Adders

Dinesh Sharma

EE Department
IIT Bombay, Mumbai

September 26, 2019

Half Adder

The truth table for addition of two bits is:

A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

$$\text{sum} = A \cdot \overline{B} + B \cdot \overline{A}$$
$$\text{carry} = A \cdot B$$

- What do we do with the carry?
- Obviously, it must be added to more significant bits.
- So we need an adder with *three* inputs.

Full Adder

Truth Table for the addition of three bits is:

A	B	Cin	Sum	Cout
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1

Which leads to the following Karnaugh maps:

Cin \ AB	00	01	11	10
0	0	1	0	1
1	1	0	1	0

SUM

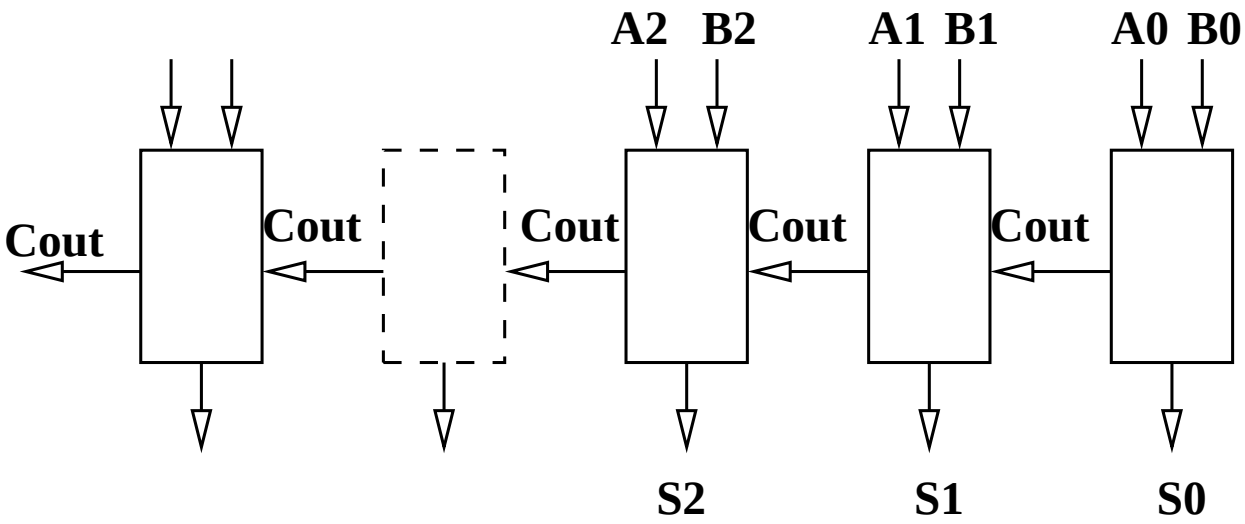
Cin \ AB	00	01	11	10
0	0	0	1	0
1	0	1	1	1

CARRY

$$\text{sum} = \overline{A} \cdot \overline{B} \cdot C_{in} + \overline{A} \cdot B \cdot \overline{C_{in}} + A \cdot B \cdot C_{in} + A \cdot \overline{B} \cdot \overline{C_{in}}$$

$$C_{out} = A \cdot B + B \cdot C_{in} + C_{in} \cdot A = A \cdot B + C_{in} \cdot (A + B)$$

Ripple Carry adder



- Carry out of one bit becomes Carry in of the next.
- This architecture is therefore called ripple carry adder.
- The critical delay path of the adder is the carry rippling from one bit to the next.

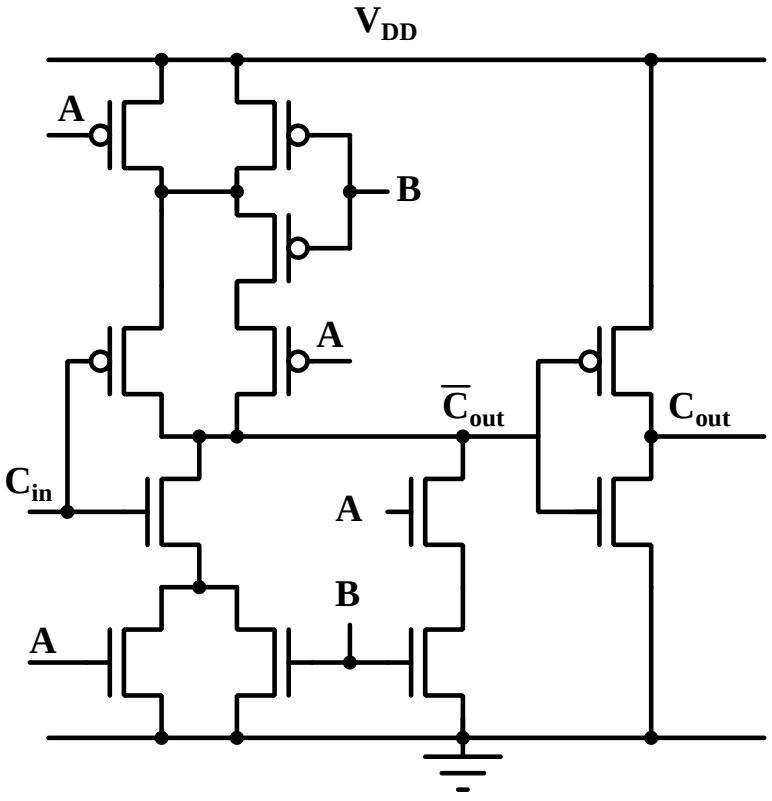
Sum derived from carry

- Because carry is on the critical path, Carry-out must be generated as quickly as possible.
- We need not optimize the delay of generating sum.
- We can in fact generate sum from Carry out.

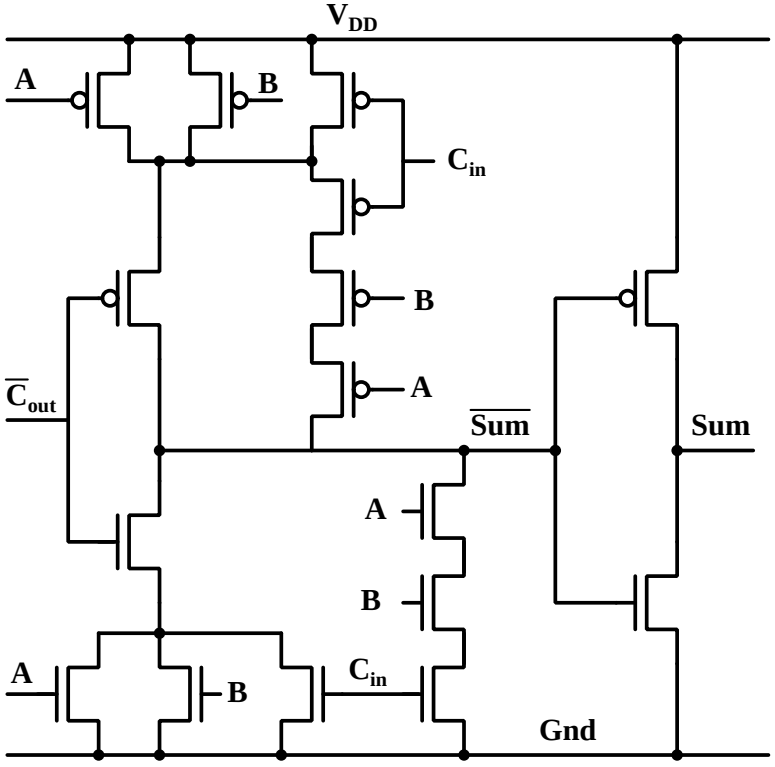
$$\begin{aligned}
\overline{C_{out}} &= \overline{A \cdot B + C_{in} \cdot (A + B)} \\
&= (\overline{A} + \overline{B}) \cdot (\overline{C_{in}} + \overline{A} \cdot \overline{B}) \\
&= \overline{A} \cdot \overline{C_{in}} + \overline{B} \cdot \overline{C_{in}} + \overline{A} \cdot \overline{B} \\
\overline{C_{out}} \cdot (A + B + C_{in}) &= A \cdot \overline{B} \cdot \overline{C_{in}} + \overline{A} \cdot B \cdot \overline{C_{in}} + \overline{A} \cdot \overline{B} \cdot C_{in}
\end{aligned}$$

$$\begin{aligned}
\text{sum} &= A \cdot \overline{B} \cdot \overline{C_{in}} + \overline{A} \cdot B \cdot \overline{C_{in}} + \overline{A} \cdot \overline{B} \cdot C_{in} + A \cdot B \cdot C_{in} \\
&= \overline{C_{out}} \cdot (A + B + C_{in}) + A \cdot B \cdot C_{in}
\end{aligned}$$

CMOS Implementation



$C_{out} = A.B + C_{in} . (A+B)$



$Sum = \overline{C}_{out} . (A + B + C_{in}) + A . B . C_{in}$

Complementation Property

Both Sum and Carry show an interesting symmetry:

$$
\begin{aligned}
\text{sum} &= \overline{A} \cdot \overline{B} \cdot C_{in} + \overline{A} \cdot B \cdot \overline{C_{in}} + A \cdot \overline{B} \cdot \overline{C_{in}} + A \cdot B \cdot C_{in} \\
\overline{\text{sum}} &= (A + B + \overline{C_{in}}) \cdot (A + \overline{B} + C_{in}) \cdot (\overline{A} + B + C_{in}) \cdot (\overline{A} + \overline{B} + \overline{C_{in}}) \\
&= (A + A \cdot \overline{B} + A \cdot C_{in} + A \cdot B + B \cdot C_{in} + \overline{C_{in}} \cdot A + \overline{C_{in}} \cdot \overline{B}) \cdot \\
&\quad (\overline{A} + \overline{A} \cdot \overline{B} + \overline{A} \cdot \overline{C_{in}} + \overline{A} \cdot B + B \cdot \overline{C_{in}} + C_{in} \cdot \overline{A} + C_{in} \cdot \overline{B}) \\
&= (A + B \cdot C_{in} + \overline{B} \cdot \overline{C_{in}}) \cdot (\overline{A} + B \cdot \overline{C_{in}} + \overline{B} \cdot C_{in}) \\
&= A \cdot B \cdot \overline{C_{in}} + A \cdot \overline{B} \cdot C_{in} + \overline{A} \cdot B \cdot C_{in} + \overline{A} \cdot \overline{B} \cdot \overline{C_{in}}
\end{aligned}
$$

This shows that the **same hardware** that produces sum from A, B and C_{in}, will produce $\overline{\text{sum}}$ if the inputs are changed to $\overline{A}$, $\overline{B}$ and $\overline{C_{in}}$

Complementation Property

$$
\begin{aligned}
C_{out} &= A \cdot B + C_{in} \cdot (A + B) \\
\overline{C_{out}} &= \overline{A \cdot B + C_{in} \cdot (A + B)} \\
&= (\overline{A} + \overline{B}) \cdot (\overline{C_{in}} + \overline{A} \cdot \overline{B}) \\
&= \overline{A} \cdot \overline{C_{in}} + \overline{B} \cdot \overline{C_{in}} + \overline{A} \cdot \overline{B}
\end{aligned}
$$

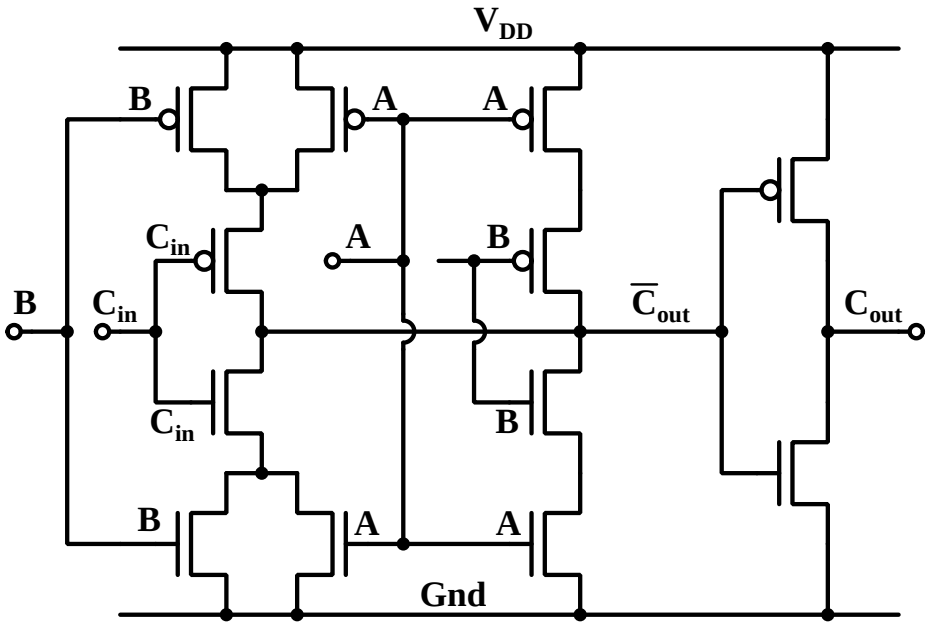
Thus the carry function also has the same property:
The same hardware which produces C_{out} from A, B and C_{in}, will produce $\overline{C_{out}}$ from $\overline{A}$, $\overline{B}$ and $\overline{C_{in}}$.

Making use of the symmetry property

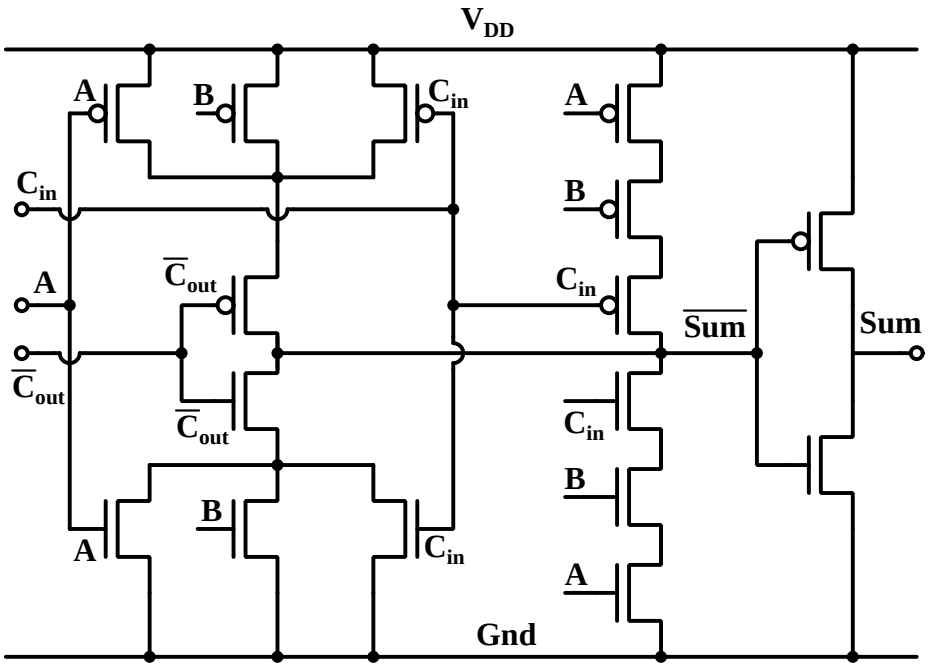
- In CMOS implementation, we interchange series and parallel configurations for the n and p channel transistors.
- This is to ensure that the pull up and pull down circuits are complementary.
- However, for sum and carry functions, we see that these functions are their own complements.
- Therefore, for implementing sum and carry, we can use the **same** configuration for n and p channel transistors.
- We use this to reduce the number of series connected transistors in pull up/pull down networks.

Mirror gates for Adders

- By making use of symmetry property of sum and carry, it is possible to simplify the implementations.



$C_{out} = A.B + C_{in} . (A+B)$



These are called mirror gates because the n and p transistors have the **same** series parallel combination.
This is highly unusual.

Speeding up the Ripple Carry Adder

- The worst case delay of the ripple carry adder is linear in number of bits to be added.
- To reduce the delay per stage, we can eliminate the inverter from the carry output.
- All even bit adders accept a, b and C_{in} as inputs. The mirror gate without inverter gives $\overline{C_{out}}$ as the output.
- All odd bit adders accept $\overline{A}$, $\overline{B}$ and $\overline{C_{in}}$ as inputs and thus produce C_{out} as output.
- Outputs of all bits are now compatible with inputs of the next stage.

Speeding up the Ripple Carry Adder

- Extra inverters are required to produce $\overline{A}, \overline{B}$ and at the outputs to produce the proper result. However, these are not on the critical path, and do not add to the worst case delay.
- Extreme care needs to be taken in layout to ensure that the loading on the tree gate producing carry output is as small as possible.

Terms Independent of Carry

- Carry propagation is the critical path for a multi-bit adder.
- To speed up the adder, we would like an architecture where logic terms are classified as those dependent on carry and those which do not depend on carry.
- To speed up the adder, we would like to pre-compute all terms which do not depend on carry.
- Now when the carry arrives, we quickly compute the output carry and pass it on to the next stage.

Carry Independent Terms

We would like to analyze what information can be pre-computed from A_i and B_i, which will help us in generating C_{out} quickly from C_{in}.

- When $A_i = 0$ and $B_i = 0$, C_{out} is 0, independent of C_{in}. We define this condition as 'Kill'. $K = \overline{A} \cdot \overline{B}$
- Similarly, when $A_i = 1$ and $B_i = 1$, C_{out} is 1, independent of C_{in}. We define this condition as 'Generate': $G = A.B$.
- Only when $A_i = 0$ and $B_i = 1$ or when $A_i = 1$ and $B_i = 0$, we need to wait for C_{in} to compute C_{out}.
  In both these cases, $C_{out} = C_{in}$.
- We call this condition as 'Propagate', and define $P = A.\overline{B} + \overline{A}.B$.

Using Carry Independent Terms

We define $K = \overline{A} \cdot \overline{B}$, $G = A.B$ and $P = A \oplus B$
Exactly one of K, G or P is true at any time.

When K = 1, C_{out} is 0, independent of C_{in}.
When G = 1, C_{out} is 1, independent of C_{in}.
When P = 1, $C_{out} = C_{in}$.

P needs to be computed using an xor gate, which can be slow. However, the only difference between xor and or logic is when both inputs are 1, i.e. G = 1.

If we can ensure that G forces C_{out} to 1 irrespective of P, we can use the simpler 'or' logic to compute P.

Carry Look Ahead

C_{in} for bit i+1 is the C_{out} of bit i.
So we can write $C_{i+1} = G_i + P_i.C_i$
Notice that the Kill signal is not required.

If $G_i = 0$, $C_{i+1} = A \oplus B = A + B$ when $G = A.B = 0$
If $G_i = 1$, $C_{i+1} = 1$, and the value of P_i does not matter anyway.
So we can use P = $A + B$ instead of $P = A \oplus B$.

Now, we have the sequence:

$$C_{i+1} = G_i + P_i.C_i = G_i + P_i.G_{i-1} + P_i.P_{i-1}.C_{i-1} = \cdots$$

and so on, till we reach C_0.

Since all G_i, P_i and C_0 can be computed in parallel on arrival of the inputs, we can compute all sum and carry terms independently if we do not mind the added complexity.

Carry Look Ahead

$$C_{i+1} = G_i + P_i.C_i = G_i + P_i.G_{i-1} + P_i.P_{i-1}.C_{i-1} = \cdots$$

Unfortunately, static implementation of these gates has almost as much delay as the ripple carry implementation.

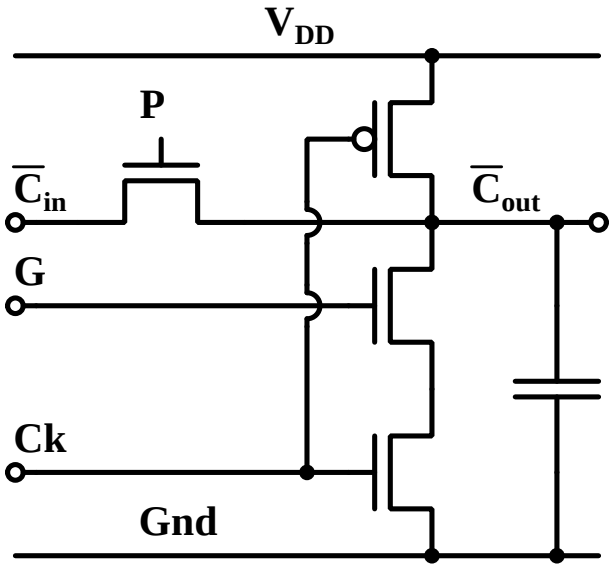
Therefore, the static implementation of computation of sum and carry terms as a logic expression depending on all A_i, B_i and C_0 is rarely used.
We can use these expressions for blocks of a small number of bits (say 4) and then propagate carry over these blocks.

Carry Look Ahead

Static implementation of look ahead carry is not really fast. We need to implement it over blocks of a small number of bits.

A dynamic implementation is useful and is widely used. It is known as the Manchester Carry Chain.

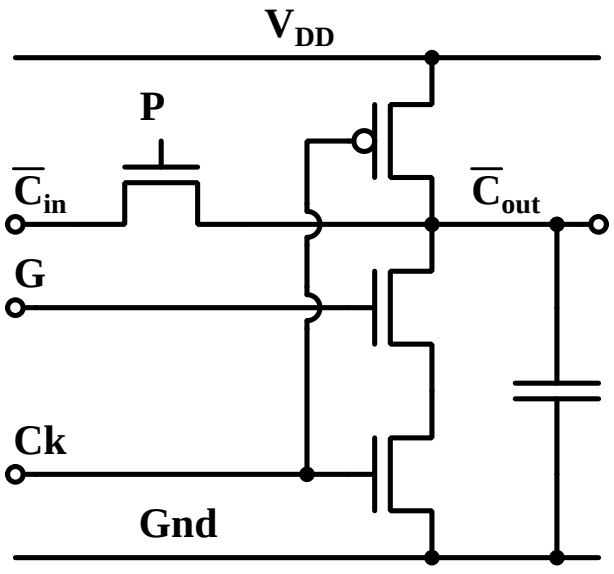
Manchester Carry Chain



When the clock is low, the output is unconditionally charged by the pMOS. When the clock goes high, the output will be pulled low if G = 1 or if P = 1 and $\overline{C_{in}} = 0$. In all other cases, the output will remain high. Thus this circuit implements the required logic.

This circuit can be concatenated for all bits and since P and G are ready before $\overline{C_{in}}$ arrives, the carry quickly ripples through from bit to bit.

Manchester Carry Chain as Carry Look Ahead



Notice that the nMOS logic can be interpreted as:

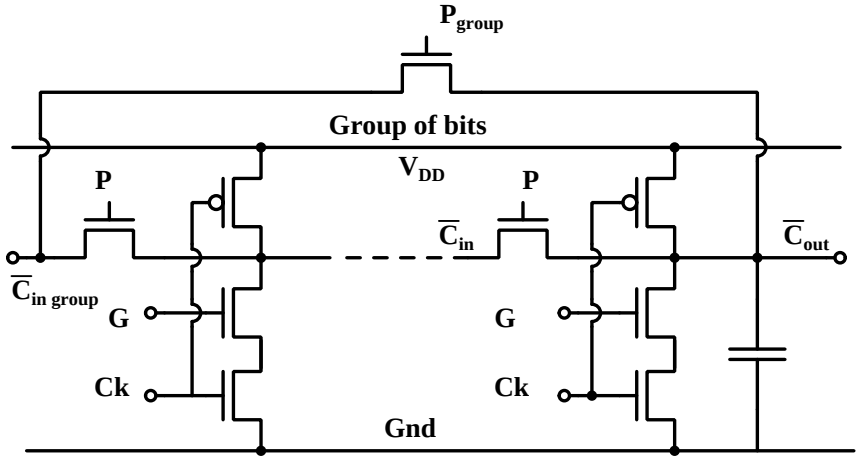
$$\overline{P.Cin + G}$$

where C_{in} itself has been recursively generated by similar logic.

- As in the static case, there is a limit to the number of bits which can be so connected.
- If P = 1 for many successive bits, the discharge path is through series connected pass transistors of all these gates. The discharge time for this critical path has an n^2 dependence.

Carry Bypass Adder

- The worst case for addition occurs when P = 1 for all bits and carry has to ripple through all bits.
- In carry bypass adder, we form groups of bits and if P = 1 for all members of a group, we pass on the carry input to this group directly to the input of the next group, without having to ripple through each bit.
- This improves the worst case delay of the adder.

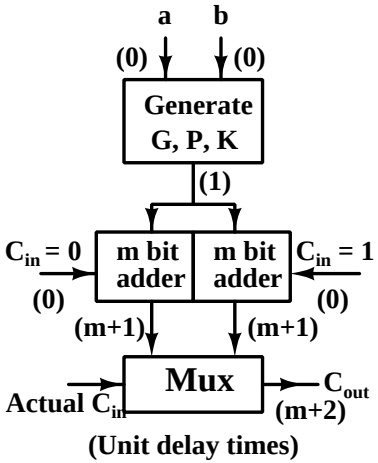


Carry Select Adder

An m bit carry select adder can be constructed as follows:

- We first compute the generate/propagate/kill signals for each bit (in parallel) from the input bits. Assuming unit gate delay model, this takes one unit of time.
- We use two m bit carry bypass adders. One of the adders assumes the carry input C_{in} to be 0, while the other assumes C_{in} to be 1. The two adders work in parallel and each takes m units of time.
- We now use a multiplexer controlled by the actual C_{in} to select the correct C_{out}. This takes one unit of time.
- The C_{out} of one such m bit adder will be used as the select input of the multiplexer of the next.
- The sum output of each bit is derived from P and C_{out} signals for the corresponding bit and appear one unit of time after C_{out} is available.

Carry Select Adder



Times of availability of various signals are noted in parentheses in the diagram.

- The two alternatives for the carry output are ready at (m+1) units of time.
- If the actual C_{in} is available at n units of time, the output will be available at (m+2) or (n+1), whichever is later.
- In case of 4 bit adders, this is at 6 units of time or at C_{in} arrival + 1, whichever is later.

Stacking in Carry Select adders

- The sub-adders in carry select adder can use any architecture.
- They could be Manchester carry chains, carry bypass or ripple carry adders.
- Obviously, these sub adders should not be very long, otherwise, their outputs will be ready after a long time and we shall lose the advantage of carry bypass additions.
- Then, how do we make long adders using carry select?

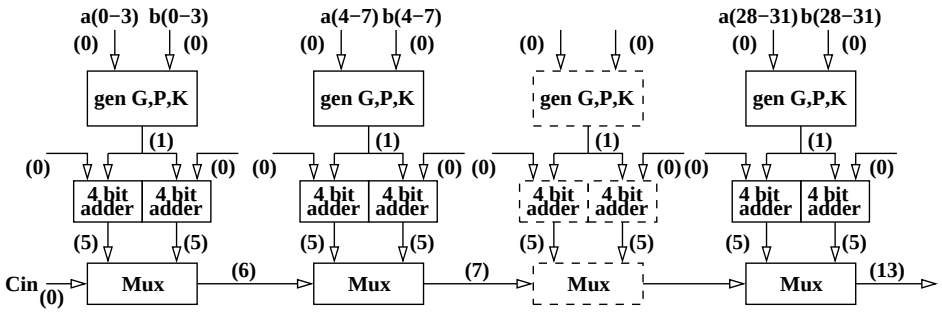
This is done by stacking several smaller carry select adders.

Linear Stacking

- We could stack several identical carry select adders.
- There is no need for carry select in the first stage, as C_{in} for this stage is available simultaneously with A_i and B_i.
- Every subsequent stage will have two sub-adders, one assuming $C_{in} = 0$, the other assuming $C_{in} = 1$.
- The correct output will be selected by the actual C_{in} when it arrives.
- Thus, after the first stage, each group of m bit adders will add only one unit of delay.
- This is much faster. However, the delay is still linear in number of bits.

Linear stacking: Example

A 32-bit adder made by cascading 8 4-bit carry select adders.



The sum generation will take another unit of time, so the overall results will be available in 14 units of time.

Bits	cy in	alt cy.s	cy out
0-3	0	5	6
4-7	6	5	7
8-11	7	5	8
12-15	8	5	9
16-19	9	5	10
20-23	10	5	11
24-27	11	5	12
28-31	12	5	13

Square-root Stacking

- Can we speed up the adder if we don't use the same no. of bits in every stage?
- In linear stacking, since all adders are identical, they are ready with their alternative outputs at the same time.
- But the carry arrives later and later at each successive group of carry select adders.
- We could have used this extra time to add up more bits in the later stages, and still be ready with the alternative results before carry arrives!
- Since the carry arrives one unit of time later at each successive group, each successive group could be longer by one bit.

Square-root Stacking

- We can do more bits of addition in the same time, if each successive stage is 1 bit longer than the previous one.
- Thus, the number of bits which can be added is given by

$$m = n_0 + n_0 + (n_0+1) + (n_0+2) + \cdots = n_0 + \frac{s(n_0 + n_0 + s - 1)}{2}$$

  where s is the number of stages following the first one without carry select.
- The total delay will be $n_0 + 1$ for the first stage. Each subsequent stage takes just 1 unit of time since the candidates for selection are available just in time. Thus the time take is just $n_0 + s + 1$ units. When $s \gg n_0$, we have $m \approx s^2/2$, while the time taken is nearly s.
- Thus the time taken to add m bits is $\approx \sqrt{2m}$

Square-root Stacking: Example

For a 32 bit adder, we could use a distribution like: 3,4,5,6,7,7.

Bits	carry in	carry alternatives	carry out
0-2	0	4	5
3-6	5	5	6
7-11	6	6	7
12-17	7	7	8
18-24	8	8	9
25-31	9	8	10

Less than 3 bits is not efficient for a carry select adder and Starting with more than 3 bits does not give faster performance.

Our sum will be ready at 11 - which is much faster. This gain will be even higher for wider additions.

Serial Adders

Up to now, we have been concerned with making fast adders, even at the cost of increased complexity and power.

In many applications, speed is not as important as low power consumption and low cost.

Serial adders are an attractive option in such cases.

A single full adder is used.

If numbers to be added are available in parallel form, these can be serialized using shift registers.

Serial Adders

- A single full adder adds the incoming bits. Bits to be added are fed to it serially, LSB first.
- The sum bit goes to the output while carry is stored in a flip-flop.
- Carry then gets added to the more significant bits which arrive next.
- Output can be converted to parallel form if needed, using another shift register.