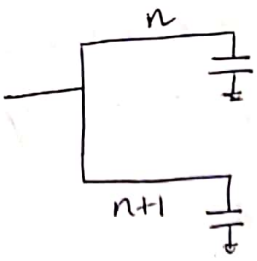


23/09 • Try with fork 2-3 & 3-4 and vary load for last que. of midsem

• ADDERS

- Sum = 1 if A, B, C have odd 1's (full adder)
  ⇒ parity circuit
- Cout = 1 if 2 or more inputs are 1

∴ an adder is counting the number of ones hence sometimes called 'counter' (not the same as conventional counter)

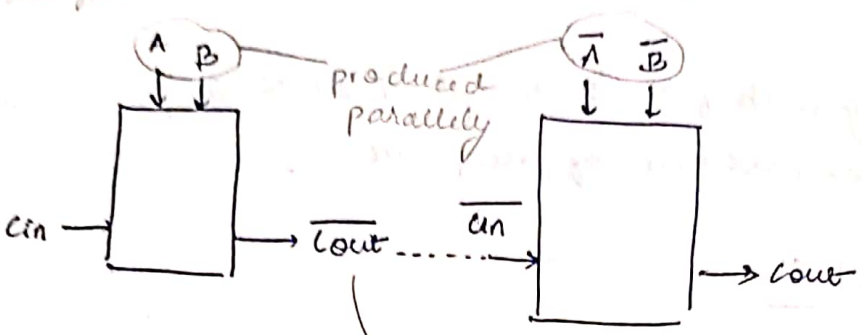
• Faster Adder
  - elemental adder faster
  - how to arrange to minimize delay

• In addition, carry is on the critical path → optimize
(thus sum path need not be made faster)
  ⇒ sum derived from carry (Cin, Cout)

{ inserting inverter is useful when we have heavy load to reduce delay
  ⇒ heavily loaded ⇒ more stage
  lightly —"— ⇒ less —"— } → Pinv becomes large otherwise

}

- **Mirror symmetry** : we can produce $\overline{sum}$ using same circuitry and complemented inputs
  - only for sum & carry attributed to the xor functions used



however inverting in bulk using inverting would lead to additional delay
+ 31 inverters removed

⇒ this helps when capacitive loading not high
(otherwise inverter would've helped)

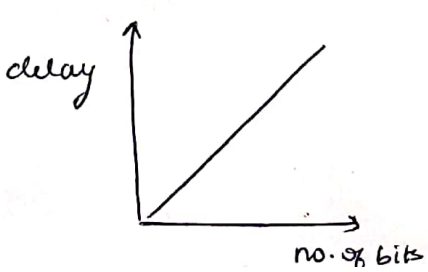
⇒ we can use the same configuration for PMOS as the NMOS in a CMOS structure

[For NOR larger transistor (PMOS) was in series, hence it is inferior to NAND)

→ But here both are same, hence faster

⇒ elemental adder is faster

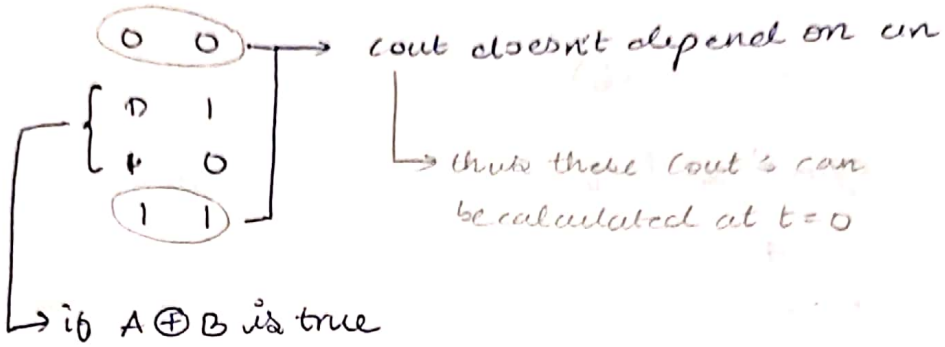
→ $f(\bar{x}_i) = \bar{f}(x_i)$



→ ripple carry adder

- Adders that use grouping have bad results for small number of bits, however good for larger bits.

- when A B



Cout is simply equal to Cin

A	B	
0	0	= Kill condⁿ (K)
0	1	= pass condⁿ (P)
1	0	= pass condⁿ (P)
1	1	= generate condⁿ (G)

- XOR difficult to implement; OR easier,
  1 1 is the only diff b/w them
  ⇒ need to make G dominante over P

- of K, P, G only one is true at a time
  ⇒ compute only 2

⇒ $C_{i+1} = G_i + P_i C_i$

$= G_i + P_i (G_{i-1} + P_{i-1} C_{i-1})$ ⇒ carry look Ahead

24/09 • For 3 inputs

if no. of 1's in ipput = n

then no. of 1's in input complement = 3 - n

complement

Input

Sum : odd → 3-n = even

carry : ≥ 2 → 3-n ≤ 2

- for ab+c (a+b)

  logical effort of

  a = 4 (12/3)

  b = 4 (12/3)

  c = 2 (6/3)

- For K&G we have minimal delay (don't have to wait for carry)

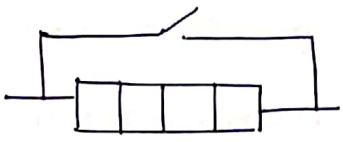
  → use a pass-gate

- Delay = n d_i

  → reducing elemental delay through mirror gates

  → reduce it through grouping

for eg: for 64 bits we break it into groups of 4 bits



→ Worst case:

$C_{i+1} = P_i + P_i C_i$

all P_i's are one (so will need to wait)

→ But if all P_i's are 1 then group output

cout = cin

⇒ so we just AND all P_i's and put a switch, bypassing the 4

⇒ carry-bypass adder

- These adders are the fastest adders (for very large number of bits)

4	3	2	1

→ group at that stage

if $C_i = G_i + P_i C_{i-1}$

$C_4 = G_4 + P_4 C_3$

$= G_4 + P_4 (G_3 + P_3 C_2)$

$= G_4 + P_4 G_3 + P_4 P_3 (G_2 + P_2 C_1)$

$= G_4 + P_4 G_3 + P_4 P_3 G_2 + P_3 P_3 P_2 C_1$

$G_{gp} = G_4 + P_4 G_3 + P_4 P_3 G_2$

$P_{gp} = \prod P_i$

⇒ $C_4 = G_{gp} + P_{gp} C_1$

- Kumar = optimizing recursive relation

- Carry Select adder

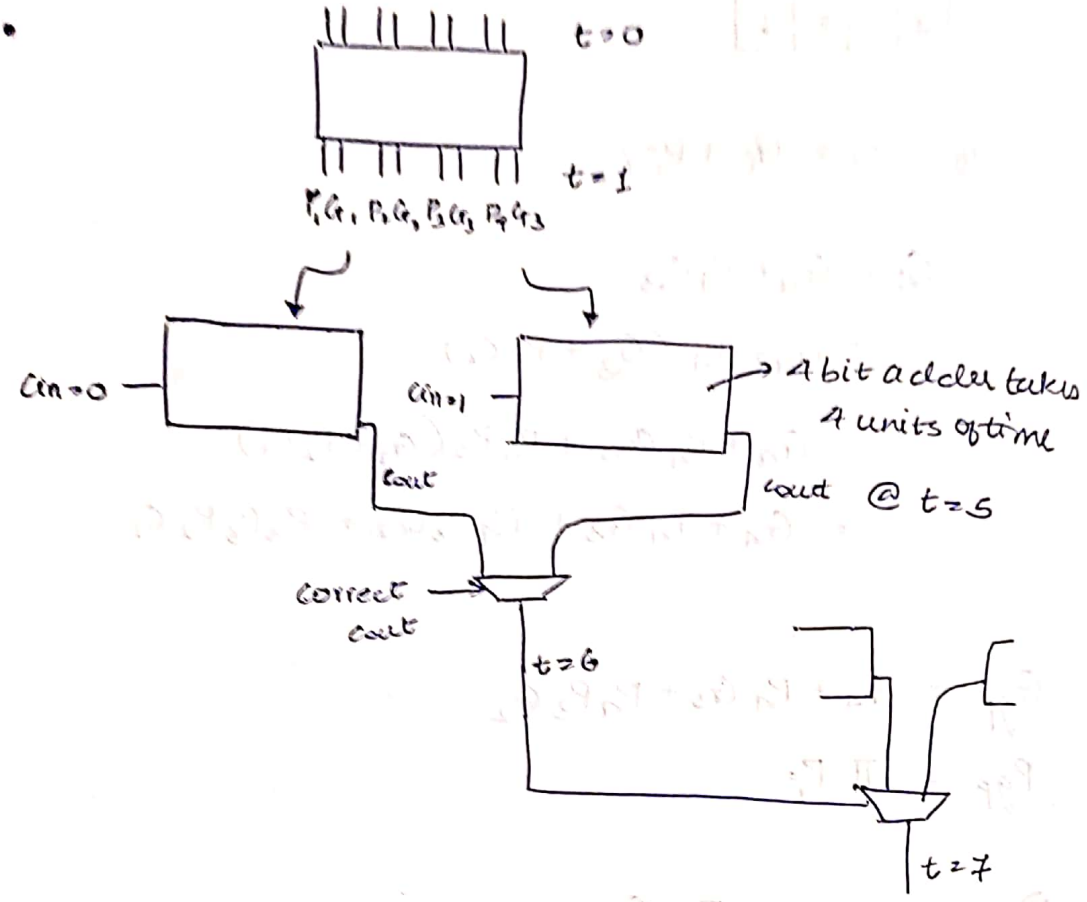
Make 2 adder — assumes carry = 1
— assumed carry = 0
} select one using MUX based on output carry

- power consumpt^n ↑ but time reduces

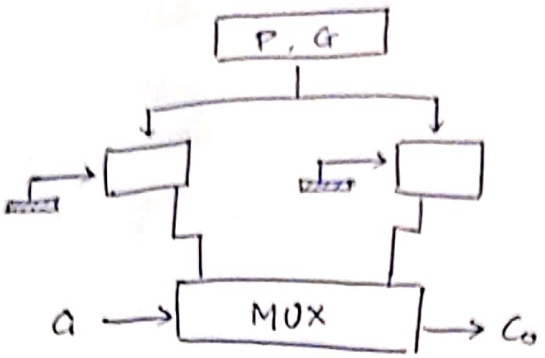
- do it for all groups, not within the group (inside the group we can have any adder)

eg: for 128 bit adder all results are ready in 4 bit adder time

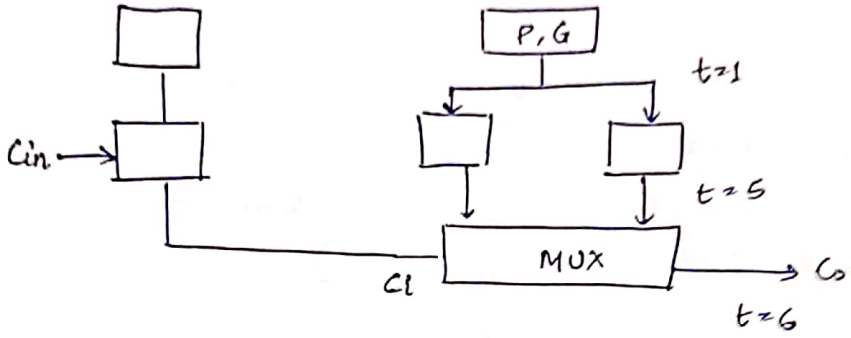
- Models to estimate delay:
  - zero delay
  - unit delay
  - fixed delay



26/09



- total delay
- $\frac{h}{g}$ - max delay
- group delay



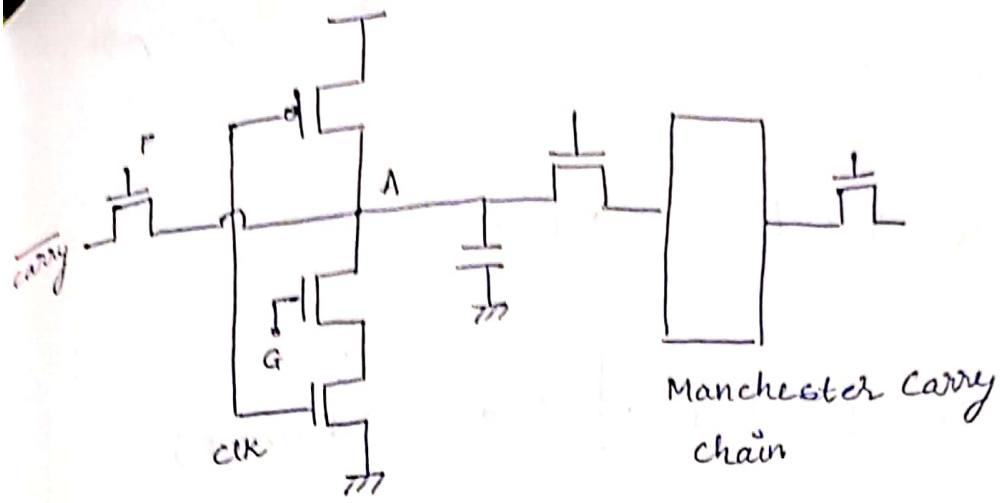
- P, G takes 1 unit of time and 4 bit adder takes 4 units of time, so Ci arrives by t=5
- As complexity ↑, time taken ↑ linearly
- To makes t smaller, then need to ↑ the length of an individual block

→ when tiling is $4+4+5+6+7+\dots$ (carry arrives at this time)

$$\Rightarrow \frac{m(m+1)}{2}$$

time is squareroot of no. of bits.

- If no. of bits ↑ significantly then the time will be too much, so want to use a different architecture then.
  - So we use "logarithmic Adders"



Carry = G + P · Cin (I want G to be dominant)

if G = 1 , capacitance discharges to 0 when clk goes high

at A if

P & $\bar{C}$ = 1 , then A will get 1

P = 1, $\bar{C}$ = 0 , A pulled down

- we can't make bigger chains through this. we take it to limited stagger (3-4) buffer it. P & G calculated using simple OR gate. output is not propagating.

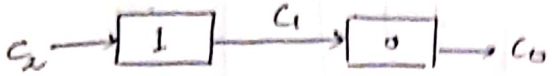
- $C_{out} = C_i + P_i C_{in}$

Let's for one convention on now

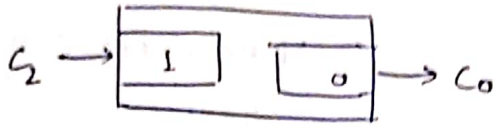
$i = 0, \ldots, n-1$

C_{i+1} → [i] → C_i , $C_{i+1} = G_i + P_i C_i$

The idea is to make and use groups for rippling



⇒



(T circuit)

Let my 2 bit addition be basic adder

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

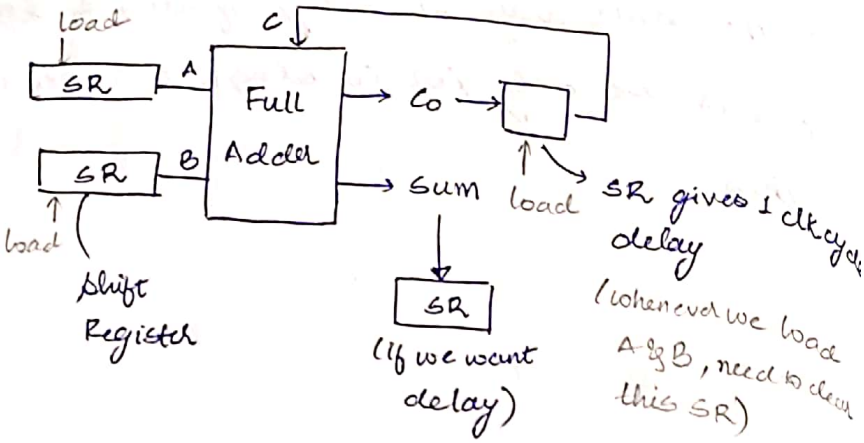
$G_1 + P_1 G_0$ don't depend on carry; $P_1 P_0 C_0$ depends on carry

→ Prefix adder or Logarithmic or Breadkwk

- It can be broken into some G_i & P_i. G can always be computed & P can also be precomputed. Any contiguous block of these adders can be treated together and these combined G's can be calculated.
- G is calculated same way as we had computed carry of single elemental adder.
- Since C is not available, it can't be calculated unlike G which is calculated as carry of single elemental adder.

For T circuit, we calculate C. G is calculated in the same way as carry of single elemental adder. So only for Co stage we need to do this.

30/09 SERIAL ADDER



Ripple carry adder (time taken) = n (addition time)

Serial adder = n (clk ~~cycle~~ period) (if clocks are matched

⇒ as good as ripple carry adder)

{ ASSGN

FO4 delay

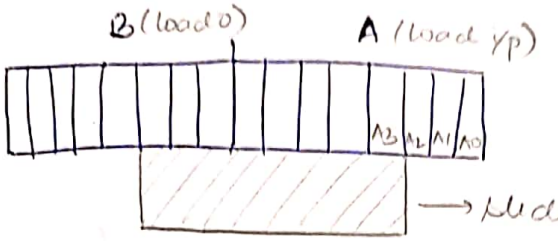
Fanout 4 → do for 64 bit Brent Kung adder}

SHIFTER

- they do shift and rotate
- shift × or ÷ by 2
- shift right — signed (replicate MSB)
  — unsigned (insert 0)
- if we want to do 27 shifts, simple shift register takes 27 clks (while i/p remains same) ⇒ not desired
- so we have parallel shifter (used is Barrel shifter)

→ ideally produces o/p in 1 clk cycle

- essentially we need to compute the new position for a bit
- if amount of shift is const (say 5) use just wire the i/p to that bit position (⇒ no extra hardware needed)



⇒ NOTION not implementation

⇒ logical right shift

→ sliding window

- if B loaded with MSB of A then it becomes Arithmetic right shift (signed RS)
- if B loaded with replica of A then it becomes rotate

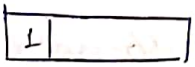
- if amount of shift variable, each output connected to a MUX. Then shifting done is (N×1 MUX) 1 clk cycle

  New position = current position + shift

  → But for large no. of bits, the circuit would be very large ⇒ clock increases.

  → Need to reduce complexity from N to log N

- $-n = 2^n - n$: in 2's complement representation

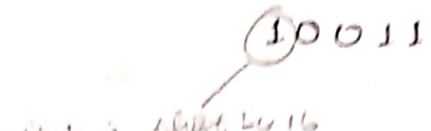
  : sign-magnitude representation

→ MUX if 8 bit processor

each bit has 8×1 MUX ⇒ 8×8 complexity

for 32 bit : 32×32 complexity

- If we want to shift by 19 position

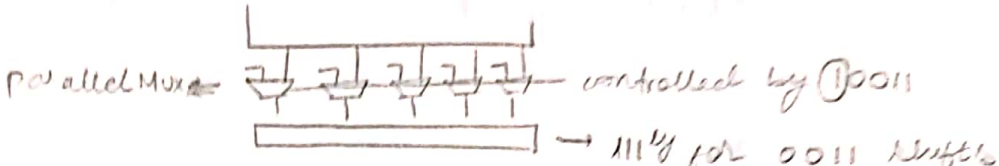


i&1 : shift by 16
i&0 : no shift

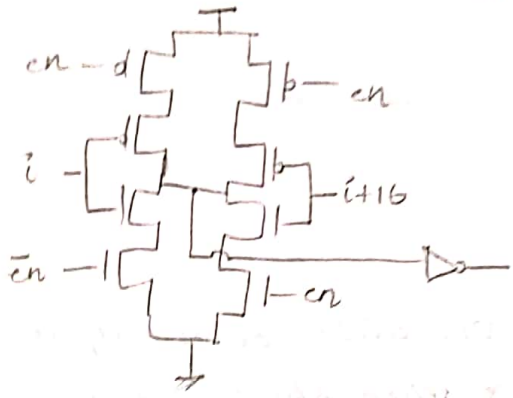
} help: reduces complexity to log n

⇒ dest. = i or i+16

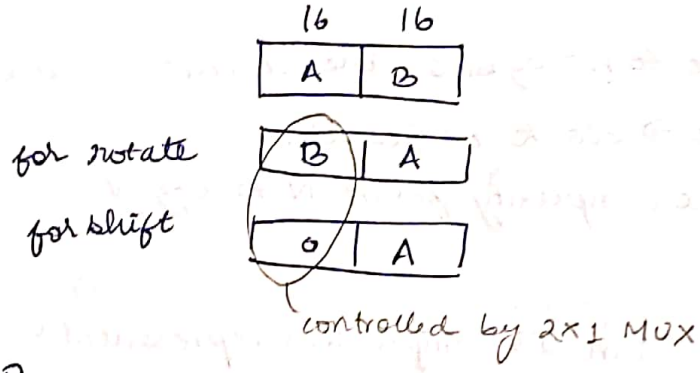
⇒ 2 way MUX



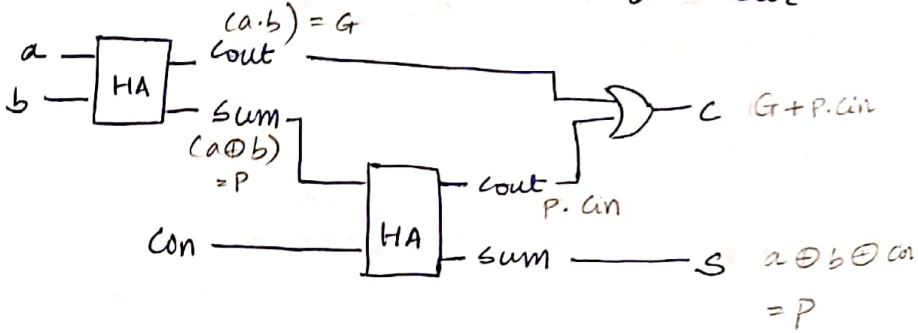
MUX:



- For a 32 bit ckt, shift or rotate by 16 requires only 1 2x1 mux



01/10 Making full adder from half adder



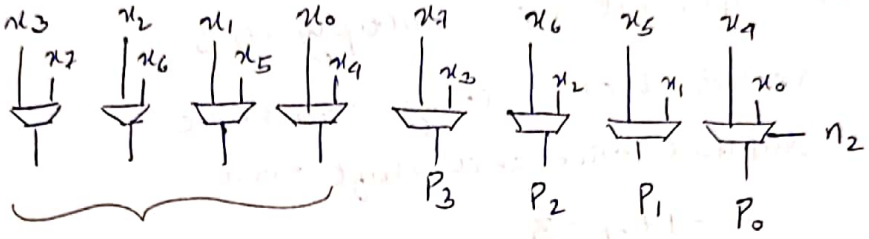
• Multiplication:

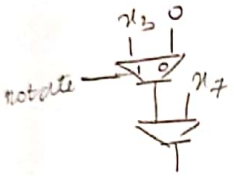
Conventional :

$$\cdots = \sum 2^i a_i$$
$$\cdots = \sum 2^j b_j$$
$$\text{Product} = \sum_{i+j=k}^{n} \sum a_i b_j 2^{i+j}$$

$k = 0, \ldots n$

• Shifting





⇒ to use same ckt for rotate & shift

- combining left and right shift

→ keeping the same ckt, reverse the bit sequence being fed.

i.e. for left shifting, reverse the bits

• Shift and Add Multiplier

Partial product : things to be added

- can add bit-by-bit or row-by-row

- in array multiplier : both sum & carry (rippling down) are in critical path (rippling left)

$2(n-1)$ = amount of rippling required

→ not good for large no. of bits

- To optimise we can reduce
  - number of stages of operation
  - time taken by operation

- $A = a_{n-1} \dots\dots\dots a_0$

⇒ Combine 2 bits of B and multiply to get the rows we add finally

for $b_1 b_0 = 11$ i.e. multiply by 3

rather than adding 2 rows we subtract 1 once and shift twice

⇒ $-1+4 = 3$

Illy, we can make $2 = -2+4$

03/10

- Multiplying 2 numbers:
  - choose B such that it has fewer bits ⇒ fewer no. of additions

→ Group B into groups of 2 bits

Possibilities:

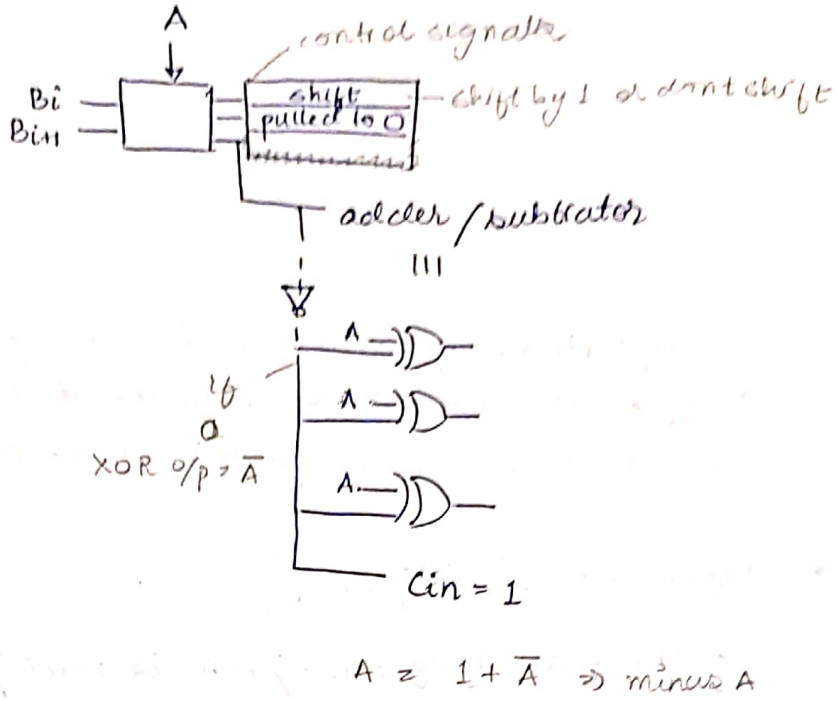
00	0	} easy to do
01	A	
10	2A	
11	3A = $-A+4A$	done by the successive 2 bits of B

3A: upon multiplying with A

(Currently we are generating partial products
- we want them to be available parallelly)

2A can also be converted to $-2A+4A$

Partial Product Generator



$A = 1 + \bar{A} \Rightarrow$ minus A

For -2, shift by 1 then negative

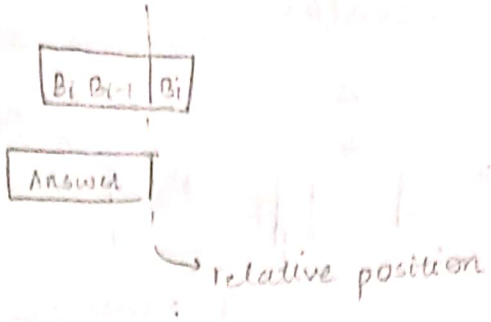
- considering some in-between bits

00 [00 00]

↳ we consider 3 bits the LSB of which tells us if we need to do +4A

- For the LSBs, add a zero at the end
- If groups get over, add zeros at the beginning

		Partial Product	
00	0	0	
	1	A	
01	0	A	
	1	2A	
10	0	-2A	(we want 2A, but MSB adds +4 so make it -2A)
	1	-A	(we want 3A, but MSB adds 4A so make it -A)
11	0	-A	
	1	0	

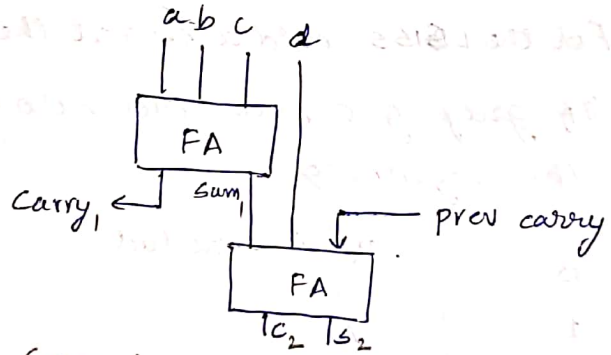


- all negative quantities become full width and in final addition we require ~~it~~ twice as much size as normal multiplier adder

- Different Multipliers
  - one feature of multiplier is multiple of adders
  - carry (due to rippling) makes adders slow

Carry Save Adders (useful for large no. of bits)

- gives 2 numbers which must be added
- saving the carry for the end, so we don't ripple carry for the in-between steps



Carry has twice the weight

- one carry saved, other sent horizontally
- c_1 & s_1 available in const. time i.e. at $t=1$
- prev. carry aval. at $t=1$, c_2, s_2 avail at $t=2$

However, these are avail. at const. time doesn't depend on their position

ASSGN

16x16 = multiplier ⇒ 32-bit adder

↳ thus needs to be faster

- carry save uses rectangular array whilst traditional adders are trapezoidal
- carry save not good for traditional addition

7/10

- Carry Save adder → addition doesn't depend on Cin

→ store all carry in 1 register and store all sum in 1 register

→ shift carry register by 1 place value

→ HA take 2/p and produces 2o/p however reduces to 1 wire at its place value

FA : 3 ⟶ 1 (at its place value)

- Wallace multiplier

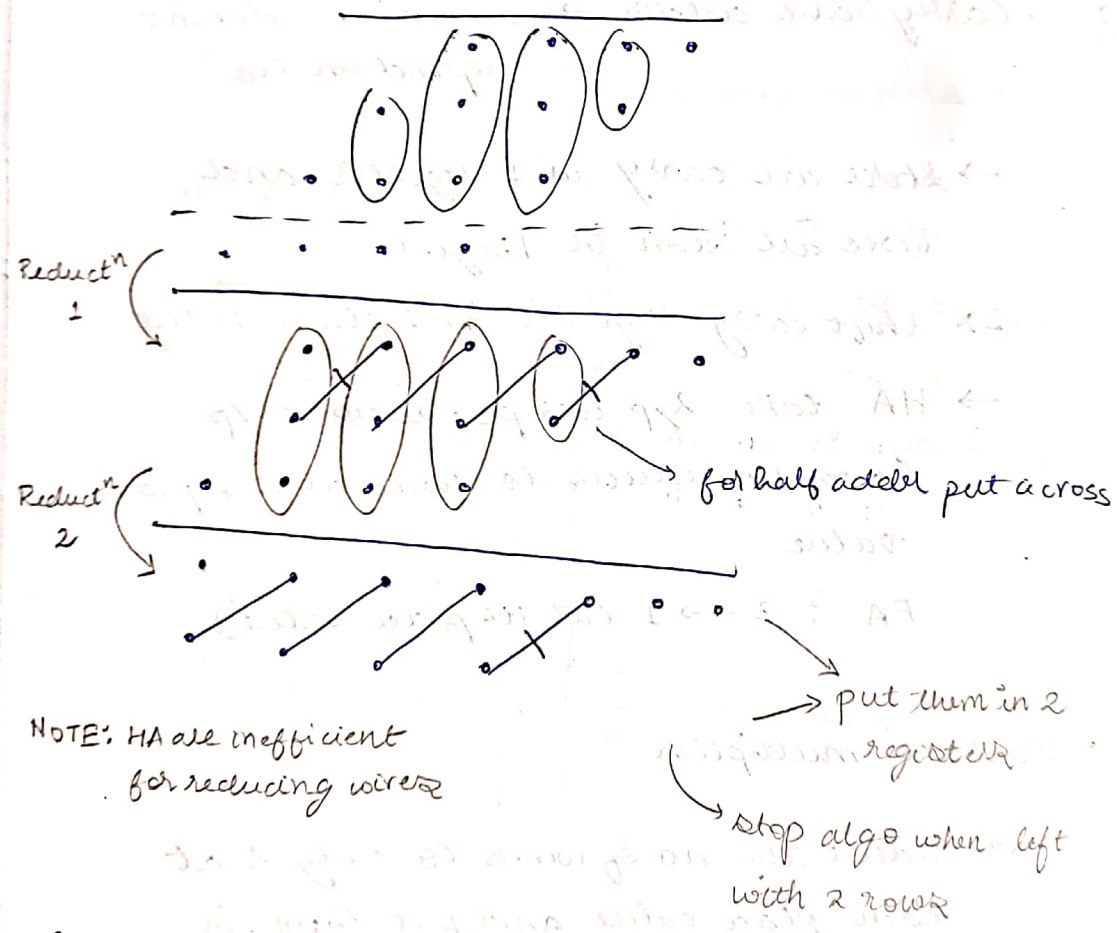
⟶ reduce the no. of wires to only 2 at each place value and put them in 2 registers

↳ add these 2 registers using fast adder

- HA uses (2,2) counter, FA is a (3,2) counter carry save is a (4,2) counter
- Take the wires and make groups of 3

  if it has
  - 3 → send to FA and send 1 to next bundle
  - 1 → do nothing; pass on to next stage
  - 2 → choice of using HA or passing it on

→ Draw dotted line as along as one can make groups of 3



NOTE: HA are inefficient for reducing wires

- 8 x 8 multiplier Wallace

→ we don't put HA in rows of 2 coz they don't reduce the no. of wires, so using only HA we end up with too many bits at MSB

{Capacity = floor{1.5 x successive no. of rows}}

(integer part)

→ fast adder becomes narrower (?)

• problem that we might get a 17th bit

→ comes up due to misinterpretation of wallace algorithm

→ happens due to usage of more HA

(propose reduction of 2 bits so that we can use a FA later on)

→ if we are not exceeding the capacity of the next stage by passing the 2 wires then pass them through

(need to back calculate the capacity for each stage)

10/10 → Get correct logic [structurally correct] [behaviorally synthesizable]

→ tick tock clock (present in library)

(backannotation)

(delay needs to be added in library)

→ all gates should be inverting

→ Critical Path: when carry and sum both made

→ For AND = NOT + NAND : make your own cell

• 8 x 8 multiplier

b_{15} b_{14} b_{13} b_{12} b_{11} b_{10} b_9 b_8 b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0

1 2 3 4 5 6 7 8 7 6 5 4 3 2 1 ← no of wires

→ when we get 2 wires:

i) if all bits are to the left are 1 wire then pass it through

ii) if on passing it through we exceed the capacity of the next layer then we use HA

(goon till max^m no of wires are carried)

Capacity: 2, 3, 4, 6, 9

→ our capacity is 9 but we have 8. So we could've added an extra layer for accumulate after adding without increasing the time a lot

	1	2	3	4	5	6	7	8	7	6	5	1	3	2	1
b_{15}	b_{14}	b_{13}	b_{12}	b_{11}	b_{10}	b_9	b_8	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
	1P	2P	1F	1F	1F	2F	2F	2F	2F	2F	1F	1F	1F	1H	1P
				1P	2P		1P	2P	1P		2P	1P			
	1	3	2	3	5	4	5	6	5	3	4	3	2	1	1
	1P	1F	2P	1F	1F	1F	1F	2F	1F	1F	1F	1F	1H	1P	1P
					2P	1P	1H		2P		1P				

→ 1+2+2 = 5 > 4 so use HA

2	1	3	2	4	4	4	3	4	2	3	2	1	1	1
2P	1P	1F	2P	1F	1F	1F	1F	1F	2P	1F	1H			
				1P	1P	1P		1P						

2	2	1	3	3	3	3	2	2	3	2	1	1	1	1
2P	2P	1P	1F	1F	1F	1F	1H	1H	1F	1H				

→ capacity now is 2

2 | 1 1 1 1

need to add only these wires now

⇒ adder now has to add 11 bits

DADA MULTIPLIER

- to reduce delay we minimize the no. of electronics used, ~~not~~ past through and no delay
- goal now is not wire reduct^n rather reduce the stage delay
- FA has more delay than HA. so we aren't reluctant to use HA now

- Wire reduct^n capability of

  FA = 2 (3 → 1 at same place)

  HA = 1

- At every stage

  even part managed by FA

  odd part —u— by HA

  (unnecessary bits by HA not generated)

  if within the capacity of next layer pass through don't use FA)

  → capacity becomes capacity of current layer - no. of carries from previous stage

b_{15}	b_{14}	b_{13}	b_{12}	b_{11}	b_{10}	b_9	b_8	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
1	2	3	4	5	6	7	8	7	6	5	4	3	2	1	
1P	2P	3P	4P	6P	1F	1F	1F	1H	6P	5P	4P	3P	2P	1P	
					3P	1H	1H	5P							
						3P	2P								

Slot available = (6-1) = 5
⇒ 3 need to be reduced
⇒ 1H + 1F

Slot available = 6
⇒ 1 needs to be reduced
⇒ 1H

1	2	3	4	6	6	6	6	6	6	5	4	3	2	1
1P	1P	1P	1F	2F	2F	2F	2F	2F	1F	1H	4P	3P	2P	1P
									1H					

TUTORIAL

10.107.90.71/72

EE671-16

P/W: EE671 → change P/W: PSSWD

→ login to cluster

→ 4 levels of metal ⇒ SCL process

15/10 Dada:

- use minimum no. of adders
- of smallest size to reduce the no. of wires to the capacity

Disadv: since we don't pass through all the time, we don't get a narrow adder

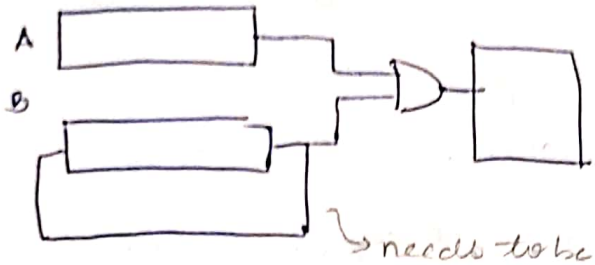
→ the wires arriving late is passed through to reduce worst case delay

→ since multiplication is shift and add we add one more wire to the set of that we add also simultaneously

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1

11 GP 5P 4P 3P 2P 1P

- Find out what should be done incase of
  - signed
  - division (hardware similar to sqroot)
- Next we try to get a less complex multiplier

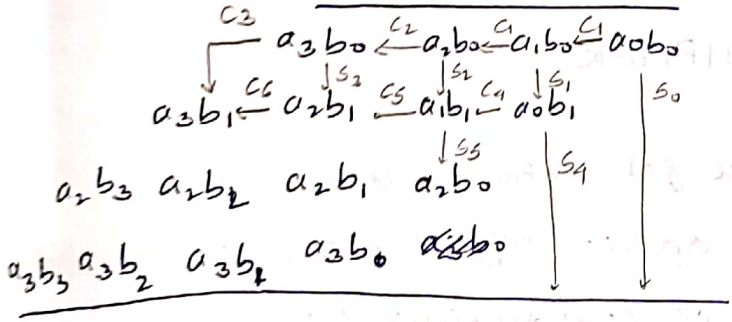


→ needs to be recirculating and faster since they keep multiplying with A

17/10

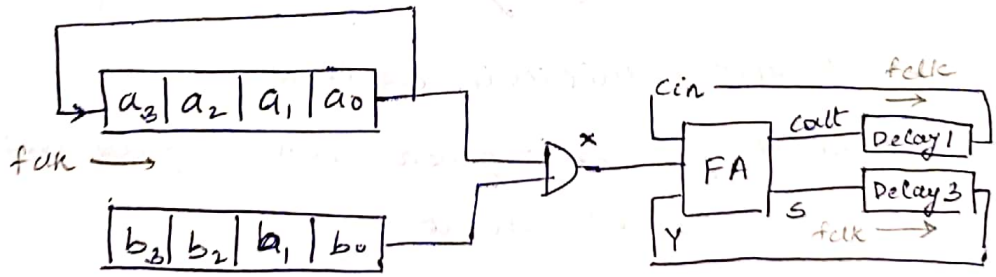
a_3 a_2 a_1 a_0 → repeated multiple times and is 4 times faster compared to B

b_3 b_2 b_1 b_0



BIT-SERIAL MULTIPLIER

→



$\frac{fclk}{4}$ →

- Partial products to be added after (M-1) clocks apart

  M = no. of bits

- ~~Cin~~ the cout have to appear in the next clock as Cin

- We need to add a reset to the carry store since the 1st row doesn't need a carry.
- $a_0 b_0$ taken out 3 clks later
- However $a_2 b_1$ is made by adding carry c_3 and carry c_6, hence we need to add a MUX whether to choose sum or carry to be sent again to the FA. The MUX takes carry at multiples of M. Also during these times, take out the sum
- The result arrives M×N clock cycles later

ROW-SERIAL MULTIPLIER

→ The adder used for $a_0 b_0$ is used only once (if adders added only 1 place value terms). Hence, we keep shifting the place value terms a given adder adds

→ Result available at M+N

→ In the process, a given adder gets the same a_i throughout
Place value keeps increasing due to b_i

- Double precision multiplier (using control flow)

→ $\begin{matrix} P & Q \\ R & S \end{matrix} \equiv \underline{2^n P + Q \\ 2^n R + S}$

→ $-P = 2^n - P \quad \Rightarrow \begin{matrix} -P \\ \times Q \end{matrix} \equiv \underline{\begin{matrix} 2^n - P \\ Q \end{matrix} \\ 2^n \cdot Q - PQ}$