EEL-4713C
Computer Architecture
Designing a Pipelined Processor

Outline

° Introduction to the pipelined datapath/control
  • Key concepts and examples
  • Beginning with multi-cycle datapath

° Reading:
  • 4.5-4.9

Recap: Multiple Cycle Implementation

° Break the instruction into smaller steps

° Execute each step (instead of the entire instruction) in one cycle
  - Strive to keep steps with similar length

° Instructions take multiple cycles to execute
  • Only one instruction uses the datapath at any given cycle

° Pipelining
  • Goal: improve performance by having *multiple instructions in the datapath at any given cycle*

Pipelining is Natural!

° Laundry Example

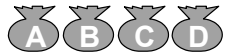
° Alice, Bob, Charlie, and Debbie each have one load of clothes to wash, dry, and fold

° Washer takes 30 minutes

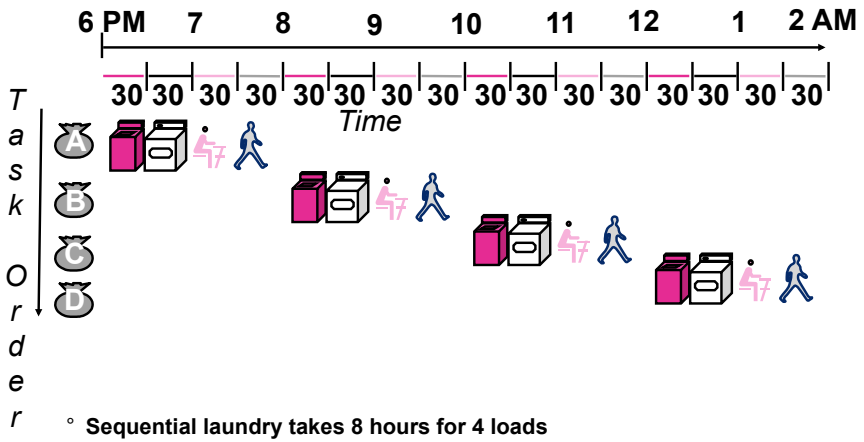
° Drier takes 30 minutes

° "Folder" takes 30 minutes

° "Stasher" takes 30 minutes to put clothes into drawers

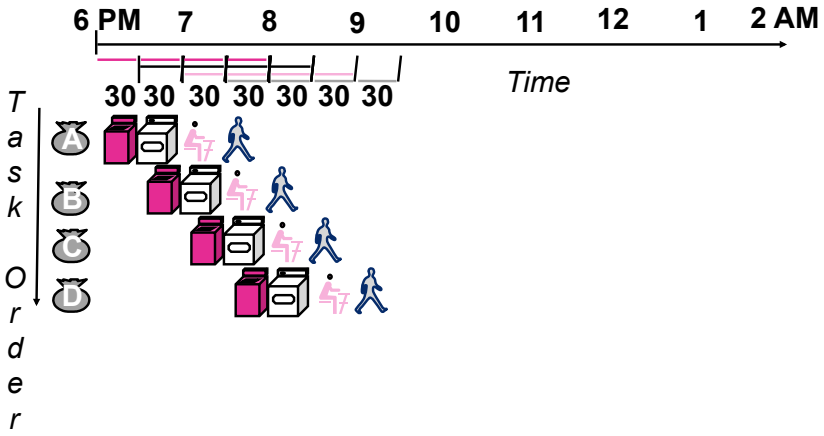


Sequential Laundry



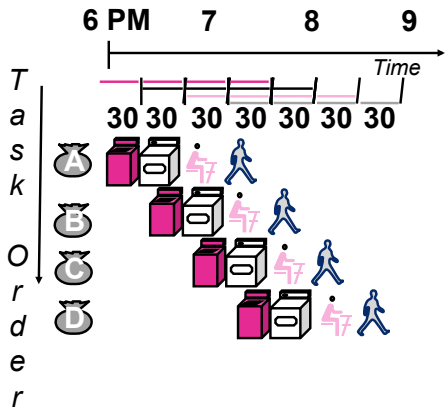
° Sequential laundry takes 8 hours for 4 loads

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 of pipe stages

° Also:
- 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
- Dependences - stalls

Why Pipeline?

° Suppose we execute 100 instructions

° Single Cycle Machine, 220MHz
- 4.5 ns/cycle x 1 CPI x 100 inst = 450 ns

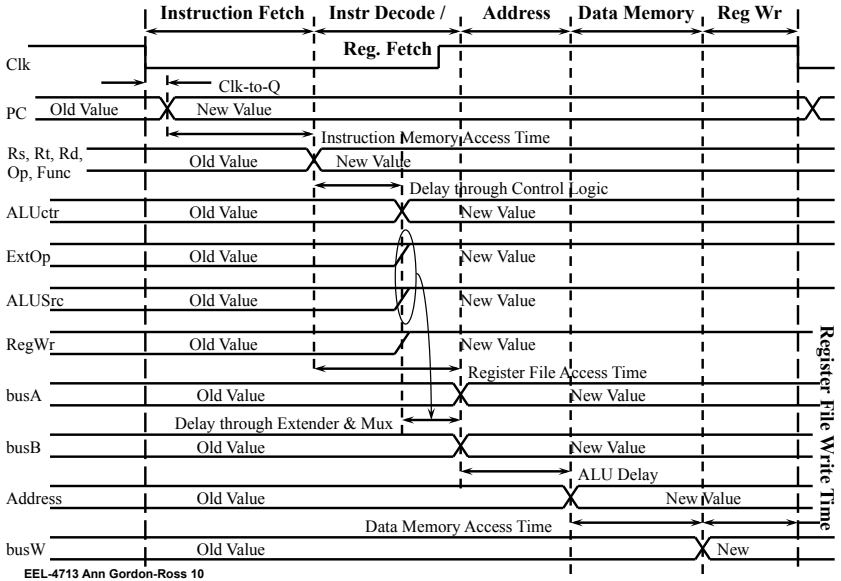
° Multicycle Machine, 1GHz, average CPI 4.6
- 1 ns/cycle x 4.6 CPI x 100 inst = 460 ns

° Ideal pipelined machine, 1GHz
- 1 ns/cycle x (1 CPI x 100 inst + 4 cycle drain) = 104 ns

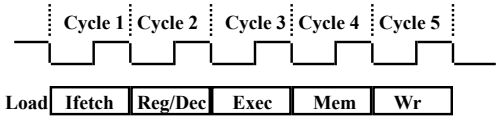
Datapath: from multicycle to pipelined

- Key approach:
  - Begin by considering each cycle a pipeline stage
  - Add registers between consecutive stages to store data and control bits flowing from one stage to the next
  - Add logic to deal with "hazards"
    - Ensure functional units are not used by more than one stage at the same time
    - Ensure producer/consumer dependences are not violated
- Example: load instruction

Timing Diagram of a Load Instruction

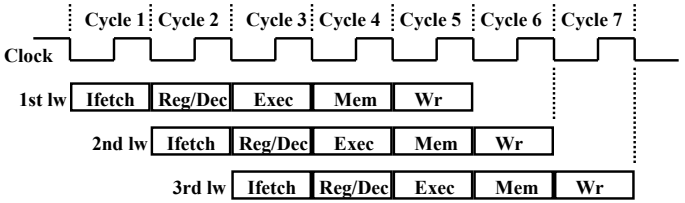


The Five Stages of Load



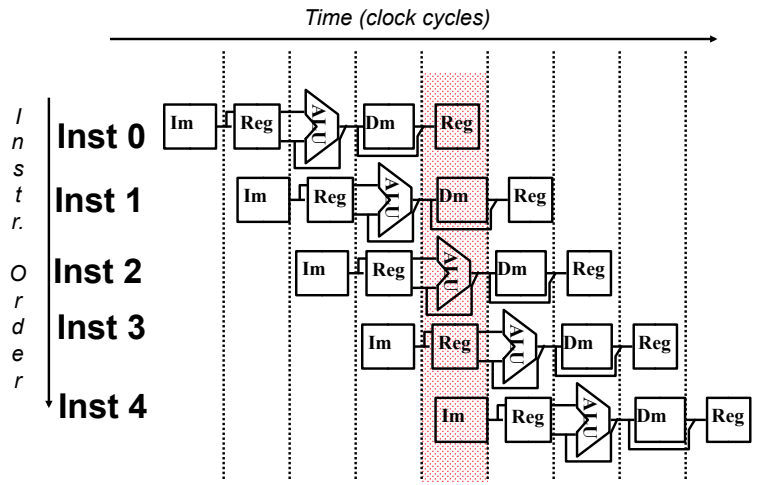
- Ifetch: Instruction Fetch
  - Fetch the instruction from the Instruction Memory
- Reg/Dec: Registers Fetch and Instruction Decode
- Exec: Calculate the memory address
- Mem: Read the data from the Data Memory
- Wr: Write the data back to the register file

Pipelining the Load Instruction

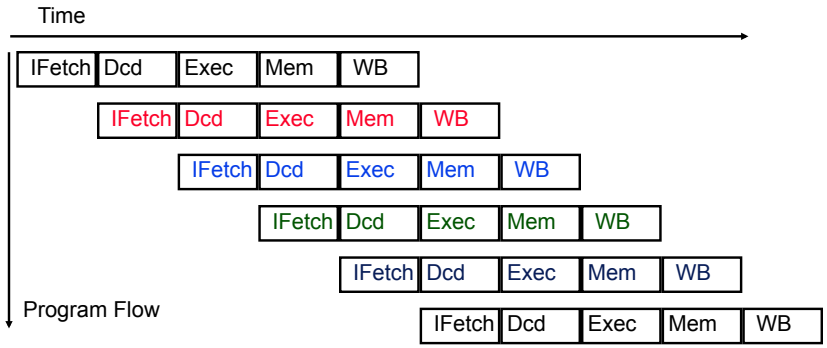


- The five independent functional units in the pipeline datapath are:
  - Instruction Memory for the Ifetch stage
  - Register file: read ports (bus A and busB) for the Reg/Dec stage
  - ALU for the Exec stage
  - Data Memory for the Mem stage
  - Register File's Write port (bus W) for the Wr stage
- One instruction enters the pipeline every cycle
  - One instruction comes out of the pipeline (complete) every cycle
  - The effective cycles per Instruction (CPI) is 1

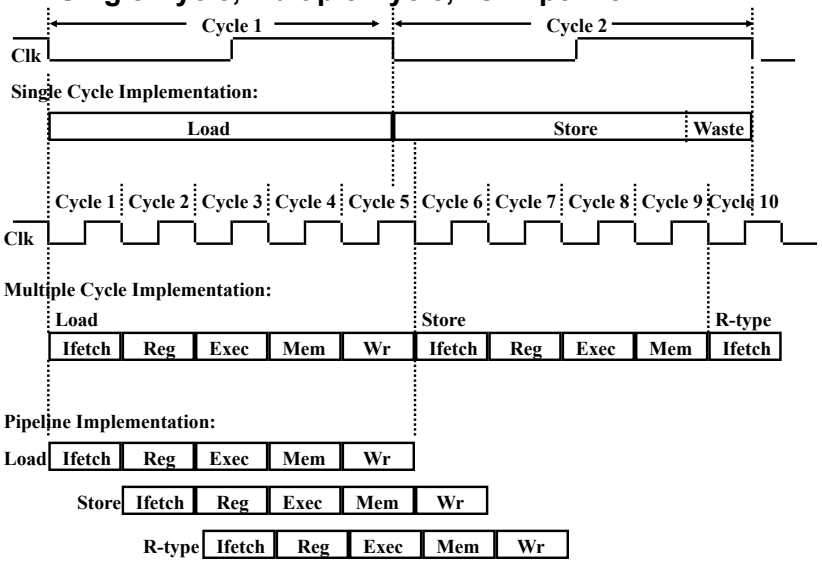
Why Pipeline? Because the resources are there!



Conventional Pipelined Execution Representation



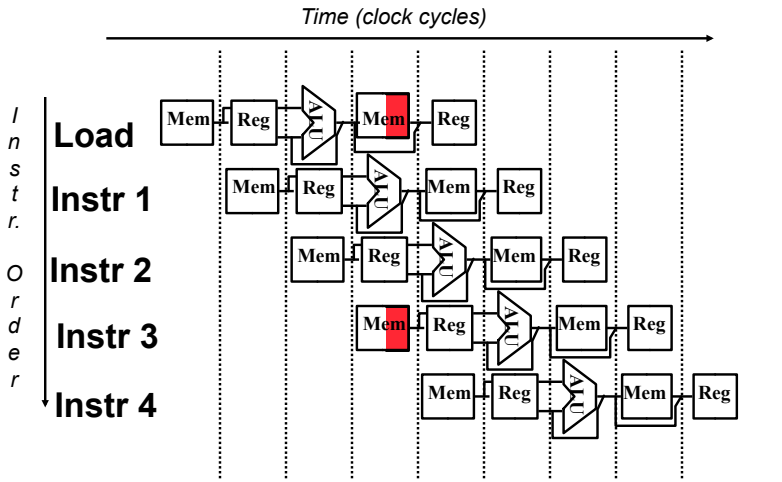
Single Cycle, Multiple Cycle, vs. Pipeline



*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)
  - 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
  - control hazards: attempt to make a decision before condition is evaluated
    - 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

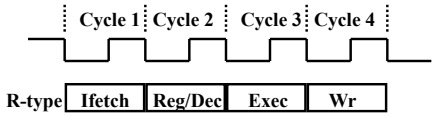


Detection is easy in this case! (right half highlight means read, left half write)

Structural Hazards limit performance

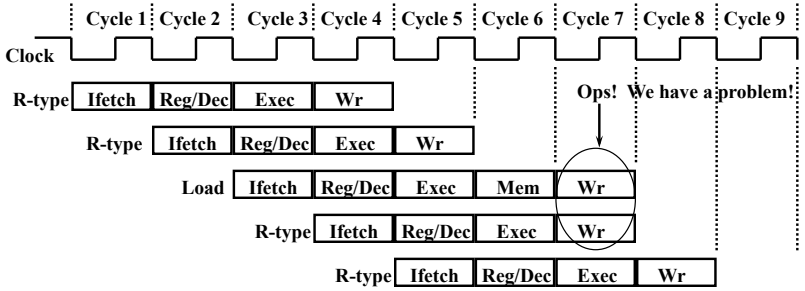
- Example: if average 1.3 memory accesses per instruction and only one memory access per cycle then
  - average CPI >= 1.3
  - otherwise resource is more than 100% utilized
  - More on Hazards later

The Four Stages of R-type



- Ifetch: Instruction Fetch
  - Fetch the instruction from the Instruction Memory
- Reg/Dec: Registers Fetch and Instruction Decode
- Exec: ALU operates on the two register operands
- Wr: Write the ALU output back to the register file

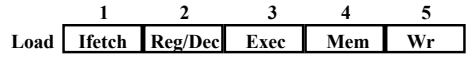
Pipelining the R-type and Load Instruction



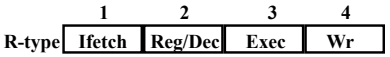
- We have a problem:
  - Two instructions try to write to the register file at the same time!

Important Observation

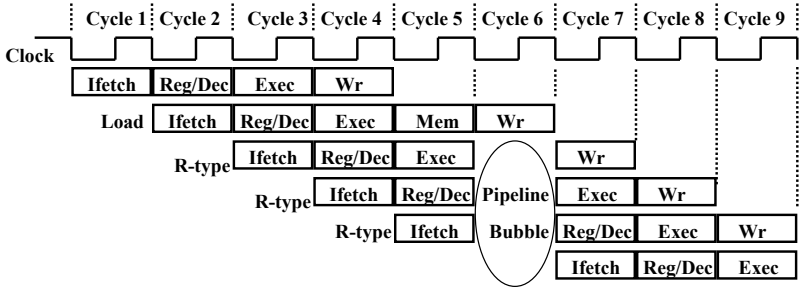
- Each functional unit can only be used once per instruction
- Each functional unit must be used at the same stage for all instructions:
  - Load uses Register File's Write Port during its 5th stage



  - R-type uses Register File's Write Port during its 4th stage



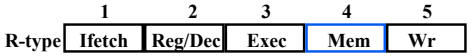
Solution 1: Insert "Bubble" into the Pipeline

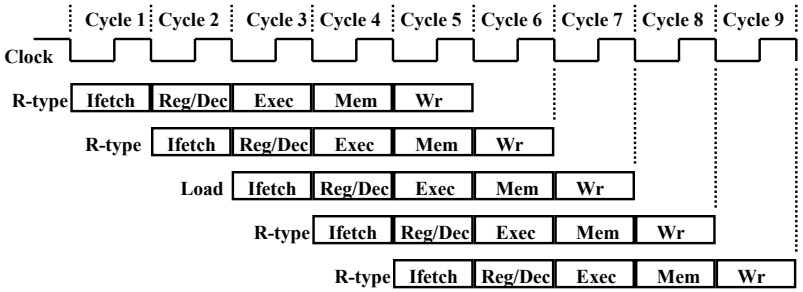


- Insert a "bubble" into the pipeline to prevent 2 writes at the same cycle
  - The control logic can be complex

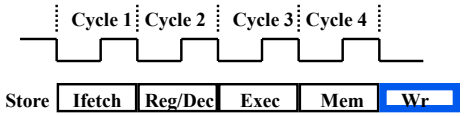
Solution 2: Delay R-type's Write by One Cycle

- Delay R-type's register write by one cycle:
  - Now R-type instructions also use Reg File's write port at Stage 5
  - Mem stage is a NOOP stage: nothing is being done



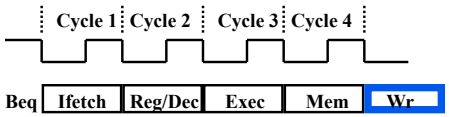


The Four Stages of Store



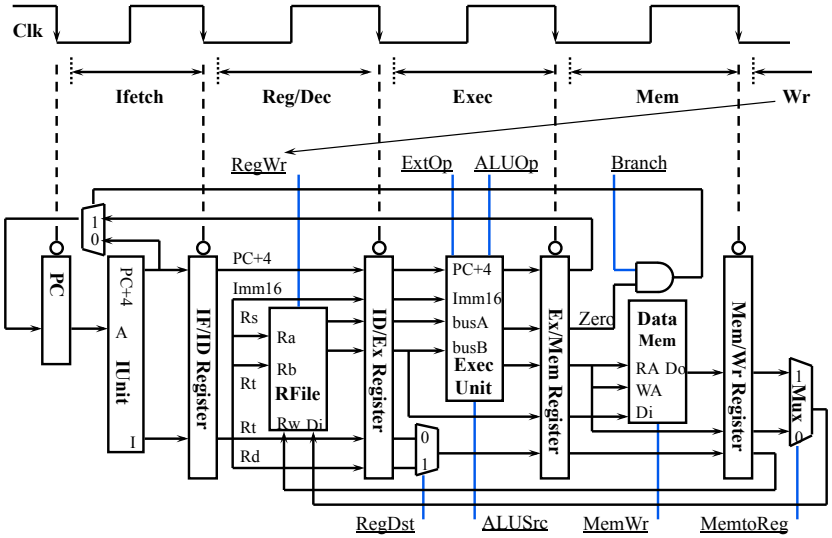
- Ifetch: Instruction Fetch
  - Fetch the instruction from the Instruction Memory
- Reg/Dec: Registers Fetch and Instruction Decode
- Exec: Calculate the memory address
- Mem: Write the data into the Data Memory

The Four Stages of Beq



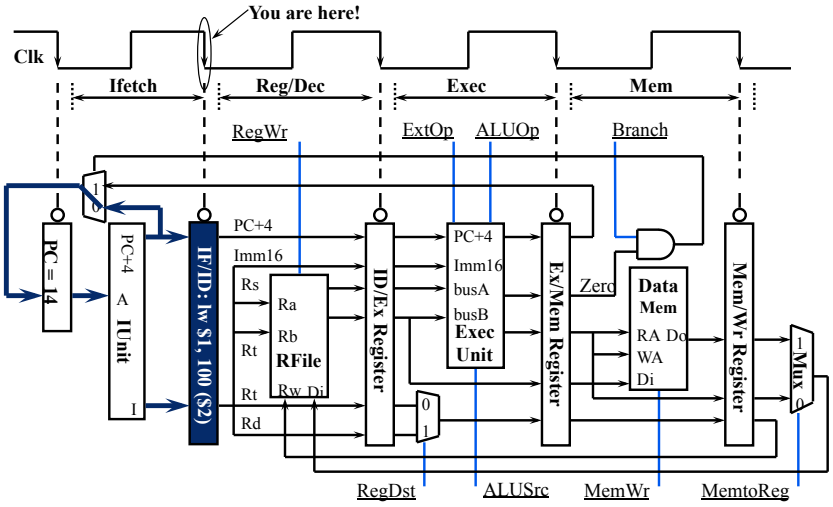
- Ifetch: Instruction Fetch
  - Fetch the instruction from the Instruction Memory
- Reg/Dec: Registers Fetch and Instruction Decode
- Exec: ALU compares the two register operands
  - Adder calculates the branch target address
- Mem: If the registers we compared in the Exec stage are the same,
  - Write the branch target address into the PC

A Pipelined Datapath



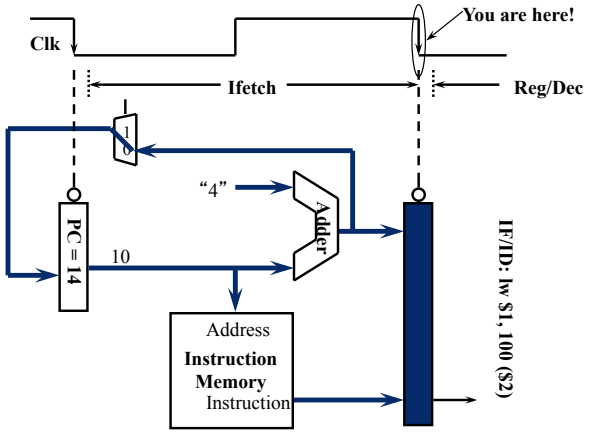
The Instruction Fetch Stage

- Location 10: lw $1, 0x100($2) $1 <- Mem[($2) + 0x100]



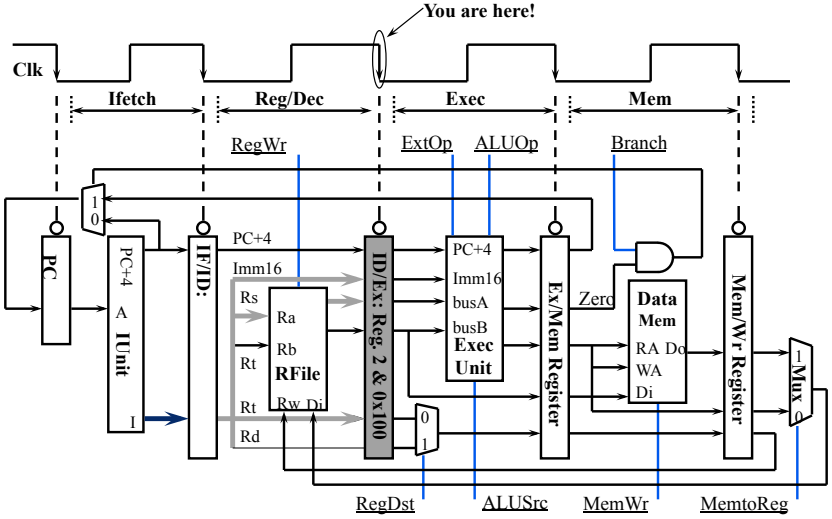
A Detailed View of the Instruction Unit

- Location 10: lw $1, 0x100($2)



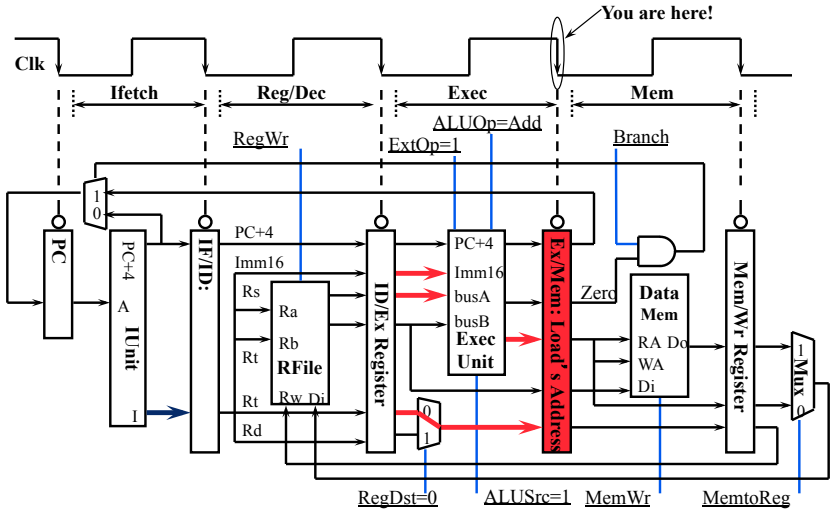
The Decode / Register Fetch Stage

° Location 10: lw $1, 0x100($2) $1 <- Mem[($2) + 0x100]

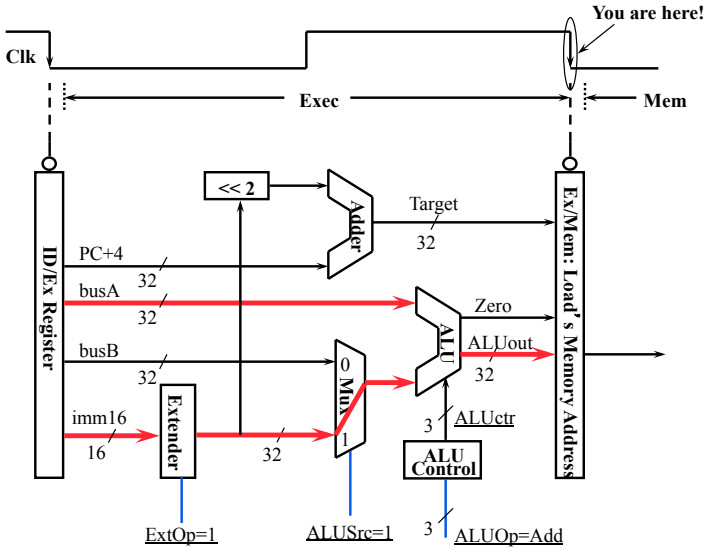


Load's Address Calculation Stage

° Location 10: lw $1, 0x100($2) $1 <- Mem[($2) + 0x100]

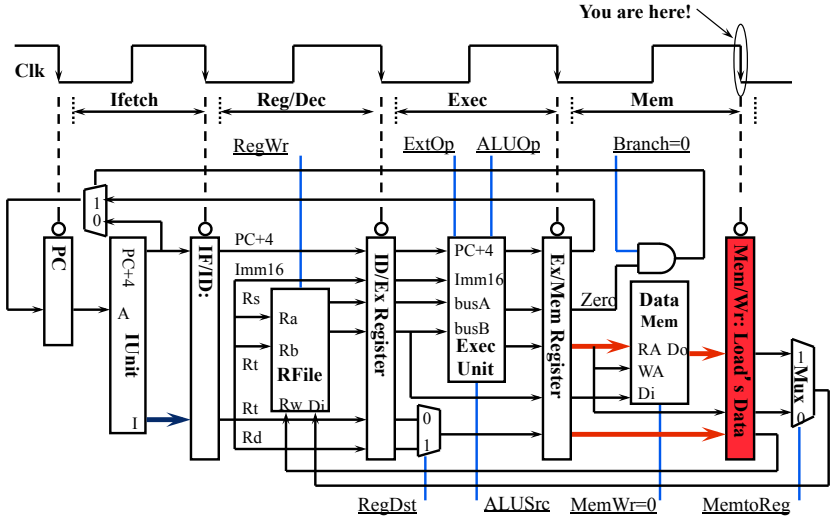


A Detailed View of the Execution Unit



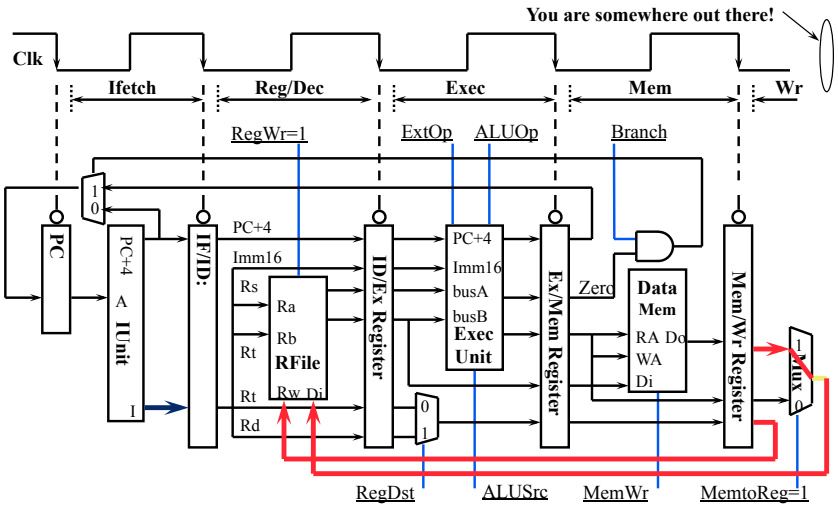
Load's Memory Access Stage

° Location 10: lw $1, 0x100($2) $1 <- Mem[($2) + 0x100]



Load's Write Back Stage

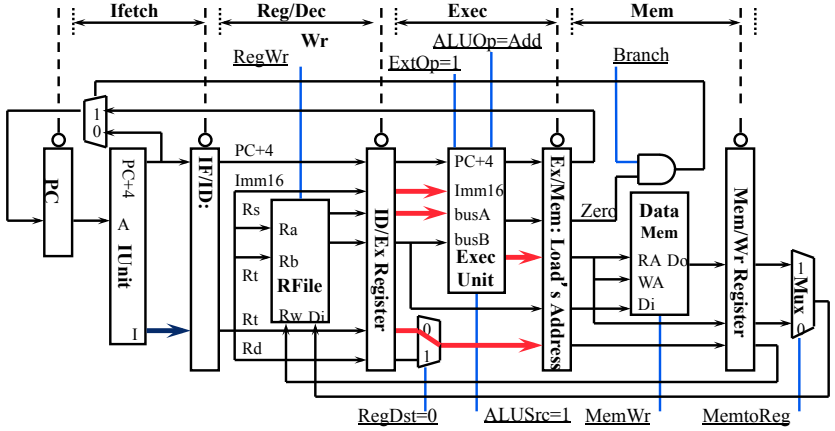
° Location 10: lw $1, 0x100($2) $1 <- Mem[($2) + 0x100]



How About Control Signals?

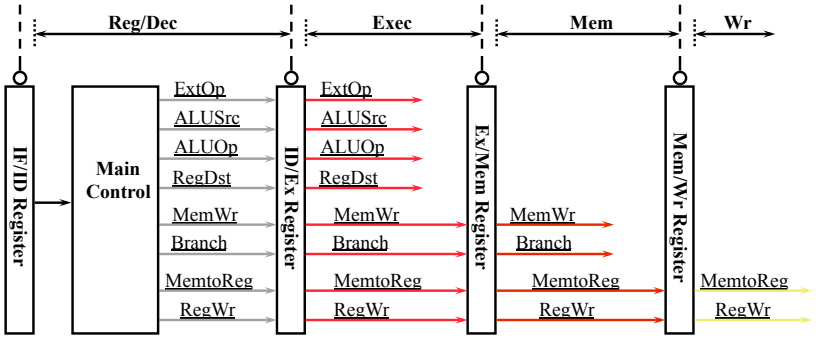
° Key Observation: Control Signals at Stage N = Func (Instr. at Stage N)
- N = Exec, Mem, or Wr
- IF and ID are the same for every instruction

° Example: Controls Signals at Exec Stage = Func(Load's Exec)

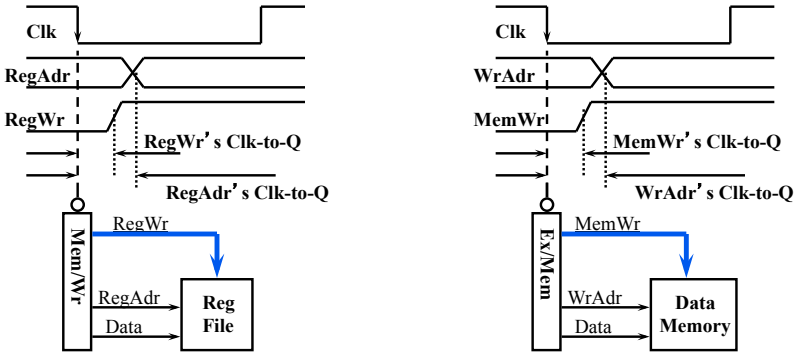


Pipeline Control

° The Main Control generates the control signals during Reg/Dec
- Control signals for Exec (ExtOp, ALUSrc, ...) are used 1 cycle later
- Control signals for Mem (MemWr Branch) are used 2 cycles later
- Control signals for Wr (MemtoReg MemWr) are used 3 cycles later



Beginning of the Wr's Stage: A Real World Problem



° At the beginning of the Wr stage, we have a problem if:
- RegAdr's (Rd or Rt) Clk-to-Q > RegWr's Clk-to-Q

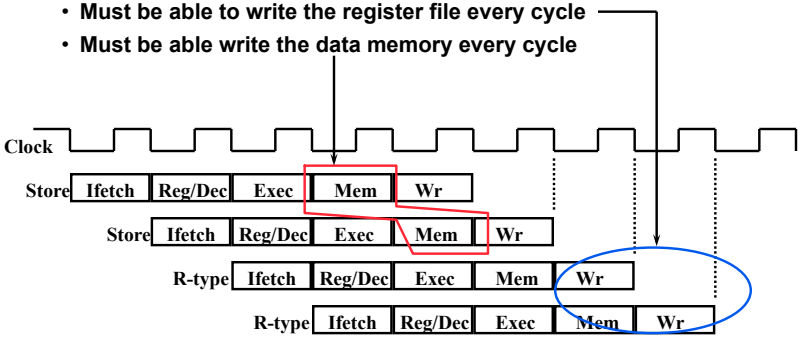
° Similarly, at the beginning of the Mem stage, we have a problem if:
- WrAdr's Clk-to-Q > MemWr's Clk-to-Q

° We have a race condition between Address and Write Enable!
- Could write to incorrect address

The Pipeline Problem

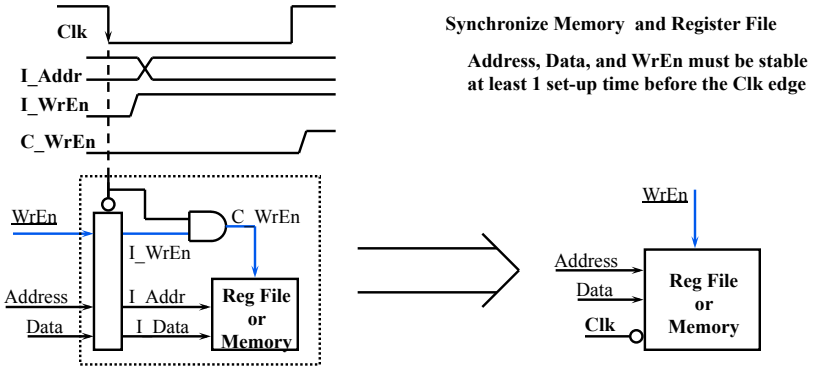
° Multiple Cycle design prevents race condition between Addr and WrEn:
- Make sure Address is stable by the end of Cycle N
- Asserts WrEn during Cycle N + 1

° This approach can NOT be used in the pipeline design because:
- Must be able to write the register file every cycle
- Must be able write the data memory every cycle

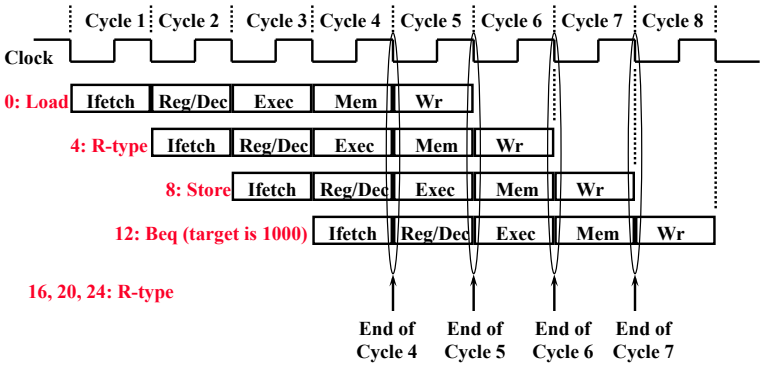


Synchronize Register File & Synchronize Memory

° Solution: And the Write Enable signal with the Clock
- Must consult circuit expert to ensure no timing violation:
  - Example: Clock High Time > Write Access Delay



*A More Extensive Pipelining Example



° End of Cycle 4: Load's Mem, R-type's Exec, Store's Reg, Beq's Ifetch

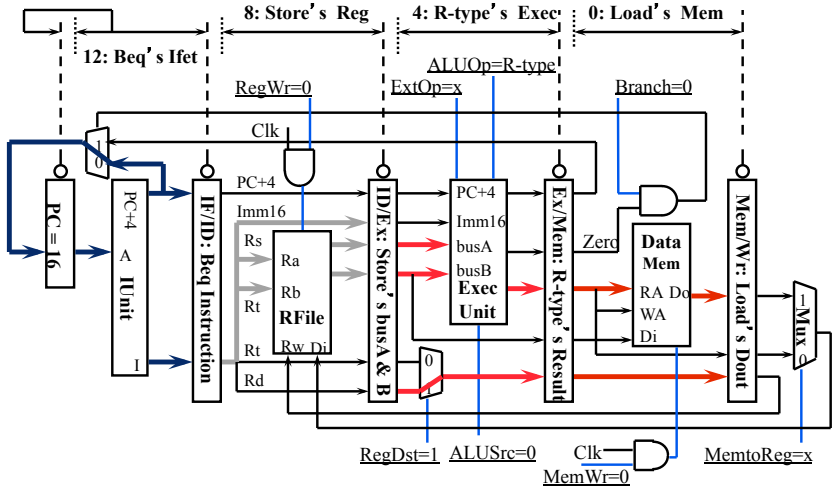
° End of Cycle 5: Load's Wr, R-type's Mem, Store's Exec, Beq's Reg

° End of Cycle 6: R-type's Wr, Store's Mem, Beq's Exec

° End of Cycle 7: Store's Wr, Beq's Mem

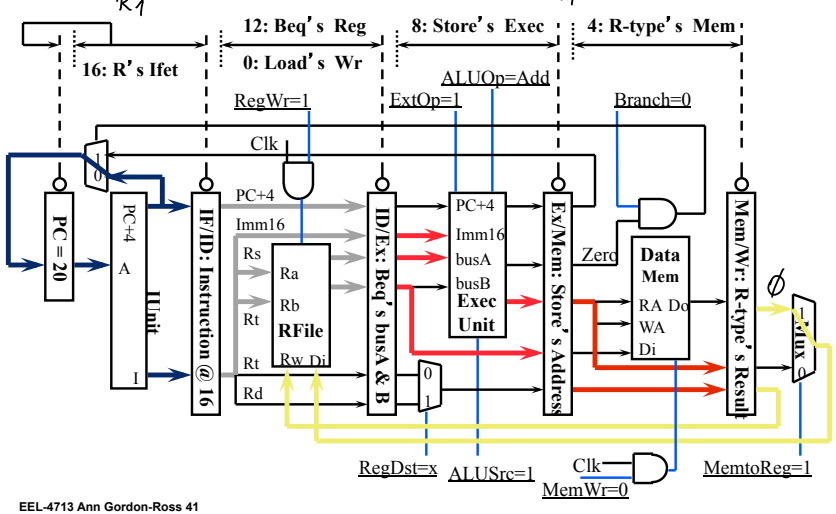
Pipelining Example: End of Cycle 4

° 0: Load's Mem 4: R-type's Exec 8: Store's Reg 12: Beq's Ifetch



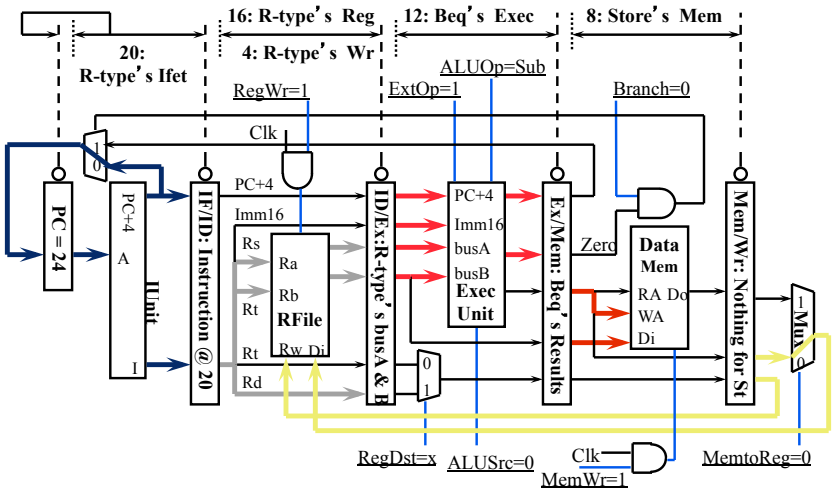
Pipelining Example: End of Cycle 5

° 0: Lw's Wr 4: R's Mem 8: Store's Exec 12: Beq's Reg 16: R's Ifetch



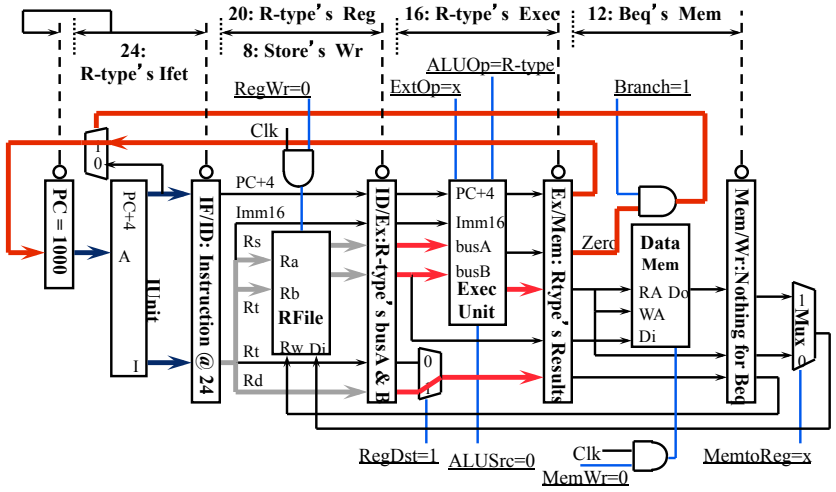
Pipelining Example: End of Cycle 6

° 4: R's Wr 8: Store's Mem 12: Beq's Exec 16: R's Reg 20: R's Ifet

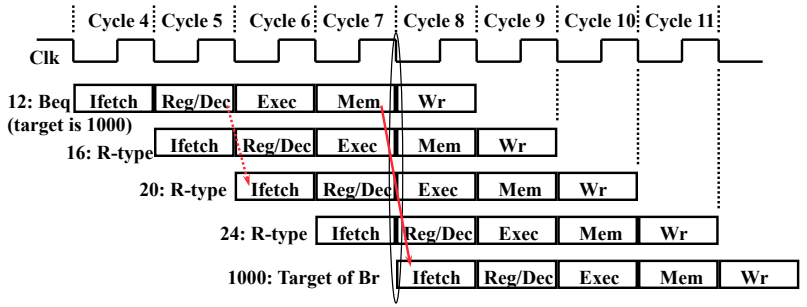


Pipelining Example: End of Cycle 7

° 8: Store's Wr 12: Beq's Mem 16: R's Exec 20: R's Reg 24: R's Ifet



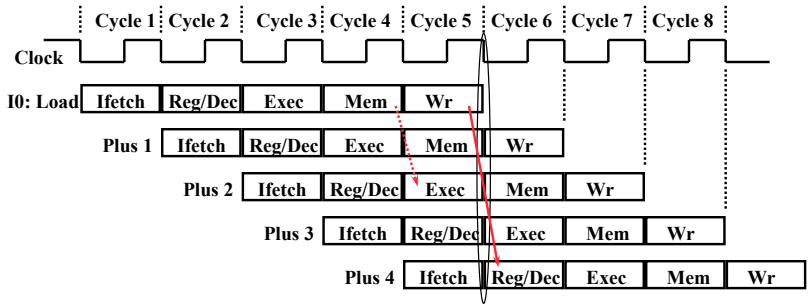
The Delay Branch Phenomenon



° Although Beq is fetched during Cycle 4:
- Target address is NOT written into the PC until the end of Cycle 7
- Branch's target is NOT fetched until Cycle 8
- 3-instruction delay before the branch take effect

° This is referred to as Branch Hazard:
- Clever design techniques can reduce the delay to ONE instruction

The Delay Load Phenomenon



- Although Load is fetched during Cycle 1:
  - The data is NOT written into the Reg File until the end of Cycle 5
  - We cannot read this value from the Reg File until Cycle 6
  - 3-instruction delay before the load take effect
- This is referred to as Data Hazard:
  - Clever design techniques can reduce the delay to ONE instruction

Summary

- Disadvantages of the Single Cycle Processor
  - Long cycle time
  - Cycle time is too long for all instructions except the Load
- Multiple Clock Cycle Processor:
  - Divide the instructions into smaller steps
  - Execute each step (instead of the entire instruction) in one cycle
- Pipeline Processor:
  - Natural enhancement of the multiple clock cycle processor
  - Each functional unit can only be used once per instruction
  - If a instruction is going to use a functional unit:
    - it must use it at the same stage as all other instructions
  - Pipeline Control:
    - Each stage's control signal depends ONLY on the instruction that is currently in that stage