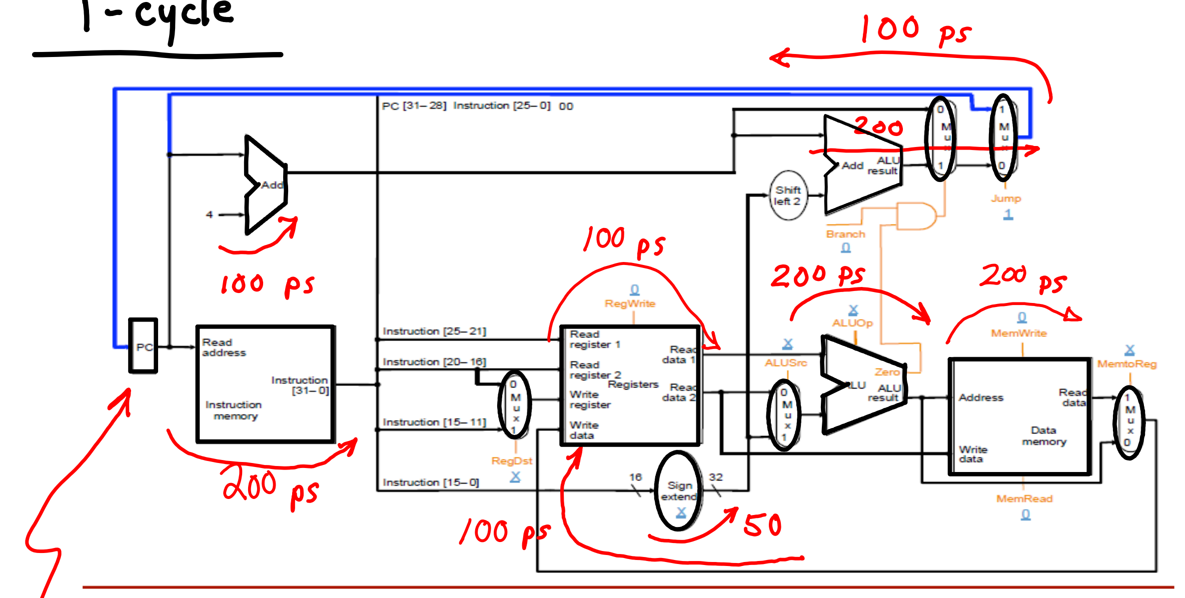
Performance

How fast can we clock?

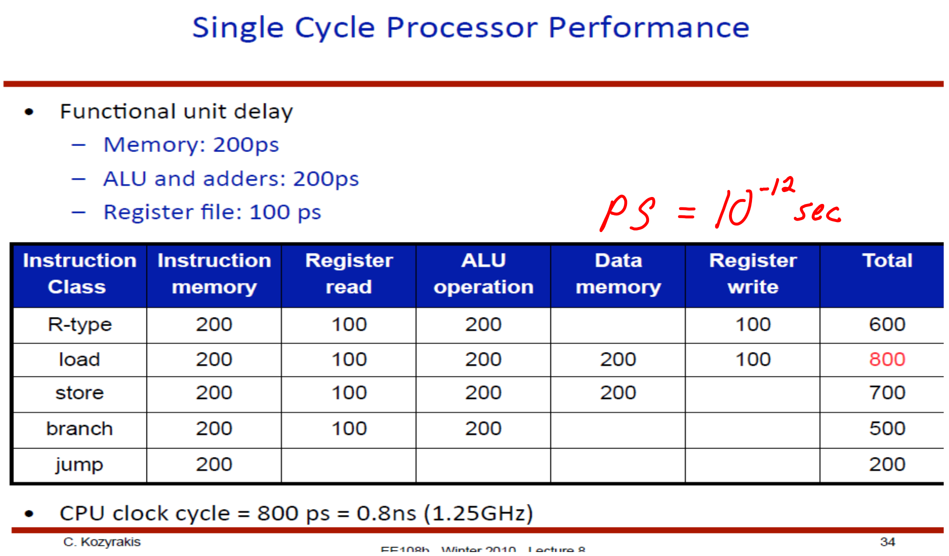
1. 1 cycle MIPS performance
3. general pipelining
8. MIPS pipe, LW
12. MIPS performance, piped vs non-piped
14. arrays, piped
15. hazards

1-cycle



clock changes PC output.

How long before we can clock PC, Regfile? critical Path?



Single Cycle Processor Performance

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

$ps = 10^{-12} sec$

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

max delay = T_{clock}

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

What if we make control more complex?
Set clock 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

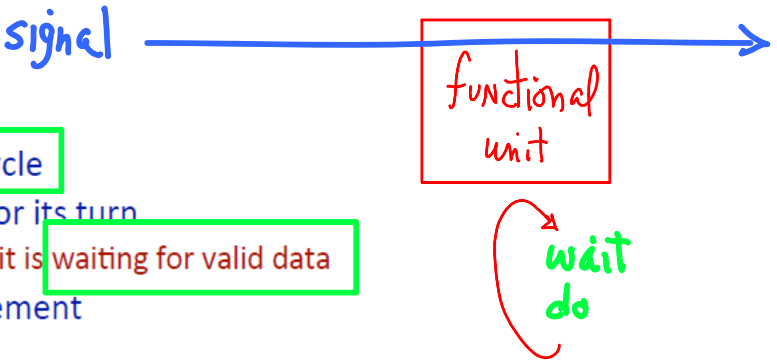
ps → 0.6 ns
0.8
0.7
0.5
0.2

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

⟹ 1.6 GHz

what is speedup? $S_{new\text{-}old} = \frac{n\ CPI\ (1/CR)}{n\ CPI\ (1/CR)} = CR_{new}/CR_{old} = 1.25/1.6$ worth it? skew?

What's the



- Problem:
  - Each functional unit used once per cycle used
  - 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

- 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↑ ?

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

$$= \uparrow CR / CPI \uparrow$$

Can we win?

If CR↑ by slicing path into k small pieces,

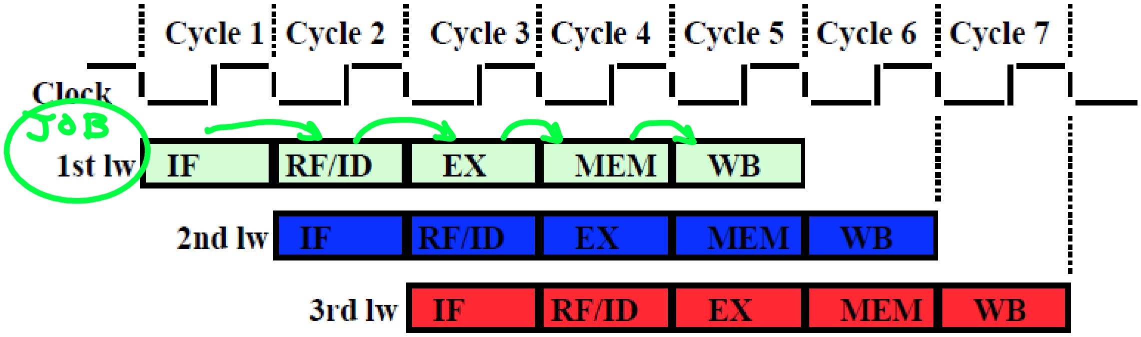
delay → delay/k ⟶ T_{clock} → T_{clock}/k ⟶ CR → kCR

#cycles → k ⟶ CPI → kCPI

$$Perf \rightarrow \frac{n}{n\ (k\,CPI)(1/kCR)}$$ no change?

Pipelining Load lw

- Load instruction takes 5 stages
  - Five independent functional units work on each stage
    - Each functional unit used only once for a single instr.
  - 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

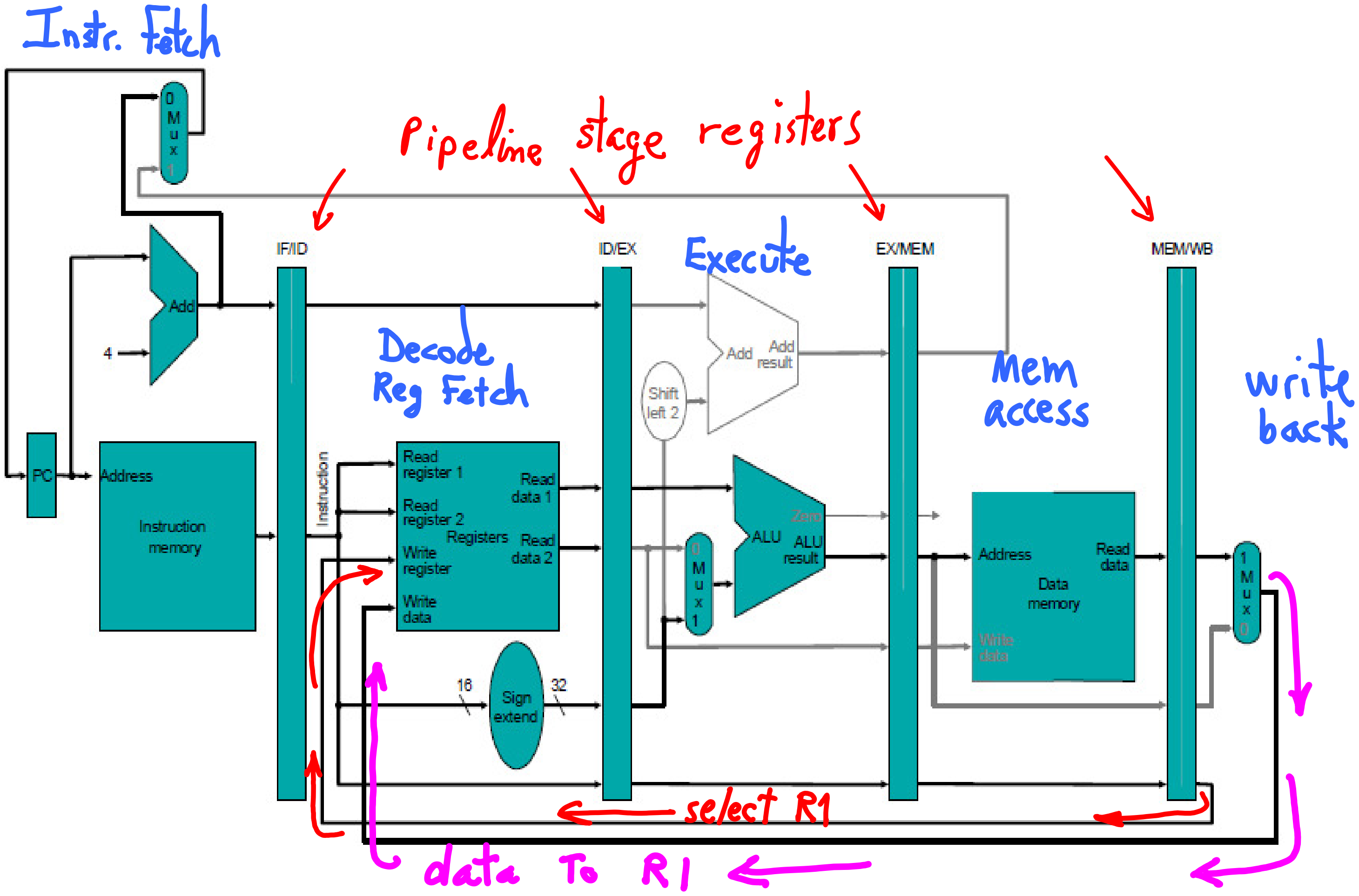


all stages busy

1 instr. exits per cycle

$$CPI = \frac{1 \text{ instr.}}{1 \text{ cycle}}$$

$$T_{latency} = 5 \text{ cycles}$$



SUB R4, R5, R1

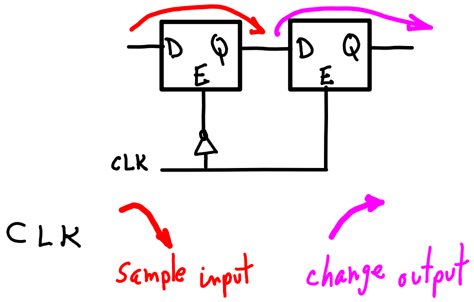
Required delay before using written data.

Insert NOPs (BRnzp #0)? Compiler does this, or HW?

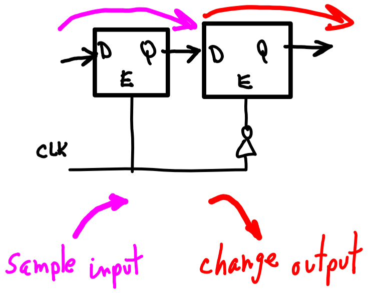
ADD R1, R2, R3

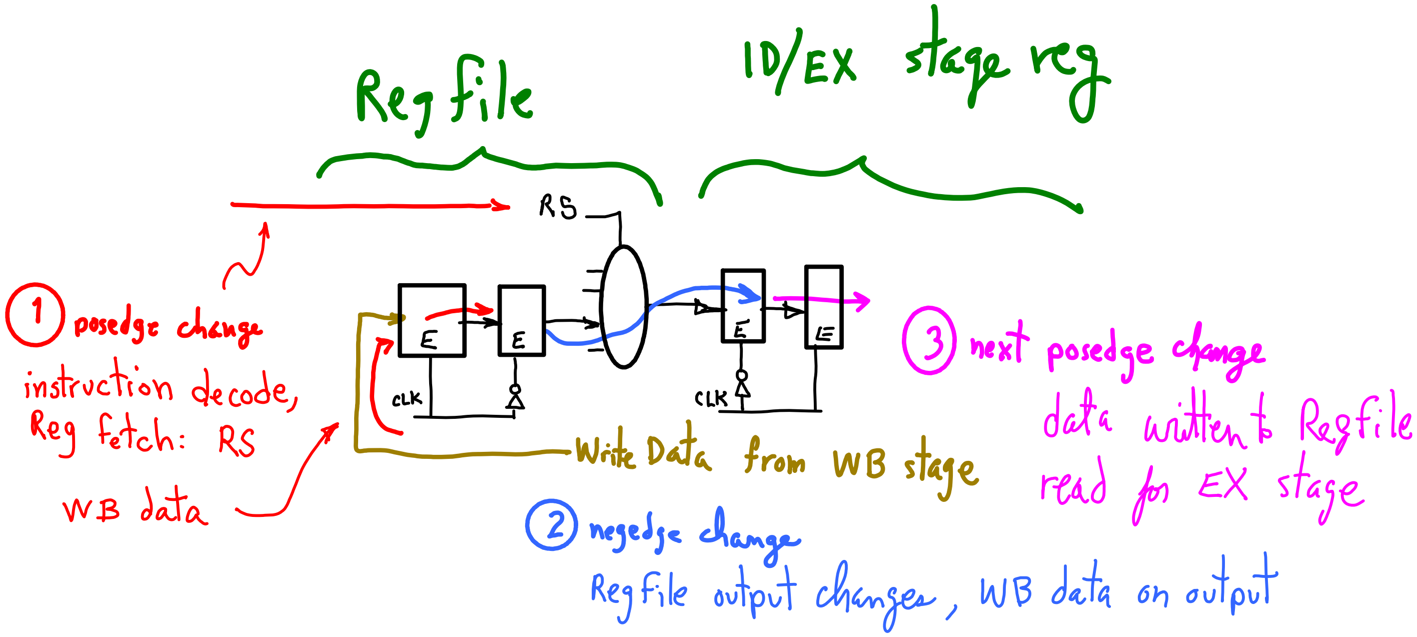
fix delay? → negedge FF for Regfile

Positive edge-triggered FF:
output changes on rising clock



Negative edge-triggered FF:
output changes on rising clock





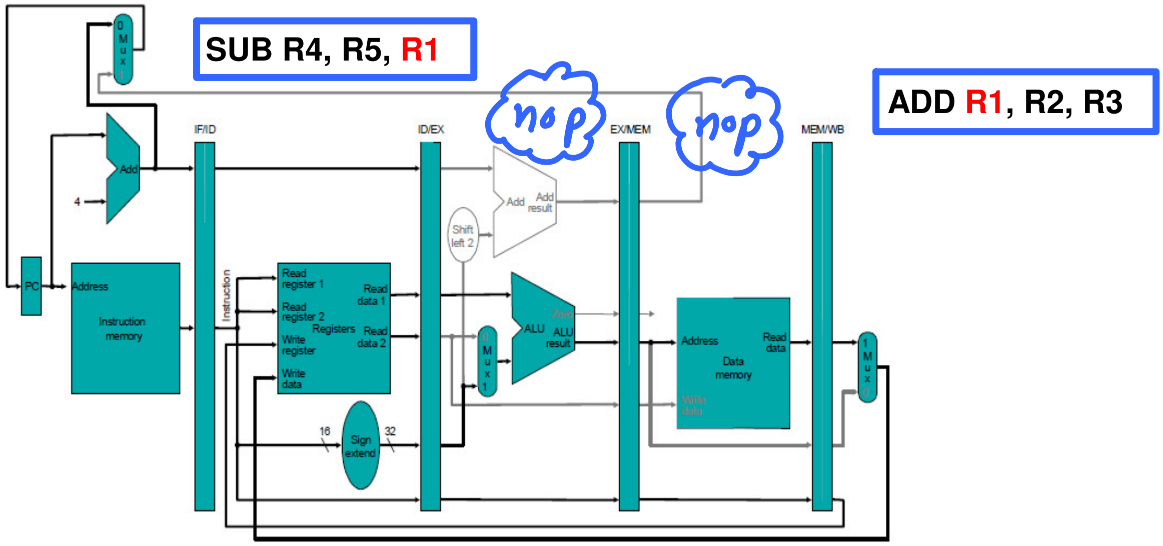
1 posedge change
instruction decode,
Reg fetch: RS
WB data

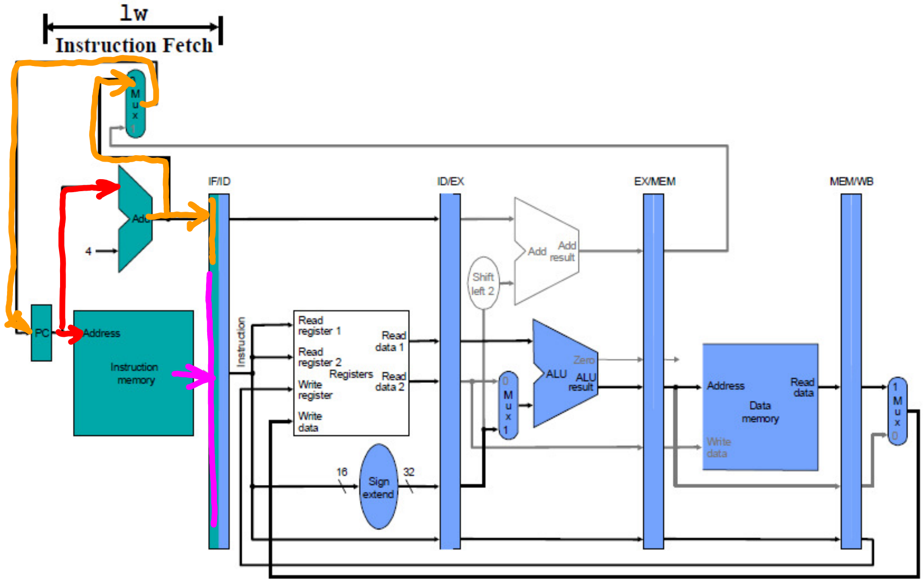
Write Data from WB stage

2 negedge change
Reg File output changes, WB data on output

3 next posedge change
data written to Regfile
read for EX stage

Written data available for read in same cycle.





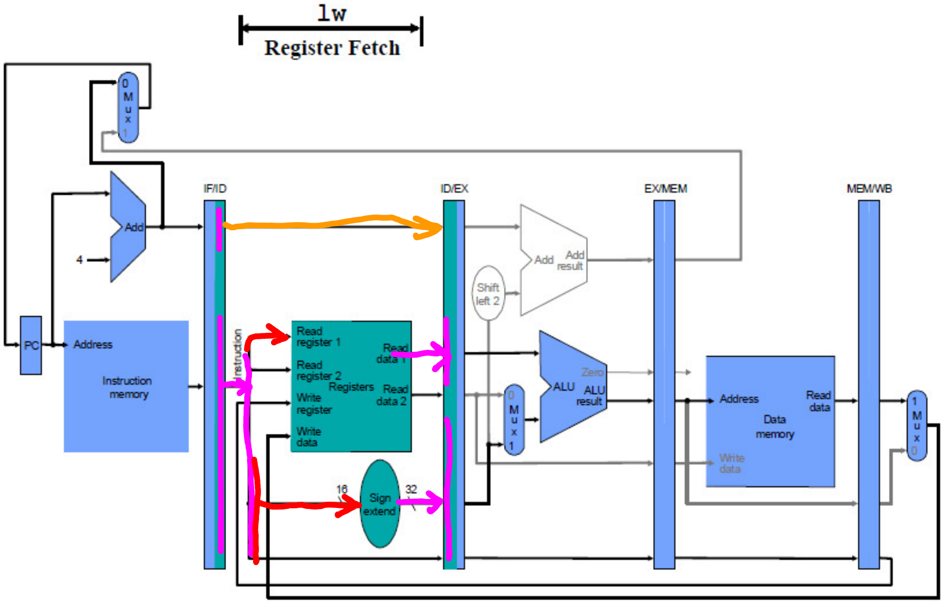
MIPS: LW $2, 15($1)
[op | rs | rt | off]

LC4: LDR R2, R1, #15
[op | SR1 | DR | off]

Instruction fetch:

PC++ ==> PC

IF/ID
PC++ ==> PC
Instruction ==> Instr



Decode/reg fetch:

IF/ID ID/EX

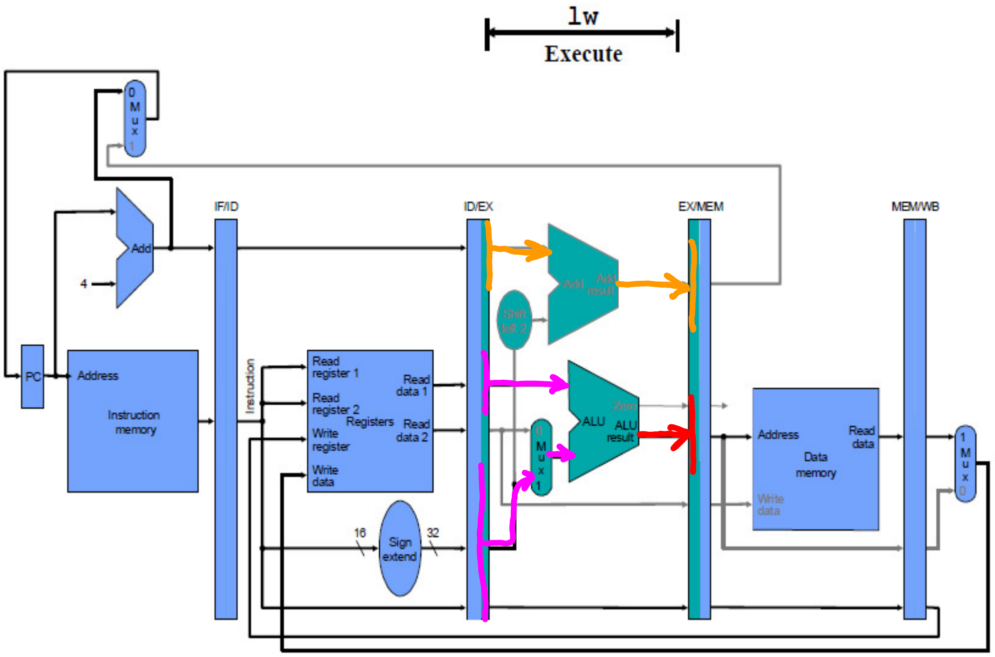
PC ==> PC

SR1 ==> SR1
SR1out ==> SR1out

offset ==> SEXT
==> offset

Instr.SR2 ==> DR

Instr.OP ==> decode
==> CTL



Execute:

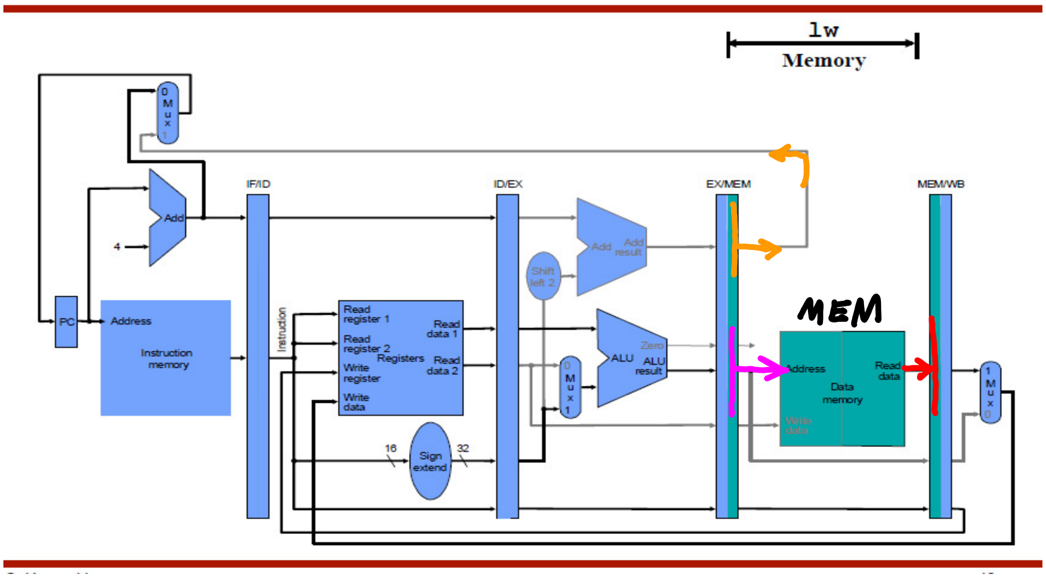
ID/EX EX/MEM

PC ==> ADD ==> PC

SR1out ==> ALU
offset ==> ALU
==> Res

DR ==> DR

CTL ==> CTL



MEM:

EX/MEM MEM/WB

PC ==> BRmux

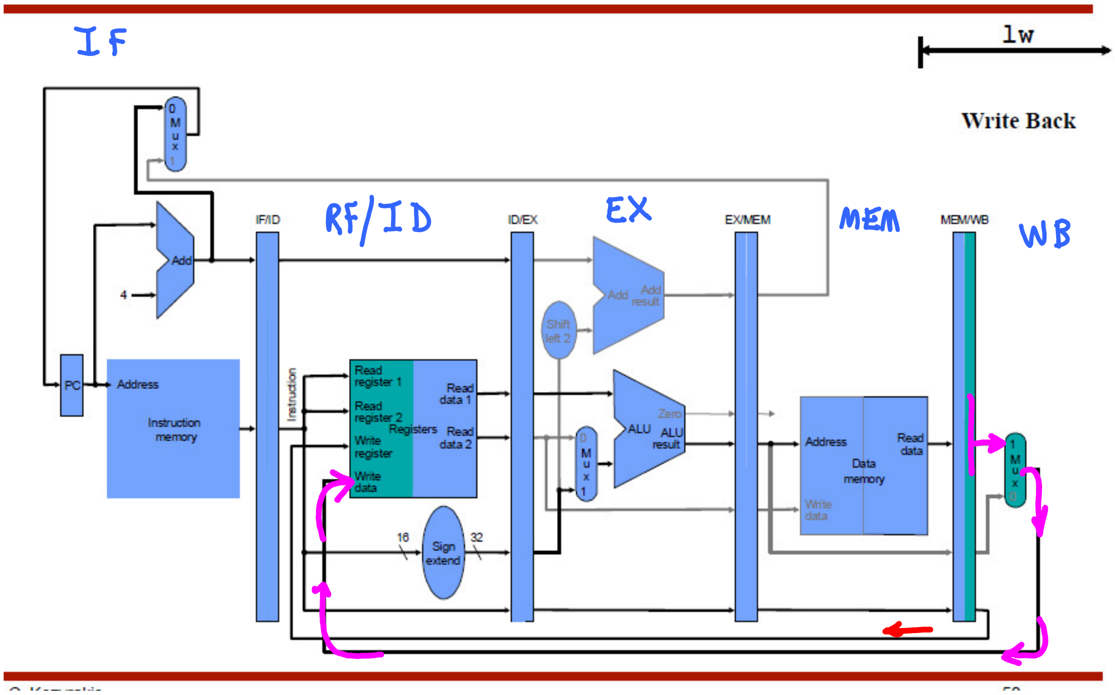
Res ==> MEM.addr

MEM.out ==> MEMdata

Res ==> REGdata

DR ==> DR

CTL ==> CTL



MEM:

MEM/WB

MEMdata ==> RegFile.in

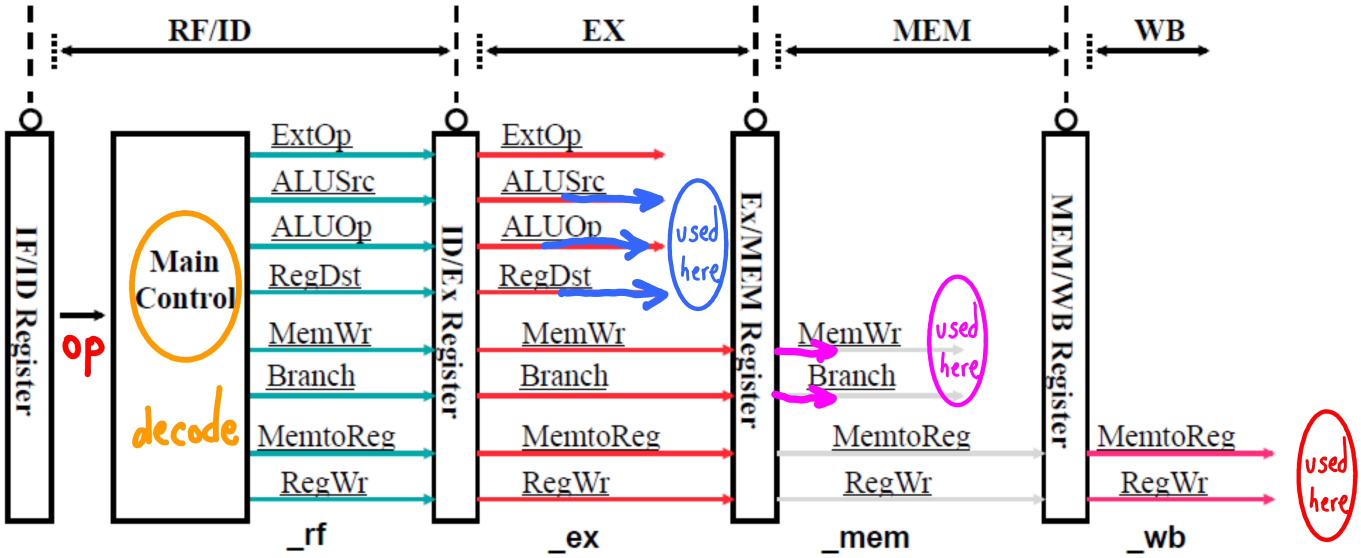
DR ==> RegFile.DR

CTL ==> RegFile.rw

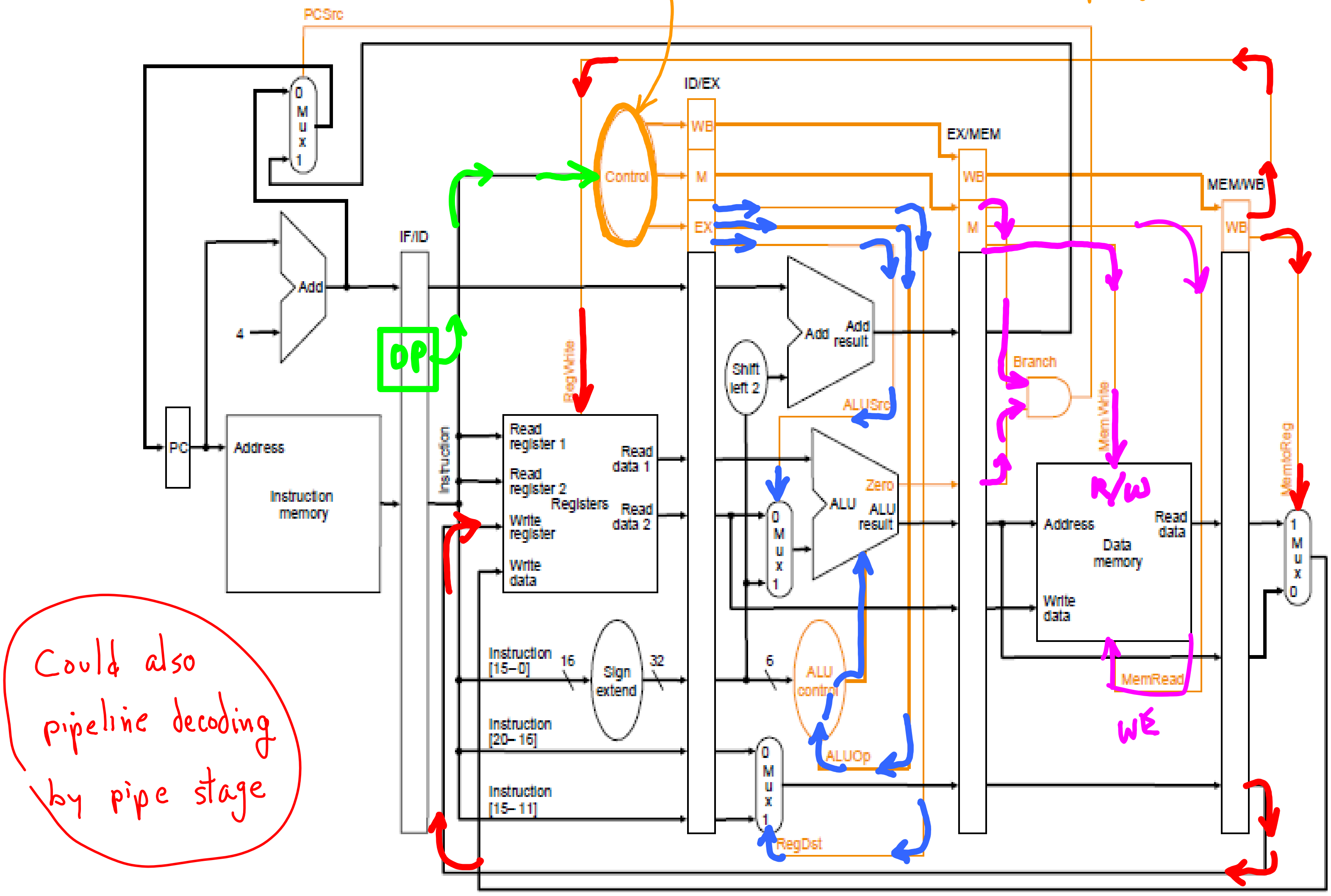
- Use a Main Control unit to generate signals during RF/ID Stage
  - Control signals for EX
    - (ExtOp, ALUSrc, ...) used 1 cycle later
  - Control signals for Mem
    - (MemWr, Branch) used 2 cycles later
  - Control signals for WB
    - (MemtoReg, MemWr) used 3 cycles later

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

Implementing Control



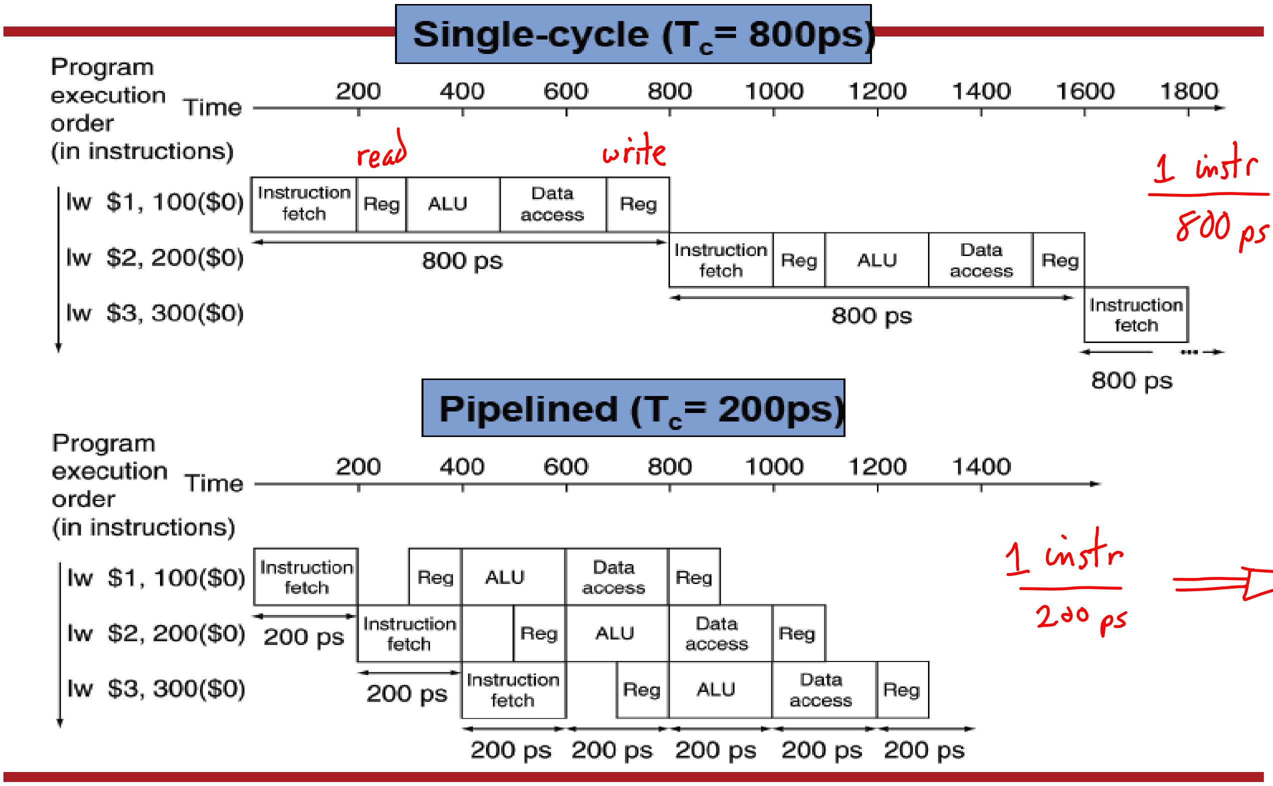
Decoding: combinational / μCode control signals (table lookup by opcode)



Could also pipeline decoding by pipe stage

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

Pipeline Performance



$\frac{1 \text{ instr}}{800 \text{ ps}}$

$\frac{1 \text{ instr}}{200 \text{ ps}} \Rightarrow S = 4$

- 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

↗ orthogonality

vs. 2 for misaligned ⇒ stall

But Something Is Fishy Here

- If dividing it into 5 parts made the clock faster 800 ps → 200ps
  - And the effective CPI is still one (if?)
- Then dividing it into 10 parts would make the clock even faster 200 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

An instruction does not complete in 1 cycle, need result before latency is done?

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 ↑

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

      at same time

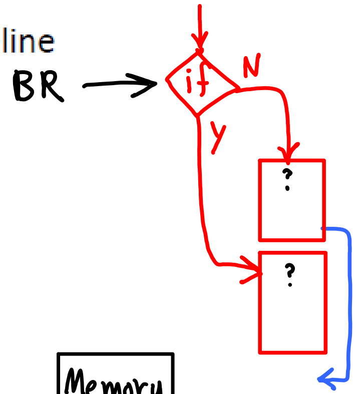
LC3's LDI

2 refs to data memory

  - Data Hazard
    - Instruction depends on result of instruction still in the pipeline

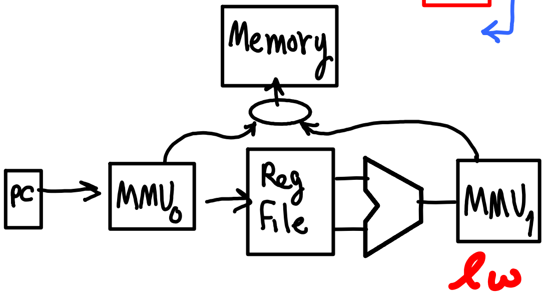
lw
add

  - Control Hazard
    - Instruction fetch depends on the result of instruction in pipeline

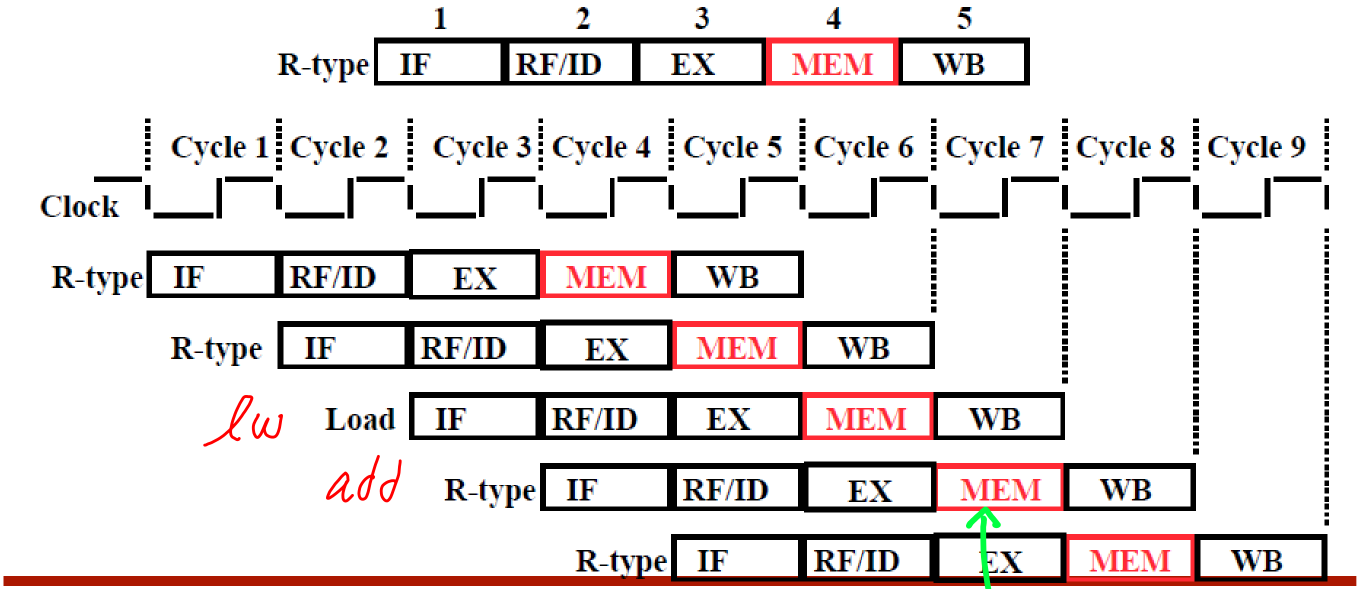


STRUCTURAL HAZARD

- 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"



- 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?



delay writing

sequential consistency

Data Dependencies

- Data dependencies for instruction *j* following instruction *i*

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

  - Read after Write (RAW) (true dependence)
    - Instruction *j* tries to read before instruction *i* tries to write it

  lw $1 write
  lw $1 write

  - Write after Write (WAW) (output dependence)
    - Instruction *j* tries to write an operand before *i* writes its value

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

  - 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

Cannot re-order effects of operations!

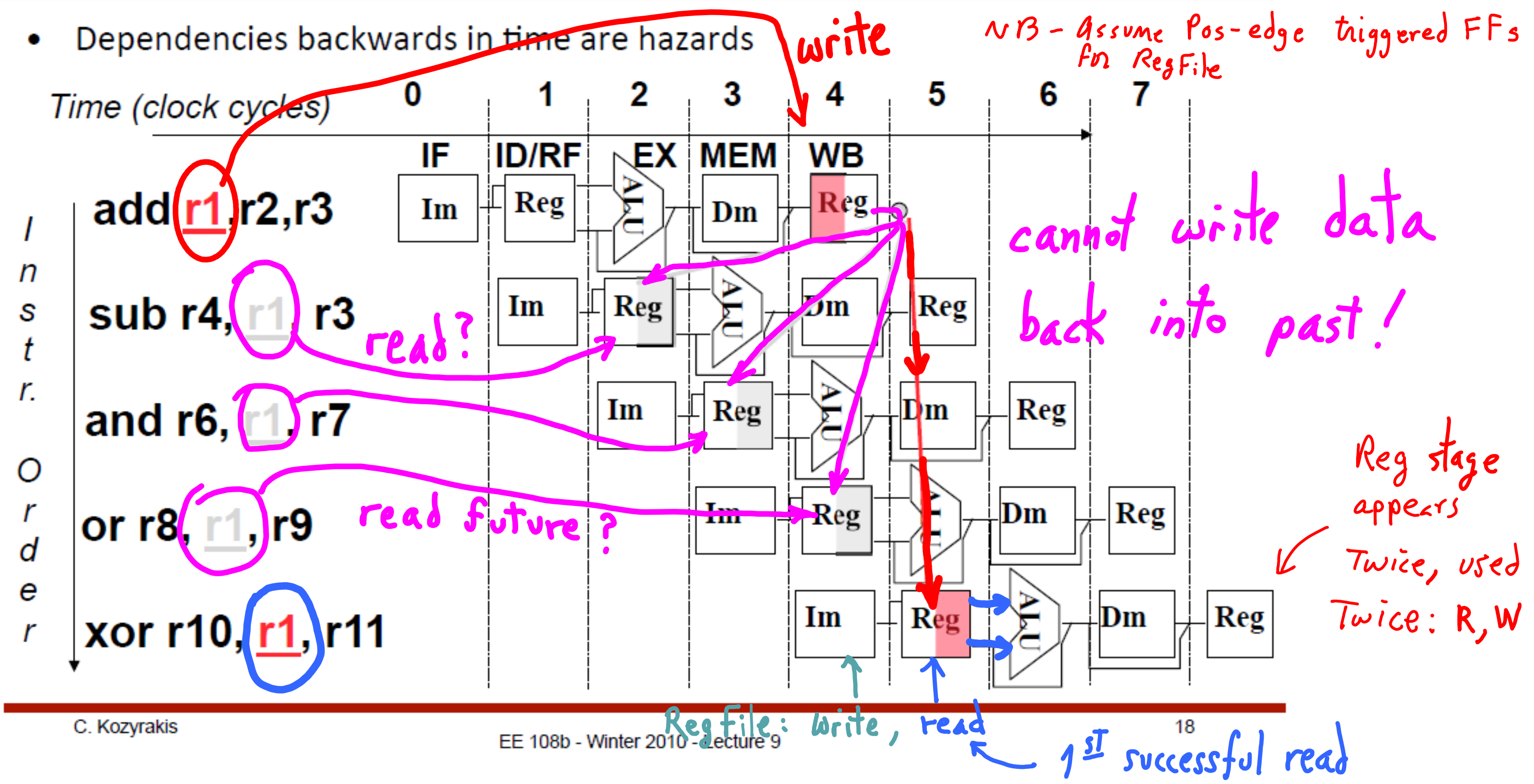
- Dependencies are a property of your program (always there)
- Dependencies may lead to hazards on a specific pipeline

RAW Hazard Example

Could we possibly send data from pipeline stage to stage?

- Dependencies backwards in time are hazards

N.B - Assume Pos-edge triggered FFs for RegFile



add r1,r2,r3

sub r4, r1, r3

and r6, r1, r7

or r8, r1, r9

xor r10, r1, r11

cannot write data back into past!

Reg stage appears Twice, used Twice: R, W

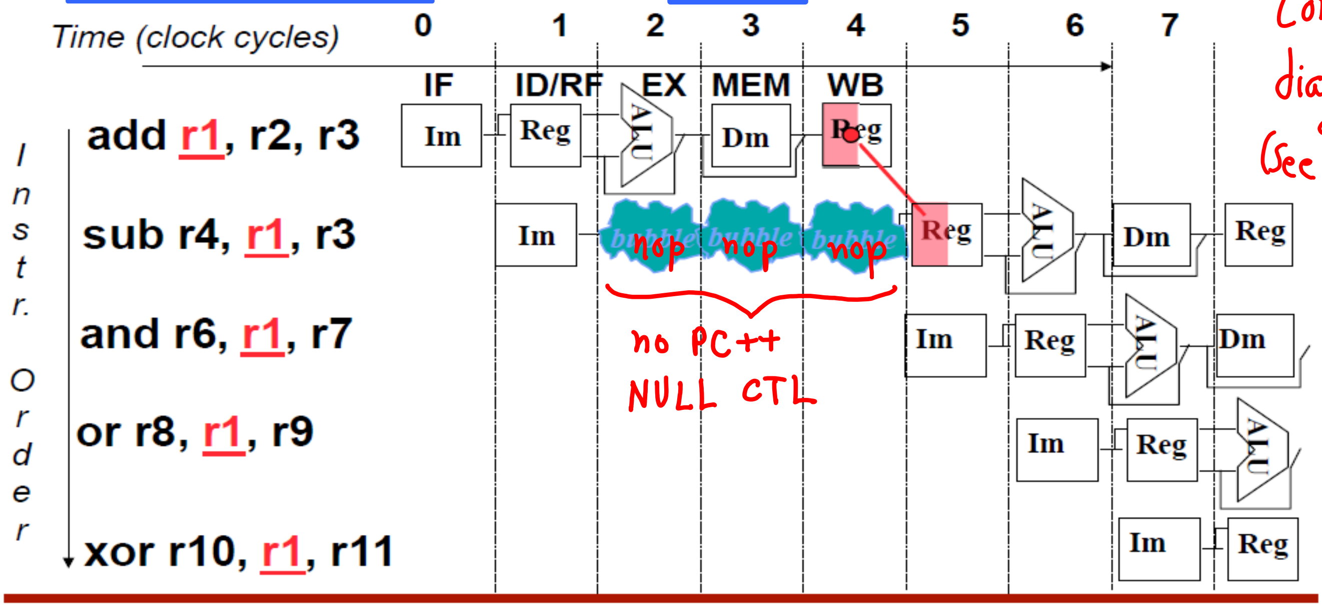
1st successful read

C. Kozyrakis

EE 108b - Winter 2010 - Lecture 9

18

- Eliminate reverse time dependency by stalling



add r1, r2, r3

sub r4, r1, r3

and r6, r1, r7

or r8, r1, r9

xor r10, r1, r11

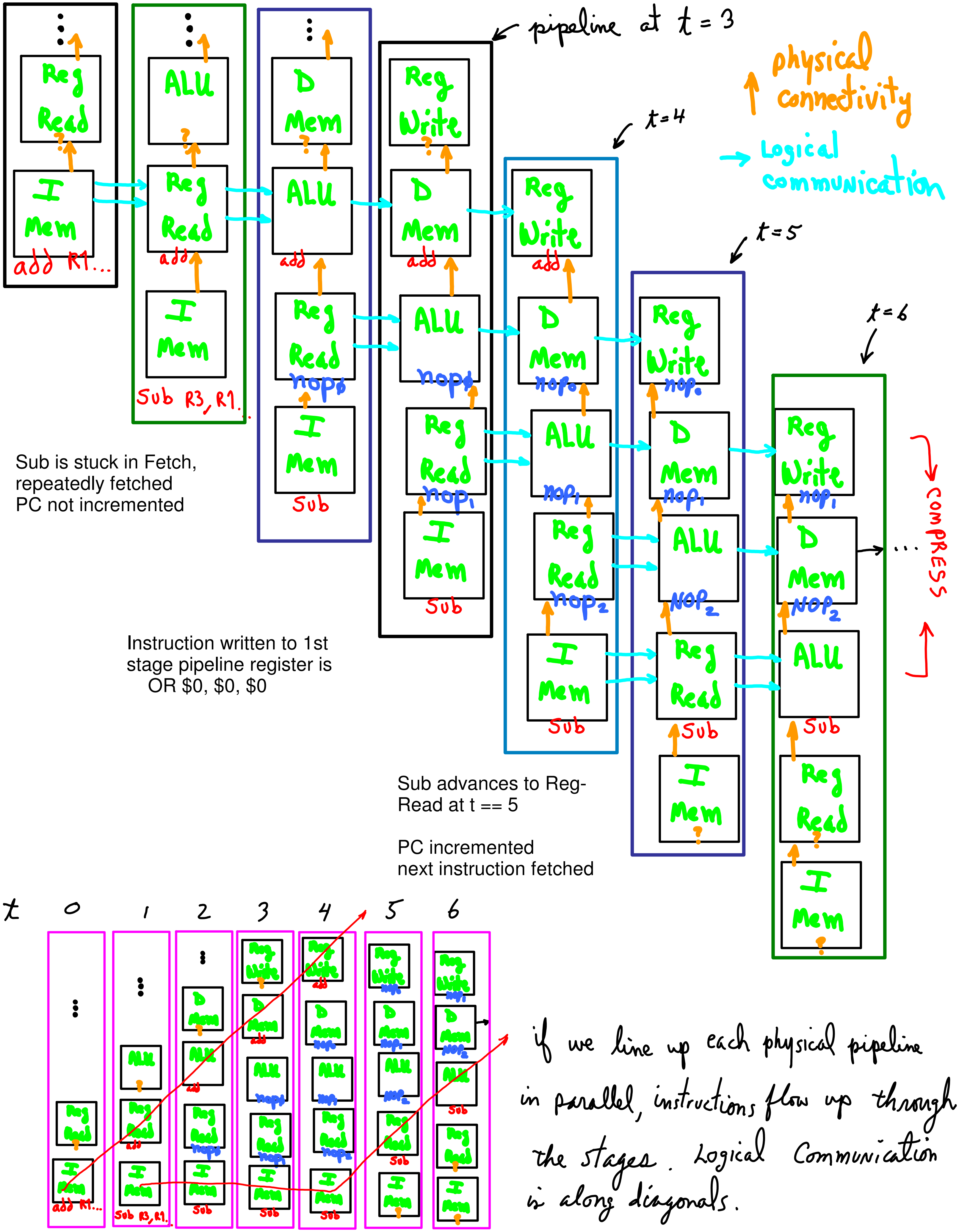
Compressed diagram (See next page)

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 → binary incompatibility between models of Processor
  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

if we line up each physical pipeline in parallel, instructions flow up through the stages. Logical Communication is along diagonals.

STALLS + Performance

Suppose 40% of instructions cause 3-bubble stalls

$$T_{w/\,stalls} = (n \text{ instructions})\Big[(60\%)(1 \text{ cycle}) + (40\%)(1 \text{ cycle} + 3 \text{ bubbles})\Big](1/CR)$$

$$T_{w/o\,stalls} = (n \text{ instructions})\Big[(100\%)(1 \text{ cycle})\Big](1/CR)$$

$$S_{w/-w/o} = [1 \cdot 1] \big/ [0.6 + 0.4(4)] = 1/2.2 < 1/2$$

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

added logic

- 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
   - Following instruction is fetched again

NOP

OR R1, R1, R1

AND R1, R1, R1

1 or 1 = 1

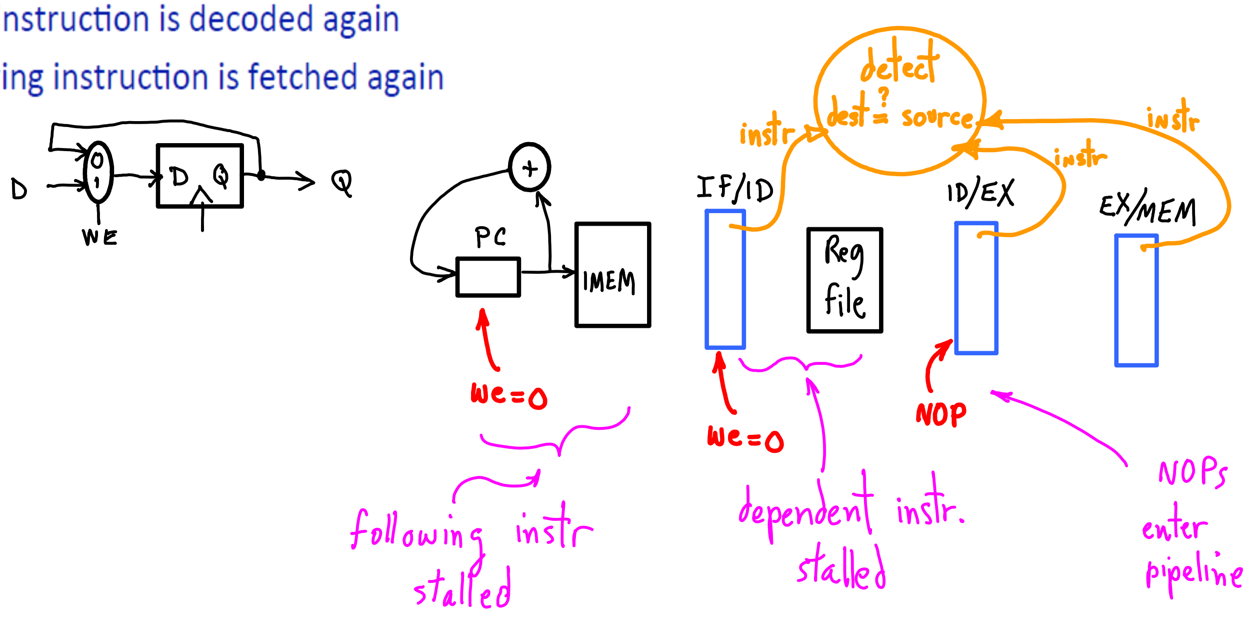
0 or 0 = 0

1 and 1 = 1

0 and 0 = 0

OR, set WE=0

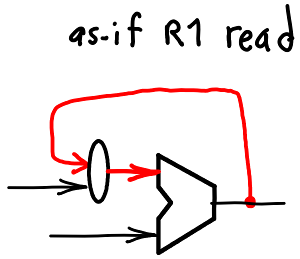
0 → WE



BETTER THAN STALLING

send data directly to ALU input? Do write later?

(new feedback path)

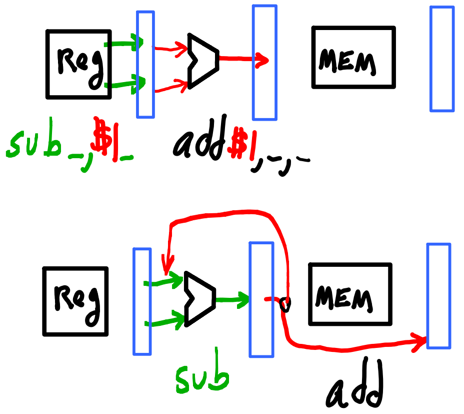


RAW reg-mode?

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

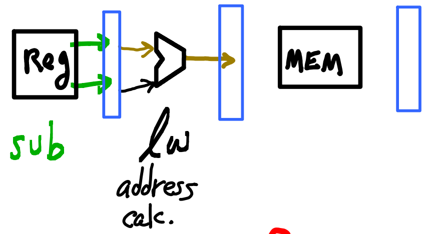
Data available next tick.

Forwarding (feedback) works.



LW?

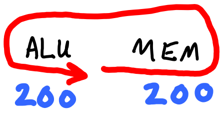
lw $1, _, _
sub _, $1, _

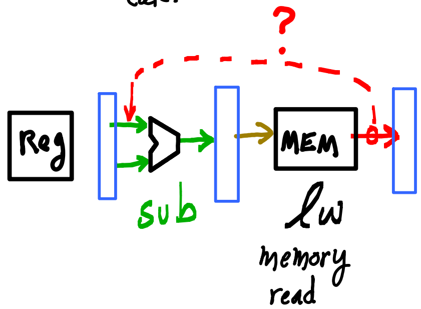


WHY NOT forward Dmem.out?

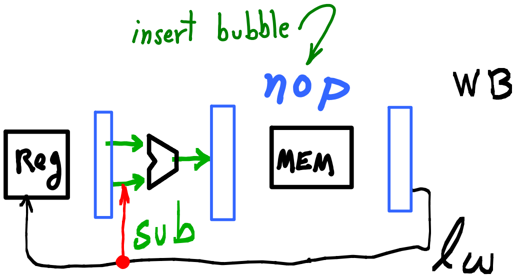
DELAY = 200ps (memory) + 200ps (ALU)

SLOW down clock to 400ps?!?

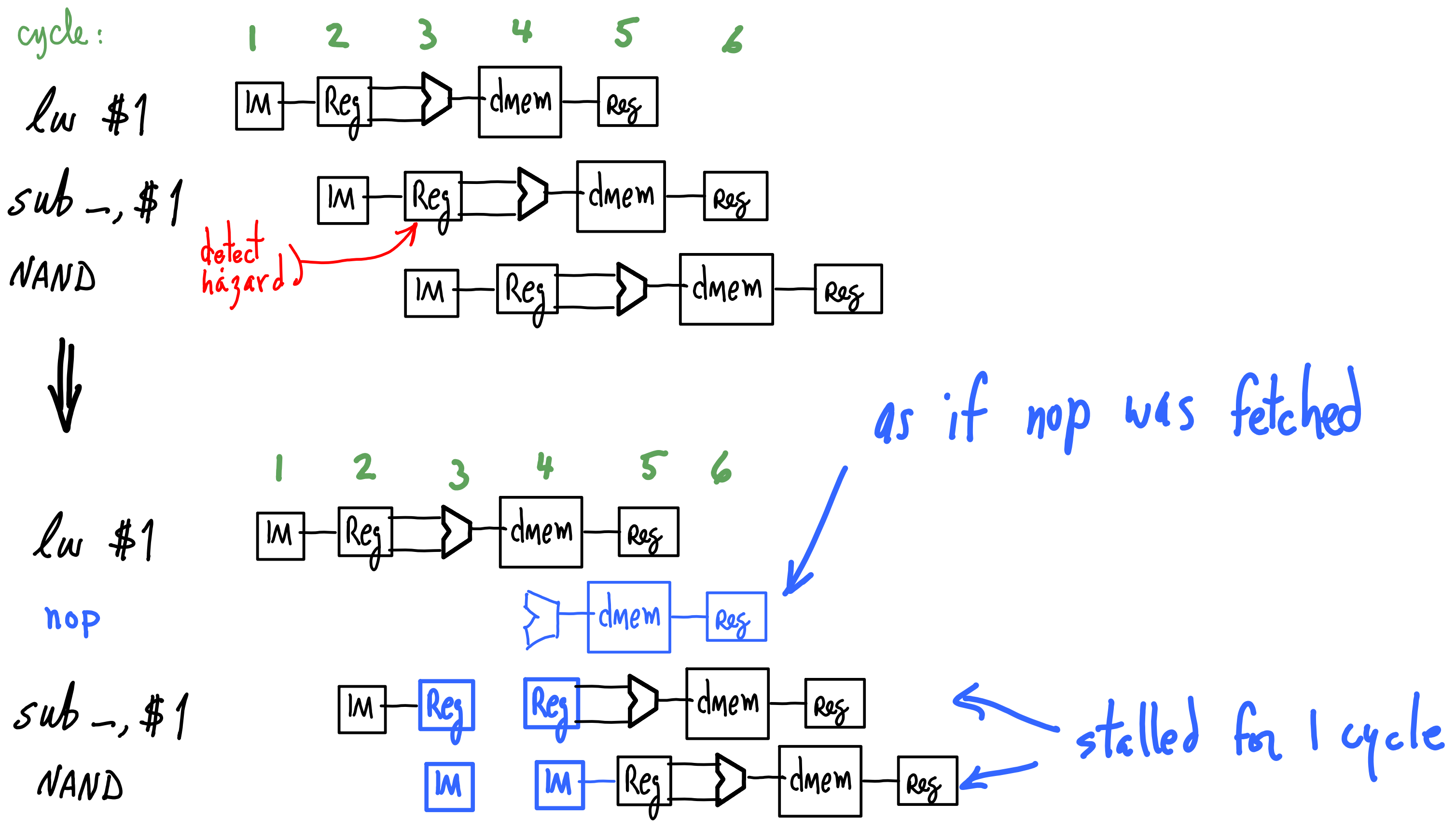


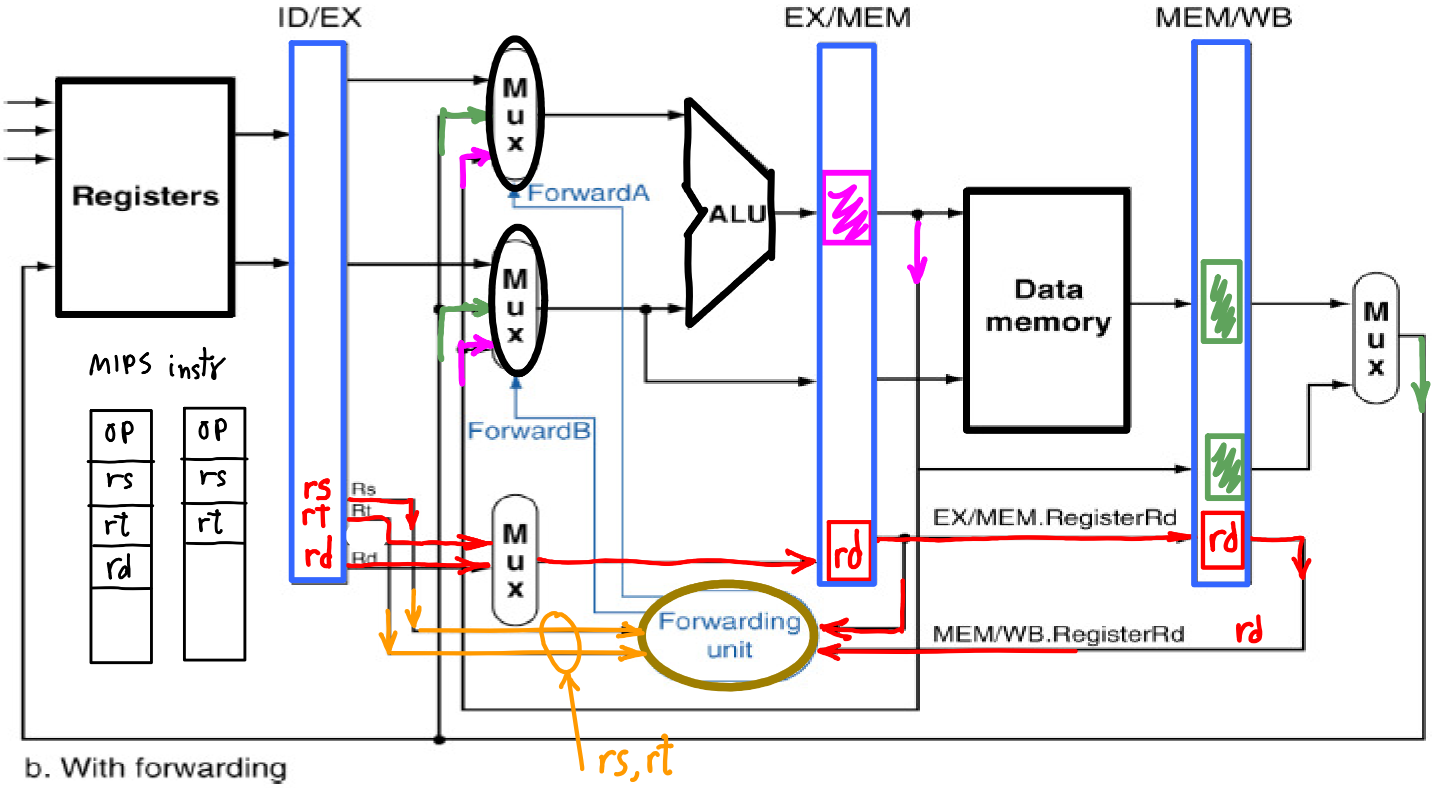


Forward from WB instead, insert NOP



LW stall





b. With forwarding

Forwarding Control: ID/EX.(rs, rt) =? (EX/MEM.rd or MEM/WB.rd)

===> Set MUXes

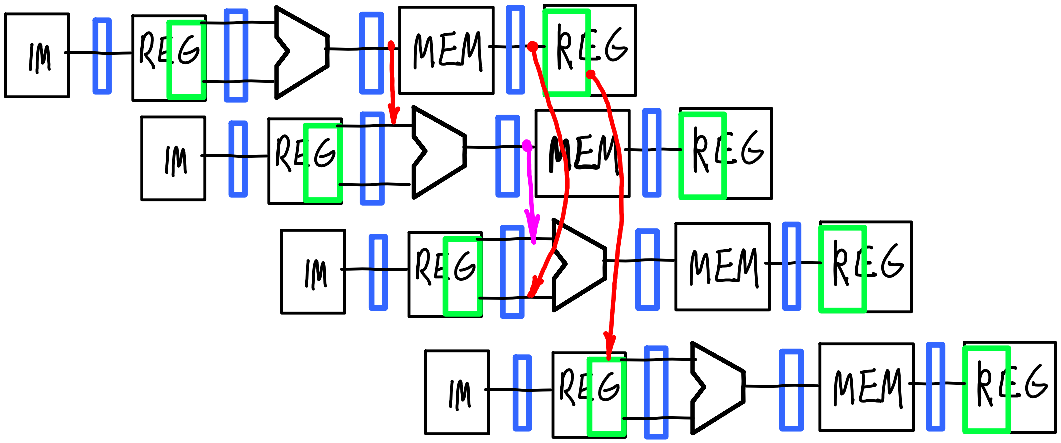
multiple feedback at once?

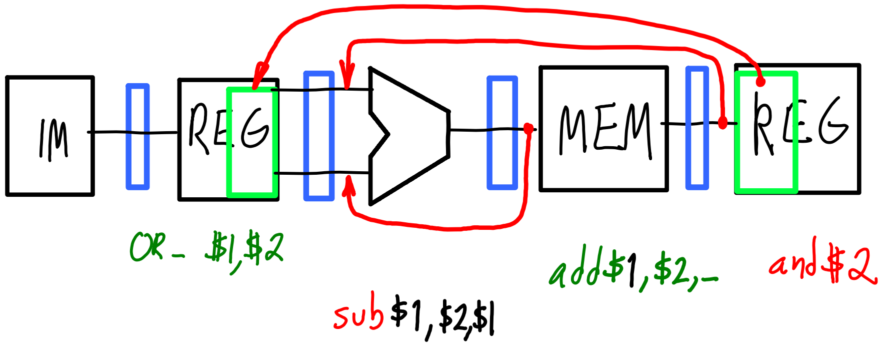
AND $2, _, _

ADD $1, $2, _

SUB _, $2, $1

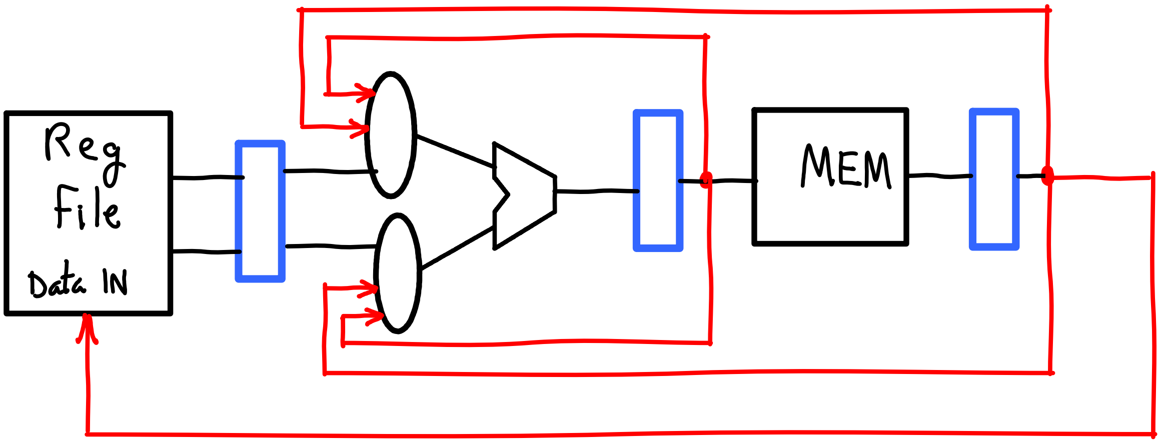
OR _, $1, $2





Can we demonstrate that there aren't any structural hazards for forwarding paths for operate instructions?

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