Computer Architecture (Fall 2022)

Pipelining

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

Can We Do Better?

- What limitations do you see with the multi-cycle design?

- Limited concurrency
  - Some hardware resources are idle during different phases of instruction processing cycle
  - “Fetch” logic is idle when an instruction is being “decoded” or “executed”
  - Most of the datapath is idle when a memory access is happening

Can We Use the Idle Hardware to Improve Concurrency?

- Goal: Concurrency → throughput (more “work” completed in one cycle)

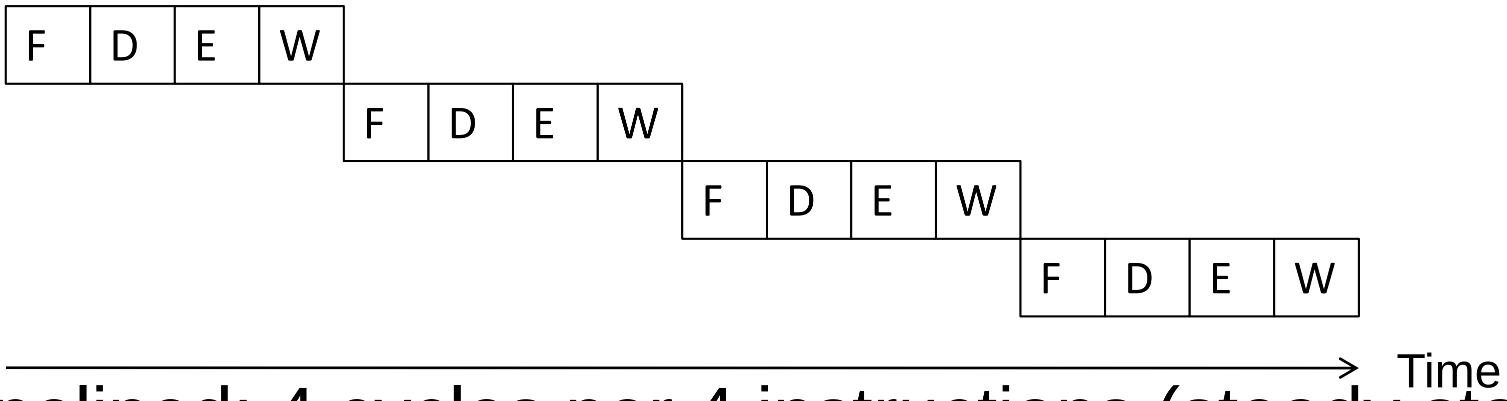
- Idea: When an instruction is using some resources in its processing phase, process other instructions on idle resources not needed by that instruction
  - E.g., when an instruction is being decoded, fetch the next instruction
  - E.g., when an instruction is being executed, decode another instruction
  - E.g., when an instruction is accessing data memory (ld/st), execute the next instruction
  - E.g., when an instruction is writing its result into the register file, access data memory for the next instruction

Pipelining: Basic Idea

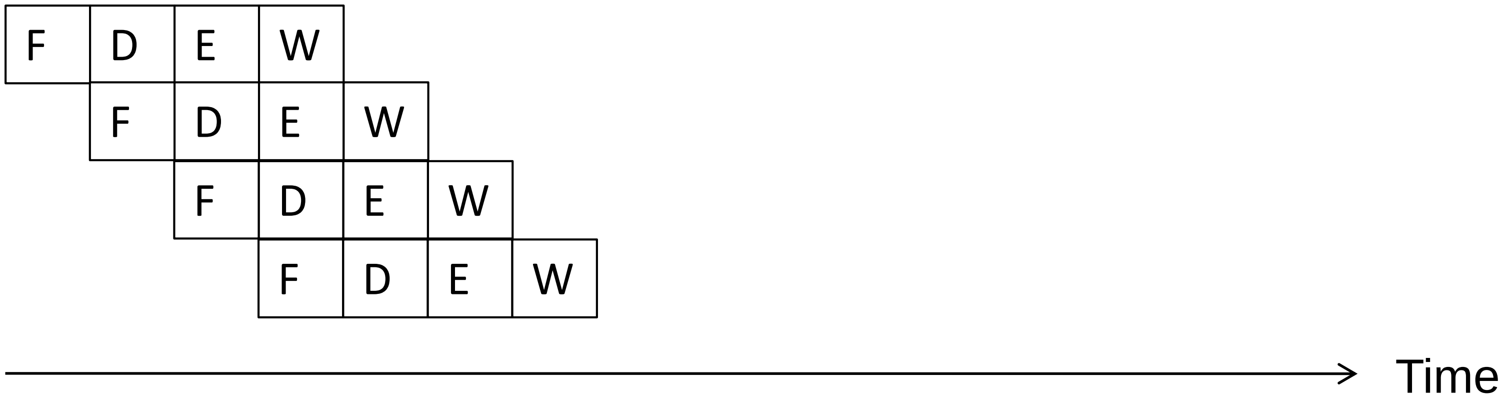
- More systematically:
  - Pipeline the execution of multiple instructions
  - Analogy: “Assembly line processing” of instructions
- Idea:
  - Divide the instruction processing cycle into distinct “stages” of processing
  - Ensure there are enough hardware resources to process one instruction in each stage
  - Process a different instruction in each stage
    - Instructions consecutive in program order are processed in consecutive stages
- Benefit: Increases instruction processing throughput (1/CPI)
- Downside: Start thinking about this…

Example: Execution of Four Independent ADDs

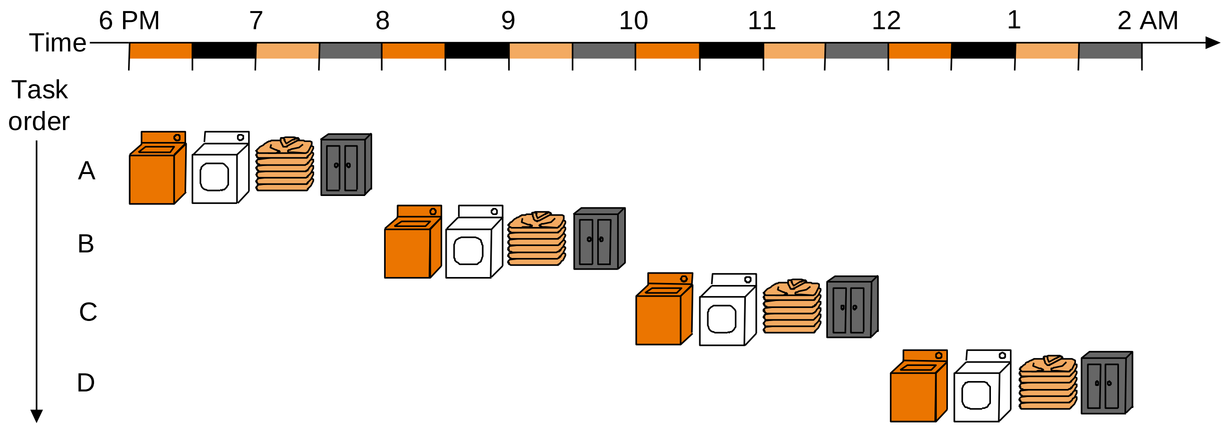
- Multi-cycle: 4 cycles per instruction



- Pipelined: 4 cycles per 4 instructions (steady state)

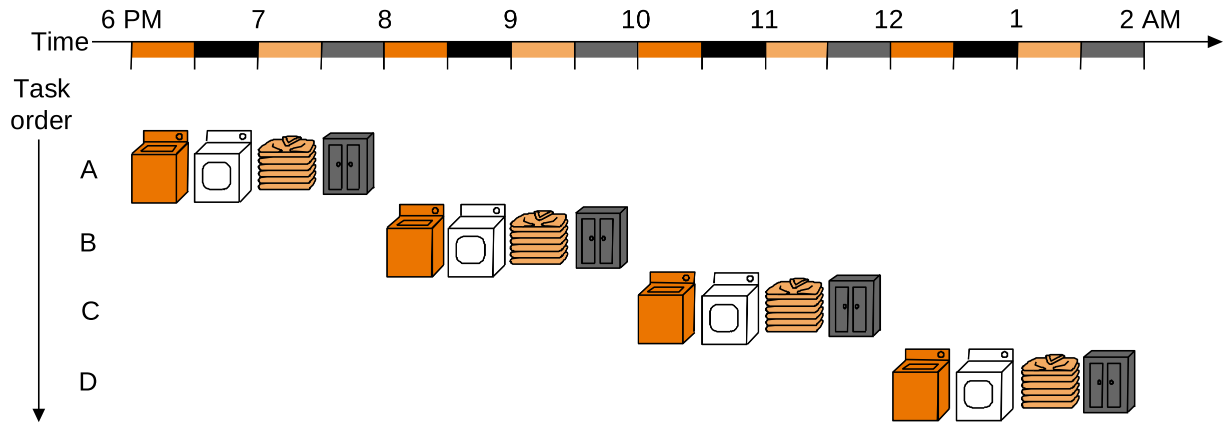


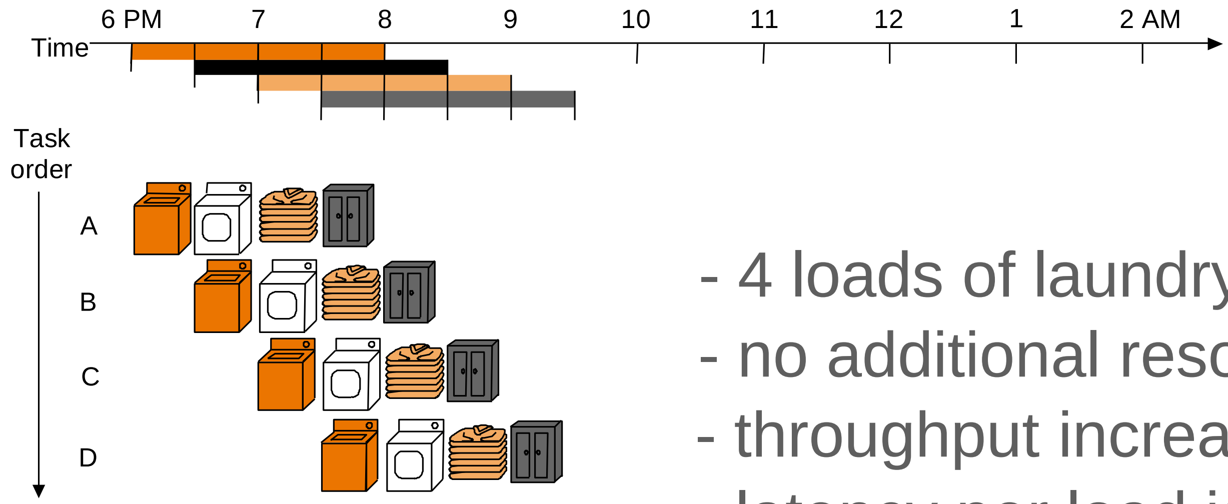
The Laundry Analogy



- “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

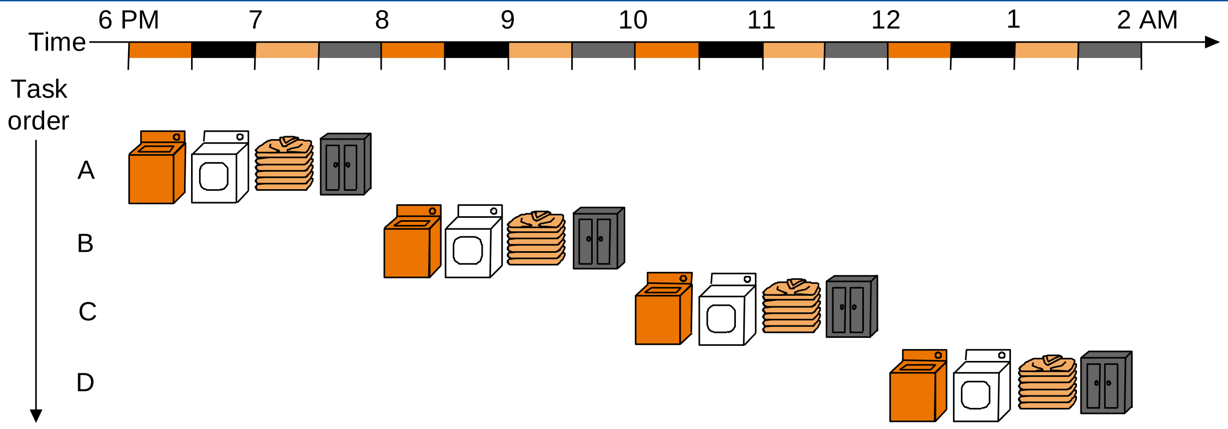
Pipelining Multiple Loads of Laundry

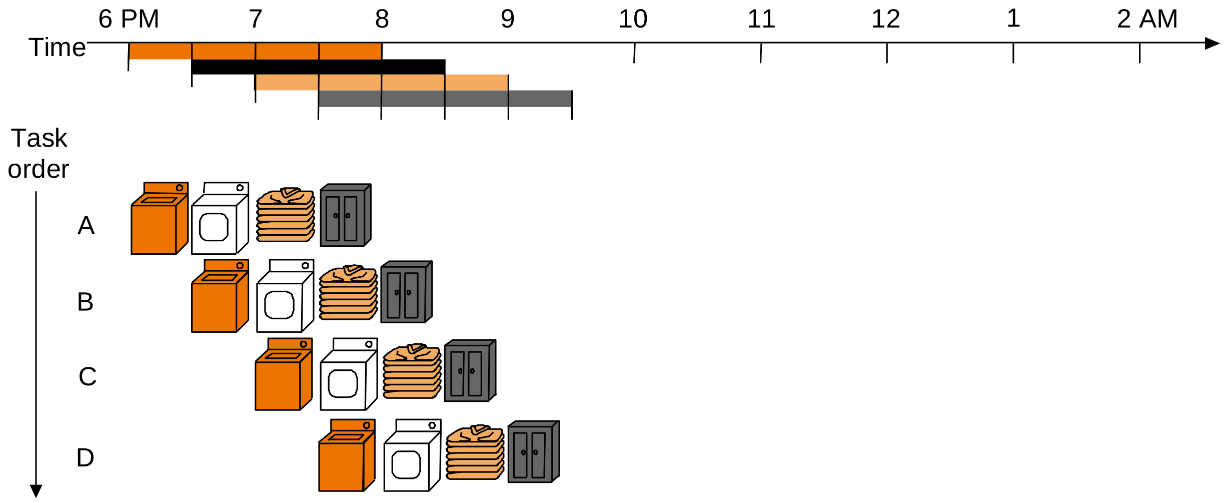




- 4 loads of laundry in parallel
- no additional resources
- throughput increased by 4
- latency per load is the same

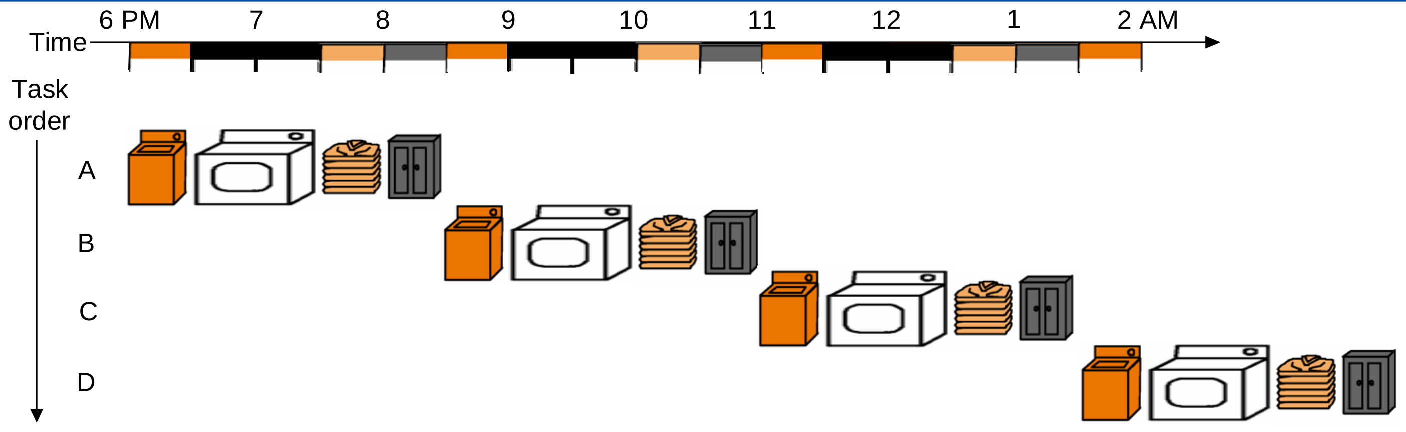
Pipelining Multiple Loads of Laundry: In Practice

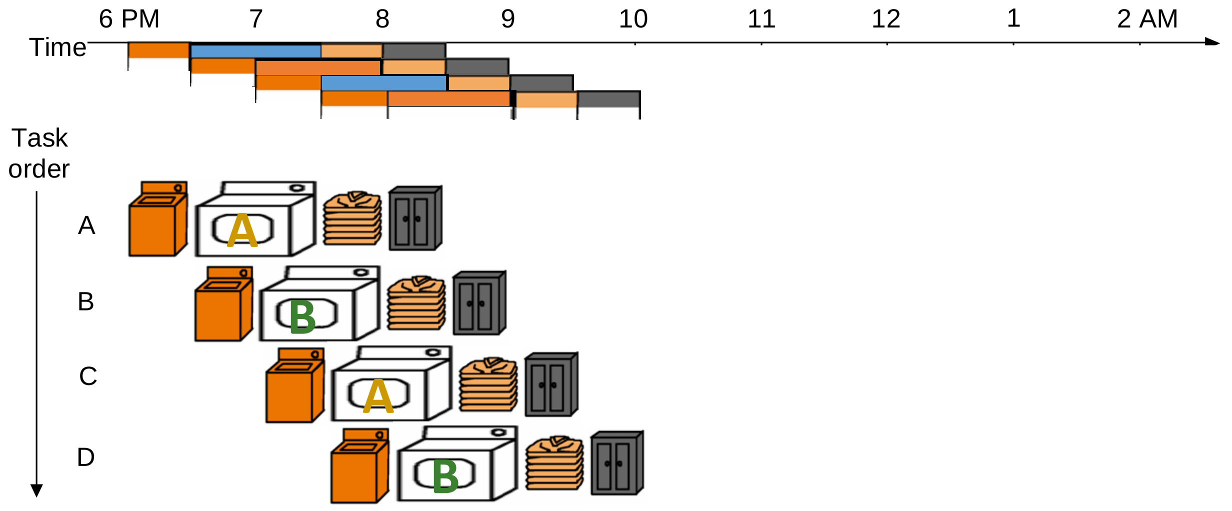




the slowest step decides throughput

Pipelining Multiple Loads of Laundry: In Practice



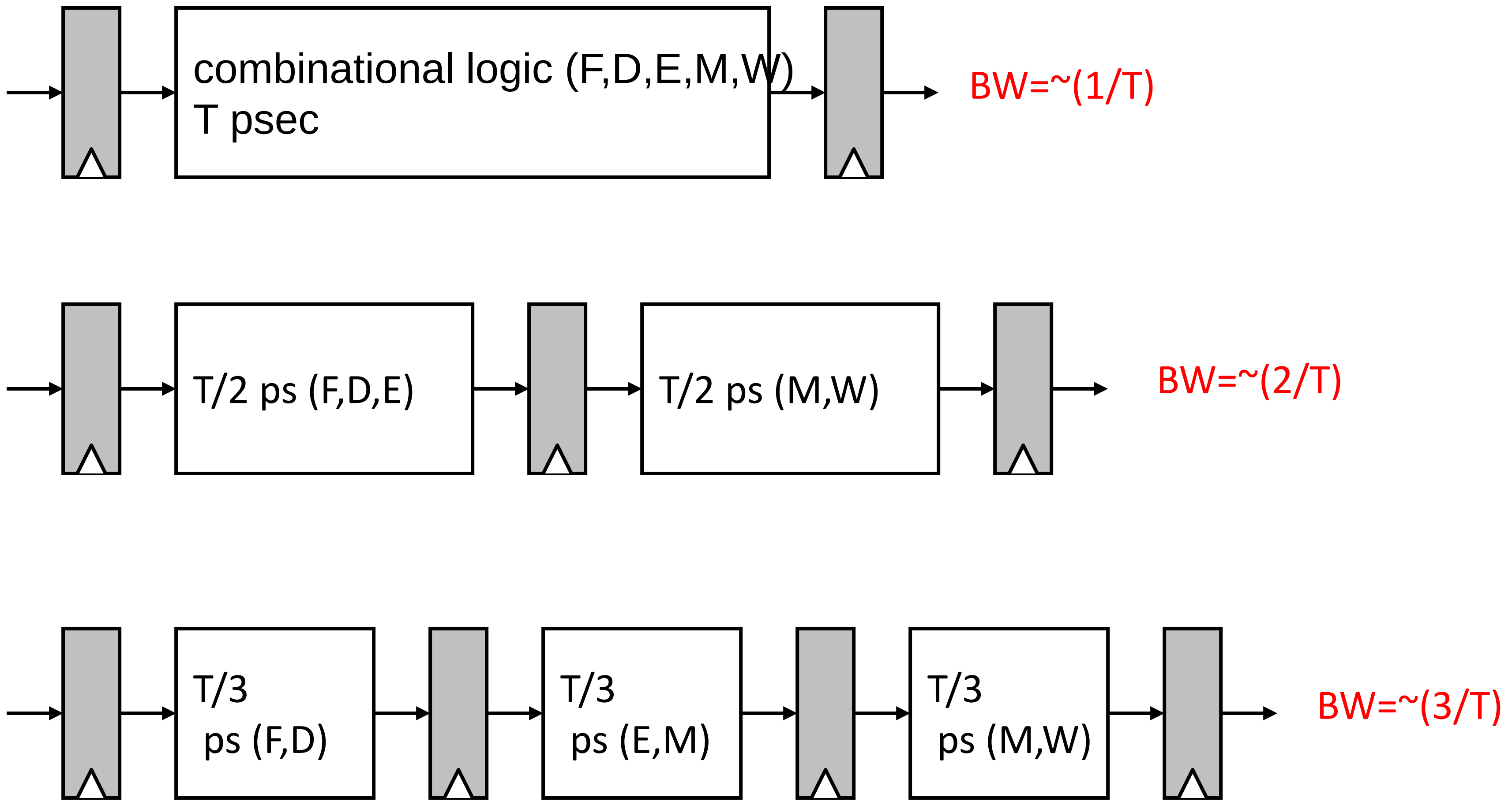


Throughput restored (2 loads per hour) using 2 dryers

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”?

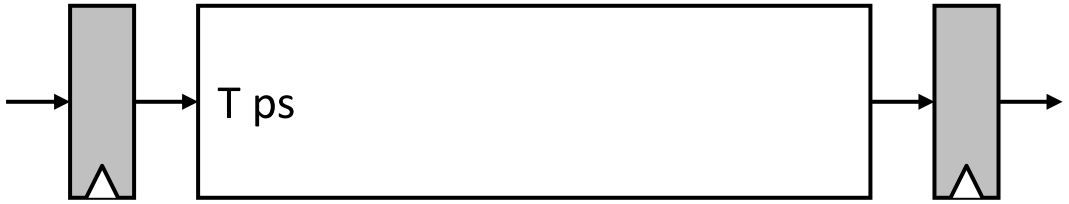
Ideal Pipelining



More Realistic Pipeline: Throughput

- Nonpipelined version with delay T

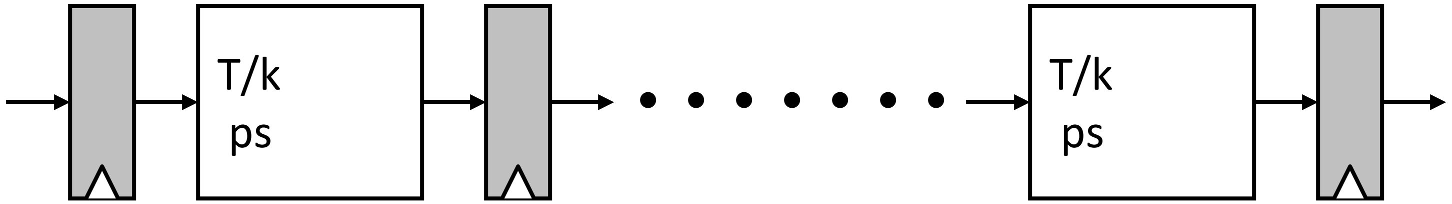
$$BW = 1/(T+S) \text{ where } S = \text{latch delay}$$



- k-stage pipelined version

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

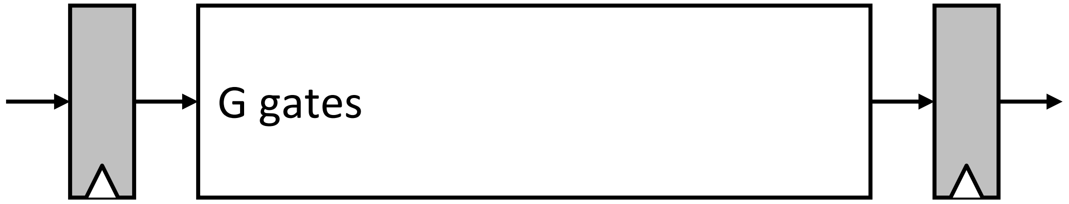
$$BW_{max} = 1 / (1 \text{ gate delay} + S)$$



More Realistic Pipeline: Cost

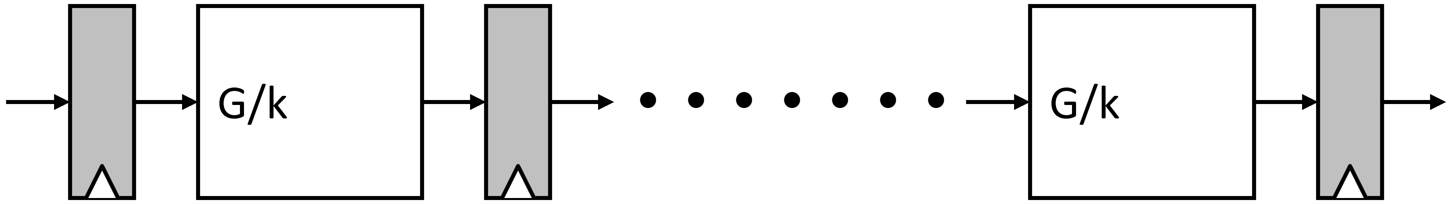
- Nonpipelined version with combinational cost G

  Cost = G+L where L = latch cost

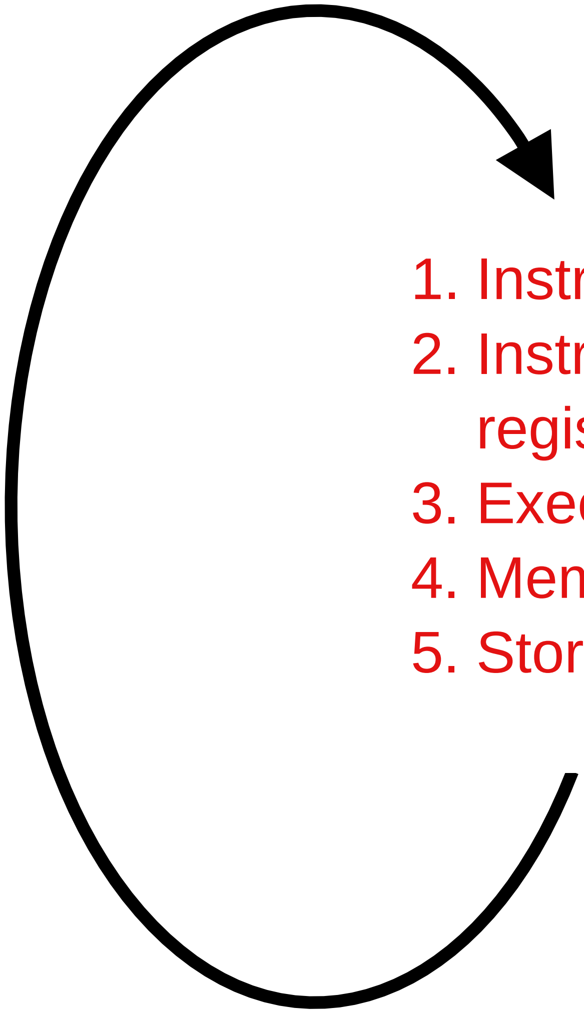


- k-stage pipelined version

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

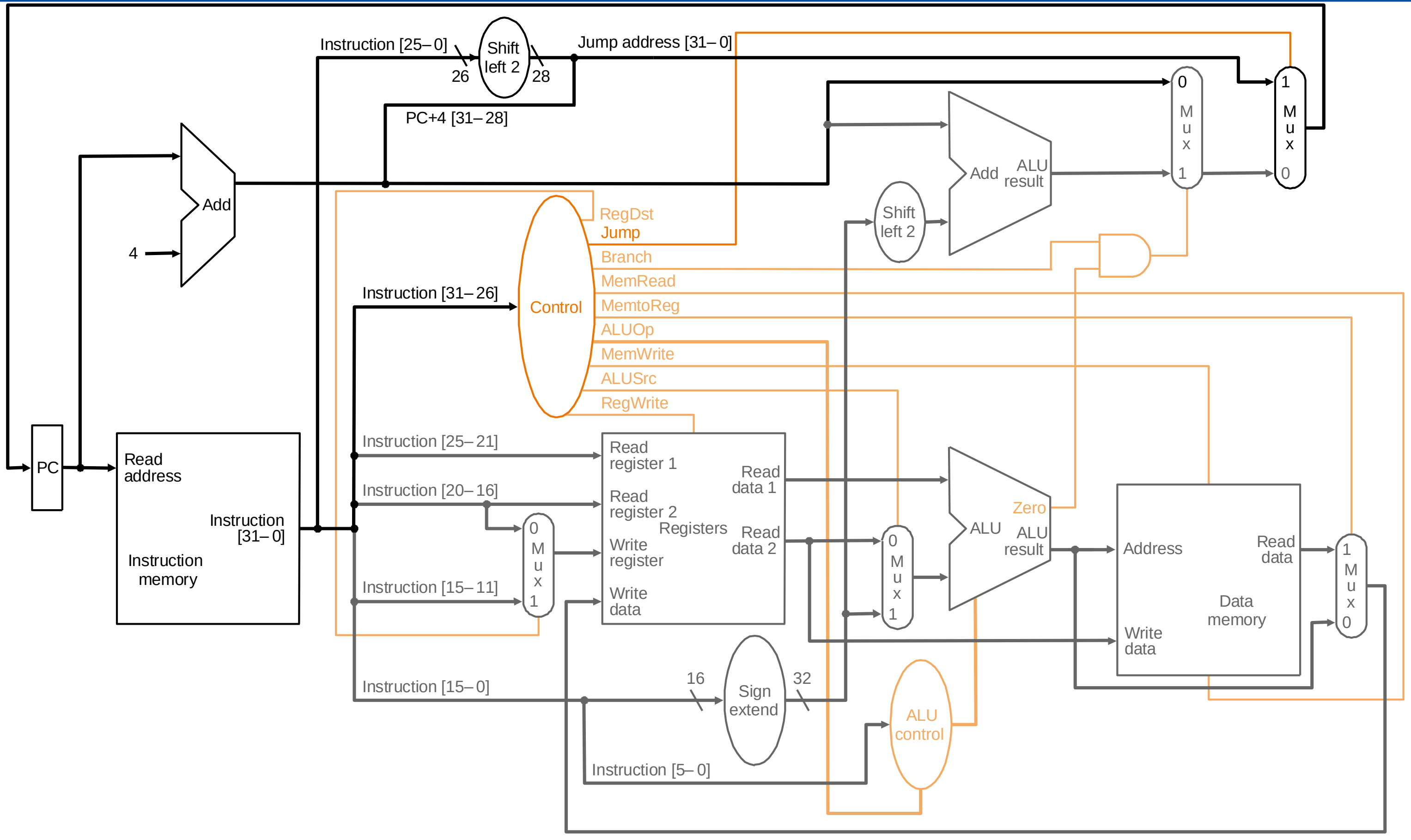


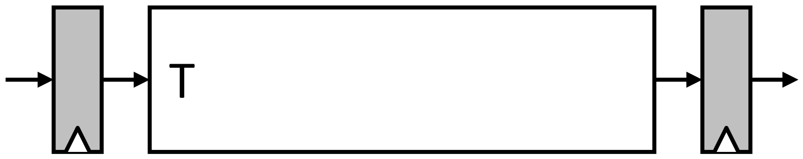
Remember: The Instruction Processing Cycle



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)

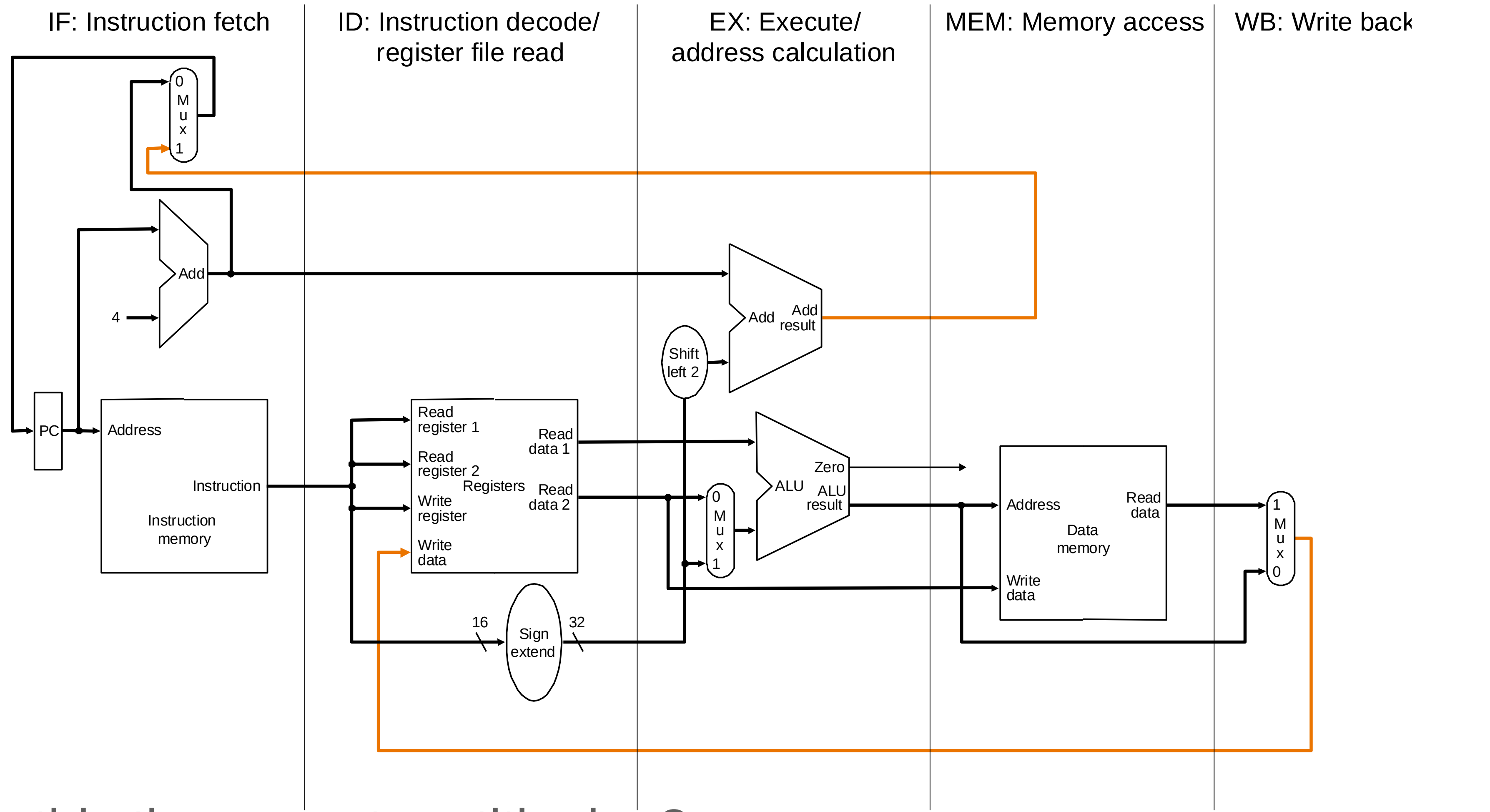
Remember the Single-Cycle Uarch





BW=~(1/T)

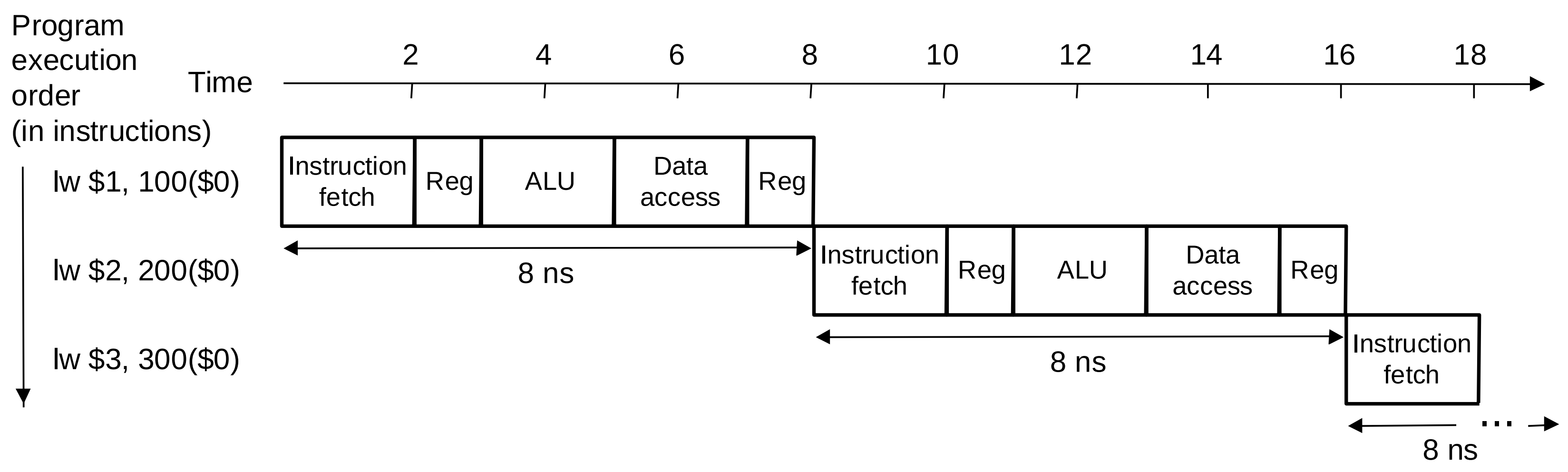
Dividing Into Stages

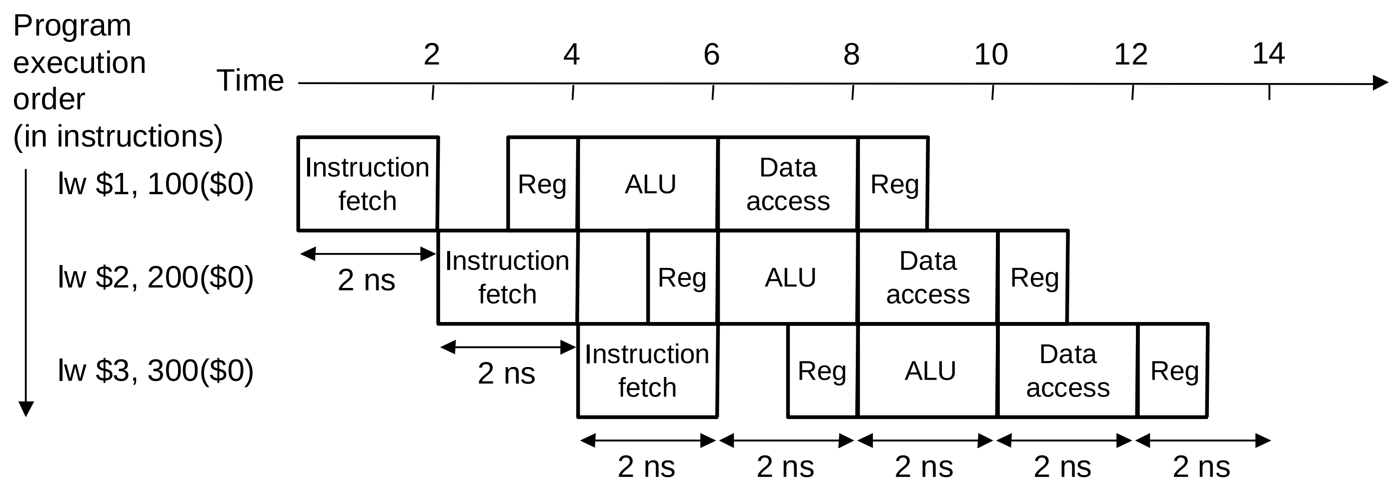


Is this the correct partitioning?

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

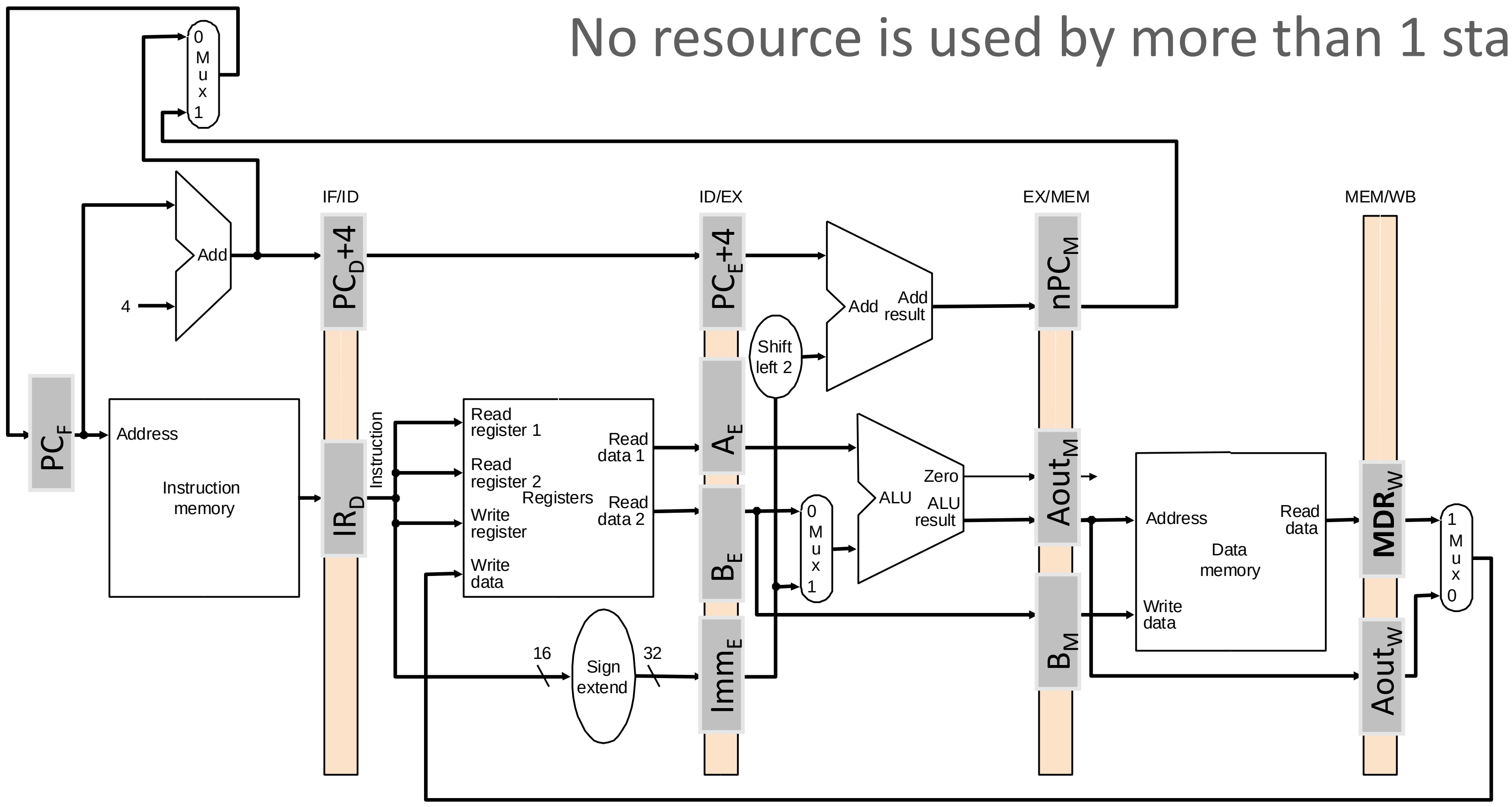
Instruction Pipeline Throughput

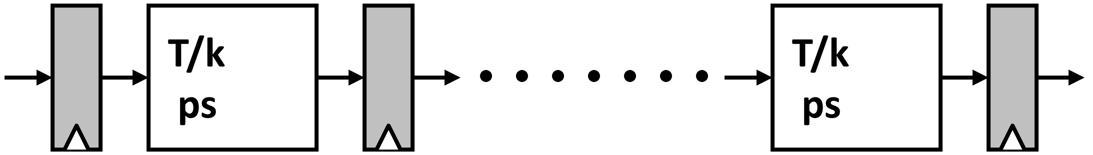




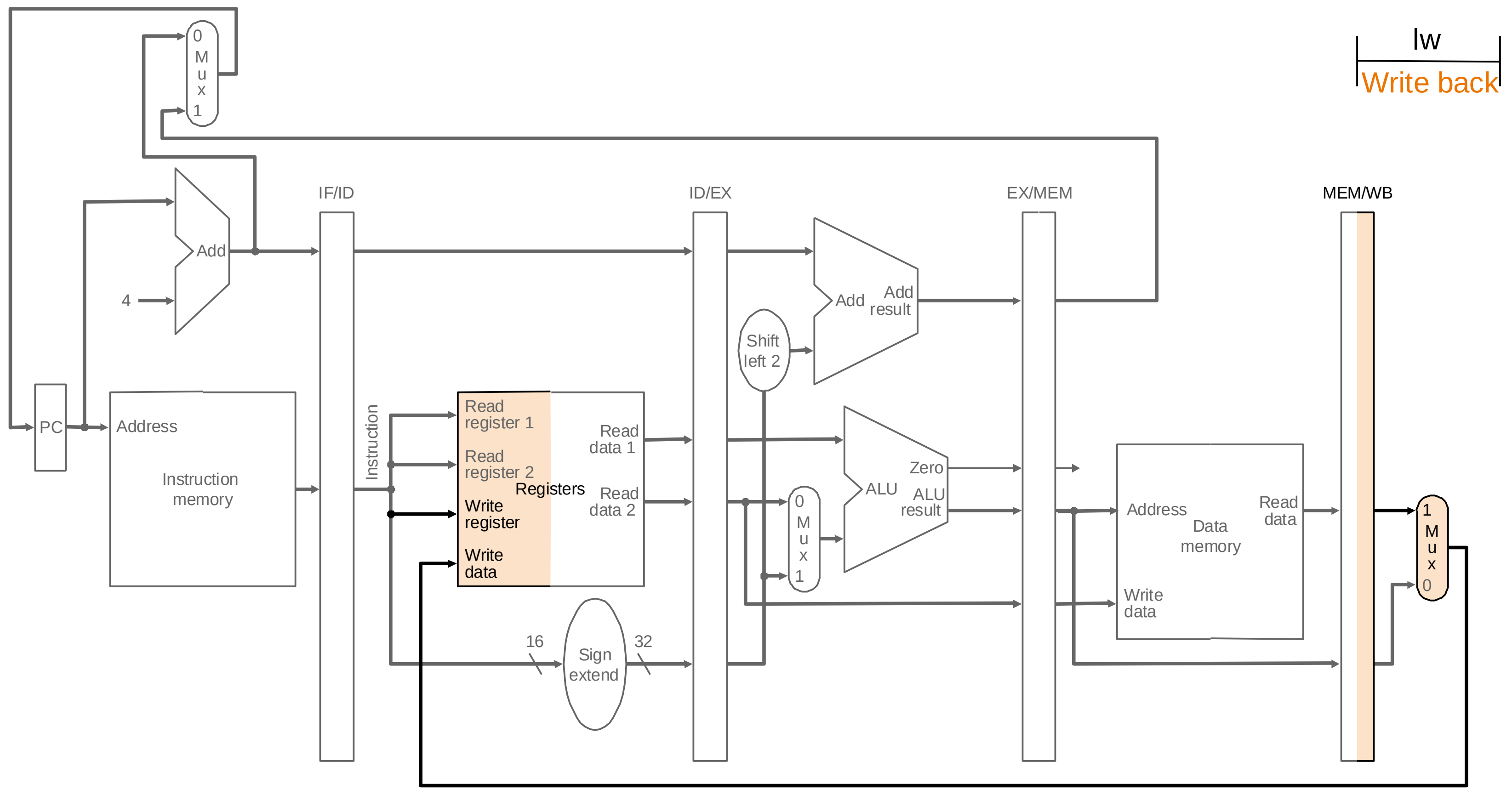
Pipelined Operation Example

No resource is used by more than 1 stage!

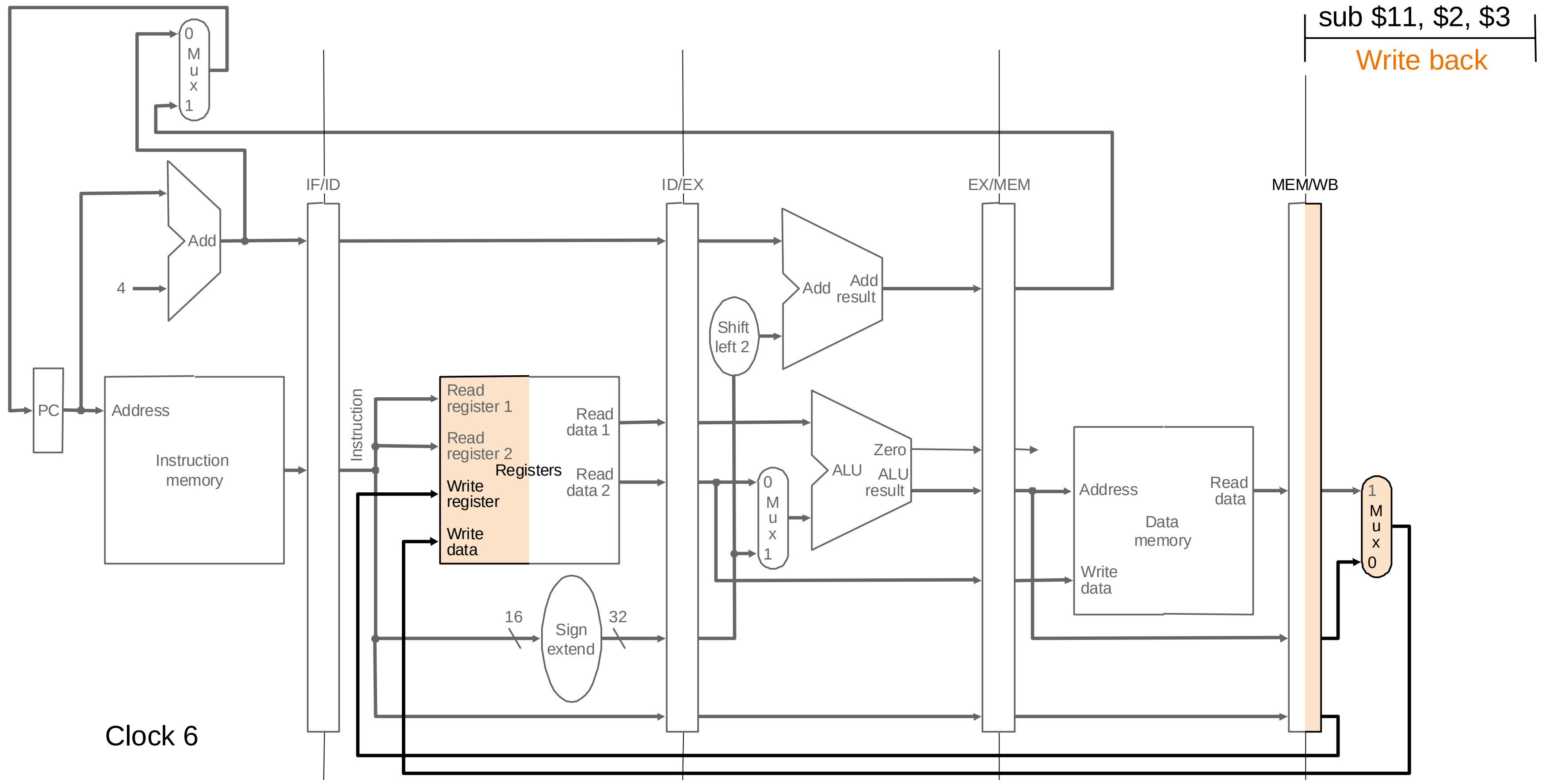




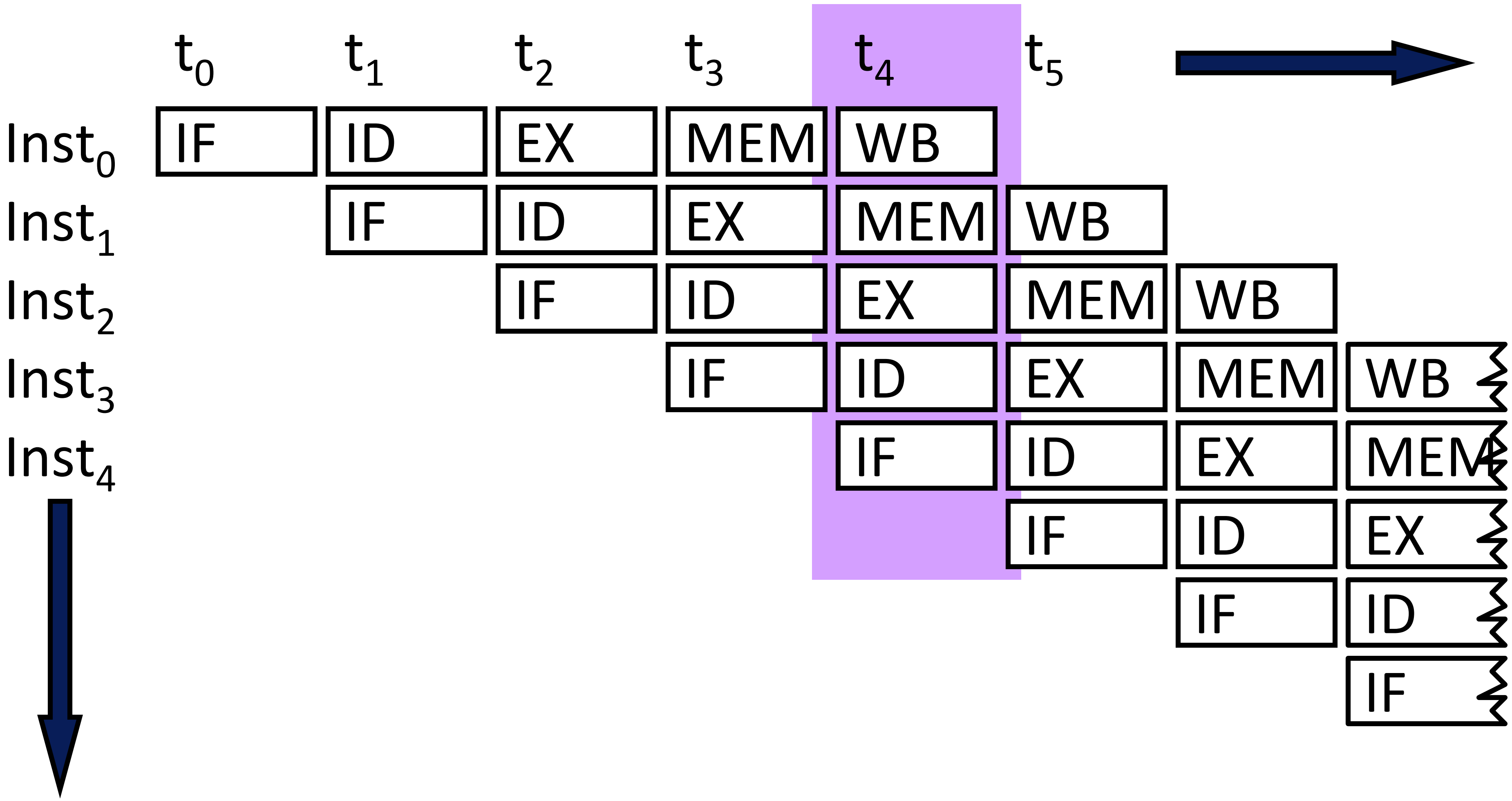
Pipelined Operation Example



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

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

- 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)

  $IR \leftarrow Mem[PC];$

  $NPC \leftarrow PC + 4$

- Instruction decode & register fetch (ID)

  $A \leftarrow Regs[IR_{6..10}];$

  $B \leftarrow Regs[IR_{11..15}];$

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

3rd Instruction cycle

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

4th Instruction cycle

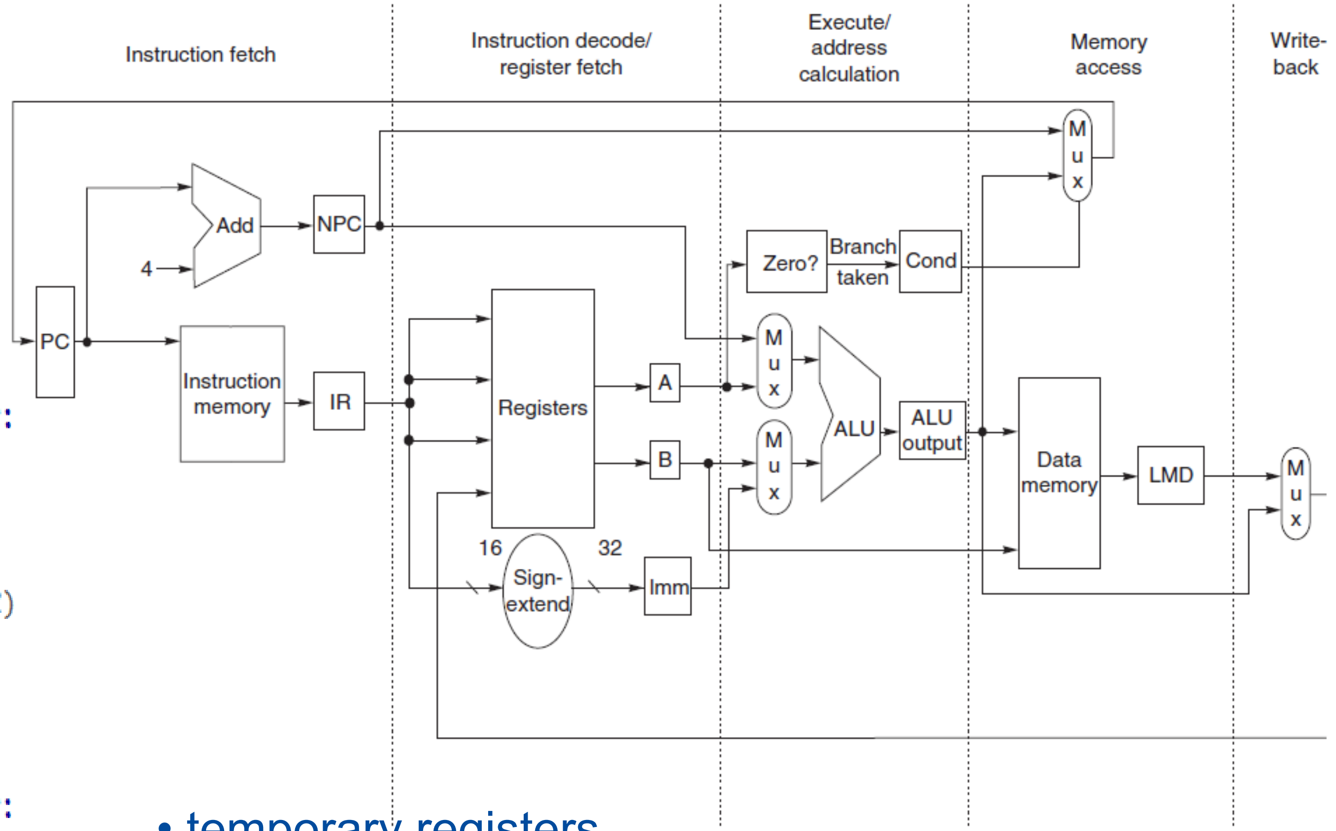
- Memory access & branch completion (MEM)
  - Memory reference
    - PC $\leftarrow$ NPC
    - LMD $\leftarrow$ Mem[ALUOutput] (load)
    - Mem[ALUOutput] $\leftarrow$ B (store)
  - Branch
    - if (cond) PC $\leftarrow$ ALUOutput; else PC $\leftarrow$ NPC

5th Instruction cycle

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

Simple Implementation of a MIPS

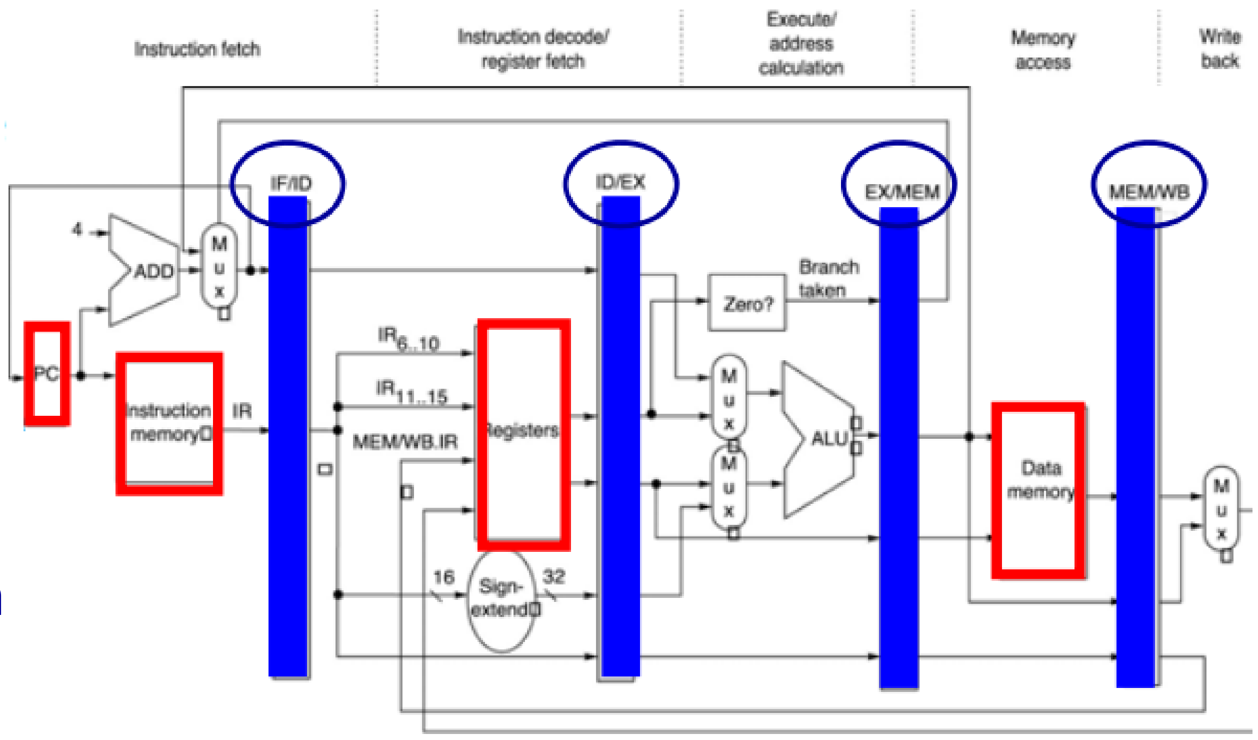
1. IR ← Mem[PC]; NPC ← PC+4;
2. A ← Regs[rs]; B ← Regs[rt]; Imm ← signExt(IR_{16})
   - fixed-field decoding
3. one of the followings:
   - ALUOutput ← A + Imm
   - ALUOutput ← func (A, B)
   - ALUOutput ← A op Imm
   - Cond ← (A == zero) *and* ALUOutput ← NPC + (Imm <<2)
4. PC ← MUX(Cond, ALUOutput, 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)

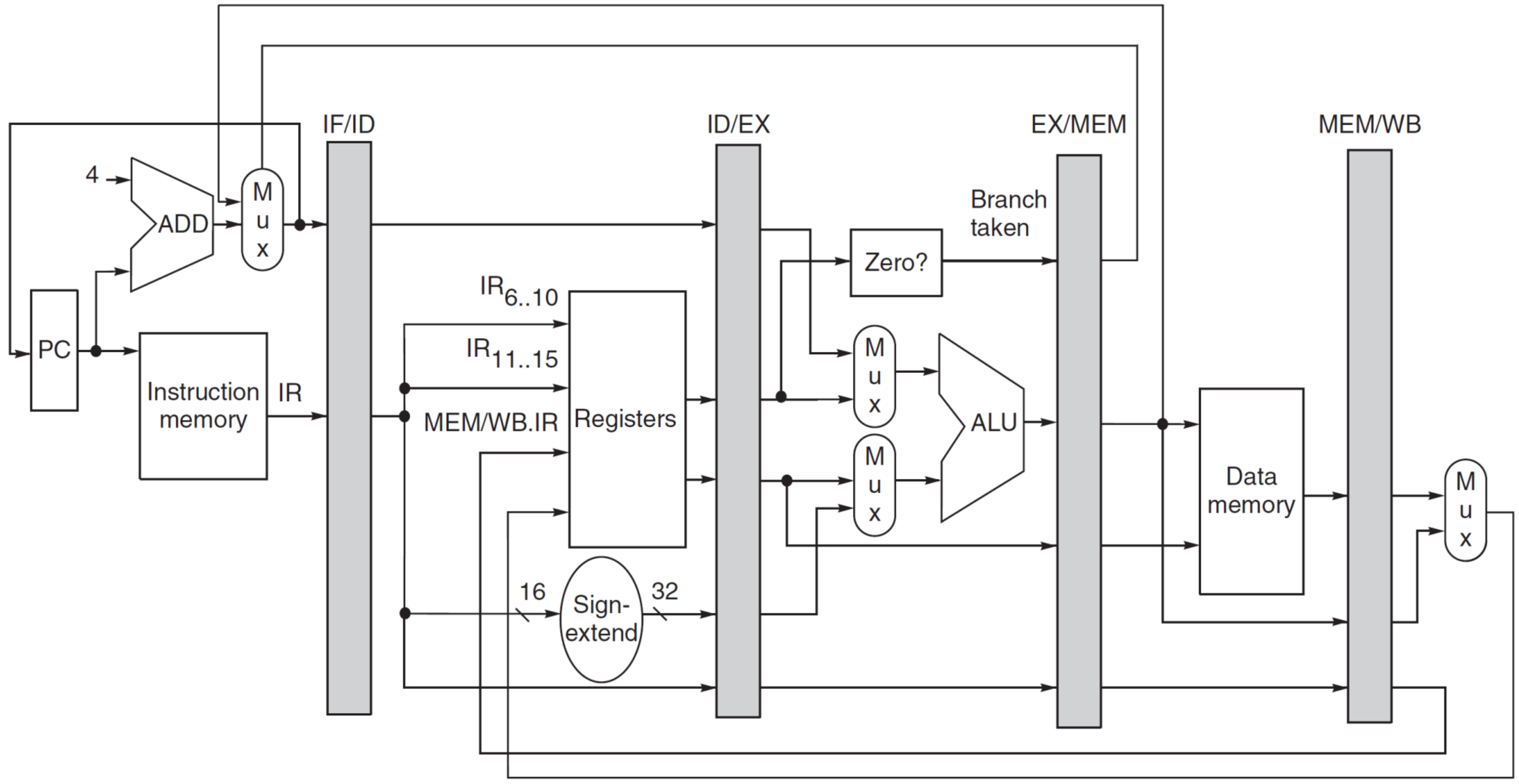
MIPS Pipeline Implementation

- Apparently easy
  - each clock cycle becomes a pipe stage
  - temporary registers become pipe registers
  - new instruction issued at each clock cycle
- Result propagation
  - 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

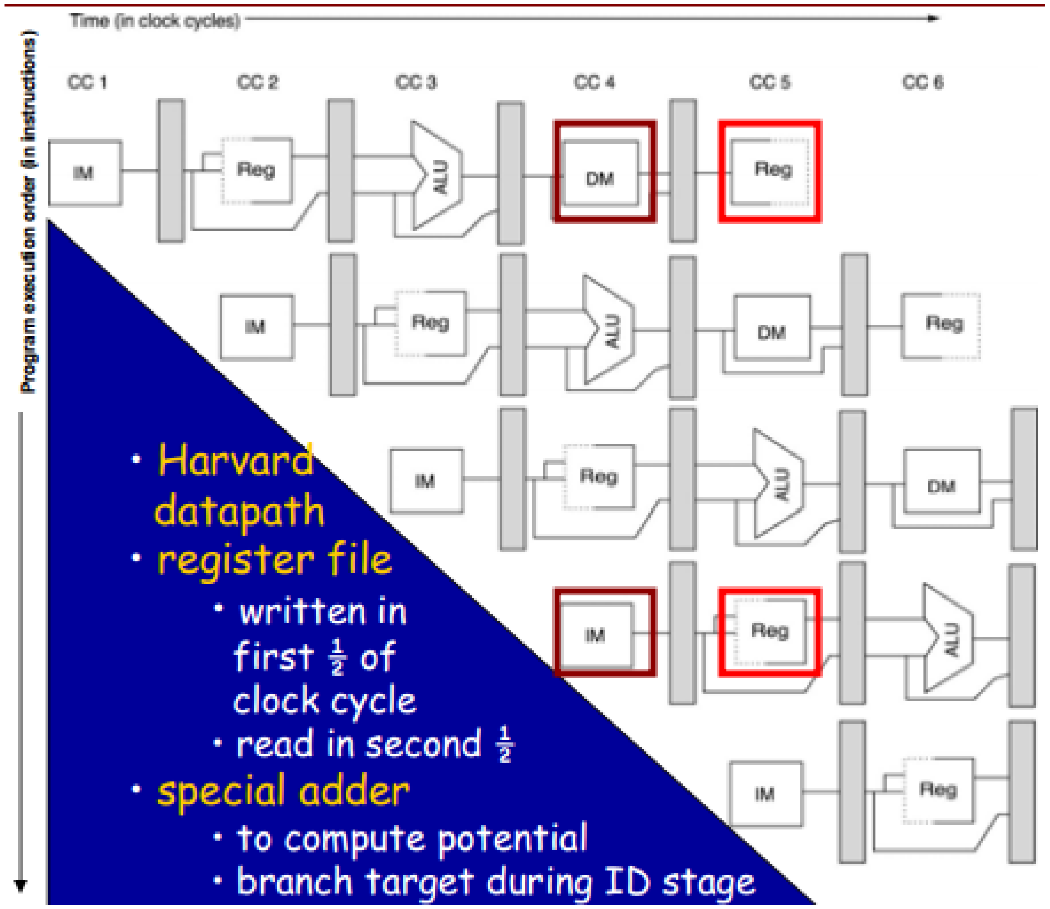
lw R10, 20(R1)

sub R11, R2, R3

add R12, R3, R4

lw R13, 24(R1)

add R14, R5, R6



- Harvard datapath
- register file
  - written in first ½ of clock cycle
  - read in second ½
- special adder
  - to compute potential
  - branch target during ID stage

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

MIPS Pipeline: Events per Stage

Stage	ALU Instruction	Load/Store Instruction	Branch Instruction

IF	IF/ID.IR ← Mem[PC];
	IF/ID.NPC ← if ((EX/MEM.opc == branch) & EX/MEM.cond) EX/MEM.AluOutput else PC+4;		

ID	ID/EX.A ← Regs[IF/ID.IR[rs]]; ID/EX.B ← Regs[IF/ID.IR[rt]];
	ID/EX.NPC ← IF/ID.NPC; ID/EX.IR ← IF/ID.IR;
	ID/EX.Imm ← sign-extend(IF/ID.IR[immediate field])		

EX	EX/MEM.IR ← ID/EX.IR; EX/MEM.ALUOutput ← ID/EX.A *op* ID/EX.B *or* EX/MEM.ALUOutput ← ID/EX.A *op* ID/EX.Imm	EX/MEM.IR ← ID/EX.IR; EX/MEM.ALUOutput ← ID/EX.A *op* ID/EX.Imm	EX/MEM.ALUOutput ← ID/EX.NPC + (ID/EX.Imm << 2) EX/MEM.Cond ← ID/EX.A == 0)
MEM	MEM/WB.IR ← EX/MEM.IR MEM/WB.ALUOutput ← EX/MEM.AluOutput	MEM/WB.IR ← EX/MEM.IR MEM/WB.LMD ← Mem[EX/MEM.AluOutput] *or* Mem[EX/MEM.AluOutput] ← EX/MEM.B	
WB	Regs[MEM/WB.IR[rd]] ← MEM/WB.AluOutput *or* Regs[MEM/WB.IR[rt]] ← MEM/WB.AluOutput	for load only: Regs[MEM/WB.IR[rt]] ← MEM/WB.LMD	

MIPS Features that Simplify Pipelining

- ISA encoding: all instructions have same length
  - balanced separation of fetch and decode stages
- ISA encoding: few instructions formats with the source register fields in the same position
  - symmetry allows reading of register file and decoding of instruction simultaneously during the second stage (fixed-field decoding)
- Memory accessed only by load/store instructions
  - memory address is “pre-computed” during the execution stage for the following stage (no operation involves operands in memory)
- Each operation writes at most on result (and near the end of the pipeline)
  - simplifies forwarding (useful for handling data hazards)
- All instructions change an entire register, memory access does not require realignment…
  - simpler instructions can be decomposed in steps that can be performed each in a single pipeline stage

Basic Pipeline

Clock number

Instr #	1	2	3	4	5	6	7	8	9
i	IF	ID	EX	MEM	WB				
i +1		IF	ID	EX	MEM	WB			
i +2			IF	ID	EX	MEM	WB		
i +3				IF	ID	EX	MEM	WB	
i +4					IF	ID	EX	MEM	WB

The First Pipelined Computers

- IBM 7030 “Stretch”
  - considered one of the first general-purpose pipelined computer (late 1950s)
  - pipeline overlapping fetch, decode, execute stages
- CDC 6600
  - Control Data Corp. (Seymour Cray)
  - the first supercomputer (1964)
  - interaction between ISA and pipeline HW was well understood
    - ISA was kept simple on purpose
- IBM 360
  - more complex pipeline with Tomasulo’s Algorithm (1964)