// Pls correct if these are wrong thank u :)

# Q.1

### Pipelining

* Key implementation technique for speeding up CPUs
* Breaks each instruction into a series of steps and executes them in parallel
* Clock rate set by the time needed for longest step

### Benefits of Pipelining

* Throughput increased by the depth of the pipeline
* Clock frequency n-times faster than a non-pipelined processor, where n is the number of stages
* Good performance if no stalls

### DLX Microprocessor

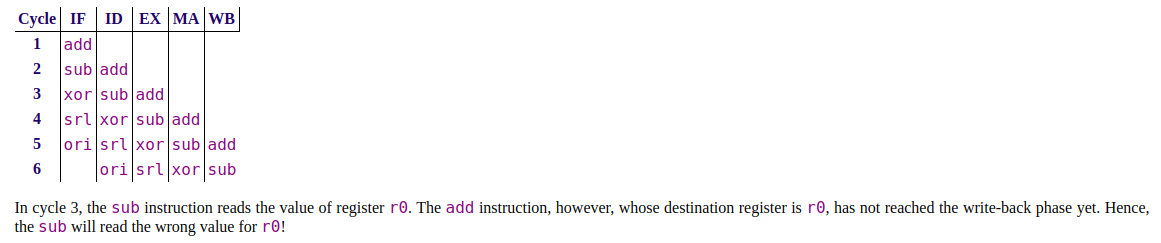
* IF - Instruction Fetch - Sends out the PC and fetches the instruction from memory into the instruction register (IR).
* ID - Instruction Decode & register fetch - Decodes the instruction and accesses the register file to read the registers.
* EX - Execution and effective address calculation - ALU operates on the operand prepared in the prior cycle.
* MA - Memory access - Accesses memory if needed.
* WB - Write back to register - Writes the result into the register file.

// The question asks to explain the operation and organisation but idk how to explain

### 

### Data Hazards

* Occur when instructions that exhibit data dependence modify data in different stages of the pipeline



### Two Techniques that Prevent Stalls that can overcome Data Hazards

1. Pipeline Forwarding
   * All registers are clocked synchronously
   * The ALU results from “previous” two instructions can be forwarded to ALU inputs from ALUOut0 and ALUOut1 pipeline registers before results are written back into register file
2. Two-Phase Clocking
   * DLX register file can be written then read in a single clock cycle
   * Written during first half of the cycle (WB phase)
   * Read during second half (ID phase)
   * No need for third forwarding reg

# Q.4

### What is the Cache Coherency Problem?

* Must guarantee that the CPU is reading the most up to date value of a memory location

### Briefly explain the states and operation of the Firefly cache coherency protocol

* Cache line can be in one of four states:
  + ~Shared & ~Dirty (Exclusive & Clean)
  + Shared & ~Dirty (Shared & Clean)
  + Shared & Dirty
  + ~Shared and Dirty (Exclusive & Dirty)
* There is no INVALID state
* A cache knows if its cache line is shared with other caches
* When a cache lien is read into a cache, the other caches which contain this cache line assert a common SHARED bus line
* Writes to exclusive cache lines are write-back
* Writes to shared cache lines are write-through, each cache that contains this cache line is updated along with memory (write-update)
* When a cache line is no longer shared, it needs an additional write-through cycle to find out that it is no longer shared (SHARED signal not asserted)
* Sharing may be illusory e.g. a processor may no longer be using a shared cache line

//idk if you need the state transition diagram

### Bus Traffic and Cache Line State Transitions

|  |  |  |
| --- | --- | --- |
| 1 | CPU0 : read a0 | *CPU0 reads a0 from cache - state S~D* |
| 2 | CPU0: read a2 | *CPU0 reads a2 from memory - state ~S~D* |
| 3 | CPU0: write a2 | *CPU0 updates a2 in cache ONLY - state ~SD* |
| 4 | CPU0: write a2 | *CPU0 updates a2 in cache ONLY - state ~SD* |
| 5 | CPU1: read a2 | *CPU1 reads a2, CPU0 cache intervenes and supplies data - state SD [NB: memory NOT updated]* |
| 6 | CPU2: read a2 | *CPU2 reads a2, CPU0 cache intervenes and supplies data - state SD [NB: memory NOT updated]* |
| 7 | CPU0: write a2 | CPU0 updates a2 in cache ONLY - state ~SD |
| 8 | CPU0: write a2 | CPU0 updates a2 in cache ONLY - state ~SD |
| 9 | CPU0: read a0 | CPU0 reads a0 from memory - state S~D |
| 10 | CPU1: read a0 | CPU1 reads a0 from cache - S~D |
| 11 | CPU2: write a2 | CPU2 updates a2 in cache ONLY - state SD |
| 12 | CPU2: write a2 | CPU2 updates a2 in cache ONLY - state SD |