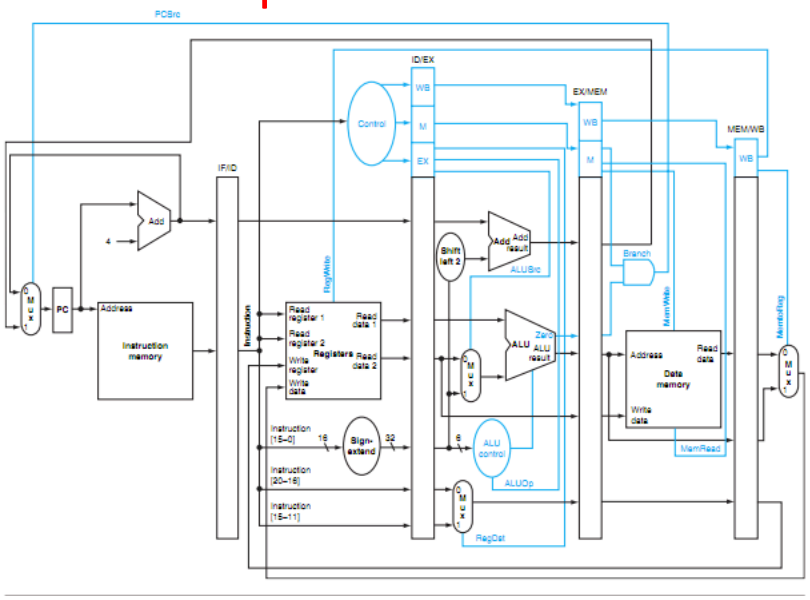
Pipelined instruction Execution

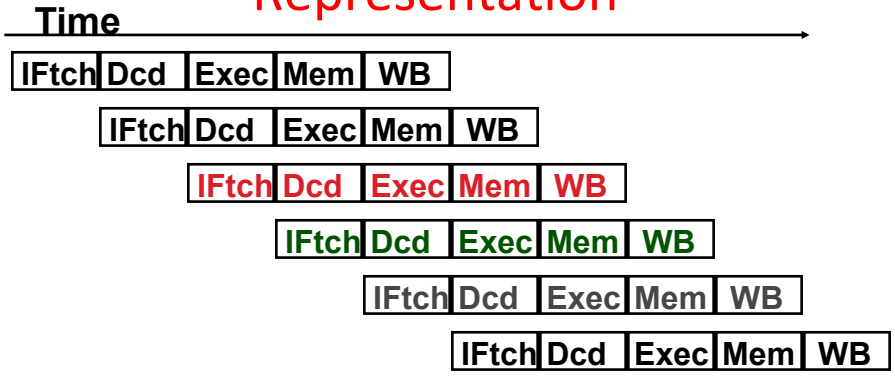
Pipelining and ISA Design

- MIPS Instruction Set designed for pipelining
- All instructions are 32-bits
  - Easier to fetch and decode in one cycle
  - x86: 1- to 17-byte instructions
  (x86 HW actually translates to internal RISC instructions!)
- Few and regular instruction formats, 2 source register fields always in same place
  - Can decode and read registers in one step
- Memory operands only in Loads and Stores
  - Can calculate address 3rd stage, access memory 4th stage
- Alignment of memory operands
  - Memory access takes only one cycle

Pipelined Control

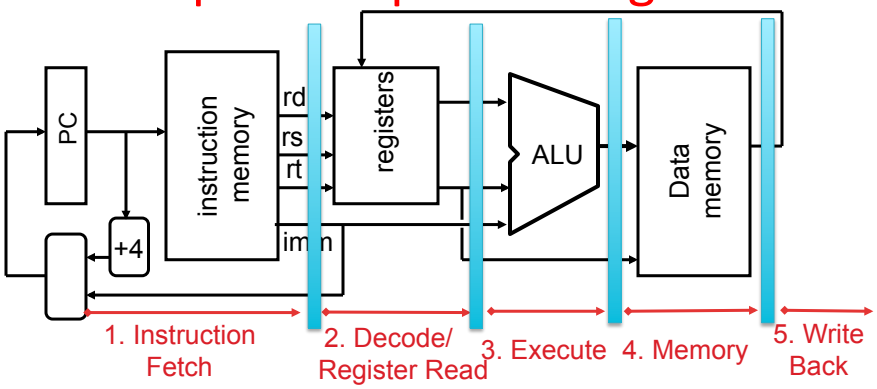


Pipelined Execution Representation

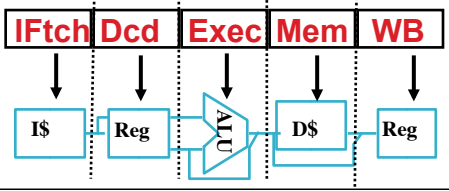


- Every instruction must take same number of steps, also called pipeline "stages", so some will go idle sometimes

Graphical Pipeline Diagrams

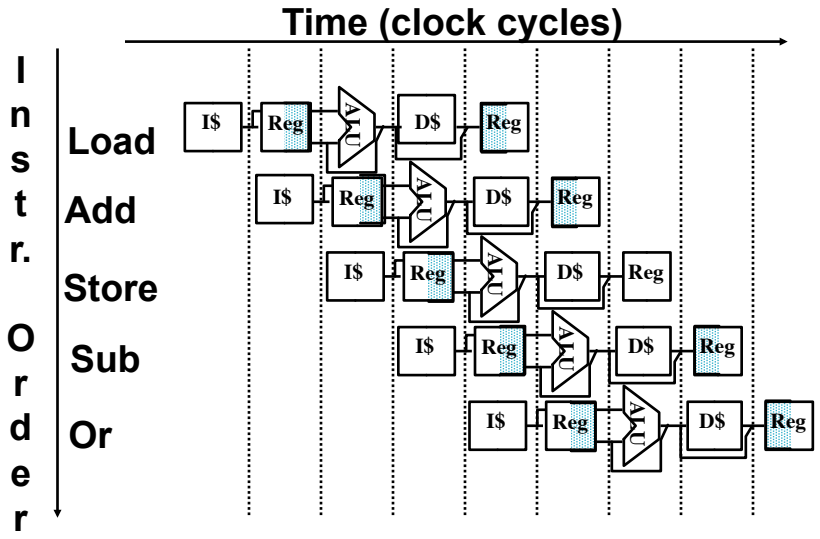


- Use datapath figure below to represent pipeline



Graphical Pipeline Representation

(In Reg, right half highlight read, left half write)

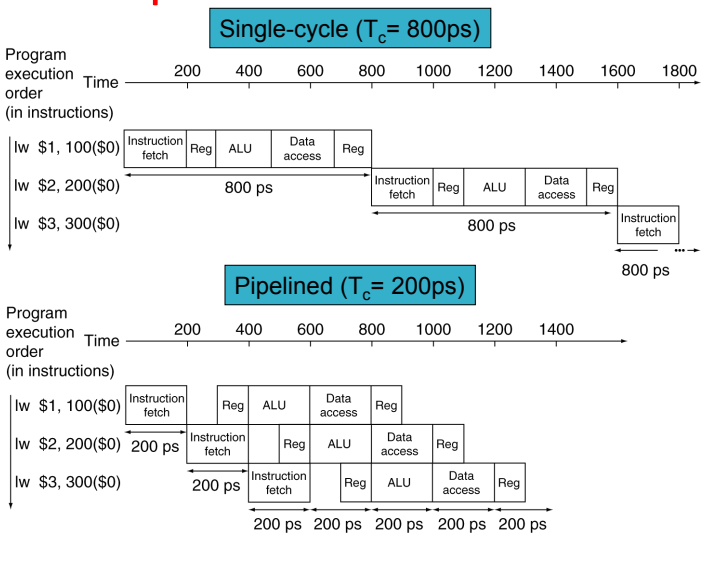


Pipeline Performance

- Assume time for stages is
  - 100ps for register read or write
  - 200ps for other stages
- What is pipelined clock rate?
  - Compare pipelined datapath with single-cycle datapath

Instr	Instr fetch	Register read	ALU op	Memory access	Register write	Total time
lw	200ps	100 ps	200ps	200ps	100 ps	800ps
sw	200ps	100 ps	200ps	200ps		700ps
R-format	200ps	100 ps	200ps		100 ps	600ps
beq	200ps	100 ps	200ps			500ps

Pipeline Performance



Pipeline Speedup

- If all stages are balanced
  - i.e., all take the same time
  - Time between instructions$_{pipelined}$ = $\frac{\text{Time between instructions}_{nonpipelined}}{\text{Number of stages}}$
- If not balanced, speedup is less
- Speedup due to increased throughput
  - Latency (time for each instruction) does not decrease

9

Hazards

Situations that prevent starting the next logical instruction in the next clock cycle

1. Structural hazards
   - Required resource is busy
2. Data hazard
   - Need to wait for previous instruction to complete its data read/write (e.g., pair of socks in different loads)
3. Control hazard
   - Deciding on control action depends on previous instruction (e.g., how much detergent based on how clean prior load turns out)

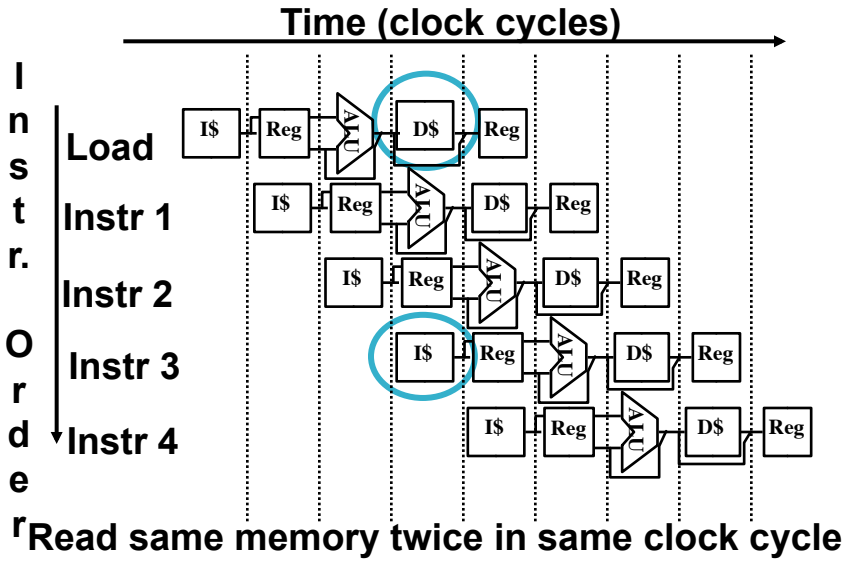
10

1. Structural Hazards

- Conflict for use of a resource
- In MIPS pipeline with a single memory
  - Load/Store requires memory access for data
  - Instruction fetch would have to *stall* for that cycle
    - Causes a pipeline "*bubble*"
- Hence, pipelined datapaths require separate instruction/data memories
  - In reality, provide separate L1 I$ and L1 D$

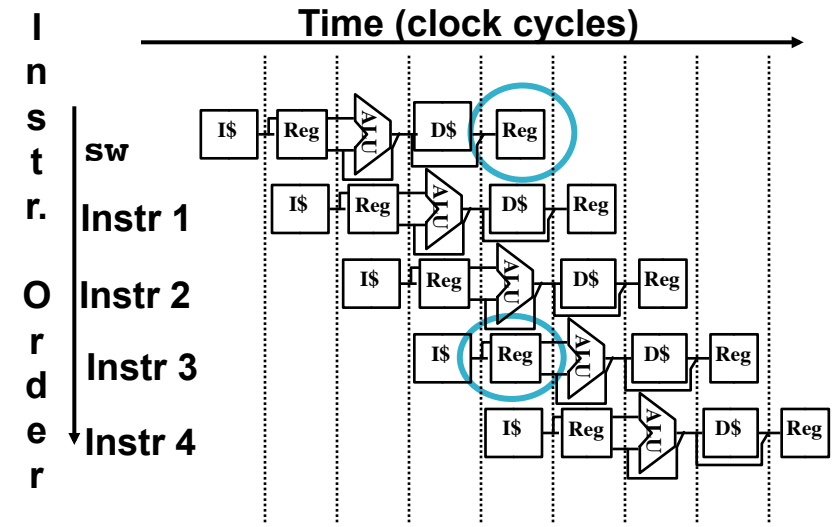
11

1. Structural Hazard #1: Single Memory



12

1. Structural Hazard #2: Registers (1/2)



Can we read and write to registers simultaneously?

13

1. Structural Hazard #2: Registers (2/2)

- Two different solutions have been used:
  1) RegFile access is *VERY* fast: takes less than half the time of ALU stage
     - Write to Registers during first half of each clock cycle
     - Read from Registers during second half of each clock cycle
  2) Build RegFile with independent read and write ports
- Result: can perform Read and Write during same clock cycle

14

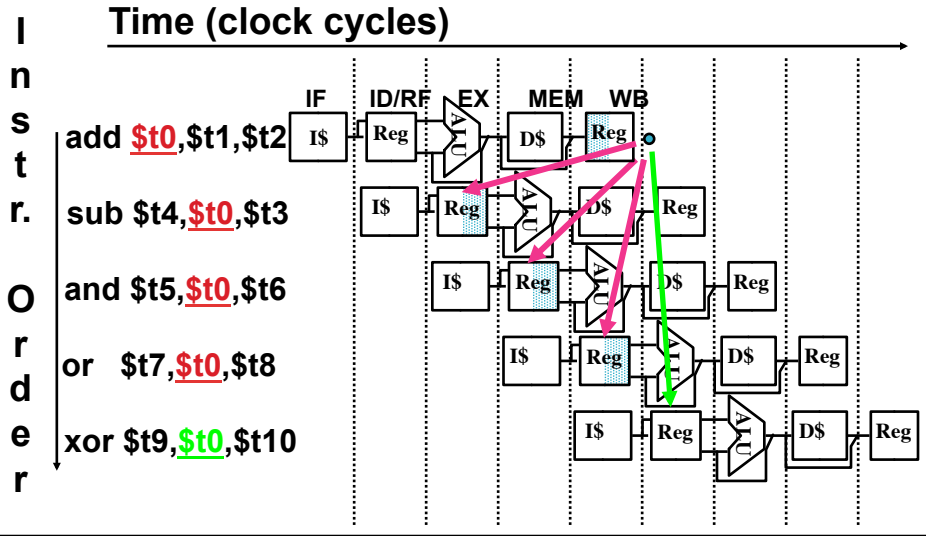
Data Hazards (1/2)

Consider the following sequence of instructions

```
add $t0, $t1, $t2
sub $t4, $t0 ,$t3
and $t5, $t0 ,$t6
or  $t7, $t0 ,$t8
xor $t9, $t0 ,$t10
```

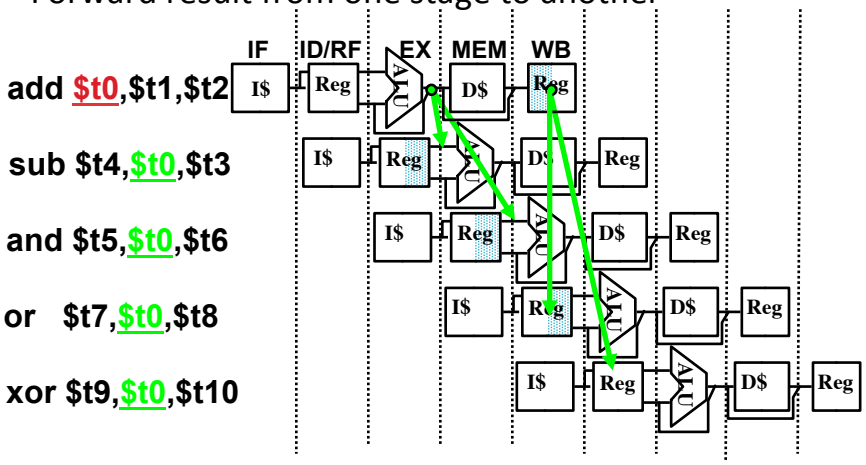
Data Hazards (2/2)

- Data-flow backward in time are hazards



Data Hazard Solution: Forwarding

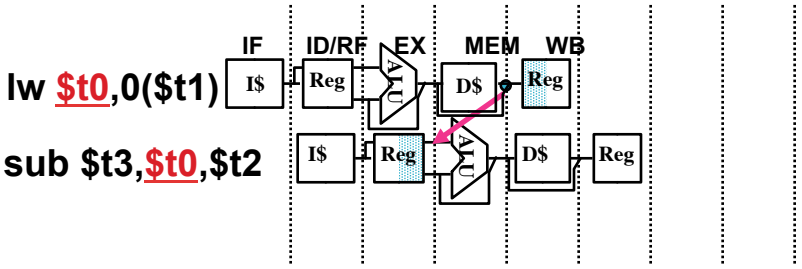
- Forward result from one stage to another



"or" hazard solved by register hardware

Data Hazard: Load/Use (1/4)

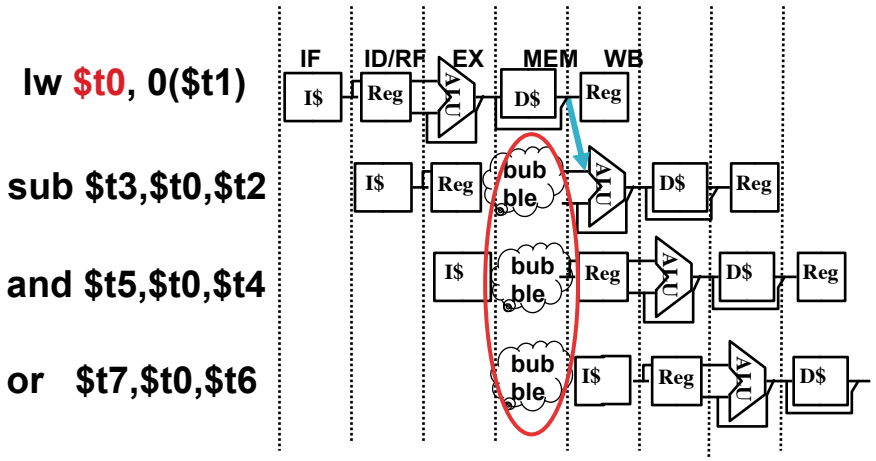
- Dataflow backwards in time are hazards



- Can't solve all cases with forwarding
- Must stall instruction dependent on load, then forward (more hardware)

Data Hazard: Load/Use (2/4)

Hardware stalls pipeline (Called "interlock")



Not in MIPS: (MIPS = Microprocessor without Interlocked Pipeline Stages)

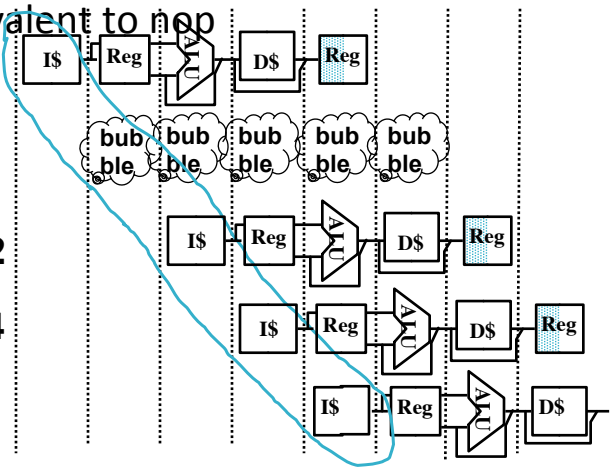
Data Hazard: Load/Use (3/4)

- Instruction slot after a load is called "load delay slot"
- If that instruction uses the result of the load, then the hardware interlock will stall it for one cycle.
- Alternative: If the compiler puts an unrelated instruction in that slot, then no stall
- Letting the hardware stall the instruction in the delay slot is equivalent to putting a nop in the slot (except the latter uses more code space)

Data Hazard: Load/Use (4/4)

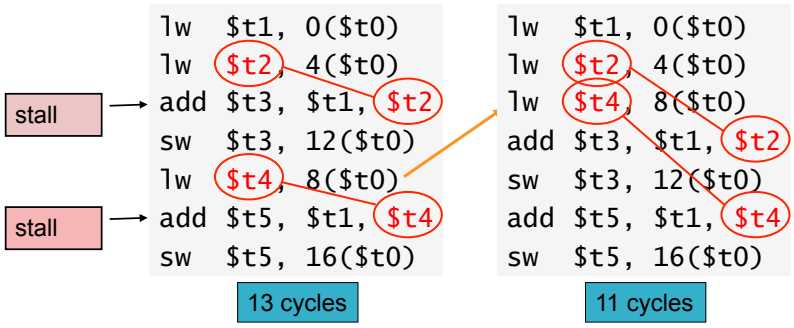
- Stall is equivalent to nop

```
lw $t0, 0($t1)
nop
sub $t3,$t0,$t2
and $t5,$t0,$t4
or  $t7,$t0,$t6
```



Data Hazards: Code Scheduling to Avoid Stalls

- Reorder code to avoid use of load result in the next instruction
- C code for `A = B + E; C = B + F;`



22

Data Hazards: Code Scheduling

	1	2	3	4	5	6	7	8	9	10	11	12	13
lw $t1, 0($t0)	IF	ID	EX	MEM	WB								
lw $t2, 4($t0)		IF	ID	EX	MEM	WB							
add $t3, $t1, **$t2**			IF		ID	EX	MEM	WB					
sw $t3,12($t0)					IF	ID	EX	MEM	WB				
lw $t4,8($t0)						IF	ID	EX	MEM	WB			
add $t5,$1, **$t4**							IF		ID	EX	MEM	WB	
sw$t5, 16($t0)									IF	ID	EX	MEM	WB
lw $t1, 0($t0)	IF	ID	EX	MEM	WB								
lw $t2, 4($t0)		IF	ID	EX	MEM	WB							
lw $t4,8($t0)			IF	ID	EX	MEM	WB						
add $t3, $t1, **$t2**				IF	ID	EX	MEM	WB					
sw $t3,12($t0)					IF	ID	EX	MEM	WB				
add $t5,$1, **$t4**						IF	ID	EX	MEM	WB			
sw$t5, 16($t0)							IF	ID	EX	MEM	WB		