ECE 3057: Architecture, Concurrency and Energy in Computation
Summer 2019

http://tusharkrishna.ece.gatech.edu/teaching/uca_f18/

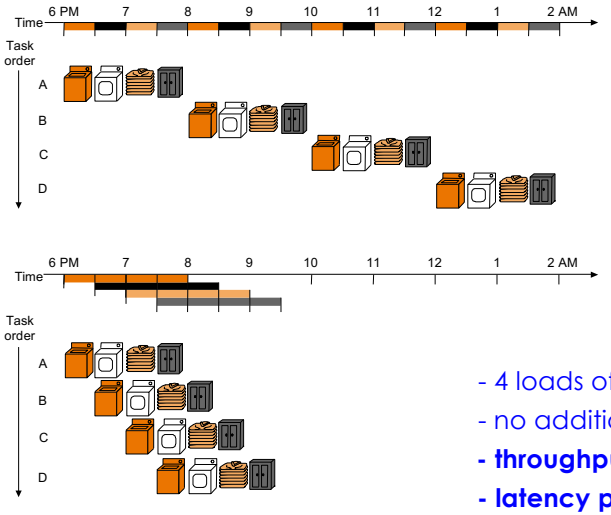
Lecture 6: Hazards and Stalls

David Anderson

School of Electrical and Computer Engineering
Georgia Institute of Technology

Acknowledgment: Lecture slides adapted from GT ECE 3056 (S. Yalamanchilli, T. Krishna), MIT 6.823 (Arvind and J. Emer) and CMU 18-447 (O. Mutlu)

Recap: Pipelining



- 4 loads of laundry in parallel
- no additional resources
- **throughput increased by 4**
- **latency per load is the same**

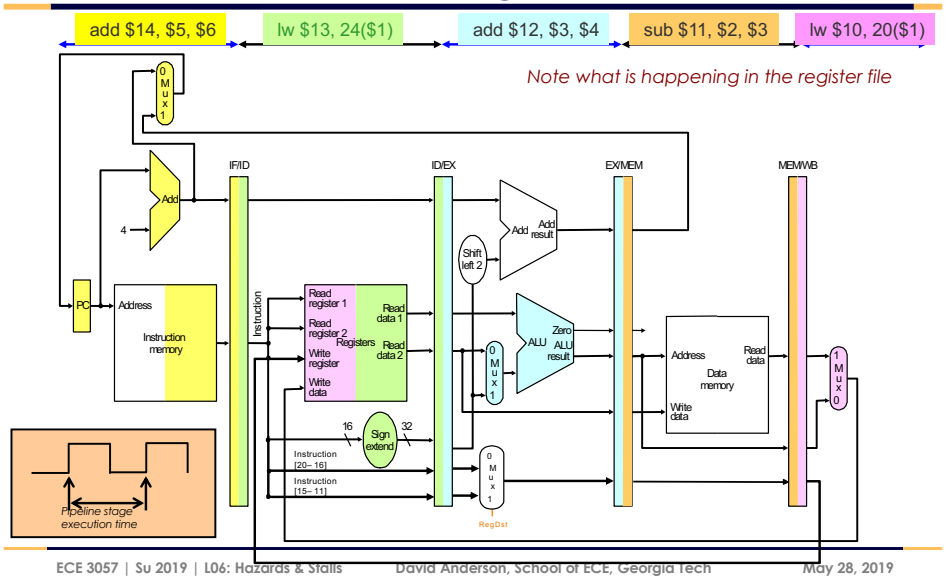
An "Ideal" Pipeline

- 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 a processor?

5-stage Processor Pipeline

- Instruction fetch (IF)
- Instruction decode and register operand fetch (ID)
- Execute/Evaluate memory address (EX)
- Memory operand fetch (MEM)
- Writeback result to register (WB)

Instruction Pipelining Example



Is the Processor an Ideal Pipeline?

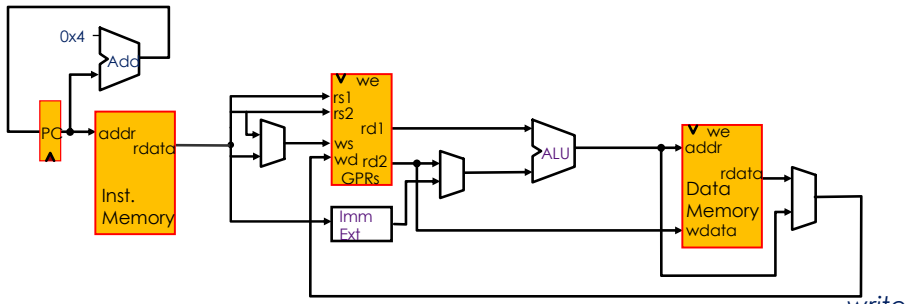
- Repetition of identical operations?
  - The same operation is repeated on a large number of different inputs (e.g., all laundry loads go through the same steps)
    - ⇒ **No.** different instructions do not all need the same stages
      e.g., ADD instruction does not use Mem
- Repetition of independent operations?
  - No dependencies between repeated operations
    - ⇒ **No.** Instructions depend on each other.
- Uniformly partitionable suboperations?
  - Processing can be evenly divided into uniform-latency suboperations (that do not share resources)
    - ⇒ **No.** Different stages may have different latency

How do we solve this?

- Repetition of identical operations?
  - ⇒ **No.** different instructions do not all need the same stages
    e.g., ADD instruction does not use Mem

    => Some pipeline stages are idle for some instructions
- Repetition of independent operations?
  - ⇒ **No.** Instructions depend on each other.
- Uniformly partitionable suboperations?
  - ⇒ **No.** Different stages may have different latency

How to decide pipeline stages?



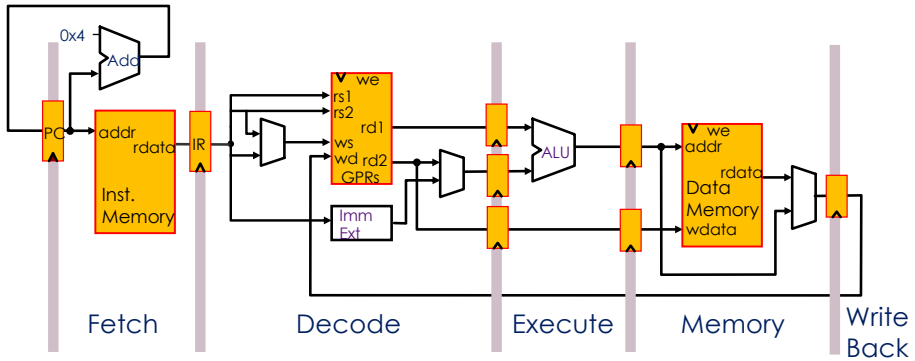
	fetch phase	*decode & Reg-fetch phase*	*execute phase*	*memory phase*	*write -back phase*
Delay	10ns	1ns	5ns	10ns	1ns

Since the slowest stage determines the clock, it may be possible to combine some stages without slowing down frequency

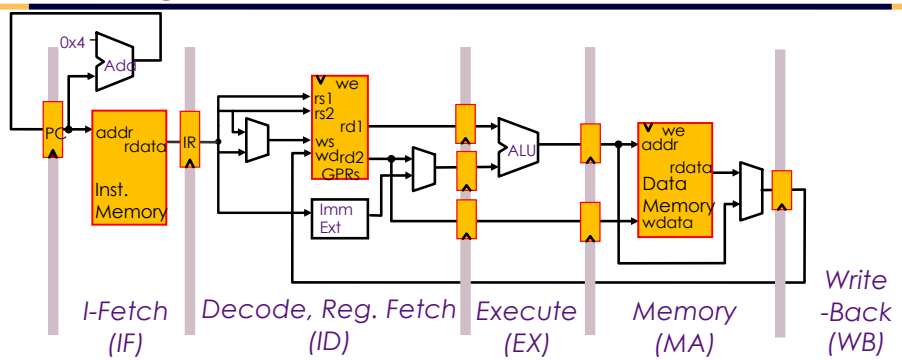
How do we solve this?

- Repetition of identical operations?
  - ⇒ **No.** different instructions do not all need the same stages
    e.g., ADD instruction does not use Mem
  - => Some pipeline stages are idle for some instructions
- Repetition of independent operations?
  - ⇒ **No.** Instructions depend on each other.
- Uniformly partitionable suboperations?
  - ⇒ **No.** Different stages may have different latency
  - => Some pipe stages are too fast but all take the same cycle (as long as the slowest stage)

Classic 5-stage Pipeline

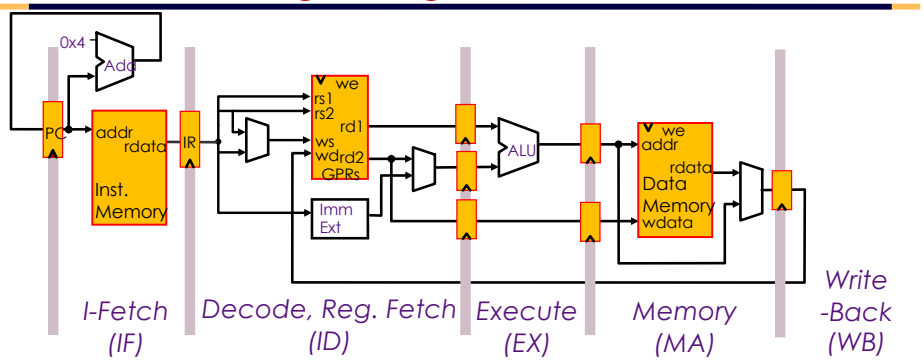


5-Stage Pipelined Execution



time	t0	t1	t2	t3	t4	t5	t6	t7	. . . .
instruction1		IF_1	ID_1	EX_1	MA_1	WB_1			
instruction2			IF_2	ID_2	EX_2	MA_2	WB_2		
instruction3				IF_3	ID_3	EX_3	MA_3	WB_3	
instruction4					IF_4	ID_4	EX_4	MA_4	WB_4
instruction5						IF_5	ID_5	EX_5	MA_5 WB_5

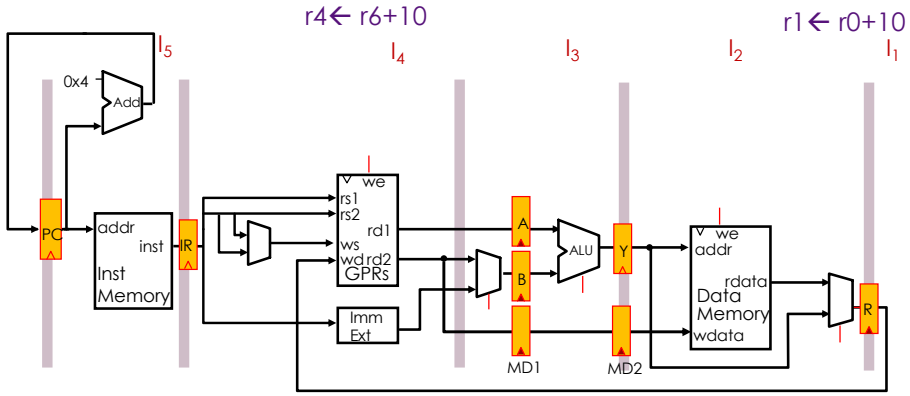
5-Stage Pipelined Execution
Resource Usage Diagram



Resources

time	t0	t1	t2	t3	t4	t5	t6	t7	. . . .
IF		I_1	I_2	I_3	I_4	I_5			
ID			I_1	I_2	I_3	I_4	I_5		
EX				I_1	I_2	I_3	I_4	I_5	
MA					I_1	I_2	I_3	I_4	I_5
WB						I_1	I_2	I_3	I_4 I_5

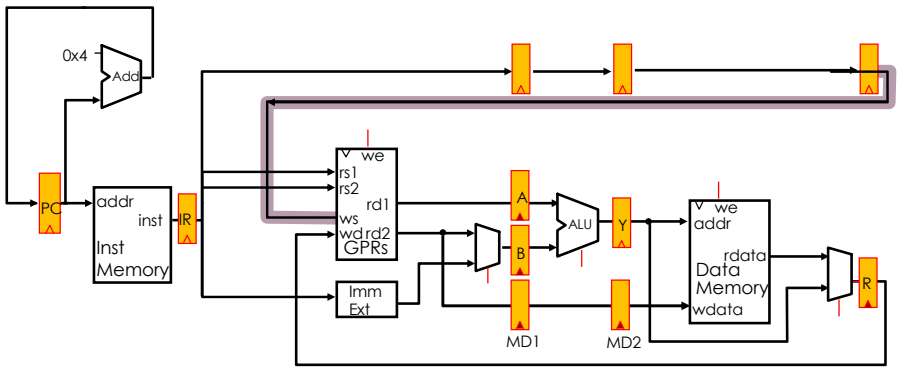
Pipeline Implementation



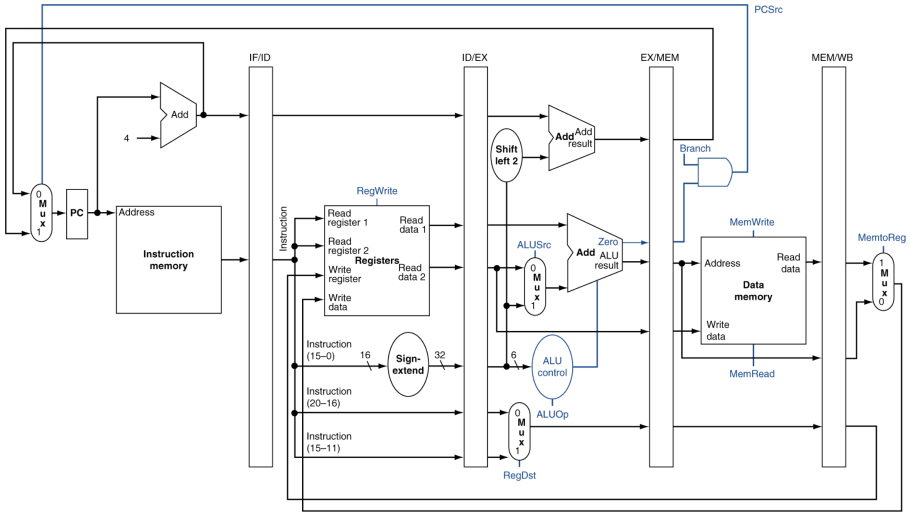
Which register should I_1 write back its result into? r1 *But ws is pointing to r4!*

Control signals per stage also need to go through pipeline registarers

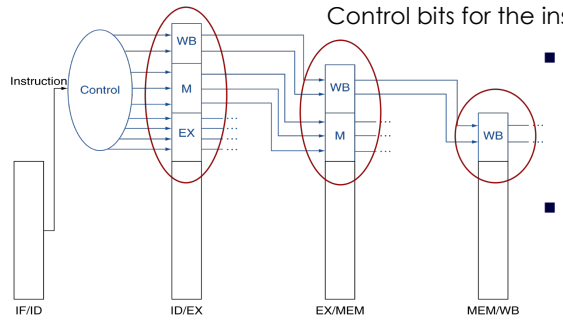
Pipeline Implementation



5-Stage Pipeline Diagram



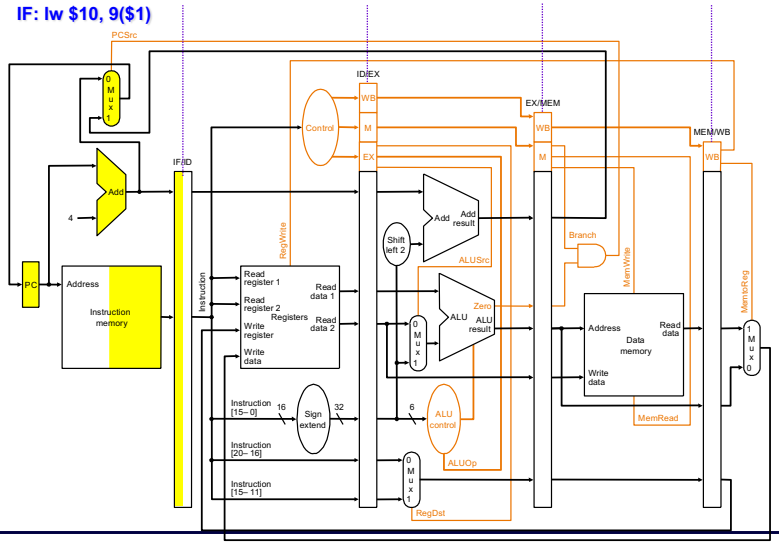
Pipelined Control



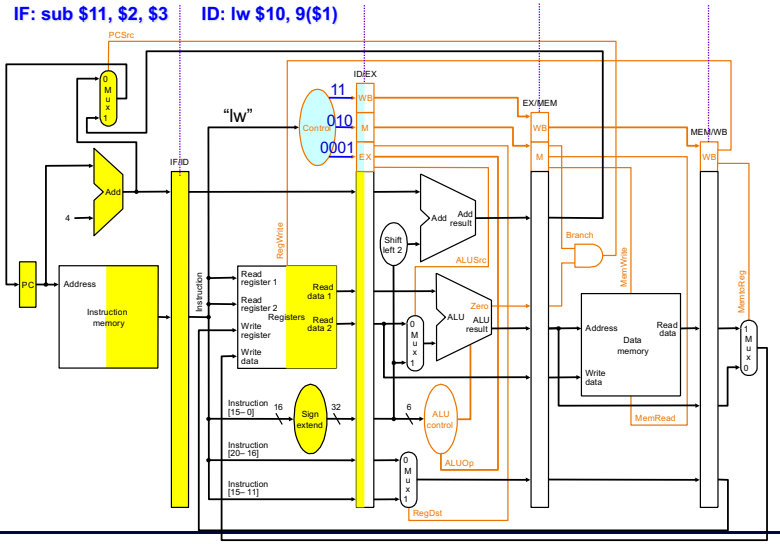
- Control signals derived from instruction
  - As in single-cycle implementation
- Pass control signals along like data

Instruction	Execution/Address Calculation stage control lines				Memory access stage control lines			Write-back stage control lines	
	Reg Dst	ALU Op1	ALU Op0	ALU Src	Branch	Mem Read	Mem Write	Reg write	Mem to Reg
R-format	1	1	0	0	0	0	0	1	0
lw	0	0	0	1	0	1	0	1	1
sw	X	0	0	1	0	0	1	0	X
beq	X	0	1	0	1	0	0	0	X

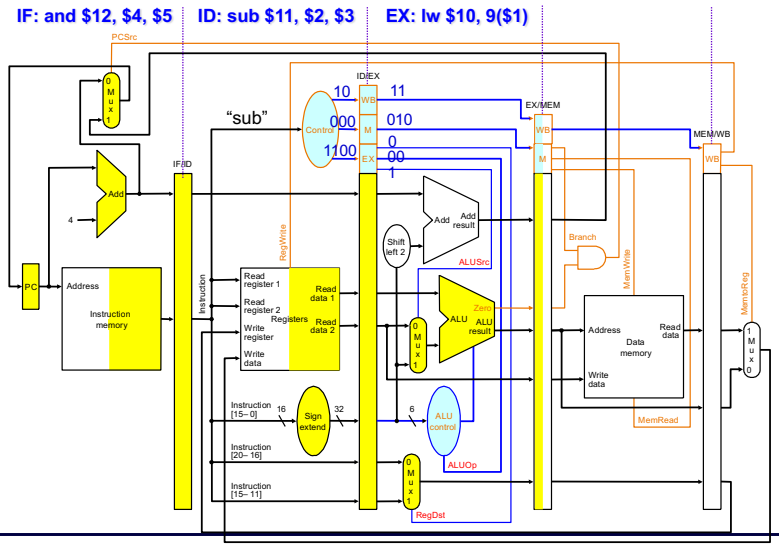
Datapath with Control



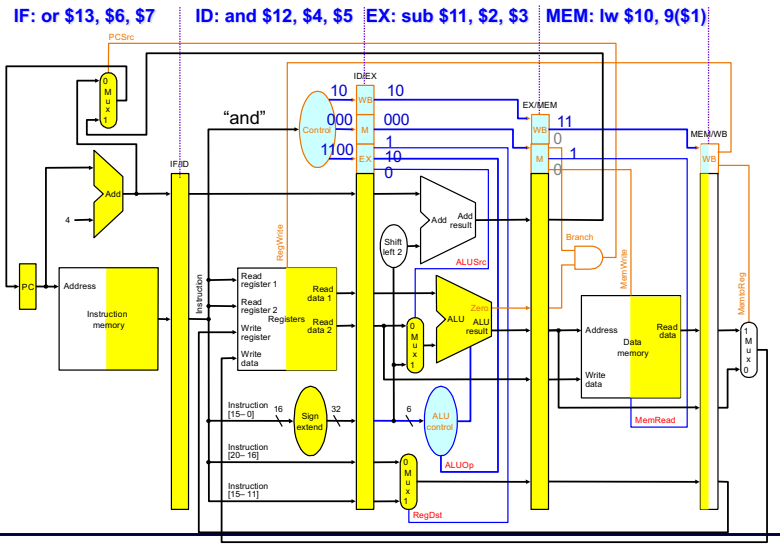
Datapath with Control



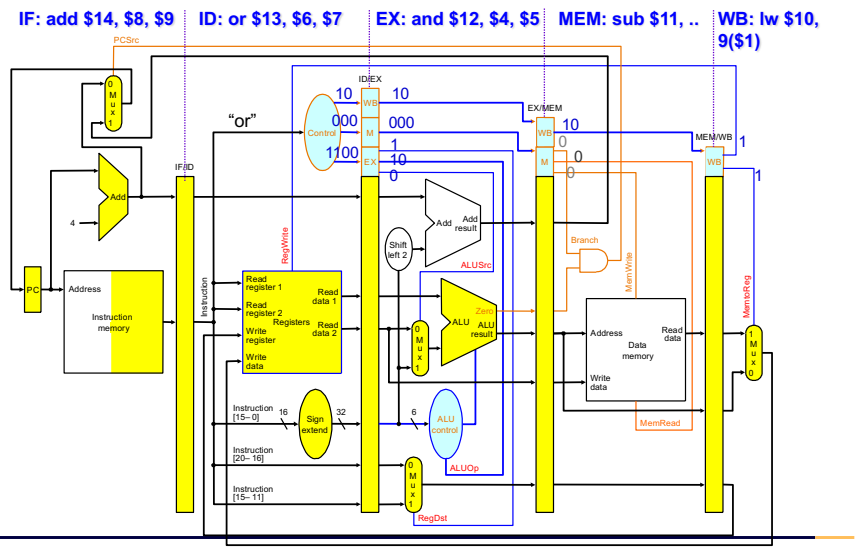
Datapath with Control



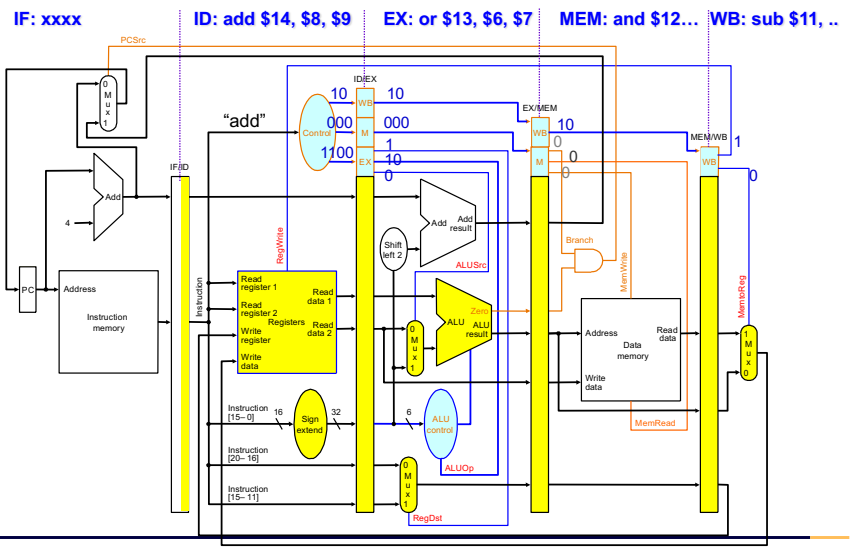
Datapath with Control



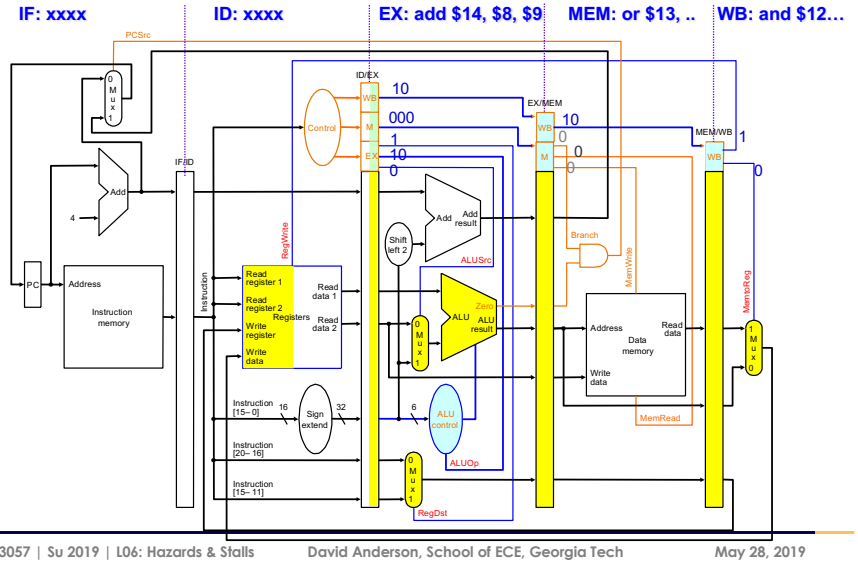
Datapath with Control



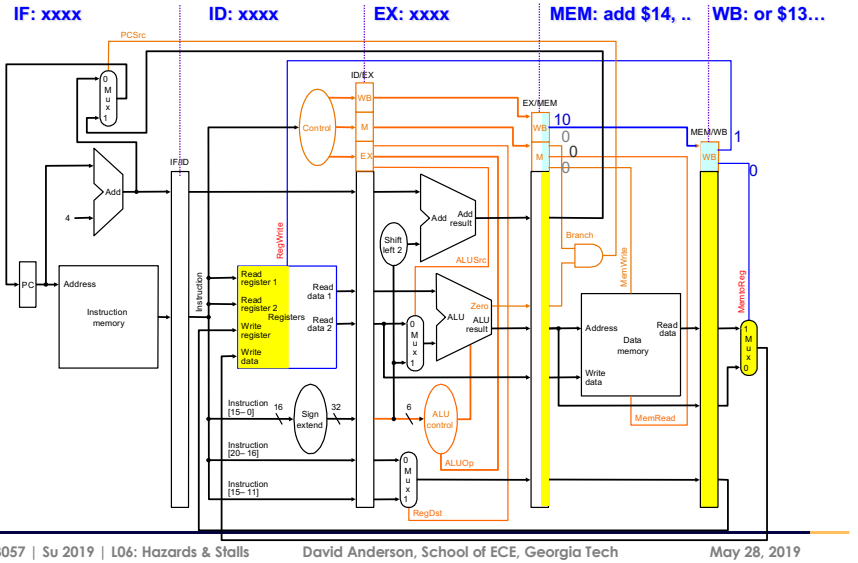
Datapath with Control



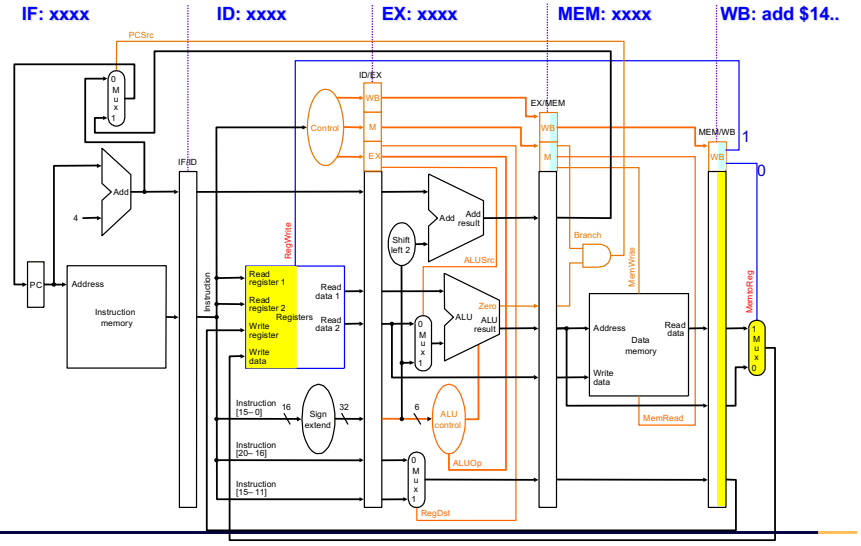
Datapath with Control



Datapath with Control



Datapath with Control



How do we solve this?

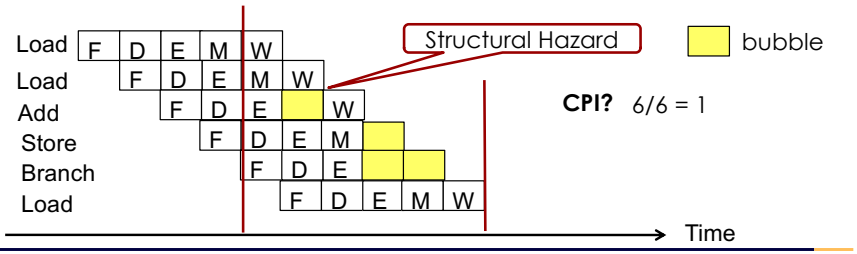
- Repetition of identical operations?
  - ⇒ **No.** different instructions do not all need the same stages
    e.g., ADD instruction does not use Mem
  - => Some pipeline stages are idle for some instructions
- Repetition of independent operations?
  - ⇒ **No.** Instructions depend on each other.
  - => Called "Pipeline Hazards"
- Uniformly partitionable suboperations?
  - ⇒ **No.** Different stages may have different latency
  - => Some pipe stages are too fast but all take the same cycle (as long as the slowest stage)

Hazards

- An instruction in the pipeline may need a resource being used by another instruction in the pipeline → *structural hazard*
- An instruction may depend on something produced by an earlier instruction
  - Dependence may be for a data value
    → *data hazard*
  - Dependence may be for the next instruction's address
    → *control hazard (branches, exceptions)*

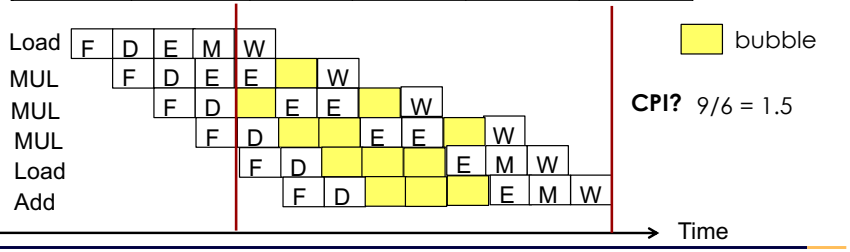
Structural Hazard Example 1

Instruction Class	Instruction Fetch 2ns	Decode + Reg 1ns	ALU Operation 2ns	Memory Access 2ns	Register Write 1ns
ALU	✓	✓	✓		✓
Load	✓	✓	✓	✓	✓
Store	✓	✓	✓	✓	
Branch	✓	✓	✓		



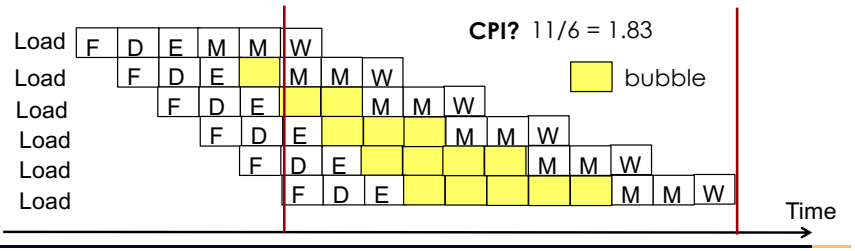
Structural Hazard Example 2

Instruction Class	Instruction Fetch 2ns	Decode + Reg 1ns	ALU **ADD: 2ns** **MUL: 4ns**	Memory Access 2ns	Register Write 1ns
ALU	✓	✓	✓		✓
Load	✓	✓	✓	✓	✓
Store	✓	✓	✓	✓	
Branch	✓	✓	✓		



Structural Hazard Example 3

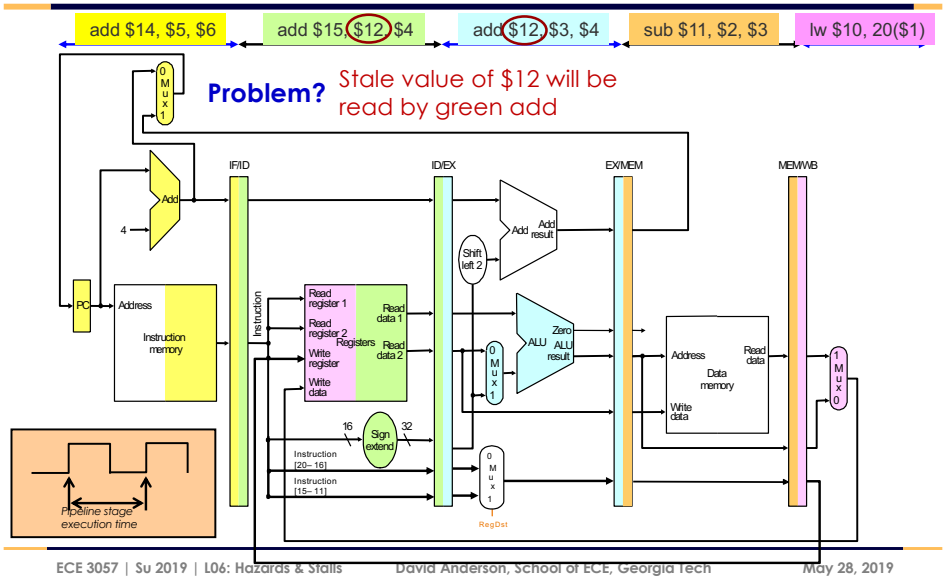
Instruction Class	Instruction Fetch 2ns	Decode + Reg 1ns	ALU Operation 2ns	Memory Access **4ns**	Register Write 1ns
ALU	✓	✓	✓		✓
Load	✓	✓	✓	✓	✓
Store	✓	✓	✓	✓	
Branch	✓	✓	✓		



Resolving Structural Hazards

- Structural hazard occurs when two instructions need same hardware resource at same time
- How to resolve a structure hazard?
  - Strategy 1: Stall
    - Stall newer instruction till older instruction finished with resource
    - Introduces bubbles in pipeline → lower CPI
  - Strategy 2: Add more hardware
    - E.g., if two instructions both need a port to memory at same time, could avoid hazard by adding second port to memory

Data Hazards



Types of Data Hazards

Flow dependence

$r_3 \leftarrow r_1$ op r_2
$r_5 \leftarrow r_3$ op r_4
Read-after-Write (RAW)

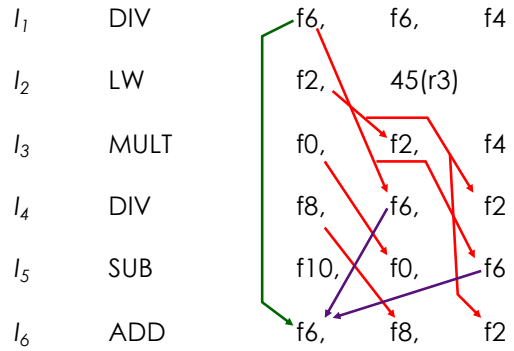
Anti dependence

$r_3 \leftarrow r_1$ 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)

Data Hazards Example



RAW Hazards
WAR Hazards
WAW Hazards

Register Data Dependence Analysis

- For a given pipeline, when is there a potential conflict between two data dependent instructions?
  - dependence type: RAW, WAR, WAW?
    - WAR and WAW do not cause a stall unless we do out-of-order instruction scheduling
  - instruction types involved?
    - **Read** – after – **Write**

ALU	LW	SW	BEQ/BNE	J	JR
✓	✓	✓	✓	X	✓

ALU	LW	SW	BEQ/BNE	J	JR
✓	✓	X	X	X	X

  - distance between the two instructions?
    - depends on pipeline length

Resolving Data/Control Hazards

- Strategy 0: remove dependence
  - Detect and eliminate the dependence at the software level
- Strategy 1: stall
  - *Wait for the result to be available by freezing earlier pipeline stages*
- Strategy 2: bypass/forward
  - *Route data as soon as possible after it is calculated to the earlier pipeline stage*
- Strategy 3: speculate
  - *Guess the result and proceed*
    - ***Guessed correctly*** → *no special action required*
    - ***Guessed incorrectly*** → *kill and restart*
- Strategy 4: do something else
  - *Switch to some other task/thread while dependence resolves*

Remove Dependence

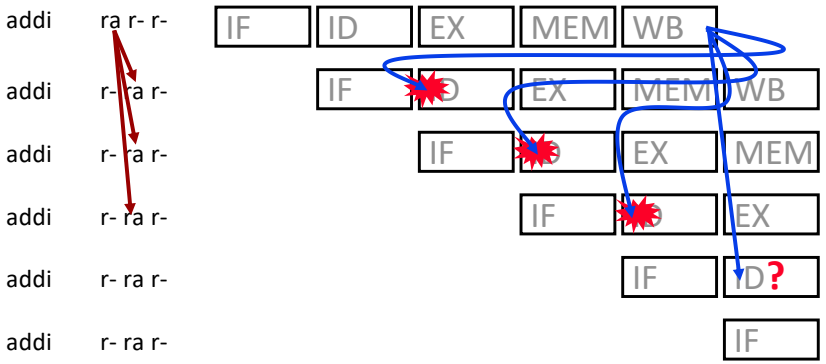
- Detect and eliminate the dependence at the software level
  - Challenge?
    - SW needs to know about the underlying implementation
    - Not scalable and inefficient

Resolving Data/Control Hazards

- Strategy 0: remove dependence
  - Detect and eliminate the dependence at the software level
- Strategy 1: stall
  - *Wait for the result to be available by freezing earlier pipeline stages*
- Strategy 2: bypass/forward
  - *Route data as soon as possible after it is calculated to the earlier pipeline stage*
- Strategy 3: speculate
  - *Guess the result and proceed*
    - ***Guessed correctly*** → *no special action required*
    - ***Guessed incorrectly*** → *kill and restart*
- Strategy 4: do something else
  - *Switch to some other task/thread while dependence resolves*

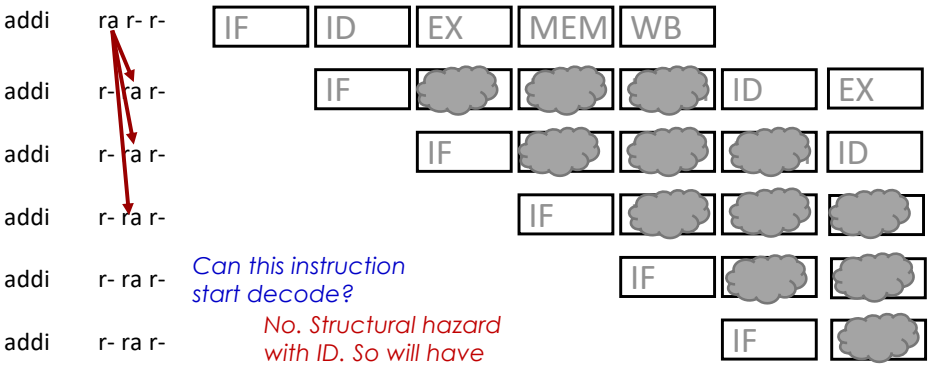
Stall Example

- Which one of the following flow dependences lead to conflicts in the 5-stage pipeline?



Stall Example

- What will be the actual execution with stalls?



Can this instruction start decode?

No. Structural hazard with ID. So will have to stall as well.

Example

I1: SUB $t5, $t0, $t0
I2: XOR $t6, $t5, $t1
I3: SW $t6, 0($t2)
I4: LW $t1, 0($t2)
I5: ADD $t3, $t1, $t0

CPI? 14/5

Time	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
SUB	F	D	E	M	W															
XOR		F	D	D	D	D	E	M	W											
SW			F	F	F	F	D	D	D	D	E	M	W							
LW							F	F	F	F	D	E	M	W						
ADD											F	D	D	D	D	E	M	W		

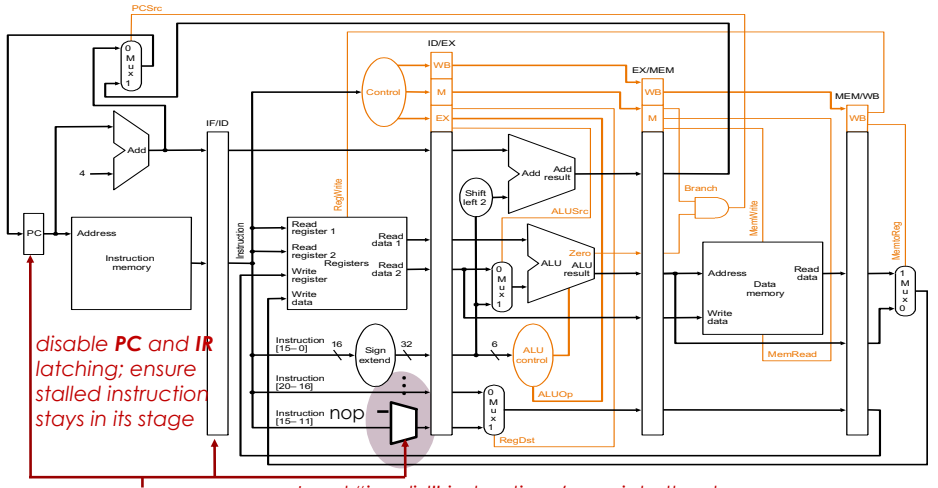
Implementing a Stall

- Detect the dependence
  - Instructions I_A and I_B (where I_A comes before I_B) have RAW dependence iff
    - I_B (R/I, LW, SW, Br or JR) reads a register written by I_A (R/I or LW)
    - $dist(I_A, I_B) \leq dist(ID, WB) = 3$
  - How do we detect this?
- Stall
  - stall the ID stage when I_B in ID stage wants to read a register to be written by I_A in EX, MEM or WB stage
  - How?

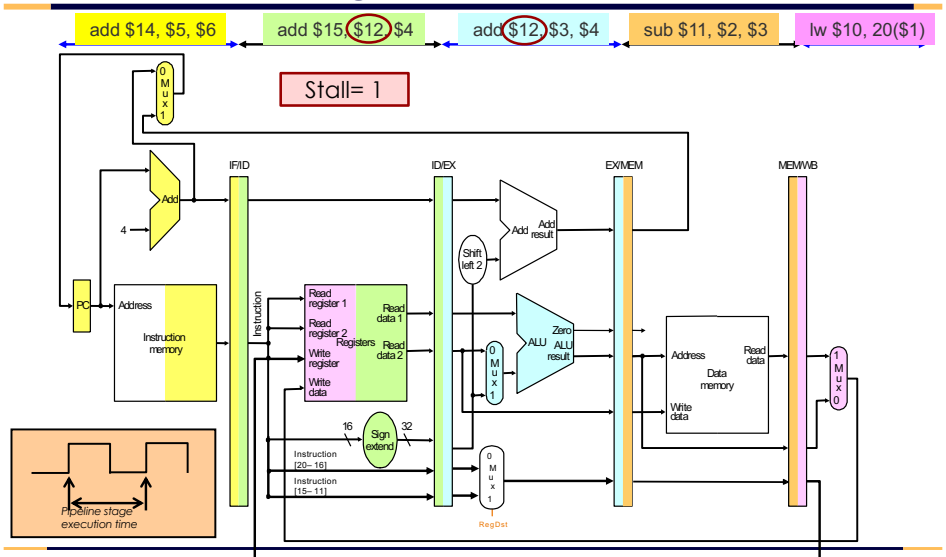
Dependence Detection (i.e., Stall Condition)

- Helper functions
  - rs(I) returns the rs field of I
  - use_rs(I) returns true if I requires RF[rs] and rs!=r0
- Stall when
  - $(rs(IR_{ID}) == dest_{EX})$ && $use\_rs(IR_{ID})$ && $RegWrite_{EX}$ or
  - $(rs(IR_{ID}) == dest_{MEM})$ && $use\_rs(IR_{ID})$ && $RegWrite_{MEM}$ or
  - $(rs(IR_{ID}) == dest_{WB})$ && $use\_rs(IR_{ID})$ && $RegWrite_{WB}$ or
  - $(rt(IR_{ID}) == dest_{EX})$ && $use\_rt(IR_{ID})$ && $RegWrite_{EX}$ or
  - $(rt(IR_{ID}) == dest_{MEM})$ && $use\_rt(IR_{ID})$ && $RegWrite_{MEM}$ or
  - $(rt(IR_{ID}) == dest_{WB})$ && $use\_rt(IR_{ID})$ && $RegWrite_{WB}$
- It is crucial that the EX, MEM and WB stages continue to advance normally during stall cycles

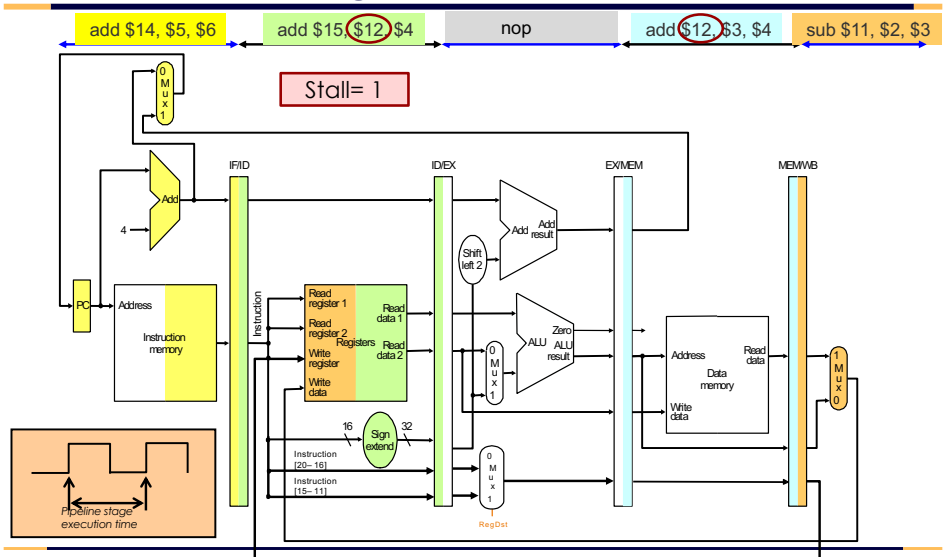
How to implement stalling



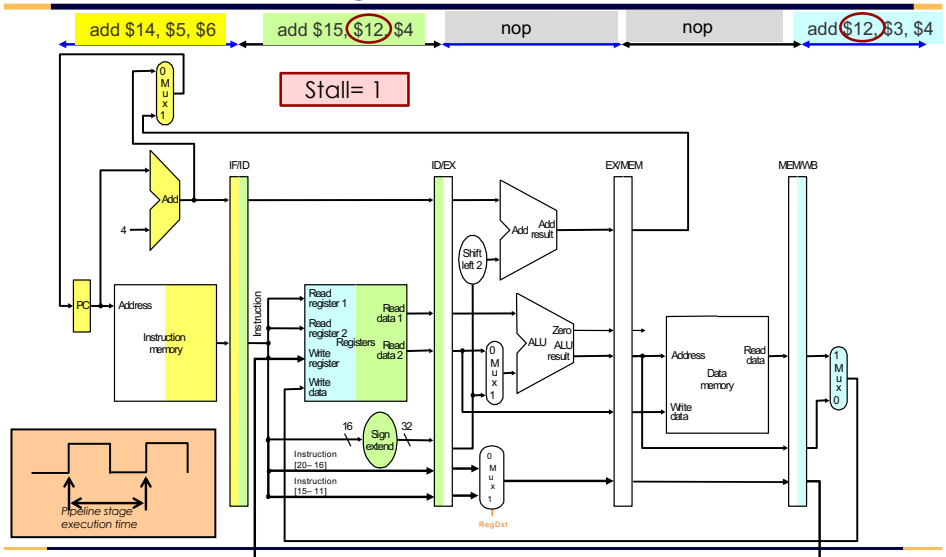
Implementing a Stall



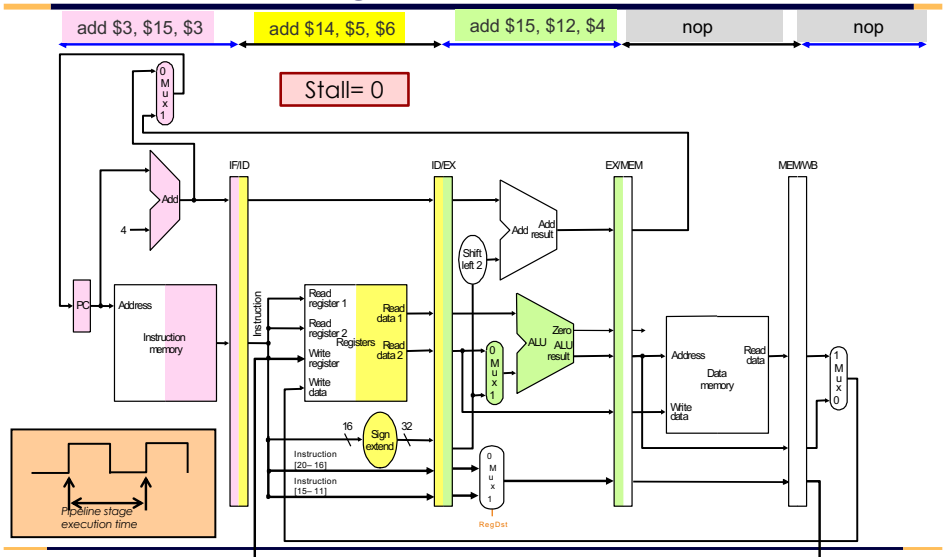
Implementing a Stall



Implementing a Stall



Implementing a Stall



Stall Condition Implementation

- **Combinational logic**
  - Special logic that checks if any instruction in later stages is supposed to write to any source register of the instruction that is being decoded ← Lab Assignment 1
  - Challenge?
    - Logic complexity increases as pipeline becomes deeper
    - Stall logic can increase critical path of pipeline stage, lowering overall frequency

- **Scoreboard**
  - 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

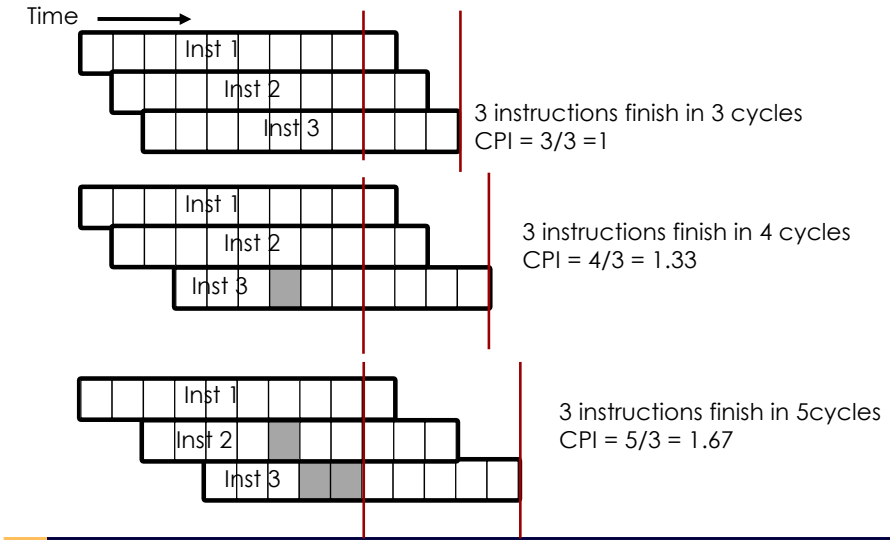
Impact of Stall on Performance

- Each stall cycle corresponds to one lost cycle in which no instruction can be completed

- For a program with N instructions and S stall cycles
  - Average CPI=(N+S)/N

- S depends on
  - frequency of RAW dependences
  - exact distance between the dependent instructions
  - distance between dependences

Pipeline CPI Examples



Hazards due to loads and stores

- Is there any possible data hazard in this instruction sequence?

```
SW r2, 7(r1)
LW r4, 5(r3)
```

What if (r1)+7 = (r3)+5 ?

  - This is not a register dependence
    - This is a potential address dependence

- Is this a problem in our 5-stage pipeline?
  - No – because we have assumed that our Mem stage finishes within one cycle.
  - Needs to be handled in more complex systems → ECE 6100 covers this