CS61C

UC Berkeley
Teaching Professor
Lisa Yan

Great Ideas in Computer Architecture
(a.k.a. Machine Structures)

UC Berkeley
Lecturer
Justin Yokota

Check out public CDSS (Computing, Data Science, and Society) events!
https://data.berkeley.edu/events

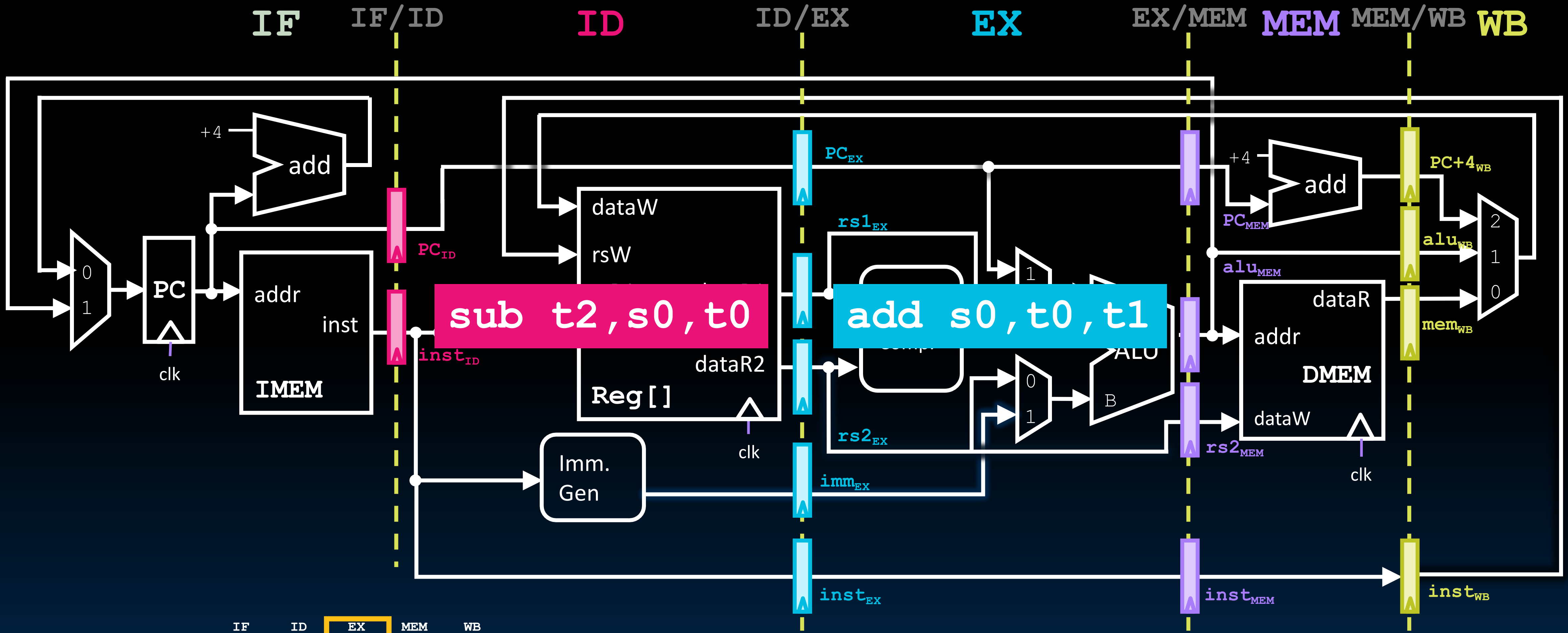
Pipeline II: Hazards

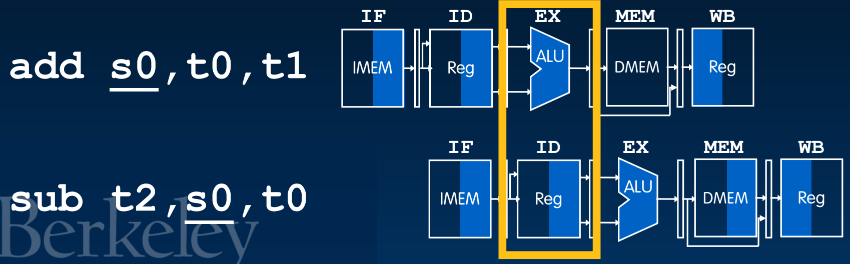
Berkeley
UNIVERSITY OF CALIFORNIA

cs61c.org

Pipelined Datapath has Pipeline Registers

Review





Pipeline registers allow multiple instructions to execute in separate stages.

Yan, Yokota

Three Types of Pipeline Hazards

A hazard is a situation in which a planned instruction cannot execute in the "proper" clock cycle.

✅ Structural hazard:
- Hardware does not support access across multiple instructions in the same cycle.

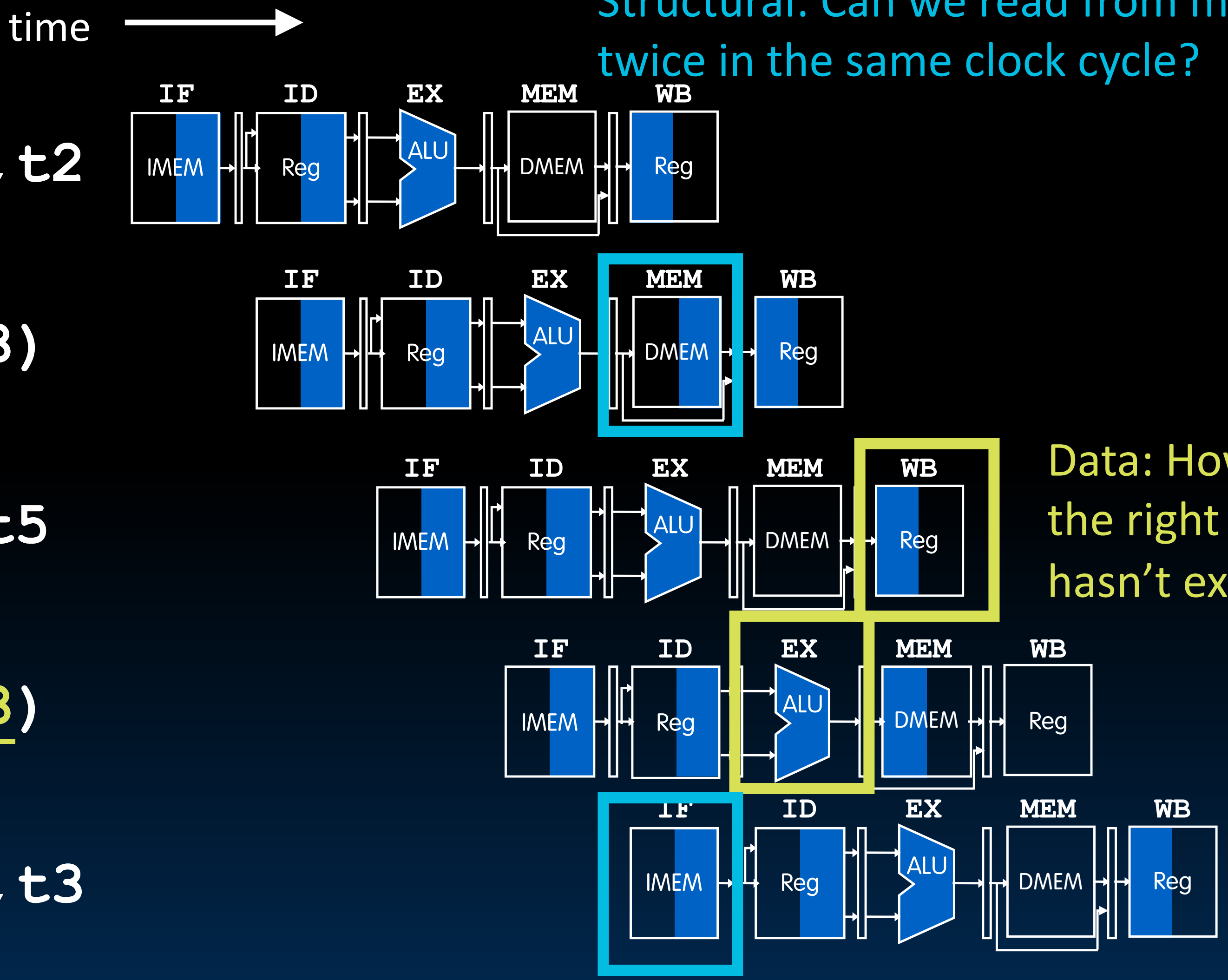
RV32I requirements:
- Reg[] block simultaneously supports two Reads, one Write
- Two simultaneous reads to memory (IMEM, DMEM separate blocks)

2. Data hazard:
   - Instructions have data dependency.
   - Need to wait for previous instruction to complete its data read/write.
3. Control hazard:
   - Flow of execution depends on previous instruction.

Pipeline Hazards Ahead!!!

Structural: Can we read from memory twice in the same clock cycle?

```
add t0,t1,t2
lw t0,8(t3)
or t3,t4,t5
sw t0,4(t3)
sll t6,t0,t3
```



Data: How does **sw**'s **EX** stage get the right value of **t3** if **or**'s **WB** stage hasn't executed yet??

Control: How do branches work???

Agenda

Data Hazards I: Register Access

- Data Hazards I: Register Access
- Data Hazards II: Stalling, Forwarding
- Data Hazards III: Load
- Control Hazards
- [Extra] Superscalar Processors and Calculating CPI

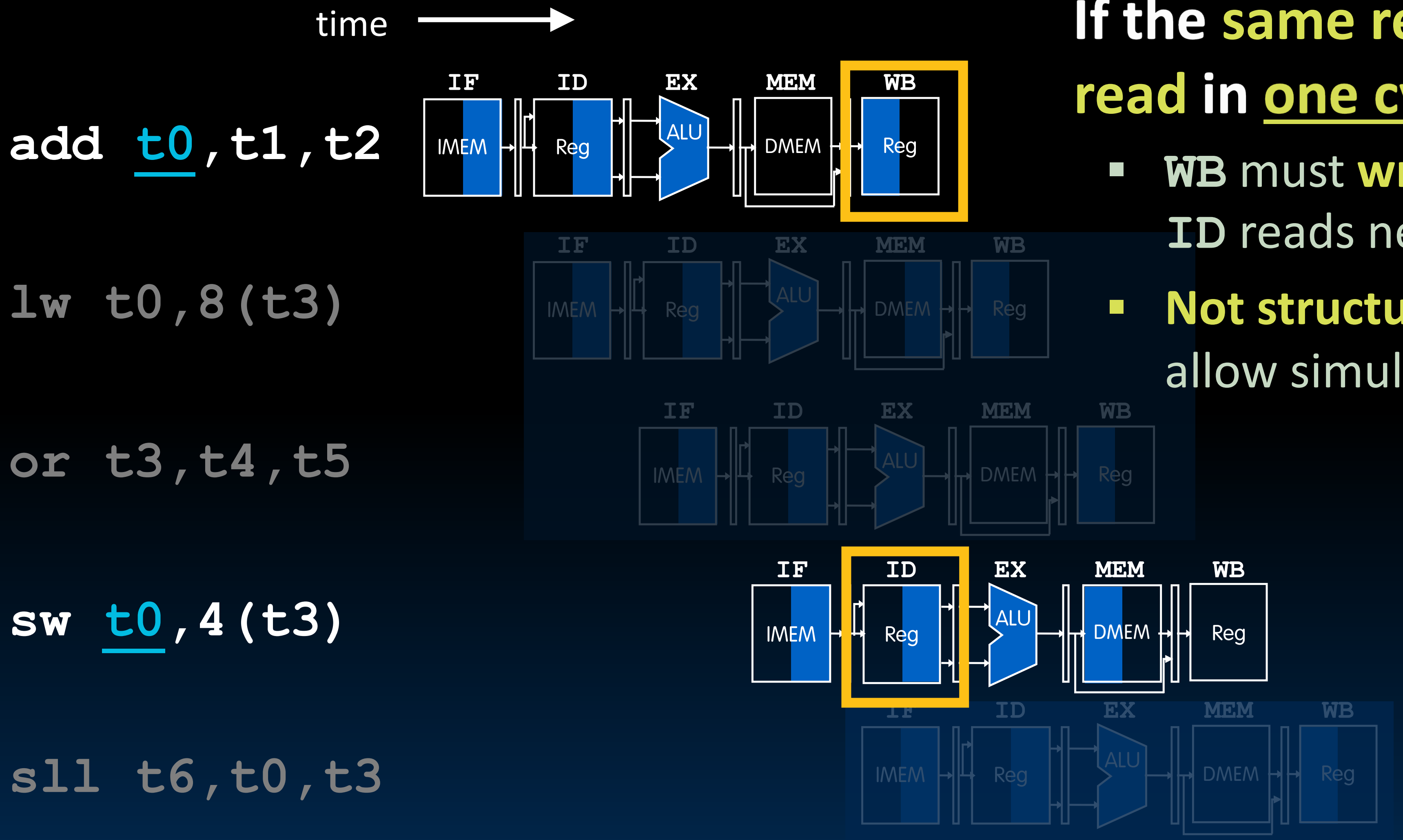
Data Hazards

- Data hazard:
  - Instructions have data dependency.
  - Need to wait for previous instruction to complete its data read/write.
- Occurs when an instruction reads a register before a previous instruction has finished writing to that register.

Three cases to consider:

1. Register access
2. ALU Result
3. Load data hazard (next time)

Data Hazard 1: Register Access



If the same register is written and read in one cycle:

- **WB must write value before** ID reads new value.
- **Not structural hazard!** Separate ports allow simultaneous R/W regardless.

Register Access, Fixed with HW Requirement

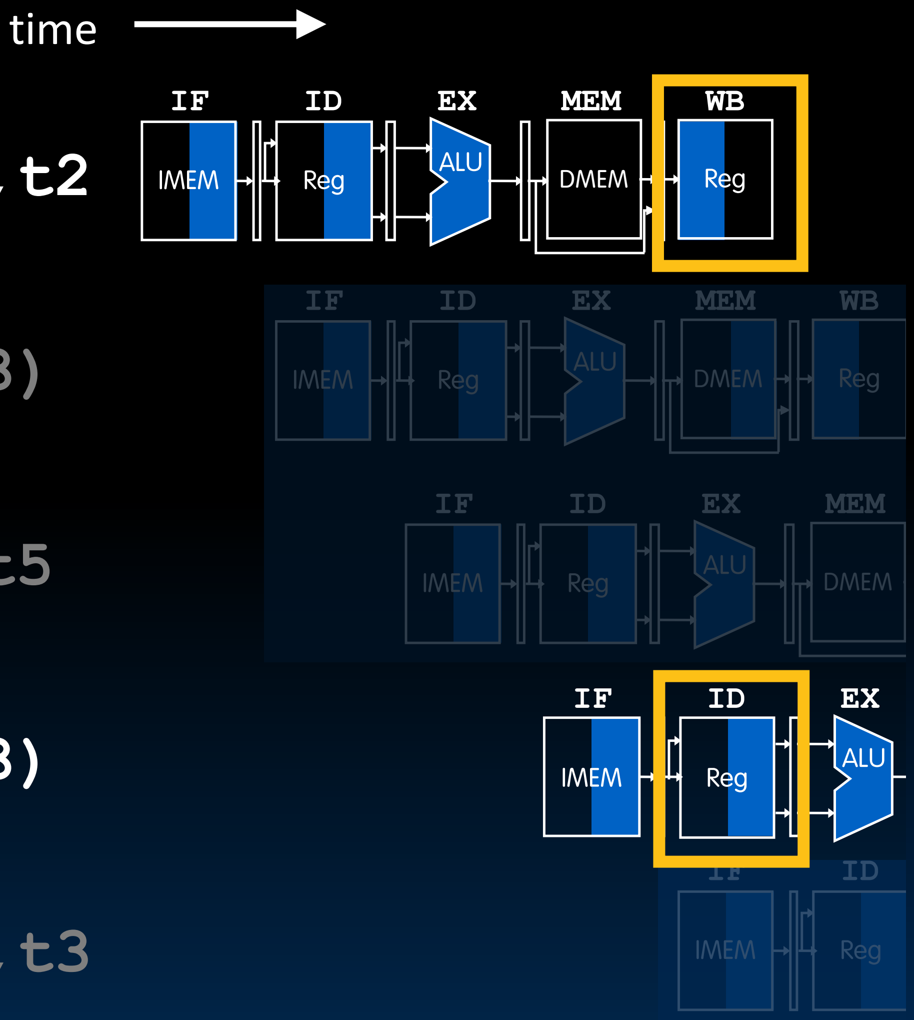
```
add t0,t1,t2

lw  t0,8(t3)

or  t3,t4,t5

sw  t0,4(t3)

sll t6,t0,t3
```



If the same register is written and read **in one cycle:**

- **WB** must write value before **ID** reads new value.
- **Not structural hazard**! Separate ports allow simultaneous R/W regardless.

Solution: RegFile HW should write-then-read in same cycle.

- Exploits high speed of RegFile (100 ps + 100 ps)
- Indicated by shading in diagram

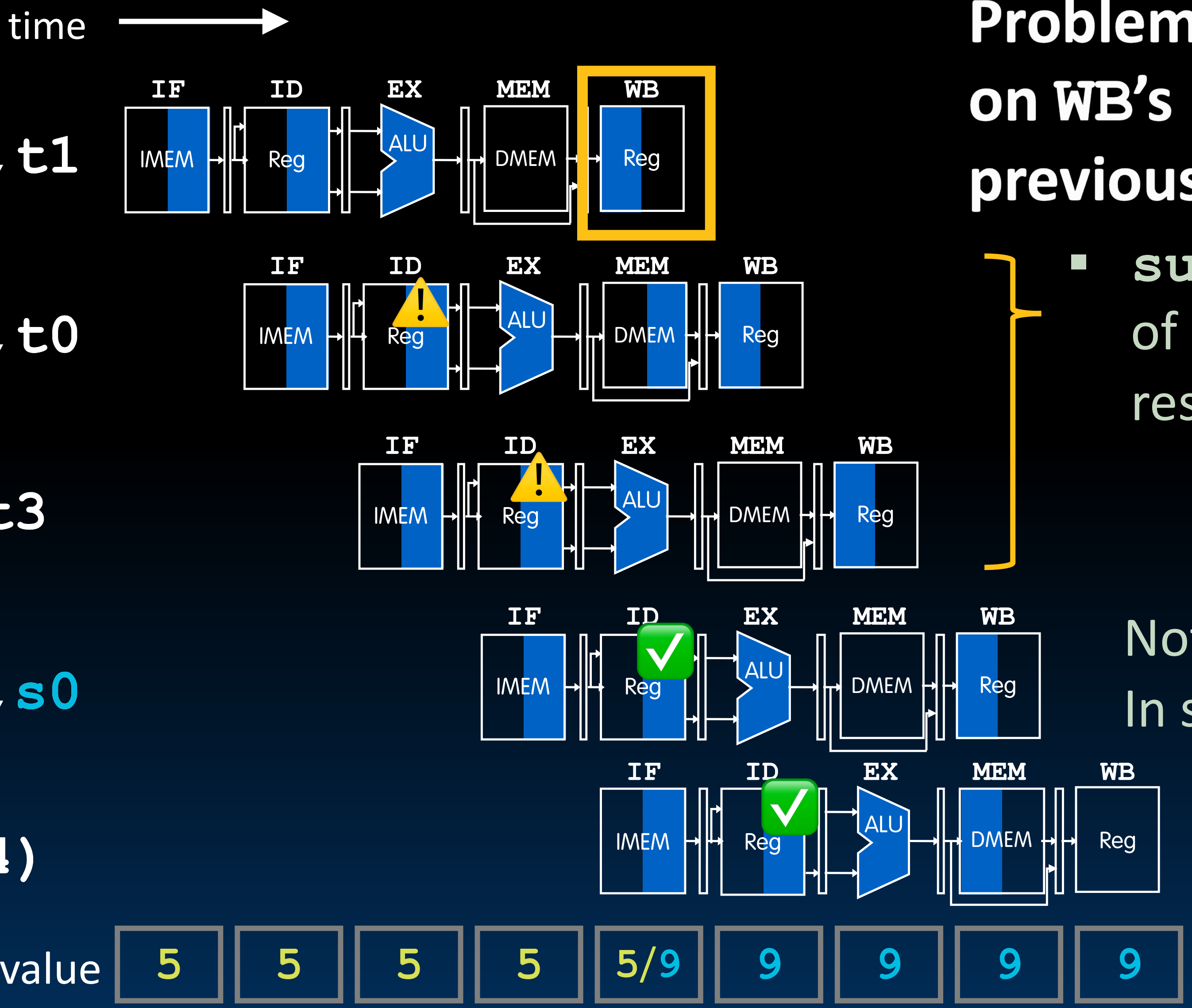
(in some designs, e.g., high frequency might not always be possible to write-then-read in same cycle)

Agenda

Data Hazards II: Stalling, Forwarding

- Data Hazards I: Register Access
- Data Hazards II: Stalling, Forwarding
- Data Hazards III: Load
- Control Hazards
- [Extra] Superscalar Processors and Calculating CPI

Data Hazard 2: ALU Result



s0 value								
5	5	5	5	5/9	9	9	9	9

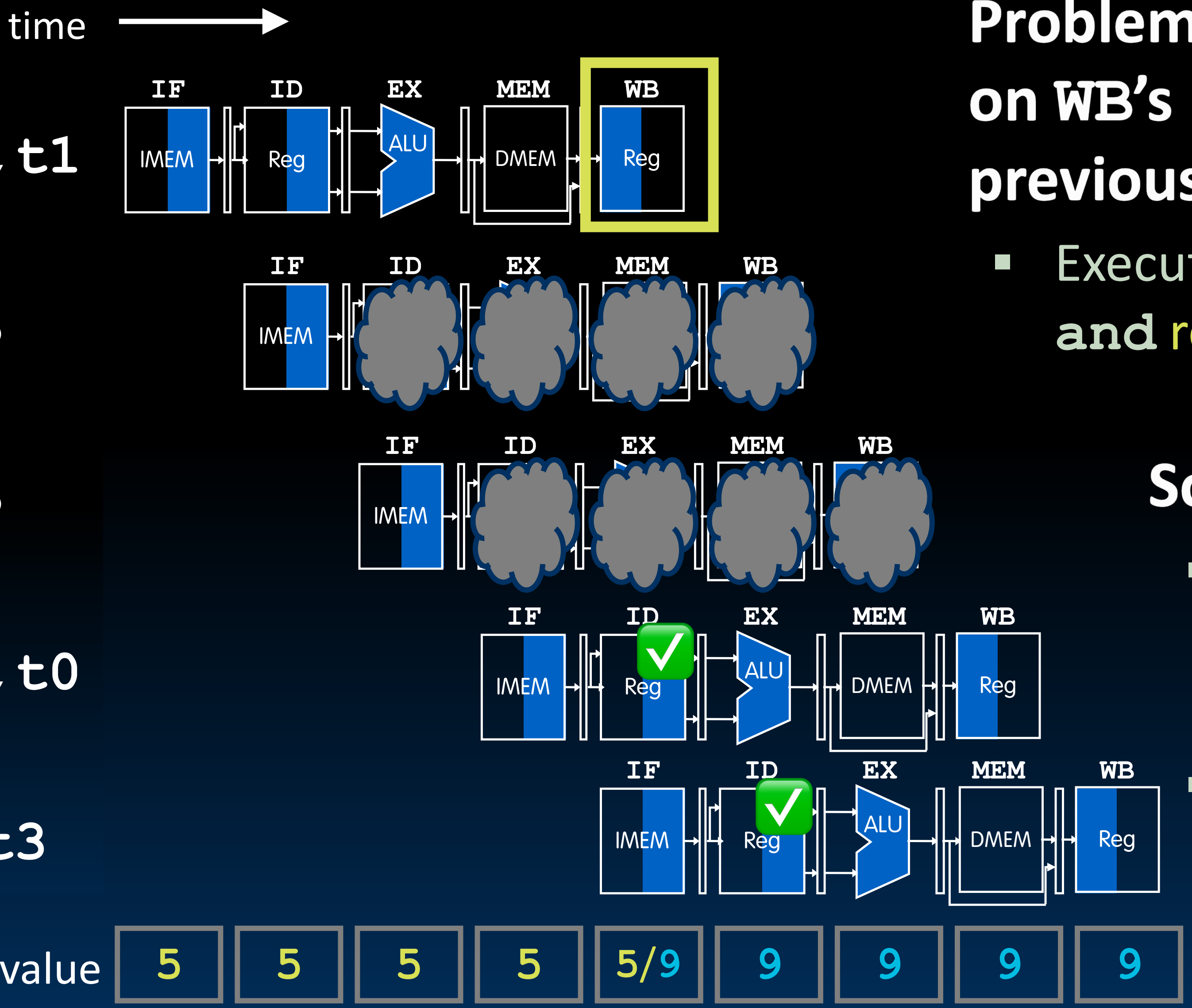
Problem: Instruction depends on WB's RegFile write from previous instruction.

- `sub`, `or`'s `ID` reads old value of `s0` and calculates wrong result.

Note: `xor` gets right value! In same cycle, write s0 then read s0.

Yan, Yokota

Data Hazards, Solution 1: Stalling



Problem: Instruction depends on WB's RegFile write from previous instruction.

- Executing `sub` immediately after `and` reads old value of `s0`.

Solution 1: Stall pipeline:

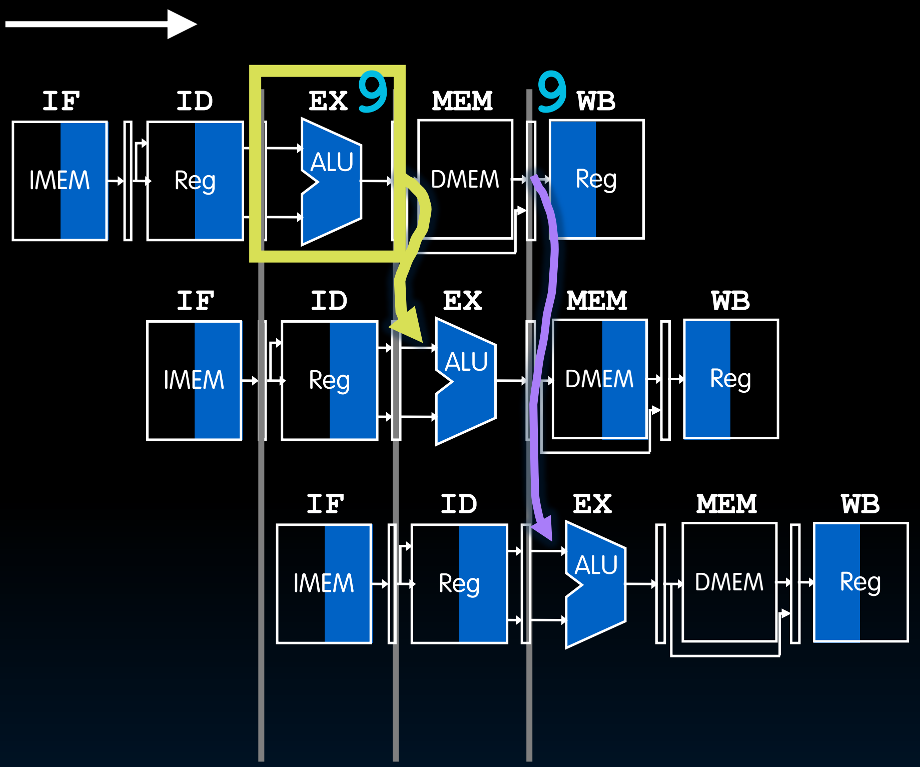
- Requires extra pipeline state to prevent register writes on stalled stages.
- The correct instruction is stalled for two clock cycles.

Yan, Yokota

Solution 2: Forwarding

time →

```
add t0 → add s0,t0,t1
```



`add s0,t0,t1`

`sub t2,s0,t0`

`or t6,s0,t3`

s0 value	5	5	5	5	5/9	9	9	9	9

Forwarding, aka bypassing, uses the result when it is computed.

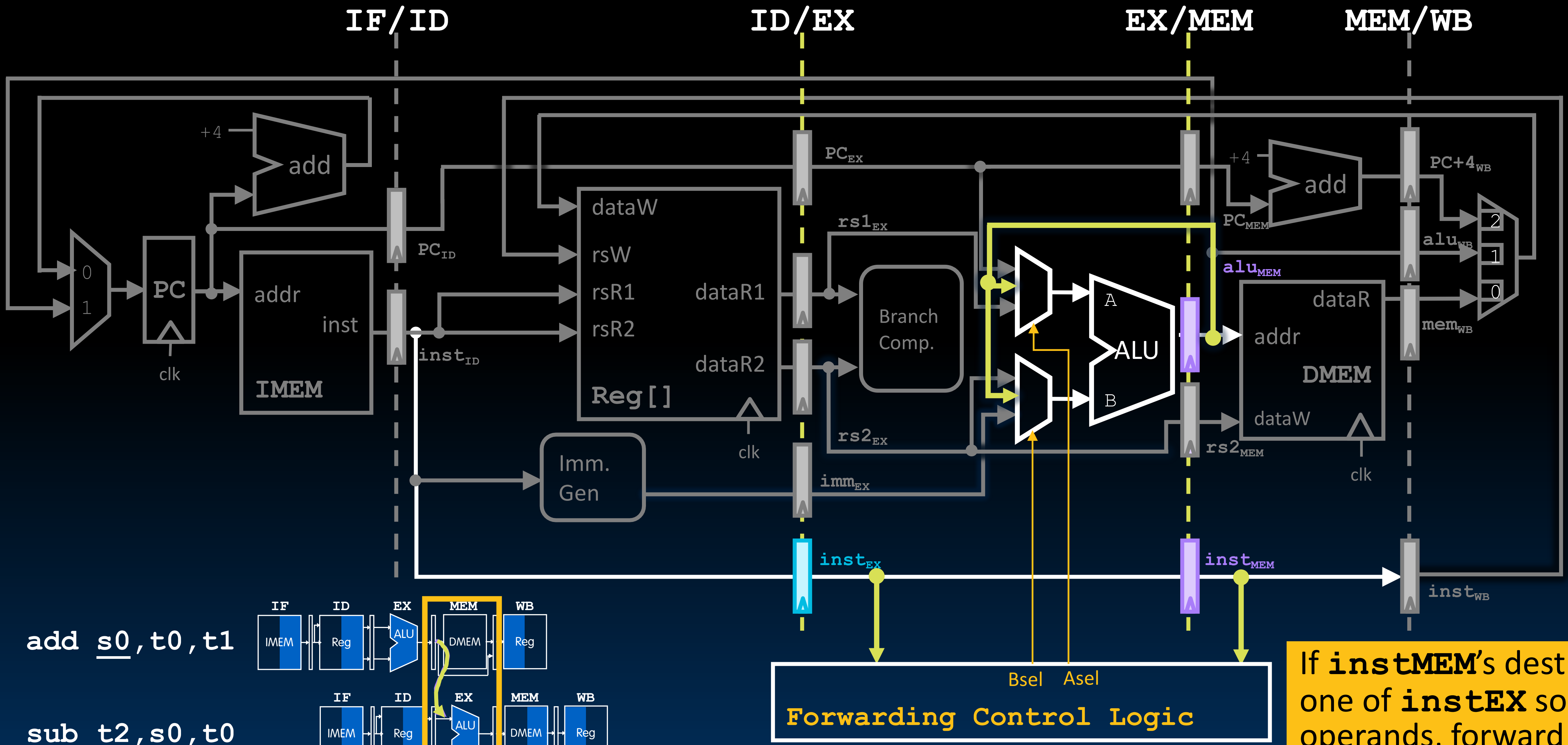
- Don't wait for value to be stored into RegFile.
- Instead, grab operand from the pipeline stage.

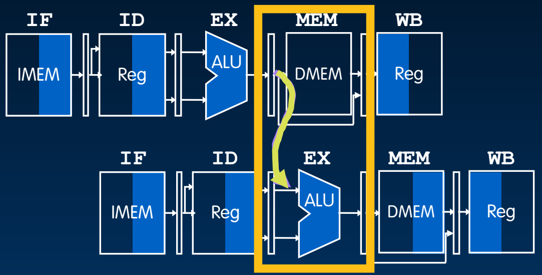
Implementation:

- Make extra connections in the datapath.
- Also add forwarding control logic.

Forwarding EX Results

(MEM Forwarding is similar but omitted for simplicity.)





add s0,t0,t1

sub t2,s0,t0

If **instMEM**'s dest. is one of **instEX** source operands, forward it.

Agenda

Data Hazards III: Load

- Data Hazards I: Register Access
- Data Hazards II: Stalling, Forwarding
- Data Hazards III: Load
- Control Hazards
- [Extra] Superscalar Processors and Calculating CPI

Which Data Hazards Can Forwarding Fix?

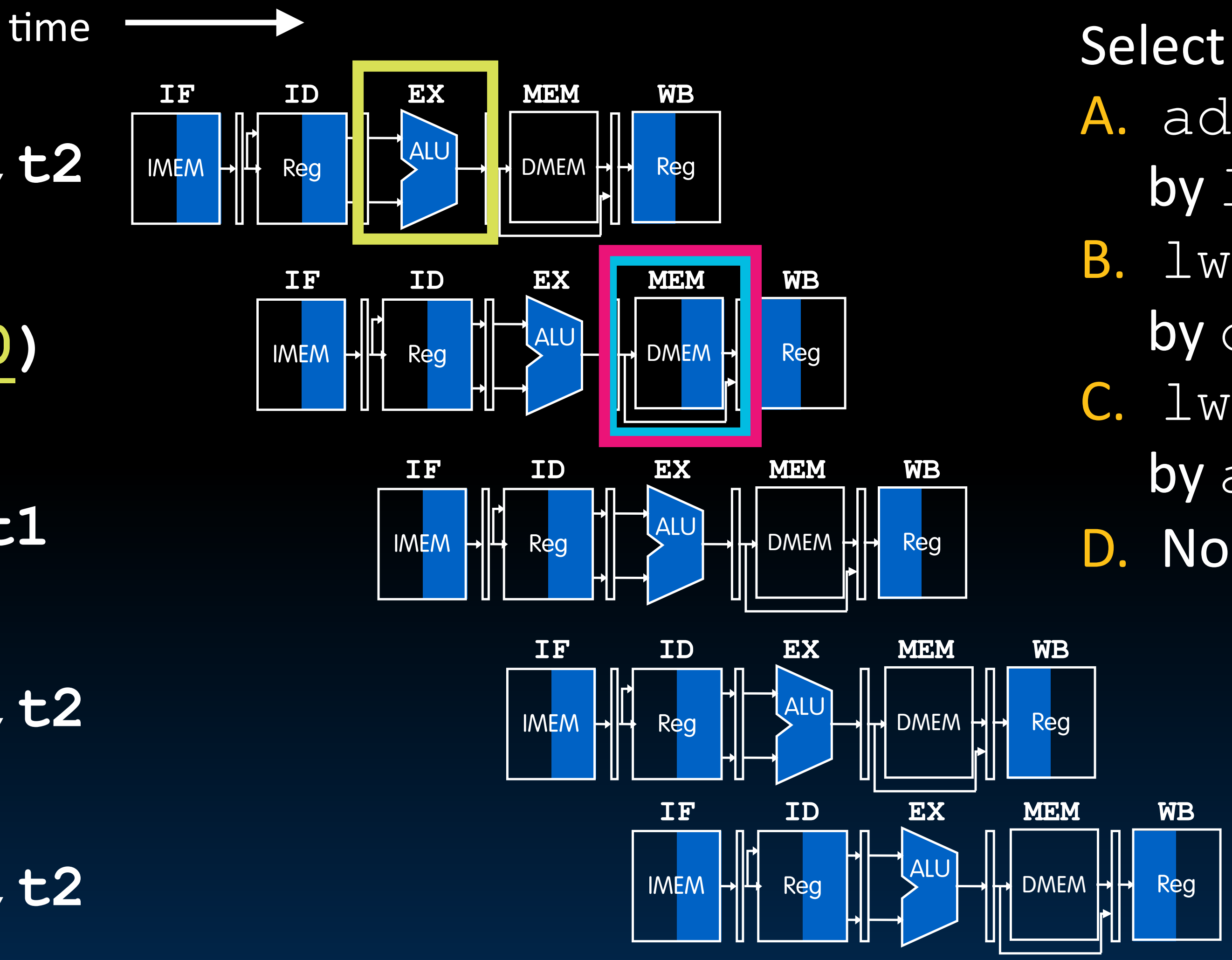
```
add s0,t1,t2

lw s1,8(s0)

or t3,s1,t1

and t4,s1,t2

sll t0,t1,t2
```

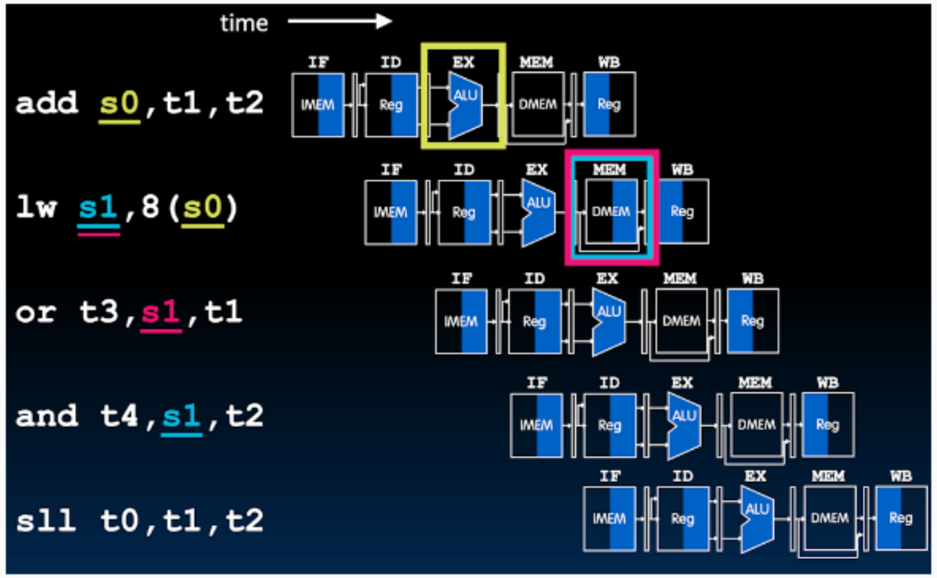


Select all:

A. `add EX` stage output needed by `lw MEM` input
B. `lw MEM` stage output needed by `or EX` input
C. `lw MEM` stage output needed by `and EX` input
D. None of the above

Which Data Hazards Can Forwarding Fix?

0



A. add EX stage output needed by lw MEM input

0

B. lw MEM stage output needed by or EX input

0

C. lw MEM stage output needed by and EX input

0

D. None of the above

0

Forwarding Cannot Fix All Data Hazards (1/2)

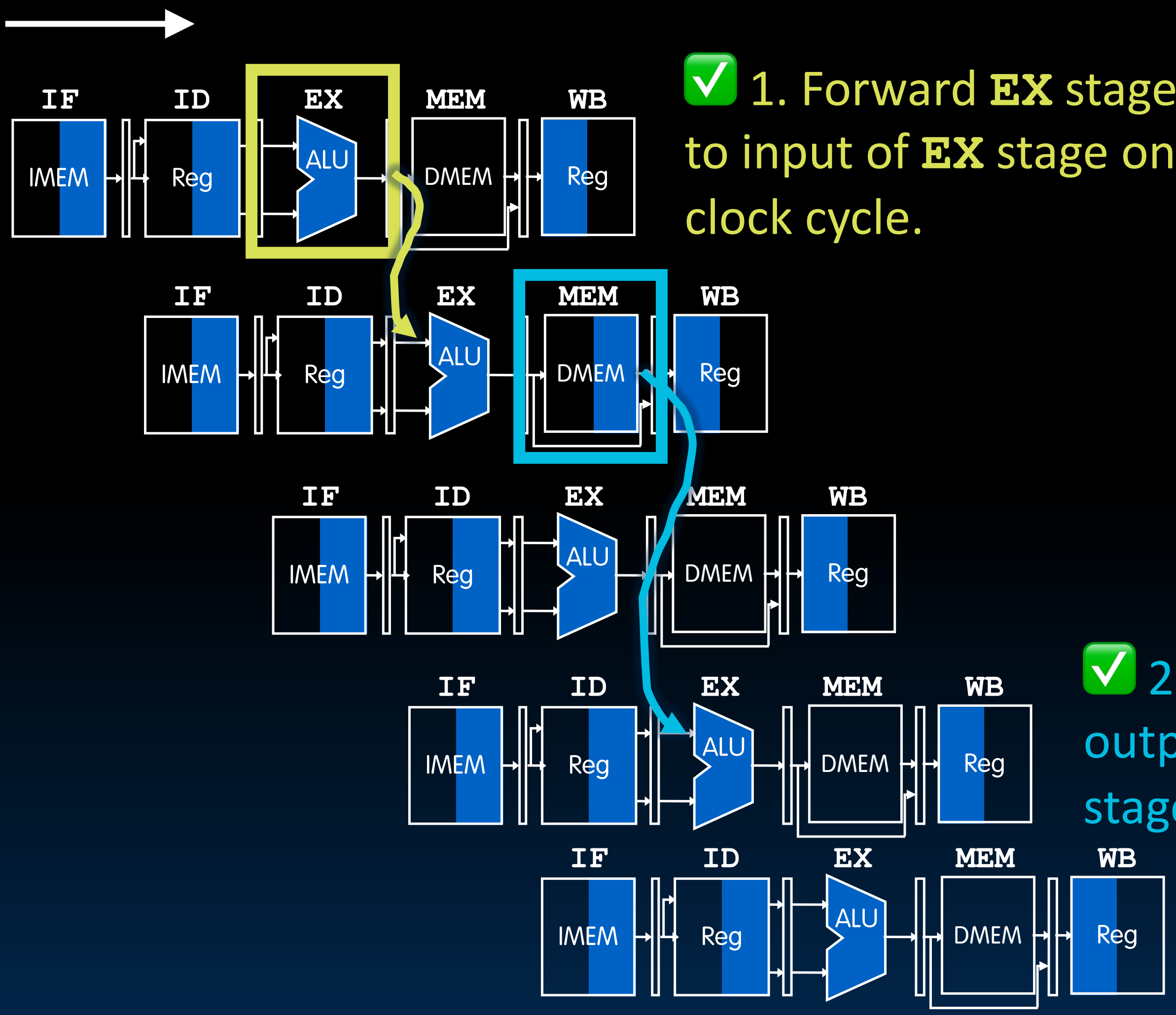
```
add s0,t1,t2

lw s1,8(s0)

or t3,s1,t1

and t4,s1,t2

sll t0,t1,t2
```



✅ 1. Forward **EX** stage output to input of **EX** stage on next clock cycle.

✅ 2. Forward **MEM** stage output to input of **EX** stage on next clock cycle.

Forwarding Cannot Fix All Data Hazards (2/2)

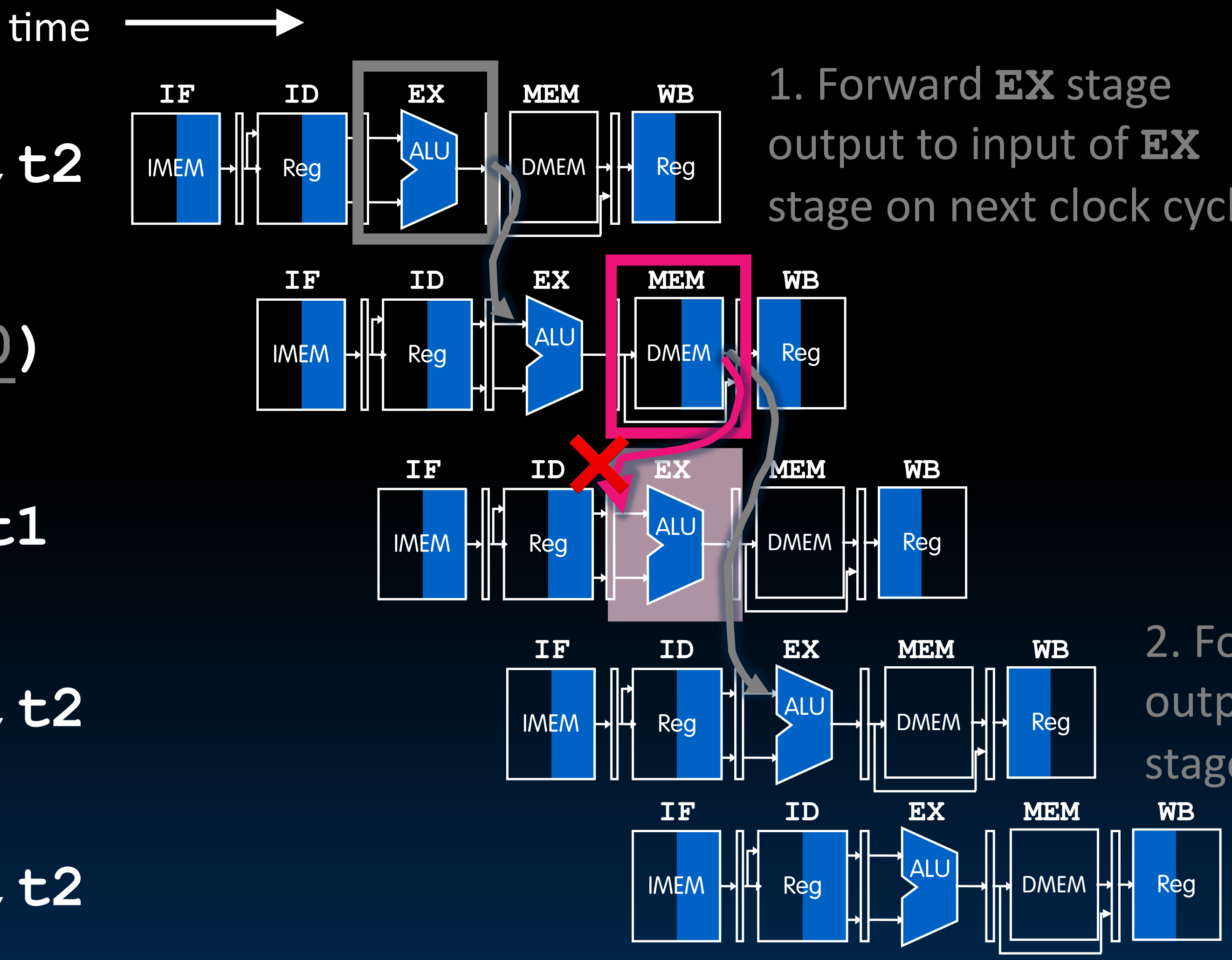
```
add s0,t1,t2

lw s1,8(s0)

or t3,s1,t1

and t4,s1,t2

sll t0,t1,t2
```



1. Forward **EX** stage output to input of **EX** stage on next clock cycle.

⚠ 3. **MEM** stage (**lw**)'s output needed as **EX** stage (**or**)'s input in the same clock cycle.

2. Forward **MEM** stage output to input of **EX** stage on next clock cycle.

Loads can result in an unavoidable pipeline stall.

Yan, Yokota

Data Hazard 3: Loads

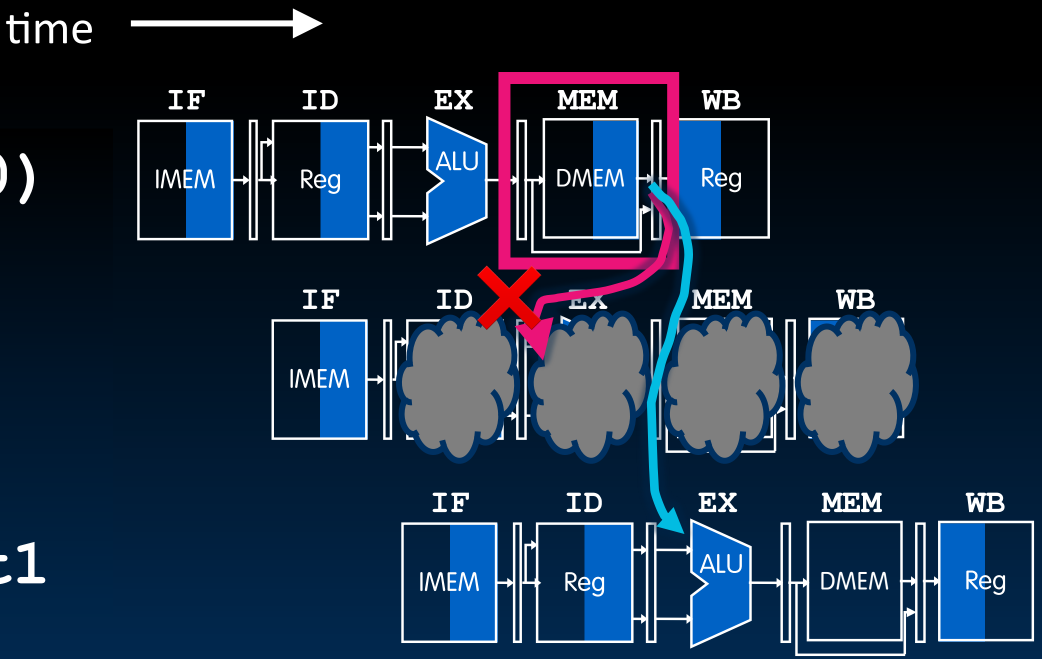
- The instruction slot after a load is called the load delay slot.
- If this instruction uses the result of load:
  - The hardware must stall for one cycle (plus *forwarding*).
  - This results in performance loss!

```
lw s1,8(s0)
```

Load delay slot:

```
or → nop
```

```
or t3,s1,t1
```



⚠ MEM stage (`lw`)'s output needed as EX stage (`or`)'s input in the same clock cycle.

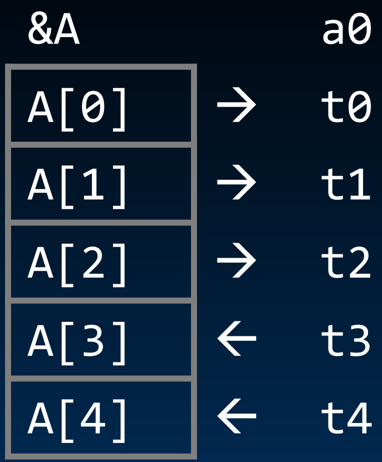
Forwarding sends data to the next clock cycle.
Cannot go backwards in time!

Yan, Yokota

Solution (1/2): Code Scheduling

C Code

```
A[3] = A[0] + A[1];
A[4] = A[0] + A[2];
```



⚠️ Simple compilation (9 cycles for 7 instructions)

```
lw   t0, 0(a0)
lw   t1, 4(a0)
add  t2, t0, t1
sw   t2, 12(a0)
lw   t3, 8(a0)
add  t4, t0, t3
sw   t4, 16(a0)
```

Stall & forward! (+1 cycle)

(+1 cycle)

Yan, Yokota

Solution (2/2): Code Scheduling

- Idea: Fix this hazard at the code compilation stage.
  - In the delay slot, put an instruction unrelated to the load result.
  - → No performance loss!

C Code

```
A[3] = A[0] + A[1];
A[4] = A[0] + A[2];
```

⚠️ Simple compilation (9 cycles for 7 instructions)

```
lw   t0, 0(a0)
lw   t1, 4(a0)
add  t2, t0, t1     Stall & forward! (+1 cycle)
sw   t2, 12(a0)
lw   t3, 8(a0)
add  t4, t0, t3     (+1 cycle)
sw   t4, 16(a0)
```

✅ Alternative (7 cycles):

```
lw   t0, 0(a0)
lw   t1, 4(a0)
lw   t3, 8(a0)
add  t2, t0, t1     Forward! (+0 cycle)
sw   t2, 12(a0)
add  t4, t0, t3
sw   t4, 16(a0)
```

Code scheduling: With knowledge of the underlying CPU pipeline, the compiler reorders code to improve performance.

Yan, Yokota

Agenda

Control Hazards

- Data Hazards I: Register Access
- Data Hazards II: Stalling, Forwarding
- Data Hazards III: Load
- Control Hazards
- [Extra] Superscalar Processors and Calculating CPI

Three Types of Pipeline Hazards

A hazard is a situation in which a planned instruction cannot execute in the "proper" clock cycle.

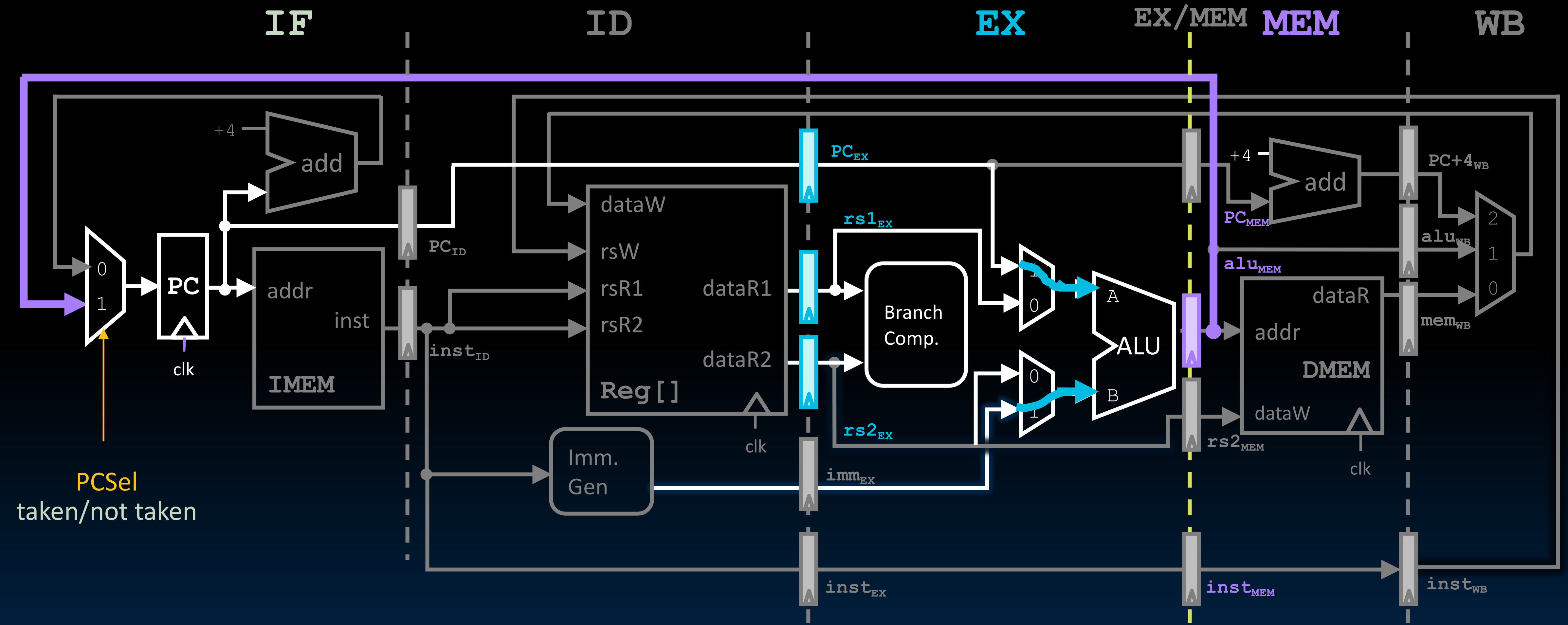
✅ Structural hazard:
- Hardware does not support access across multiple instructions in the same cycle.

✅ Data hazard:
- Instructions have data dependency.
- Need to wait for previous instruction to complete its data read/write.

3. Control hazard:
   - Flow of execution depends on previous instruction.

Branch Results Computed During MEM

Review



In **MEM** stage: **EX/MEM** pipeline reg. feeds **IF** stage MUX. PCSel control is set.

On the next clock cycle in the **IF** stage, PC updates, and the correct instruction is fetched.

Control Hazard: Conditional Branches

Control hazards occur when the instruction fetched may not be the one needed. For example, if the `beq` branch is **taken**:

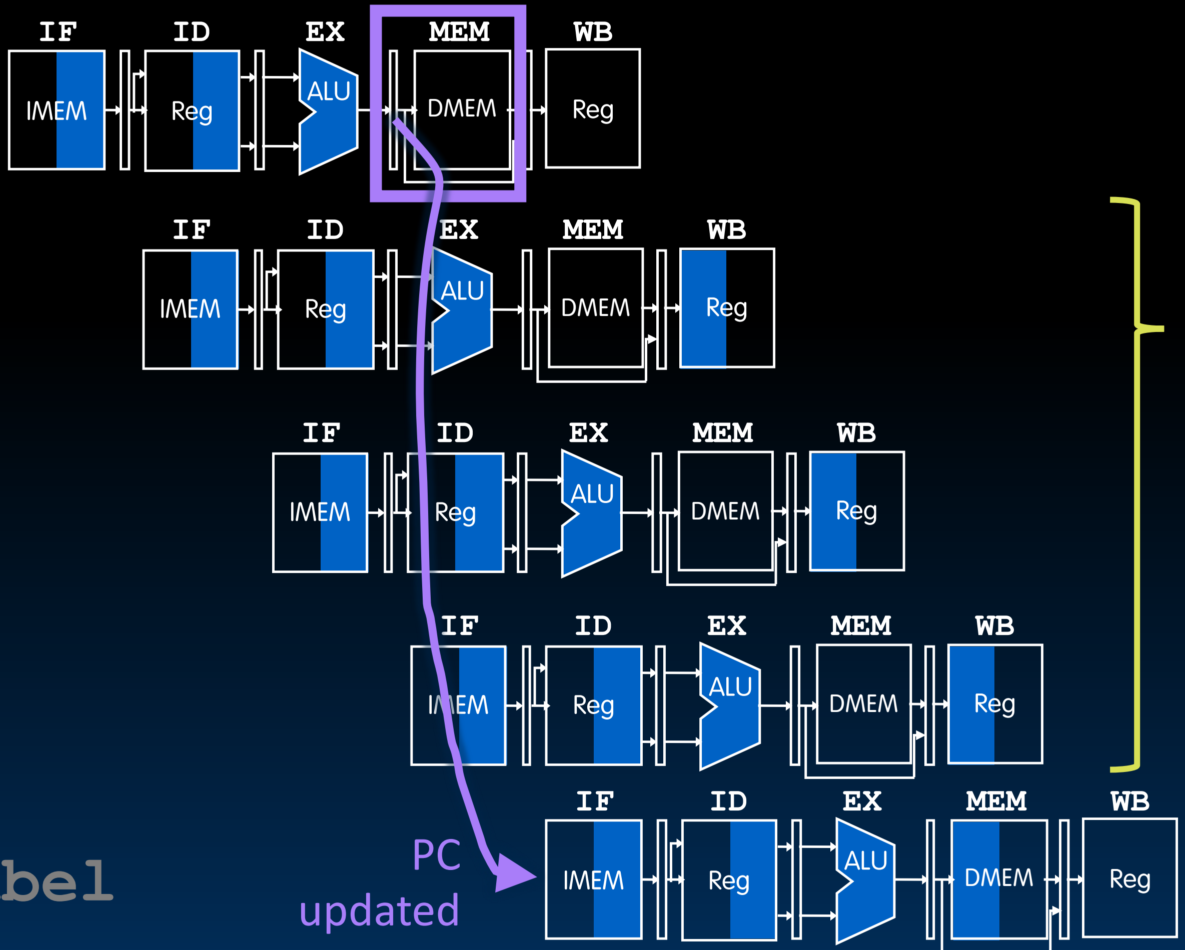
```
0x40 beq t0,t1,Label

0x44 sub t2,s0,t0

0x48 or t6,s0,t3

0x4c xor t5,t1,s0

0x70 sw s0,8(t3) # Label
```



Instruction execution starts before branch outcome is known!

Correct instruction starts executing

Yan, Yokota

Kill Instructions after Branch (If Taken)

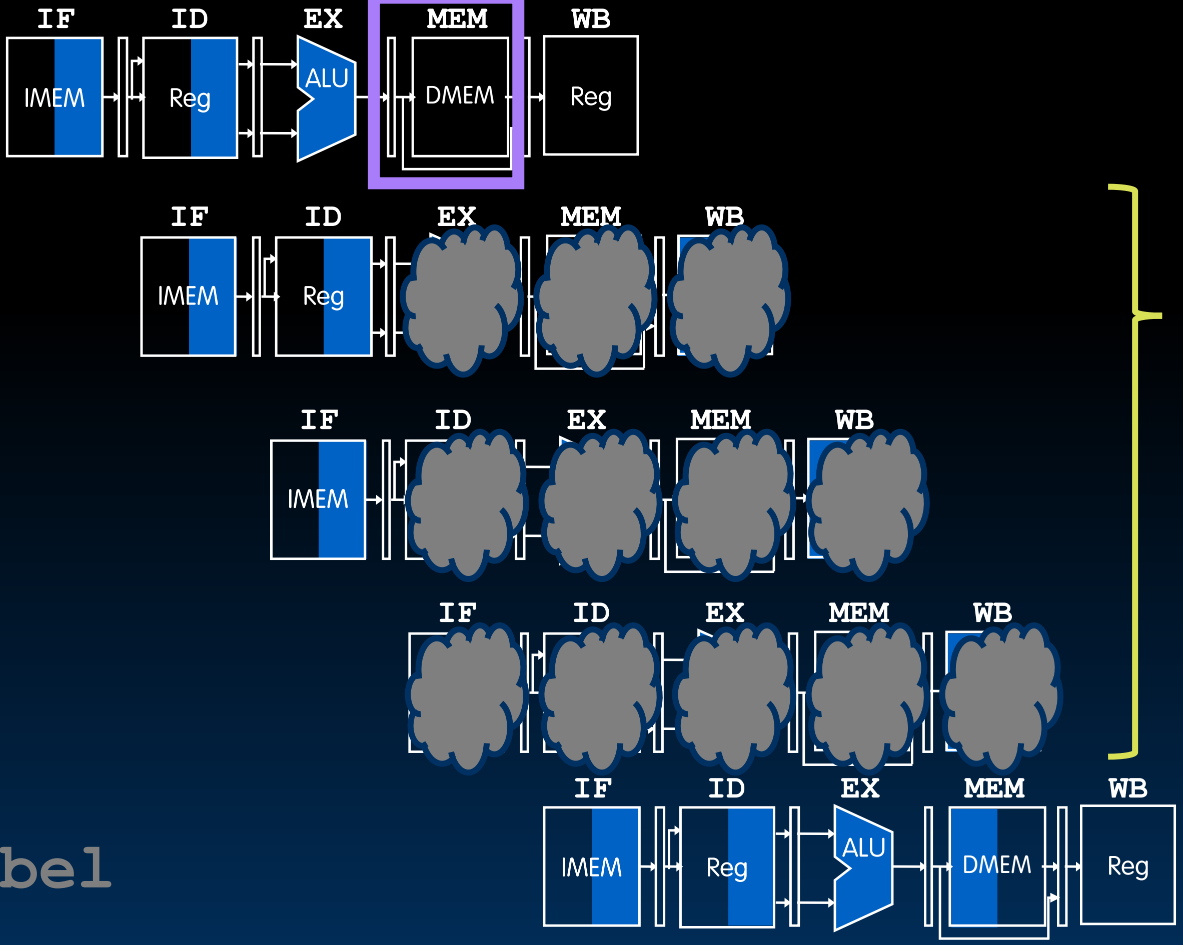
```
0x40 beq t0,t1,Label

0x44 sub t2,s0,t0

0x48 or t6,s0,t3

0x4c xor t5,t1,s0

0x70 sw s0,8(t3) # Label
```



Flush pipeline by converting 3 incorrect instructions to nops.

PC updated, correct instruction loaded

Branch Prediction to Reduce Penalties

- Every **taken branch** in the simple RV32I pipeline costs 3 clock cycles.
  - Note if branch is not taken, then pipeline is not stalled; the correct instructions are correctly fetched sequentially after the branch instruction.
  - (See textbook for an *out of scope* hardware cost improvement.)
- We can improve the CPU performance on average through branch prediction.
  - Early in the pipeline, guess which way branches will go.
  - Flush pipeline if branch prediction was incorrect.

Naïve Predictor: *Don't Take Branch*

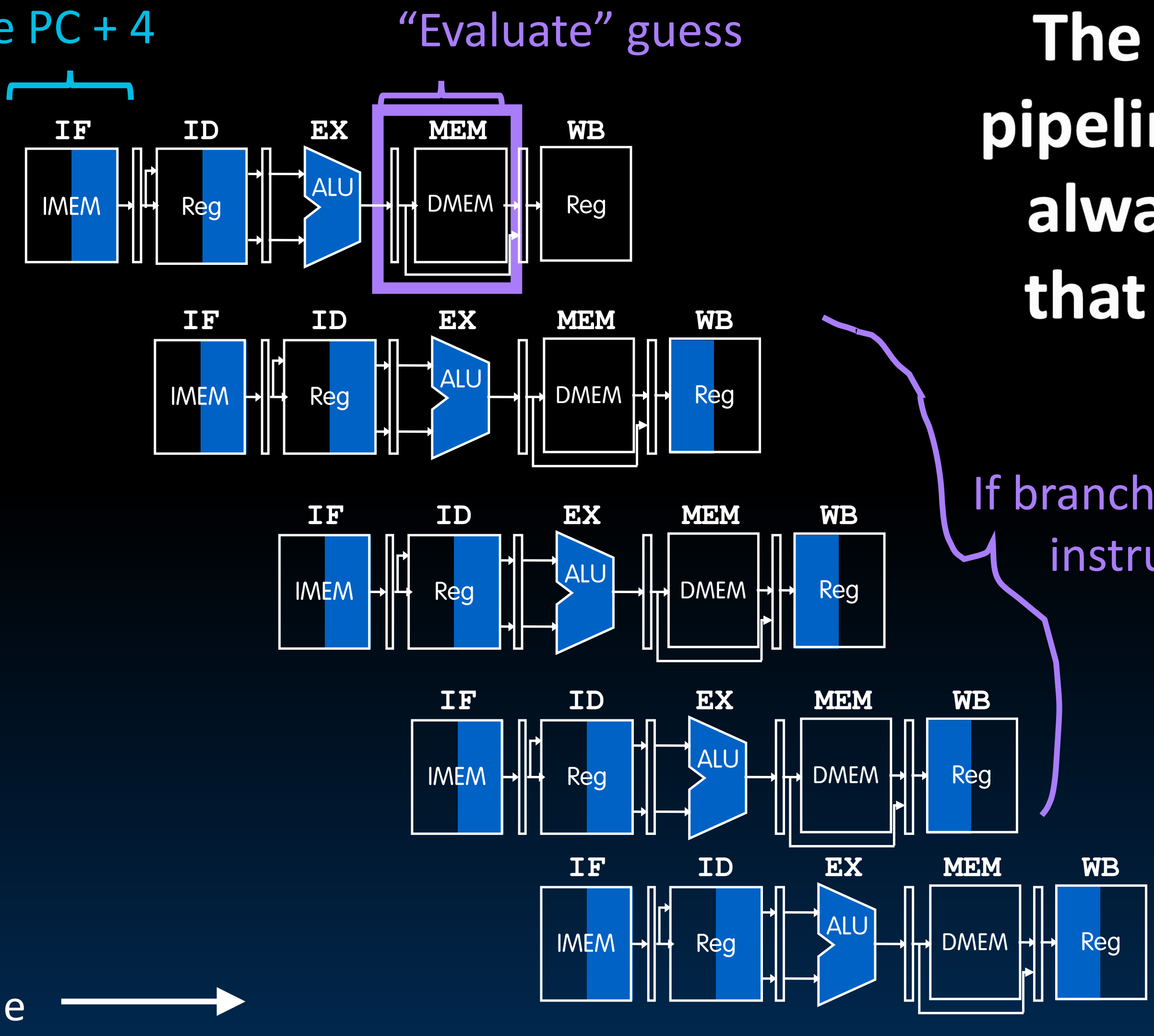
```
0x40 beq t0,t1,Label

0x44 sub t2,s0,t0

0x48 or t6,s0,t3

0x4c xor t5,t1,s0

0x50 add t2,s0,s0
```



The simple RV32I pipeline effectively always "predicts" that branches are not taken.

If branch not taken, correct instructions are already executed.

Penalty for incorrect prediction is still 3 clock cycles! Branch prediction tries to improve **average performance**.

Agenda

[Extra] Superscalar Processors and Calculating CPI

- Data Hazards I: Register Access
- Data Hazards II: Stalling, Forwarding
- Data Hazards III: Load
- Control Hazards
- [Extra] Superscalar Processors and Calculating CPI

Pipelining and ISA Design

- The RISC-V ISA is designed for pipelining:
  - All instructions are 32 bits wide.
    - Easy to fetch and decode, each in one clock cycle.
    - Contrast with CISC (Complex) x86: 1- to 15-byte-wide instructions!
  - A small set of standard instruction formats
    - Can decode/read registers in one stage.
  - Load/store addressing conceptually:
    - Calculate address in 3rd stage (with ALU); and
    - Access memory in 4th stage.
  - Memory operands are all aligned
    - Memory access takes only one cycle.

The 5-stage pipeline we have studied is commonplace in many devices: cars, appliances, etc.

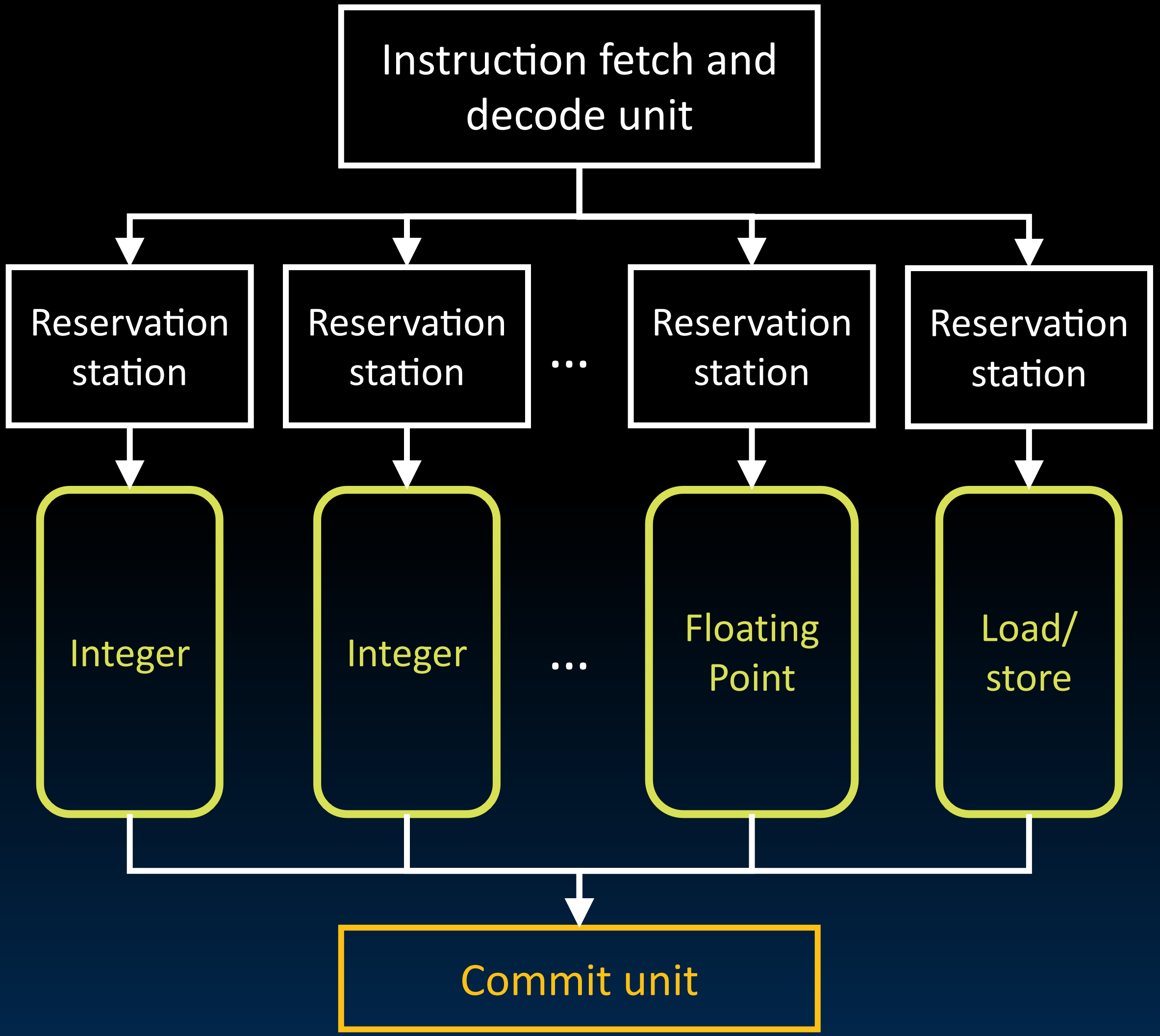
Further Increasing Processor Performance?

1. Increase clock rate.
   - Limited by technology and power dissipation
2. Increase pipeline depth.
   - “Overlap” instruction execution through deeper pipeline, e.g., 10 or 15 stages.
     - Less work per stage → shorter clock cycle/lower power
     - But more potential for all three types of hazards! (more stalling → CPI > 1)
3. Design a “superscalar” processor.
   - Desktops, laptops, cell phones, etc. often have a few of these, combined with simpler 5-stage pipeline processors.

Yan, Yokota

Superscalar Processors

- Multiple-issue: Start multiple instructions per clock cycle.
  - Multiple execution units execute instructions in parallel.
    - Each execution unit has its own pipeline.
    - *CPI < 1*: multiple instructions completed per clock cycle.
- Dynamic "out-of-order" execution:
  - Reorder instructions dynamically in HW to reduce impact of hazards.



Take CS152 for more!

Computing Cycles Per Instruction (CPI)

- ARM Cortex-A53 Core: 2 GHz, dual-issue processor:
  - 4 BIPS (billion instructions per second), *Peak CPI* = 0.5.
  - However, instruction/pipeline dependencies reduce performance.
- In practice, measure CPI on various benchmarks:
  - Known benchmark programs from a variety of application domains:
    - Data compression, code compilation, video decoding, network simulation, etc.

From the "Iron Law" of Processor Performance:

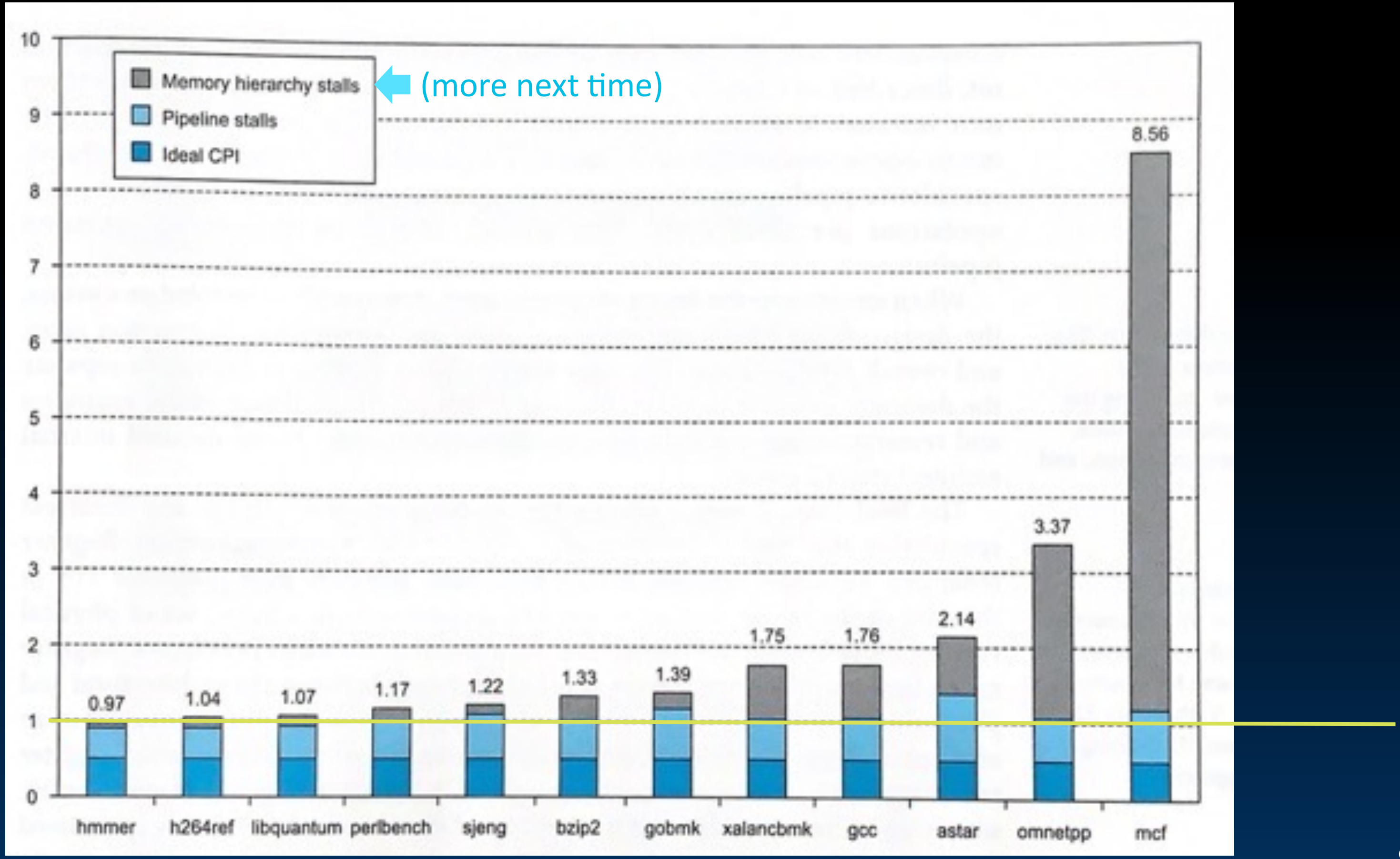
$$\underbrace{\frac{time}{program}}_{\text{(can measure)}} = \underbrace{\frac{instructions}{program}}_{\text{(can count)}} \times \underbrace{\frac{cycles}{instruction}}_{\text{CPI}} \times \underbrace{\frac{time}{cycle}}_{\text{1/clock rate}}$$

We compute CPI on different benchmarks:

$$\boxed{\text{CPI} = \frac{time}{program} \div \left(\frac{instructions}{program} \times \frac{time}{cycle}\right)}$$

ARM A53 Benchmark

- ARM Cortex-A53 Core: 2 GHz, dual-issue processor:
  - 4 BIPS (billion instructions per second), *Peak CPI* = 0.5.
  - However, instruction/pipeline dependencies reduce performance.



If CPI < 1, can also measure in IPC (Instructions Per Cycle).

P&H 2nd edition, Section 4.12 (Fig 4.76, p.357)

Yan, Yokota