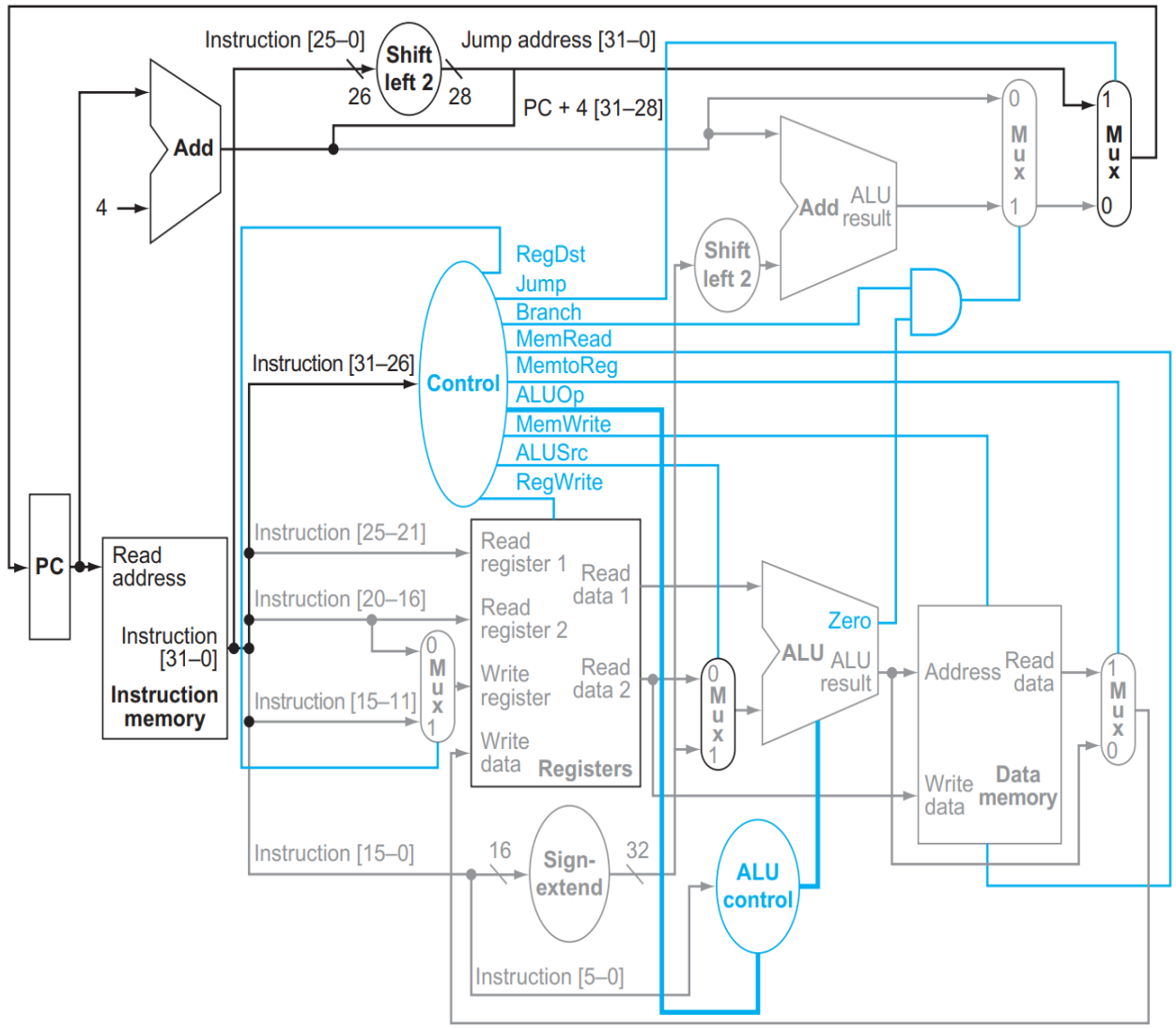
Computer Architecture
(ENE1004)

Lec – 14: The Processor (Chapter 4) - 5

Upcoming Schedule

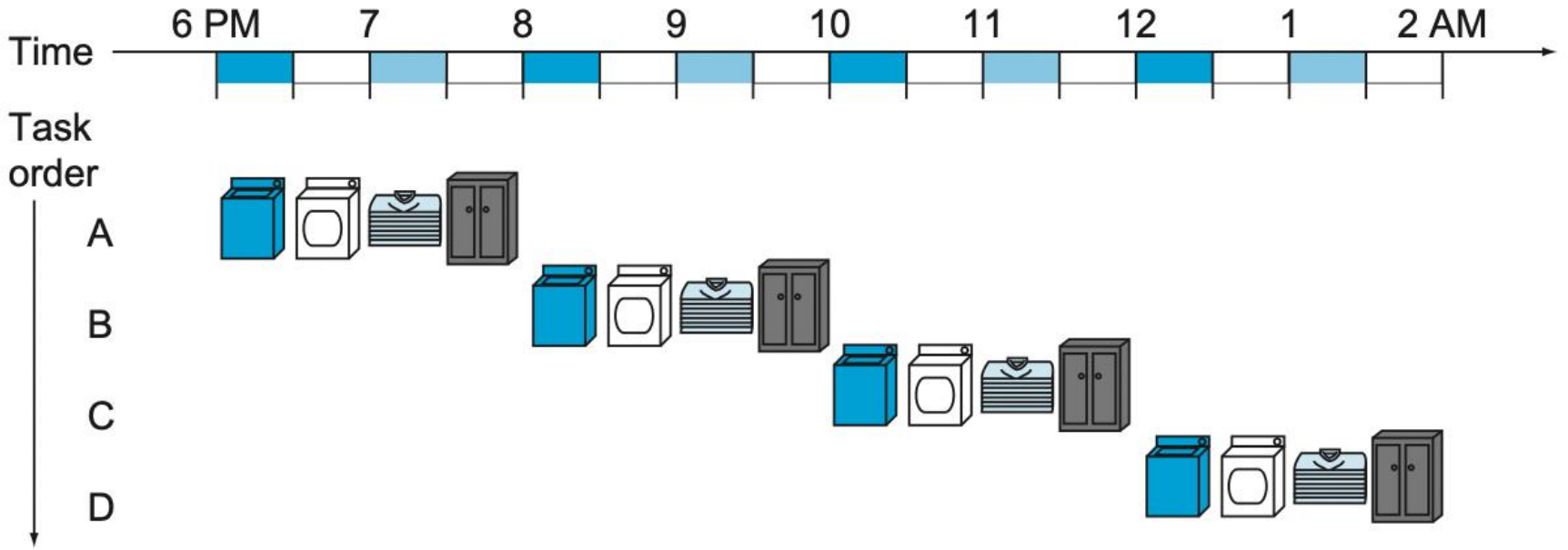
- May 4 (this Thursday) – No class
- May 15 – recorded video lecture
- May 21 – Assignment #1 deadline

A Simple Implementation of DataPath



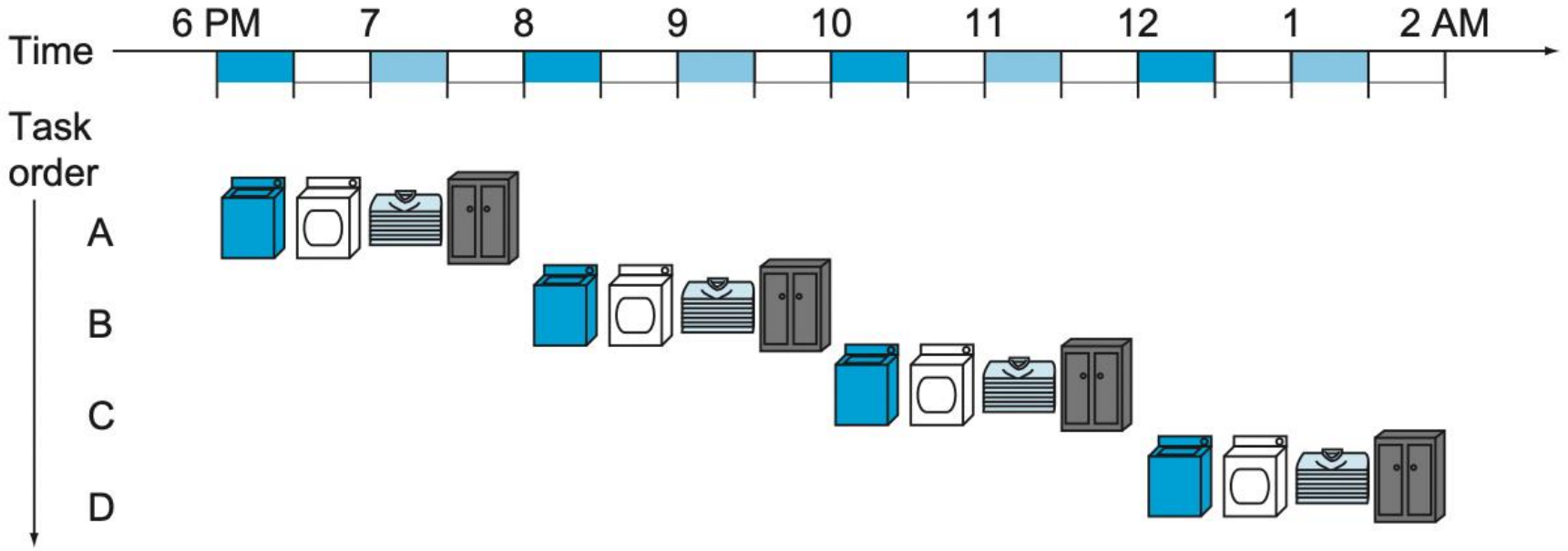
- This implementation can process any given instruction
  - R-type, load/store, branch, jump
- However, this is not used in today's processors; why?
  - A new instruction cannot be executed till its previous instruction is completed; only one instruction at a time!
  - Data flow of an instruction involves various units, which needs a long time
    - **lw** takes the longest time, as it needs (i) fetching instruction, (ii) reading registers, (iii) performing an ALU operation, (iv) reading memory, and (v) writing to a register
  - So, this implementation is not a good choice from a performance angle
- Let us see an advanced version

An analogy of Laundry



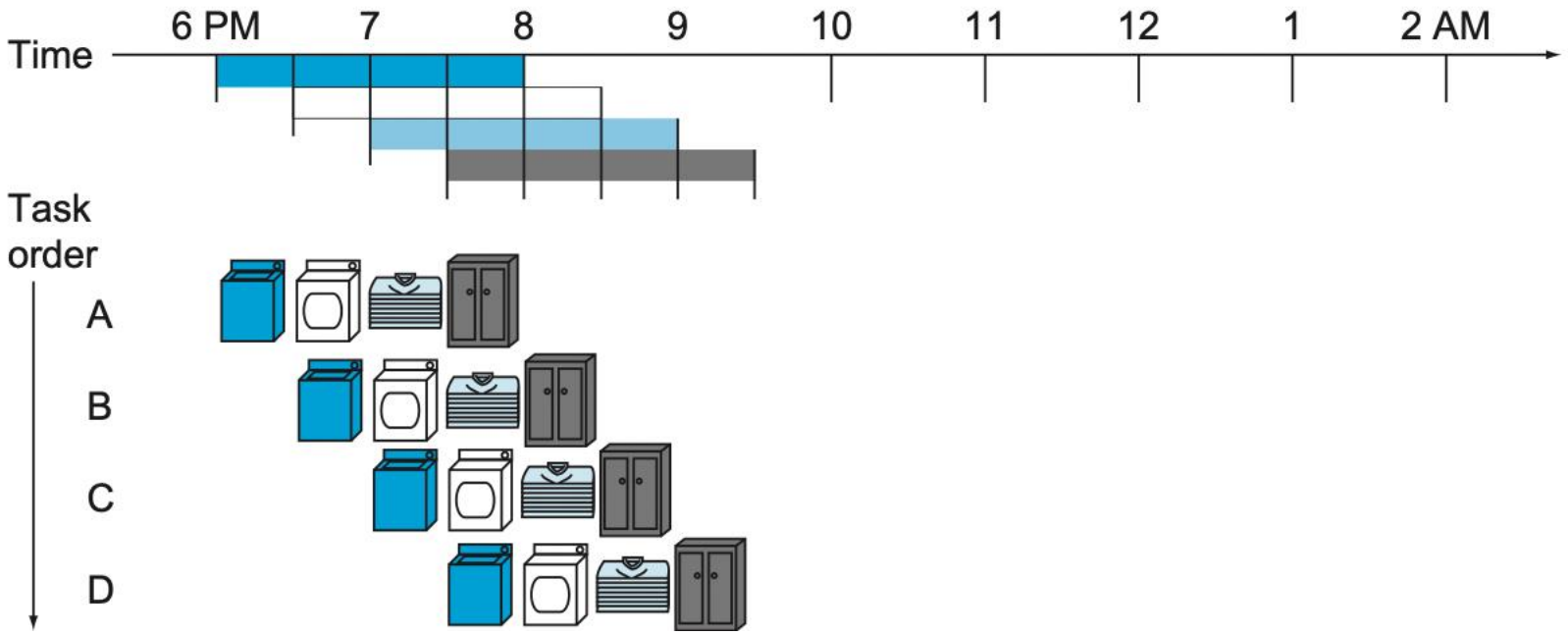
- For a single load of clothes,
  - (1) Place the load in the washer
  - (2) When the washer is finished, place the wet load in the dryer
  - (3) When the dryer is finished, place the dry load on a table and fold
  - (4) When folding is finished, put the clothes in the wardrobe
- Assume that
  - Each of the four steps takes 30 minutes (30 minutes for a job)
  - There are multiple loads of clothes (Task A, B, C, D, …)

An analogy of Laundry (2)



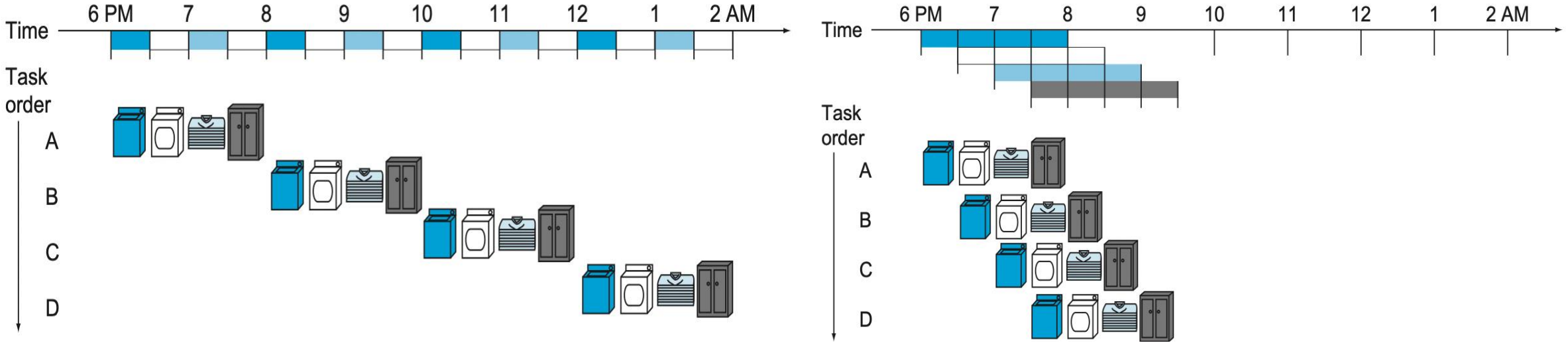
- One way of doing the laundry is to process tasks one after another
  - For Task A, do everything (washer-dryer-folding-wardrobe)
  - When Task A is done, start over the entire process for Task B
  - When Task B is done, start over the entire process for Task C …
- This way takes a total of 8 hours for the four tasks
  - It starts at 6PM and ends at 2AM
- Is there any better way to reduce the total time?

Pipelined Approach



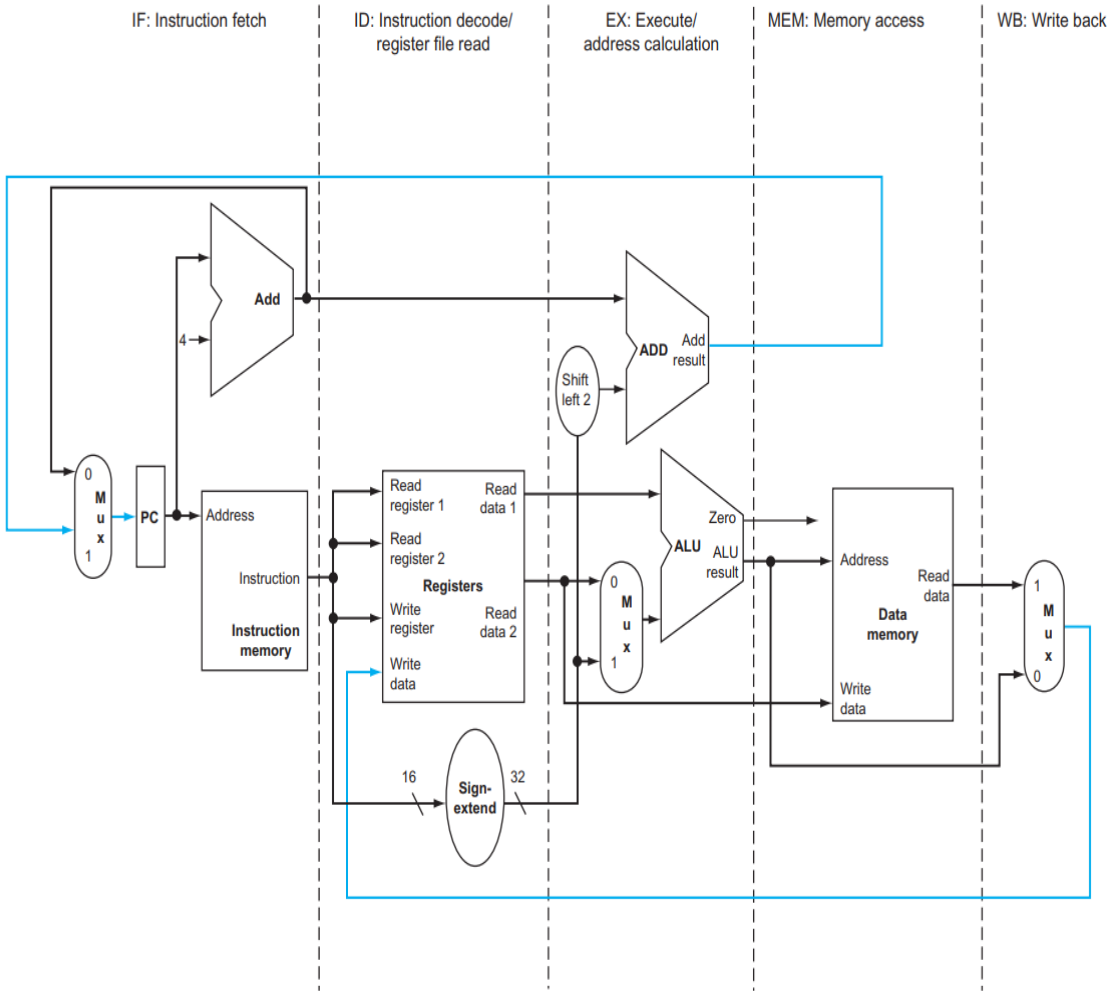
- Why don't we start the next tasks as soon as possible?
  - When the washer is finished with Task A (at 6:30), place Task A in the dryer & Task B in the washer
  - When drying A is done (at 7:00), start folding it, move B to the dryer, & put C into the washer
  - When A is put to wardrobe (at 7:30), start folding TB, dry C, & put D into the washer
- In this way, the total time for A, B, C, & D is significantly reduced from 8 to 3.5 hours
- Here, all steps (washer, dryer, folding, wardrobe) are operating concurrently
  - We call this way of processing tasks "pipelining" and each step of the pipelining "stage"

Pipelined Approach: Response Time vs Throughput



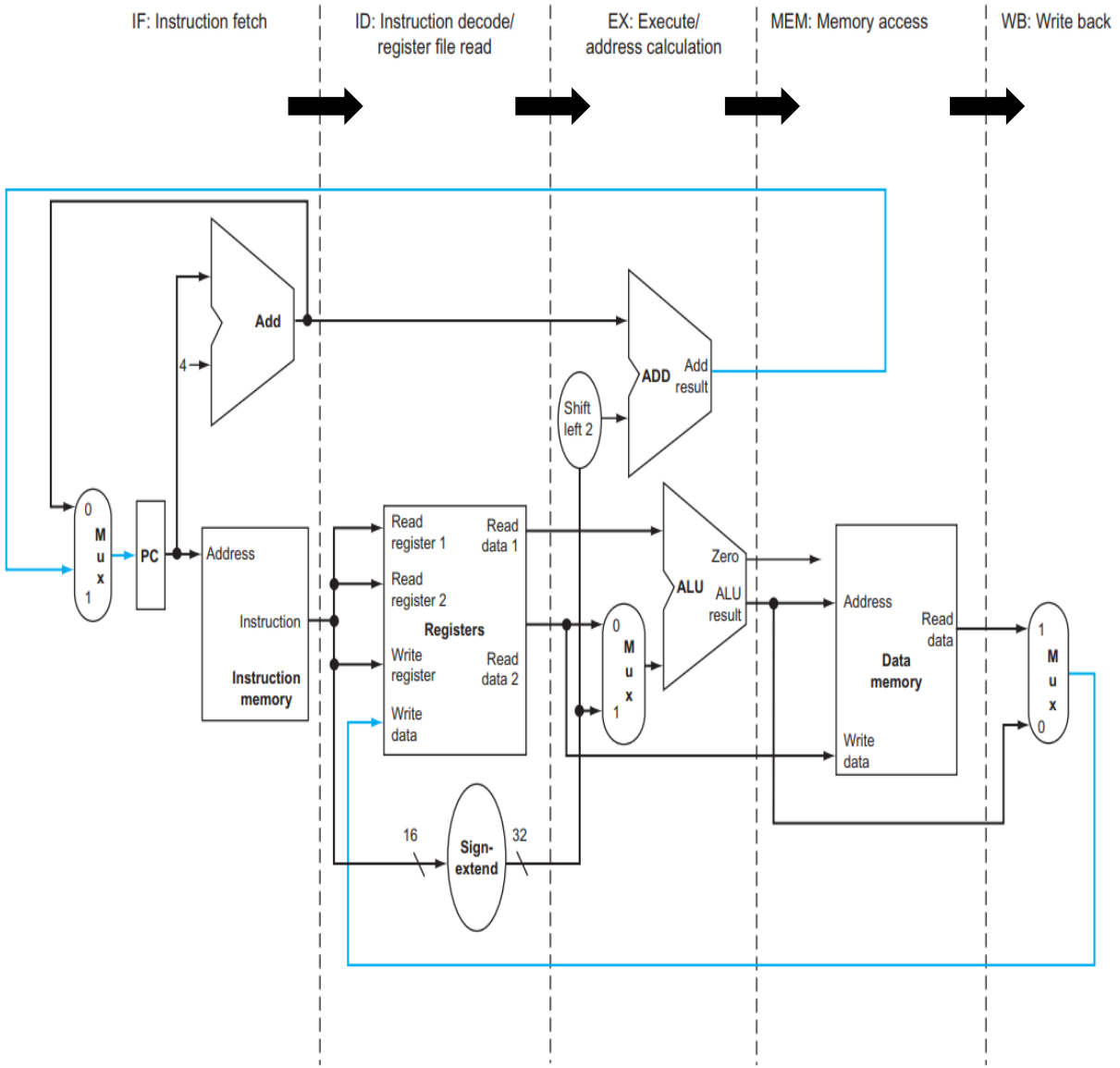
- Pipelined approach is better than non-pipelined approach in terms of performance
  - Is the time to do laundry for a single load (a single task) reduced?
    - No, it still takes 2 hours to go through the four stages — from washer, dryer, folding, to wardrobe
  - However, the time to do laundry for the four loads (four tasks) is reduced
- Recall the two performance metric, response time vs throughput, discussed in Lec-1
  - Pipelining can improve the throughput (the number of tasks to be processed in a unit time)
  - Pipelining cannot improve the response time (the execution time of each single task)

Five Stages of a Datapath



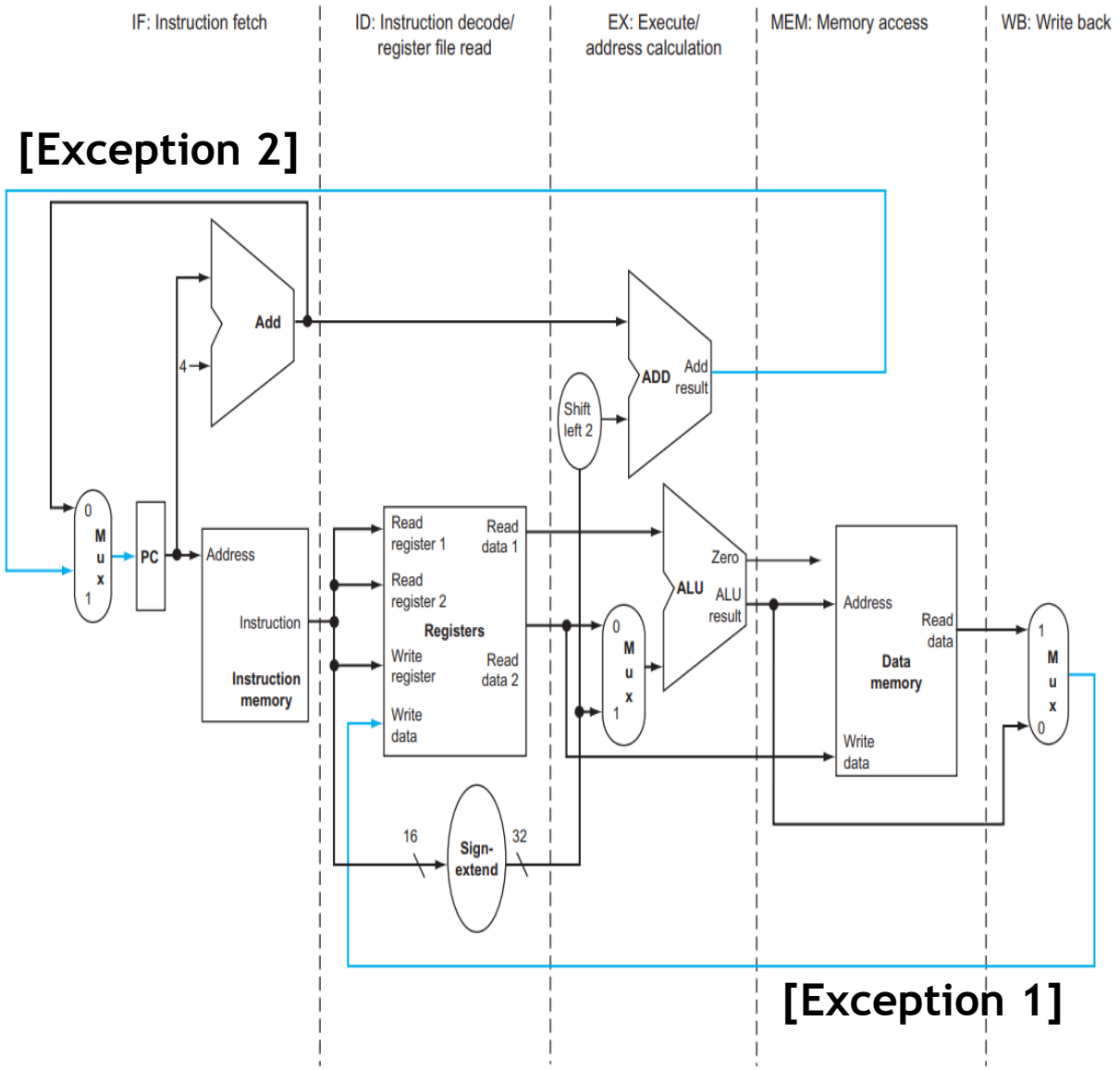
- Analogy
  - Task = instruction
  - Stage = an activity with a hardware unit while processing the instruction
- Stages in the datapath
  - (1) IF: Fetch instruction from memory
  - (2) ID: Read registers while decoding instruction
  - (3) EX: Execute operation or compute address
  - (4) MEM: Access an operand in data memory
  - (5) WB: Write the result into a register

Five Stages of a Datapath: Left-to-Right Flow



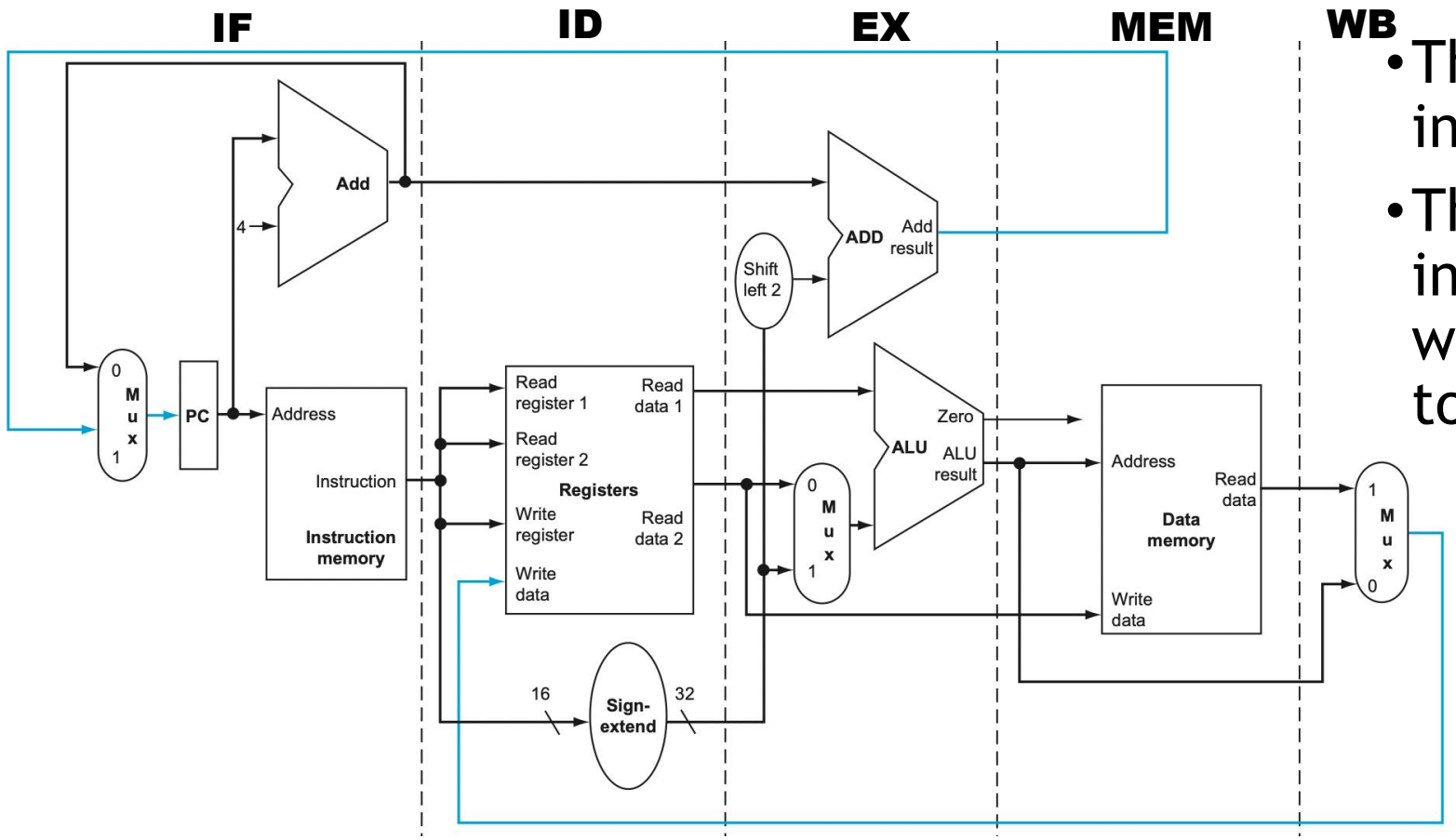
- Let us employ pipelining in this datapath
  - Once 1st instruction moves from IF to ID, 2nd instruction can be started in IF
  - When 1st instruction moves from ID to EX, 2nd and 3rd instructions move to ID and IF, respectively
- Pipelining is feasible in a datapath since instructions move generally from left to right through the five stages
  - In the laundry analogy, clothes also get cleaner, drier, and more organized as they move through the line from left to right
  - But, they never move backward

Five Stages of a Datapath: Two Exceptions



- There are <u>two exceptions</u> to this left-to-right flow of instructions
  - Data can move from right to left
  - These reverse data movements influence following instructions in the pipeline
- <u>Exception 1</u>: WB stage
  - This stage places the result back into the register file in the middle of the datapath
  - This influences the instruction now in ID stage
- <u>Exception 2</u>: Update of PC
  - The next value of the PC should be selected from the incremented PC (PC+4) or the branch target address
  - The branch target address is obtained in EX stage, and it needs to move to the PC
  - This influences the instruction now in IF stage

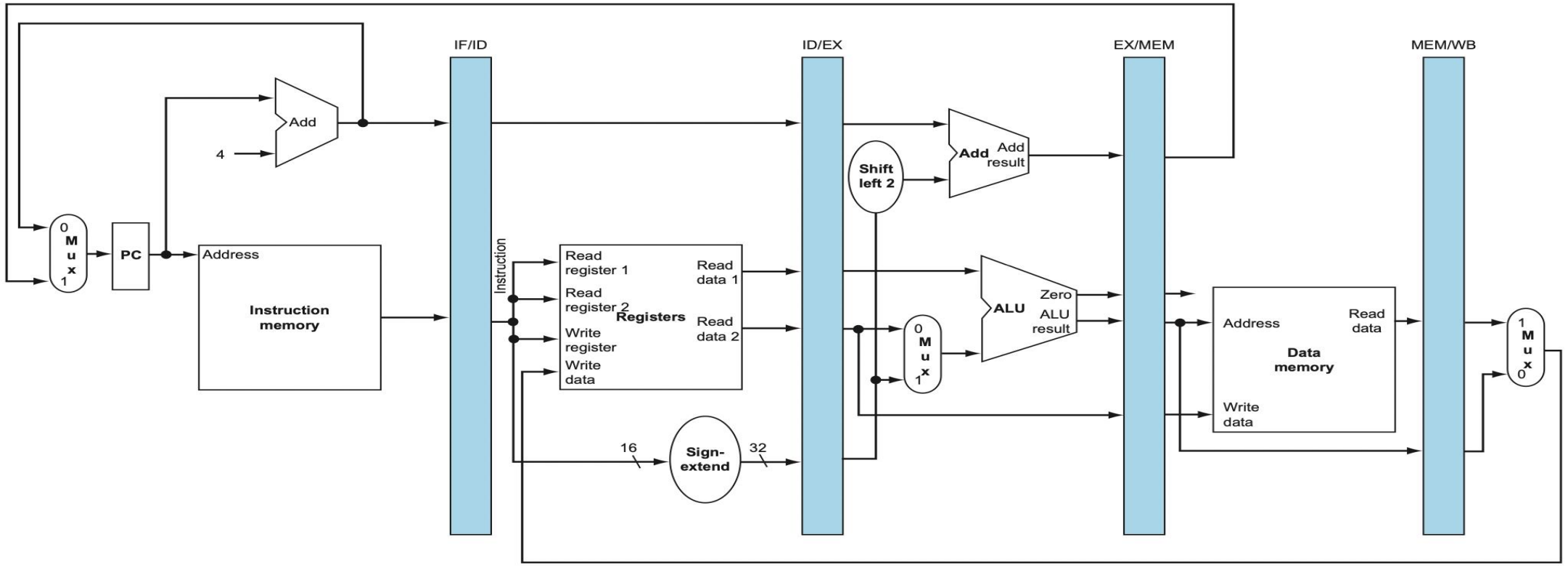
Need of Keeping Information of Instructions



- There is another serious problem in the non-pipelined datapath
- The information of an instruction in a stage is lost when the next instruction enters to the stage
  - @t, **beq** is fetched in IF
  - @t+1, **beq** moves to ID
  - @t+1, **lw** is fetched in IF
    - Here, (PC+4)+4; but, **beq** needs PC+4
  - @t+2, **beq & lw** move to EX & ID
    - Here, a sign-extended value of 10 can be given to the adder; but, **beq** needs a sign-extended value of 100
- Similar problematic situations can happen to every stage

@t	**beq $t1,$t2,100**		
@t+1	**lw $s1,10($s2)**	**beq $t1,$t2,100**	
@t+2	3rd instruction	**lw $s1,10($s2)**	**beq $t1,$t2,100**

Pipelined Datapath with Pipeline Registers



- We place (pipeline) registers between any two stages
  - In the laundry analogy, we might have a basket between any two steps to hold the clothes
  - The registers are named for the two stages separated by that register (e.g., ID/EX btw ID and EX)
- All instructions advance from one pipeline register to the next at a time
  - As an instruction advances, its required information is also copied to the next pipeline registers