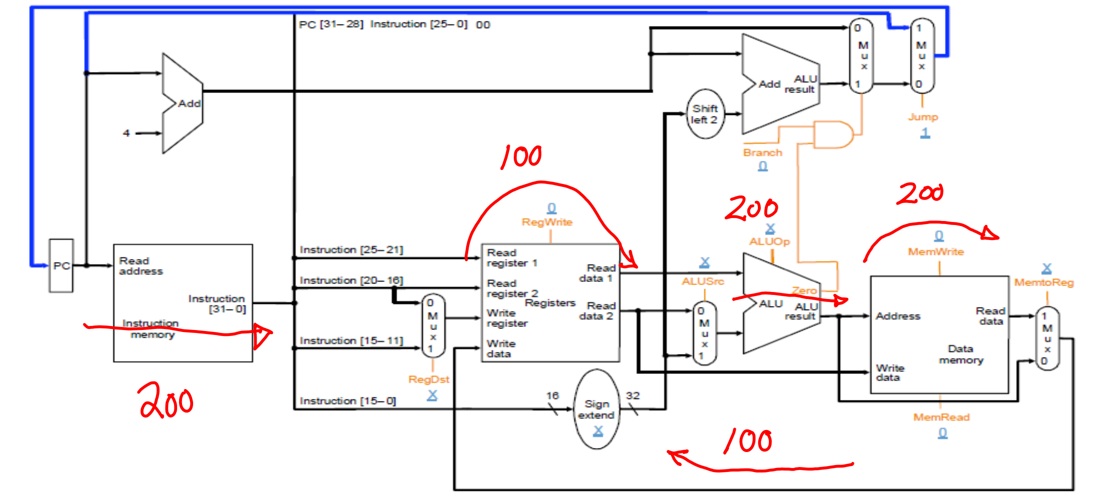
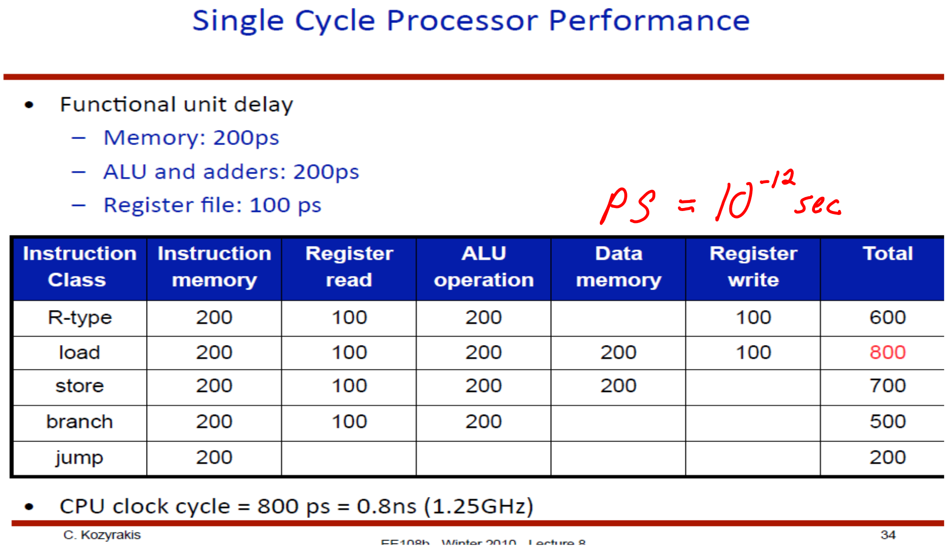
1. 1 cycle MIPS performance
3. general pipelining
8. MIPS pipe, LW
12. MIPS performance, piped vs non-piped
14. arrays, piped
15. hazards





Single Cycle Processor Performance

- Functional unit delay
  - Memory: 200ps
  - ALU and adders: 200ps
  - Register file: 100 ps

Instruction Class	Instruction memory	Register read	ALU operation	Data memory	Register write	Total
R-type	200	100	200		100	600
load	200	100	200	200	100	800
store	200	100	200	200		700
branch	200	100	200			500
jump	200					200

- CPU clock cycle = 800 ps = 0.8ns (1.25GHz)

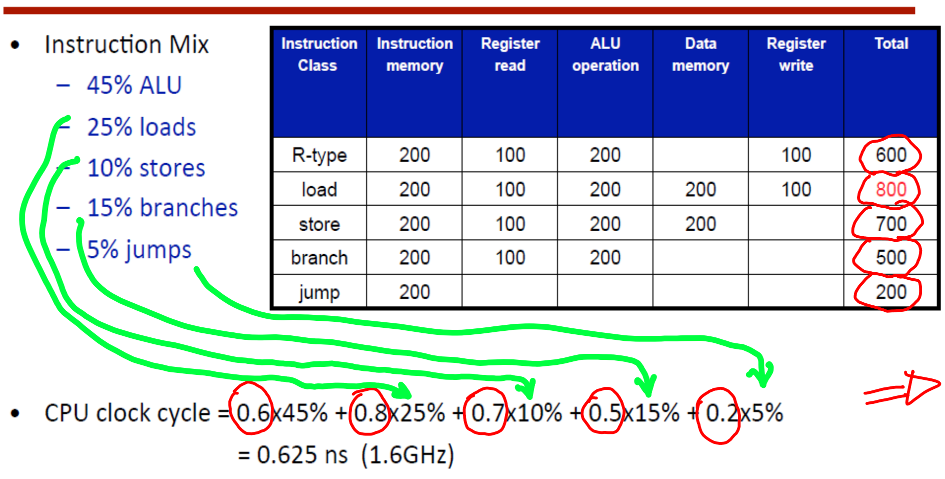
C. Kozyrakis EE108b Winter 2010 Lecture 8 34

$ps = 10^{-12}$ sec

max delay = T_{clock}

$1/T_{clock} = \frac{1}{0.8 ns} = 1.25 GHz$

what if we let clock trigger delay by opcode?



- Instruction Mix
  - 45% ALU
  - 25% loads
  - 10% stores
  - 15% branches
  - 5% jumps

Instruction Class	Instruction memory	Register read	ALU operation	Data memory	Register write	Total
R-type	200	100	200		100	600
load	200	100	200	200	100	800
store	200	100	200	200		700
branch	200	100	200			500
jump	200					200

- CPU clock cycle = 0.6x45% + 0.8x25% + 0.7x10% + 0.5x15% + 0.2x5%
  = 0.625 ns (1.6GHz)

ps → 0.6 ns
0.8
0.7
0.5
0.2

⇒ 1.6 GHz

what is speedup? $S_{new-old}$ =

what's the

- Problem?
  - Each functional unit used once per cycle
  - Most of the time it is sitting waiting for its turn
    - Well it is calculating all the time, but it is waiting for valid data
  - There is no parallelism in this arrangement

used

signal → functional unit

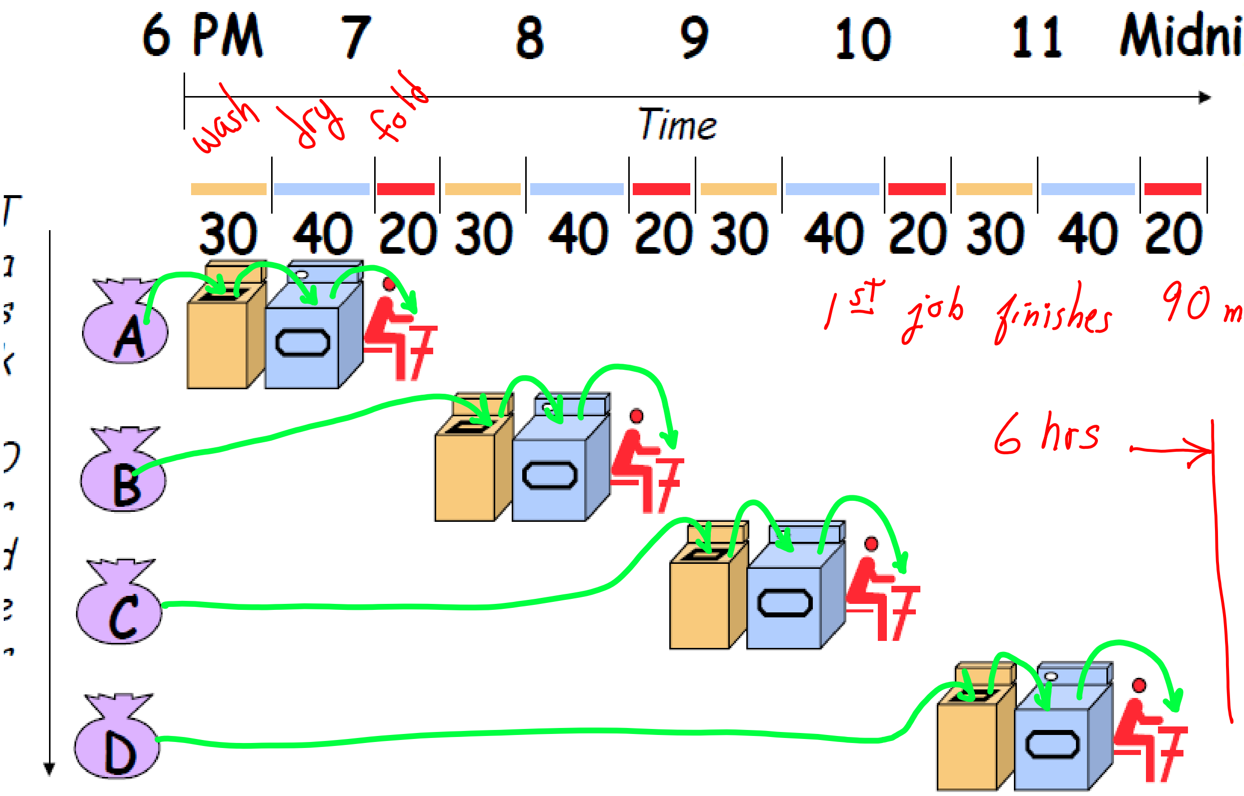
wait
do

- Making instructions take more cycles can make machine faster!?!
  - Each instruction takes roughly the same time
    - While the CPI is much worse, the clock freq is much higher
  - Overlap execution of multiple instructions at the same time
    - Different instructions will be active at the same time
  - This is called "Pipelining"
  - We will look at a 5 stage pipeline
    - Modern machines (Core 2) have order 15 cycles/instruction

CPI ↑ CR ↑

$$Perf = \frac{n}{T} = \frac{n}{n\ CPI\ (1/CR)} = \uparrow CR / CPI \uparrow$$

Sequential Laundry



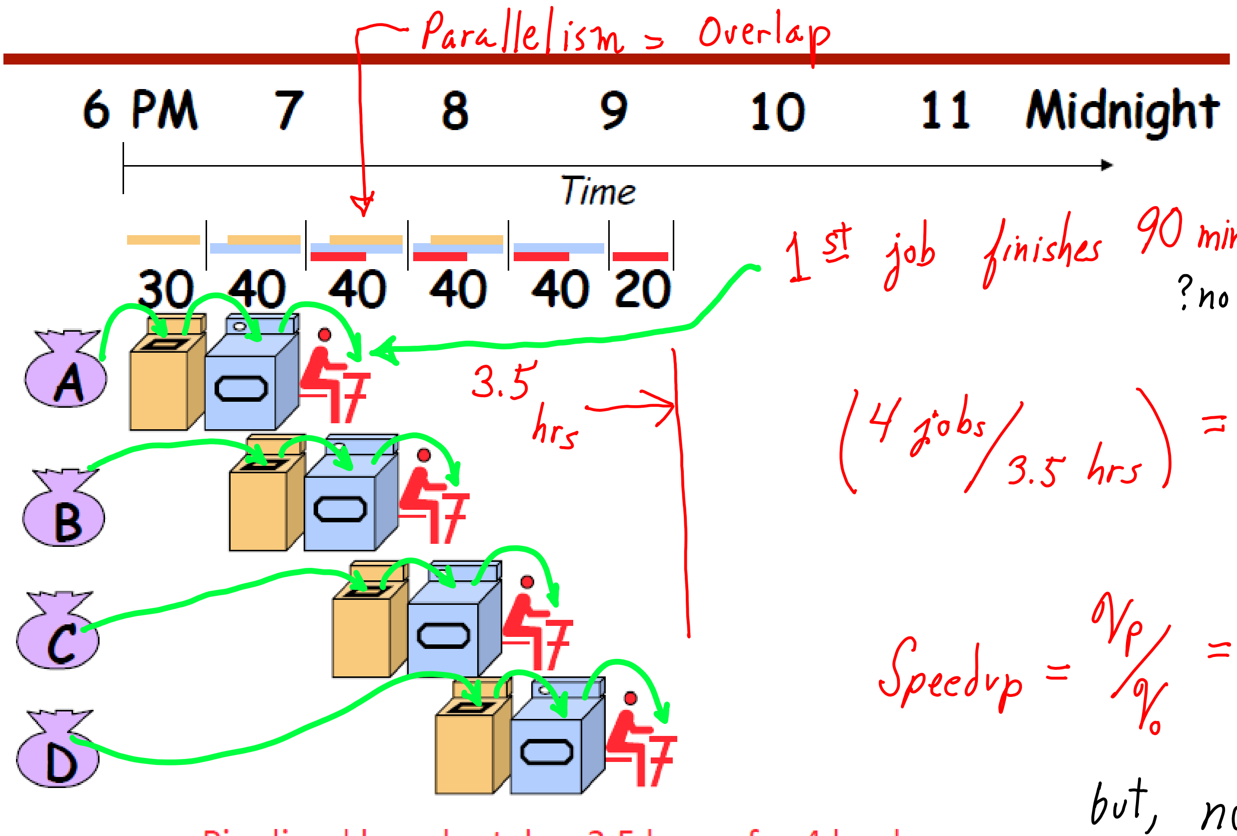
latency

1st job finishes 90 mins = 1.5 hrs

6 hrs → $\left(\frac{4 \text{ jobs}}{6 \text{ hrs}}\right)$ = %

Throughput

Sequential laundry takes 6 hours for 4 loads



1st job finishes 90 mins = 1.5 hrs

latency

? no improvement ?

Throughput

$\left(4 \text{ jobs} / 3.5 \text{ hrs}\right) = 1/p$

$\text{Speedup} = \frac{1/p}{1/o} = \frac{6}{3.5} \approx 2$

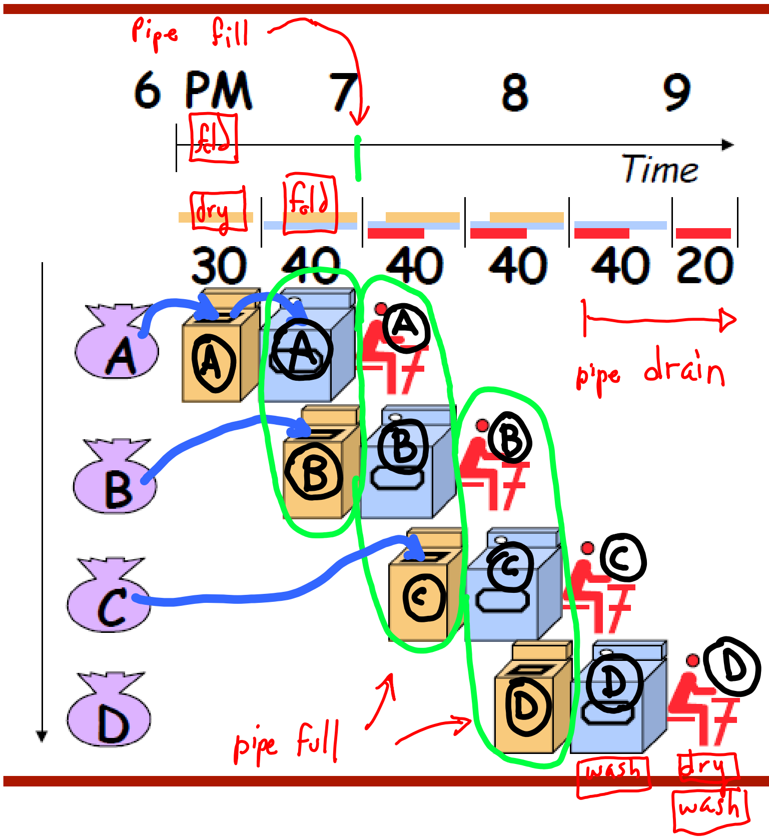
but, no CR↑

Pipelined laundry takes 3.5 hours for 4 loads

What does Amdahl's Law say?

Pipelining Lessons

throughput ↑

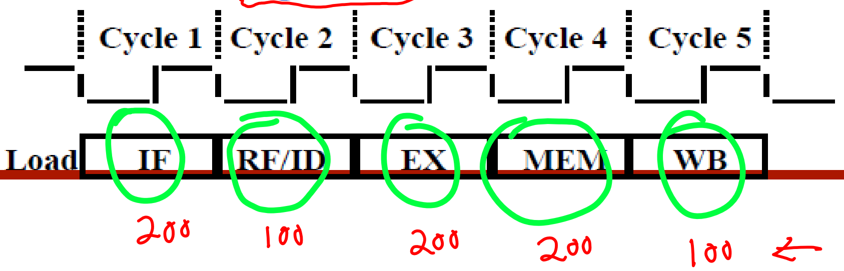


- Pipelining doesn't help latency of single task, it helps throughput of entire workload
- Multiple tasks operating simultaneously
- ~~Potential~~ max speedup ≤ Number pipe stages
- Pipeline rate limited by slowest pipeline stage
- Unbalanced lengths of pipe stages reduces speedup
- Time to "fill" pipeline and time to "drain" it reduces speedup

- IF: Instruction Fetch
  - Fetch the instruction from memory
  - Increment the PC
- RF/ID: Register Fetch and Instruction Decode
  - Fetch base register
- EX: Execute
  - calc. address
  - Calculate base + sign-extended offset
- MEM: Memory
  - Read the data from the data memory
- WB: Write back
  - Write the results back to the register file

lw (slowest instr.)

Pipe stages



← delays, ps

$(1/_{CR}) \geq 200_{ps}$

Pipelining Load lw

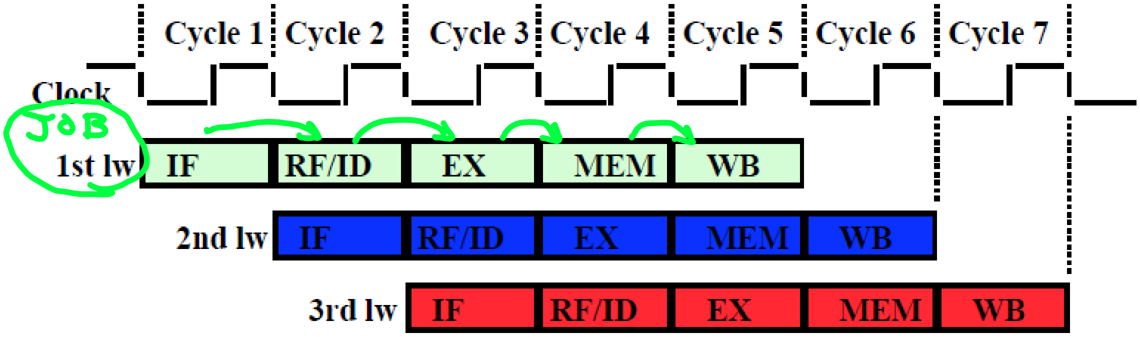
- Load instruction takes 5 stages
  - Five independent functional units work on each stage
    - Each functional unit used only once
  - Another load can start as soon as 1st finishes IF stage
  - Each load still takes 5 cycles to complete
  - The *throughput*, however, is much higher

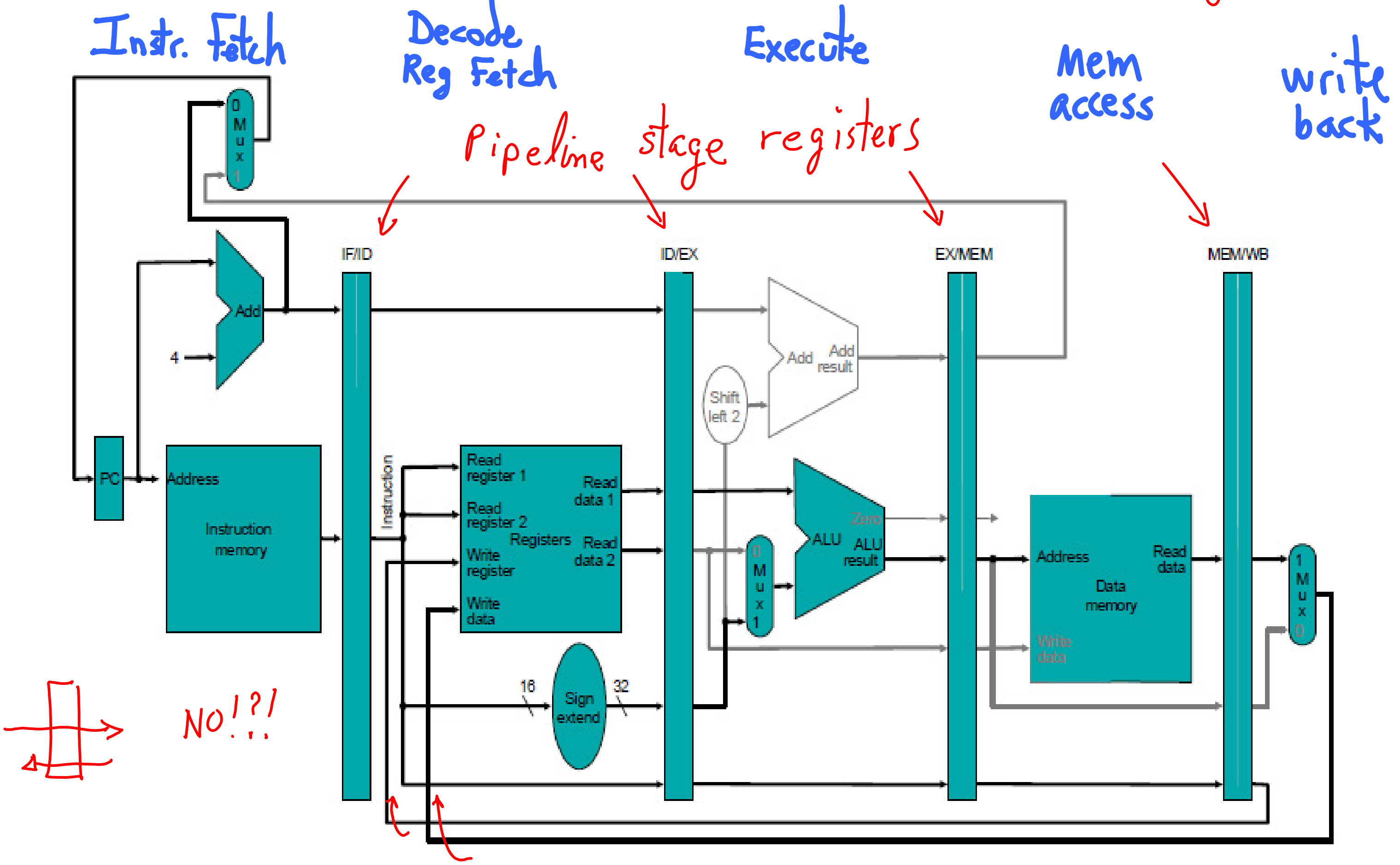
each stage busy

1 job exits per cycle

⇒ CPI = $\frac{1 \text{ job}}{1 \text{ cycle}}$

T_{latency} = 5 cycles

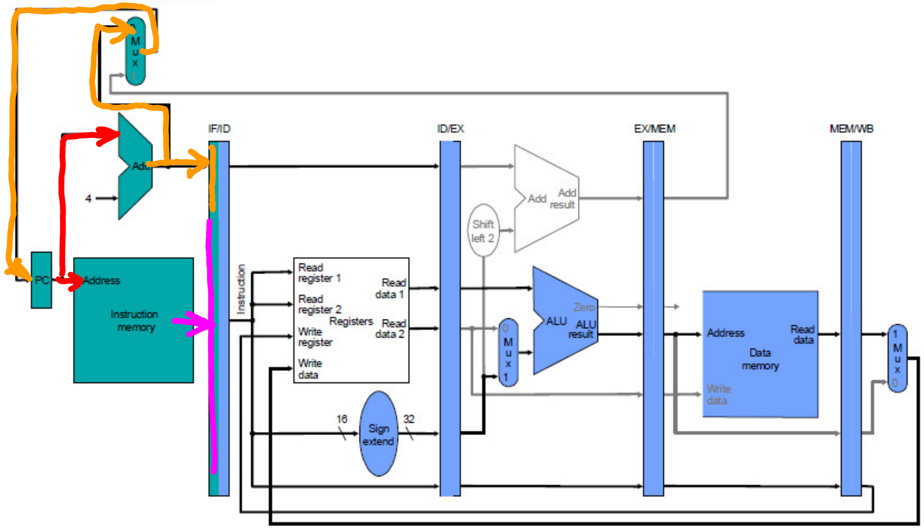




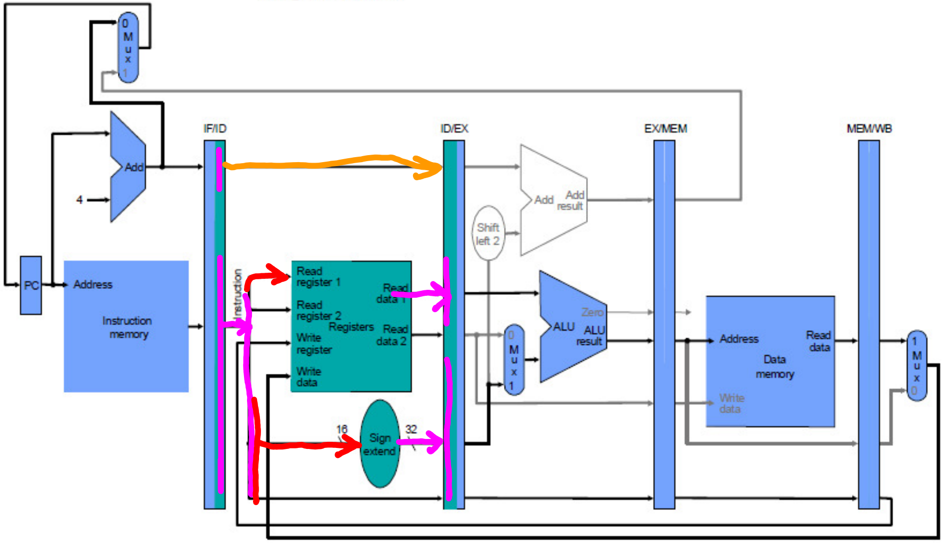
op $4, $1, $5 ⊗ ⊗ ⊗ op $1, $2, $3

required delay before using written data.

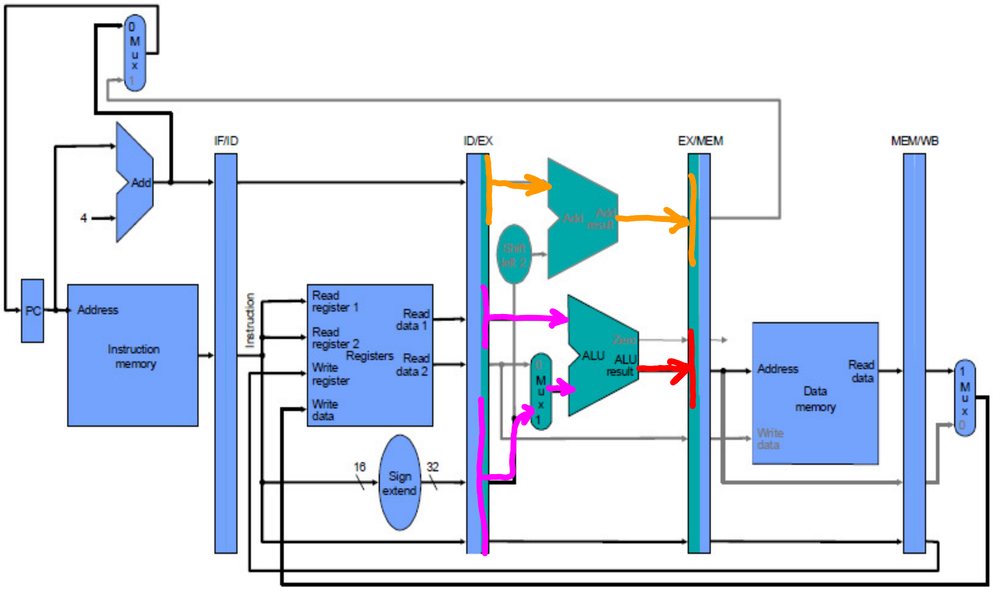
lw
Instruction Fetch



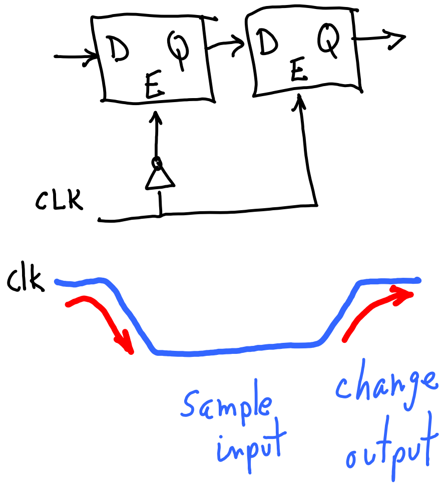
lw
Register Fetch

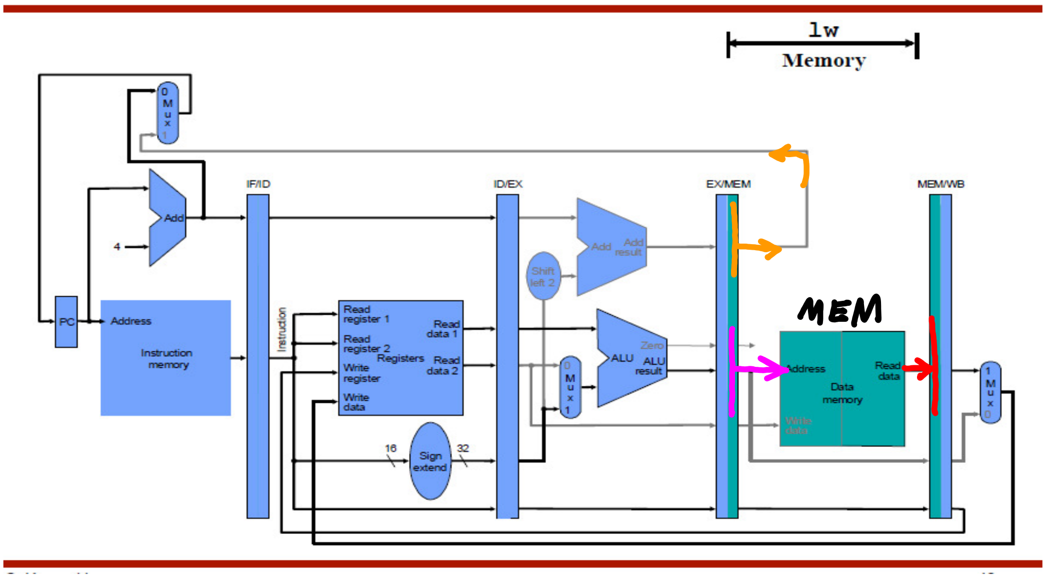


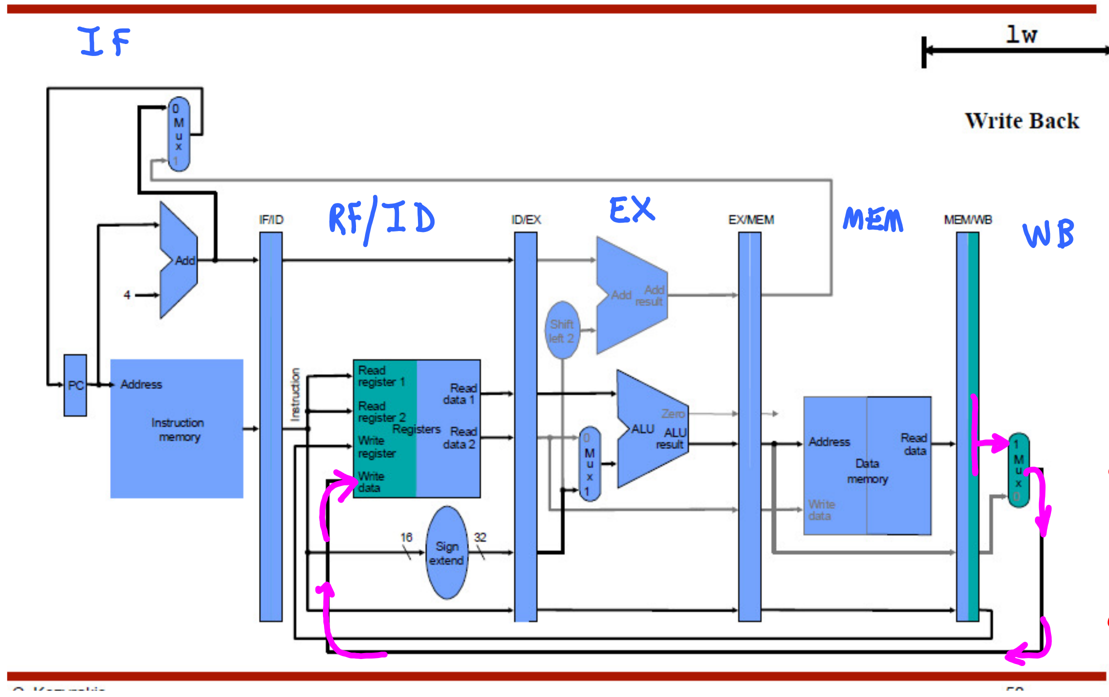
lw
Execute

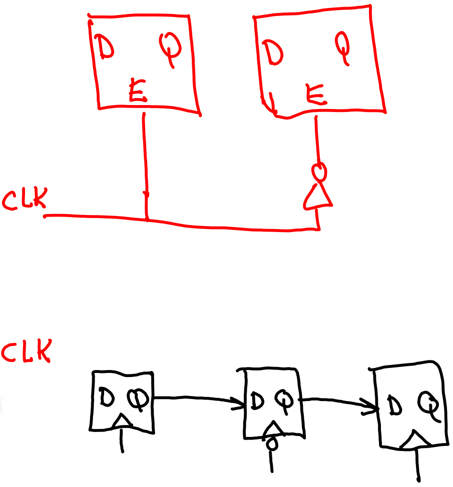


Positive edge-triggered FF:
output changes on rising clock



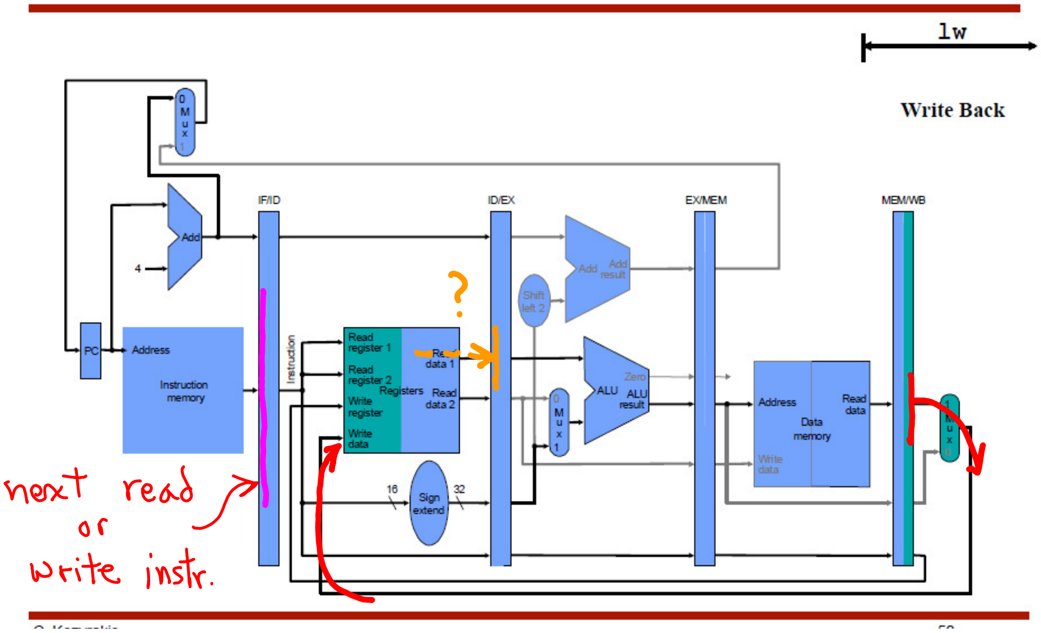




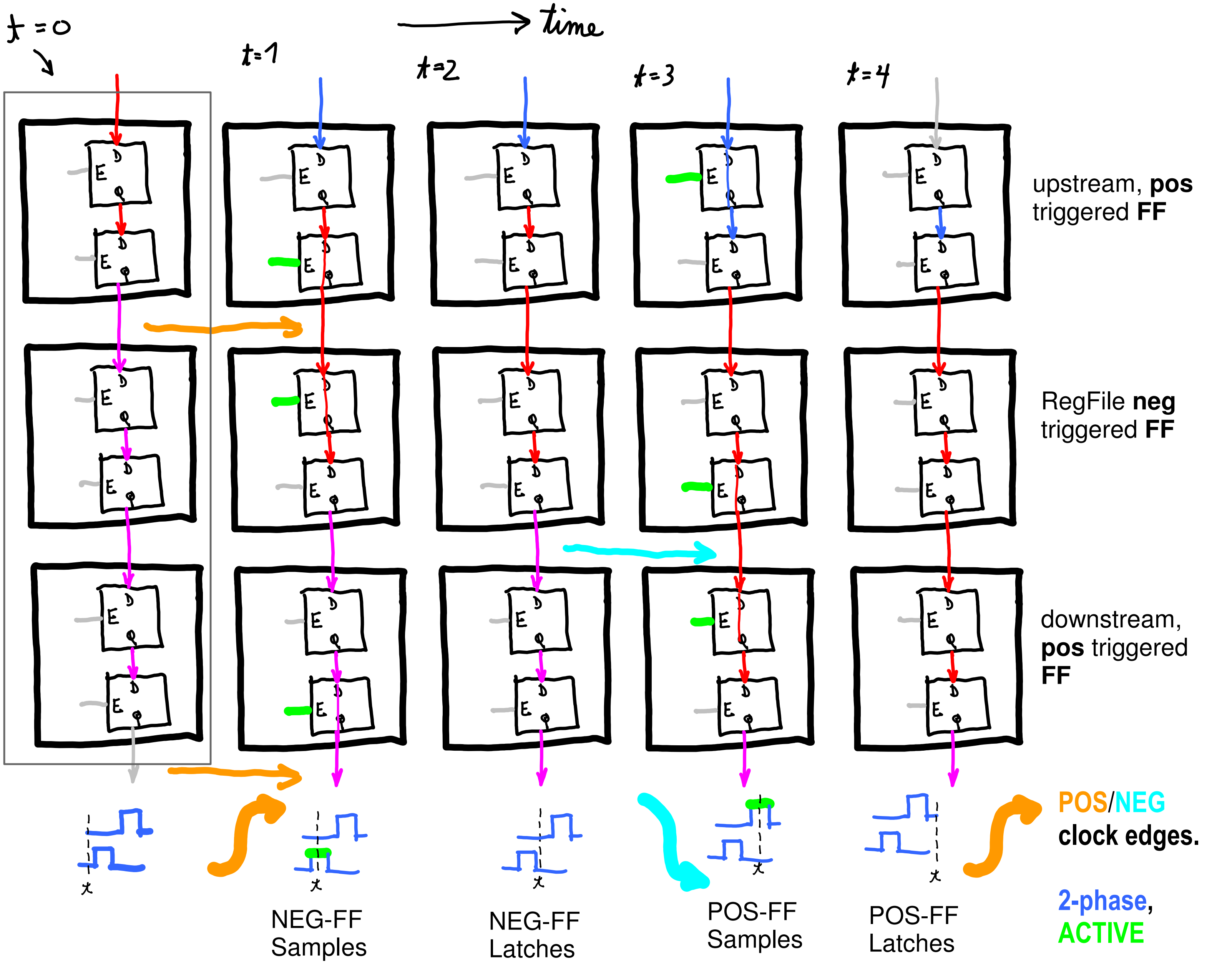


- Use a Main Control unit to generate signals during RF/ID Stage
  - Control signals for EX
    - (ExtOp, ALUSrc, ...) used 1 cycle later
  - Control signals for Mem
    - (MemWr, Branch) used 2 cycles later
  - Control signals for WB
    - (MemtoReg, MemWr) used 3 cycles later

use pipe regs for control signals; could also pass along OP field, decode as needed



lw $1, ...
nop
nop
nop
add $2, $1, ...



upstream, **pos** triggered **FF**

RegFile **neg** triggered **FF**

downstream, **pos** triggered **FF**

NEG-FF Samples

NEG-FF Latches

POS-FF Samples

POS-FF Latches

POS/NEG clock edges.

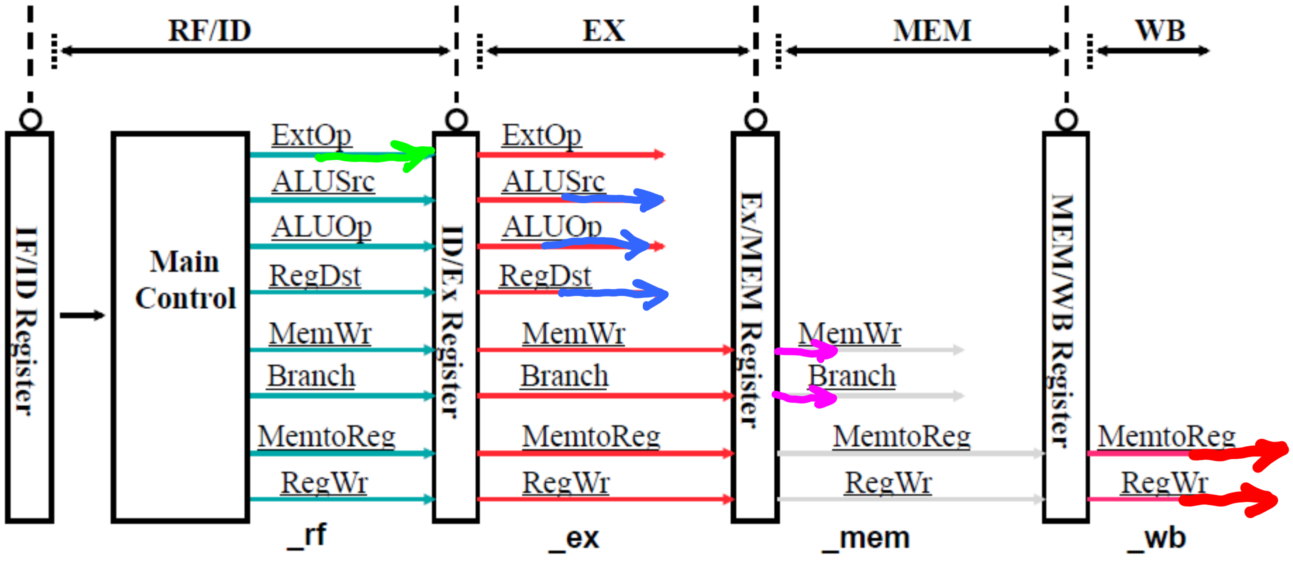
2-phase, ACTIVE

→ new data X into RegFile

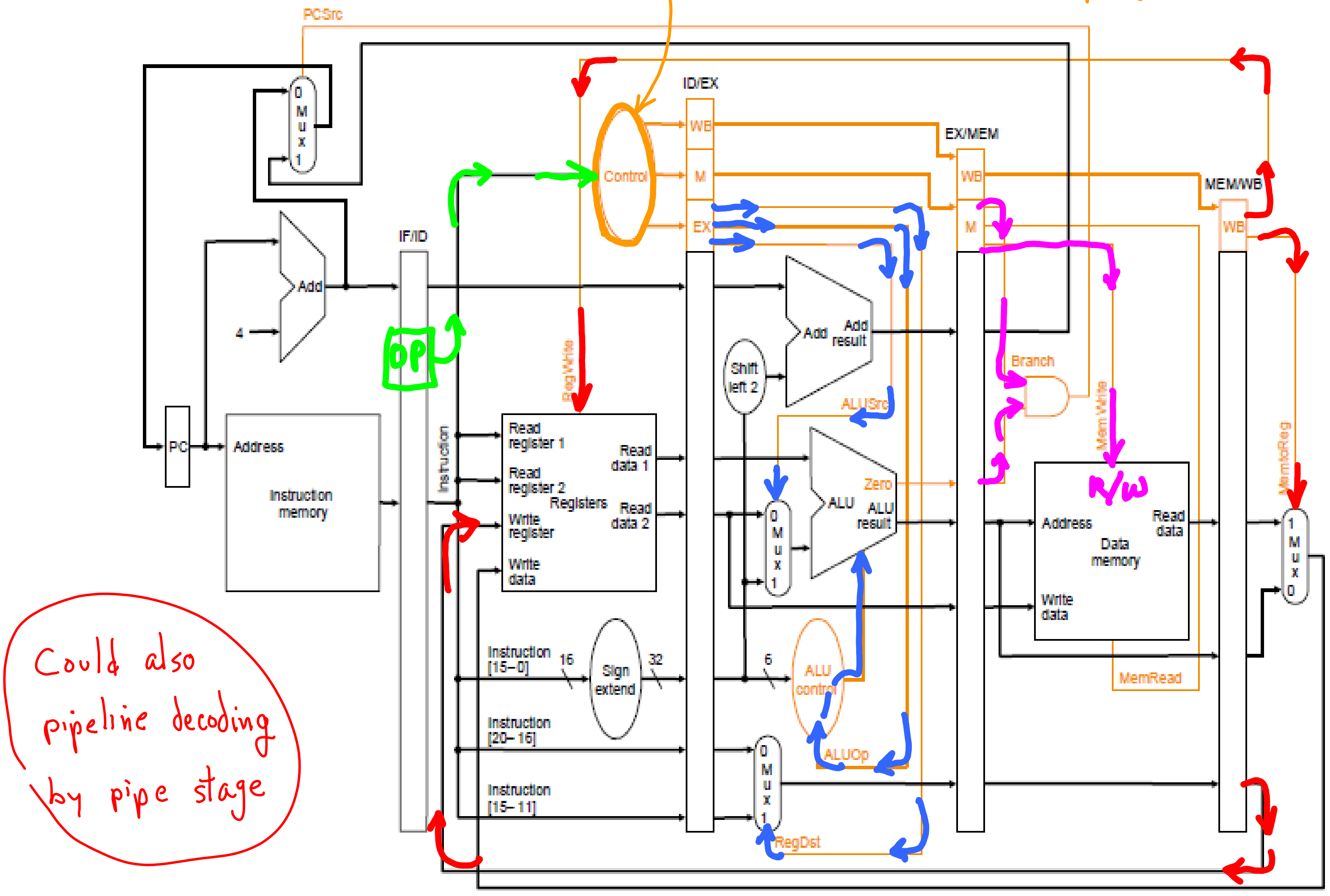
X is output by RegFile

next stage outputs X

Implementing Control



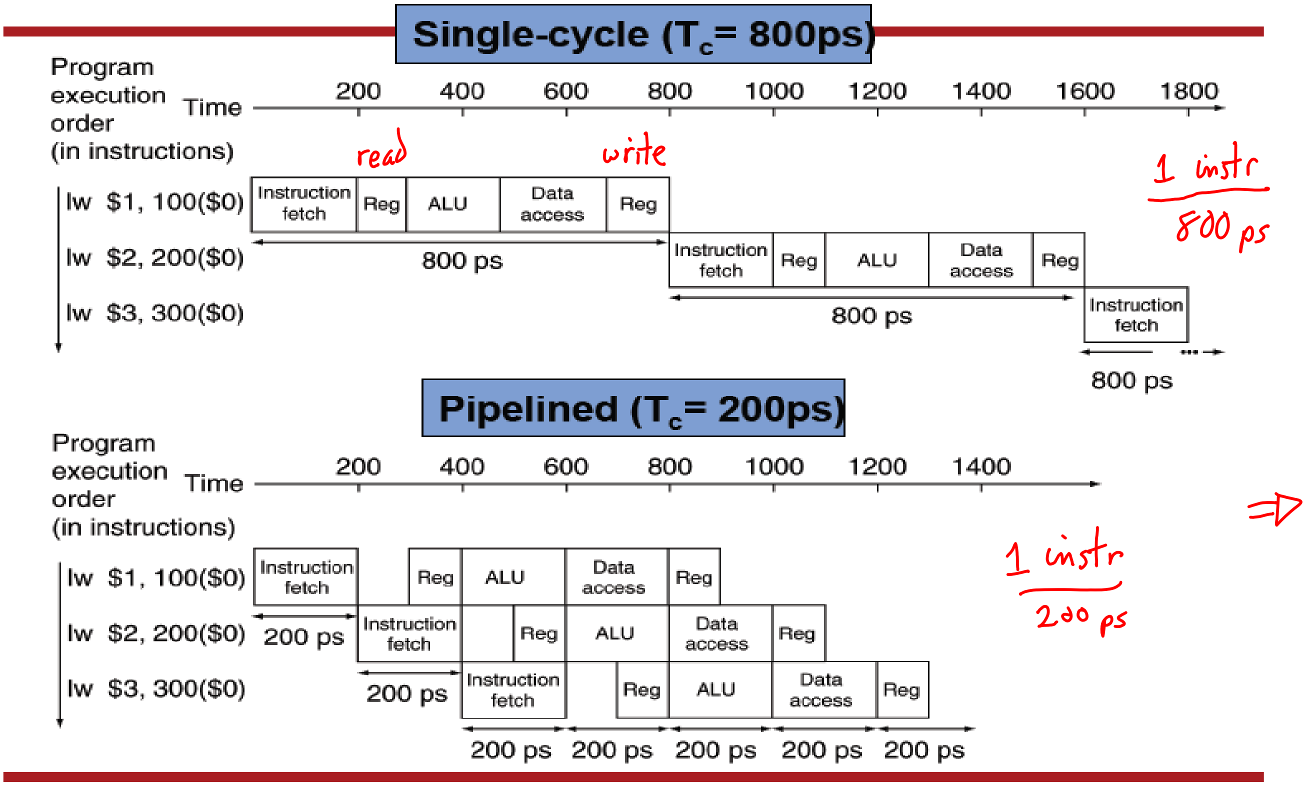
Decoding: combinational / μCode control signals (table lookup by opcode)



Could also pipeline decoding by pipe stage

- Assume time for stages is
  - 100ps for register read or write
  - 200ps for other stages

Pipeline Performance



1 instr / 800 ps

1 instr / 200 ps

⇒ S = 4

- MIPS ISA designed for pipelining
  - All instructions are 32-bits
    - Easier to fetch and decode in one cycle
    - c.f. x86: 1- to 17-byte instructions
  - Few and regular instruction formats
    - Can decode and read registers in one step
  - Load/store addressing
    - Can calculate address in 3rd stage, access memory in 4th stage
  - Alignment of memory operands
    - Memory access takes only one cycle

LDI

orthogonality

vs. 2 for misaligned ⇒ stall

But Something Is Fishy Here

- If dividing it into 5 parts made the clock faster 800 ps → 200 ps clock
  - And the effective CPI is still one if...
- Then dividing it into 10 parts would make the clock even faster 800 ps → 100 ps
  - And wouldn't the CPI still be one?
- Then why not go to twenty cycles?
- Really two issues
  - Some things really have to complete in a cycle ← cannot divide every operation
    - Find next PC from current PC
  - CPI is not really one
    - Sometimes you need the results a previous instruction that is not done

↑ the longer the pipeline, the more bubbles

⇒ CPI ↑

Can Pipelining Lead to an Arbitrary Short Clock Cycle?

- Min clock cycle = longest combinatorial delay + FF setup + clock skew

- Pipelining reduces the combinatorial delay
  - Less work per pipeline stage
  - Ideally, N stages reduce delay to 1/N
  - Best you can achieve is Clock cycle > FF setup + clock skew
    - Diminishing returns from ever longer pipelines…

- Imbalance between stages also reduces benefits from subdividing

- Even if you could continuously improve clock frequency
  - ↑ Power consumption ∞ Frequncy ↑

Pipelining & CPI

Dependencies and Hazards

- Hazards: situations that prevent starting the next instruction in the next cycle
  - Wasted cycles, CPI>1
- Hazards are due to dependencies between instructions
  - Two instructions share resources or data
  - Pipelining may lead to overlapping their execution
- Types of hazards
  - Structural Hazard (resource conflict)
    - Two instructions need to use the same piece of hardware
  - Data Hazard
    - Instruction depends on result of instruction still in the pipeline
  - Control Hazard
    - Instruction fetch depends on the result of instruction in pipeline

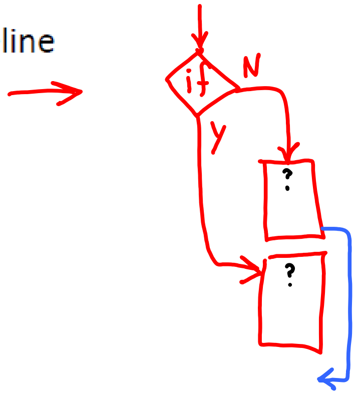
BR

LC3's LDI

2 refs to data memory

⇒ can't send

lw, lw, e.g.

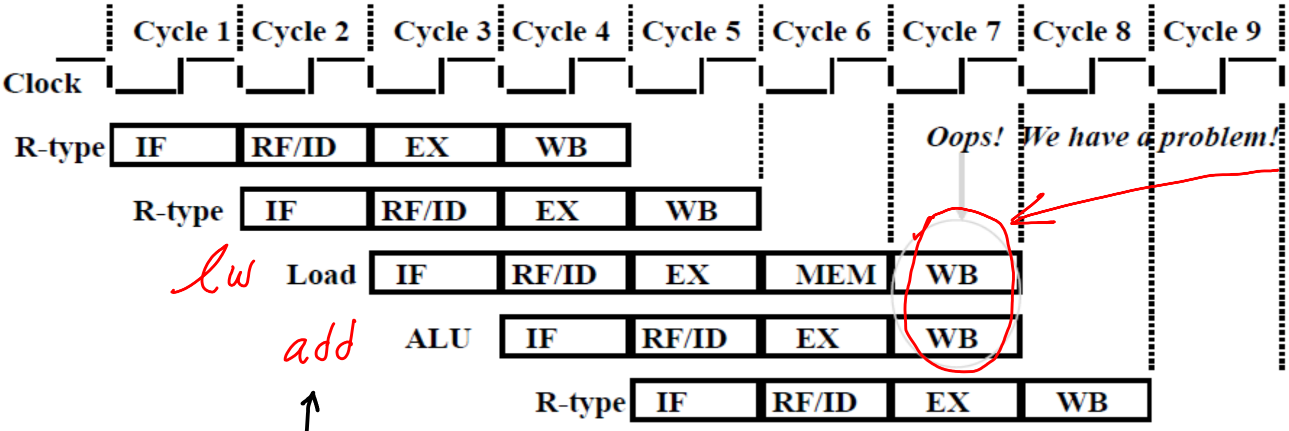


- Simple example: MIPS pipeline with a single unified memory
  - No separate instruction & data memories
  - Load/store requires data access
  - Instruction fetch would have to *stall* for that cycle
    - Would cause a pipeline "bubble"
  - Also used for units that are not fully pipelined (mult, div)

STRUCTURAL HAZARD

- Consider a load followed immediately by an ALU operation
  - Register file only has a single write port
  - But need to write the results of the ALU and the memory back

STRUCTURAL HAZARD

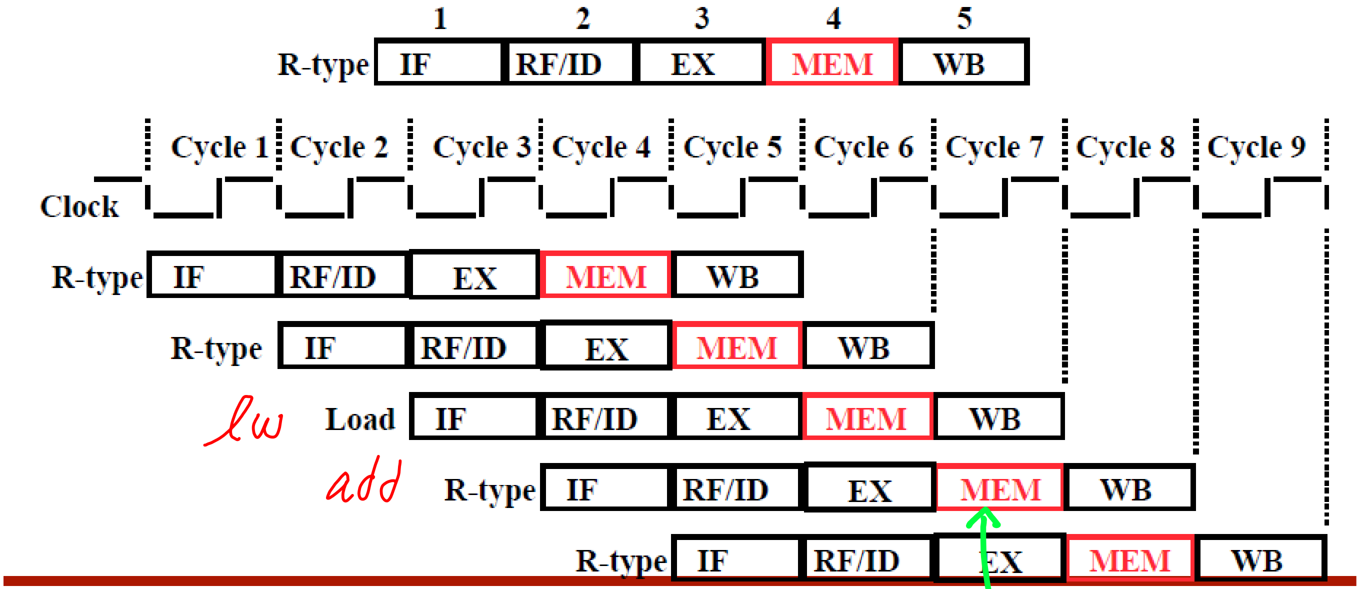


	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 6	Cycle 7	Cycle 8	Cycle 9
R-type	IF	RF/ID	EX	WB					
R-type		IF	RF/ID	EX	WB				
lw Load			IF	RF/ID	EX	MEM	WB		
add ALU				IF	RF/ID	EX	WB		
R-type					IF	RF/ID	EX	WB	

2 writes to Regfile at same time?

short-circuited execution: does WB in MEM instead of in WB

- Delay R-type register write by one cycle → don't short-circuit
  - Does this increase the CPI of instruction? → what was CPI above?
  - What is the cost?



	1	2	3	4	5
R-type	IF	RF/ID	EX	MEM	WB

	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 6	Cycle 7	Cycle 8	Cycle 9
R-type	IF	RF/ID	EX	MEM	WB				
R-type		IF	RF/ID	EX	MEM	WB			
lw Load			IF	RF/ID	EX	MEM	WB		
add R-type				IF	RF/ID	EX	MEM	WB	
R-type					IF	RF/ID	EX	MEM	WB

delay writing

Data Dependencies

sequential consistency

- Data dependencies for instruction *j* following instruction *i*
  - Read after Write (RAW) (true dependence)
    - Instruction *j* tries to read before instruction *i* tries to write it
  - Write after Write (WAW) (output dependence)
    - Instruction *j* tries to write an operand before *i* writes its value
  - Write after Read (WAR) (anti dependence)
    - Instruction *j* tries to write a destination before it is read by *i*
- No such thing as a Read after Read (RAR) hazard since there is never a problem reading twice

lw $1 ... write
add $2, $1, $3 read

lw $1 write
lw $1 write

add $2, $1, $3 read
lw $1 write

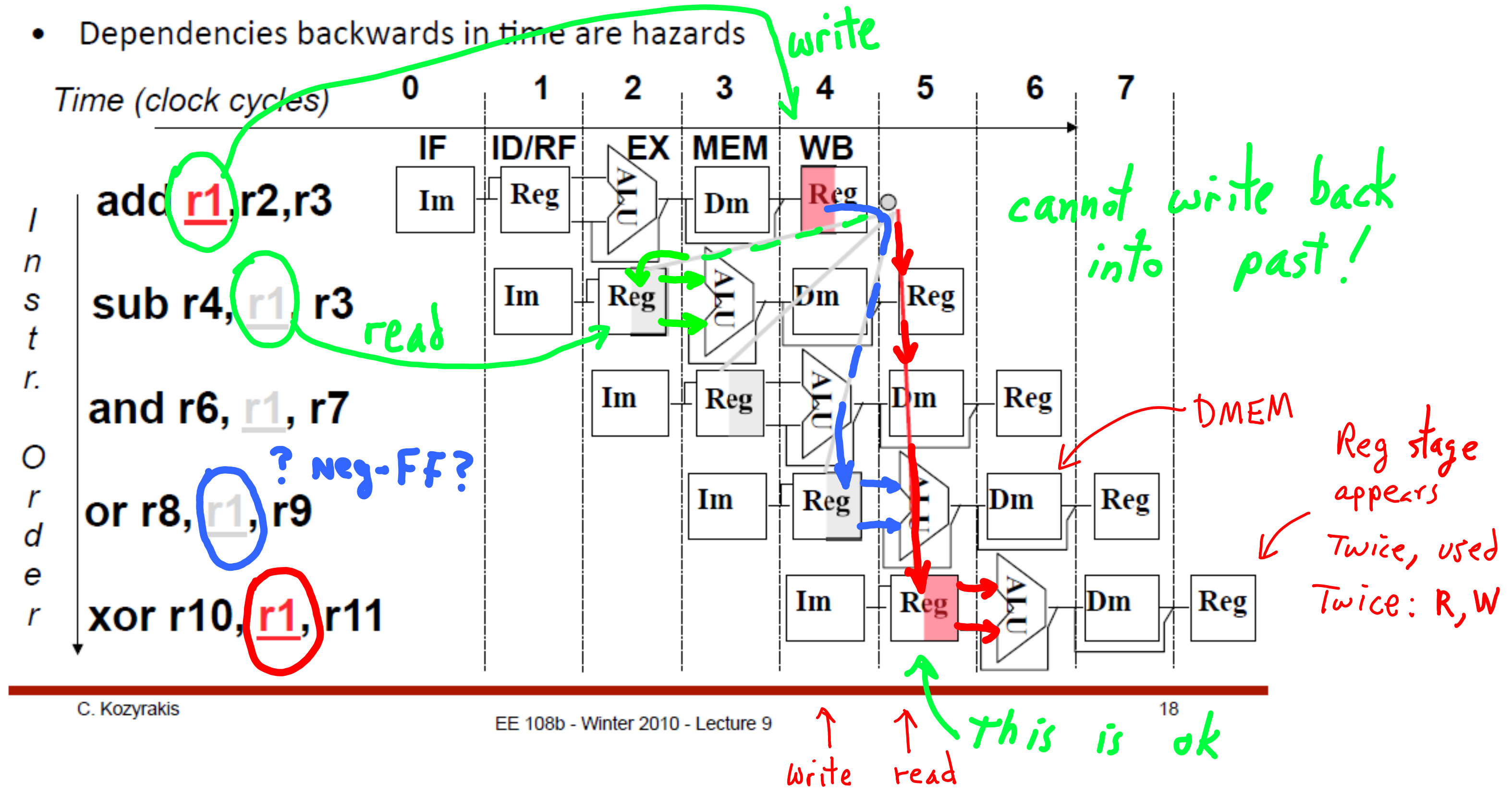
Cannot re-order effects of operations!

- Dependencies are a property of your program (always there)
- Dependencies may lead to hazards on a specific pipeline

RAW Hazard Example

Could we possibly send data from pipeline stage to stage?

- Dependencies backwards in time are hazards

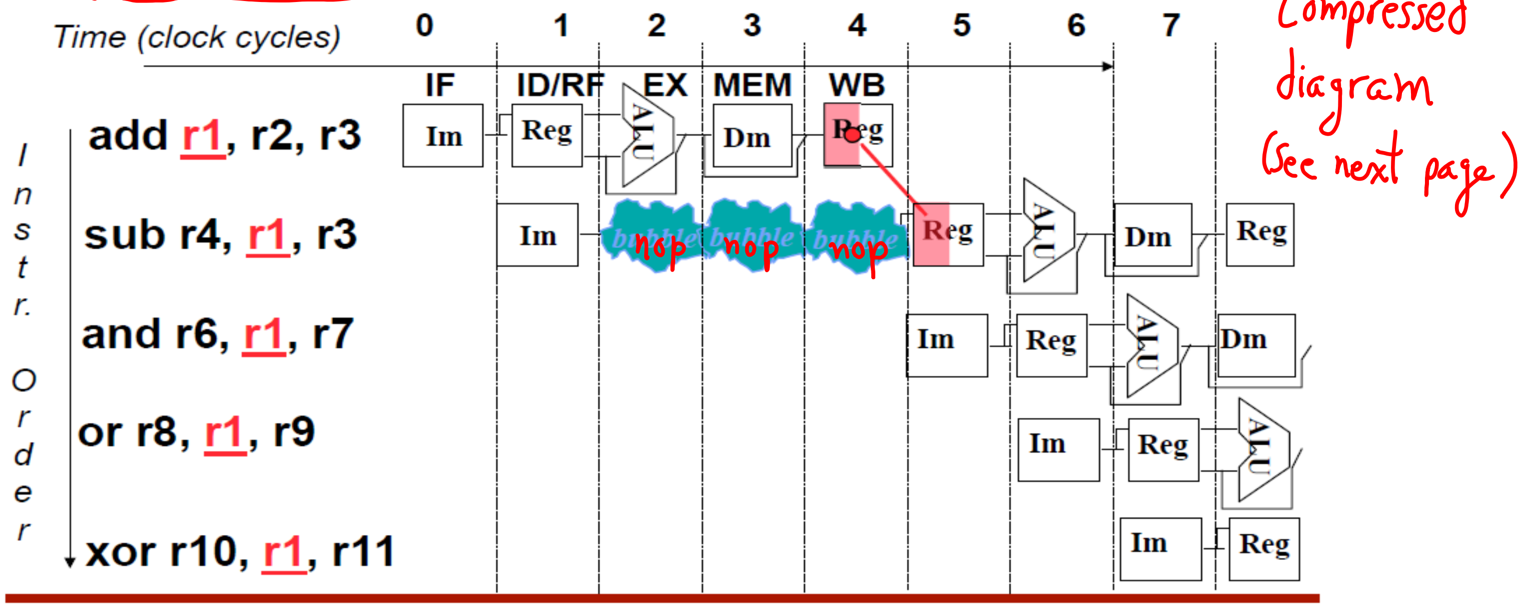


C. Kozyrakis

EE 108b - Winter 2010 - Lecture 9

18

- Eliminate reverse time dependency by stalling



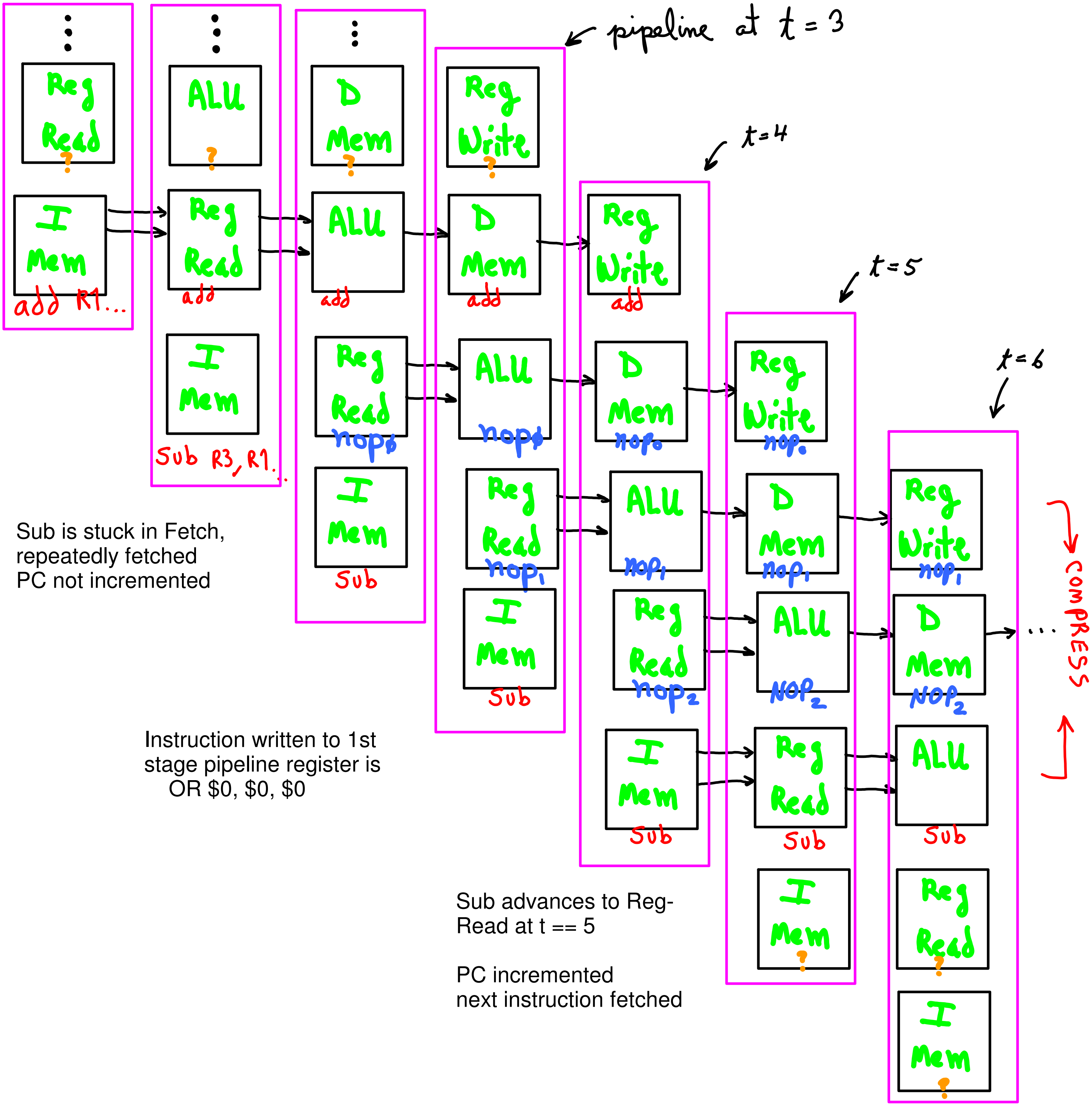
C. Kozyrakis

EE 108b - Winter 2010 - Lecture 9

20

- How can we delay the 2nd instruction?
  1. – Compiler insert independent work or NOPS ahead of it
     - NOP example: or $0, $0, $0
     - Disadvantage: pipeline-specific binary program

  But So
  2. – Hardware inserts NOPs as needed
     - AKA: pipeline interlocks
     - Advantage: correct operation for all programs/pipelines
     - Disadvantage: may miss some optimization opportunities ?
  3. – Most modern machines
     - Hardware inserts NOPs but compiler may try to minimize need



Sub is stuck in Fetch,
repeatedly fetched
PC not incremented

Instruction written to 1st
stage pipeline register is
OR $0, $0, $0

Sub advances to Reg-
Read at t == 5

PC incremented
next instruction fetched

- Stalls can have a significant effect on performance
- Consider the following case
  - The ideal CPI of the machine is 1
  - A RAW hazard causes a 3 cycle stall
- If 40% of the instructions cause a stall?
  - The new effective CPI is 1 + 3x0.4 = 2.2
  - And the real % is probably higher than 40%
- You get less than ½ the desired performance!

$60\%(1 \text{ cycle}) + 40\%(1+3 \text{ cycles})$

$= (1 \text{ cycle})(60\%+40\%) + (40\%)(3 \text{ cycles})$

$= 2.2$

$\Rightarrow S \rightarrow \frac{1}{2}$

added Logic

How to Stall the Pipeline OR How to Insert a NOP or Bubble

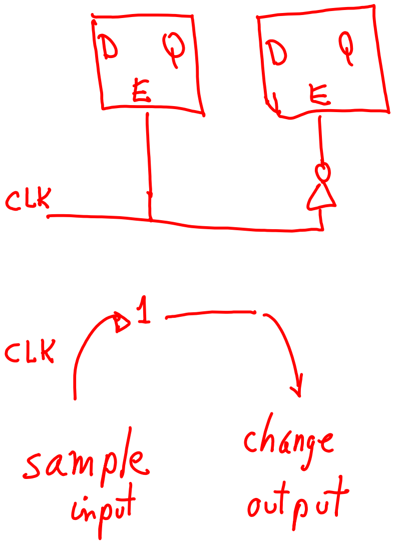
- You discover the need to stall when 2nd instruction is in ID stage
  - Idea: repeat its ID stage until hazard resolved; let all instructions ahead of it move forward; stall all instructions behind it

1. Force control values in ID/EX register a NOP instruction
   - As if you fetched or \$0, \$0, \$0
   - When it propagates to EX, MEM and WB on following cycles, nothing will happen (nop = no-operation)
2. Prevent update of PC and IF/ID register
   - Using instruction is decoded again — so what
   - Following instruction is fetched again — so what

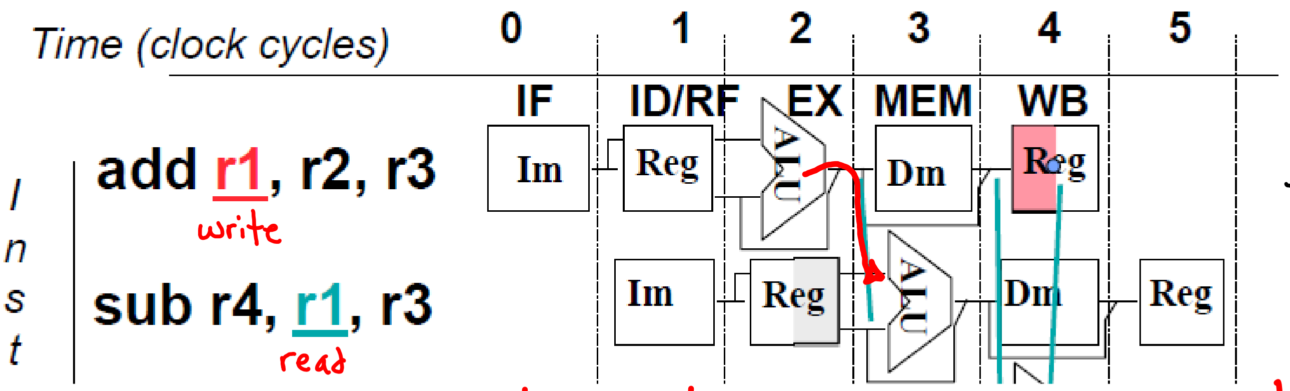
$R0 = \$0$ is always $= 0$

- We can allow data to flow through register file
  - If you read a register when it is being written, you get new value
  - Or assume write during 1st ½ of cycle, read during 2nd ½
  - Now you stall only 2 cycles

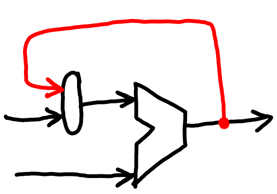
reg file register



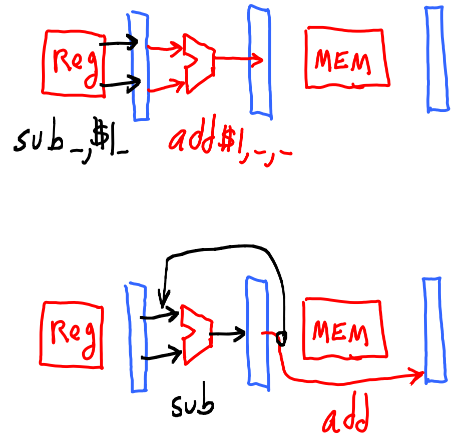
- “Forward” the data to the appropriate unit



as-if R1 read



send data directly to ALU input? Do write later?
(new feedback path)

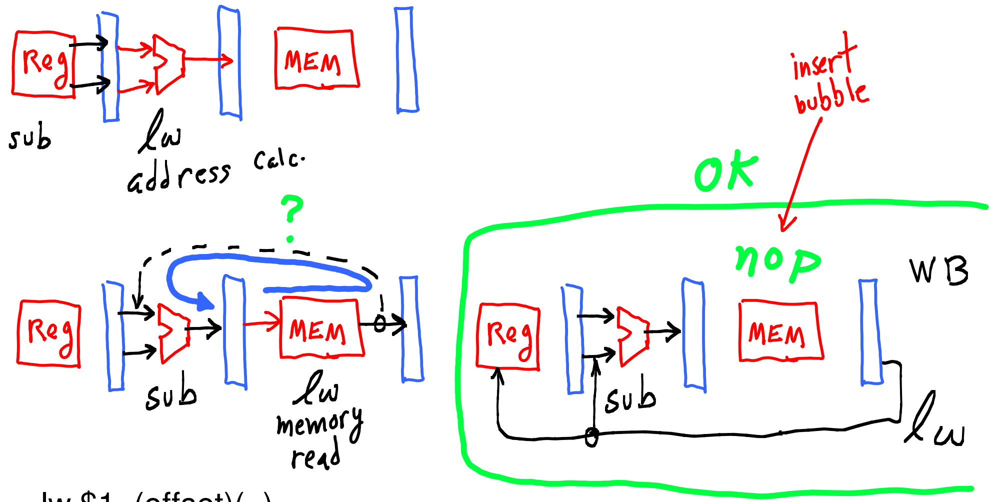


add $1, _, _
sub _, $1, _

Data available next tick.

Forwarding (feedback) works.

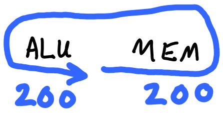
RAW hazard



lw $1, (offset)(_)
sub _, $1, _

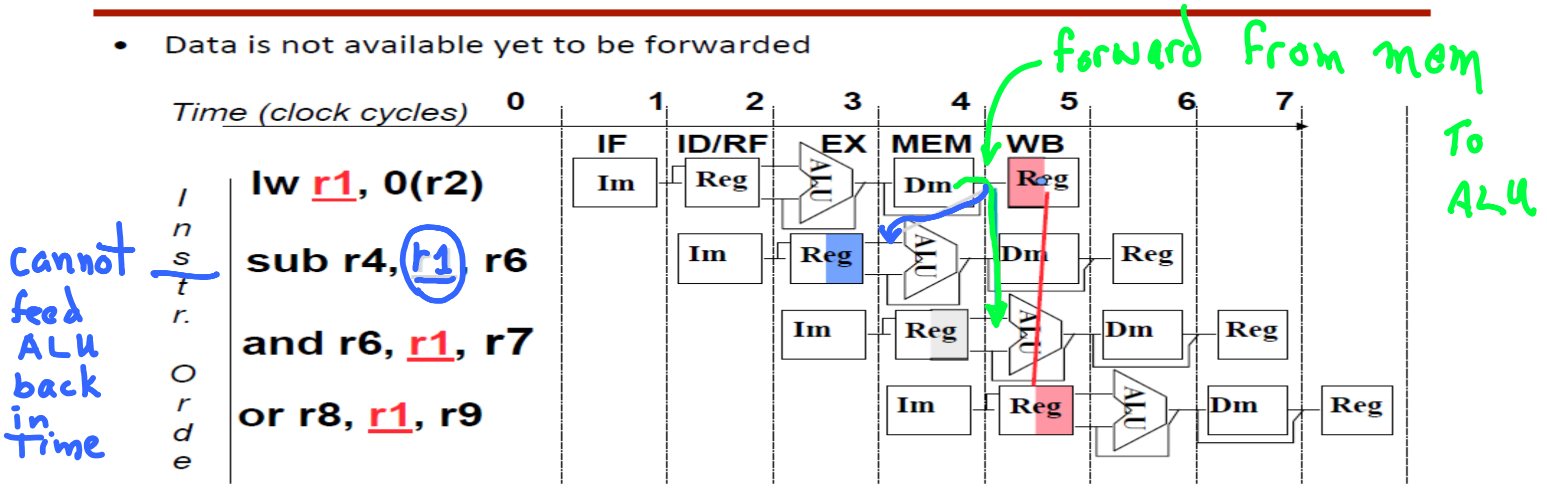
WHY NOT forward Dmem.out?

DELAY = 200ps (memory) + 200ps (ALU)

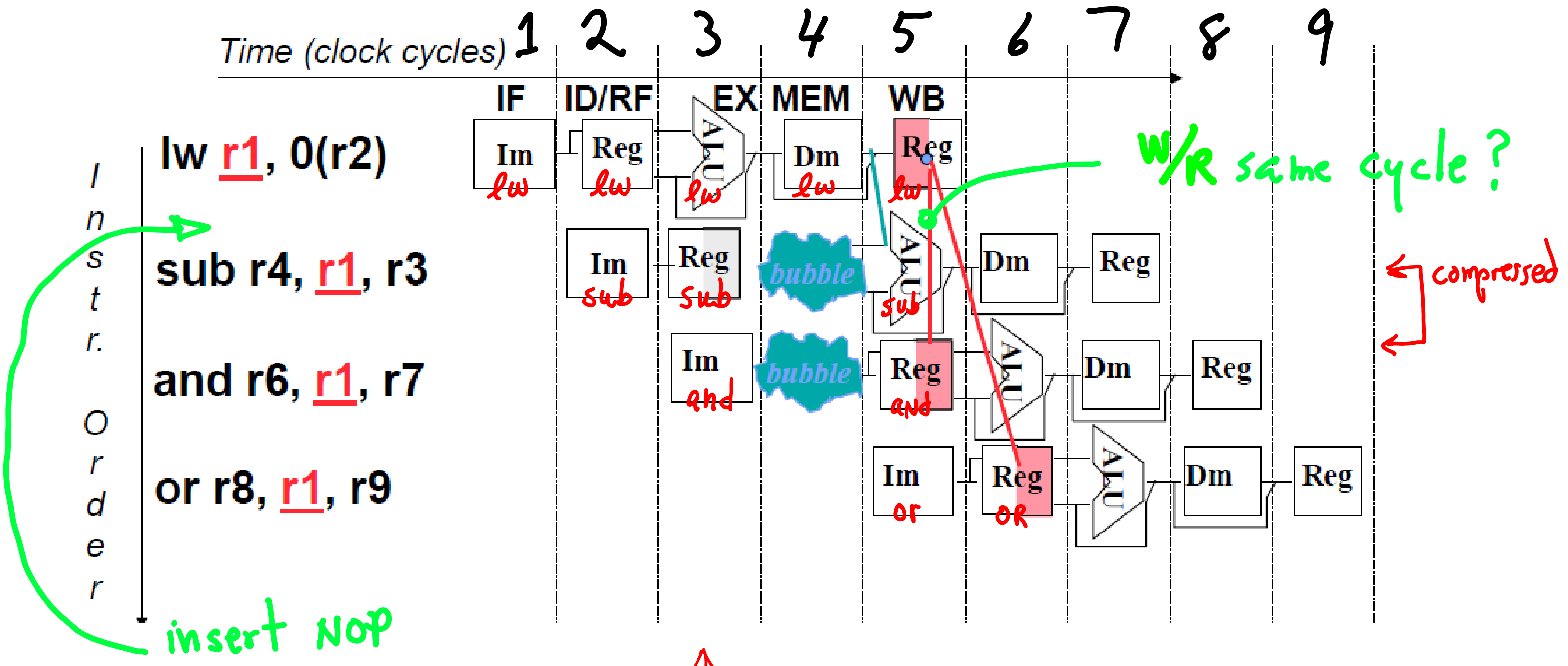


forward from WB instead, insert NOP

- Data is not available yet to be forwarded



- A *pipeline interlock* checks and stops the *instruction issue*

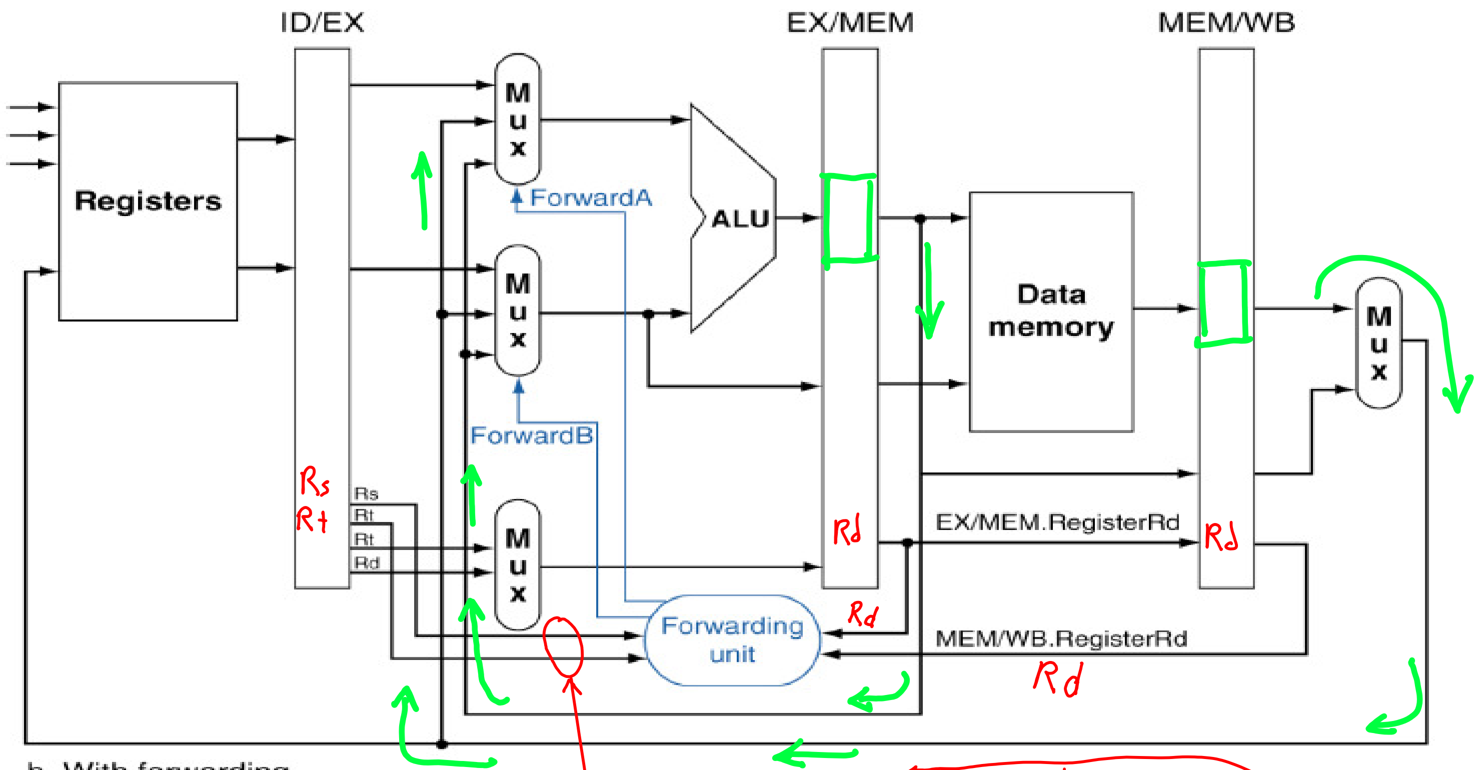


"Load Delay"

detect hazard

freeze IF and and ID sub

insert 1 nop



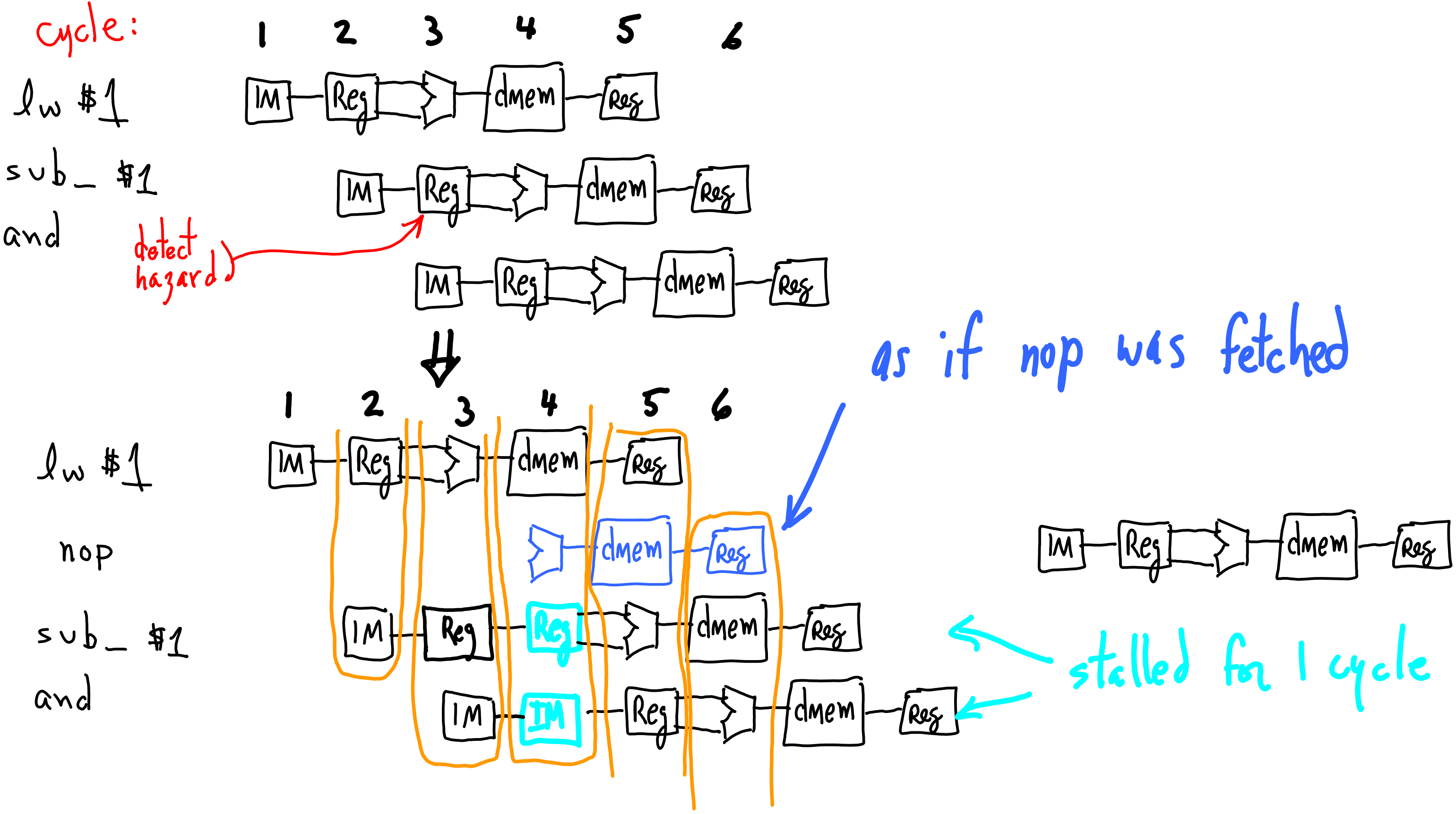
b. With forwarding

- Compare fields (Rs, Rt vs Rd)

- set muxes

Rs, Rt

Forwarding Paths



Load Delay

Bypassing can't fix the problem with ADD since the data simply isn't available! We have to add some *pipeline interlock hardware* to stall ADD's execution.

XOR

1st `LD(r1, 0, r4)`
2nd `ADD(r1, r4, r5)`
3rd `XOR(r3, r4, r6)`

Pipe stages

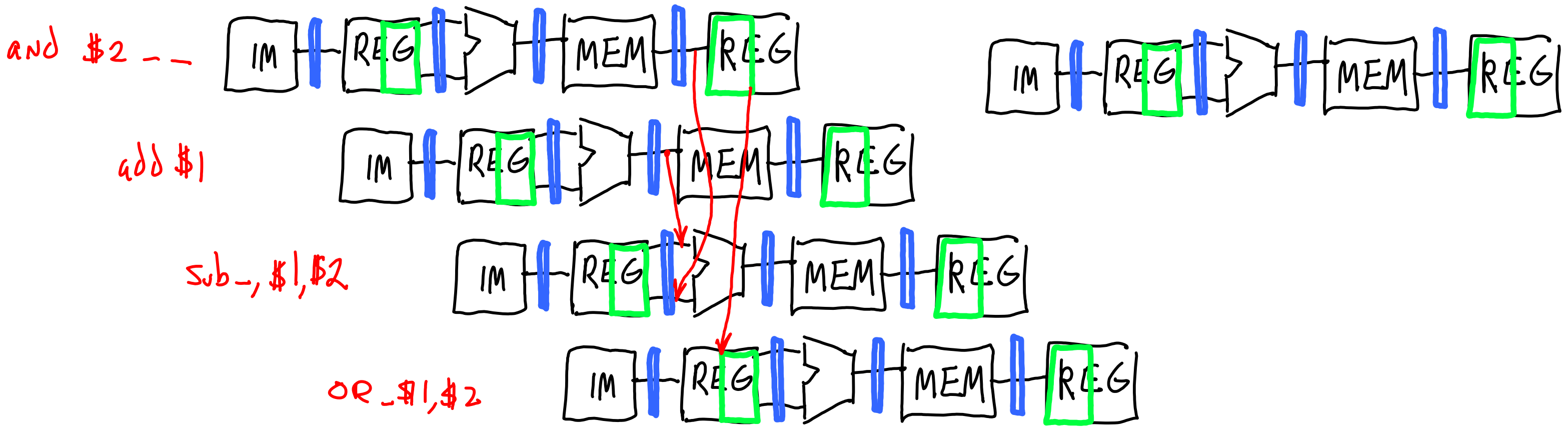
	0	1	2	3	4	5	6
IF	LD	ADD	XOR	XOR			
RF		LD	ADD	ADD	XOR		
ALU			LD	NOP	ADD	XOR	
WB				LD	NOP	ADD	XOR

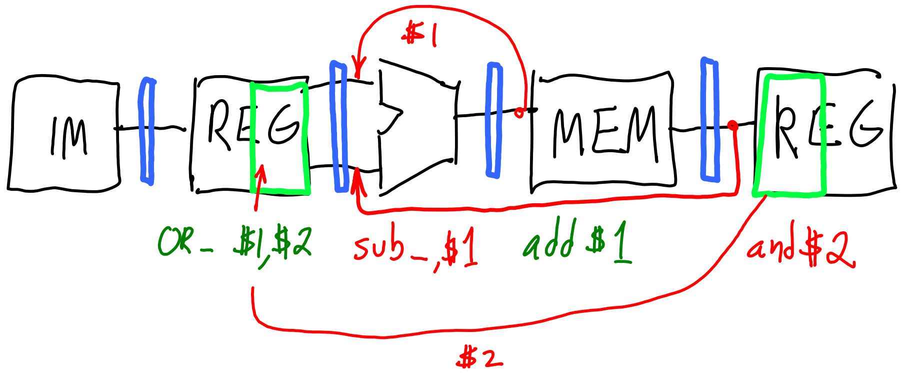
← XOR stalled

← ADD stalled (frozen in IF or RF at 2-3)

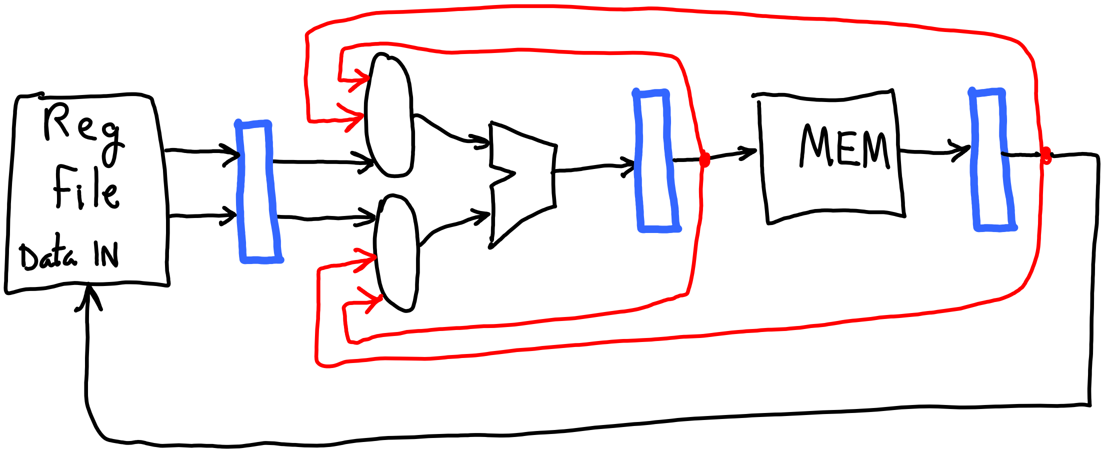
If the compiler knows about a machine's load delay, it can often rearrange code sequences to eliminate such hazards. Many compilers provide machine-specific instruction scheduling. → binary arch. dependent?

multiple feedback at once?





Feedback paths to ALU go to both inputs.
Hazard detection sets MUXes: Opcode needed in pipe stage registers for detection.


Reg File
Data IN
MEM