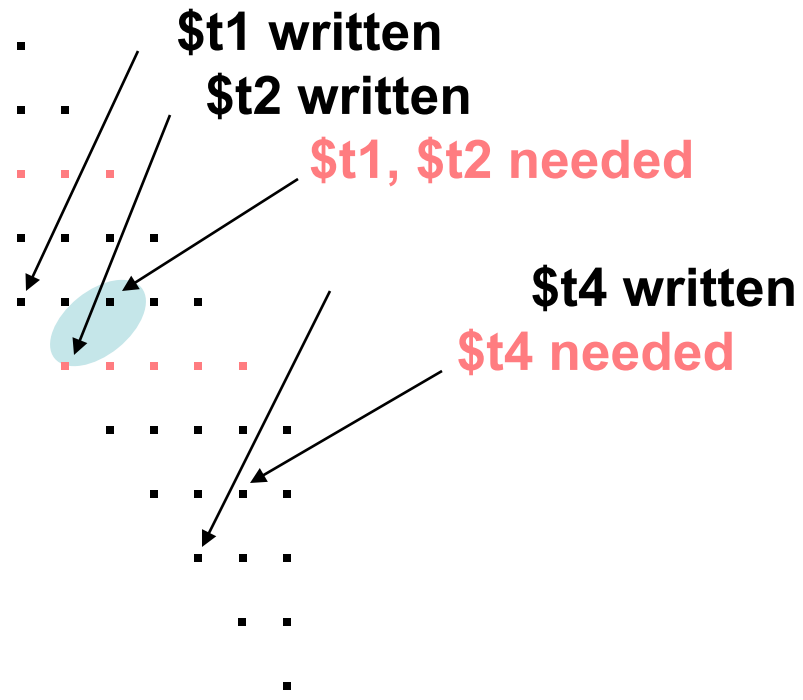
Avoiding Stall by Code Reorder

C code:

```
A = B + E;
C = B + F;
```

MIPS code:

```
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)
```

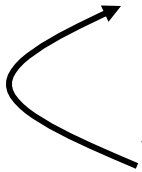


Reordered Code

C code:

```
A = B + E;
C = B + F;
```

MIPS code:



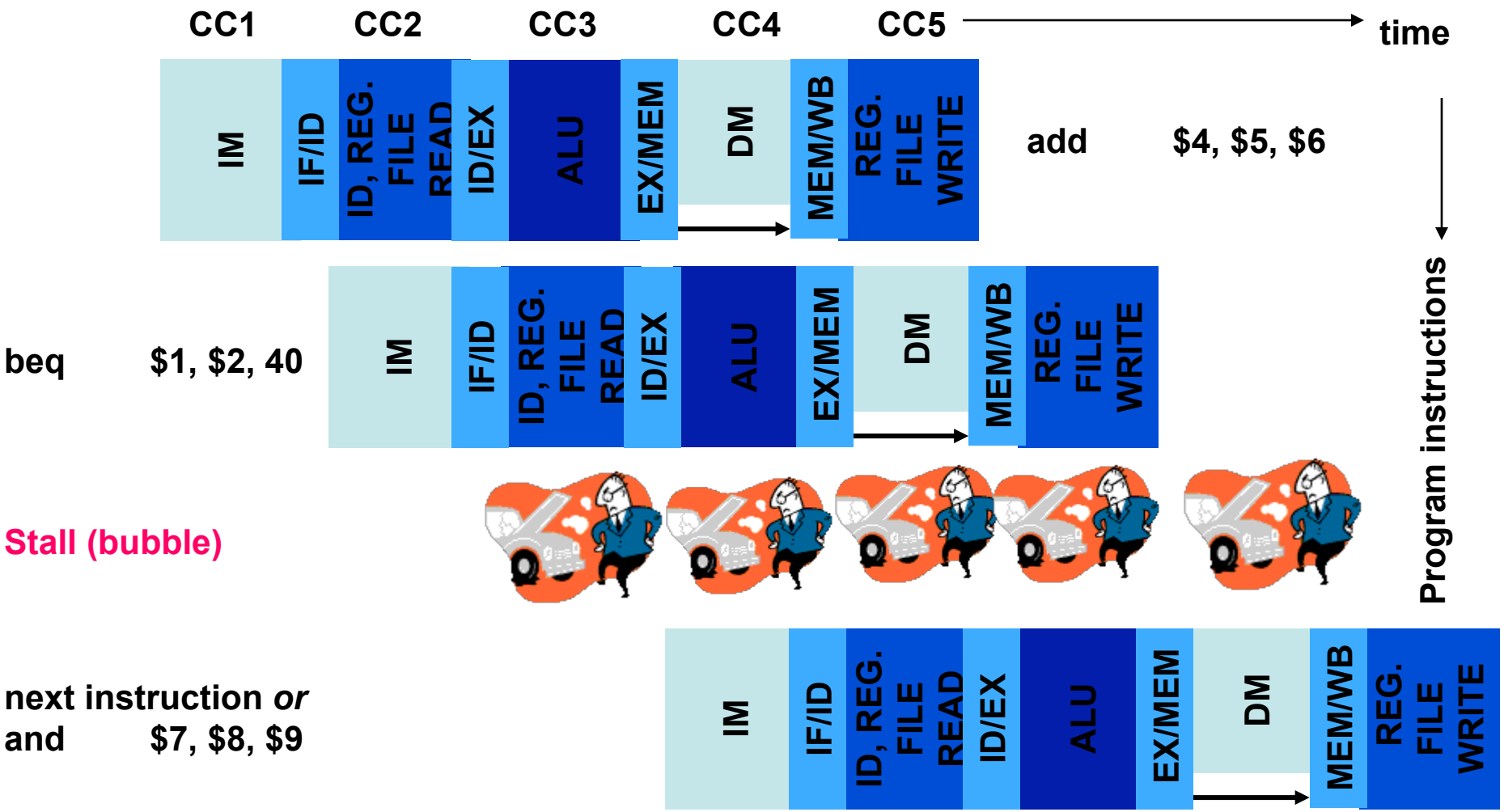
```
lw    $t1,   0($t0)
lw    $t2,   4($t0)
lw    $t4,   8($t0)
add   $t3,   $t1, $t2      no hazard
sw    $t3,   12($t0)
add   $t5,   $t1, $t4      no hazard
sw    $t5,   16,($t0)
```

Control Hazard

- Instruction to be fetched is not known!
- Example: Instruction being executed is branch-type, which will determine the next instruction:

```
        add    $4, $5, $6
        beq    $1, $2, 40
        next instruction
        . . .
40      and    $7, $8, $9
```

Stall on Branch



Why Only One Stall?

- Extra hardware in ID phase:
  - Additional ALU to compute branch address
  - Comparator to generate zero signal
  - Hazard detection unit writes the branch address in PC

Ways to Handle Branch

- Stall or bubble
- Delayed branch
- Branch prediction:
  - Heuristics
    - Next instruction
    - Prediction based on statistics (dynamic)
    - Hardware decision (dynamic)
  - Prediction error: pipeline flush

Delayed Branch Example

- Stall on branch

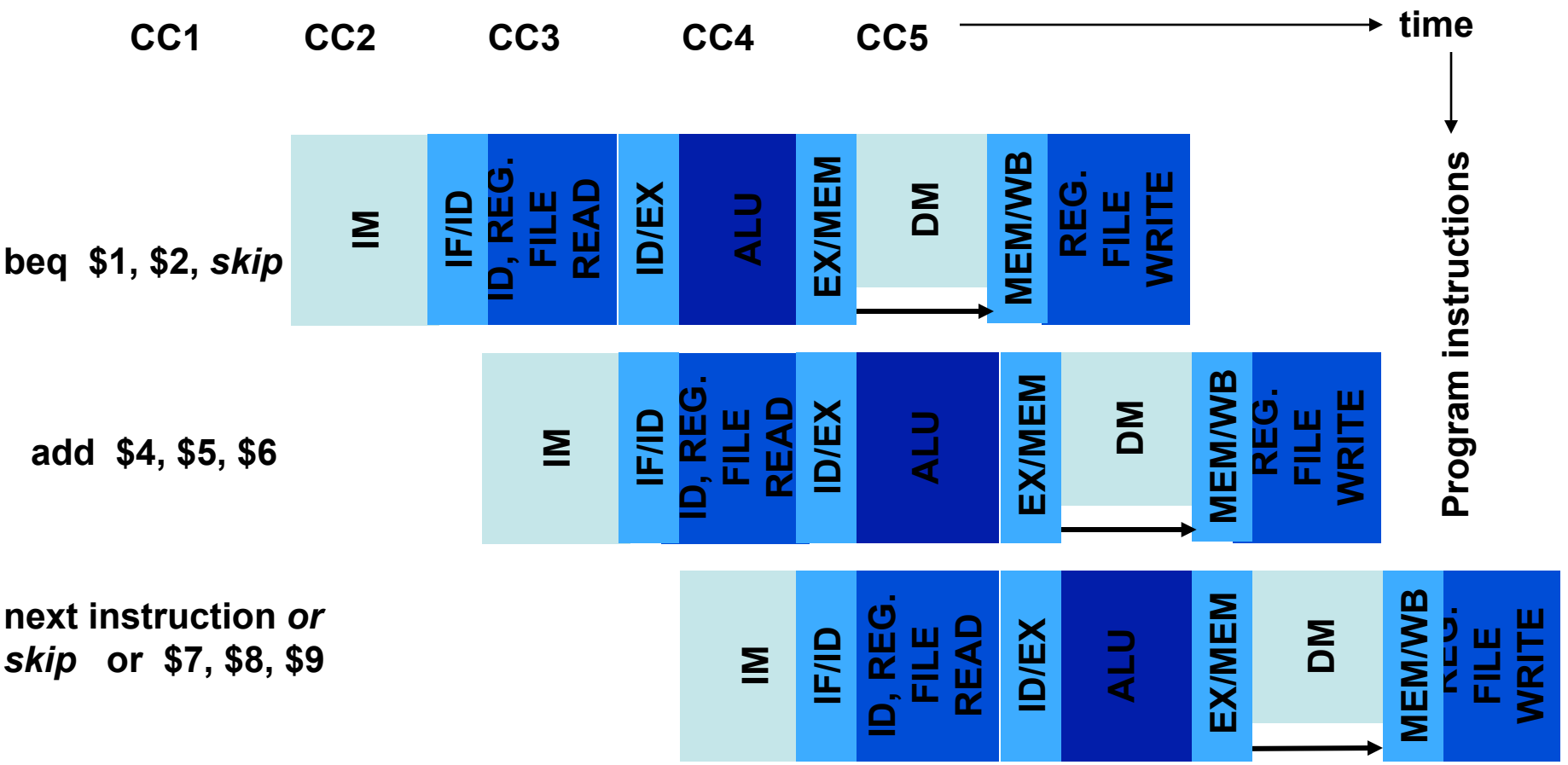
```
        add $4, $5, $6
        beq $1, $2, skip
        next instruction
        . . .
skip    or $7, $8, $9
```

- Delayed branch

```
        beq $1, $2, skip
        add $4, $5, $6
        next instruction
        . . .
skip    or $7, $8, $9
```

Instruction executed irrespective of branch decision

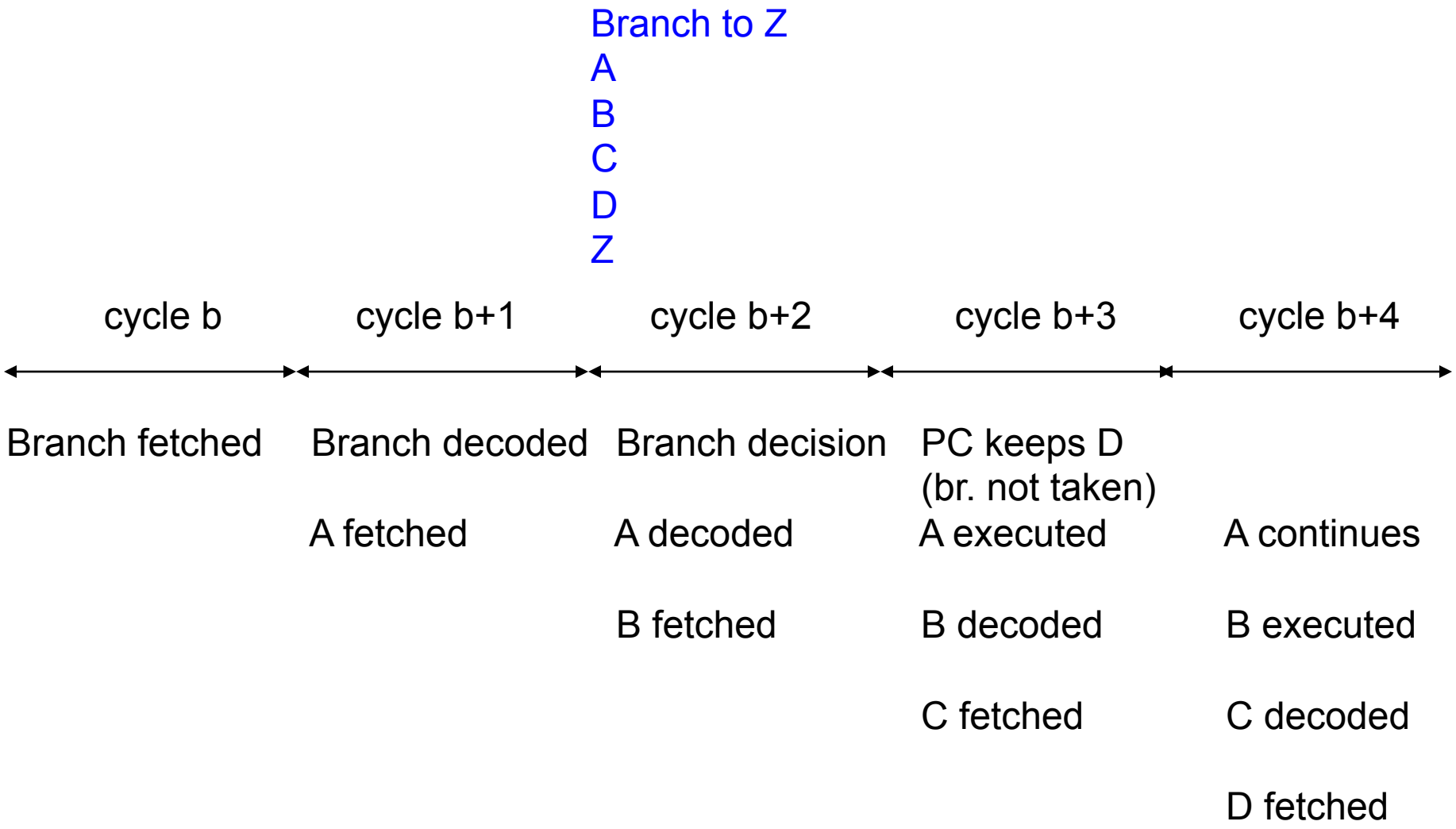
Delayed Branch



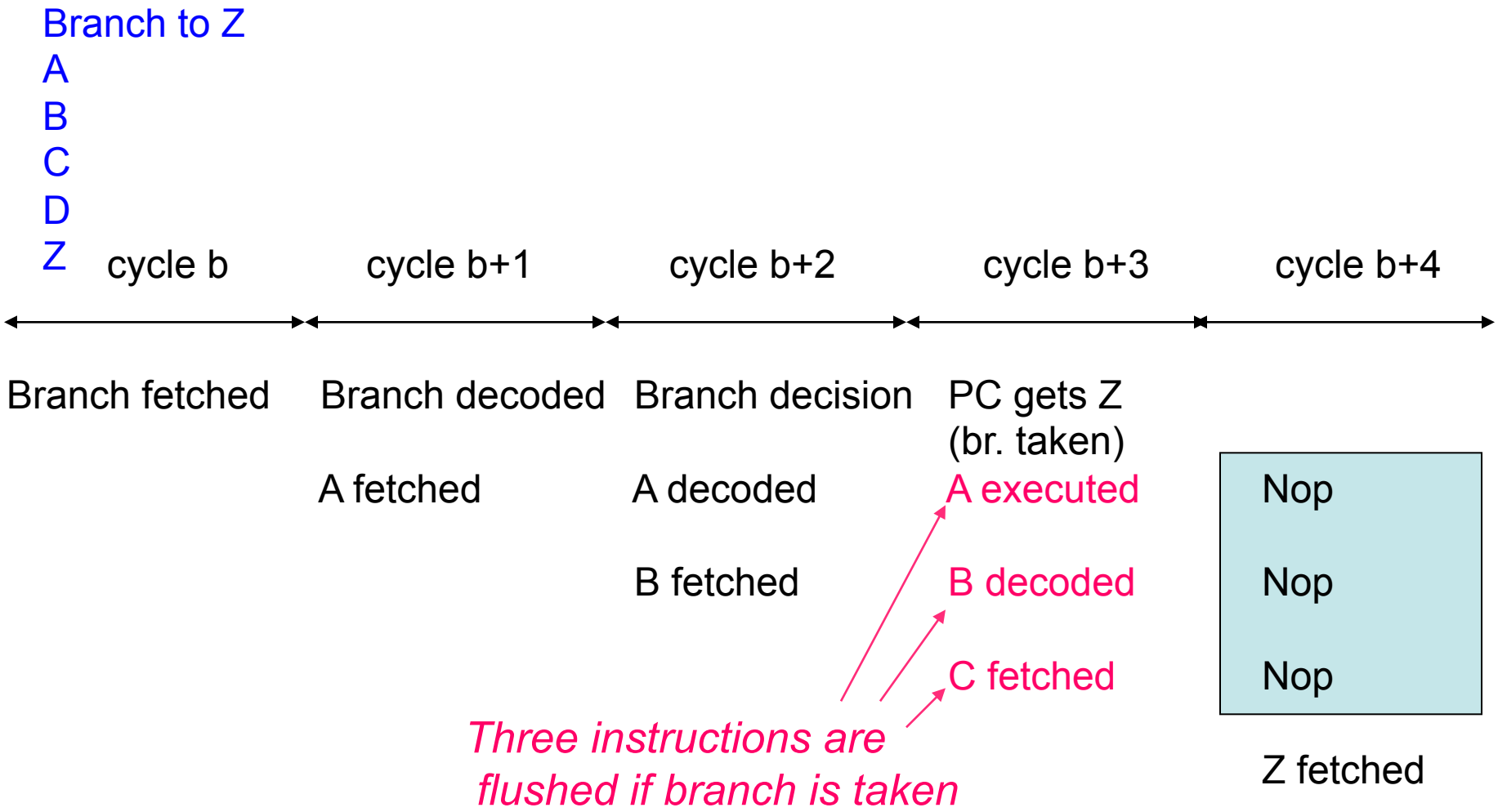
Branch Hazard

- Consider heuristic – branch not taken.
- Continue fetching instructions in sequence following the branch instructions.
- If branch is taken (indicated by *zero* output of ALU):
  - Control generates *branch* signal in ID cycle.
  - *branch* activates *PCSource* signal in the MEM cycle to load PC with new branch address.
  - *Three instructions in the pipeline must be flushed if branch is taken – can this penalty be reduced?*

Branch Not Taken

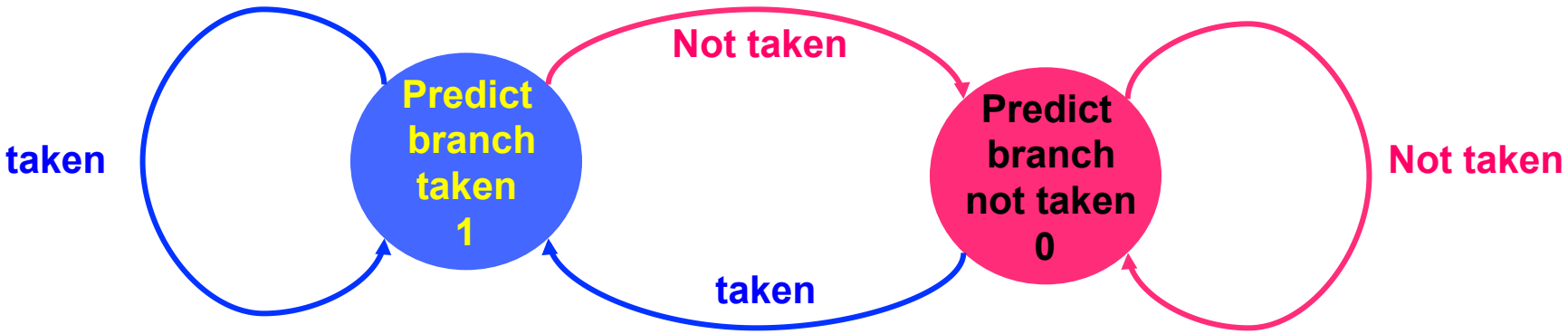


Branch Taken

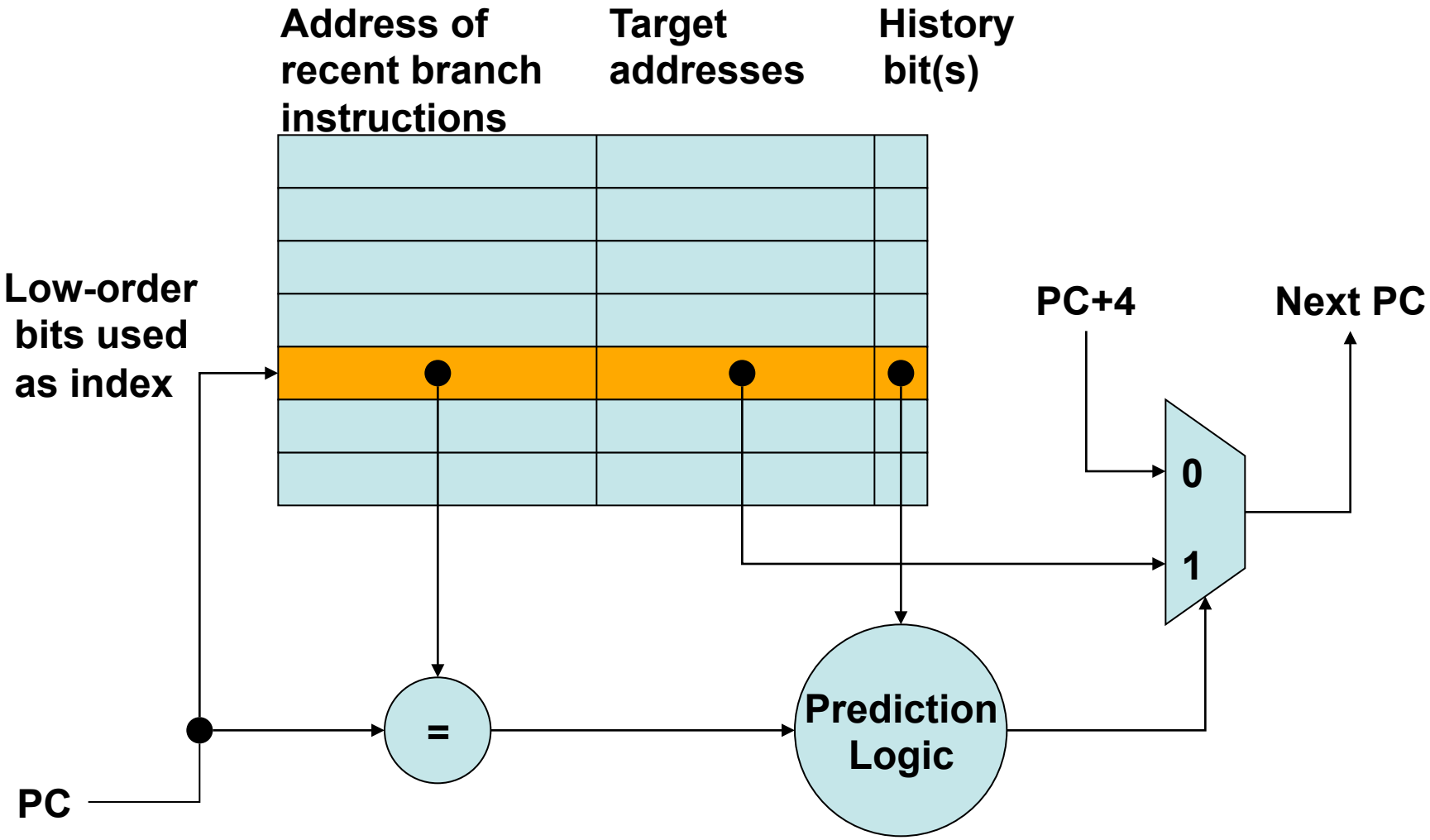


Branch Prediction

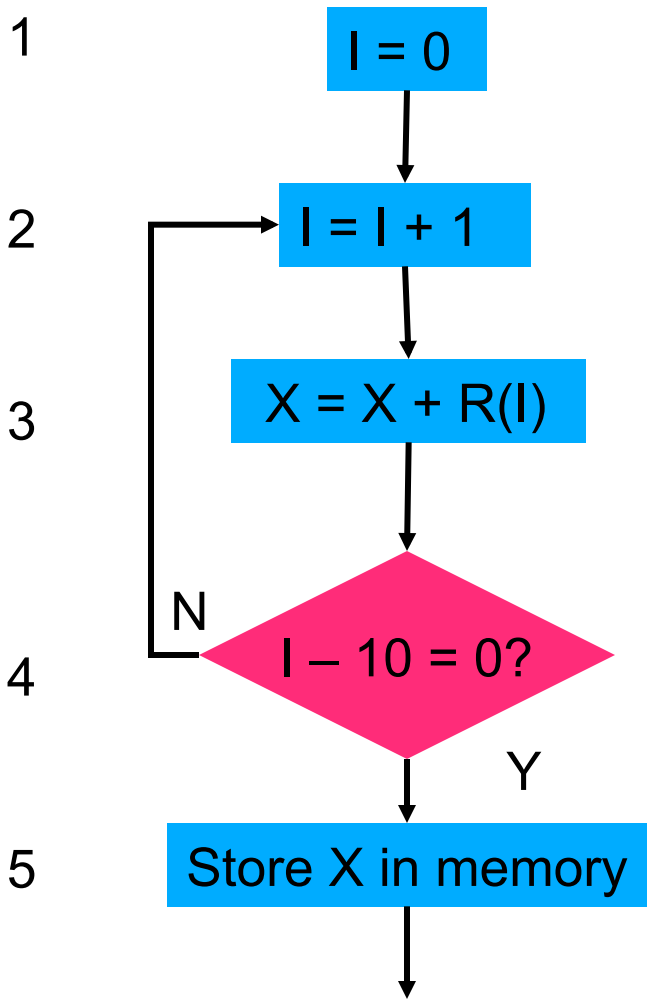
- Useful for program loops.
- A one-bit prediction scheme: a one-bit buffer carries a "history bit" that tells what happened on the last branch instruction
  - History bit = 1, branch was taken
  - History bit = 0, branch was not taken



Branch Prediction



Branch Prediction for a Loop



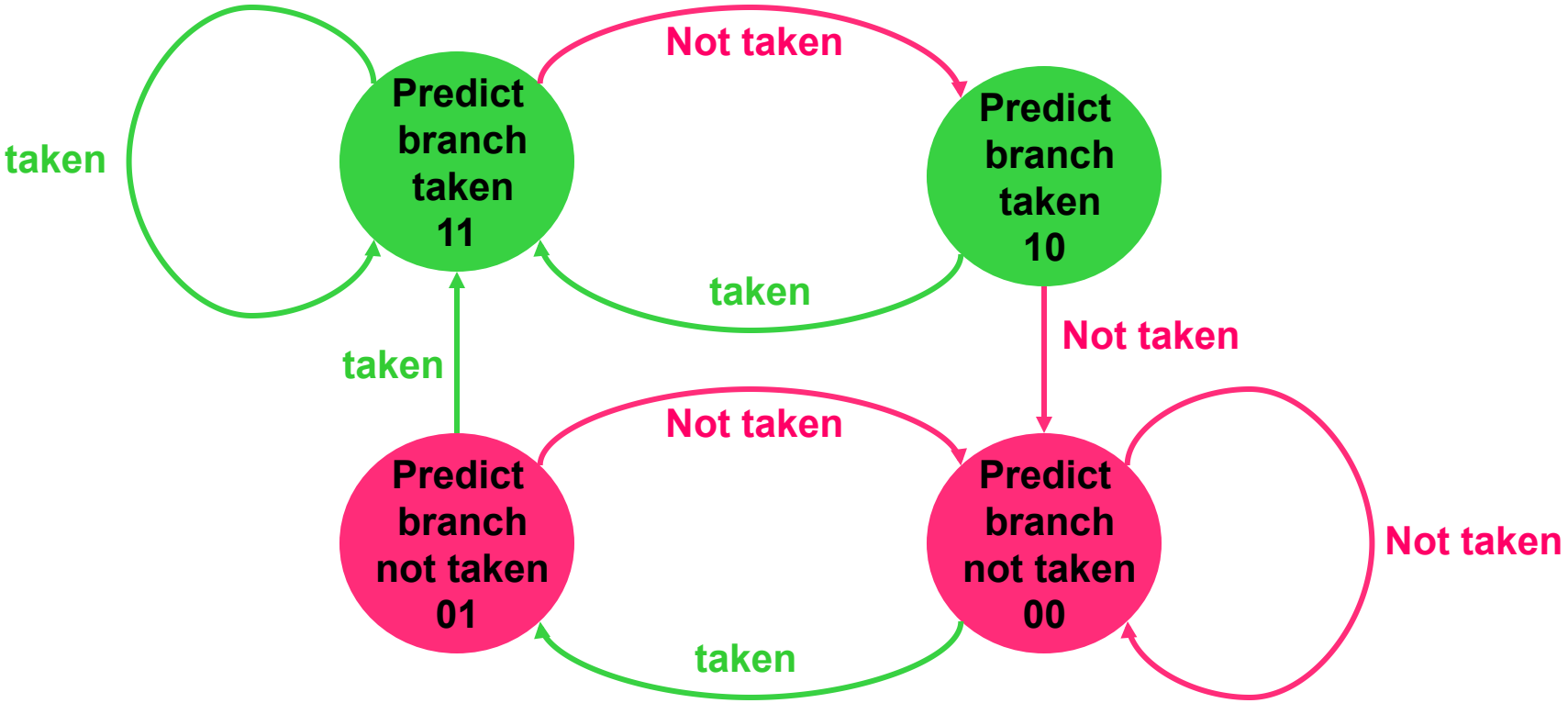
Execution of Instruction 4

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

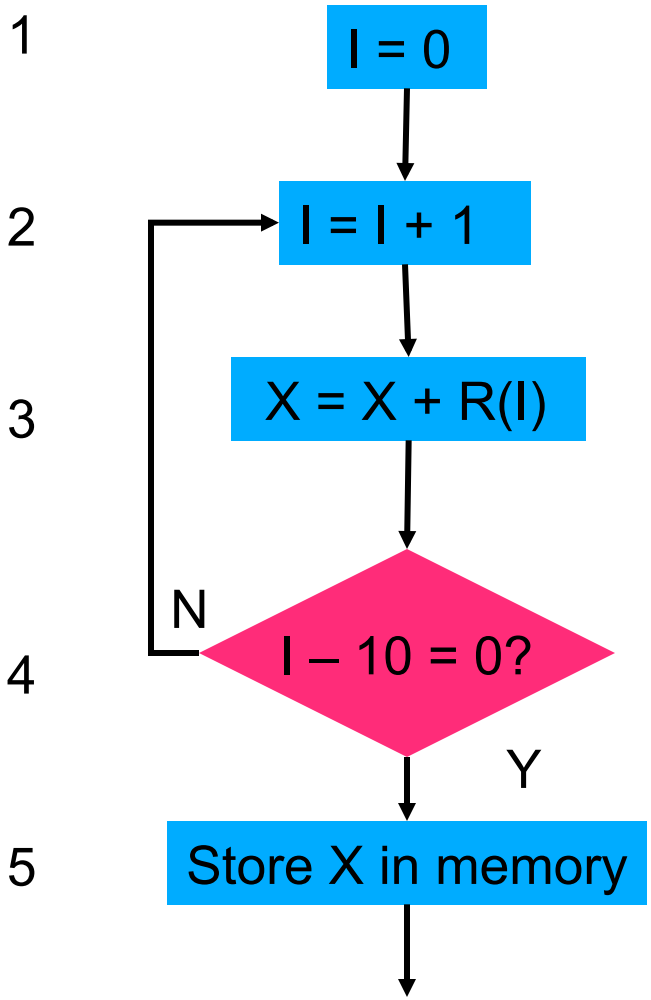
h.bit = 0 *branch not taken*, h.bit = 1 *branch taken*.

Two-Bit Prediction Buffer

- Can improve correct prediction statistics.



Branch Prediction for a Loop



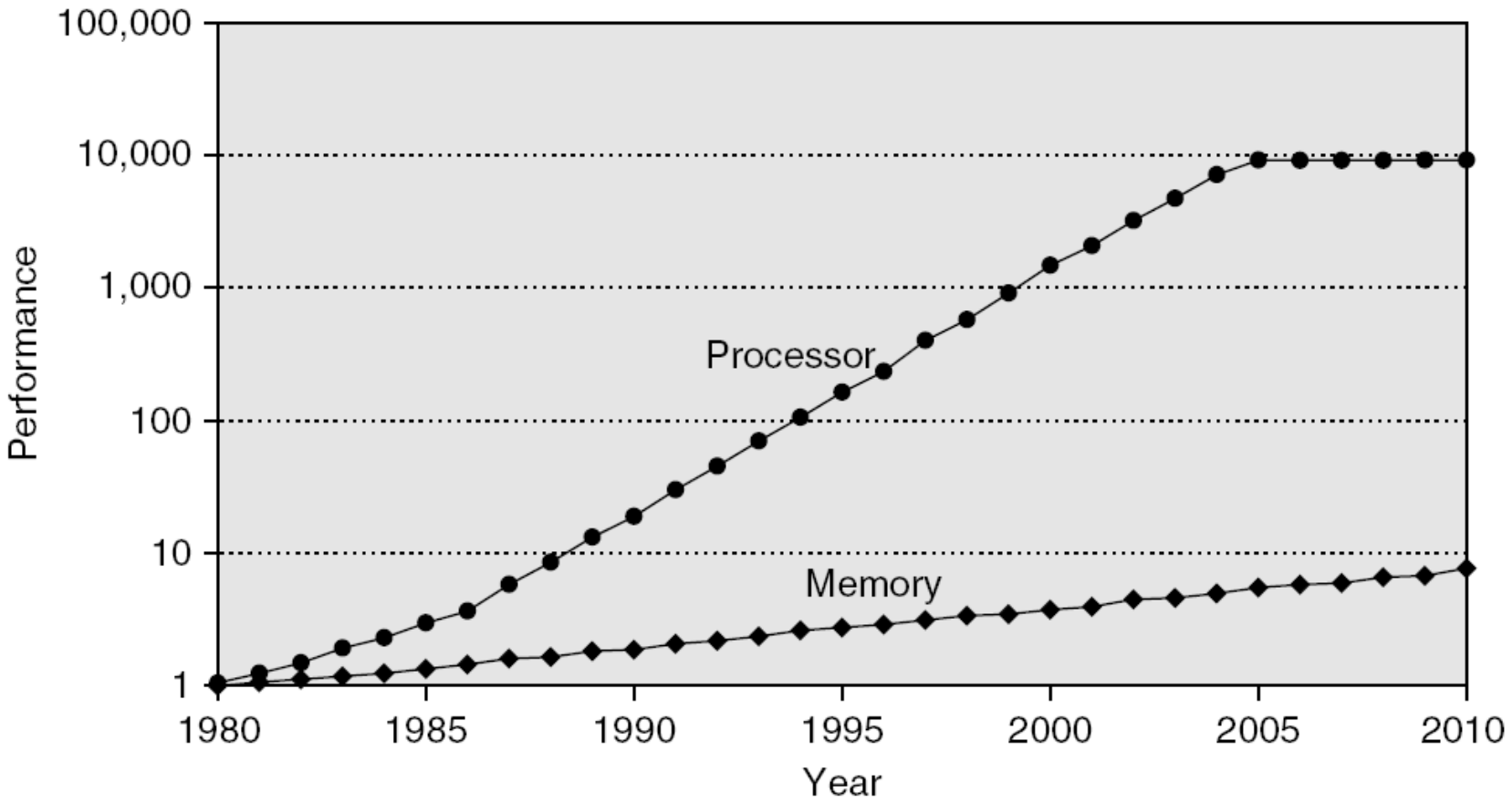
Execution of Instruction 4

Execu-tion seq.	Old Pred. Buf	Next instr.			New pred. Buf	Predi ction
		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

Summary: Hazards

- Structural hazards
  - Cause: resource conflict
  - Remedies: (i) hardware resources, (ii) stall (bubble)
- Data hazards
  - Cause: data unavailablity
  - Remedies: (i) forwarding, (ii) stall (bubble), (iii) code reordering
- Control hazards
  - Cause: out-of-sequence execution (branch or jump)
  - Remedies: (i) stall (bubble), (ii) branch prediction/pipeline flush, (iii) delayed branch/pipeline flush

Memory Performance Gap



Why Memory Hierarchy?

- Need lots of bandwidth

$$BW = \frac{1.0inst}{cycle} \times \left[\frac{1Ifetch}{inst} \times \frac{4B}{Ifetch} + \frac{0.4Dref}{inst} \times \frac{4B}{Dref} \right] \times \frac{1Gcycles}{\text{sec}}$$
$$= \frac{5.6GB}{\text{sec}}$$

- Need lots of storage
  - 64MB (minimum) to multiple TB
- Must be cheap per bit
  - (TB x anything) is a lot of money!
- These requirements seem incompatible

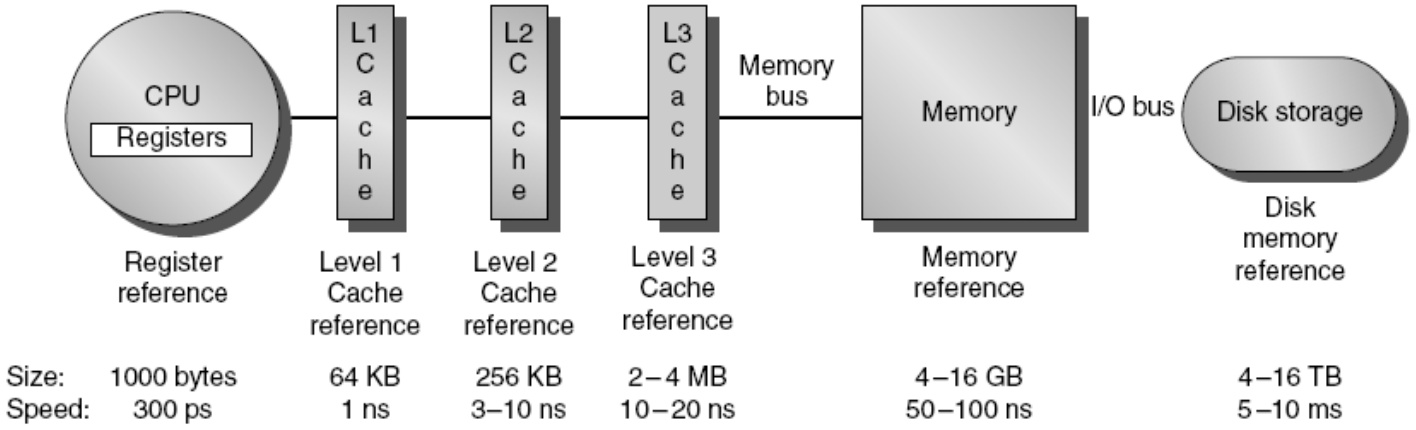
Memory Hierarchy Design

- Memory hierarchy design becomes more crucial with recent multi-core processors:
  - Aggregate peak bandwidth grows with # cores:
    - Intel Core i7 can generate two references per core per clock
    - Four cores and 3.2 GHz clock
      - 25.6 billion 64-bit data references/second +
      - 12.8 billion 128-bit instruction references
      - = 409.6 GB/s!
    - DRAM bandwidth is only 6% of this (25 GB/s)
    - Requires:
      - Multi-port, pipelined caches
      - Two levels of cache per core
      - Shared third-level cache on chip

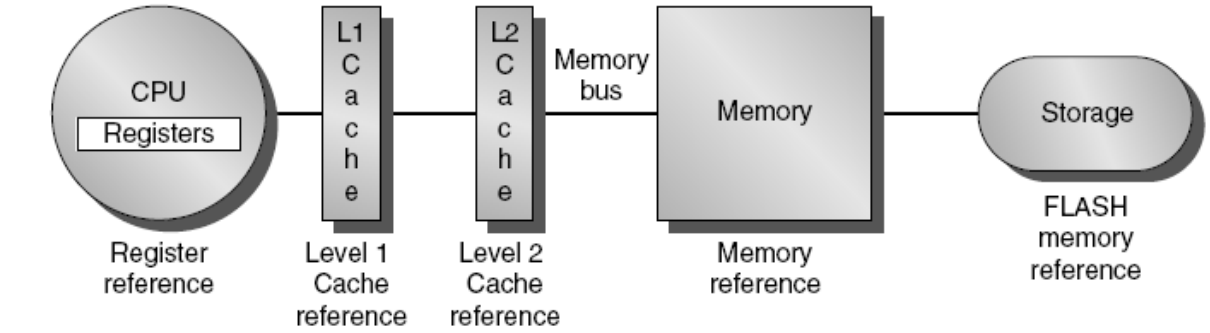
Why Memory Hierarchy?

- Fast and small memories
  - Enable quick access (fast cycle time)
  - Enable lots of bandwidth (1+ L/S/I-fetch/cycle)
- Slower larger memories
  - Capture larger share of memory
  - Still relatively fast
- Slow huge memories
  - Hold rarely-needed state
  - Needed for correctness
- All together: provide appearance of large, fast memory with cost of cheap, slow memory

Memory Hierarchy



(a) Memory hierarchy for server



(b) Memory hierarchy for a personal mobile device

Why Does a Hierarchy Work?

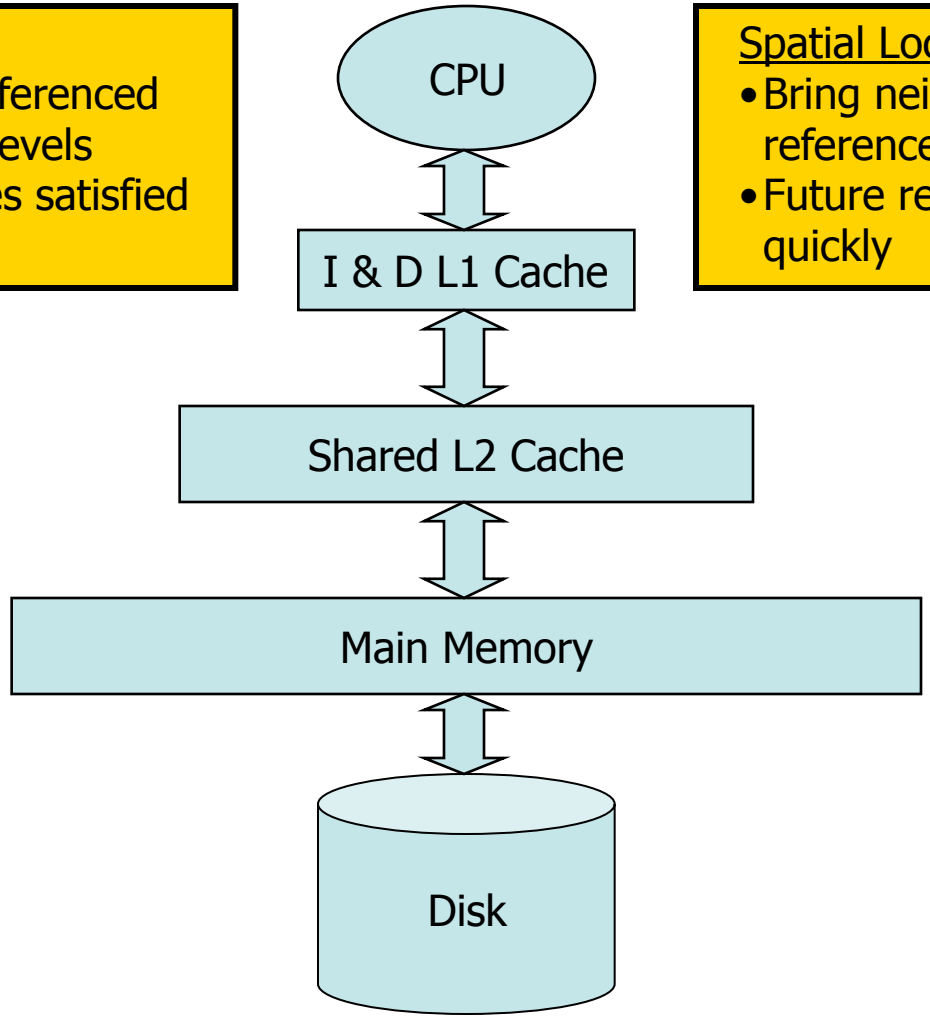
- Locality of reference
  - Temporal locality
    - Reference same memory location repeatedly
  - Spatial locality
    - Reference near neighbors around the same time
- Empirically observed
  - Significant!
  - Even small local storage (8KB) often satisfies >90% of references to multi-MB data set

Why Locality?

- Analogy:
  - Library (Disk)
  - Bookshelf (Main memory)
  - Stack of books on desk (off-chip cache)
  - Opened book on desk (on-chip cache)
- Likelihood of:
  - Referring to same book or chapter again?
    - Probability decays over time
    - Book moves to bottom of stack, then bookshelf, then library
  - Referring to chapter n+1 if looking at chapter n?

Memory Hierarchy

Temporal Locality
- Keep recently referenced items at higher levels
- Future references satisfied quickly



Spatial Locality
- Bring neighbors of recently referenced to higher levels
- Future references satisfied quickly

Thank You