

# CS 316: Pipelined Architectures

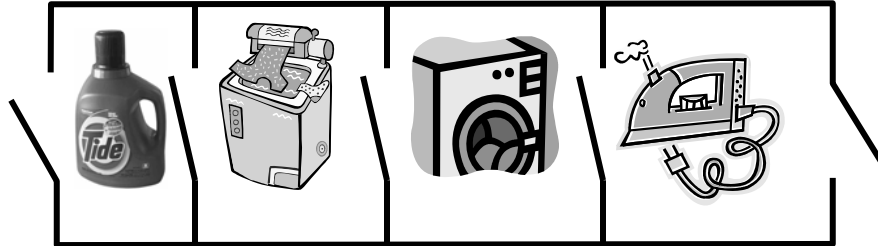
**Kavita Bala**  
**Fall 2007**  
Computer Science  
Cornell University

## Announcements

---

- PA 3
  - Lectures on it this Tue/Thu/Fri
  - Due on the Friday after Fall break
- Don't wait till the last minute
  - We are happy to help
  - Hazards will take time

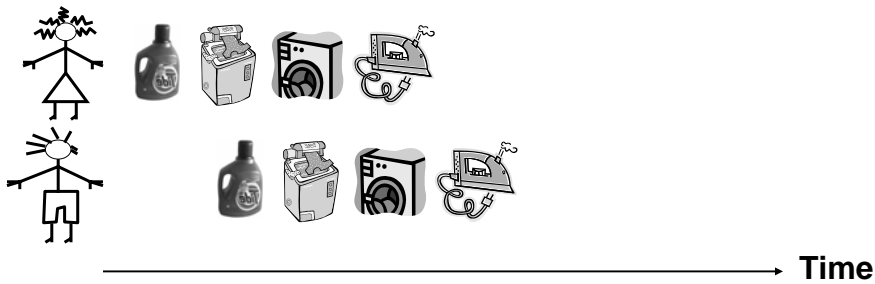
## Laundry Room Design #2



- The room is partitioned into stages
- One person owns a stage at a time, the room can hold up to four people simultaneously

Kavita Bala, Computer Science, Cornell University

## Laundry Room Design #2

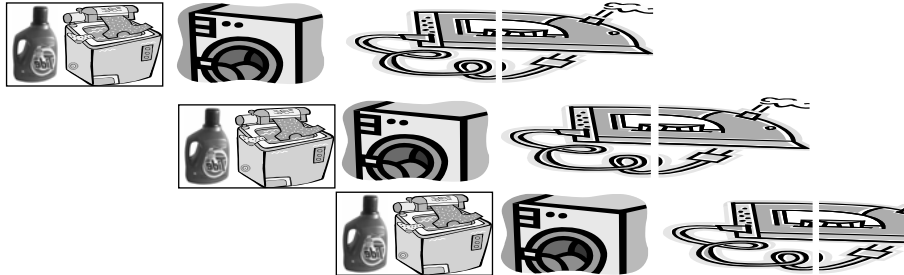


- Elapsed Time for Alice: 4
- Elapsed Time for Bob: 4
- Elapsed Time for both: 5!!!

Kavita Bala, Computer Science, Cornell University

## Scenario with varying stage times

---



- Latency: ?
- Throughput: Batch every 45 minutes

Kavita Bala, Computer Science, Cornell University

## Pipelining

---

- Principle: Latencies can be masked by running operations in parallel
- Need to identify “stages”
- Need mechanisms for isolating the operations
- Need mechanisms for handling dependencies between stages
- Let’s apply this principle to processor design...

Kavita Bala, Computer Science, Cornell University

## Basic Pipelining

---

### Five stage “RISC” load-store architecture

1. Instruction fetch (IF)
  - get instruction from memory
2. Instruction Decode (ID)
  - translate opcode into control signals and read regs
3. Execute (EX)
  - perform ALU operation
4. Memory (MEM)
  - Access memory if load/store
5. Writeback (WB)
  - update register file

Following slides thanks to Sally McKee

Kavita Bala, Computer Science, Cornell University

## Pipelined Implementation

---

- Break the execution of the instruction into cycles (five, in this case)
- Design a separate stage for the execution performed during each cycle
- Build pipeline registers (latches) to communicate between the stages

Slides thanks to Sally McKee

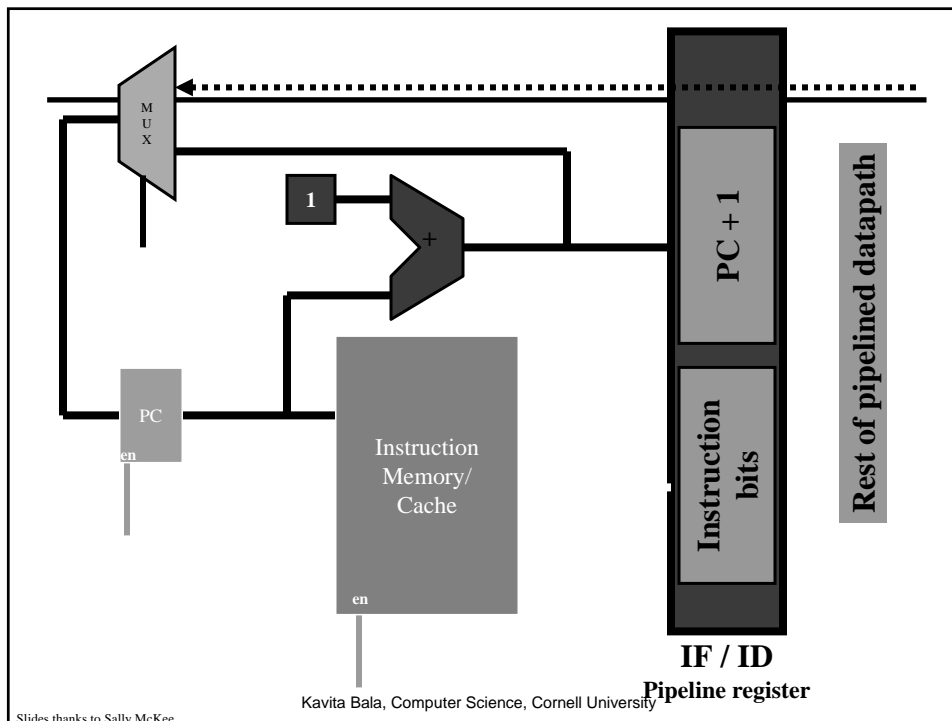
Kavita Bala, Computer Science, Cornell University

## Stage 1: Fetch and Decode

- Design a datapath that can fetch an instruction from memory **every** cycle
  - Use PC to index memory to read instruction
  - Increment the PC (assume no branches for now)
- Write everything needed to complete execution to the pipeline register (IF/ID)
  - The next stage will read this pipeline register
  - Note that pipeline register must be edge triggered

Kavita Bala, Computer Science, Cornell University

Slides thanks to Sally McKee



Kavita Bala, Computer Science, Cornell University

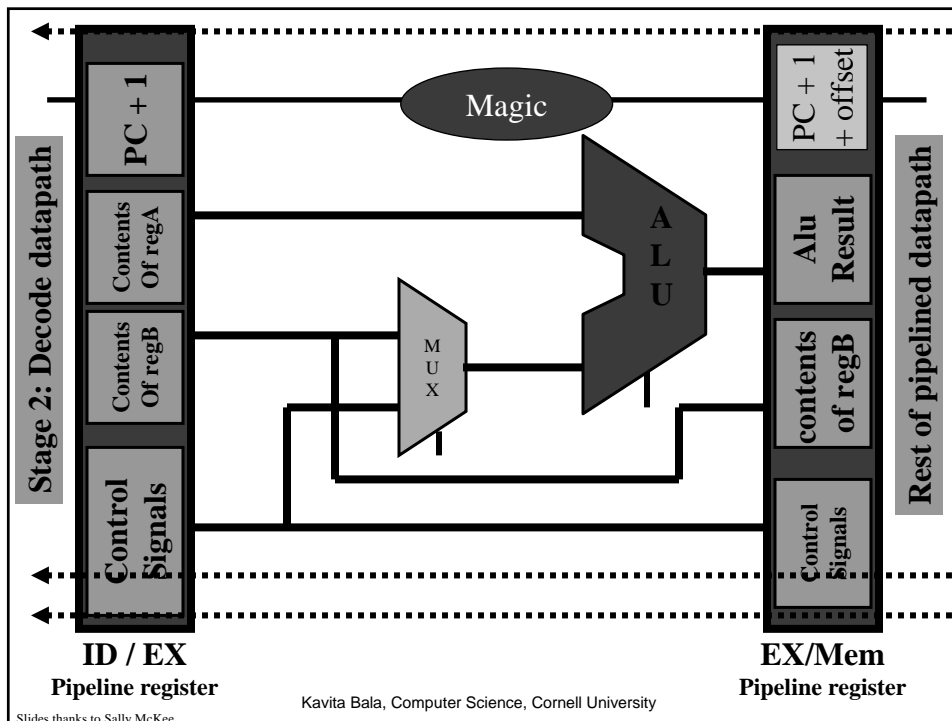
Slides thanks to Sally McKee



## Stage 3: Execute

- Design a datapath that performs the proper ALU operation for the instruction specified and values present in the ID/EX pipeline register
  - The inputs are the contents of regA and either the contents of regB or the offset field in the instruction
  - Also, calculate  $PC+1+offset$ , in case this is a branch
- Write everything needed to complete execution to the pipeline register (EX/Mem)
  - ALU result, contents of regB and  $PC+1+offset$
  - Instruction bits for opcode and destReg specifiers

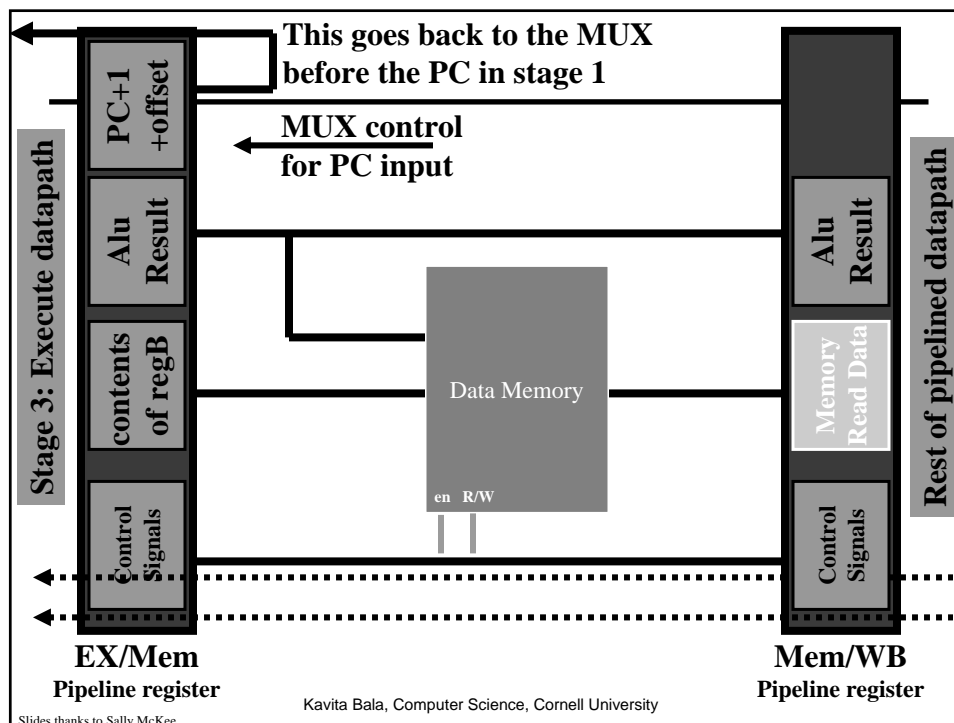
Kavita Bala, Computer Science, Cornell University



## Stage 4: Memory Operation

- Design a datapath that performs the proper memory operation for the instruction specified and values present in the EX/Mem pipeline register
  - ALU result contains address for **ld** and **st** instructions
  - Opcode bits control memory R/W and enable signals
- Write everything needed to complete execution to the pipeline register (Mem/WB)
  - ALU result and MemData
  - Instruction bits for opcode and destReg specifiers

Kavita Bala, Computer Science, Cornell University



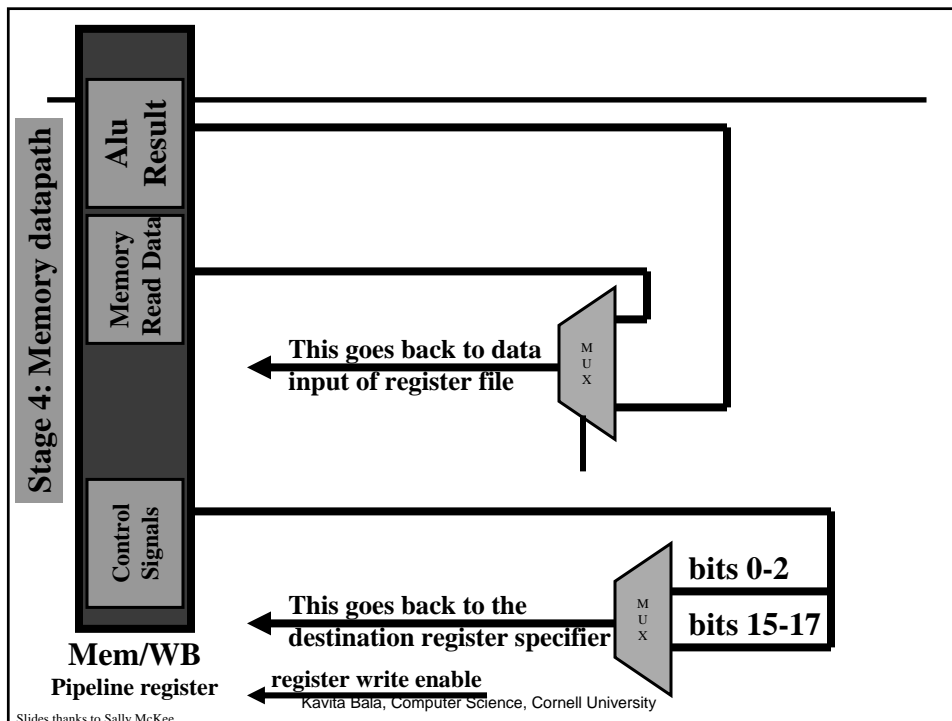


## Stage 5: Write Back

- Design a datapath that completes the execution of this instruction, writing to the register file if required
  - Write MemData to destReg for ld instruction
  - Write ALU result to destReg for arithmetic/logic instructions
  - Opcode bits also control register write enable signal

Kavita Bala, Computer Science, Cornell University

Slides thanks to Sally McKee



Slides thanks to Sally McKee

## Sample Code (Simple)

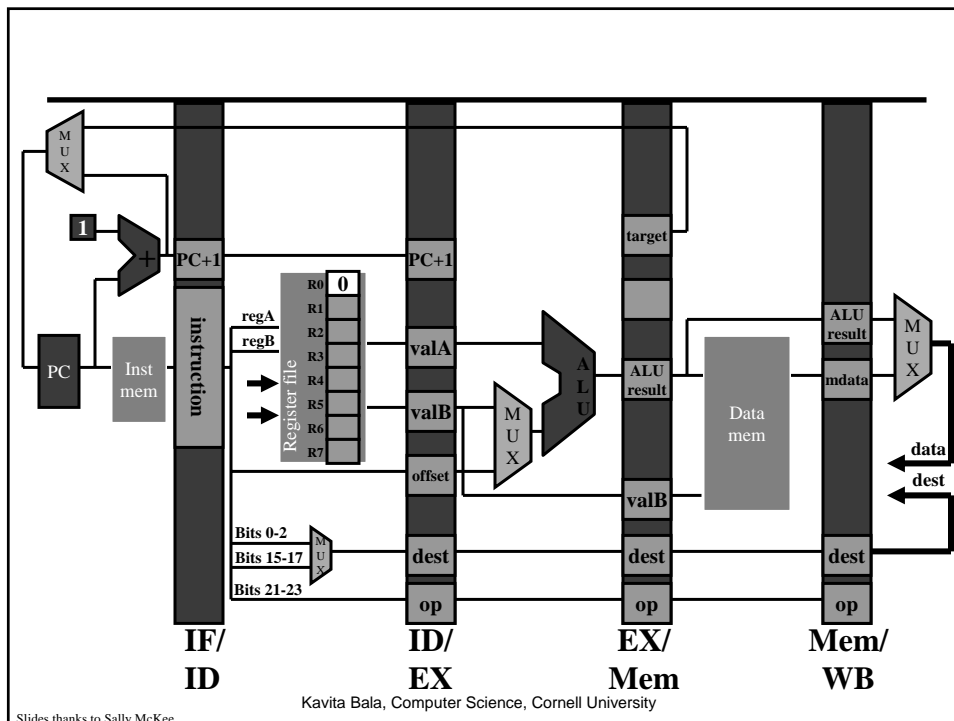
- Assume eight-register machine
- Run the following code on a pipelined datapath

```

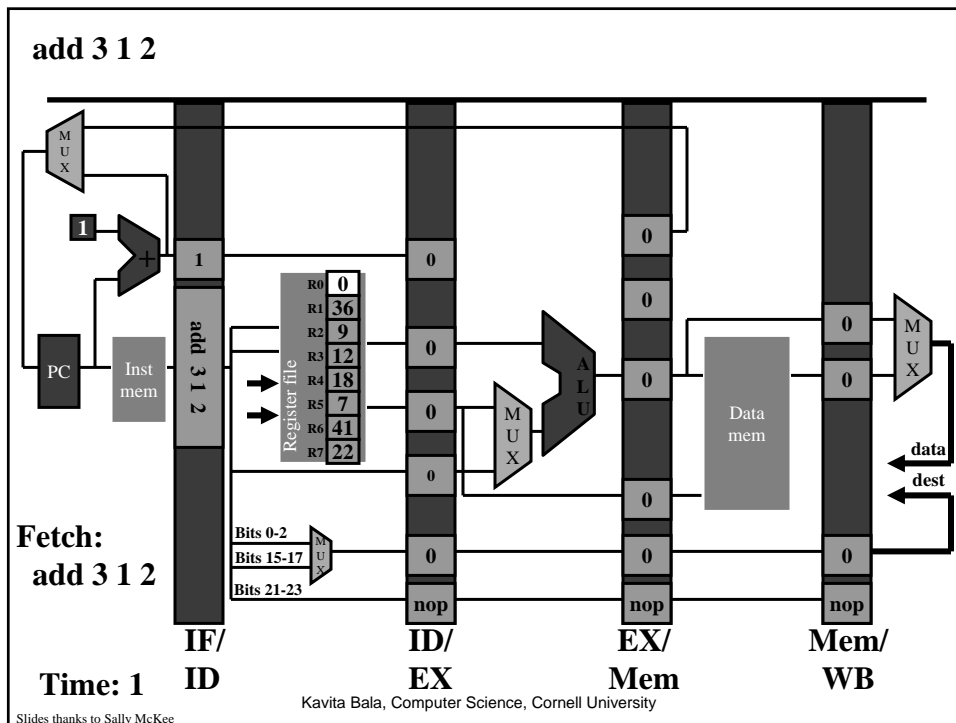
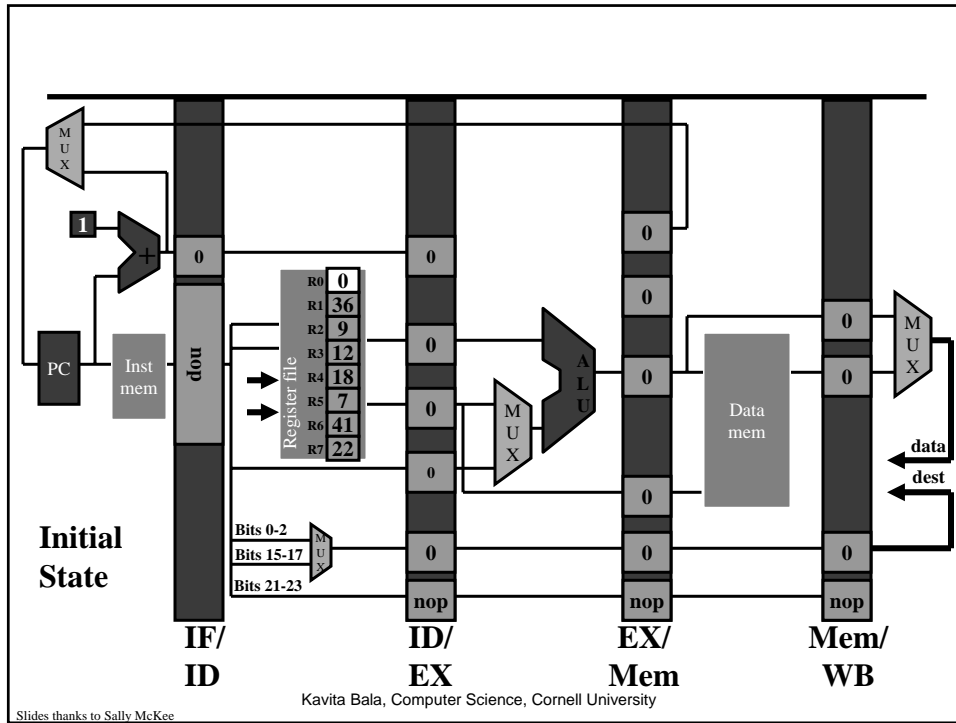
add   3 1 2 ; reg 3 = reg 1 + reg 2
nand  6 4 5 ; reg 6 = ~(reg 4 & reg 5)
lw    4 20(2) ; reg 4 = Mem[reg2+20]
add   5 2 5 ; reg 5 = reg 2 + reg 5
sw    7 12(3) ; Mem[reg3+12] = reg 7
    
```

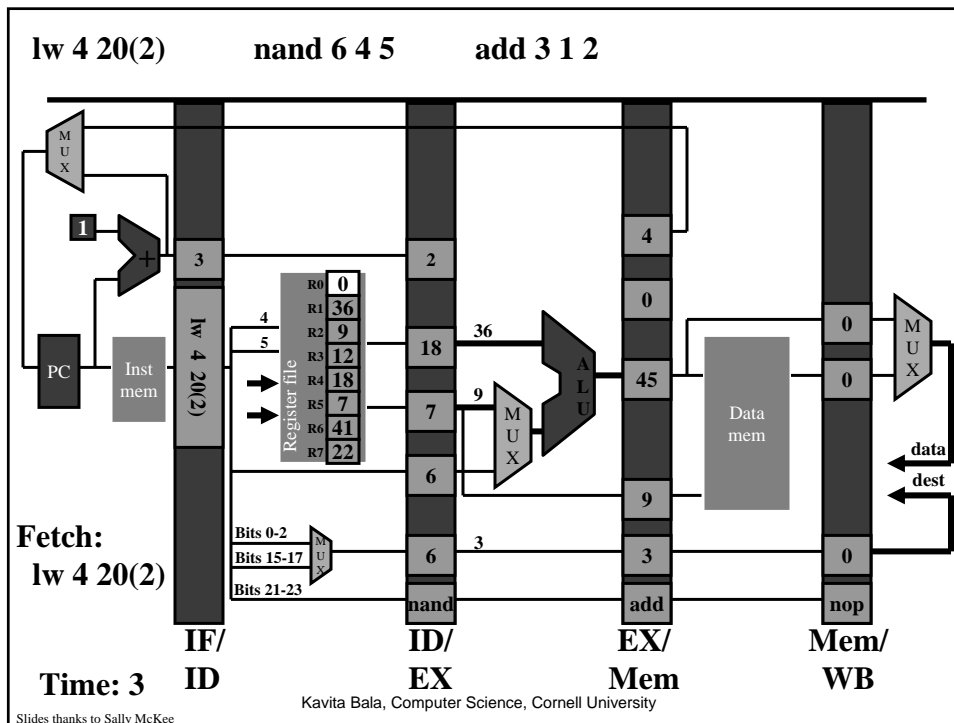
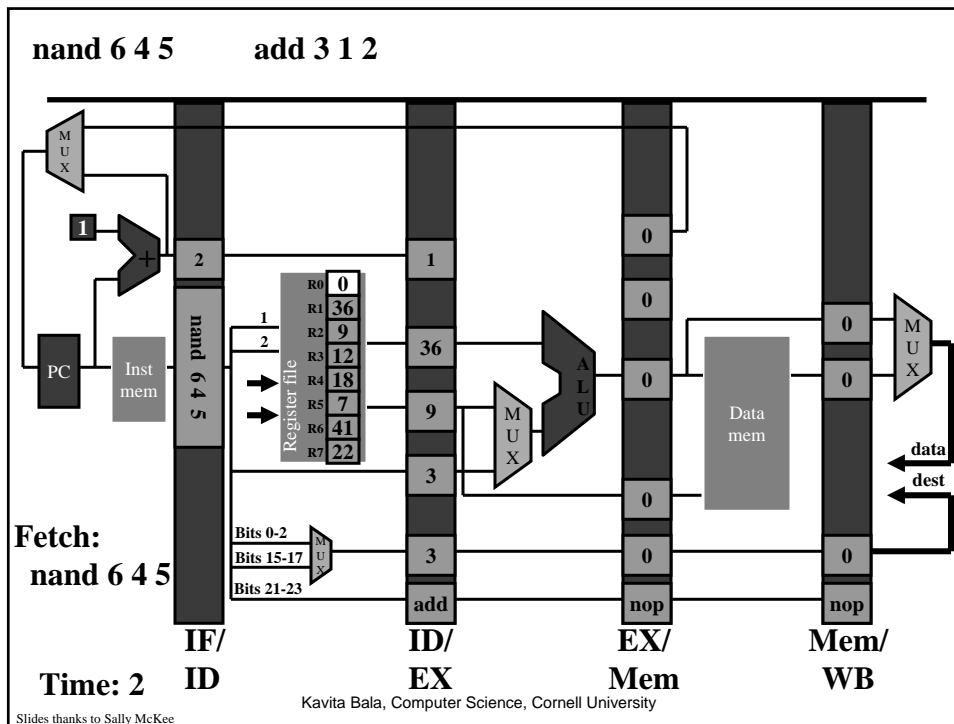
Kavita Bala, Computer Science, Cornell University

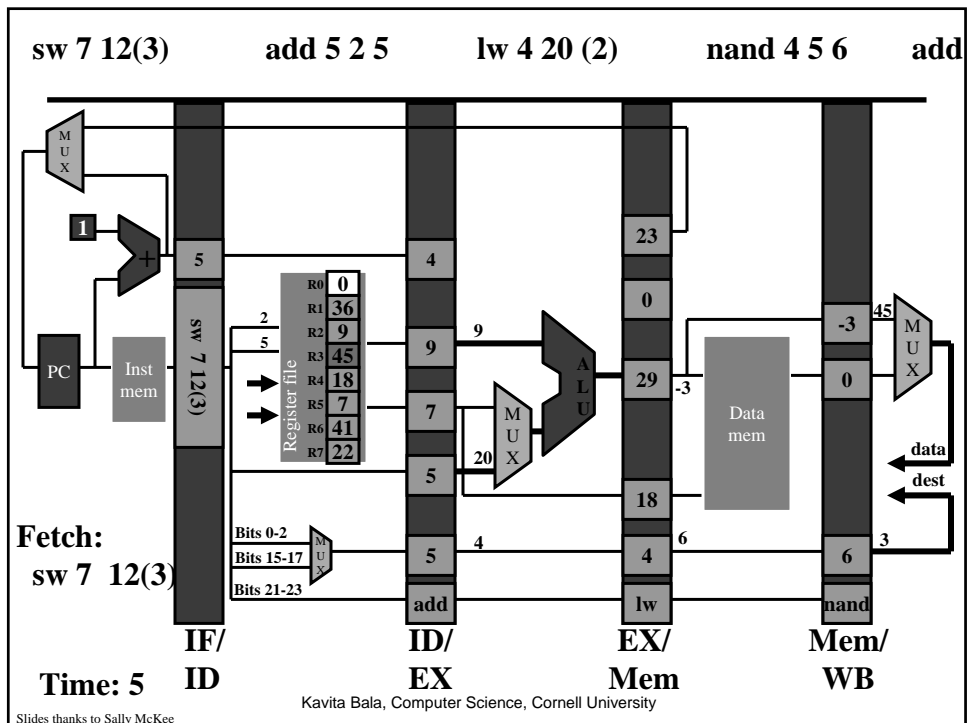
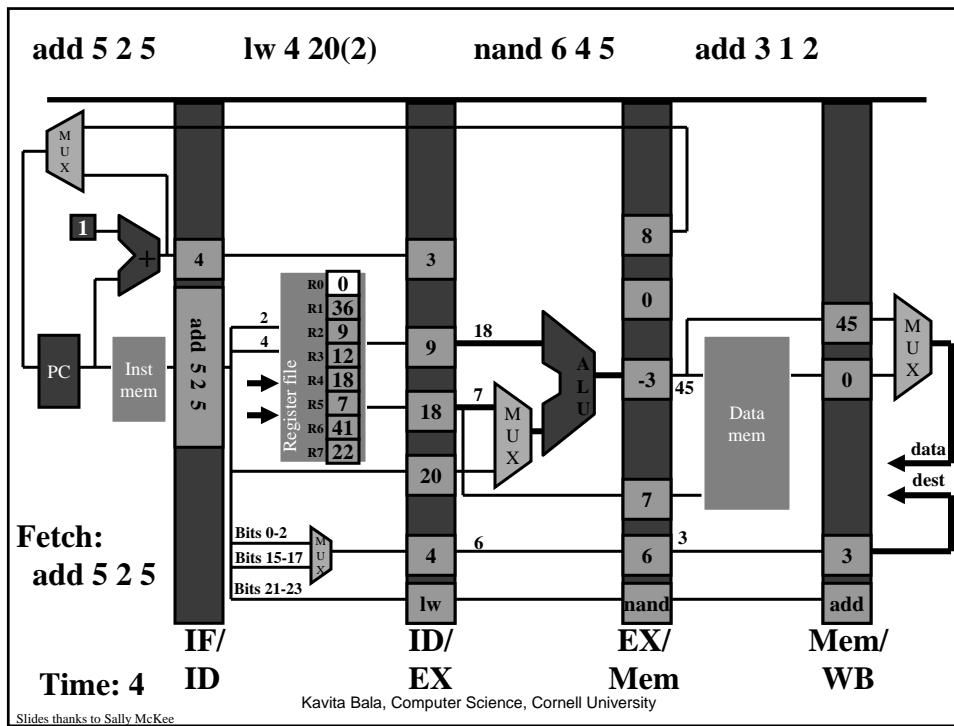
Slides thanks to Sally McKee

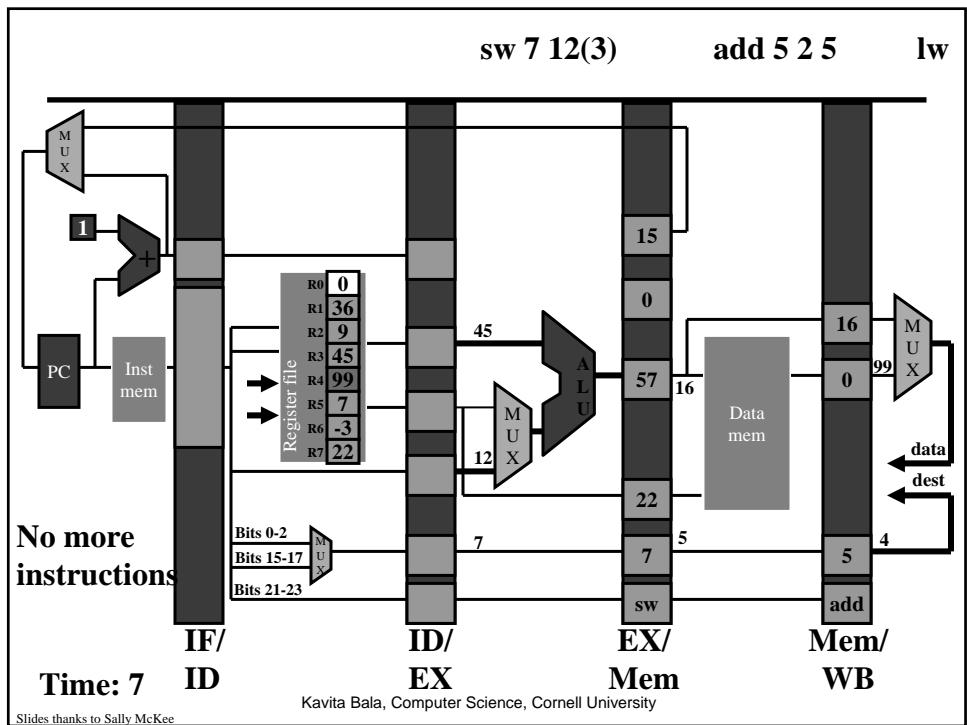
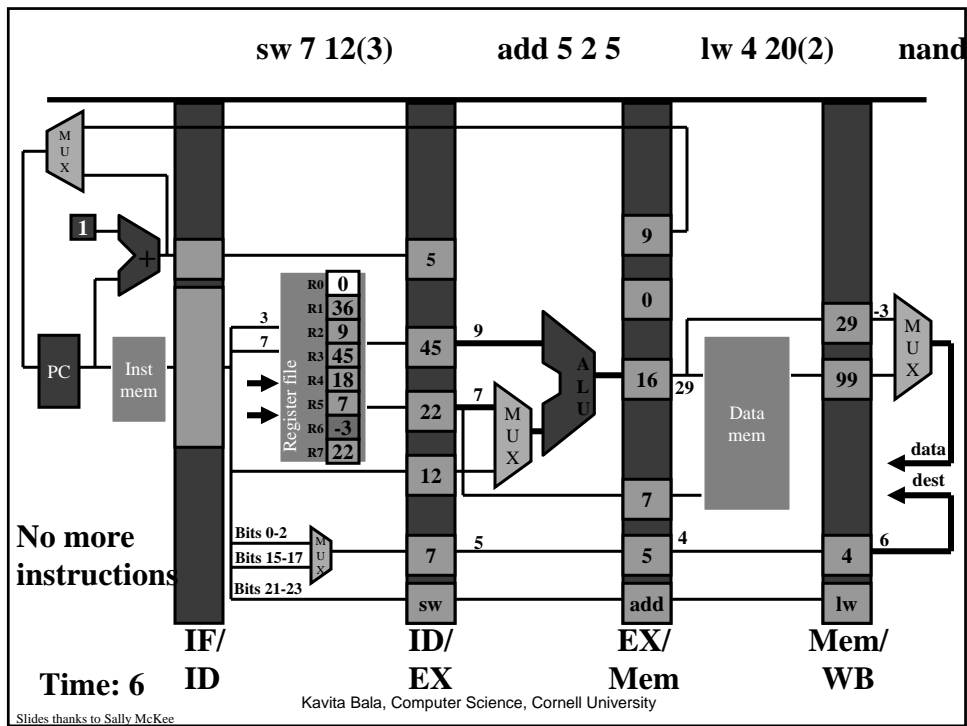


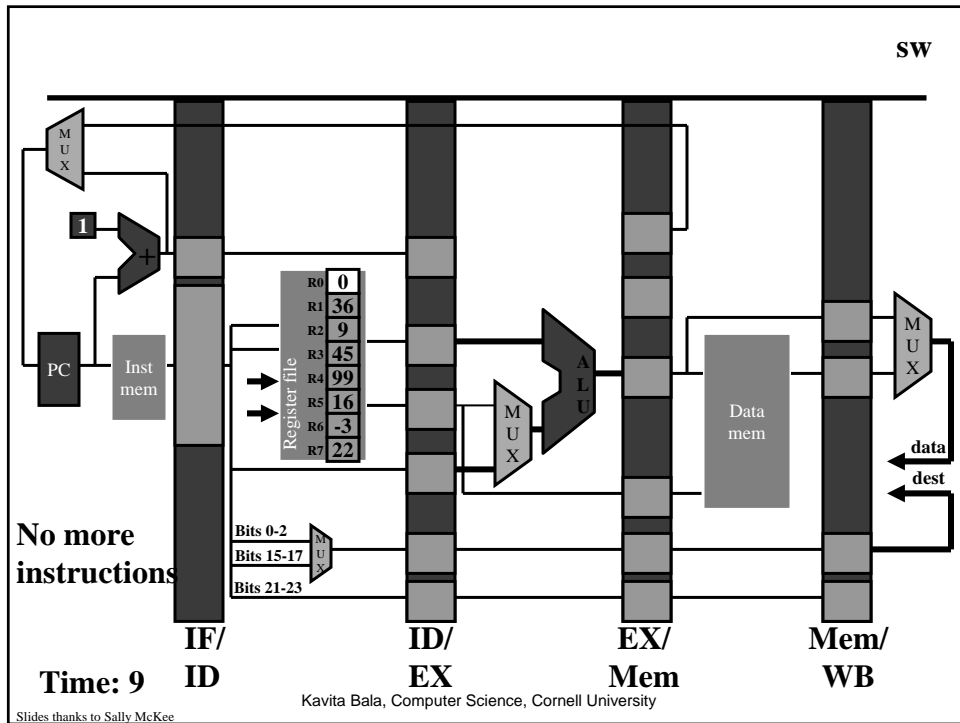
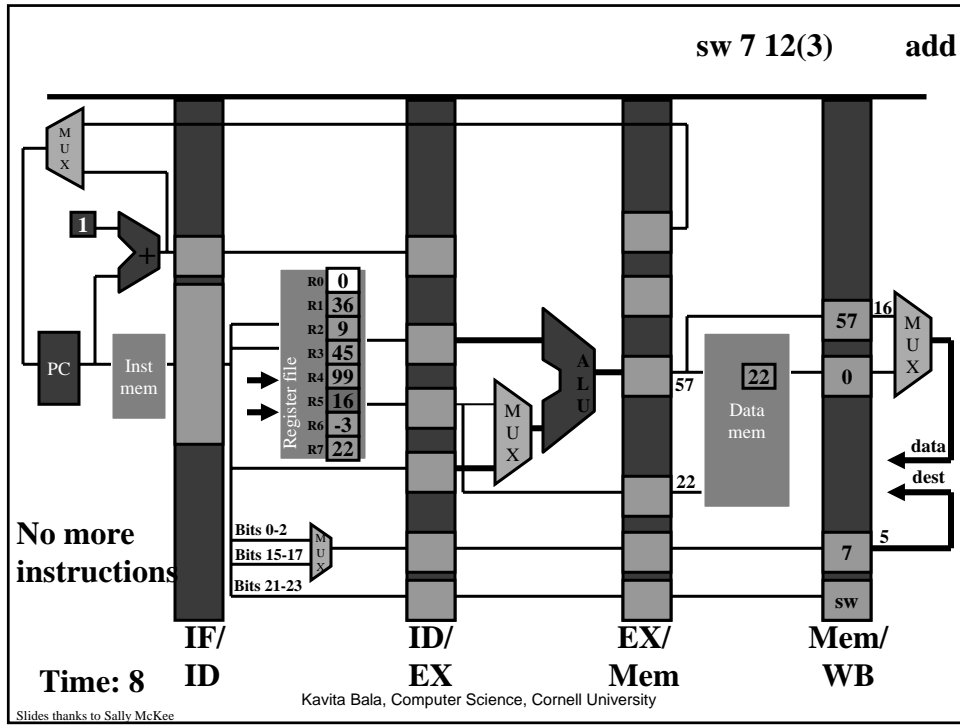
Slides thanks to Sally McKee











## Time Graphs

	Time: 1	2	3	4	5	6	7	8	9
add	fetch	decode	execute	memory	writeback				
nand		fetch	decode	execute	memory	writeback			
lw			fetch	decode	execute	memory	writeback		
add				fetch	decode	execute	memory	writeback	
sw					fetch	decode	execute	memory	writeback

Kavita Bala, Computer Science, Cornell University

## Pipelining Recap

- Powerful technique for masking latencies
  - Logically, instructions execute one at a time
  - Physically, instructions execute in parallel
    - Instruction level parallelism
- Decouples the processor model from the implementation
  - Interface vs. implementation
- BUT dependencies between instructions complicate the implementation

Kavita Bala, Computer Science, Cornell University



## What can go wrong?

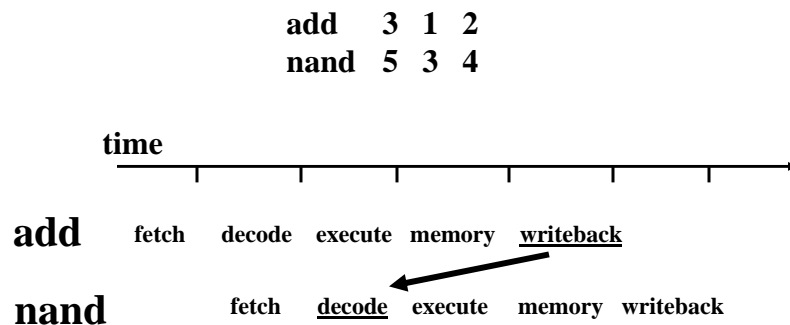
---

- Structural hazards
  - Two instructions in the pipeline try to simultaneously access the same resource
- Data hazards
  - A required operand is not ready
  - Usually because a previous instruction in the pipeline has not committed it to the register file yet
- Control hazards
  - The next instruction to fetch cannot be determined
  - Usually because a jump or branch instruction has not determined the next PC yet

Kavita Bala, Computer Science, Cornell University

## Data Hazards

---



**If not careful, you read the wrong value of R3**

Kavita Bala, Computer Science, Cornell University

## Handling Data Hazards

---

- Avoidance
  - Make sure there are no hazards in the code
  - Some compilers have done this (Multiflow Trace)
- Detect and Stall
  - If hazards exist, stall the processor until they go away
  - Safe, but not great for performance
- Detect and Forward
  - If hazards exist, fix up the pipeline to get the correct value (if possible)
  - Most common solution for high performance

Kavita Bala, Computer Science, Cornell University

## Handling Data Hazards I

---

- Just Avoid the Problems
- Compiler problem
- Beyond scope of this class
- BUT:
  - Know it's an option
  - Know it's doable

Kavita Bala, Computer Science, Cornell University

## Handling Data Hazards II

---

- Detect and Stall
- Detection:
  - Compare regA with previous DestRegs
  - Compare regB with previous DestRegs
- Stall:
  - Insert a bubble in pipeline
  - Keep current instructions in fetch and decode
  - Pass a nop to execute

Kavita Bala, Computer Science, Cornell University

## Handling Data Hazards III:

---

- Detect: same as detect and stall
- Forward:
  - New bypass datapaths route computed data to where it is needed
  - New MUX and control to pick the right data
- Beware: Stalling may still be required even in the presence of forwarding

Kavita Bala, Computer Science, Cornell University