Pipelines Hazards

Structural hazards:

Structural hazards are those that occur because of *resource* conflicts

- Most common type: When a functional unit is **not fully pipelined**
  The use of the functional unit requires more than one clock cycle
  If an instruction follows an instruction that is using it, and the second instruction also requires the resource, it must stall

- A second type involves resources that are **shared** between pipe stages
  Occurs when two different instructions want to use the resource in the same clock cycle
  In this case, the *lack of duplication* of the resource does not allow all combinations of instructions in the pipeline to execute

These stalls increase the CPI from the ideal pipelined value of 1

Structural Hazards

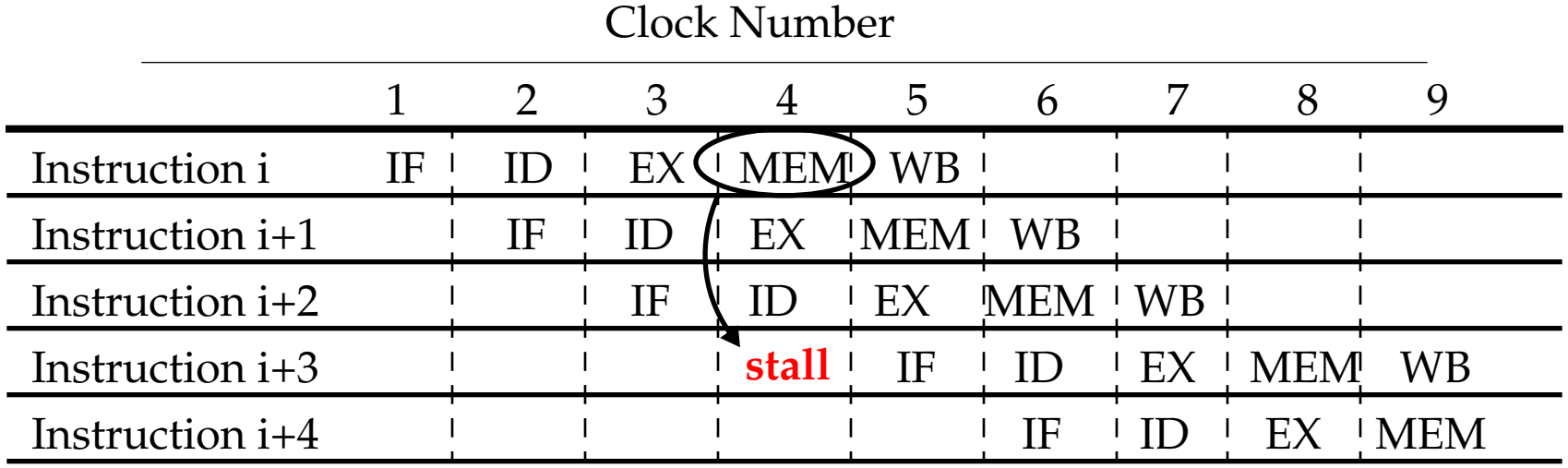
Example 1:

For cost-saving reasons, a CPU may be designed with a single interface to memory

This interface is always used during IF

It is also used during MEM for Load or Store operations

When a Load or Store gets to the MEM stage, the instruction in the IF stage must be stalled



Structural Hazards

Example 2: Consider branches with complex conditions:

Let's modify DLX pipeline to allow branches that:

- First perform a comparison (during the EX cycle)
- And then the address calculation if the branch was taken (during the MEM cycle) -- which requires the ALU in the EXE stage

In such a case, the MEM cycle of a branch would interfere with the EX cycle of the following instruction, causing a stall

In both cases, the problem could be solved with **additional** CPU hardware

In the first case, a second memory port

In the second case, an additional ALU

Therefore, structural hazards are caused solely by insufficient hardware

Structural Hazards

Machine with**OUT** structural hazards will always have a lower CPI

If this is the case, then why allow them?

- *To reduce cost*

  i.e. adding split caches, requires twice the memory bandwidth

  Also, a fully pipelined floating point multiplier costs lots of gates

  It is not worth the cost if the hazard does not occur very often

- *To reduce latency of the unit*

  Making functional units pipelined adds delay (pipeline overhead -> registers)

  An unpipelined version may require fewer clocks per operation

  Reducing latency has other performance benefits, as we will see

Data Hazards

Pipelining changes the relative timing of instructions by overlapping them in time

This introduces possible hazards by reordering accesses

- To the register file (data hazards)
- To the program counter (control hazards)

Consider the code:

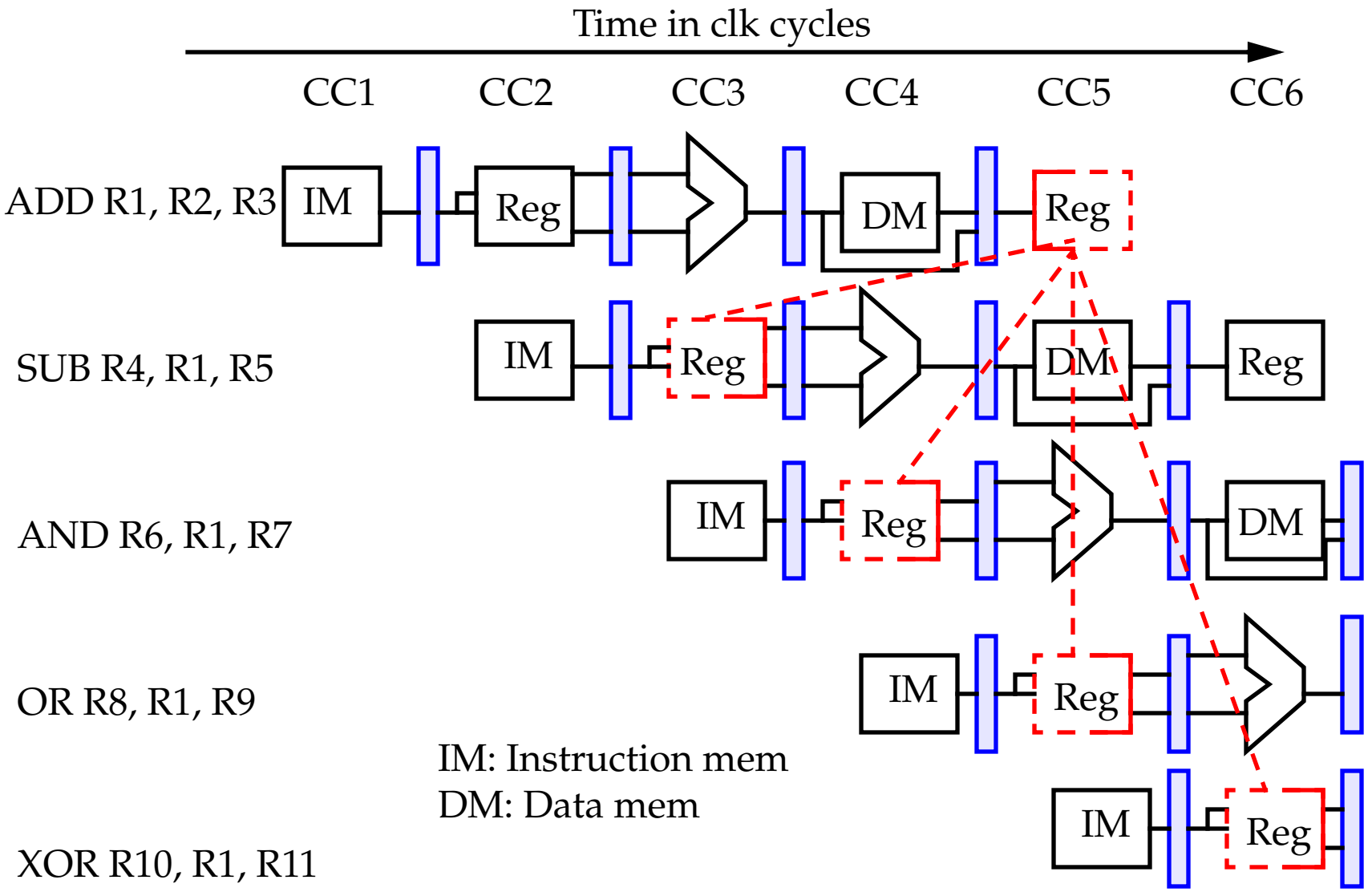
```
ADD R1, R2, R3
SUB R4, R5, R1
AND R6, R1, R7
OR R8, R1, R9
XOR R10, R1, R11
```

All of the instructions *after* ADD use the result of the ADD instruction

Since the standard DLX pipeline waits until WB to write the value back, the SUB, AND and OR instructions **read** the wrong value

Also, the error may *not be deterministic* if an interrupt occurs between the ADD and the AND, which would allow the ADD to write its result

Data Hazards



Data Hazards

Memory reference data hazards:

We used registers in our example

It is also possible for a pair of instructions to create a dependence by writing and reading the same memory location

In DLX, however, we always keep the memory references in order, preventing this type of hazard

Consider cache misses

These could cause memory references to get out of order if we allowed the processor to continue to work on later instructions

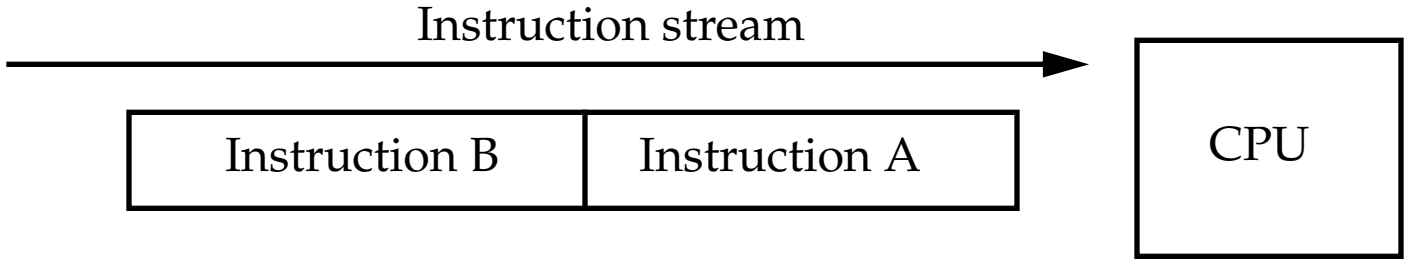
For DLX, we stall in entire pipeline on cache misses

There are architectures that allow Load and Stores to be executed out of order

Data Hazards

Types of data hazards:

Consider two instructions, A and B. - A occurs before B

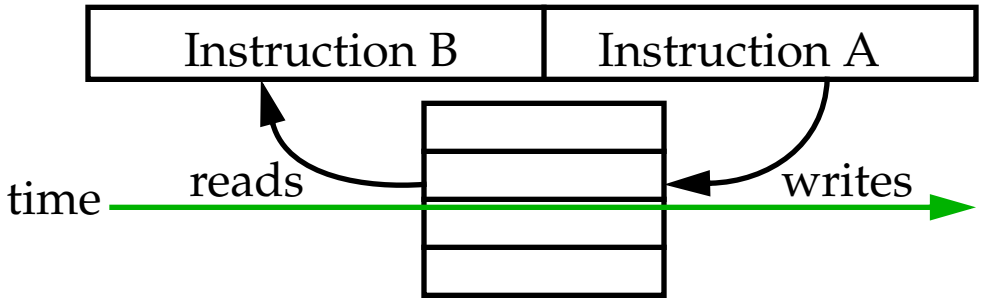


Hazards are named according to the ordering that MUST be preserved by the pipeline

- *RAW (read after write)*

  B tries to read a register *before* A has written it and gets the old value

  This is common, and forwarding helps to solve it

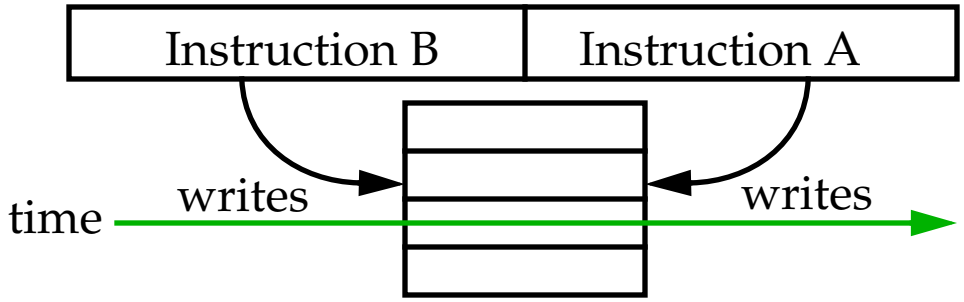


Data Hazards

Types of data hazards:

- *WAW (write after write)*

  B tries to write an operand *before* A has written it



After instruction B has executed, the value of the register should be B's result, but A's result is stored instead

This can only happen with pipelines that write values in more than one stage, or in variable-length pipelines (i.e. FP pipelines)

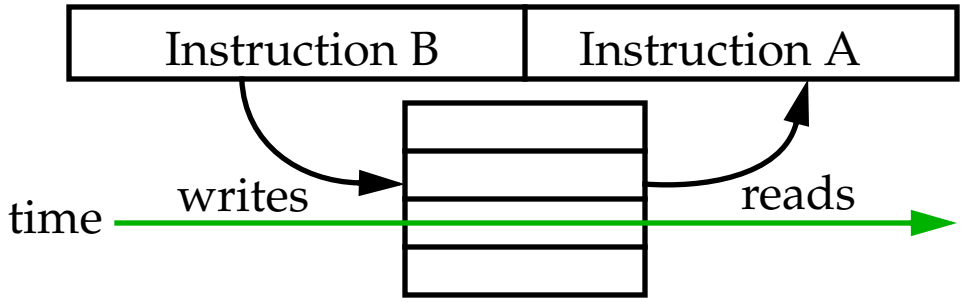
It does not happen in our version of the DLX pipeline, but a modified version might allow it

Data Hazards

Types of data hazards:

- *WAR (write after read)*

B tries to write a register *before* A has read it



In this case, A uses the new (incorrect) value

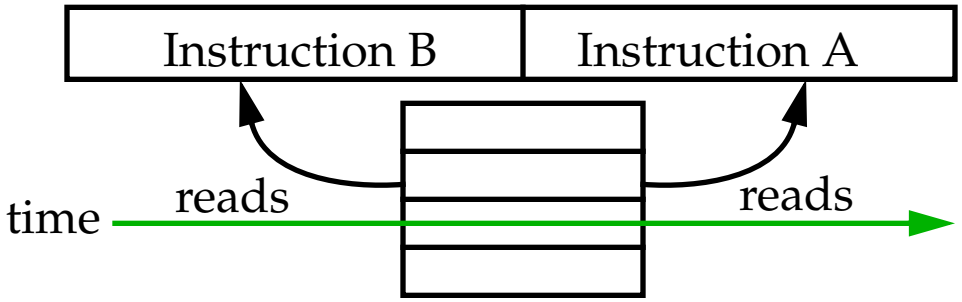
This type of hazard is rare because most pipelines read values early and write results late

However, it might happen for a CPU that had complex addressing modes. i.e. autoincrement

Data Hazards

Types of data hazards:

- *RAR (read after read)*



This is **NOT** a hazard since the register value does NOT change

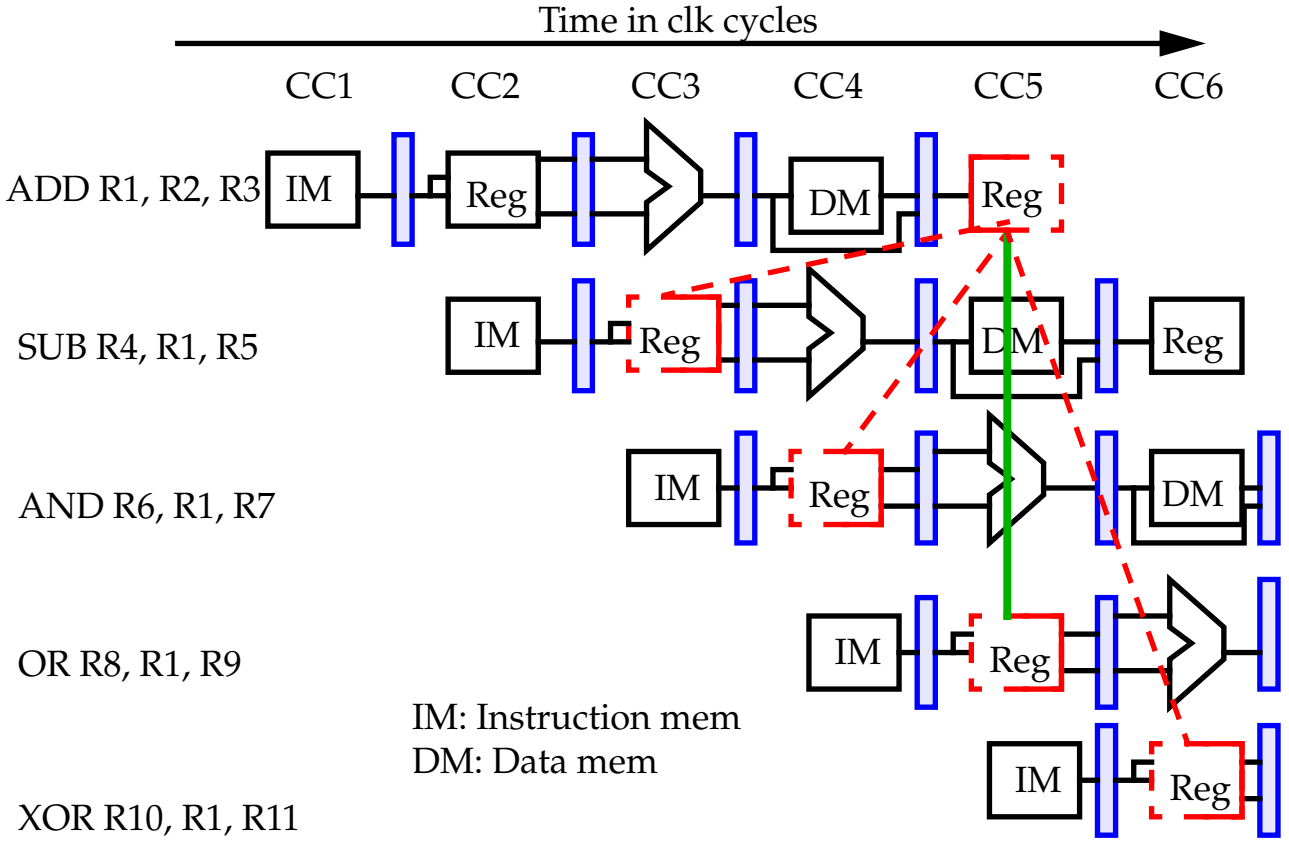
The order of the two reads is not important

Data Hazards

Fixing data hazards:

- **Simple solution**

The first thing that most pipelines do to *avoid* hazards is to write the register file in the first half of the cycle and read it in the second half



Fixes the hazard shown in green

Data Hazards

Fixing data hazards:

- *Forwarding* (also called *bypassing* and *short-circuiting*)

A key observation is that the necessary register value is often available but is not in the right place, i.e. the register file

This occurs because of the structure of our pipeline

The fix: Allow the CPU to move a value directly from one instruction to another without going through the register file

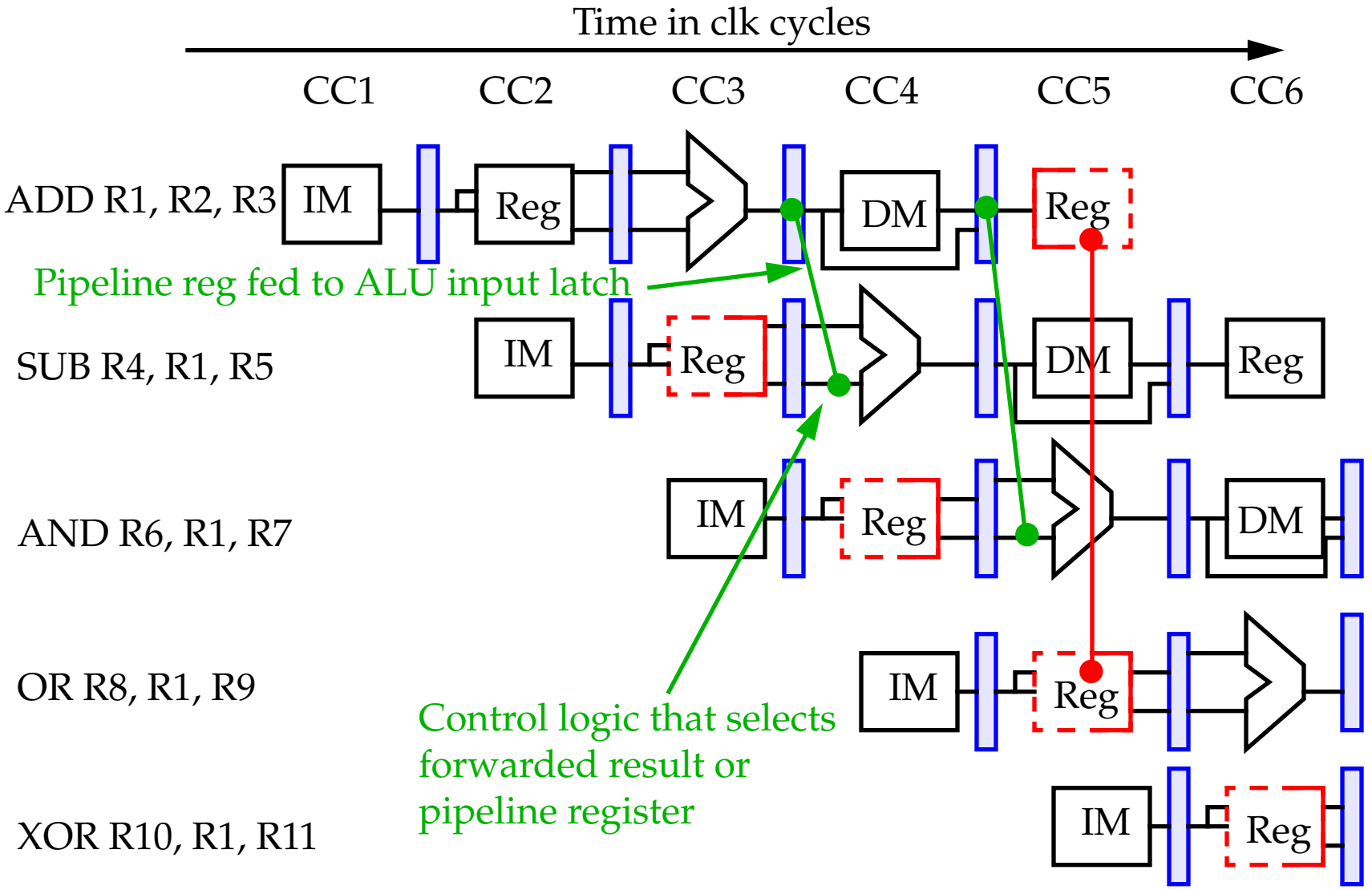
This is done by *feeding back* the data values from the pipeline registers to the inputs of functional units behind them in the datapath

The values are *forwarded* to future instructions

Note that they are actually moving *backward* in the datapath

Data Hazards

Forwarding:



Data Hazards

Fixing data hazards:

Note that forwarding is possible between:

- The output of one functional unit and the input of the same functional unit (previous example)
- The output of one functional and the input of another functional unit:

```
ADD R1, R2, R3
LW R4, 0(R1)
SW 12(R1), R4
```

In this case, the pipeline register values for R1 and R4 need forwarded to the input of the ALU and data memory inputs

R1 (in EX/MEM) to the input of the ALU for the LW instruction

R1 (in MEM/WB) to the input of the ALU for the SW instruction

R4 (in MEM/WB) (from memory) to the input of DM for the SW instruction (to memory)

Data Hazards

Problematic data hazards:

Note that forwarding *always* works in the DLX pipeline for Reg-Reg instructions (prevents stalls)

Because all Reg-Reg operations do the real work in the EX stage

This may not always be the case for other instructions,

i.e. *Load* instruction

Forwarding helps Loads, but it does NOT solve all the problems

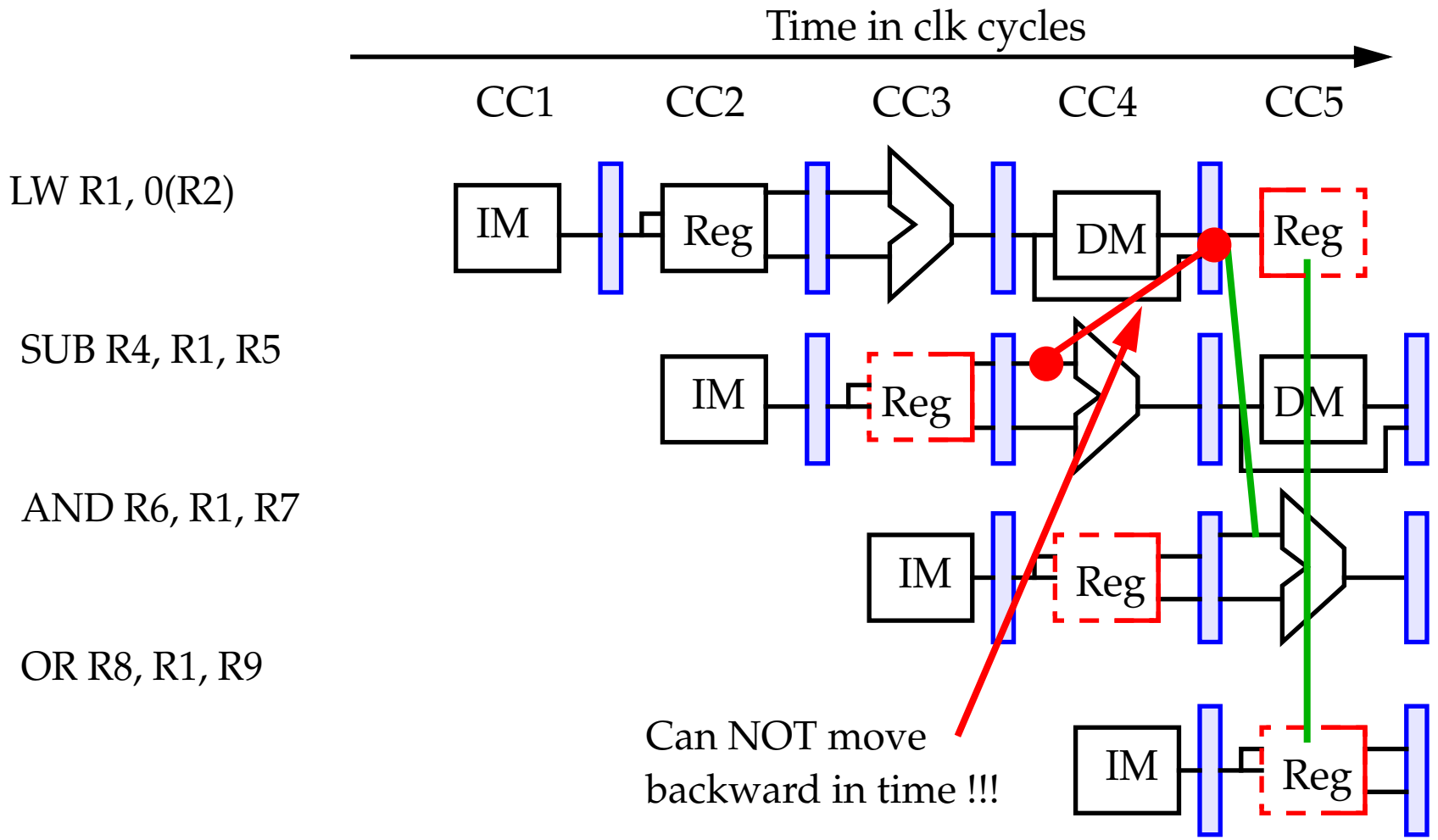
The Load is not completed until after MEM, which is after the EX stage that the following instruction completes

Note that the previous example worked because R4 was going directly to memory and was not needed by the ALU

Sometimes no amount of forwarding can help, as with a Load followed by an ALU operation

Data Hazards

For example, consider:



Data Hazards

Stalling is necessary in this case for proper execution.

This is done with a *pipeline interlock*, which stalls the pipeline until the hazard is cleared

This inserts a bubble into the pipeline just as the structural hazard did.

Just as with structural hazards, no instructions are started during the cycle in which the bubble is inserted

This increases the number of cycles required and thus the CPI

	Clock Number								
	1	2	3	4	5	6	7	8	9
LW R1, 0(R2)	IF	ID	EX	MEM	WB				
SUB R4, R1, R5		IF	ID	**stall**	EX	MEM	WB		
AND R6, R1, R7			IF	**stall**	ID	EX	MEM	WB	
OR R8, R1, R9				**stall**	IF	ID	EX	MEM	WB

Data Hazards

Example:

Assume 30% of the instructions are loads

Half the time, instruction following a load instruction depends on the result of the load

If hazard causes a single cycle delay, how much faster is the ideal pipeline?

Solution:

Ratio of the CPIs.

CPI for instructions following the load is 1.5, since they stall half of the time

Since loads are 30%, the effective CPI is:

0.7*1 + 0.3*1.5 = 1.15