KING'S College LONDON

Hardware Design

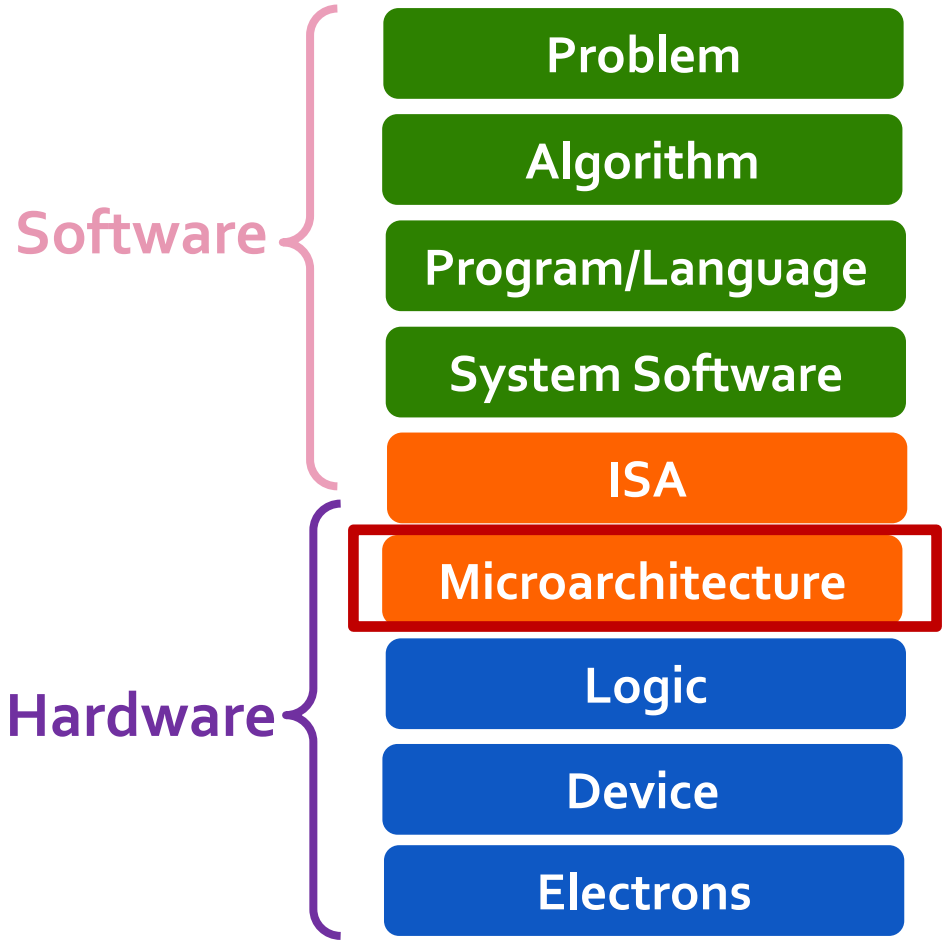
Lecture 6: Pipelined Processor Design

Dr. Haiyu Mao

05.03.2026

What We Learned & Will Learn

- The von Neumann model
- Instruction Set Architectures (ISA)
- Assembly programming: LC-3 and MIPS
- Microarchitecture: basics
- Microarchitecture: Single-cycle
- Microarchitecture: Multi-cycle
- **Pipelining**
- Cache and Memory



Recall: 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 the program order are processed in consecutive stages

- Benefit: Increases instruction processing throughput (1/CPI)
- Downside: Start thinking about this…

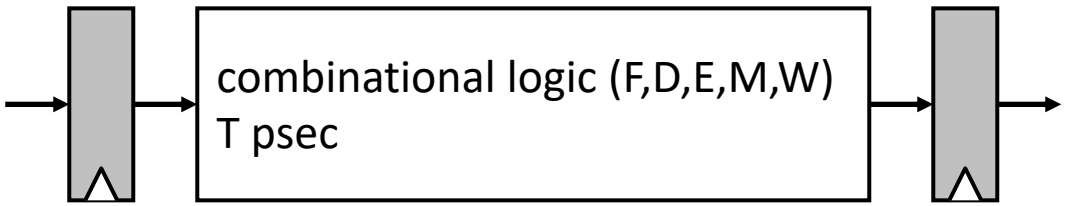
Recall: An Ideal Pipeline

- Goal: Increase throughput with little increase in cost (hardware cost, in case of instruction processing)

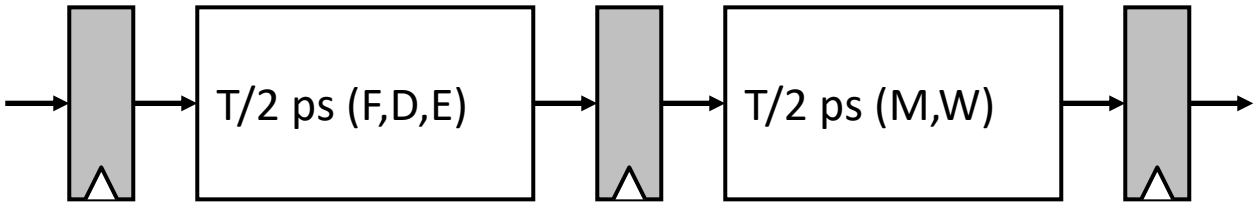
- Repetition of identical operations
  - The same operation is repeated on a large number of different inputs (e.g., all laundry loads go through the same steps)
- Repetition of independent operations
  - No dependencies between repeated operations
- Uniformly partitionable suboperations
  - Processing can be evenly divided into uniform-latency suboperations (that do not share resources)
- Fitting examples: automobile assembly line, doing laundry
  - What about the instruction processing "cycle"?

Recall: Ideal Pipelining

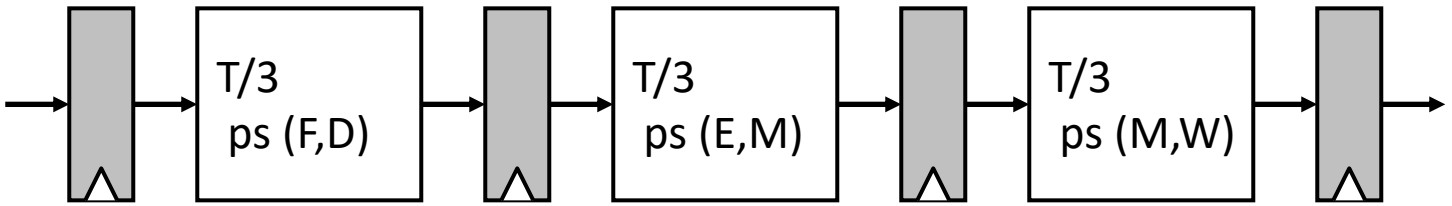
Tput = Throughput



Tput=~(1/T)



Tput=~(2/T)



Tput=~(3/T)

Recall: More Realistic Pipeline: Throughput

- Nonpipelined version with delay T

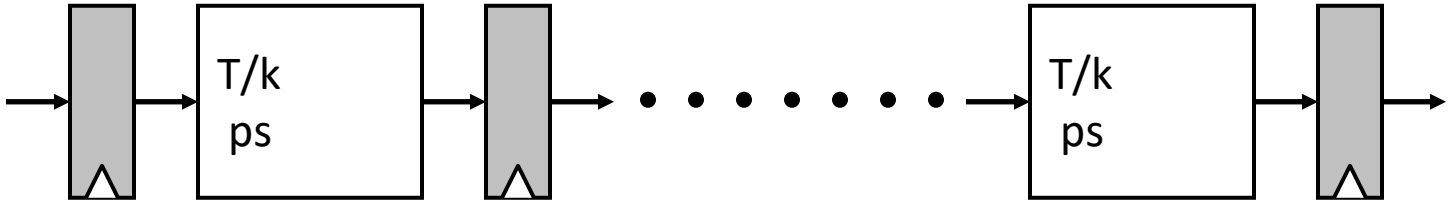
  Tput = 1 / (T+S) where S = register (sequential logic) delay

- k-stage pipelined version

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

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

Register delay reduces throughput (sequencing overhead between stages)



This picture assumes "perfect division of work between stages (T/k)"

Recall: More Realistic Pipeline: Cost

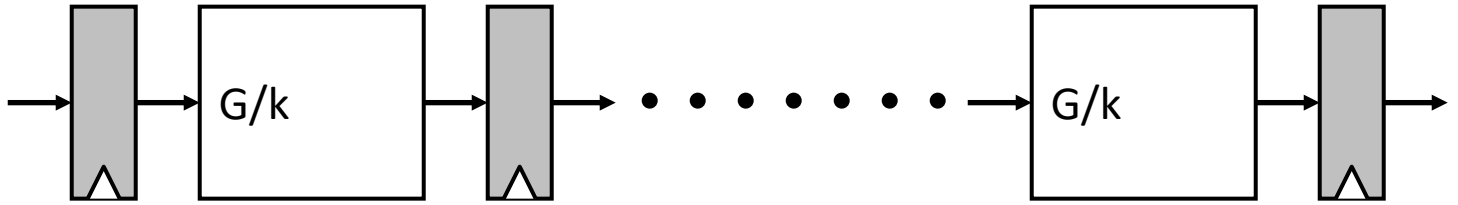
- Nonpipelined version with combinational cost G

  Cost = G+R where R = register cost

- k-stage pipelined version

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

Registers increase hardware cost

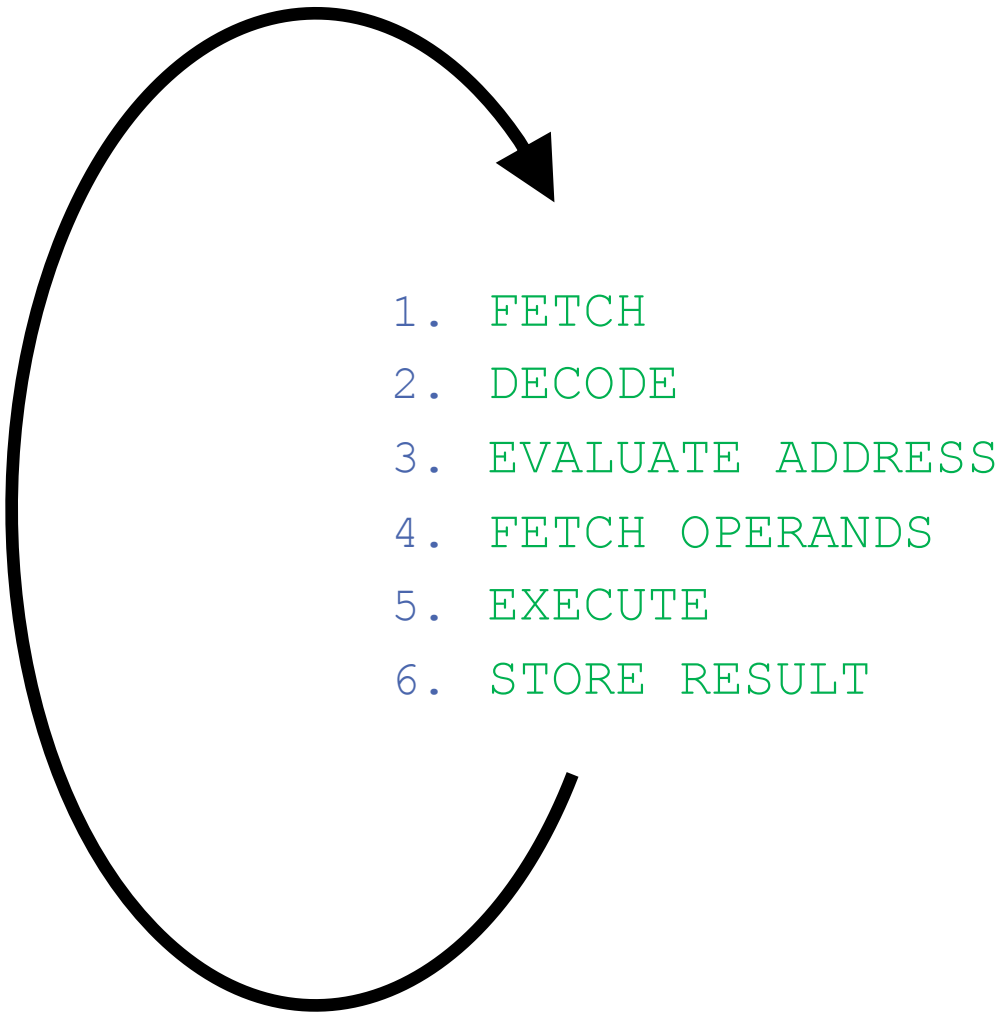


This picture ignores resource and register replication that is likely needed (G/k and Rk)

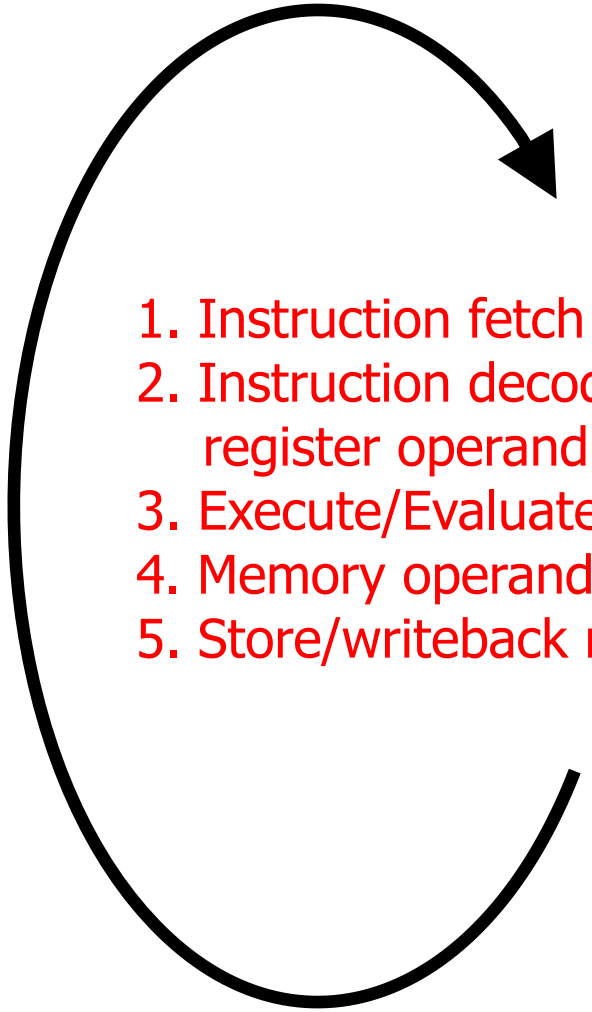
Hardware Design

Pipelining Instruction Processing

Remember: The Instruction Processing Cycle

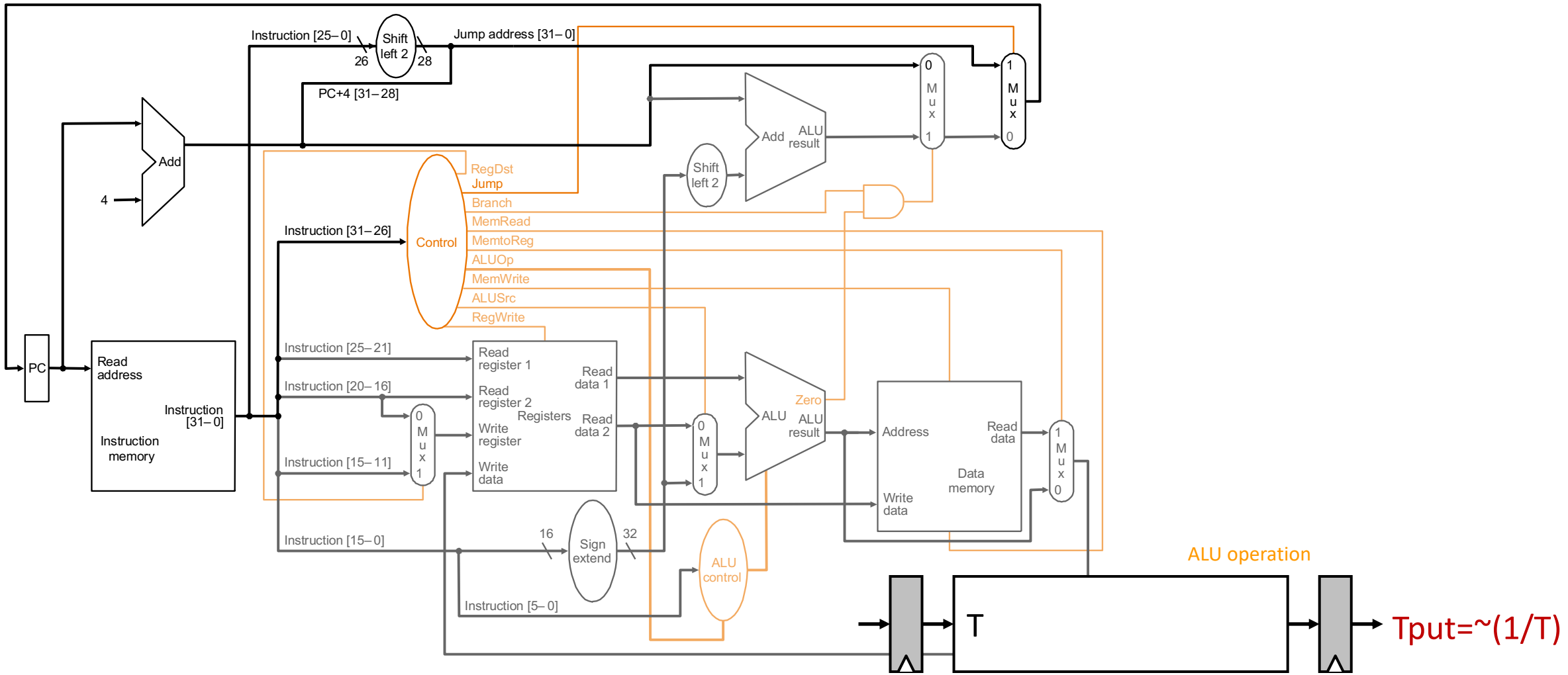


1. FETCH
2. DECODE
3. EVALUATE ADDRESS
4. FETCH OPERANDS
5. EXECUTE
6. STORE RESULT

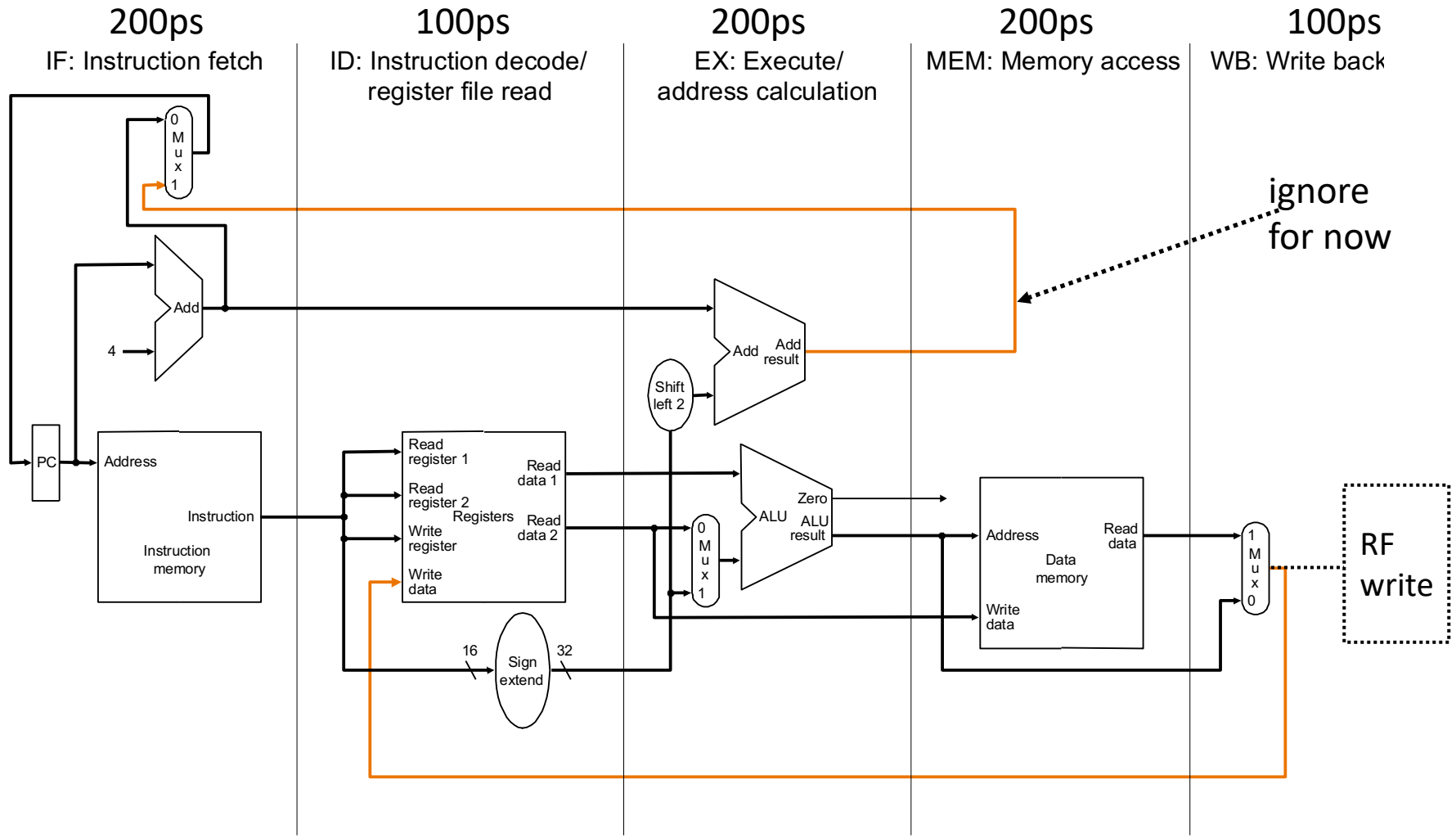


1. Instruction fetch (IF)
2. Instruction decode and
   register operand fetch (ID/RF)
3. Execute/Evaluate memory address (EX/AG)
4. Memory operand fetch (MEM)
5. Store/writeback result (WB)

Remember the Single-Cycle Microarchitecture

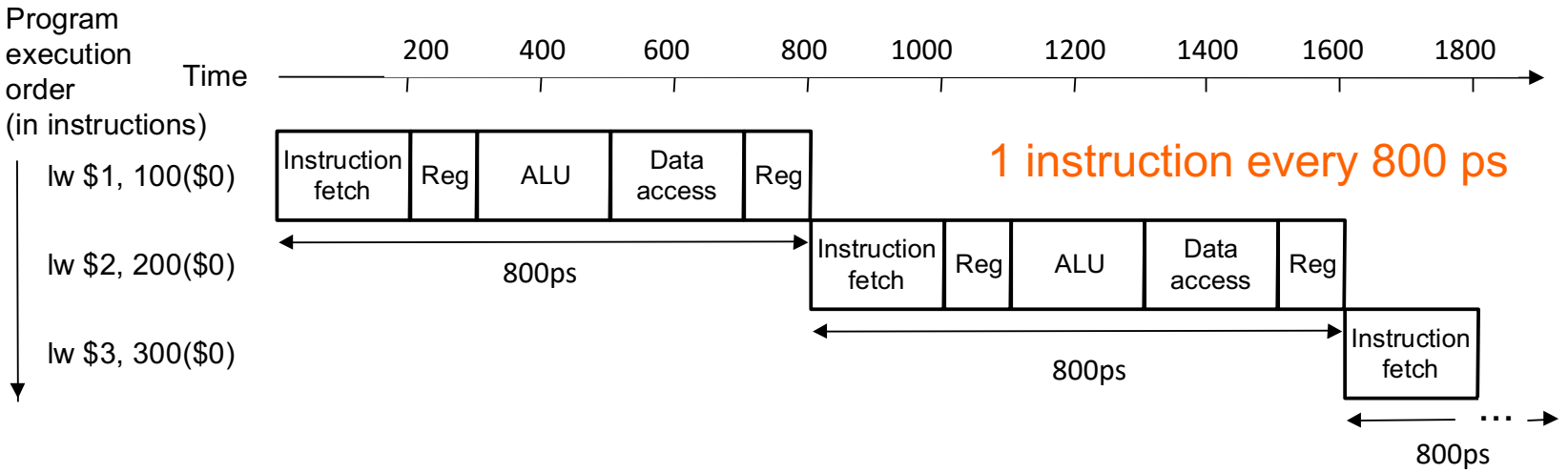


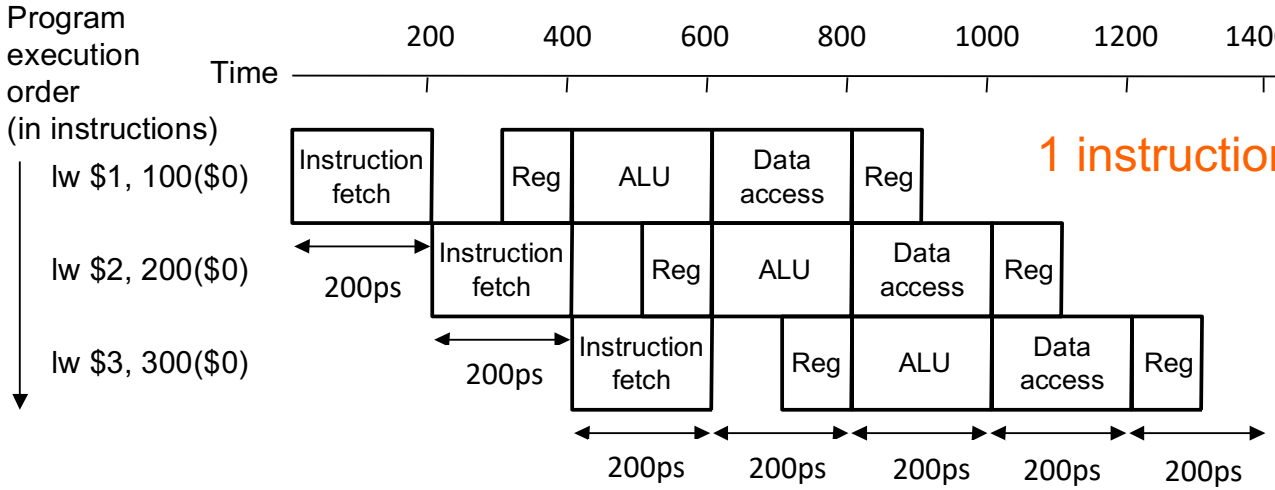
Dividing the Single-Cycle Uarch Into Stages



Is this the correct partitioning? Why not 4 or 6 stages? Why not different boundaries?

Instruction Pipeline Throughput



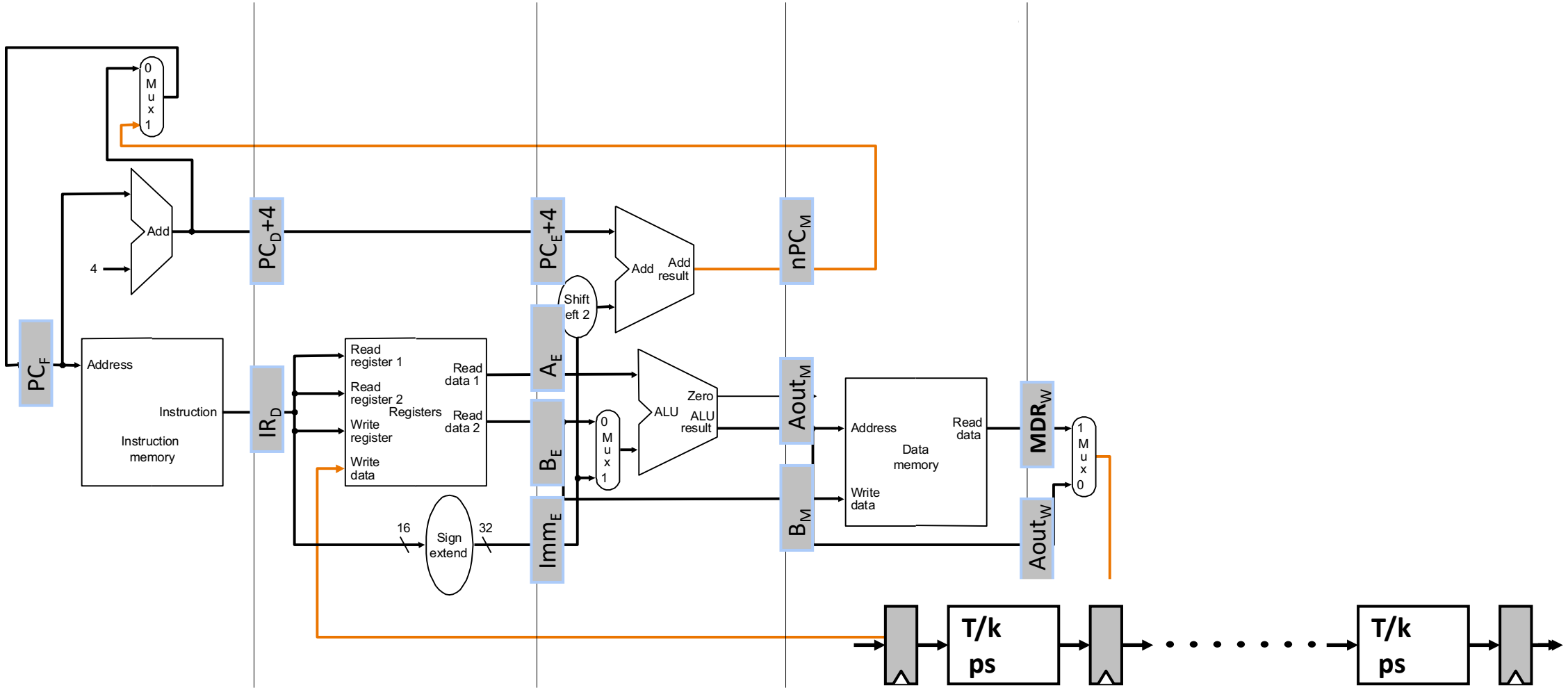


5-stage speedup is 4, not 5 as predicted by the ideal model. Why?

Half of the clock cycle is wasted in two stages (ID/RF and WB)

Enabling Pipelined Processing: Pipeline Registers

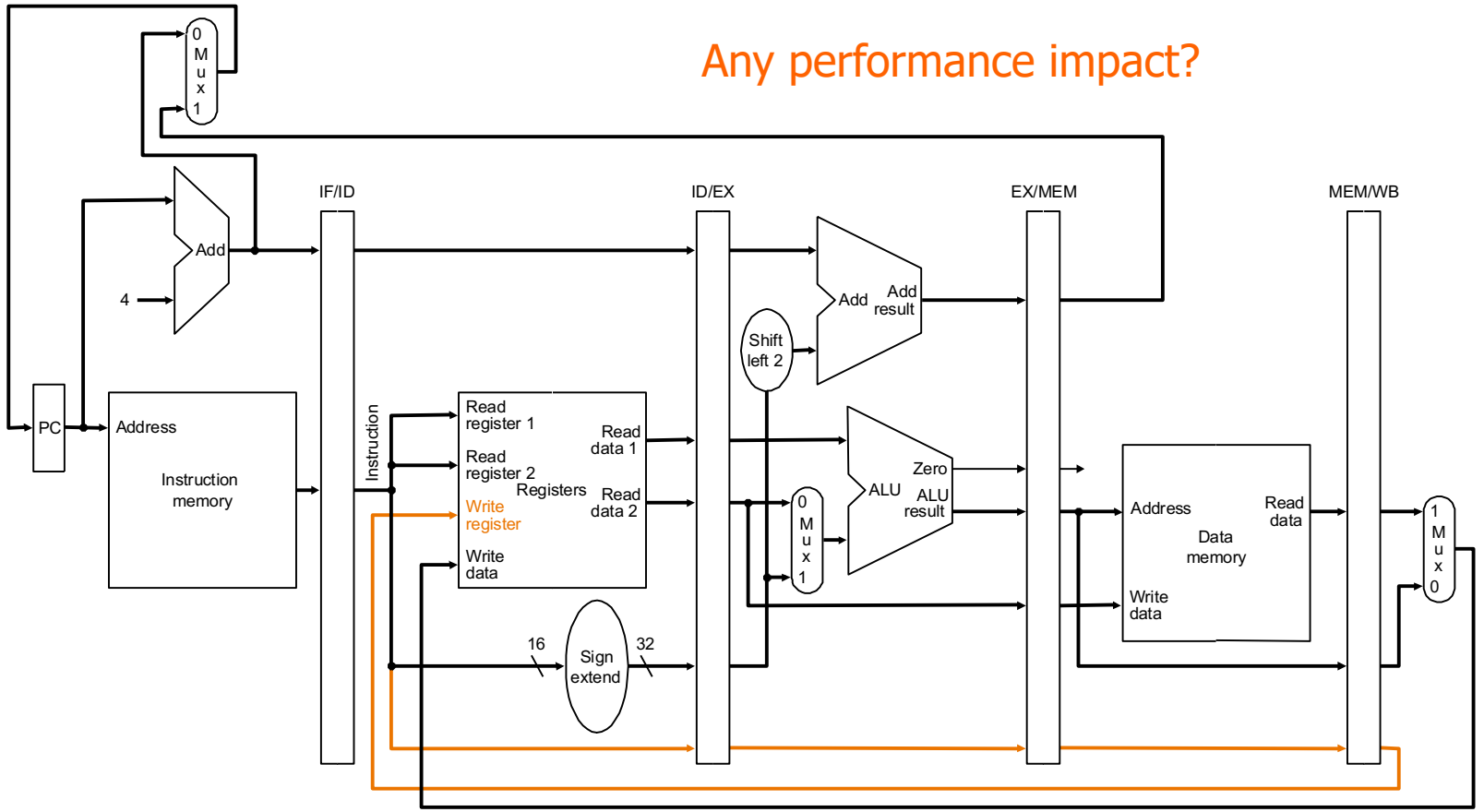
No resource is used by more than one stage



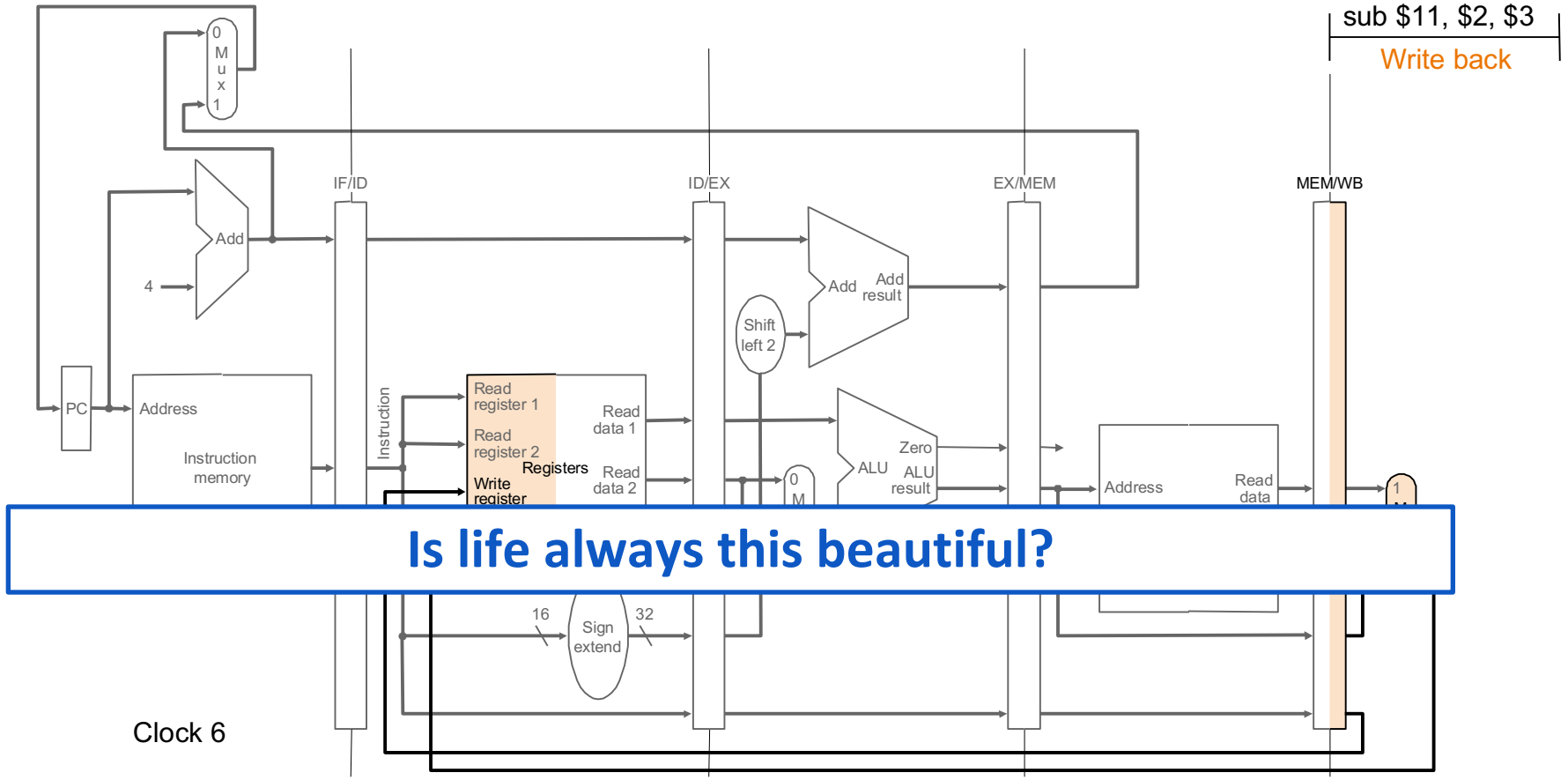
Pipelined Operation Example

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

Any performance impact?

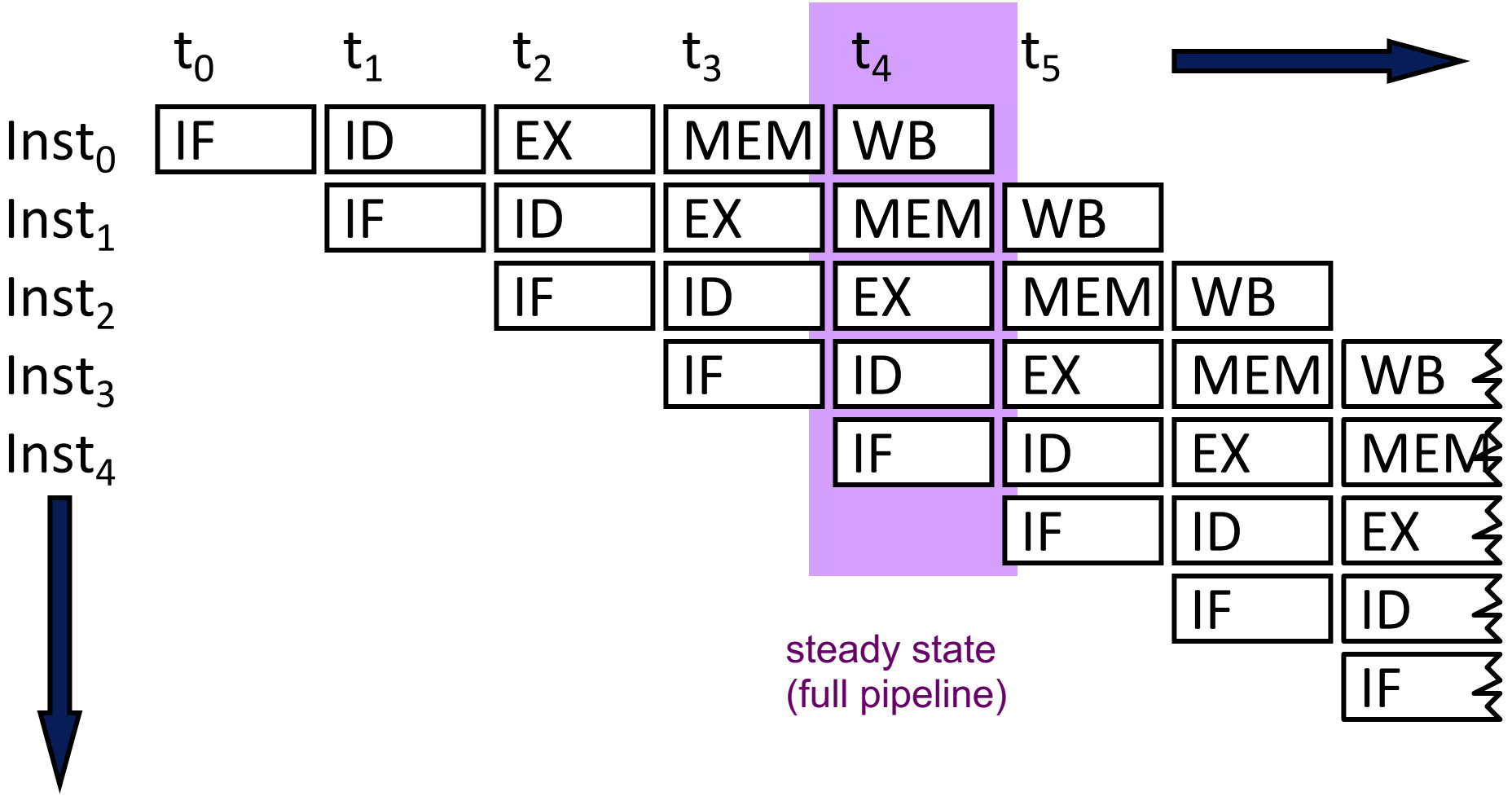


Pipelined Operation Example



Is life always this beautiful?

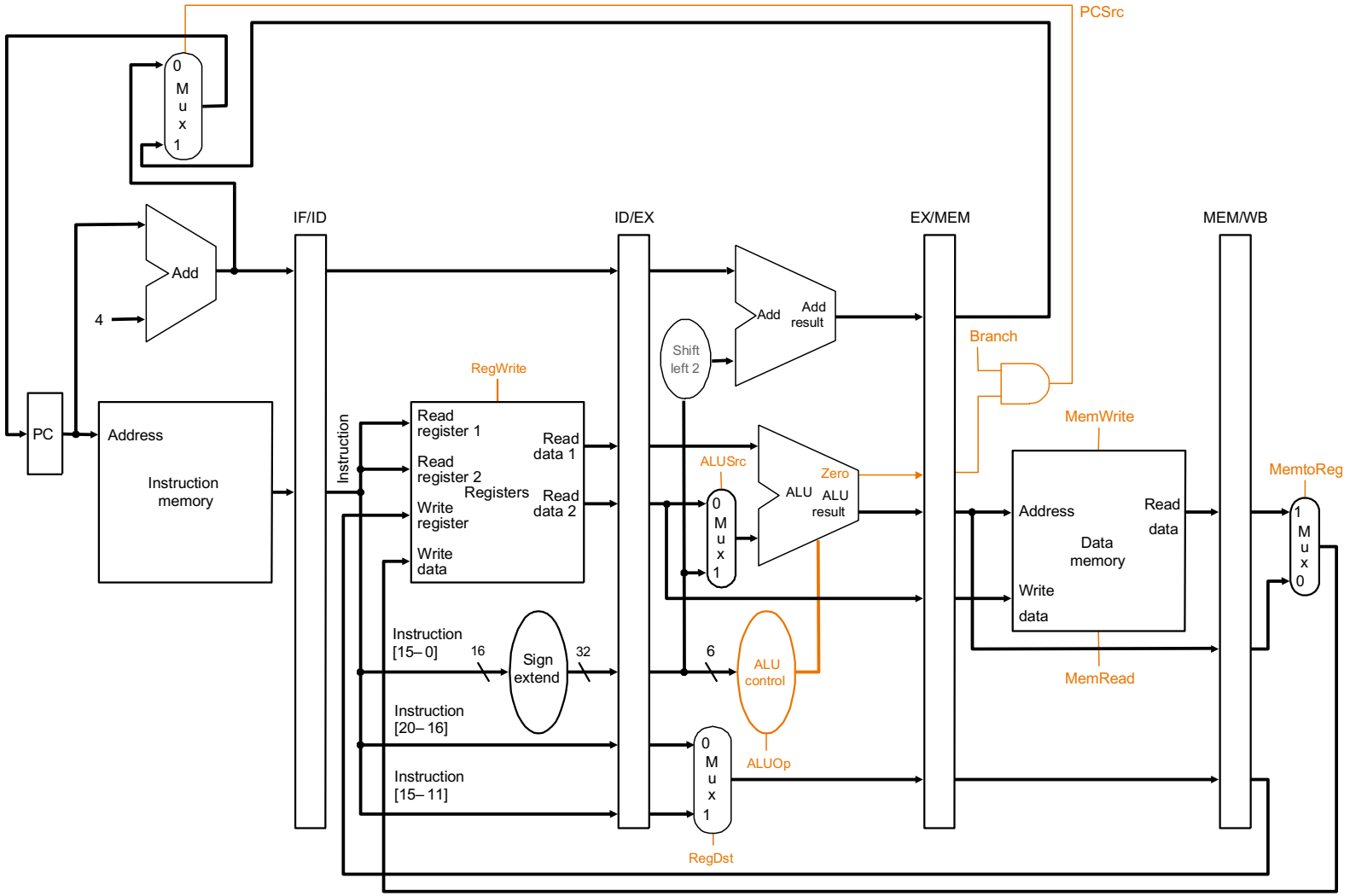
Illustrating Pipeline Operation: Operation View



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

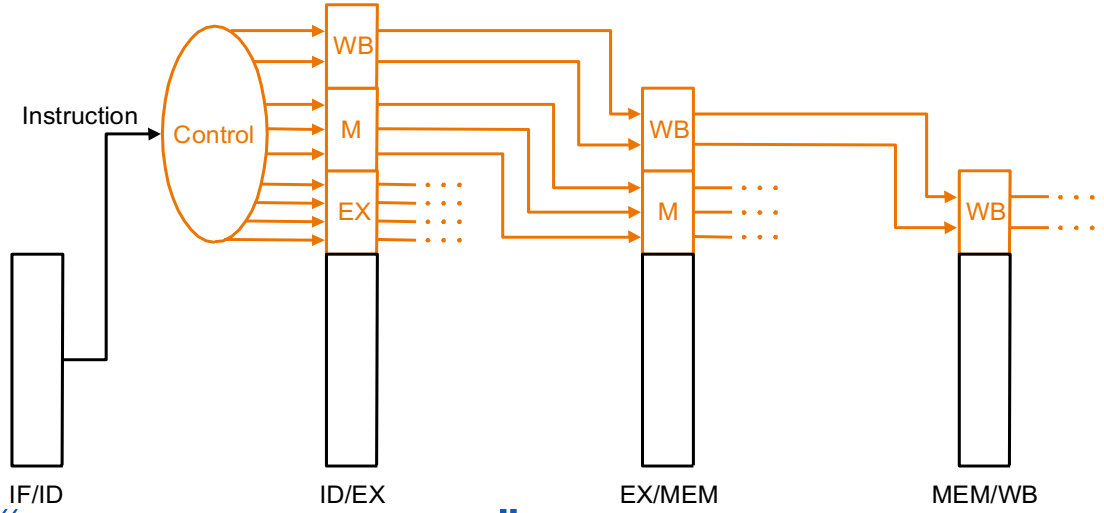


An identical set of control points as the single-cycle datapath

Control Signals in a Pipeline

- For a given instruction
  - same control signals as single-cycle, but
  - control signals required at different cycles, depending on the stage

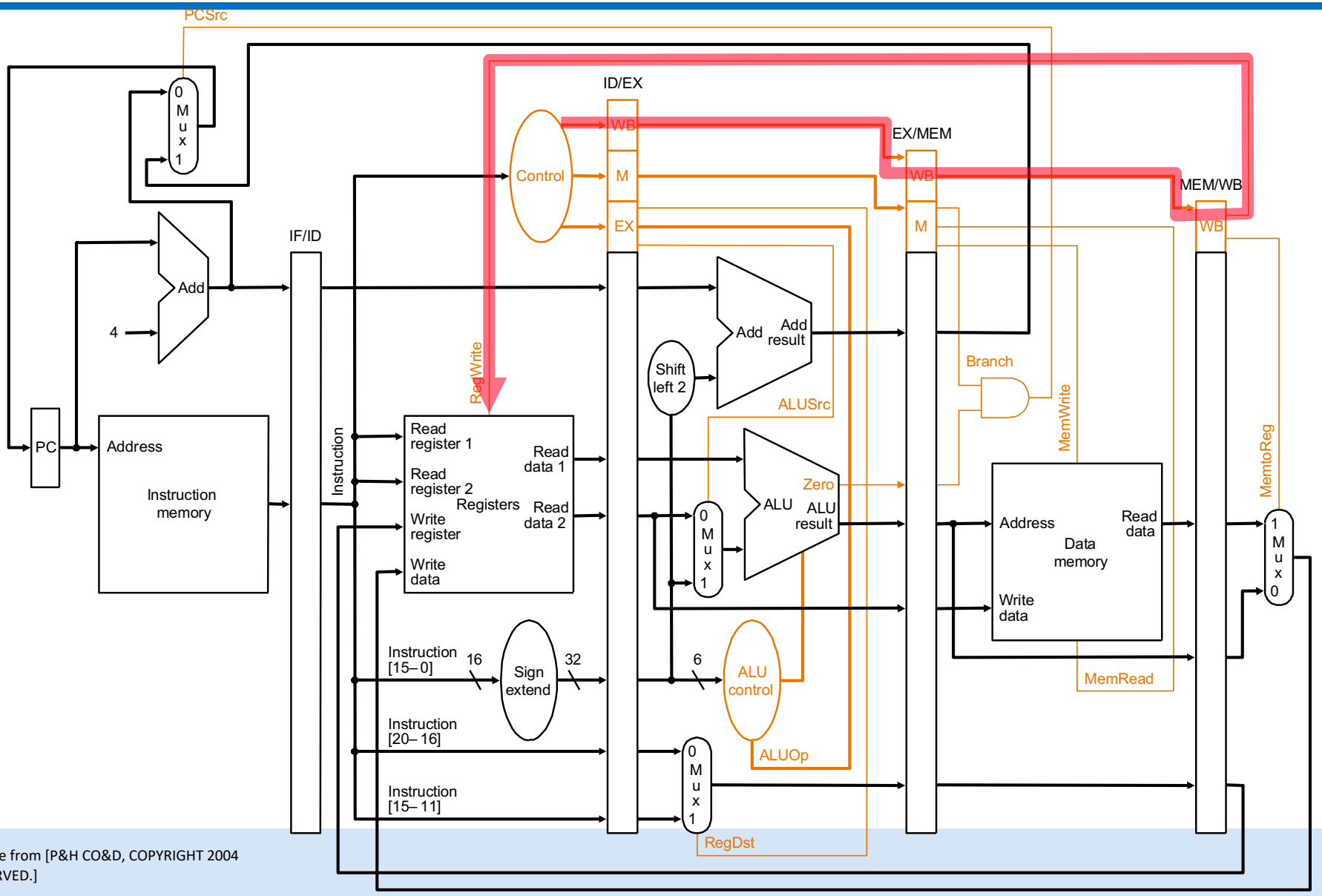
⇒ Option 1: decode once using the same logic as single-cycle and buffer signals until consumed



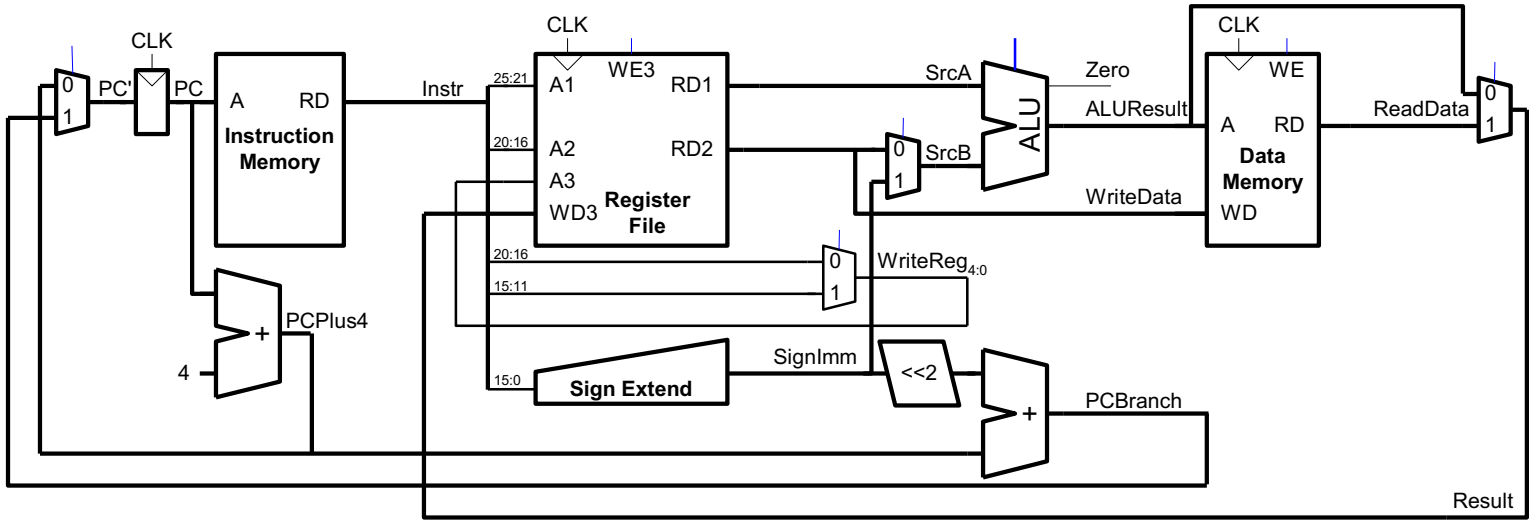
⇒ Option 2: carry relevant "instruction word/field" down the pipeline and decode locally within each or in a previous stage

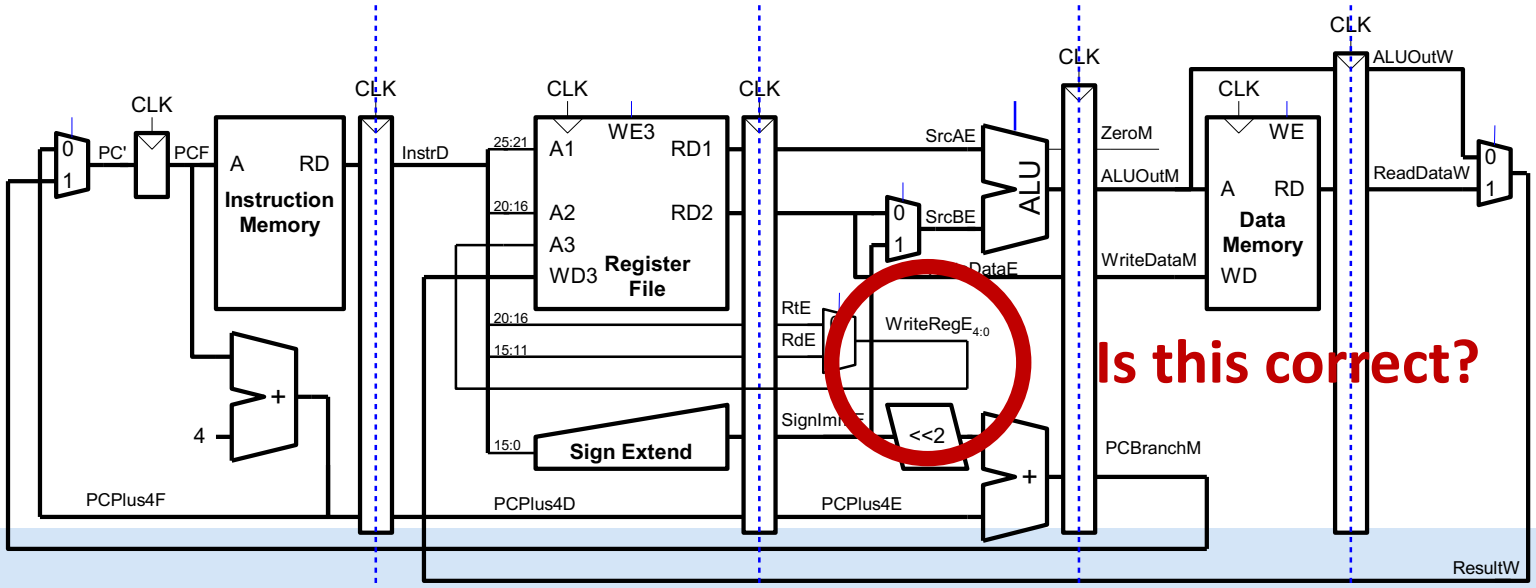
Which one is better?

Pipelined Control Signals



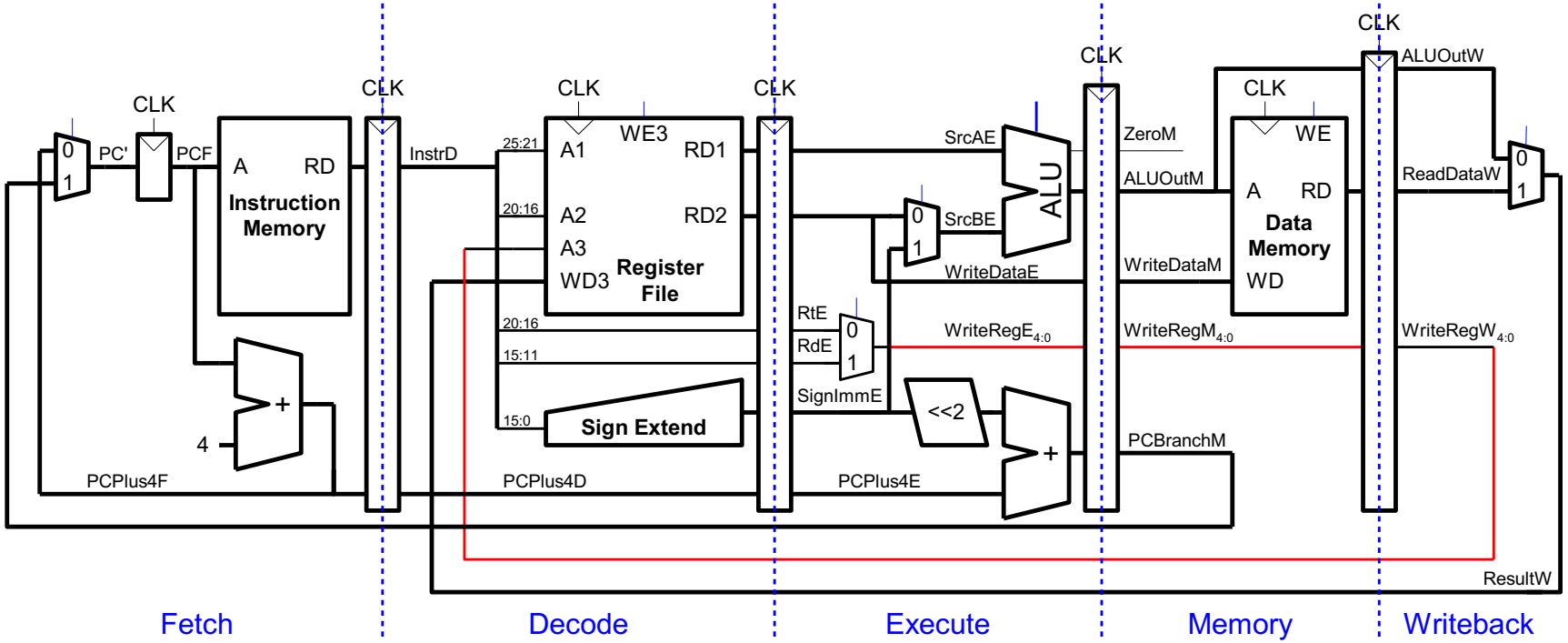
Another Example: Single-Cycle and Pipelined





Fetch | Decode | Execute | Memory | Writeback

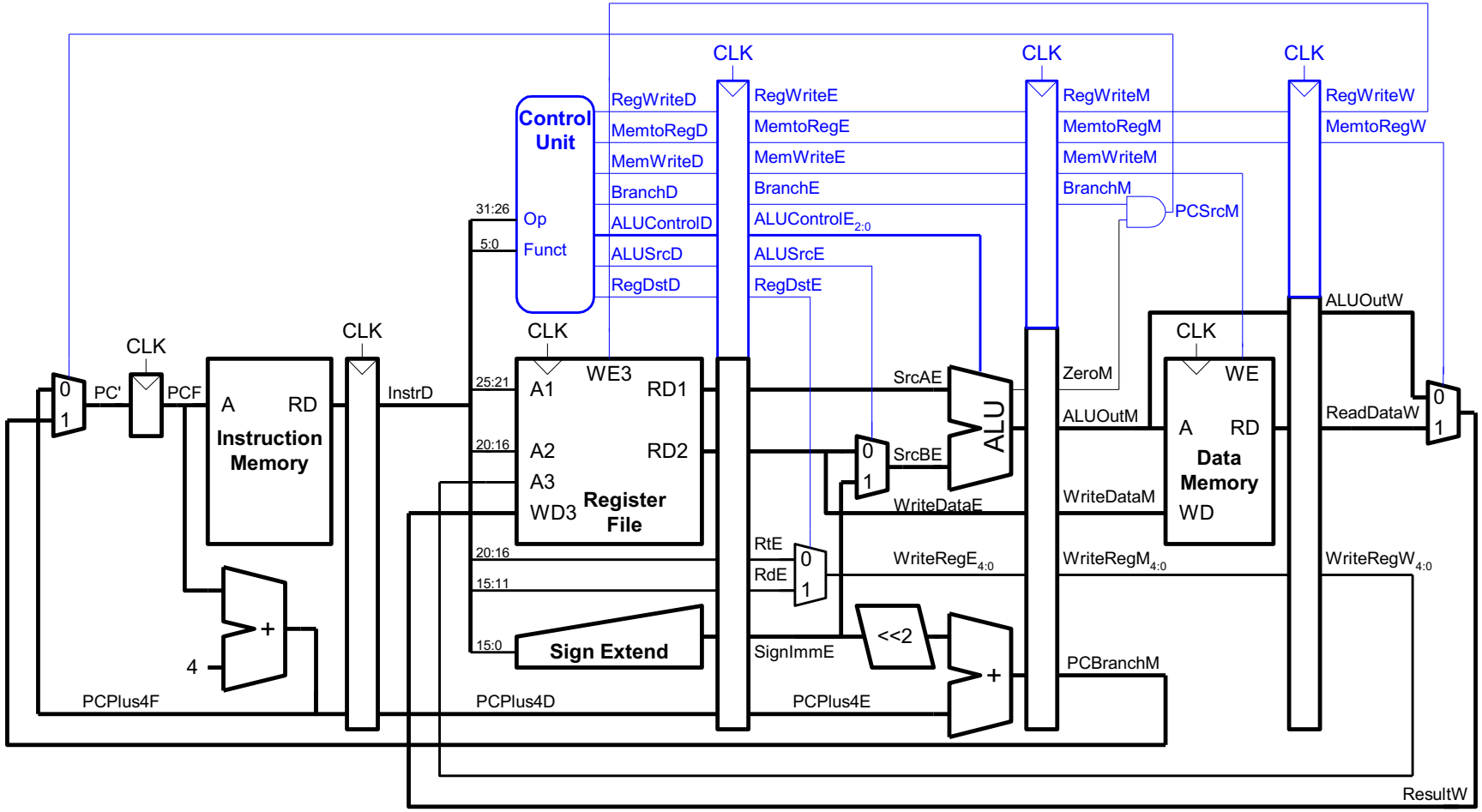
Another Example: Correct Pipelined Datapath



- WriteReg control signal must arrive at the same time as Result

Pipelined processor. Harris and Harris, Chapter 7.5

Another Example: Pipelined Control



Same control unit as a single-cycle processor. Control is delayed to the proper pipeline stage

Mainstream modern big cores: Pipeline Examples

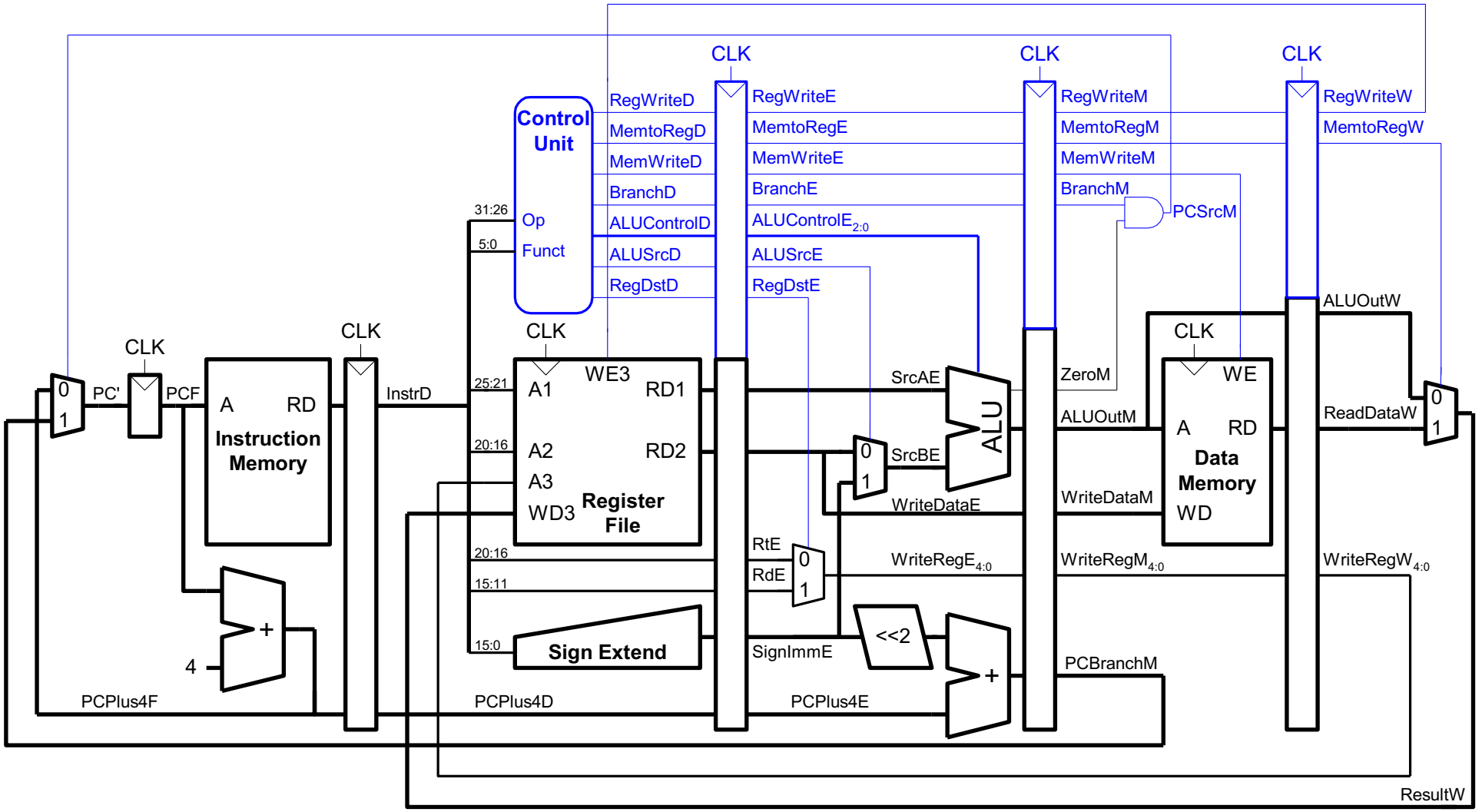
CPU family	Example core (representative)	Pipeline depth
Apple M-series	M1 “Firestorm” (performance core)	~13 cycles
AMD Zen	Zen 4 (Ryzen 7000 / EPYC Genoa class)	~13 cycles common-case
Intel Core	Recent “Core” P-cores (general)	~12
ARM Cortex-A series	Cortex-A “big” / Cortex-X class	~10-ish cycles (proxy, varies)
RISC-V U74 (SiFive)	SiFive U74	8 stages (explicit)
RISC-V U8x (SiFive)	SiFive U84 (U8-series)	12 stages

Pipelined Datapath and Control Logic

- Pipelined datapath design
  - Start with the single-cycle datapath
  - Decide how to break it down into stages
  - Add pipeline registers to separate stages from each other
  - Propagate data signals to stages where they are needed
  - Make sure the correct action is taken at the correct stage
    - Wires that cross stages need to be handled carefully

- Pipelined control logic design
  - Start with the single-cycle control signals and control logic
  - Propagate each control signal to the stages where it is needed
    - Need to extend pipeline registers to do so

Basic Pipelined Processor



How do we handle dependent instructions?

How do we handle multi-cycle memory access?

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 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"?

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 fast but still have to take the same clock cycle time)
- Independent operations … NOT!
  - ⇒ Instructions are not independent of each other
    - Need to detect and resolve inter-instruction dependences 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 (Hazard) and Their Types

- Also called "dependency" or "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?

Example Resource Dependence: RegFile

- With careful design, the register file can be read and written in the same cycle:
  - write takes place during the 1st half of the cycle
  - read takes place during the 2nd half of the cycle => no problem!!!
  - read/write from/to the register file has only ***half a clock cycle*** to complete

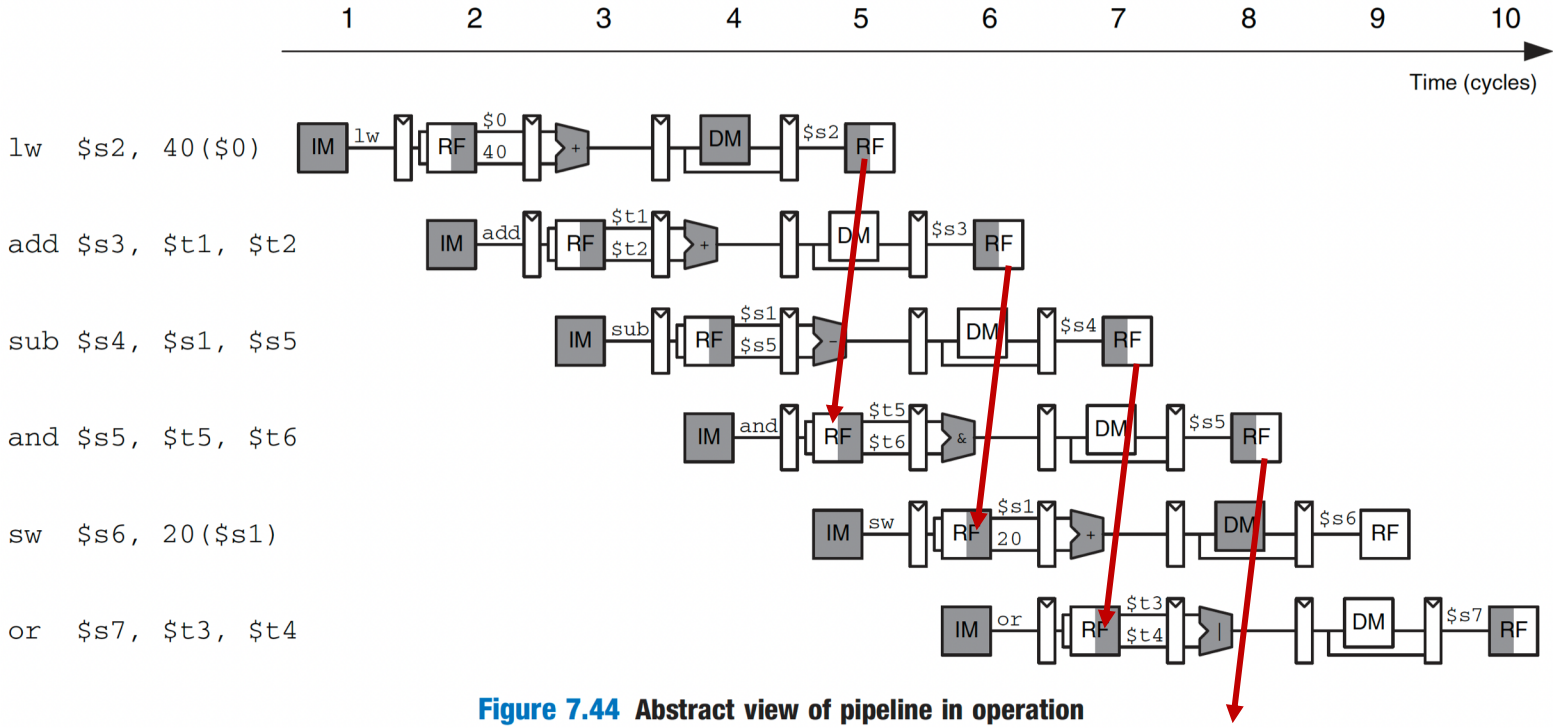


Figure 7.44 Abstract view of pipeline in operation

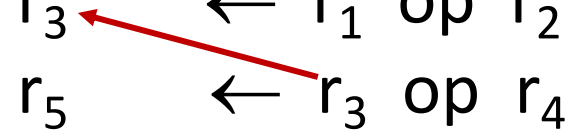
Data Dependences

- Data dependence types
  - Flow dependence (true data dependence – read after write) - Real
  - Anti dependence (write after read) - Fake
  - Output dependence (write after write) - Fake

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

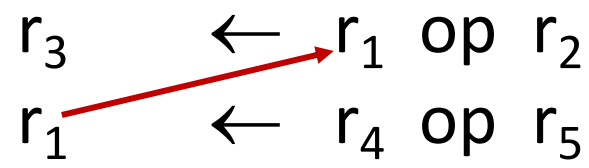
Data Dependence (Hazard) Types

Flow dependence



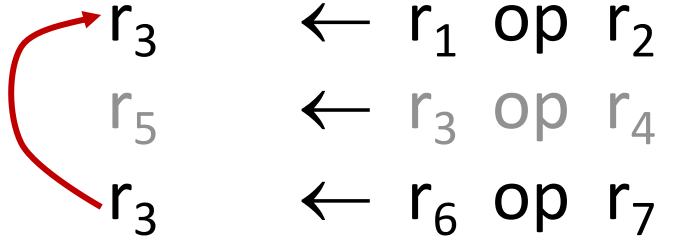
Read-after-Write (RAW)

Anti dependence



Write-after-Read (WAR)

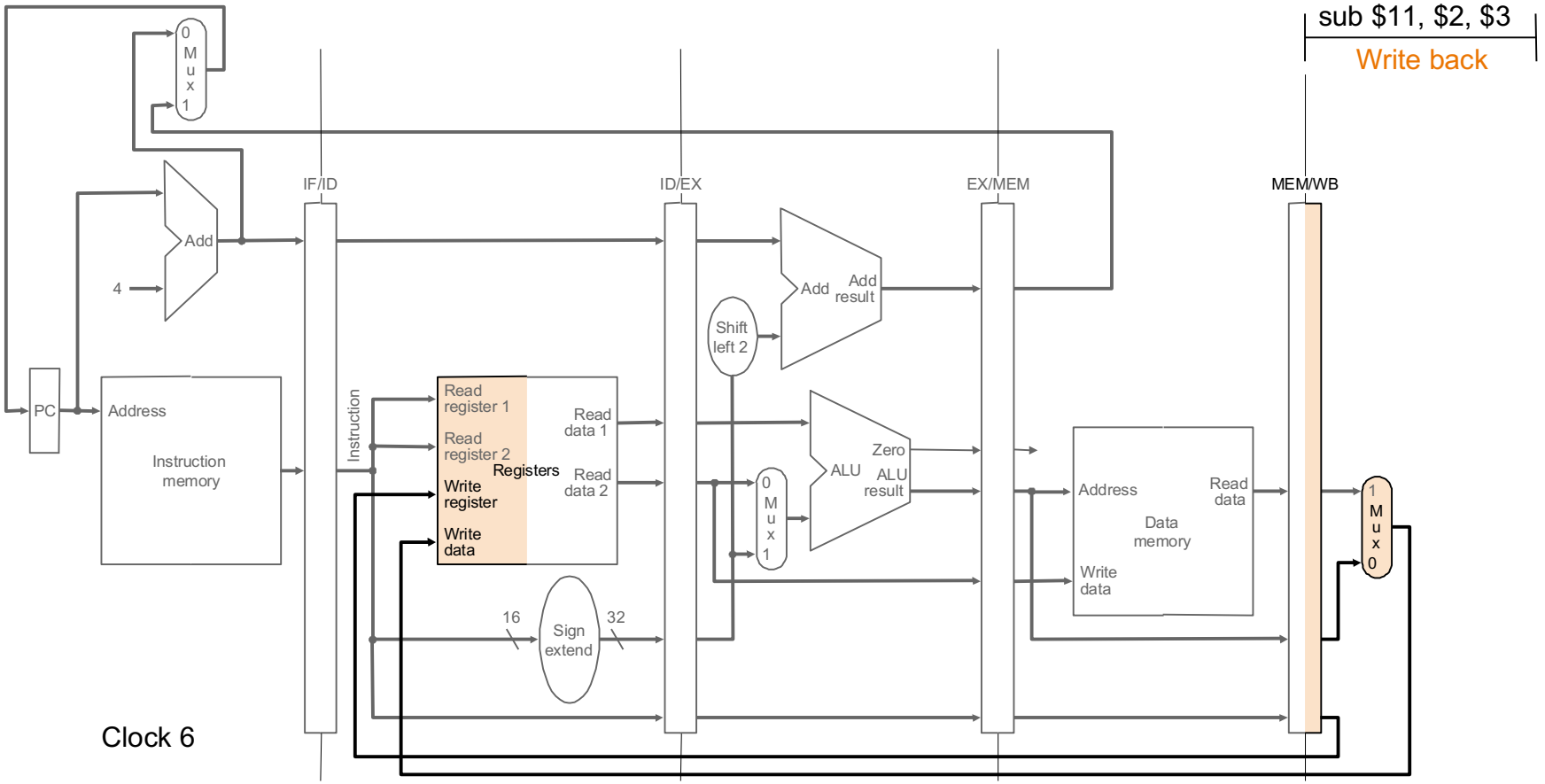
Output dependence



Write-after-Write (WAW)

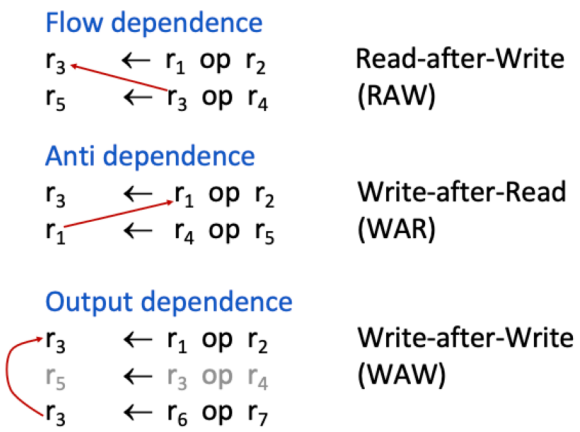
Pipelined Operation Example

What if the SUB were dependent on LW?



Hardware Design

Data Dependence (Hazard) Handling

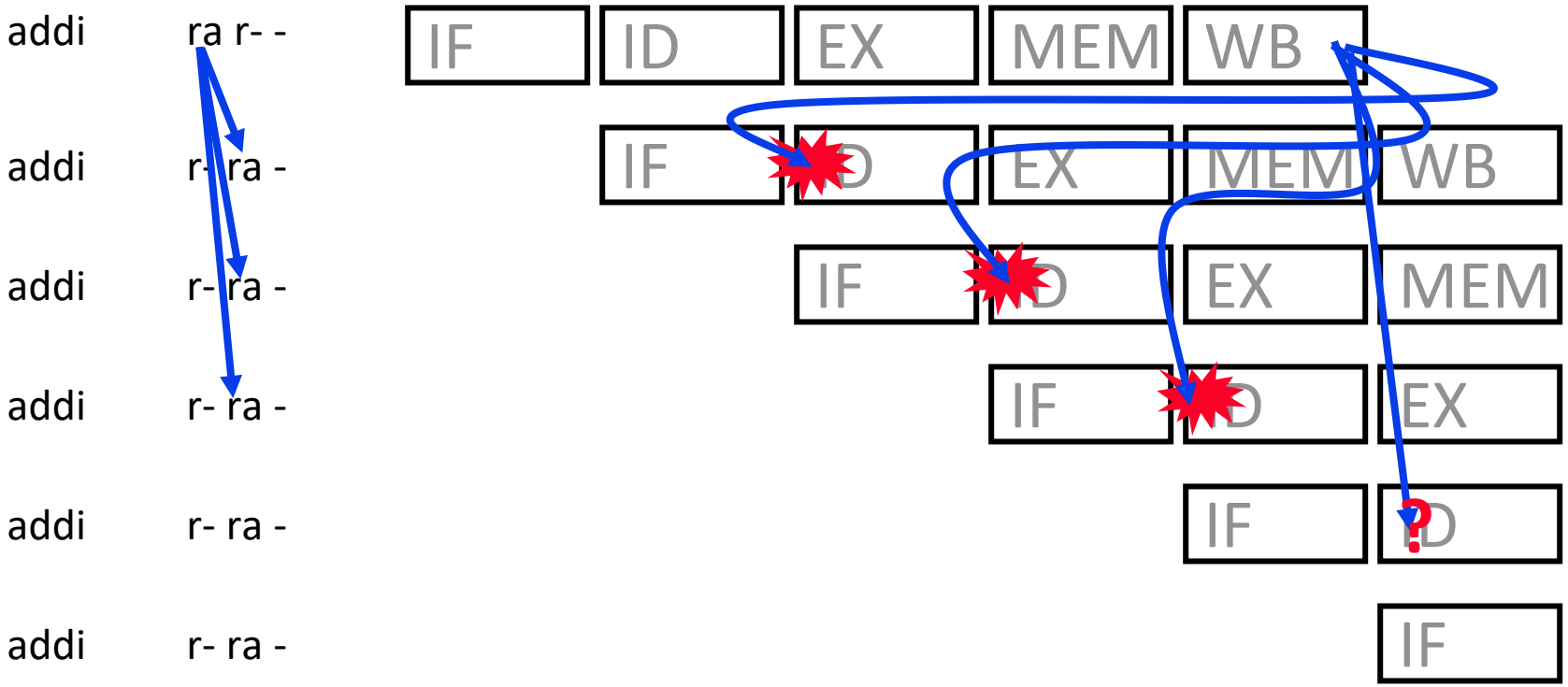


How to Handle Data Dependences

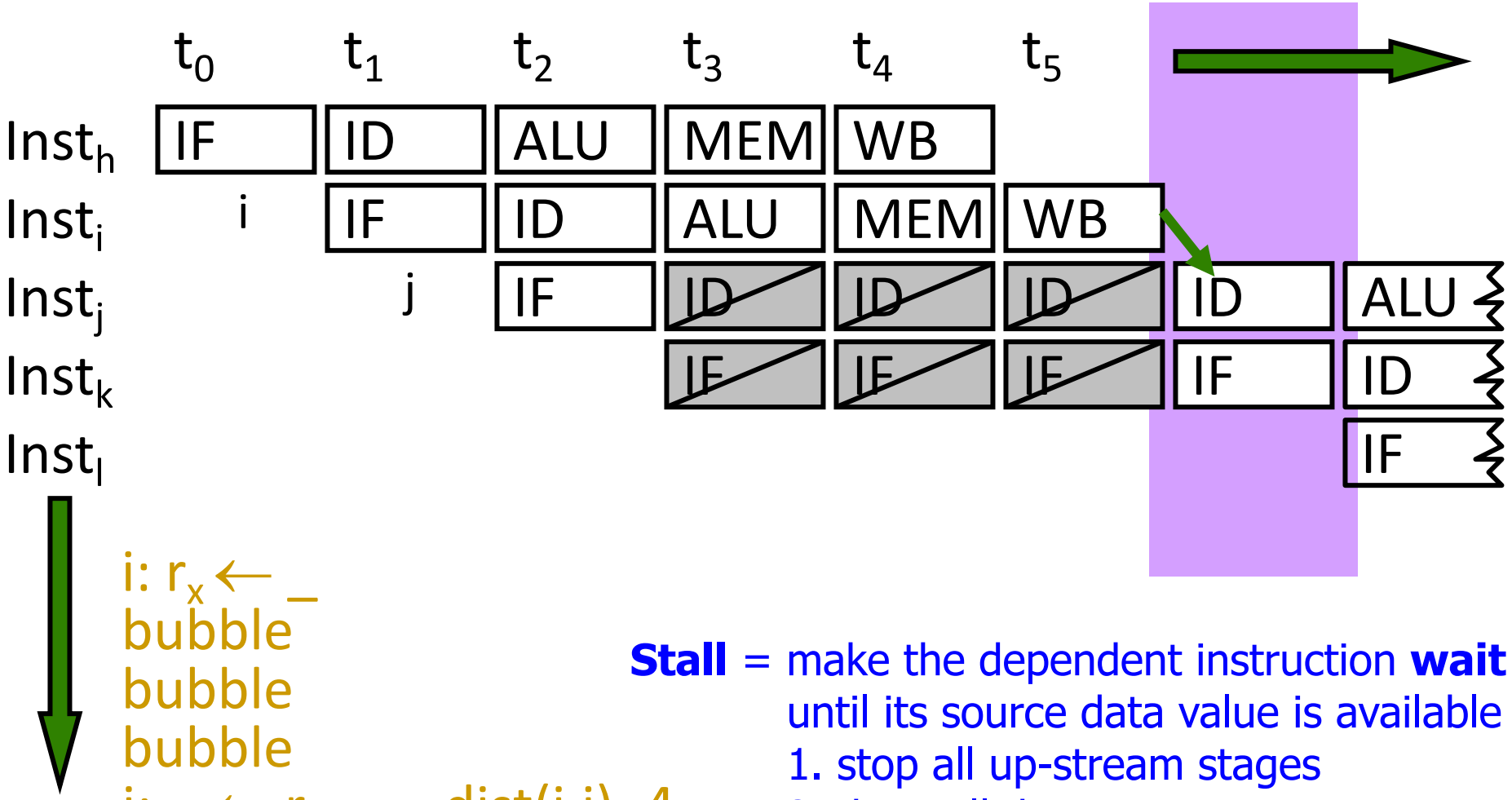
- Anti and output dependences are easier to handle
  - Write to the destination only in the last stage and in program order
- Flow dependences are more interesting & challenging
- Six fundamental ways of handling flow dependences
  - Detect and wait until the value is available in the register file
  - 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
  - Detect and move it out of the way for independent instructions
  - 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 leads to conflicts in the 5-stage pipeline?



Pipeline Stall: Resolving Data Dependence



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

Interlocking

- Interlocking: Detection of dependence between instructions in a pipelined processor to guarantee correct execution

- Software-based interlocking

  vs.

- Hardware-based interlocking

- "MIPS" acronym stand for in architecture?
  - Microprocessor without Interlocked Pipeline Stages

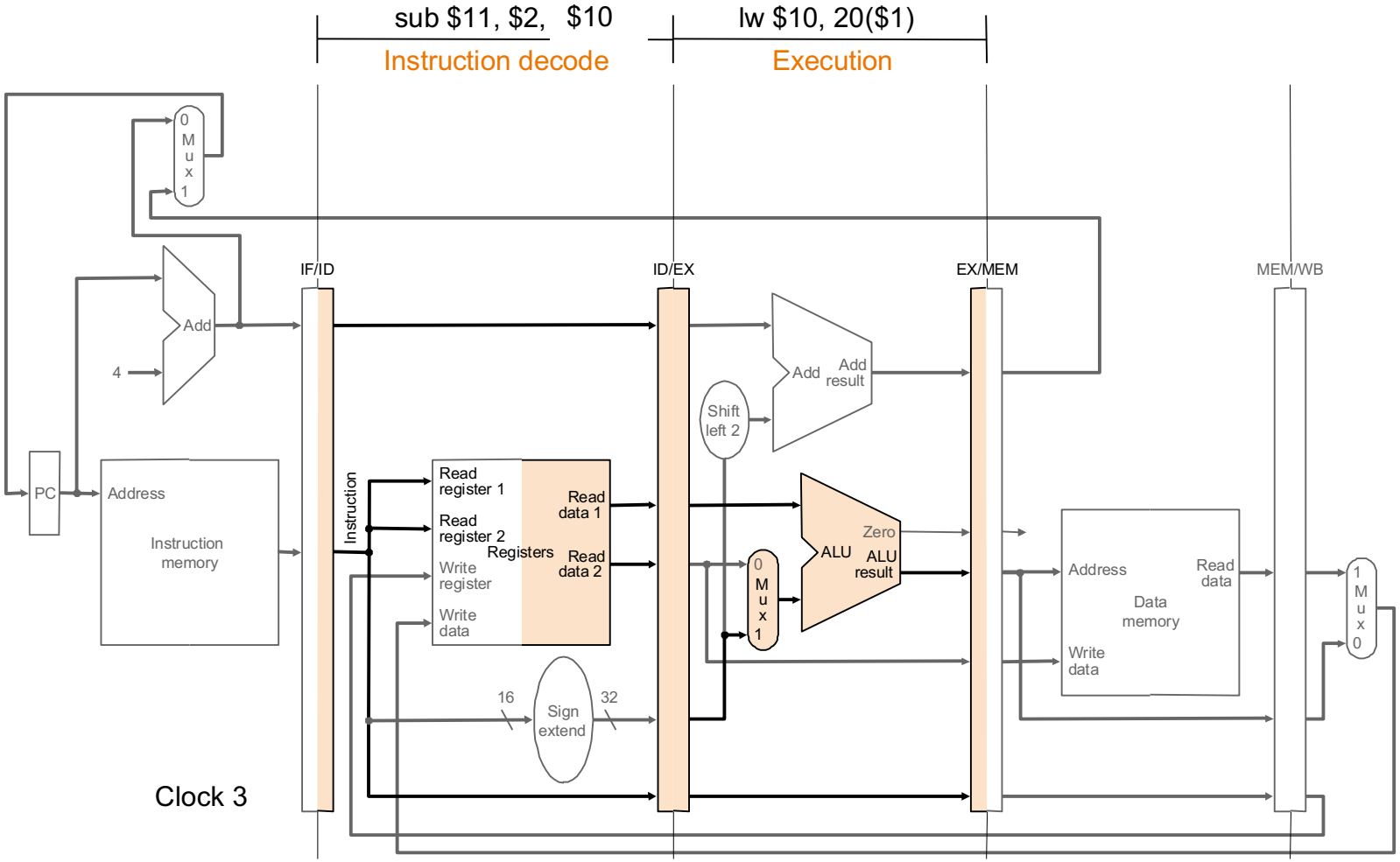
Approaches to Dependence Detection (I)

- Scoreboarding
  - Each register in the register file has a Valid bit associated with it
  - An instruction that writes to the register resets the Valid bit
  - An instruction in the 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 dependencies, not only flow dep.

Pipelined Operation Example



Now, assume SUB is dependent on LW

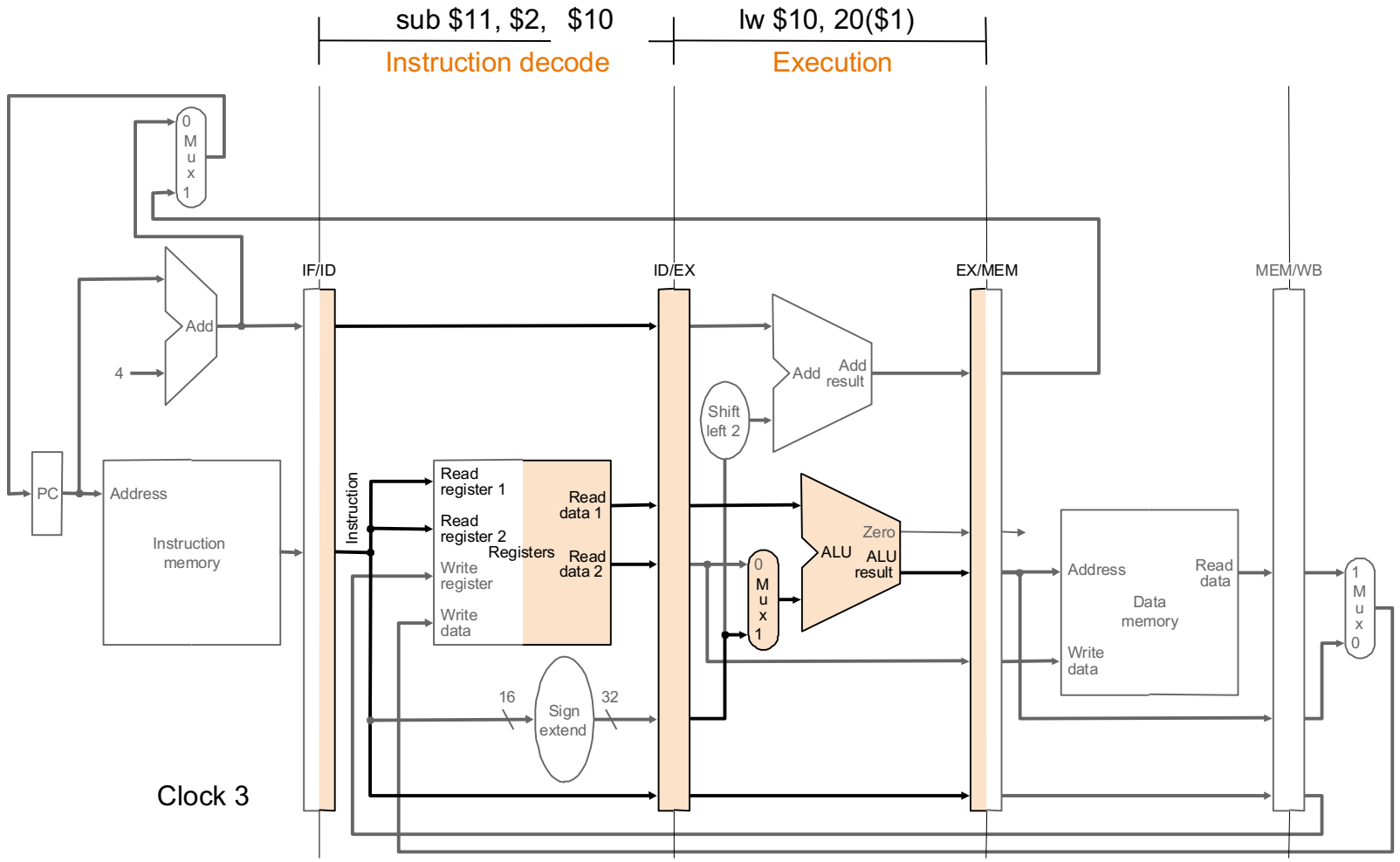
Approaches to Dependence Detection (II)

- Combinational dependence check logic
  - Special logic 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

Pipelined Operation Example



Now, assume SUB is dependent on LW

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

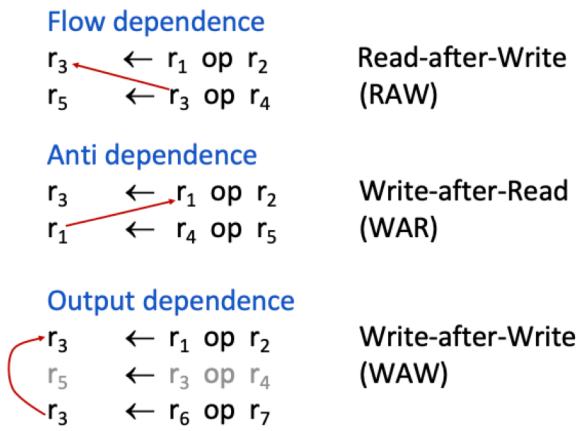
- Option 1: Stall the dependent instruction right away
- Option 2: Stall the dependent instruction only when necessary → data forwarding/bypassing
- Option 3: …

Data Forwarding/Bypassing

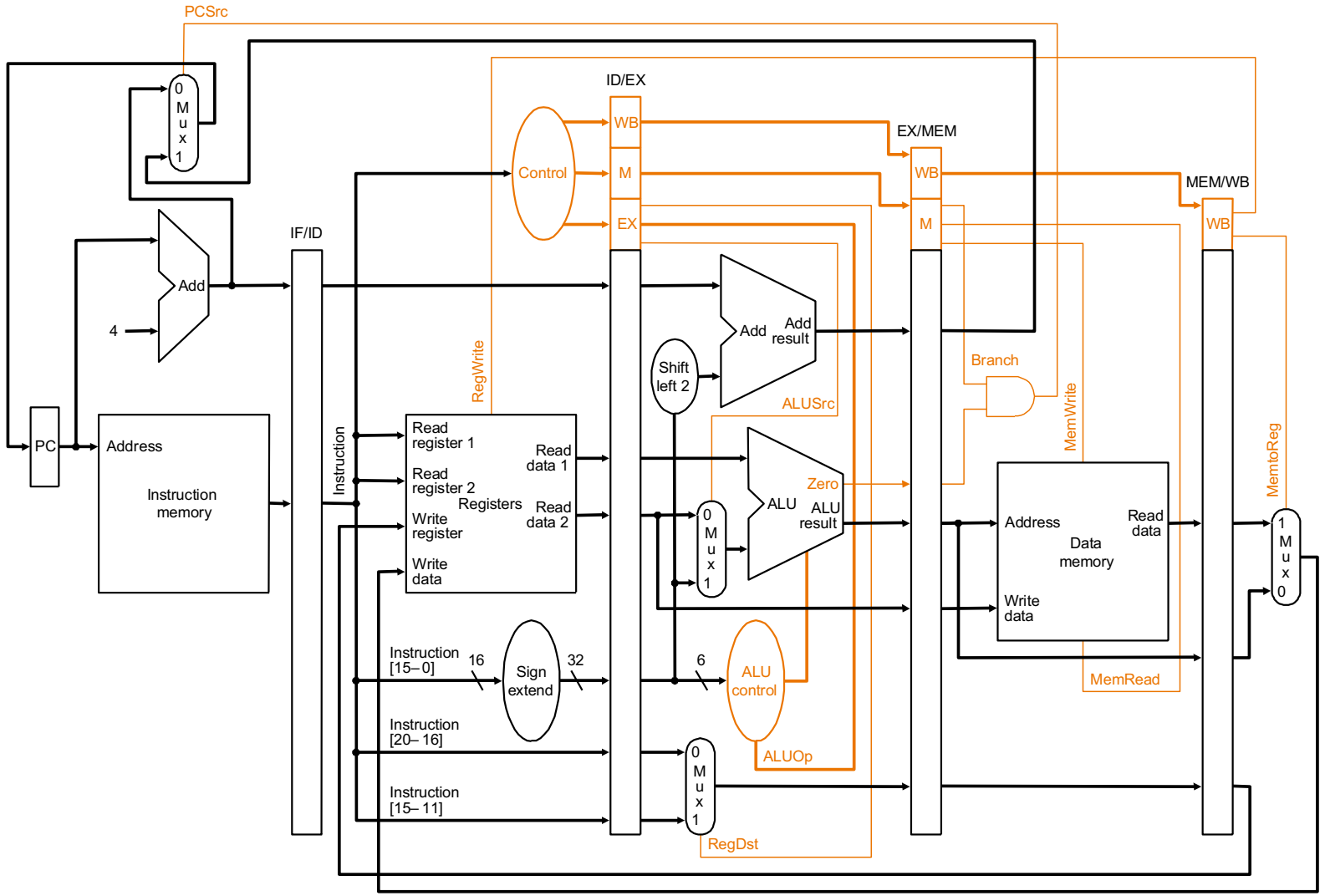
- Problem: A consumer (dependent) instruction has to wait in the decode stage until the producer instruction writes its value in the register file
- Goal: We do not want to stall the pipeline unnecessarily
- Observation: The data value needed by the consumer instruction can be supplied directly from 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 where the value can be supplied → **less stalling**

Hardware Design

Data Dependence Handling: Concepts and Implementation



How to Implement Stalling



Stall

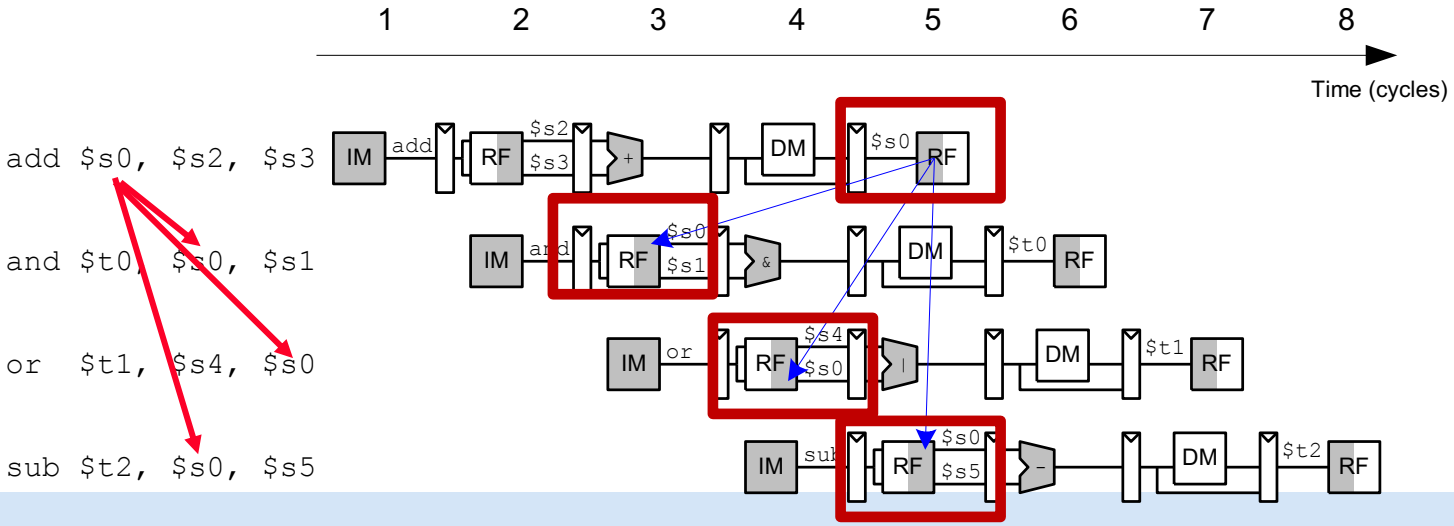
- Disable **PC** and **IF/ID** latching; ensure stalled instruction stays in its stage
- Insert "**invalid**" instructions/nops into the stage following the stalled one (called "**bubbles**")

RAW Data Dependence Example

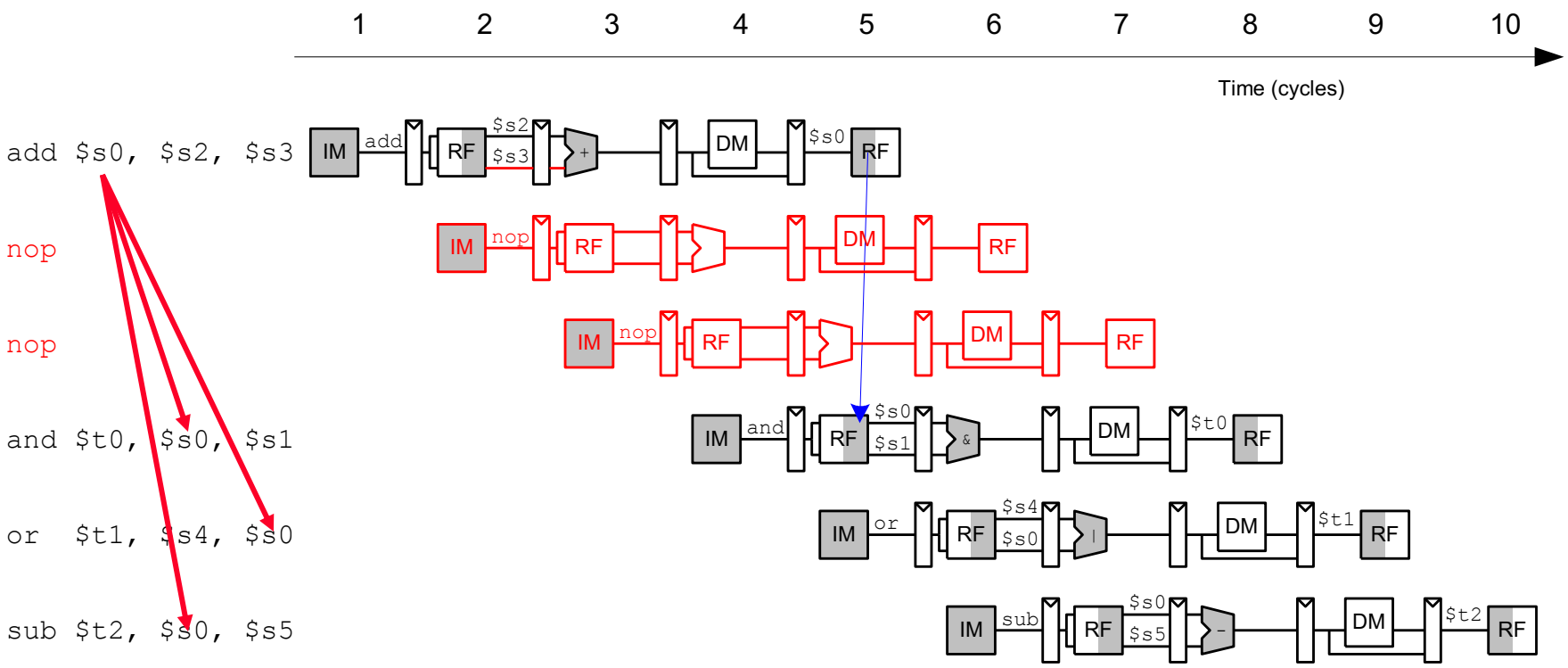
- One instruction writes a register ($so) and the next instructions read this register => read after write (RAW) dependence.

Wrong results happen only if the pipeline handles data dependences incorrectly!

  - Subsequent instructions read the correct value of $so



Compile-Time Detection and Elimination



- Insert enough independent instructions for the required result to be ready by the time it is needed by a dependent one
  - Reorder/reschedule/insert instructions at the compiler level

Data Forwarding

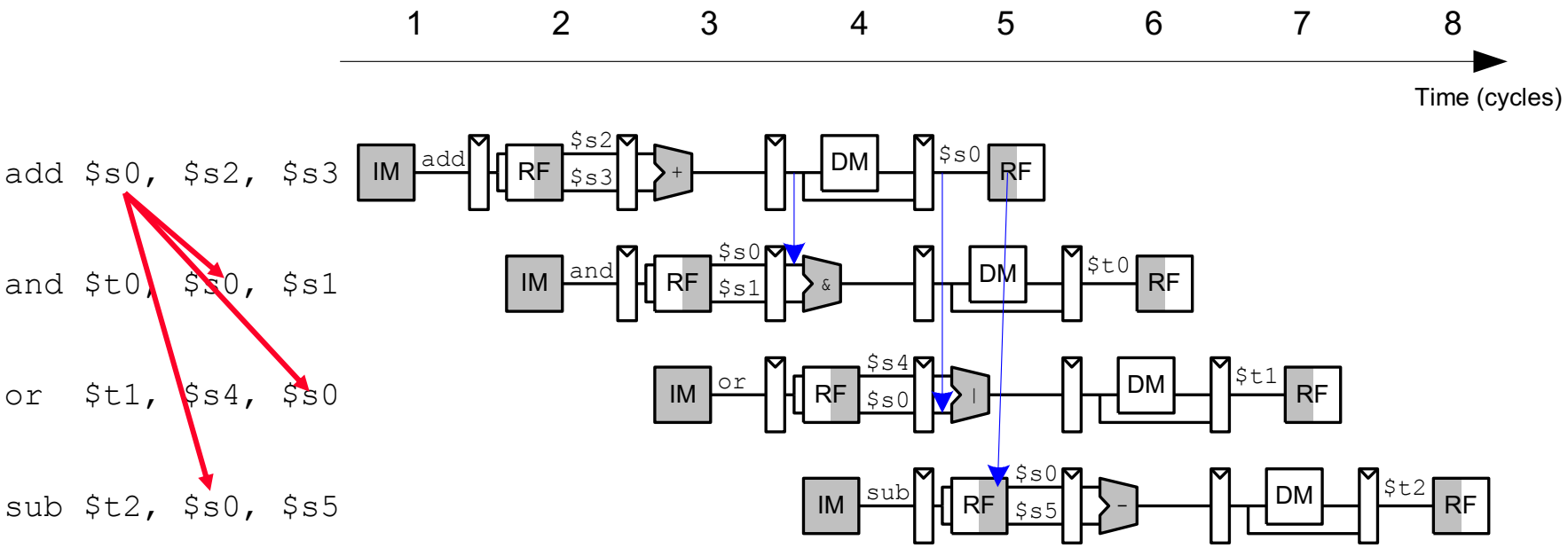
- Also called Data Bypassing

- Forward the result value to the dependent instruction as soon as the value is available

- We have already seen the basic idea before
- Remember dataflow?
  - Data value is supplied to the dependent instruction as soon as it is available
  - Instruction executes when all its operands are available

- Data forwarding brings a pipeline closer to dataflow execution principles

Data Forwarding: Locations in Datapath



From latched output of ALU to input of ALU

From WB to input of ALU

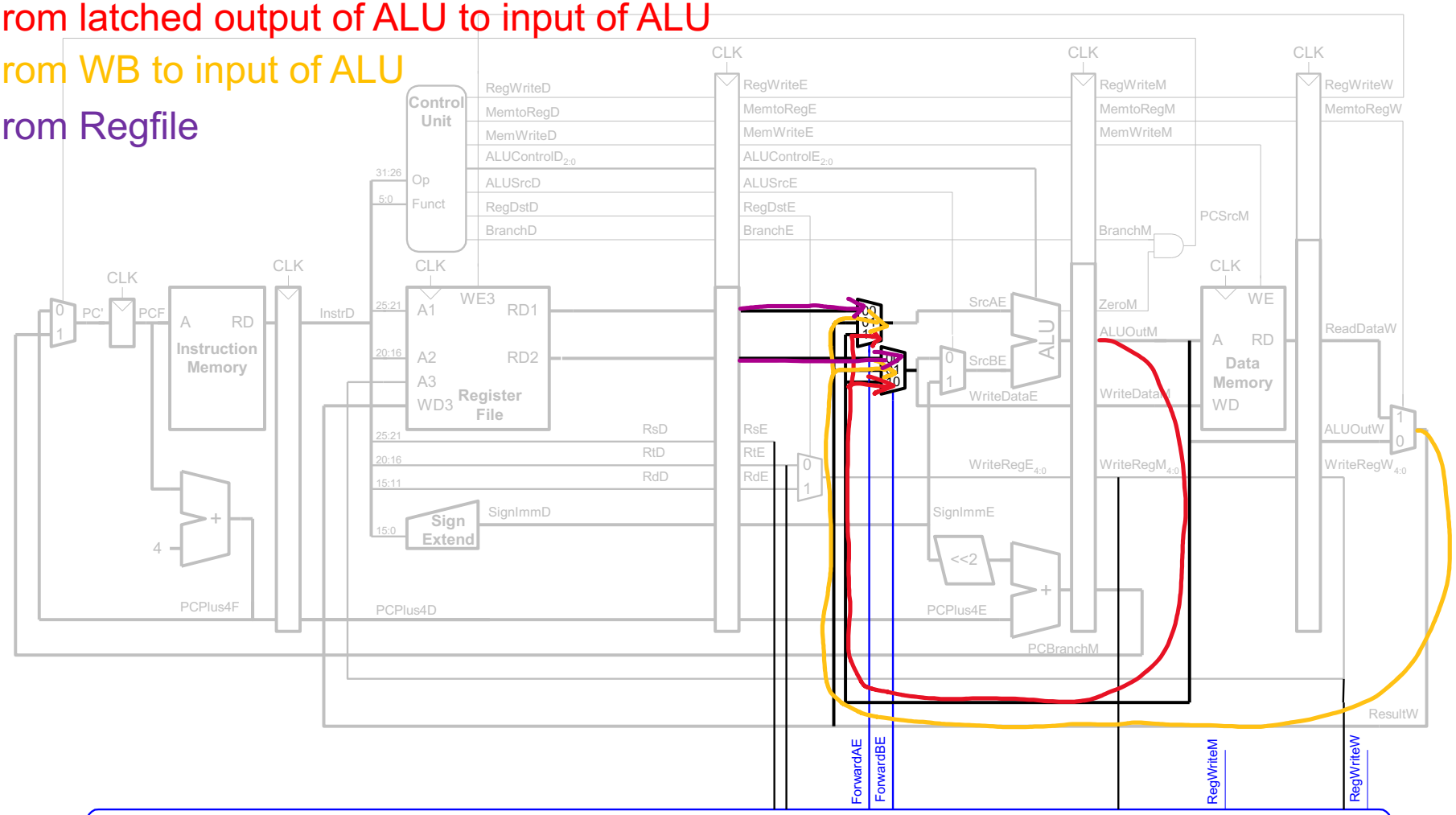
From WB to RF (internal in Register File)

Data Forwarding: Datapath & Control

From latched output of ALU to input of ALU

From WB to input of ALU

From Regfile



Data Forwarding: Implementation

- Forward to the Execute stage from either:
  - Memory stage or
  - Writeback stage

- When should we forward from either the Memory or Writeback stage?
  - If that stage writes to a destination register, and the destination register matches the source register
  - If both the Memory & Writeback stages contain matching destination registers, the Memory stage has priority to forward its data, because it contains the *more recently executed* instruction

Data Forwarding (in Pseudocode)

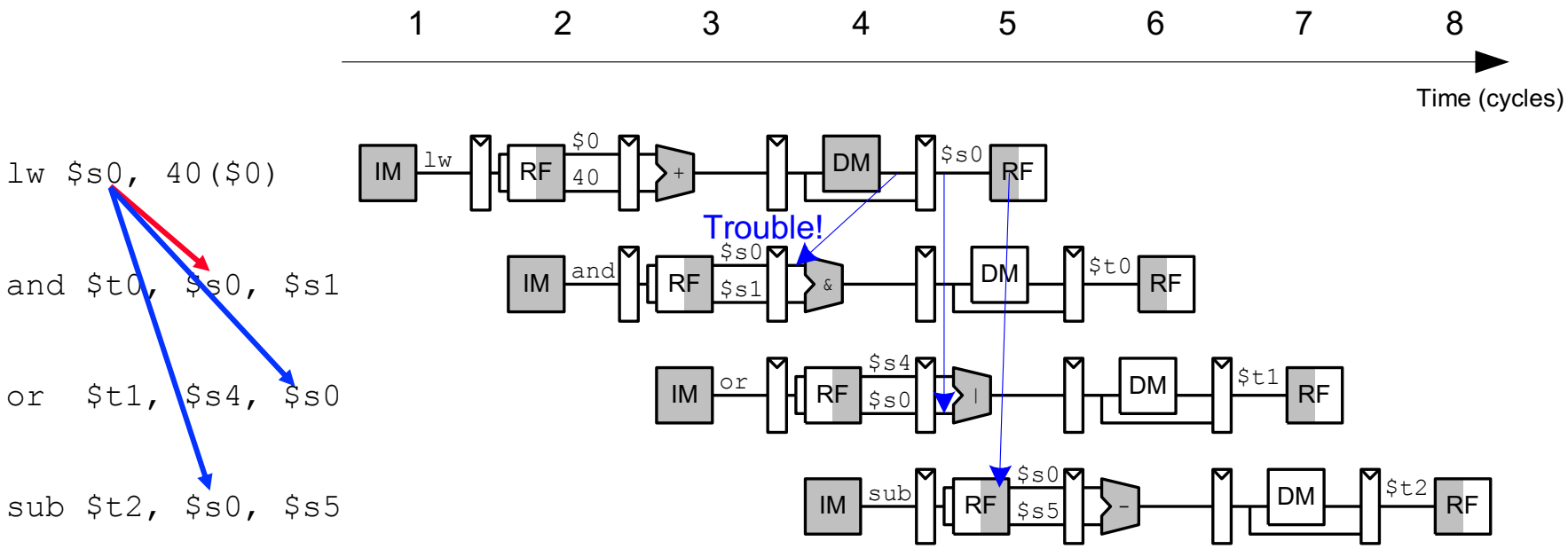
- Forward to the Execute stage from either:
  - Memory stage or
  - Writeback stage

- Forwarding logic for *ForwardAE (pseudo code)*:

```
if  ((rsE != 0) AND (rsE == WriteRegM) AND RegWriteM) then

     ForwardAE = 10  # forward from Memory stage


else if ((rsE != 0) AND (rsE == WriteRegW) AND RegWriteW) then

     ForwardAE = 01  # forward from Writeback stage
else
     ForwardAE = 00  # no forwarding
```

- Forwarding logic for *ForwardBE* same, but replace *rsE* with *rtE*

Data Forwarding Is **Not** Always Possible

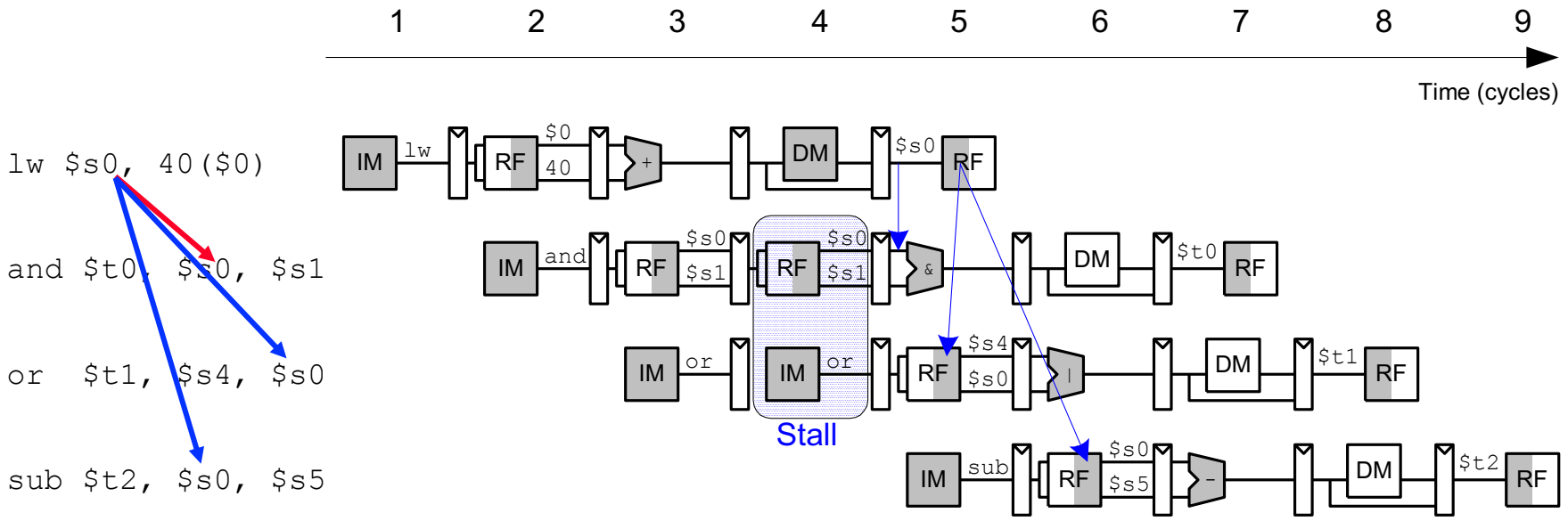


- Forwarding is usually sufficient to resolve RAW data dependences
- Unfortunately, there are cases when forwarding is **not possible**
  - due to pipeline design and instruction latencies
  - The `lw` instruction does not finish reading data until the end of Memory stage

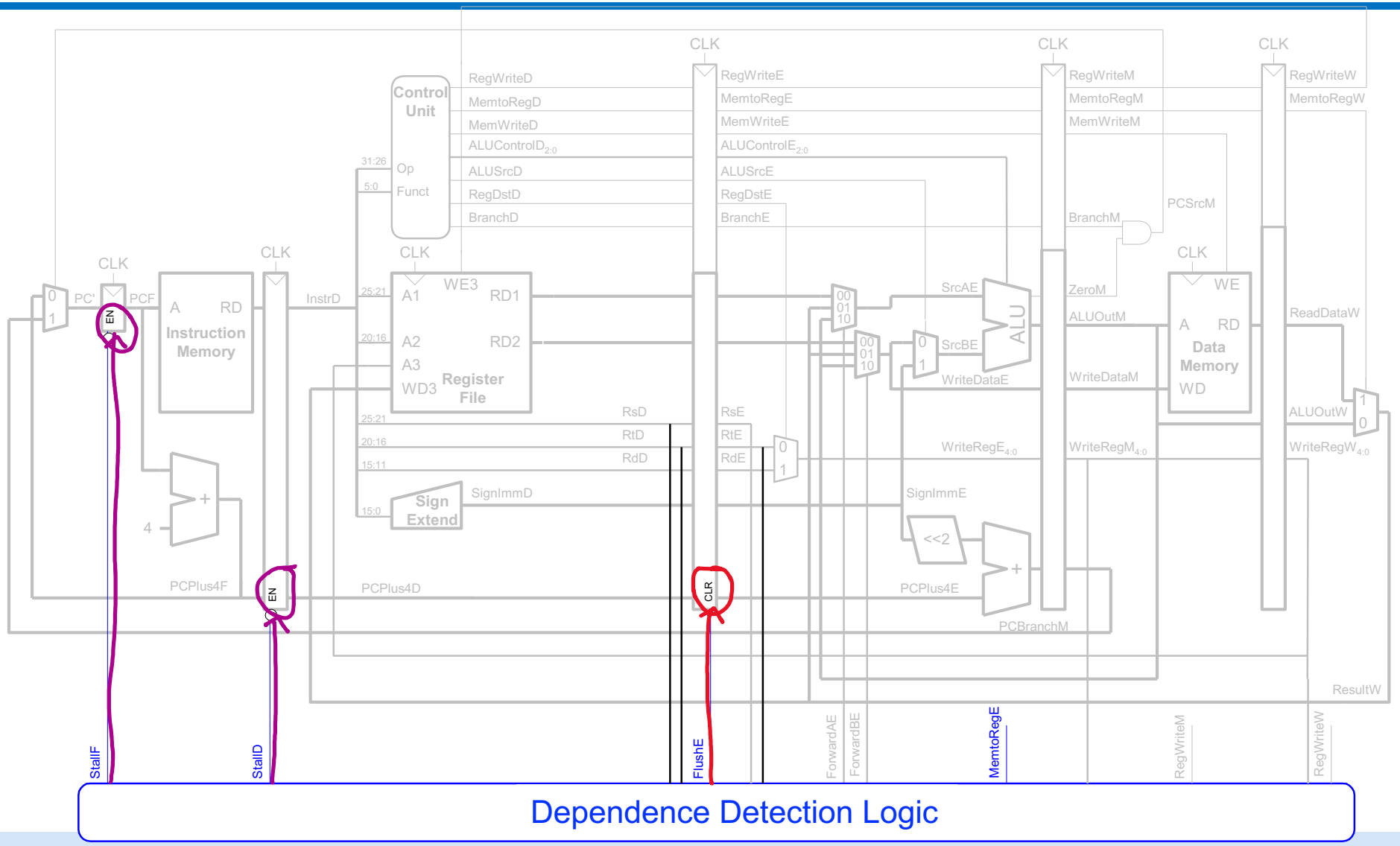
  → its result cannot be forwarded to the Execute stage of the next instruction

  unless we want a long critical path → breaks critical path design principle

Stalling Necessary for MEM-EX Dependence



Stalling and Dependence Detection Hardware



Hardware Needed for Stalling

- Stalls are supported by adding
  - enable inputs (EN) to the Fetch and Decode pipeline registers
  - synchronous reset/clear (CLR) input to the Execute pipeline register
    - or an INV bit associated with each pipeline register, indicating that contents are INValid

- When a lw stall occurs
  - Keep the values in the Decode and Fetch stage pipeline registers
    - StallD and StallF are asserted
  - Clear the contents of the Execute stage register, introducing a bubble
    - FlushE is also asserted

A Special Case of Data Dependence

- Control dependence
  - Data dependence on the Instruction Pointer / Program Counter

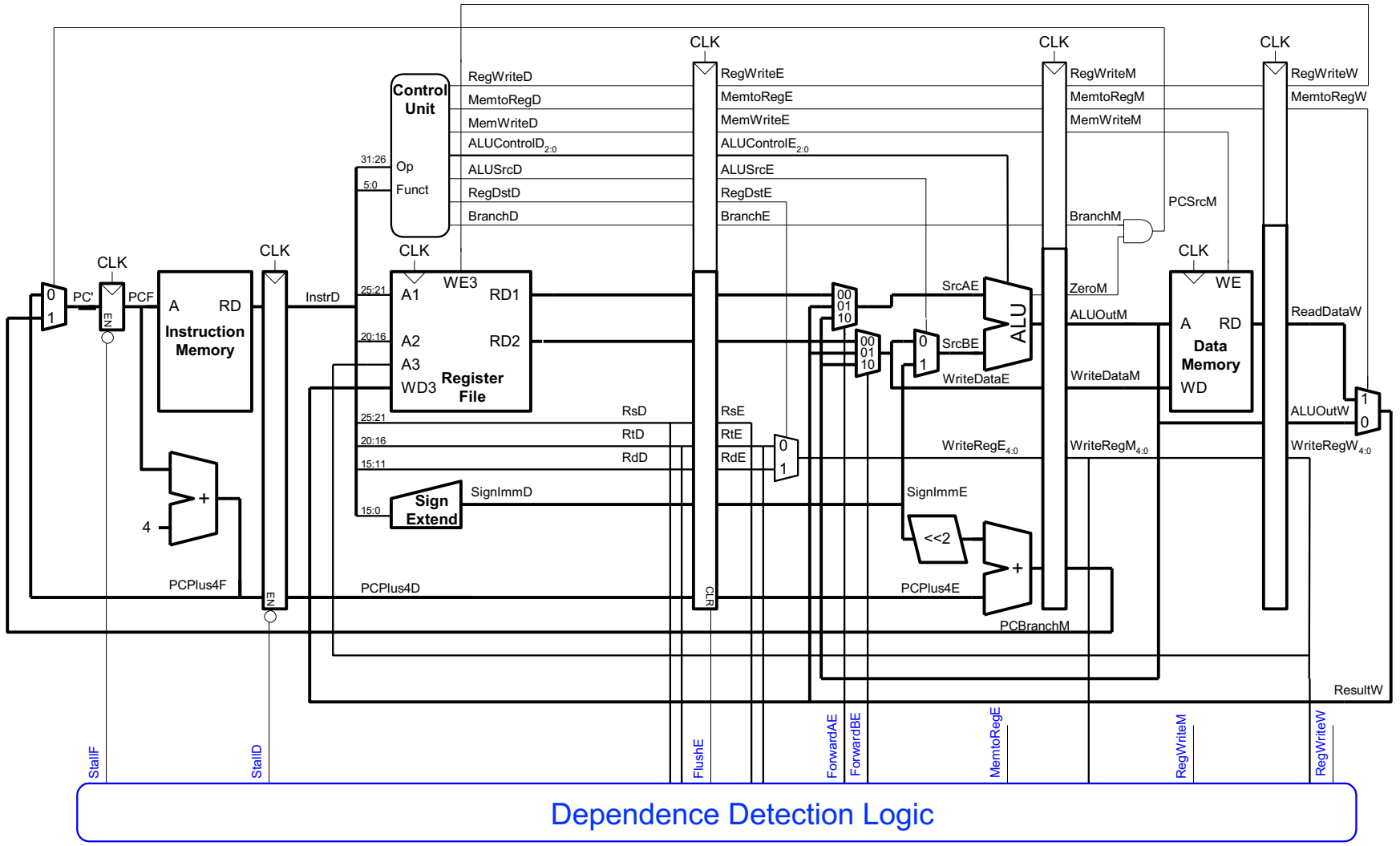
Control Dependence

- Question: What should the fetch PC be in the next cycle?
- Answer: The address of the next instruction
  - All instructions are control-dependent on previous ones. Why?
- If the fetched instruction is a non-control-flow instruction:
  - Next Fetch PC is the address of the next-sequential instruction
  - Easy to determine if we know the size of the fetched instruction (e.g., PC = PC+4 in MIPS)
- If the instruction that is fetched is a control-flow instruction:
  - How do we determine the next Fetch PC?
- In fact, how do we know whether or not the fetched instruction is a control-flow instruction?

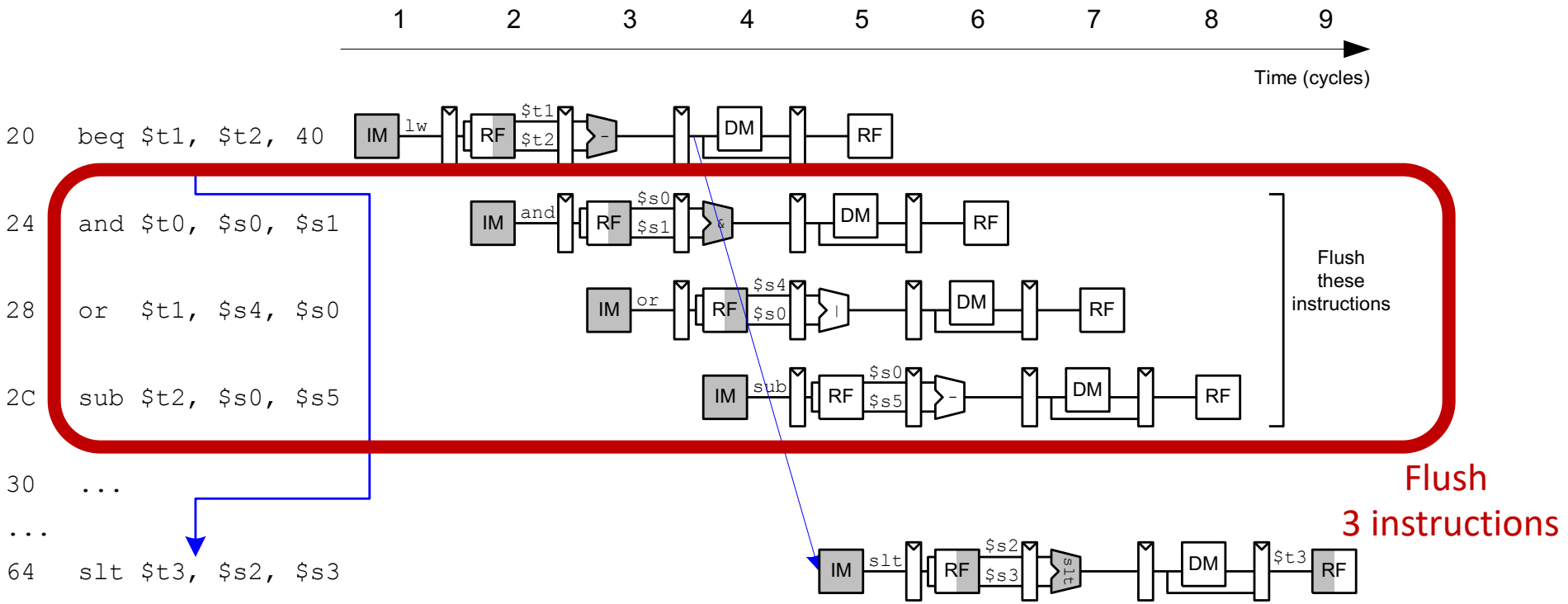
Branch Prediction

- Special case of data dependence: dependence on PC
- beq:
  - Conditional branch is not resolved until the fourth stage of the pipeline
  - **Instructions after the branch are fetched before branch is resolved**
  - Simple "**branch prediction**" example:
    - Always predict that the next sequential instruction is fetched
    - Called "**Always not taken**" prediction
    - Flush (invalidate) "not-taken path" instructions if the branch is taken
- Branch misprediction penalty
  - **number of instructions flushed when branch is incorrectly predicted**
  - Penalty can be reduced by resolving the branch earlier
    - Called "**Early branch resolution**"

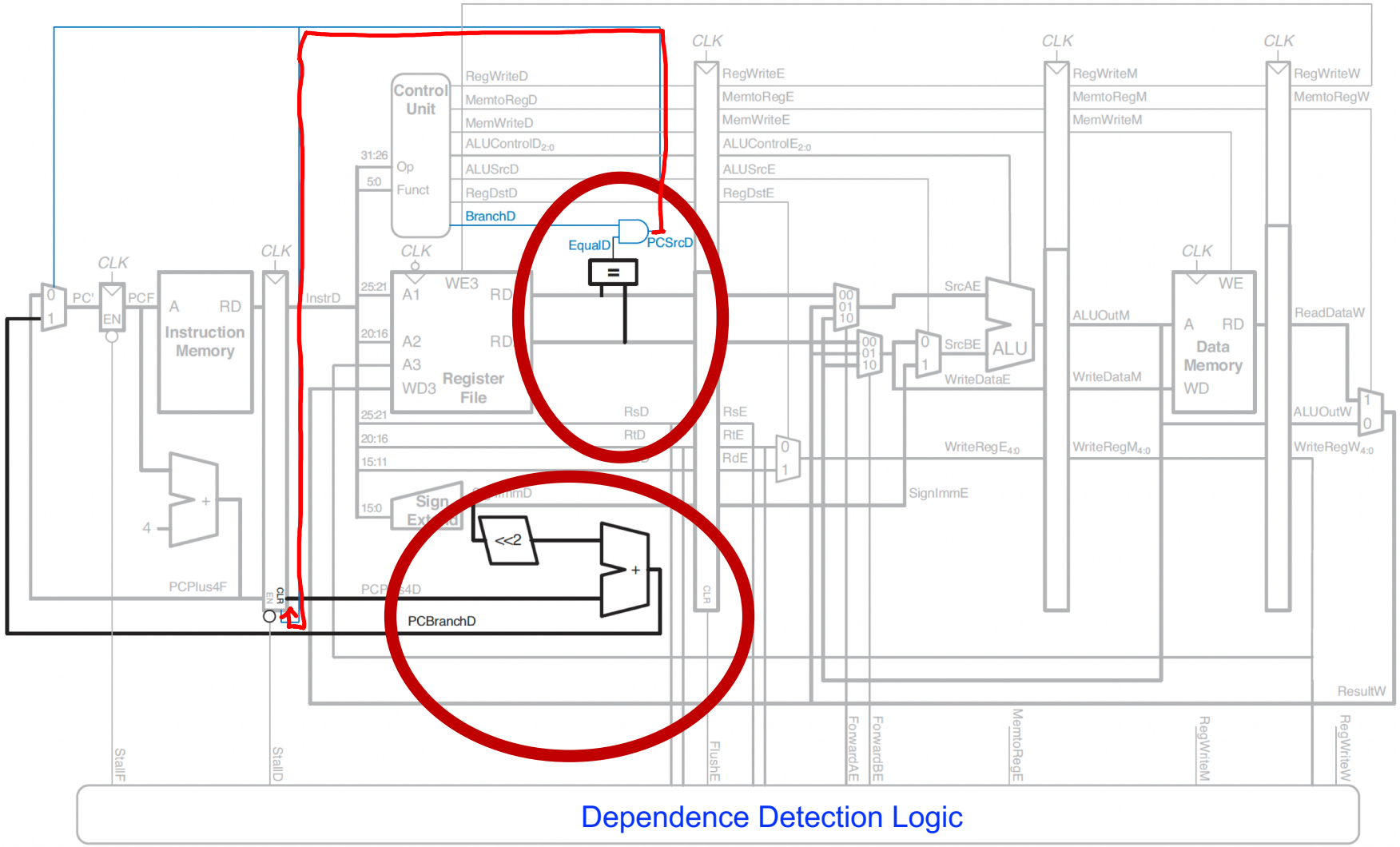
Our Pipeline So Far



Control Dependence: Flush on Misprediction

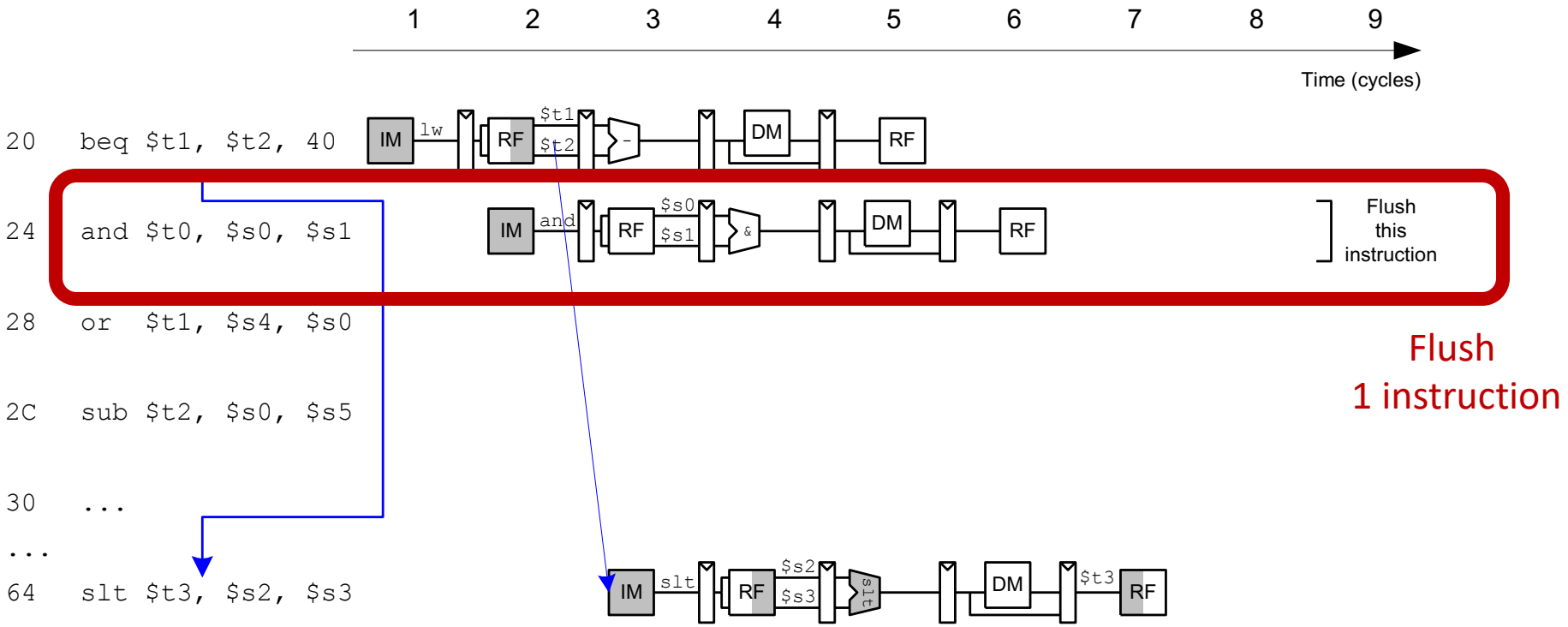


Pipeline with Early Branch Resolution



Need to calculate branch target and condition in the Decode Stage

Early Branch Resolution



Flush
1 instruction

Early Branch Resolution: Good Idea?

- Advantages
  - Reduced branch misprediction penalty
    - → Reduced CPI (cycles per instruction)

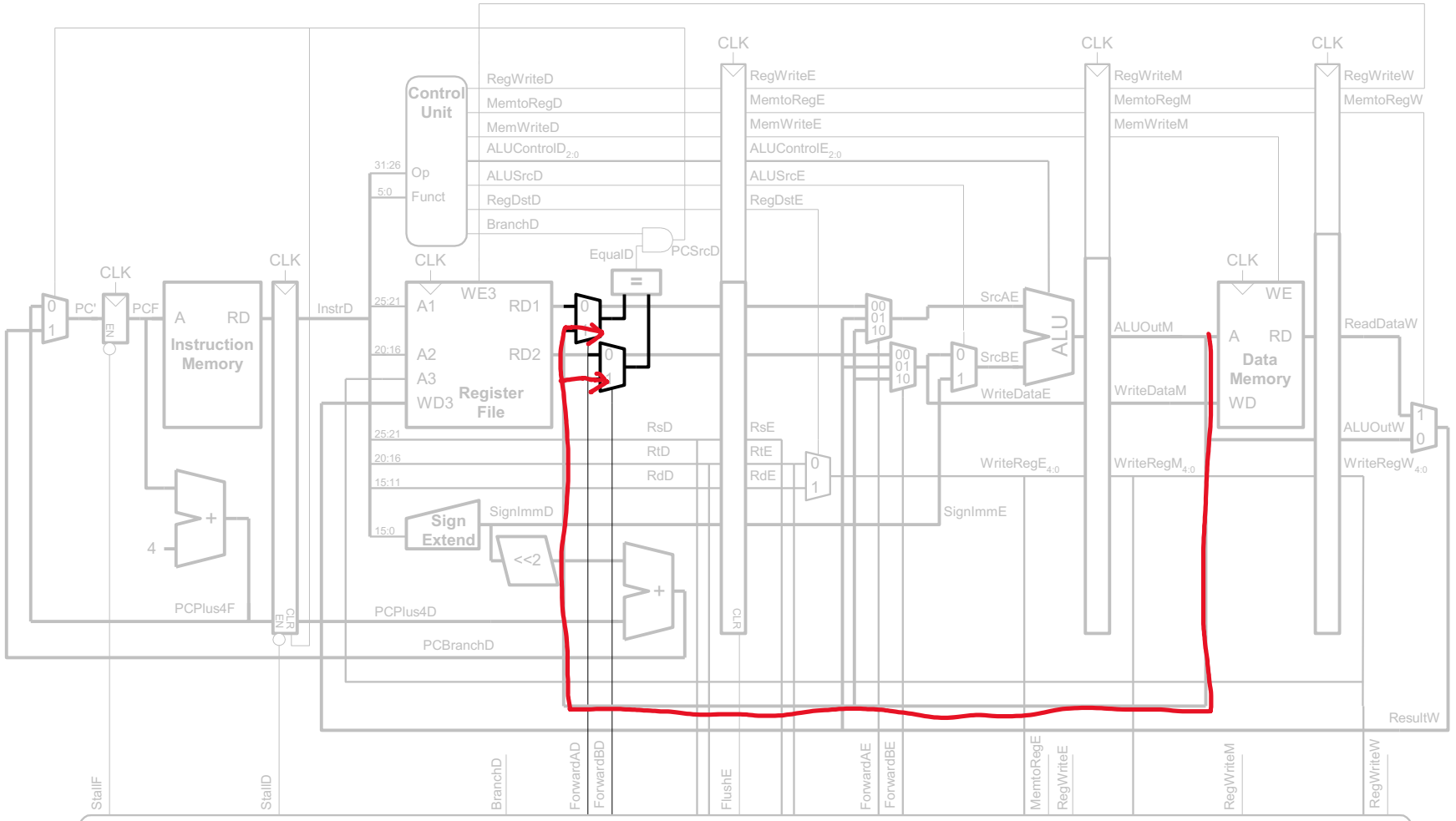
- Disadvantages
  - Potential increase in clock cycle time?
    - → Higher clock period and lower frequency?
  - Additional hardware cost
    - → Specialized and likely not used by other instructions

Recall: Performance Analysis Basics

- Execution time of a single instruction
  - **{CPI} x {clock cycle time}**
    - CPI: Number of cycles it takes to execute an instruction

- Execution time of an entire program
  - Sum over all instructions [{CPI} x {clock cycle time}]
  - **{# of instructions} x {Average CPI} x {clock cycle time}**

Data Forwarding for Early Branch Resolution



Forwarding and Stalling Hardware Control

```
// Forwarding logic:
assign ForwardAD = (rsD != 0) & (rsD == WriteRegM) & RegWriteM;
assign ForwardBD = (rtD != 0) & (rtD == WriteRegM) & RegWriteM;

//Stalling logic:
assign lwstall = ((rsD == rtE) | (rtD == rtE)) & MemtoRegE;

assign branchstall = (BranchD & RegWriteE &
                        (WriteRegE == rsD | WriteRegE == rtD))
                        |
                      (BranchD & MemtoRegM &
                      (WriteRegM == rsD | WriteRegM == rtD));

// Stall signals;
assign StallF = lwstall | branchstall;
assign StallD = lwstall | branchstall;
assign FLushE = lwstall | branchstall;
```

Final Pipelined MIPS Processor

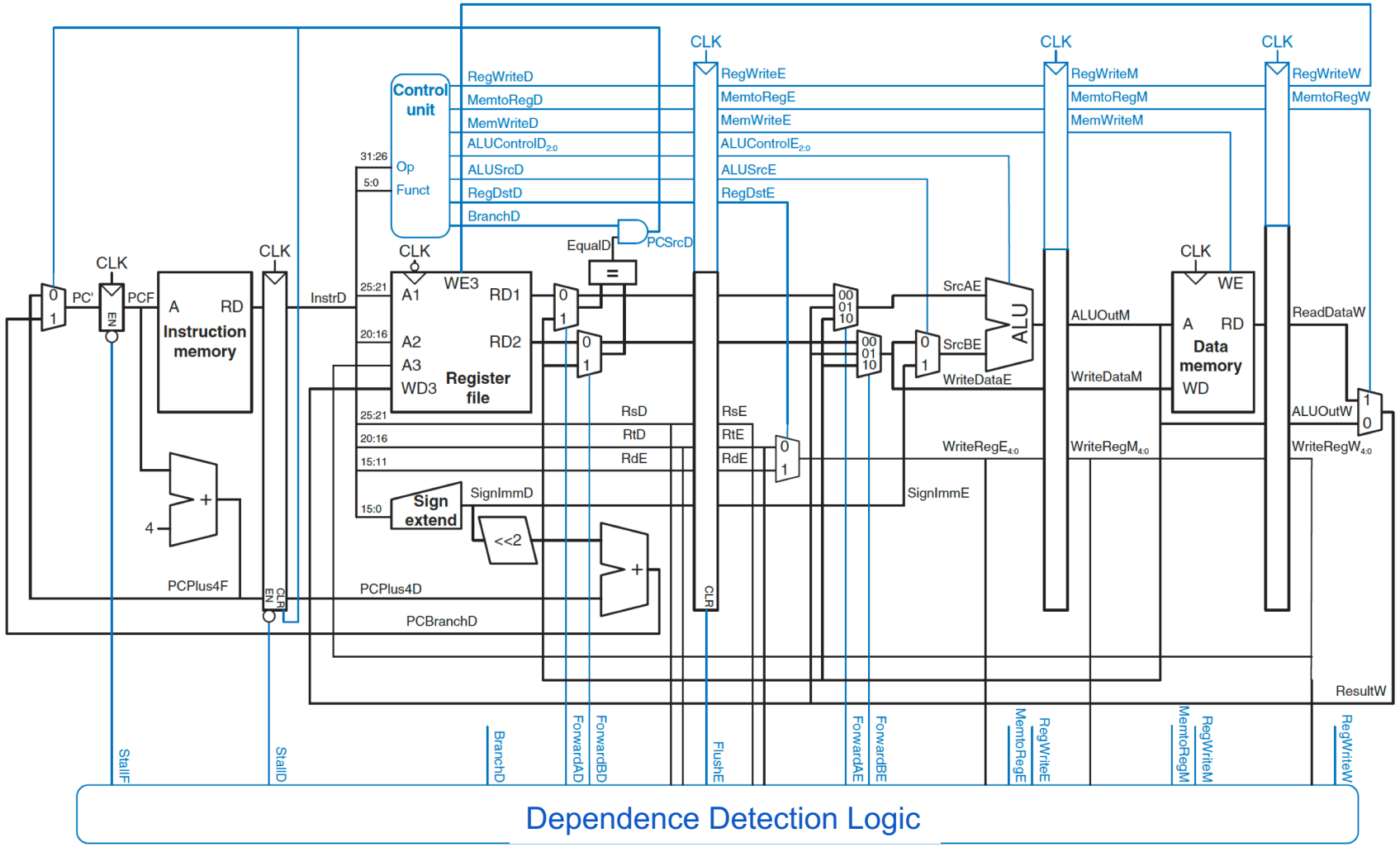


Figure 7.58 Pipelined processor with full hazard handling

Doing Better: Smarter Branch Prediction

- Guess whether or not the branch will be taken
  - Backward branches are usually taken (loops iterate many times)
  - The history of whether the branch was previously taken can improve the guess
- Accurate branch prediction reduces the fraction of branches requiring a flush
- Many sophisticated techniques are employed in modern processors
  - Including simple machine learning methods (perceptrons)

KING'S College LONDON

Hardware Design

Lecture 6: Pipelined Processor Design

Dr. Haiyu Mao

05.03.2026