Pipelining

- Recompose the data path we just did.
- Each instruction takes from 3 to 5 clock cycles.
- However, there are parts of hardware that are idle many times.
- We can reorganize the operation.
  - Make each hardware block independent:
    1. Instruction Fetch Unit
    2. Register Read Unit
    3. ALU Unit
    4. Data Memory Read/Write Unit
    5. Register Write Unit
- Units in 3 and 5 cannot be independent, but operations can be.
- Let each unit do its required job for each instruction.
  - If for some instruction, a unit need not do anything, it can simply perform a noop.

Gain of Pipelining

- Improve performance by increasing instruction throughput.
- Ideal speedup is number of stages in the pipeline.
- Do we achieve this? No, why not?

Basic Idea

- What do we need to add to actually split the datapath into stages?

Pipelined Data Path

Can you find a problem even if there are no dependencies? What instructions can we execute to manifest the problem?

Corrected Data Path
Pipeline Operation

- In pipeline one operation begins in every cycle
- Also, one operation completes in each cycle
- Each instruction takes 5 clock cycles (k cycles in general)
- When a stage is not used, no control needs to be applied
- Different stages are executing different instructions
- How to generate control signals for them is an issue

Graphically Representing Pipelines

- Can help with answering questions like:
  - how many cycles does it take to execute this code?
  - what is the ALU doing during cycle 4?
  - use this representation to help understand datapaths

Pipeline Control

- We have 5 stages. What needs to be controlled in each stage?
  - Instruction Fetch and PC increment
  - Instruction Decode / Register Fetch
  - Execution
  - Memory Stage
  - Write Back
- How would control be handled in an automobile plant?
  - a fancy control center telling everyone what to do?
  - should we use a finite state machine?

Pipeline Control

- Pass control signals along just like the data

Data Path with Control
Dependencies

• Problem with starting next instruction before first is finished
  – dependencies that “go backward in time” are data hazards

Solution: Software No-ops/Hardware Bubbles

• Have compiler guarantee no hazards
• Where do we insert the “no-ops”?
  
  ```
  sub $2, $1, $3
  and $12, $2, $5
  or $13, $6, $2
  add $14, $2, $2
  sw $15, 100($2)
  ```

  Problem: this really slows us down!
  – Also, the program will always be slow even if a technique like forwarding is employed afterwards in newer version
• Hardware can detect dependencies and insert no-ops in hardware by not accepting a new instruction
  – This is a bubble in pipeline and waste one cycle at all stages
  – Need two or three bubbles between write and read of a register

Stalling

• Hardware detection and no-op insertion is called stalling
• We stall the pipeline by keeping an instruction in the same stage

Forwarding

• Use temporary results, don’t wait for them to be written
  – register file forwarding to handle read/write to same register
  – ALU forwarding

Can’t always forward

• Load word can still cause a hazard:
  – an instruction tries to read a register following a load instruction that writes to the same register.

  ```
  lw $2, 20($1)
  ```

  Thus, we need a hazard detection unit to “stall” the load instruction