

Computer Architecture

06. Pipelining & Hazards

Jianhua Li

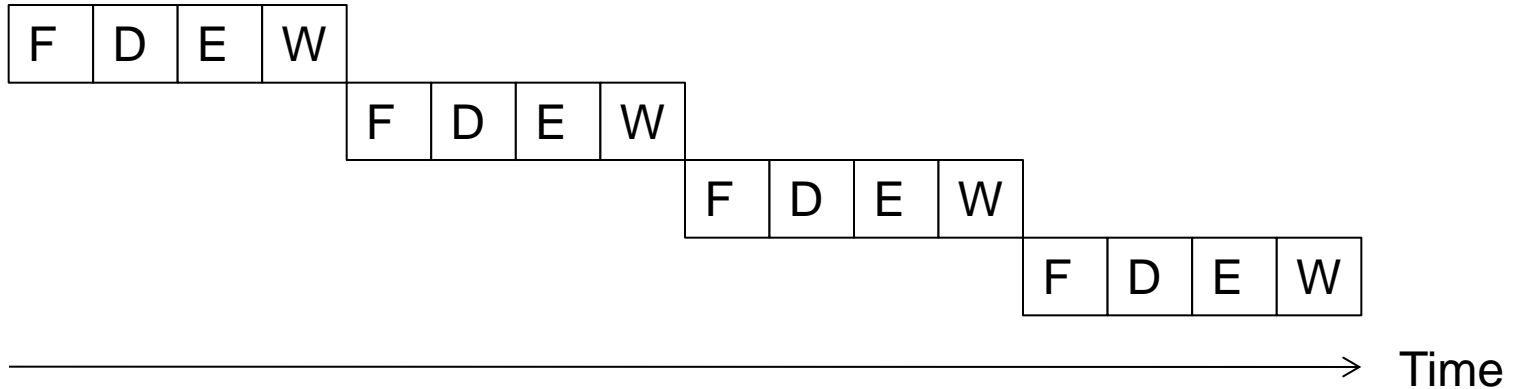
College of Computer and Information
Hefei University of Technology

复习：流水线的基本思想

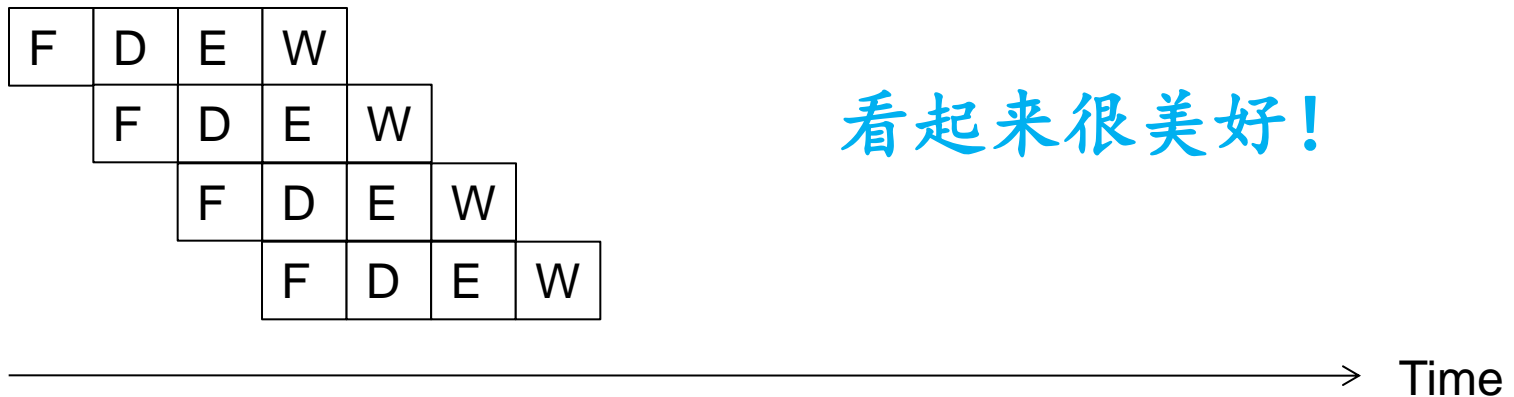
- More systematically:
 - **Pipelining** the execution of multiple instructions
- Idea:
 - Divide the instruction processing cycle into **distinct “stages”** of processing
 - Ensure there are enough hardware resources to process one instruction in each stage
 - Process a different instruction in each stage **simultaneously**
 - Instructions consecutive in program order are processed in consecutive stages
- Benefit: Increases instruction processing throughput

示例：加法操作的处理

- Multi-cycle: 4 cycles per instruction



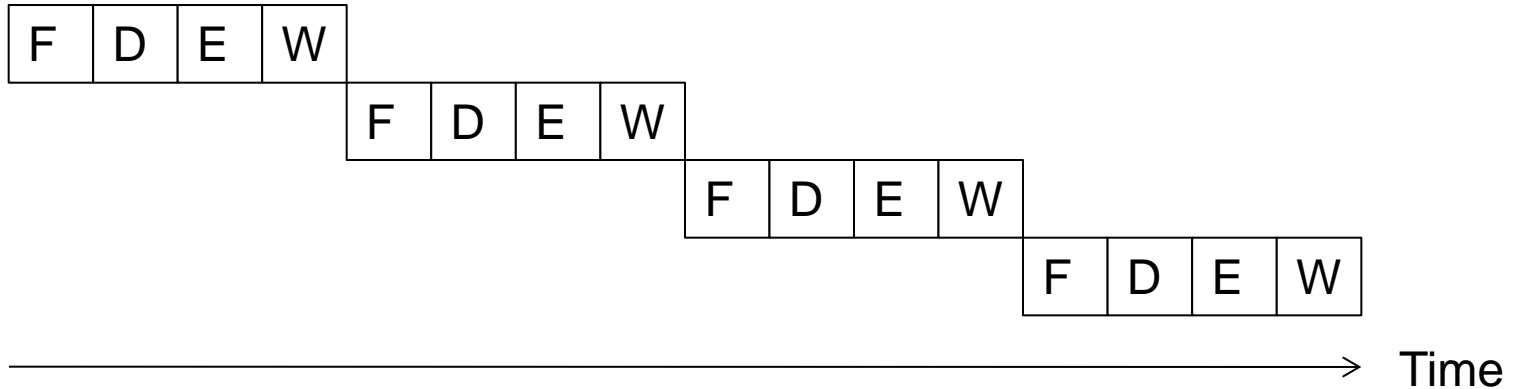
- Pipelined: 4 cycles per 4 instructions



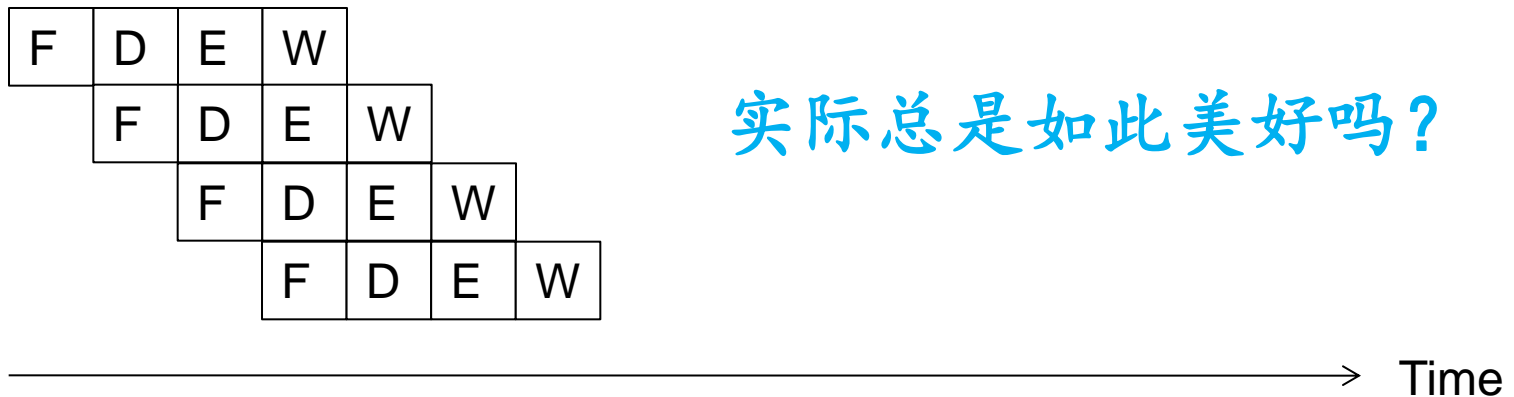
看起来很美好！

示例：加法操作的处理

- Multi-cycle: 4 cycles per instruction



- Pipelined: 4 cycles per 4 instructions

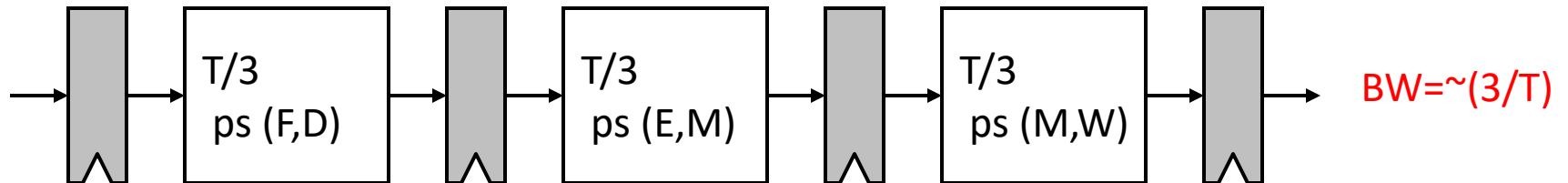
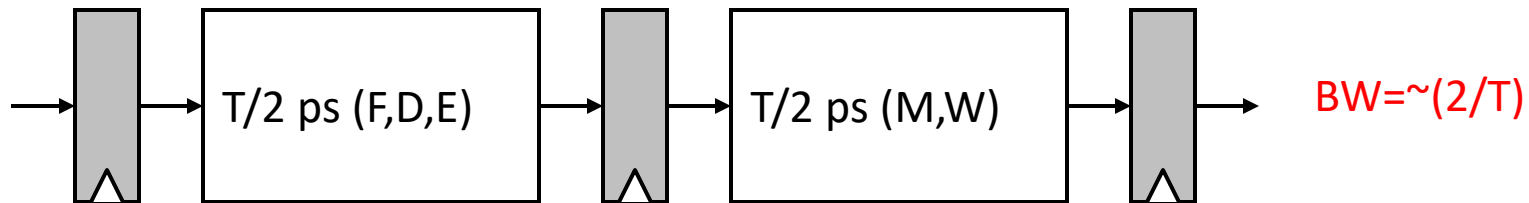
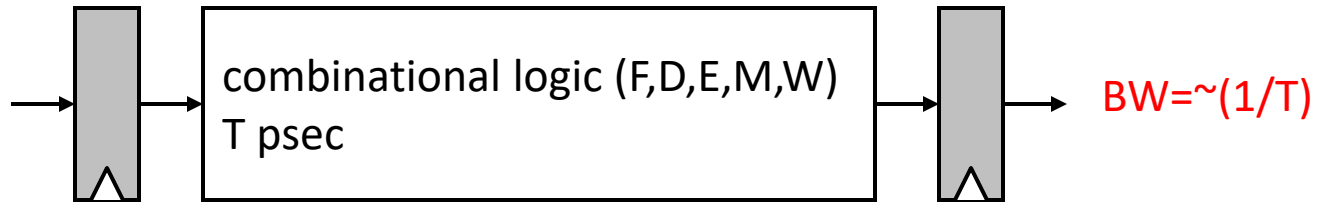


实际总是如此美好吗？

理想的流水线

- 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

理想的流水线



现实的流水线: Throughput

- Nonpipelined version with delay T

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

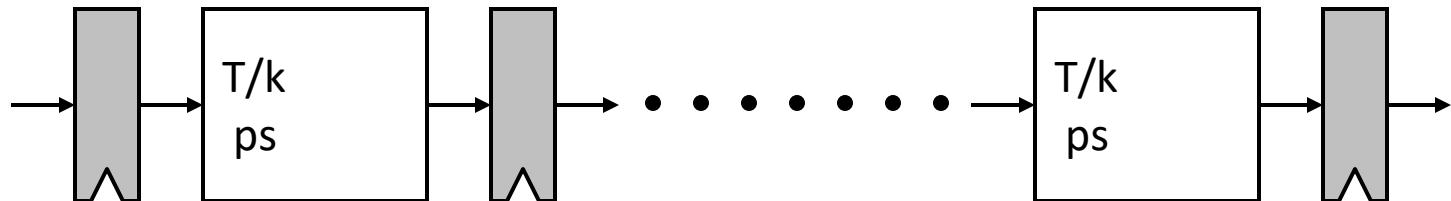


- k-stage pipelined version

Latch delay reduces throughput

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

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



现实的流水线: Cost

- Nonpipelined version with combinational cost G

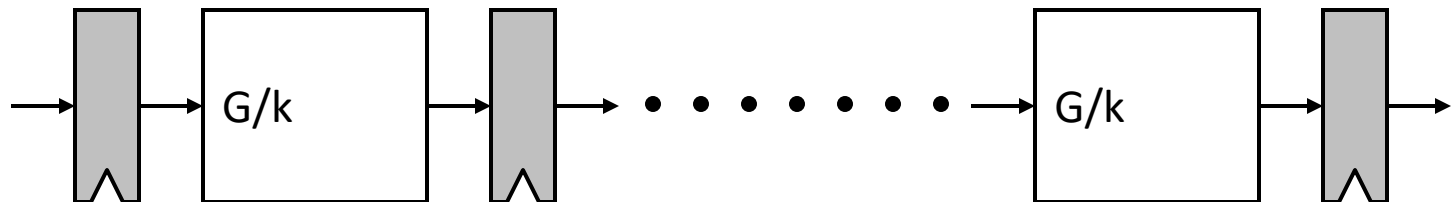
$\text{Cost} = G + L$ where L = latch cost



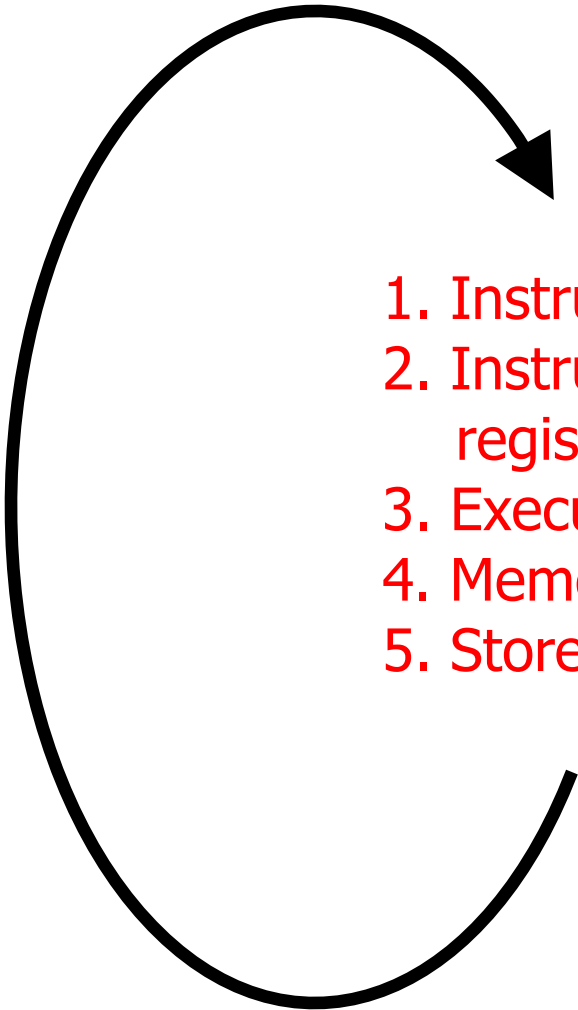
- k -stage pipelined version

Latches increase hardware cost

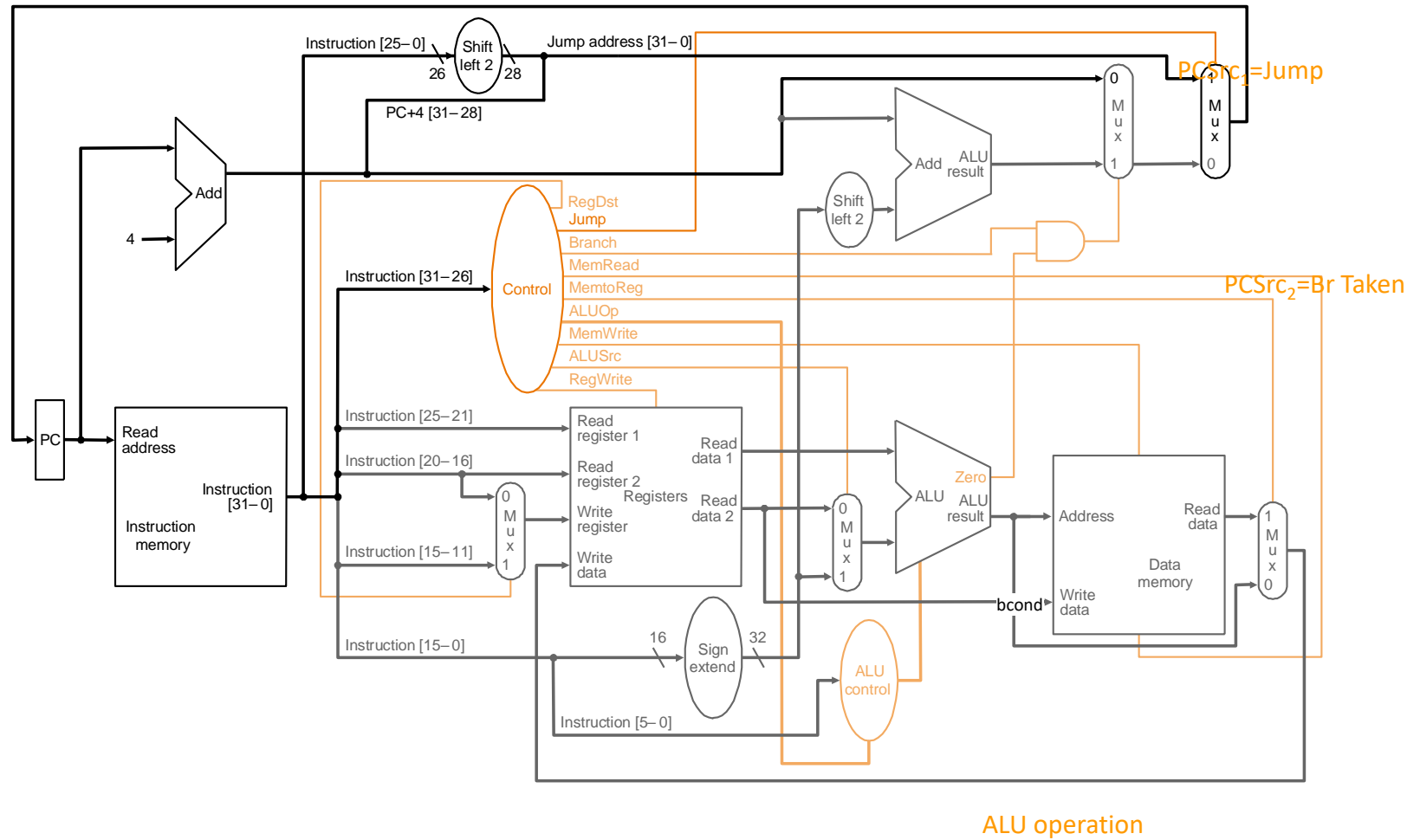
$\text{Cost}_{k\text{-stage}} = G + Lk$



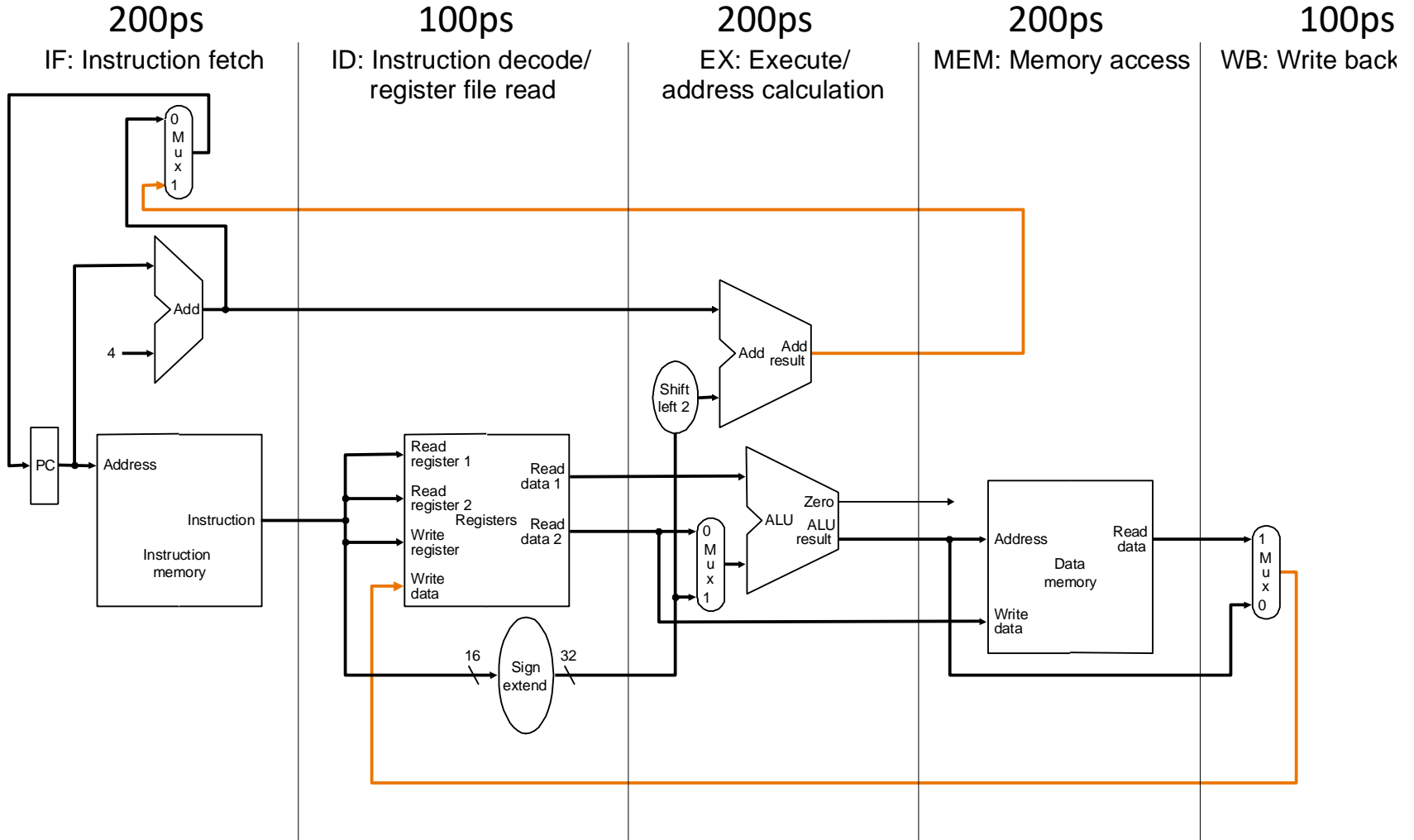
Pipeline: 指令处理过程

- 
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)

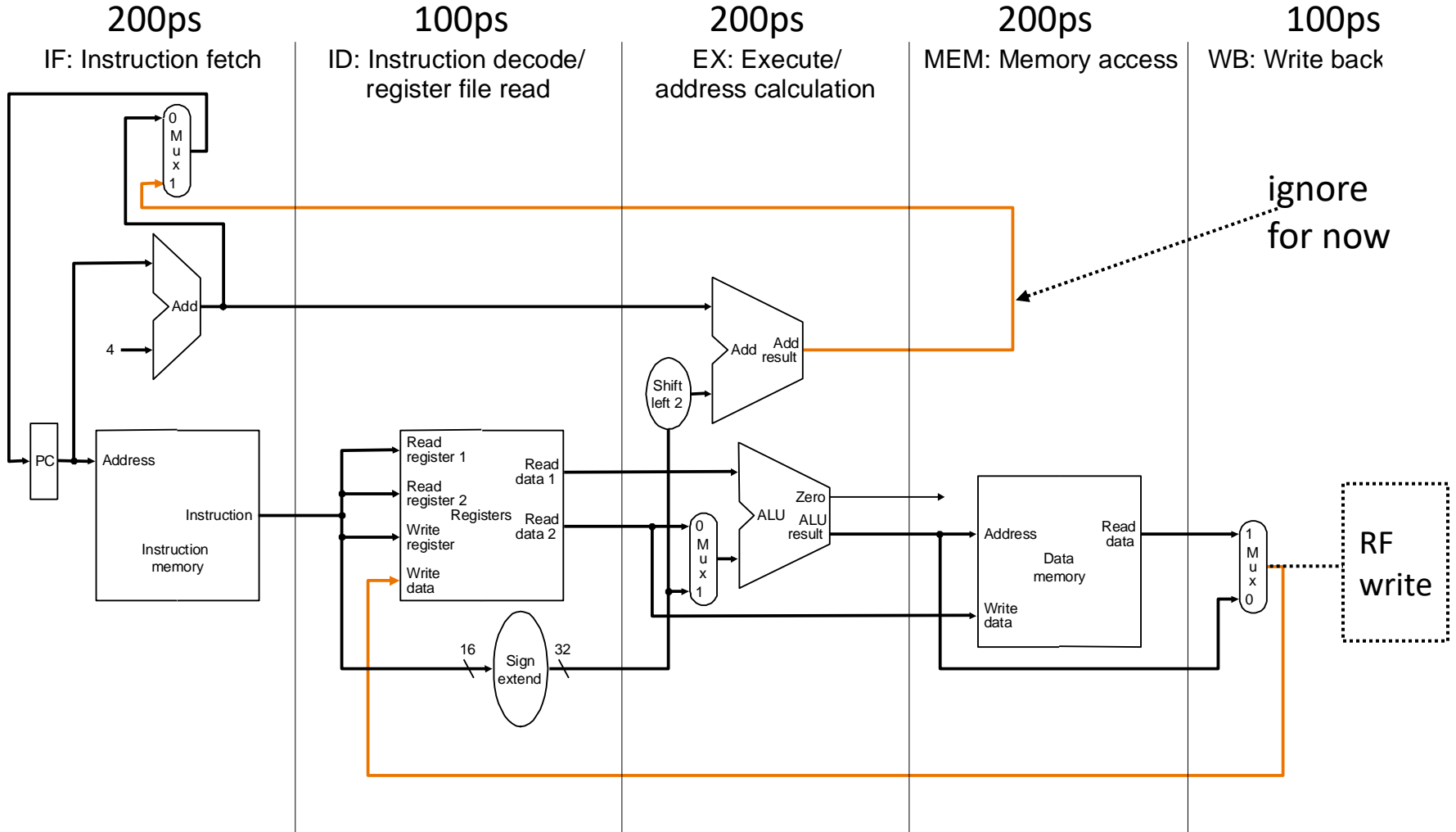
记住单周期微架构



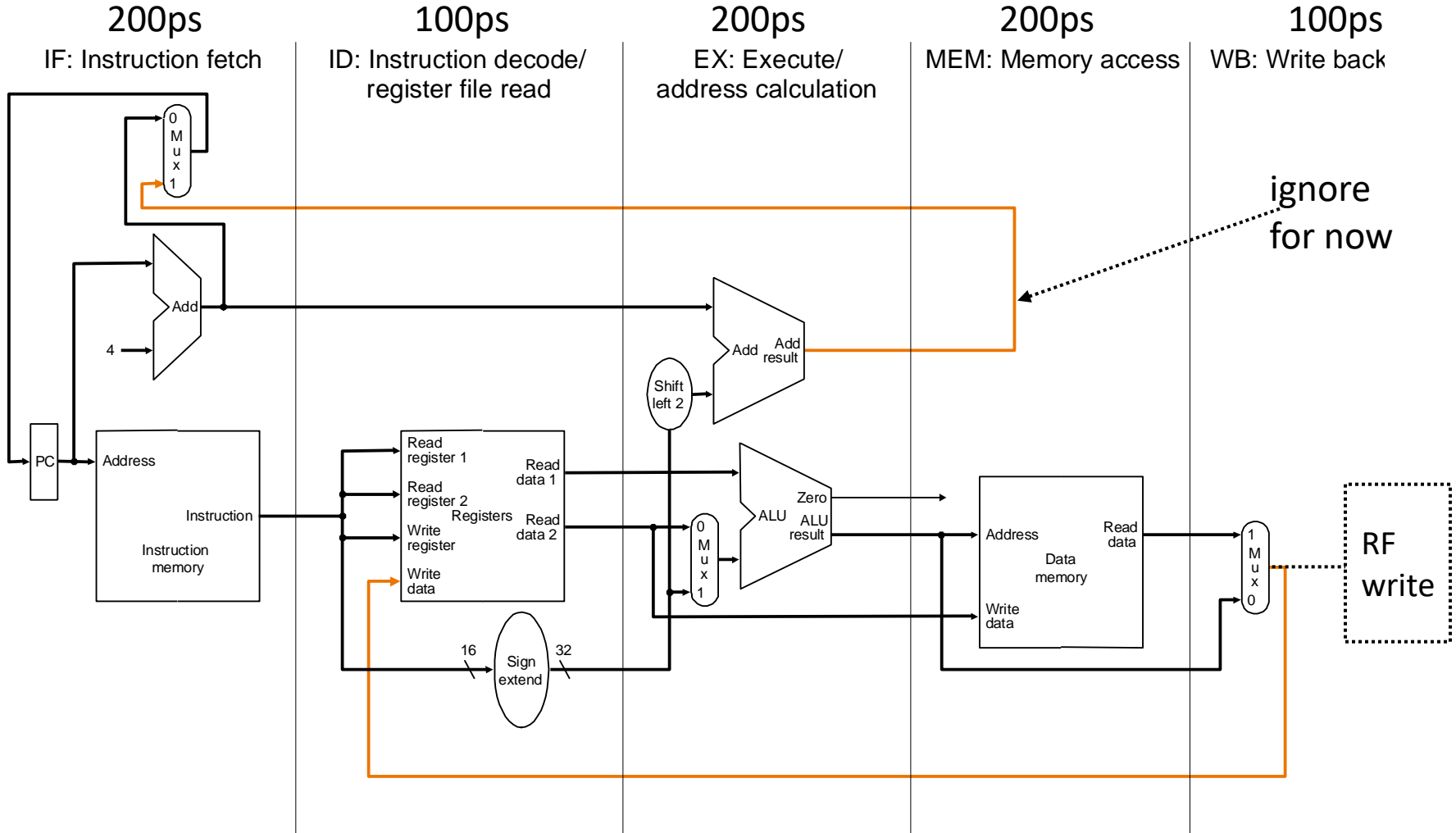
分割为阶段



分割为阶段

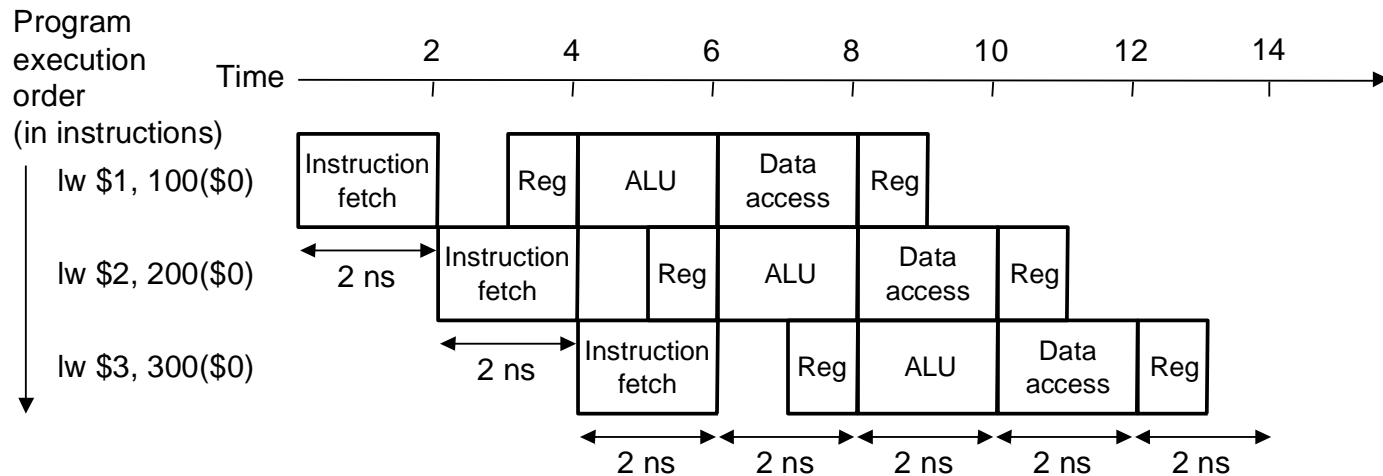
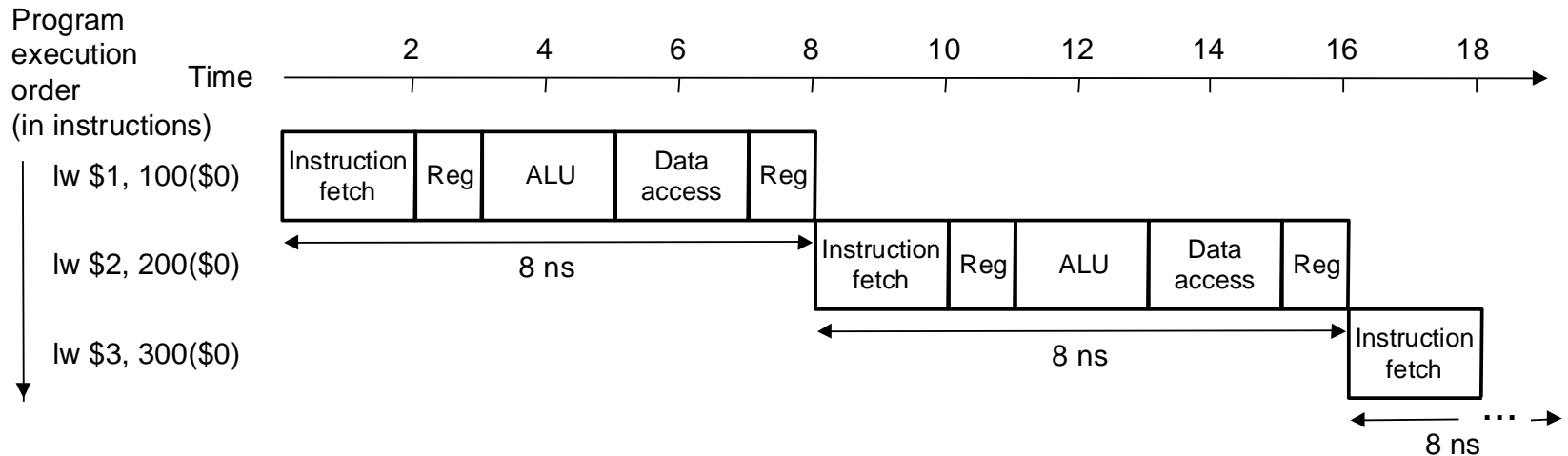


分割为阶段

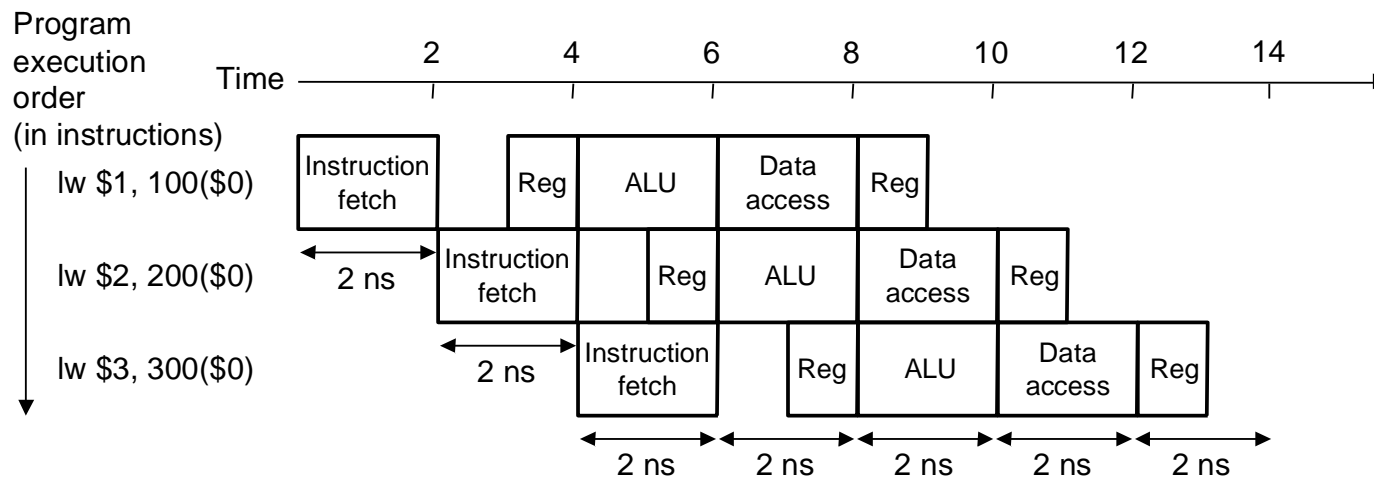
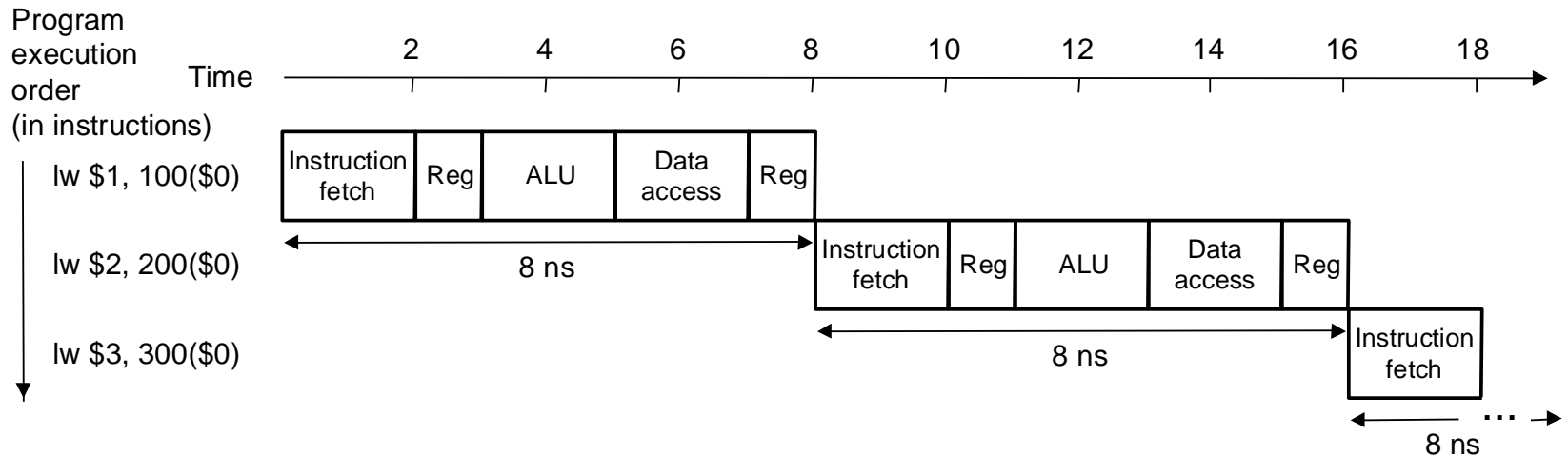


Is this the correct partitioning?
Why not 4 or 6 stages?
Why not different boundaries?

Instruction Pipeline Throughput



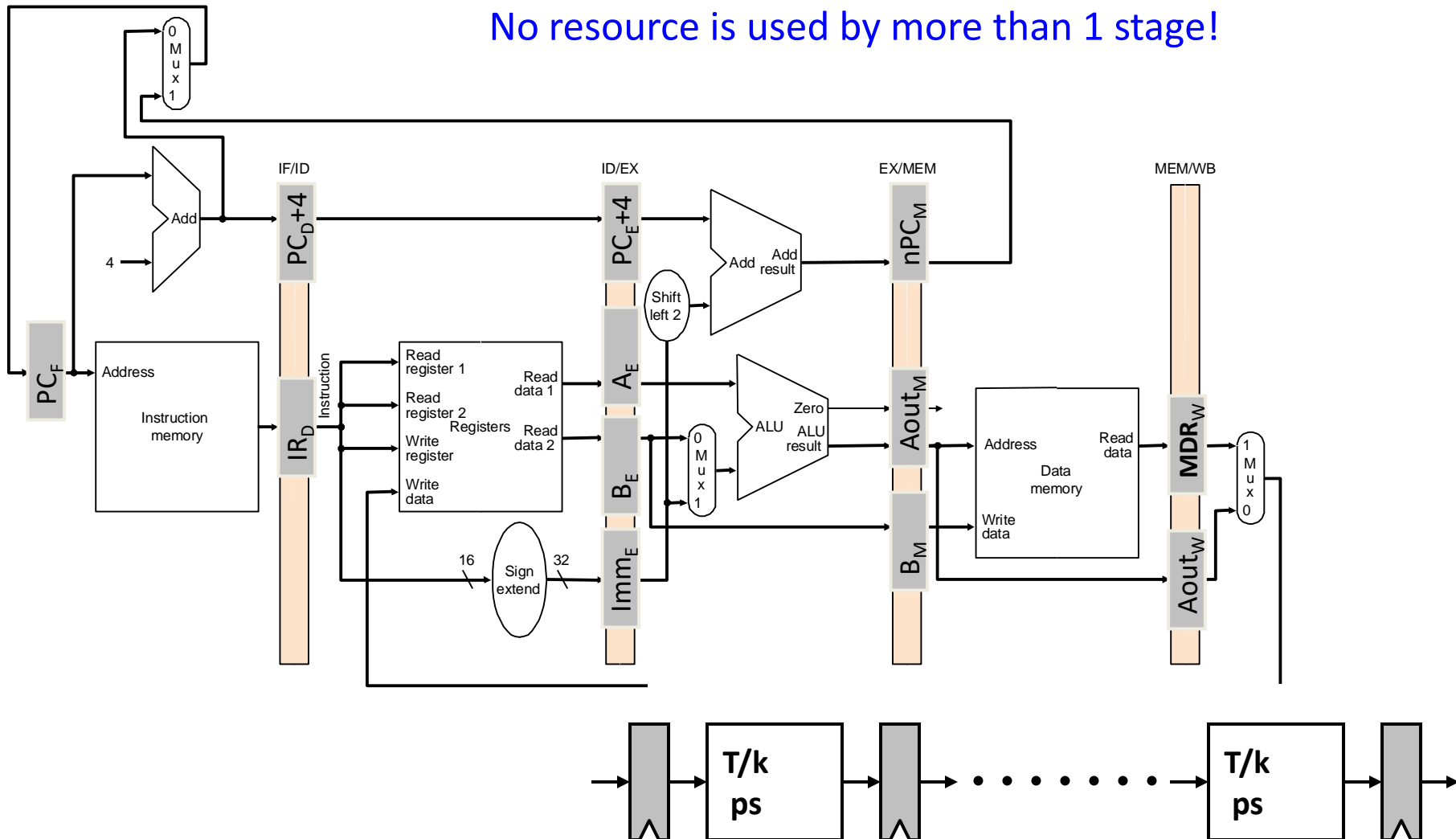
Instruction Pipeline Throughput



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

流水线的重要元素: Pipeline Registers

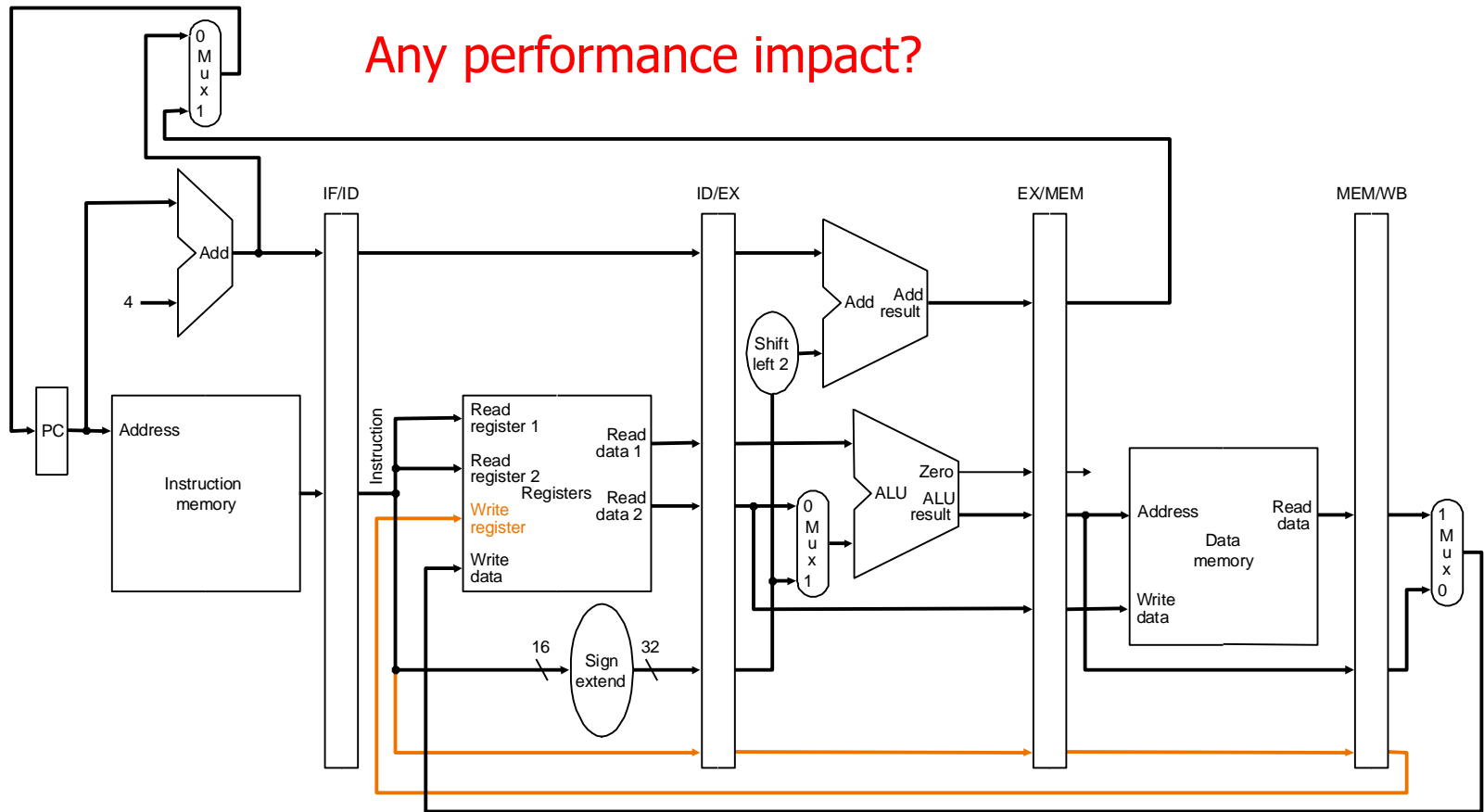
No resource is used by more than 1 stage!



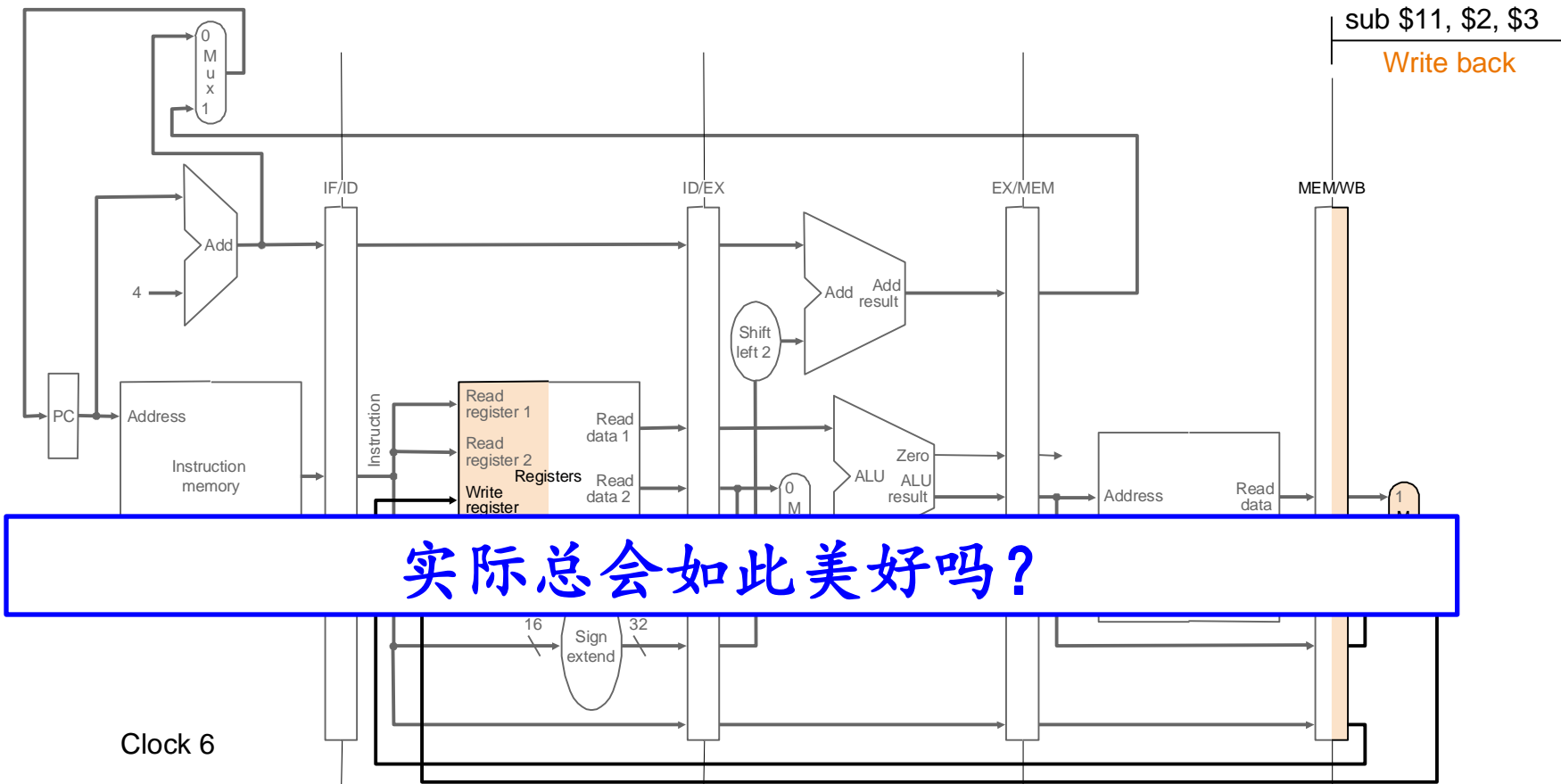
流水操作示例

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

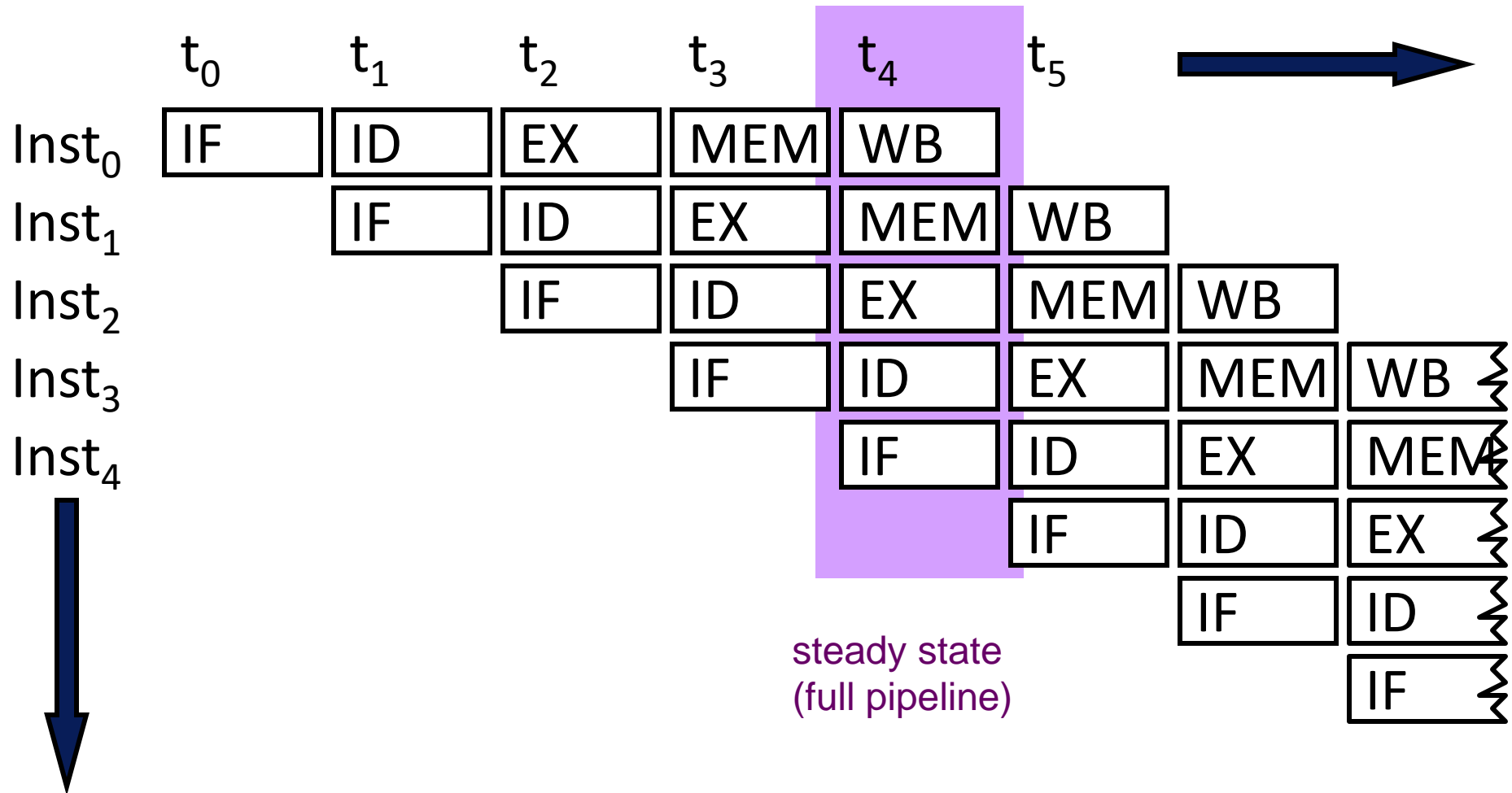
Any performance impact?



流水操作示例



描述流水操作: Operation View



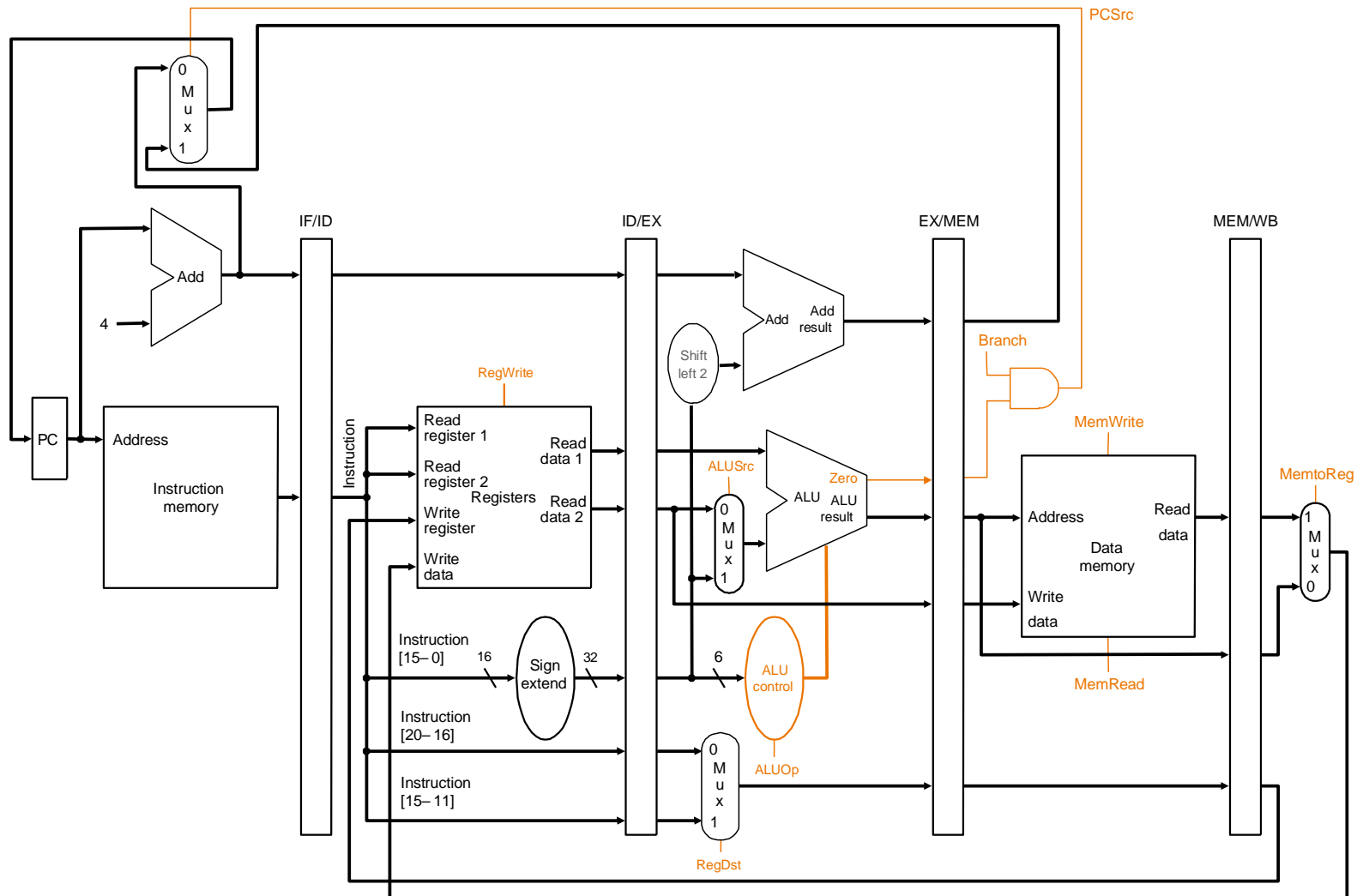
描述流水操作：时空图

	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



steady state (full pipeline)

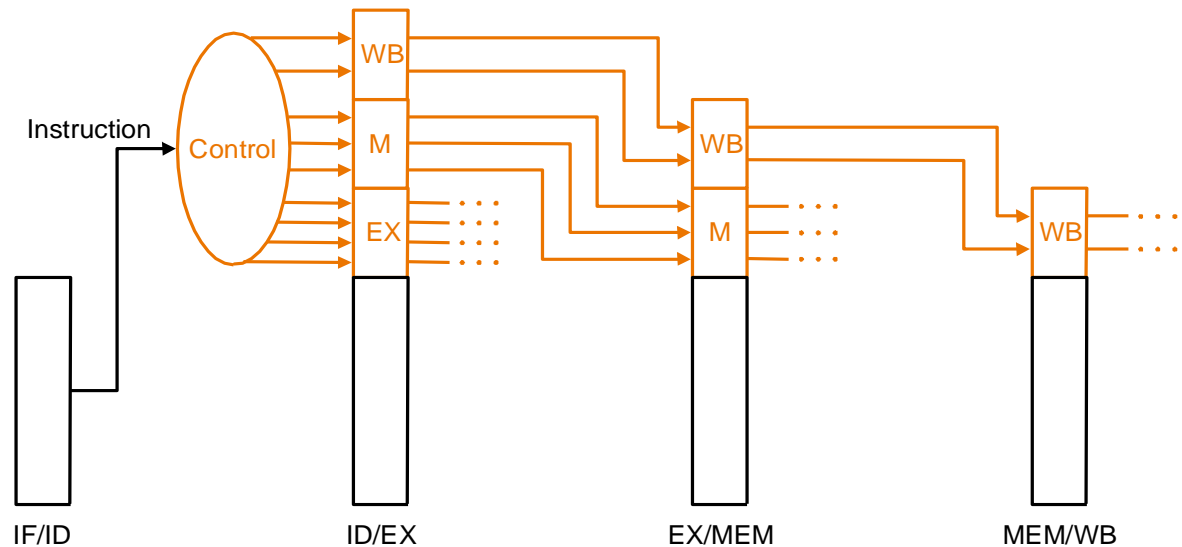
流水线的控制



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

流水线的控制

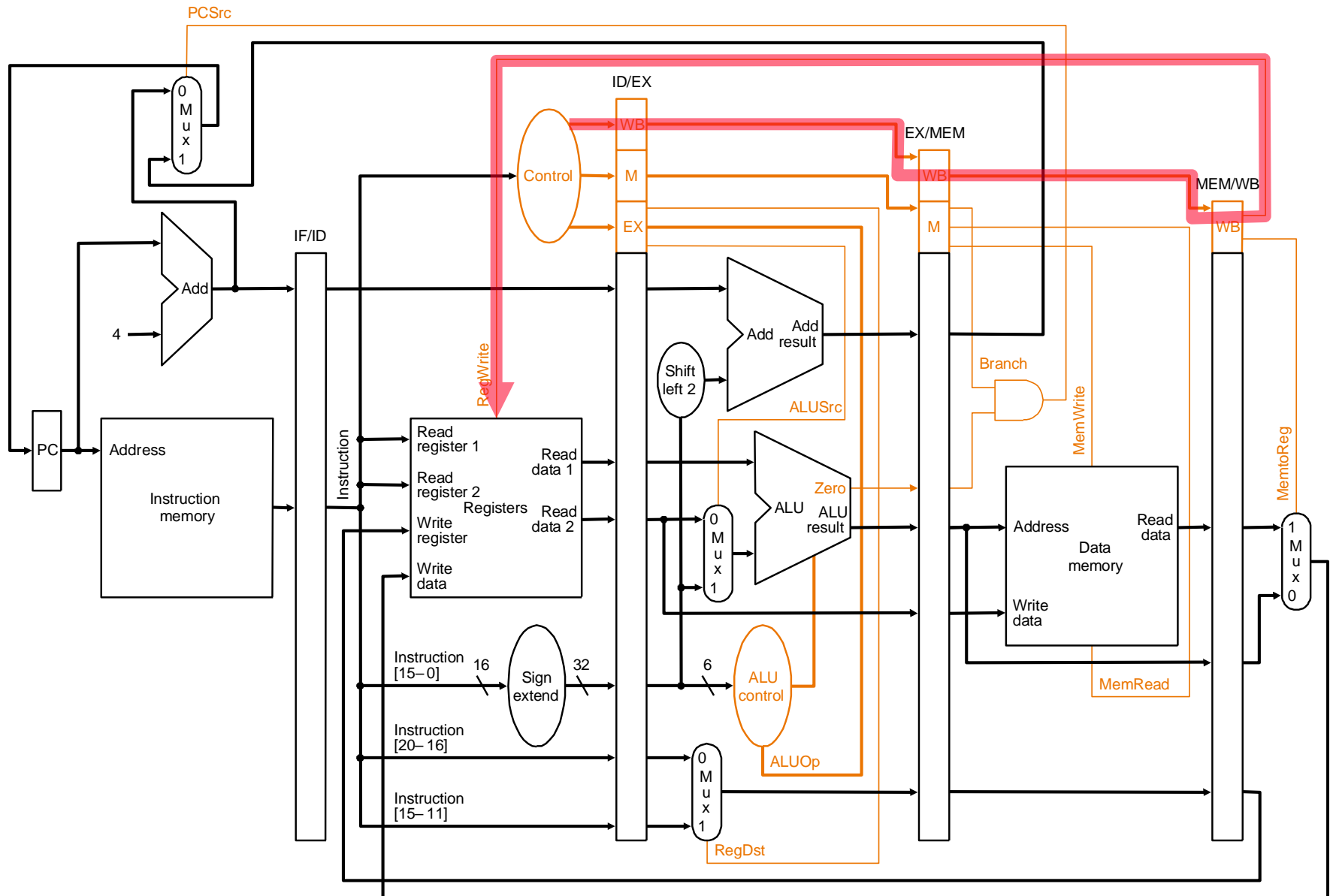
- For a given instruction
 - same control signals as single-cycle, but
 - **control signals required at different cycles, depending on stage**
 - ⇒ Option 1: decode once using the same logic as single-cycle and buffer signals until consumed



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

Which one is better?

流水化的控制信号



Remember: 理想的流水线

- 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)

Instruction Pipeline: 不是理想流水线

- 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)

流水线设计面临的问题

- 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
- Overall: Improving pipeline throughput
 - Minimizing *stalls*

引起流水线停顿的原因

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

相关(dependence)及其类型

- 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 program semantics.

处理资源冲突 (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?


数据相关(Data Dependence)

- Types of data dependences
 - Flow dependence (true 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 registers in the CPU
 - They are **dependence on a name**, not a value
 - We will later see what we can do about them

数据相关示例

Flow dependence


$r_3 \leftarrow r_1 \text{ op } r_2$
 $r_5 \leftarrow r_3 \text{ op } r_4$



Read-after-Write
(RAW)

Anti dependence


$r_3 \leftarrow r_1 \text{ op } r_2$
 $r_1 \leftarrow r_4 \text{ op } r_5$



Write-after-Read
(WAR)

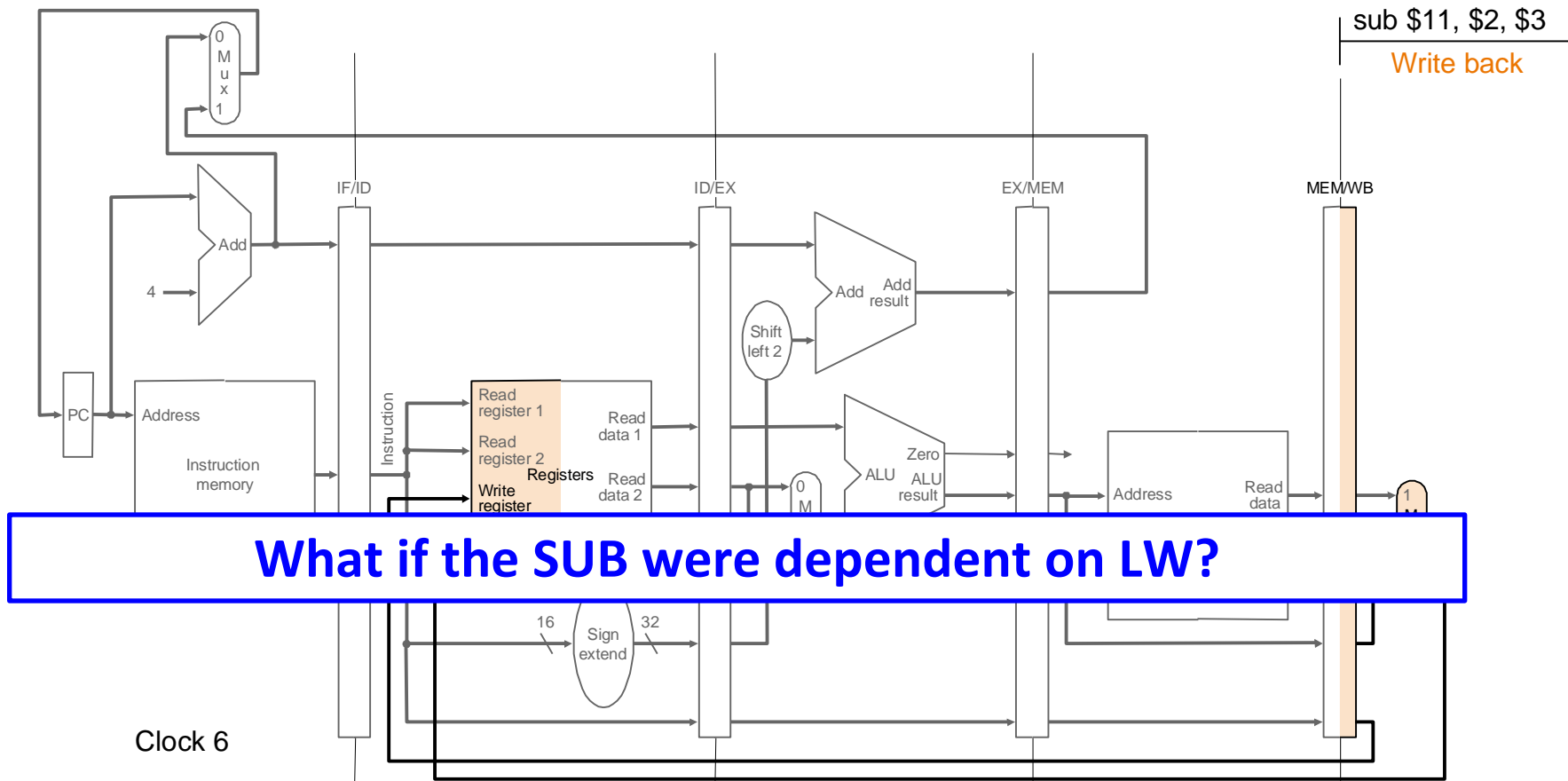
Output-dependence

$r_3 \leftarrow r_1 \text{ op } r_2$
 $r_5 \leftarrow r_3 \text{ op } r_4$
 $r_3 \leftarrow r_6 \text{ op } r_7$



Write-after-Write
(WAW)

指令流水线操作示例



如何处理数据相关

- Anti and output dependences are easier to handle
 - write to the destination in one stage and in program order
- Flow dependences are more interesting
- Four typical 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

Pipeline Interlocking (流水线互锁)

- Detection of dependence between instructions in a pipelined processor to guarantee correct execution
- Software based interlocking
- Hardware based interlocking
- MIPS acronym?

相关检测的方法 (1)

- Scoreboarding (记分牌算法)
 - Each register in register file has a **Valid bit** associated with it
 - An instruction that is writing to the register resets the Valid bit (set to 0)
 - An instruction in Decode stage checks if all its source and destination registers are Valid (value is 1)
 - 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 dependence.

如何在输出相关和反相关不停顿？

- What changes would you make to the scoreboard to enable this?
 - Think about it !

相关检测的方法 (2)

- Combinational dependence check logic
 - Special logic that checks if any instruction in **later stages (pipeline 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 (**think superscalar execution**)

通过硬件检测到相关之后呢……

- 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: ...

Next Topic

Dependence Handling

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
- 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?