

---

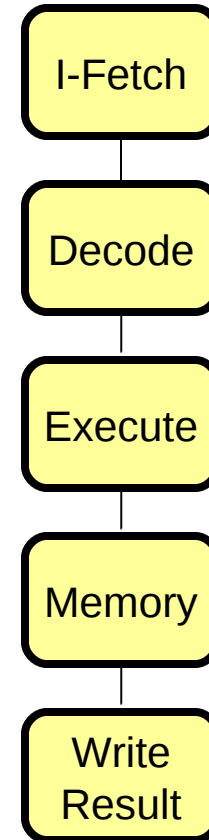
# Pipelining

Giorgio Richelli

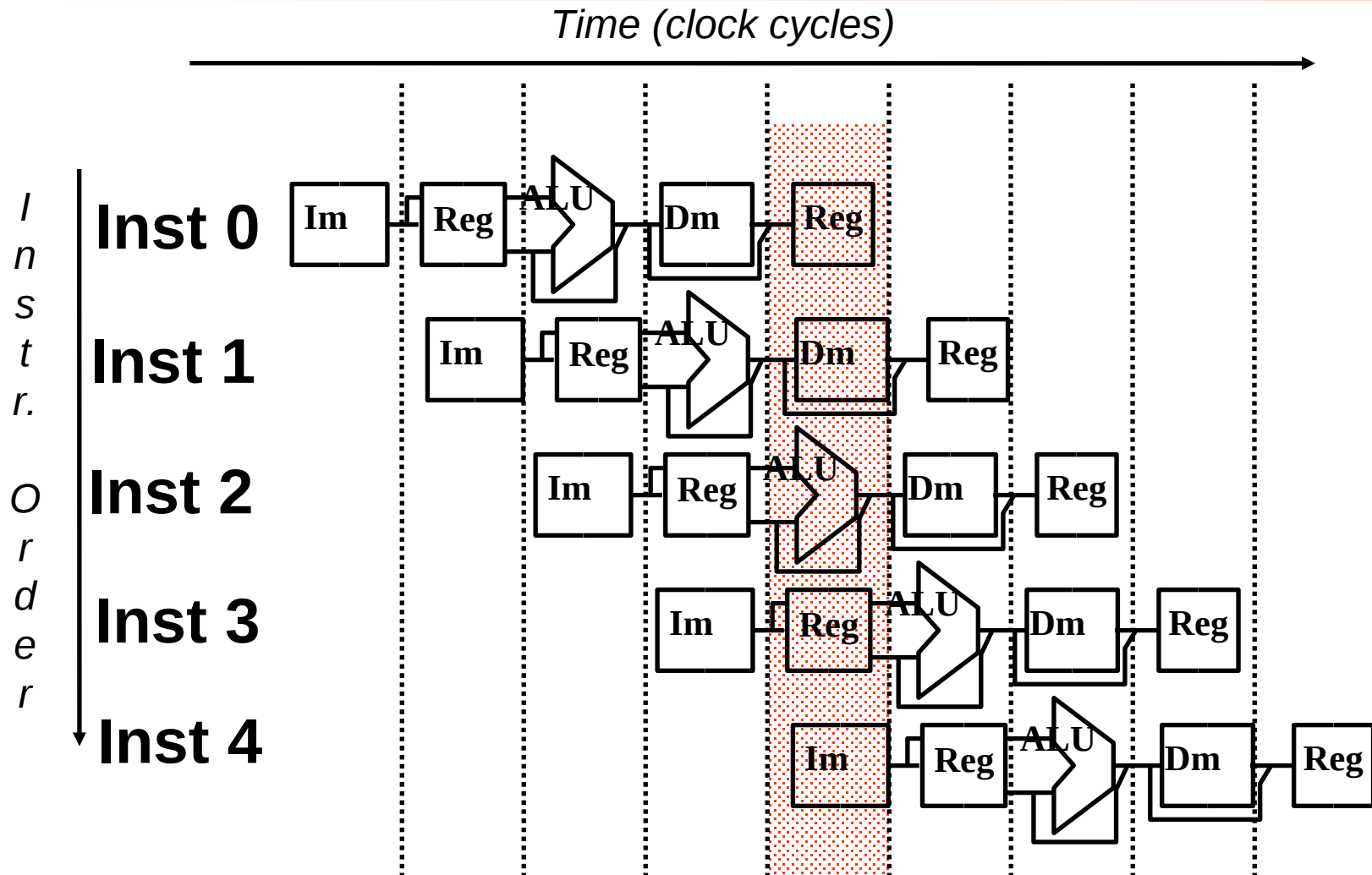
# Instruction Execution

---

- 5 basic steps
  - fetch instruction (F)
  - decode instruction/read registers (R)
  - execute (X)
  - access memory (M)
  - store result (W)



# Why Pipeline



# Why Pipeline?

---

- Suppose we execute 100 instructions
- Single Cycle Machine
  - $50 \text{ ns/cycle} \times 1 \text{ CPI} \times 100 \text{ inst} = 5000 \text{ ns}$
- Multicycle Machine
  - $10 \text{ ns/cycle} \times 5 \text{ CPI} \times 100 \text{ inst} = 5000 \text{ ns}$
- Ideal pipelined machine
  - $10 \text{ ns/cycle} \times (1 \text{ CPI} \times 100 \text{ inst} + 4 \text{ cycle drain}) = 1040 \text{ ns}$

# Benefits of Pipelining

---

- Before pipelining:

- Throughput: 1 instruction per cycle

$$t_{clk} = t_F + t_R + t_X + t_M + t_W$$

- (or lower cycle time and CPI=5)

- After pipelining (multiple instructions in pipe at one time )

- Throughput: 1 instruction per cycle

$$t_{clk} = \max(t_F, t_R, t_X, t_M, t_W) + t_{latch}$$

# Pipeline Hazards

---

- **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).

# Pipeline Hazards

---

- Data hazards

- an instruction uses the result of a previous instruction

```
ADD  R1, R2, R3  or  SW  R1, 3(R2)
ADD  R4, R1, R5      LW  R3, 3(R2)
```

- Control hazards

- the location of an instruction depends on a previous instruction

```
JMP R25
```

```
...
```

```
LOOP: ADD  R1, R2, R3
```

- Structural hazards

- two instructions need access to the same resource
  - e.g., single unit shared for instruction fetch and load/store
  - collision in reservation table

# Data Hazards: RAW

---

- Read After Write (RAW)

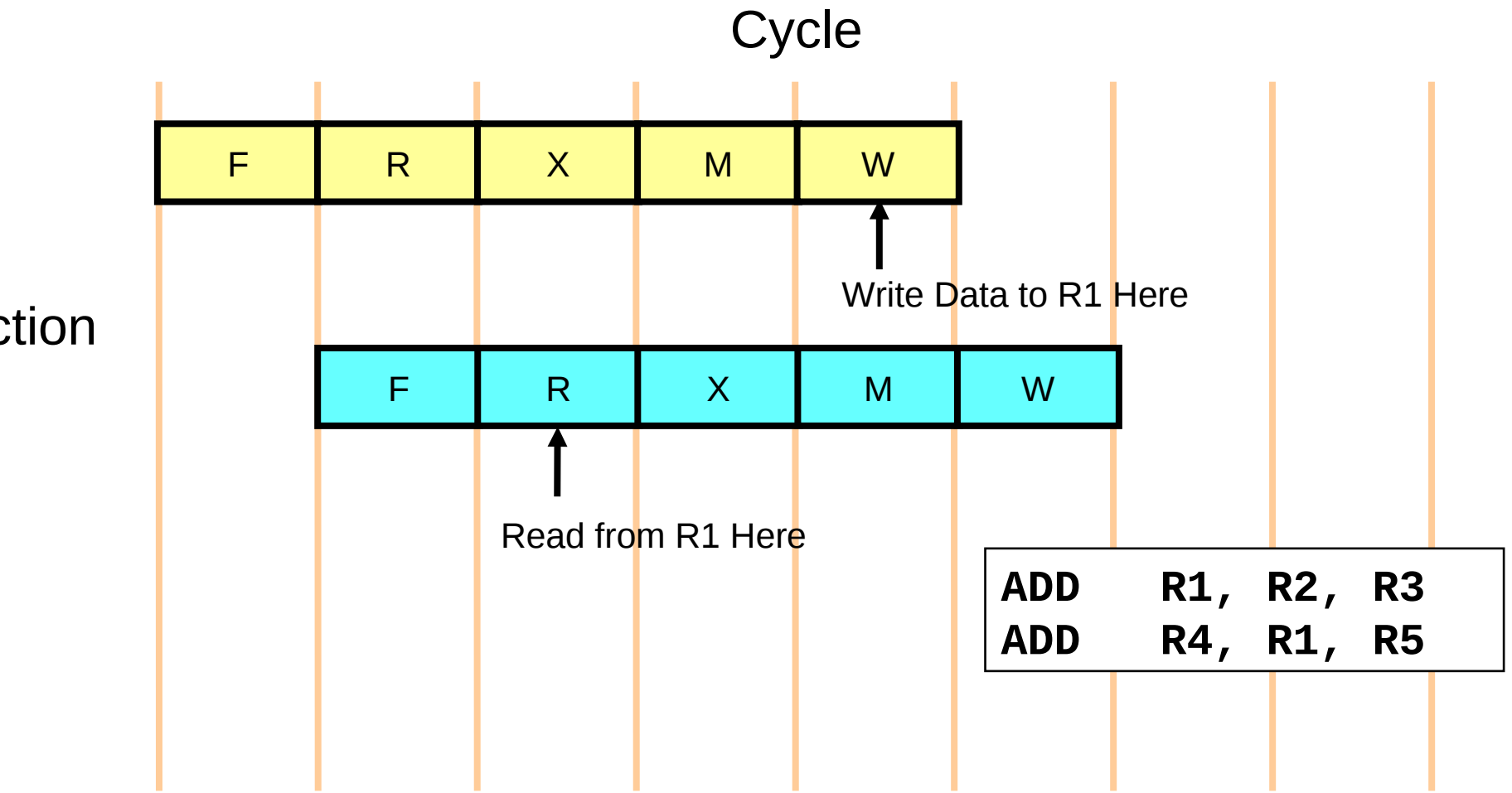
Instr<sub>j</sub> tries to read operand before Instr<sub>i</sub> writes it

 **I: add r1, r2, r3**  
**J: sub r4, r1, r3**

- Caused by a “Dependence” (in compiler nomenclature).
- This hazard results from an actual need for communication between pipeline stages.



# Data Hazards (RAW)

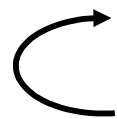


# Data Hazards: WAR

---

- Write After Read (WAR)

Instr<sub>j</sub> writes operand before Instr<sub>i</sub> reads it

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

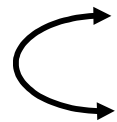
- Called an “anti-dependence” by compiler writers. This results from reuse of the name “r1”.
- Can’t happen in the sample 5-stage pipeline because:
  - All instructions take 5 stages, and
  - Reads are always in stage 2, and
  - Writes are always in stage 5

# Data Hazards: WAW

---

- Write After Write (WAW)

Instr<sub>j</sub> writes operand before Instr<sub>i</sub> writes it.

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

- Called an “output dependence” by compiler writers

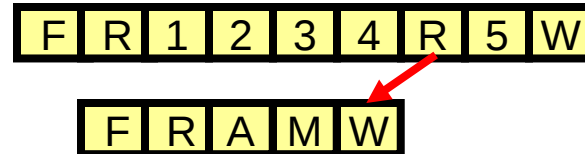
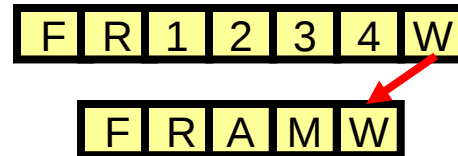
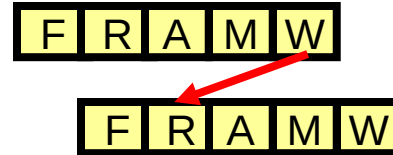
This also results from the reuse of name “r1”.

- Can be solved by register renaming

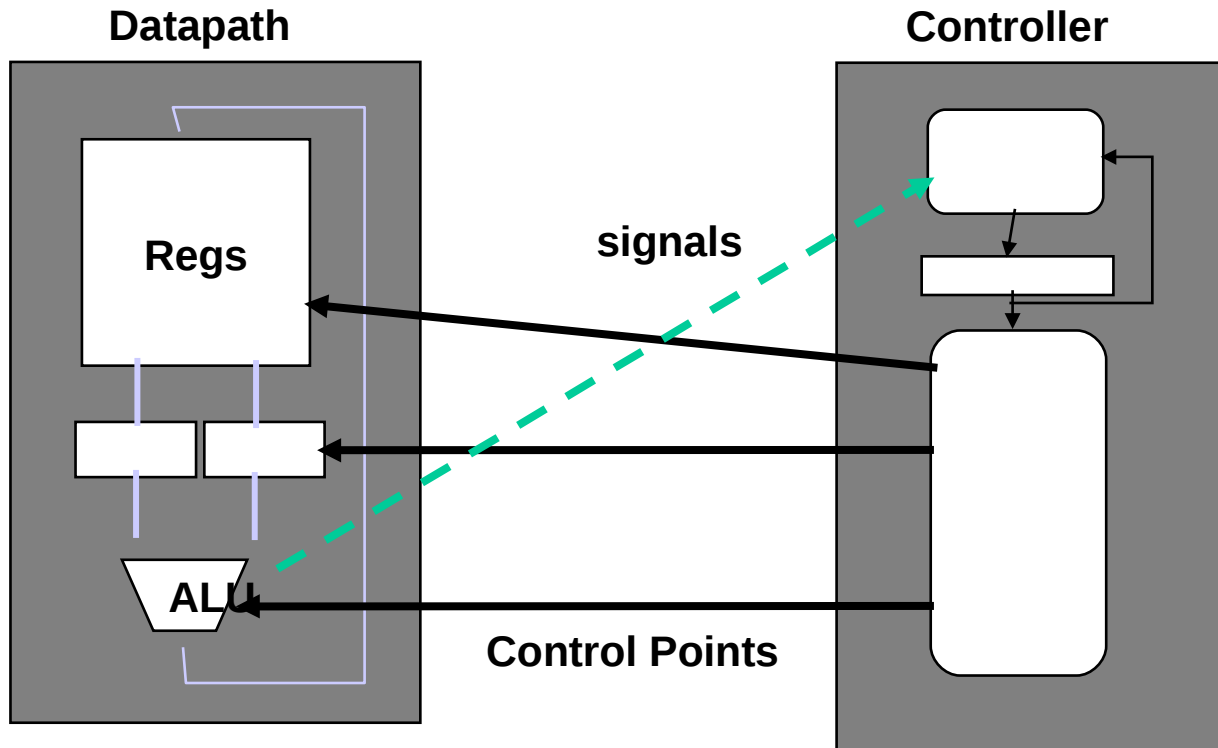
# Types of Data Hazards

---

- RAW (read after write)
  - only hazard for 'fixed' pipelines
  - later instruction must *read* after earlier instruction *writes*
- WAW (write after write)
  - variable-length pipeline (e.g. FP/int)
  - later instruction must *write* after earlier instruction *writes*
- WAR (write after read)
  - pipelines with late read
  - later instruction must *write* after earlier instruction *reads*

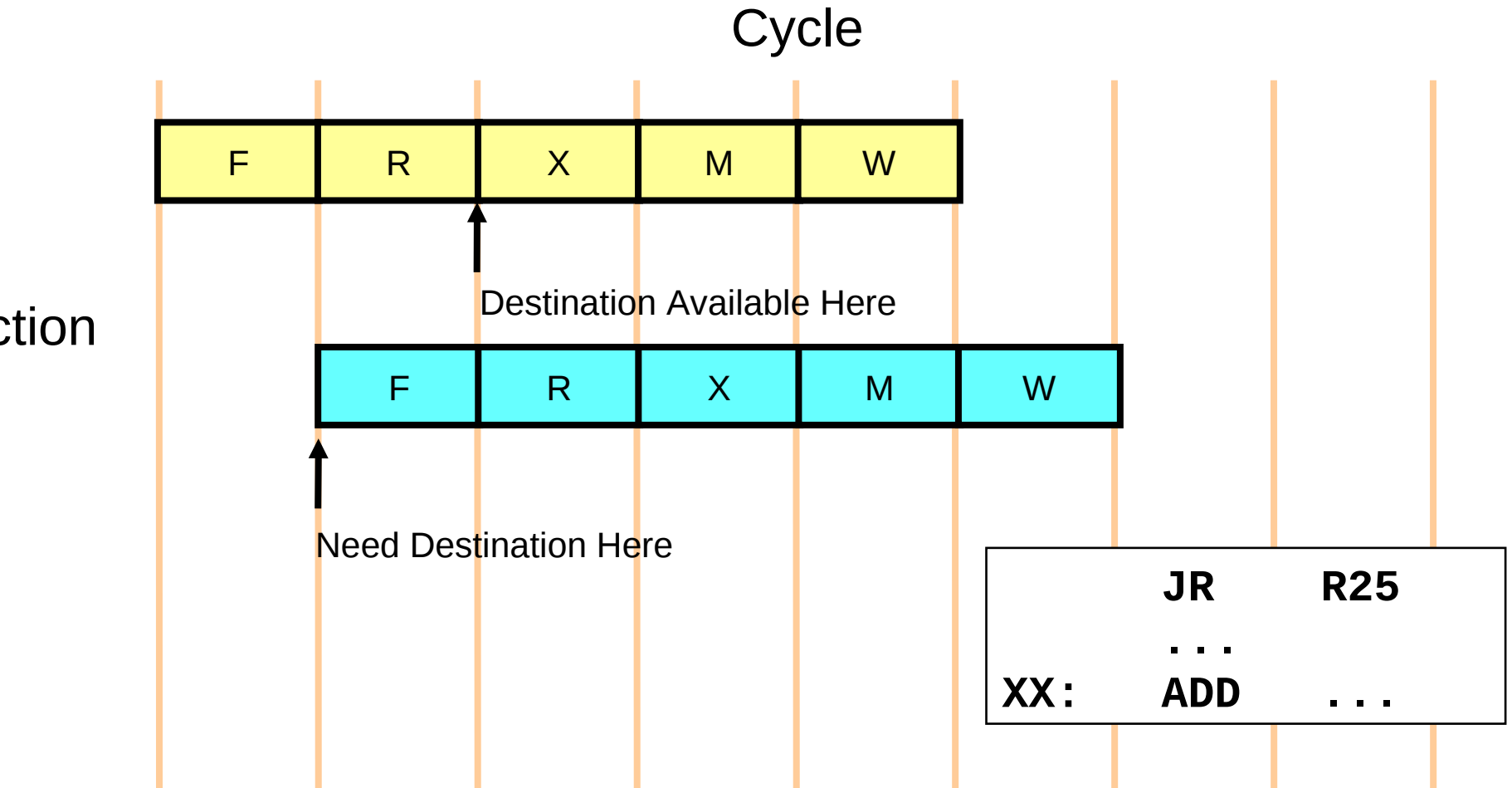


# Datapath vs Control



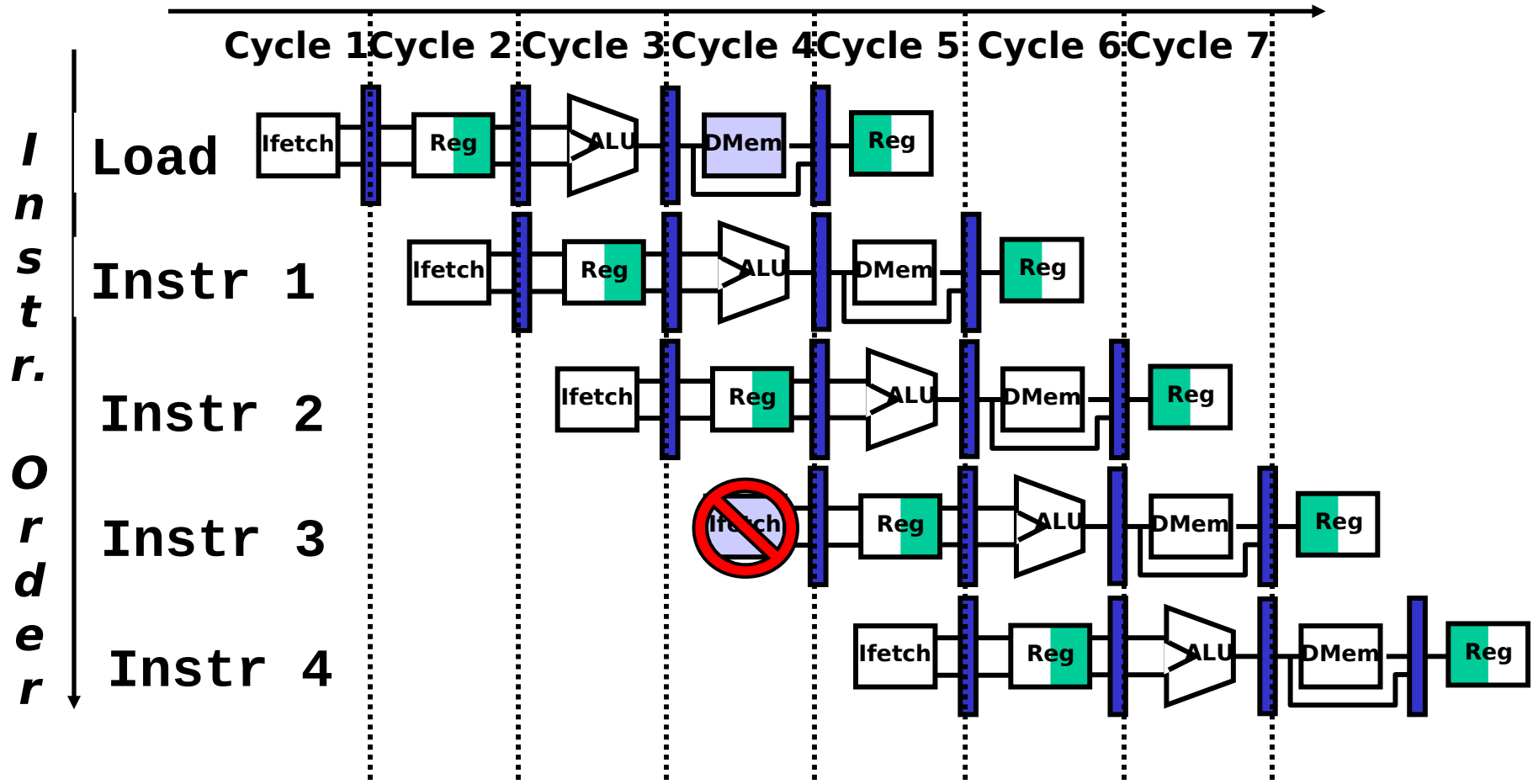
- Datapath: Storage, FU, interconnect sufficient to perform the desired functions
  - Inputs are Control Points
  - Outputs are signals
- Controller: State machine to orchestrate operation on the data path
  - Based on desired function and signals

# Control Hazards



# Control Hazards

*Time (clock cycles)*



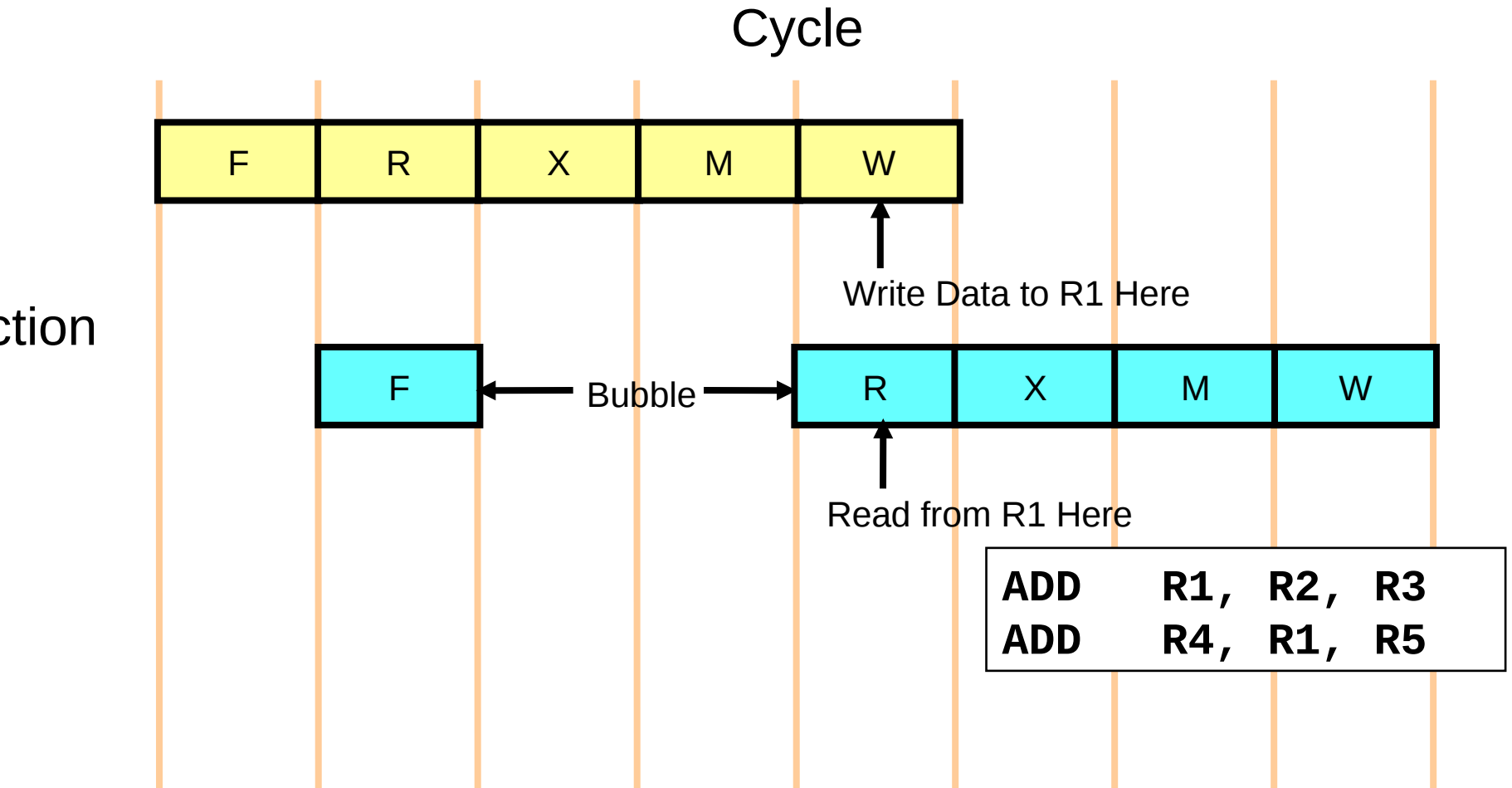
# Resolving Hazards: Pipeline Stalls

---

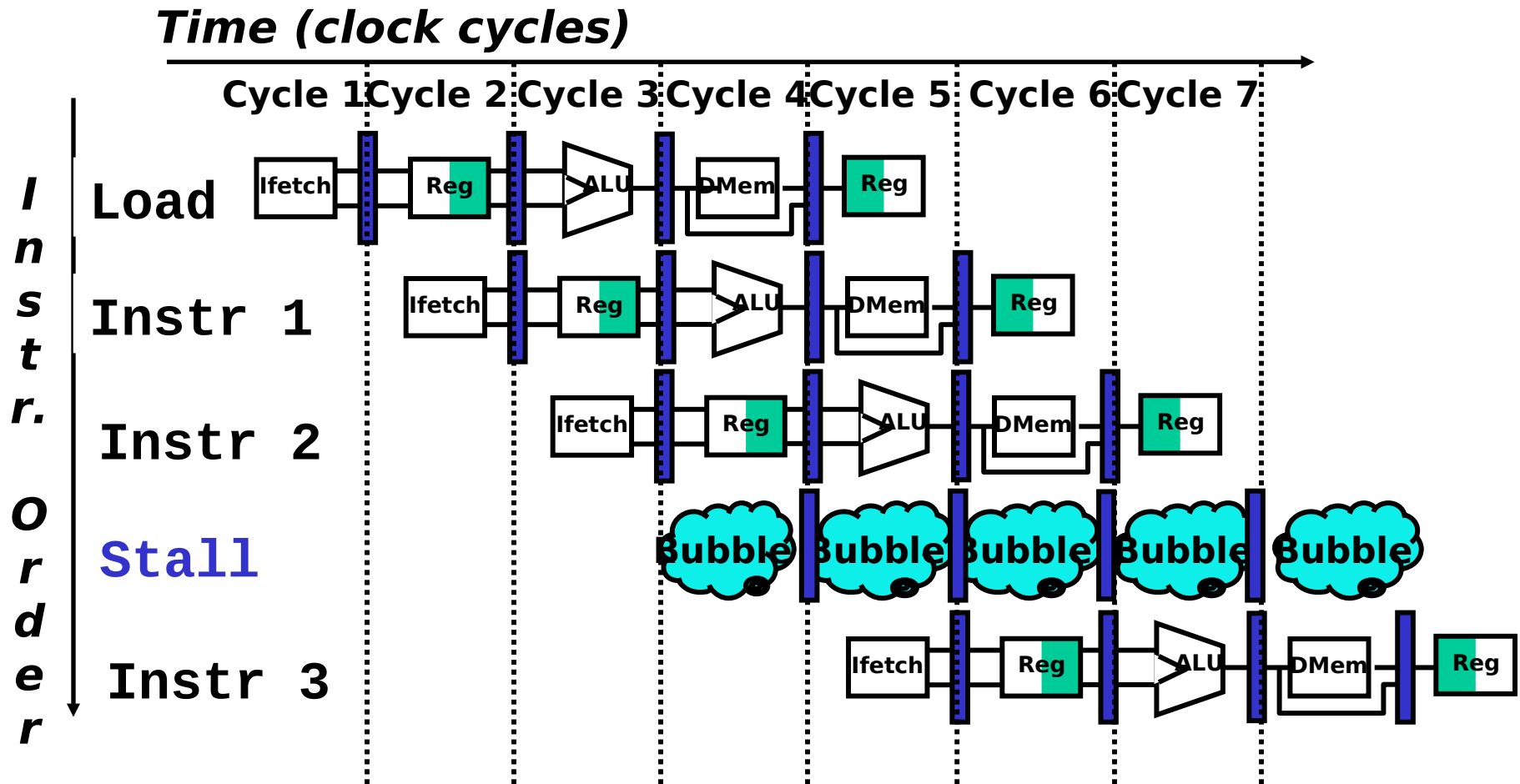
- Can resolve any type of hazard
  - data, control, or structural
- Detect the hazard
- Freeze the pipeline up to the dependent stage until the hazard is resolved



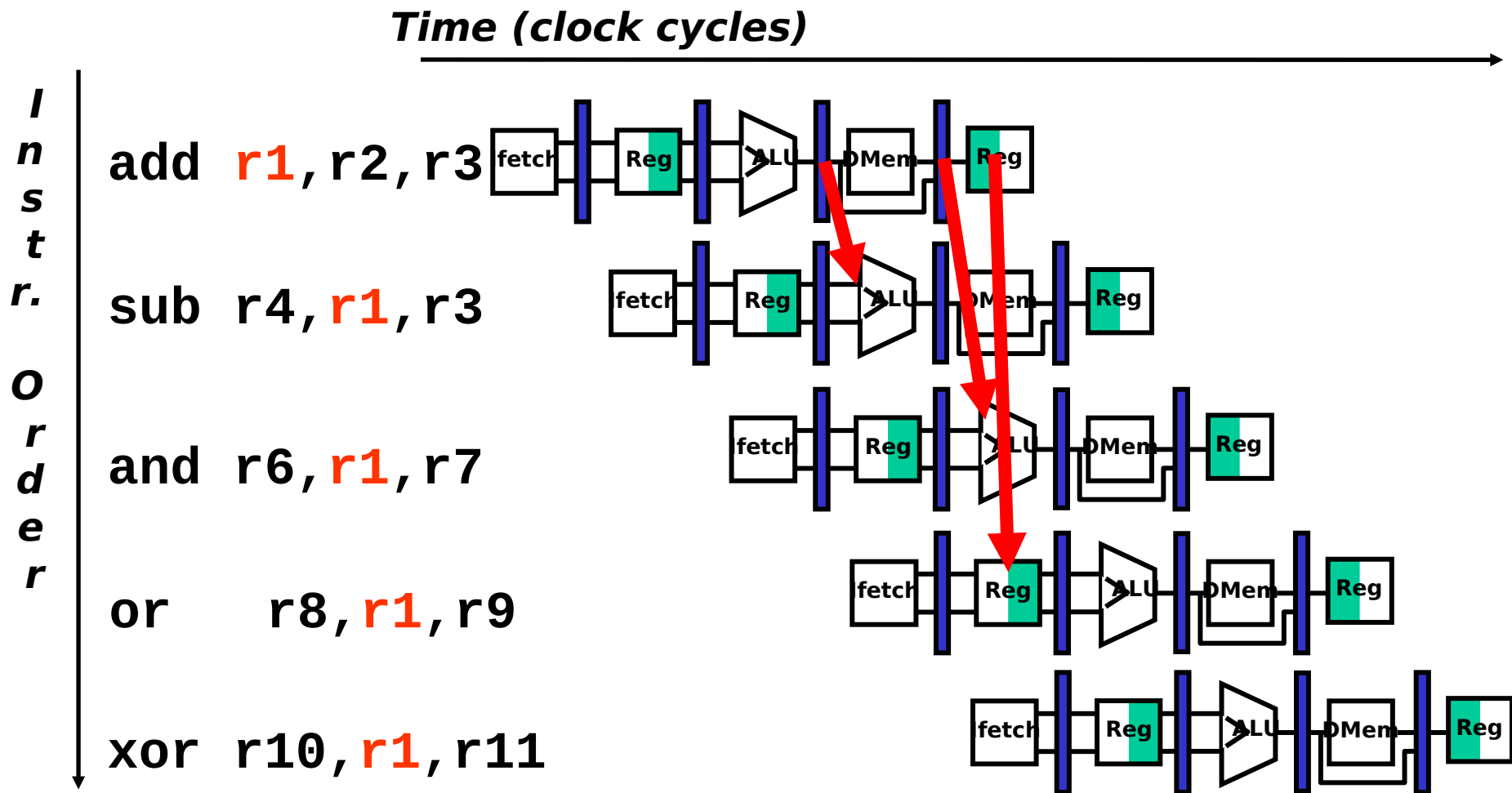
# Example Pipeline Stall (Diagram)



# Resolving Hazards



# Forwarding to Avoid Data Hazard



# Speedup Equation for Pipelining

---

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

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

**For simple RISC pipeline, Ideal 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
- Loads are 40% of instructions executed

$$\begin{aligned}\text{SpeedUp}_A &= \text{Pipeline Depth} / (1 + 0) \times (\text{clock}_{\text{unpipe}} / \text{clock}_{\text{unpipe}}) \\ &= \text{Pipeline Depth}\end{aligned}$$

$$\begin{aligned}\text{SpeedUp}_B &= \text{Pipeline Depth} / (1 + 0.4 \times 1) \times (\text{clock}_{\text{unpipe}} / (\text{clock}_{\text{unpipe}} / 1.05)) \\ &= (\text{Pipeline Depth} / 1.4) \times 1.05 \\ &= 0.75 \times \text{Pipeline Depth}\end{aligned}$$

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

- Machine A is 1.33 times faster

# Software Scheduling to Avoid Load Hazards

---

Producing fast code for

$a = b + c$

$d = e - f$

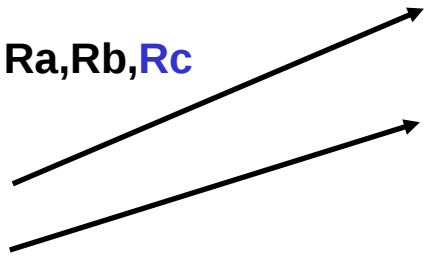
assuming  $a, b, c, d, e,$  and  $f$  in memory.

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
```

Faster 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
```



# Branch Stall Impact

---

- If  $\text{CPI} = 1$ , 30% branch,  
Stall 3 cycles  $\Rightarrow$  new  $\text{CPI} = 1.9$
- Solution:
  - Determine branch taken or not sooner, AND
  - Compute taken branch address earlier

# Branch Hazard Alternatives

---

- Stall until branch direction is clear
- Predict Branch (Taken or Not Taken):
  - Execute successor instructions in sequence (if not taken) and “squash” instructions in pipeline if branch mispredicted
  - Modern CPUs have multiple predictors
  - Must have already calculated branch target address



# Branch Hazard Alternatives

---

## Delayed Branch

- Define branch to take place **AFTER** a following instruction

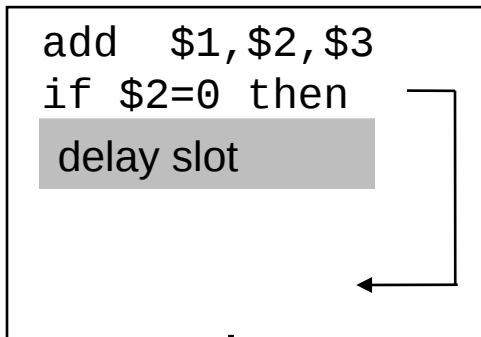
```
branch instruction
  sequential successor1
  sequential successor2
  .....
  sequential successorn
branch target if taken
```



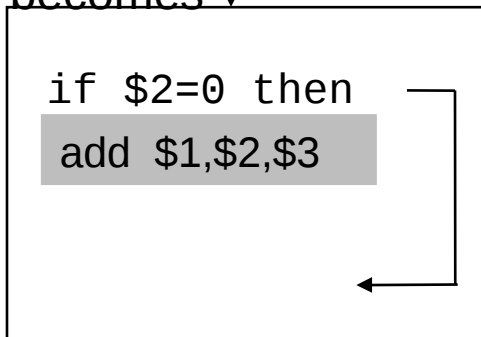
- MIPS, SPARC
- Experience has shown it has issues:
  - Makes code difficult to maintain and debug
  - Compiler has to find an instruction which is safe to execute regardless if the branch is taken or not
  - If the hardware changes, old programs may not work at all

# Scheduling Branch Delay Slots

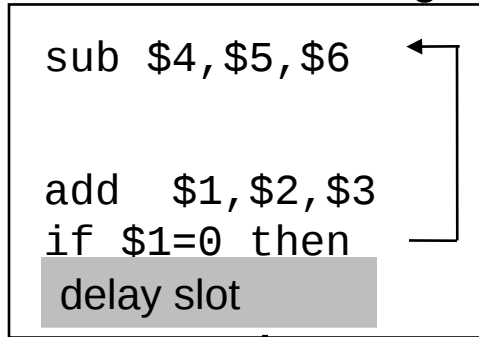
A. From before branch



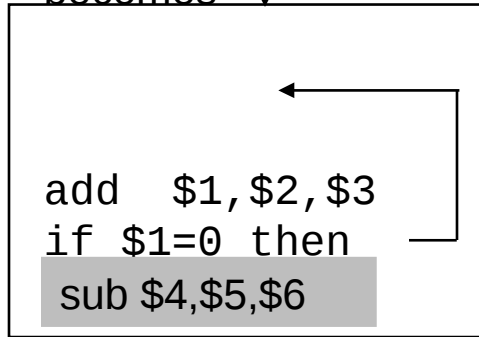
becomes



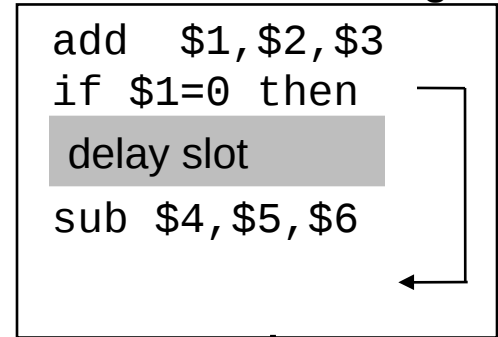
B. From branch target



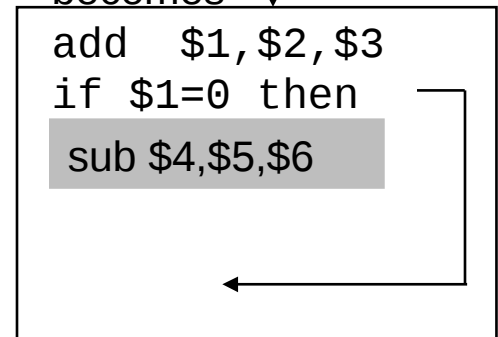
becomes



C. From fall through



becomes



- 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

# 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% ( $60\% \times 80\%$ ) of slots usefully filled
- Delayed branching has lost popularity compared to more expensive but more flexible dynamic approaches
- Growth in available transistors has made dynamic approaches relatively cheaper

# Exceptions and Interrupts

---

- **Exception:** An unusual event happens to an instruction during its execution
  - Examples: divide by zero, undefined opcode
- **Interrupt:** Hardware signal to switch processor to new instruction stream
  - Example: sound card interrupts when it needs more audio output samples (audio “click” happens if it is left waiting)

# Exceptions and Interrupts

---

- Exception or interrupt must appear to happen between 2 instructions ( $I_i$  and  $I_{i+1}$ )
- Interrupt (exception) handler either aborts program or restarts at instruction  $I_{i+1}$

# Precise Exceptions (Sequential Processor)

---

- When interrupt occurs, state of interrupted process is saved, including PC, registers, and memory
- Interrupt is **precise** if the following three conditions hold
  - All instructions preceding  $u$  have been executed, and have modified the state correctly
  - All instructions following  $u$  are unexecuted, and have not modified the state
  - If the interrupt was caused by an instruction, it was caused by instruction  $u$ , which is either completely executed (e.g.: overflow) or completely unexecuted (e.g: VM page fault)
- Precise interrupts are desirable if software is to fix up error that caused interrupt and execution has to be resumed
  - Easy for external interrupts, could be complex and costly for internal
  - Imperative for some interrupts (VM page faults, IEEE FP standard)