ELEC 5200-001/6200-001 Computer Architecture and Design

Fall 2013

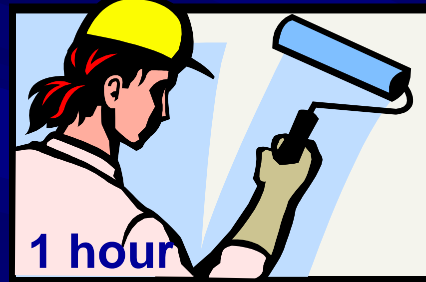
Pipelining (Chapter 4.5, 4.6)

Vishwani D. Agrawal & Victor P. Nelson
Department of Electrical and Computer Engineering
Auburn University, Auburn, AL 36849

ILP: Instruction Level Parallelism

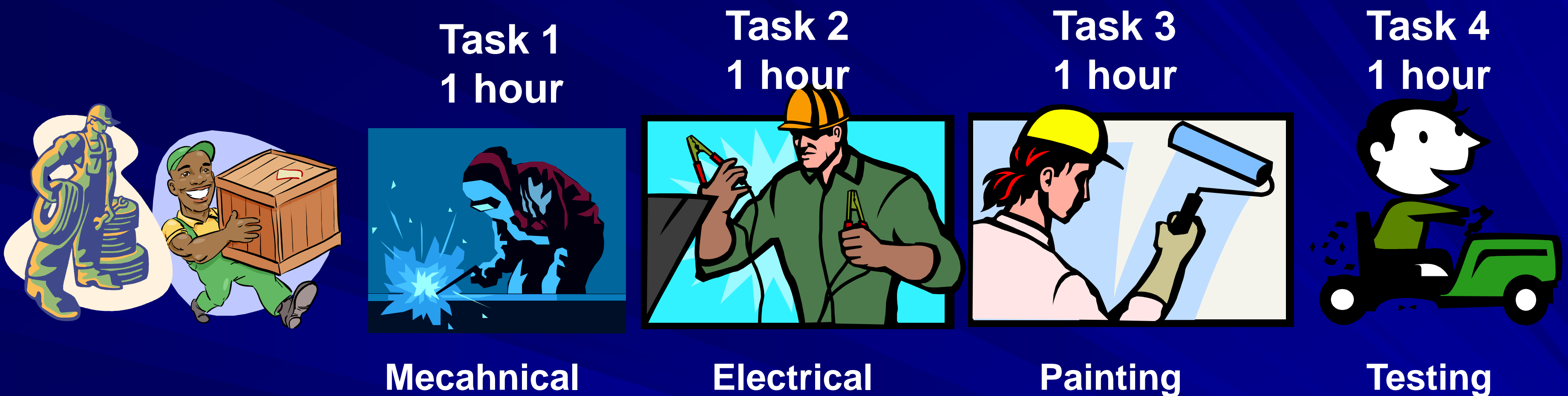
- Single-cycle and multi-cycle datapaths execute one instruction at a time.
- How can we get better performance?
- Answer: Execute multiple instructions at a time:
  - Pipelining – Enhance a multi-cycle datapath to fetch one instruction every cycle.
  - Parallelism – Fetch multiple instructions every cycle.

Automobile Team Assembly



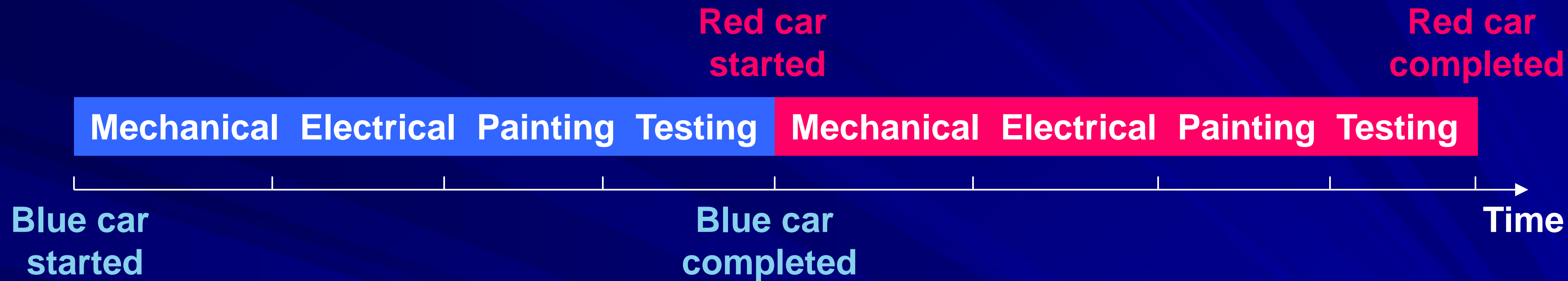
1 car assembled every four hours
6 cars per day
180 cars per month
2,040 cars per year

Automobile Assembly Line



First car assembled in 4 hours (pipeline latency)
thereafter, 1 car completed per hour
21 cars on first day, thereafter 24 cars per day
717 cars per month
8,637 cars per year
What gives 4X increase?

Throughput: Team Assembly

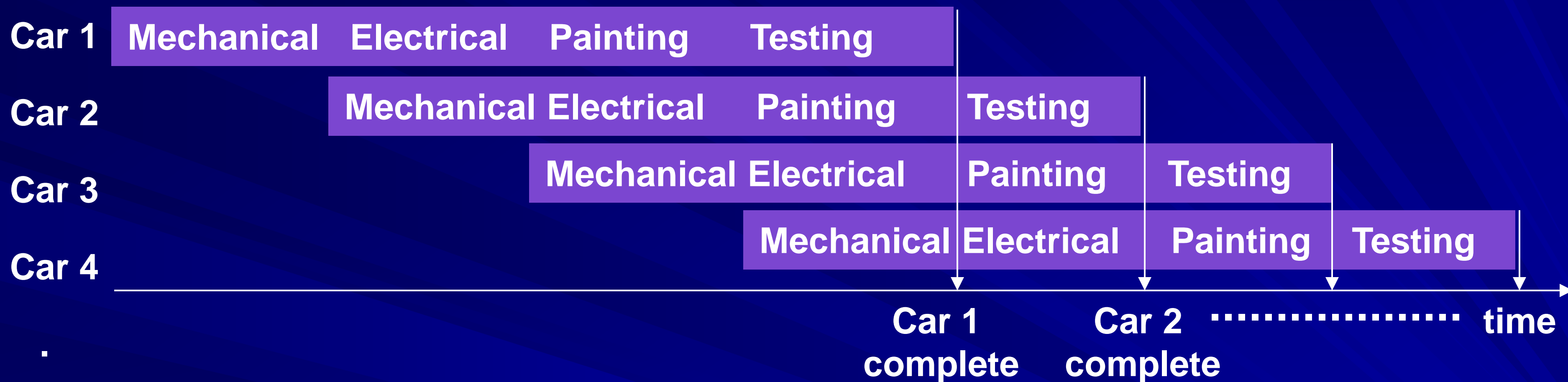


Time of assembling one car = *n* hours

where *n* is the number of nearly equal subtasks, each requiring 1 unit of time

Throughput = 1/*n* cars per unit time

Throughput: Assembly Line



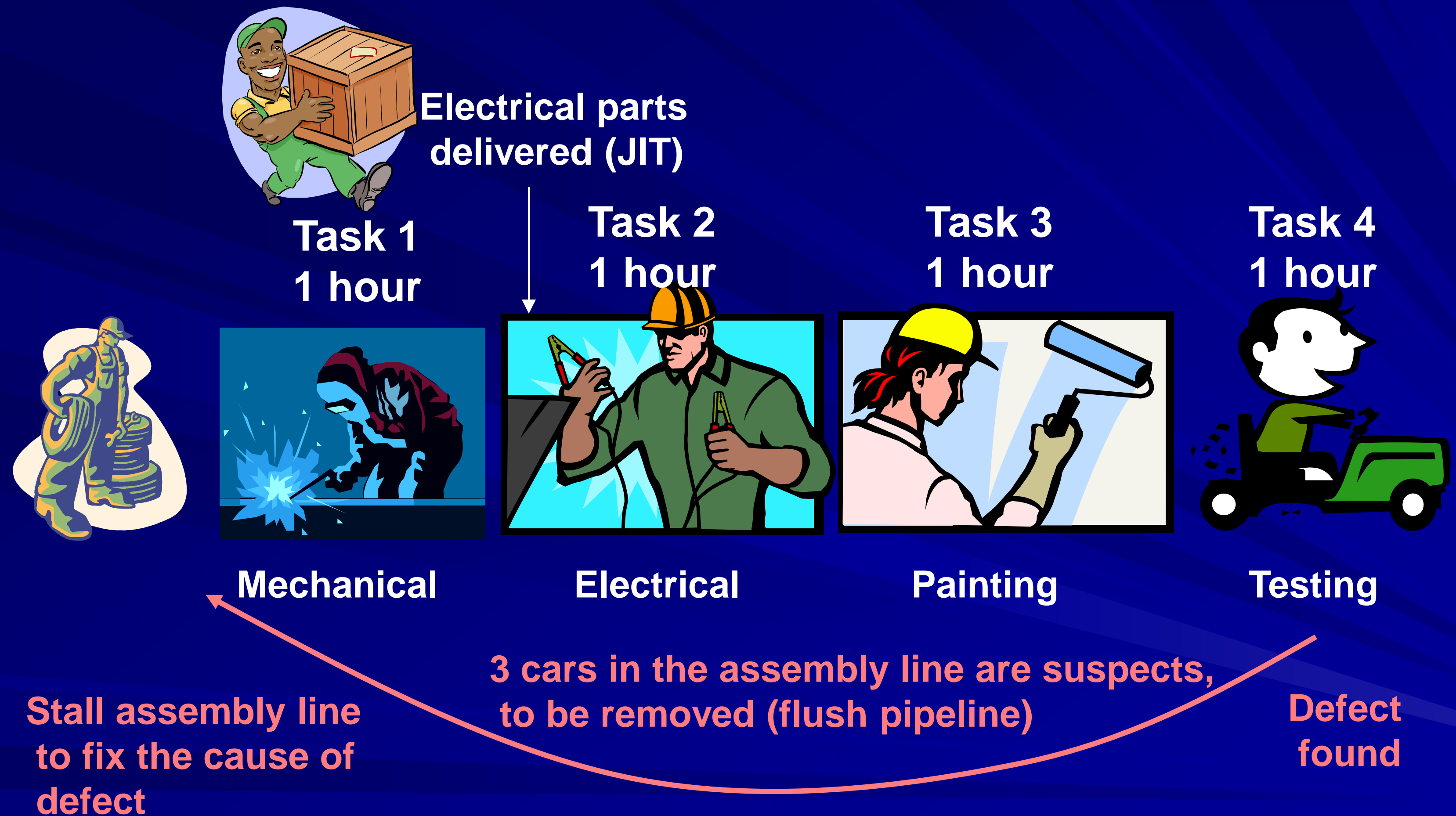
- Time to complete first car $= n$ time units (latency)

  Cars completed in time T $= T - n + 1$

  Throughput $= 1 - (n - 1)/T$ cars per unit time

$$\frac{\text{Throughput (assembly line)}}{\text{Throughput (team assembly)}} = \frac{1 - (n-1)/T}{1/n} = n - \frac{n(n-1)}{T} \rightarrow n \quad \text{as } T \rightarrow \infty$$

Some Features of Assembly Line



Pros and Cons

- Advantages:
  - Efficient use of labor.
  - Specialists can do better job.
  - Just in time (JIT) methodology eliminates warehouse cost.
- Disadvantages:
  - Penalty of defect latency.
  - Lack of flexibility in production.
  - Assembly line work is monotonous and boring.
  - http://www.youtube.com/watch?v=c8LxscnmdNY&feature=related
  - http://www.metacafe.com/watch/752497/chaplin_with_a_spanner_set_modern_times/
  - http://www.metacafe.com/watch/762944/crazy_chaplin_screwing_up_everything_modern_times

Pipelining in a Computer

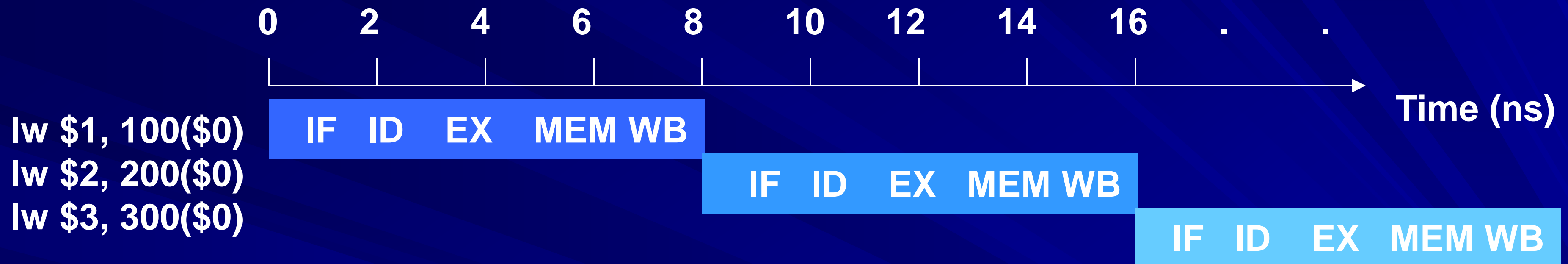
- Divide datapath into nearly equal tasks, to be performed serially and requiring non-overlapping resources.
- Insert registers at task boundaries in the datapath; registers pass the output data from one task as input data to the next task.
- Synchronize tasks with a clock having a cycle time that just exceeds the time required by the longest task.
- Break each instruction down into the set of tasks so that instructions can be executed in a staggered fashion.

Pipelining a Single-Cycle Datapath

Instruction class	Instr. fetch (IF)	Instr. Decode (also reg. file read) (ID)	Execution (ALU Operation) (EX)	Data access (MEM)	Write Back (Reg. file write) (WB)	Total time
lw	2ns	1ns	2ns	2ns	1ns	8ns
sw	2ns	1ns	2ns	2ns		8ns
R-format add, sub, and, or, slt	2ns	1ns	2ns		1ns	8ns
B-format, beq	2ns	1ns	2ns			8ns

No operation on data; idle times equalize instruction lengths.

Execution Time: Single-Cycle



Clock cycle time = 8 ns

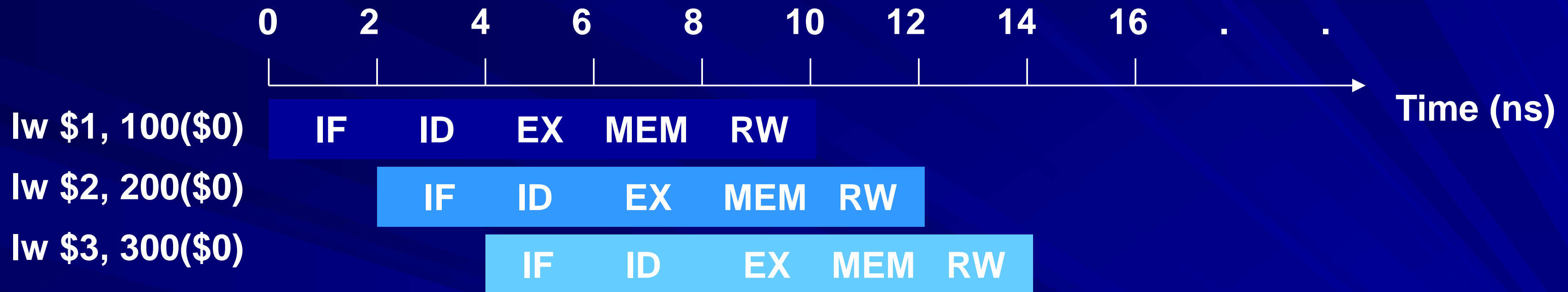
Total time for executing three lw instructions = 24 ns

Pipelined Datapath

Instruction class	Instr. fetch (IF)	Instr. Decode (also reg. file read) (ID)	Execution (ALU Operation) (EX)	Data access (MEM)	Write Back (Reg. file write) (WB)	Total time
lw	2ns	~~1ns~~ 2ns	2ns	2ns	~~1ns~~ 2ns	10ns
sw	2ns	~~1ns~~ 2ns	2ns	2ns	~~1ns~~ 2ns	10ns
R-format: add, sub, and, or, slt	2ns	~~1ns~~ 2ns	2ns	2ns	~~1ns~~ 2ns	10ns
B-format: beq	2ns	~~1ns~~ 2ns	2ns	2ns	~~1ns~~ 2ns	10ns

No operation on data; idle time inserted to equalize instruction lengths.

Execution Time: Pipeline



Clock cycle time = 2 ns, *four times faster than single-cycle clock*

Total time for executing three lw instructions = 14 ns

Performance ratio $= \dfrac{\text{Single-cycle time}}{\text{Pipeline time}} = \dfrac{24}{14} = 1.7$

Pipeline Performance

Clock cycle time = 2 ns

1,003 *lw instructions:*

Total time for executing 1,003 lw instructions = 2,014 ns

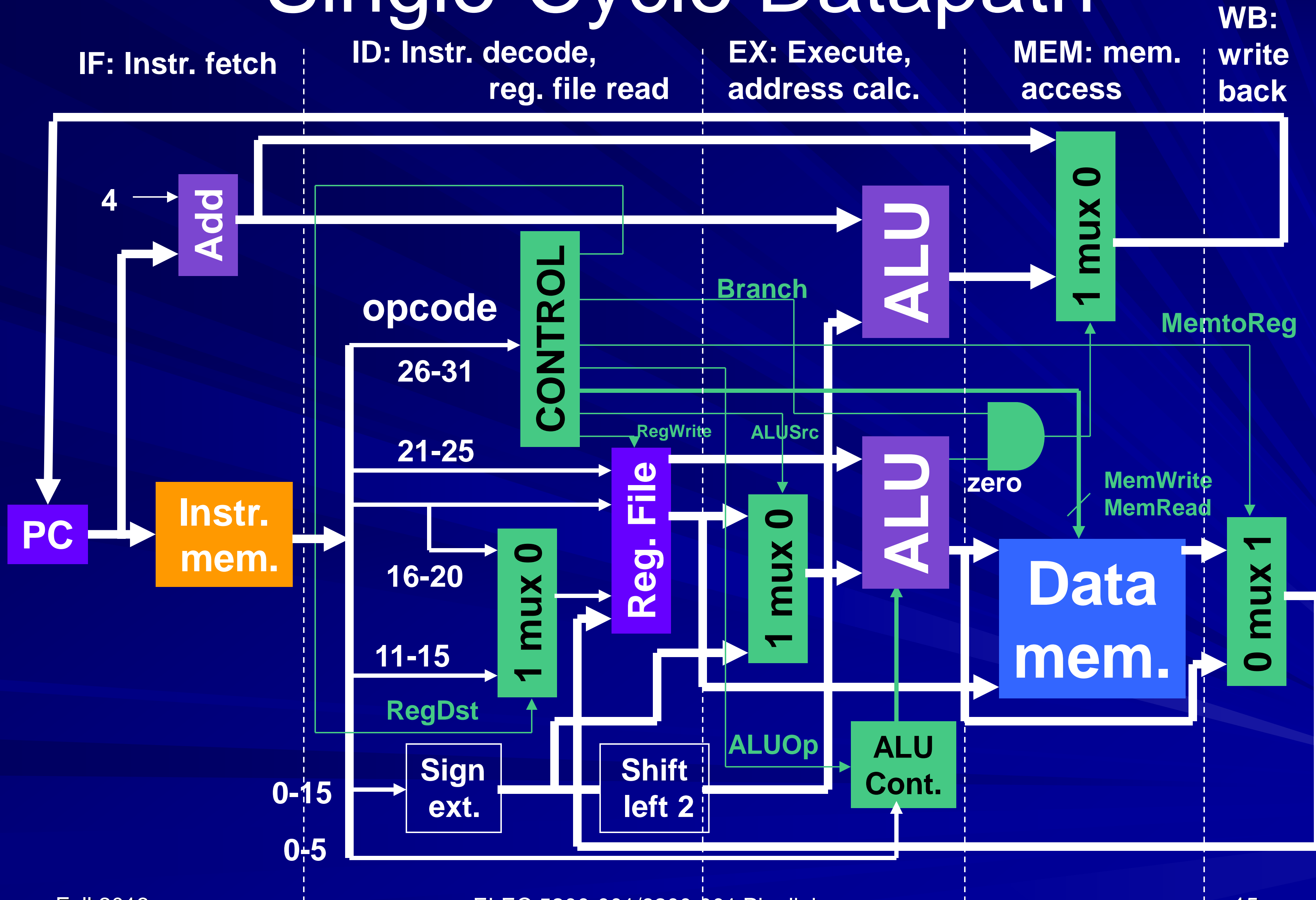
$$\text{Performance ratio} = \frac{\text{Single-cycle time}}{\text{Pipeline time}} = \frac{8{,}024}{2{,}014} = 3.98$$

10,003 *lw instructions:*

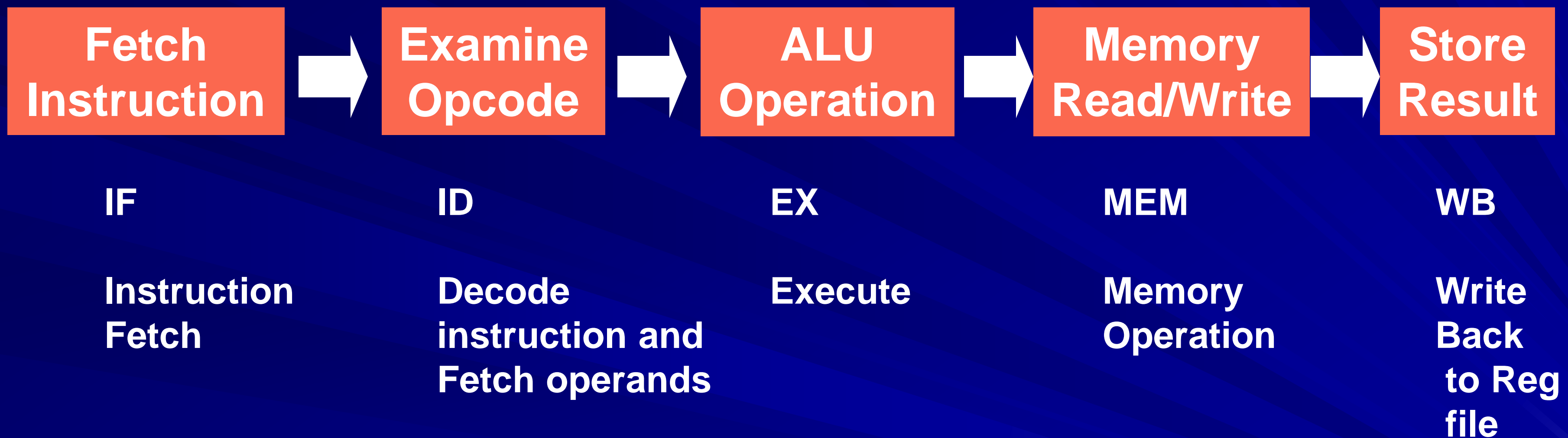
Performance ratio = 80,024 / 20,014 = 3.998 → Clock cycle ratio (4)

Pipeline performance approaches clock-cycle ratio for long programs.

Single-Cycle Datapath

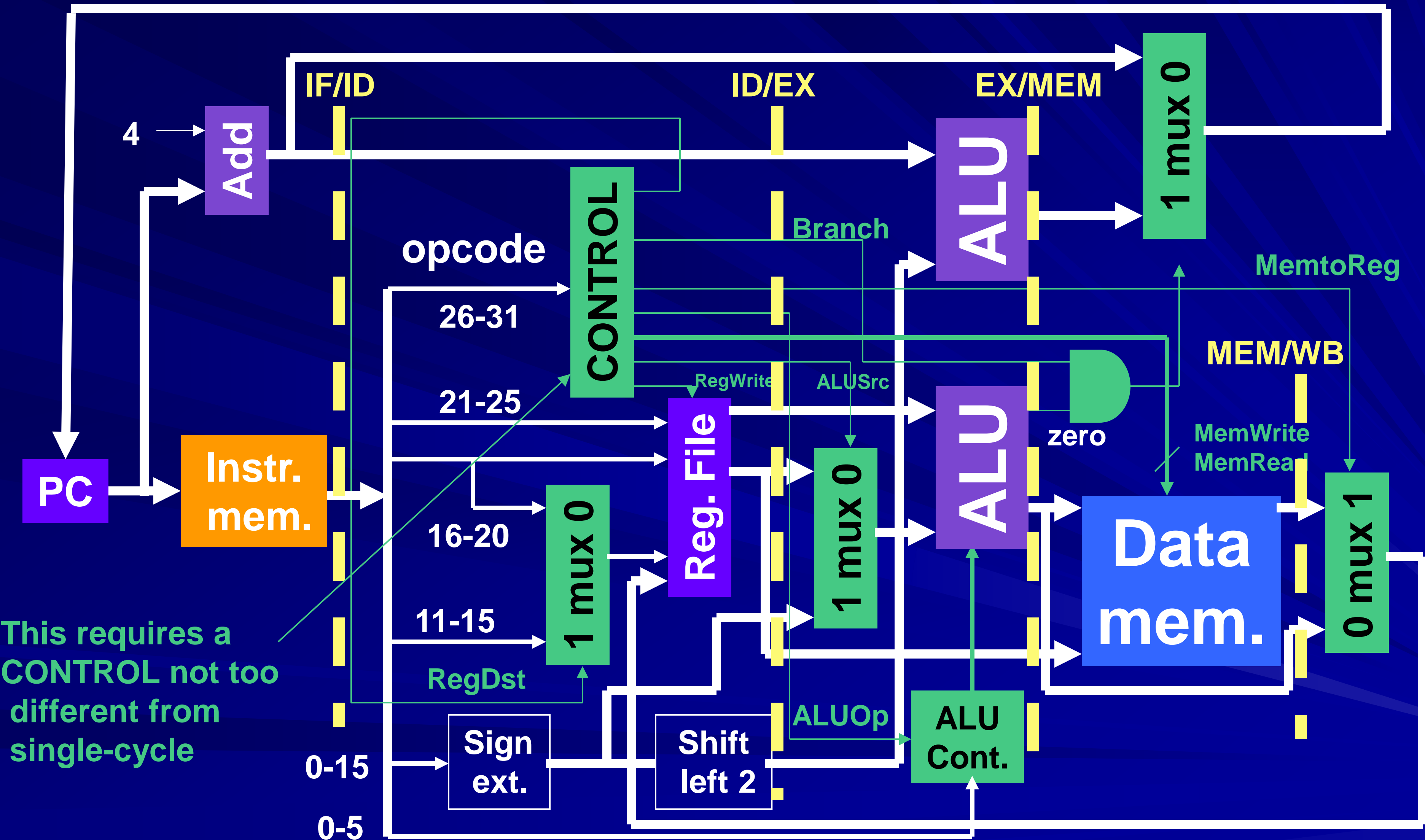


Pipelining of RISC Instructions (From Lecture 3, Slide 6)



Although an instruction takes five clock cycles, one instruction is completed every cycle.

Pipeline Registers

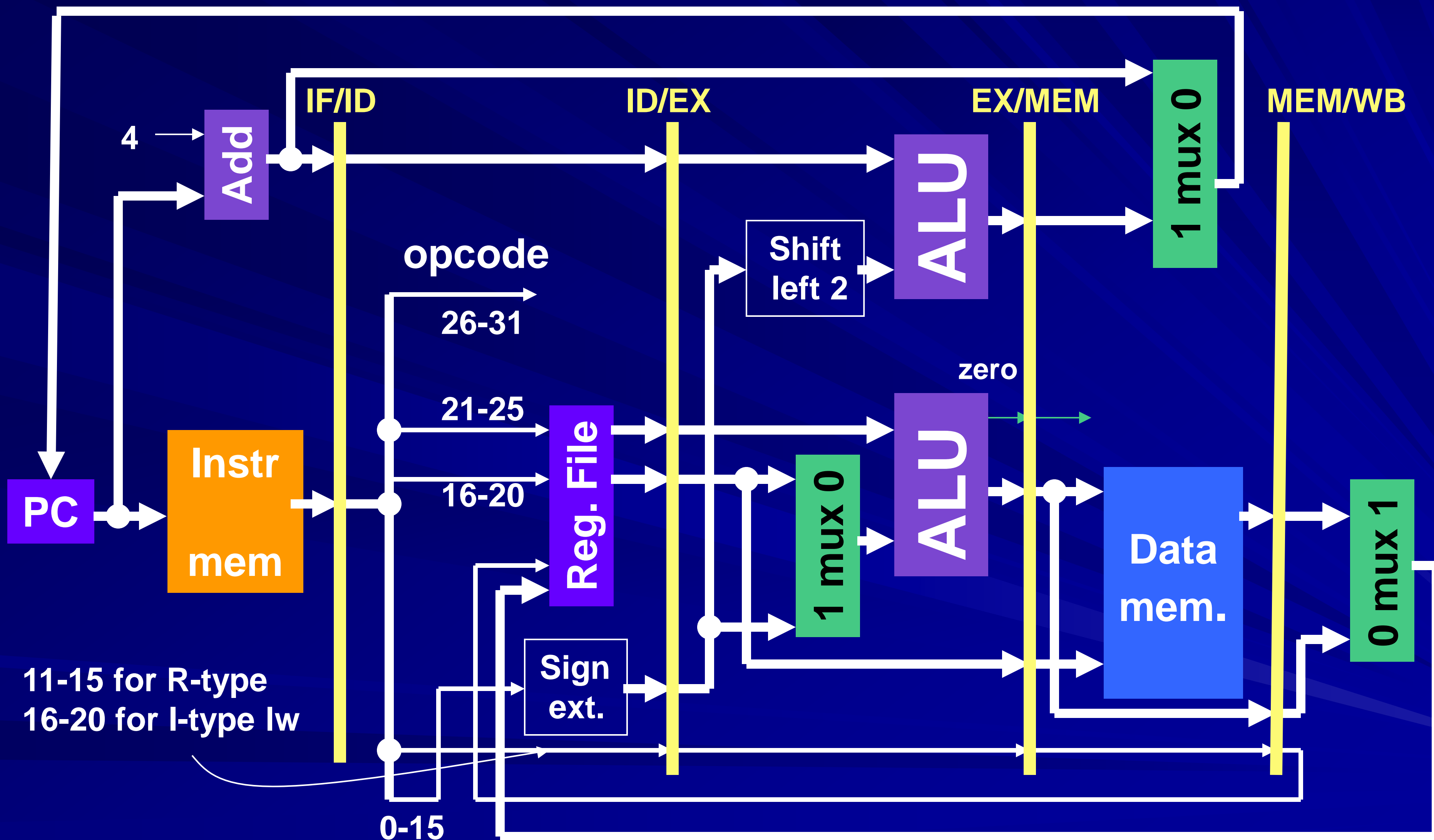


Pipeline Register Functions

- Four pipeline registers are added:

Register name	Data held
IF/ID	PC+4, Instruction word (IW)
ID/EX	PC+4, R1, R2, IW(0-15) sign ext., IW(11-15)
EX/MEM	PC+4, zero, ALUResult, R2, IW(11-15) or IW(16-20)
MEM/WB	M[ALUResult], ALUResult, IW(11-15) or IW(16-20)

Pipelined Datapath



Five-Cycle Pipeline

Add Instruction

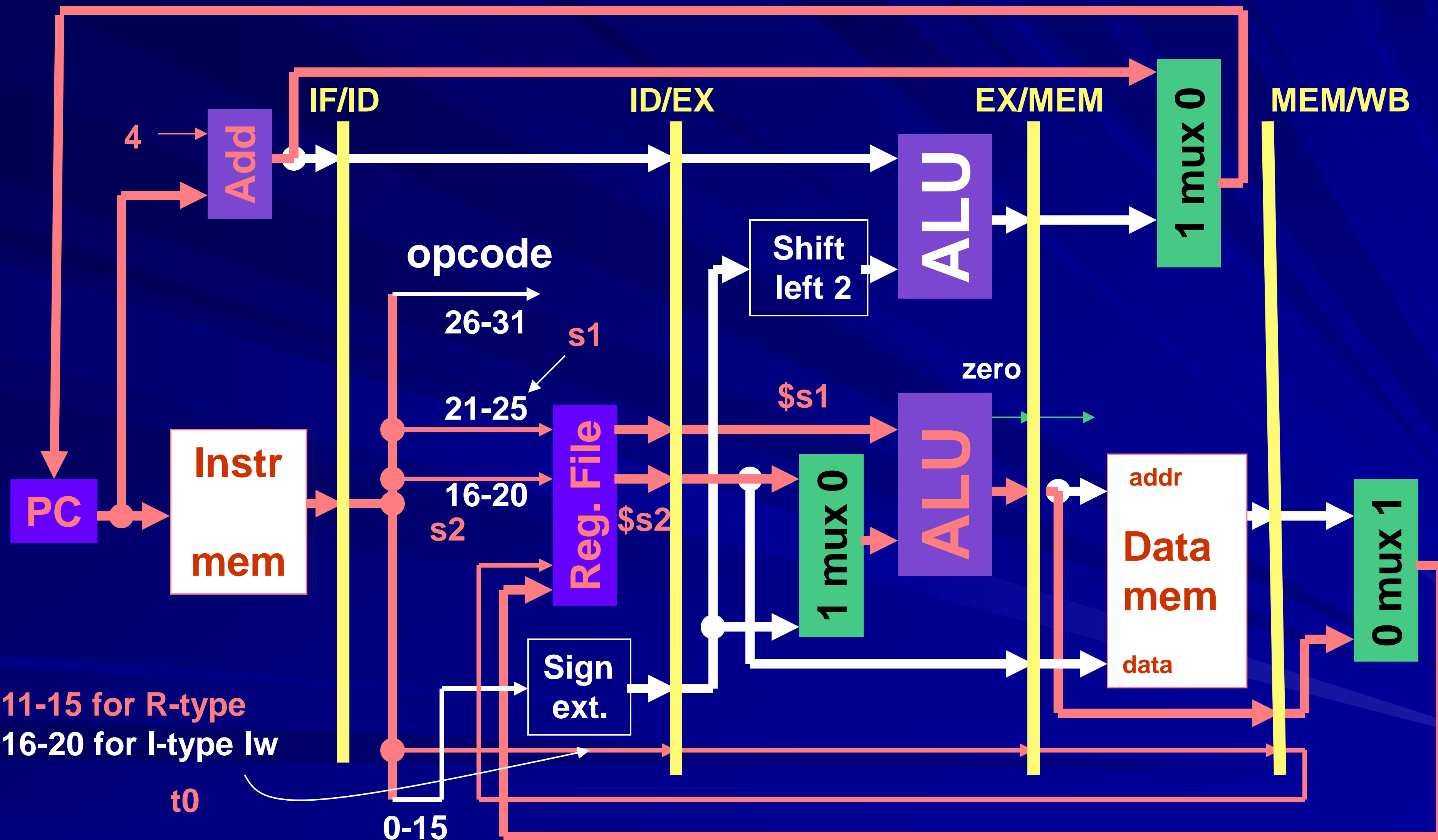
- add $t0, $s1, $s2

  Machine instruction word

  000000 10001 10010 01000 00000 100000

  opcode $s1 $s2 $t0 function

Pipelined Datapath Executing add

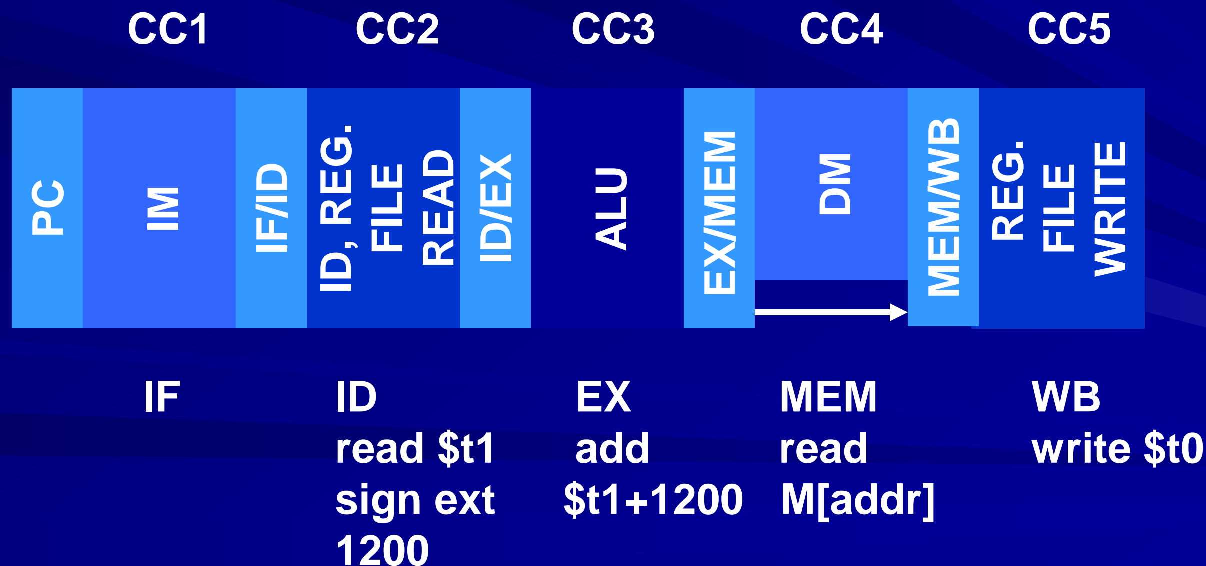


Load Instruction

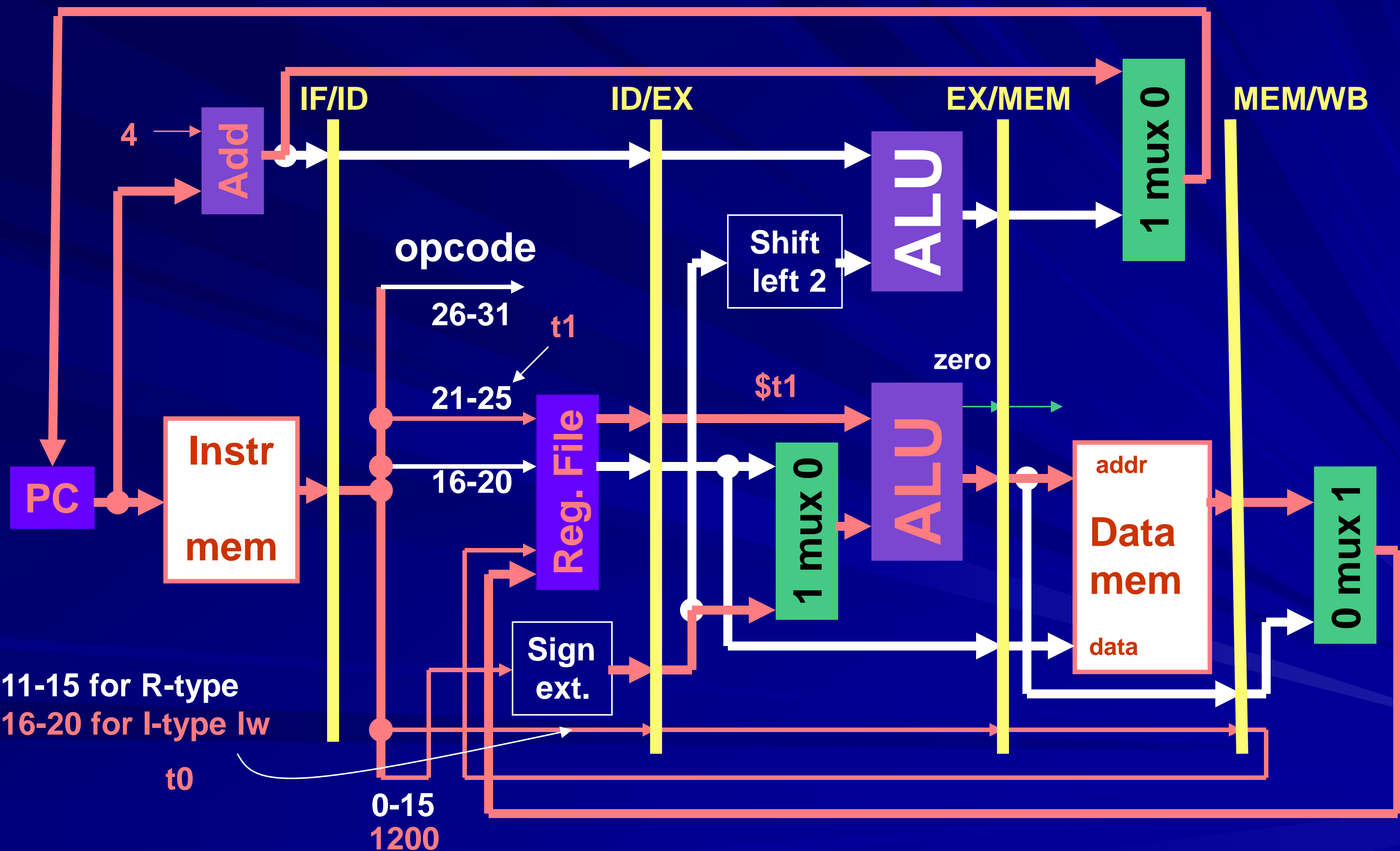
- lw $t0, 1200 ($t1)

100011 01001 01000 0000 0100 1000 0000

opcode $t1 $t0 1200



Pipelined Datapath Executing lw

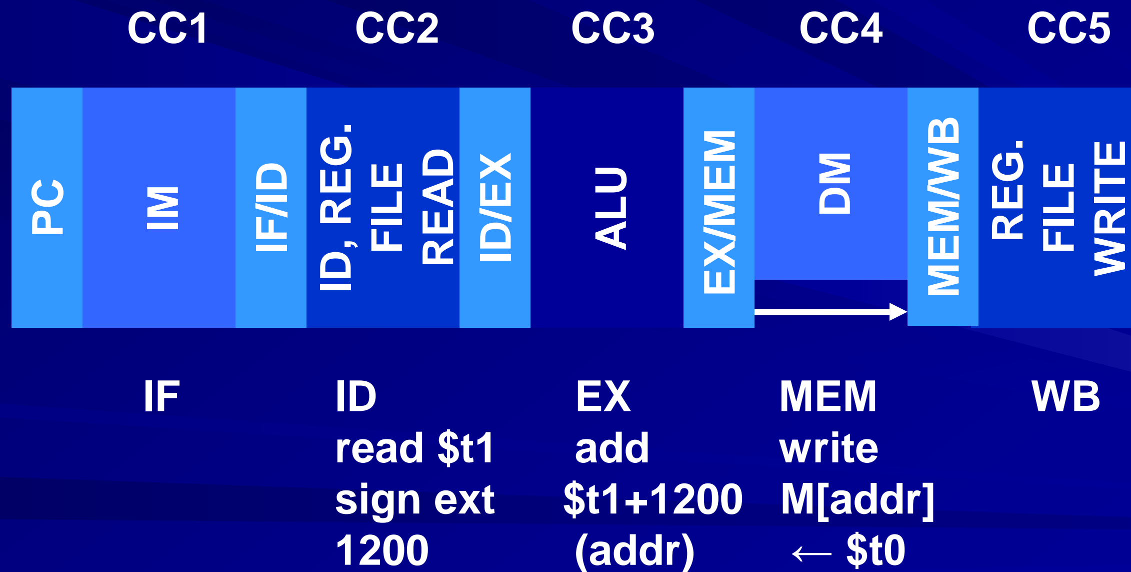


Store Instruction

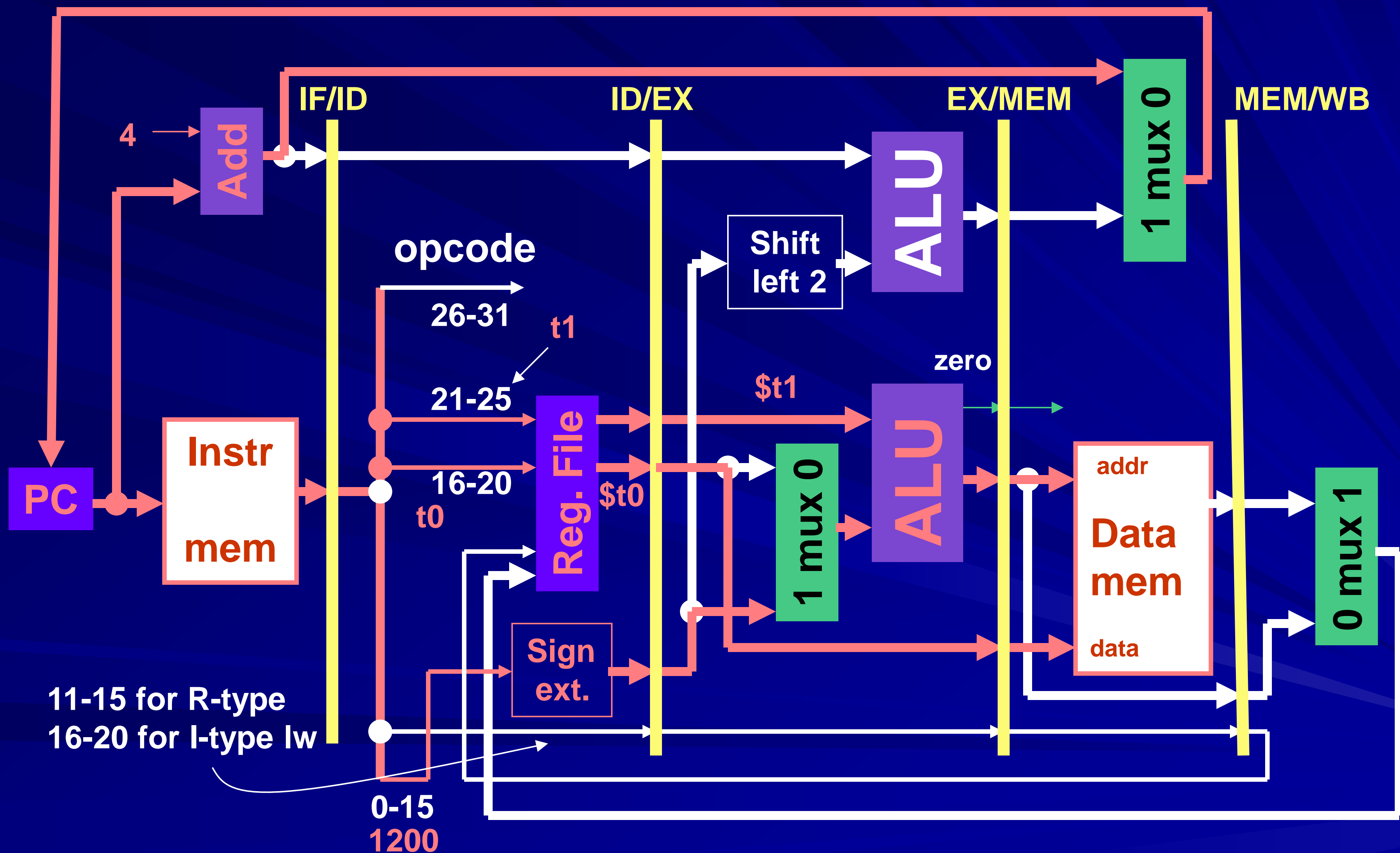
- sw $t0, 1200 ($t1)

101011 01001 01000 0000 0100 1000 0000

opcode $t1 $t0 1200



Pipelined Datapath Executing sw

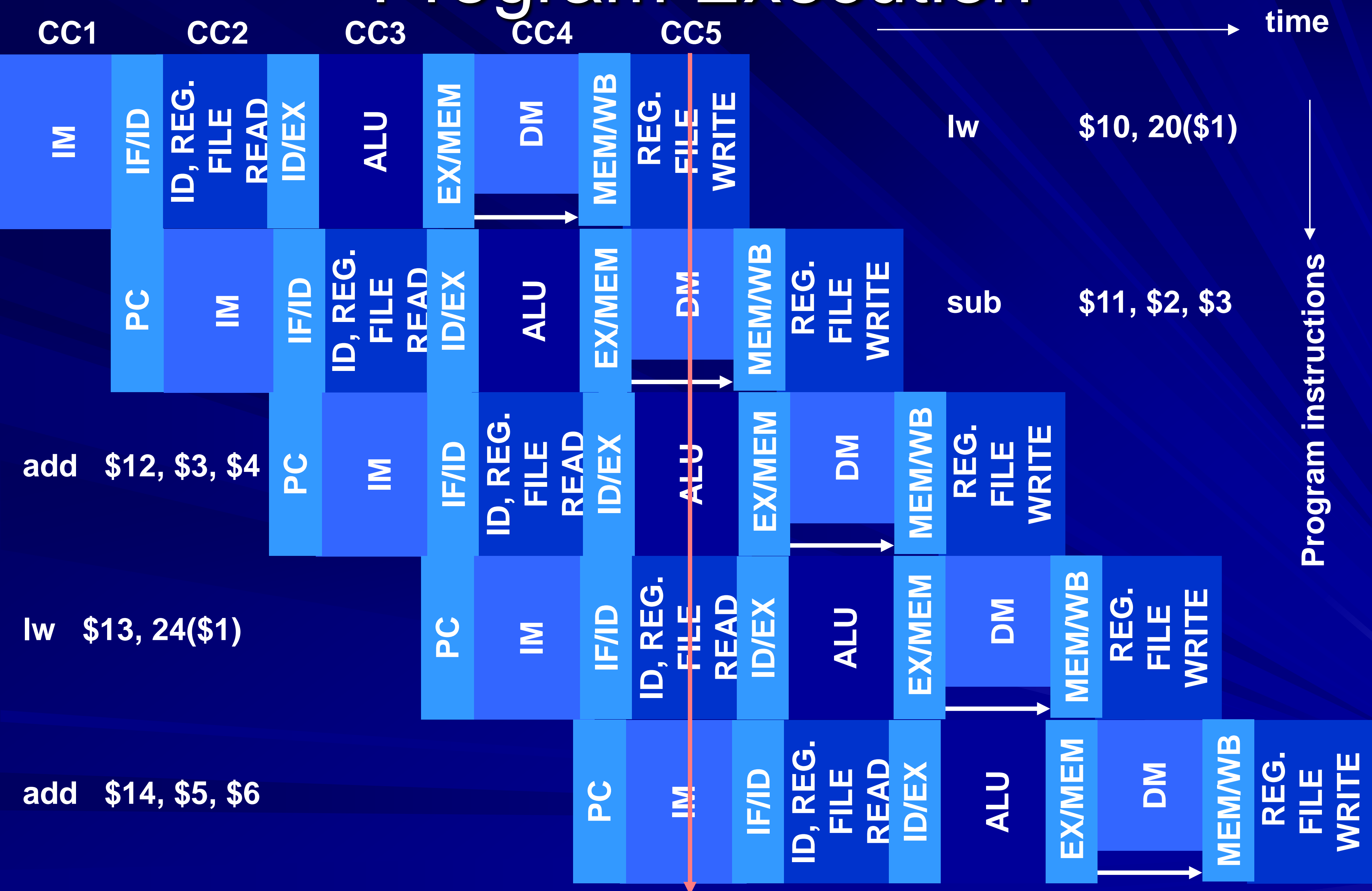


Executing a Program

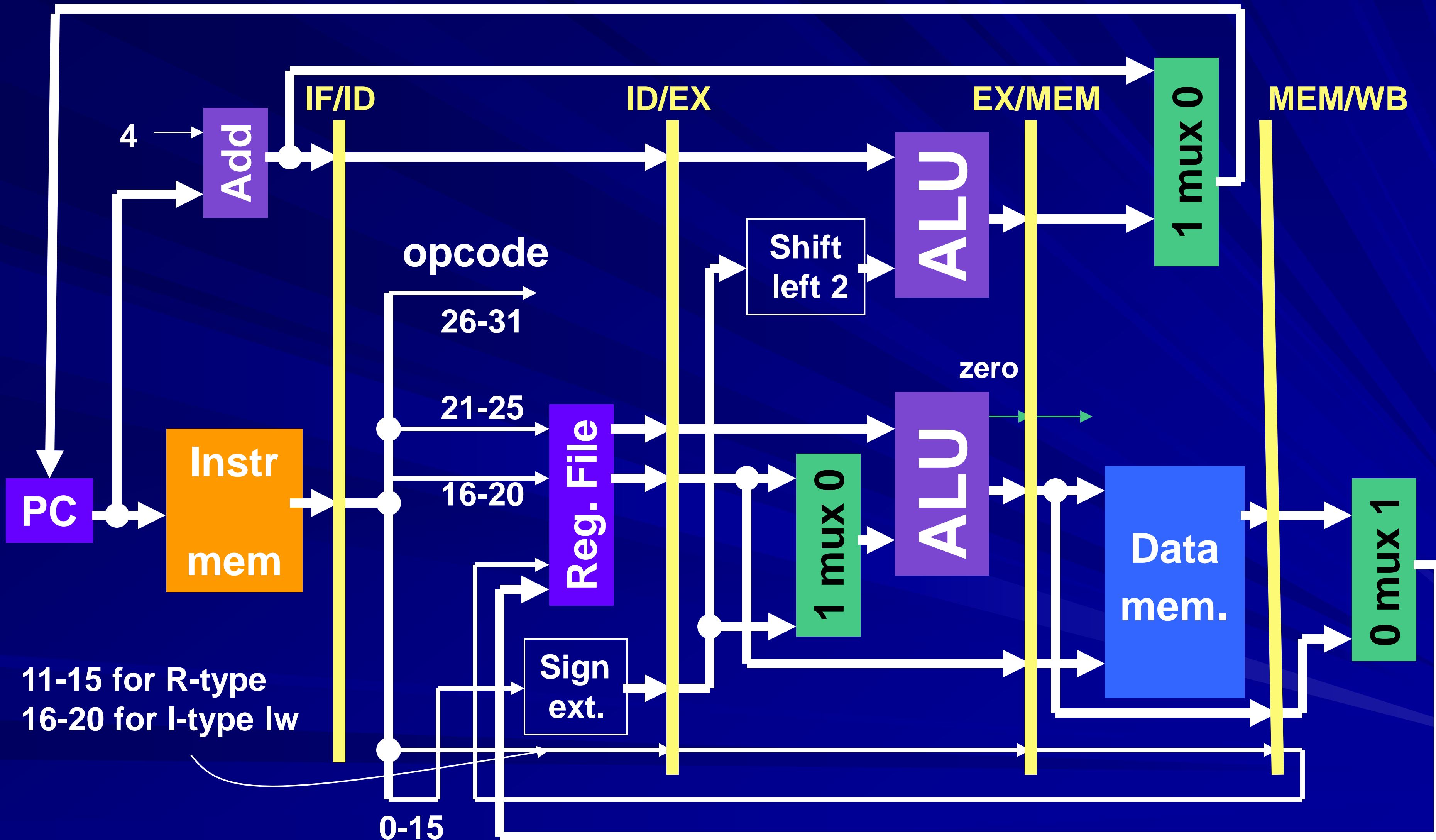
Consider a five-instruction segment:

```
lw    $10, 20($1)
sub   $11, $2, $3
add   $12, $3, $4
lw    $13, 24($1)
add   $14, $5, $6
```

Program Execution



CC5



Advantages of Pipeline

- After the fifth cycle (CC5), one instruction is completed each cycle; CPI ≈ 1, neglecting the initial pipeline latency of 5 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.
- For multicycle datapath, CPI = 3. ….
- So, pipelined execution is faster, but . . .

Science is always wrong. It never solves a problem without creating ten more.

George Bernard Shaw

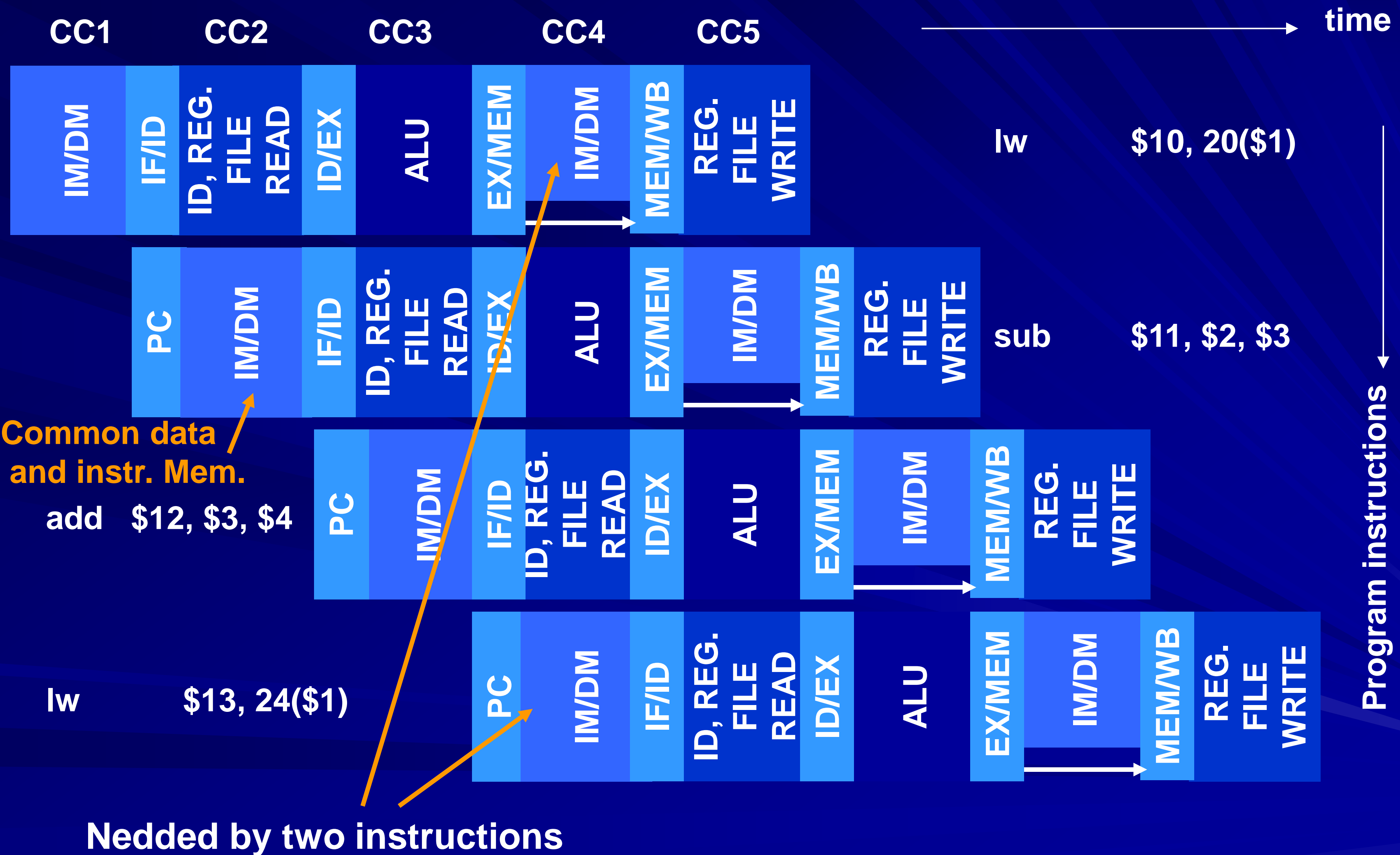
Pipeline Hazards

- 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, being 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

	CC1	CC2	CC3	CC4	CC5	
PC	IM	IF/ID — ID, REG. FILE READ	ID/EX — ALU	EX/MEM — DM	MEM/WB — REG. FILE WRITE	add $s0, $t0, $t1
	PC	IM	IF/ID — ID, REG. FILE READ	ID/EX — ALU	EX/MEM — DM	MEM/WB — REG. FILE WRITE — sub $t2, $s0, $t3

time →

Program instructions ↓

Write s0 in CC5

Read s0 and t3 in CC3

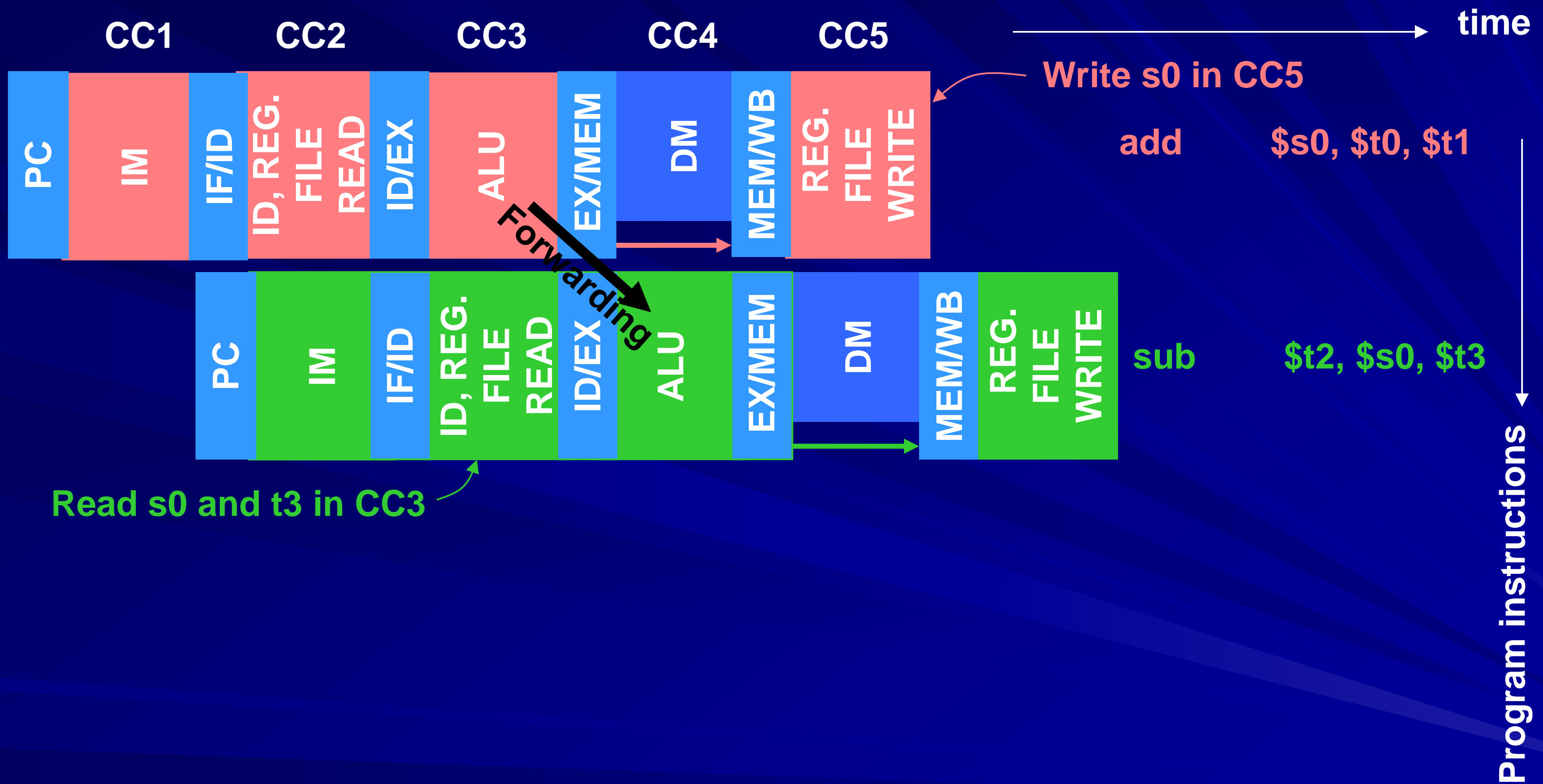
We need to read s0 from reg file in cycle 3
But s0 will not be written in reg file until cycle 5

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

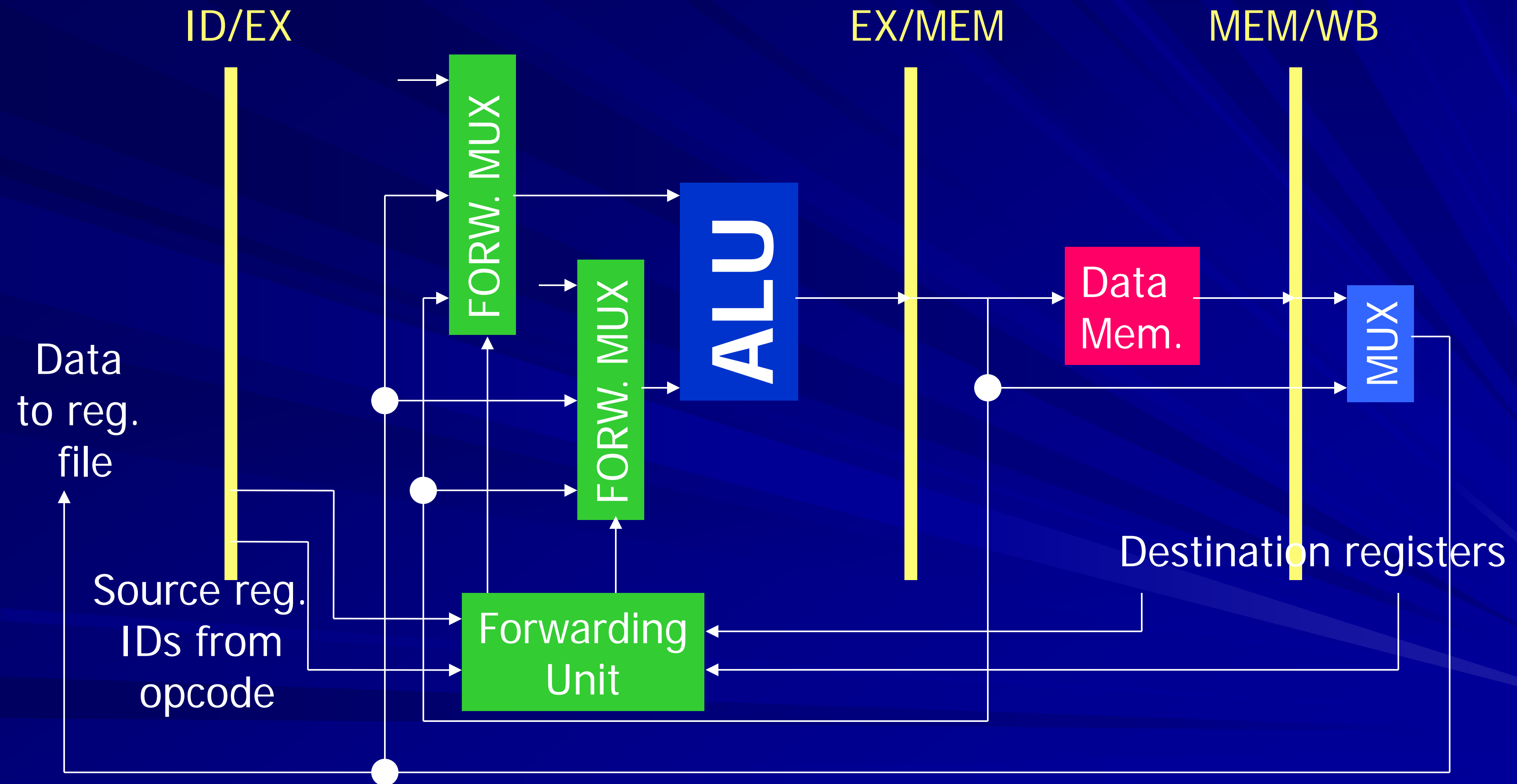
Forwarding or Bypassing

- 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.

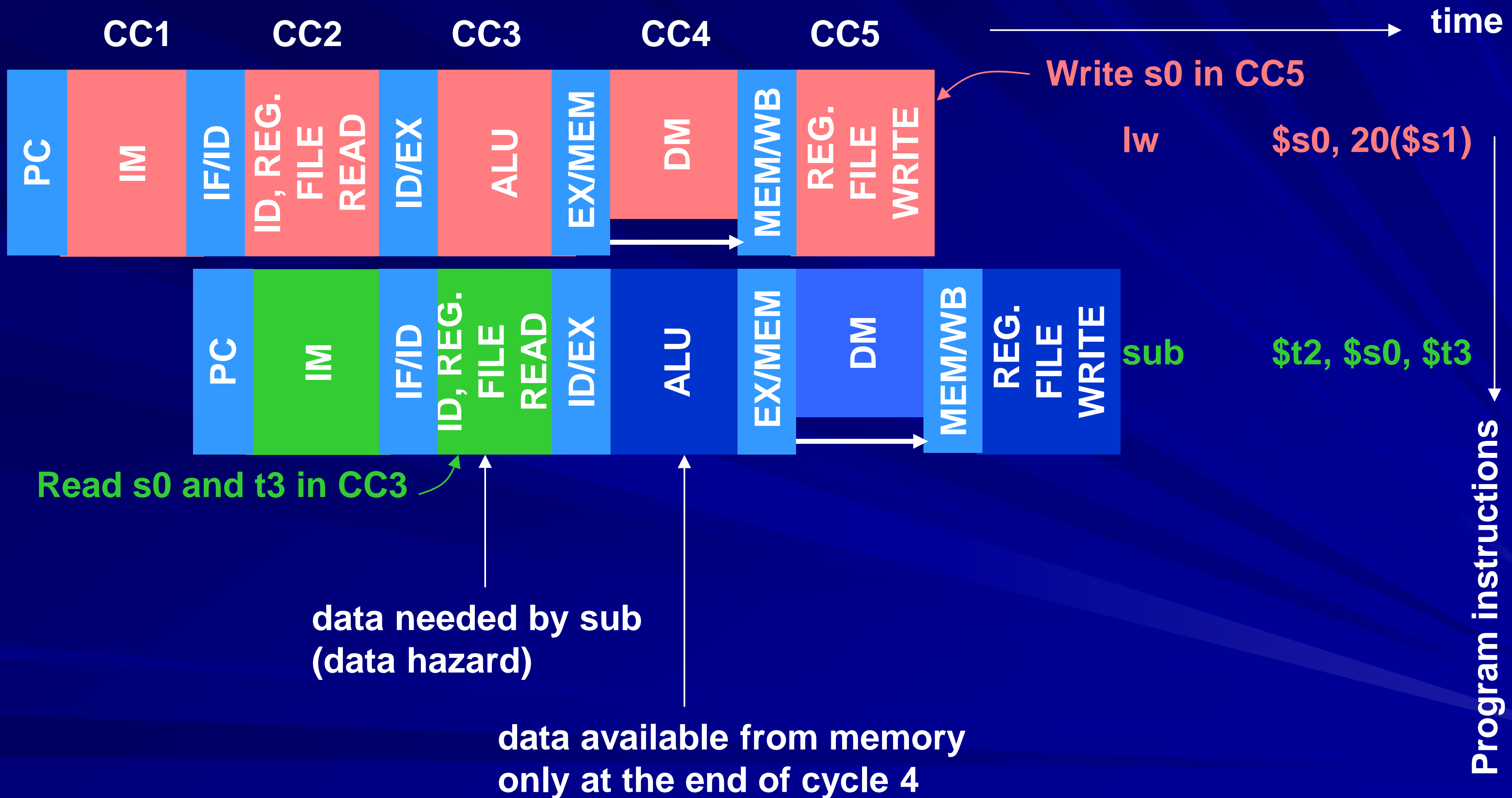
Forwarding for Data Hazard



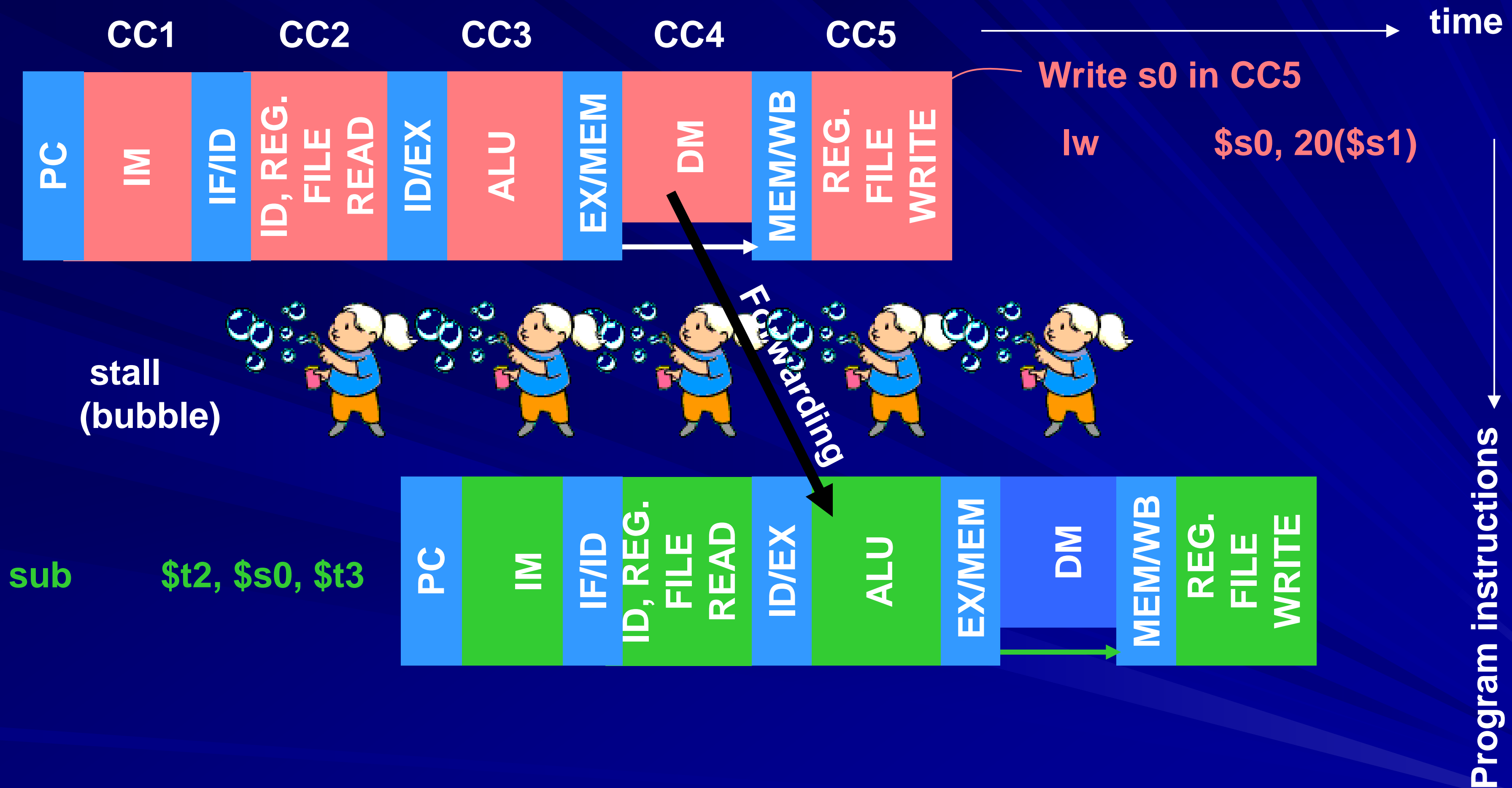
Forwarding Unit Hardware



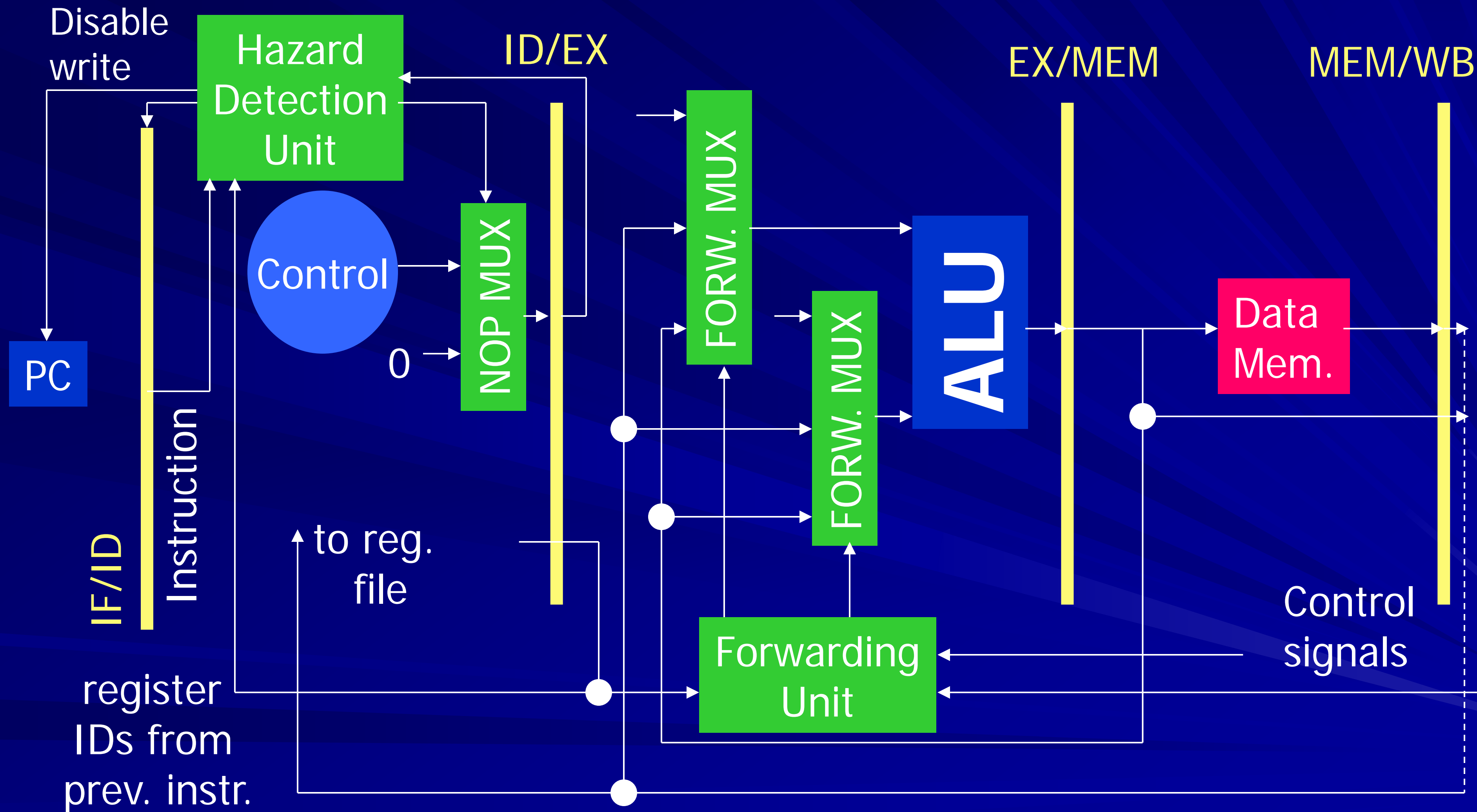
Forwarding Alone May Not Work



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.

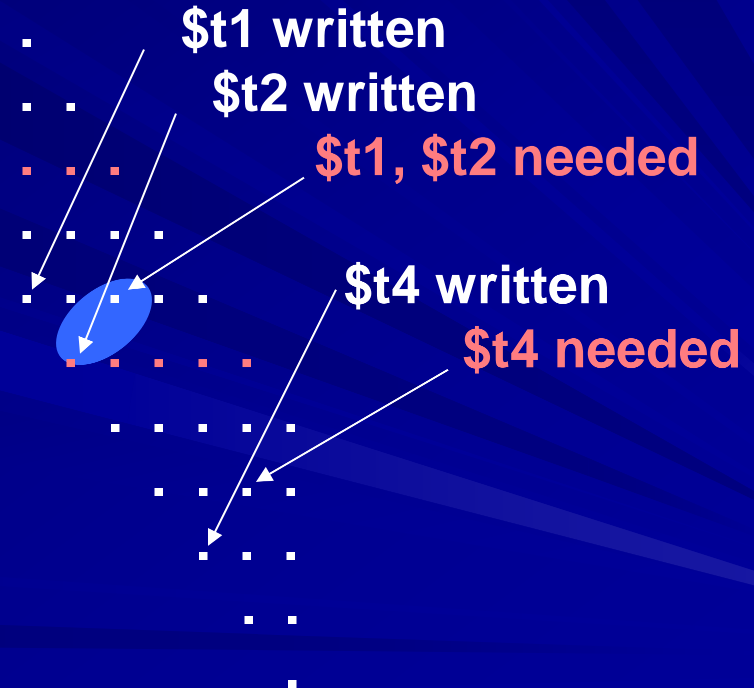
Avoiding Stall by Code Reorder

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:

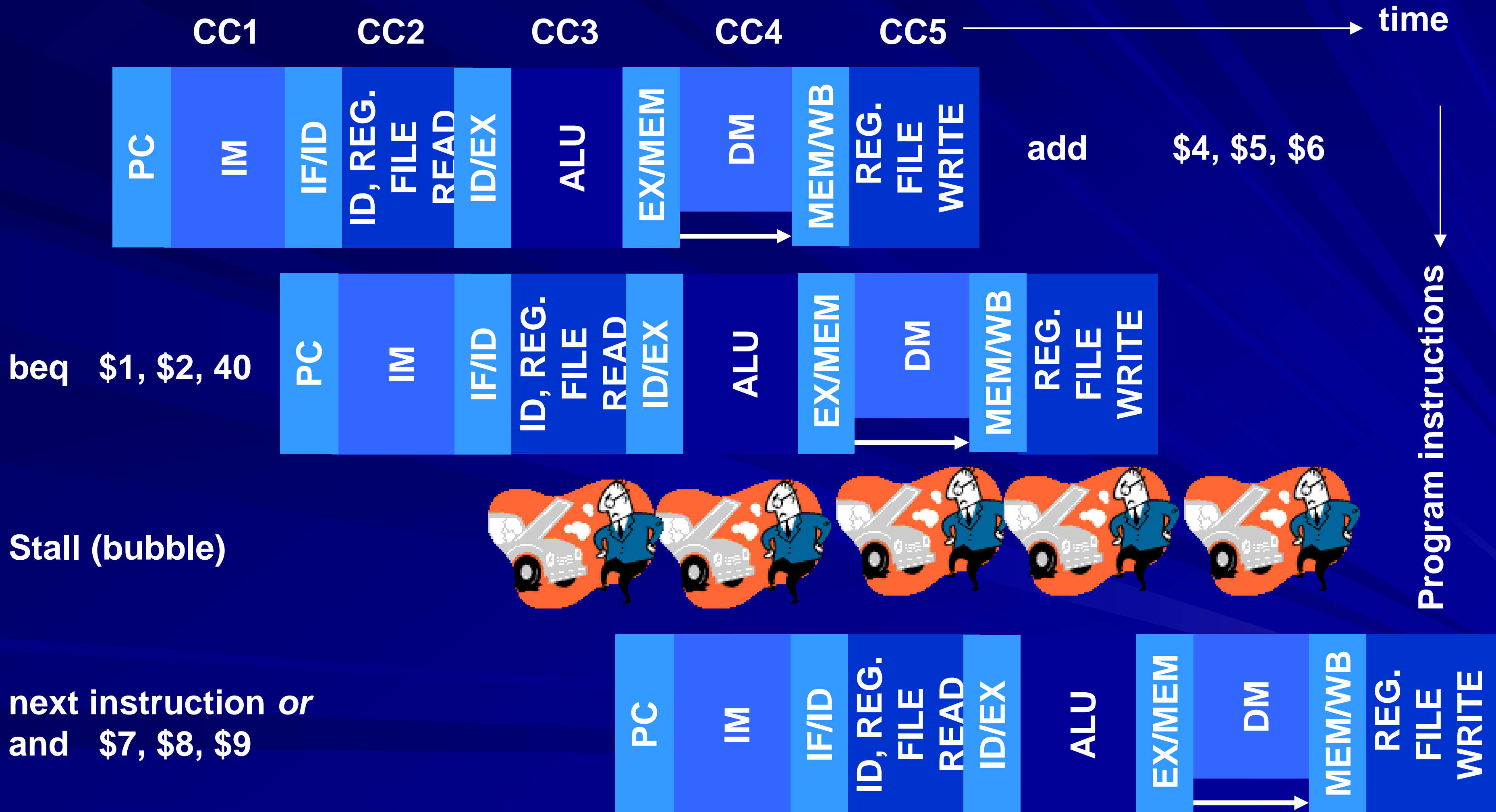
```
lw    $t1,  0($t0)
lw    $t2,  4($t0)
lw    $t4,  8($t0)
add   $t3,  $t1, $t2        no hazard
sw    $t3,  12($t0)
add   $t5,  $t1, $t4        no hazard
sw    $t5,  16($t0)
```

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

Delayed Branch Example

- Stall on branch

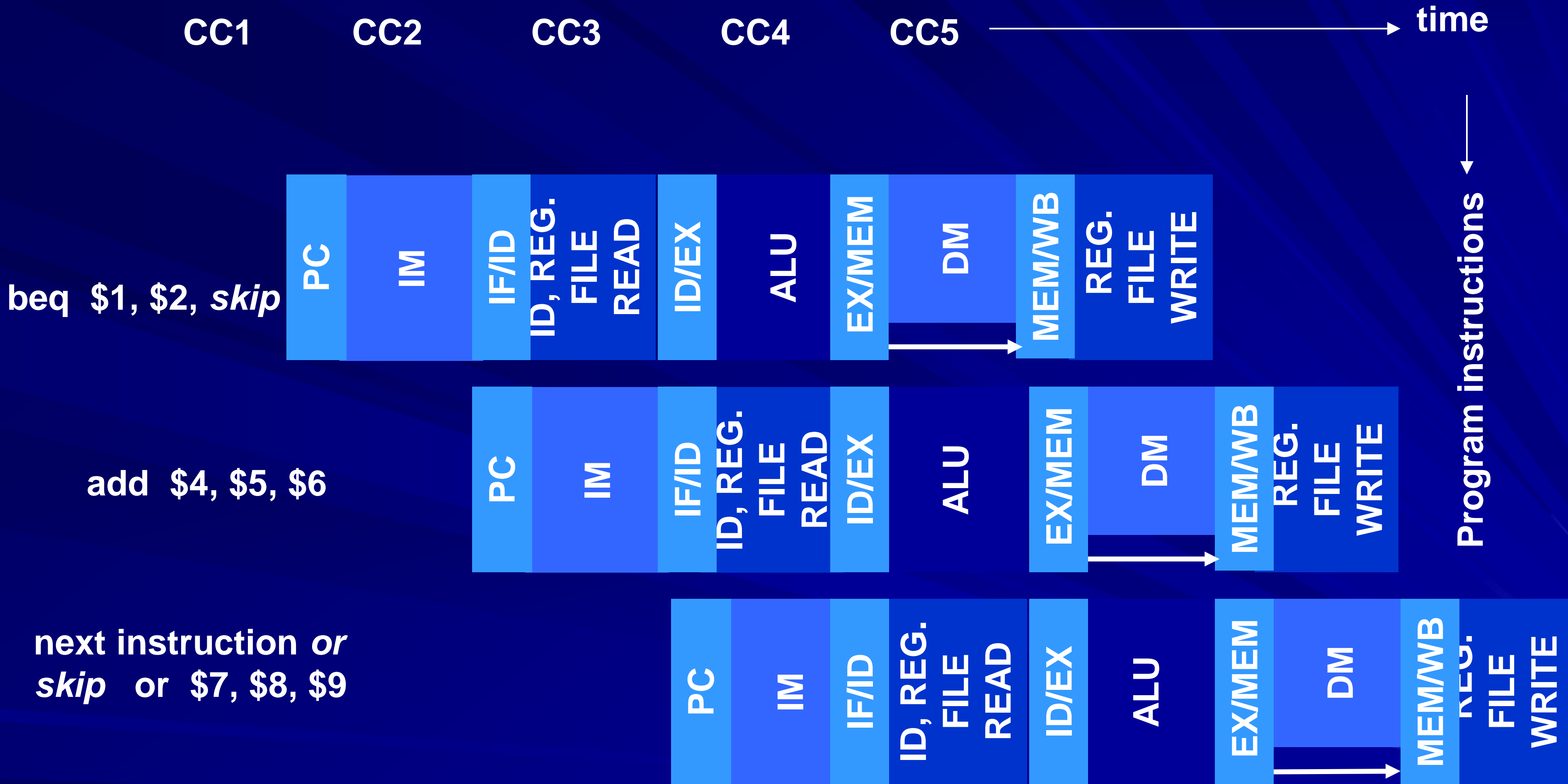
```
      add $4, $5, $6
      beq $1, $2, skip
      next instruction
      . . .
skip  or $7, $8, $9
```

- 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



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