DEPT. OF ELECTRONICS ENGINEERING & INST. OF ELECTRONICS NCTU

# Computer Architecture

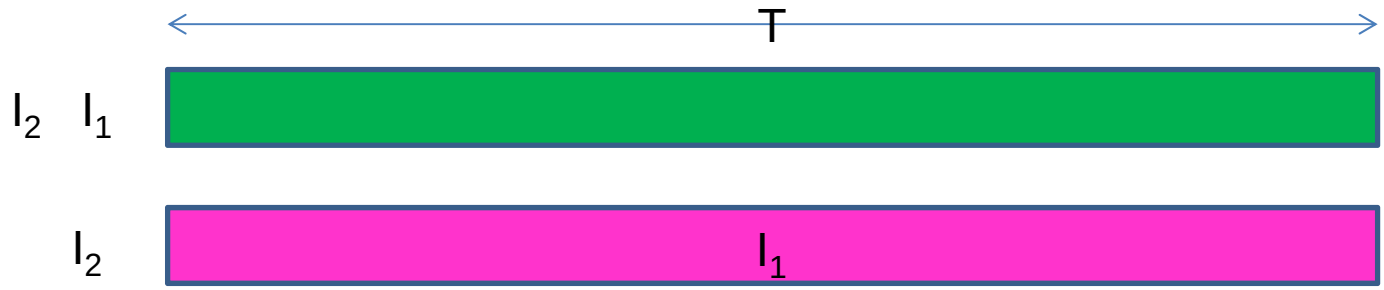
# Lecture 4: Pipelining (Appendix C)

Chih-Wei Liu 劉志尉

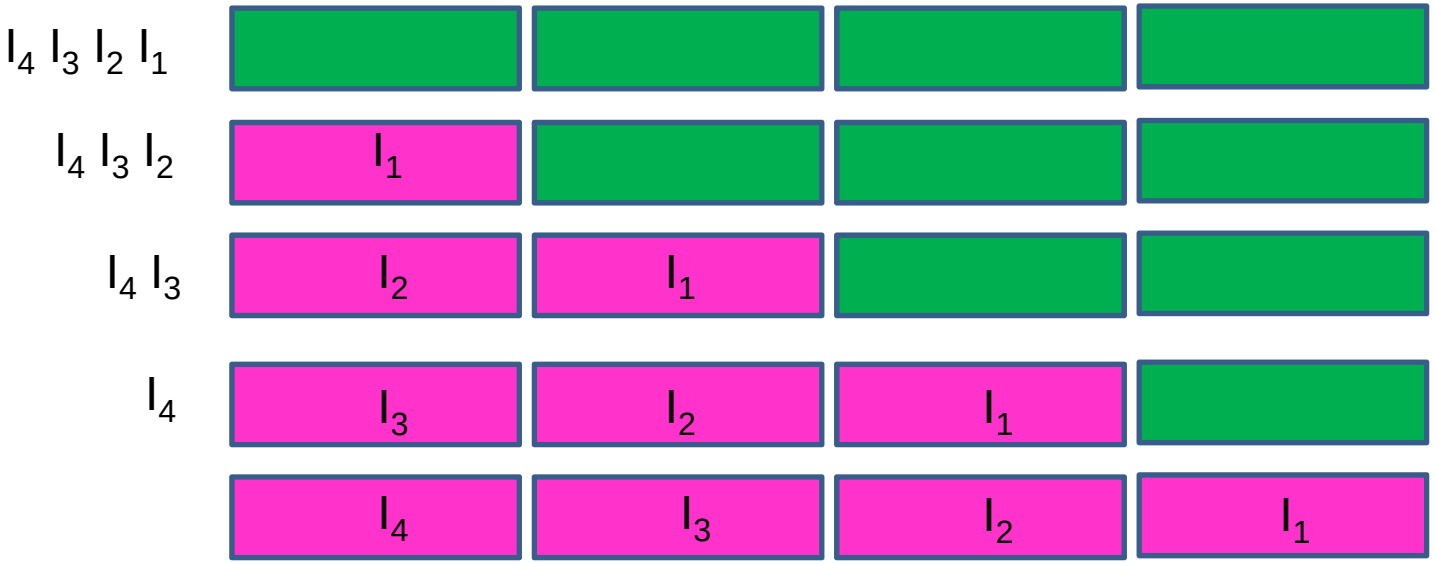
National Chiao Tung University

cwliu@twins.ee.nctu.edu.tw

# Why Pipeline?



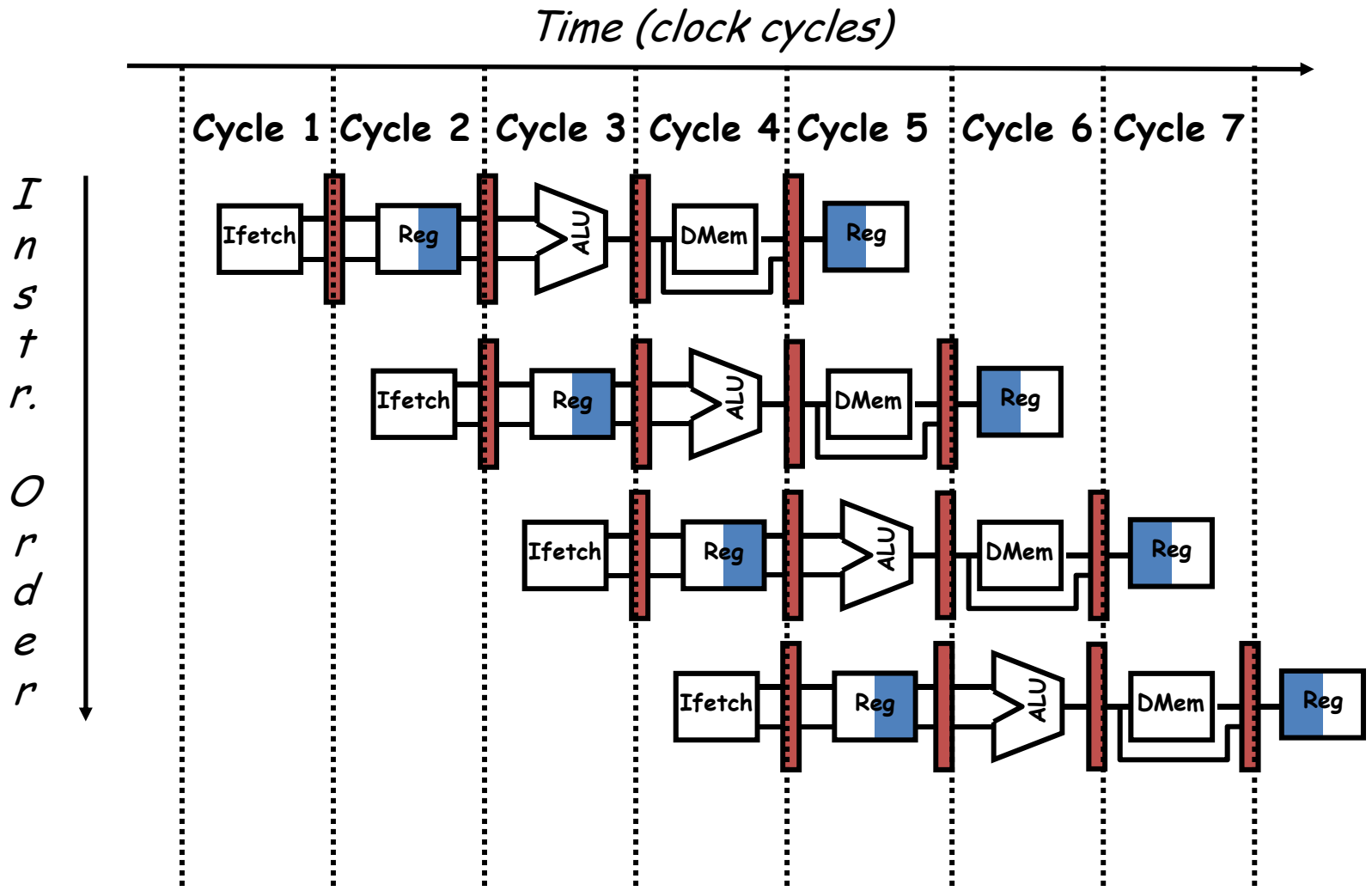

Throughput = 1/T




Throughput = 4/T

# Visualizing Pipelining

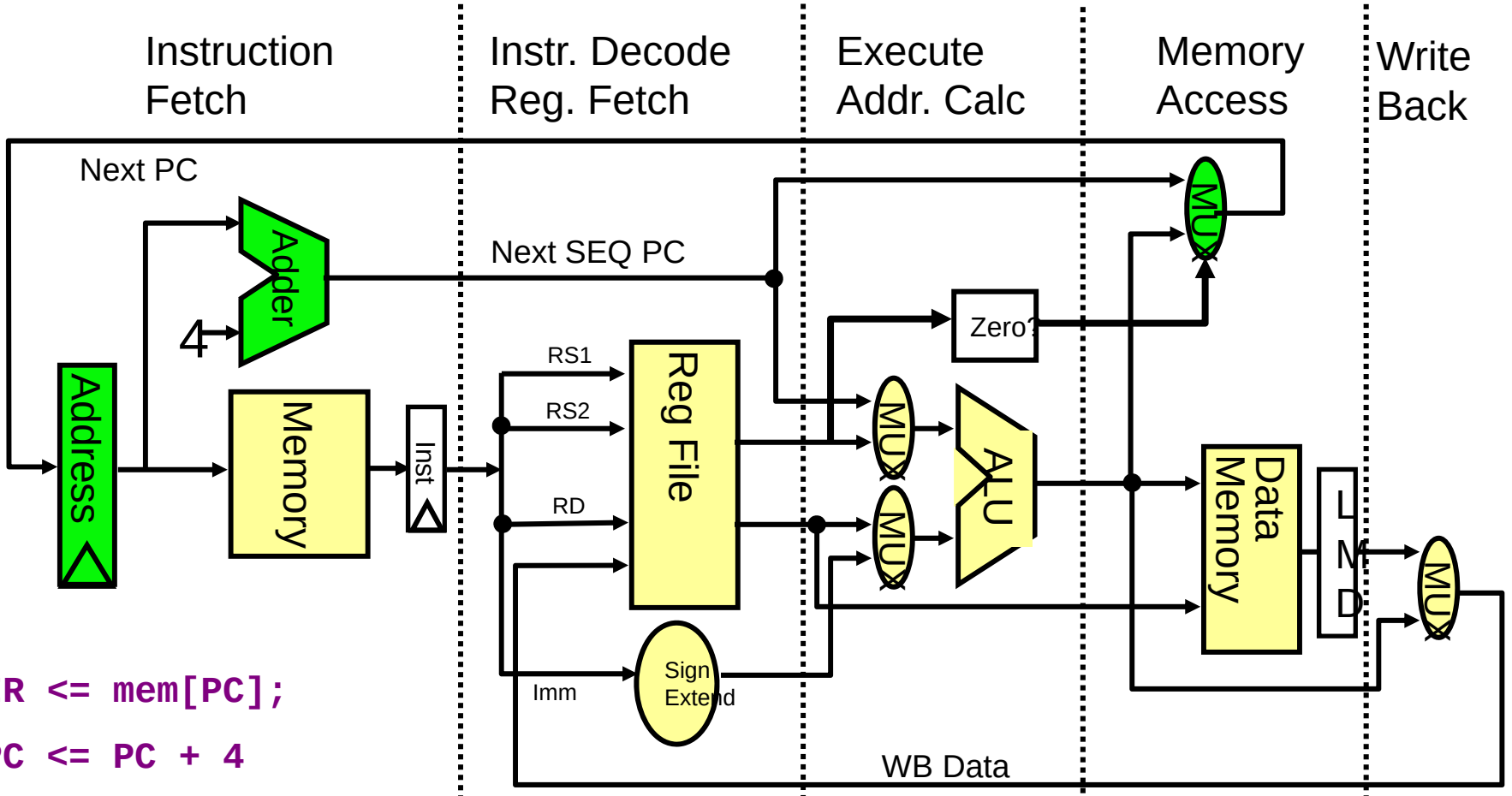
Figure C.3, Page C-9



# Designing a Pipelined Processor

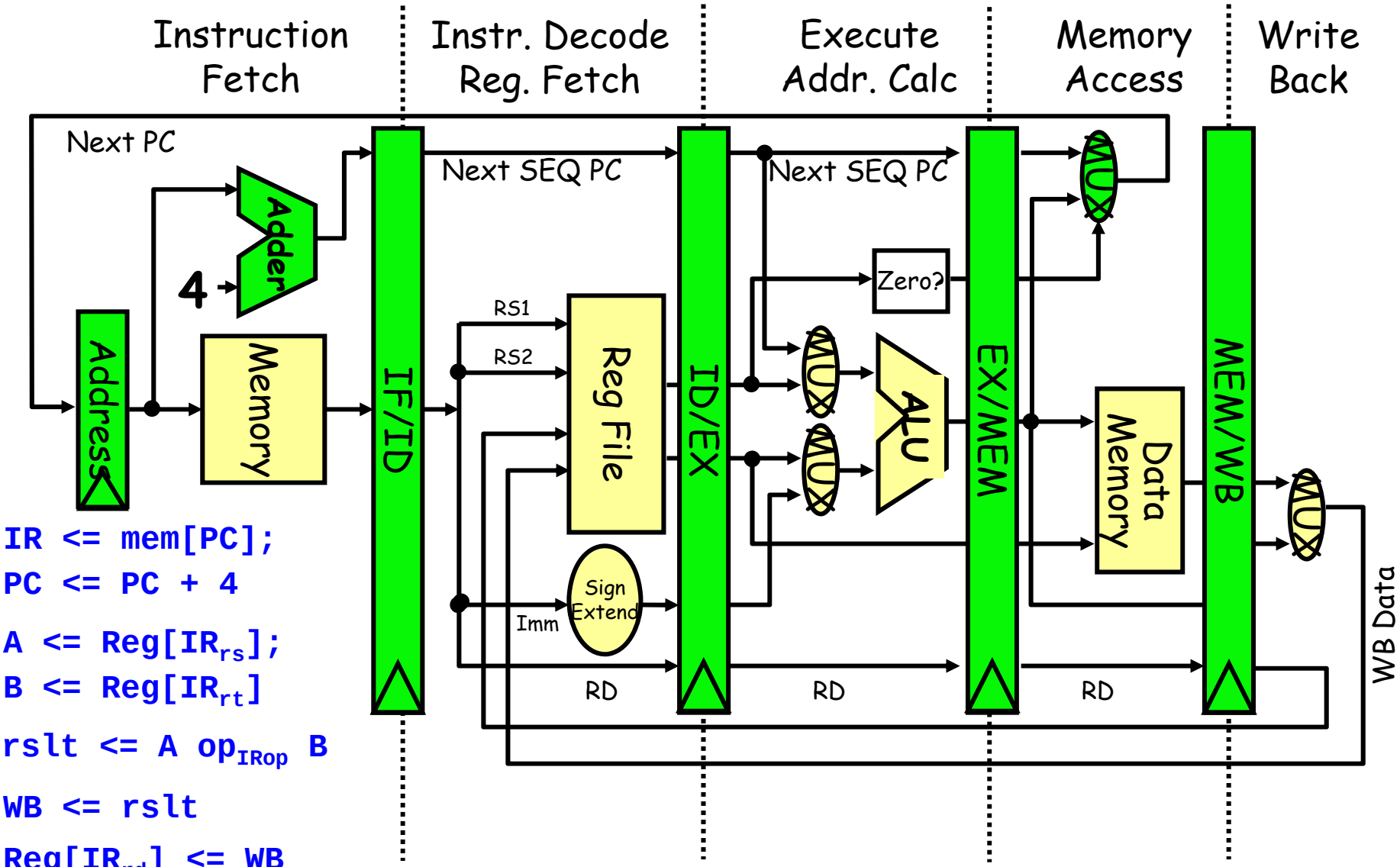
- Start with ISA
- Examine the datapath and control diagram
  - Starting with single- or multi-cycle datapath?
  - Single- or multi-cycle control?
- Partition datapath into steps
- Insert pipeline registers between successive steps
- Associate resources with steps
- Ensure that flows do not conflict, or figure out how to resolve
- Assert control in appropriate stage

# Example: 5 Steps of MIPS Datapath

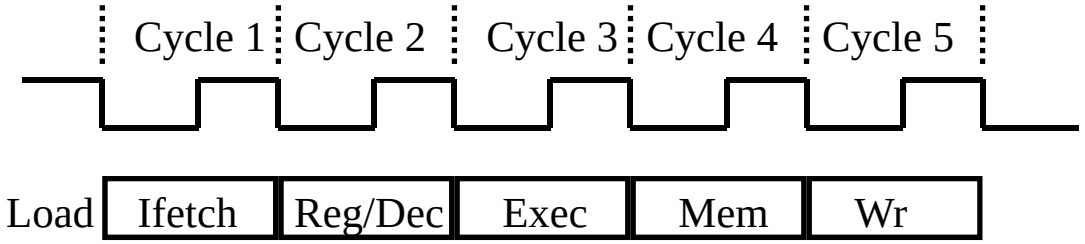



```
IR <= mem[PC];
PC <= PC + 4
Reg[IR_rd] <= Reg[IR_rs] op_IROp Reg[IR_rt]
```

# 5 Steps of MIPS Datapath



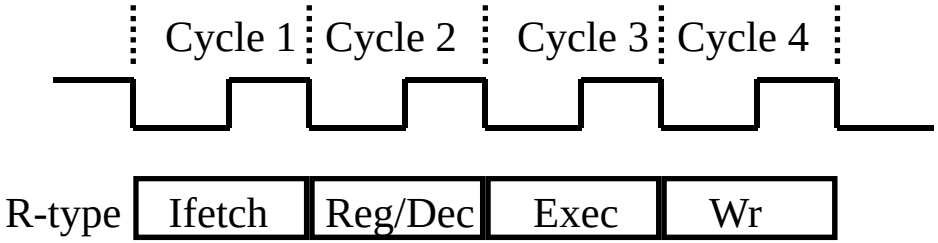
# The 5-Step of the Lw-Instruction




- Lw-instruction can be implemented in 5 clock cycles
- Ifetch: Instruction Fetch
  - Fetch the instruction from the Instruction Memory
- Reg/Dec: Registers Fetch and Instruction Decode
- Exec: Execution and calculate the memory address
- Mem: Read the data from the Data Memory
- Wr: Write the data back to the register file

Branch requires ? cycles, Store requires ? cycles, others require ? cycles

# The 4-Step of R-type Instruction

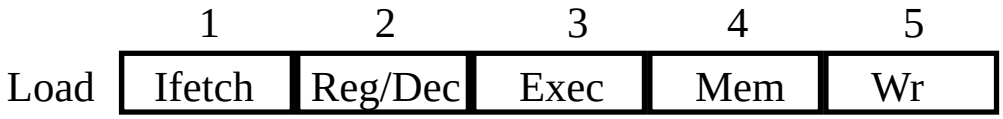



does not access data memory…

- Ifetch: Instruction Fetch
  - Fetch the instruction from the Instruction Memory
  - Update PC
- Reg/Dec: Registers Fetch and Instruction Decode
- Exec:
  - ALU operates on the two register operands
- Wr: Write the ALU output back to the register file

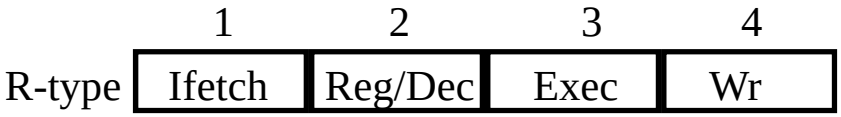
# Important Observation

- Each functional unit can only be used once per instruction:
  - Load uses Register File's Write Port during its 5th step



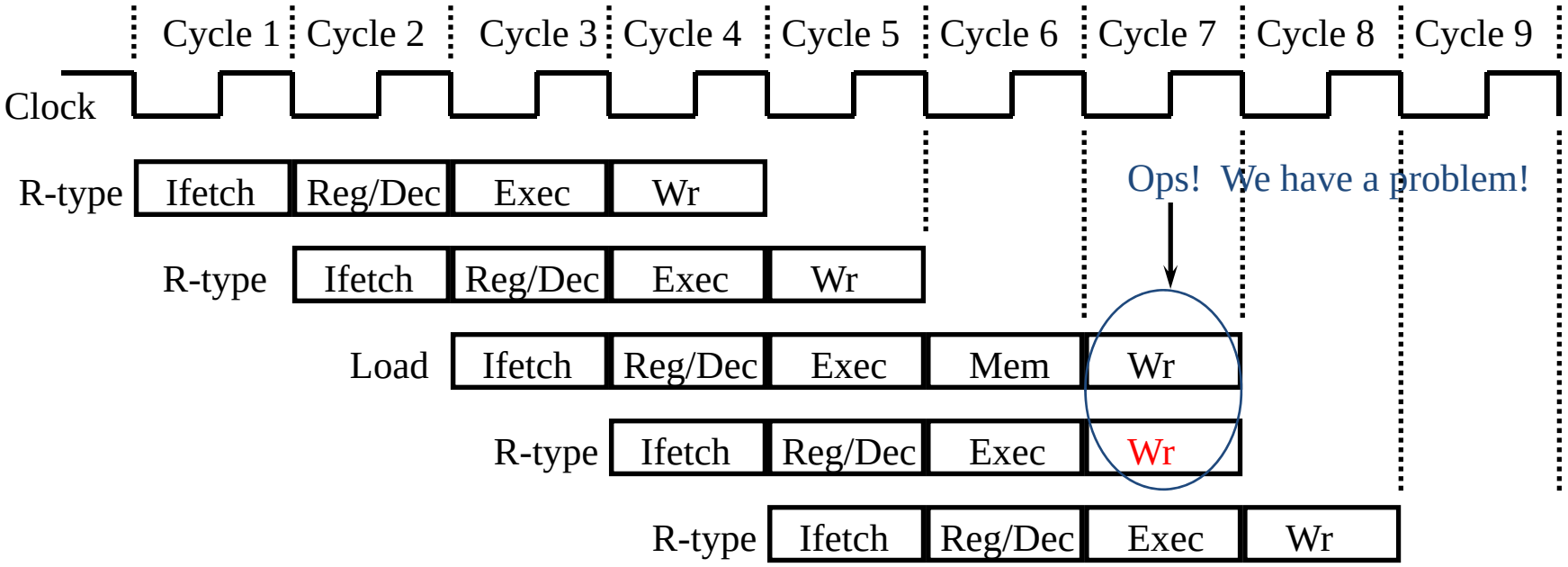

This's what caused the problem

  - R-type uses Register File's Write Port during its 4th step



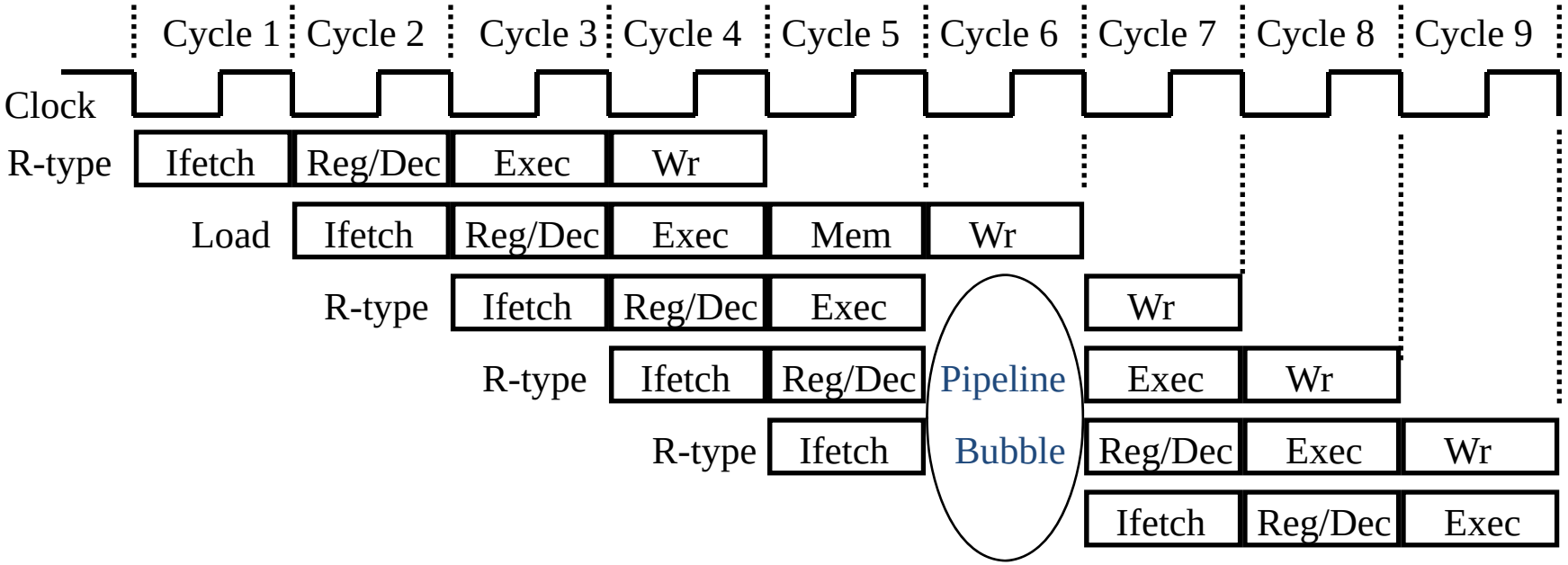

A structural hazard will be happened !!

# Pipelining the R-type and Load Instructions




- Structural hazard:
  - Two instructions try to write to the register file at the same time!
  - Only one write port

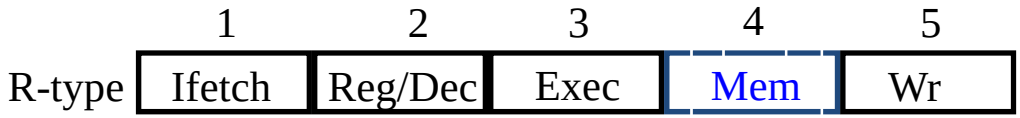
# Sol 1: Insert "Bubble" into the Pipeline

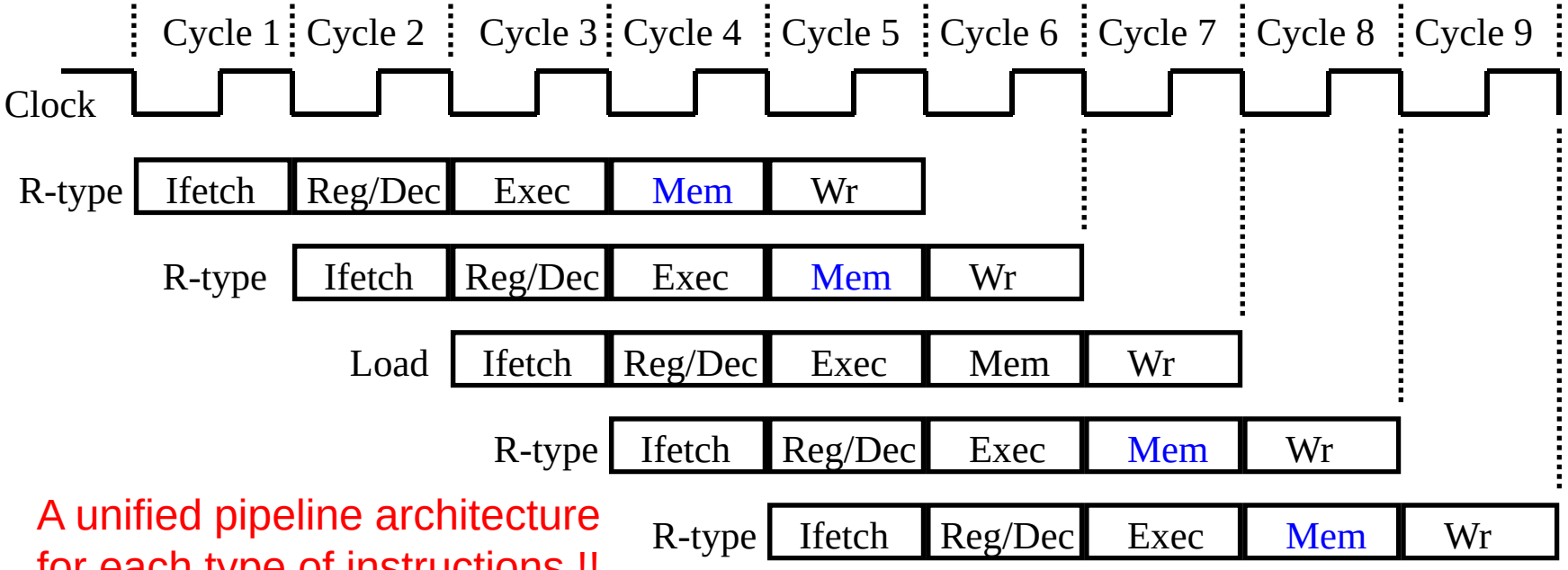



- Insert a "bubble" into the pipeline to prevent 2 writes at the same cycle
  - The control logic can be complex.
  - Lose instruction fetch and issue opportunity.

# Sol 2: Delay R-type's Write by One Cycle

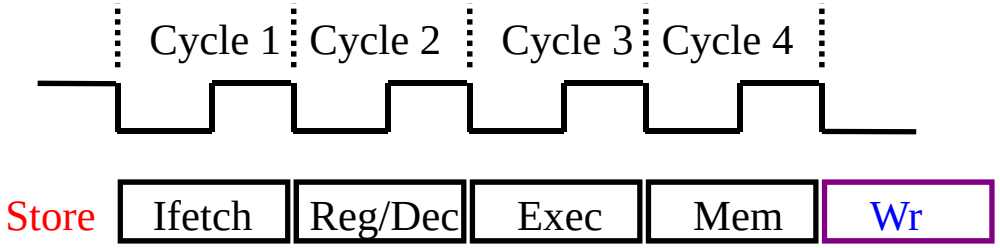
- 5-step R-type instructions:
  - Mem step for R-type inst. is a NOOP : nothing is being done.

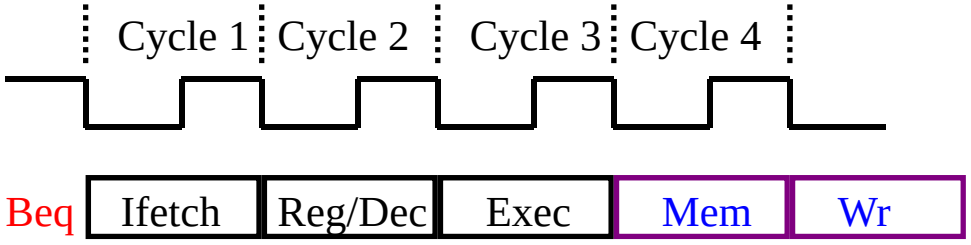



A unified pipeline architecture for each type of instructions !!

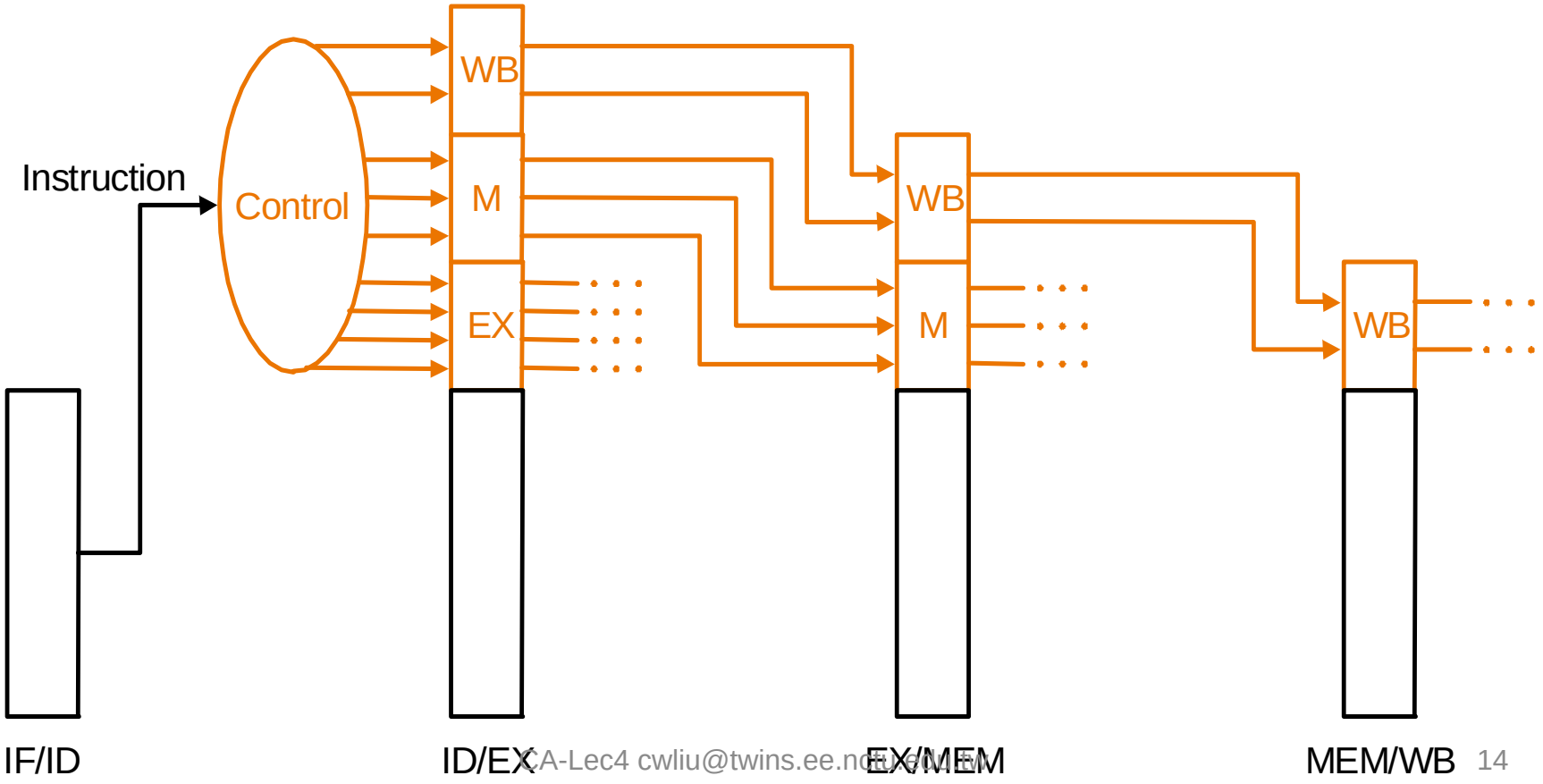
# Similarly, for Store and Branch Instructions



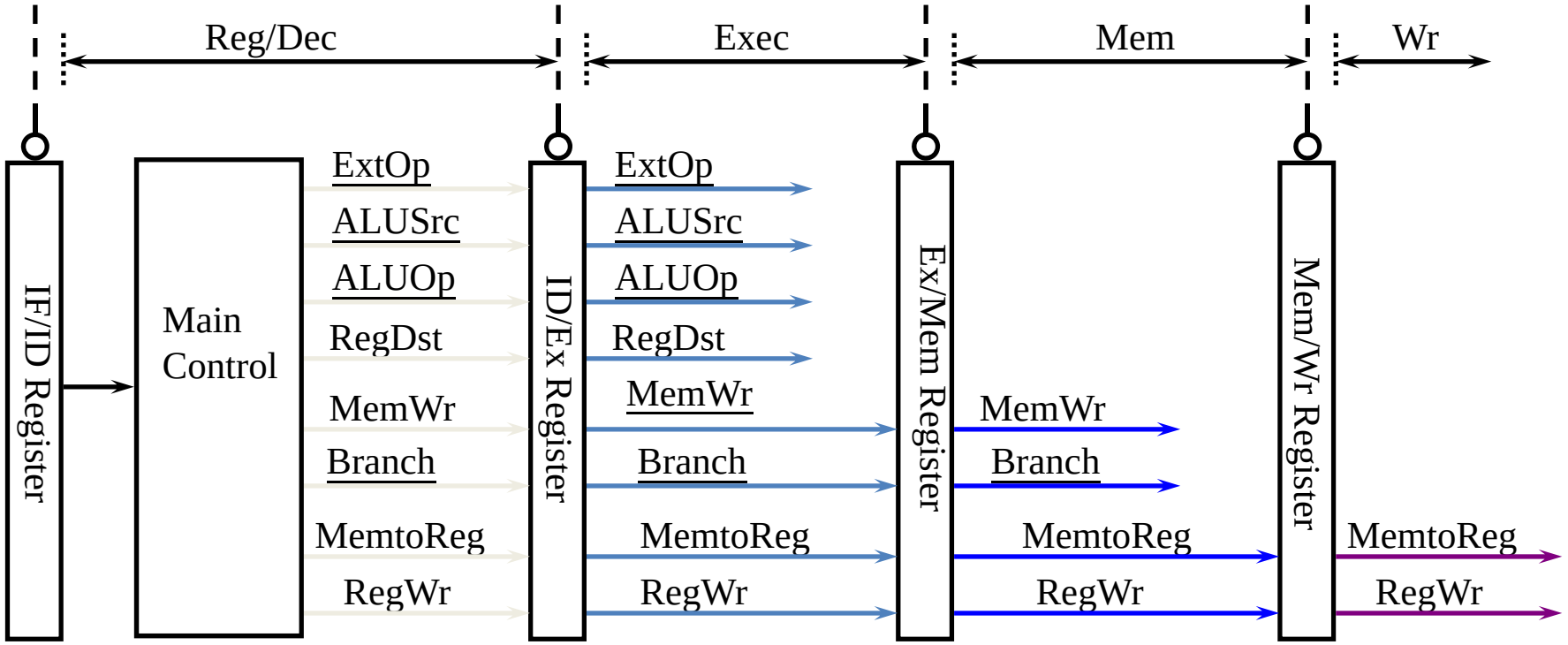
# Data Stationary Control

- Pass control signals along just like the data
  - Main control generates control signals during ID



# Use of "Data Stationary Control"

- The Main Control generates the control signals during Reg/Dec
  - Control signals for Exec (ExtOp, ALUSrc, ...) are used 1 cycle later
  - Control signals for Mem (MemWr Branch) are used 2 cycles later
  - Control signals for Wr (MemtoReg MemWr) are used 3 cycles later



# Pipeline Summary

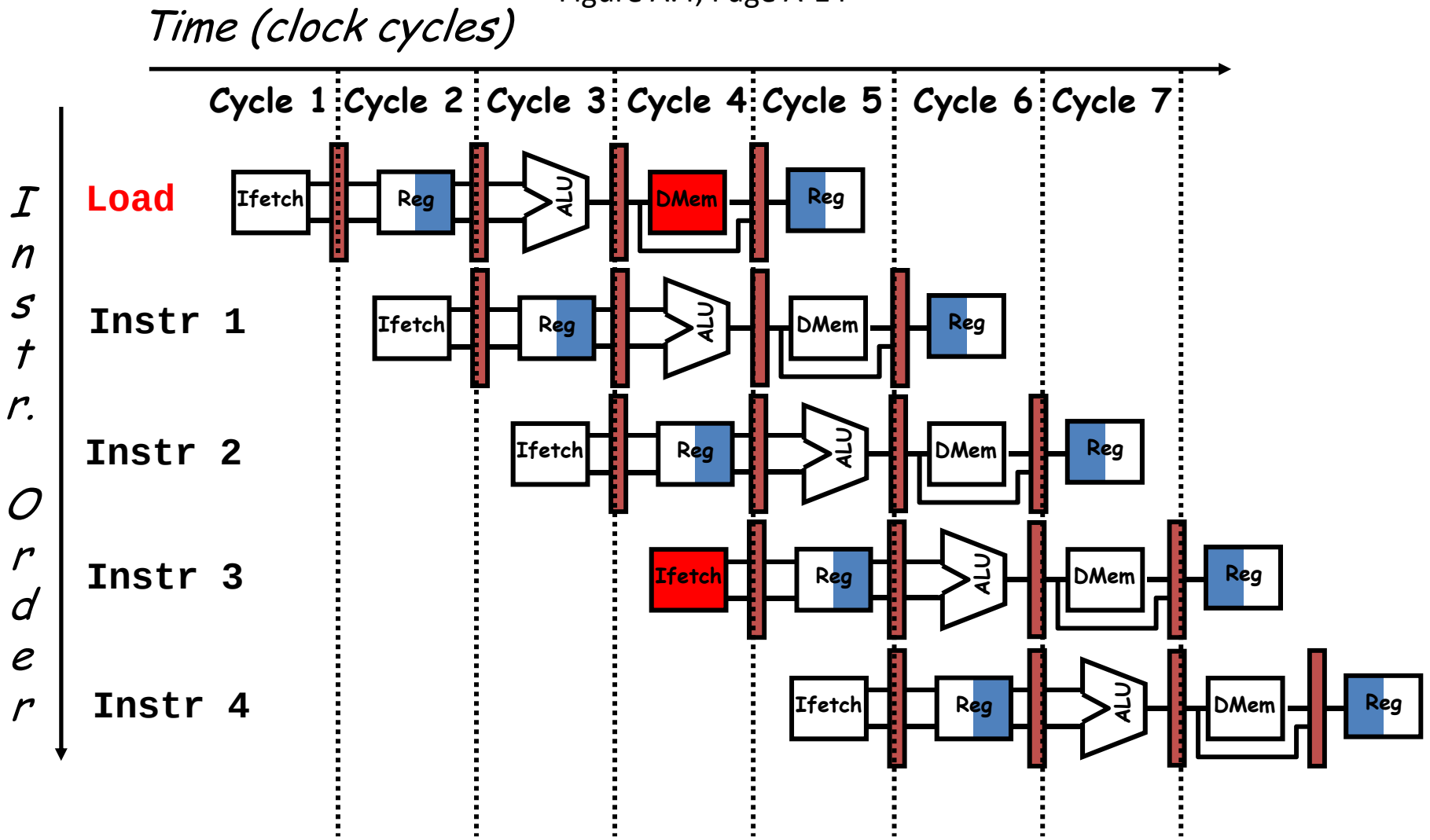
- A pipeline is like an hooked assembly line.
- Pipelining, in general, is not visible to the programmer (vs ILP)
- Pipelining doesn't help latency of single task, it helps throughput of entire workload
- Pipeline rate limited by slowest pipeline stage
- Multiple tasks operating simultaneously using different resources
- Potential speedup = Number pipe stages, if perfectly balanced stage.
- Unbalanced lengths of pipe stages reduces speedup
- Time to "fill" pipeline and time to "drain" it reduces speedup
- Pipeline hazard

# Pipelining is not quite that easy!

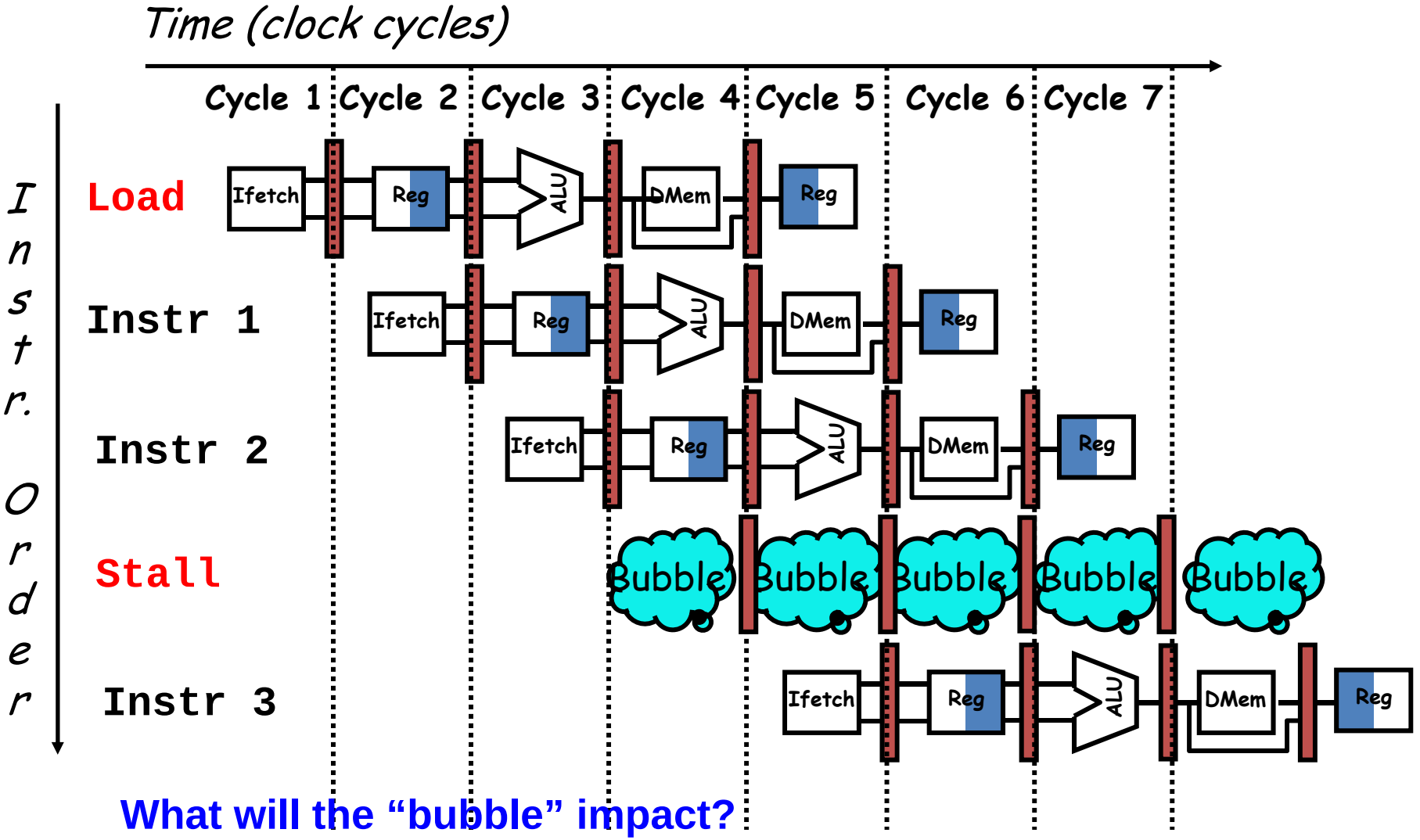
- Limits to pipelining: Hazards prevent next instruction from executing during its designated clock cycle
  - Structural hazards: HW cannot support this combination of instructions (single person to fold and put clothes away)
  - Data hazards: Instruction depends on result of prior instruction still in the pipeline (missing sock)
  - Control hazards: Caused by delay between the fetching of instructions and decisions about changes in control flow (branches and jumps).

# One Memory Port/Structural Hazards

Figure A.4, Page A-14



# One Memory Port/Structural Hazards



# Speed Up Equations for Pipelining

$$Speedup = \frac{Average\ instruction\ \ time\ unpipelined}{Average\ instruction\ \ time\ pipelined} = \frac{CPI_{unpipelined}}{CPI_{pipelined}} \times \frac{Clock\ cycle\ unpipelined}{Clock\ cycle\ pipelined}$$

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

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

for balanced pipelining

For simple RISC pipeline, CPI = 1:

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

# Example: Dual-port vs. Single-port

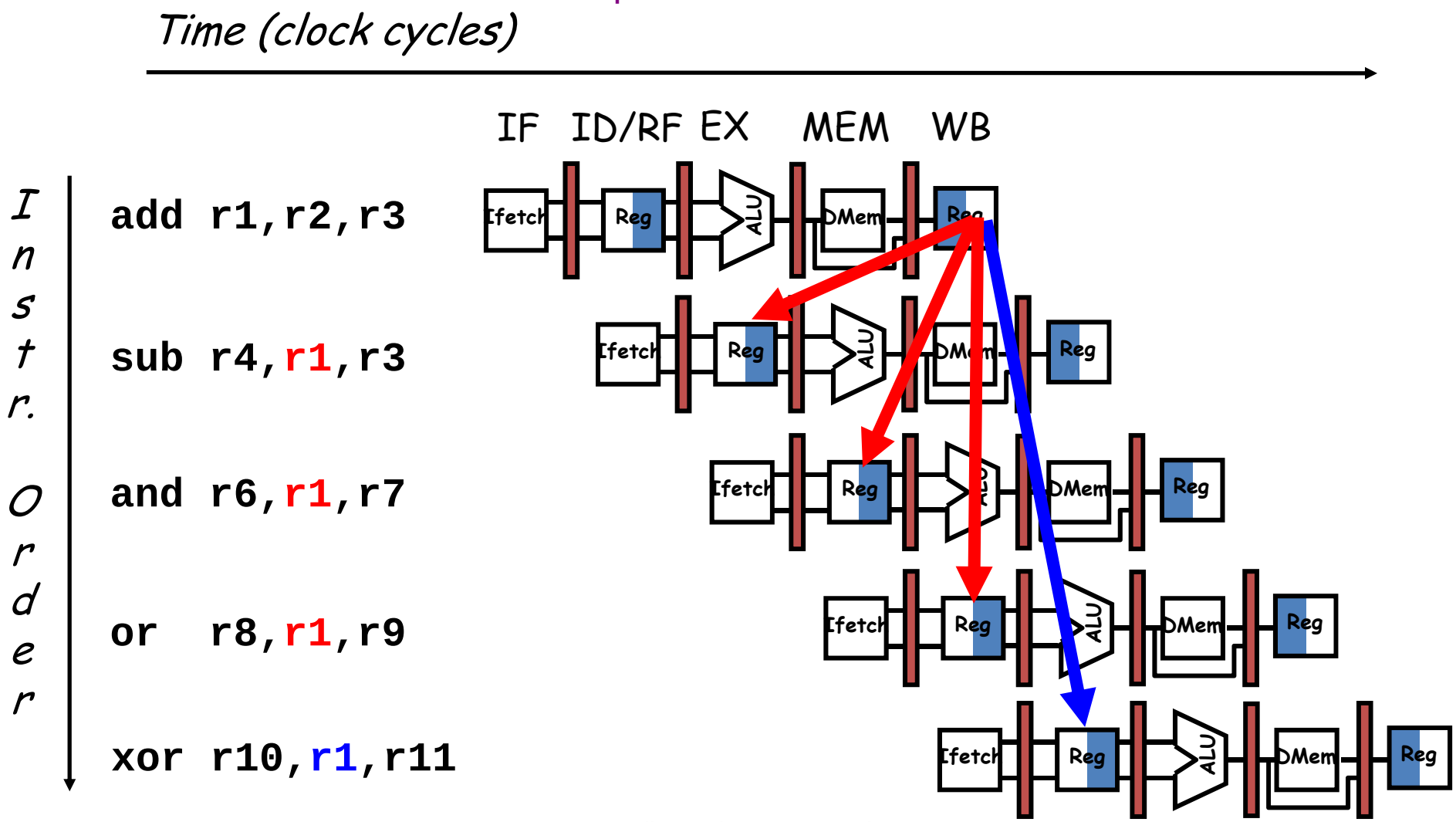
- Machine A: Dual ported memory ("Harvard Architecture")
- Machine B: Single ported memory, but its pipelined implementation has a 1.05 times faster clock rate
- Ideal CPI = 1 for both
- Suppose that Loads are 40% of instructions executed

$$\text{SpeedUp}_A = \text{Pipeline Depth}/(1 + 0) \times (\text{clock}_{\text{unpipe}}/\text{clock}_{\text{pipe}})$$
$$= \text{Pipeline Depth}$$
$$\text{SpeedUp}_B = \text{Pipeline Depth}/(1 + 0.4 \times 1) \times (\text{clock}_{\text{unpipe}}/(\text{clock}_{\text{unpipe}} / 1.05)$$
$$= (\text{Pipeline Depth}/1.4) \times 1.05$$
$$= 0.75 \times \text{Pipeline Depth}$$
$$\text{SpeedUp}_A / \text{SpeedUp}_B = \text{Pipeline Depth}/(0.75 \times \text{Pipeline Depth}) = 1.33$$

- Machine A is 1.33 times faster

# Data Hazard Problem on r1

Dependencies backwards in time are hazards

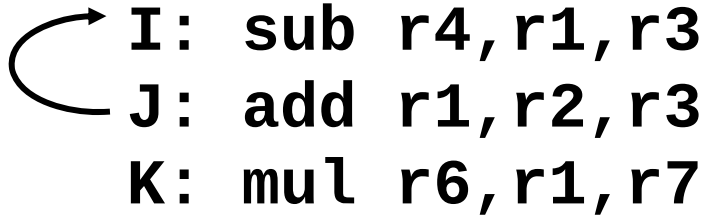


# Types of Data Hazards

- RAW (read after write): true data dependence
  - Get wrong propagation result
- WAR (write after read): anti-dependence
  - Get wrong operand
- WAW (write after write): output dependence
  - Leave wrong result

# Name Dependence Data Hazards

- Write After Read (WAR)
  $Instr_J$ writes operand *before* $Instr_I$ reads it



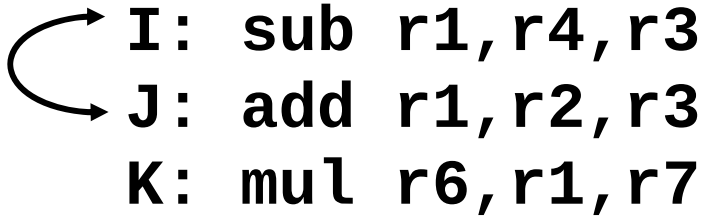
```
I: sub r4,r1,r3
J: add r1,r2,r3
K: mul r6,r1,r7
```

- Called an "anti-dependence" by compiler writers.
- This results from reuse of the name "r1".

- Can't happen in MIPS 5 stage pipeline because:
  - All instructions take 5 stages. Reads are always in stage 2 and Writes are always in stage 5
- Can happen in OOO processor (discussed in chapter 3).

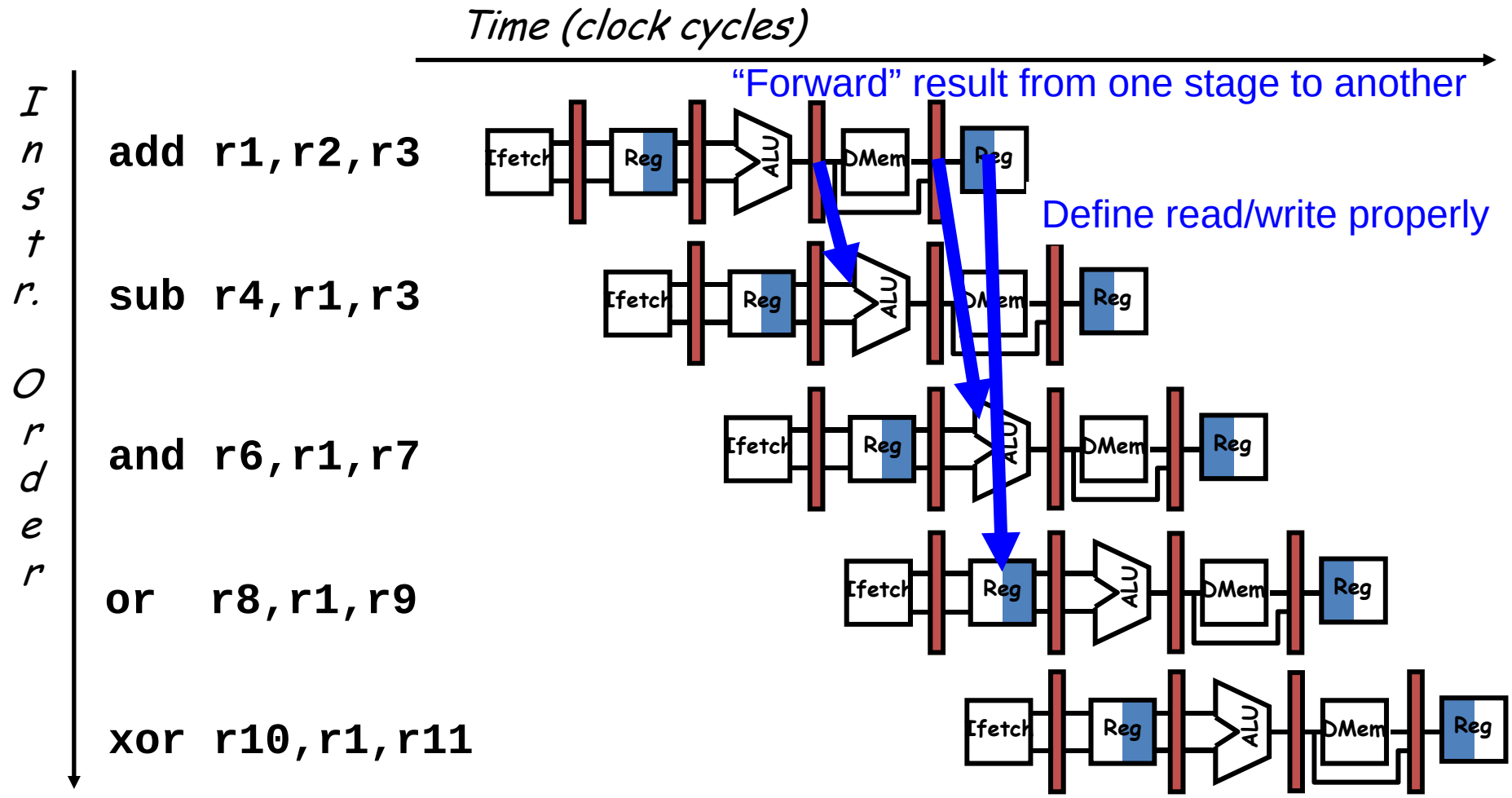
# Name Dependence Data Hazards

- Write After Write (WAW)
  Instr$_J$ writes operand *before* Instr$_I$ writes it.



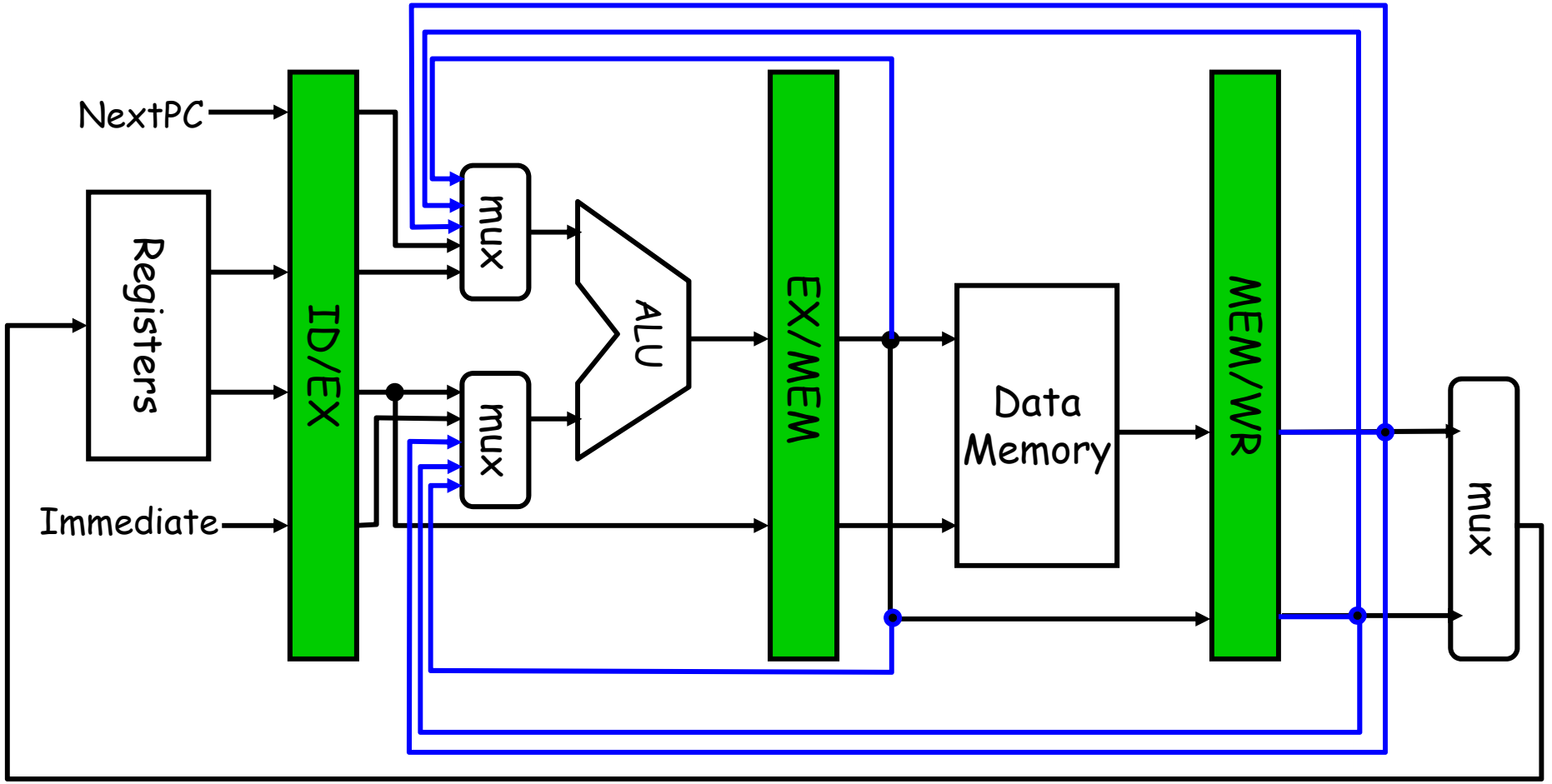

- Called an "output dependence" by compiler writers
- This also results from the reuse of name "r1".
- Can't happen in MIPS 5 stage pipeline because:
  - All instructions take 5 stages, and Writes are always in stage 5
- Will see WAR and WAW in more complicated pipes and OOO processor

# Forwarding to Avoid Data Hazard



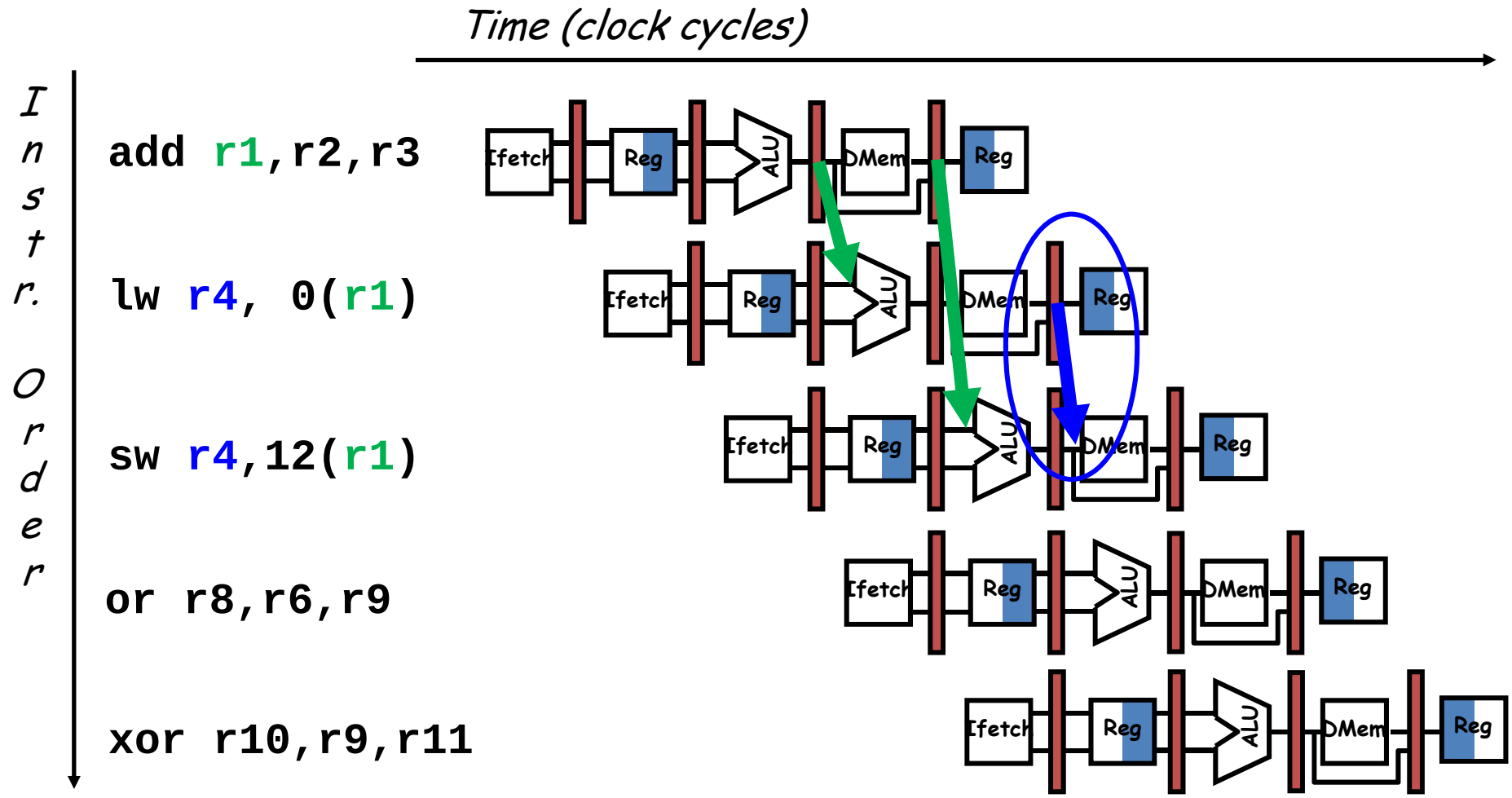
# HW Change for Forwarding

Additional hardware is required.

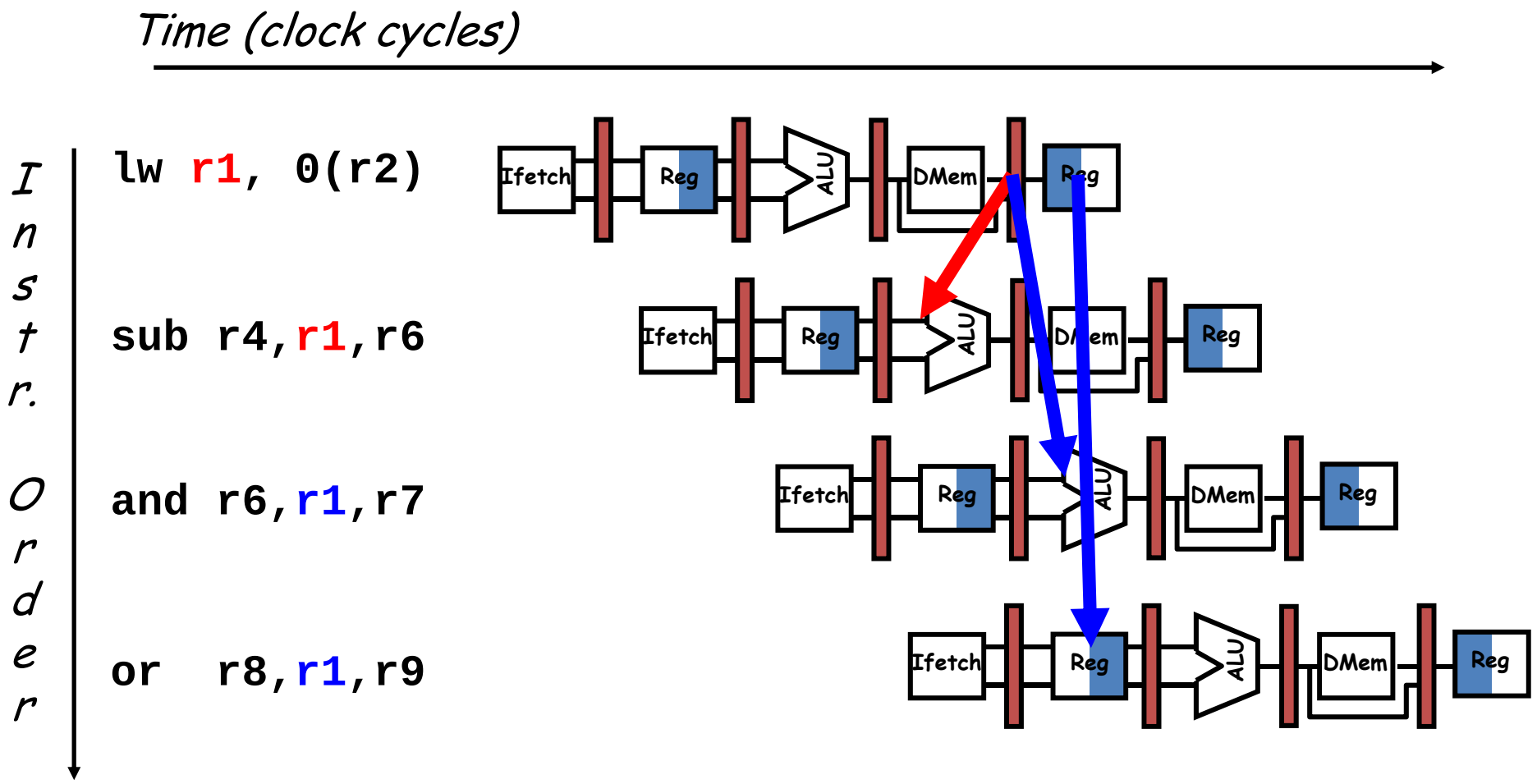



**What circuit detects and resolves this hazard?**

# Forwarding to Avoid LW-SW Data Hazard

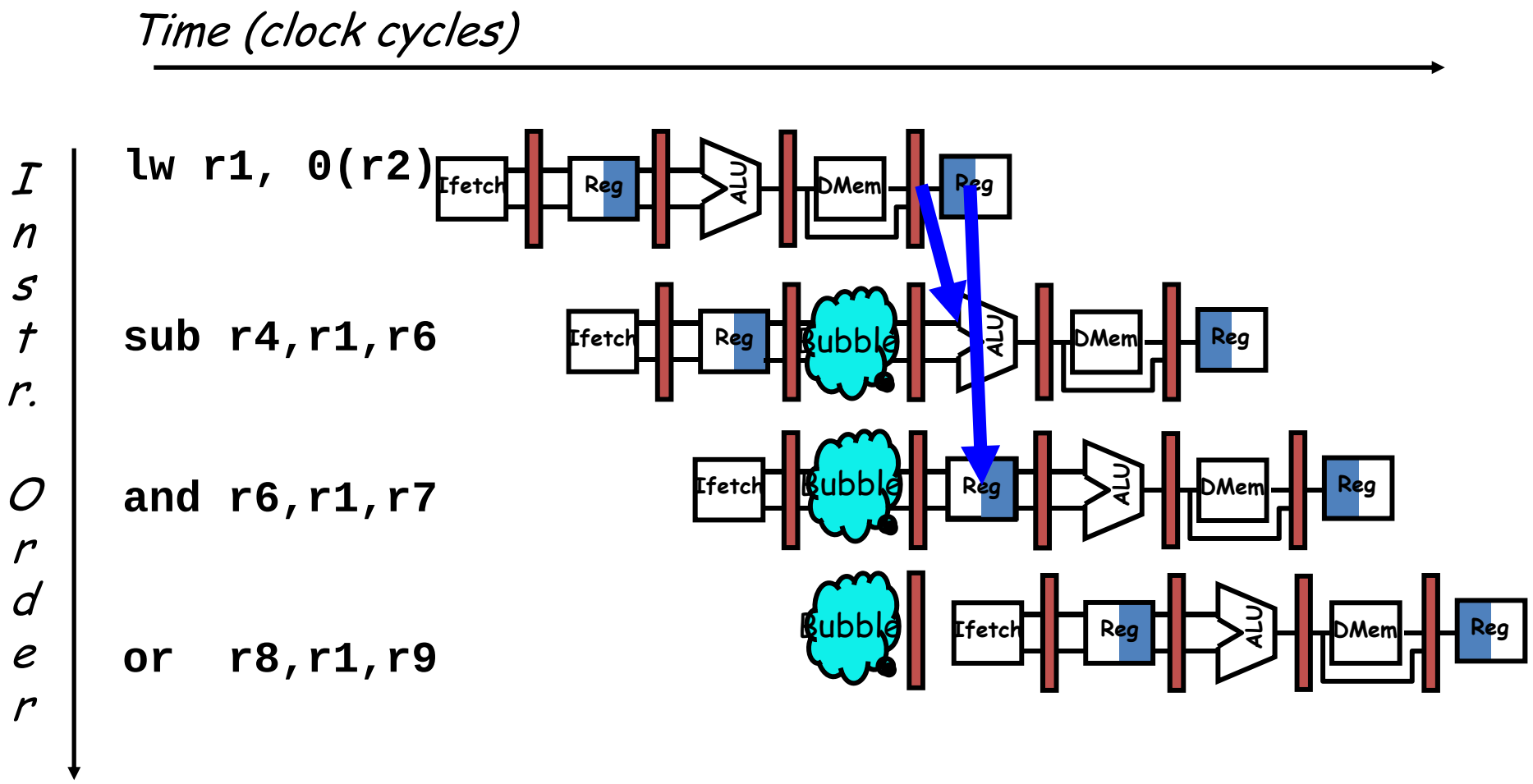


# Data Hazard Even with Forwarding




**Must delay/stall instruction dependent on loads: Load stall cycles**

# Data Hazard Even with Forwarding



# Software Scheduling to Avoid Load Hazards

Try producing fast code for

a = b + c;

d = e – f;

assuming a, b, c, d ,e, and f in memory.

Slow code:

```
LW    Rb,b
LW    Rc,c
ADD   Ra,Rb,Rc
SW    a,Ra
LW    Re,e
LW    Rf,f
SUB   Rd,Re,Rf
SW    d,Rd
```

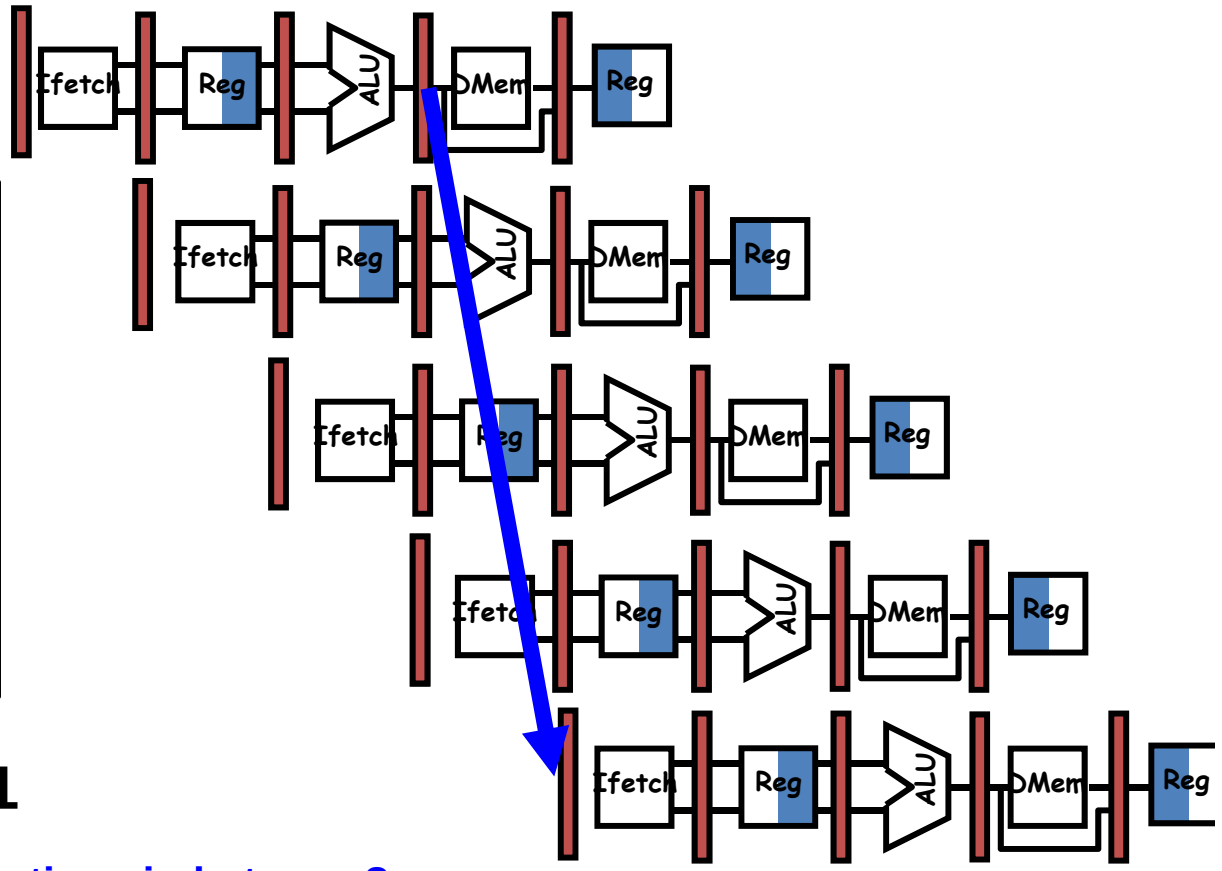
Fast code:

```
LW    Rb,b
LW    Rc,c
LW    Re,e
ADD   Ra,Rb,Rc
LW    Rf,f
SW    a,Ra
SUB   Rd,Re,Rf
SW    d,Rd
```

Compiler optimizes for performance. Hardware checks for safety.

# Control Hazard on Branches

```
10: beq r1,r3,36
14: and r2,r3,r5
18: or  r6,r1,r7
22: add r8,r1,r9
36: xor r10,r1,r11
```



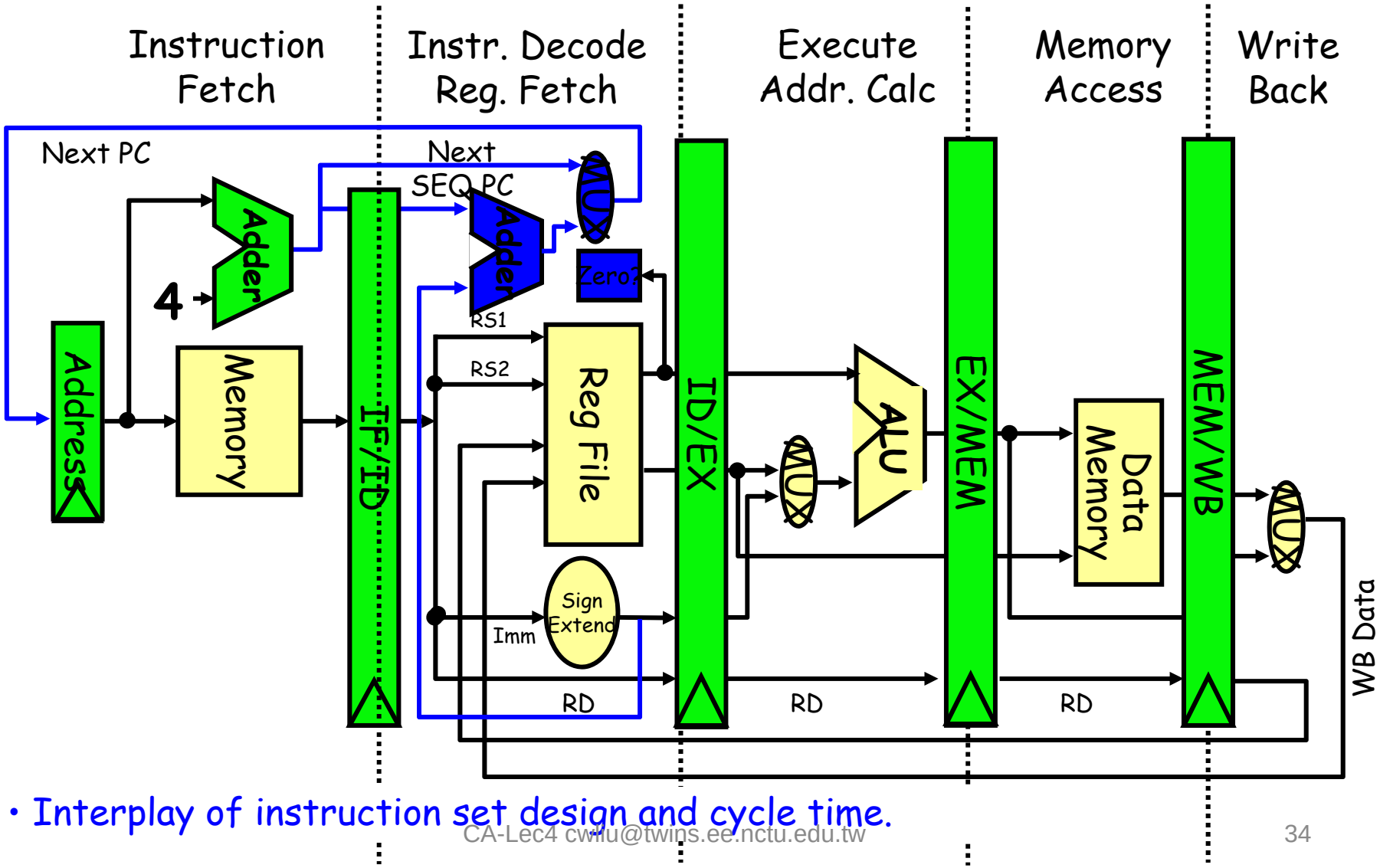

**What do you do with the 3 instructions in between?**

The simplest solution is to stall the pipeline as soon as a branch instruction is detected

# Branch Stall Impact

- If CPI = 1, 30% branch,
  Stall 3 cycles => new CPI = 1.9!
- Two-part solution:
  - Determine branch taken or not sooner, AND
  - Compute taken branch address earlier
- MIPS branch tests if register = 0 or ≠ 0
- MIPS 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

# Pipelined MIPS Datapath




- Interplay of instruction set design and cycle time.

# 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% MIPS branches not taken on average
- PC+4 already calculated, so use it to get next instruction

#3: Predict Branch Taken

- 53% MIPS branches taken on average
- But haven't calculated branch target address in MIPS
  - MIPS still incurs 1 cycle branch penalty
  - Other machines: branch target known before outcome
- What happens when hit not-taken branch?

# Four Branch Hazard Alternatives

#4: Delayed Branch – make the stall cycle useful

- Define branch to take place AFTER a following instruction

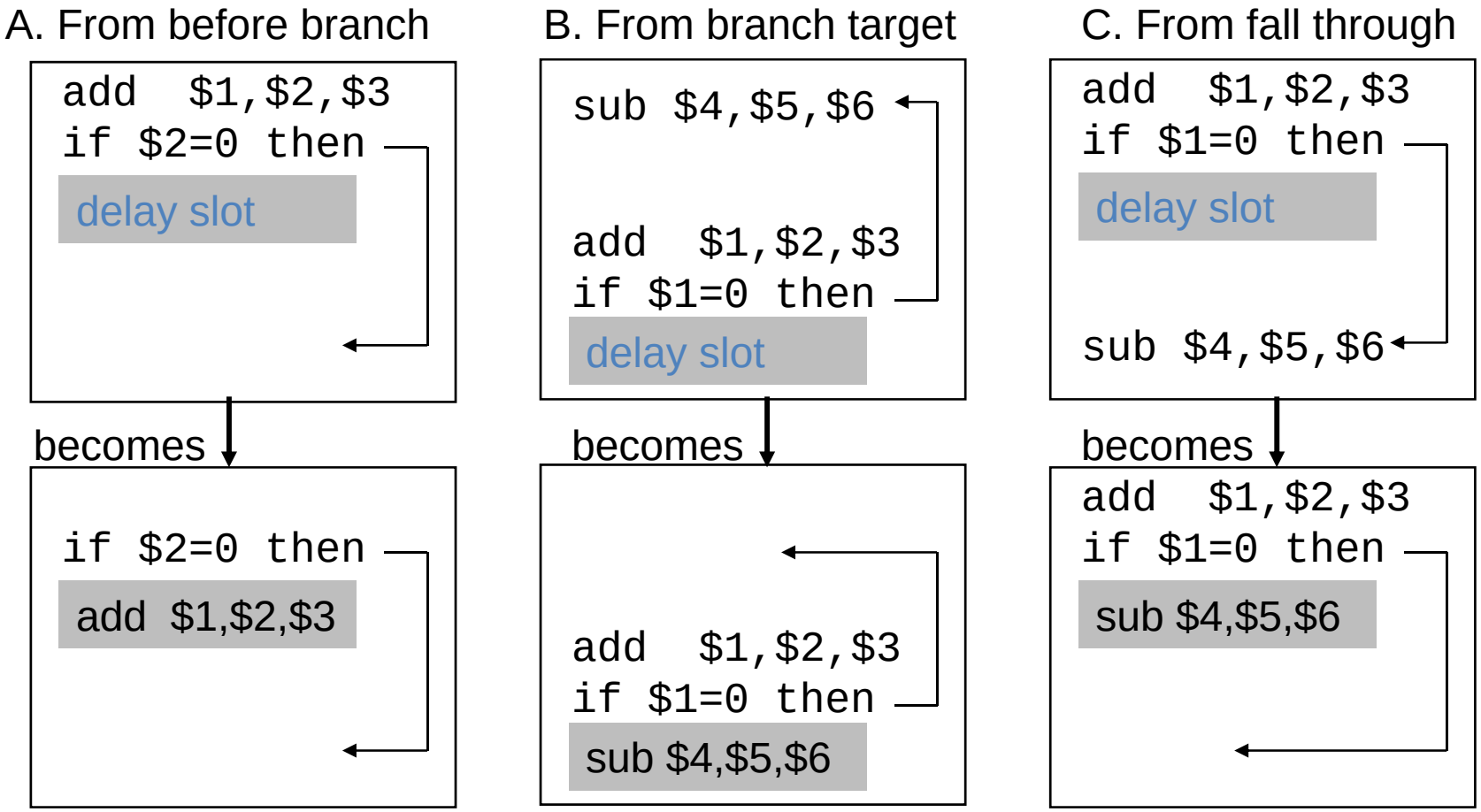
```
branch instruction
 sequential successor1
 sequential successor2
 ........
 sequential successorn
branch target if taken
```

Branch delay of length *n*

These insts. are executed !!

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

# 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 when branch fails

# Delay-Branch Scheduling Schemes and Their Requirements

| Scheduling Strategy | Requirements | Improve Performance When? |
|---|---|---|
| From before | Branch must not depend on the rescheduled instructions | Always |
| From target | Must be OK to execute rescheduled instructions if branch is not taken. May need to duplicate instructions | When branch is taken. May enlarge program if instructions are duplicated |
| From fall through | Must be OK to execute instructions if branch is taken | When branch is not taken. |

# 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 slots
  - Delayed branch (a static way) has lost popularity compared to more expensive but more flexible dynamic approaches
  - Growth in available transistors has made dynamic approaches relatively cheaper

# Performance of Branch Schemes Example

- For MIPS R4K, a deeper pipeline, it takes at least 3 pipeline stages before the branch-target address is known and an additional cycle before the branch condition is evaluated.

| Scheme | $Penalty_{Uncond}$ | $Penalty_{untaken}$ | $Penalty_{taken}$ |
|---|---|---|---|
| Stall | 2 | 3 | 3 |
| Predicted taken | 2 | 3 | 2 |
| Predicted not taken | 2 | 0 | 3 |

# Evaluating Branch Alternatives

$$\text{Pipeline speedup} = \frac{\text{Pipeline depth}}{1 + \text{Branch frequency} \times \text{Branch penalty}}$$

Assume: 4% unconditional branch, 6% conditional branch- untaken, 10% conditional branch-taken

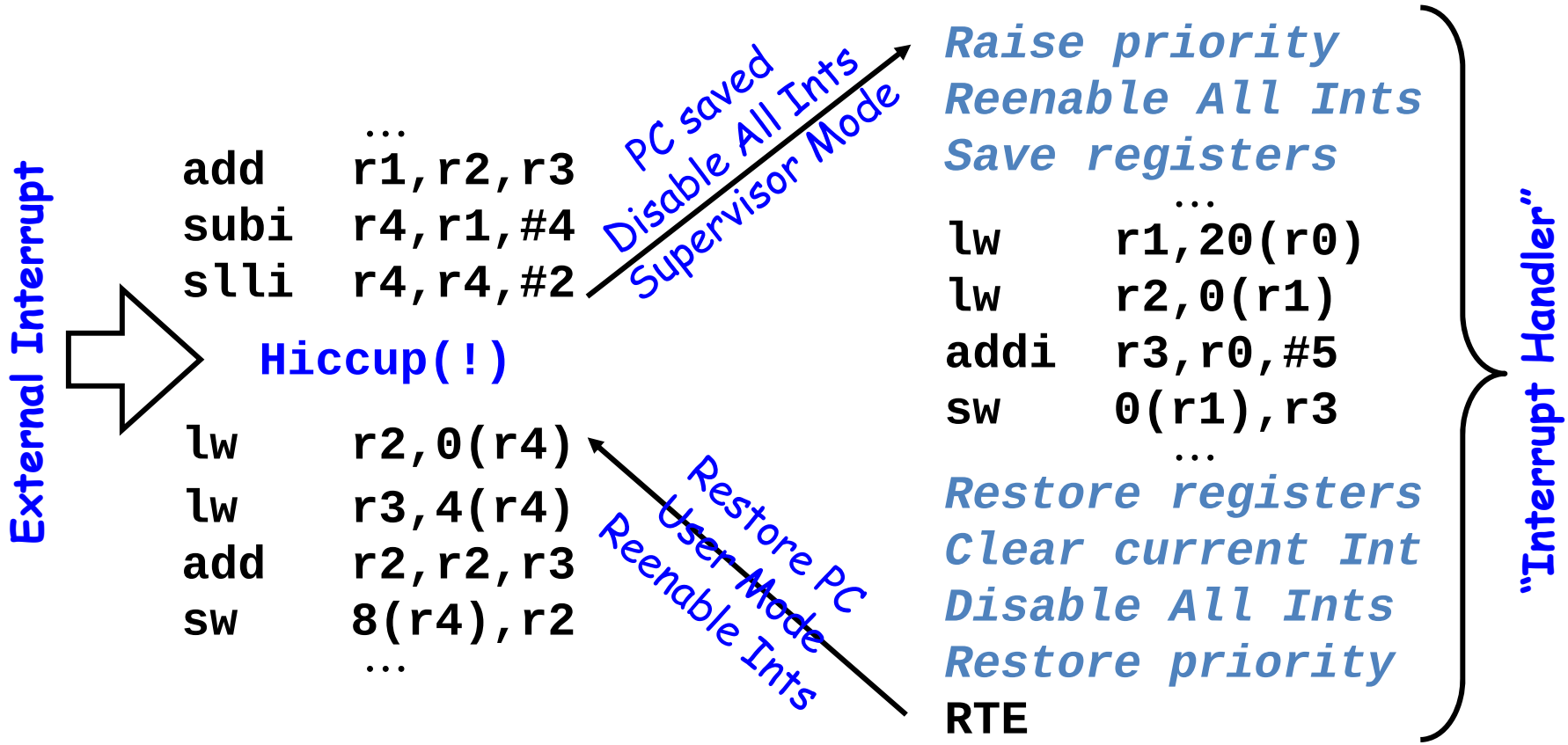
| *Scheduling scheme* | *Branch penalty* | *CPI* | *speedup v. unpipelined* | *speedup v. stall* |
|---|---|---|---|---|
| **Stall pipeline** | 2x0.04+3x0.16 | 1.56 | n/1.56 | 1.0 |
| **Predict not taken** | 2x0.04+3x0.06+2x0.1 | 1.46 | n/1.46 | 1.07 |
| **Predict taken** | 2x0.04+0x0.06+3x0.1 | 1.38 | n/1.38 | 1.13 |
| **Delayed branch** | 0.5x(0.04+0.16) | 1.10 | n/1.1 | 1.42 |

# Problems with Pipelining

- 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
  - Example: a sound card interrupts when it needs more audio output samples (an audio "click" happens if it is left waiting)
- 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 totalling complete
  - No effect of any instruction after $I_i$ can take place
- The interrupt (exception) handler either aborts program or restarts at instruction $I_{i+1}$

# Example: Device Interrupt

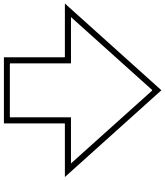
(Say, arrival of network message)



# Alternative: Polling

(again, for arrival of network message)

External Interrupt



```
        Disable Network Intr
              ...
        subi  r4,r1,#4
        slli  r4,r4,#2
        lw    r2,0(r4)
        lw    r3,4(r4)
        add   r2,r2,r3
        sw    8(r4),r2
        lw    r1,12(r0)      }  Polling Point
        beq   r1,no_mess     }  (check device register)
        lw    r1,20(r0)      }
        lw    r2,0(r1)       }
        addi  r3,r0,#5       }  "Handler"
        sw    0(r1),r3       }
        Clear Network Intr   }
no_mess:  ...
```

# Polling is faster/slower than Interrupts.

- Polling is faster than interrupts because
  - Compiler knows which registers in use at polling point. Hence, do not need to save and restore registers (or not as many).
  - Other interrupt overhead avoided (pipeline flush, trap priorities, etc).
- Polling is slower than interrupts because
  - Overhead of polling instructions is incurred regardless of whether or not handler is run. This could add to inner-loop delay.
  - Device may have to wait for service for a long time.
- When to use one or the other?
  - Multi-axis tradeoff
    - Frequent/regular events good for polling, *as long as device can be controlled at user level.*
    - Interrupts good for infrequent/irregular events
    - Interrupts good for ensuring regular/predictable service of events.

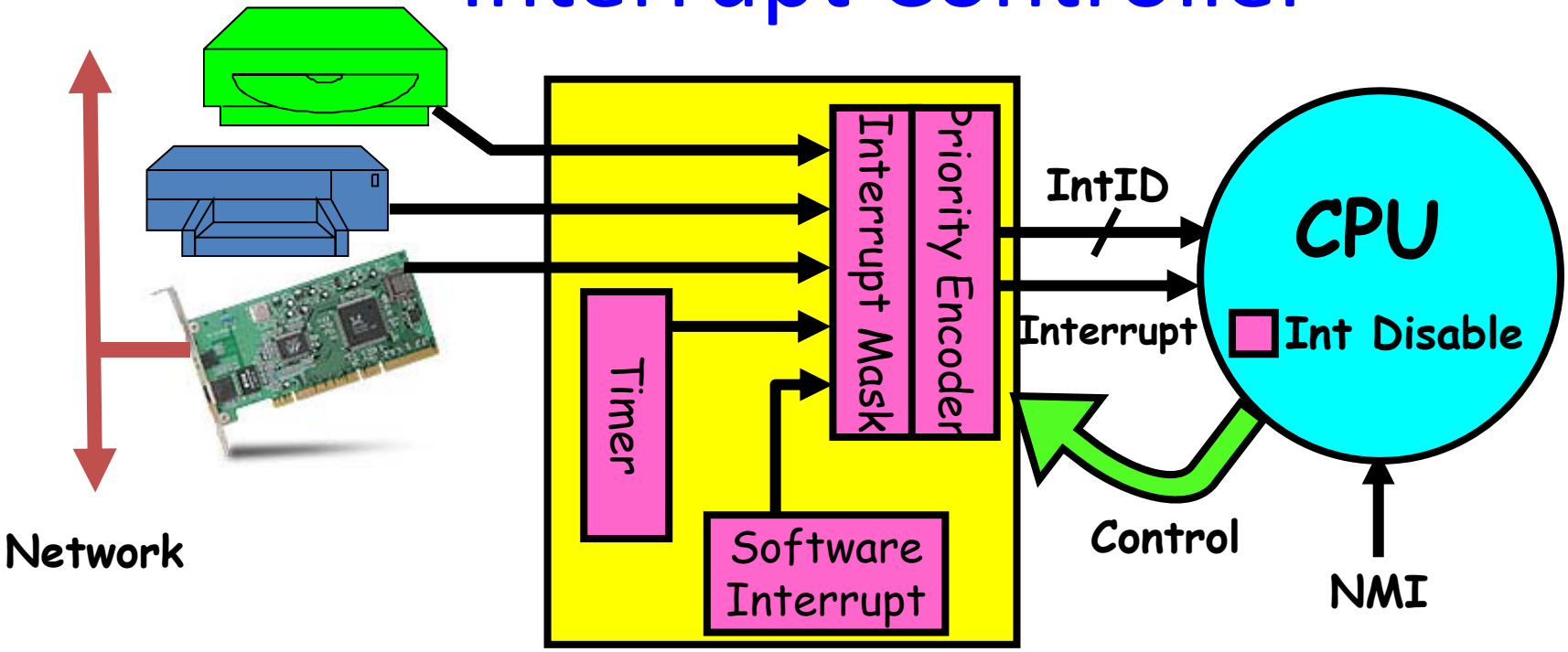
# Trap/Interrupt Classifications

- *Traps:* relevant to the current process
  - Faults, arithmetic traps, and synchronous traps
  - Invoke software on behalf of the currently executing process
- *Interrupts:* caused by asynchronous, outside events
  - I/O devices requiring service (DISK, network)
  - Clock interrupts (real time scheduling)
- *Machine Checks:* caused by serious hardware failure
  - Not always restartable
  - Indicate that bad things have happened.
    - Non-recoverable ECC error
    - Machine room fire
    - Power outage

# A related classification: Synchronous vs. Asynchronous

- *Synchronous:* means related to the instruction stream, i.e. during the execution of an instruction
  - Must stop an instruction that is currently executing
  - Page fault on load or store instruction
  - Arithmetic exception
  - Software Trap Instructions
- *Asynchronous:* means unrelated to the instruction stream, i.e. caused by an outside event.
  - Does not have to disrupt instructions that are already executing
  - Interrupts are asynchronous
  - Machine checks are asynchronous
- *SemiSynchronous (or high-availability interrupts):*
  - Caused by external event but may have to disrupt current instructions in order to guarantee service

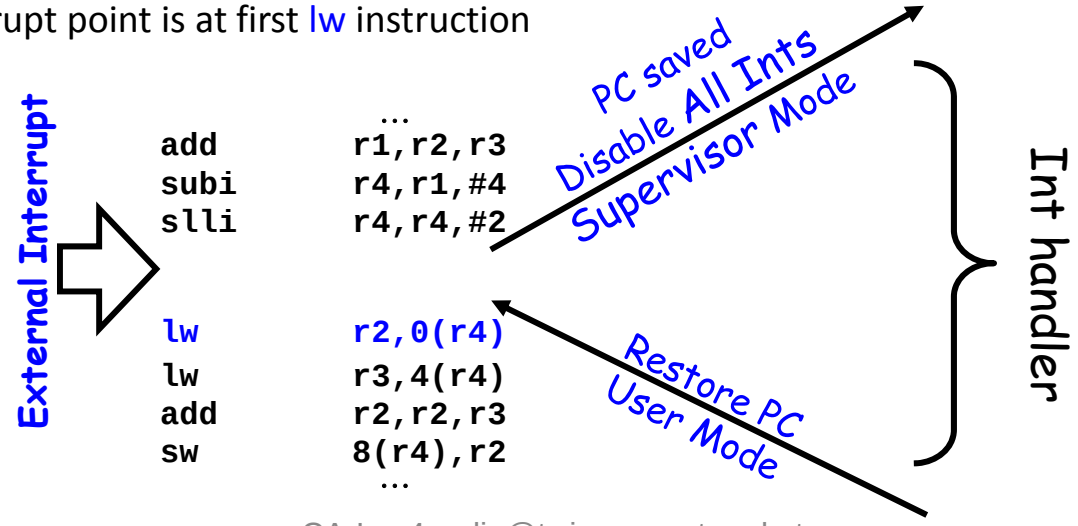
# Interrupt Controller




- Interrupts invoked with interrupt lines from devices
- Interrupt controller chooses interrupt request to honor
  - Mask enables/disables interrupts
  - Priority encoder picks highest enabled interrupt
  - Software Interrupt Set/Cleared by Software
  - Interrupt identity specified with ID line
- CPU can disable all interrupts with internal flag
- Non-maskable interrupt line (NMI) can't be disabled

# Precise Interrupts/Exceptions

- An interrupt or exception is considered *precise* if there is a single instruction (or interrupt point) for which:
  - All instructions before that have committed their state
  - No following instructions (including the interrupting instruction) have modified any state.
- This means that you can restart execution at the interrupt point and "get the right answer"
  - Implicit in our previous example of a device interrupt:
    - Interrupt point is at first lw instruction



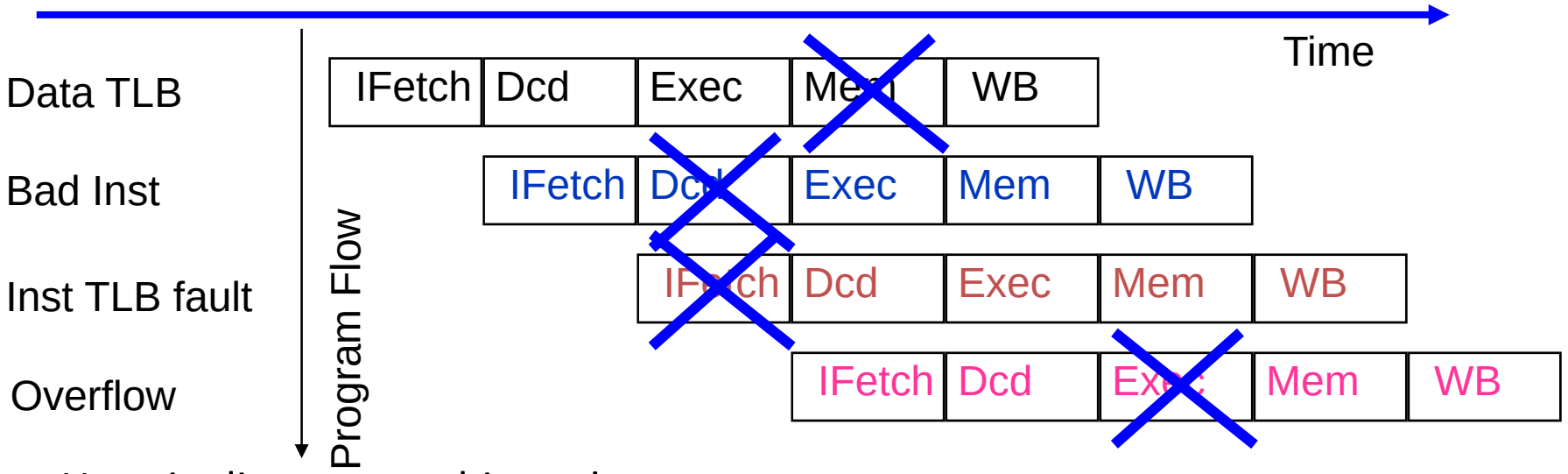
# Why are precise interrupts desirable?

- Many types of interrupts/exceptions need to be restartable, which is easier to figure out what actually happened:
  - e.g. TLB faults. Need to fix translation, then restart load/store
  - e.g. IEEE gradual underflow, illegal operation, etc
- Restartability doesn't require preciseness. However, preciseness makes it a lot easier to restart.
- Simplify the task of the operating system a lot
  - Less state needs to be saved away if unloading process.
  - Quick to restart (making for fast interrupts)

# Precise Exceptions in simple 5-stage pipeline:

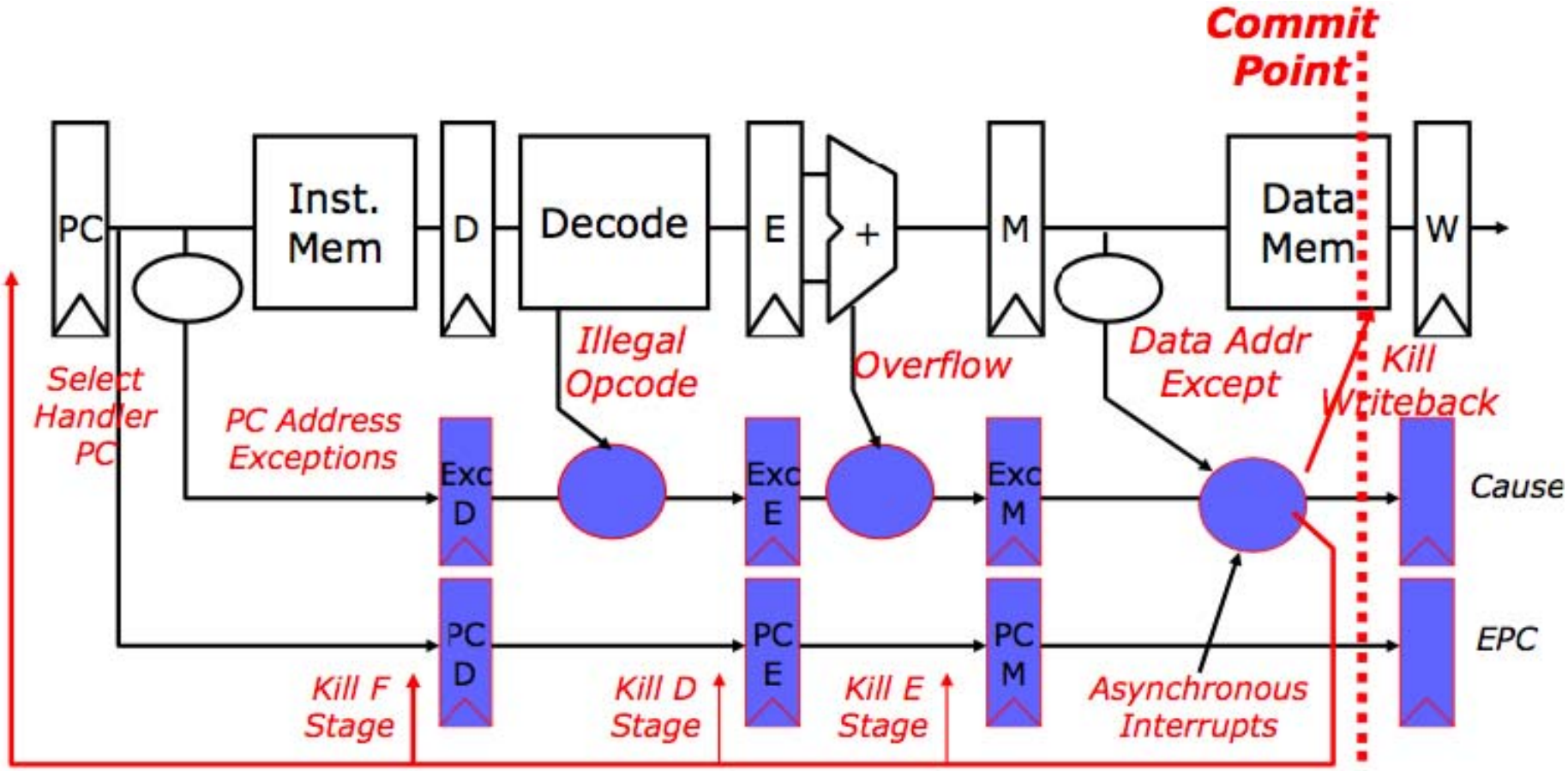
- Exceptions/interrupts may occur at different stages in pipeline :
  - Arithmetic exceptions occur in execution stage
  - TLB faults can occur in instruction fetch or memory stage
- What about interrupts?
  - try to interrupt the pipeline as little as possible
- All of this solved by tagging instructions in pipeline as “cause exception or not” and wait until end of memory stage to flag exception
  - Interrupts become marked NOPs (like bubbles) that are placed into pipeline instead of an instruction.
  - Assume that interrupt condition persists in case NOP flushed
  - Clever instruction fetch might start fetching instructions from interrupt vector, but this is complicated by need for supervisor mode switch, saving of one or more PCs, etc

# Another Exception Problem




- Use pipeline to sort this out!
  - Pass exception status along with instruction.
  - Keep track of PCs for every instruction in pipeline.
  - Don't act on exception until it reache WB stage
- Handle interrupts through "faulting noop" in IF stage
- When instruction reaches WB stage:
  - Save PC ⇒ EPC, Interrupt vector addr ⇒ PC
  - Turn all instructions in earlier stages into noops!

# Precise Exceptions in Static Pipelines




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