POLITECNICO DI MILANO

Parallelism in wonderland:
are you ready to see how deep the rabbit hole goes?

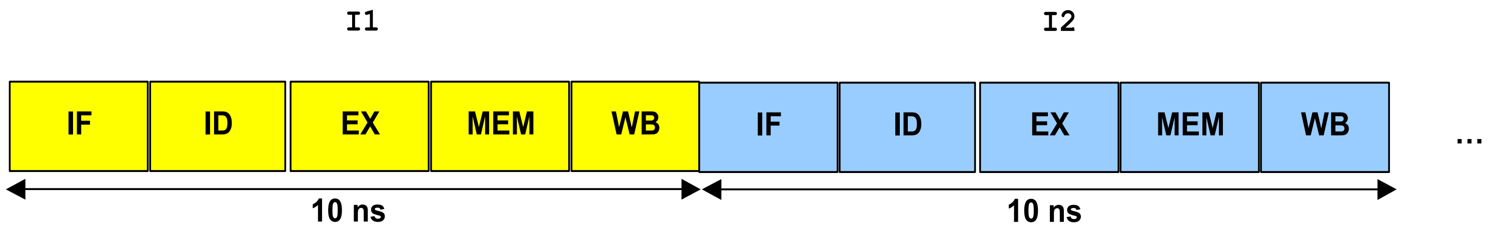
Pipelining: Hazards

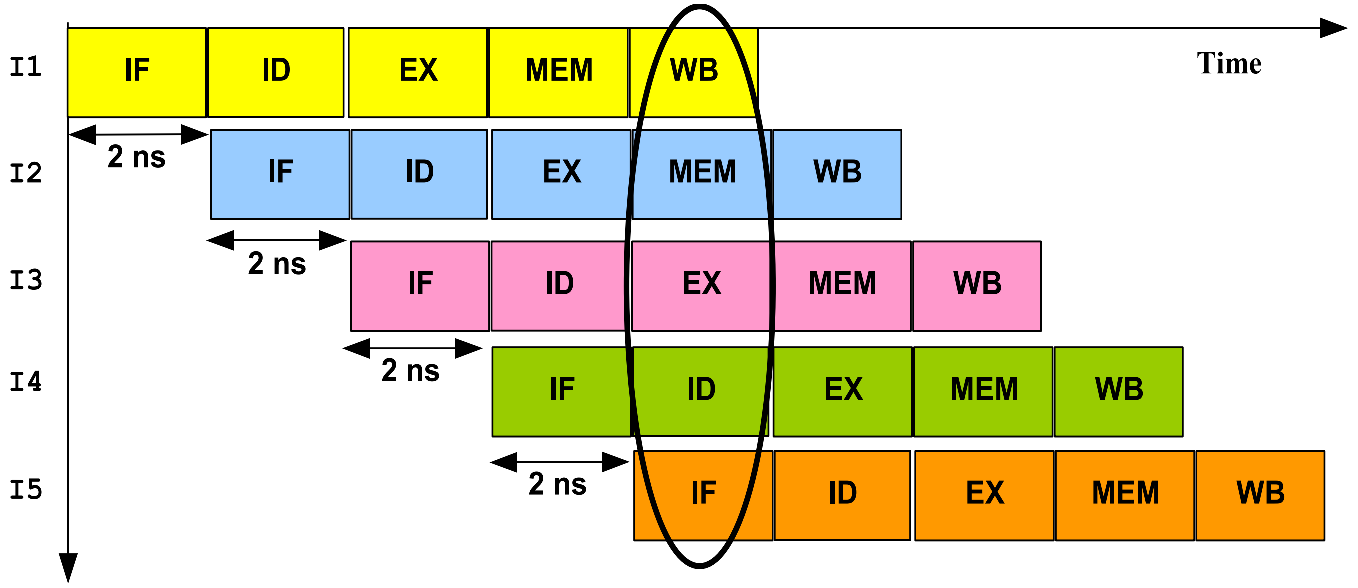
Ver. Jan 14, 2014

Marco D. Santambrogio: marco.santambrogio@polimi.it
Simone Campanoni: xan@eecs.harvard.edu

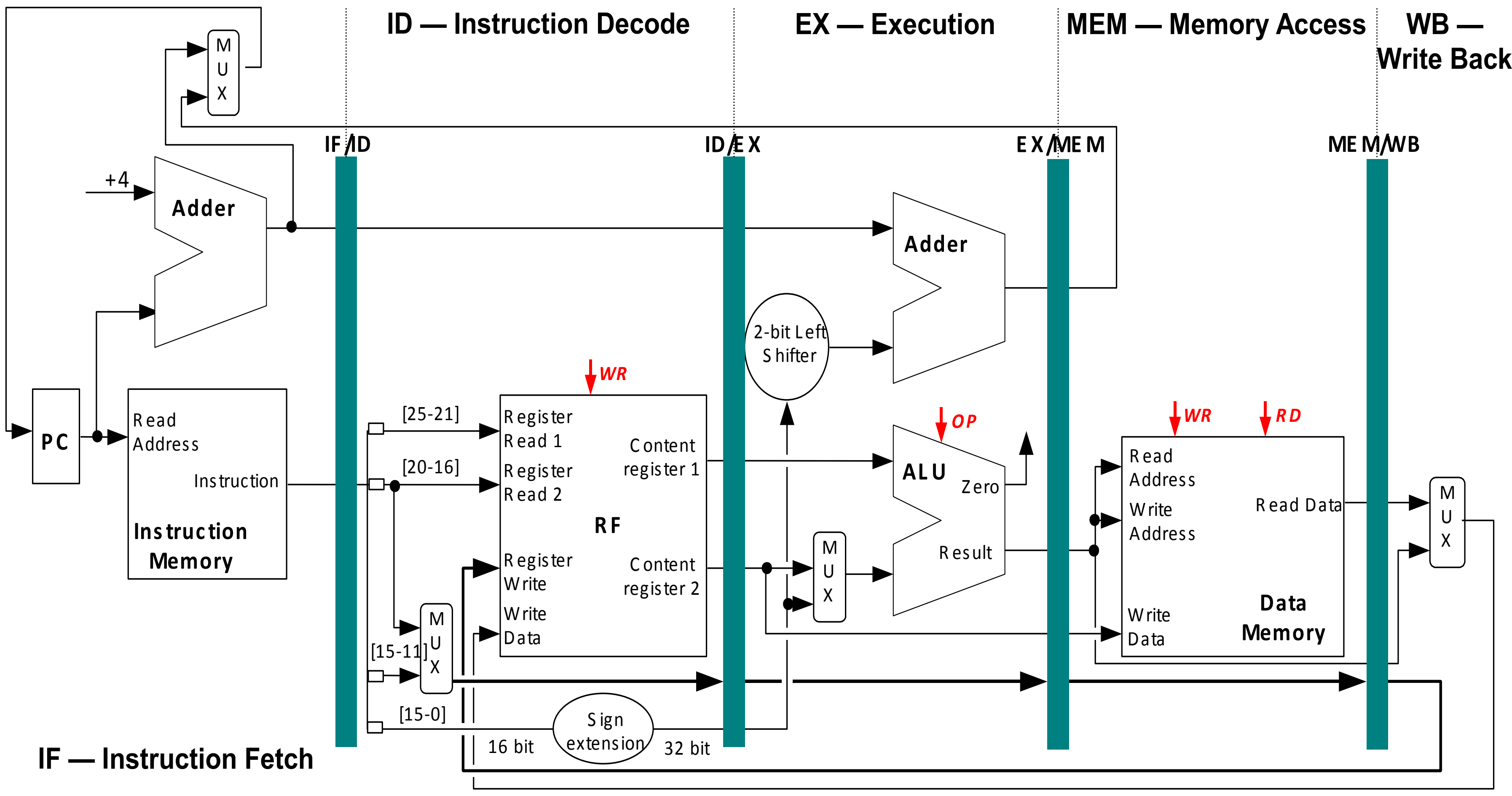
Back to the Intro to the Pipeline

Sequential vs. Pipelining Execution





Implementation of MIPS pipeline



Note: Optimized Pipeline

- Register File used in 2 stages: Read access durin" '" and write access during WB
  - What happens if read a· · register in · ·

From now on, this is the Pipeline we are going to use

ιυ ιne same

·· ·cycle?

·ιιcessary to insert one stall

Note: Optimized Pipeline

- Register File used in 2 stages: Read access during ID and write access during WB
  - What happens if read and write refer to the same register in the same clock cycle?
    - It is necessary to insert one stall

- **Optimized Pipeline:** the RF read occurs in the second half of clock cycle and the RF write in the first half of clock cycle
  - What happens if read and write refer to the same register in the same clock cycle?
    - It is not necessary to insert one stall

Today…

STEVEN SPIELBERG PRESENTS
BACK TO THE FUTURE
PG
A ROBERT ZEMECKIS FILM
OUTATIME

Outline

- The Problem of Pipeline Hazards
- Performance Issues in Pipelining

The Problem of Hazards

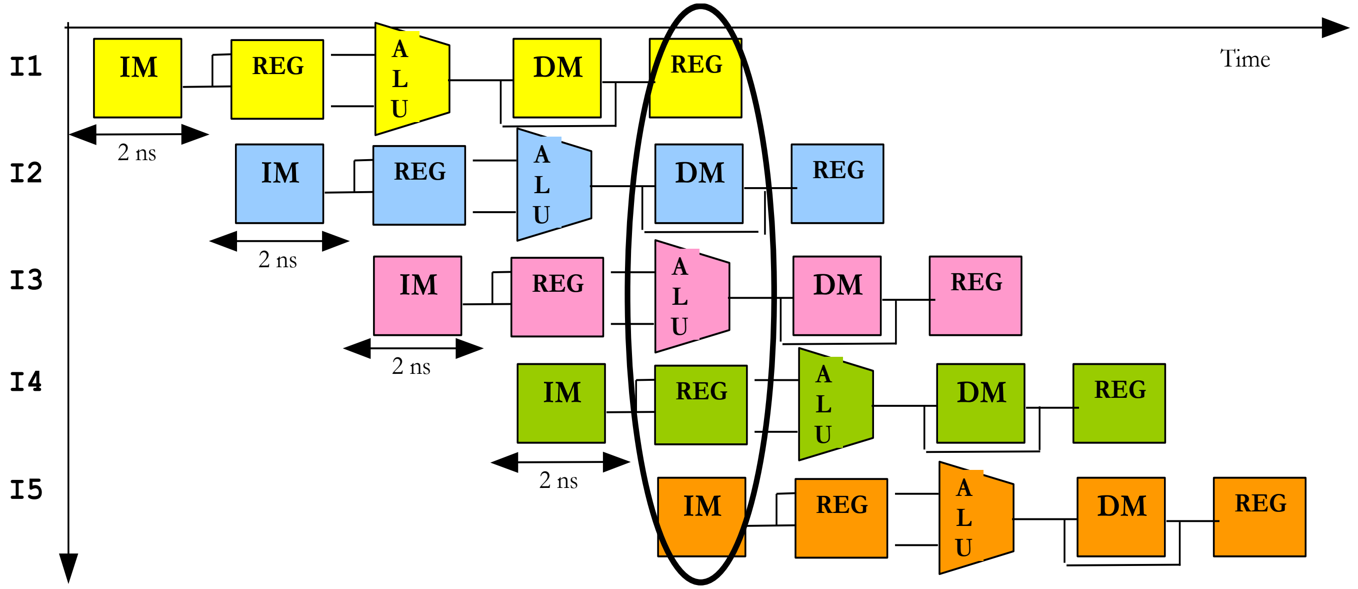
- A hazard is created whenever there is a dependence between instructions, and instructions are close enough that the overlap caused by pipelining would change the order of access to the operands involved in the dependence.
- Hazards prevent the next instruction in the pipeline from executing during its designated clock cycle.
- Hazards reduce the performance from the ideal speedup gained by pipelining.

Three Classes of Hazards

- **Structural Hazards**: Attempt to use the same resource from different instructions simultaneously
  - Example: Single memory for instructions and data
- **Data Hazards**: Attempt to use a result before it is ready
  - Example: Instruction depending on a result of a previous instruction still in the pipeline
- **Control Hazards**: Attempt to make a decision on the next instruction to execute before the condition is evaluated
  - Example: Conditional branch execution

Structural Hazards

- No structural hazards in MIPS architecture:
  - Instruction Memory separated from Data Memory
  - Register File used in the same clock cycle: Read access by an instruction and write access by another instruction



Speed Up Equation for Pipelining

$$CPI_{pipelined} = \text{Ideal CPI} + \text{Average Stall cycles per Inst}$$

$$\text{Speedup} = \frac{\text{Ideal CPI} \times \text{Pipeline depth}}{\text{Ideal CPI} + \text{Pipeline stall CPI}} \times \frac{\text{Cycle Time}_{unpipelined}}{\text{Cycle Time}_{pipelined}}$$

- For simple RISC pipeline, CPI = 1:

$$\text{Speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall CPI}} \times \frac{\text{Cycle Time}_{unpipelined}}{\text{Cycle Time}_{pipelined}}$$

Data Hazards

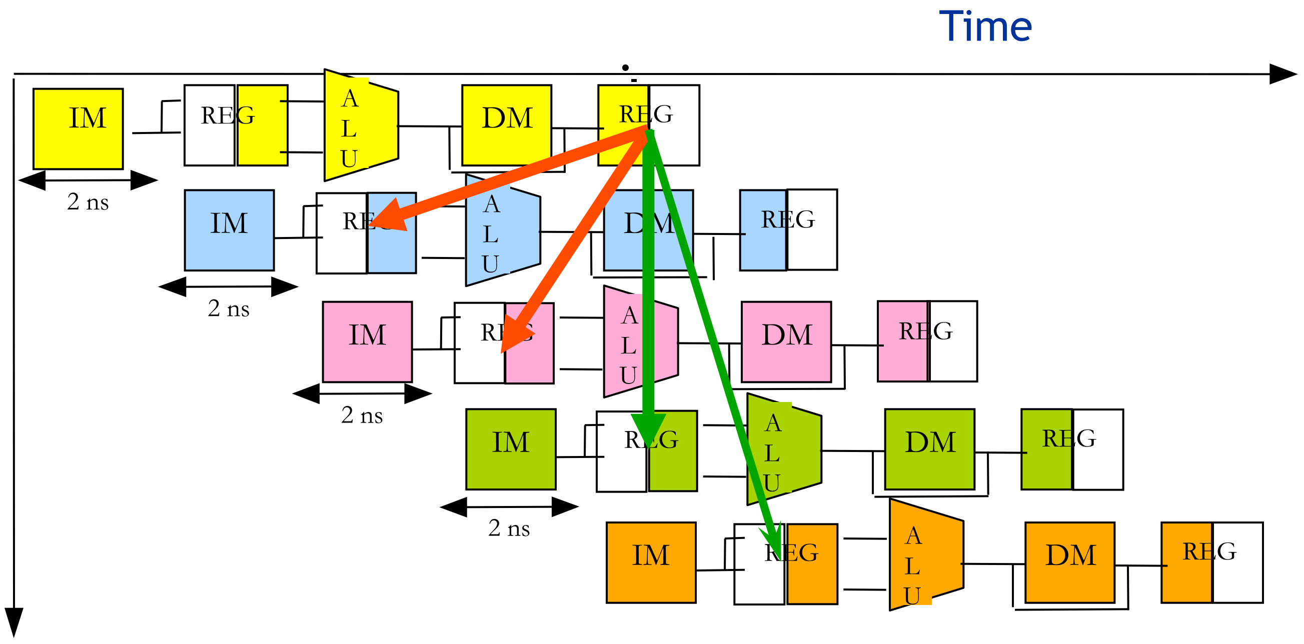
- If the instructions executed in the pipeline are **dependent**, data hazards can arise when instructions are too close

- Example:

```
sub  $2, $1, $3 # Reg. $2 written by sub
and $12, $2, $5 # 1° operand ($2) depends on sub
or $13, $6, $2  # 2° operand ($2) depend on sub
add $14, $2, $2 # 1° ($2) & 2° ($2) depend on sub
sw $15,100($2)  # Base reg. ($2) depends on sub
```

Data Hazards in the Optimized Pipeline: Example

```
sub  $2, $1, $3
and $12, $2, $5
or  $13, $6, $2
add $14, $2, $2
sw  $15,100($2)
```



Instruction order

<u>It is necessary to insert two stalls</u>

Type of Data Hazard

- Read After Write (RAW)
  Instr$_J$ tries to read operand before Instr$_I$ writes it

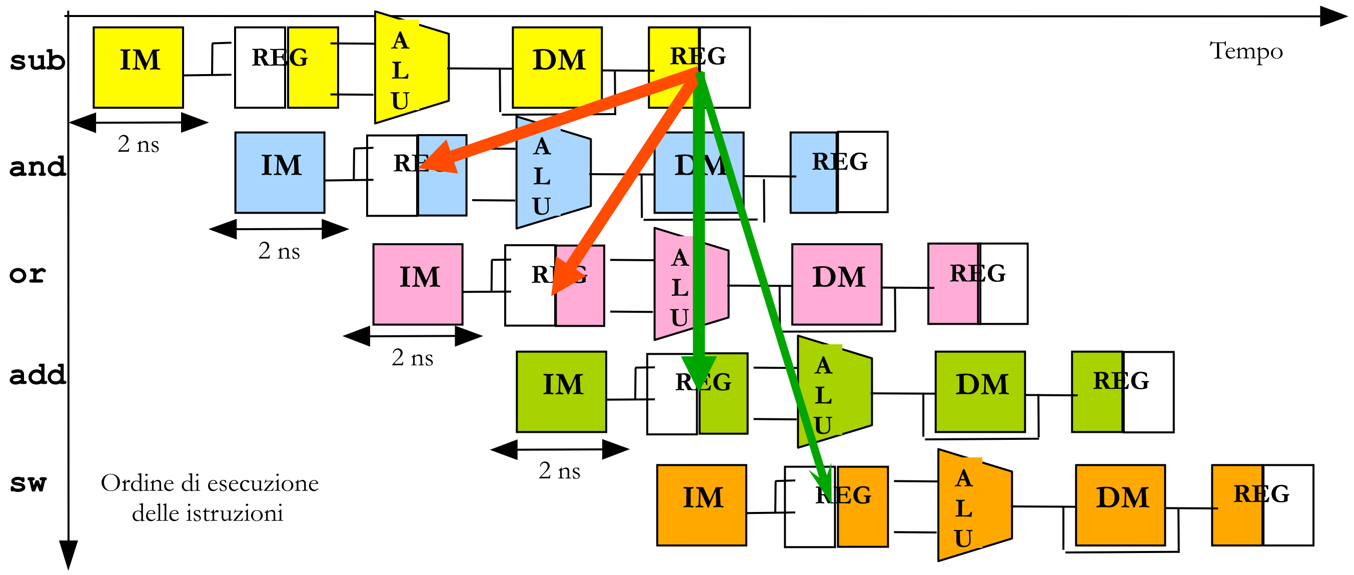
```
I: add r1,r2,r3
J: sub r4,r1,r3
```

- Caused by a "Dependence" (in compiler nomenclature). This hazard results from an actual need for communication.

Data Hazards: Possible Solutions

- **Compilation Techniques:**
  - Insertion of **`nop`** (*no operation)* instructions
  - Instructions Scheduling to avoid that correlating instructions are too close
    - The compiler tries to insert independent instructions among correlating instructions
    - When the compiler does not find independent instructions, it insert **`nops`**.
- **Hardware Techniques:**
  - Insertion of *"bubbles"* or *stalls* in the pipeline
  - Data Forwarding or Bypassing

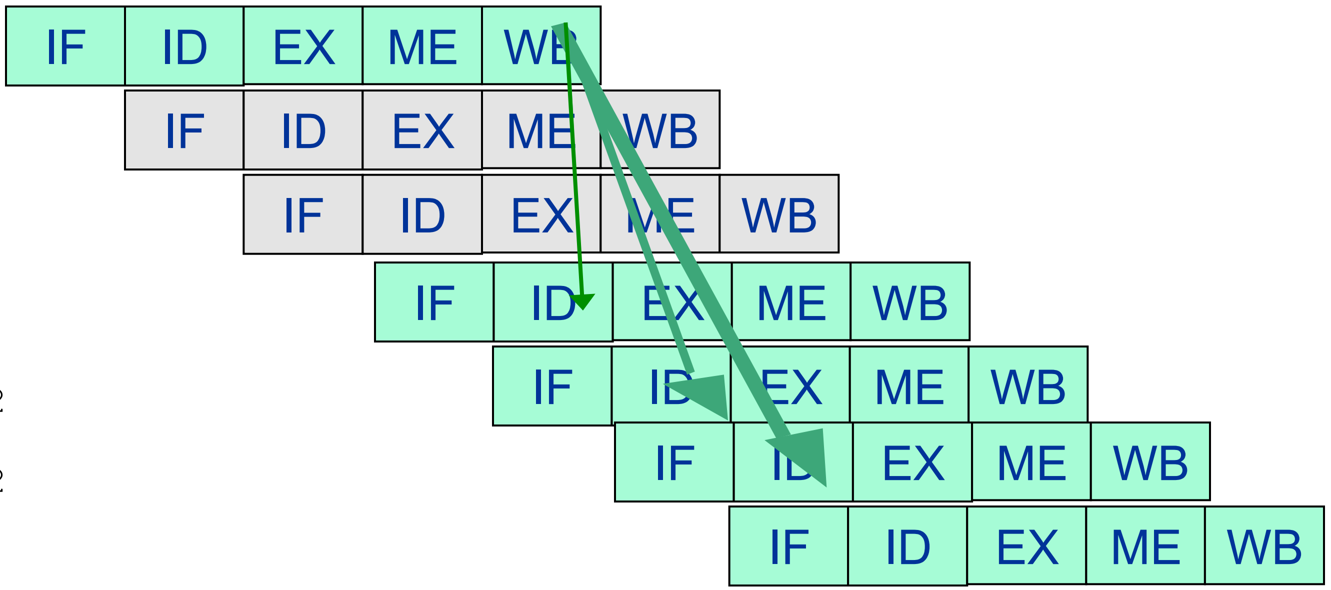
Just an example...



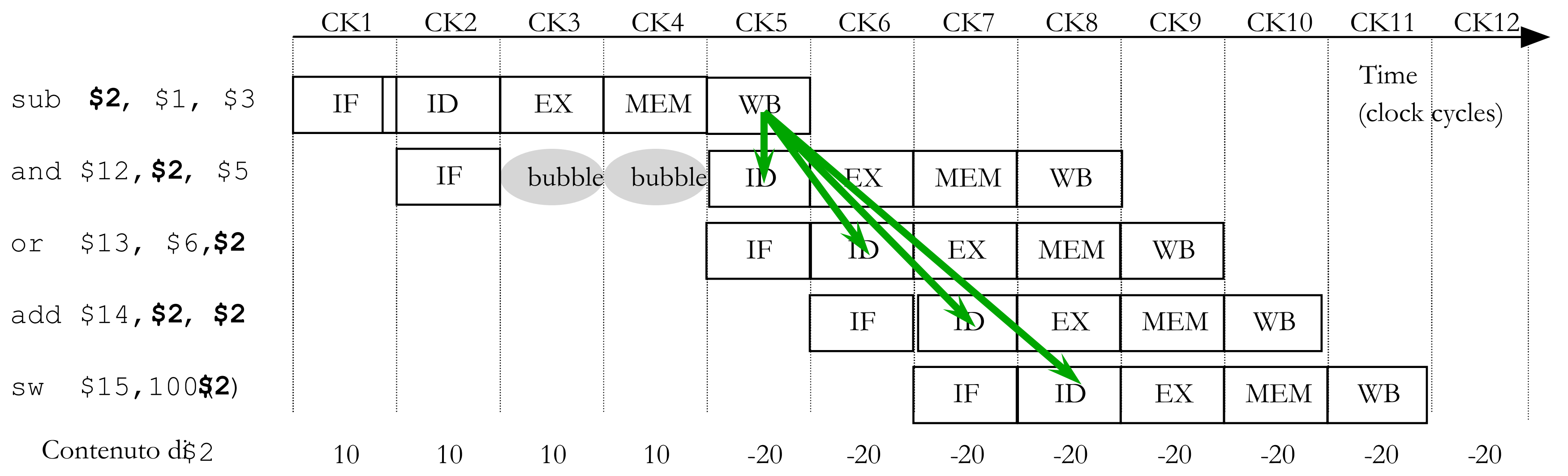
I need to insert 2 bubbles or 2 stalls

Insertion of nops

```
sub  $2, $1, $3
nop
nop
and $12, $2, $5
 or  $13, $6, $2
 add $14, $2, $2
sw  $15,100($2)
```



Insertion of bubbles and stalls



Scheduling: Example

```
sub  $2, $1, $3
and $12, $2, $5
or $13, $6, $2
add $14, $2, $2
sw $15,100($2)
add $4, $10, $11
and $7, $8, $9
lw $16, 100($18)
lw $17, 200($19)
```

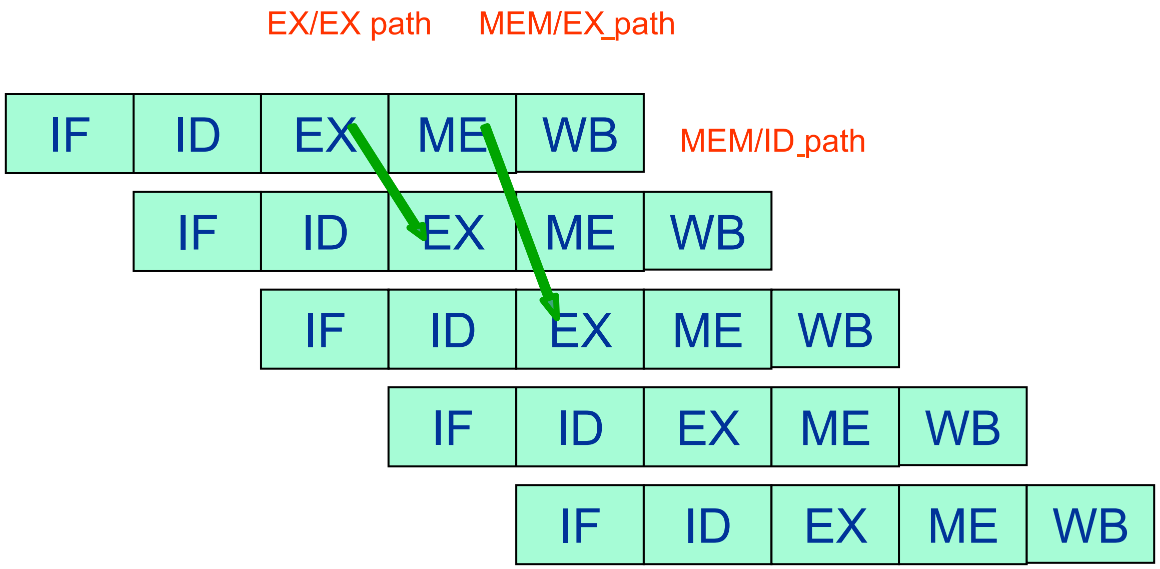
```
sub $2, $1, $3
add $4, $10, $11
      and $7, $8, $9
 lw $16, 100($18)
       lw $17, 200($19)
      and   $12, $2, $5
      or $13, $6, $2
      add   $14, $2, $2
      sw $15,100($2)
```

Forwarding

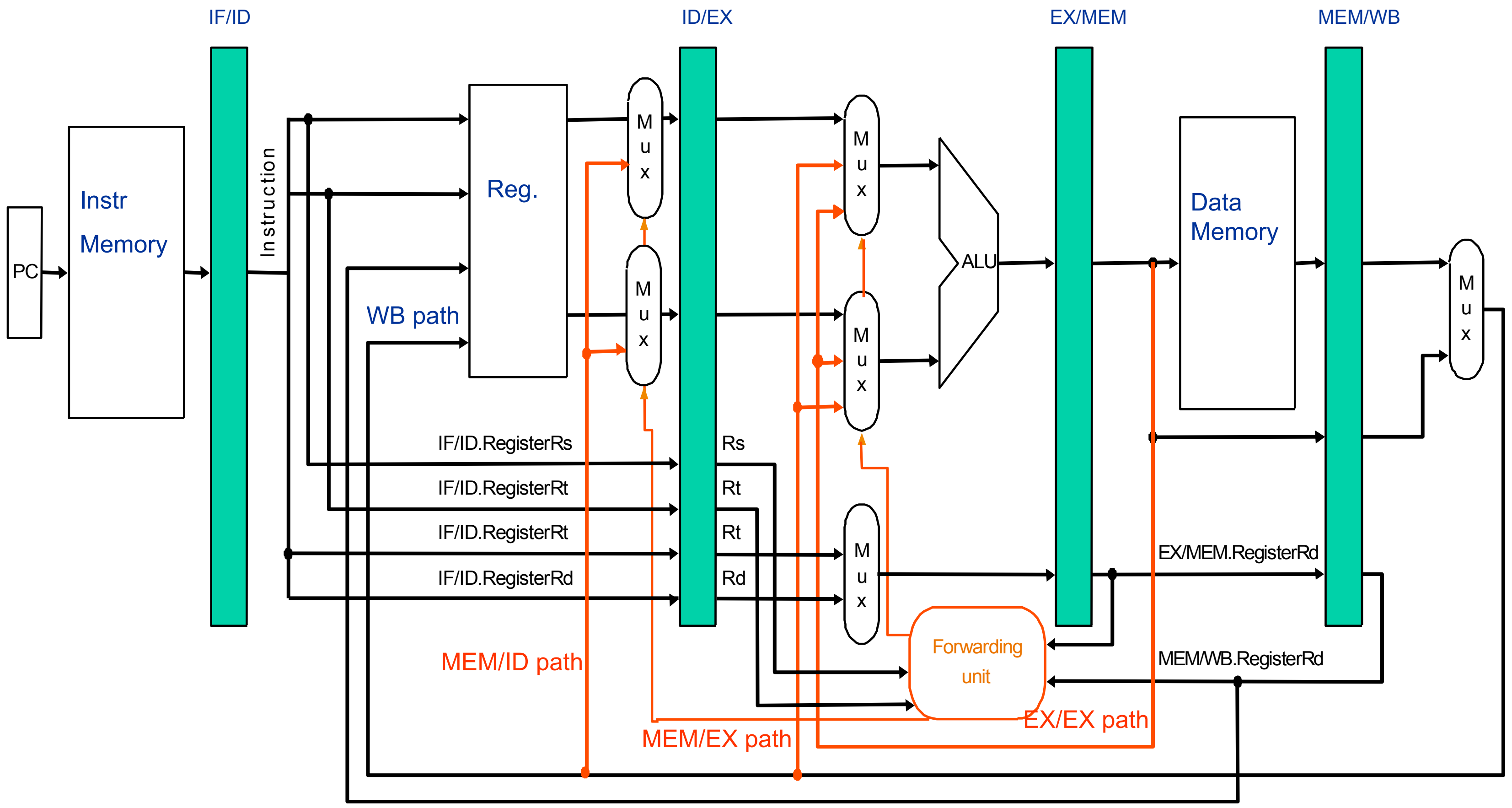
- Data forwarding uses temporary results stored in the pipeline registers instead of waiting for the write back of results in the RF.
- We need to add **multiplexers** at the inputs of ALU to fetch inputs from pipeline registers to avoid the insertion of stalls in the pipeline.

Forwarding: Example

```
sub  $2, $1, $3

and $12, $2, $5

or  $13, $6, $2

add $14, $2, $2

sw  $15,100($2)
```

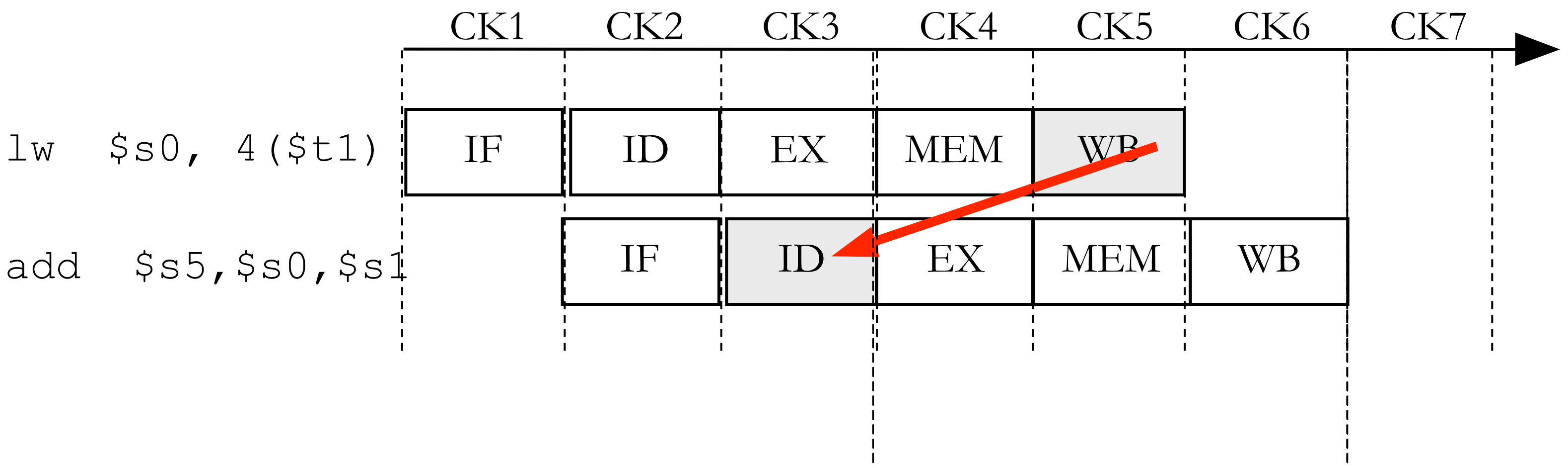


Implementation of MIPS with Forwarding Unit



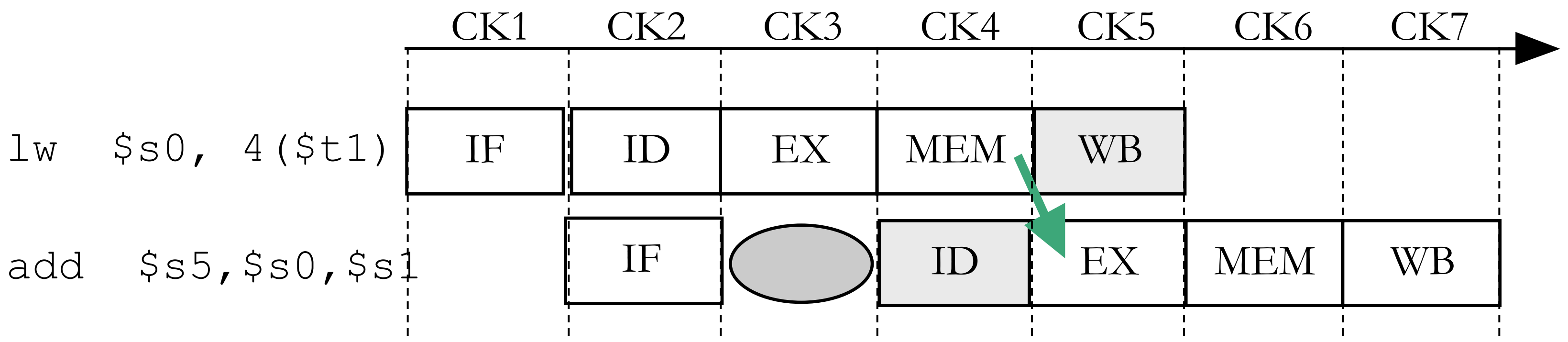
Data Hazards: Load/Use Hazard

```
L1: lw $s0, 4($t1)     # $s0 <- M [4 + $t1]
L2: add $s5, $s0, $s1 # 1° operand depends from L1
```



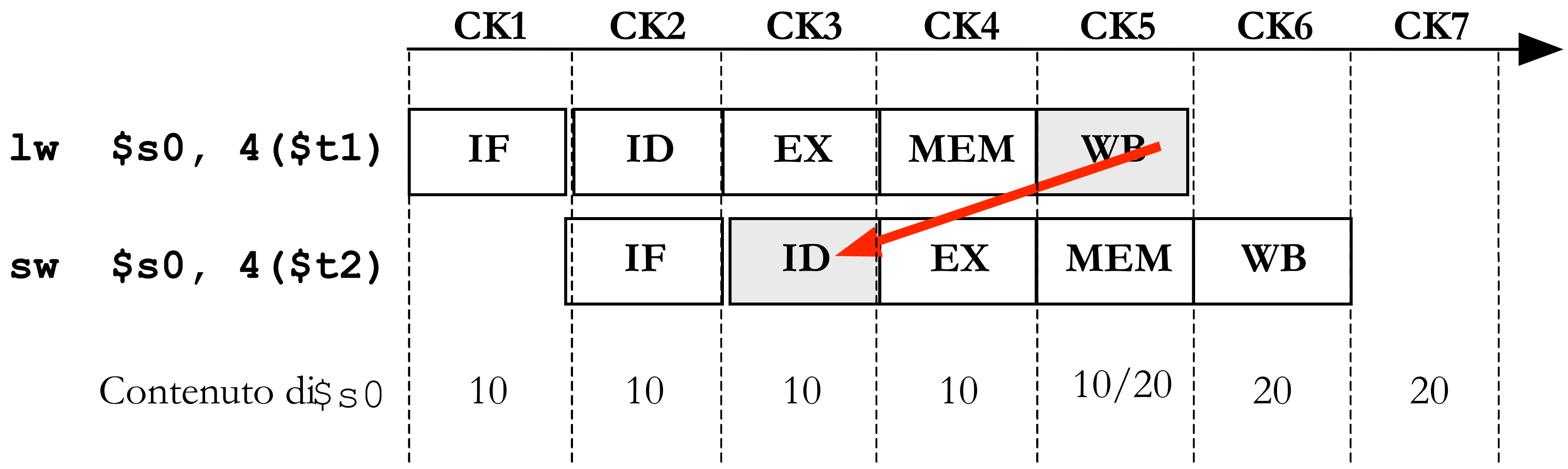
Data Hazards: Load/Use Hazard

- With forwarding using the MEM/EX path: 1 stall



Data Hazards: Load/Store

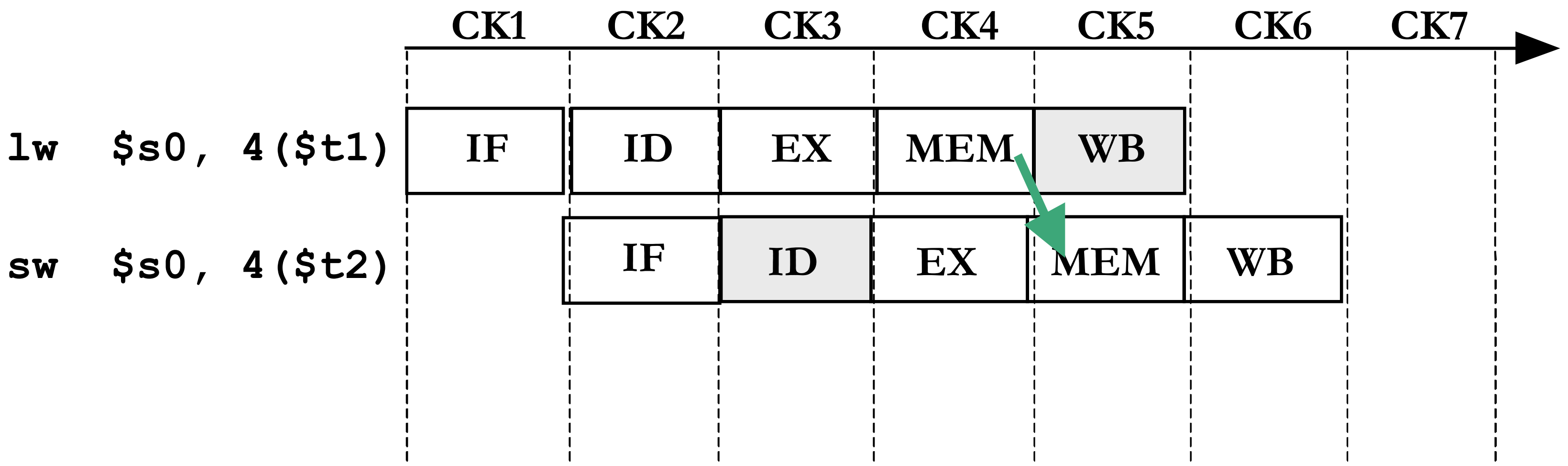
```
L1: lw $s0, 4($t1)     # $s0 <- M [4 + $t1]
L2: sw $s0, 4($t2)     # M[4 + $t2] <- $s0
```



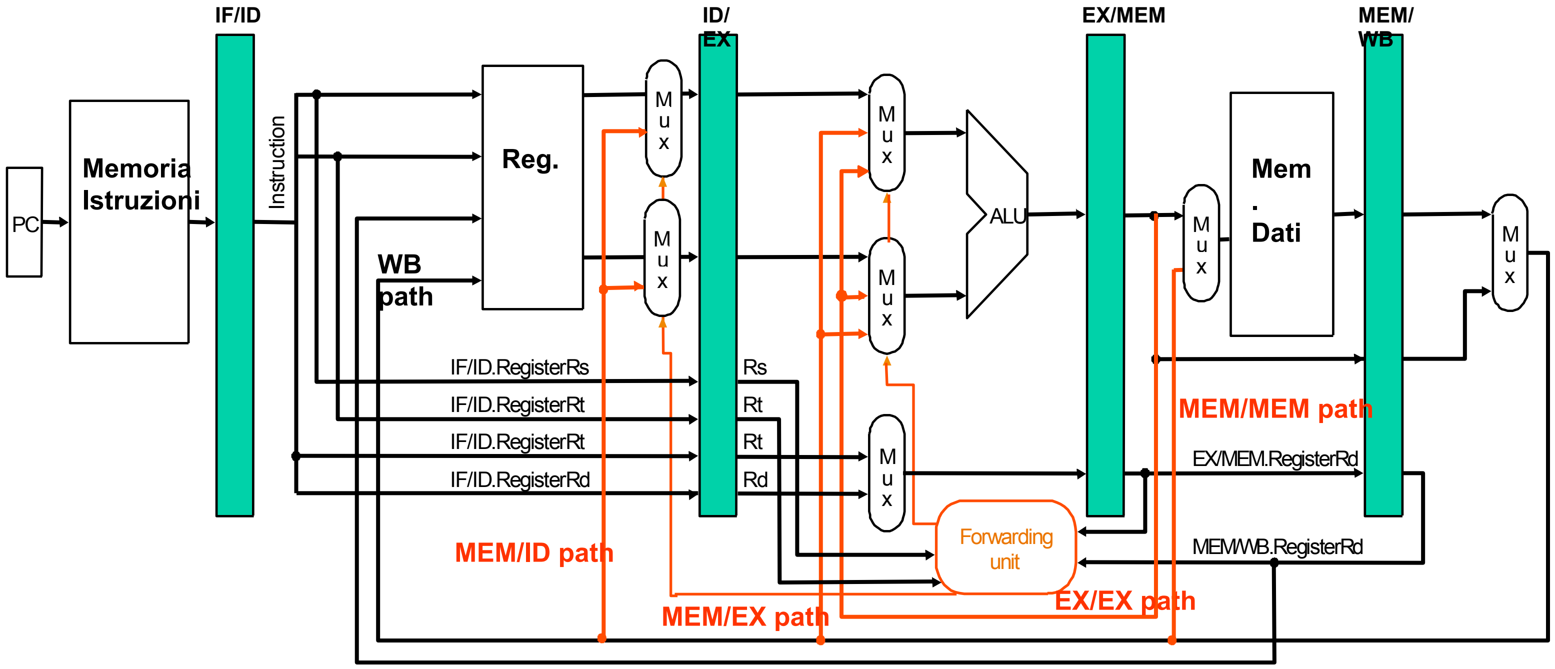
- Without forwarding : 3 stalls

Data Hazards: Load/Store

- Forwarding: Stall = 0
- We need a forwarding path to bring the *load* result from the memory (in MEM/WB) to the memory's input for the *store*.

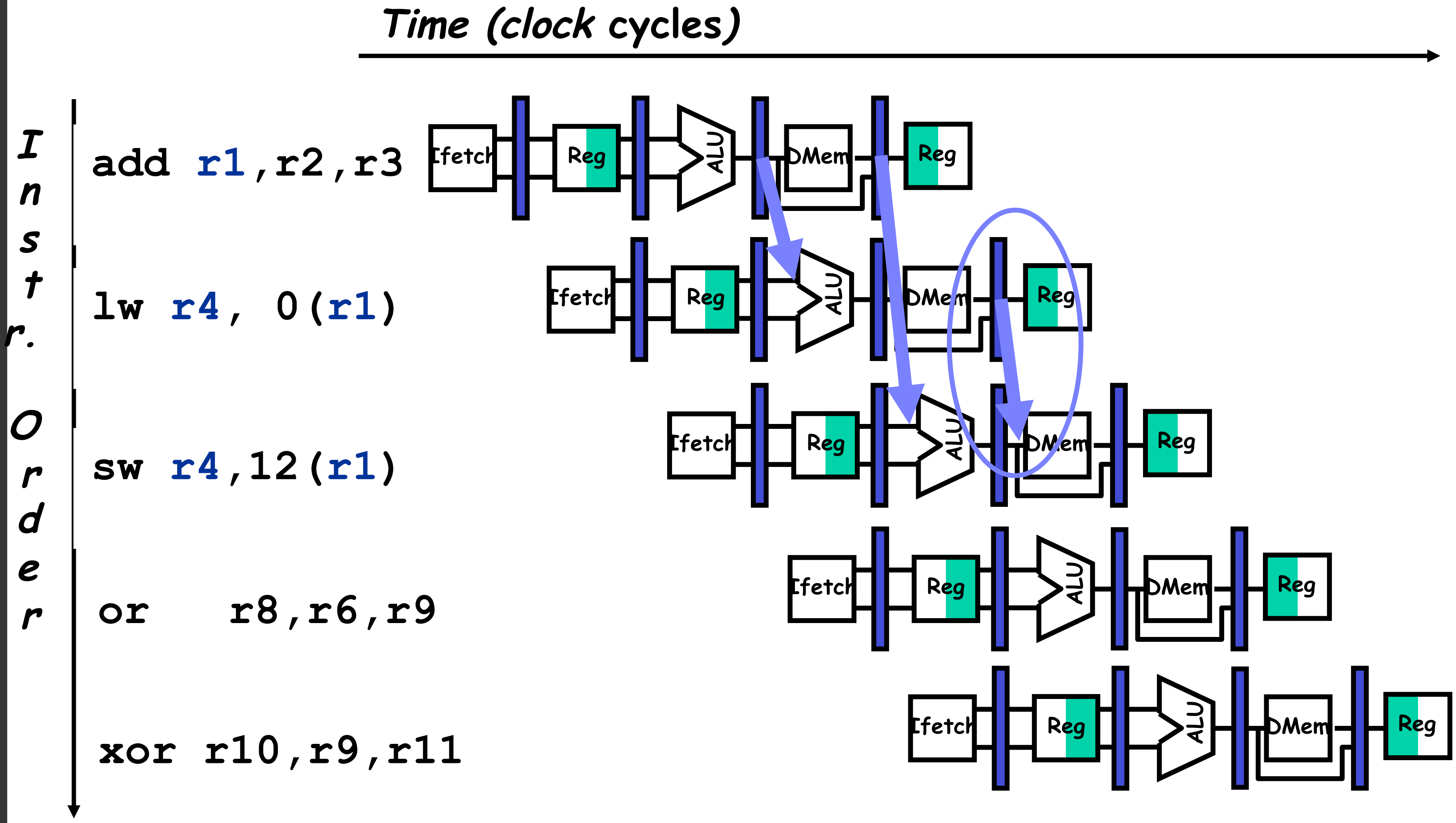


Implementation of MIPS with Forwarding Unit

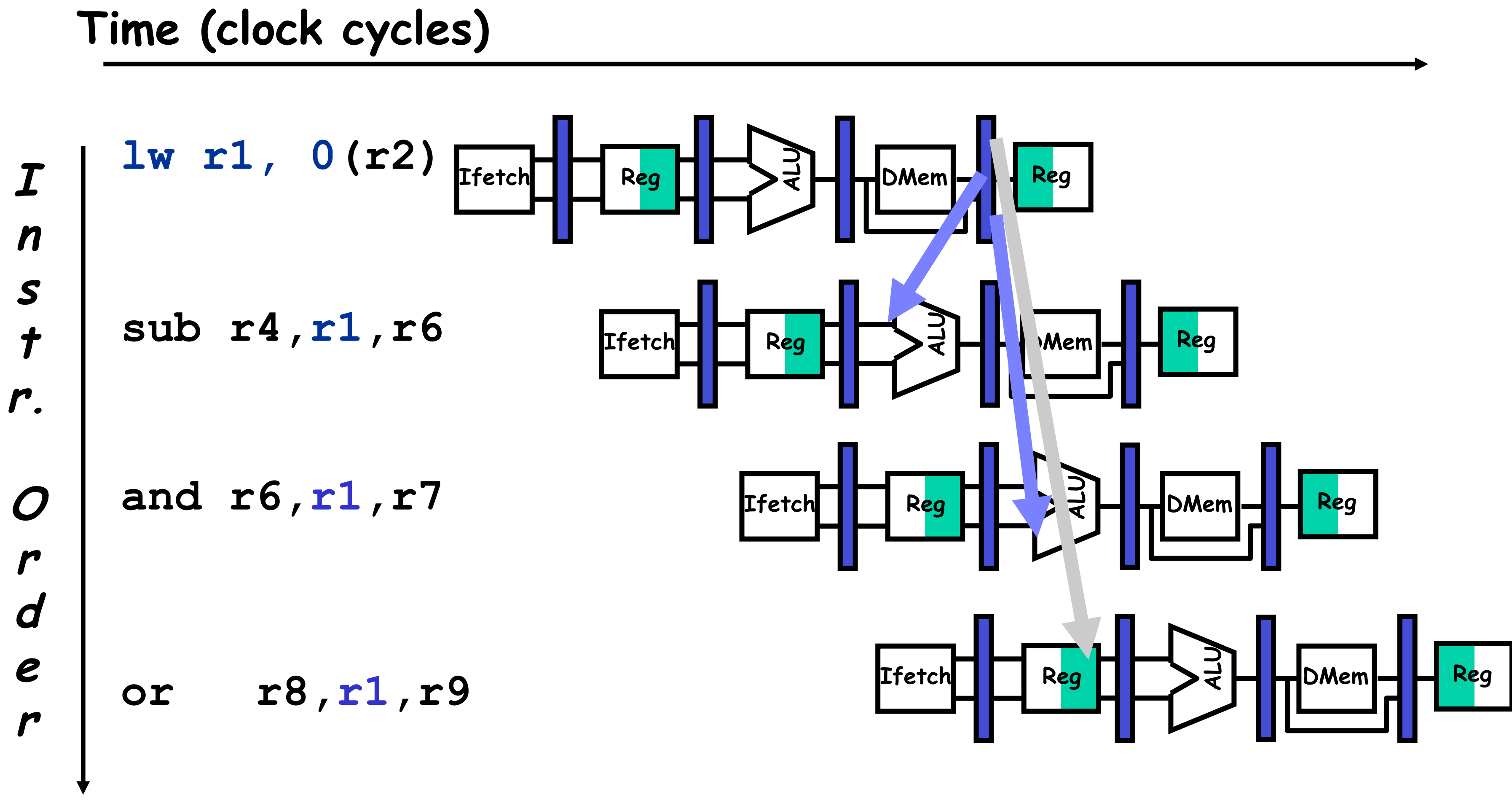


- EX/EX path
- MEM/EX path
- MEM/ID path
- MEM/MEM path

Forwarding to Avoid LW-SW Data Hazard



Data Hazard Even with Forwarding



Software Scheduling to Avoid Load Hazards

Try producing fast code for

```
a = b + c;
d = e - f;
```

assuming a, b, c, d ,e, and f in memory.

Slow code:

```
LW      Rb,b
LW      Rc,c
ADD     Ra,Rb,Rc
SW      a,Ra
LW      Re,e
LW      Rf,f
SUB     Rd,Re,Rf
SW      d,Rd
```

Fast code:

```
LW      Rb,b
LW      Rc,c
LW      Re,e
ADD     Ra,Rb,Rc
LW      Rf,f
SW      a,Ra
SUB     Rd,Re,Rf
SW      d,Rd
```

Compiler optimizes for performance.

Hardware checks for safety.

Data Hazards

- Data hazards analyzed up to now are:
  - **RAW (READ AFTER WRITE) hazards:** instruction *n+1* tries to read a source register before the previous instruction *n* has written it in the RF.
  - Example:

```
add $r1, $r2, $r3
sub $r4, $r1, $r5
```

- By using forwarding, it is always possible to solve this conflict without introducing stalls, except for the load/use hazards where it is necessary to add one stall

Data Hazards

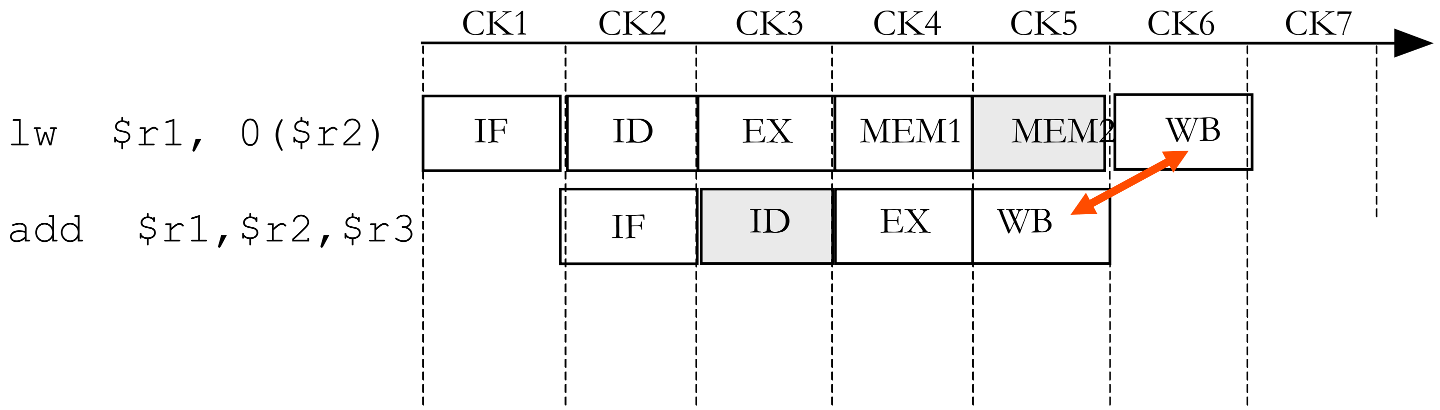
- Other types of data hazards in the pipeline:
  - **WAW (WRITE AFTER WRITE)**
  - **WAR (WRITE AFTER READ)**

Data Hazards: WAW (WRITE AFTER WRITE)

- Instruction *n+1* tries to write a destination operand before it has been written by the previous instruction *n* ⇒ write operations executed in the wrong order
- This type of hazards could not occur in the MIPS pipeline because all the register write operations occur in the WB stage

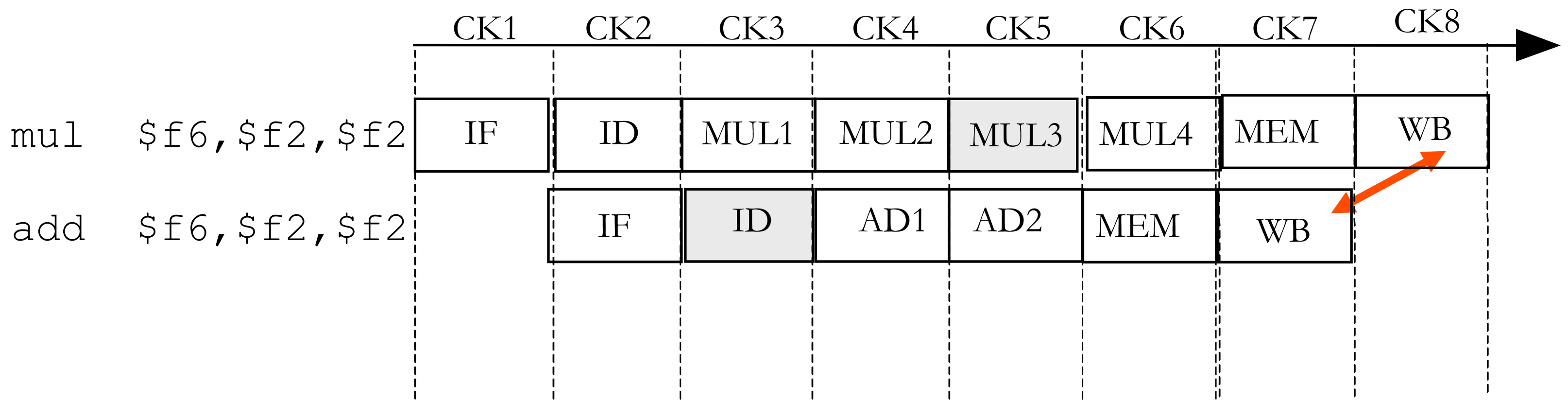
Data Hazards: WAW (WRITE AFTER WRITE)

- Example: If we assume the register write in the ALU instructions occurs in the fourth stage and that load instructions require two stages (MEM1 and MEM2) to access the data memory, we can have:



Data Hazards: WAW (WRITE AFTER WRITE)

- Example: If we assume the floating point ALU operations require a multi-cycle execution, we can have:



WAW Data Hazards

- Write After Write (WAW)
  Instr_J writes operand *before* Instr_I writes it.

```
I: sub r1,r4,r3
J: add r1,r2,r3
K: mul r6,r1,r7
```

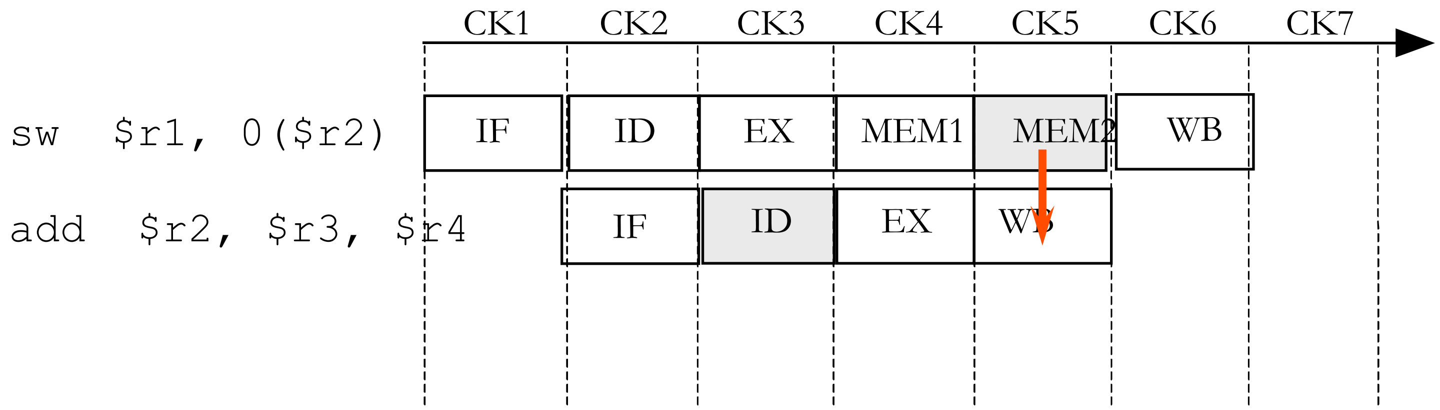
- Called an "output dependence" by compiler writers
  This also results from the reuse of name "r1".
- Can't happen in MIPS 5 stage pipeline because:
  - All instructions take 5 stages, and
  - Writes are always in stage 5
- Will see WAR and WAW in more complicated pipes

Data Hazards: WAR (WRITE AFTER READ)

- Instruction *n+1* tries to write a destination operand before it has been read from the previous instruction *n* $\Rightarrow$ instruction *n* reads the wrong value.
- This type of hazards could not occur in the MIPS pipeline because the operand read operations occur in the ID stage and the write operations in the WB stage.
- As before, if we assume the register write in the ALU instructions occurs in the fourth stage and that we need two stages to access the data memory, some instructions could read operands too late in the pipeline.

Data Hazards: WAR (WRITE AFTER READ)

- Example: Instruction **`sw`** reads **`$r2`** in the second half of MEM2 stage and instruction **`add`** writes **`$r2`** in the first half of WB stage ⇒ **`sw`** reads the new value of **`$r2`**.



WAR Data Hazards

- Write After Read (WAR)
  Instr_J writes operand *before* Instr_I reads it

```
I: sub r4,r1,r3
J: add r1,r2,r3
K: mul r6,r1,r7
```

- Called an “anti-dependence” by compiler writers.
  This results from reuse of the name “r1”.

- Can’t happen in MIPS 5 stage pipeline because:
  - All instructions take 5 stages, and
  - Reads are always in stage 2, and
  - Writes are always in stage 5

Performance Issues in Pipelining

- Pipelining increases the CPU instruction **throughput** (number of instructions completed per unit of time), but it does not reduce the execution time (latency) of a single instruction.
- Pipelining usually slightly increases the latency of each instruction due to imbalance among the pipeline stages and overhead in the control of the pipeline.
  - Imbalance among pipeline stages reduces performance since the clock can run no faster than the time needed for the slowest pipe stage.
  - Pipeline overhead arises from pipeline register delay and clock skew.

Performance Issues in Pipelining

- The average instruction execution time for the unpipelined processor is:

Ave. Exec. Time Unpipelined = Ave. CPI Unp. x Clock Cycle Unp.

$$\text{Pipeline Speedup} = \frac{\text{Ave. Exec. Time Unpipelined}}{\text{Ave. Exec. Time Pipelined}} =$$

$$= \frac{\text{Ave. CPI Unp.}}{\text{Ave. CPI Pipe}} \times \frac{\text{Clock Cycle Unp.}}{\text{Clock Cycle Pipe}} =$$

Performance Issues in Pipelining

- The ideal CPI on a pipelined processor is almost always 1, but stalls cause the pipeline performance to degrade form the ideal performance, so we have:

Ave. CPI Pipe = Ideal CPI + Pipe Stall Cycles per Instruction
= 1 + Pipe Stall Cycles per Instruction

- Pipe Stall Cycles per Instruction are due to Structural Hazards + Data Hazards + Control Hazards

Performance Issues in Pipelining

- If we ignore the cycle time overhead of pipelining and we assume the stages are perfectly balanced, the clock cycle time of two processors can be equal, so:

$$\text{Pipeline Speedup} = \frac{\text{Ave. CPI Unp.}}{1 + \text{Pipe Stall Cycles per Instruction}}$$

- Simple case: All instructions take the same number of cycles, which must also equal to the number of pipeline stages (called pipeline depth):

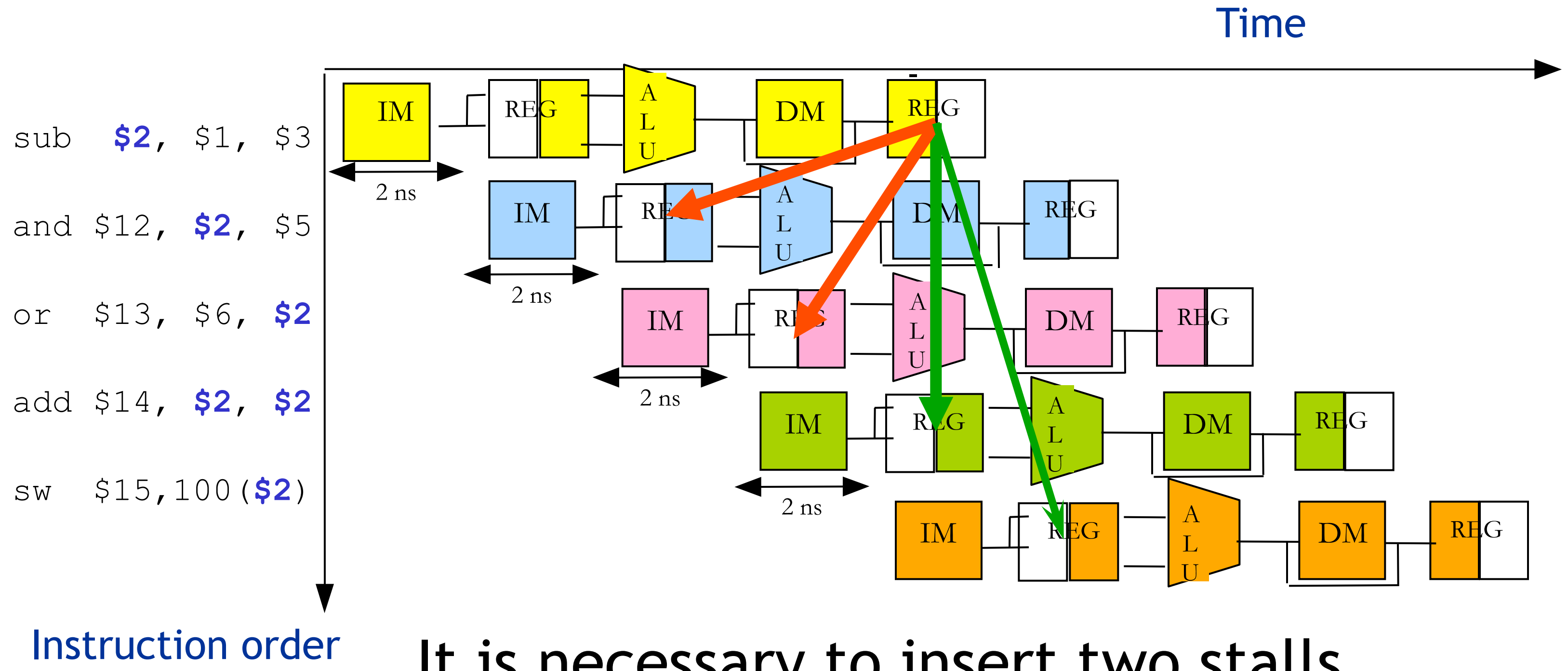
$$\text{Pipeline Speedup} = \frac{\text{Pipeline Depth}}{1 + \text{Pipe Stall Cycles per Instruction}}$$

- If there are no pipeline stalls (ideal case), this leads to the intuitive result that pipelining can improve performance by the depth of the pipeline.

Performances

- CI = # of Instructios
- # Clock Cycles = CI + # Stall Cycles + 4
- CPI = # Clock Cycles / CI = (CI + # Stall Cycles + 4) / CI
- MIPS = f_{clock} / (CPI * 10^{6})

Performance



It is necessary to insert two stalls

IC = 5

# Clock Cycles = IC + # Stall Cycles + 4 = 5 + 2 + 4 = 11

CPI = # Clock Cycles/ CI = 11 / 5 = 2.2

MIPS = f_{clock} / (CPI * 10^6) = 500 MHz / 2.2 * 10^6 = 227.3

Asymptotic Performance

- We have n iterations of a cycle defined using m instructions. We have k stalls for the m instructions

- $IC_{AS} = m * n$
- $\# \text{ Clock Cycles} = IC_{AS} + (\# \text{ Stall Cycles})_{AS} + 4$
- $CPI_{AS} = \lim_{n \to \infty} (IC_{AS} + \# \text{ Stall Cycles}_{AS} + 4) / IC_{AS}$

  $= \lim_{n \to \infty} (m * n + k * n + 4) / m * n$

  $= (m + k) / m$
- $MIPS_{AS} = f_{clock} / (CPI_{AS} * 10^6)$

Questions

I have no idea what you're talking about...
...so here's a bunny with a pancake on its head.