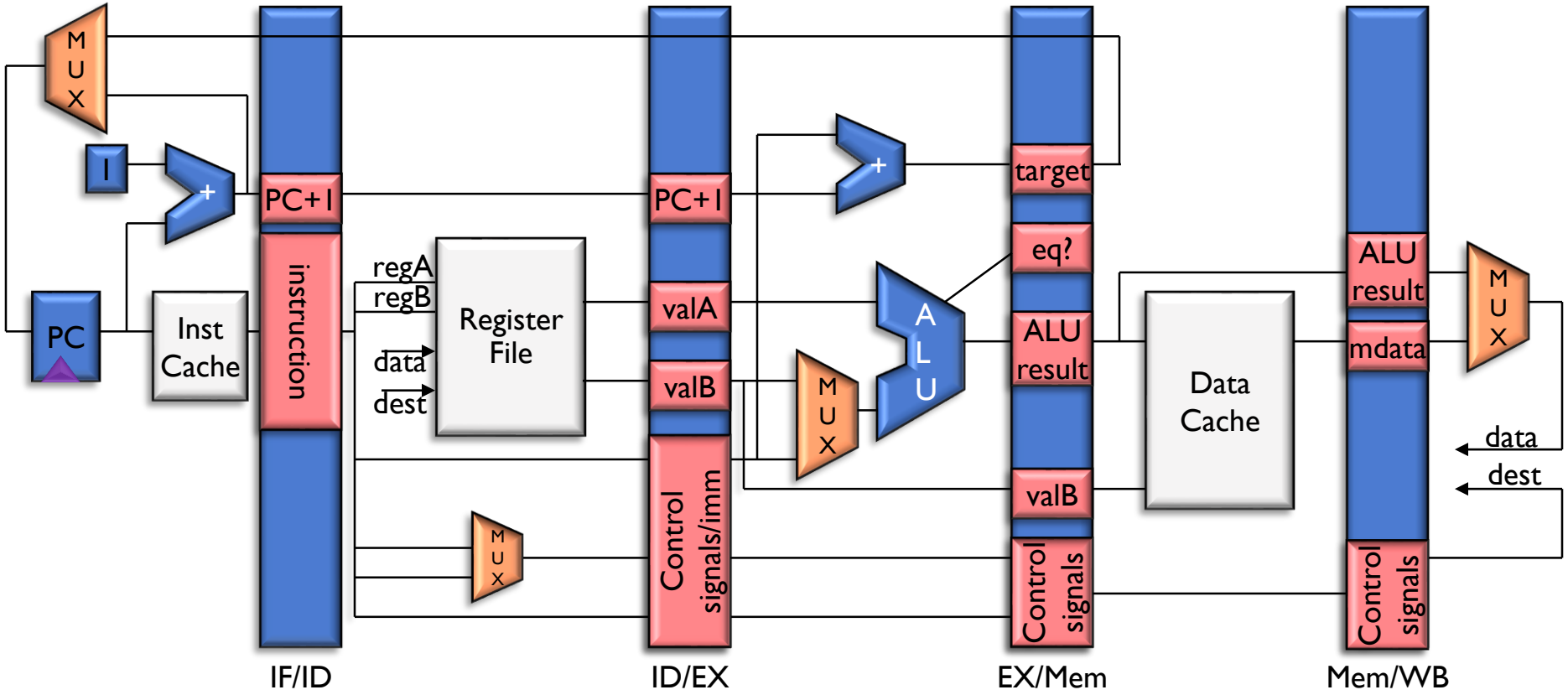
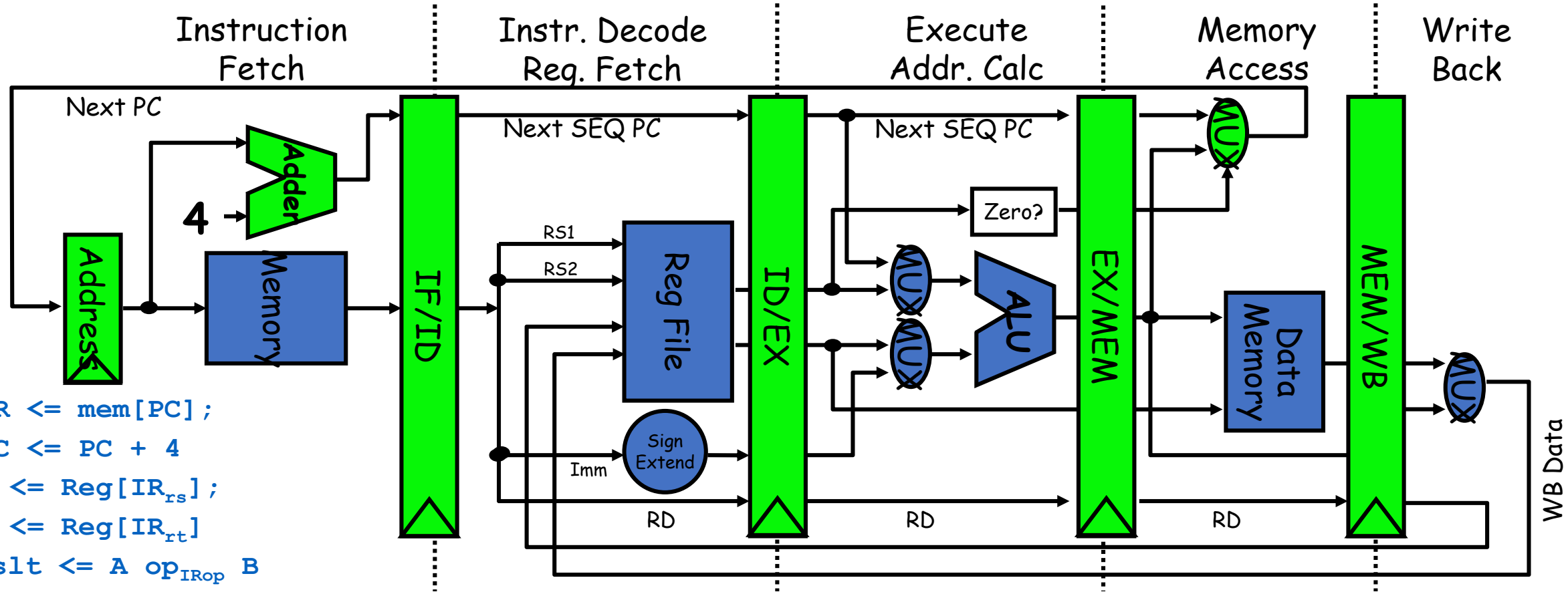
Lecture-5 & 6 (Data/Control Hazards) CS422-Spring 2020

Biswa@CSE-IITK

Putting It All Together



Another view



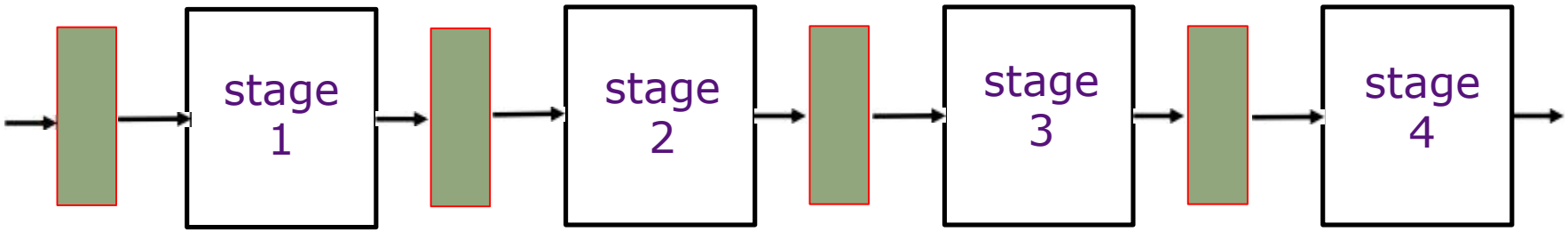

```
IR <= mem[PC];
PC <= PC + 4
A <= Reg[IR_rs];
B <= Reg[IR_rt]
rslt <= A op_IRop B
WB <= rslt
Reg[IR_rd] <= WB
```

- Data stationary control
  - local decode for each instruction phase / pipeline stage

Pipelining Idealism

- Uniform Sub-operations
  - Operation can partitioned into uniform-latency sub-ops
- Repetition of Identical Operations
  - Same ops performed on many different inputs
- Independent Operations
  - All ops are mutually independent

Idealism Continued



- All objects go through the same stages
- No sharing of resources between any two stages
- Propagation delay through all pipeline stages is equal
- The scheduling of an object entering the pipeline is not affected by the objects in other stages

These conditions generally hold for industrial assembly lines.

But what about an instruction pipeline?

Pipeline Realism

- Uniform Sub-operations … NOT!
  - Balance pipeline stages
- Repetition of Identical Operations … NOT!
  - Unifying instruction types
- Independent Operations … NOT!
  - Resolve data and resource hazards

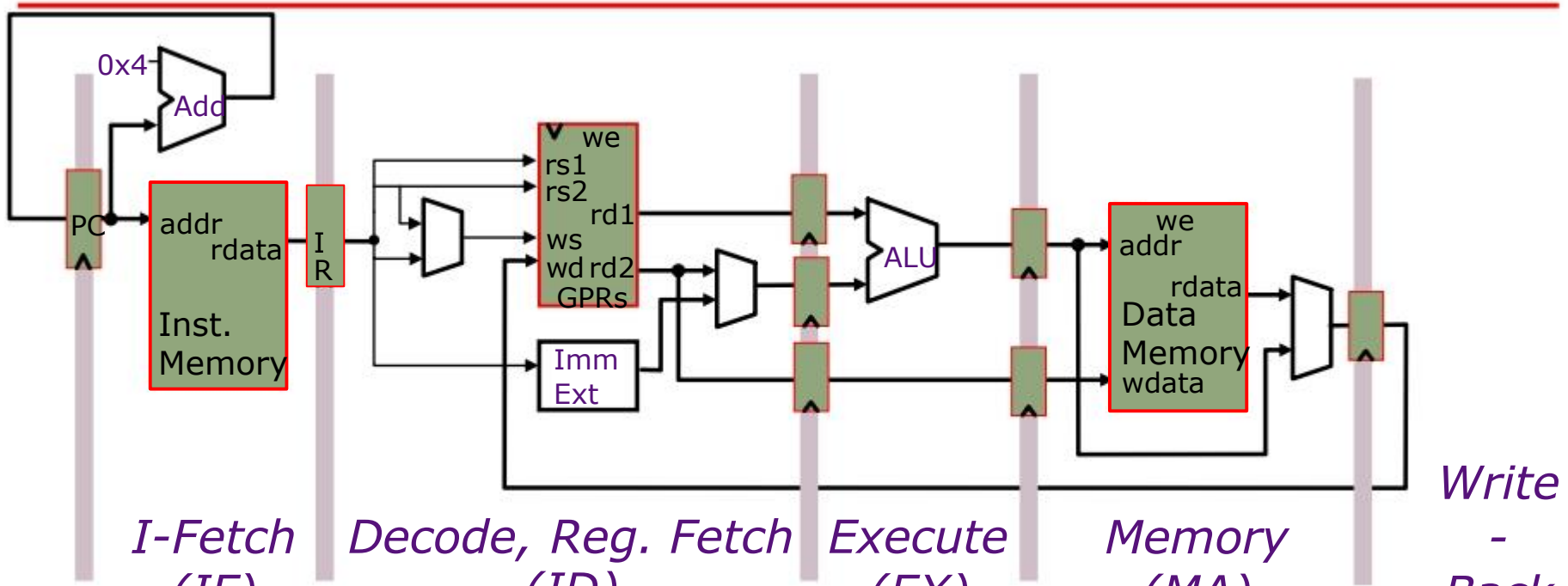
How to Divide the Datapath?

Suppose memory is significantly slower than other stages.
For example, suppose

t_{IM} = 10 units
t_{DM} = 10 units
t_{ALU} = 5 units
t_{RF} = 1 unit
t_{RW} = 1 unit

Since the slowest stage determines the clock, it may be possible to combine some stages without any loss of performance

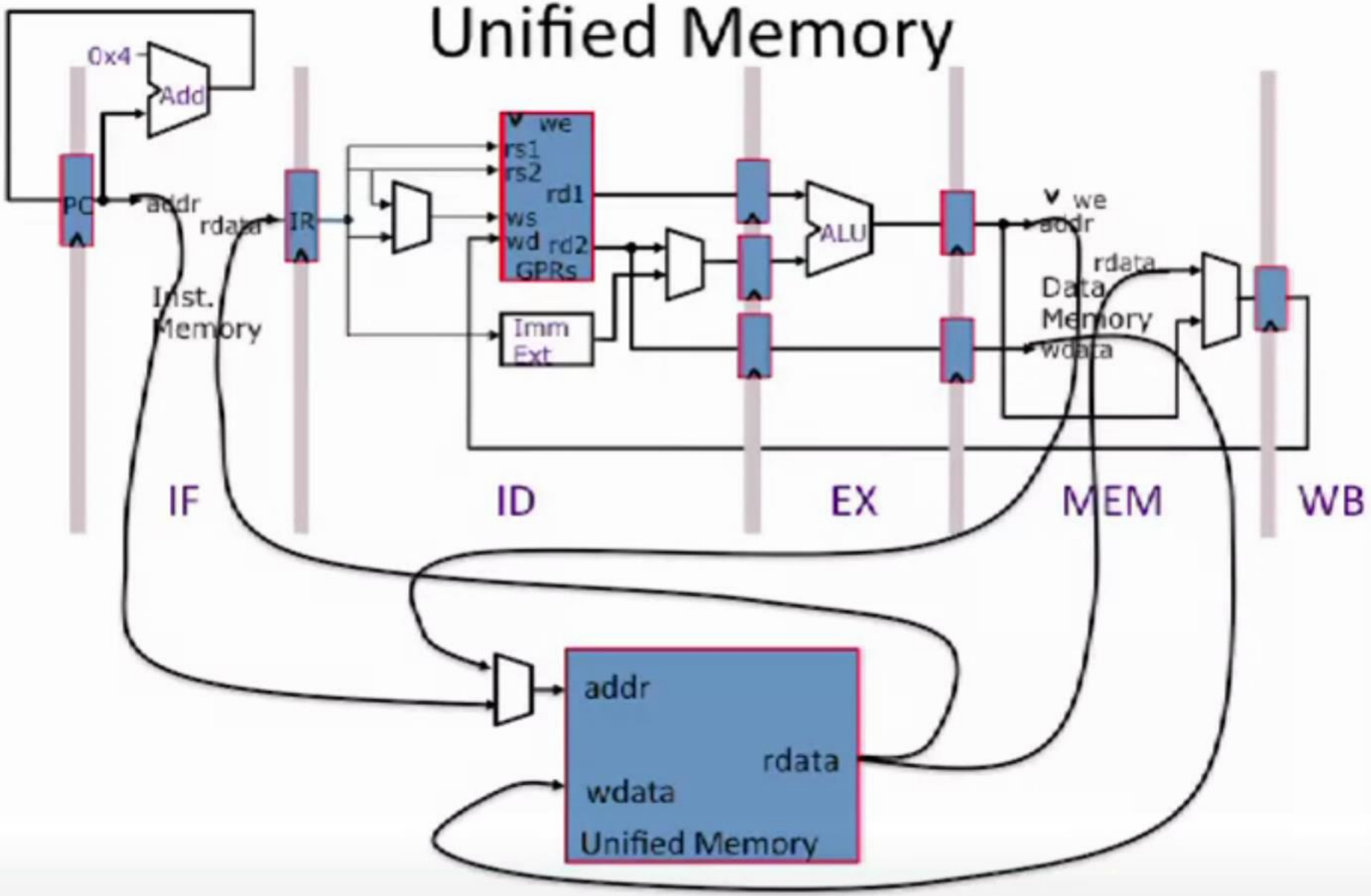
Resource Usage



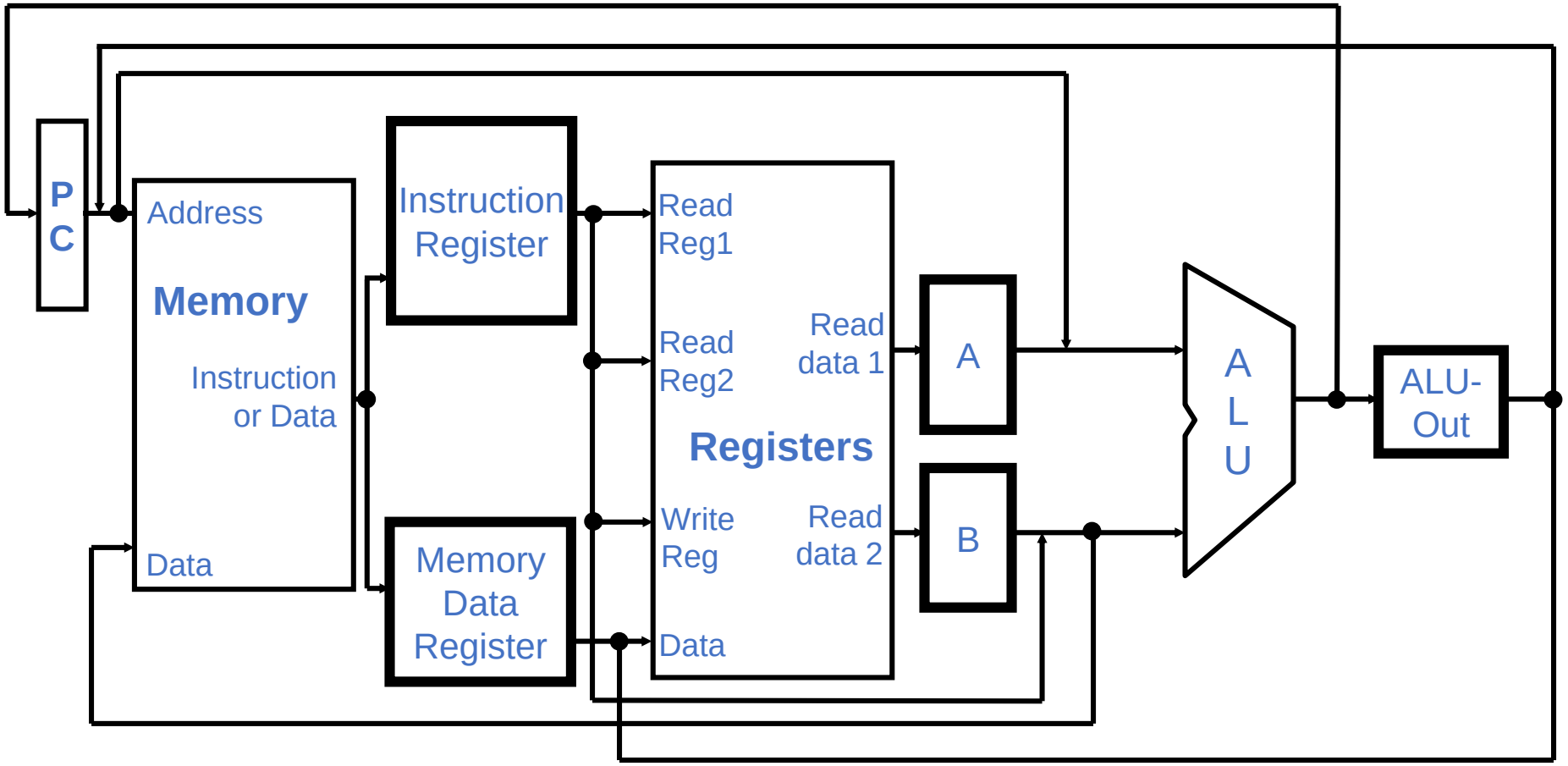
I-Fetch (IF) | *Decode, Reg. Fetch (ID)* | *Execute (EX)* | *Memory (MA)* | *Write - Back (WB)*

	time	t0	t1	t2	t3	t4	t5	t6	t7	. . . .
Resources	*IF*	I_1	I_2	I_3	I_4	I_5				
	ID		I_1	I_2	I_3	I_4	I_5			
	EX			I_1	I_2	I_3	I_4	I_5		
	MA				I_1	I_2	I_3	I_4	I_5	
	WB					I_1	I_2	I_3	I_4	I_5

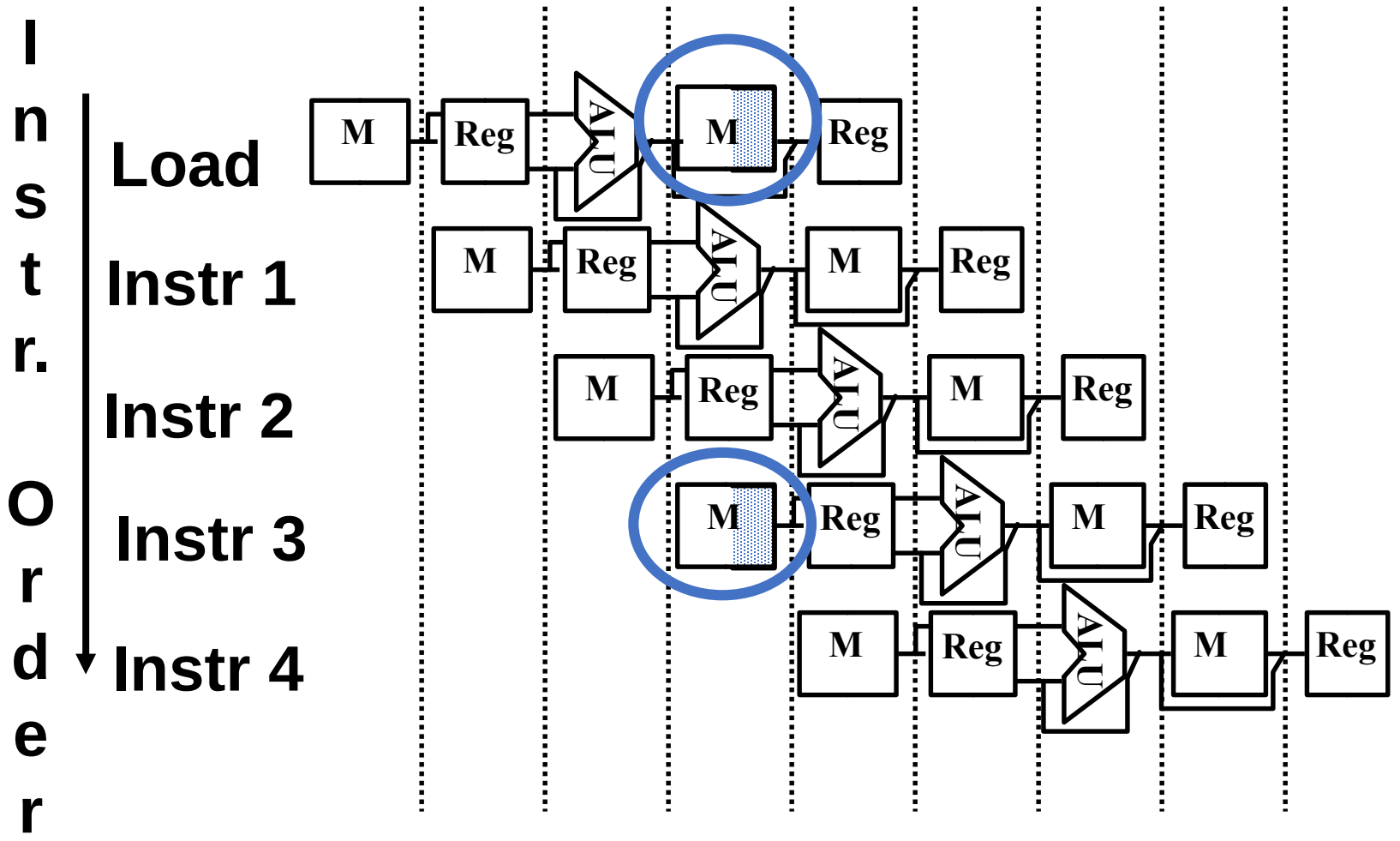
Example Structural Hazard: Unified Memory



A Simplified View



Example

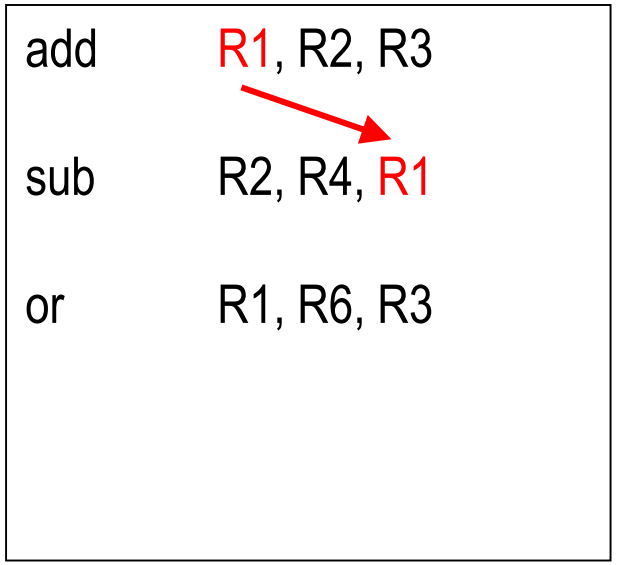


- Can't read same memory twice in same clock cycle

Data Dependences/hazards

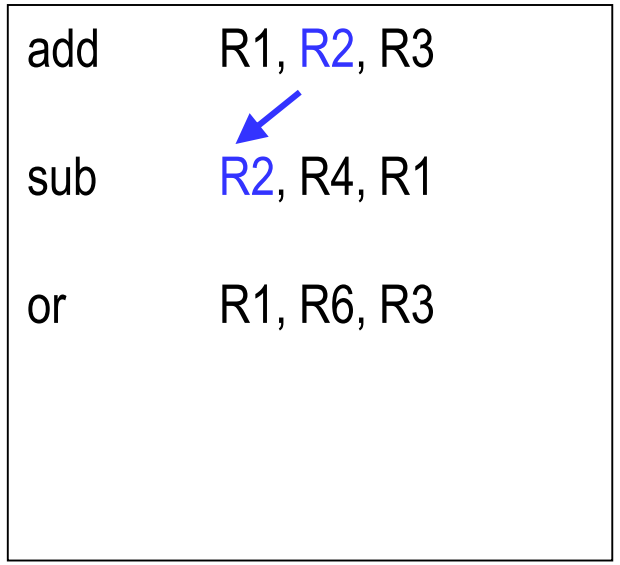
- Data Dependence
  - *Read-After-Write* (*RAW*) (the only true dependence)
    - Read must wait until earlier write finishes
  - *Anti-Dependence* (*WAR*)
    - Write must wait until earlier read finishes (avoid clobbering)
  - *Output Dependence* (*WAW*)
    - Earlier write can’t overwrite later write

RAW/WAR/WAW



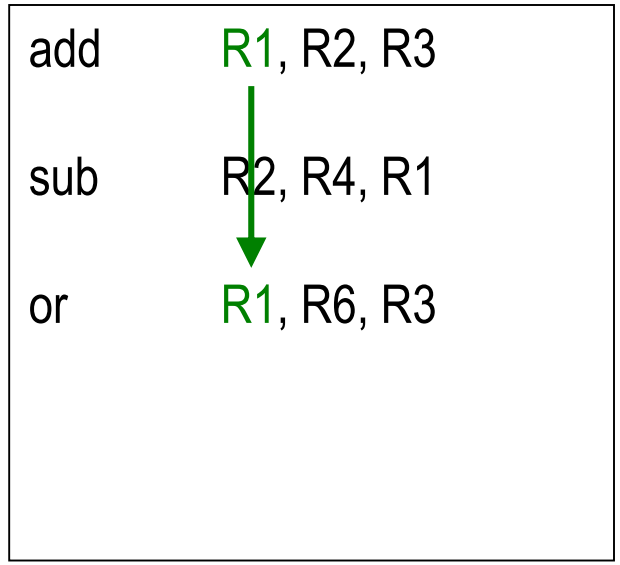
read-after-write
(RAW)

True dependence
(real)



write-after-read
(WAR)

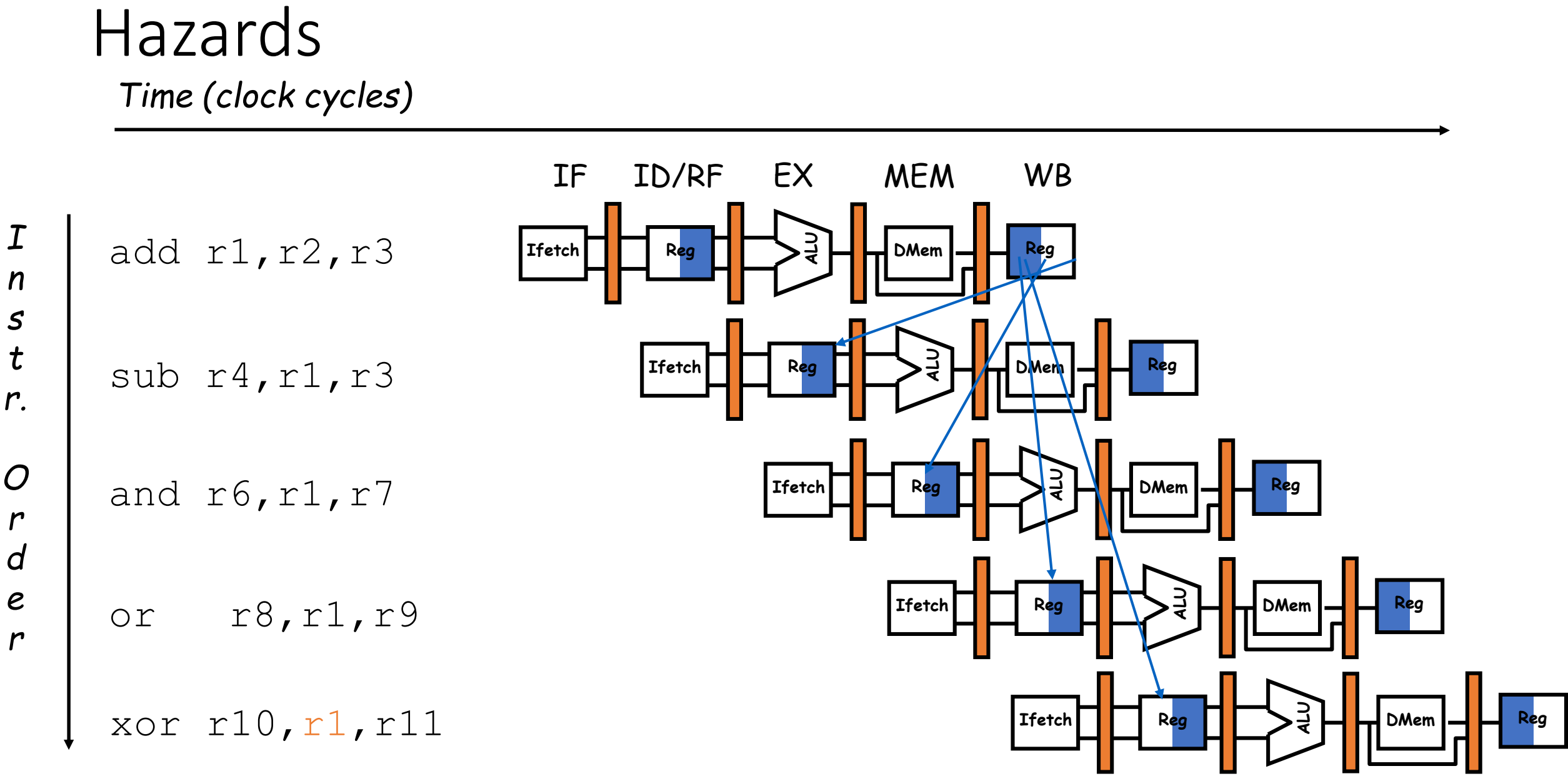
anti dependence
(artificial)



write-after-write
(WAW)

output dependence
(artificial)

Hazards


Time (clock cycles)
Instr. Order
add r1,r2,r3
sub r4,r1,r3
and r6,r1,r7
or r8,r1,r9
xor r10,r1,r11
IF
ID/RF
EX
MEM
WB
Ifetch
Reg
ALU
DMem
Reg

Strategies

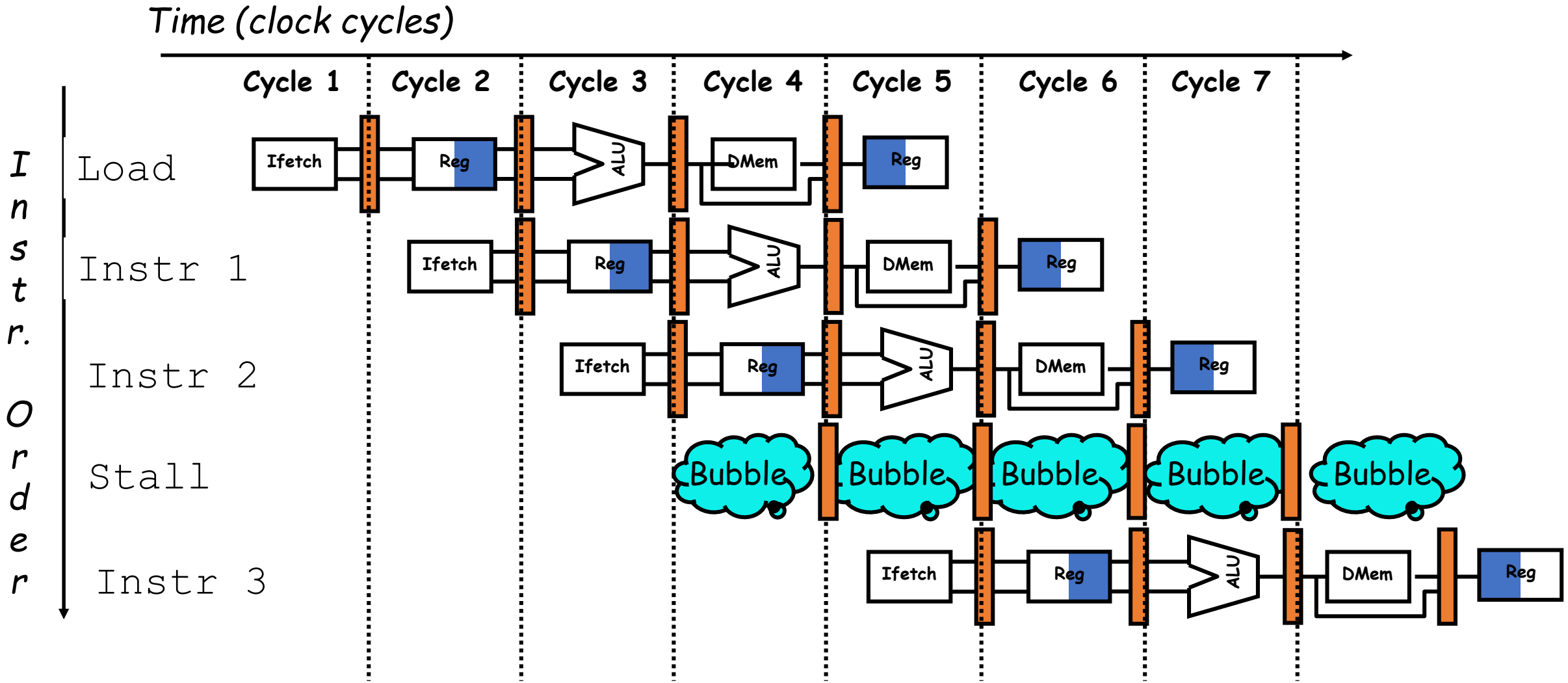
Strategy 1: *Wait for the result to be available by freezing earlier pipeline stages* → *stall*

Strategy 2: *Route data as soon as possible after it is calculated to the earlier pipeline stage* → *bypass*

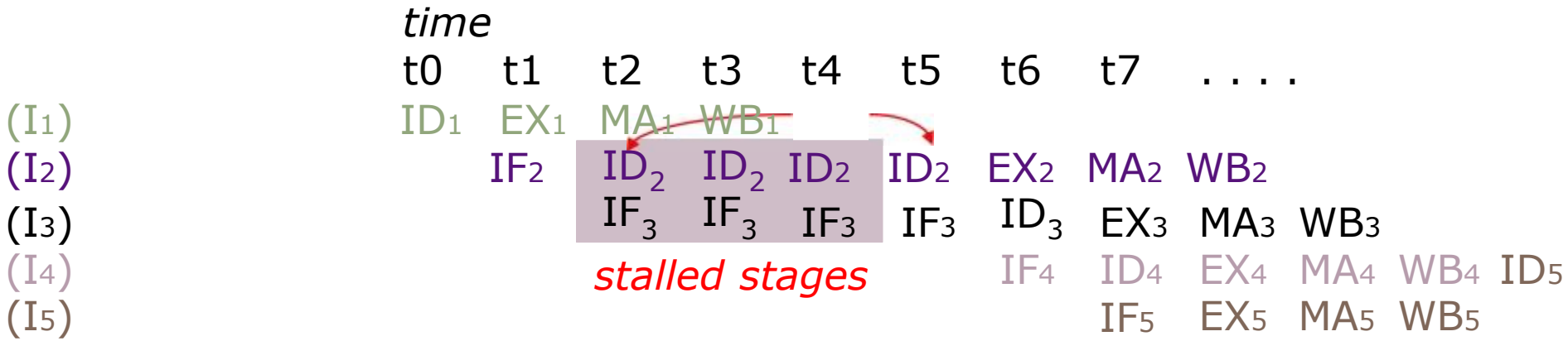
Strategy 3: *Speculate on the dependence Two cases:*

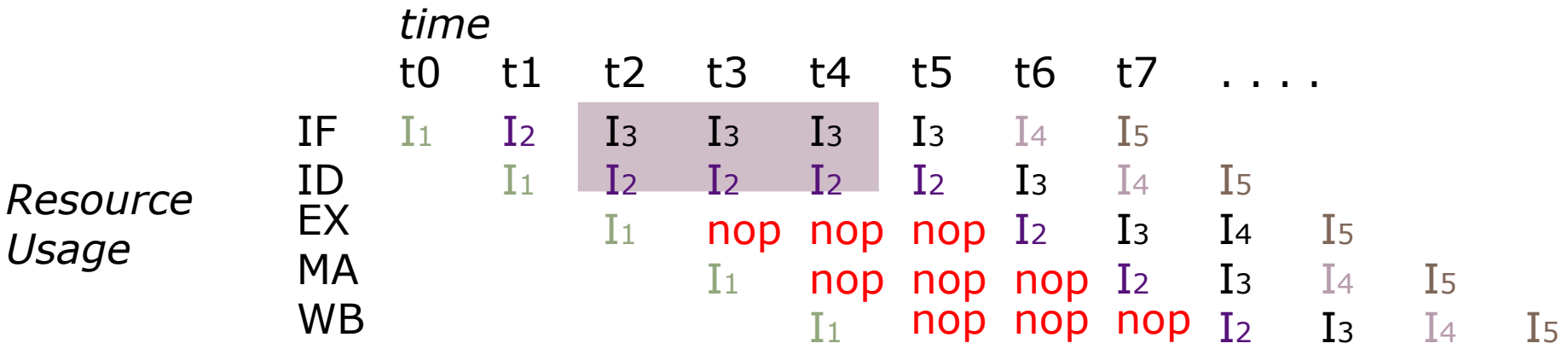
Guessed correctly → do nothing Guessed incorrectly → kill and restart

Bubble


Time (clock cycles)
Cycle 1
Cycle 2
Cycle 3
Cycle 4
Cycle 5
Cycle 6
Cycle 7
Instr. Order
Load
Instr 1
Instr 2
Stall
Instr 3
Ifetch
Reg
ALU
DMem
Reg
Bubble
Bubble
Bubble
Bubble
Bubble

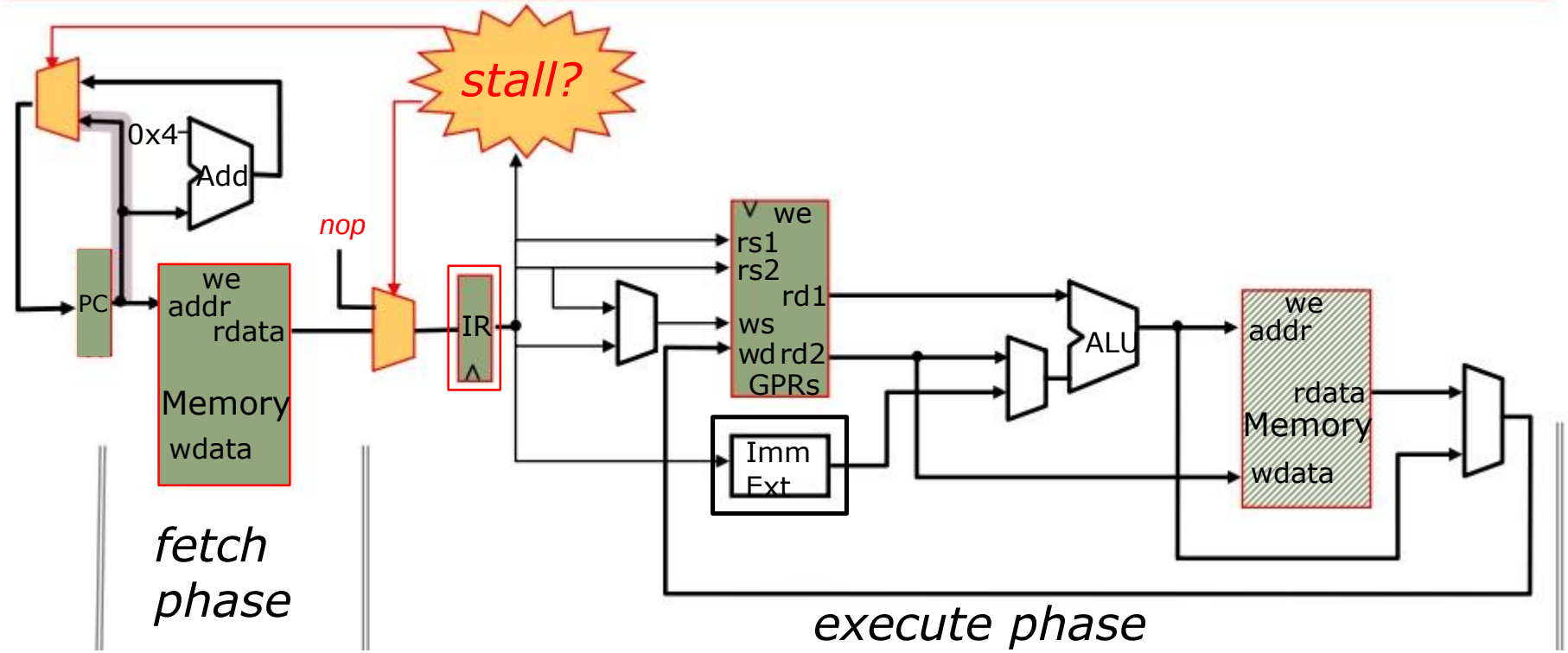
Stalled Stages & Bubbles





nop: pipeline bubble

Stall



When stall condition is indicated

- *don't fetch a new instruction and don't change the PC* • *insert a nop in the IR*

Strategy 1: *Wait for the result to be available by freezing earlier pipeline stages* → *stall*

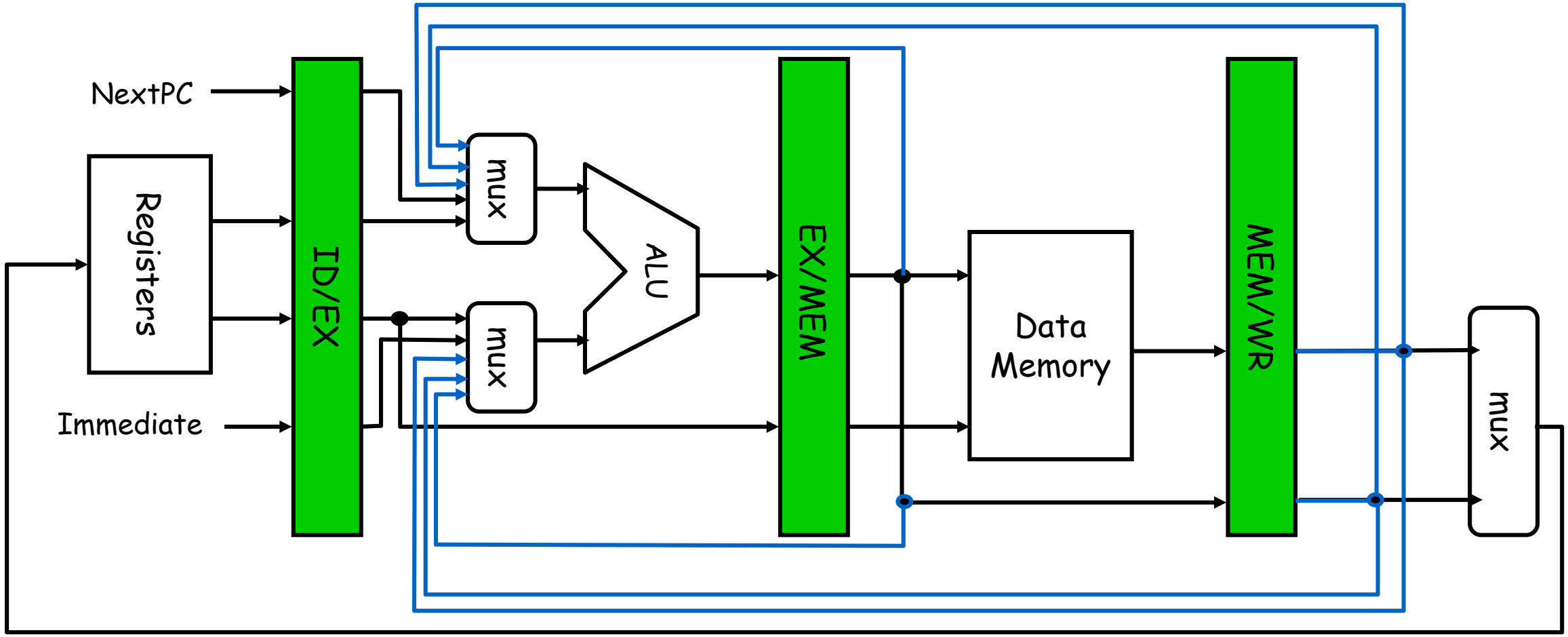
Strategy 2: *Route data as soon as possible after it is calculated to the earlier pipeline stage* → *bypass*

Strategy 3: *Speculate on the dependence*

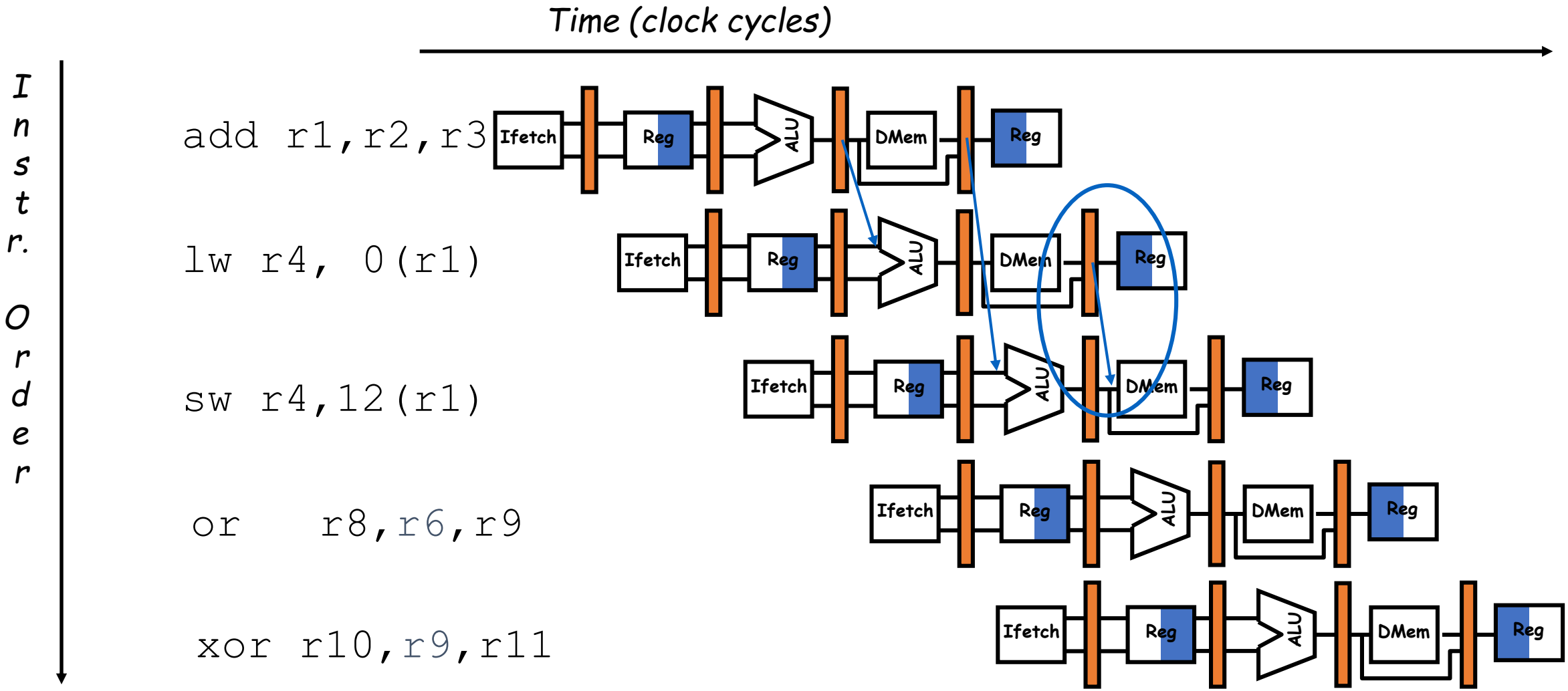
Two cases:

Guessed correctly → do nothing Guessed incorrectly → kill and restart

Bypass Network



Does it Help?

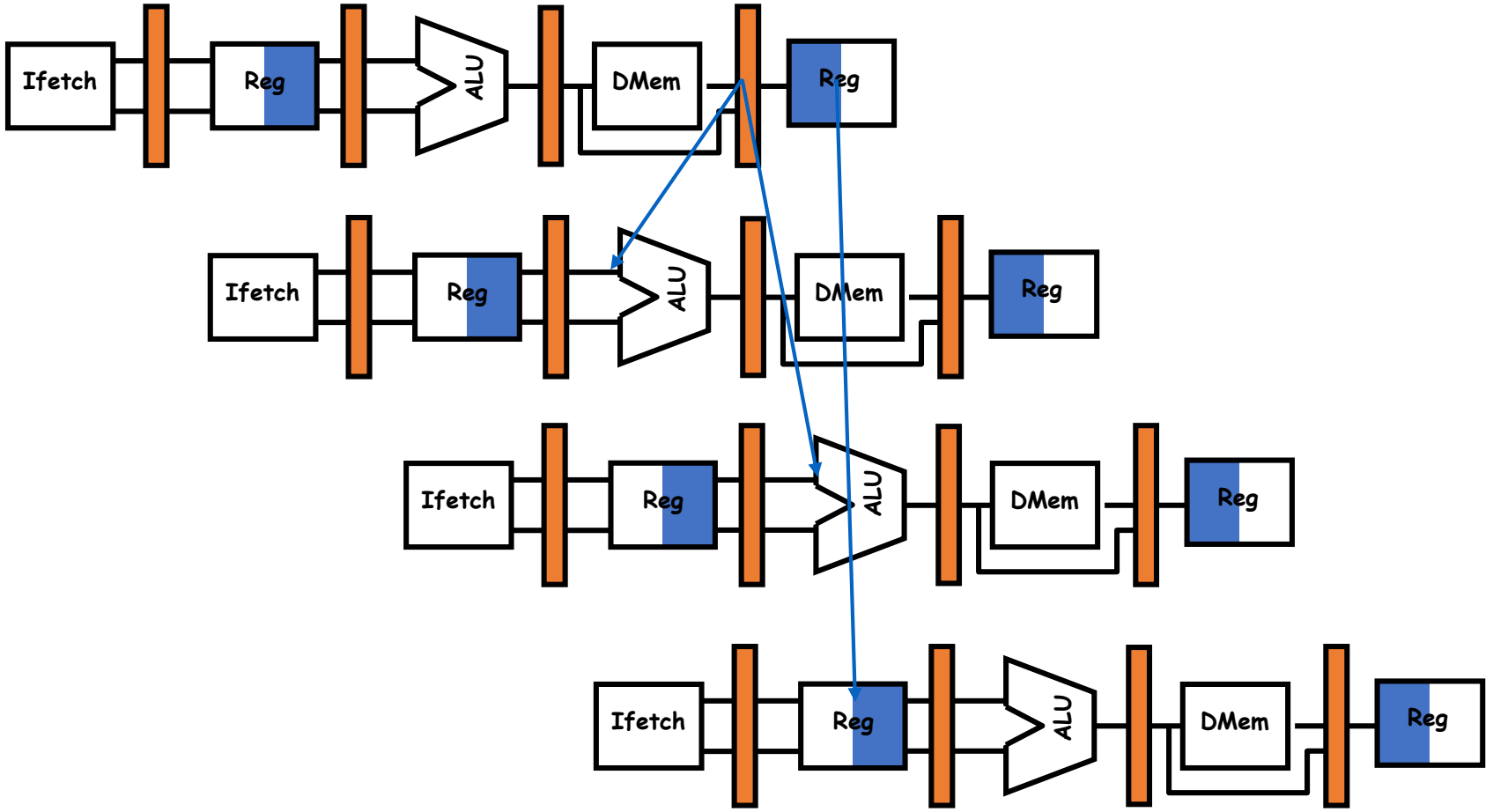

Time (clock cycles)
Instr. Order
add r1,r2,r3
lw r4, 0(r1)
sw r4,12(r1)
or r8,r6,r9
xor r10,r9,r11
Ifetch
Reg
ALU
DMem
Reg

Always?

Time (clock cycles)

Instr. Order

```
lw  r1, 0(r2)
sub r4,r1,r6
and r6,r1,r7
or  r8,r1,r9
```



Control Hazard

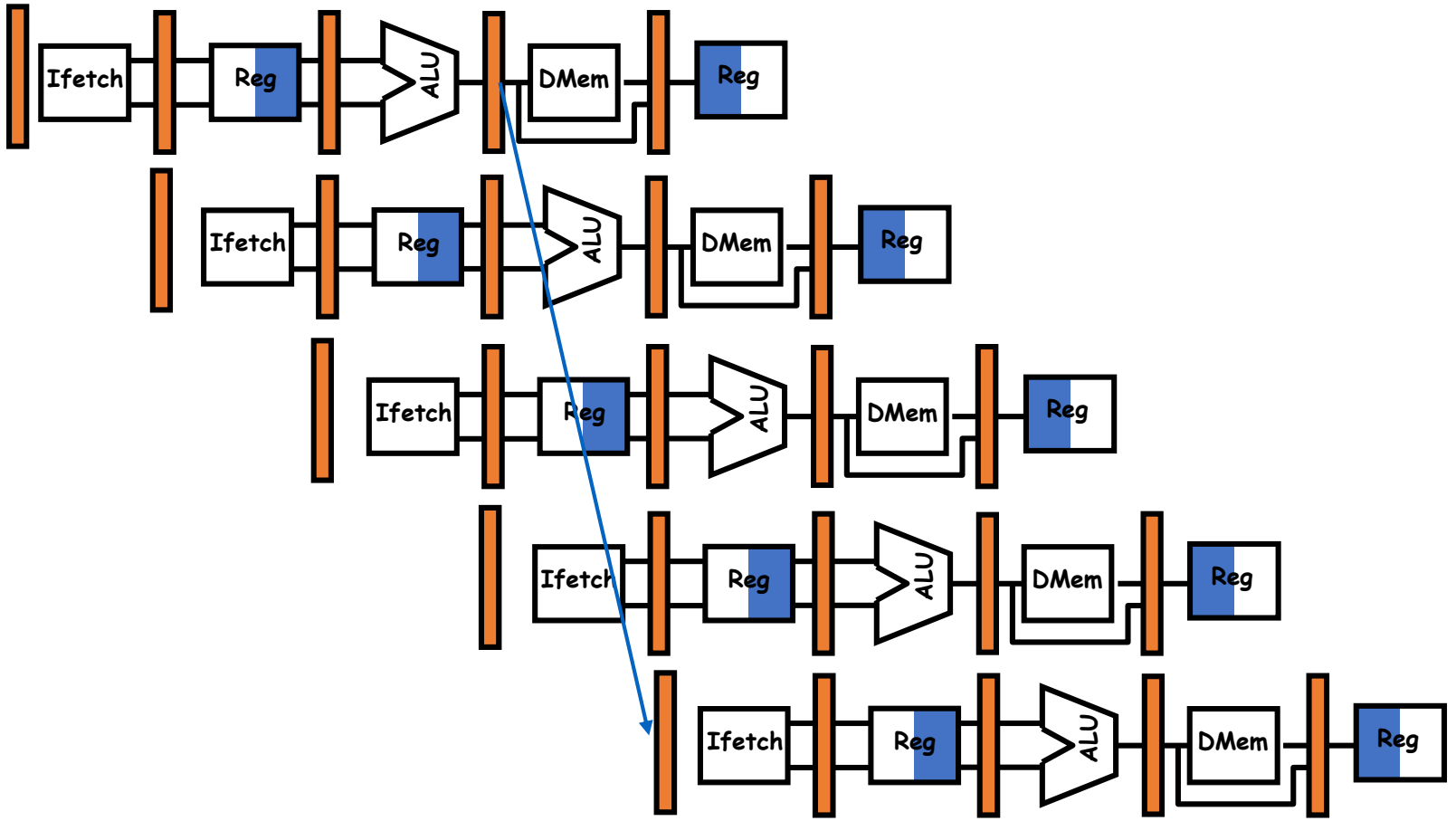
```
10: beq r1,r3,36

14: and r2,r3,r5

18: or  r6,r1,r7

22: add r8,r1,r9

36: xor r10,r1,r11
```



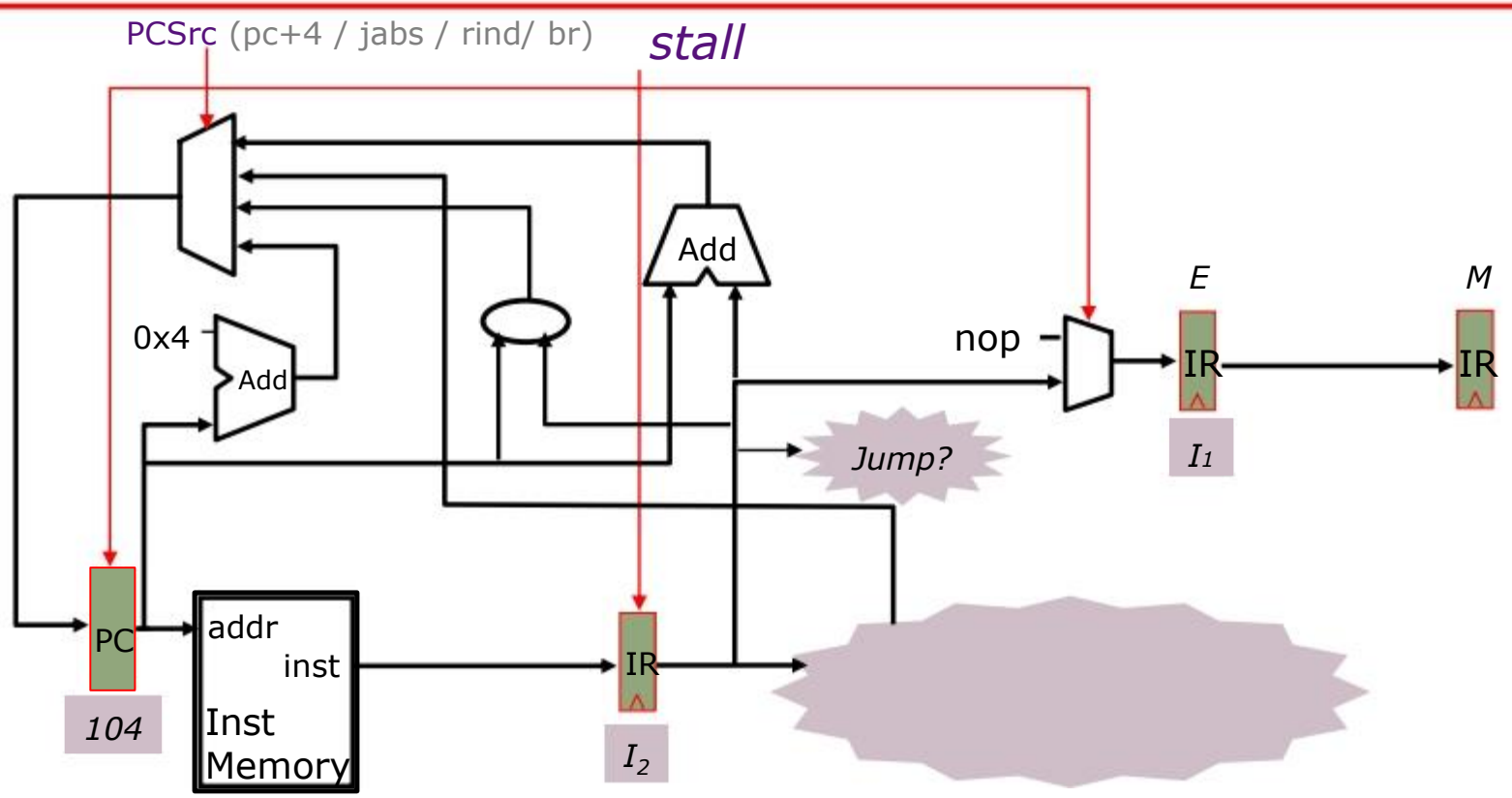
What do you do with the 3 instructions in between?
How do you do it?

Hazard Detection

- For Jump
  - Opcode, offset, and PC
- For Conditional Branches
  - Opcode, offset, PC, and register (for condition)

- In what stage do we know these?
  - PC → Fetch
  - Opcode, offset → Decode
  - Register value → Decode
  - Branch condition ((rs)==0) → Execute

Speculate: PC = PC+4

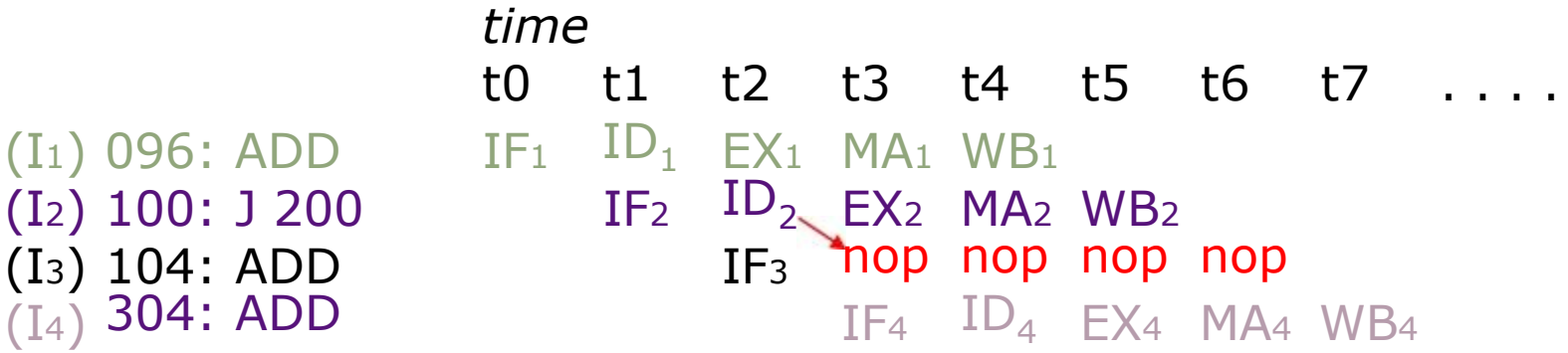


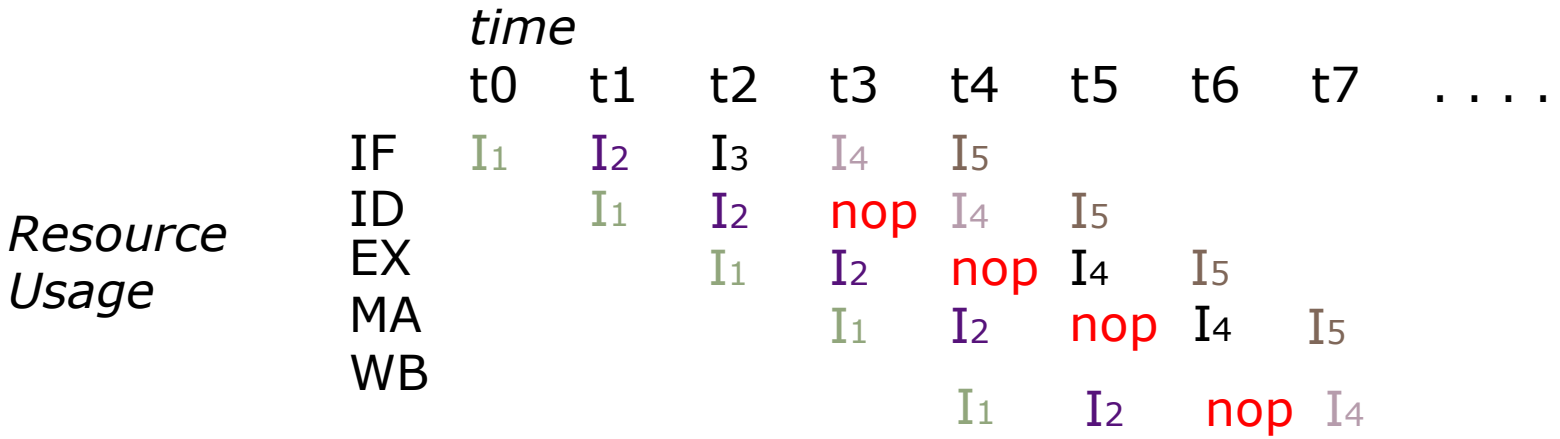
I_1	096	ADD	
I_2	100	J304	
I_3	~~104~~	~~ADD~~	*kill*
I_4	304	ADD	

What happens on mis-speculation, i.e., when next instruction is not PC+4?

How?

Jump Pipeline Usage





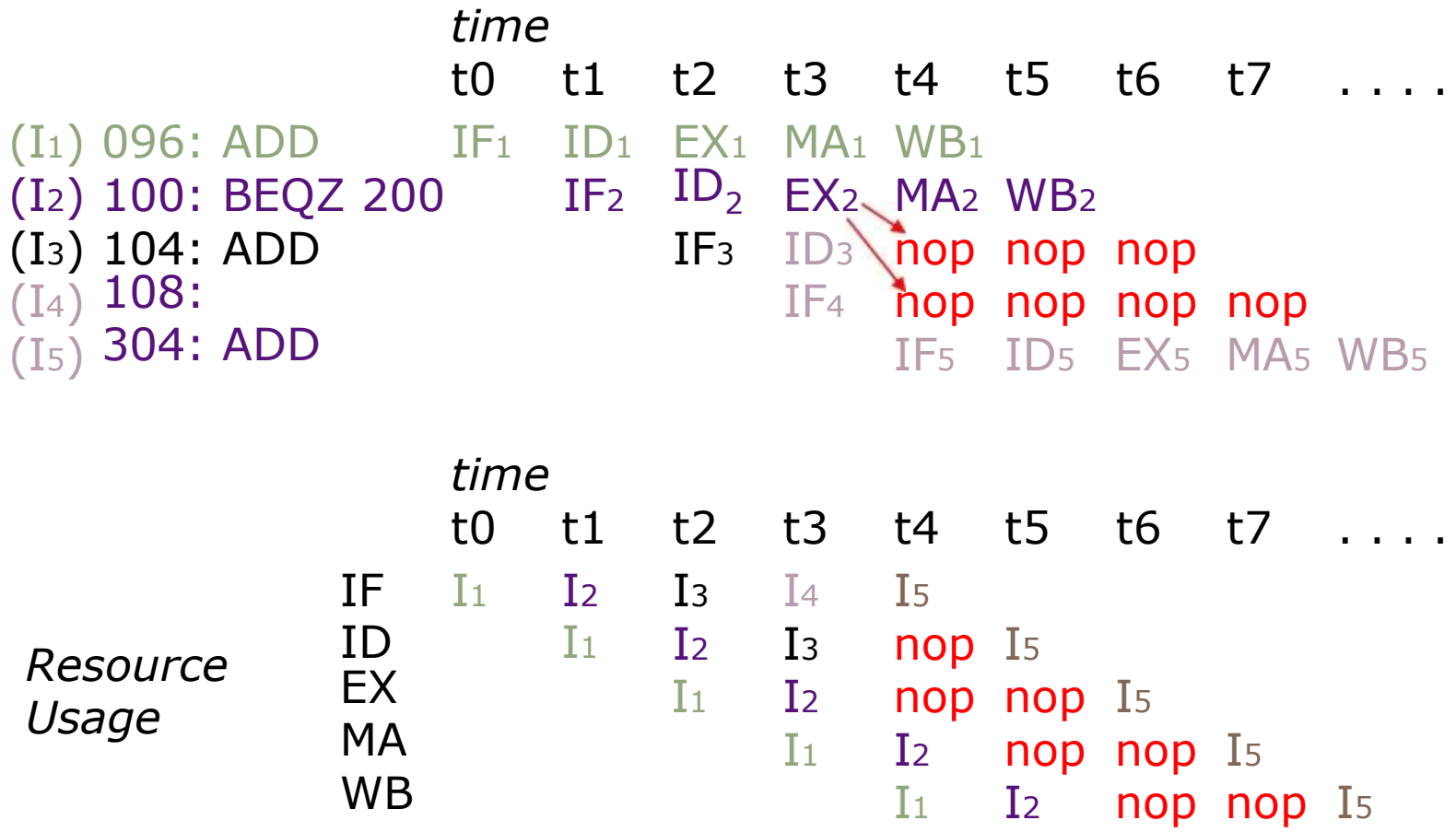
Conditional Branches (And the notion of Kill)

I_1	096	ADD
I_2	100	BEQZ r1 200
I_3	104	ADD
I_4	304	ADD

Branch condition is not known until the execute stage

what action should be taken in the decode stage?

Again? What Else ? Delay Slots?



Welcome to the World of Predictors

Impact of a Branch?

Average dynamic instruction mix of SPEC CPU 2017 [Limaye and Adegbiya , ISPASS'18]:

	SPECint	SPECfp
Branches	19 %	11 %
Loads	24 %	26 %
Stores	10 %	7 %
Other	47 %	56 %

SPECint17: *perlbench, gcc, mcf, omnetpp, xalancbmk, x264, deepsjeng, leela, exchange2, xz*
SPECfp17: *bwaves, cactus, lbm, wrf, pop2, imagick, nab, fotonik3d, roms*

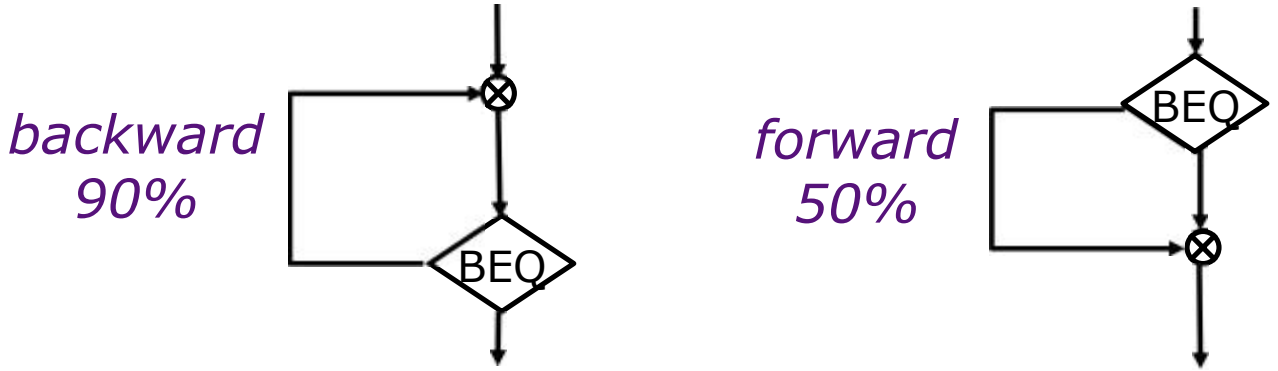
What is the average run length *between branches?*

Roughly 5-10 instructions

Branches and Jumps

Instruction	*Taken known?*	*Target known?*
J	After Inst. Decode	After Inst. Decode
BEQZ/BNEZ	After Inst. Execute	After Inst. Execute

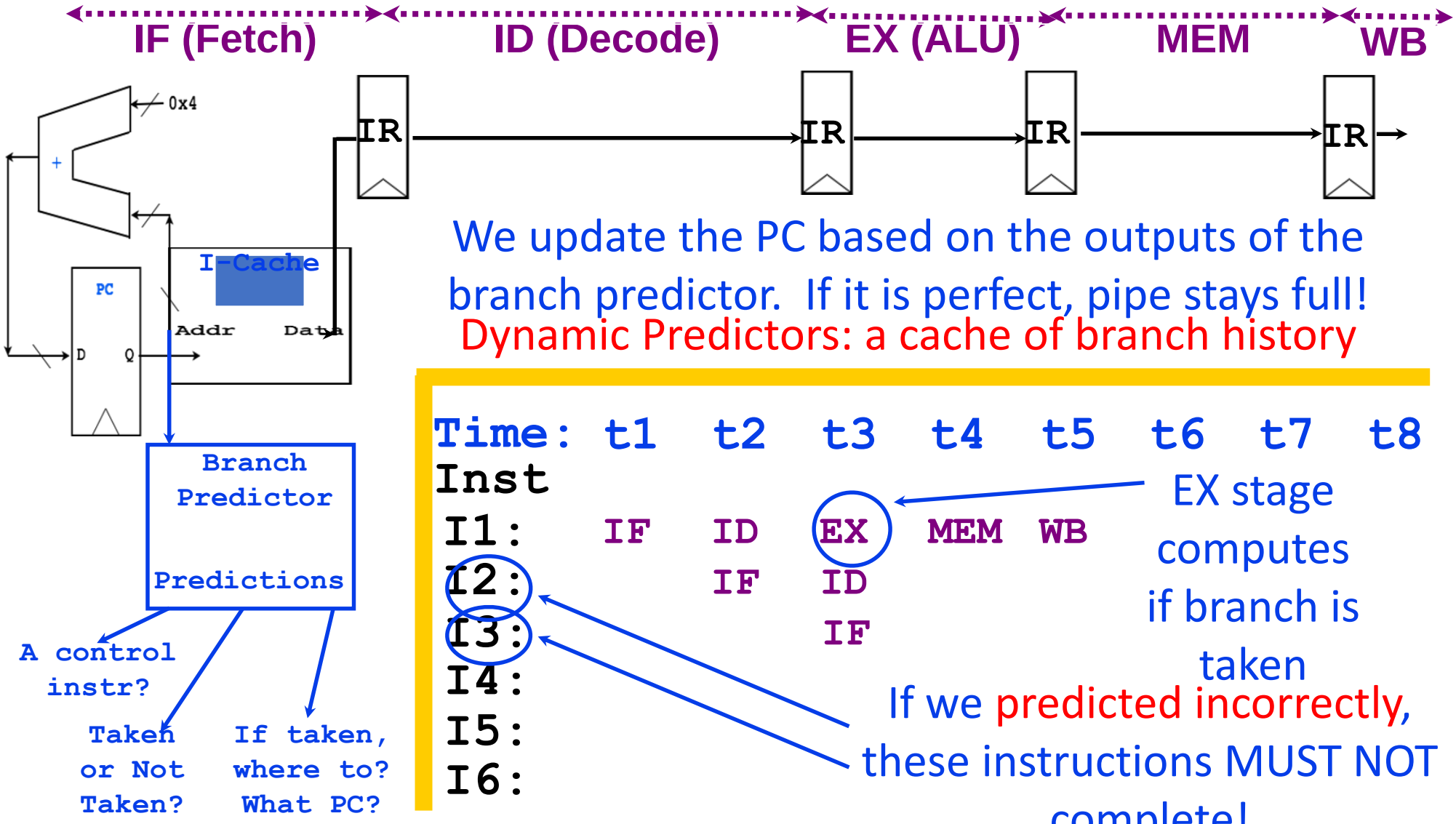
Static Branch Prediction



ISA can attach preferred direction semantics to branches,
e.g., Motorola MC88110
bne0 *(preferred taken)* beq0 *(not taken)*

ISA can allow arbitrary choice of statically predicted direction,
e.g., HP PA-RISC, Intel IA-64
typically reported as ~80% accurate

Dynamic Branch Predictor



We update the PC based on the outputs of the branch predictor. If it is perfect, pipe stays full!

Dynamic Predictors: a cache of branch history

Time:	t1	t2	t3	t4	t5	t6	t7	t8
Inst								
I1:	IF	ID	EX	MEM	WB			
I2:		IF	ID					
I3:			IF					
I4:								
I5:								
I6:								

EX stage computes if branch is taken

If we predicted incorrectly, these instructions MUST NOT complete!