

ECEEC-355: Computer Organization & Architecture

Dr. Anup K. Das

*Electrical and Computer Engineering
Drexel University*

Academic Year – 2021-2022

Pipeline Design Steps

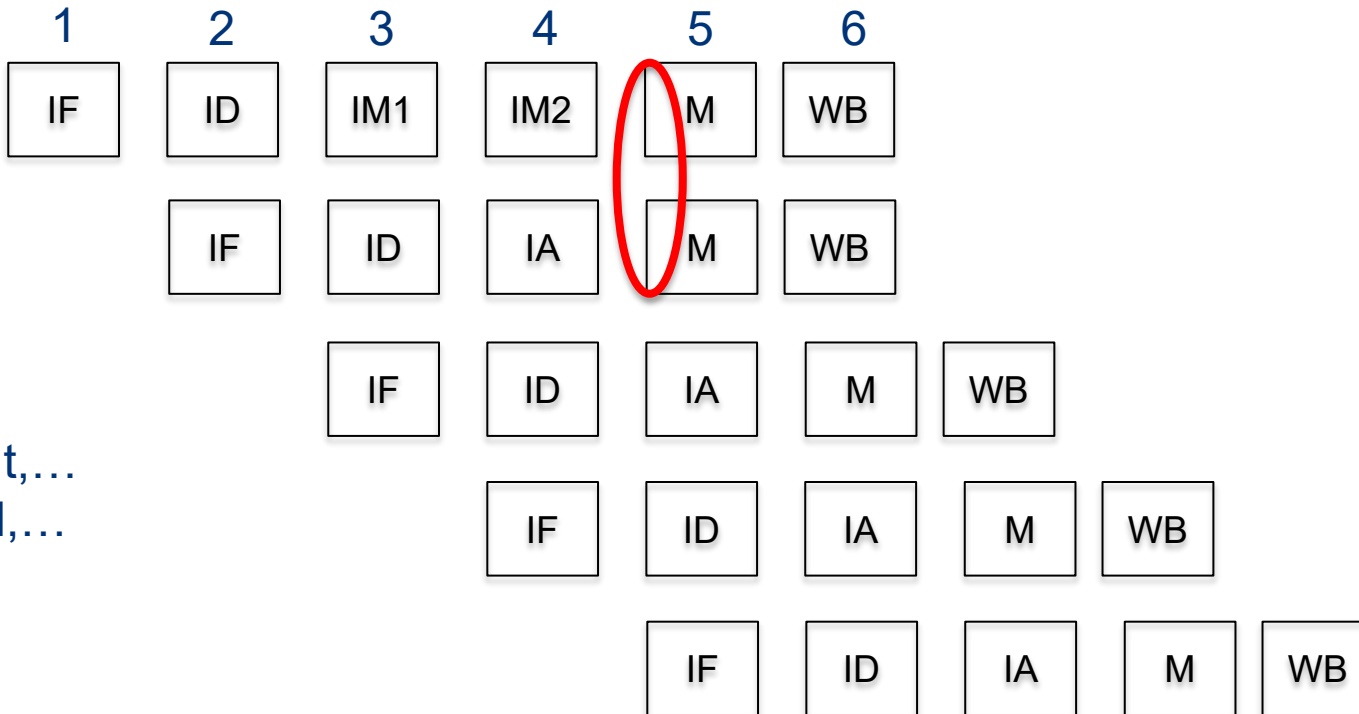
- Balancing work in pipeline stages
- Keeping the pipeline correct, moving, and full in the presence of events that disrupt pipeline flow
 - Dependencies
 - Resource contentions
 - Long latency operations
- Handling exceptions, interrupts
- Improving pipeline throughput

Hazards

- **Situations that prevent starting the next instruction in the next cycle**
- **Structure hazards**
 - A required resource is busy
- **Data hazard**
 - Need to wait for previous instruction to complete its data read/write
- **Control hazard**
 - Deciding on control action depends on previous instruction

Structure Hazards

- Conflict for use of a resource



I1: imult,...

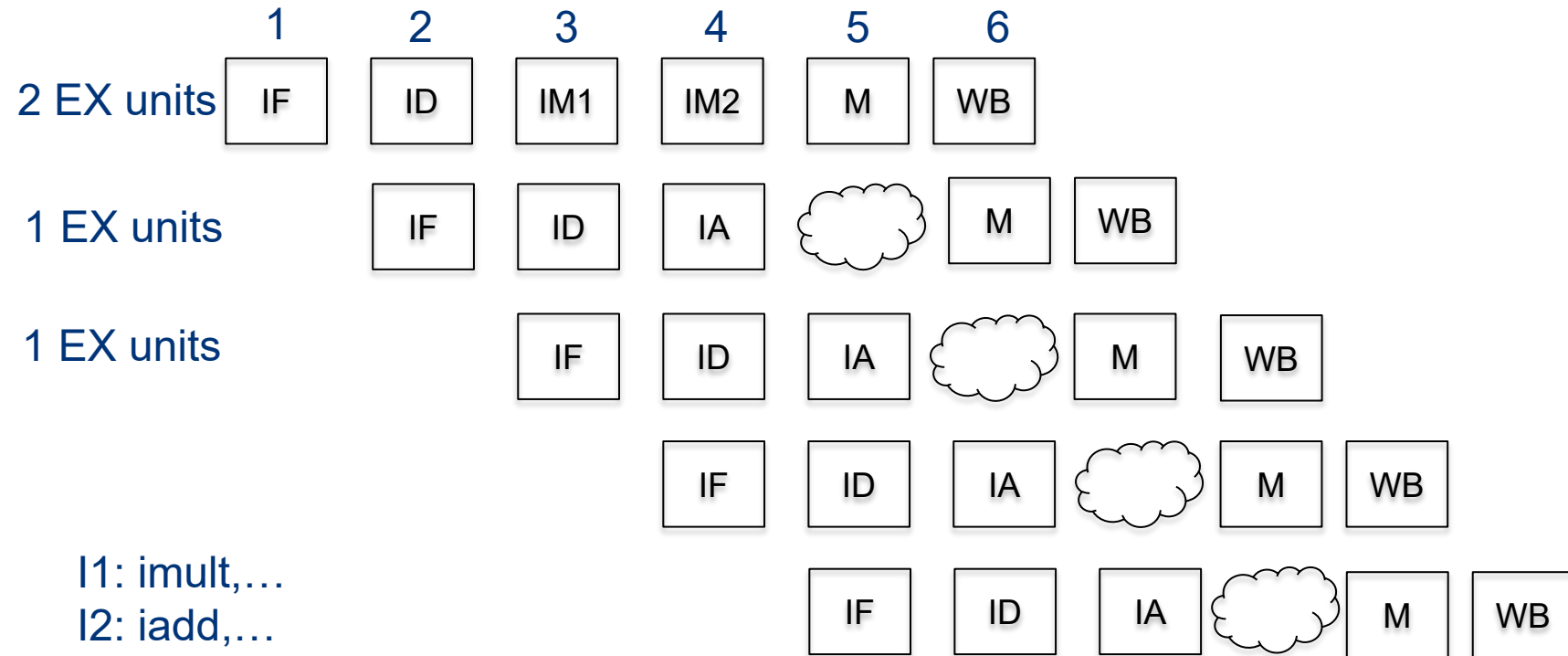
I2: iadd,...

imult takes 2 cycles

All other execution takes 1 cycle

Structure Hazards

- **Conflict for use of a resource**



I1: imult,...

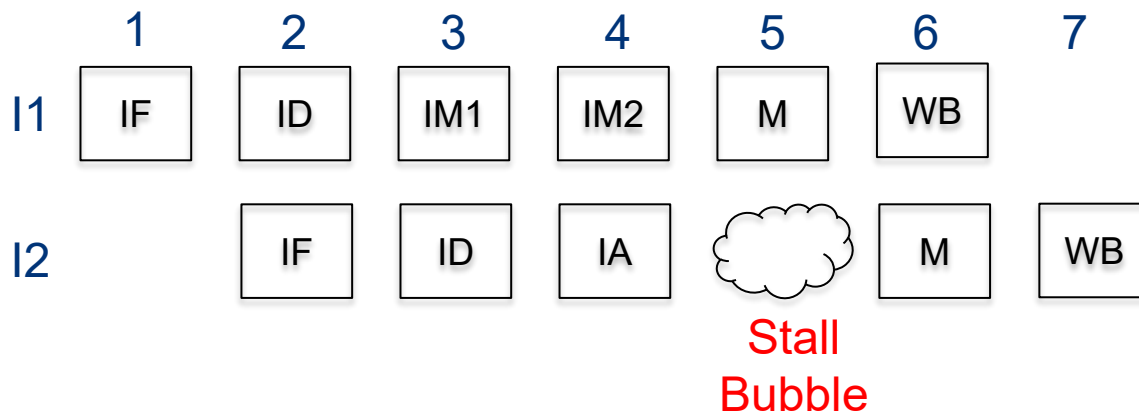
I2: iadd,...

imult takes 2 cycles

All other execution takes 1 cycle

Resource Contention

- Happens when instructions in two pipeline stages need the same resource
- **Solution 1: Eliminate the cause of contention**
 - Duplicate the resource or increase its throughput
- **Solution 2: Detect the resource contention and stall one of the contending stages**



Hazards

- **Situations that prevent starting the next instruction in the next cycle**
- **Structure hazards**
 - A required resource is busy
- **Data hazard**
 - Need to wait for previous instruction to complete its data read/write
- **Control hazard**
 - Deciding on control action depends on previous instruction

Data Hazards

- An instruction depends on completion of data access by a previous instruction

– add **x19**, x0, x1 Expected x19 = 15
 sub x2, **x19**, x3 Expected x2 = 8

- For simplicity we assume both these instructions takes 1 cycles to execute

1 2 3 4 5 6

X0	0
X1	15
X19	11
X2	14
X3	7

Data Hazards

- An instruction depends on completion of data access by a previous instruction

– add **x19**, x0, x1 Expected x19 = 15
 sub x2, **x19**, x3 Expected x2 = 8

- For simplicity we assume both these instructions takes 1 cycles to execute



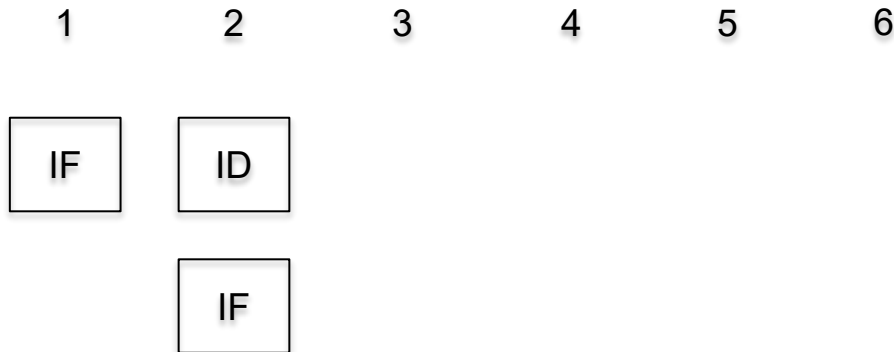
X0	0
X1	15
X19	11
X2	14
X3	7

Data Hazards

- An instruction depends on completion of data access by a previous instruction

– add **x19**, x0, x1 Expected x19 = 15
 sub x2, **x19**, x3 Expected x2 = 8

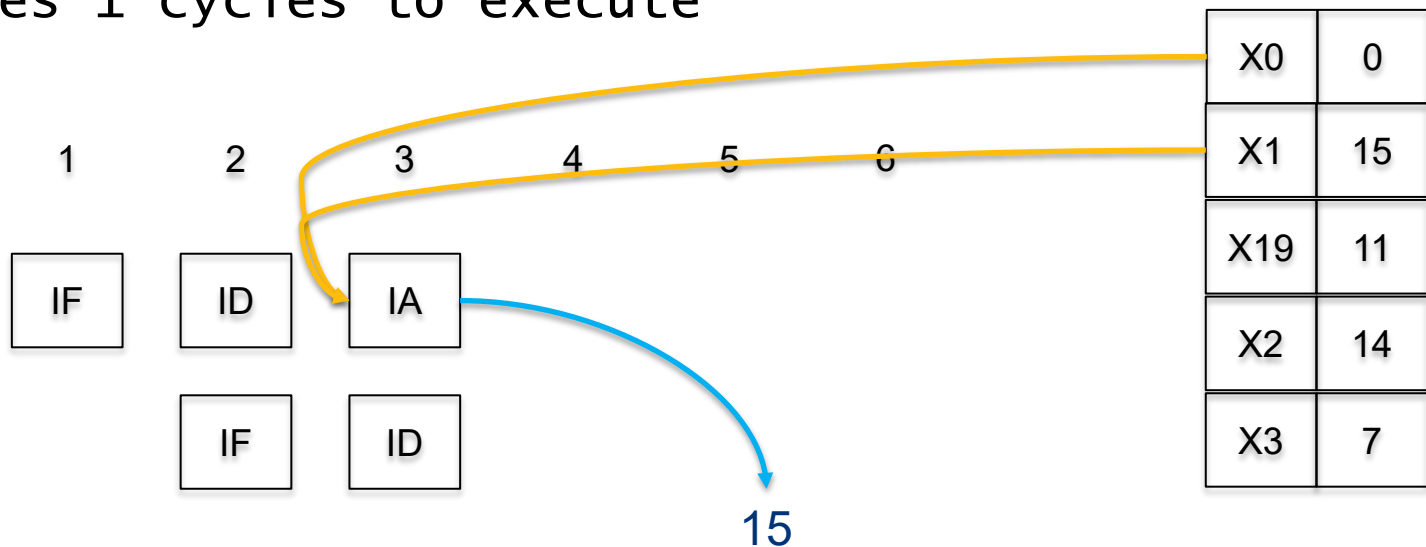
- For simplicity we assume both these instructions takes 1 cycles to execute



X0	0
X1	15
X19	11
X2	14
X3	7

Data Hazards

- An instruction depends on completion of data access by a previous instruction
 - add **x19**, x0, x1 Expected x19 = 15
 - sub x2, **x19**, x3 Expected x2 = 8
- For simplicity we assume both these instructions takes 1 cycles to execute

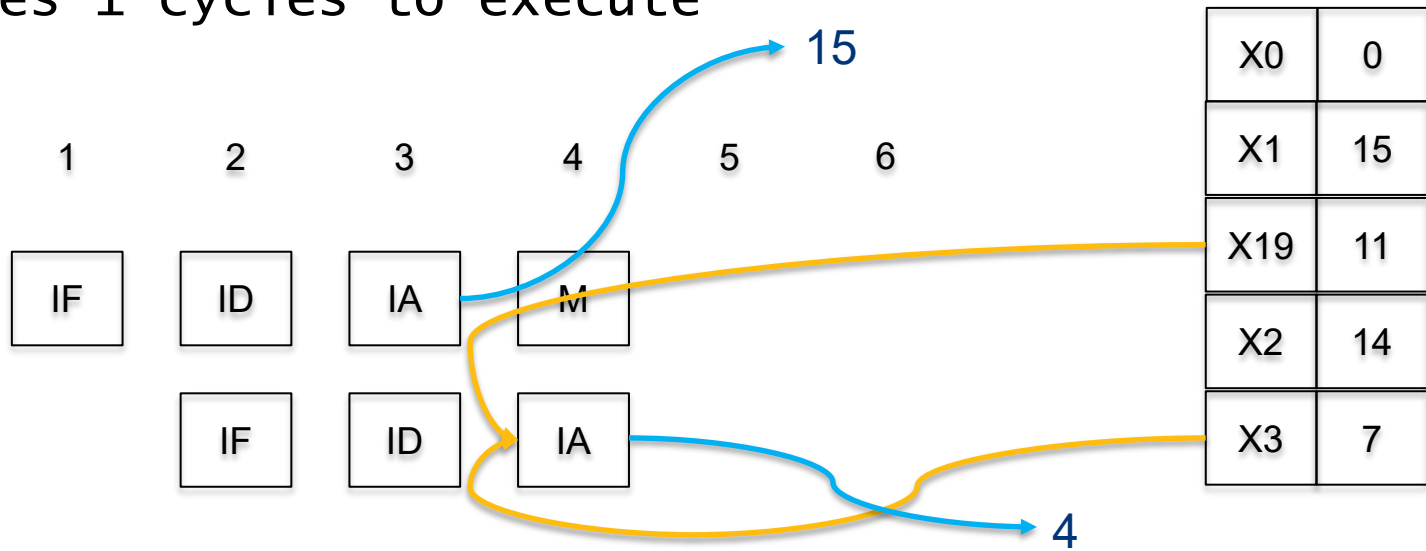


Data Hazards

- An instruction depends on completion of data access by a previous instruction

– add **x19**, x0, x1 Expected x19 = 15
 sub x2, **x19**, x3 Expected x2 = 8

- For simplicity we assume both these instructions takes 1 cycles to execute

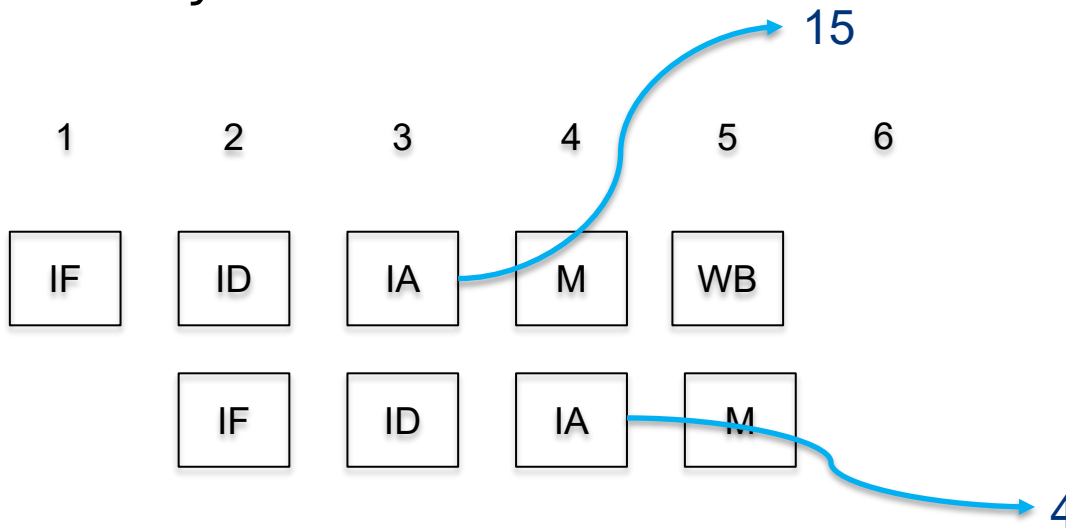


Data Hazards

- An instruction depends on completion of data access by a previous instruction

– add **x19**, x0, x1 Expected x19 = 15
 sub x2, **x19**, x3 Expected x2 = 8

- For simplicity we assume both these instructions takes 1 cycles to execute



X0	0
X1	15
X19	11
X2	14
X3	7

Data Hazards

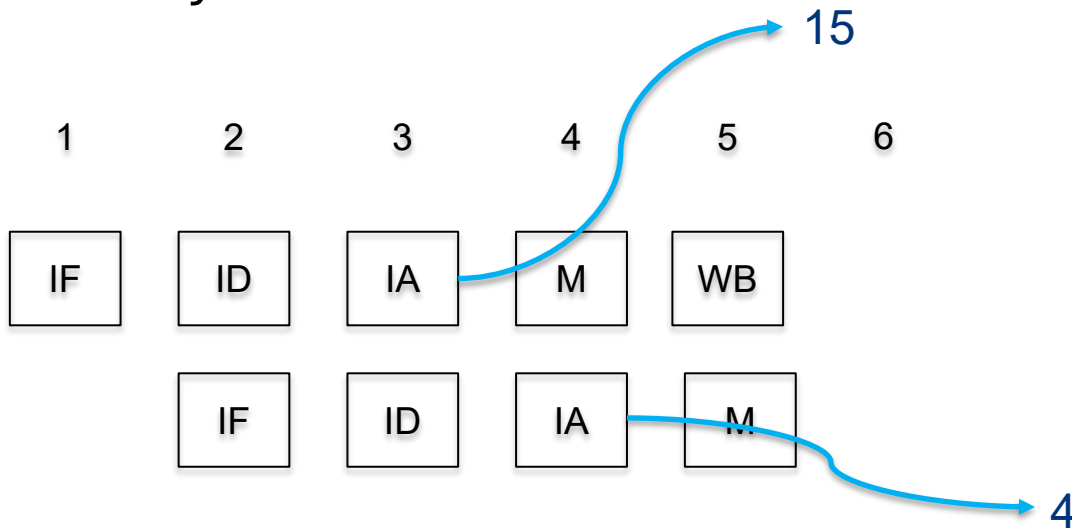
- An instruction depends on completion of data access by a previous instruction

– add **x19**, x0, x1
sub x2, **x19**, x3

Expected x19 = 15

Expected x2 = 8

- For simplicity we assume both these instructions takes 1 cycles to execute



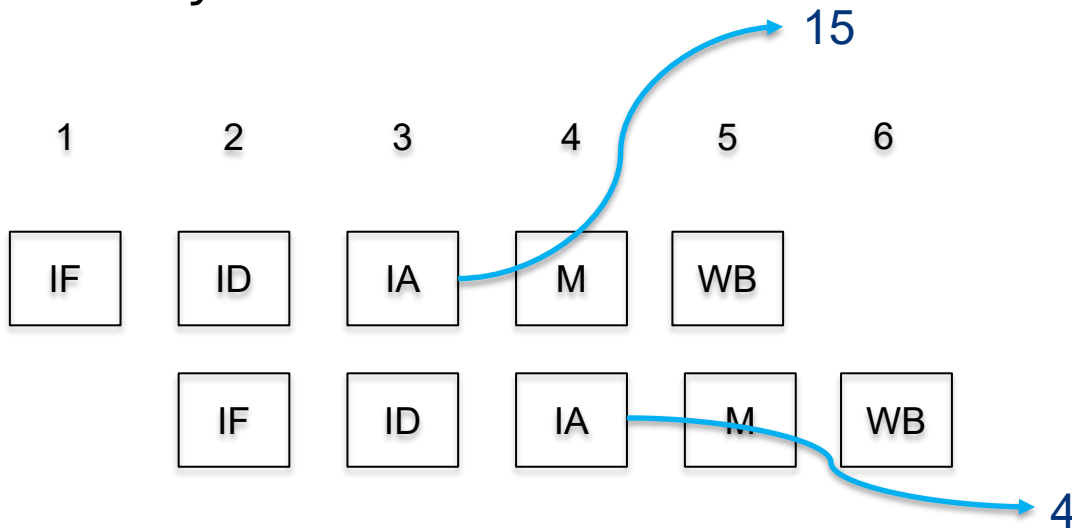
X0	0
X1	15
X19	15
X2	14
X3	7

Data Hazards

- An instruction depends on completion of data access by a previous instruction

– add **x19**, x0, x1 Expected x19 = 15
 sub x2, **x19**, x3 Expected x2 = 8

- For simplicity we assume both these instructions takes 1 cycles to execute



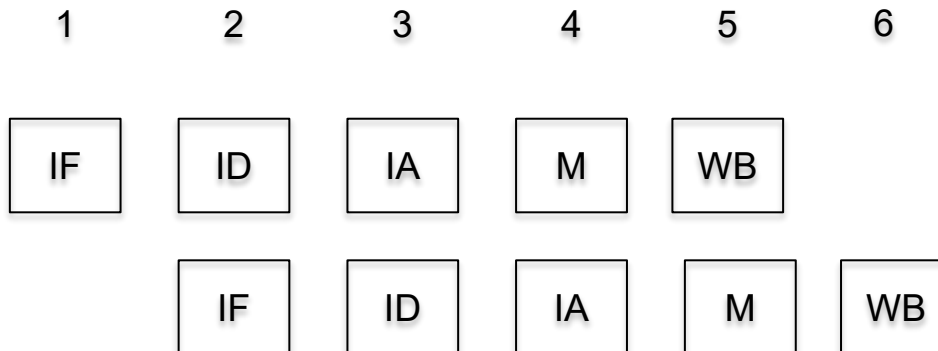
X0	0
X1	15
X19	15
X2	14
X3	7

Data Hazards

- An instruction depends on completion of data access by a previous instruction

– add **x19**, x0, x1 Expected x19 = 15
 sub x2, **x19**, x3 Expected x2 = 8

- For simplicity we assume both these instructions takes 1 cycles to execute



X0	0
X1	15
X19	15
X2	4
X3	7


Data Hazards or Dependence

- **Happens when the same data (register or memory) is being used by multiple instructions in the pipeline**
- **True or Flow dependence (RAW)**
 - Read after write
 - Read must wait until earlier write finishes
- **Anti-dependence (WAR)**
 - Write after read
 - Write must wait until earlier read finishes
- **Output dependence (WAW)**
 - Write after write
 - Earlier write can't overwrite later write

Data Dependence Type


Flow/True dependence

$r_3 \leftarrow r_1 \text{ op } r_2$ Read-after-Write
 $r_5 \leftarrow r_3 \text{ op } r_4$ (RAW)




Anti dependence

$r_3 \leftarrow r_1 \text{ op } r_2$ Write-after-Read
 $r_1 \leftarrow r_4 \text{ op } r_5$ (WAR)



Output-dependence

$r_3 \leftarrow r_1 \text{ op } r_2$ Write-after-Write
 $r_5 \leftarrow r_3 \text{ op } r_4$ (WAW)
 $r_3 \leftarrow r_6 \text{ op } r_7$



Dependencies Example

- **Int a = 10;**
- **Int b = 12;**
- **Int c = 100;**
- **Ind d = 21;**
- **Int e;**

Registers

x1 \leftarrow a

x2 \leftarrow b

x3 \leftarrow c

x4 \leftarrow d

x5 \leftarrow e

This is a value dependence scenario

- **d = a f(*) b;** `mul x4, x1, x2`
- **e = c+d;** `add x5, x3, x4`

This is a real (true) dependence: RAW

RAW

1	2	3	4	5	6	7	8	9	10	11	12
F ₀ D ₀ E ₀ E ₀ E ₀ E ₀ E ₀ M ₀ W ₀											
F ₁ D ₁									E ₁ M ₁ W ₁		

Dependencies Example

- **Int a = 10;**
- **Int b = 12;**
- **Int c = 100;**
- **Int d = 21;**
- **Int e;**
- **Int f;**

Registers

x1 \leftarrow a

x2 \leftarrow b

x3 \leftarrow c

x4 \leftarrow d

x5 \leftarrow e

This is a no-dependence scenario

- **e = a f(*) b;**
- **f = c + d;**

Lets consider a scenario where only 5 registers in the ISA and x0 is reserved

We have to assign a register for the variable f

Dependencies Example

- **Int a = 10;**
- **Int b = 12;**
- **Int c = 100;**
- **Int d = 21;**
- **Int e;**
- **Int f;**

Registers

x1 \leftarrow a

x2 \leftarrow b

x3 \leftarrow c

x4 \leftarrow d

x5 \leftarrow e

x1 \leftarrow f Reuse the variable name

This is a no-dependence scenario

- **e = a f(*) b;**
- **f = c + d;**

Lets consider a scenario where only 5 registers in the ISA and x0 is reserved

We have to assign a register for the variable f

Dependencies Example

- **Int a = 10;**
- **Int b = 12;**
- **Int c = 100;**
- **Int d = 21;**
- **Int e;**
- **Int f;**

Registers

x1 \leftarrow a

x2 \leftarrow b

x3 \leftarrow c

x4 \leftarrow d

x5 \leftarrow e

x1 \leftarrow f Reuse the variable name

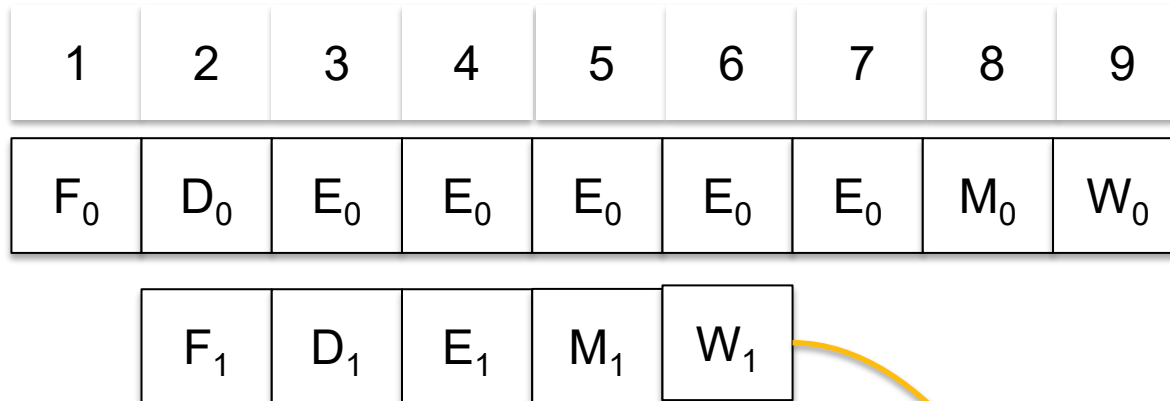
This is a no-dependence scenario

- **e = a f(*) b;** mul x5, x1, x2
- **f = c + d;** add x1, x3, x4

Anti-dependence: WAR

Anti-dependence (WAR) exists because there are limited number of ISA registers
WAR is a name dependency not value dependency. So its not a true dependence

WAR



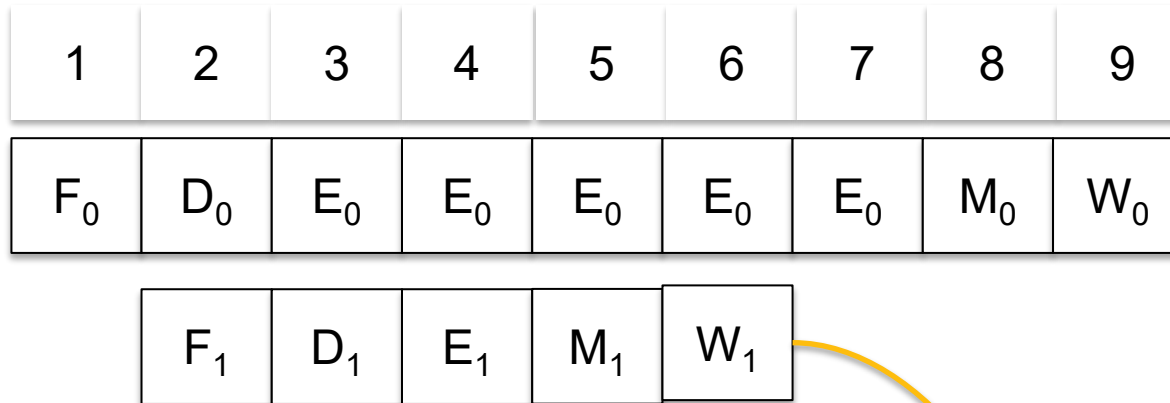
mul x5, x1, x2

add x1, x3, x4

x1 is updated to 13

x1	10
x2	2
x3	5
x4	8

WAR



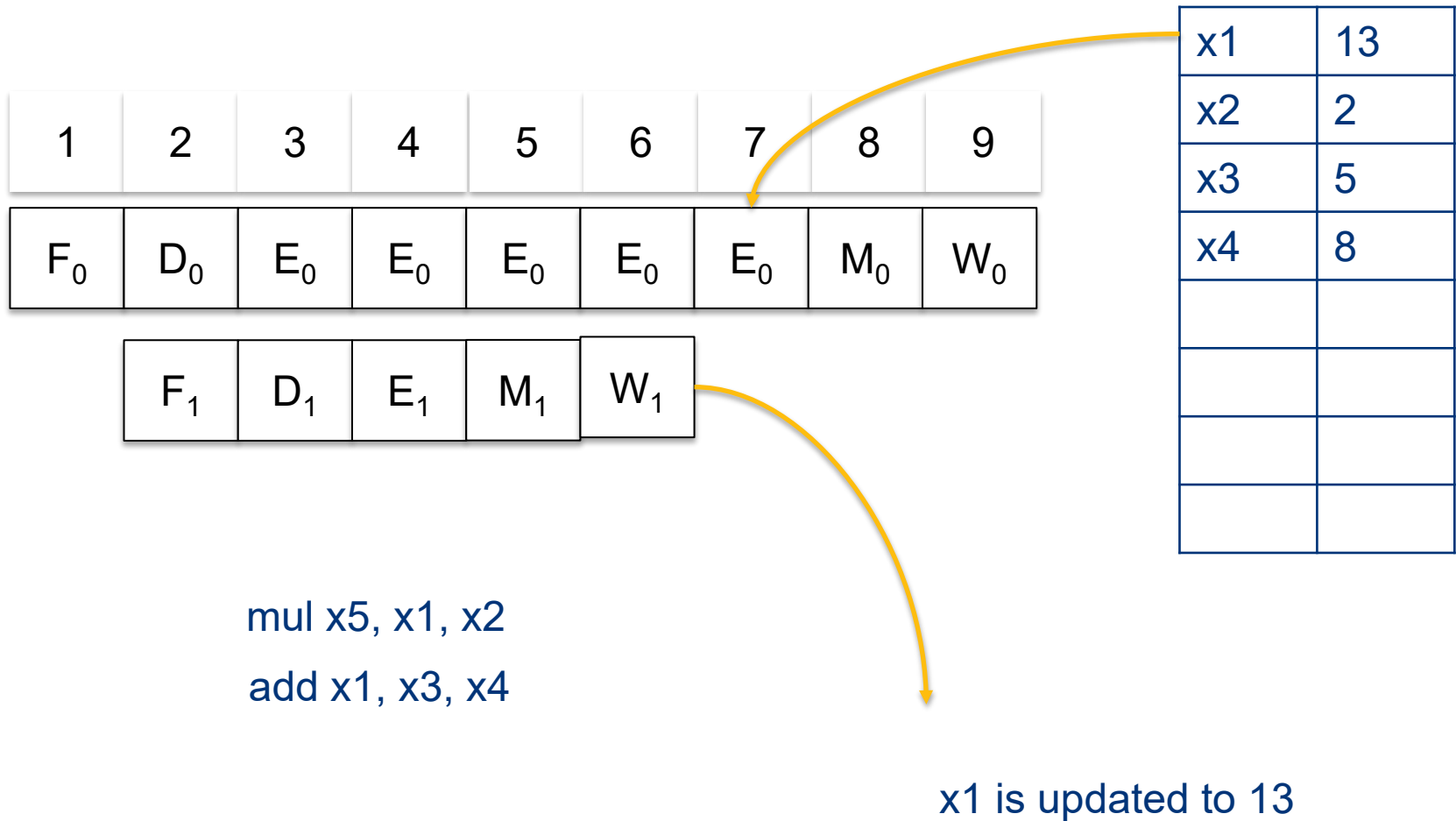
x1	13
x2	2
x3	5
x4	8

mul x5, x1, x2

add x1, x3, x4

x1 is updated to 13

WAR



Dependencies Example

- **Int a = 10;**
- **Int b = 12;**
- **Int c = 100;**
- **Int d = 21;**
- **Int e;**
- **Int f;**

Registers

x1 \leftarrow a

x2 \leftarrow b

x3 \leftarrow c

x4 \leftarrow d

x5 \leftarrow e

This is a no-dependence scenario

- **e = a f(*) b;**
- **f = c + d;**

Lets consider a scenario where only 5 registers in the ISA and x0 is reserved

We have to assign a register for the variable f

Dependencies Example

- **Int a = 10;**
- **Int b = 12;**
- **Int c = 100;**
- **Int d = 21;**
- **Int e;**
- **Int f;**

Registers

x1 \leftarrow a

x2 \leftarrow b

x3 \leftarrow c

x4 \leftarrow d

x5 \leftarrow e

x5 \leftarrow f Reuse the variable name

This is a no-dependence scenario

- **e = a f(*) b;**
- **f = c + d;**

Lets consider a scenario where only 5 registers in the ISA and x0 is reserved

We have to assign a register for the variable f

Dependencies Example

- **Int a = 10;**
- **Int b = 12;**
- **Int c = 100;**
- **Int d = 21;**
- **Int e;**
- **Int f;**

Registers

x1 ← a

x2 ← b

x3 ← c

x4 ← d

x5 ← e

x5 ← f Reuse the variable name

This is a no-dependence scenario

- **e = a f(*) b;** mul x5, x1, x2
- **f = c + d;** add x5, x3, x4

Output-dependence: WAW

Output-dependence (WAW) exists because there are limited number of ISA registers
WAW is a name dependency not value dependency. So its not a true dependence

Dependence

- **For all of them, we need to ensure semantics of the program is correct**
- **True dependences always need to be obeyed because they constitute true dependence on a value**
- **Anti and output dependences exist due to limited number of architectural registers**
 - They are dependence on a name, not a value
 - Write to the destination in one stage and in program order
- **We will focus on true dependence for now**

Data Dependence

- How to detect dependence?
- How to handle dependence?

Data Dependence

- How to detect dependence?
- How to handle dependence?

How to Detect Dependence?

- **Interlocking:** Detection of dependence between instructions in a pipelined processor to guarantee correct execution
- **Hardware-based interlocking**
 - Techniques at the hardware level to detect and resolve
- **Software based interlocking**
 - Compiler guarantees correctness by inserting no-ops or independent instructions between dependent instructions
 - Static detection

How to Detect Dependence?

- **Interlocking:** Detection of dependence between instructions in a pipelined processor to guarantee correct execution
- **Hardware-based interlocking**
 - Techniques at the hardware level to detect and resolve
- **Software based interlocking**
 - Compiler guarantees correctness by inserting no-ops or independent instructions between dependent instructions
 - Static detection
- **MIPS = Microprocessor without Interlocked Pipeline Stages**

Hardware Interlocking

- **Scoreboarding**
- **Combinational dependence check logic**

Hardware Interlocking

- **Scoreboarding**
- **Combinational dependence check logic**

Scoreboarding

- Each register in register file has a **Valid Bit** associated with it
- An instruction that is writing to the register **resets** the Valid Bit
- An instruction in **Decode** stage **checks** if all its source and destination registers are Valid
 - **Yes**: No need to stall... No dependence
 - **No**: Stall the instruction
- **Advantage:**
 - Simple. 1 bit per register
- **Disadvantage:**
 - Need to stall for all types of dependences, not only true dependence

Hardware Interlocking

- Scoreboarding
- Combinational dependence check logic

Combinational dependence check logic

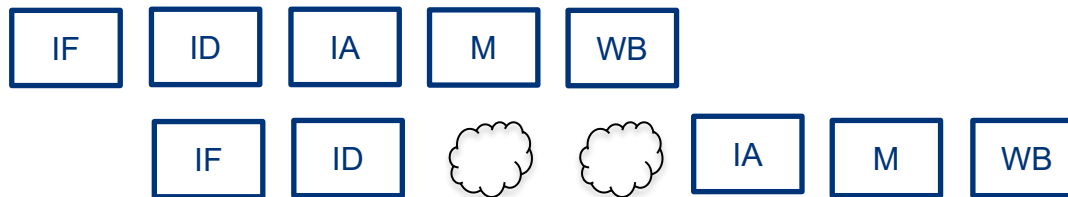
- **Special logic that checks if any instruction in later stages is supposed to write to any source register of the instruction that is being decoded**
 - **Yes**: stall the instruction/pipeline
 - **No**: no need to stall... no flow dependence
- **Advantage:**
 - No need to stall on anti and output dependences
- **Disadvantage:**
 - Logic is more complex than a scoreboard
 - Logic becomes more complex as we make the pipeline deeper and wider (superscalar execution)

How to Detect Dependence?

- **Interlocking:** Detection of dependence between instructions in a pipelined processor to guarantee correct execution
- **Hardware-based interlocking**
 - Techniques at the hardware level to detect and resolve
- **Software-based interlocking**
 - Compiler guarantees correctness by inserting no-ops or independent instructions between dependent instructions
 - Static detection

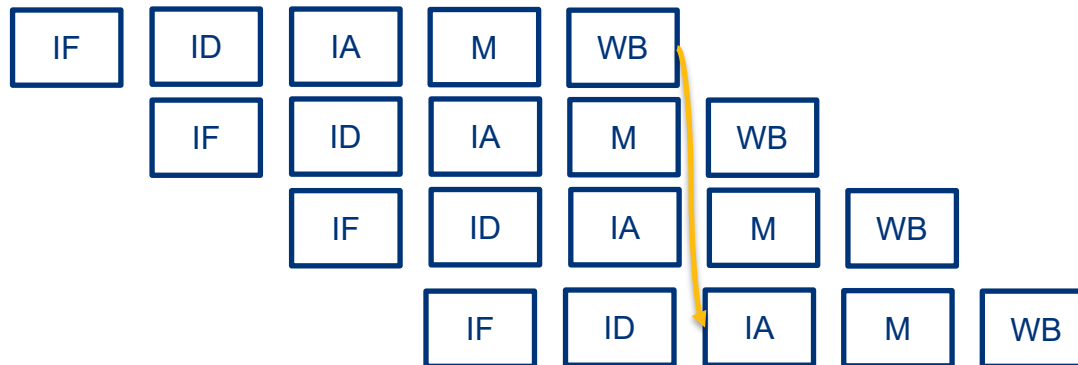
Software Dependence Detection

- **Software based scheduling of instructions → static scheduling**
 - Compiler orders the instructions, hardware executes them in that order



Lw R1 \leftarrow 0x20 (R2)
iAdd R3 \leftarrow R1 + R4

Remember: Once bubbles are introduced, they stay for ever



Lw R1 \leftarrow 0x20 (R2)
iadd....
isub...
iAdd R3 \leftarrow R1 + R4

Software Dependence Detection

- **Software based scheduling of instructions → static scheduling**
 - Compiler orders the instructions, hardware executes them in that order
 - How does the compiler know the latency of each instruction?
- **What information does the compiler not know that makes static scheduling difficult?**
 - Answer: Anything that is determined at run time
 - Variable-length operation latency, memory addr, branch direction
- **How can the compiler alleviate this (i.e., estimate the unknown)?**
 - Answer: Profiling

Data Dependence

- How to detect dependence?
- How to handle dependence?

How to Handle Data Dependences

- **Five fundamental ways of handling flow dependences**
 - **Detect and wait** until value is available in register file
 - **Detect and forward/bypass** data to dependent instruction
 - **Detect and eliminate** the dependence at the software level
 - No need for the hardware to detect dependence
 - **Predict** the needed value(s), execute “speculatively”, **and verify**
 - **Do something else** (fine-grained multithreading)
 - No need to detect

How to Handle Data Dependences

- **Five fundamental ways of handling flow dependences**
 - Detect and wait until value is available in register file
 - Detect and forward/bypass data to dependent instruction
 - Detect and eliminate the dependence at the software level
 - No need for the hardware to detect dependence
 - Predict the needed value(s), execute “speculatively”, and verify
 - Do something else (fine-grained multithreading)
 - No need to detect
- **Basically stall and insert bubble in the pipeline**

How to Stall the Pipeline?

- **Force control values in ID/EX register to 0**
 - EX, MEM and WB do nop (no-operation)
- **Prevent update of PC and IF/ID register**

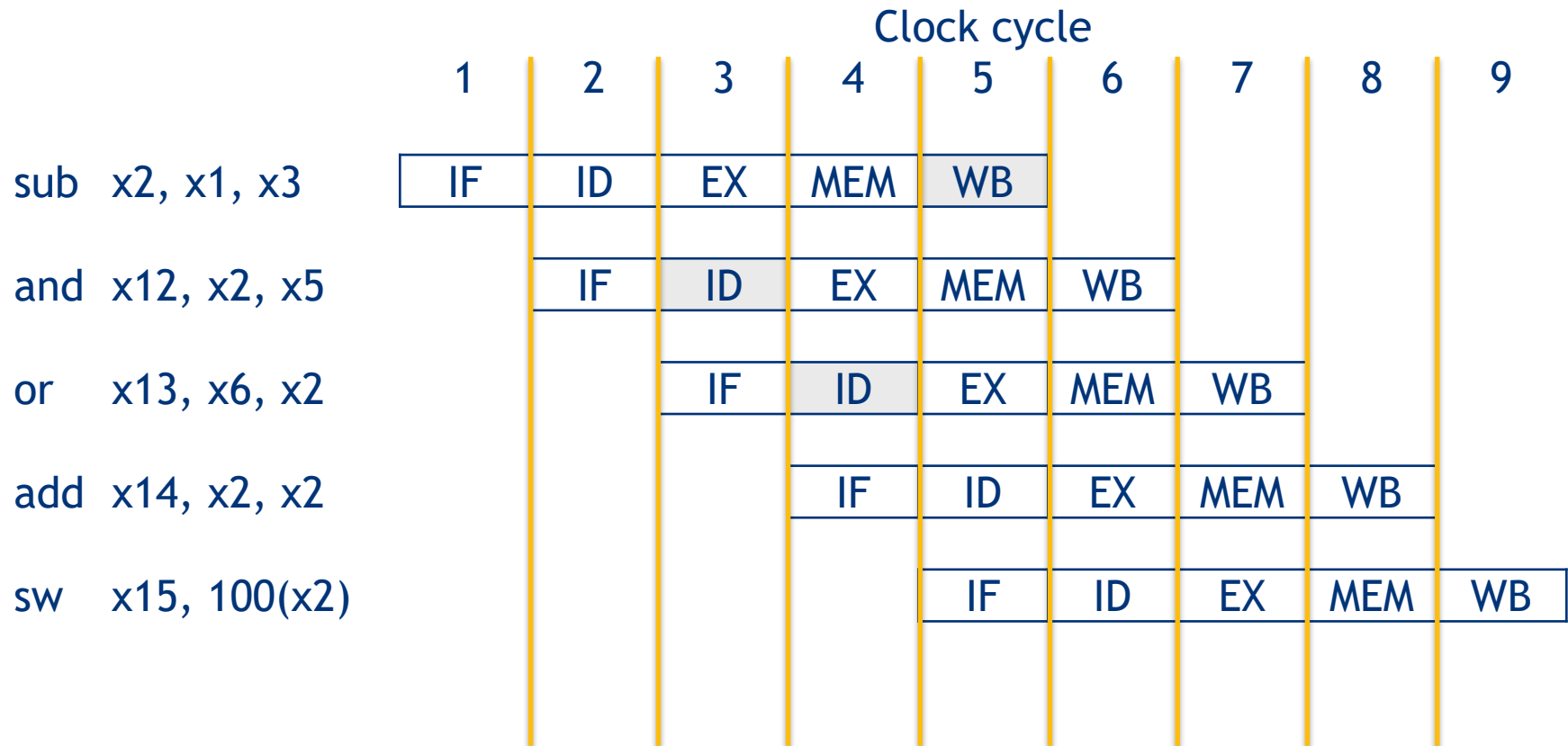
How to Handle Data Dependences

- **Five fundamental ways of handling flow dependences**
 - Detect and wait until value is available in register file
 - **Detect and forward/bypass** data to dependent instruction
 - Detect and eliminate the dependence at the software level
 - No need for the hardware to detect dependence
 - Predict the needed value(s), execute “speculatively”, and verify
 - Do something else (fine-grained multithreading)
 - No need to detect

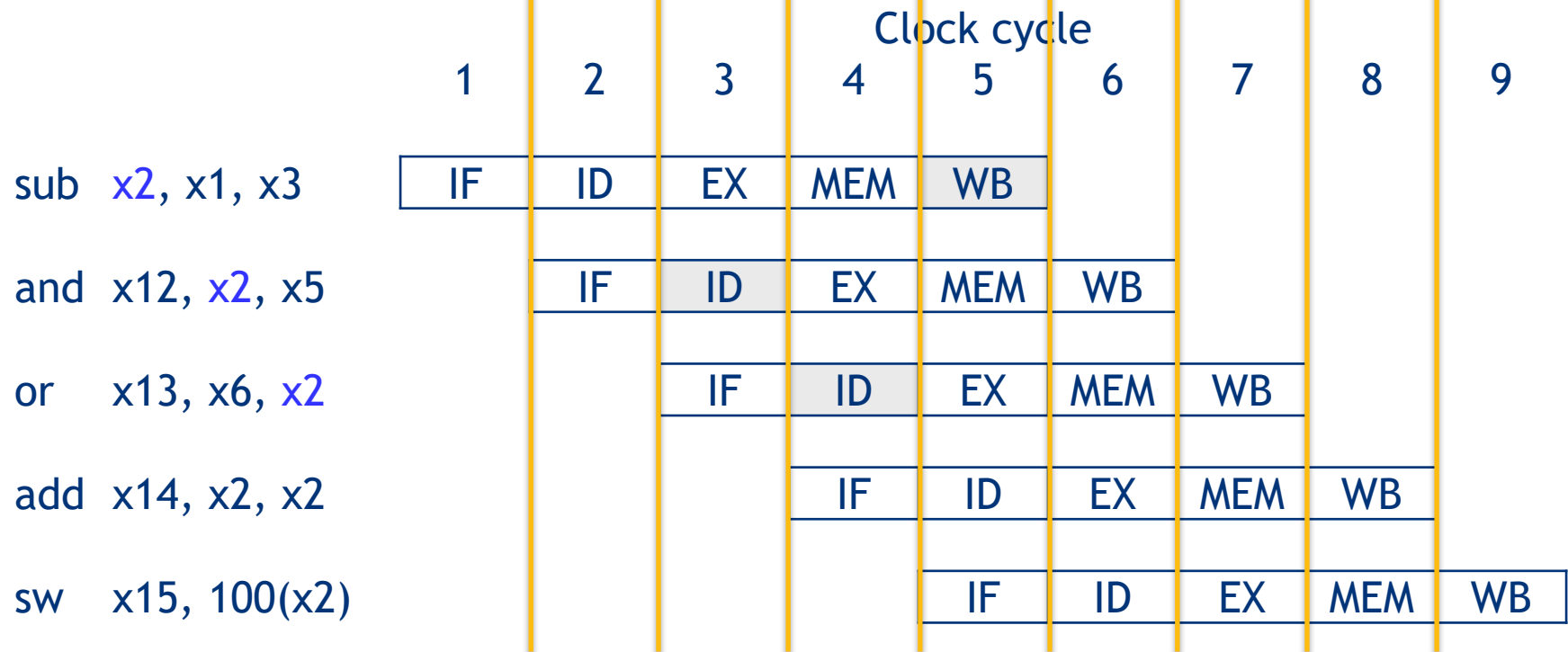
Data Forwarding/Bypassing

- **Problem:** A consumer (dependent) instruction has to wait in decode stage until the producer instruction writes its value in the register file
- **Goal:** We do not want to stall the pipeline unnecessarily
- **Observation:** The data value needed by the consumer instruction can be supplied directly from a later stage in the pipeline (instead of only from the register file)
- **Idea:** Add additional dependence check logic and data forwarding paths (buses) to supply the producer's value to the consumer right after the value is available
- **Benefit:** Consumer can move in the pipeline until the point the value can be supplied → less stalling

An Example

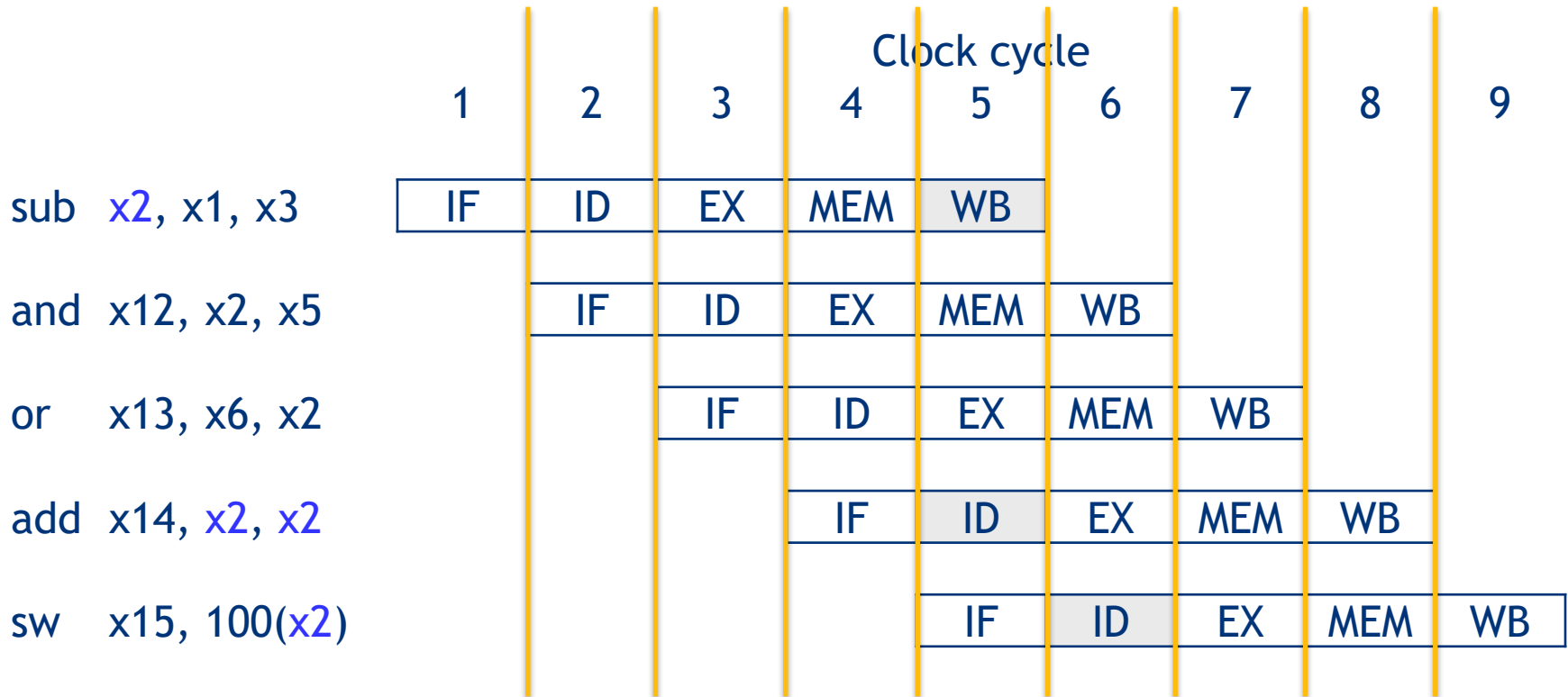


Instructions that have dependence



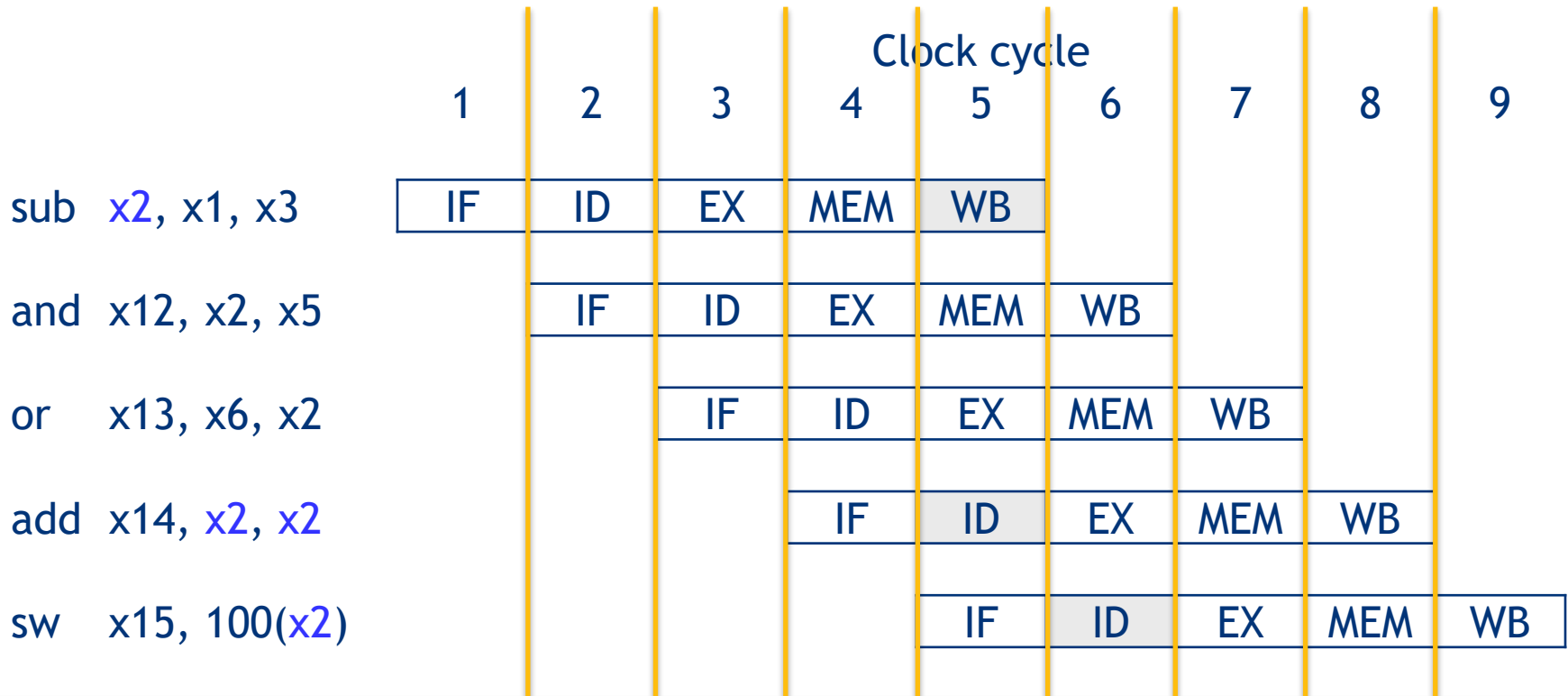
- The SUB instruction does not write to register \$2 until clock cycle 5. This causes two **data hazards** in our current pipelined datapath
 - The AND reads register \$2 in cycle 3. Since SUB hasn't modified the register yet, this will be the *old* value of \$2, not the new one.
 - Similarly, the OR instruction uses register \$2 in cycle 4, again before it's actually updated by SUB.

Instructions that are OK



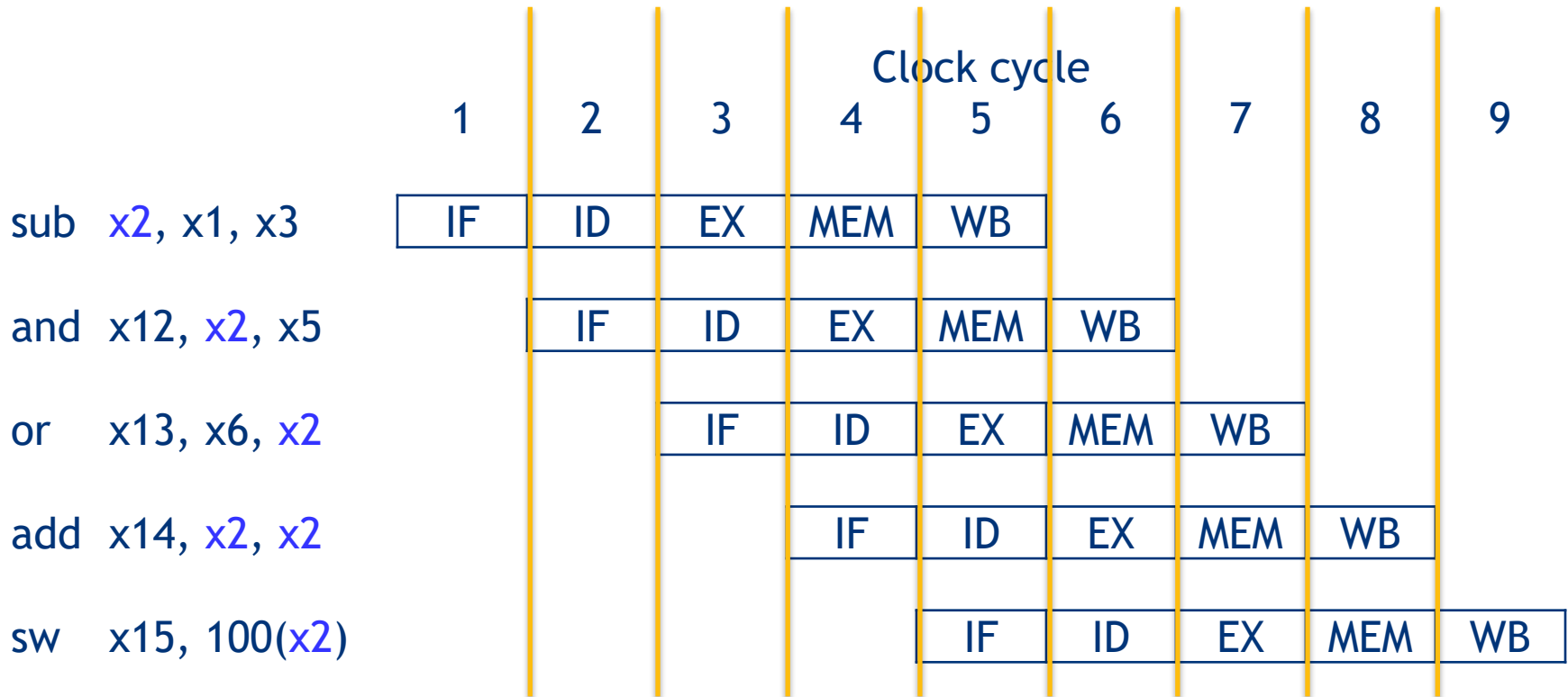
- **The ADD instruction is okay, because of the register file design.**
 - Registers are written at the beginning of a clock cycle.
 - The new value will be available by the end of that cycle.
- **The SW is no problem at all, since it reads \$2 after the SUB finishes.**

Instructions that are OK

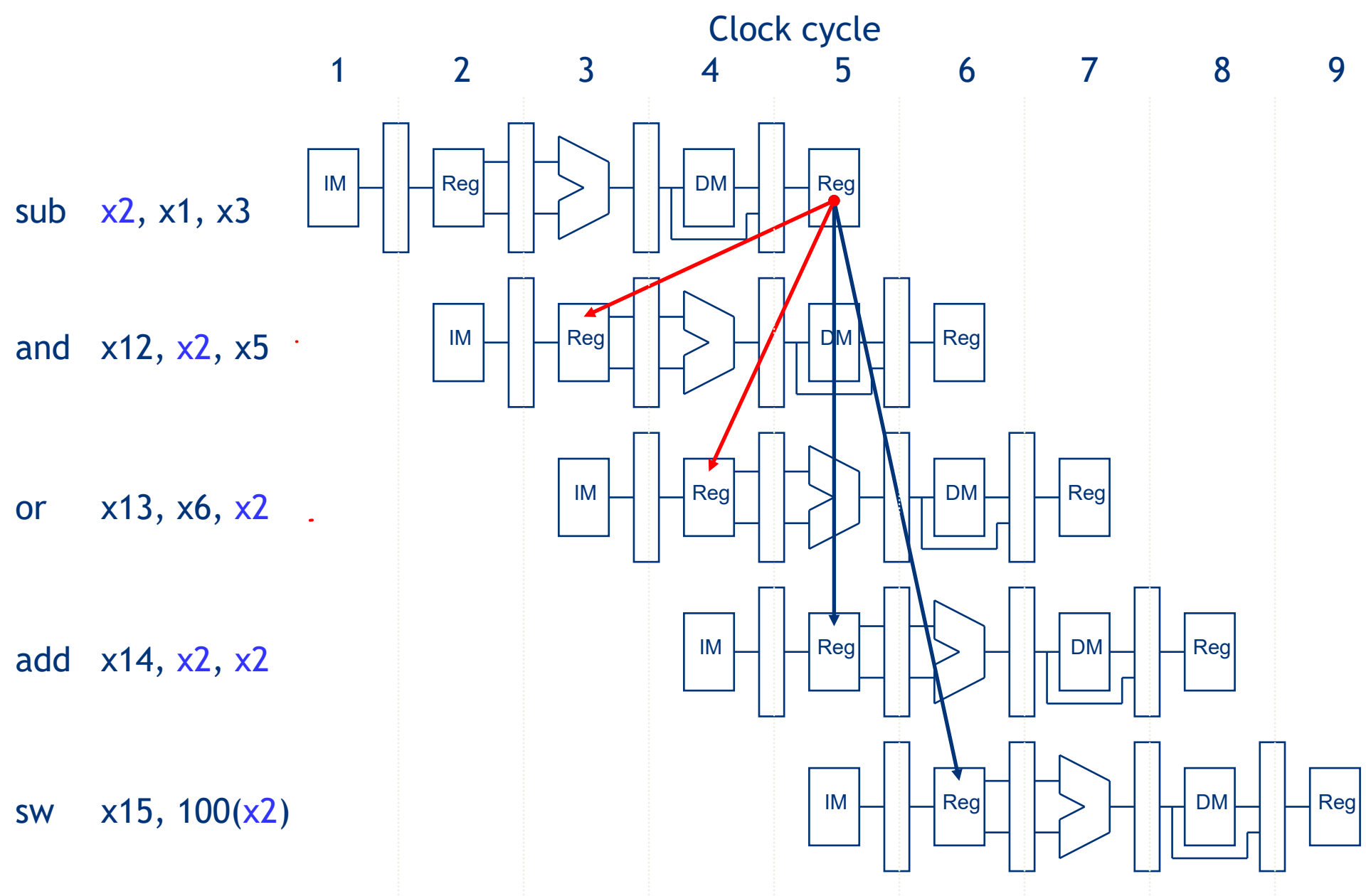


If two instructions have distance of 3 or more, there will not be any hazard

Instructions that are OK

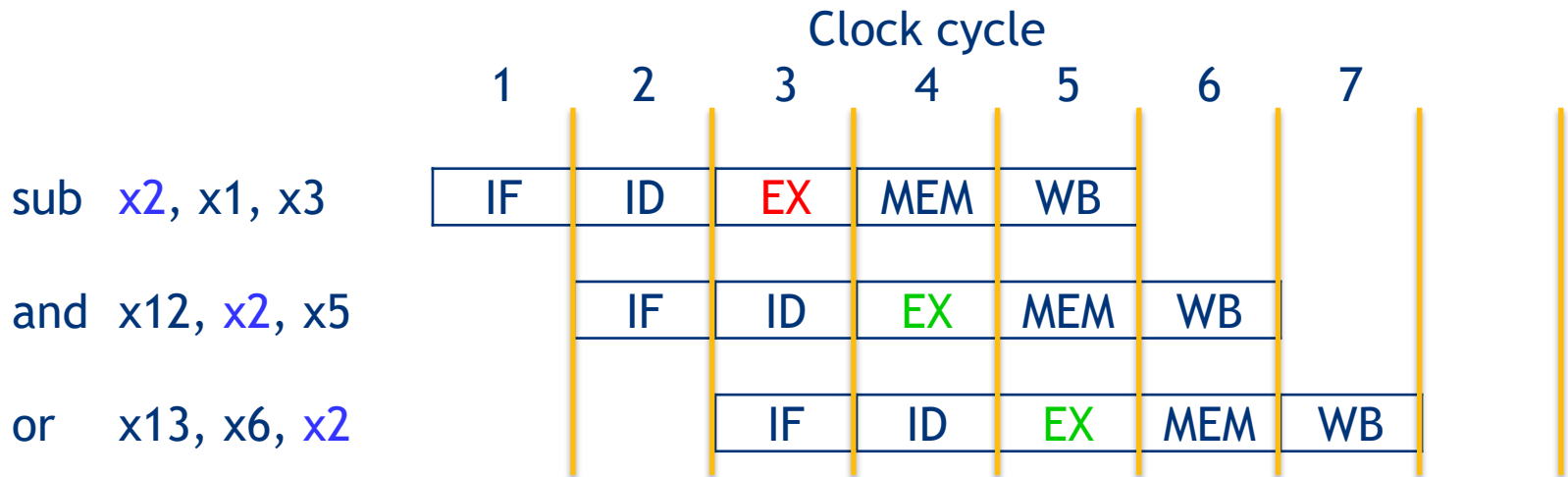


- **Arrows indicate the flow of data between instructions.**
 - The tails of the arrows show when register x2 is written.
 - The heads of the arrows show when x2 is read.
- **Any arrow that points backwards in time represents a **data hazard** in our basic pipelined datapath. Here, hazards exist between instructions 1 & 2 and 1 & 3.**



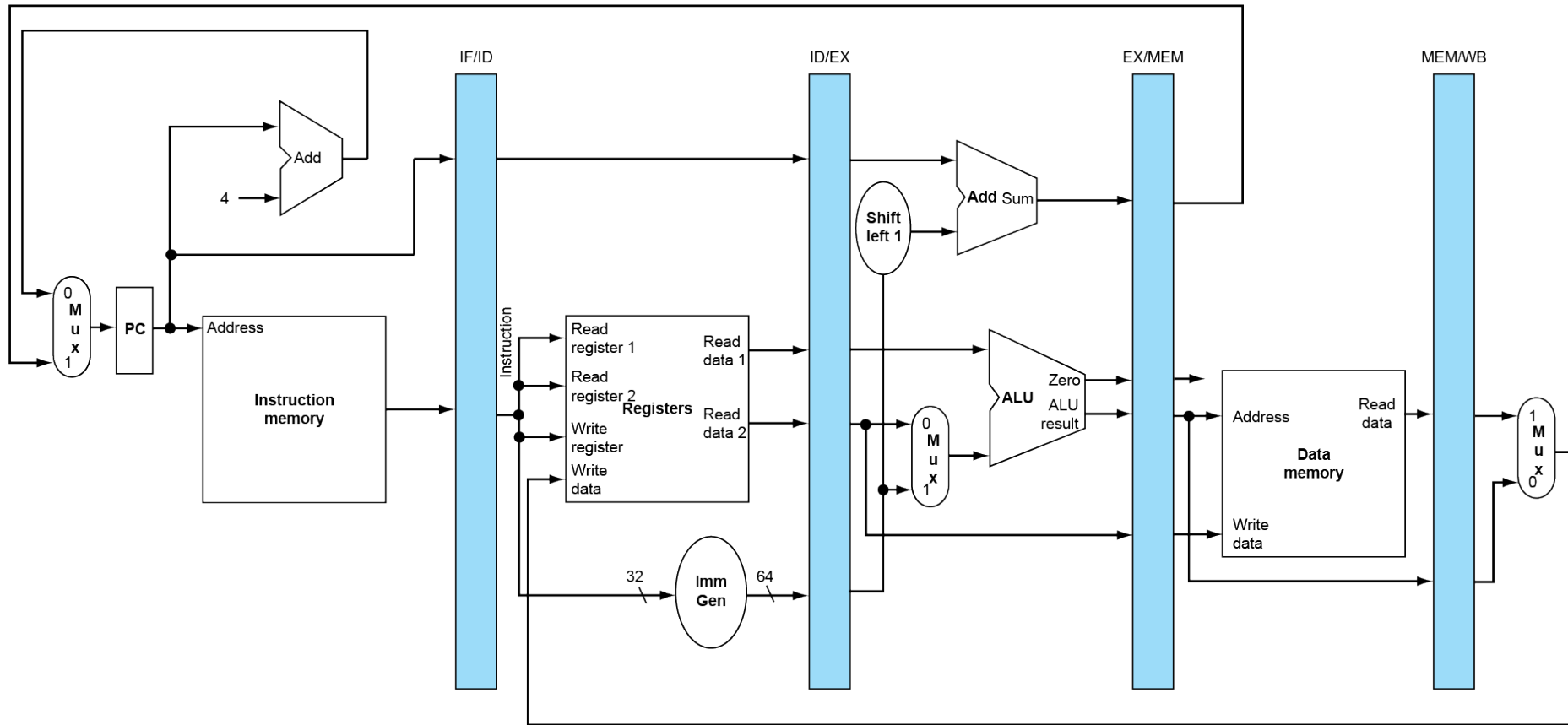
Analyze the problem scenario

- **Why data dependence?**
 - x2 is updated at the end of clock 5 and x2 is needed at the beginning of clock 4 and clock 5 for the “and” and “or” instructions respectively
- **Where is the value for x2 actually generated**
 - At the end of “Ex” stage of the “sub” instruction
 - At the end of clock 3

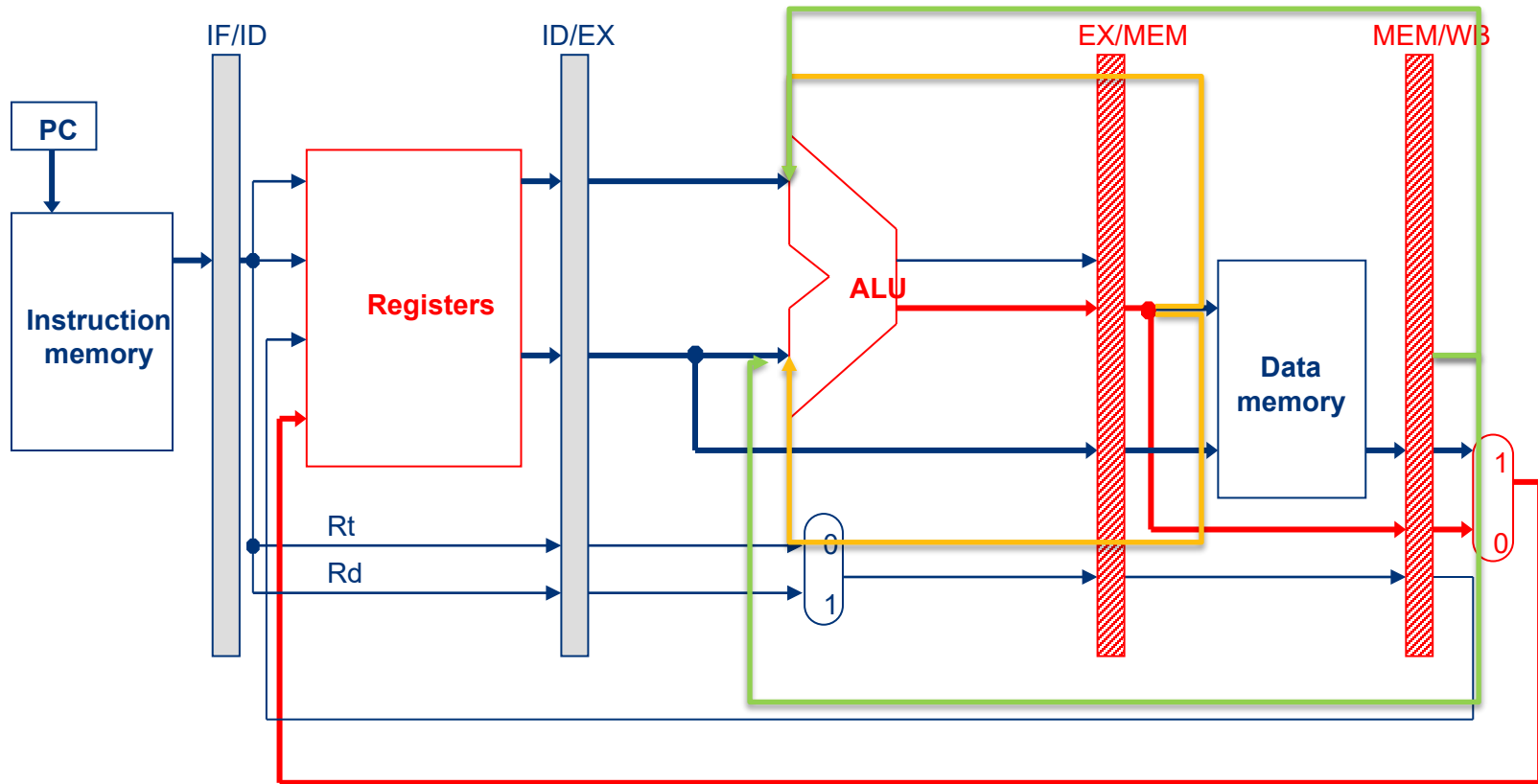




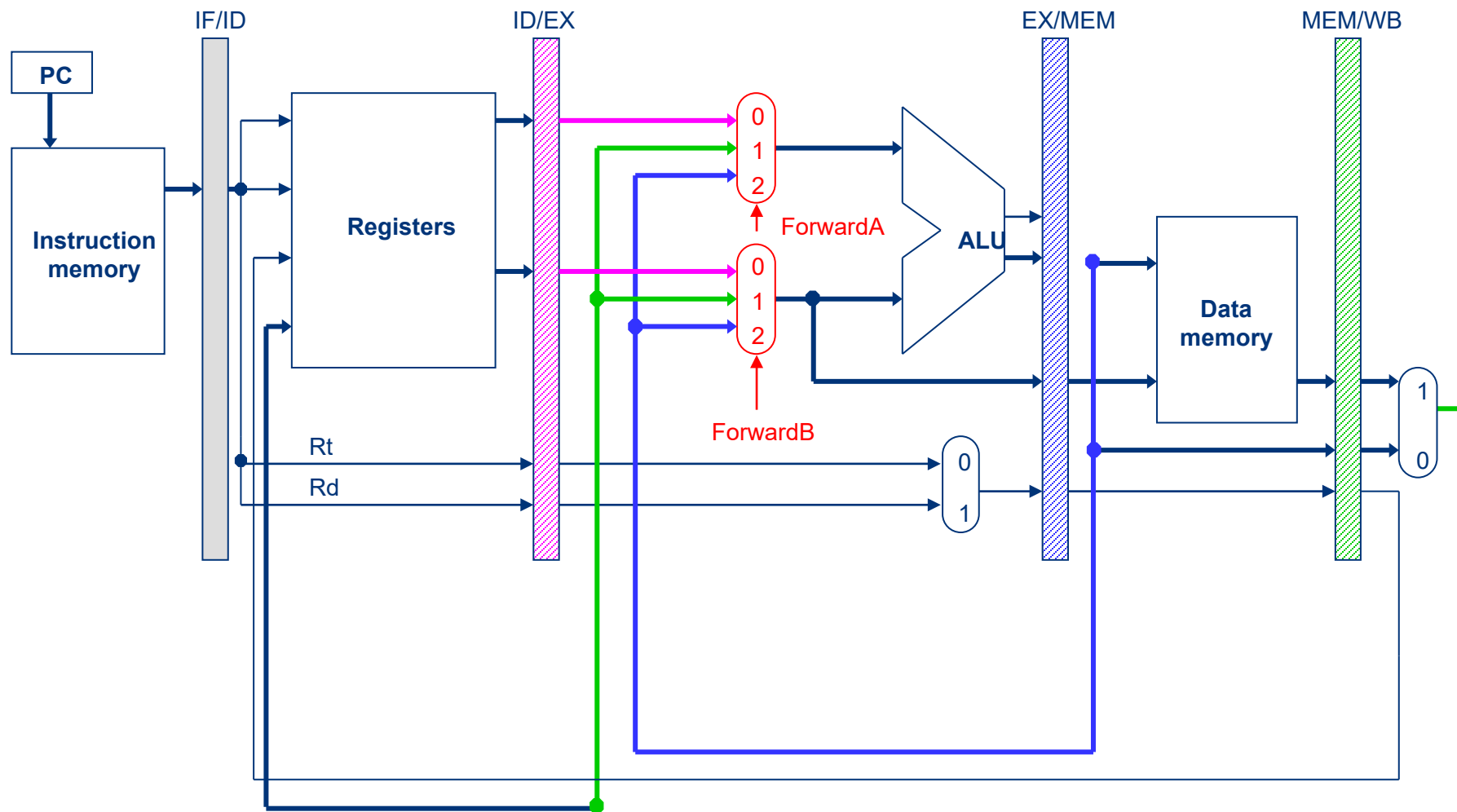
Recap of the Pipeline



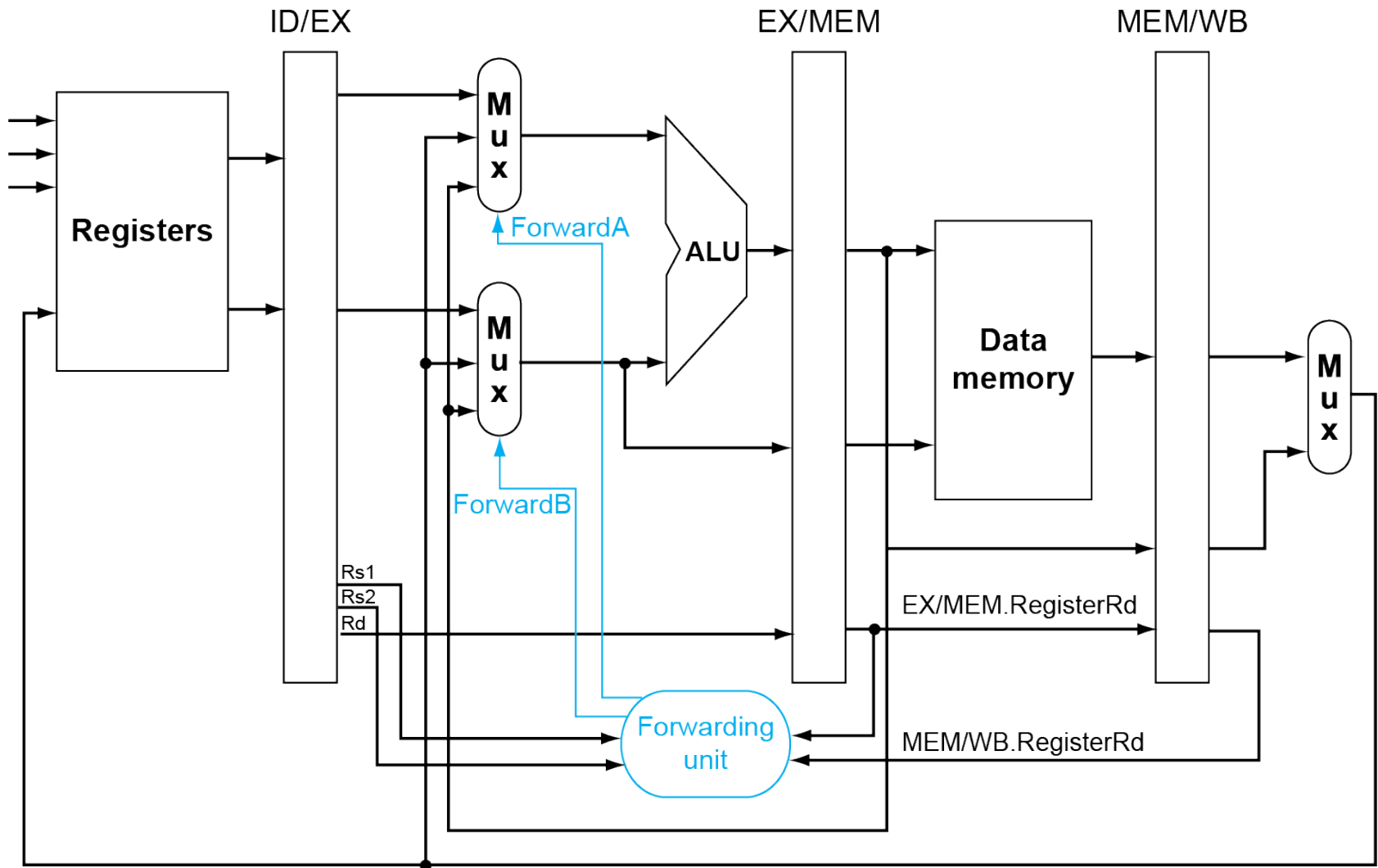
Finding the data



Forwarding Hardware



Forwarding Unit



Forwarding Conditions

Mux control	Source	Explanation
ForwardA = 00	ID/EX	The first ALU operand comes from the register file.
ForwardA = 10	EX/MEM	The first ALU operand is forwarded from the prior ALU result.
ForwardA = 01	MEM/WB	The first ALU operand is forwarded from data memory or an earlier ALU result.
ForwardB = 00	ID/EX	The second ALU operand comes from the register file.
ForwardB = 10	EX/MEM	The second ALU operand is forwarded from the prior ALU result.
ForwardB = 01	MEM/WB	The second ALU operand is forwarded from data memory or an earlier ALU result.

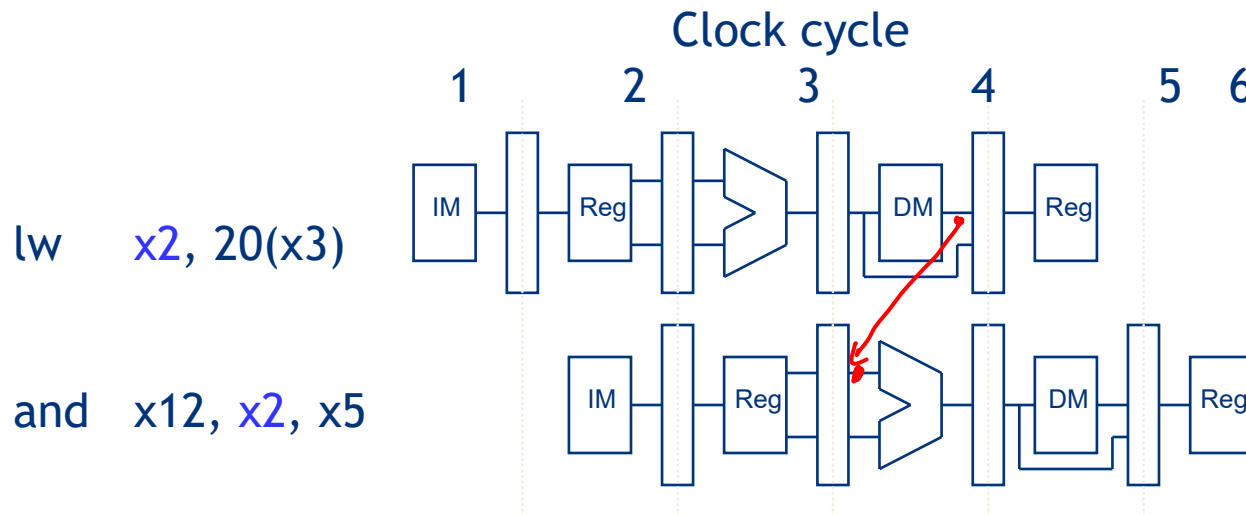
RAW Dependence Analysis

	R/I-Type	LW	SW	Br	J	Jr
IF						
ID	read RF	read RF	read RF	read RF		read RF
EX						
MEM						
WB	write RF	write RF				

- Instructions I_A and I_B (where I_A comes before I_B) have **RAW dependence** iff
 - I_B (R/I, LW, SW, Br or JR) reads a register written by I_A (R/I or LW)
 - $\text{dist}(I_A, I_B) \leq \text{dist}(\text{ID}, \text{WB}) = 3$

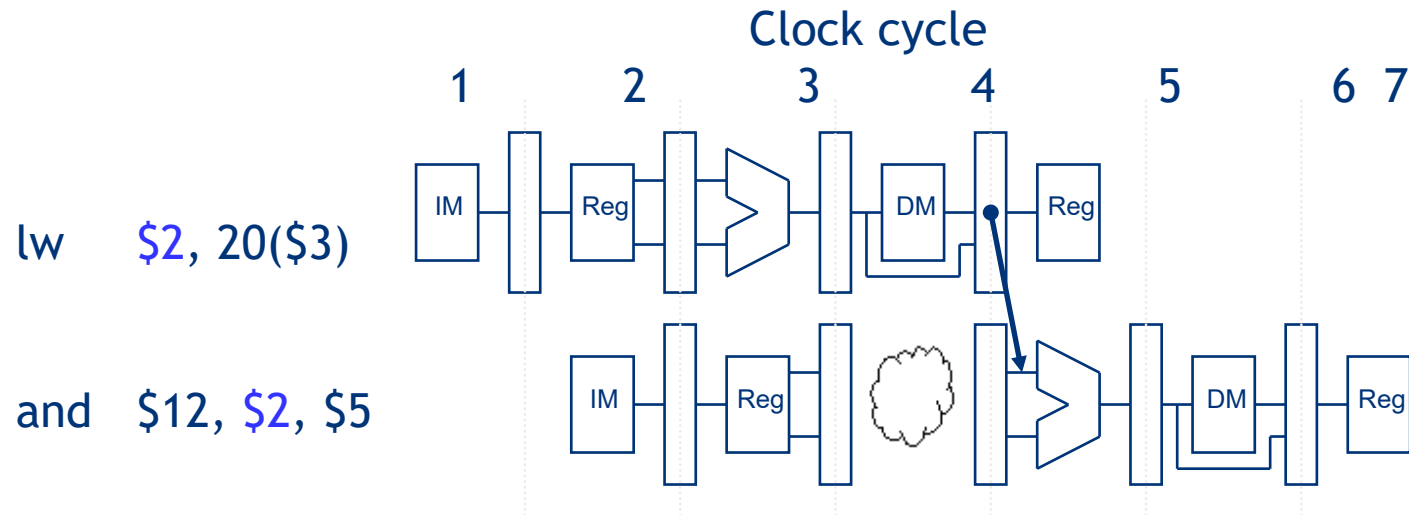
Case with loads

- Imagine if the first instruction in the example was LW instead of SUB.
 - How does this change the data dependence?
 - The load data doesn't come from memory until the *end* of cycle 4.
 - But the AND needs that value at the *beginning* of the same cycle!



Case with Load

- The easiest solution is to **stall** the pipeline.
- We could delay the AND instruction by introducing a one-cycle delay into the pipeline, sometimes called a **bubble**.



- Notice that we're still using forwarding in cycle 5, to get data from the MEM/WB pipeline register to the ALU.

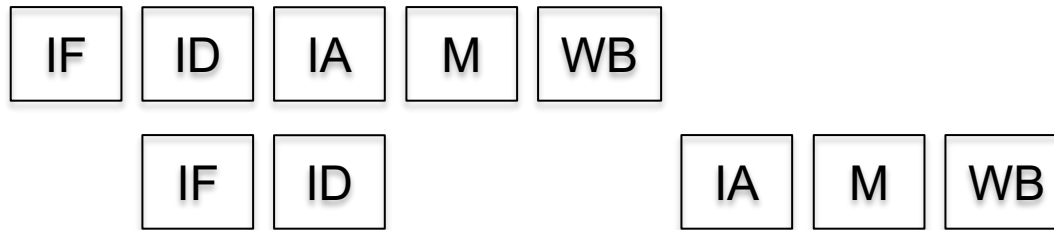
Impact of Stall on Performance

- Each stall cycle corresponds to **one lost cycle** in which no instruction can be completed
- For a program with N instructions and S stall cycles,
Average $CPI = (N + S) / N$
- **S depends on**
 - frequency of RAW dependences
 - exact distance between the dependent instructions
 - distance between dependences

suppose i_1, i_2 and i_3 all depend on i_0 , once i_1 's dependence is resolved, i_2 and i_3 must be okay too

Quick Recap: RAW

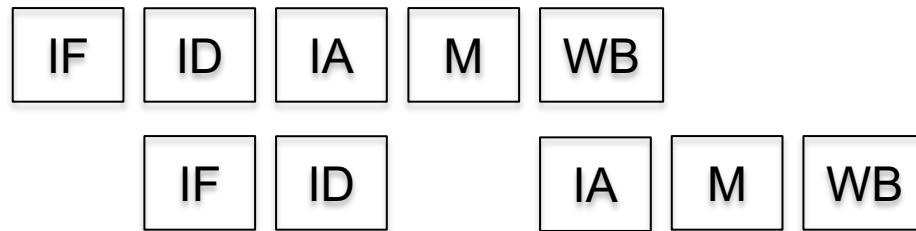
- Load **x3**, Mem(0x200)
- **x4** = **x3** + x5



Without Forwarding

Quick Recap: RAW

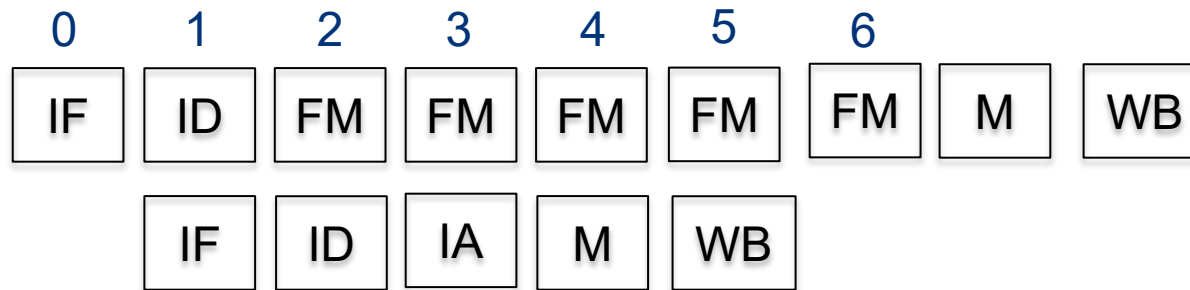
- Load **x3**, Mem(0x200)
- **x4** = **x3** + x5



Forwarding

Quick Recap: WAR

- $x3 = x1 \text{ (f*) } x2$
- $x1 = x4 + x5$

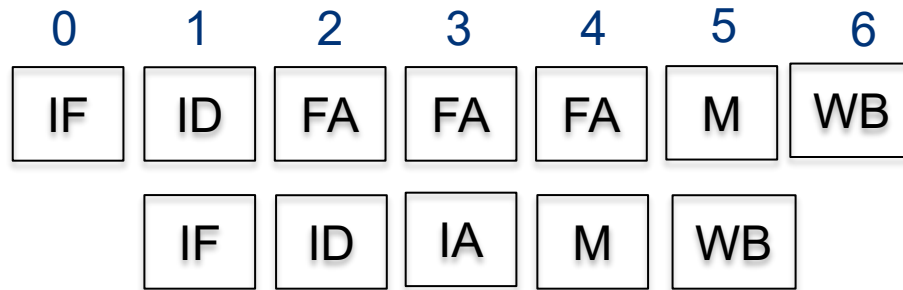


Observe that $x1$ is overwritten at the end of clock 5

Observe that $x1$ may be needed at the beginning of clock 6

Quick Recap: WAW

- $x3 = x1 \text{ (f+)} x2$
- $x3 = x4 + x5$



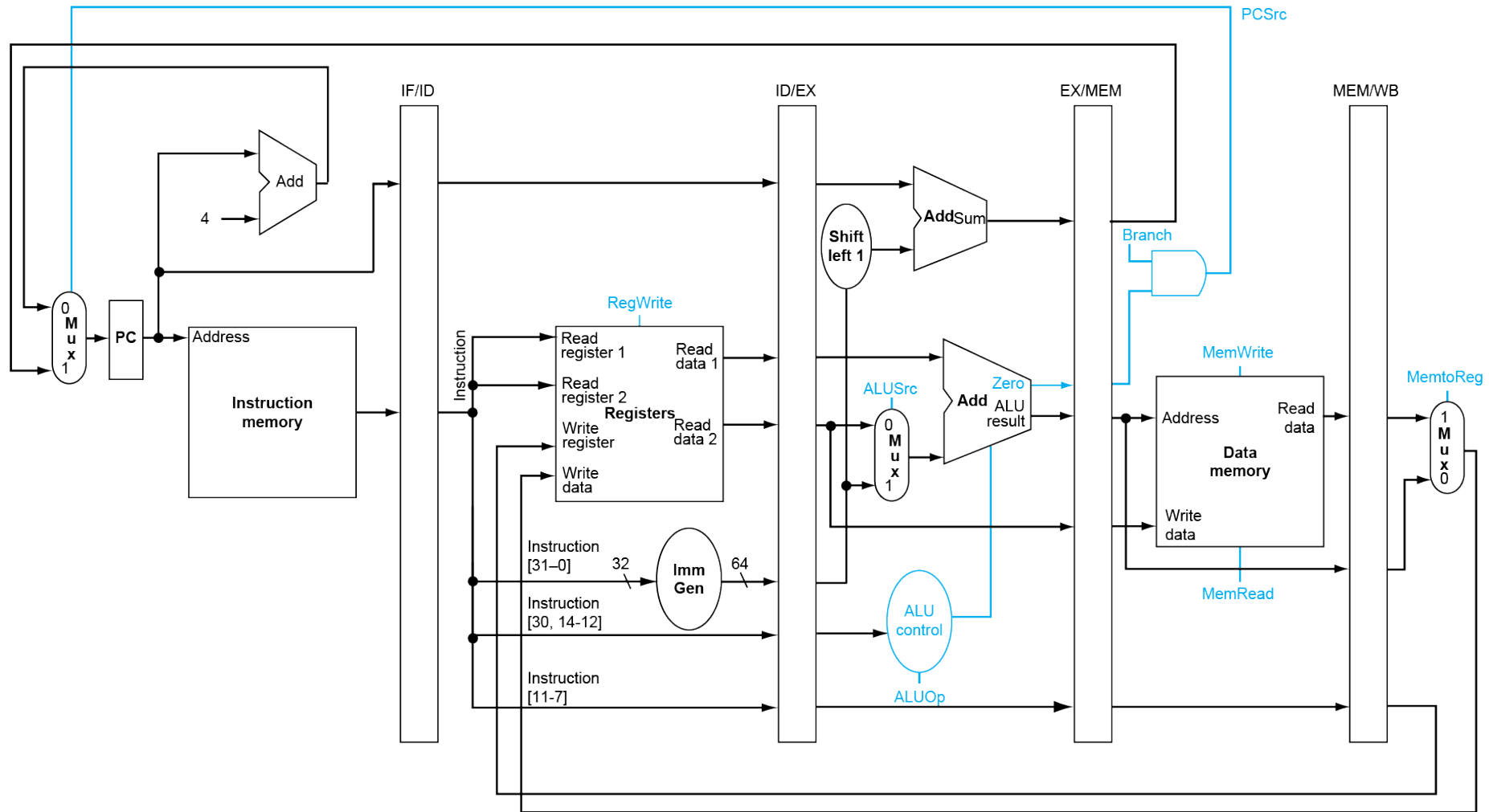
Observe that \$3 is written back at the end of clock 5 from inst 2

Observe that \$3 is written back at the end of clock 6 from inst 1

Hazards

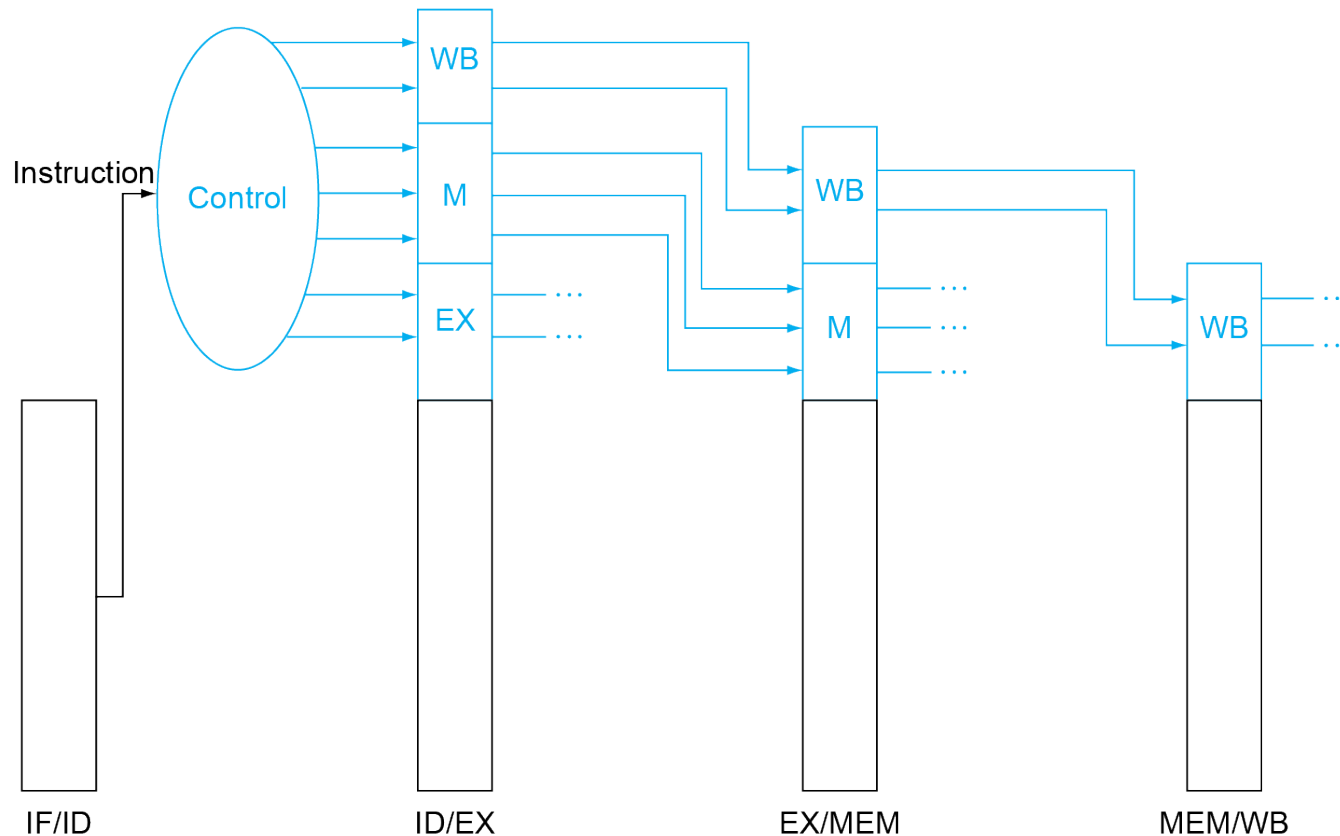
- **Situations that prevent starting the next instruction in the next cycle**
- **Structure hazards**
 - A required resource is busy
- **Data hazard**
 - Need to wait for previous instruction to complete its data read/write
- **Control hazard**
 - Deciding on control action depends on previous instruction

Pipelined Control (Simplified)



Pipelined Control

- **Control signals derived from instruction**
 - As in single-cycle implementation



Pipelined Control

