Instruction Pipeline : *An Introduction*

Appendix A - Computer Architecture : *A Quantitative Approach* - Hennessy & Patterson

Instruction Pipeline

- **Instruction Pipeline is a hardware mechanism by which** fetch and execution **of several instructions can overlap in time.**
- **It exploits the instruction level parallelism present in the stream of instructions of a program.**
- **It is transparent to the programmer.**

Instruction Pipeline

- **Fetch and execution of an instruction is subdivided into stages.**
- **Different hardware units perform operations of different stages.**
- **Every instruction goes through these stages starting from the first stage.**
- **Different instructions are active at different stages.**

An Example

- **Let the fetch-execution be divided into five (5) stages.**
- **If the i^{th} instruction is in the 5^{th} stage, then the $(i+1)^{th}$ instruction is in the 4^{th} stage, and the $(i+4)^{th}$ instruction is in the 1^{st} stage (without branch or jump).**
- **Each instruction will move one step (the i^{th} one will be over and $(i+5)^{th}$ one will enter the 1^{st} stage) at every processor cycle.**
- **Every stage should finish its operation within this period.**

Instruction Pipeline

- The **processor cycle** is determined by the **slowest stage**! A designer tries to **balance different stages**.
- **Throughput** of a pipeline is the **number of instructions** completed per **unit time**.
- In a **perfectly balanced ideal pipeline**, the **execution time** per instruction is

$$\frac{\text{Time per instruction on nonpipelined CPU}}{\text{Number of pipe stages}}$$

Instruction Pipeline

- **Under the ideal condition the speedup is the number of stages in the pipeline.**
- **But in a real pipeline often the stages are not balenced, and**
- **The transfer of partially completed instruction from one stage of the pipeline to another takes some time.**
- **The actual speedup is less than the ideal.**

Clock Cycles per Instruction (CPI)

- **Multiple clock cycles per instruction in a nonpipelined CPU.**
- **Reduction in clock cycles per instruction (CPI) in a pipelined implementation.**
- **Single clock cycle per instruction in a nonpipelined CPU.**
- **Reduction in the clock cycle time (high frequency) in a pipelened implementation.**

RISC and Instruction Pipeline

- RISC instruction set is more suitable for the pipelined implementation.
- Pentium processors uses a RISC instruction set internally to support its original instruction set externally!
- Instructions should be simple and without side-effect.

Basic RISC

- **All instructions are of same length and only a few instruction formats.**
- **All ALU operations are performed on registers.**
- **The memory is accessed only to load a register and store the content of a register. The data size may be of 8, 16, 32 or 64 bits.**
- **There are essentially three types of instructions for basic processing - ALU, Load/Store and Branch/Jump.**
- **MIPS instruction set is an example.**

MIPS64 Registers

- **MIPS64 has 32 64-bit general-purpose registers (GPRs). They are named as $R0, R1, \cdots, R31$.**
- **32 floating-point registers (FPRs) each can hold 32 single or double-precision values.**
- **The value of $R0$ is always zero (0).**

ALU Instruction

- **The operation, one or two operands and the destination are specified in the instruction.**
- **One operand may be a sign-extended immediate data (16-bit size in MIPS).**
- **Operands (except immediate) come from registers and the destination is also a registers.**

Load/Store Instruction

- **Two registers are specified in an instruction.**
- **The first register is the destination (source) of the load (store) instruction.**
- **The memory address is specified by the second (base) register and a displacement or offset (16-bits in MIPS) is specified.**

Load/Store Instruction

- The effective memory address is the sum of the content of the base register and the offset (sign-extended).
- LD loads a 64-bit data in the register.

Branch/Jump Instruction

- Conditional branches are of two types - branch on condition codes or branch on the content of a register. MIPS uses the second type.
- The branch address is calculated relative to the PC. A signed offset is specified in the instruction.

A Simple nonpipelined Implementation

- Each instruction takes at most five (5) clock cycles.
- The implementation is not very efficient, but can easily be modified to a pipelined implementation.
- We consider a subset of integer instructions.

Five Phases of Instruction Execution (IF)

- **Instruction Fetch (IF): opeartions done in parallel -**
  - **The PC is used to fetch the current instruction.**
  - **PC is incremented by 4 (an instruction size is 4B) to point the next instruction.**

Five Phases of Instruction Execution (ID)

- **Instruction Decode and Register Read (ID): operations done in parallel -**
  - **The fetched instruction is decoded.**
  - **The register address fields (two - for read) of the instruction are used to read the register file and the equality test is performed, (for a banch instruction).**
  - **The branch target address is computed by adding the sign-extended offset to the new value of PC. A branch can be completed at this stage.**

Five Phases of Instruction Execution (EX)

- Executation and Effective Address (EX): any one of the following operations are performed.
  - The memory address of the operand (in case of load/store instruction) is computed by adding the value of the base register and the sign-extended offset.
  - The ALU operation is performed on two register operands.
  - ALU operation is performed on a register operand and a sign-extended immediate data.

Five Phases of Instruction Execution (MM)

- **Memory Access (MM): any one of the following operations are performed.**
  - **The memory is read using the effective address for a load instruction.**
  - **The memory is written from a register already read (in stage ID) in case of a store instruction. The store is over at this stage.**

Five Phases of Instruction Execution (WB)

- Write Back (WB): any one of the following operations are performed.
  - The result of ALU operation is written into the register.
  - The register is loaded at the end of load instruction.

Five Stage Pipeline

IF	ID	EX	MM	WB

- **IF: instruction fetch.**
- **ID: instruction decode, register fetch and branch address computation.**
- **EX: ALU operation, effective address computation.**
- **MM: Memory access for load or store.**
- **WB: register write-back or load register.**

Five Stage Pipeline

Inst.	Clock 1	2	3	4	5	6	7	8	
i	IF	ID	EX	MM	WB				
$i+1$		IF	ID	EX	MM	WB			
$i+2$			IF	ID	EX	MM	WB		
$i+3$				IF	ID	EX	MM	WB	
$i+4$					IF	ID	EX	MM	WB

More Data Path Resource

- **Each of the five stages are active simultaneously for different instructions.**
- **Data path resources cannot be shared for different phases.**

More Data Path Resources

- **Simultaneous memory access for instruction fetch (IF) and data fetch (MM).**
- **Separate instruction and data memory. Five times memory bandwidth.**

More Data Path Resources

- **Simultaneous use of adder (ALU) to increment the PC (IF), add operands (EX) and branch address computation (ID).**
- **One adder to increment PC, another adder for branch target computation and the standard ALU for operations.**
- **PC is incremented at every IF stage to fetch the next instruction in the next clock.**

More Data Path Resources

- **Register file access for data read (two registers) and also for data write. Read and write may be on the same register.**
- **Write in the first-half of the clock and read in the second-half of the clock.**

More Data Path Resources

- Five instructions are active simultaneously.
- Part of the instruction and partially computed data are to be saved for every instruction. Also the interference is to be avoided.
- Pipeline registers are used between stages.

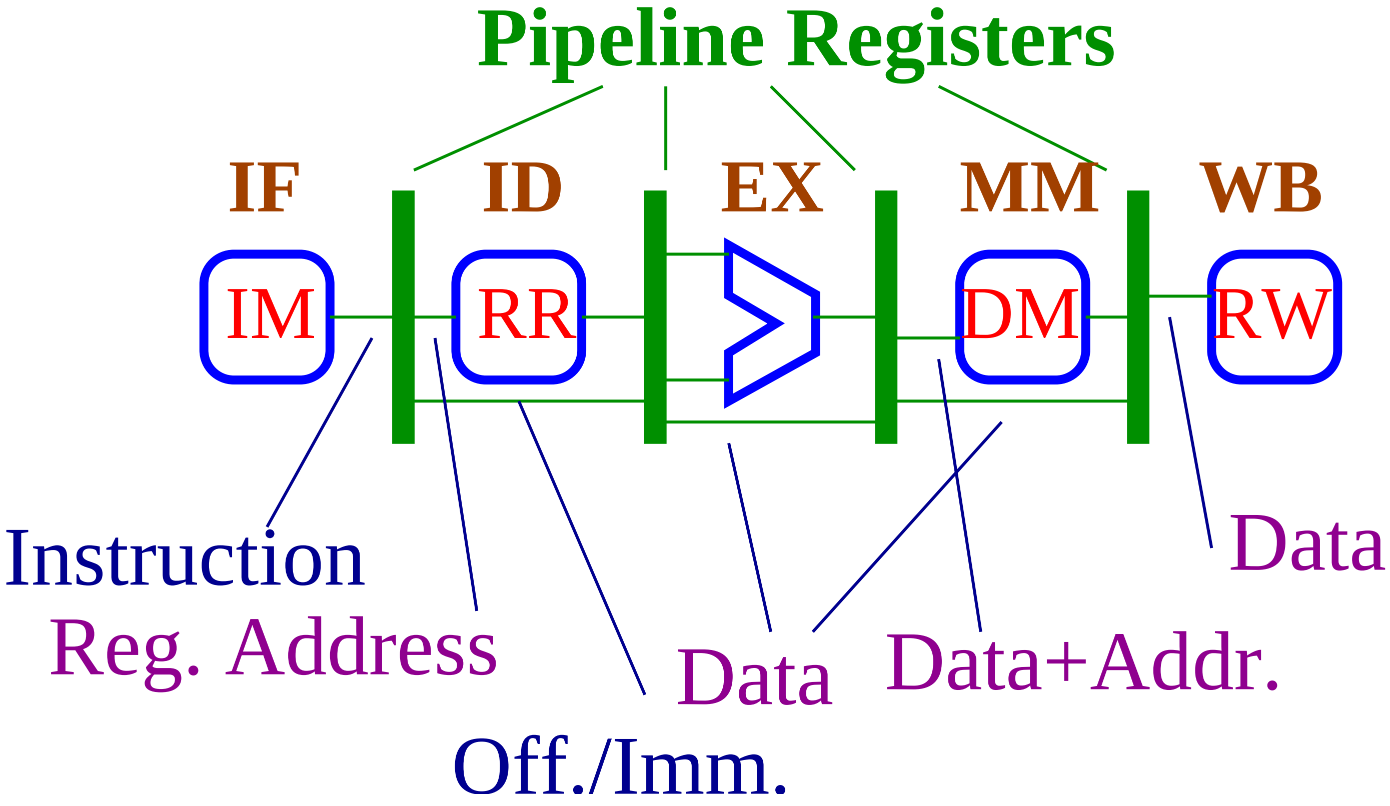

Figure 1: **Pipeline Stages**

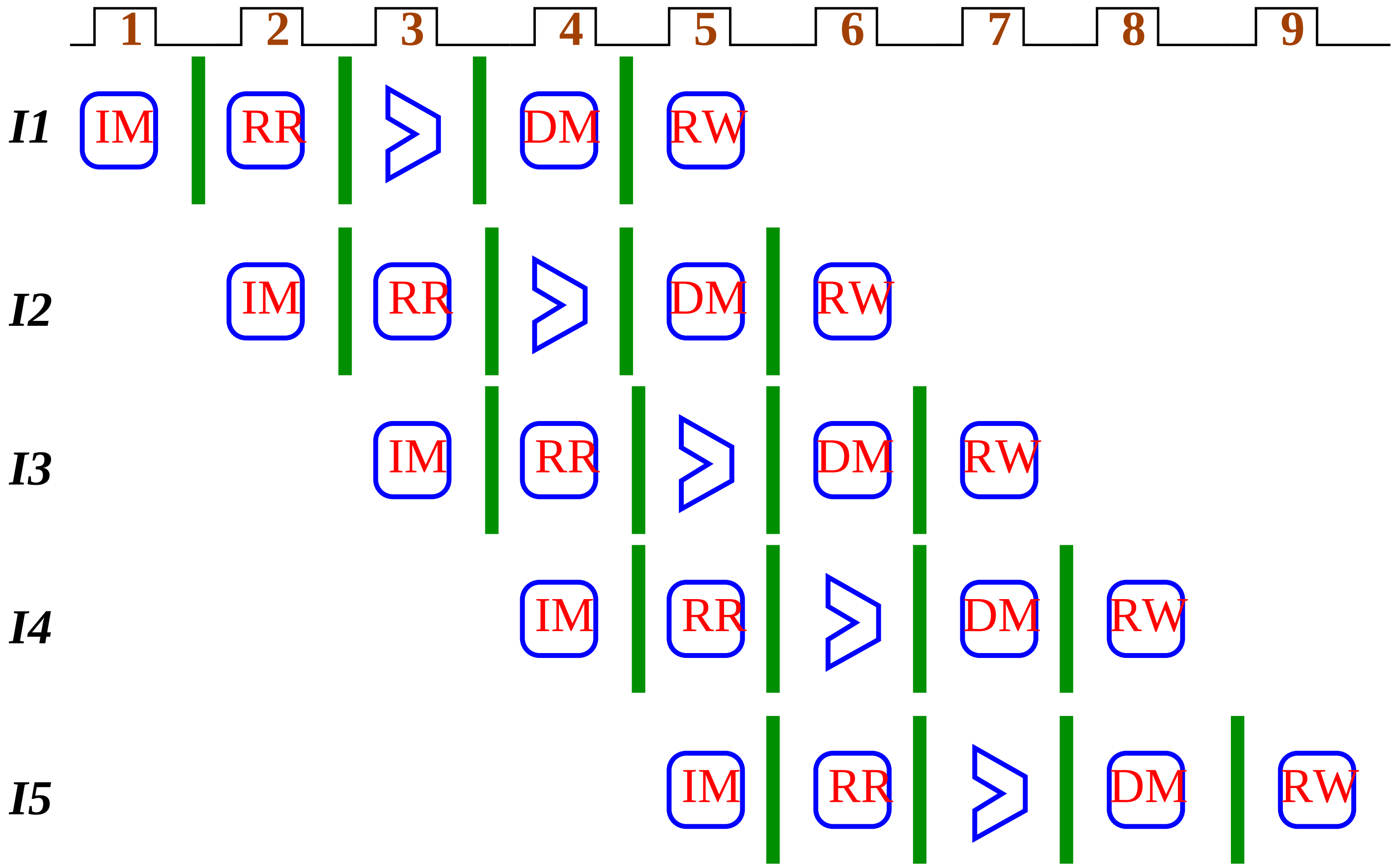

Figure 2: **Pipeline Stages**

Use of Pipeline Registers: Examples

- **Immediate date is a part of the instruction. It is used in EX phase and is to be passed through the pipeline registers.**
- **The value read from the register-file at RR stage can be stored in the memory only in the DM phase.**

Pipeline Registers

IF/ID	**between IF and ID**
ID/EX	**between ID and EX**
EX/MM	**between EX and MM**
MM/WB	**between MM and WB**

Pipeline Performance

- A **pipelined CPU** executes **more instructions per unit time** resulting a faster execution of a program.
- It **does not reduces** the **execution time** of an **individual instruction** (in fact it gets slightly increased due to presence of the pipeline registers etc.).
- The **slowest pipe stage** dictates the **clock speed** of the CPU. **Clock skew** and **register setteling time** also puts a limit on clock speed.

Pipeline Hazards

- **There would not have been any problem in a integer pipeline (what about multipliction?) if all instructions are independent.**
- **But due to data dependency, change in the flow of control and conflicting resource requirements, prevents the physically next instruction to be executed.**

Pipeline Hazards

- **Structural Hazards**: The **processor cannot execute two consecutive instructions** due to **resource conflict**.
- **Data Hazards**: **Execution of two instructions cannot overlap** because the **computation of the second instruction** depends on the **result of the first**, which is not yet complete.
- **Control Hazard**: This is due to **sudden change in the execution flow**.

Pipeline Stall

- Hazard can be avoided if the execution of some instructions are stalled.
- In our simple model of the pipeline, all instructions following a stalled instruction are also stalled and instructions fetched earlier are allowed to proceed.
- No new instruction is fetched as long as the pipeline is stalled.

Pipeline Performance witout Stall: an Example

The clock in a processor without pipeline is 1ns. The ALU (40%) and branch (20%) operation takes 4 cycles and the memory access (40%) takes 5 cycles.

Due to overhead in the pipeline, the clock in the pipelined processor is 1.2ns. Calculate the speedup (CPI is 1).

Pipeline Performance: an Example

- **Average instruction execution time (without pipeline):** $1 \times ((0.4 + 0.2) \times 4 + 0.4 \times 5) = 4.4nS$.
- **Instruction execution time in pipeline: 1.2nS.**
- **Speedup in pipeline:** $\frac{4.4}{1.2} = 3.67$.

Pipeline Performance

speedup

$$= \frac{\textbf{Average Instruction Execution Time - unpipelined}}{\textbf{Average instruction Execution Time - pipelined}}$$

$$= \frac{\textbf{CPI} \times \textbf{Clock Cycle - unpipelined}}{\textbf{CPI} \times \textbf{Clock Cycle - pipelined}}$$

Pipeline Performance with Stall

Let the CPI be one (1) in a ideal pipeline; and the average loss of clock cycles per instruction for the pipe-stall is τ_s. Then the CPI of a pipeline with stall is $1 + \tau_s$.

Ignoring the increase in clock cycle time in a pipelined processor we get the following speedup.

Pipeline Performance with Stall

$$\text{speedup} = \frac{\textbf{CPI - unpipelined}}{1+\tau_s} = \frac{\textbf{Pipeline depth}}{1+\tau_s},$$

We assume that each instruction takes the same number of clock cycles and it is equal to the depth of the pipeline (number of stages).

Pipeline Performance with Stall

In a different view (we may take CPI for both unpipelined and pipelined processor to be one(1) and the pipeline improves the clock cycle. The speedup is -

Pipeline Performance with Stall

$$\text{speedup} = \frac{1}{1+\tau_s} \times \frac{\textbf{Clock cycle - unpipelined}}{\textbf{Clock cycle - pipelined}}$$

$$= \frac{1}{1+\tau_s} \times \textbf{Pipeline Depth}$$

Execution is divided into pipe stages and a faster clock can be used. If the stages are balanced,

$$\textbf{Pipeline Depth} = \frac{\textbf{Clock cycle - unpipelined}}{\textbf{Clock cycle - pipelined}}$$

Structural Hazards

- If one stage of a pipeline shares resources from another stage, all combinations of instructions may not be possible to execute in parallel.
- If there is only one single port memory, simultaneous instruction fetch and data access is not possible.
- As a solution the memory is divided into data and instruction memory (at the level of L1 cache) or there may be an instruction buffer.

Pipeline Bubble

- **A structural conflict stalls one of the instructions as long as the resource is not available. This will increase avareage CPI.**
- **A stall is often called a pipeline bubble and is indicated by a simple diagram.**

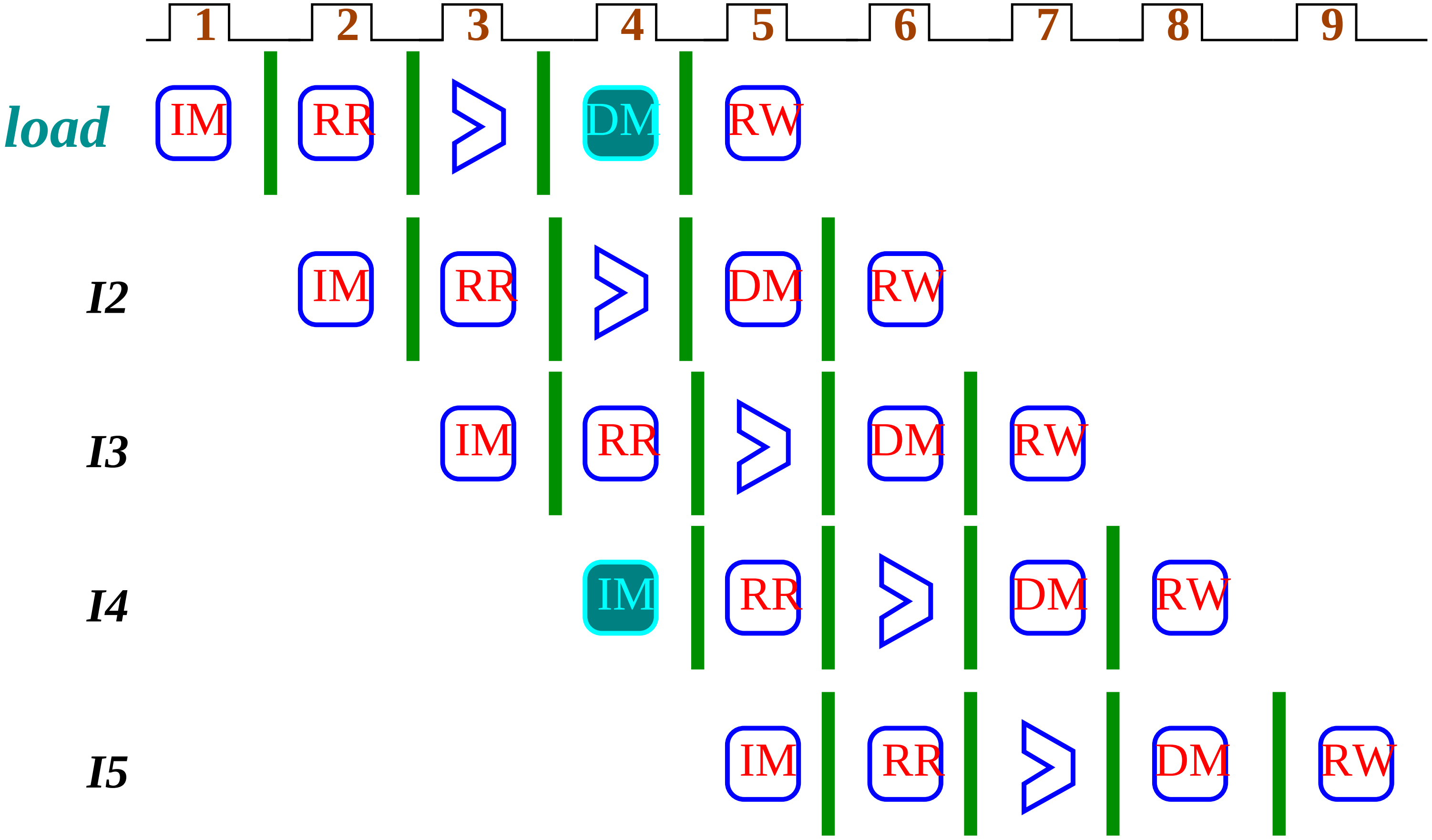

Figure 3: **Structural Hazard - Single-port Memory**

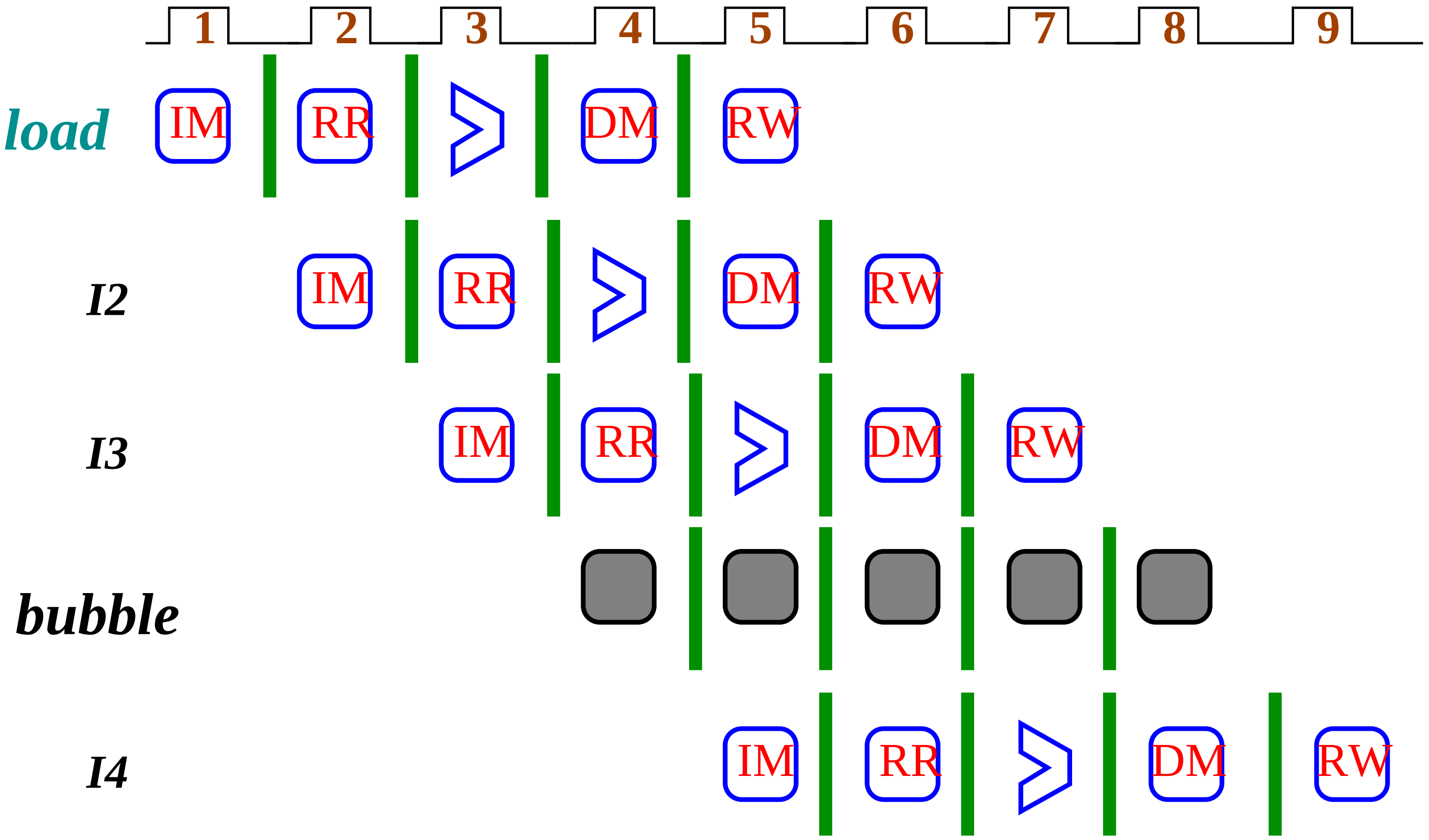

Figure 4: **Pipeline Bubble**

Pipeline Bubble

	Clock								
Inst.	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	
load	**IF**	**ID**	**EX**	**MM**	**WB**				
$i+1$		**IF**	**ID**	**EX**	**MM**	**WB**			
$i+2$			**IF**	**ID**	**EX**	**MM**	**WB**		
$i+3$				**stall**	**IF**	**ID**	**EX**	**MM**	**WB**

Performance: an Example

- **Let the data reference is 40% of the instructions and every load or store increases the average CPI which in ideal case is 1.**
- **The CPI with memory reference hazard is 1.4.**
- **But then avoidance of structural hazards is costly e.g. more bandwidth or more number of ports in the memory etc.**

Data Hazards

- **In a normal sequential execution if there is read after a write.**
- **In a pipeline implementation the same sequence of instructions may try to read the data before the write and give rise to data hazard.**

Data Hazards

```
DADD  R1, R2, R3   # R1 = R2 + R3
DSUB  R4, R1, R5   # R4 = R1 - R5
AND   R6, R1, R7   # R6 = R1 & R7
OR    R8, R1, R9   # R8 = R1 | R9
XOR   R10,R1, R11  # R10 = R1 e| R11
```

All instructions after DADD uses the value written in R1.

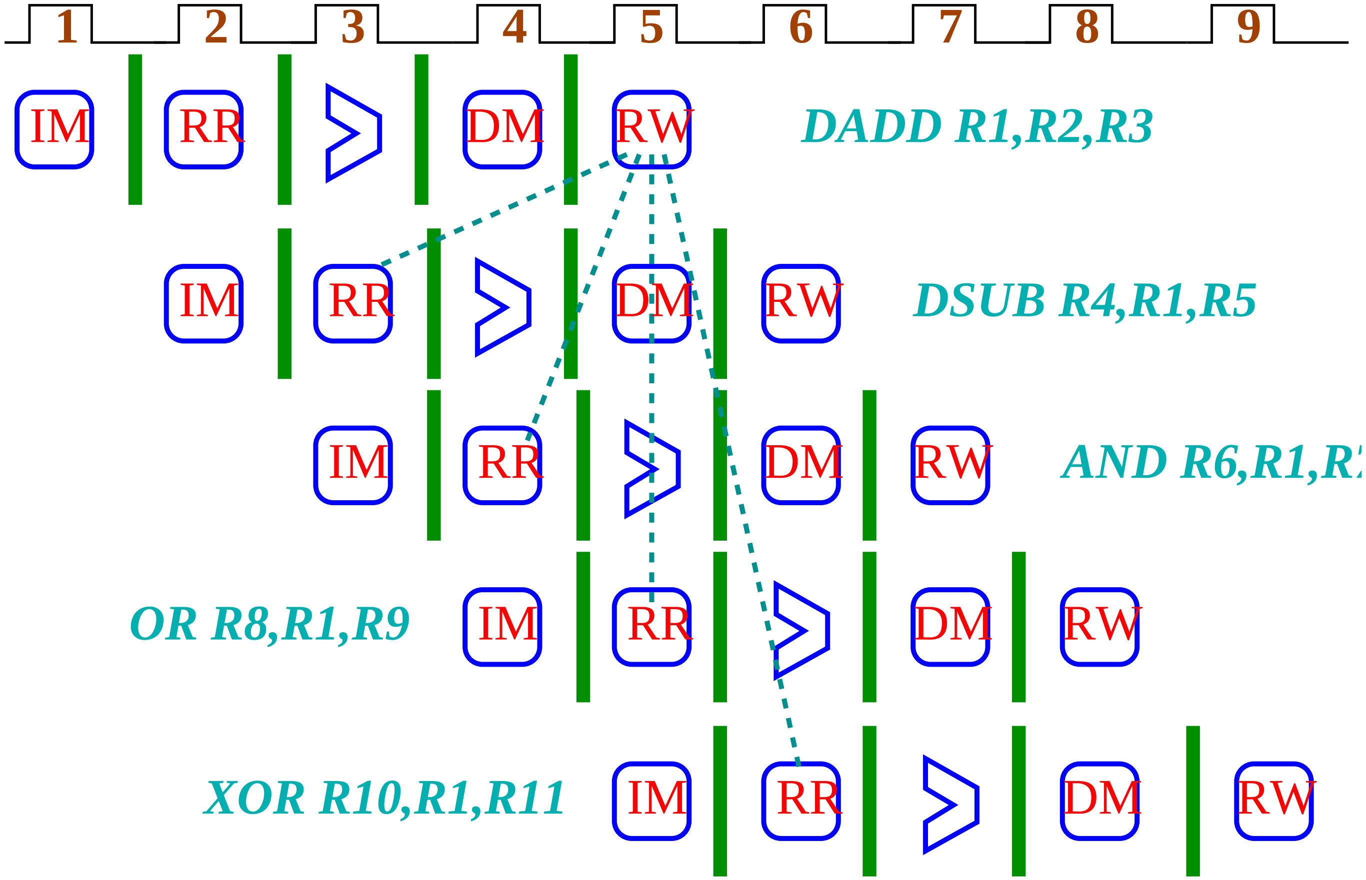

Figure 5: **Read before Write: Data Hazard**

Data Hazard

- Unless precaution is taken, the instruction DSUB and AND will take the old content of the register R1.
- The instruction OR reads the register R1 in the same clock when it is written. The register write is done in the first half and the read in the second half of the clock to avoid any race condition.
- There is no problem with the XOR instruction.

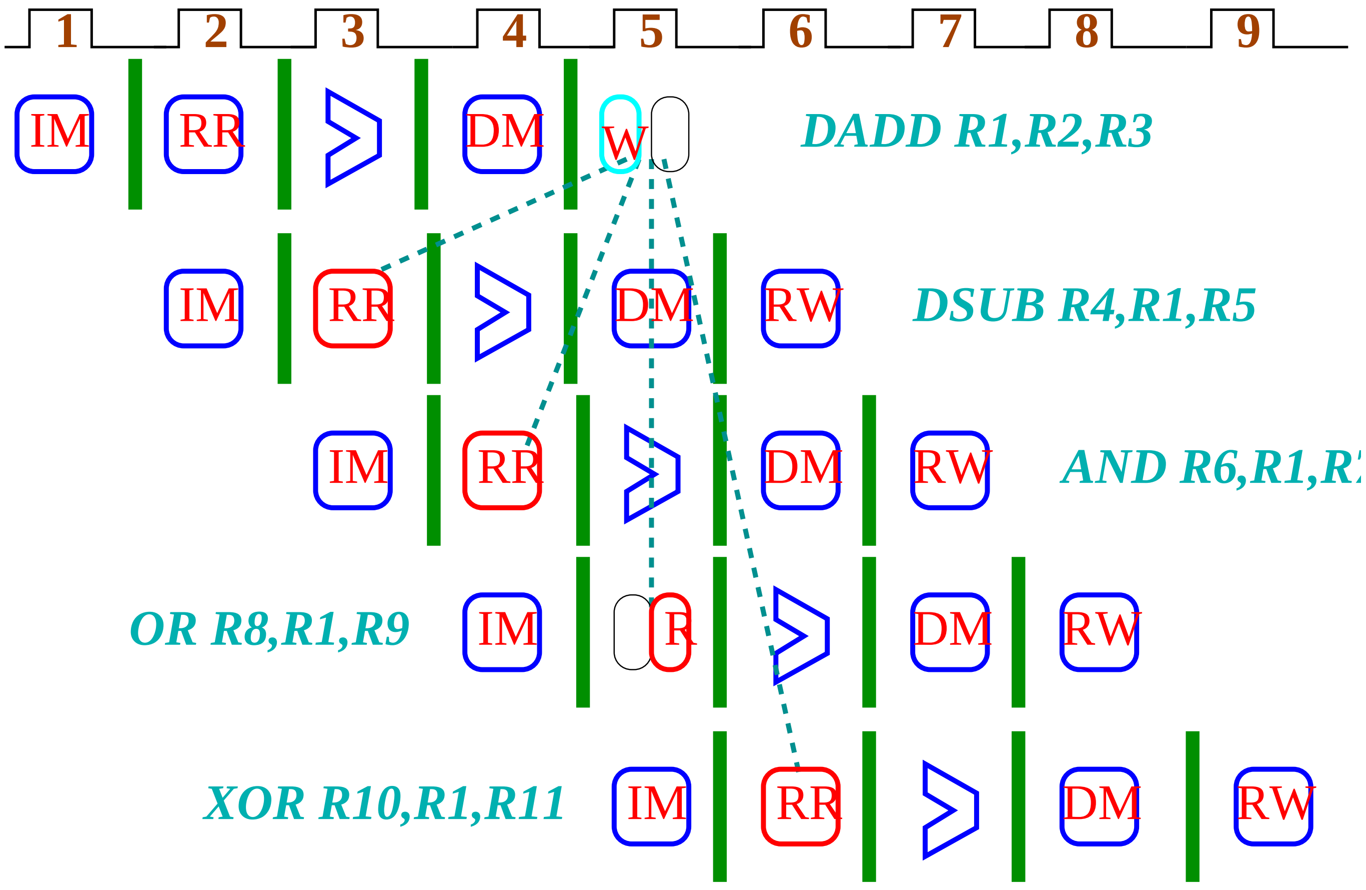

Figure 6: **Write then Read**

Result Forwarding

- **The actual data required by DSUB and AND instructions is ready (from the ALU) before its use but not written in the register R1.**
- **The data may be directly forwarded (short circuit) to the ALU through the pipeline registers - EX/MM and MM/WB.**

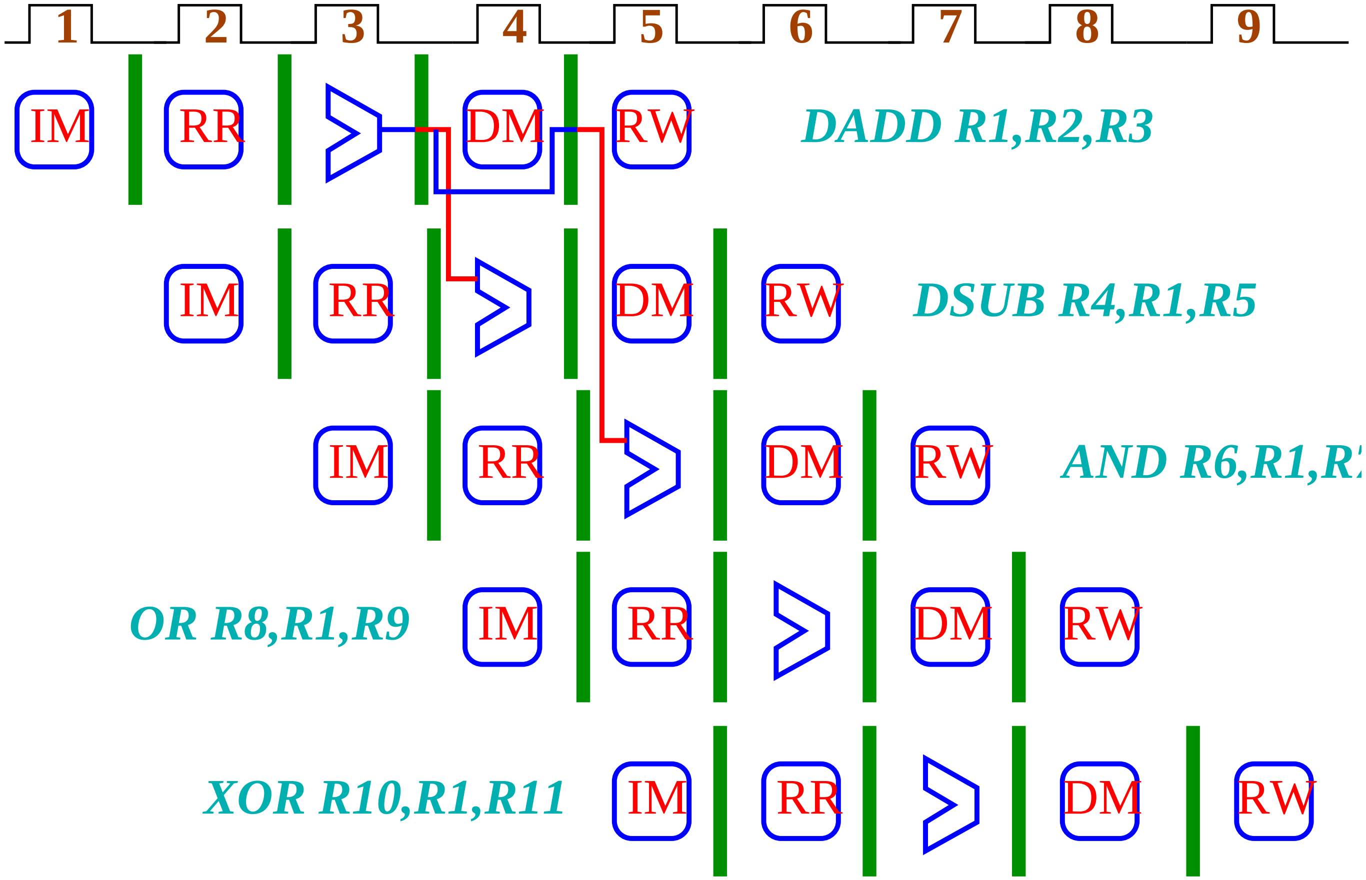

Figure 7: **Data Short Circuit**

Result Forwarding

- **The result of R2 + R3 will be written in a portion of EX/MM and MM/WB registers as the final write to the register R1 takes place in the stage WB.**
- **If there is a dependency, the hardware will forward the data to the appropriate input of the ALU.**
- **Forwarding to different ALU input: DSUB R4, R1, R5 or DSUB R4, R5, R1.**

Result Forwarding

- **If there is no data forwarding in the hardware, the execution will be stalled.**

```
DADD  R1, R2, R3  # R1 = R2 + R3
DSUB  R4, R1, R5  # R4 = R1 - R5
AND   R6, R1, R7  # R6 = R1 & R7
OR    R8, R1, R9  # R8 = R1 | R9
XOR   R10,R1, R11 # R10 = R1 e| R11
```

Stall without Short Circuit

	Clock							
Inst.	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
DADD	**IF**	**ID**	**EX**	**MM**	**WB**			
DSUB		**IF**	**stall**	**stall**	**ID**	**EX**	**MM**	**WB**
AND					**IF**	**ID**	**EX**	**MM**
OR						**IF**	**ID**	**EX**
XOR							**IF**	**ID**

Forwarding

```
DADD  R1, R2, R3    # R1 = R2 + R3
LD    R4, 0(R1)     # R4 = Mem[R1 + 0]
SD    R4, 12(R1)    # Mem[R1+12] = R4
```

- **The ALU output (EX/MM) is forwarded to ALU input (EX).**
- **The same ALU output is also forwarded from (MM/WB) to the ALU input.**
- **The memory output (MM/WB) is also forwarded to the memory input.**

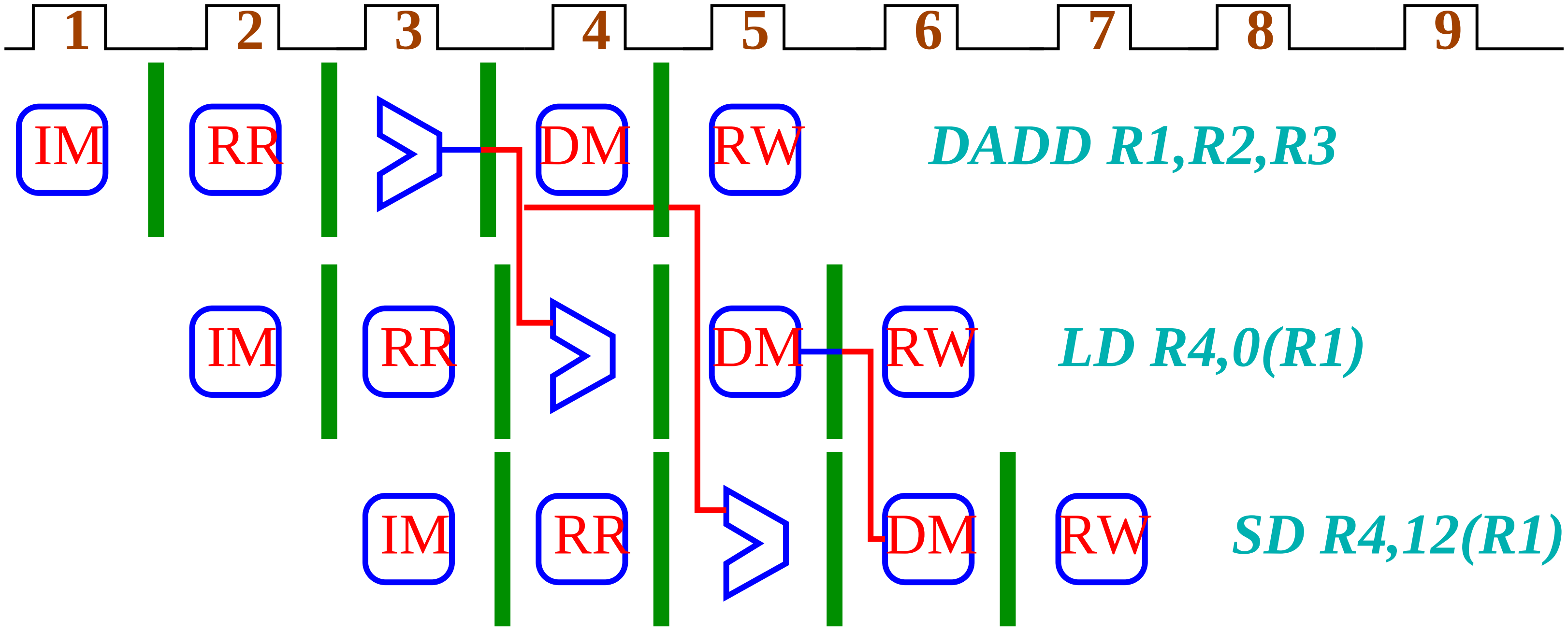

Figure 8: **Data Short Circuit**

All Stall cannot be Avoided

```
LD   R1,0(R2)
DSUB R4,R1,R5
AND  R6,R1,R7
OR   R8,R1,R9
```

- The memory data is available only at the end of MM phase and cannot be forwarded to the EX phase of DSUB which is earlier.

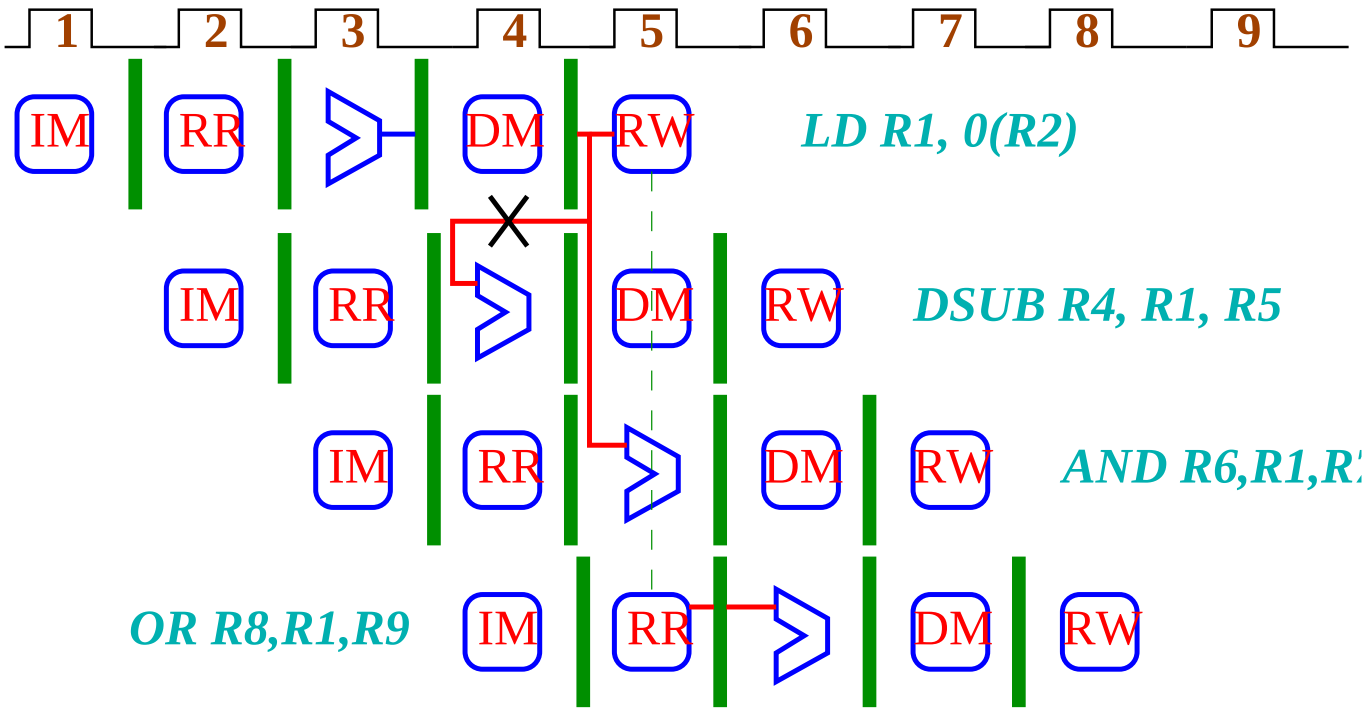

Figure 9: **Stall!**

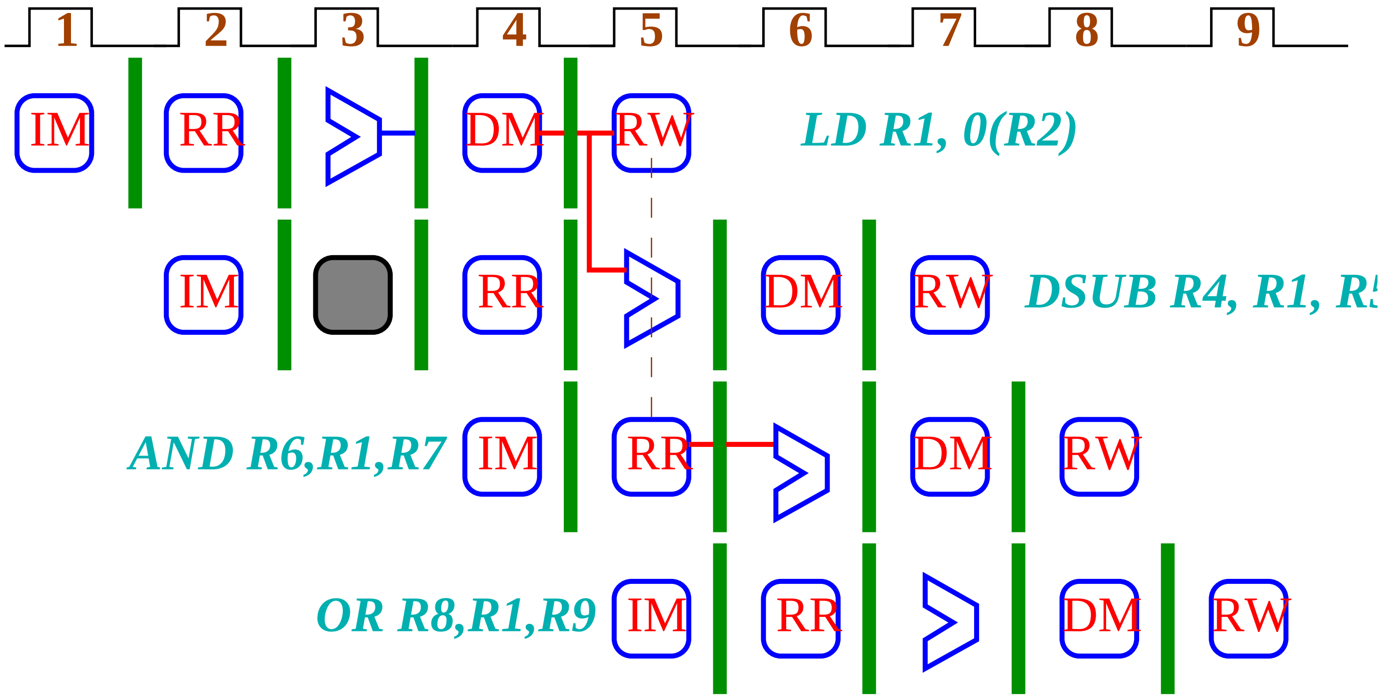

Figure 10: **Stall!**

Stall cannot be Avoided

Inst.	Clock							
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
LD	**IF**	**ID**	**EX**	**MM**	**WB**			
DSUB		**IF**	**stall**	**ID**	**EX**	**MM**	**WB**	
AND				**IF**	**ID**	**EX**	**MM**	**WB**
OR					**IF**	**ID**	**EX**	**MM**
XOR						**IF**	**ID**	**EX**

Control Hazard

- **A branch may be taken (the branch condition is satisfied and the PC is loaded with target address specified), or it may be not taken (the PC is incremented by 4).**
- **The branch taken or not taken can be detected in MIPS in the ID phase. By this time the physically next instruction will be fetched. But it may be necessary in case of a taken branch to load the instruction from the target.**

Branch Stall

Inst.	Clock 1	2	3	4	5	6	7	8
Branch	IF	ID	EX	MM	WB			
Br. Succ.		IF	IF	ID	EX	MM	WB	
Br. Succ.+1				IF	ID	EX	MM	WB
Br. Succ.+2					IF	ID	EX	MM
Br. Succ.+3						IF	ID	EX

Freeze/Flush the Pipeline

- **Hold execution of next instruction until the branch destination is known.**
- **Simple hardware and software.**

Hardware Assumes: Branch not Taken

- **Hardware continues with the next instruction.**
- **If the branch is taken, the fetched instruction is turned to no-op and the fetch is restarted from the target.**
- **It is important to take care of processor state change (does not happen at this stage in MIPS).**

Branch not Taken

Inst.	Clock							
	1	2	3	4	5	6	7	8
Branch(i)	IF	ID	EX	MM	WB			
i+1		IF	ID	EX	MM	WB		
i+2			IF	ID	EX	MM	WB	
i+3				IF	ID	EX	MM	WB
i+4					IF	ID	EX	MM

Branch Taken

Inst.	Clock 1	2	3	4	5	6	7	8
Branch(i)	IF	ID	EX	MM	WB			
i+1		IF	nop	nop	nop	nop		
Br.Tgt.			IF	ID	EX	MM	WB	
Br.Tgt+1				IF	ID	EX	MM	WB
Br.Tgt+2					IF	ID	EX	MM

Predicted Taken

- The hardware assumes that the branch will be taken. Once the branch target address is computed, it fetches instruction from that address.
- In MIPS this scheme does not have any advantage as both the target address and the branch outcome are known at the ID stage.
- This method may be useful for an implementation where the target address is known before the branch outcome.

Roll of a Compiler

- **The simplest solution for a compiler is to insert a no-op.**
- **But if the branch pattern of a code and the prediction scheme of a hardware is known, the compiler can generate better code.**

An Example: C Code

```
int main() {
 int n, i, fact = 1 ;
 scanf("%d", &n) ;
 for(i=1; i<=n; ++i) fact = fact * i ;
 printf("%d! = %d\n", n, fact) ;
}
```

On DEC Alpha

```
    cmplt   $3, 1, $4    # if(n<1)$4=1; else $4=0
    bne     $4, $34      # if($4!=0) goto $34
$32:
    ldl     $5, 24($sp)
    ldl     $6, 32($sp)
    mull    $5, $6, $7
     ...............
    ldl     $23, 40($sp)  # $23 = n
    cmple   $22, $23, $24 # if(i<=n)$24=1; else $24=0
    bne     $24, $32      # if($24!=0) goto $32
$34:
```

On GCC on Intel x86

```
.L2:
   movl    -8(%ebp), %eax   # eax = i
   cmpl    -4(%ebp), %eax   #
   jle     .L5              # if(i<=n) goto .L5
   jmp     .L3              # else goto .L3
.L5:
    movl    -12(%ebp), %eax
    imull   -8(%ebp), %eax
    movl    %eax, -12(%ebp)
    leal    -8(%ebp), %eax
    incl    (%eax)
    jmp     .L2
```

.L3:

Delayed Branch

- **Used in RISC processors.**
- **Works well in five-stage pipeline.**
- **Following is the execution cycle with a branch delay one(1).**

Branch Instruction
Sequentially next instruction
Branch target if taken

The sequentially next instruction is in the branch delay slot.

Branch Delay Slot

- **The instruction in the branch delay slot is executed irrespective of the branch taken or not taken.**

Branch not Taken

Inst.	Clock 1	2	3	4	5	6	7	8
Branch(i)	IF	ID	EX	MM	WB			
BDI(i+1)		IF	ID	EX	MM	WB		
i+2			IF	ID	EX	MM	WB	
i+3				IF	ID	EX	MM	WB
i+4					IF	ID	EX	MM

Branch Taken

Inst.	Clock 1	2	3	4	5	6	7	8
Branch(i)	IF	ID	EX	MM	WB			
BDI(i+1)		IF	ID	EX	MM	WB		
Target			IF	ID	EX	MM	WB	
T+1				IF	ID	EX	MM	WB
T+2					IF	ID	EX	MM

Job of a Compiler

- Compiler should choose a successor instruction that is valid and also useful.
- It is difficult to predict the branch at the compile time.
- Processor introduces cancelling or nullifying branch instruction where the predicted branch direction is specified.
- If the branch is as predicted, the delay slot instruction is normally executed, otherwise it is turned no-op.

Scheduling the Delay Slot

DADD R1, R2, R3 **BEQ R2, .L1** **Delay Slot** ... **.L1:**	$\Rightarrow$	**BEQ R2, .L1** **DADD R1, R2, R3** ... **.L1:**

The instruction DADD R1, R2, R3 has to be executed irrespective of the direction of control flow. There will be problem if there is a label in the branch instruction.

Scheduling the Delay Slot

Before		After
.L1:		DSUB R4,R5,R6
DSUB R4,R5,R6		.L1:
. . .	$\Rightarrow$	. . .
DADD R1, R2, R3		DADD R1, R2, R3
BEQ R1, .L1		BEQ R1, .L1
Delay Slot		DSUB R4,R5,R6

- This is good if most often the branch is taken (loop).
- What if the branch is not taken?

Scheduling the Delay Slot

DADD R1, R2, R3 BEQ R1, .L1 Delay Slot . . . OR R7, R8, R9 . . . .L1: DSUB R4,R5,R6	$\Rightarrow$	DADD R1, R2, R3 BEQ R1, .L1 OR R7, R8, R9 . . . .L1: DSUB R4,R5,R6

Scheduling the Delay Slot

- **Value of R7 will not be used if the branch is taken.**
- **There is no use of R7 in the original code from the branch to the OR.**

Pipeline Performance with Branch

$$\text{speedup} = \frac{\textbf{Pipeline Depth}}{1+\tau_b},$$

where τ_b = pipeline stall cycle for branch, which is branch frequency $\times$ branch penelty.

$$\text{speedup} = \frac{\textbf{Pipeline Depth}}{1+\textbf{ branch frequency} \times \textbf{branch penelty}},$$