Pipelining

This part of the lecture is based on slides from

Prof. Onur Mutlu

ETH Zurich

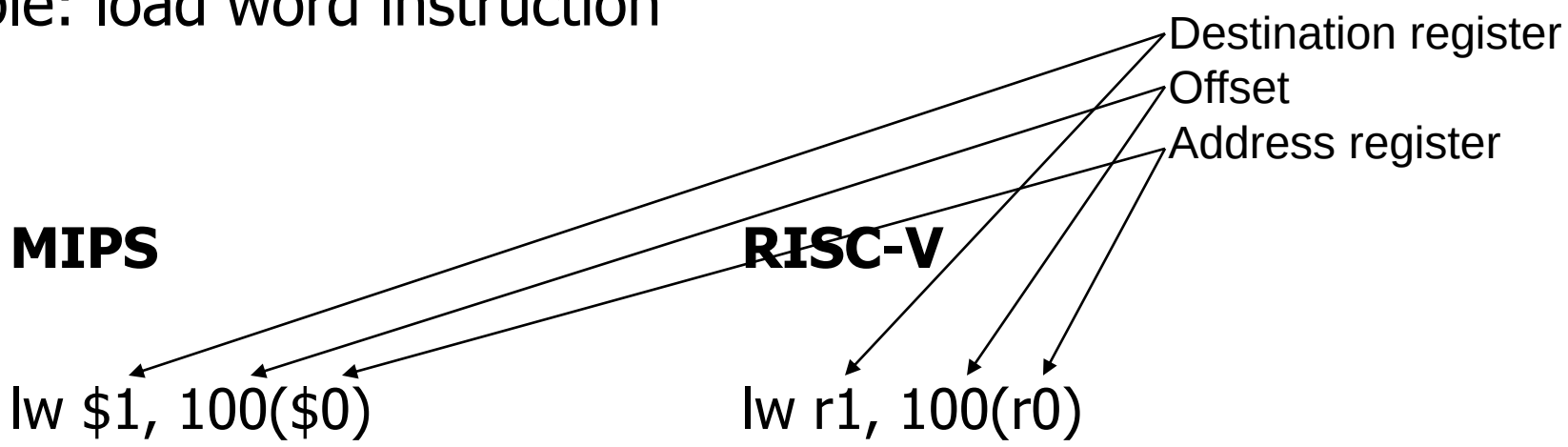
The slide sequence has been changed and slides have been combined from multiple lectures; minor textual updates; Original source:

https://safari.ethz.ch/digitaltechnik/spring2019/doku.php?id=schedule

RISC-V vs. MIPS

- **Note:** The following slides are based on the MIPS instruction set, but the concepts equally hold for RISC-V
- In MIPS assembler, the $ sign stands for a register.

Example: load word instruction



Reading

- Smith and Sohi, "The Microarchitecture of Superscalar Processors," Proceedings of the IEEE, 1995
  - More advanced pipelining
  - Interrupt and exception handling
  - Out-of-order and superscalar execution concepts

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: More concurrency → Higher instruction throughput (i.e., 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

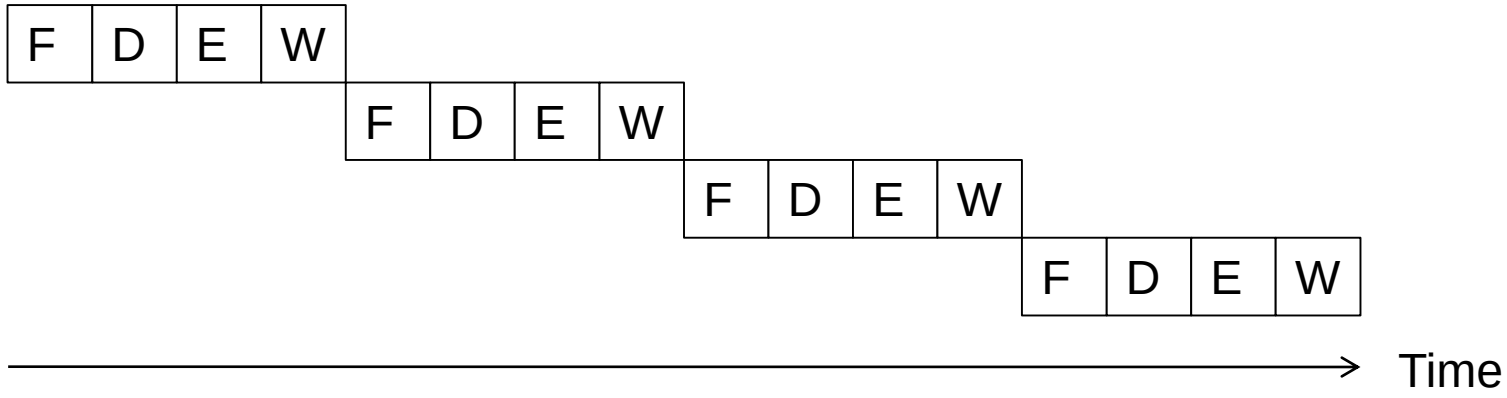
Pipelining

Pipelining: Basic Idea

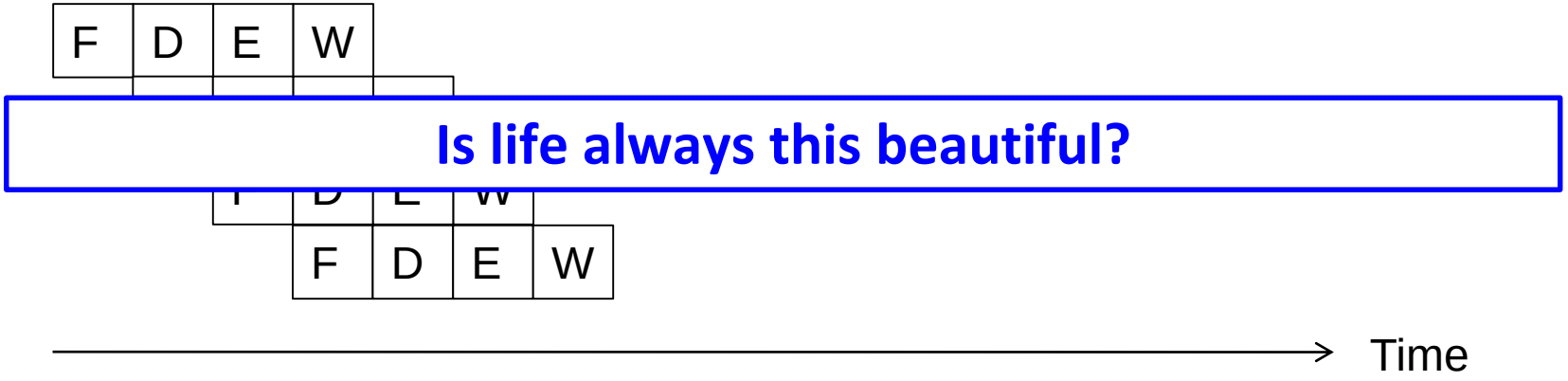
- More systematically:
  - Pipeline the execution of multiple instructions
  - Analogy: "Assembly line processing" of instructions
- Idea:
  - Divide the instruction processing cycle into distinct "stages" of processing
  - Ensure there are enough hardware resources to process one instruction in each 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

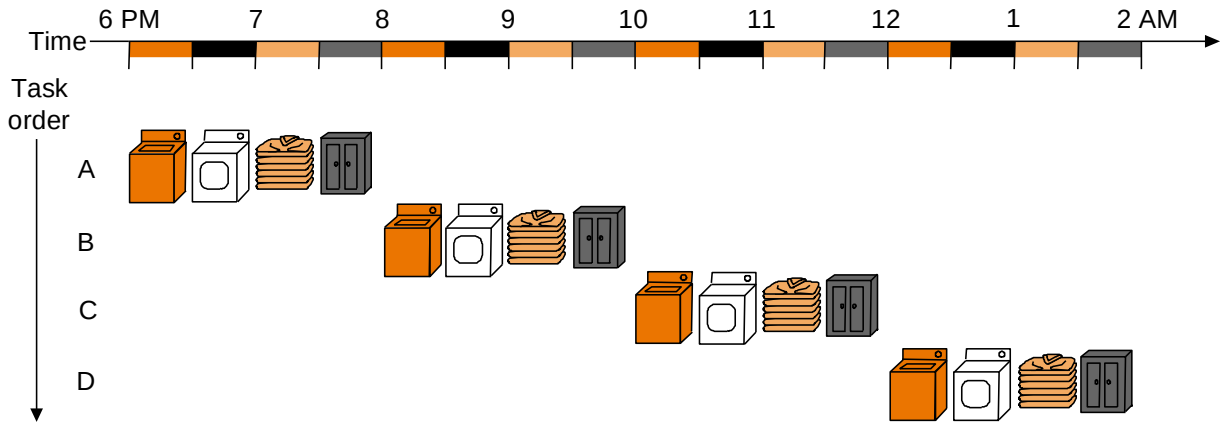
- Multi-cycle: 4 cycles per instruction



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



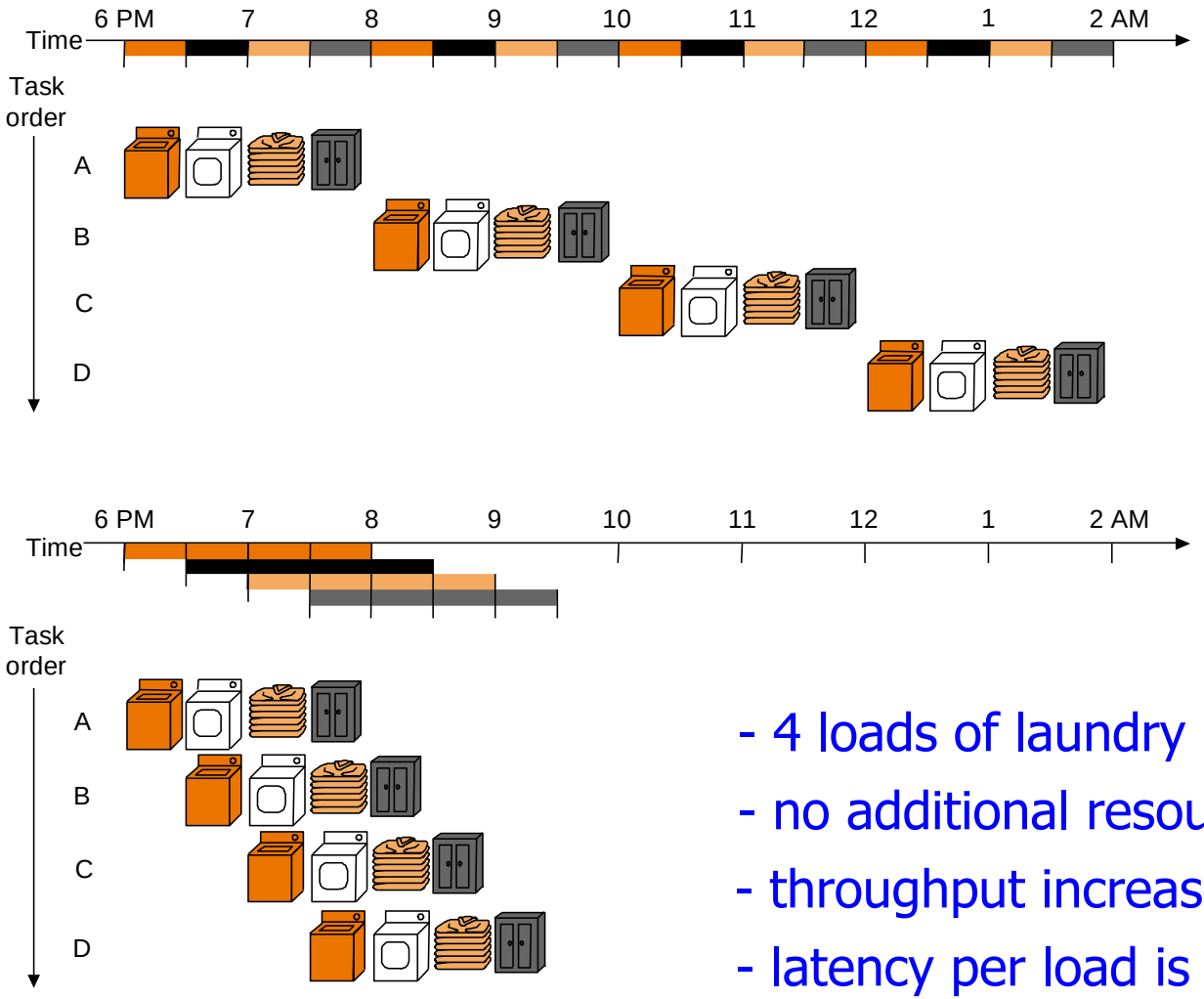
The Laundry Analogy



- "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"

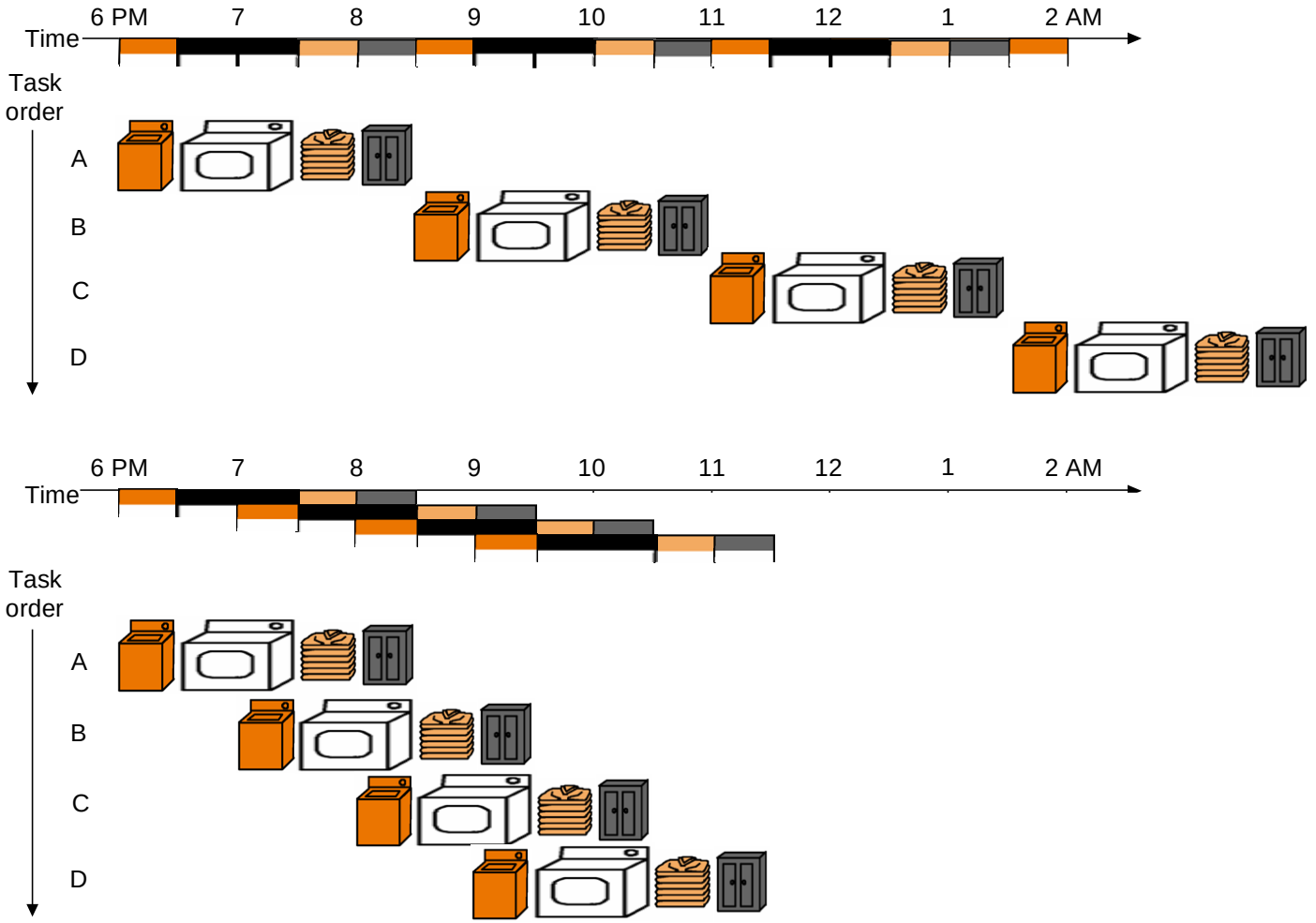
- steps to do a load are sequentially dependent
- no dependence between different loads
- different steps do not share resources

Pipelining Multiple Loads of Laundry



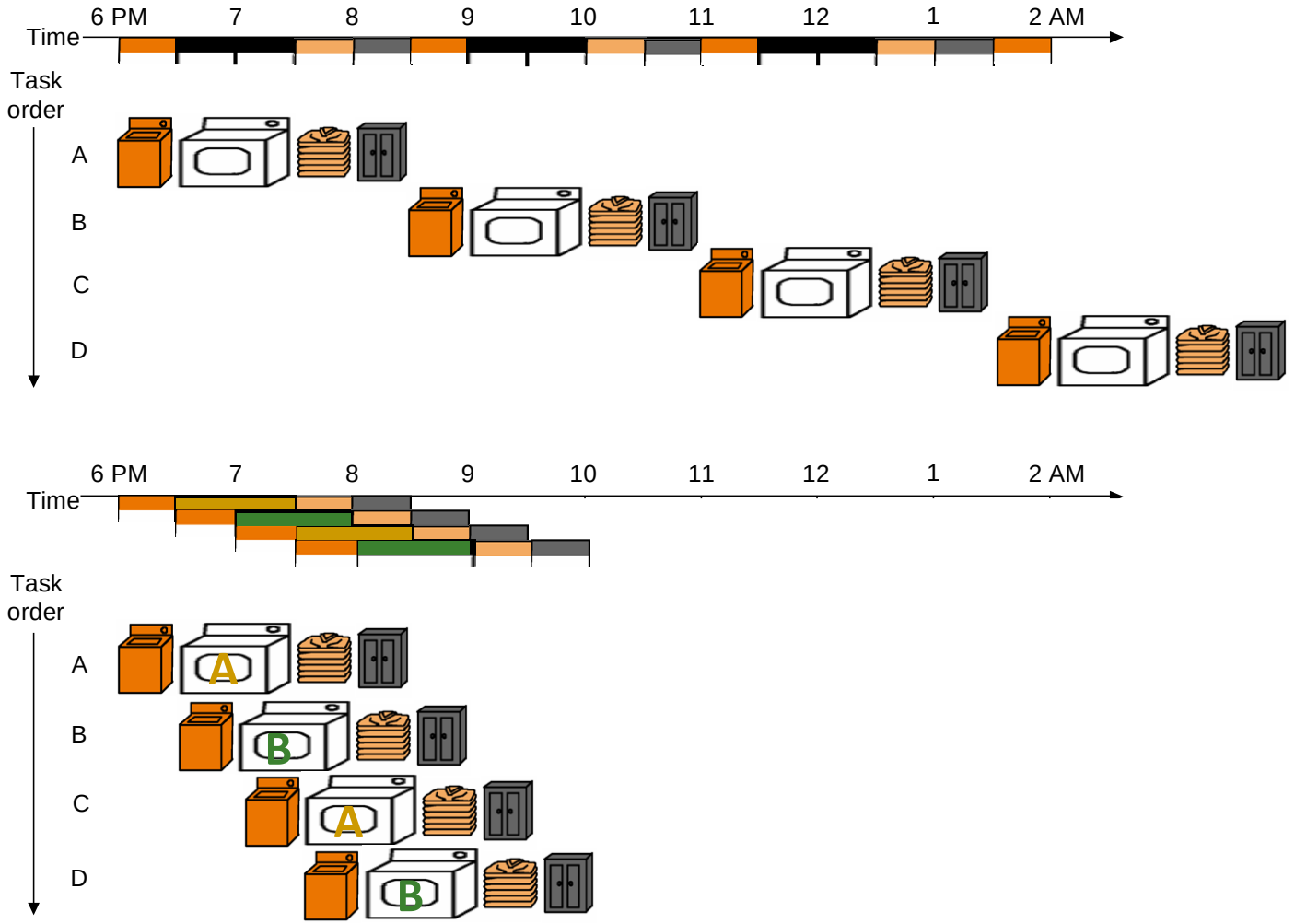
- 4 loads of laundry in parallel
- no additional resources
- throughput increased by 4
- latency per load is the same

Pipelining Multiple Loads of Laundry: In Practice



the slowest step decides throughput

Pipelining Multiple Loads of Laundry: In Practice

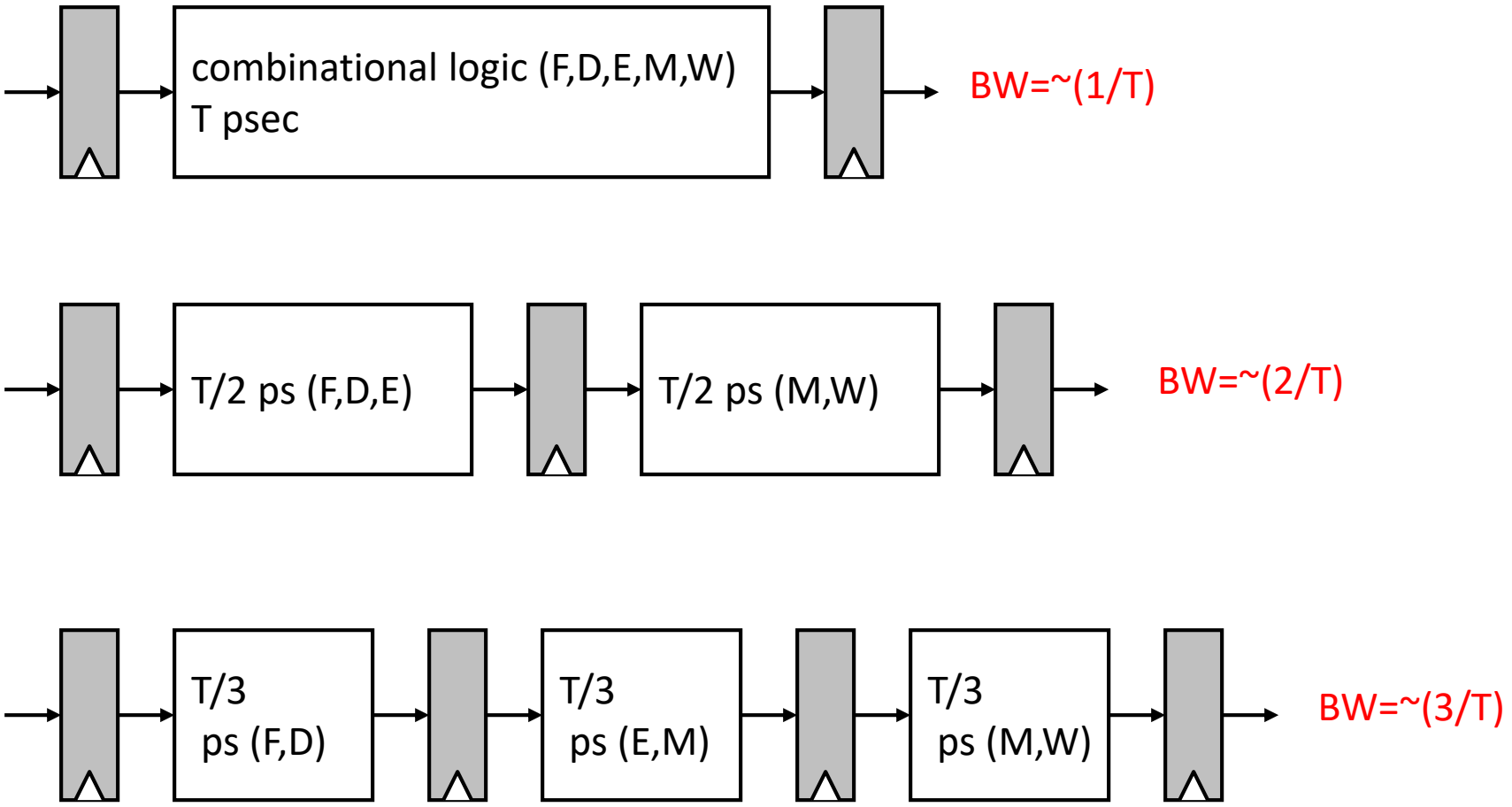


throughput restored (2 loads per hour) using 2 dryers

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 (e.g., all laundry loads go through the same steps)
- 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

Ideal Pipelining



More Realistic Pipeline: Throughput

- Nonpipelined version with delay T

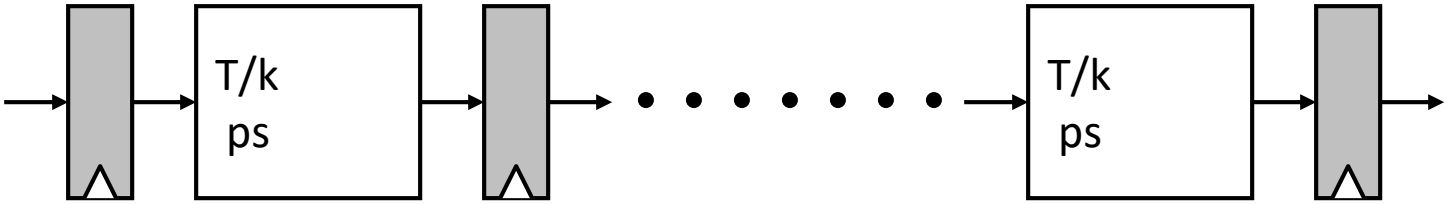
  BW = 1/(T+S) where S = register delay

- k-stage pipelined version

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

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

Register delay reduces throughput (switching overhead between stages)



More Realistic Pipeline: Cost

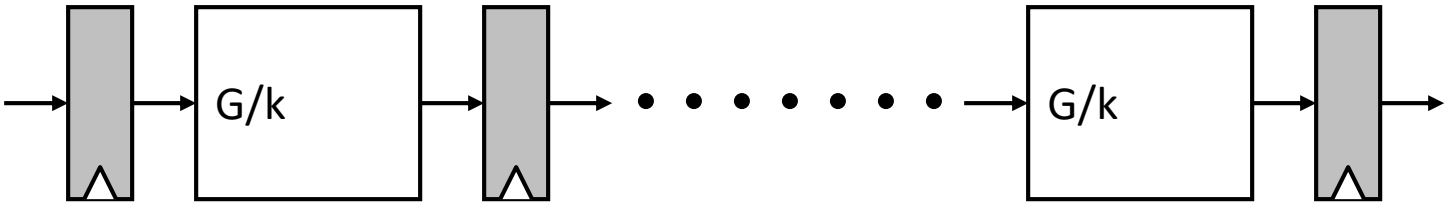
- Nonpipelined version with combinational cost G

  Cost = G+L where L = register cost

- k-stage pipelined version

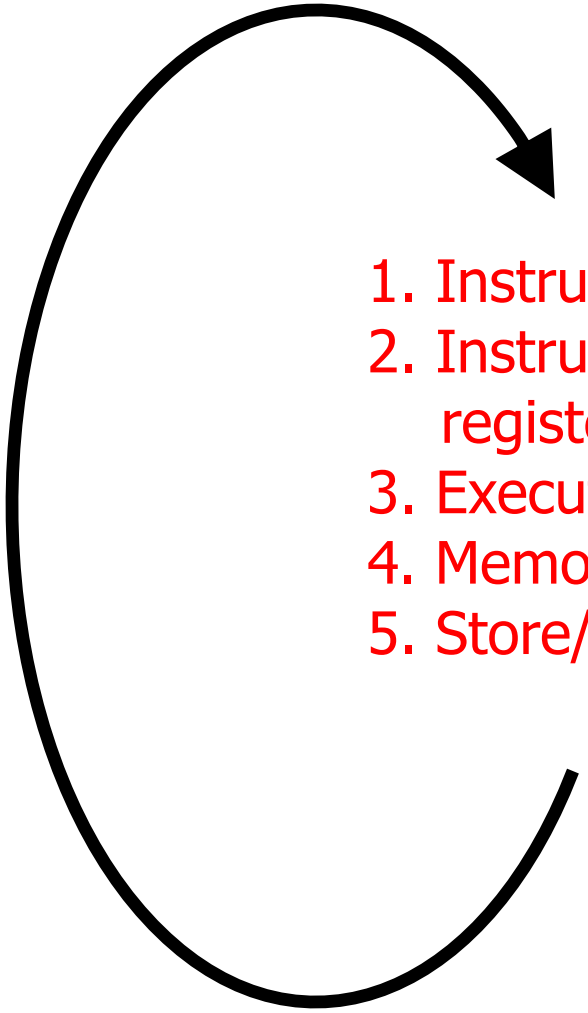
  $Cost_{k\text{-}stage} = G + Lk$

Registers increase hardware cost



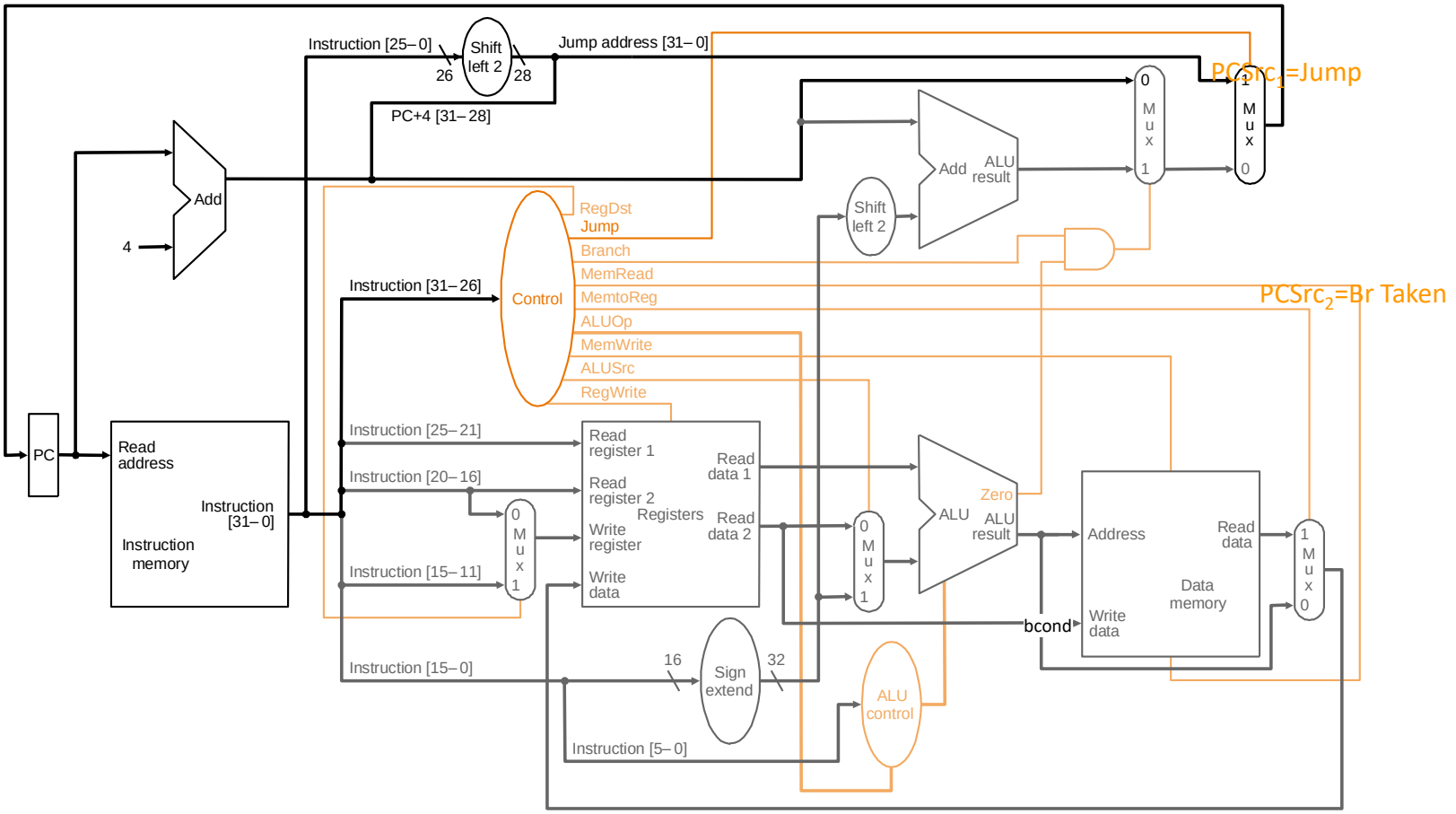
Pipelining Instruction Processing

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)

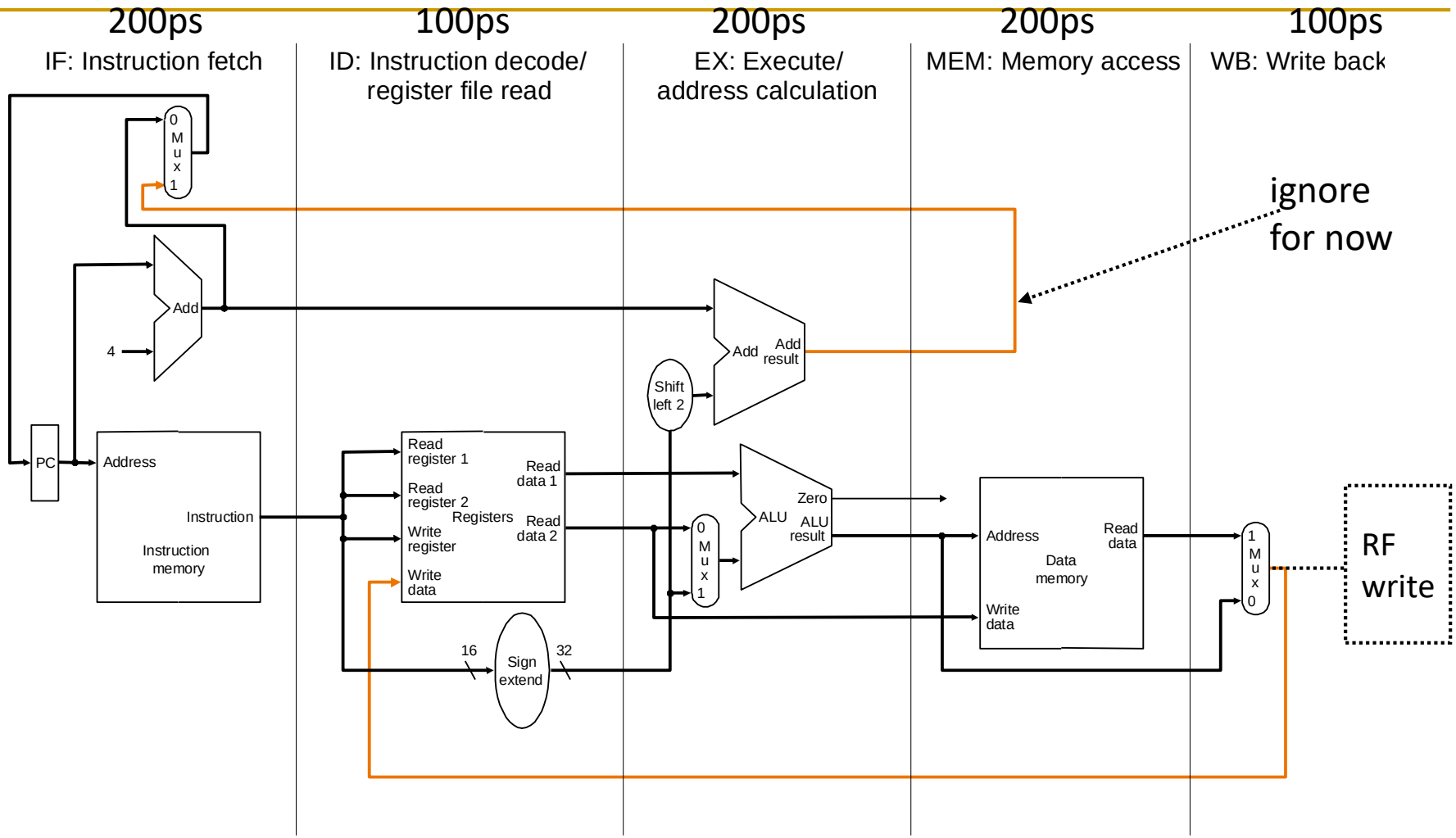
Remember the Single-Cycle Uarch



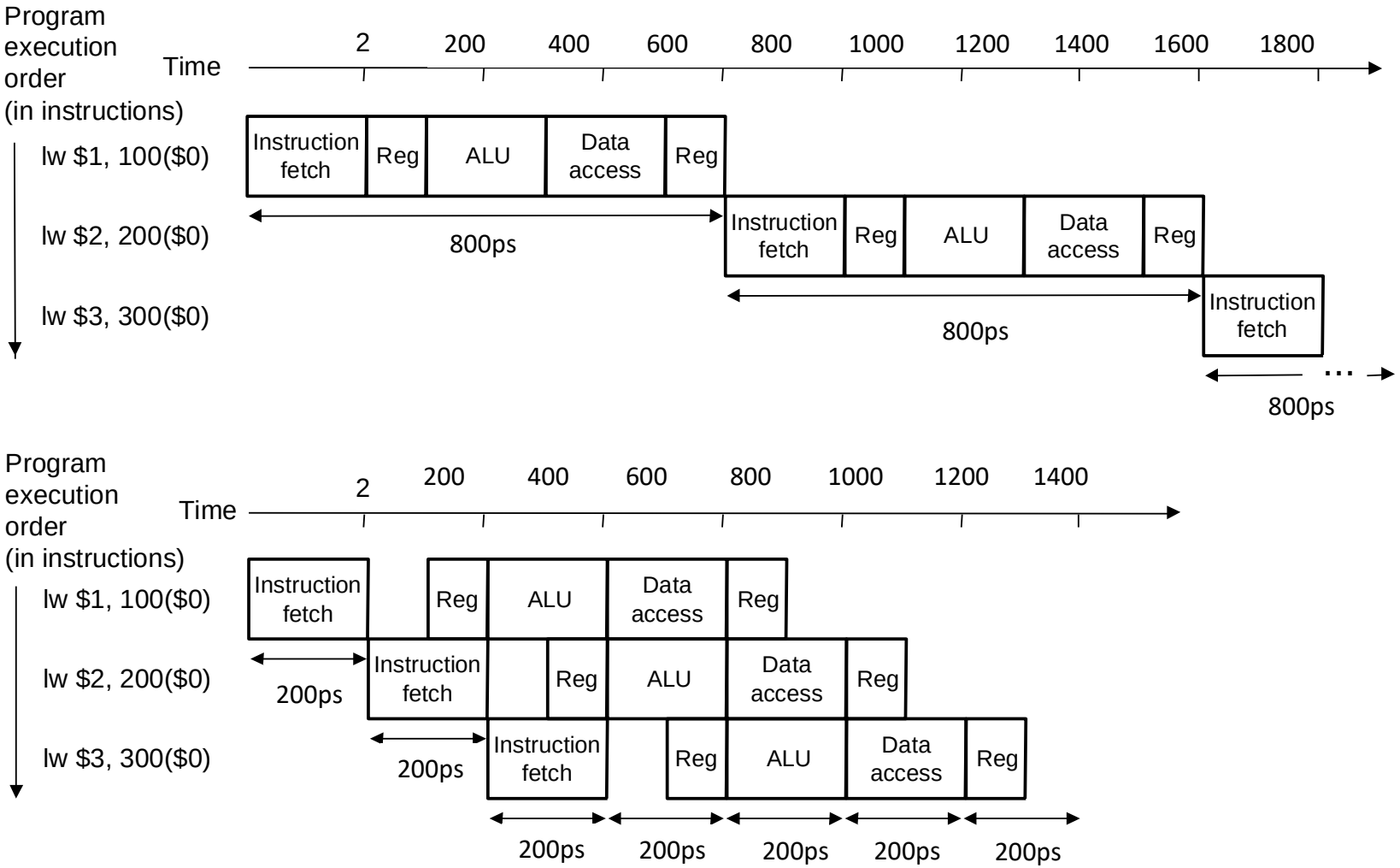
ALU operation

BW=~(1/T)

Dividing Into Stages



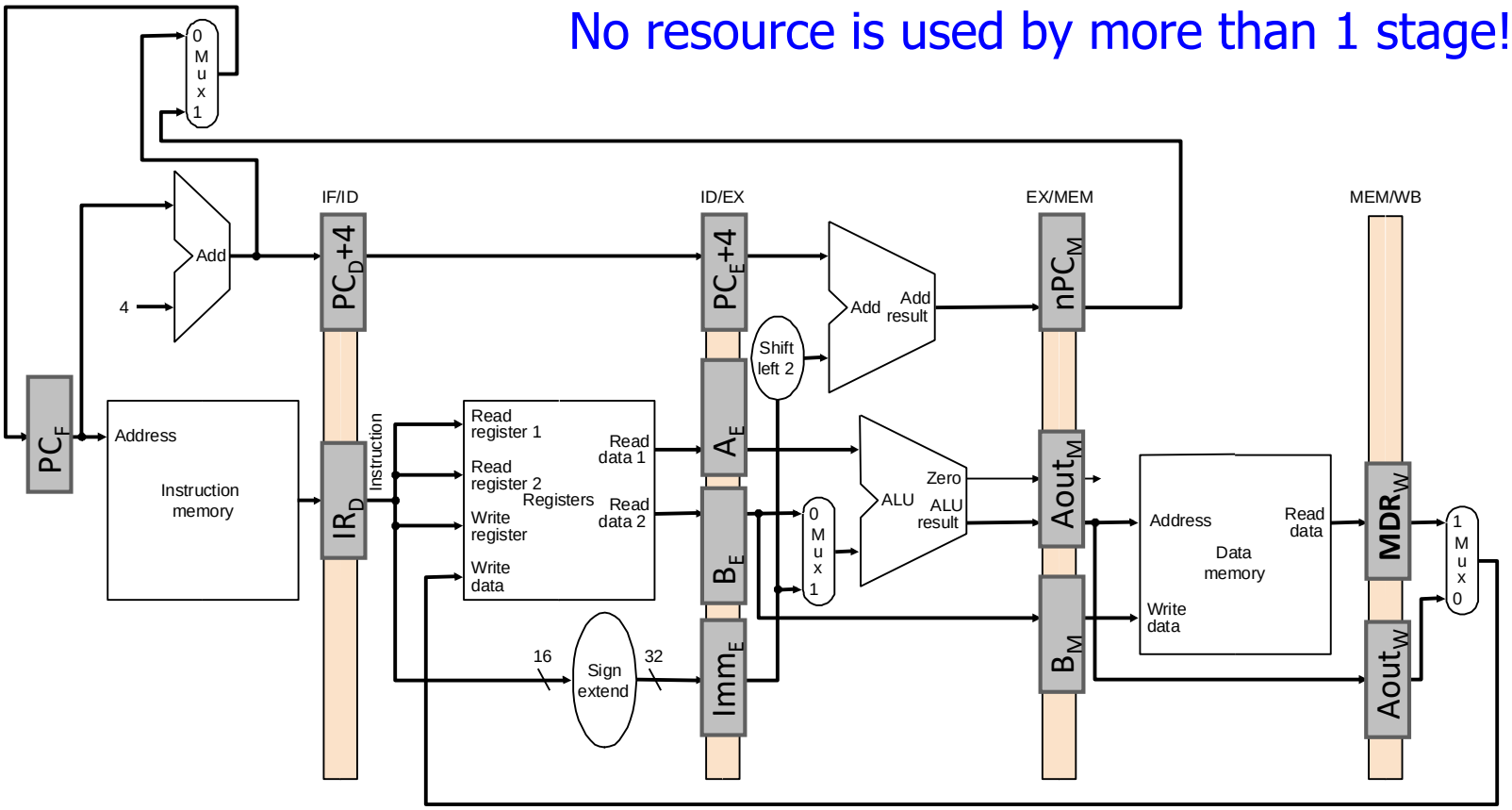
Instruction Pipeline Throughput

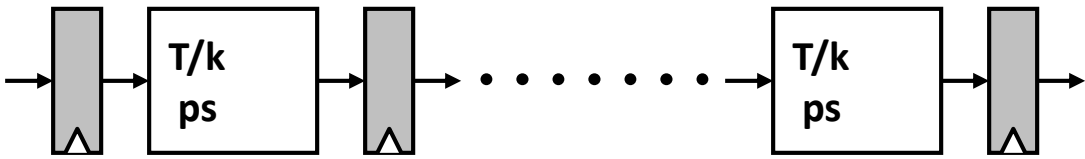


5-stage speedup is 4, not 5 as predicted by the ideal model. Why?
(We complete an instruction every 200ps instead of every 800ps)

Enabling Pipelined Processing: Pipeline Registers

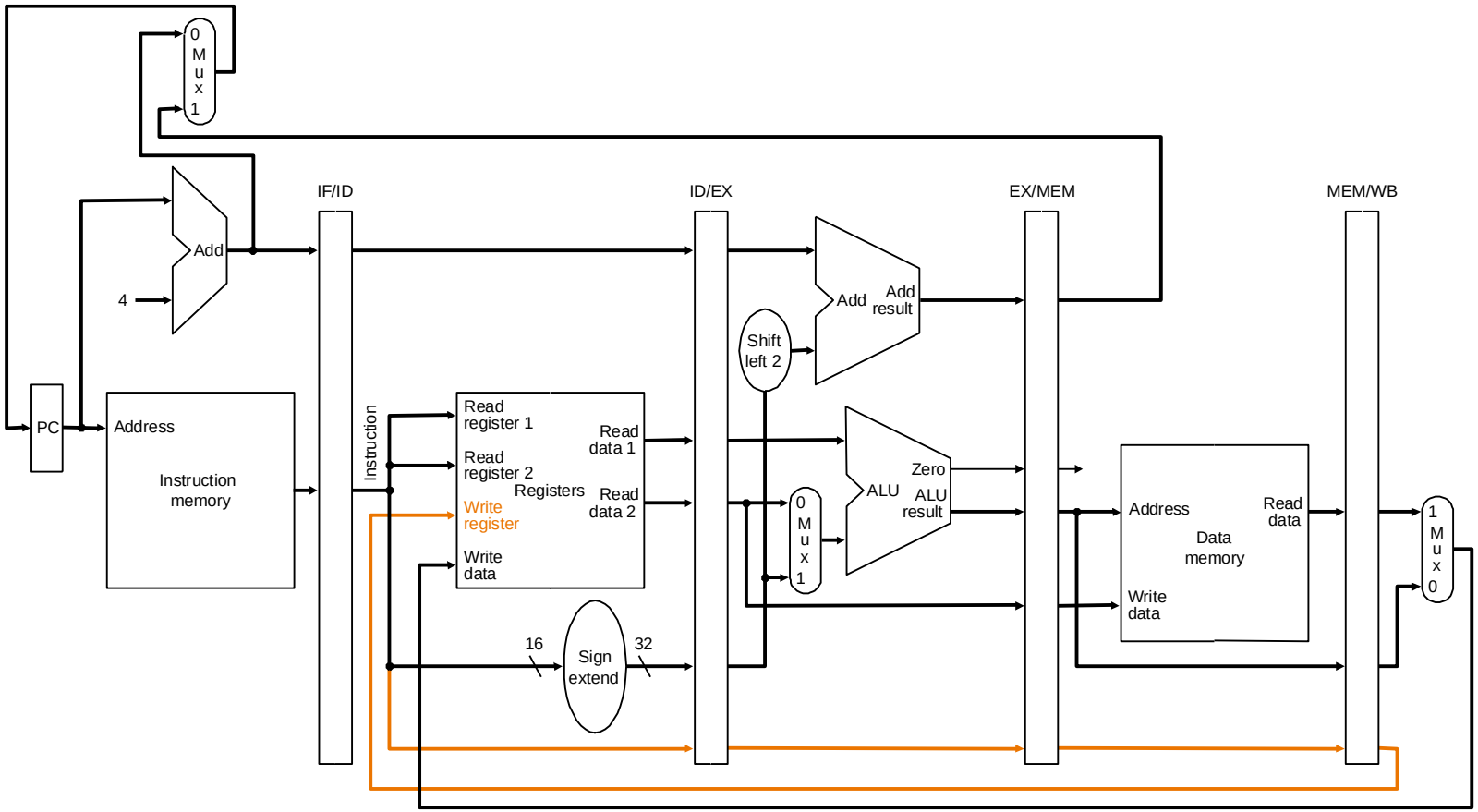
No resource is used by more than 1 stage!



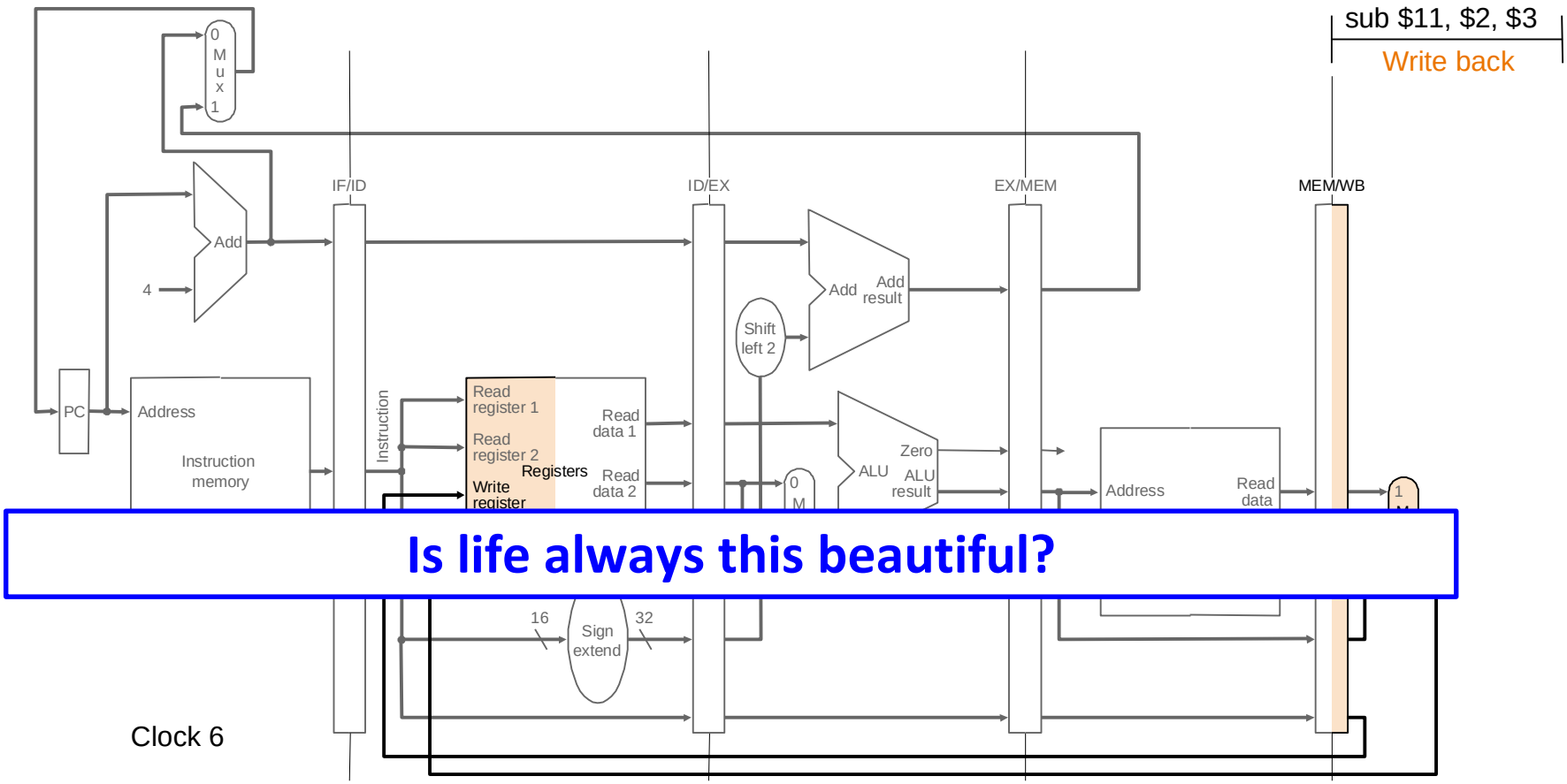


Pipelined Operation Example

All instruction classes must follow the same path and timing through the pipeline stages.

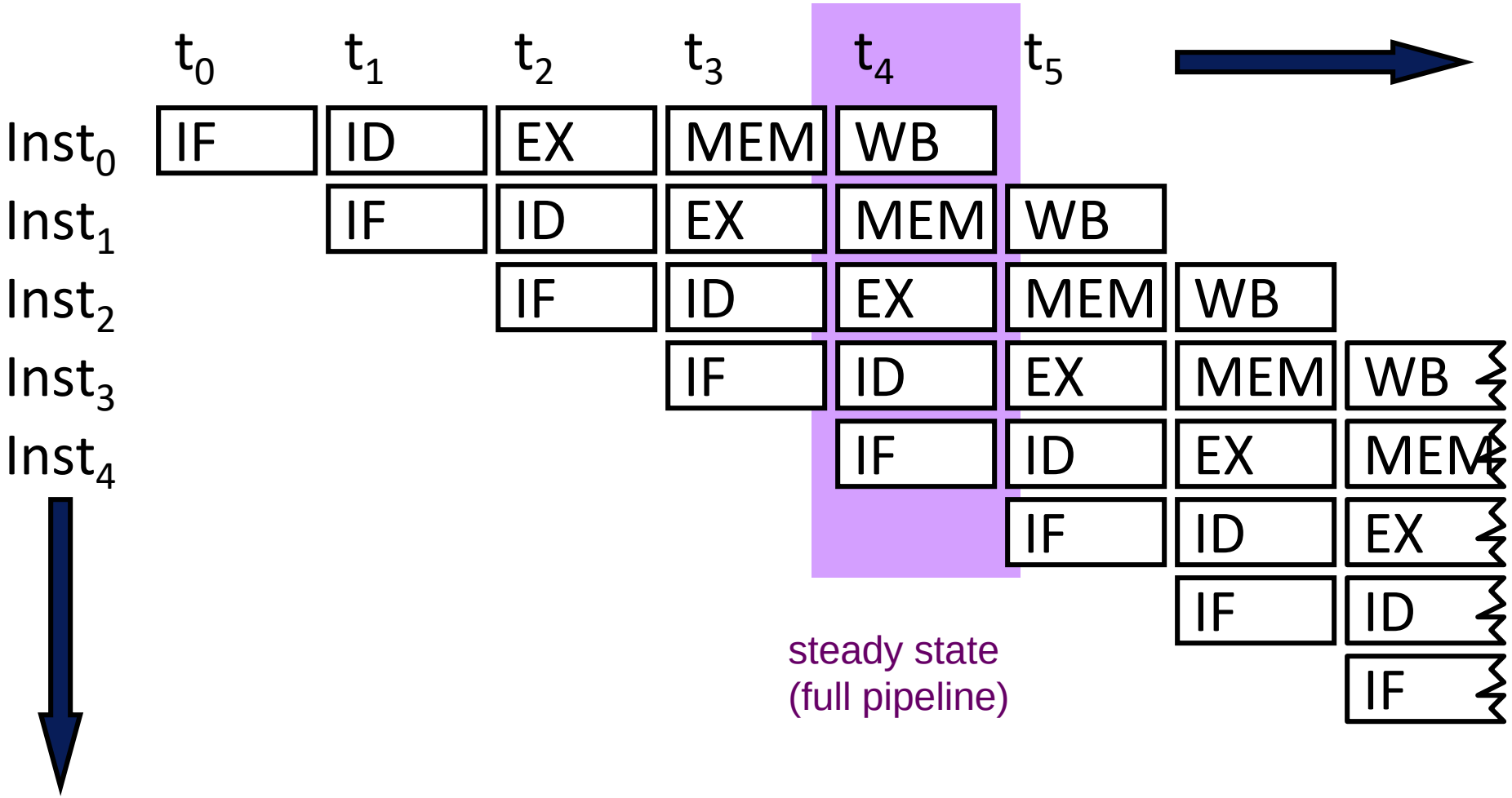


Pipelined Operation Example



Is life always this beautiful?

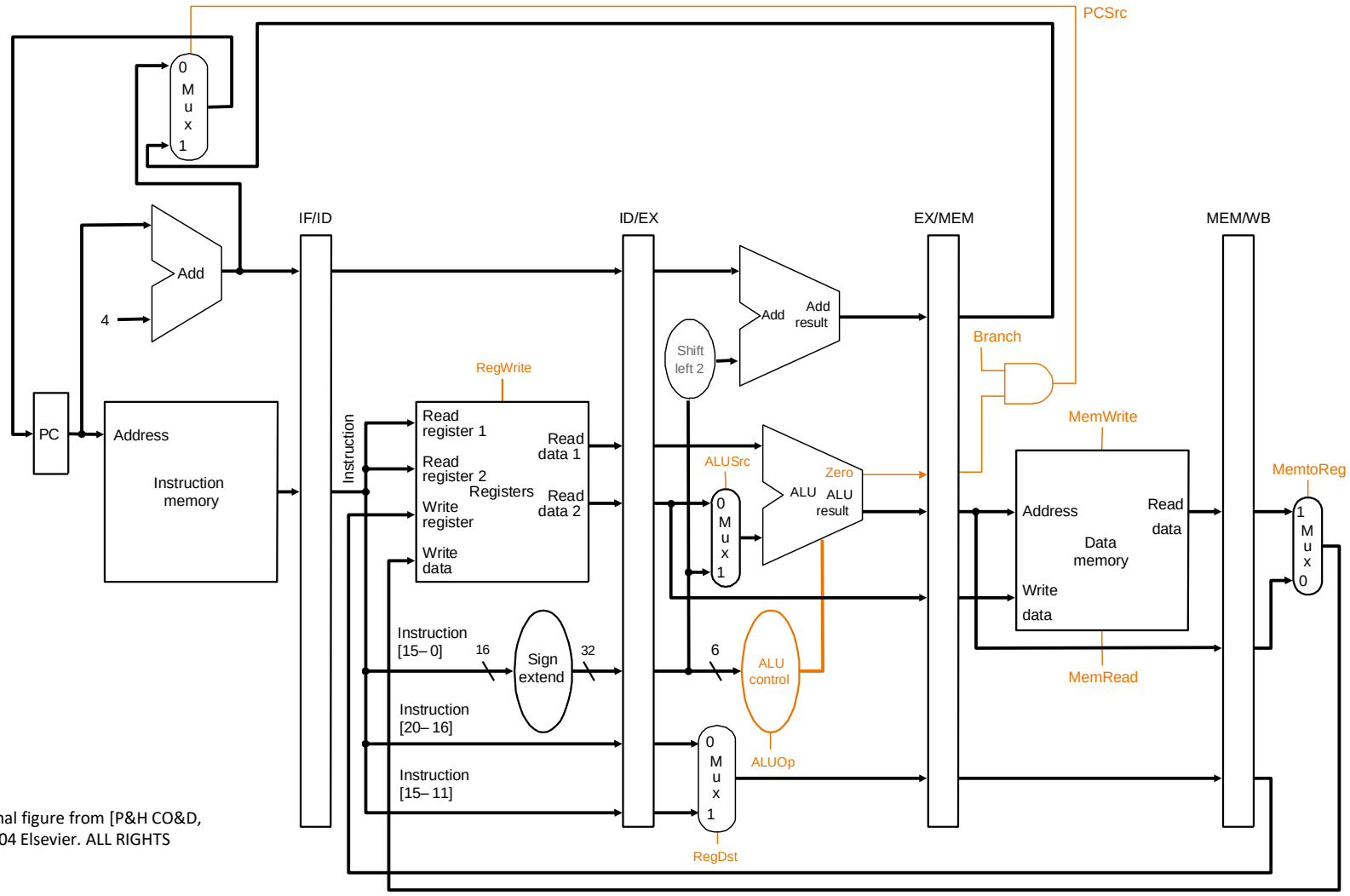
Illustrating Pipeline Operation: Operation View



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

Control Points in a Pipeline

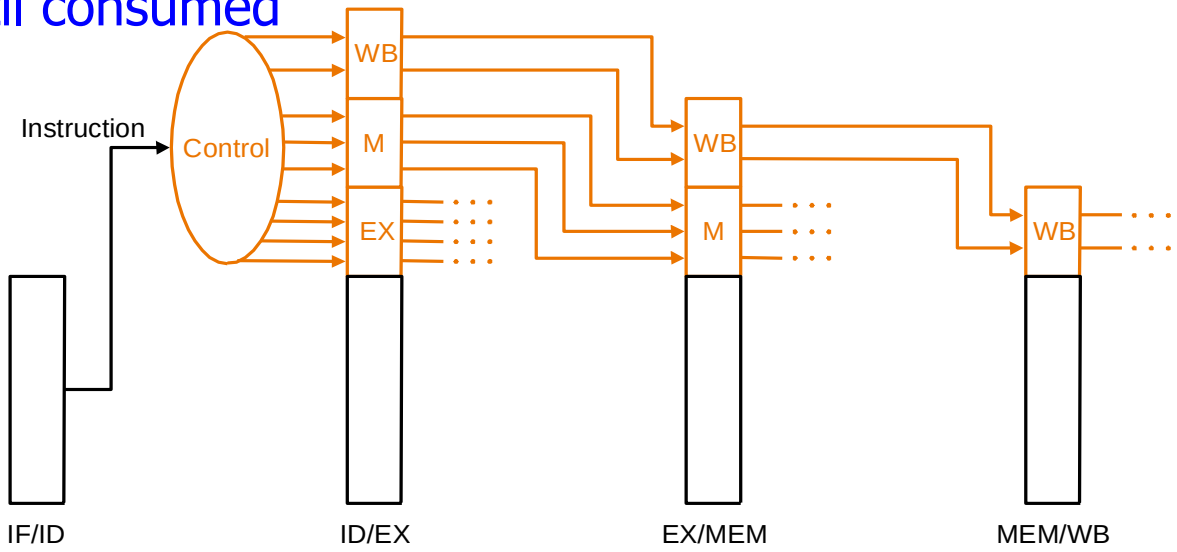


Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

Identical set of control points as the single-cycle datapath!!

Control Signals in a Pipeline

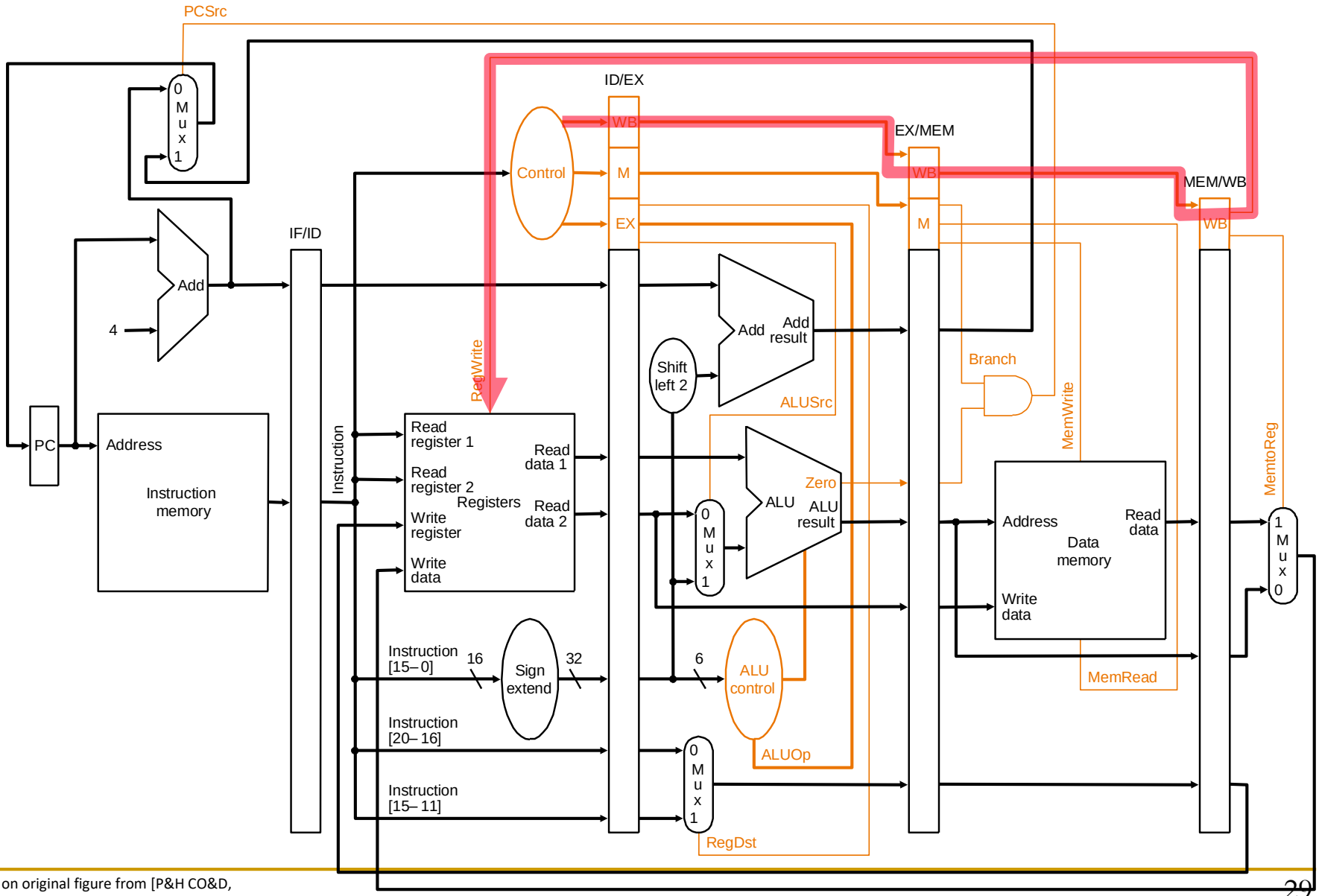
- For a given instruction
  - same control signals as single-cycle, but
  - control signals required at different cycles, depending on stage
  - ⇒ Option 1: decode once using the same logic as single-cycle and buffer signals until consumed



  - ⇒ Option 2: carry relevant "instruction word/field" down the pipeline and decode locally within each or in a previous stage

Which one is better?

Pipelined Control Signals



Remember: 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 (e.g., all laundry loads go through the same steps)
- Repetition of independent operations
  - No dependencies between repeated operations
- Uniformly partitionable suboperations
  - Processing an be evenly divided into uniform-latency suboperations (that do not share resources)
- Fitting examples: automobile assembly line, doing laundry

Instruction Pipeline: Not An Ideal Pipeline

- Identical operations ... NOT!
  - ⇒ different instructions → not all need the same stages
    Forcing different instructions to go through the same pipe stages
  - → external fragmentation (some pipe stages idle for some instructions)

- Uniform suboperations ... NOT!
  - ⇒ different pipeline stages → not the same latency
    Need to force each stage to be controlled by the same clock
  - → internal fragmentation (some pipe stages are too fast but all take the same clock cycle time)

- Independent operations ... NOT!
  - ⇒ instructions are not independent of each other
    Need to detect and resolve inter-instruction dependencies to ensure the pipeline provides correct results
  - → pipeline stalls (pipeline is not always moving)

Issues in Pipeline Design

- Balancing work in pipeline stages
  - How many stages and what is done in each stage
- Keeping the pipeline correct, moving, and full in the presence of events that disrupt pipeline flow
  - Handling dependences
    - Data
    - Control
  - Handling resource contention
  - Handling long-latency (multi-cycle) operations
- Handling exceptions, interrupts
- Advanced: Improving pipeline throughput
  - Minimizing *stalls*

Causes of Pipeline *Stalls*

- Stall: A condition when the pipeline stops moving
- Resource contention
- Dependences (between instructions)
  - Data
  - Control
- Long-latency (multi-cycle) operations

Dependences and Their Types

- Also called "dependency" or *less desirably* "hazard"
- Dependences dictate ordering requirements between instructions
- Two types
  - Data dependence
  - Control dependence
- Resource contention is sometimes called resource dependence
  - However, this is not fundamental to (dictated by) program semantics, so we will treat it separately

Handling Resource Contention

- Happens when instructions in two pipeline stages need the same resource

- Solution 1: Eliminate the cause of contention
  - Duplicate the resource or increase its throughput
    - E.g., use separate instruction and data memories (caches)
    - E.g., use multiple ports for memory structures

- Solution 2: Detect the resource contention and stall one of the contending stages
  - Which stage do you stall?
  - Example: What if you had a single read and write port for the register file?

Data Dependences

- Types of data dependences
  - Flow dependence (true data dependence – read after write)
  - Output dependence (write after write)
  - Anti dependence (write after read)

- Which ones cause stalls in a pipelined machine?
  - For all of them, we need to ensure semantics of the program is correct
  - Flow dependences always need to be obeyed because they constitute true dependence on a value
  - Anti and output dependences exist due to limited number of architectural registers
    - They are dependence on a name, not a value
    - We will later see what we can do about them

Data Dependence Types

Flow dependence

$r_3 \leftarrow r_1$ op r_2

$r_5 \leftarrow r_3$ op r_4

Read-after-Write (RAW)

Anti dependence

$r_3 \leftarrow r_1$ op r_2

$r_1 \leftarrow r_4$ op r_5

Write-after-Read (WAR)

Output-dependence

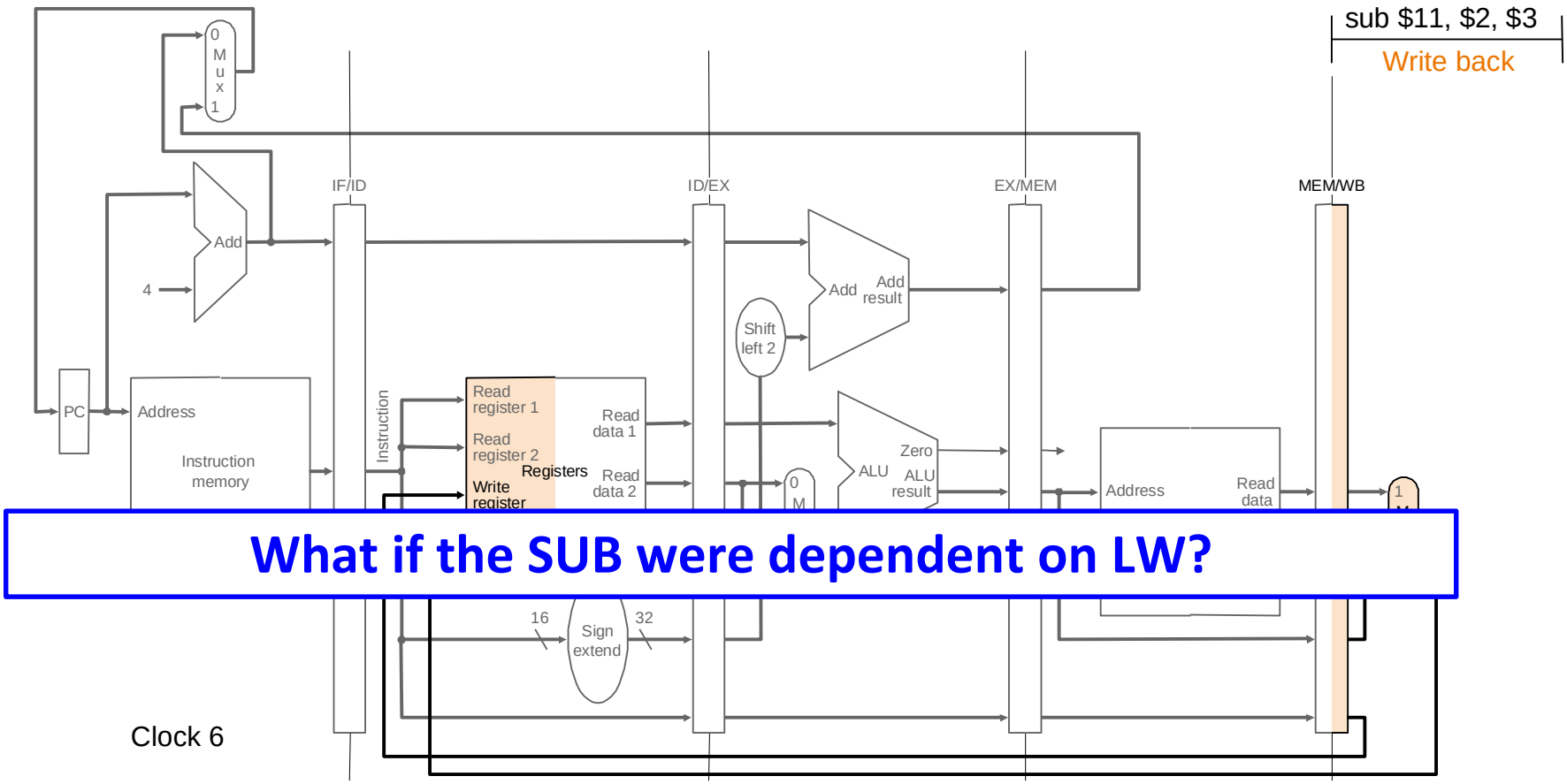
$r_3 \leftarrow r_1$ op r_2

$r_5 \leftarrow r_3$ op r_4

$r_3 \leftarrow r_6$ op r_7

Write-after-Write (WAW)

Pipelined Operation Example



What if the SUB were dependent on LW?

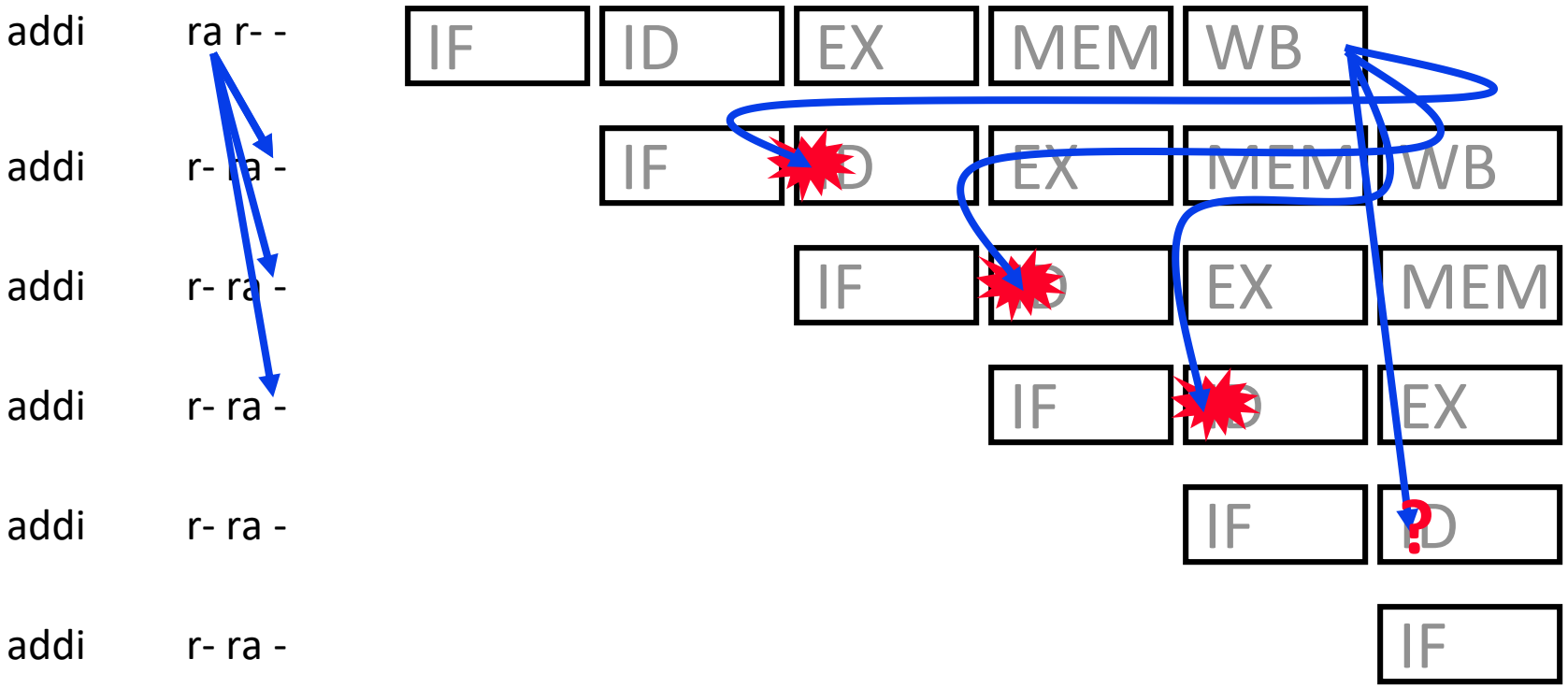
Data Dependence Handling

How to Handle Data Dependences

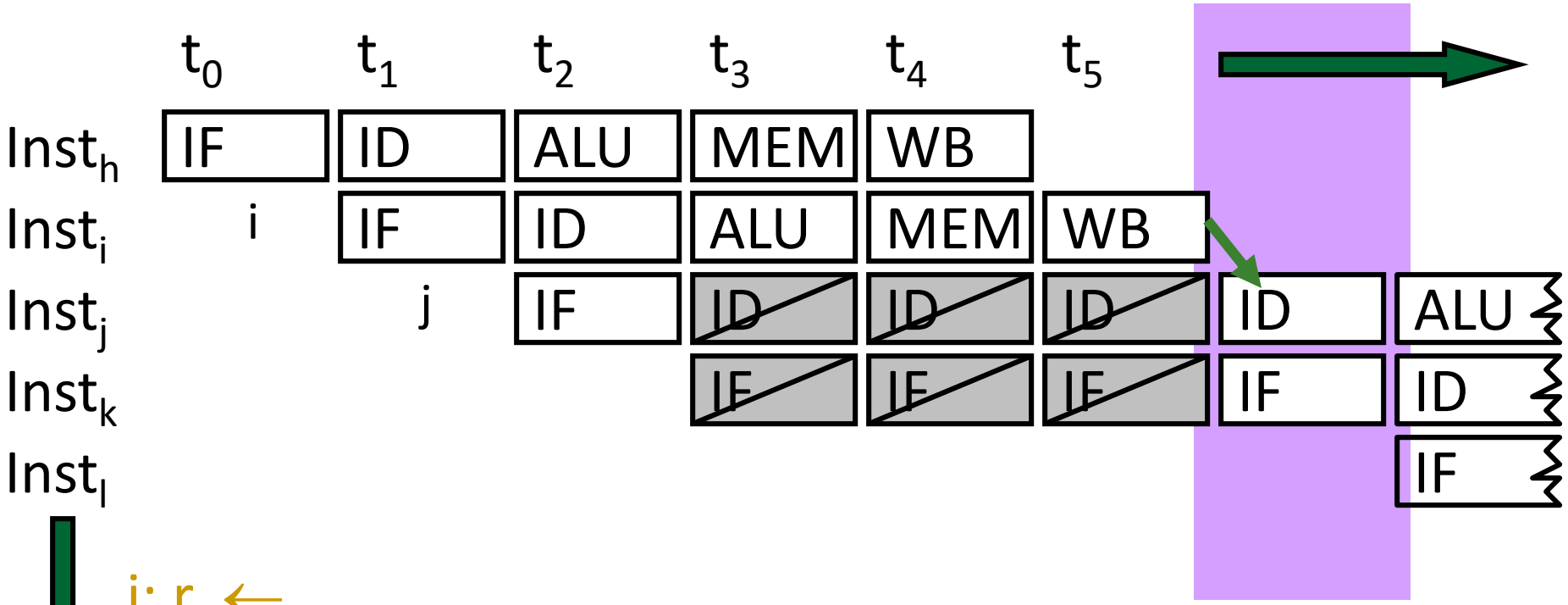
- Anti and output dependences are easier to handle
  - write to the destination in one stage and in program order
- Flow dependences are more interesting
- Five fundamental ways of handling flow dependences
  - Detect and wait until value is available in register file
  - Detect and forward/bypass data to dependent instruction
  - Detect and eliminate the dependence at the software level
    - No need for the hardware to detect dependence
  - Predict the needed value(s), execute "speculatively", and verify
  - Do something else (fine-grained multithreading)
    - No need to detect

RAW Dependence Handling

- Which one of the following flow dependences lead to conflicts in the 5-stage pipeline?



Pipeline Stall: Resolving Data Dependence



i: $r_x \leftarrow$ _
bubble
bubble
bubble
j: _ $\leftarrow r_x$

dist(i,j)=4

Stall = make the dependent instruction wait until its source data value is available

1. stop all up-stream stages
2. drain all down-stream stages

Approaches to Dependence Detection (I)

- Scoreboarding
  - Each register in register file has a Valid bit associated with it
  - An instruction that is writing to the register resets the Valid bit
  - An instruction in Decode stage checks if all its source and destination registers are Valid
    - Yes: No need to stall… No dependence
    - No: Stall the instruction

- Advantage:
  - Simple. 1 bit per register

- Disadvantage:
  - Need to stall for all types of dependences, not only flow dep.

Approaches to Dependence Detection (II)

- Combinational dependence check logic
  - Special logic that checks if any instruction in later stages is supposed to write to any source register of the instruction that is being decoded
  - Yes: stall the instruction/pipeline
  - No: no need to stall… no flow dependence

- Advantage:
  - No need to stall on anti and output dependences

- Disadvantage:
  - Logic is more complex than a scoreboard
  - Logic becomes more complex as we make the pipeline deeper and wider

Once You Detect the Dependence in Hardware

- What do you do afterwards?

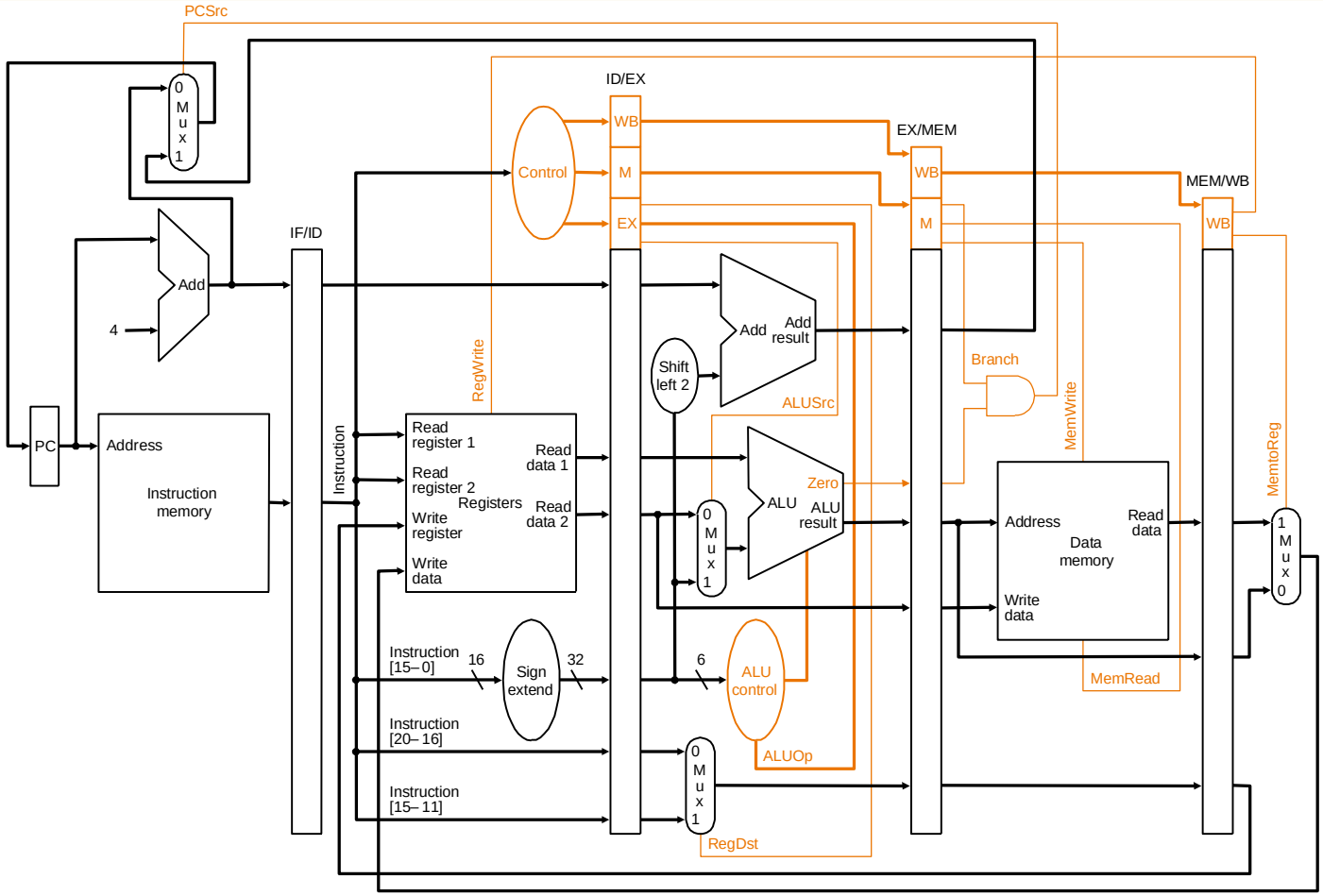
- Observation: Dependence between two instructions is detected before the communicated data value becomes available

- Option 1: Stall the dependent instruction right away
- Option 2: Stall the dependent instruction only when necessary → data forwarding/bypassing
- Option 3: …

Data Forwarding/Bypassing

- Problem: A consumer (dependent) instruction has to wait in decode stage until the producer instruction writes its value in the register file
- Goal: We do not want to stall the pipeline unnecessarily
- Observation: The data value needed by the consumer instruction can be supplied directly from a later stage in the pipeline (instead of only from the register file)
- Idea: Add additional dependence check logic and data forwarding paths (buses) to supply the producer's value to the consumer right after the value is available
- Benefit: Consumer can move in the pipeline until the point the value can be supplied → less stalling

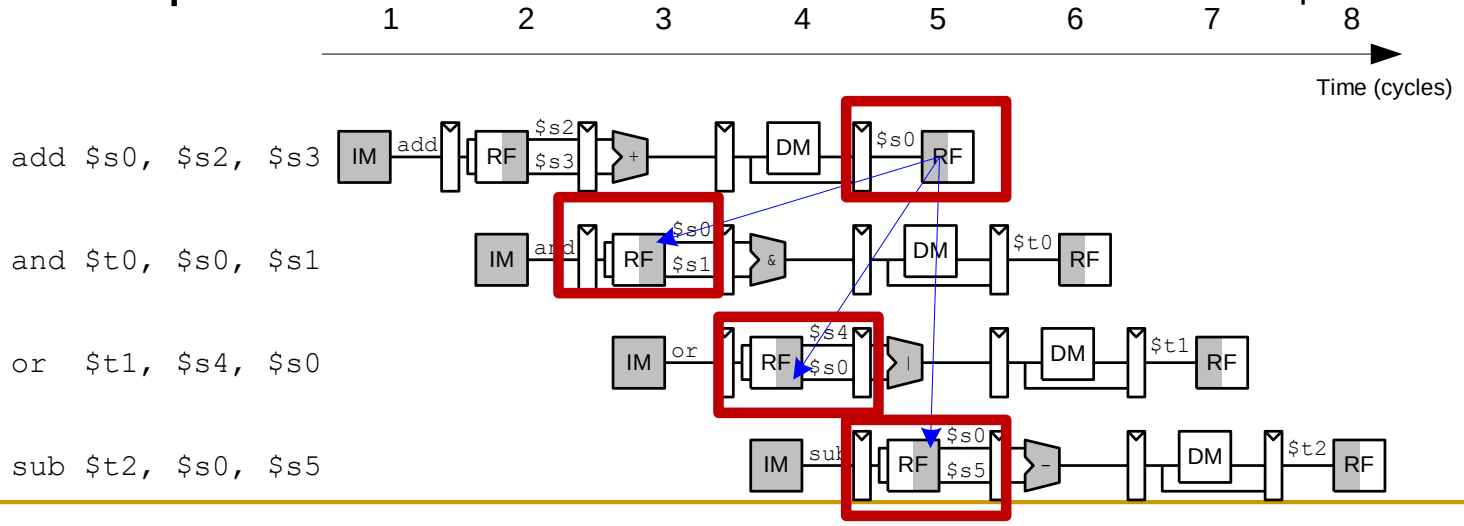
How to Implement Stalling



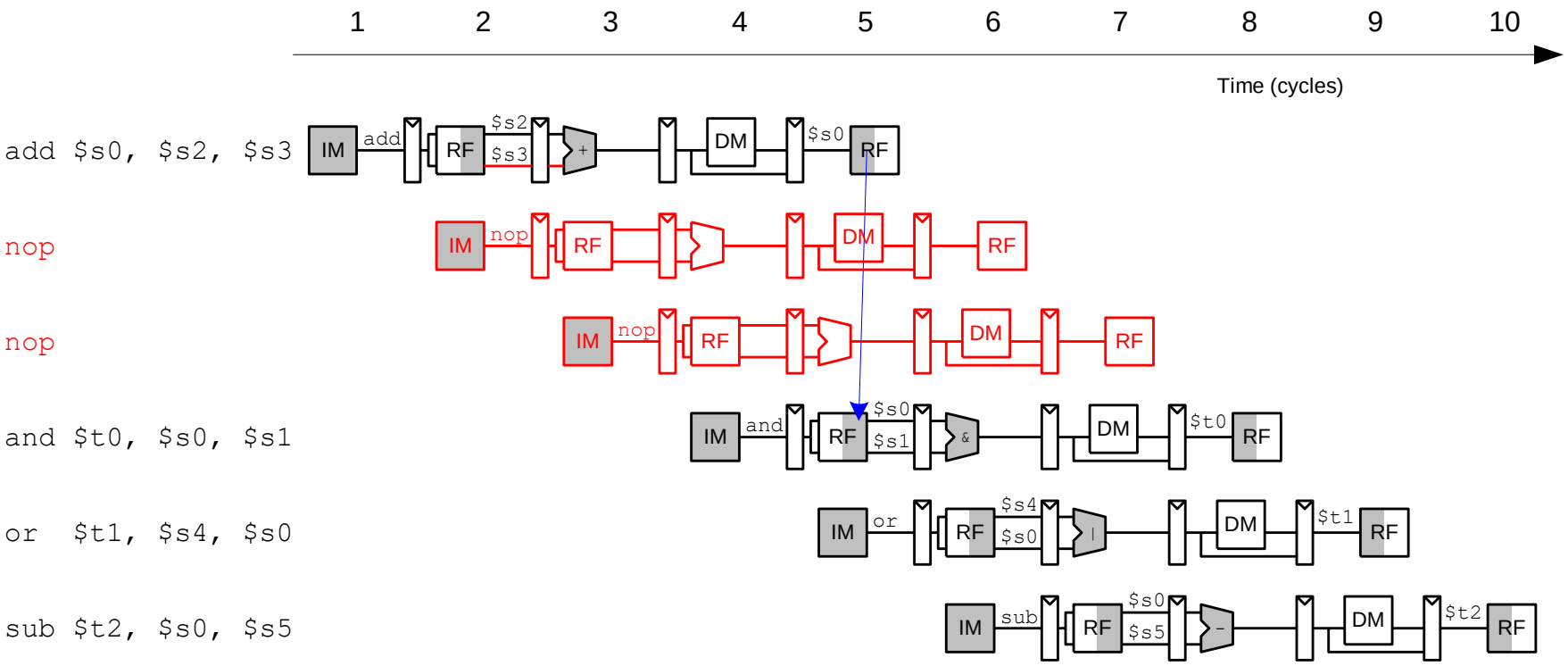
- **Stall**
  - disable **PC** and **IF/ID** latching; ensure stalled instruction stays in its stage
  - Insert **"invalid"** instructions/nops into the stage following the stalled one (called "**bubbles**")

RAW Data Dependence Example

- One instruction writes a register ($s0) and next instructions read this register => read after write (RAW) dependence.
  - ***add*** writes into $s0 in the first half of cycle 5
  - ***and*** reads $s0 on cycle 3, obtaining the wrong value
  - ***or*** reads $s0 on cycle 4, again obtaining the wrong value.
  - ***sub*** reads $s0 in the second half of cycle 5, obtaining the correct value
  - subsequent instructions read the correct value of $s0



Compile-Time Detection and Elimination

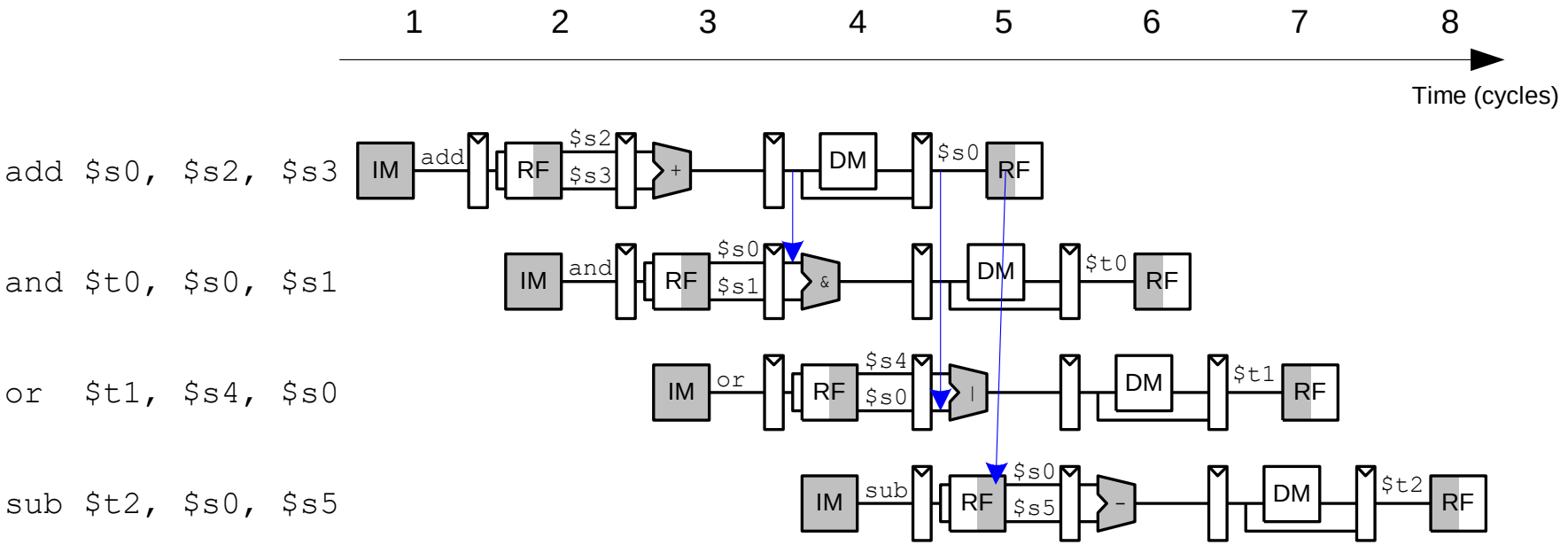


- Insert enough NOPs for the required result to be ready
- Or (if you can) move independent useful instructions up

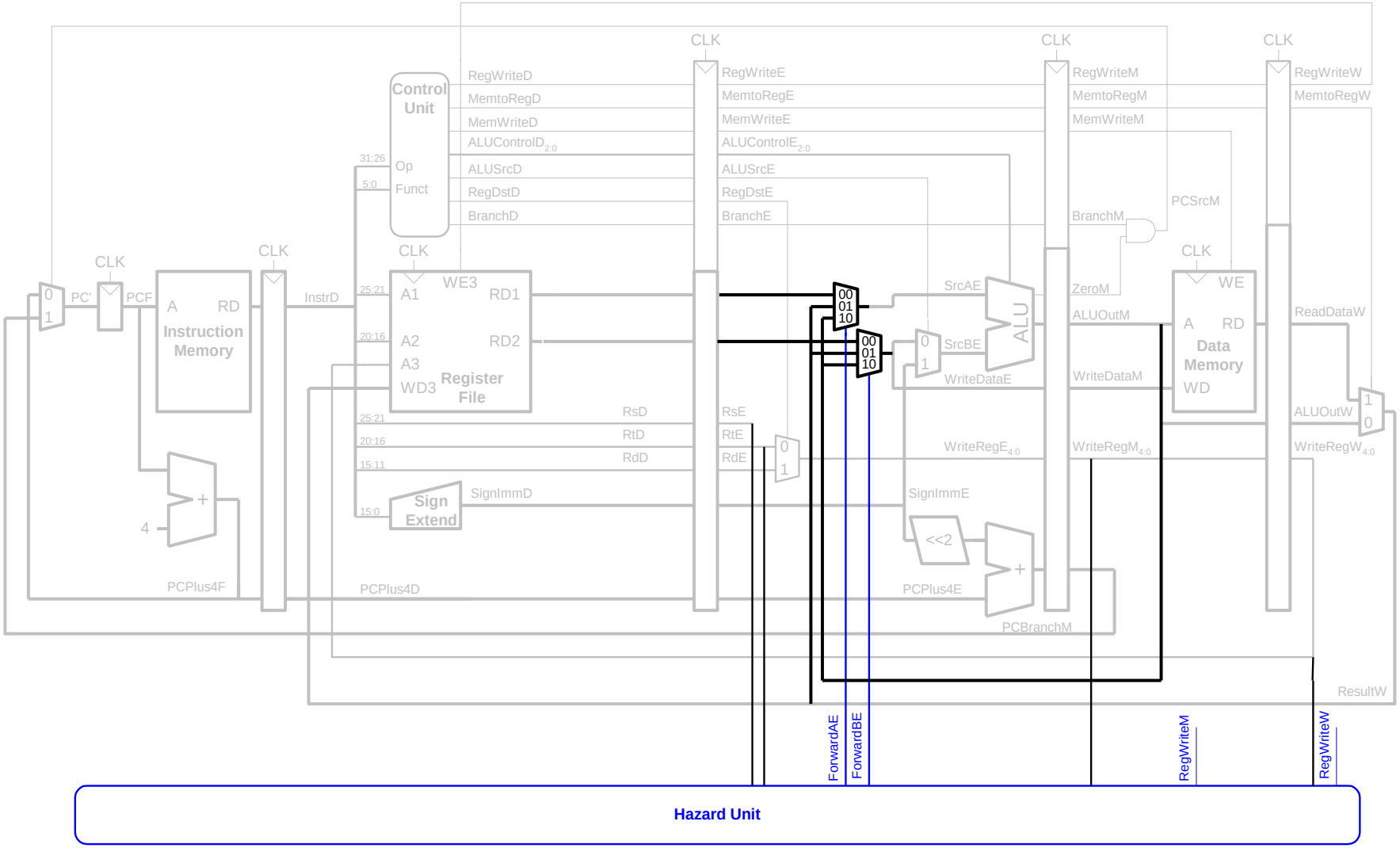
Data Forwarding

- Also called Data Bypassing
- We have already seen the basic idea before
- Forward the result value to the dependent instruction as soon as the value is available
- Remember dataflow?
  - Data value supplied to dependent instruction as soon as it is available
  - Instruction executes when all its operands are available
- Data forwarding brings a pipeline closer to data flow execution principles

Data Forwarding



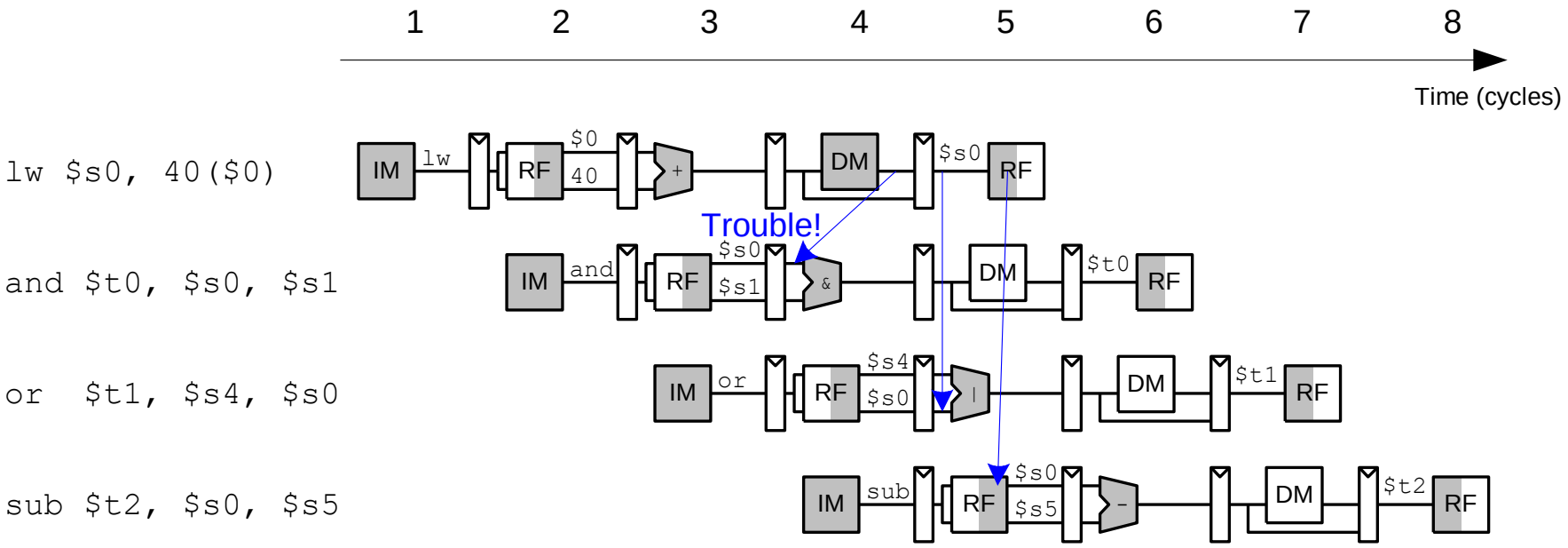
Data Forwarding



Data Forwarding

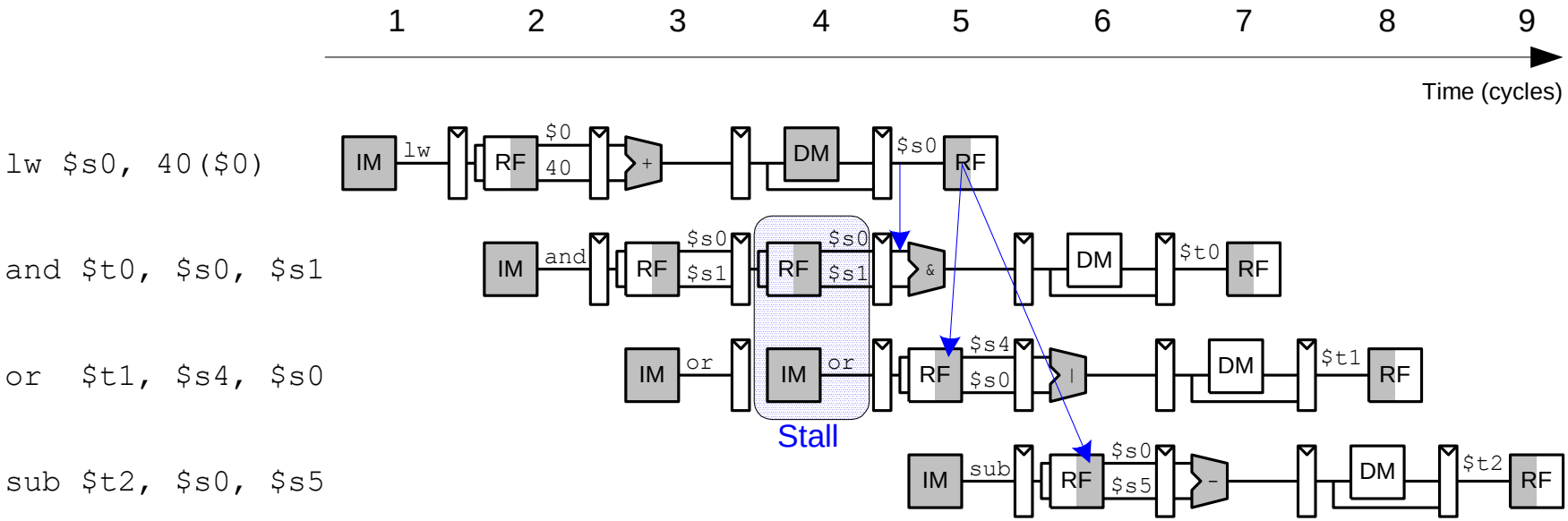
- Forward to Execute stage from either:
  - Memory stage or
  - Writeback stage
- When should we forward from one either Memory or Writeback stage?
  - If that stage will write a destination register and the destination register matches the source register.
  - If both the Memory and Writeback stages contain matching destination registers, the Memory stage should have priority, because it contains the more recently executed instruction.

Stalling



- Forwarding is sufficient to resolve RAW data dependences
- *but …There are cases when forwarding is not possible due to pipeline design and instruction latencies*
- The `lw` instruction *does not finish* reading data until the end of the Memory stage,
- Therefore its result *cannot be forwarded* to the Execute stage of the next instruction.

Stalling


1
2
3
4
5
6
7
8
9
Time (cycles)
lw $s0, 40($0)
and $t0, $s0, $s1
or $t1, $s4, $s0
sub $t2, $s0, $s5
IM
lw
RF
$0
40
+
DM
$s0
RF
IM
and
RF
$s0
$s1
RF
$s0
$s1
&
DM
$t0
RF
IM
or
IM
or
RF
$s4
$s0
|
DM
$t1
RF
Stall
IM
sub
RF
$s0
$s5
-
DM
$t2
RF

How to Handle Data Dependences

- Anti and output dependences are easier to handle
  - write to the destination in one stage and in program order
- Flow dependences are more interesting
- Five fundamental ways of handling flow dependences
  - Detect and wait until value is available in register file
  - Detect and forward/bypass data to dependent instruction
  - Detect and eliminate the dependence at the software level
    - No need for the hardware to detect dependence
  - Predict the needed value(s), execute "speculatively", and verify
  - Do something else (fine-grained multithreading)
    - No need to detect

Fine-Grained Multithreading

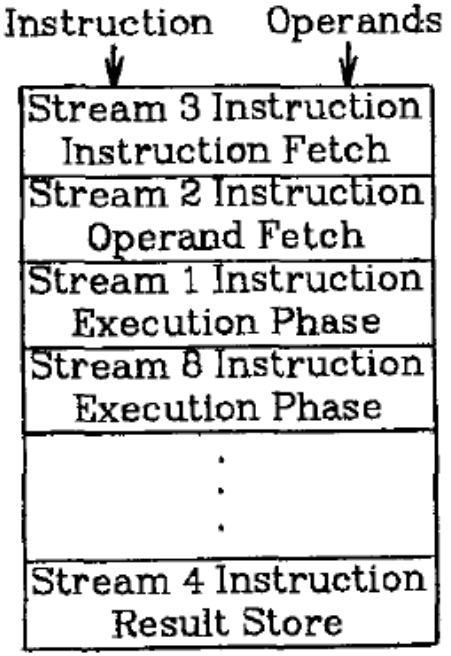
- Idea: Hardware has multiple thread contexts (PC+registers). Each cycle, fetch engine fetches from a different thread.
  - By the time the fetched branch/instruction resolves, no instruction is fetched from the same thread
  - Branch/instruction resolution latency overlapped with execution of other threads' instructions

+ No logic needed for handling control and data dependences within a thread

-- Single thread performance suffers

-- Extra logic for keeping thread contexts

-- Does not overlap latency if not enough threads to cover the whole pipeline



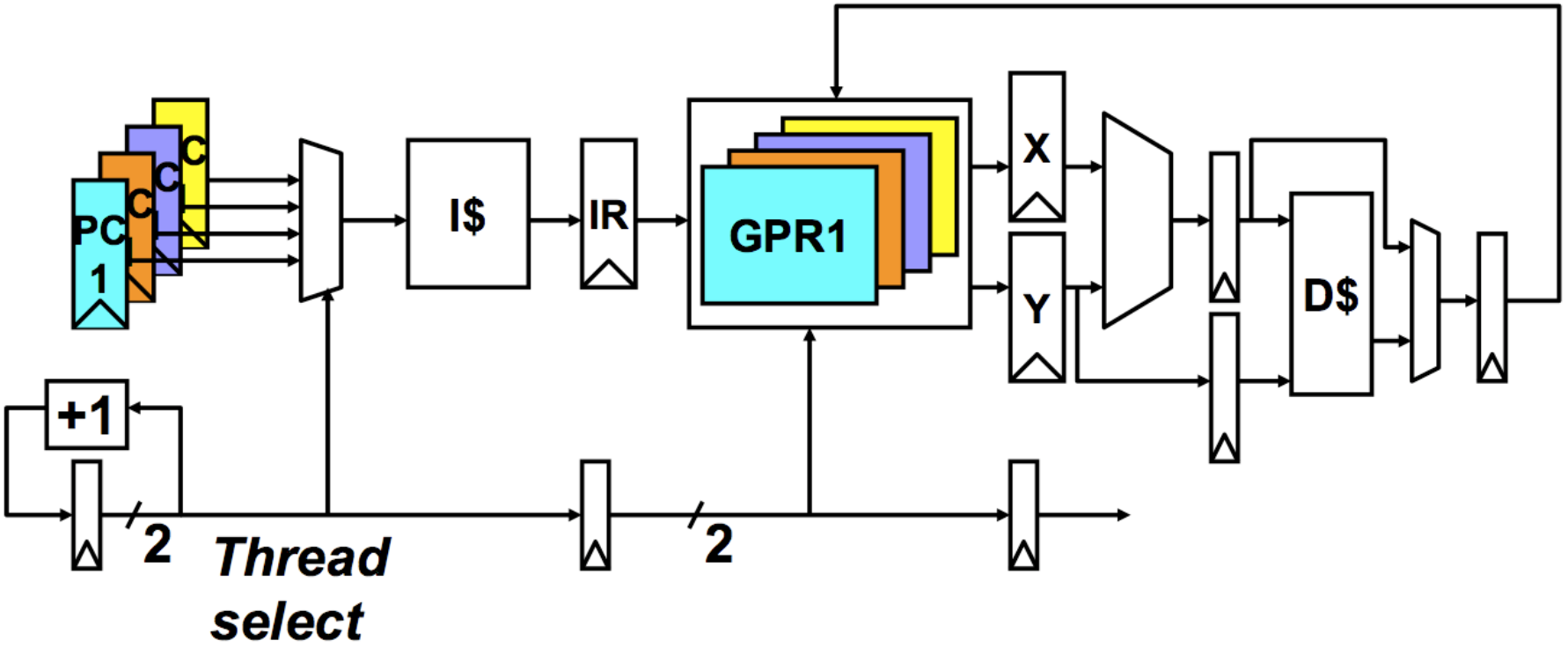
Fine-Grained Multithreading (II)

- Idea: Switch to another thread every cycle such that no two instructions from a thread are in the pipeline concurrently

- Tolerates the control and data dependency latencies by overlapping the latency with useful work from other threads
- Improves pipeline utilization by taking advantage of multiple threads

- Thornton, "Parallel Operation in the Control Data 6600," AFIPS 1964.
- Smith, "A pipelined, shared resource MIMD computer," ICPP 1978.

Multithreaded Pipeline Example



Slide credit: Joel Emer

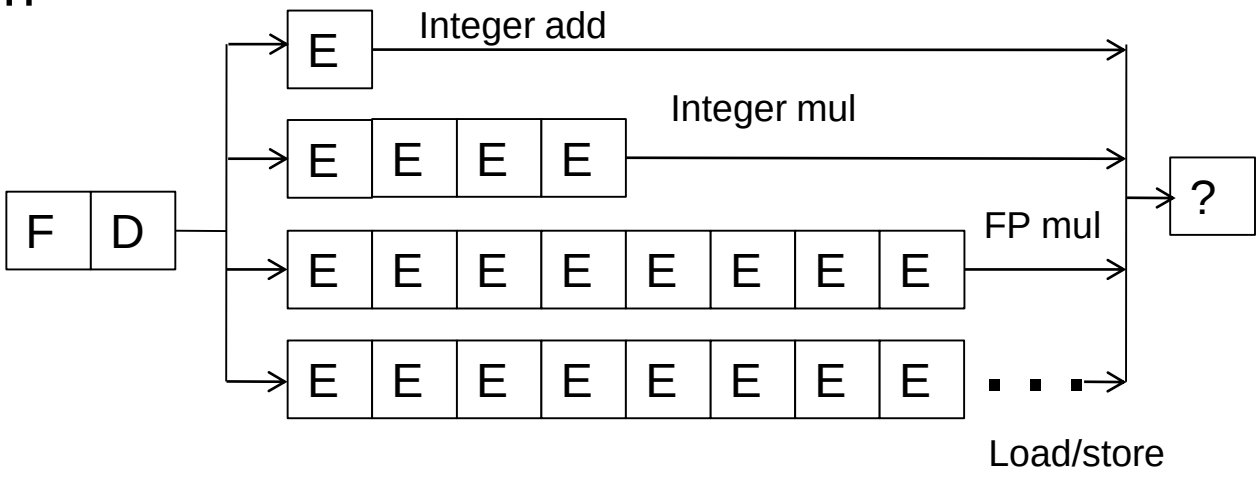
Fine-grained Multithreading

- Advantages
  - + No need for dependency checking between instructions (only one instruction in pipeline from a single thread)
  - + No need for branch prediction logic
  - + Otherwise-bubble cycles used for executing useful instructions from different threads
  - + Improved system throughput, latency tolerance, utilization

- Disadvantages
  - - Extra hardware complexity: multiple hardware contexts (PCs, register files, …), thread selection logic
  - - Reduced single thread performance (one instruction fetched every N cycles from the same thread)
  - - Resource contention between threads in caches and memory
  - - Some dependency checking logic *between* threads remains (load/store)

Multi-Cycle Execution

- Not all instructions take the same amount of time for "execution"
- Idea: Have multiple different functional units that take different number of cycles
  - Can be pipelined or not pipelined
  - Can let independent instructions start execution on a different functional unit before a previous long-latency instruction finishes execution

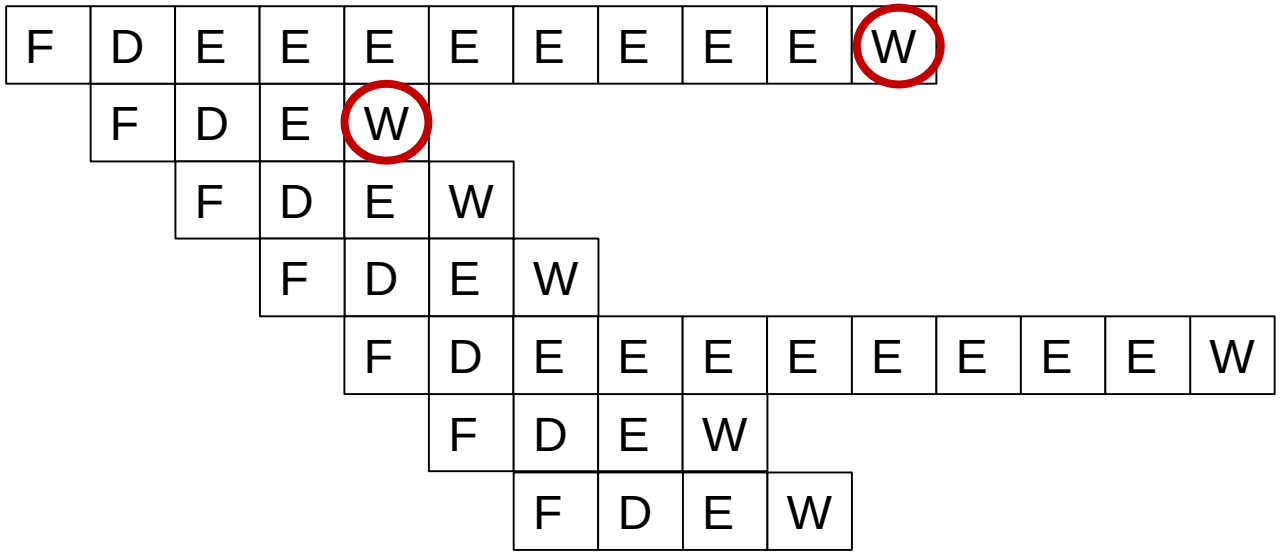


Issues in Pipelining: Multi-Cycle Execute

- Instructions can take different number of cycles in EXECUTE stage
  - Integer ADD versus FP MULtiply

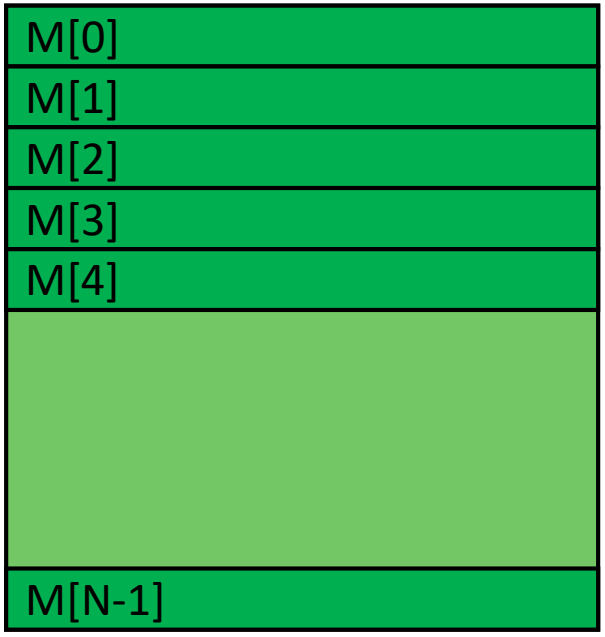
FMUL R4 ← R1, R2
ADD R3 ← R1, R2

FMUL R2 ← R5, R6
ADD R7 ← R5, R6



  - What is wrong with this picture in a Von Neumann architecture?
    - Sequential semantics of the ISA NOT preserved!
    - What if FMUL incurs an exception?

Programmer Visible (Architectural) State



Memory
array of storage locations
indexed by an address

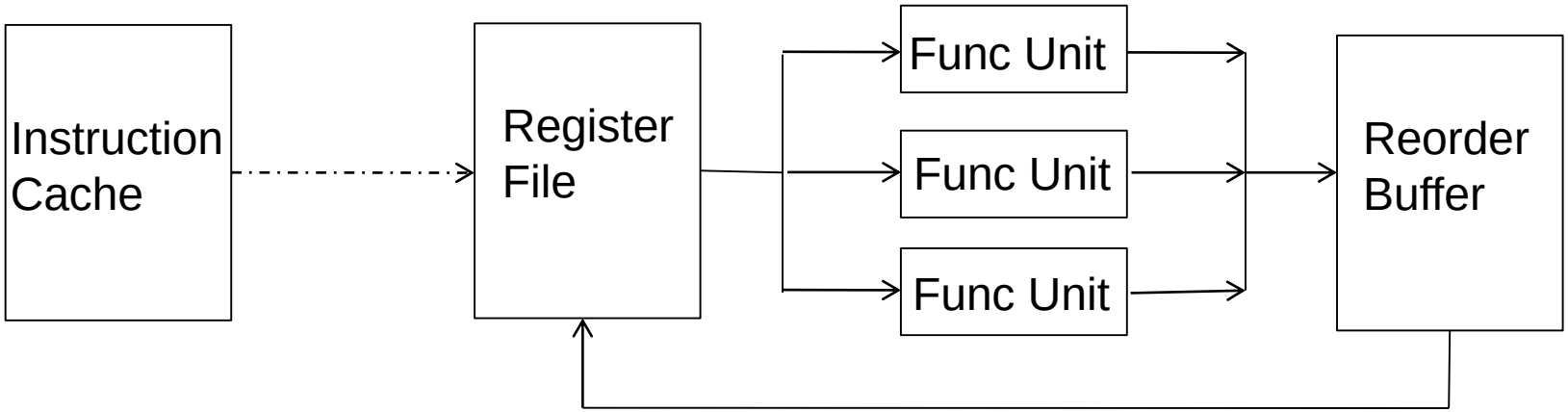
Registers
- given special names in the ISA
  (as opposed to addresses)
- general vs. special purpose

Program Counter
memory address
of the current instruction

Instructions (and programs) specify how to transform
the values of programmer visible state

Reorder Buffer (ROB)

- Idea: Complete instructions out-of-order, but reorder them before making results visible to architectural state
- When instruction is decoded it reserves the next-sequential entry in the ROB
- When instruction completes, it writes result into ROB entry
- When instruction oldest in ROB and it has completed without exceptions, its result moved to reg. file or memory



Reorder Buffer

- Buffers information about all instructions that are decoded but not yet retired/committed

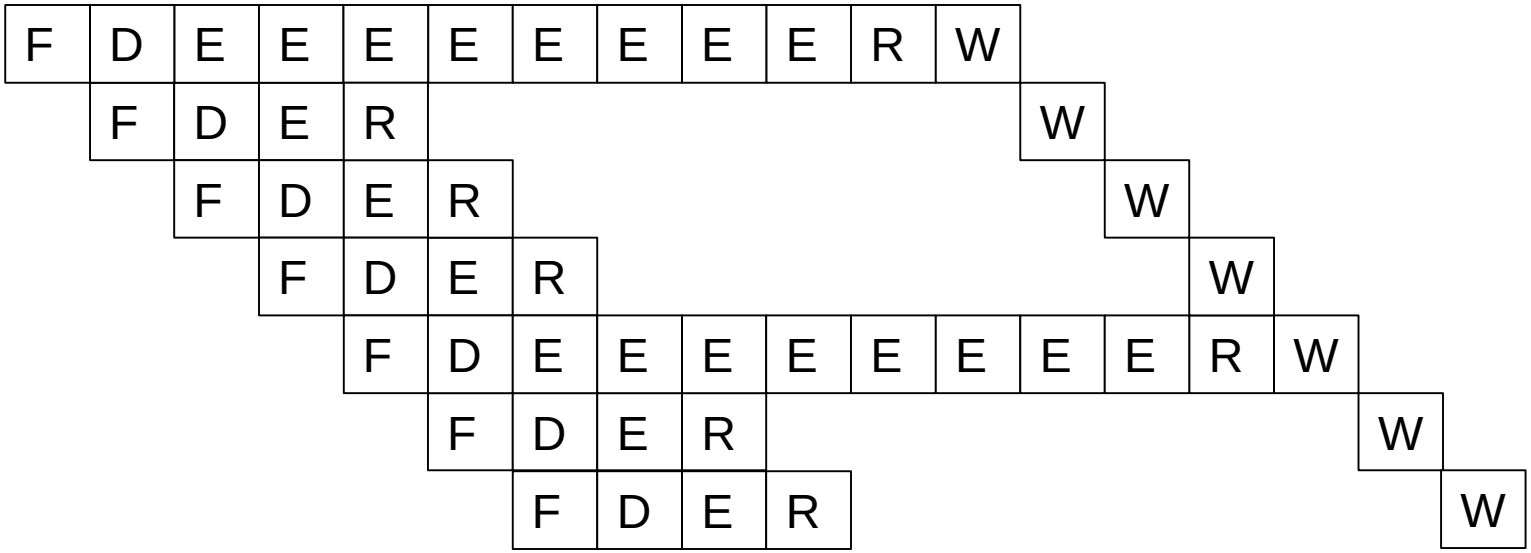
What's in a ROB Entry?

V	DestRegID	DestRegVal	StoreAddr	StoreData	PC	Valid bits for reg/data + control bits	Exception?

- Everything required to:
  - correctly reorder instructions back into the program order
  - update the architectural state with the instruction's result(s), if instruction can retire without any issues
  - handle an exception/interrupt precisely, if an exception/interrupt needs to be handled before retiring the instruction
- Need valid bits to keep track of readiness of the result(s) and find out if the instruction has completed execution

Reorder Buffer: Independent Operations

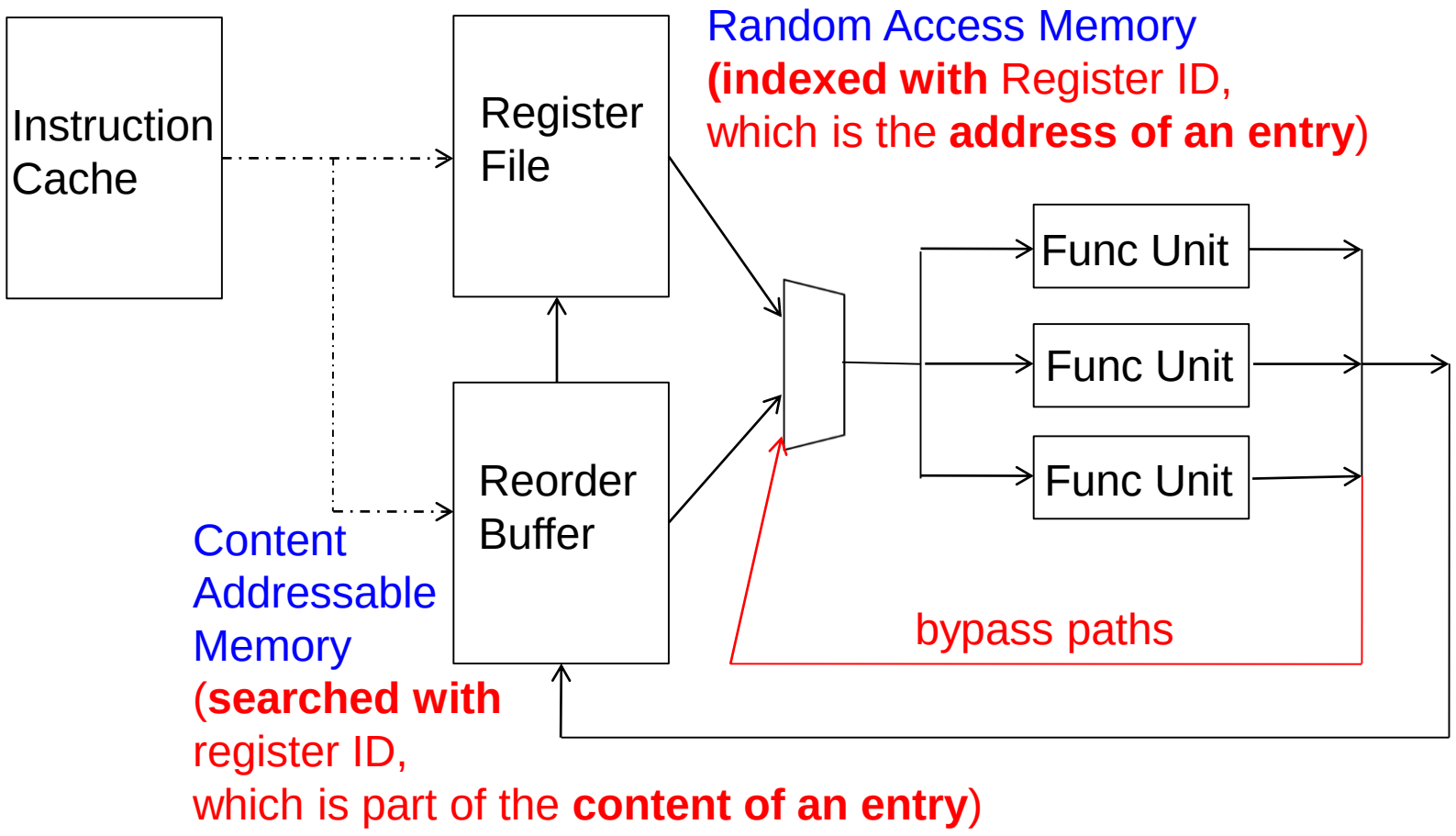
- Result first written to ROB on instruction completion
- Result written to register file at commit time



- What if a later instruction needs a value in the reorder buffer?
  - One option: stall the operation → stall the pipeline
  - Better: Read the value from the reorder buffer. How?

Reorder Buffer: How to Access?

- A register value can be in the register file, reorder buffer, (or bypass/forwarding paths)



Simplifying Reorder Buffer Access

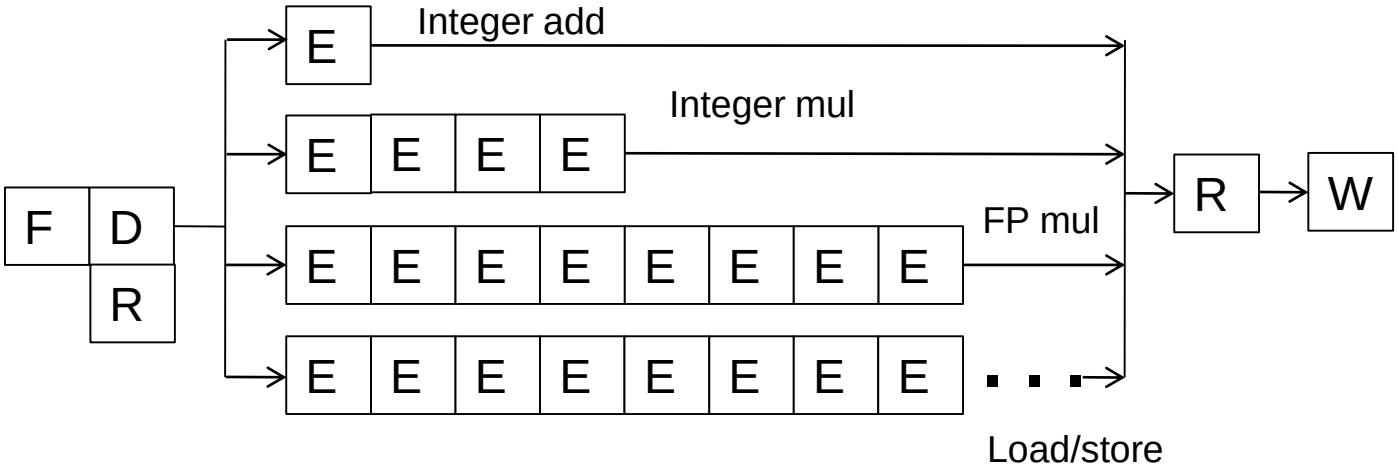
- Idea: Use indirection

- Access register file first (check if the register is valid)
  - If register not valid, register file stores the ID of the reorder buffer entry that contains (or will contain) the value of the register
  - Mapping of the register to a ROB entry: Register file maps the register to a reorder buffer entry if there is an in-flight instruction writing to the register

- Access reorder buffer next

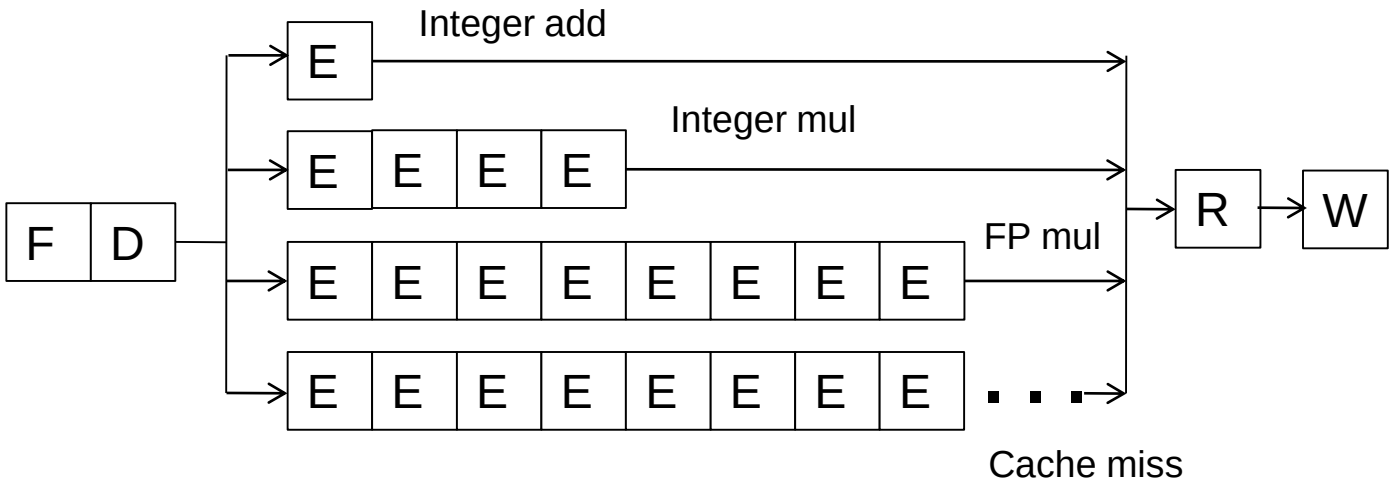
In-Order Pipeline with Reorder Buffer

- Decode (D): Access regfile/ROB, allocate entry in ROB, check if instruction can execute, if so **dispatch** instruction
- Execute (E): Instructions can complete out-of-order
- Completion (R): Write result to reorder buffer
- Retirement/Commit (W): Check for exceptions; if none, write result to architectural register file or memory; else, flush pipeline and start from exception handler
- In-order dispatch/execution, out-of-order completion, in-order retirement



Out-of-Order Execution (Dynamic Instruction Scheduling)

An In-order Pipeline



- Dispatch: Act of sending an instruction to a functional unit
- Renaming with ROB eliminates stalls due to false dependencies
- Problem: A true data dependency stalls dispatch of younger instructions into functional (execution) units

Can We Do Better?

- What do the following two pieces of code have in common (with respect to execution in the previous design)?

```
IMUL  R3 ← R1, R2
ADD   R3 ← R3, R1
ADD   R4 ← R6, R7
IMUL  R5 ← R6, R8
ADD   R7 ← R9, R9
```

```
LD    R3 ← R1 (0)
ADD   R3 ← R3, R1
ADD   R4 ← R6, R7
IMUL  R5 ← R6, R8
ADD   R7 ← R9, R9
```

- Answer: First ADD stalls the whole pipeline!
  - ADD cannot dispatch because its source registers unavailable
  - Later **independent** instructions cannot get executed

- How are the above code portions different?
  - Answer: Load latency is variable (unknown until runtime)
  - What does this affect? Think compiler vs. microarchitecture

Preventing Dispatch Stalls

- Problem: **in-order** dispatch (scheduling, or execution)
- Solution: **out-of-order** dispatch (scheduling, or execution)
- Actually, we have seen the basic idea before:
  - **Dataflow**: "fire" an instruction only when its inputs are ready
  - We will use similar principles, but not expose it in the ISA
- Aside: Any other way to prevent dispatch stalls?
  1. Compile-time instruction scheduling/reordering
  2. Value prediction
  3. Fine-grained multithreading

Out-of-order Execution (Dynamic Scheduling)

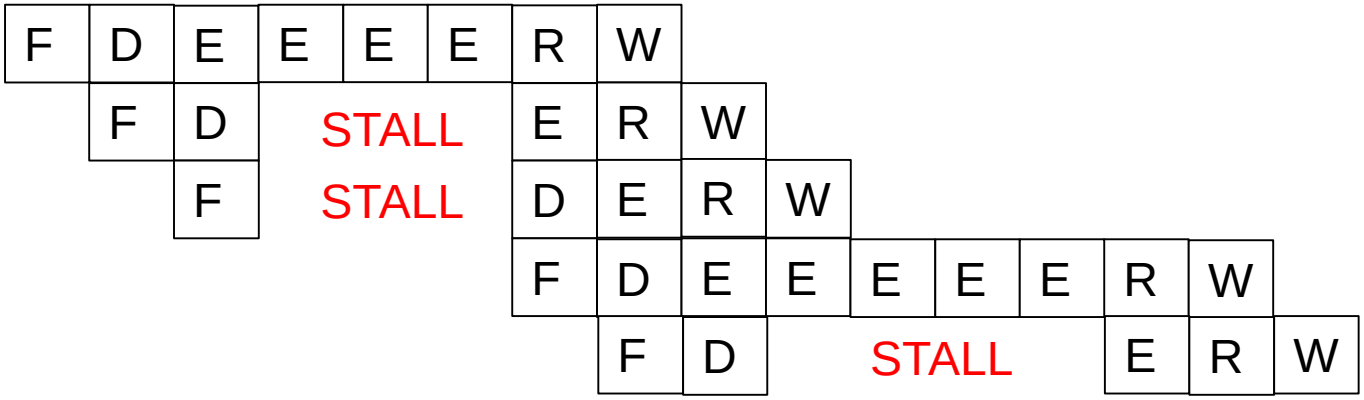
- Idea: Move the dependent instructions out of the way of independent ones (s.t. independent ones can execute)
  - Rest areas for dependent instructions: Reservation stations

- Monitor the source "values" of each instruction in the resting area
- When all source "values" of an instruction are available, "fire" (i.e. dispatch) the instruction
  - Instructions dispatched in dataflow (not control-flow) order

- Benefit:
  - Latency tolerance: Allows independent instructions to execute and complete in the presence of a long-latency operation

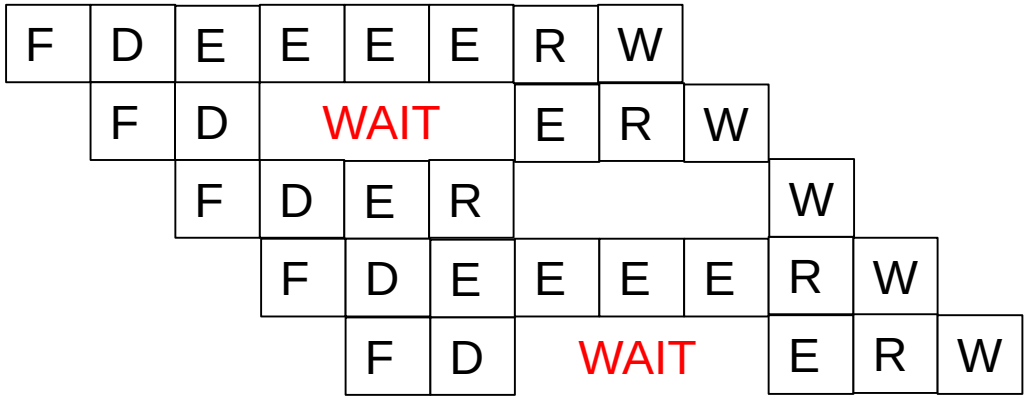
In-order vs. Out-of-order Dispatch

- In order dispatch + precise exceptions:



IMUL R3 ← R1, R2
ADD R3 ← R3, R1
ADD R1 ← R6, R7
IMUL R5 ← R6, R8
ADD R7 ← R3, R5

- Out-of-order dispatch + precise exceptions:



- 16 vs. 12 cycles