Pipelining

The limitation in Multi-cycle design.

Limited concurrency

Some hardware resources are idle during different phases of instruction process cycle.

For example:

- "Fetch" logic is idle when an instruction is being "decoded" or "executed"
- Most of the datapath is idle when a memory access is happening.

Goal: ① More concurrency → Higher instruction throughout (more `work` completed in one cycle)

② Increase throughput with little increase in cost

(hardware cost, in case of instruction processing)

i. Repetition of identical operations

ii. Repetition of independent operations

iii. Uniformly partitionable suboperations (统一划分操作)

Throughput = IPC (Instruction per cycle)

Multi-cycle: 4 cycles per instruction

pipeline: 4 cycles per 4 instructions (steady state)

(1 instruction completed every 1 cycle)

More Realistic Pipeline: Throughput

- Nonpipelined version with delay T

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

- k-stage pipelined version

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

Register delay reduces throughput (sequencing overhead b/w stages)

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

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

More Realistic Pipeline: Cost

- Nonpipelined version with combinational cost G

  Cost = G+R where R = register cost

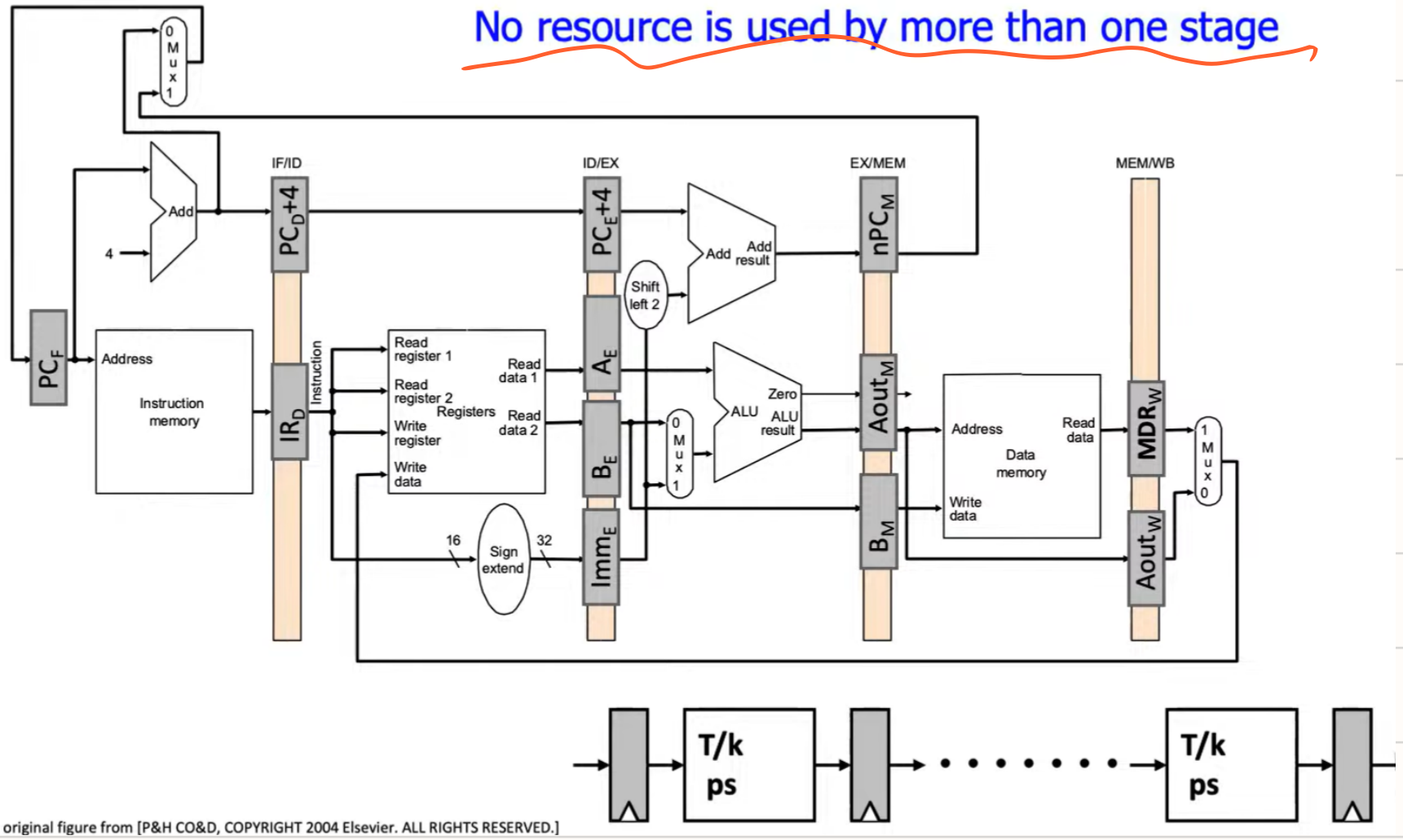
- k-stage pipelined version

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

  Registers increase hardware cost

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

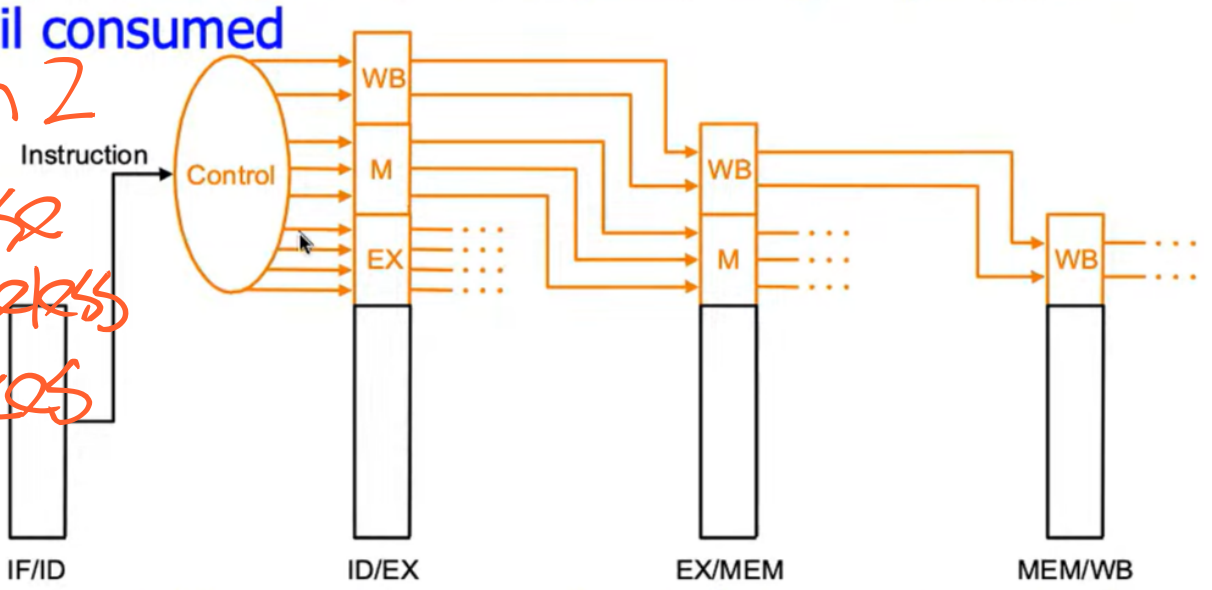
Enabling Pipelined Processing: Pipeline Registers



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

In fact, option 2 is better because it saved the useless hardware resources

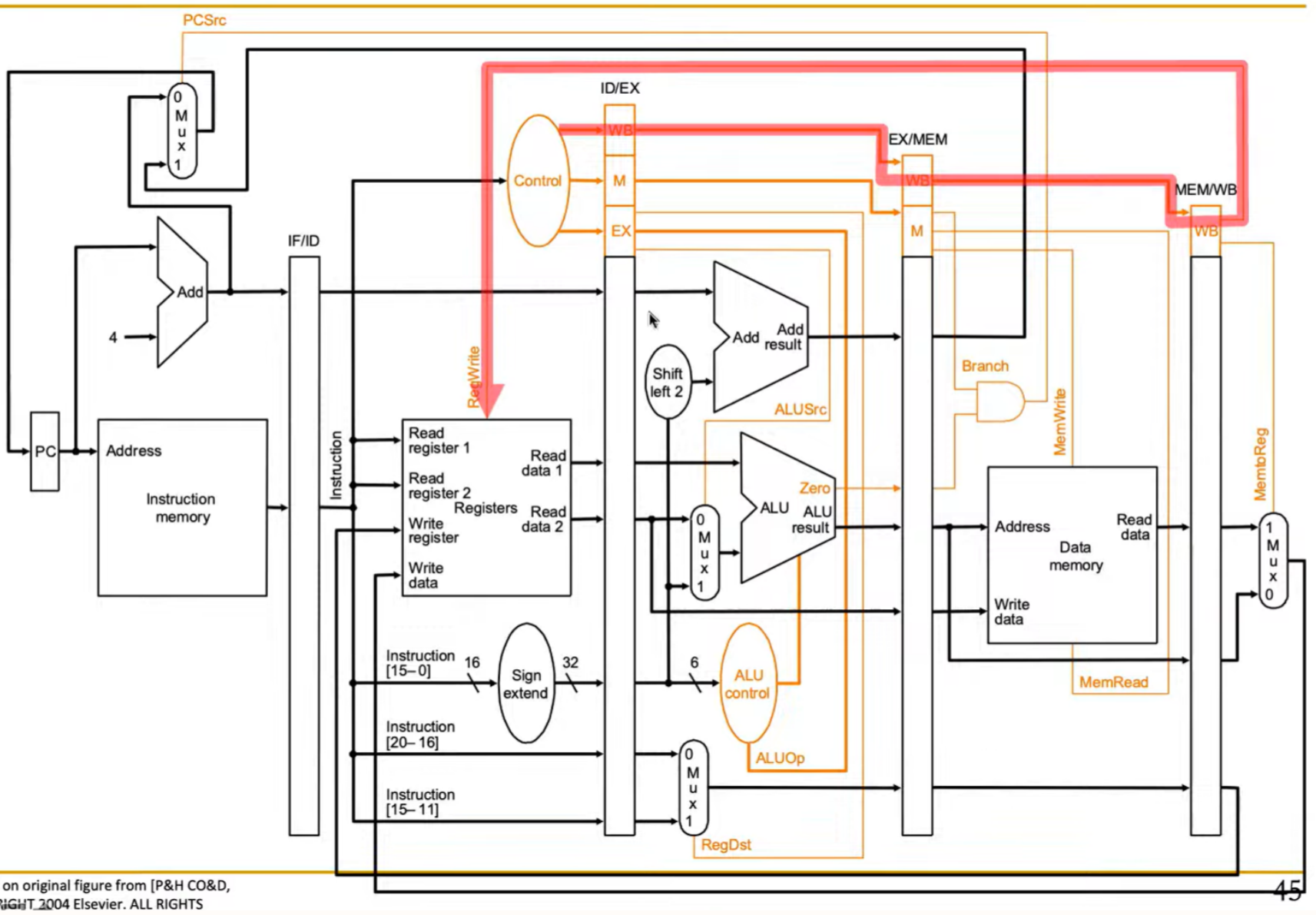


- ⇒ Option 2: carry relevant "instruction word/field" down the pipeline and decode locally within each or in a previous stage

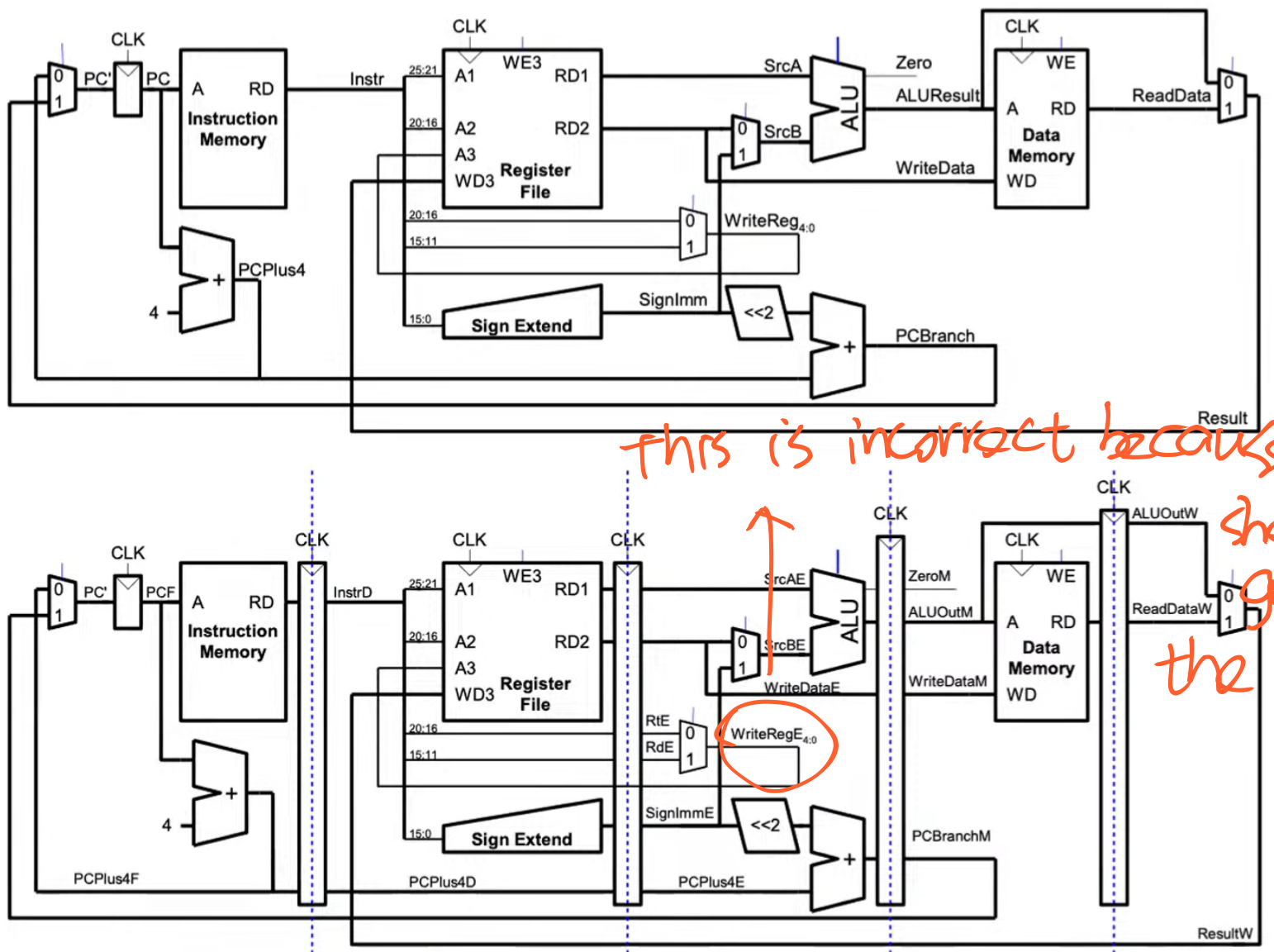
Which one is better?

Pipelined Control Signals

Below is Option 1



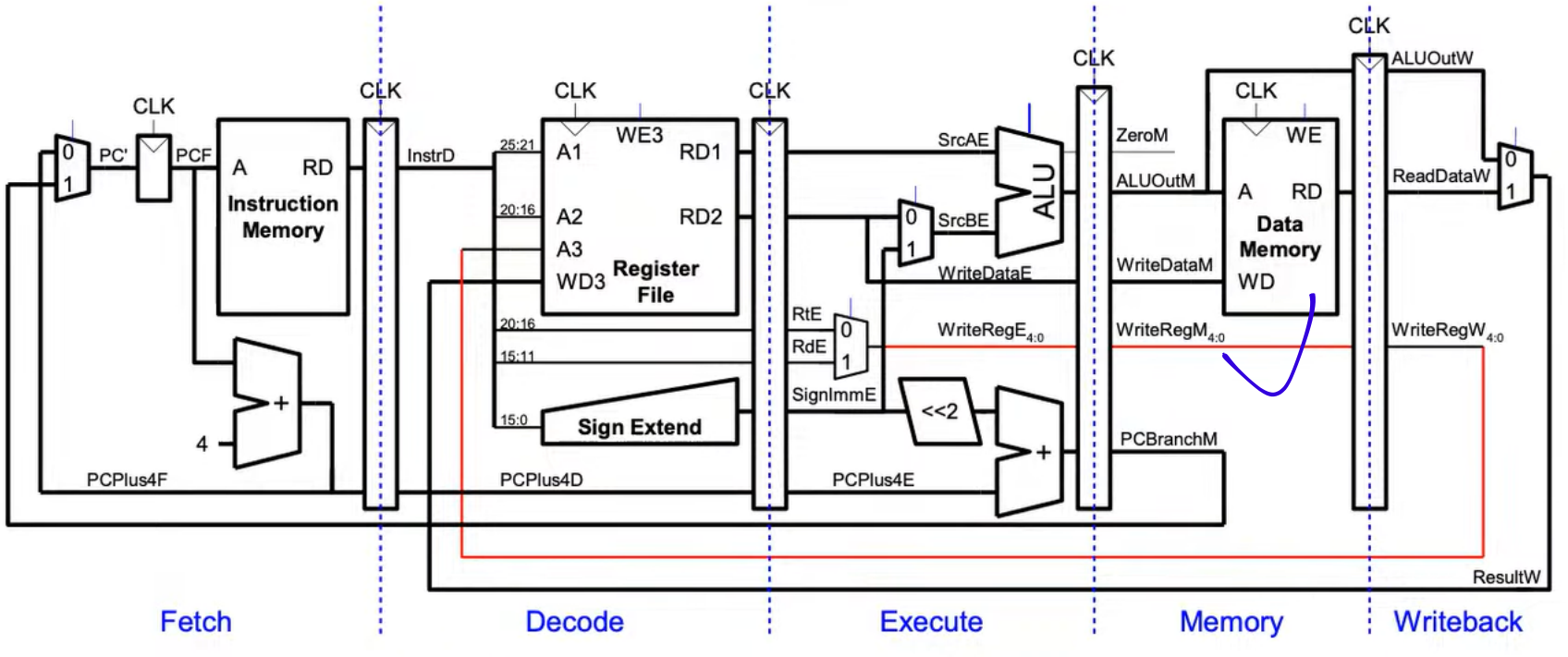
Another Example: Single-Cycle and Pipelined



this is incorrect because it shouldn't go back to the previous stage

Fetch | Decode | Execute | Memory | Writeback

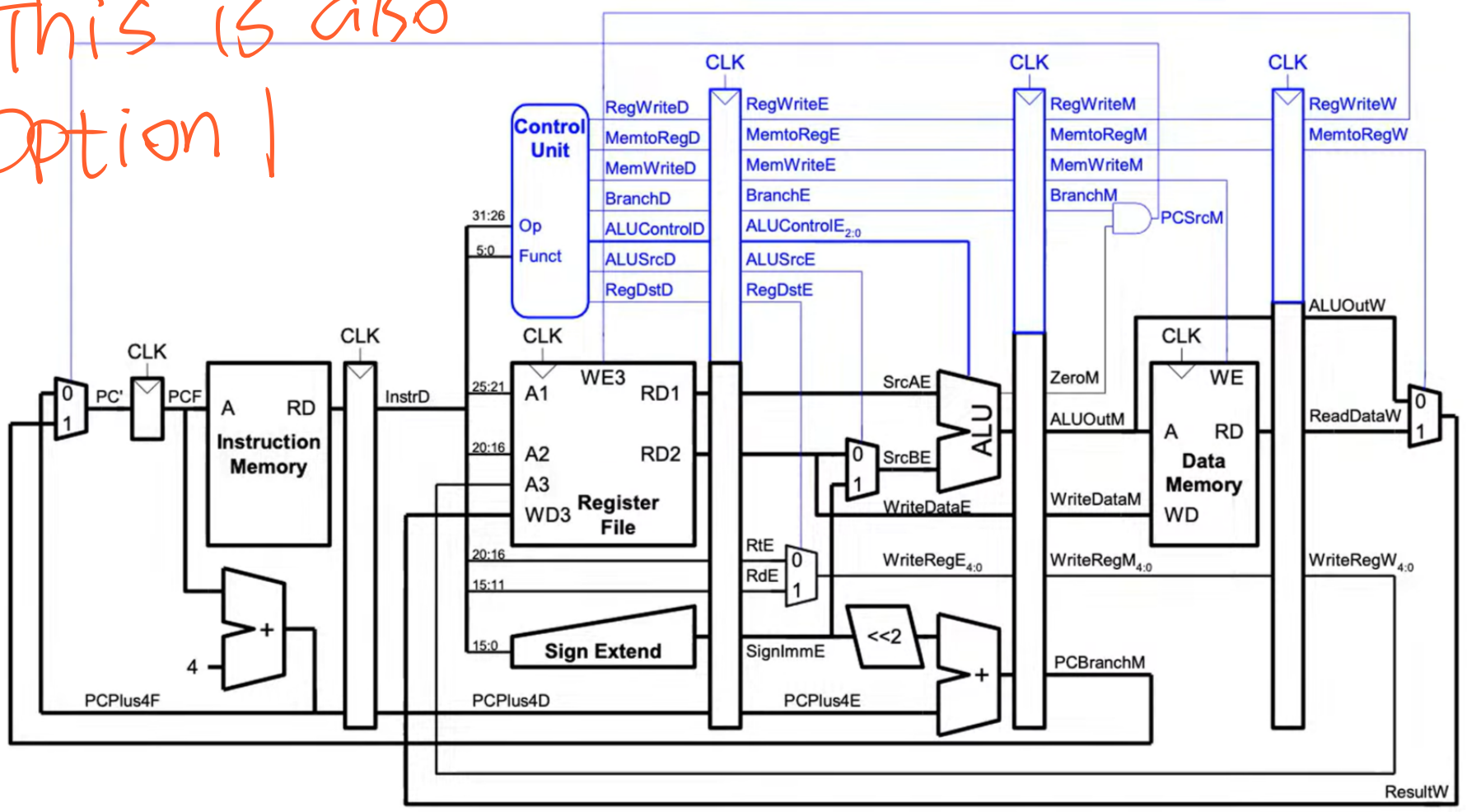
Another Example: Correct Pipelined Datapath



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

Another Example: Pipelined Control

This is also Option 1



- **Same control unit as single-cycle processor**
- **Control delayed to proper pipeline stage**

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)

Dependences and their types

Also called "dependency" or less desirably "hazard"

they dictate ordering requirements between instrs.

Two type:

- Data dependence
- Control dependence

Resource contention is sometimes called resource dependence (However, this is not dictated by program semantics)

Handling Resource Contention

- Happens when instructions in two pipeline stages need the same resource keeping the pipeline moving is the principle
- 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

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

false data dependence not serious

- 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)

How to Handle Data Dependences

- Anti and output dependences are easier to handle
  - write to the destination only in last stage and in program order
- Flow dependences are more interesting & challenging
- Six 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
  - 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

Advanced Pipeline
out-of-order execution

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.

eg. WAW

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 (flash-forward: think superscalar execution)

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**