



UC Berkeley
Teaching Professor
Dan Garcia

CS61C

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

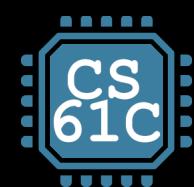


UC Berkeley
Teaching Professor
Lisa Yan

Post-Midterm Ed Post:

- Do not talk about the exam
- Reach out to cs61c@berkeley.edu with any exam-related concerns
- Student Support appointment meetings (non-exam related) are open

Pipeline III: More Hazards, Superscalar Processors



Talk announcement! Prof. Jae W. Lee

SLICE Lab Meeting, Wednesday 10/19/22, 1:05 PM on Zoom

Specialized Architectures for Machine Learning and Data Analytics



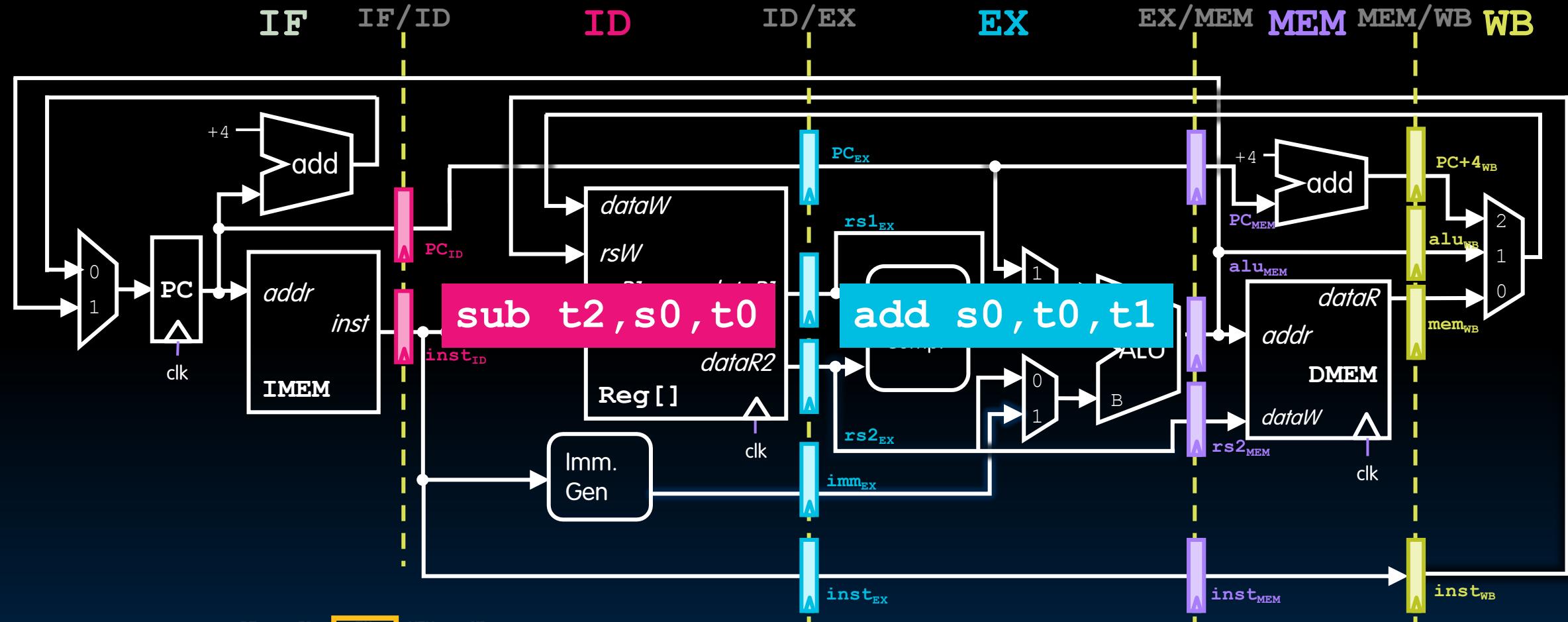
- Abstract: In this talk I will introduce some of my recent work on domain-specific hardware accelerators and systems, especially in the domains of machine learning and data analytics. With the end of Moore's Law and Dennard Scaling, specialization of hardware, algorithm and software stack has become the primary source of additional performance, and we start seeing an explosion of novel computer architectures. With this trend my research group has recently proposed various hardware accelerators and/or specialized software stacks in these domains. At the beginning of the talk, I will identify some of the challenges in designing scalable ML/data analytics computing platforms and briefly overview our work addressing these challenges. And then, I will present in greater depth a couple of case studies of hardware/software co-design---one from ML and the other from data analytics.
- Bio: I am a Professor of CSE at Seoul National University (SNU), South Korea, and a Visiting Faculty Researcher at Google Brain, Mountain View, USA. I have received my Ph.D. degree in CS from MIT under supervision of Krste Asanovic, M.S. degree in EE from Stanford, and B.S. degree in EE from SNU. I am a computer architect by training and working at the intersection of hardware, system software and algorithms to make computers faster and more cost/energy-efficient, with emphasis on ML and data analytics applications. I worked on various projects in the past, including MIT Raw microprocessor, first silicon PUF (Physically Unclonable Function), architecture support for dynamic information flow tracking (DIFT), QoS-capable uncore design, tagless DRAM cache architectures, self-attention/garbage collection/(de)serialization/search/ genomics/page-fault handling/... /hardware accelerators, to name a few. (Sorry - they were not "a few".)

Garcia, Yan

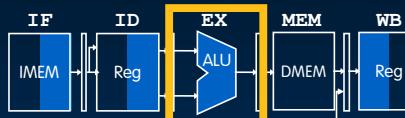
Data Hazards II: Forwarding

- Data Hazards II:
Forwarding
- Data Hazards III: Load
- Control Hazards
- Superscalar Processors
and Measuring CPI

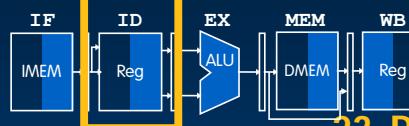
MEM/WB Pipeline Registers



add s0, t0, t1



sub t2, s0, t0

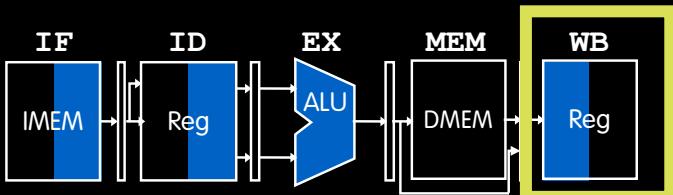


Pipeline registers allow multiple instructions to execute in separate stages.

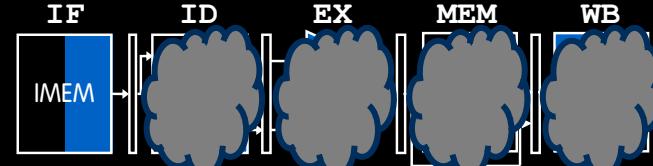
Data Hazards, Solution 1: Stalling

Review

time →

add s0, t0, t1

sub → nop



sub → nop

sub t2, s0, t0or t6, s0, t3

s0 value

5	5	5	5	5/9	9	9	9	9
---	---	---	---	-----	---	---	---	---

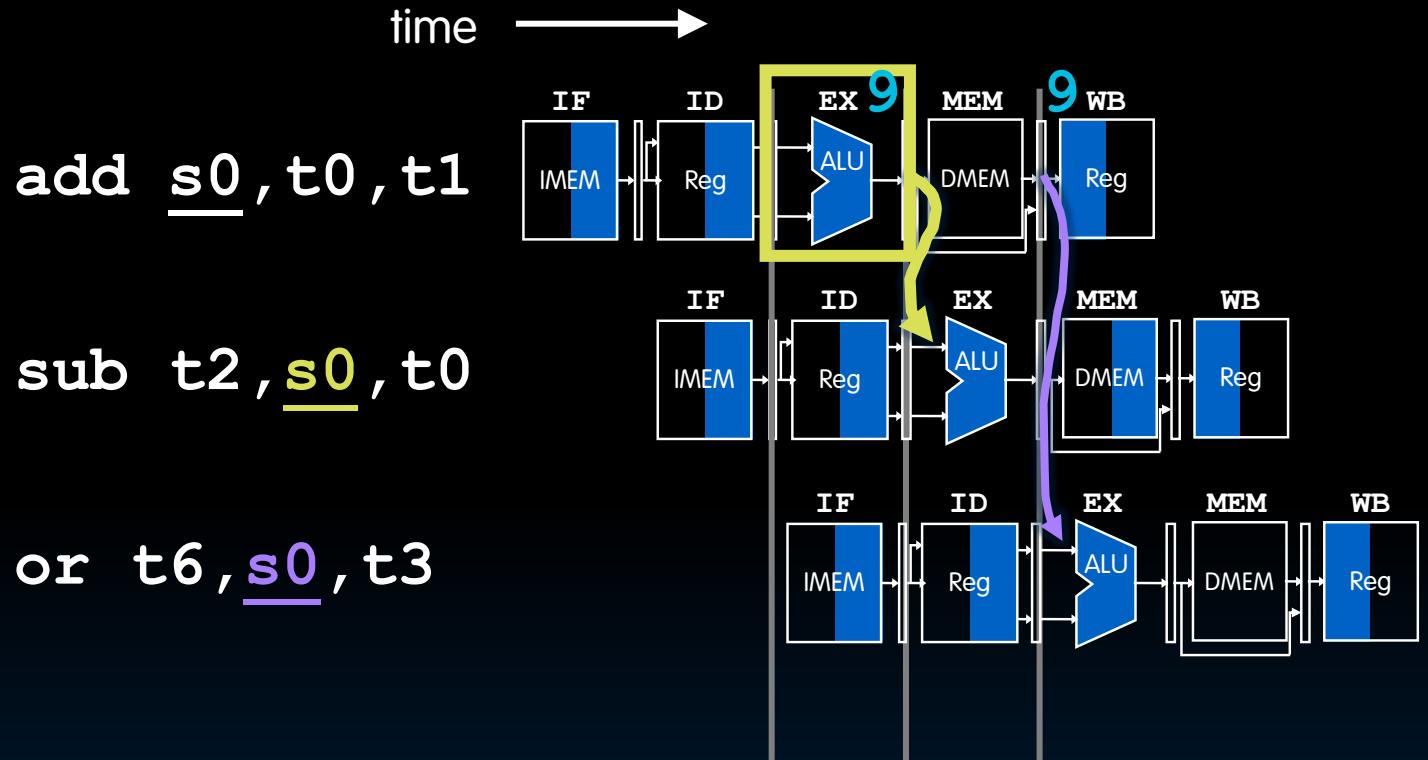
Problem: Instruction depends on WB's RegFile write from previous instruction.

- 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*.

Solution 2: Forwarding



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.*

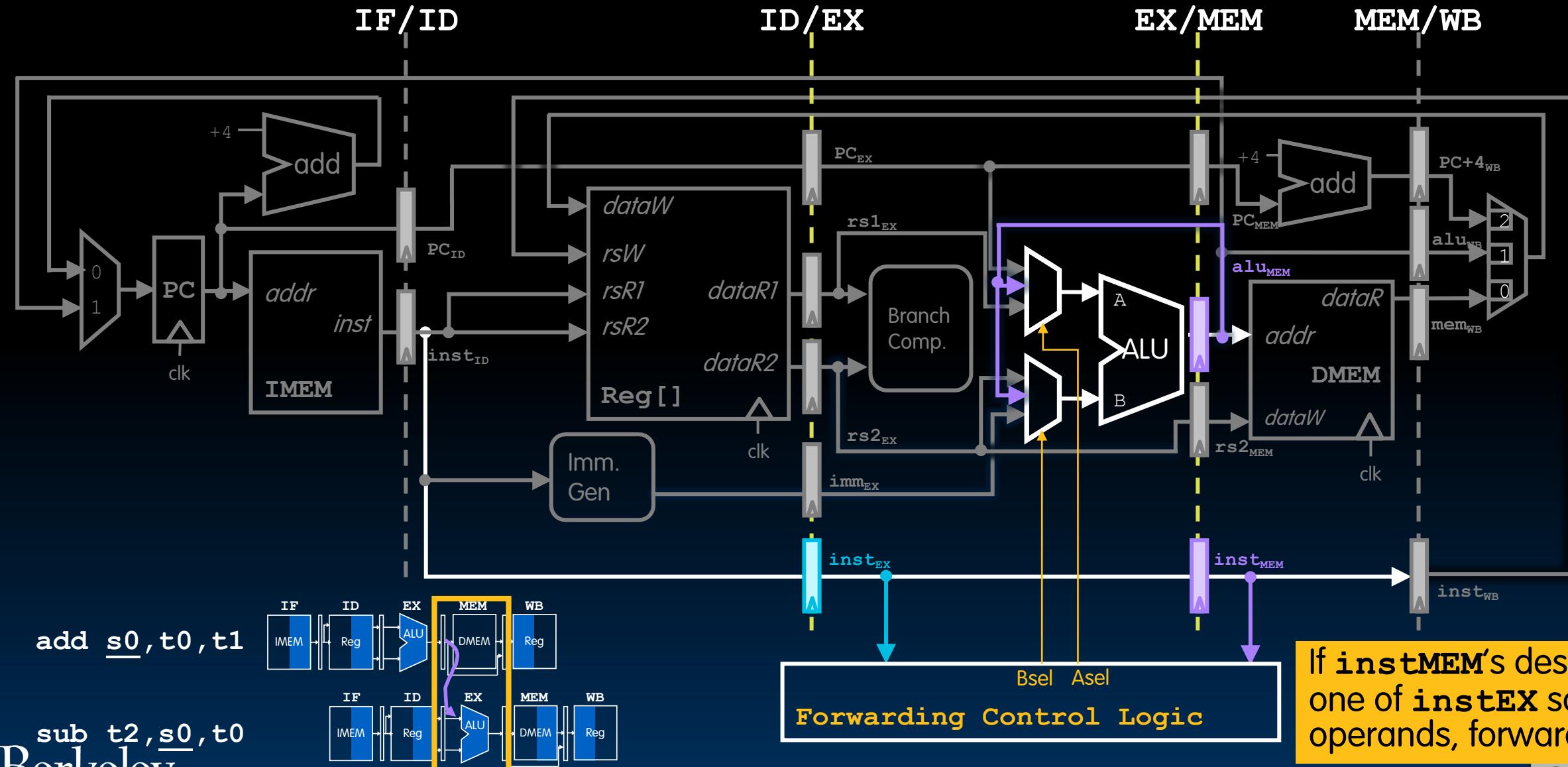
Implementation:

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

s0 value [5] [5] [5] [5] [5/9] [9] [9] [9] [9]

Forwarding EX Results

(MEM Forwarding is similar but omitted for simplicity.)

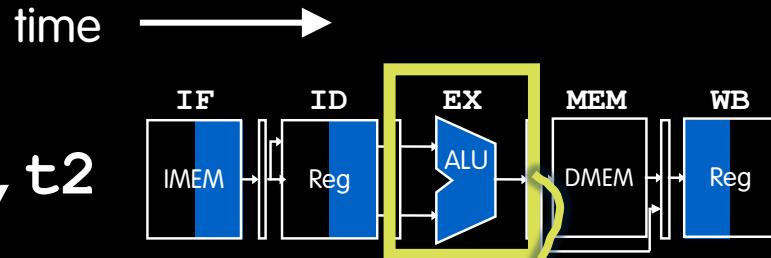


Data Hazards III: Load

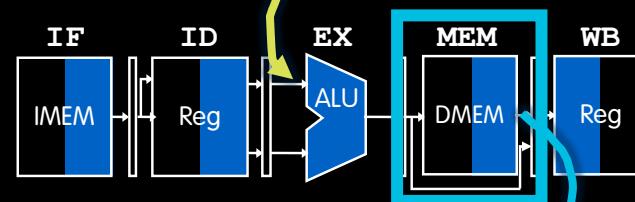
- Data Hazards II:
Forwarding
- Data Hazards III: Load
- Control Hazards
- Superscalar Processors
and Measuring CPI

Forwarding Cannot Fix All Data Hazards (1/2)

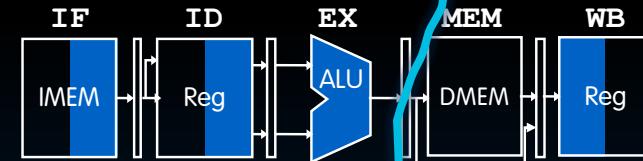
add s0, t1, t2



lw s1, 8(s0)



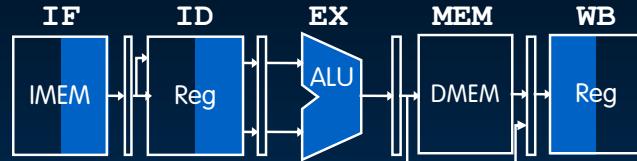
or t3, s1, t1



and t4, s1, t2



sll t0, t1, t2

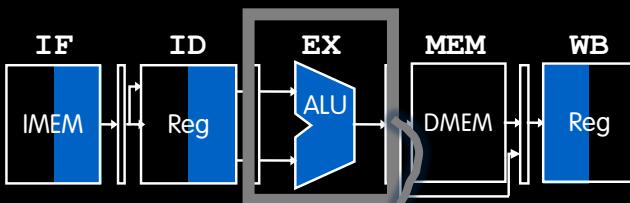
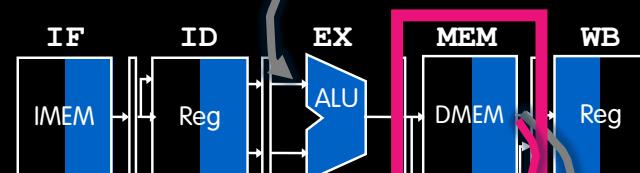
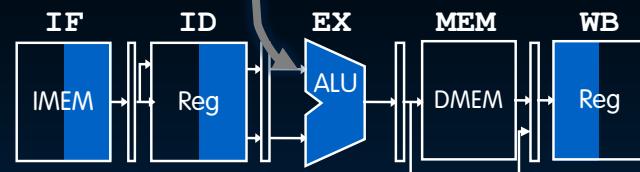


- ✓ 1. Forward EX stage output to input of EX stage on next clock cycle.

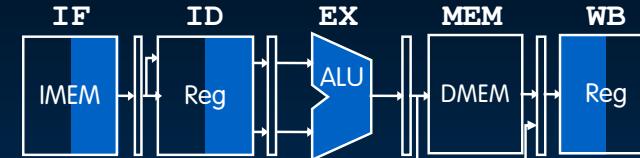
- ✓ 2. Forward MEM stage output to input of EX stage on next clock cycle.

Forwarding Cannot Fix All Data Hazards (2/2)

time →

add s0, t1, t2lw s1, 8(s0)or t3, s1, t1and t4, s1, t2

sll t0, t1, t2



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

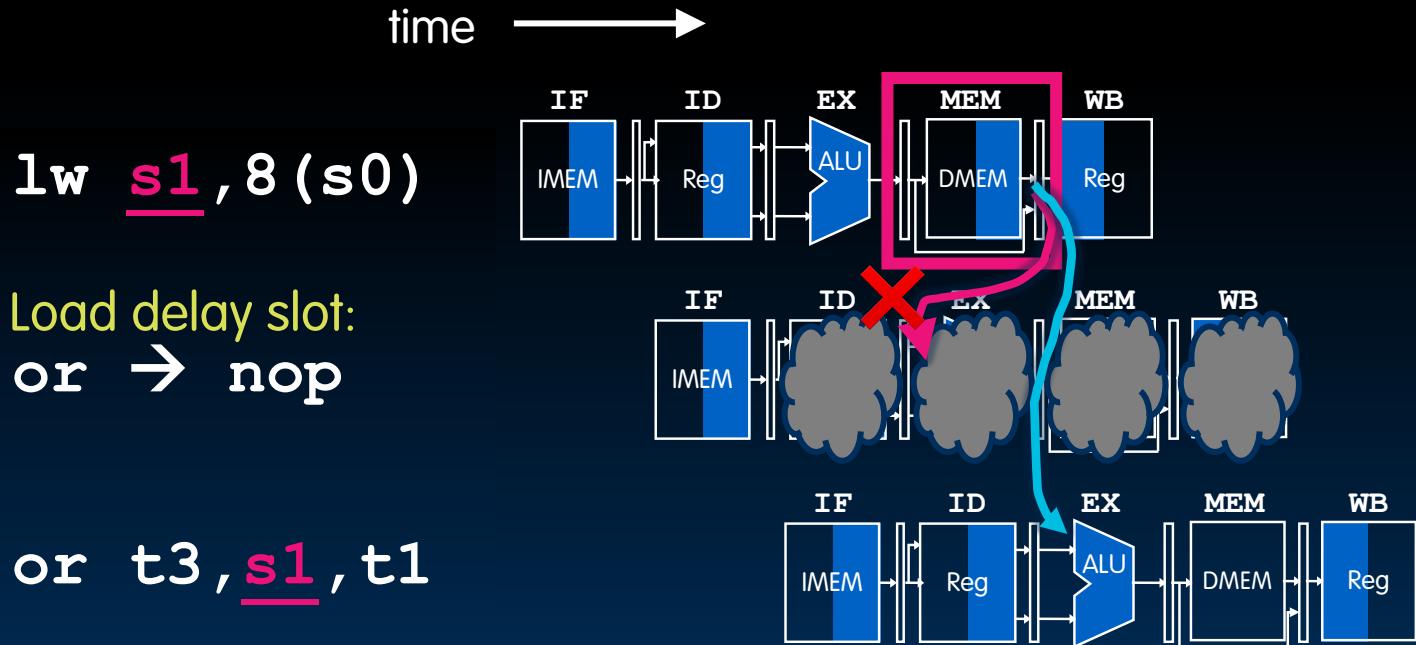
⚠ 3. **MEM** stage (**lw**)'s output needed as **EX** stage (**or**)'s input *in the same clock cycle*.

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

Loads can result in an unavoidable pipeline stall.

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!



⚠ **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!

Solution: 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];
```

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

⚠ Simple compilation
(9 cycles for 7 instructions)

```
lw t1, 0(t0)  
lw t2, 4(t0)  
add t3, t1, t2  
sw t3, 12(t0)  
lw t4, 8(t0)  
add t5, t1, t4  
sw t5, 16(t0)
```

Stall & forward! (+1 cycle) (+1 cycle)

✓ Alternative
(7 cycles):

```
lw t1, 0(t0) Forward!  
lw t2, 4(t0) (+0 cycle)  
lw t4, 8(t0)  
add t3, t1, t2  
sw t3, 12(t0)  
add t5, t1, t4  
sw t5, 16(t0)
```

Control Hazards

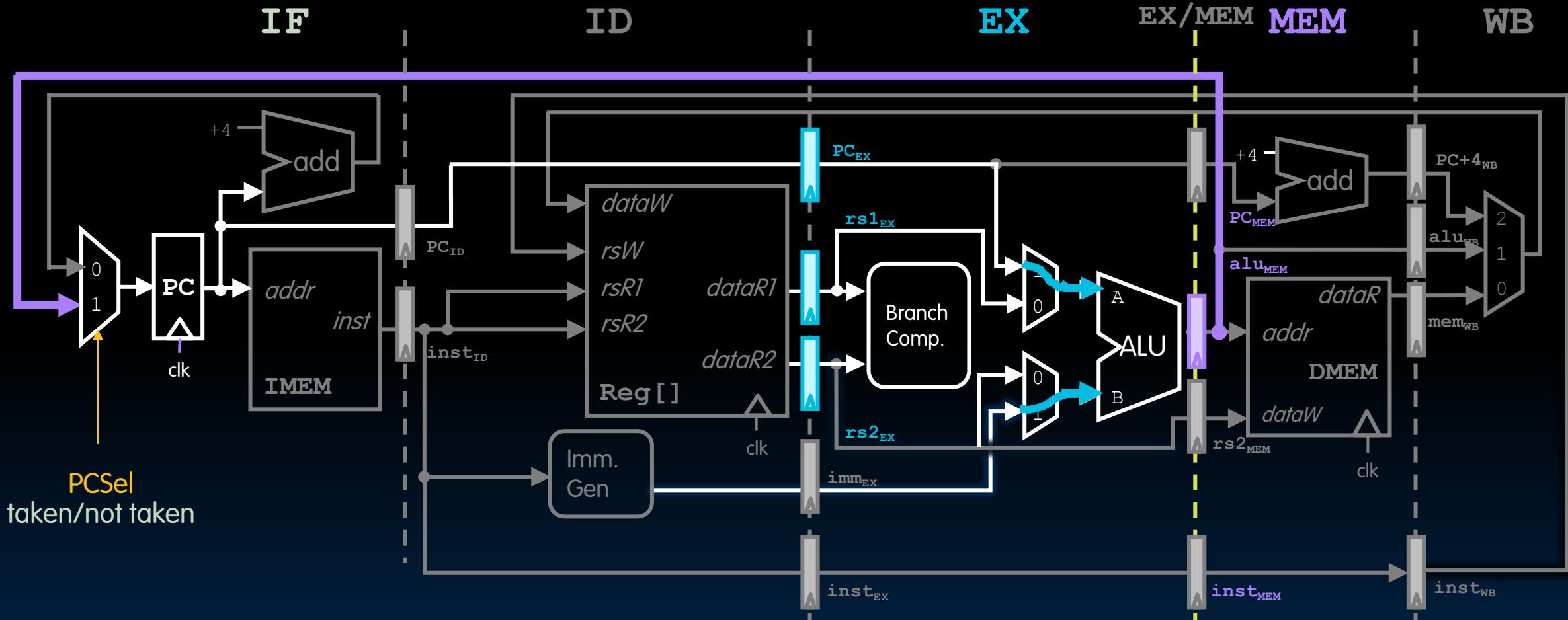
- Data Hazards II: Forwarding
- Data Hazards III: Load
- Control Hazards
- Superscalar Processors and Measuring CPI

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

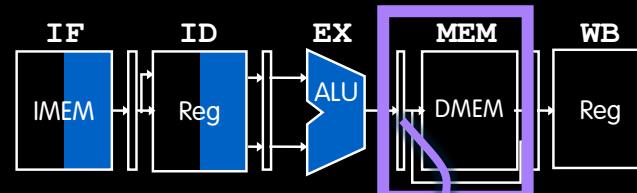


- In **MEM** stage: **EX/MEM** pipeline reg. feeds **IF** stage MUX. PCSel control is set.
- On the next clock cycle in the **IF** stage, PC updates, and the correct instruction is fetched.; fetches the correct instruction.

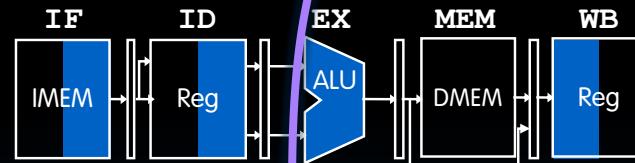
Control Hazard: Conditional Branches

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

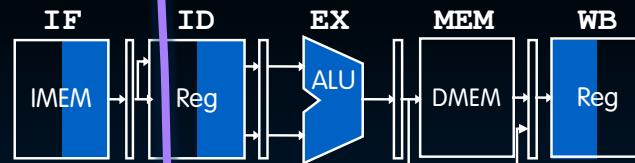
`0x40 beq t0, t1, Label`



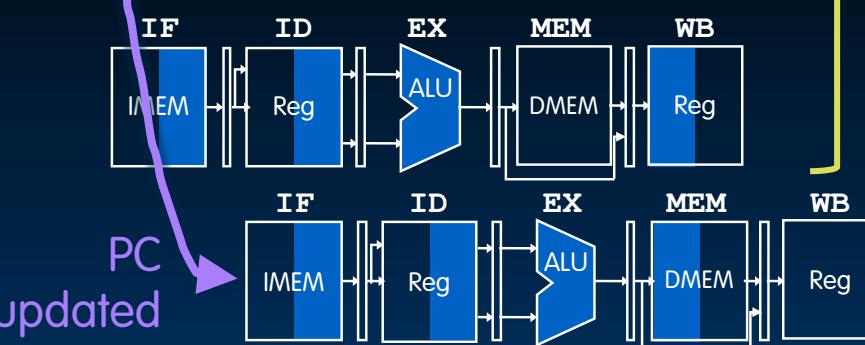
`0x44 sub t2, s0, t0`



`0x48 or t6, s0, t3`



`0x4c xor t5, t1, s0`



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

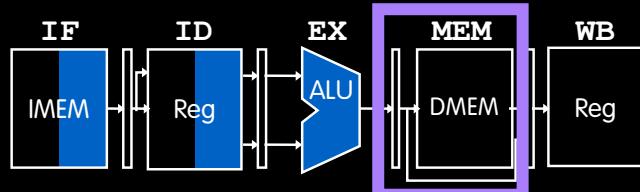
Instruction execution starts before branch outcome is known!

Correct instruction starts executing

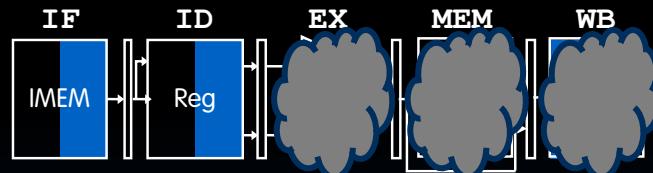
Garcia, Yan

Kill Instructions after Branch (If Taken)

0x40 beq t0, t1, Label



0x44 sub t2, s0, t0



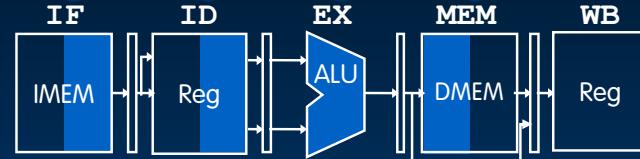
0x48 or t6, s0, t3



0x4c xor t5, t1, s0



0x70 sw s0, 8(t3) # Label



Flush pipeline by
converting
incorrect
instructions to
nops.

PC updated, correct
instruction
loaded

Garcia, Yan



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*

"Guess" next PC to be PC + 4

0x40 beq t0, t1, Label

"Evaluate" guess

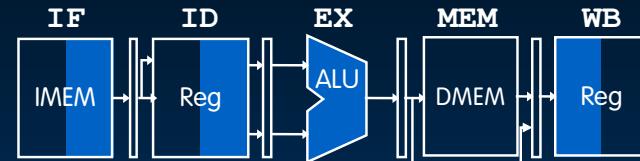
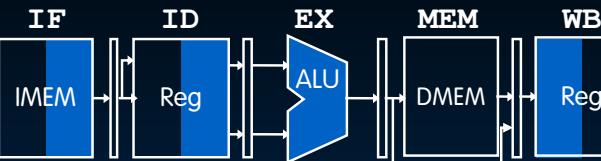
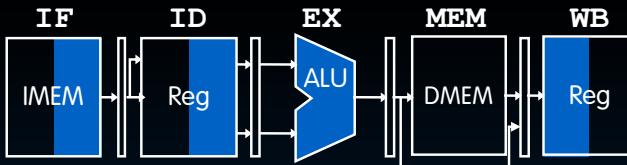
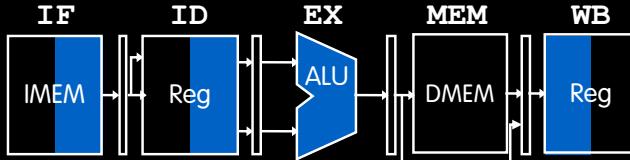
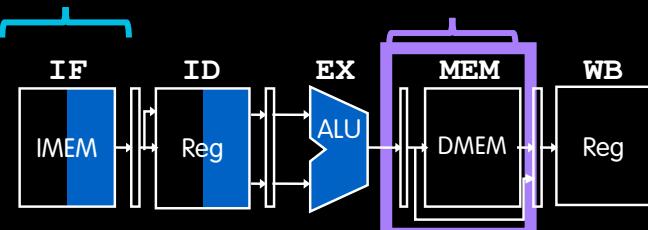
0x44 sub t2, s0, t0

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

0x48 or t6, s0, t3

0x4c xor t5, t1, s0

0x50 add t2, s0, s0



time →

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*.

Superscalar Processors and Calculating CPI

- Data Hazards II:
Forwarding
- Data Hazards III: Load
- Control Hazards
- Superscalar Processors
and Measuring CPI

- **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.

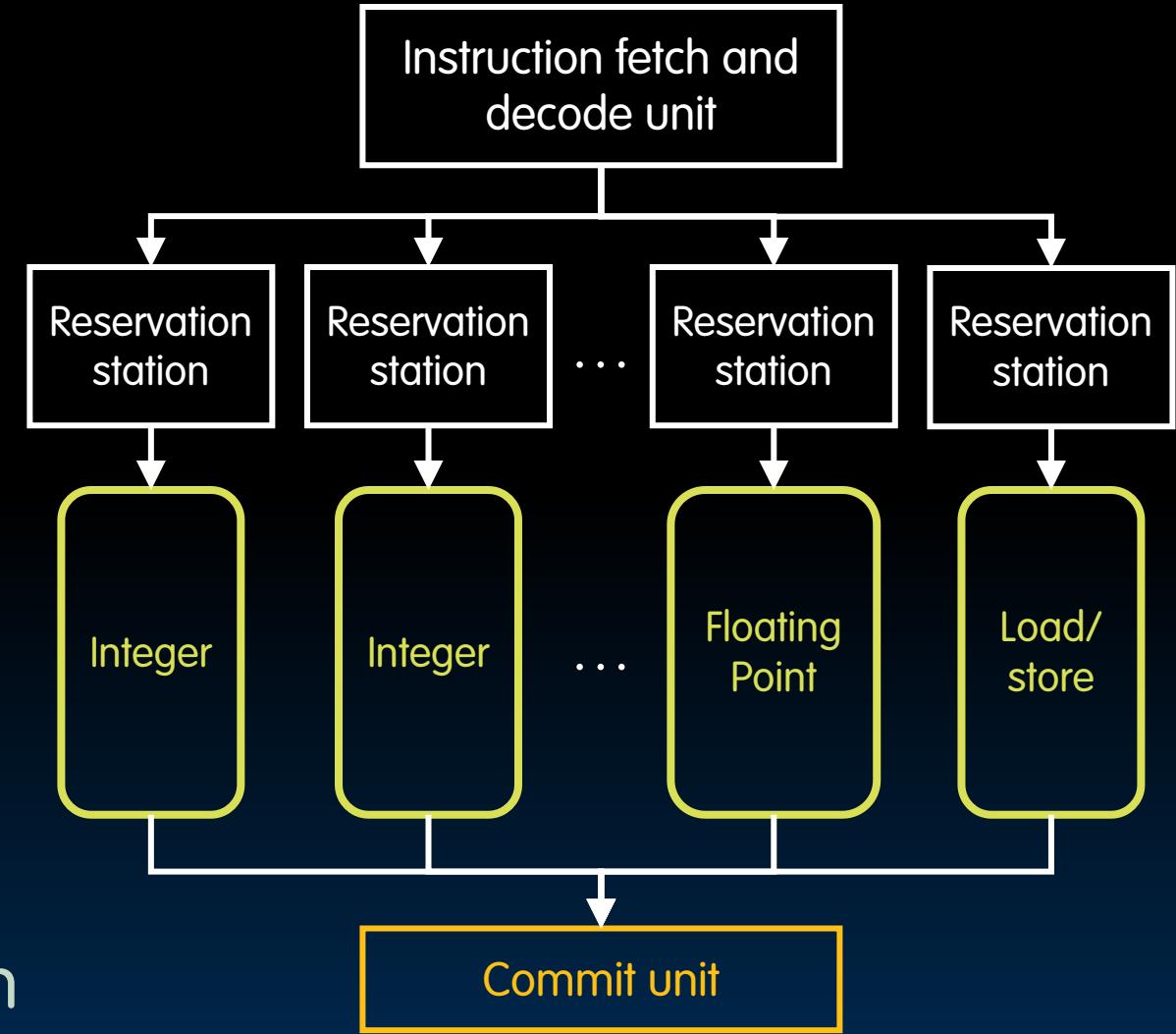
Superscalar Processors

- **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.



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:

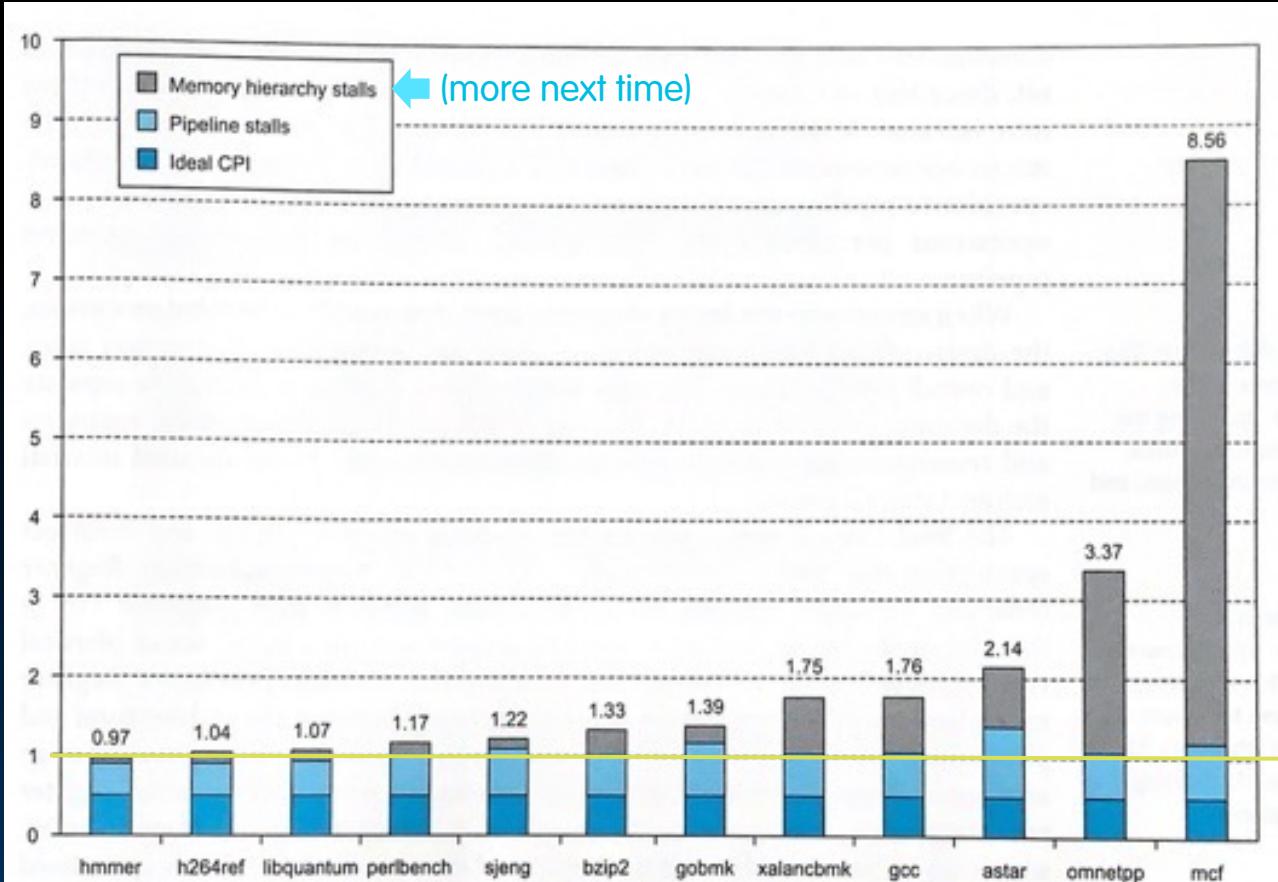
$$\frac{\text{time}}{\text{program}} = \underbrace{\frac{\text{instructions}}{\text{program}}}_{\text{(can measure)}} \times \underbrace{\frac{\text{cycles}}{\text{instruction}}}_{\text{(can count)}} \times \underbrace{\frac{\text{time}}{\text{cycle}}}_{\text{CPI}} \times \underbrace{\frac{1}{\text{clock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

We compute CPI on different benchmarks:

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

ARM A53 Benchmark

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



If $\text{CPI} < 1$, can also measure in *IPC* (*Instructions Per Cycle*).

$\text{CPI} = 1$

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