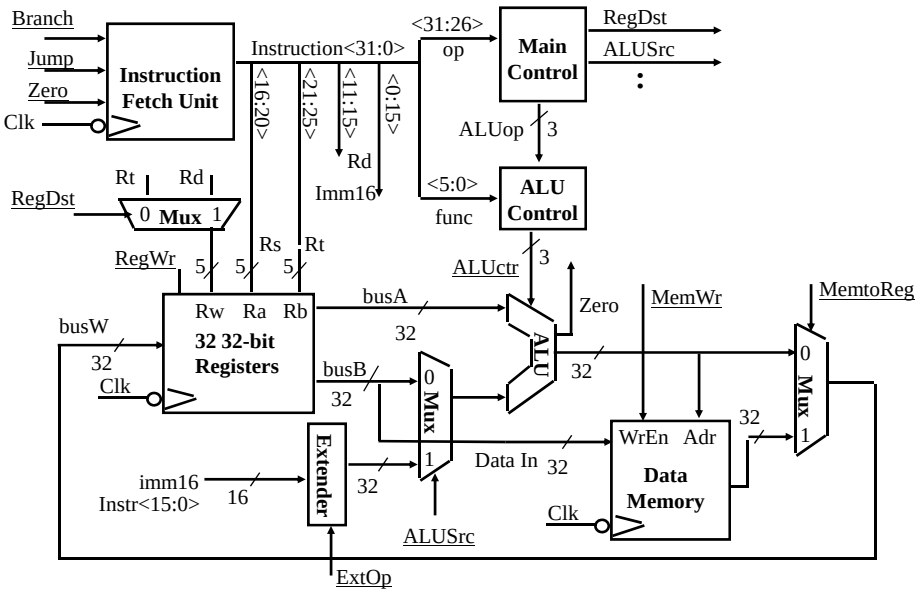
CS420/520
Computer Architecture I

Designing a Pipeline Processor
(CA4: Appendix A)

Dr. Xiaobo Zhou
Department of Computer Science

Recap: A Single Cycle Processor



Recap: Drawbacks of this Single Cycle Processor

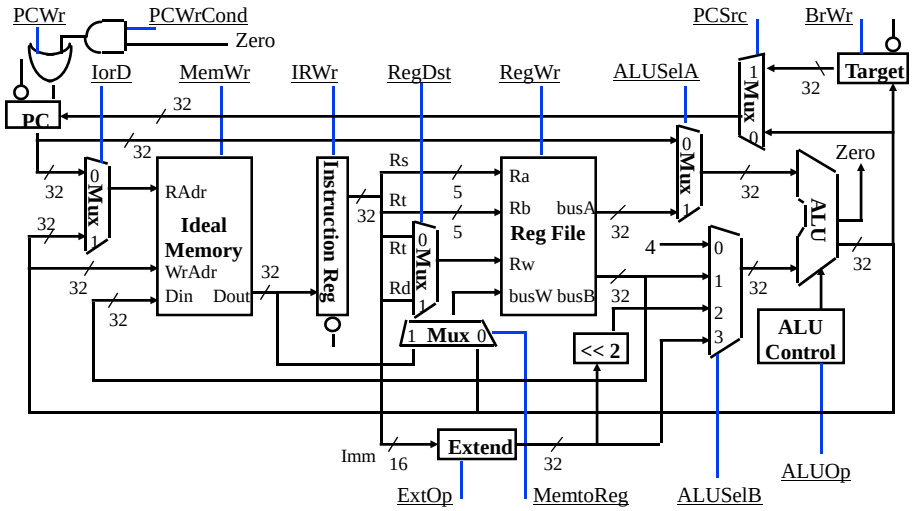
- Long cycle time:
  - Cycle time must be long enough for the load instruction:
    - PC's Clock -to-Q +
    - Instruction Memory Access Time +
    - Register File Access Time +
    - ALU Delay (address calculation) +
    - Data Memory Access Time +
    - Register File Setup & Writing Time +
    - Clock Skew
- Cycle time is much longer than needed for all other instructions. Examples:
  - R-type instructions do not require data memory access
  - Jump does not require ALU operation nor data memory access

Recap: Overview of a Multiple Cycle Implementation

- The root of the single cycle processor's problems:
  - The cycle time has to be long enough for the slowest instruction
- Solution:
  - Break the instruction into smaller steps
  - Execute each step (instead of the entire instruction) in one cycle
    - Cycle time: time it takes to execute the longest step
    - Keep all the steps to have similar length
  - This is the essence of the multiple cycle processor
- The advantages of the multiple cycle processor:
  - Cycle time is much shorter
  - Different instructions take different number of cycles to complete
    - Load takes five cycles
    - Jump only takes three cycles
  - Allows a functional unit to be used more than once per instruction

Recap: Multiple Cycle Processor

° MCP: A functional unit to be used more than once per instruction



Outline of Today's Lecture--- Pipelining

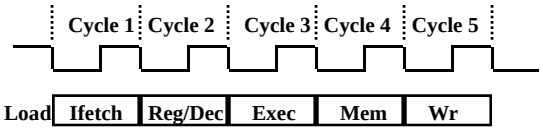
° Introduction to the Concept of Pipelined Processor

° Pipelined Datapath and Pipelined Control

° How to Avoid Race Condition in a Pipeline Design?

° Pipeline Example: Instructions Interaction

Preview: The Five Stages of Load

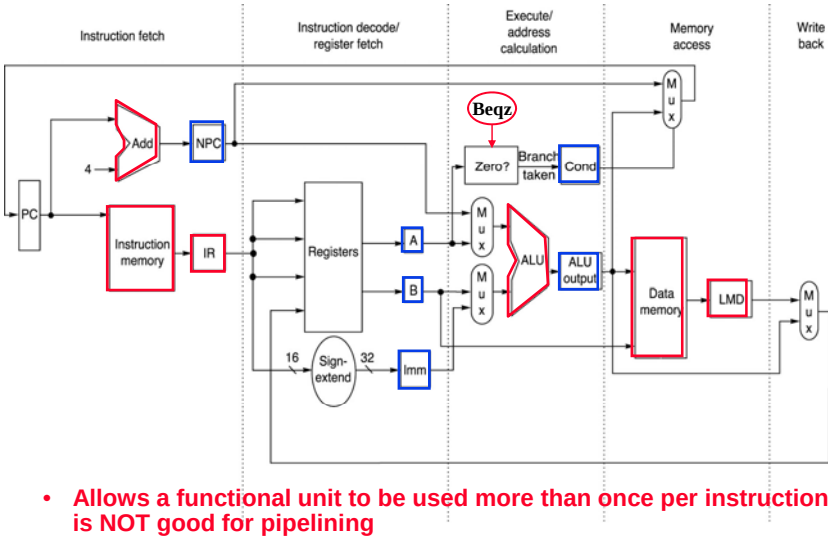


- Ifetch: Instruction Fetch
  - Fetch the instruction from the Instruction Memory
- Reg/Dec: Registers Fetch and Instruction Decode
- Exec: Calculate the memory address
- Mem: Read the data from the Data Memory
- Wr: Write the data back to the register file

Pipelining: Its Natural!

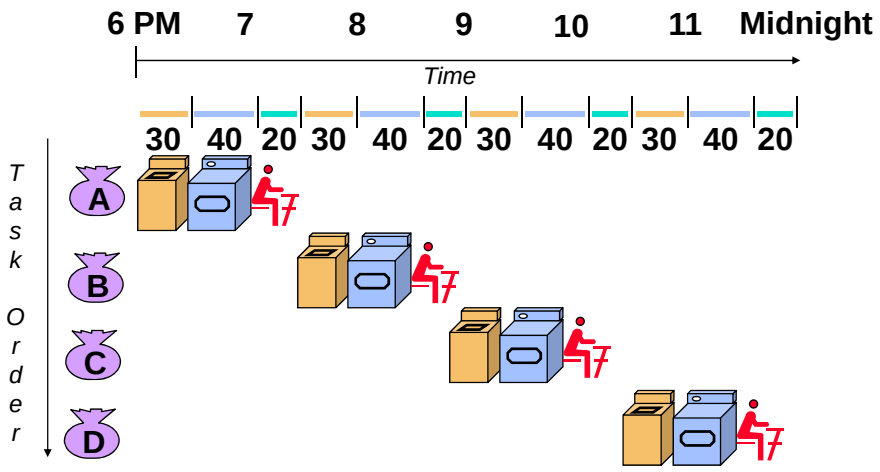
- Laundry Example
- Ann, Brian, Cathy, Dave each have one load of clothes to wash, dry, and fold
- Washer takes 30 minutes
- Dryer takes 40 minutes
- "Folder" takes 20 minutes

Recap: A Multiple Cycle Datapath (base for pipelining)



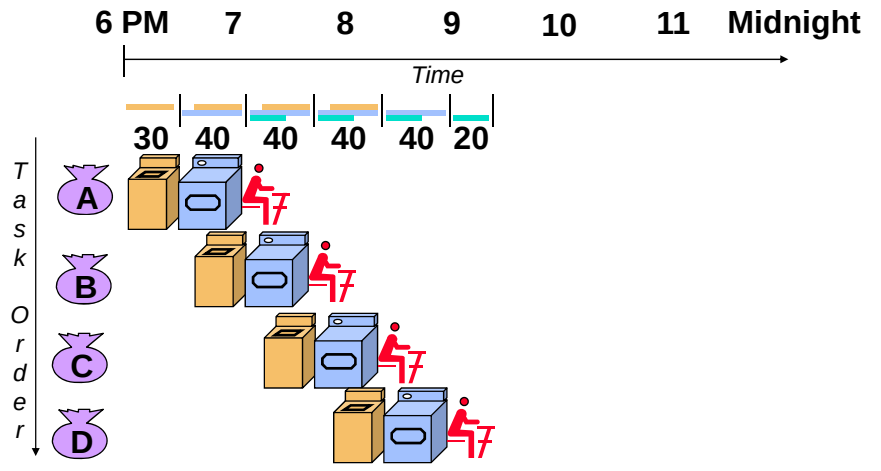
- **Allows a functional unit to be used more than once per instruction is NOT good for pipelining**
  - **Adder + ALU; Instruction mem + Data mem**

Sequential Laundry



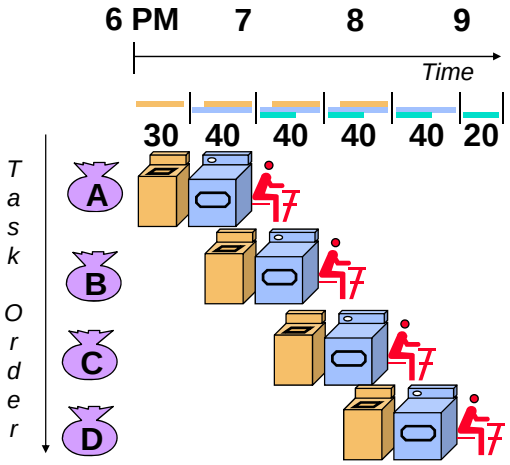
- **Sequential laundry takes 6 hours for 4 loads**
- **If they learned pipelining, how long would laundry take?**

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

° Pipeline rate limited by slowest pipeline stage

° Multiple tasks operating simultaneously (overlapped in execution, invisible to programmers)

° Potential speedup = Number pipe stages

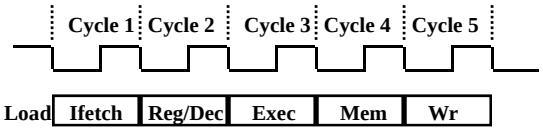
° Unbalanced lengths of pipe stages reduces speedup

° Time to "fill" pipeline and time to "drain" it reduces speedup

Key Ideas Behind Pipelining

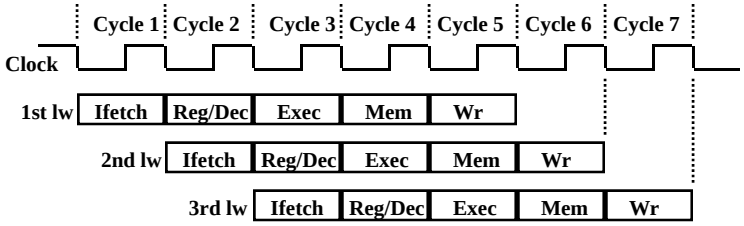
- Grading the mid term exams:
  - 5 problems, five people grading the exam
  - Each person ONLY grades one problem
  - Pass the exam to the next person as soon as one finishes his part
  - Assume each problem takes 0.5 hour to grade
    - Each individual exam still takes 2.5 hours to grade
    - But with 5 people, all exams can be graded much quicker
- The load instruction has 5 stages:
  - Five independent functional units to work on each stage
    - Each functional unit is used only once
  - The 2nd load can start as soon as the 1st finishes its left stage
  - Each load still takes five cycles to complete
  - The throughput, however, is much higher

The Five Stages of Load



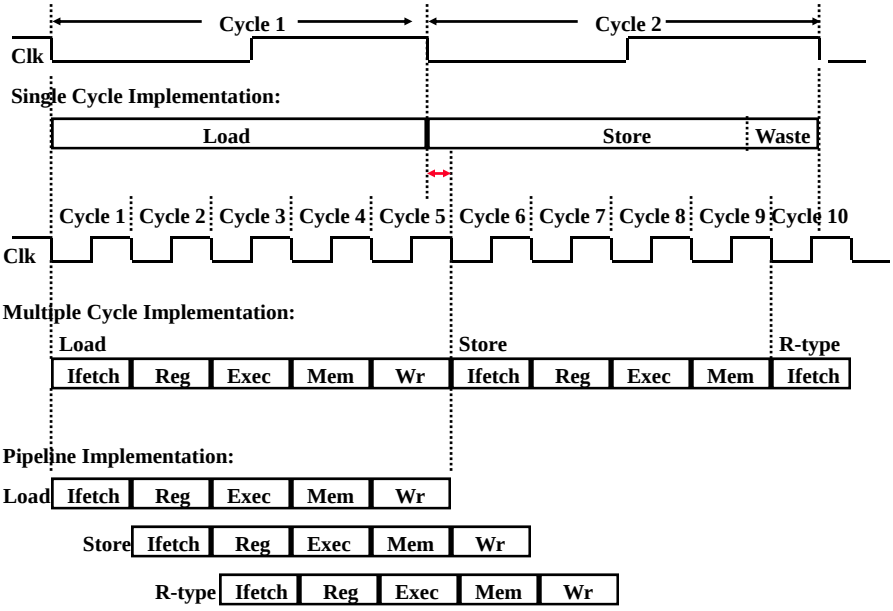
- Ifetch: Instruction Fetch
  - Fetch the instruction from the Instruction Memory
- Reg/Dec: Registers Fetch and Instruction Decode
- Exec: Calculate the memory address
- Mem: Read the data from the Data Memory
- Wr: Write the data back to the register file

Pipelining the Load Instruction



- The five independent functional units in the pipeline datapath are:
  - Instruction Memory for the Ifetch stage
  - Register File's 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

Single Cycle, Multiple Cycle, vs. Pipeline



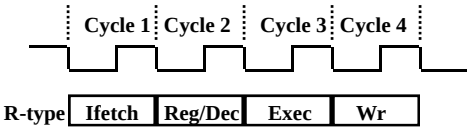
Why Pipeline?

- Suppose we execute 100 instructions
- Single Cycle Machine
  - 45 ns/cycle x 1 CPI x 100 inst = 4500 ns
- Multicycle Machine
  - 10 ns/cycle x 4.1 CPI (due to inst mix) x 100 inst = 4100 ns
- Ideal pipelined machine
  - 10 ns/cycle x (1 CPI x 100 inst + 4 cycle drain) = 1040 ns

Compared to the Multi-cycle implementation, pipelining reduces the CPI!

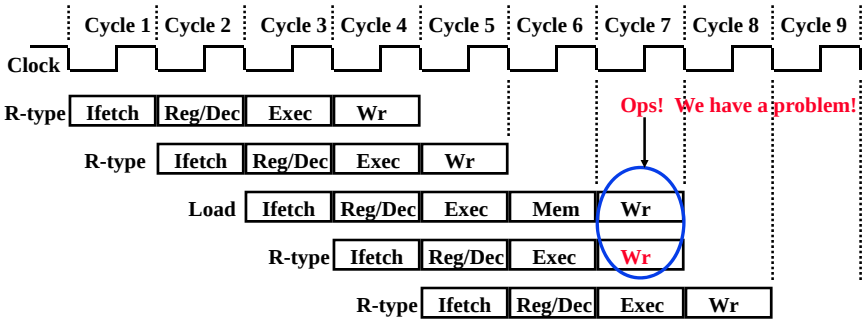
Compared to the Single-cycle implementation, pipelining reduces the clock cycle time!

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
  - ALU operates on the two register operands
  - Update PC
- Wr: Write the ALU output back to the register file

Pipelining the R-type and Load Instruction

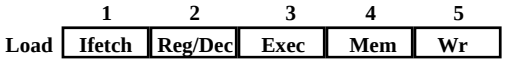


° We have a problem:
- Two instructions try to write to the register file at the same time!
- Only one write port

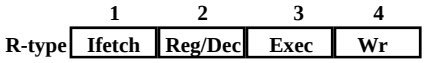
Important Observation

° Each functional unit can only be used once per instruction
(pipelining vs. multiple cycle)

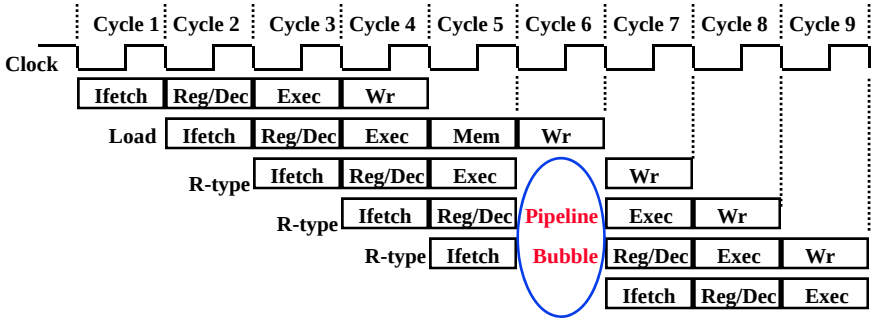
° Each functional unit must be used at the same stage for all instructions:
- Load uses Register File's Write Port during its 5th stage



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



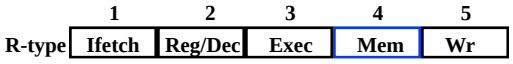
Solution 1: Insert “Bubble” into the Pipeline

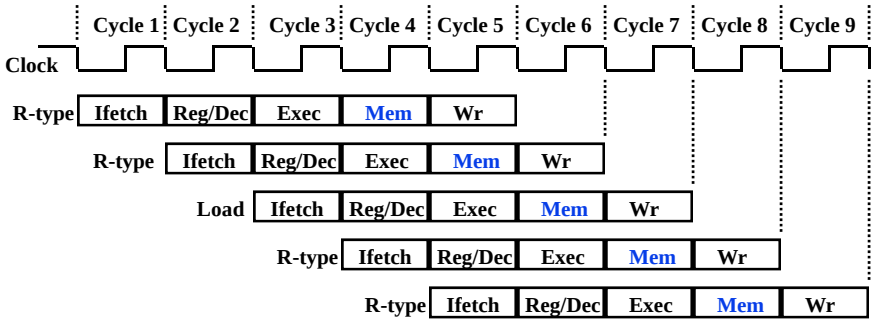


- Insert a “bubble” into the pipeline to prevent 2 writes at the same cycle
  - The control logic can be complex
- No instruction is completed during Cycle 5:
  - The “Effective” CPI for load is 2

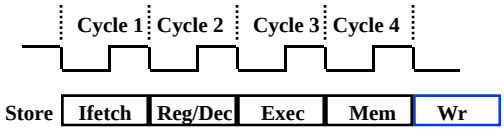
Solution 2: Delay R-type’s Write by One Cycle

- 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



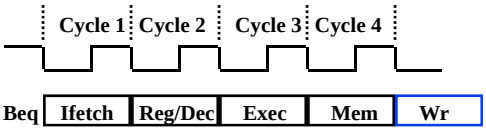


The Four Stages of Store



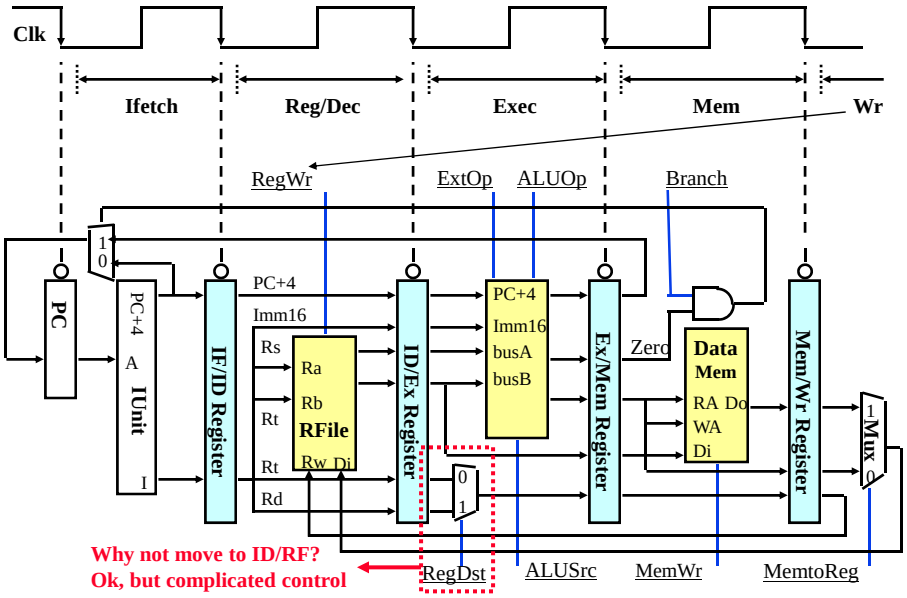
- Ifetch: Instruction Fetch
  - Fetch the instruction from the Instruction Memory
- Reg/Dec: Registers Fetch and Instruction Decode
- Exec: Calculate the memory address
- Mem: Write the data into the Data Memory

The Four Stages of Beq



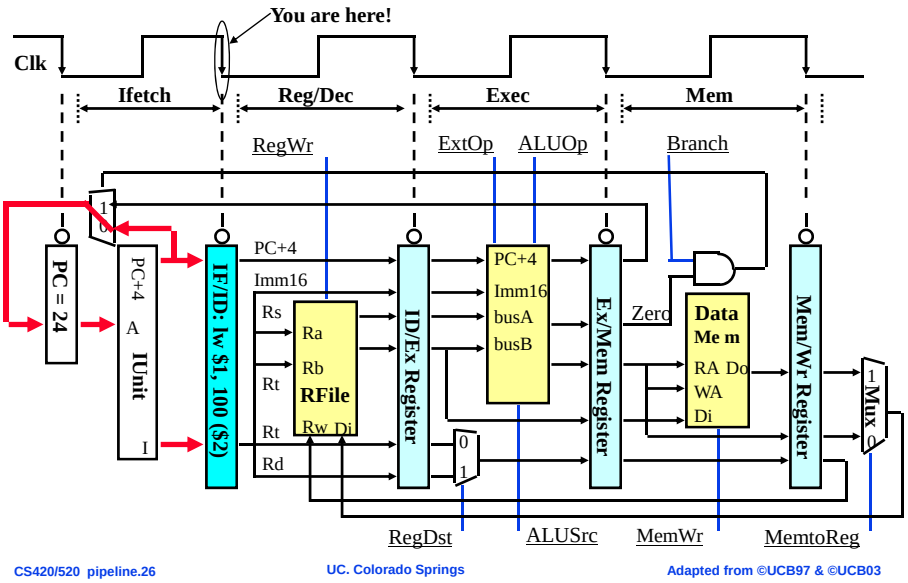
- Ifetch:
  - Instruction Fetch
  - Fetch the instruction from the Instruction Memory
- Reg/Dec:
  - Registers Fetch and Instruction Decode
- Exec:
  - ALU compares the two register operands
  - Adder calculates the branch target address
- Mem:
  - If the registers we compared in the Exec stage are the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

A Pipelined Datapath



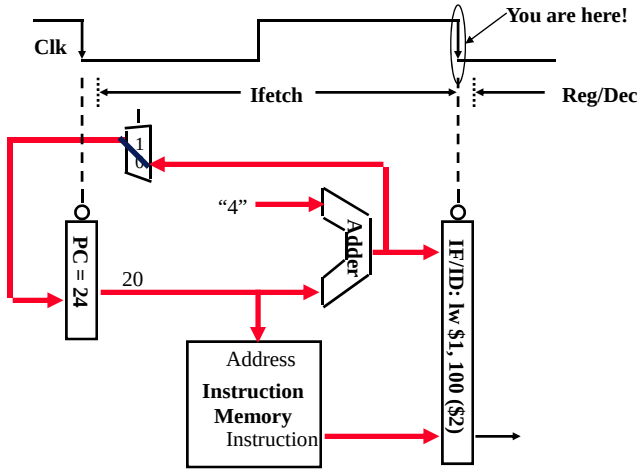
The Instruction Fetch Stage

° Location 20: lw $1, 0x100($2) $1 <- Mem[($2) + 0x100]



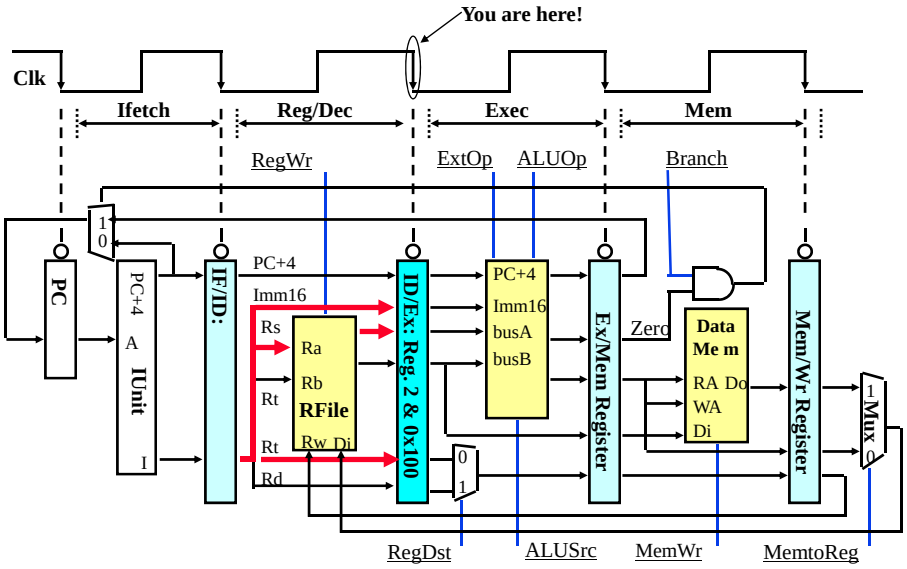
A Detail View of the Instruction Unit

° Location 20: lw $1, 0x100($2)



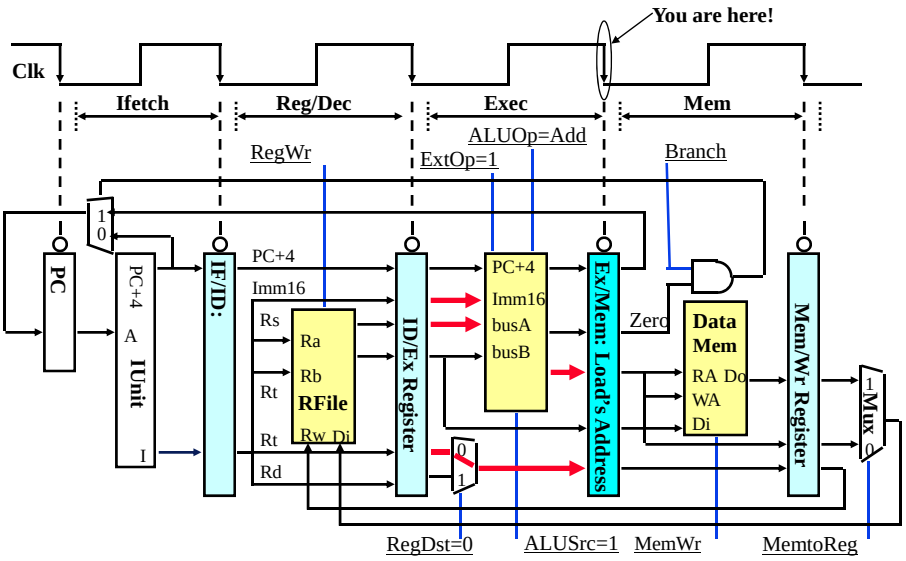
The Decode / Register Fetch Stage

° Location 20: lw $1, 0x100($2) $1 <- Mem[($2) + 0x100]

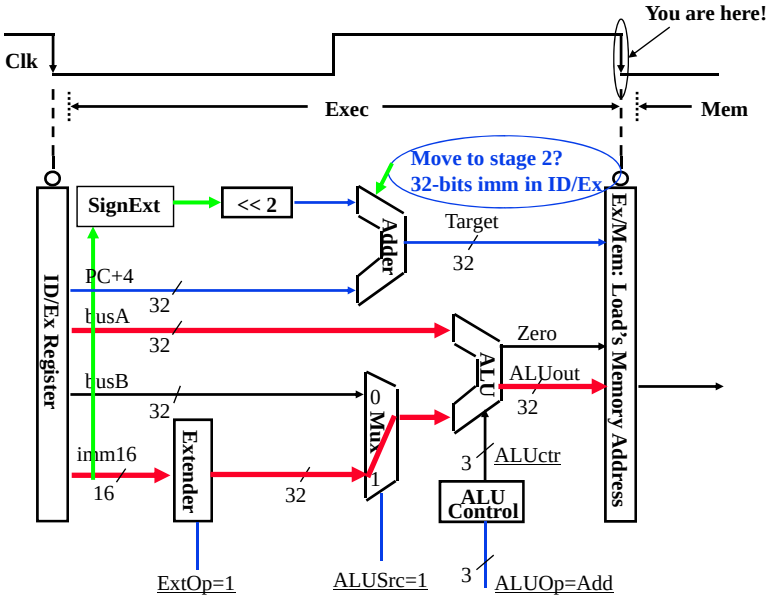


Load's Address Calculation Stage

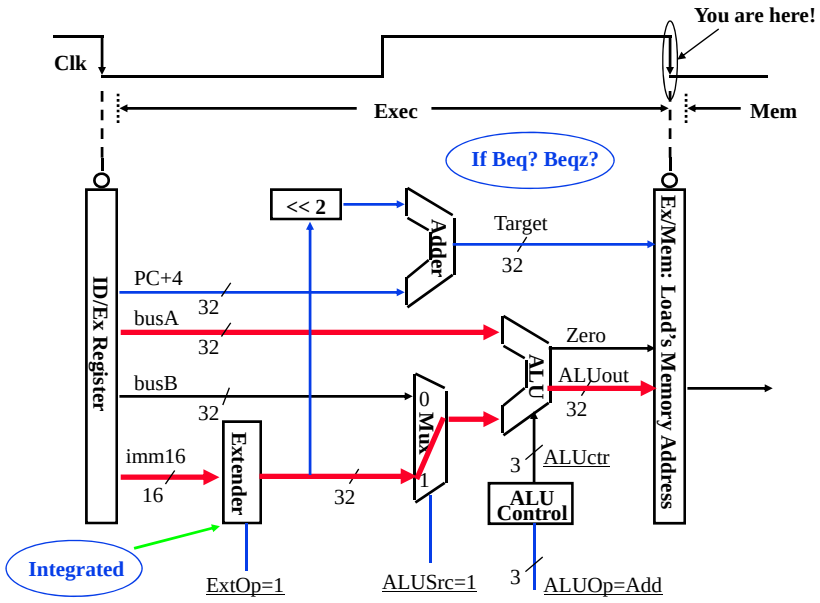
° Location 20: lw $1, 0x100($2) $1 <- Mem[($2) + 0x100]



A View of the Execution Unit (like in Single Cycle)

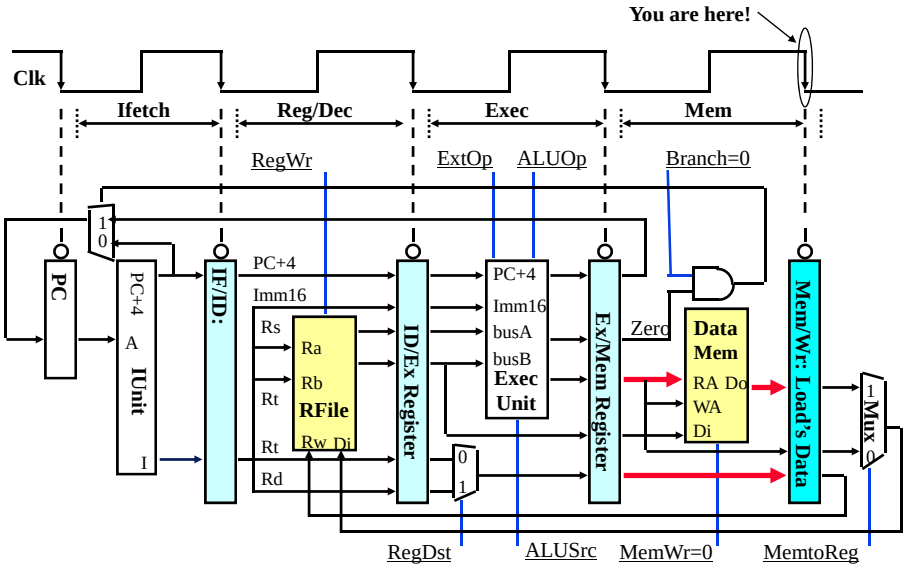


A Detail View of the Execution Unit (Integrated)



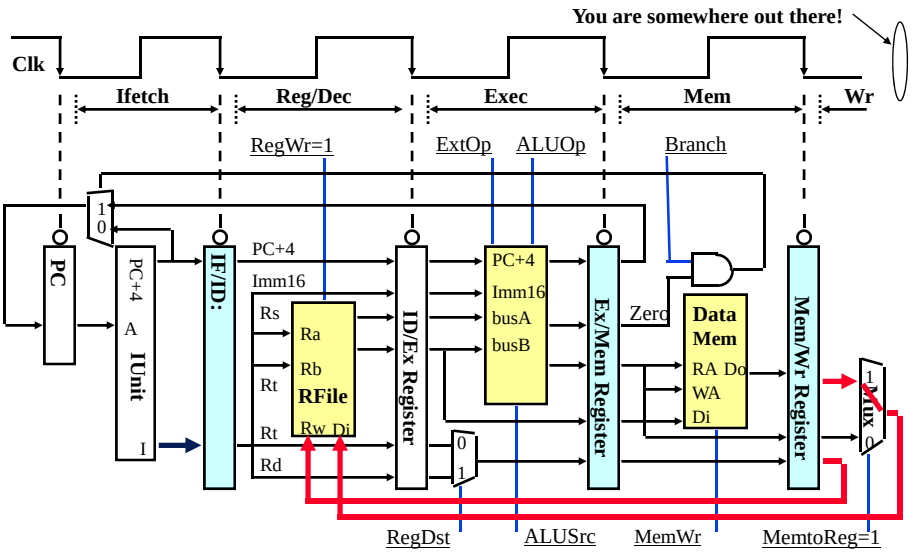
Load's Memory Access Stage

° Location 20: lw $1, 0x100($2) $1 <- Mem[($2) + 0x100]



Load's Write Back Stage

° Location 20: lw $1, 0x100($2) $1 <- Mem[($2) + 0x100]

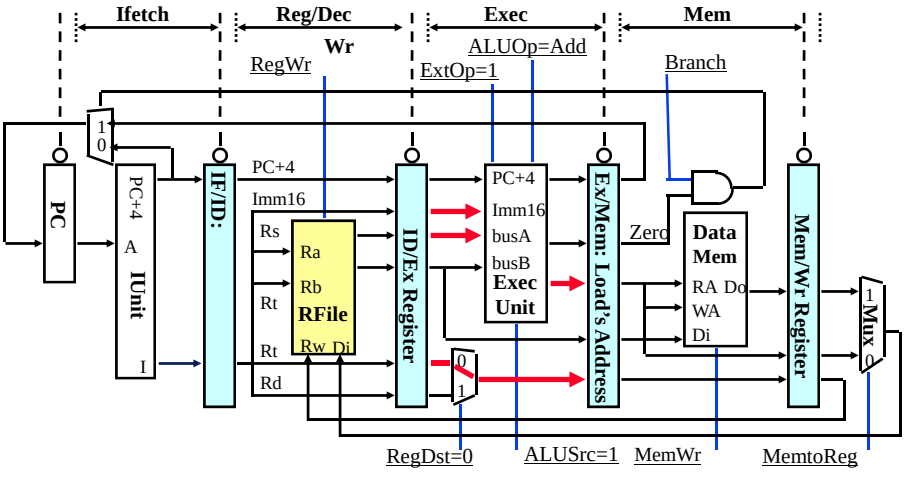


How About Control Signals?

° Key Observation: Control Signals at Stage N = Func (Instr. at Stage N)

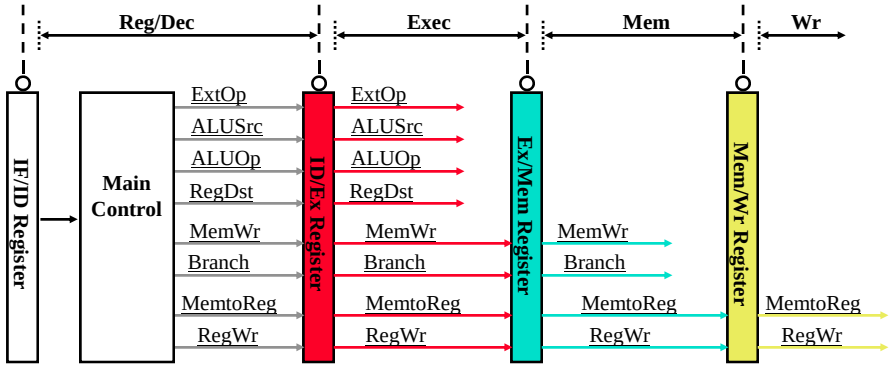
- N = Exec, Mem, or Wr

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

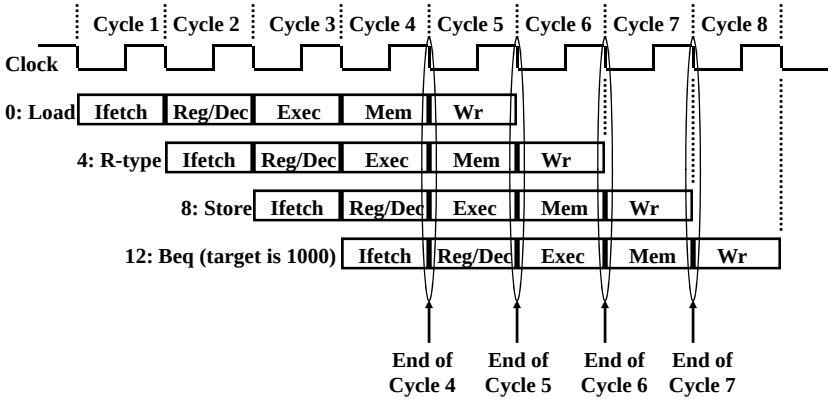


Pipeline Control

- 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



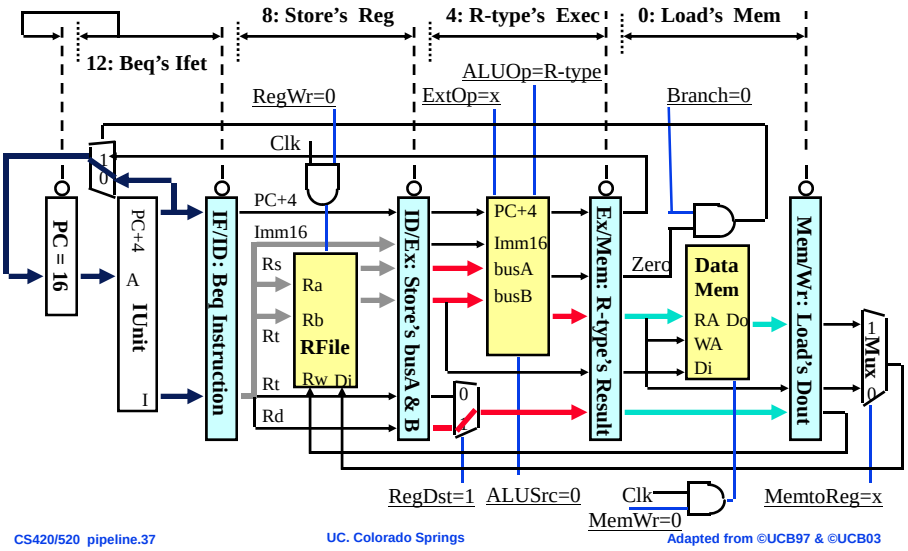
A More Extensive Pipelining Example



- 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

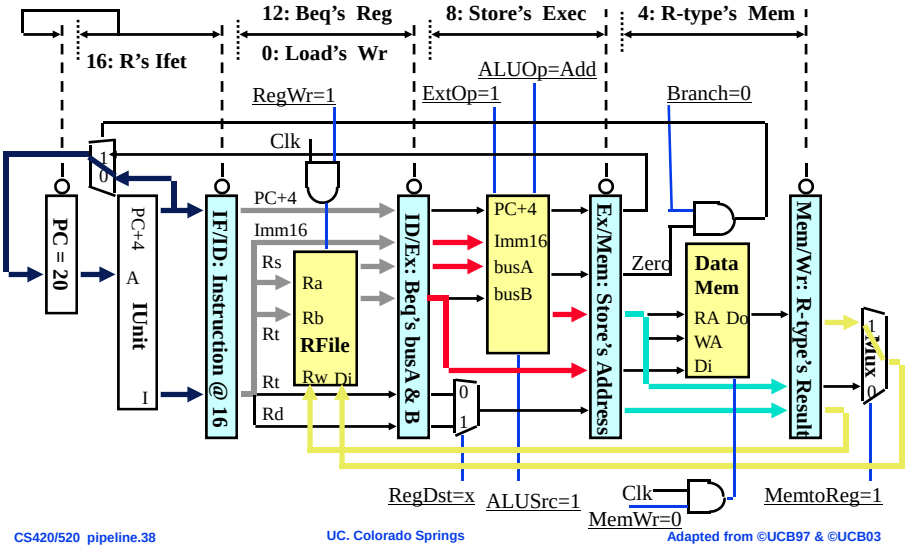
Pipelining Example: End of Cycle 4

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



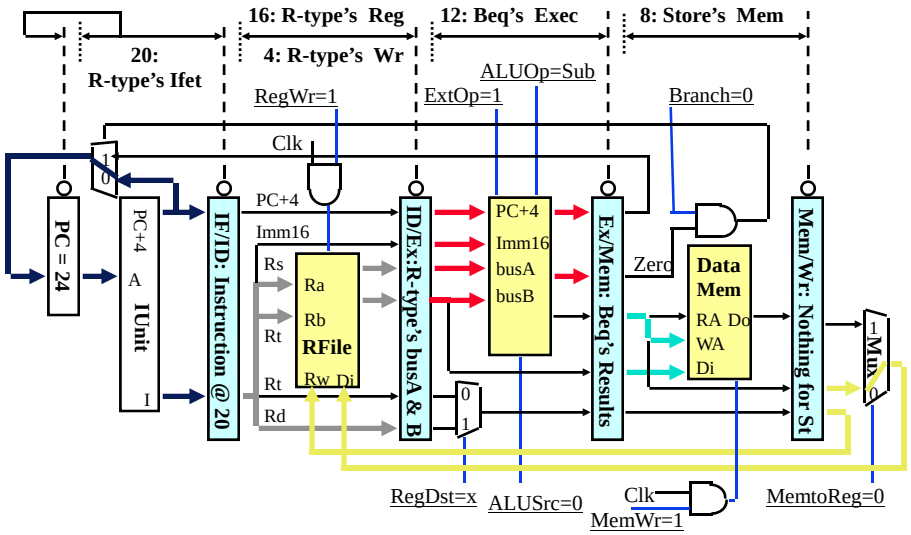
Pipelining Example: End of Cycle 5

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



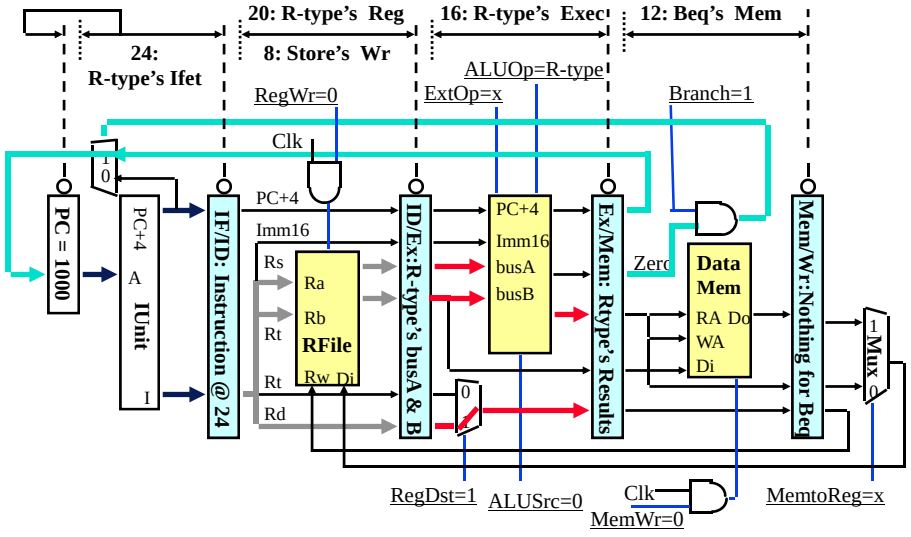
Pipelining Example: End of Cycle 6

° 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



Basic Performance Issues in Pipelining

° Pipelining increases the CPU instruction throughput, but does not reduce the execution time of an individual instruction
  - In fact, it slightly increases the execution time of an instruction

° Pipelining performance limitations
  - Pipelining latency due to hazards
  - Imbalance limits
    - Clock cannot run faster than the time needed for the slowest pipeline stage; hardware also limits the stage partitioning
  - Pipeline overhead
    - Pipeline registers setup and latency (separating instructions at different stages so as to avoid interfering with each others)
    - Clock skews, maximum delay between the clock arrives at any two registers (delay in signal arrival times)

° When pipelining is useless?

  ° once the clock cycle is as small as the sum of the clock skew and pipeline register (*latch)* latency, since no time left for useful work!

Pipelining Performance Example

° A un-pipelined (multi-cycle) processor has a 1ns clock cycle, and it uses 4 cycles for *ALU* operations and *Branche*s, 5 for *Memory* operations. The relative frequencies of three operations is 40%, 20%, and 40%.

° Due to clock skew and setup, pipelining the processor adds 0.2ns into clock cycle. Suppose there is no pipelining hazard so that pipelining CPI is 1, how much speedup will we gain from a pipeline?

Answer:

For un-pipelined processor:

Ave. instruction exec. Time = clock cycle time * average CPI

(AIET) = 1 ns (40% * 4 + 20% *4 + 40% * 5)

= 4.4 ns

For pipelined processor:

Ave. instruction exec. Time = (1 + 0.2) ns * 1 = 1.2 ns

Speedup = AIET_w/o pipeling / AIET_w/pipeline = 4.4 ns / 1.2 ns = 3.7

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

Where to get more information?

- Appendix A of CA4 (or CA3) text book:
  - Chapter A.1 and A.3:
- CO2: Chapter 6.1 – 6.3

  CO3: Chapter 6.1 – 6.3
  - David Patterson and John Hennessy, "Computer Organization & Design: The Hardware / Software Interface," Morgan Kaufman Publishers; CO2 (2nd edition) and CO3 (3rd edition)