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 without 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

![Datapath Diagram]
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)

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)

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 (load instruction)
ADDI R7, R2, #35

5 Steps of a DLX Instruction

Visualizing Pipelining

The Pipelined DLX Datapath
The Pipeline in Motion

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

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

- 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 – two instructions trying to use same hw resource.
  - **Data hazards**: Instruction depends on result of prior instruction still in the pipeline
  - **Control hazards**: Pipelining of branches & other instructions that change the PC
- One common solution is to stall the pipeline until the hazard is resolved, inserting one or more “bubbles” in the pipeline
One memory port - alternate view

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