
CMPE011 Fall 2003

Pipeline hazards

- Structural hazards
- Data hazards
- Control (branch) hazards
- Branch prediction

Textbook: 6.4, 6.5, 6.6



What are “hazards”?

Hazards are situations that prevent the next instruction(s) in the instruction stream from executing during its (their) designated clock cycle.

There are three kinds of hazard:

- structural hazards — caused by conflicts in resource usage
- data hazards — caused by instructions that depend on previous instructions' outcome
- control (branch) hazards — caused by pipelining of instructions that change the PC



Structural hazards

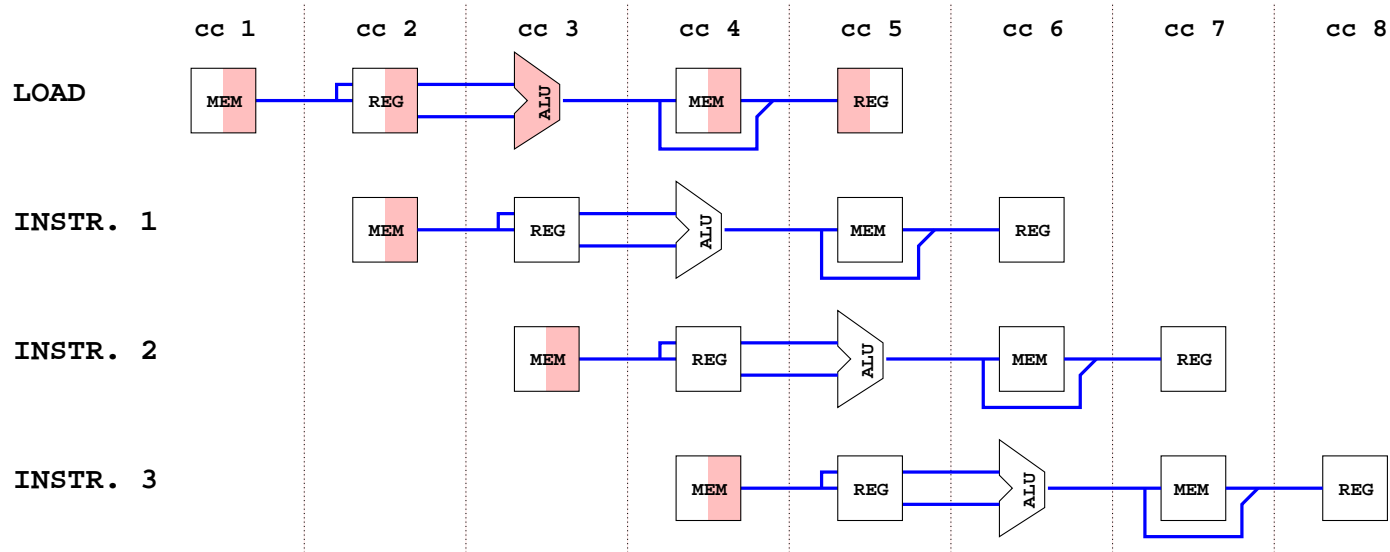
Overlapping instructions execution requires duplication of resources to allow all possible combinations of instructions in the pipeline.

The machine is said to have a *structural hazard* if some combinations of instructions can not be accomodated.



Structural hazards

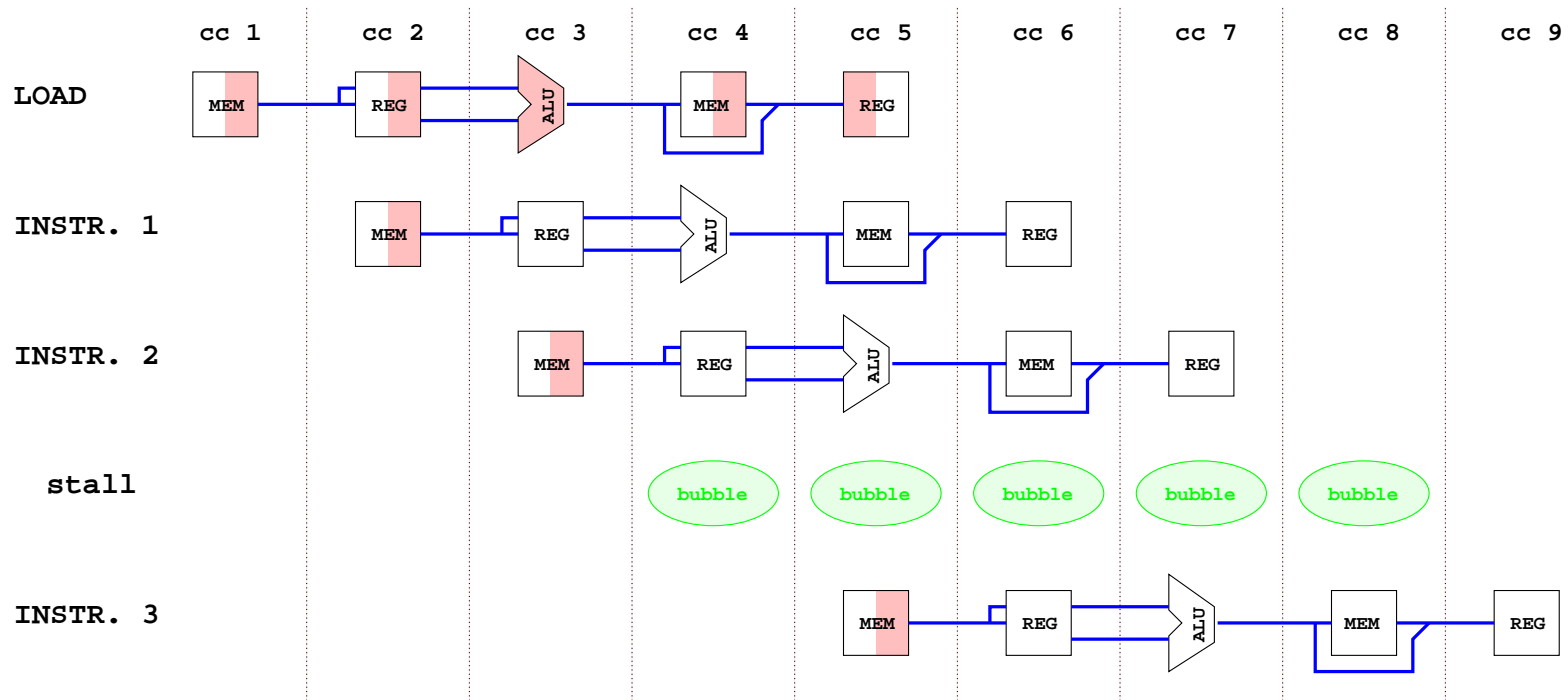
Example: if our CPU had a single memory for instructions and data, then we could not read the PC while a load instruction is accessing the memory.



Structural hazards

What does the CPU do in case of a structural hazard?

The pipeline is *stalled* (a *bubble* is inserted).



Data hazards

A major effect of pipelining is to change the relative timing of instructions by overlapping their execution.

Data hazards occur when the pipeline changes the order of read/write accesses to operands with respect to the order in the unpipelined machine.

Two kinds:

- those that can be resolved with *forwarding*
- those that require pipeline stalls



Example:

sub \$2, \$1, \$3

and \$12, \$2, \$5

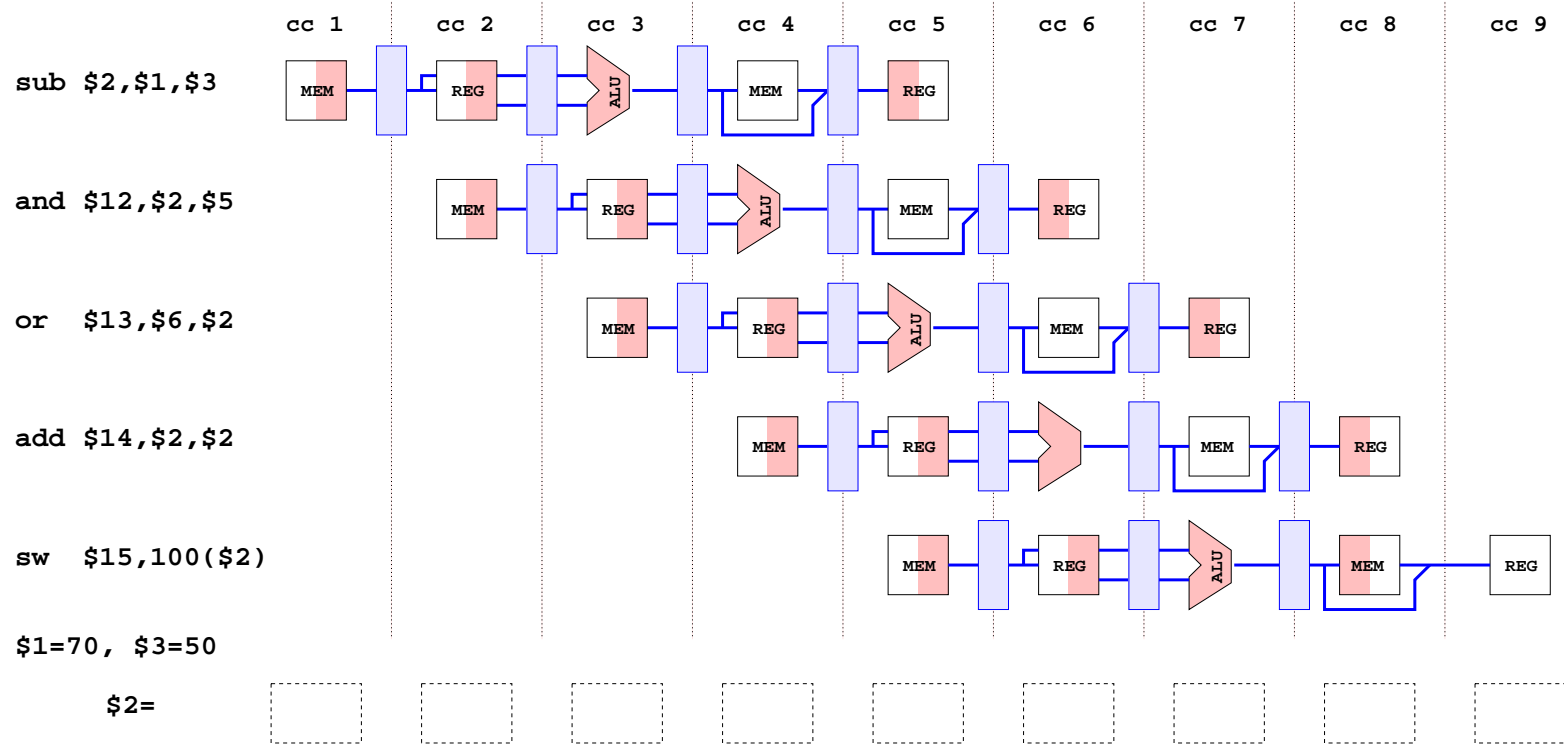
or \$13, \$6, \$2

add \$14, \$2, \$2

sw \$15, 100(\$2)

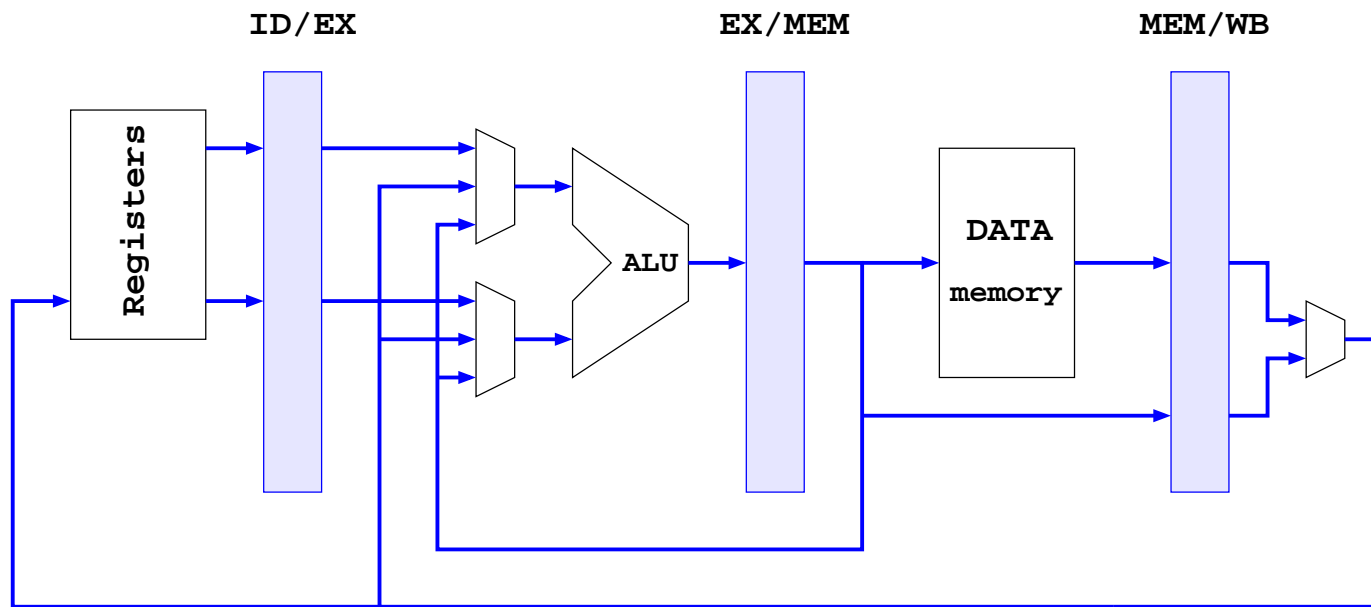


Example (cont.)



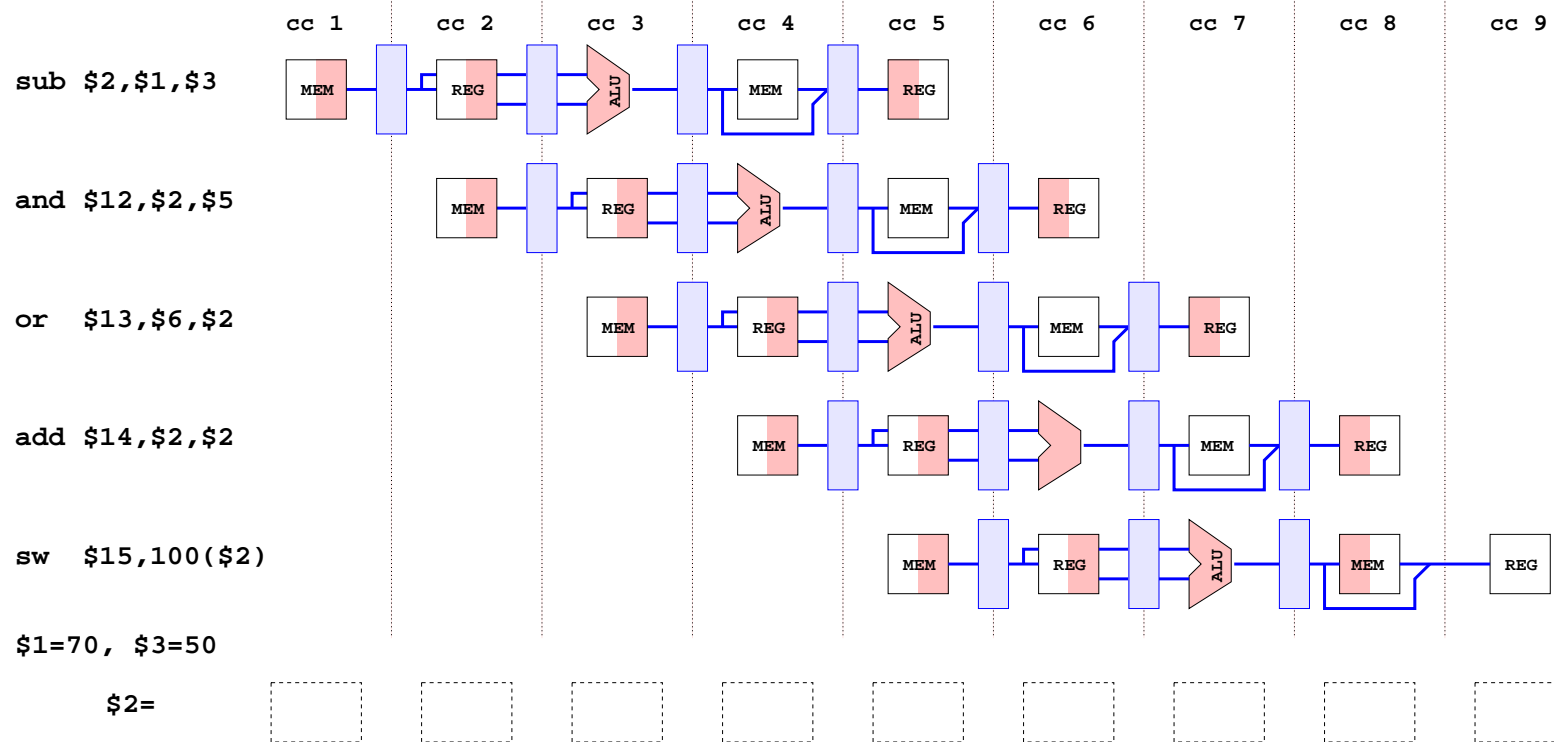
Forwarding or bypassing

The result of an ALU operation is ready at the end of the EX stage, in the EX/MEM pipeline register → no need to wait until it is written into a register to use it as an operand.

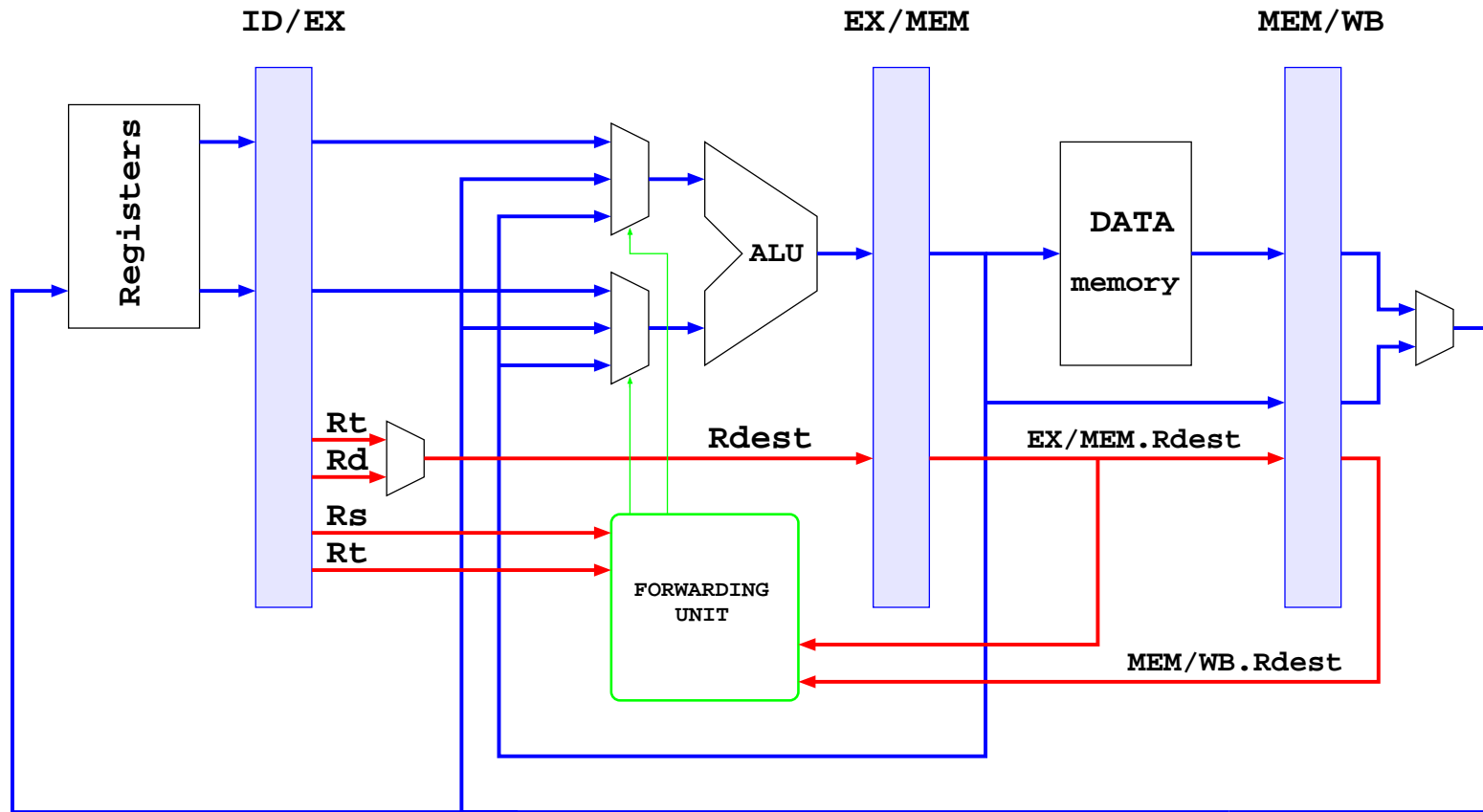


Data hazards

Example (cont.)



Forwarding unit



“Forwarding” hazards summary

“MEM stage” hazards:

add \$1, \$2, \$3

add \$4, \$1, \$5

or

add \$1, \$2, \$3

add \$4, \$5, \$1

condition:

EX/MEM.RegisterRd = ID/EX.RegisterRs or

EX/MEM.RegisterRd = ID/EX.RegisterRt



“WB stage” hazards:

add \$1, \$2, \$3

add \$7, \$7, \$7

add \$4, \$1, \$5

or

add \$1, \$2, \$3

add \$7, \$7, \$7

add \$4, \$5, \$1

condition:

MEM/WB.RegisterRd = ID/EX.RegisterRs or

MEM/WB.RegisterRd = ID/EX.RegisterRt



Pipeline stalls

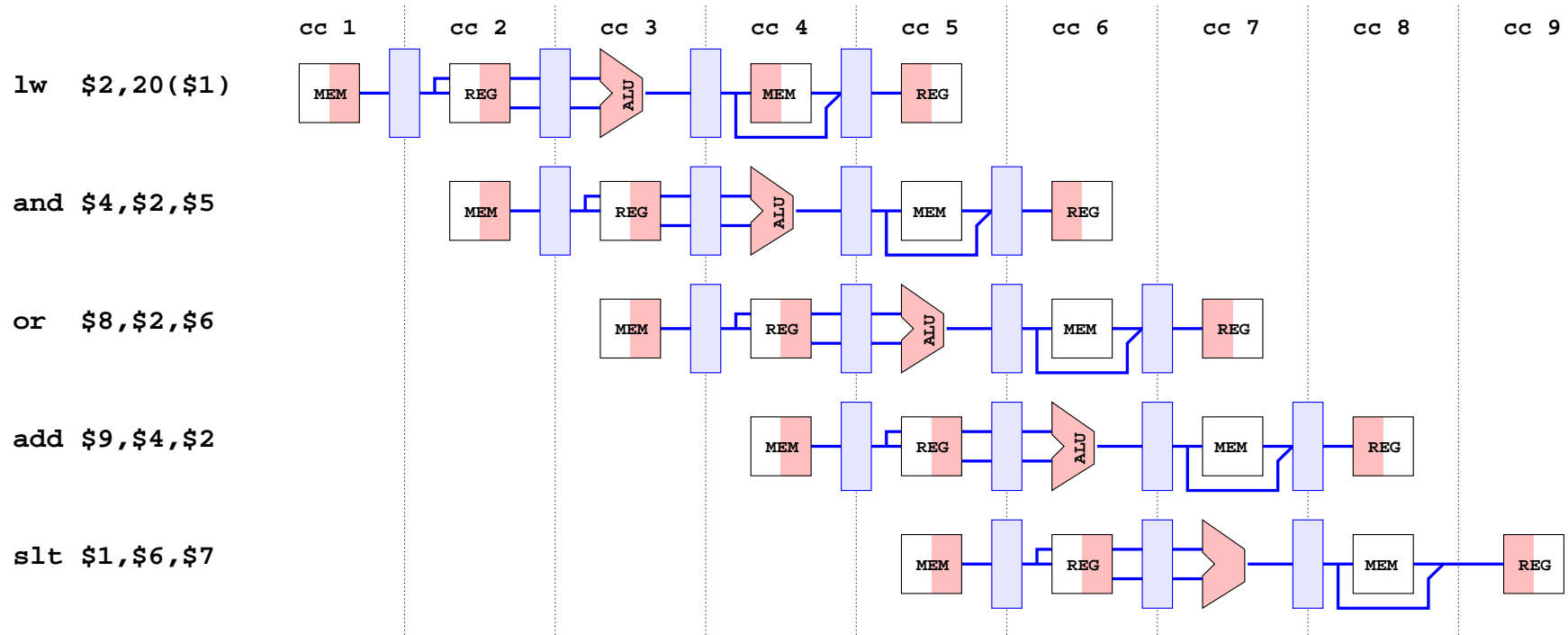
In one case, forwarding cannot prevent a pipeline stall - when an instruction reads a register that is the destination of the preceding LOAD instruction.

Example:

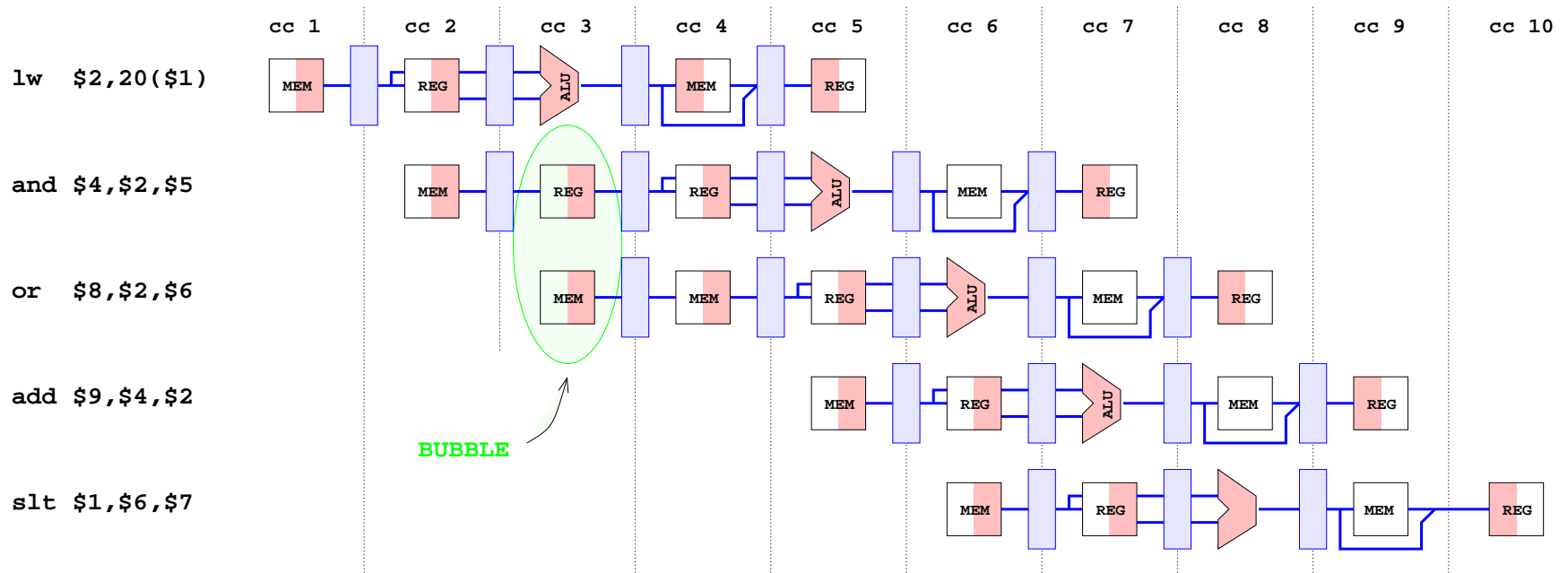
```
lw    $2, 20($1)
and   $4, $2, $5
or    $8, $2, $6
add   $9, $4, $2
slt   $1, $6, $7
```



Example (cont.)

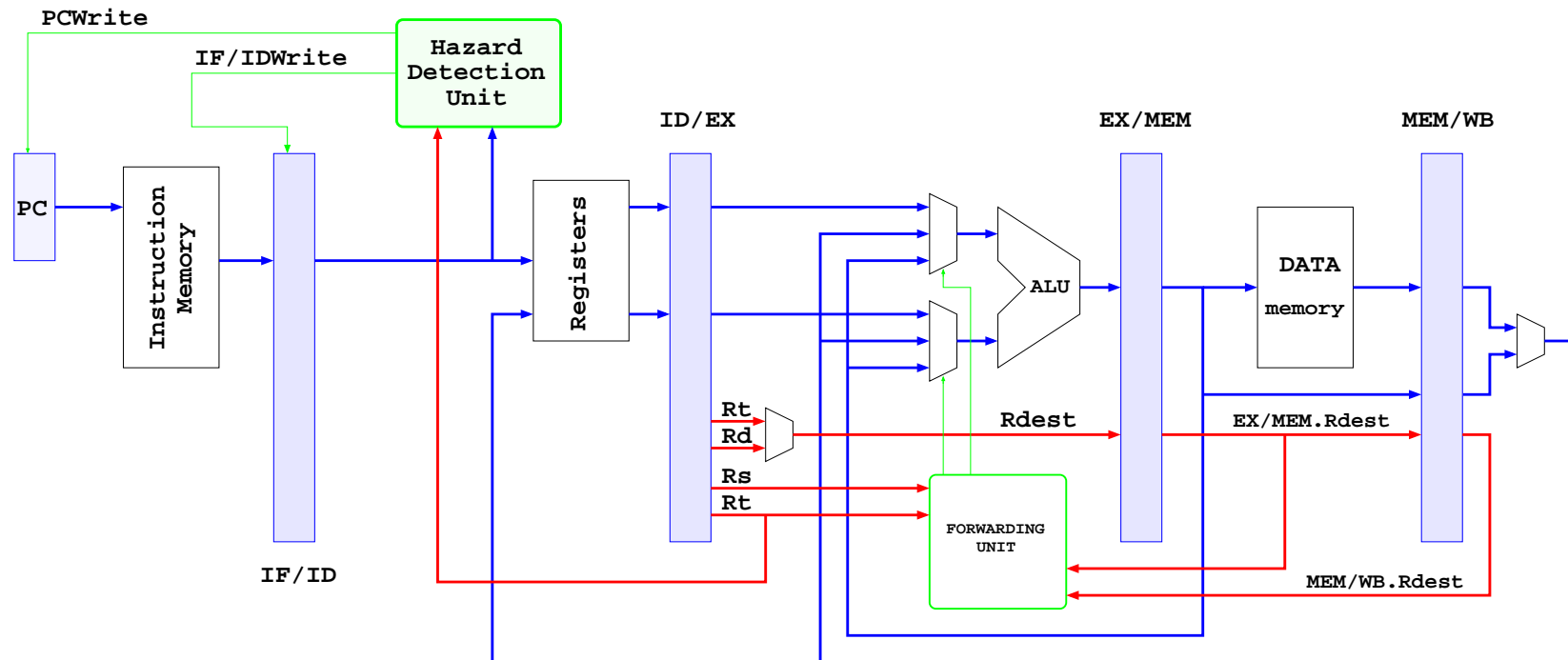


Example (cont.)



Hazard detection unit (or *pipeline interlock*)

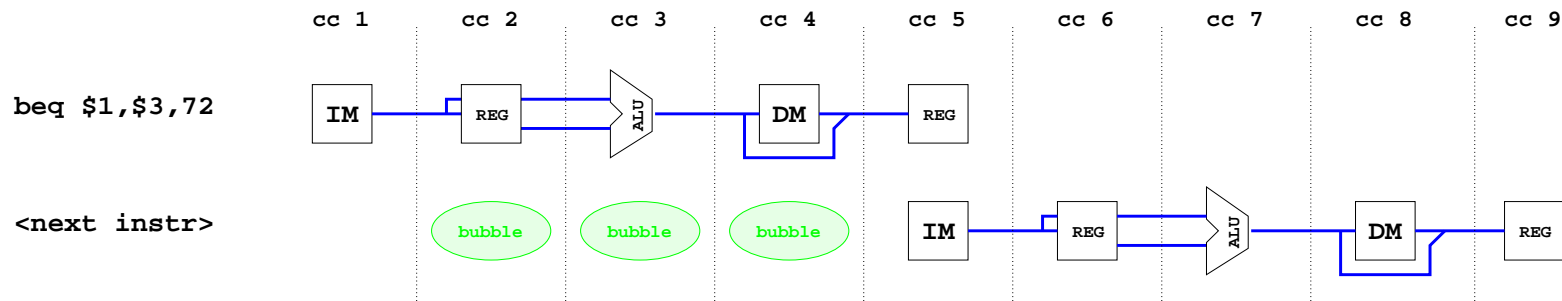
During the ID stage, the hazard detection unit checks if the instruction following a LOAD uses the LOAD's destination register as an operand. If this is the case, the HDU stalls the instruction following the load for one cycle.



Control (branch) hazards

Branch instructions change the execution flow (the PC) based on a condition. Normally, the condition is evaluated in the EX stage, “known” in the MEM stage, and used in the next stage.

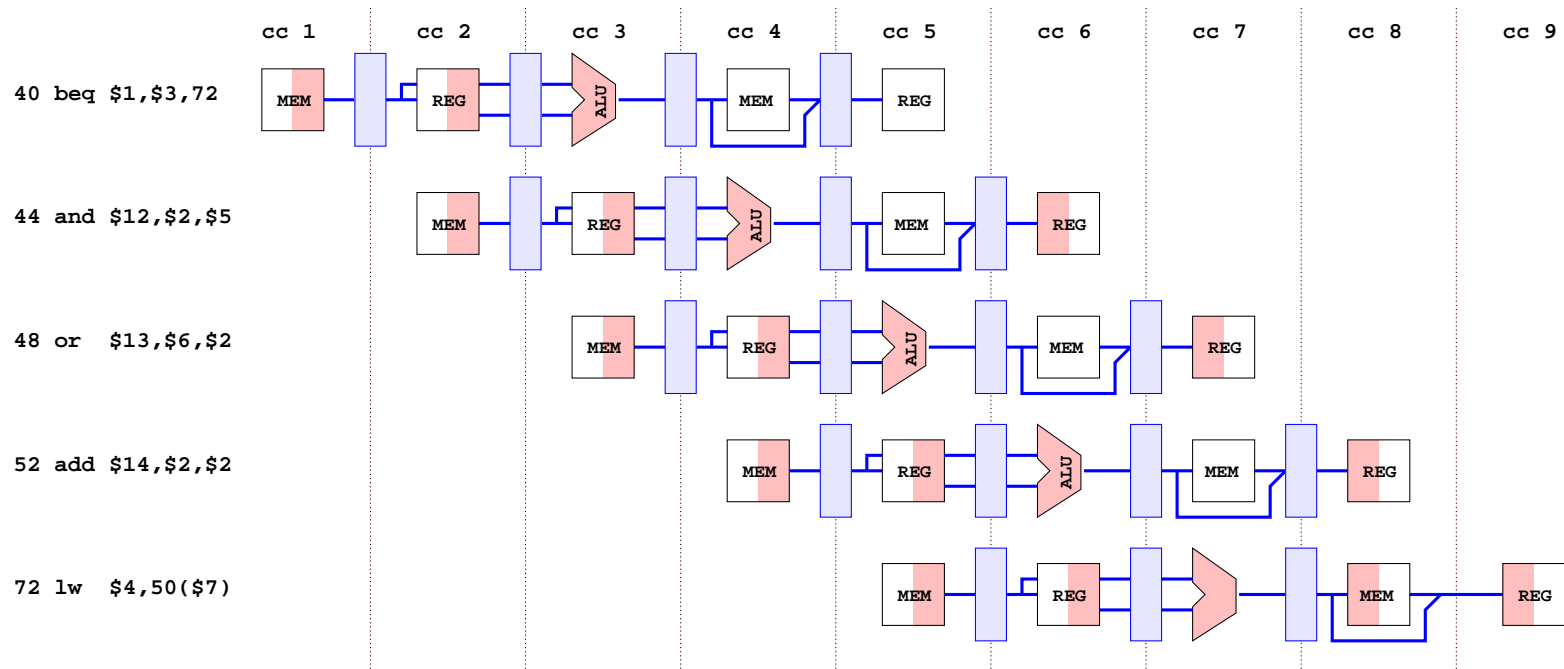
First solution: stall the pipeline until the branch is resolved.



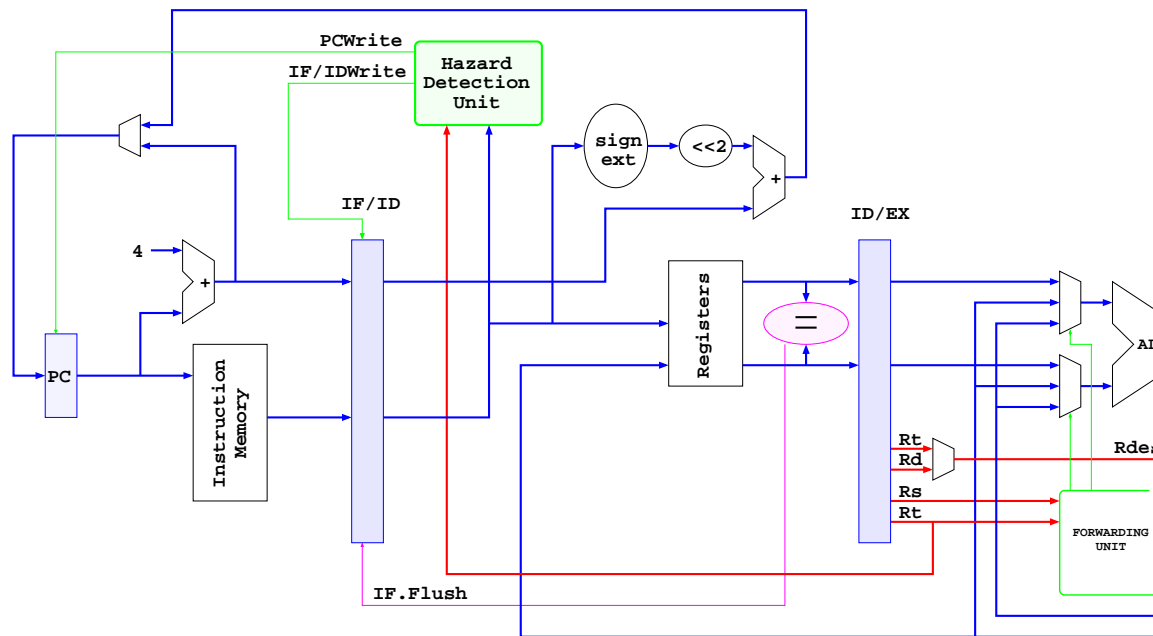
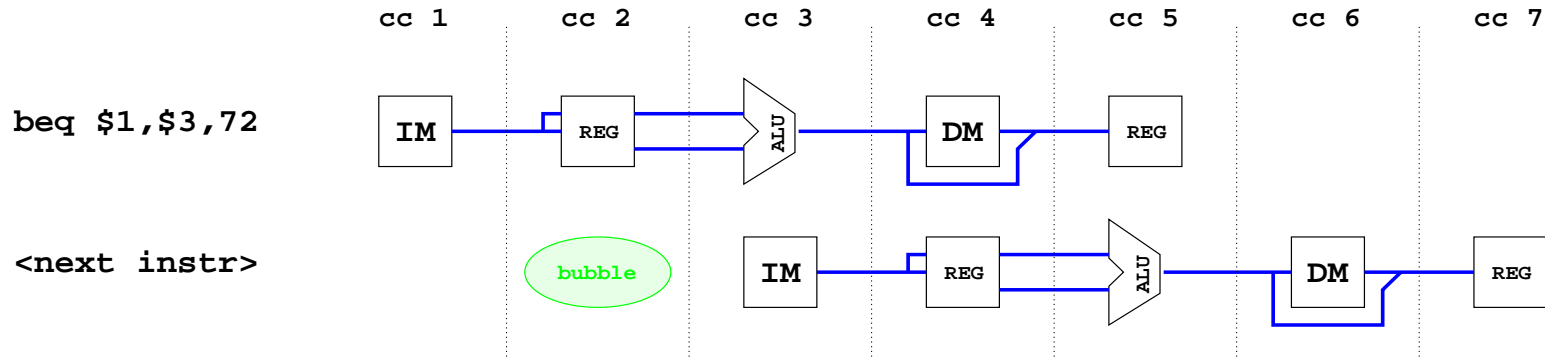
Second solution: try to guess — “assume branch not taken”

Always start executing the next instruction (do not stall).

If the branch is taken, flush the pipeline.



Third solution: try to detect the branch earlier

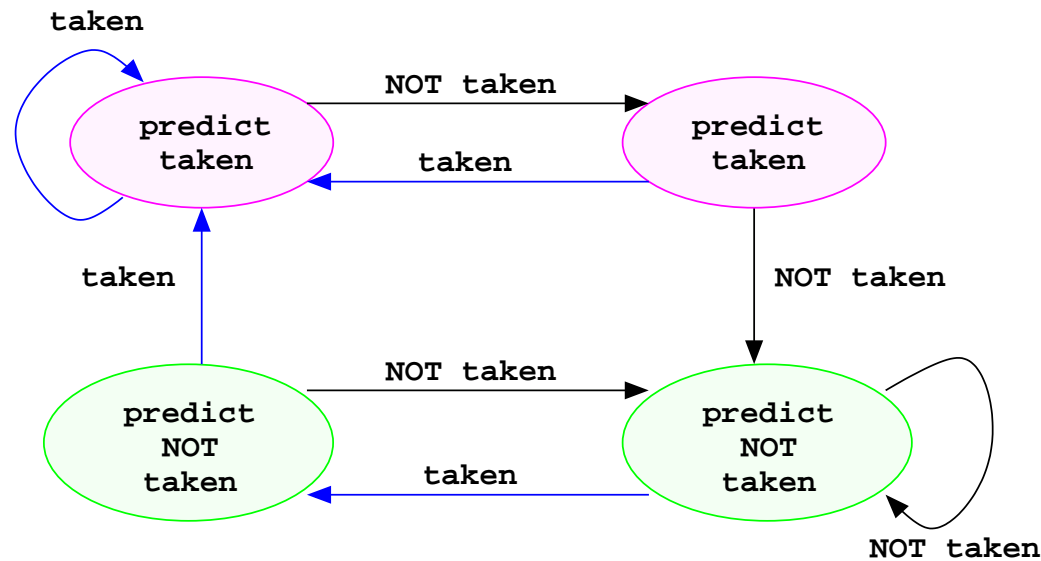


Fourth solution: **dynamic branch prediction**

A *branch prediction buffer* (or *branch history table*) keeps track of previous decisions.

We can keep just 1 bit or 2 bits:

- with 1 bit, the bit is set every time
- with 2 bits, the prediction must be wrong twice before it is changed



Delayed branches

The instruction immediately following the branch instruction is always executed.

The compiler takes care of filling the *delay slot* with a proper instruction.

Example:

```
        beq    $1, $2, label1
        add    $3, $4, $5
        or     $6, $7, $8
        sub    $9, $10, $11
label1: lw     $12, 0($13)
```



Recommended exercises

Ex 6.1, 6.2, 6.4, 6.11, 6.12, 6.13, 6.14, 6.15, 6.20

