lecture 15

MIPS data path and control 3

Multicycle model: Pipelining

March 7, 2016

Pipelining

- factory assembly line (Henry Ford - 100 years ago)
- car wash
- cafeteria
- .....

Main idea: achieve efficiency by minimizing worker/processor idle time

Modern Times (1936) by Charlie Chaplin

https://www.youtube.com/watch?v=DfGs2Y5WJ14

Five stages of a MIPS (CPU) instruction

IF : instruction fetch (from Memory)

ID : instruction decode & register read

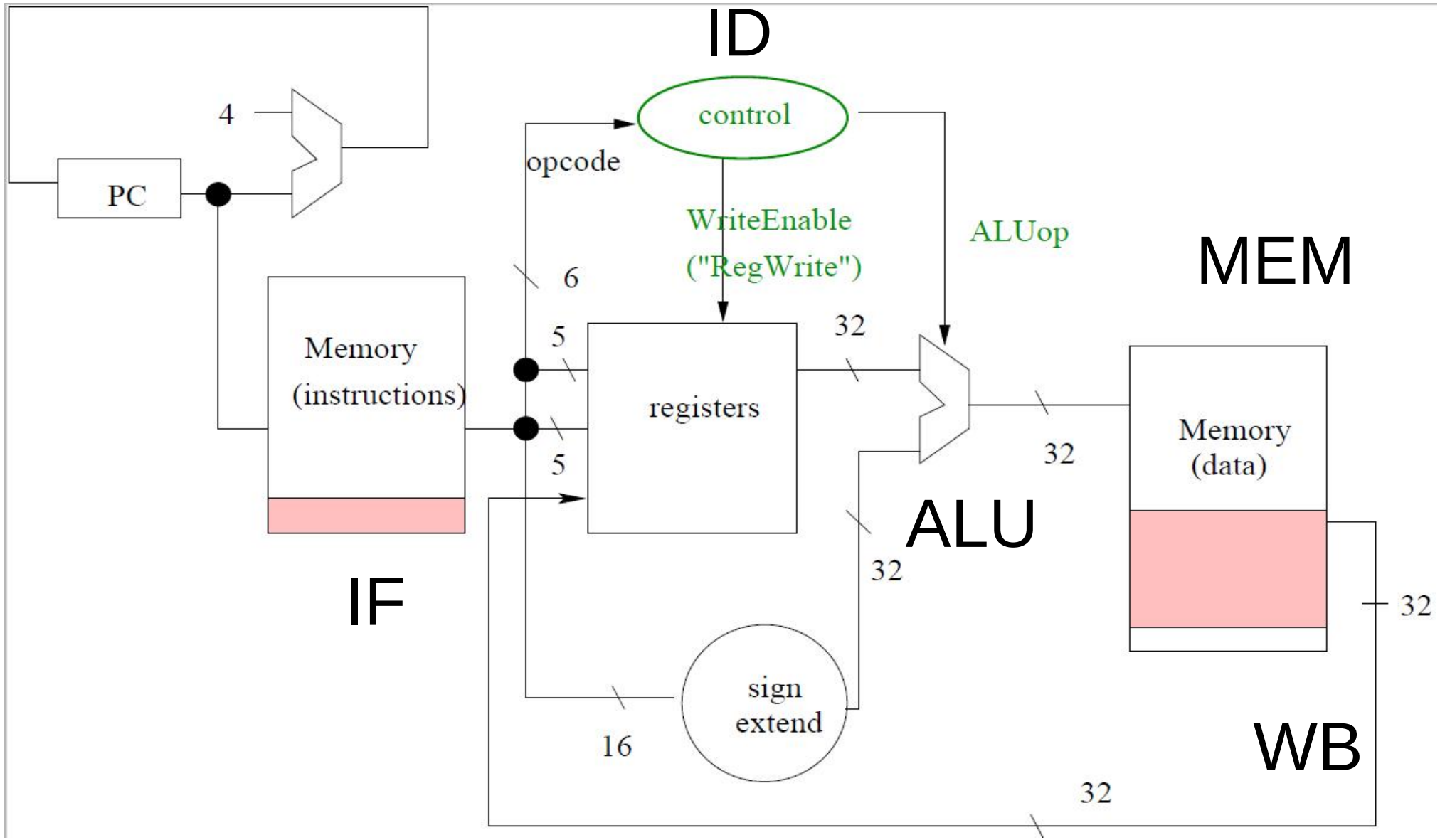
ALU : ALU execution

MEM : Memory access (data: read or write)

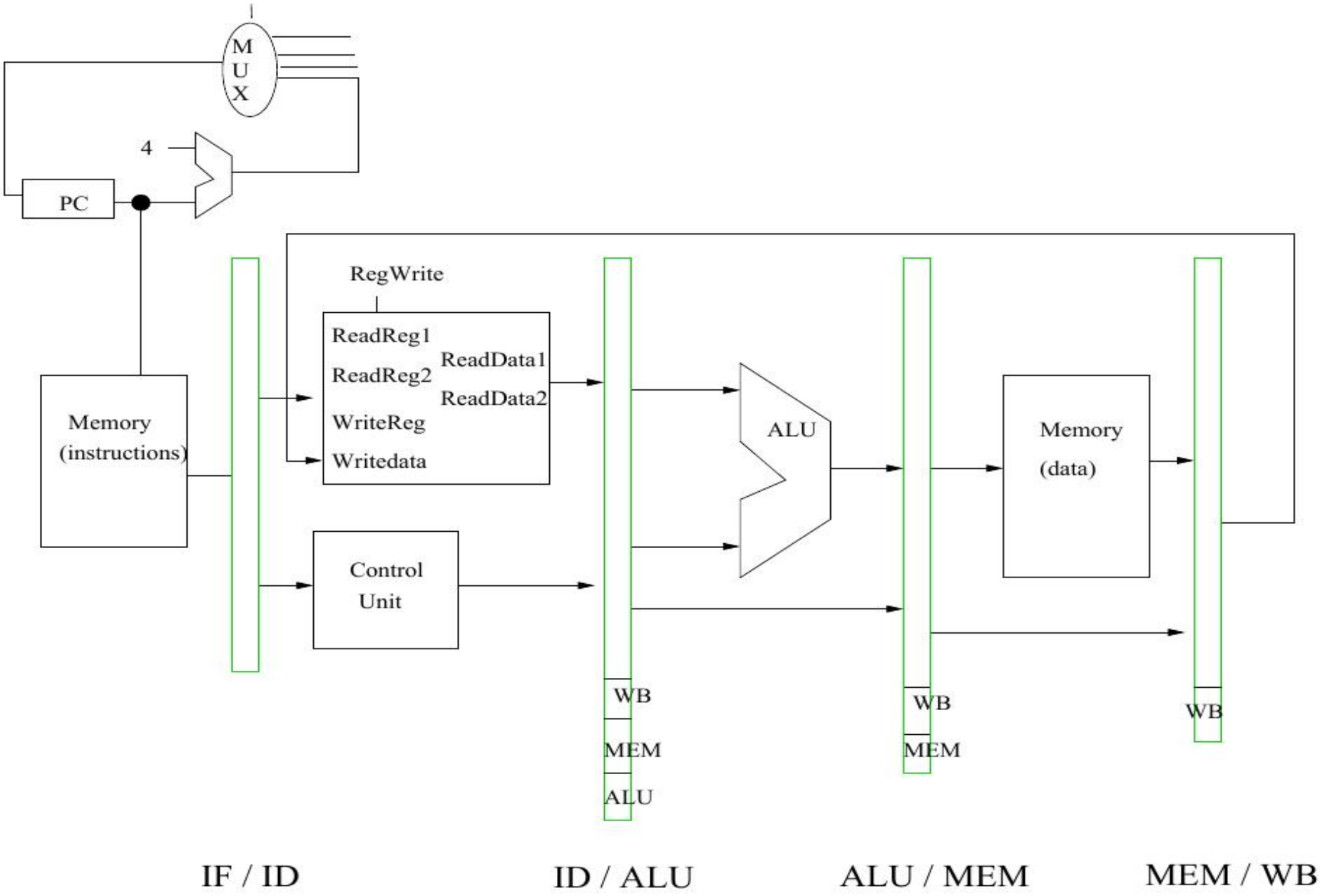
WB : write back into register

With pipelining, rather than completing all stages in a single clock cycle, one stage is completed in each clock cycle.

Recall single cycle model (e.g. load word, lw)



For pipelining, we use extra registers to keep track of "state" information between pipeline stages. All necessary instruction information is stored (including controls, value(s) read from register(s), values computed by ALU)



Pipeline registers

IF/ID : contains the instruction

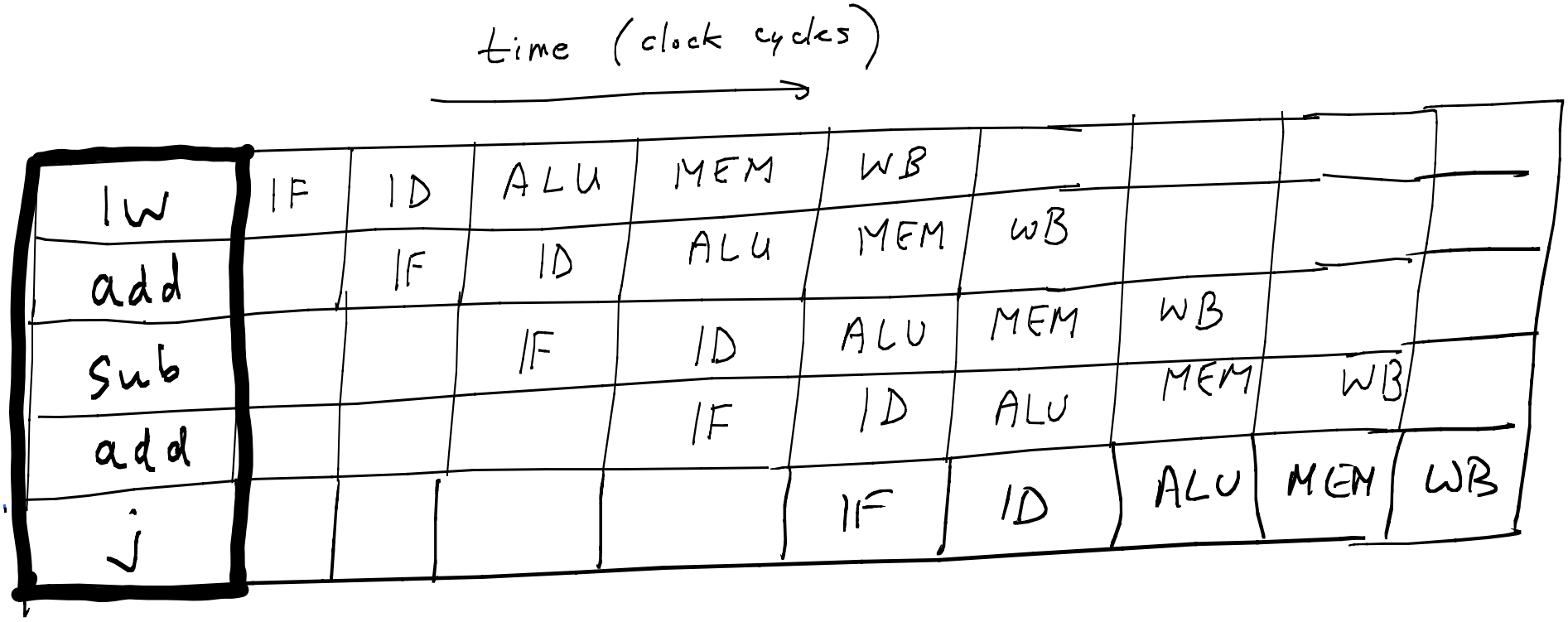
ID/ALU: contains controls that can be computed from instruction such as ALUop, and controls for following three stages (ALU, MEM, WB)

ALU/MEM: contains ALU results, and controls for MEM, WB

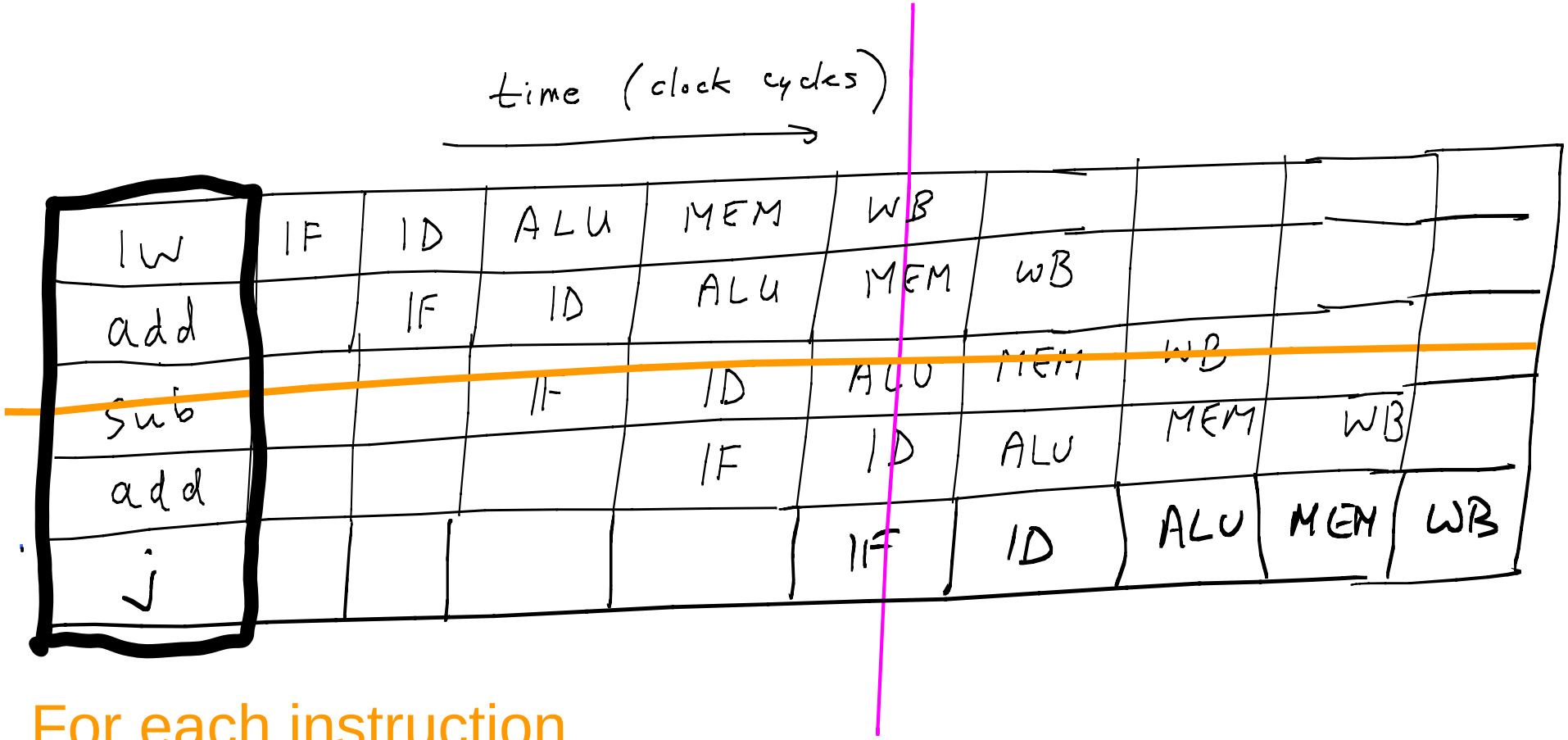
MEM / WB: value read from Memory, control for WB

Each of the 4 pipeline registers is updated at the end of each clock cycle.

Each instruction goes through all 5 stages of the pipeline.



Pipelining gives a potential for 5x speedup relative to single cycle model. Why?



For each instruction, which stage is executed in each clock cycle?

For each clock cycle, which instructions is in each stage of the pipeline?

Some instructions use all of the pipeline stages e.g. lw

but some use only some of the pipeline stages

e.g. add, sw, j

Which stages do nothing?

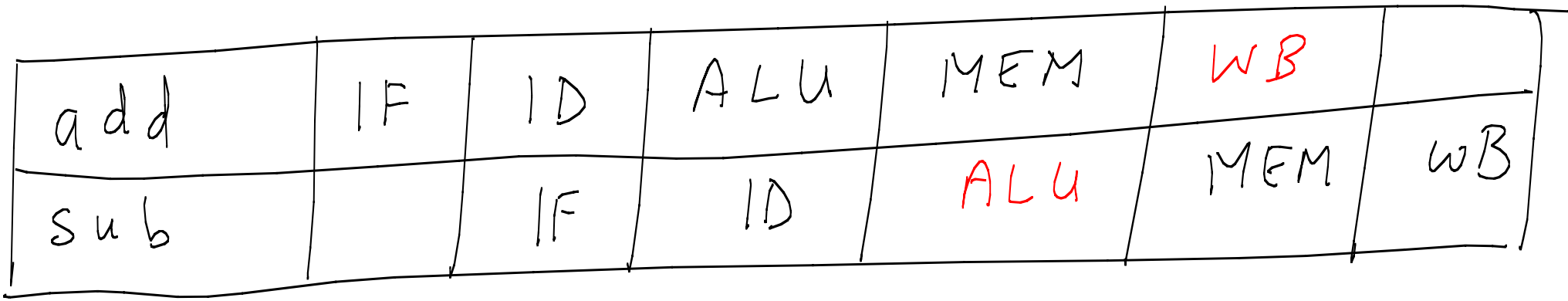
Pipelining Hazards (sketch only)

- data hazards

- control hazards

Data Hazard: Example 1

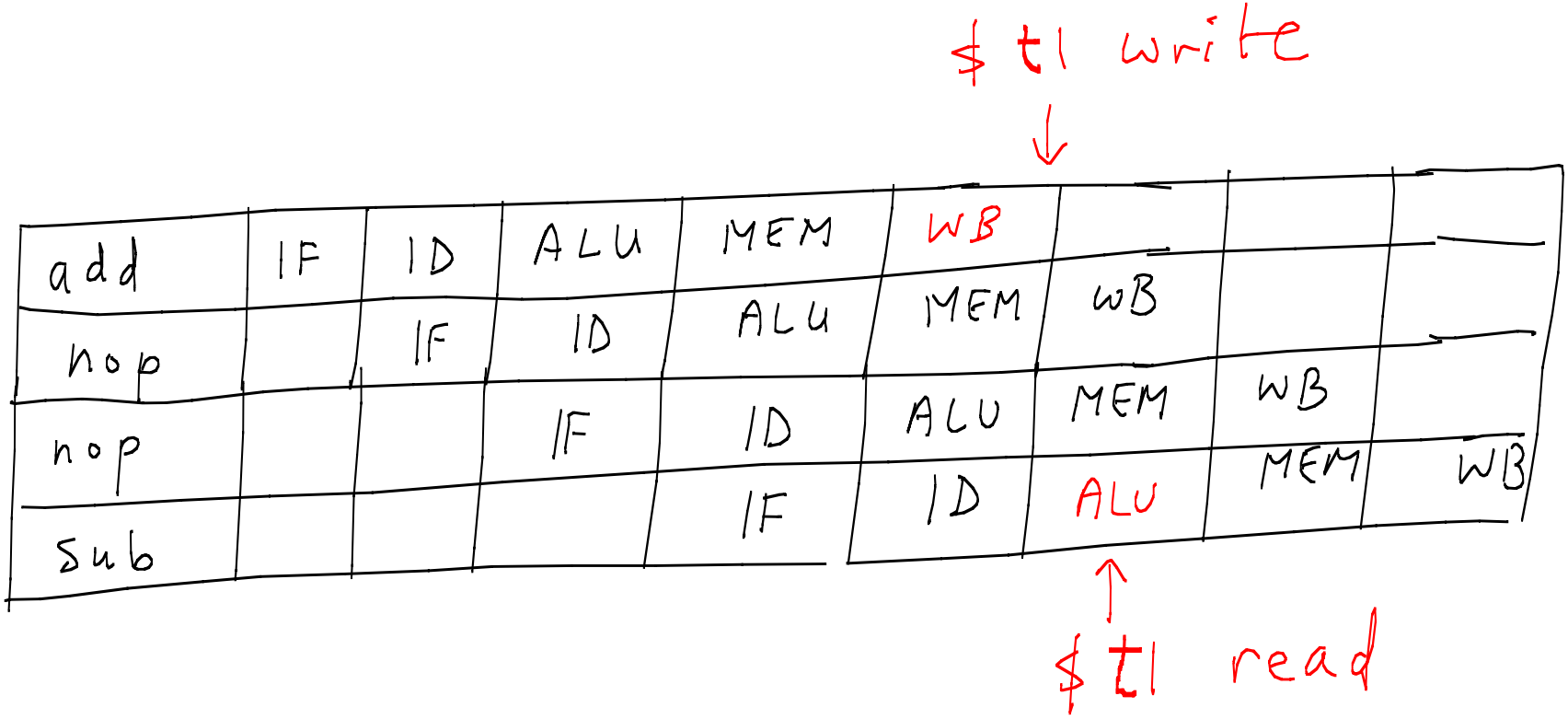
```
add   $t1,  $s2,  $s5
sub   $s1,  $t1,  $s3
```



Solution 1: "stall"

'nop' is a MIPS instruction that does nothing

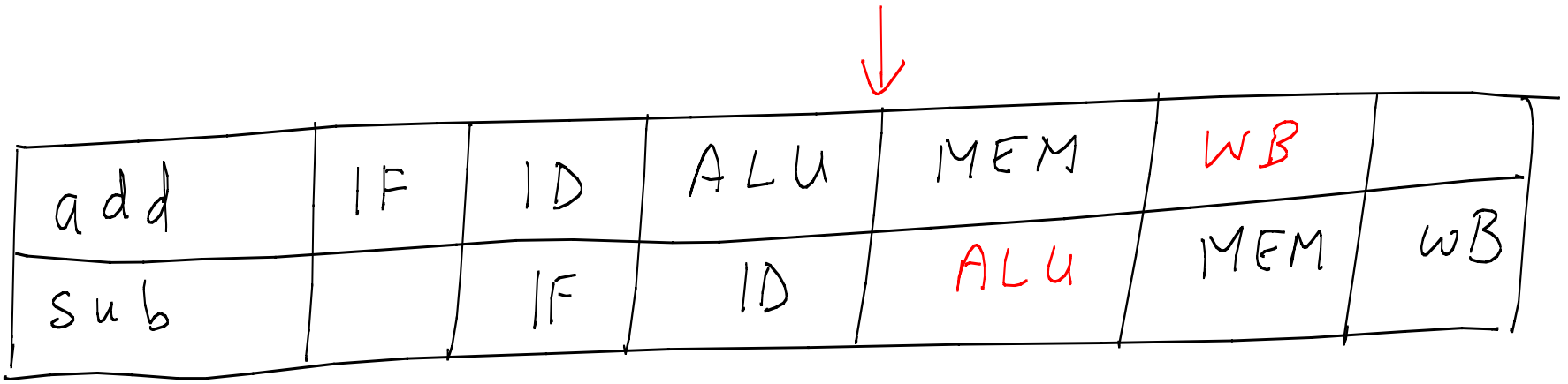
```
add   $t1,  $s2,  $s5
nop
nop
sub   $s1,  $t1,  $s3
```



Solution 2: "data forwarding"

```
add   $t1,  $s2,  $s5
sub   $s1,  $t1,  $s3
```

The result of the 'leading' instruction (add) has been computed by end of its ALU stage and is written into the ALU/MEM register (short cut).



The result is used by the 'trailing' instruction (sub) in its ALU stage.

What does circuit look like for data forwarding ?

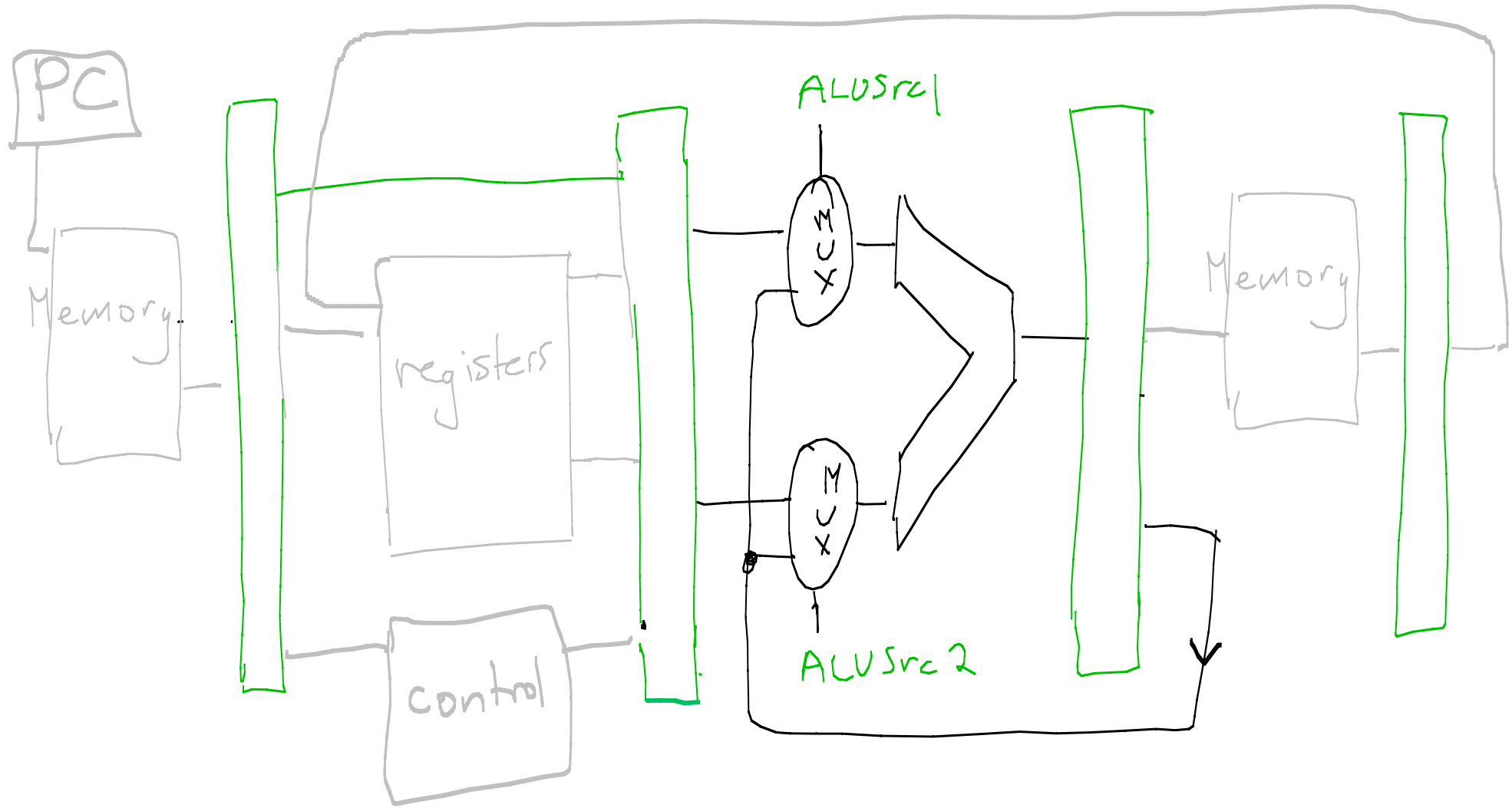
Note that data hazard can occur for either (or both) of the source registers in the trailing instruction.

```
add   $t1,  $s2,  $s5
sub   $s1,  $t1,  $s3
```

"Forward" the data computed by the leading instruction (add) to the ALU where is used by the trailing instruction (sub).

This data is used, but it is not yet written in the $t1 register.

sub add



IF ID ALU MEM
WB

How can these ALUsrc control signals be defined ?

e.g. "leading" instruction in the MEM stage
"trailing" instruction in the ALU stage

Data forwarding condition:

ALUsrc1 = ALU/MEM.RegWrite and
(ID/ALU.rs == ALU/MEM.rd)

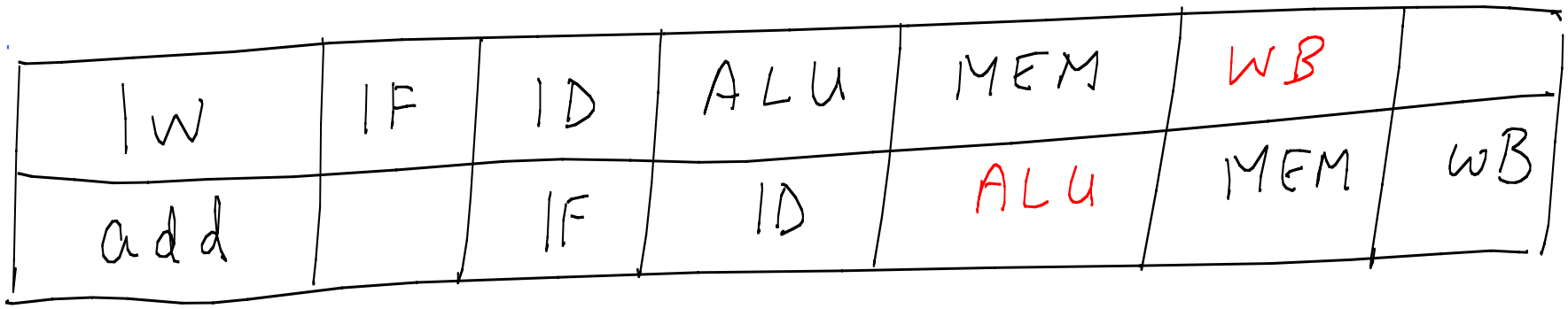
ALUsrc2 = ALU/MEM.RegWrite and
(ID/ALU.rt == ALU/MEM.rd)

Note that both of these conditions can be true e.g.

```
add  $t1, $s2, $s5
sub  $s1, $t1, $t1
```

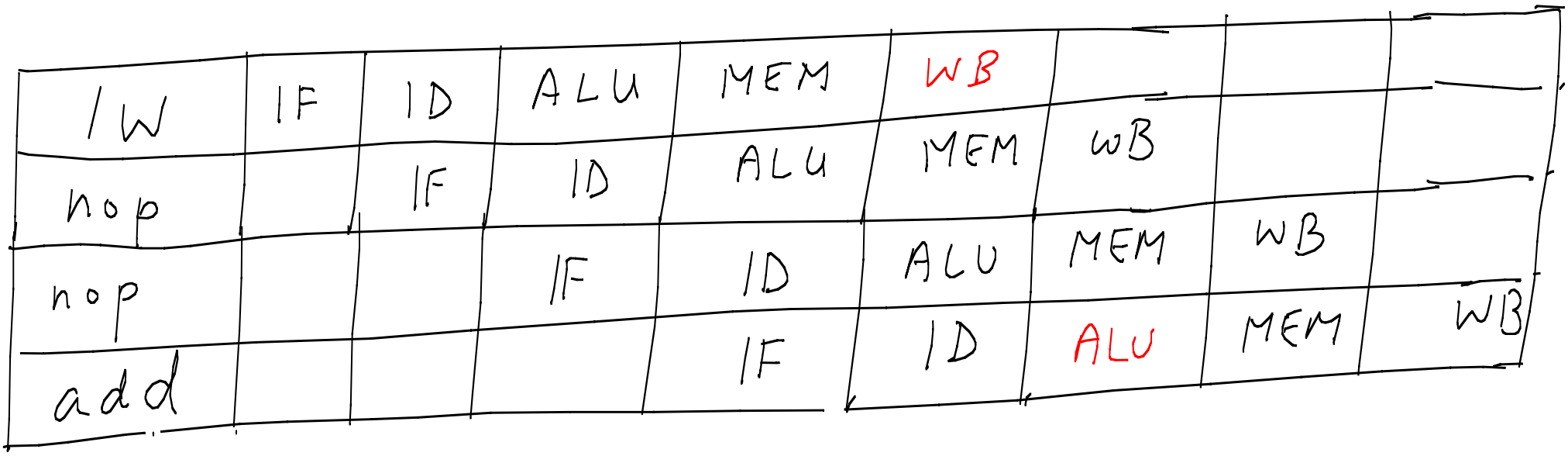
Data Hazard: Example 2

```
lw    $s1, 24( $s0 )
add   $t0, $s1, $s2
```



How is this similar to (and different from) the previous example ?

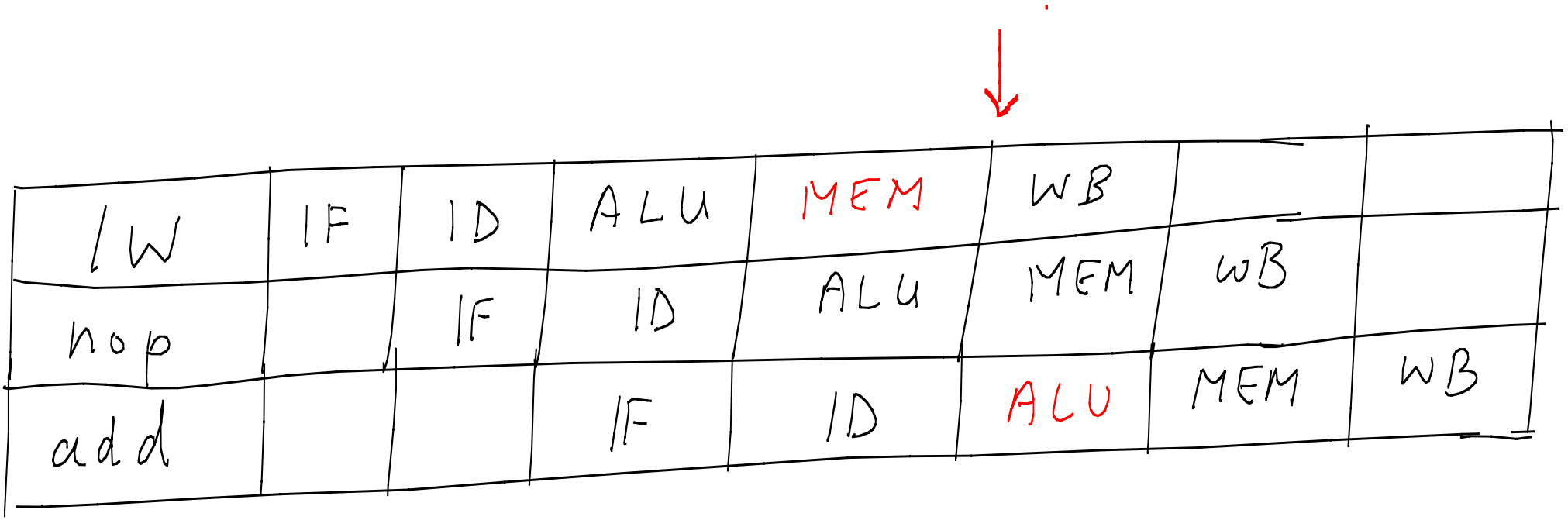
Solution 1: "stall"



Solution 2: "data forwarding"

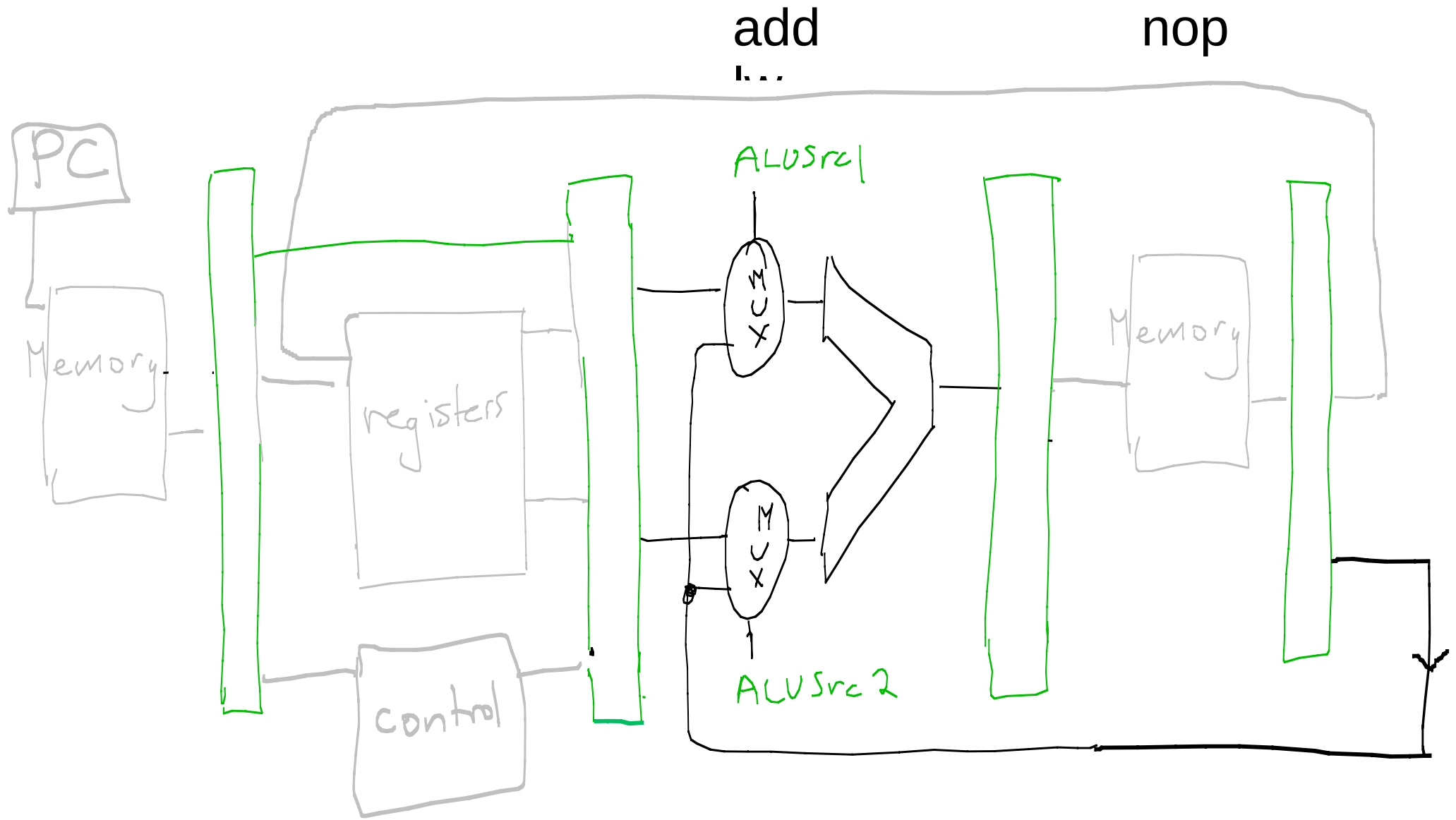
Insert one nop (no operation) instruction.

In the "leading" instruction (lw), a word is read from Memory and is written into the MEM/WB register. In the next clock cycle, that word can be forwarded to the ALU stage of the "trailing" instruction (addi) .



In the next few slides, I will give a data forwarding solution that is similar to the one I gave earlier.

The two solutions would need to be integrated, but let's ignore that fact and treat this second instance of data forwarding on its own.



"Forward" the data computed by the leading instruction (lw) directly into

In this case, data forwarding can be done when:

ALUsrc1 = MEM/WB.RegWrite and
(ID/ALU.rs == MEM/WB.rd)

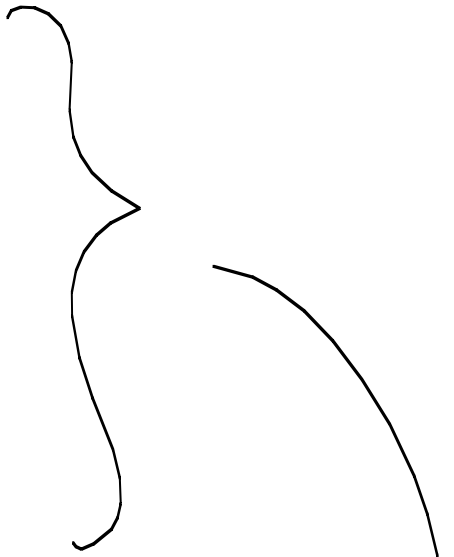
ALUsrc2 = MEM/WB.RegWrite and
(ID/ALU.rt == MEM/WB.rd)

Again, both of these conditions can be true.

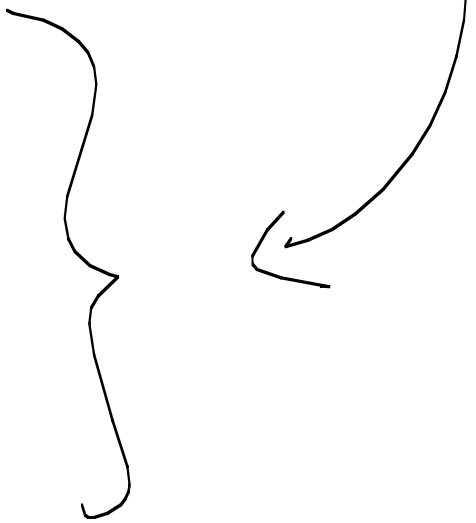
```
lw   $t1, 0($s2)
add  $s1, $t1, $t1
```

Solution 3: reordering instructions

```
sub  $t3, $t2, $s0
lw   $s1, 24($s0)
add  $t0, $0,  $s1
or   $s5, $t5, $s0
```



```
lw   $s1, 24($s0)
sub  $t3, $t2, $s0
or   $s5, $t5, $s0
add  $t0, $0,  $s1
```



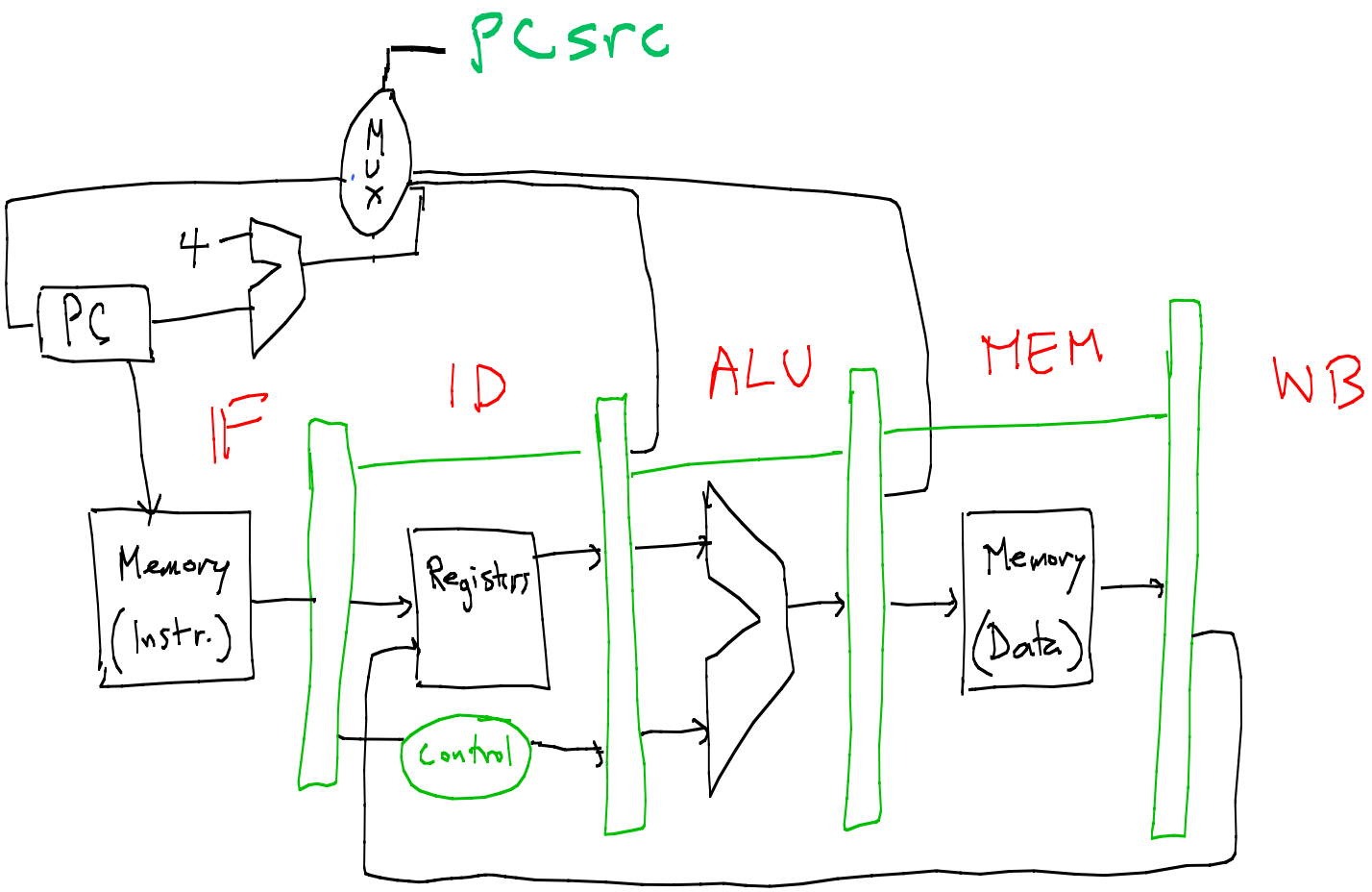
Pipelining Hazards (sketch only)

- data hazards

- control hazards

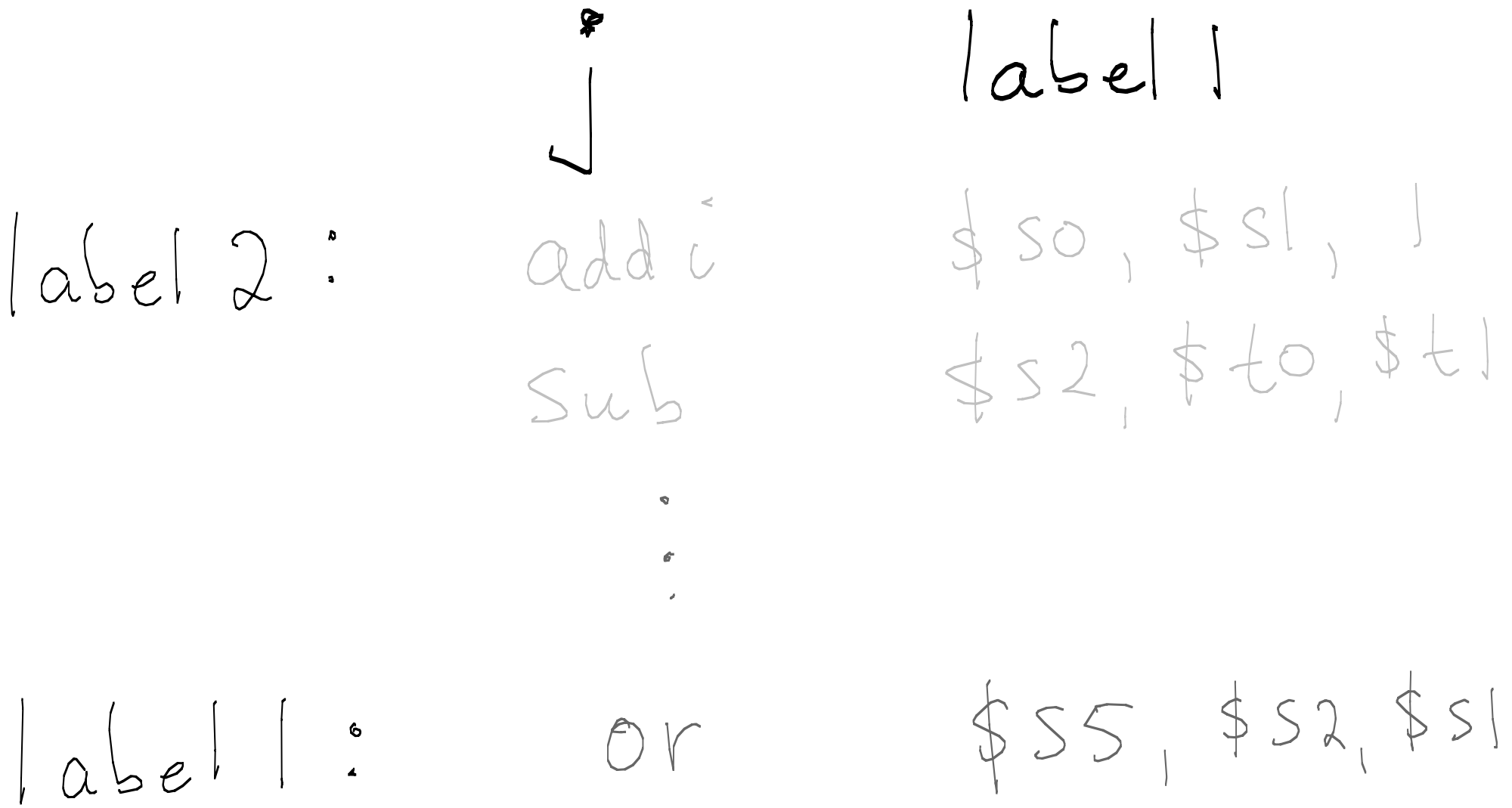
  - unconditional branches
  - conditional branches

How to handle branches ?

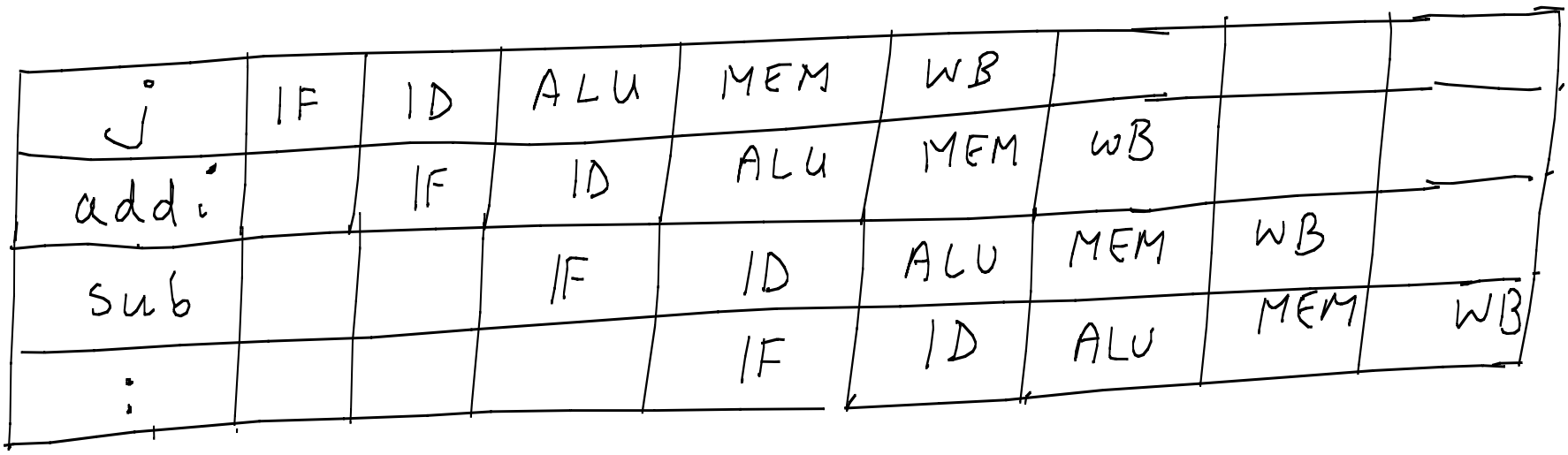


What is the general problem?
Default is PC <--- PC+4 on every clock cycle (IF).
Thus, next instruction enters pipeline (hazard!)
PCsrc cannot be determined at IF stage.

Control Hazard: Example 1

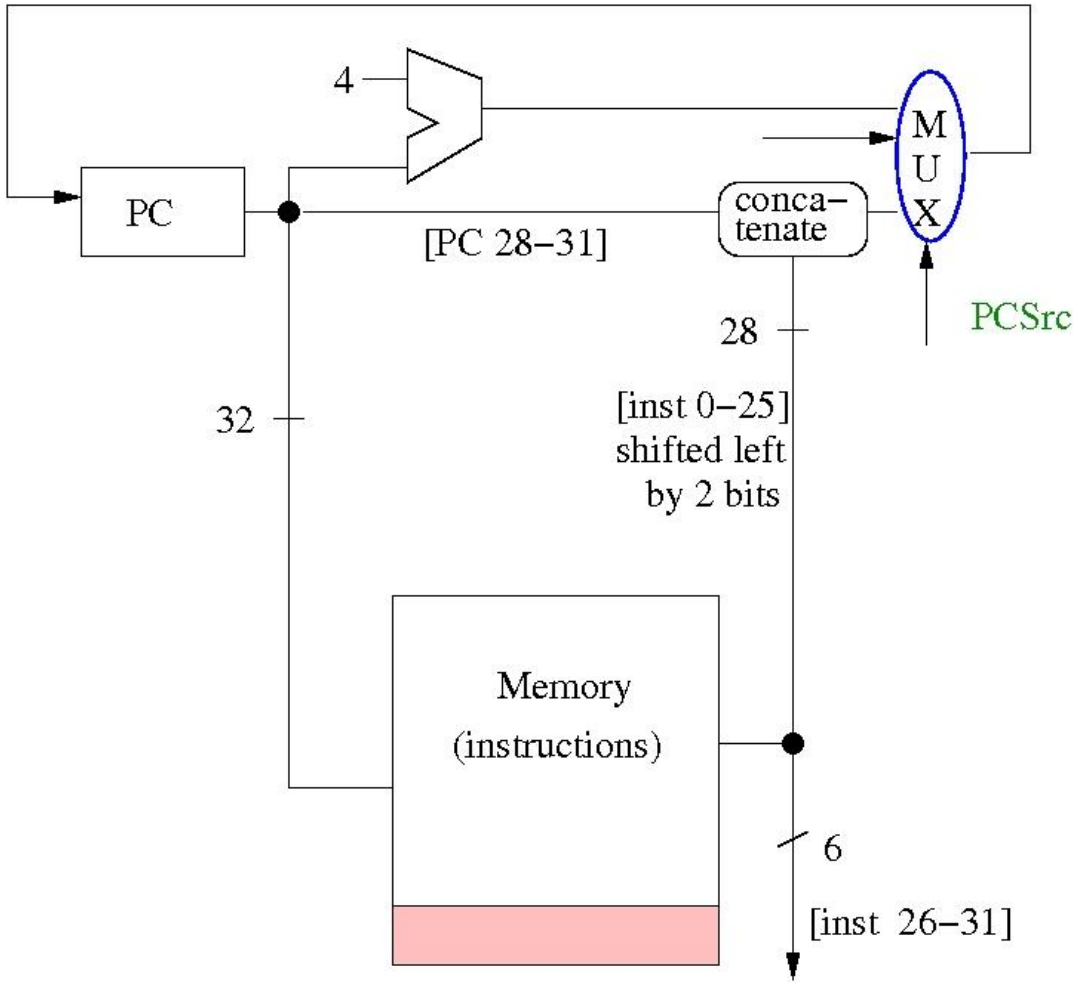


```
        j       label1
label2: addi    $s0, $s1, 1
        sub     $s2, $t0, $t1
        :
label1: or      $s5, $s2, $s1
```



The trailing instruction (addi) enters the pipeline but it should not be executed. (It can only be executed if you branch to label2 from somewhere else in code).

Recall lecture 14 (single cycle model)

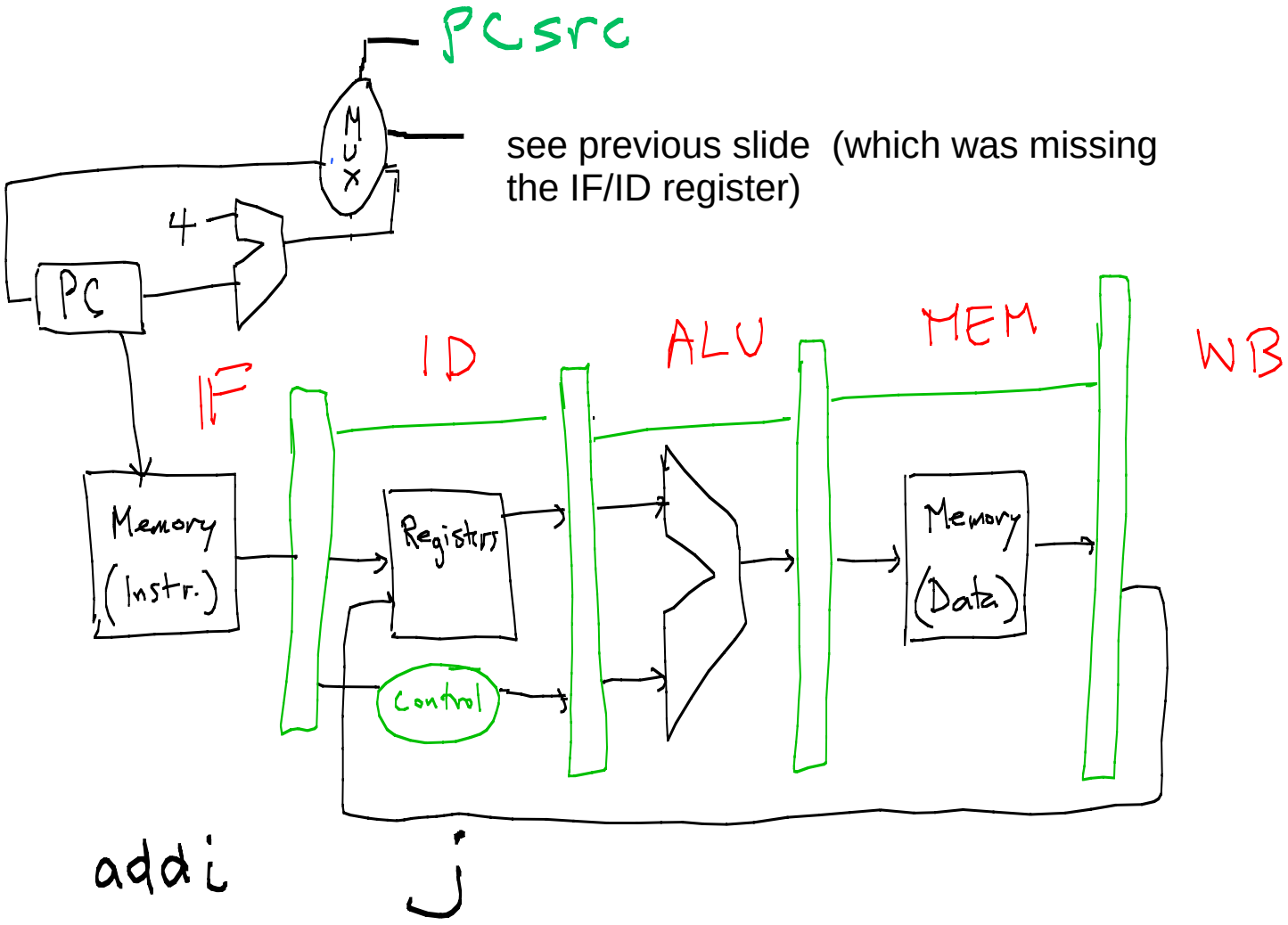


Solution ?

Observe that:

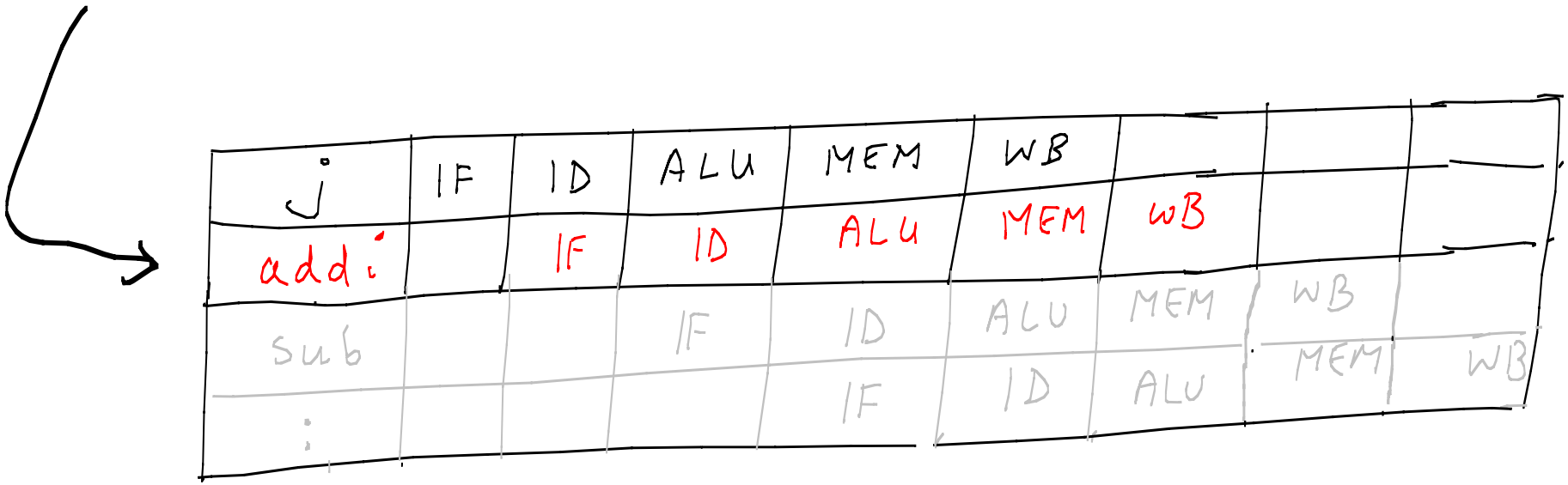
- jump can be detected in the ID stage
- PCsrc can be determined at the end of jump's ID stage

Inserting a 'nop' after 'j' would work.



Slightly different solution: replace (at runtime) the instruction that follows the jump with a 'nop'. This has equivalent effect of inserting a 'nop' into the program.

if IF/ID. instruction == j // current clock cycle
then IF/ID. instruction = nop // next clock cycle

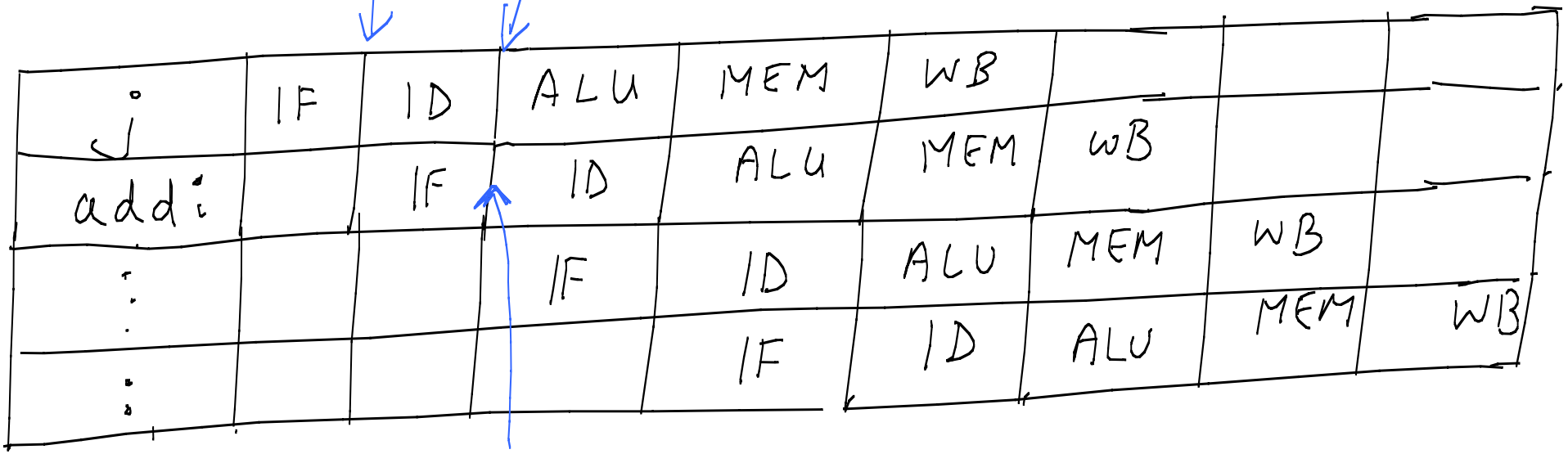


```
label2:   j     label1
          addi  $s0, $s1, 1
          :

label1:
```

PC <-- PC+4

PC <-- label1



IF/ID.inst = nop

Control Hazard: Example 2

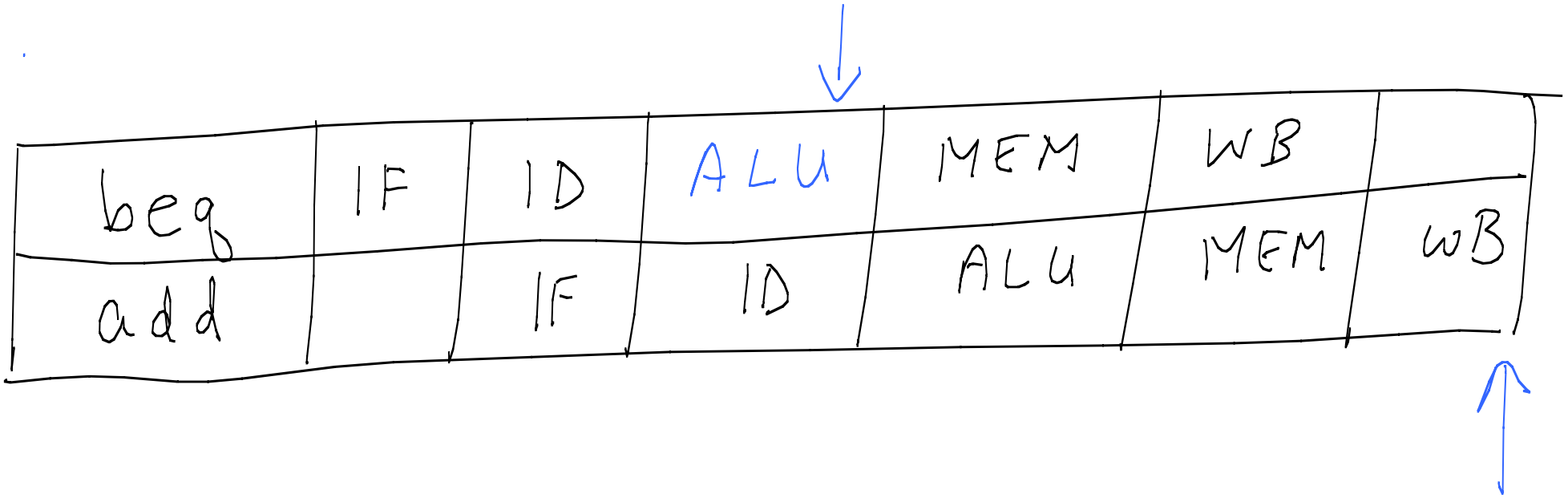
```
beq  $s1, $s3, label
add  $t0, $s3, $s4
```

Sometimes the trailing instruction (add) is executed.

Sometimes not.

Solution ?

Here is where PCsrc is determined (for beq).
PC potentially could take the branch at the end of this clock cycle.



here is where 'add' writes
(and could do its damage)

Solution ?

- stall (insert 2 nop's)

- reorder if possible to reduce the number of nop's (see Exercises)

- set the RegWrite control of the trailing instruction (add) to off, if the branch condition is true