> Pipelining

Execution Instructions

1) Increment PC, fetch opcode for next instruction
2) Decode opcode to determine operands
3) Route the operands to ALU
4) ALU operation
5) Store ALU output in register

- Normally these are done one at a time per inst. Each go thru 5 phases.
- W/ pipeline, we can execute seperate phases of consecutive inst concurrently

Pipeline Latency: Time for inst to traverse all phases

Pipe Throughput: $\frac{\text{no. inst}}{\text{time}}$

Pipeline stall: Occurs when some portion of CPU must wait for another portion to finish.

↳ Must wait one latency cycle to CPU can start completing inst

↳ Causes:
- Cache / TLB miss
- Data hazard: Dependencies b/w instruction results
  - ↳ fixed by data forwarding, directly forward result from ALU
    - requires extra hardware
- Conditional branches: Inst that alter program flow based on condition.
  - ↳ fixed by branch target cache, store predicted branch target

Parallelism: Processing inst in parallel requires no mutual dependencies b/w operands

> Super Scalar Processing

CPU manage multiple pipelines. N ALUs potentially N times faster than CPU w/ 1 ALU. High Inst level parallelism required.

Limitations:

- Data dependency: 2 inst attempt to write to same destination reg, order of execution impacts outcome.
  - ↳ Fixed by: Stalling / data forwarding
- Procedural dependency: Branch outcome causes delay.
  - ↳ Fixed by: Stalling or speculative execution: Predict branch outcome very expensive
- Resource conflict: Not enough execution units
  - ↳ Fixed by: Adding more EUs or better resource allocation
- Anti-dependency: Inst requires write to reg that a previous inst reads
  - ↳ Not an issue in pipelined CPU as in order
    - But super scalar CPU uses Out of Order Execution may need stall to fix anti-dependency