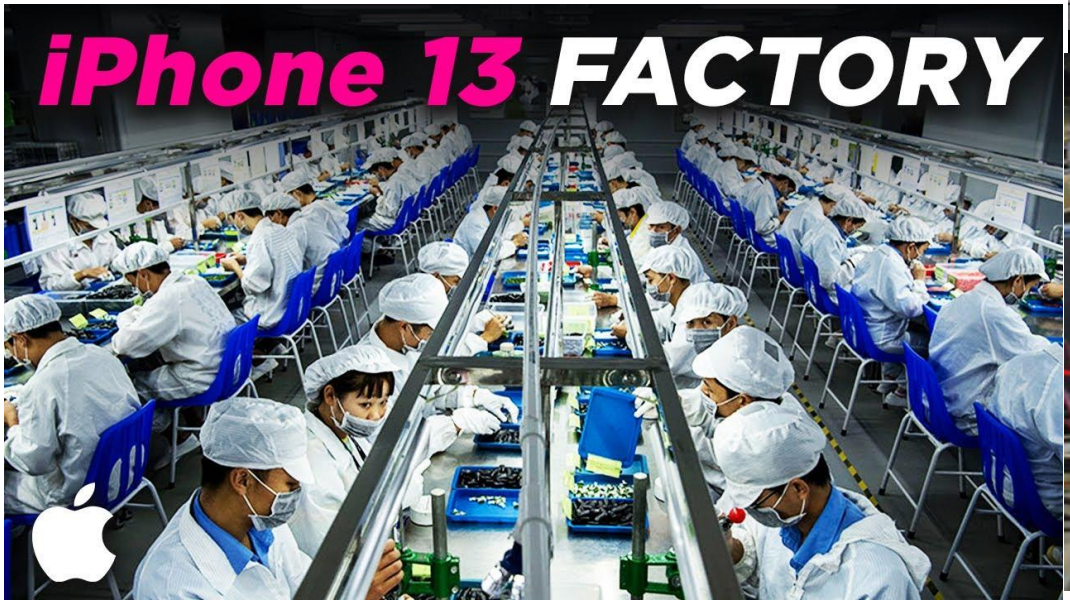
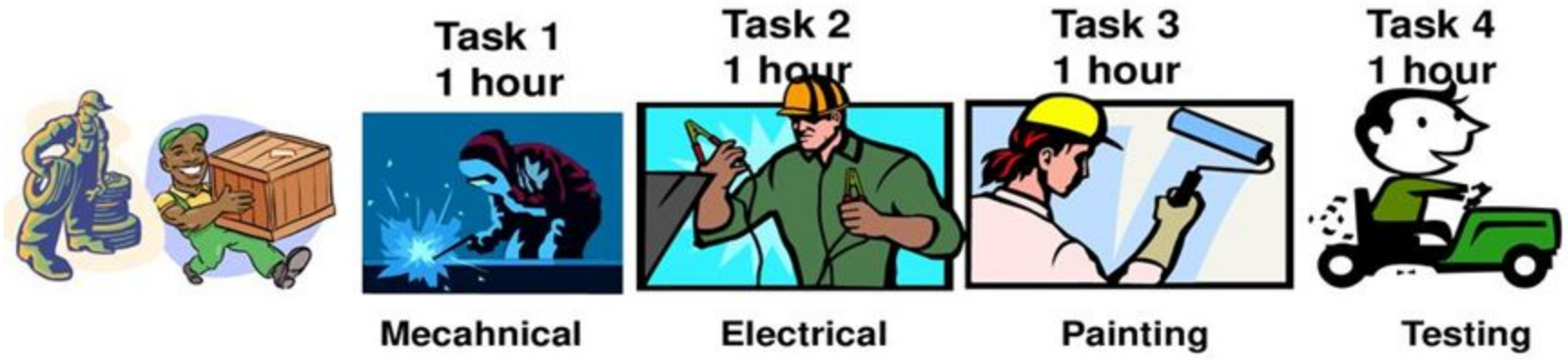
Pipelining/Assembly line: to increase production



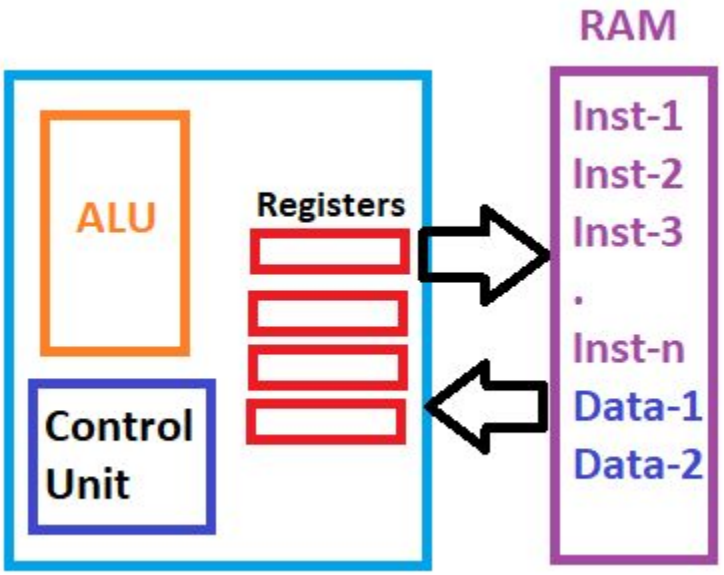
Pipelining example

Automobile Assembly Line



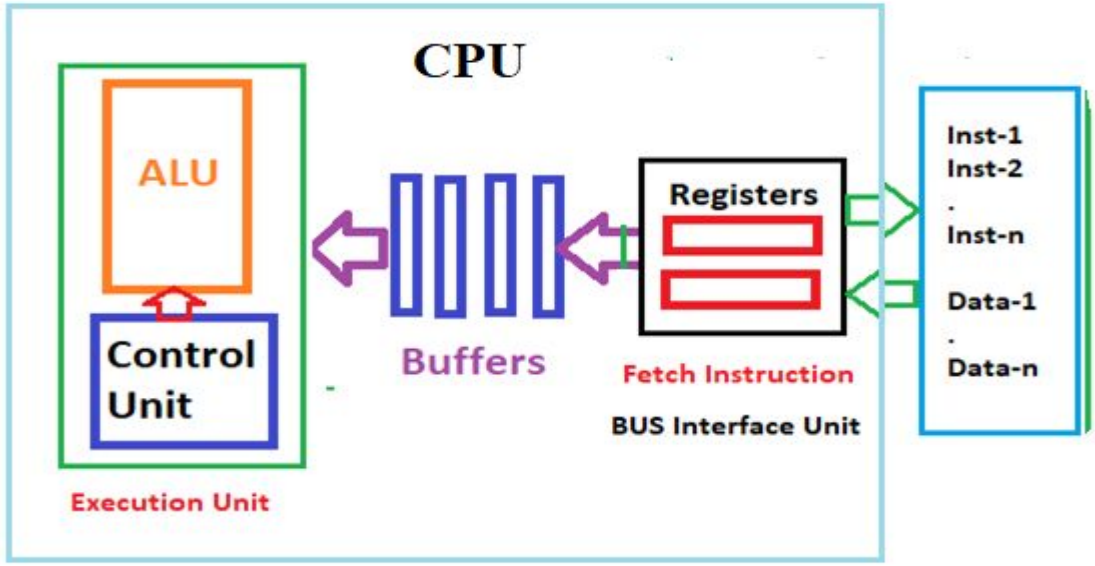
First car assembled in 4 hours (pipeline latency)
thereafter 1 car per hour
21 cars on first day, thereafter 24 cars per day
717 cars per month
8,637 cars per year

Non-pipeline vs Pipeline Architecture

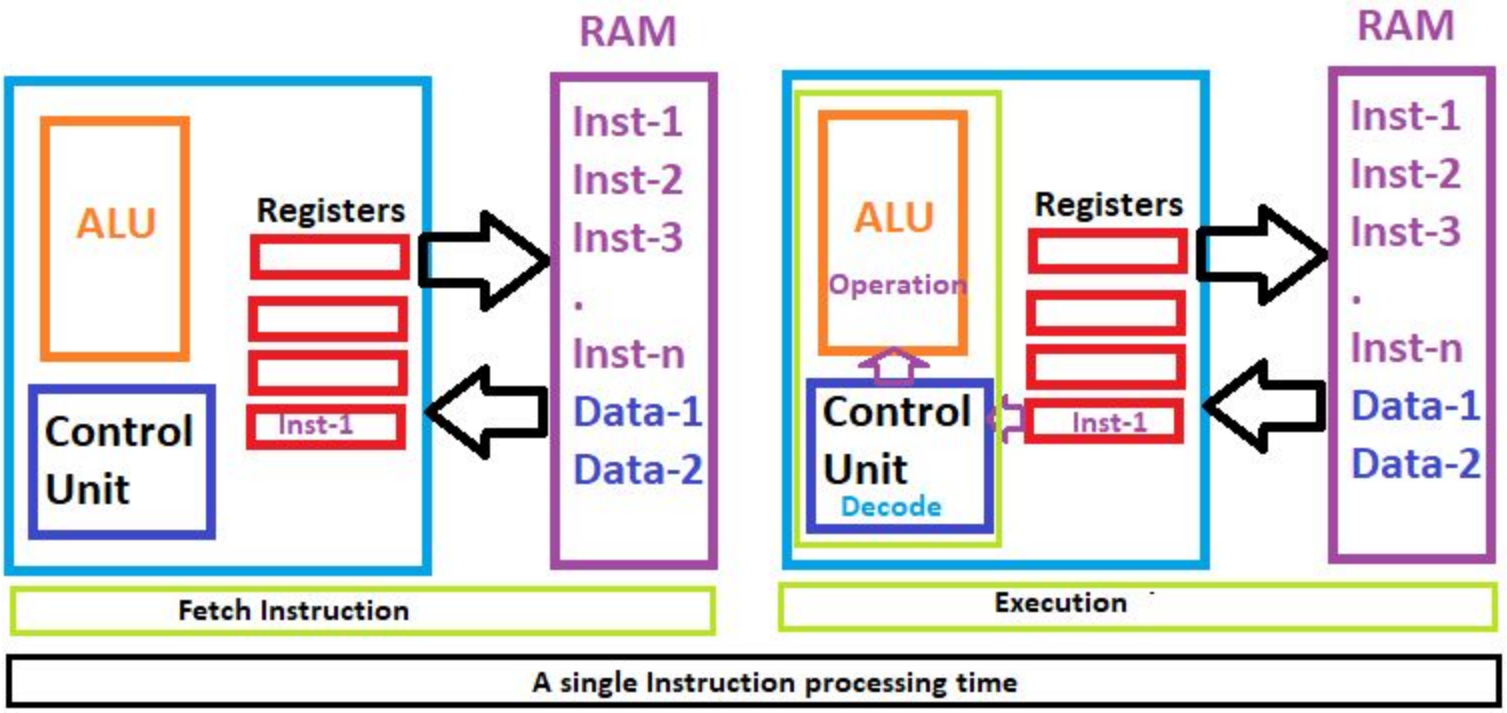


Conventional/non-pipeline architecture is designed to process just one instruction at a time. Once an instruction is completely processed (result stored), the CPU will begin the processing of next instruction.

The Pipelining architecture is designed like an assembly line for instruction processing.

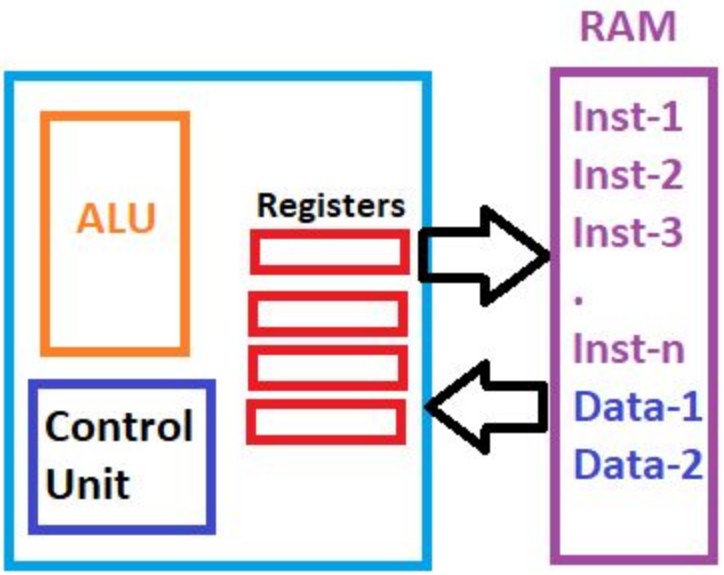


Non-pipeline Architecture (Intel-8085)



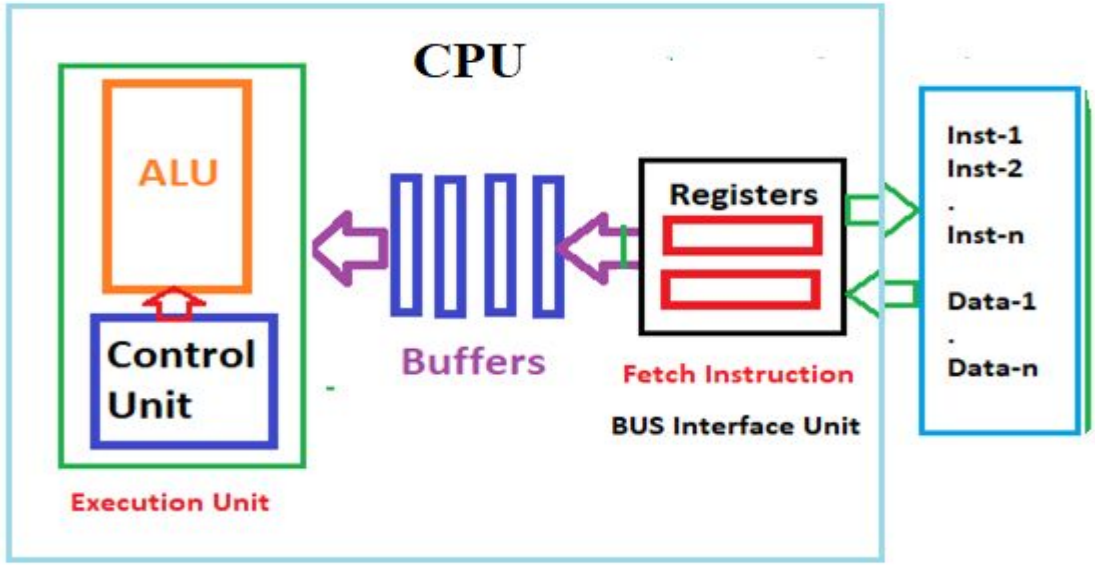
Non-pipeline processor can process one instruction at a time. A number of sub-tasks may require to complete the processing of each instruction and CPU will perform only one task related to the processing of an instruction at a time.

Pipeline vs non-pipeline Architecture



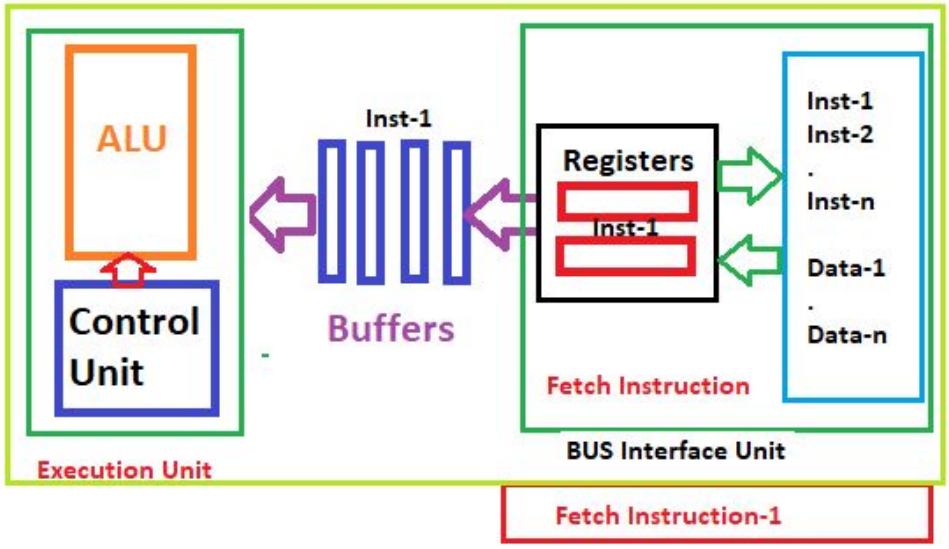
The Pipelining architecture is designed like an assembly line for instruction processing.

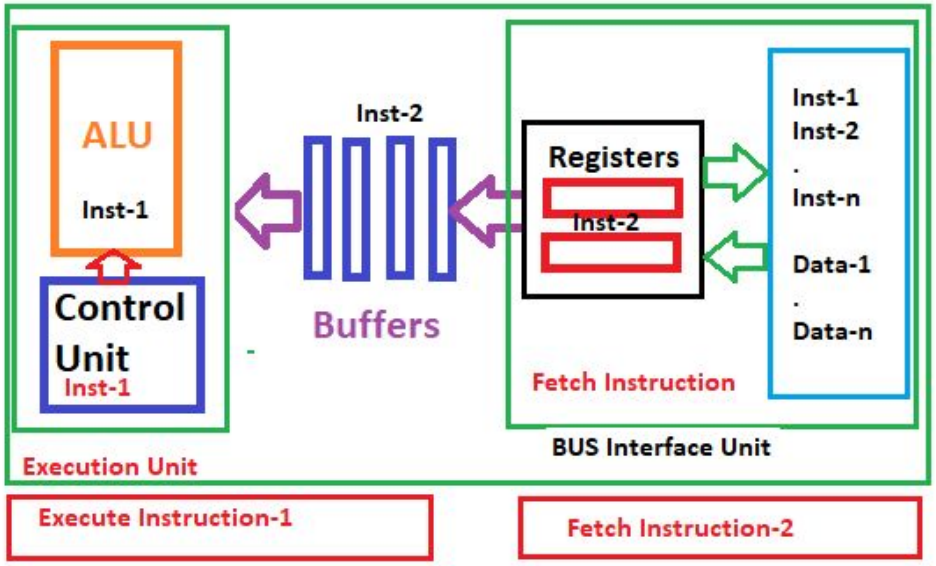
Processing task of an instruction is split into a number of sub-tasks and the datapath is designed like an assembly line having dedicated processing circuits for each sub-task.



Pipeline Architecture (Intel-8086)

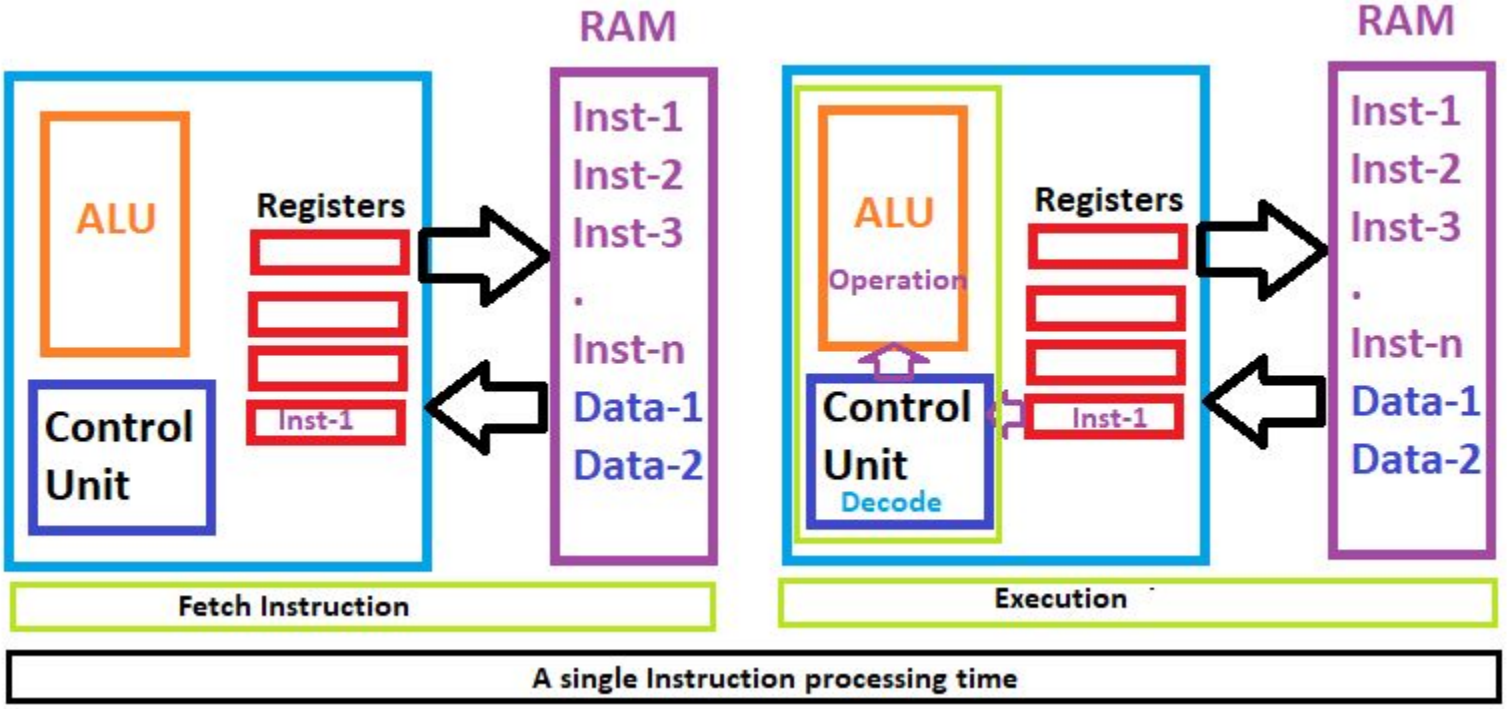
Processing of instruction is split into sub-tasks and Pipelining architecture is designed to process multiple sub-tasks of different instructions at the same time.

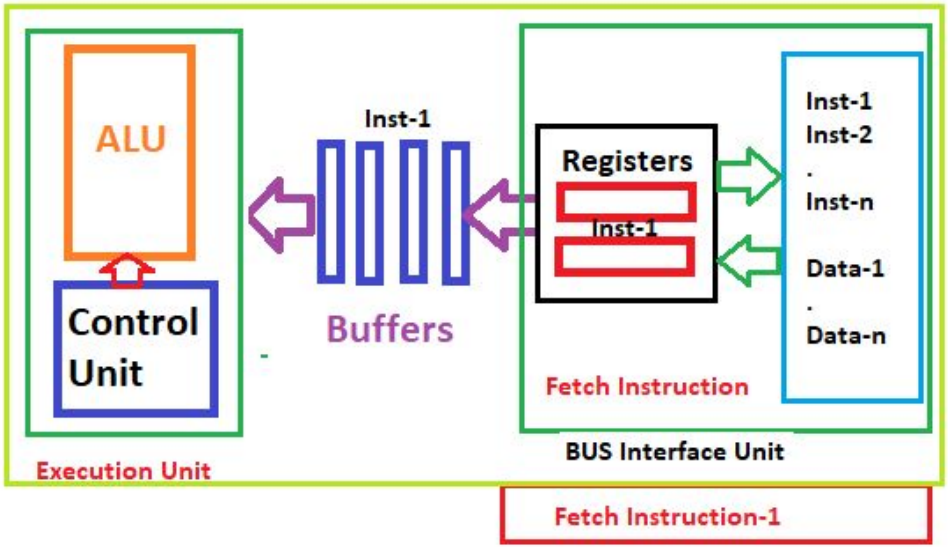


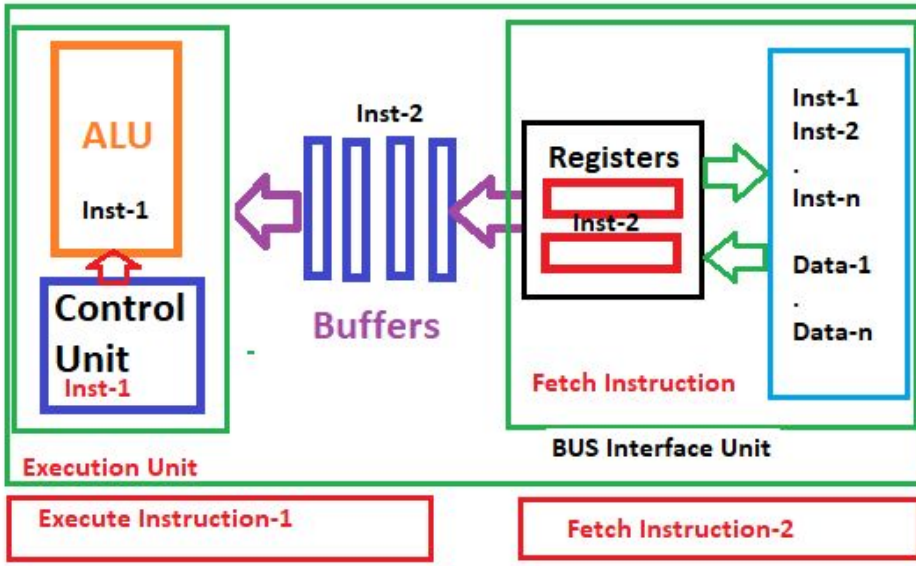


Two instructions are being processed at the same time.

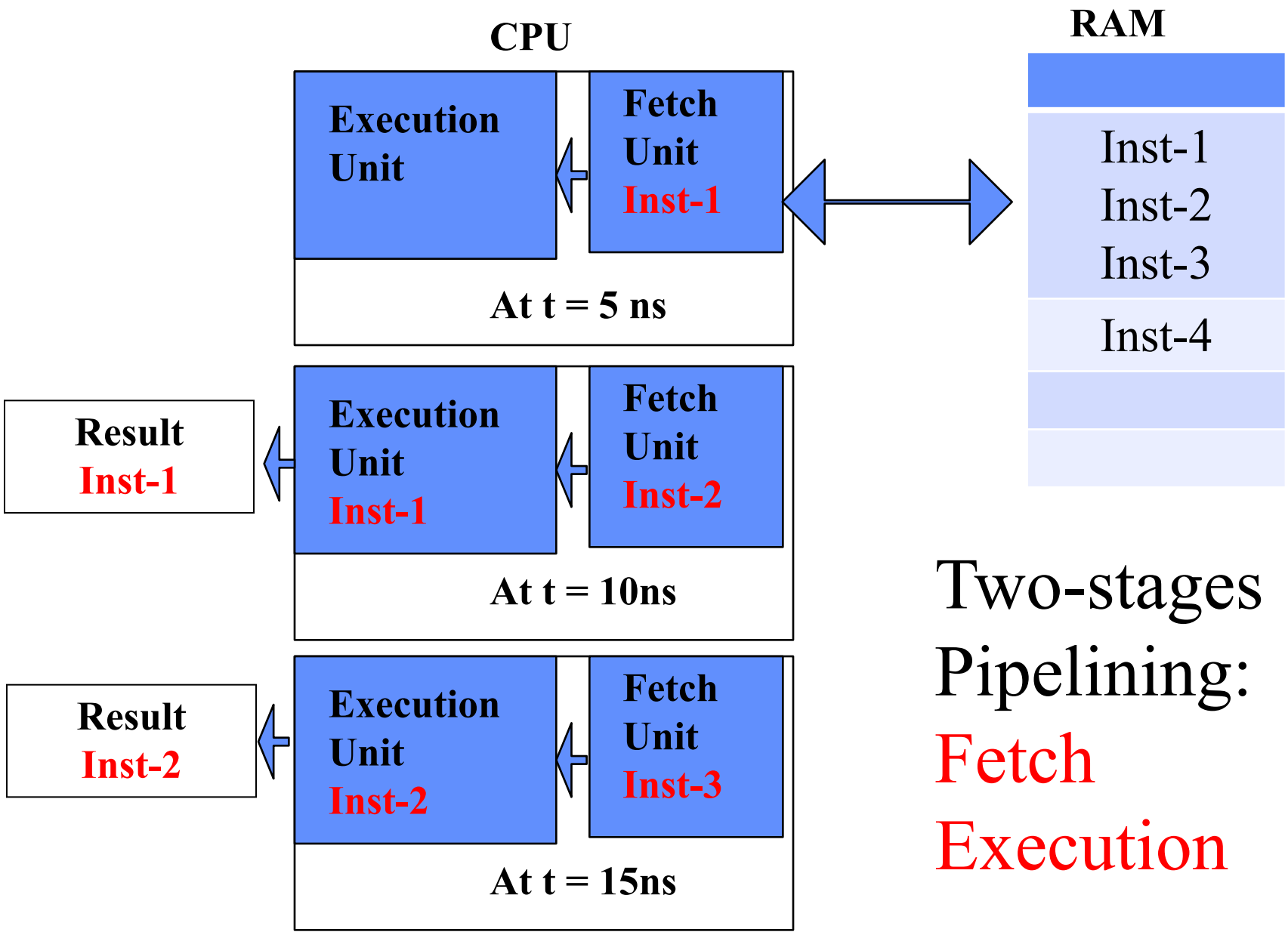
Pipeline vs non-pipeline







Two-stages Pipelining



Two-stages Pipelining:

Fetch

Execution

Time diagram

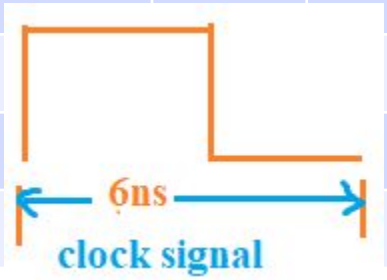
Non-pipelined CPU(8085)

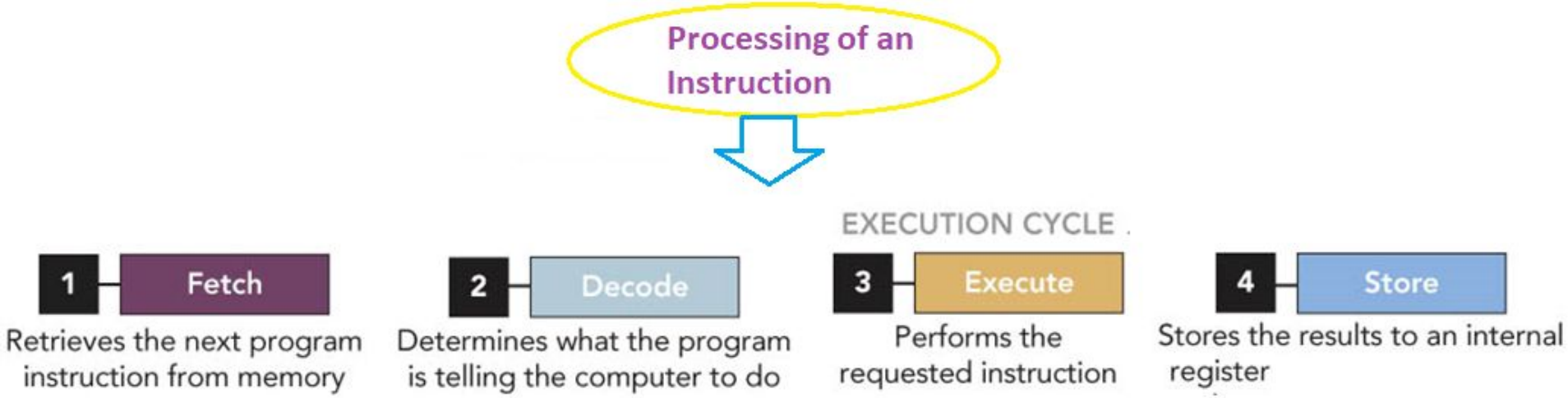
Clock cycles

Instruction	1	2	3	4	5	6	7	8	9	10
Instruction-1	IF	EU								
Instruction-2			IF	EU						
Instruction-3					IF	EU				
Instruction-4							IF	EU		
Instruction-5									IF	EU
Instruction-6										

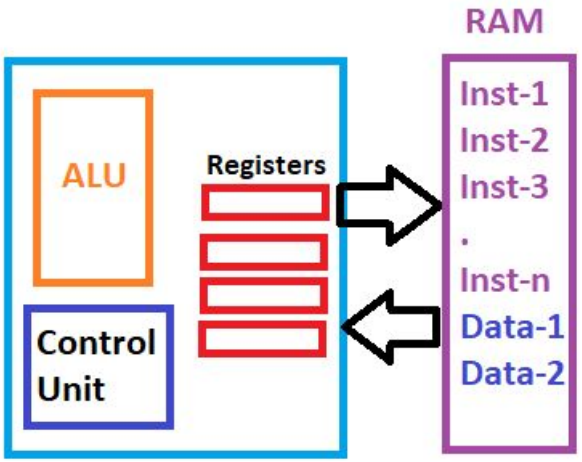
Pipelined CPU(8086)

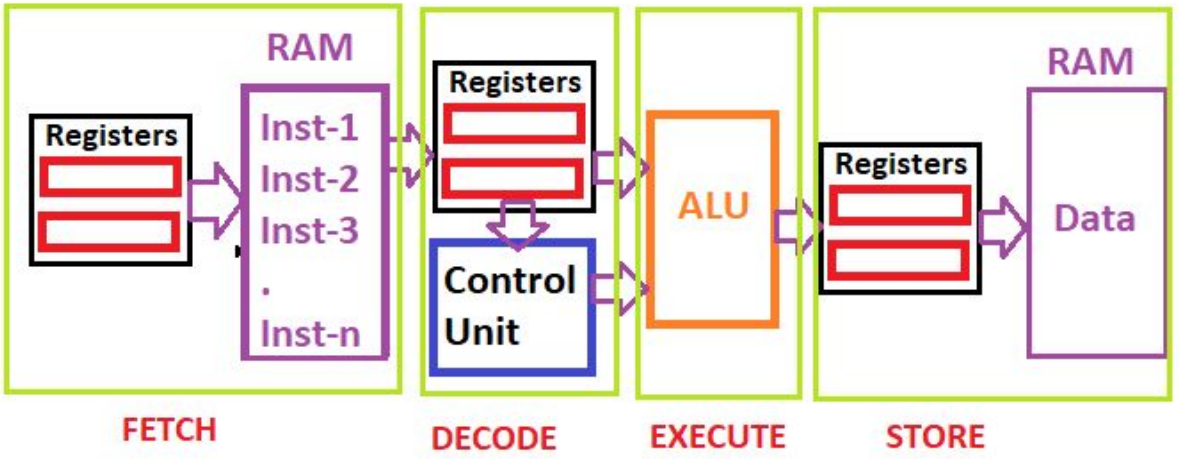
Instruction	1	2	3	4	5	6	7	8	9	10
Instruction-1	IF	EU								
Instruction-2		IF	EU							
Instruction-3			IF	EU						
Instruction-4				IF	EU					
Instruction-5					IF	EU				
Instruction-6										

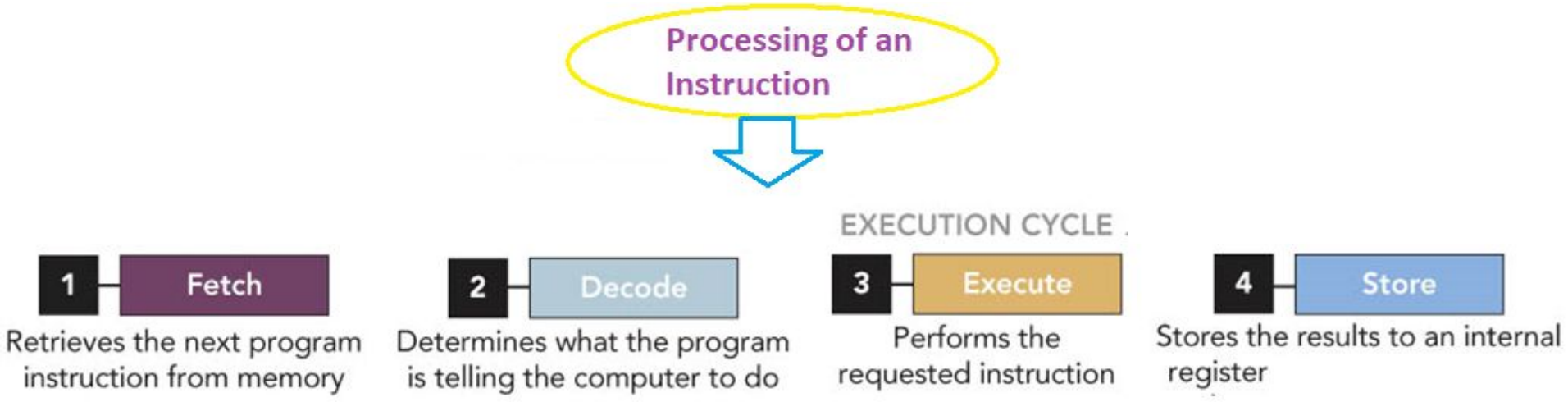




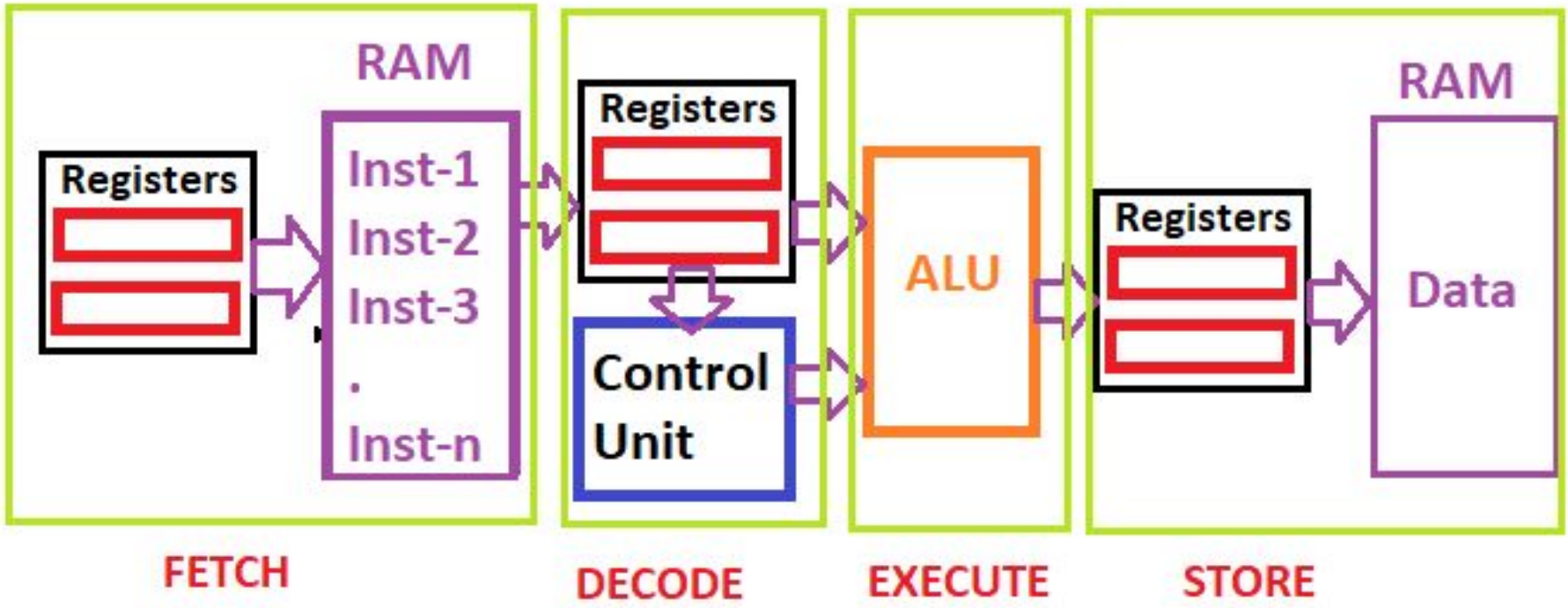
Four-stage pipelining

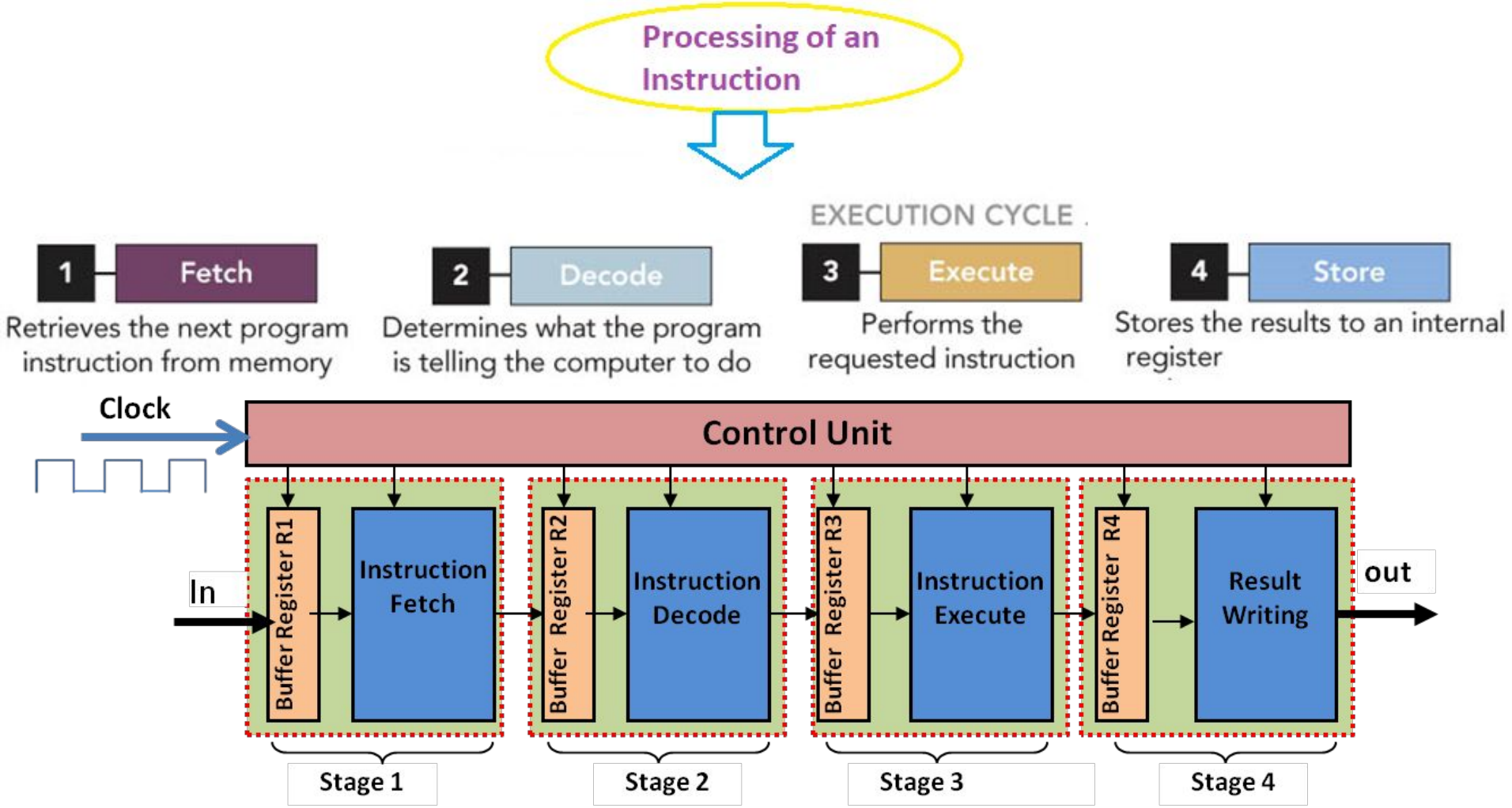






Four-stage pipelining





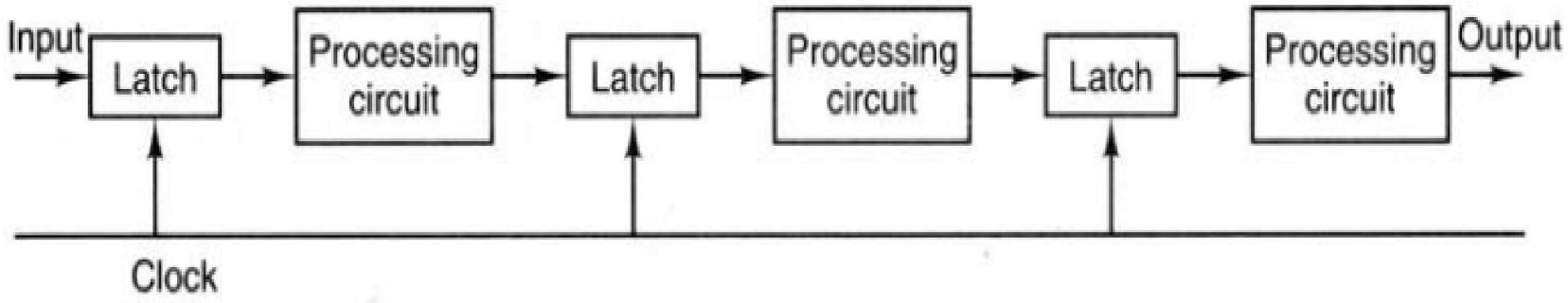
Inst	Clock cycle-1	Clock cycle-2	Clock cycle-3	Clock cycle-4	Clock cycle-5
Inst-1	Fetch	Decode	ALU	WR	
Inst-2		Fetch	Decode	ALU	
Inst-3			Fetch	Decode	
Inst-4				Fetch	

Pipelining

- Pipelining is an implementation technique that allows multiple instructions overlapped in execution.
- The Pipelining architecture is designed like an assembly line for instruction processing.
- Pipeline architecture involves not only executing an instruction over multiple cycles, but also executing multiple instructions per cycle.

Pipelining

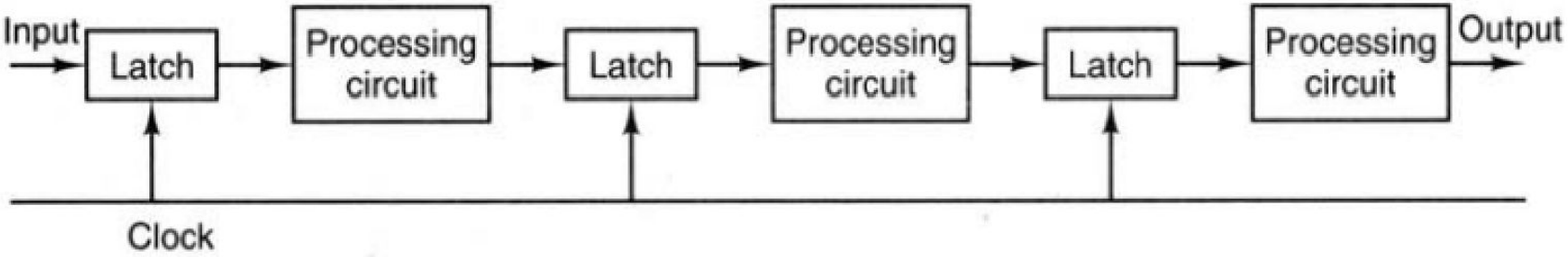
The pipeline design technique decomposes a sequential process into several subprocesses, called stages or segments. A *stage* performs a particular function and produces an intermediate result. It consists of an input latch, also called a register or buffer, followed by a processing circuit. (A processing circuit can be a combinational or sequential circuit.) The processing circuit of a given stage is connected to the input latch of the next stage. A clock signal is connected to each input latch. At each clock pulse, every stage transfers its intermediate result to the input latch of the next stage. In this way, the final result is produced after the input data have passed through the entire pipeline, completing one stage per clock pulse.

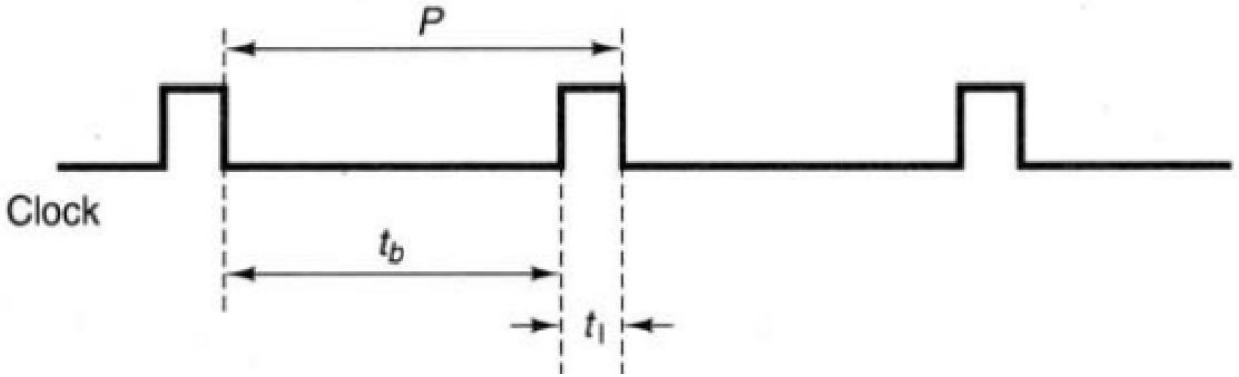


Pipelining

The period of the clock pulse should be large enough to provide sufficient time for a signal to traverse through the slowest stage, which is called the *bottleneck* (i.e., the stage needing the longest amount of time to complete). In addition, there should be enough time for a latch to store its input signals.

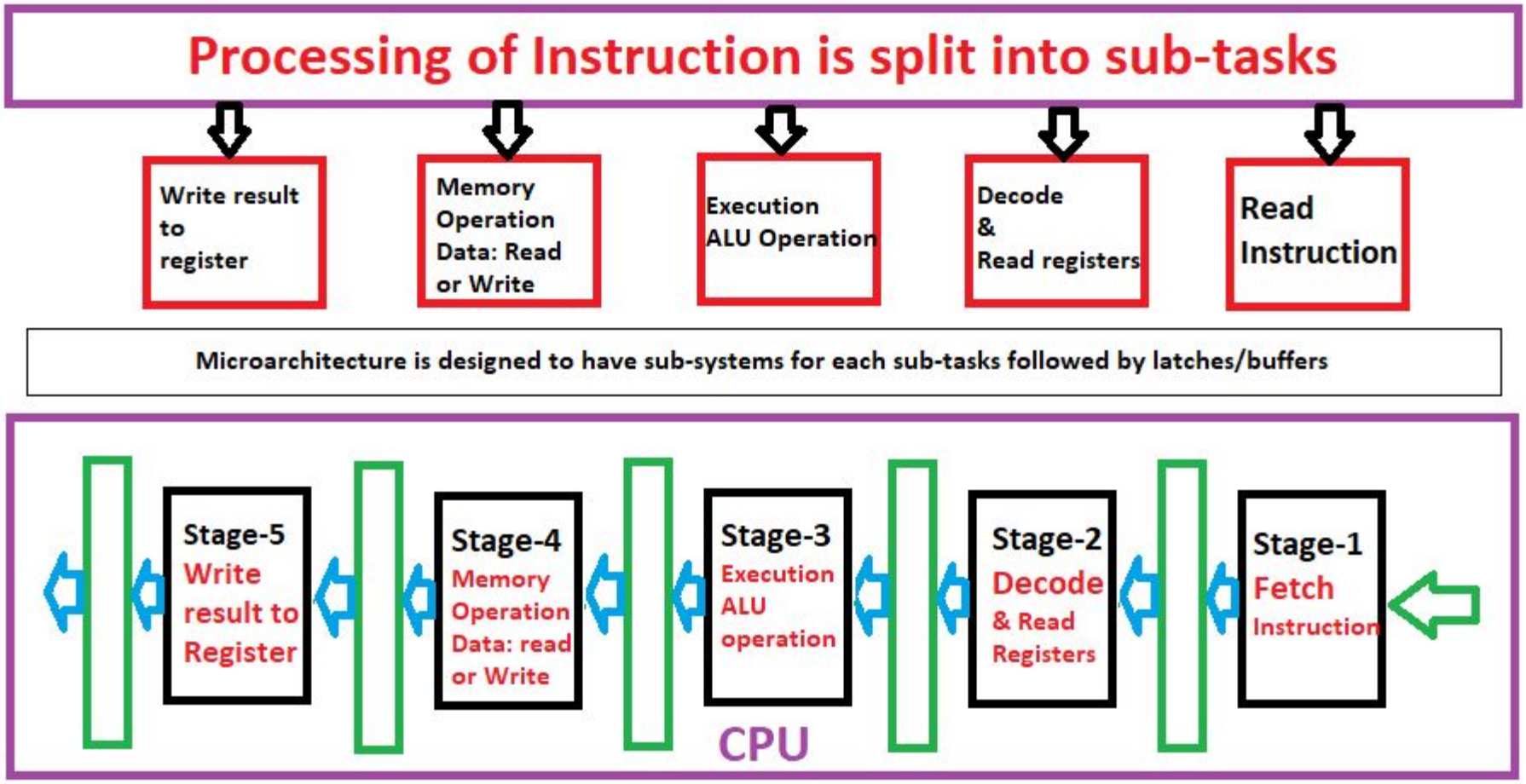
If the clock's period, P, is expressed as $P = tb + tl$, then tb should be greater than the maximum delay of the bottleneck stage, and tl should be sufficient for storing data into a latch.





- A pipeline works much as an assembly line in a manufacturing plant enabling different stages of execution of different instructions to occur at the same time along the pipeline.
- Pipelining is one way of improving the overall processing performance of a processor. This architectural approach allows the simultaneous execution of several instructions.
- Pipelining is transparent to the programmer; it exploits parallelism at the instruction level by overlapping the execution process of instructions. It is analogous to an assembly line where workers perform a specific task and pass the partially completed product to the next worker.

5-stage Pipelining: Concept & Design

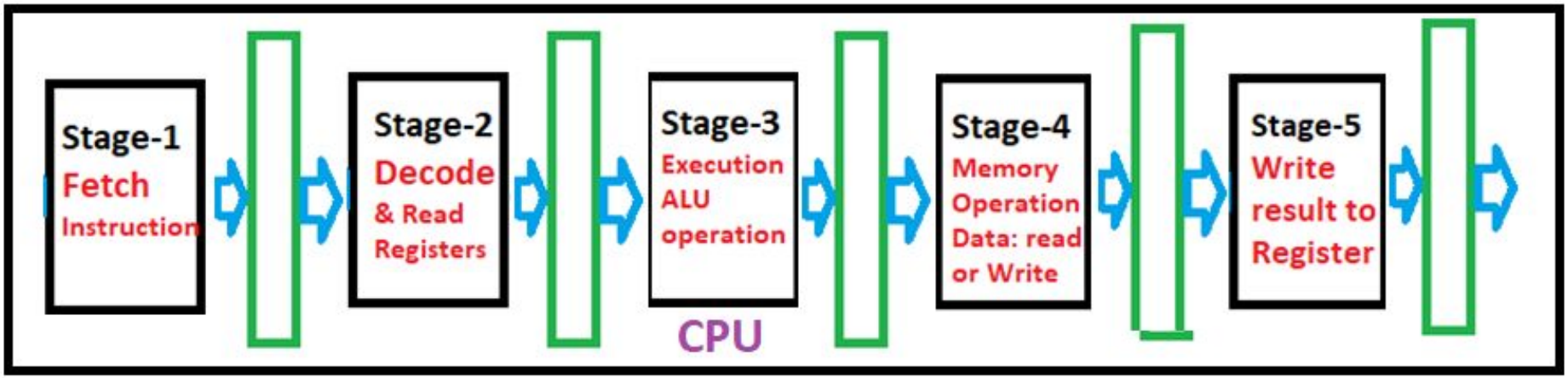


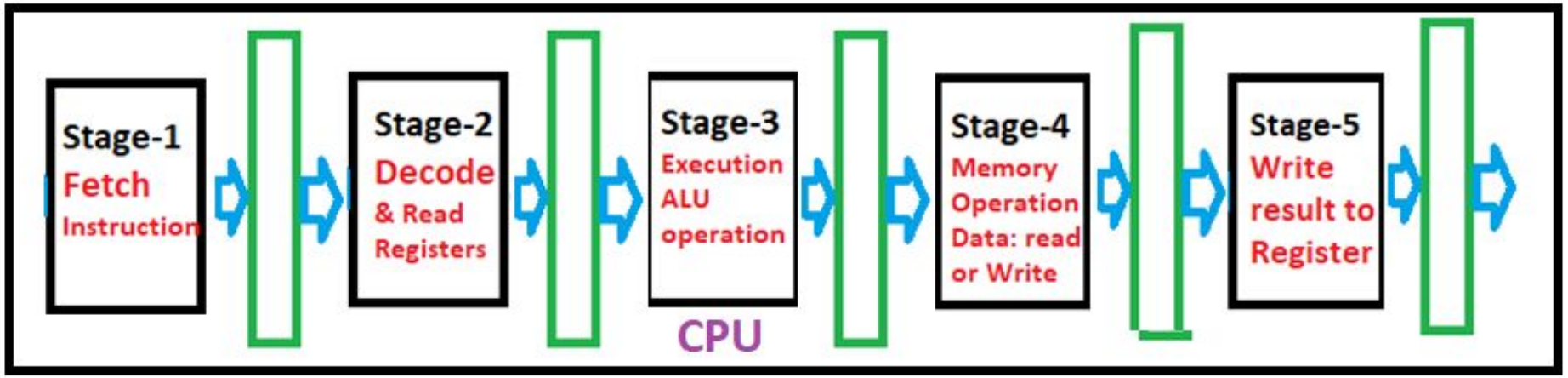
5-stage pipelining (RISC)

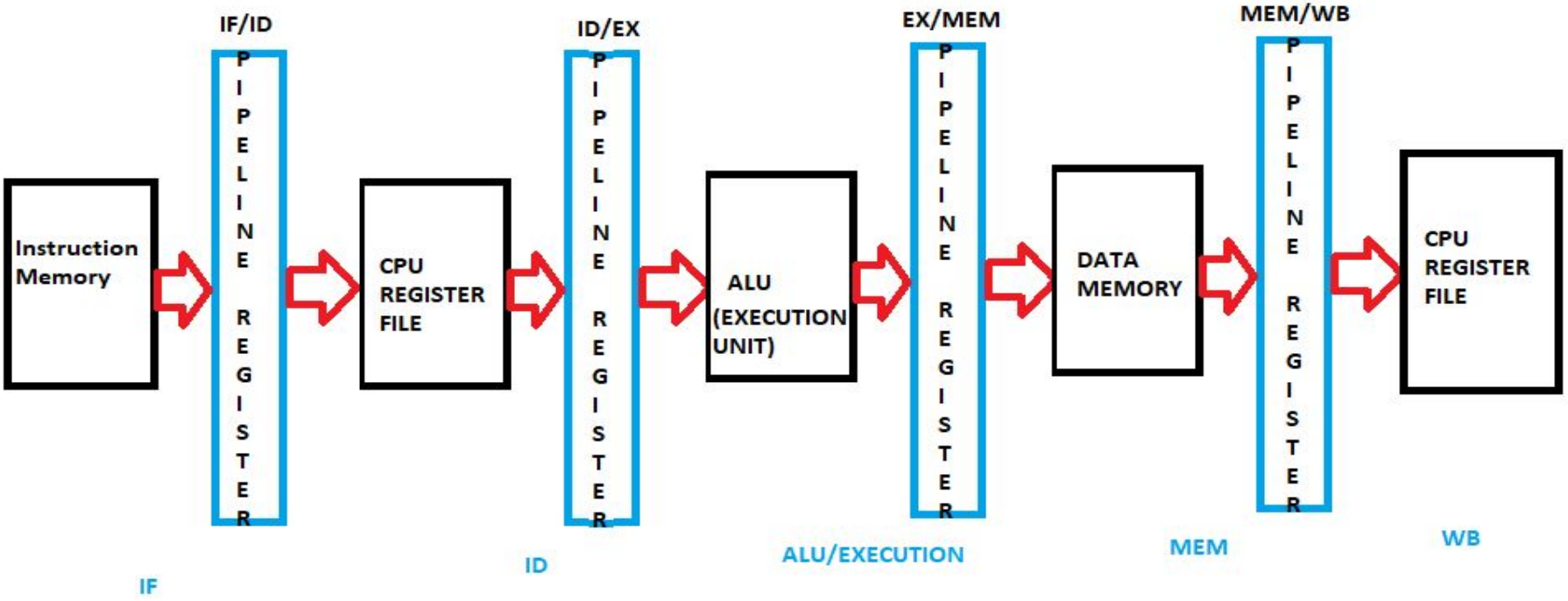
5-stage processing of an instruction	5-stages Hardware Within processor
Fetch Instruction from Memory(IF)	**Fetch Unit (FU)**
Decode Instruction and register fetch(ID)	Decode Unit (D&R)
ALU Operation(EX)	Execution (EU)
Memory operation, if required (MA)	Memory Unit (DM)
Write the result in register(WB)	Write Unit (WB)

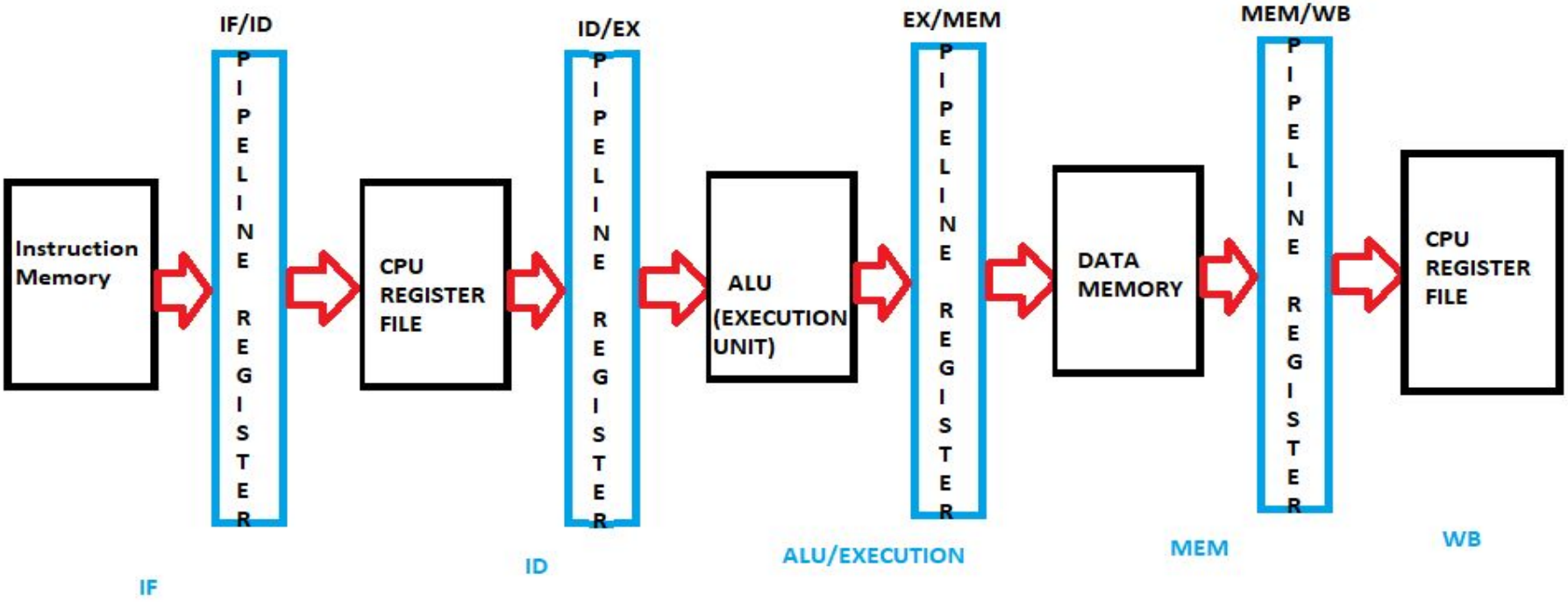
5-stage pipelining (RISC-processors)

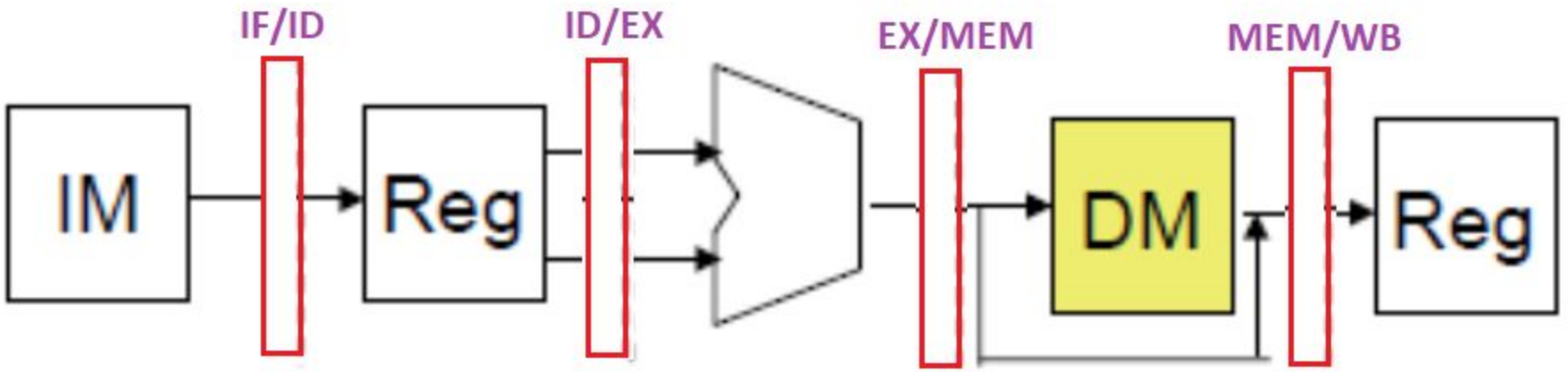
- 5-stage processing of each Instruction, in general, although some instructions may NOT require all stages
- **ALU** instructions are register based
- **LOAD** and **STORE** Instructions are used to read data from RAM and store to RAM
- **Harvard Architecture**: Separate memories for Instructions and Data
- Latches/buffers between stages to allow sections work simultaneously on different Instructions

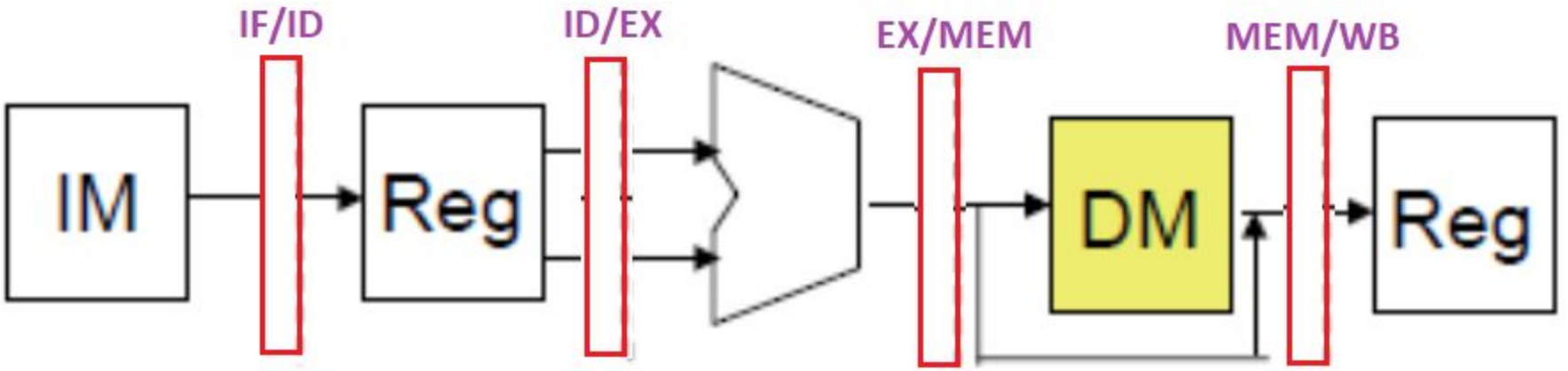


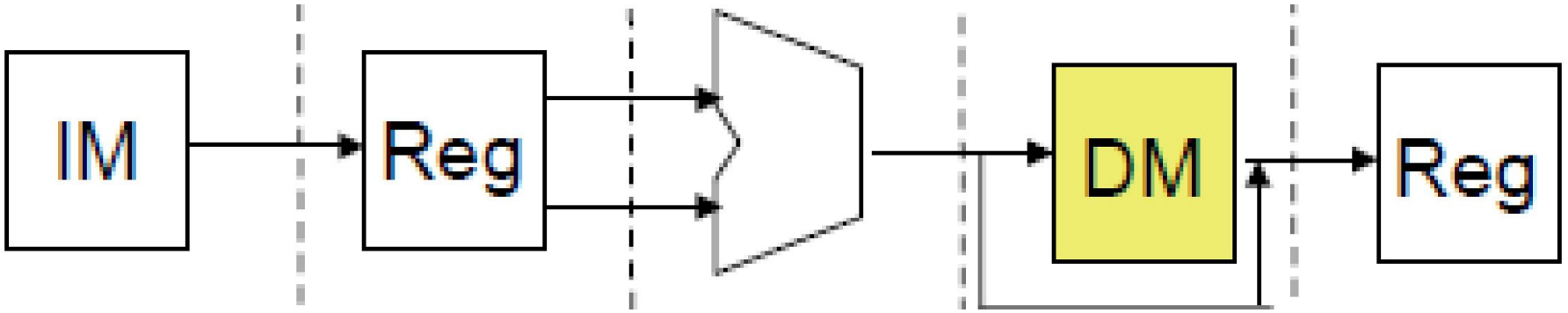

Stage-1
Fetch
Instruction
Stage-2
Decode
& Read
Registers
Stage-3
Execution
ALU
operation
Stage-4
Memory
Operation
Data: read
or Write
Stage-5
Write
result to
Register
CPU

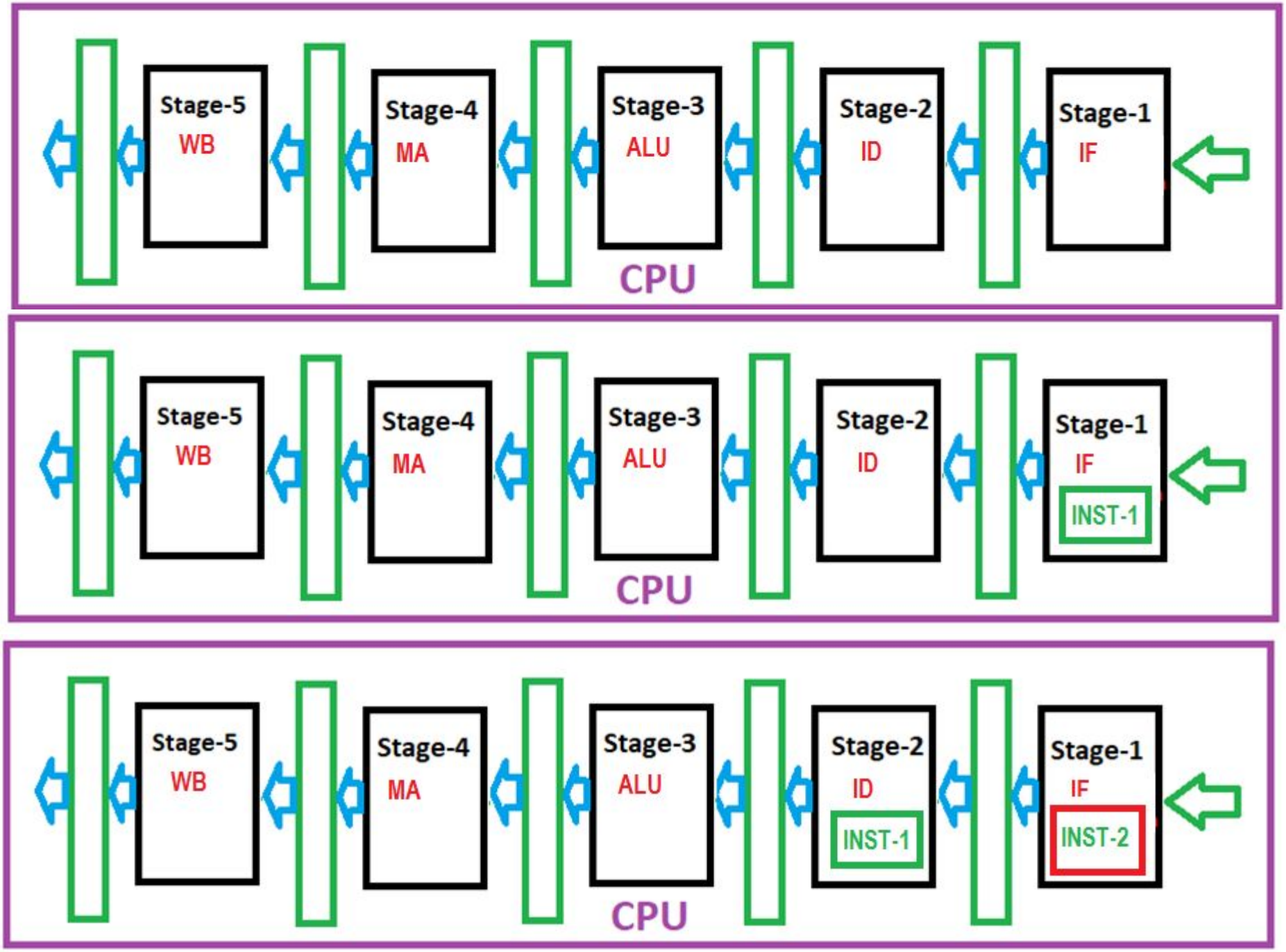

IF/ID
ID/EX
EX/MEM
MEM/WB
Instruction Memory
PIPELINE REGISTER
CPU REGISTER FILE
PIPELINE REGISTER
ALU (EXECUTION UNIT)
PIPELINE REGISTER
DATA MEMORY
PIPELINE REGISTER
CPU REGISTER FILE
IF
ID
ALU/EXECUTION
MEM
WB


IF/ID
ID/EX
EX/MEM
MEM/WB
Instruction Memory
PIPELINE REGISTER
CPU REGISTER FILE
PIPELINE REGISTER
ALU (EXECUTION UNIT)
PIPELINE REGISTER
DATA MEMORY
PIPELINE REGISTER
CPU REGISTER FILE
IF
ID
ALU/EXECUTION
MEM
WB


IF/ID
ID/EX
EX/MEM
MEM/WB
IM
Reg
DM
Reg


IF/ID
ID/EX
EX/MEM
MEM/WB
IM
Reg
DM
Reg


IM
Reg
DM
Reg


Stage-5
WB
Stage-4
MA
Stage-3
ALU
Stage-2
ID
Stage-1
IF
CPU
Stage-5
WB
Stage-4
MA
Stage-3
ALU
Stage-2
ID
Stage-1
IF
INST-1
CPU
Stage-5
WB
Stage-4
MA
Stage-3
ALU
Stage-2
ID
INST-1
Stage-1
IF
INST-2
CPU

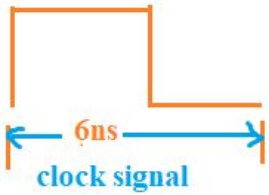
Pipelining (RISC): Time steps
Clock cycles progress left to right
Instructions progress top to bottom
Time at which each instruction is present in each pipeline stage is shown by labelling appropriate cell with pipeline name
Information from one instruction to any successor must always move from left to right

Instruction\ cycle	1	2	3	4	5	6	7	8
Instruction-1	IF	ID	EX	MA	WB			
Instruction-2		IF	ID	EX	MA	WB		
Instruction-3			IF	ID	EX	MA		
Instruction-4				IF	ID	EX		
Instruction-5					IF	ID		
Instruction-6						IF		
Instruction-7								
Instruction-8								

CPI and Average CPI

Instruction\ cycle	1	2	3	4	5	6	7	8
Instruction-1	IF	ID	EX	MA	WB			
Instruction-2		IF	ID	EX	MA	WB		
Instruction-3			IF	ID	EX	MA	WR	
Instruction-4				IF	ID	EX	MA	WR
Instruction-5					IF	ID	EX	MA
Instruction-6						IF	ID	EX
Instruction-7							IF	ID
Instruction-8								IF

6ns 12ns 18ns 24ns 30ns 36ns 42ns



- **CPI of each Instruction = 5**
- **Average CPI of a program having 10 Instructions**
  - **Total clock cycles for 10 Inst. = 5 + (10-1) x 1 = 14 clock cycles**
  - **Average CPI = 14/10 = 1.4**
- **Average CPI of a program having 100 Instructions**
  - **Total clock cycles for 100 Inst. = 5 + (100-1) x 1 = 104 clock cycles**
  - **Average CPI = 104/100 = 1.04**

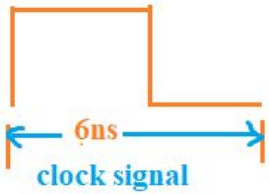
CPI and Average CPI

Instruction\ cycle	1	2	3	4	5	6	7	8
Instruction-1	IF	ID	EX	MA	WB			
Instruction-2		IF	ID	EX	MA	WB		
Instruction-3			IF	ID	EX	MA	WR	
Instruction-4				IF	ID	EX	MA	WR
Instruction-5					IF	ID	EX	MA
Instruction-6						IF	ID	EX
Instruction-7							IF	ID
Instruction-8								IF
	6ns	12ns	18ns	24ns	30ns	36ns	42ns	

- **CPI of each Instruction = 5**
- **Average CPI of a program having 1000 Instructions**
  - **Total clock cycles for 1000 Inst. = 5 + (1000-1) x 1 = 1004 clock cycles**
  - **Average CPI = 1004/1000 = 1.004**
- **As instruction count increases, Average CPI reduces to 1 (Minimum possible)**

Run time and Speedup

Instruction\ cycle	1	2	3	4	5	6	7	8
Instruction-1	IF	ID	EX	MA	WB			
Instruction-2		IF	ID	EX	MA	WB		
Instruction-3			IF	ID	EX	MA	WR	
Instruction-4				IF	ID	EX	MA	WR
Instruction-5					IF	ID	EX	MA
Instruction-6						IF	ID	EX
Instruction-7							IF	ID
Instruction-8								IF
	6ns	12ns	18ns	24ns	30ns	36ns	42ns	



- **Run time of each Instruction = 30ns**
- **Total run time of a program having 10 Instructions**
  - **Total run time for 10 Inst. = 30 + (10-1) x 6 = 84 ns**
  - **Run time for 10 Inst on non-pipelined machine = 30x10 = 300ns**
- **Speed up = 300ns/84ns = 3.57**

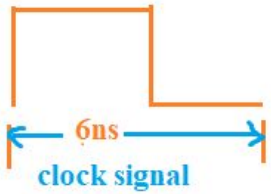
Run time and Speedup

Instruction\ cycle	1	2	3	4	5	6	7	8
Instruction-1	IF	ID	EX	MA	WB			
Instruction-2		IF	ID	EX	MA	WB		
Instruction-3			IF	ID	EX	MA	WR	
Instruction-4				IF	ID	EX	MA	WR
Instruction-5					IF	ID	EX	MA
Instruction-6						IF	ID	EX
Instruction-7							IF	ID
Instruction-8								IF
	6ns	12ns	18ns	24ns	30ns	36ns	42ns	

- **Run time of each Instruction = 30ns**
- **Total run time of a program having 100 Instructions**
  - **Total run time for 100 Inst. = 30 + (100-1) x 6 = 624 ns**
  - **Run time for 100 Inst on non-pipelined machine = 30x100 = 3000ns**
- **Speed up = 3000ns/624ns = 4.80**

Run time and Speedup

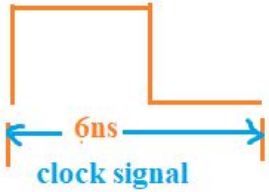
Instruction\ cycle	1	2	3	4	5	6	7	8
Instruction-1	IF	ID	EX	MA	WB			
Instruction-2		IF	ID	EX	MA	WB		
Instruction-3			IF	ID	EX	MA	WR	
Instruction-4				IF	ID	EX	MA	WR
Instruction-5					IF	ID	EX	MA
Instruction-6						IF	ID	EX
Instruction-7							IF	ID
Instruction-8								IF
	6ns	12ns	18ns	24ns	30ns	36ns	42ns	



- **Run time of each Instruction = 30ns**
- **Total run time of a program having 1000 Instructions**
  - **Total run time for 1000 Inst. = 30 + (1000-1) x 6 = 6024 ns**
  - **Run time for 1000 Inst on non-pipelined machine = 30x1000 = 30000ns**
- **Speed up = 30000ns/6024ns = 4.98**
- **As Instruction increases, speedup increases upto maximum 5, number of stages in pipeline architecture.**

Run time and Throughput

Instruction\ cycle	1	2	3	4	5	6	7	8
Instruction-1	IF	ID	EX	MA	WB			
Instruction-2		IF	ID	EX	MA	WB		
Instruction-3			IF	ID	EX	MA	WR	
Instruction-4				IF	ID	EX	MA	WR
Instruction-5					IF	ID	EX	MA
Instruction-6						IF	ID	EX
Instruction-7							IF	ID
Instruction-8								IF
	6ns	12ns	18ns	24ns	30ns	36ns	42ns	

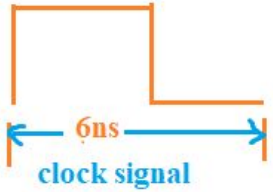


- **Run time of each Instruction = 30ns**
- **Total run time of a program having 10 Instructions**
  - **Total run time for 10 Inst. = 30 + (10-1) x 6 = 84 ns**
- **Throughput (Instruction per second) = 10/84ns**

= 119.04 million inst/sec

Run time and Throughput

Instruction\ cycle	1	2	3	4	5	6	7	8
Instruction-1	IF	ID	EX	MA	WB			
Instruction-2		IF	ID	EX	MA	WB		
Instruction-3			IF	ID	EX	MA	WR	
Instruction-4				IF	ID	EX	MA	WR
Instruction-5					IF	ID	EX	MA
Instruction-6						IF	ID	EX
Instruction-7							IF	ID
Instruction-8								IF
	6ns	12ns	18ns	24ns	30ns	36ns	42ns	

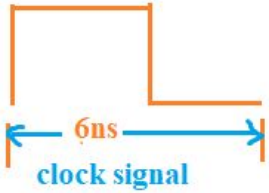


- **Run time of each Instruction = 30ns**
- **Total run time of a program having 100 Instructions**
  - **Total run time for 100 Inst. = 30 + (100-1) x 6 = 624 ns**
- **Throughput (Instruction per second) = 100/624ns**

= 160.25 million inst/sec

Run time and Throughput

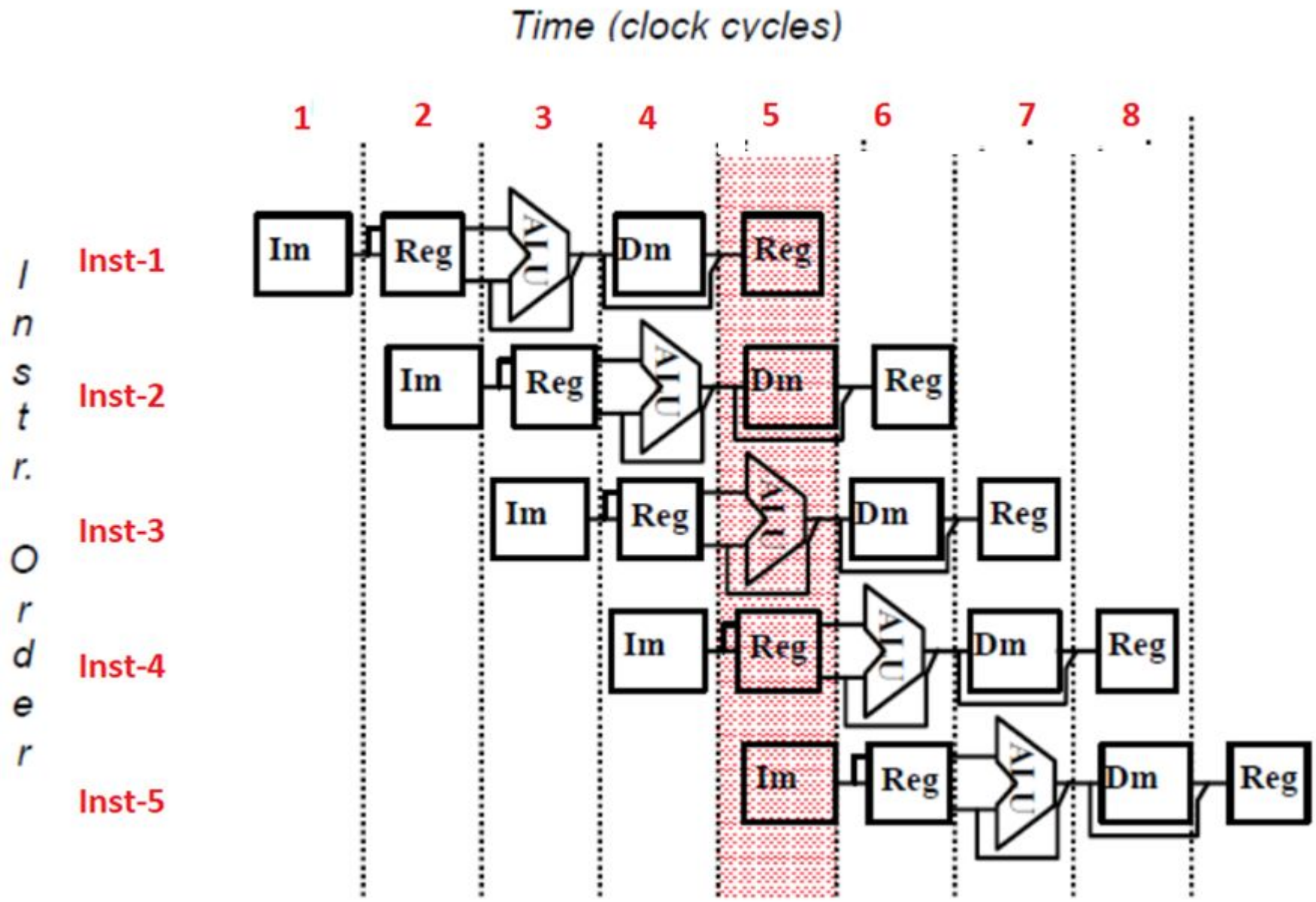
Instruction\ cycle	1	2	3	4	5	6	7	8
Instruction-1	IF	ID	EX	MA	WB			
Instruction-2		IF	ID	EX	MA	WB		
Instruction-3			IF	ID	EX	MA	WR	
Instruction-4				IF	ID	EX	MA	WR
Instruction-5					IF	ID	EX	MA
Instruction-6						IF	ID	EX
Instruction-7							IF	ID
Instruction-8								IF
	6ns	12ns	18ns	24ns	30ns	36ns	42ns	



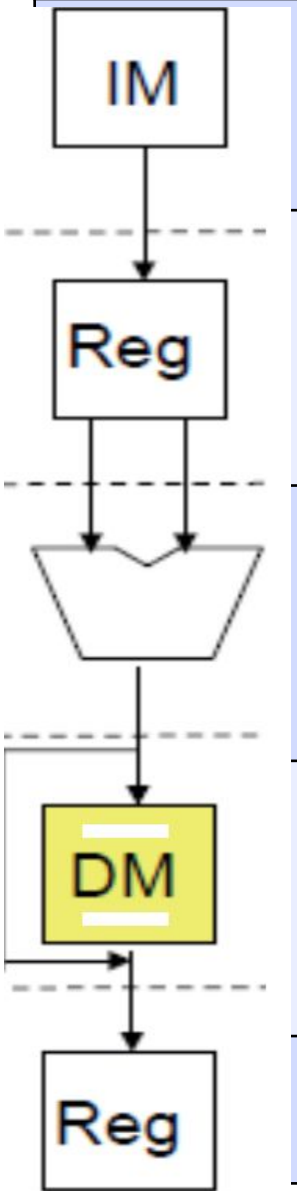
- **Run time of each Instruction = 30ns**
- **Total run time of a program having 1000 Instructions**
  - **Total run time for 1000 Inst. = 30 + (1000-1) x 6 = 6024 ns**
- **Throughput (Instruction per second) = 1000/6024ns**
  = 166 million inst/sec

conclusion

- **Pipeline does not improve processing time of a single instruction**
- **Pipeline improves run time of a program**
- **Pipeline improves throughput(Instruction/sec) of the system**


Time (clock cycles)
1
2
3
4
5
6
7
8
Instr. Order
Inst-1
Inst-2
Inst-3
Inst-4
Inst-5
Im
Reg
ALU
Dm
Reg

Pipelining (RISC): Resources used by Instructions



Resources/Cycles	1	2	3	4	5	6	7	8
	I-1	I-2	I-3	I-4	I-5			
		I-1	I-2	I-3	I-4	I-5		
			I-1	I-2	I-3	I-4	I-5	
				I-1	I-2	I-3	I-4	I-5
					I-1	I-2	I-3	I-4

RISC Pipeline: Time steps: Instructions are independent. The pipeline is full.

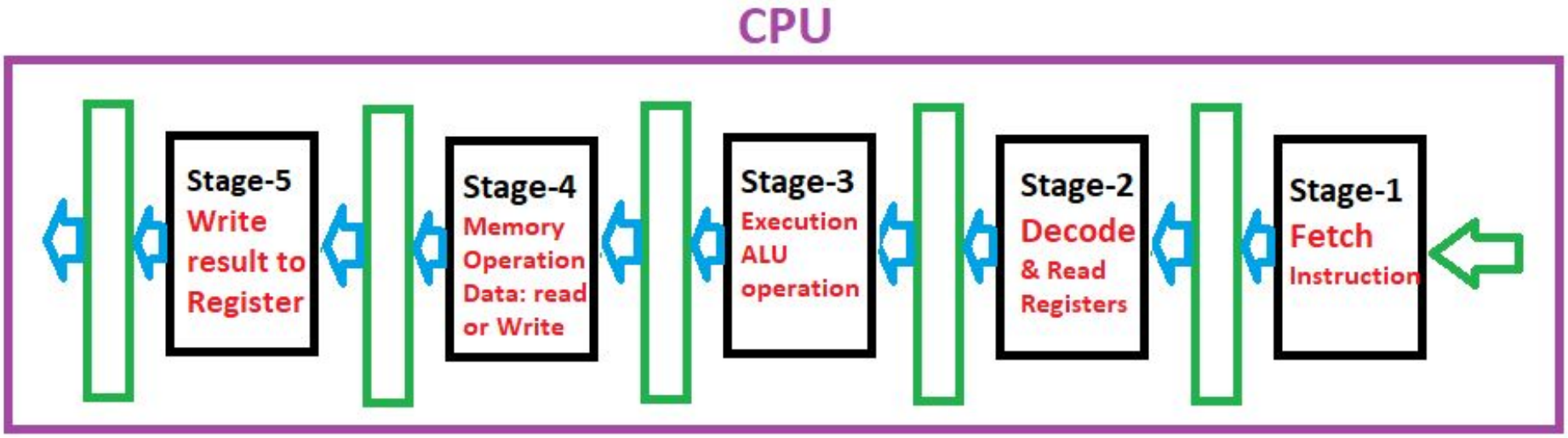
Instruction\ cycle	1	2	3	4	5	6	7	8	9
(I-1) r1 ← r0 + 10	IF	ID	EX	MA	WB				
(I-2) r3 ← r2 + 12		IF	ID	EX	MA	WB			
(I-3) r5 ← r4 + 14			IF	ID	EX	MA	WB		
(I-4) r7 ← r6 + 16				IF	ID	EX	MA	WB	
(I-5) r9 ← r8 + 18					IF	ID	EX	MA	**WB**

Over long term…..

- What is CPI: 1
- What is Instruction per cycle (IPC): 1
- Instruction execution time: 5 clock cycles
- Processing time of first five Instructions: 9 clock cycles
- What is Speed up over non-pipelined: 5 (approx.)

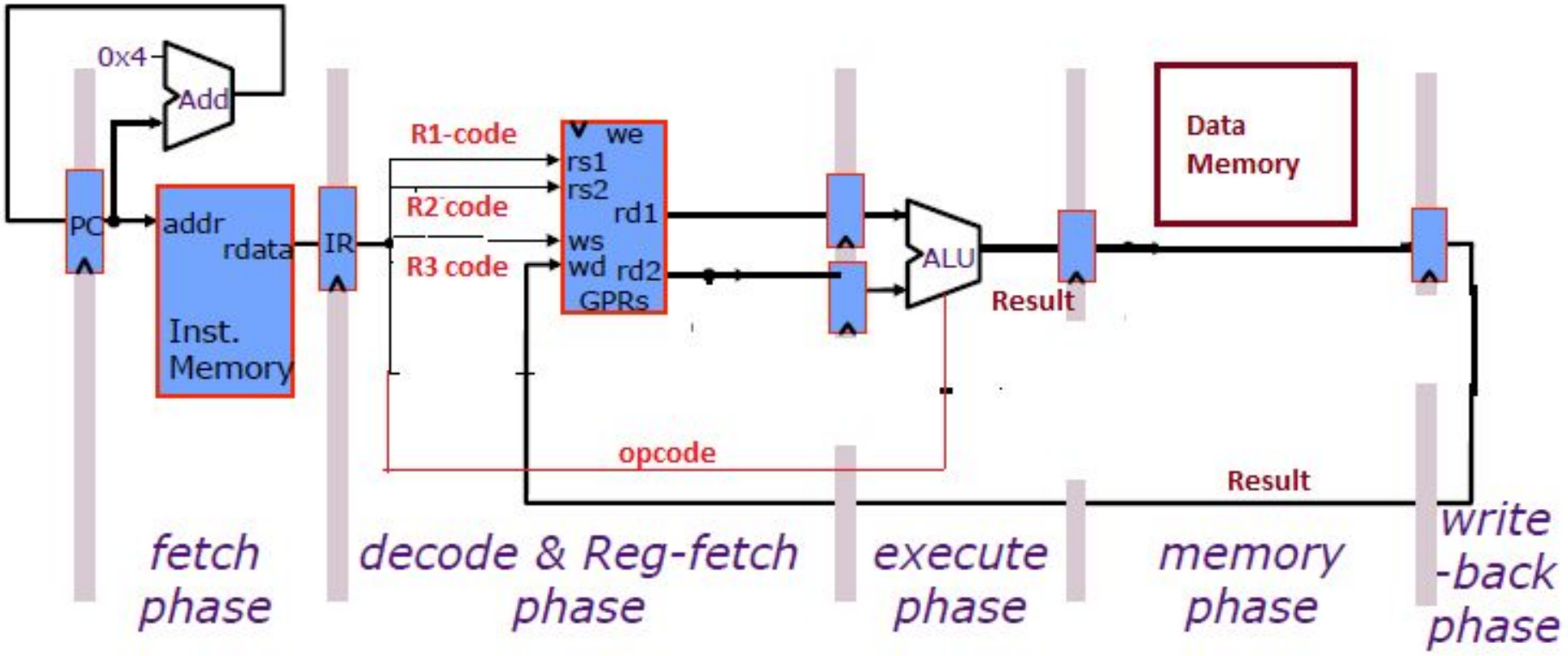
Datapath design: 5-stage pipelining RISC-processors

- 5-stage processing of each Instruction, in general, although some instructions may NOT require all stages
- **ALU** instructions are register based
- **LOAD** and **STORE** Instructions are used to read data from RAM and store to RAM
- **Harvard Architecture**: Separate memories for Instructions and Data
- Latches/buffers between stages to allow sections work simultaneously on different Instructions



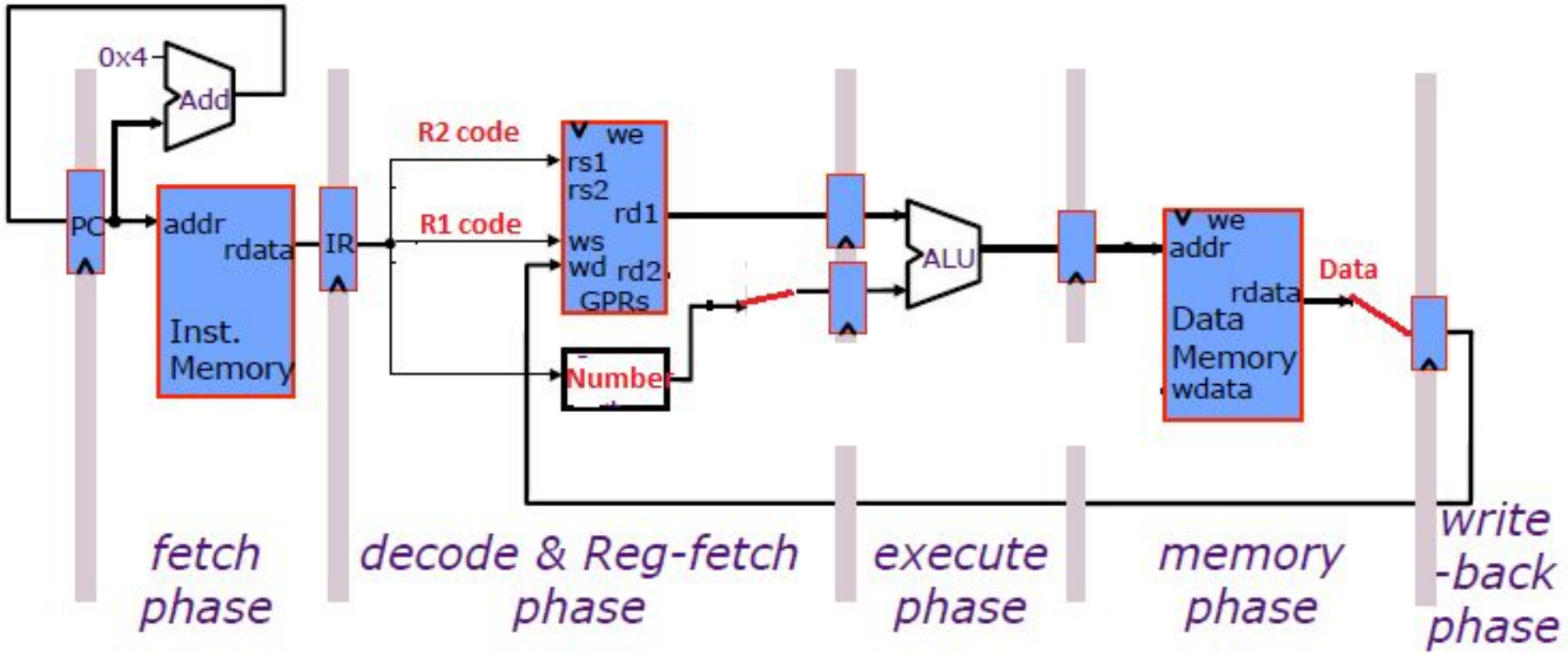
Datapath design: ALU Instruction (RISC)

ADD	R1 (Data-1)	R2 (Data-2)	R3 (result field)



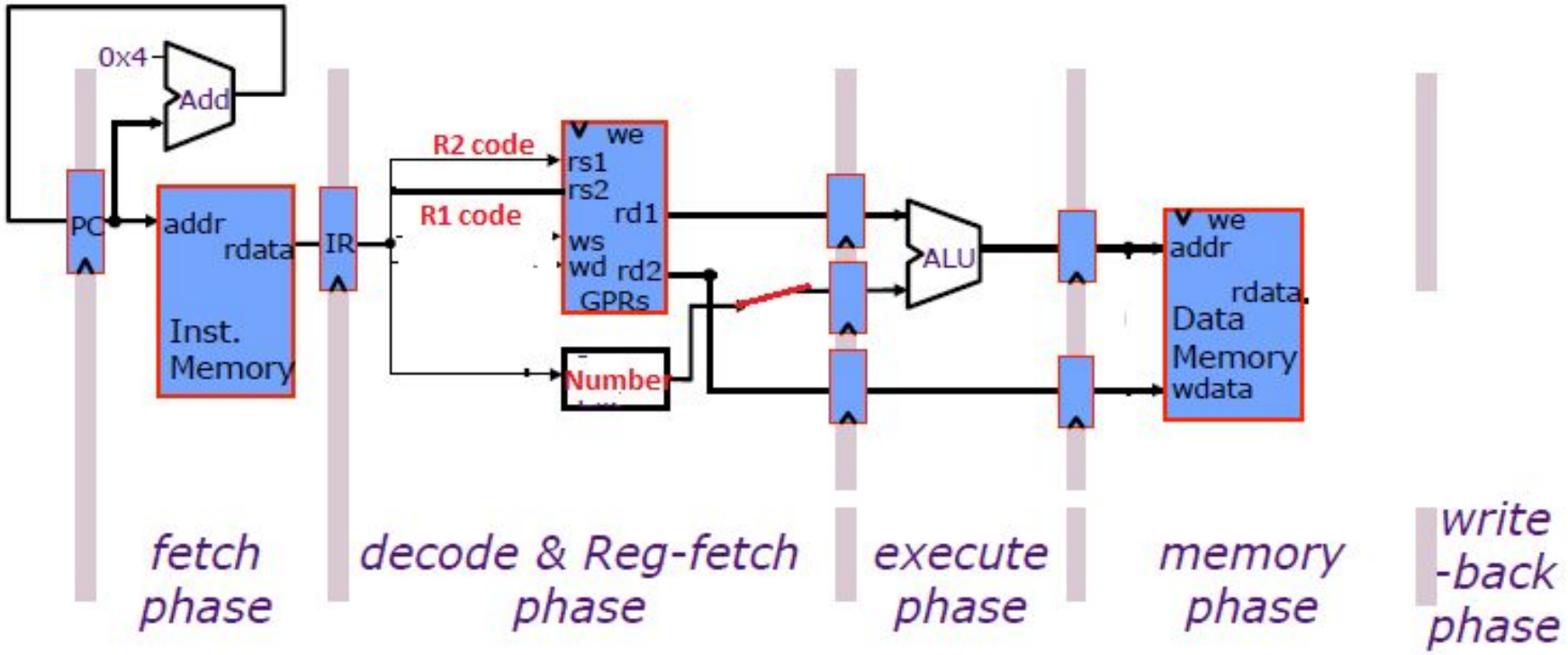
LOAD Instruction

LOAD	R1 (Destination)	NUMBER (R2) (Source field)

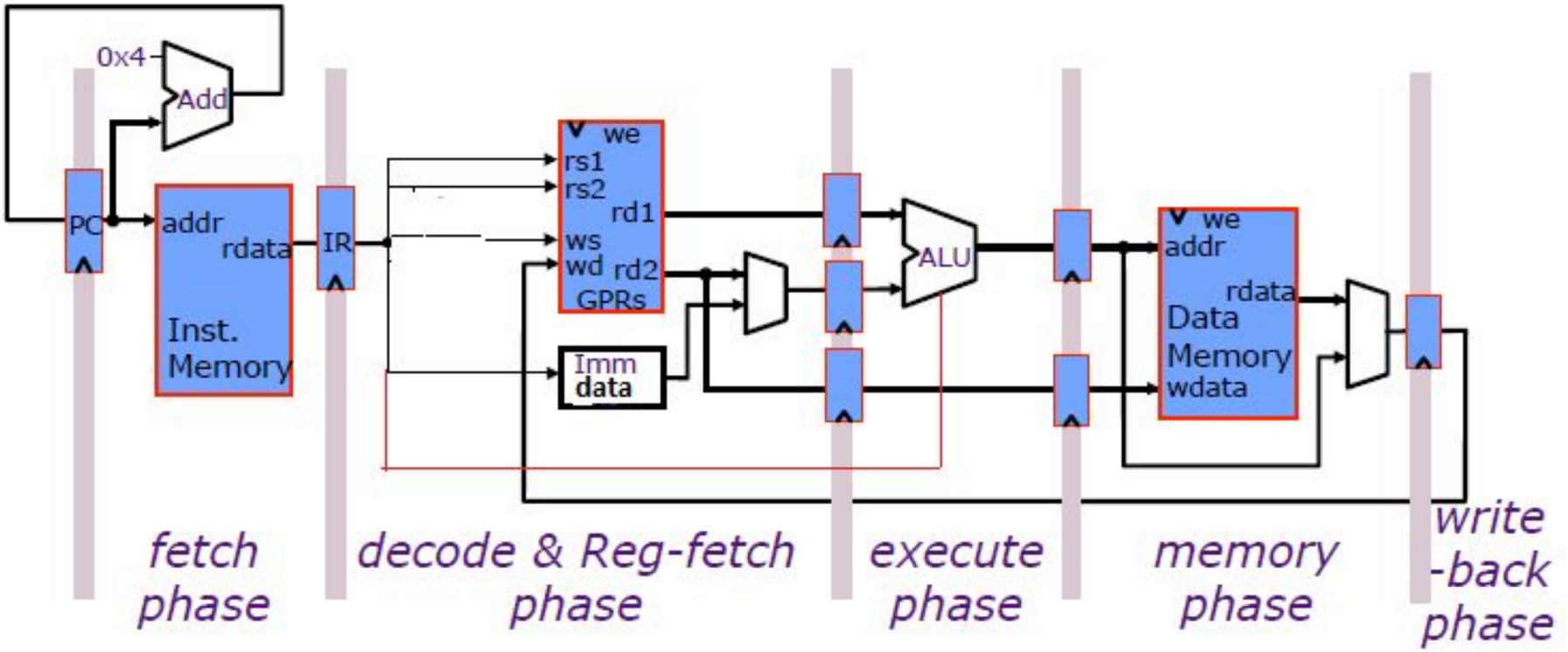


STORE Instruction

STORE	R1 (SOURCE)	NUMBER (R2) (DESTINATION)



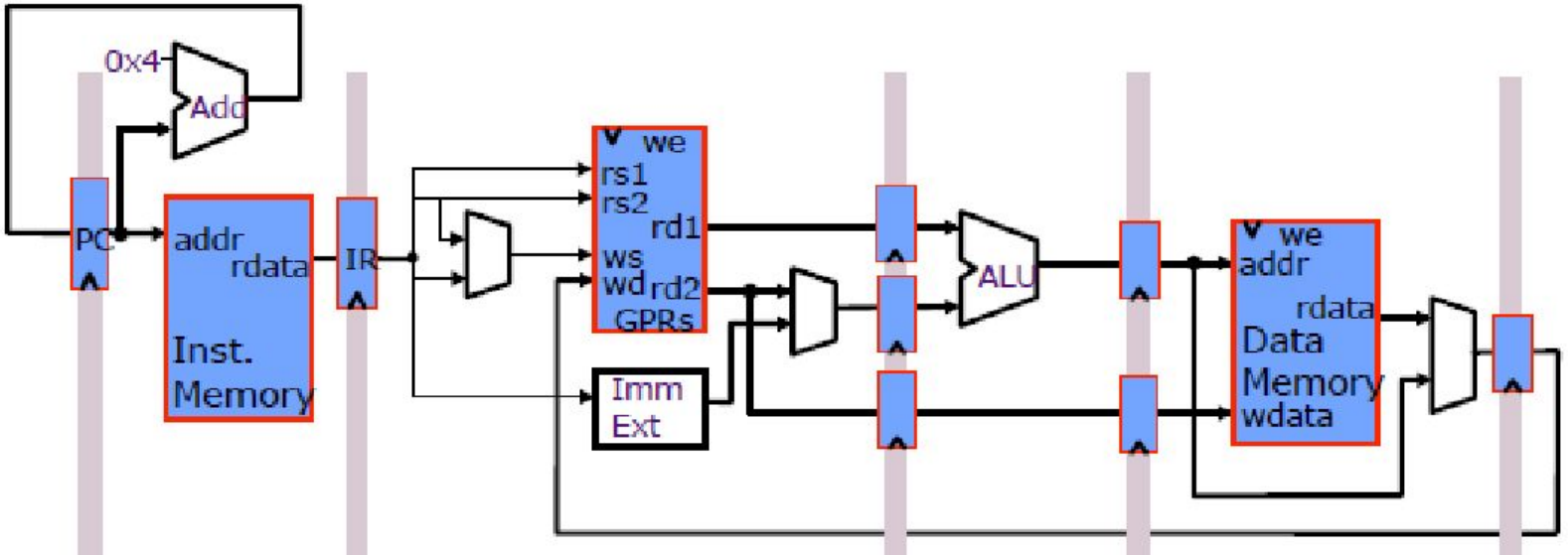
Pipelined datapath



Pipelining (RISC): Time steps
Clock cycles progress left to right
Instructions progress top to bottom
Time at which each instruction is present in each pipeline stage is shown by labelling appropriate cell with pipeline name
Information from one instruction to any successor must always move from left to right

Instruction\ cycle	1	2	3	4	5	6	7	8
Instruction-1	IF	ID	EX	MA	WB			
Instruction-2		IF	ID	EX	MA	WB		
Instruction-3			IF	ID	EX	MA		
Instruction-4				IF	ID	EX		
Instruction-5					IF	ID		
Instruction-6						IF		
Instruction-7								
Instruction-8								

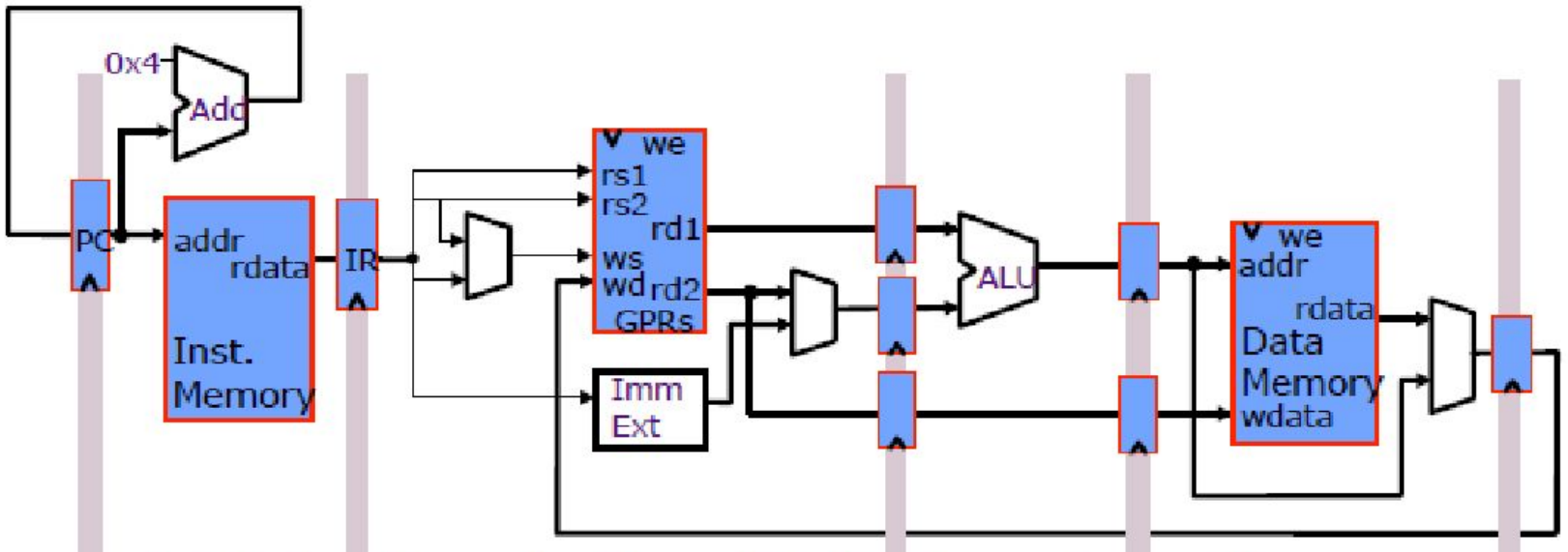
5-Stage Pipelined Execution



time	t0	t1	t2	t3	t4	t5	t6	t7	. . . .
instruction1	IF_1	ID_1	EX_1	MA_1	WB_1				
instruction2		IF_2	ID_2	EX_2	MA_2	WB_2			
instruction3			IF_3	ID_3	EX_3	MA_3	WB_3		
instruction4				IF_4	ID_4	EX_4	MA_4	WB_4	
instruction5					IF_5	ID_5	EX_5	MA_5	WB_5

5-Stage Pipelined Execution

Resource Usage Diagram



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

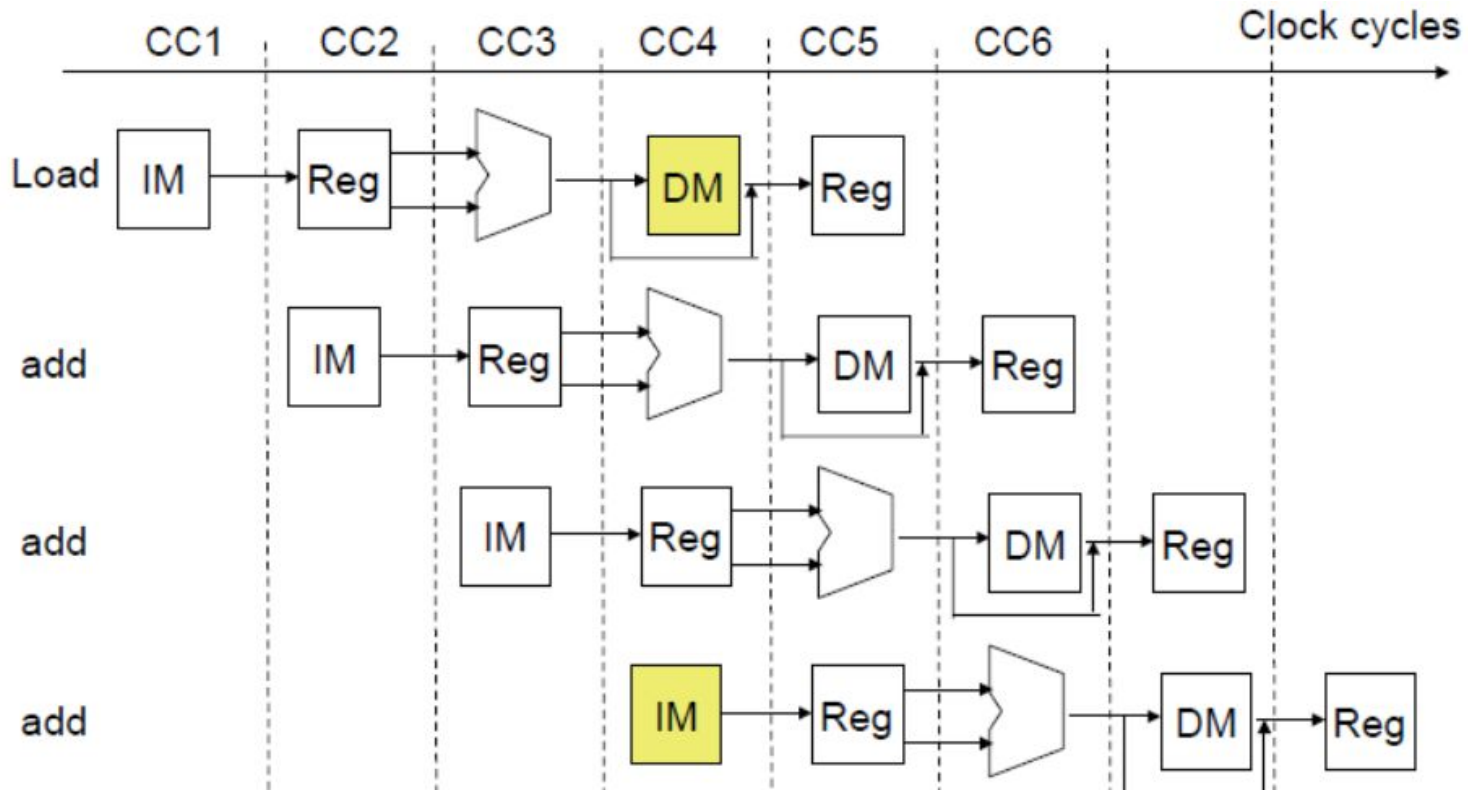
	time	t0	t1	t2	t3	t4	t5	t6	t7	. . . .
Resources	*IF*	I_1	I_2	I_3	I_4	I_5				
	ID		I_1	I_2	I_3	I_4	I_5			
	EX			I_1	I_2	I_3	I_4	I_5		
	MA				I_1	I_2	I_3	I_4	I_5	
	WB					I_1	I_2	I_3	I_4	I_5

Example Problem

Clock cycles

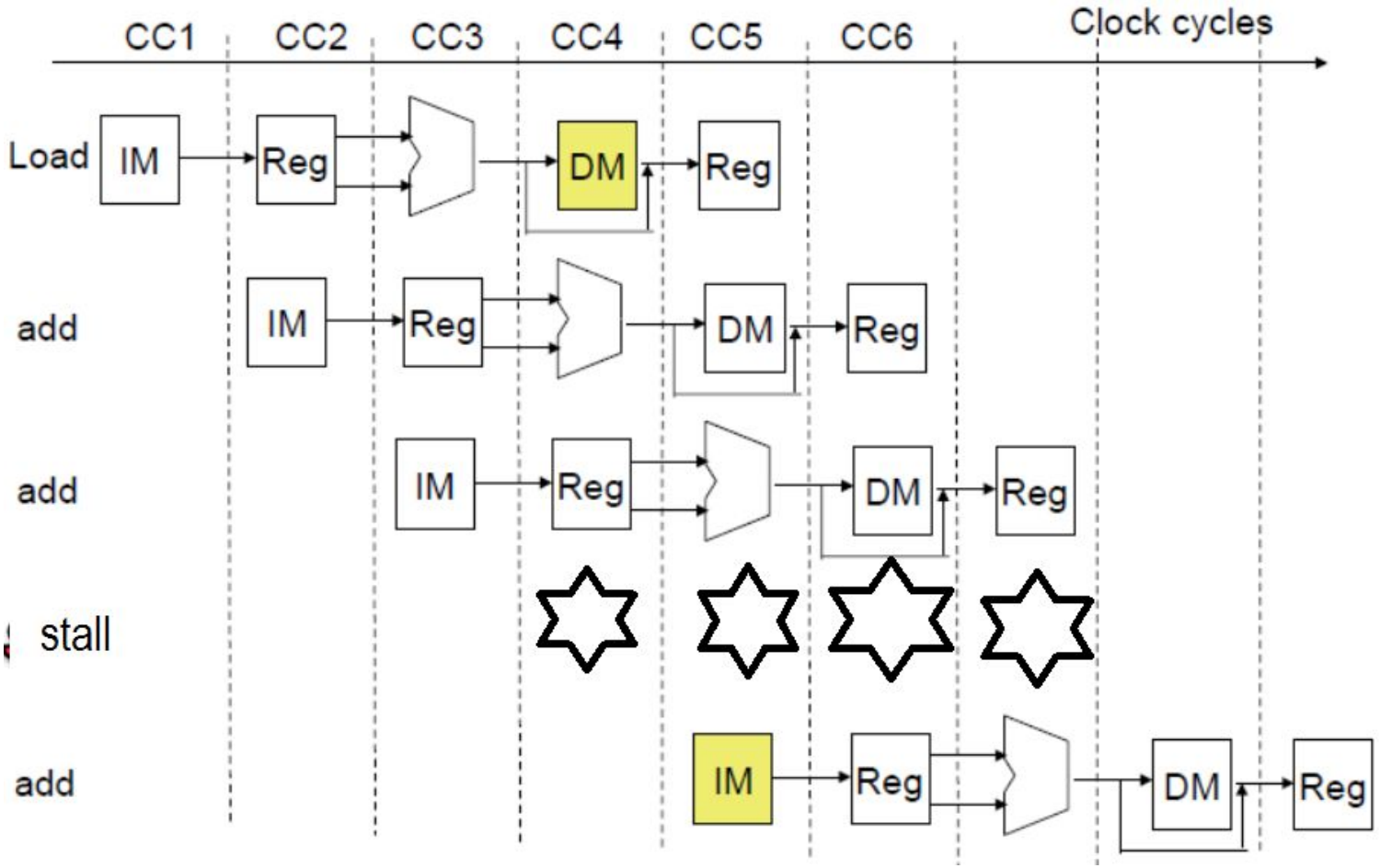
Instruction	1	2	3	4	5	6	7	8	9	10	11
Instruction-1											
Instruction-2											
Instruction-3											
Instruction-4											
Instruction-5											
Instruction-6											
Instruction-7											
Instruction-8											
Instruction-9											
Instruction-10											
Instruction-11											

Pipeline Implementation challenges: Structural Hazard

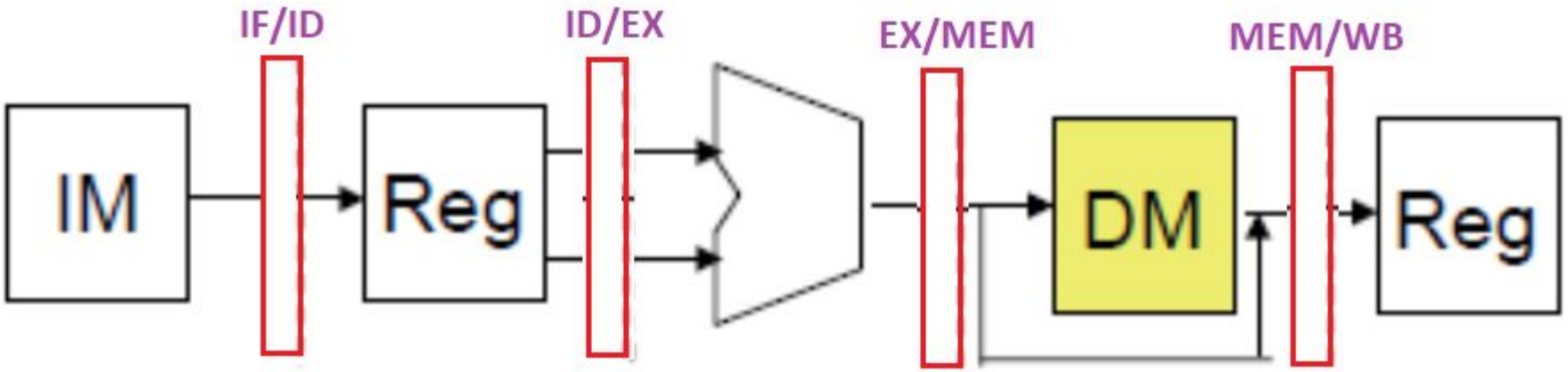


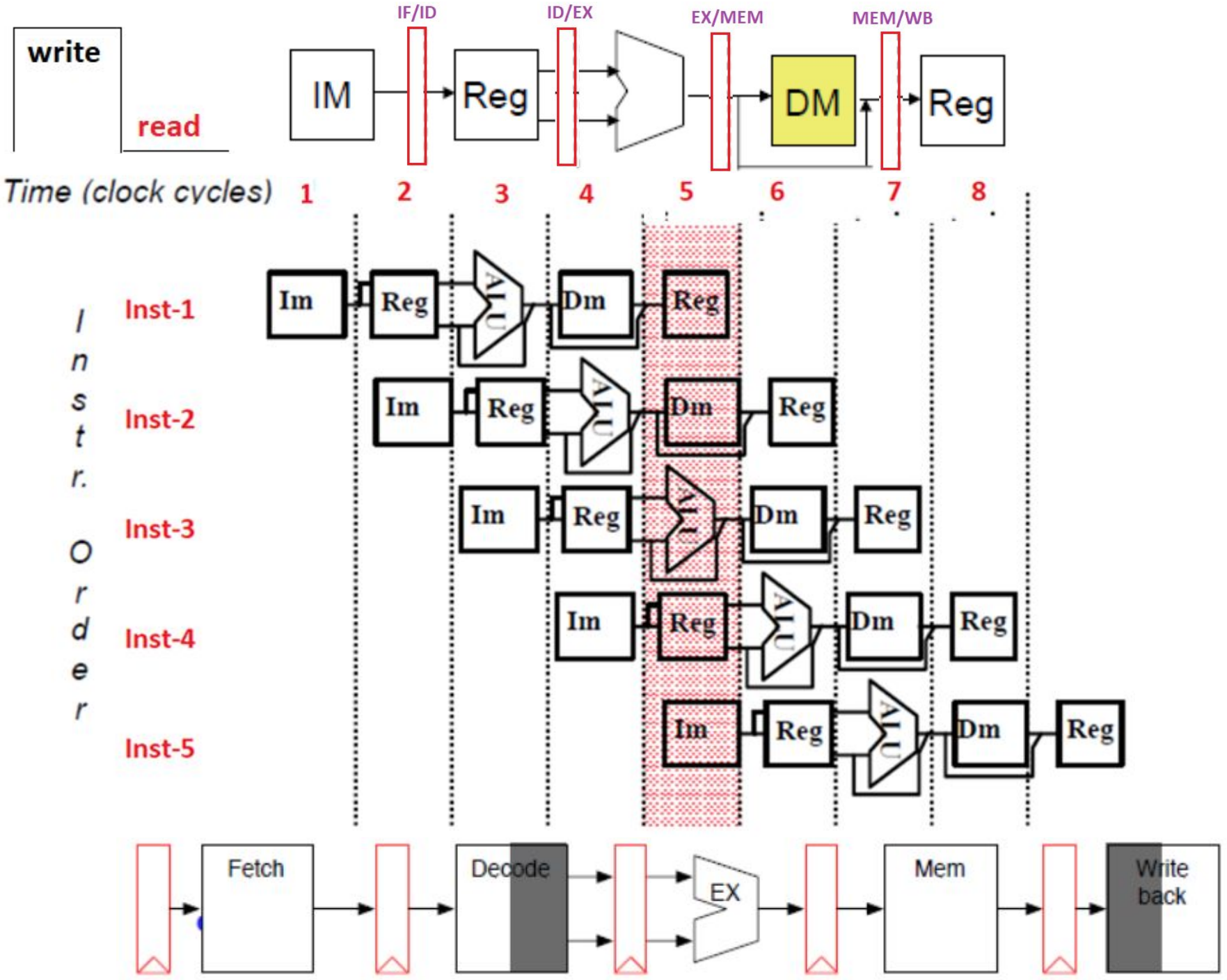
At clock cycle 4, 1st(load) and add(4th) instructions need to access RAM. If a single memory is used, both IF and MA cannot proceed at clock cycle 4. The pipeline will be stalled

Pipeline Implementation challenges: Structural Hazard



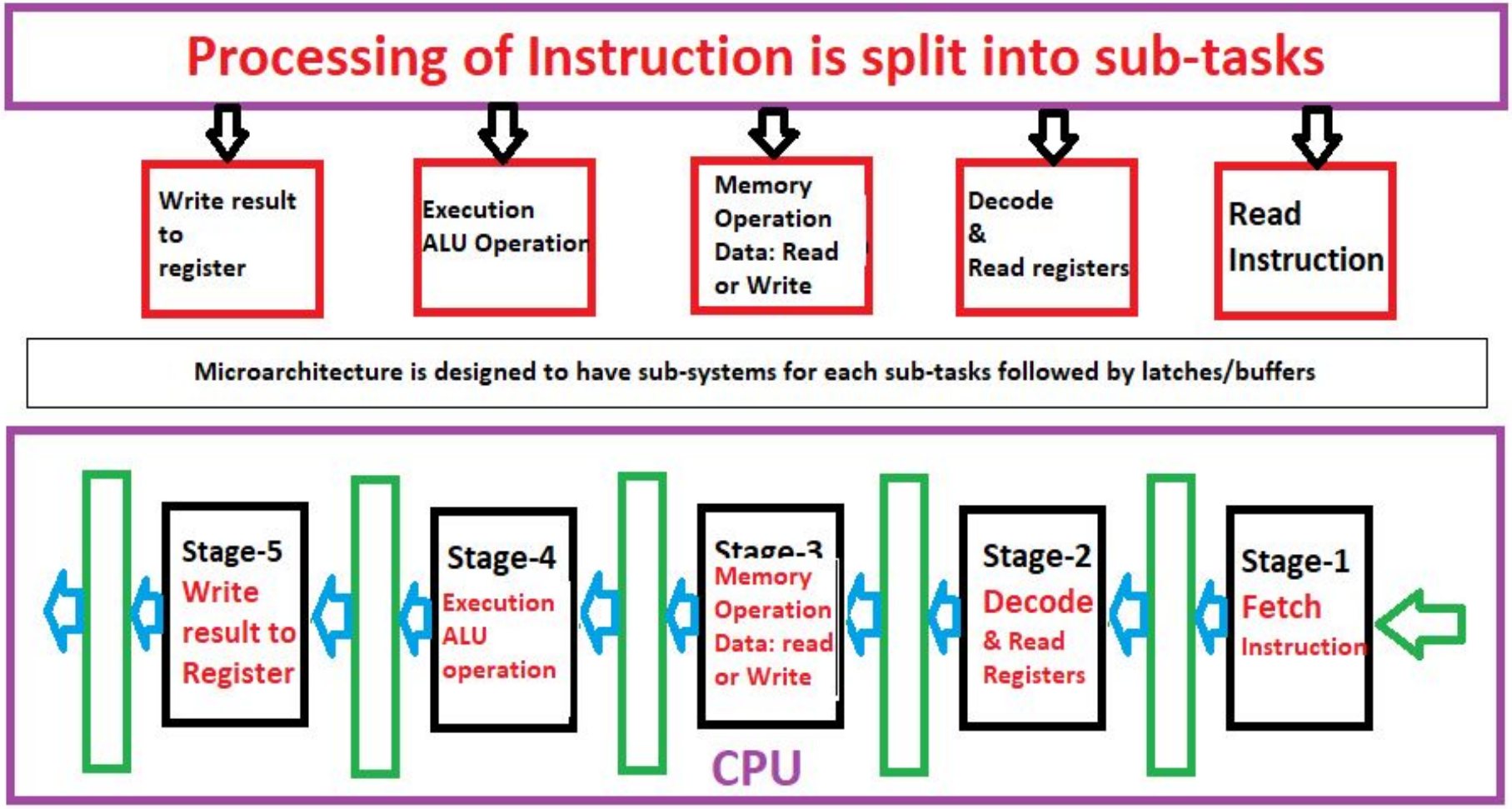
- Both the decode and write back stage have to access the register file.
- There is only one registers file. A structural hazard!!
- Solution: Write early, read late
  - Writes occur at the clock edge and complete long before the end of the cycle
  - This leave enough time for the outputs to settle for the reads.
- Hazard avoided!




write
read
IF/ID
ID/EX
EX/MEM
MEM/WB
IM
Reg
DM
Reg
Time (clock cycles)
1
2
3
4
5
6
7
8
Instr. Order
Inst-1
Inst-2
Inst-3
Inst-4
Inst-5
Im
Reg
ALU
Dm
Reg
Fetch
Decode
EX
Mem
Write back

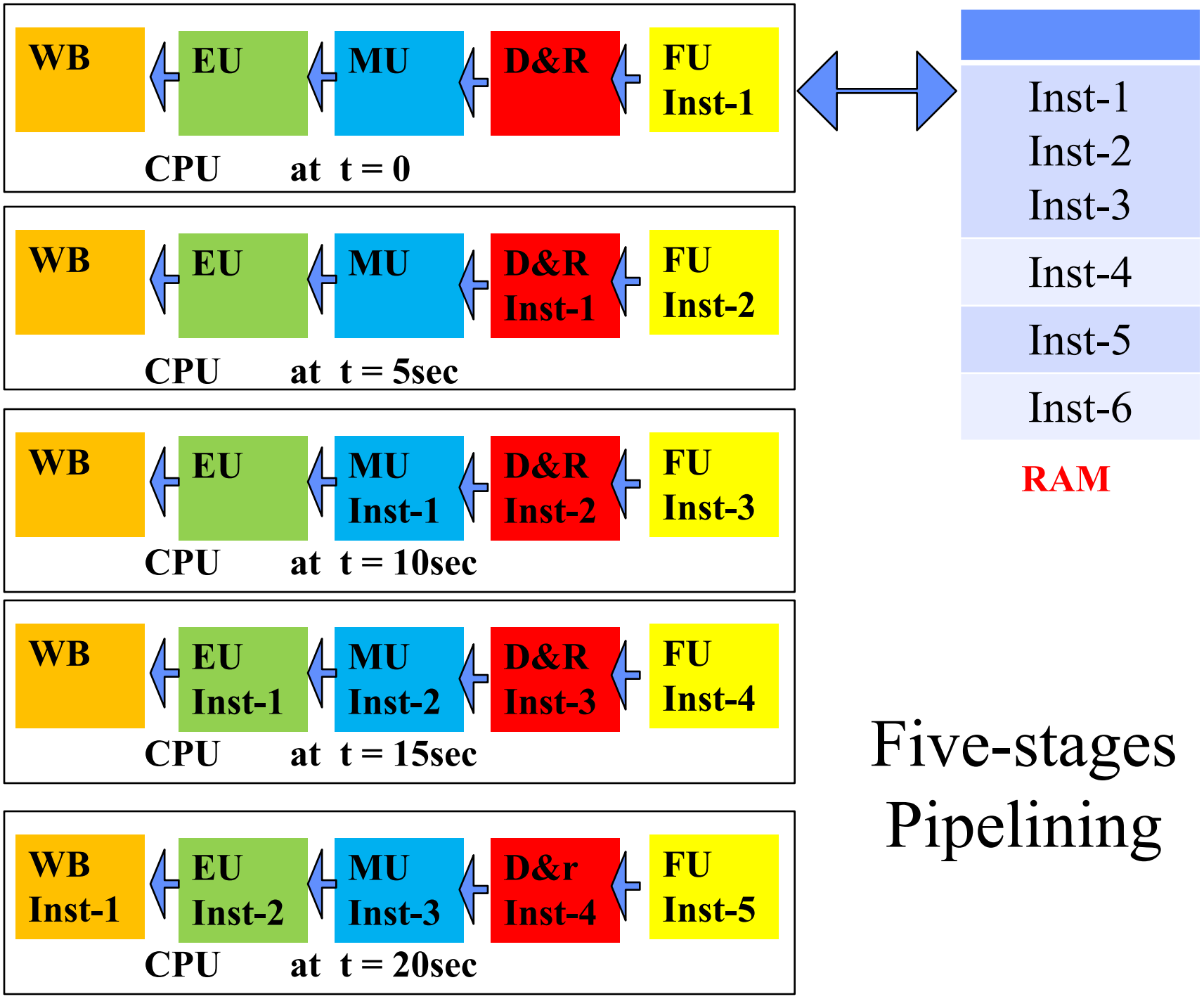
5-stage Pipelining of CISC processors

- **ALU instructions are Register-based as well as Memory based**
- **ALU instructions can access Memory**



Five-stage pipelining(CISC)

Five-stage processing of an instruction	Five-stages Hardware Within processor
Fetch Instruction from Memory (IF)	**Fetch Unit (FU)**
Decode Instruction (ID)	Decode Unit (D&R)
Calculate data address and read data from memory (MA)	Memory Data (MU)
Execute the operation (EX)	Execute Unit (EU)
Write the result in register (WB)	Write Unit (WB)


WB
EU
MU
D&R
FU
Inst-1
CPU at t = 0
WB
EU
MU
D&R
Inst-1
FU
Inst-2
CPU at t = 5sec
WB
EU
MU
Inst-1
D&R
Inst-2
FU
Inst-3
CPU at t = 10sec
WB
EU
Inst-1
MU
Inst-2
D&R
Inst-3
FU
Inst-4
CPU at t = 15sec
WB
Inst-1
EU
Inst-2
MU
Inst-3
D&r
Inst-4
FU
Inst-5
CPU at t = 20sec
Inst-1
Inst-2
Inst-3
Inst-4
Inst-5
Inst-6
RAM
Five-stages
Pipelining

Pipelining (CISE): Time steps
Clock cycles progress left to right
Instructions progress top to bottom
Time at which each instruction is present in each pipeline stage is shown by labelling appropriate cell with pipeline name
Information from one instruction to any successor must always move from left to right

Instruction\ cycle	1	2	3	4	5	6	7	8
Instruction-1	IF	ID	MA	EX	WB			
Instruction-2		IF	ID	MA	EX	WB		
Instruction-3			IF	ID	MA	EX		
Instruction-4				IF	ID	MA		
Instruction-5					IF	ID		
Instruction-6						IF		
Instruction-7								
Instruction-8								

Pipelining: Six stages

- **Fetch instruction (FI):** Read the next expected instruction into a buffer.
- **Decode instruction (DI):** Determine the opcode and the operand specifiers.
- **Calculate operands (CO):** Calculate the effective address of each source operand. This may involve displacement, register indirect, indirect, or other forms of address calculation.
- **Fetch operands (FO):** Fetch each operand from memory. Operands in registers need not be fetched.
- **Execute instruction (EI):** Perform the indicated operation and store the result, if any, in the specified destination operand location.
- **Write operand (WO):** Store the result in memory.

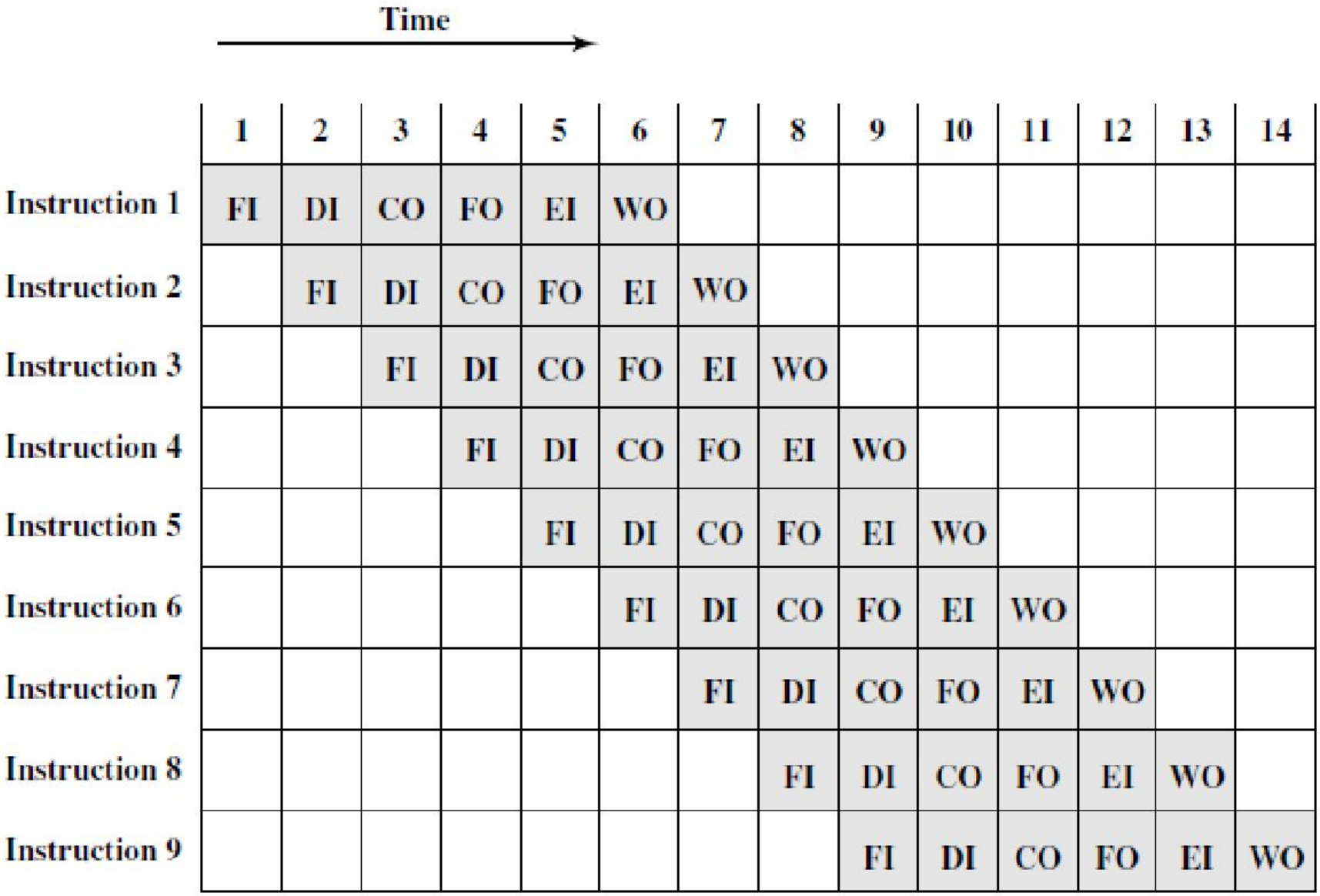


Figure 12.10 Timing Diagram for Instruction Pipeline Operation

Pipelining Performance

We develop some simple measures of pipeline performance and relative speedup. The cycle time τ of an instruction pipeline is the time needed to advance a set of instructions one stage through the pipeline; each column in Figures represents one cycle time. The cycle time can be determined as

$$\tau = \max_i [\tau_i] + d = \tau_m + d \qquad 1 \leq i \leq k$$

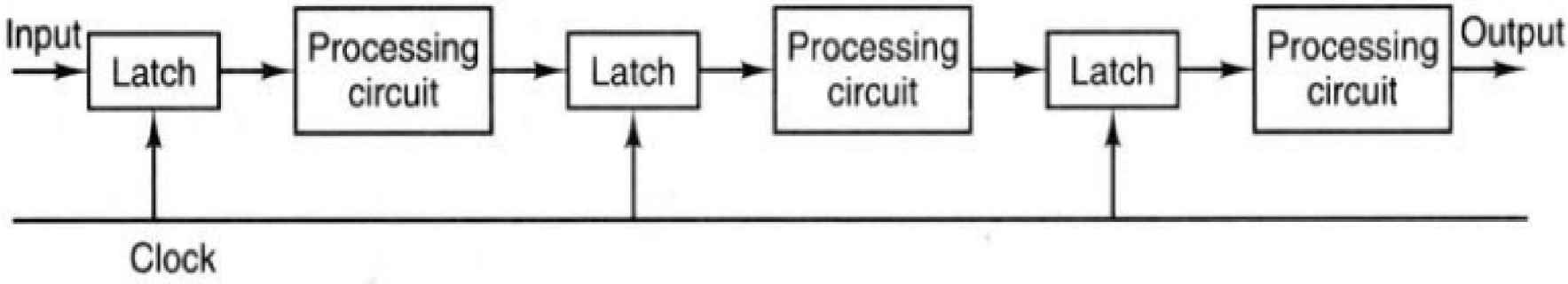
τ_i = time delay of the circuitry in the ith stage of the pipeline

τ_m = maximum stage delay (delay through stage which experiences the largest delay)

k = number of stages in the instruction pipeline

d = time delay of a latch, needed to advance signals and data from one stage to the next

In general, the time delay d is equivalent to a clock pulse and $\tau_m \gg d$.



Pipelining Performance

suppose that n instructions are processed, with no branches. Let $T_{k,n}$ be the total time required for a pipeline with k stages to execute n instructions. Then

$$T_{k,n} = [k + (n - 1)]\tau$$

A total of k cycles are required to complete the execution of the first instruction, and the remaining $n - 1$ instructions require $n - 1$ cycles.

Now consider a processor with equivalent functions but no pipeline, and assume that the instruction cycle time is $k\tau$. The speedup factor for the instruction pipeline compared to execution without the pipeline is defined as

$$S_k = \frac{T_{1,n}}{T_{k,n}} = \frac{nk\tau}{[k + (n - 1)]\tau} = \frac{nk}{k + (n - 1)}$$

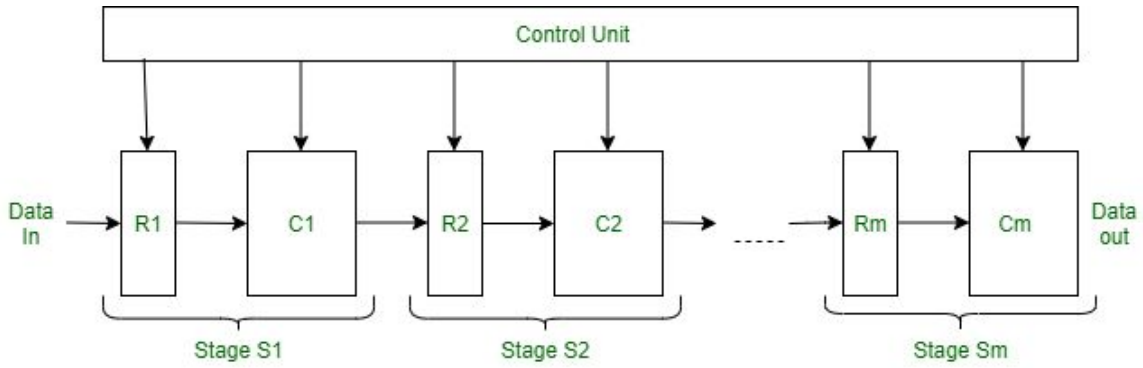
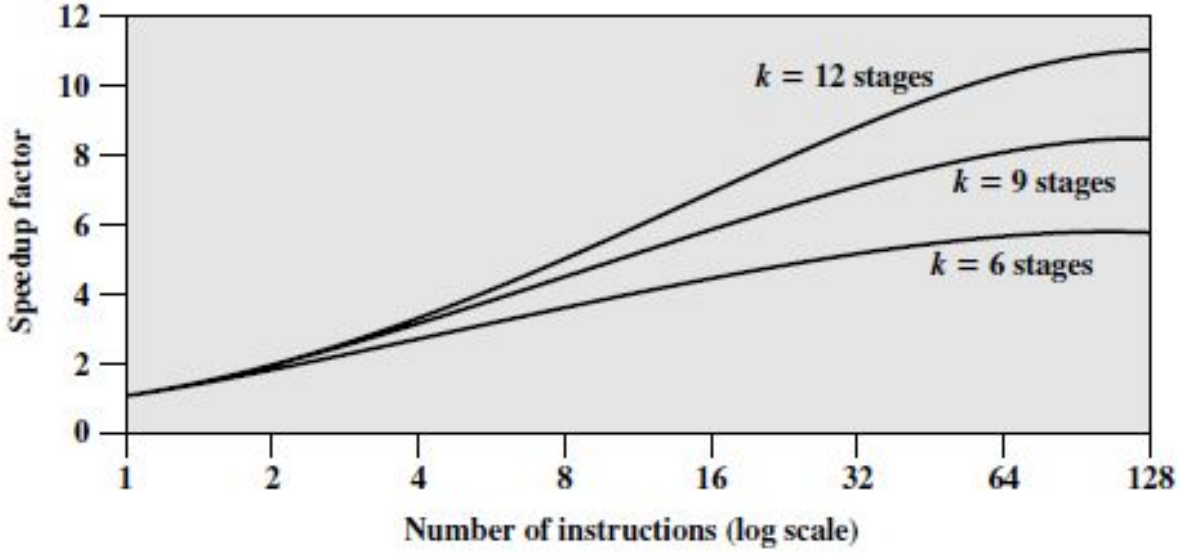
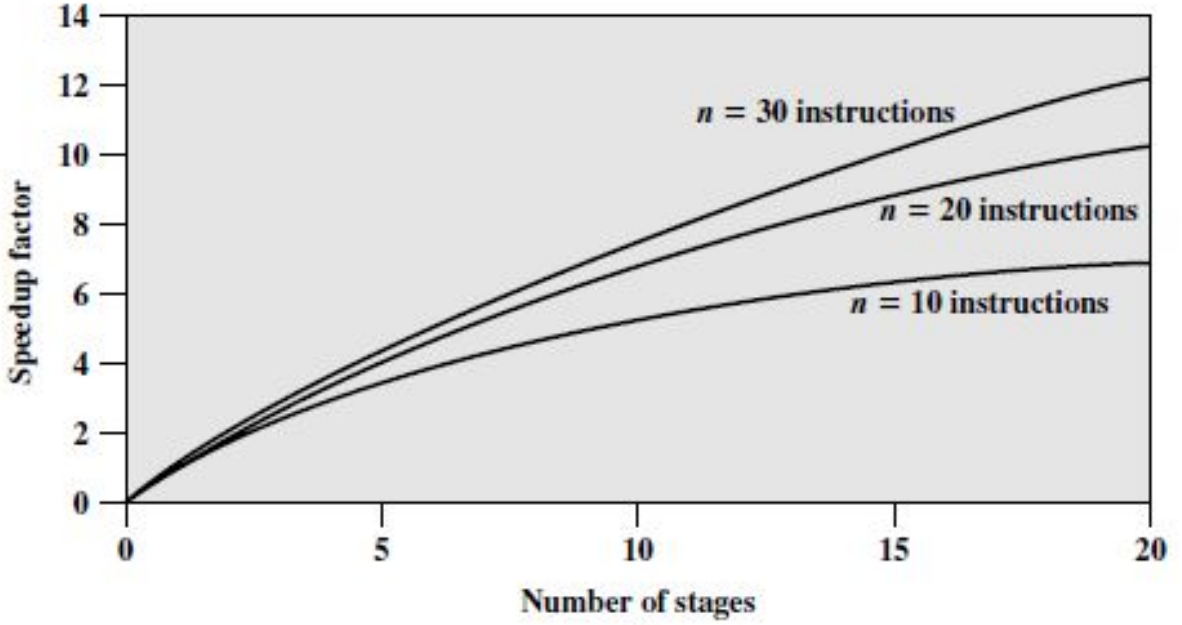


Figure - Structure of a Pipeline Processor



(a)



(b)

Speedup Factors with Instruction Pipelining

Example Problem

Clock cycles

Instruction	1	2	3	4	5	6	7	8	9	10	11
Instruction-1											
Instruction-2											
Instruction-3											
Instruction-4											
Instruction-5											
Instruction-6											
Instruction-7											
Instruction-8											
Instruction-9											
Instruction-10											
Instruction-11											

Pipeline Hazards

- Hazards are pipeline events that restrict the pipeline flow
- They occur in circumstances where two or more activities cannot proceed in parallel
- There are three types of hazard:
  - Structural Hazards
    - Arise from resource conflicts, when a set of actions have to be performed sequentially because there is not sufficient resource to operate in parallel
  - Data Hazards
    - Occur when one instruction depends on the result of a previous instruction, and that result is not yet available. These hazards are exposed by the overlapped execution of instructions in a pipeline
  - Control Hazards
    - These arise from the pipelining of branch instructions, and other activities that change the PC.

- **Data hazards**
  - Instruction depends on data that is not ready yet

  (Data dependencies through the register file)

- **Control hazards**
  - The choice of next instruction depends on results that aren't ready yet

  (We'll see this next)

- **Structural hazards**
  - Hardware cannot support a combination of instructions

  (We saw this with reading and writing the register file at the same time)

Instructions interact with each other in pipeline

- An instruction in the pipeline may need a resource being used by another instruction in the pipeline → *structural hazard*
- An instruction may depend on something produced by an earlier instruction
  - Dependence may be for a data value
    → data hazard
  - Dependence may be for the next instruction's address
    → control hazard (branches, exceptions)

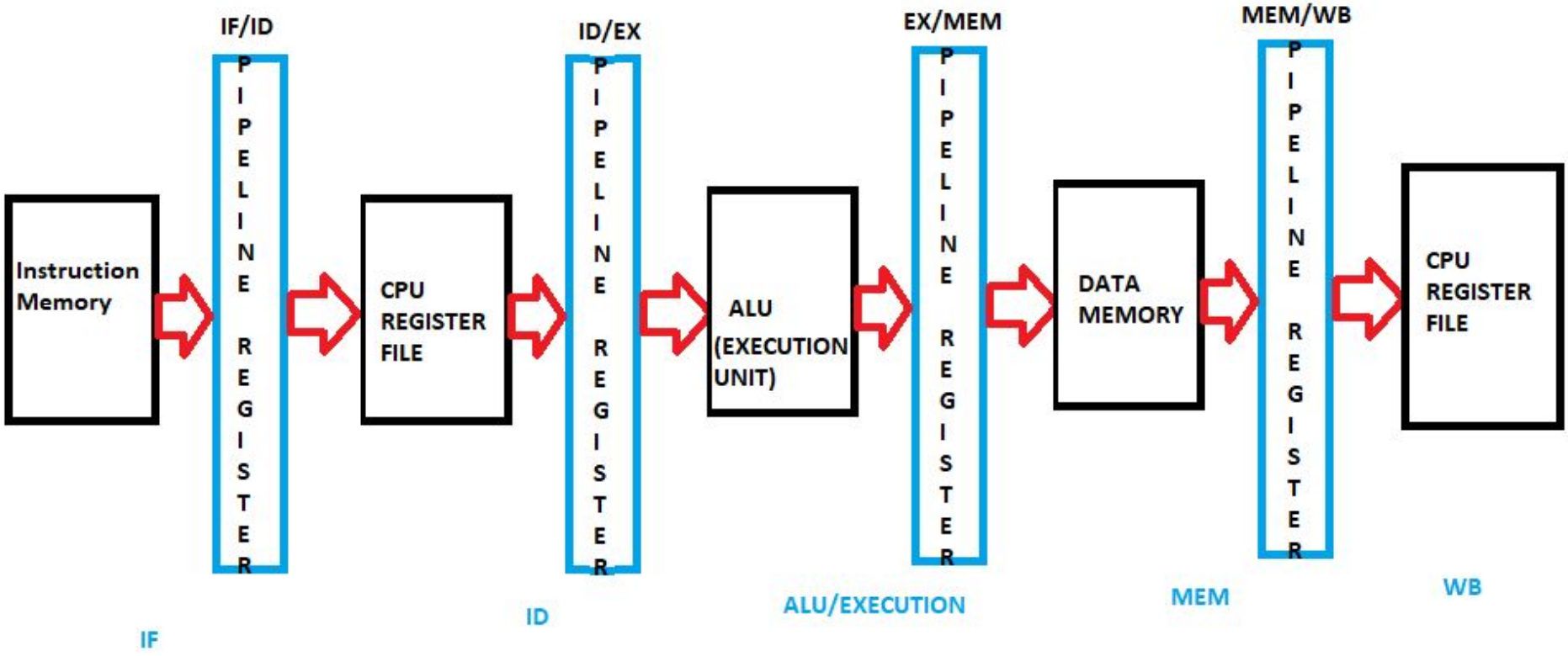
Pipeline hazard

- A **pipeline hazard** occurs when the pipeline, or some portion of the pipeline, must stall because conditions do not permit continued execution. Such a pipeline stall is also referred to as a *pipeline bubble*. There are three types of hazards:

- Structural/Resource

- Data

- Control


IF/ID
ID/EX
EX/MEM
MEM/WB
Instruction Memory
PIPELINE REGISTER
CPU REGISTER FILE
PIPELINE REGISTER
ALU (EXECUTION UNIT)
PIPELINE REGISTER
DATA MEMORY
PIPELINE REGISTER
CPU REGISTER FILE
IF
ID
ALU/EXECUTION
MEM
WB

Resolving Structural Hazards

- Structural hazards occurs when two instruction need same hardware resource at same time
  - Can resolve in hardware by stalling newer instruction till older instruction finished with resource
- A structural hazard can always be avoided by adding more hardware to design
  - E.g., if two instructions both need a port to memory at same time, could avoid hazard by adding second port to memory
- Our 5-stage pipe has no structural hazards by design
  - Thanks to MIPS ISA, which was designed for pipelining

Data Hazards

- A data hazard occurs when there is a conflict in the access of an operand location.
- Two instructions in a program are to be executed in sequence and both access a particular memory or register operand. If the two instructions are executed in strict sequence, no problem occurs.
- However, if the instructions are executed in a pipeline, then it is possible for the operand value to be updated in such a way as to produce a different result than would occur with strict sequential execution. In other words, the program produces an incorrect result because of the use of pipelining.

Types of Data Hazards:

• **Read after write (RAW), or true dependency:** An instruction modifies a register or memory location and a succeeding instruction reads the data in that memory or register location. A hazard occurs if the read takes place before the write operation is complete. (read too soon)

```
ADD R1, R2, R3
ADD R4, R1, R5
```

• **Write after read (WAR), or antidependency:** An instruction reads a register or memory location and a succeeding instruction writes to the location. A hazard occurs if the write operation completes before the read operation takes place. (written too soon)

```
AND   R1, R2, R3;  READ R2
OR      R2, R4, R5 ; WRITE TO R2
```

• **Write after write (WAW), or output dependency:** Two instructions both write to the same location. A hazard occurs if the write operations take place in the reverse order of the intended sequence.(written out-of-order)

```
AND  R1, R2, R3; R1 IS USED TO SAVE RESULT
OR     R1, R4, R5; R1 IS USED TO SAVE RESULT
```

Resolving Data Hazards

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

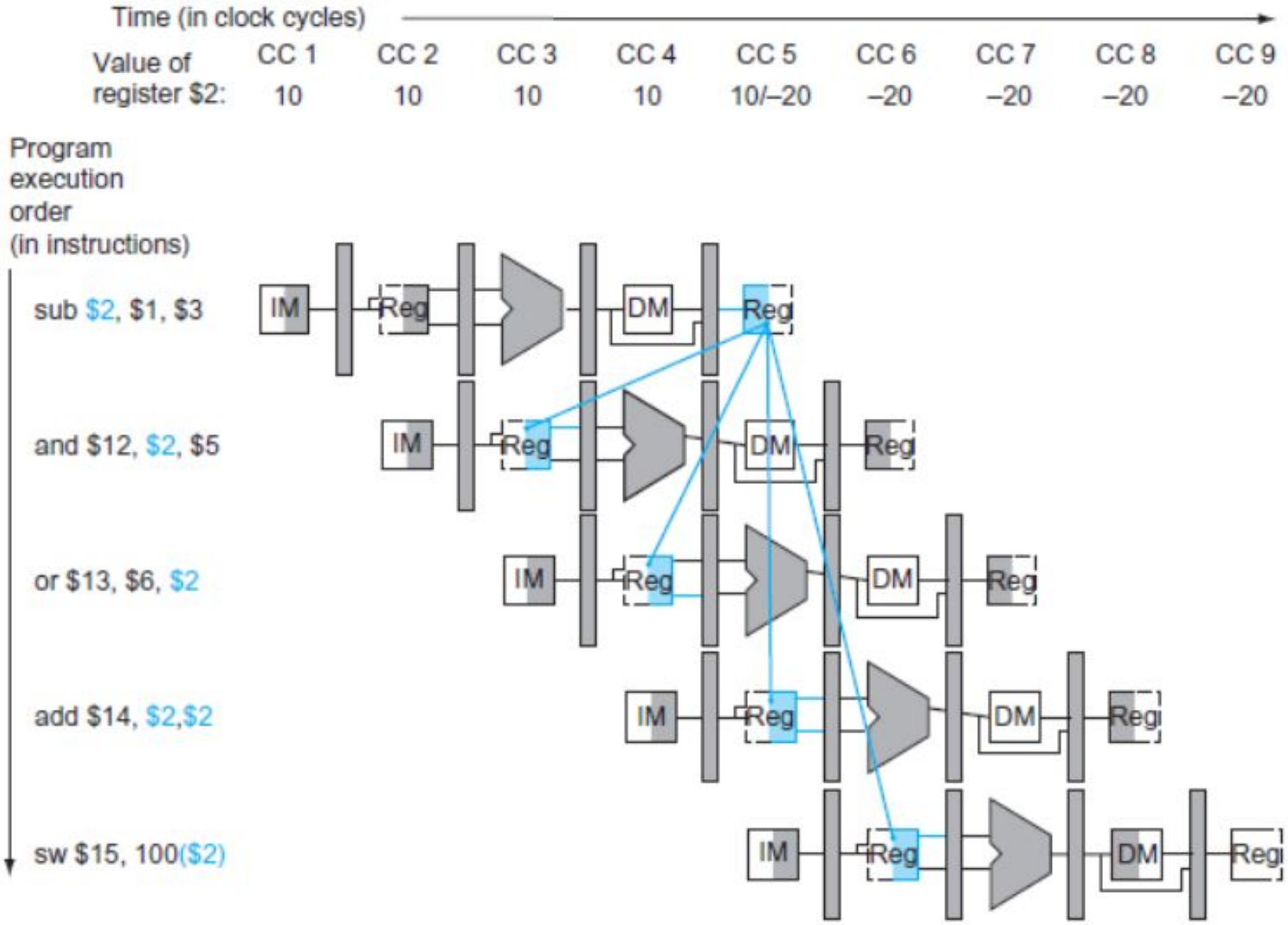
Strategy 2: *Route data as soon as possible after it is calculated to the earlier pipeline stage* → *bypass*

Strategy 3: *Speculate on the dependence*
Two cases:
Guessed correctly → no special action required
Guessed incorrectly → kill and restart

DATA Dependency/Data Hazard (RISC):
Result of an instruction is used as input in the next instruction
ADD R1, R2, R3; R1 is result field
SUB R6, R1, R5; R1 is operand

Instruction\ cycle	**1**	**2**	**3**	**4**	**5**	**6**
ADD R1,R2, R3	IF	ID	EX	MA	WB (R1 updated)	
SUB R6, R1, R5		IF	ID (R1 is read) Old value!	EX	MA	WB
Instruction-3			IF	ID	EX	MA
Instruction-4				IF	ID	EX

Result in 2nd Instruction will be wrong!


Time (in clock cycles)
Value of register $2:
CC 1
CC 2
CC 3
CC 4
CC 5
CC 6
CC 7
CC 8
CC 9
10
10
10
10
10/–20
–20
–20
–20
–20
Program execution order (in instructions)
sub $2, $1, $3
and $12, $2, $5
or $13, $6, $2
add $14, $2,$2
sw $15, 100($2)
IM
Reg
DM

DATA Dependency/Data Hazard (RISC): how to detect/identify?

ADD R1, R2, R3; R1 is result field

SUB R6, R1, R5; R1 is operand

Instruction\ cycle	**1**	**2**	**3**	**4**	**5**	**6**
ADD R1,R2, R3	IF	ID	EX	MA	WB (R1 updated)	
SUB R6, R1, R5		IF	ID (R1 is read) Old value!	EX	MA	WB
Instruction-3			IF	ID	EX	MA
Instruction-4				IF	ID	EX

Compare the *source registers* of the instruction in the decode stage with the *destination register* of the *uncommitted* instructions.

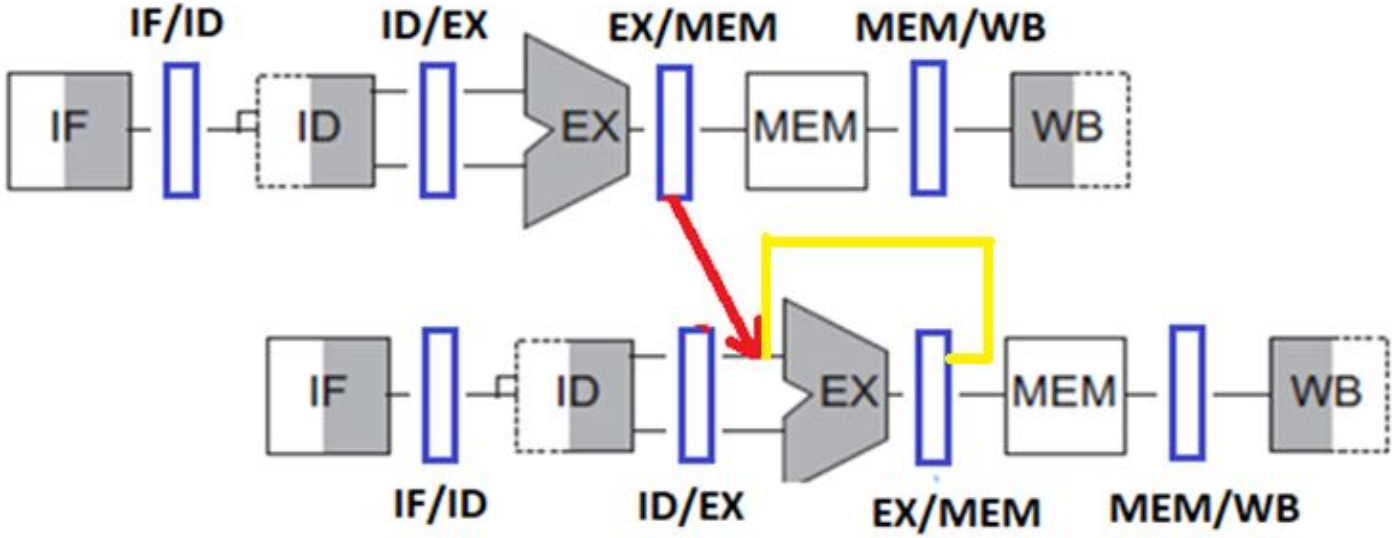
Solution: Interlocks-wait for the result to be available by freezing earlier pipeline stages

Instructions	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
ADD R1,R2, R3	IF	ID	EX	MA	WB						
SUB R6, R1, R5		IF	ID	ID	ID	ID	EX	MA			
Instruction-3			IF	IF	IF	IF	ID	EX			
Instruction-4			Stalled stages				IF	ID			
Instruction-5								IF			

Resources	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
IF	I1	I2	I3	I3	I3	I3	I4	I5	I6		
ID		I1	I2	I2	I2	I2	I3	I4	I5		
EX			I1	NO	NO	NO	I2	I3	I4		
MA				I1	NO	NO	NO	I2	I3		
WB					I1	NO	NO	NO	I2		

Better solution: Forwarding: A new datapath should be created to pass the data from the output of the ALU to its input for the following instruction in next time step.

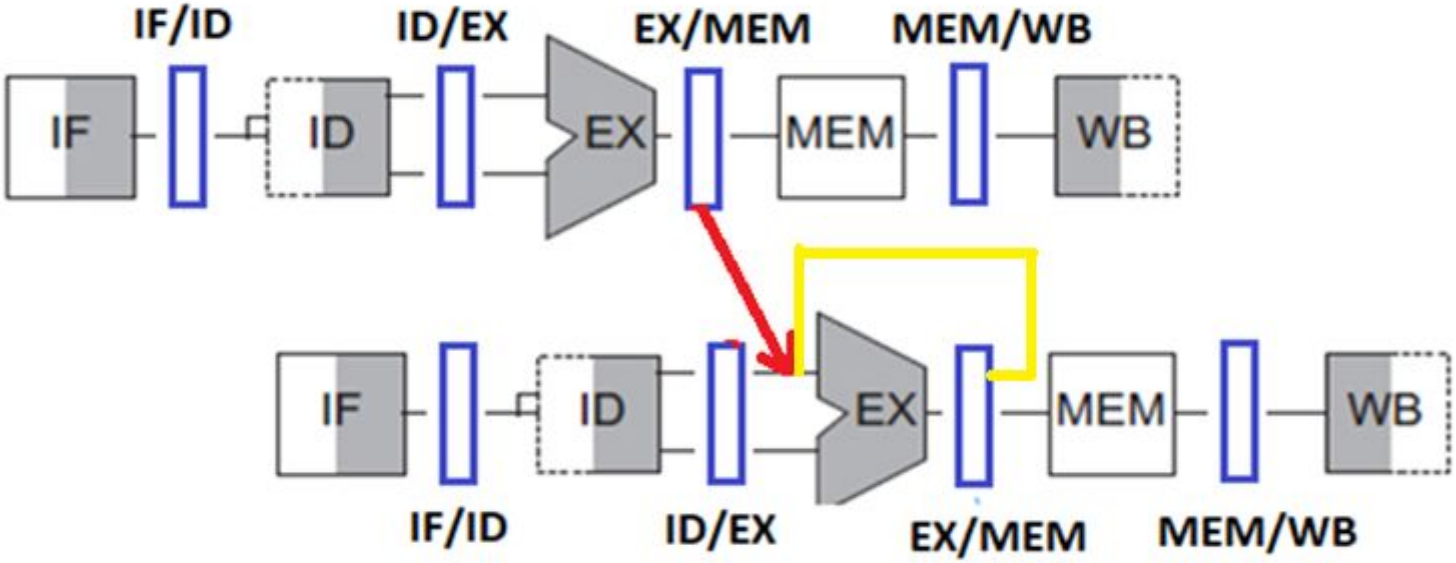
Instruction	1	2	3	4	5	6	7	8	9	10	11
ADD R1,R2, R3	IF	ID	EX	MA	WB						
SUB R6, R1, R5		IF	ID	EX	MA	WB					
Instruction-3			IF	ID	EX	MA					
Instruction-4				IF	ID	EX					
Instruction-5					IF	ID					
Instruction-6						IF					

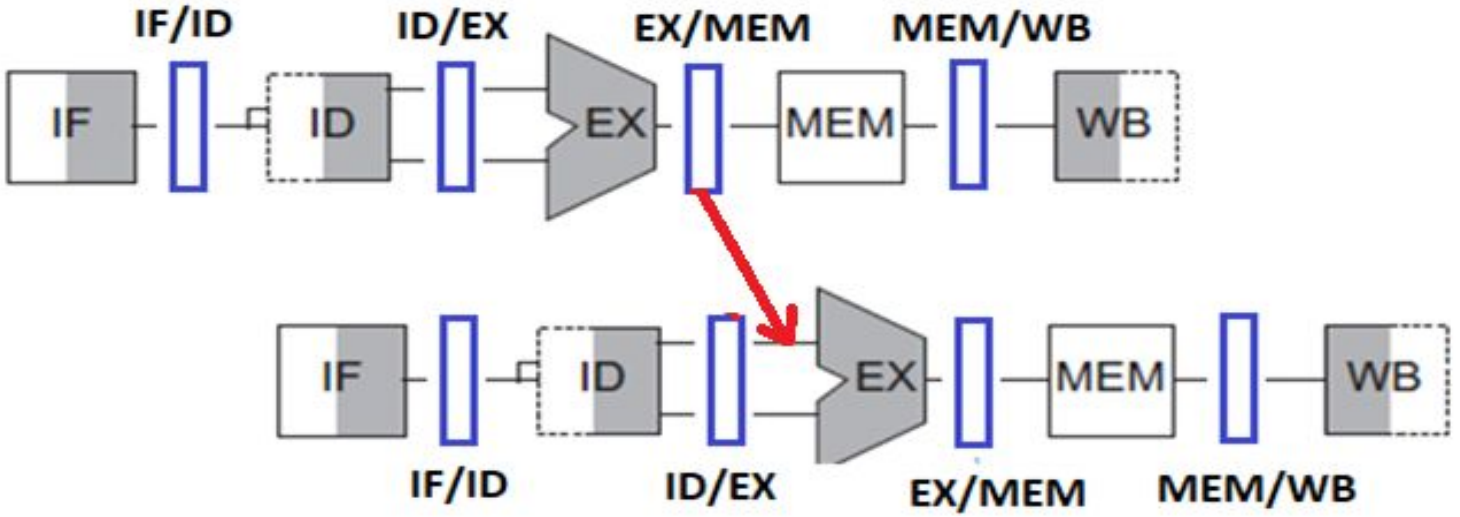


Forwarding: A new datapath should be created to pass the data from the output of the ALU to its input for the following instruction in next time step.

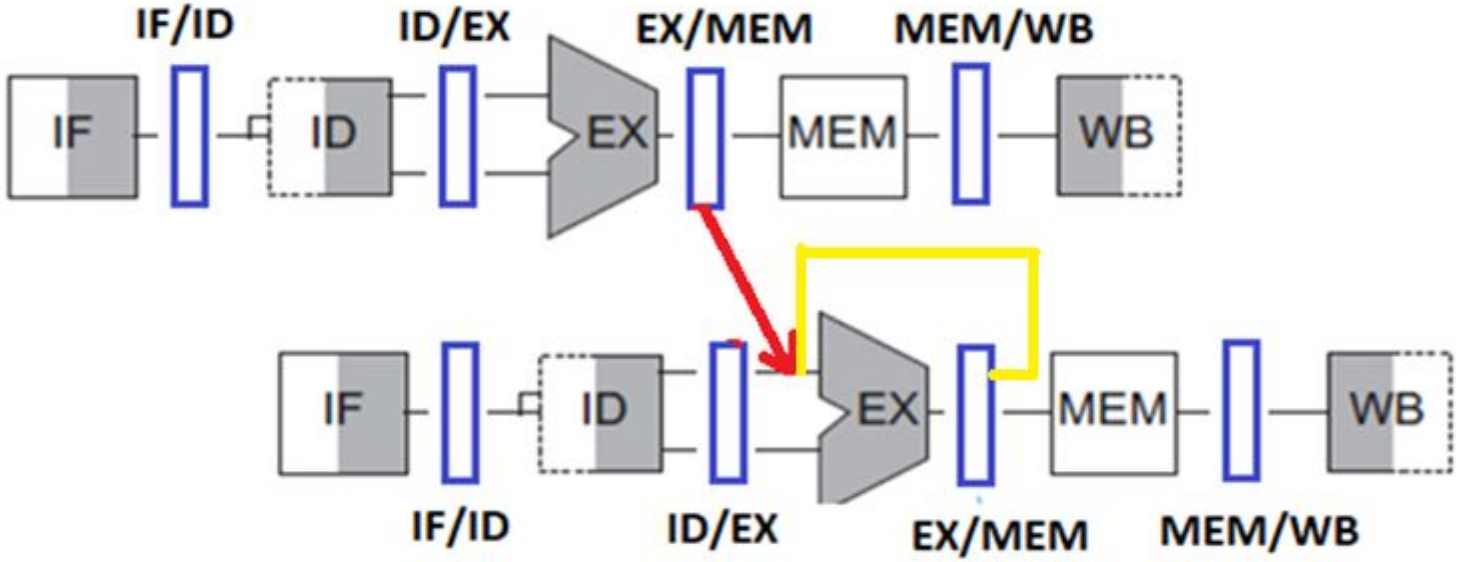
For Instruction: SUB R6, R1, R5; ALU will get one of the inputs (the result of preceding instruction) from pipeline register EX/MEM at clock cycle-4 instead of reading from R1 at clock cycle-3. .

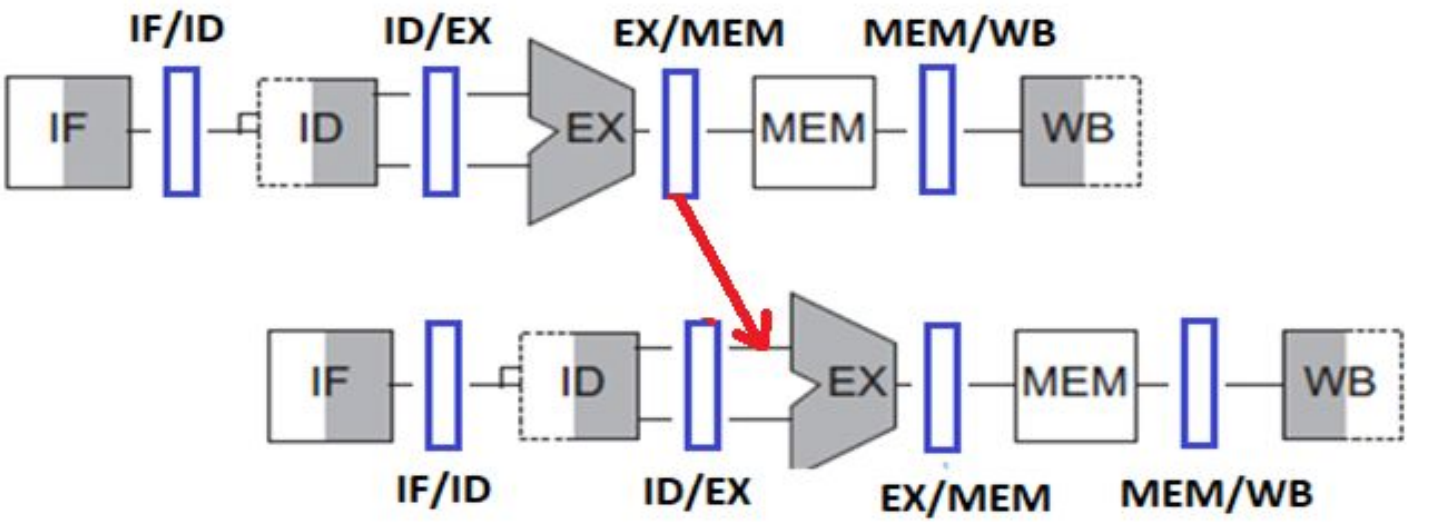
Instruction	1	2	3	4	5	6	7
ADD R1,R2, R3	IF	ID	EX	MA	WB		
SUB R6, R1, R5		IF	ID	EX	MA	WB	
Instruction-3			IF	ID	EX	MA	
Instruction-4				IF	ID	EX	
Instruction-5					IF	ID	
Instruction-6						IF	

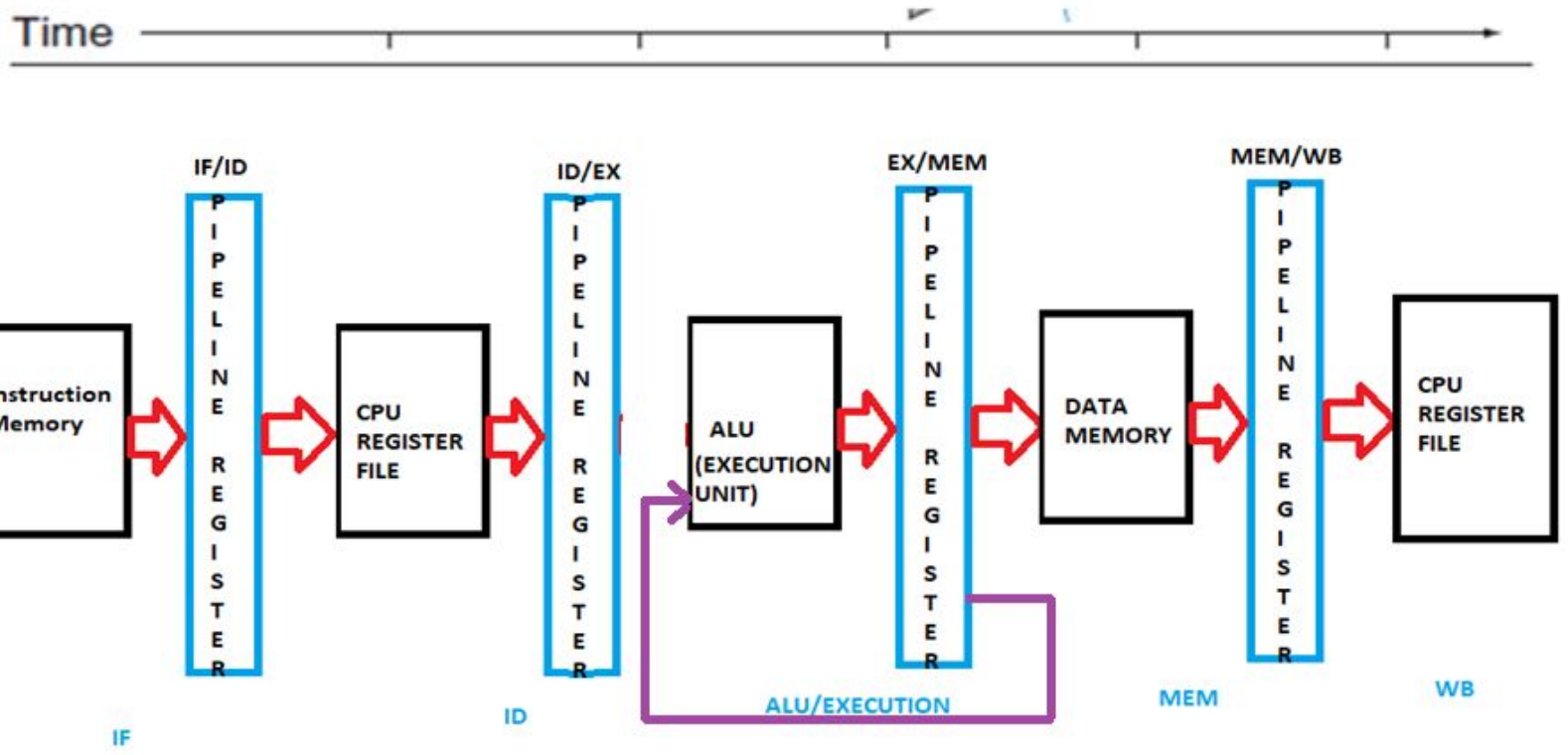




Time	Clock Cycle-1	Clock Cycle-2	Clock Cycle-3	Clock Cycle-4	Clock Cycle-5	Clock Cycle-6	Clock Cycle-7

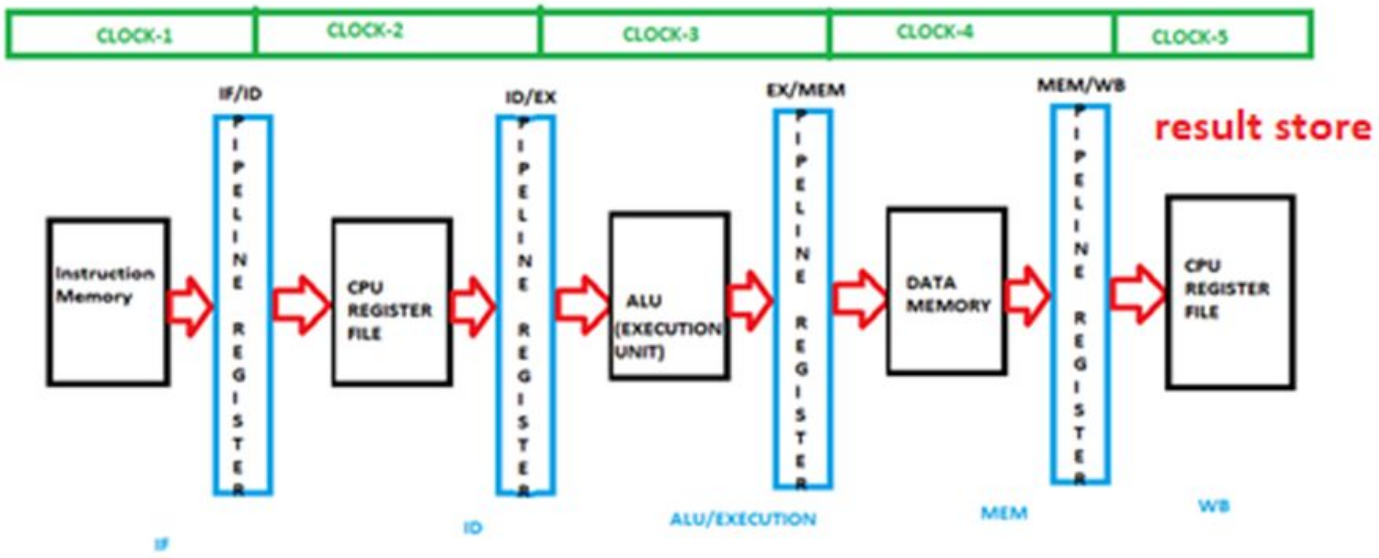



IF/ID
ID/EX
EX/MEM
MEM/WB
IF
ID
EX
MEM
WB
IF
ID
EX
MEM
WB
IF/ID
ID/EX
EX/MEM
MEM/WB

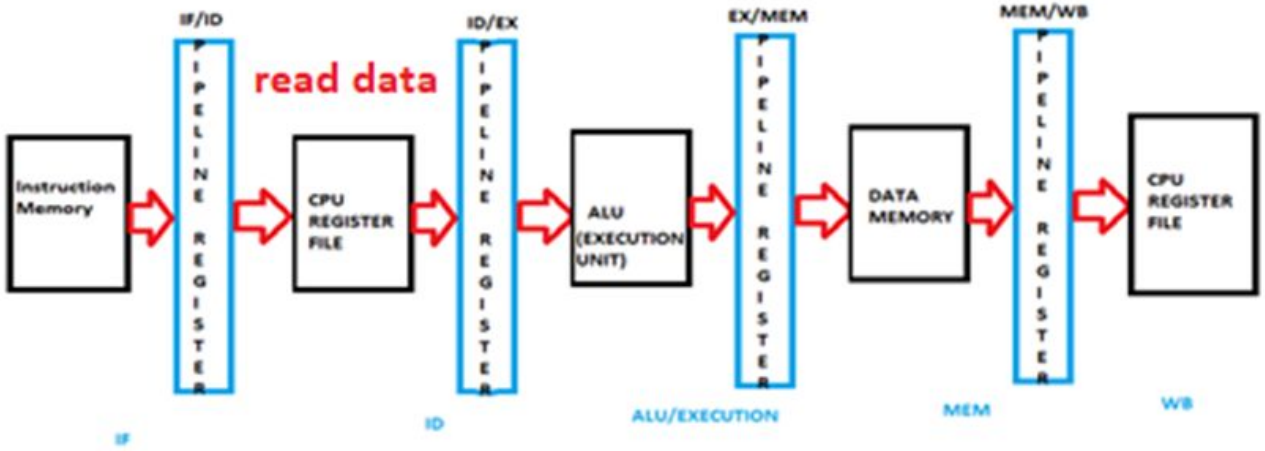

Time
IF/ID
ID/EX
EX/MEM
MEM/WB
Instruction Memory
PIPELINE REGISTER
CPU REGISTER FILE
PIPELINE REGISTER
ALU (EXECUTION UNIT)
PIPELINE REGISTER
DATA MEMORY
PIPELINE REGISTER
CPU REGISTER FILE
ALU/EXECUTION
IF
ID
MEM
WB

forwarding

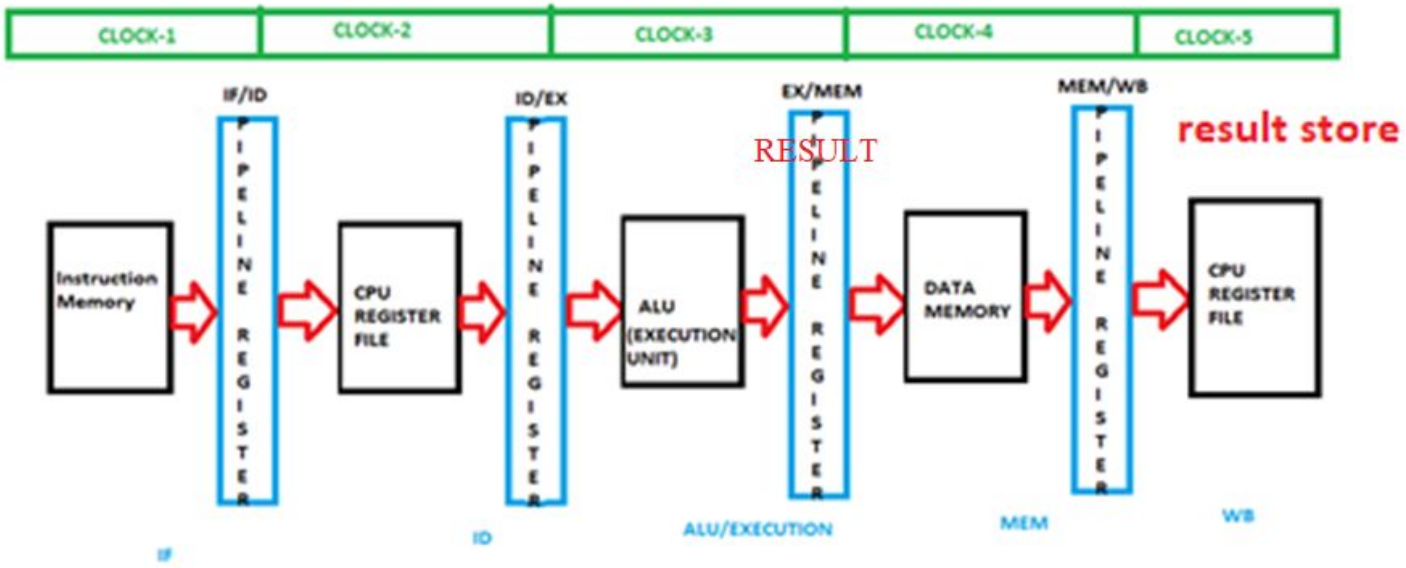
ADD R1,R2, R3



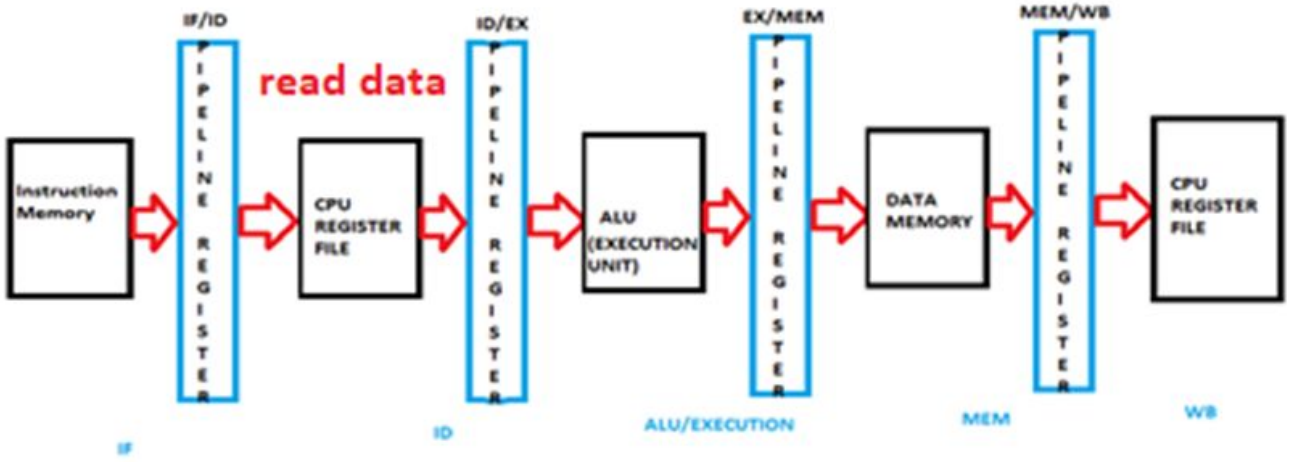
SUB R6, R1, R5



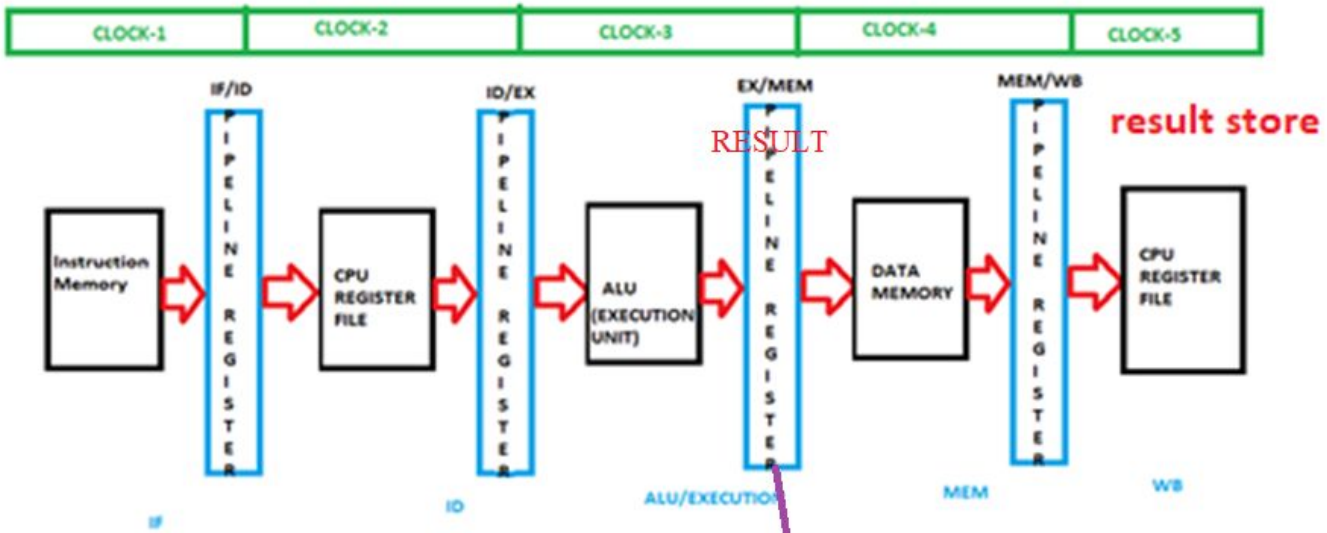
ADD R1,R2, R3



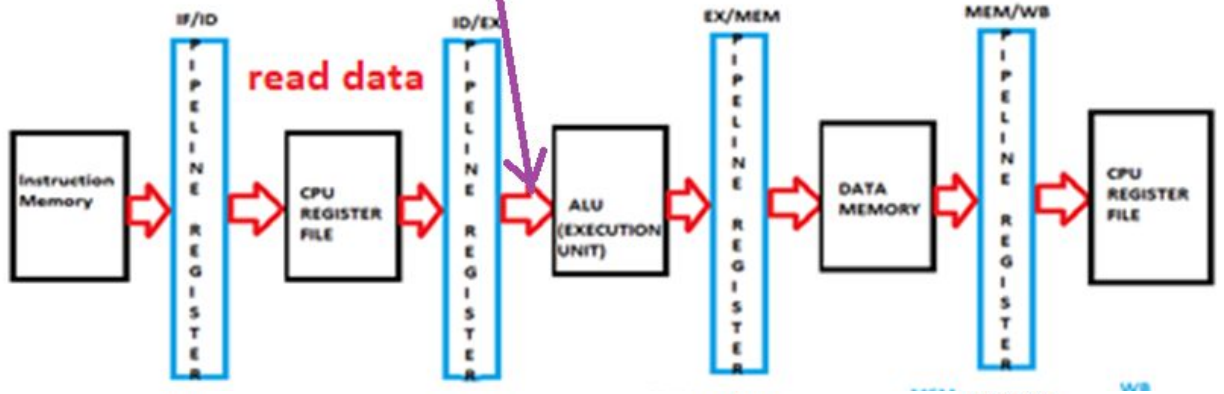
SUB R6, R1, R5

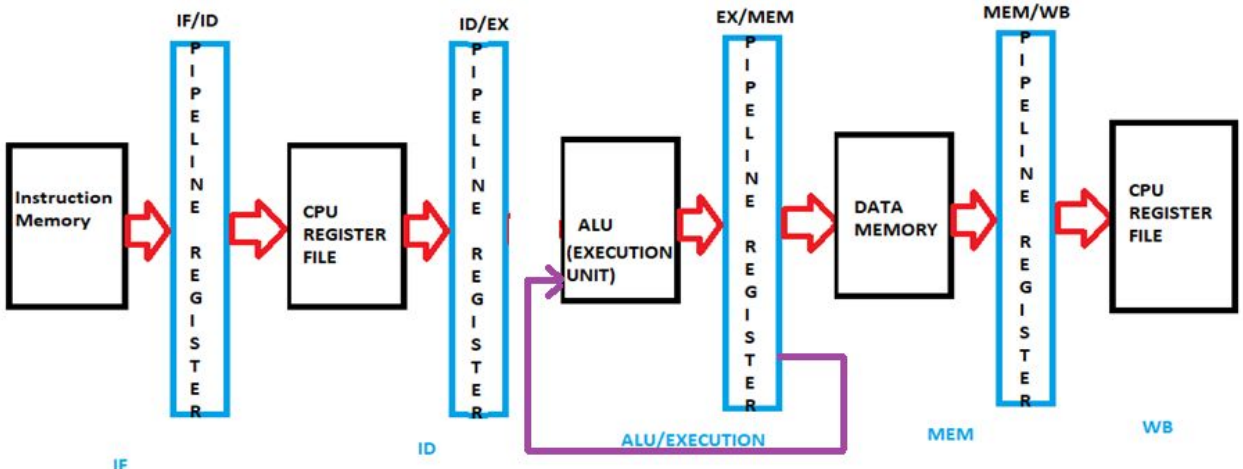


ADD R1,R2, R3



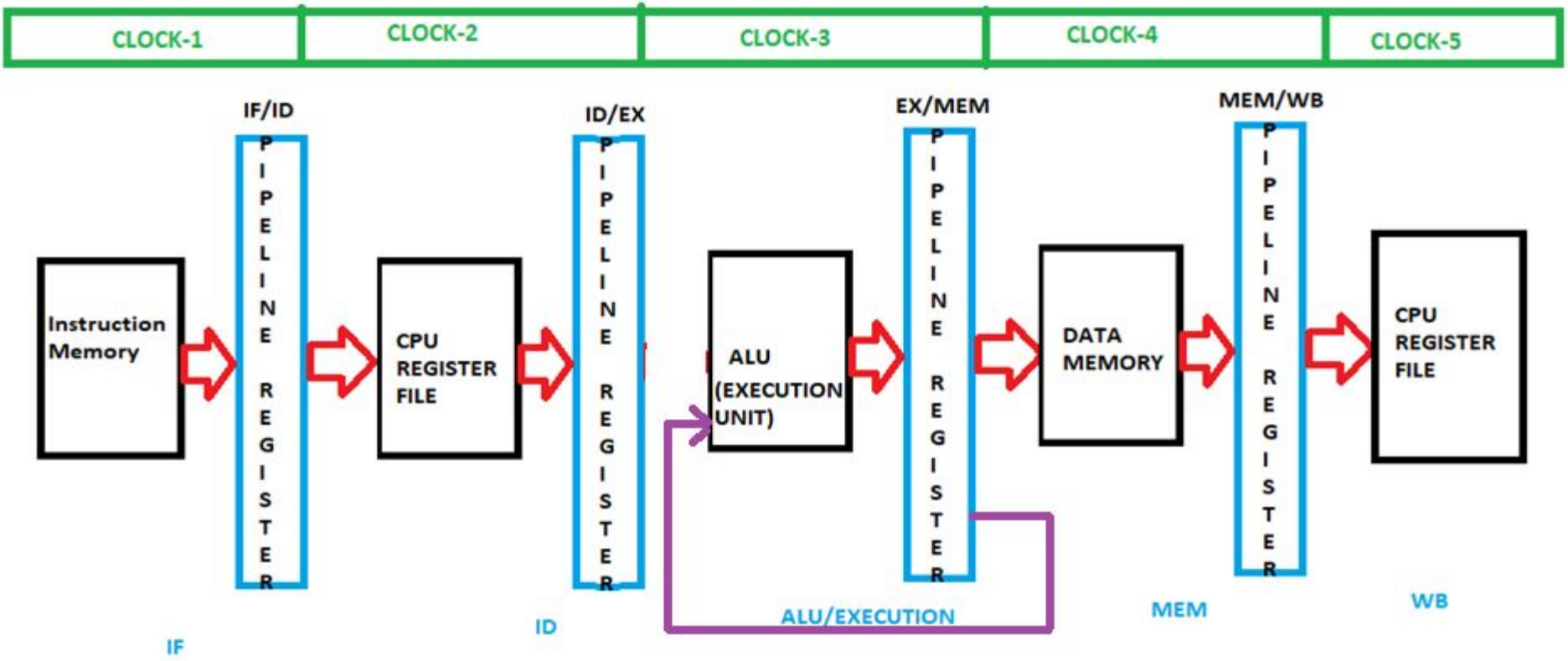
SUB R6, R1, R5

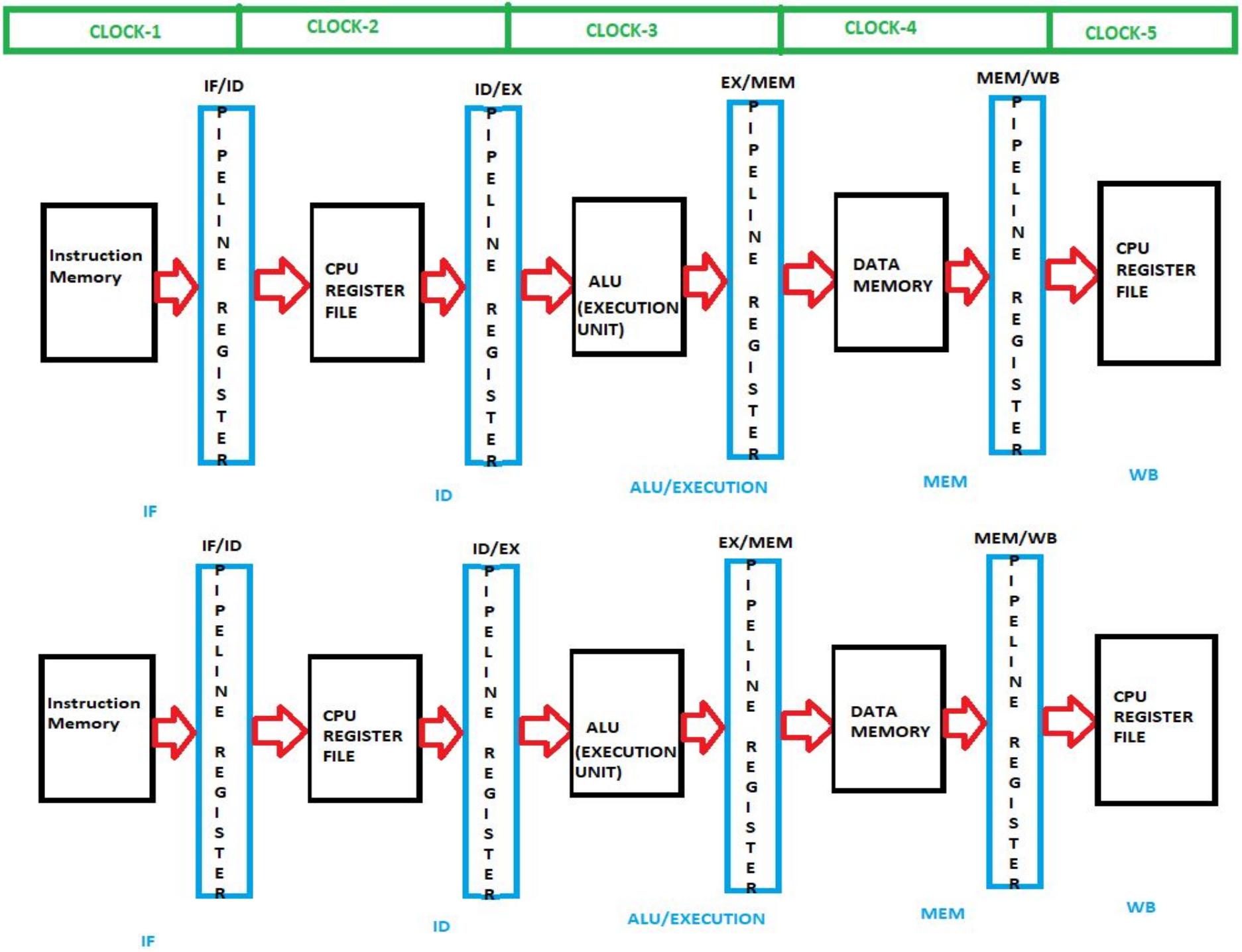




Forwarding: A new datapath should be created to pass the data from the output of the ALU to its input for the following instruction in next time step.
For Instruction: SUB R6, R1, R5; ALU will get one of the inputs (the result of preceding instruction) from pipeline register EX/MEM at clock cycle-4 instead of reading from R1 at clock cycle-3.

Using forwarding, delay/stall can be fully avoided in case of true data dependency as noticed in following pair of instruction.
ADD R1, R2, R3; R1 is result field
SUB R6, R1, R5; R1 is operand




CLOCK-1
CLOCK-2
CLOCK-3
CLOCK-4
CLOCK-5
IF/ID
ID/EX
EX/MEM
MEM/WB
Instruction Memory
PIPELINE REGISTER
CPU REGISTER FILE
PIPELINE REGISTER
ALU (EXECUTION UNIT)
PIPELINE REGISTER
DATA MEMORY
PIPELINE REGISTER
CPU REGISTER FILE
IF
ID
ALU/EXECUTION
MEM
WB
IF/ID
ID/EX
EX/MEM
MEM/WB
Instruction Memory
PIPELINE REGISTER
CPU REGISTER FILE
PIPELINE REGISTER
ALU (EXECUTION UNIT)
PIPELINE REGISTER
DATA MEMORY
PIPELINE REGISTER
CPU REGISTER FILE
IF
ID
ALU/EXECUTION
MEM
WB

Data dependency: LOAD instruction followed by Arithmetic Instruction

Example:

LOAD R1, R2(10)

ADD R2, R1, R3

In the first instruction, a data is read from RAM and loaded into a Register then the same data is used as one of the operands in the following ALU instruction that appears next to LOAD instruction.

When these instructions are processed through a pipelined processor

Inst	**1**	**2**	**3**	**4**	**5**	**6**	**7**
LOAD R1, R2(10)	IF	ID	ALU	MA	WB(data stored to R1)		
ADD R2, R1, R3		IF	ID (R1 is read)	ALU	MA		
Inst-3			IF	ID	ALU		
Inst-4				IF	ID		
Inst-5					IF		

Data dependency: LOAD instruction followed by Arithmetic Instruction

Example:

LOAD R1, R2(10)

ADD R1, R2, R3

Load instruction reads data from RAM and stores to R1 at clock cycle-5 but ADD instruction reads data from R1 at clock cycle-3.

Consequence: Old data is read

Inst	**1**	**2**	**3**	**4**	**5**	**6**	**7**
LOAD R1, R2(10)	IF	ID	ALU	MA	WB(data stored to R1)		
ADD R1, R2, R3		IF	ID (R1 is read)	ALU	MA		
Inst-3			IF	ID	ALU		
Inst-4				IF	ID		
Inst-5					IF		

Solution: Data dependency: Interlocking

Example:

LOAD R1, R2(10)

ADD R1, R2, R3

Register fetch for ADD instruction and following tasks should be delayed by at least three/four clock cycles

Inst	**1**	**2**	**3**	**4**	**5**	**6**	**7**
LOAD R1, R2(10)	IF	ID	ALU	MA	WB		
ADD R1, R2, R3		IF				ID	ALU
Inst-3						IF	ID
Inst-4							IF
Inst-5							

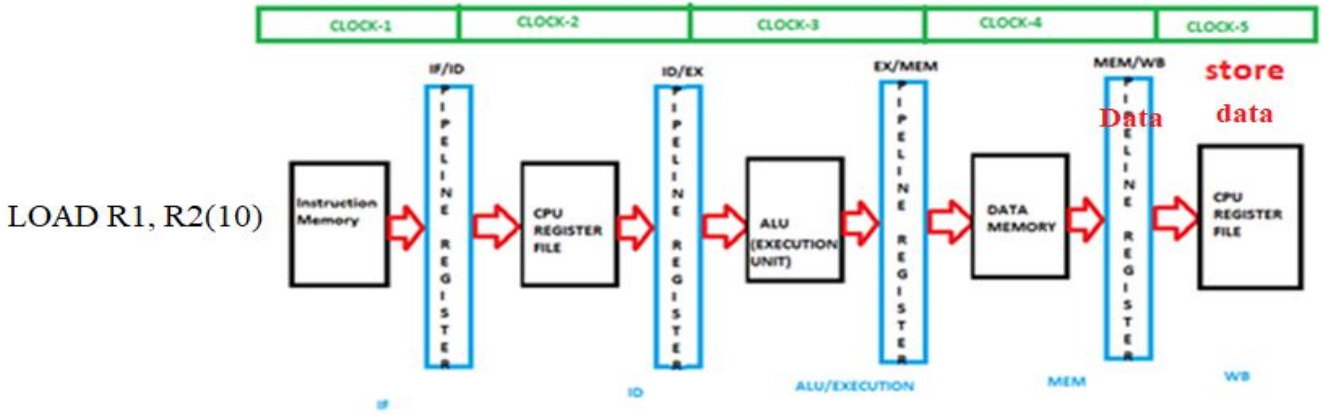
Solution: Data dependency: Forwarding

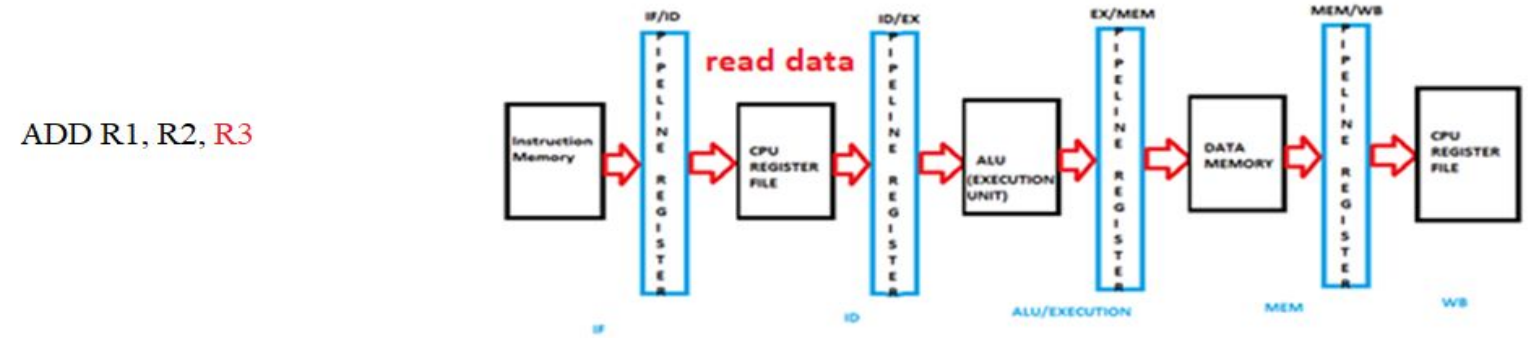
Example:

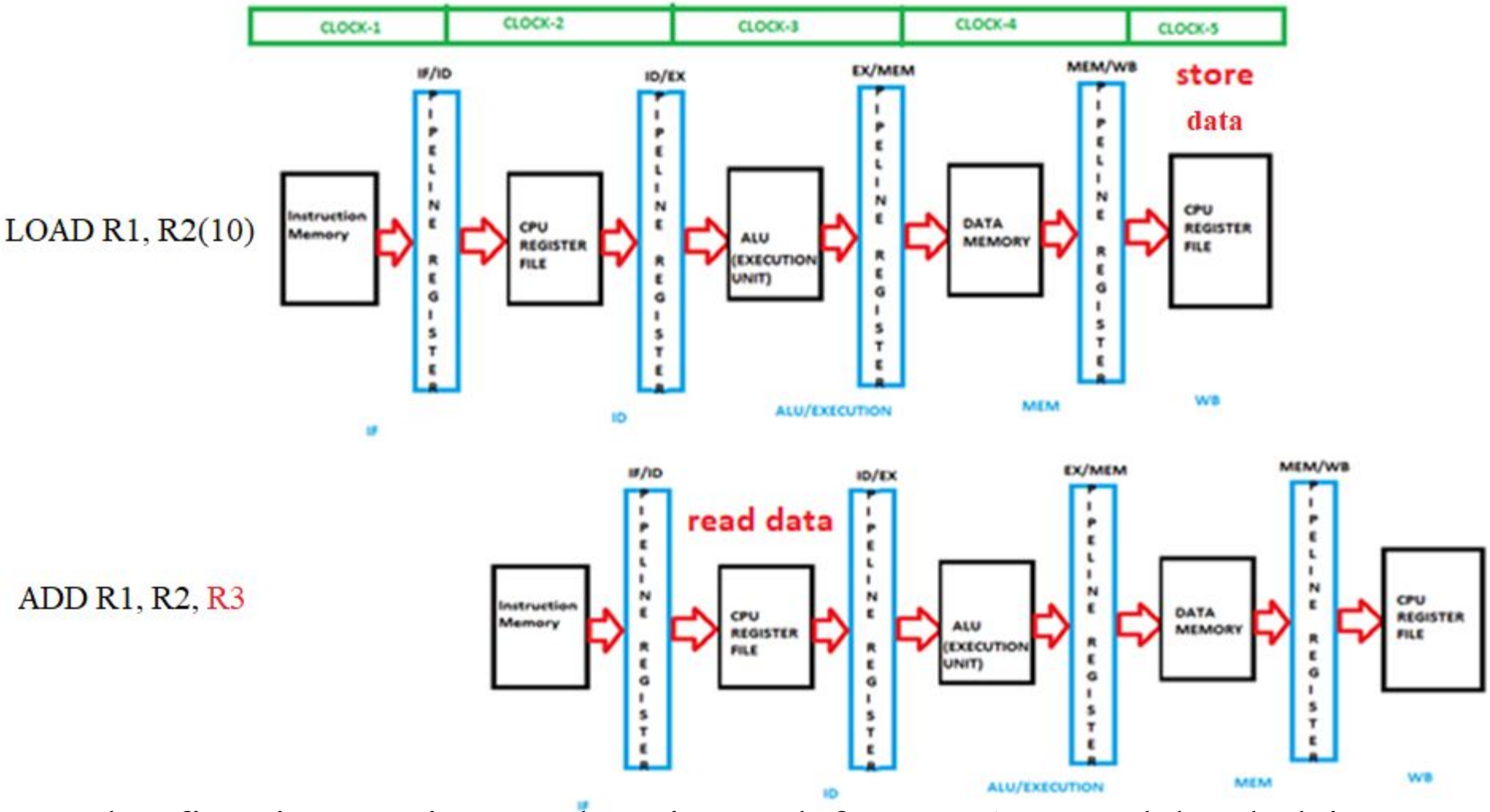
LOAD R1, R2(10)

ADD R2, R1, R3

Inst	**1**	**2**	**3**	**4**	**5**
LOAD R1, R2(10)	IF	ID	ALU	MA	WB(data stored to R1)
ADD R2, R1, R3		IF	ID (R1 is read)	ALU	MA
Inst-3			IF	ID	ALU
Inst-4				IF	ID
Inst-5					IF



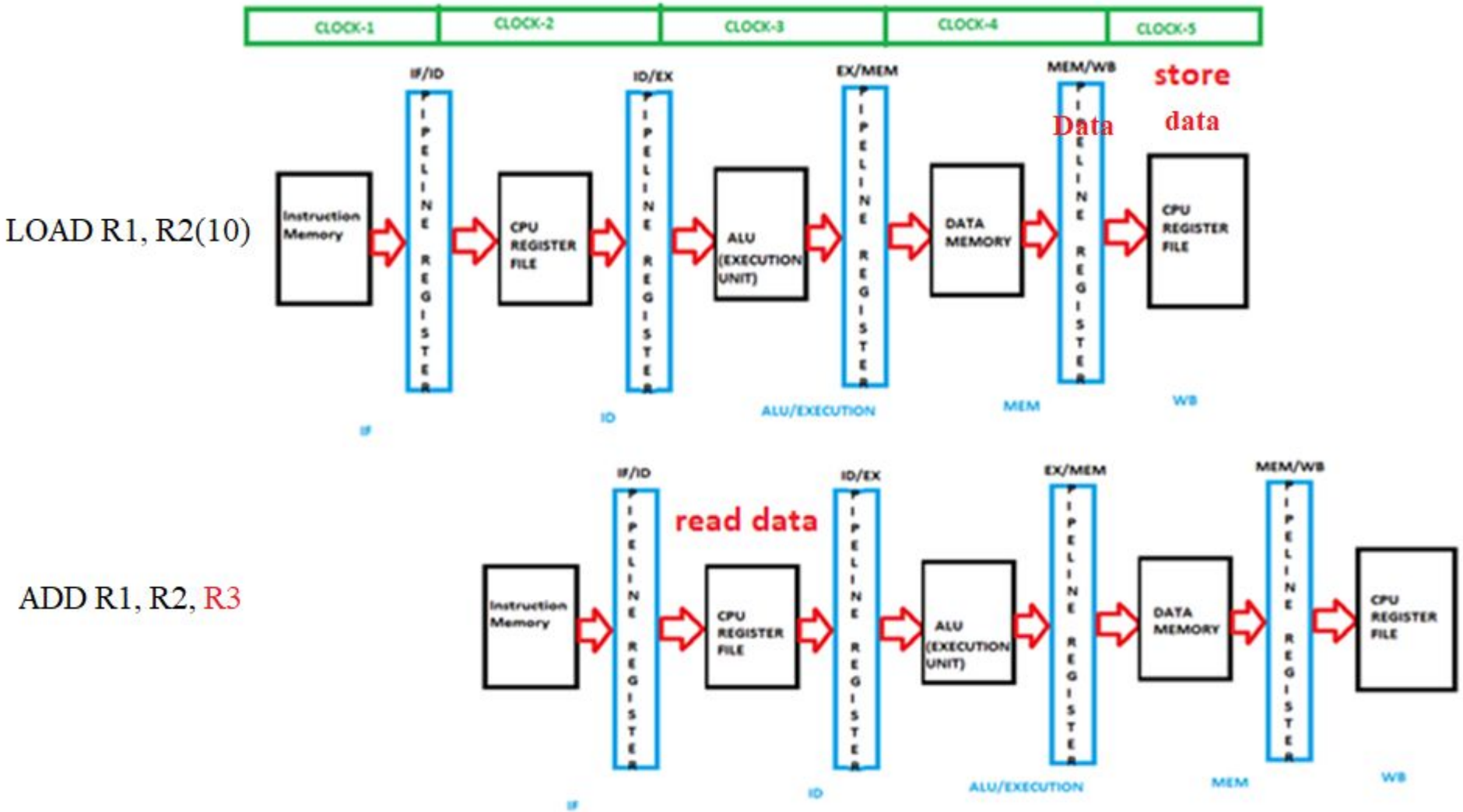




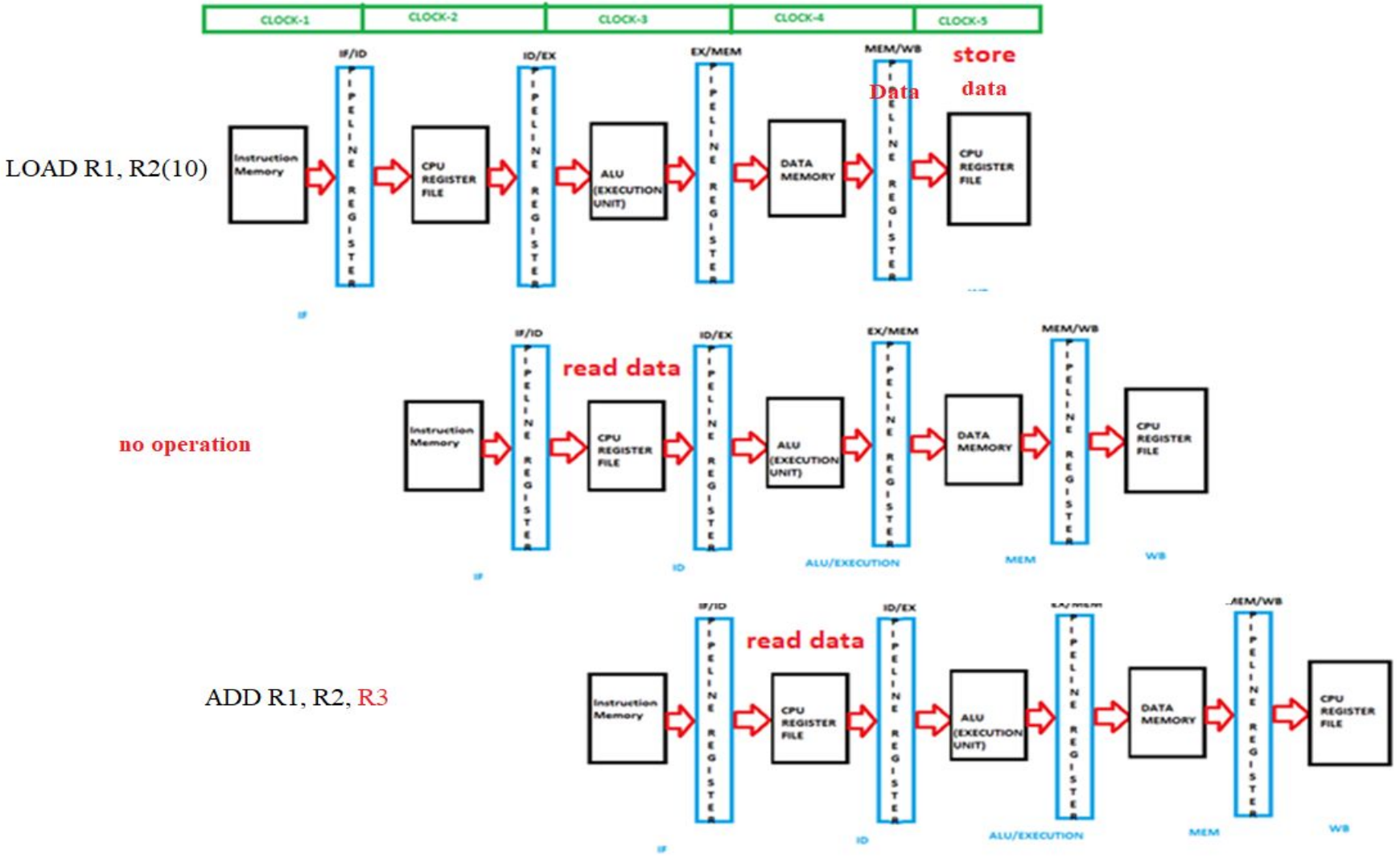
In the first instruction, a data is read from RAM and loaded into a Register then the same data is used as one of the operands in an ALU instruction.

LOAD R1, R2(10)

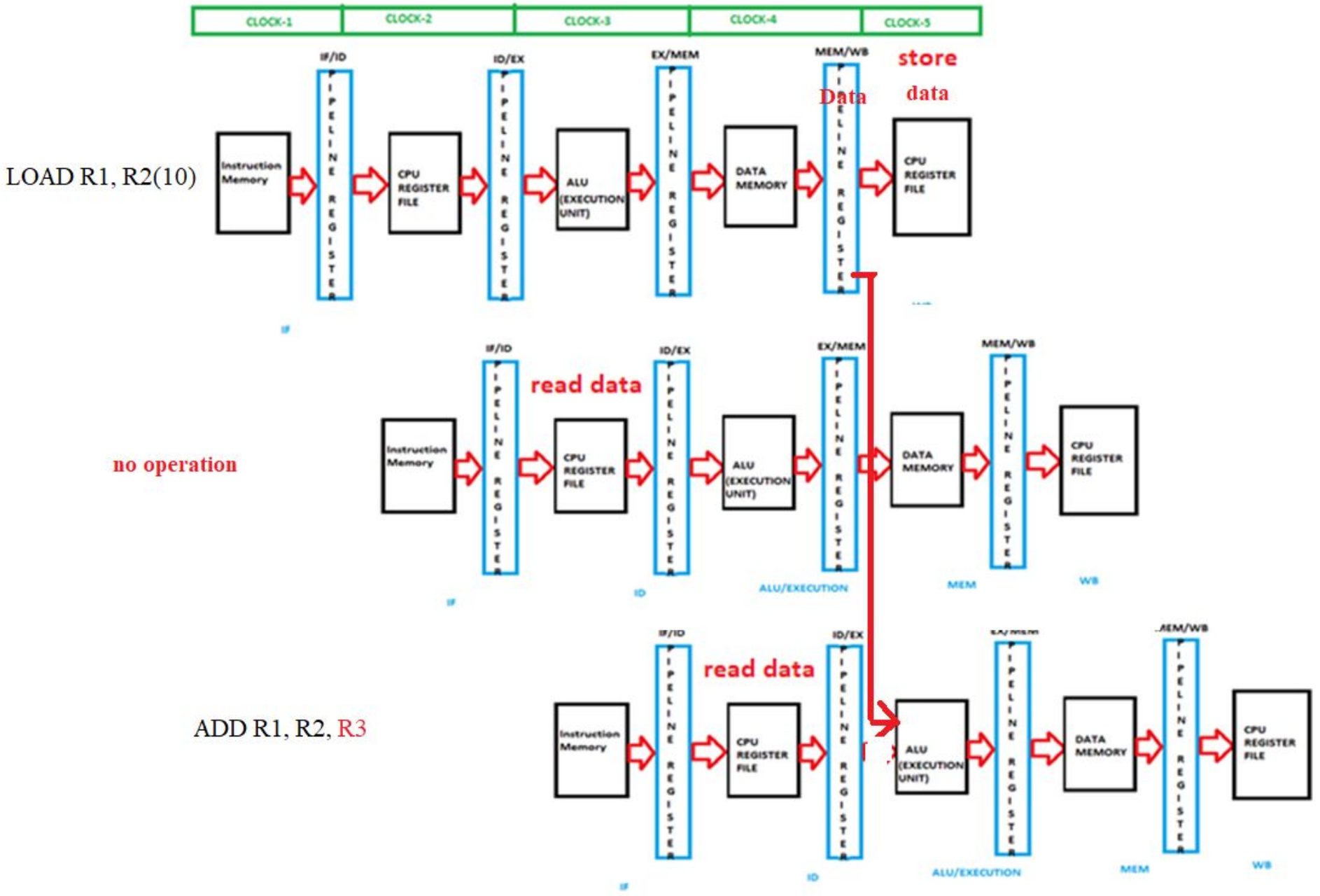
ADD R1, R2, R3



Memory data is available at MEM/WB register at clock cycle-4. The earliest possible option to forward this data to following instruction is also clock cycle-4 or later. The ALU operation of the ADD instruction can also be performed at clock cycle-5 at the earliest if data read from memory is passed to ALU from MEM/WB register at clock cycle-4. So the pipeline must be delayed by 1 clock cycle despite forwarding path from MEM/WB to input to ALU.



Memory data is available at MEM/WB register at clock cycle-4. The earliest possible option to forward this data to following instruction is also clock cycle-4 or later. The ALU operation of the ADD instruction can also be performed at clock cycle-5 at the earliest if data read from memory is passed to ALU from MEM/WB register at clock cycle-4. So the pipeline must be delayed by 1 clock cycle despite forwarding path from MEM/WB to input to ALU.



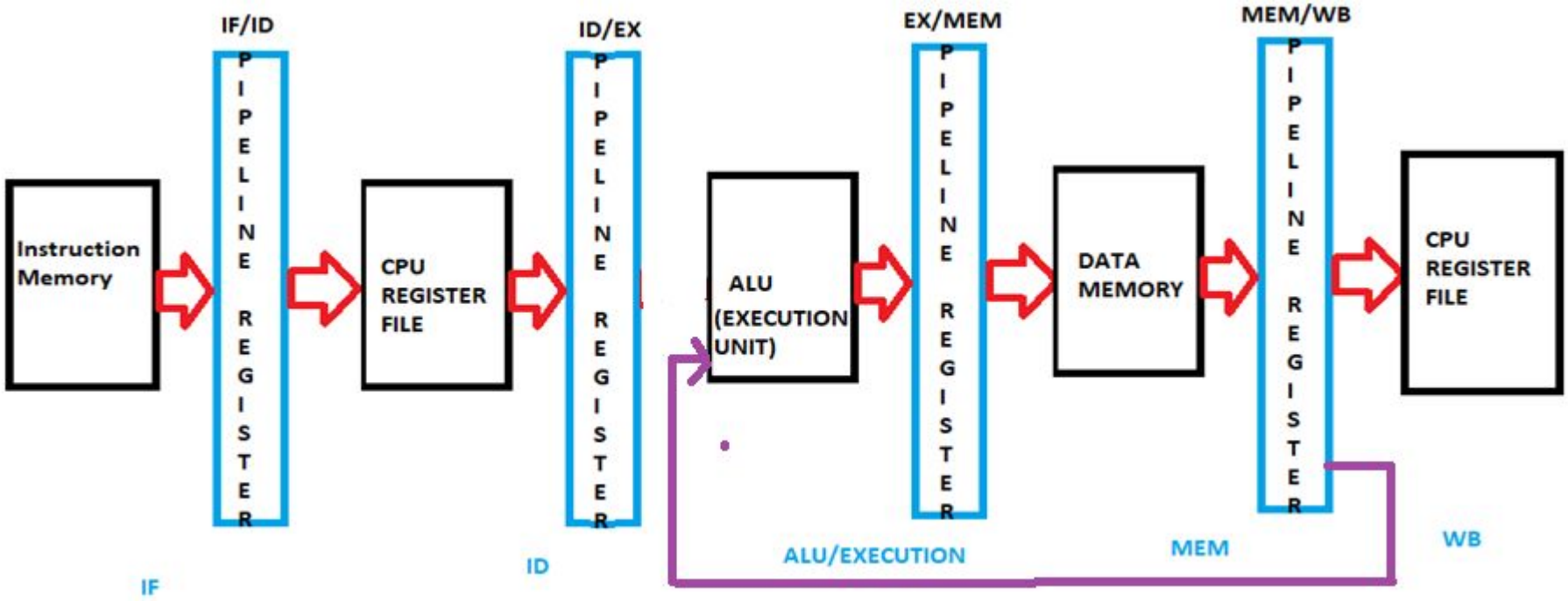
One clock delay and forwarding (MEM/WB to ALU) is required.

Forwarding in case of Load followed by ALU instruction

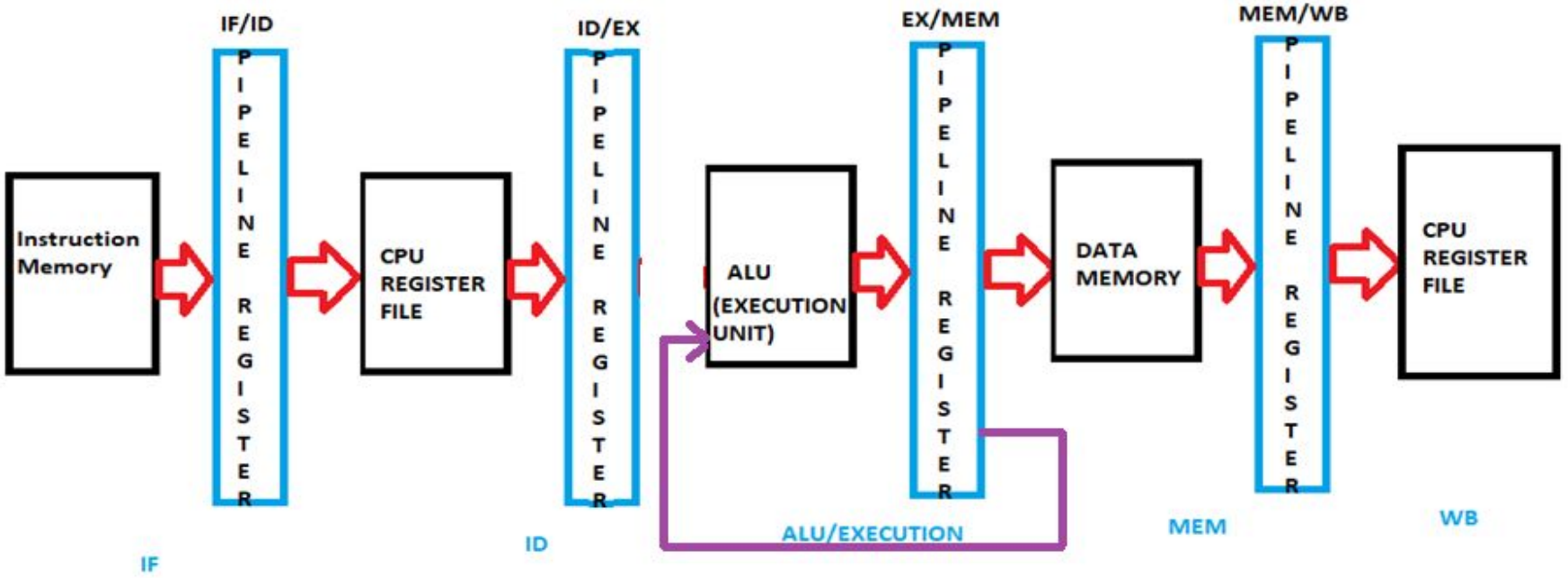
In the first instruction, a data is read from RAM and loaded into a Register then the same data is used as one of the operands in an ALU instruction. One clock delay and forwarding is required.

LOAD R1, R2(10)

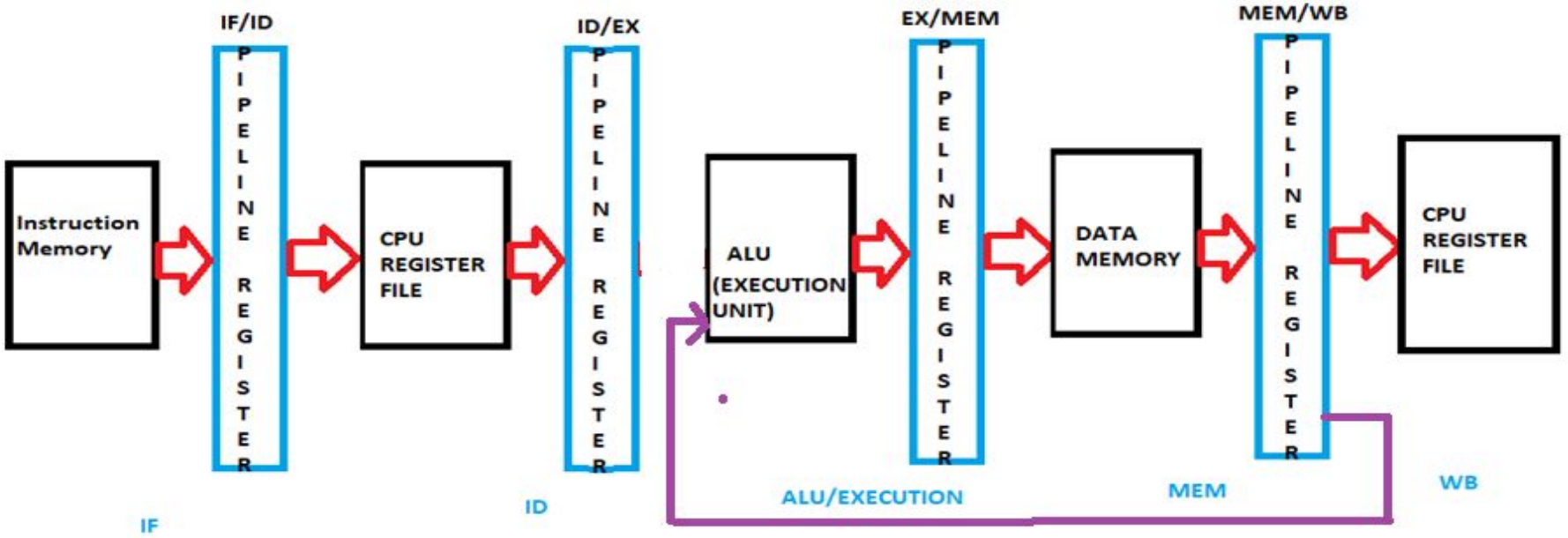
ADD R1, R2, R3

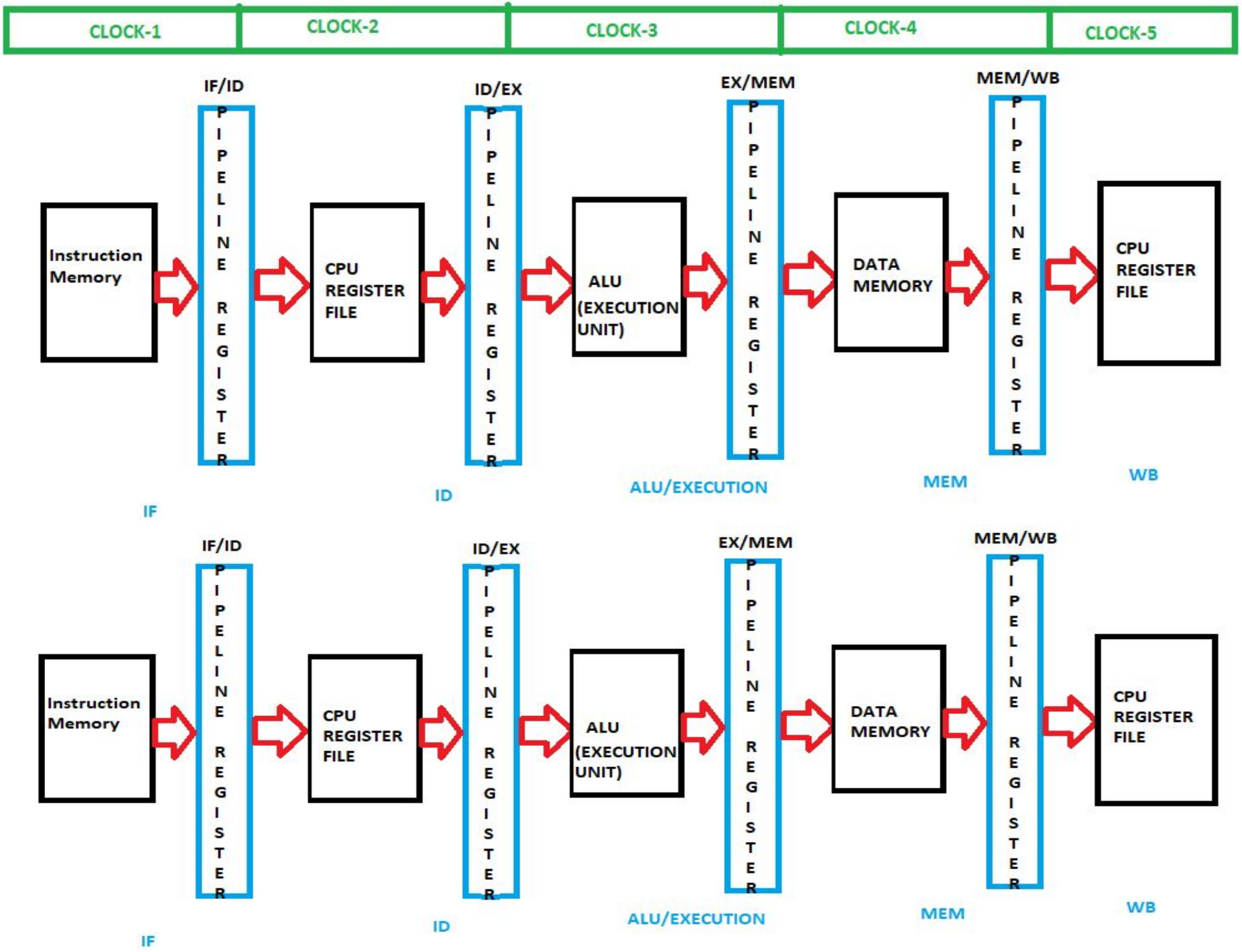


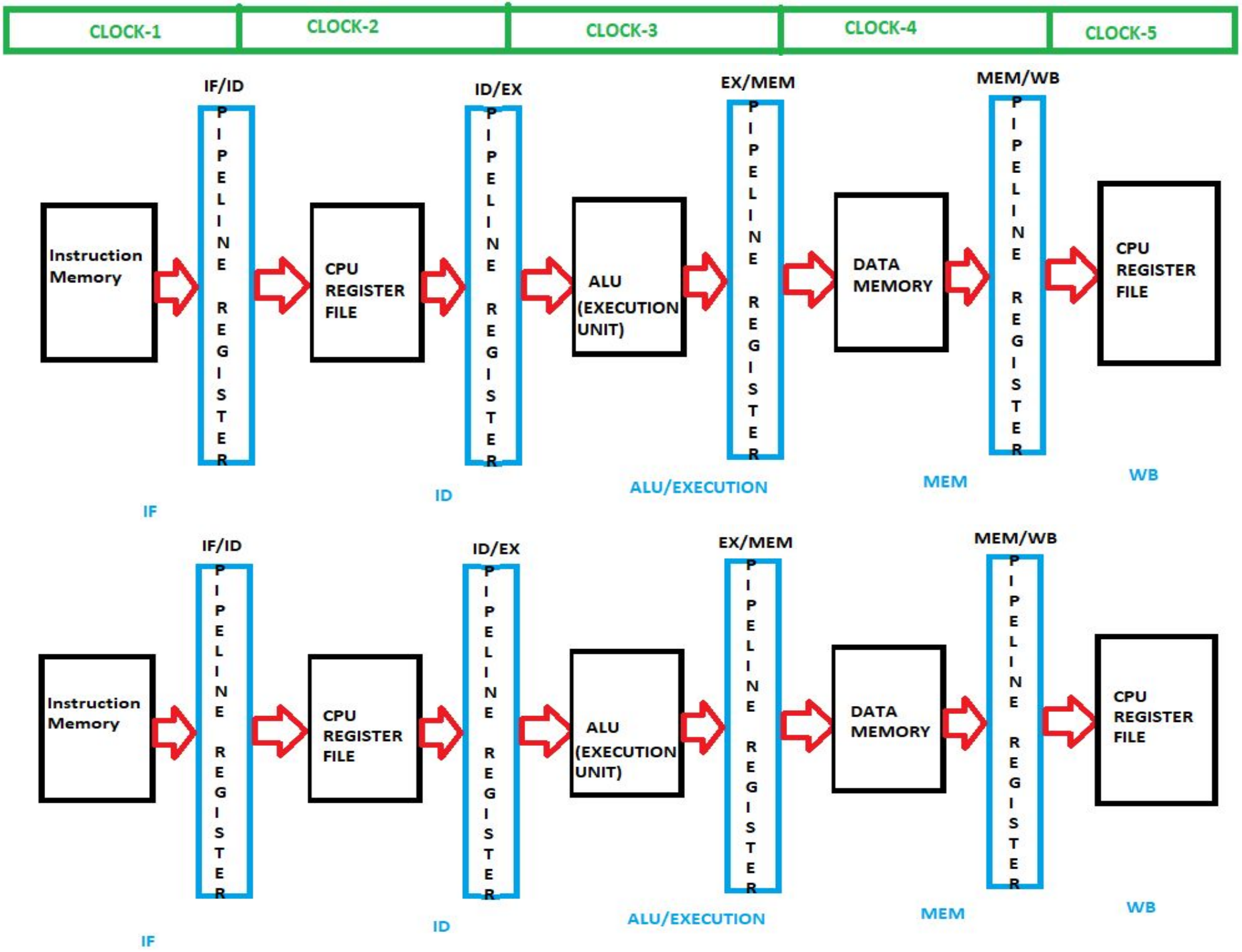
Comparison: forwarding: ALU instruction followed by ALU instruction



Comparison: forwarding: LOAD instruction followed by ALU instruction




CLOCK-1
CLOCK-2
CLOCK-3
CLOCK-4
CLOCK-5
IF/ID
ID/EX
EX/MEM
MEM/WB
Instruction Memory
PIPELINE REGISTER
CPU REGISTER FILE
PIPELINE REGISTER
ALU (EXECUTION UNIT)
PIPELINE REGISTER
DATA MEMORY
PIPELINE REGISTER
CPU REGISTER FILE
IF
ID
ALU/EXECUTION
MEM
WB
IF/ID
ID/EX
EX/MEM
MEM/WB
Instruction Memory
PIPELINE REGISTER
CPU REGISTER FILE
PIPELINE REGISTER
ALU (EXECUTION UNIT)
PIPELINE REGISTER
DATA MEMORY
PIPELINE REGISTER
CPU REGISTER FILE
IF
ID
ALU/EXECUTION
MEM
WB


CLOCK-1
CLOCK-2
CLOCK-3
CLOCK-4
CLOCK-5
IF/ID
ID/EX
EX/MEM
MEM/WB
Instruction Memory
PIPELINE REGISTER
CPU REGISTER FILE
PIPELINE REGISTER
ALU (EXECUTION UNIT)
PIPELINE REGISTER
DATA MEMORY
PIPELINE REGISTER
CPU REGISTER FILE
IF
ID
ALU/EXECUTION
MEM
WB
IF/ID
ID/EX
EX/MEM
MEM/WB
Instruction Memory
PIPELINE REGISTER
CPU REGISTER FILE
PIPELINE REGISTER
ALU (EXECUTION UNIT)
PIPELINE REGISTER
DATA MEMORY
PIPELINE REGISTER
CPU REGISTER FILE
IF
ID
ALU/EXECUTION
MEM
WB

To overcome data dependencies

- **The compiler must fill the delay slots**
- **Ideally, with useful instructions, but nops will work too.**

```
add R0, R10, R11
sub R12, R0, R13
add R13, R0, R14
and R17, R15, R14
```

```
add R0, R10, R11
and R17, R15, R14
nop
sub R12, R0, R13
add R13, R0, R14
```

CONTROL HAZARDS

- A control hazard, also known as a *branch hazard,* occurs when the pipeline makes the wrong decision on a branch prediction and therefore brings instructions into the pipeline that must subsequently be discarded.
- Until the instruction is actually executed, it is impossible to determine whether the branch will be taken or not.

Control Hazard

Addr	Instruction
40	add R30, R30, R30
44	beq R1, R3, 24
48	and R12, R2, R5
52	or R13, R6, R2
56	add R14, R2, R2
60	...
72	lw R4, 50(R7)
76	...

Control Hazard

- Let's look at the following code:
  - If branch is taken: 40, 44, 72, 76, …
  - If branch is not taken: 40, 44, 48, 52, 56, 60, …

```
Addr    Instruction
40      add  R30, R30, R30
44      beq  R1,  R3,  24
48      and  R12, R2,  R5
52      or   R13, R6,  R2
56      add  R14, R2,  R2
60      ...

72      lw   R4,  50(R7)
76      ...
```

Jump here if R1 == R3

Q: In which stage do we determine if the branch is taken?

- EX
- MEM
- WB

A: MEM

The comparison is done in EX, but we don't use the result until the next stage, MEM.

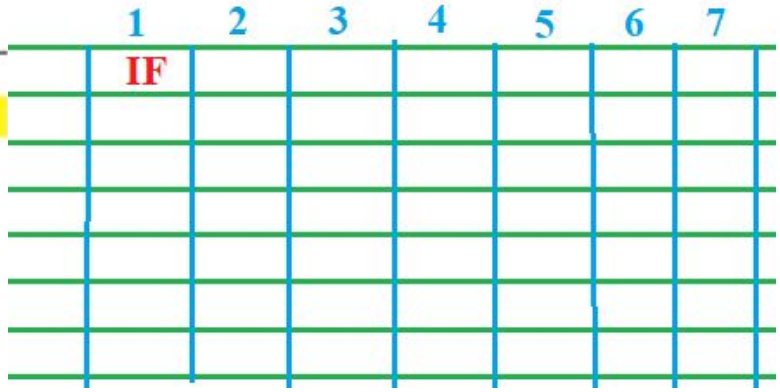
- Let's look at the following code:
  - If branch is taken: 40, 44, 72, 76, …
  - If branch is not taken: 40, 44, 48, 52, 56, 60, …

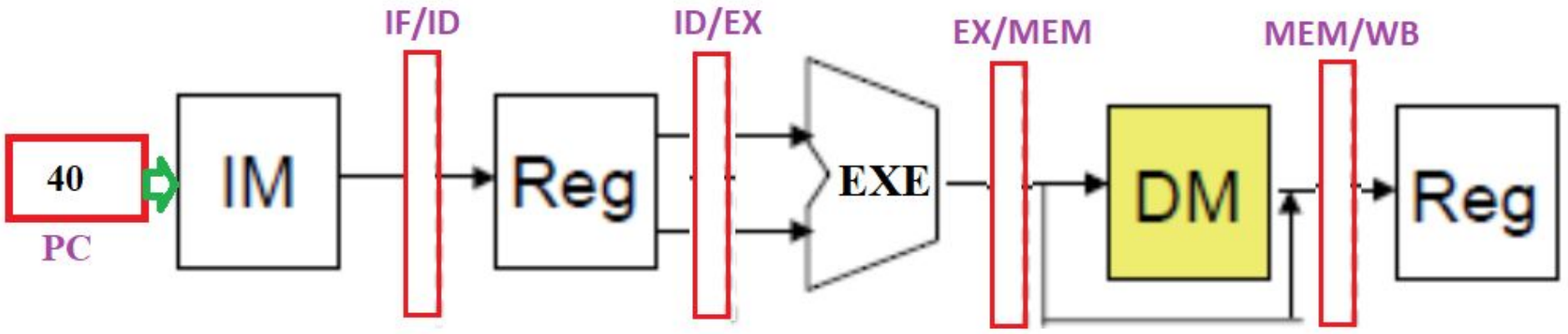

```
Addr    Instruction
40      add  R30, R30, R30
44      beq  R1,  R3,  24
48      and  R12, R2,  R5
52      or   R13, R6,  R2
56      add  R14, R2,  R2
60      ...

72      lw   R4,  50(R7)
76      ...
```

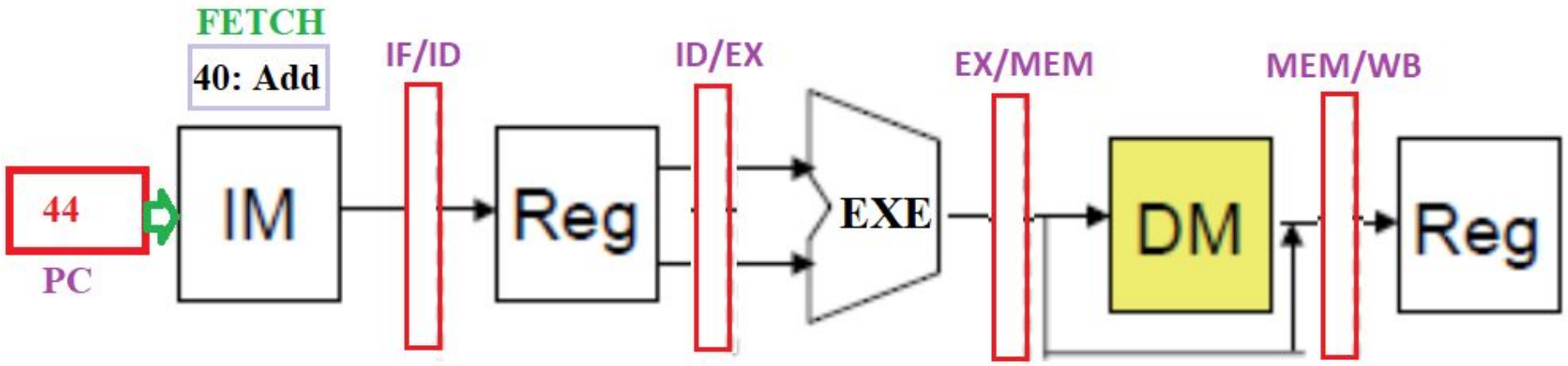
Execute all of these if R1 != R3

Jump here if R1 == R3

Addr	Instruction
40	add R30, R30, R30
44	beq R1, R3, 24
48	and R12, R2, R5
52	or R13, R6, R2
56	add R14, R2, R2
60	...
72	lw R4, 50(R7)
76	...

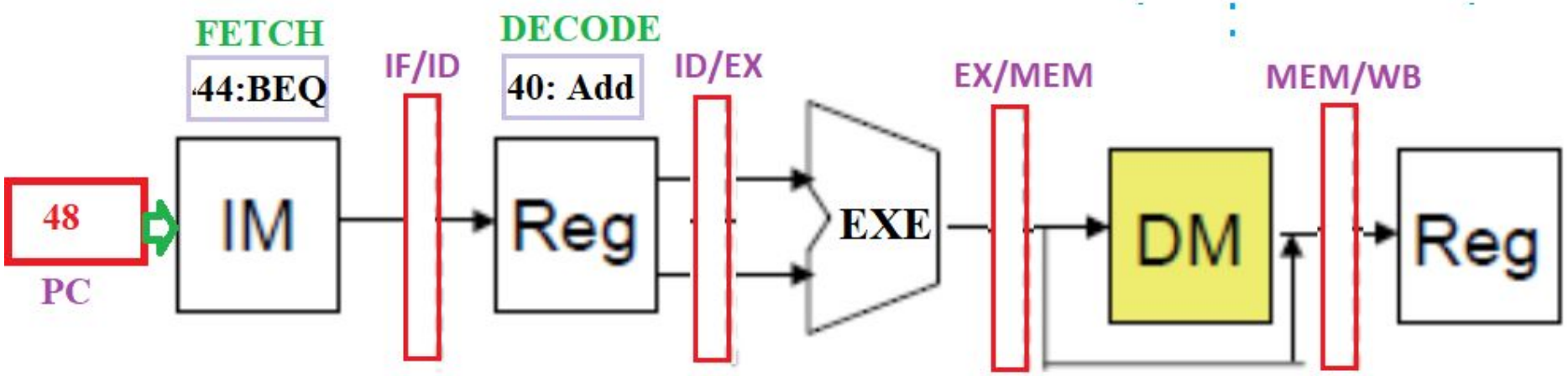






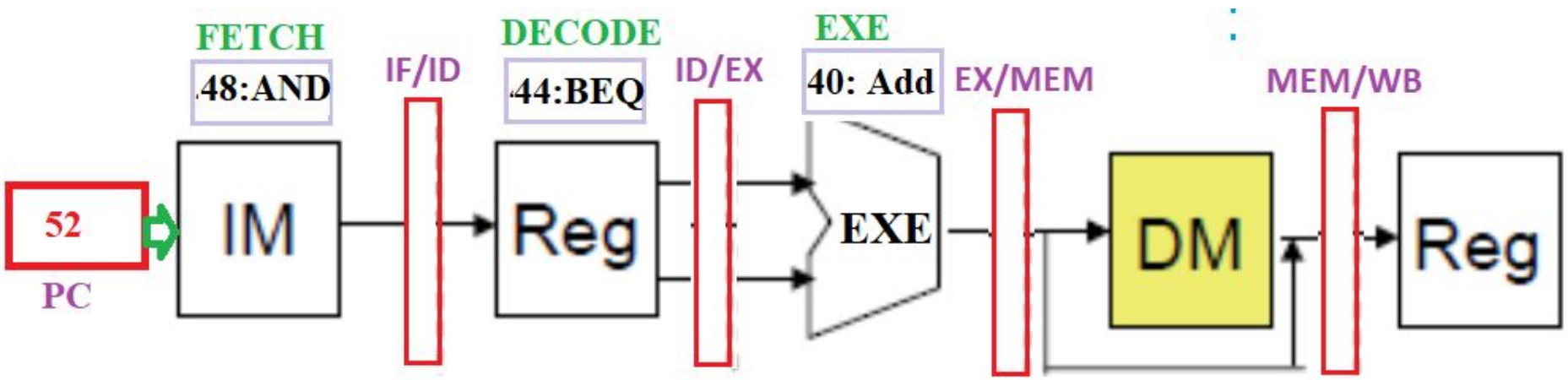
Addr	Instruction
40	add R30, R30, R30
44	beq R1, R3, 24
48	and R12, R2, R5
52	or R13, R6, R2
56	add R14, R2, R2
60	...
72	lw R4, 50(R7)
76	...

	1	2	3	4	5	6	7
	IF	ID					
		IF					

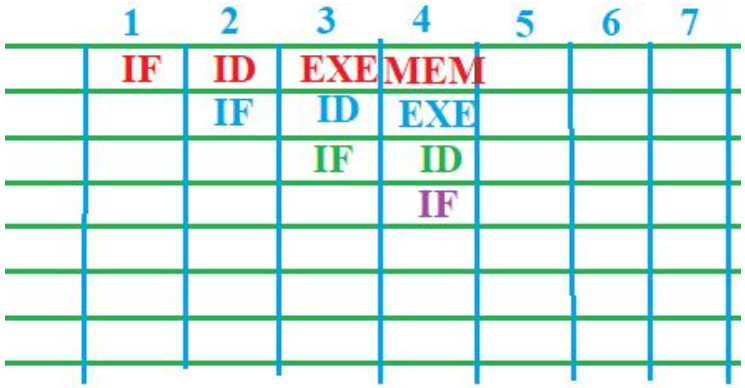


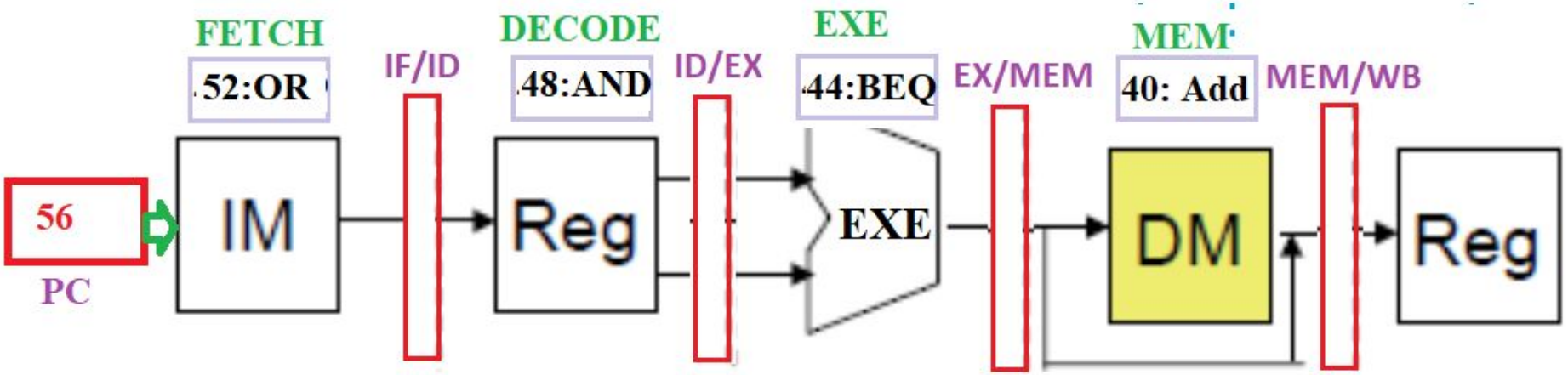
```
Addr        Instruction
40          add   R30, R30, R30
44          beq   R1,  R3,  24
48          and   R12, R2,  R5
52          or    R13, R6,  R2
56          add   R14, R2,  R2
60          ...

72          lw    R4,  50(R7)
76          ...
```

	1	2	3	4	5	6	7
	IF	ID	EXE				
		IF	ID				
			IF				



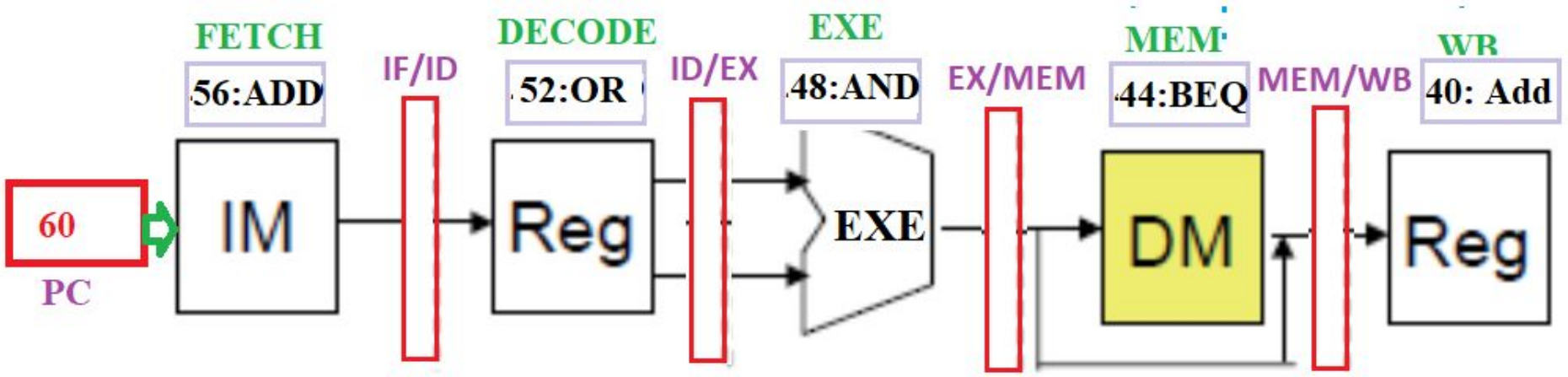
Addr	Instruction
40	add R30, R30, R30
44	beq R1, R3, 24
48	and R12, R2, R5
52	or R13, R6, R2
56	add R14, R2, R2
60	...
72	lw R4, 50(R7)
76	...

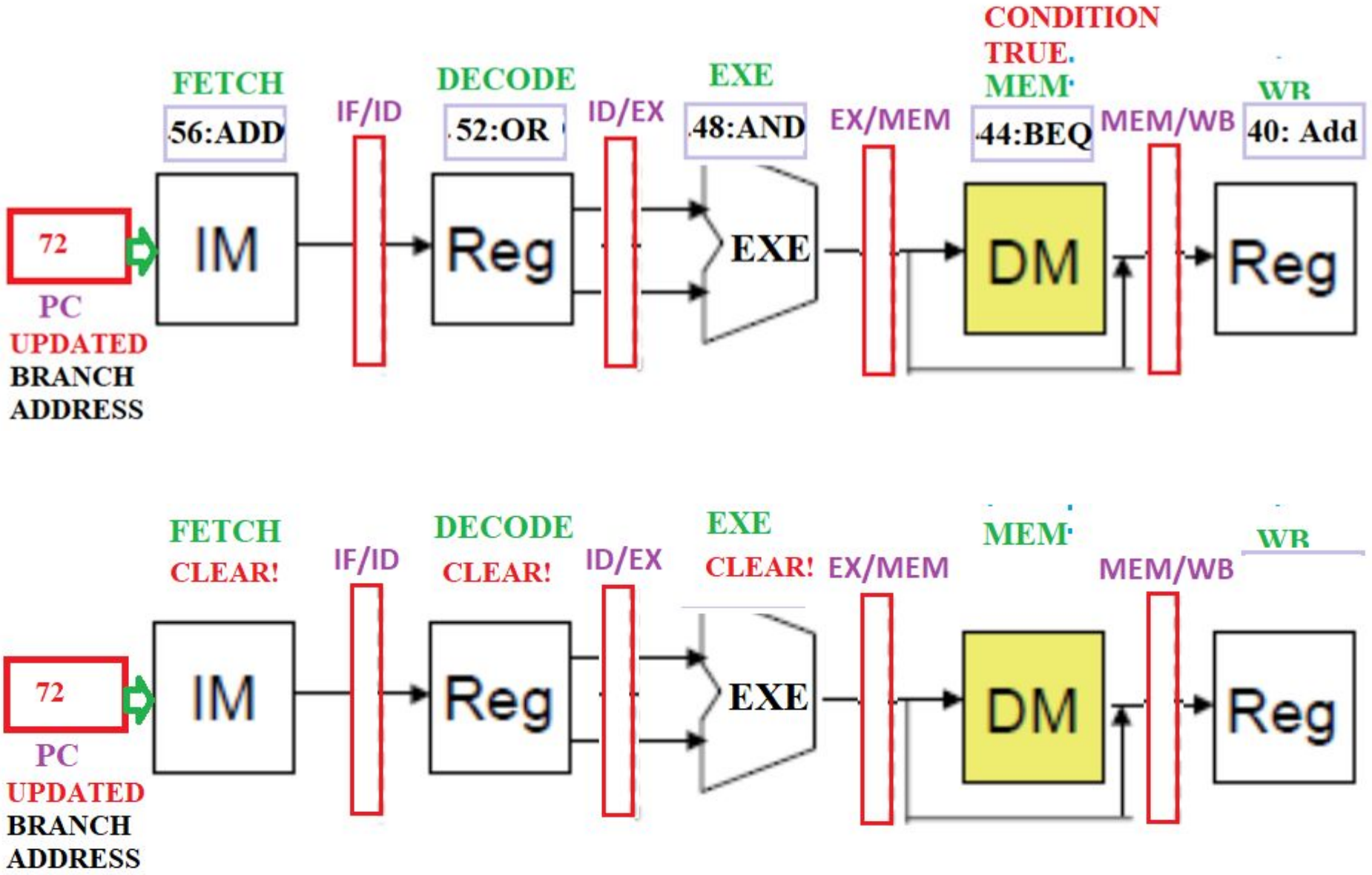




Addr	Instruction
40	add R30, R30, R30
44	beq R1, R3, 24
48	and R12, R2, R5
52	or R13, R6, R2
56	add R14, R2, R2
60	...
72	lw R4, 50(R7)
76	...

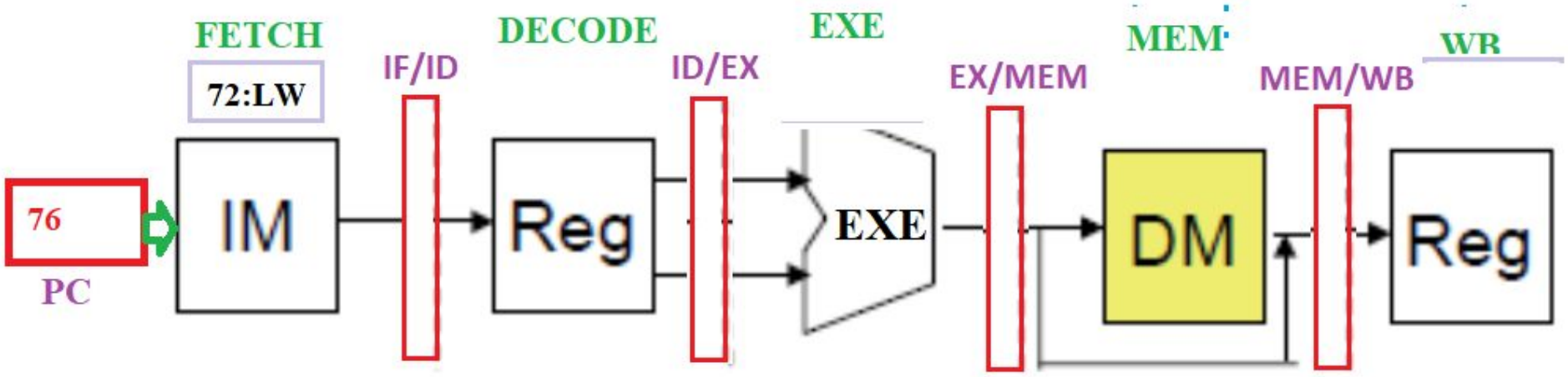
	1	2	3	4	5	6	7
	IF	ID	EXE	MEM	WB		
		IF	ID	EXE	MEM		
			IF	ID	EXE		
				IF	ID		
					IF		




CONDITION
TRUE.
FETCH
DECODE
EXE
MEM
WR
56:ADD
IF/ID
52:OR
ID/EX
48:AND
EX/MEM
44:BEQ
MEM/WB
40: Add
72
IM
Reg
EXE
DM
Reg
PC
UPDATED
BRANCH
ADDRESS
FETCH
CLEAR!
DECODE
CLEAR!
EXE
CLEAR!
MEM
WR
IF/ID
ID/EX
EX/MEM
MEM/WB
72
IM
Reg
EXE
DM
Reg
PC
UPDATED
BRANCH
ADDRESS

Addr	Instruction
40	add R30, R30, R30
44	beq R1, R3, 24
48	and R12, R2, R5
52	or R13, R6, R2
56	add R14, R2, R2
60	...
72	lw R4, 50(R7)
76	...

	1	2	3	4	5	6	7
	IF	ID	EXE	MEM	WB		
		IF	ID	EXE	MEM	CLEAR	
			IF	ID	EXE		
				IF	ID		
					IF		
						IF	



Example: Control Hazard (CISC processor)

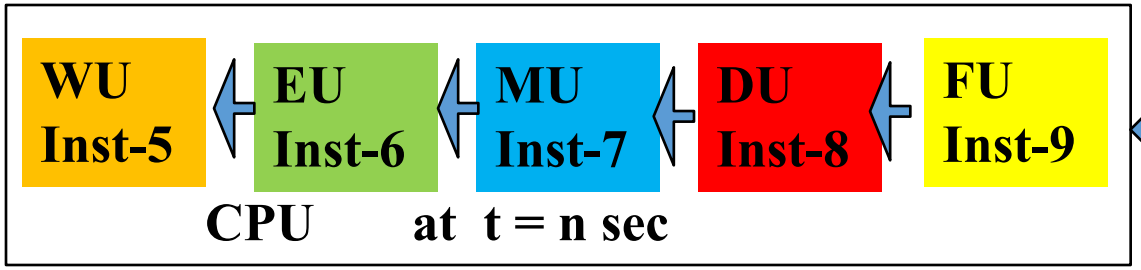
- When a branch is executed, Program Counter is not affected until the branch instruction reaches the EX stage. By this time 3 instructions have been fetched from the fall-through path.
- If the condition if TRUE, CPU reads instruction from new address and the new address is calculated at the end of EX Phase of the five stage pipelined processor.

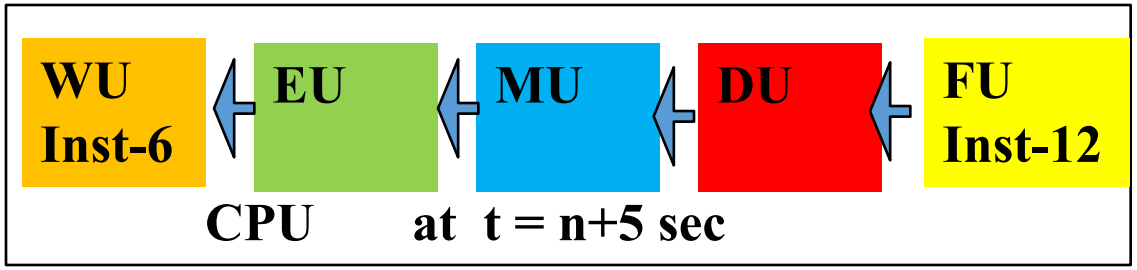
Instruction\cycle	1	2	3	4	5	6	7	8
SUB R1, R3, R3	IF	ID	MA	EX	WR			
BEQZ R1, label		IF	ID	MA	EX	WR		
OR R8, R4, R9			IF	ID	MA	Condition in **BEQZ R1, label** Is evaluated TRUE. Clear Instructions in FU, DU, MU, EU. Read new Instruction as indicated by label		
MOV R2, R5				IF	ID			
DIV R10, R7					IF			
label: XOR R2, R3, R11						IF	ID	MEM
SUB R5, R3, R1							IF	ID
ADD M1, R6, R8								IF

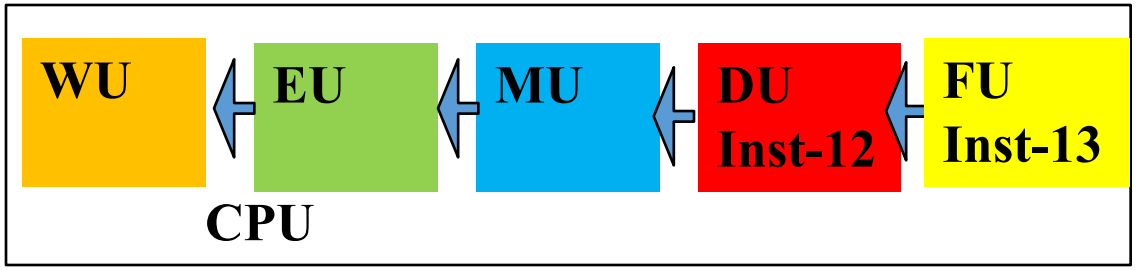
Example: Control Hazard (CISC processor)

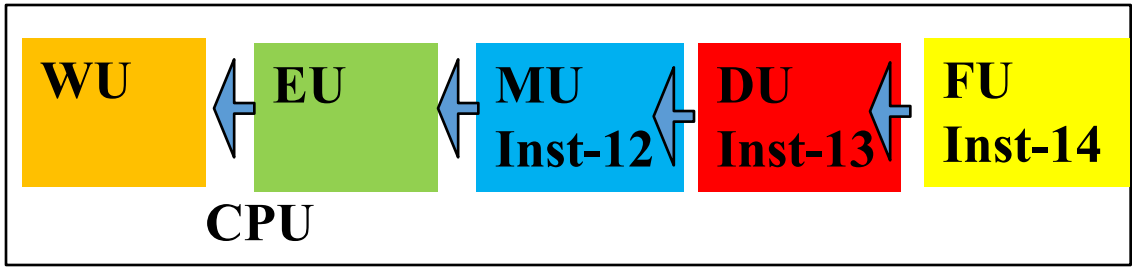
- What would happen if the condition is evaluated FALSE?
- Processing will continue. It means, the processor is designed to work normally assuming Branch instructions are evaluated FALSE.

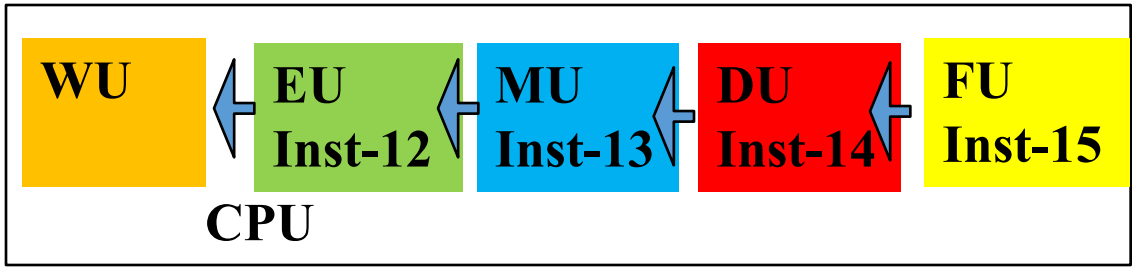
Instruction\cycle	1	2	3	4	5	6	7	8
ADD R1, R3, R3	IF	ID	MA	EX	WR			
BEQZ R1, label		IF	ID	MA	EX	WR		
OR R8, R4, R9			IF	ID	MA	EX	WR	
MOV R2, R5				IF	ID	MA	EX	WR
DIV R10, R7					IF	ID	MA	EX
label: XOR R2, R3, R11						IF	ID	MEM
SUB R5, R3, R1							IF	ID
ADD M1, R6, R8								IF











Inst-6 conditional branch

If condition true: next Instruction-12

If condition False: next Instruction-7

Example Problem

Assume that Instruction-3 is conditional branch. If the condition is TRUE, program control jumps to instruction-8. Show the pipelining stages assuming that the condition is evaluated TRUE.

Instruction	1	2	3	4	5	6	7	8	9	10
Instruction-1	IF	ID	MEM	EX	WR					
Instruction-2		IF	ID	MEM	EX	WR				
Instruction-3			IF	ID	MEM	EX	WR			
Instruction-4				IF	ID	MEM				
Instruction-5					IF	ID				
Instruction-6						IF				
Instruction-7										
Instruction-8							IF	ID	MEM	EX
Instruction-9								IF	ID	MEM
Instruction-10									IF	ID
Instruction-11										IF

Control Hazards: Branch Instructions

- What is branch penalty in a 5-stage CISC pipeline processor?

The pipeline will stall 3-clock cycles if Branch is taken (condition is evaluated True).

Problems:

If a program contains 30% branch instructions and all branches are taken. Calculate the CPI if pipeline stalls 3-clock cycles for each branch instruction.

If a program contains 30% branch instructions and 60% branches are taken. Calculate the CPI if pipeline stalls 3-clock cycles for each branch instruction.

CPI pipelined = Ideal CPI + Pipeline stall cycles per instruction
= 1 + Pipeline stall cycles per instruction

Instruction-3: conditional Branch
If Branch condition is TRUE, Jump to Instruction-15

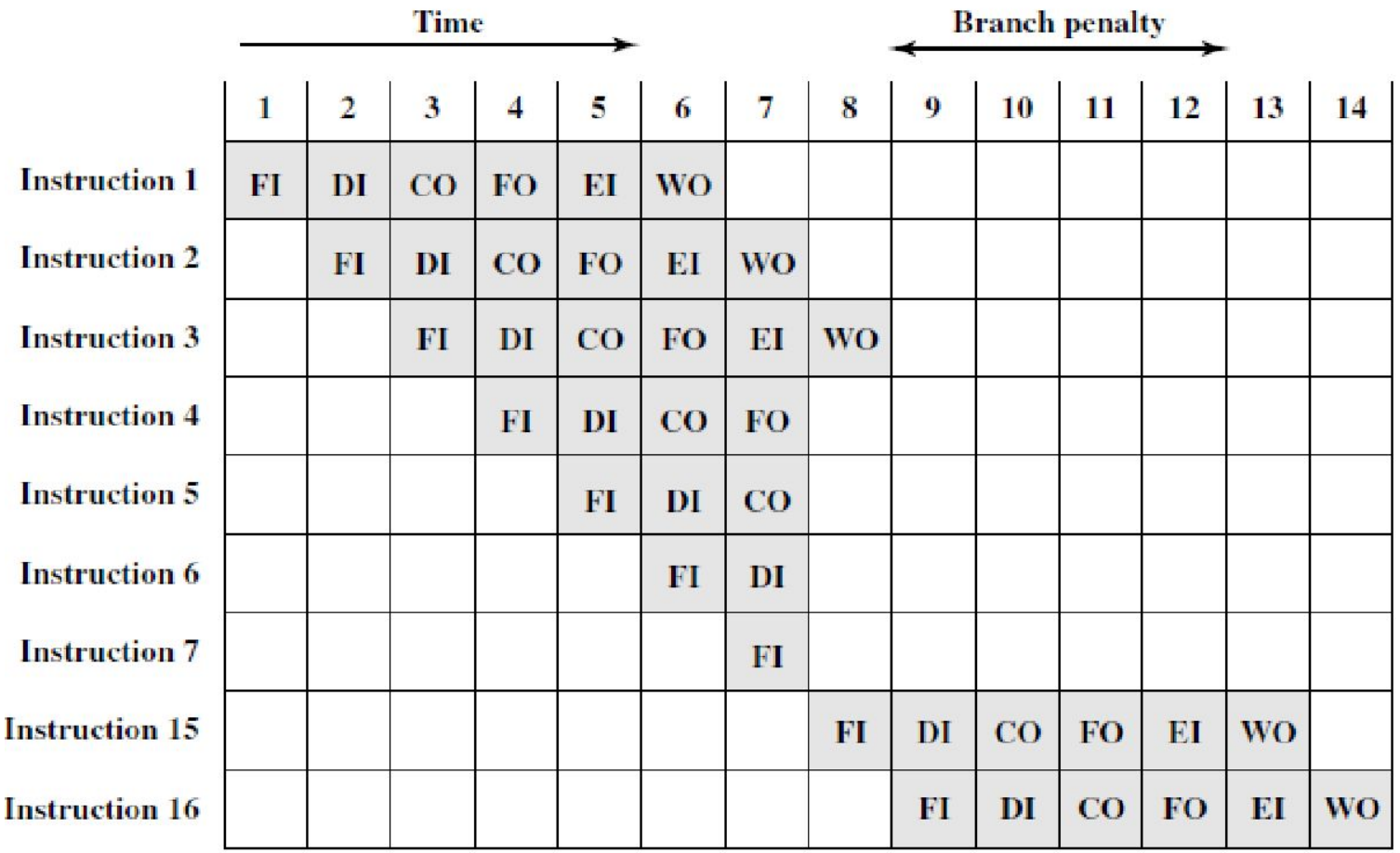


Figure 12.11 The Effect of a Conditional Branch on Instruction Pipeline Operation

Example Problem

Assume that Instruction-4 is conditional branch. If the condition is TRUE, program control jumps to instruction-9. Show the pipelining stages assuming that the condition is evaluated TRUE.

Instruction	1	2	3	4	5	6	7	8	9	10	11
Instruction-1											
Instruction-2											
Instruction-3											
Instruction-4											
Instruction-5											
Instruction-6											
Instruction-7											
Instruction-8											
Instruction-9											
Instruction-10											
Instruction-11											

Conditional branch on MIPS processor

operation of the branch-on-equal instruction, such as

BEQ R1, R2, 25

It operates much like an ALU instruction, but the ALU output is used to determine whether the PC is written with PC + 4 or the branch target address.

Four steps in execution:

1. An instruction is fetched from the instruction memory, and the PC is incremented.

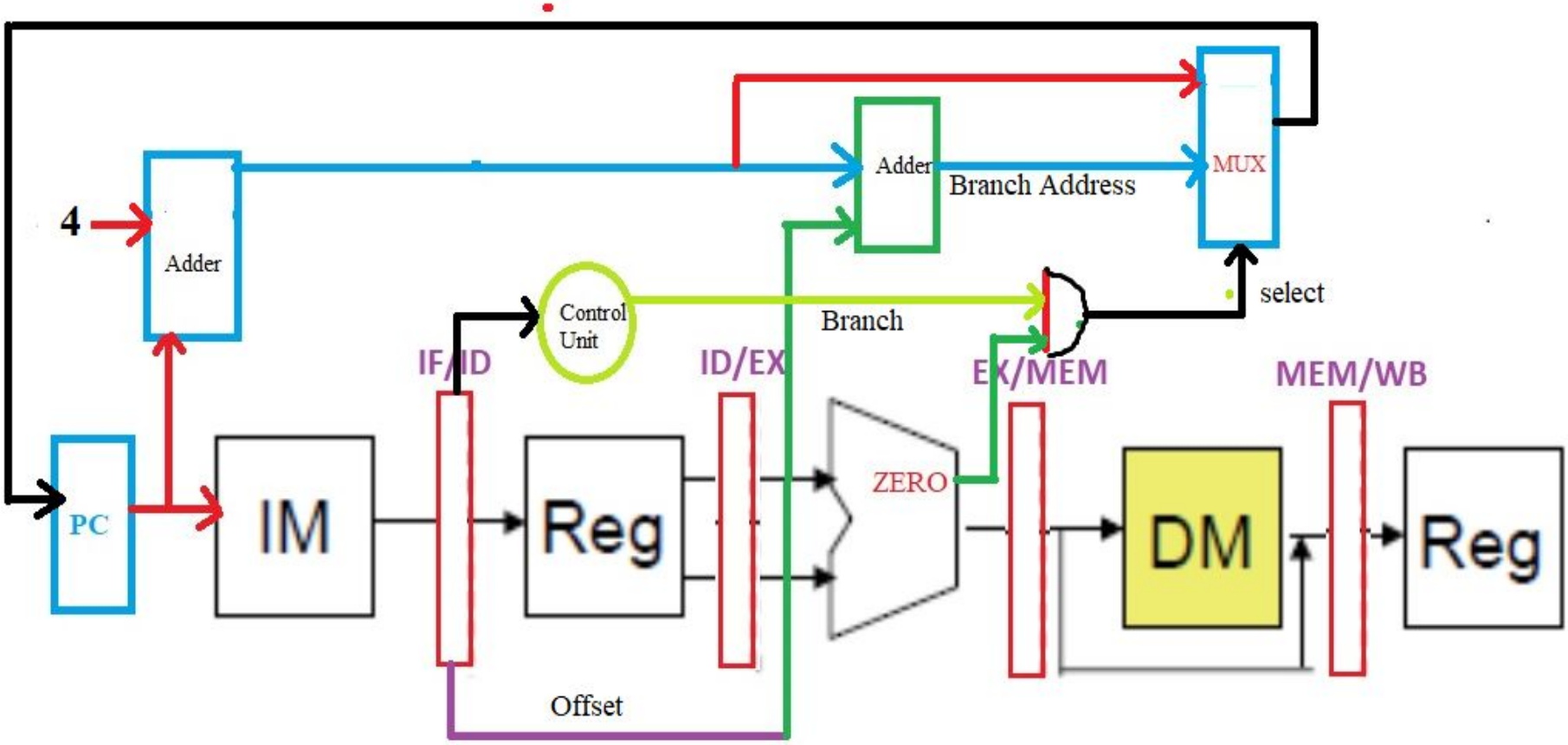
2. Two registers, R1 and R2 are read from the register fi le.

3. The ALU performs a subtract on the data values read from the register file. The value of PC + 4 is added to the sign-extended, lower 16 bits of the instruction (offset) shifted left by two; the result is the branch target address.

4. The Zero result from the ALU is used to decide which adder result to store into the PC.

If the condition if TRUE, CPU reads instruction from new address and the new address is calculated at the end of Memory Access (MA) Phase of the five stage pipelined processor.

Datapath for Branch Instruction:
BEQ R1, R2, OFFSET



Problem (RISC processor): IF-ID-EXE-MEM-WB

Assume a machine using Five-stage pipelining runs a program. Instruction-3 is a conditional branch instruction. If the condition is TRUE, CPU jumps to Instruction-8. Show the time steps of pipelining stages assuming that condition is evaluated TRUE.

If the condition if TRUE, CPU reads instruction from new address and the new address is calculated at the end of EXE phase of the five stag e pipelined processor.

Inst	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**
Inst-1	IF	ID	EXE	MEM	WB									
Inst-2		IF	ID	EXE	MEM	WB								
Inst-3			IF	ID	EXE									
Inst-4				IF	ID									
Inst-5					IF									
Inst-6														
Inst-7														
Inst-8						IF	ID	EXE	MEM					
Inst-9							IF	ID	EXE					
Inst-10								IF	ID					
Inst-11									IF					

Problem (RISC processor): IF-ID-ALU-MA-WB

Assume a machine using Five-stage pipelining runs a program. Instruction-3 is a conditional branch instruction. If the condition is TRUE, CPU jumps to Instruction-8. Show the time steps of pipelining stages assuming that condition is evaluated TRUE.

If the condition if TRUE, CPU reads instruction from new address and the new address is calculated at the end of Memory Access (MA) Phase of the five stage pipelined processor.

Inst	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Inst-1	IF	ID	ALU	MA	WB									
Inst-2		IF	ID	ALU	MA	WB								
Inst-3			IF	ID	ALU	MA								
Inst-4				IF	ID	ALU								
Inst-5					IF	ID								
Inst-6						IF								
Inst-7														
Inst-8							IF	ID	ALU	MA				
Inst-9								IF	ID	ALU				
Inst-10									IF	ID				
Inst-11										IF				

Control Hazards: Branch Instructions

- What is branch penalty in a 5-stage RISC pipeline processor?

The pipeline will stall 3-clock cycles if Branch is taken (condition is evaluated True).

Problems:

If a program contains 30% branch instructions and all branches are taken. Calculate the CPI if pipeline stalls 3-clock cycles for each branch instruction.

If a program contains 30% branch instructions and 60% branches are taken. Calculate the CPI if pipeline stalls 3-clock cycles for each branch instruction.

CPI pipelined = Ideal CPI + Pipeline stall cycles per instruction
= 1 + Pipeline stall cycles per instruction

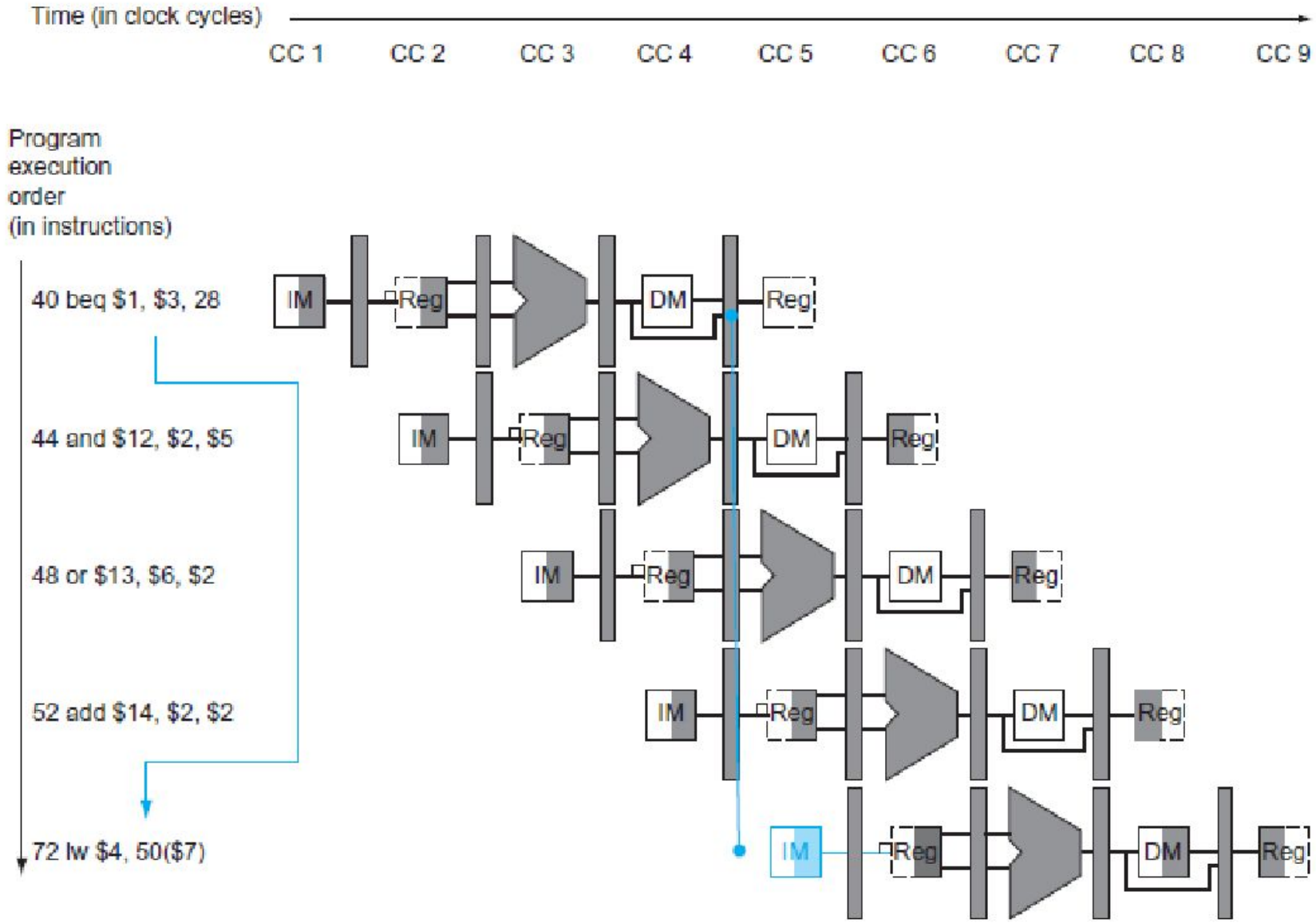


FIGURE 4.61 The impact of the pipeline on the branch instruction. The numbers to the left of the instruction (40, 44, ...) are the addresses of the instructions. Since the branch instruction decides whether to branch in the MEM stage—clock cycle 4 for the `beq` instruction above—the three sequential instructions that follow the branch will be fetched and begin execution. Without intervention, those three following instructions will begin execution before `beq` branches to `lw` at location 72. (Figure 4.31 assumed extra hardware to reduce the control hazard to one clock cycle; this figure uses the nonoptimized datapath.)

Problem

- Assume a machine using Five-stage pipelining runs a program. Instruction-3 is a conditional branch instruction. If the condition is TRUE, CPU skips next three instructions. Instruction-8 is also a conditional branch instruction and if it is TURE, program control returns to Instruction-4. Show the time steps of pipelining stages assuming that both Instructions 3 and 8 are evaluated TRUE.

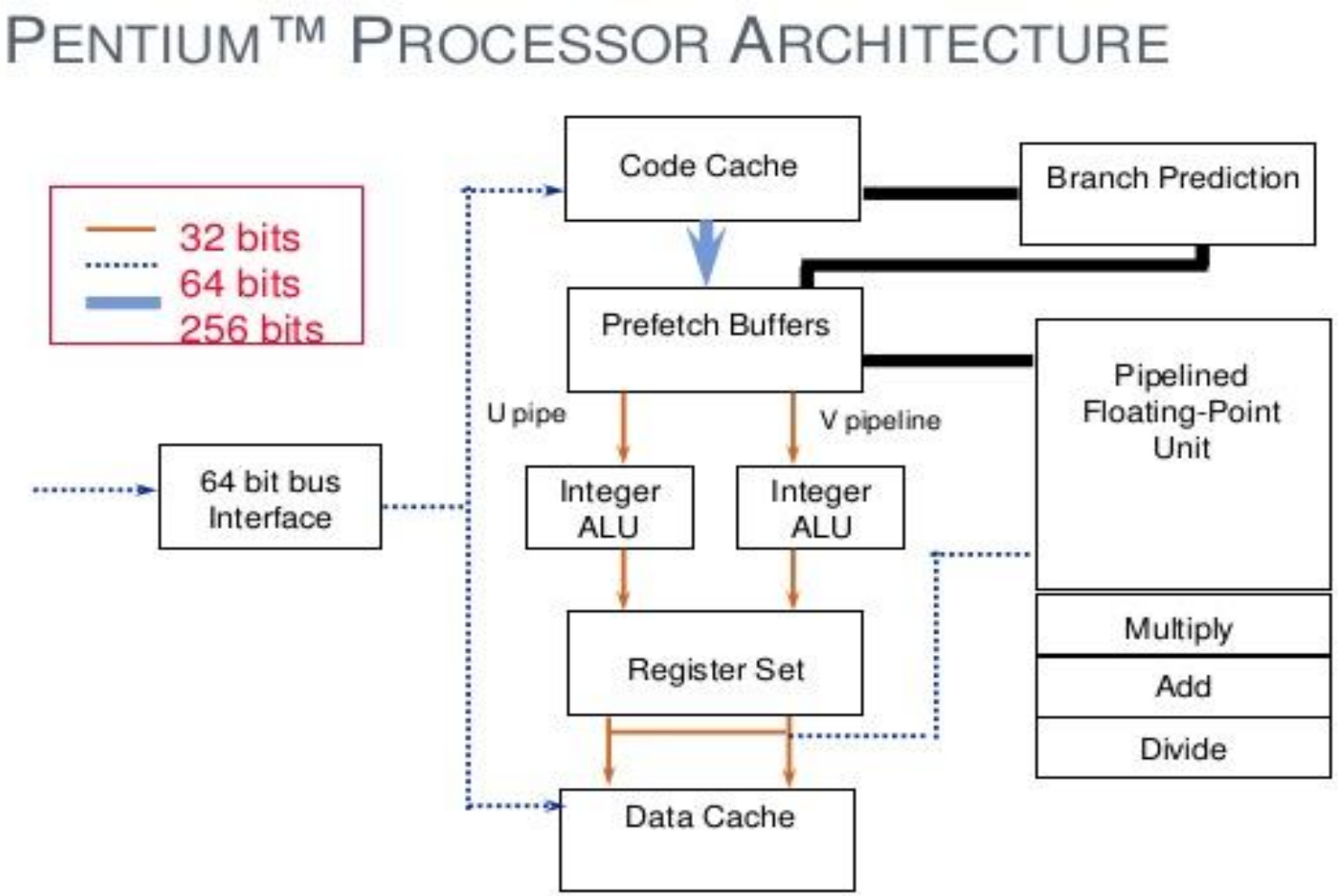
Dealing with Control Hazards

- We can stall until branch outcome is known
  - Once branch is known, then fetch
  - But this is wasteful

Dealing with Control Hazards - Predict Branch

- Predict that the branch is not taken
  - Attempt to get next instruction from the fall thru of the branch (i.e., next sequential address)
- We are gambling that the branch isn't ever going to be taken
- When we're right - there is no stall
- But what happens when we're wrong????

Branch Prediction: Pentium



- **Assuming the following assembly instructions are executed in the order that they are found in the table below, fill out the chart indicating the stage of the standard 5-stage pipeline that the instruction will be in during the clock cycles. If an instruction is not in any stage during a cycle, simply leave that box blank. Indicate the stages of the pipeline using: IF, ID, MEM, EX, and WR. If there is gap in stages due to a data hazard, make sure to indicate the data hazard using d*.**

	1	2	3	4	5	6	7	8	9
ADD R10, R2, R3									
SUB R4, R1, R5									
AND R6, R4, R7									
OR R8, R12, R9									
XOR R10, R1, R11									