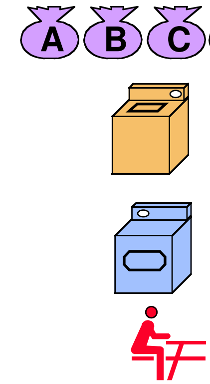
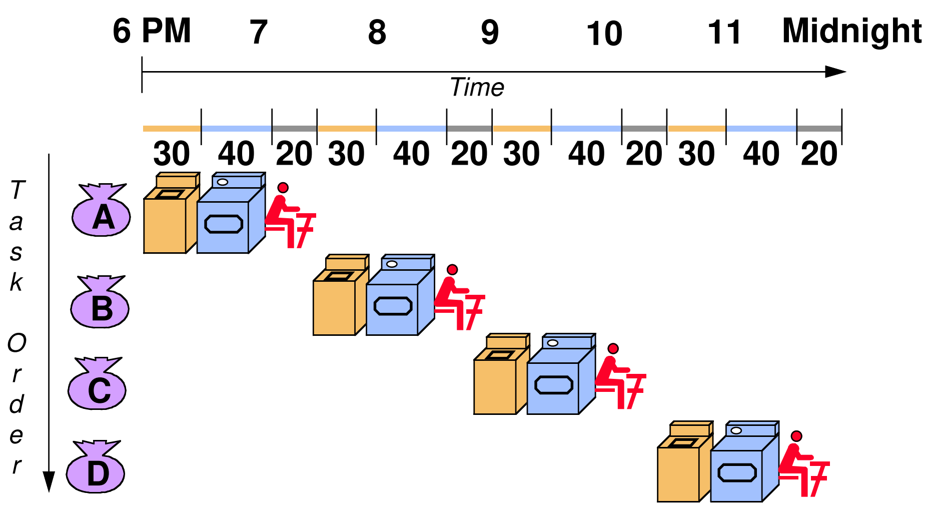
PIPELINING

Pipelining: Its Natural!

- Laundry Example
- Ann, Brian, Cathy, Dave each have one load of clothes to wash, dry, and fold
- Washer takes 30 minutes
- Dryer takes 40 minutes
- “Folder” takes 20 minutes

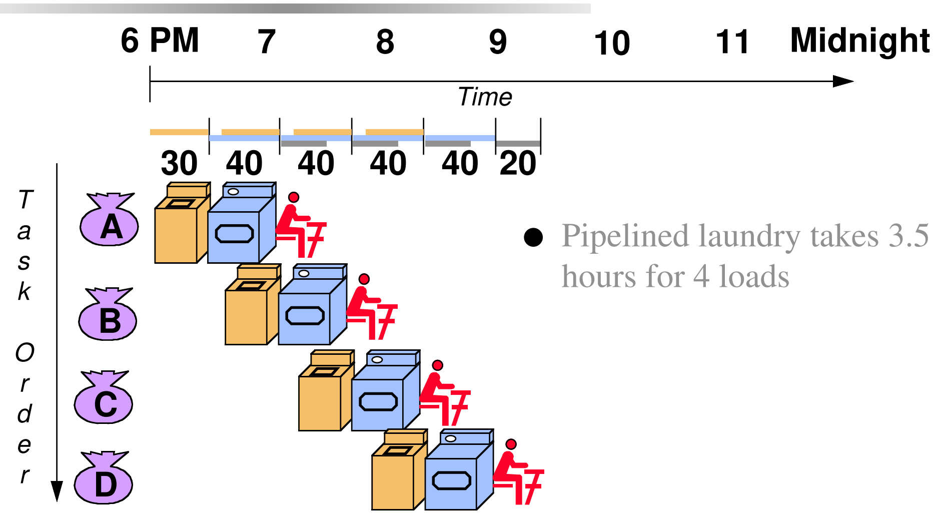


Sequential Laundry



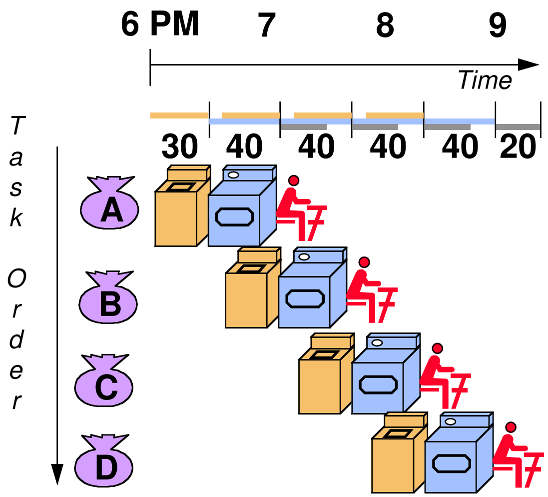
- Sequential laundry takes 6 hours for 4 loads
- If they learned pipelining, how long would laundry take?

Pipelined Laundry Start work ASAP



- Pipelined laundry takes 3.5 hours for 4 loads

Pipelining Lessons



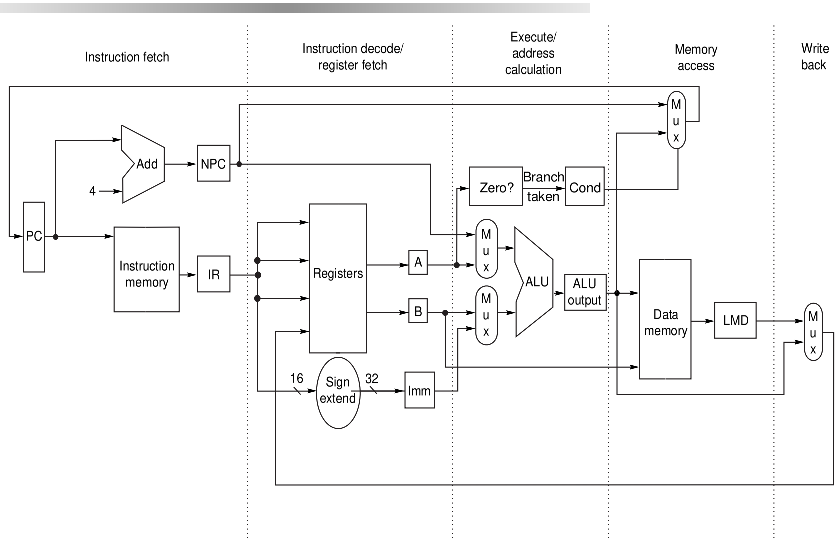
- Pipelining doesn’t help *latency* of single task, it helps *throughput* of entire workload
- Pipeline rate limited by *slowest pipeline stage*
- Multiple tasks operating simultaneously
- Potential speedup = Number pipe stages
- Unbalanced lengths of pipe stages reduces speedup
- Time to “fill” pipeline and time to “drain” it reduces speedup

Pipelining

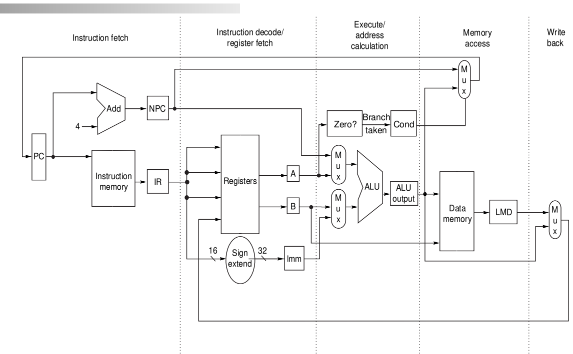
- Requires separable jobs/stages
- Requires separate resources
- Achieves parallelism with replication
- Improves throughput
- Often increases single-task (e.g., instruction, laundry load) latency
- Pipeline efficiency (keeping the pipeline full) critical to performance

5 Steps of the DLX Datapath

Figure 3.1, Page 130



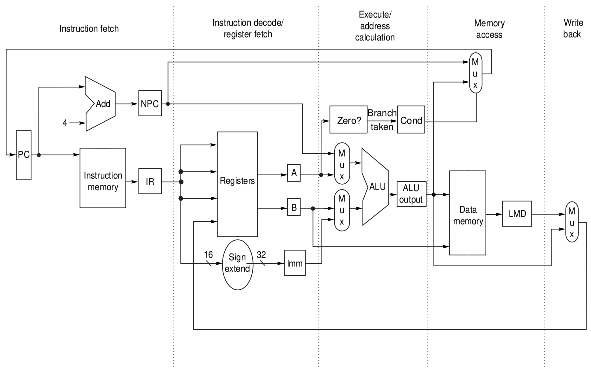
5 Steps of a DLX Instruction



- Instruction Fetch (**IF**)
  - IR <- M[PC]
  - NPC <- PC + 4
- Instruction Decode/register fetch (**ID**)
  - A <- Reg[IR6..10]
  - B <- Reg[IR11..15]
  - Imm <- Sign_extend(IR16..31)

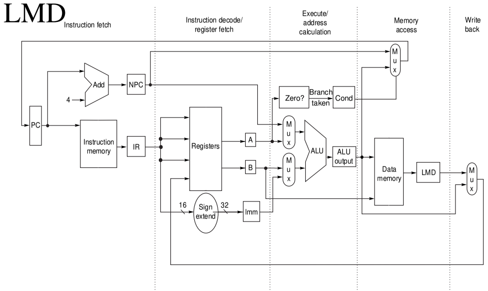
5 Steps of a DLX Instruction

- Execute/Effective Address (**EX**)
  - ALUOutput <- A + Imm (memory ref)
  - ALUOutput <- A op B (register-register alu instruction)
  - ALUOutput <- A op Imm (register-immediate alu instruction)
  - ALUOutput <- NPC + Imm; Cond <- (A op 0) (Branch)

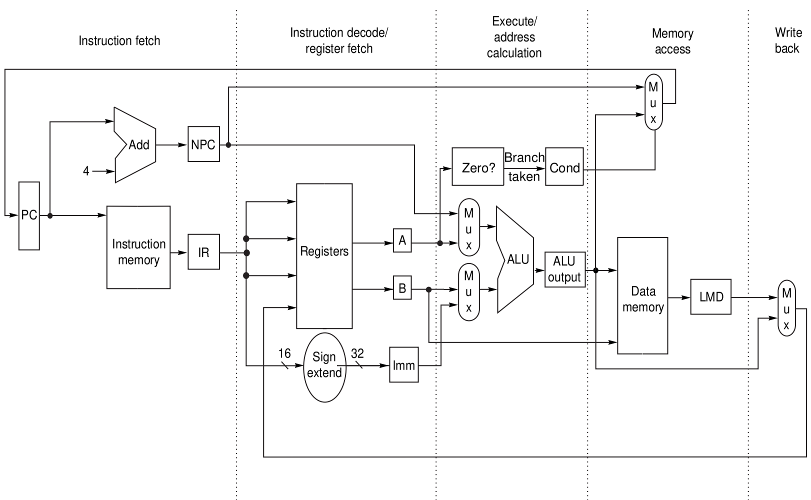


5 Steps of a DLX Instruction

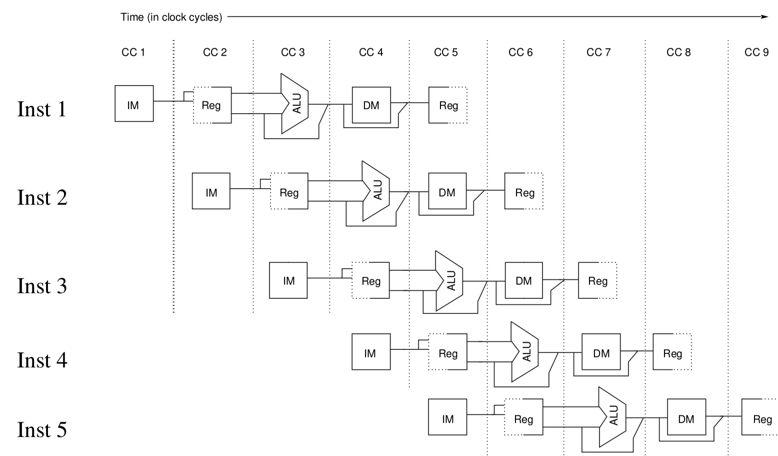
- Memory access/branch completion (**MEM**)
  - LMD <- M[ALUOutput] *or* M[ALUOutput] <- B (load or store)
  - if (cond) PC <- ALUOutput (branch)
    else PC <- NPC
- Write-Back (**WB**)
  - Reg[IR16..20] <- ALUOutput (reg-reg alu instruction)
  - Reg[IR11..15] <- ALUOutput (reg-imm alu instruction)
  - Reg[IR11..15] <- LMD



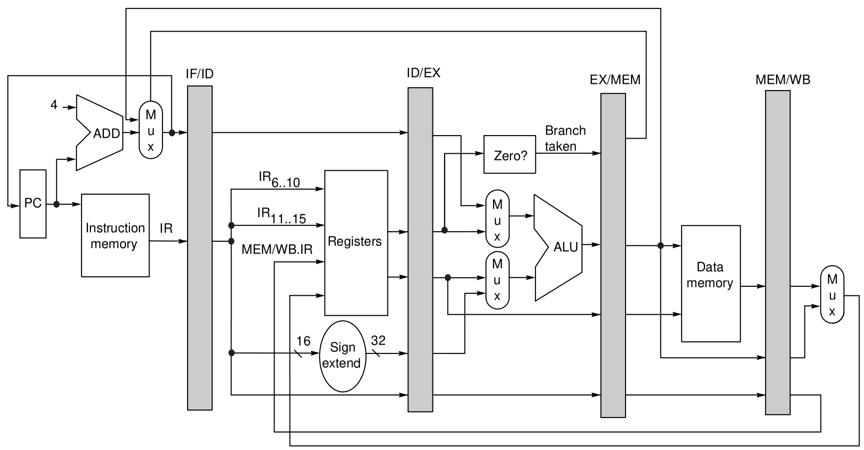
ADDI R7, R2, #35



Visualizing Pipelining



The Pipelined DLX Datapath

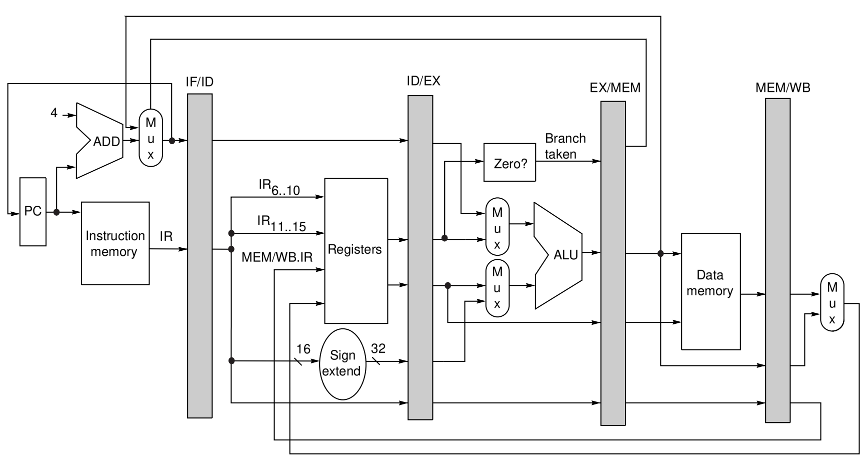


The Pipeline in Motion

- addi R5, R1, #35
- add R6, R2, R1
- lw R8, 10000(R3)

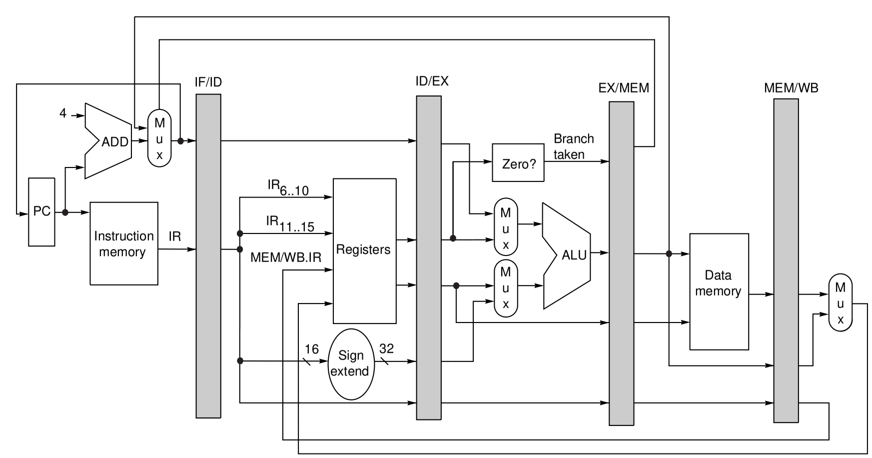
The Pipeline In Motion

addi R5, R1, #35



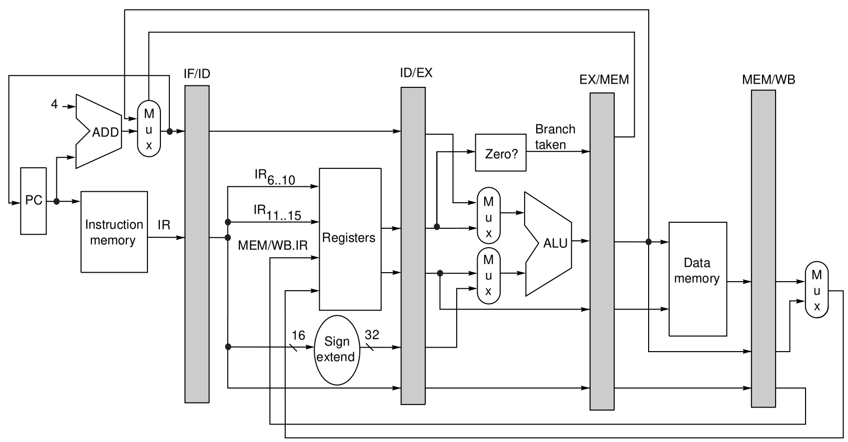
The Pipeline In Motion

add R6, R2, R1 addi R5, R1, #35



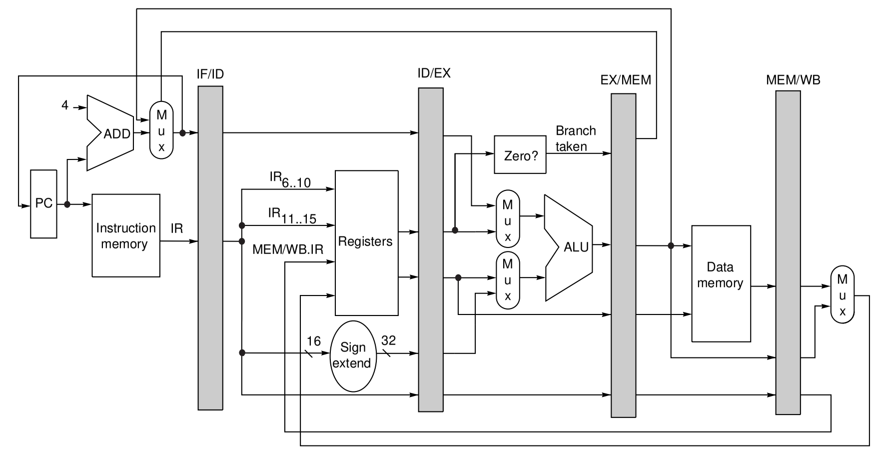
The Pipeline In Motion

lw R8, 10000(R3) add R6, R2, R1 addi R5, R1, #35



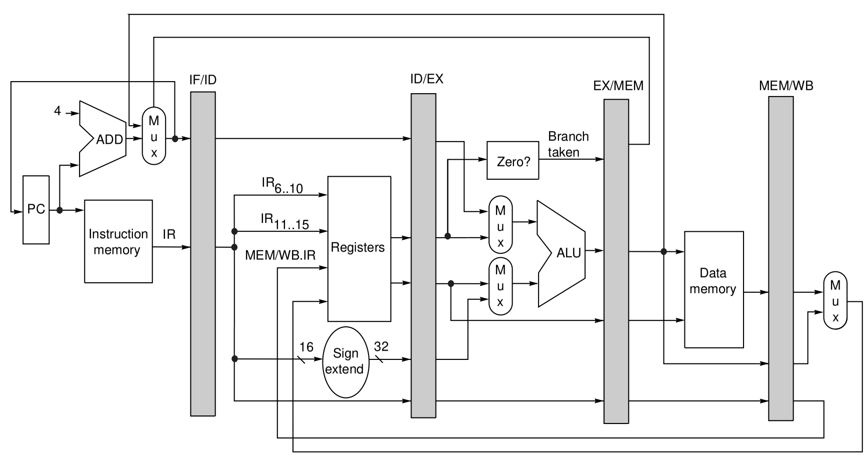
The Pipeline In Motion

lw R8, 10000(R3) add R6, R2, R1 addi R5, R1, #35



The Pipeline In Motion

lw R8, 10000(R3) add R6, R2, R1 addi R5, R1, #35



The Pipeline in Motion

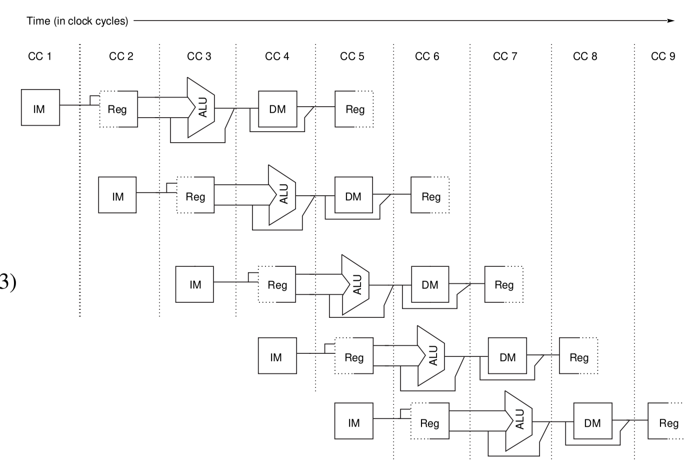
addi R5, R1, #35

add R6, R2, R1

lw R8, 10000(R3)

Inst 4

Inst 5



Pipeline Performace

- ET = IC * CPI * CT
  - single-cycle processor
  - multiple-cycle processor
  - pipelined processor
- complexity has a cost
  - e.g., latch overhead
  - uneven stage latencies
- Can’t always keep the pipeline full
  - why not?

When Things Go Wrong -- Pipeline Hazards

- 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*: Pipelining of branches & other instructions that change the PC
- Common solution is to stall the pipeline until the hazard is resolved, inserting one or more “bubbles” in the pipeline

One Memory Port/Structural Hazards

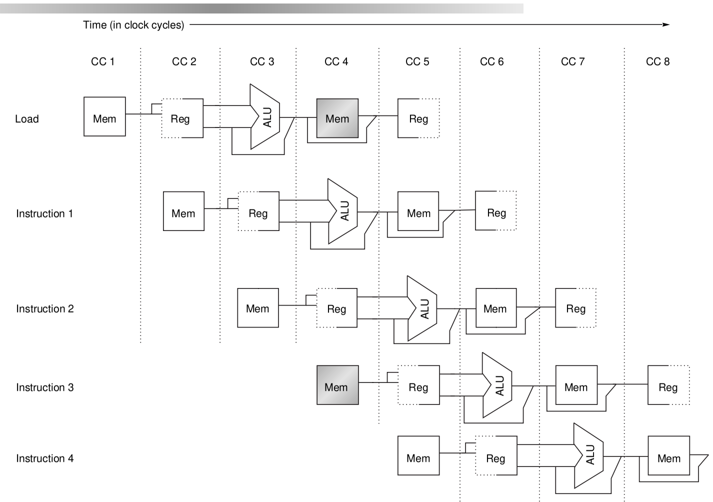


FIGURE 3.6 A machine with only one memory port will generate a conflict whenever a memory reference occurs.

One Memory Port/Structural Hazards

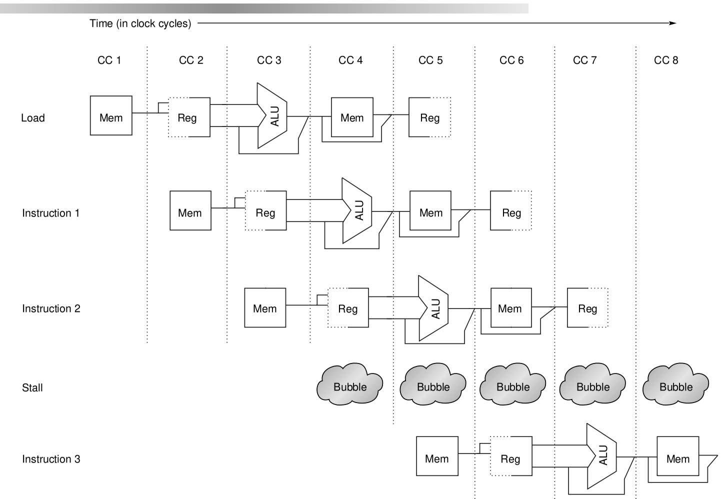


FIGURE 3.7 The structural hazard causes pipeline bubbles to be inserted.

One memory port - alternate view

Load	IF	ID	EX	MEM	WB				
I2		IF	ID	EX	MEM	WB			
I3			IF	ID	EX	MEM	WB		
I4				(stall)	IF	ID	EX	MEM	WB

Example: Dual-port vs. Single-port

- Machine A: Dual ported memory
- Machine B: Single ported memory, but its pipelined implementation has a 1.05 times faster clock rate
- Ideal CPI = 1 for both
- Loads/stores are 30% of instructions executed

Key Points

- Pipeline improves throughput rather than latency
- Pipelining gets parallelism without replication
- ET = IC * CPI * CT
- Keeping the pipeline full is no easy task
  - structural hazards
  - data hazards
  - control hazards