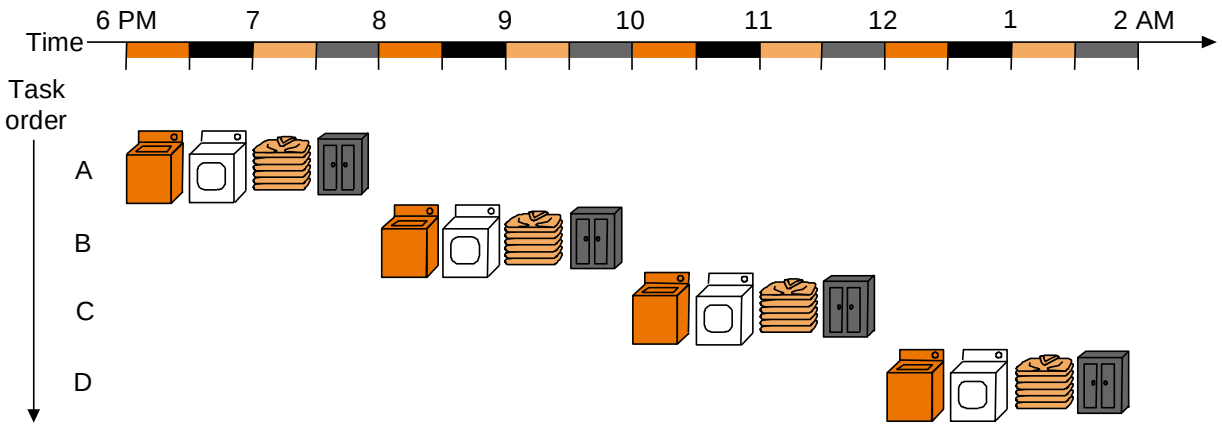
Computer Architecture (Spring 2020)

Pipelining

Dr. Duo Liu (刘铎)
Office: Main Building 0626
Email: liuduo@cqu.edu.cn

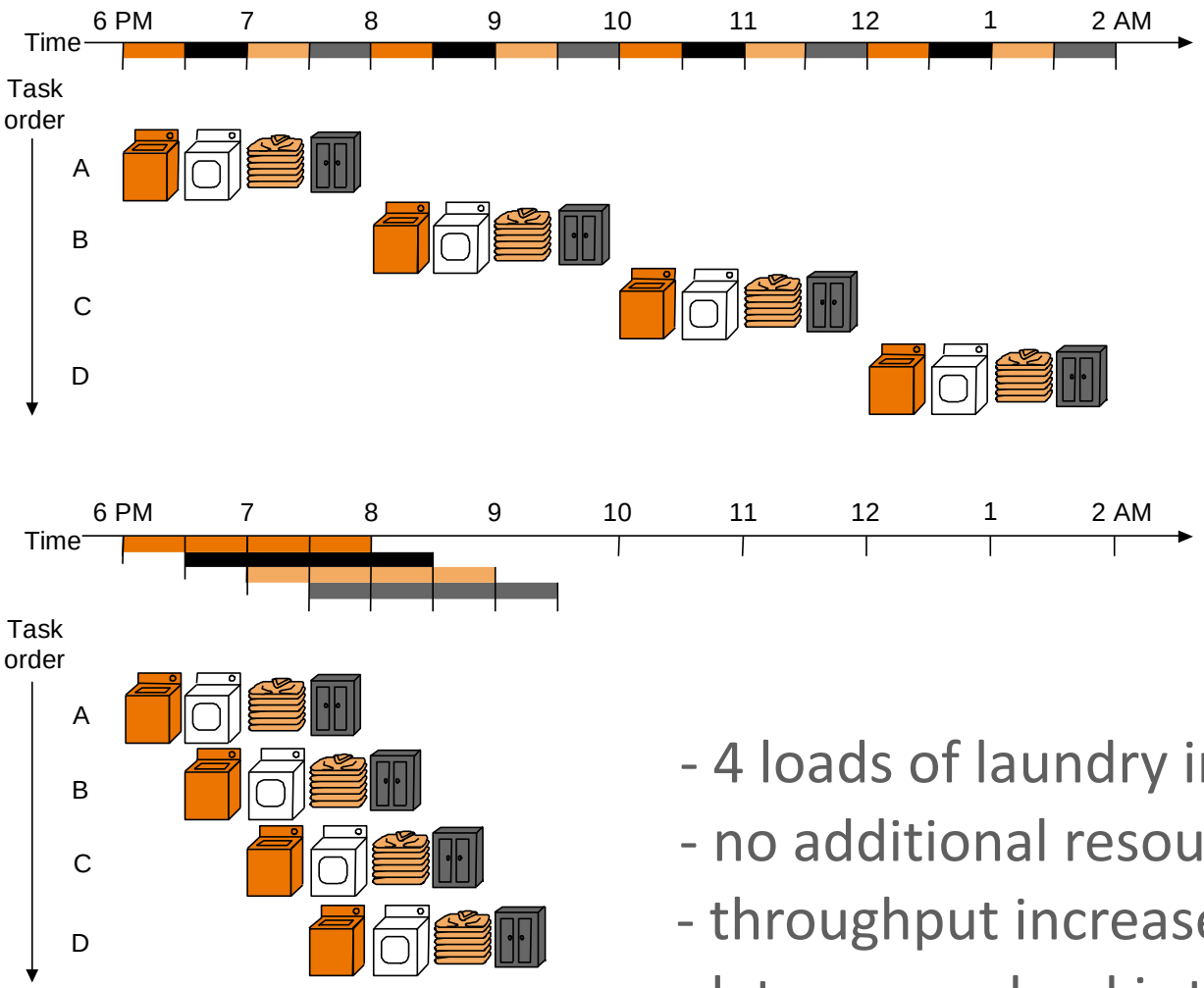
The Laundry Analogy



- "place one dirty load of clothes in the washer"
- "when the washer is finished, place the wet load in the dryer"
- "when the dryer is finished, take out the dry load and fold"
- "when folding is finished, ask your roommate (??) to put the clothes away"

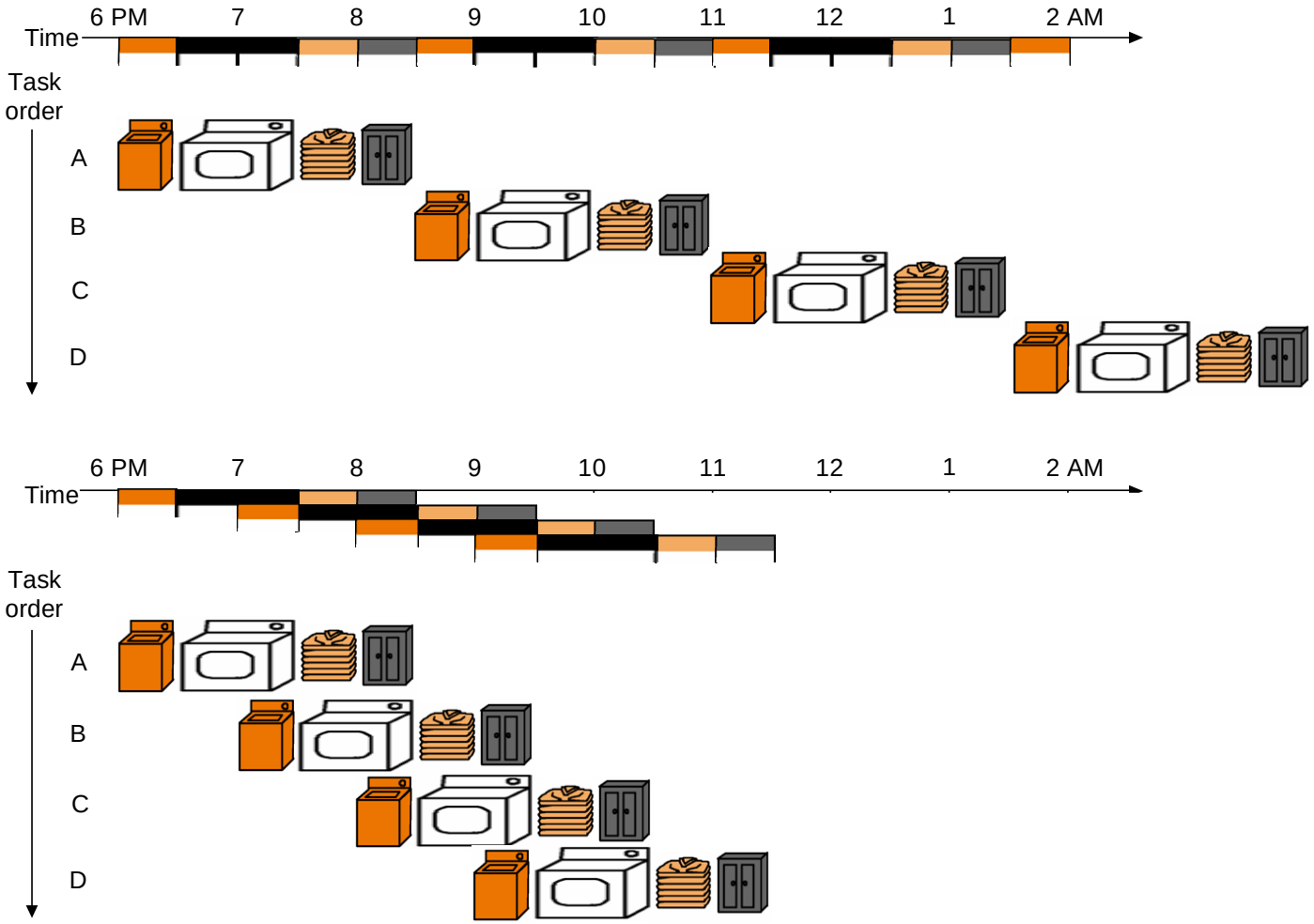
  - steps to do a load are sequentially dependent
  - no dependence between different loads
  - different steps do not share resources

Pipelining Multiple Loads of Laundry



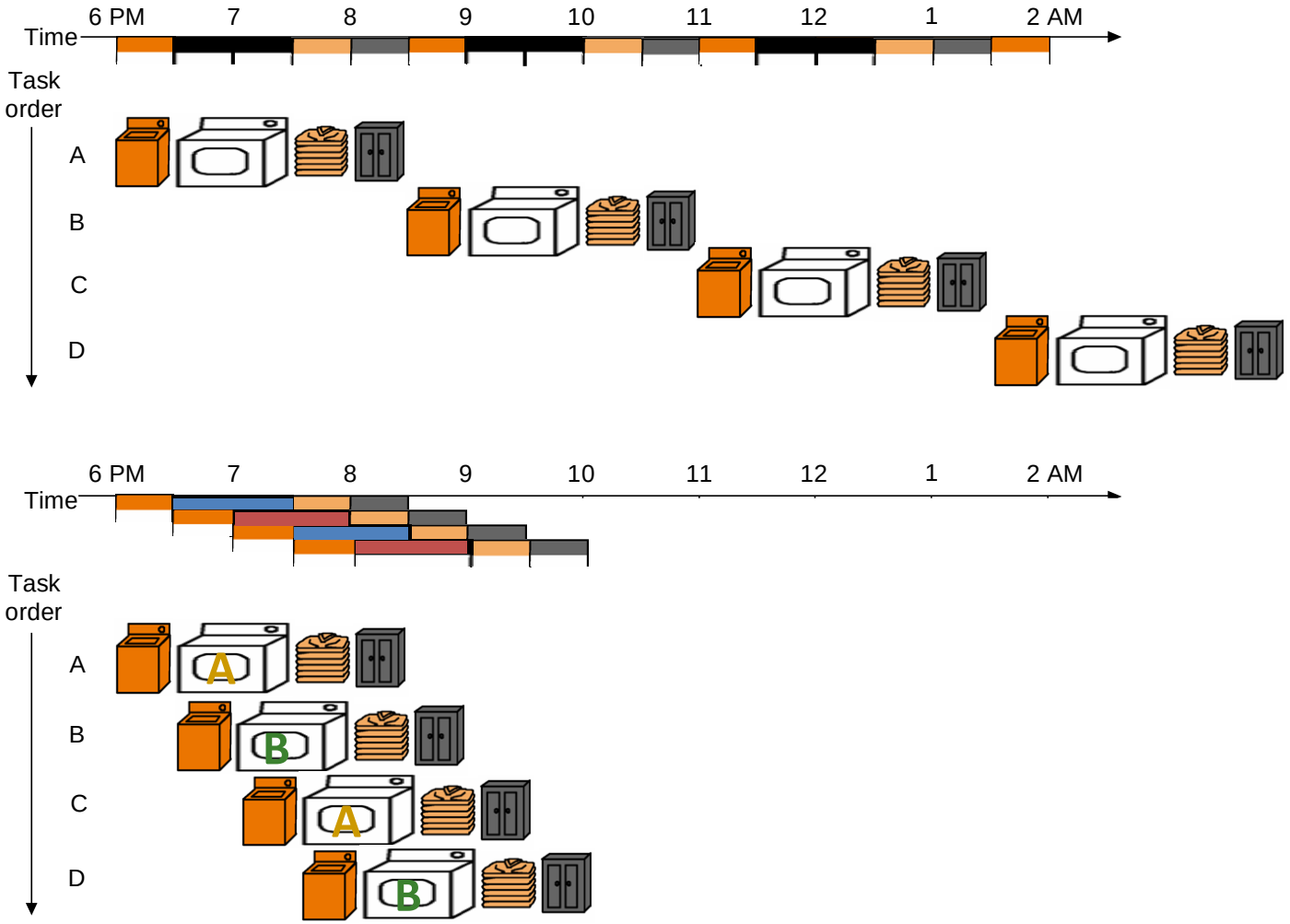
- 4 loads of laundry in parallel
- no additional resources
- throughput increased by 4
- latency per load is the same

Pipelining Multiple Loads of Laundry: In Practice



the slowest step decides throughput

Pipelining Multiple Loads of Laundry: In Practice

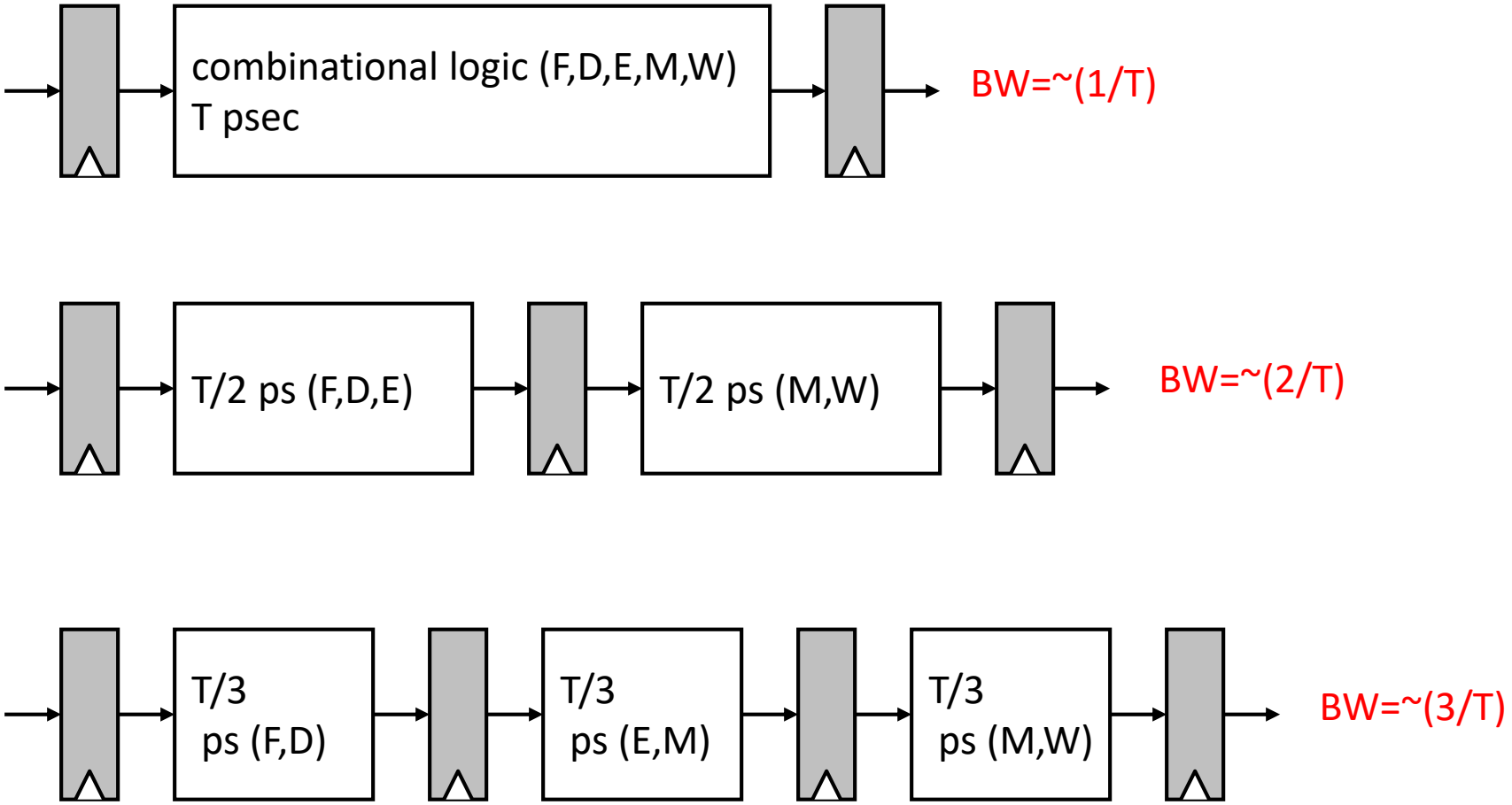


Throughput restored (2 loads per hour) using 2 dryers

An Ideal Pipeline

- Goal: Increase throughput with little increase in cost (hardware cost, in case of instruction processing)
- Repetition of identical operations
  - The same operation is repeated on a large number of different inputs
- Repetition of independent operations
  - No dependencies between repeated operations
- Uniformly partitionable suboperations
  - Processing can be evenly divided into uniform-latency suboperations (that do not share resources)
- Fitting examples: automobile assembly line, doing laundry
  - What about the instruction processing "cycle"?

Ideal Pipelining



More Realistic Pipeline: Throughput

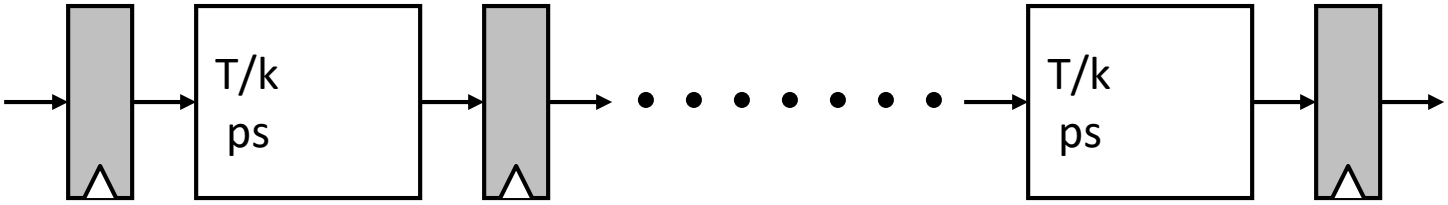
- Nonpipelined version with delay T

  BW = 1/(T+S) where S = latch delay

- k-stage pipelined version

  $BW_{k\text{-stage}} = 1 / (T/k + S)$

  $BW_{max} = 1 / (1 \text{ gate delay} + S)$



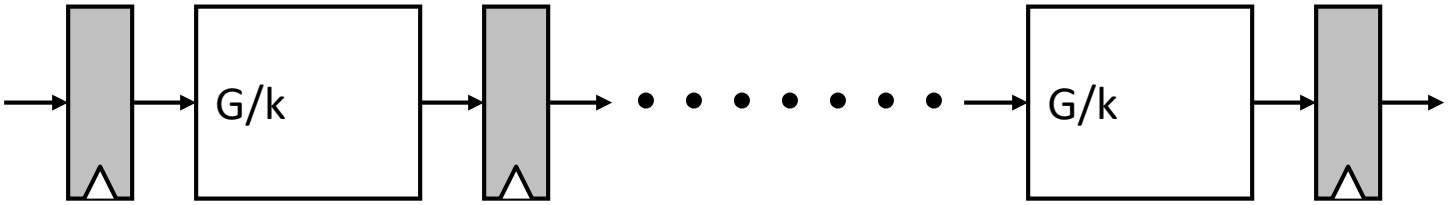
More Realistic Pipeline: Cost

- Nonpipelined version with combinational cost G

  Cost = G+L where L = latch cost

- k-stage pipelined version

  $Cost_{k\text{-}stage} = G + Lk$

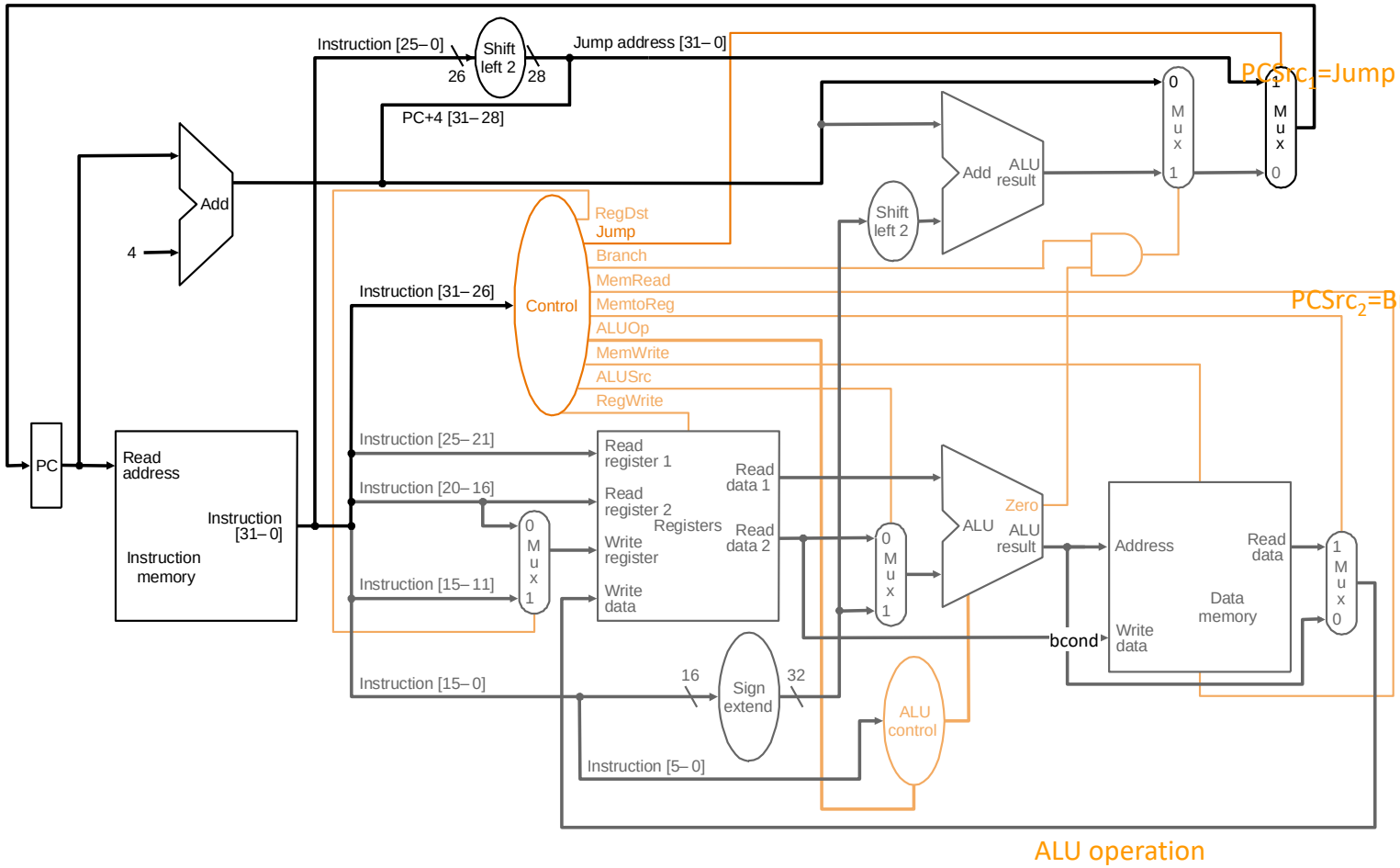


Pipelining Instruction Processing

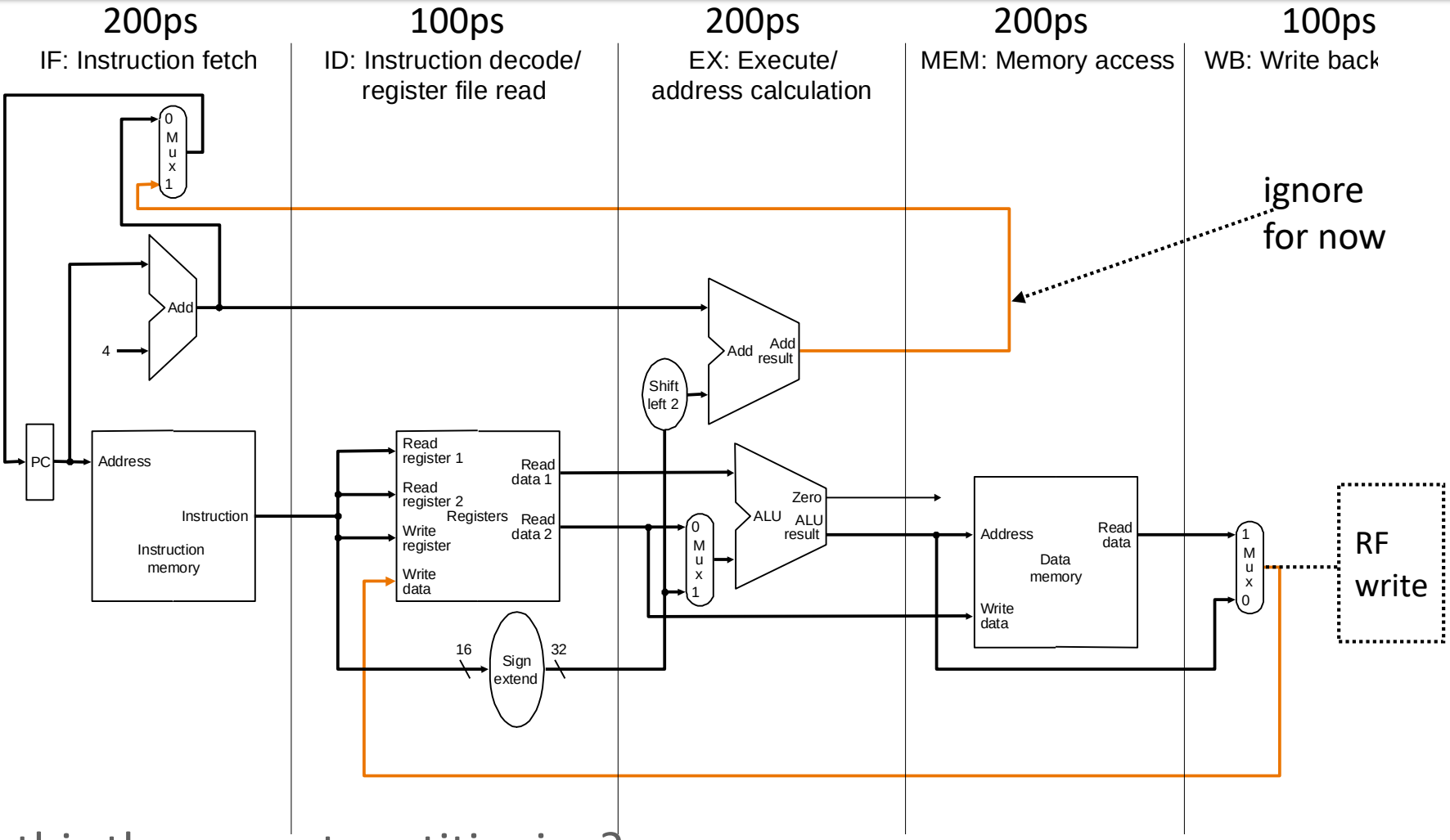
Remember: The Instruction Processing Cycle

1. Instruction fetch (IF)
2. Instruction decode and
   register operand fetch (ID/RF)
3. Execute/Evaluate memory address (EX/AG)
4. Memory operand fetch (MEM)
5. Store/writeback result (WB)

Remember the Single-Cycle Uarch



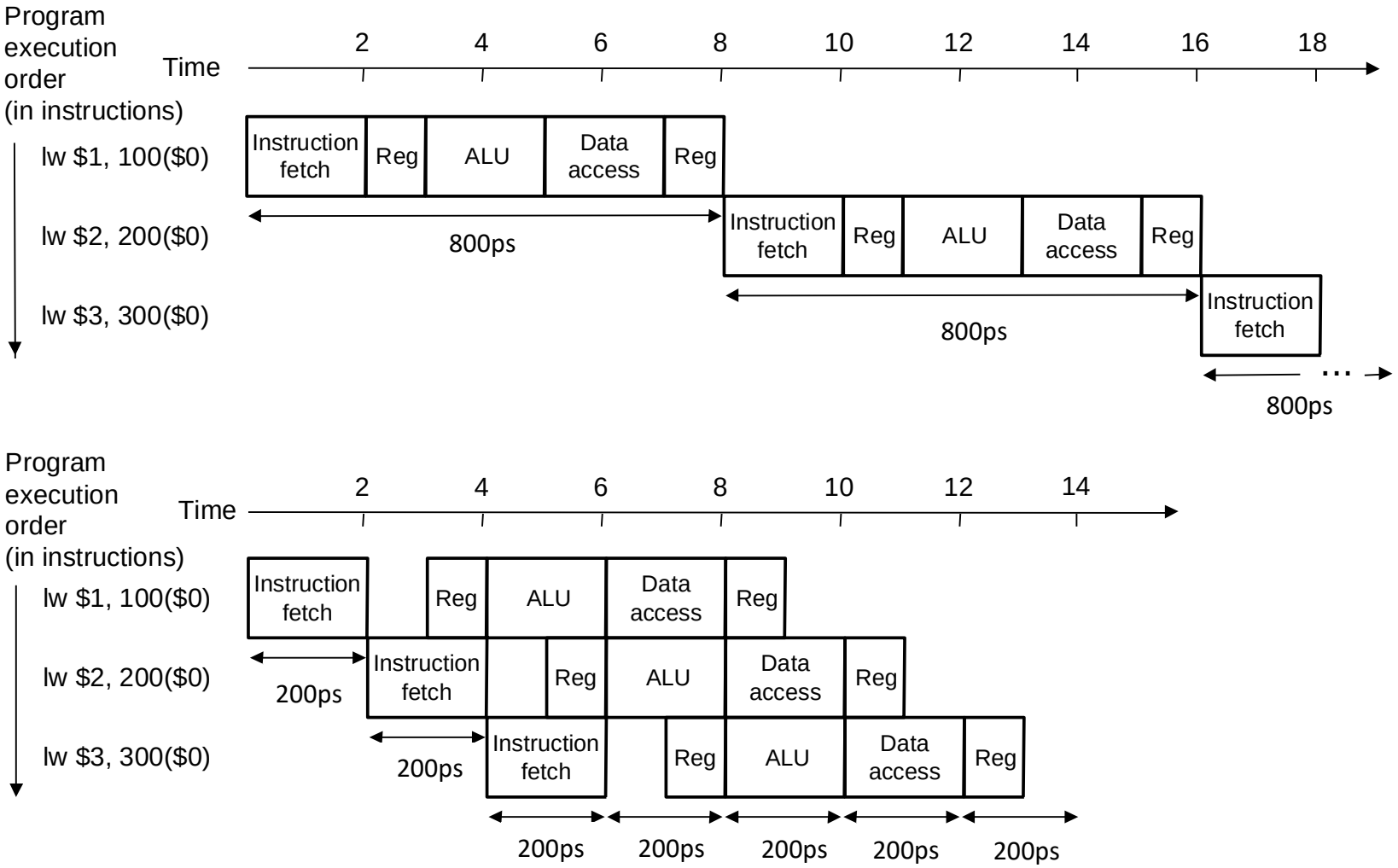
Dividing Into Stages



Is this the correct partitioning?

Why not 4 or 6 stages? Why not different boundaries?

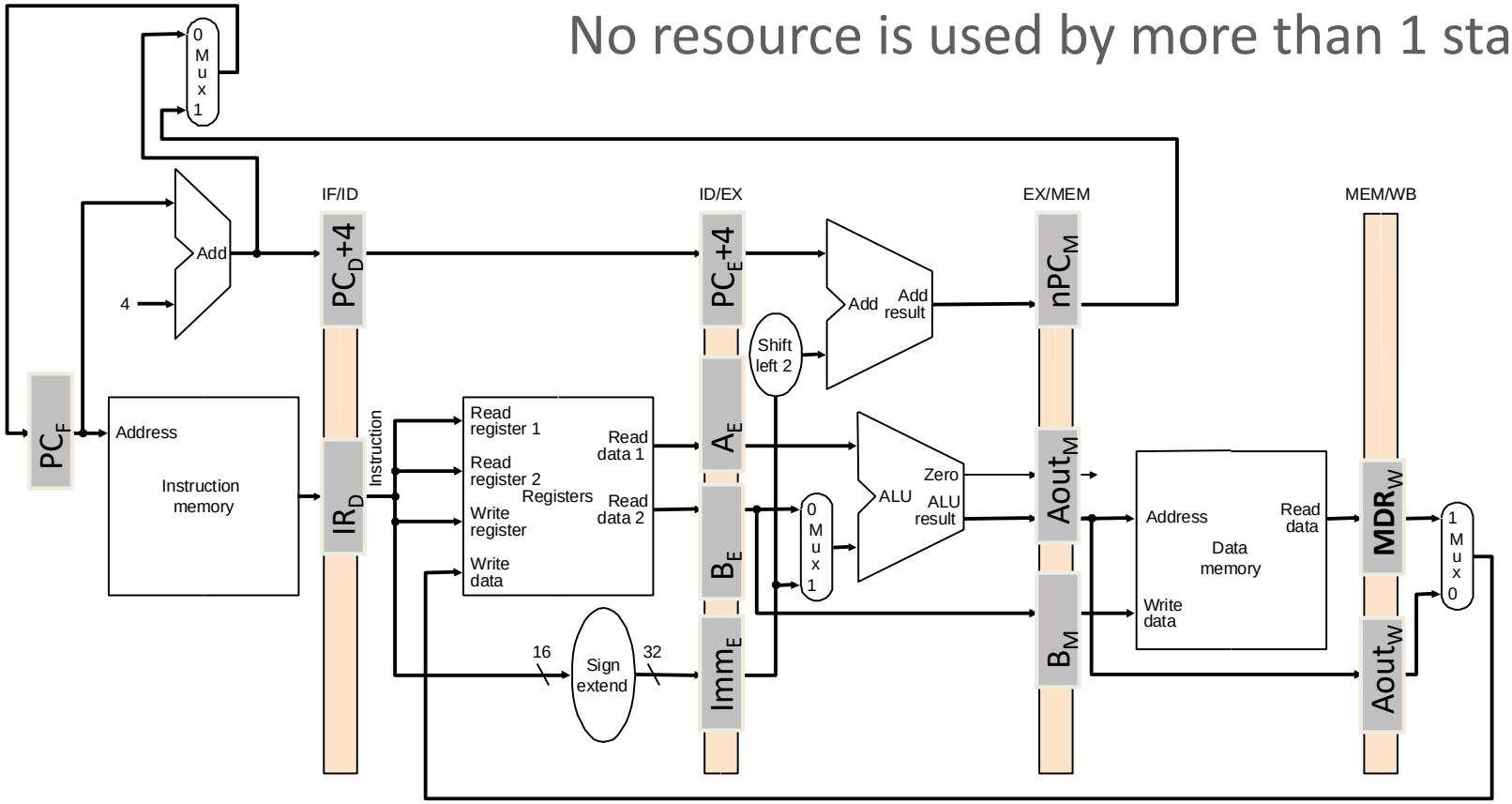
Instruction Pipeline Throughput

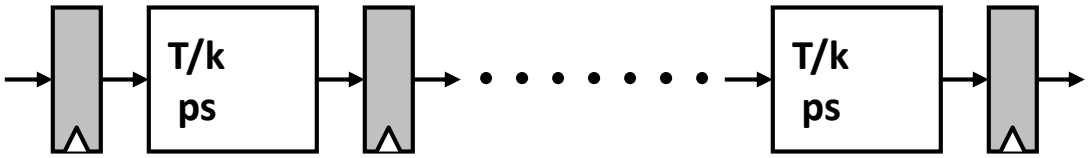


5-stage speedup is 4, not 5 as predicated by the ideal model. Why?

Enabling Pipelined Processing: Pipeline Registers

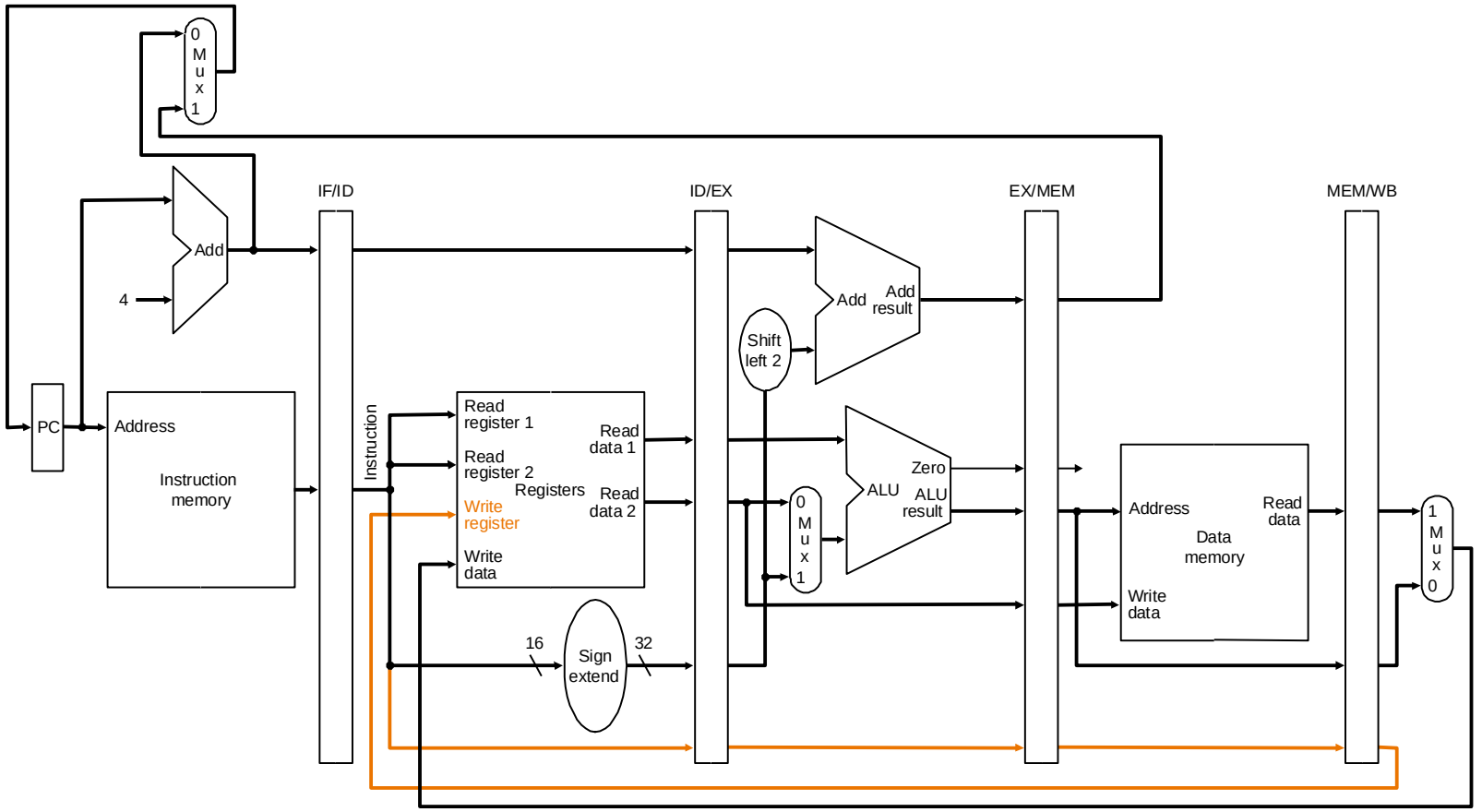
No resource is used by more than 1 stage!



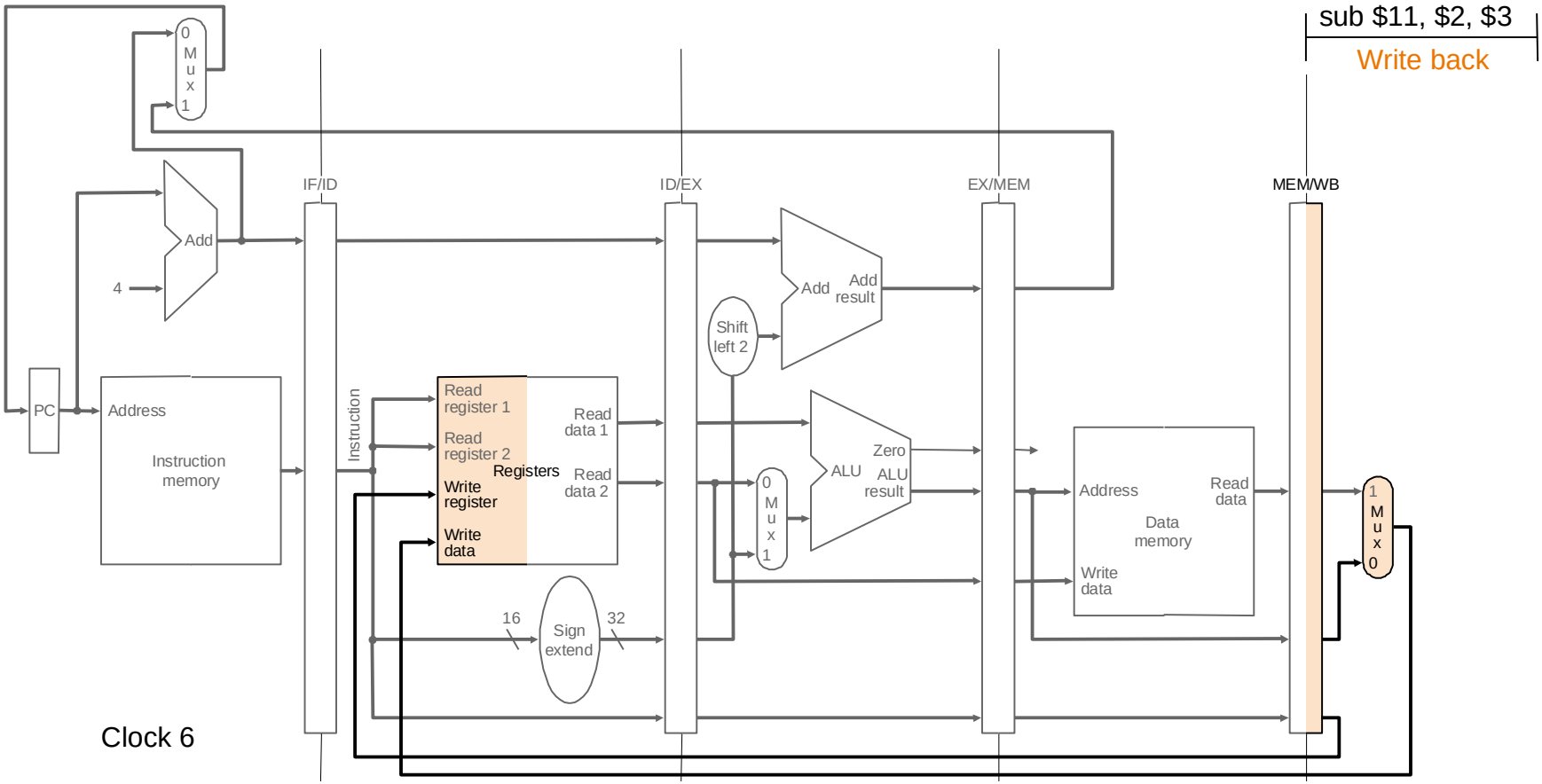


Pipelined Operation Example

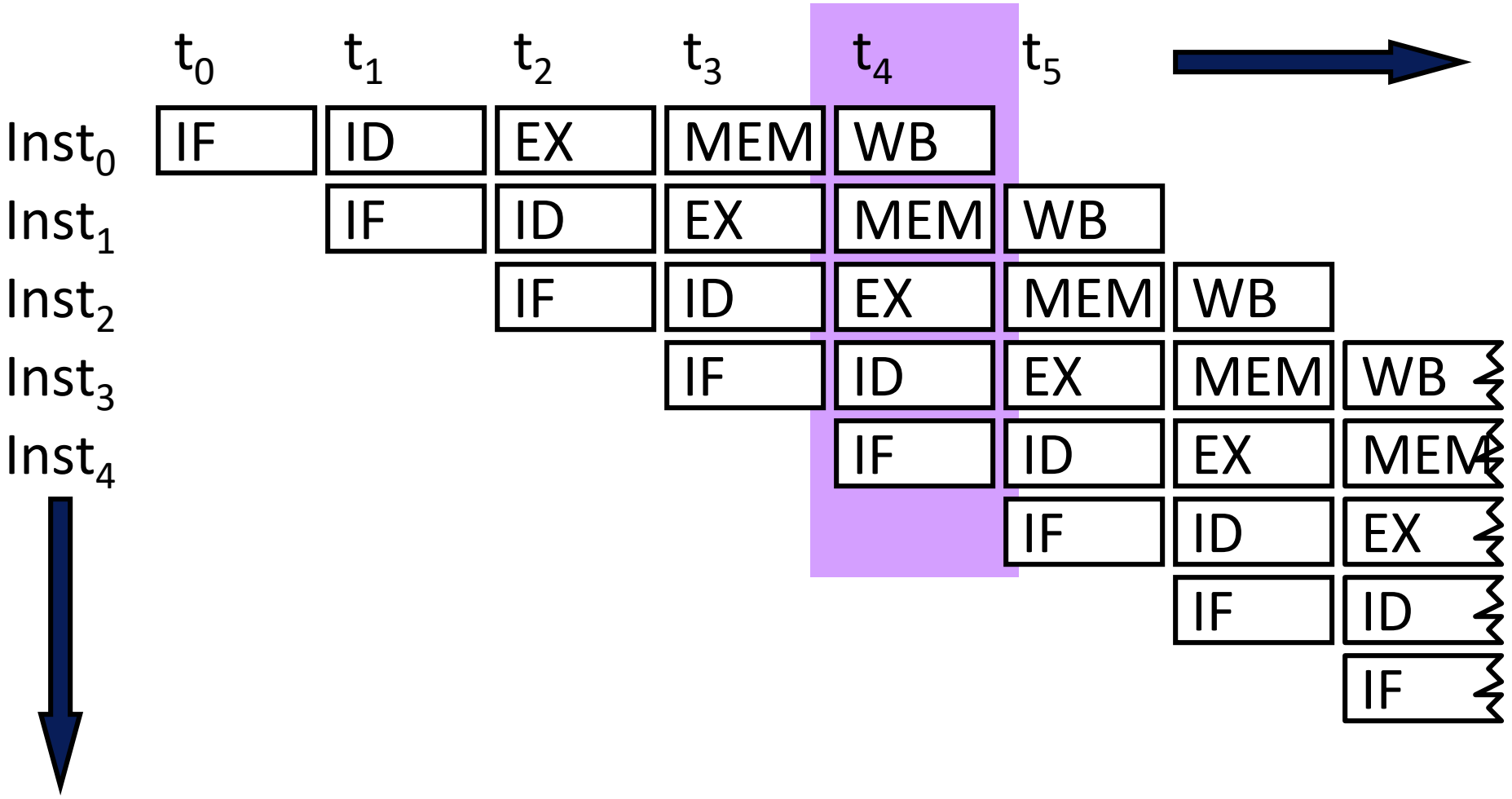
All instruction classes must follow the same path and timing through the pipeline stages. Any performance impact?



Pipelined Operation Example



Illustrating Pipeline Operation: Operation View



Illustrating Pipeline Operation: Resource View

	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}
IF	I_0	I_1	I_2	I_3	I_4	I_5	I_6	I_7	I_8	I_9	I_{10}
ID		I_0	I_1	I_2	I_3	I_4	I_5	I_6	I_7	I_8	I_9
EX			I_0	I_1	I_2	I_3	I_4	I_5	I_6	I_7	I_8
MEM				I_0	I_1	I_2	I_3	I_4	I_5	I_6	I_7
WB					I_0	I_1	I_2	I_3	I_4	I_5	I_6

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
- Potential speedup = Number pipe stages
- Unbalanced lengths of pipe stages reduces speedup
- Time to "fill" pipeline and time to "drain" it reduces speedup

Other Definitions

- Pipe stage or pipe segment
  - A decomposable unit of the fetch-decode-execute paradigm
- Pipeline depth
  - Number of stages in a pipeline
- Machine cycle
  - Clock cycle time
- Latch
  - Per phase/stage local information storage unit

Design Issues

- Balance the length of each pipeline stage

$$\text{Throughput} = \frac{\text{Depth of the pipeline}}{\text{Time per instruction on unpipelined machine}}$$

- Problems
  - Usually, stages are not balanced
  - Pipelining overhead
  - Hazards (conflicts)

- Performance (throughput ⟶ CPU performance equation)
  - Decrease of the CPI
  - Decrease of cycle time

1st and 2nd Instruction cycles

- Instruction fetch (IF)

  IR ← Mem[PC];

  NPC ← PC + 4

- Instruction decode & register fetch (ID)

  A ← Regs[$IR_{6..10}$];

  B ← Regs[$IR_{11..15}$];

  Imm ← ((IR_{16})^{16} # # $IR_{16..31}$)

3rd Instruction cycle

- Execution & effective address (EX)
  - Memory reference
    - ALUOutput ← A + Imm
  - Register - Register ALU instruction
    - ALUOutput ← A *func* B
  - Register - Immediate ALU instruction
    - ALUOutput ← A *op* Imm
  - Branch
    - ALUOutput ← NPC + Imm; Cond ← (A *op* 0)

4th Instruction cycle

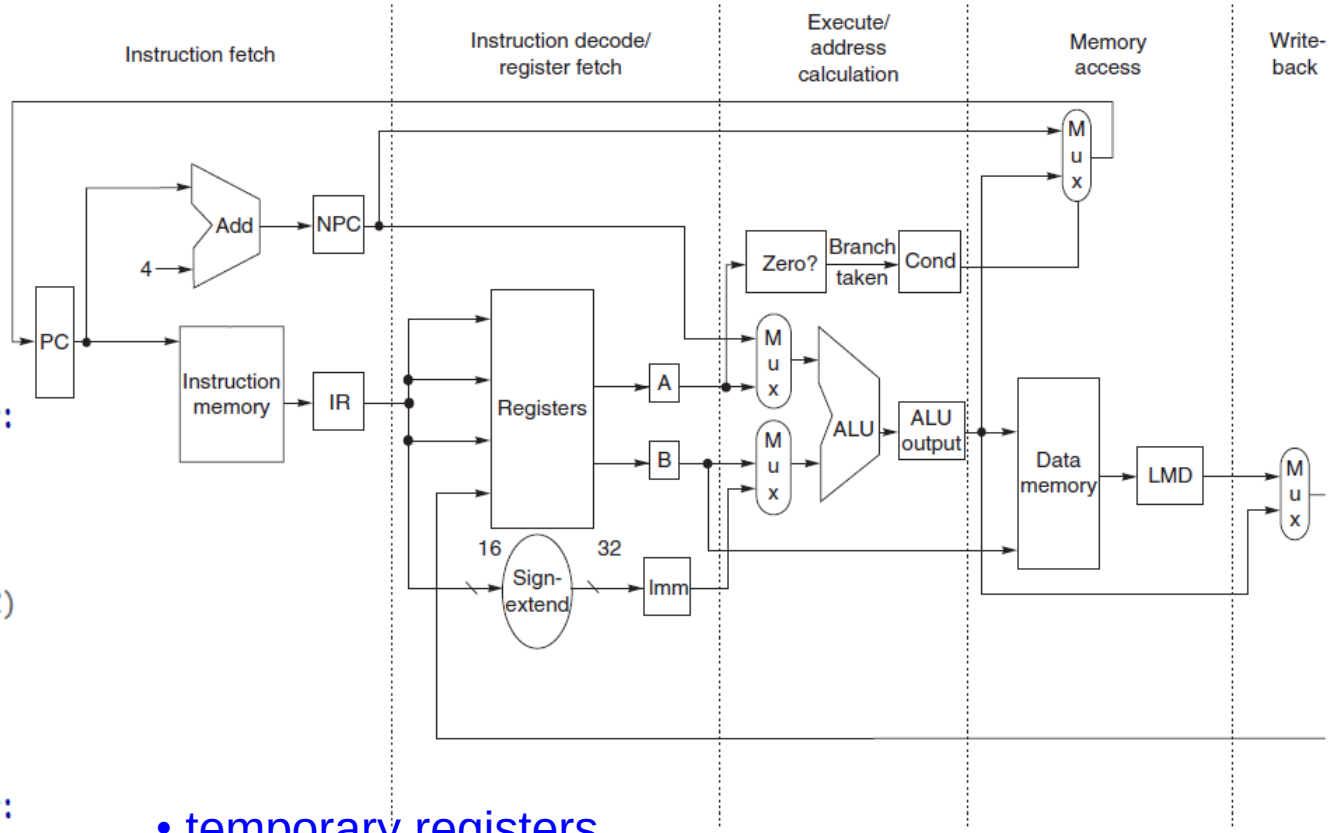
- Memory access & branch completion (MEM)
  - Memory reference
    - PC ← NPC
    - LMD ← Mem[ALUOutput] (load)
    - Mem[ALUOutput] ← B (store)
  - Branch
    - if (cond) PC ← ALUOutput; else PC ← NPC

5th Instruction cycle

- Write-back (WB)
  - Register - register ALU instruction
    - $Regs[IR_{16..20}] \leftarrow ALUOutput$
  - Register - immediate ALU instruction
    - $Regs[IR_{11..15}] \leftarrow ALUOutput$
  - Load instruction
    - $Regs[IR_{11..15}] \leftarrow LMD$

Simple Implementation of a MIPS

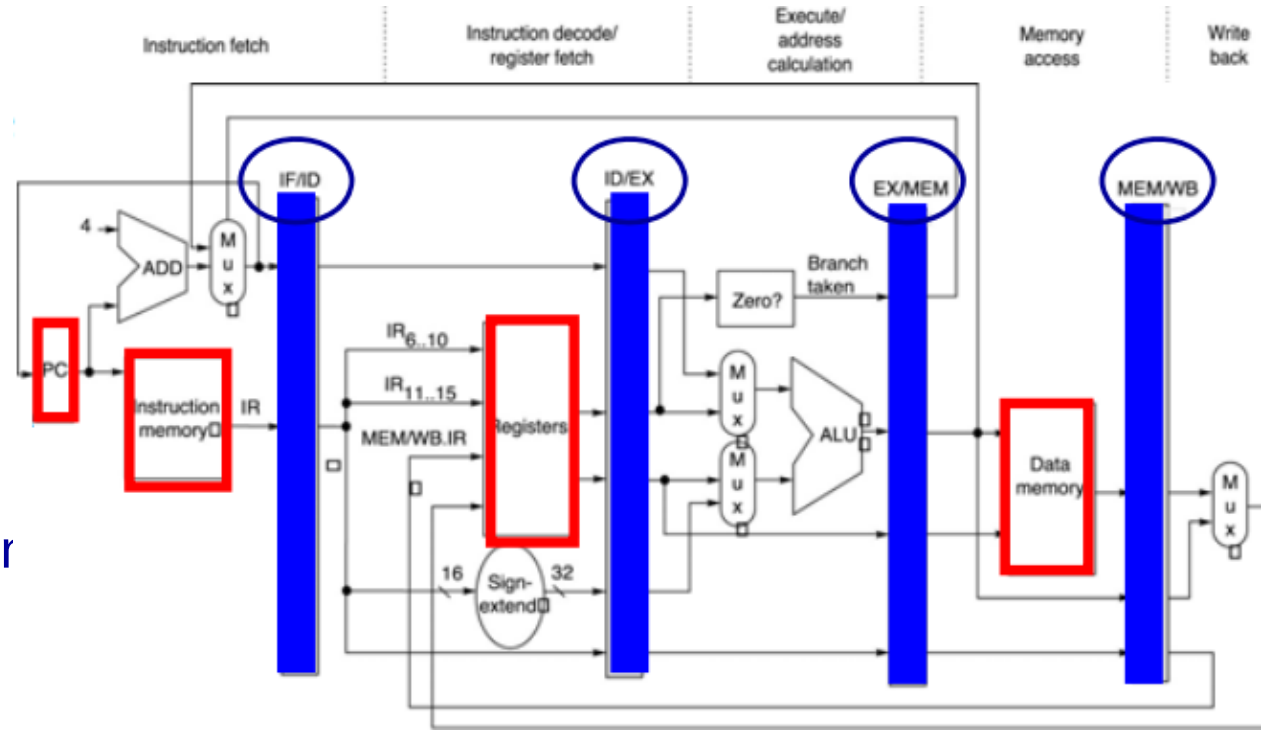
1. IR ← Mem[PC];
   NPC ← PC+4;
2. A ← Regs[rs];
   B ← Regs[rt];
   Imm ← signExt(IR16)
   - fixed-field decoding
3. one of the followings:
   - ALUOutput ← A + Imm
   - ALUOutput ← func (A, B)
   - ALUOutput ← A op Imm
   - Cond ← (A == zero) *and* ALUOutput ← NPC + (Imm <<2)
4. PC ← MUX(Cond, ALUOutput, NPC)
   - LMD ← Mem[ALUOutput] or
   - Mem[ALUOutput] ← B
5. one of the followings:
   - Regs[rt] ← LMD
   - Regs[rd] ← ALUOutput
   - Regs[rt] ← ALUOutput



- temporary registers
  - hold values between clock cycles for an instruction
- state elements ("visible part of the state")
  - hold values between successive instructions
- control logic (FSM or microcode controller)
  - (not illustrated in the diagram above)

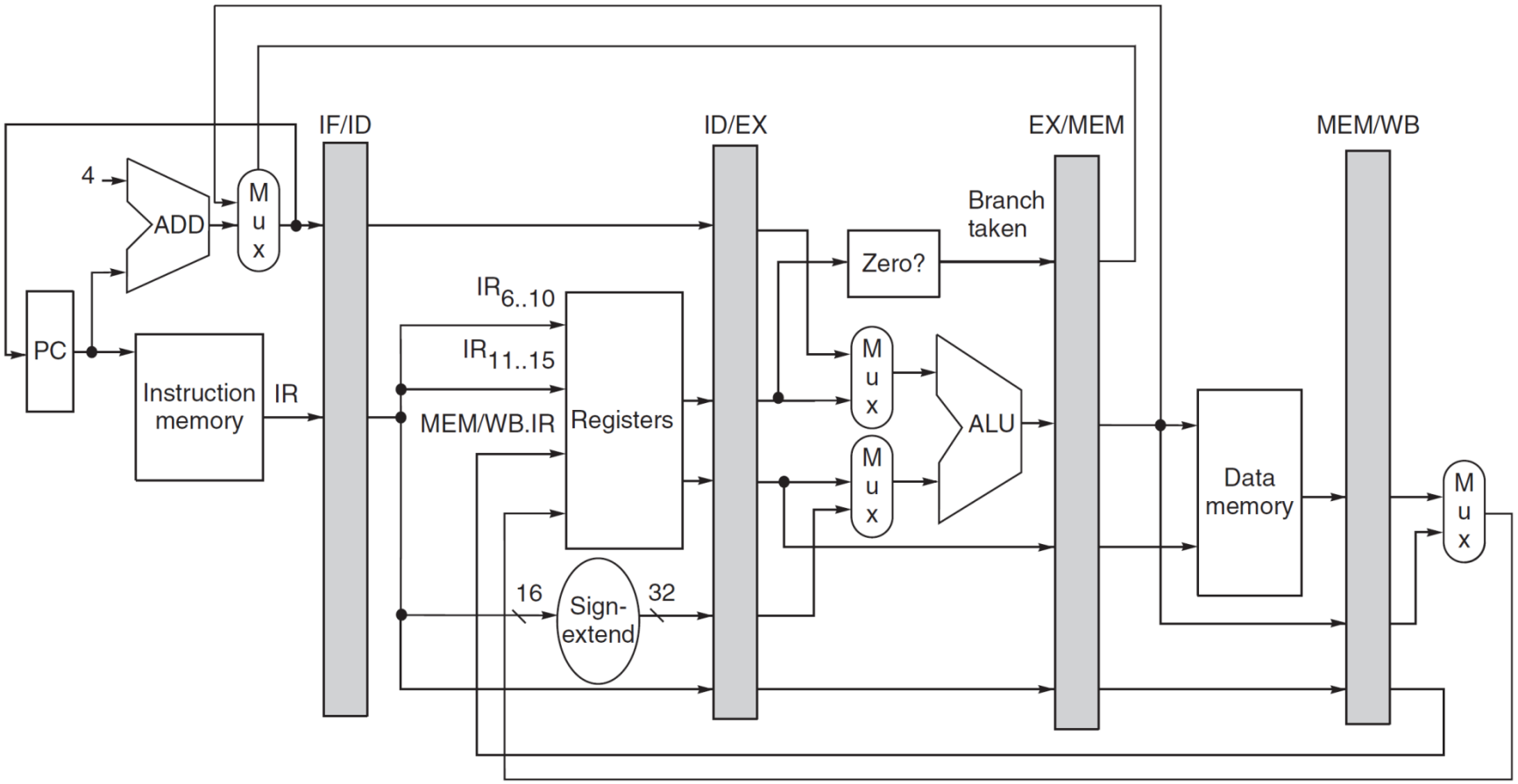
MIPS Pipeline Implementation

- Apparently easy
  - each clock cycle becomes a pipe stage
  - temporary registers become pipe registers
  - new instruction issued at each clock cycle
- Result propagatior
  - register value to be stored is read during ID and used in MEM
  - ALU result computed during EX (or loaded during MEM) and store in WB



- pipeline registers
  - hold values between clock cycles for an instruction
  - prevent interference (edge-triggered flip-flops)
- state elements (“visible part of the state”)
  - hold values between successive instructions

MIPS Pipeline Implementation



Multiple-Clock Cycle Pipeline Diagram

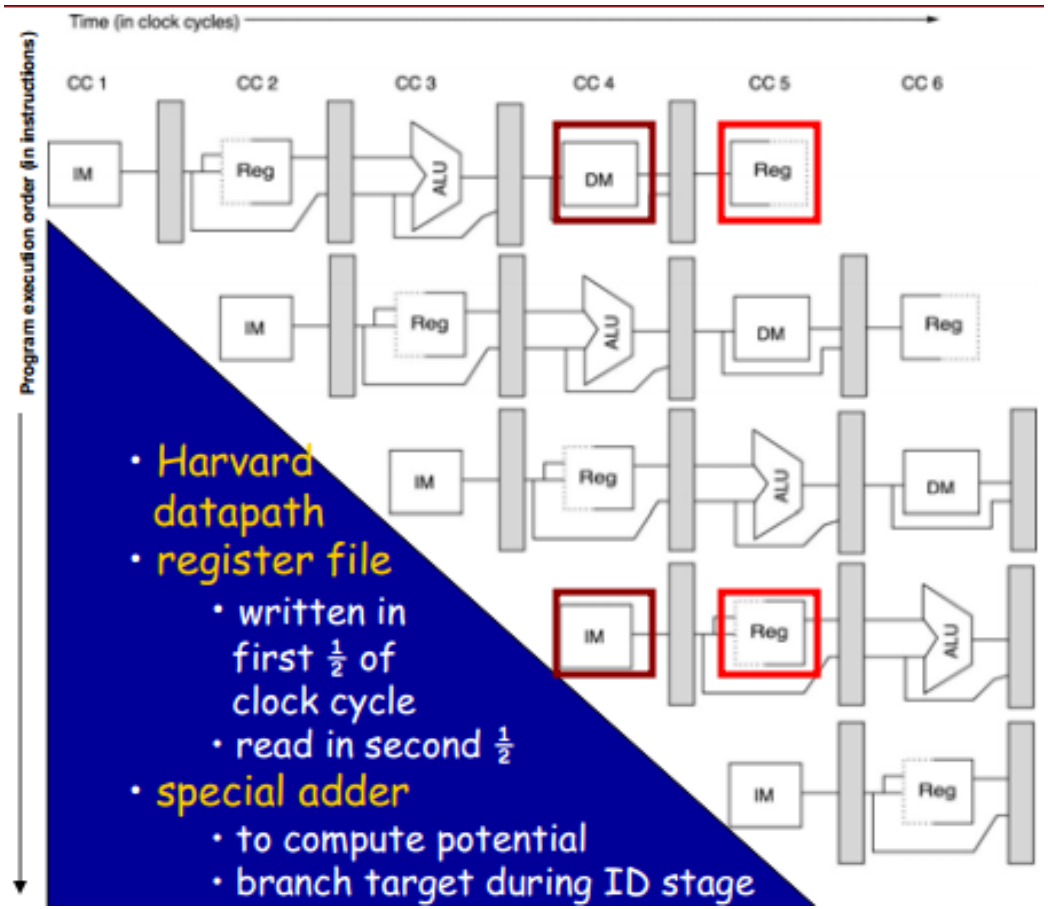
lw R10, 20(R1)

sub R11, R2, R3

add R12, R3, R4

lw R13, 24(R1)

add R14, R5, R6



Unlike some other speedup techniques, pipelining is fundamentally transparent to the programmer