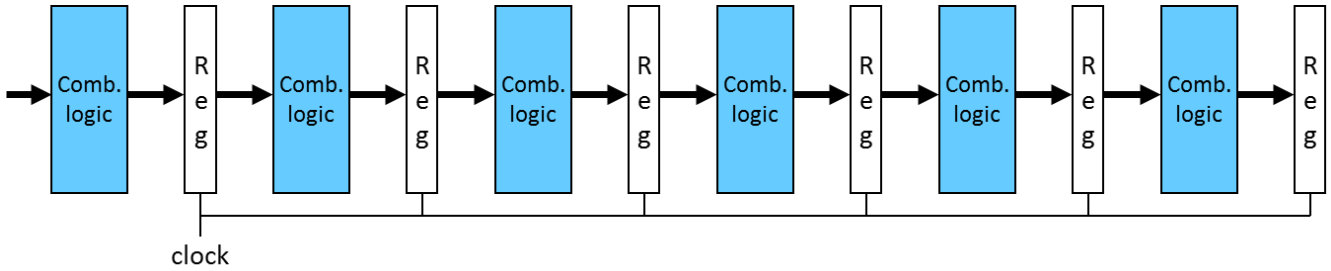
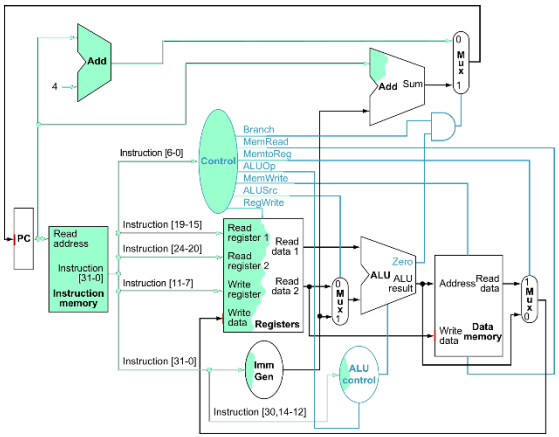
Processor Architecture

Principles of Pipelining



Module Outline

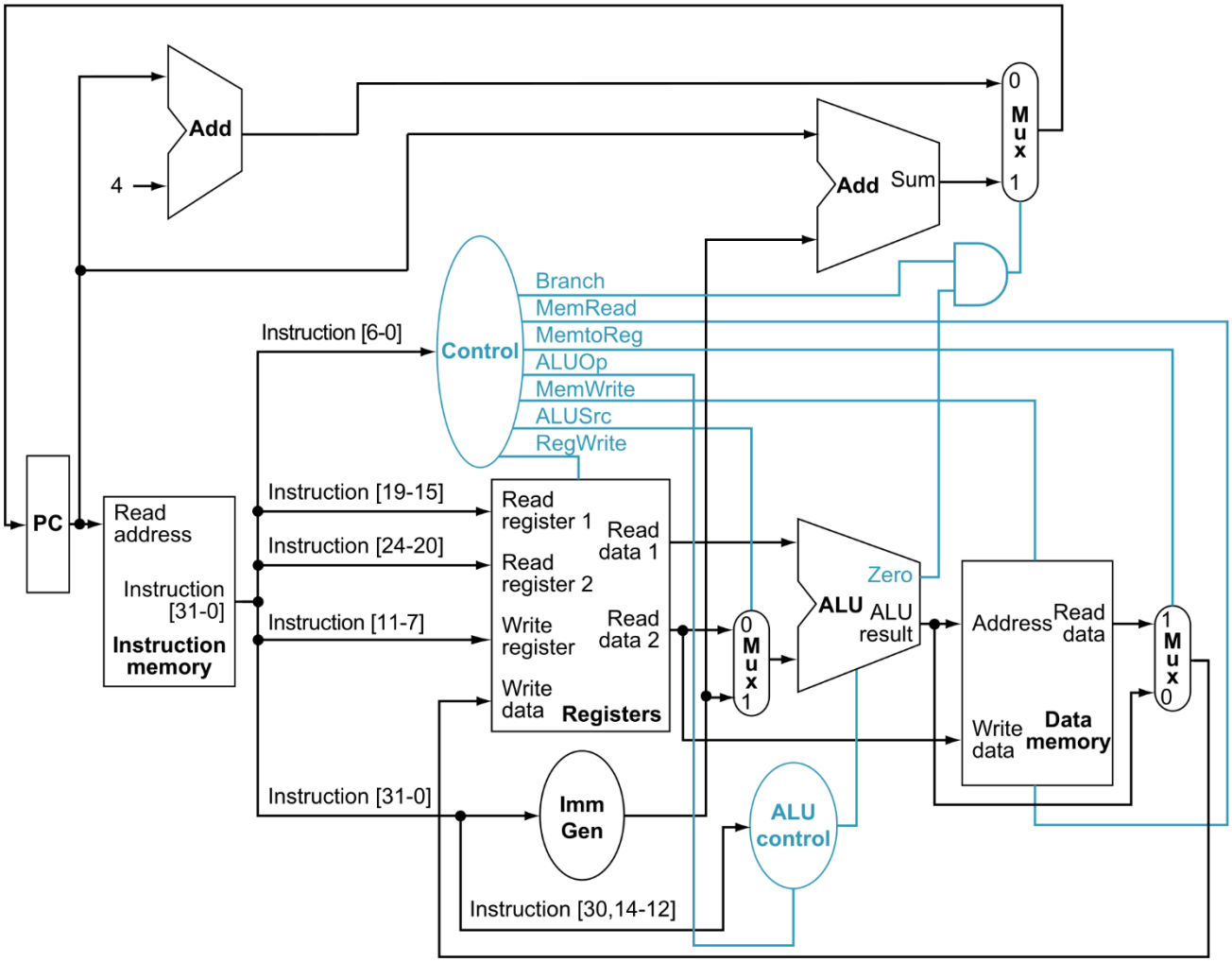
- Limitations of Single-Cycle Implementations
- Parallel Architectures
- Limitations and Hurdles of Pipelining
  - Structural Hazards
  - Data Hazards
  - Control Hazards
- Module Summary



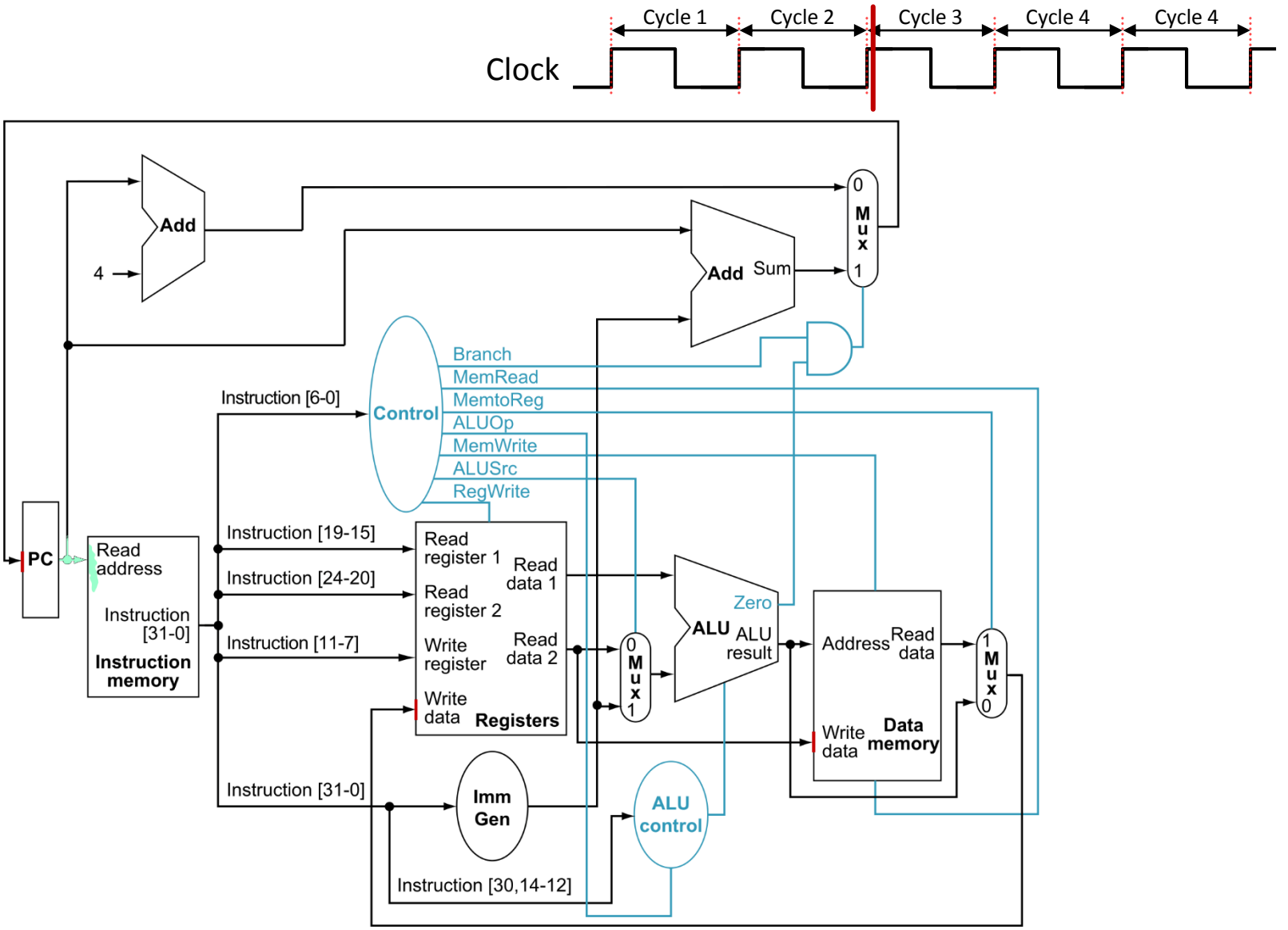
Limitations of Single-Cycle Implementations

Limitations of Single-Cycle Implementations

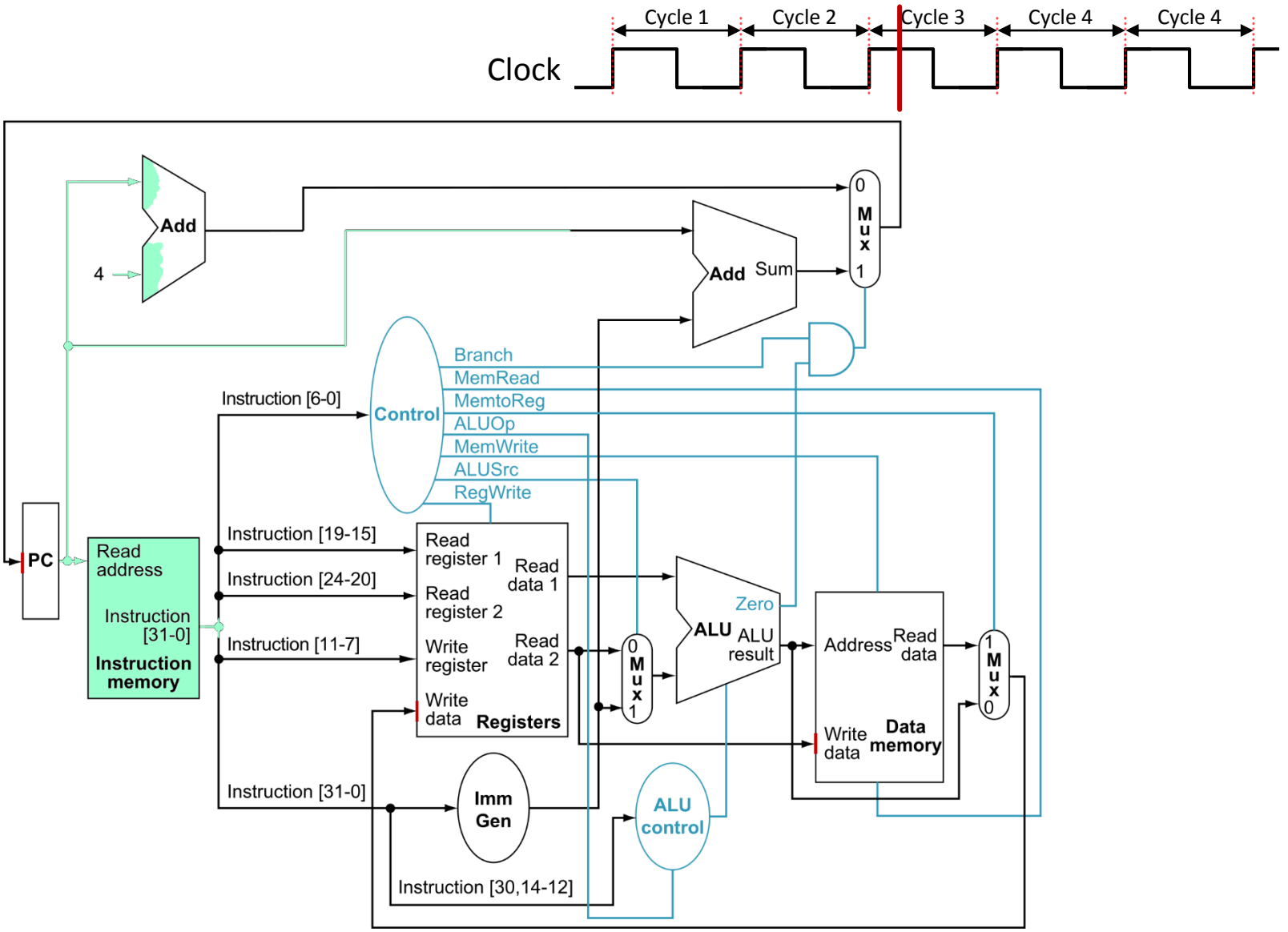
- **Single-Cycle (Simplified) RISC-V Implementation**



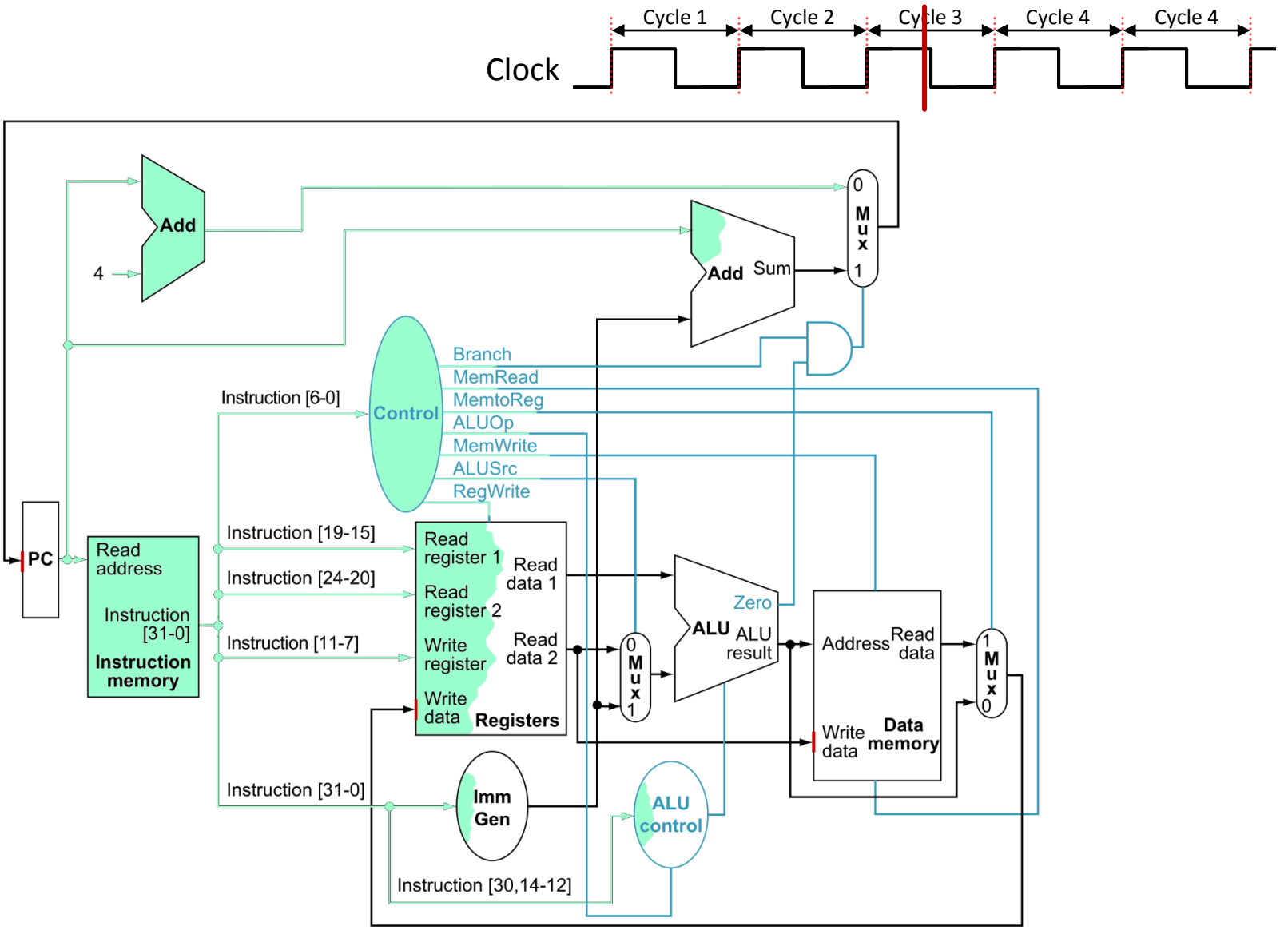
Single-Cycle (Simplified) RISC-V Implementation



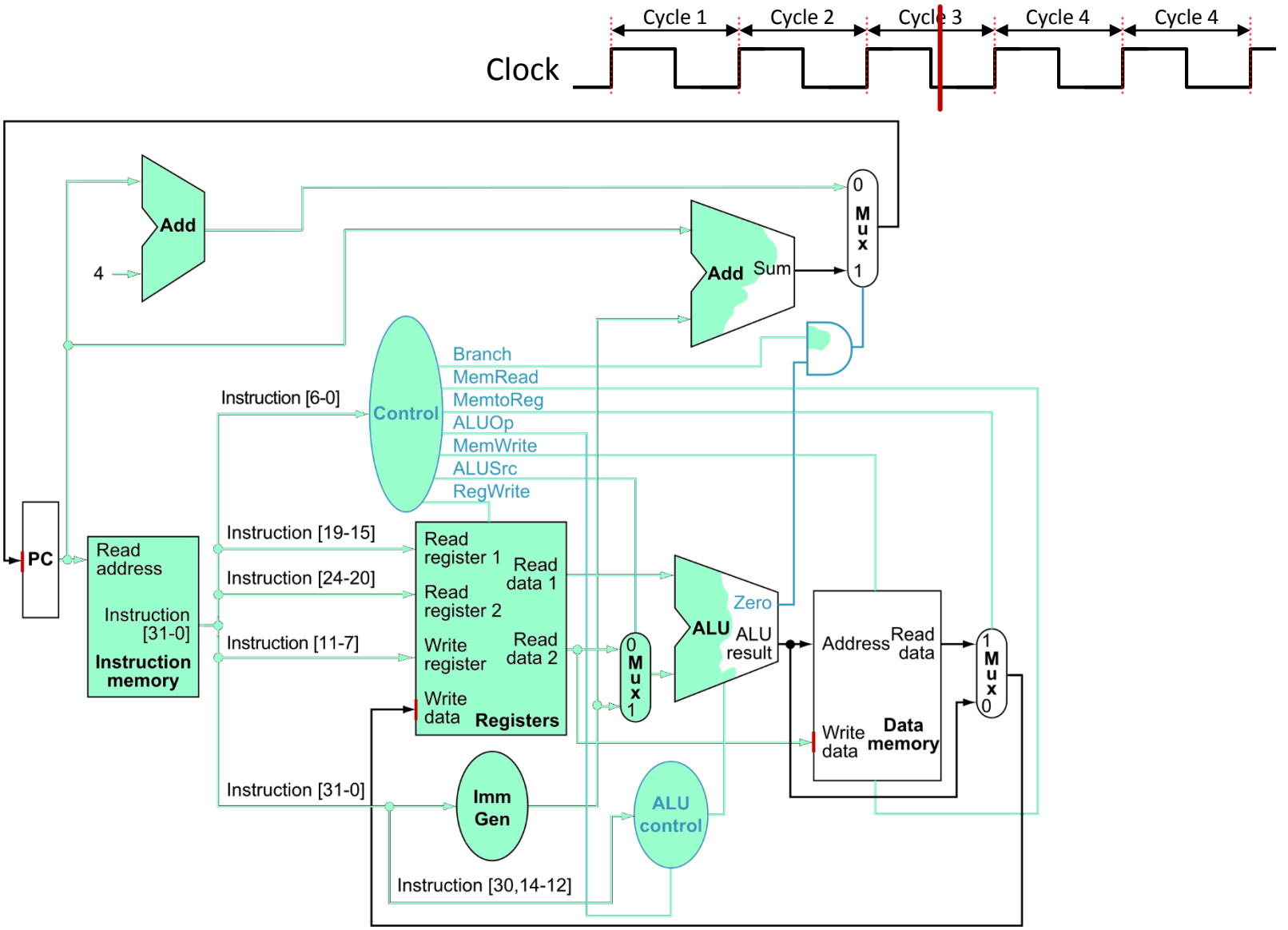
Single-Cycle (Simplified) RISC-V Implementation



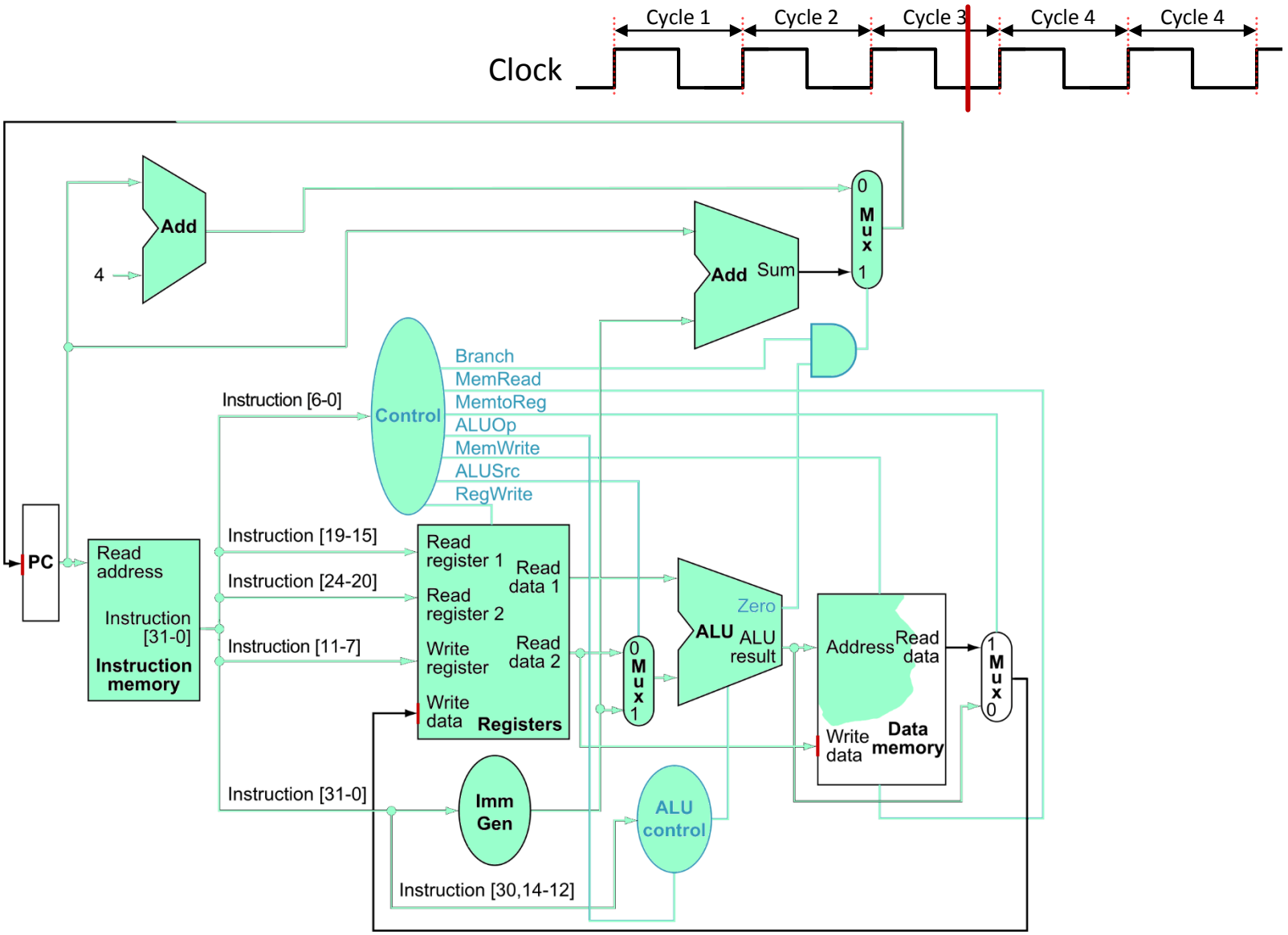
Single-Cycle (Simplified) RISC-V Implementation



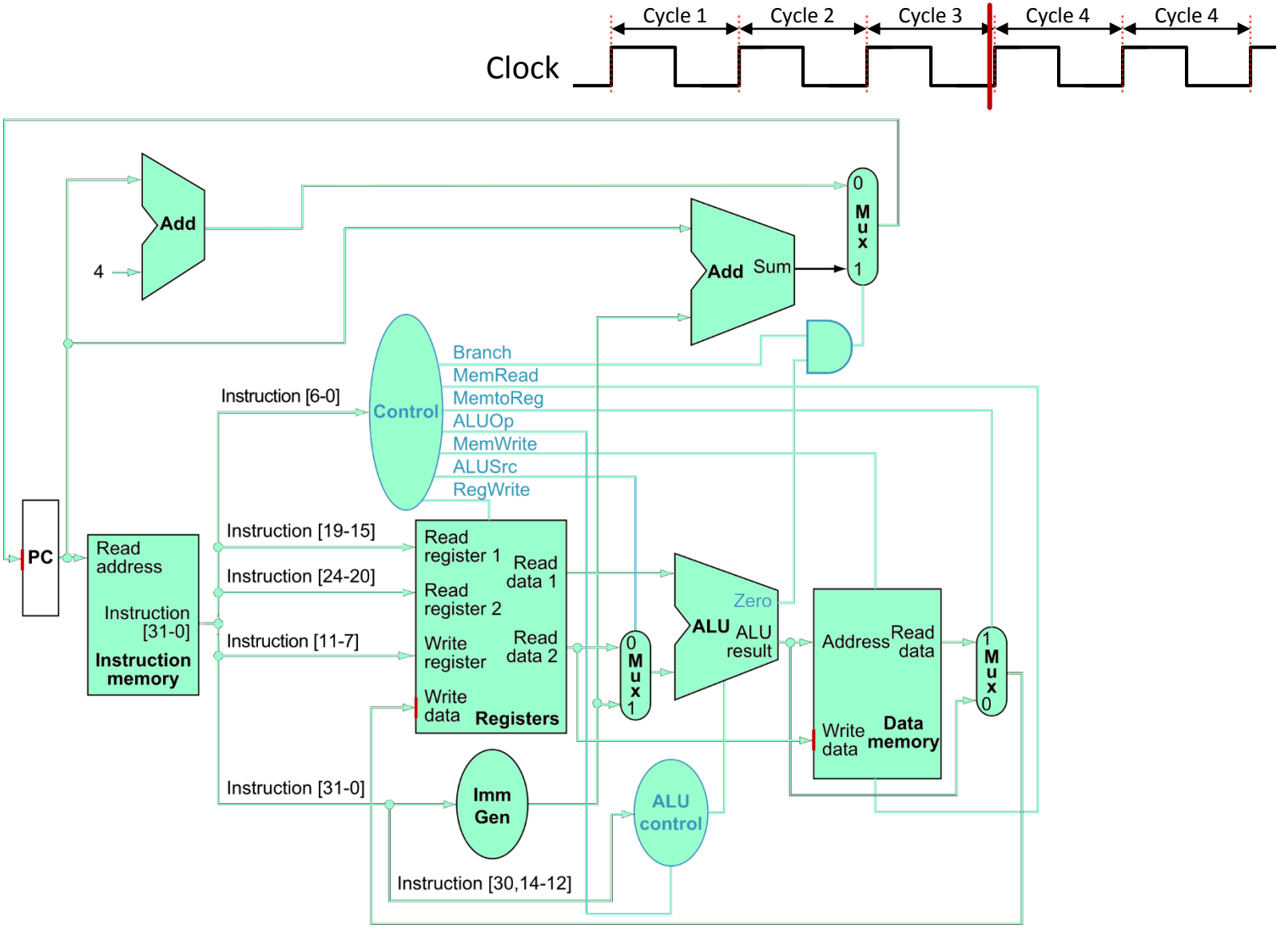
Single-Cycle (Simplified) RISC-V Implementation



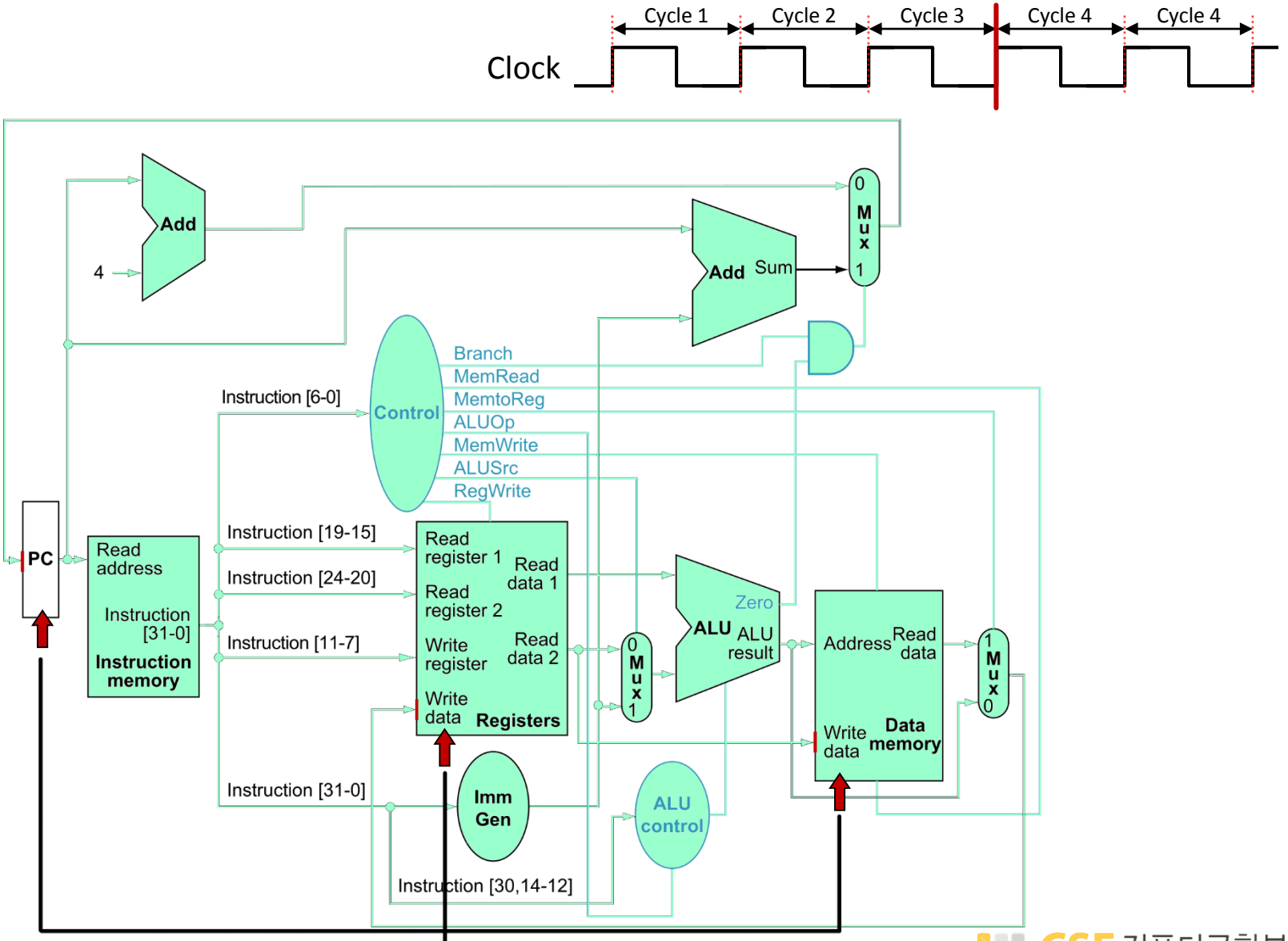
Single-Cycle (Simplified) RISC-V Implementation



Single-Cycle (Simplified) RISC-V Implementation

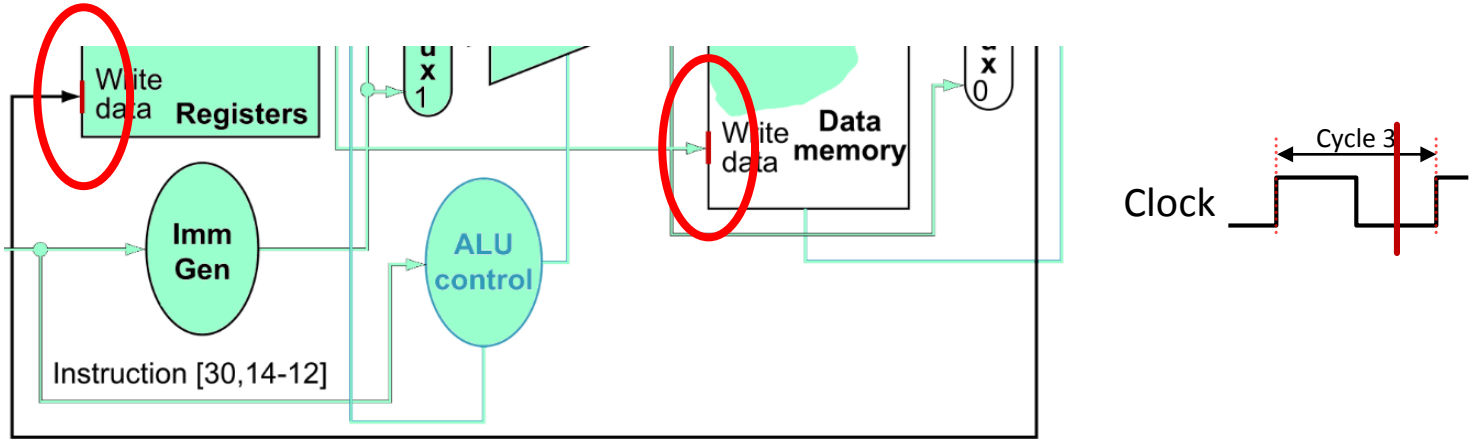


Single-Cycle (Simplified) RISC-V Implementation



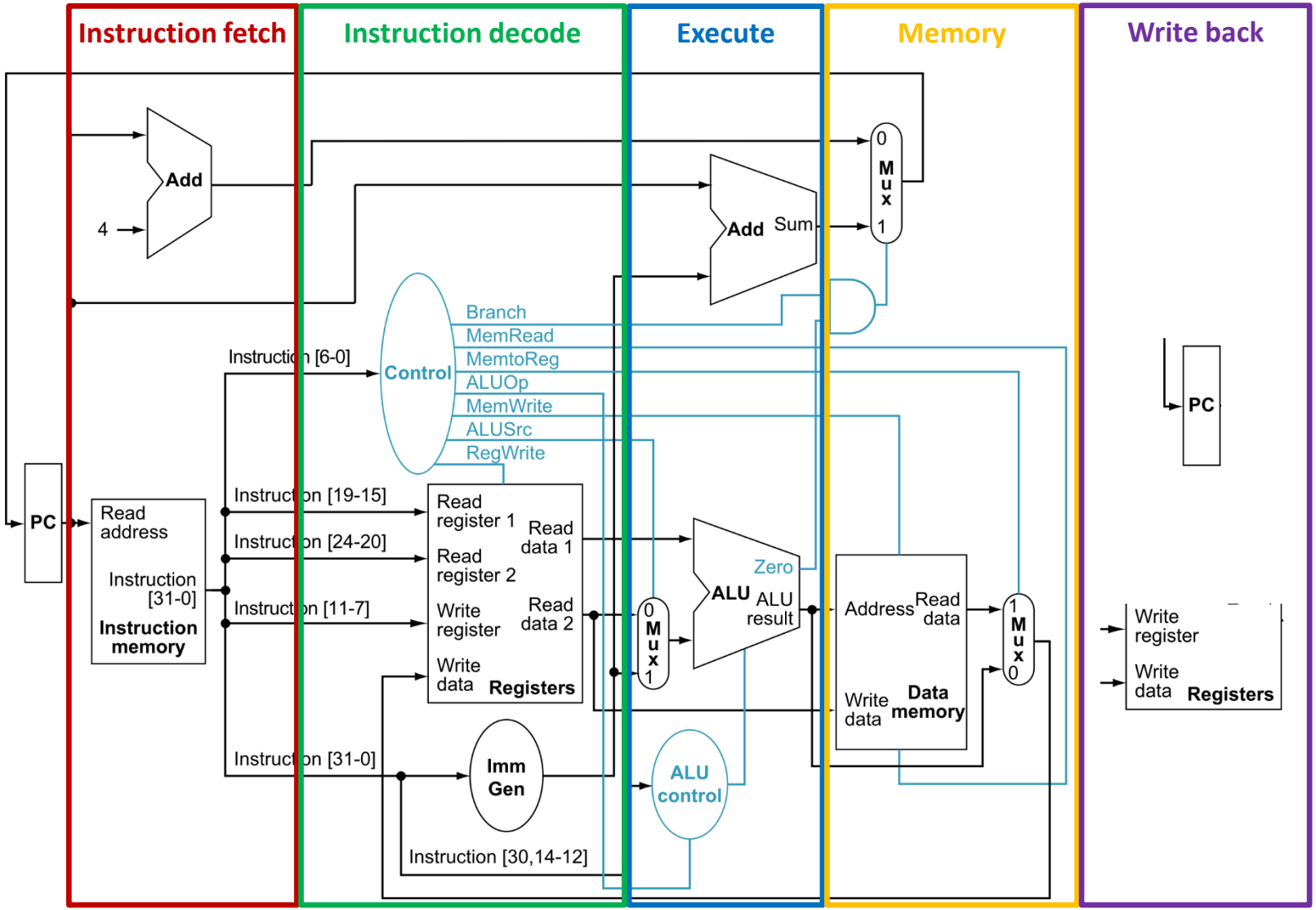
Single-Cycle (Simplified) RISC-V Implementation

- **Observations**
  - Signals take a long time to propagate through entire circuit
  - Next clock cannot be triggered before slowest signal has reached the “gate” (write port) of its register



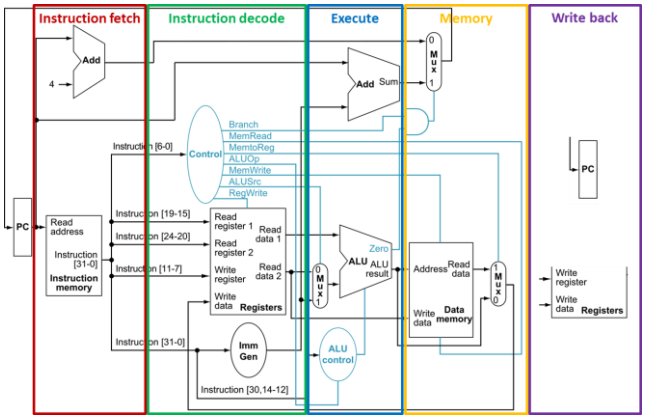
  - Critical path = of all possible paths through the circuit the one with the longest delay
  - Clock frequency is limited by the delay of the critical path

Single-Cycle (Simplified) RISC-V Implementation



Single-Cycle (Simplified) RISC-V Implementation

- **Latencies of different stages**
  - Instruction fetch: 200 ps
  - Instruction decode: 100 ps
  - Execute: 200 ps
  - Memory: 200 ps
  - Write back: 100 ps



critical path

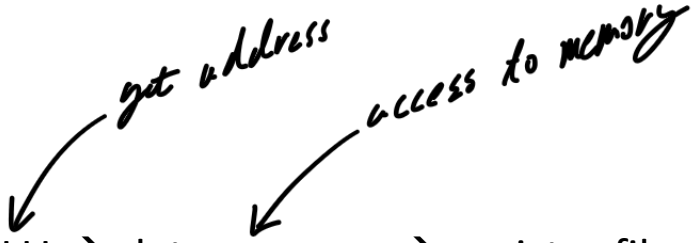
Instruction class	IF	ID	EX	MEM	WB	Total
Load word	200	100	200	200	100	**800**
Store word	200	100	200	200		**700**
R-format	200	100	200		100	**600**
Branch	200	100	200			**500**

Max. clock frequency f_{max} = 1/800 ps = 1/(800x10^{-12} sec)=1.25x10^{9} sec^{-1}= 1.25GHz

Max. throughput = 1.25 GIPS (Giga Instruction per Second)

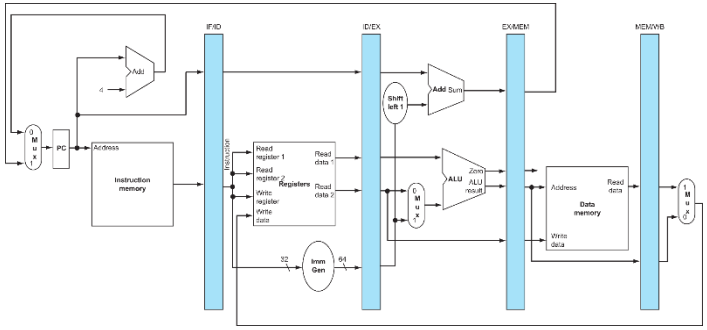
Single-Cycle Performance Issues

- Longest delay determines clock period
  - Critical path: load instruction
    - Instruction memory → register file → ALU → data memory → register file
  - Not feasible to vary period for different instructions



- Violates design principle "Make the common case fast" ∴ ∀ case is same speed

- Let's improve performance by pipelining



Parallel Architectures

Different Types of Parallelism

- **Sequential (no parallelism)**

Different Types of Parallelism

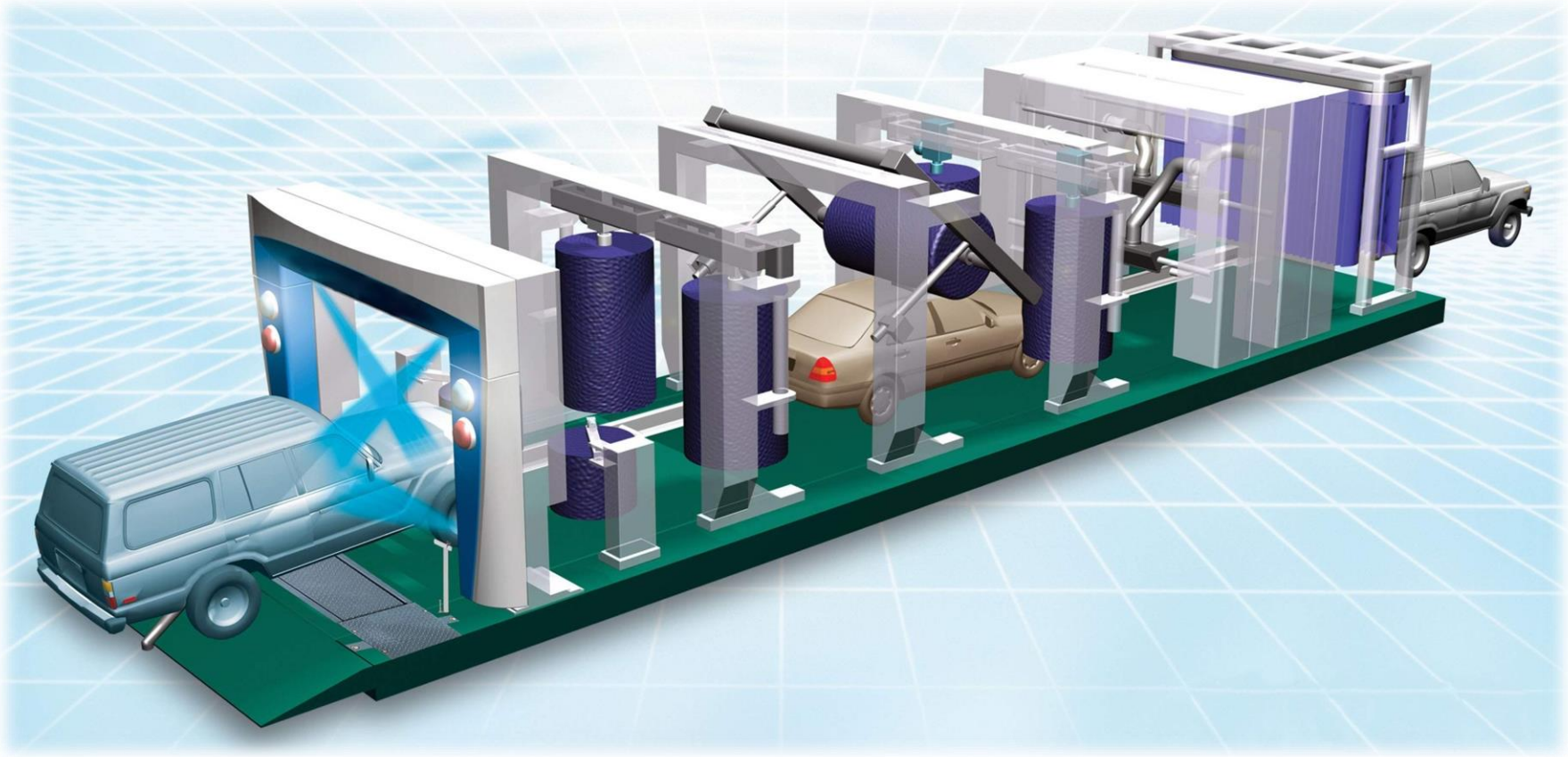
- **Coarse-grained parallelism**

Different Types of Parallelism

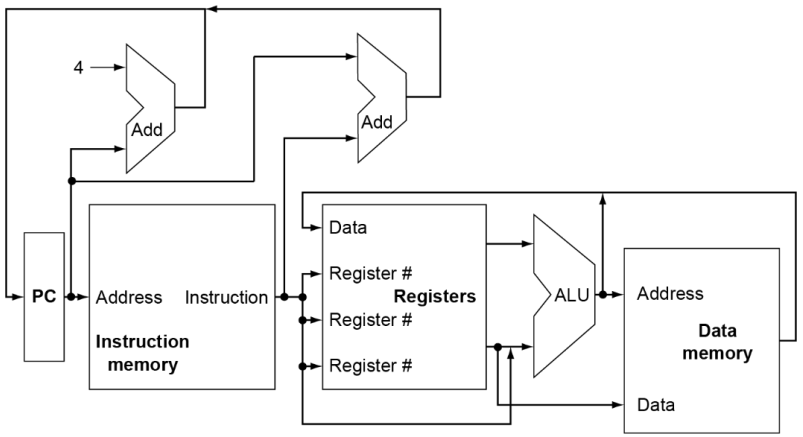
- **Fine-grained parallelism**

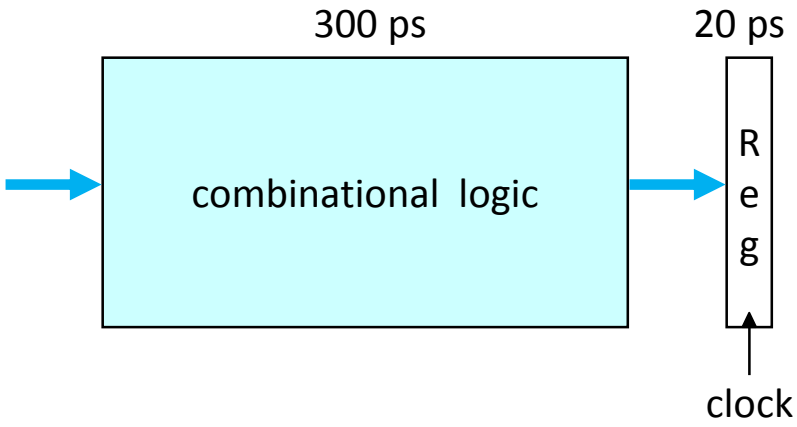
Different Types of Parallelism

- **Pipeline parallelism**

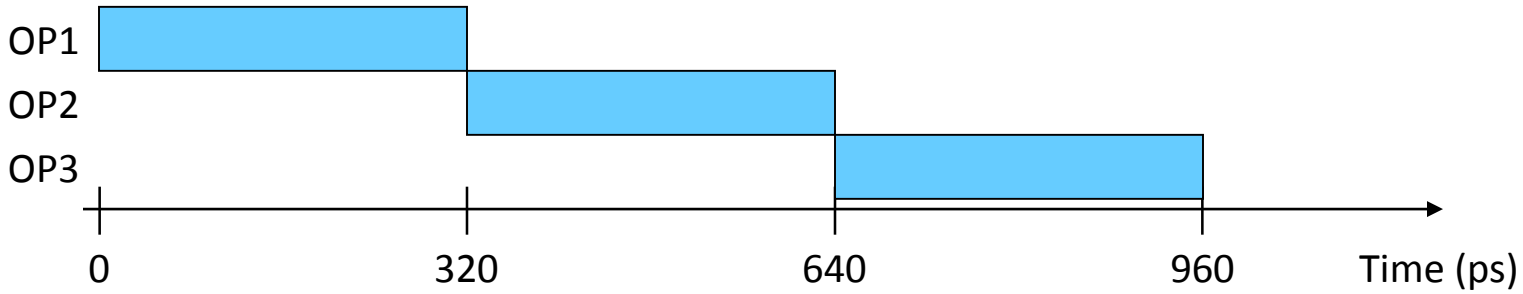


Single-Cycle Architectures



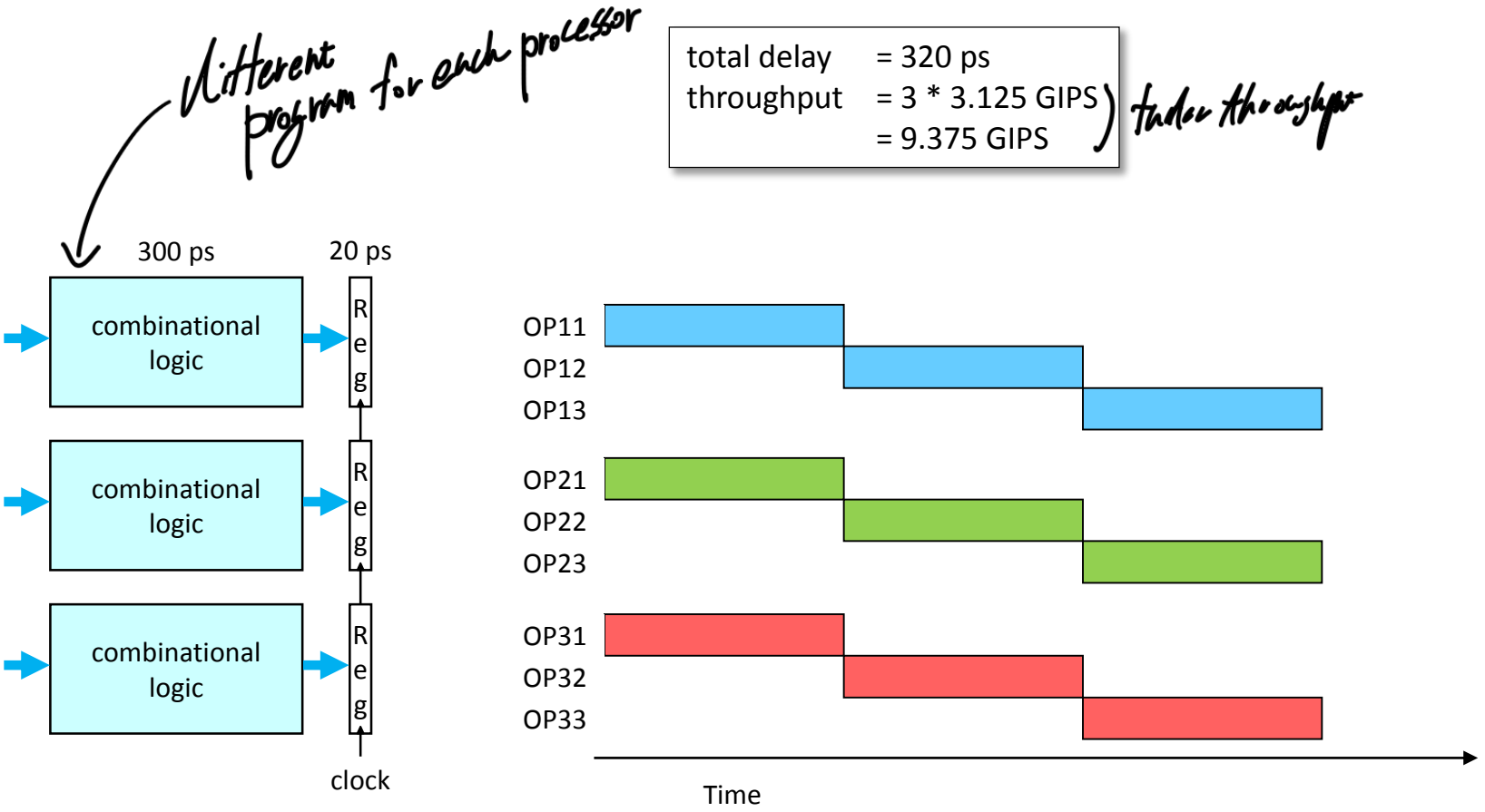


total delay = 320 ps
throughput = 3.125 GIPS



Parallel Architectures

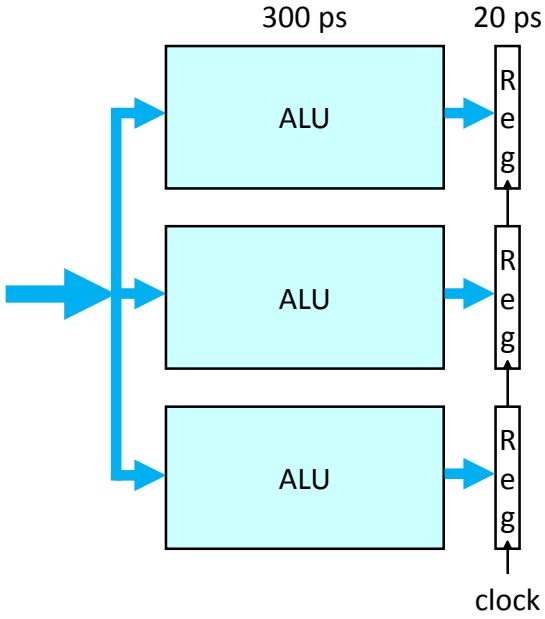
- **Coarse-grained parallelism – multiprocessor, multicore**



Parallel Architectures

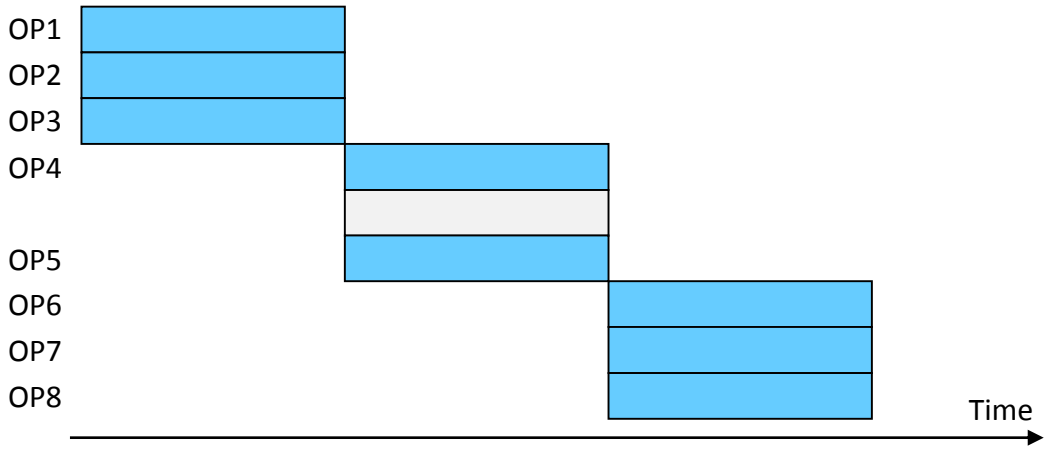
- **Fine-grained parallelism – superscalar, VLIW processors**

run multiple instruction of single program at a same time



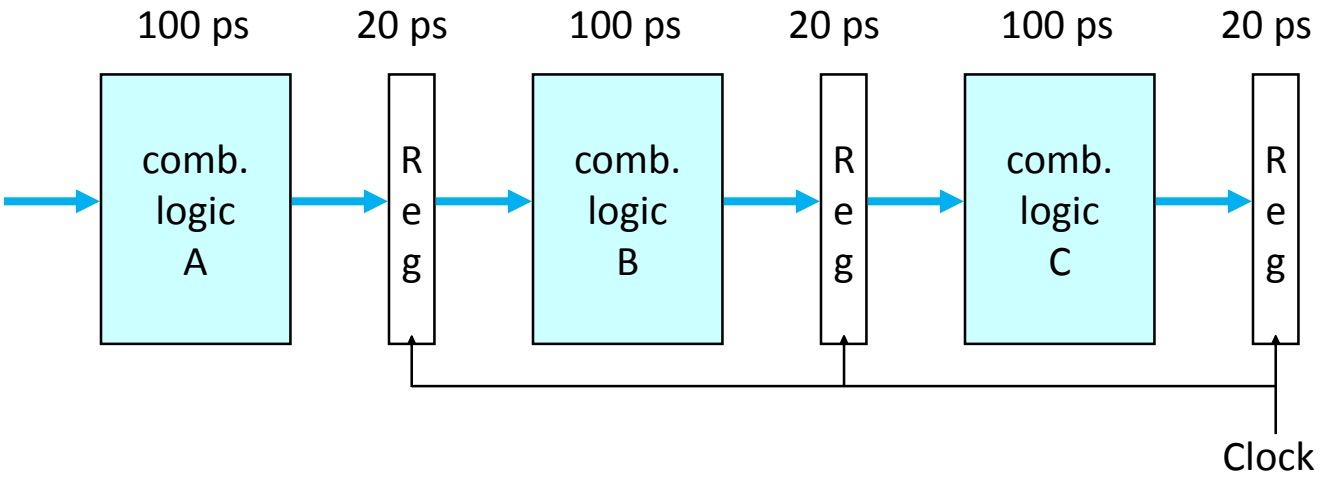
total delay = 320 ps
throughput <= 3 * 3.125 GIPS
<= 9.375 GIPS

) throughput faster



Pipelined Architectures

- **Pipeline parallelism – (almost) all modern processors**
  - Split work up into several independent phases
  - Run independent phases in parallel



delay ↑

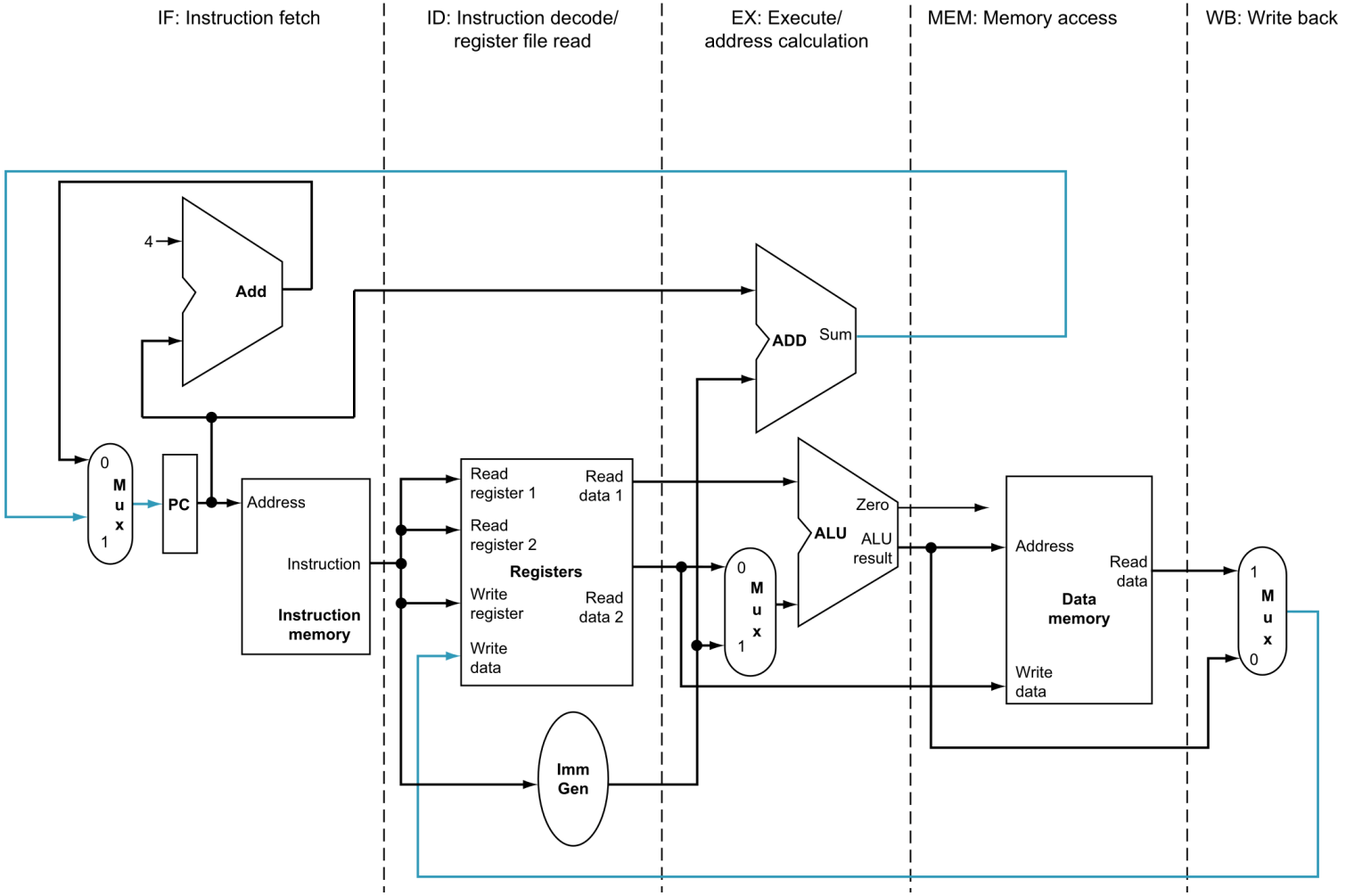
total delay = 360 ps
throughput = 8.33 GIPS

thruput ↓

- Can begin new operation as soon as previous has passed through stage A
- But: overall latency (for one operation) *increases*

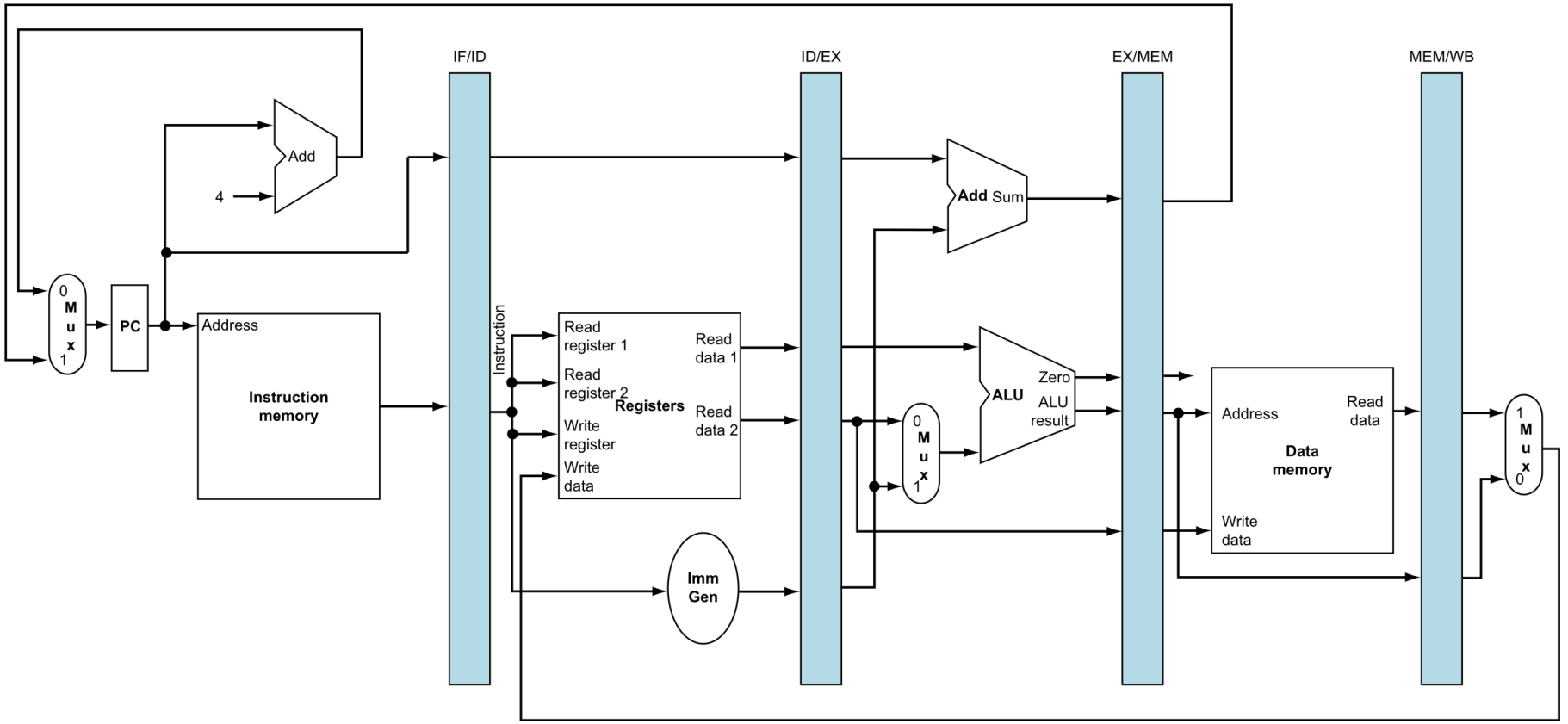
Pipelined Architectures

- **Logical separation into stages**



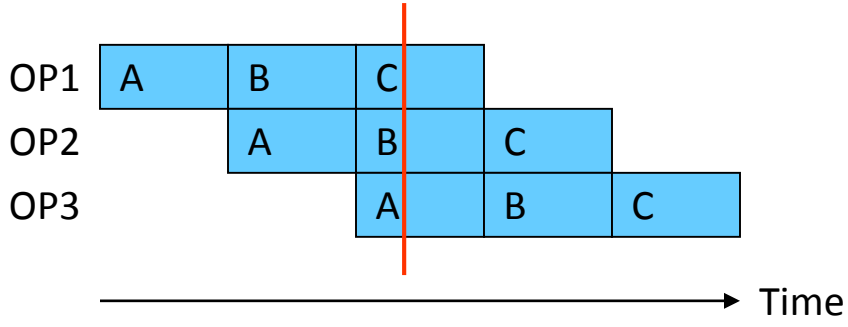
Pipelined Architectures

- **Physical separation into stages**
  - Separate stages with registers

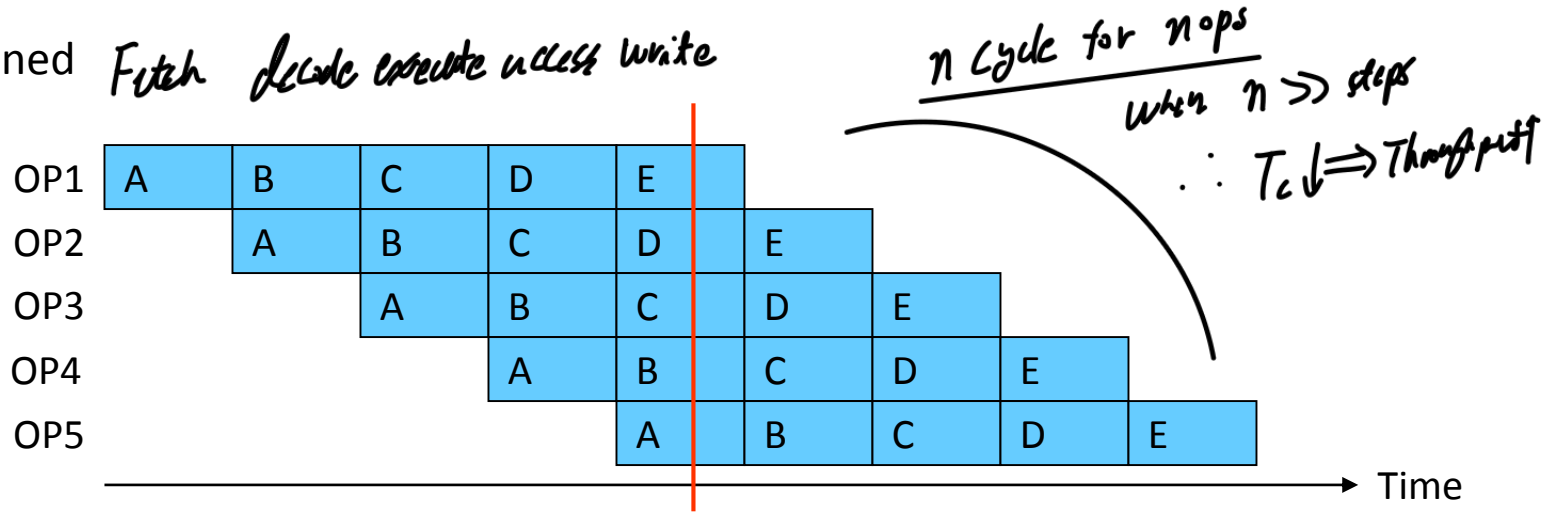


Pipeline Stages

- 3-way pipelined

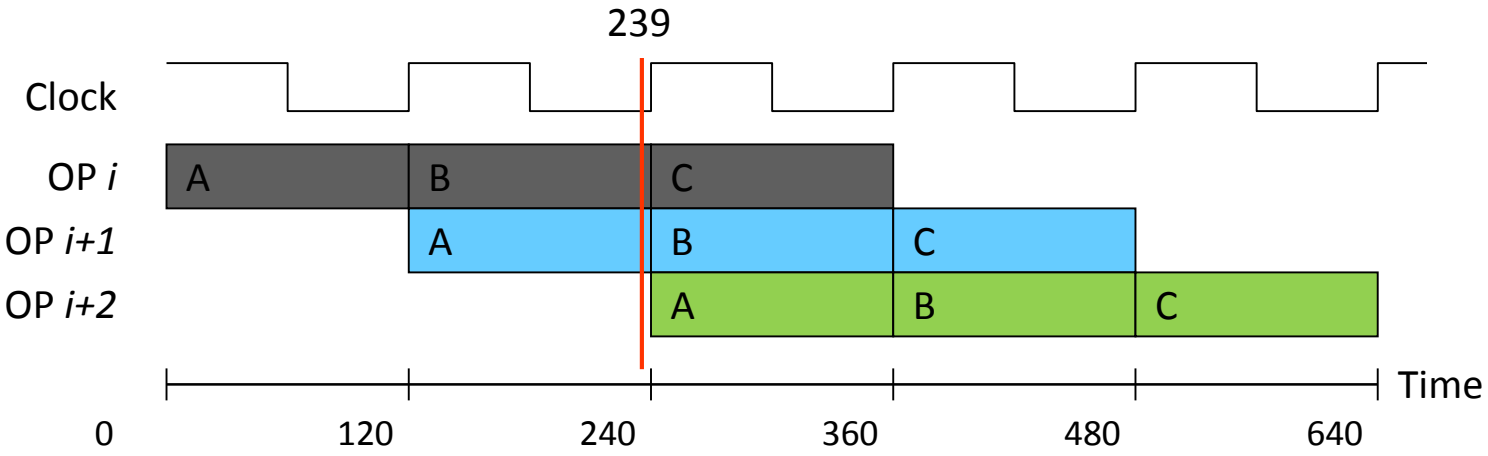


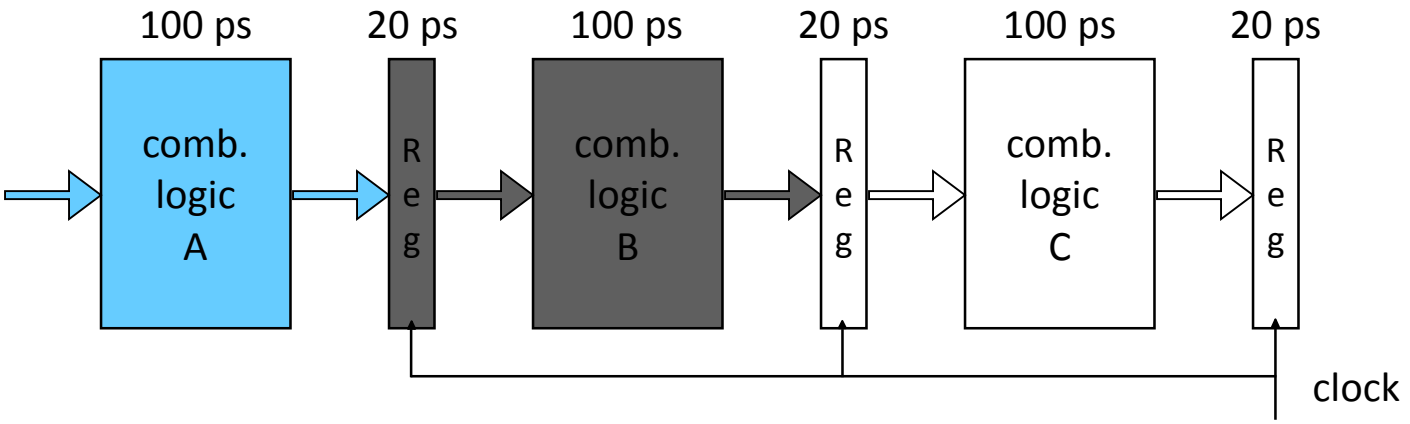
- 5-way pipelined



Pipeline Operation

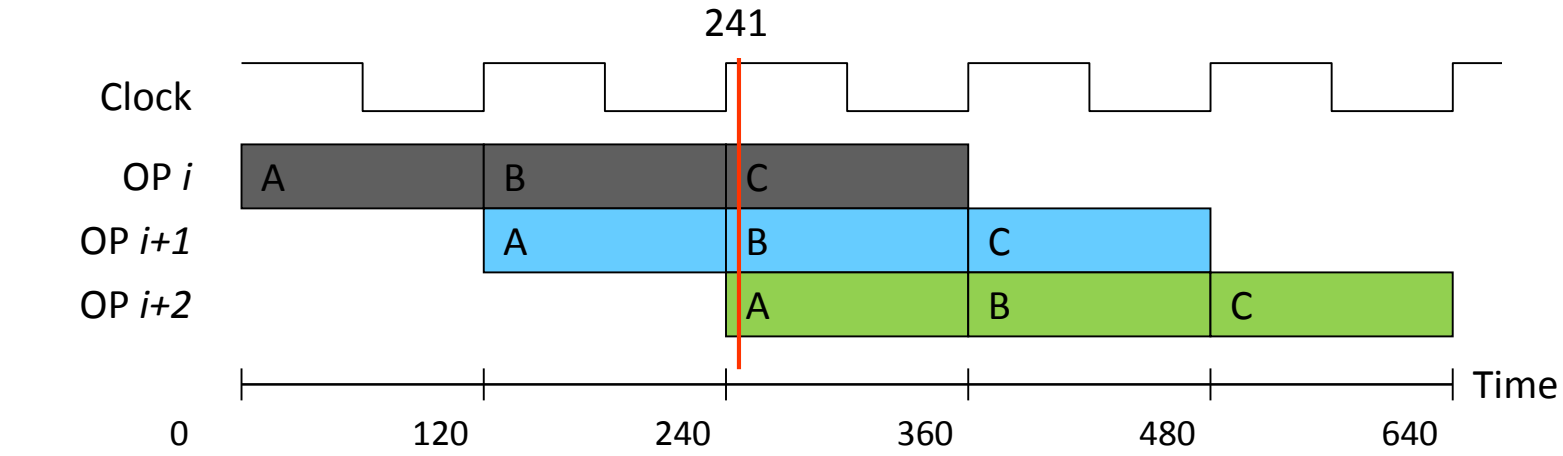
- Immediately before a new clock cycle starts

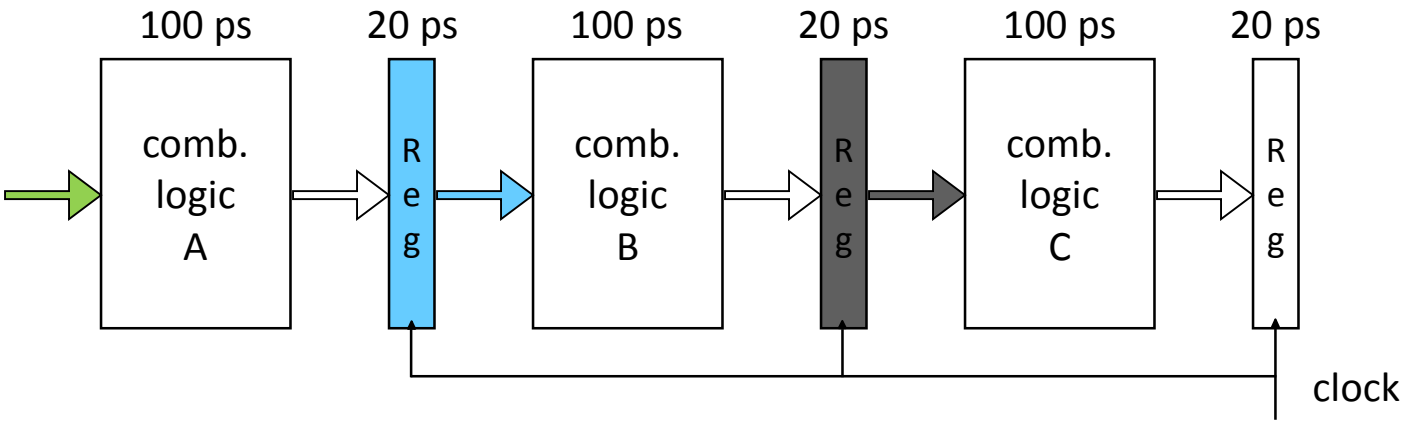




Pipeline Operation

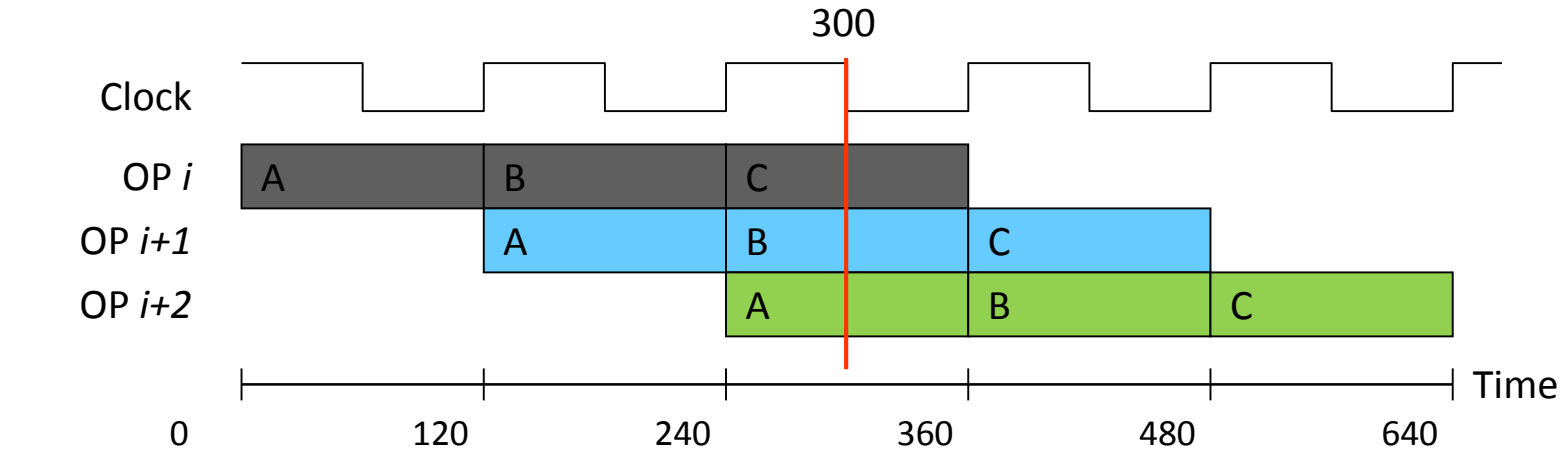
- Immediately after a new clock cycle has started

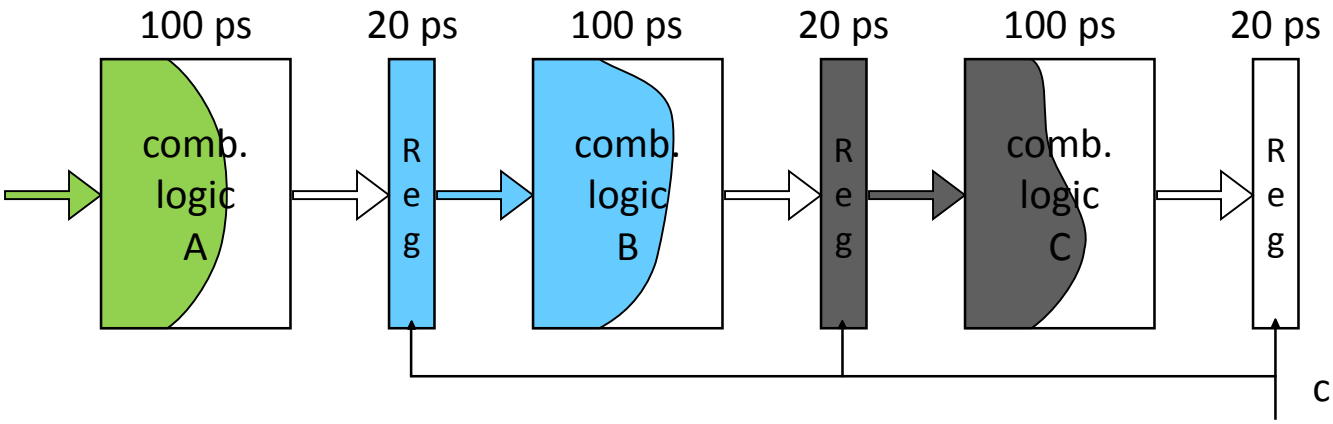




Pipeline Operation

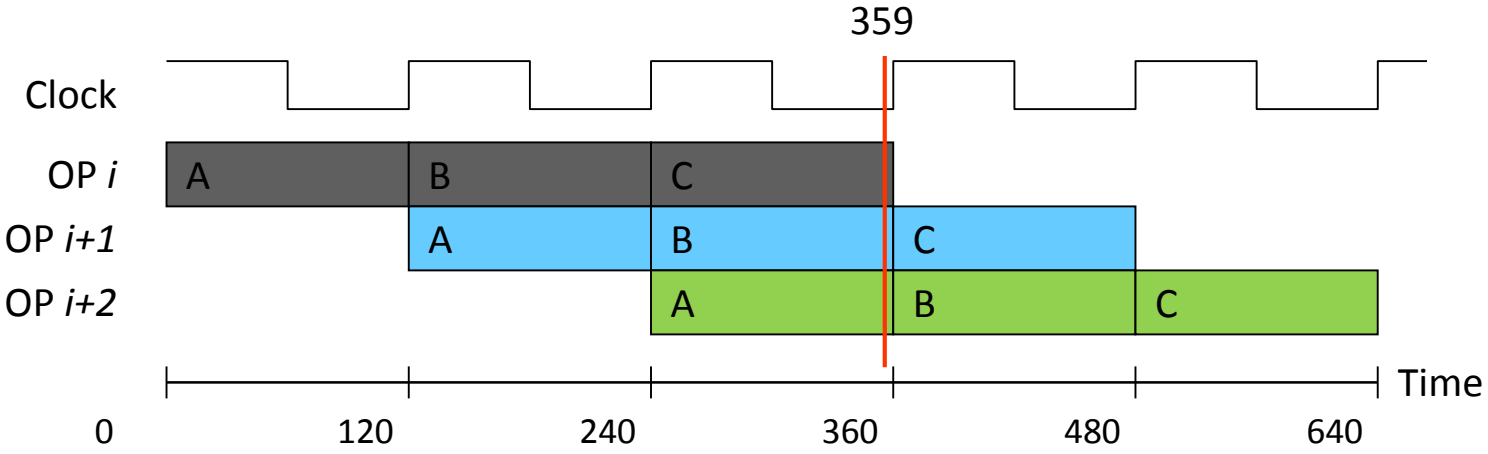
- In the middle of a clock cycle

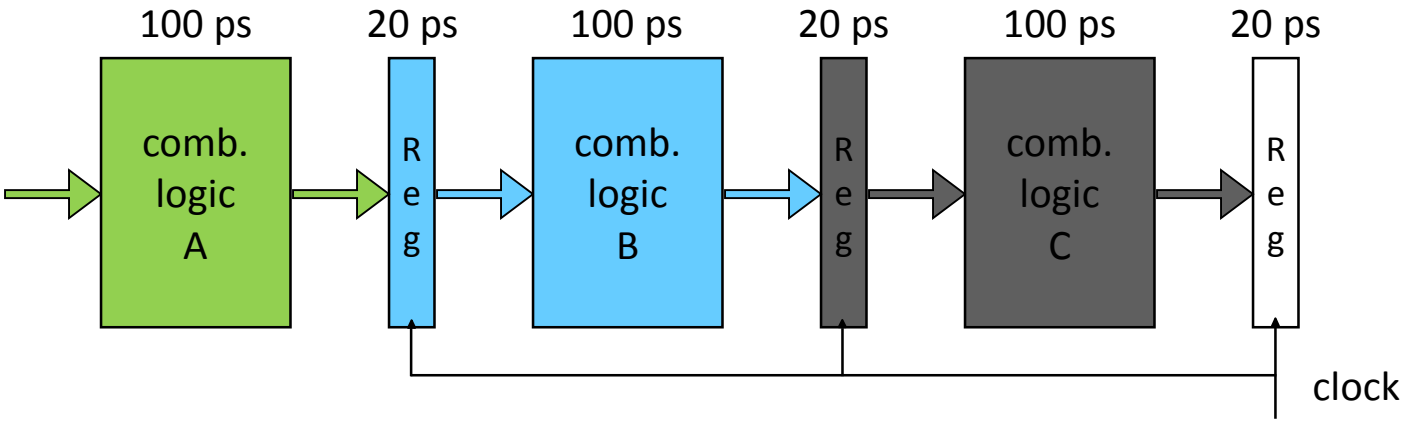




Pipeline Operation

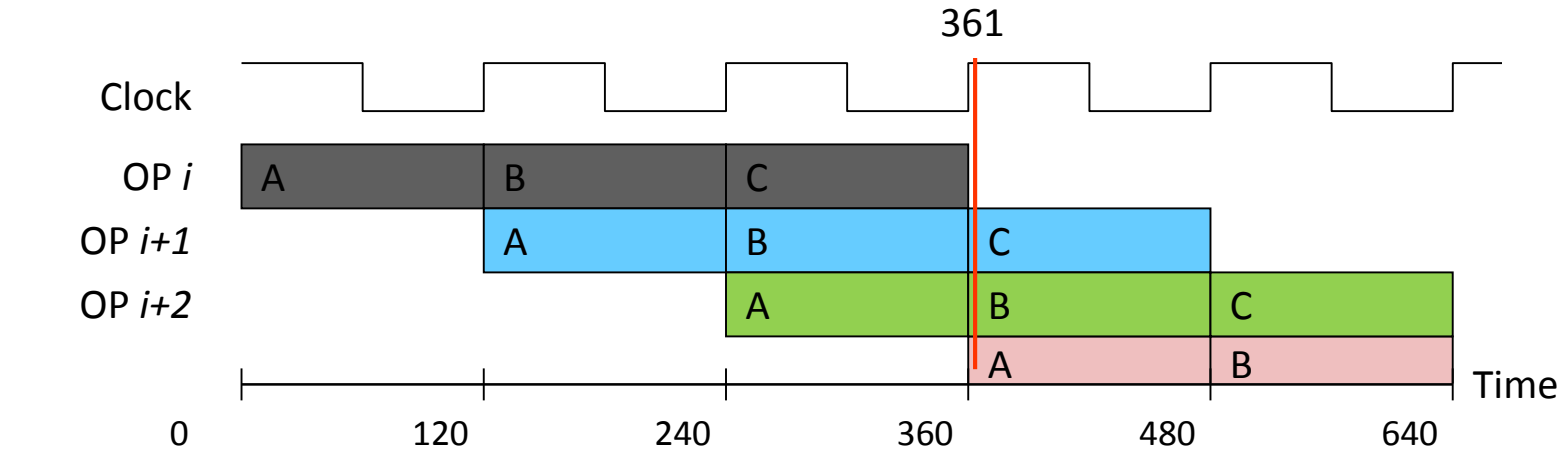
- Immediately before a clock cycle ends

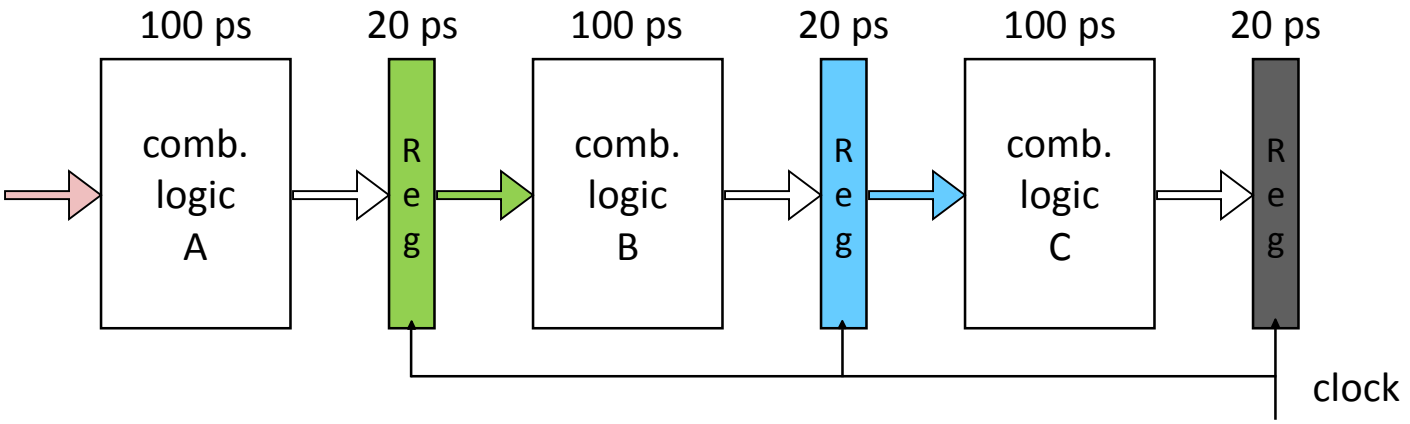




Pipeline Operation

- Immediately after next clock cycle has started





Basic Pipeline

- **Pipeline Stages**
  1. Instruction fetch step — `IF`
  2. Instruction decode/register fetch step — `ID`
  3. Execution/effective address step — `EX`
  4. Memory access — `MEM`
  5. Register write-back step — `WB`

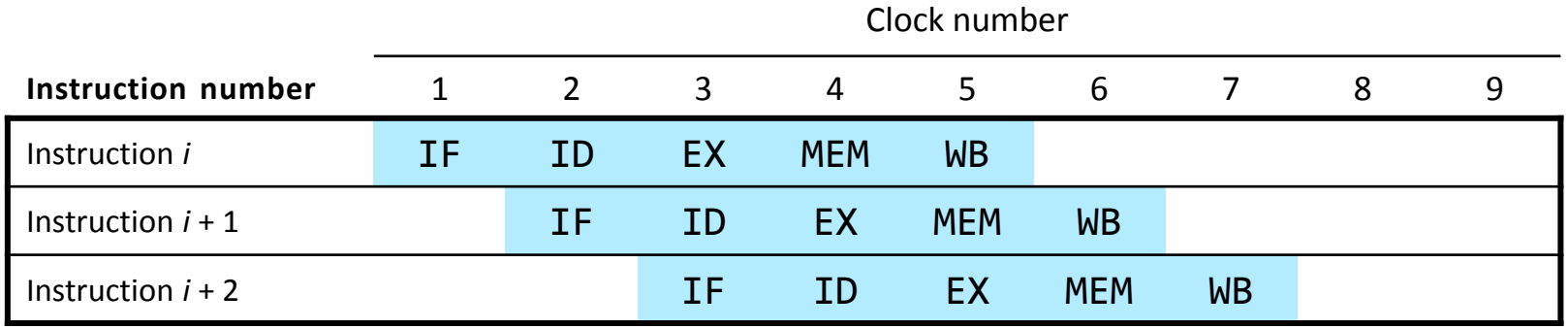
- **Pipeline Operation**

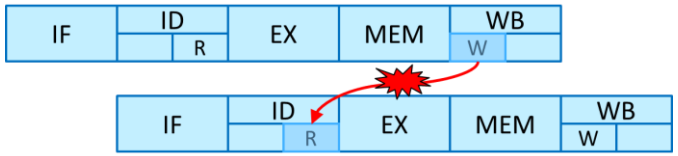
	Clock								
Instruction number	1	2	3	4	5	6	7	8	9
Instruction *i*	IF	ID	EX	MEM	WB				
Instruction *i* + 1		IF	ID	EX	MEM	WB			
Instruction *i* + 2			IF	ID	EX	MEM	WB		
Instruction *i* + 3				IF	ID	EX	MEM	WB	
Instruction *i* + 4					IF	ID	EX	MEM	WB

Basic Pipeline Performance of an N-Stage Pipeline

- **Latency**
  - 1 instruction — N * clock delay
  - k instructions — (k + N – 1) * clock delay

- **Throughput**
  - k instructions — (k + N – 1) * clock delay / k
  - ∞ instructions — clock delay





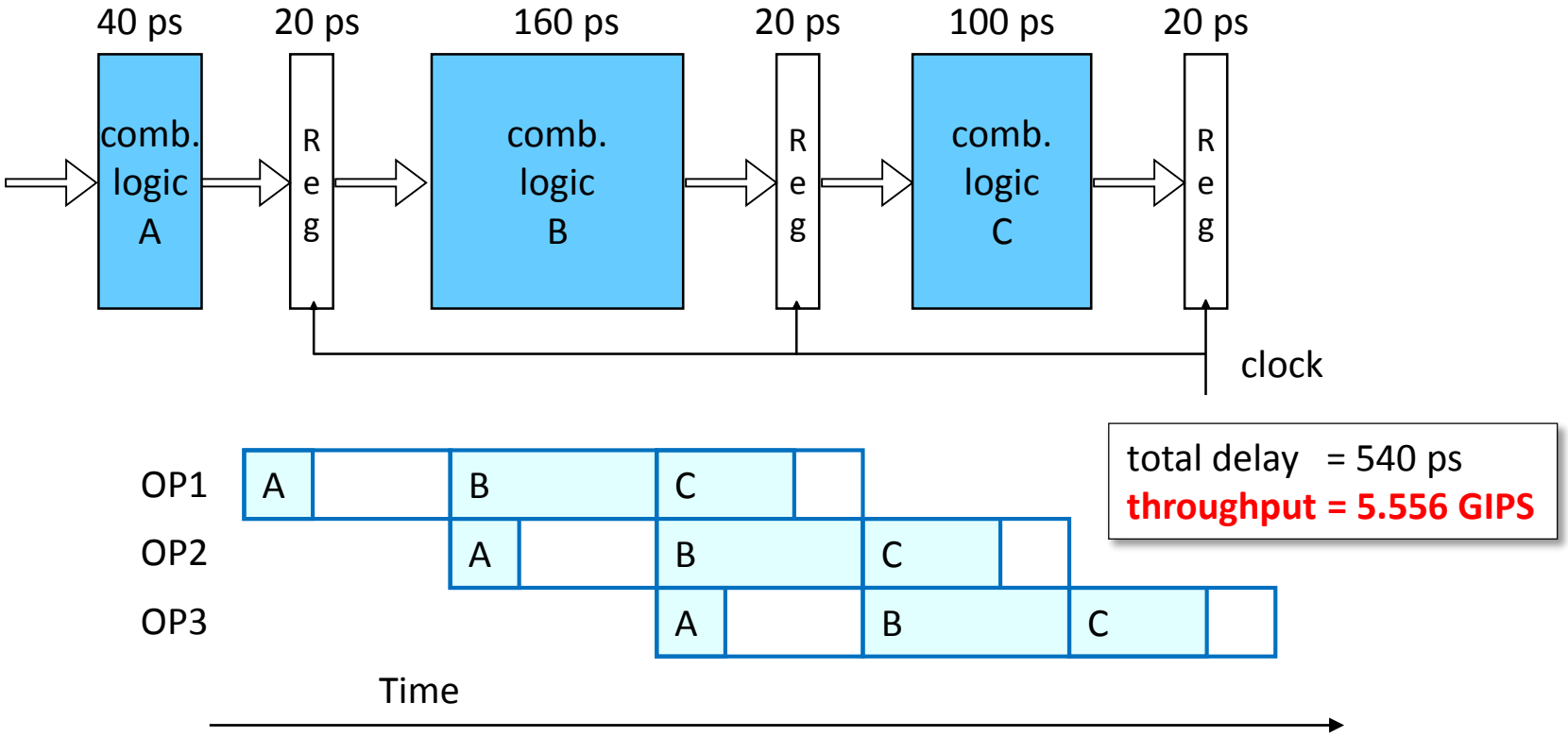
Limitations and Hurdles of Pipelining

Limitations and Major Hurdles of Pipelining

or - **Why we can't just add more pipeline stages to increase performance.**

- Non-uniform delays
- Overhead
- Pipeline hazards

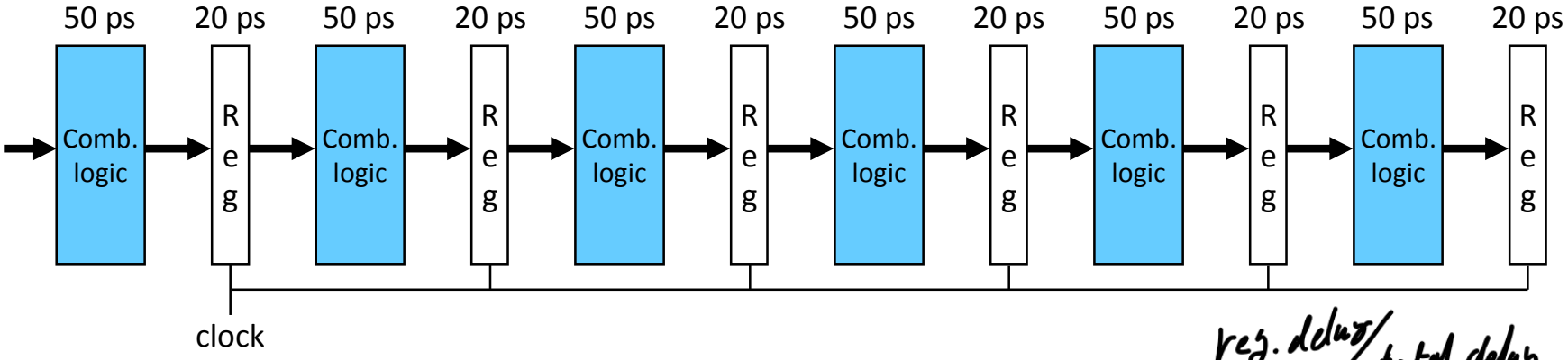
Limitation 1: Non-uniform Delays



- Throughput limited by slowest stage
- Other stages sit idle for much of the time
- **Challenge: partition system into balanced stages**

Limitation 2: Register Overhead

total delay = 420 ps
throughput = 14.286 GIPS



- The deeper the pipeline, the larger the overhead of pipeline registers
- Percentage of clock cycle spent loading pipeline registers:

	stage delay	reg. delay	total delay	overhead
• 1-stage pipeline:	300	20	320	6.25%
• 3-stage pipeline:	100	60	360	16.67%
• 6-stage pipeline:	50	120	420	28.57%

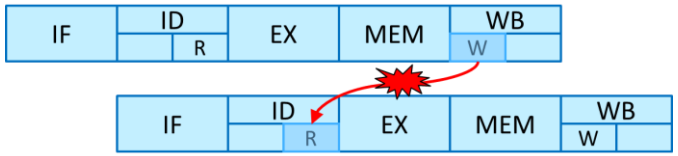
reg. delay increase

- High speeds of modern processor designs obtained through very deep pipelining

Limitation 3: Pipeline Hazards

- **Hazard**: a situation in a pipelined processor where the next instruction cannot execute in the following cycle due to dependencies or contention.

- Three types of hazards
  - *Resource contention* leading to **structural hazards**
  - *Data dependencies* leading to **data hazards**
  - *Control flow* leading to **control hazards**



Limitations and Hurdles of Pipelining

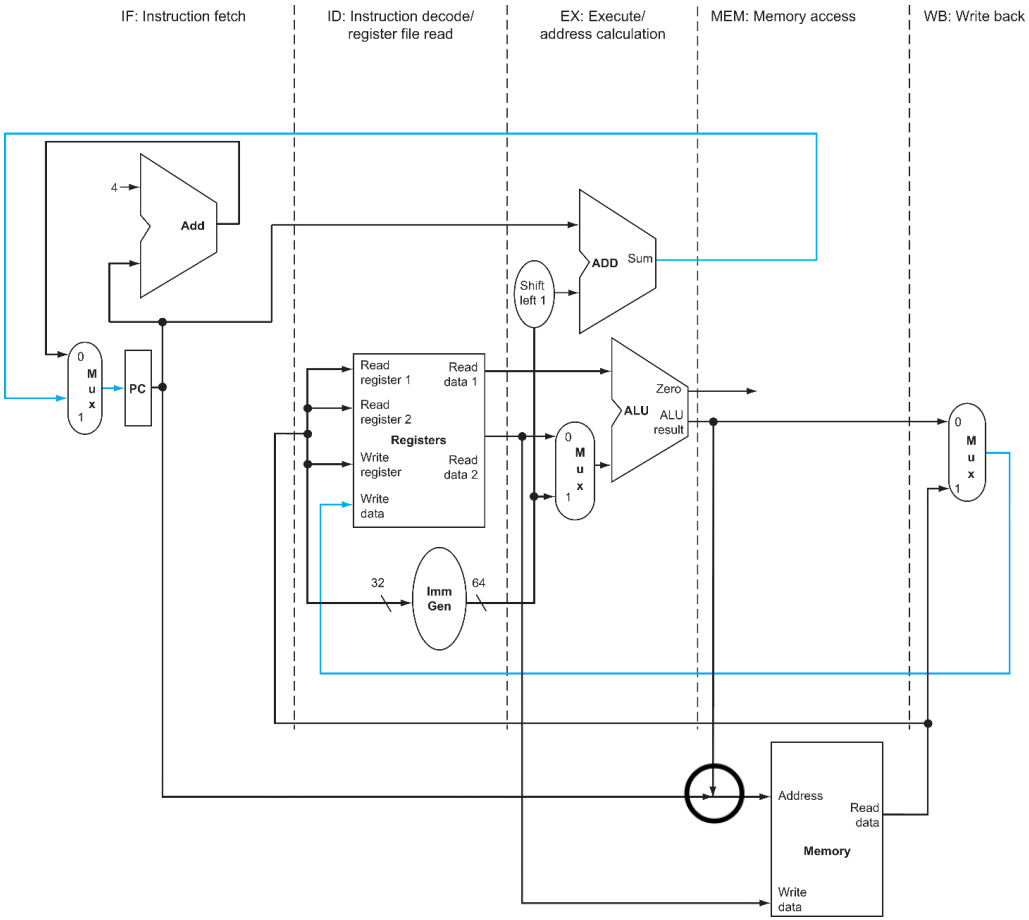
Pipeline Hazards

Structural Hazard

- **Structural hazard: hazard caused by resource contention**

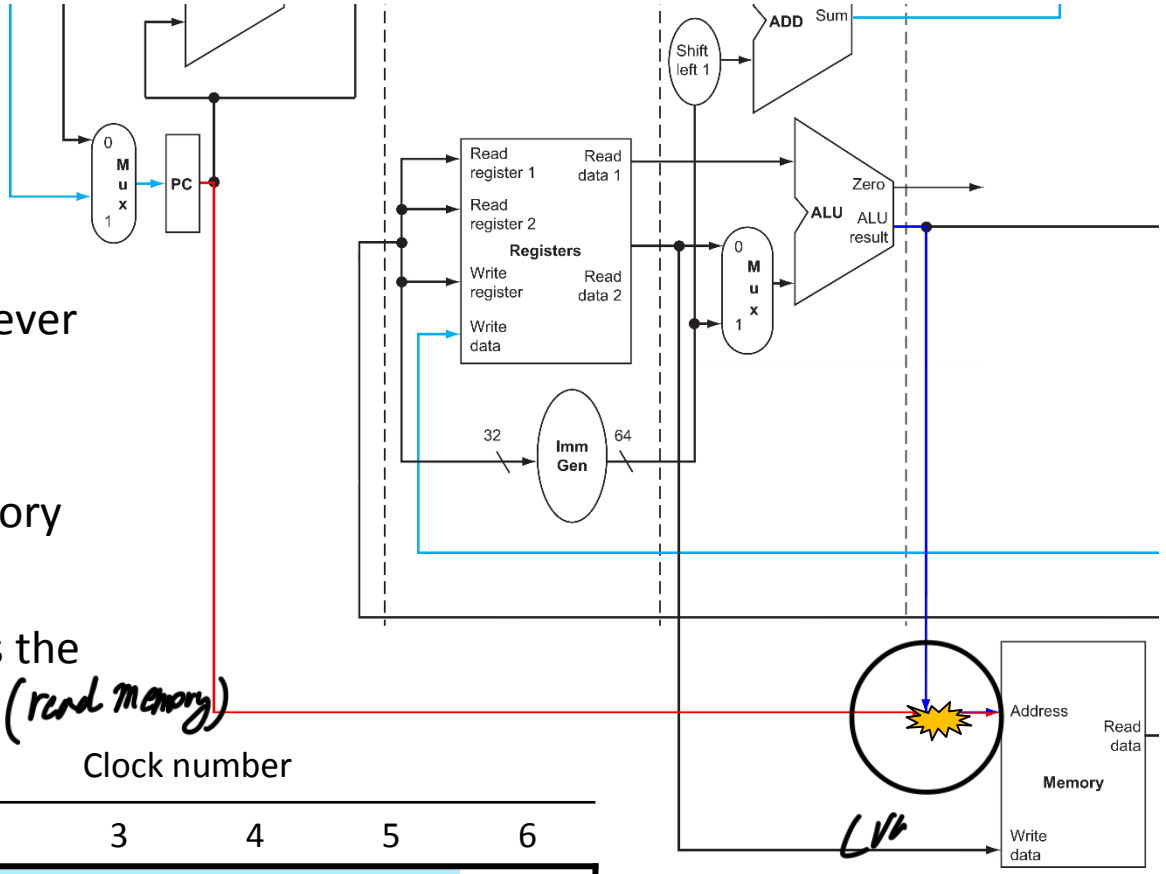
- Assume our processor only had one memory to serve both instruction and data requests

- Simultaneous accesses to the memory by two separate pipeline stages are not possible



Structural Hazard

- Structural hazard occurs whenever a memory instruction is in the MEM stage
  - IF stage accesses the memory in every cycle (read code)
  - memory operations access the memory in the MEM stage (read memory)

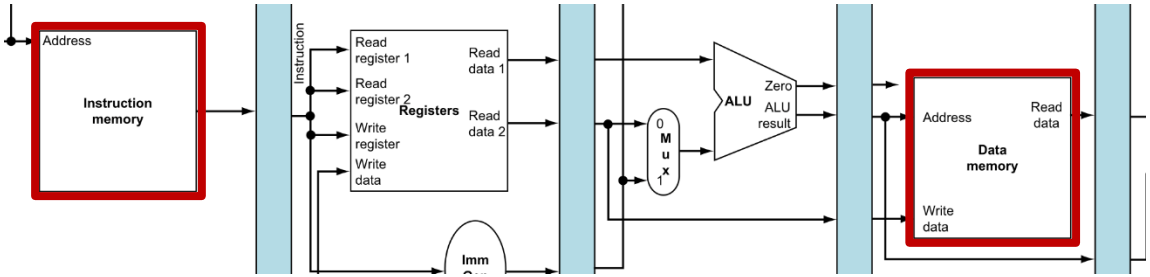


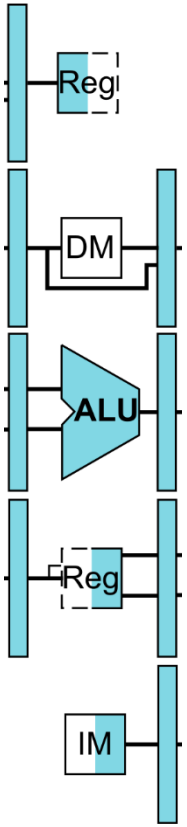
Clock number

Instruction number	1	2	3	4	5	6
Instruction *i*	IF	ID	EX	MEM	WB	
Instruction *i* + 1		IF	ID	EX	MEM	WB
Instruction *i* + 2			IF	ID	EX	MEM
Instruction *i* + 3				IF	ID	EX
Instruction *i* + 4					IF	ID

Resolving Structural Hazards

- **Resource Duplication**
  - Examples
    - Separate I and D memories (caches) to resolve memory access conflicts





    - Time-multiplexed or multi-port register file to resolve register file access conflicts to the same register

Write
1st half of clock

Read
2nd half of clock

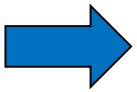
or separate port (Read, Write)

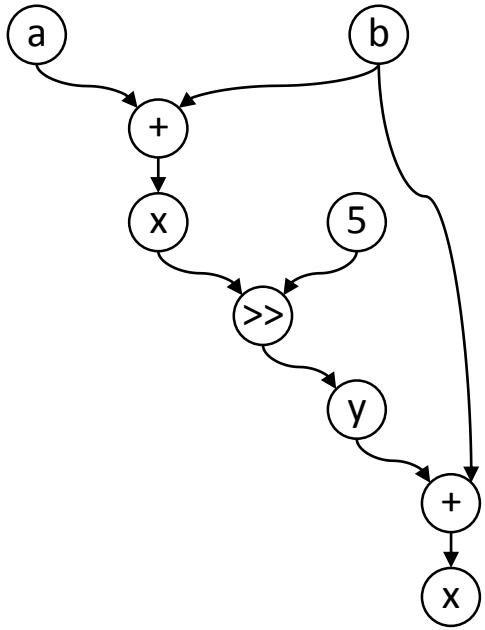
neg / pos

Data Dependencies

- Dependencies caused by data flow
  - Programs can be expressed by a data flow graph

```
x = a + b
y = x >> 5
x = b + y
```

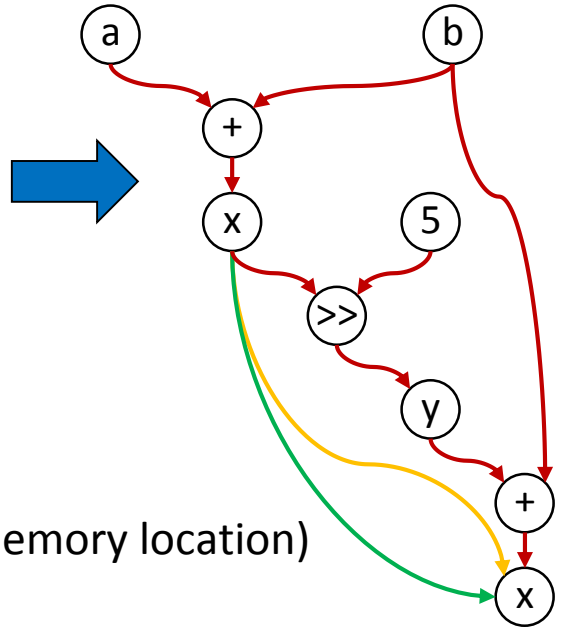




Data Dependencies

- **True dependency**
  - Read After Write (RAW)
  - Represent data flow
  - Cannot be eliminated

```
x = a + b
y = x >> 5
x = b + y
```



- **False dependency**
  - Write After Write (WAW) or output dependency
  - Occur when writing to the same variable (register, memory location)
  - Can be eliminated by renaming

- **Anti-dependency**
  - Write After Read (WAR)
  - Occur when writing to a location that is previously read
  - Can be eliminated by renaming

Data Hazard

- Data hazards are caused by data dependencies
  - Not a problem in single-cycle implementations
  - But may cause data hazards in pipelined implementations
- True dependencies (RAW)
  - Instruction i_t requires result produced by prior instruction i_{t-n} ($n > 0$)
  - Instruction i_t cannot execute before instruction i_{t-n} has produced its result
- False and Anti-dependencies (WAW, WAR)
  - Instruction i_t writes to register/memory x that is read/written by prior instruction i_{t-n}

RAW Data Hazard

OP *i* add x5, x11, x12

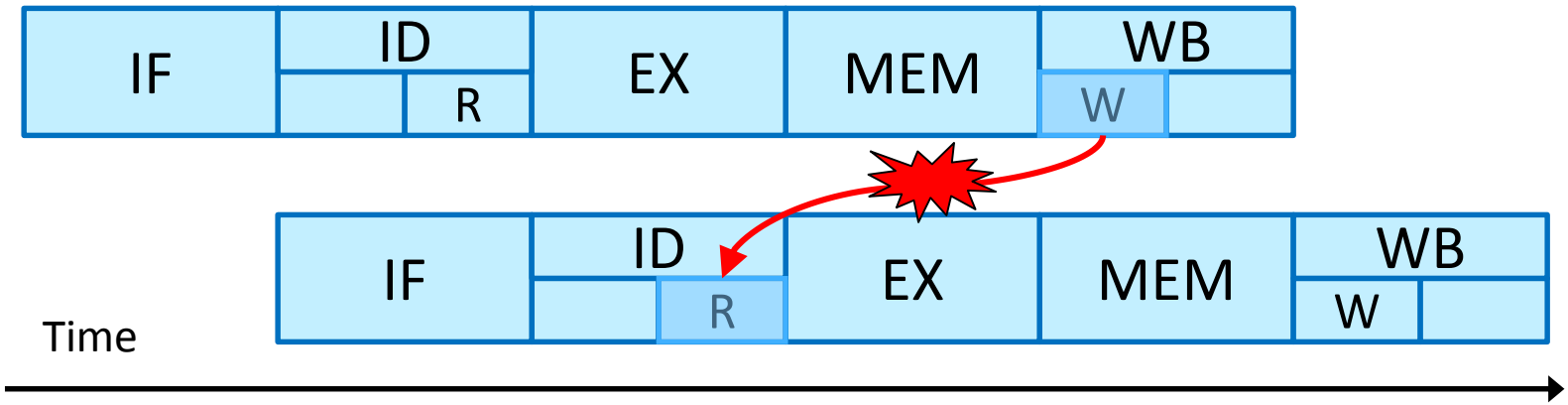
OP *i+1* sub x7, x5, x7

OP i+1 cannot execute before x5 completed (OP i)

RAW Data Hazard

OP *i* add x5, x11, x12

OP *i+1* sub x7, x5, x7



Information cannot flow backwards in time!

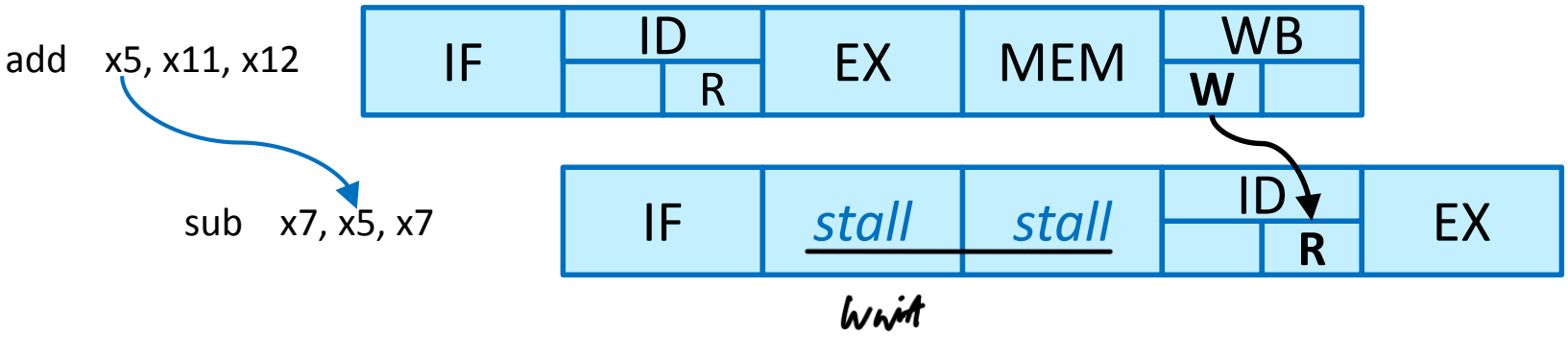
Resolving Data Hazards

- Freezing the pipeline
- (Internal) Forwarding
- Compiler scheduling

Data Hazard – Freezing The Pipeline

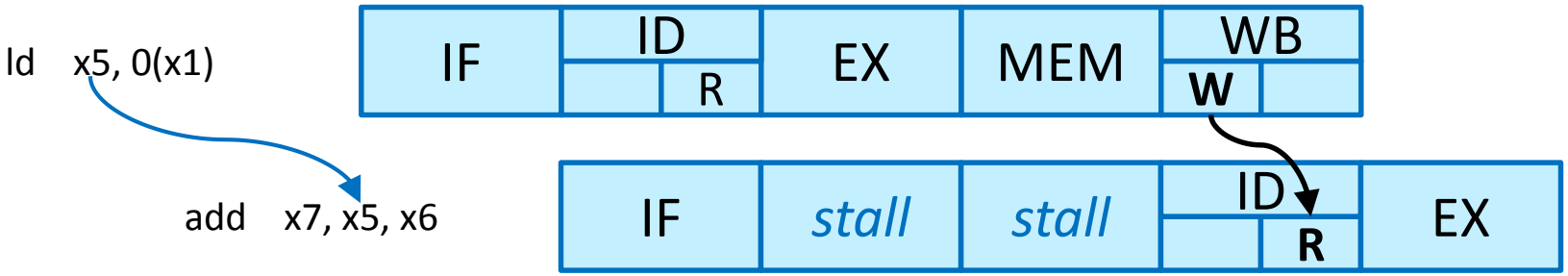
Stall

- ALU result to next instruction



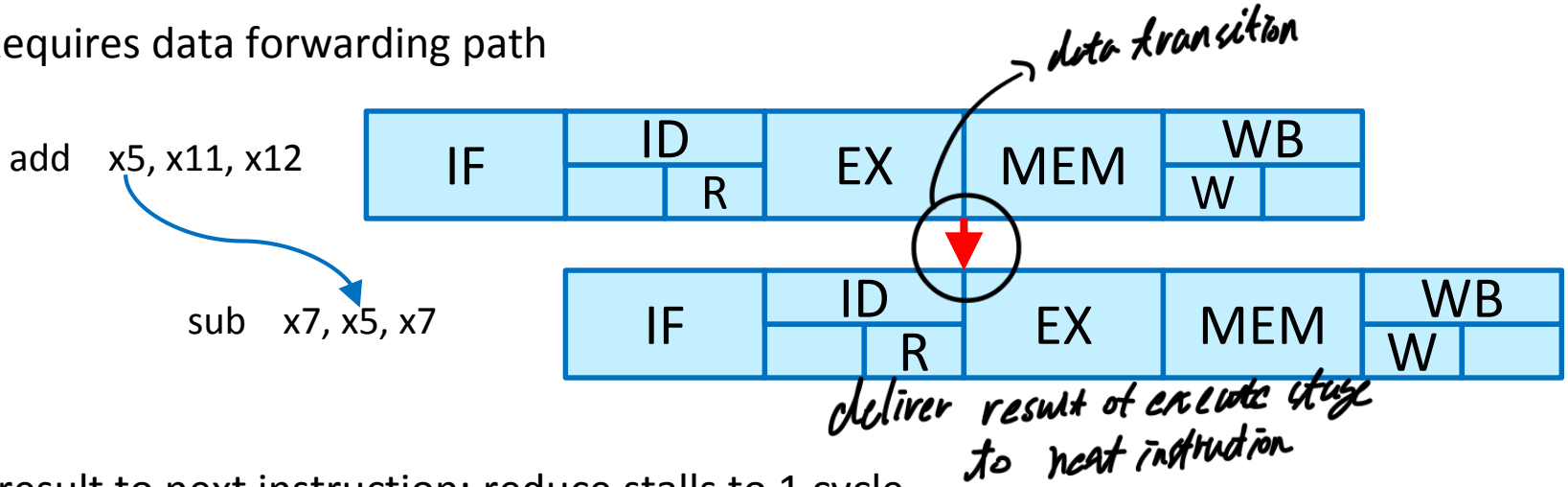
wait

- Load result to next instruction

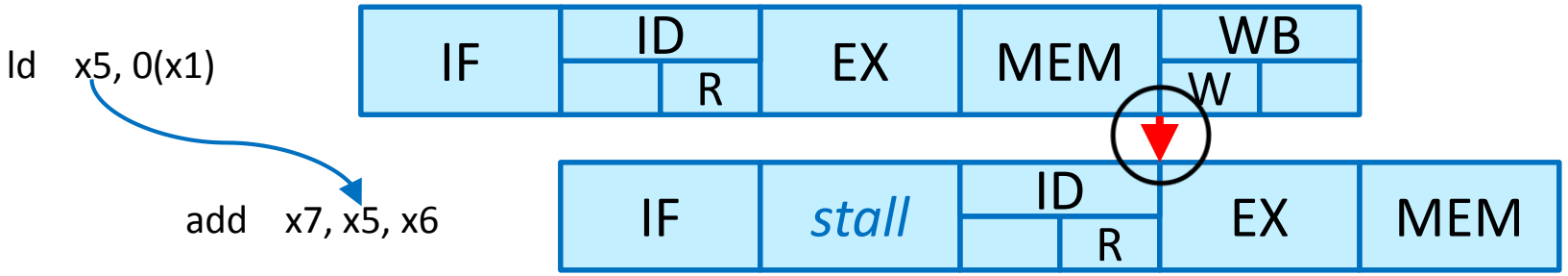


Data Hazard – (Internal) Forwarding

- ALU result to next instruction: stall can be avoided
  - Requires data forwarding path



- Load result to next instruction: reduce stalls to 1 cycle
  - Requires forwarding path

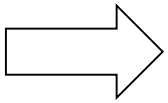


load interlock / load-use data hazard

Data Hazard – Compiler Scheduling

- Rearranging instructions to avoid data hazards

```
...
add    x5, x6, x7
sd     x5, 0(x1)
ld     x6, 0(x5)
addi   x8, x6, 3
addi   x6, x0, 1
...
```



```
...
add    x5, x6, x7
ld     x6, 0(x5)
sd     x5, 0(x1)
addi   x8, x6, 3
addi   x6, x0, 1
...
```

make term between dependencies

- Requires a careful data flow analysis and knowledge about the inner workings of the pipeline

Control Hazard

- Caused by PC-changing instructions
  - Conditional branches, function call/return

Branch Instruction	IF	ID	EX	MEM	WB				
Branch successor		IF	*stall*	*stall*	IF	ID	EX	MEM	WB
Branch successor + 1						IF	ID	EX	MEM	WB
Branch successor + 2							IF	ID	EX	MEM
Branch successor + 3								IF	ID	EX
Branch successor + 4									IF	ID
Branch successor + 5										IF

For 5-stage pipeline, 3 cycle penalty
15% branch frequency. CPI = 1.45

Resolving Control Hazards

- Stall – too slow
- Branch prediction
- Branch delay slots (delayed decision)

Control Hazard – Branch Prediction

- Predict-taken

Taken branch instruction	IF	ID	EX	MEM	WB				
Branch target		IF	ID	EX	MEM	WB			
Branch target + 1			IF	ID	EX	MEM	WB		
Branch target + 2				IF	ID	EX	MEM	WB	
Branch target + 3					IF	ID	EX	MEM	WB

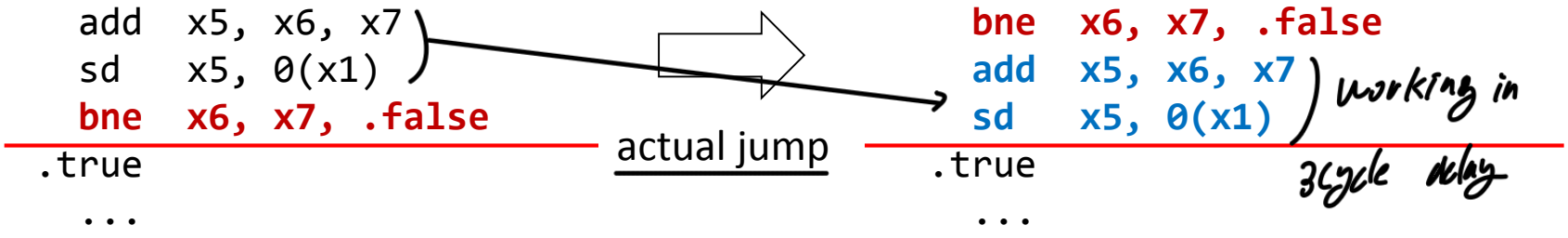
Untaken branch instruction	IF	ID	EX	MEM	WB				
Branch target : ignoring		IF	*idle*	*idle*	*idle*	*idle*			
Untaken branch instruction + 1			IF	ID	EX	MEM	WB		
Untaken branch instruction + 2				IF	ID	EX	MEM	WB	
Untaken branch instruction + 3					IF	ID	EX	MEM	WB

~60% success rate

- other techniques:
  - predict backwards-taken, forward-untaken
  - hardware branch prediction

Control Hazard – Branch Delay Slots

- Execute the next *n* instructions following a branch no matter what the result of the branch
  - Up to the compiler to make good use of the branch delay slots
  - Exposes the pipeline structure to the user



branch taken: stall 2 cycles
not taken: no stall

add, sd executed no matter whether the jump is taker or not (no stall)

IF	ID	EX	MEM	WB				
	IF	*idle*	*idle*	*idle*	*idle*			
		IF	ID	EX	MEM	WB		
			IF	ID	EX	MEM	WB	
				IF	ID	EX	MEM	WB

Module Summary

Module Summary

- **Pipeline parallelism**
  - break up (long) sequential circuit into smaller pieces
  - separate pieces by registers
  - called *stages*
  - used by (almost) all processors these days

- **Difficulties** (there is no free lunch)
  - increased complexity, latency
  - even stages
  - hazards
    - structural hazards: resource contention
    - data hazards: data dependencies
    - control hazards: changes in control flow