POLITECNICO DI MILANO

μ-LAB

High Performance Processors and Systems

Pipelining

Donatella Sciuto: sciuto@elet.polimi.it

HPPS

Outline

- Processors and Instruction Sets
- Review of pipelining
- Reduced Instruction Set of MIPS™ Processor
- Implementation of MIPS Processor Pipeline
- The Problem of Pipeline Hazards
- Performance Issues in Pipelining

Main Characteristics of MIPS™ Architecture

- **RISC (Reduced Instruction Set Computer) Architecture**
  Based on the concept of executing only simple instructions in a reduced basic cycle to optimize the performance of CISC CPUs.
- **LOAD/STORE Architecture**
  ALU operands come from the CPU general purpose registers and they cannot directly come from the memory.
  Dedicated instructions are necessary to:
  - *load* data from memory to registers
  - *store* data from registers to memory
- **Pipeline Architecture:**
  Performance optimization technique based on the overlapping of the execution of multiple instructions derived from a sequential execution flow.

A Typical RISC ISA

- 32-bit fixed format instruction (3 formats)
- 32 32-bit GPR (R0 contains zero, DP take pair)
- 3-address, reg-reg arithmetic instruction
- Single address mode for load/store:
  base + displacement
  - no indirection
- Simple branch conditions
- Delayed branch

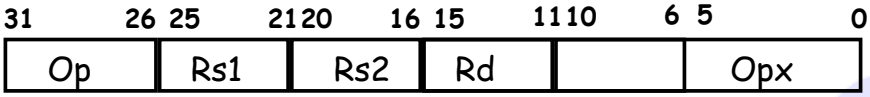
- Example: SPARC, MIPS, HP PA-Risc, DEC Alpha, IBM PowerPC, CDC 6600, CDC 7600, Cray-1, Cray-2, Cray-3

Approaching an ISA

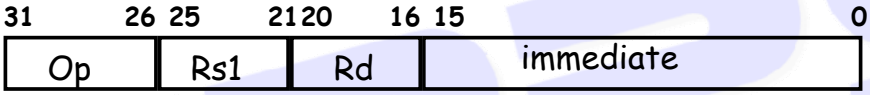
- Instruction Set Architecture
  - Defines set of operations, instruction format, hardware supported data types, named storage, addressing modes, sequencing
- Meaning of each instruction is described by RTL on *architected registers* and memory
- Given technology constraints assemble adequate datapath
  - Architected storage mapped to actual storage
  - Function units to do all the required operations
  - Possible additional storage (eg. MAR, MBR, …)
  - Interconnect to move information among regs and FUs
- Map each instruction to sequence of RTLs
- Collate sequences into symbolic controller state transition diagram (STD)
- Implement controller

Example: MIPS

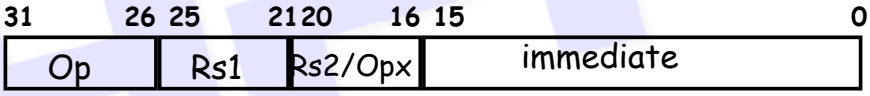
Register-Register



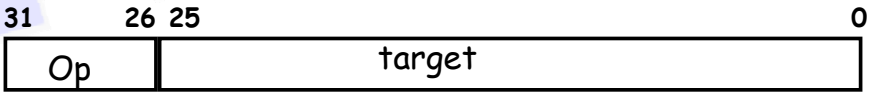
Register-Immediate



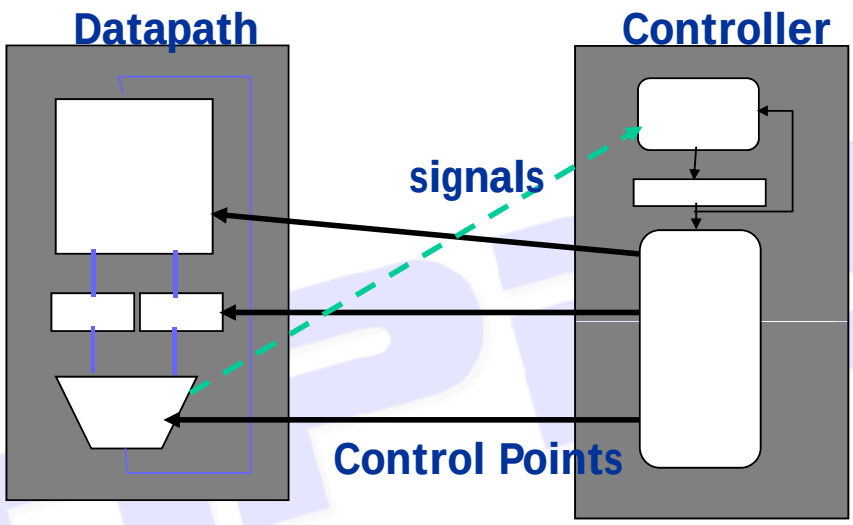
Branch



Jump / Call

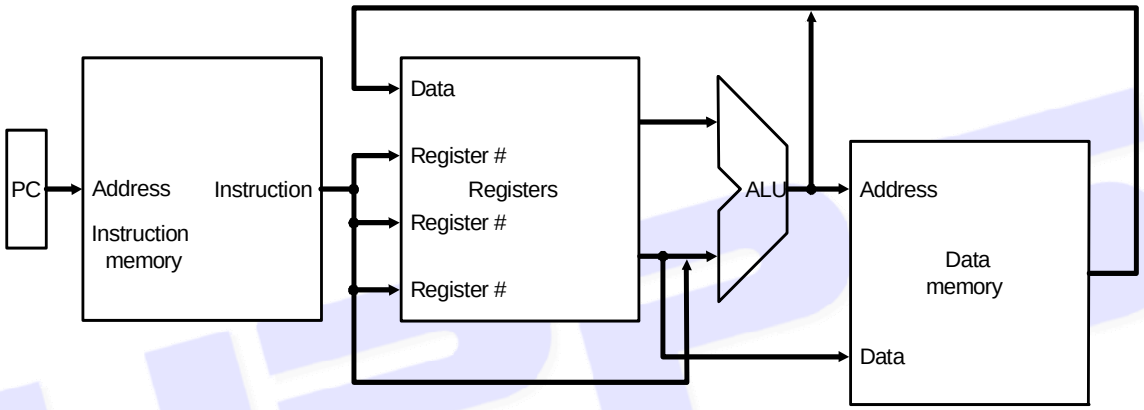


Datapath vs Control



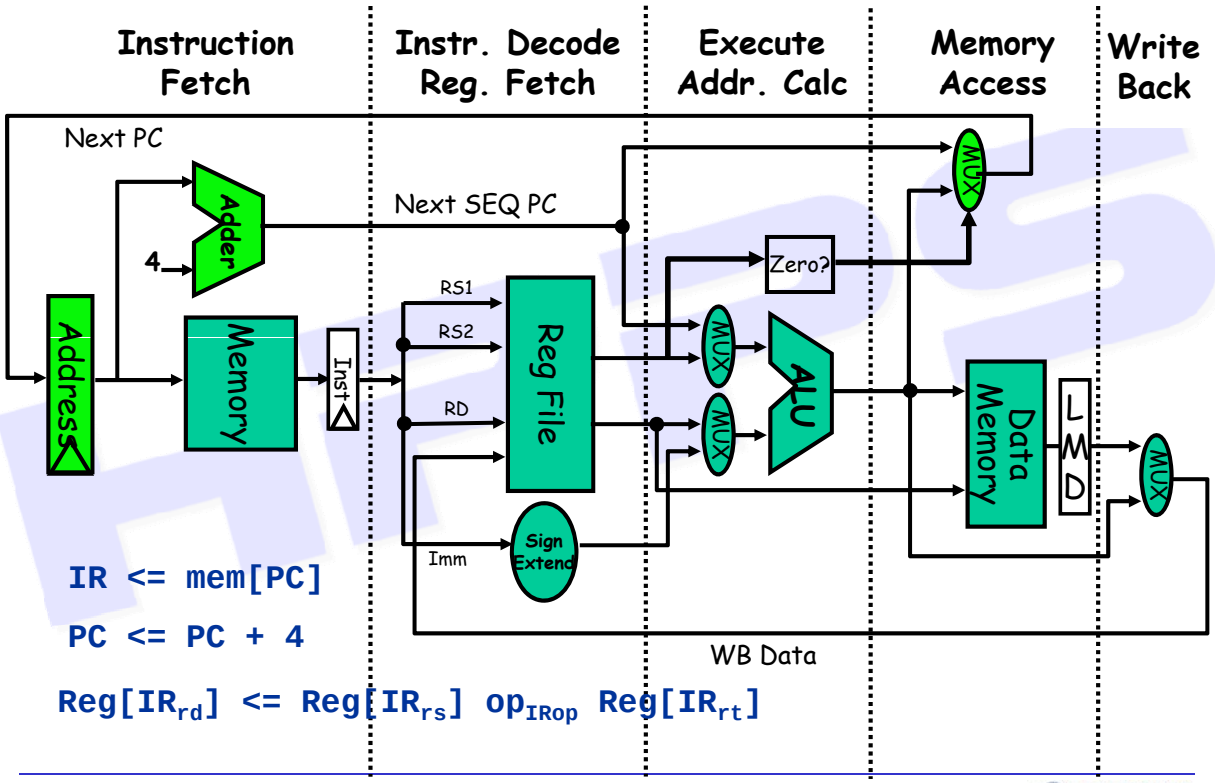
- Datapath: Storage, FU, interconnect sufficient to perform the desired functions
  - Inputs are Control Points
  - Outputs are signals
- Controller: State machine to orchestrate operation on the data path
  - Based on desired function and signals

Basic Implementation of MIPS data path



- Instruction Memory (read-only memory) separated from Data Memory
- 32 General-Purpose Registers organized in a Register File (RF) with 2 read ports and 1-write port.

MIPS Data path



Reduced Instruction Set of MIPS Processor

- ALU instructions:

```
add $s1, $s2, $s3        # $s1 ← $s2 + $s3
addi $s1, $s1, 4         # $s1 ← $s1 + 4
```

- Load/store instructions:

```
lw $s1, offset ($s2)     # $s1 ← M[$s2+offset]
sw $s1, offset ($s2)      M[$s2+offset] ← $s1
```

- Branch instructions to control the control flow of the program:
  - **Conditional branches:** the branch is taken only if the condition is satisfied. Examples: **beq** (*branch on equal*) and **bne** (*branch on not equal*)

```
beq $s1, $s2, L1    # go to L1 if ($s1 == $s2)
bne $s1, $s2, L1    # go to L1 if ($s1 != $s2)
```

  - **Unconditional jumps:** the branch is always taken. Examples: **j** *(jump)* and **jr** *(jump register)*

```
j   L1              # go to L1
jr  $s1             # go to add. contained in $s1
```

Execution of MIPS Instructions

Every instruction in the MIPS subset can be implemented in **at most 5 clock cycles** as follows:

- **Instruction Fetch Cycle:**
  - Send the content of **Program Counter** register to **Instruction Memory** and fetch the current instruction from Instruction Memory.
    Update the PC to the next sequential address by adding 4 to the PC (since each instruction is 4 bytes).
- **Instruction Decode and Register Read Cycle**
  - Decode the current instruction **(fixed-field decoding)** and read from the Register File of one or two registers corresponding to the registers specified in the instruction fields.
  - Sign-extension of the offset field of the instruction in case it is needed.

Execution of MIPS instructions

- **Execution Cycle**

  The ALU operates on the operands prepared in the previous cycle depending on the instruction type:

  - Register-Register ALU Instructions:
    - ALU executes the specified operation on the operands read from the RF
  - Register-Immediate ALU Instructions:
    - ALU executes the specified operation on the first operand read from the RF and the sign-extended immediate operand
  - Memory Reference:
    - ALU adds the base register and the offset to calculate the **effective address.**
  - Conditional branches:
    - Compare the two registers read from RF and compute the possible **branch target address** by adding the sign-extended offset to the incremented PC.

Execution of MIPS instructions

- **Memory Access (ME)**
  - *Load* instructions require a read access to the Data Memory using the effective address
  - *Store* instructions require a write access to the Data Memory using the effective address to write the data from the source register read from the RF
  - Conditional branches can update the content of the PC with the branch target address, if the conditional test yielded true.
- **Write-Back Cycle (WB)**
  - Load instructions write the data read from memory in the destination register of the RF
  - ALU instructions write the ALU results into the destination register of the RF.

Execution of MIPS Instructions

ALU Instructions: `op $x,$y,$z`

Instr. Fetch & PC Increm.	Read of Source Regs. `$y` and `$z`	ALU OP (`$y op $z`)	Write Back of Destinat. Reg. `$x`

Load Instructions: `lw $x,offset($y)`

Instr. Fetch & PC Increm.	Read of Base Reg. `$y`	ALU Op. (`$y+offset`)	Read Mem. `M($y+offset)`	Write Back of Destinat. Reg. `$x`

Store Instructions: `sw $x,offset($y)`

Instr. Fetch & PC Increm.	Read of Base Reg. `$y` & Source `$x`	ALU Op. (`$y+offset`)	Write Mem. `M($y+offset)`

Conditional Branch: `beq $x,$y,offset`

Instr. Fetch & PC Increm.	Read of Source Regs. `$x` and `$y`	ALU Op. (`$x-$y`) & (`PC+4+offset`)	Write PC

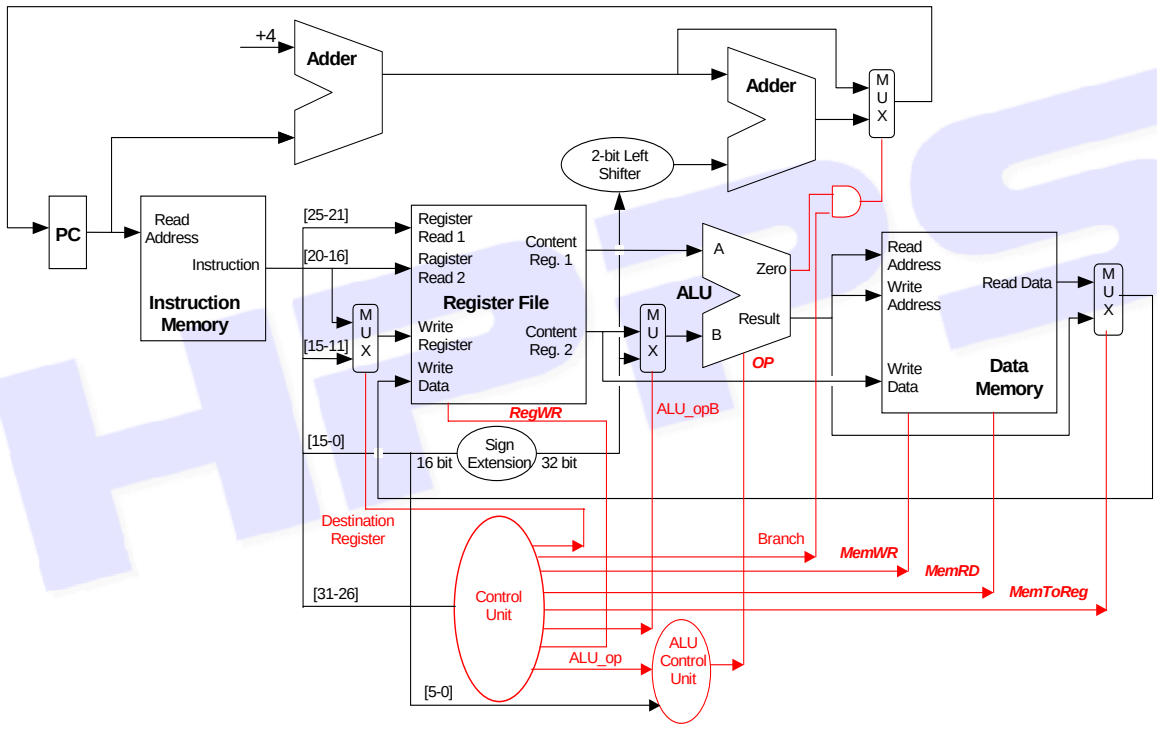
Instructions Latency

Instruction Type	Instruct. Mem.	Register Read	ALU Op.	Data Memory	Write Back	Total Latency
ALU Instr.	2	1	2	0	1	6 ns
Load	2	1	2	2	1	8 ns
Store	2	1	2	2	0	7 ns
Cond. Branch	2	1	2	0	0	5 ns
Jump	2	0	0	0	0	2 ns

Single-cycle Implementation of MIPS

- The length of the clock cycle is defined by the critical path given by the load instruction: *T = 8 ns (f = 125 MHz).*
- We assume each instruction is executed in a **single clock cycle**
  - Each module must be used once in a clock
  - The modules used more than once in a cycle must be duplicated.
- We need an Instruction Memory separated from the Data Memory.
- Some modules must be duplicated, while other modules must be shared from different instruction flows
- To share a module between two different instructions, we need a **multiplexer** to enable multiple inputs to a module and select one of different inputs based on the configuration of control lines.

Implementation of MIPS data path with Control Unit



Multi-cycle Implementation

- The instruction execution is distributed on multiple cycles (5 cycles for MIPS)
- The basic cycle is smaller
  (2 ns ⇒ instruction latency = 10 ns)
- Implementation of multi-cycle CPU:
  - Each phase of the instruction execution requires a clock cycle
  - Each module can be used more than once per instruction in different clock cycles: possible sharing of modules
  - We need internal registers to store the values to be used in the next clock cycles.

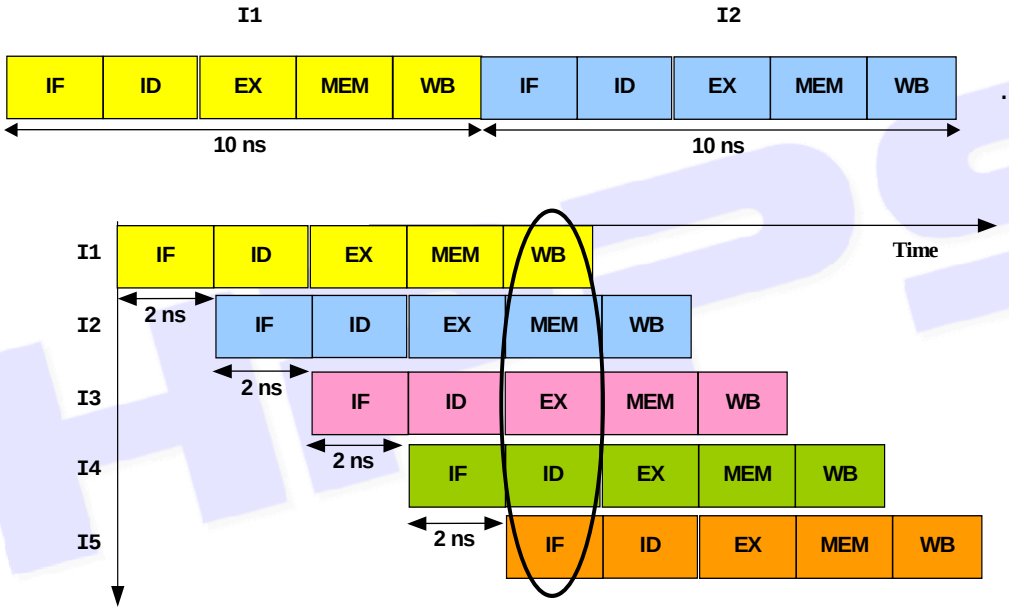
Pipelining

- Performance optimization technique based on the overlap of the execution of multiple instructions deriving from a sequential execution flow.
- Pipelining exploits the parallelism among instructions in a sequential instruction stream.
- **Basic idea:**
  The execution of an instruction is divided into different phases **(pipelines stages),** requiring a fraction of the time necessary to complete the instruction.
- The stages are connected one to the next to form the pipeline: instructions enter in the pipeline at one end, progress through the stages, and exit from the other end, as in an assembly line.

Pipelining

- Advantage: technique transparent for the programmer.
- Technique similar to a assembly line: a new car exits from the assembly line in the time necessary to complete one of the phases.
- An assembly line does not reduce the time necessary to complete a car, but increases the number of cars produced simultaneously and the frequency to complete cars.

Sequential vs. Pipelining Execution



Pipelining

- The time to advance the instruction of one stage in the pipeline corresponds to a clock cycle.
- The pipeline stages must be **synchronized:** the duration of a clock cycle is defined by the time requested by the slower stage of the pipeline *(i.e. 2 ns).*
- The goal is to **balance** the length of each pipeline stage
- If the stages are perfectly balanced, the **ideal speedup** due to pipelining is equal to the number of pipeline stages.

Performance Improvement

- Ideal case (asymptotically):If we consider the single-cycle unpipelined *CPU1* with clock cycle of *8 ns* and the pipelined *CPU2* with 5 stages of 2 ns :
  - The **latency** (total execution time) of each instruction is worsened: from *8 ns* to *10 ns*
  - The **throughput** (number of instructions completed in the time unit) is improved of **4 times:**
    (1 instruction completed each *8 ns)* vs.
    (1 instruction completed each *2 ns)*

Performance Improvement

- Ideal case (asymptotically): If we consider the multi-cycle unpipelined *CPU3* composed of 5 cycles of *2 ns* and the pipelined *CPU2* with 5 stages of 2 ns :
  - The **latency** (total execution time) of each instruction is not varied *(10 ns)*
  - The **throughput** (number of instructions completed in the time unit) is improved of *5* **times:**
    (1 instruction completed every *10 ns) vs.*
    (1 instruction completed every *2 ns)*

Pipeline Execution of MIPS Instructions

IF Instruction Fetch	ID Instruction Decode	EX Execution	ME Memory Access	WB Write Back

ALU Instructions: `op $x,$y,$z`

Instr. Fetch & PC Increm.	Read of Source Regs. `$y` and `$z`	ALU Op. (`$y op $z`)		Write Back Destinat. Reg. `$x`

Load Instructions: `lw $x,offset($y)`

Instr. Fetch & PC Increm.	Read of Base Reg. `$y`	ALU Op. (`$y+offset`)	Read Mem. `M($y+offset)`	Write Back Destinat. Reg. `$x`

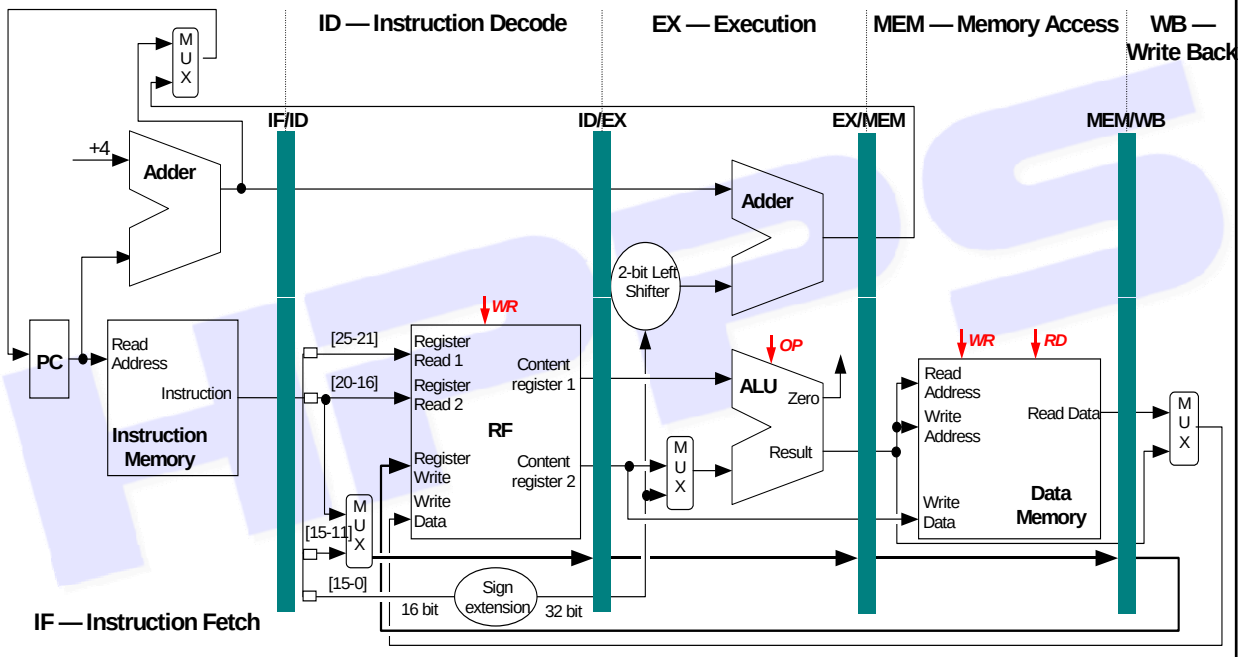
Store Instructions: `sw $x,offset($y)`

Instr. Fetch & PC Increm.	Read of Base Reg. `$y` & Source `$x`	ALU Op. (`$y+offset`)	Write Mem. `M($y+offset)`	

Conditional Branches: `beq $x,$y,offset`

Instr. Fetch & PC Increm.	Read of Source Regs. `$x` and `$y`	ALU Op. (`$x-$y`) & (`PC+4+offset`)	Write PC	

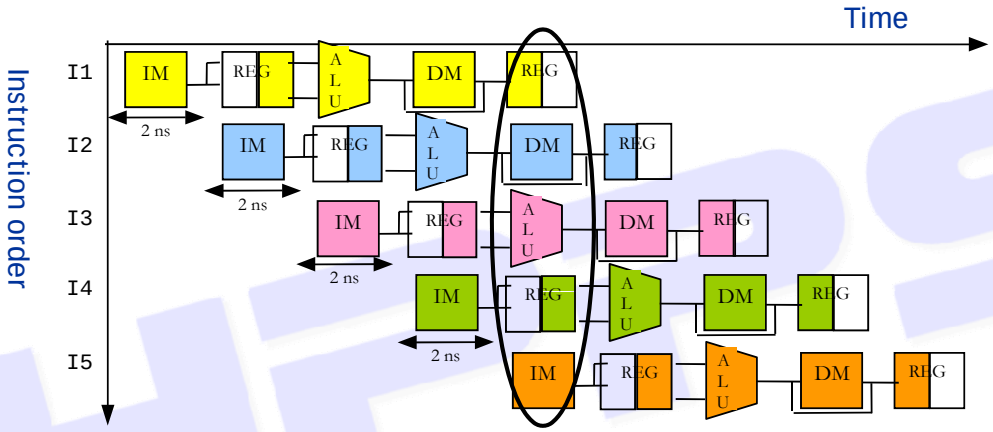
Implementation of MIPS pipeline



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
  - Therefore a read and write can refer to the same register in the same clock cycle

Resources Used in the Optimized Pipeline

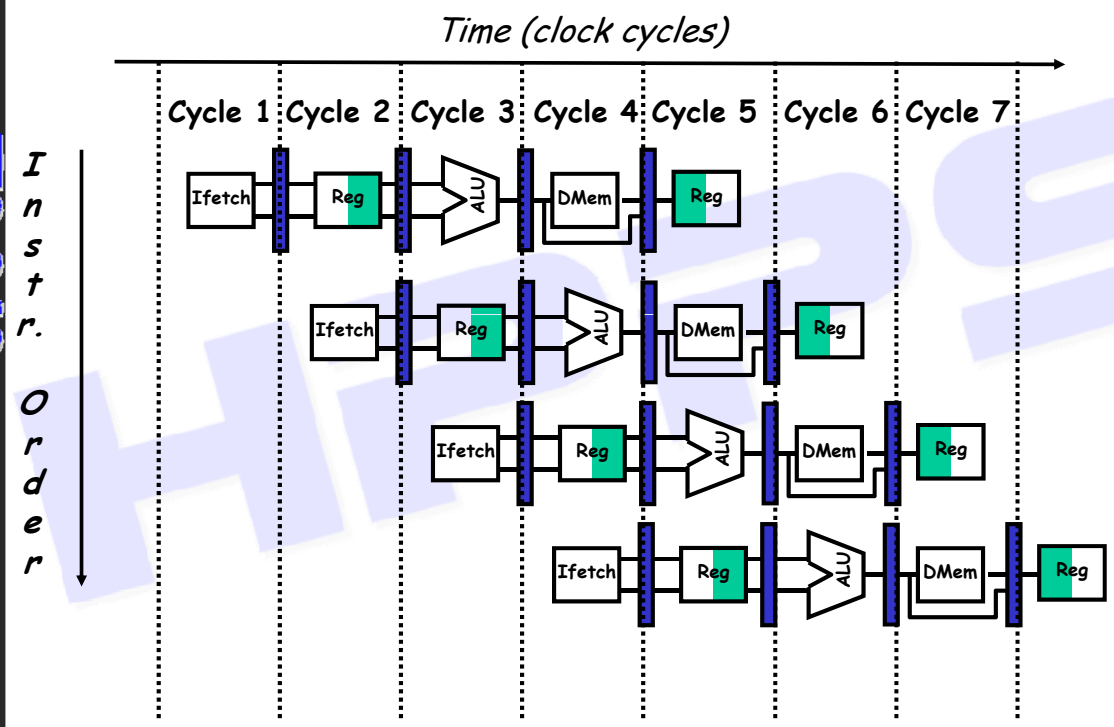


IM = Instruction Memory

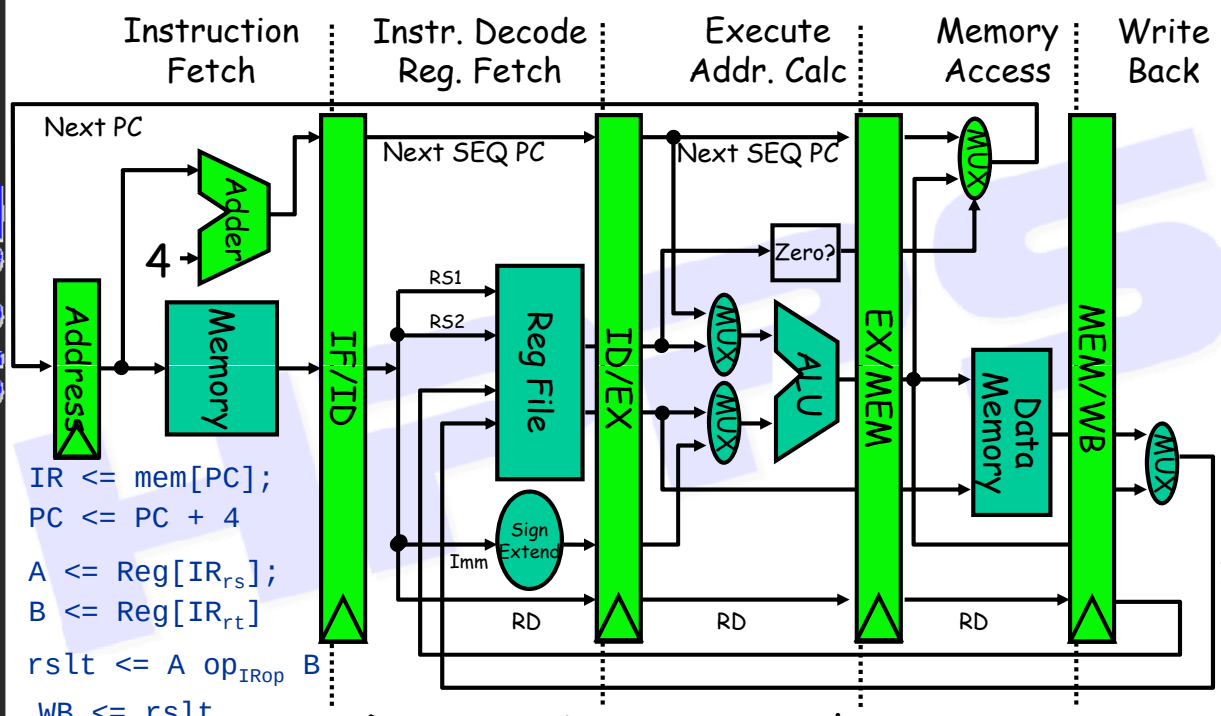
REG = Register File

DM = Data Memory

Visualizing Pipelining



5 Steps of MIPS Datapath



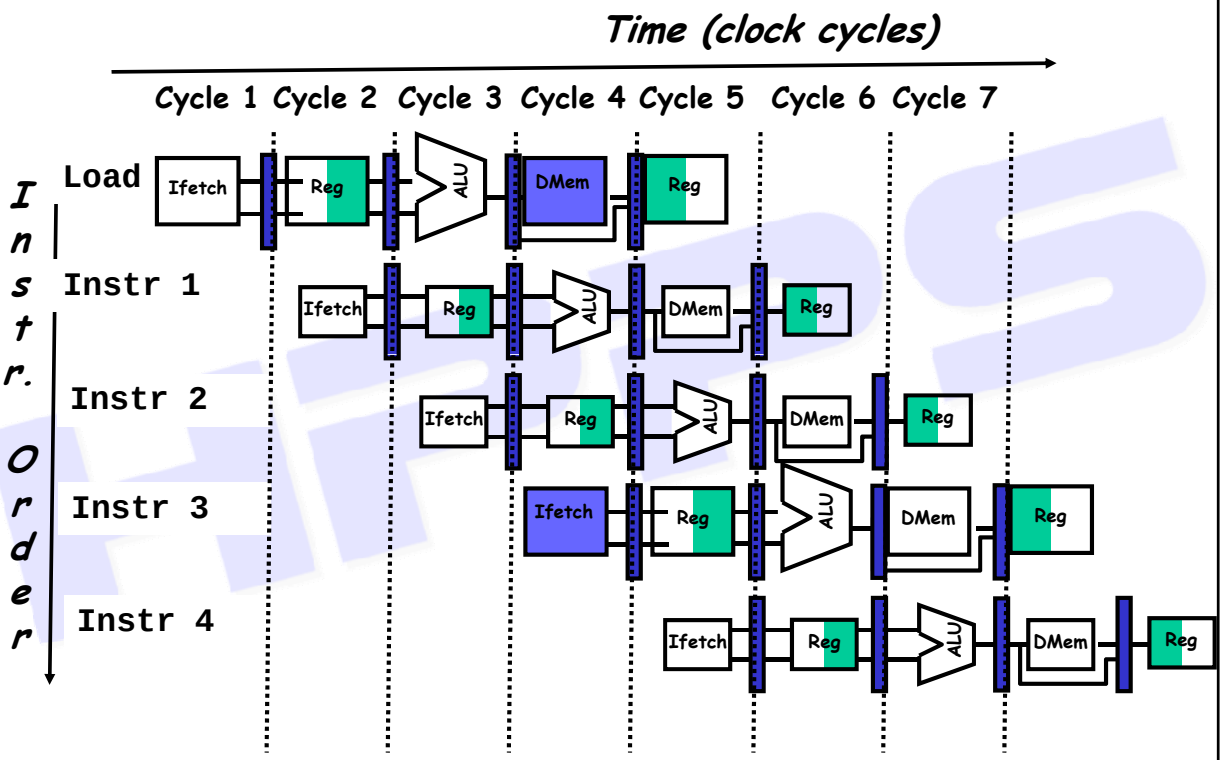
The Problem of Hazards

- 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

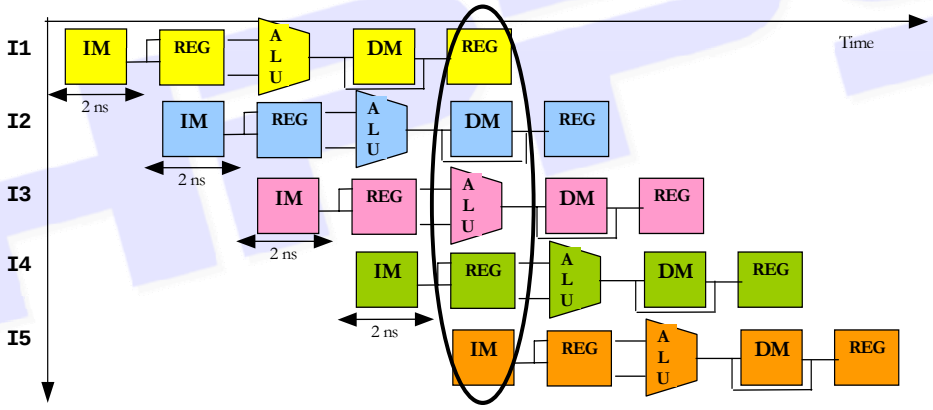
- **Structural Hazards:** Attempt to use the same resource from different instructions simultaneously
  - Example: Single memory for instructions and data
- **Data Hazards:** Attempt to use a result before it is ready
  - Example: Instruction depending on a result of a previous instruction still in the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

One Memory Port/Structural Hazards



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:

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

Example: Dual-port vs. Single-port

- Machine A: Dual ported memory ("Harvard Architecture")
- Machine B: Single ported memory, but its pipelined implementation has a 1.05 times faster clock rate
- Ideal CPI = 1 for both
- Loads are 40% of instructions executed

$SpeedUp_A$ = Pipeline Depth/(1 + 0) x ($clock_{unpipe}$/$clock_{pipe}$)
= Pipeline Depth

$SpeedUp_B$ = Pipeline Depth/(1 + 0.4 x 1) x ($clock_{unpipe}$/($clock_{unpipe}$ / 1.05)
= (Pipeline Depth/1.4) x 1.05
= 0.75 x Pipeline Depth

$SpeedUp_A$ / $SpeedUp_B$ = Pipeline Depth/(0.75 x Pipeline Depth) = 1.33

- Machine A is 1.33 times faster

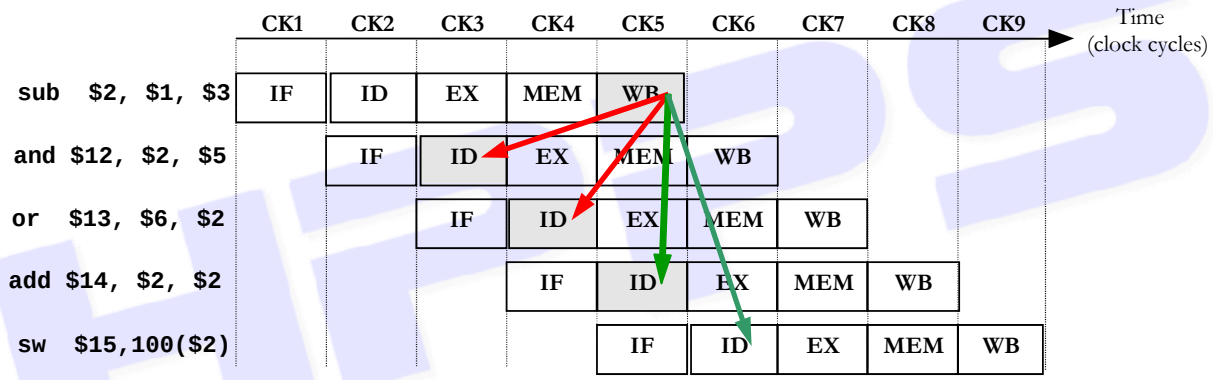
Data Hazards

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

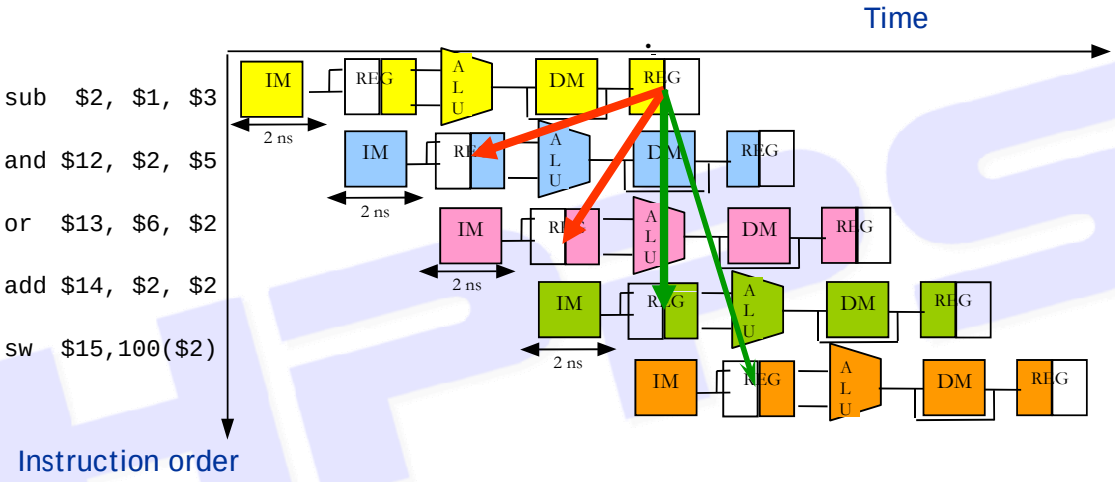
- Example:

```
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
```

Just an example…

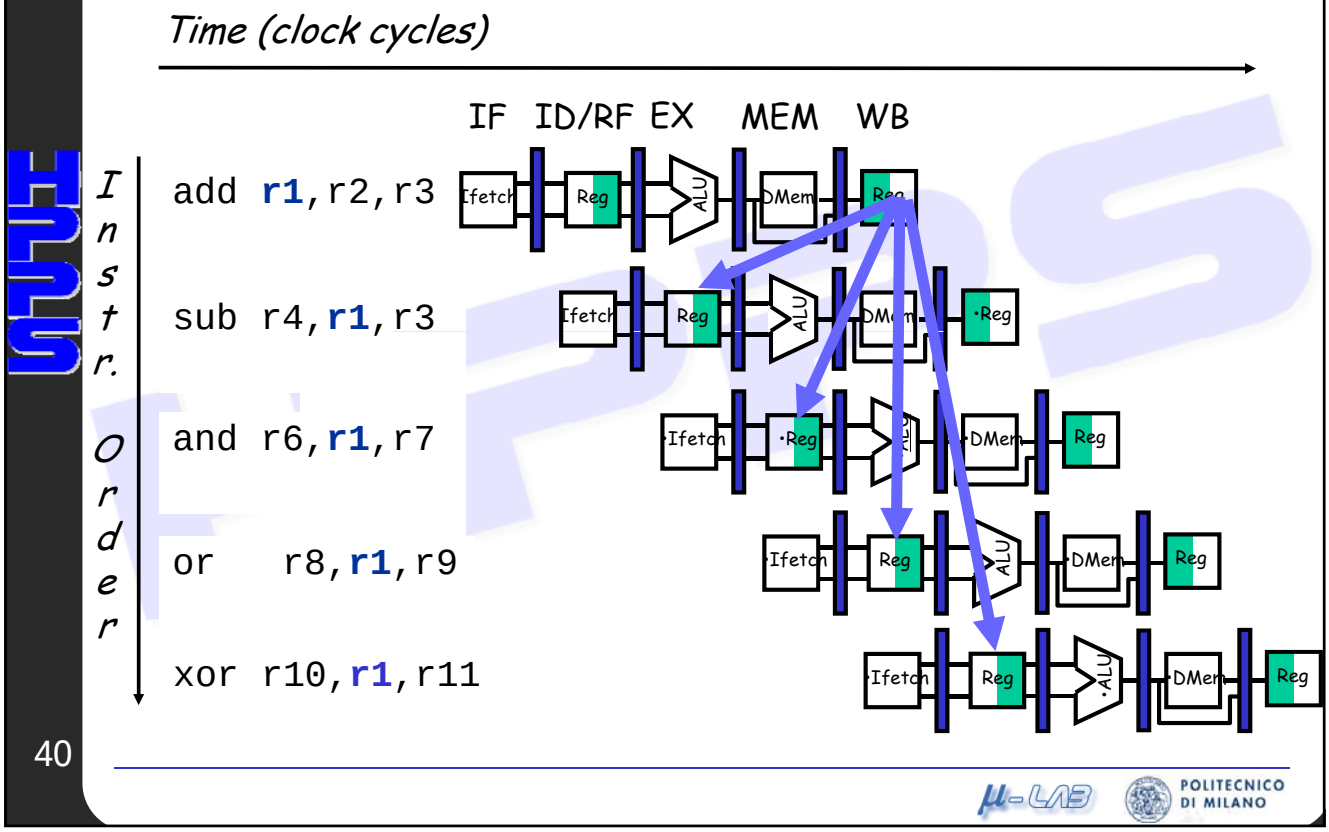


Data Hazards: Example



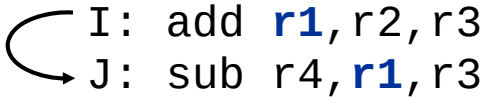
It is necessary to insert two stalls

Data Hazard on R1 example



Type of Data Hazard

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



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

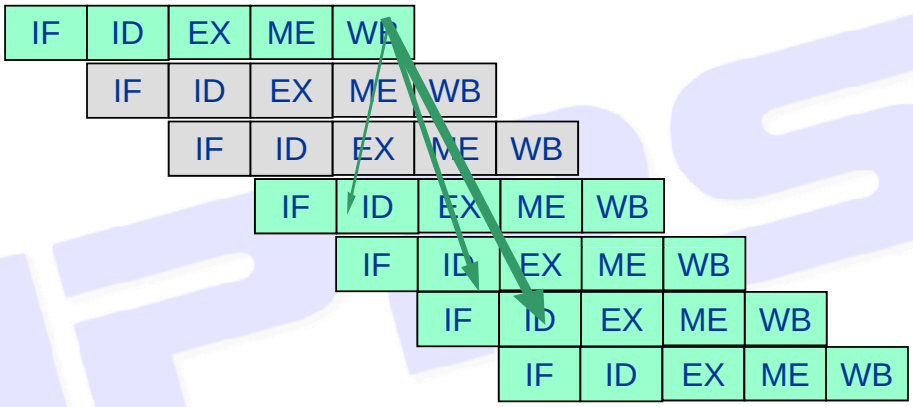
Data Hazards: Possible Solutions

- **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

Insertion of **nops**: Example

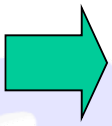
```
sub  $2, $1, $3
nop
nop

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



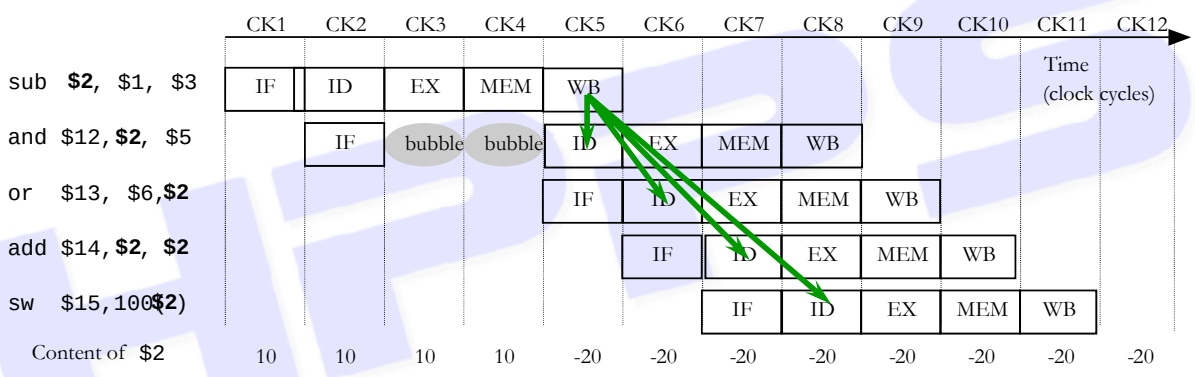
Scheduling: Example

```
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)
```

Insertion of stalls

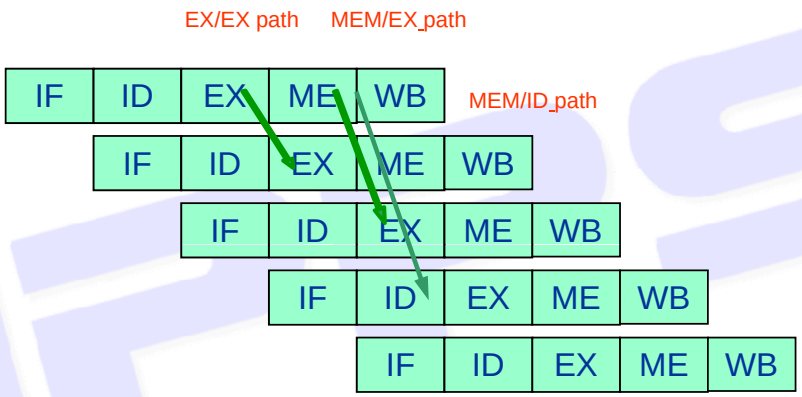


Forwarding

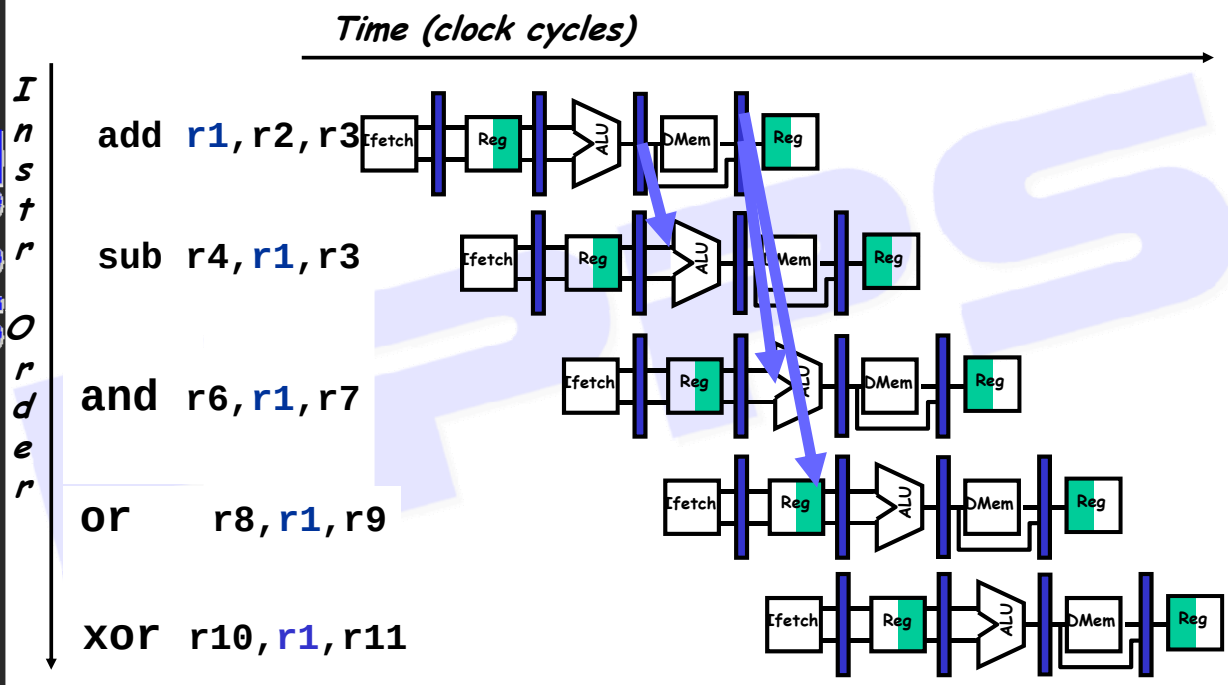
- Data forwarding uses temporary results stored in the pipeline registers instead of waiting for the write back of results in the RF.
- We need to add **multiplexers** at the inputs of ALU to fetch inputs from pipeline registers to avoid the insertion of stalls in the pipeline.

Forwarding: Example

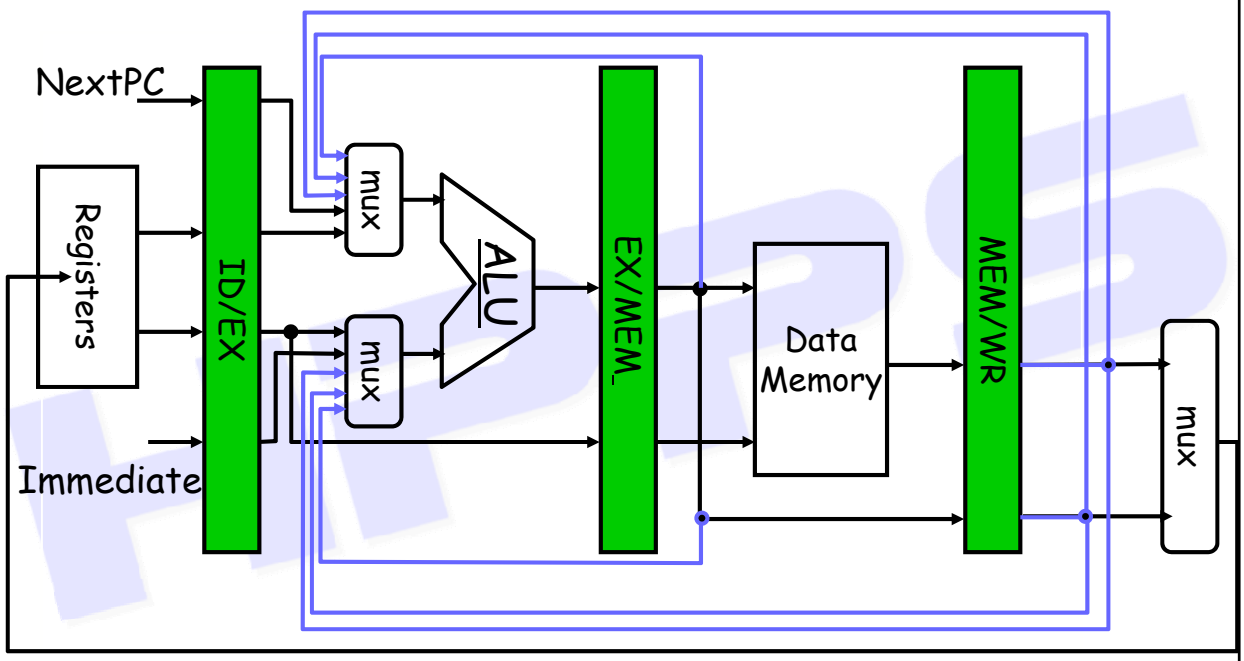
```
sub  $2, $1, $3
and $12, $2, $5
or  $13, $6, $2
add $14, $2, $2
sw  $15,100($2)
```



Forwarding to Avoid Data Hazard

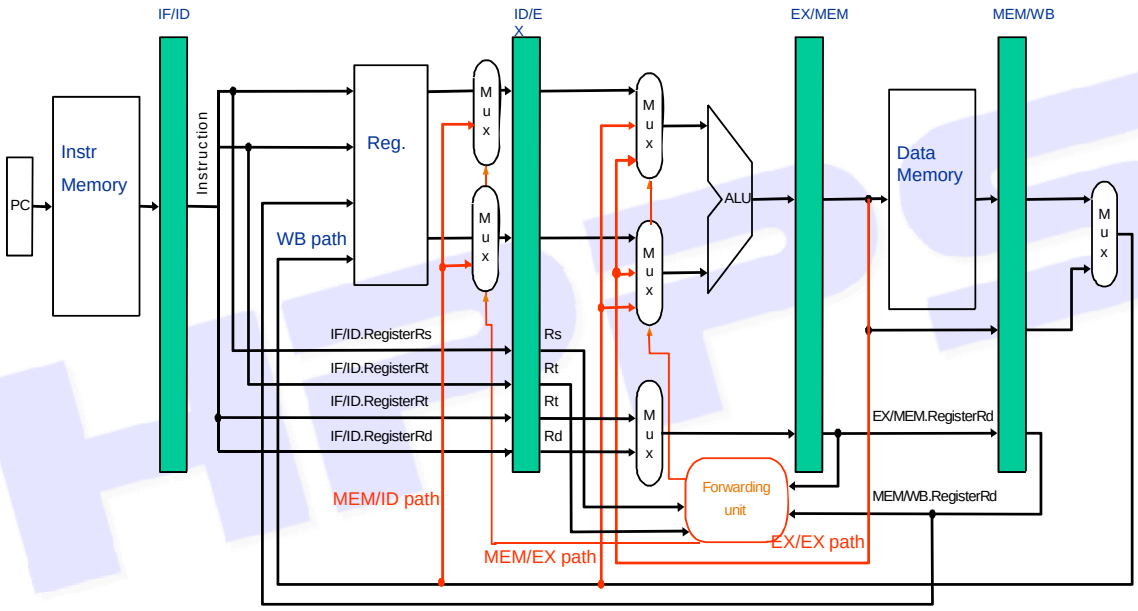


HW Change for Forwarding



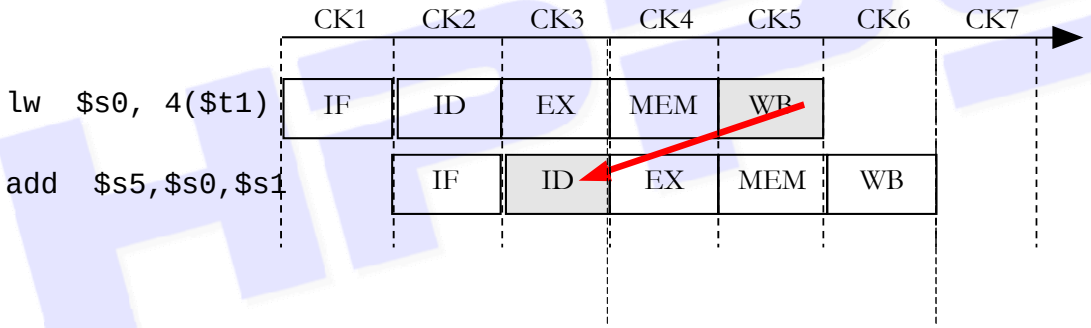
What circuit detects and resolves this hazard?

Implementation of MIPS with Forwarding Unit



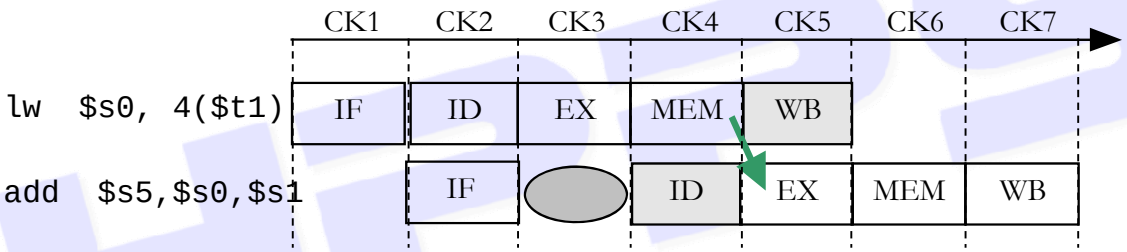
Data Hazards: Load/Use Hazard

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



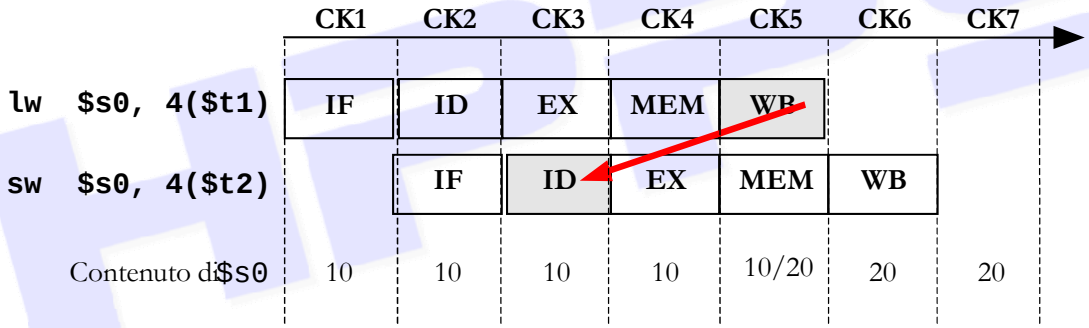
Data Hazards: Load/Use Hazard

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



Data Hazards: Load/Store

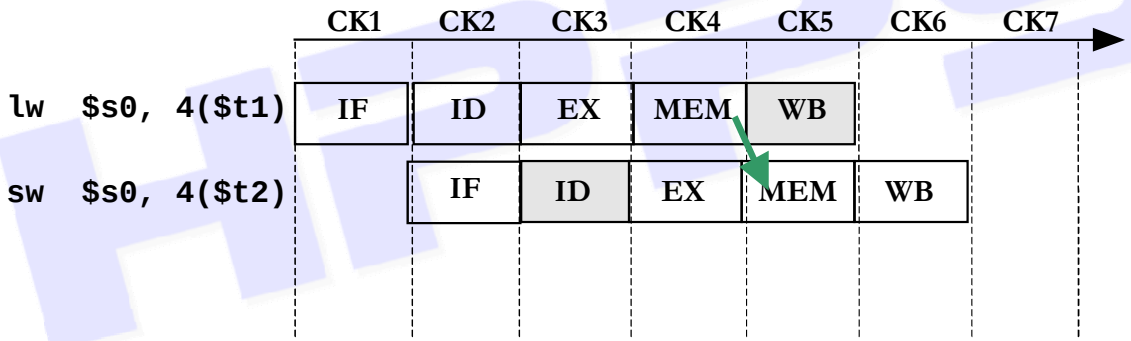
```
L1: lw $s0, 4($t1)    # $s0 <- M [4 + $t1]
L2: sw $s0, 4($t2)    # M[4 + $t2] <- $s0
```



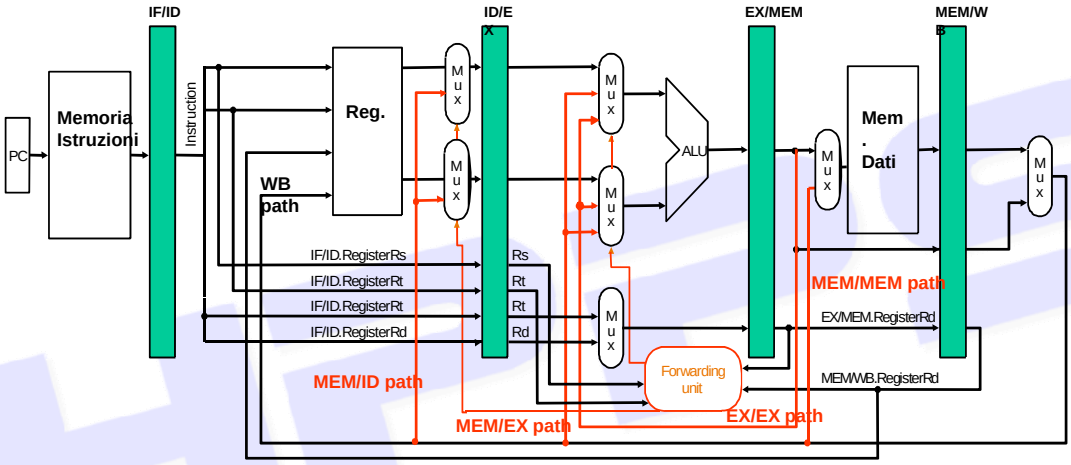
- Without forwarding : 3 stalls

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*.

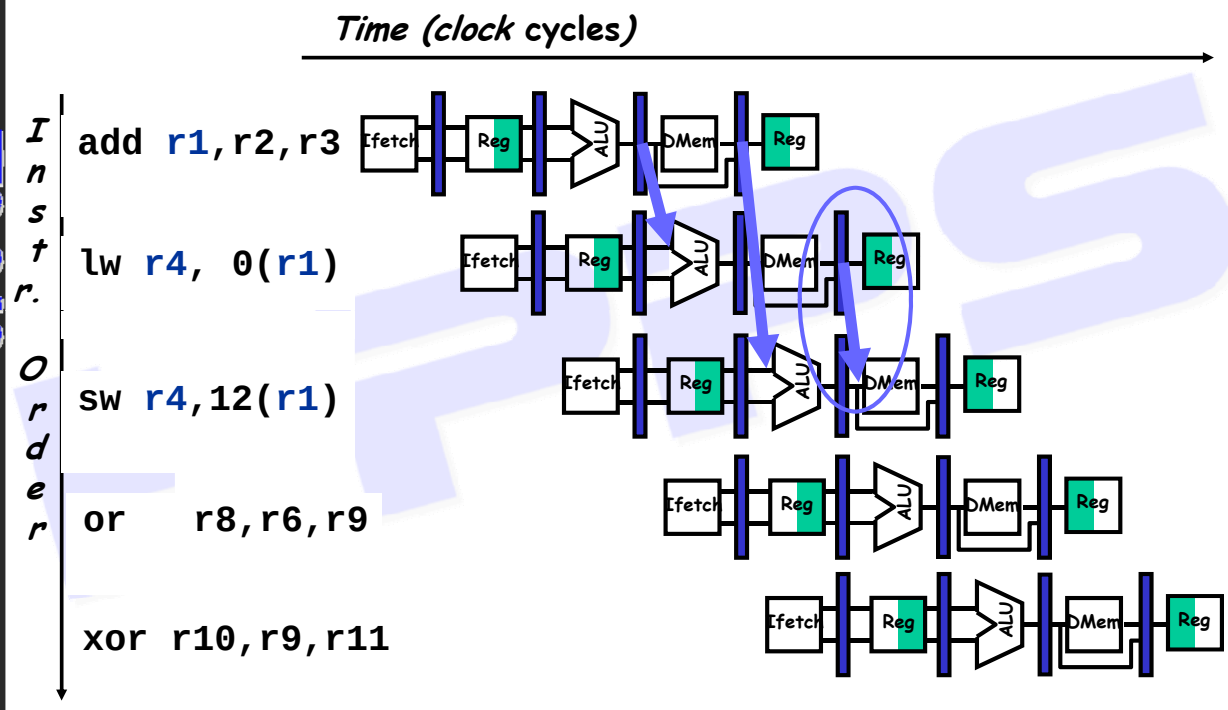


Implementation of MIPS with Forwarding Unit

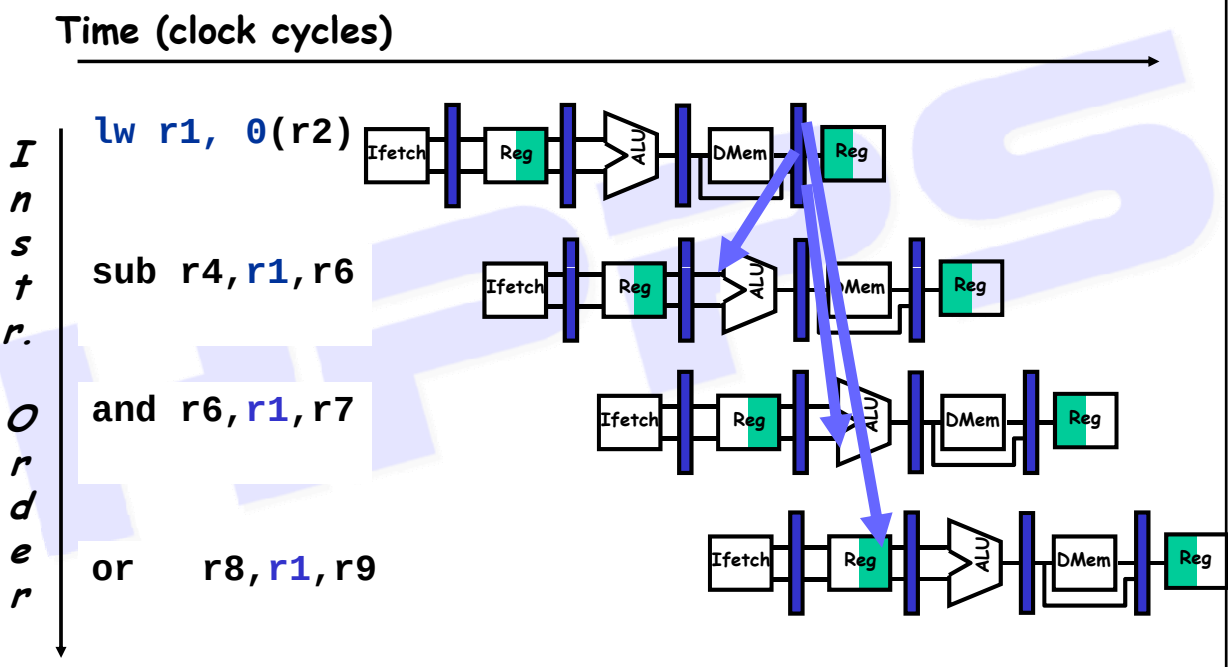


- **EX/EX path**
- **MEM/EX path**
- **MEM/ID path**
- **MEM/MEM path**

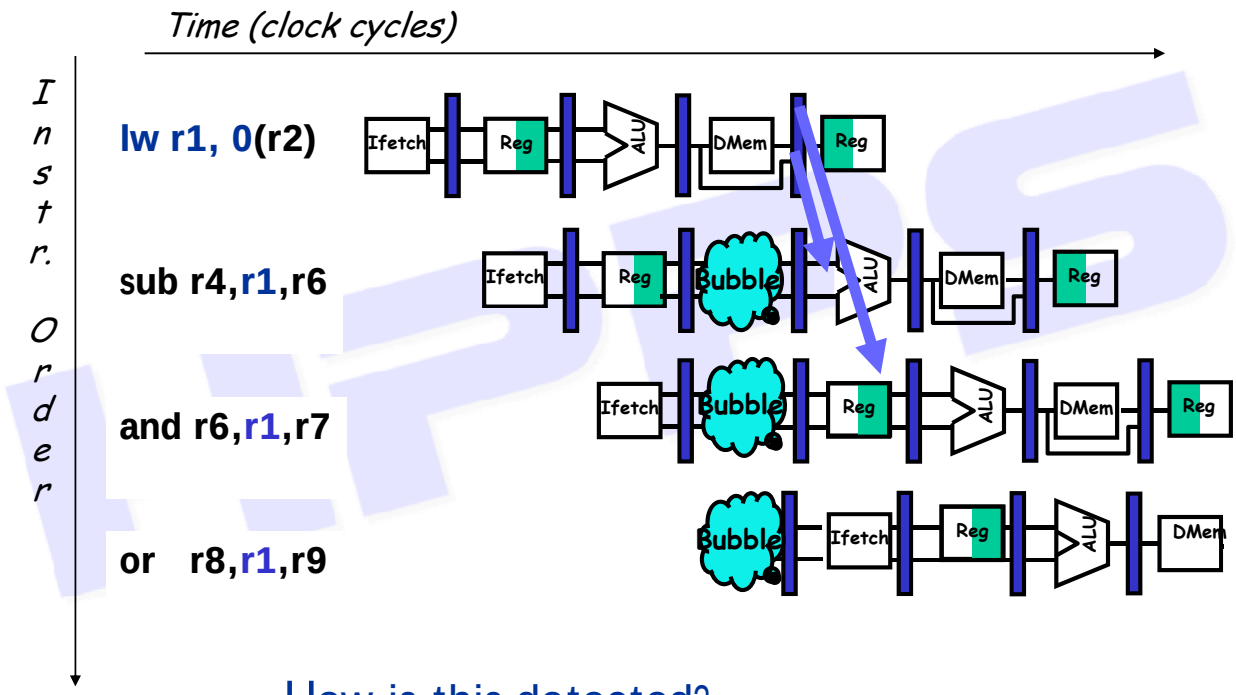
Forwarding to Avoid LW-SW Data Hazard



Data Hazard Even with Forwarding



Data Hazard Even with Forwarding



How is this detected?

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
```

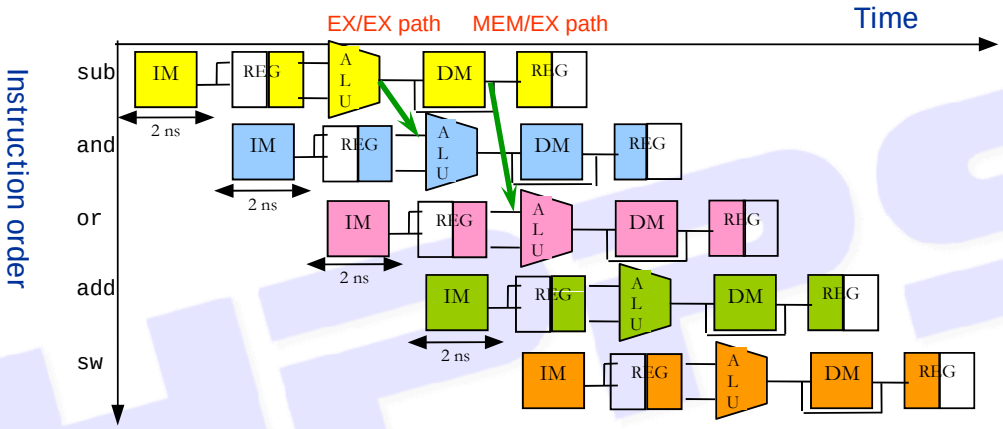
Fast code:

```
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.

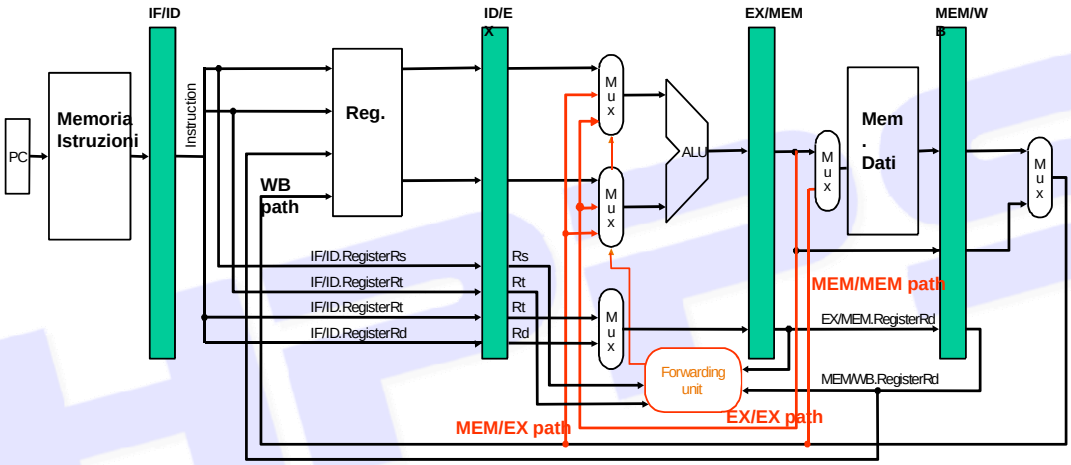
Hardware checks for safety.

Forwarding: Example



- Two forwarding paths:
  - EX/EX path
  - MEM/EX path

Implementation of MIPS with Forwarding Unit



- **EX/EX path**
- **MEM/EX path**
- **MEM/MEM path**

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)**

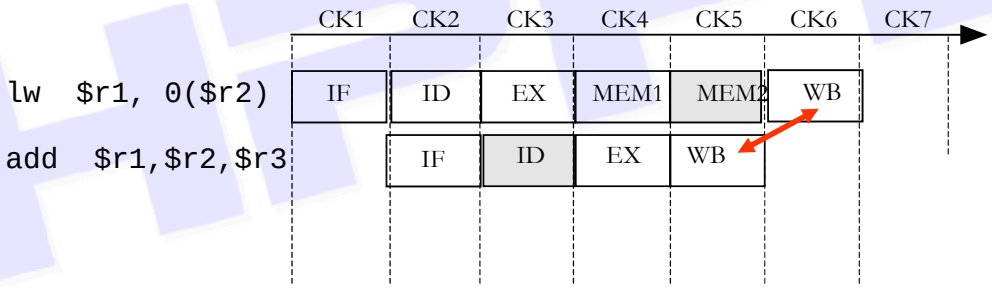
Data Hazards: WAW (WRITE AFTER WRITE)

- 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

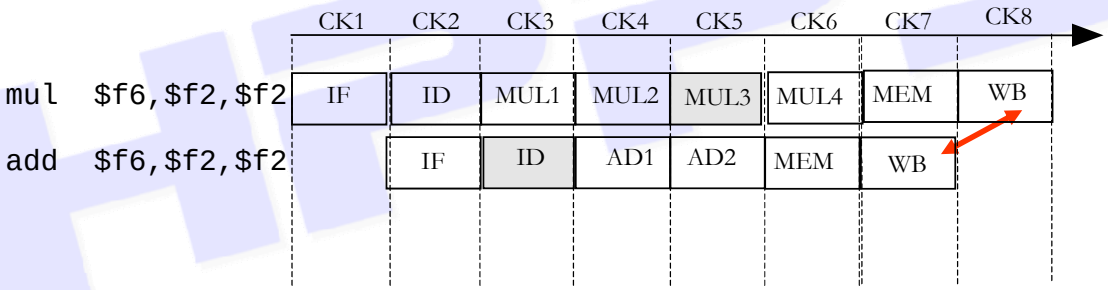
Data Hazards: WAW (WRITE AFTER WRITE)

- 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:



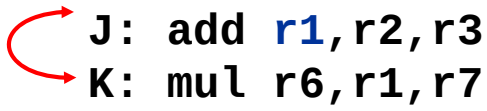
Data Hazards: WAW (WRITE AFTER WRITE)

- 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.



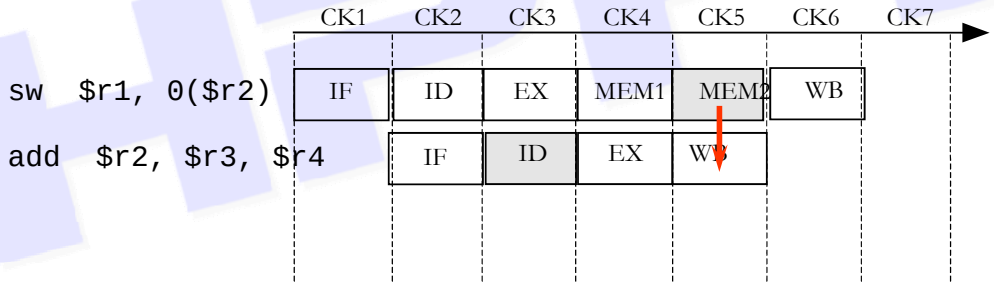
- Called an "output dependence" by compiler writers
  This also results from the reuse of name "r1".
- Can't happen in MIPS 5 stage pipeline because:
  - All instructions take 5 stages, and
  - Writes are always in stage 5
- Will see WAR and WAW in more complicated pipes

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* ⇒ 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.

Pipeline Speedup = Ave. Exec. Time Unpipelined =
Ave. Exec. Time Pipelined

= Ave. CPI Unp. x Clock Cycle Unp. =
Ave. CPI Pipe 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):

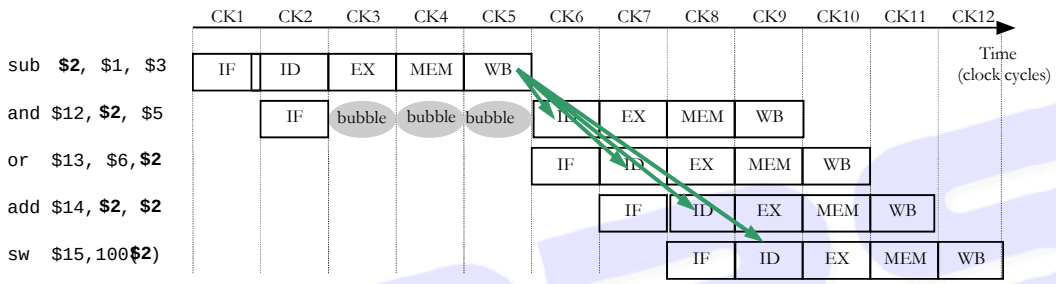
  $$\text{Pipeline Speedup} = \frac{\text{Pipeline Depth}}{1 + \text{Pipe Stall Cycles per Instruction}}$$

- 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.

Performance

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

Example…



- IC = 5
- # Clock Cycles = IC + # Stall Cycles + 4 = 5 + 3 + 4 = 12
- CPI = # Clock Cycles/ CI = 12 / 5 = 2.4
- MIPS = f_{clock} / (CPI * 10^6) = 500 MHz / 2.4 * 10^6 = 208.3

Asymptotic Performance

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

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

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

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

Questions