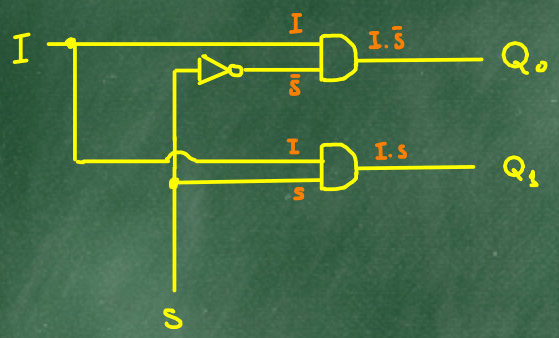
# Demultiplexing (DeMux)

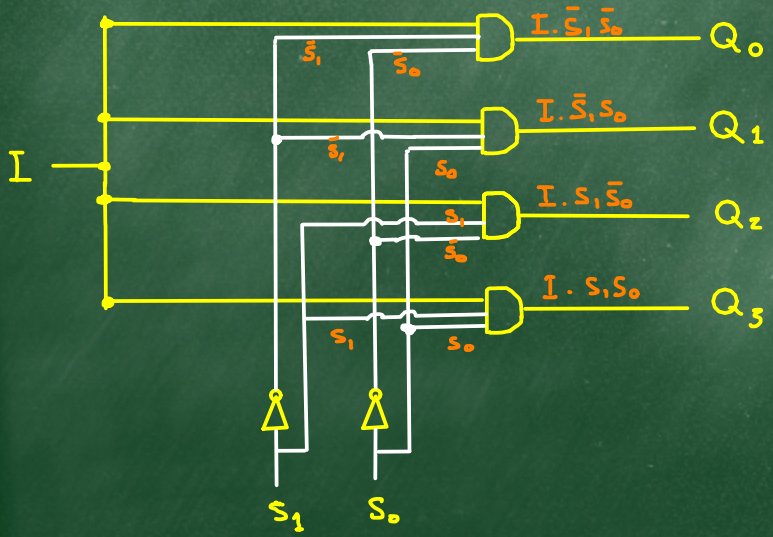
## Example : 1-2-line DeMux




$Q_0 = I.\bar{S}$

$Q_1 = I.S$

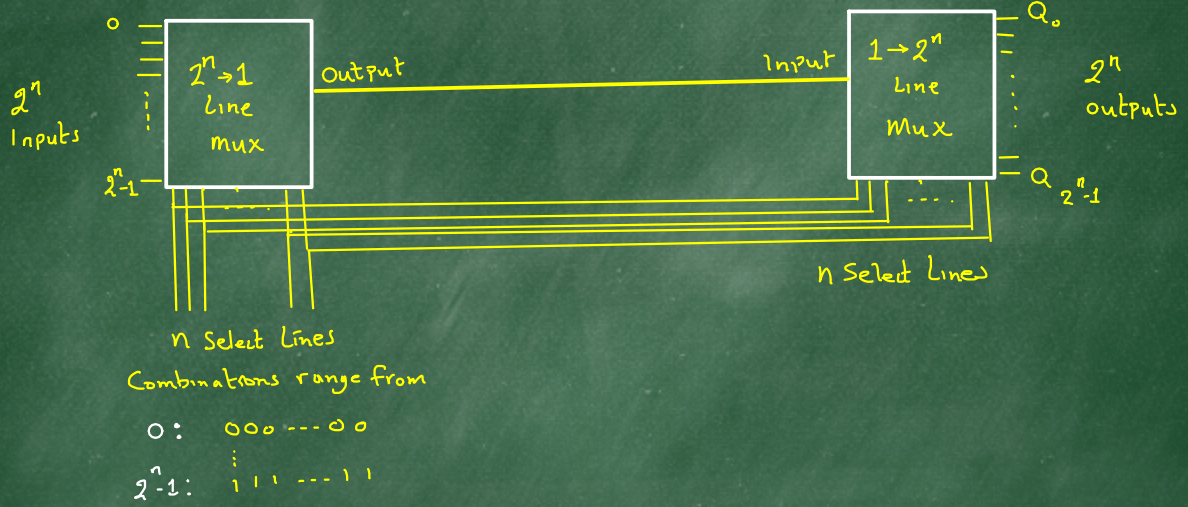
## Scaling-up : 1-4-line DeMux




| I | $S_1$ | $S_0$ | $Q_0$ | $Q_1$ | $Q_2$ | $Q_3$ |
|---|---|---|---|---|---|---|
| I | 0 | 0 | I | 0 | 0 | 0 |
| I | 0 | 1 | 0 | I | 0 | 0 |
| I | 1 | 0 | 0 | 0 | I | 0 |
| I | 1 | 1 | 0 | 0 | 0 | I |

Multiplexers and Demultiplexers can be used together to reduce the number of lines (wires) needed to transmit information from one place to another.

Here, $n+1$ lines replace $2^n$ lines, in sending a collection of $2^n$ bits from one place to another.



In addition to their usefulness in dynamically routing inputs to outputs and creating pathways through a circuit, you might have already begun to notice another powerful property of multiplexers:

the can be used to implement the function described by any Truth Table!

How?

Consider a simple example in which we'll implement $A \oplus B$ using a 4-1-line Mux:

Recall $A \oplus B$ can be expressed as the Truth Table:

| A | B | $A \oplus B$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

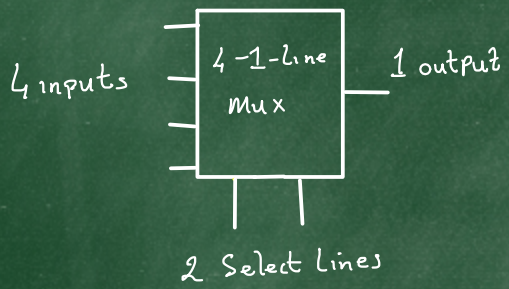
What do we see here?

2 Inputs

4 Combinations of the Inputs

1 Output per Input Combination

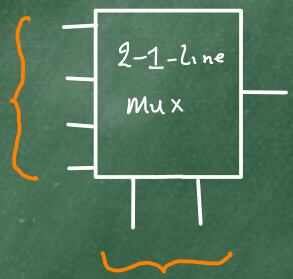
What do these characteristics have in Common with a 4-1-Line Mux?



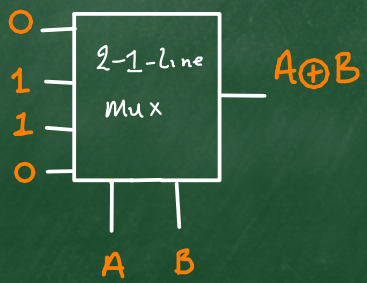

we have:

2 Select lines capable of generating 4 Combinations, each of which Identifies a specific Input (value) and associates it with the output

| A | B | A⊕B |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |




Thus:



## Example 2 : Implement a Full-ADDER Using Multiplexers

### Full-ADDER TRUTH TABLE:

| $C_{in}$ | A | B | Sum | Cout |
|---|---|---|---|---|
| 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 |

**Note 1:**

There are 3 Inputs So a $2^3$-1-Line mux is most appropriate.

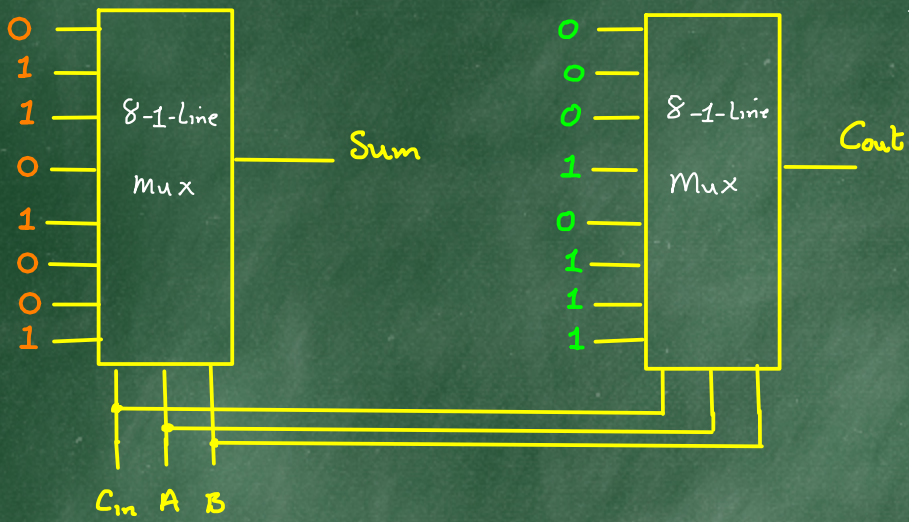
[i.e. 1 Input to each Select line.]

**Note 2:**

There are 2 outputs So we will use 2 Multiplexers - one for each output.

# Implementation

| Cin | A | B | Sum | Cout |
|---|---|---|---|---|
| 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 |




Note: while Implementing Boolean functions using multiplexers in the manner is 100% functionally correct, and simple, - it is most likely NOT the "most efficient" way of doing so.

In light of the Implementations described above, Can we Say anything profound about the Implementation of Combinatorial functions and the use of AND, OR and NOT ?