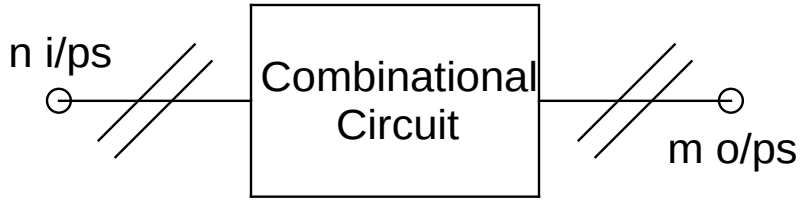
Types of Digital Logic Circuits:

* **Combinational** (or **Combinatorial**)
* **Sequential**

Combinational Circuit:

* Consists of ***logic gates*** whose outputs at any time are determined by the ***present*** combination of inputs
* Performs an operation specified ***logically*** by a set of ***Boolean functions***
* Has ***input variables***, ***logic gates***, and ***output variables***
* **Logic gates accept signals from inputs and generate signals to outputs**

* **Transforms binary information from given input data to required output data**



* ***For n i/p variables, 2^n possible binary i/p combinations***
* ***For each possible i/p combination, there is one possible o/p value***
  - $\Rightarrow$ Can be represented by a ***Truth Table***
  - $\Rightarrow$ Can also be described by m ***Boolean functions***, one for each o/p variable
  - $\Rightarrow$ ***Each o/p function is expressed in terms of n i/p variables***

* ***Examples***:

- **Adders**
- **Subtractors**
- **Multipliers**
- **Comparators**
- **Decoders**
- **Encoders**
- **Multiplexers (*MUX*)**
- **De-Multiplexers (*De-MUX*)**

Sequential Circuit:

* Employ ***storage elements*** in addition to ***logic gates***
* O/ps function of i/ps and ***state*** of these storage elements
* State of storage elements function of ***previous i/ps***
* ***O/ps depend not only on present values of i/ps, but also on past i/ps***
* Circuit behavior must be specified by a ***time sequence*** of ***i/ps*** and ***internal states***

How to identify a Combinational Circuit ?

* Would have ***only logic gates*** with ***no feedback paths or memory elements***
* **Feedback paths immediately denote sequential circuits**

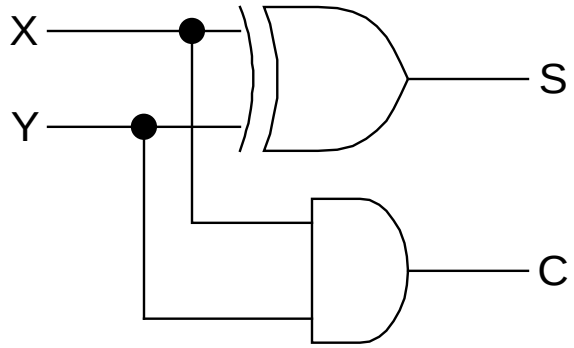
Adder:

* 0 + 0 = 0, 0 + 1 = 1, 1 + 0 = 1, 1 + 1 = 10
* First three cases have ***sums*** of 0 or 1, but no ***carry*** (or ***0 carry***)
* Fourth case has a ***carry*** of 1, which should be ***added*** to the next 2 significant bits
* ***Simple addition of 2 bits*** $\rightarrow$ **Half Adder** (***HA***)
* ***Addition of 3 bits*** (***2 significant bits and 1 carry***) $\rightarrow$ **Full Adder** (***FA***)
* ***Note***: **2 HAs can be used to implement a FA**

Half Adder (HA):

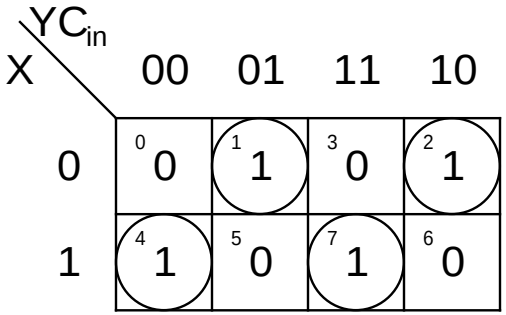
* ***2 binary i/ps and 2 binary o/ps*** [**Sum** (S) and **Carry** (C)]
* Thus, 0 + 0 = 0 (S) and 0 (C), 0 + 1 = 1 (S) and 0 (C), 1 + 0 = 1 (S) and 0 (C), 1 + 1 = 0 (S) and 1 (C)
* Thus, it can be clearly seen that for i/ps X and Y:

  $S = X \oplus Y = X'Y + XY'$, and $C = XY$



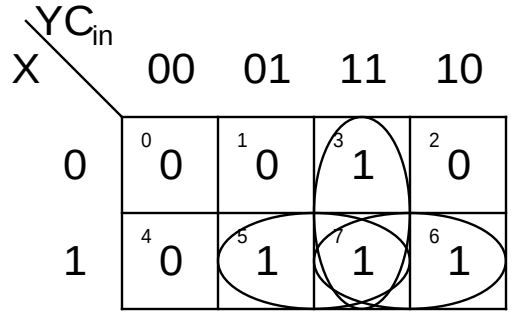
Full Adder (FA):

* ***Arithmetic sum of 3 bits***: **3 i/ps** (X, Y, and C_{in}) and **2 o/ps** (S and C_{out})



K-Map for S

$S = X'Y'C_{in} + X'YC'_{in} + XY'C'_{in} + XYC_{in}$



K-Map for C_{out}

$C_{out} = XC_{in} + XY + YC_{in}$

X	Y	C_{in}	S	C_{out}
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

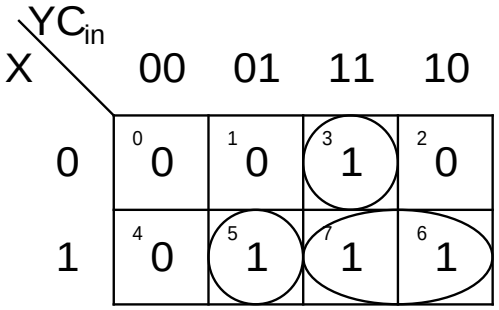
Optimization:

* ***Note***: $S = XYC_{in} + X'Y'C_{in} + XY'C'_{in} + X'YC'_{in}$

$= C_{in}(XY + X'Y') + C'_{in}(XY' + X'Y)$

$= C_{in}(X \oplus Y)' + C'_{in}(X \oplus Y)$

$= C_{in} \oplus (X \oplus Y)$



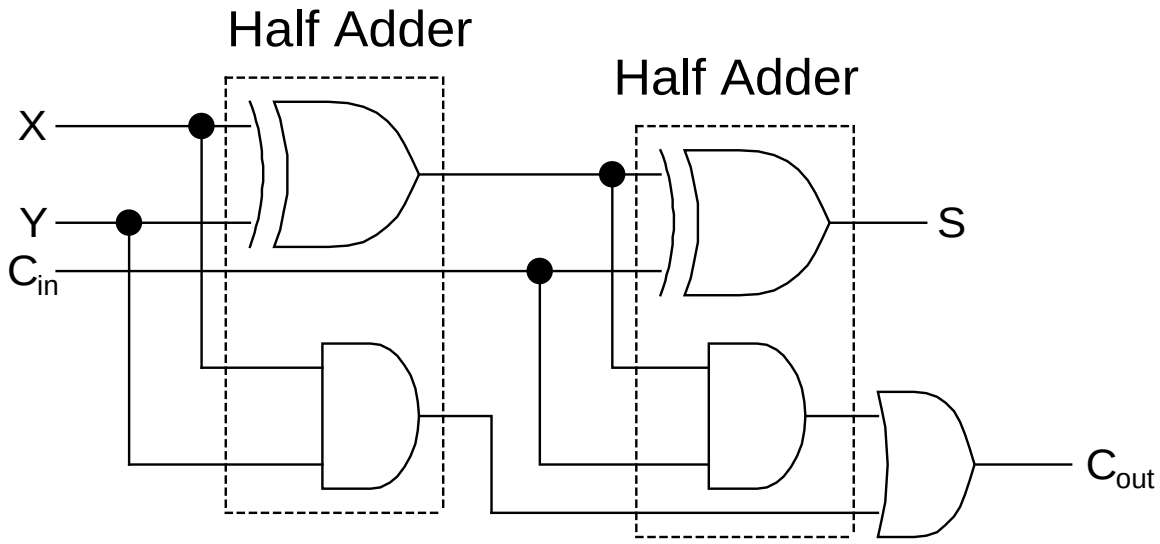
* Alternate K-map for C_{out} yields:

$C_{out} = XY'C_{in} + X'YC_{in} + XY$

$= C_{in}(XY' + X'Y) + XY$

$= C_{in}(X \oplus Y) + XY$

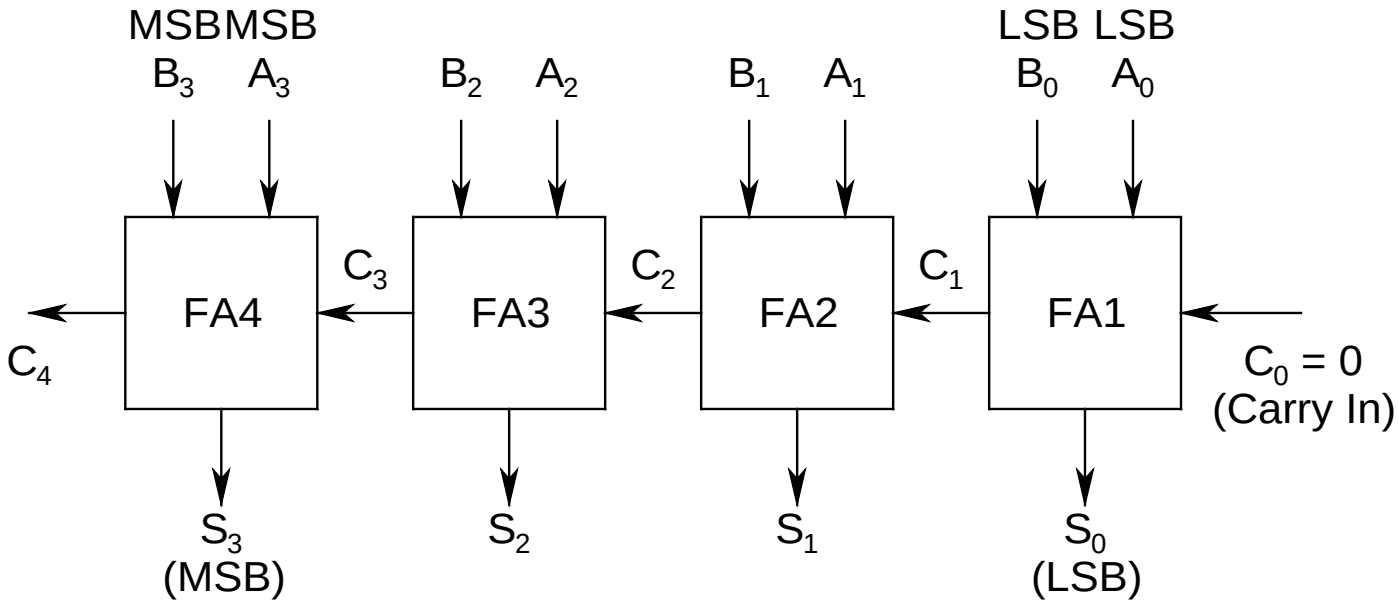
Implementation:



* $S = C_{in} \oplus (X \oplus Y)$ and $C_{out} = C_{in}(X \oplus Y) + XY$
* ***Note***: **A Full Adder can be constructed using two Half Adders and an OR gate**

Binary Adder:

* Performs ***arithmetic sum*** of two binary numbers ($B_3B_2B_1B_0$ and $A_3A_2A_1A_0$) and provides ***Sum*** ($S_3S_2S_1S_0$) and ***Carry Out*** (C_4)

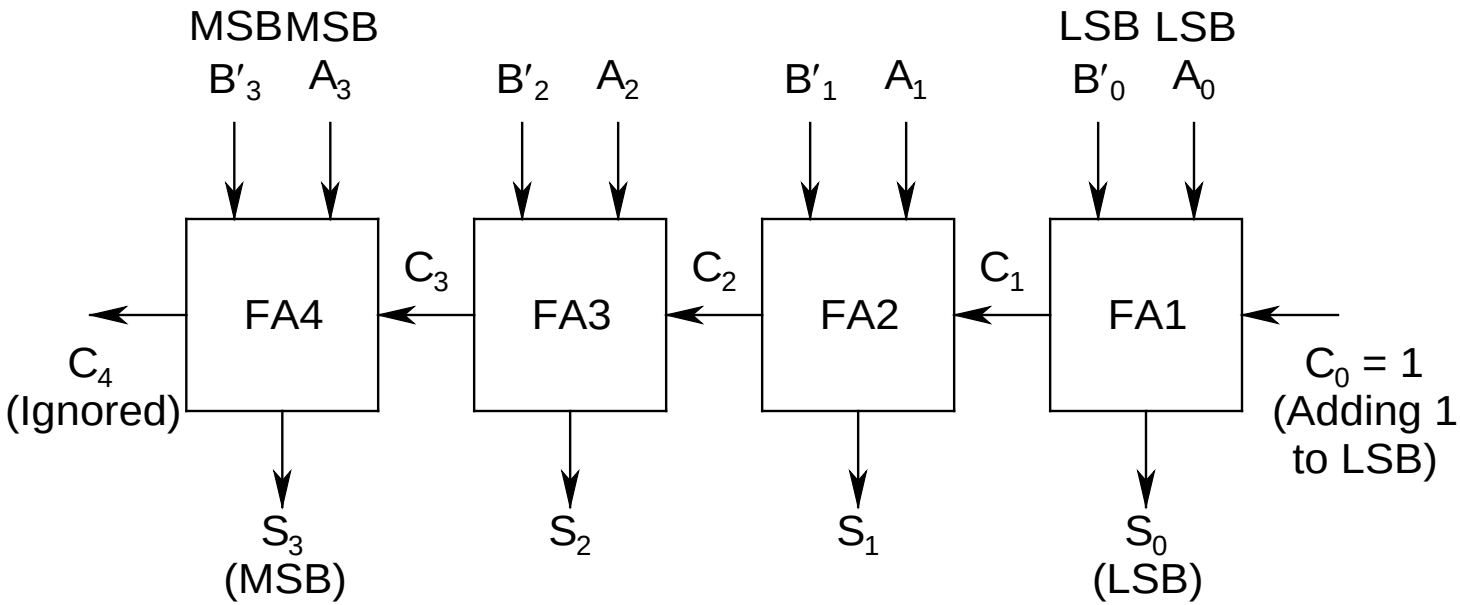


4-Bit Adder

* FAs ***connected in cascade***, with the o/p carry from each FA connected to the i/p carry of the next FA
* Also known as **Ripple Carry Adder**
* ***Limitation***: **Gate Delay :**
  - S_0 and C_1 generated first, then S_1 and C_2, then S_2 and C_3, and finally S_3 and C_4
    - $\Rightarrow$ Thus, all o/ps will be **valid** only after ***4 gate delays***
    - $\Rightarrow$ This is known as the **Carry Propagation Problem**

Binary Subtractor:

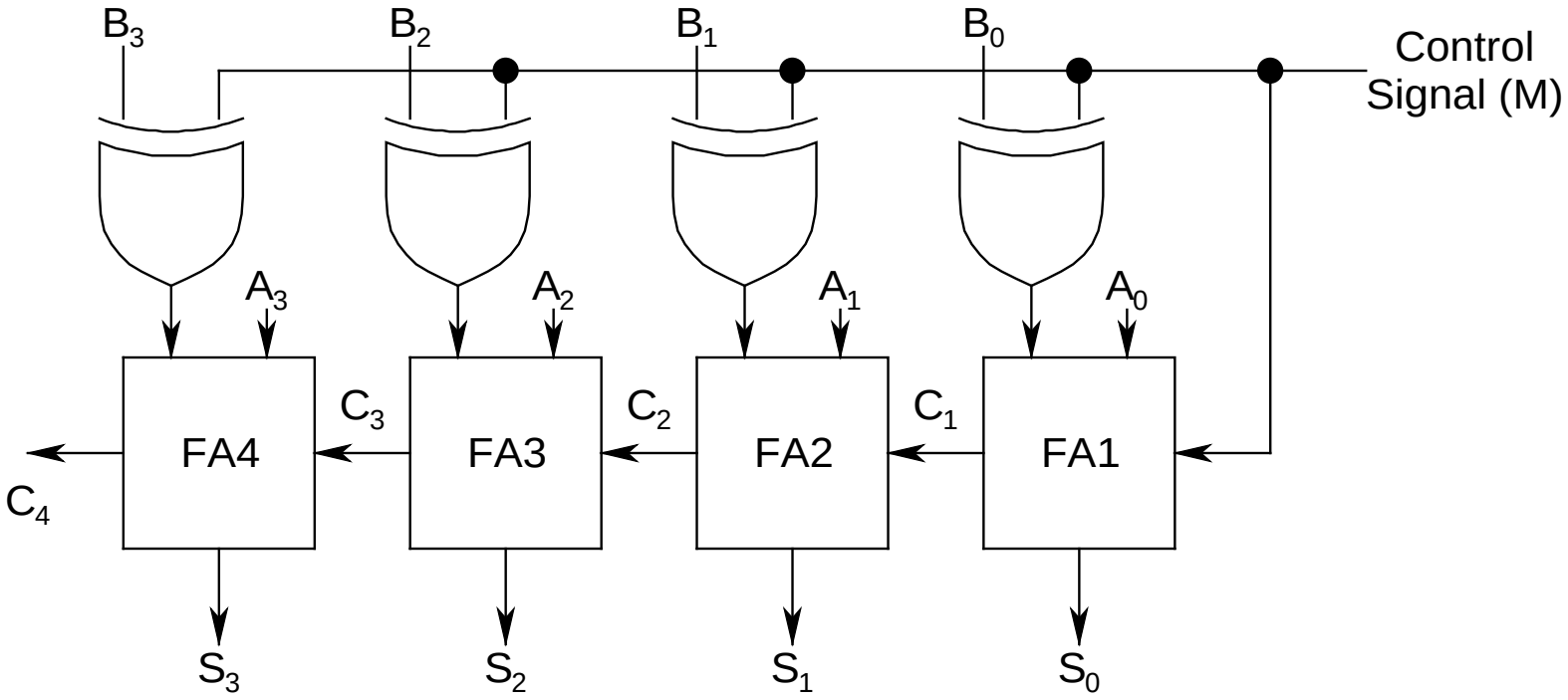
* ***Recall***: Subtraction is ***equivalent*** to adding two numbers, with the number to be subtracted expressed in ***2's complement notation***
* ***2's complement***: **Invert all bits and add 1 to LSB**



4-Bit Subtractor (Subtracting $B_3B_2B_1B_0$ from $A_3A_2A_1A_0$)

Binary Adder/Subtractor:

* ***Uses a Control Signal*** M (**= 0 for Adder** and **= 1 for Subtractor**) (***Note***: $1 \oplus B = B'$, $0 \oplus B = B$)



4-Bit Adder/Subtractor (M = 0: Adder, M = 1: Subtractor)

Binary Multiplier:

* ***Note***: Multiplication is an ***AND*** operation, i.e., the product will be 1 only if both the bits are 1

* ***Example***: B_1B_0 multiplied by A_1A_0

$$
\begin{array}{cccc}
 & & B_1 & B_0 \\
 & & A_1 & A_0 \\
\hline
 & & A_0B_1 & A_0B_0 \\
 & A_1B_1 & A_1B_0 & \\
\hline
S_3 & S_2 & S_1 & S_0
\end{array}
$$

* $S_0 = (A_0$ **AND** $B_0)$

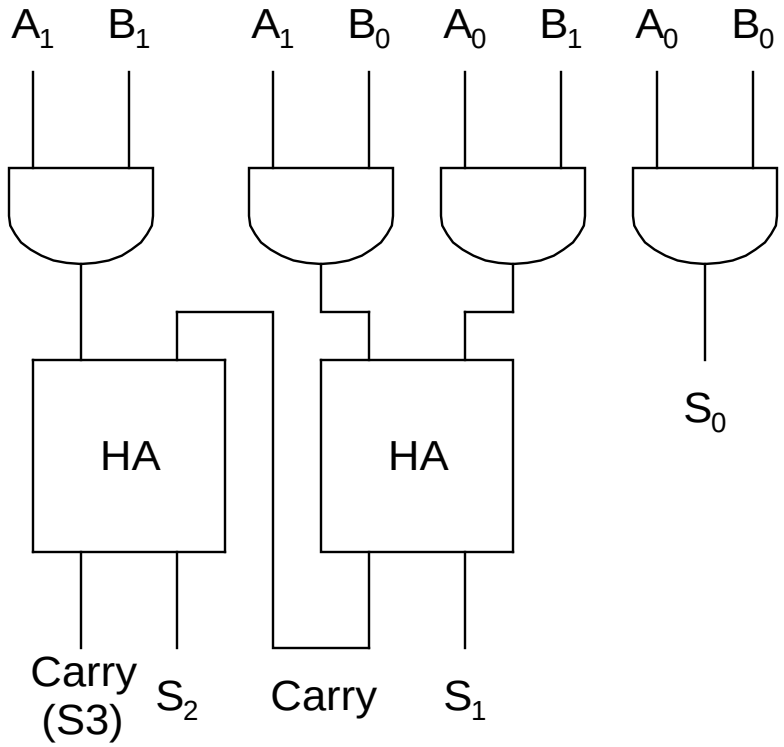
* $S_1 = (A_0$ **AND** $B_1)$ **ADD**

$(A_1$ **AND** $B_0) \rightarrow$ can be implemented using an ***Half Adder*** (HA)

* S_2 = (A_1 **AND** B_1) **ADD** (Any Carry of S_1)

* S_3 = Carry of S_2 $\rightarrow$ S_2 and S_3 can be implemented by using ***another HA***

Implementation:



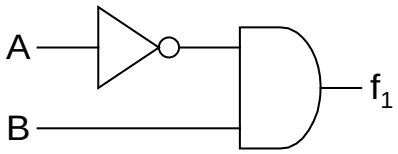
Magnitude Comparator:

* ***Compares*** 2 bits A and B, and produces ***3 outputs*** (f_1, f_2, and f_3 respectively) depending on whether: i) A < B, ii) A = B, or iii) A > B

i) A < B

A \ B	0	1
0	0	1
1	0	0

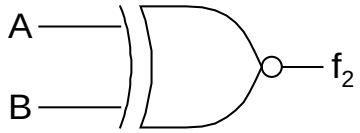
$f_1 = A'B$



ii) A = B

A \ B	0	1
0	1	0
1	0	1

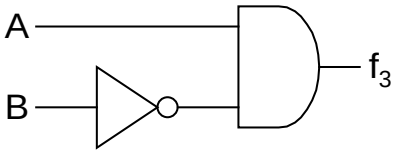
$f_2 = (A \oplus B)'$



iii) A > B

A \ B	0	1
0	0	0
1	1	0

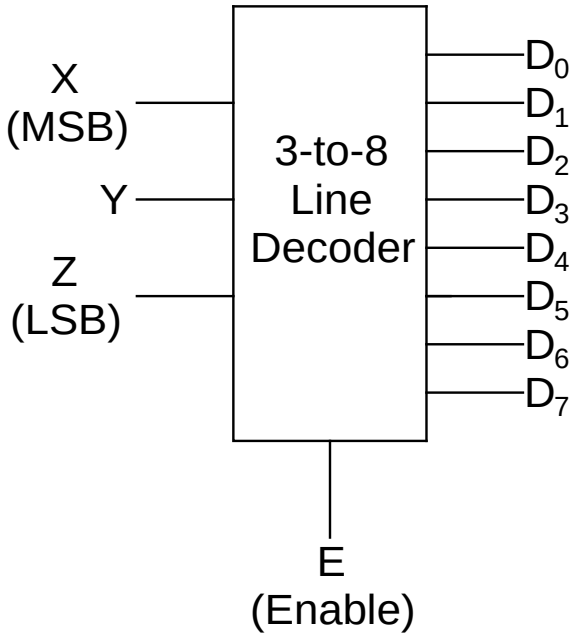
$f_3 = AB'$



Decoder:

* ***Note***: Binary code of ***n bits*** is capable of representing ***2^n coded information***
* Decoder is a ***combinational circuit*** that ***converts binary information from n i/p lines to 2^n o/p lines***
* May have ***fewer than 2^n o/ps***, if some of the combinations are ***unused***
* Called **n-to-m Line Decoder**, where $m \leq 2^n$

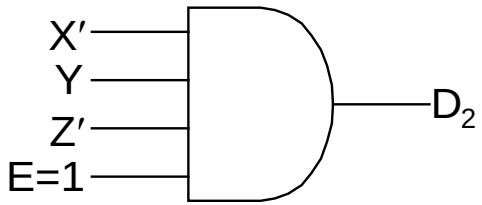
* ***Example***: **3-to-8 Line Decoder**, where for inputs XYZ (X = MSB, Z = LSB), ***one of the output lines D_0-D_7 will go high in sequence***
  - XYZ = 000, D_0 = 1, D_1-D_7 = 0
  - XYZ = 001, D_1 = 1, D_0 and D_2-D_7 = 0
  - ⋮
  - XYZ = 111, D_7 = 1, D_0-D_6 = 0
* Sometimes, uses ***Enable*** (E) signal: For ***E = 0***, ***all o/ps are disabled*** (0), while for ***E = 1***, the decoder circuit is ***enabled***

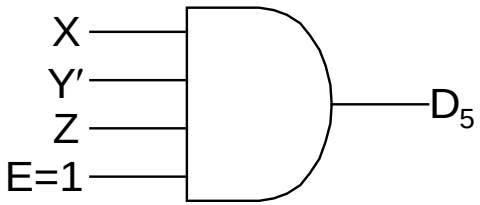


Positive Logic:

E = 0 (Disable): all D lines 0

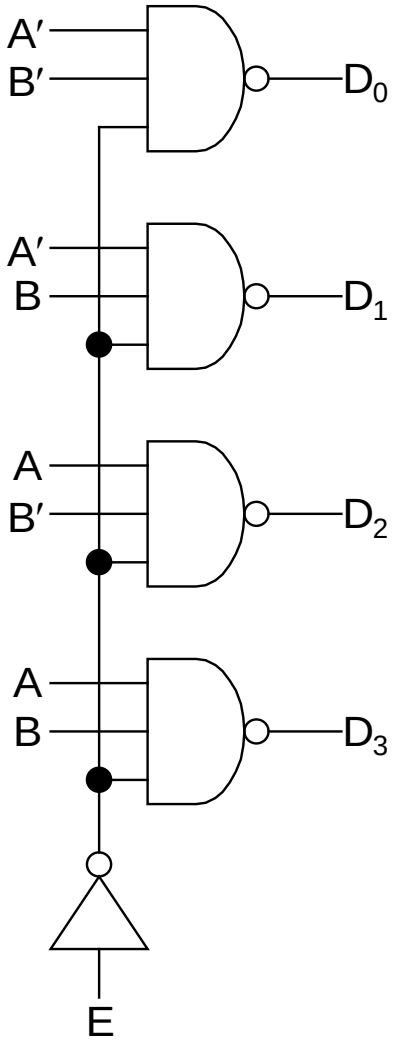
E = 1 (Enable): depending on X, Y, and Z, only one of the D lines will go high





2-to-4 Line Decoder Using NAND Gates:

* Uses **negative logic** (***E = 0 enables the decoder***, while ***E = 1 disables it***)
* ***The selected D line goes low under this case***
* ***Recall***: **NAND gate o/p is 0 if and only if all the inputs are 1 : If any of the i/ps is 0, o/p is 1**
* Since ***E = 0*** enables the circuit, it is also known as **Active-Low Enable Circuit**



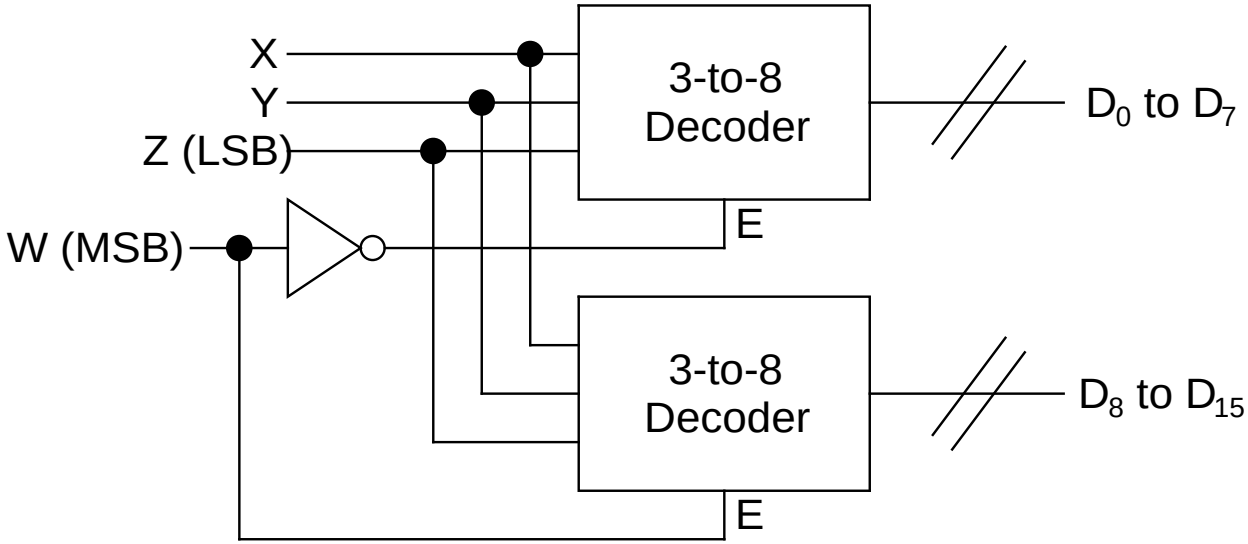
Truth Table:

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

* ***The first row corresponds to disabled decoder with all output lines high*** **(i/ps DON'T CARE)**

Constructing Large Decoders:

Example: Construction of a **4-to-16 Line Decoder** using two **3-to-8 Line Decoders**:



Uses positive logic: When W = 0, top decoder enabled and bottom decoder disabled: Outputs D_0 to D_7 correspond to XYZ = 000 to 111; and when W = 1, top decoder disabled and bottom decoder enabled: Outputs D_8 to D_{15} correspond to WXYZ = 1000 to 1111 (One of the o/p lines D_0-D_{15} goes high depending on WXYZ)

Combinational Logic Implementation Using Decoders:

Example: **Full Adder**:

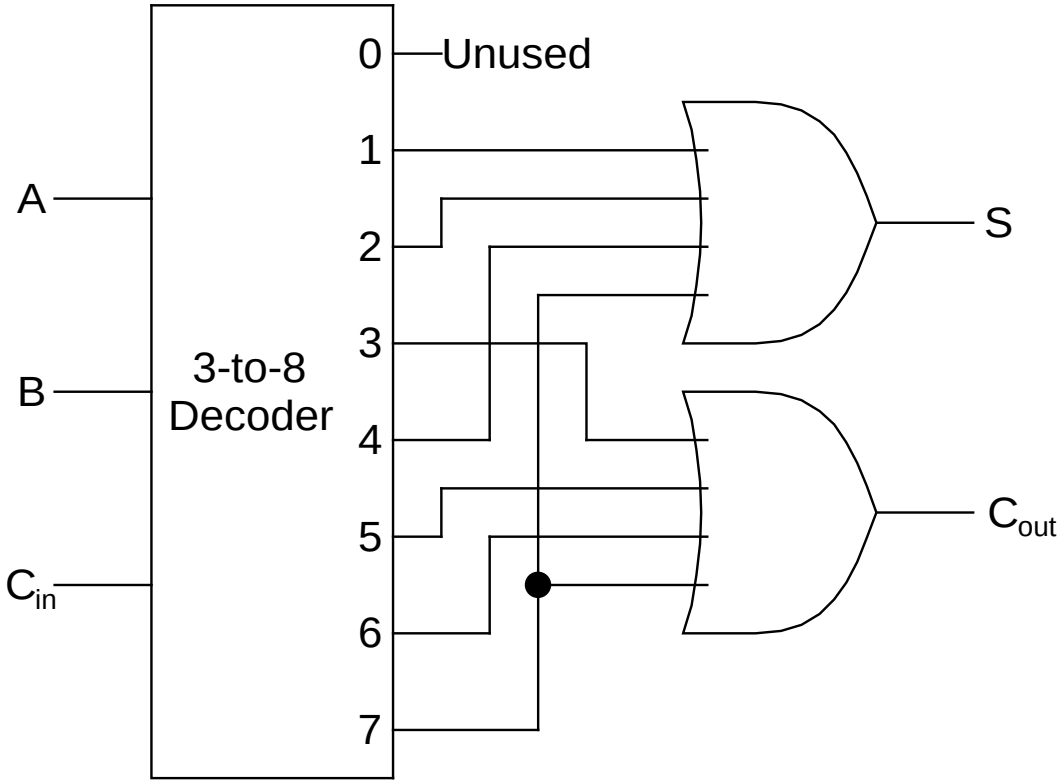
A	B	C_{in}	S	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

A \ BC_{in}	00	01	11	10
0	^{0}0	^{1}1	^{3}0	^{2}1
1	^{4}1	^{5}0	^{7}1	^{6}0

$S = \Sigma_m(1,2,4,7)$

A \ BC_{in}	00	01	11	10
0	^{0}0	^{1}0	^{3}1	^{2}0
1	^{4}0	^{5}1	^{7}1	^{6}1

$C_{out} = \Sigma_m(3,5,6,7)$



Full Adder Using 3-to-8 Line Decoder
(May or may not have Enable signal)

Encoder:

* Performs ***inverse*** operation of a Decoder
* Has 2^n ***or fewer i/p lines***, and ***n o/p lines***
* **O/p lines generate binary code corresponding to i/p values**
* ***Only one i/p should be high, and corresponding to that i/p, only one particular o/p combination will be generated***
* **Uses Active High logic**
* ***May or may not have Enable signal***
* ***Anomaly***: **When all o/ps are 0, it may be due to either the least significant i/p is 1, or all i/ps 0**

Example: **Octal-to-Binary Encoder**:

* ***Has eight i/ps (I_7 -I_0) and three o/ps (O_2 -O_0)***

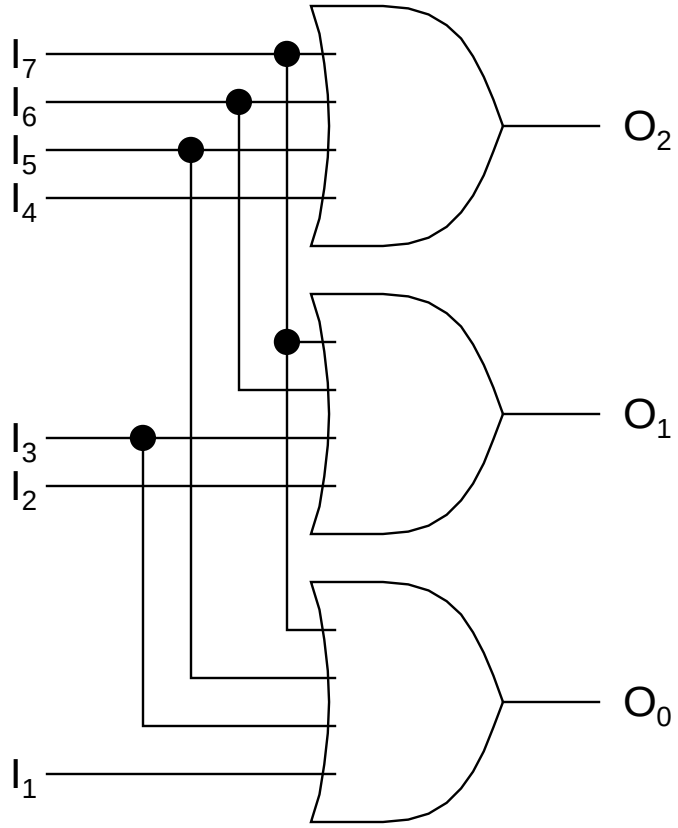
I_7	I_6	I_5	I_4	I_3	I_2	I_1	I_0	O_2	O_1	O_0
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	1	1	0
1	0	0	0	0	0	0	0	1	1	1

* From the ***Truth Table***: $O_0 = I_1 + I_3 + I_5 + I_7$,

$O_1 = I_2 + I_3 + I_6 + I_7$, and $O_2 = I_4 + I_5 + I_6 + I_7$

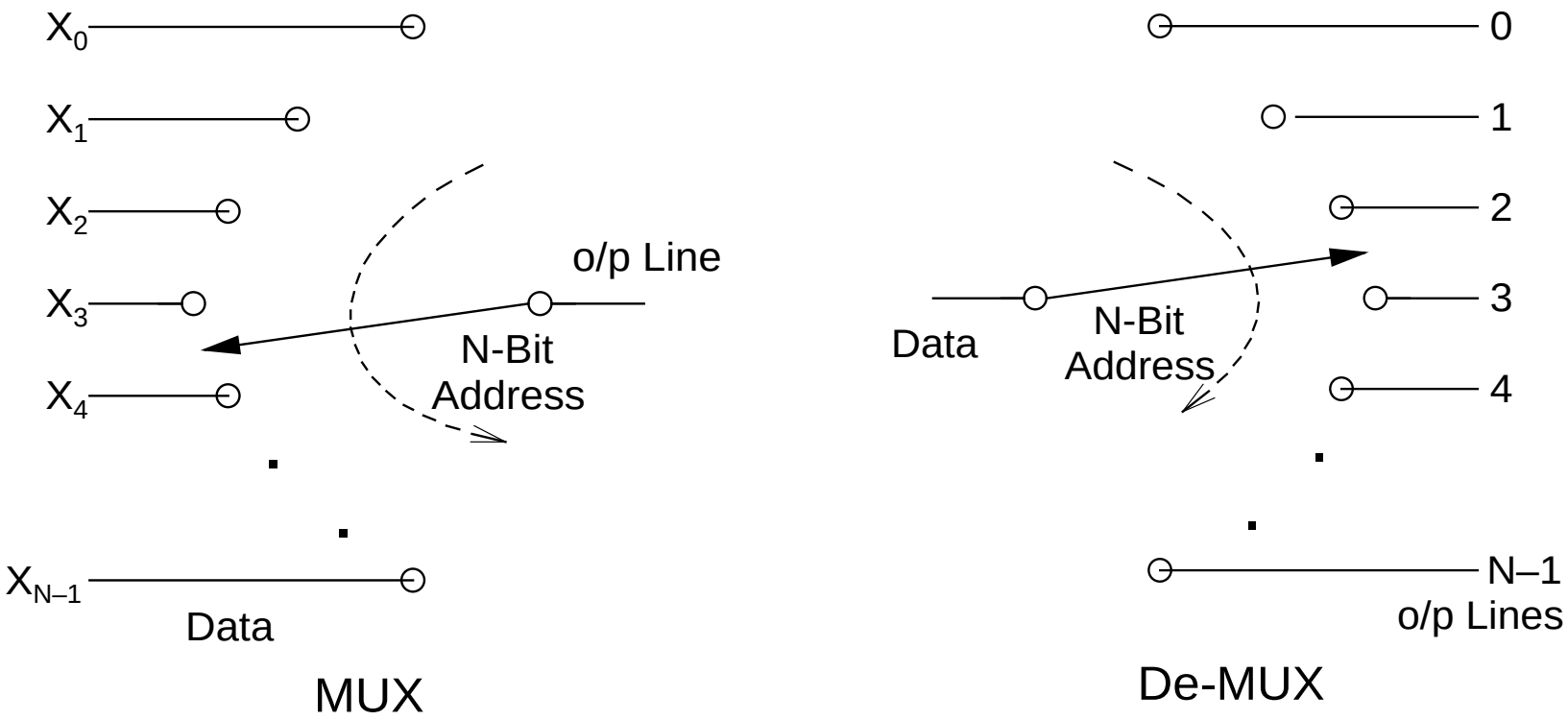
* ***Implementation***:

I_0: Not Used
Note: O_2-O_0 all are 0, either for I_0 = 1, or all i/ps 0
$\Rightarrow$ Anomaly



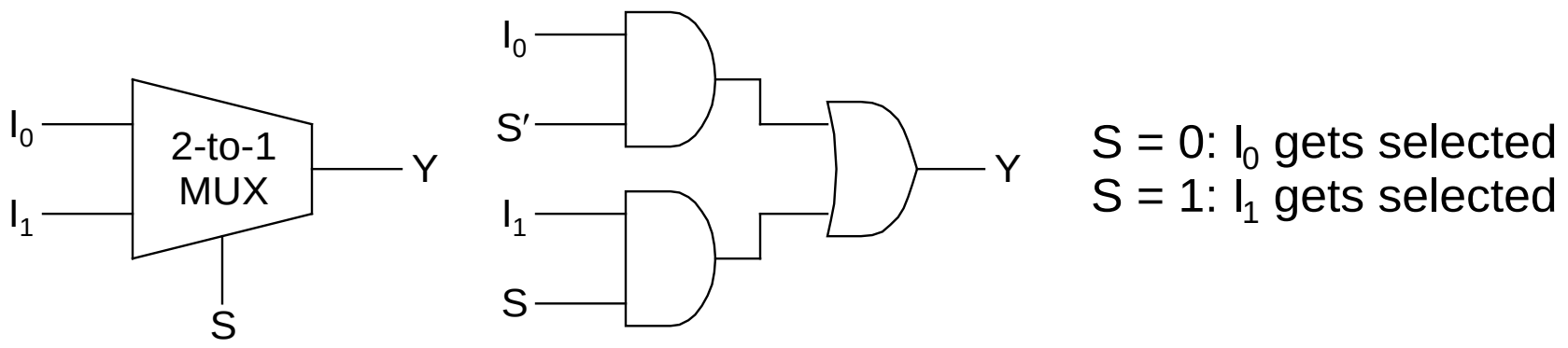
Mutiplexer (MUX) and De-Multiplexer (De-MUX):

*** MUX**: ***Many-to-1***, and **De-MUX**: ***1-to-Many***



Multiplexer (*MUX*):

* Also known as **Data Selector**
* ***Selects binary information from one of many i/p lines and directs it to a single o/p***
* **Needs Selection Lines to select which i/p goes to o/p**
* ***2^n i/p lines and n Selection Lines***
* ***Example*: 2-to-1 Line MUX:**



Boolean Function Implementation Using MUX:

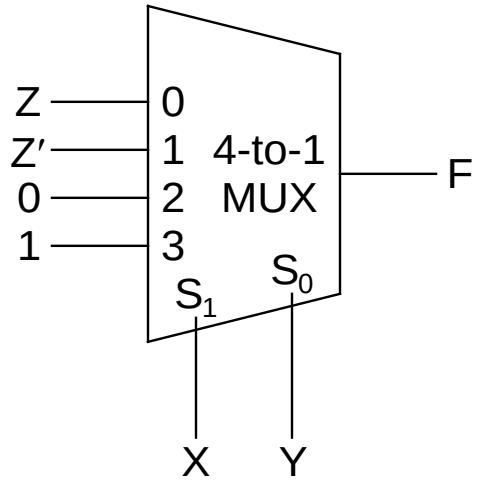
* If proper i/ps can be routed to the o/p, then a ***MUX*** can implement ***Boolean functions***
* **Individual minterms can be selected by data i/ps**
* ***A Boolean function having n variables can be implemented with a MUX having (n – 1) Selection i/ps***

* **The first (n – 1) variables of the function are connected to the Selection i/ps of the MUX**
* ***The remaining single variable of the function is used for the data i/ps***
* **If the single varible is denoted by Z, then the data i/ps of the MUX would be Z, Z′, 1, or 0**

Example : F(X,Y,Z) = $\sum_m (1,2,6,7)$:

MSB LSB

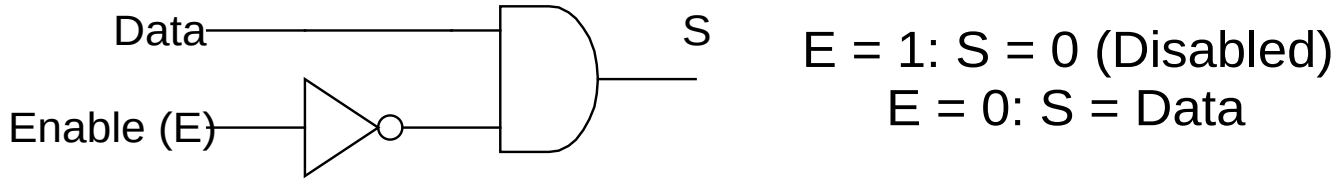
X	Y	Z	F	
0	0	0	0	
0	0	1	1	F = Z
0	1	0	1	
0	1	1	0	F = Z′
1	0	0	0	
1	0	1	0	F = 0
1	1	0	1	
1	1	1	1	F = 1



Exercise: Implement F(A,B,C,D) = $\pi_M (0,2,4,5,9,13,15)$ using an 8-to-1 MUX

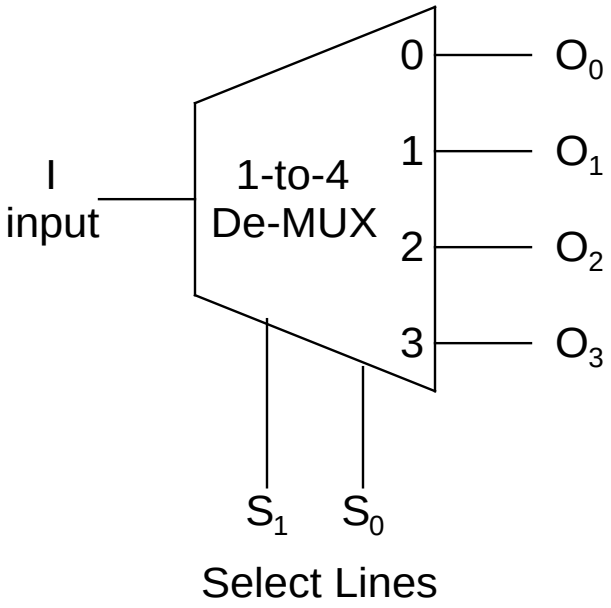
De-Multiplexer (*De-MUX*):

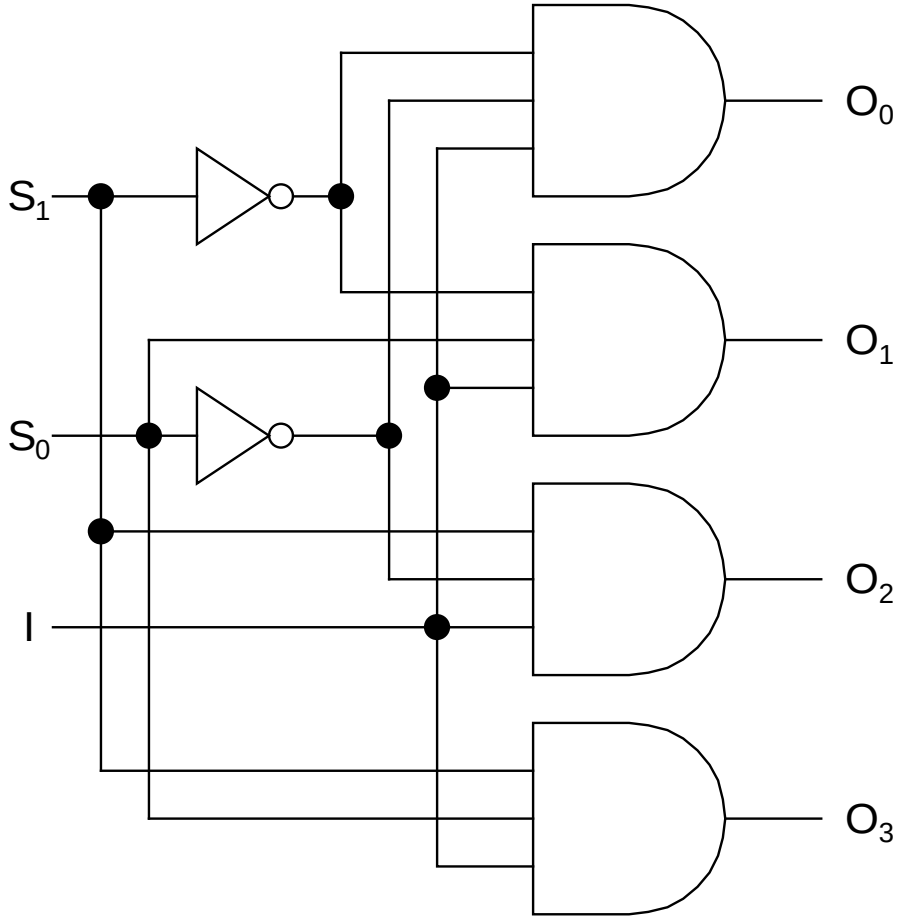
* Also known as **Data Router**
* ***Selects binary information from one i/p line and routes it to one of many o/p lines***
* **Needs Selection lines to select the o/p line to which the i/p gets routed**
* ***2^n o/p lines and n selection lines***
* **(Decoder + Enable) $\leftrightarrow$ De-MUX**



Example : 1-to-4 De-MUX:

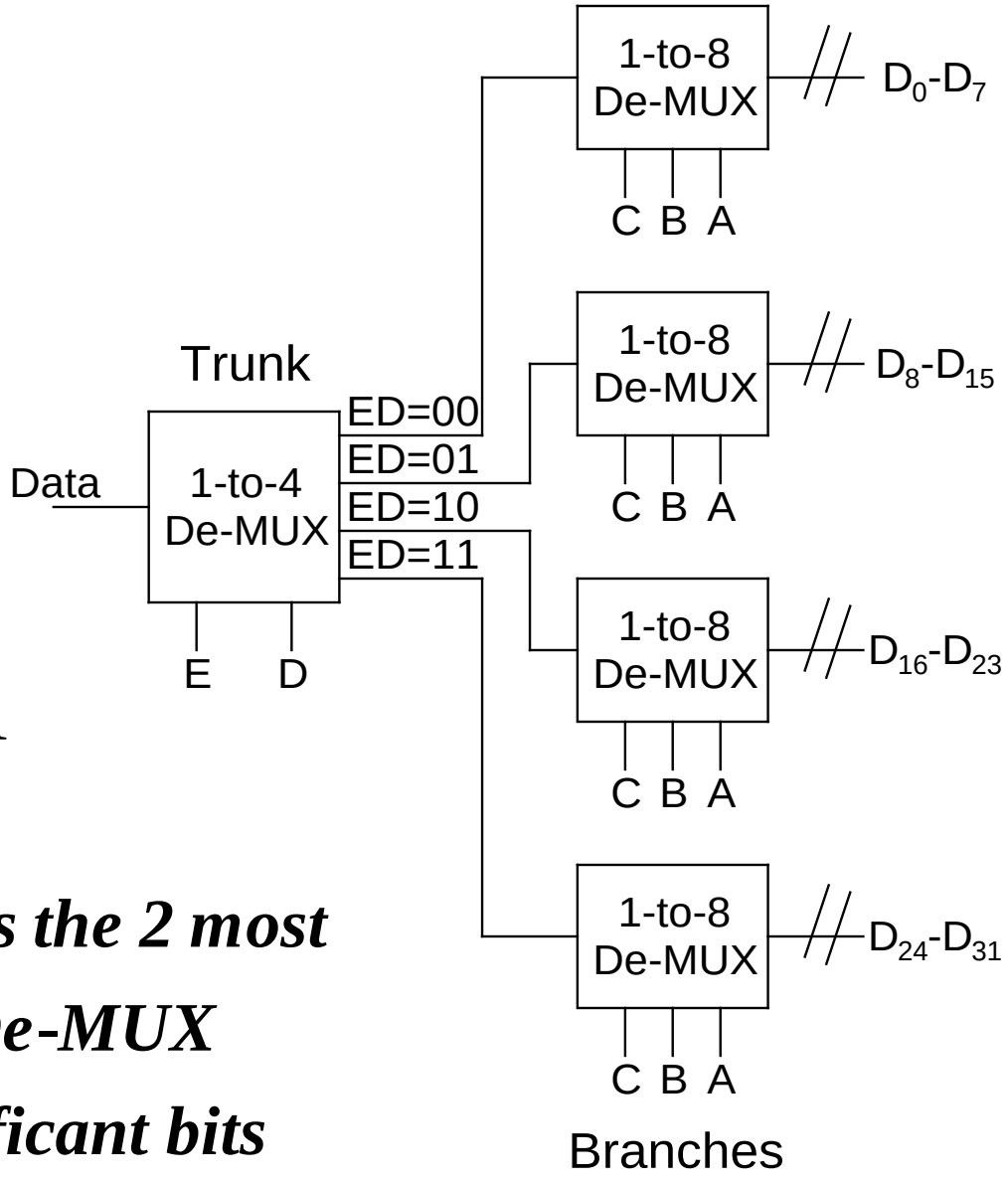
S_1	S_0	I goes to
0	0	O_0
0	1	O_1
1	0	O_2
1	1	O_3



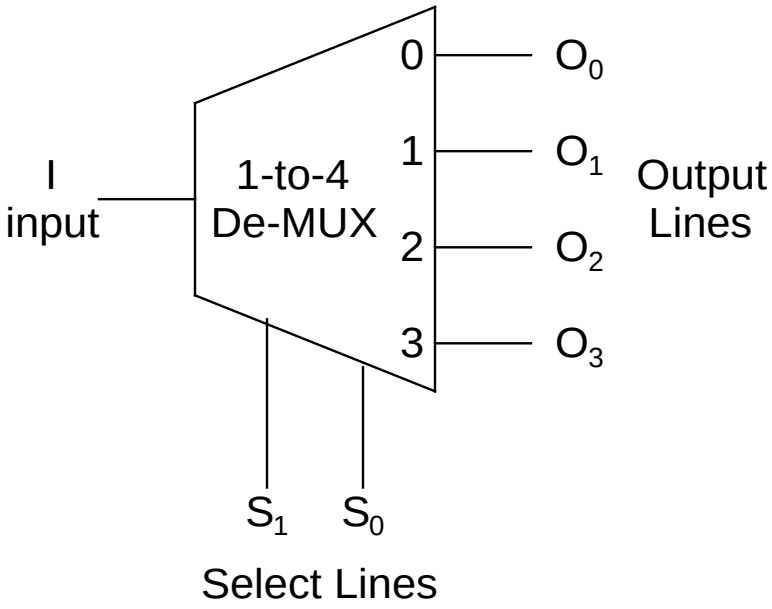


Higher-Order De-MUX:

* Construction of a **1-to-32 De-MUX** using **1-to-4 and 1-to-8 De-MUX**
* ***Trunk and Branches Architecture***
* **Control Signal**: EDCBA (E: MSB, A: LSB)
* ***1-to-4 De-MUX controls the 2 most significant bits, 1-to-8 De-MUX control the 3 least significant bits***

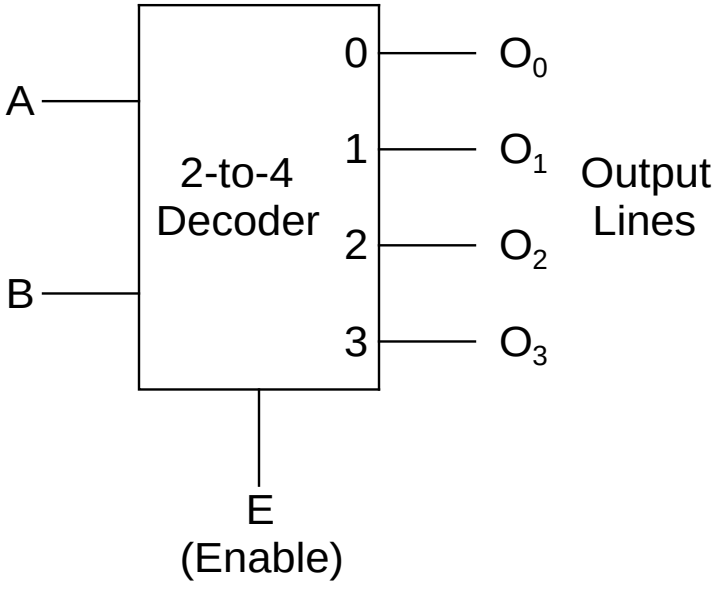


Decoder as De-MUX & Vice-Versa:



1-to-4 De-MUX as
2-to-4 Decoder

* **Use the i/p line (I) as the Enable (E) signal**
* ***Use the Select lines (S_1-S_0) as the i/p lines of the decoder***
* **When I (or E) = 0, all o/ps 0**
* **When I (or E) = 1:**
  - ***$S_1S_0 = 00, O_0 = 1$, all others 0***
  - ***$S_1S_0 = 01, O_1 = 1$, all others 0***
  - ***$S_1S_0 = 10, O_2 = 1$, all others 0***
  - ***$S_1S_0 = 11, O_3 = 1$, all others 0***



2-to-4 Decoder as
1-to-4 De-MUX

* **Use Enable (E) line as the i/p line (I)**
* ***Use the i/ps (A and B) as Select lines (S_1-S_0)***
* **When E (or I) = 0, all o/ps 0**
* **When E (or I) = 1:**
  - ***AB = 00, O_0 = 1, all others 0***
  - ***AB = 01, O_1 = 1, all others 0***
  - ***AB = 10, O_2 = 1, all others 0***
  - ***AB = 11, O_3 = 1, all others 0***