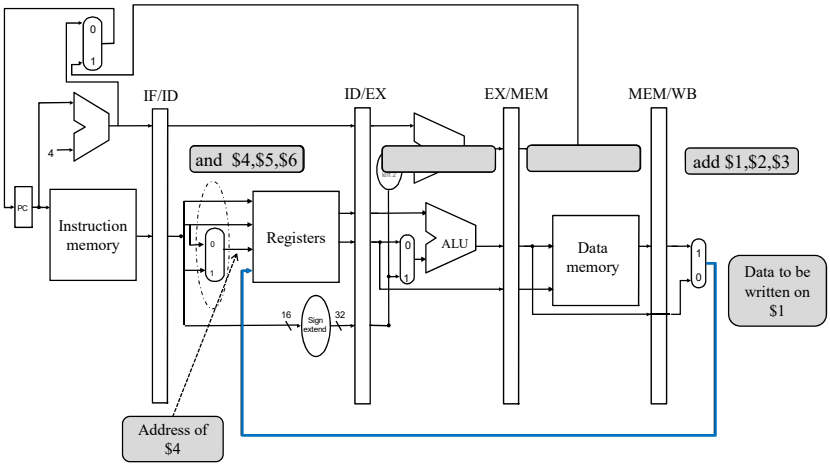
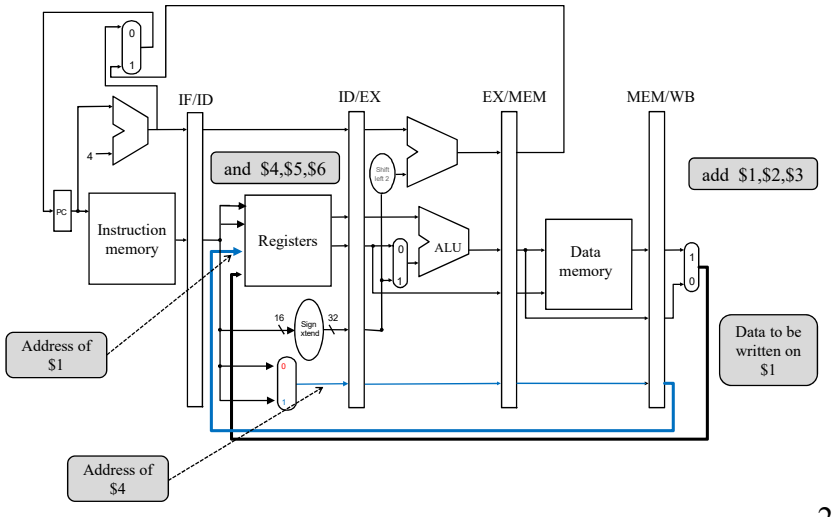
Reconsider the Pipelined datapath (slide 21)

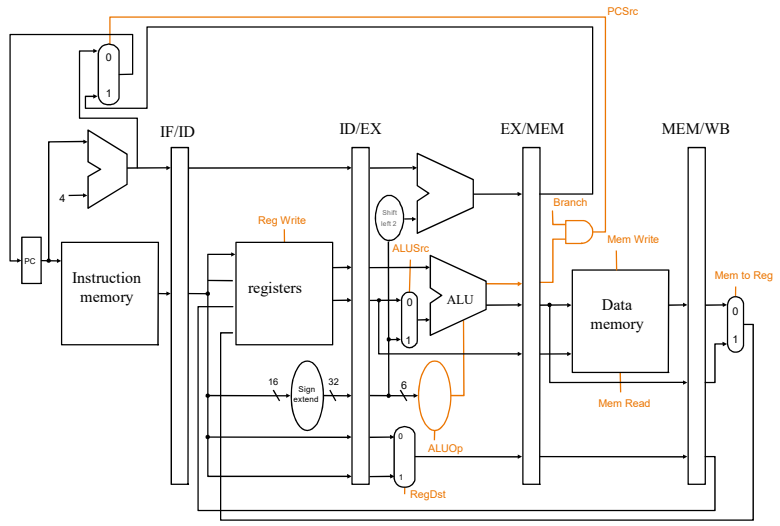


There is a problem with the above datapath!!

Corrected datapath



Pipeline Control

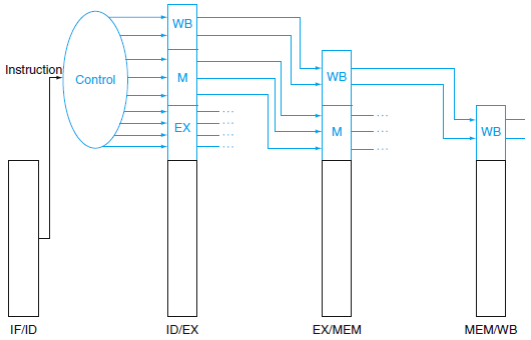


We have 5 stages, each having its own control signals

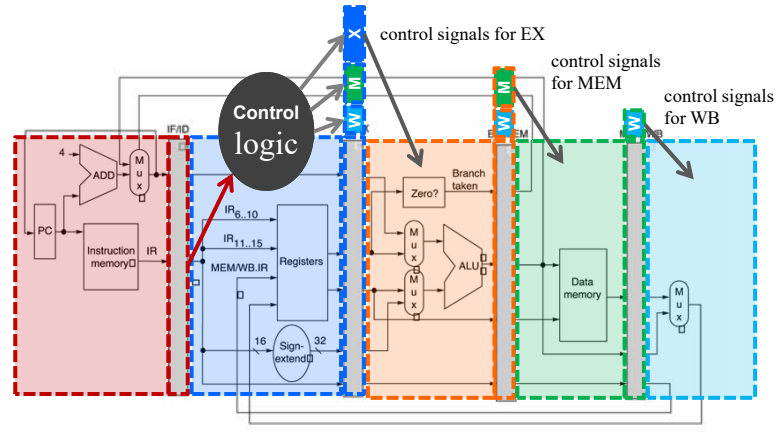
Pipeline Control

- **Pass control signals along just like the data**

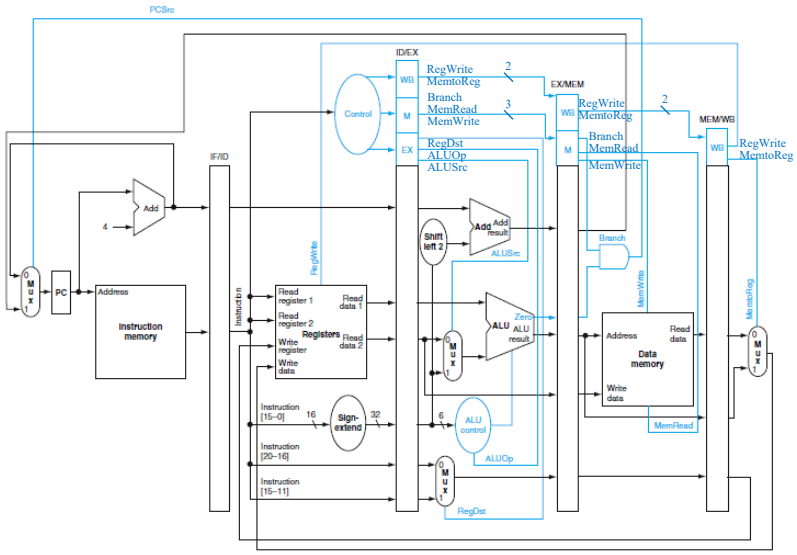
Instruction	Execution/Address Calculation stage control lines				Memory access stage control lines			Write-back stage control lines	
	Reg Dst	ALU Op1	ALU Op0	ALU Src	Branch	Mem Read	Mem Write	Reg write	Mem to Reg
R-format	1	1	0	0	0	0	0	1	0
lw	0	0	0	1	0	1	0	1	1
sw	X	0	0	1	0	0	1	0	X
beq	X	0	1	0	1	0	0	0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



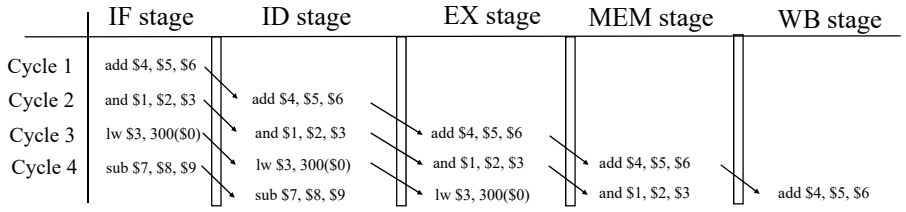
Pipeline control



Datapath with Control



representation of pipelining

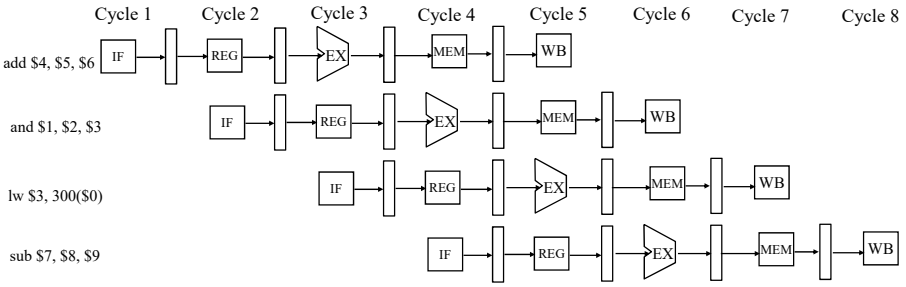


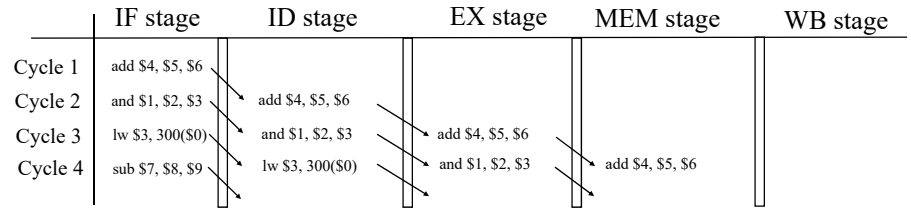
Can help with answering questions like:

- how many cycles does it take to execute this code?
- what is the ALU doing during cycle 4?
- At what cycle does a specific instruction exits the pipeline?

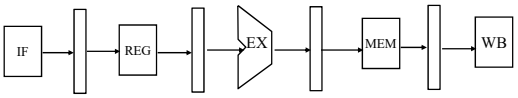
Pipeline throughput: Because the pipeline completes the execution of one instruction every clock cycle (after an initial delay to fill up the pipeline), then the CPI = 1.

Equivalent representations of pipelining





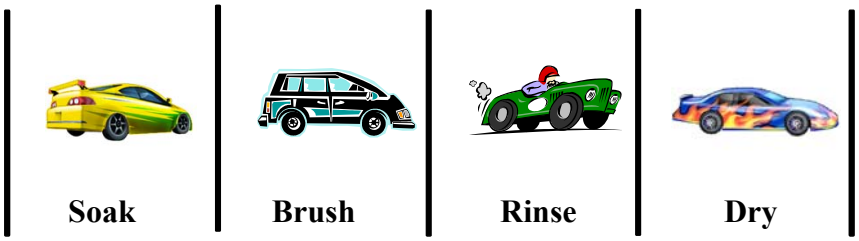
Pipeline hazards



- What makes pipelining easy for MIPS instructions
  - all instructions have the same length
  - just a few instruction formats
  - memory operands appear only in loads and stores
- What can go wrong?
  - structural hazards: would occur if we had only one memory
  - data hazards: may occur if an instruction depends on a previous instruction
  - control hazards: may occur when executing branch instructions
- We'll look at these hazards in the context of our simple pipeline

Structural hazards

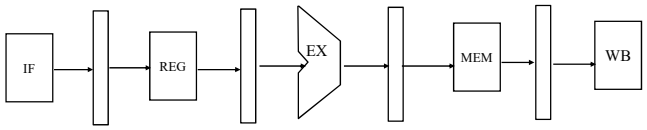
Example: imagine a pipelined car wash with one hose for both *soaking* and *rinsing*.



How do you solve the problem?
(share hardware, replicate hardware, compete for hardware)

Are there similar problems in our MIPS architecture?

Structural hazards in MIPS



Potential problem: Both IF and MEM use memory
Solution : use separate memories (caches)

Potential problem: Both REG and WB use the register file
Solution : Read from register file during the first half of a cycle and write to register file during the second half of a cycle.

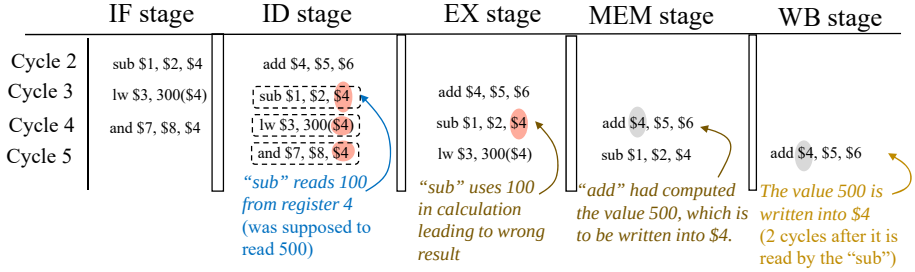
Data hazards (computing with wrong data)

Assume that registers 4, 5 and 6 contain the values 100, 200 and 300, respectively

Expected execution:

$4 contains 100 →	add $4, $5, $6	// write 500 into register 4
$4 should contain 500 →	sub $1, $2, $4	//reads the value 500 from register 4
$4 should contain 500 →	lw $3, 300($4)	// reads the value 500 from register 4
$4 should contain 500 →	and $7, $8, $4	// reads the value 500 from register 4

Pieplined execution:



What can be done to mitigate data hazards?

Data Hazards

- **Definition:** two instructions are data-dependent if the first writes into a register, R, and the other follows it and reads from R.
- To reduce the possibility of data hazards, writing into the register file should be performed in the first half of a cycle, and reading from the register file in the second half.
- If data-dependent instructions are not separated by at least two instructions, they will cause data hazards when executing on the pipeline.

Example:

```
add $3, $1, $2
lw  $1, 0($3)
add $2, $4, $3
lw  $4, $5, $1
sw  $2, 20($4)
add $2, $1, $1
```

Arrows show data dependence

A software solution:
The compiler can insert no-ops to separate data-dependent instructions by at least two instructions

```
add $3, $1, $2
no-op
no-op
lw  $1, 0($3)
add $2, $4, $3
no-op
lw  $4, $5, $1
sw  $2, 20($4)
add $2, $1, $1
```


A compiler optimization

The compiler can also reorder instructions to eliminate or reduce the number of inserted no-ops.

```
lw  $t1, 0($t0)
add $t1, $t1, $s1
sw  $t1, 0($t0)
lw  $t2, 4($t0)
add $t2, $t2, $s1
sw  $t2, 4($t0)
```

The order of two instructions can be changed if they are not data-dependent

```
lw  $t1, 0($t0)
lw  $t2, 4($t0)
add $t1, $t1, $s1
add $t2, $t2, $s1
sw  $t1, 0($t0)
sw  $t2, 4($t0)
```

```
lw  $t1, 0($t0)
lw  $t2, 4($t0)
no-op
add $t1, $t1, $s1
add $t2, $t2, $s1
no-op
sw  $t1, 0($t0)
sw  $t2, 4($t0)
```