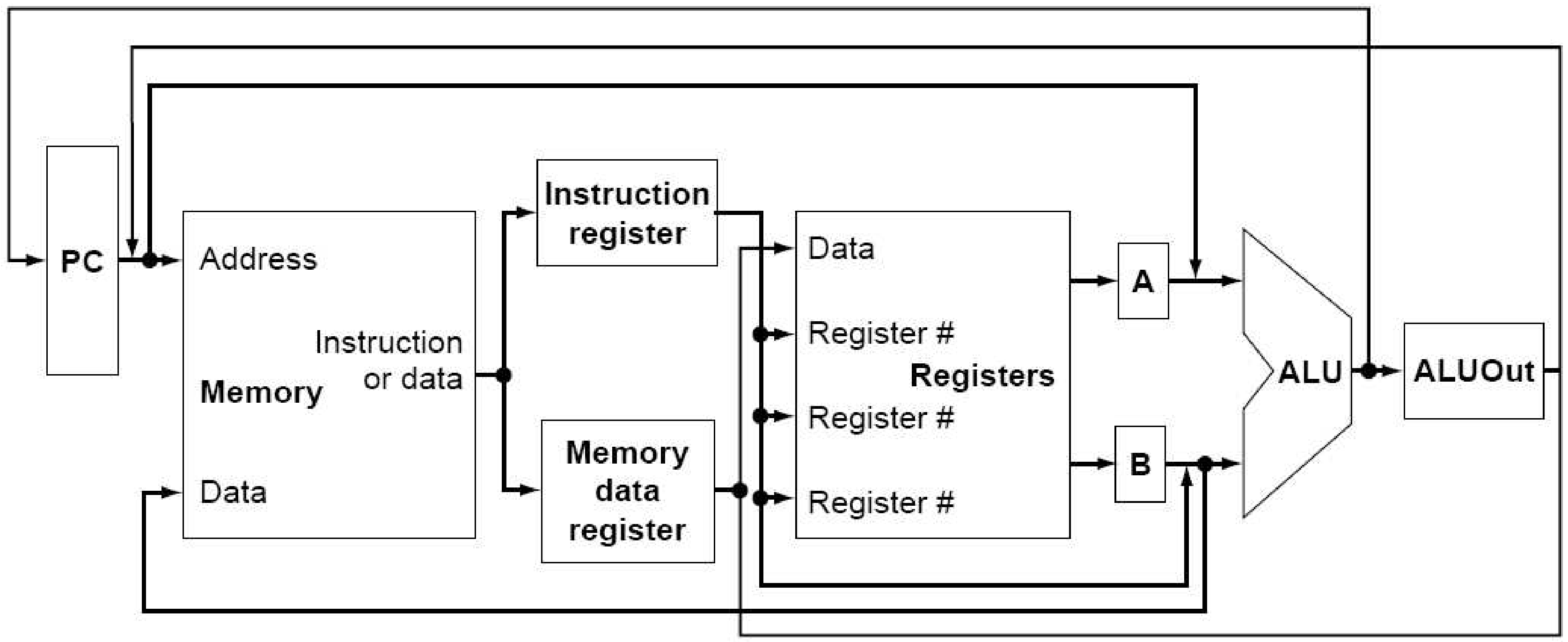
Lecture 17: Basic Pipelining

- Today's topics:
  - 5-stage pipeline
  - Hazards and instruction scheduling

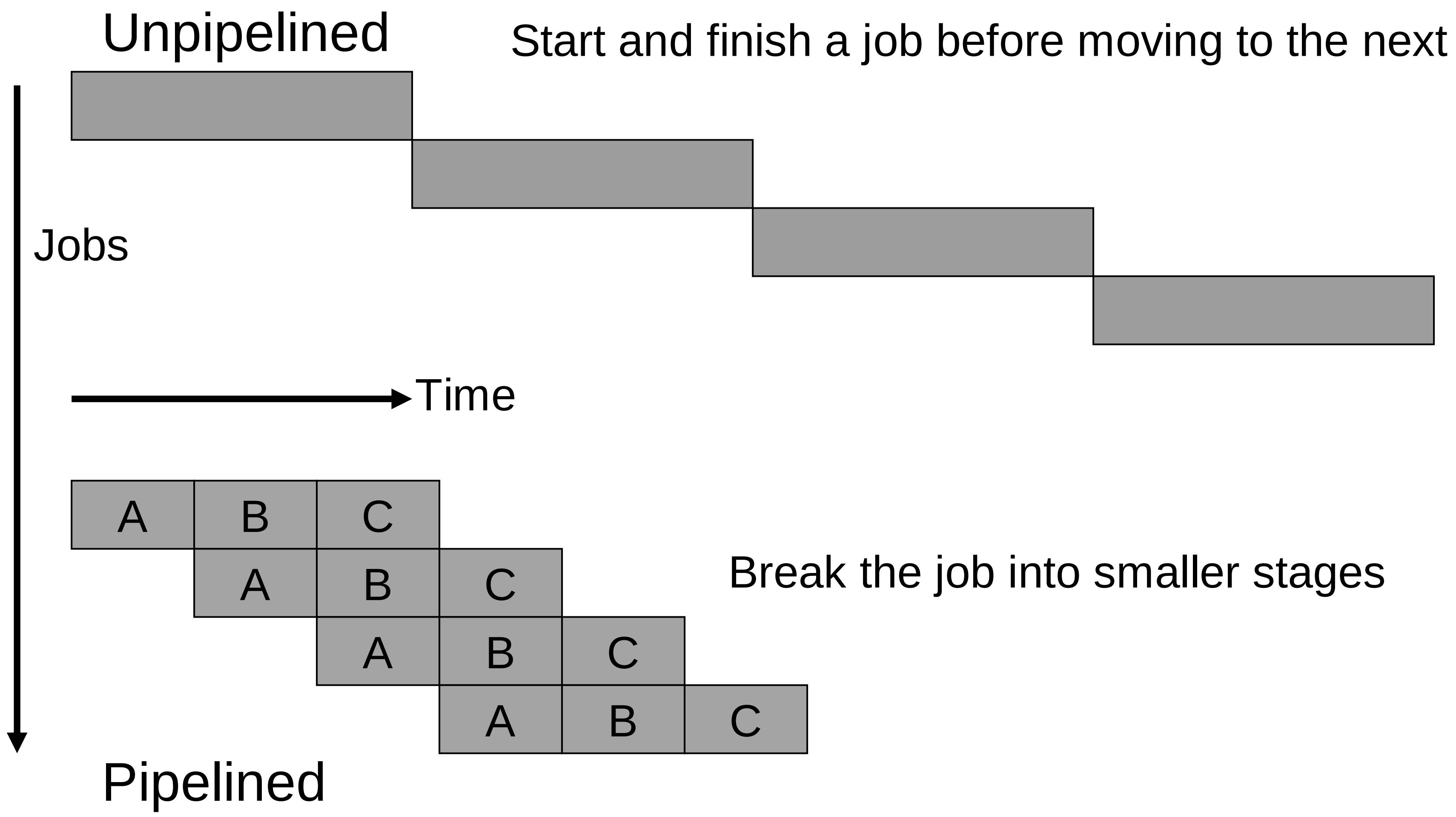
- Mid-term exam stats:
  - Highest: 90, Mean: 58

Multi-Cycle Processor



- Single memory unit shared by instructions and memory
- Single ALU also used for PC updates
- Registers (latches) to store the result of every block

The Assembly Line


Unpipelined
Start and finish a job before moving to the next
Jobs
Time
A
B
C
A
B
C
A
B
C
A
B
C
Break the job into smaller stages
Pipelined

Performance Improvements?

- Does it take longer to finish each individual job?
- Does it take shorter to finish a series of jobs?
- What assumptions were made while answering these questions?
- Is a 10-stage pipeline better than a 5-stage pipeline?

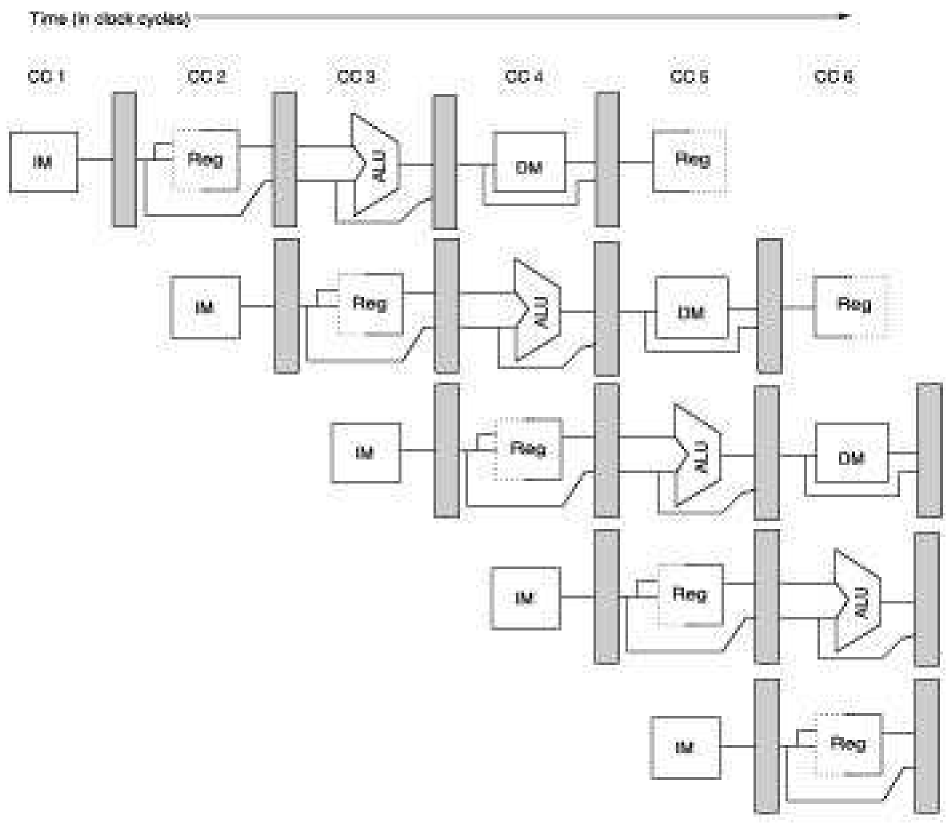
Quantitative Effects

- As a result of pipelining:
  - ➢ Time in ns per instruction goes up
  - ➢ Each instruction takes more cycles to execute
  - ➢ But… average CPI remains roughly the same
  - ➢ Clock speed goes up
  - ➢ Total execution time goes down, resulting in lower average time per instruction
  - ➢ Under ideal conditions, speedup

    = ratio of *elapsed times between successive instruction completions*

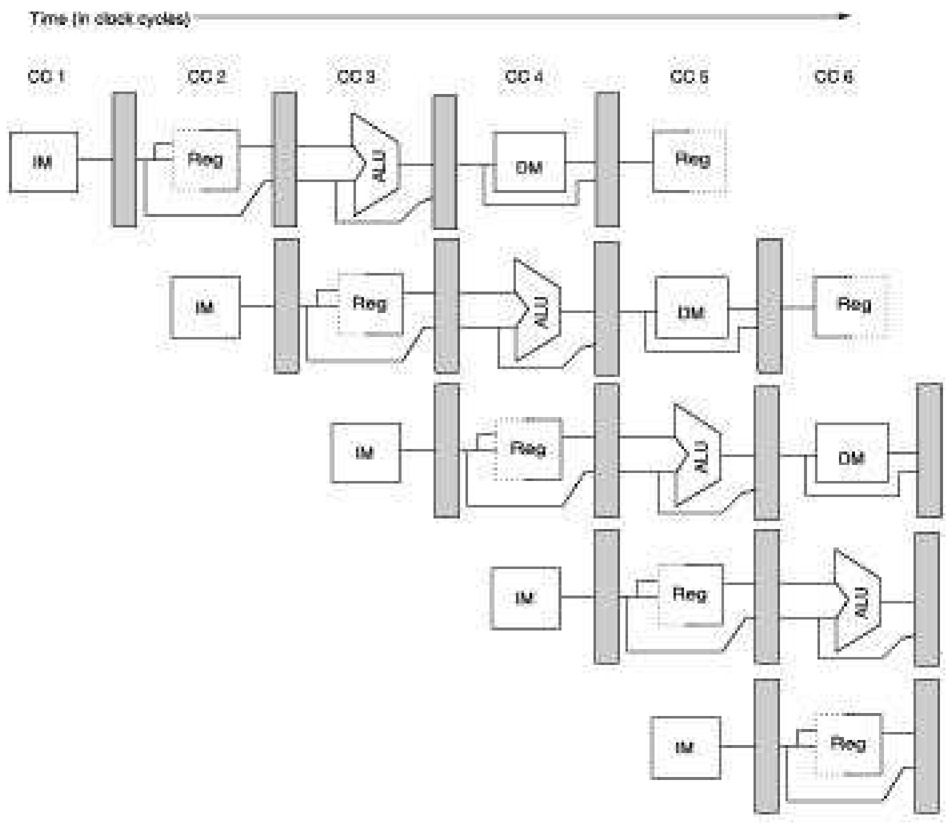
    = number of pipeline stages = increase in clock speed

A 5-Stage Pipeline



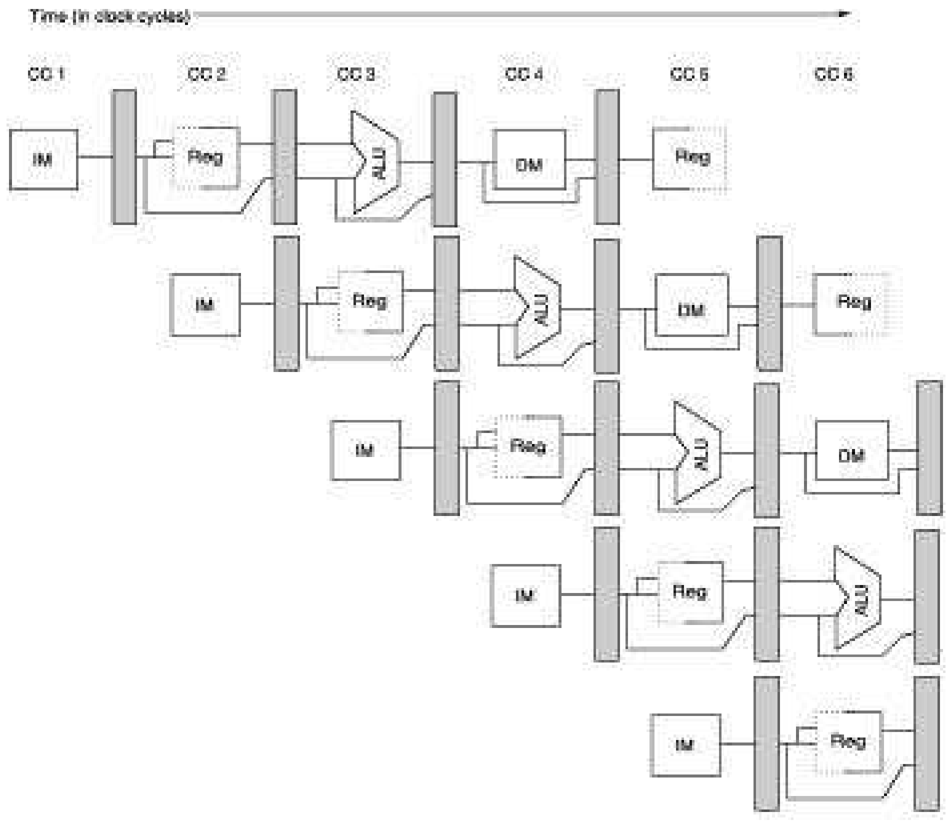
A 5-Stage Pipeline

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



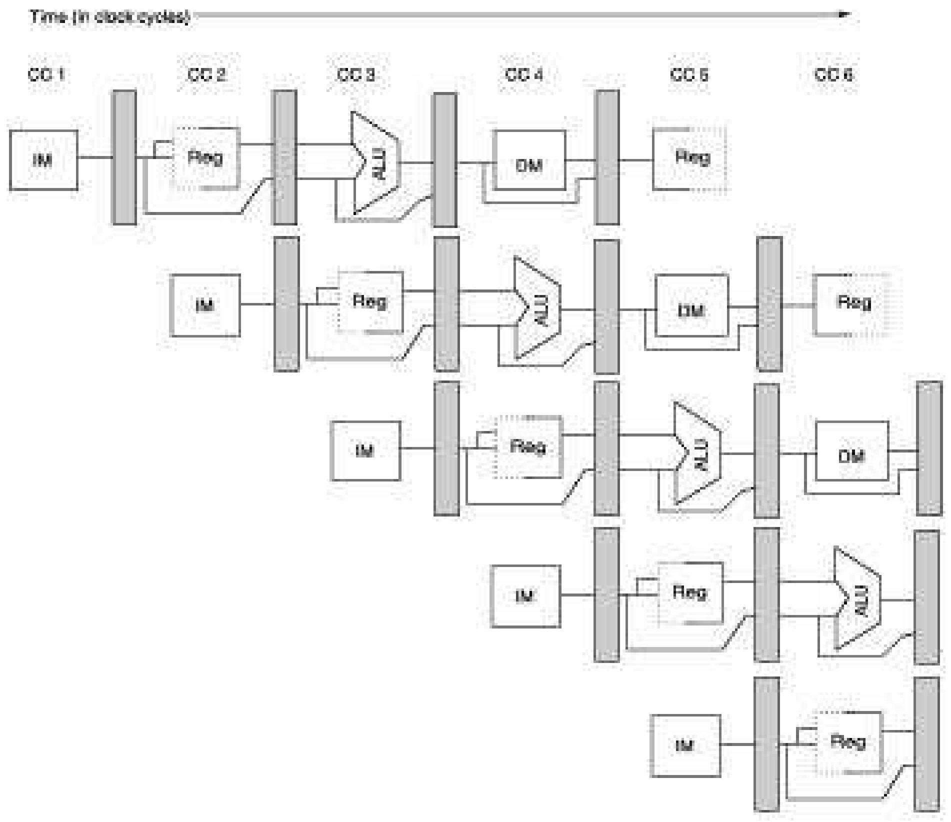
A 5-Stage Pipeline

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



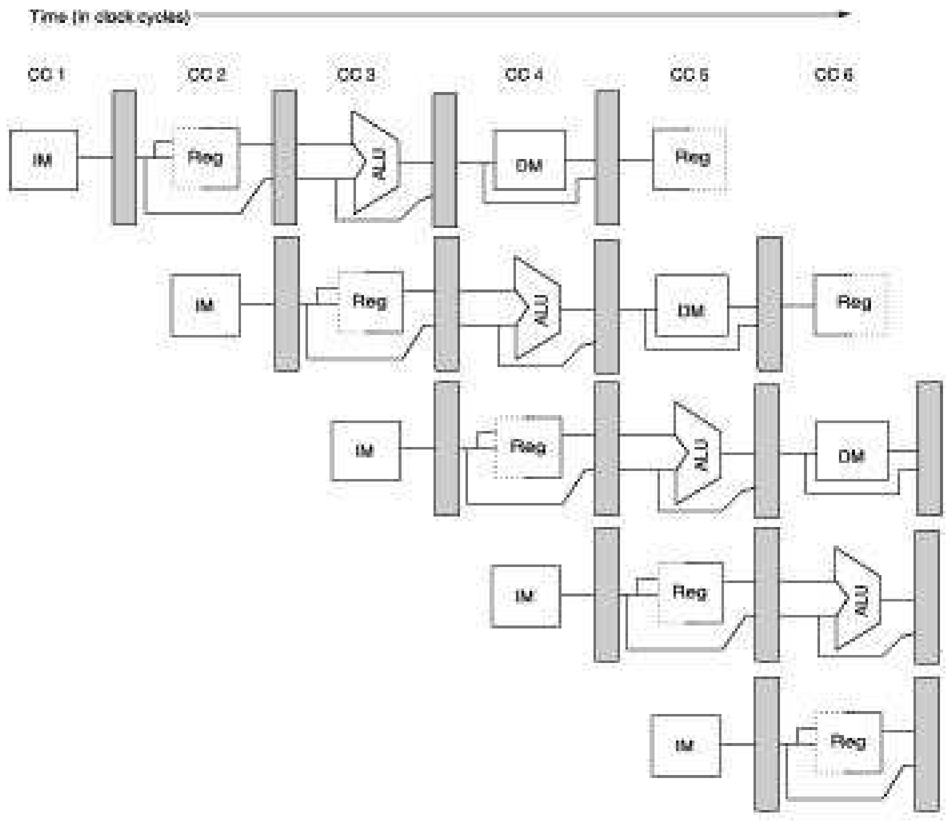
A 5-Stage Pipeline

ALU computation, effective address computation for load/store



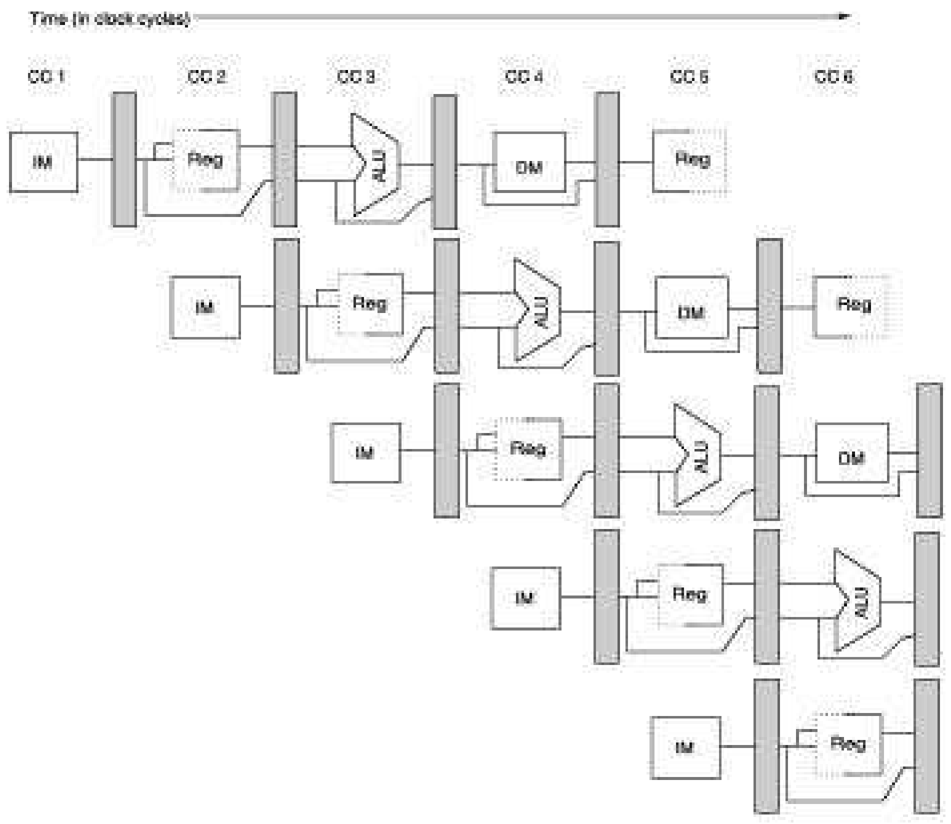
A 5-Stage Pipeline

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)

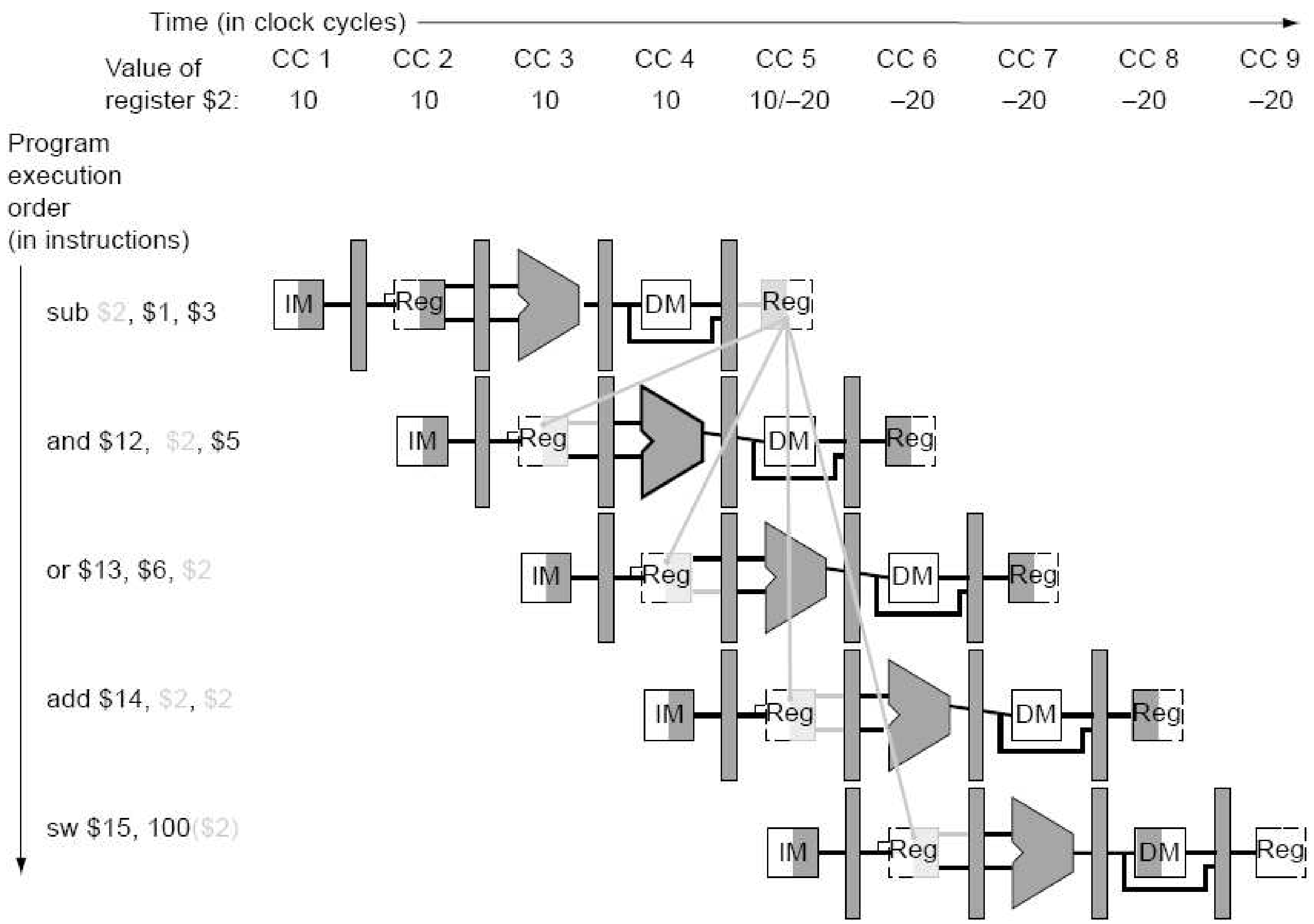
Hazards

- 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)

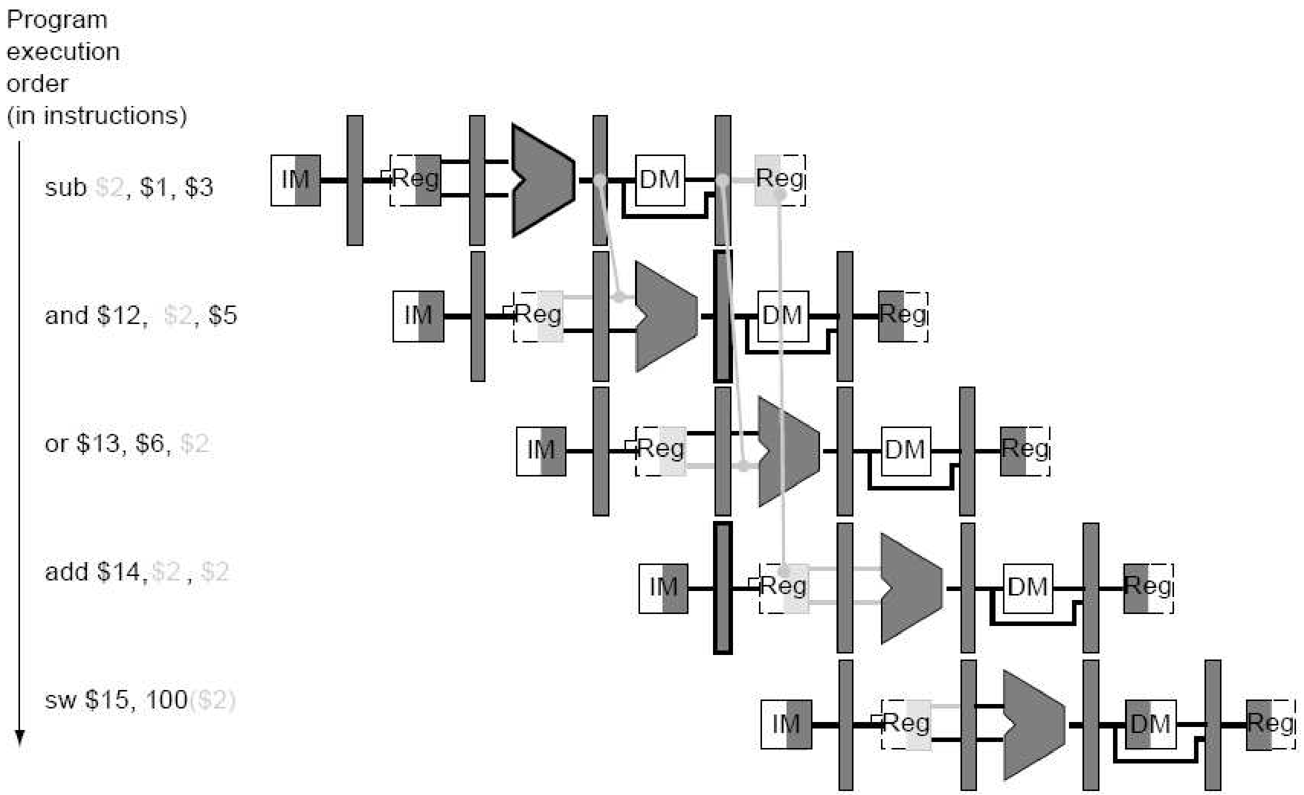
Data Hazards



Bypassing

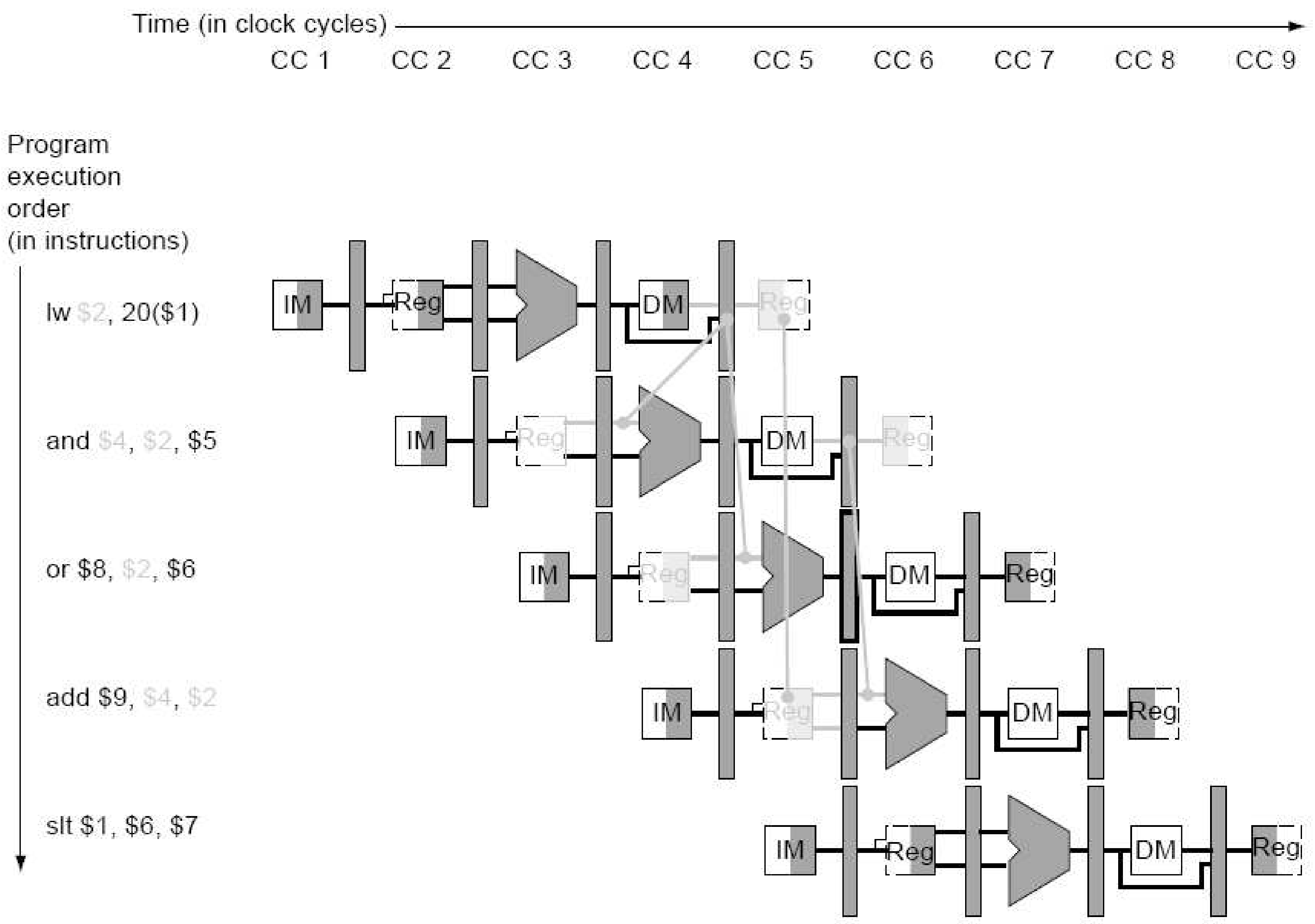
Time (in clock cycles) →

	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

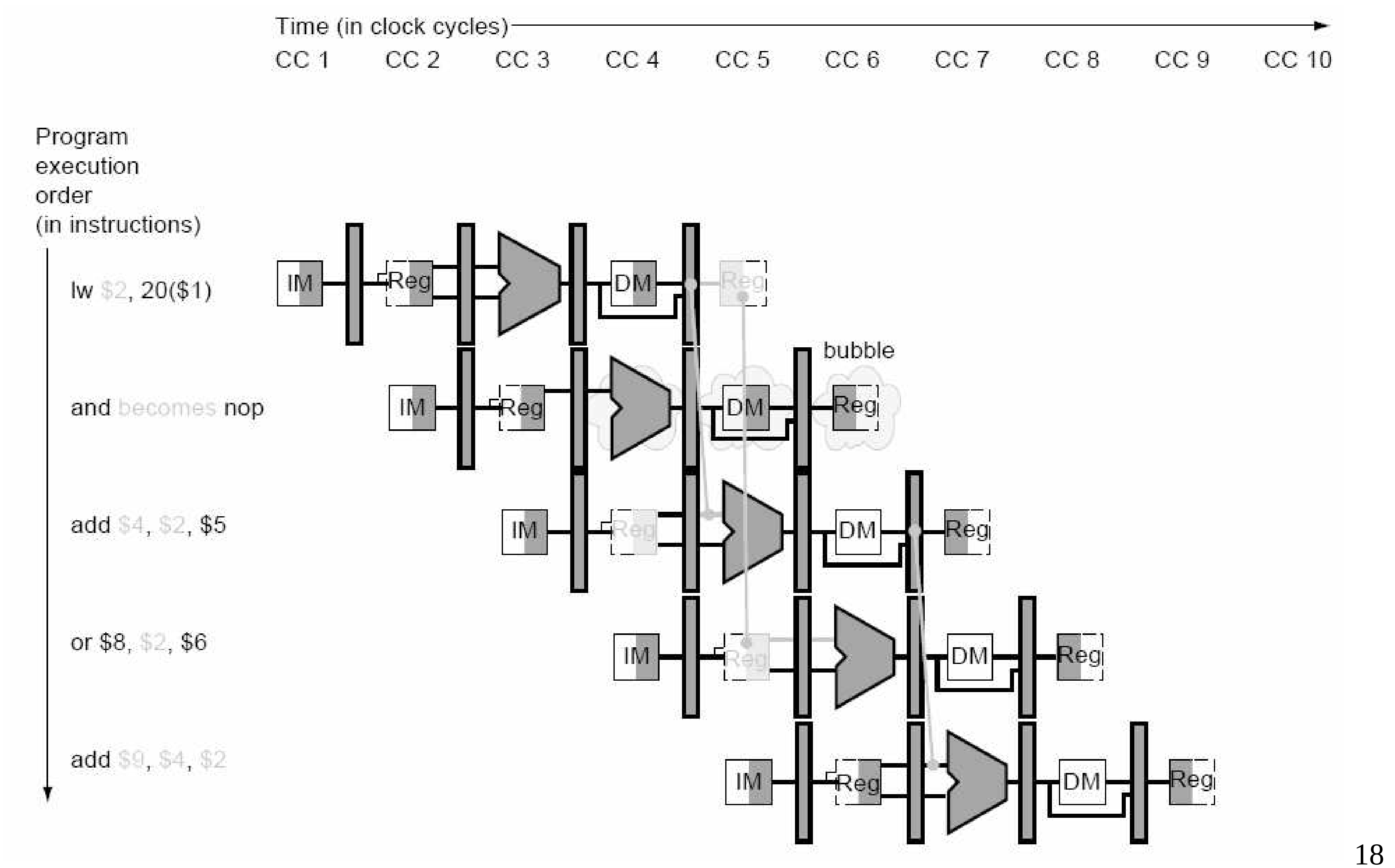


• Some data hazard stalls can be eliminated: bypassing

Data Hazard Stalls



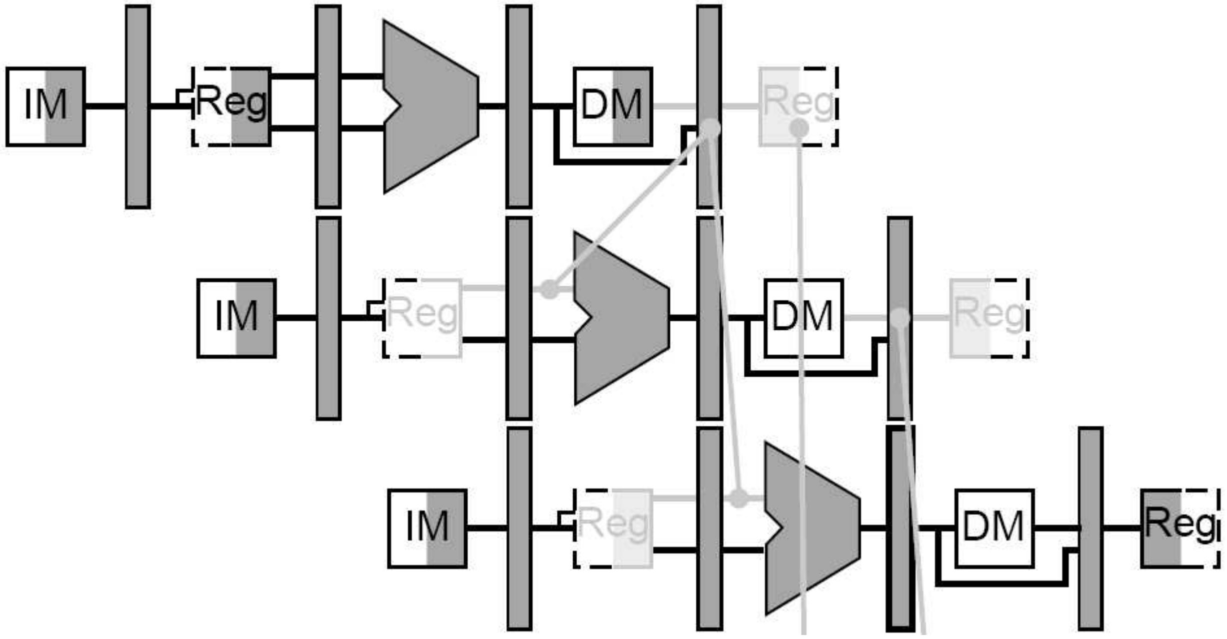
Data Hazard Stalls



Example

add $1, $2, $3

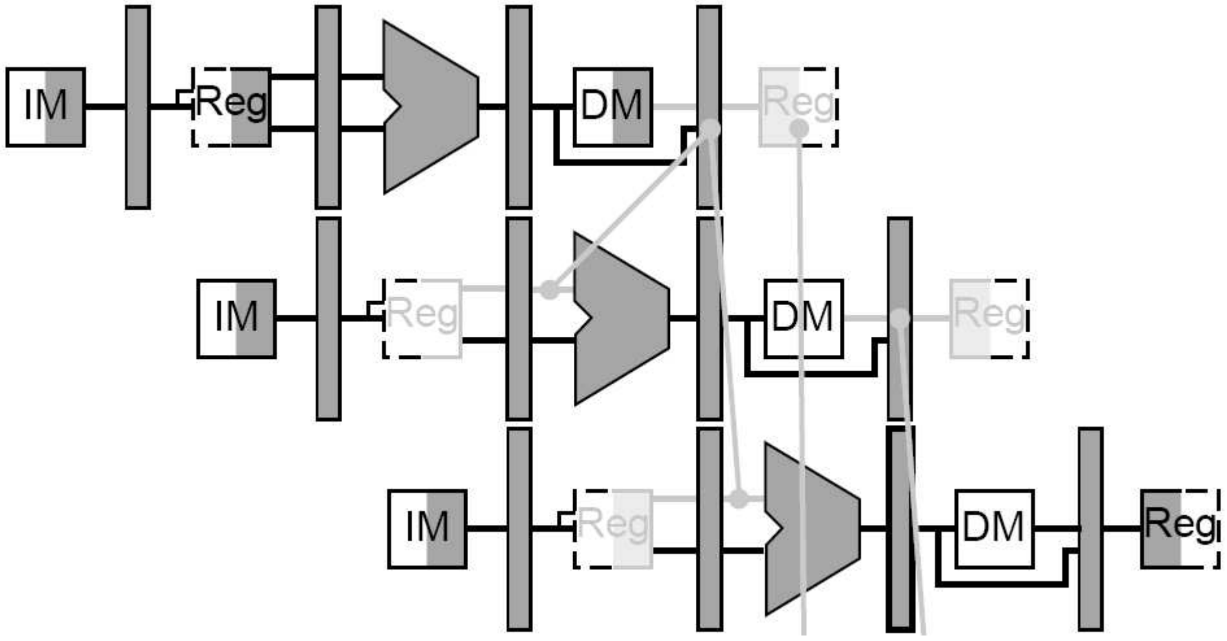
lw $4, 8($1)



Example

lw $1, 8($2)

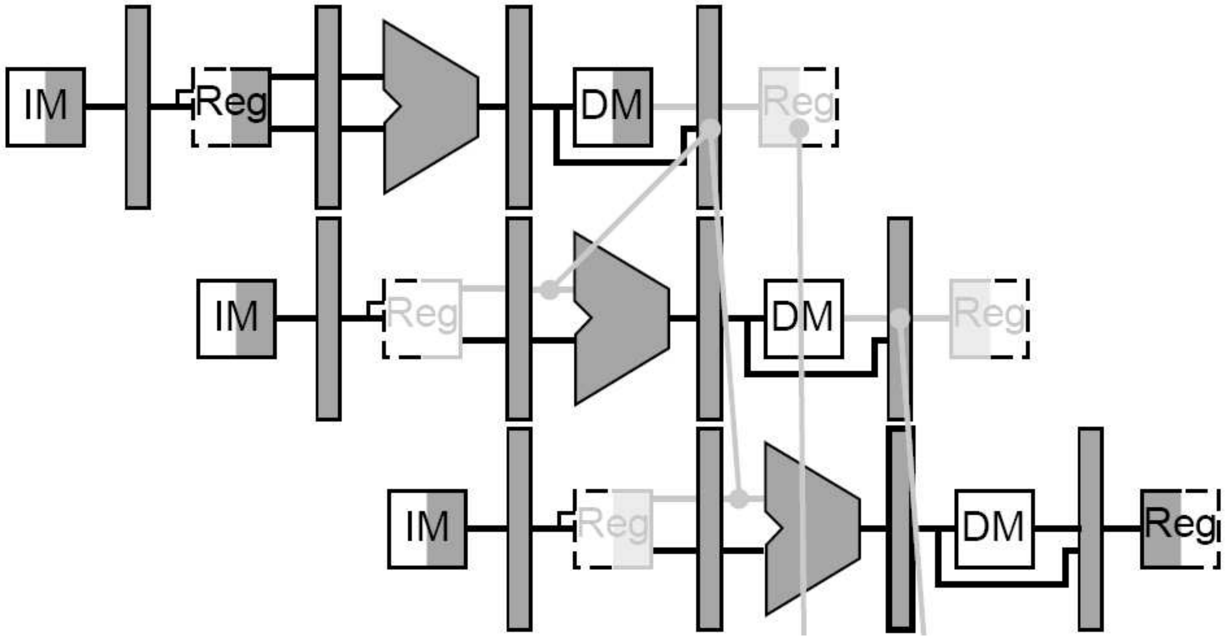
lw $4, 8($1)



Example

lw $1, 8($2)

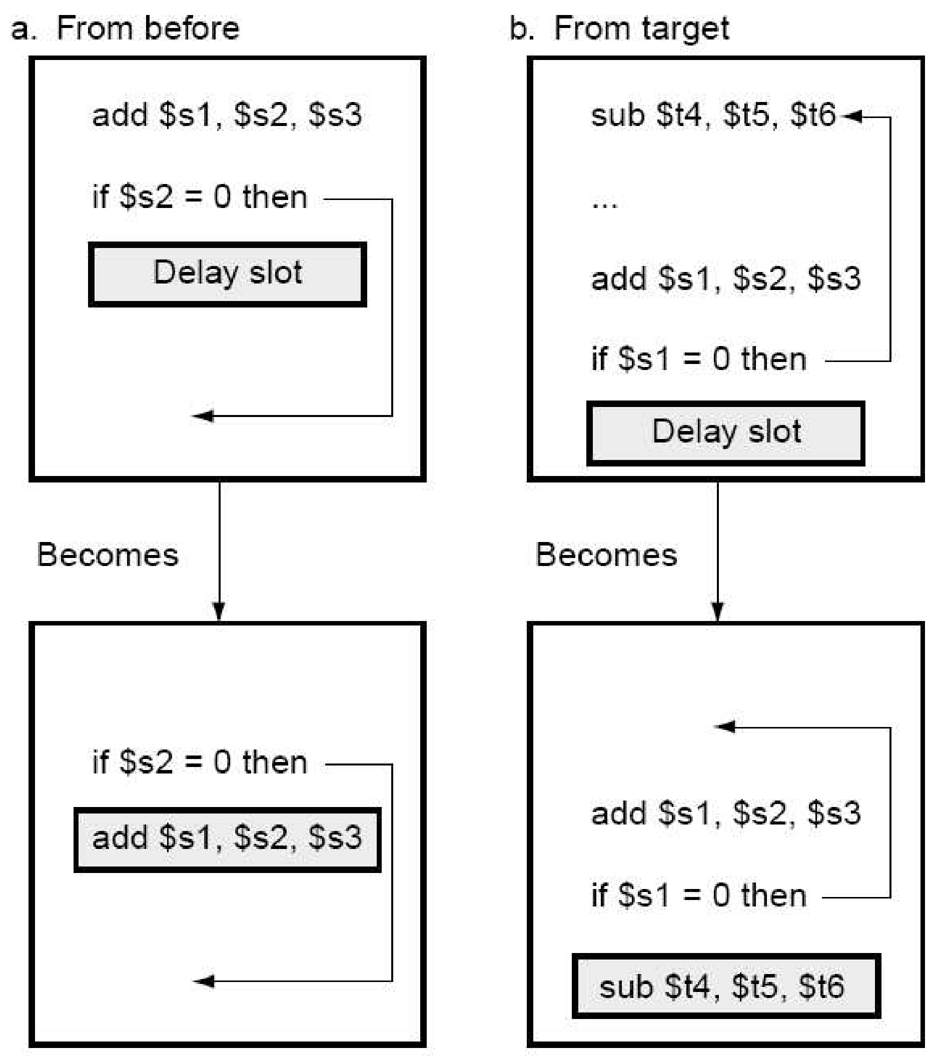
sw $1, 8($3)



Control Hazards

- Simple techniques to handle control hazard stalls:
  - for every branch, introduce a stall cycle (note: every 6th instruction is a branch!)
  - assume the branch is not taken and start fetching the next instruction – if the branch is taken, need hardware to cancel the effect of the wrong-path instruction
  - fetch the next instruction (branch delay slot) and execute it anyway – if the instruction turns out to be on the correct path, useful work was done – if the instruction turns out to be on the wrong path, hopefully program state is not lost

Branch Delay Slots



Slowdowns from Stalls

- Perfect pipelining with no hazards → an instruction completes every cycle (total cycles ~ num instructions) → speedup = increase in clock speed = num pipeline stages
- With hazards and stalls, some cycles (= stall time) go by during which no instruction completes, and then the stalled instruction completes
- Total cycles = number of instructions + stall cycles

Title

- Bullet