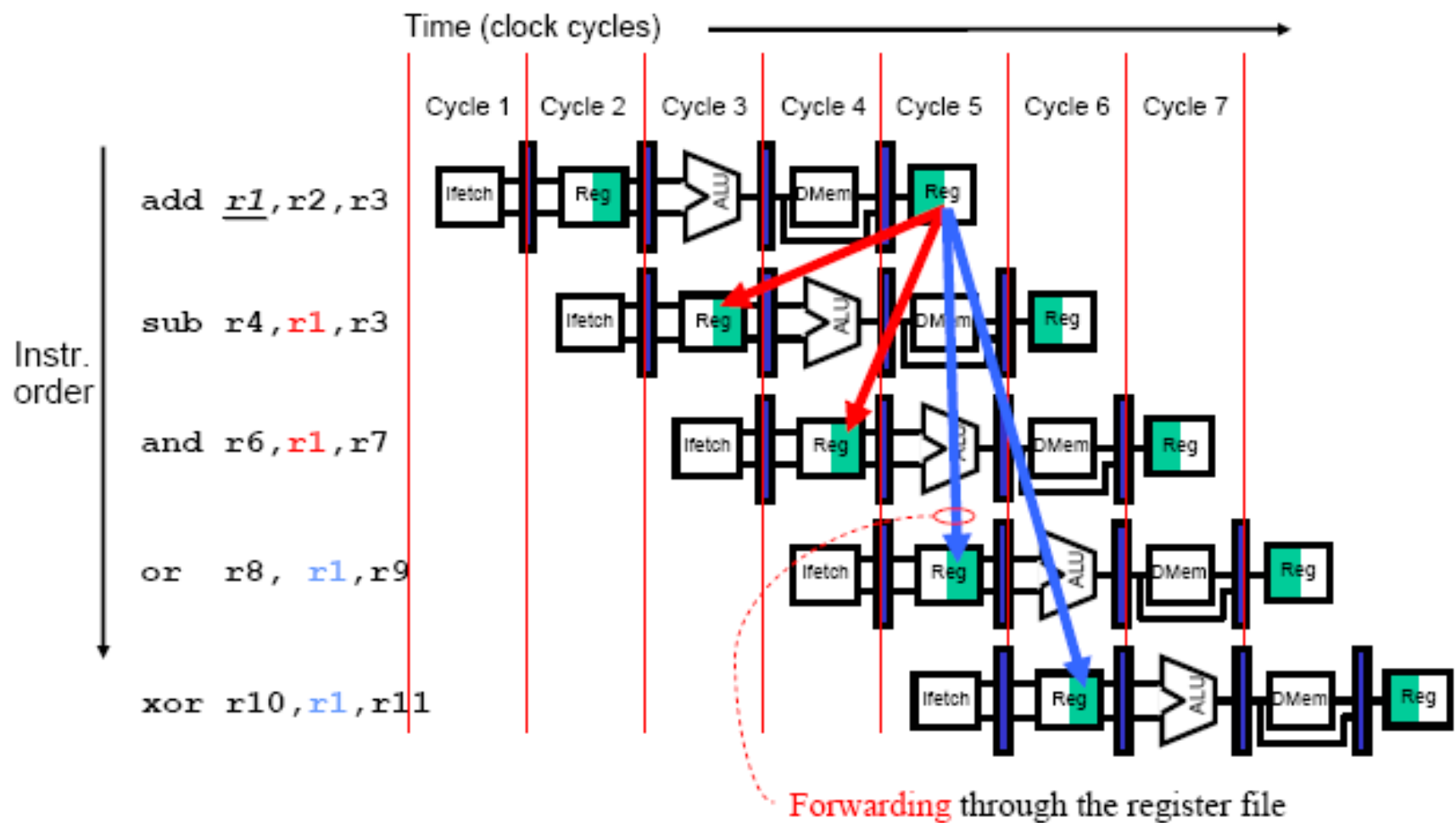


Techniques to Reduce Data Hazards

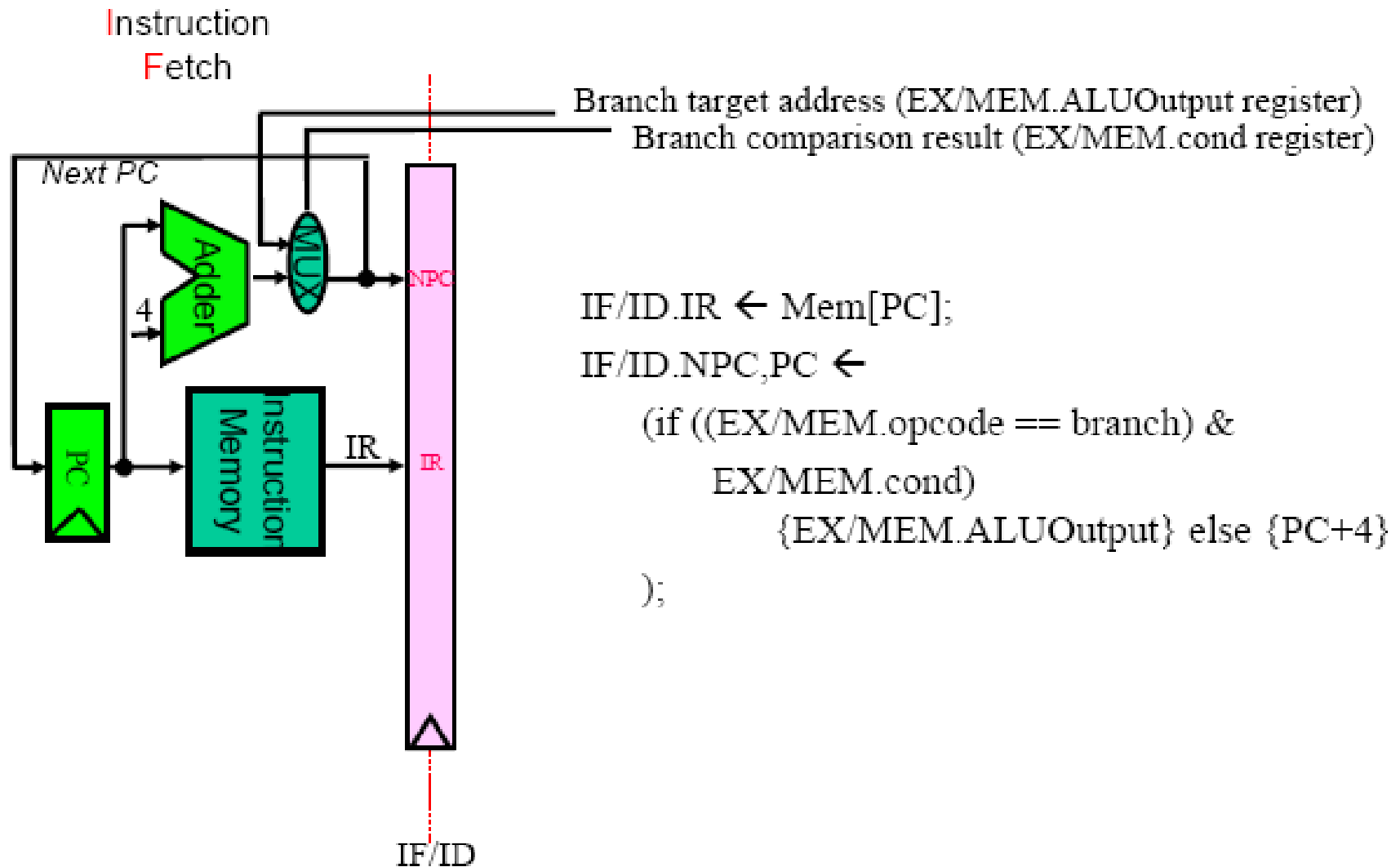
Data Hazard Example



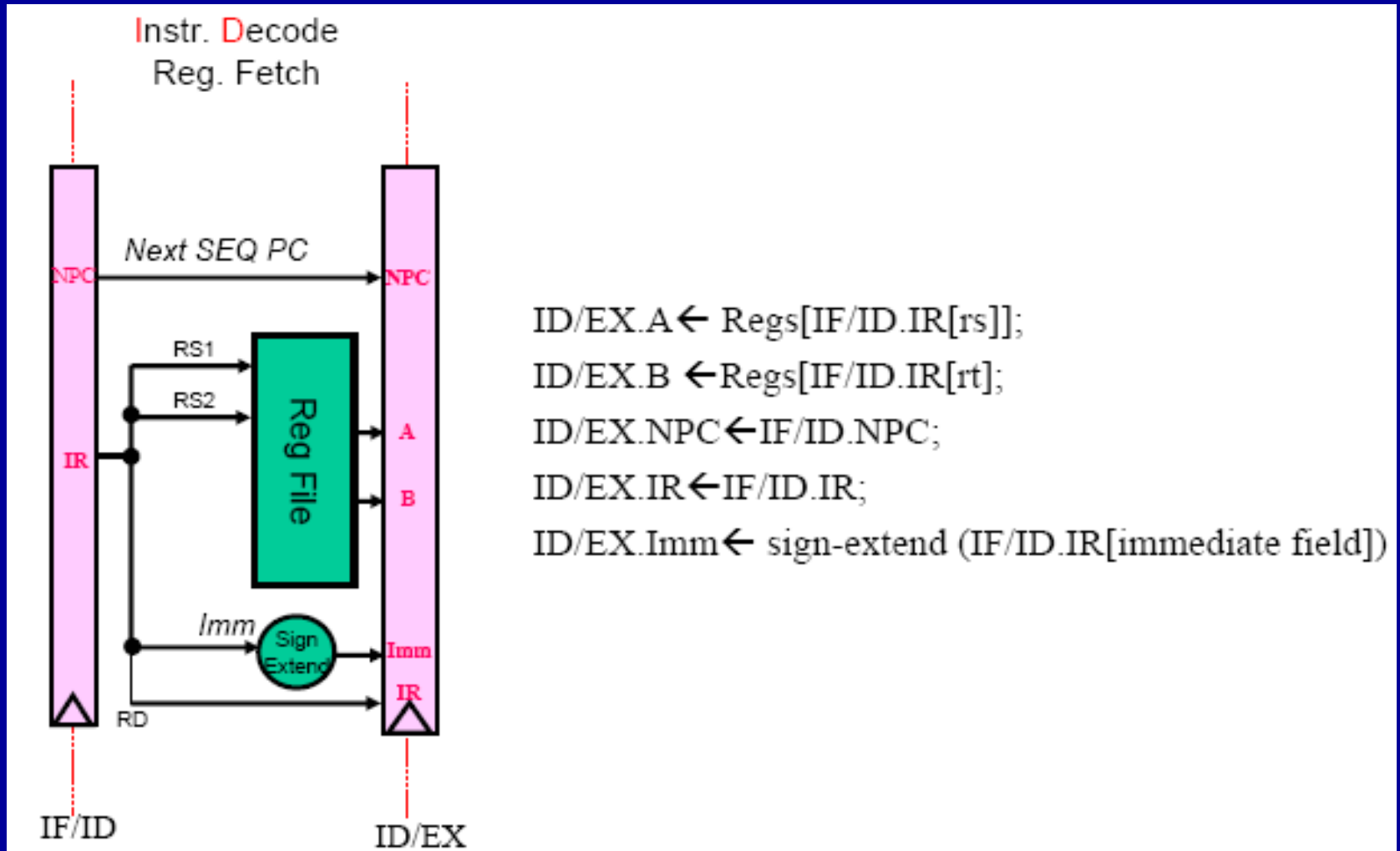
Techniques to Reduce Data Hazards

- Forwarding and bypassing
- Basic compiler pipeline scheduling
- Basic dynamic scheduling
- Dynamic scheduling with renaming
- Hardware speculation

Instruction Fetch Stage

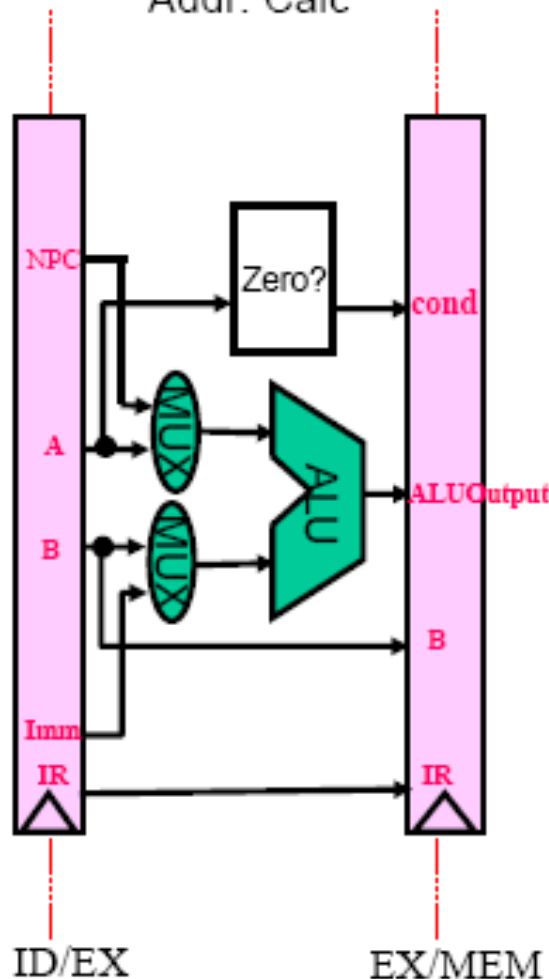


Instruction Decode Stage



Instruction Execute Stage

Execute
Addr. Calc



ALU instruction

$EX/MEM.IR \leftarrow ID/EX.IR;$

$EX/MEM.ALUOutput \leftarrow$
 $ID/EX.A \text{ func } ID/EX.B;$

or

$EX/MEM.ALUOutput \leftarrow$
 $ID/EX.A \text{ op } ID/EX.Imm;$

Branch instruction

$EX/MEM.ALUOutput \leftarrow$
 $ID/EX.NPC +$
 $(ID/EX.Imm \ll 2);$

$EX/MEM.cond \leftarrow$
 $(ID/EX.A == 0);$

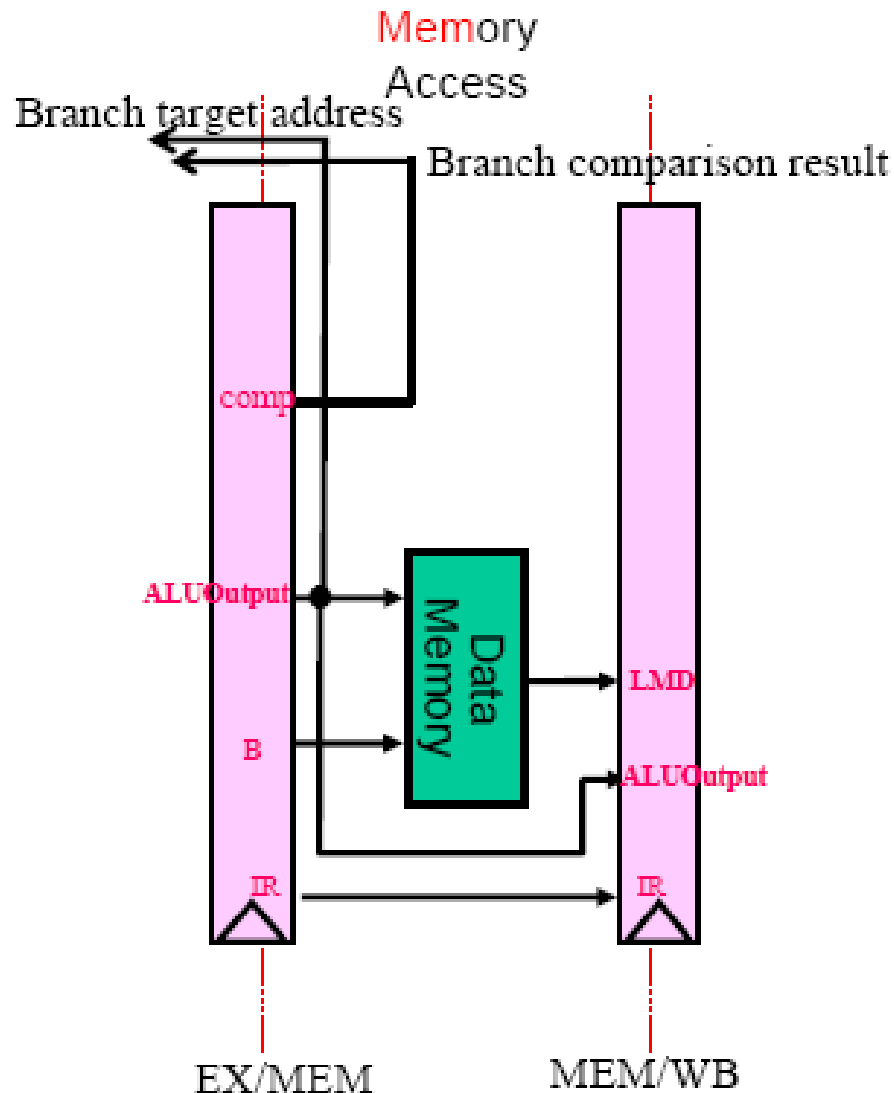
Load/store instruction

$EX/MEM.IR \leftarrow ID/EX.IR;$

$EX/MEM.ALUOutput \leftarrow$
 $ID/EX.A + ID/EX.Imm;$

$EX/MEM.B \leftarrow ID/EX.B;$

Memory Access Stage



ALU instruction

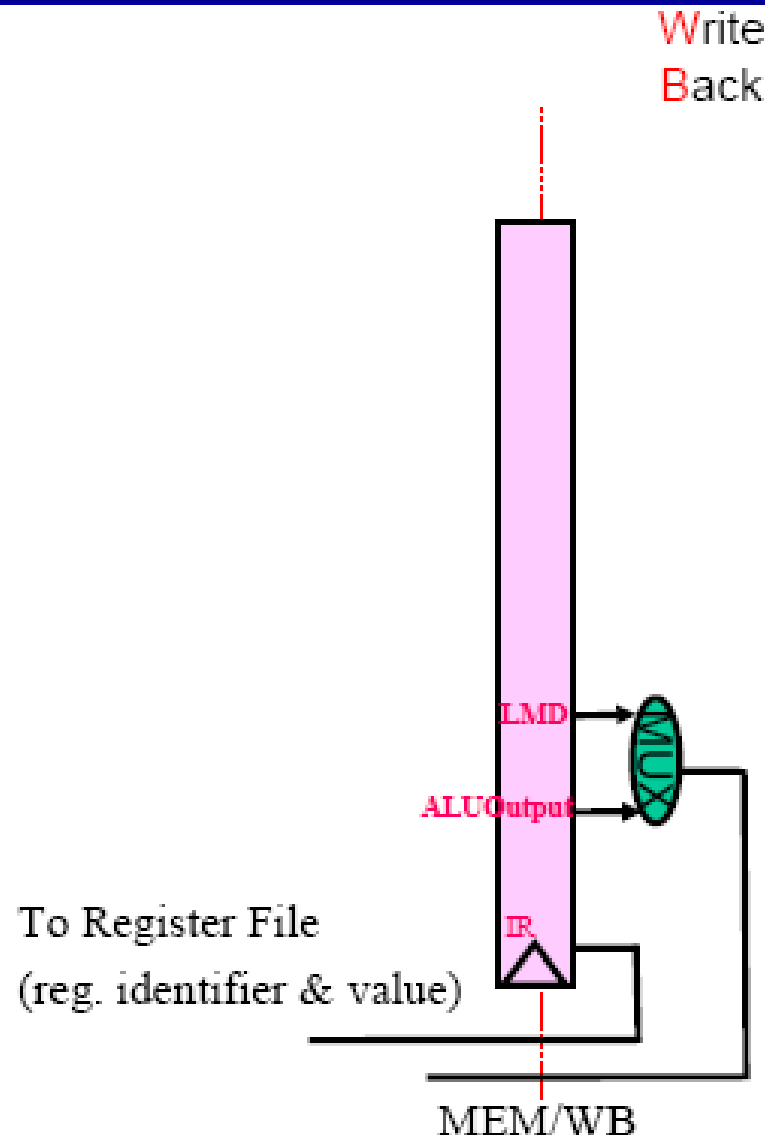
MEM/WB.IR \leftarrow EX/MEM.IR;
 MEM.WB.ALUOutput \leftarrow
 EX/MEM.ALUOutput;

Load/store instruction

MEM/WB.IR \leftarrow EX/MEM.IR;

MEM/WB.LMD \leftarrow
 Mem[EX/MEM.ALUOutput];
 or
 Mem[EX/MEM.ALUOutput] \leftarrow
 EX/MEM.B;

Write Back Stage



ALU instruction

$\text{Regs}[\text{MEM}/\text{WB}.\text{IR}[\text{rd}]] \leftarrow$
 $\text{MEM}.\text{WB}.\text{ALUOutput};$

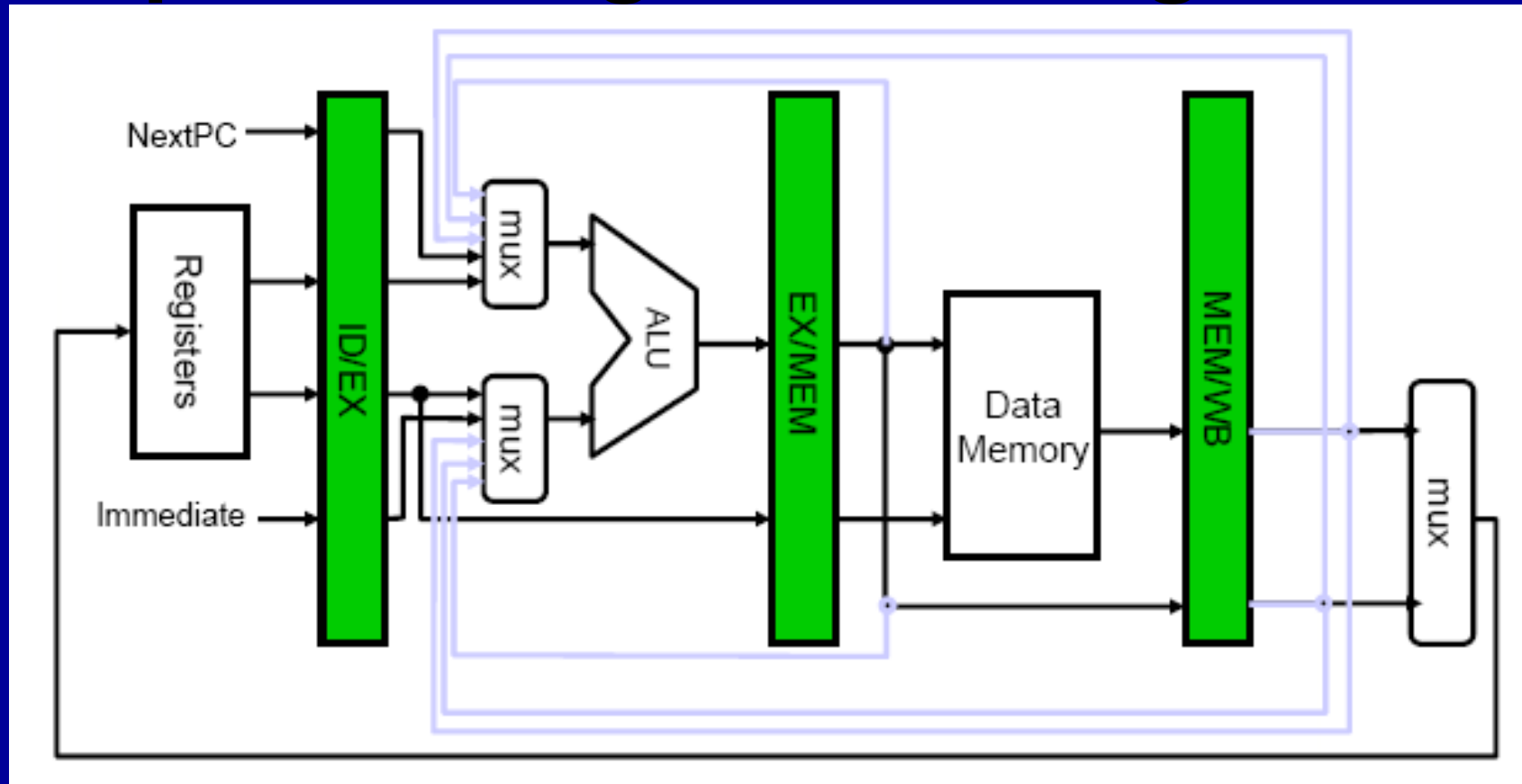
or

$\text{Regs}[\text{MEM}/\text{WB}.\text{IR}[\text{rt}]] \leftarrow$
 $\text{MEM}.\text{WB}.\text{ALUOutput};$

Load instruction only

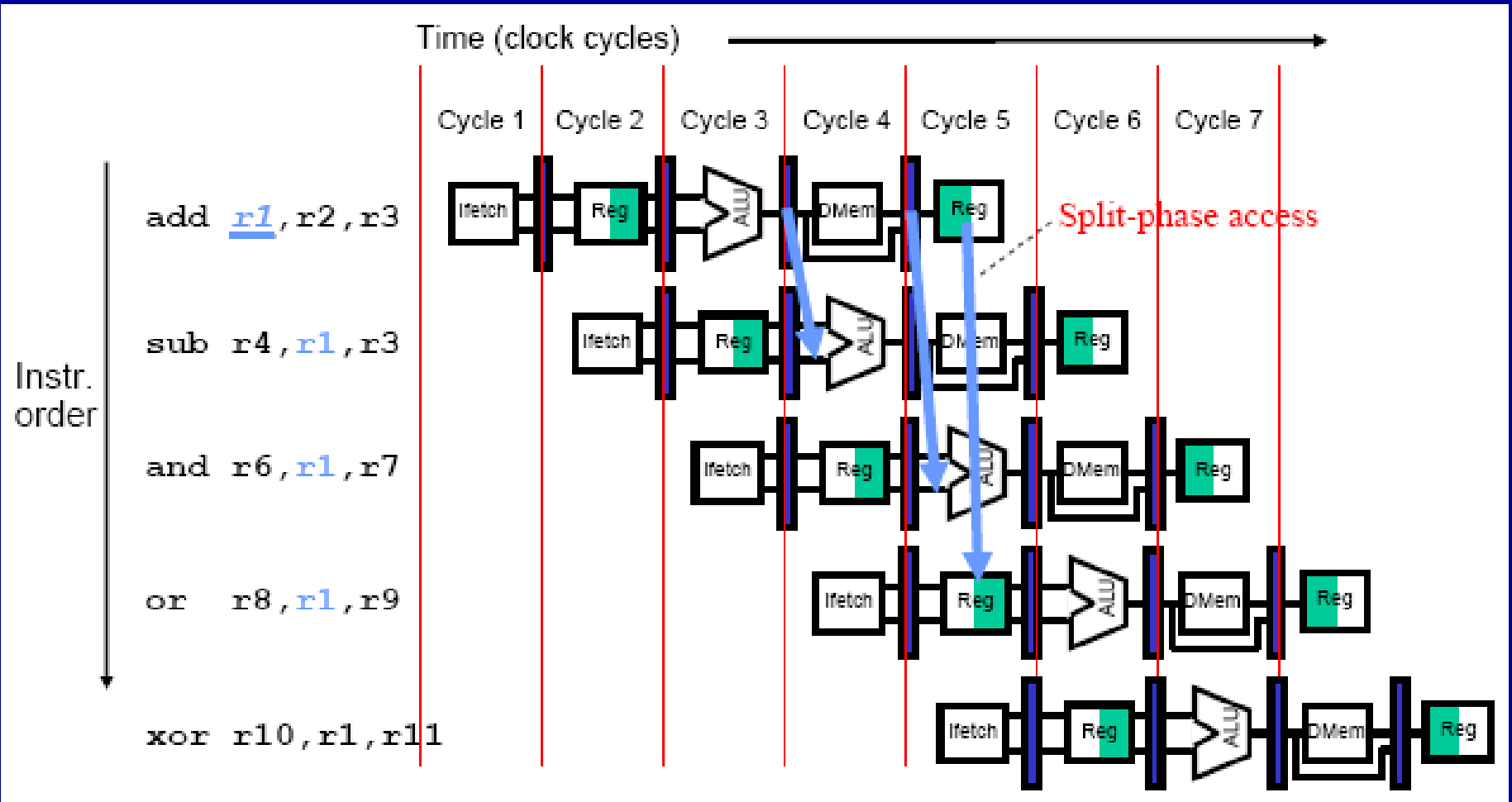
$\text{Regs}[\text{MEM}/\text{WB}.\text{IR}[\text{rt}]] \leftarrow$
 $\text{MEM}/\text{WB}.\text{LMD}$

Implementing Forwarding in HW

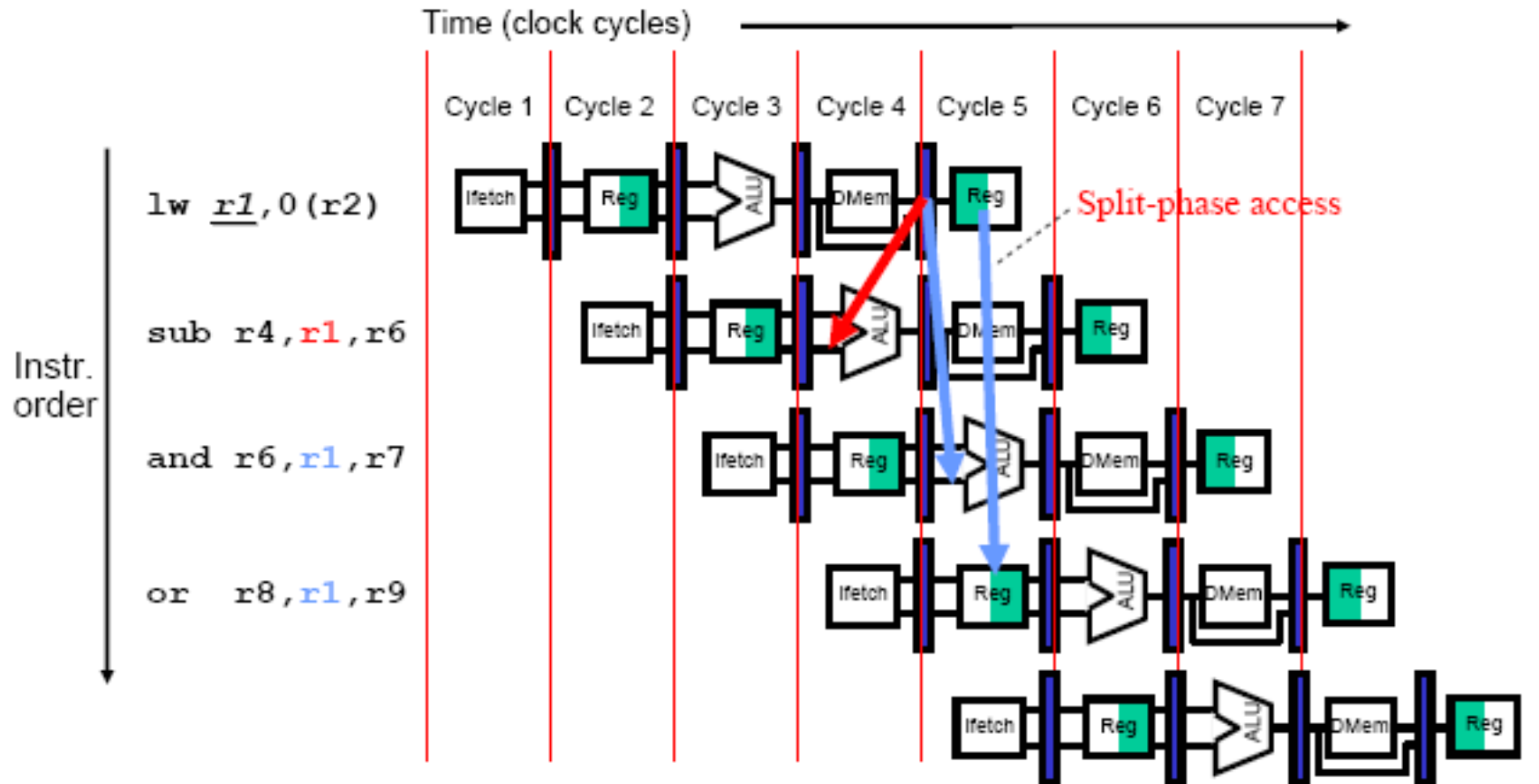


- To support data forwarding, additional hardware is required; **multiplexers** to allow data to be transferred back and **control logic** for the multiplexers

Avoidance of Data Hazards using Forwarding



An Unavoidable Stall



Basic compiler pipeline scheduling

- Idea – find **sequences of unrelated instructions** (no hazard) that can be overlapped in the pipeline to exploit ILP
 - A dependent instruction must be separated from the source instruction by a distance in clock cycles equal to **latency** of the source instruction to avoid **stall**
- A clever compiler can often reschedule instructions to avoid a stall (**instruction scheduling**)
 - A simple example:
 - Original code:
LW R2, 0(R4)
ADD R1, R2, R3 ← Stall happens here
LW R5, 4(R4)
 - Transformed code:
LW R2, 0(R4)
LW R5, 4(R4)
ADD R1, R2, R3 ← No stall needed

A Loop Program

➤ How the compiler can increase the amount of **ILP** by transforming loops?

➤ A simple example:

- Original code:

```
for (i = 1000; i>0; i = i-1)
```

```
    x[i] = x[i] + s
```

- MIPS code:

loop: L.D	F0,0(R1); F0 = array element
ADD.D	F4,F0,F2; add scalar in F2
S.D	F4,0(R1); store result
DADDUI	R1,R1,#-8; decrement pointer
BNE	R1,R2,loop

Execution of the Loop

➤ Executing the loop on MIPS pipeline without scheduling

- **loop: L.D F0,0(R1)**

stall

ADD.D F4,F0,F2

stall

stall

S.D F4,0(R1)

DADDUI R1,R1,#-8

stall

BNE R1,R2,loop

- **It requires 9 clock cycles**

Source Ins	User Ins	Latency
FP ALU Op	FP ALU Op	3
FP ALU Op	Store Double	2
Load Double	FP ALU Op	1
Load Double	Store Double	0

Execution with scheduling

➤ Executing the loop on MIPS pipeline with scheduling

- **loop:**

L.D	F0,0(R1)
DADDUI	R1,R1,#-8
ADD.D	F4,F0,F2
stall	
stall	
S.D	F4,8(R1)
BNE	R1,R2,loop

- It requires 7 clock cycles (gain of 2 cycles)
- To avoid a pipeline stall, a dependent instruction must be separated from the source instruction by a distance equal to the pipeline latency of the source instruction

Loop Unrolling

➤ Loop unrolling with **four copies** stalls

➤ loop:

Three branches and three decrements of R1 have been eliminated. The loop will run in 27 cycles, including 13 stalls. Gain of 9 cycles

Adjusted loop overhead instructions

L.D	F0,0(R1)	1
ADD.D	F4,F0,F2	2
S.D	F4,0(R1)	
L.D	F6,-8(R1)	1
ADD.D	F8,F6,F2	2
S.D	F8,-8(R1)	
L.D	F10,-16(R1)	1
ADD.D	F12,F10,F2	2
S.D	F12,-16(R1)	
L.D	F14,-24(R1)	1
ADD.D	F16,F14,F2	2
S.D	F16,-24(R1)	
DADDUI	R1,R1,#-32	1
BNE	R1,R2,loop	

Note the renamed registers

Note the adjustments for store and load offsets (only store highlighted red)!

Loop Unrolling with Scheduling

➤ loop:

L.D
L.D

F0,0(R1)
F6,-8(R1)

L.D F10,-16(R1)
L.D F14,-24(R1)
ADD.D F4,F0,F2
ADD.D F6,F6,F2
ADD.D F12,F10,F2
ADD.D F16,F14,F2

S.D F4,0(R1)
S.D F8,-8(R1)
DADDUI R1,R1,#-32
S.D F12,16(R1)
S.D F16,8(R1)
BNE R1,R2,loop

This loop will run in 14 clock cycles, **without any stalls**. It requires symbolic substitution and simplification. Gain of **22 cycles**

Loop Unrolling with Scheduling

- **Decisions and transformations taken:**
 - **Identify that loop iterations are independent**
 - **Use different registers to avoid unnecessary constraints**
 - **Eliminate the extra test and branch instructions and adjust the loop termination and iteration code**
 - **Determine the loads and stores that can be interchanged in the unrolled loop**
 - **Schedule the code, preserving any data dependences needed to yield the same result as the original code**
- **Key requirement is to understand how instructions depend on one another and how they can be changed and reordered**

Loop Unrolling with Scheduling

- Three different types of limits:
 - Decrease in the **amount of overhead** amortized with each unroll
 - The growth in code size due to loop unrolling (may increase **cache miss rates**)
 - Shortfall of registers created by aggressive unrolling and scheduling (**register pressure**)
- Loop unrolling improves the performance by eliminating overhead instructions
- Loop unrolling is a simple but useful method to increase the size of straight-line code fragments
- Sophisticated high-level transformations led to significant increase in complexity of the compilers

Dynamic Instruction Scheduling: The Need

- We have seen that primitive pipelined processors tried to overcome data dependence through:
 - Forwarding:
 - But, many data dependences can not be overcome this way
 - Interlocking: brings down pipeline efficiency
- Software based instruction restructuring:
 - Handicapped due to inability to detect many dependences

Dynamic Instruction Scheduling

- **Scheduling:** Ordering the execution of instructions in a program so as to improve performance
- **Dynamic Scheduling:**
 - The hardware determines the order in which instructions execute
 - This is in contrast to statically scheduled processor where the compiler determines the order of execution

Summary

- Crucial in modern microprocessors, which issue/execute multiple instructions every cycle
- **Pipeline Hazards** arises when ideal conditions are not met
- Discussed three possible types of pipeline hazards
 - **Structural Hazard** – can be overcome using additional hardware
 - **Data Hazards** – can be overcome using software (**compiler**) or additional hardware (**forwarding**)

Points to Remember

- What is pipelining?
 - It is an implementation technique where multiple tasks are performed in an **overlapped** manner
- When can it be implemented?
 - It can be implemented when a task can be divided into two or subtasks, which can be performed **independently**
 - The earliest use of parallelism in designing CPUs (since 1985) to enhance processing speed was **Pipelining**
- Pipelining does not reduce the execution time of a single instruction, it increases the **throughput**
- **CISC** processors are not suitable for pipelining because of:
 - Variable instruction format
 - Variable execution time
 - Complex addressing mode
- **RISC** processors are suitable for pipelining because of:
 - Fixed instruction format
 - Fixed execution time
 - Limited addressing modes