

# Lecture 3: Pipelining Basics

---

- Today: chapter 1 wrap-up, basic pipelining implementation (Sections C.1 - C.4)
- Reminders:
  - Sign up for the class mailing list
  - First assignment is on-line, due next Tuesday
  - TA office hours: Ali Shafiee, Monday 3-4pm
  - Class notes

# Defining Fault, Error, and Failure

---

- A *fault* produces a *latent error*; it becomes *effective* when activated; it leads to *failure* when the observed actual behavior deviates from the ideal specified behavior
- Example I : a programming mistake is a fault; the buggy code is the latent error; when the code runs, it is effective; if the buggy code influences program output/behavior, a failure occurs
- Example II : an alpha particle strikes DRAM (fault); if it changes the memory bit, it produces a latent error; when the value is read, the error becomes effective; if program output deviates, failure occurs

# Defining Reliability and Availability

---

- A system toggles between
  - Service accomplishment: service matches specifications
  - Service interruption: services deviates from specs
- The toggle is caused by *failures* and *restorations*
- Reliability measures continuous service accomplishment and is usually expressed as mean time to failure (MTTF)
- Availability measures fraction of time that service matches specifications, expressed as  $MTTF / (MTTF + MTTR)$

# Amdahl's Law

---

- Architecture design is very bottleneck-driven – make the common case fast, do not waste resources on a component that has little impact on overall performance/power
- Amdahl's Law: performance improvements through an enhancement is limited by the fraction of time the enhancement comes into play
- Example: a web server spends 40% of time in the CPU and 60% of time doing I/O – a new processor that is ten times faster results in a 36% reduction in execution time (speedup of 1.56) – Amdahl's Law states that maximum execution time reduction is 40% (max speedup of 1.66)

# Principle of Locality

---

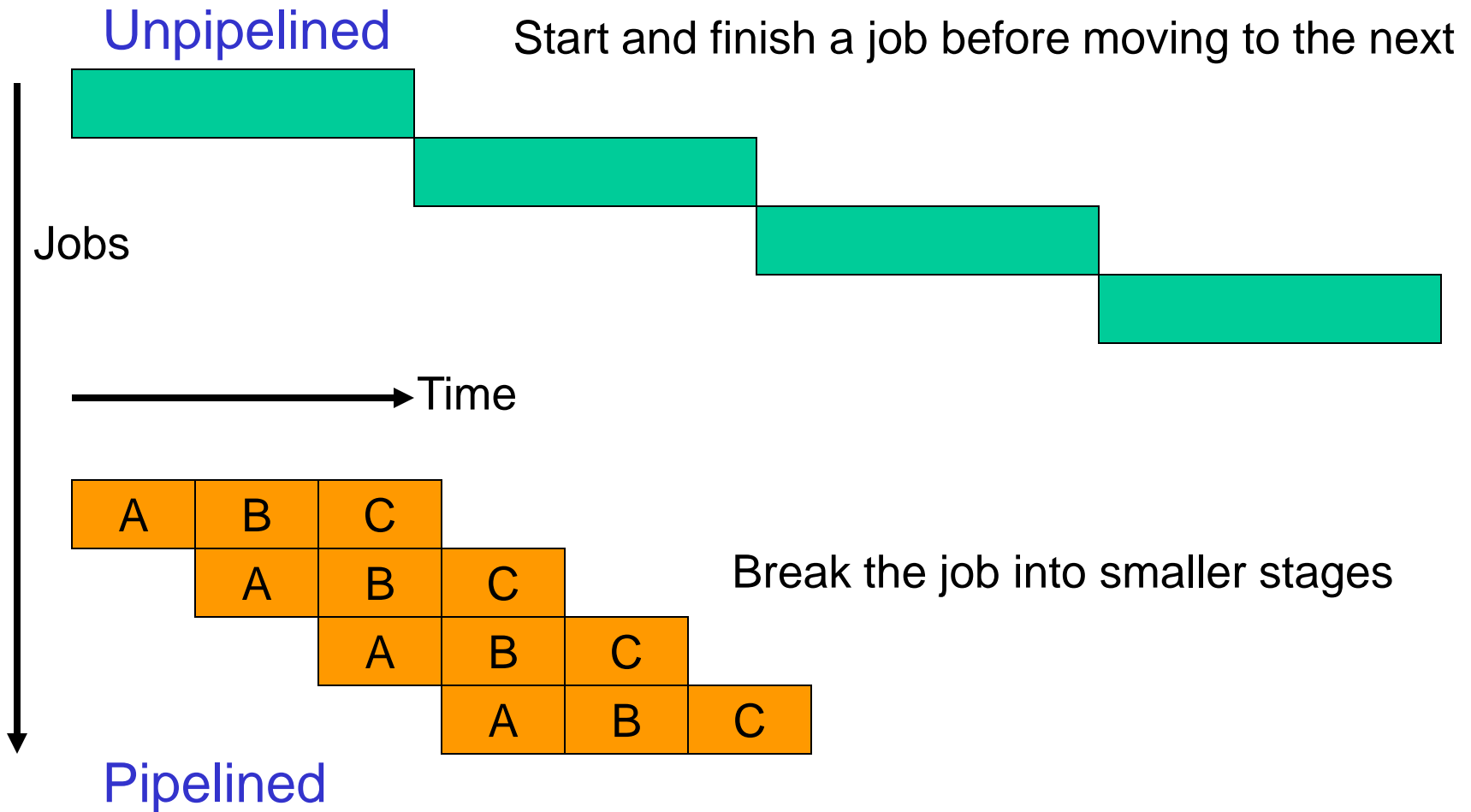
- Most programs are predictable in terms of instructions executed and data accessed
- The 90-10 Rule: a program spends 90% of its execution time in only 10% of the code
- Temporal locality: a program will shortly re-visit  $X$
- Spatial locality: a program will shortly visit  $X+1$

# Exploit Parallelism

---

- Most operations do not depend on each other – hence, execute them in parallel
- At the circuit level, simultaneously access multiple ways of a set-associative cache
- At the organization level, execute multiple instructions at the same time
- At the system level, execute a different program while one is waiting on I/O

# The Assembly Line



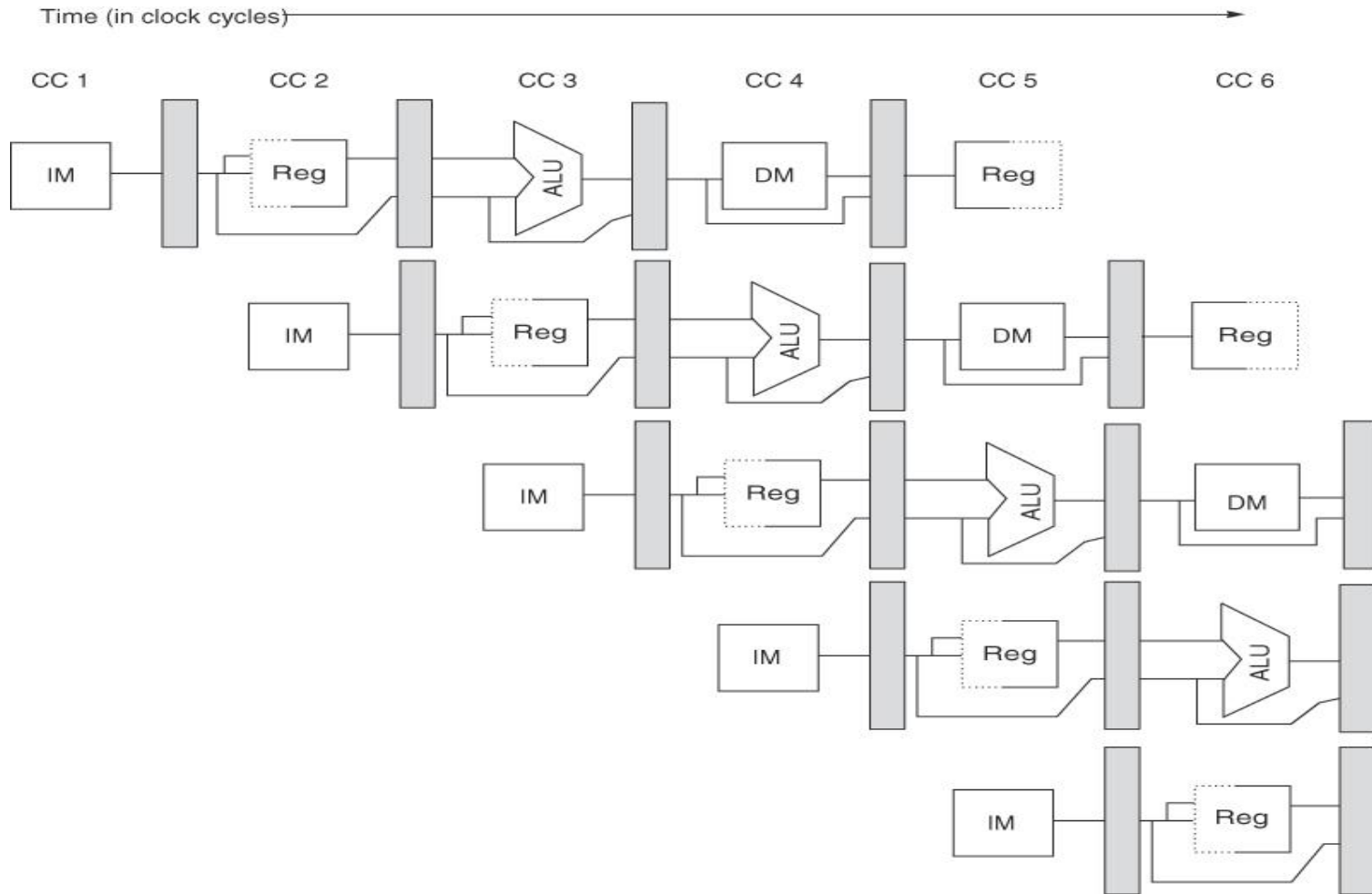
# Quantitative Effects

---

- As a result of pipelining:
  - Time in ns per instruction goes up
  - Number of cycles per instruction goes up (note the increase in clock speed)
  - Total execution time goes down, resulting in lower time per instruction
  - Average cycles per instruction increases slightly
  - Under ideal conditions, speedup
    - = ratio of *elapsed times between successive instruction completions*
    - = number of pipeline stages = increase in clock speed

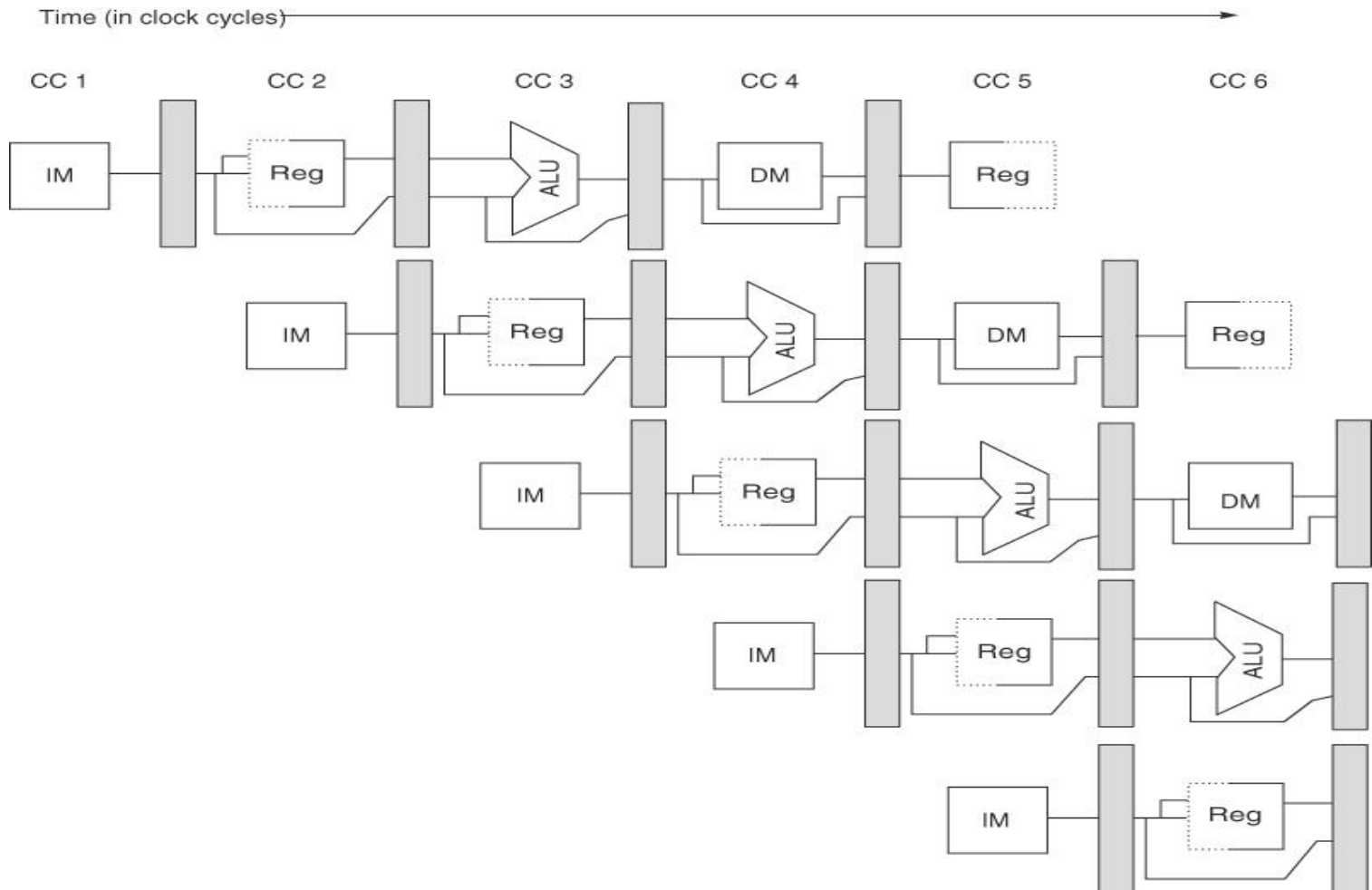


# A 5-Stage Pipeline



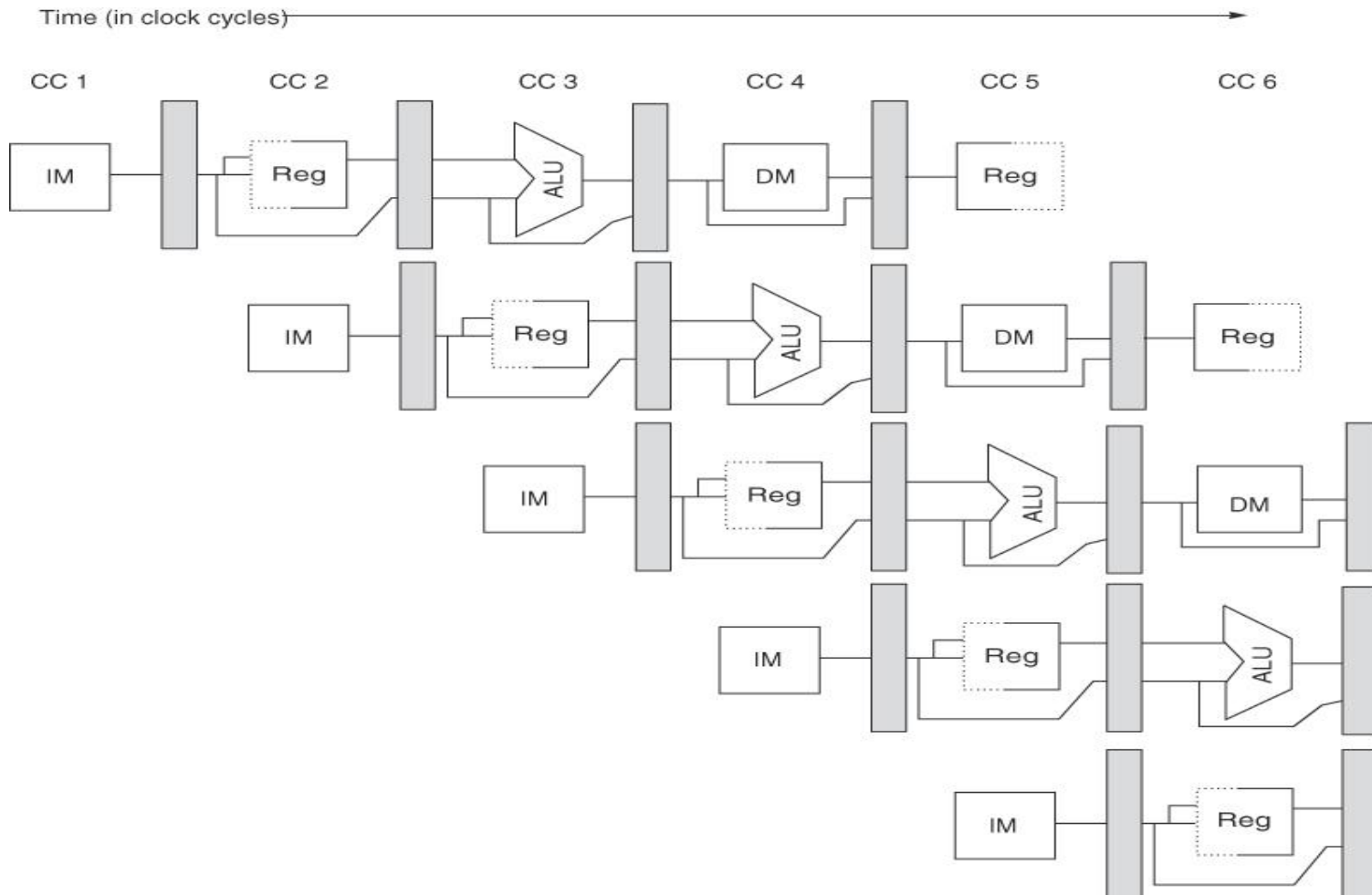
# A 5-Stage Pipeline

Use the PC to access the I-cache and increment PC by 4



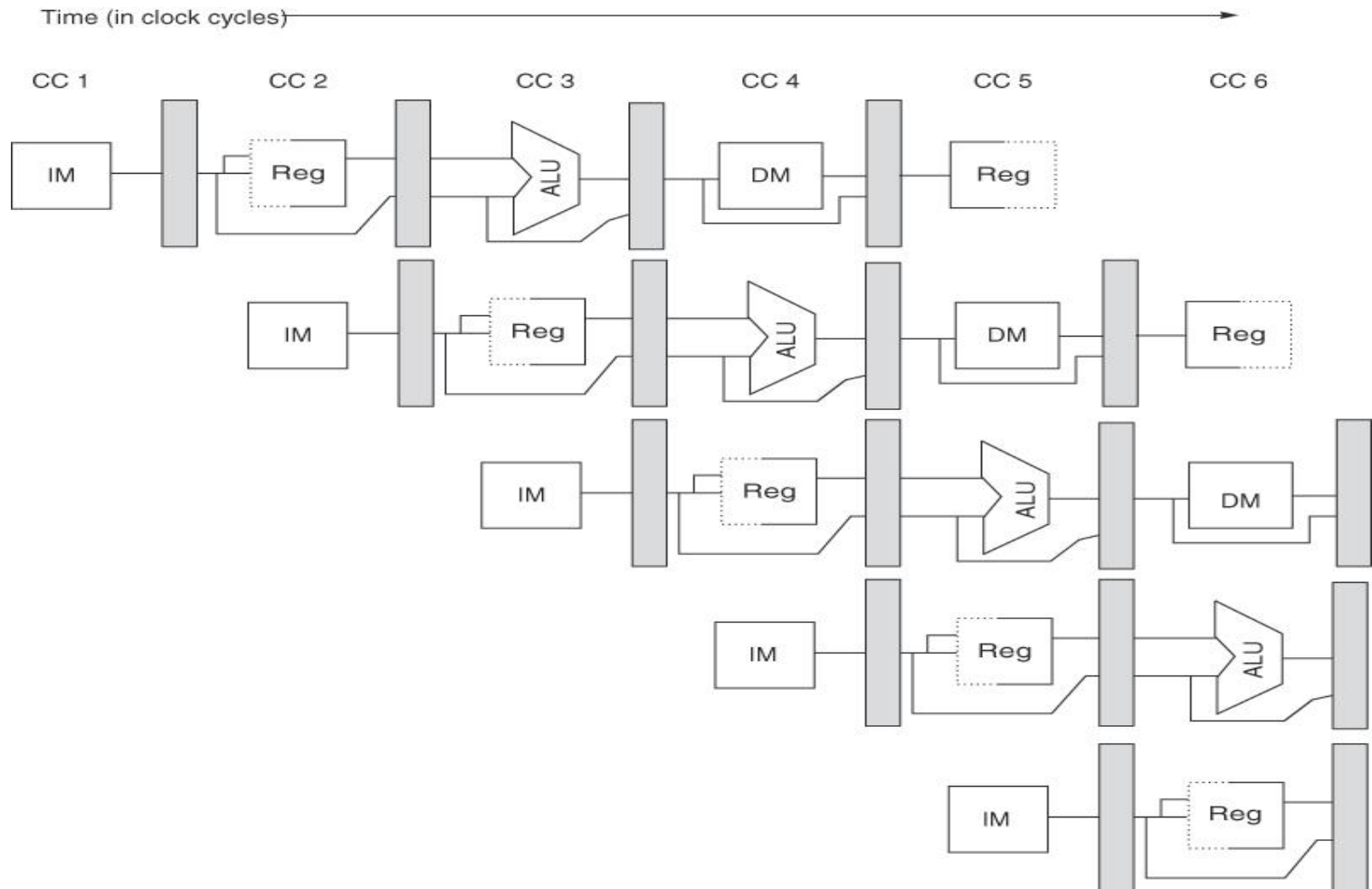
# A 5-Stage Pipeline

Read registers, compare registers, compute branch target; for now, assume branches take 2 cyc (there is enough work that branches can easily take more)



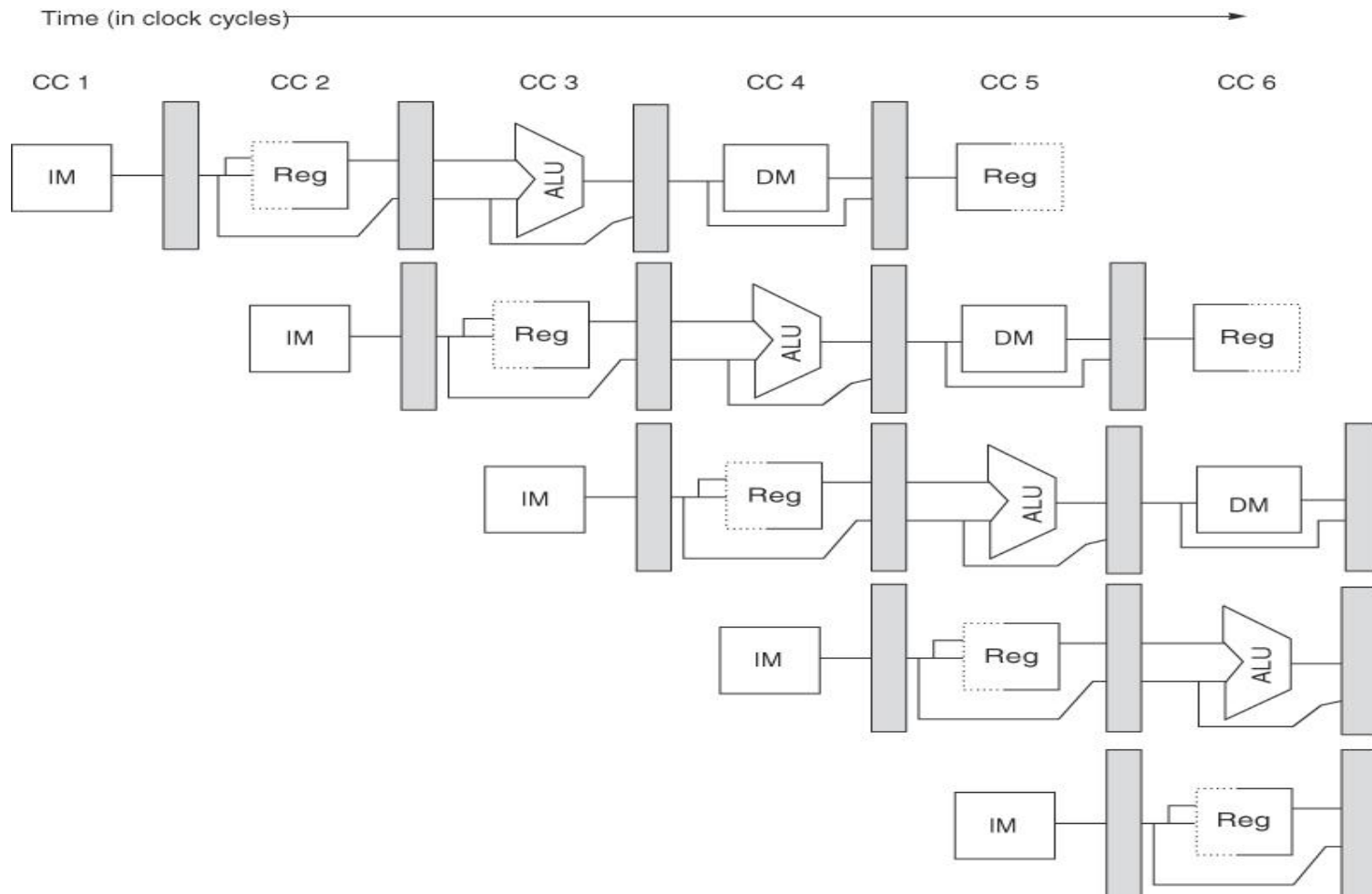
# A 5-Stage Pipeline

ALU computation, effective address computation for load/store



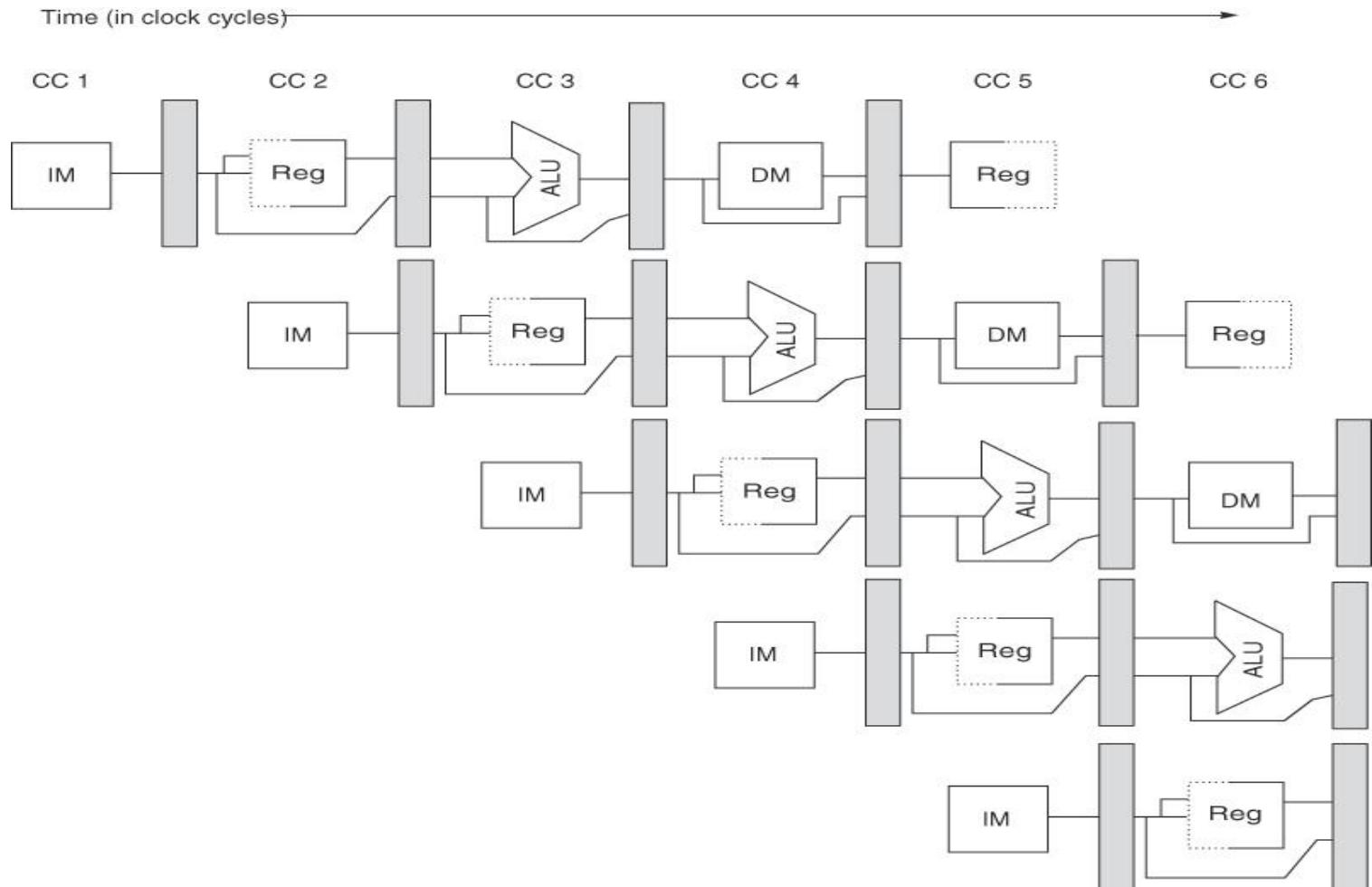
# A 5-Stage Pipeline

Memory access to/from data cache, stores finish in 4 cycles



# A 5-Stage Pipeline

Write result of ALU computation or load into register file



# Conflicts/Problems

---

- I-cache and D-cache are accessed in the same cycle – it helps to implement them separately
- Registers are read and written in the same cycle – easy to deal with if register read/write time equals cycle time/2 (else, use bypassing)
- Branch target changes only at the end of the second stage -- what do you do in the meantime?
- Data between stages get latched into registers (overhead that increases latency per instruction)

# Hazards

---

- Structural hazards: different instructions in different stages (or the same stage) conflicting for the same resource
- Data hazards: an instruction cannot continue because it needs a value that has not yet been generated by an earlier instruction
- Control hazard: fetch cannot continue because it does not know the outcome of an earlier branch – special case of a data hazard – separate category because they are treated in different ways

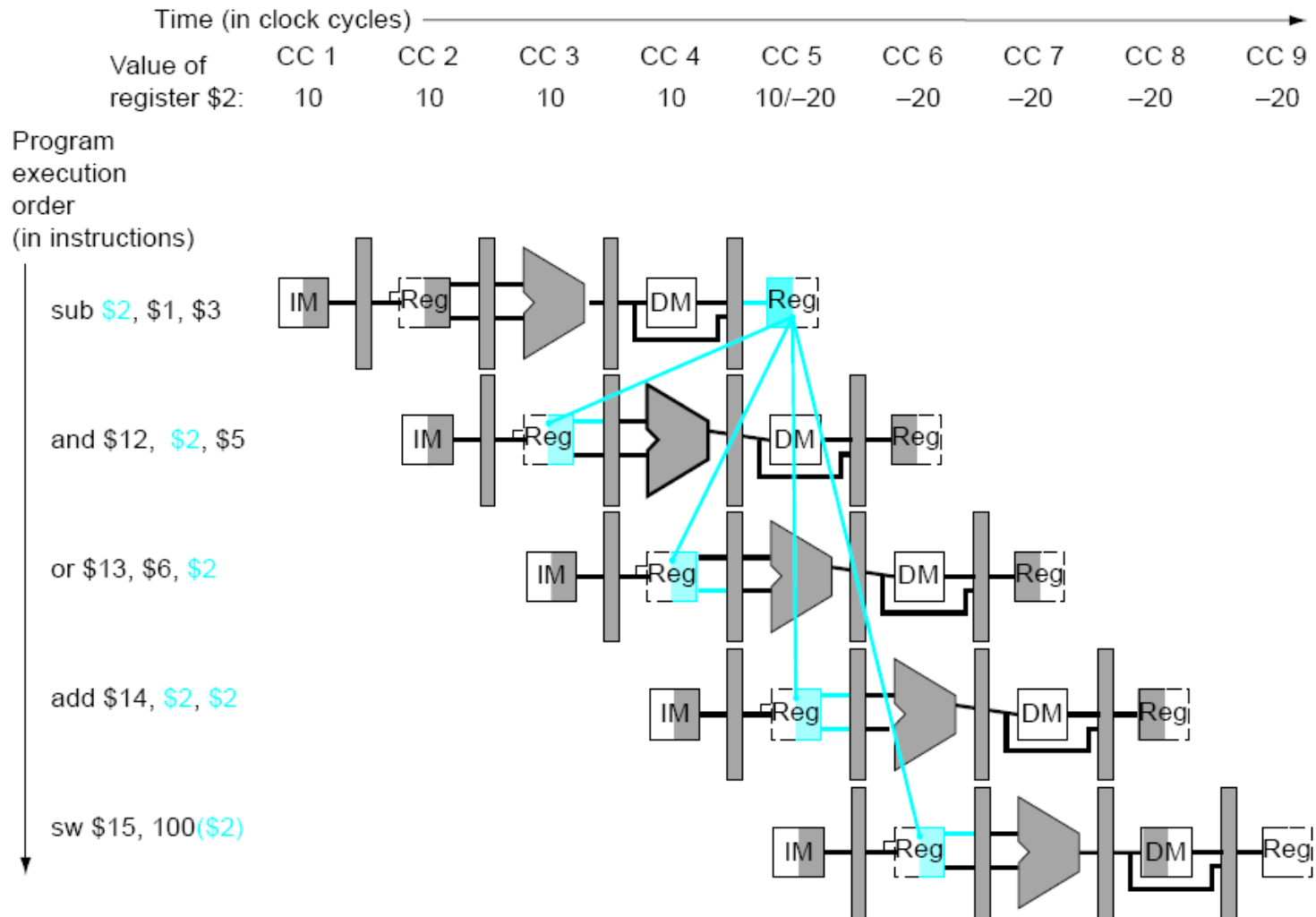


# Structural Hazards

---

- Example: a unified instruction and data cache → stage 4 (MEM) and stage 1 (IF) can never coincide
- The later instruction and all its successors are delayed until a cycle is found when the resource is free → these are pipeline bubbles
- Structural hazards are easy to eliminate – increase the number of resources (for example, implement a separate instruction and data cache)

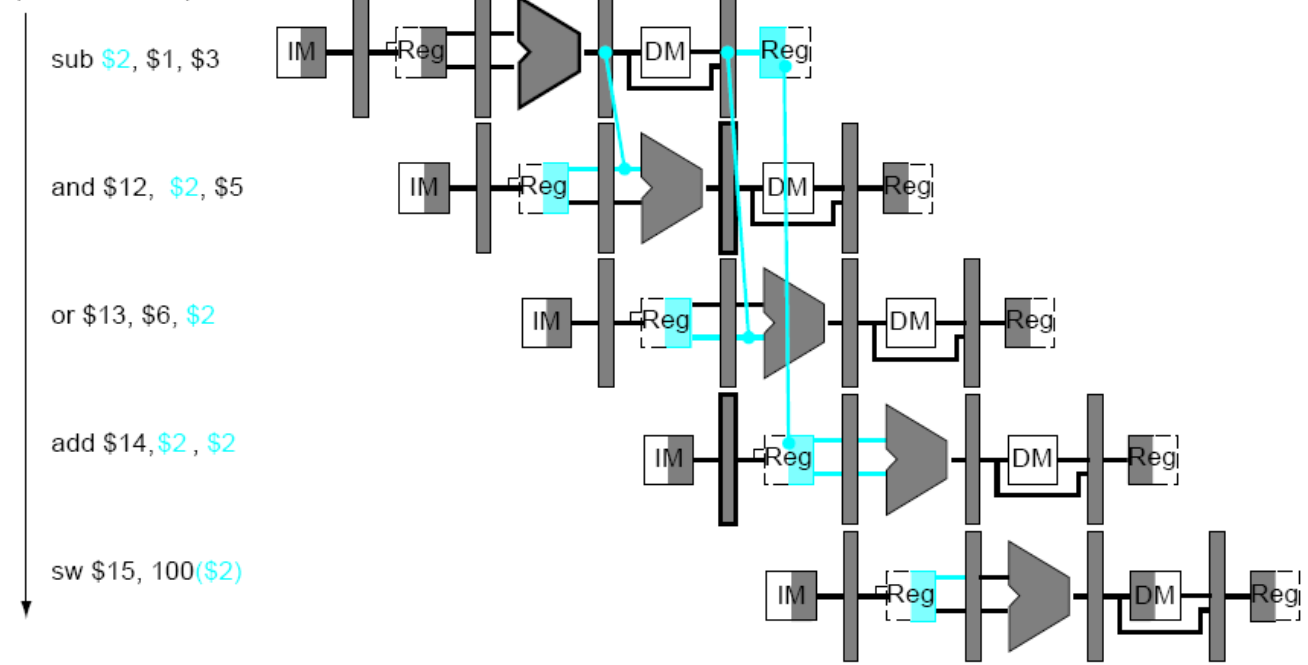
# Data Hazards



# Bypassing

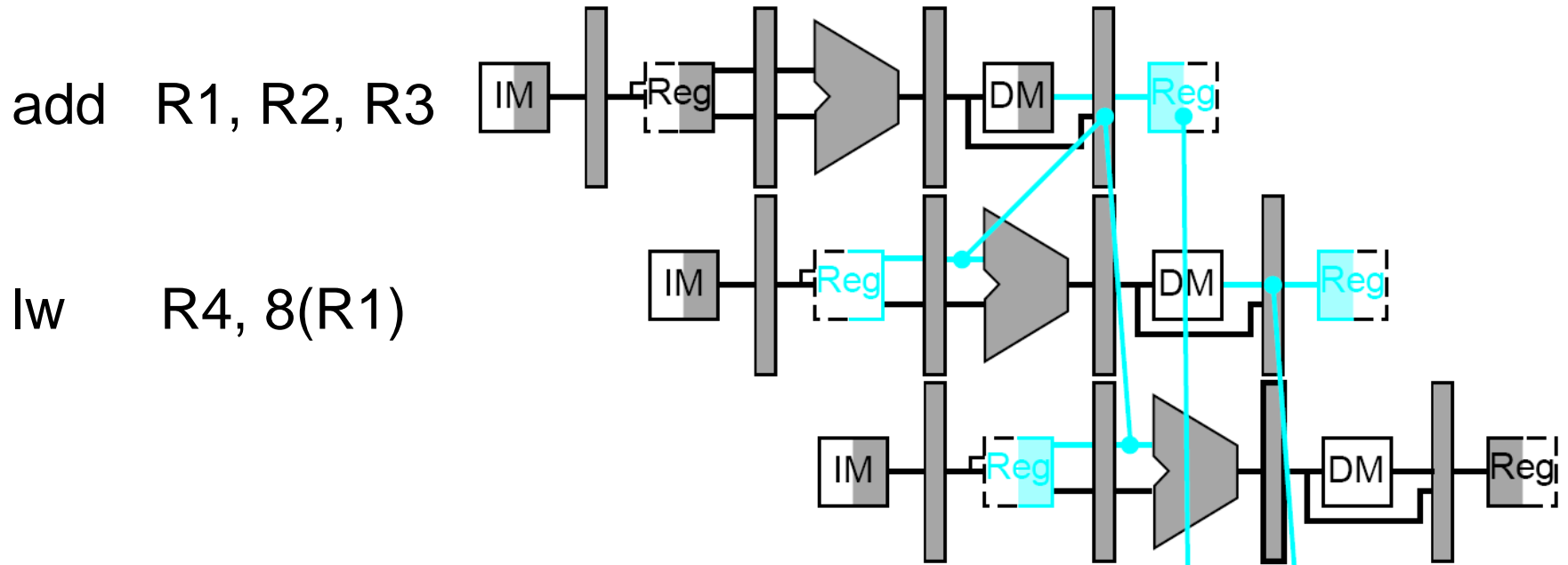
	Time (in clock cycles) →									
	CC 1	CC 2	CC 3	CC 4	CC 5	CC 6	CC 7	CC 8	CC 9	
Value of register \$2:	10	10	10	10	10/-20	-20	-20	-20	-20	
Value of EX/MEM:	X	X	X	-20	X	X	X	X	X	
Value of MEM/WB:	X	X	X	X	-20	X	X	X	X	

Program  
execution  
order  
(in instructions)



- Some data hazard stalls can be eliminated: bypassing

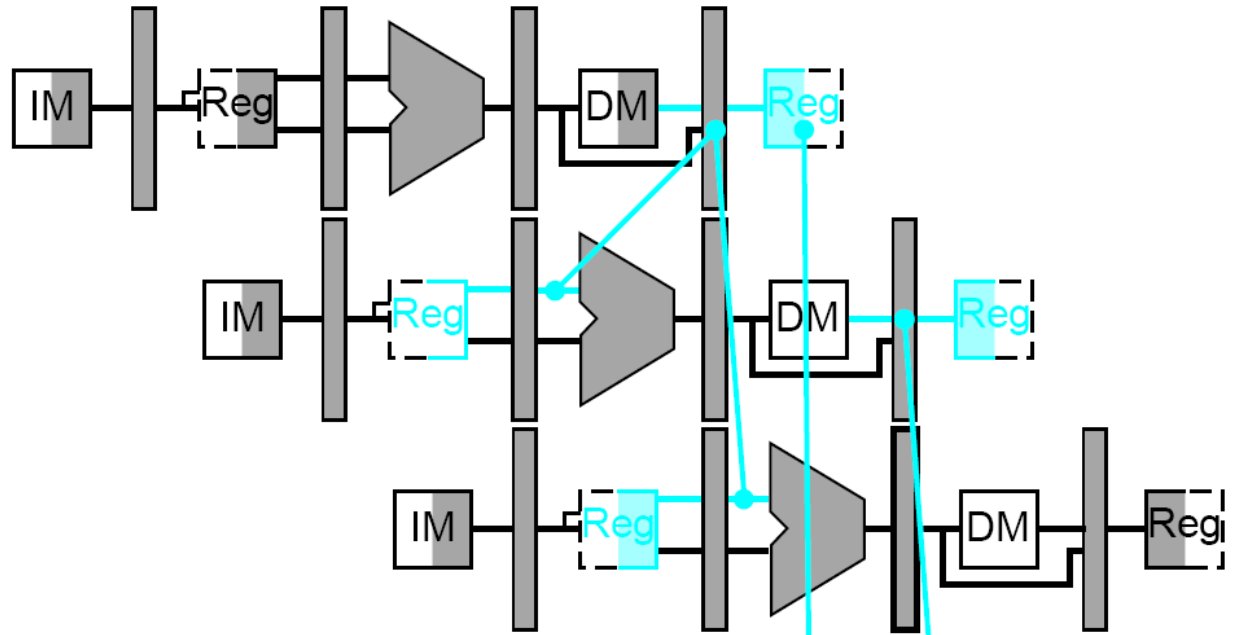
# Example



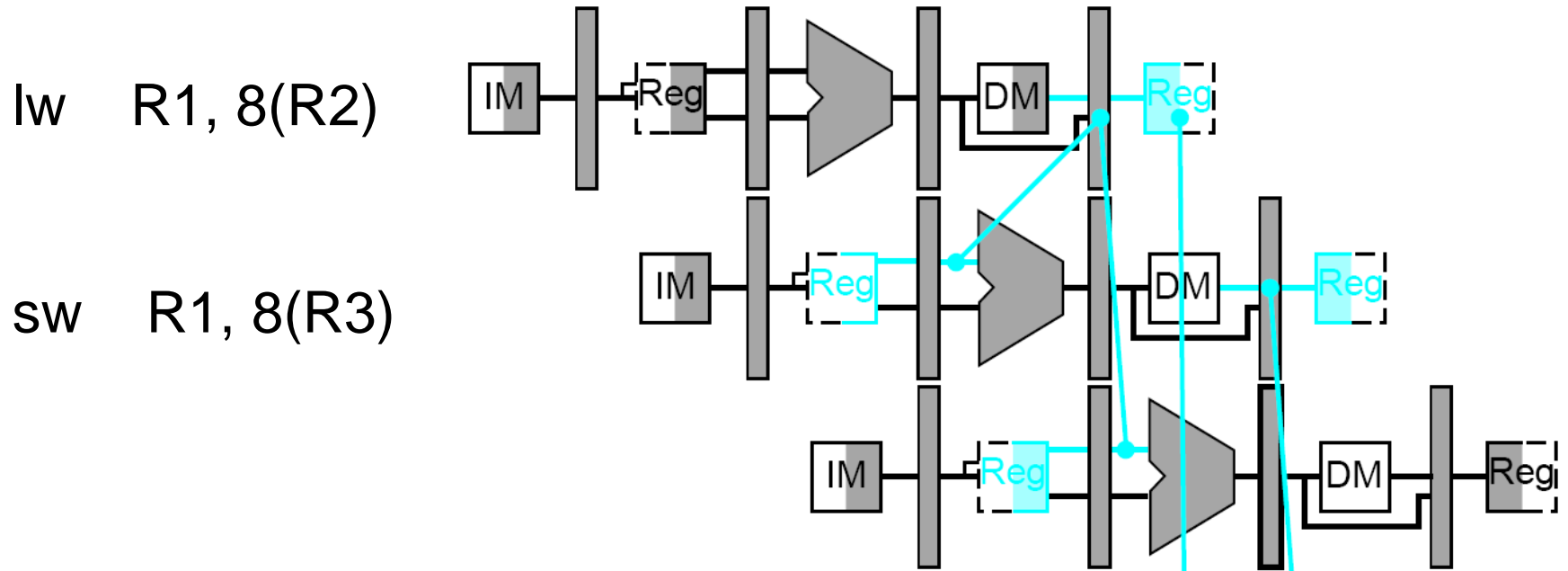
# Example

lw R1, 8(R2)

lw R4, 8(R1)



# Example



# Summary

---

- For the 5-stage pipeline, bypassing can eliminate delays between the following example pairs of instructions:

add/sub            R1, R2, R3  
add/sub/lw/sw    R4, R1, R5

lw            R1, 8(R2)  
sw            R1, 4(R3)

- The following pairs of instructions will have intermediate stalls:

lw            R1, 8(R2)  
add/sub/lw    R3, R1, R4      or    sw    R3, 8(R1)

fmul          F1, F2, F3  
fadd          F5, F1, F4

# Title

---

- Bullet