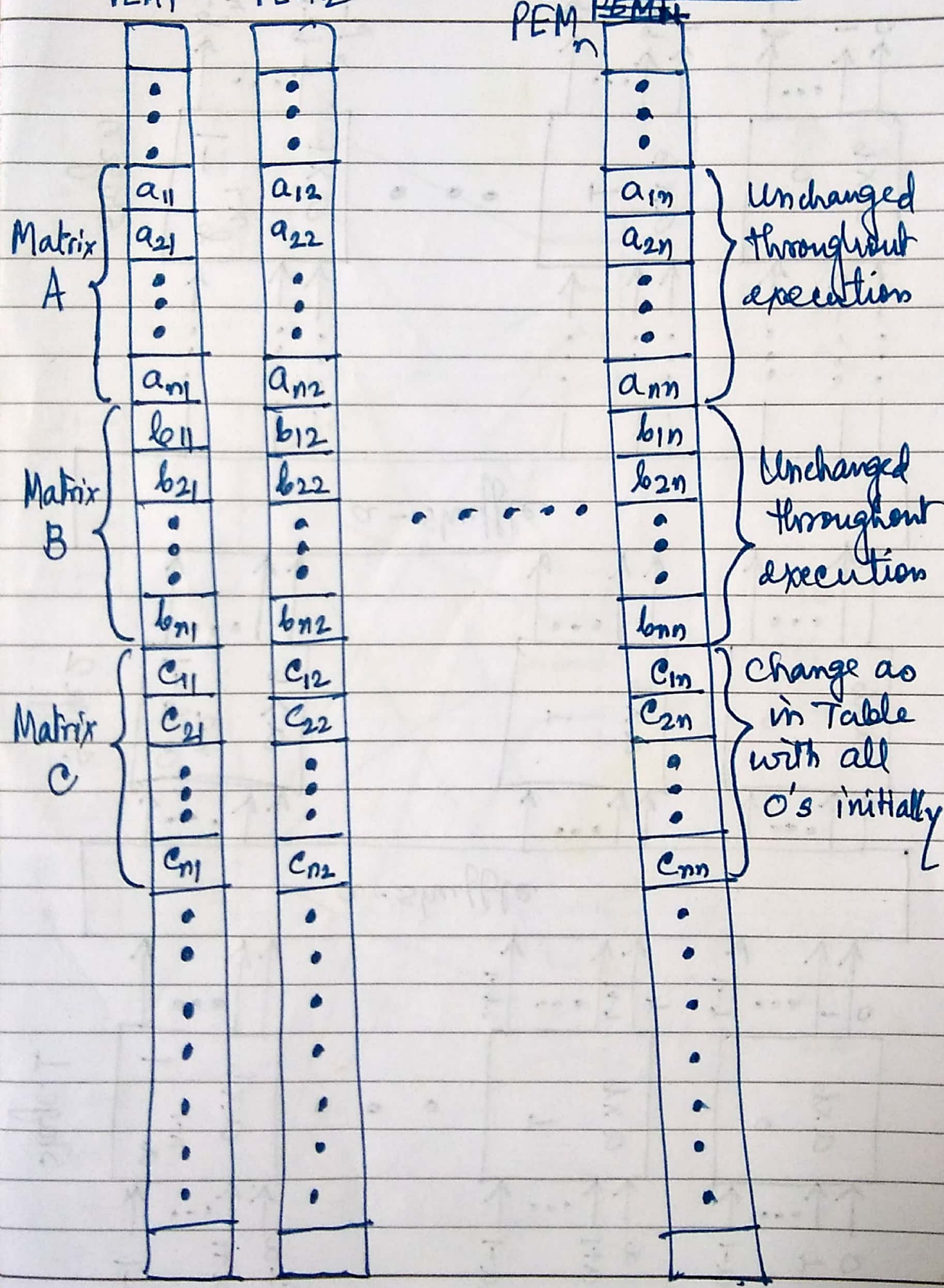
Memory Allocation for SIMD matrix multiplication

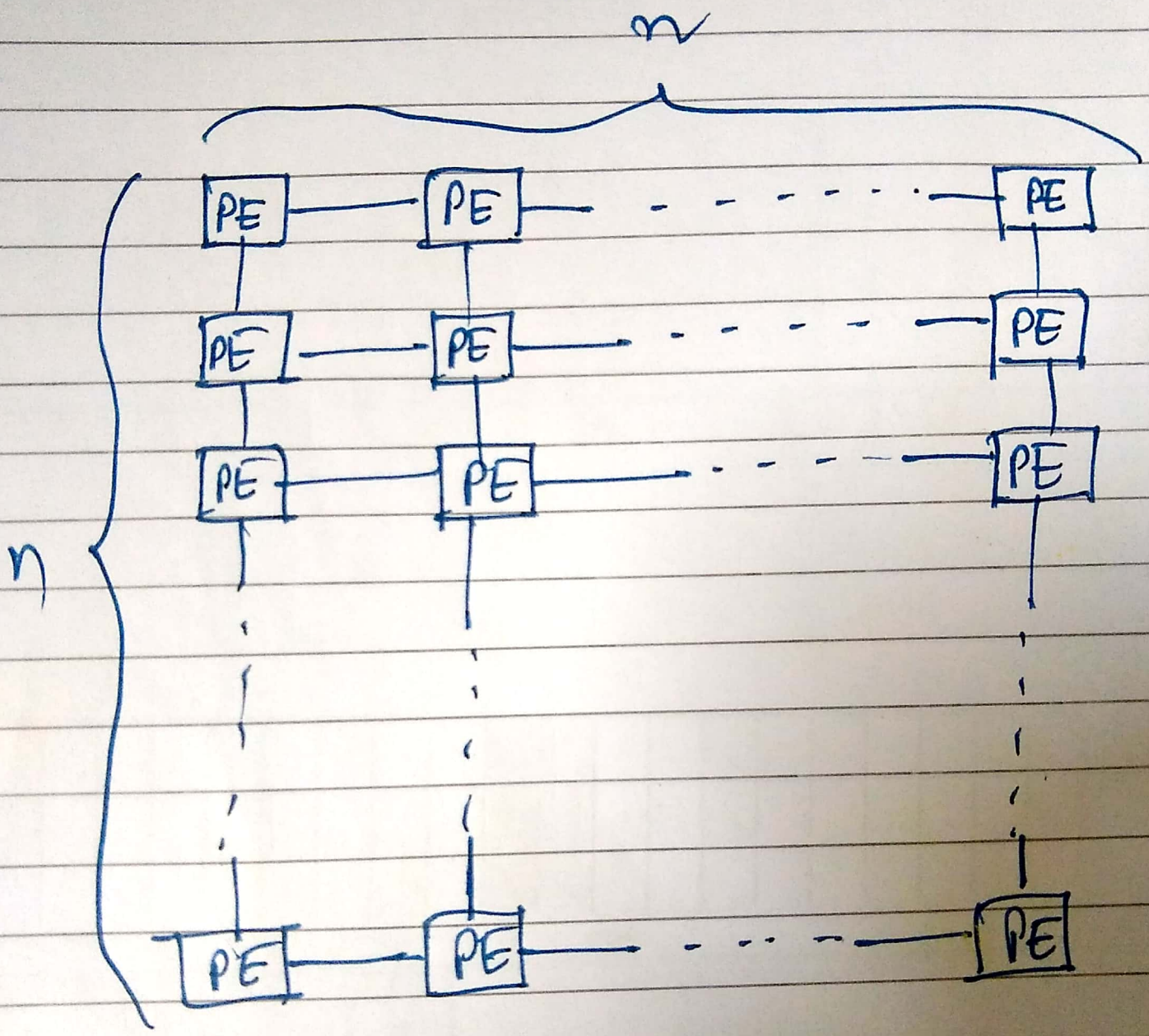


Outer loop (i)
1
2
Loca

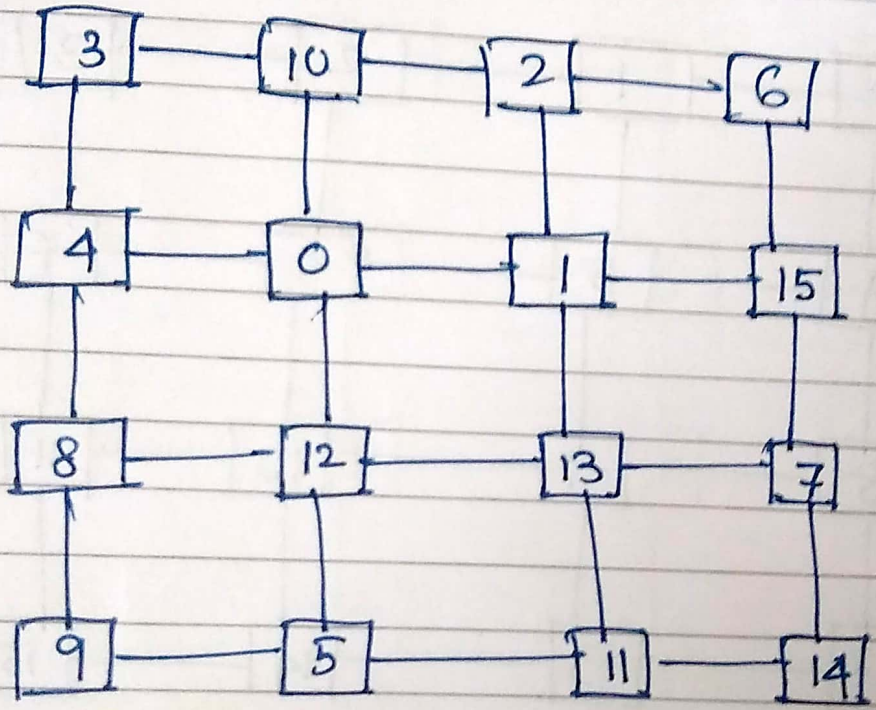
Successive contents of the C array in memory

Outer loop (i)	Inner loop (j)	Parallel SIMD operations on k = 1, 2, ..., n			
		$c_{i1} \leftarrow c_{i1} + a_{ij} \times b_{j1}$	$c_{i2} \leftarrow c_{i2} + a_{ij} \times b_{j2}$	. . . . .	$c_{in} \leftarrow c_{in} + a_{ij} \times b_{jn}$
1	1	$c_{11} \leftarrow c_{11} + a_{11} \times b_{11}$	$c_{12} \leftarrow c_{12} + a_{11} \times b_{12}$	. . . . .	$c_{1n} \leftarrow c_{1n} + a_{11} \times b_{1n}$
	2	$c_{11} \leftarrow c_{11} + a_{12} \times b_{21}$	$c_{12} \leftarrow c_{12} + a_{12} \times b_{22}$		$c_{1n} \leftarrow c_{1n} + a_{12} \times b_{2n}$
	⋮	⋮	⋮		⋮
	n	$c_{11} \leftarrow c_{11} + a_{1n} \times b_{n1}$	$c_{12} \leftarrow c_{12} + a_{1n} \times b_{n2}$		$c_{1n} \leftarrow c_{1n} + a_{1n} \times b_{nn}$
2	1	$c_{21} \leftarrow c_{21} + a_{21} \times b_{11}$	$c_{22} \leftarrow c_{22} + a_{21} \times b_{12}$	. . . . .	$c_{2n} \leftarrow c_{2n} + a_{21} \times b_{1n}$
	2	$c_{21} \leftarrow c_{21} + a_{22} \times b_{21}$	$c_{22} \leftarrow c_{22} + a_{22} \times b_{22}$		$c_{2n} \leftarrow c_{2n} + a_{22} \times b_{2n}$
	⋮	⋮	⋮		⋮
	n	$c_{21} \leftarrow c_{21} + a_{2n} \times b_{n1}$	$c_{22} \leftarrow c_{22} + a_{2n} \times b_{n2}$		$c_{2n} \leftarrow c_{2n} + a_{2n} \times b_{nn}$
⋮	⋮	⋮	⋮	⋮	⋮
n	1	$c_{n1} \leftarrow c_{n1} + a_{n1} \times b_{11}$	$c_{n2} \leftarrow c_{n2} + a_{n1} \times b_{12}$	. . . . .	$c_{nn} \leftarrow c_{nn} + a_{n1} \times b_{1n}$
	2	$c_{n1} \leftarrow c_{n1} + a_{n2} \times b_{21}$	$c_{n2} \leftarrow c_{n2} + a_{n2} \times b_{22}$		$c_{nn} \leftarrow c_{nn} + a_{n2} \times b_{2n}$
	⋮	⋮	⋮		⋮
	n	$c_{n1} \leftarrow c_{n1} + a_{nn} \times b_{n1}$	$c_{n2} \leftarrow c_{n2} + a_{nn} \times b_{n2}$		$c_{nn} \leftarrow c_{nn} + a_{nn} \times b_{nn}$
Local memory		PEM_1	PEM_2	. . . . .	PEM_n

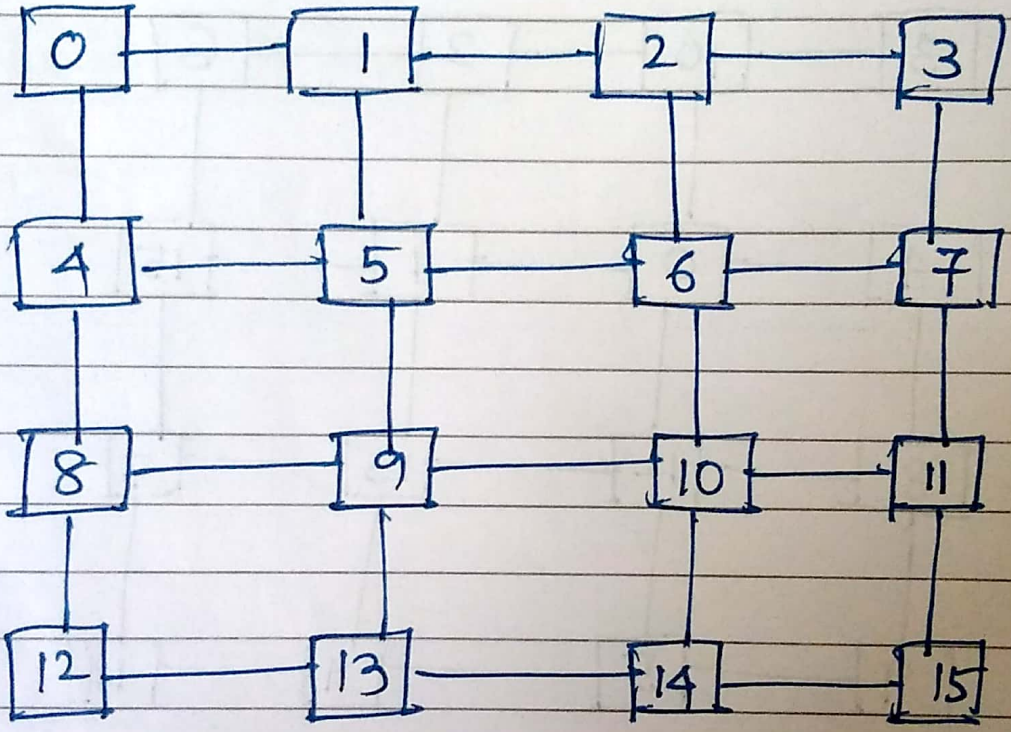
A mesh connection of PEs without boundary wraparound for SIMD sorting



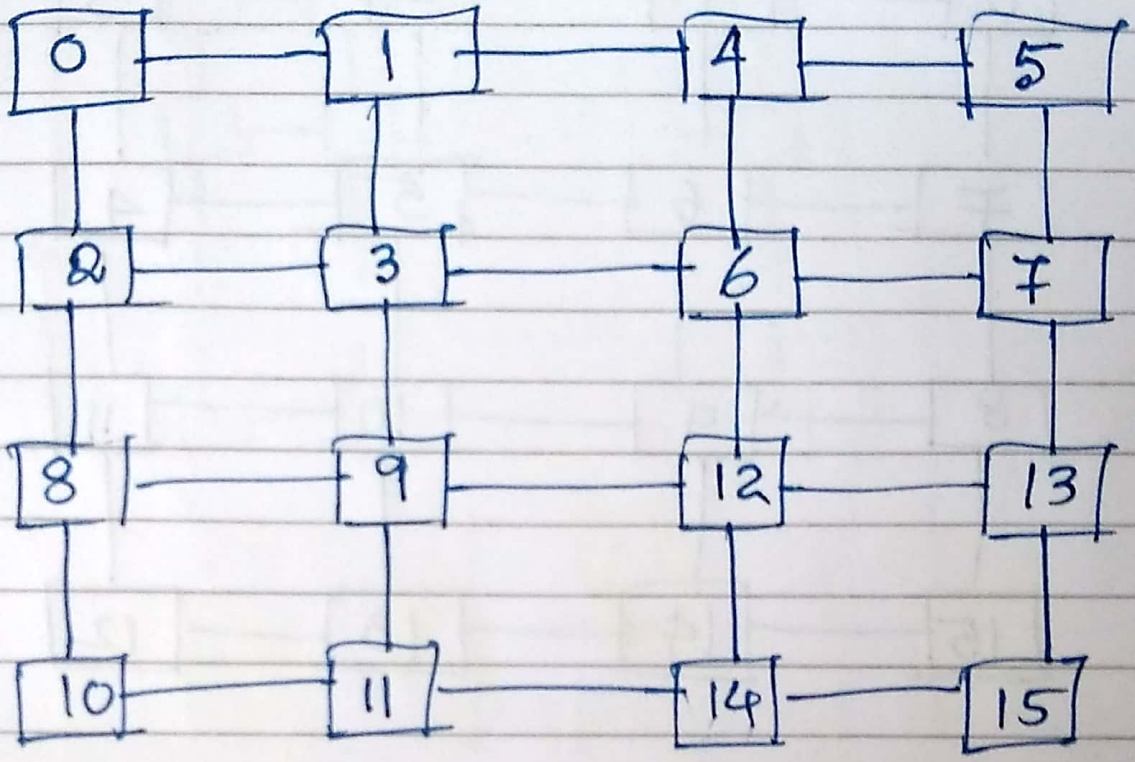
Initial Loading Pattern before sorting



Sorted Pattern with
Row-major indexing.

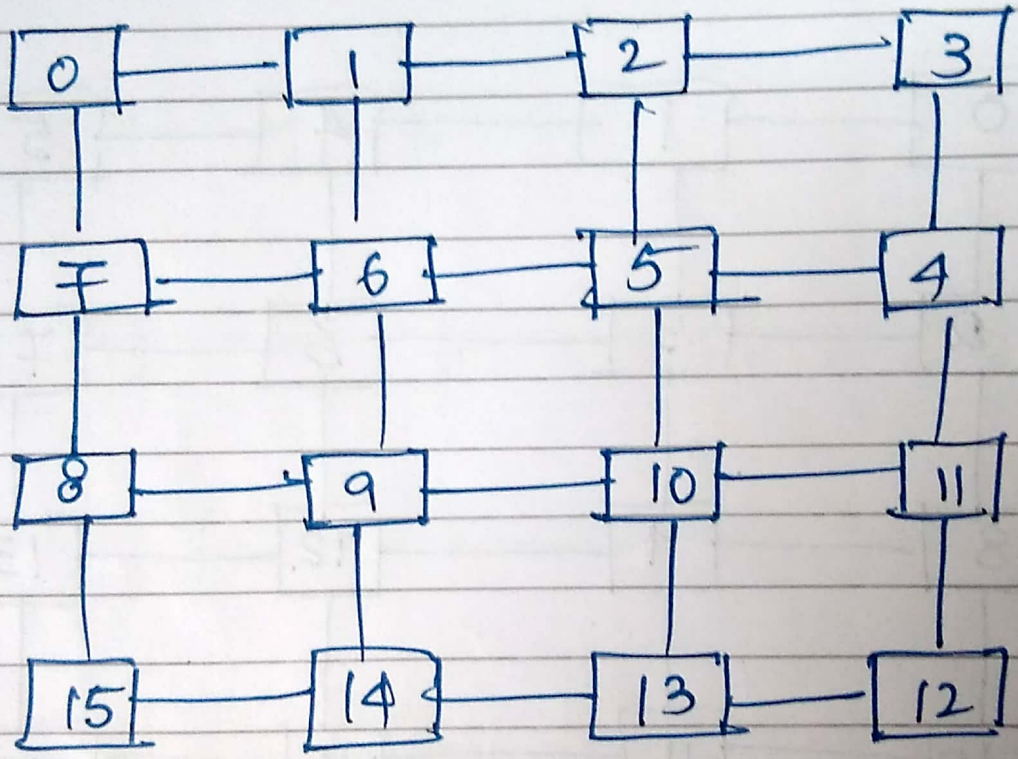


Sorted Pattern with Shuffled Row-major indexing

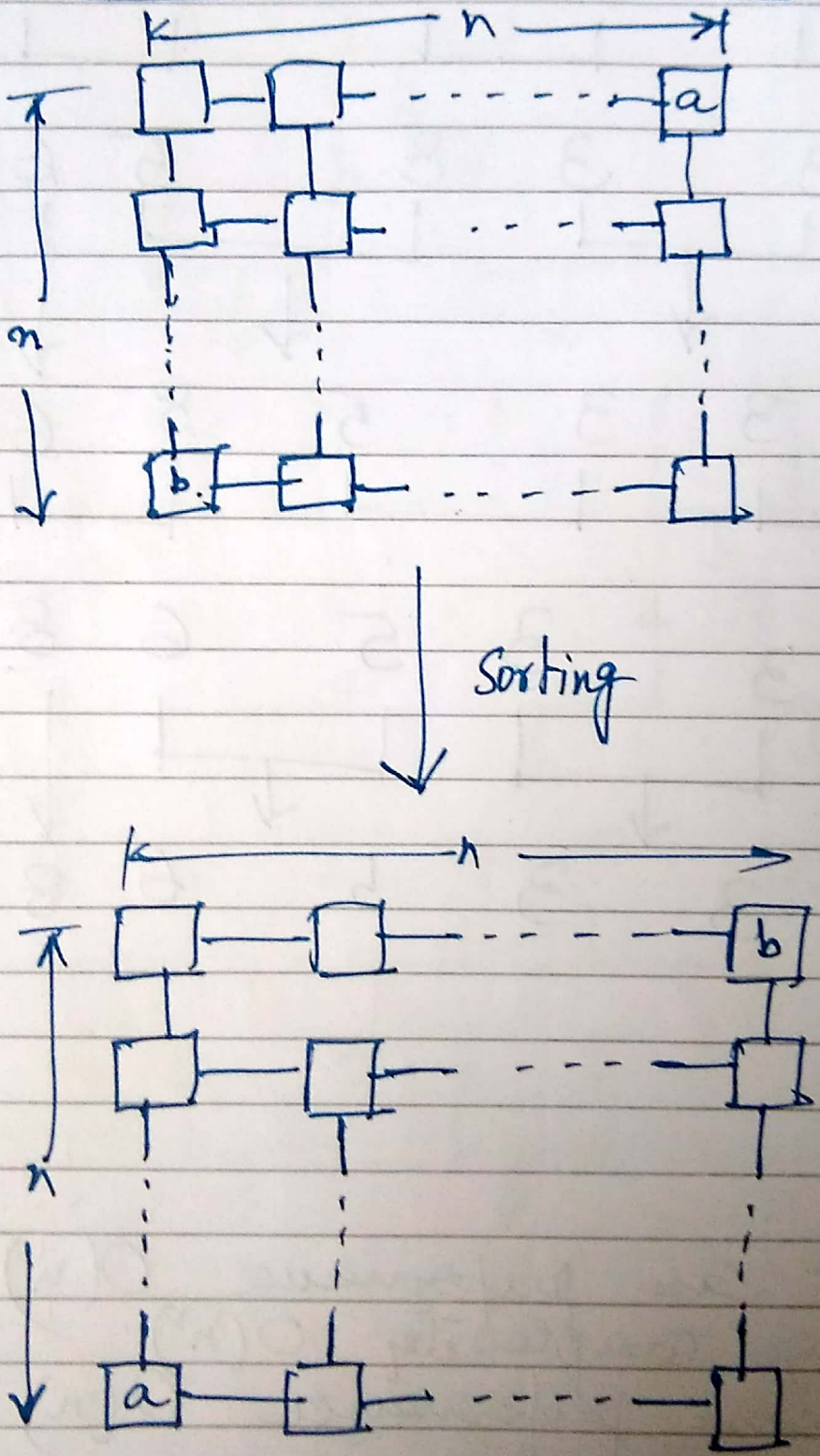


0	0000 → 0000	8	1000 → 1000
1	0001 → 0001	9	1001 → 1001
2	0010 → 0100	10	1010 → 1100
3	0011 → 0101	11	1011 → 1101
4	0100 → 0010	12	1100 → 1010
5	0101 → 0011	13	1101 → 1011
6	0110 → 0110	14	1110 → 1110
7	0111 → 0111	15	1111 → 1111

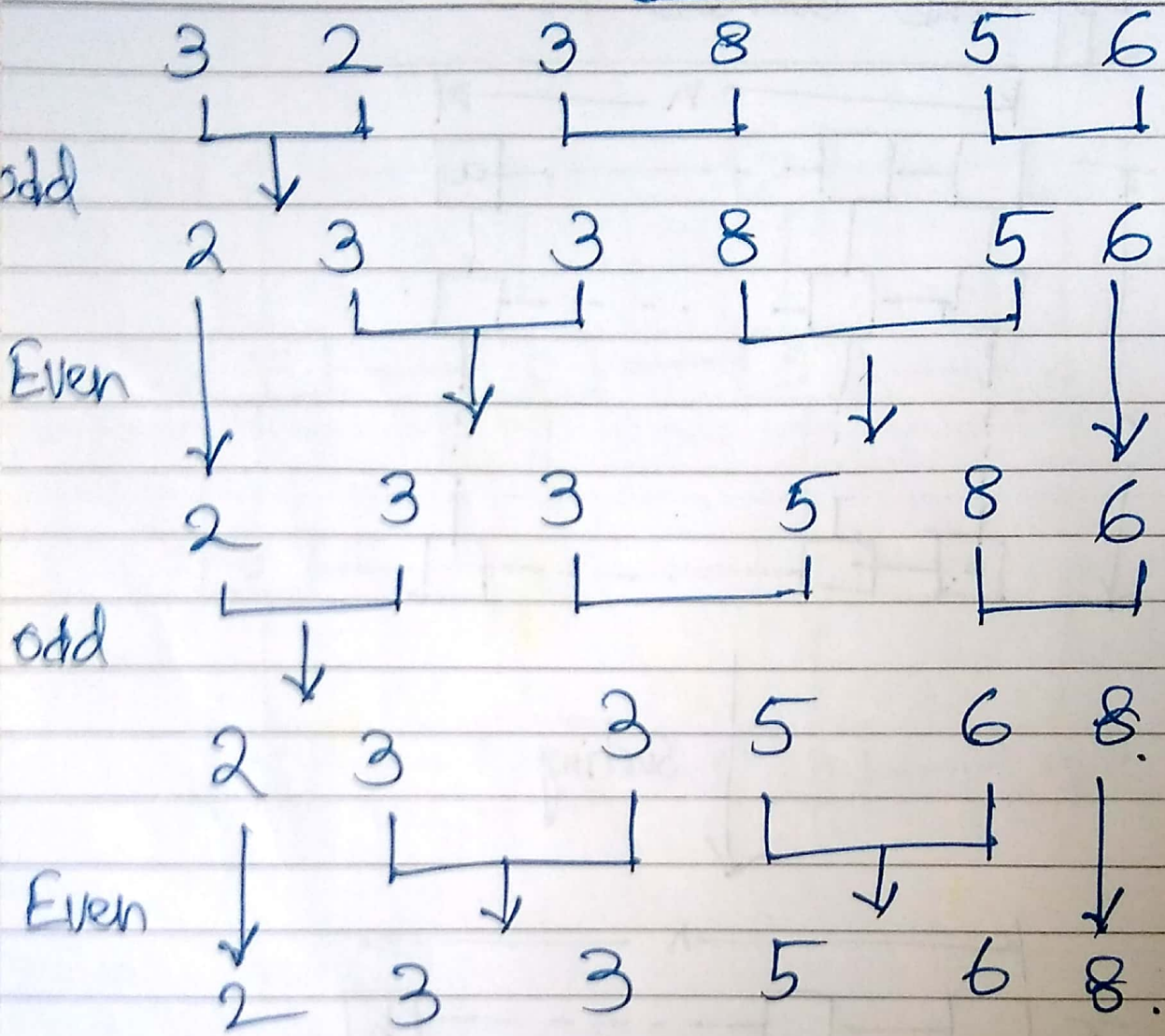
Sorted pattern with Snakelike row-major indexing



Transposition of two elements at opposite corner PEs



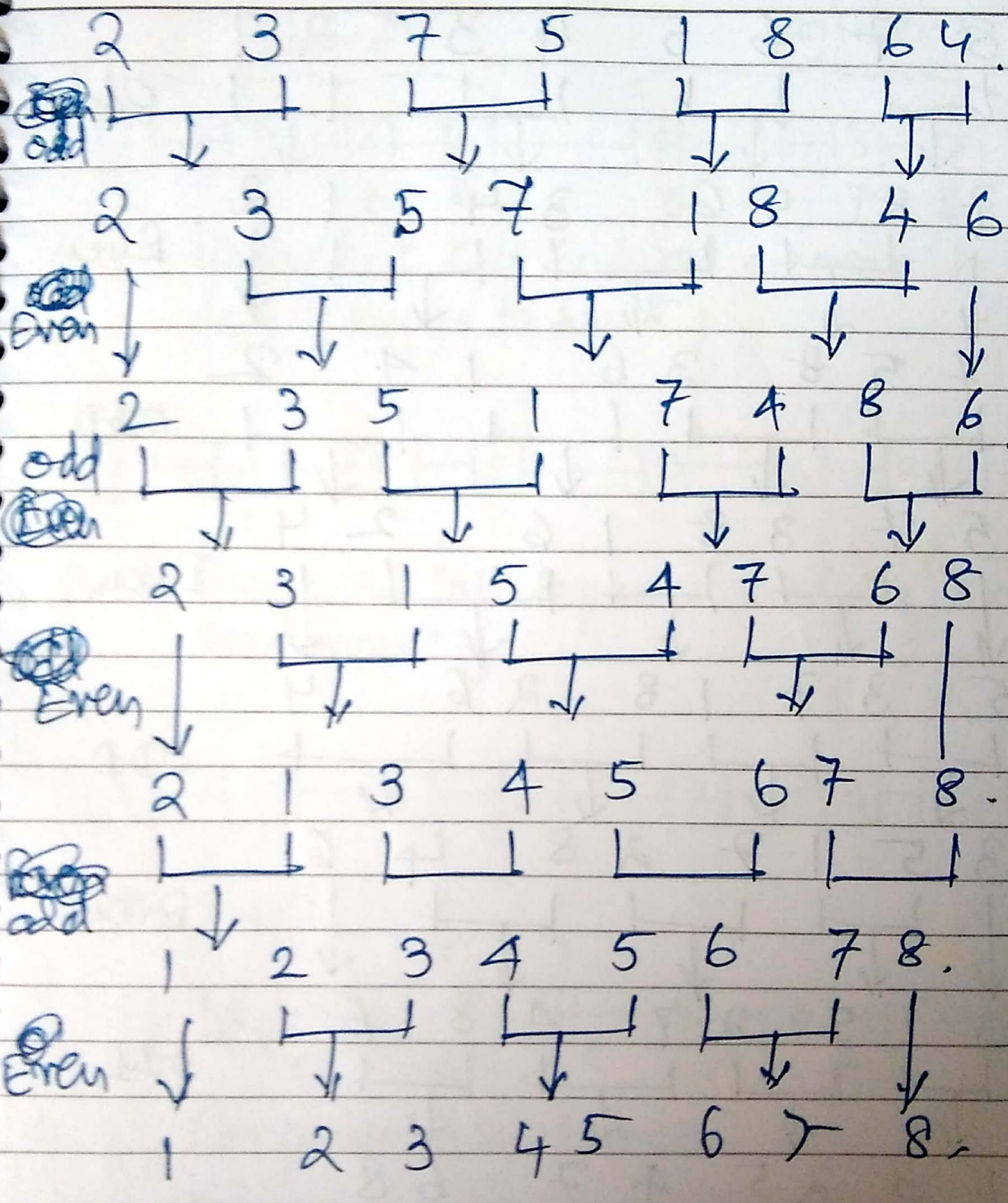
Example of odd-even Transposition Sort



Worst case performance $O(n^2)$
Average complexity $O(n^2)$
Best case performance $O(n)$

Time complexity is $O(n)$ due to parallel computation.

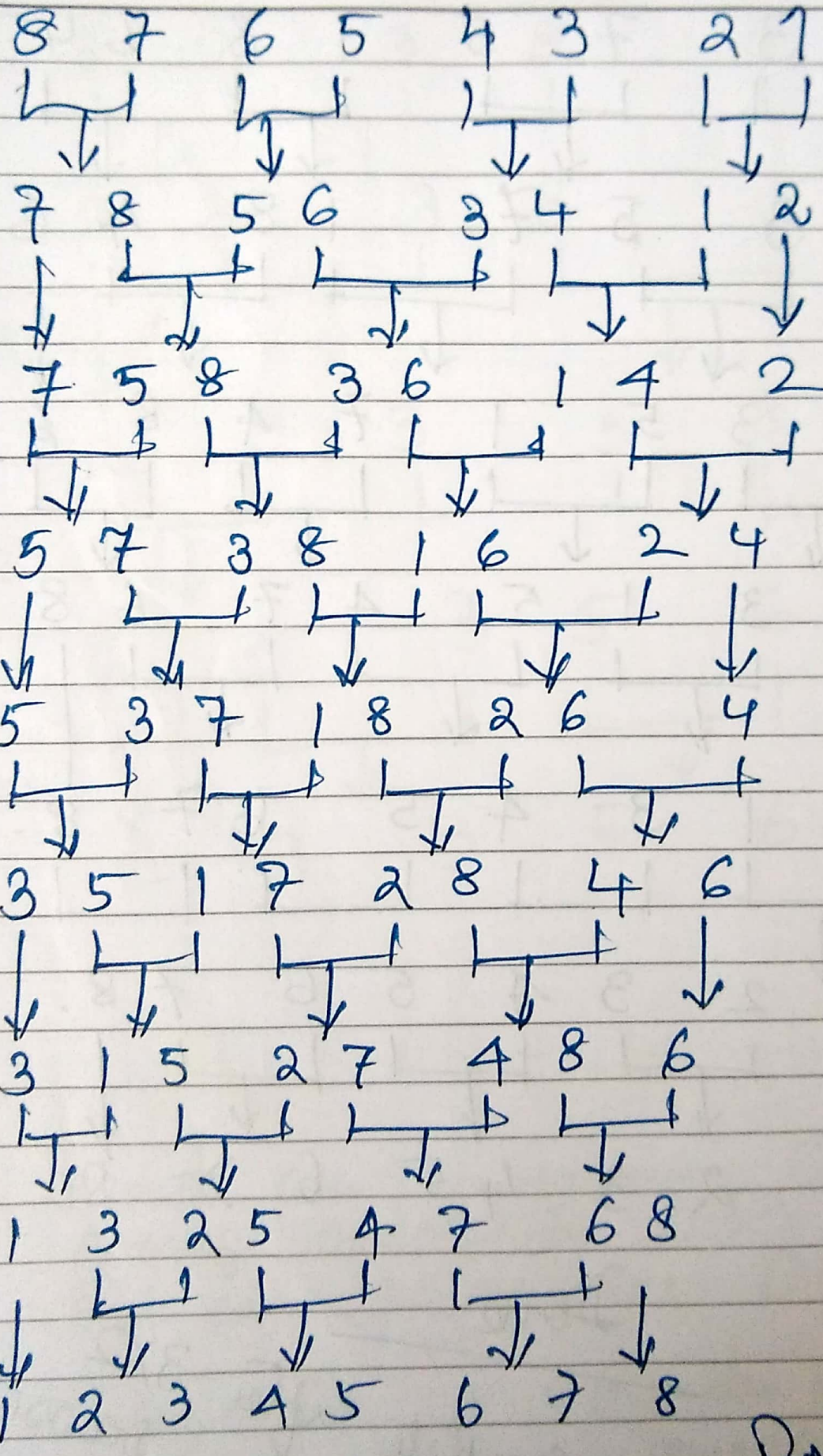
Another example of odd-even sort



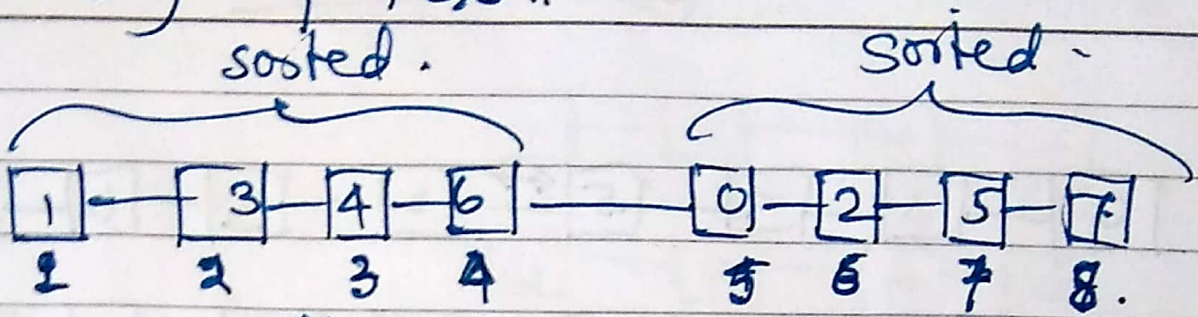
Done

complete after 3X2 passes

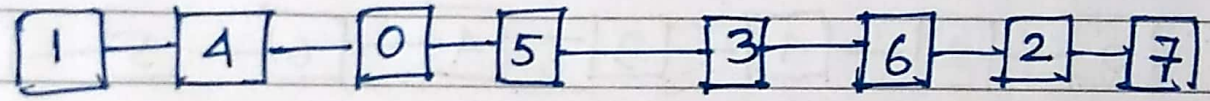
Woost case example of odd even sort



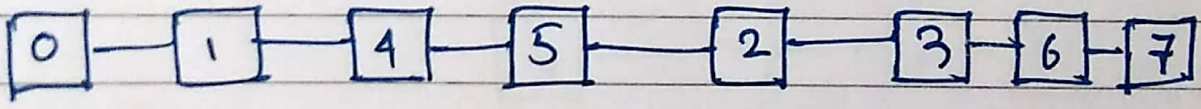
Batcher's odd-even merge of two sorted sequences on a linear array of PEs..



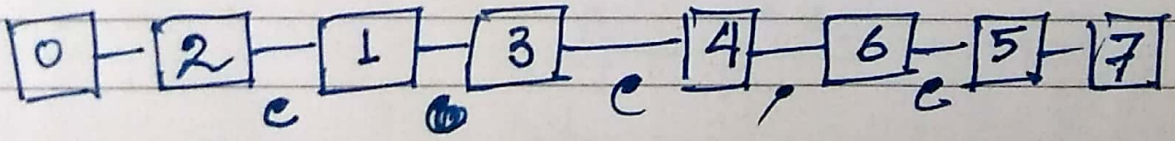
L.1 Unshuffle: Odd indexed elements to left, evens to right ↓



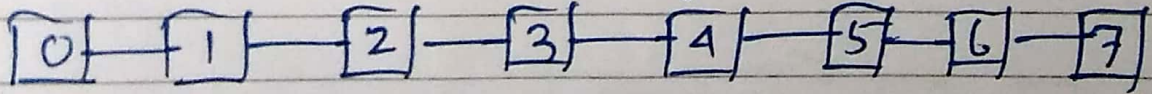
L.2 Merge the "odd sequences" and the "even sequences" ↓



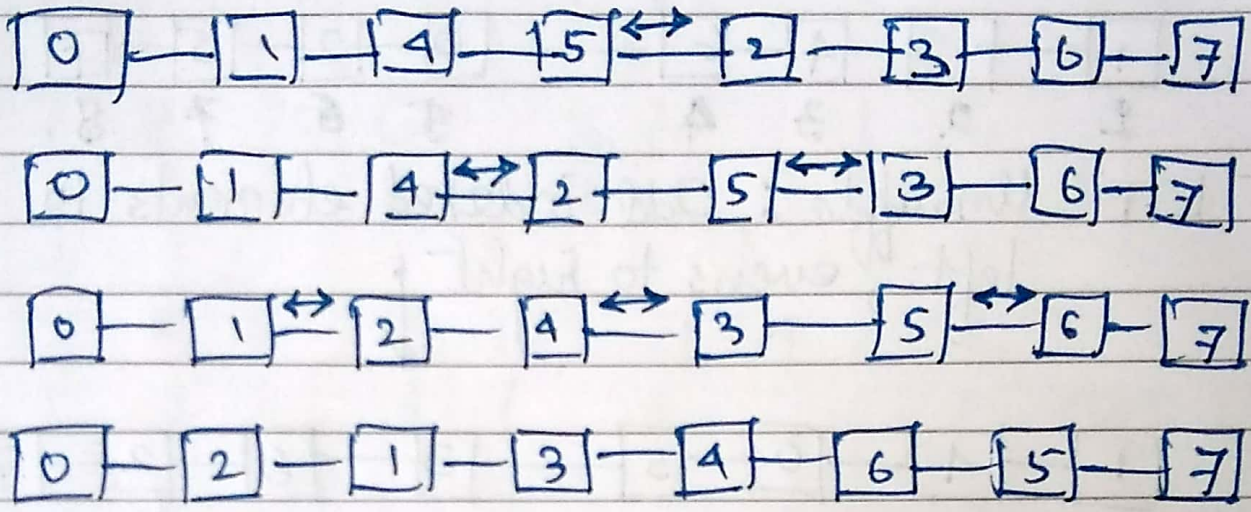
L.3 Shuffle ↓



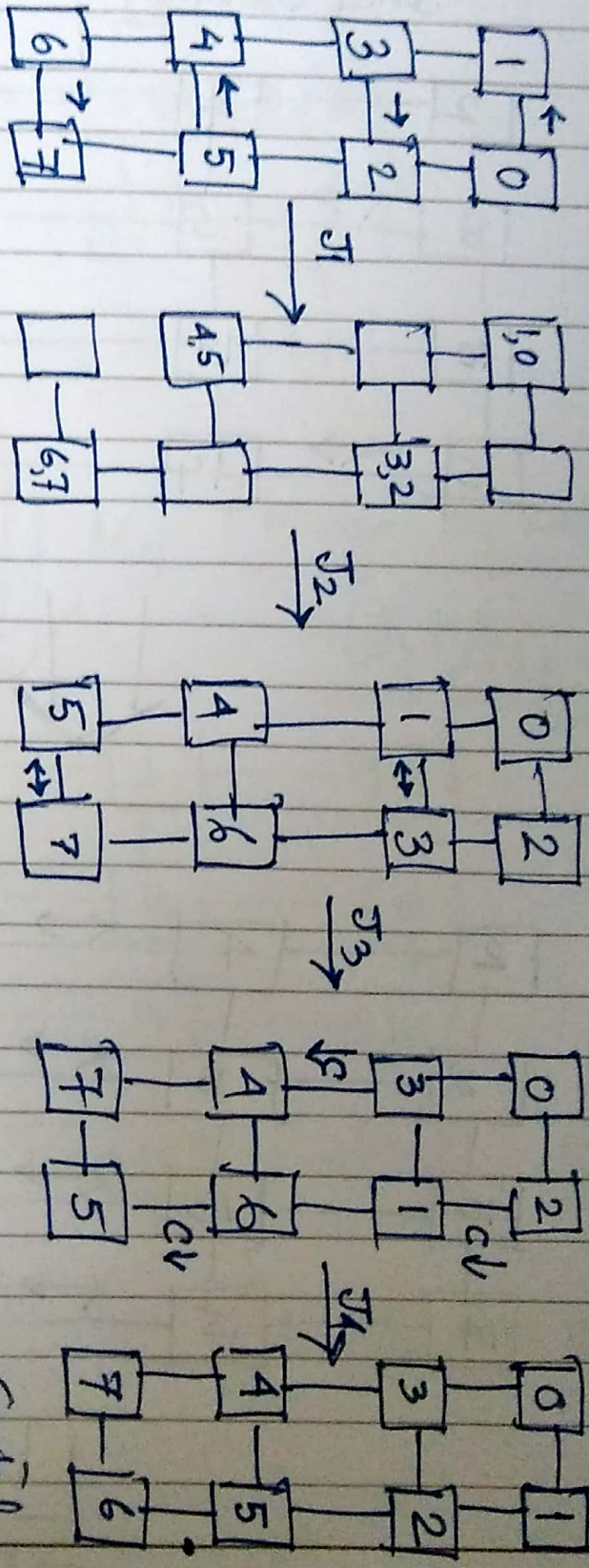
L.4 Comparison-interchange



Implementation of a perfect Shuffle by a sequence of interchange operations

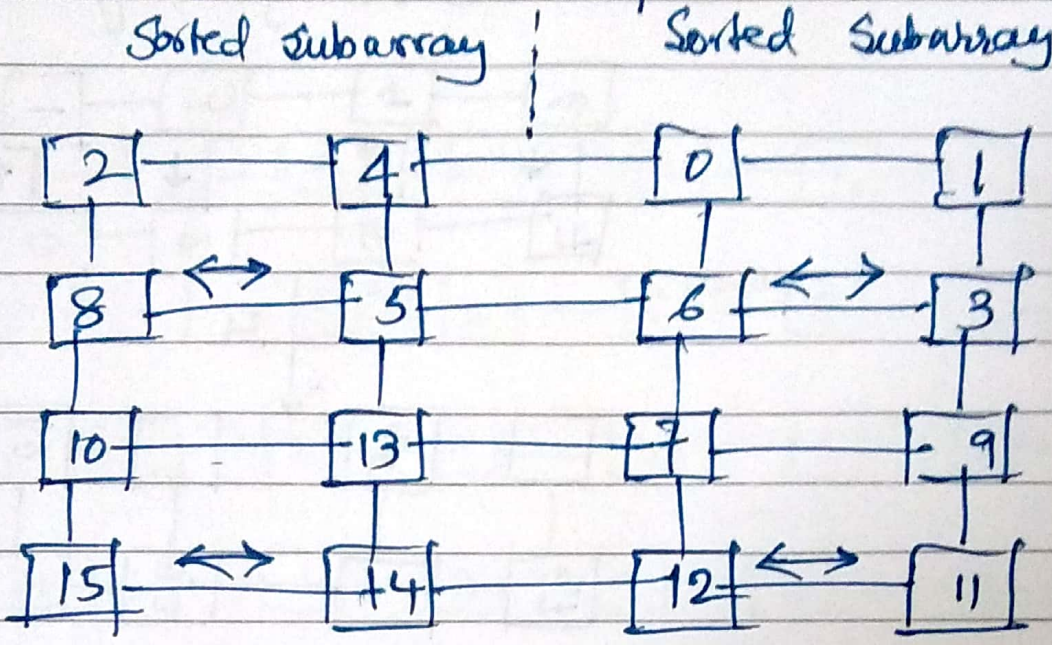


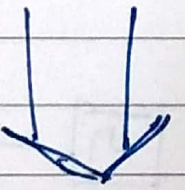
Data Routing, Comparision and interchange operations performed in the M(4,2) sorting algorithm



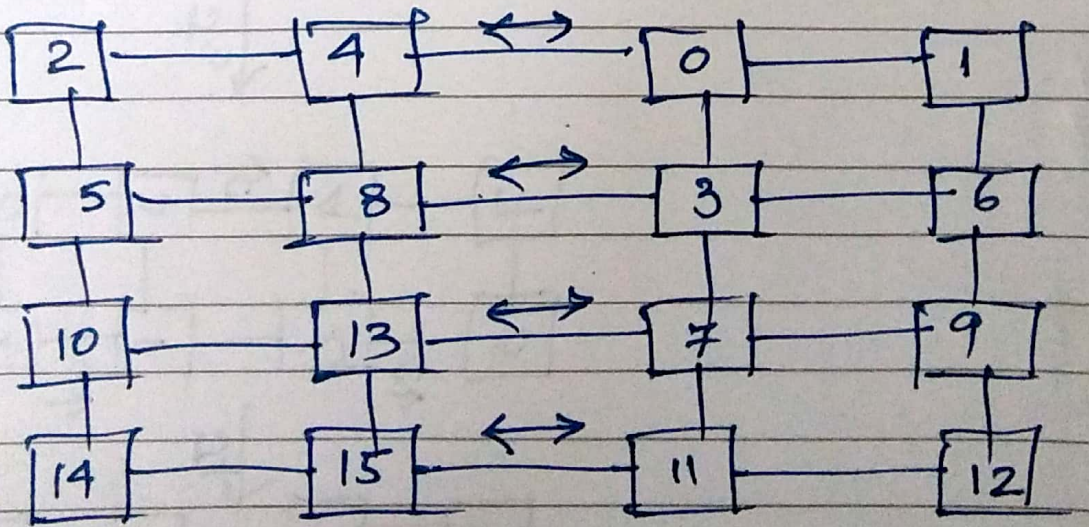
M(4,4) Sorting Algorithm

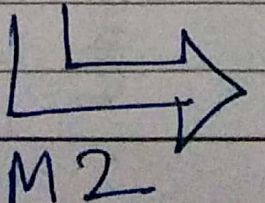
Example.



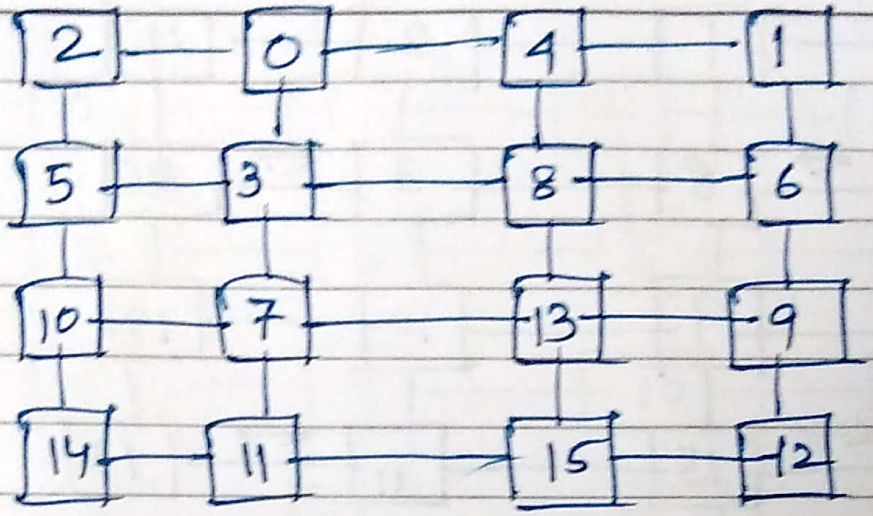


M1 ($j > 2$ single interchange on even rows)

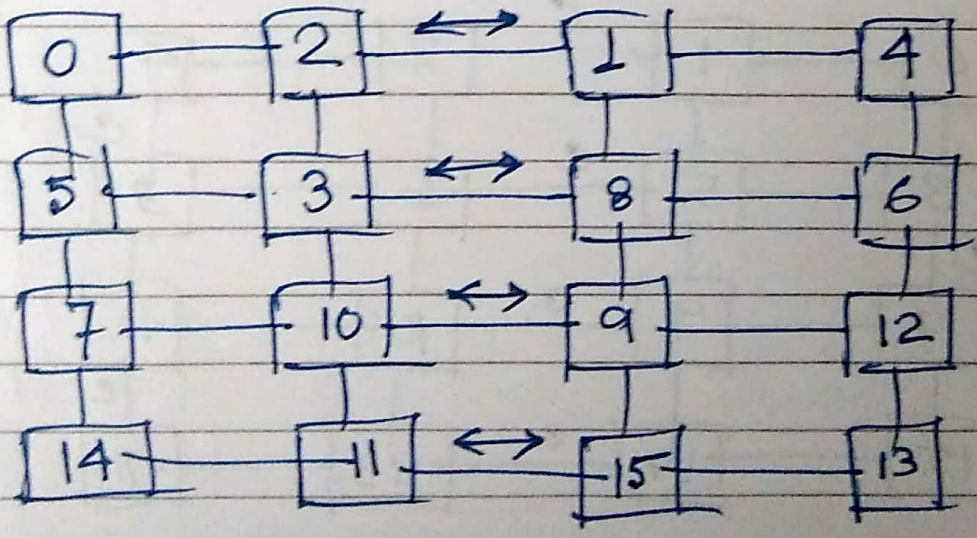




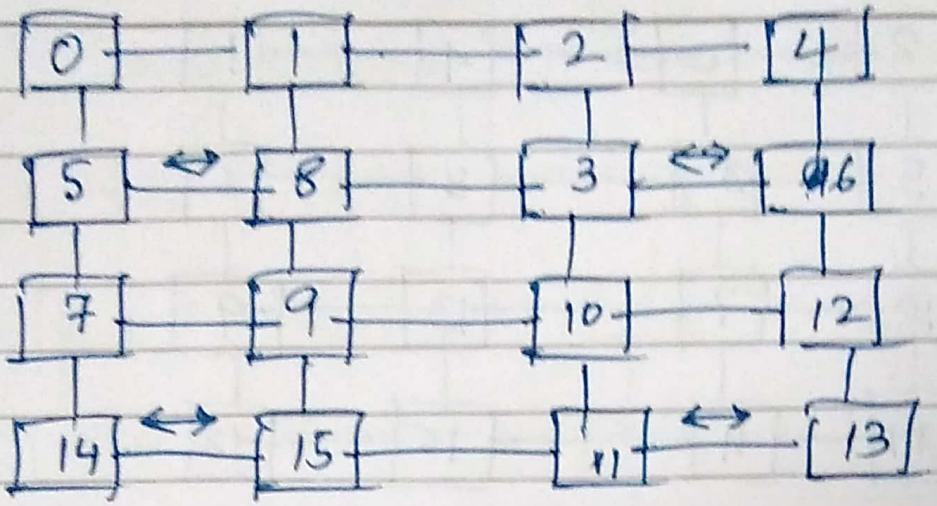
M2 (unshuffle each row)



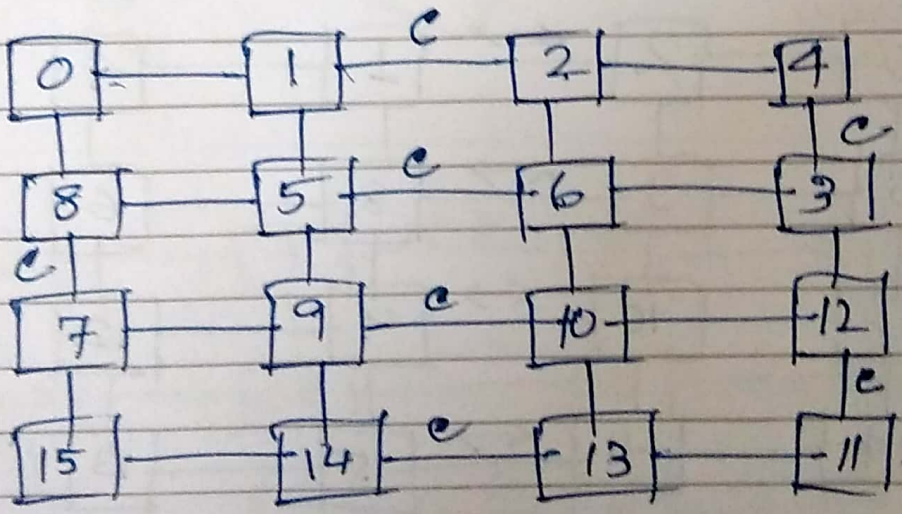
M_3 (Merge by calling $M(j, k/2)$ on each half)



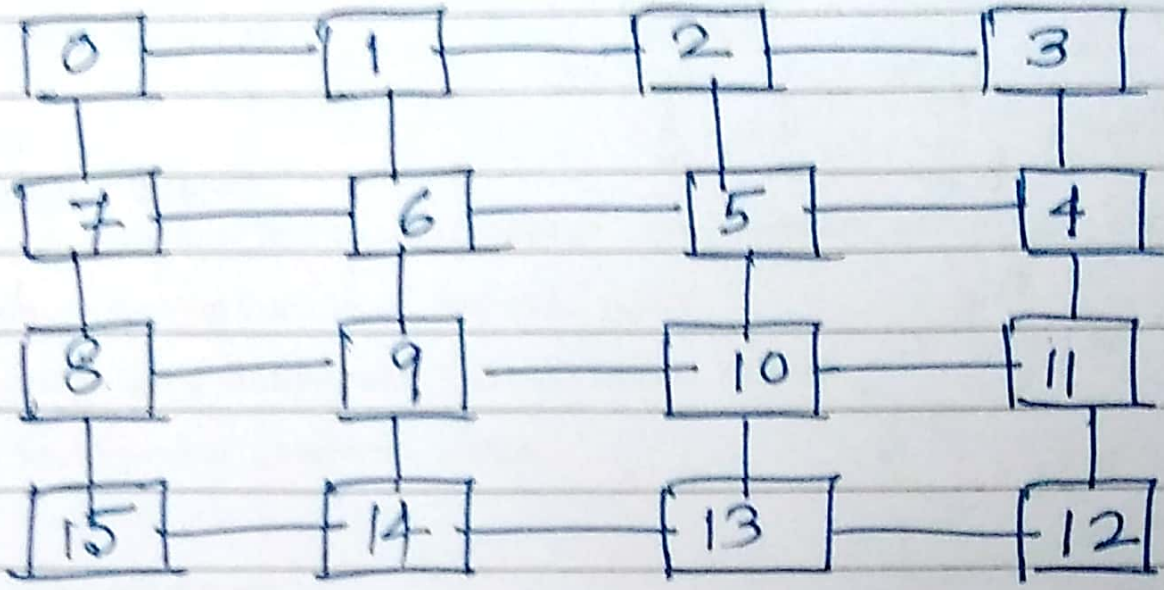
M_4 (Shuffle each row)



M_5 Interchange on even rows.

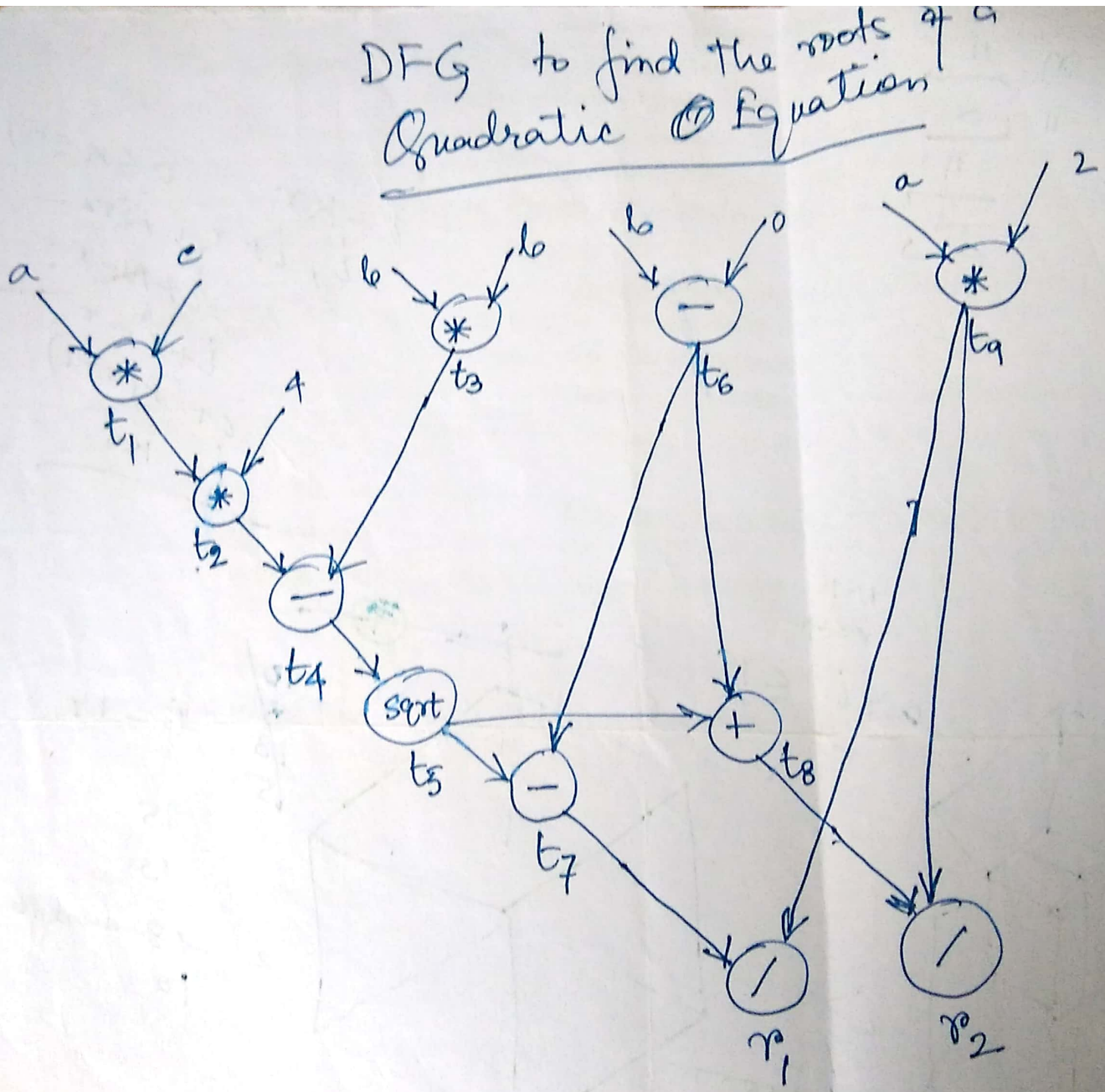


M_6 Comparison-interchange of adjacent elements (every even with the next odd)

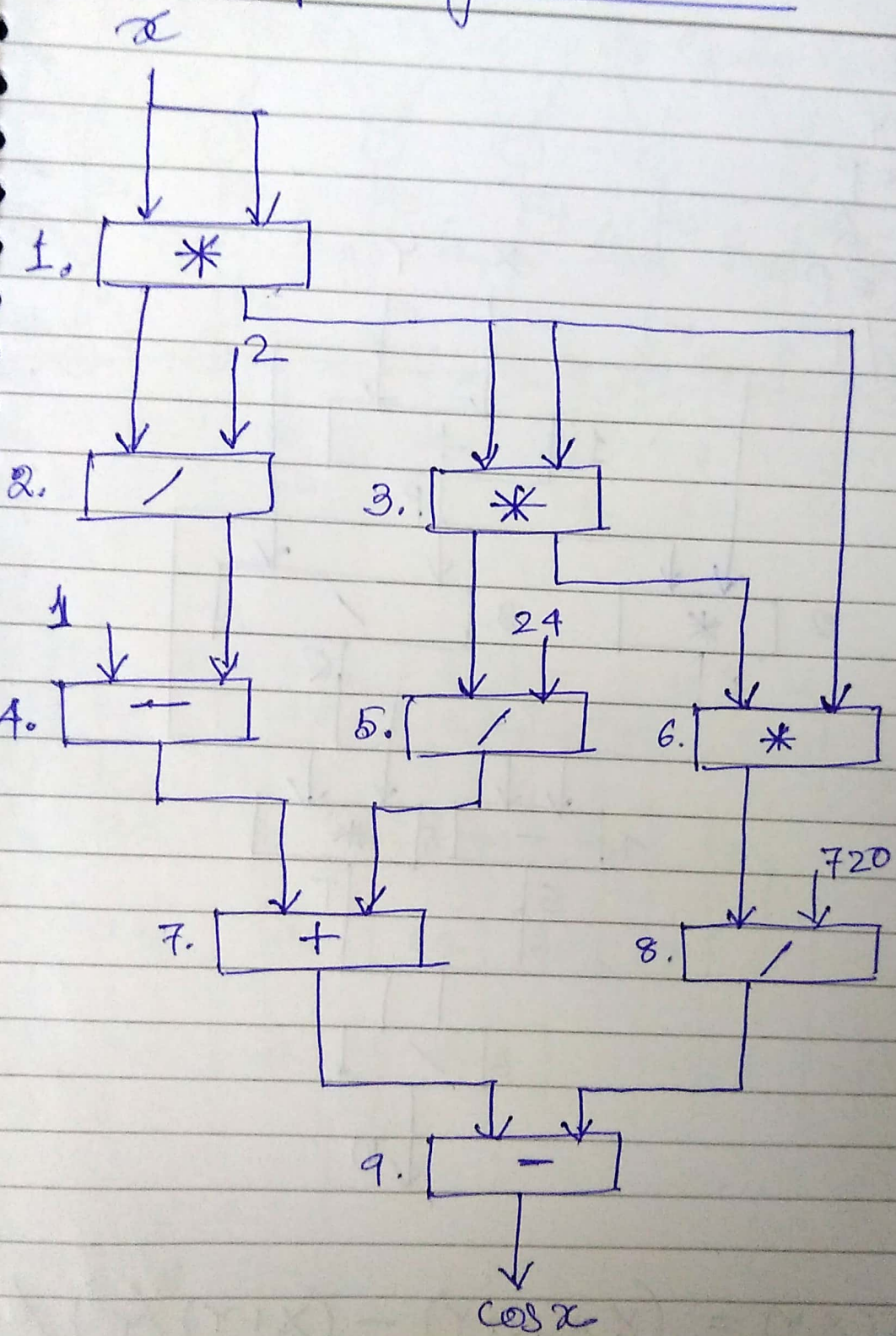


(SORTED)

DFG to find the roots of a Quadratic Equation



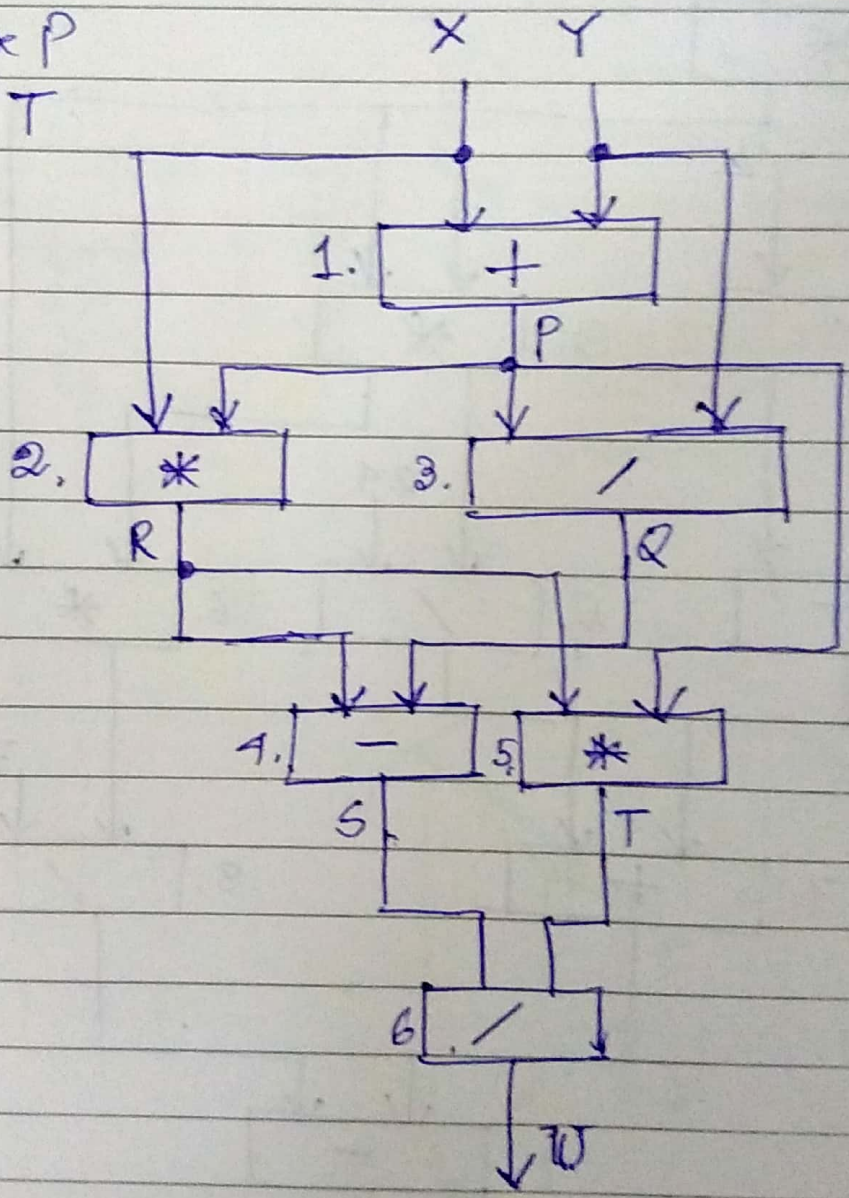
Data Flow Graph for Computing Cos x



$$\cos x \simeq 1 - \frac{x^2}{2} + \frac{x^4}{24} - \frac{x^6}{720}$$

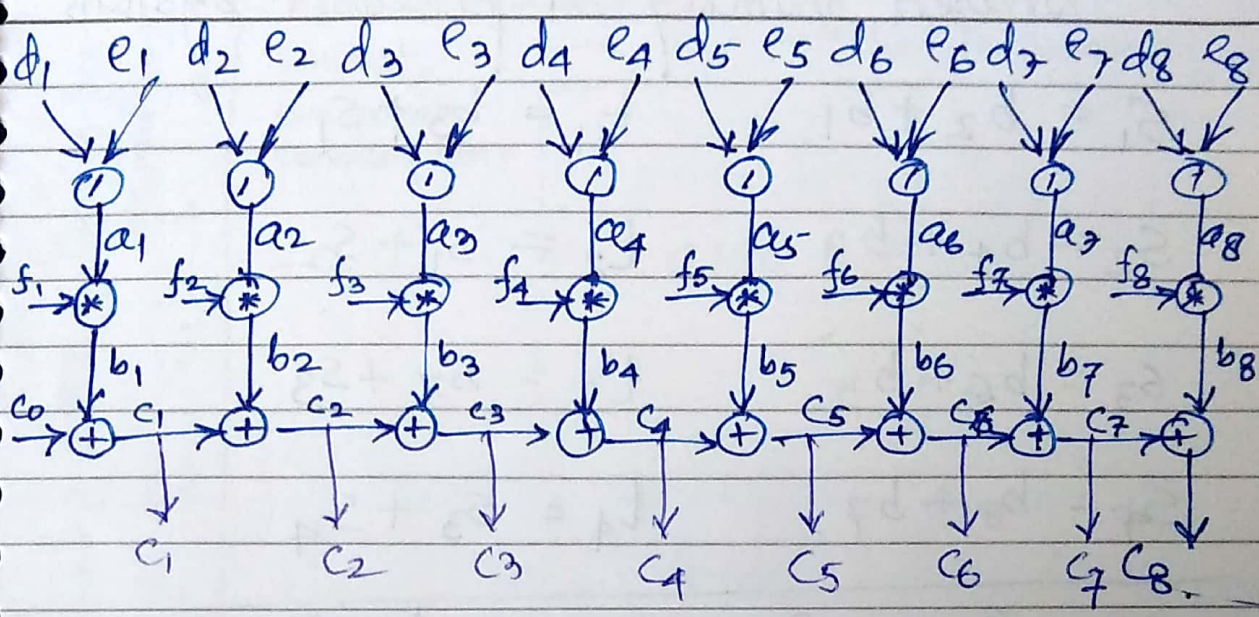
DFG for a set of instructions

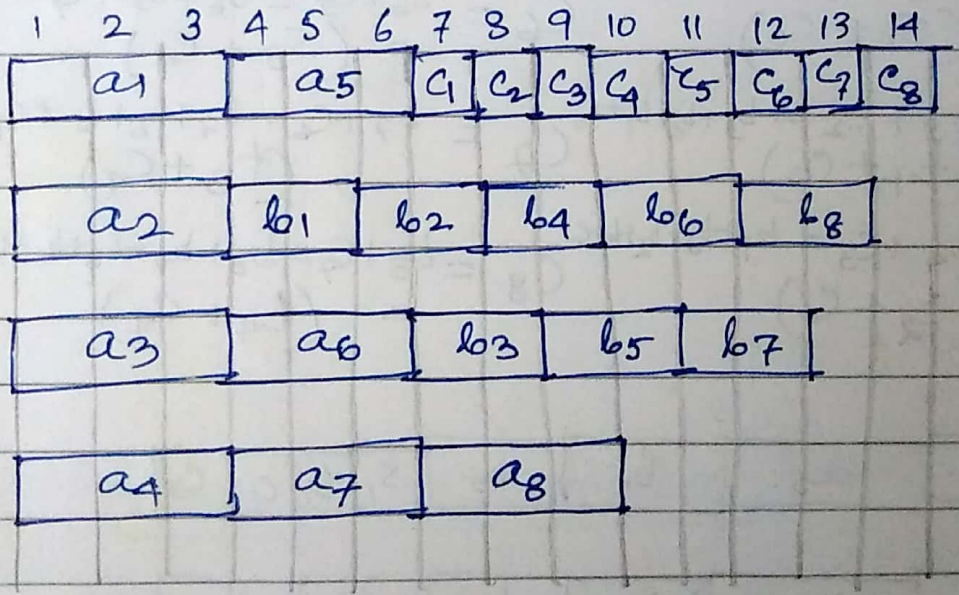
1. P = X + Y
2. Q = P / Y
3. R = X * P
4. S = R - Q
5. T = R * P
6. U = S / T



$$U = f(X,Y) = \frac{(X*(X+Y) - (X+Y)/Y)}{(X*(X+Y)*(X+Y))}$$

DFG for the program





Parallel execution on a shared memory 4-processor system

$S_1 = b_2 + b_1$ $\quad t_1 = b_3 + S_1$

$S_2 = b_4 + b_3$ $\quad t_2 = S_1 + S_2$

$S_3 = b_6 + b_5$ $\quad t_3 = b_7 + S_3$

$S_4 = b_8 + b_7$. $\quad t_4 = S_3 + S_4$

$C_1 = (b_1 + C_0)$ $\quad C_5 = b_5 + C_4 = (b_5 + C_4)$

$C_2 = b_2 + C_1 = b_2 + b_1 + C_0$ $\quad C_6 = b_6 + C_5 = b_6 + b_5 + C_4$

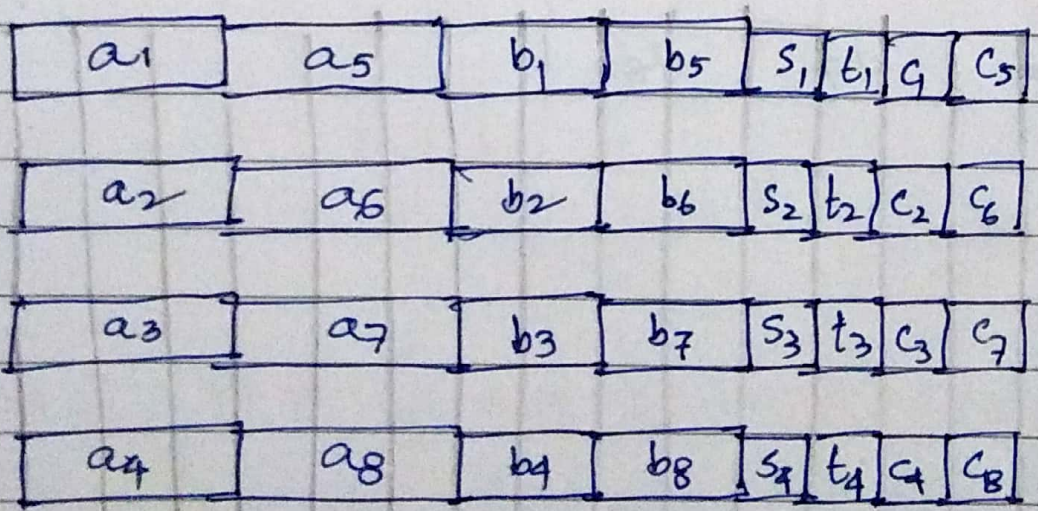
$(S_1 + C_0)$ $\quad = (S_3 + C_4)$

$C_3 = b_3 + C_2 = b_3 + b_2 + b_1 + C_0$ $\quad C_7 = b_7 + C_6 = b_7 + b_6 + b_5 + C_4$

$(t_1 + C_0)$ $\quad (t_3 + C_4)$

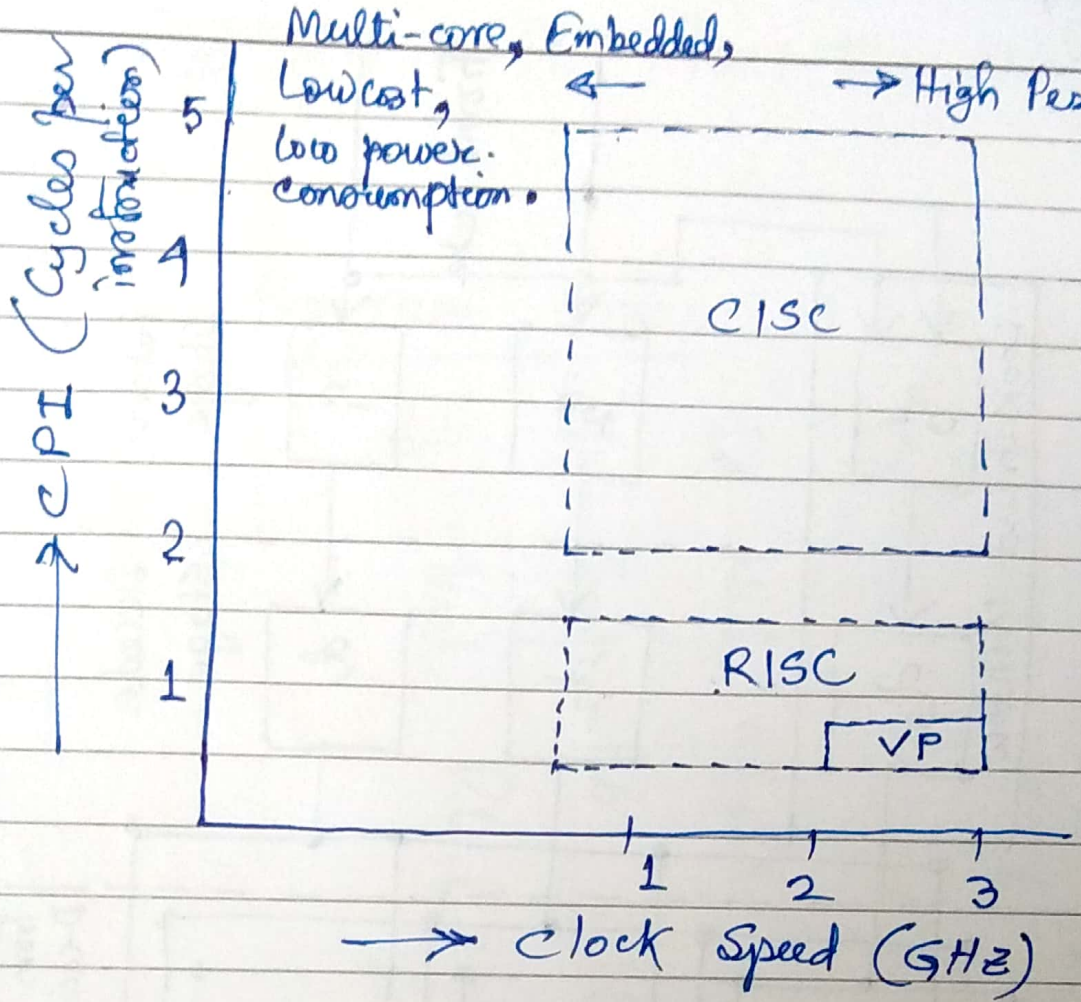
$C_4 = b_4 + C_3 = b_4 + b_3 + b_2 + b_1 + C_0$ $\quad C_8 = b_8 + C_7 = b_8 + b_7 + b_6 + b_5 + C_4$

$(t_2 + C_0)$ $\quad = (t_4 + C_4)$



CPU (cycles)

CPI vs Processor clock Speed



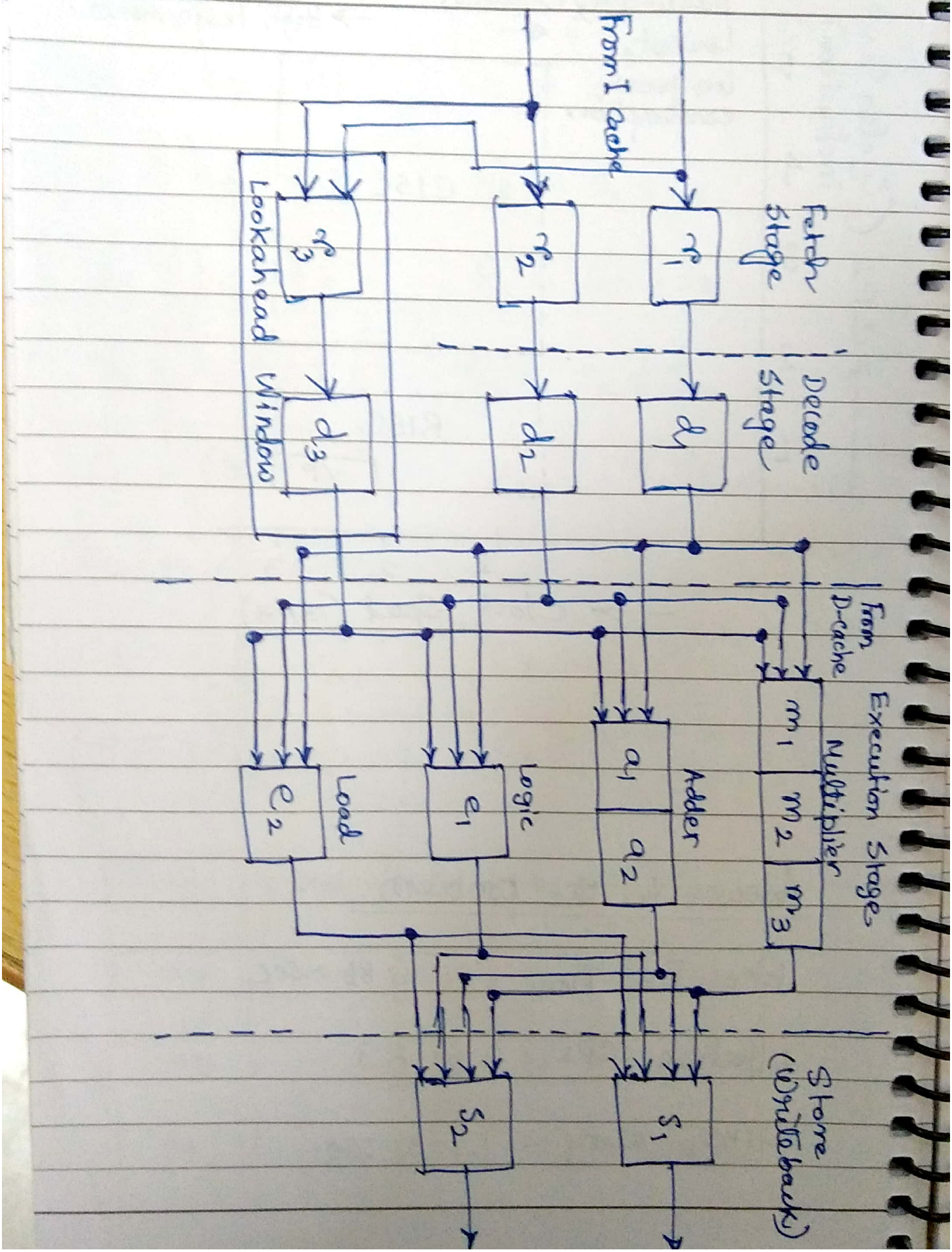
Answer to the problem

Execution time : 6.86 msec

Effective CPI : 2.3

MIPS rating : 21.72

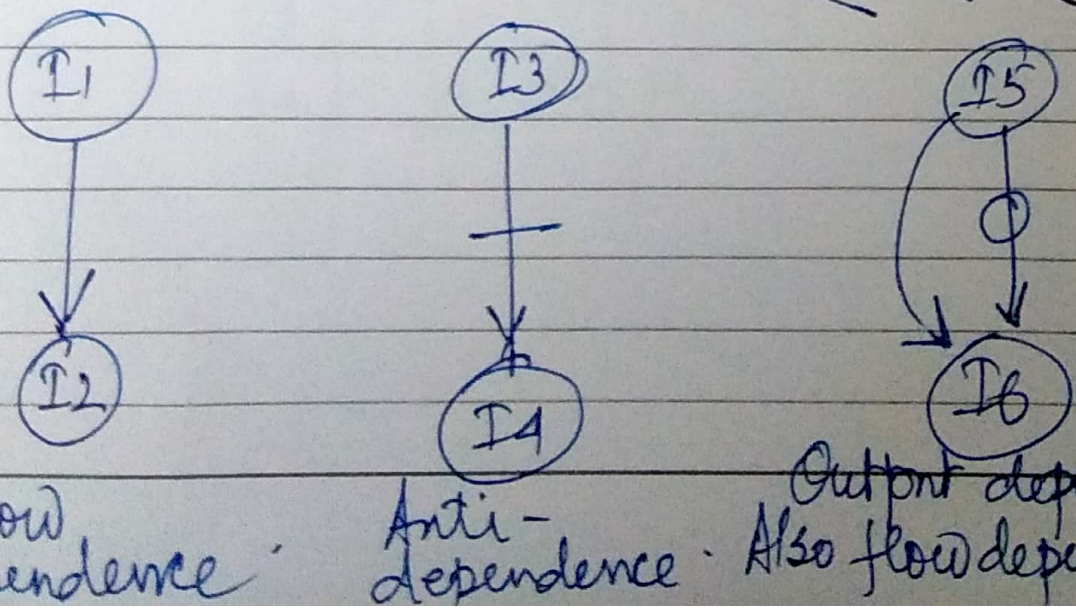
A dual-pipeline, superscalar processor with 4 functional units and a lookahead window.



A sample program and its dependence graph. I2 and I3 share the adder and ~~I4 & I6 share the multiplier.~~

Program Order

I1 Load R1, A / R1 ← Memory(A) /
I2 Add R2, R1 / R2 ← (R2) + (R1) /
I3 Add R3, R4 / R3 ← (R3) + (R4) /
I4 Mul R4, R5 / R4 ← (R4) * (R5) /
I5 Comp R6 / R6 ← (R6)' /
I6 Mul R6, R7 / R6 ← (R6) * (R7) /



Flow dependence. Anti-dependence. Output dependence. Also flow dependence

From Execution Stage

Examples of Pipeline Stalling

Fig. (a) No data dependences.

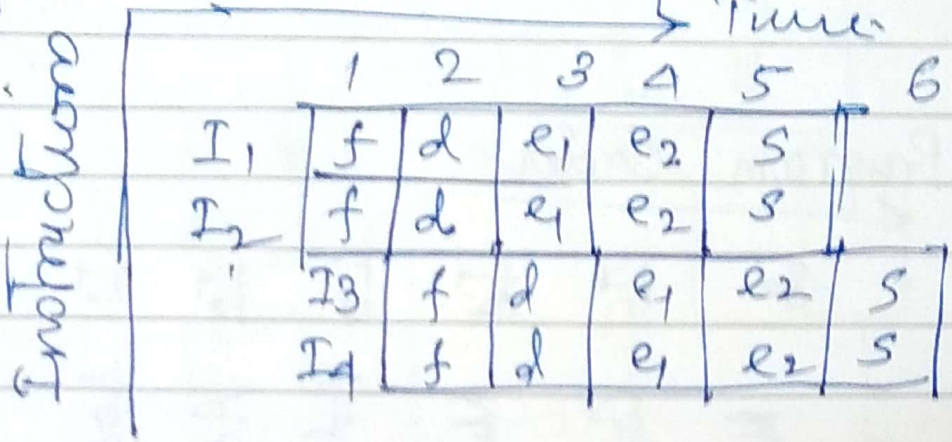


Fig (b) I2 uses data generated by I1

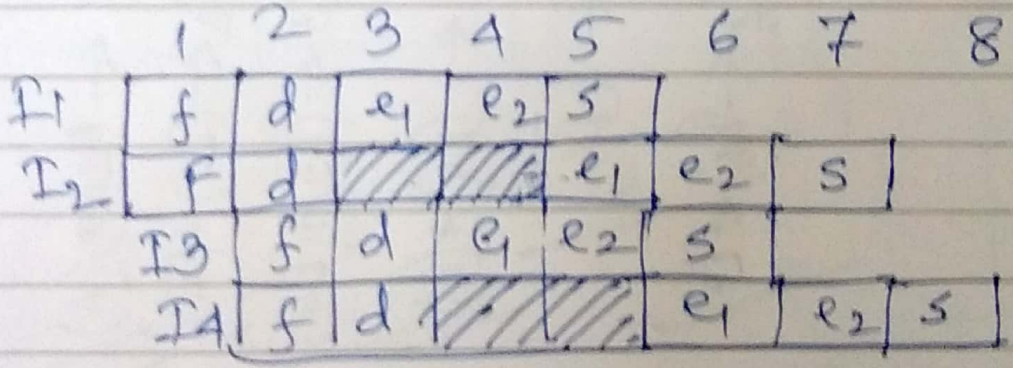


Fig (c) Branch instruction I2 causes a delay slot of length 4 in both pipelines

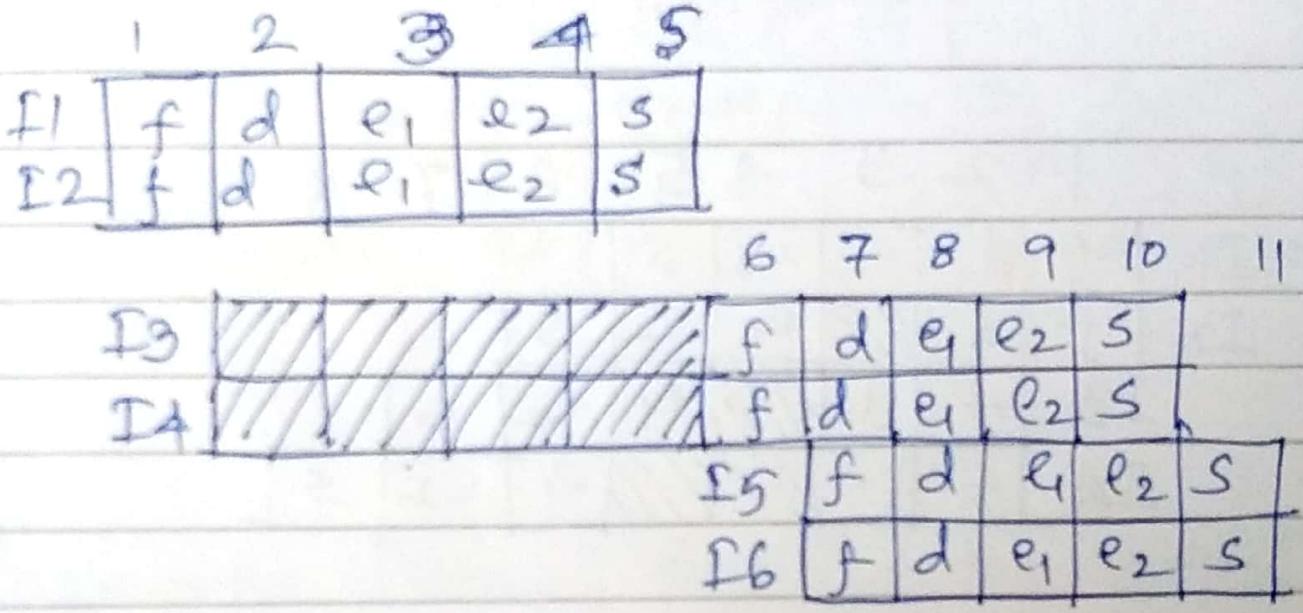


Fig (d) No Resource Conflicts.

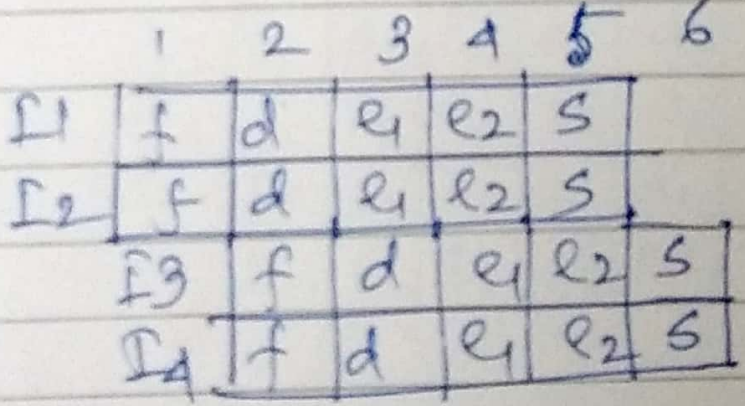
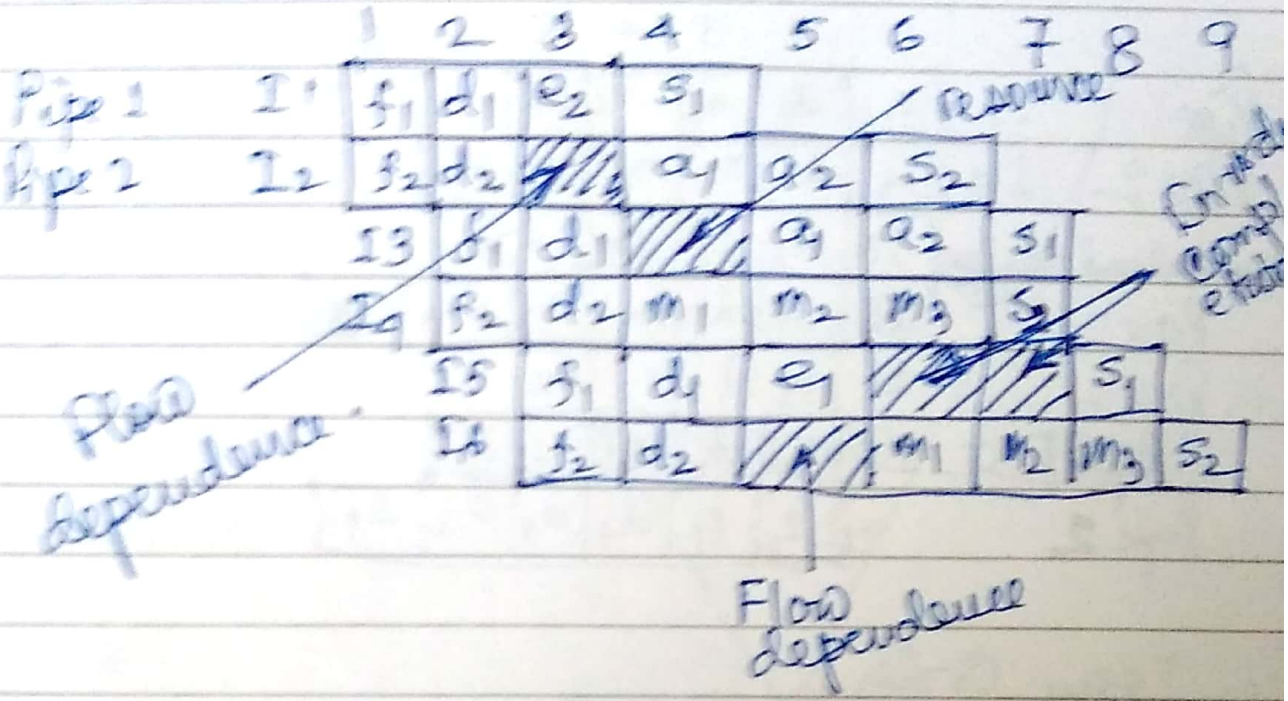
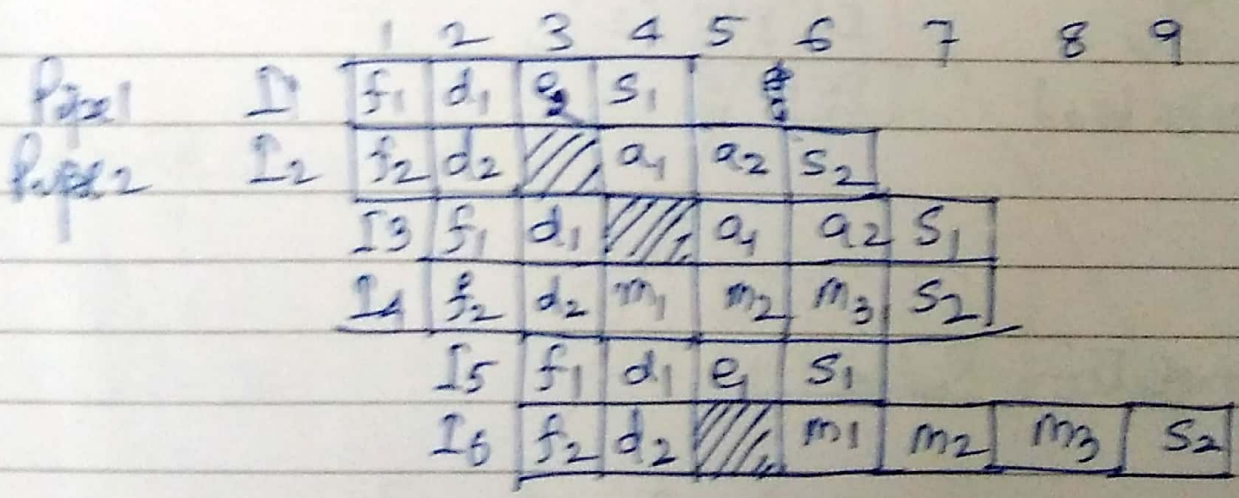


Fig (e) I_1 and I_2 conflict in using the same functional unit, and I_4 uses data generated by I_2.

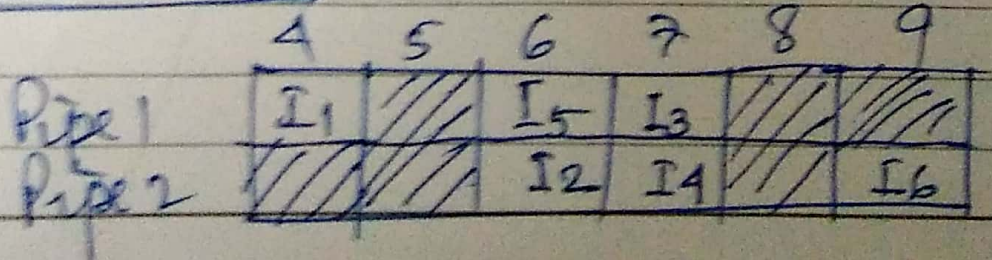
In-order issue with in-order completion.



In-order issue, Out-of-order completion.



Completion Order



Out of order issue, out-of-order completion

		1	2	3	4	5	6	7
Pipe 1	I_3	f_1	d_1	a_1	a_2	s_1		
Pipe 2	I_4	f_2	d_2	m_1	m_2	m_3	s_2	
Look Ahead.	I_5	f_3	d_3	e_1	s_1	(Pipe 1)		
Pipe 1	I_6		f_1	d_1	m_1	m_2	m_3	s_2
Pipe 2	I_1		f_2	d_2	e_2	s_2		
Pipe 1	I_2			f_1	d_1	a_1	a_2	s_1

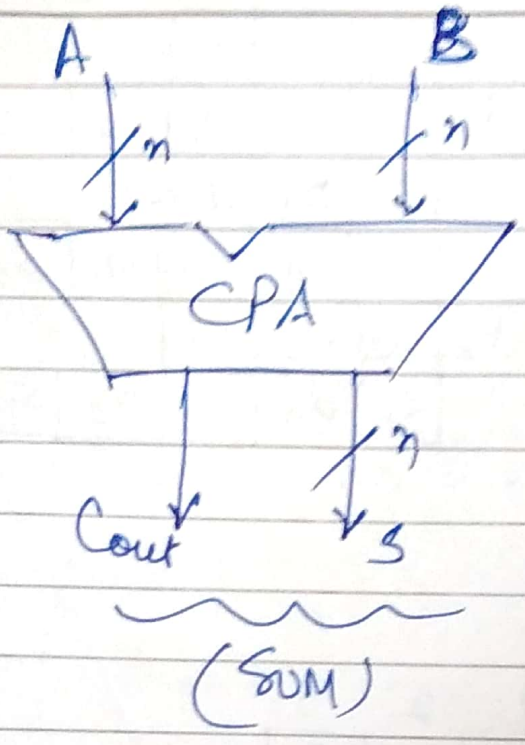
Issue order

	1	2	3
Pipe 1	I_3	I_6	I_2
Pipe 2	I_4	I_1	
Look Ahead.	I_5		

Completion Order

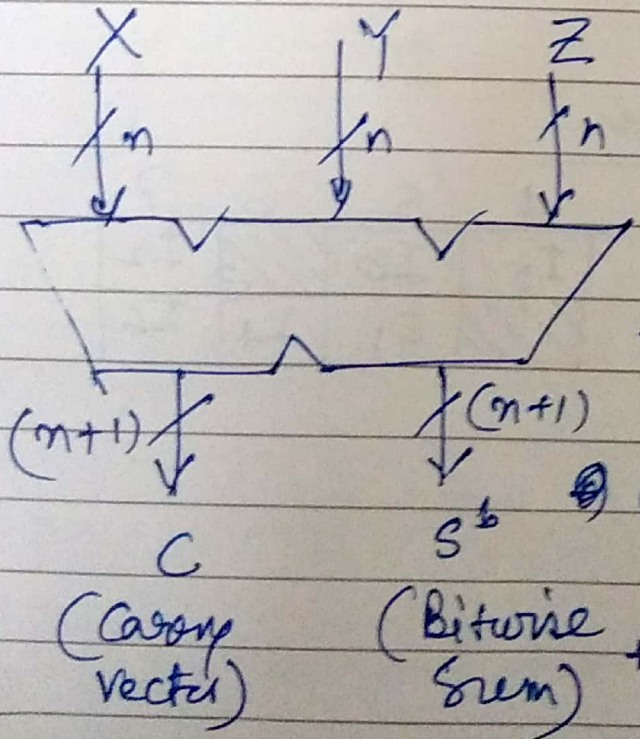
	4	5	6	7
Pipe 1	I_5	I_3		I_2
Pipe 2		I_1	I_4	I_6

Carry Propagation Adder



A = 1011
+ B = 0111
S = 10010

Carry Save Adder



X = 001011
B = 0101 01
C = 1111 01

S^b = 01 000 11

~~C = 0 1 1 1 1 0 1~~

0000000

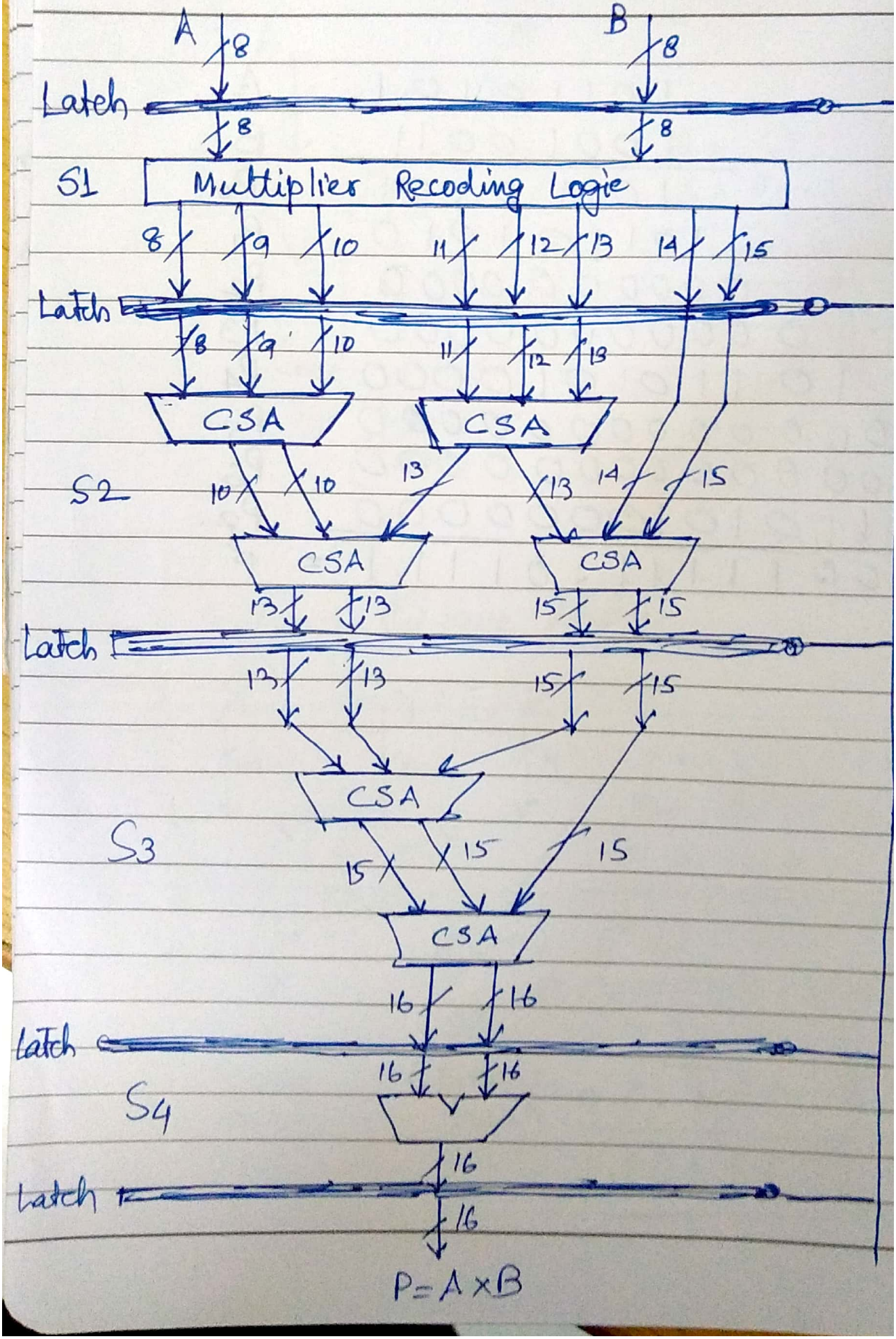
+ 0 1 1 1 0 1 0

1 0 1 1 1 0 1

Example of multiplication

10110101	A
10010011	B
10110101	P_0
1011010100	P_1
0000000000	P_2
0000000000000	P_3
1011010100000	P_4
0000000000000000	P_5
0000000000000000	P_6
1011010100000000	P_7
1100111111011111	= P

A pipeline unit for fixed point multiplication of 8-bit integers

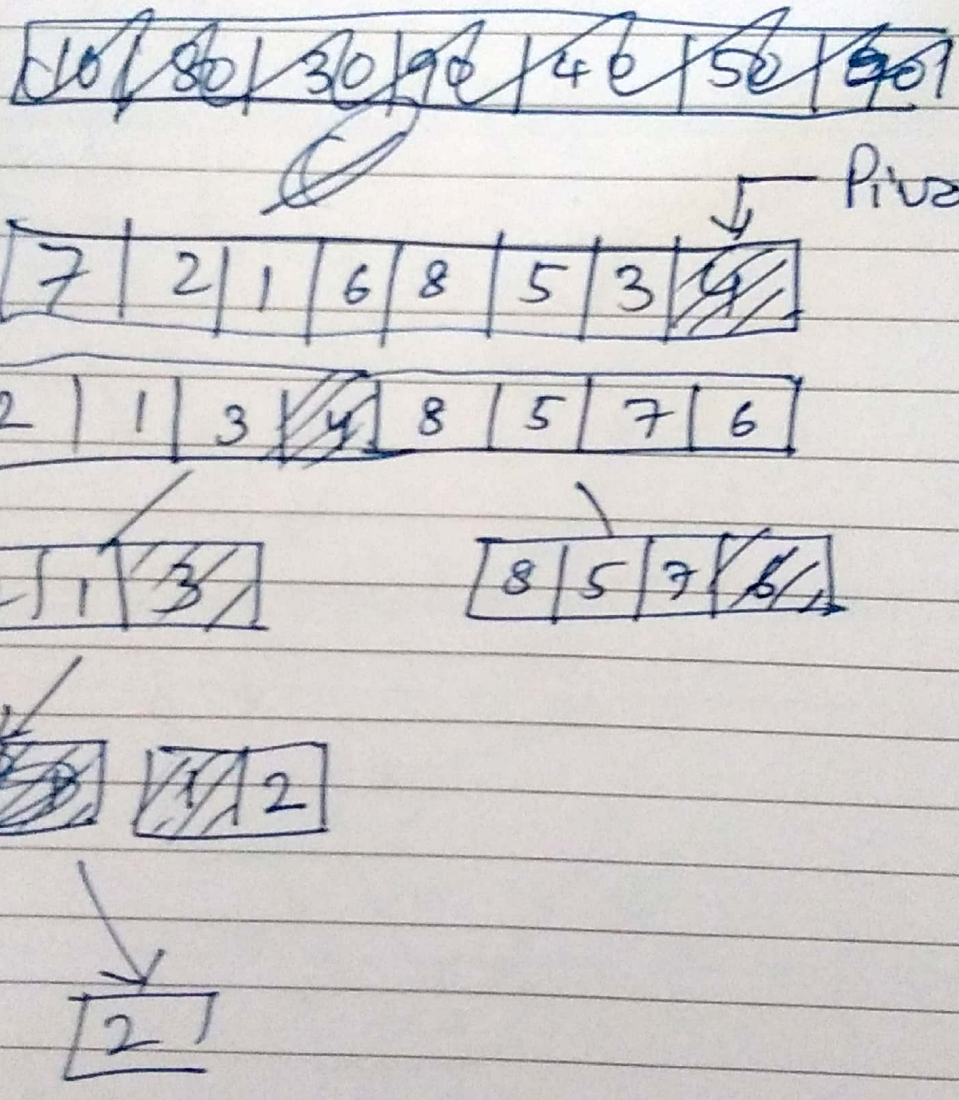


Quick Sort

Pivot (last element may be)

Left Half → Smaller than Pivot

Right half → greater than Pivot

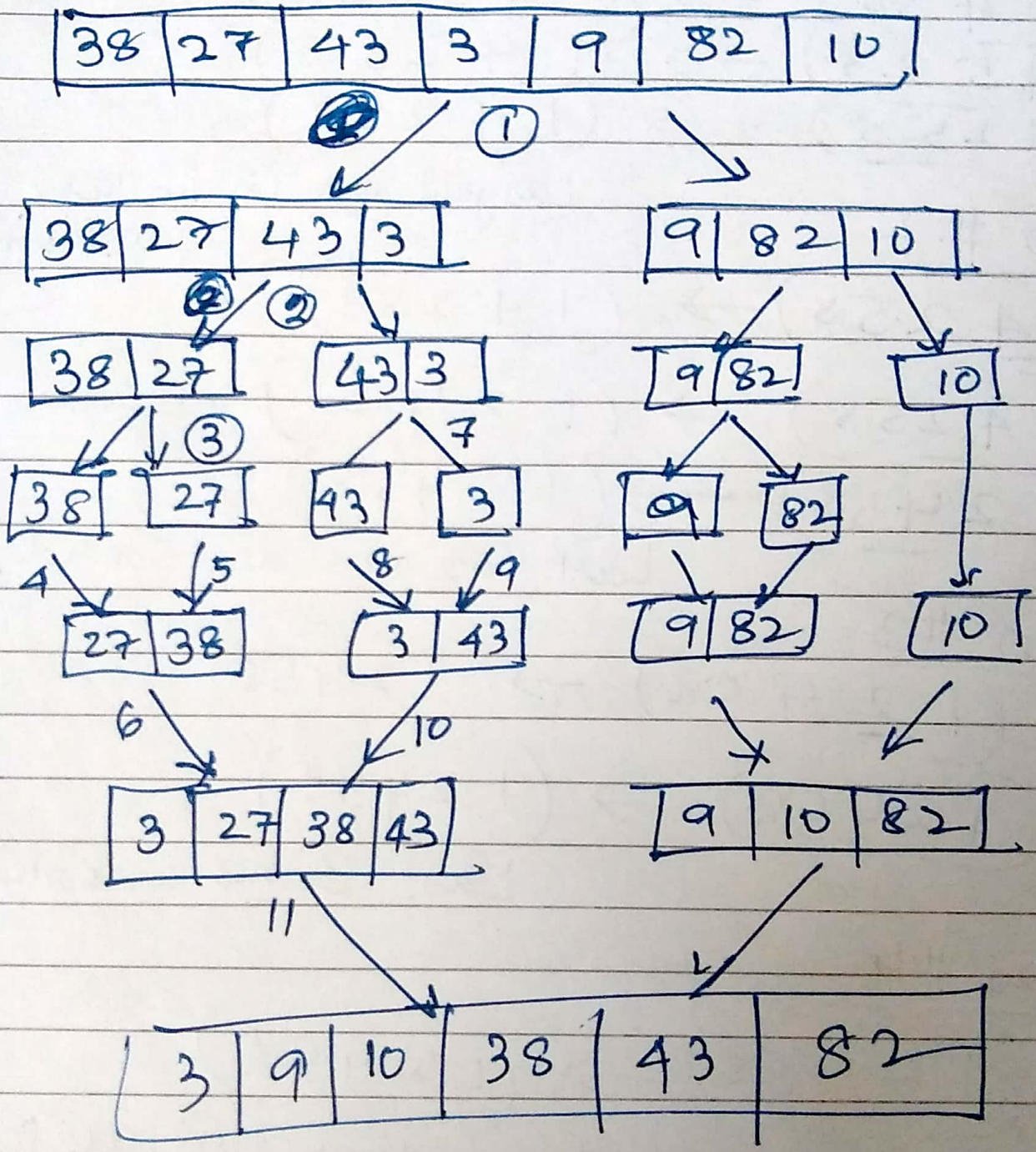


$O(n \log n)$ Av. case

$O(n^2)$ Worst case

Merge Sort

Divide and Conquer



$O(n \log n)$ in worst case

Bubble Sort

Pass #1

(5 1 4 2 8) → (1 5 4 2 8)
(1 5 4 2 8) → (1 4 5 2 8)
(1 4 5 2 8) → (1 4 2 5 8)
(1 4 2 5 8) → (1 4 2 5 8)

Largest no in the last position.

Pass #2

(1 4 2 5 8) → (1 4 2 5 8)
(1 4 2 5 8) → (1 2 4 5 8)
(1 2 4 5 8) → (1 2 4 5 8)

Last two nos are sorted.

Pass #3

(1 2 4 5 8) → (1 2 4 5 8)
(1 2 4 5 8) → (1 2 4 5 8)

Last 3 nos are sorted

Pass #4

(1 2 4 5 8) → (1 2 4 5 8)

Last four nos are sorted

∴ Sorting Done.

$O(n^2)$ in average case.