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

Spring 2019

Pipelining

Christopher B. Harris
Assistant Professor
Department of Electrical and Computer Engineering
Auburn University, Auburn, AL 36849

(Adapted from slides by Vishwani D. Agrawal)

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

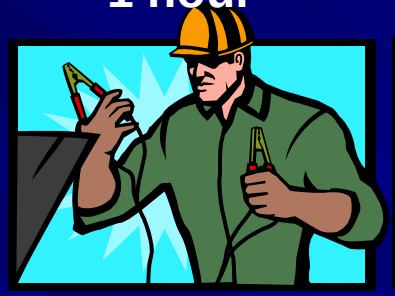
1 car assembled every four hours
6 cars per day
180 cars per month
2,040 cars per year

Automobile Assembly Line

Task 1
1 hour

Mecahnical

Task 2
1 hour



Electrical

Task 3
1 hour

Painting

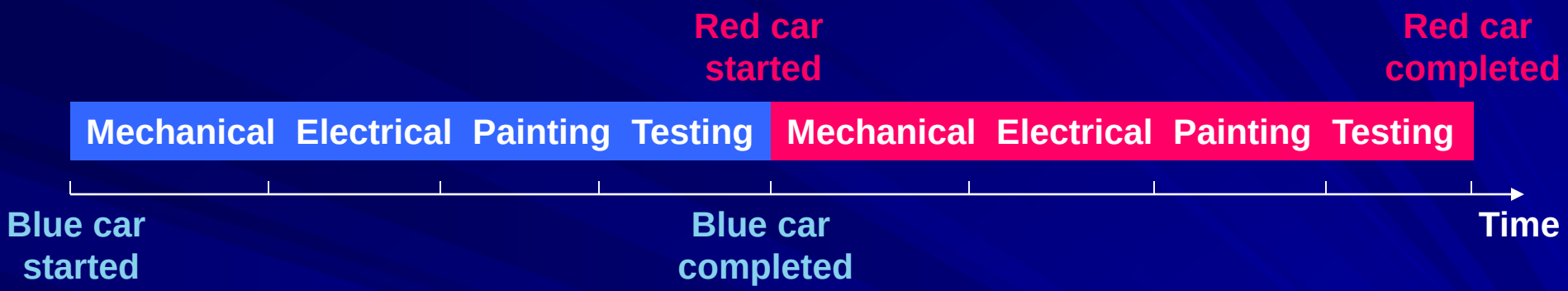
Task 4
1 hour

Testing

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

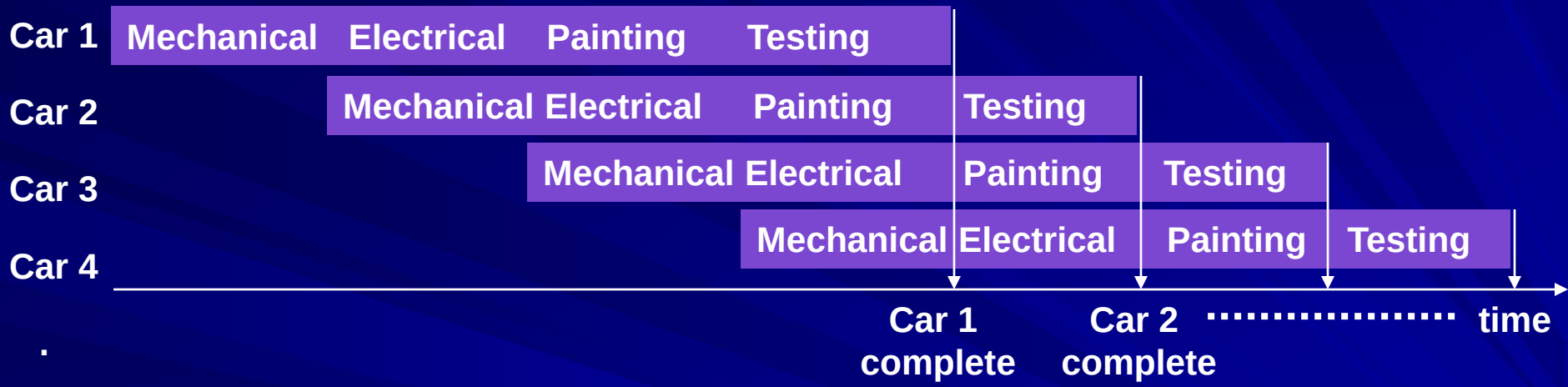


Time of assembling one car = *n* hours

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

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

Some Features of Assembly Line

Electrical parts delivered (JIT)

Task 1
1 hour

Task 2
1 hour

Task 3
1 hour

Task 4
1 hour

Mechanical

Electrical

Painting

Testing

3 cars in the assembly line are suspects, to be removed (flush pipeline)

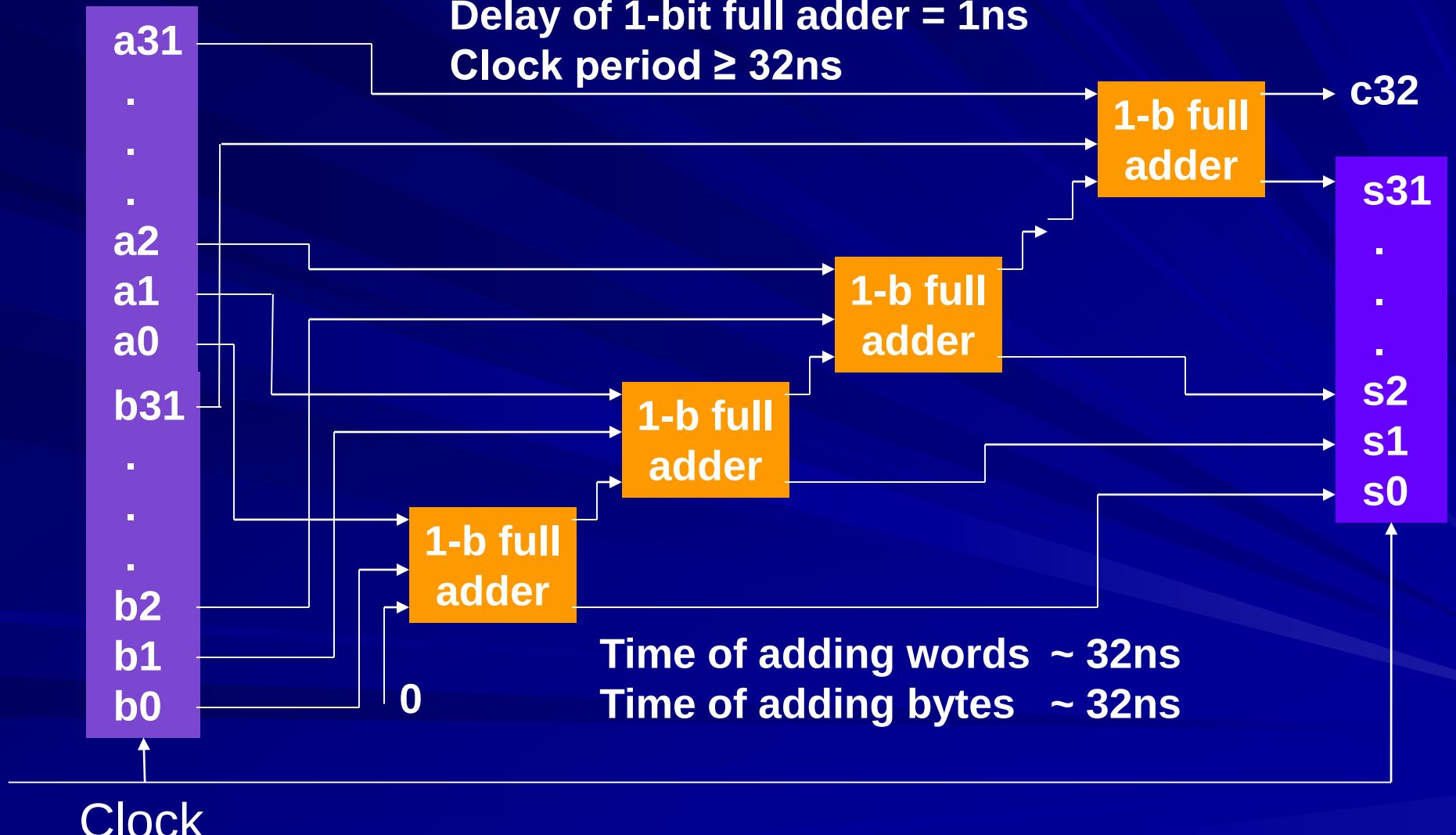
Stall assembly line to fix the cause of defect

Defect found

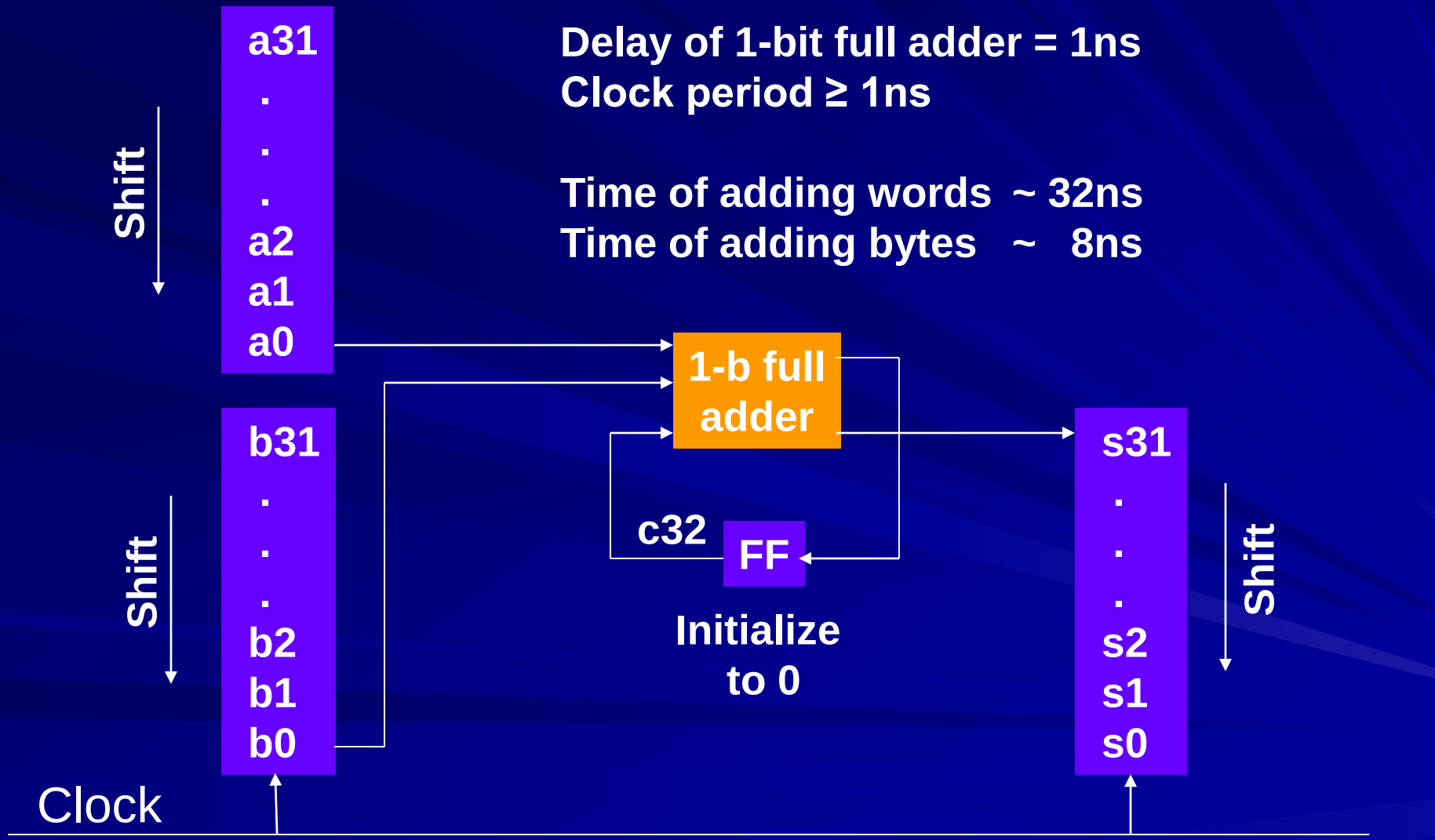
Pros and Cons

- 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.
  - https://www.youtube.com/watch?v=IjarLbD9r30
  - https://www.youtube.com/watch?v=ANXGJe6i3G8
  - https://www.youtube.com/watch?v=5lp4EbfPAtI

A Single Cycle Example

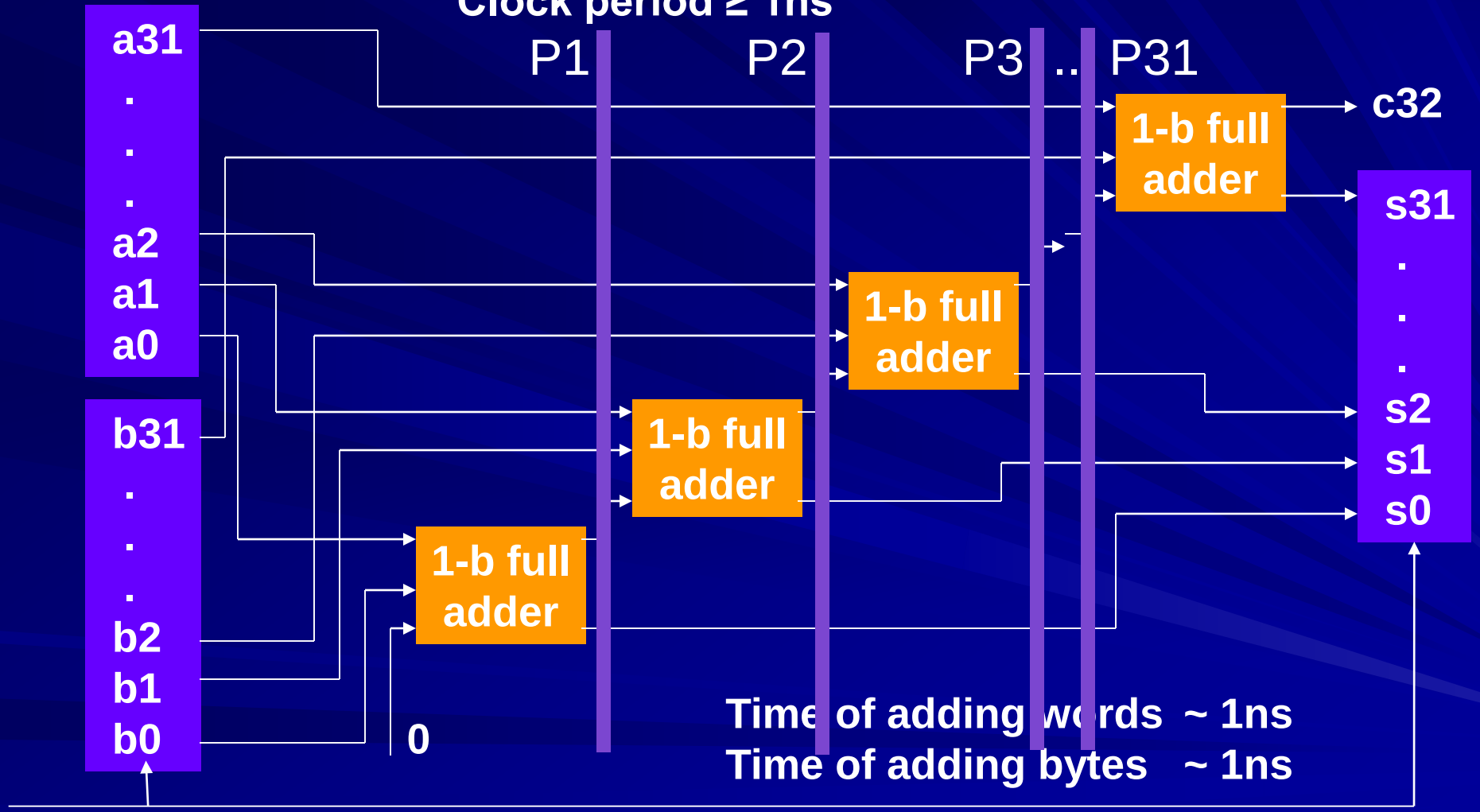


A Multicycle Implementation



A Pipeline Implementation

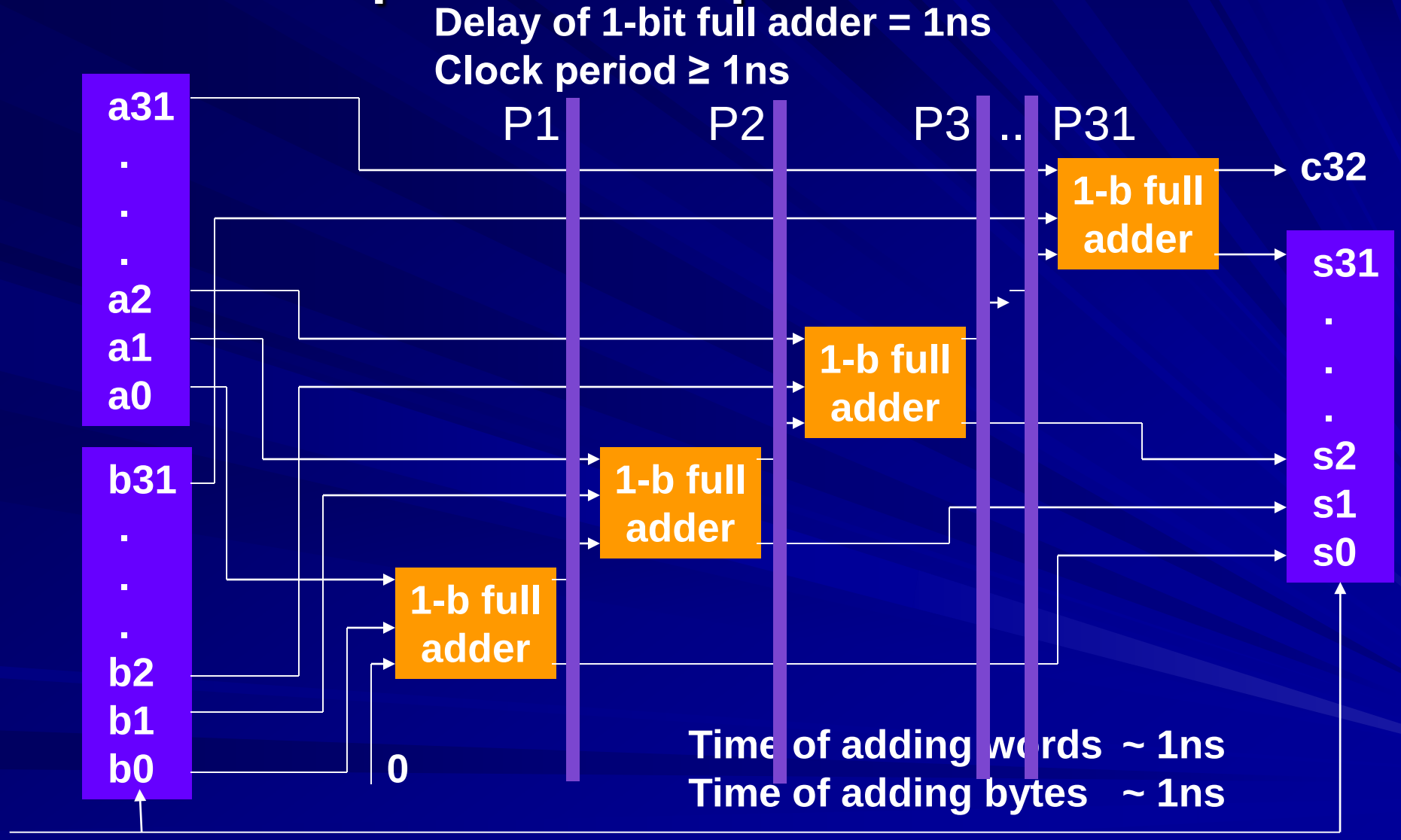
Delay of 1-bit full adder = 1ns
Clock period ≥ 1ns



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 just enough to complete the longest task.
- Break each instruction down into a set of tasks so that instructions can be executed in a staggered fashion.

A Pipeline Implementation



Pipelining of RISC Instructions (From Lecture 3)

IF	ID	EX	MEM	WB
Instruction Fetch	Decode instruction and Fetch operands	Execute	Memory Operation	Write Back to Reg file

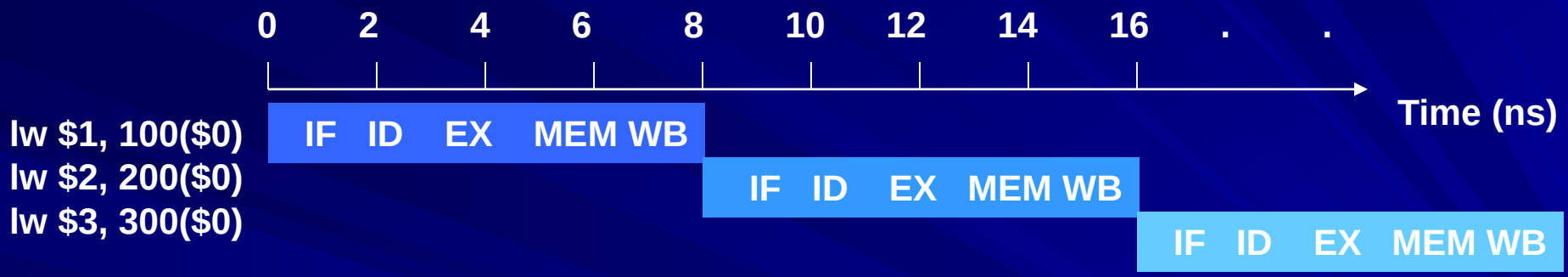
Although an instruction takes five clock cycles, one instruction is completed every cycle.

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

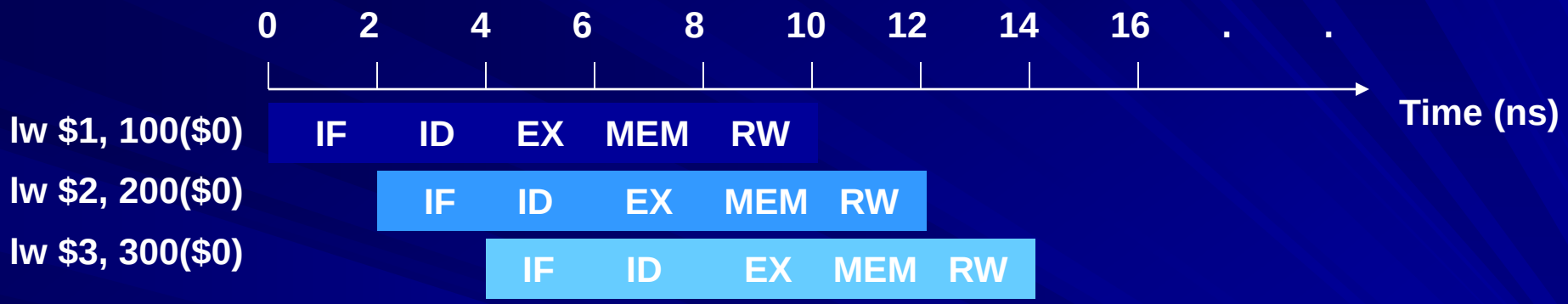
Total time for executing three lw instructions = 24 ns

Pipelined Datapath

Instruction class	Instr. fetch (IF)	Instr. Decode (also reg. file read) (ID)	Execu-tion (ALU Opera-tion) (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

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

Pipeline Performance

Clock cycle time = 2 ns

1,003 *lw instructions:*

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

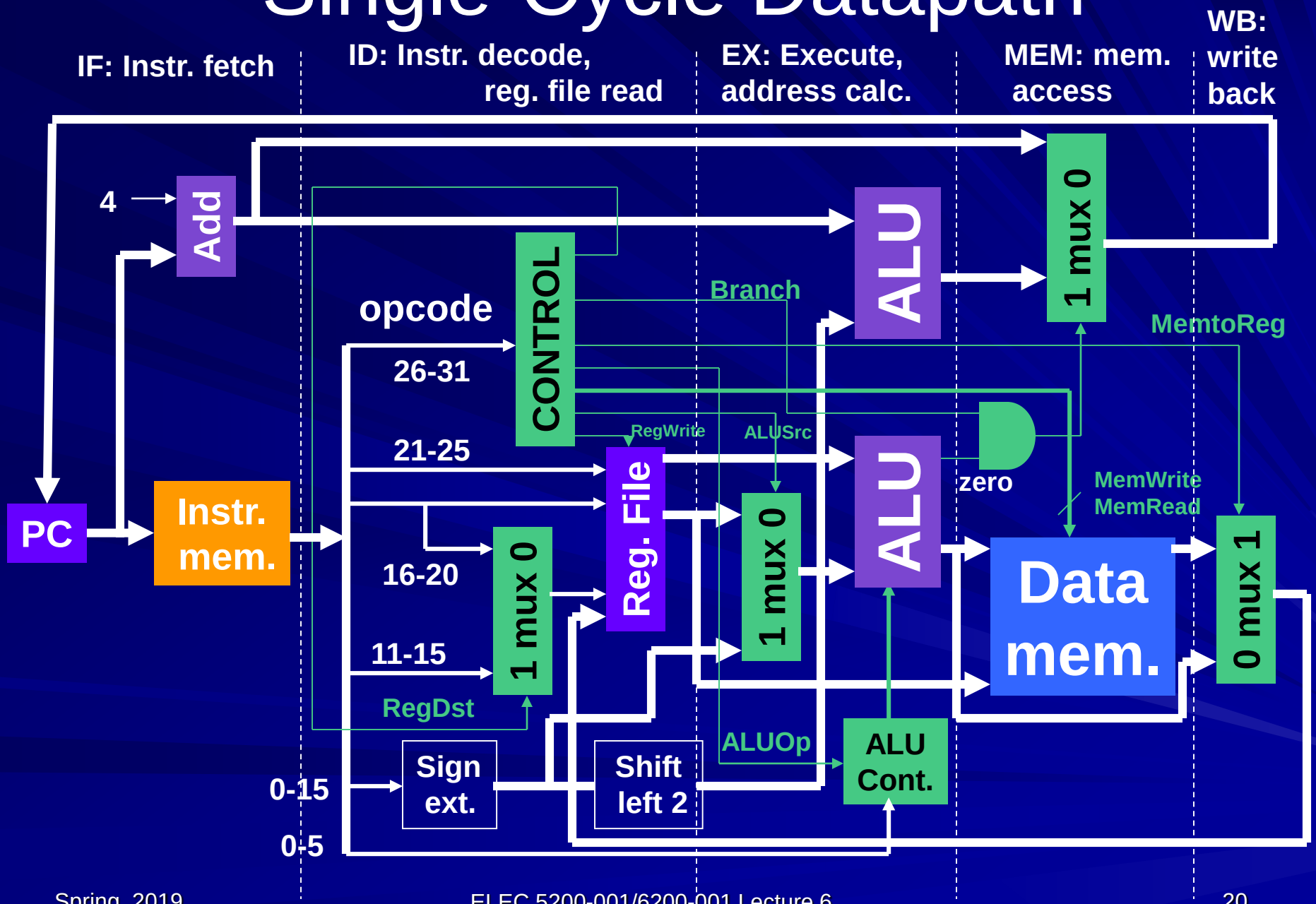
$$\text{Performance ratio} = \frac{\text{Single-cycle time}}{\text{Pipeline time}} = \frac{8{,}024}{2{,}014} = 3.98$$

10,003 *lw instructions:*

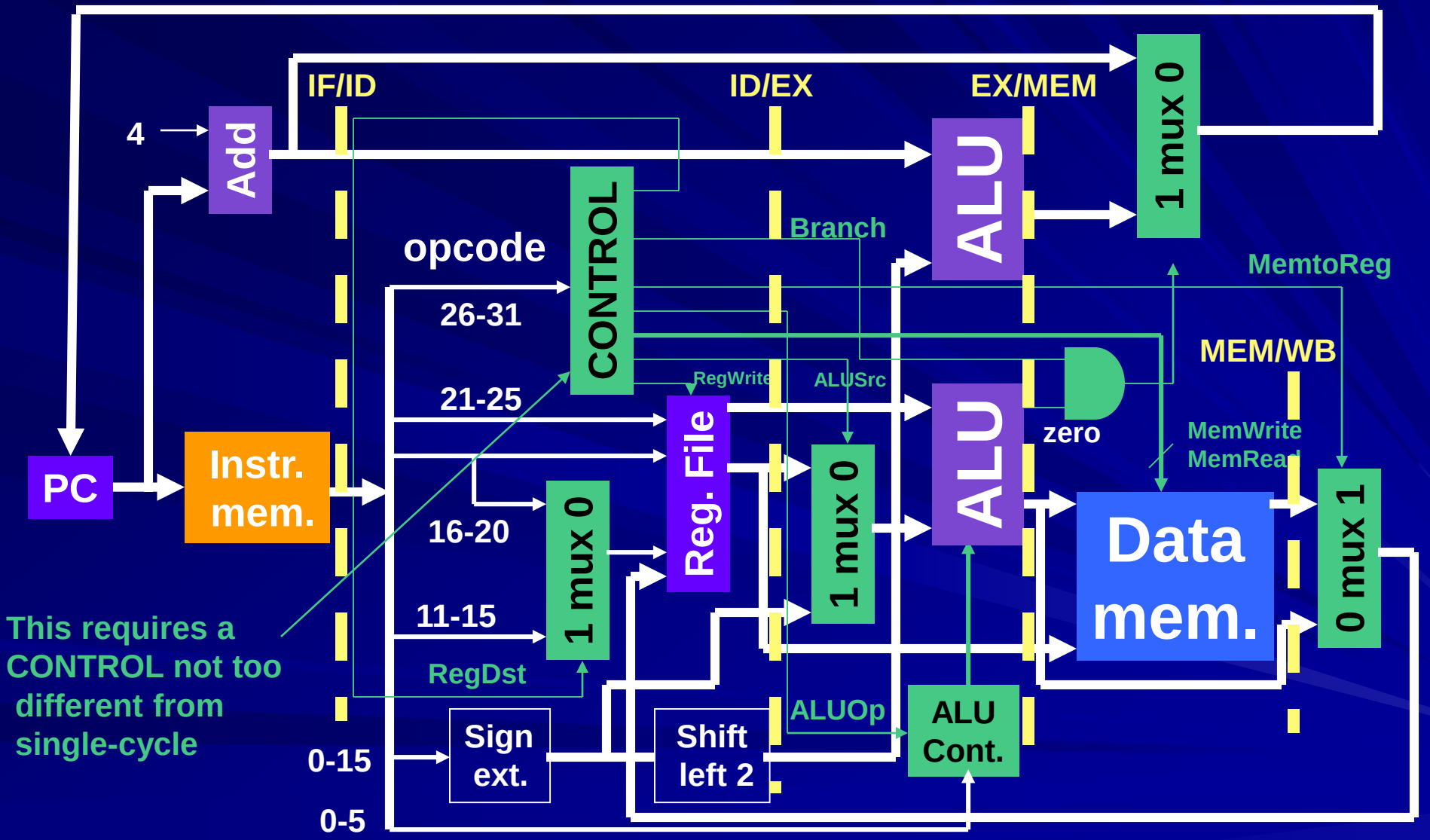
Performance ratio = 80,024 / 20,014 = 3.998 → Clock cycle ratio (4)

Pipeline performance approaches clock-cycle ratio for long programs.

Single-Cycle Datapath



Pipeline Registers

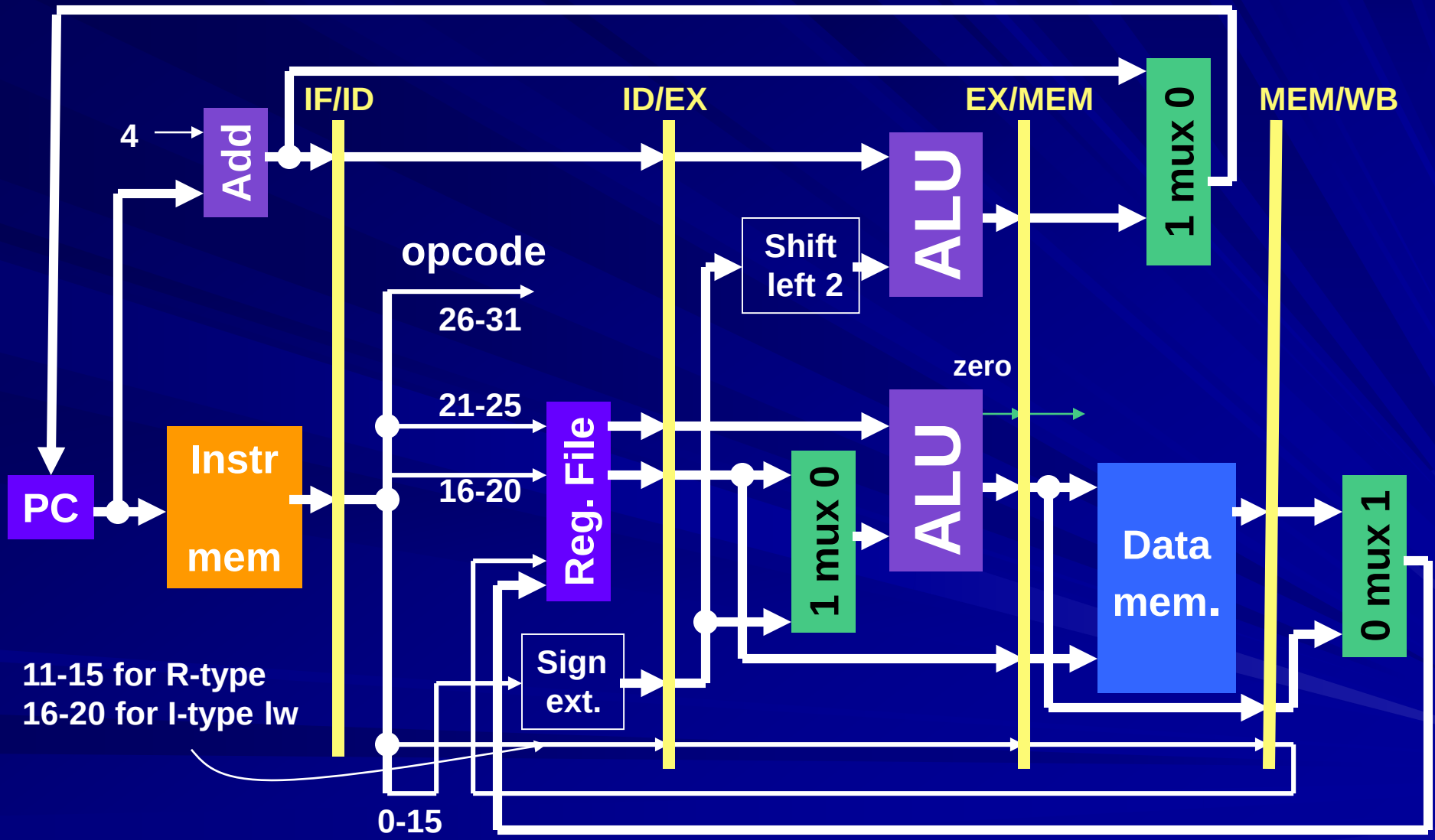


Pipeline Register Functions

- Four pipeline registers are added:

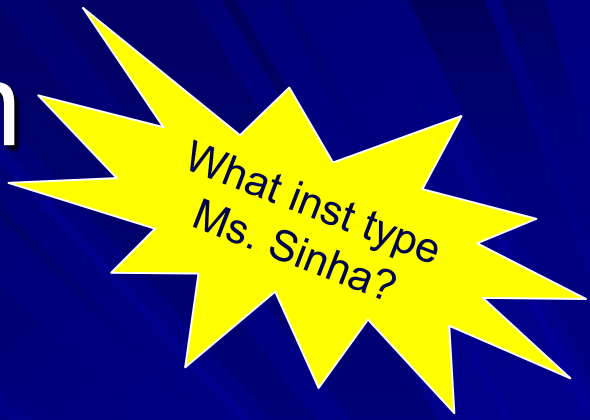
Register name	Data held
IF/ID	PC+4, Instruction word (IW)
ID/EX	PC+4, R1, R2, IW(0-15) sign ext., IW(11-15)
EX/MEM	PC+4, zero, ALUResult, R2, IW(11-15) or IW(16-20)
MEM/WB	M[ALUResult], ALUResult, IW(11-15) or IW(16-20)

Pipelined Datapath



Five-Cycle Pipeline

Add Instruction

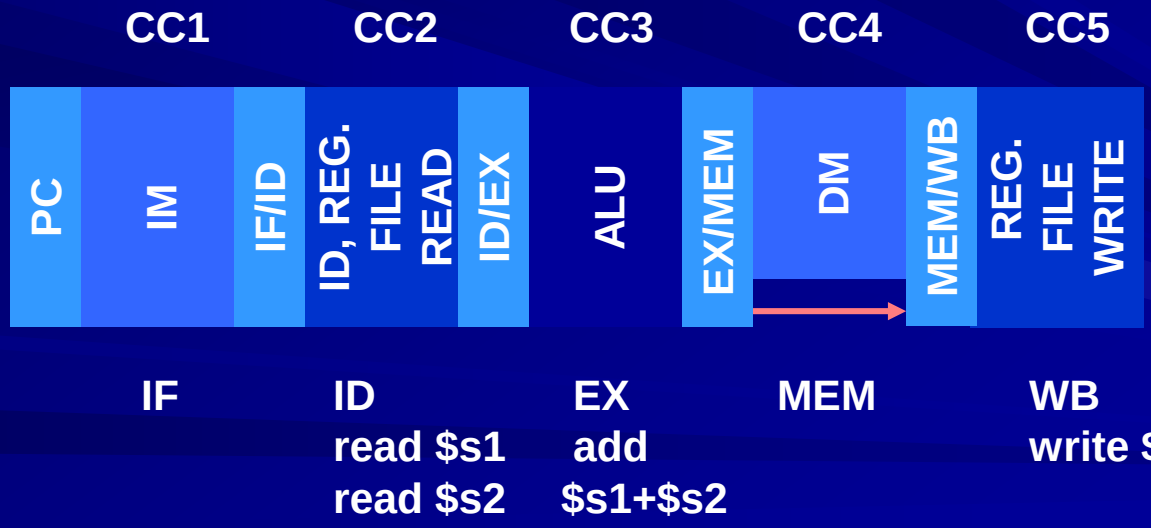


- add $t0, $s1, $s2

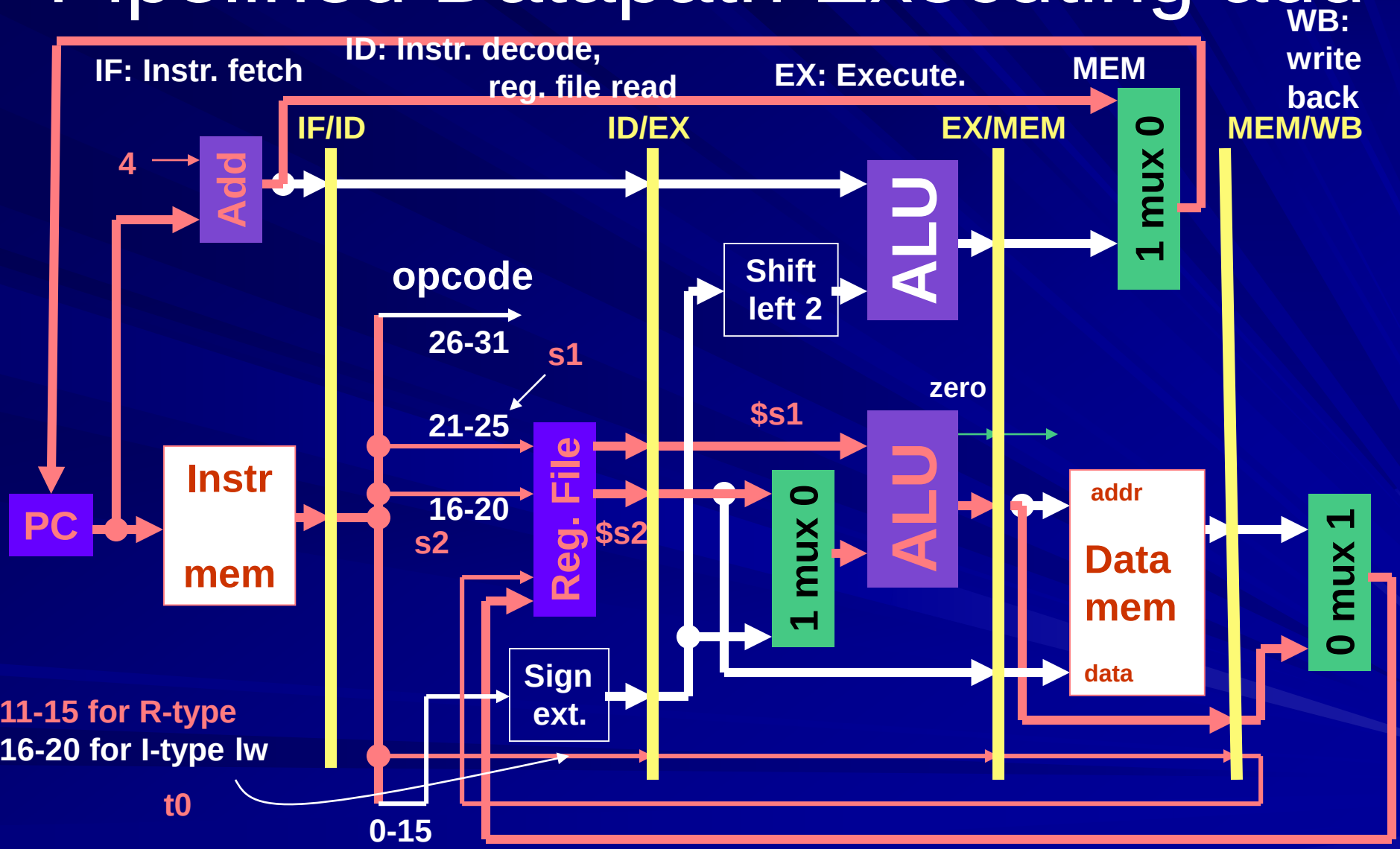
  Machine instruction word

  000000 10001 10010 01000 00000 100000

  opcode $s1 $s2 $t0 function



Pipelined Datapath Executing add

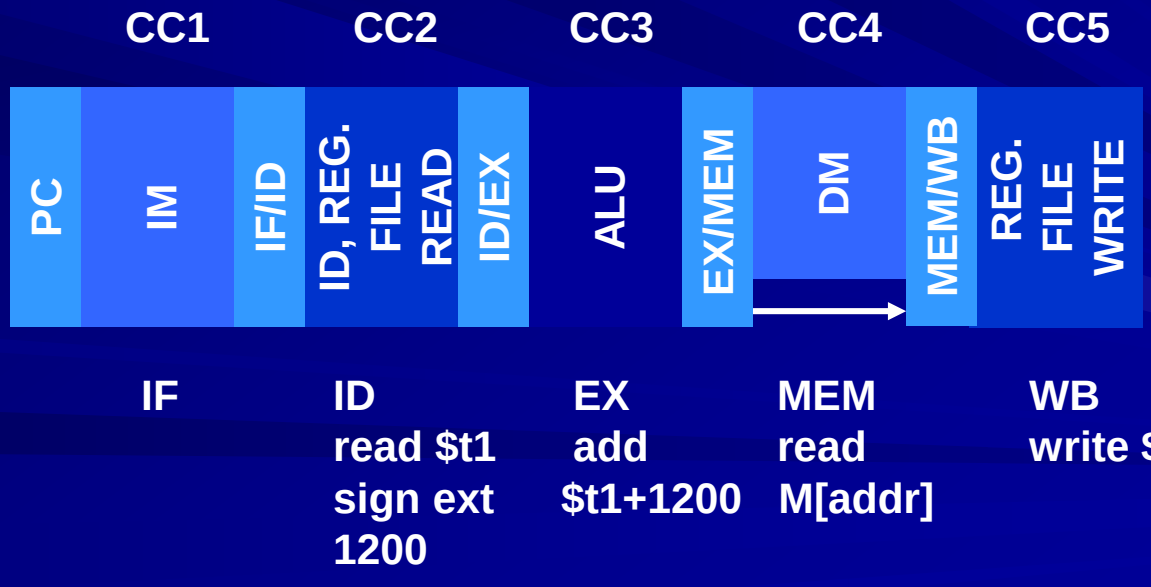


Load Instruction

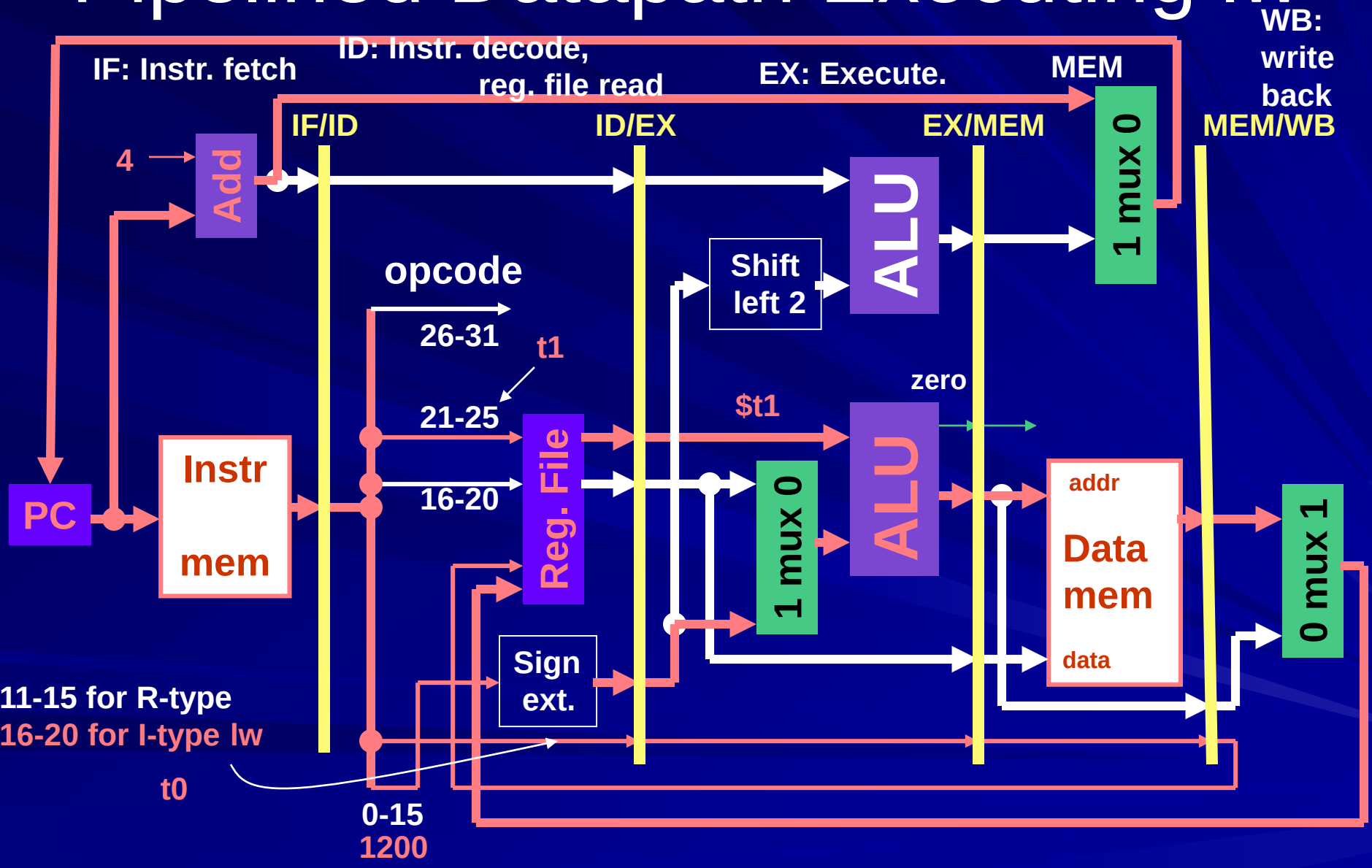
- lw $t0, 1200 ($t1)

100011 01001 01000 0000 0100 1000 0000

opcode $t1 $t0 1200



Pipelined Datapath Executing lw

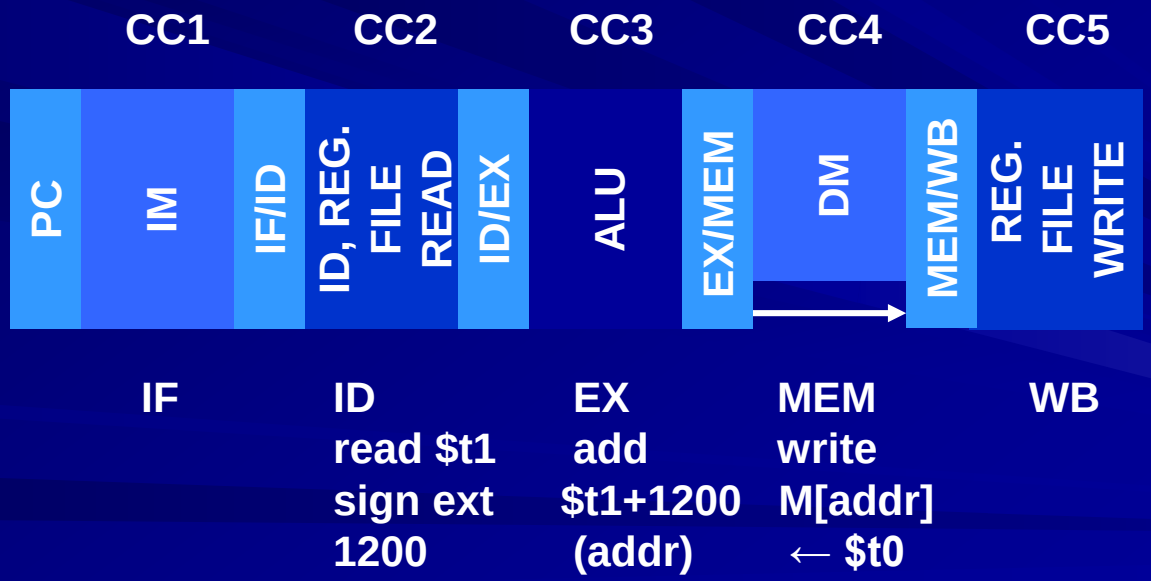


Store Instruction

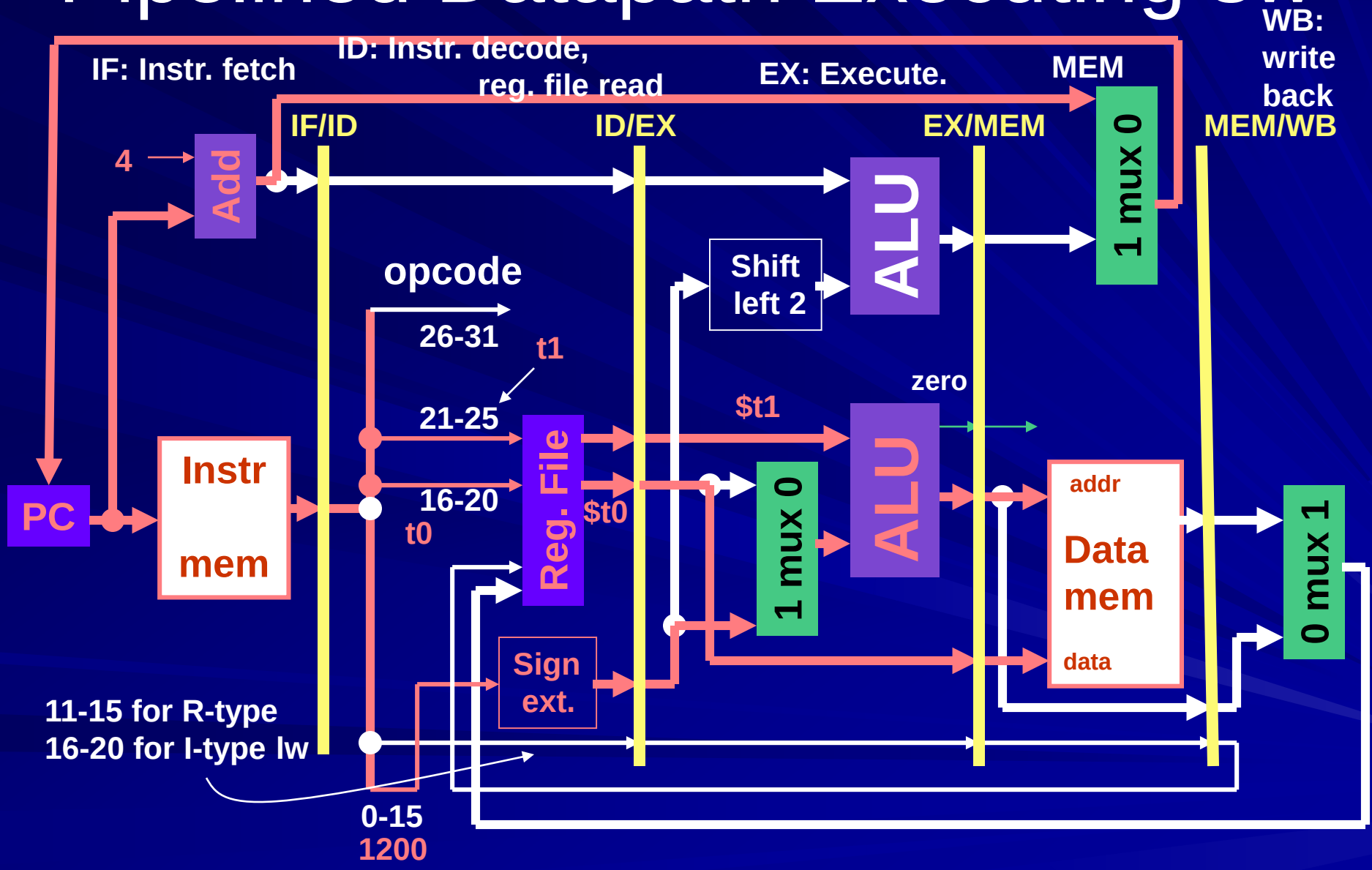
- sw $t0, 1200 ($t1)

101011 01001 01000 0000 0100 1000 0000

opcode $t1 $t0 1200



Pipelined Datapath Executing sw

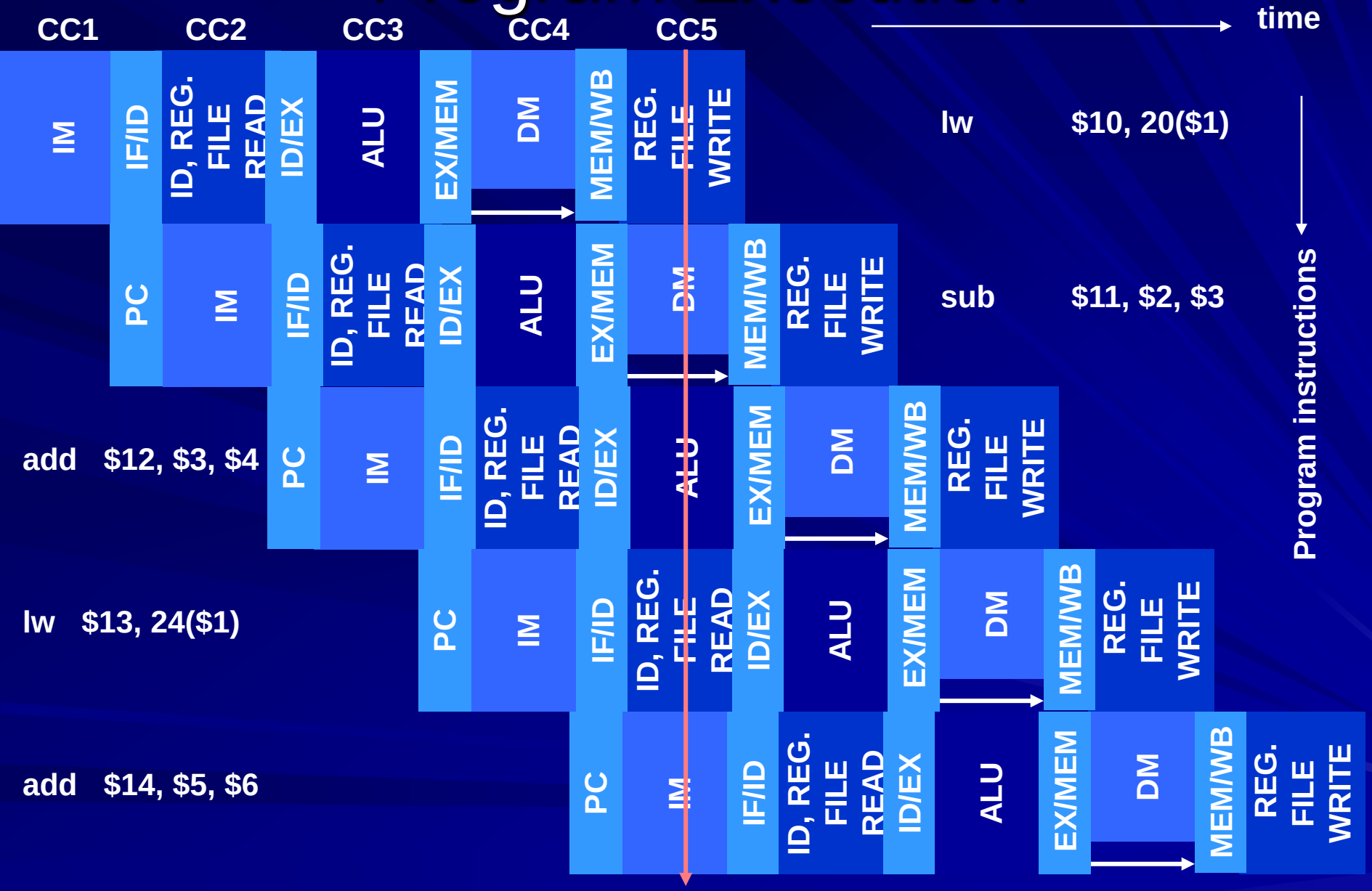


Executing a Program

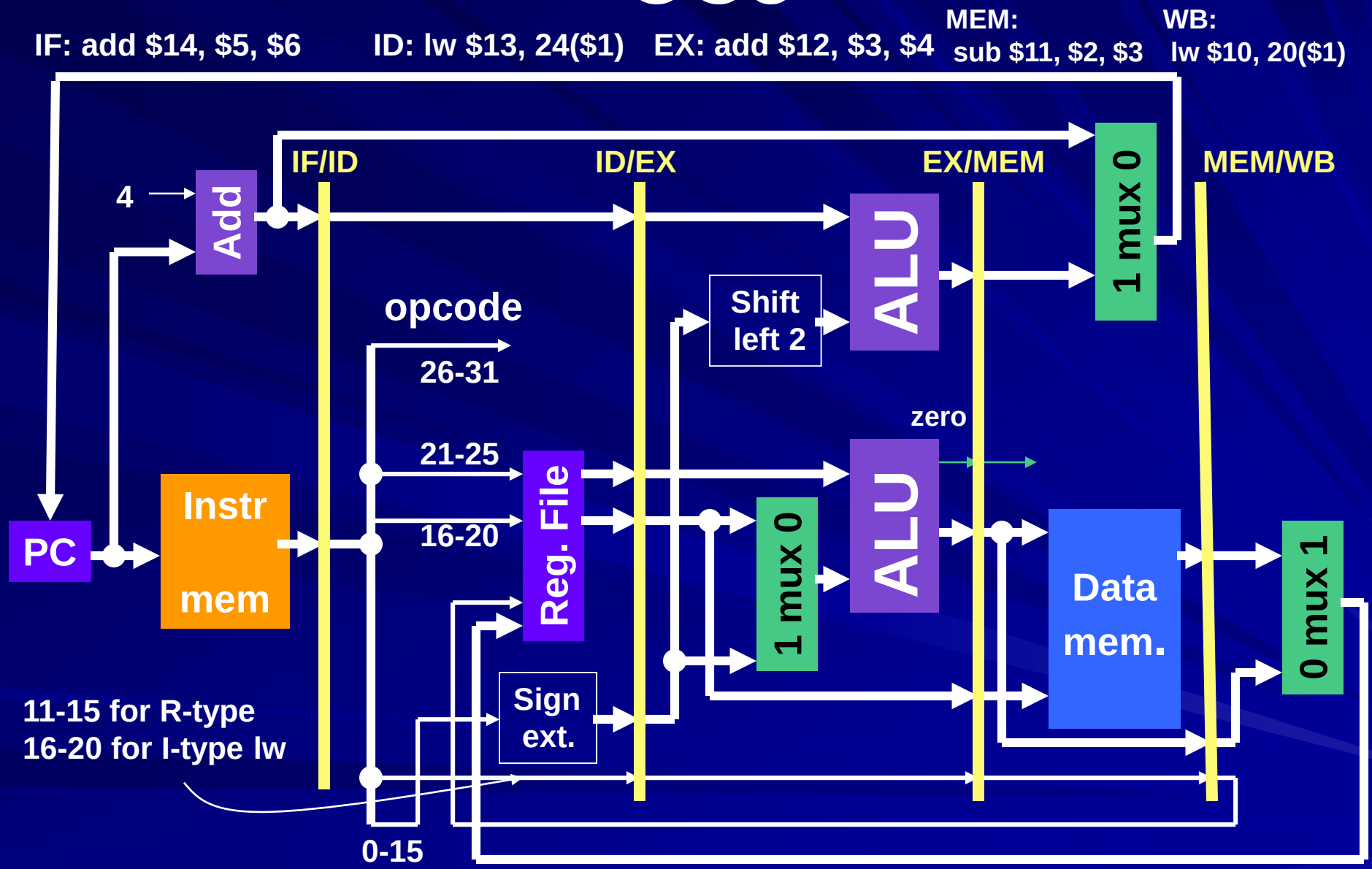
Consider a five-instruction segment:

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

Program Execution



CC5


IF: add $14, $5, $6
ID: lw $13, 24($1)
EX: add $12, $3, $4
MEM: sub $11, $2, $3
WB: lw $10, 20($1)
IF/ID
ID/EX
EX/MEM
MEM/WB
4
Add
PC
Instr mem
opcode
26-31
21-25
16-20
Reg. File
Shift left 2
ALU
ALU
zero
1 mux 0
1 mux 0
Data mem.
0 mux 1
Sign ext.
11-15 for R-type
16-20 for I-type lw
0-15

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

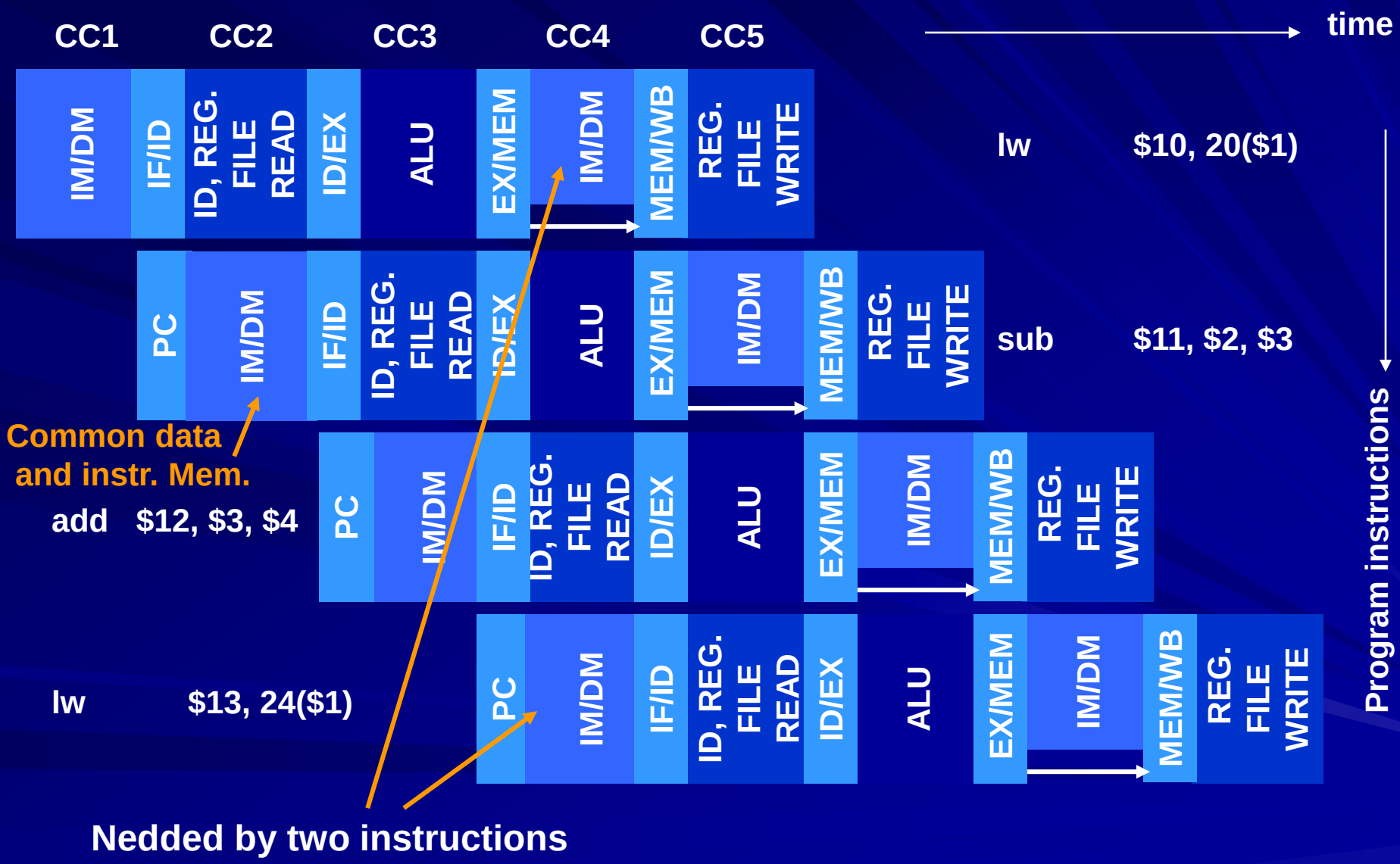
Pipeline Hazards

- 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
  - ~~Dukes of Hazzard~~

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

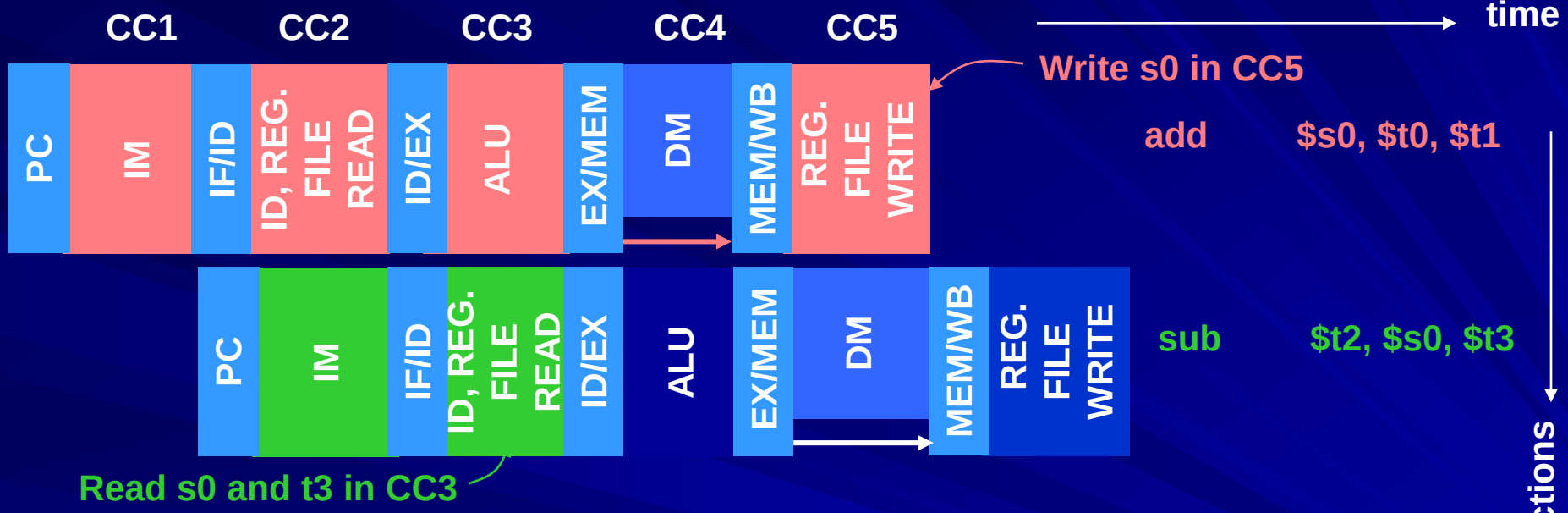
- Provide duplicate hardware resources in datapath.
- Control unit or compiler can insert delays (no-op cycles) between instructions. This is known as pipeline *stall* or *bubble*.

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



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

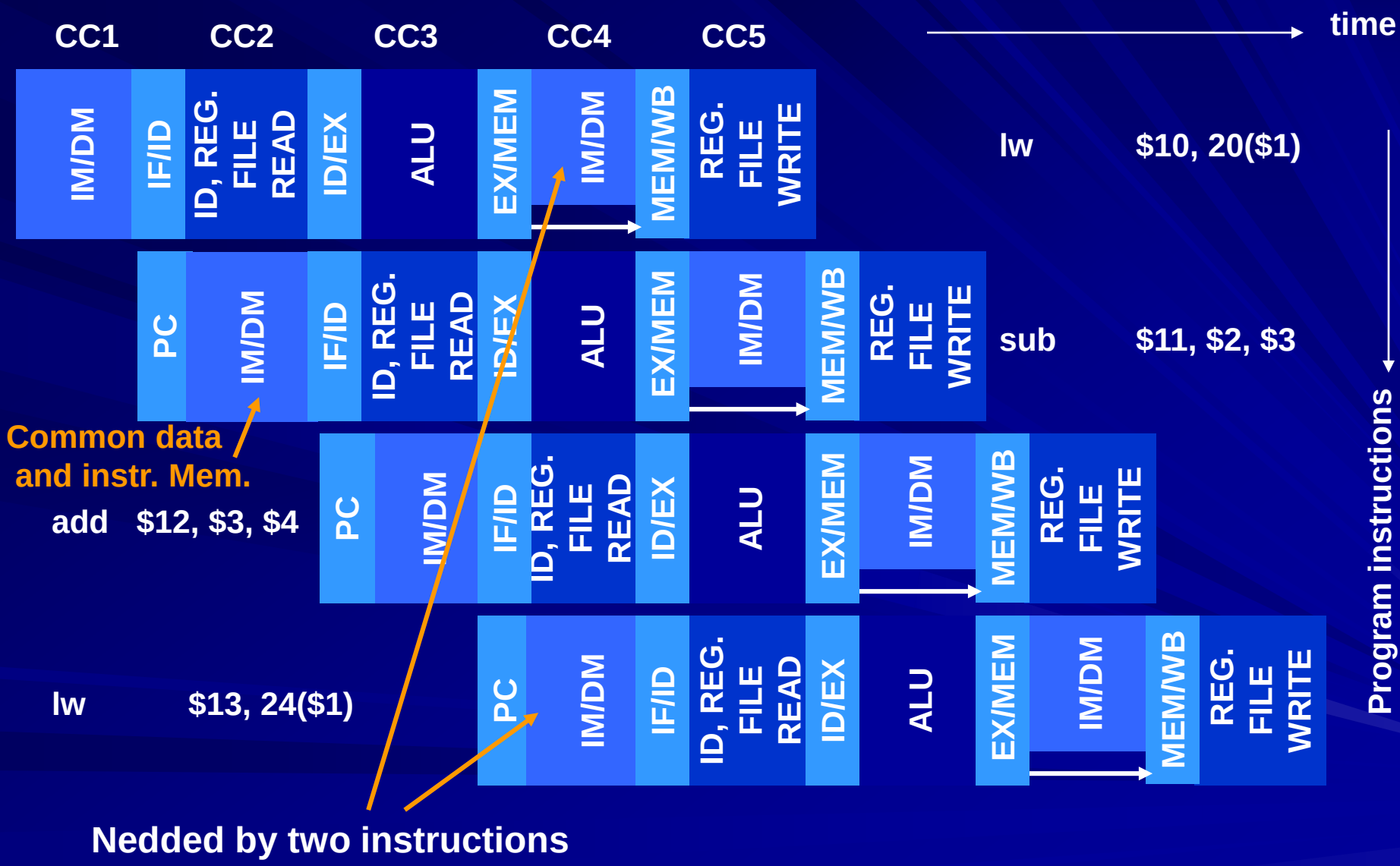
Forwarding or Bypassing

- 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.

Pipeline Hazards

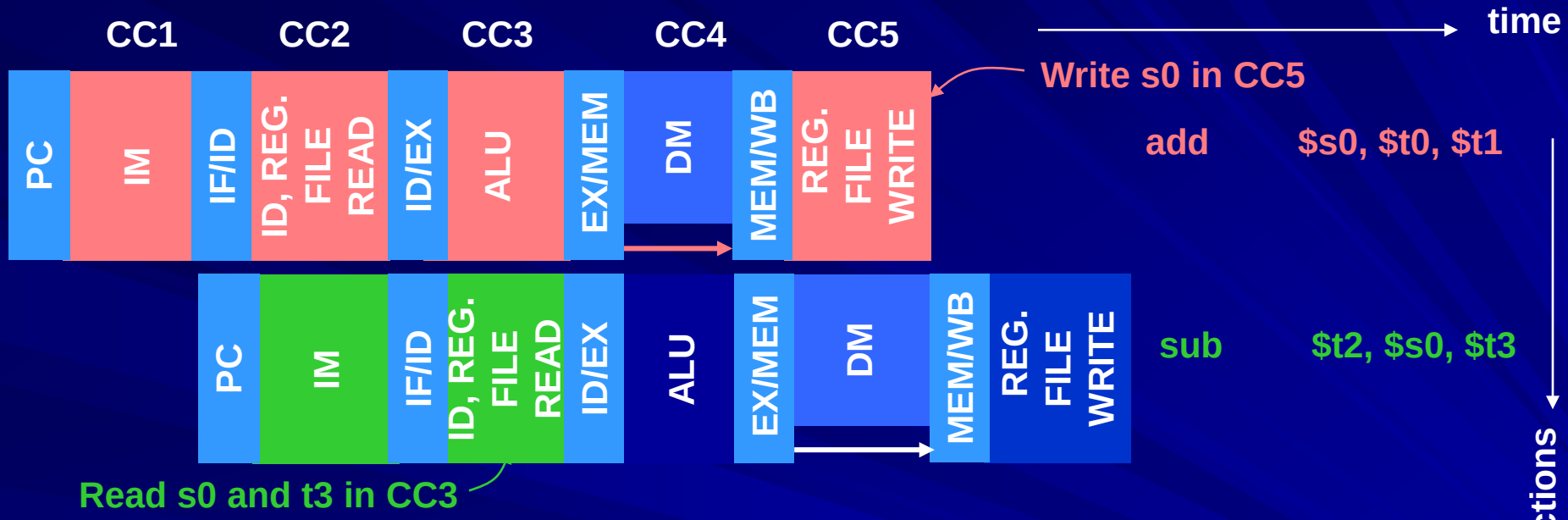
- Definition: *Hazard in a pipeline is a situation in which the next instruction cannot complete execution one clock cycle after completion of the present instruction*.
- Three types of hazards:
  - Structural hazard (resource conflict)
  - Data hazard
  - Control hazard

Example of Structural Hazard



Stall (Bubble) for Structural Hazard

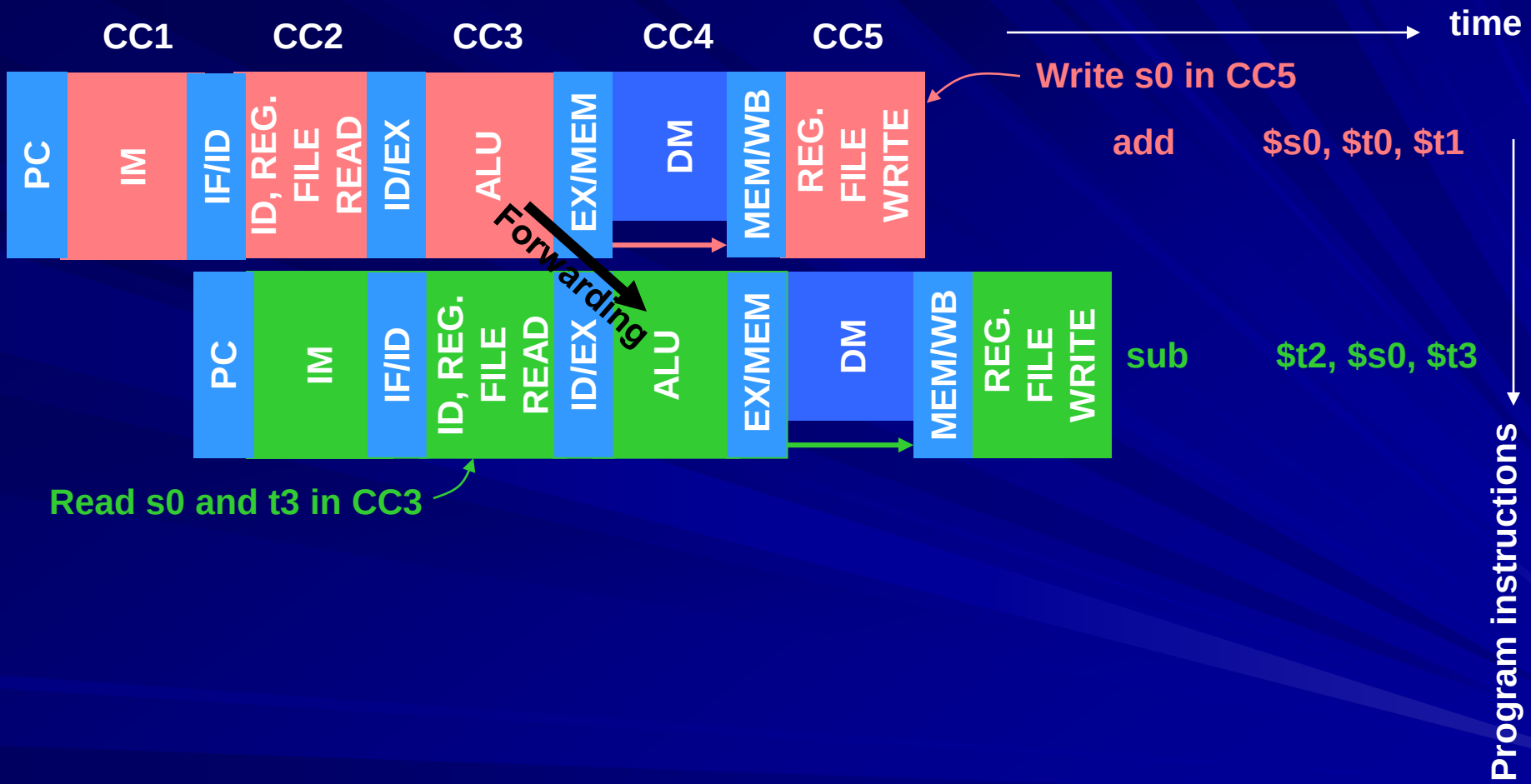
Example of Data Hazard



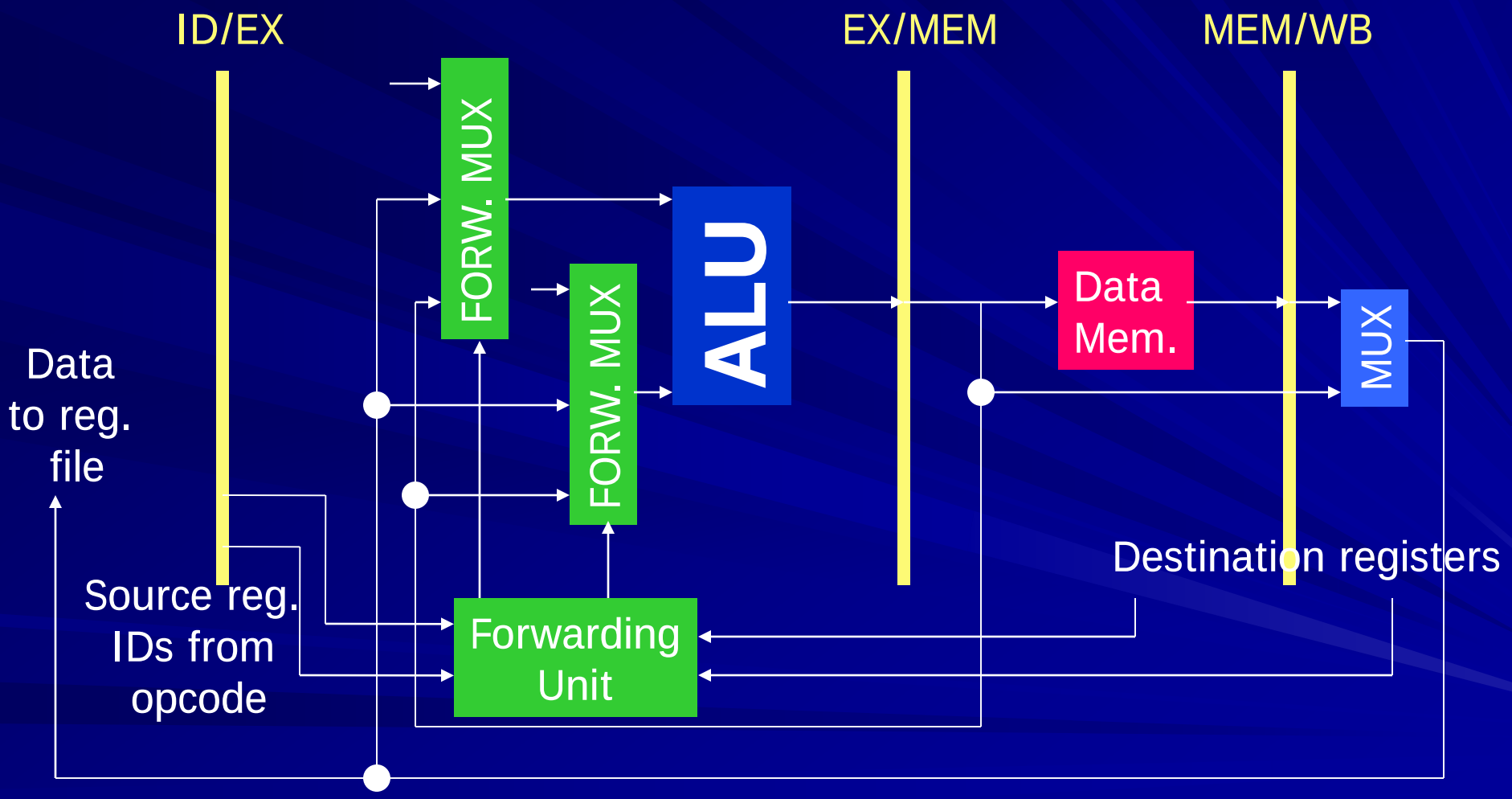
We need to read s0 from reg file in cycle 3
But s0 will not be written in reg file until cycle 5

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

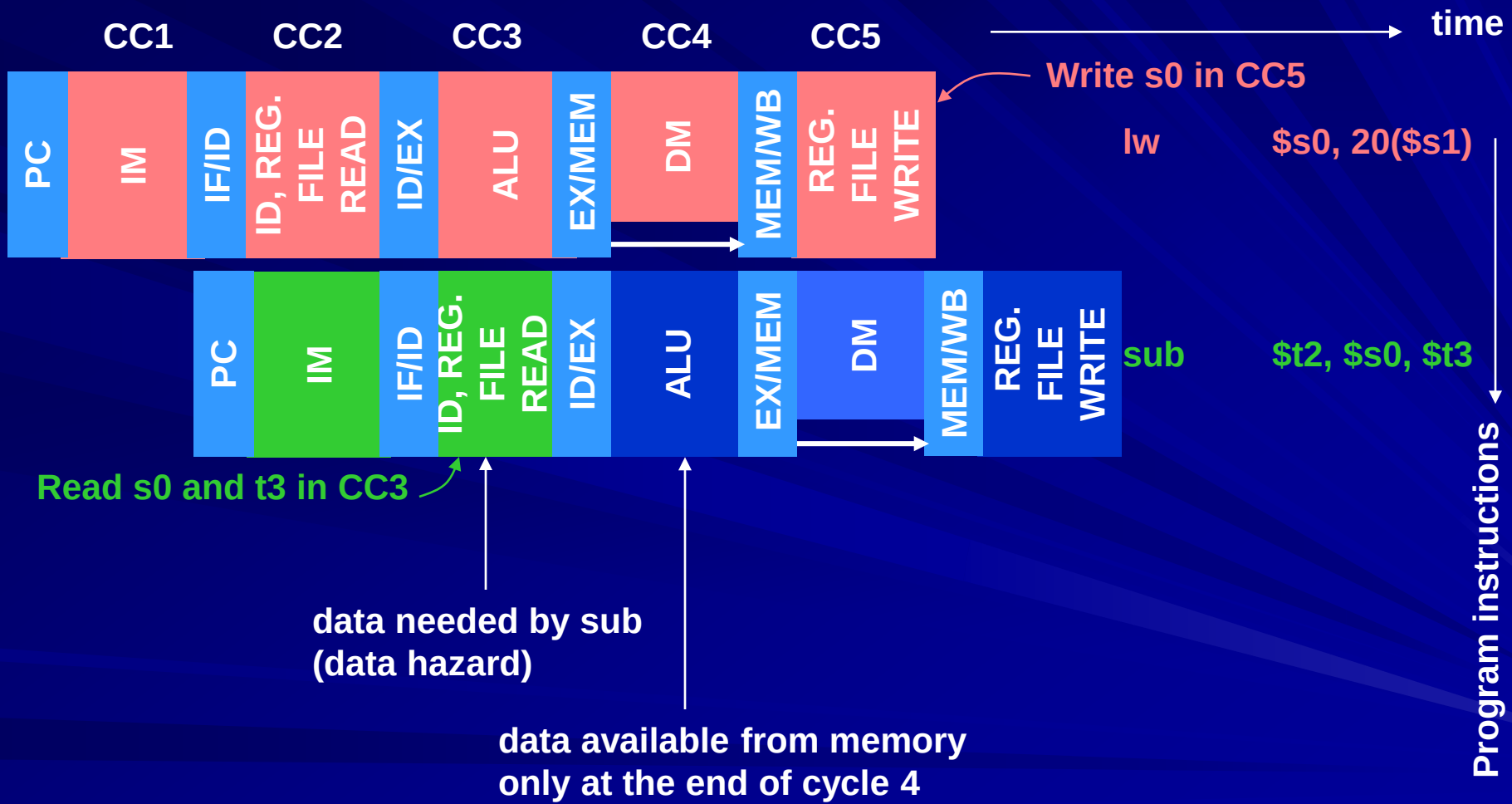
Forwarding for Data Hazard



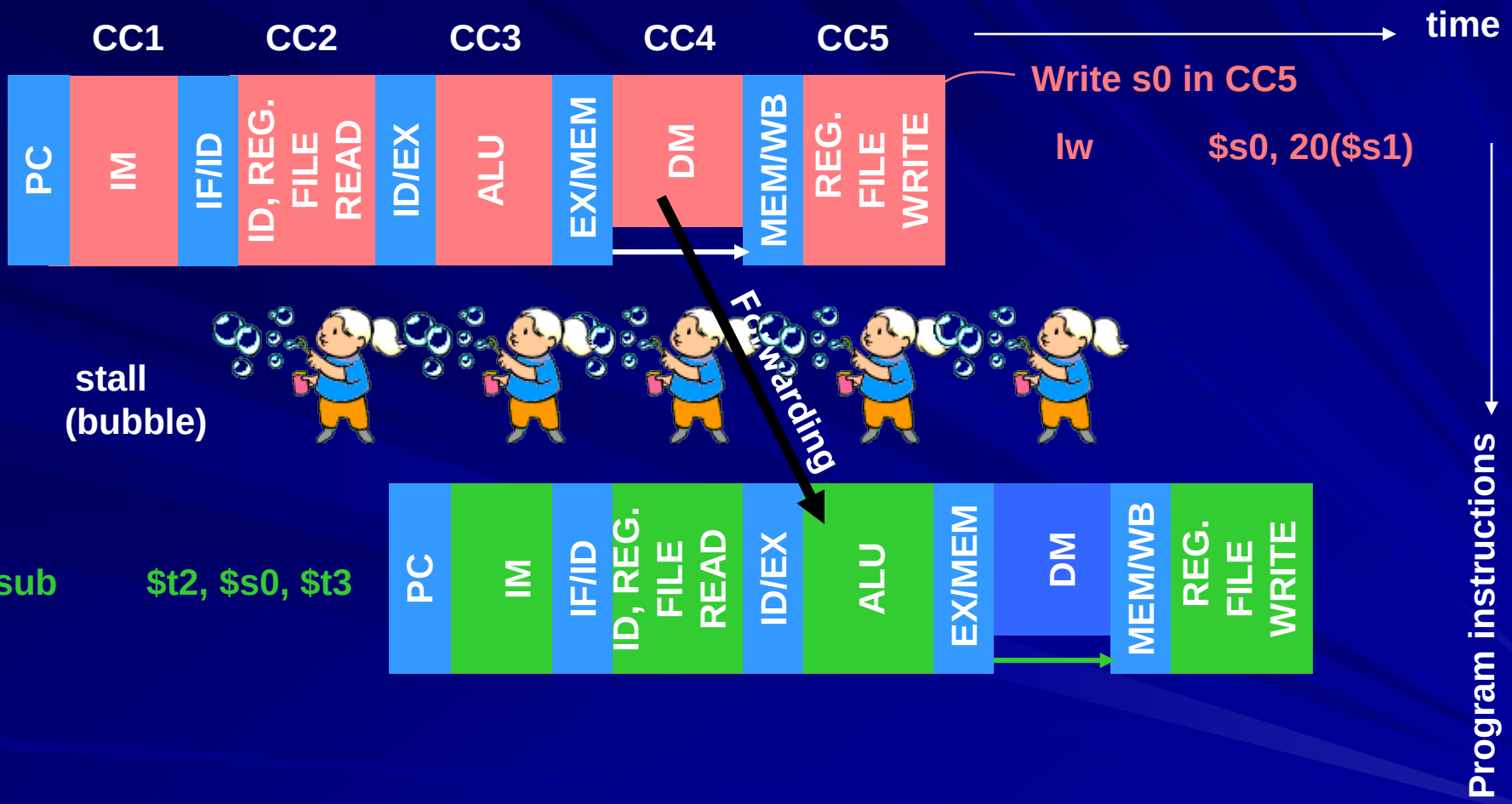
Forwarding Unit Hardware



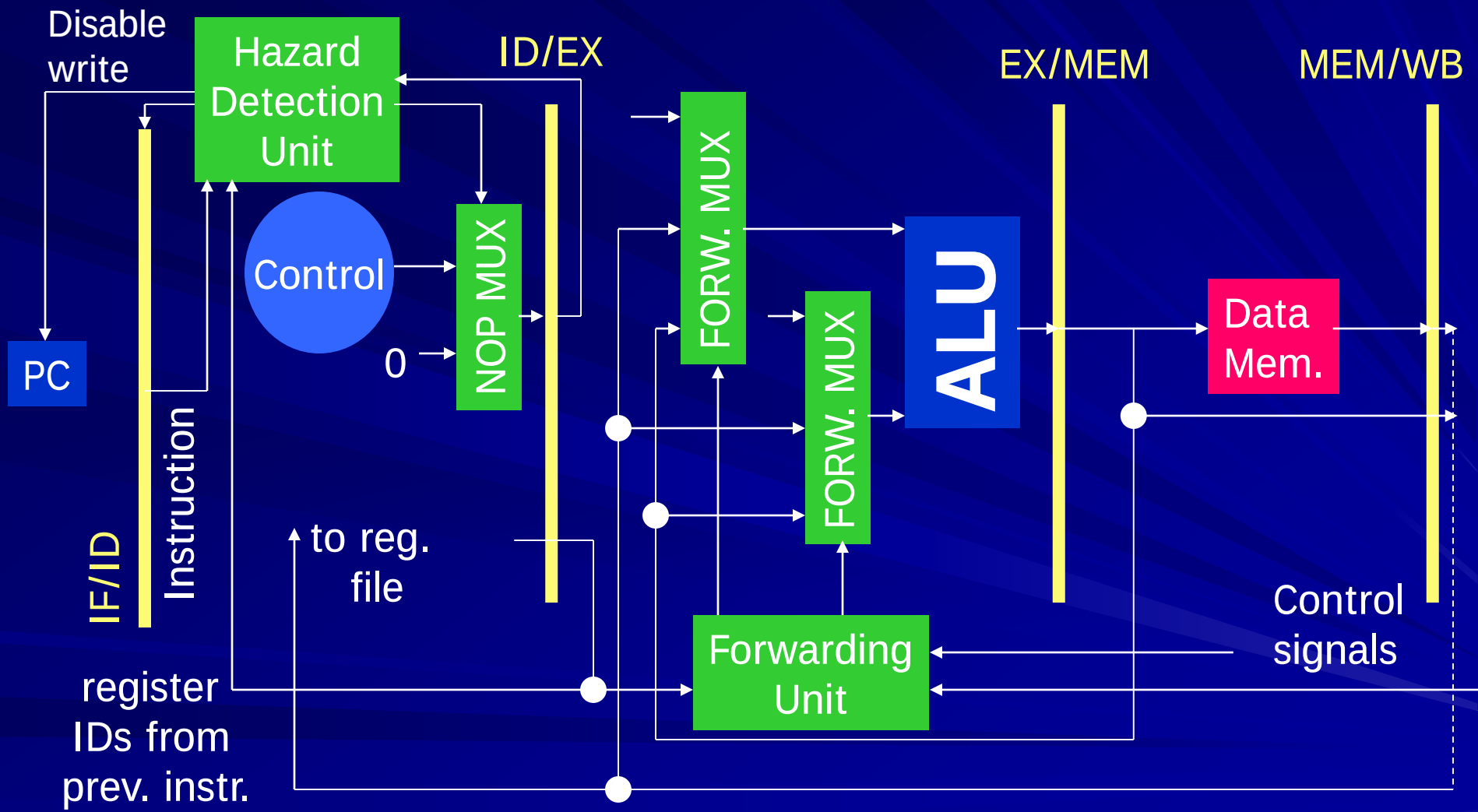
Forwarding Alone May Not Work



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:

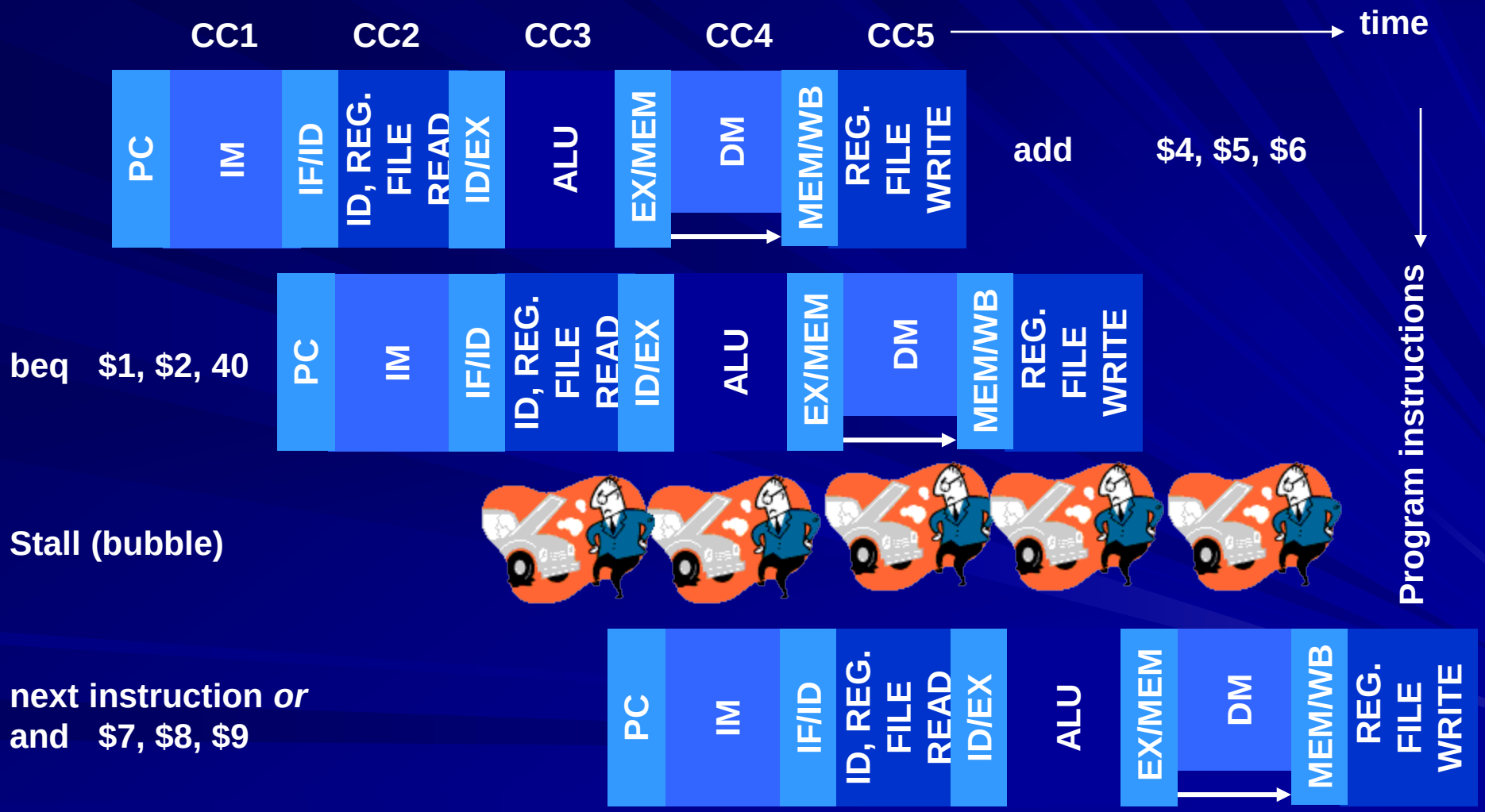
```
lw    $t1,  0($t0)
lw    $t2,  4($t0)
lw    $t4,  8($t0)
add   $t3,  $t1, $t2      no hazard
sw    $t3,  12($t0)
add   $t5,  $t1, $t4      no hazard
sw    $t5,  16($t0)
```

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

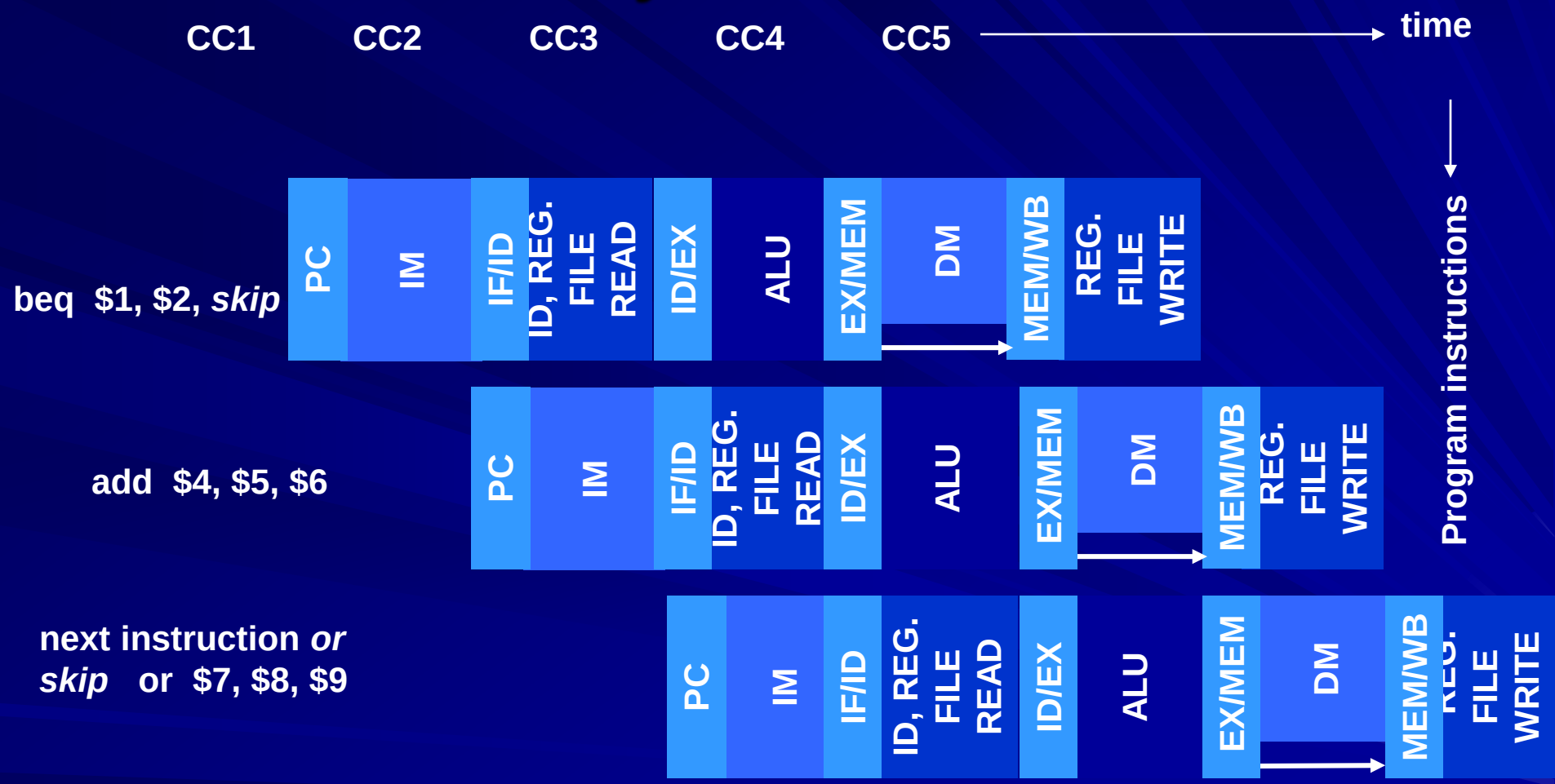
```
        add $4, $5, $6
        beq $1, $2, skip
        next instruction
        . . .
skip    or $7, $8, $9
```

- Delayed branch

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

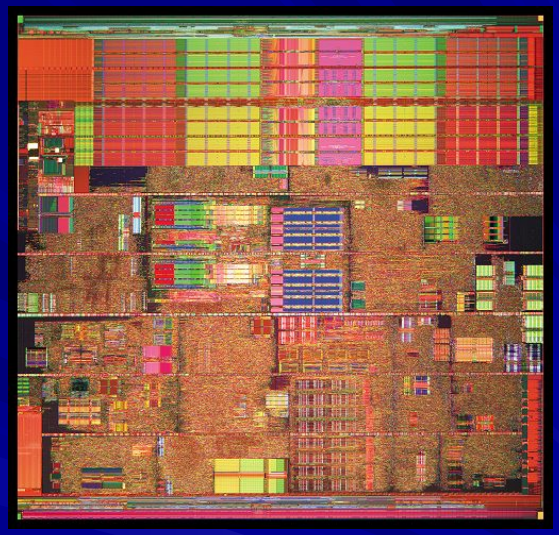
Instruction executed irrespective of branch decision

Delayed Branch



Pipelineing in the Pentium 4

- 20 stage pipeline in (2002) Northwood microarchitecture.



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
TC Nxt IP		TC Fetch		Drive	Alloc	Rename		Que	Sch	Sch	Sch	Disp	Disp	RF	RF	Ex	Flgs	Br Ck	Drive

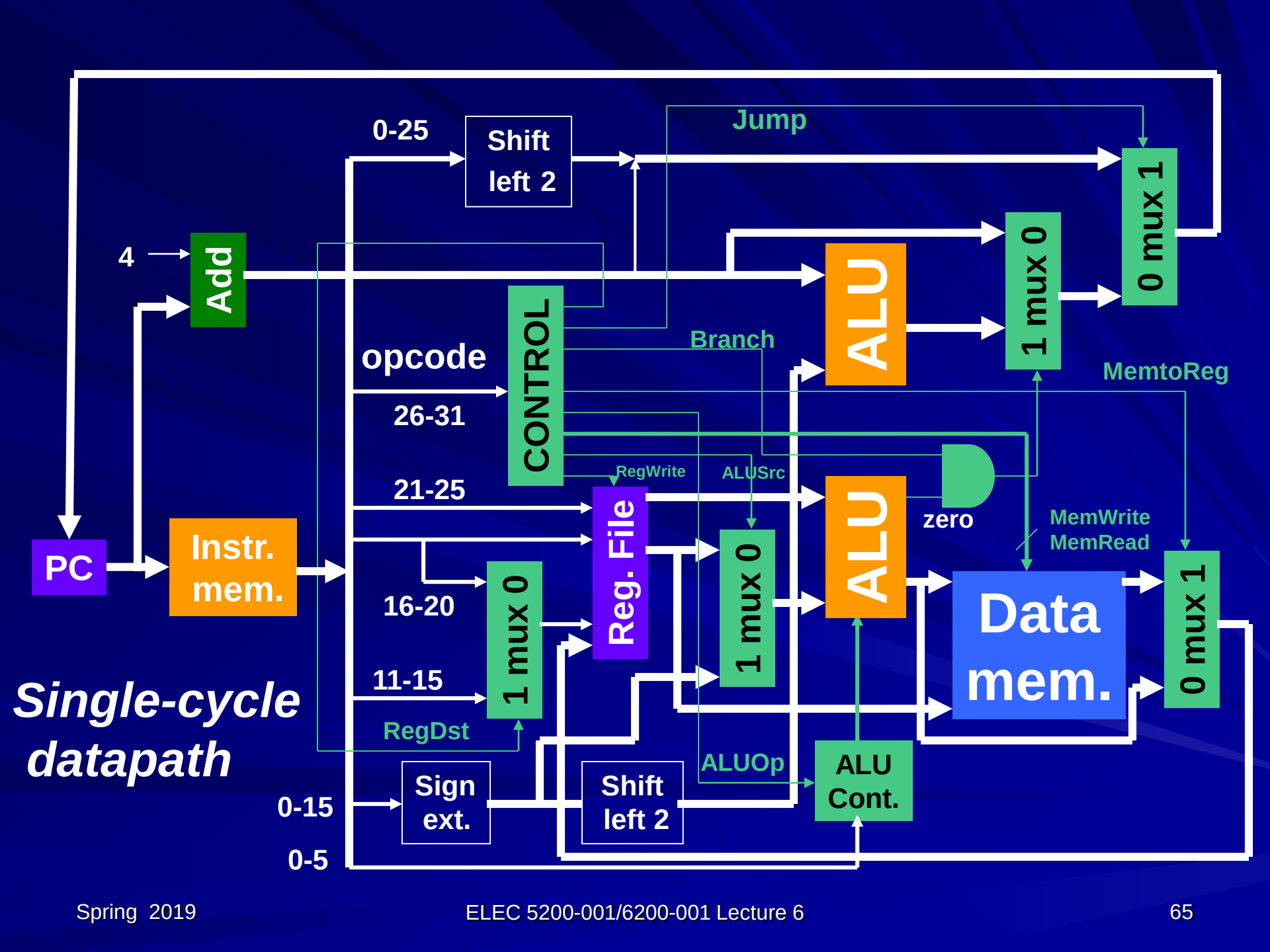
Summary: Hazards

- 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

But Wait, There's More…

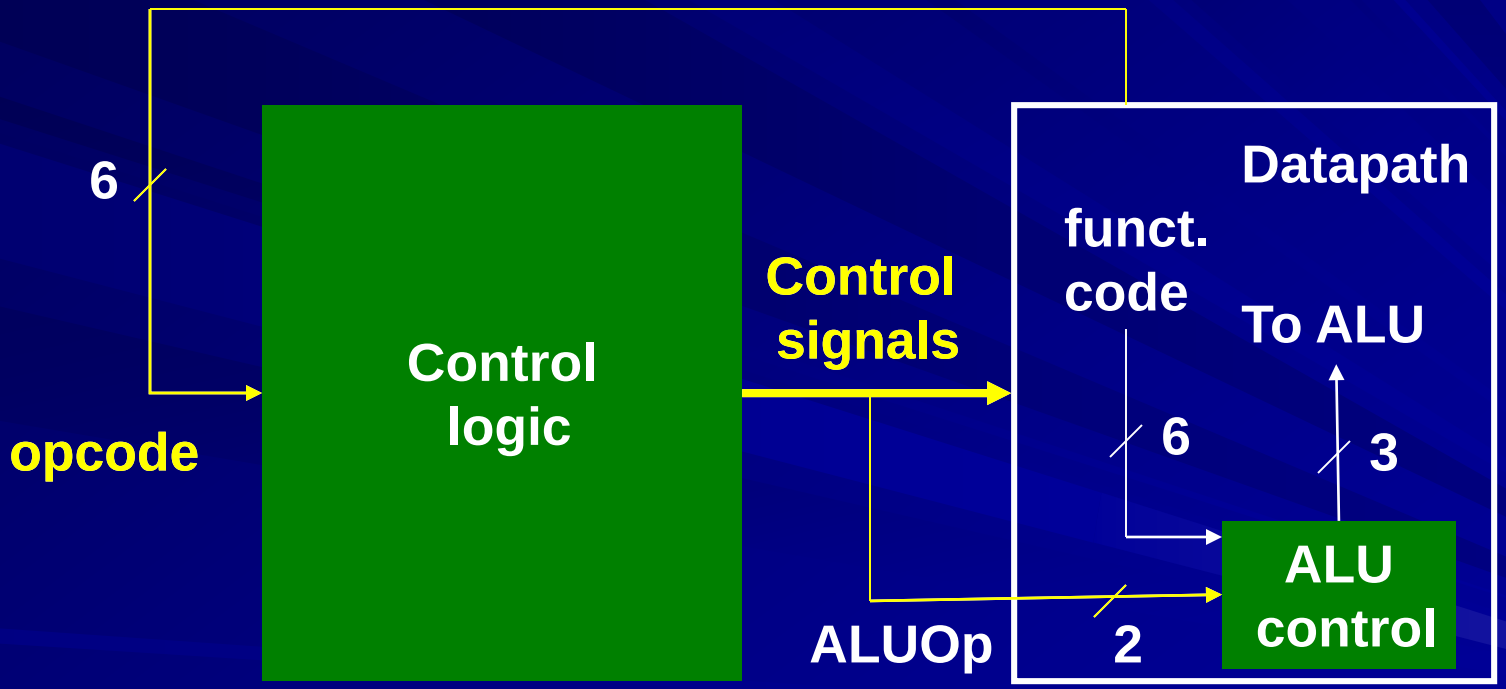
- Processors have two major components, the datapath and…

Control!!!


Single-cycle datapath
PC
Instr. mem.
Add
4
0-25
Shift left 2
Jump
0 mux 1
1 mux 0
ALU
CONTROL
opcode
26-31
Branch
MemtoReg
RegWrite
ALUSrc
21-25
Reg. File
zero
MemWrite
MemRead
16-20
11-15
RegDst
Data mem.
ALUOp
ALU Cont.
Sign ext.
0-15
0-5
Shift left 2

Hardwired CU: Single-Cycle

- Implemented by combinational logic.



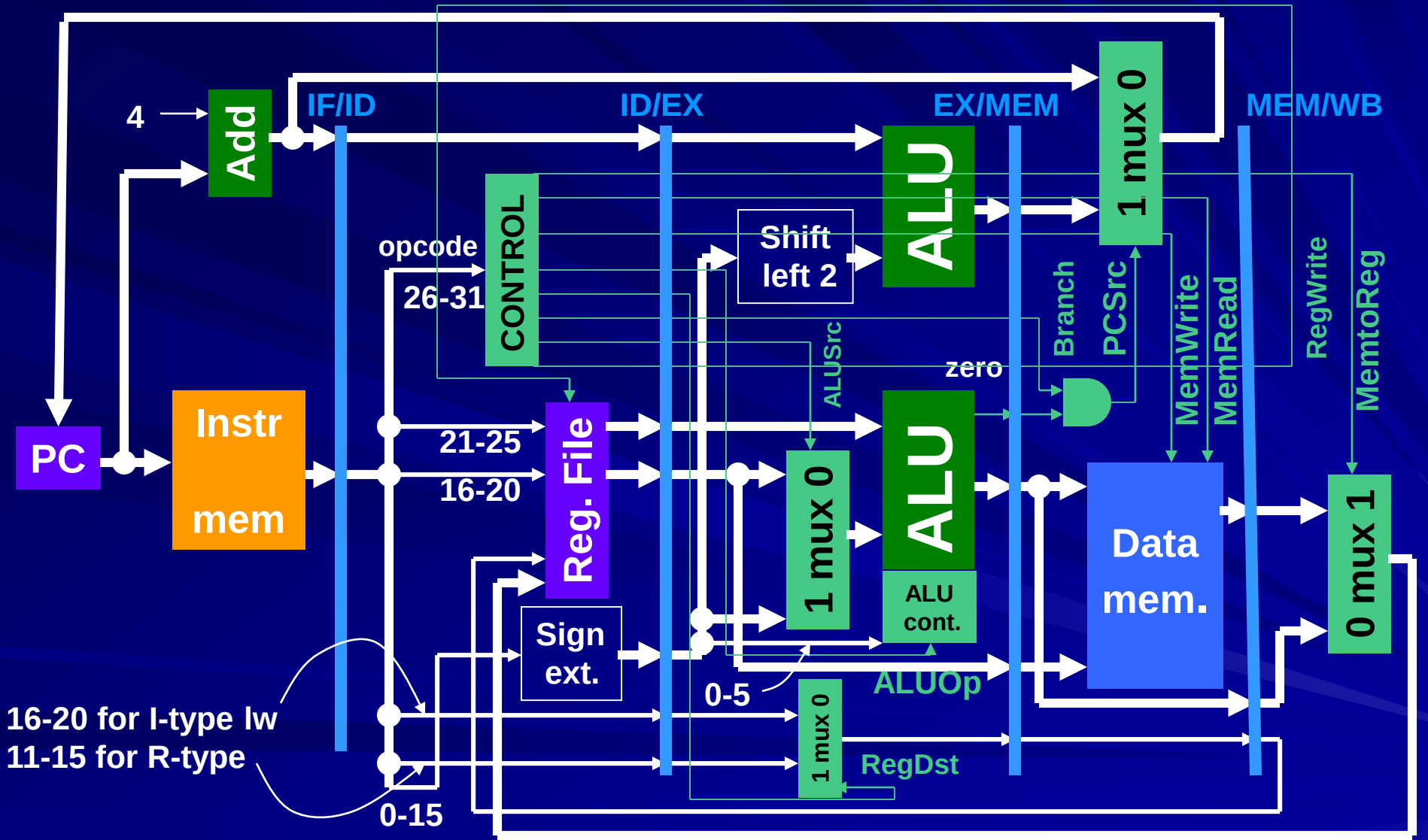
Single-Cycle Control Logic

Inputs							Outputs									
Instr. type	Opcode Instruction bits 31 31 29 28 27 26						RegDst	ALUSrc	MemtoReg	RegWrite	MemRead	MemWrite	Branch	ALUOp1	ALUOp0	Jump
R	0	0	0	0	0	0	1	0	0	1	0	0	0	1	0	0
lw	1	0	0	0	1	1	0	1	1	1	1	0	0	0	0	0
sw	1	0	1	0	1	1	X	1	X	0	0	1	0	0	0	0
beq	0	0	0	1	0	0	X	0	X	0	0	0	1	0	1	0
J	0	0	0	0	1	0	X	X	X	0	X	0	X	X	X	1

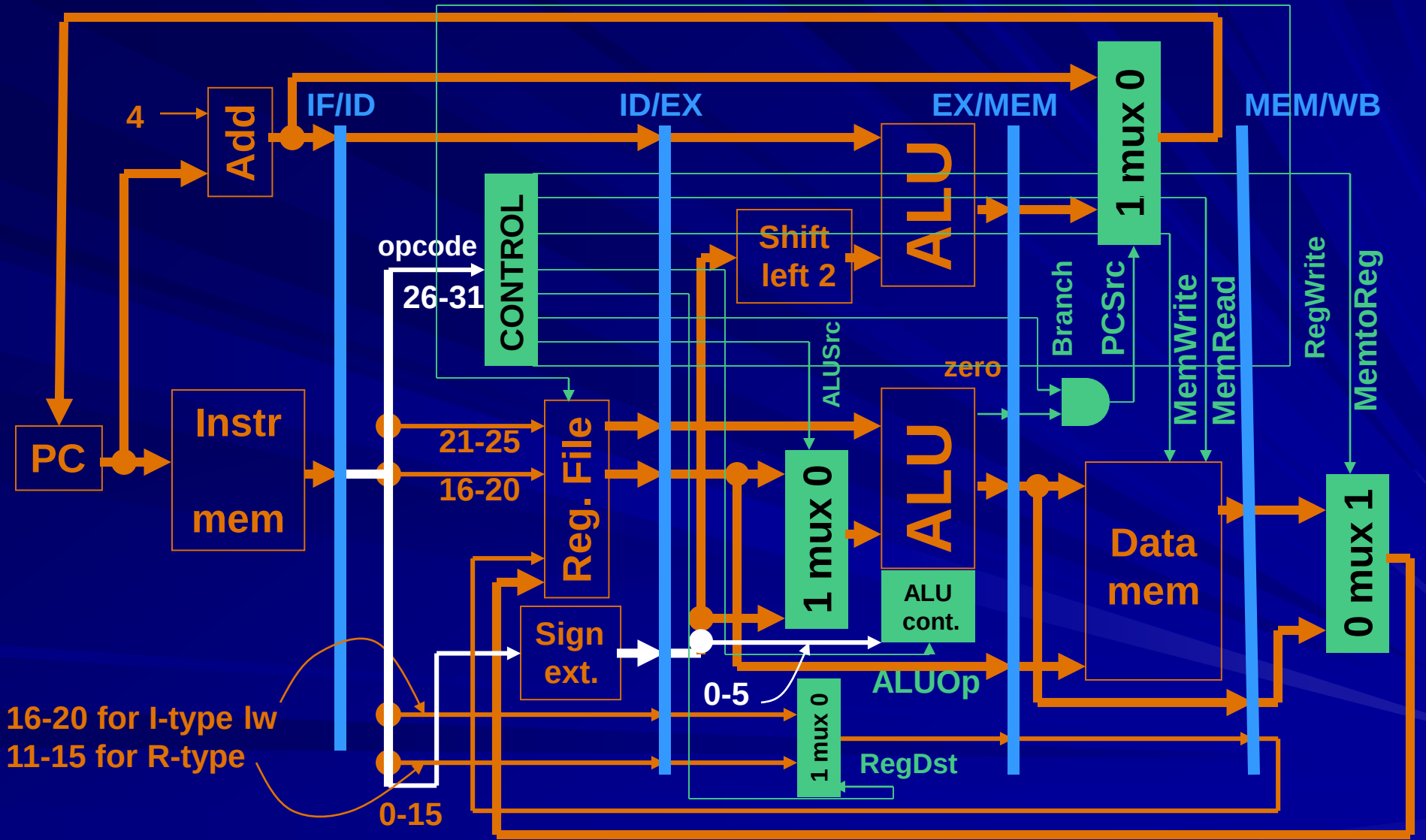
Returning to Pipelined Control

- Opcode input to control is supplied by the pipeline register IF/ID in the ID (instruction decode) cycle.
- Nine control signals are generated in the ID cycle, but none is used. They are saved in the pipeline register ID/EX.
- ALUSrc, RegDst and ALUOp (2 bits) are used in the EX (execute) cycle. Remaining 5 control signals are saved in the pipeline register EX/MEM.
- Branch, MemWrite and MemRead are used in the MEM (memory access) cycle. Remaining 2 control signals are saved in the pipeline register MEM/WB.
- MemtoReg and RegWrite are used in the WB (write back) cycle.
- Pipelined control is shown without Jump.

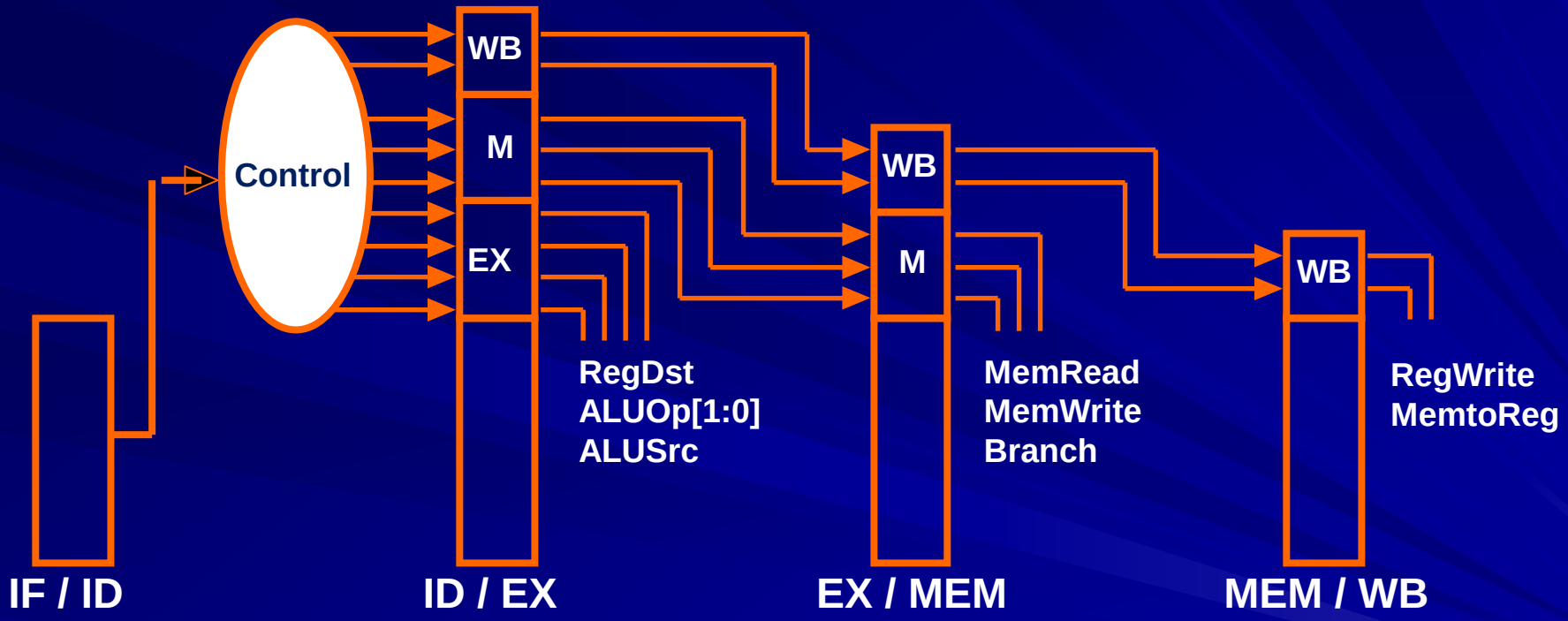
Placing Control in Pipelined Datapath



Highlighted Pipelined Control

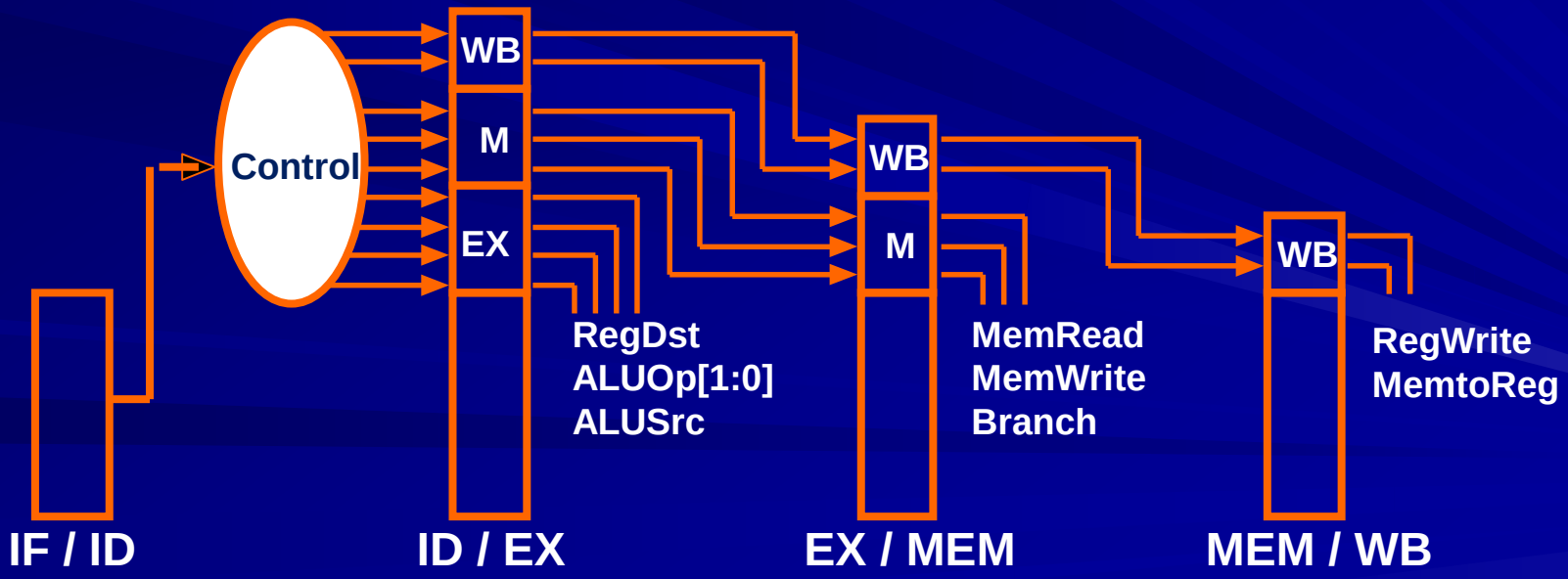


Control For Pipelined Datapath

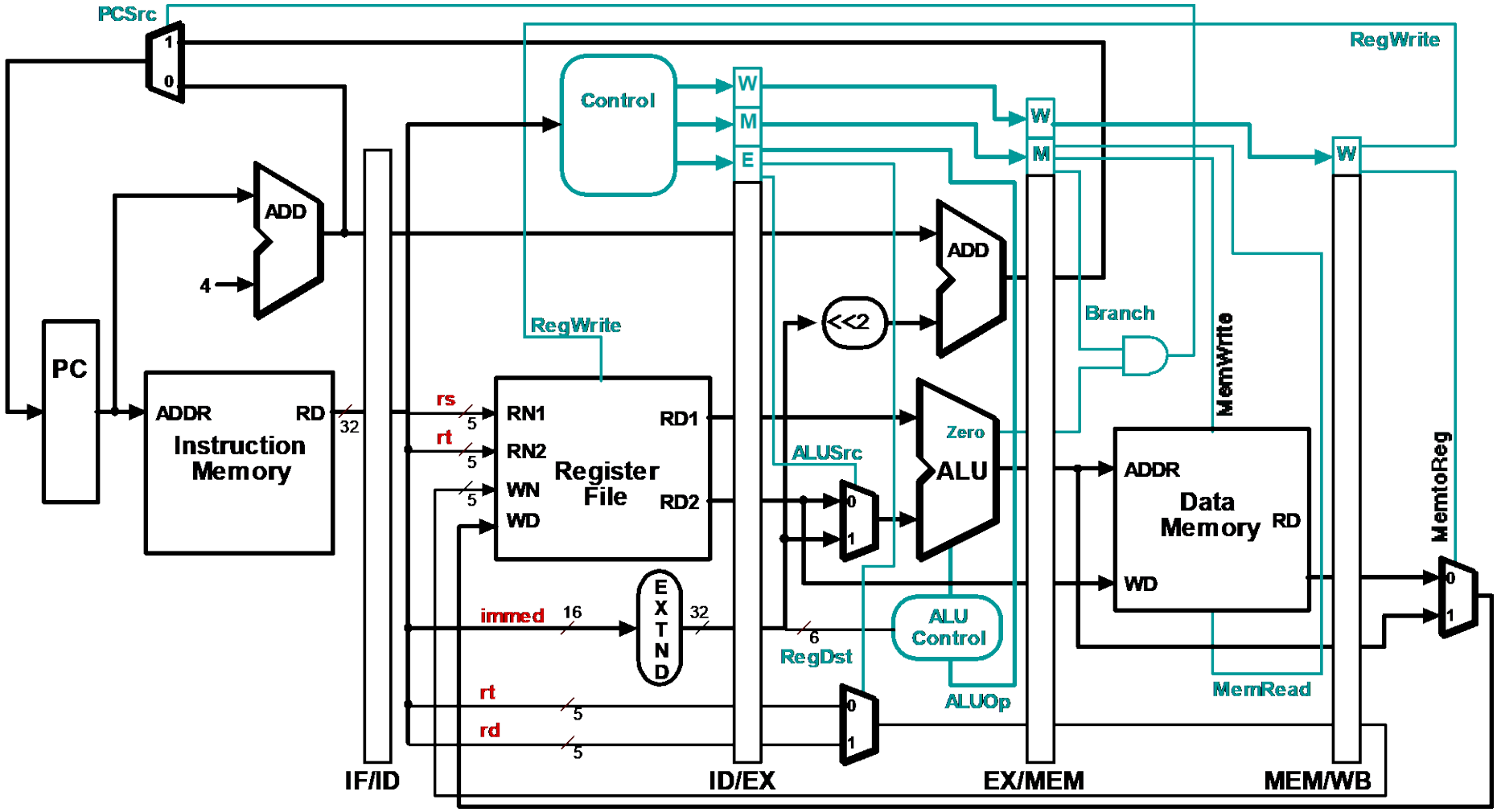


Control For Pipelined Datapath

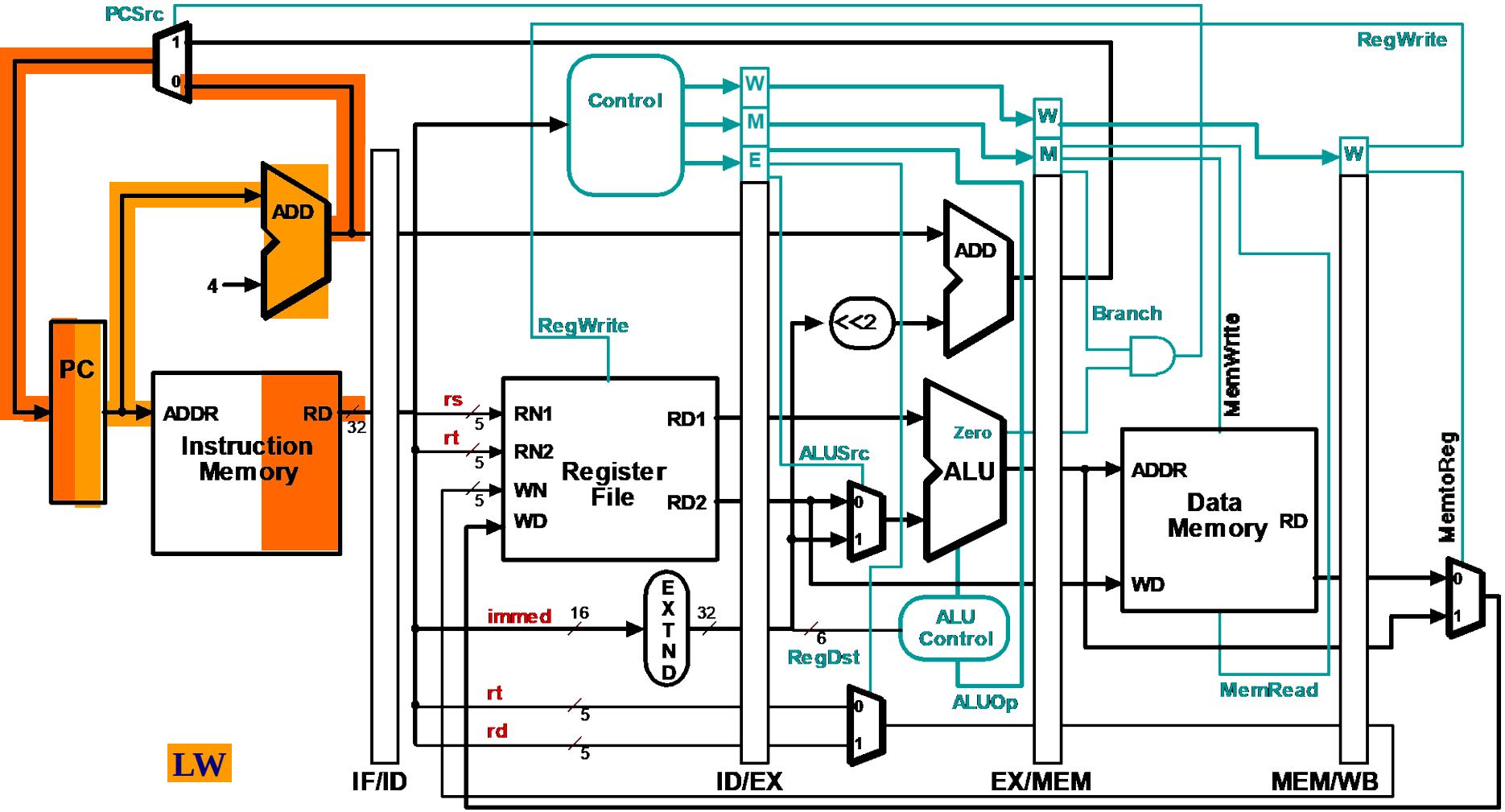
	Execution/Address Calculation stage control lines				Memory access stage control lines			Write-back stage control lines	
Instruction	Reg Dst	ALU Op1	ALU Op0	ALU Src	Branch	Mem Read	Mem Write	Reg write	Mem to Reg
R-format	1	1	0	0	0	0	0	1	0
lw	0	0	0	1	0	1	0	1	1
sw	X	0	0	1	0	0	1	0	X
beq	X	0	1	0	1	0	0	0	X



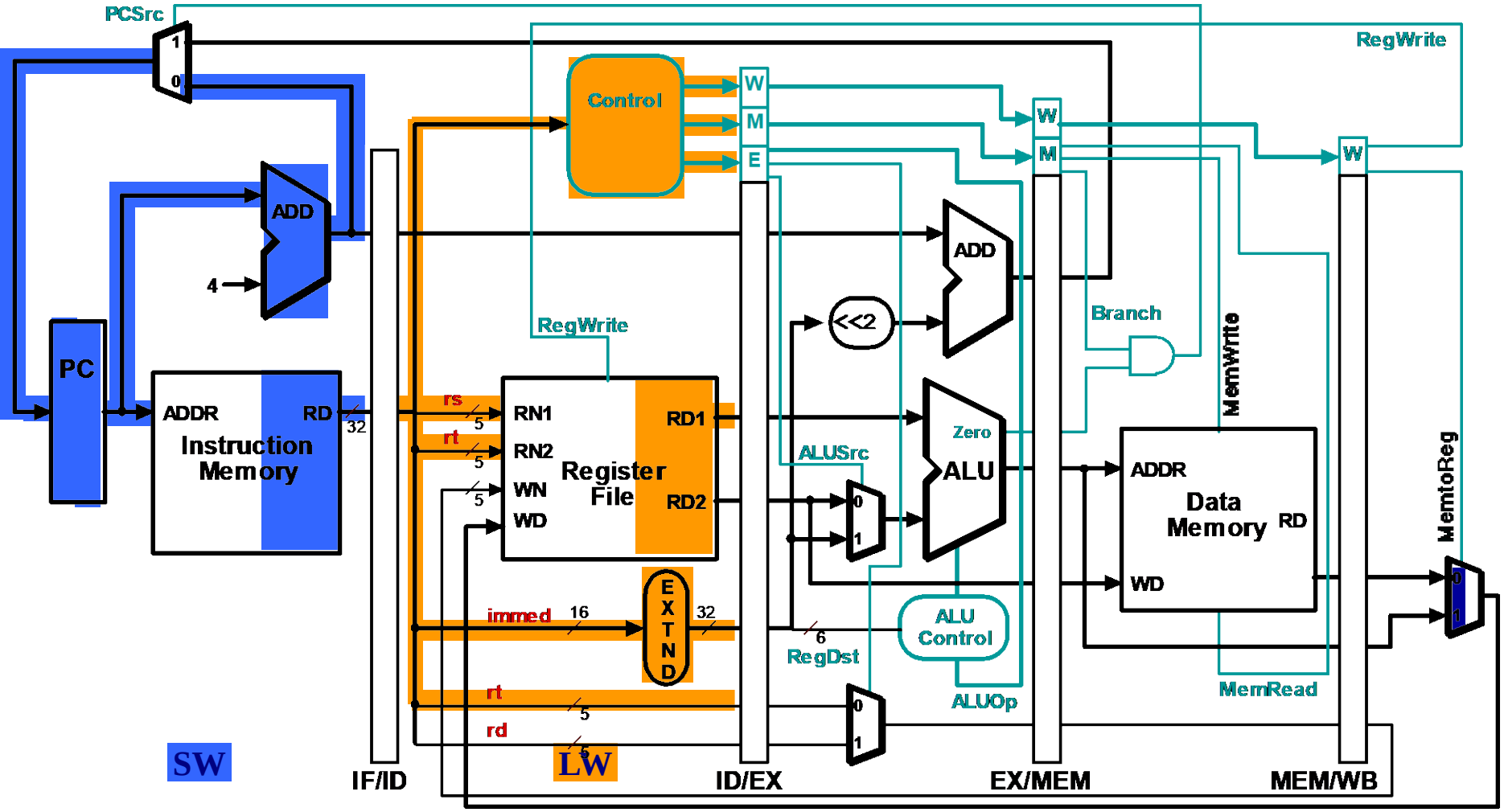
Datapath and Control Unit



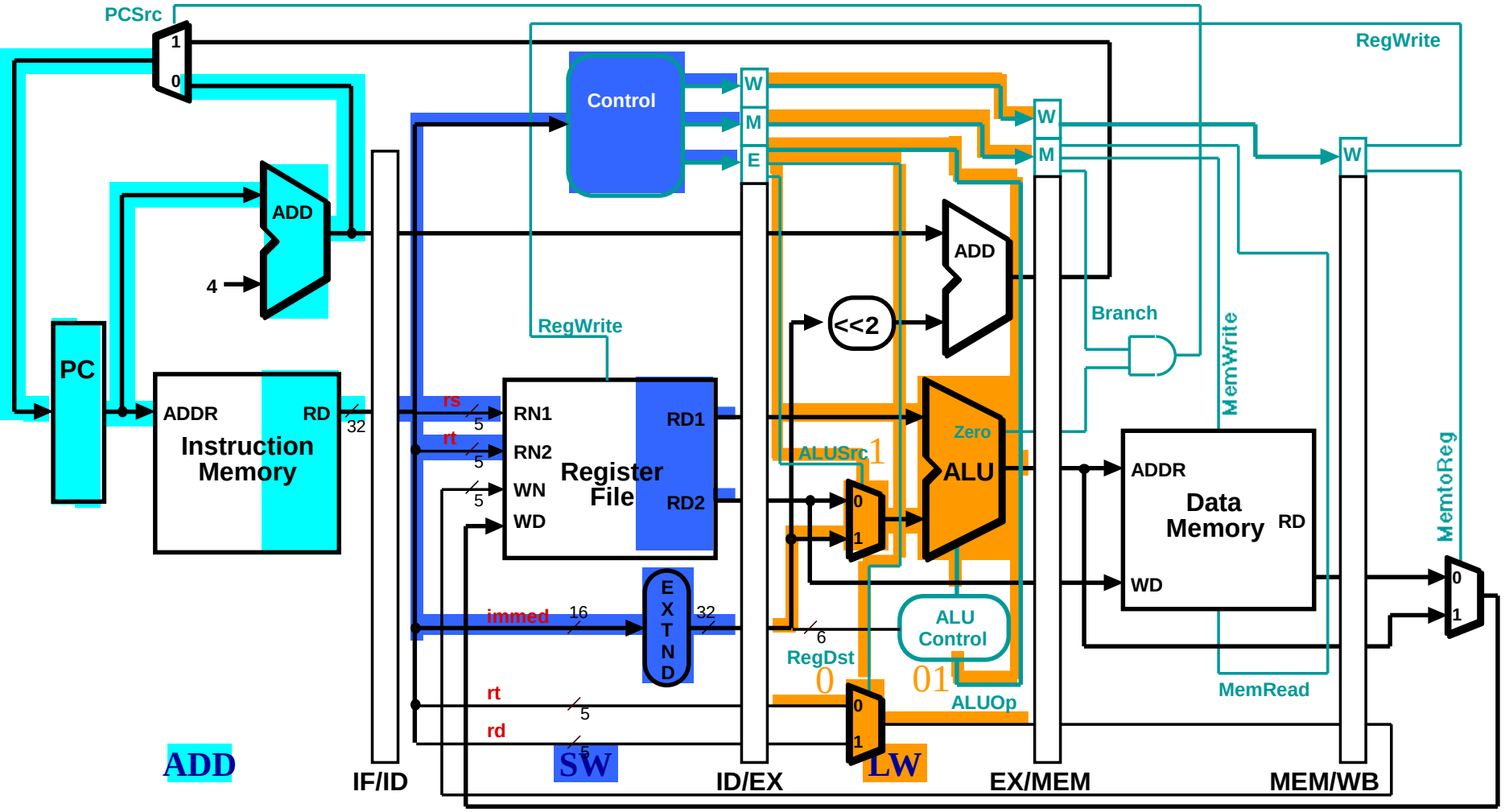
Tracking Control Signals – Cycle 1



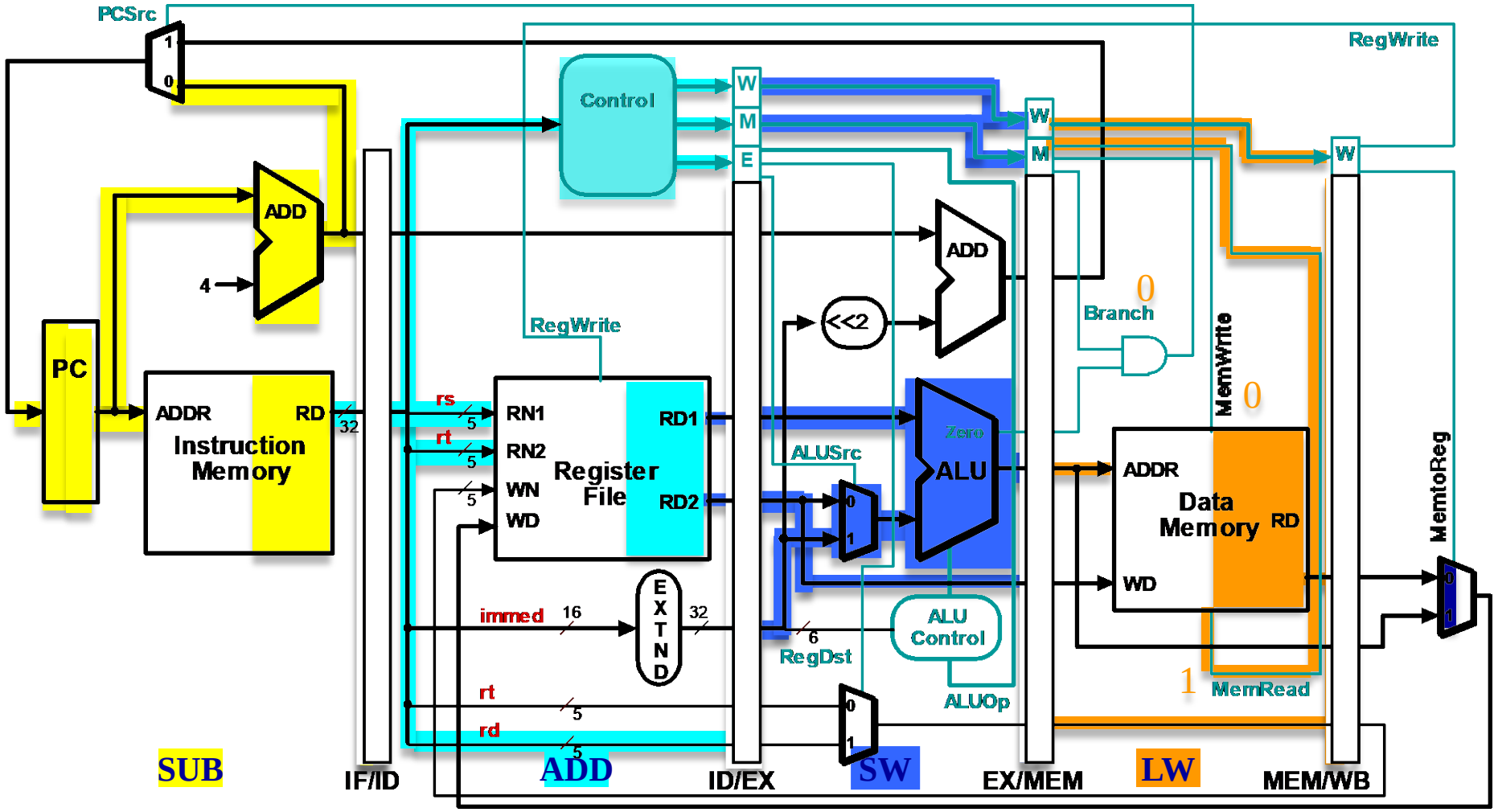
Tracking Control Signals – Cycle 2



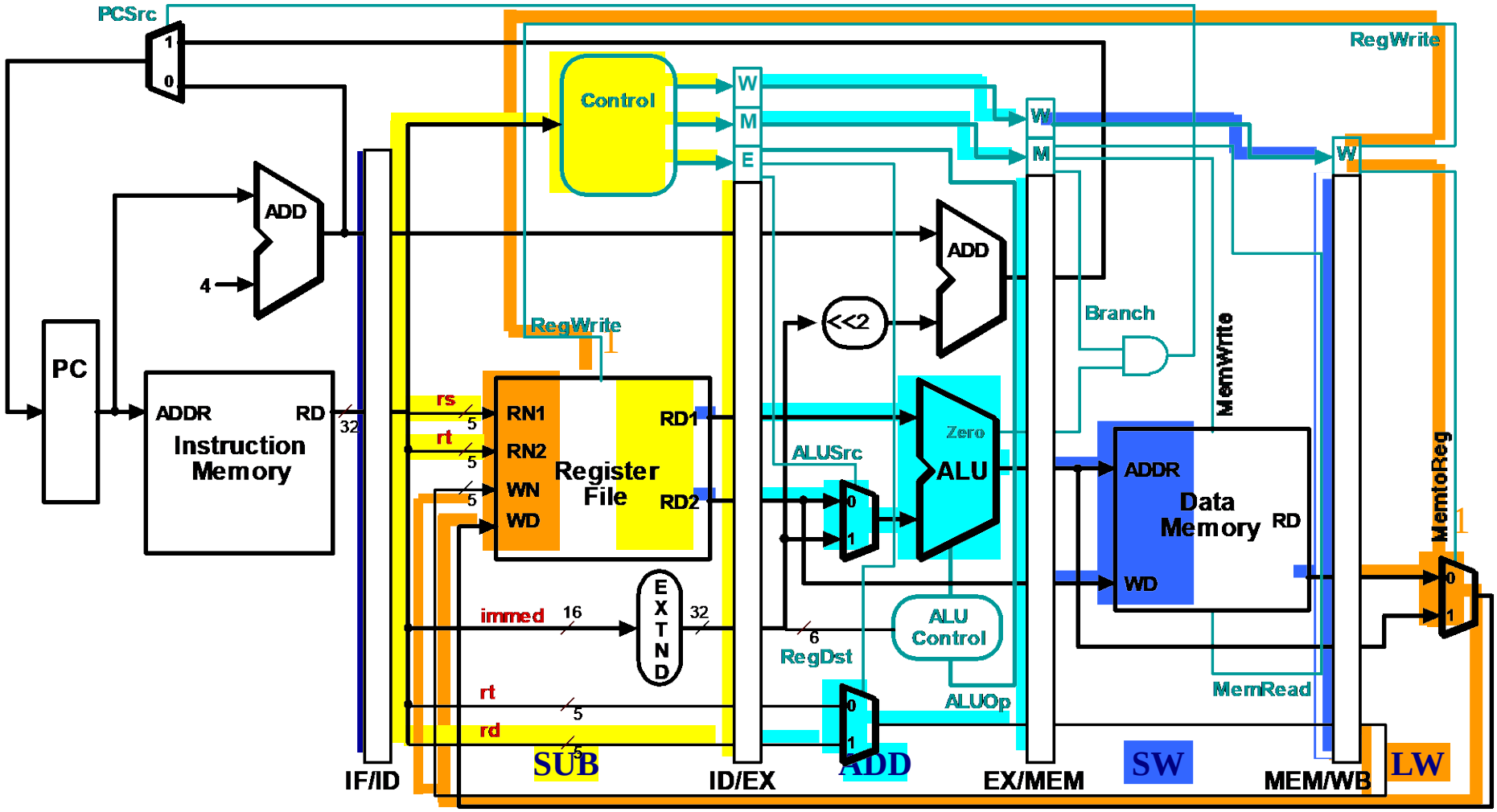
Tracking Control Signals – Cycle 3



Tracking Control Signals – Cycle 4

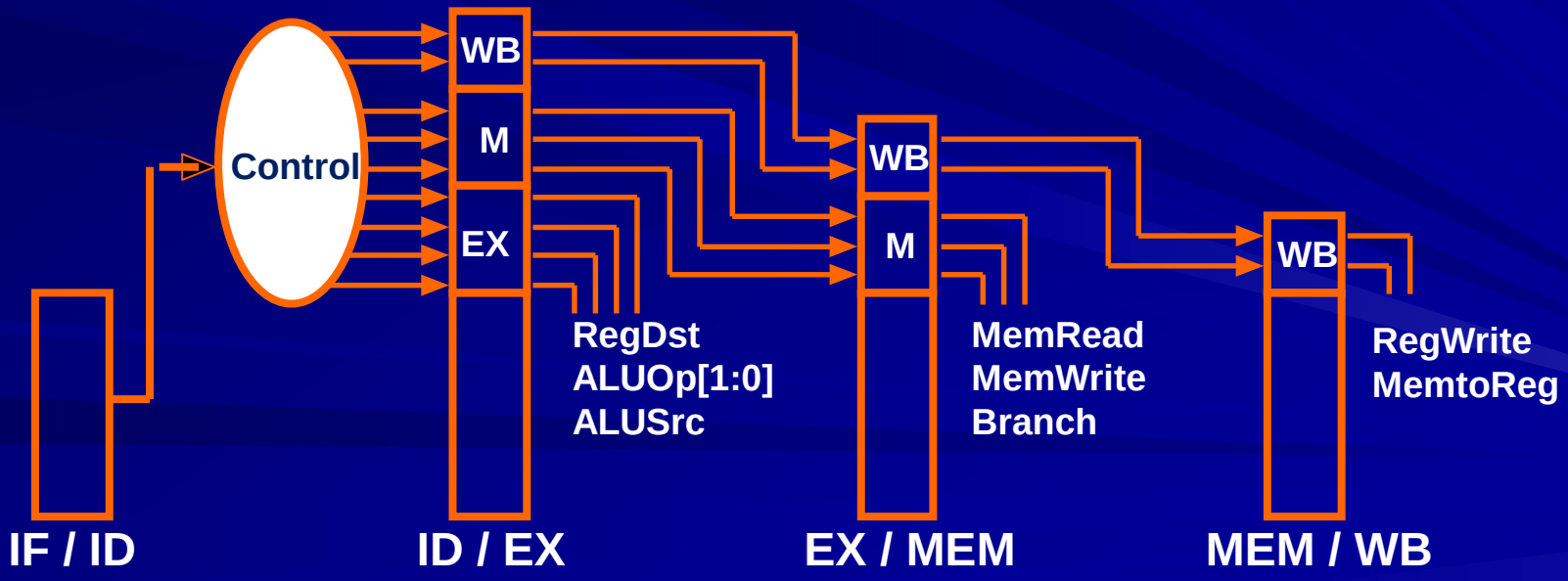


Tracking Control Signals – Cycle 5



Control For Pipelined Datapath

	Execution/Address Calculation stage control lines				Memory access stage control lines			Write-back stage control lines	
Instruction	Reg Dst	ALU Op1	ALU Op0	ALU Src	Branch	Mem Read	Mem Write	Reg write	Mem to Reg
R-format	1	1	0	0	0	0	0	1	0
lw	0	0	0	1	0	1	0	1	1
sw	X	0	0	1	0	0	1	0	X
beq	X	0	1	0	1	0	0	0	X



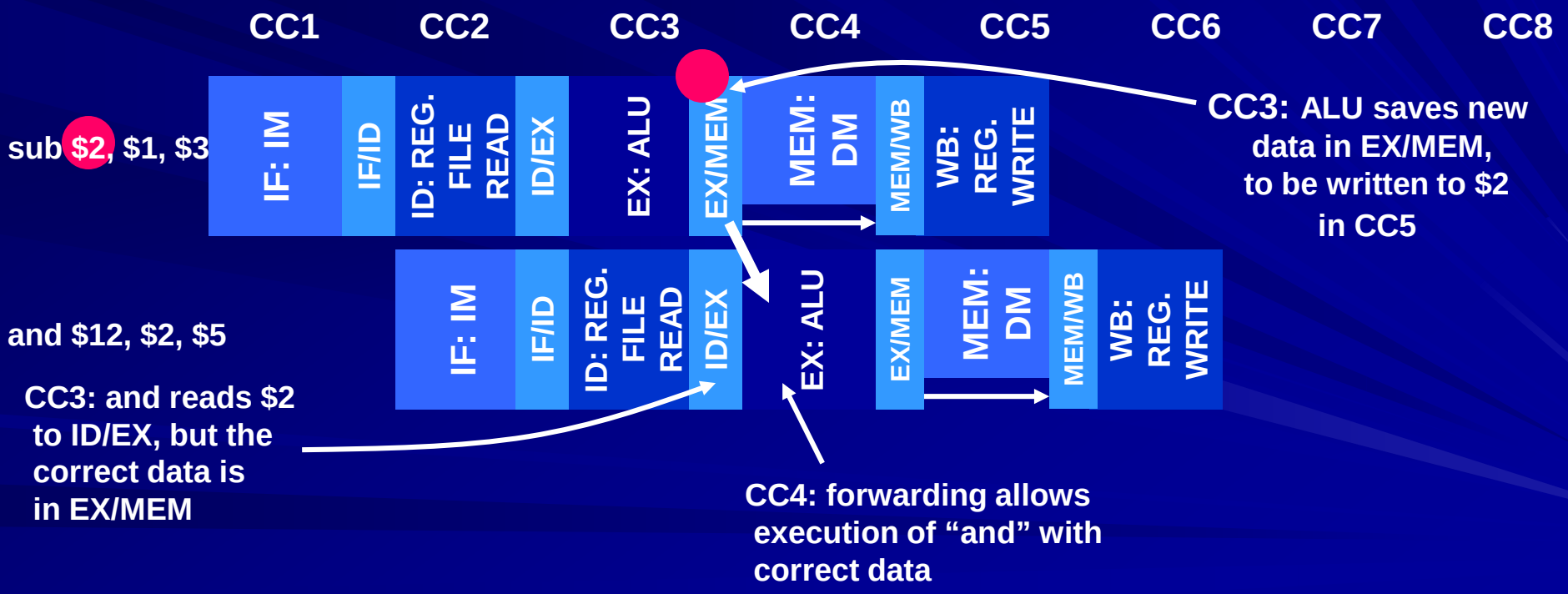
Other Controls for Pipeline

- We must add control for:
  - Forwarding
  - Stall / bubbles
  - Branch hazard and branch prediction
  - Instruction flush
  - Exceptions

Forwarding

- Consider a data hazard:

```
sub  $2, $1, $3      # computes result in CC3, writes in $2 in CC5
and  $12, $2, $5     # reads $2 in CC3, adds in CC4
```



Understanding Forwarding

- Let's ask following questions:

  Q: Why is there a hazard?

  A: Source register for the present instruction is the same as the destination register of the previous instruction.

  Q: When is the source register data needed?

  A: In the execute cycle (CC4).

  Q: Is source register data available in CC4?

  A: Yes – use forwarding. No – use stall.

  Q: Where is the required data in CC4?

  A: In the pipeline register EX/MEM as ALU output.

Forwarding Hardware

- A forwarding unit is added to execute (ALU) cycle hardware.
- Functions of forwarding unit:
  - Hazard detection.
  - Forward correct data to ALU.
- Inputs to forwarding unit:
  - Source registers of the instruction in EX.
  - Destination registers of instructions in DM and WB.
- Outputs of forwarding unit:
  - Multiplexer controls to route correct data to the ALU.

Recall Register Definitions

- R-type instruction (add, sub, and, or, . . .)

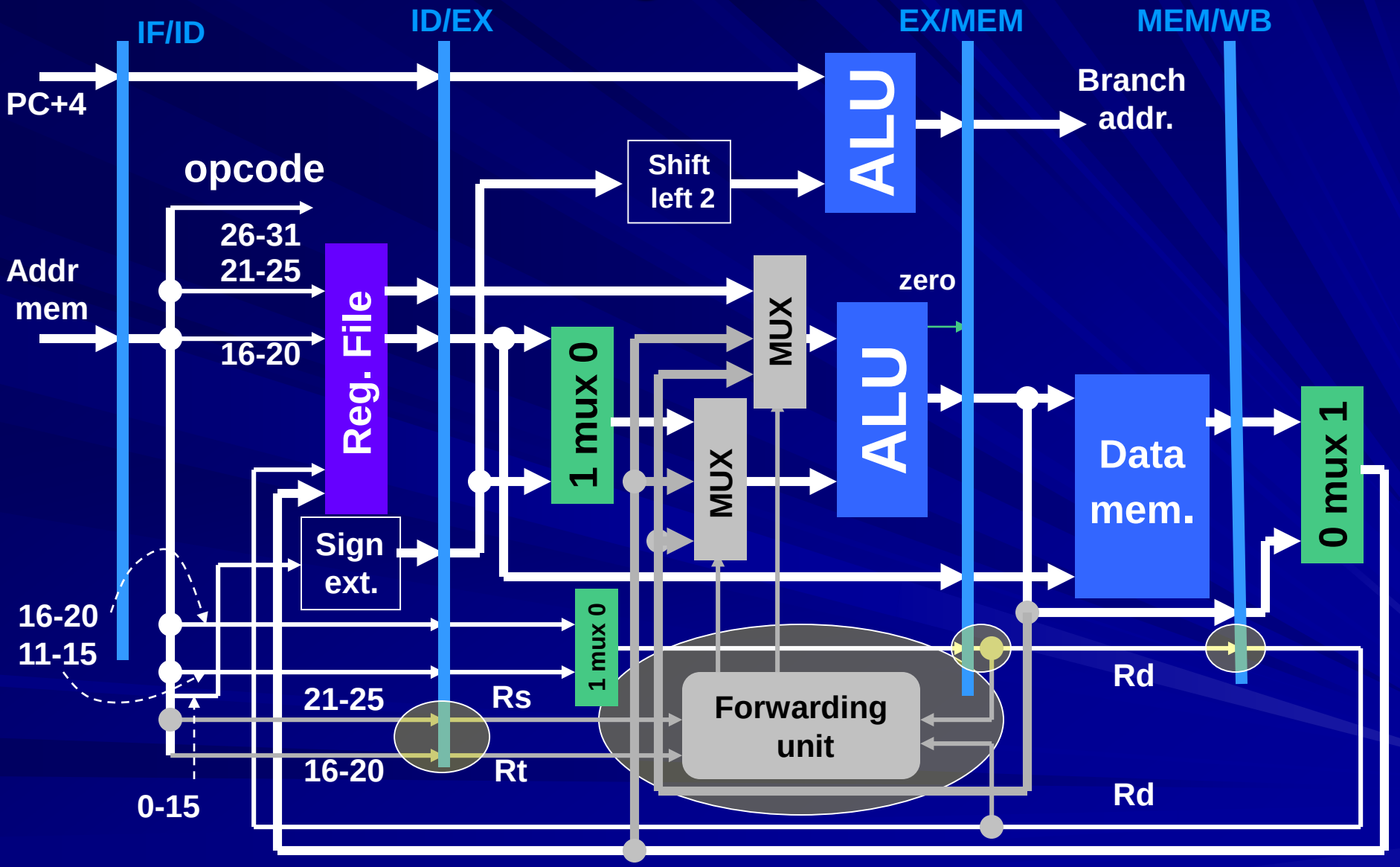
  opcode Rs Rt Rd shamt funct
- I-type instruction (beq, lw, sw, addi, . . .)

  opcode Rs Rt constant_or_address
- J-type instruction (j, jal, jr)

  opcode a___d___d___r___e___s___s

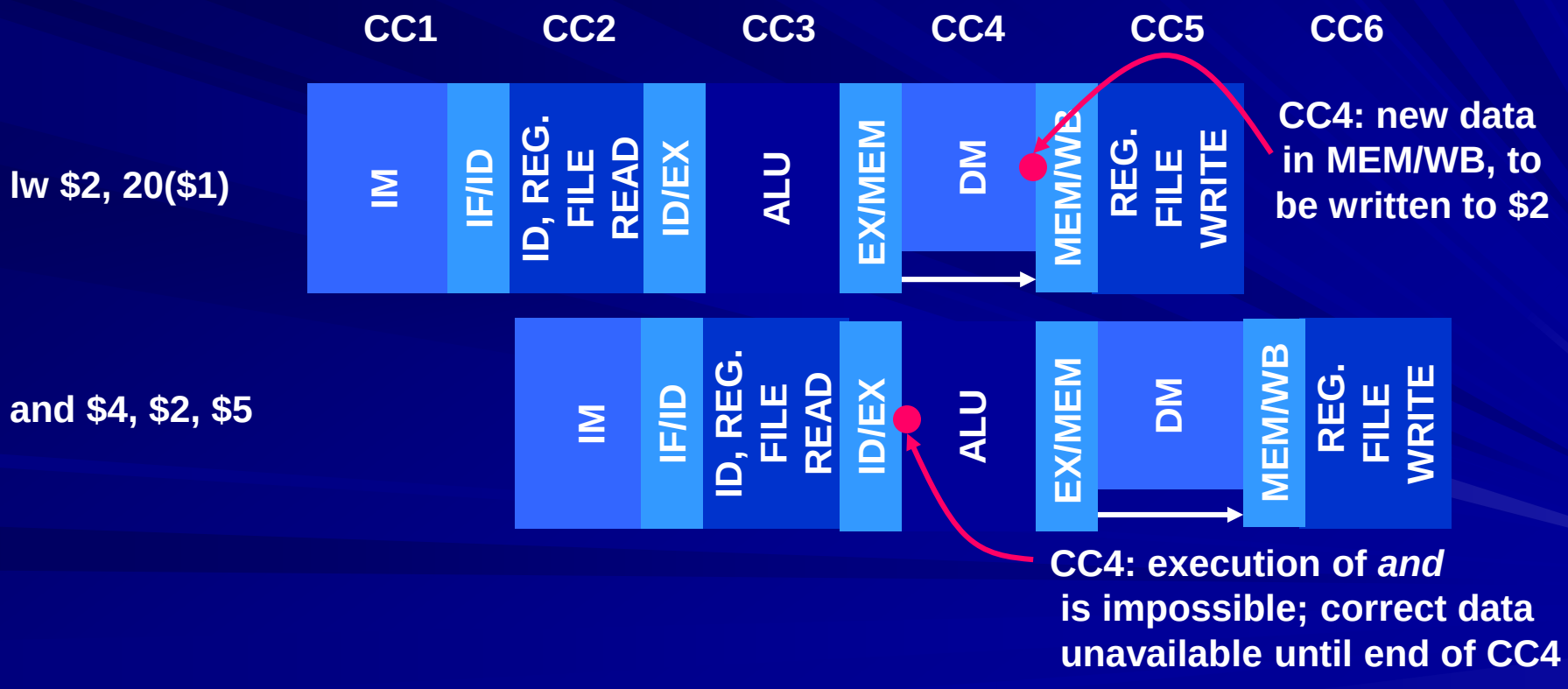
  where
  - Rs is the first source register
  - Rt is the second source register
  - Rd is the destination register

Forwarding Implemented



Stall

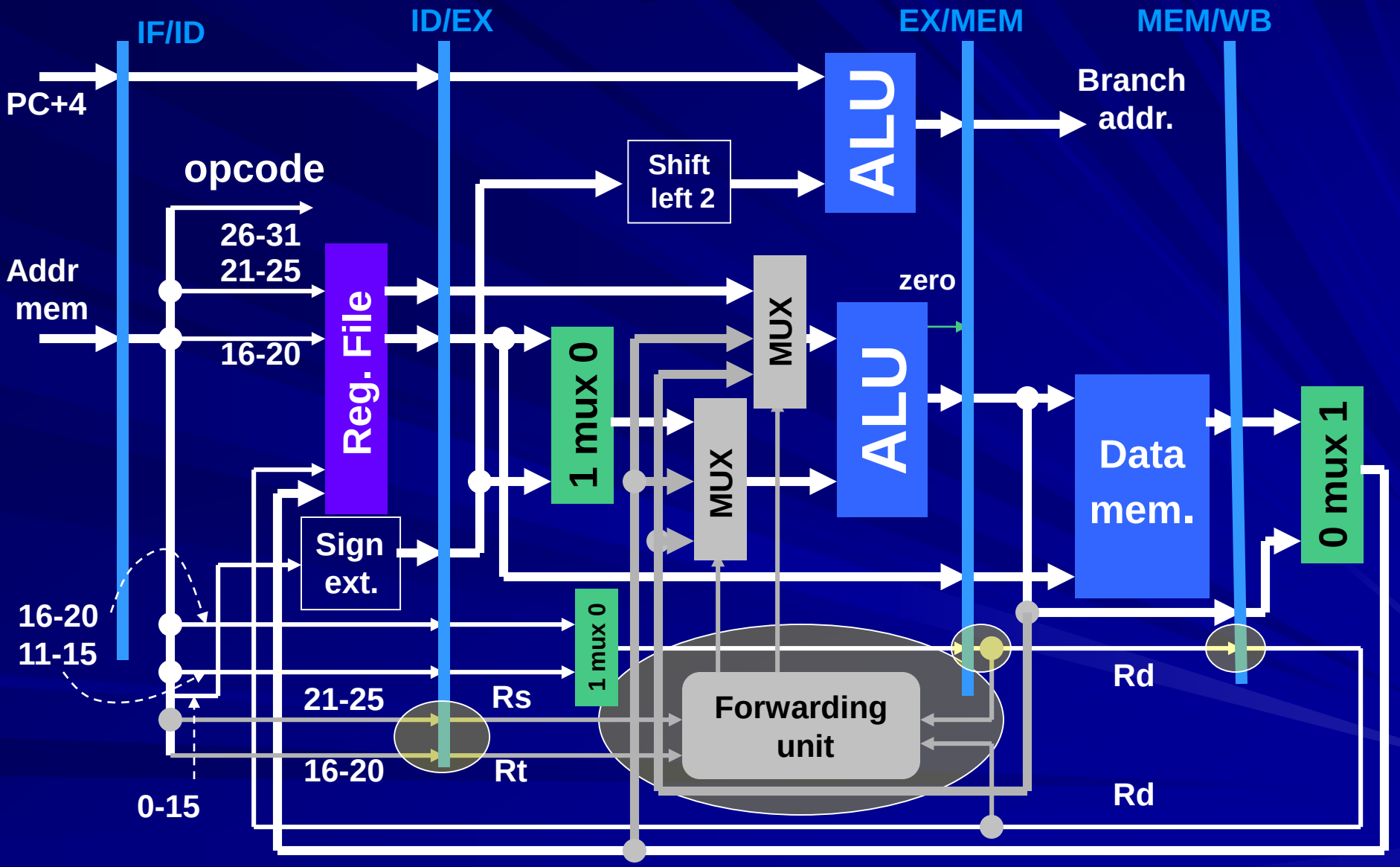
- Delay next instruction by sending no-op through pipeline.
- Necessary when hazard not resolved by forwarding.



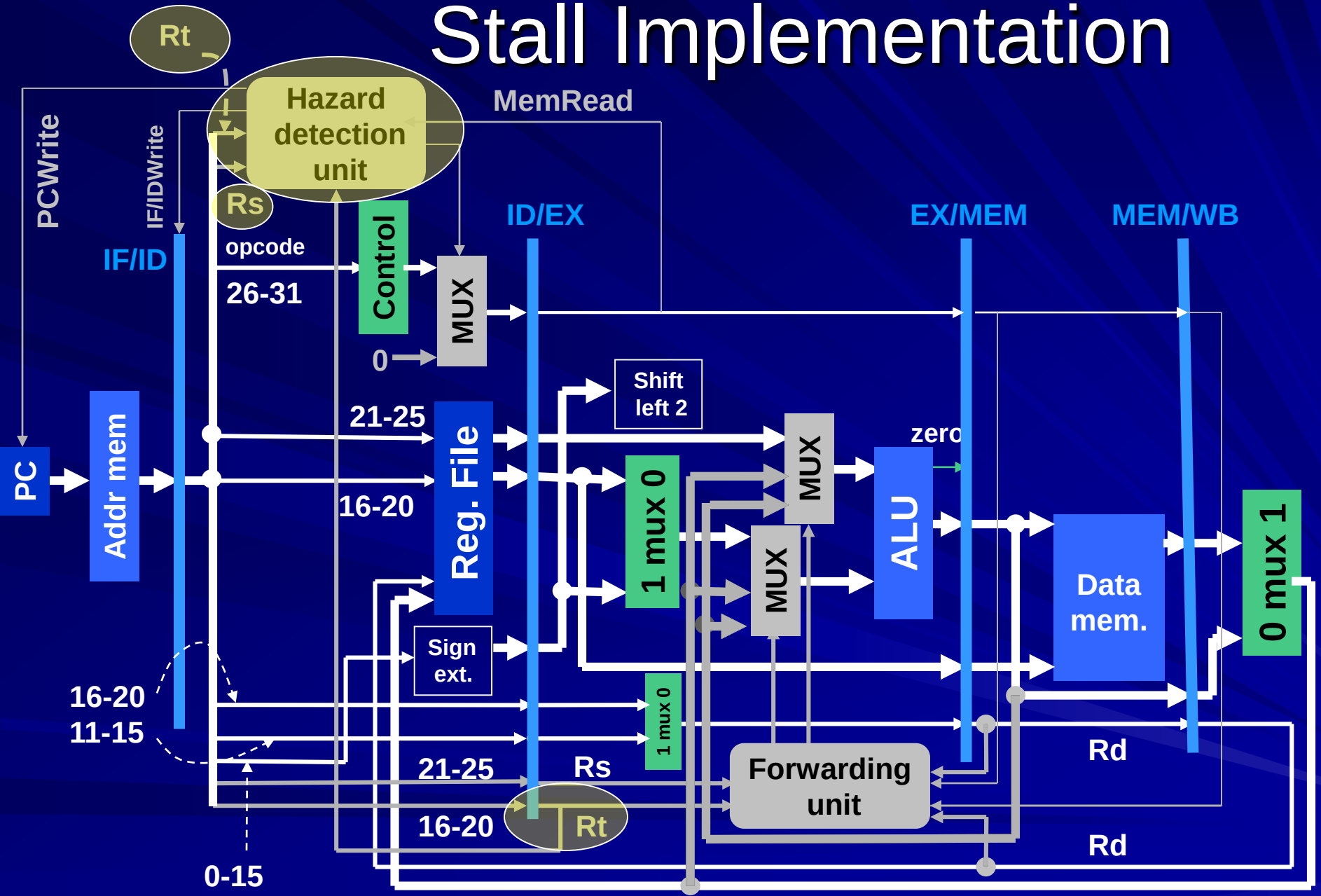
Detecting Hazard Requiring Stall

- Consider instruction in IF/ID being decoded:
- *If…*
  - Previous instruction (lw) activated MemRead, and
  - Instruction being decoded has a source register (Rs or Rt) same as the destination register (Rt for lw) of the previous instruction
- *Then*… stall the pipeline:
  - Force all control outputs to 0
  - Prevent PC from changing
  - Prevent IF/ID from changing

Forwarding Implemented

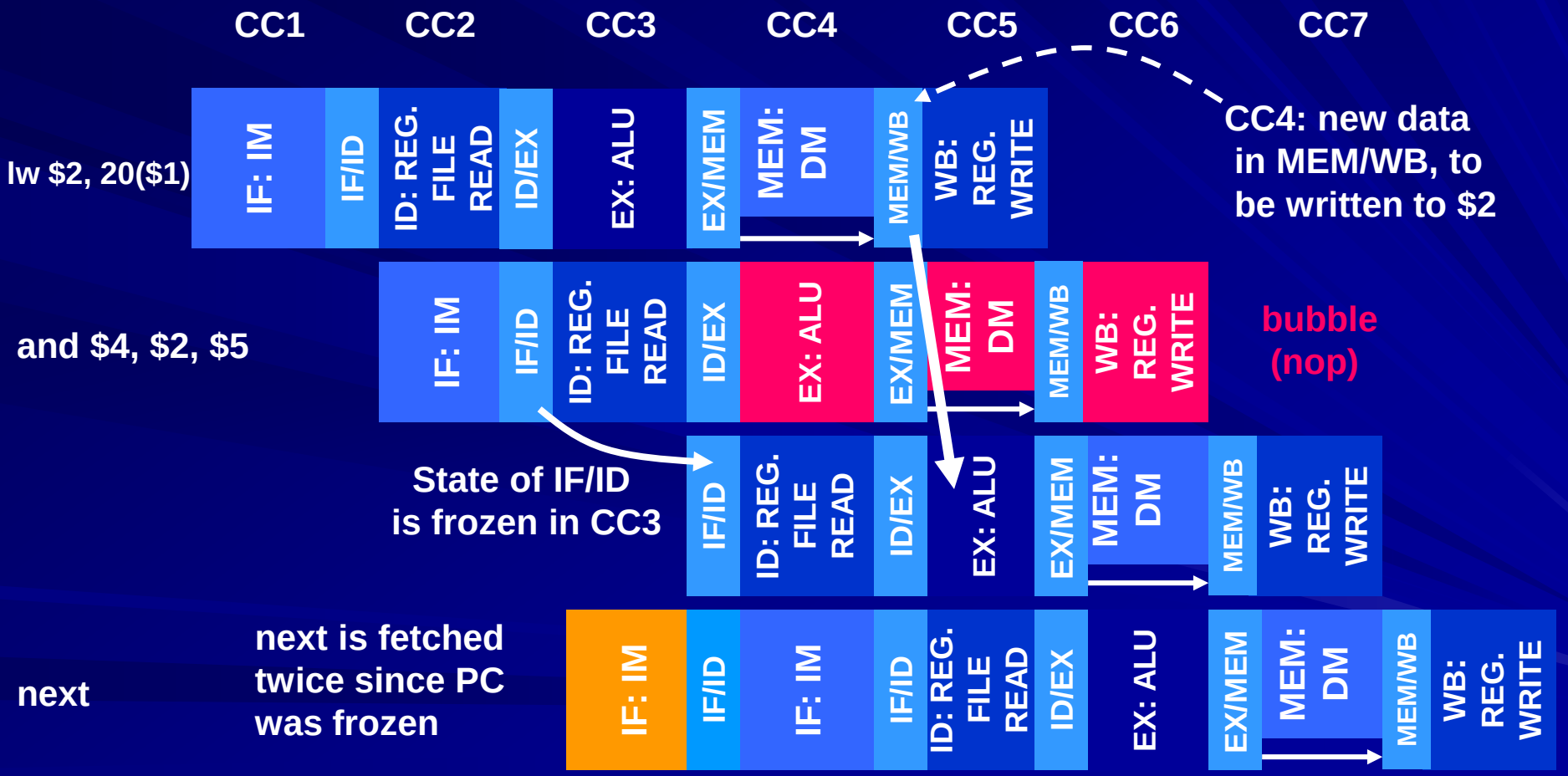


Stall Implementation



Stall

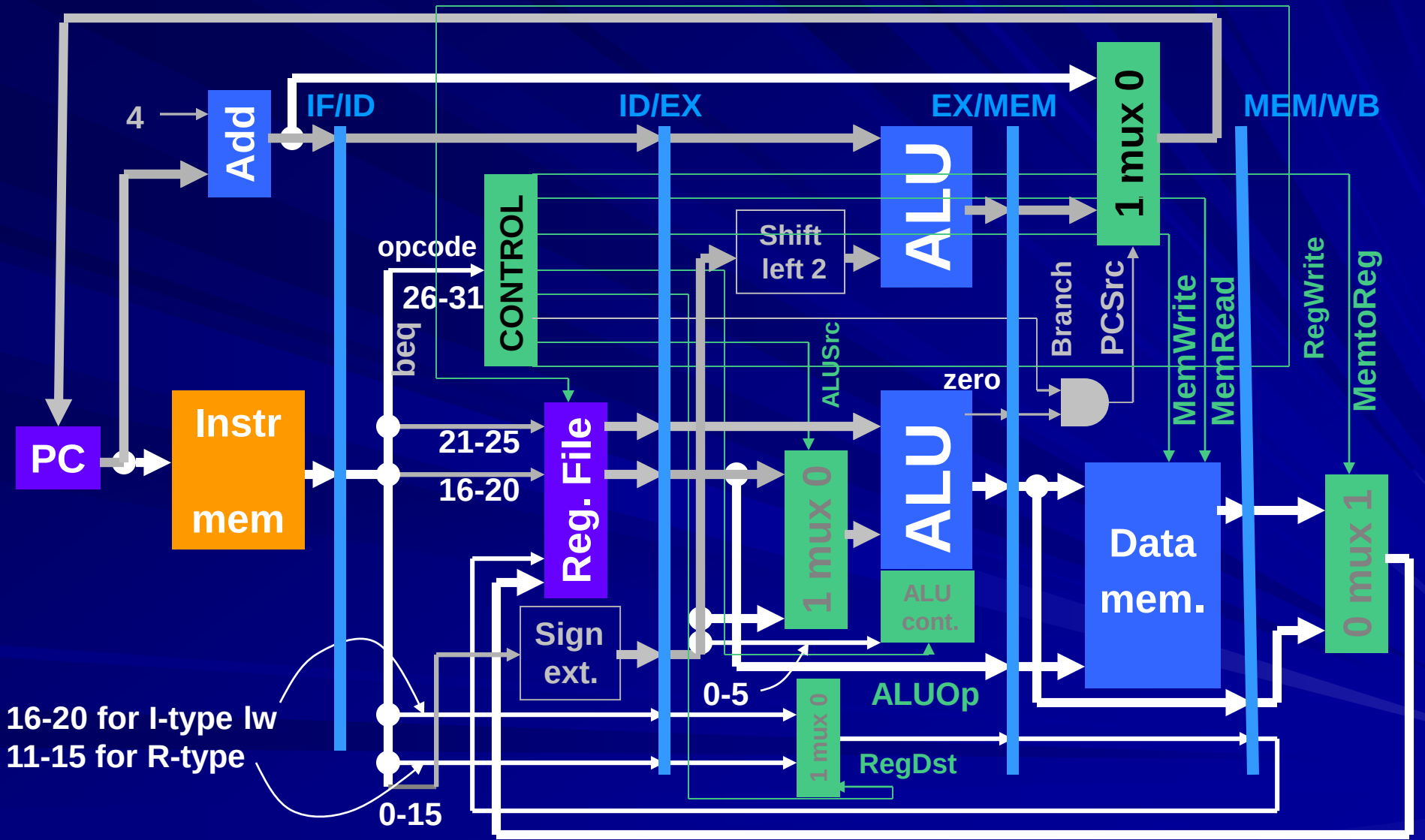
- Execution with stall and forwarding:



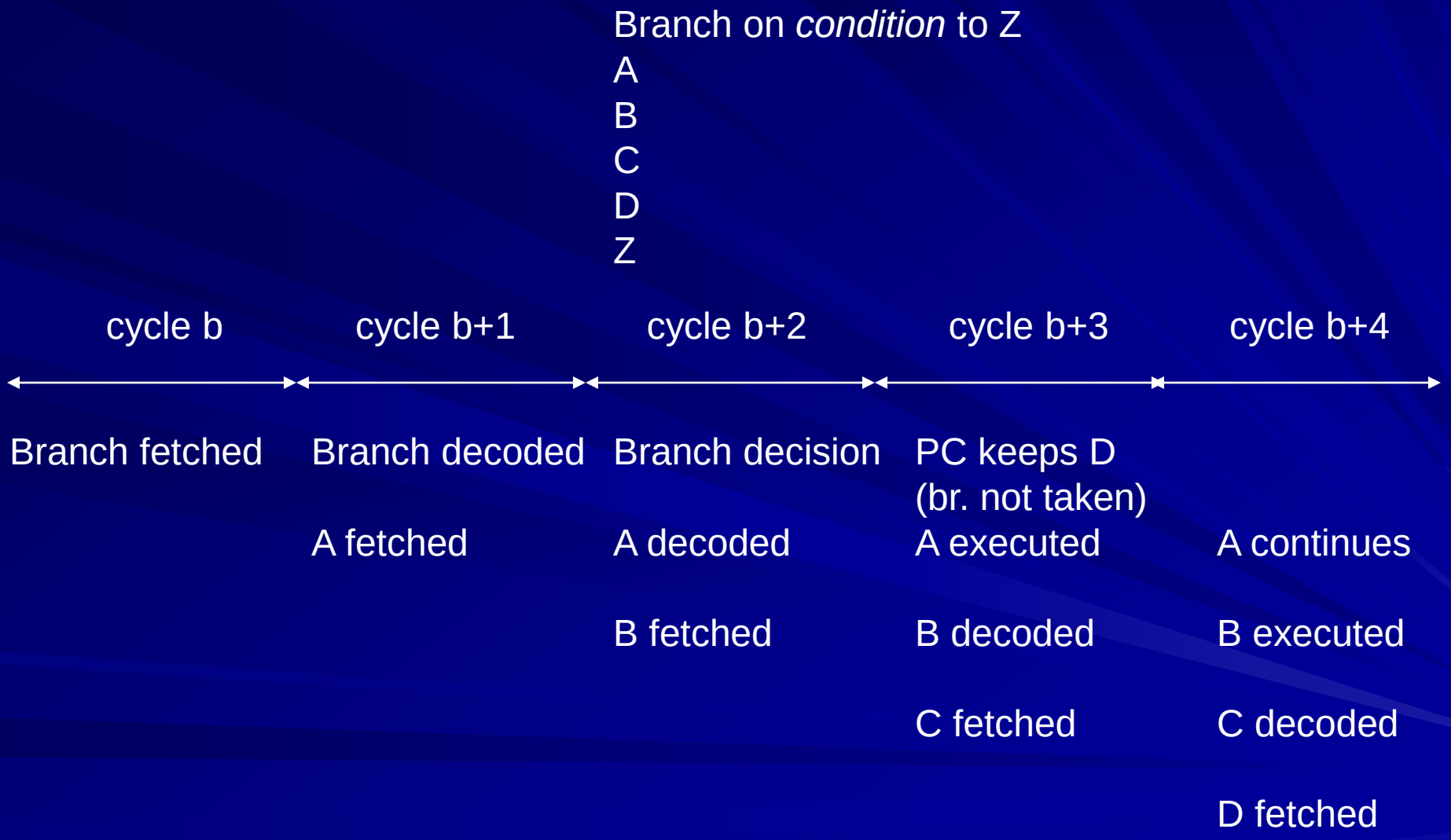
Branch Hazard

- 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?*

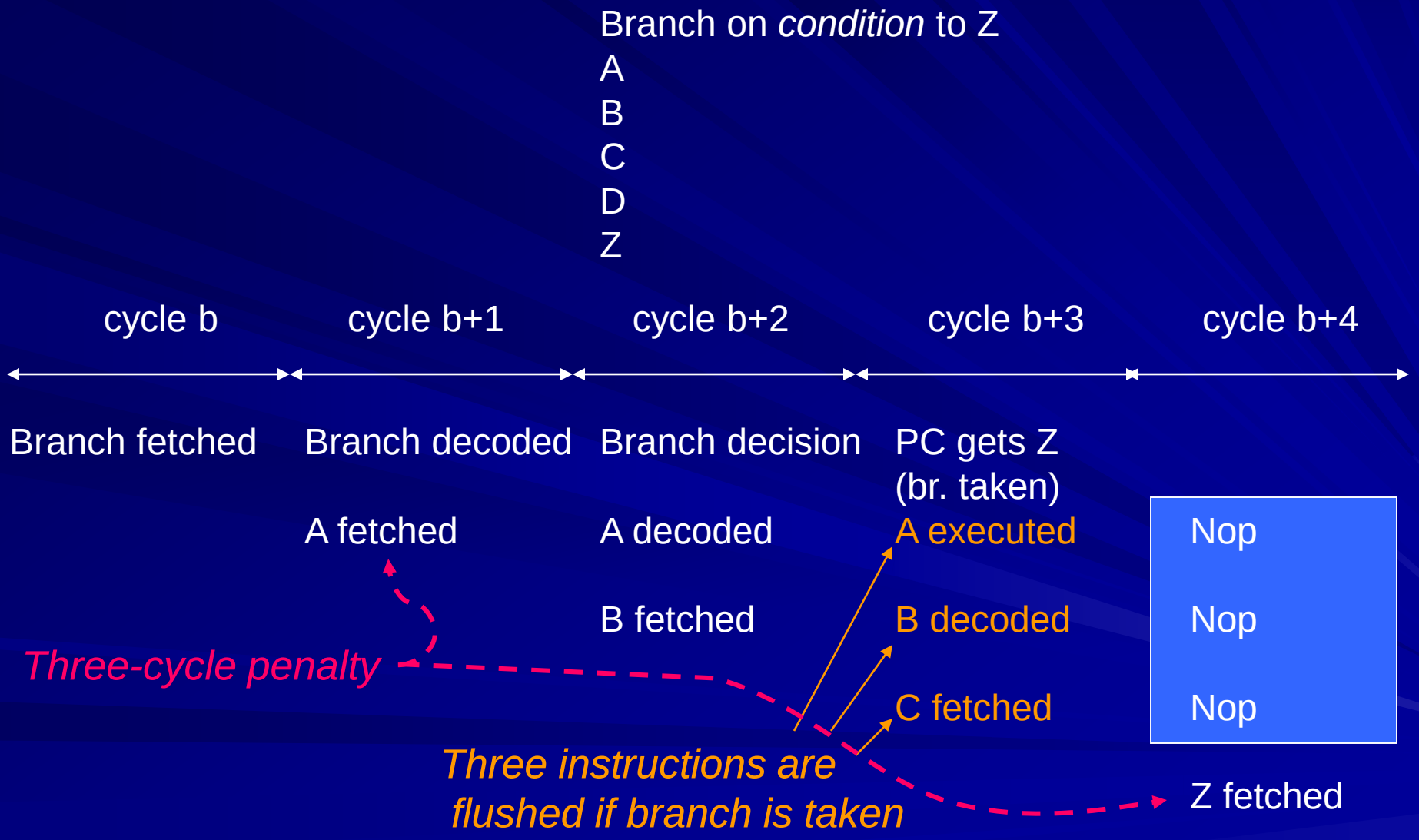
Branch Hazard



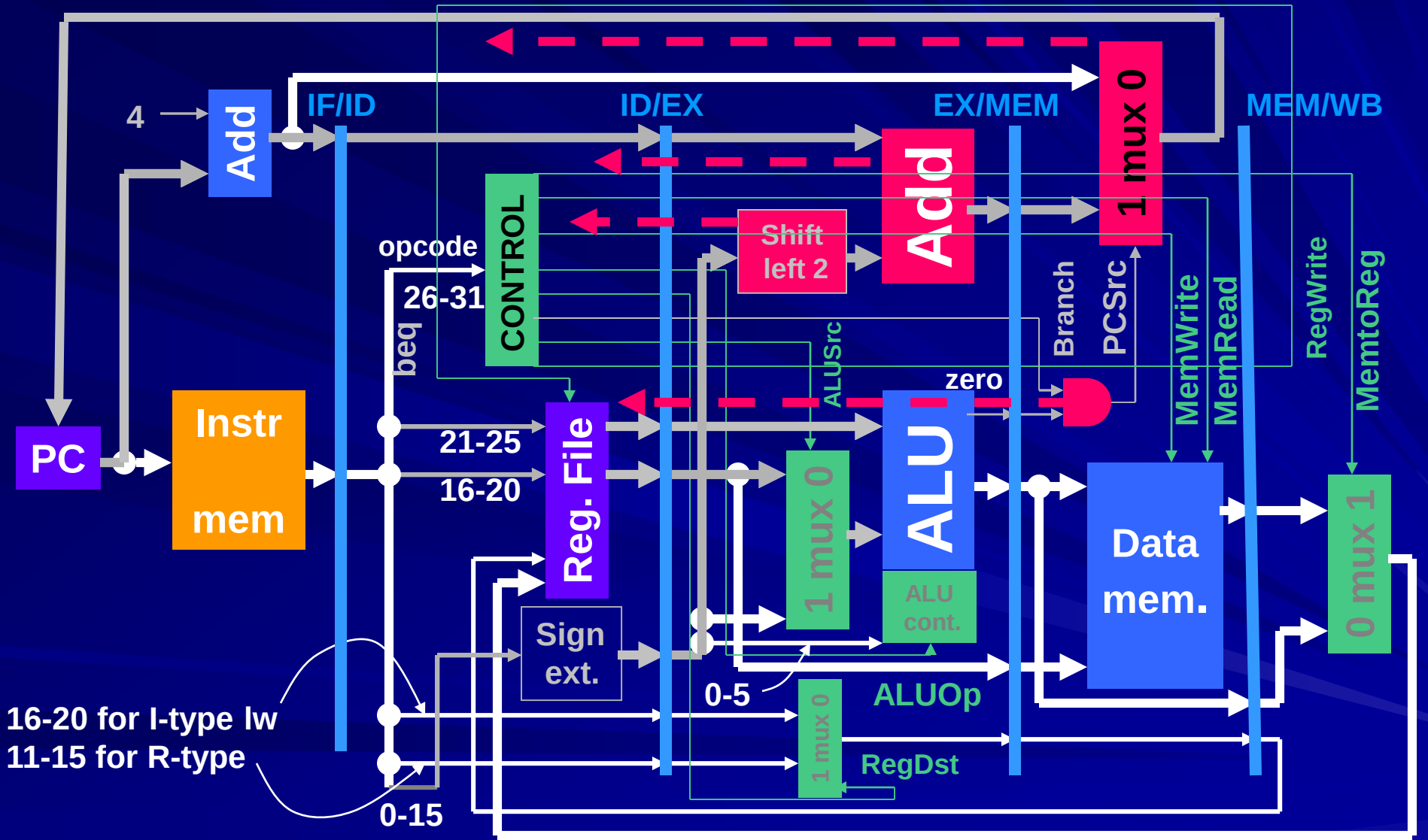
Branch Not Taken



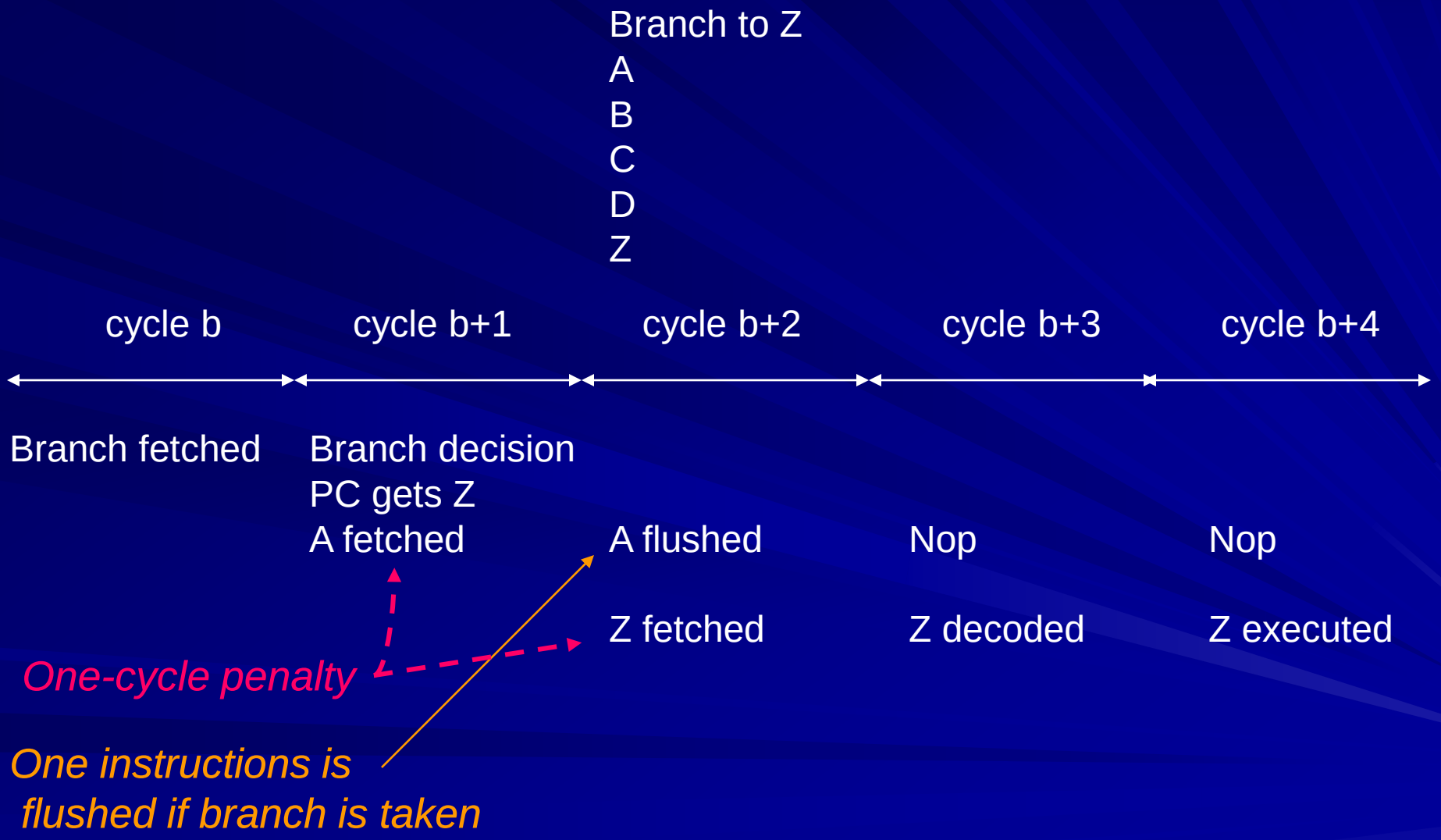
Branch Taken



Branch Penalty Reduction



Branch Taken

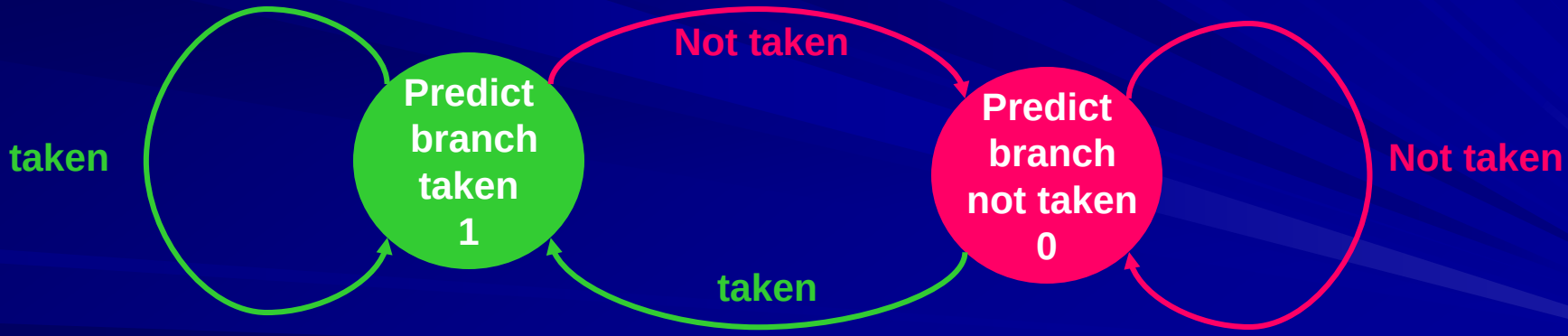


Pipeline Flush

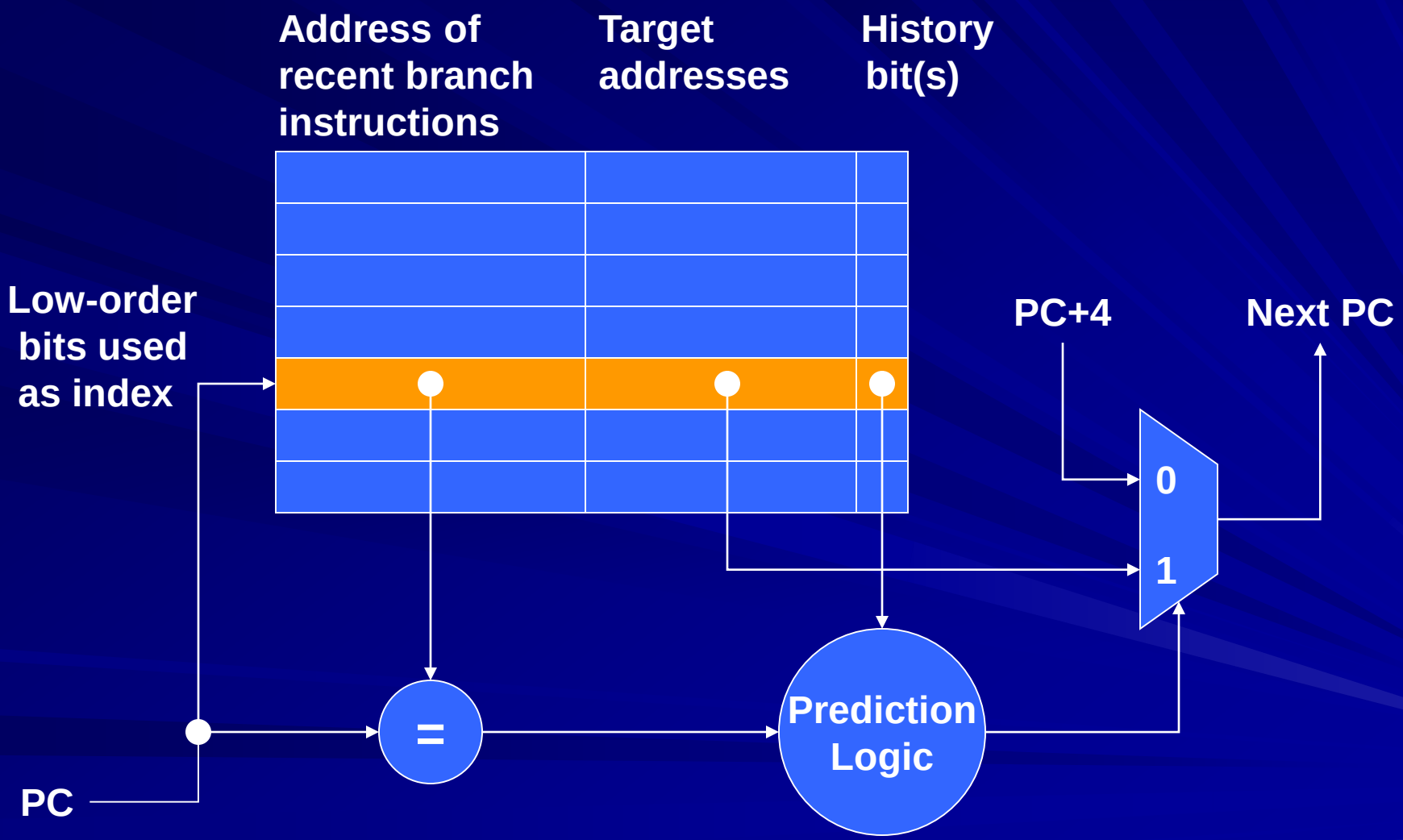
- If branch is taken (as indicated by *zero*), then control does the following:
  - Change all control signals to 0, similar to the case of stall for data hazard, i.e., insert bubble in the pipeline.
  - Generate a signal *IF.Flush* that changes the instruction in the pipeline register IF/ID to 0 (nop).
- Penalty of branch hazard is reduced by
  - Adding branch detection and address generation hardware in the decode cycle – **one bubble needed** – *a next address generation logic in the decode stage writes PC+4, branch address, or jump address into PC.*
  - Unrolling loops.
  - What's better than branch detection? Branch prediction!

Branch Prediction

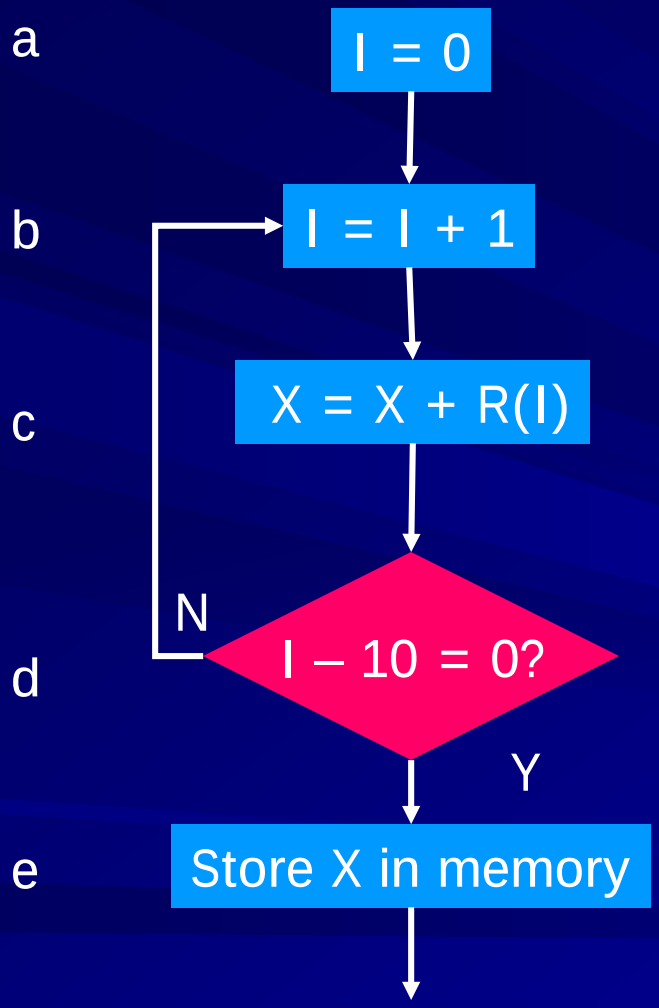
- Useful for program loops.
- A one-bit prediction scheme: a one-bit buffer carries a “history bit” that tells what happened on the last branch instruction
  - History bit = 1, branch was taken
  - History bit = 0, branch was not taken



Branch Prediction



Branch Prediction for a Loop



Execution of Instruction d

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

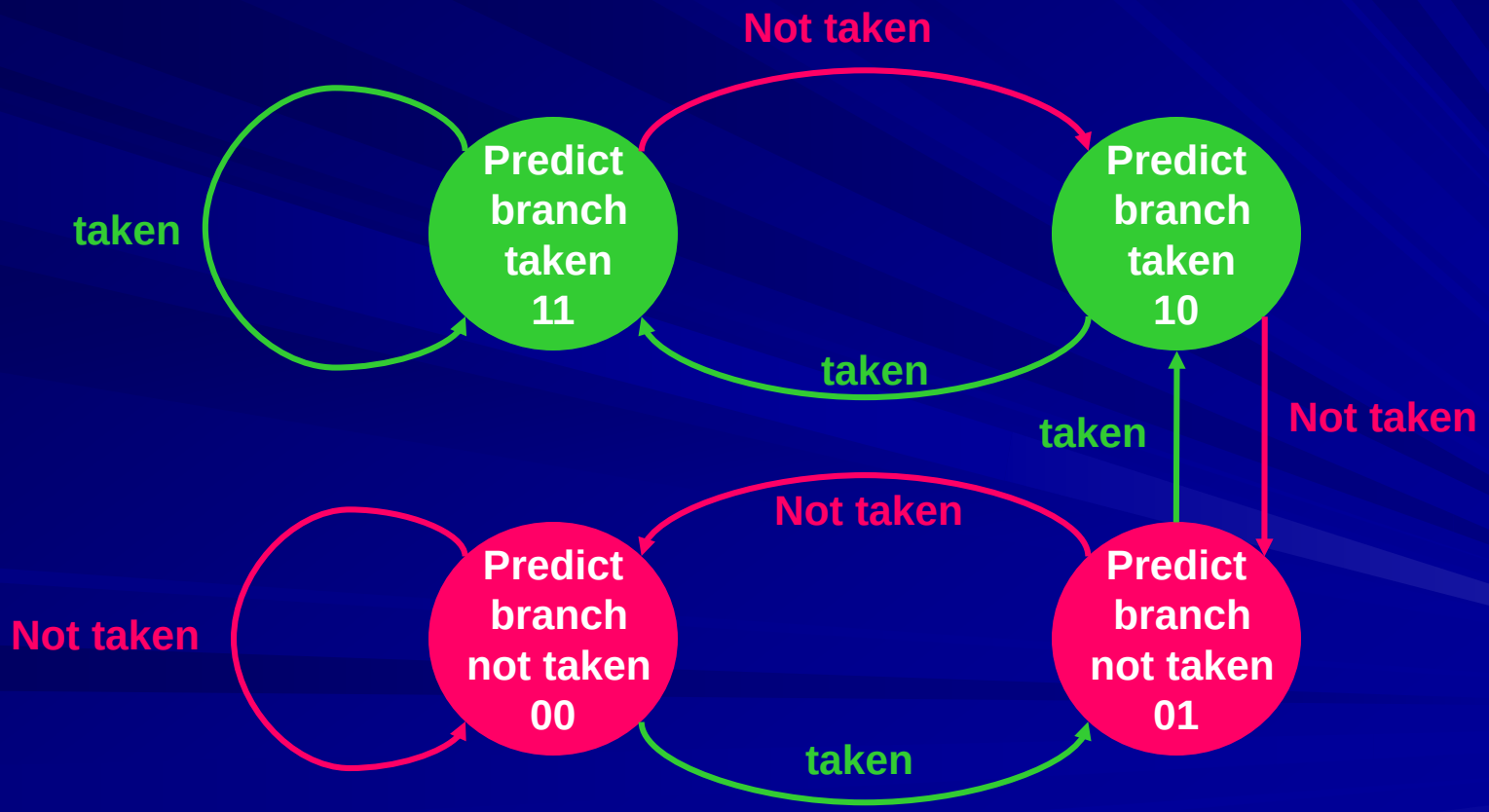
h.bit = 0 *branch not taken*, h.bit = 1 *branch taken*.

Prediction Accuracy

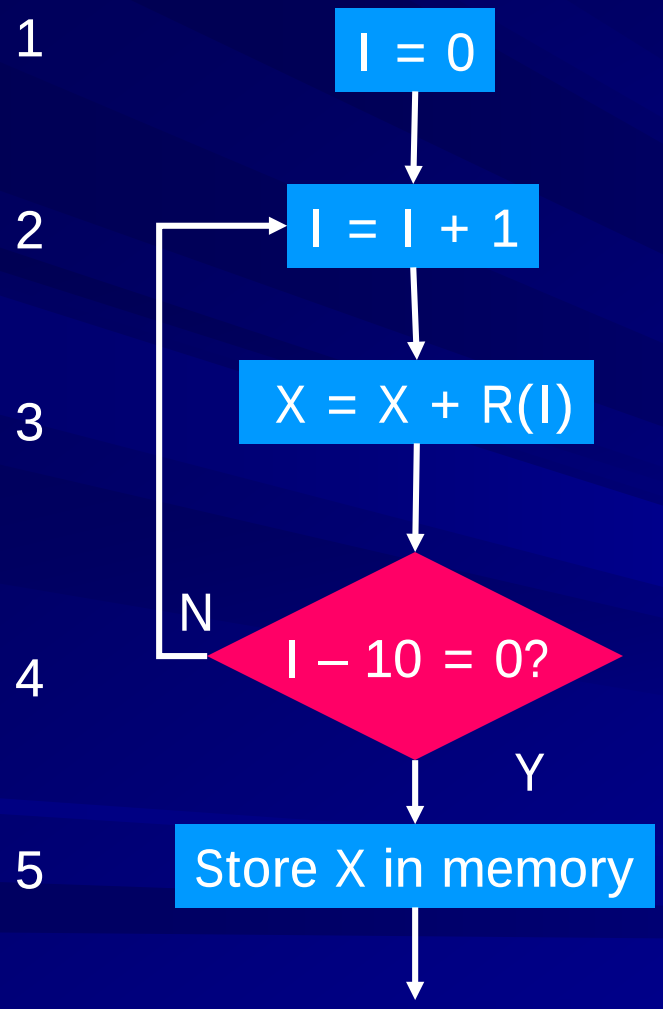
- One-bit predictor:
  - 2 errors out of 10 predictions
  - Prediction accuracy = 80%
- To improve prediction accuracy, use two-bit predictor:
  - A prediction must be wrong twice before it is changed

Two-Bit Prediction Buffer

- Implemented as a two-bit counter.
- 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

Exceptions

- A typical exception occurs when ALU produces an *overflow* signal.
- Control asserts following actions on exception:
  - Change the PC address to 4000 0040hex. This is the location of the exception routine. This is done by adding an additional input to the PC input multiplexer.
  - Overflow is detected in the EX cycle. Similar to data hazard and pipeline flush,
    - Set IF/ID to 0 (nop).
    - Generate ID.Flush and EX.Flush signals to set all control signals to 0 in ID/EX and EX/MEM registers. This also prevents the ALU result (presumed contaminated) from being written in the WB cycle.

Single-Cycle Performance

- Assume
  - 200 ps for memory access
  - 100 ps for ALU operation
  - 50 ps for register file read or write
- Cycle time set according to longest instruction:

  lw ≡ IF + ID/RegRead + ALU + MEM + RegWrite

  = 200 + 50 +100 + 200 + 50

  = 600 ps
- Av. instruction execution time = clock cycle time

  = 600 ps

Multicycle Performance

- Consider SPECINT2000* instruction mix:
  - 25% lw 5 cycles
  - 10% sw 4 cycles
  - 11% branch 3 cycles
  - 2% jump 3 cycles
  - 52% ALU instr. 4 cycles
- Av. CPI = 0.25×5 + 0.10×4 + 0.11×3 + 0.02×3 + 0.52×4
  = 4.12
- Clock cycle time determined from longest operation (memory access) = 200 ps
- Av. instruction execution time = 4.12×200 = 824 ps

Set of benchmark programs used for performance evaluation.

Pipeline Performance

- Neglect initial latency (reasonable for long programs).
- One instruction completed every clock cycle unless delayed by hazard. Average CPI:

lw	2 cycles in 50% cases due to hazard	1.5 cycles
sw		1 cycle
ALU		1 cycle
branch	2 cycles in 25% cases due to hazard	1.25 cycles
jump		2 cycles

- For SPECINT2000

  Av. CPI = 0.25×1.5 + 0.10×1 + 0.11×1.25 + 0.02×2.0 + 0.52×1

  = 1.17

- Clock cycle time (longest operation: memory access) = 200 ps

- Av. instruction execution time = 1.17×200 = 234 ps

Comparing Alternatives

Type of datapath and control	Clock cycle time	Average CPI	Av. instruction execution time
Single-cycle	600 ps	1.00	600 ps
Multicycle	200 ps	4.12	824 ps
Pipelined	200 ps	1.17	234 ps

Next Class

Chapter 5

Memory Organization