Parallel Adder Design

Sekhar Mandal

September 12, 2020

Ripple Carry Adder

- Ripple carry adder mimics the way of manual addition. Addition is done stage by stage and carry is accumulated and it goes to the nest stage.
- To create a n-bit parallel adder, we have to cascade n full adders.
- The Connection will be such that carry output from the i^{th} stage will propagate as carry input to the $(i+1)^{th}$ stage.
- In the worst case, carry ripples through all the stages.

carry	1	1	1	0
	0	1	1	1
	0	0	0	1
	1	0	0	0

Ripple Carry Adder

A four bit ripple carry adder is shown below.

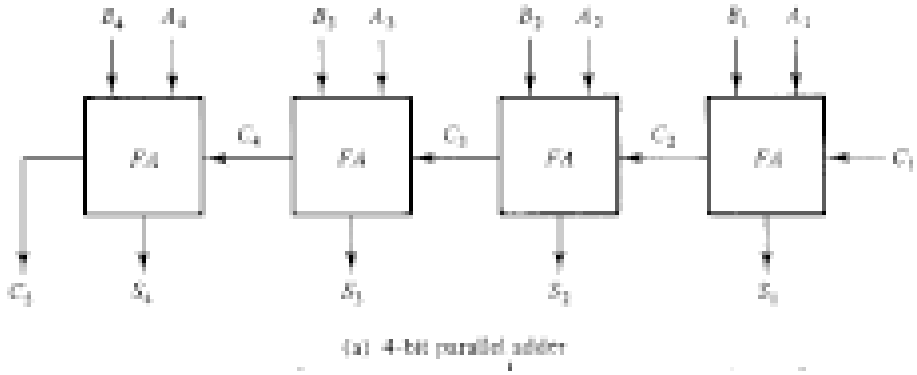


Figure: 4-bit parallel adder.

- Two number: $A_4A_3A_2A_1$ and $B_4B_3B_2B_1$
- Input carry C_1
- Sum: $S_4S_3S_2S_1$ and Output Carry: C_5
- Suppose to generate carry, a full adder needs δ time. Delay for $C_2 = \delta$, delay for $C_3 = 2 \times \delta$ . . . delay for $C_n = n \times \delta$

Parallel Sub-tractor

Observations

1. Computing $A - B$ is same as adding 2's complement of B to A
2. 2's complement = 1's complement + 1
3. XOR gate can be used as NOT gate with its one input is connected to 1.

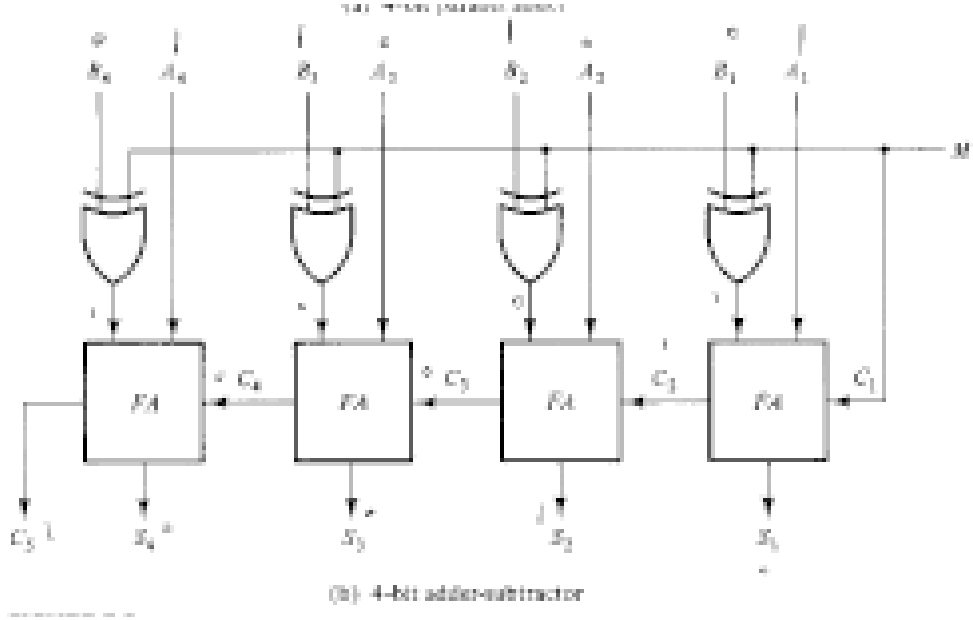


Figure: 4-bit Adder/Subtractor.

Drawback of Ripple Carry Adder

- The total delay of the circuit is proportional to number of bits n.
- Performance degradation for large value of n.
- The main bottleneck is carry, which propagates from one stage to the next stage.

Look-ahead Carry Adder

- The drawback of ripple carry adder can be overcome, if we generate all carry bits in parallel before actual addition starts.
- After initial delay, all addition can be done in parallel.
- The time complexity reduces from O(n) to O(1).
- Hardware complexity increases rapidly with n.

Consider the i^{th} stage of a parallel adder. We define two terms $G_i = A_i.B_i \Rightarrow$ carry generate, it produces an output carry when both A_i and B_i are 1, regardless of the input carry.

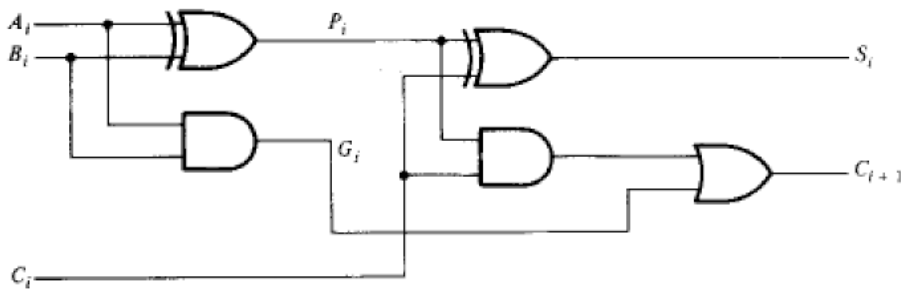


Figure: Full Adder Circuit.

Look-ahead Carry Adder

$C_{i+1} = G_i + P_i.C_i$
$P_i = A_i \oplus B_i$ is called carry propagate as it is associated with propagation of the carry from C_i to C_{i+1}.

- Consider 4 bit parallel adder, the Boolean function for the carry output of each stage can be written as:
- $C_2 = G_1 + P_1 C_1$
- $C_3 = G_2 + P_2 C_2 = G_2 + P_2(G_1 + P_1 C_1) = G_2 + P_2 G_1 + P_2 P_1 C_1$
- $C_4 = G_3 + P_3 C_3 = G_3 + P_3(G_2 + P_2 G_1 + P_2 P_1 C_1)$
  $C_4 = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 C_1$
- As the Boolean function of each output carry is expressed as sum of the product terms then, each function can be implemented with one level AND gates followed by an OR gate (or two level NAND gates)

Look-ahead Carry Adder

The Boolean functions C_2, C_3 and C_4 are implemented in look-ahead carry generator shown in figure. Note that here C_2, C_3 and C_4 are generated at the same time.

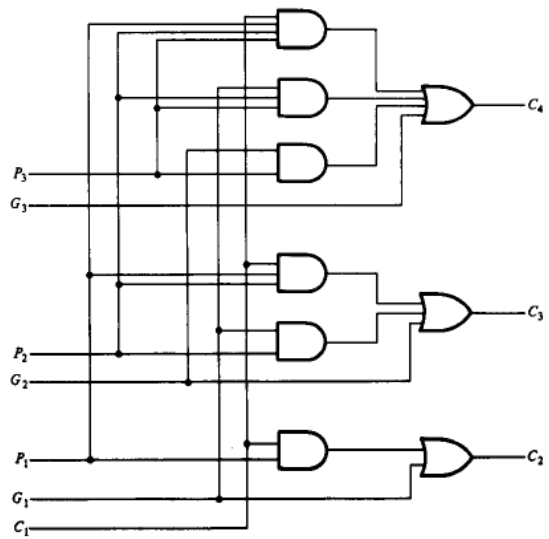


Figure: Logic diagram of a look-ahead carry generator.

Look-ahead Carry Adder

The circuit diagram of a 4-bit parallel adder with look-ahead carry generation scheme is shown below.

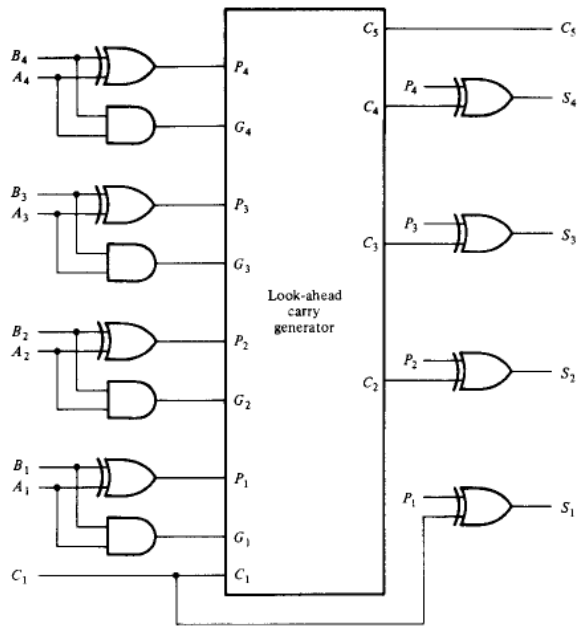


Figure: A 4-bit parallel adder with look-ahead carry.

- The first XOR gate generates P_i and AND gate produces G_i. All the P_i's and G_i's are generated in one gate-level.
- The carries are propagated through the look-ahead generator and applied as inputs of the second XOR gates.
- After P_i and G_i signals settle into their steady state values, all the outputs carry are generated after a delay of two gate-level.
- The outputs S_2 through S_4 have equal delay time.

Magnitude Comparator

- A magnitude comparator is a combinational circuit that compares two numbers, A and B, and determines their relative magnitudes. This circuit has three outputs to indicate $A > B$, $A = B$ and $A < B$.
- If we follow the classical method of design then, for two n-bit numbers, there are 2^{2n} entries in the truth table which is difficult to handle.
- Here, we shall follow an algorithm to design the circuit.

Multiplexer

- A multiplexer is an electronic switch that has a set of n input lines $(I_0, I_1, \ldots I_n)$, a set of m select lines $(s_0, s_1, \ldots s_m)$, and one output line.
- One of the input lines will be connected to the output line based on the bit combination of select lines.
- Usually, $n = 2^m$ and called 2^m-to-1 multiplexer.

Implementation of a 2-to-1 MUX using Gates

- The truth table of a 2-to-1 MUX is as follows:

s_0	I_1	I_0	Y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

- The simplified Boolean expression for Y is as follows:
  $Y = \overline{s_0} I_0 + s_0 I_1$

Implementation of a 4-to-1 MUX using Gates

- The Function table of a 4-to-1 MUX is as follows:

s_0	s_1	Y
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

- The circuit diagram of a 4-to-1 MUX is given below.

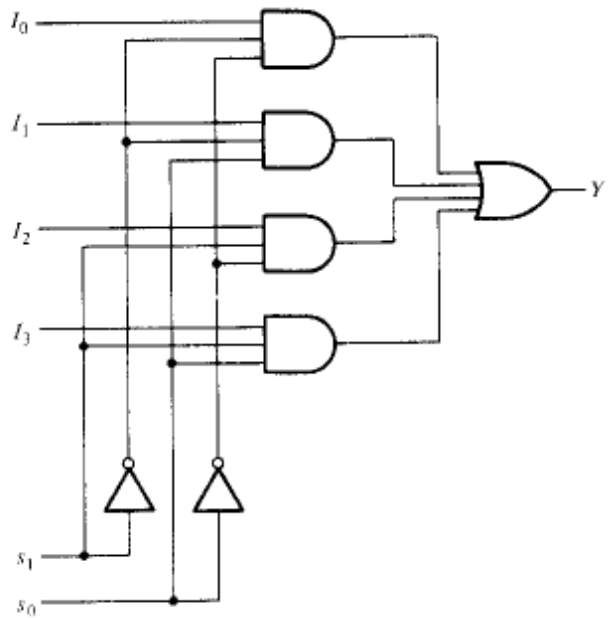


Figure: Logic diagram of a 4-to-1 MUX.

Implementation of Logic Functions using MUX

Implement an n variable function using 2^n-to-1 line MUX

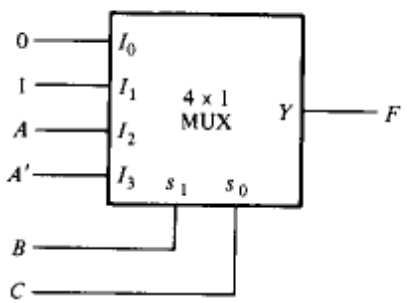
- Connect the n variables to the select lines.
- Connect the truth table output column values to the data inputs.
- Consider the following truth table of a function.

A	B	C	f
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

Implement an n-variable function using 2^{n-1}-to-1 lines MUX

- Connect (n -1) variables to the select lines.
- Connect the remaining variable, its complement, 0 and 1 to the data input.
- Use Implementation table for this purpose.

Consider the following function: $F(A, B, C) = \sum(1, 3, 5, 6)$.
Implement the function F using 4-to-1 line MUX.



(a) Multiplexer implementation

Minterm	A	B	C	F
0	0	0	0	0
1	0	0	1	1
2	0	1	0	0
3	0	1	1	1
4	1	0	0	0
5	1	0	1	1
6	1	1	0	1
7	1	1	1	0

(b) Truth table

	I_0	I_1	I_2	I_3
A'	0	①	2	③
A	4	⑤	⑥	7
	0	1	A	A'

(c) Implementation table

Implementation Arbitrary Functions using 2to-1 line MUX

- We repeatedly split a function into smaller sub-functions using Shannon's decomposition theorem.
- Shannon's Decomposition Theorem: states that an n-variable function $f(x_1, x_2, \ldots, x_n)$ can be decomposed with respect to any of the n-variables (say x_1) as
  $f(x_1, x_2, \ldots, x_n) = \overline{x_1} f(0, x_2, \ldots, x_n) + x_1(1, x_2, \ldots, x_n)$
  $= \overline{x_1} f_1^0 + x_1 f_1^1$
- Apply x_1 to the select line and f_1^0 and f_1^1

De-Multiplexer

A demultiplexer is a electronic switch that works in a reverse manner as compared to a multiplexer.

- It has a set of n output lines $(D_0, D_1, D_2, \ldots, D_{n-1})$.
- A set of m select lines $(s_0, s_1, s_2, \ldots, s_{m-1})$.
- One data input line I.
- The input data line will be connected to one of the output lines depending upon the bit combinations of the select lines.
- The output that are not selected are set to zero.
- Usually, $n = 2^m$, called 1-to-2^m demultiplexer.

Implementation of 1-to-2 DEMUX using gates

Table: Truth Table of 1-to-2 DEMUX

I	s_0	D_0	D_1
0	0	0	0
0	1	0	0
1	0	1	0
1	1	0	1

$D_0 = \overline{s_0} I$ and $D_1 = s_0 I$

Implementation of 1-to-4 DEMUX

Table: Truth table of 1-to-4 DEMUX

I	s_1	s_0	D_3	D_2	D_1	D_0
0	0	0	0	0	0	0
0	0	1	0	0	0	0
0	1	0	0	0	0	0
0	1	1	0	0	0	0
1	0	0	0	0	0	1
1	0	1	0	0	1	0
1	1	0	0	1	0	0
1	1	1	1	0	0	0

$D_0 = \overline{s_1}\ \overline{s_0}\ I$, $D_1 = \overline{s_1}\ s_0\ I$
$D_2 = s_1\ \overline{s_0}\ I$ and $D_3 = s_1\ s_0\ I$

DECODER

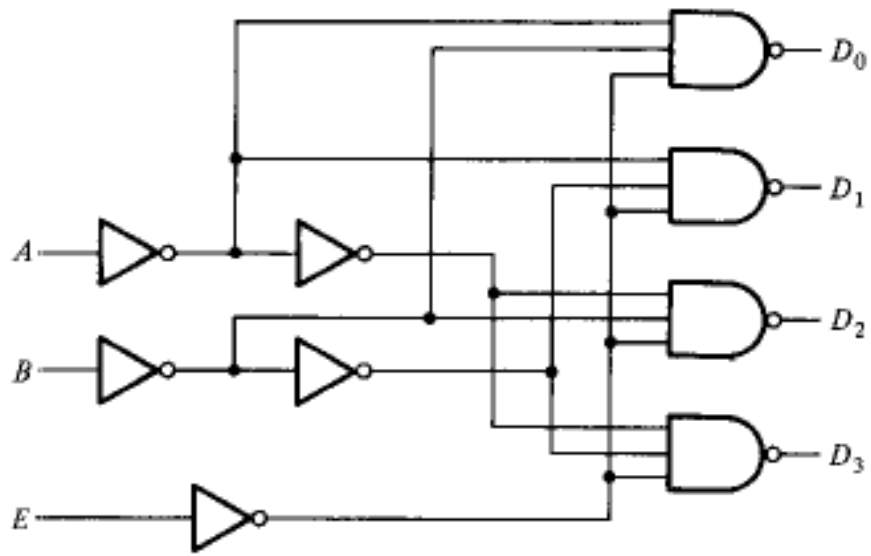
A decoder can be consider as a special case of a Demultiplexer.

- A DEMUX with data input line always set to 1.
- It has n input lines and 2^n output lines.
- Depending on the applied inputs, exactly one of the output lines is set to 1 which all the other lines are set to 0.
- In some decoder, the reverse convention is also followed. That is selected output line is set to 0 and other lines are set to 1.
-

2-to-4 Decoder

Table: Truth table of 2-to-4 Decoder.

E	A	B	D_0	D_1	D_2	D_3
1	X	X	1	1	1	1
0	0	0	0	1	1	1
0	0	1	1	0	1	1
0	1	0	1	1	0	1
0	1	1	1	1	1	0



Realization of Logic Functions using Decoders

We can use an n-to-2^n decoder and an OR gate to realize any logic function of n variable.

- The input variables are connected to the decoder inputs.
- All the outputs corresponding to the true minterms are fed to the inputs of the OR gate.
- OR gate output generate the function.

ENCODER

- An encoder typically has 2^n input lines and n output lines.
- Only one input line is at 1 at a particular time.
- The output lines contain the binary code corresponding to the input line which is at 1.

Table: Truth table of octal-to-binary encoder

Input								Output		
D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7	x	y	z
1	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	0	0	1	0	0	0	0	0	1	1
0	0	0	0	1	0	0	0	1	0	0
0	0	0	0	0	1	0	0	1	0	1
0	0	0	0	0	0	1	0	1	1	0
0	0	0	0	0	0	0	1	1	1	1

Encoder

The Boolean functions for the outputs of the encoder are as follows: $z = D_1 + D_3 + D_5 + D_7$

$y = D_2 + D_3 + D_6 + D_7$

$x = D_4 + D_5 + D_6 + D_7$

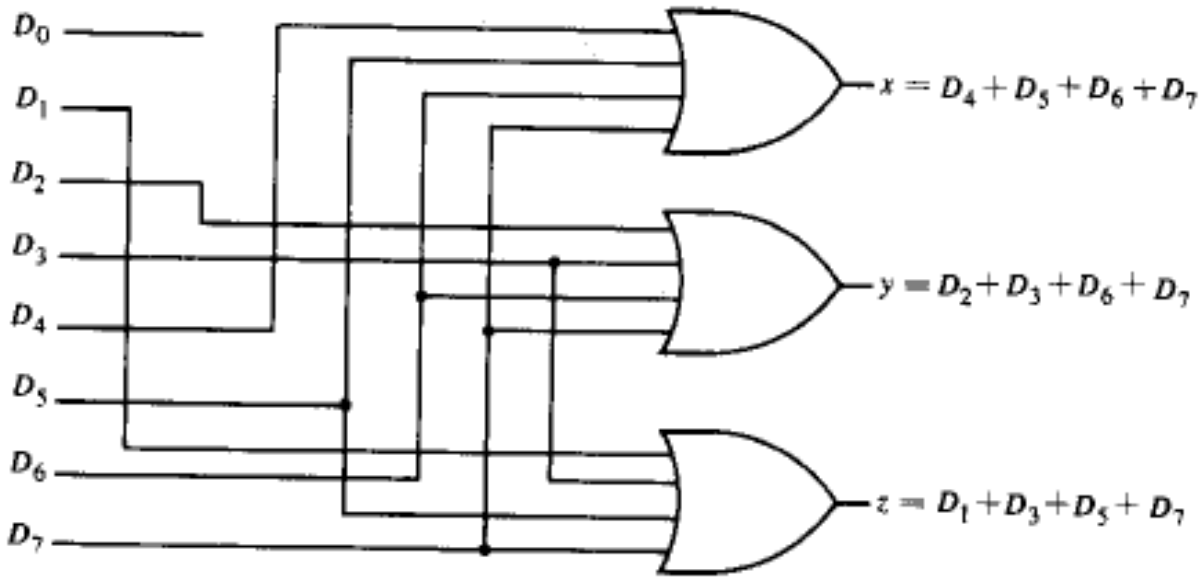


Figure: Logic diagram of Octal-to-Binary Encoder.

Priority Encoder

Basic Concept

- The input lines are assumed to request some service.
- When two inputs D_i and D_j $(i > j)$ request service simultaneously, line D_i is assumed to have higher priority than line D_j.
- The outputs of the encoder generate the binary code indicating which of the input lines requesting service has the highest priority.

Table: Truth Table of a 8-to-3 Priority Encoder

Input								Output		
D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7	x	y	z
1	0	0	0	0	0	0	0	0	0	0
x	1	0	0	0	0	0	0	0	0	1
x	x	1	0	0	0	0	0	0	1	0
x	x	x	1	0	0	0	0	0	1	1
x	x	x	x	1	0	0	0	1	0	0
x	x	x	x	x	1	0	0	1	0	1
x	x	x	x	x	x	1	0	1	1	0
x	x	x	x	x	x	x	1	1	1	1