Computer Organisation & Architecture

Computer architecture define the need or ~~purpose~~ purpose of computer that to be desire.

Computer organisation then realizes the computer to full fill the needs define by the corresponding architecture.

* Adder :→

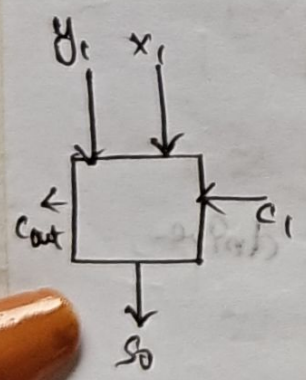
x_0	y_0	c_0	S_0	c_1
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

$$S_0 = \bar{x}_0 \bar{y}_0 c_0 + \bar{x}_0 y_0 \bar{c}_0 + x_0 \bar{y}_0 \bar{c}_0 + x_0 y_0 c_0$$

$$c_1 = \bar{x}_0 y_0 c_0 + x_0 \bar{y}_0 c_0 + x_0 y_0 \bar{c}_0 + x_0 y_0 c_0$$

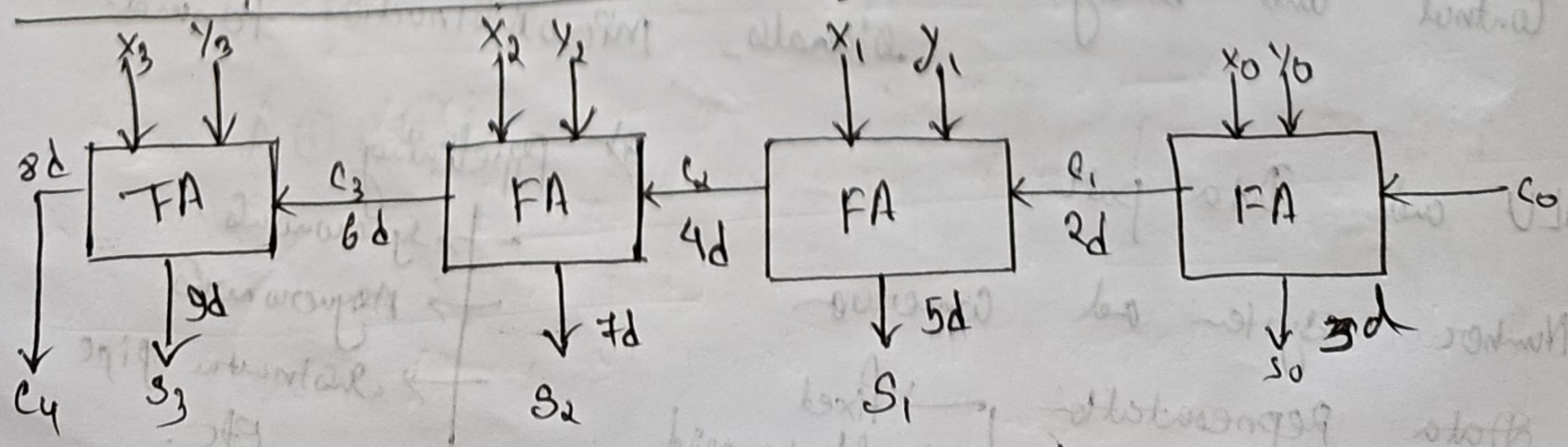
$$= x_0 y_0 + y_0 c_0 + x_0 c_0$$

$$= x_0 y_0 + c_0 (x_0 + y_0)$$



Thus, Sum needs 3 gate dellay and carry needs 2 gate delay.

4-bit full adder circuit (carry properties adder) :→



Carry → 2n
Sn → 2n - 1
Overflow → 2n+1

- Total delay = 9d for 4 bit adder.
- For n-bit adder delay will be (2n+1) d.
- For 16 bit adder delay will be 33d.

This delay is prohibitive for high speed systems and the delay also increases with the size of adder.

This creates the necessity to devise a new technique to increase the speed of operation known as carry-look-ahead. In this approach extra hardware is used to generate carry ($c_i, i>0$) directly from c_0.

Let us recall that in a full adder the output carry c_{i+1} is related to its carry input c_i as follows:

$$c_{i+1} = x_i y_i + x_i c_i + y_i c_i$$

$$= x_i y_i + (x_i + y_i) c_i$$

$$= G_i + P_i c_i$$

where $G_i = x_i y_i$ and $P_i = (x_i + y_i)$

G_i is called carry generate function since a carry is generated when $x_i = y_i = 1$.

P_i is referred as carry propagate function since if x_i or y_i is 1 then the input carry c_i is propagated to the next. Using G_i and P_i, c_1, c_2, c_3 and c_4 can be expressed

as follows :→

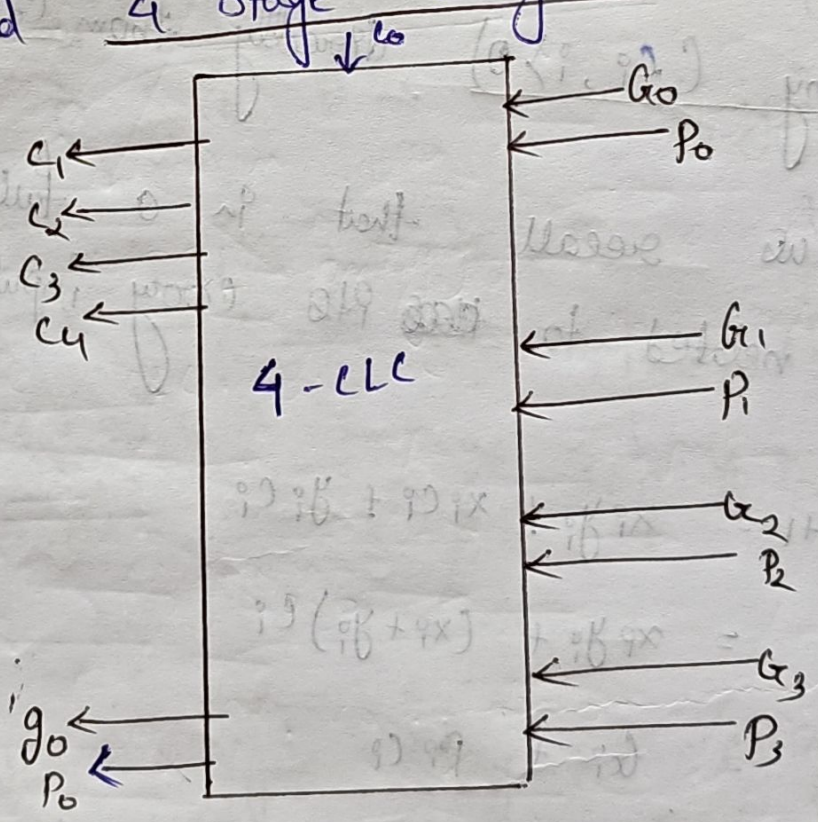
$C_1 = G_0 + P_0 C_0$

$C_2 = G_1 + P_1 C_1 = G_1 + P_1 (G_0 + P_0 C_0) = G_1 + P_1 G_0 + P_1 P_0 C_0$

$C_3 = G_2 + P_2 C_2 = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_0$

$C_4 = G_3 + P_3 C_3 = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 C_0$

The above results suggest that C_1, C_2, C_3, and C_4 can be generated directly from C_0, thus these equation are called Carry - look - ahead equations, and the hardware that implement these equations is called 4 stage carry look ahead circuit (4CLC)



$g_0 = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0$

$P_0 = P_3 P_2 P_1 P_0$.

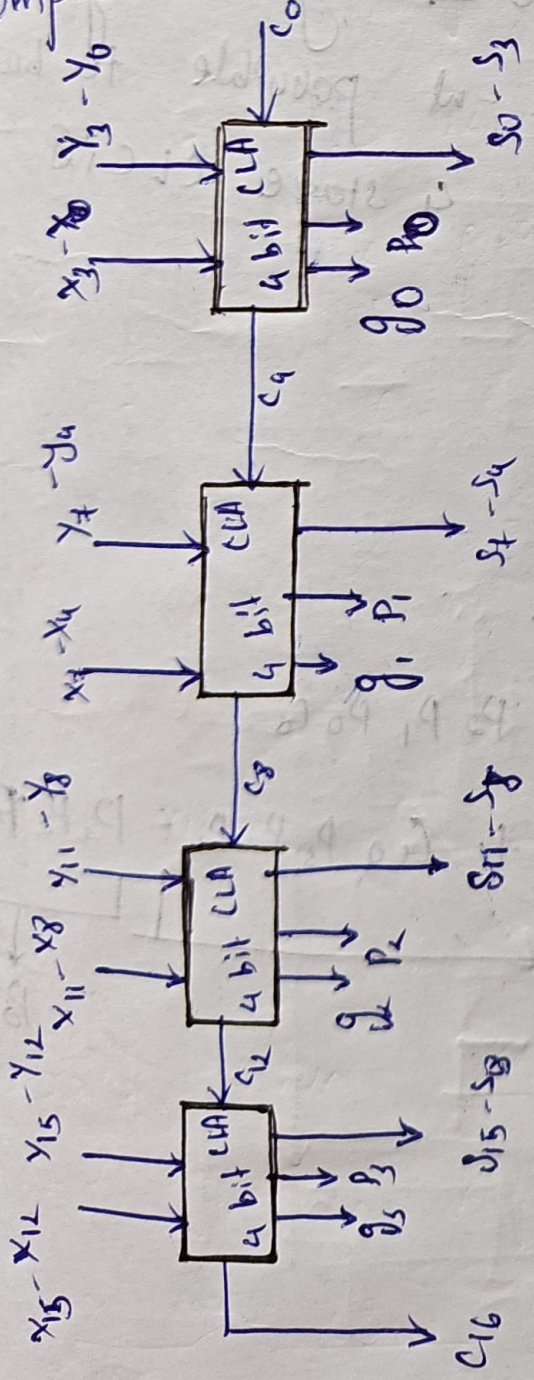
A 4-CLC can be implemented as a two-level AND/OR logic circuit (First level comprises AND gates and second level comprises OR gates) then needs 2 gate delays.

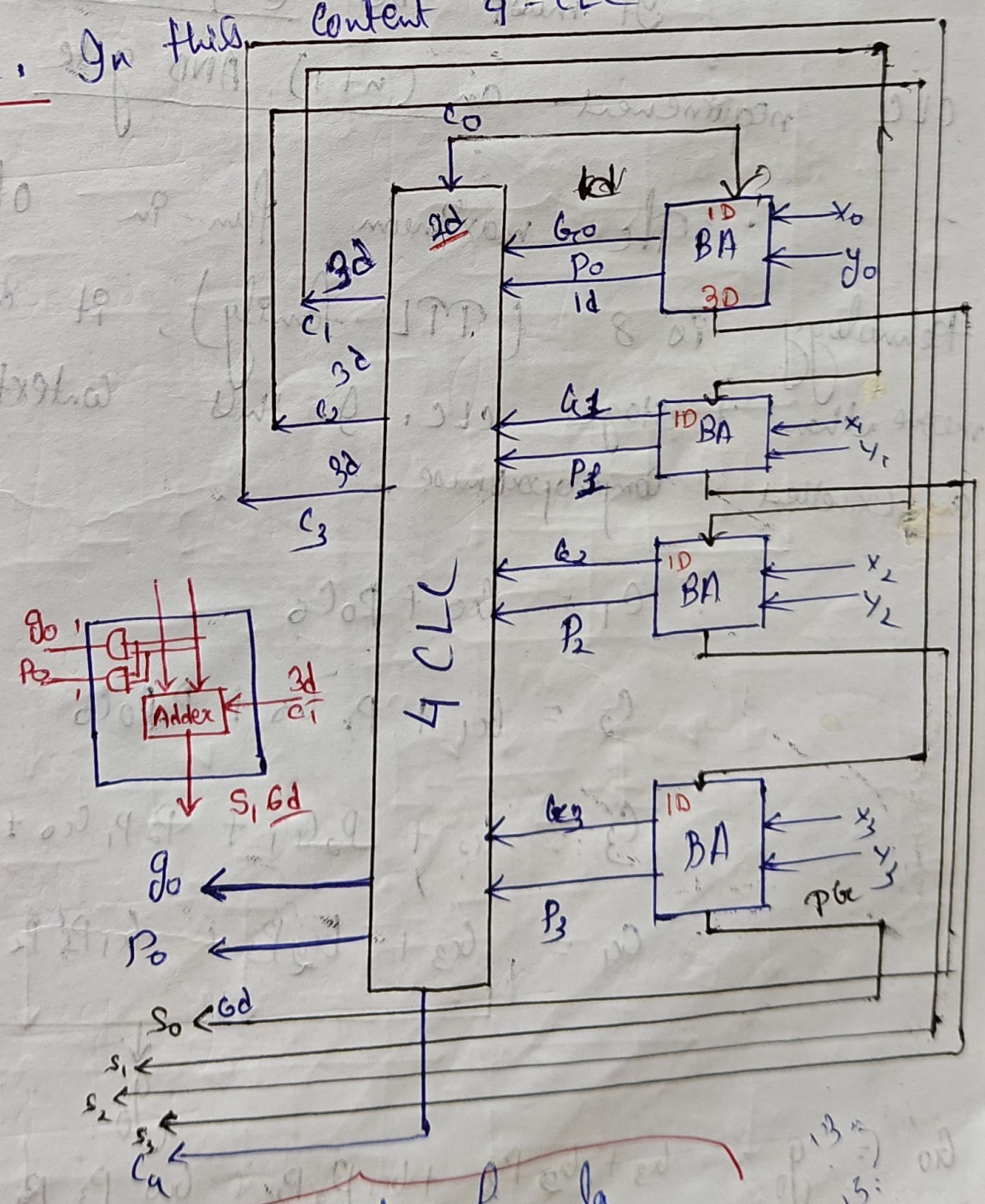
All G, P points are generated parallelly from.

x_i y_i with a need of 1 gate delay.

In particular the fan-in requirement for 4-CLC realization is 5 (5 input AND gate, to generate $P_3 P_2 P_1 P_0 C_0$ and to generate C_4 from C_0 a 5 input OR gate is needed)

It must be emphasized that realization of n-CLC requires an n+1 input AND gates and an n+1 input OR gate. Since the maximum fan-in offered by existing technology is 8, it is not possible to build more than 7-CLC. In this content 4-CLC is an excellent compromise.





n bit delay formula

~~n-D = 2d + [n/4] x 2d + 3d~~

$$\boxed{1d + \left\lceil \frac{n}{4} \right\rceil \times 2d + 3d}$$

13
15
17
19

fan out → max number of input that one gate drive

fan in → max number of input that one gate can have.

If we have n stage C-L-C, the gate delay remain 6d for n bit adder. where n could be large

In particular the fan-in requirement for 4-bit realisation is 5 (5 input and gate to generate $P_3 P_2 P_1 P_0 C_0$ & 5 input OR-gate is required to generate C_4 from C_0

It must be emphasized that realisation of n-CLC requirement an (n+1) AND gate and (n+1) OR gate

∵ The maximum fan-in offered by existing technology is 8 (TTL family), it is not possible to build more than 7 stage CLC. In this context, 4 stage CLC is an excellent compromise.

$$C_1 = G_0 + P_0 C_0$$

$$C_2 = G_1 + P_1 G_0 + P_1 P_0 C_0$$

$$C_3 = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_0$$

$$C_4 = \underbrace{G_3 + G_2 P_3 + G_1 P_3 P_2 + G_0 P_3 P_2 P_1}_{g_0} + \underbrace{P_3 P_2 P_1 P_0}_{p_0} C_0$$

$$G_0' = g_0 = G_3 + G_2 P_3 + G_1 P_3 P_2 + G_0 P_3 P_2 P_1$$

$$P_0' = p_0 = P_3 P_2 P_1 P_0$$

$G_1' = g_{..} = G_7 + G_6 P_7 + G_5 P_7 P_6 + G_4 P_7 P_6 P_5$

$$C_1 = G_0' + P_0' C_0$$

$$= G_9 + G_2 P_3 + G_1 P_3 P_2 + G_0 P_3 P_2 P_1 + P_3 P_2 P_1 P_0 C_0$$

$$= C_9$$

$$C_2' = G_1' + G_0' P_1' + P_1' P_0' C_0$$

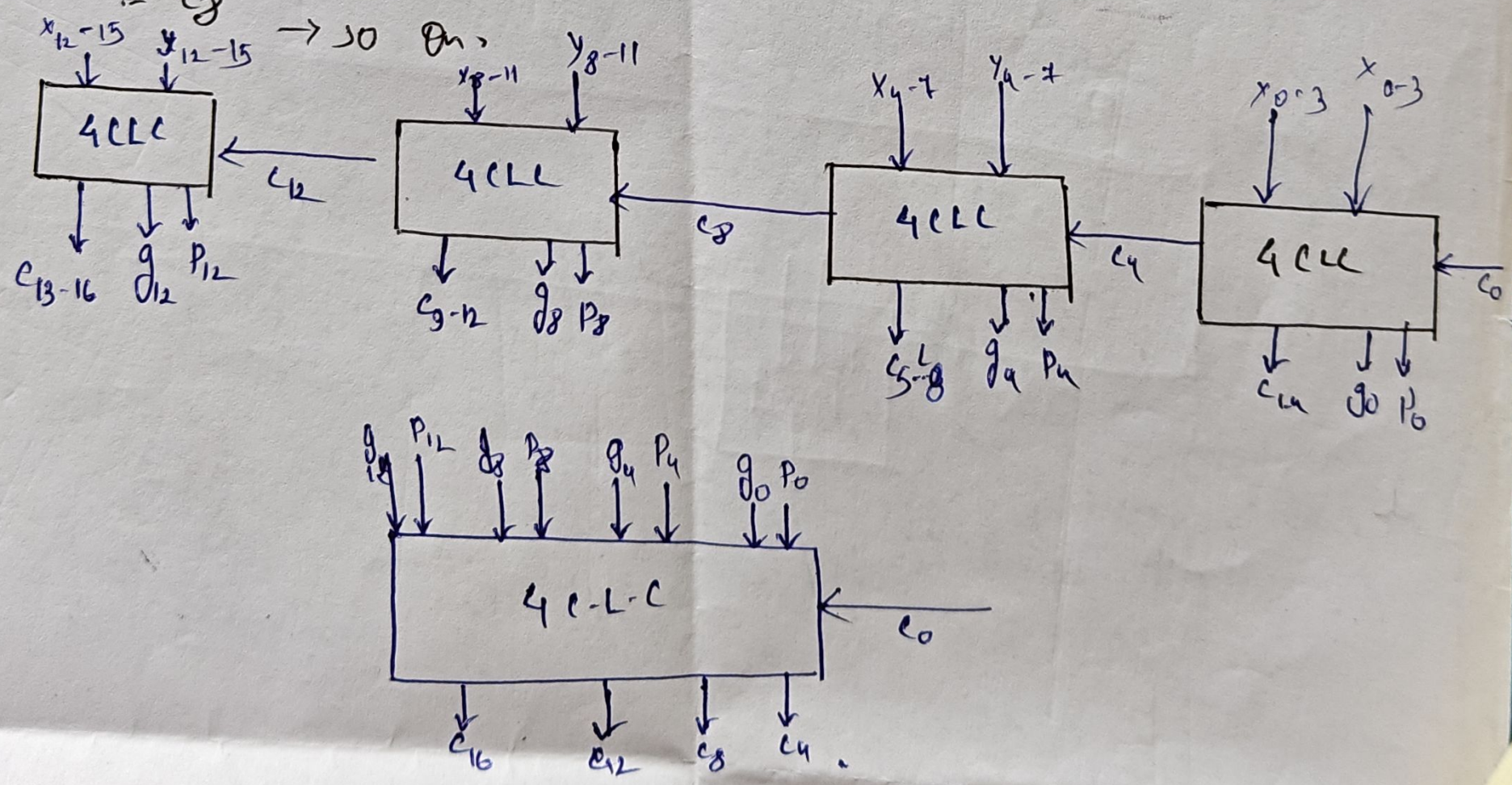
$$= (G_7 + G_6 P_7 + G_5 P_7 P_6 + G_4 P_7 P_6 P_5)$$

$$+ (G_3 + G_2 P_3 + G_1 P_3 P_2 + G_0 P_3 P_2 P_1) P_7 P_6 P_5 P_4$$

$$+ (P_7 P_6 P_5 P_4)(P_3 P_2 P_1 P_0) C_0$$

$$= (G_7 + G_6 P_7 + G_5 P_7 P_6 + G_4 P_7 P_6 P_5) + (G_3 P_7 P_6 P_5 P_4 + G_2 P_7 P_6 P_5 P_4 P_3 + G_1 P_7 P_6 P_5 P_4 P_3 P_2 + G_0 P_7 P_6 P_5 P_4 P_3 P_2 P_1 P_0 C_0$$

$$\doteq C_8$$



Q) Find out the delay using 4 bit des using single layer des :->

$$n\,th = 1d + \left(\frac{n}{4}\right) \times 2d + 3d$$

$$= 1d +$$

$G_1' = g_4 = G_7 + G_6 P_7 + G_5 P_7 P_6 + G_4 P_7 P_6 P_5$

$P_4' = P_7 P_6 P_5 P_4$

$G_2' = g_8 = G_{11} + G_{10} P_{11} + G_9 P_{11} P_{10} + G_8 P_{11} P_{10} P_9$

$P_2' = P_8 = P_{11} P_{10} P_9 P_8$

$G_3' = G_{15} + G_{14} P_{15} + G_{13} P_{15} P_{14} + G_{12} P_{15} P_{14} P_{13}$

$P_3' = P_{15} P_{14} P_{13} P_{12}$

Carry save addition :→

If there is need to add more than two operands, a technique known as carry save addition or carry select addition is used. To see the effectives let us consider this example →

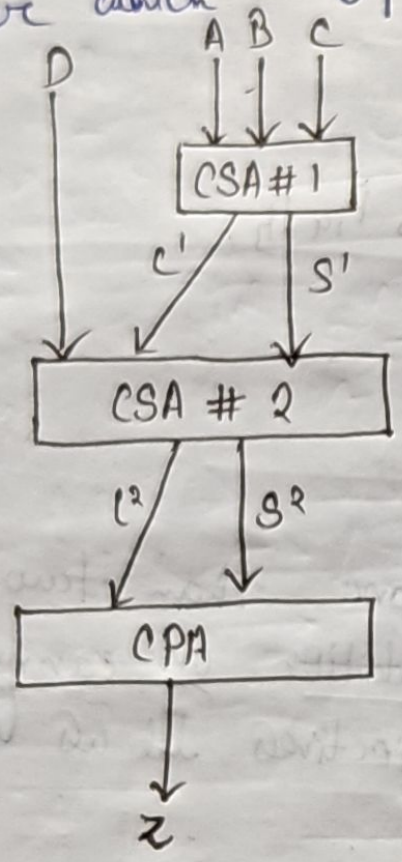
```
   44
   28
   32
   79
  ----
   63
  12
  ----
  183
```

In this example 4 two digit decimal numbers are added. First the unit digits are added producing sum digit 3 & carry digit 2. Similarly the tens digit are added producing sum digit 6 and carry digit 1.

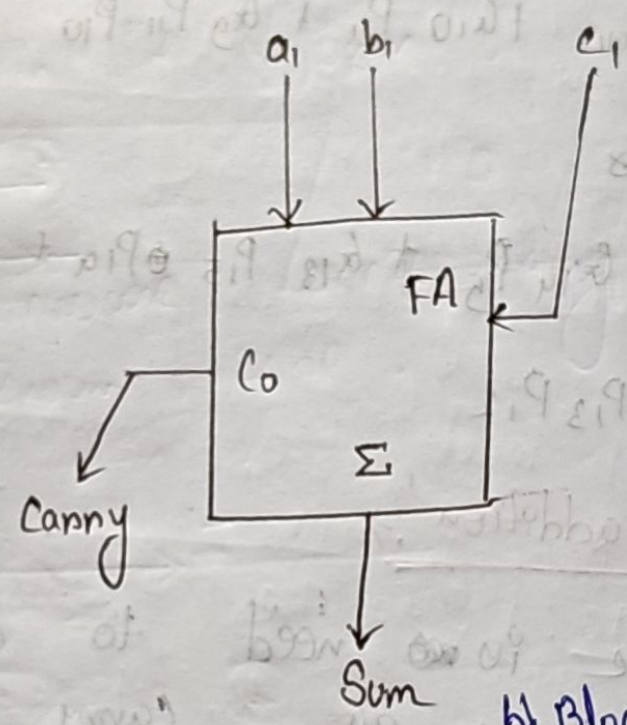
Since there is no carry propagation, these summations can be carried out in parallel to produced sum vector 63 & carry vector 12.

When all operands are exhausted the sum vector & shifted carry vector (1 position left shift) are added. In a convential manner which produced final result 183.

When all operands are exhausted the sum and shifted carry vectors are added in the conventional manner which product the final result.



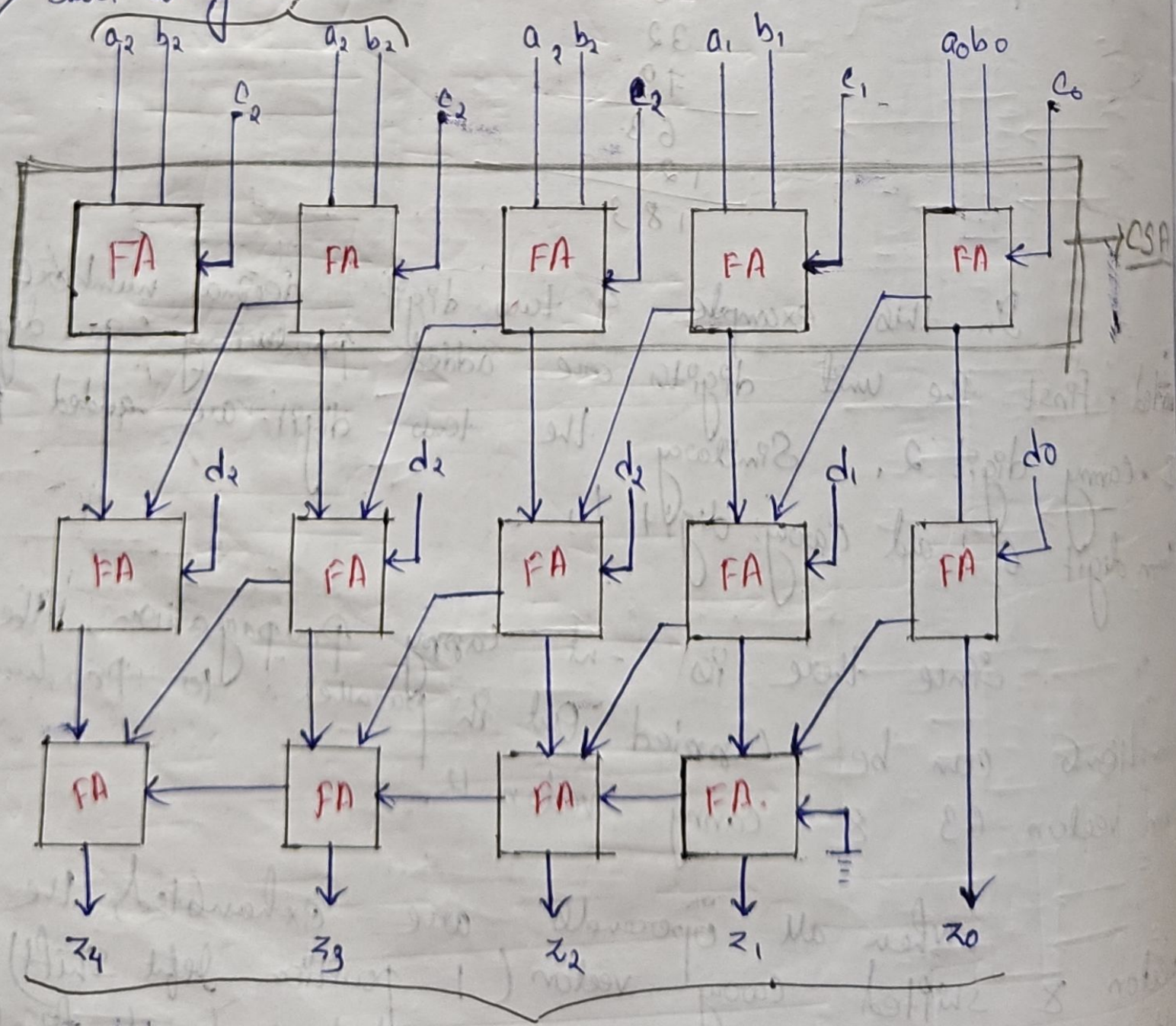
a) Block Diagram



b) Block cell

Sign-extended bits.

c) Hardware Semantic



Final Answer.

Let us consider the example $Z = A + B + C + D$ where A, B, C, D and D are 3 bit 2's complement number. The block diagram for this summation process sum vector's and carry vector's.

The sum vector S' and shifted carry vector C' and fourth operands D are applied to second CSA. The result produced by second CSA are finally fed by CPA generate Z. The carry propagate only in the last step. So the total time require to add 4-operators is →

$$\boxed{T(4) = 2 * (\text{CSA add time}) + (\text{CPA time})}$$

In general, n operands can be added is

$$\boxed{T(n) = (n-2) * (\text{CSA add time}) + (\text{CPA time})}$$

The result can further improve by using CLA in the last stage replacing CPA.

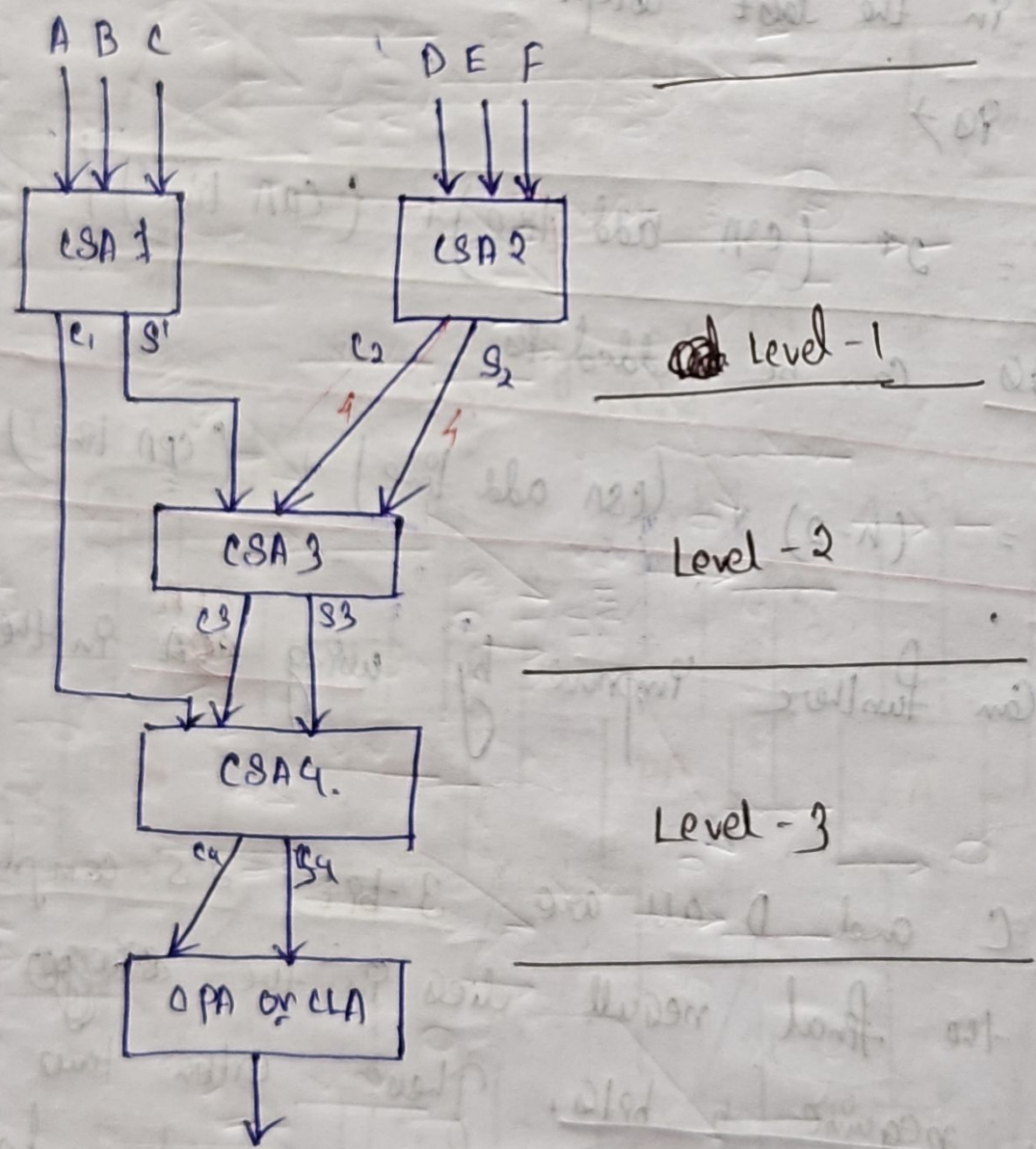
Note → A, B, C and D all are 3-bit 2's complement number. However the final result lies in the range -16 to +12, then Z require 5 bits. These extra two bits are called Guard Bits. Thus, form first level it will be treated an 5-bit addition. Last CPA/CLA will be of 4-bit since least significant bit (LSB) does not participate in last stage addition.

No of required guard bit plays an important role in size of adder as well as time.

No of required guard bits for n operants ⇒ $\lceil \log_2 n \rceil$

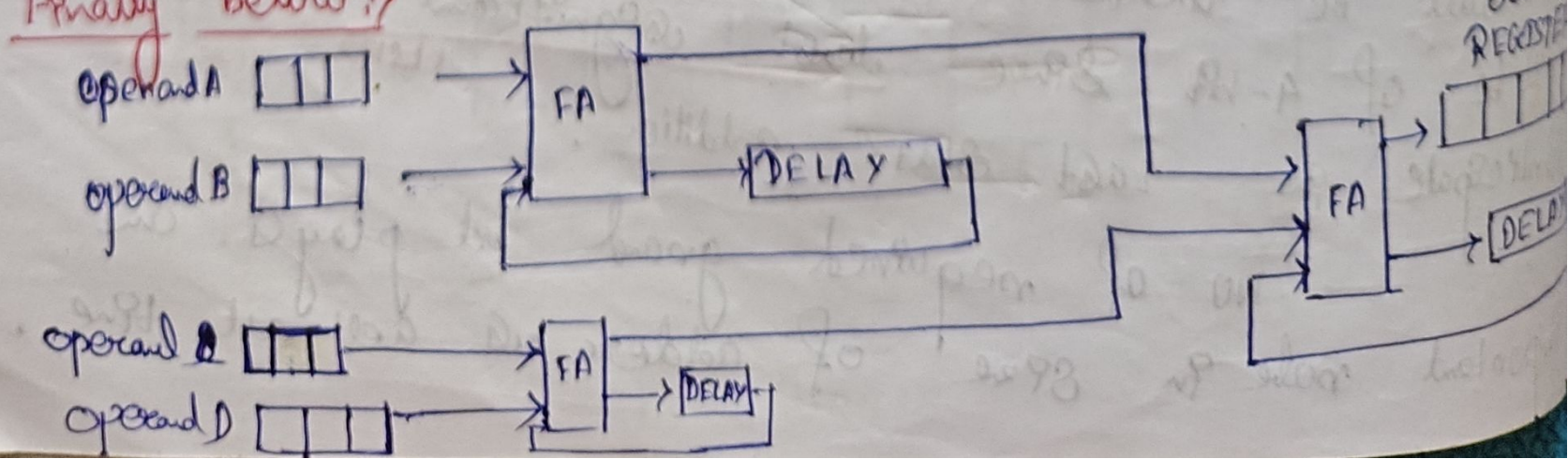
The no of CSA stages can be reduced the CSAs are organized in the form of a tree, popularly known as <u>Wallace Tree</u>.

The tree structure corresponding to a 6-operand summation is shown beside.



$z = A + B + C + D + E + F$

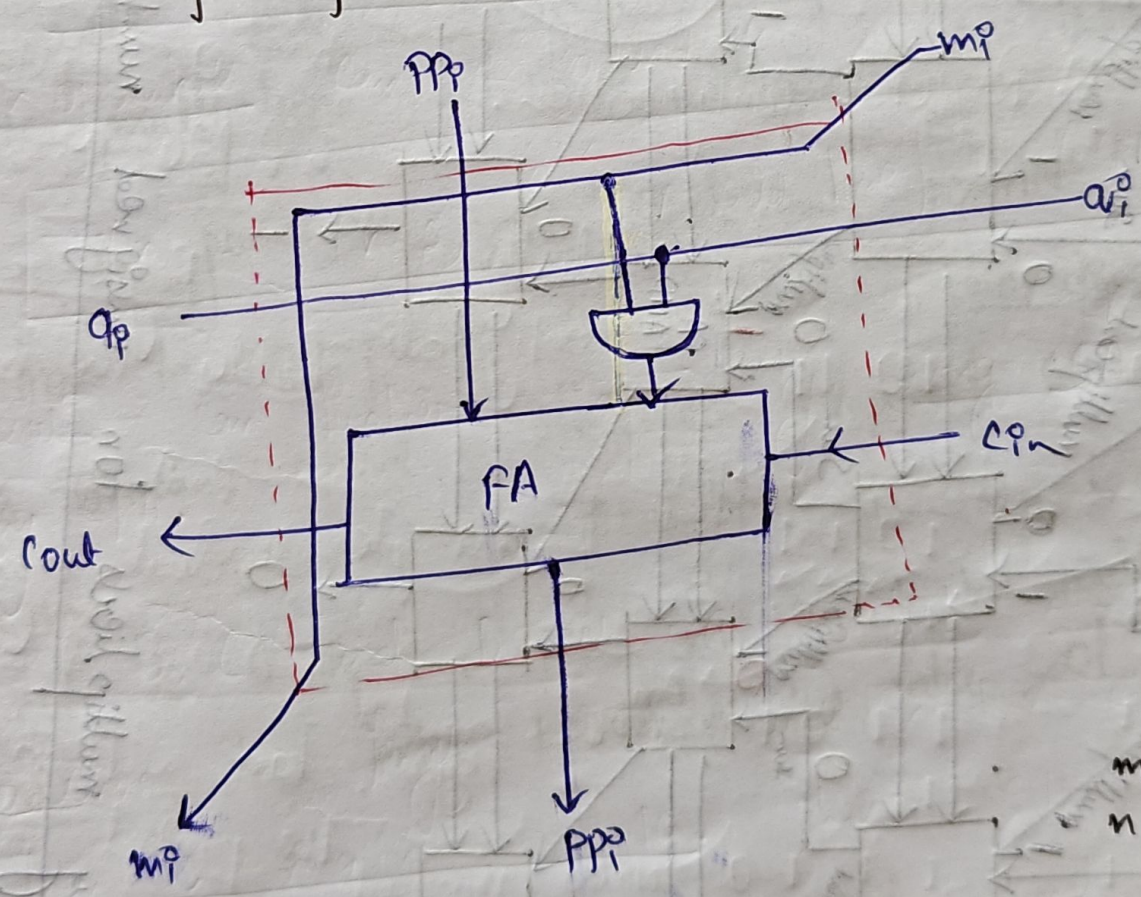
Finally below :-



1101 → multiplicant (M)
1011 → multiplier (Q)

```
  1101   → PP1
  1101
 1001111 → PP2
 0000xx
 1001111 → PP3
 1101xxx
10001111 → PP4.
```

Paper pencil method of multiplication

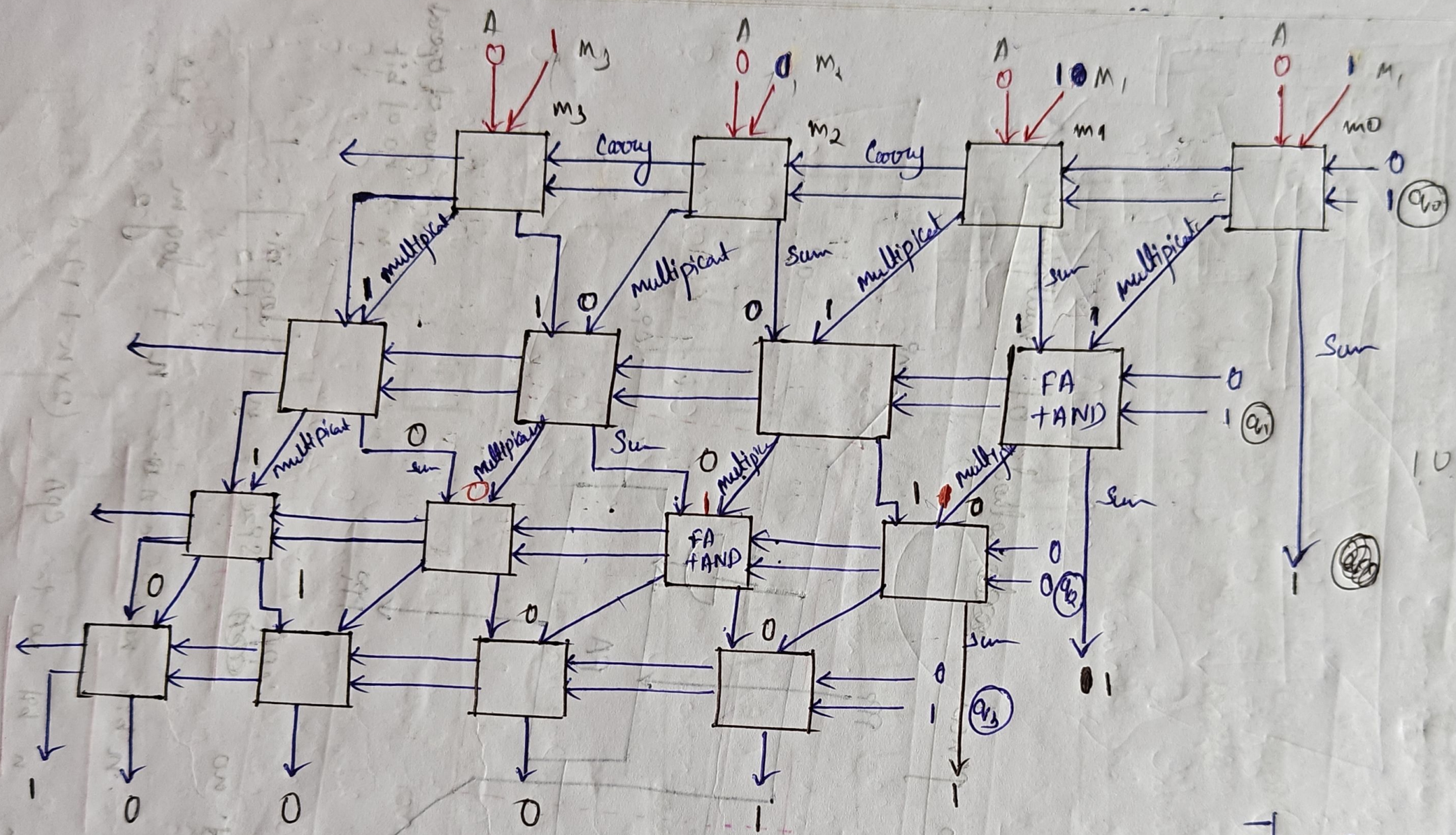


m → no of operand
n → no of bit

CPA size = $n + \lceil \log_2 m \rceil - 1$

n bit no in CSA = $n + \log_2 m$

no of adder in n bit no in CPA = $(2 \times n + 1)$



Array multipliers for unsigned number

Extra

$$\begin{array}{r} 1\,1\,1\,0 \\ 1\,0\,1\,0 \\ \hline 0\,0\,0\,0 \\ 0\,0\,0\,0\,x \\ 0\,0\,0\,x\,x \\ 0\,0\,x\,x\,x \\ 0\,x\,x\,x\,x \\ x\,x\,x\,x\,x \\ \hline 0\,0\,0\,1\,1\,1\,0 \end{array}$$

Multiplication of positive Numbers :→

The usual papers pencil algorithm for multiplication of integer represented in any positional number system is illustrated below for binary system, assuming two 4-bit operands.

The producing of two n-digit number can be accomodated in 2n digits, so the product in this example first first 8-bits

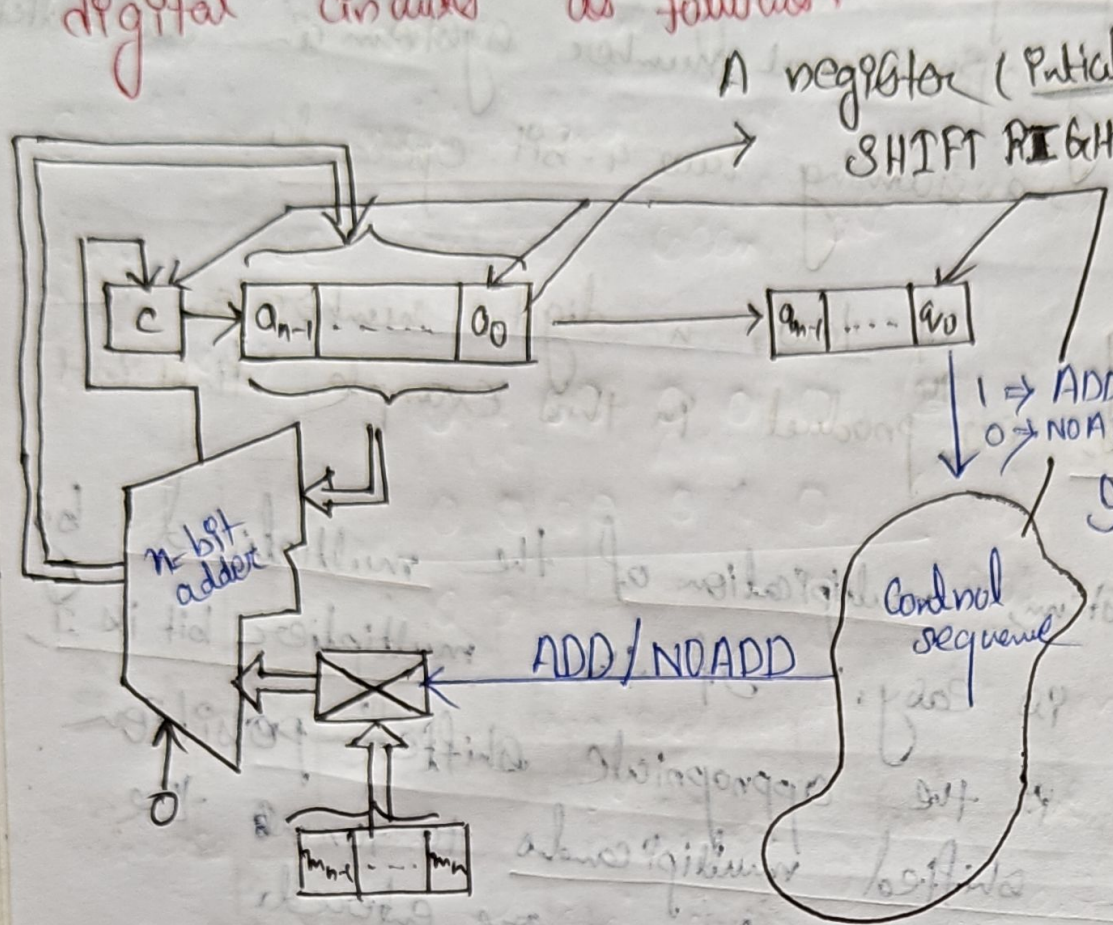
In the binary system, multiplication of the multiplicand by 1 bit of the multiplier is easy. If the multiplier bit is 1, the multiplicand is entered in the appropriate shifted position for addition with other shifted multiplicands to form the product. If the multiplier bit is 0, then 0's are entered.

It is possible to implement positive operand binary multiplication in a purely combinational two-dimensional (2-D) logic array as shown above. The main component in each cell is an Adder ~~as~~ ~~shown~~ circuit, the AND gate in the cell determines whether or not a multiplicand bit m_i in to be added to the incoming partial product bit, based on the value of the multiplier bit q_j.

Although the above combinational multiplier is quite easy to understand but it is usually impractical to use in general-purpose computer because of its cost. There are two solution →

1. Computer don't support exclusive ~~impractical~~ instruction for multiplication or division, i.e they do not support hardware circuit, implemented by loop programming

2. Provide hardware multiplication and division by sequential digital circuits as follows.



a) Register configuration.

This circuit performs multiplication by using simple adder of size n, for n times.

Initially,

Register A of size n bits set to zero.

Counter bit cn cleared to be 0. Register Q of size n holds the multiplier.

Register M of size n holds the multiplicand.

C, A, and Q are all three individual registers, however C, A and Q are connected for shift right to hold the partial product after n steps, multiplier gets lost and registers Q0 and Q0 together hold the final 2n-bit multiplication result. A has upper n-bit and Q has lower n-bit.

	M		
	1 1 0 1		
0	0 0 0 0	1 0 1 1	→ Initially Configuration
C	A		
			→ Add shift
0	1 1 0 1	1 0 1 1	→ First cycle
0	0 1 1 0	1 1 0 1	
			→ Add shift
1	0 0 1 1	1 1 0 1	→ Second cycle
0	1 0 0 1	1 1 1 0	
			→ No add shift
0	1 0 0 1	1 1 1 0	→ Third cycle
0	0 1 0 0	1 1 1 1	
1	0 0 0 1	1 1 1 1	
0	1 0 0 0	1 1 1 1	
	R		

Signed operation multiplication →

According to multiple rule if both multiplicand and multiplier are negative the result will be as good as both are positive.

Next let us consider the case of a positive multiplier and a negative multiplicand, when we add a negative multiplicand to a partial product, we must extend the sign bit value of the multiplicand to the left as far as extend of the eventual product.

Let us consider the example of multiplying -13 (multiplicand) with +11 (multiplier) below, where sign extension of the multiplicand do shown underline.

```
      10011   (-13)
      01011   (+11)
  ----------
  11111 10011
  11111 10011x
  00000 00000xx
  11100 11xxx
  000000 xxxx
  ----------
  11011 10001 (-143)
```

Significance of any number be positive or negative does not change on sign extension.

Here the partial products are negative, every partial product needs to be represented with sign extension to the size of 10-bits.

(Here multiplier cannot be negative)

Booth Algorithm :→

A powerful direct algorithm for signed-number multiplication is the Booth algorithm. It generates a 2n-bit Product and treat positive and negative number uniformly.

Let us consider a multiplication operation in which a positive multiplier has a single block of 1s

with at least 0 at each end for example 0011110. $(30)_{10}$ To derive the product we need to add four opportunely shifted multiplicand as in the standard process. However this case can be seen as difference of two numbers as follows :→

$$
\begin{array}{rl}
 & 0100000 \quad (32_{10}) \\
- & 0000010 \quad (2_{10}) \\
\hline
 & 0011110 \quad (30_{10})
\end{array}
$$

This suggests that the final product result can be achieved by one addition and one-substraction. For convience now onward we will represent the standard scheme by writing the multiplier as 0 0 +1 +1 +1 +1 0 and the new recoding scheme by writing the multiplier as.

0 +1 0 0 0 −1 0.

Note → the recoding scheme as (scan from right to left)

0 → 0	0	Consider 0 at the extreme right
0 → 1	−1	01101 equivalent +1 0 −1 +1 −1
1 → 0	+1	(multiply by −1 produces 2's complement
1 → 1	0	of multiplicand)

Normal & Booth multiplication scheme $(45_{10} \times 39_{10})$

Booth multiplication can handle negative multiplier as well, in 2's complement obviously.

0 1 1 0 1 (HB)

Normal method

										0	1	1	0	1	
								0	1	0	1	1	0	1	$(45)_{10}$
								0	0	1	1	1	1	0	$(30)_{10}$
							0	0	0	0	0	0	0	0	
						0	1	0	1	1	0	1	x		
					0	1	0	1	1	0	1	x	x		
				0	1	0	1	1	0	1	x	x	x		
			0	1	0	1	1	0	1	x	x	x	x		
		0	0	0	0	0	0	0	x	x	x	x	x		
	0	0	0	0	0	0	0	x	x	x	x	x	x		
0	0	0	0	0	0	0	x	x	x	x	x	x	x		
0	0	0	0	1	0	1	0	1	0	0	0	1	1	0	

Booth Algonitm.

								0	1	0	1	1	0	1	
								0	1	0	0	0	-1	0	
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
1	1	1	1	1	1	1	0	1	0	0	1	1	x		
0	0	0	0	0	0	0	0	0	0	0	0	x	x		
0	0	0	0	0	0	0	0	0	0	0	x	x	x		
0	0	0	0	0	0	0	0	0	0	x	x	x	x		
0	0	0	1	0	1	1	0	1	x	x	x	x	x		
0	0	0	0	0	0	0	0	x	x	x	x	x	x		
0	0	0	1	0	1	0	1	0	0	0	1	1	0		

00100101
+1
10100111

```
0 1 1 0 1   (13)          0 1 1 0 1
1 1 0 1 0   (-6)   →      0 -1 +1 -1 0
```

```
                          0 1 1 0 1
                          0 -1 +1 -1 0
                          ---------------
                          0 0 0 0 0
                        1 1 1 1 1 0 0 1 1   → 2's complement multiplication
                      0 0 0 1 1 0 1
                    0 0 0 1 1
                  1 1 1 0 0 1 1
                  ---------------------------
                  0 0 0 0 0 0 0 0 0 1 1 0
                  1 1 1 0 1 1 0 0 1 1 0
```

Booth algorithm achieves two purpose :-

1/ uniformly transforms booth positive and negative n-bit multiplicn.

2/ ~~achieves~~ achives some efficiency in the number of summands generated when the multiplier has few long boolean of 1s.

Booth speed-up technique :-

Now let us describe a multiplication speed-up technique which guarantees that an n-bit multiplier will generate at most n/2 summands and uniformly handles signed-operand cases.

This provides a multiplication speed-up by a factor of 2 over the worst case Booth algorithm situation.

The new technique is derived from the Booth

Technique where multiplier bit q_i selects a summand as a function of the bit q_{i-1} on its right, the summand selected is shifted i binary position to the left of the LSB position of the product before the summand is added.

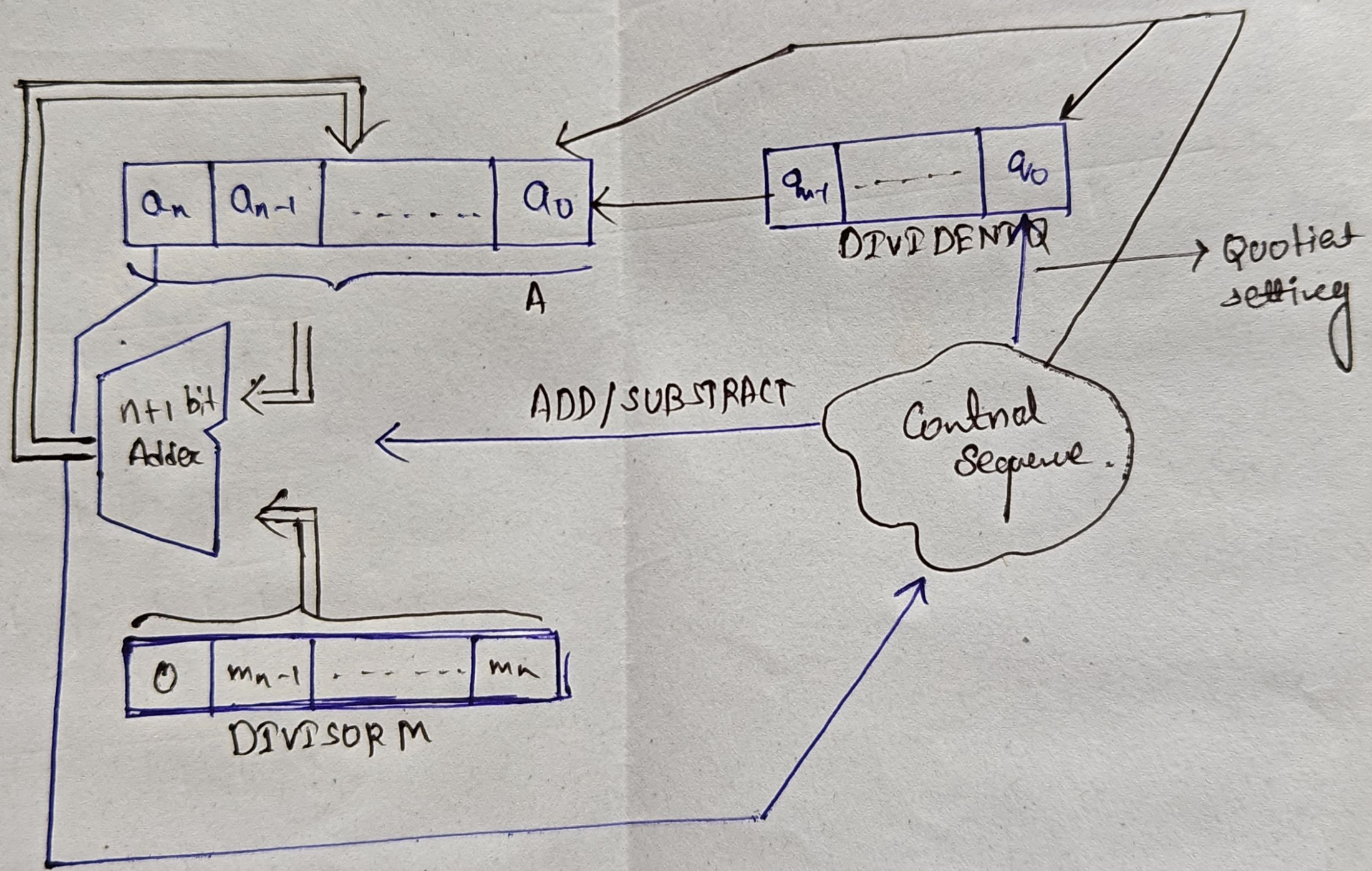
The basic idea of the speed up is that bits $i+1$ and i select one summand as a function of bit $i-1$ to be added for bit i. Thus there will be $n/2$ summands for n-bit multiplier.

Sign → (1) 1 1 0 1 0 (0) ← Implied 0 and the right of LSB.

extension → 0 0 (−1 +1) (−1 0)

0 −1 −2

Multiplier bit pairs q_{i+1}	q_i	multiplier bit on the right q_{i-1}	multiplicand selected at ith position
0	0	0	0 × M
0	0	1	+1 × M
0	1	0	+1 × M
0	1	1	+2 × M
1	0	0	−2 × M
1	0	1	−1 × M
1	1	0	−1 × M
1	1	1	0 × M


a_n
a_{n-1}
a_0
A
q_{n-1}
q_0
DIVIDENDQ
Quotient setting
n+1 bit Adder
ADD/SUBSTRACT
Control Sequence
0
m_{n-1}
m_n
DIVISOR M

Integer division :→

```
        021
13 | 274   (Dividend)
     26
     ----
      14
      13
     ----
       1 → Reminder
```

(Devicer)

~ Decimal division ~

```
        00001010 1
1101 | 10001 0010
       1101
       -----
       ØØ10000
        1101
       -----
       ØØØ1110
         1101
         ----
         ØØØ1
```

~ Boolean in division ~

* Restone in type division,

The important point to conclude is that the process of trial determination of the Quocent digit is difficult to automate in a logic circuit.

The simplest circuit that implement binary division must methodically position of division with respect to the divident and perform a substruction. If the result of substruction zero or positive a Quotient bit of 1 is determine. The reminder is extended another bit of the devident, the division re-position and another substraction in perform. The another hand, is the reminder is -ve a Quotient bit of zero is determine. And divident restone adding back. The division re-position for the substruction.

An n-bit positive integer divison M loaded register M and n-bit Positive dividend in loaded into register Q at the start of the operation. Register A is to 0.

After the division is Complete

, the n-bit quotent will be 3n register Q and the remainder in A. The required substrions are 2's comp arithmatic.

The extra bit position at the left end of both A and M in for the sign bit for there substraction.

*Restoring algorithm →

Do n times

- Shift A and Q left one binary position.
- Substract M from A, placing the answer back in A.
- If the sign of A is 1, set q_0 to 0 and add M back to A otherwise, set q_0 to 1.

It is possible to improve on this algorithm by adding the need for restoring A after an unsuccessful substraction.

Let us consider the ~~sequential~~ sequence of operation that takes place after substruction operation in restoring algorithm.

If A is positive, we shift it left and substract M; that is we are perform 2A − M.

If A is negative, we restore it by performing A+M and then we shift it left and substract M, that is equivalent perform 2A+M.

The q_0 bit to appropriately set to 0 and 1.

Non-Restoring Algorithm

S_1: Do n times

Shift A and Q left one binary position. If sign of A is 0, substract M from A otherwise add M to A. Next if sign of A is 0, set q_0 to 1 otherwise set q_0 to 0

S_2: If the sign of A is 1, add M to A.

Example :→ Restoring & Non-restoring

Restoring

Intially

	A	Q
	0 0 0 0 0	1 0 0 0
M	0 0 0 1 1	
Shift	0 0 0 0 1	0 0 0 □
Substra (A-M)	1 1 1 0 1	
set q_0	1 1 1 1 0	
Restore	0 0 0 1 1	
	0 0 0 1 1	0 0 0 0

→ First cycle

Shift	0 0 0 1 0	0 0 0 □
Substract (A-M)	1 1 1 0 1	
set q_0	1 1 1 1 1	
Restore	0 0 0 1 1	
	0 0 0 1 0	0 0 0 0

→ 2nd cycle.

Shift	0 0 1 0 0	0 0 0 □
Substract (A-M)	1 1 1 0 1	
set q_0	0 0 0 0 1	0 0 0 1

→ Third.

Shift	0	0	0	1	0	0	0	1	□
Substract A-M	1	1	1	0	1				
seq 0	1	1	1	1	1				
Restore				1	1	0	0	1	0
	0	0	0	1	0	0	0	1	0

Remainder — Quotient

Non-restoring

		A	Q
Intially	M 0 0	0 0	1 0 0 0
	A 0 0	0 1 1	
Shift	0 0	0 0 1	0 0 0 □
Substract (M-A) set q0 →	1 1	1 0 1	0 0 0 0
	1 1	1 1 0	
Shift	1 1	1 0 0	0 0 0 □
Add (M+A) set q0	0 0	0 1 1	0 0 0 □
	1 1	1 1 1	
Shift	1 1	1 1 0	0 0 0 □
Add (M+A) set q0	0 0	0 1 1	0 0 0 1
	0 0	0 0 1	
shift	0 0	0 1 0	0 0 1 □
Substrat set q0	1 1	0 1 0 1	0 0 1 0
	1 1	1 1 1	

First cycle → Substract → Third cycle → Fourth cycle

Quotient

Add.

$$\begin{array}{ccccc} 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 \\ \hline 0 & 0 & 0 & 1 & 0 \end{array}$$

→ Positive remainder.

Reminder

Floating point Number Representation

> Floating-point representation of a number has two parts.

→ 1) Mantissa. a signed fixed-point number (may be a fraction or an integer).

2) Exponent, designates the position of the decimal point.

3) If the mantissa is m & exponent is e then the two parts representation a number obtained from multiplying m times a radix r raised to the value of e; thus $m \times r^e$.

> Example :→ Assume a fraction representation and a radix 10. The decimal number 537.25 is represented in a register with m = 53725 and e = 3 and is interpreted to represent the floating point number $.53725 \times 10^3$.

> Purpose → It increase the range of numbers that can be accommodated in a given register.

Normalize of Floating point Number :->

> A floating point number is normalized if the most significant digit of mantissa is non zero.

> Mantissa contains the maximum possible number of significant digits.

> A zero cannot be normalize because it doesn't have a non-zero digit.

> It is represent in floating point by all 0's in the mantissa & exponent.

Type of Representation of Floating point Numbers :->

> IEEE standard for floating point number :->

1. IEEE Single precision format.
2. IEEE Double precission format.

IEEE Standards for Floating point Numbers :->

1. IEEE Single precision format (32 bits)

> Develope by IEEE.

> Standard describes both the represent & the way in which the four basic arithmatic operations are to be performed.

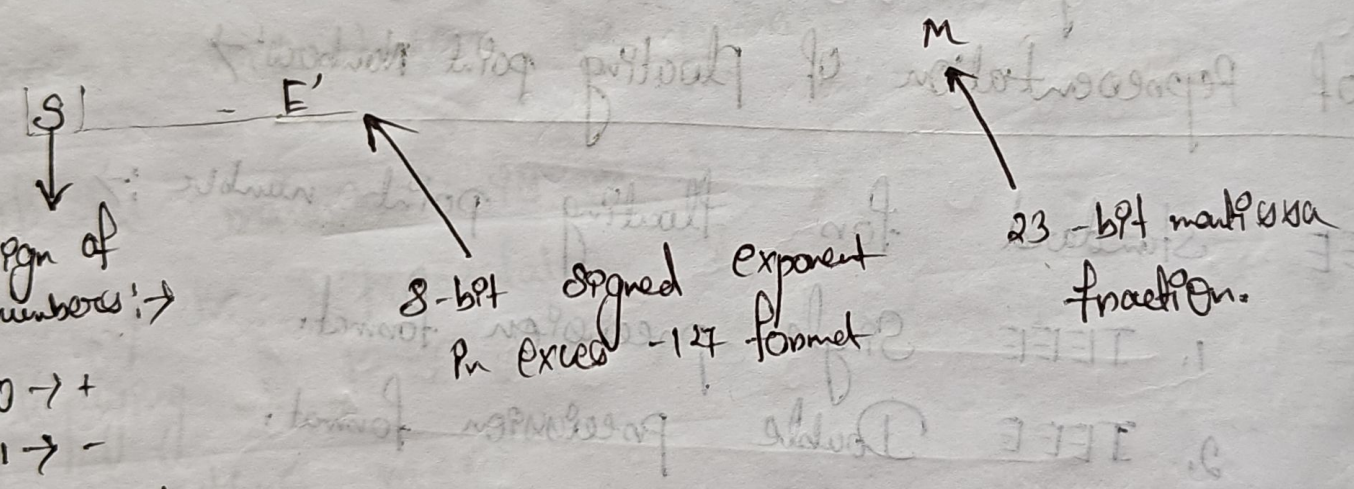
> The sign of the number is given in the 1st bit, followed by a representation for the exponent of 8 bits & last 23 bits representation the mantissa.

> Instead of the signed exponent, E, the value actually stored in the exponent field is an unsigned integer $E' = E + 127$. This is called the excess-127 format.

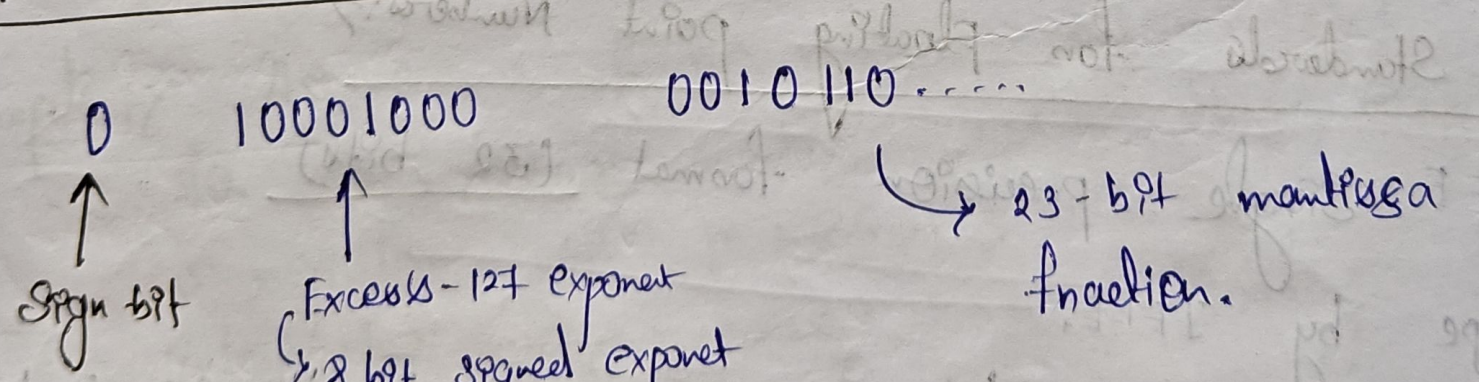
> E' is in the range $0 \leq E' \leq 255$. The end value of this range - 0 and 255, are used to represent special value.

> Therefore, the range of E' for normal value is $1 \leq E' \leq 254$.

> This means that the actual exponent, E, is in the range $-126 \leq E \leq 127$.



* Un-normalized value:→



value representation = $+0.0010110\ldots \times 2^{9}$

* Normalize value:→

value representation = $1.0110..\times 2^6$

- The 32-bit standard representation is called a single Precision representation because it occupies a single 32 bit word.

- Scale factor range: 2^{-126} to 2^{+127}

- The 23 bit mantissa provides approximately the same Precision as a 7-digit decimal value.

- A computer must provide at least single precision representation to conform to the IEEE standard.

IEEE Double precision format (64 bits)

- To provide more precision and range for floating Point numbers, IEEE specifies a double precision format.

- The double precision format has increased exponent & mantissa ranged.

- The sign of the number is given in the 1st bit, followed by a representation for the exponent of 11 bits & last 52 represent the mantissa.

- The 11-bits excess-1023 exponent E' has the range $1 \leq E' \leq 2046$ for normal values with 0 and 2047 used to indicate special values, as before. Thus the actual exponent E is in range $-1022 \leq E \leq 1023$. providing scales factor of 2^{-1022} to 2^{+1023}.

Thus 53-bit mantissa provides a precision equivalent to about 16 decimal digit.

S	E'	M
Sign of number	11-bit signed exponent in excess-1023 format	52-bit mantissa fraction.

— Arithmatic operation on Floating point Numbers —

*** Add / Substract Rule :→**

1. choose the number with the smaller exponent and shift its mantissa right a number of steps equal to the difference in exponent.
2. Set the exponent of the result equal to the large exponent.
3. Perform addition/substraction on the mantissa and determine the sign of result.
4. Normalize the resulting value, if necessary.

*** Multiply rules :→**

(i) Add the exponents & substract 127.

(ii) Multiply the mantissas and determine the sign of the result.

(iii) Normalize the resulting values, if necessary.

* Divide Rules :->

1) Substract the exponent of divisor from the exponent of divident & add 127.

~~2) Divided the mantissas and determine the sign of the result.~~

2) Divide the mantissa of dividend by the mantissa of divisor ~~the~~ and determine the sign the result.

3) Normalize the resulting value,

Methods of Truncation :

1. chopping
2. von Neumann Rounding
3. Rounding.

Chopping :->

> Simplest way to remove the guard bits and mantissa no changes in retained bits.

> Examples :-> Suppose we want to ~~to~~ truncate a fraction from six to three bits by this method. All fractions in the range 0. b-1 b-2 b3 000 to 0. b-1 b-2 b-3 111 are truncated to 0. b-1 b-2 b-3. The error in the 3- bit result ranges from 0 to 0.000111.

* The result of chopping is a biased approximation because the error range is not symmetrical about 0.

* Von-Neuman Rounding :→

* The excess bits are removed and the LSB of the retained bits is set 1 irrespective of the bits which are removed.

*) Error Range → b/w -1 and +1 in the LSB position of the retained bits.

*) It is an unbiased approximation.

Rounding :→

*) It achieves the closest approximation to the number being truncated and is an unbiased technique.

*) Procedure :→ A '1' is added to the LSB position of the bit to be retained if there is a '1' in MSB position of bits being removed. Thus 0. b-1 b-2 b-3 1 ... is rounded to ~~0. b-1 b-2 b-3 is rounded to~~ 0. b-1 b-2 b-3 + 0.001 and 0. b-1 b-2 b-3 0 ... is rounded to 0. b-1 b-2 b-3

*) Error Range :→ The error range is approximately -1/2 to +1/2 in the LSB position of the retained bits.

$$
\begin{array}{ccccc}
0. & b-1 & b-2 & b-3 & 1 \\
0. & 0 & 0 & 0 & 1 \\
\hline
0. & b'_{-1} & b'_{-2} & b'_{-3} & 0
\end{array}
$$

0. b-1 b-2 b-3

Cache Memory

Analysis of a large number of typical programs has shown that the references to memory at any given interval of time to be confined with a few localized areas in memory

The phenomena is known as the property of Locality of Reference.

i) Temporal locality of reference.
ii) Spatial locality of reference.

Temporal locality :→ The current instruction which is being fetched may be need again soon. Ex → loop

Spatial locality :→ The adjacent instruction to current instruction may be need soon. Ex → Every programs

In view of those two properties, while accessing the main memory for any instruction, instead of fetching just one instruction from main memory, several consecutive instructions are fetched together and stored in cache memory.

Cache read policy :→ Read through policy
Read back policy.

Cache write policy :→ Write through policy.
Write back policy.

Read Through:- If the addressed location is not in cache the content of the addressed location is forward to the cache as well as sent to the CPU for execution and remaining consecutive locations are copied to cache along with execution by CPU.

Read back:- The entire block containing the address location is transferred from main memory to cache and then the addressed location is forward to CPU for execution.

Write through:- If the content of the address location needs to be changed during execution, the cache location is changed with the new value as well as the corresponding main memory location is also update accordingly.

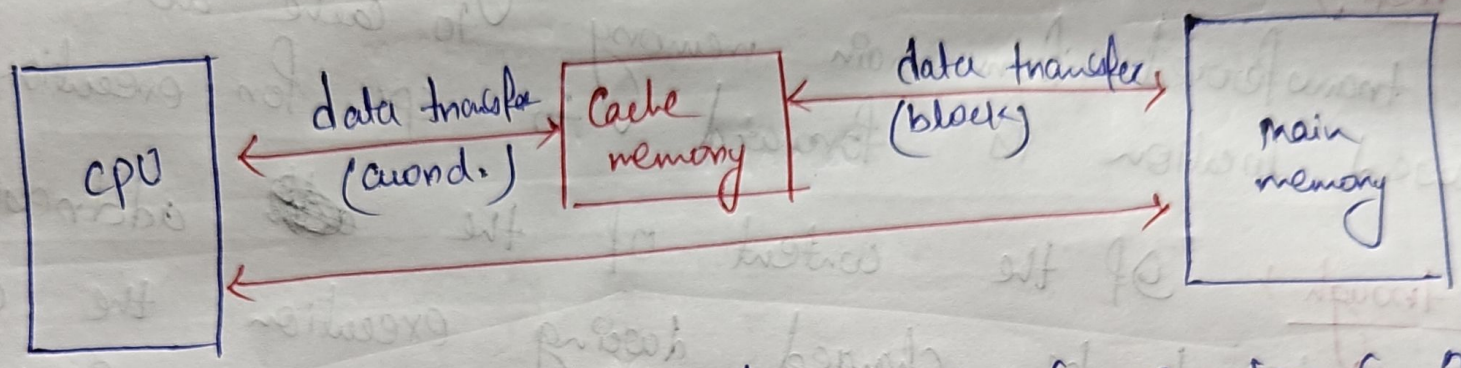
Write Back:- If the content of the address location needs to be changed during execution the cache content is update only and marked for the change. Finally at the termination or for replacement need the entire cache page is copied back to corresponding main memory block. The logic behind is that in many cases the same addressed location is repeatedly update during the stay in cache, we need the last change to be update the main memory.

The basic characteristic of cache memory is its fast access time. Therefore, very little or no time must be wasted. Therefore, very little or no time must be wasted when searching for word in the cache. The transfers of data from main memory to cache memory

is reffered to as a mapping process.

Three type of mapping procedures are of practical interest when considering the organization of cache memory :→

1) Direct mapping
2) Associative mapping
3) Set - Associative mapping



Cache performance → is measured in terms in terms of Hit Ratio

Cache Hit

If the required word is found in cache.

$$\text{Hit Ratio} = \frac{\text{Hits}}{\text{Hit + miss}}$$

$$= \frac{\text{no of Hits}}{\text{Total no of cpu reference}}$$

Cache miss

If the required word is not found in cache.

$$\text{Miss Ratio} = \frac{\text{Misses}}{\text{Hits + misses}}$$

$$= \frac{\text{no of misses}}{\text{Total no of cpu reference.}}$$

Cache Access Time (CHT) :→ Time require to access word form the cache.

Miss Penality :→ Cache miss the penalty

The time require to fech the require block from main memory

***

$$\text{Average Access Time of cpu} = \text{Hit Ratio} \times \text{Cache Access time} + (1 - \text{Hit Ratio}) * \text{miss penalty}$$

$$= h \times T_c + \underbrace{(1-h)}_{\text{miss ratio}} \times T_m.$$

Formulas

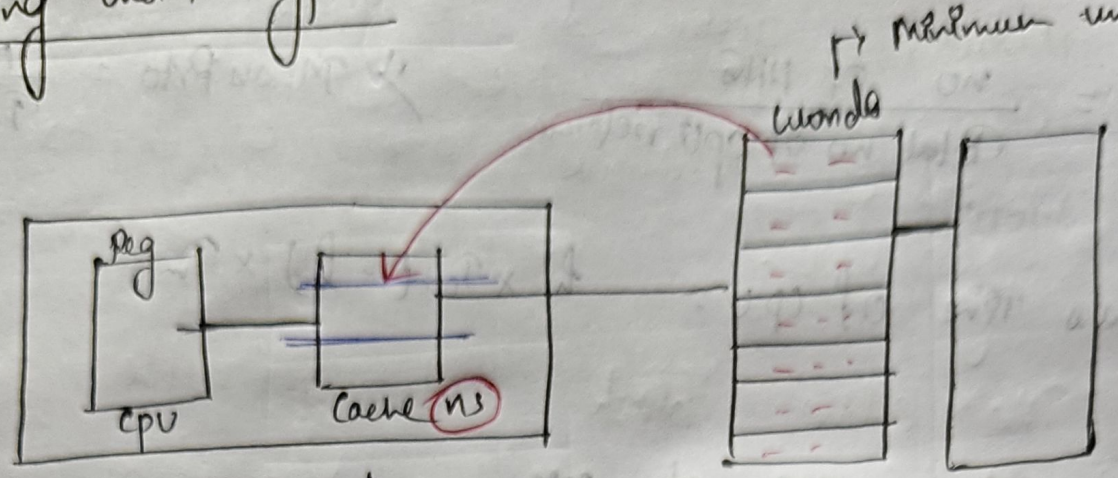
- Hit Ratio (h) = $\frac{\text{no of Hits}}{\text{Total no of cpu reference}}$
- Miss Ratio = $\frac{\text{No of Misses}}{\text{Total no cpu reference}}$
- Average Access Time of cpu = $h \times T_c + (1-h) \times T_m$
- Block no = $\frac{\text{size of MM}}{\text{size of Block}}$
- Page no = $\frac{\text{size of CM}}{\text{size of page}}$
- set no = $\frac{\text{Frame no}}{\text{(no of page include in one set)}}$

Bus → A bus refer to a communication system that transfer data b/w components inside a computer or b/w computers. It's like a highway that allow different parts of a computer to communicate with each other.

Bus

- **Address Bus**
  It is unidirectional bus that carries memory addresses from the CPU to memory modules or other devices. It determine where data should be read from or written to in memory
- **Data Bus**
  The bidirectional bus carries data b/w the CPU, memory, and other peripheral devices. It transfers the actual data being processed by the computer.
- **Control Bus.**
  It is a set of signals that co-ordinates and manage the various operations of the computer system. The includes signals for tasks like reading from or writing to memory, initiating input/output operations and controlling the flow of data b/w differ compone.

Cache Mapping and its type

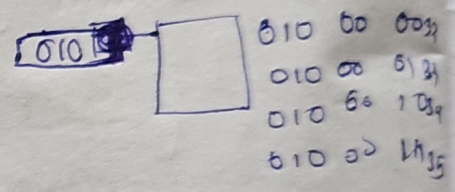


MIPS 10^6

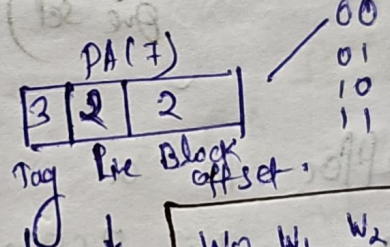
GIPS 10^9.

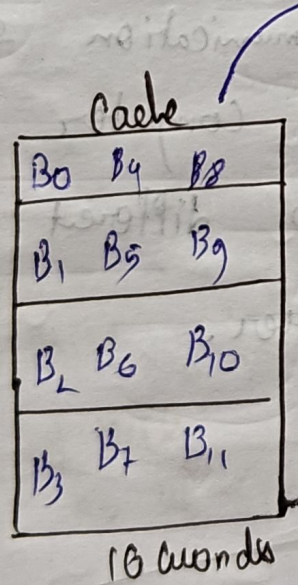
Address

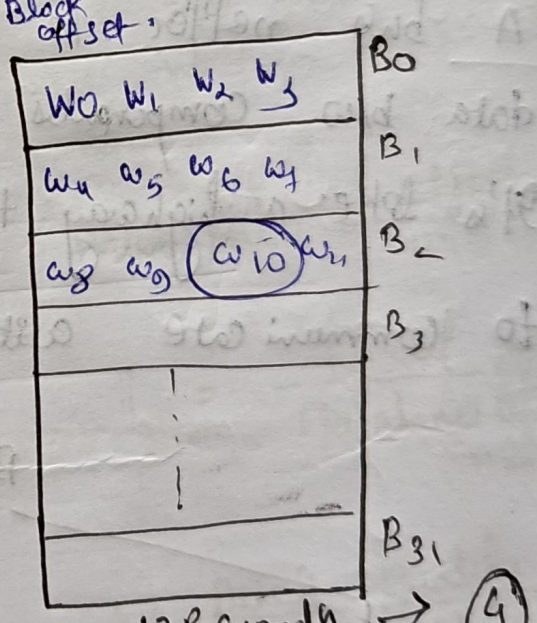
- Direct
- Fully Address
- K-way set address



Direct mapping :-







Cache Hit

Cache miss

Rule

K mod n

↓ Block no ↓ no of line.

B_0

0 mod 4

1 mod 4.

2 mod 4.

line size = Block size

$\frac{16}{4} = 4$

$\frac{128 \text{ words}}{4} \rightarrow 4$

PA(7) = 32 words

5	2

Block no ↳ Block offset (Size)

$0 - 127 = 7$ bit $= 2^7 = 128$

$0 - 3 = 2^2 = 4$

0001 | 10

Q) Consider a direct mapped cache of size 32 KB with block size 32 bytes. The cpu generate 32 bits address. The no of bit require for cache indexing and tag bits respectively.

$$\boxed{n \text{ byte} = \log_2 n}$$

PA

32 bit

17	10	5
Tag	Line	

Block no

block offset (size)

32 byte $= \log_2 32 = 5$

32
15
17

32 KB

viv. Imp

$$\boxed{\text{line} = \text{Block size}}$$

$$\frac{32KB}{32B} = 1K = 2^{10}$$

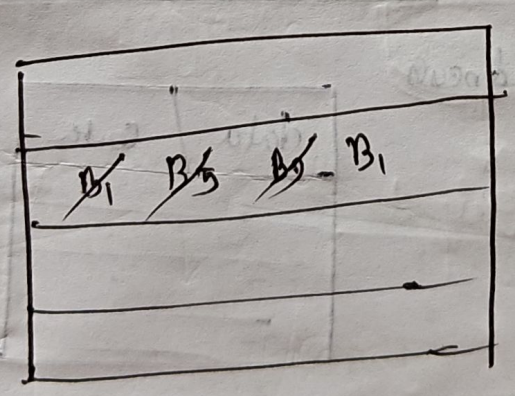
K → 2^{10} byte
M → 2^{20} byte
G → 2^{30} byte
T → 2^{40} byte

Advantage and disadvantage of Direct method:→

01	01	10
tag	line no	Offset.

Conflic miss –

space available but miss mapping.



1, 5, 9, 1

K mod n

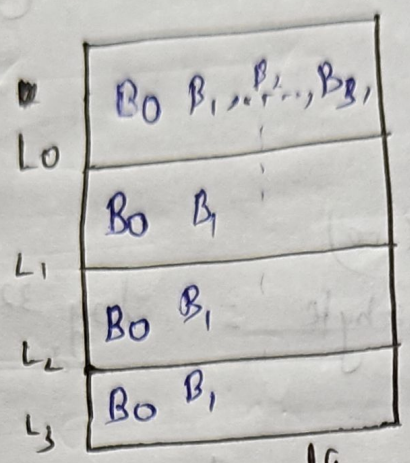
1 mod 4

5 mod 4

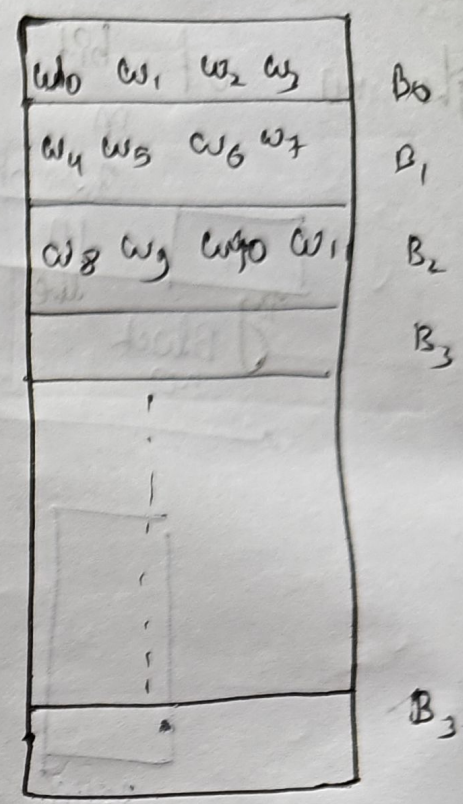
9 mod 4

1 mod 4

Fully associative mapping

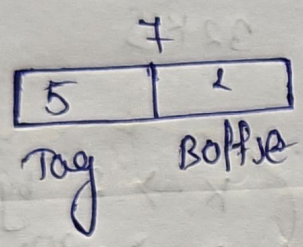


16 words

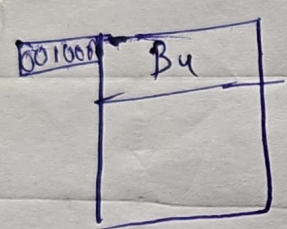


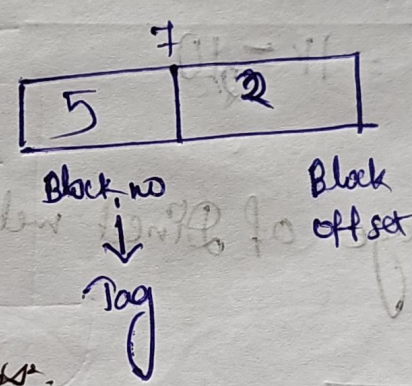
128 words
M/M

0 – 127



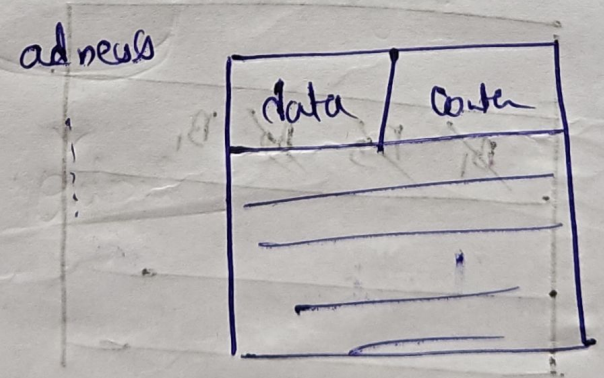
00100 00
00100 01
00100 10
00100 11





disadvantage → (i) Comparison
(ii) Tag value increase.

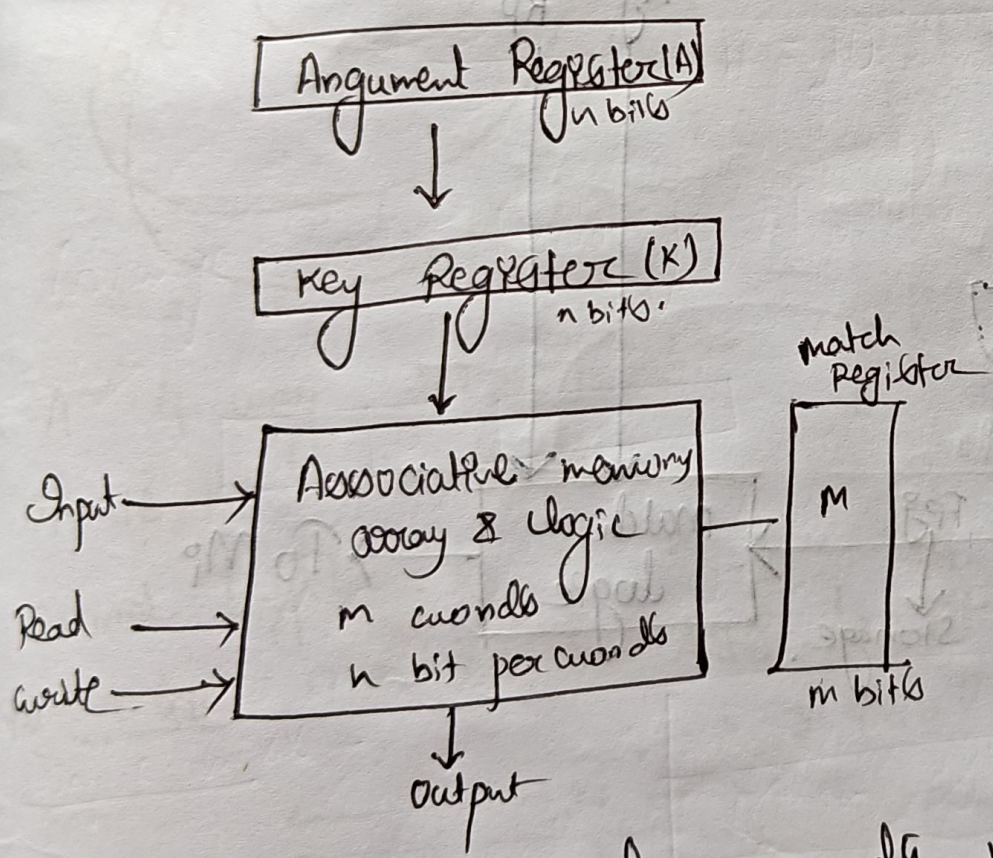
Associative memory



a memory unit accessed by content is called an Associative memory.

→ accessed simultaneously & in parallely.
↓
datacontent (basis of)

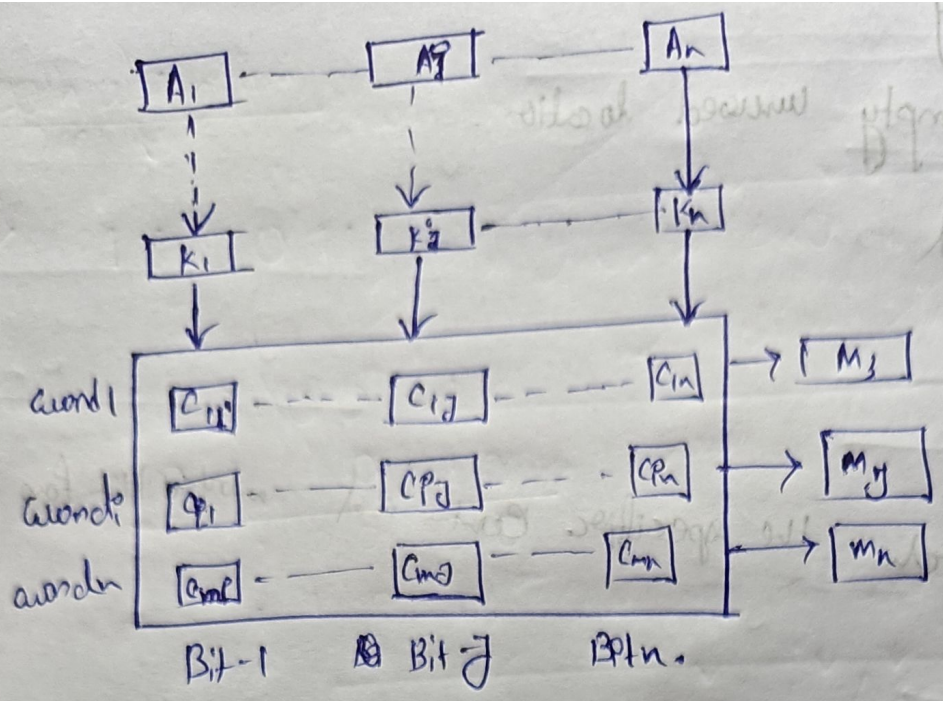
- no address are associative memory
- capable of finding an empty unused location.
- read → the content of word
  Part of word.
- Parallel Searches.
- locate all words which match the specific content and marked them for reading.
- Expensive.



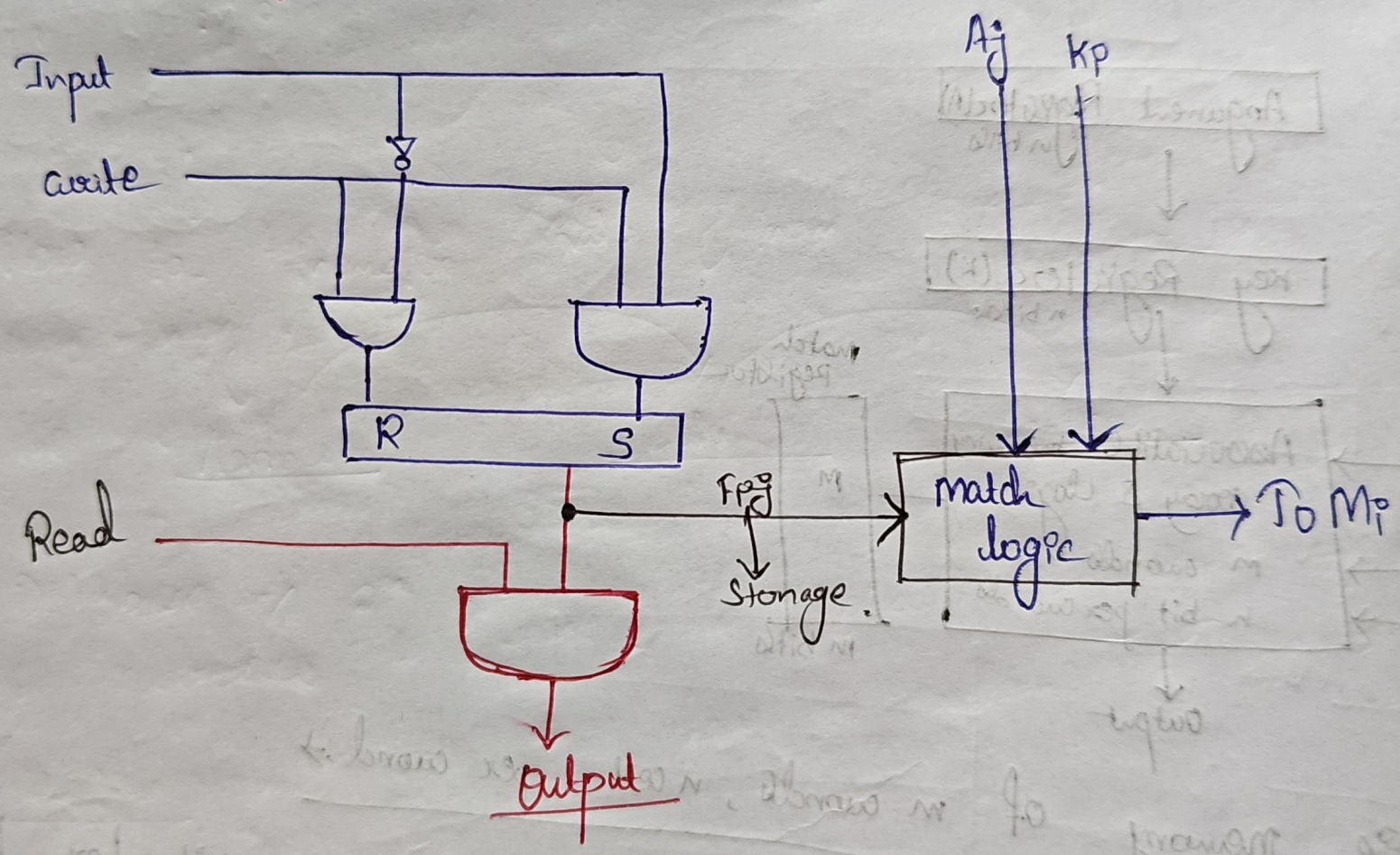
Associative memory of m words, n cell per word :→

The cell in array is represented by C_{ij} a cell for bit 'j' in word 'i'

where i = word numbers
j = bit position in the word.



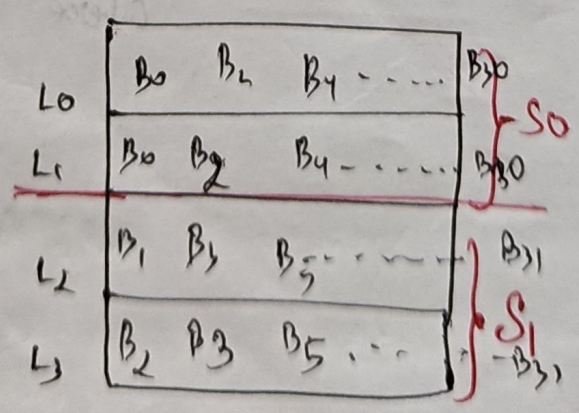
Internal organization of cell C_{pj} →

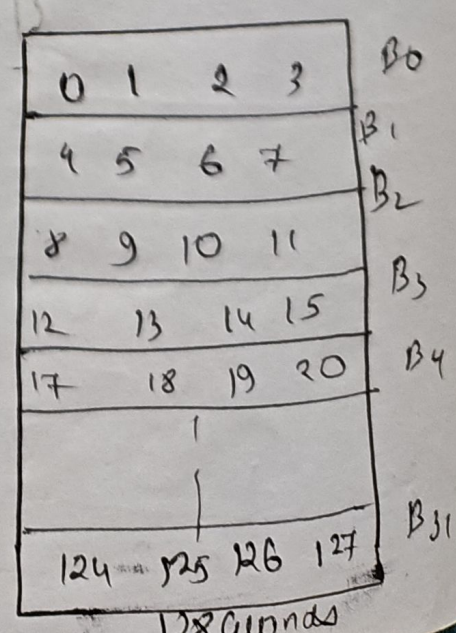


K-way set associative

$\frac{128}{4} = 32$

$\frac{16}{4} = 4$





2 cust set

4 way set

8 way

$= \frac{\text{No of line}}{K} = \frac{4}{2} = 2.$

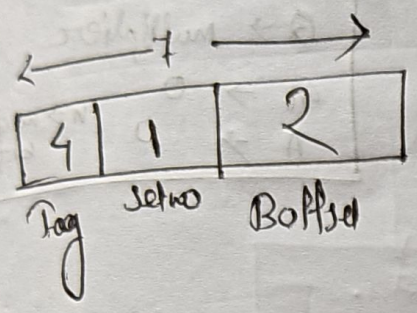
Direct

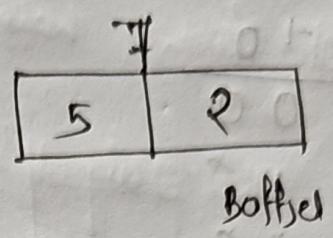
K mod (n) → set.

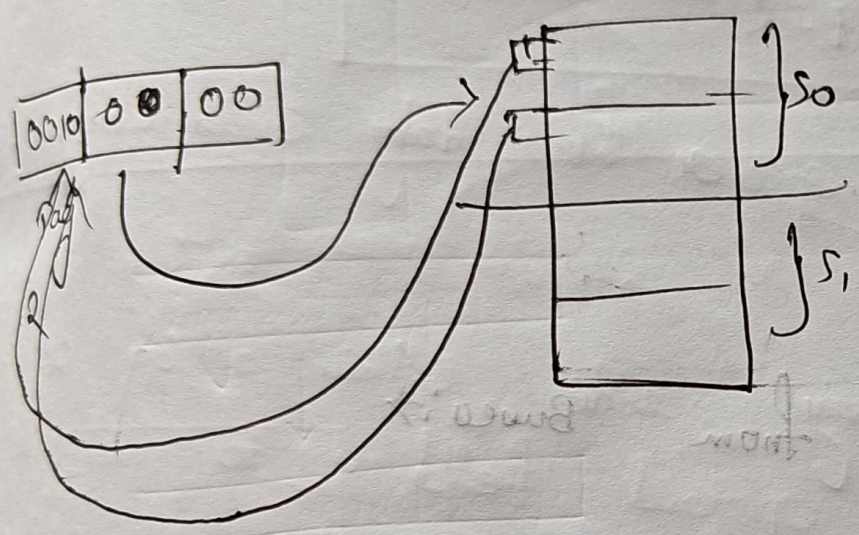
0 mod 2

↓

Associative.



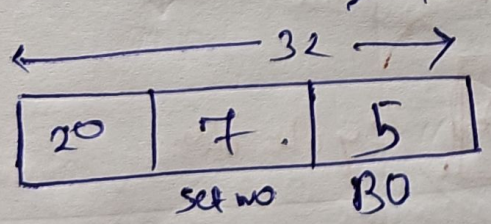


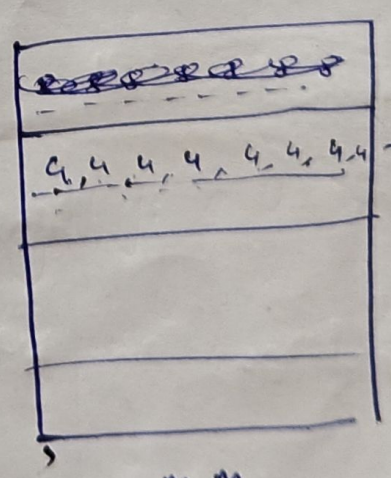


A 4-way set associative cache memory with a capacity of 16 KB is built using block size of 8 words. The word length is 32 bits. The size of physical space is 4 GB. The no of bits for the tag field?

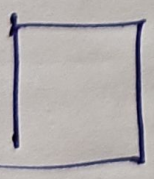
4 GB = 4×2^{30}
$= 2^2 \times 2^{30} = 2^{32}$

1 W = 4 Byte





16 KB



Catch

$\frac{16\,KB}{32\,B}$

$= \frac{2^{10}}{2}$

$= \frac{2^9}{2^2} = 2^7$

4 GB

1 byte = 8 bit

Booth algorithm

5) 110011 (-13) × 110110 (-10)

Start

M → Multiplicand
Q → Multiplier
q_0 → 0
A → 0
n → no of bit

						1	1	0	0	1	1
					1	0	-1	+1	0	-1	0
0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	0	0	1	1	0	1	x
0	0	0	0	0	0	0	0	0	0	x	x
0	0	0	1	1	0	0	1	1	x	x	x
1	1	0	0	1	1	0	1	x	x	x	x
0	0	0	0	0	0	0	0	x	x	x	x
1	1	1	0	0	0	0	0	0	0	1	0

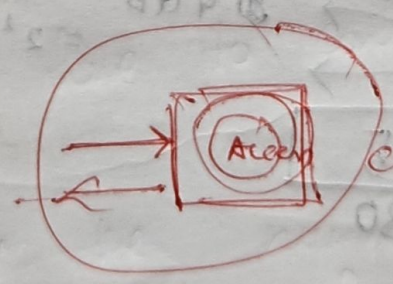
128 64 32 16 8 4 2

* How to transfer data from Buses :→

Address → Data → Control.

* Database → 2^n → n of bit,

Cycle time →



Access time → The time memory text complete one operation (R&W) is the access time.

The minimum time must be lapse b/w two consecutive instruction in CPU. Obviously, cycle time include access time.

Depending on the RAM choose access time fixed

the device Store depend on cycle time.

Computer Instruction set :-

An Instruction manipulate the store data and a sequence of instruction constitute a programe. In general, an Instruction have two components :-

a) ~~output~~ op-code

b) operand / Address field.

The op-code field specier how data could ~~manipulate~~ manipulate.

The data items may reside within a CPU or in main memory. The perpose operand field is to indicate the source ~~destination~~, and or destination of data.

When operation require data from 1 or 2 addresses as store the result on some on address the operand field will have to contain more than one address.

Depending the number of require addresses an instruction can have different format

() → Contain

3 address : ADD R_0, R_1, R_2 $(R_0)+(R_1) \rightarrow (R_2)$

2 address : ADD Acc, R_0 $(Acc) \leftarrow (Acc) + R_0$

↳ accumulator

1 address : ADD R_0

0 - address ADD Stack Oriented architecture.

Since all instructions stored in main memory, instruction format at define. So that instruction size is optimized and also have perform processing capability.

Obviously cpu architecture has considerable influence on specific instruction format.

:-

Hypothetical Instruction set →

LDA	addr	(ACC) ← (adder)
STA	addr	(ACC) ← (adder)
ADD	addr	(ACC) ← (acc) + (addr)
AND	addr	(ACC) ← (ACC) ∧ (addr)
CMA		(ACC) ← $(\overline{ACC})$
INC		(ACC) ← ((ACC) + (1))
JMP	adder	unconditional branch address
HLT		cpu termination.

a unic binary patter must be assigned to each of code. This process none as op-code encoding.

The simplest way to carry out op-code encoding to assigned fixed length of binary pattern to each op-code.

A K-bit binary patter can be repeat 2^K distinct op-code. The concept as move log code encoding.

The op-code of this Hypothetical Instruction set can be decoded using a 3-8 decoder