Review: Pipelining

Pipelining Analogy

- Pipelined laundry: overlapping execution
  - Parallelism improves performance

- Four loads:
  - Speedup
    \[ \frac{8}{3.5} = 2.3 \]

- Non-stop:
  - Speedup
    \[ \frac{2n}{0.5n + 1.5} \approx 4 \]
    - number of stages
MIPS Pipeline

- Five stages, one step per stage
  1. IF: Instruction fetch from memory
  2. ID: Instruction decode & register read
  3. EX: Execute operation or calculate address
  4. MEM: Access memory operand
  5. WB: Write result back to register

Pipeline Performance

Single-cycle ($T_c = 800\text{ps}$)

Pipelined ($T_c = 200\text{ps}$)
### Pipeline Speedup

- If all stages are balanced
  - i.e., all take the same time
  - Time between instructions \( t_{\text{pipelined}} \)
    \[
    t_{\text{pipelined}} = \frac{t_{\text{nonpipelined}}}{\text{Number of stages}}
    \]
- If not balanced, speedup is less
- Speedup due to increased throughput
  - Latency (time for each instruction) does not decrease

### Hazards

- Situations that prevent starting the next instruction in the next cycle
- Structure hazards
  - A required resource is busy
- Data hazard
  - Need to wait for previous instruction to complete its data read/write
- Control hazard
  - Deciding on control action depends on previous instruction
Structure Hazards

- Conflict for use of a resource
- In MIPS pipeline with a single memory
  - Load/store requires data access
  - Instruction fetch would have to *stall* for that cycle
    - Would cause a pipeline “bubble”
- Hence, pipelined datapaths require separate instruction/data memories
  - Or separate instruction/data caches

Data Hazards

- An instruction depends on completion of data access by a previous instruction
  - `add $s0, $t0, $t1`
  - `sub $t2, $s0, $t3`
Control Hazards

- Branch determines flow of control
  - Fetching next instruction depends on branch outcome
  - Pipeline can’t always fetch correct instruction
    - Still working on ID stage of branch

- In MIPS pipeline
  - Need to compare registers and compute target early in the pipeline
  - Add hardware to do it in ID stage

Stall on Branch

- Wait until branch outcome determined before fetching next instruction
Branch Prediction

- Longer pipelines can’t readily determine branch outcome early
  - Stall penalty becomes unacceptable
- Predict outcome of branch
  - Only stall if prediction is wrong
- In MIPS pipeline
  - Can predict branches not taken
  - Fetch instruction after branch, with no delay

More-Realistic Branch Prediction

- Static branch prediction
  - Based on typical branch behavior
  - Example: loop and if-statement branches
    - Predict backward branches taken
    - Predict forward branches not taken
- Dynamic branch prediction
  - Hardware measures actual branch behavior
    - e.g., record recent history of each branch
  - Assume future behavior will continue the trend
    - When wrong, stall while re-fetching, and update history
Pipeline registers

- Need registers between stages
- To hold information produced in previous cycle
Multi-Cycle Pipeline Diagram

Form showing resource usage

Multi-Cycle Pipeline Diagram

Traditional form