






Computer System Architecture

Pipelining Part I

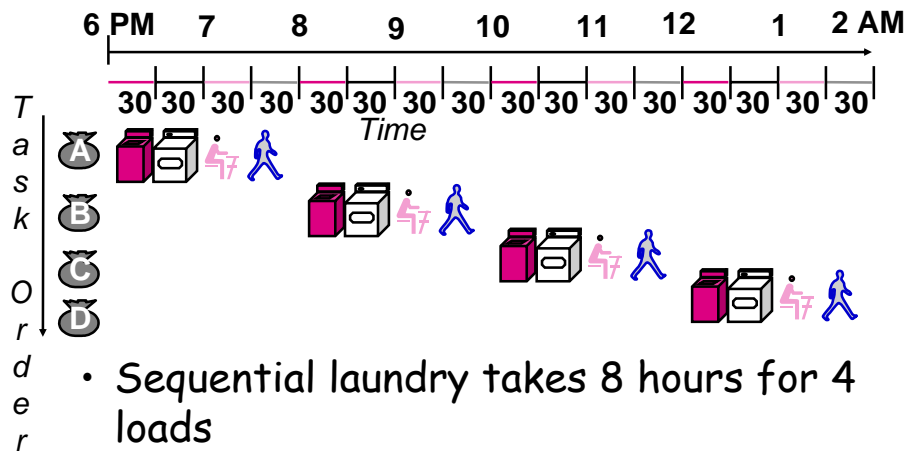
Chalermek Intanagonwiwat

Slides courtesy of David Patterson

Pipelining is Natural!

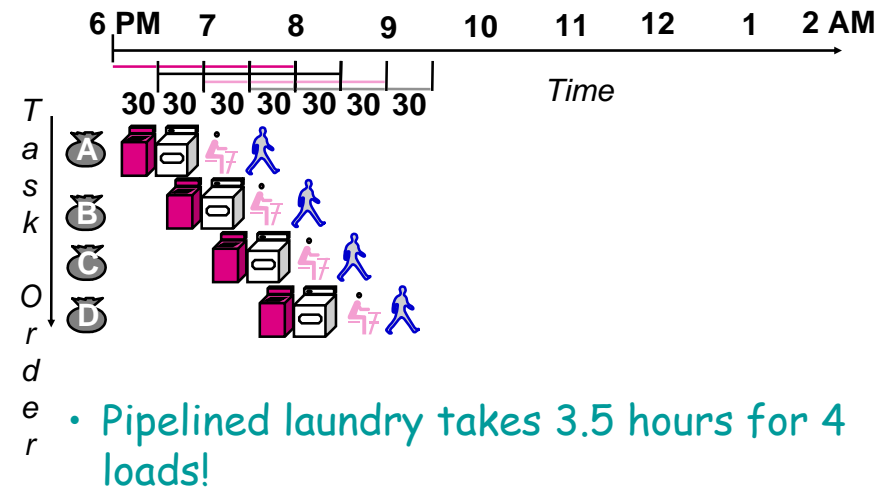
- Laundry Example 
- Ann, Brian, Cathy, Dave each have one load of clothes to wash, dry, and fold
- Washer takes 30 minutes 
- Dryer takes 30 minutes 
- "Folder" takes 30 minutes 
- "Stasher" takes 30 minutes to put clothes into drawers 

Sequential Laundry



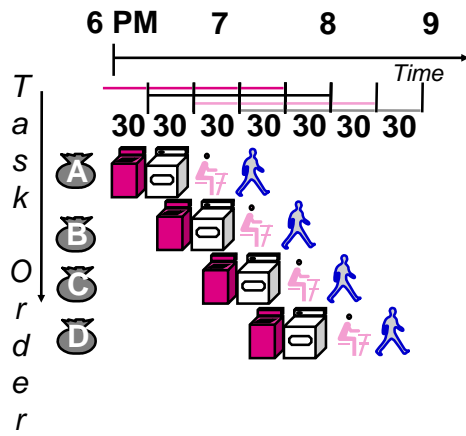
- Sequential laundry takes 8 hours for 4 loads
- If they learned pipelining, how long would laundry take?

Pipelined Laundry: Start work ASAP



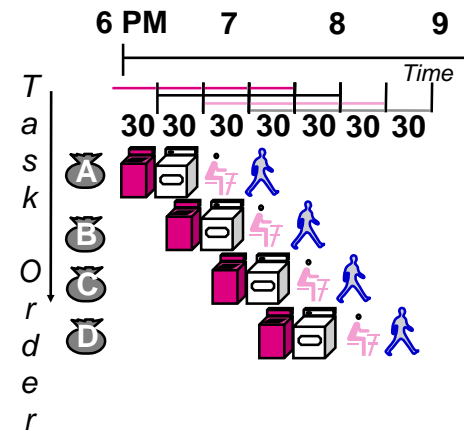
- Pipelined laundry takes 3.5 hours for 4 loads!

Pipelining Lessons



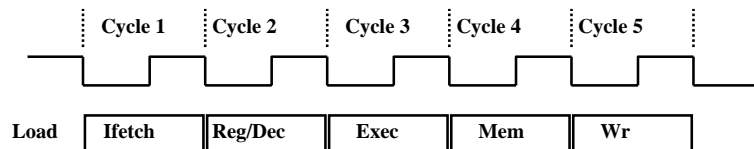
- Pipelining doesn't help **latency** of single task, it helps **throughput** of entire workload
- **Multiple** tasks operating simultaneously using different resources
- Potential speedup = **Number pipe stages**

Pipelining Lessons (cont.)



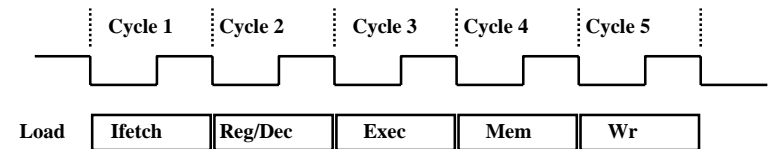
- Pipeline rate limited by **slowest** pipeline stage
- Unbalanced lengths of pipe stages reduces speedup
- Time to "fill" pipeline and time to "drain" it reduces speedup
- Stall for Dependences

The Five Stages of Load



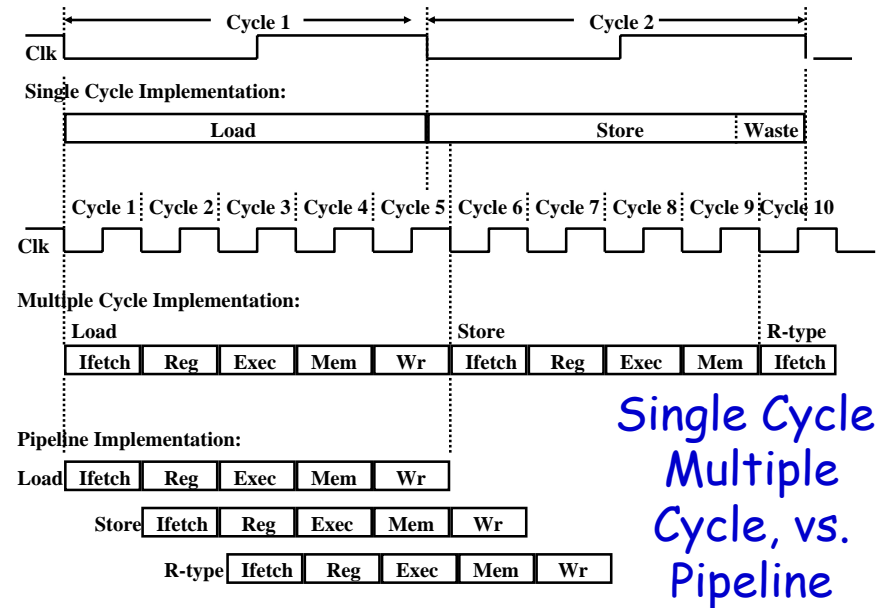
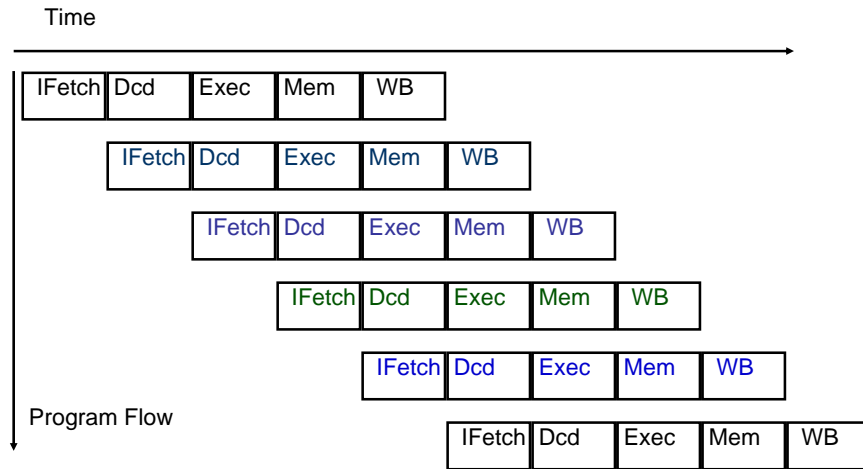
- **Ifetch**: Instruction Fetch
 - Fetch the instruction from the Instruction Memory
- **Reg/Dec**: Registers Fetch and Instruction Decode
- **Exec**: Calculate the memory address

The Five Stages of Load (cont.)



- **Mem**: Read the data from the Data Memory
- **Wr**: Write the data back to the register file

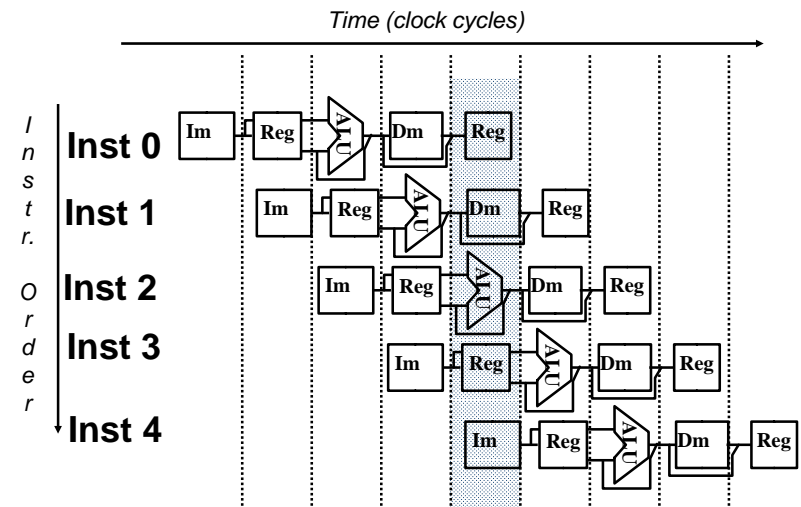
Conventional Pipelined Execution Representation



Why Pipeline?

- Suppose we execute 100 instructions
- Single Cycle Machine
 - $45 \text{ ns/cycle} \times 1 \text{ CPI} \times 100 \text{ inst} = 4500 \text{ ns}$
- Multicycle Machine
 - $10 \text{ ns/cycle} \times 4.6 \text{ CPI (due to inst mix)} \times 100 \text{ inst} = 4600 \text{ ns}$
- Ideal pipelined machine
 - $10 \text{ ns/cycle} \times (1 \text{ CPI} \times 100 \text{ inst} + 4 \text{ cycle drain}) = 1040 \text{ ns}$

Why Pipeline? Because the resources are there!



Can pipelining get us into trouble?

- **Yes: Pipeline Hazards**
 - **structural hazards**: attempt to use the same resource two different ways at the same time
 - E.g., combined washer/dryer would be a structural hazard or folder busy doing something else (watching TV)

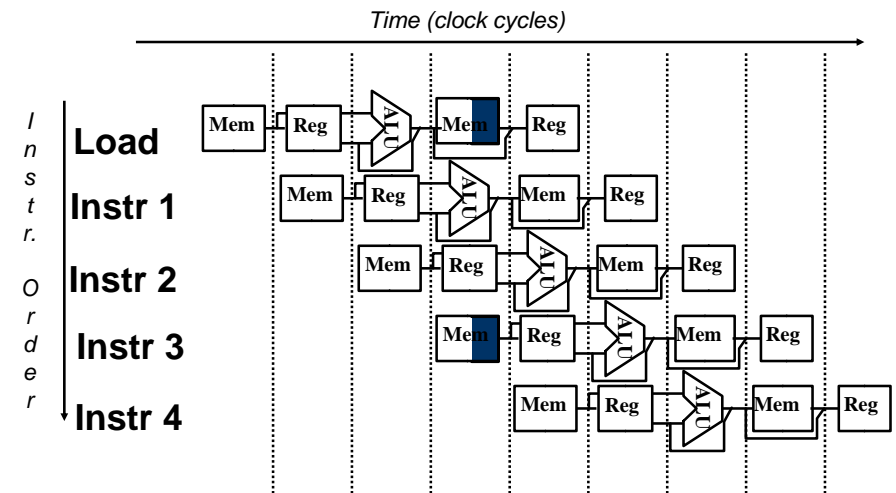
Can pipelining get us into trouble? (cont.)

- **data hazards**: attempt to use item before it is ready
 - E.g., one sock of pair in dryer and one in washer; can't fold until get sock from washer through dryer
 - instruction depends on result of prior instruction still in the pipeline

Can pipelining get us into trouble? (cont.)

- **control hazards**: attempt to make a decision before condition is evaluated
 - E.g., washing football uniforms and need to get proper detergent level; need to see after dryer before next load in
 - branch instructions
- Can always resolve hazards by **waiting**
 - pipeline control must detect the hazard
 - take action (or delay action) to resolve hazards

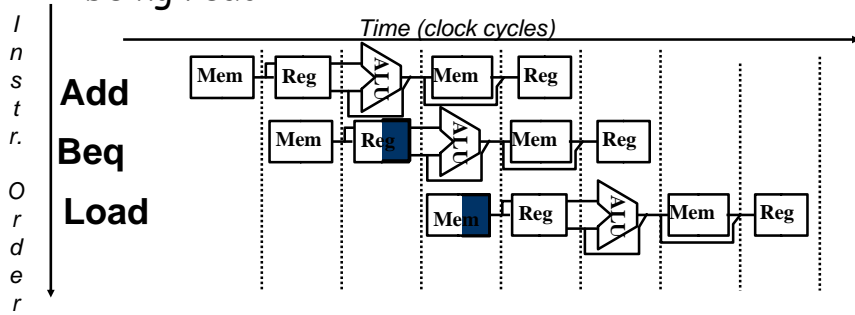
Single Memory is a Structural Hazard



Detection is easy in this case! (right half highlight means read, left half write)

Control Hazard Solutions

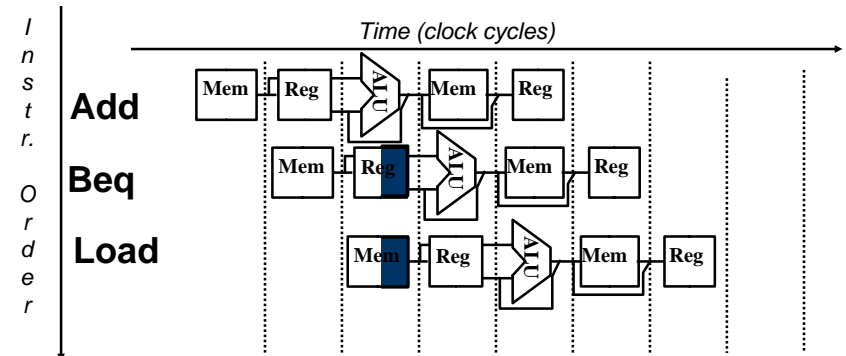
- **Stall:** wait until decision is clear
 - Its possible to move up decision to 2nd stage by adding hardware to check registers as being read



- Impact: 2 clock cycles per branch instruction => slow

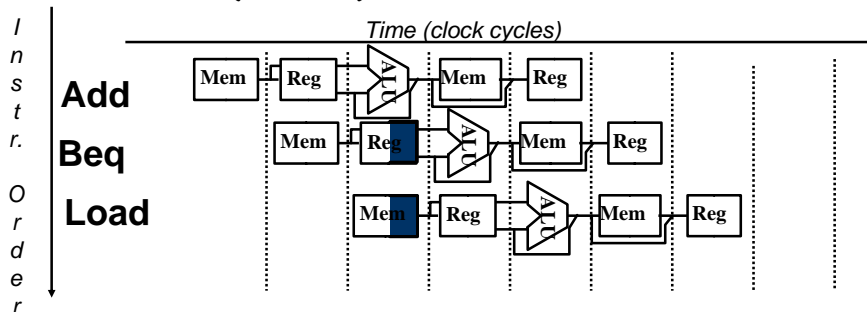
Control Hazard Solutions (cont.)

- **Predict:** guess one direction then back up if wrong
 - Predict not taken



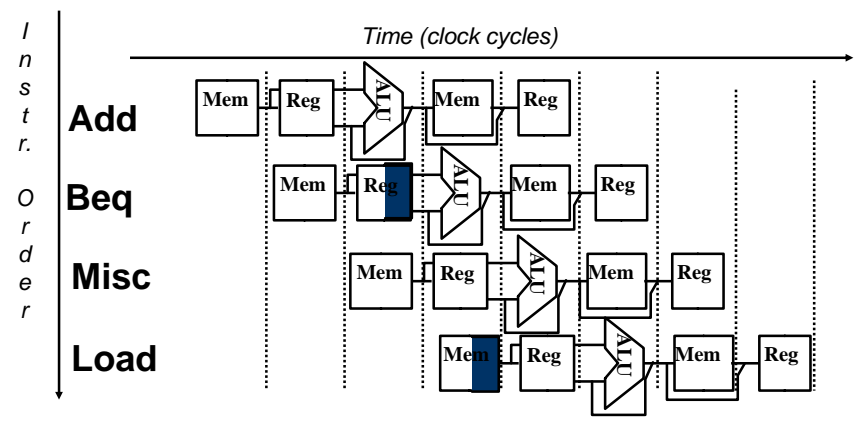
Control Hazard Solutions (cont.)

- Impact: 1 clock cycles per branch instruction if right, 2 if wrong (right - 50% of time)
- More dynamic scheme: history of 1 branch (- 90%)



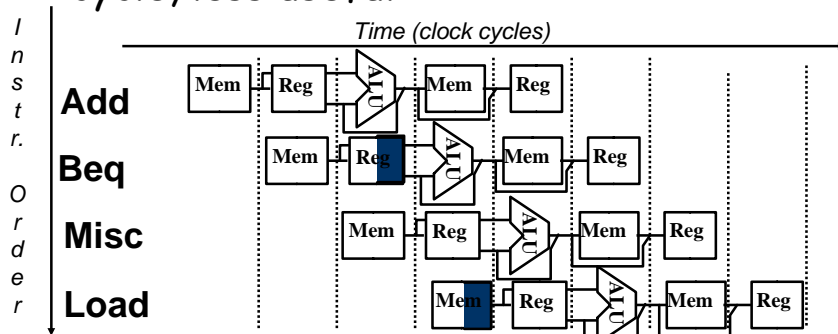
Control Hazard Solutions (cont.)

- **Redefine branch behavior** (takes place after next instruction) "delayed branch"



Control Hazard Solutions (cont.)

- Impact: 0 clock cycles per branch instruction if can find instruction to put in "slot" (- 50% of time)
- As launch more instruction per clock cycle, less useful



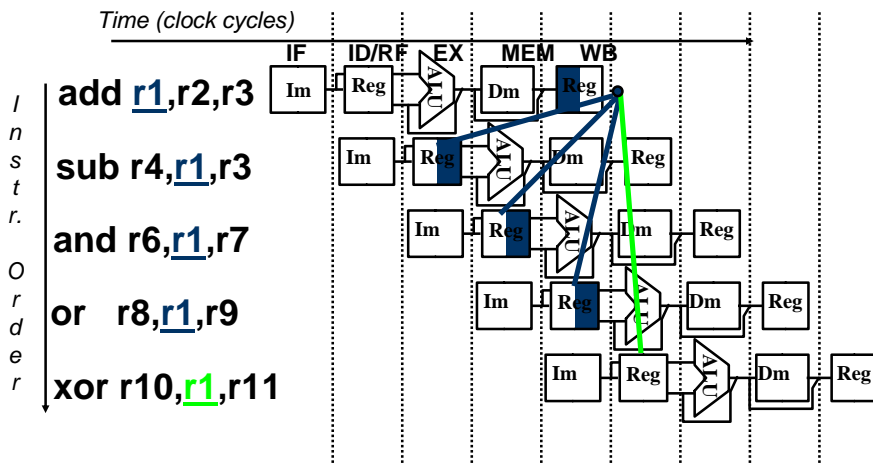
Data Hazard on r1

```

add r1, r2, r3
sub r4, r1, r3
and r6, r1, r7
or r8, r1, r9
xor r10, r1, r11
    
```

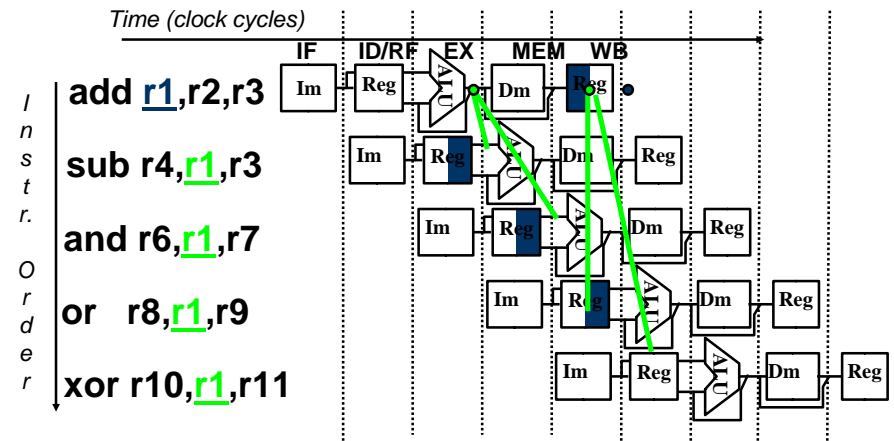
Data Hazard on r1: (cont.)

- Dependencies backwards in time are hazards



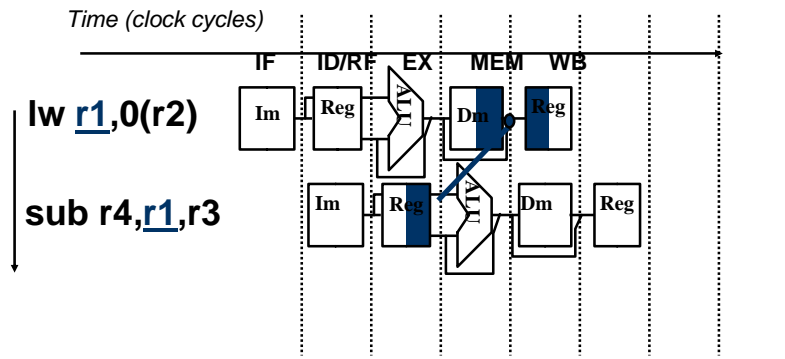
Data Hazard Solution:

- "Forward" result from one stage to another



- "or" OK if define read/write properly

Forwarding (or Bypassing): What about Loads



- Dependencies backwards in time are hazards
- Can't solve with forwarding:
- Must delay/stall instruction dependent on loads