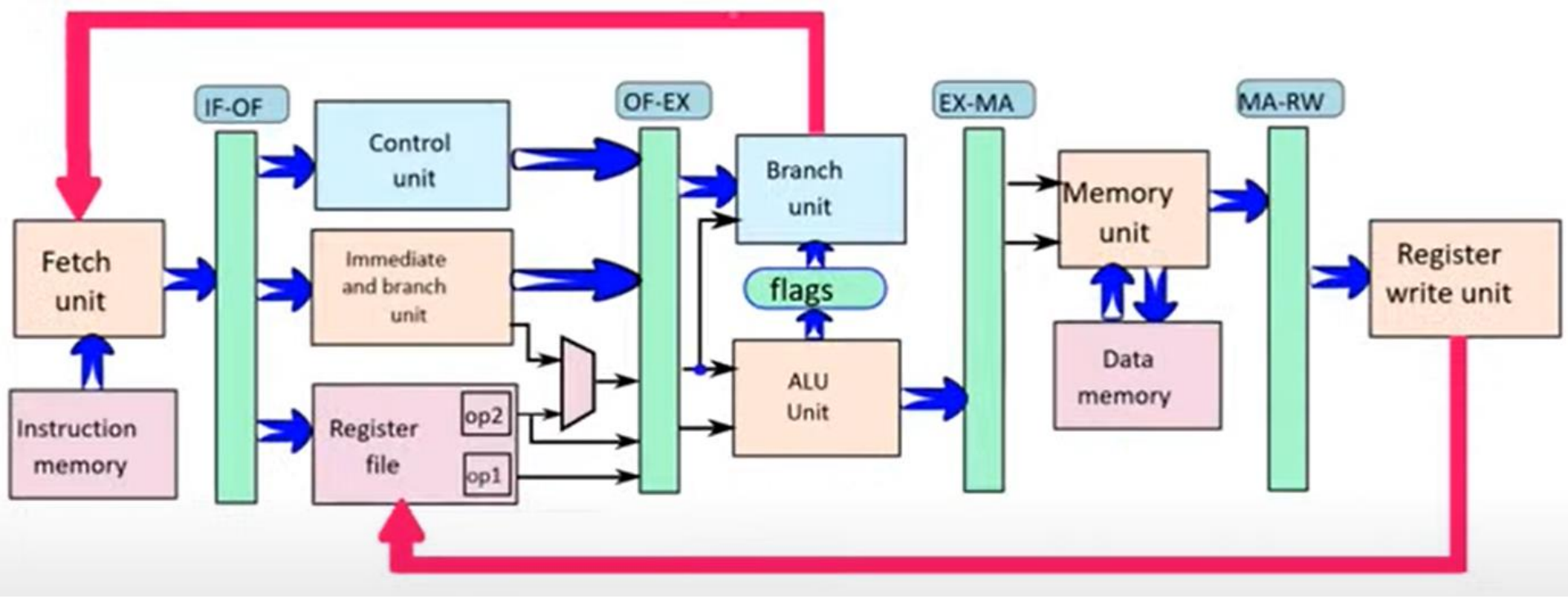
Chapter 4 – Annexure - Pipeline Hazards & Solutions

- Single Pipeline Flow
- Hazards - Data, Structural, & Control
- Examples & Solutions to certain hazards

In-Order Pipeline Components and Flow Mechanism



Instruction Fetch stage

Operand Fetch & Decode stage

Execution stage

Memory access stage

Register write-back stage

Ref: Computer Org. & Arch, 5th Edition, 2015, SR Sarangi.

R-A-W Hazard

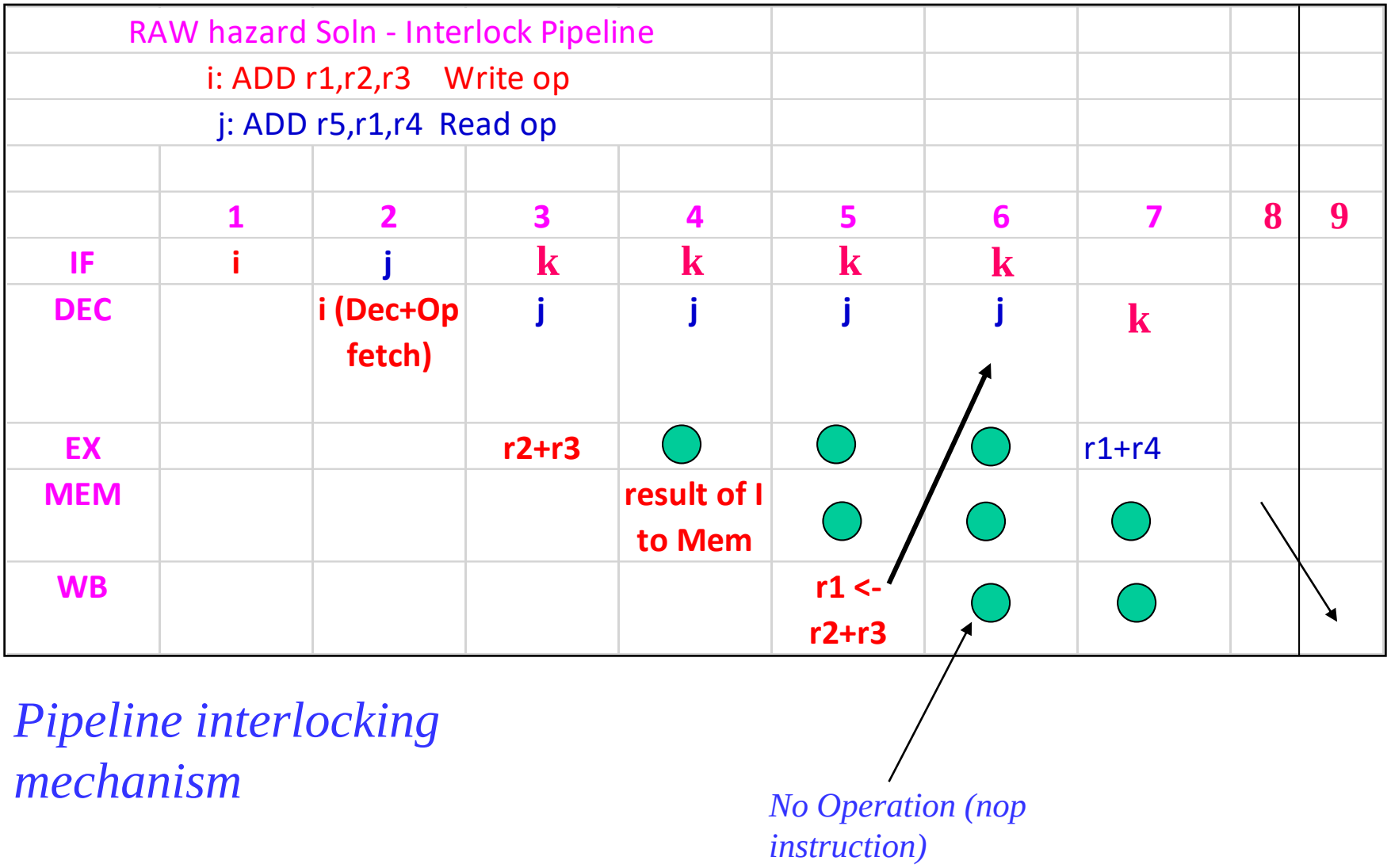
i: ADD r1,r2,r3 Write
j: ADD r5,r1,r4 Read *RAW Hazard*

Data Dependence RAW hazard							
i: ADD r1,r2,r3 Write op							
j: ADD r5,r1,r4 Read op							
	1	2	3	4	5	6	7
IF	i	j					
DEC		i (Dec+Op fetch)	j(Dec+Op fetch)				
EX			r2+r3	X			
MEM				result of i to Mem	X		
WB					r1 <- r2+r3	X	

Wrong value fetched for register r1

Solution?

R-A-W Hazard – Solution – Delay (introducing "bubbles")

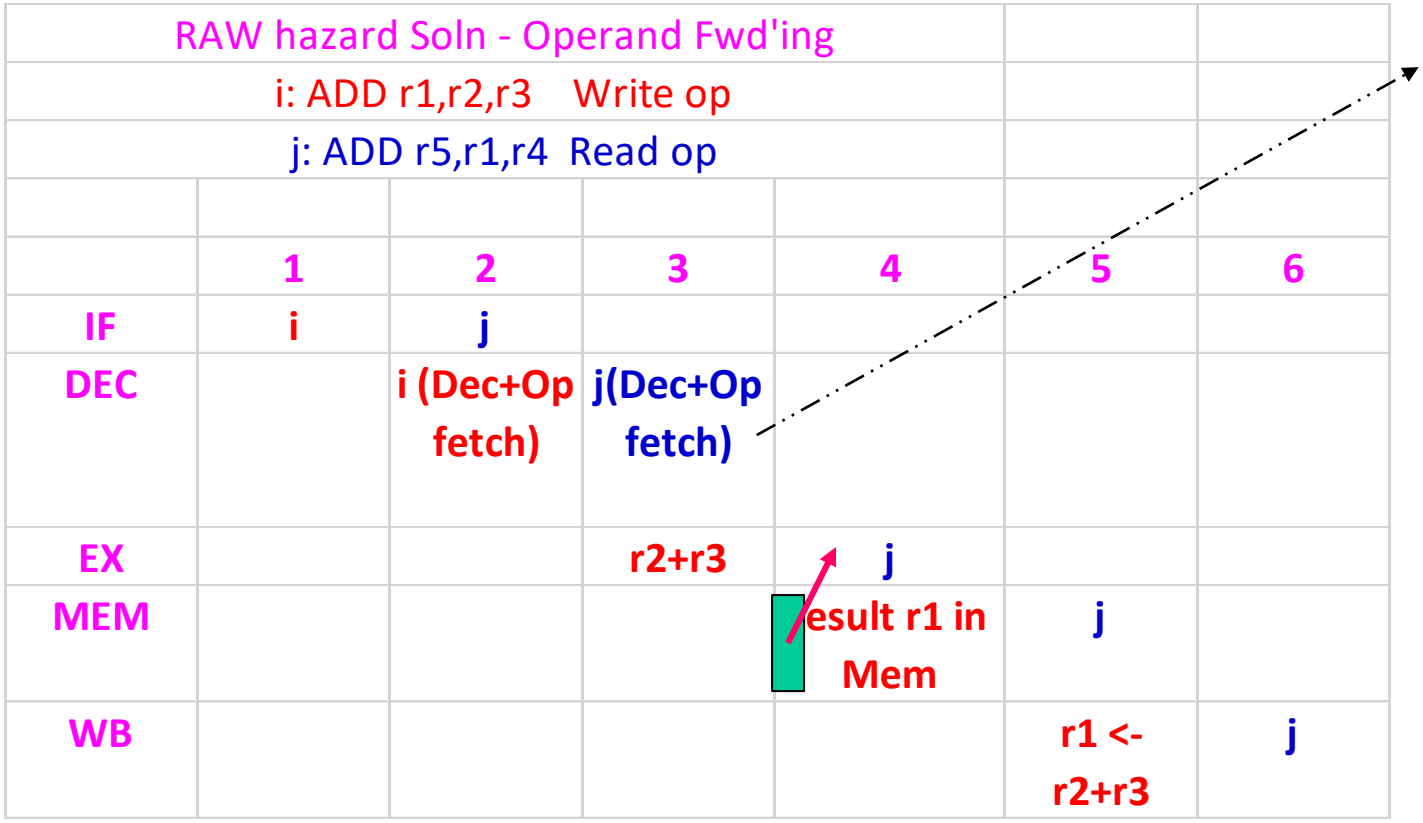


Pipeline interlocking mechanism

R-A-W Hazard – Solution – Operand Forwarding

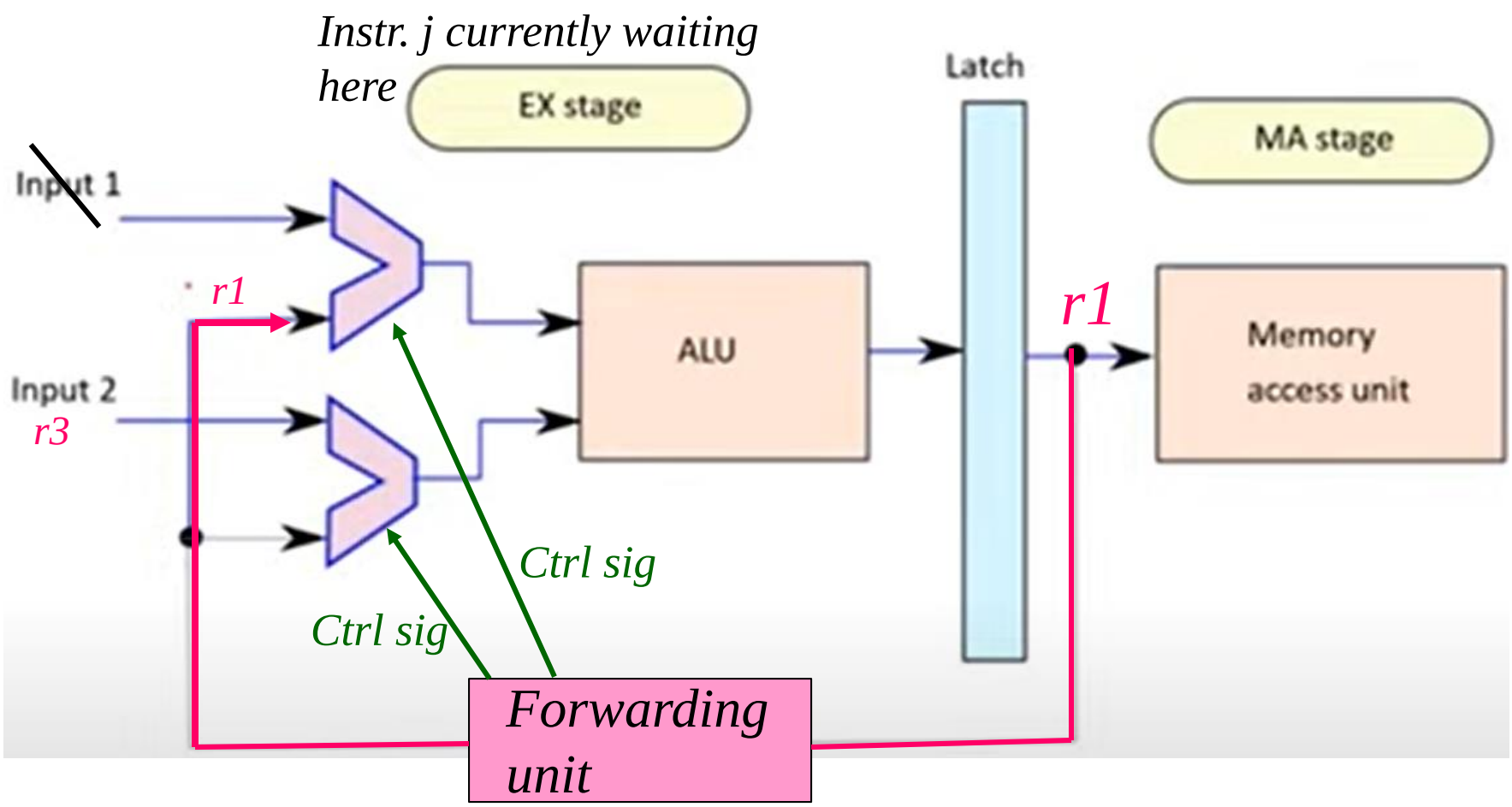
Operand forwarding – Forwarding from the Mem stage to Ex stage – No pipeline stalling!

Forward the result of the execution unit of the first instruction from the Mem stage at the time of execution of the second instruction (at the start of t=4 in this example).



*Note- We allow inst. **j** to read r1 in cycle 3 as it will not use it until cycle 4!!*

R-A-W Hazard – Solution – Operand Forwarding – Hardware realization



Note that we fwd almost just-in-time so that more cycles are not wasted; this is rather a design choice; We may also need to fwd from WB stage to Ex stage, if required. Consequently, we have 4 fwd'ing options (see the next slide).

R-A-W - Operand Forwarding – Different possible pathways

Forwarding Path Options & Examples			
WB -> Mem	LD r1,100[r2] ST r1,100[r3]	*LD & ST are mem write ops*	

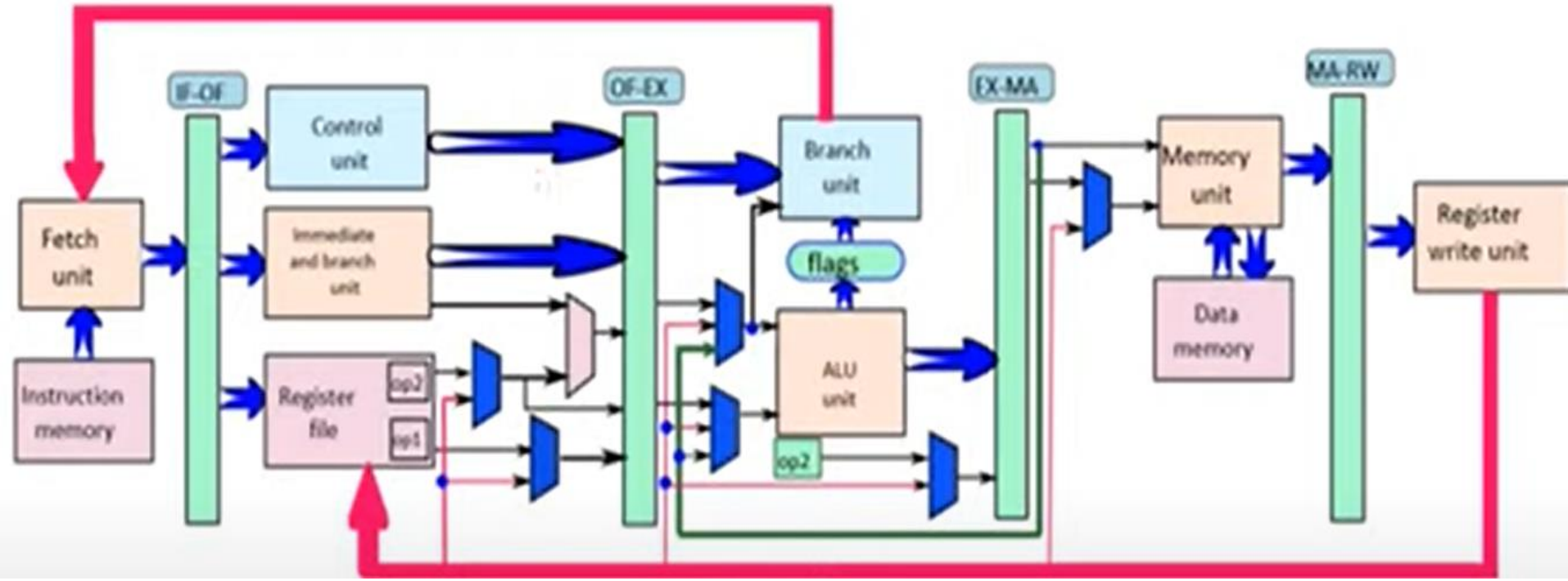
WB -> Ex	LD r1,100[r2]
	SUB r5,r6,r7
	ADD r3,r2,r1		

WB -> Dec/OF	LD r1,100[r2]
	SUB r5,r6,r7
	SUB r8,r9,r10
	ADD r3,r2,r1		

Mem -> Ex	ADD r1,r2,r3 SUB r5,r1,r4		

Only the above 4 paths are required for any scenario as far as RAW is concerned when the pipeline has 5 stags as seen in the examples above. For longer pipelines, fwd'ing paths may be many and can be complex.

In-Order Pipeline Components and Flow Mechanism – Complete design

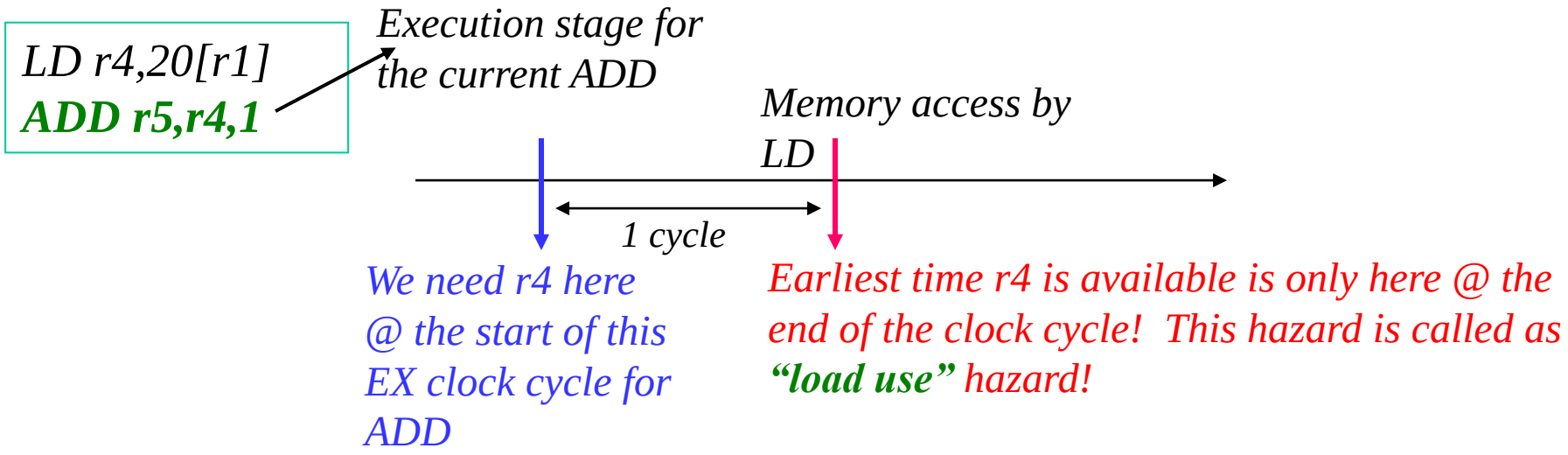


Forwarding unit (FU) is not shown in the above pic; Role of the FU is to know which instrs stall the pipeline, dependent instrs, and where to move the result of one stage to other and what are these stages;

Is there a case wherein operand forwarding technique cannot take care?

Classroom Discussions

Say, **in EX stage**, we have an instr: **ADD r5,r4,1** and let us say the previous instr now at the Mem stage is carrying out is: **LD r4, 20[r1]** - ***RAW hazard!***



Thus, "load use" hazard cannot avoid stalling and in this case, 1 cycle stall is unavoidable!

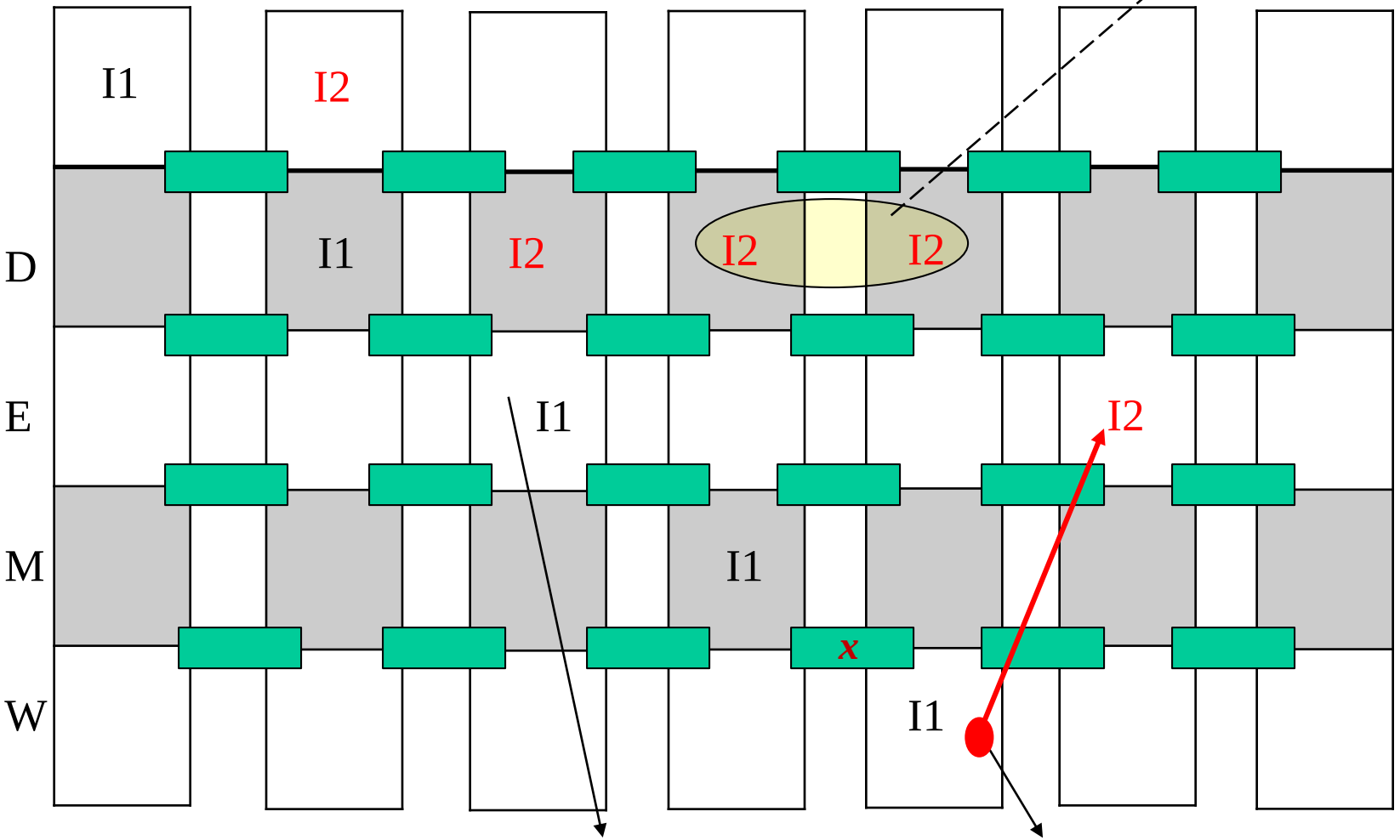
But, during WB (reg file writing) stage, value of r4 can be forwarded.

Classroom Discussions

I1: LD r7, 70[r1]
I2: ADD r5,r7,1

No operand forwarding

So, from D stage, 2 cycles stall is inevitable



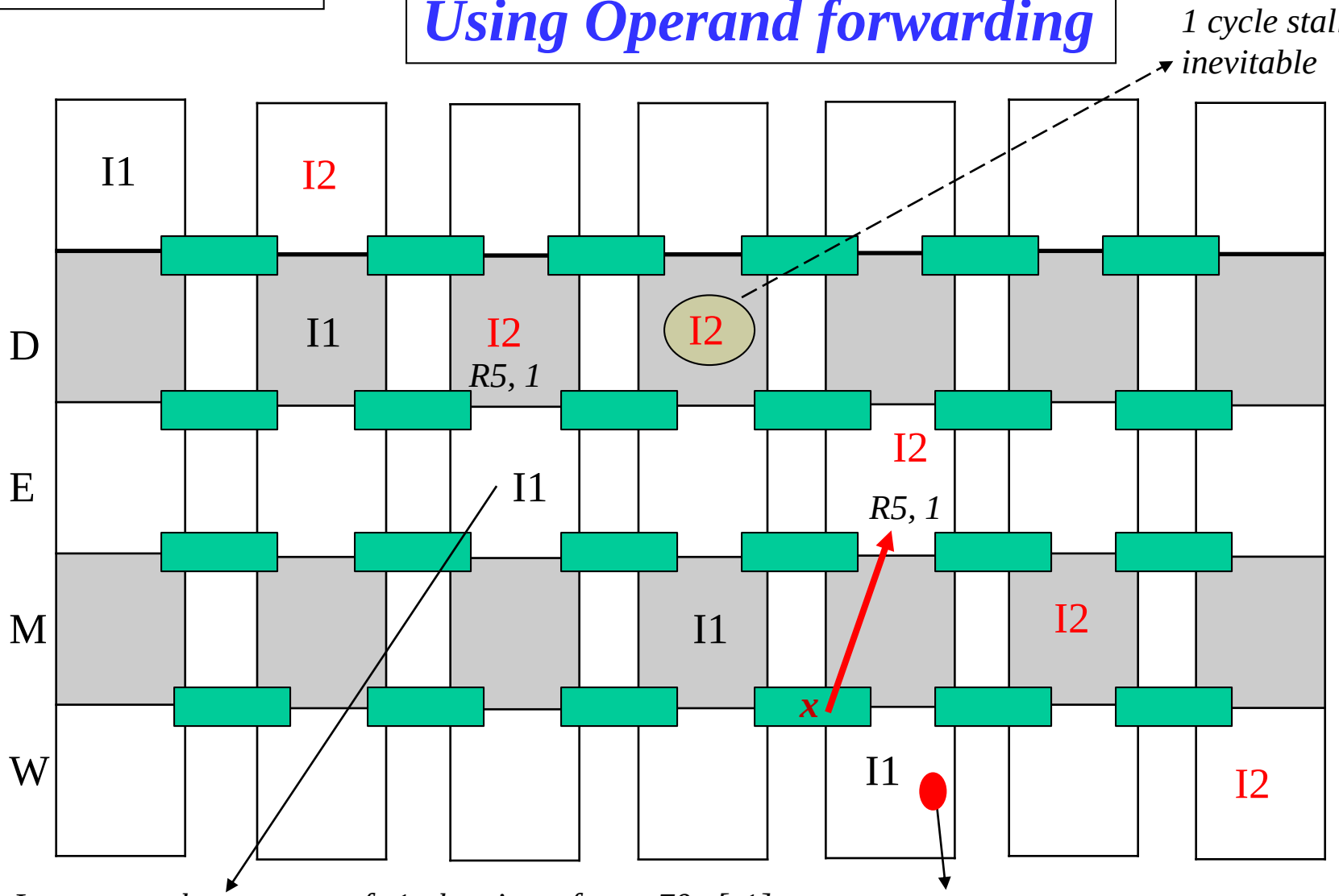
*It accesses the contents of r1, then it performs 70+[r1] and during the data mem stage, it accesses the location 70+[r1] and retrieves its contents – say, it is **x***

Result in r7 available only here

Classroom Discussions

I1: LD r7, 70[r1]
I2: ADD r5,r7,1

Using Operand forwarding



It accesses the contents of r1, then it performs 70+[r1] ***and*** *during the data mem stage, it accesses the location 70+[r1] and retrieves its contents – say, it is* ***x***

Result in r7 available only here

Classroom Discussions

Consider the following 5 instructions. Compute the CPI, *with and without using operand forwarding techniques*. Use a standard 5 stage pipeline and assume each stage uses 1 clk cycle.

```
1.  LOAD R1, [R0]      // Load data from memory to R1
2.   ADD R2, R1, R3    // Add R1 and R3, store result in R2
3.  SUB R4, R2, R1     // Subtract R1 from R2, store result in R4
4.   STORE [R5], R2    // Store R2 to memory
5.  ADD R6, R4, R3     // Add R4 and R3, store result in R6
```

Without using operand forwarding

Instruction | Completion Time (cycles)

1. *LOAD R1, [R0] | 5*
2. *ADD R2, R1, R3 | 7(wait for R1) – load use hazard*
3. *SUB R4, R2, R1 | 8(wait for R2, R1)*
4. *STORE [R5], R2 | 9*
5. *ADD R6, R4, R3 | 10(wait for R4)*

Total # of clk cycles = 10; CPI = 10/5 = 2

With operand forwarding

1. *LOAD R1, [R0] | 5*
2. *ADD R2, R1, R3 | 6 – with op fwd'ing load use uses only 1 clk cycle*
3. *SUB R4, R2, R1 | 7 (forward R2, R1)*
4. *STORE [R5], R2 | 8*
5. *ADD R6, R4, R3 | 9 (forward R4)*

Total # of cycles = 9; CPI = 9/5 = 1.8;
Thus there is a 10% gain in performance!

Real-life Pipeline execution

In reality, load/store and arithmetic operations may consume different cycles during execution;

Also, there may be more cycles needed to fetch, decode and register reading (RR), memory access and writing, etc.

Example:

We will now see an example in which we have several stages using different cycles for memory and arithmetic operations

Real-life Pipeline execution

Example:

Assume we have the following pipeline that uses different cycles for memory and arithmetic operations

Load – **9 stages** (2 IF, 2 Dec, 1 RR, 1 ALU, 2 DM, 1 RW)

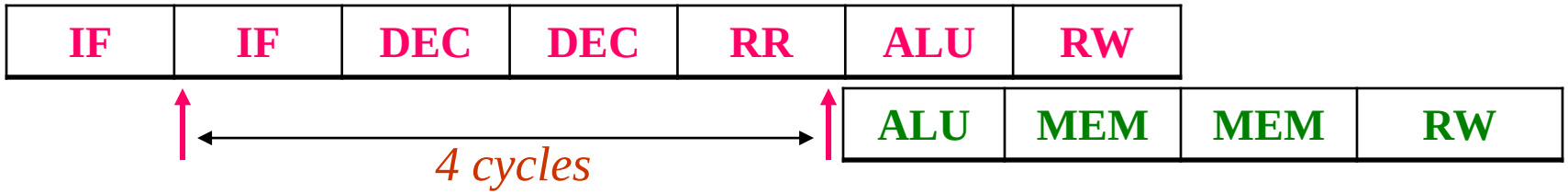
IF	**IF**	**DEC**	**DEC**	**RR**	**ALU**	**Mem**	**Mem**	**RW**

Arithmetic – 7 stages (2 IF, 2 Dec, 1 RR, 1 ALU, 1 RW)

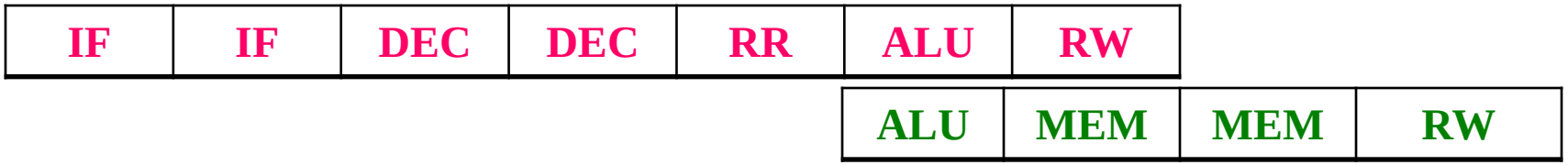
IF	**IF**	**DEC**	**DEC**	**RR**	**ALU**	**RW**

Consider executing the following instructions. Compute the total finish time of the instructions executed in that order and identify the number of stall cycles needed when using (i) No operand forwarding (ii) using operand forwarding

I1: LD r7, 70(r1) I2: ADD r5,r7,r8



(i) No operand forwarding – The result **x** is available only at the end of 2nd MEM cycle and then it will complete writing to r7 in cycle 9 in our case; So, I2 has to wait at the DEC stage for 4 cycles (note – ADD will enter in cycle 2 in the first IF stage); So, # of stall cycles = 4 for this example.



(ii) Using operand forwarding– In this case, I2 needs to stall for ___?____ cycles.

Pipeline Performance

- *IPC (depends on stalls, compiler, fwd'ing techniques)*
- *Instructions (compiler)*
- *frequency (transistor tech, # of pipelines, etc)*

What can we say about the IPC performance of an in-order pipeline?

= 1, if there are no stalls

< 1, if there are stalls – *This is a more practical case!*

What are the ways to improve the IPC performance?

- *Forwarding techniques*
- *Not many branch taken instructions*
- *faster data & instruction memories*

What can we say about Frequency?

Frequency = 1/ (clock period)

1 clk period means,

- 1 pipeline stage is expected to take 1 clk cycle
- So, 1 clock period = max {latency of the stages}

How to decrease the clock period?

- *make each stage smaller and increase the # of pipeline stages*
- *use faster transistor tech!*

Q: Can we arbitrarily increase the clock speed?

How many stages we can have and what is the influencing factor?

- More logic+latch - fewer stages
- Moderate logic size + latch - slightly more stages
- Smaller size logic + latch – large # of stages

In any case, we are constrained by the latch delays! Theoretically, even using infinite stages, the minimum clock period will be decided (equal to) by the latch delay!

With these issues, CPI is decided by the frequency of stalls happening in the pipeline! We redefine CPI for a pipeline as:

We know that, CPI = 1 / IPC

So, a revised pipeline CPI equation will be:

$CPI = CPI_{ideal} + stall\_rate * stall\_penalty$

Cycles per instr.

Note that,

- *stall_rate - # of stalls per instruction, is almost constant per instruction with the # of pipeline stages;*
- *stall_penalty - # of cycles per stall, expressed in clk cycles, will increase*

Thus, CPI will increase and this means there will be a loss in IPC!

Note that as we increase # of stages, IPC will decrease!

Example – *Effect of structural hazard* - Consider an instruction pipeline for a RISC processor where data references constitute 40% of the instructions, and the ideal CPI ignoring memory structural hazards is 1.25. Let the depth of the pipeline be k stages.

How much faster is the ideal machine without the memory structural hazard versus the machine with the hazard?

Solution approach –

We need to determine the ratio of the ideal case to the stalling case, which gives us the comparison needed.

Solution:

Speedup = (Ideal case CPI x Pipeline Depth) **/ (**Ideal case CPI + # of stall cycles per instruction**)**

(1) Ideal case Speedup = 1.25 * k / (1.25 + 0) = k

(2) Real case Speedup = 1.25 * k / (1.25 + 0.40 x 1) = 1.25*k / 1.65

(1)/(2) = 1.65 / 1.25 = 1.32
