Pipelining Hazards

Structural, Data and Control Hazards

Pipeline Hazards

- There are situations, called hazards, that prevent the next instruction in the instruction stream from being executing during its designated clock cycle. Hazards reduce the performance from the ideal speedup gained by pipelining. There are three classes of hazards: –
  - Structural Hazards: They arise from resource conflicts when the hardware cannot support all possible combinations of instructions in simultaneous overlapped execution.
  - Data Hazards: They arise when an instruction depends on the result of a previous instruction in a way that is exposed by the overlapping of instructions in the pipeline.
  - Control Hazards: They arise from the pipelining of branches and other instructions that change the PC

➢ Structural Hazards

- ❖ When a machine is pipelined, the overlapped execution of instructions requires pipelining of functional units and duplication of resources to allow all possible combinations of instructions in the pipeline.
- ❖ If some combination of instructions cannot be accommodated because of a *resource conflict*, the machine is said to have a **Structural Hazards**

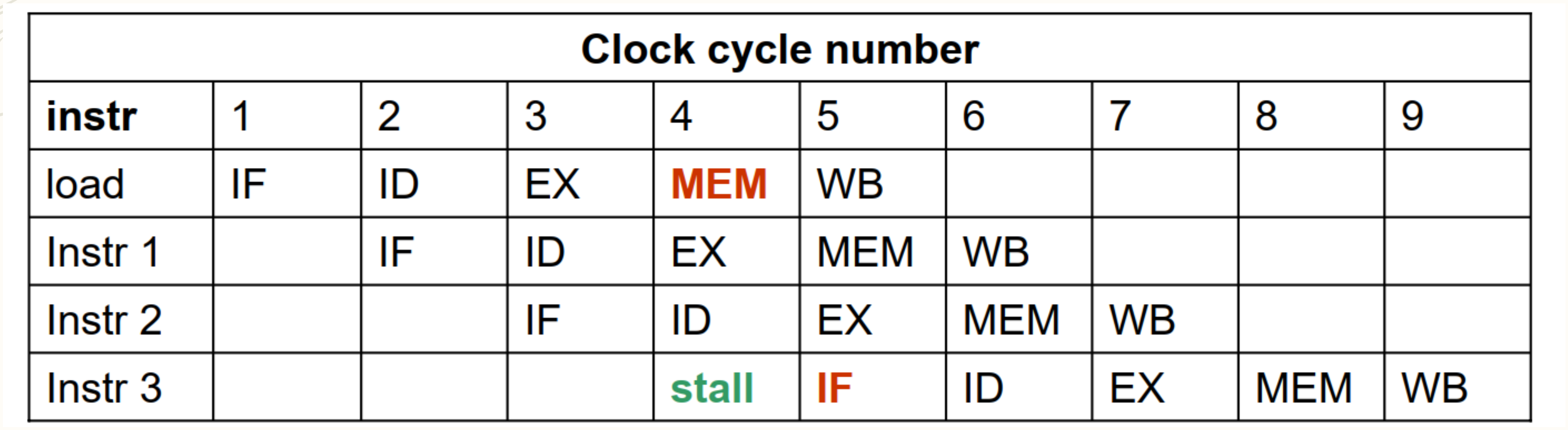
Example

- A machine has shared a **single-memory** pipeline for data and instructions. As a result, when an instruction contains a data-memory reference (load), it will conflict with the instruction reference for a later instruction (instr 3)

Clock cycle number								
instr	1	2	3	4	5	6	7	8
load	IF	ID	EX	**MEM**	WB			
Instr 1		IF	ID	EX	MEM	WB		
Instr 2			IF	ID	EX	MEM	WB	
Instr 3				**IF**	ID	EX	MEM	WB

Solution(1/2)

❖ To resolve this, we *stall* the pipeline for one clock cycle when a data-memory access occurs. The effect of the stall is actually to occupy the resources for that instruction slot. The following table shows how the stalls are actually implemented.



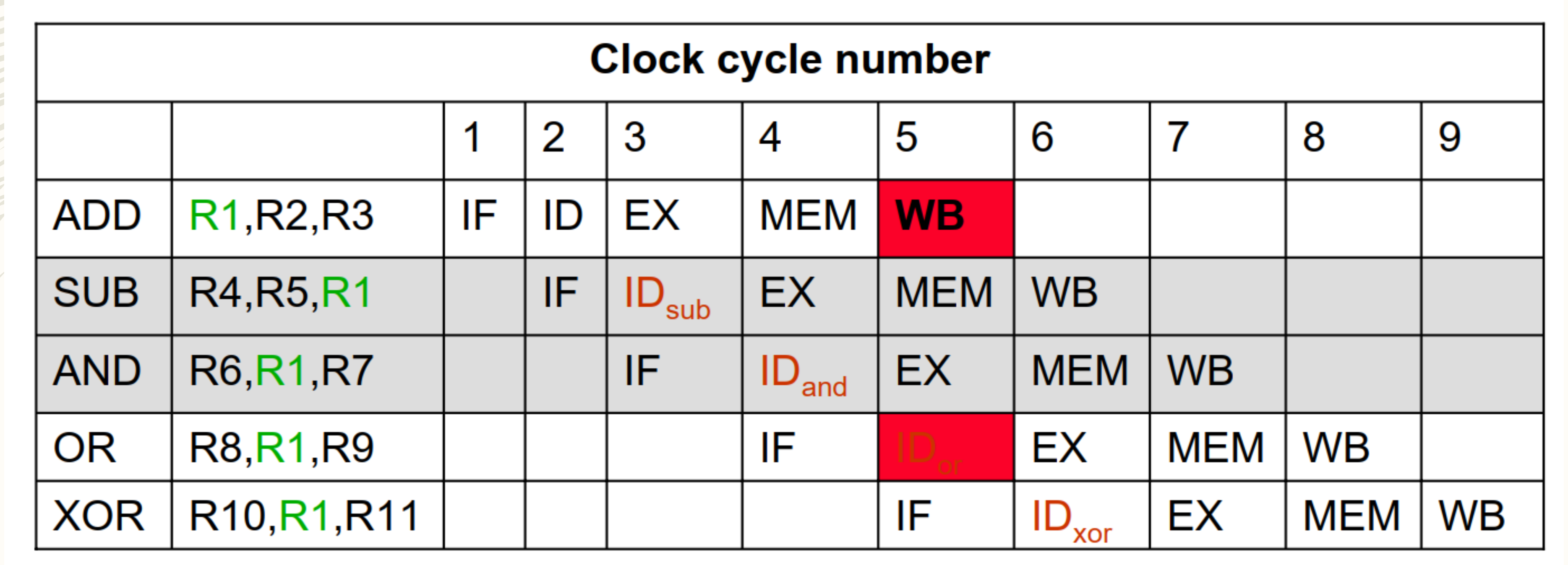
	Clock cycle number								
instr	1	2	3	4	5	6	7	8	9
load	IF	ID	EX	**MEM**	WB				
Instr 1		IF	ID	EX	MEM	WB			
Instr 2			IF	ID	EX	MEM	WB		
Instr 3				**stall**	**IF**	ID	EX	MEM	WB

Solution(2/2)

- ❖ Another solution is to use separate instruction and data memories.
- ❖ ARM is use **Harvard** architecture, so we do not have this hazard

➢ Data Hazards

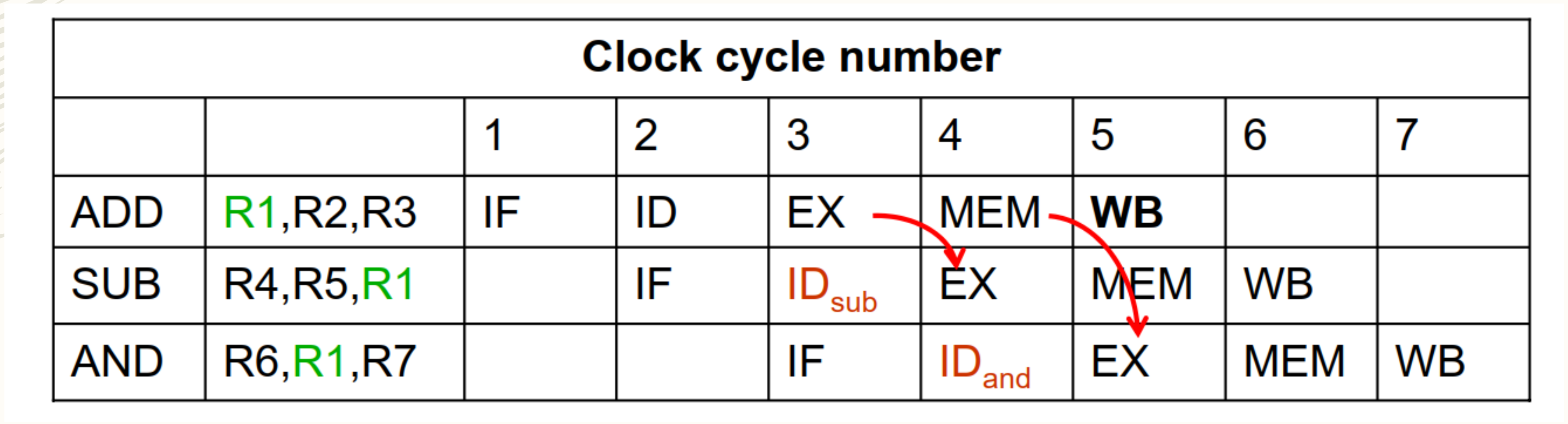
❖ **Data hazards** occur when the pipeline changes the order of read/write accesses to operands so that the order differs from the order seen by sequentially executing instructions on the unpipelined machine.



		Clock cycle number								
		1	2	3	4	5	6	7	8	9
ADD	R1,R2,R3	IF	ID	EX	MEM	**WB**				
SUB	R4,R5,R1		IF	ID$_{sub}$	EX	MEM	WB			
AND	R6,R1,R7			IF	ID$_{and}$	EX	MEM	WB		
OR	R8,R1,R9				IF	ID$_{or}$	EX	MEM	WB	
XOR	R10,R1,R11					IF	ID$_{xor}$	EX	MEM	WB

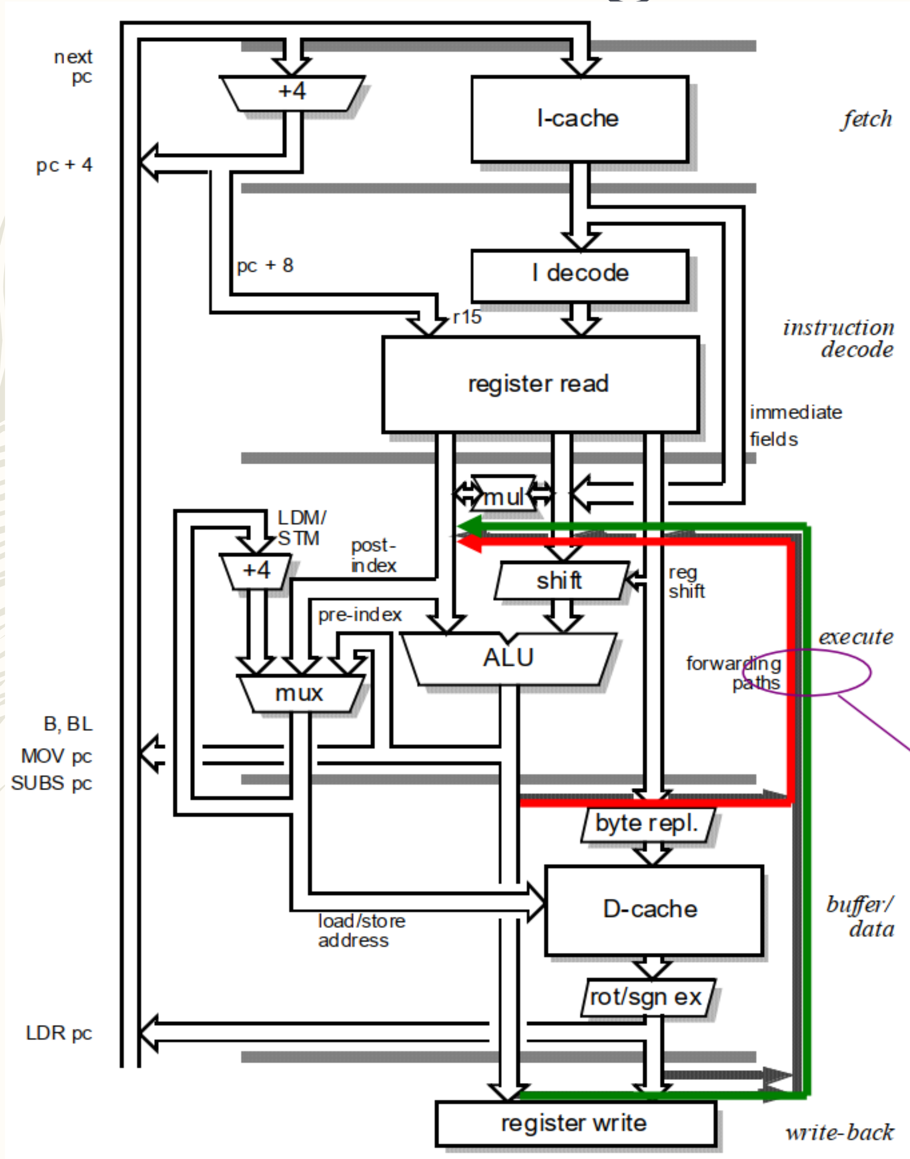
Forwarding

- The problem with data hazards, introduced by this sequence of instructions can be solved with a simple hardware technique called ***forwarding***



Clock cycle number								
		1	2	3	4	5	6	7
ADD	R1,R2,R3	IF	ID	EX	MEM	**WB**		
SUB	R4,R5,R1		IF	ID_{sub}	EX	MEM	WB	
AND	R6,R1,R7			IF	ID_{and}	EX	MEM	WB

Forwarding Architecture



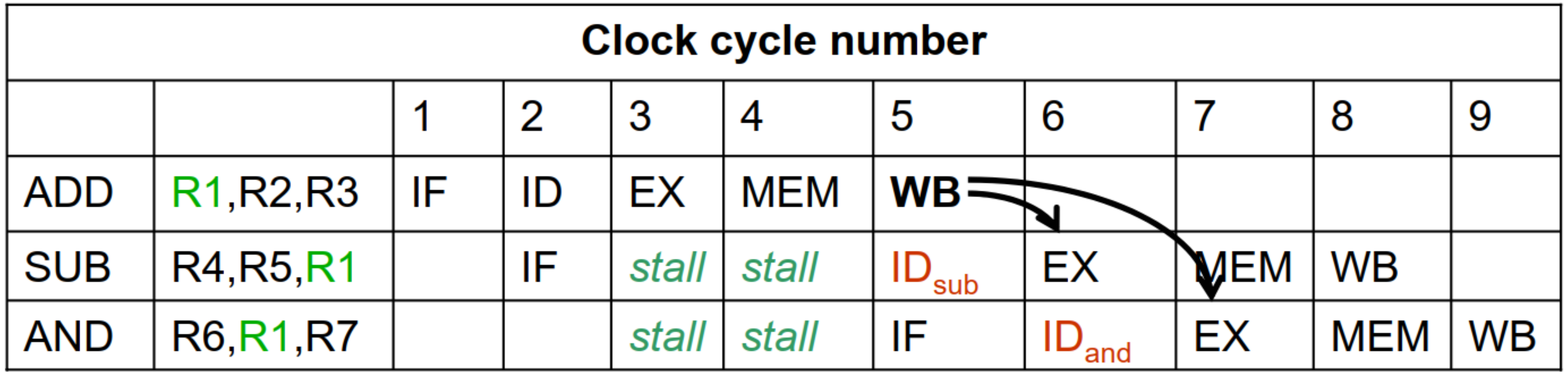
- Forwarding works as follows:
  - The ALU result from the EX/MEM register is always ***fed back*** to the ALU input latches.
  - If the forwarding hardware detects that the previous ALU operation has written the register corresponding to the source for the current ALU operation, ***control logic*** selects the forwarded result as the ALU input rather than the value read from the register file.

Forward Data

- The first forwarding is for value of **R1** from **EXadd** to **EXsub**. The second forwarding is also for value of **R1** from **MEMadd** to **EXand**. This code now can be executed without stalls.
- Forwarding can be generalized to include passing the result directly to the functional unit that requires it: a result is forwarded from the output of one unit to the input of another, rather than just from the result of a unit to the input of the same unit.

Clock cycle number								
		1	2	3	4	5	6	7
ADD	R1,R2,R3	IF	ID	EX_{add}	MEM_{add}	WB		
SUB	R4,R5,R1		IF	ID	EX_{sub}	MEM	WB	
AND	R6,R1,R7			IF	ID	EX_{and}	MEM	WB

Without Forward



Clock cycle number										
		1	2	3	4	5	6	7	8	9
ADD	R1,R2,R3	IF	ID	EX	MEM	**WB**				
SUB	R4,R5,R1		IF	*stall*	*stall*	ID_{sub}	EX	MEM	WB	
AND	R6,R1,R7			*stall*	*stall*	IF	ID_{and}	EX	MEM	WB

Data Forwarding

- ❖ Data dependency arises when an instruction needs to use the result of one of its predecessors before the result has returned to the register file => pipeline hazards
- ❖ Forwarding paths allow results to be passed between stages as soon as they are available
- ❖ 5-stage pipeline requires each of the three source operands to be forwarded from any of the intermediate result registers
- ❖ Still one load stall

  ```
  LDR rN, […]
  ADD r2,r1,rN ;use rN immediately
  ```

  – One stall

  – Compiler rescheduling

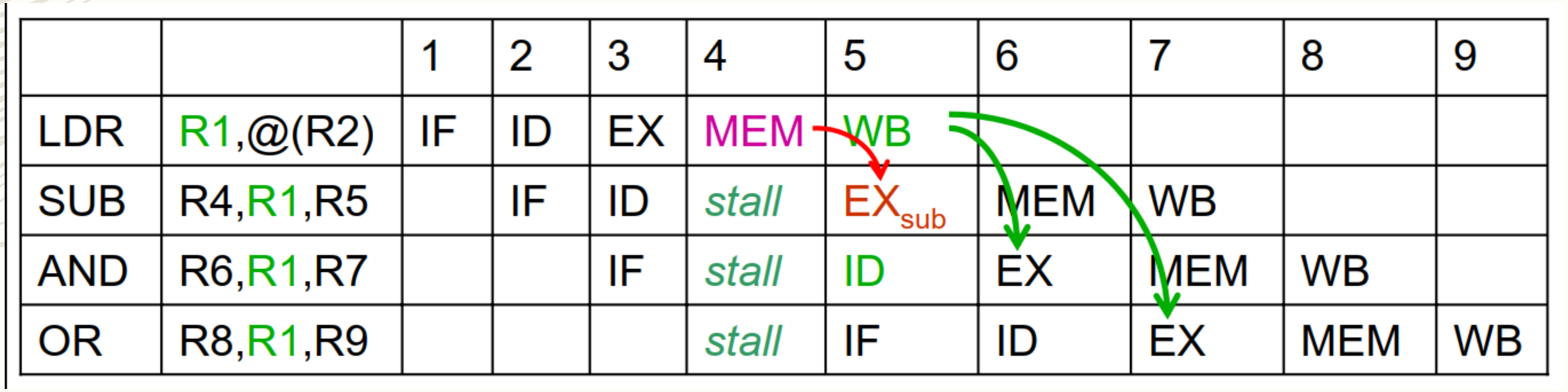
Stalls are Required

❖ The load instruction has a delay or latency that cannot be eliminated by forwarding alone.

		1	2	3	4	5	6	7	8
LDR	R1,@(R2)	IF	ID	EX	MEM	WB			
SUB	R4,R1,R5		IF	ID	EX_{sub}	MEM	WB		
AND	R6,R1,R7			IF	ID	EX_{and}	MEM	WB	
OR	R8,R1,R9				IF	ID	EXE	MEM	WB

The pipeline with one stall

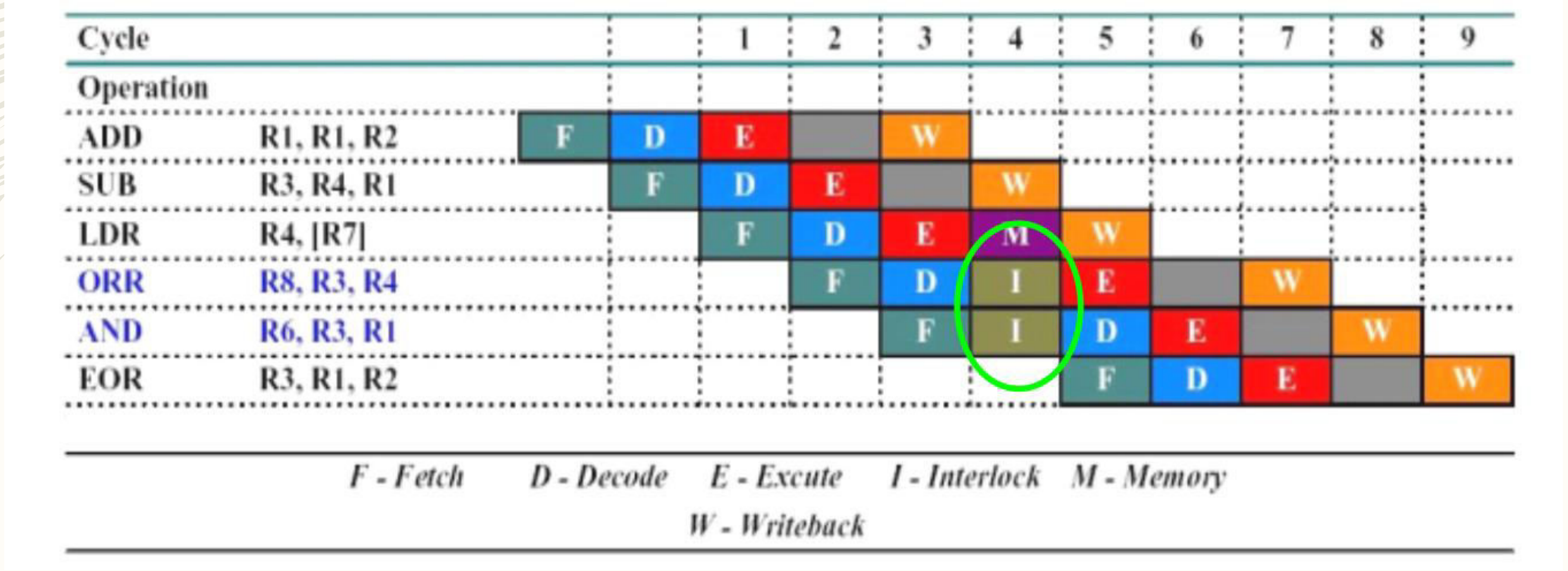
❖ The only necessary forwarding is done for R1 from **MEM** to **EXsub**.



		1	2	3	4	5	6	7	8	9
LDR	R1,@(R2)	IF	ID	EX	MEM	WB				
SUB	R4,R1,R5		IF	ID	*stall*	EX_{sub}	MEM	WB		
AND	R6,R1,R7			IF	*stall*	ID	EX	MEM	WB	
OR	R8,R1,R9				*stall*	IF	ID	EX	MEM	WB

LDR Interlock

- ❖ In this example, it takes 7 clock cycles to execute 6 instructions, CPI of 1.2
- ❖ The LDR instruction immediately followed by a data operation using the same register cause an interlock

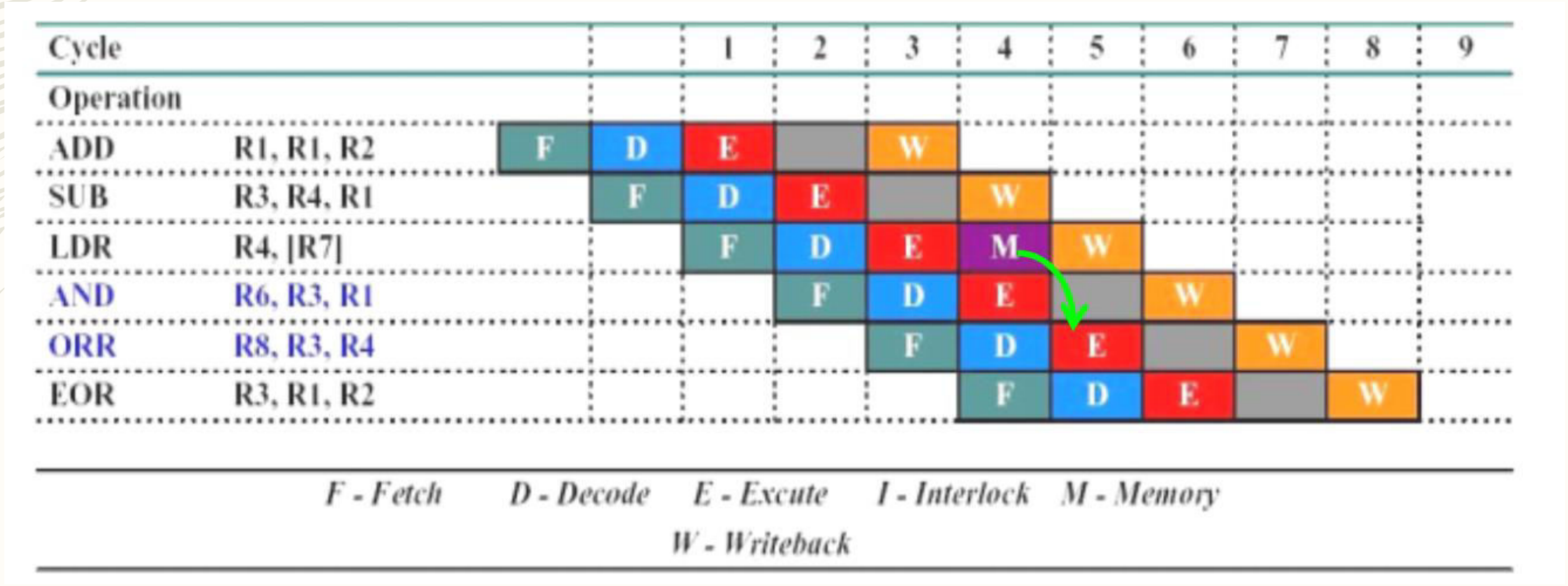


Cycle				1	2	3	4	5	6	7	8	9
Operation												
ADD	R1, R1, R2	F	D	E		W						
SUB	R3, R4, R1		F	D	E		W					
LDR	R4, [R7]			F	D	E	M	W				
ORR	R8, R3, R4				F	D	I	E		W		
AND	R6, R3, R1					F	I	D	E		W	
EOR	R3, R1, R2							F	D	E		W

F - Fetch *D - Decode* *E - Excute* *I - Interlock* *M - Memory*
W - Writeback

Optimal Pipelining

- In this example, it takes 6 clock cycles to execute 6 instructions, CPI of 1
- The LDR instruction does not cause the pipeline to interlock

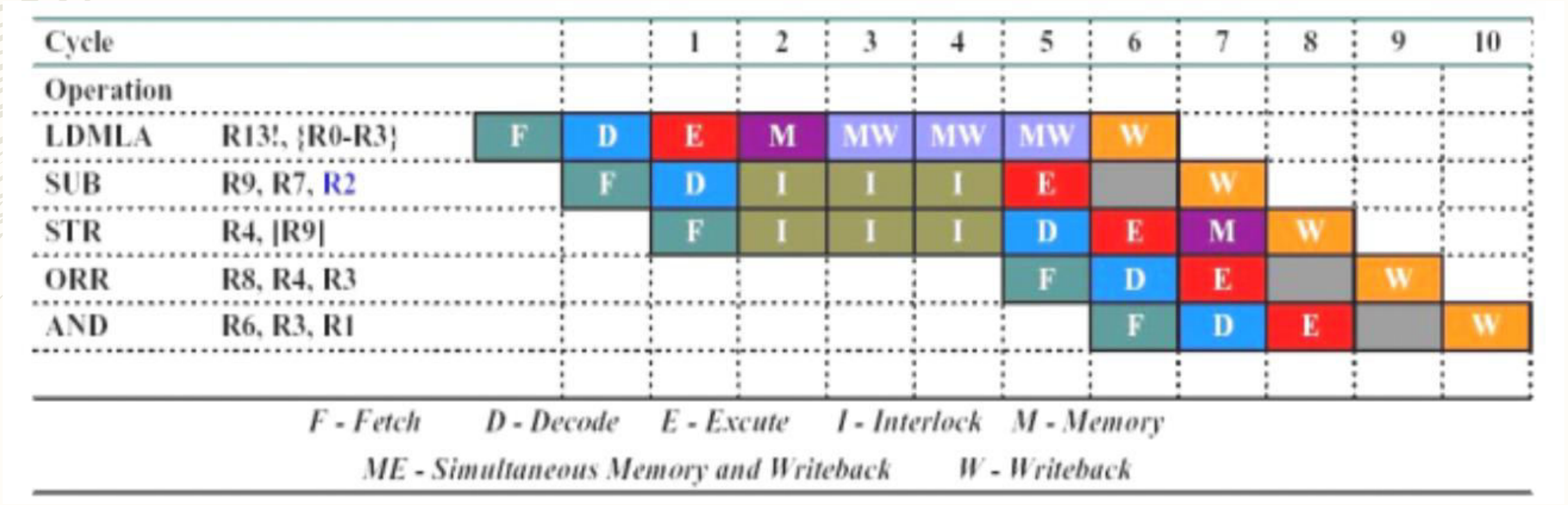


Cycle				1	2	3	4	5	6	7	8	9
Operation												
ADD	R1, R1, R2	F	D	E		W						
SUB	R3, R4, R1		F	D	E		W					
LDR	R4, [R7]			F	D	E	M	W				
AND	R6, R3, R1				F	D	E		W			
ORR	R8, R3, R4					F	D	E		W		
EOR	R3, R1, R2						F	D	E		W	

F - Fetch *D - Decode* *E - Excute* *I - Interlock* *M - Memory*
W - Writeback

LDM Interlock (1/2)

- In this example, it takes 8 clock cycles to execute 5 instructions, CPI of 1.6
- During the LDM there are parallel memory and writeback cycles



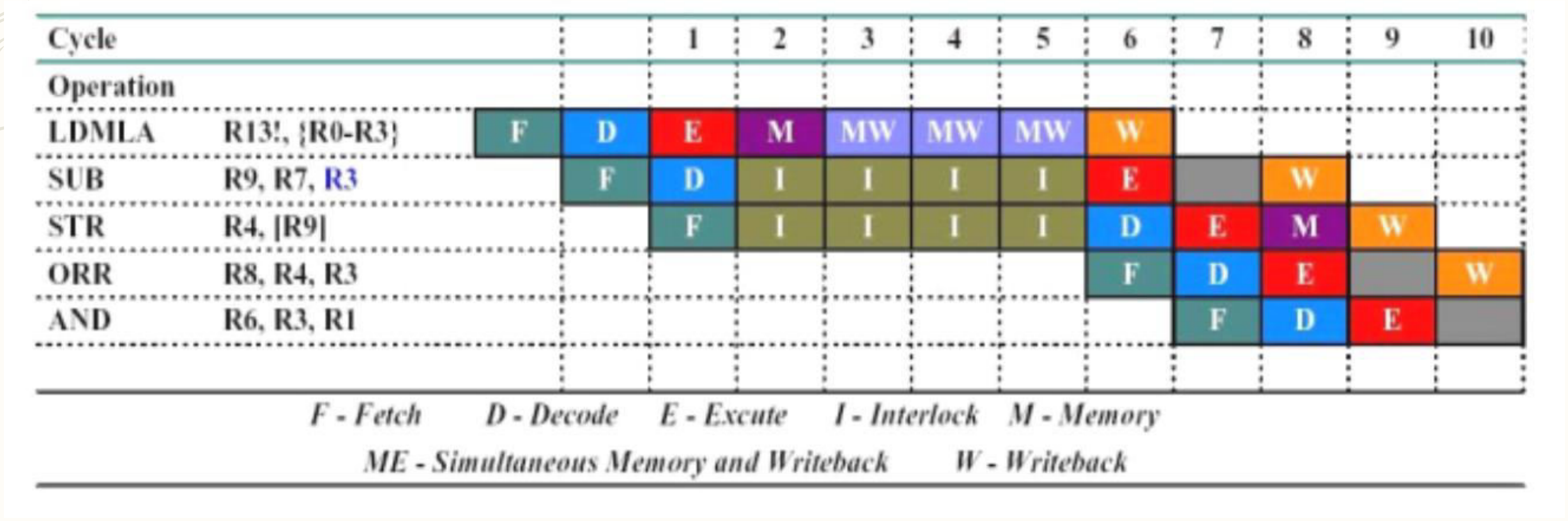
Cycle				1	2	3	4	5	6	7	8	9	10
Operation													
LDMLA	R13!, {R0-R3}	F	D	E	M	MW	MW	MW	W				
SUB	R9, R7, R2		F	D	I	I	I	E	▨	W			
STR	R4, [R9]			F	I	I	I	D	E	M	W		
ORR	R8, R4, R3							F	D	E	▨	W	
AND	R6, R3, R1								F	D	E	▨	W

F - Fetch D - Decode E - Excute I - Interlock M - Memory

ME - Simultaneous Memory and Writeback W - Writeback

LDM Interlock (2/2)

- In this example, it takes 9 clock cycles to execute 5 instructions, CPI of 1.8
- The SUB incurs a further cycle of interlock due to it using the highest specified register in the LDM instruction



Cycle				1	2	3	4	5	6	7	8	9	10
Operation													
LDMLA	R13!, {R0-R3}	F	D	E	M	MW	MW	MW	W				
SUB	R9, R7, R3		F	D	I	I	I	I	E		W		
STR	R4, [R9]			F	I	I	I	I	D	E	M	W	
ORR	R8, R4, R3								F	D	E		W
AND	R6, R3, R1									F	D	E	

F - Fetch *D - Decode* *E - Excute* *I - Interlock* *M - Memory*

ME - Simultaneous Memory and Writeback *W - Writeback*

Thank You!