RISC 5-stage Pipeline and Extension

Lecture 3
September 13th, 2023

Jae W. Lee (jaewlee@snu.ac.kr)
Computer Science and Engineering
Seoul National University

***Slide credits**: Some slides taken from UC Berkeley CS 152, MIT 6.823, and instructor's slides from Elsevier Inc.*

Review – CPU Performance

$$\text{CPU time} = \frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Cycle}}$$

- **Performance depends on**
  - Algorithm
  - Programming language
  - Compiler
  - Instruction set architecture (ISA)
  - Core organization (also called microarchitecture)
  - Fabrication technology

Software stack

- Architecture

Implementation

Review – Determinates of CPU Performance

- **CPU time = Instruction_count x CPI x clock_cycle**

	Instruction_ count	CPI	clock_cycle
Algorithm			
Programming language			
Compiler			
ISA			
Core organization			
Technology			

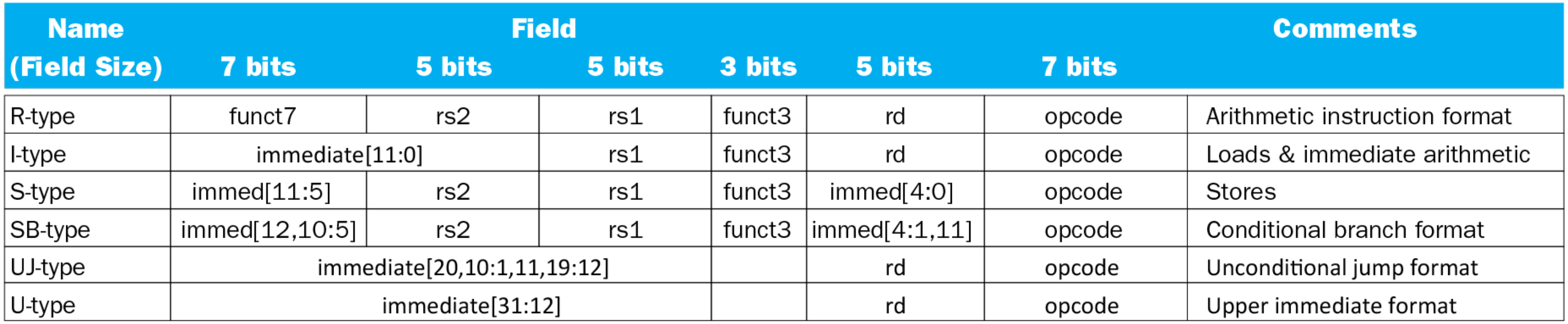
Review – Determinates of CPU Performance

- **CPU time = Instruction_count x CPI x clock_cycle**

	Instruction_ count	CPI	clock_cycle
Algorithm	X	X	
Programming language	X	X	
Compiler	X	X	
ISA	X	X	X
Core organization		X	X
Technology			X

Review – RISC-V ISA

- **32-bit fixed format instruction (6 formats)**



Name	Field						Comments
(Field Size)	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	
R-type	funct7	rs2	rs1	funct3	rd	opcode	Arithmetic instruction format
I-type	immediate[11:0]		rs1	funct3	rd	opcode	Loads & immediate arithmetic
S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode	Stores
SB-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode	Conditional branch format
UJ-type	immediate[20,10:1,11,19:12]				rd	opcode	Unconditional jump format
U-type	immediate[31:12]				rd	opcode	Upper immediate format

- **32 general-purpose registers (GPRs)**
  - x0 is hardwired to zero (read-only)
- **3-operand, reg-reg arithmetic instruction**
  - e.g., add `x28, x5, x6  # x28 = x5 + x6`
- **Single address mode for load/store: base + displacement**
  - no indirection

Outline

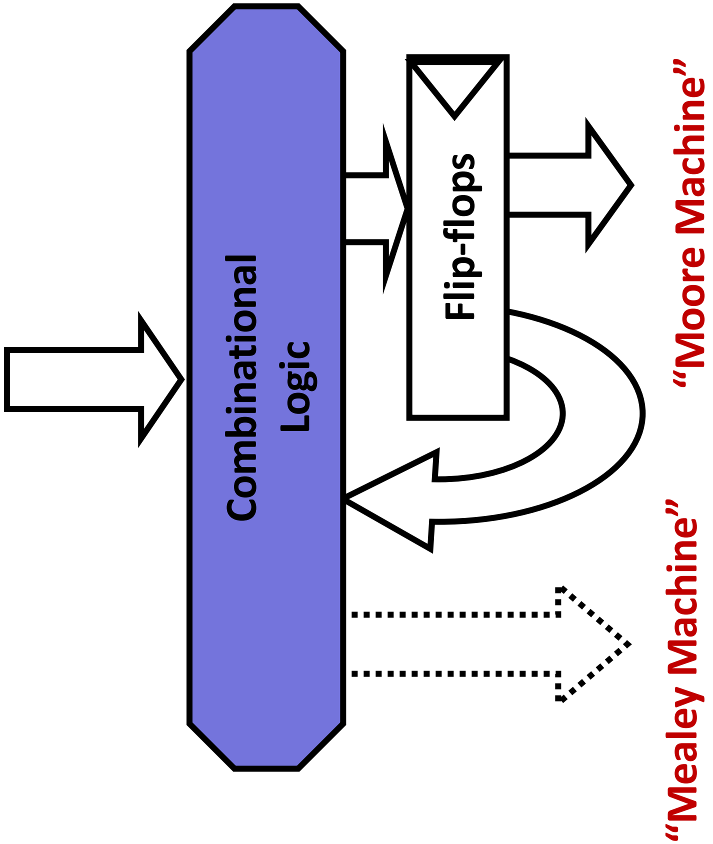
Textbook: Appendix C.1-C.5

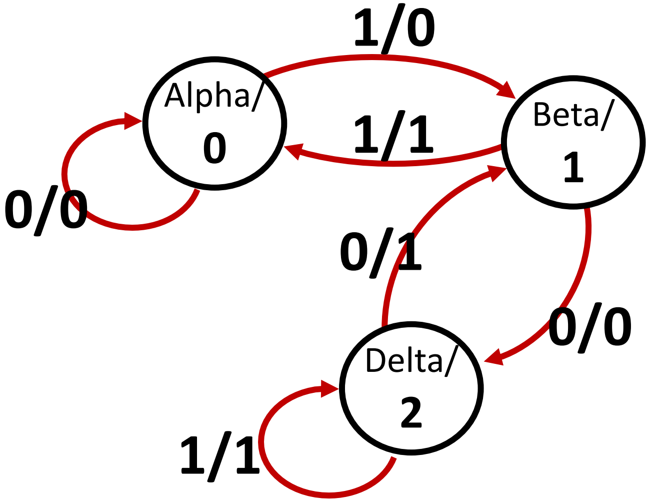
- **ISA Implementation Basics**
- **Classic 5-stage Pipeline for RISC: A First Shot**
- **Pipeline Hazards**
  - Structural Hazards
  - Data Hazards
  - Control Hazards
- **Exception/Interrupt Handling**
- **Handling Multi-Cycle Operations**

ISA Implementation Basics

Synchronous + digital

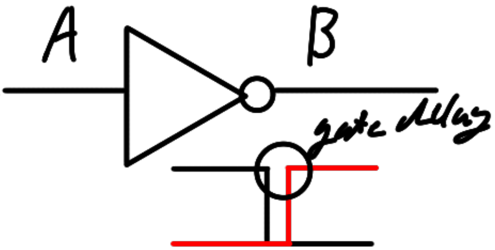
- Finite State Machines = Combinational logic + Flip-Flops



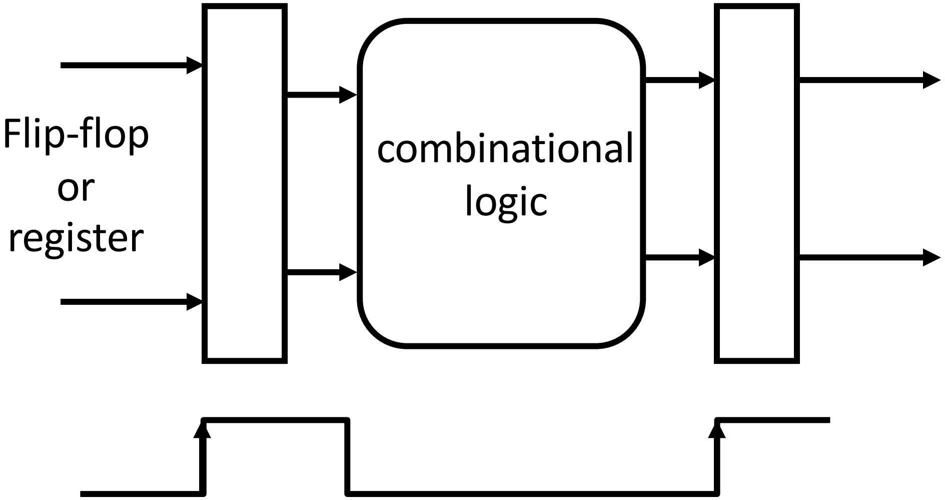


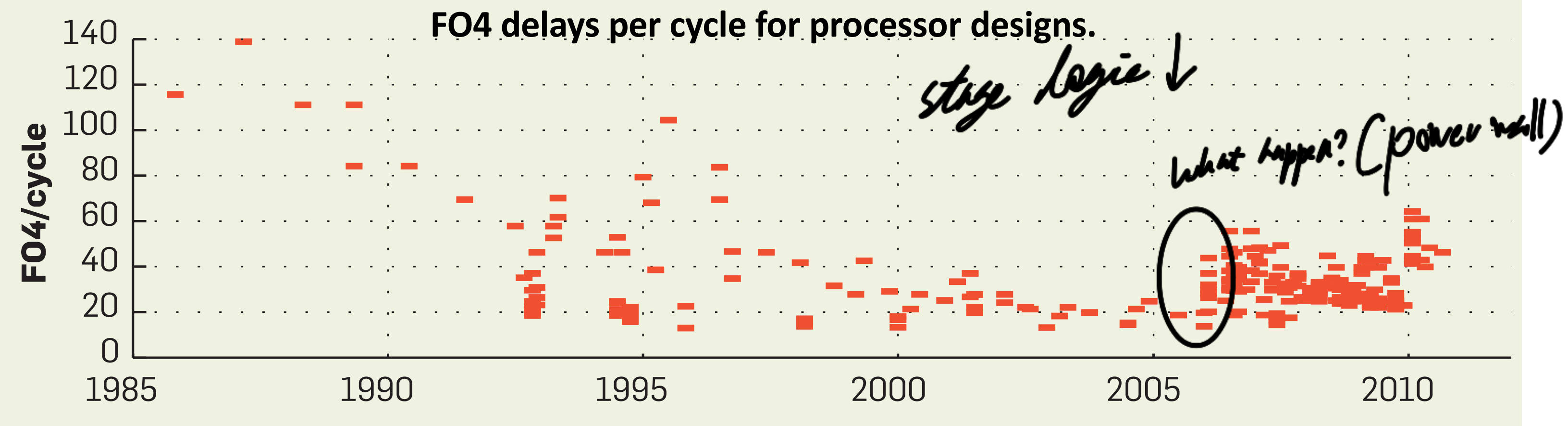
Input	$State_{old}$	$State_{new}$	Div
0	00	00	0
0	01	10	0
0	10	01	1
1	00	01	0
1	01	00	1
1	10	10	1

ISA Implementation Basics



- **What's a Clock Cycle?**
  - Old days: 10 levels of gates
  - Today: determined by numerous time-of-flight issues + gate delays + Wire
    - clock propagation, wire

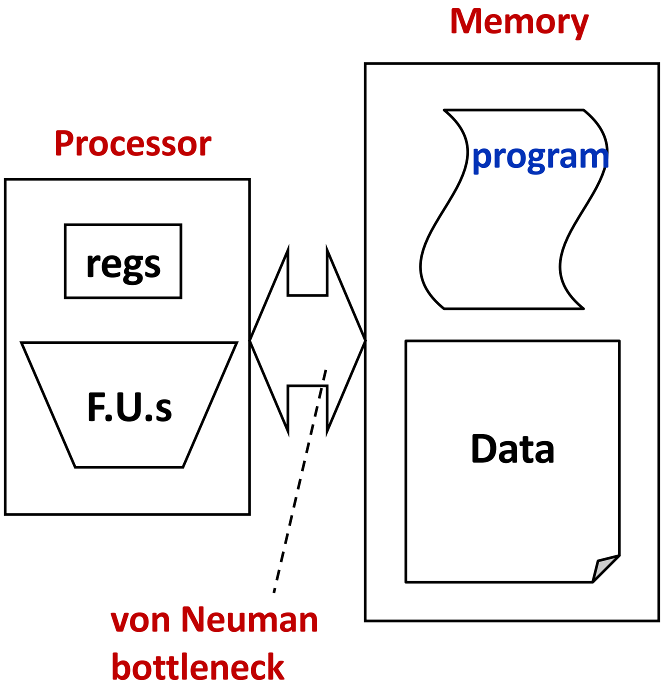


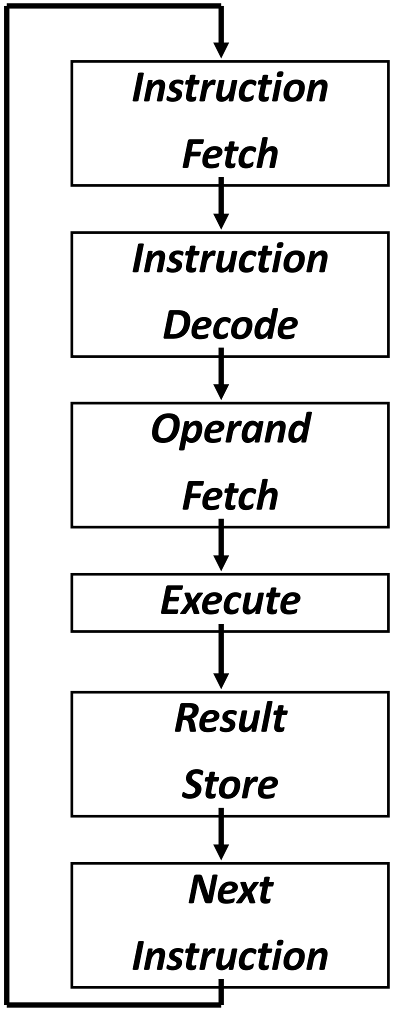


Source: A. Danowitz et al., CPU DB: Recording microprocessor history, ACM Queue, 2012.

ISA Implementation Basics

- Fundamental Execution Cycle





Instruction Fetch – Obtain instruction from program storage

Instruction Decode – Determine required actions and instruction size

Operand Fetch – Locate and obtain operand data

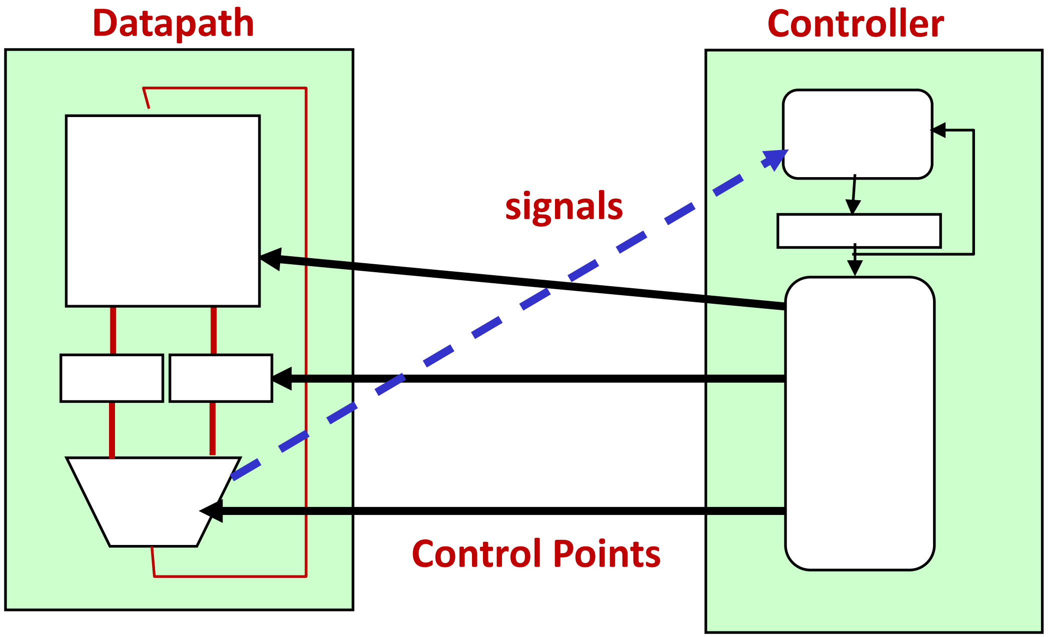
Execute – Compute result value or status

Result Store – Deposit results in storage for later use

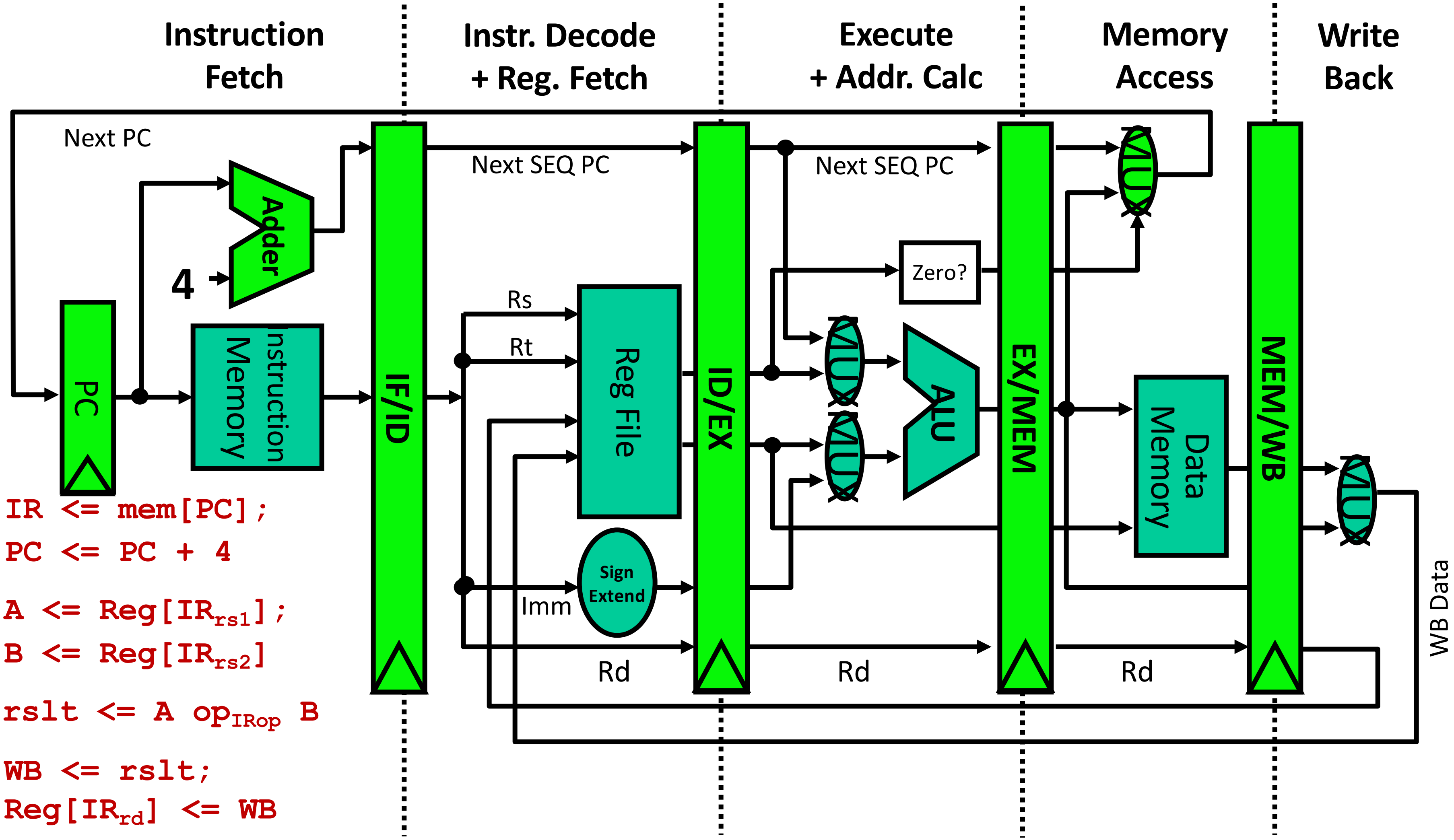
Next Instruction – Determine successor instruction

ISA Implementation Basics

- **Datapath vs Control**
  - Datapath: Storage, FU, interconnect sufficient to perform desired functions
    - Inputs are Control Points
    - Outputs are signals
  - Controller: State machine to orchestrate operation on data path
    - Based on desired function and signals



Classic 5-stage Pipeline for RISC: A First Shot



```
IR <= mem[PC];
PC <= PC + 4
```

$A \Leftarrow Reg[IR_{rs1}];$

$B \Leftarrow Reg[IR_{rs2}]$

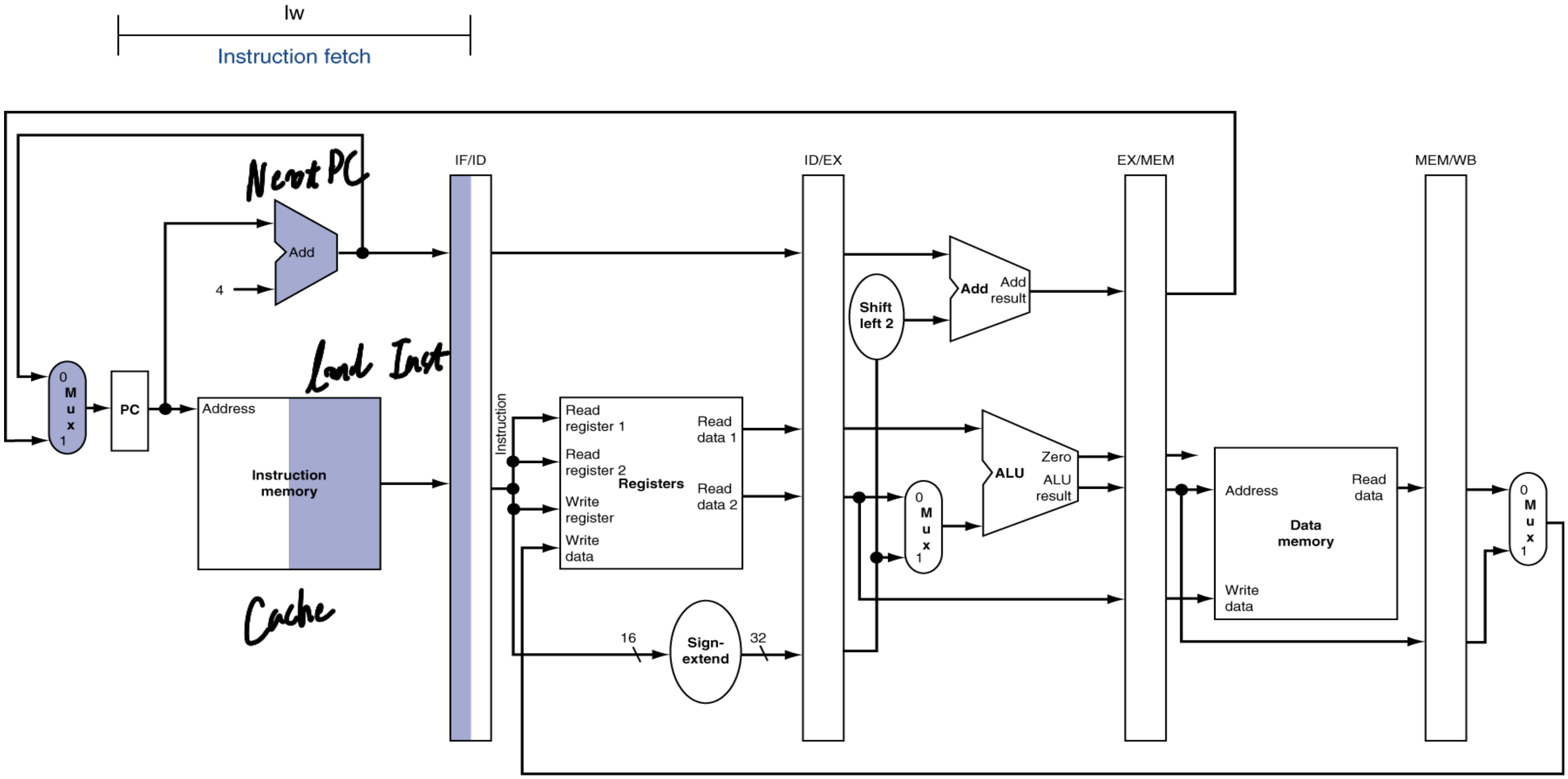
$rslt \Leftarrow A\ op_{IRop}\ B$

$WB \Leftarrow rslt;$

$Reg[IR_{rd}] \Leftarrow WB$

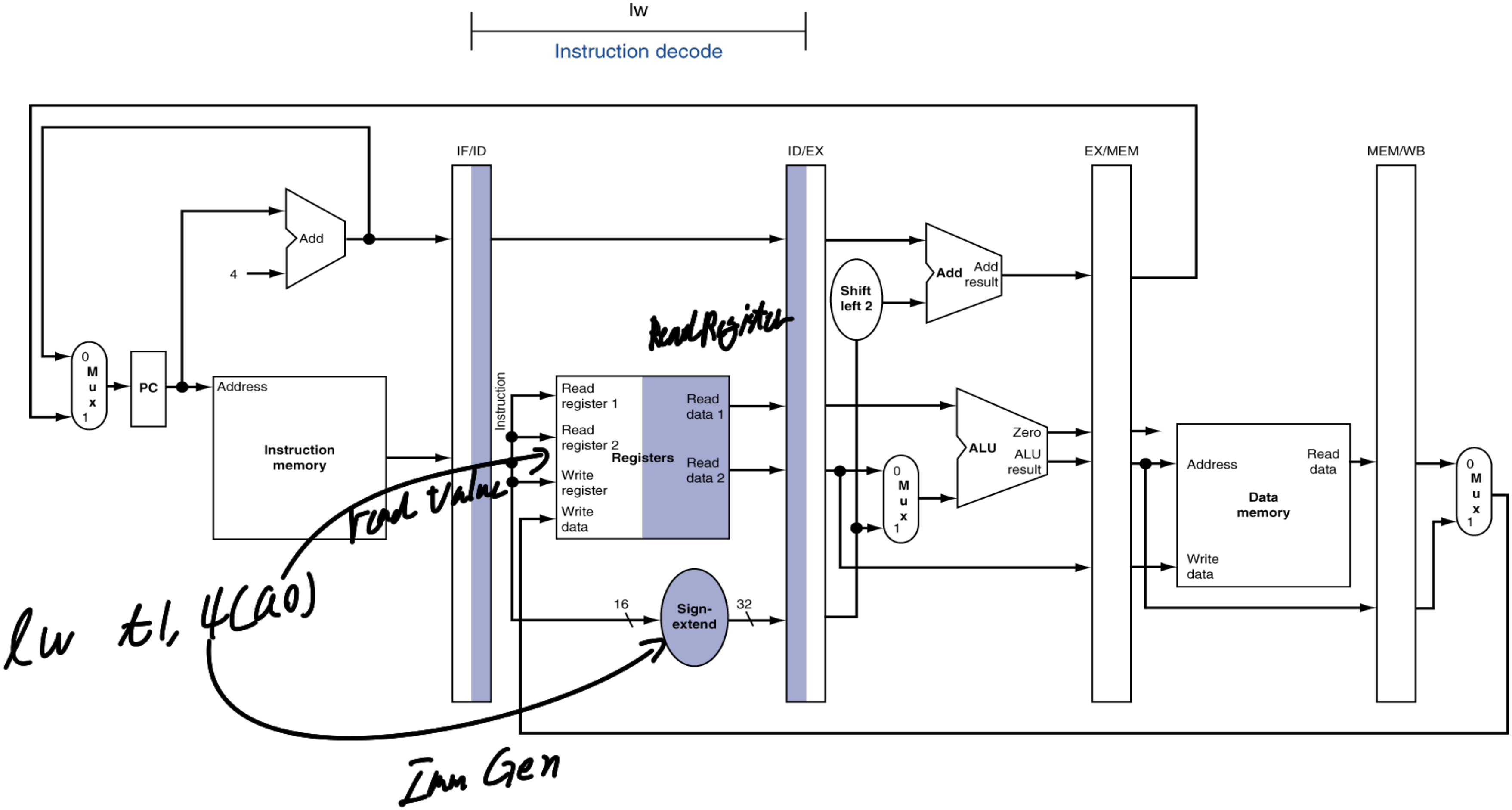
Classic 5-stage Pipeline for RISC: A First Shot

- **Example: executing load (lw) instruction – IF stage**



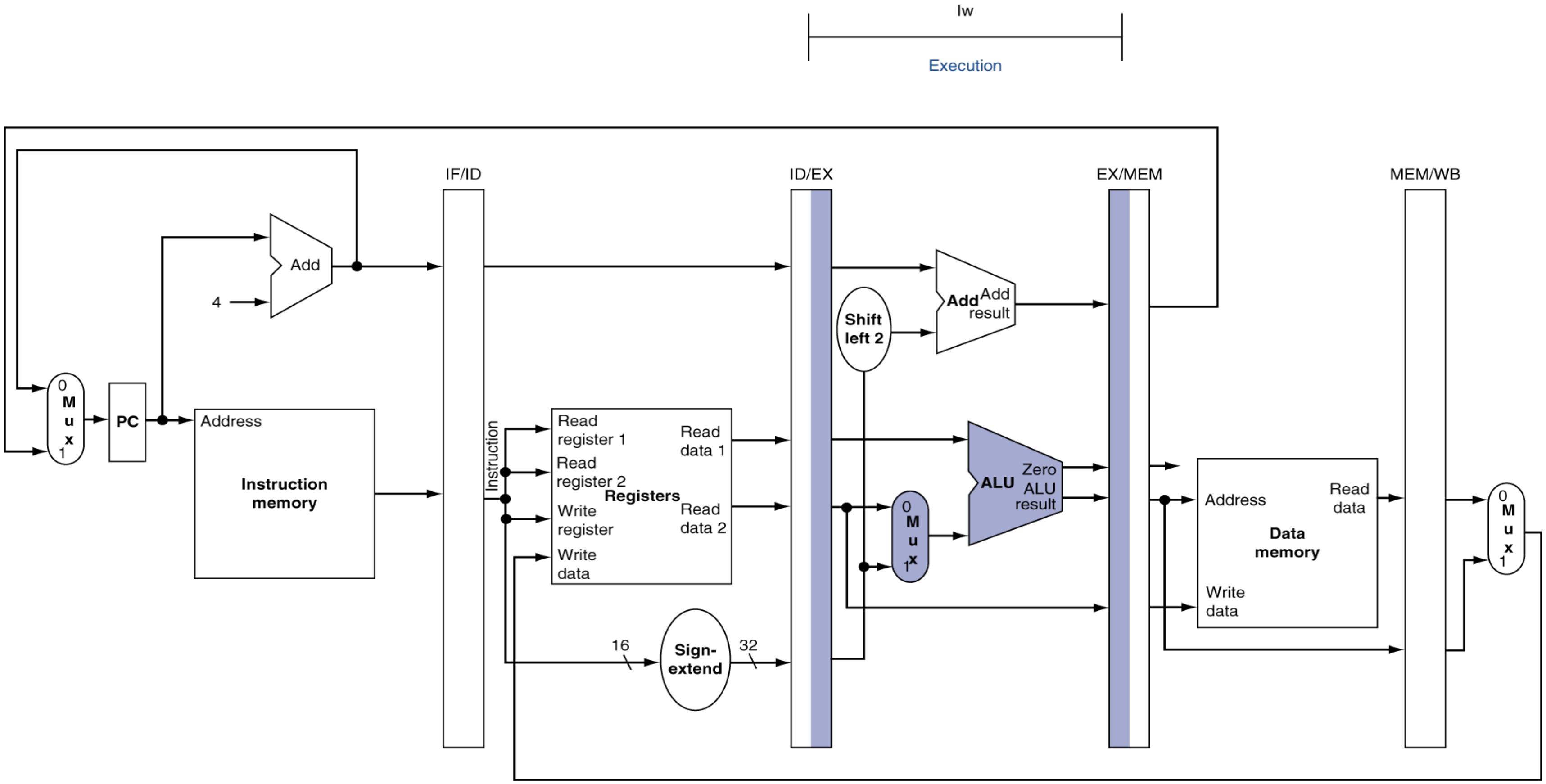
Classic 5-stage Pipeline for RISC: A First Shot

- **Example: executing load (lw) instruction – ID stage**



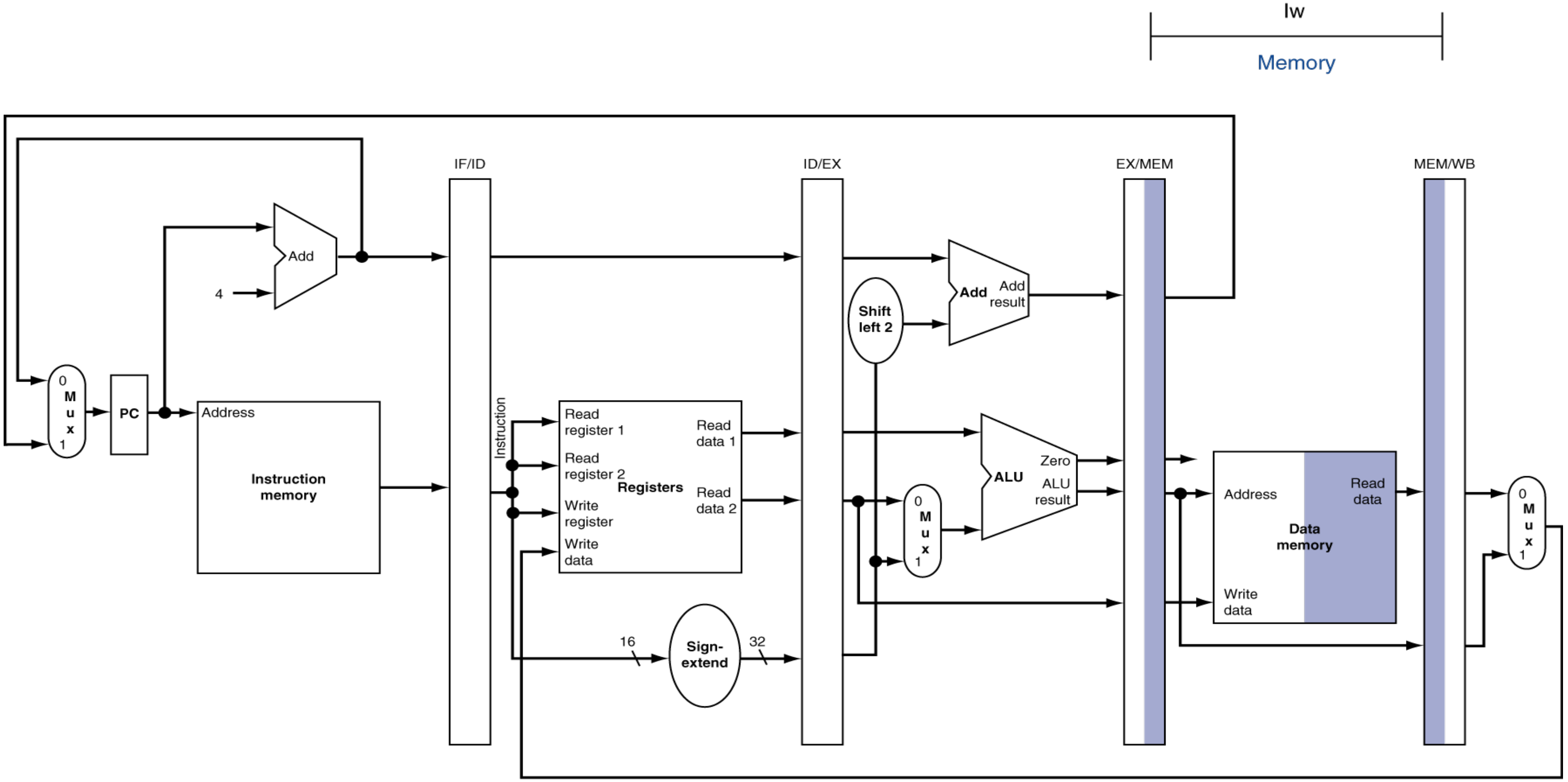
Classic 5-stage Pipeline for RISC: A First Shot

- **Example: executing load (lw) instruction – EX stage**



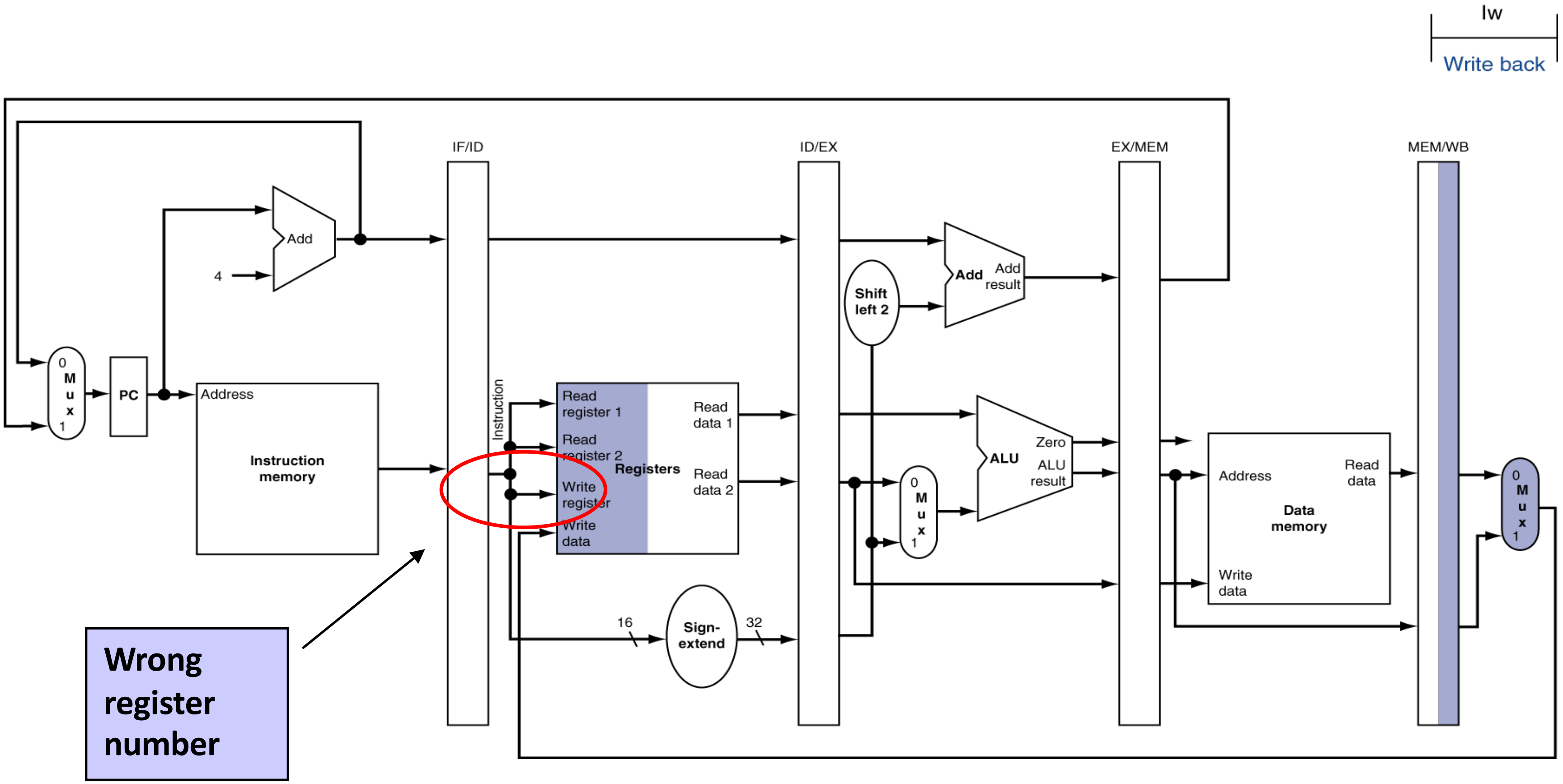
Classic 5-stage Pipeline for RISC: A First Shot

- Example: executing load (lw) instruction – MEM stage



Classic 5-stage Pipeline for RISC: A First Shot

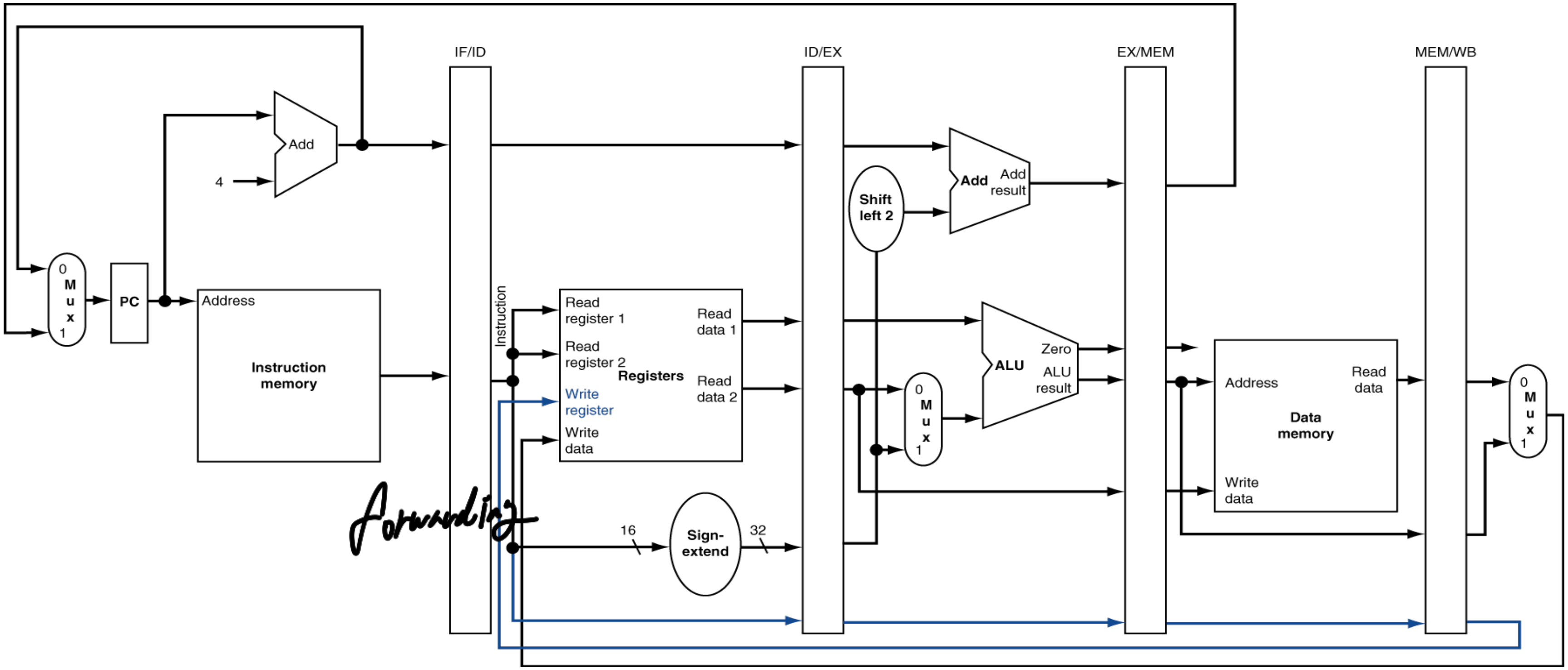
- **Example: executing load (lw) instruction – WB stage**



Wrong register number

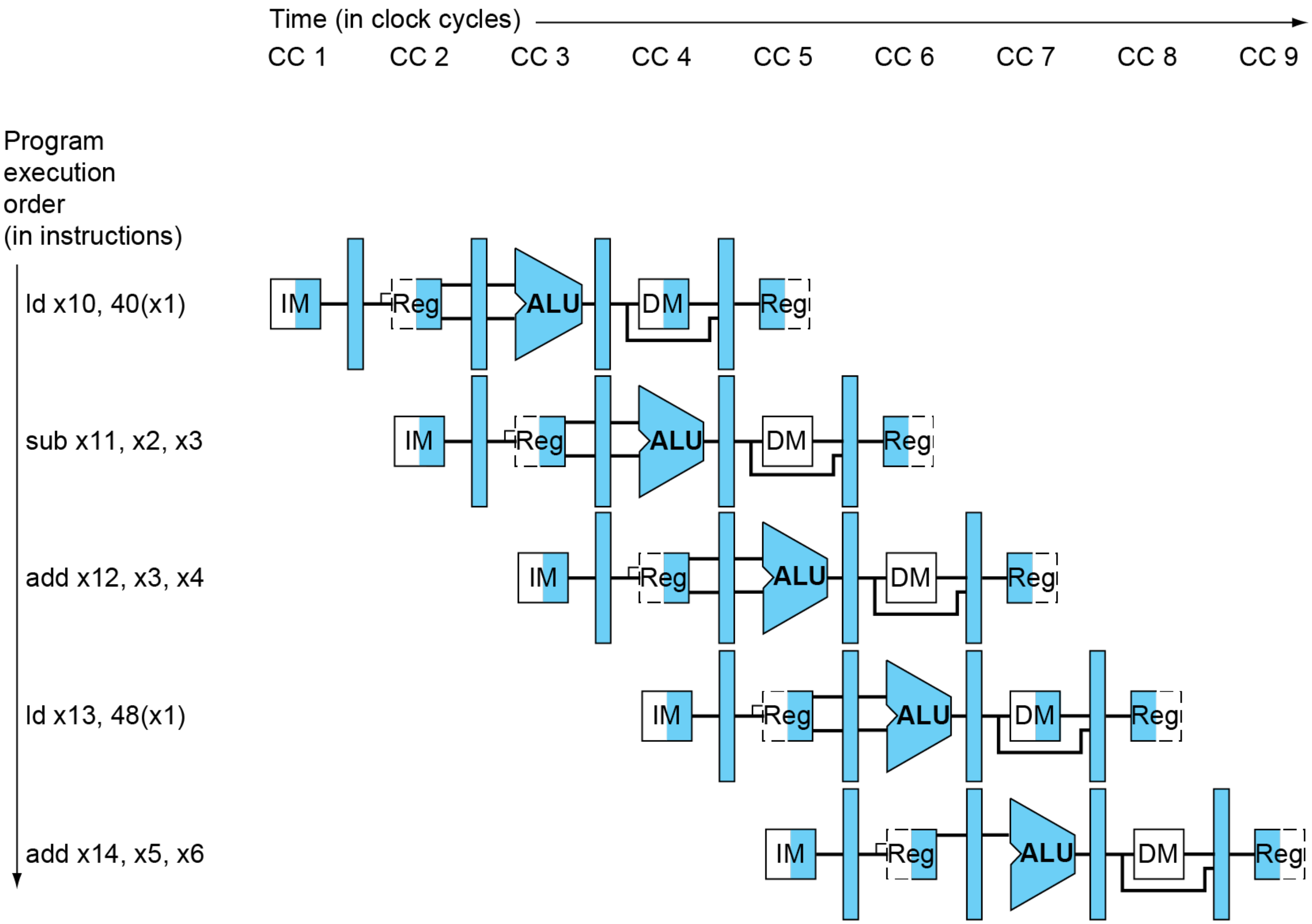
Classic 5-stage Pipeline for RISC: A First Shot

- Example: executing load (lw) instruction – WB stage
  - With corrected datapath



Classic 5-stage Pipeline for RISC: A First Shot

- **Visualizing pipelining: showing resource usage**

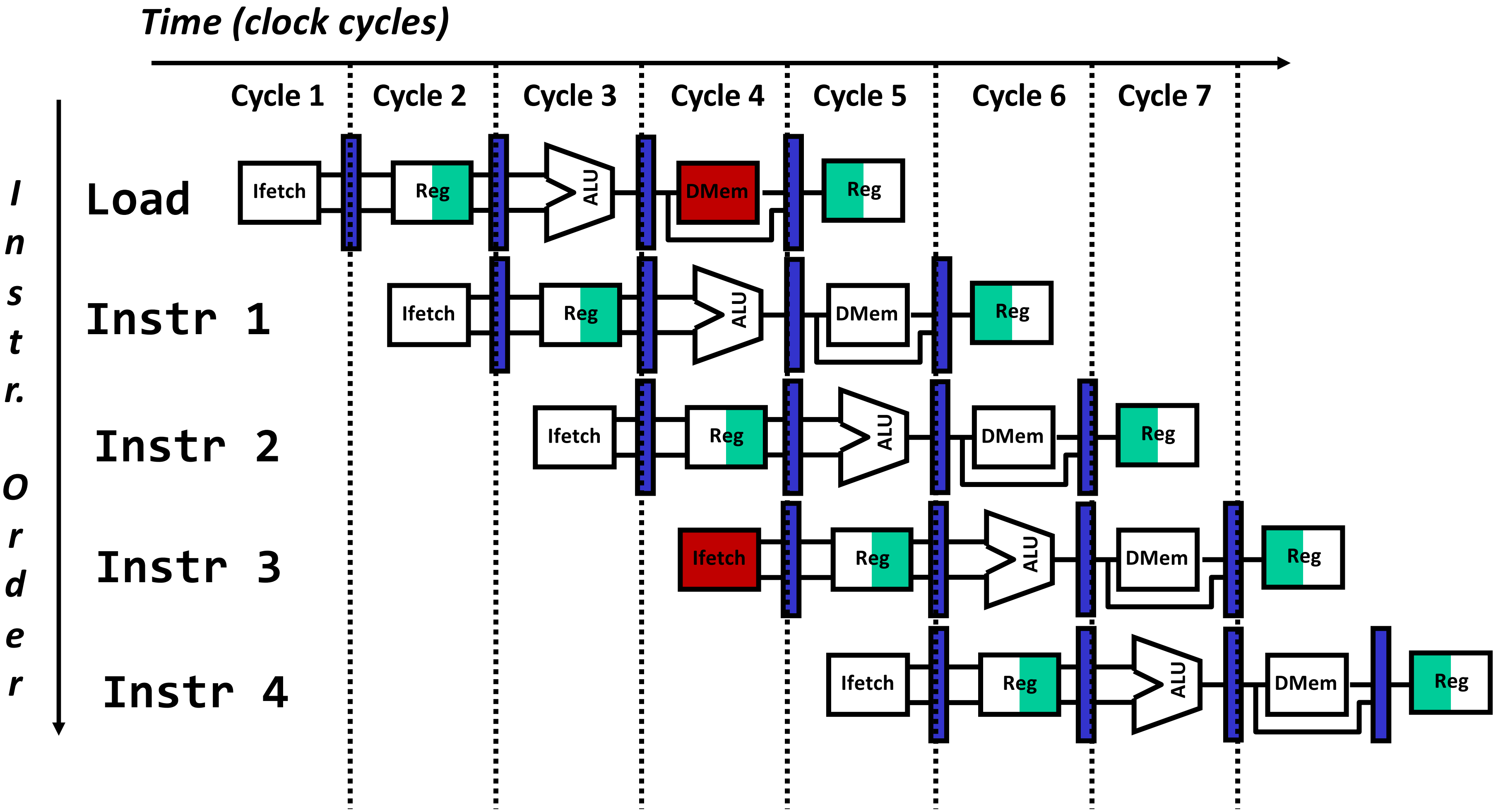


Pipeline Hazards

- **Pipelining is not quite that easy!**
- **Limits to pipelining: Hazards prevent next instruction from executing during its designated clock cycle**
  - Structural hazards: HW cannot support this combination of instructions
  - Data hazards: Instruction depends on result of prior instruction still in the pipeline
  - Control hazards: Caused by delay between the fetching of instructions and decisions about changes in control flow (branches and jumps).

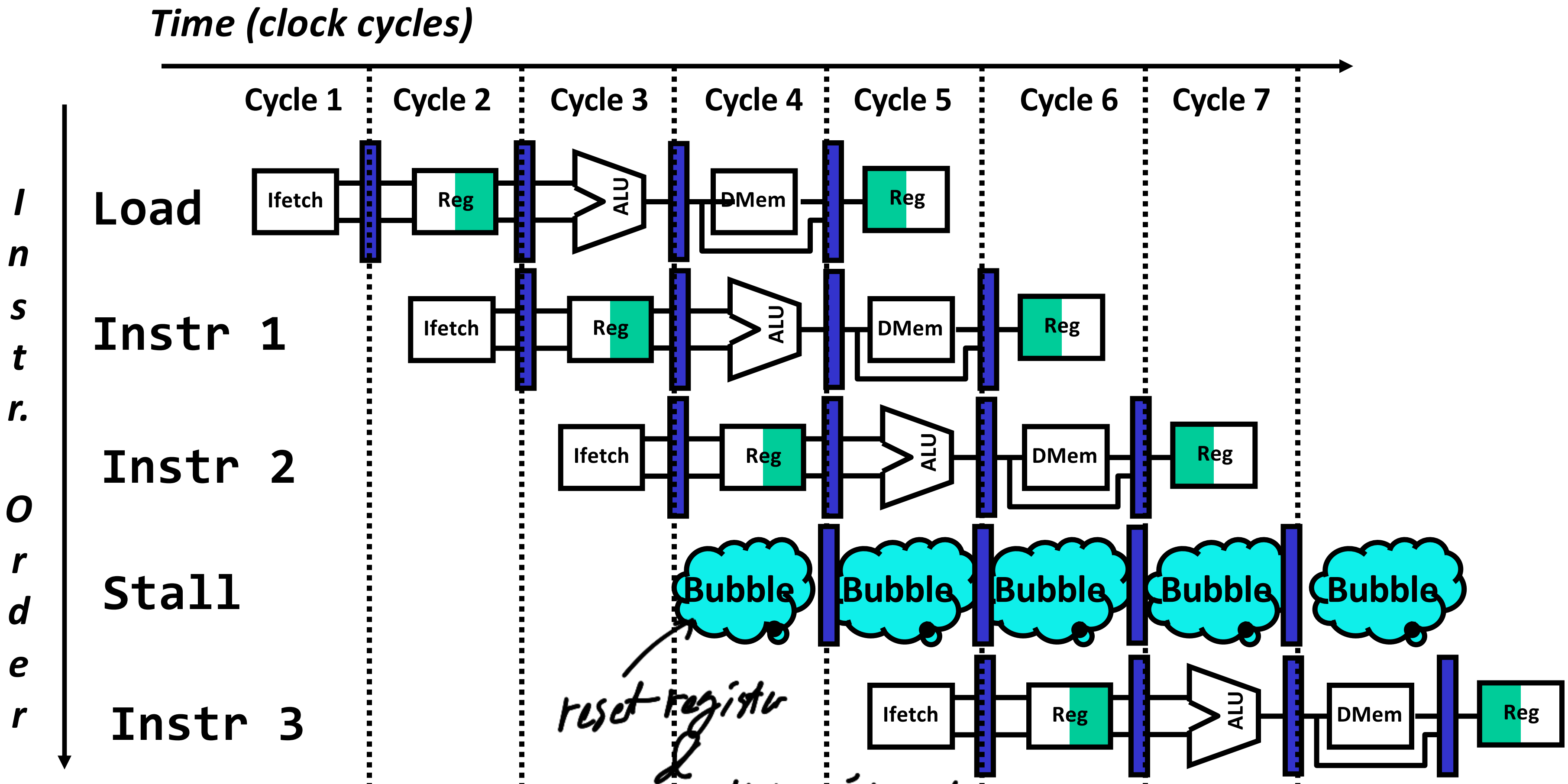
Pipeline Hazards: Structural Hazards

- Structural hazard with only one memory port



Pipeline Hazards: Structural Hazards

- Structural hazard with only one memory port

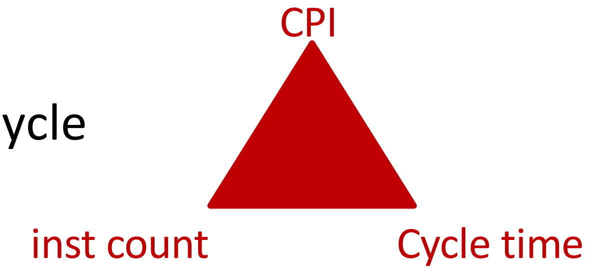


reset register
&
Keep their register values

How do you "bubble" the pipe?

Pipeline Hazards: Structural Hazards

- **Speedup equation for pipelining**
  - CPU time = Instruction_count x CPI x clock_cycle



$$CPI_{pipelined} = \text{Ideal CPI} + \text{Average Stall cycles per Inst}$$

$$\text{Speedup} = \frac{\text{Pipeline depth}}{\text{Ideal CPI} + \text{Pipeline stall CPI}} \times \frac{\text{Cycle Time}_{unpipelined}}{\text{Cycle Time}_{pipelined}}$$

For simple RISC pipeline, CPI = 1:

stage count

$$\text{Speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall CPI}} \times \frac{\text{Cycle Time}_{unpipelined}}{\text{Cycle Time}_{pipelined}}$$

pipeline stage ↑: more register,

Pipeline Hazards: Structural Hazards

- **Example: dual-port vs. single-port**
  - Machine A: Dual ported memory ("Harvard Architecture")
  - Machine B: Single ported memory ("Princeton Architecture"), but its pipelined implementation has a 1.05x faster clock rate
  - Ideal CPI = 1 for both
  - Suppose that Loads are 40% of instructions executed

$$\textbf{SpeedUp}_A = \text{Pipeline Depth}/(1 + 0) \times (\text{clock}_{\text{unpipe}}/\text{clock}_{\text{pipe}})$$

$$= \text{Pipeline Depth}$$

$$\textbf{SpeedUp}_B = \text{Pipeline Depth}/(1 + 0.4 \times 1) \times (\text{clock}_{\text{unpipe}}/(\text{clock}_{\text{unpipe}} / 1.05)$$

$$= (\text{Pipeline Depth}/1.4) \times 1.05$$

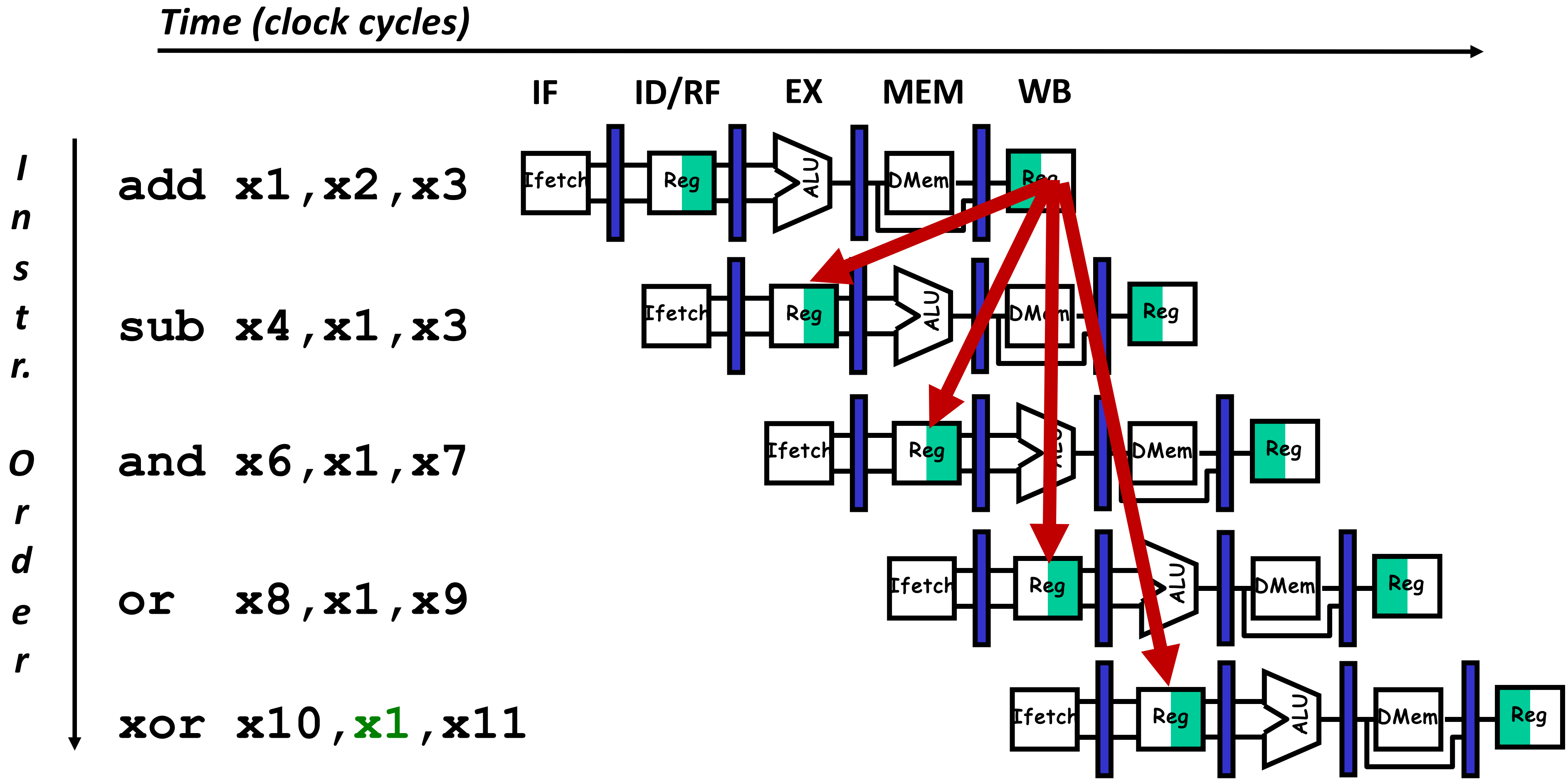
$$= 0.75 \times \text{Pipeline Depth}$$

$$\textbf{SpeedUp}_A / \textbf{SpeedUp}_B = \text{Pipeline Depth}/(0.75 \times \text{Pipeline Depth}) = 1.33$$

  - Machine A is 1.33x faster

Pipeline Hazards: Data Hazards

- Data hazard on R1



Time (clock cycles)

Instr. Order

```
add x1,x2,x3
sub x4,x1,x3
and x6,x1,x7
or  x8,x1,x9
xor x10,x1,x11
```

Pipeline Hazards: Data Hazards

- **Generic data hazards 1: Read After Write (RAW)**
  - Instr_J tries to read operand before Instr_I writes it

```
I:  add x1,x2,x3
J:  sub x4,x1,x3
```

  - Caused by a "Dependence" (in compiler nomenclature). This hazard results from an actual need for communication.

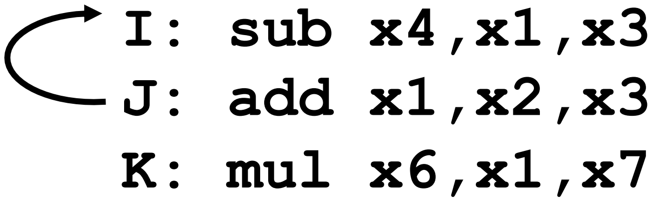
Hazard ⟶ dependency

potential crisis (but not clear)

① far enough

Pipeline Hazards: Data Hazards

- **Generic data hazards 2: Write After Read (WAR)**
  - Instr_J writes operand *before* Instr_I reads it



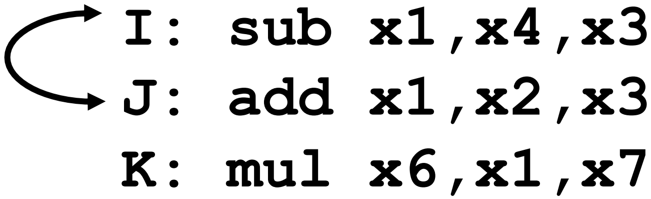
```
I: sub x4,x1,x3
J: add x1,x2,x3
K: mul x6,x1,x7
```

  - Called an "anti-dependence" by compiler writers. This results from reuse of the name "x1".
  - Can't happen in our RISC 5-stage pipeline because:
    - All instructions take 5 stages, and
    - Reads are always in stage 2, and
    - Writes are always in stage 5

Sol: register renaming ?

Pipeline Hazards: Data Hazards

- **Generic data hazards 3: Write After Write (WAW)**
  - Instr$_J$ writes operand *before* Instr$_I$ writes it.



```
I:  sub x1,x4,x3
J:  add x1,x2,x3
K:  mul x6,x1,x7
```

  - Called an "output dependence" by compiler writers. This also results from the reuse of name "x1".
  - Can't happen in our RISC 5 stage pipeline because:
    - All instructions take 5 stages, and
    - Writes are always in stage 5
  - Will see WAR and WAW in more complicated pipes

Pipeline Hazards: Data Hazards

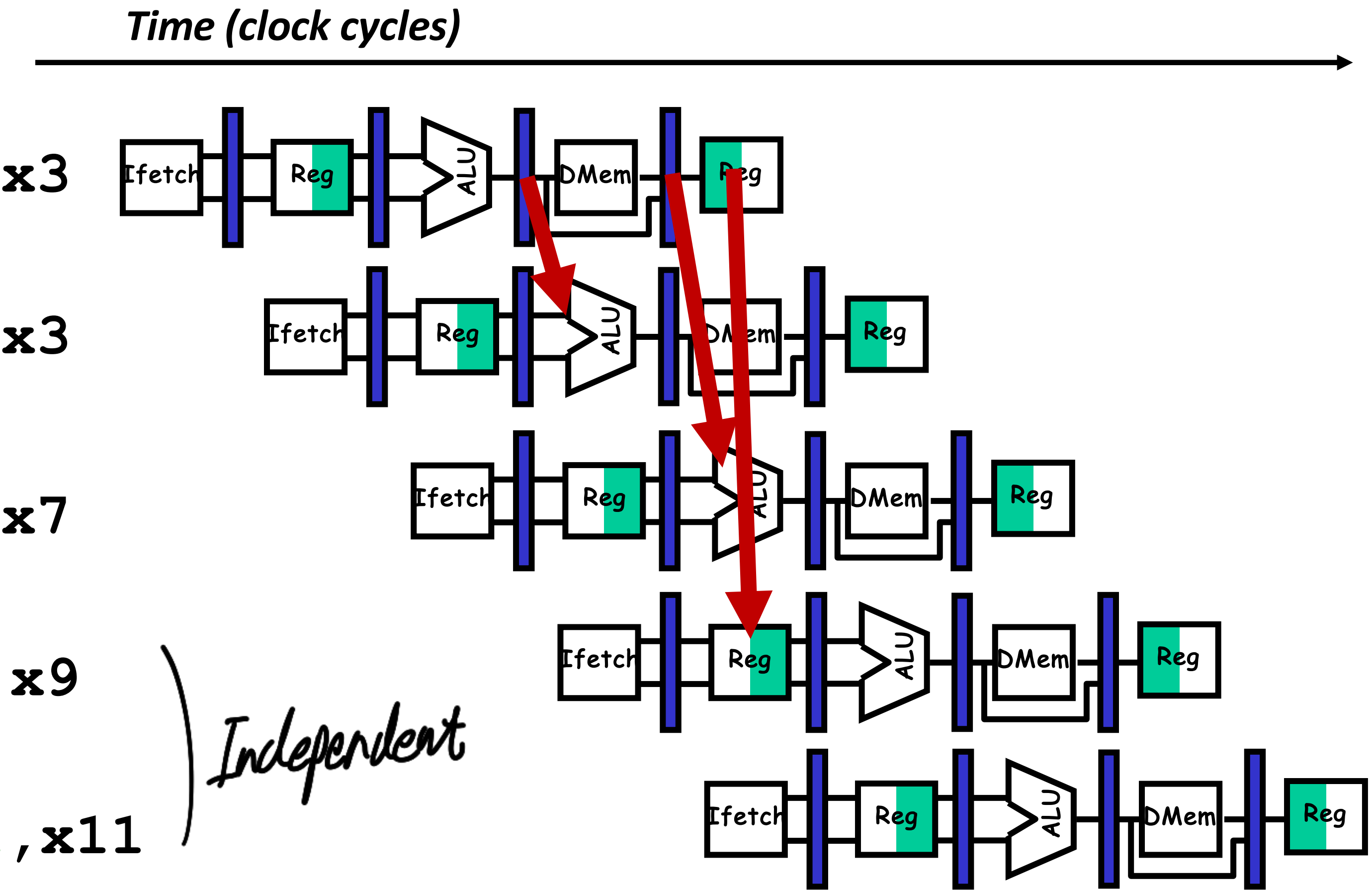
- Forwarding to avoid data hazard

Instr. Order

```
add x1,x2,x3
sub x4,x1,x3
and x6,x1,x7
or  x8,x1,x9
xor x10,x1,x11
```

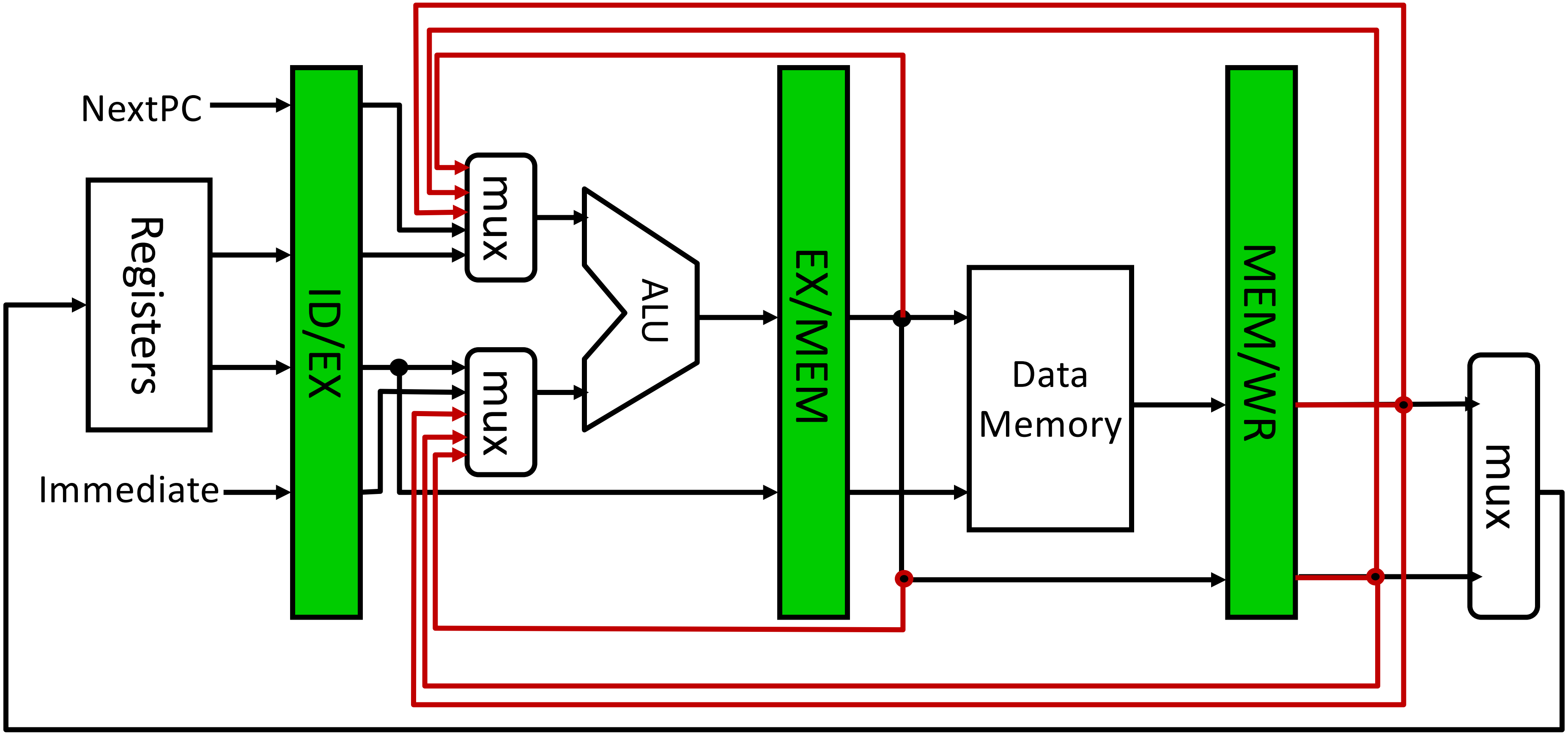
Independent

Time (clock cycles)



Pipeline Hazards: Data Hazards

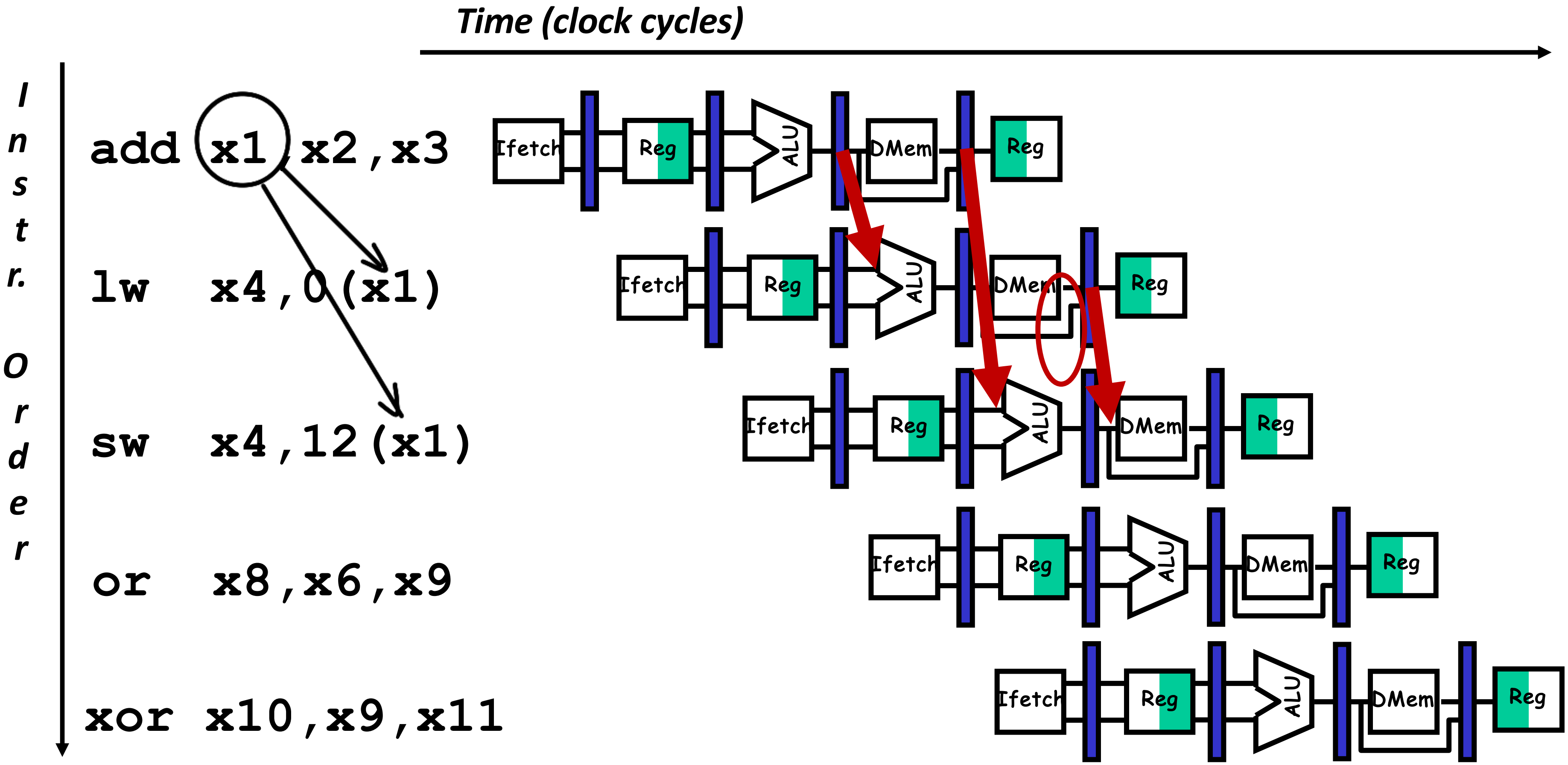
- **HW change for forwarding**



What circuit detects and resolves this hazard?

Pipeline Hazards: Data Hazards

- Forwarding to avoid LW-SW data hazard



Time (clock cycles)

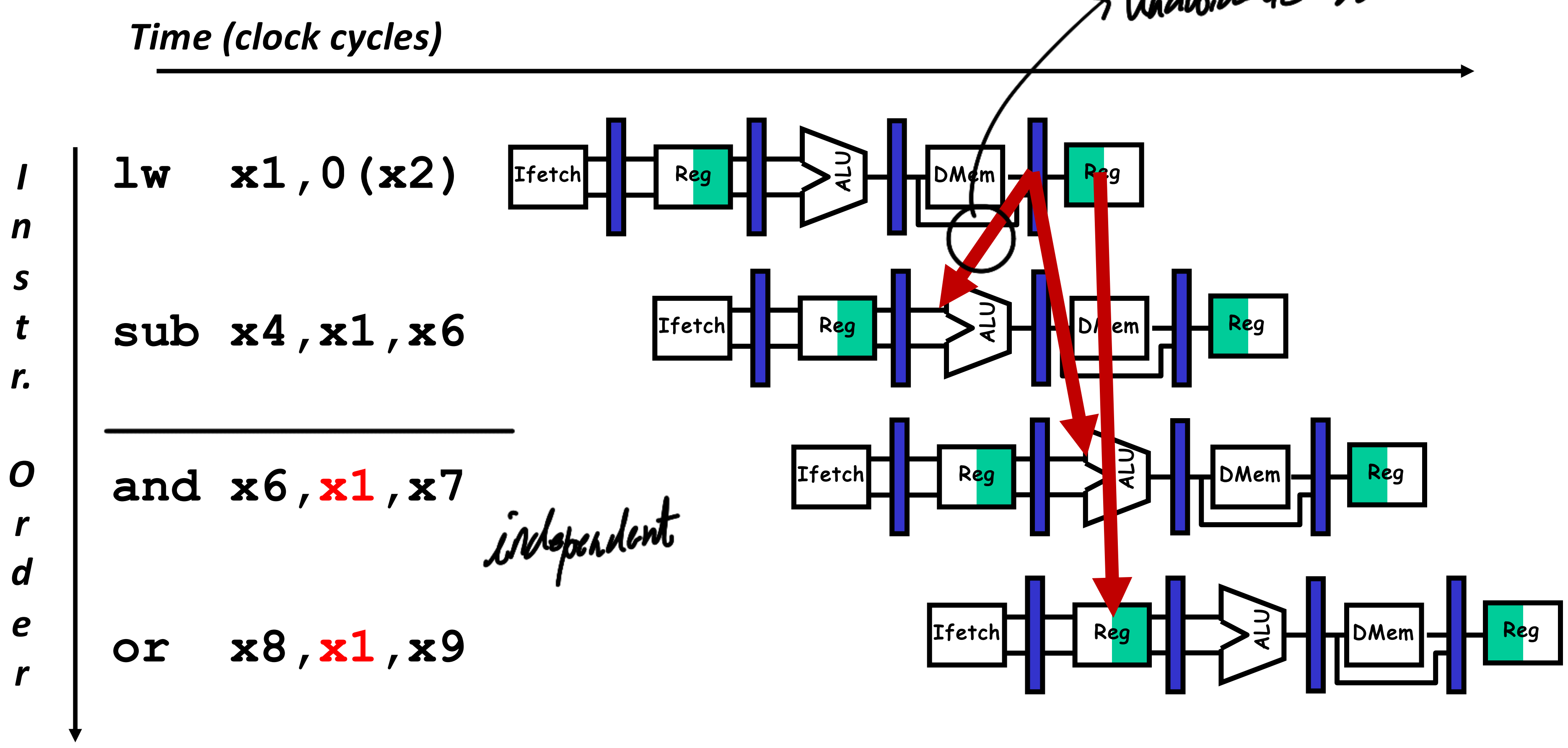
Instr. Order

```
add x1,x2,x3
lw  x4,0(x1)
sw  x4,12(x1)
or  x8,x6,x9
xor x10,x9,x11
```

Pipeline Hazards: Data Hazards

- Data hazard even with forwarding (aka load-use hazard)

→ unavoidable stall



Time (clock cycles)

Instr. Order

```
lw   x1,0(x2)
sub  x4,x1,x6
and  x6,x1,x7
or   x8,x1,x9
```

independent

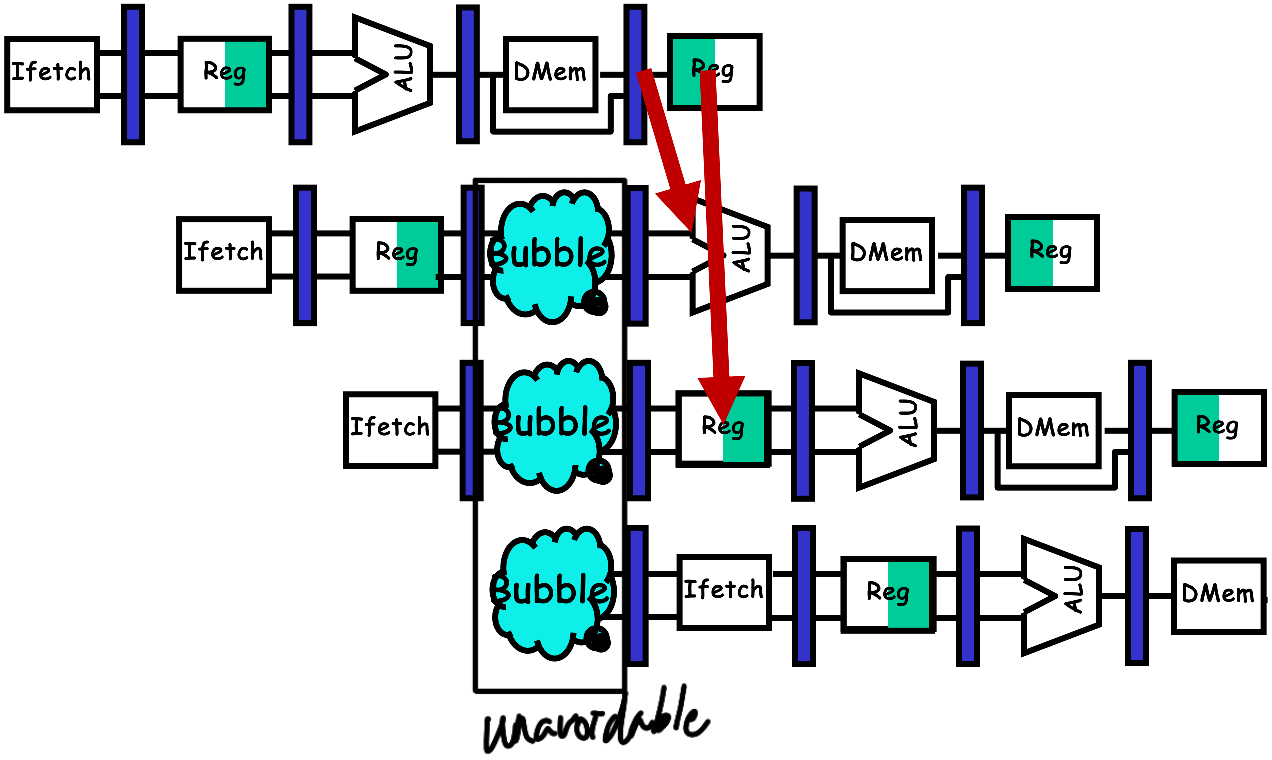
Pipeline Hazards: Data Hazards

- Data hazard even with forwarding (aka load-use hazard)

Time (clock cycles)

Instr. Order

```
lw  x1,0(x2)
sub  x4,x1,x6
and  x6,x1,x7
or   x8,x1,x9
```

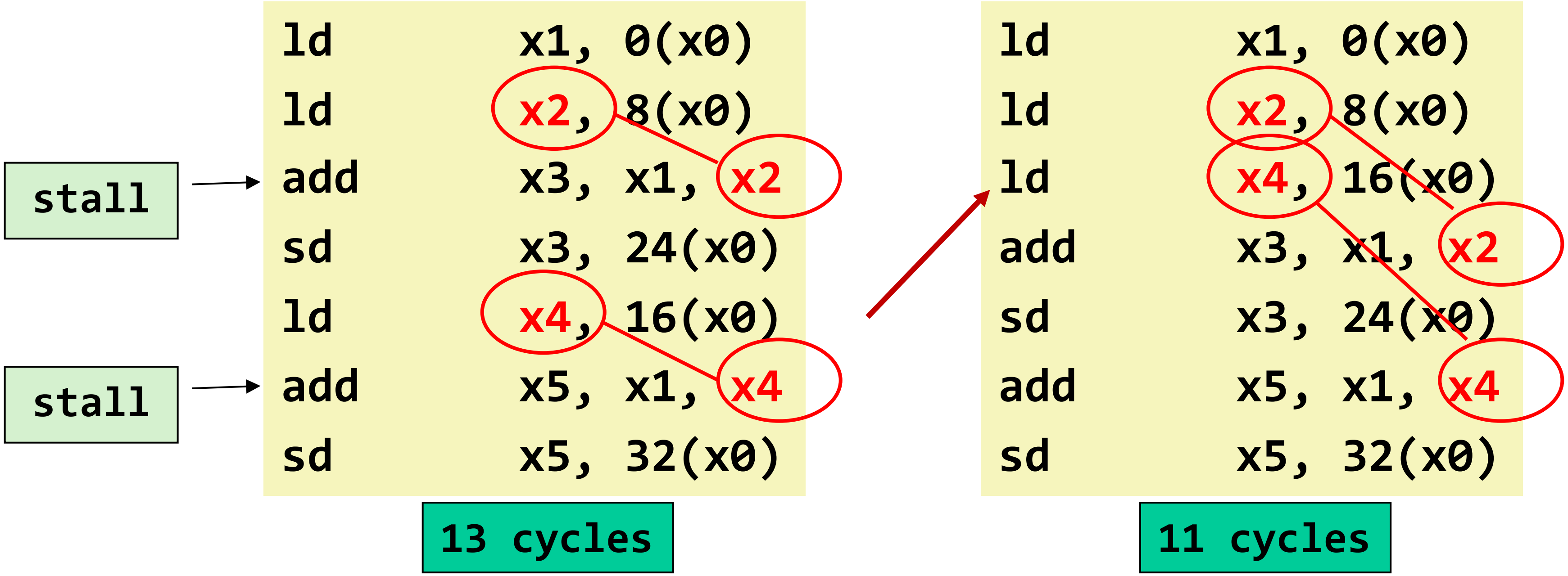


unavoidable

Pipeline Hazards: Data Hazards

- **Software scheduling to avoid load hazards**
  - **Reorder code to avoid use of load result in the next instruction**

```
a = b + e;
d = b + f;
```



```
      ld      x1, 0(x0)
      ld      x2, 8(x0)
stall add     x3, x1, x2
      sd      x3, 24(x0)
      ld      x4, 16(x0)
stall add     x5, x1, x4
      sd      x5, 32(x0)
```

13 cycles

```
ld      x1, 0(x0)
ld      x2, 8(x0)
ld      x4, 16(x0)
add     x3, x1, x2
sd      x3, 24(x0)
add     x5, x1, x4
sd      x5, 32(x0)
```

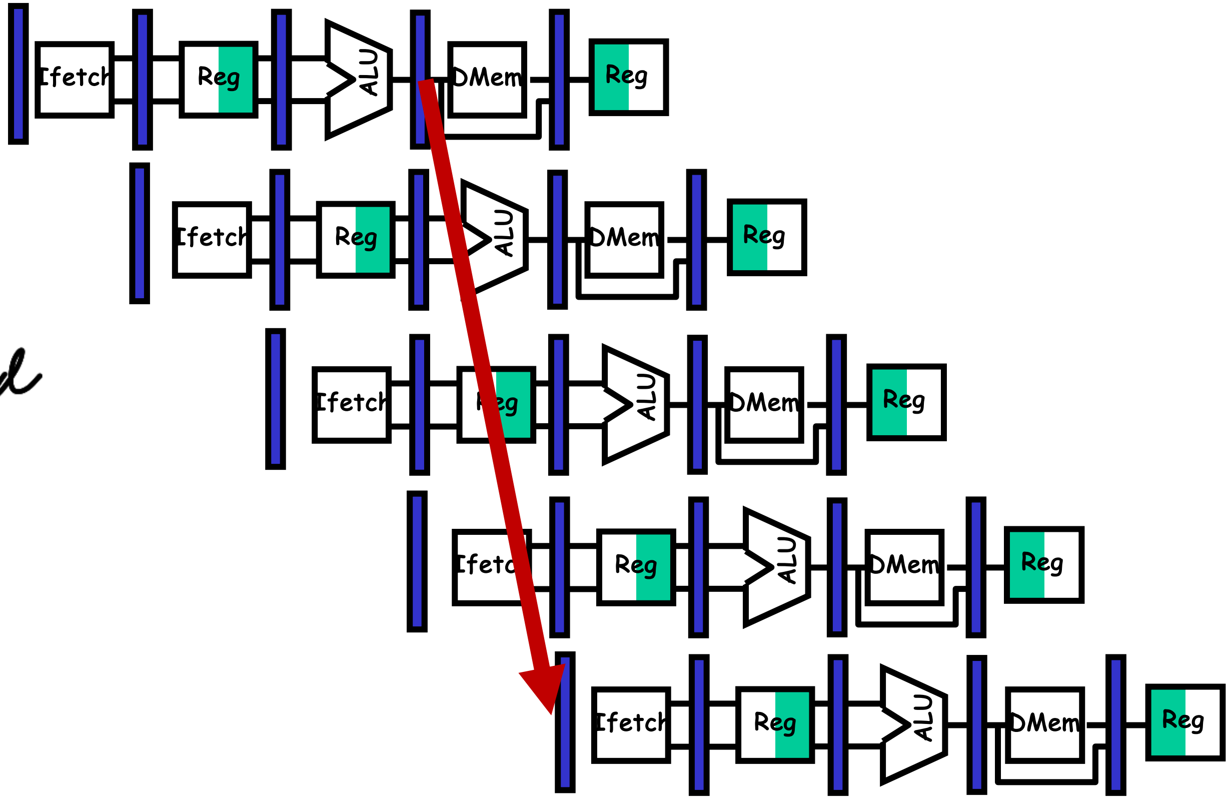
11 cycles

Pipeline Hazards: Control Hazards

- Control hazard on branches: three-stage stall

```
10: beq x1,x0,36

14: and x2,x3,x5

18: or  x6,x1,x7

22: add x8,x1,x9

36: xor x10,x1,x11
```

wasted



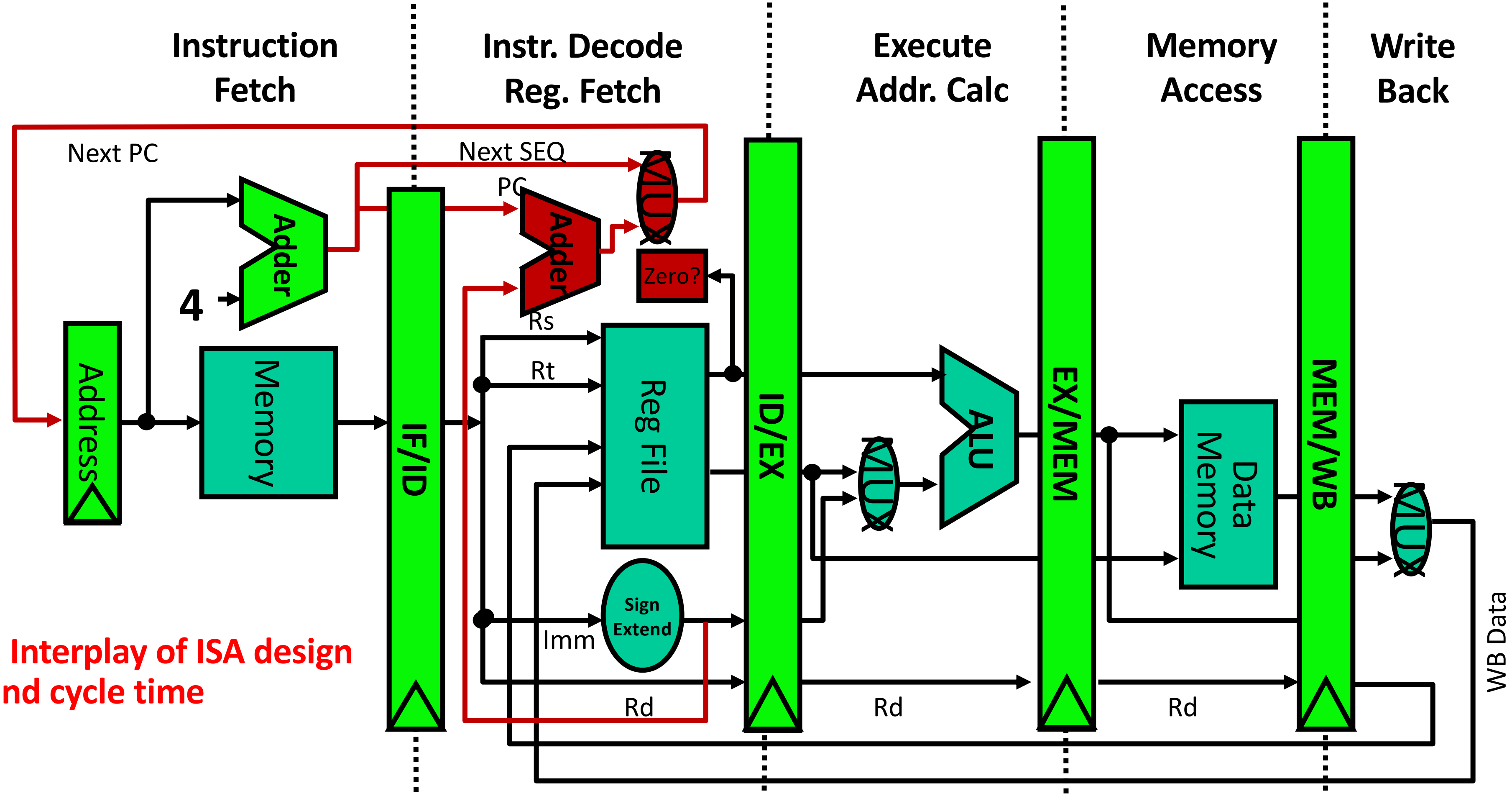
What do you do with the 3 instructions in between?
How do you do it?
Where is the "commit"?

Pipeline Hazards: Control Hazards

- **Performance impact of branch stalls**
  - If CPI = 1, 30% branch, stall 3 cycles
    - New CPI = **1.9!**
  - Two part solution:
    - Determine branch taken or not sooner, AND : prediction
    - Compute taken branch address earlier : add compare logic at ID stage
  - Branch tests if register = 0 or ≠ 0
  - Solution:
    - Move Zero test to ID/RF stage
    - Adder to calculate new PC in ID/RF stage
    - 1 clock cycle penalty for branch (versus 3)

Pipeline Hazards: Control Hazards

- Pipelined RISC datapath (new)



* Interplay of ISA design and cycle time

Pipeline Hazards: Control Hazards

- **Four branch hazard alternatives**

  #1: Stall until branch direction is clear

  #2: Predict Branch Not Taken

  - Execute successor instructions in sequence
  - "Squash" instructions in pipeline if branch actually taken
  - Advantage of late pipeline state update
  - 47% branches not taken on average (MIPS)
  - PC+4 already calculated, so use it to get next instruction

  #3: Predict Branch Taken

  - 53% branches taken on average (MIPS)
  - But haven't calculated branch target address in MIPS
    - It still incurs 1 cycle branch penalty – no performance benefit
    - Other machines: branch target known before outcome

Pipeline Hazards: Control Hazards

- **Four branch hazard alternatives (cont)**

  #4: Delayed Branch

  - Define branch to take place AFTER a following instruction

branch instruction

  sequential successor$_1$

  sequential successor$_2$

  ....

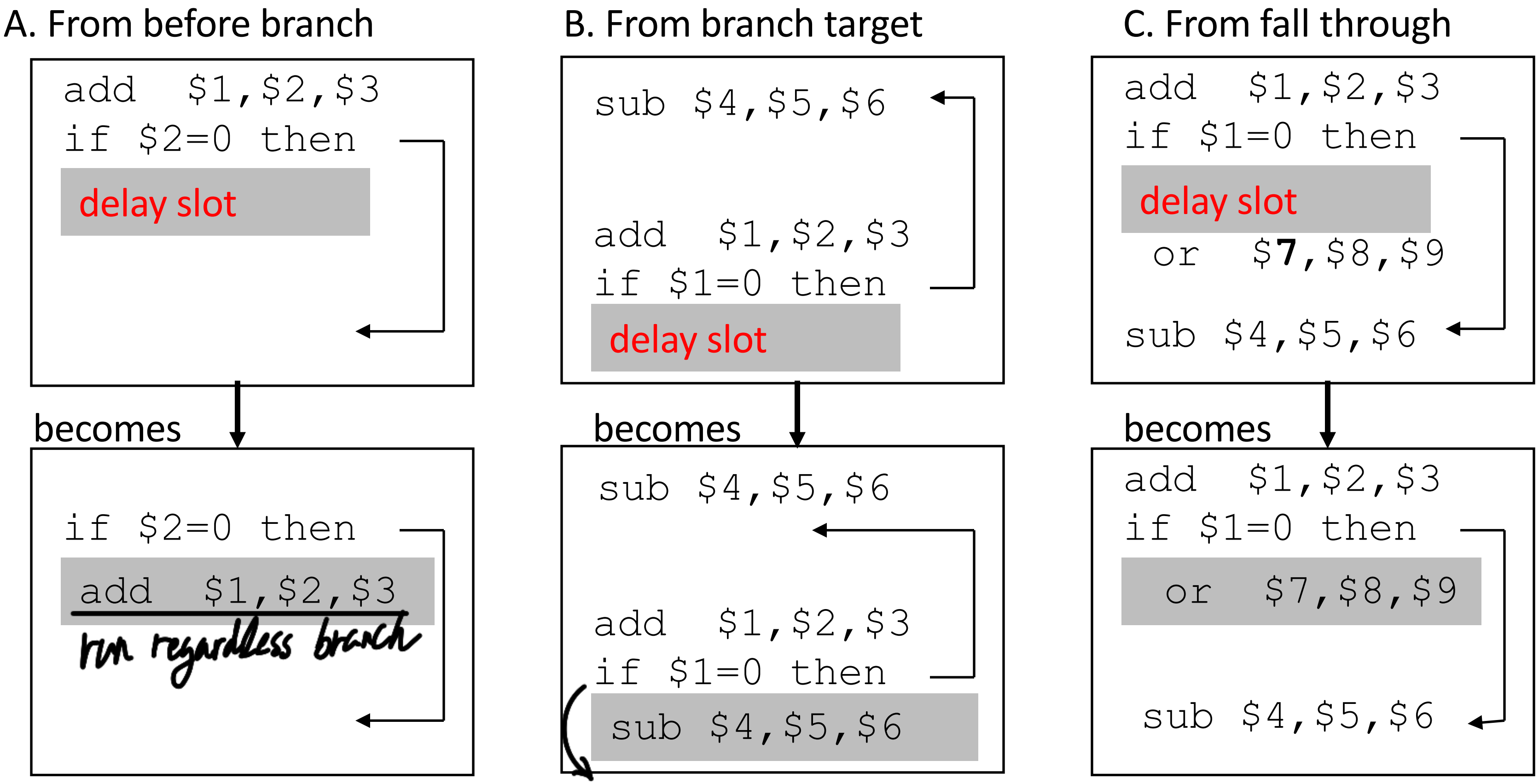
  sequential successor$_n$

branch target if taken

Branch delay of length *n*

  - 1 slot delay allows proper decision and branch target address in 5 stage pipeline
  - MIPS uses this

Pipeline Hazards: Control Hazards



A. From before branch

```
add   $1,$2,$3
if $2=0 then
delay slot
```

becomes

```
if $2=0 then
add   $1,$2,$3
```

run regardless branch

B. From branch target

```
sub $4,$5,$6

add   $1,$2,$3
if $1=0 then
delay slot
```

becomes

```
sub $4,$5,$6

add   $1,$2,$3
if $1=0 then
sub $4,$5,$6
```

C. From fall through

```
add   $1,$2,$3
if $1=0 then
delay slot
 or   $7,$8,$9

sub $4,$5,$6
```

becomes

```
add   $1,$2,$3
if $1=0 then
 or   $7,$8,$9

 sub $4,$5,$6
```

- **Instruction scheduling branch delay slots**
  - A is the best choice, fills delay slot & reduces instruction count (IC)
  - In B, the `sub` instruction may need to be copied, increasing IC
  - In B and C, must be okay to execute ~~`sub`~~ `sub` or `or` when branch fails

Pipeline Hazards: Control Hazards

- **Delayed branch**
  - Compiler effectiveness for single branch delay slot:
    - Fills about 60% of branch delay slots
    - About 80% of instructions executed in branch delay slots useful in computation
    - About 50% (60% x 80%) of slots usefully filled
  - Delayed Branch downside: As processor go to deeper pipelines and multiple issue, the branch delay grows and need more than one delay slot
    - Delayed branching has lost popularity compared to more expensive but more flexible dynamic approaches
    - Growth in available transistors has made dynamic approaches relatively cheaper

Branch prediction

Exception/Interrupt Handling

- **Exceptions vs. interrupts** internal
  - Exception: An unusual event happens to an instruction during its execution
    - Examples: divide by zero, undefined opcode
  - Interrupt: Hardware signal to switch the processor to a new instruction stream external
    - what? Example: a sound card interrupts when it needs more audio output samples (an audio “click” happens if it is left waiting)
- **“Precise” exceptions**
  - Problem: It must appear that the exception or interrupt must appear between 2 instructions (I_i and I_{i+1})
    - The effect of all instructions up to and including I_i is complete
    - No effect of any instruction after I_i can take place
  - The interrupt (exception) handler either aborts program or restarts at instruction I_{i+1}

Exception/Interrupt Handling

Exception handling in RISC-V

return PC

- Save PC of offending (or interrupted) instruction
  - In RISC-V: Supervisor Exception Program Counter (SEPC)

reason

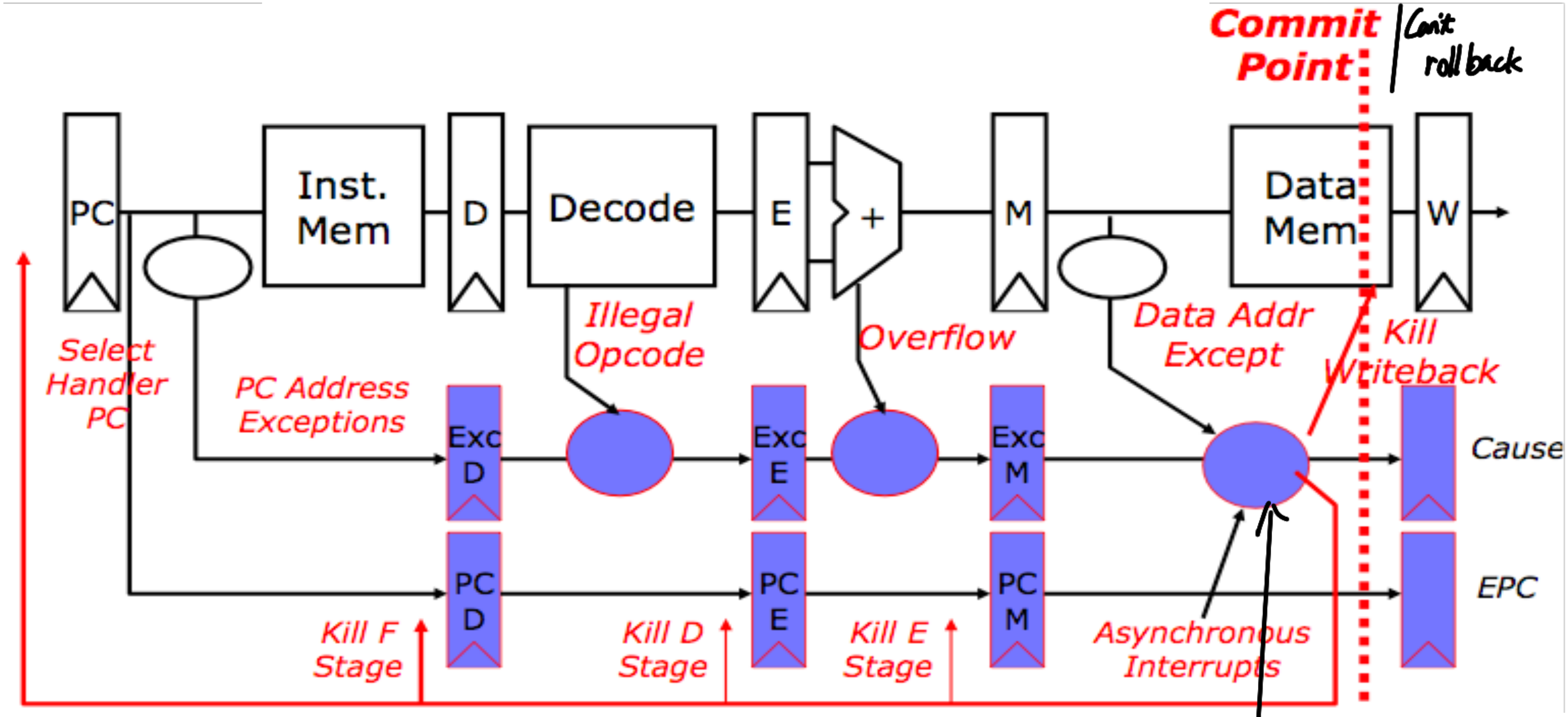
- Save indication of the problem
  - In RISC-V: Supervisor Exception Cause Register (SCAUSE)
  - 64 bits, but most bits unused
    - Exception code field: 2 for undefined opcode, 12 for hardware malfunction, …
- Jump to handler (inside OS)
  - Assume at 0000 0000 1C09 0000$_{hex}$

Exception/Interrupt Handling

- **Handler Actions**
  - Read cause, and transfer to relevant handler
  - Determine action required
  - If restartable
    - Take corrective action
    - use SEPC to return to program
  - Otherwise
    - Terminate program
    - Report error using SEPC, SCAUSE, …

Exception/Interrupt Handling

- **Precise exceptions in static pipelines**



Key observation: architected state only change in memory and register write stages.

priority

Exceptions in a Pipeline

- **Another form of control hazard**
- **Consider malfunction on add in EX stage**
  `add x1, x2, x1`
  - Prevent x1 from being clobbered
  - Complete previous instructions
  - Flush add and subsequent instructions
  - Set SEPC and SCAUSE register values
  - Transfer control to handler
- **Similar to mispredicted branch**
  - Use much of the same hardware

Handling Multi-Cycle Operations

- **Pipelining becomes complex when we want high performance in the presence of:**
  - Long latency or partially pipelined floating-point units
  - Memory systems with variable access time
  - Multiple arithmetic and memory units

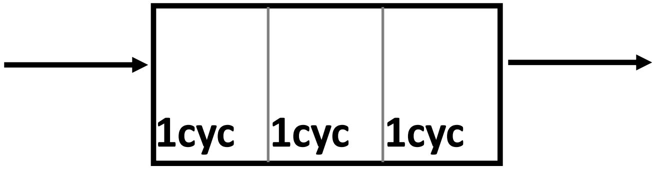
Handling Multi-Cycle Operations

- **Floating-Point Unit (FPU)**
  - Much more hardware than an integer unit
    - Single-cycle FPU is a bad idea – why?
  - Common to have several FPU's
  - Common to have different types of FPU's: Fadd, Fmul, Fdiv, …
  - An FPU may be pipelined, partially pipelined or not pipelined
  - To operate several FPU's concurrently the FP register file needs to have more read and write ports

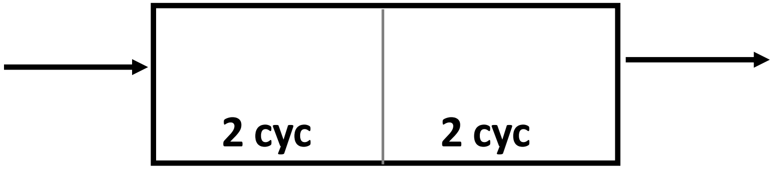
Handling Multi-Cycle Operations

- **Functional Unit Characteristics**

fully pipelined



partially pipelined



- **Functional units have internal pipeline registers**
  - Operands are latched when an instruction enters a functional unit
  - Following instructions are able to write register file during a long-latency operation

Handling Multi-Cycle Operations

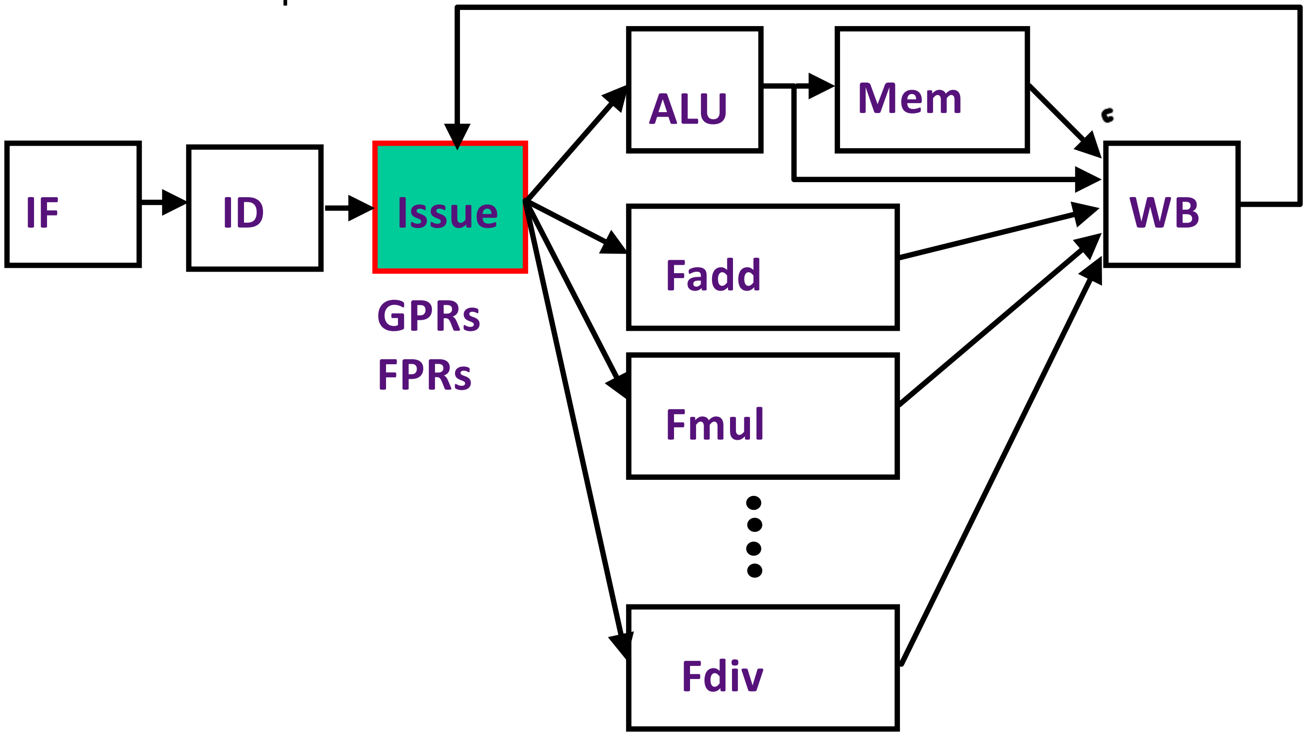
- **Floating-Point ISA**
  - Interaction between FP datapath and integer datapath is determined by ISA
  - RISC-V ISA
    - Separate register files for FP and Integer instructions
      - the only interaction is via a set of move/convert instructions (some ISA's don't even permit this)
    - Separate load/store for FPR's and GPR's but both use GPR's for address calculation
    - Branches on FP values
      - Utilizing GPR for holding comparison results
      - feq (equal), flt (less than), fle (less than or equal to)

Handling Multi-Cycle Operations

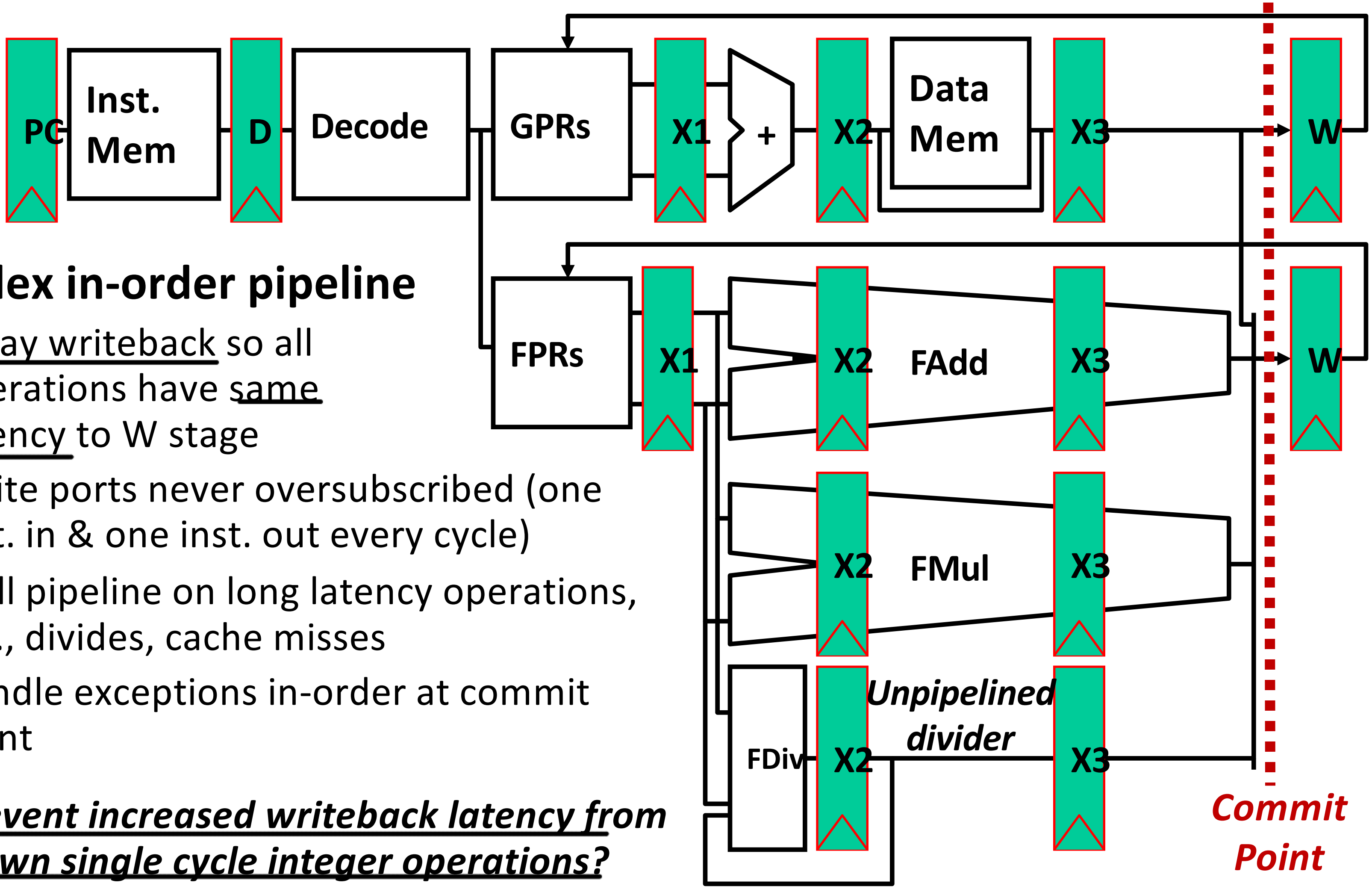
- **Common approaches to improving memory performance in realistic memory systems:**
  - Caches - single cycle except in case of a miss ⇒ stall
  - Banked memory - multiple memory accesses ⇒ bank conflicts
  - split-phase memory operations (separate memory request from response), many in flight ⇒ out-of-order responses
- **Latency of access to the main memory is usually much greater than one cycle and often unpredictable**
  - Solving this problem is a central issue in computer architecture

Handling Multi-Cycle Operations

- **Issues in Complex Pipeline Control**
  - Structural conflicts at the execution stage if some FPU or memory unit is not pipelined and takes more than one cycle
  - Structural conflicts at the write-back stage due to variable latencies of different functional units
  - Out-of-order write hazards due to variable latencies of different functional units
  - How to handle exceptions?



Handling Multi-Cycle Operations

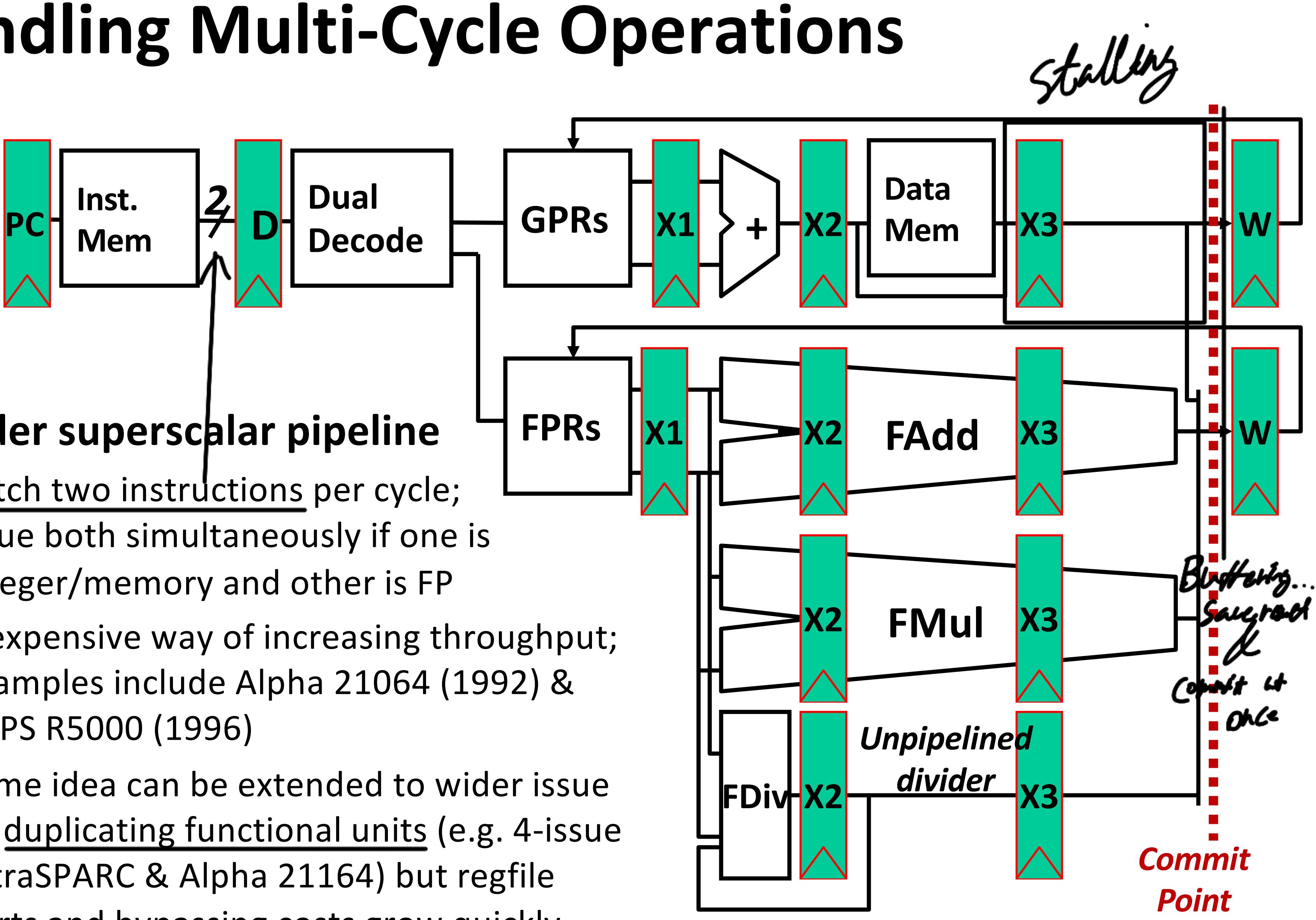


- **Complex in-order pipeline**
  - Delay writeback so all operations have same latency to W stage
  - Write ports never oversubscribed (one inst. in & one inst. out every cycle)
  - Stall pipeline on long latency operations, e.g., divides, cache misses
  - Handle exceptions in-order at commit point

How to prevent increased writeback latency from slowing down single cycle integer operations?

Bypassing (=Forwarding)

Handling Multi-Cycle Operations



- **In-order superscalar pipeline**
  - Fetch two instructions per cycle; issue both simultaneously if one is integer/memory and other is FP
  - Inexpensive way of increasing throughput; examples include Alpha 21064 (1992) & MIPS R5000 (1996)
  - Same idea can be extended to wider issue by duplicating functional units (e.g. 4-issue UltraSPARC & Alpha 21164) but regfile ports and bypassing costs grow quickly

Summary

- **Just overlap tasks; easy if tasks are independent**
- **Speed Up ≤ Pipeline Depth; if ideal CPI is 1, then:**

$$\text{Speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall CPI}} \times \frac{\text{Cycle Time}_{\text{unpipelined}}}{\text{Cycle Time}_{\text{pipelined}}}$$

- **Hazards limit performance on computers:**
  - Structural: need more HW resources
  - Data (RAW,WAR,WAW): need forwarding, compiler scheduling
  - Control: delayed branch, prediction
- **Exceptions, interrupts add complexity**
- **Handling multi-cycle operations adds additional complexity**