DESIGNING AN ALU:-

⇒ So, from where are you start designing an ALU, from multiplicat (or multiplier)

No.

From division (or divider)

No

So, From addition (or adder)

Ok

⇒ Before that you have to understand how to represent numbers if I say multiply 5x9 using Booth's Algorithm

You may take 1001 as 9, and your multiplication will go wrong (01001)

The issue is how to store -ve nos.

⇒ So, first we have to know how to represent a signed number.

⇒ There are 3 proposed ways to represent a signed number.

1. Signed magnitude
2. 1's complement
3. 2's complement

First tell me do you want to design your ALU only for +ve numbers or Signed nos.? OFCOURSE SIGNED Numbers

I am going to explain why your -ve nos are stored in 2's complement form.

Decimal	Sign Magnitude	1's Complement	2's Complement
3	0 1 1	0 1 1	0 1 1
2	0 1 0	0 1 0	0 1 0
1	0 0 1	0 0 1	0 0 1
0	0 0 0	0 0 0	0 0 0
-1	1 0 1	1 1 0	1 1 1
-2	1 1 0	1 0 1	1 1 0
-3	1 1 1	1 0 0	1 0 1

As you can observe in all signed representation +ve numbers are stored in same way.
because There is no issues in storing +ve numbers,
However, when we write -ve number on paper they can be represented by a '—' sign. but '—' minus cannot be stored in memory.
So, we require same way to store -ve numbers.

In sign magnitude, your MSB for +ve numbers is zero
For -ve numbers, you just make your MSB one.

Now, out of 8 combinations, which combination is left.
100 this number represents — 0.
But there is nothing like -zero.
So, this representation has one rubbish value.
This is the reason why signed magnitude representation failed.

So, if you add 2 numbers
$$\begin{array}{r} {\scriptstyle 1\ 1\ \ \ } \\ 011 \\ 001 \\ \hline 100 \end{array}$$
Can you guarantee that this number is 0 + 4 or — 0?
No, ∴ this system fails.

Now, as signed magnitude representation failed lets discuss another alternate representation.
+ve nos. will again be represented by having 0 as its MSB
For -ve nos. don't just store MSB as 1,
take 1's complement of that number.

What this representation says if you have a 3-bit number look at its MSB —|— —
if MSB is zero no. is +ve
take it as it is
if MSB is one no. is -ve
if you want to know what no. it is don't to take it on face value, take its one's complement.

Ex:- So I have this no 011 stored in memory.

as MSB = 0 no. is +ve take it as its

I have another no. 100

as MSB = 1 no. is −ve, now don't take it on face value. find its one's complement to understand its value.

1's complement of 100 is 011 this no. is −3

Now consider another no. 111

as MSB = 1 no. is −ve, to find its actual value find its 1's complement

1's complement of 111 is 000.

Wrong. there can be no no. as −0.

So, this system also fails.

Ex:- we are adding two numbers

$$\begin{array}{r} 011 \\ 100 \\ \hline 111 \end{array}$$

now once this no. is stored in memory after some time when I will retrieve it should it represent +7 or −0?

+7 of course ∴ This system fails.

Now, lets consider another number system, 2's complement

* Now, what is 2's complement ⇒ 1's complement + (plus) one

There is a shortcut to use 1's complement that works in every case.

⇒ From R.H.S. (right hand side) copy the no. as it is till 1st 1 is encountered after that just invert the digits

⇒ So, how 000 will be represented in 2's complent.

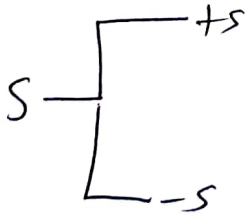
you will get 000

So, now you have a single (and same) representation for ~~00~~ zero in both Orignal and its 2's complement form.

So, whenever 000 is stored in memory and readd out after some time it will still represent zero.

Therefore, hence proved negative nos are stored in its 2's complement form.

Q.



101 is this +5 NO,

MSB is 1

0101 is +5

00101 is +5

So, if someone ask you to multiply 7*5 using Booth's Algorithm and you use 101 as 5, you will get incorrect answe

Designing An ALU (Adder Circuit):-

```
  3    0011
+ 2    0010
-------------
  5    0101
```

To perform this addition I require how many bits adder?

4-bit Adder.

Now, if I had to perform addition of two 8 bit nos. 100, 125 what would I do?

I have to design an 8-bit adder.

Now, Is the logic of 4-bit addition different from 8-bit addition

No,

So, will the logic of 1-bit addition be different from 4-bit addition?

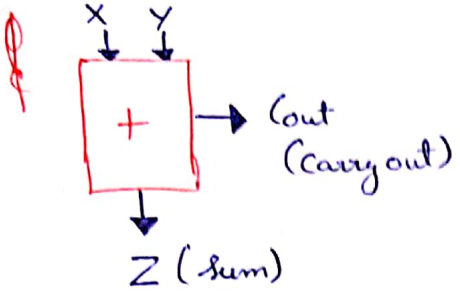
Yes / No Yes (for half ~~bit~~ adder)
No (for full adder)

No,

So, to understand and design 'b' 8-bits adder I should first understand how to ~~add~~ ~~do~~ design 1-bit adder.

Half-adder:-

Now, consider the following circuit.



Now, when will my sum be 1,

Q. when both of my inputs X & Y ~~=0~~ are zero?

No, 0+0 = 0

Q. So, when both of my inputs X & Y are one?

No. 1+1 = 0 (Sum), 1 (carry)

Q. So, then when will my sum becomes ~~zero~~ one?

when exactly one of my input is one.

i.e. ~~when~~

My Sum Z = 1, if $X \cdot \overline{Y} + \overline{X} \cdot Y$

↳ X is one ~~AND~~ Y is zero.

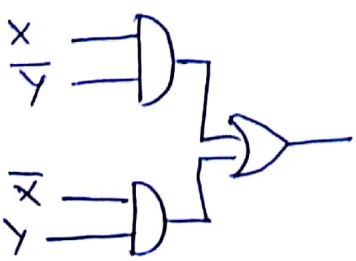
OR

Y is one AND X is zero.

This expression $Z = X \cdot \overline{Y} + \overline{X} \cdot Y$

is my XOR gate.

It can also be represented by a circuit.



SIMILARLY,

HALF - ADDER (Continued)

Similarly,

When will my carry out bit will be 1

Q. When both of my inputs are zero?

No.

Q. When ~~X=0, Y=1~~ one of my input is zero?

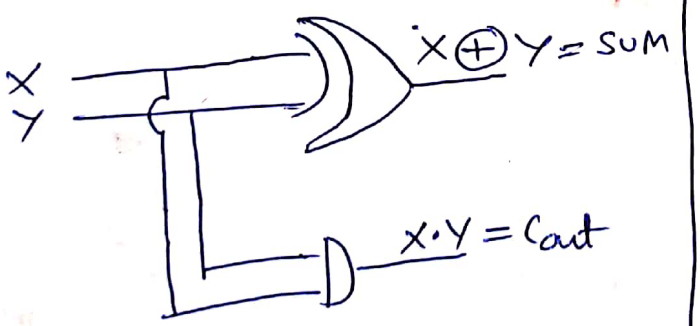
No.

Q. Exactly, My carry out bit will only be one when both of my inputs are ~~zero~~. one

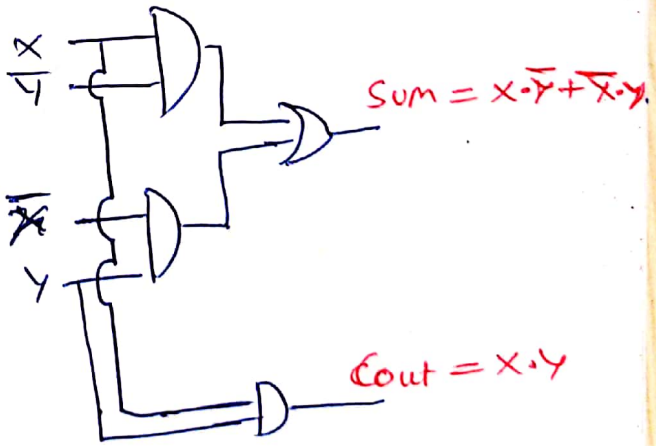
i.e., $C_{out} = X \cdot Y$

X AND Y.

∴ either my adder can be represented as



or



Now, So pens down? done and dusted? Finished learned A.I. ☺

No, The task is not finished, let me tell you why?

What was my task my task was to design a 1-bit adder that can be used to perform '6'-bit addition in combination with other adders.

Now, take example

$$\begin{array}{r} 3 \\ +2 \\ \hline 5 \end{array} \qquad \begin{array}{r} 0\,0\,1\,1 \\ 0\,0\,1\,0 \\ \hline 0\,1 \end{array}$$

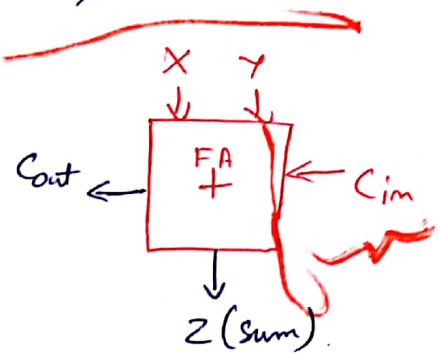
But this! This doesn't work with half-adders as I have no input that can accept carry from previous stage/step.

This also works with half adder

This works with half-adder

So, what should I do? If I want to add two number along with carry input?

I will design a new 1-bit adder called as FULL ADDER



Now, tell me when will my Sum be 1?

Q. When ~~None~~ None of the Inputs are 1
OR
When all the Inputs are 0.

No.

Q. When Exactly one of the Inputs is 1?

Yes. $X \cdot \bar{Y}\bar{C}_{in} + \bar{X}\,Y\,\bar{C}_{in} + \bar{X}\,\bar{Y}\,C_{in}$

When X is 1 AND Y is zero as well as C_{in} is zero.
When X is 0 AND Y is one AND C_{in} is zero.
When X is 0 AND Y is also zero AND C_{in} is one.

Q. Now tell me when else my Sum will be 1?

Q. When Exactly 2 inputs are 1

No, at that time 1+1=0 (so Sum will be zero carry will be 1).

Full adder (continued)

Q. So, when else my sum will be 1?

when all 3 of my inputs are 1?

Yes,

$\therefore Z = XY\bar{C}_{in} + X\bar{Y}C_{in} + \bar{X}YC_{in} + \bar{X}\bar{Y}\bar{C}_{in}$

$= X(Y\bar{C}_{in} + \bar{Y}C_{in}) + \bar{X}(Y\bar{C}_{in} + \bar{Y}C_{in})$

$= X(Y \oplus C_{in}) + \bar{X}(Y \oplus C_{in})$

Similarly, lets just consider carry out.

Q. When will my carry out bit (add bit) will be 1?

Q. when all of my inputs are 0?

No,

Q. when exactly one of my inputs is 1?

No.

Q. when exactly two of my inputs are 1?

Yes.

So, $XY\bar{C}_{in} + X\bar{Y}C_{in} + \bar{X}YC_{in}$.

Q. what happens when all of my 3 bit are 1?

Yes, it will also generate carry out.

So, my $C_{out} = XYC_{in} + XY\bar{C}_{in} + X\bar{Y}C_{in} + \bar{X}YC_{in}$

Now do a simple boolean algebra.

$C_{out} = XY(C_{in} + \bar{C}_{in})$

$= XY + X\bar{Y}C_{in} + \bar{X}YC_{in}$

$= XY + C_{in}(X\bar{Y} + \bar{X}Y)$

$= XY + C_{in}(X \oplus Y)$

$A + \bar{A} = 1$

Now, consider $X \oplus C_{in}$ as A

$Z = X\bar{A} + \bar{X}A$

$\Rightarrow Z = X \oplus A$

$= X \oplus (Y \oplus C_{in})$

$= X \oplus Y \oplus C_{in}$

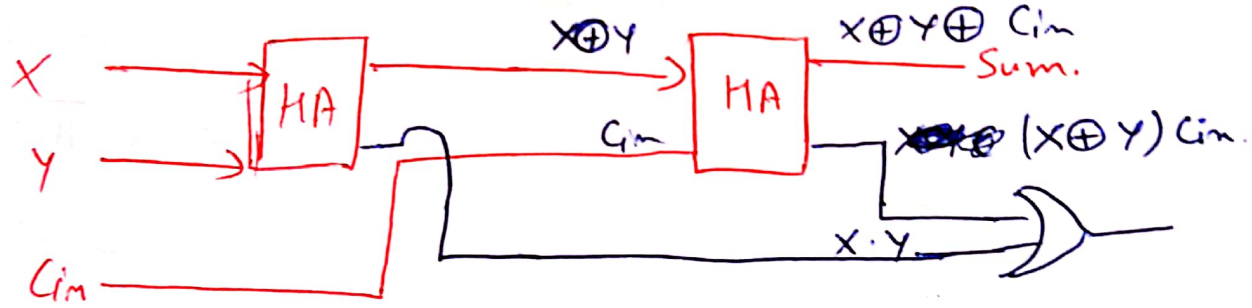
So, going through previous ~~diff~~ equation.

$X \oplus Y$ is what?

Sum of half-adder.

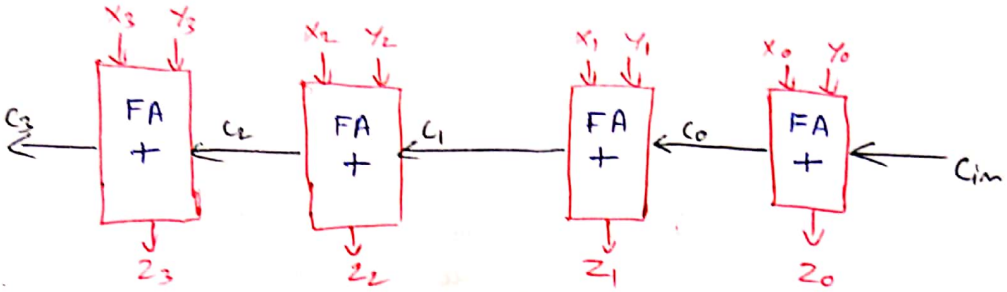
$X \cdot Y$ is wha?

Carry out of half-adder.



Serial / Slow adder / Ripple carry adder / Serial adder

So we have seen how to add two 1-bit added numbers,
Now, we will see how to make 4-bit adder using two-bit adders.



INPUTS

$X_3\ X_2\ X_1\ X_0$

$Y_3\ Y_2\ Y_1\ Y_0$

Cin

Outputs

$Z_3\ Z_2\ Z_1\ Z_0$

Cout.

// This is called a slow adder

// The main drawback is that I have to calculate C_0 before I can use Full Adder 1 to generate correct output.

Similarly, I have to wait for C_1 before I can use Full Adder 2

Therefore, it is slow adder.

// But, I want fast adder. I cannot afford slow adder.

If I have 8-bit adder it will require 8 clock cycles

16-bit ——— 16 clock cycles.

So, I want fast adders.

// Now, for that I require some special mechanism that can generate all the carry simultaneously.

In other words, some circuit that can predict these carries simultaneously.

predict can also be termed as "looking ahead into the future".
i.e. I have to design a Carry look ahead adder.

So, Now the question is how a circuit can predict a carry.

lets look at the Inputs that I have

$x_3\ x_2\ x_1\ x_0$

$y_3\ y_2\ y_1\ y_0$

C_{in}

To predict C_0, what do I require.

$$C_0 = x_0 y_0 + y_0 C_{in} + C_{in} x_0$$

$$C_0 = x_0 \cdot y_0 + C_{in} \cdot (x_0 + y_0)$$

$$\boxed{C_0 = g_0 + C_{in} \cdot P_0}$$

g_0 = generate, P_0 = propagate.

$$C_1 = x_1 y_1 + y_1 C_0 + C_0 \cdot x_1$$

$$= x_1 y_1 + C_0 (x_1 + y_1)$$

$$\boxed{C_1 = g_1 + C_0 \cdot P_1}$$

$$= g_1 + P_1 (g_0 + C_{in} P_0)$$

$$= g_1 + P_1 g_0 + P_0 P_1 C_{in}.$$

$$C_2 = x_2 y_2 + y_2 C_1 + C_1 \cdot x_2$$

$$= x_2 y_2 + C_1 (x_2 + y_2)$$

$$\boxed{C_2 = g_2 + C_1 \cdot P_2}$$

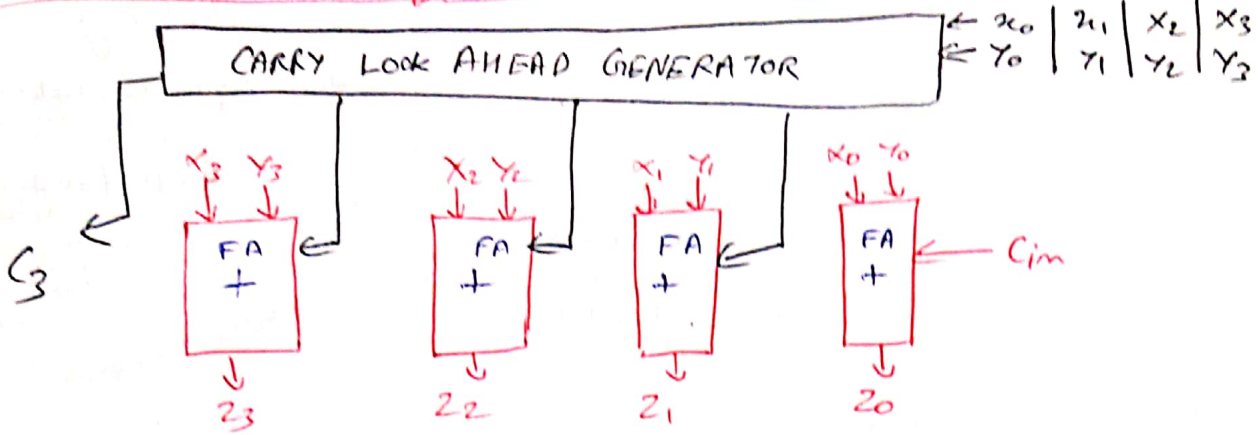
$$= g_2 + P_2 (g_1 + C_0 \cdot P_1)$$

$$= g_2 + P_2 g_1 + C_0 \cdot P_1$$

The ultimate conclusion is we can get and C_0, C_1, C_2, C_3 directly using our Inputs x_0, x_1, x_2, x_3, y_0, y_1, y_2, y_3, and C_{in}. So we can predict our carry before hand using a specialized circuit termed as carry look ahead generator.

FAST SUBTRACTOR

4-bit ~~subtractor~~ using 4-bit Fast Adder.



Now what I have to do is, I have to ~~add~~ create a 4-bit subtractor.

So, what is subtraction?

It is addition of 2's complemented no.

because 2's complement of any positive no. is its -ve no.

ex:- 9 $\Rightarrow$ +ve 9

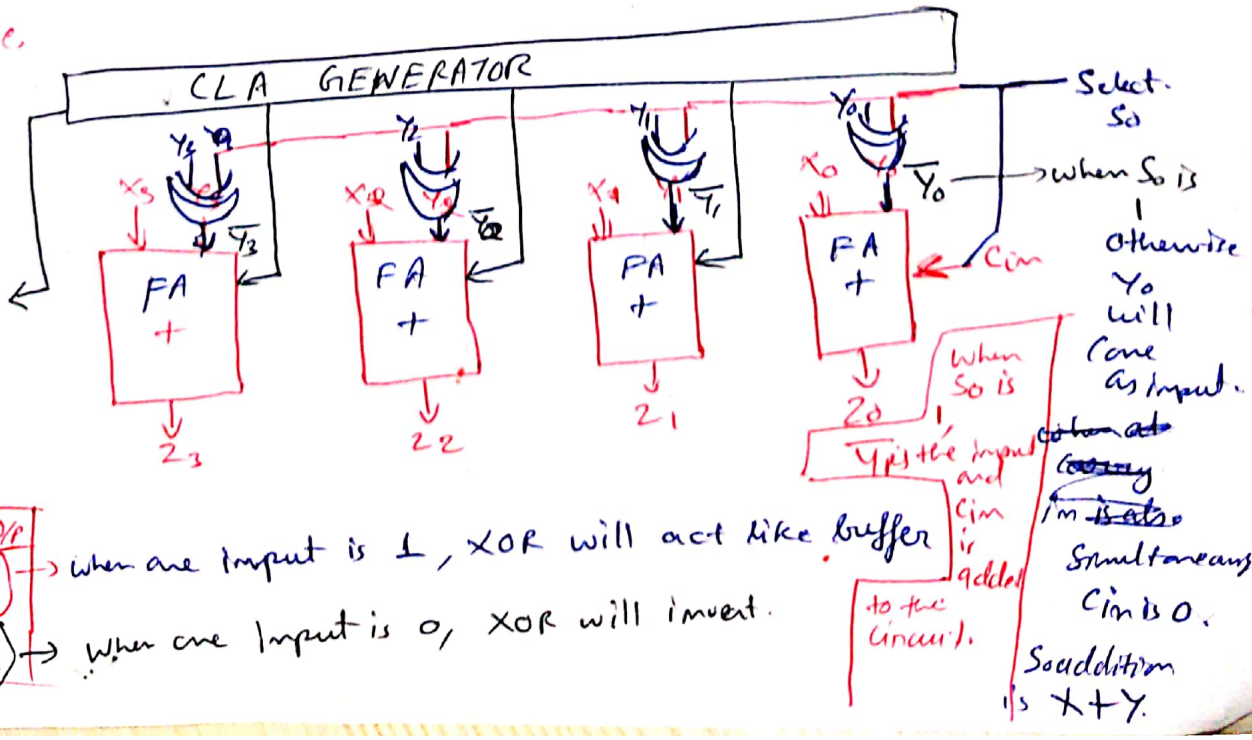
-5 $\Rightarrow$ 2's complement of 5 $\Big]$ Add them.

Now, how can we get 2's complement of a no.?

It is 1's complement Plus 1.

So, all that I have to do is add a circuit to my 4-bit adder that can generate 2's complement of one of its input while other input remains same.

i.e.



XOR

A	B	O/P
0	0	0
0	1	1
1	0	1
1	1	0

→ When one input is 1, XOR will act like buffer.

→ When one input is 0, XOR will invert.

Multiplication using Booth's Algorithm

5 (Multiplicand)
× 3 (Multiplier)
15 (Product)

```
  0101
  0011
  0101
  0101x
  0000xx
 0000xxx
 0001111
```

This method is faster than repeated addition.

```
 15
*15
225
```

15 +15 (15 times addition)

Our method requires only 4 steps.

```
 255
×255
```

255 +255 (255 times)

our method 8 steps.

Now, what kind of circuit do you need to implement this? Adder.

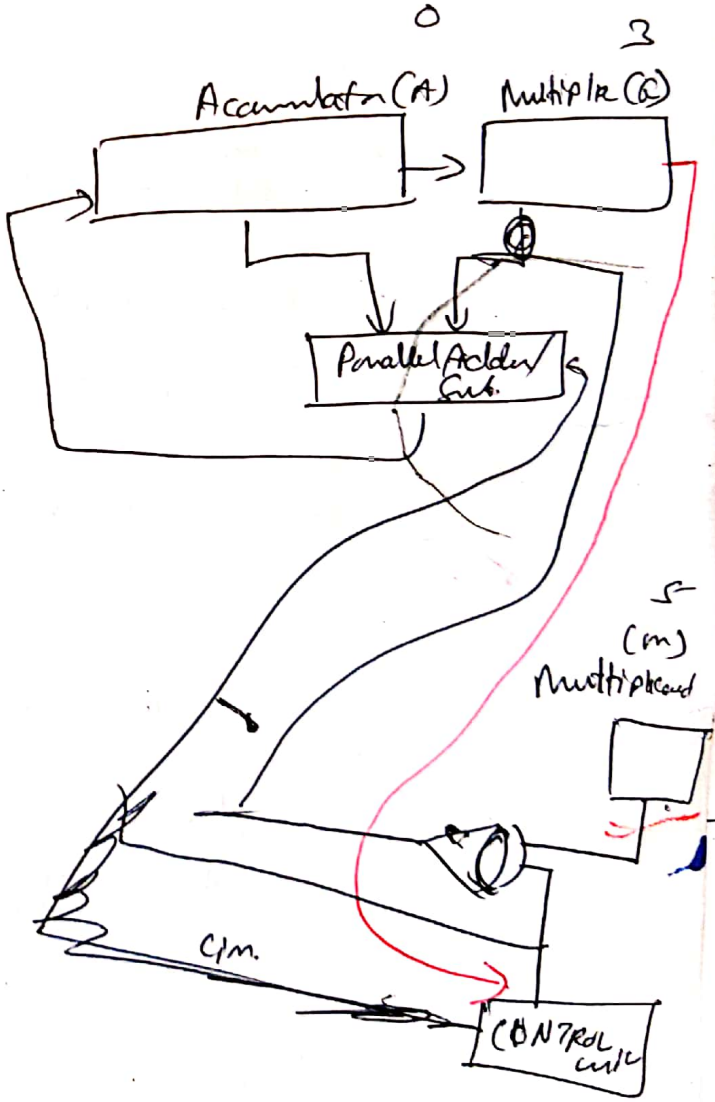
Accumulator

0000	0000

0101

shift Right

	0101	1000
	0111	1000
LS	0011	1100
	0000	
	0011	1100
	0001	1110
	0000	
	0001	1110
	0000	1111



Multiplication using booth's Algorithm:-

⇒ What are you gonna do if 9 ask you to multiply two 4-bit numbers using adders ~~only~~? Subtractor ONLY?

Sol⇒ Let say 5 x 3 what will you do?
Repeatedly add 5 3 times OR
Repeatedly add 3 5 times?

if yes, how much time does it takes 5 cycles, 3 cycles.
OR 15 cycles!

Largest value that can be stored in 4-bit no. is 15,
15 x 15 will require 14 repeated additions,
So, if ~~1 cycle~~ 1 addition requires 1 cycles
14 additions will require 14 cycles.

This was for 4-bit no.
for 8-bit no. it might take 254 cycles.
for 16-bit no. it ——"——, ——"—— 34 cycles.
which is a very hughe amount of time.

So, there must be a faster way to do this multiplication operation.
Hence, comes into picture your booth's Algorithm.

Now, before starting booth's Algorithm you have to recall basic multiplication

So, let's take an example.

```
5 x 3   0101
          0011
        -------
          0101
         0101x
        0000xx
       0000xxx
      -------
       0001111
```

What kind of circuit do you require to implement this circuit?

~~Addition and Shift~~ Addition
~~Addition~~ and shift ← —"—
shift ← —"—
shift ← —"—

We don't need multiplication here for obtaining the ~~so~~ multiplication.

⇒ No. of ~~bits~~ Steps = No. of bits in the

⇒ At each step, examine the multiplier's S (step no.) bit.
from LSB.

if 1, take that no. as its

if 0, take all 0's

Shift the no. by (Total steps − Step no) positions.

Add it to value obtained till previous step.

⇒ You keep on addition new value to previous step's value.
(in other words "you keep on accumulating the results")

⇒ The minimum size of the accumulator register is size of multiplicand + size of multiplier.

Now, lets see how this calculation is performed using the circuit?

⇒ Initially, your ~~circuit~~ Accumulator has value 00000000 (All zeros)

Accumulator

0000 0000

At step1:-
As multiplier's ~~LSB~~ 1st bit from ~~RHS~~ LSB is 1 ⇒ add 0101 to 0000 0000

0101

0101 0000

Right Shift 0010 1000

At step2:- 2nd ~~(TS-S)~~ bit from ~~RHS~~ LSB
As multiplier's ~~LSB~~ is 1 ⇒ add 0101 to 0010 1000

0101

0111 1000

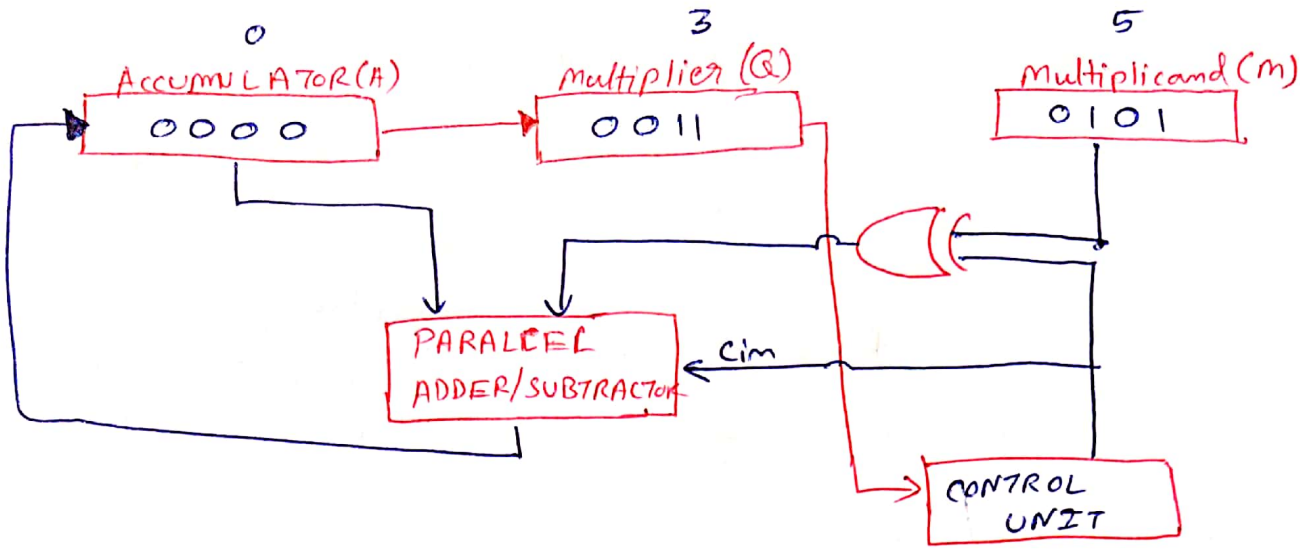
Right Shift 0011 1100

At step3:- As (3rd ~~TS-S~~) bit from LSB is 0 ⇒ add 0000 to

Add 0000 is not required so directly shift

At step4:- As 4th bit from LSB is 0 ⇒ Add 0000 to directly shift

0001 1110



The generic multiplication fails when we try to multiply ~~two~~ a -ve no.

Ex:- S 0101
X-3 1101

0101
0000x
0101xx
0101xxx

10000001

---

this is in 2's complement form.

Now check what no. it is?

01111111

∴ no. is −63. Wrong.

Why the answer comes out to be −ve? as we have considered these position as '0' (i.e. +ve by default)

∴ ~~lets take an example,~~ what Booths Algorithm does well is it takes MSB and extends the no. ~~by~~ copying MSB.

Lets look at the tabular method of solving the algorithm:

1. The no. of steps = no. of bits in the multiplier
2. At every step examine the last 2 bits near the LSB of the multiplier (including imaginary position)

   if bits are 10, subtract the ~~multiplier~~ multiplicand (add 2's complement of multiplicand)

   if bits are 01, ~~subtract~~ add the ~~multiplier~~ multiplicand. (as it is)

   if bits are 00 or if bits are 11 } Do Nothing.

3. Perform Right shift
4. ~~Repeat~~ Repeat steps 2 and 3 ~~till~~ till it reaches the no. of steps.

Another way to write this Algo.

1. —— " ——
2. —— " ——

   if transition is from 0 to 1, subtract the multiplicand (i.e. add 2's complement of the multiplier)

   if transition is from 1 to 0, ~~subtract~~ add the multiplicand as it is

   if ~~the~~ No transition i.e. 1 to 1 or 0 to 0 } Do nothing

3. —— " ——
4. —— " ——

TABULAR METHOD OF Booth's Algorithm for signed multiplication

Ex:- 5 × 7

5 = 0101   7 = 0111

−5 = 1011   −7 = 1001

7	A (0) ACCUMULATOR	Q (7) Multiplier	imaginary Q-1	M (5) Multiplicand
	0 0 0 0	0 1 1 1	0	0 1 0 1
Transition 0 to 1 SUBTRACT M	1 0 1 1			
	1 0 1 1	0 1 1 1		
RS	1 1 0 1	1 0 1 1	1	
Transition 1 to 1 DO NOTHING				
RS	1 1 1 0	1 1 0 1	1	
Transition 1 to 1 Do nothing				
RS	1 1 1 1	0 1 1 0	1	
Transition 1 to 0 ADD M	0 1 0 1			
	0 1 0 0	0 1 1 0		
RS	0 0 1 0	0 0 1 1	0	

This no. is +ve ∴ answer is 35 $(2^7\ 2^6\ 2^5\ 2^4\ 2^3\ 2^2\ 2^1\ 2^0)$

0 0 1 0 0 0 1 1

NOTES:- Things to remember to avoid mistakes.

⇒ [1] Write the nos./bits one below the other to avoid errors during shifting.

⇒ [2] Take both multiplier and multiplicand of same size (i.e., if multiplier is 4 bits and multiplicand is 5-bits take both nos. as 5 bits to avoid making errors)

⇒ [3] Take both multiplier and multiplicand one bit more than they actually are example take 5 as 0101, instead of 101 this helps avoiding mistakes while taking 2's complement of these nos.

Ex 1-2:- 5x−7

5 = 0101 7 = 0111
−5 = 1011 −7 = 1001

	A (0) ACCUMULATOR	Q (−7) MULTIPLIER	Q−1	M (5) MULTIPLICAND
	0 0 0 0	1 0 0 1	0	0 1 0 1
Transition is 0 to 1 SUBT. M (Add −5)	1 0 1 1	1 0 0 1		
RS	1 1 0 1	1 1 0 0	1	
Transition is from 1 to 0 ∴ Add M	0 1 0 1	1 1 0 0		
RS	0 0 0 1	0 1 1 0	0	
No transition ∴ Do nothing				
RS	0 0 0 0	1 0 1 1	0	
Transition from 0 to 1 SUB. M	1 0 1 1	1 0 1 1		
RS	1 1 0 1	1 1 0 1	1	

Result

To check its value i will take 2's complement of this no.

it will comes out to be.

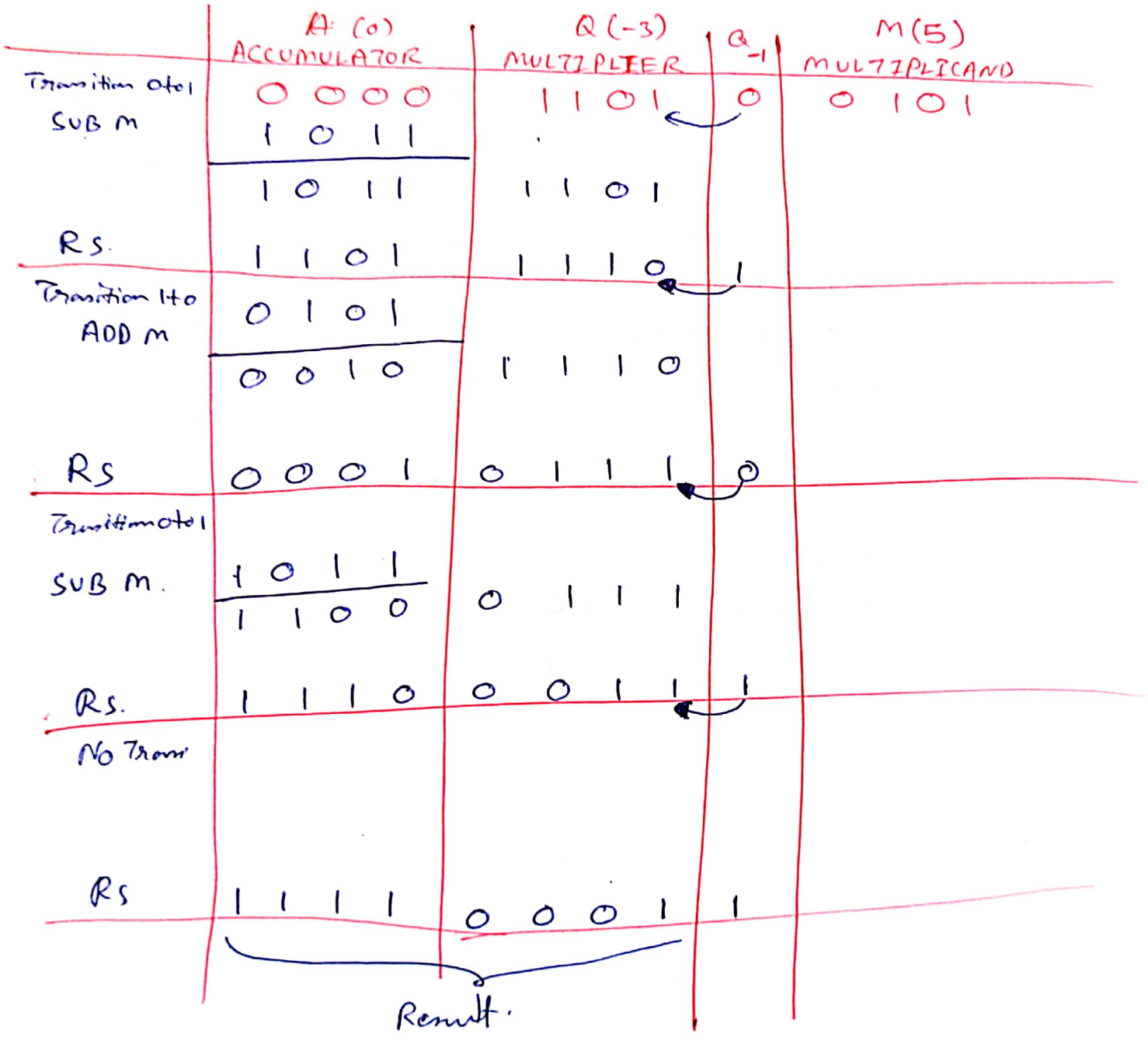
0 0 1 0 0 0 1 1

∴ no. is ⟹ −35

Q:- 5 × -3.

5 (Multiplicand) 5 = 0101 3 = 0011

-3 (multiplier) -5 = 1011 -3 = 1101



2's complement.

0 0 0 0 1 1 1 1

∴ my answer is -15. correct done

5×-3

0101×1101

0101

$0000X$

$0101XX$

$0101XXX$

$0000 \quad 0101 \quad 0 \quad 1101$

if Tranition if from 0 to 1, Subtract $\Rightarrow$ Add 2's complement of 0101 | i.e. 1011

if transition is from 1 to 0, Add.

$0000 \quad 0101$

$1011 \quad 0101$

$1101 \quad 1010 \leftarrow 1$

Now, transition is from 1 to 0 $\Rightarrow$ Add. 0101

0101

$0010 \quad 1010$

$0001 \quad 0101 \leftarrow 0$

Now, transition is from 0 to 1 $\Rightarrow$ Subtract $\Rightarrow$ Add 1011

1011

$1100 \quad 0101$

$1110 \quad 0010 \leftarrow 1$

Now, transition is from 1 to 0 $\Rightarrow$ Add 0101

0101

$0011 \quad 0010$

$0001 \quad 1001 \quad 0$

This is +ve no.

~~Repeat~~ DESIGNING AN ALU (DIVISION)

Example:-

Case 1:

$$4 \,\overline{)\,64\,}\, 1$$
$$-4$$
$$+2$$

So, our ~~division~~ step is succesful.

Case 2:

$$8 \,\overline{)\,64\,}$$
$$-8$$
$$-2$$

As our subtraction gives −ve no. So, our step is unsuccesful. So, Now we have to get back 6, ~~for that~~ (This is why this ~~division~~ method is called restoring division).

So, how can we get 6 back
⇒ you have subtracted 8, So, to restore back add 8.

There are two types of division Algorithms available to us.

- Restoring Division
  AND
- NON - Restoring division

We, will discuss the restoring division first

Algo.

1. The total no. of steps = No. of digits in the dividend
2. At each step Left shift the dividend
   - Make an attempt to subtract divisor
     - if result is +ve, step is successful — Append 1 to Quotient ~~is~~ No restoration is required
     - if result is −ve, step is not succesful
       - if step is unsuccesful restore the dividend. by adding divisor to the current value (~~make~~ ~~get~~ ~~append~~ ~~quotient~~) (append 0 to the quotient)
3. Repeat step 2 till ~~all~~ ~~the~~ total no. of steps are reached.

Designing of An ALU (DIVISION):-

Q So, why just like repeated addition, for performing multiplication, Why can't I use repeated subtraction, for performing division?

Sol⇒ As repeated addition was slow,
Similarly, repeated subtraction is slow.

Example:- ~~when you convert decimal to binary.~~

100 divide by 2

Now you will require 50 subtractions (approx.).

Therefore, we require special Algorithms that can perform division in lesser time & lesser steps.

OR.

Ex:-

$$2\overline{)100}\,0$$

$$\begin{array}{r} -2 \\ \hline -1 \end{array}$$

Not successful

Now, you want your no. back, so, just add 2.

Restoring.

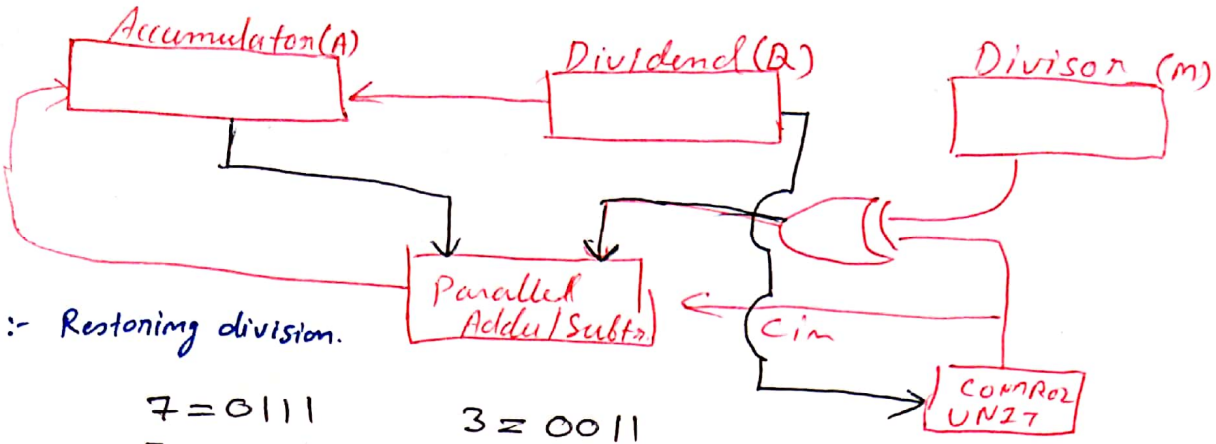
Now, subtract 2 from 10 (way you do that yes. how many times

5 times

So, your ~~correct~~ answer is

$$2\overline{)100}\,50$$

$$\begin{array}{r} -10 \\ \hline 00 \end{array}$$

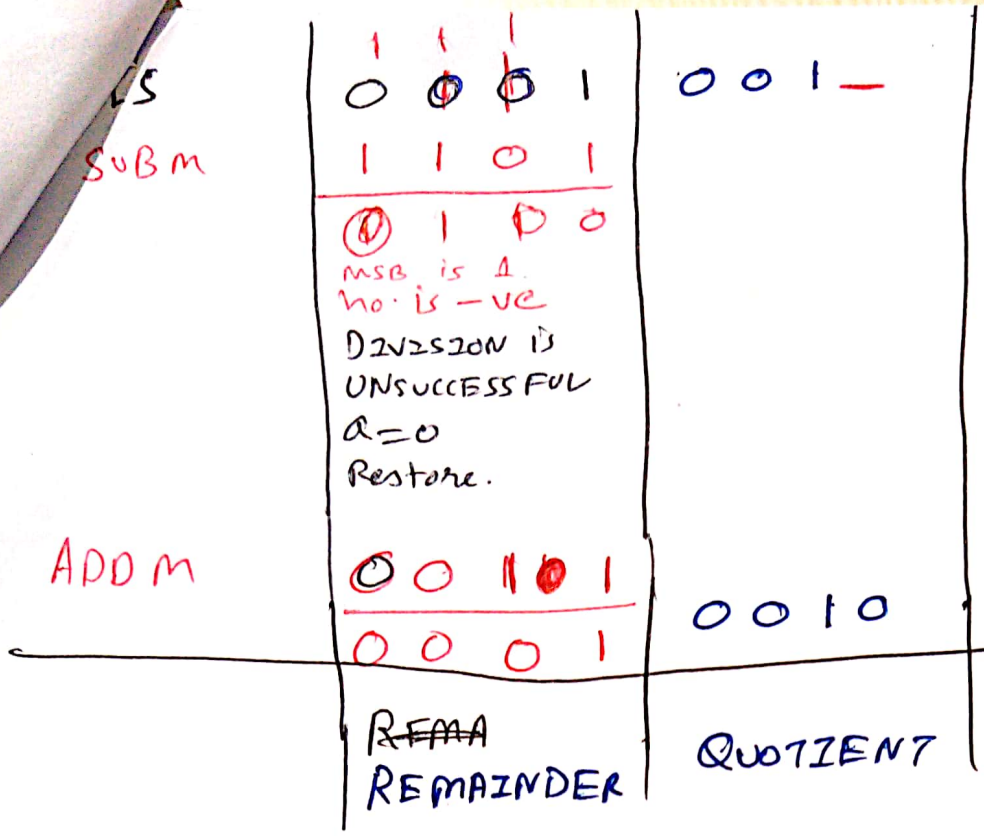


Example :- Restoring division.

$7 = 0111$
$-7 = 1001$

$3 = 0011$
$-3 = 1101$

	A (0) ACCUMULATOR	Q (7) DIVIDEND	M (3) DIVISOR
	0000	0111	0011
LS	0000	1110	
SUB M	+1101		
	(1)101		
	As no. is -ve ∴ unsuccessful ∴ Q = 0 Restore by Adding DIVISOR (3)		
ADD M	+0011	1110	
	0000		
LS	0001	110_	
SUB M	+1101		
	(1)110		
	MSB is 1, No is -ve, ~~DIVISION~~ UNSUCCESSFUL SUBTRACTION DIVISION Q = 0 Restore		
ADD M	+0011	1100	
	0001		
LS	0011	100_	
SUB M	+1101		
	0000		
	MSB is 0, No. is +ve, ~~SUBT~~ DIVISION SUCCESSFUL Q = 1 No Restoration		
		1001	



So, ~~8.~~ $7 \div 2$ has given remainder = 1

Quotient = 2

Your remainder is in AX (Accumulator)

and your ~~Q~~ quotient is in DX (register).

Non-Restoring DIVISION :-

So, Now why do we require non-restoring division?
If we already have restoring division.

→ Restoring Division is fine, but, typically in most of the division most of the subtraction attempts are unsuccessful
So, we have to perform restoration w.r.t. each unsuccessful division.
This creates an overhead.
To, remove this overhead we are using Non-restoring division.

NOTE:- Non-restoring division works only for +ve nos. i.e. for unsigned nos.

On the other hand,

Restoring division works for both +ve as well as -ve nos. i.e. for signed nos.

Therefore, in 8086 up (microprocessor) you have seperate instructions for performing division for
Signed (IDIV)
Unsigned nos. (DIV)

Algorithm for Non-restoring division :-

1. The total no. of steps = No. of digits in the dividend
2. At each step left shift the dividend
   - ↳ Make an attempt to subtract divisor
     - ↳ if result is +ve, step is successful, append 1 to Quotient.
     - ↳ if result is -ve, step is unsuccessful, append 0 to Quotient and make the ~~addition~~ in the next step instead of ~~this~~ subtraction.
3. Repeat step 2 till total no. of steps are reached.
4. ONLY IF Last step is unsuccessful restoration is needed.

Example Non-restoring division

7 = 0111
−7 = 1001

3 = 0011
−3 = 1101

	A(0) ACCUMULATOR	Q(7) DIVIDEND	M(3) DIVISOR
	0000	0111	0011
LS	0000	111_	
SUB M	+1101		
	(1)101		
	No. is −ve DIVISION is UNSUCCESSFUL		
NEXT STEP ADD M instead of SUB M		1110	
LS	1011	110_	
ADD M	0011		
	(1)100		
	No. is −ve DIVISION is UNSUCCESSFUL	1100	
NEXT STEP ADD M			
LS	1001	100_	
ADD M	0011		
	(1)000		
	No. is +ve DIVISION is SUCCESSFUL	1001	
~~Next Step ADD M~~			
LS	0001	001_	
SUB M	+1101		
	(1)110		
	−ve. DIVISION is UNSUCCESSFUL	0010	
	PERFORM RESTORATION AS IT IS LAST STEP.		
ADD M	0011		
	0001		
	QUOTIENT	REMAINDER	

RESTORING DIVISION FOR SIGNED NUMBERS:-

1. let M register hold the divisor, Q register hold the dividend.
2. A register should be signed extension of Q.
3. On completion of the algorithm, Q will get the quotient and A will get the remainder

Algorithm:-

The number of steps required is equal to the no. of bits in the dividend

1. At each step, Left Shift the dividend by 1 position.
2. if Sign of A and M is same then Subtract the divisor from A
   (perform A-M; Add 2's complement of M)

   Else Add M to A.
3. After the operation,

   if Sign of A remains the same
   OR
   the dividend (in A and Q) becomes zero } then Step is successful
   Q bit is 1
   No Restoration.

   if Sign of A changes
   then step is unsuccessful
   Q bit is "0"
   Restoration is required
4. Repeat steps 1 to 3 till all the steps are completed.

NOTE:- The result of the algorithm is such that, the quotient will always be positive and the remainder will get same sign as the dividend.

RESTORING DIVISION FOR SIGNED NUMBERS:-

$-8/-4$

$8 = 01000$ $\quad 4 = 00100$

$-8 = 11000$ $\quad -4 = 11100$

	A (SIGN EXTENSION) ACCUMULATOR	Q (-8) DIVIDEND	M (-4) DIVISOR
INITIAL VALUES	(1)1111	11000	(1)1100
LS	11111	1000_	
Sign A, M same: A+M ⇒ SUB M	+ 00100		
Sign of A changes Step is unsuccessful	(0)0011	1000(0)	11100
Restore A Add M	+ 11100		
	11111	10000	
LS	11111	0000_	
SUB M	+ 00100		
Sign of A changes Step is unsuccessful	(0)0011	00000	
Restore A by adding m	+ 11100		
	11111		
LS	11110	0000_	
SUB M	+ 00100		
Sign of A changes Step unsuccessful.	(0)0010		
Restore A by adding M	+ 11100		
	11110	00000	
LS	11100	0000_	
SUB M	+ 00100		
DIVIDEND(A,Q)=0 Step is successful. NO Restoration	00000	(0)0001	
LS	00000	0001_	
Sign (A,M) Different: A+M	+ 11100		
Sign of A changes Step unsuccessful Restore By Sub M. i.e. add 2's complement of M	(1)11100		
	+ (0)0100	00010	
	00000		

Ex:- (-19)/7

19 = 010011 — 7 = 000111

-19 = 101101 — -7 = 111001

	Sign Extension of A: ACCUMULATOR (A)	(-19) DIVIDEND (Q)	(7) DIVISOR (M)
Initial values.	0 1 1 1 1 1 1	1 0 1 1 0 1	0 0 0 1 1 1
LS	0 1 1 1 1 1 1	0 1 1 0 1 0	
Sign (A,M): Diff. A+M	0 0 0 1 1 1		
A's Sign changes: Unsuccessful	0 0 0 1 T 0		
Restore: Sub A-M	+ 1 1 1 0 0 1		
	1 1 1 1 1 1		
LS	1 1 1 1 1 0	1 1 0 1 0 0	
Sign(A,M): Diff. A+M	+ 0 0 0 1 1 1		
A's sign changes: Unsuccessful	0 0 0 1 0 1		
Restore: Sub A-M	+ 1 1 1 0 0 1		
	1 1 1 1 1 0		
LS	1 1 1 1 0 1	1 0 1 0 0 0	
Sign(A,M): Diff. A+M	+ 0 0 0 1 1 1		
A's Sign changes: Unsuccessful	0 0 0 1 0 0		
Restore: Sub. A-M	+ 1 1 1 0 0 1		
	1 1 1 1 0 1		
LS			

EXECUTION OF AN INSTRUCTION:-

⇒ A typical computing task consists of a series of steps specified by a sequence of machine instructions that constitute a program.

⇒ A program is a set of instruction.
An instruction is a set of micro instruction.
A microinstruction is a set of nanoinstructions.

⇒ To execute a program, you have to execute several instructions.
To execute an instruction, you have to execute several microinstructions.
To execute a microinstruction, you have to execute a few nanoinstructions.

⇒ To execute a program,
the processor fetches one instruction at a time and performs the operations specified.

⇒ Program Counter (PC) stores the address of next instruction to be executed.

⇒ Instruction Register (IR) ~~has~~ stores the current instruction to be executed.

⇒ Suppose each instruction is of 4 bytes and stored in a byte addressable memory. then, to execute an instruction a processor has to perform following steps:-

Fetch phase:
1. Fetch the contents of the memory location pointed to by the PC. ~~The contents of this location are interpreted as an instruction to be executed. Hence,~~ I into IR.

$$IR \leftarrow [[PC]]$$

2. Increment PC.

$$PC \leftarrow [PC] + 4.$$

Execution phase:
3. Execute the instruction in the IR register.

if size of the instruction is more than 4 bytes, repeat step 1 and 2, till the entire instruction is fetched.

Internal organization of a processor :-

⇒ In this simple organization there is a Single processor bus (Internal bus)

⇒ The Internal bus should not be confused with External bus; External bus connects ~~with~~ processor with Memory and I/O devices.

⇒ Note :- Memory Address Register (MAR)
AND
Memory data Register (MDR)
has two inputs and two outputs.

⇒ MDR can take data from memory (Input) and ~~se~~ send ~~the same data~~ it to processor (output)

OR

MDR can take data from processor (Input) and send it to memory (output)

These ~~operations~~ instructions are decoded and controlled by the control unit.

⇒ R_0, $R(n-1)$ may be used as general purpose or special purpose registers such as Accumulator, ~~register~~, Count register etc.

⇒ The registers Y, Z and temp are internal to the processor and not directly accessible to the programmer (user).

The ~~ALU~~, ~~registers~~, ALU, and the interconnecting buses are collectively ~~called as~~ refered as DATAPATH.

BASIC STEPS OF Execution:-

- Transfer a data from one register to another register (Register transfer)
- Perform an arithematic or logic operation and store the obtained result in a processor register.
- Fetch the contents of a given memory location and load them into a processor register.
- Store the result back into the memory location.

Register transfer:-

⇒ The input and output of Register R_i are controlled by signals $R_{i_{in}}$ and $R_{i_{out}}$ respectively.

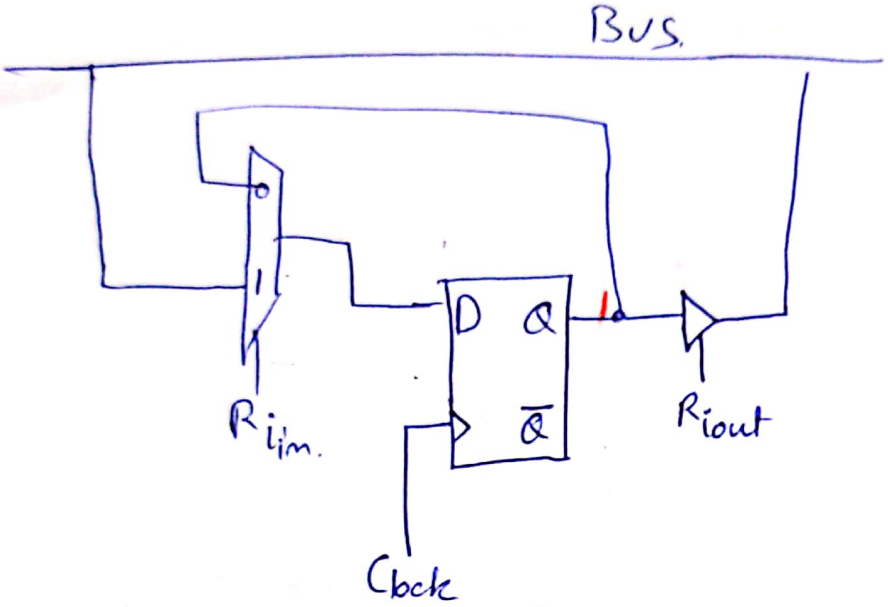
When $R_{i_{in}} = 1$, data available on the bus is loaded into the register.

when $R_{i_{out}} = 1$; data ~~of~~ the register is ~~out~~ placed on the internal bus.

⇒ To transfer data from R_1 to R_4.

Enable $R_{i_{out}} (=1)$

Enable $R_{4_{in}} (=1)$.



Input and output gating of one register bit

⇒ The Q output of D flip-flop is connected to the bus via a tri-state gate.

when $R_{iout} = 0$, gate's O/p is high impedance.

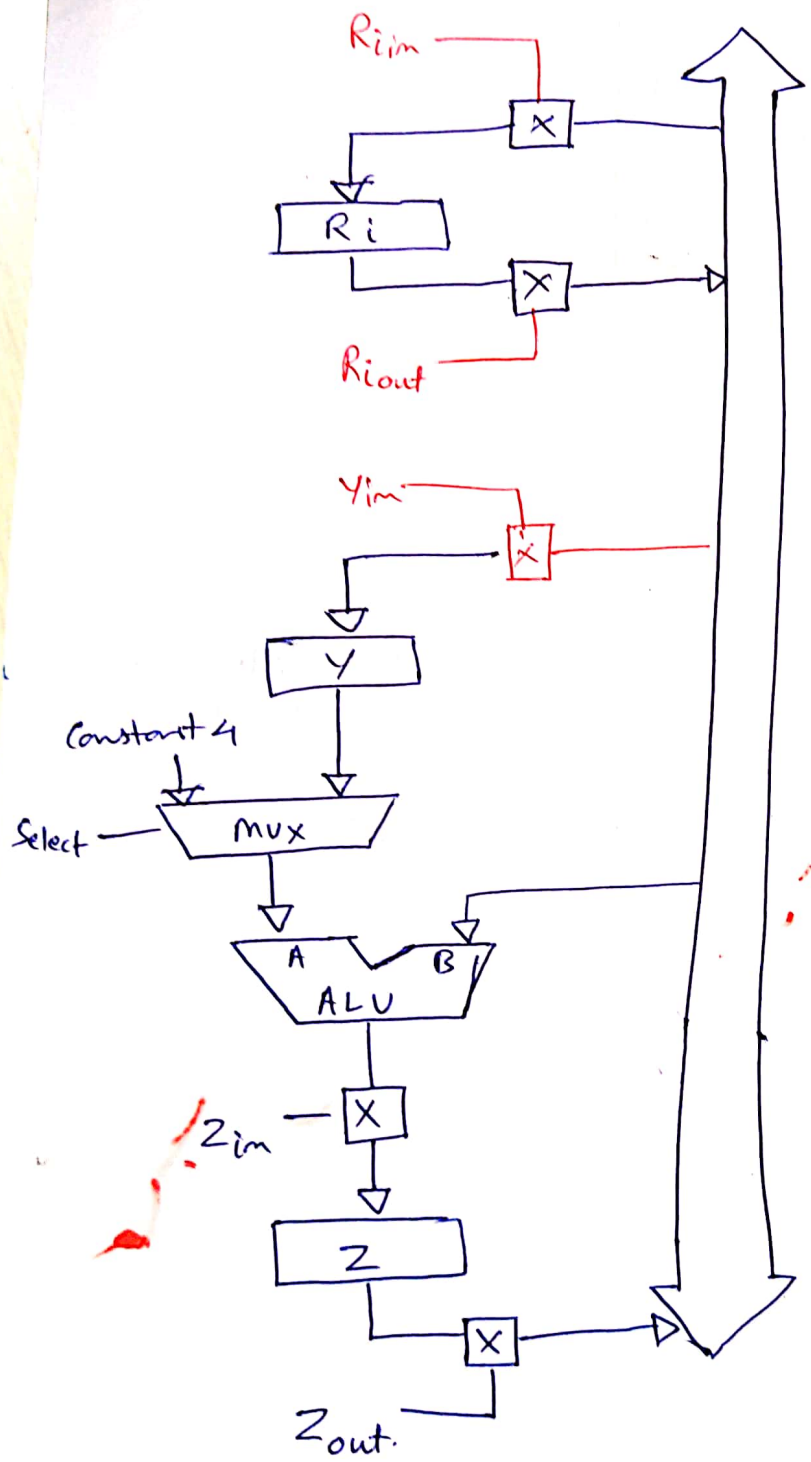
$R_{iout} = 1$, gate's O/p is 0 or 1 depending on value of Q.

Performing an arithmetic or Logic operation.

Add R_3, R_2, R_1.

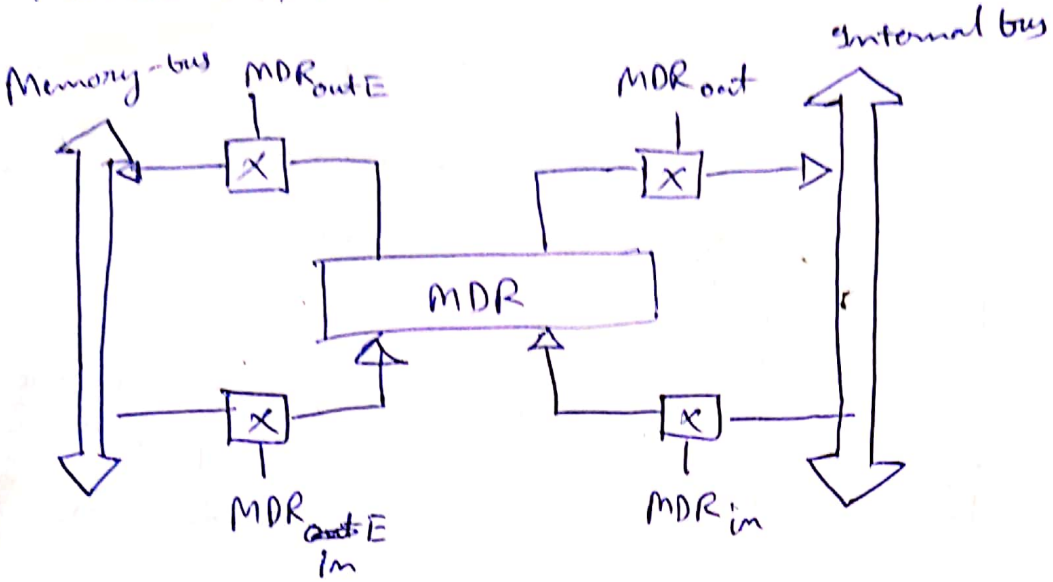
1. R_1 out, Yin
2. R_2 out, SelectY, Add, Zin
3. Zout, R_3in.

Observe; which value is placed on the processor's bus.



Fetching a word from memory:-

Move (R_1), R_2



1. R_1 out, MAR in, Read
2. MDR inE, WMFC
3. MDR out, R_2 in.

Wait for memory function completed.
Wait for Memory fetch to Complete

Storing a word in memory :-

Move R_2, (R_1)

1. R_1 out, MAR in
2. R_2 out, MDR in, Write
3. MDR outE, WMFC

EXECUTION OF A COMPLETE INSTRUCTION :-

Add $(R_3), R_1$

1. PCout, MARin, Read, Select 4, Add, Zin
2. Zout, PCin, Yin, WMFC

} Program counter is updated to point to the next instruction

3. MDRout, IRin → Instruction fetch
4. R_3out, MARin, Read # Address of memory location given by R_3 is read into MAR
5. R_1out, Yin, WMFC # 2nd operand R_1 is placed on the bus.
6. MDRout, Select Y, Add, Zin # value of R_3 is placed on the bus and added; result is stored in Zin.
7. Zout, R_1in, End. # The result is stored back in R_1.

Unconditional branch Instruction :-

Execution of an

1. PCout, MARin, Read, Select 4, Add, Zin
2. Zout, PCin, Yin, WMFC
3. MDRout, IRin
4. Offset-field-of-IRout, Add, Zin
5. Zout, PCin, End

~~JMP~~ JMP LABEL1

1000 JMP 36

1040

Notes:- A branch instruction replaces the contents of the PC with the branch target address.

This address is usually obtained by adding an offset X, given by the branch instruction. (w.r.t. updated value of PC)

Eg:- if the branch instruction is at location 1000, and if branch target is 2040, the value of X must be 36.

Conditional branch :-

~~jdz~~ jlz (branch < 0)

then the 4th line will be updated as.

4. Offset-field-of IR_{out}, Add, Z_{in}, If N=0 then End.

∴ if N=0, processor returns to step 1 immediately.

∴ if N=1, step 5 is performed

three bus organization

Add R_4, R_5, R_6

1. PC_{out}, R=B, MAR_{in}, Read, IncPC
2. WMFC
3. $MDR_{out}B$, R=B, IR_{in}
4. R_4 outA, R_5 outB, Select A, Add, R_{in}, End

The contents of PC are passed to bus C through ALU. then same content is passed to MAR using bus C. as MAR is connected only to bus C.

Operands comes from buses A and B and the results are transferred to the destination over bus C.

ALU can pass one of its two input operands unmodified to BUS C. such control signals are represented by R=A, or R=B.

PC has a seperate incrementor unit, Nevertheless, constant 4 at the ALU is still useful for instructions like load multiple store multiple (assuming we are ~~storing~~ loading/storing) 4 bytes at a time.

#

RISC	v/s	CISC
1. RISC architecture utiliges a small, highly optimized set of instructions.		1. CISC architectures utilizes a ~~small~~ specialized set of Instructions.
2. Longer code Short instructions		2. Shorter code Long Instructions
3. ~~Emphasis is on developing hardware for complex instructions.~~		3.
3. Emphasis is on developing software		3. Emphasis is on developing hardware for complex set of instruction.
4. Each Instruction takes 1 clock cycle		4. Each Instruction may takes multiple clock cycles.
5. Easy to pipeline		5. Hard to pipeline
6. Ex of RISC processors:- Sun UltraSPARC, IBM Power PC.		6. Ex:- intel x86 family of Architectures. Motorola 6800, 6809.

Inter stage buffers:

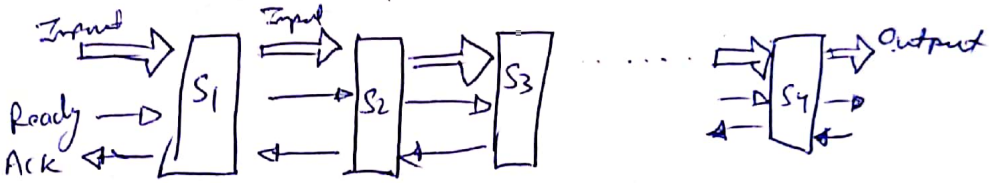
Within each stage of pipeline a Buffer is needed to hold the information being passed from one stage to another.

This is a part of SYNCHRONOUS PIPELINE

Asynchronous pipeline:- In asynchronous pipeline.

Transfer of information b/w two stages takes place when individual stages are ready.

In Asynchronous pipelines there is no concept of buffers, handshaking takes place using Ready and Ack signals.



Asynchronous

=> ~~Different~~ ~~This kind of~~ pipelining is used when time required to execute one stage of pipelining is more/less than the another stage.

Pipelining can implemented in two manners Synchronously and Asynchronously.

$$\text{Throughput} = \frac{\text{Total no. of Instructions being executed}}{\text{Total time taken to execute these instructions.}}$$

Pipelining can be implemented when a task can be divided into two or more subtasks, which can be performed independently.

Different subtasks are performed by different hardware blocks known as stages.

The intermediate results generated by each stage is temporarily buffered in latches and then passed on to the next stage.

2-stage pipelining:

F: fetch an instruction from the memory.
E: perform the operation specified by the instruction

	1	2	3	4	5	6	7
I_1	F_1	E_1					
I_2		F_2	E_2				
I_3			F_3	E_3			
I_4				F_4	E_4		

Throughput $= \frac{4}{5} = 0.8$ | if Non-pipeline execution $= \frac{4}{8} = 0.5$

4-stage pipelining:

F: read the instruction from the memory
D: decode the instruction and fetch the source operand(s)
E: perform the operation specified by the instruction
W: store the result in the destination location

Pipelining is most effective in improving performance if the task being assigned in different stages requires about the same amount of time.

⇒ The clock period should be chosen such that it corresponds to the longest duration among all the stages.

A pipeline stage is not able to finish its task within a allocated time, it impacts the execution of entire pipelining.

A pipeline is said to have been stalled for these no. of cycles.

Any condition that causes a pipeline to stall is called a hazard.

Data hazard :- A data hazard is any condition in which either the source or the destination operands of an instruction are not available at the expected time in the pipeline

Instruction hazards :- A pipeline may be stalled due to unavailability of an instruction due to an event such as cache miss., thereby requiring instruction to be fetched from main memory, this type of hazards are known as instruction hazards or Control hazards.

Structural hazards :- This type of hazards may occur when two or more instructions requires the use of a given hardware resource at the same time.

Ex- Load X(R1), R2

Effect of an execution operation taking more than one clock cycle.

Clock cycle	1	2	3	4	5	6	7	8	9	10
I_1	F_1	D_1	E_1	W_1						
I_2		F_2	D_2	E_2	E_2	E_2	W_2			
I_3			F_3	D_3	id	id	E_3	W_3		
I_4				F_4	id	id	D_4	E_4	W_4	
I_5					id	id	F_5	D_5	E_5	W_5

Form of Data hazard

Pipelined operation is stalled for two clock cycles.
Any condition that causes the pipeline to stall is called hazards

Clock cycle	1	2	3	4	5	6	7	8	9
I_1	F_1	D_1	E_1	W_1					
I_2		F_2	F_2	F_2	D_2	E_2	W_3		
				Didle	Didle	waita	widle		
I_3				Eidle	F_3	D_3	E_3	W_3	
					Eidle				
I_4									
I_5									

Didle 3, 4
Eidle 4, 5
Widle 5, 6

These types of stalls are also called bubbles, once created they will move downstream until it reaches the last unit.

This is an example of instruction hazard due to unavailability of instruction due to event such as cache miss.

The no. of cycles required to execute 'N' no. of instructions in a k-stage pipeline are.

$$K + (N-1).$$

..	1	2	3	4	5	6	7	8	9	10	11	12
I_1	F_1	D_1	E_1	W_1								
I_2		F_2	D_2	E_2	W_2							
I_3			F_3	D_3	E_3	W_3						
I_4				F_4	D_4	E_4	W_4					
I_5					F_5	D_5	E_5	W_5				
I_6						F_6	D_6	E_6	W_6			
I_7							F_7	D_7	E_7	W_7		
I_8								F_8	D_8	E_8	W_8	
I_9									F_9	D_9	E_9	W_9

$K = 4$, $N = 9$

Total no. of cycles $= K + (N-1) = 4 + (9-1) = 12$

Total time of execute 'N' instruction in k stage pipelining with each stage taking τ unit of time. $= [k + (N-1)]\tau$

However, it is not always possible to make time of each stage equal.

For eg:- in 3-stage organization, we have seen it require 3 control steps to ~~generate~~ fetch the instruction and ~~two stage~~ one control step to execute the instruction.

Therefore, in such a scenario where time of each stage cannot be made equal.
The clock period should be chosen such that, it corresponds to the longest duration among all the stages.

Speed up = ~~Performance of pipelined processor~~ / Performance of Non-pipelined processor

=

speed = distance / time. = ~~time~~ cycles/time / cycles/time.

= cycles / time.

= time taken by non-pipelined processor / time taken by pipelined processor

$= \frac{n * k * \tau}{[k + (n-1)]\tau} = \frac{n * k}{k + (n-1)}$

for large value of n, $= k$.

∴ performance of pipeline increases with ~~time~~ no. of stages

However, this is not always true as we haven't considered buffer delay in this calculation in the pipeline