EE2000 Logic Circuit Design

Lecture 5 – Combinational Functional Blocks

electronicscoach.com

What will you learn?

5.1 What is an equality comparator and how to implement it using a modular design

5.2 Learn various arithmetic functional blocks

- Half adder, Full adder, Ripple carry adder
- Half and Full subtractors, Ripple borrow subtractor
- Carry-look-ahead adder

5.3 Learn various logical functional blocks

- Decoder
- Encoder
- Multiplexer
- Demultiplexer

5.1 Equality Comparator

- A circuit to compare two binary numbers to determine whether they are equal or not
- The inputs consist of two variables: A and B
- The output of the circuit is a variable E
- E is equal to 1 if A and B are equal
- E is equal to 0 if A and B are different

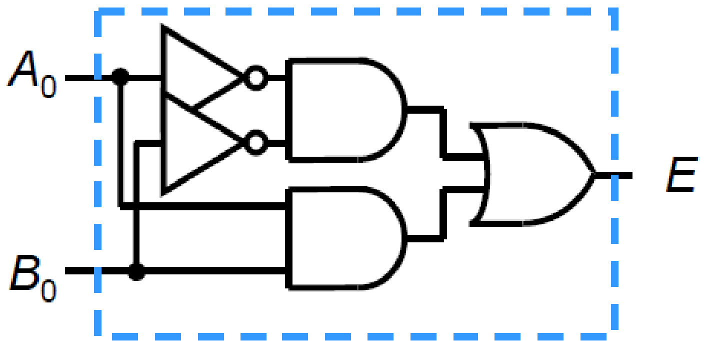
1-bit Equality Comparator

- Formulation:

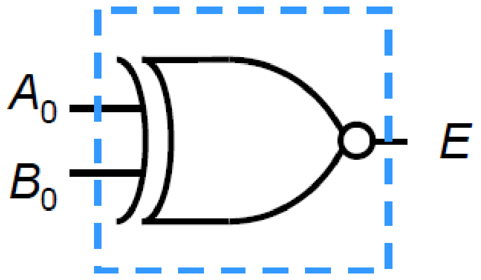
Inputs		Output
A_0	B_0	E
0	0	1
0	1	0
1	0	0
1	1	1

- Optimization:
  - $E(A_0, B_0) = \Sigma m(0, 3)$
  - $= A_0'B_0' + A_0B_0$
  - $= A_0 \otimes B_0$

- Final logic diagram:



or



2-bit Equality Comparator

Inputs				Output
A_1	A_0	B_1	B_0	E
0	0	0	0	1
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	1
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	1
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

$$E(A_1, A_0, B_1, B_0)$$
$$= \Sigma\, m(0, 5, 10, 15)$$
$$= A_1'A_0'B_1'B_0' +$$
$$A_1'A_0B_1'B_0 + A_1A_0'B_1B_0'$$
$$+ A_1A_0B_1B_0$$

4-bit Equality Comparator

- Formulation:
  - How many inputs?
  - How many outputs?
  - How many rows?

Problem:

Not easy to design

Difficult in simplification

K-map? QM ?

Solution:

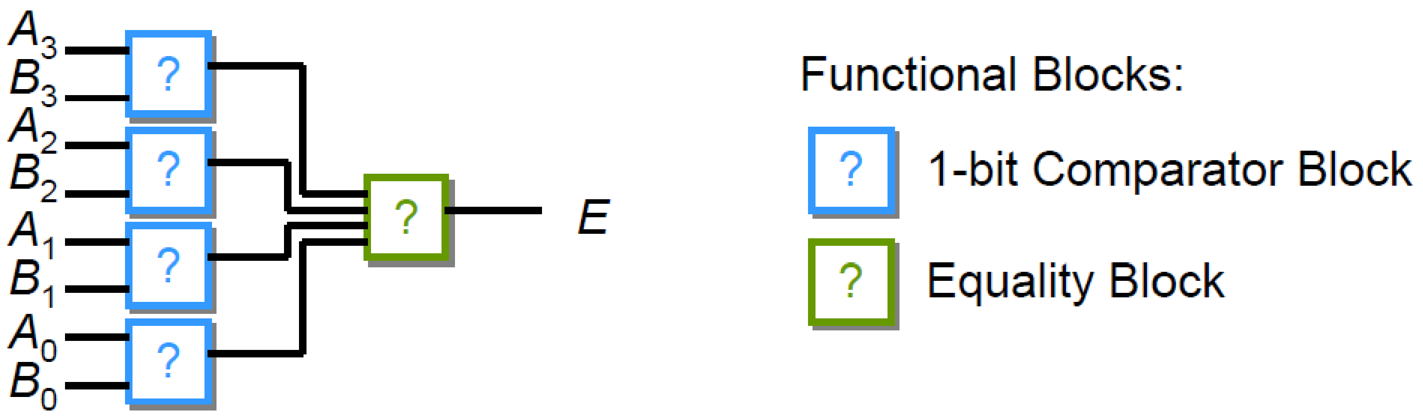
Modular design

Functional circuit blocks

Inputs								Output
A_3	A_2	A_1	A_0	B_3	B_2	B_1	B_0	E
0	0	0	0	0	0	0	0	1
0	0	0	0	0	0	0	1	0
0	0	0	0	0	0	1	0	0
0	0	0	0	0	0	1	1	0
0	0	0	0	0	1	0	0	0
0	0	0	0	0	1	0	1	0
0	0	0	0	0	1	1	0	0
0	0	0	0	0	1	1	1	0
0	0	0	0	1	0	0	0	0
0	0	0	0	1	0	0	1	0
0	0	0	0	1	0	1	0	0
0	0	0	0	1	0	1	1	0
0	0	0	0	1	1	0	0	0
1	1	1	1	1	1	1	1	1

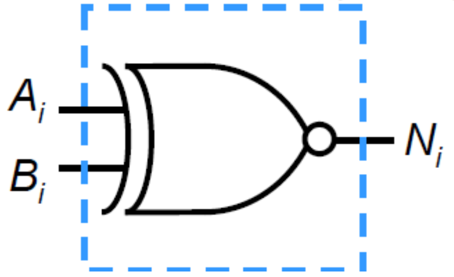
Modular Design

- Modular design
  - Decompose the problem into four 1-bit comparison
  - Compare bit by bit, then combine all results
- Logic diagram

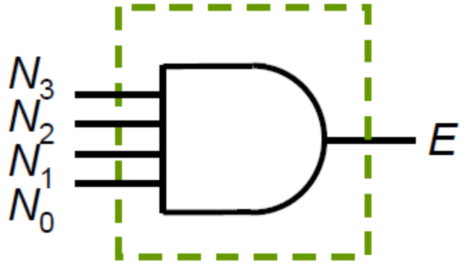


Modular Design

- 1-bit Comparator Block
  - The output is 1 if the inputs are the same
  - The output is 0 if the inputs are different
  - i.e. 1-bit equality comparator $N_i = A_i \otimes B_i$

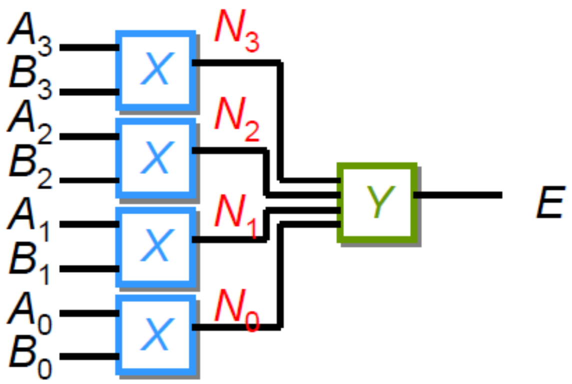


- Equality Block
  - The output E is 1 if all N_i values are 1
  - The output E is 0 if not all N_i values are 1
  - $E = N_3 \cdot N_2 \cdot N_1 \cdot N_0$



Modular Design

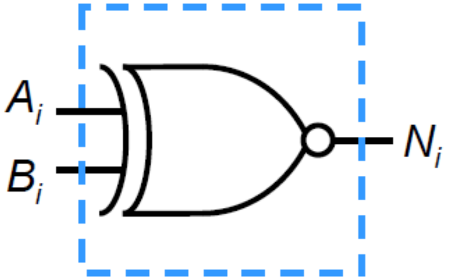
- Final logic diagram



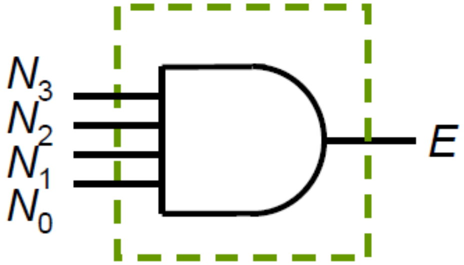
Functional Blocks:

X	1-bit Comparator Block
Y	Equality Block

Block *X*:



Block *Y*:



Summary

- Instead of designing a complex *n*-bit equality comparator circuit
- Design only a 1-bit comparator block and a simple equality block
- Re-use the 1-bit comparator block for *n* times
- Reusable small circuits are called **combinational functional blocks**

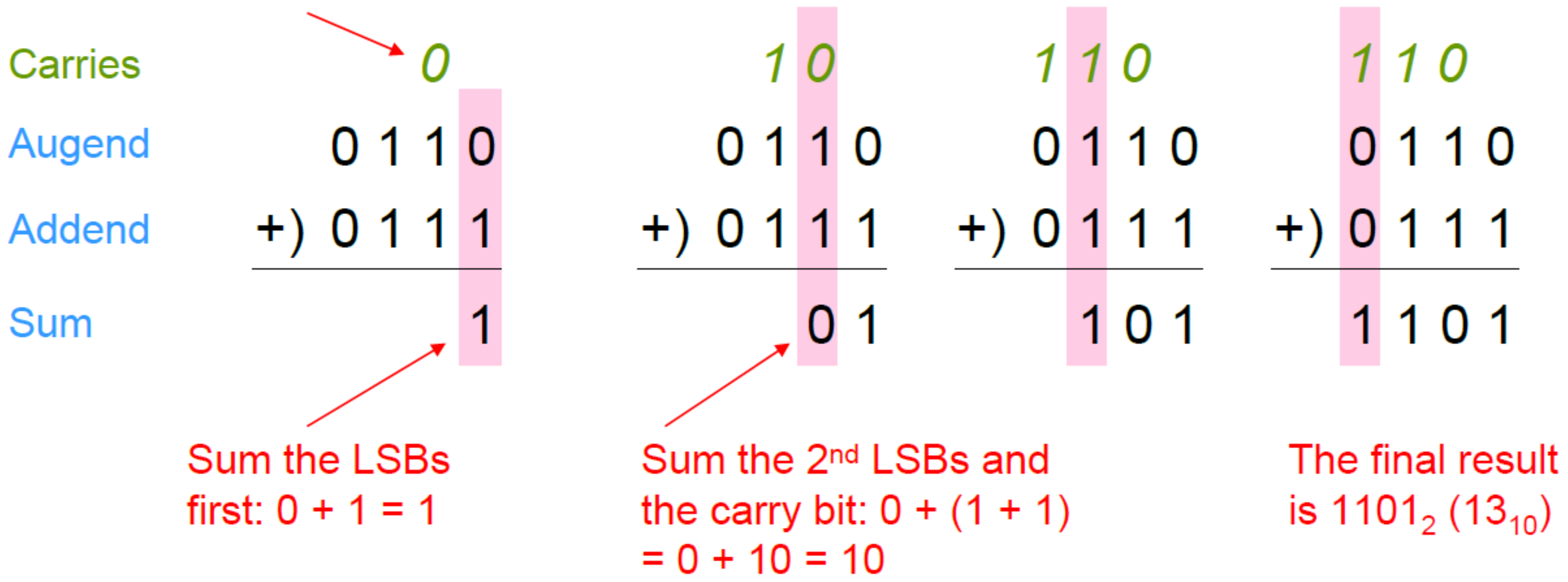
5.2 Arithmetic Functional Blocks

- Special class of functional blocks that perform arithmetic operations
- Operate on binary numbers (input) and produce binary numbers (output)
- Each bit position has the same sub-function
- Design a functional block for the sub-function and use **repeatedly** for each bit position
- Example arithmetic functional blocks

  - Adders, subtractors

Addition

- Compute the sum of $(0110)_2$ and $(0111)_2$
  - $(0110)_2 = (6)_{10}$
  - $(0111)_2 = (7)_{10}$


Carries
Augend
Addend
Sum
0
0 1 1 0
+) 0 1 1 1
1
1 0
0 1 1 0
+) 0 1 1 1
0 1
1 1 0
0 1 1 0
+) 0 1 1 1
1 0 1
1 1 0
0 1 1 0
+) 0 1 1 1
1 1 0 1
Sum the LSBs first: 0 + 1 = 1
Sum the 2nd LSBs and the carry bit: 0 + (1 + 1) = 0 + 10 = 10
The final result is 1101₂ (13₁₀)

Half Adder (1-bit Adder)

Operation: 1-bit binary addition (the addition of two numbers, *x* and *y*)

Inputs: *x* and *y*   Outputs: *s* (sum) and *c* (carry-out)

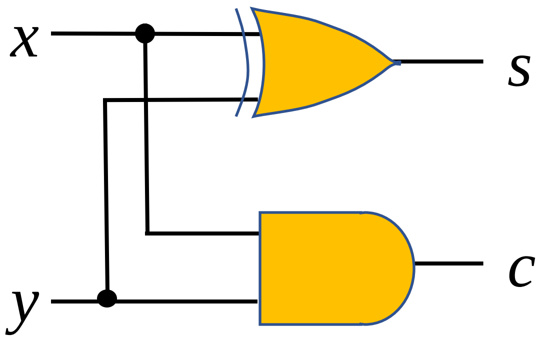
$$\begin{array}{rccc} & & 0 & x \\ (+) & & 0 & y \\ \hline & 0 & 0 & \\ & c & s & \end{array} \qquad \begin{array}{rccc} & & 0 & x \\ (+) & & 1 & y \\ \hline & 0 & 1 & \\ & c & s & \end{array}$$

$$\begin{array}{rccc} & & 1 & x \\ (+) & & 0 & y \\ \hline & 0 & 1 & \\ & c & s & \end{array} \qquad \begin{array}{rccc} & & 1 & x \\ (+) & & 1 & y \\ \hline & 1 & 0 & \\ & c & s & \end{array}$$

Inputs		Outputs	
x	y	c	s
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

$$s = x \oplus y$$

$$c = x \cdot y = xy$$



Full Adder

Operation: 1-bit binary addition (the addition of two numbers, *x* and *y;* and a carry-in bit, c_{in}

Inputs: *x*, *y* and c_{in} Outputs: *s* (sum) and c_o (carry-out)

$$\begin{array}{lccl} & & 0 & x \\ & & 0 & y \\ (+) & & 0 & c_{in} \\ \hline & 0 & 0 & \\ & c_o & s & \end{array} \quad \begin{array}{lccl} & & 0 & x \\ & & 1 & y \\ (+) & & 0 & c_{in} \\ \hline & 0 & 1 & \\ & c_o & s & \end{array} \quad \begin{array}{lccl} & & 1 & x \\ & & 0 & y \\ (+) & & 0 & c_{in} \\ \hline & 0 & 1 & \\ & c_o & s & \end{array} \quad \begin{array}{lccl} & & 1 & x \\ & & 1 & y \\ (+) & & 0 & c_{in} \\ \hline & 1 & 0 & \\ & c_o & s & \end{array}$$

$$\begin{array}{lccl} & & 0 & x \\ & & 0 & y \\ (+) & & 1 & c_{in} \\ \hline & 0 & 1 & \\ & c_o & s & \end{array} \quad \begin{array}{lccl} & & 0 & x \\ & & 1 & y \\ (+) & & 1 & c_{in} \\ \hline & 1 & 0 & \\ & c_o & s & \end{array} \quad \begin{array}{lccl} & & 1 & x \\ & & 0 & y \\ (+) & & 1 & c_{in} \\ \hline & 1 & 0 & \\ & c_o & s & \end{array} \quad \begin{array}{lccl} & & 1 & x \\ & & 1 & y \\ (+) & & 1 & c_{in} \\ \hline & 1 & 1 & \\ & c_o & s & \end{array}$$

Inputs			Outputs	
x	y	c_{in}	c_o	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Example

$$\begin{array}{llll} & & 0 & x\\ & & 0 & y\\ (+) & & 0 & c_{in}\\ \hline & 0 & 0 & \\ & c_o & s & \end{array}
\quad
\begin{array}{llll} & & 0 & x\\ & & 1 & y\\ (+) & & 0 & c_{in}\\ \hline & 0 & 1 & \\ & c_o & s & \end{array}
\quad
\begin{array}{llll} & & 1 & x\\ & & 0 & y\\ (+) & & 0 & c_{in}\\ \hline & 0 & 1 & \\ & c_o & s & \end{array}
\quad
\begin{array}{llll} & & 1 & x\\ & & 1 & y\\ (+) & & 0 & c_{in}\\ \hline & 1 & 0 & \\ & c_o & s & \end{array}$$

$$\begin{array}{llll} & & 0 & x\\ & & 0 & y\\ (+) & & 1 & c_{in}\\ \hline & 0 & 1 & \\ & c_o & s & \end{array}
\quad
\begin{array}{llll} & & 0 & x\\ & & 1 & y\\ (+) & & 1 & c_{in}\\ \hline & 1 & 0 & \\ & c_o & s & \end{array}
\quad
\begin{array}{llll} & & 1 & x\\ & & 0 & y\\ (+) & & 1 & c_{in}\\ \hline & 1 & 0 & \\ & c_o & s & \end{array}
\quad
\begin{array}{llll} & & 1 & x\\ & & 1 & y\\ (+) & & 1 & c_{in}\\ \hline & 1 & 1 & \\ & c_o & s & \end{array}$$

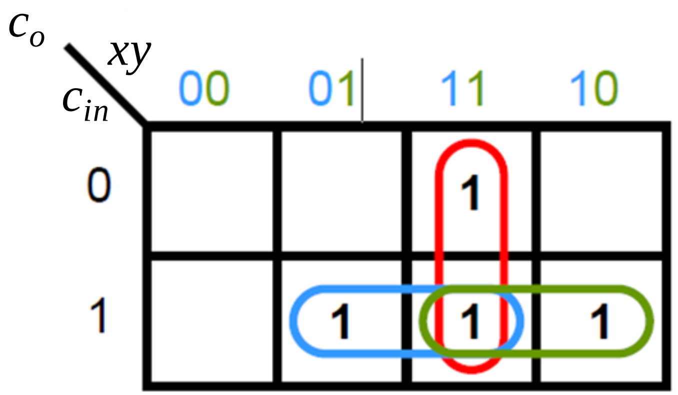
Inputs			Outputs	
x	y	c_{in}	c_o	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

a) Work out the algebraic functions of *s* and c_o using K-map

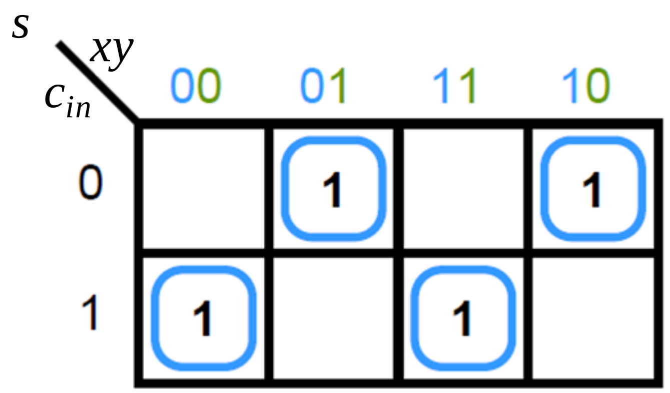
b) Draw the logic circuit diagram of full adder

$c \backslash ab$	00	01	11	10
0	m_0	m_2	m_6	m_4
1	m_1	m_3	m_7	m_5

Example



$$c_o = xy + xc_{in} + yc_{in} = xy + c_{in}(x + y)$$
$$= xy + c_{in}(xy' + x'y + xy)$$
$$= xy(1 + c_{in}) + c_{in}(xy' + x'y)$$
$$= xy + c_{in}(x \oplus y)$$

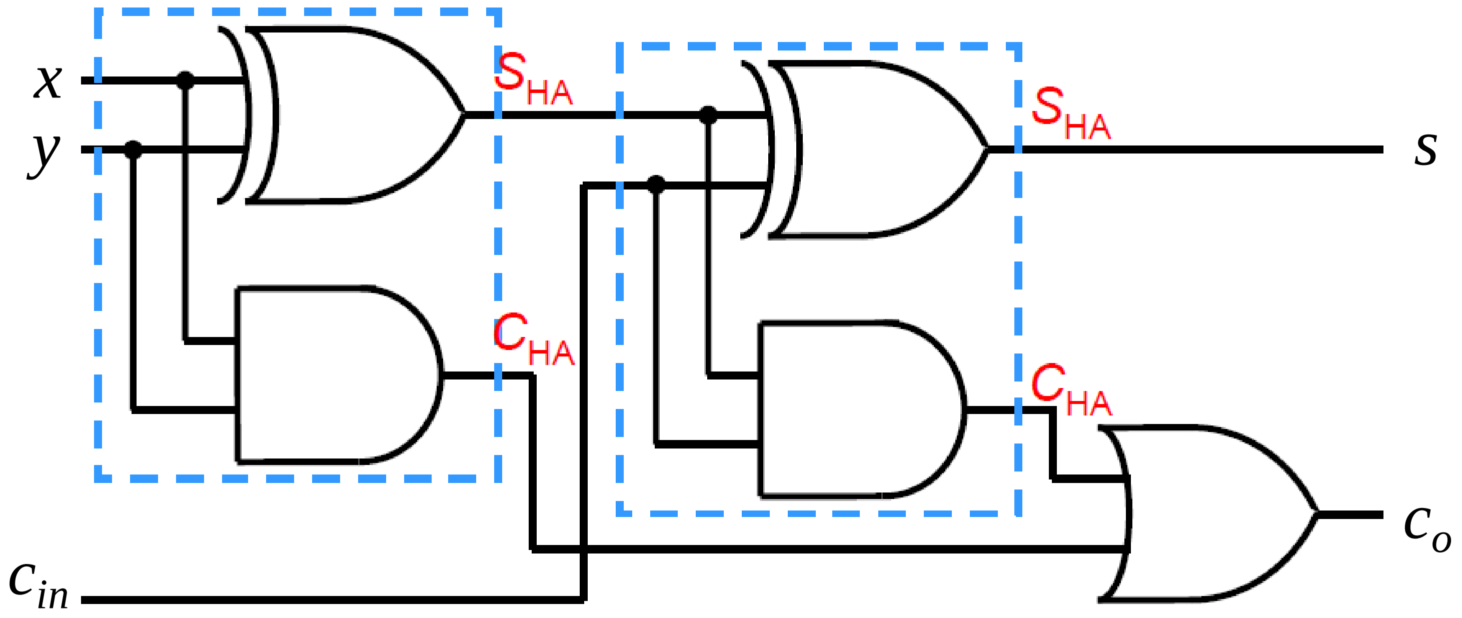


$$s = x'y'c_{in} + xyc_{in} + x'yc'_{in} + xy'c'_{in}$$
$$= c_{in}(x'y' + xy) + c'_{in}(x'y + xy')$$
$$= c_{in}(x \oplus y)' + c'_{in}(x \oplus y)$$
$$= c_{in} \oplus (x \oplus y)$$

Example

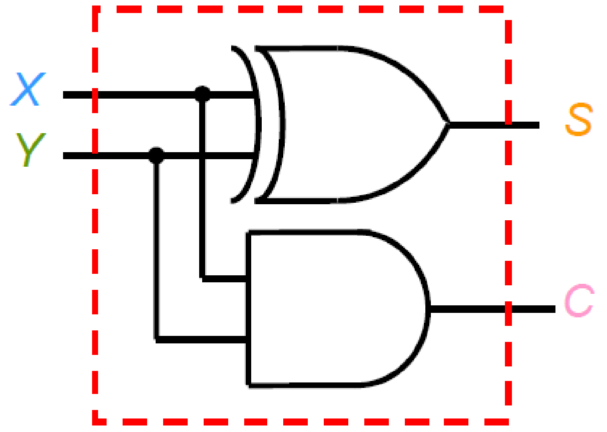
$c \backslash ab$	00	01	11	10
0	m_0	m_2	m_6	m_4
1	m_1	m_3	m_7	m_5

$c_o = xy + c_{in}(x \oplus y)$ $\qquad$ $s = c_{in} \oplus (x \oplus y)$

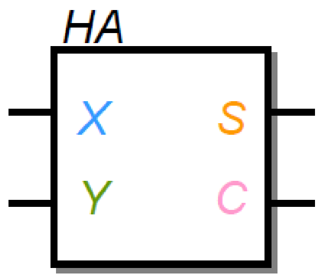


Half Adder and Full Adder

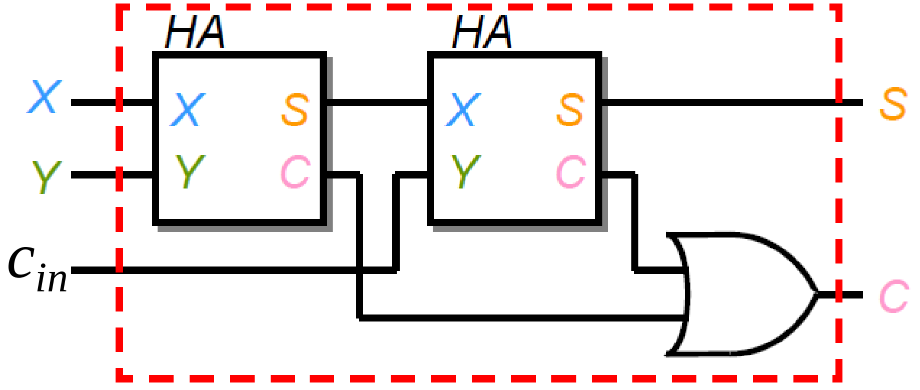
- Logic circuit diagram



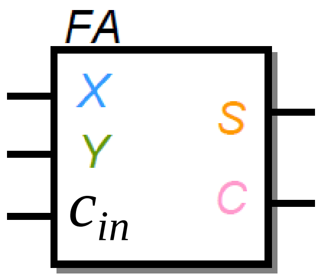
- Symbol



- Logic circuit diagram

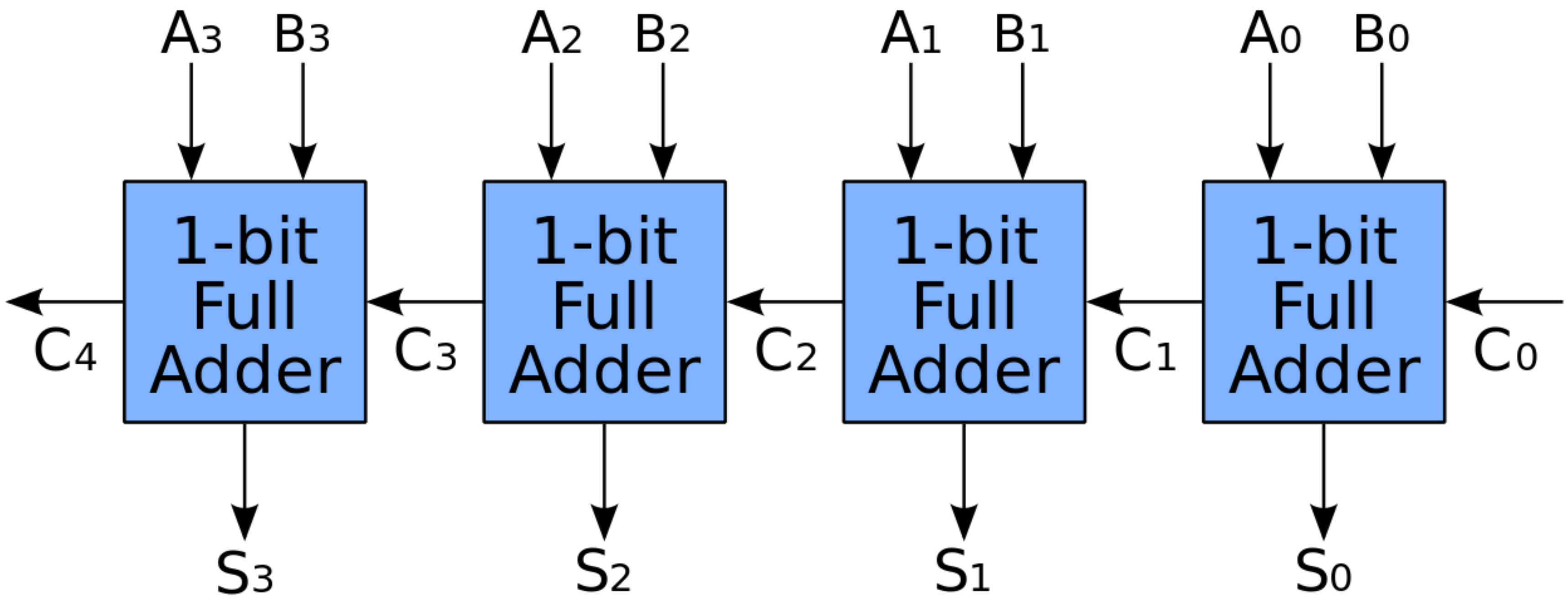


- Symbol

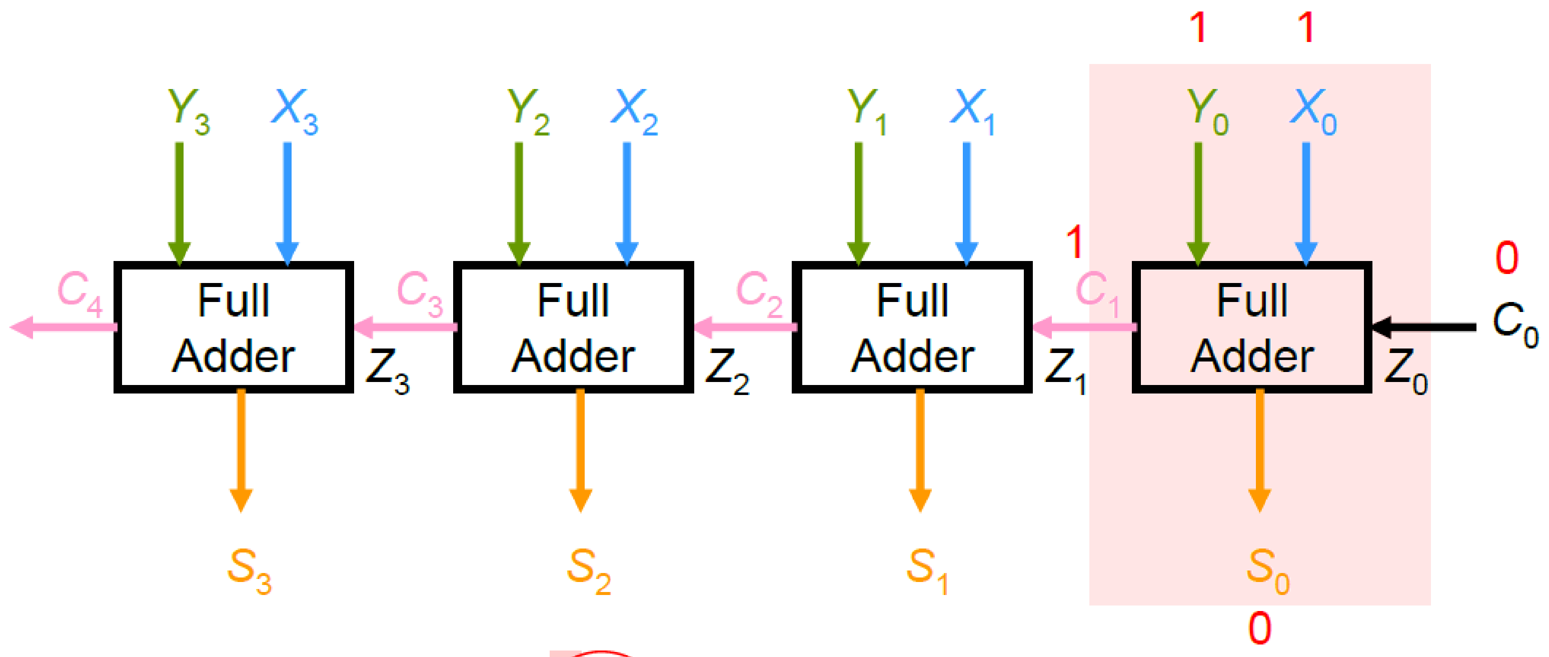


Ripple Carry Adder

- Connect ***n*** 1-bit adders to build an ***n***-bit adder



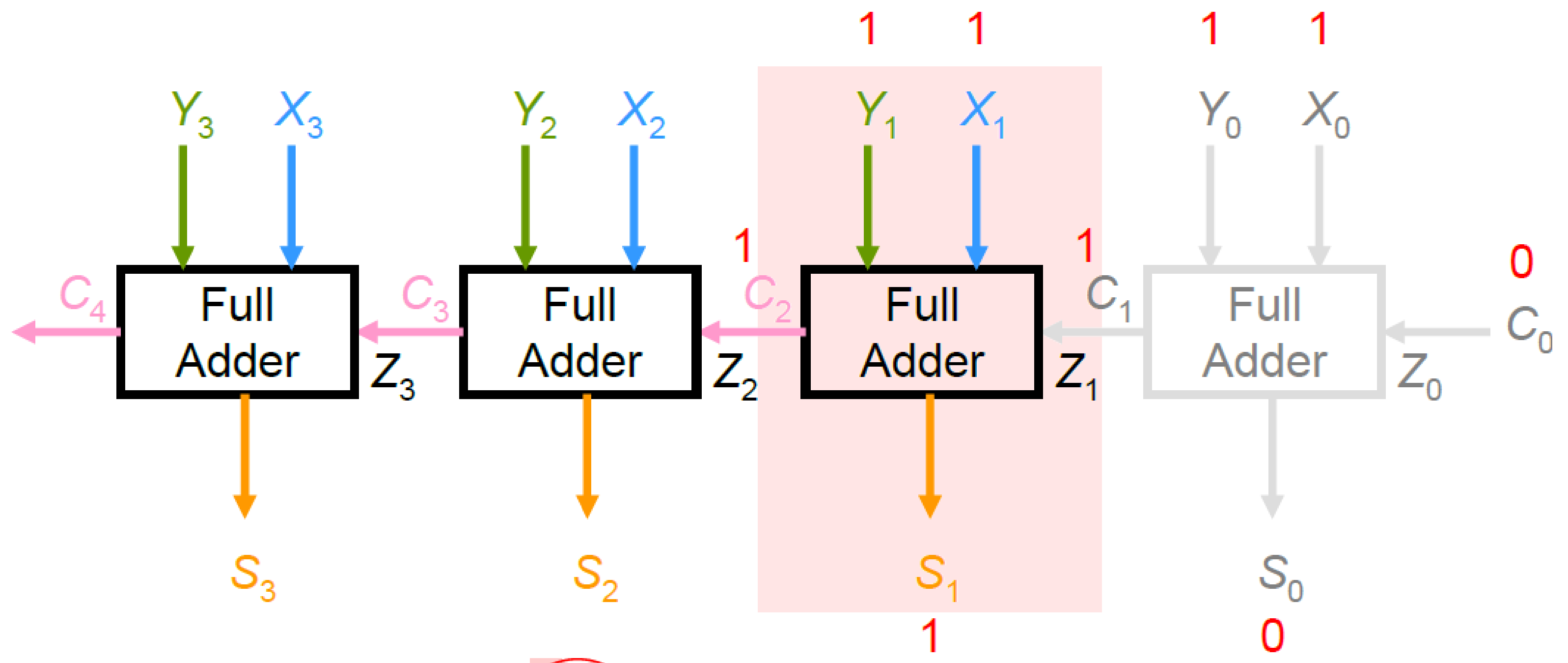
Example (0011 + 1011)



Carry-in (Z_i)	10
Augend (X_i)	1011
Addend (Y_i)	0011
Sum (S_i)	0
Carry-out (C_i)	1

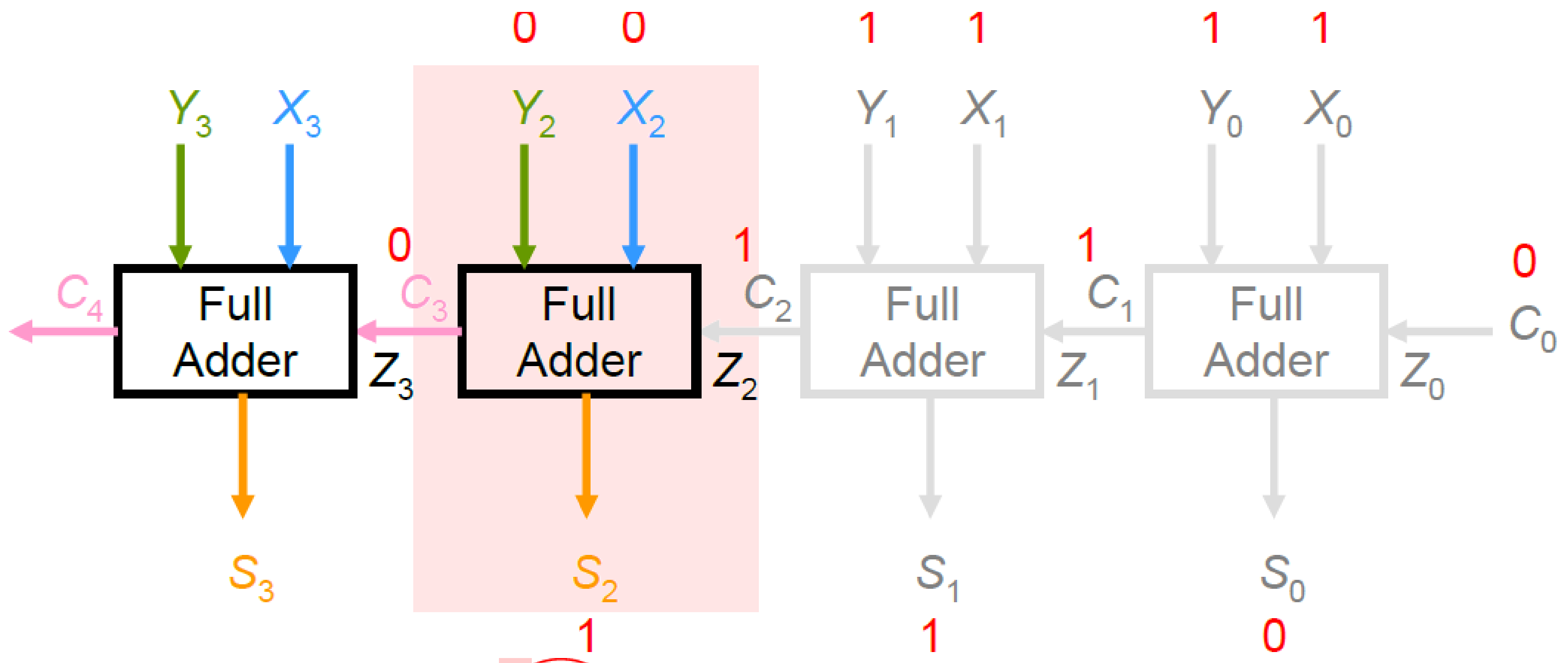
Carry out connect to the carry input of next full adder

Example



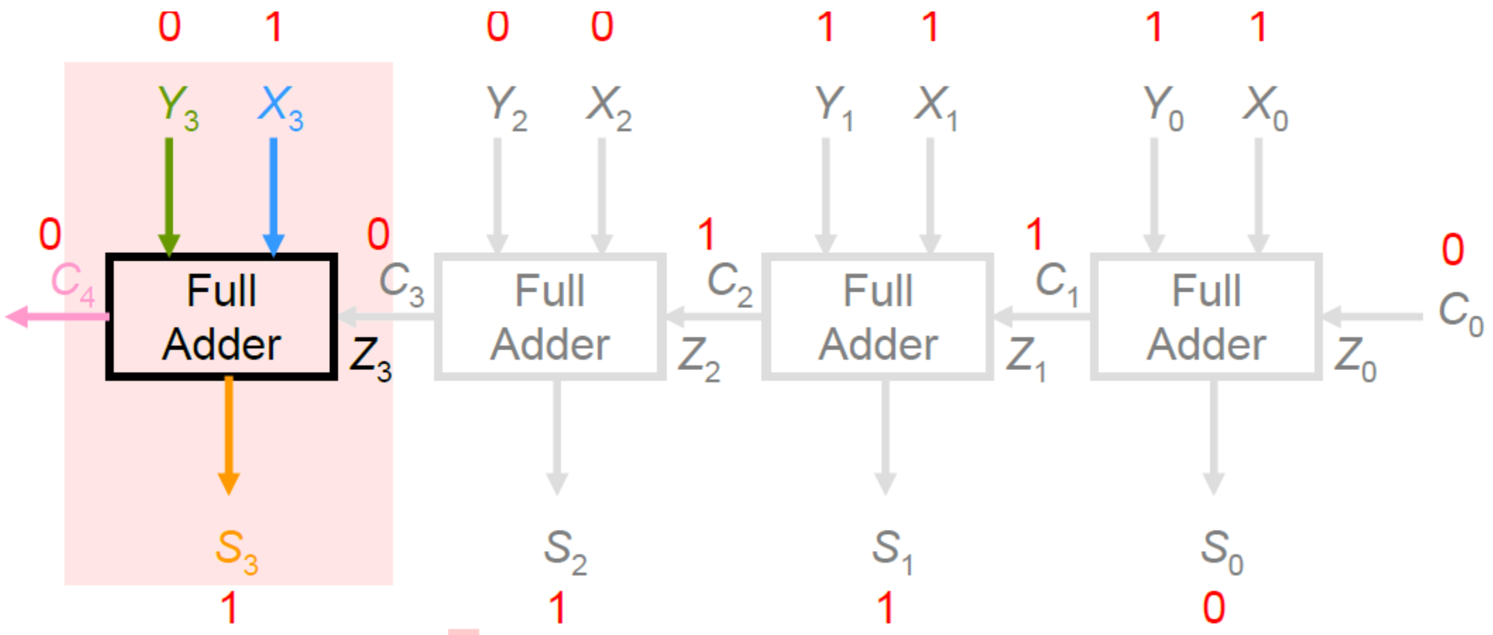
Carry-in (Z_i)	110
Augend (X_i)	1011
Addend (Y_i)	0011
Sum (S_i)	10
Carry-out (C_i)	11

Example

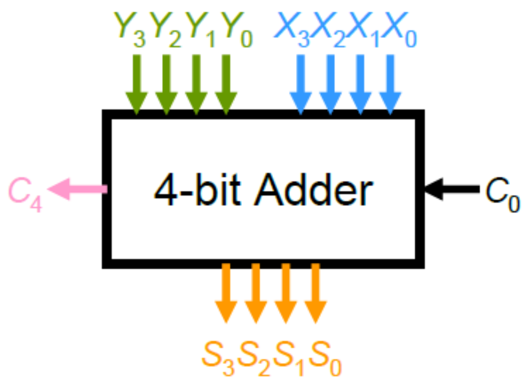


Carry-in (Z_i)	0110
Augend (X_i)	1011
Addend (Y_i)	0011
Sum (S_i)	110
Carry-out (C_i)	011

Example



Carry-in (Z_i)	0110
Augend (X_i)	1011
Addend (Y_i)	0011
Sum (S_i)	1110
Carry-out (C_i)	0011



HA vs FA *vs* RCA

Augend (X)		1
Addend (Y)	+)	0
Sum (S)		1

HA: performs simple two single-bit addition

Carry-in (Z_i)		*0*
Augend (X_i)		1
Addend (Y_i)	+)	1
Sum (S_i)		0

FA: performs simple three single-bit addition

Carry-in (Z_i)		*0*	*1*	*1*	*0*
Augend (X_i)		1	0	1	1
Addend (Y_i)	+)	0	0	1	1
Sum (S_i)		1	1	1	0

RCA: performs real two *n*-bit addition

Subtractors

Minuend (X)	1
Subtrahend (Y) -)	0
Difference (D)	1

Half Subtractor: perform simple two single-bit subtraction

Borrow-in (Z_i)	$\downarrow$ *0*
Minuend (X_i)	1
Subtrahend (Y_i) -)	1
Difference (D_i)	0

Full Subtractor: perform simple three single-bit subtraction

Borrow-in (Z_i)	?	?	?	?
Minuend (X_i)	1	0	1	1
Subtrahend (Y_i) -)	0	0	1	1
Difference (D_i)	?	?	?	?

Ripple Borrow Subtractor: perform real two ***n***-bit subtraction

Half Subtractor

Operation: 1-bit binary subtraction (*x* - *y*)

Inputs: *x* and *y*

Outputs: *d* (difference) and *b* (Borrow-out)

$$\begin{array}{cccc} b & 0 & 0 & x \\ (-) & & 0 & y \\ \hline & & 0 & \\ & & d & \end{array}$$

$$\begin{array}{cccc} b & 1 & 0 & x \\ (-) & & 1 & y \\ \hline & & 1 & \\ & & d & \end{array}$$

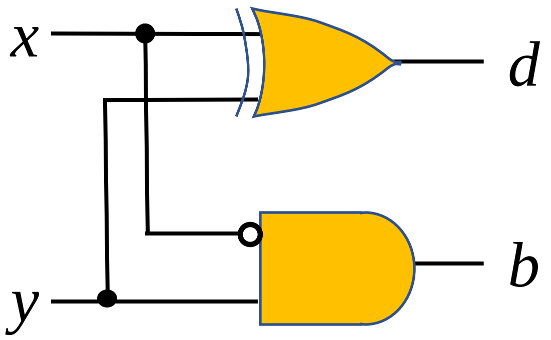
$$\begin{array}{cccc} b & 0 & 1 & x \\ (-) & & 0 & y \\ \hline & & 1 & \\ & & d & \end{array}$$

$$\begin{array}{cccc} b & 0 & 1 & x \\ (-) & & 1 & y \\ \hline & & 0 & \\ & & d & \end{array}$$

Inputs		Outputs	
x	y	b	d
0	0	0	0
0	1	1	1
1	0	0	1
1	1	0	0

$$d = x \oplus y$$

$$b = x' \cdot y = x'y$$



Full Subtractor

Operation: 1-bit binary subtraction ($x - y - b_{in}$)

Inputs: x, y and b_{in} Outputs: d and b_o

$$\begin{array}{lccl} b_o & 0 & 0 & x \\ (-) & & 0 & y \\ (-) & & 0 & b_{in} \\ \hline & & 0 & \\ & & d & \end{array}$$

$$\begin{array}{lccl} b_o & 1 & 0 & x \\ (-) & & 1 & y \\ (-) & & 0 & b_{in} \\ \hline & & 1 & \\ & & d & \end{array}$$

$$\begin{array}{lccl} b_o & 0 & 1 & x \\ (-) & & 0 & y \\ (-) & & 0 & b_{in} \\ \hline & & 1 & \\ & & d & \end{array}$$

$$\begin{array}{lccl} b_o & 0 & 1 & x \\ (-) & & 1 & y \\ (-) & & 0 & b_{in} \\ \hline & & 0 & \\ & & d & \end{array}$$

$$\begin{array}{lccl} b_o & 1 & 0 & x \\ (-) & & 0 & y \\ (-) & & 1 & b_{in} \\ \hline & & 1 & \\ & & d & \end{array}$$

$$\begin{array}{lccl} b_o & 1 & 0 & x \\ (-) & & 1 & y \\ (-) & & 1 & b_{in} \\ \hline & & 0 & \\ & & d & \end{array}$$

$$\begin{array}{lccl} b_o & 0 & 1 & x \\ (-) & & 0 & y \\ (-) & & 1 & b_{in} \\ \hline & & 0 & \\ & & d & \end{array}$$

$$\begin{array}{lccl} b_o & 1 & 1 & x \\ (-) & & 1 & y \\ (-) & & 1 & b_{in} \\ \hline & & 1 & \\ & & d & \end{array}$$

Inputs			Outputs	
x	y	b_{in}	b_o	d
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	1	0
1	0	0	0	1
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

Exercise

$$\begin{array}{rrrl} b_o & 0 & 0 & x \\ (-) & & 0 & y \\ (-) & & 0 & b_{in} \\ \hline & & 0 & \\ & & d & \end{array} \quad \begin{array}{rrrl} b_o & 1 & 0 & x \\ (-) & & 1 & y \\ (-) & & 0 & b_{in} \\ \hline & & 1 & \\ & & d & \end{array} \quad \begin{array}{rrrl} b_o & 0 & 1 & x \\ (-) & & 0 & y \\ (-) & & 0 & b_{in} \\ \hline & & 1 & \\ & & d & \end{array} \quad \begin{array}{rrrl} b_o & 0 & 1 & x \\ (-) & & 1 & y \\ (-) & & 0 & b_{in} \\ \hline & & 0 & \\ & & d & \end{array}$$

$$\begin{array}{rrrl} b_o & 1 & 0 & x \\ (-) & & 0 & y \\ (-) & & 1 & b_{in} \\ \hline & & 1 & \\ & & d & \end{array} \quad \begin{array}{rrrl} b_o & 1 & 0 & x \\ (-) & & 1 & y \\ (-) & & 1 & b_{in} \\ \hline & & 0 & \\ & & d & \end{array} \quad \begin{array}{rrrl} b_o & 0 & 1 & x \\ (-) & & 0 & y \\ (-) & & 1 & b_{in} \\ \hline & & 0 & \\ & & d & \end{array} \quad \begin{array}{rrrl} b_o & 1 & 1 & x \\ (-) & & 1 & y \\ (-) & & 1 & b_{in} \\ \hline & & 1 & \\ & & d & \end{array}$$

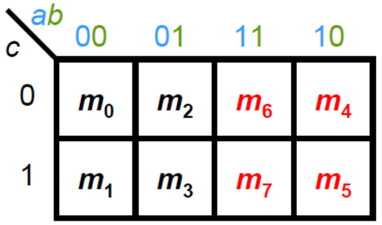
Inputs			Outputs	
x	y	b_{in}	b_o	d
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	1	0
1	0	0	0	1
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

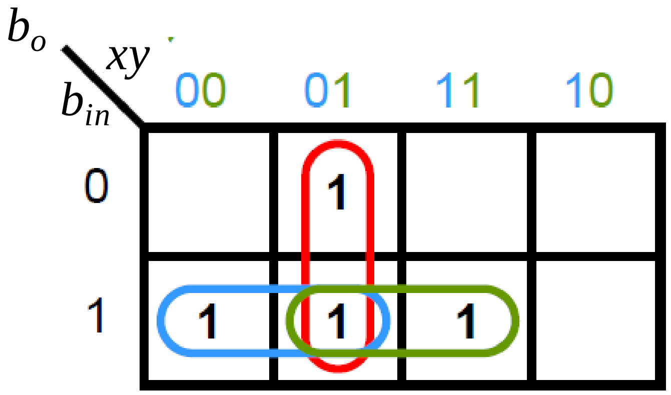
a) Work out the algebraic functions of *d* and b_o using K-map

b) Draw the logic circuit diagram of full subtractor

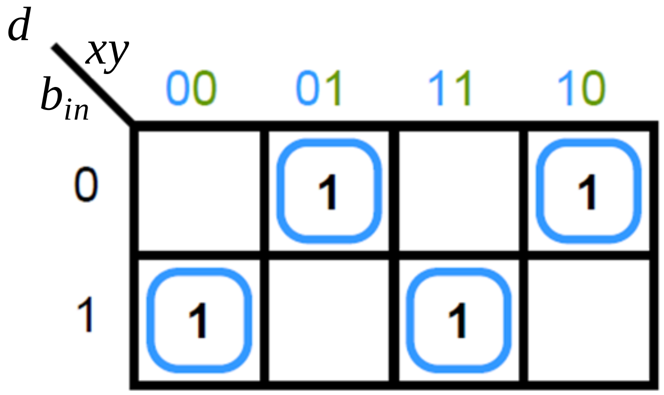
$c \backslash ab$	00	01	11	10
0	m_0	m_2	m_6	m_4
1	m_1	m_3	m_7	m_5

Exercise





$$b_o = x'y + x'b_{in} + yb_{in}$$



$$d = x'y'b_{in} + xyb_{in} + x'yb'_{in} + xy'b'_{in}$$

Exercise

$$\begin{array}{c|cccc} c \backslash ab & 00 & 01 & 11 & 10 \\ \hline 0 & m_0 & m_2 & m_6 & m_4 \\ 1 & m_1 & m_3 & m_7 & m_5 \end{array}$$

x

y

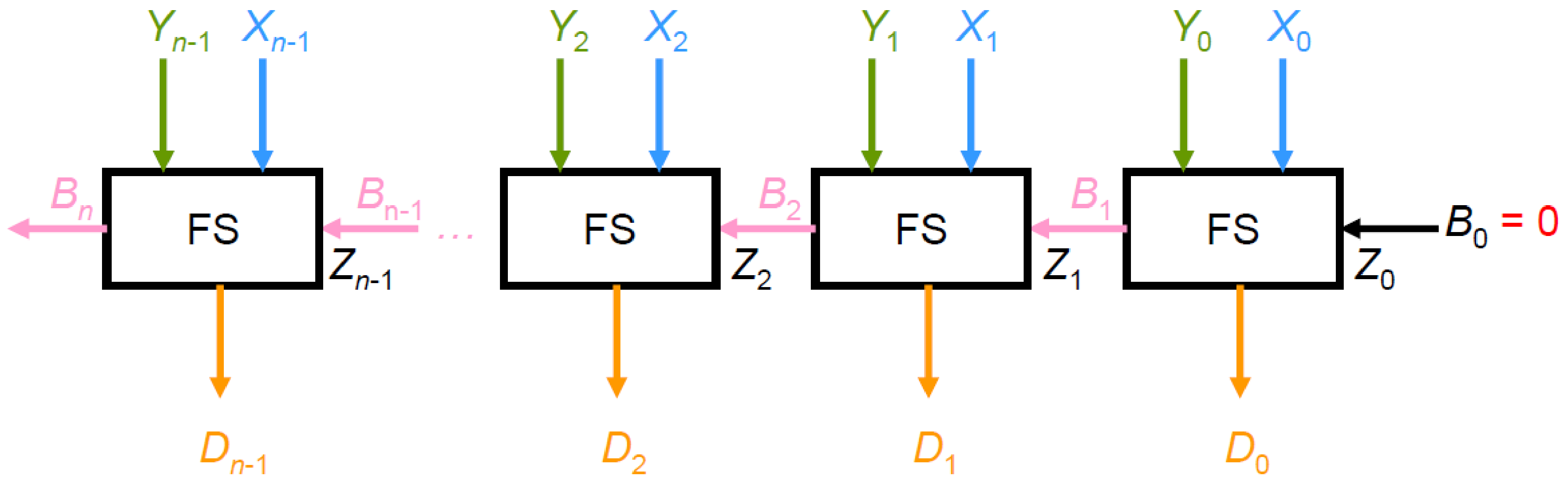
d

b_o

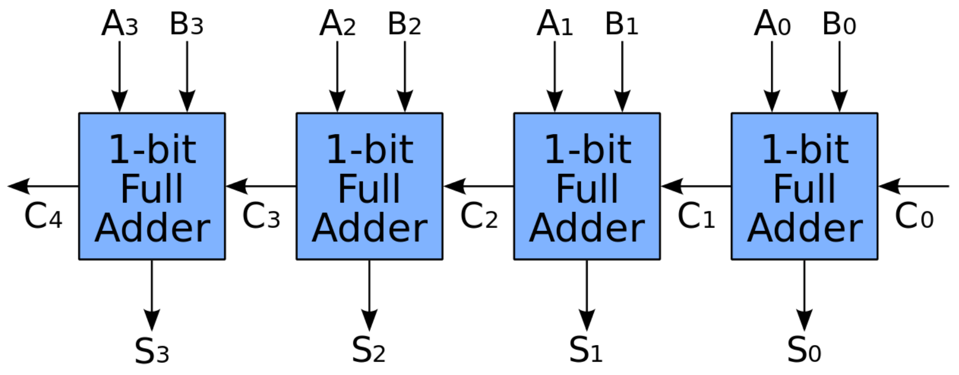
b_{in}

Ripple Borrow Subtractor

- Connect ***n*** 1-bit subtractors to build an ***n***-bit subtractor



Propagation Delay in RCA



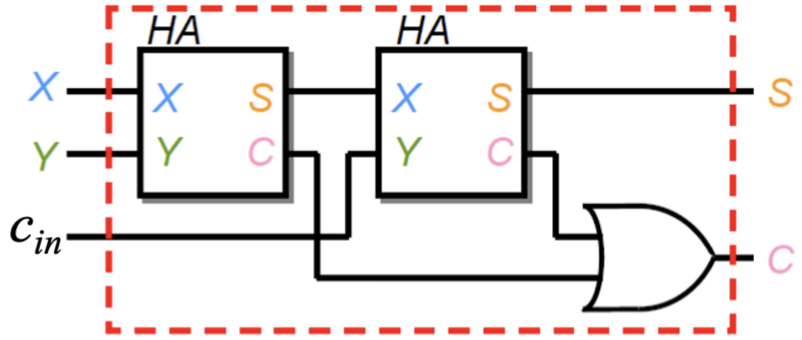
$$c_{out} = xy + c_{in}(x \oplus y)$$

$$s = c_{in} \oplus (x \oplus y)$$

- The carry output of each full-adder stage is connected to the carry input of the next stage
- Assume that the delay for generating the carry output is Δ*τ*, for ***n***-bit adder, the total delay is ***n*** Δ*τ*
- Serious delay problem if ***n*** is a large number
- Solution: Calculate the carry bits beforehand, then construct the carry-look-ahead adder

Carry Bits Calculation

$$c_{out} = xy + c_{in}(x \oplus y)$$



- G_i is defined as the generate bit

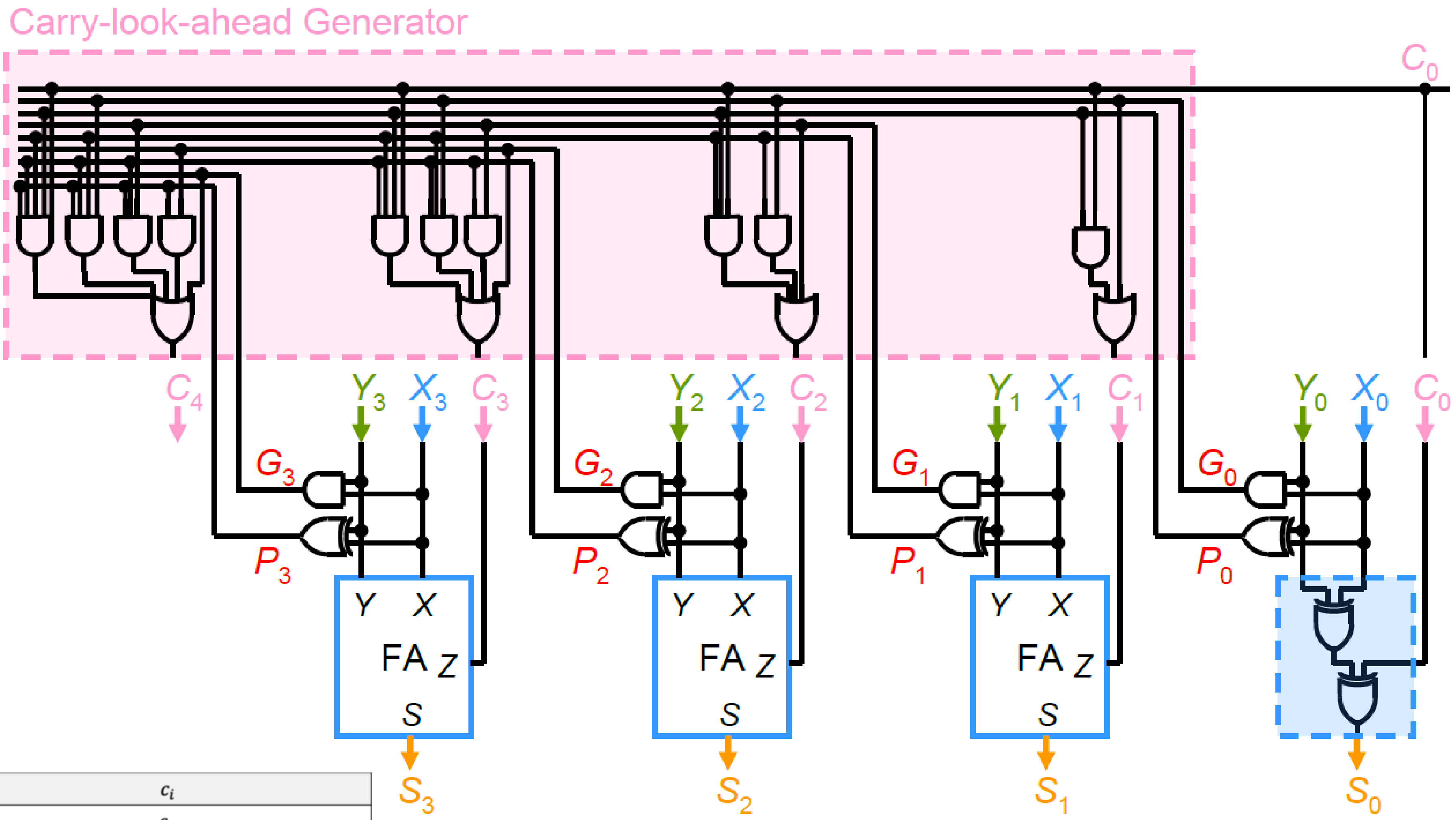
$$G_i = x_i y_i$$

- P_i is defined as the propagate bit $\quad P_i = x_i \oplus y_i$

$$c_i = x_{i-1}y_{i-1} + c_{i-1}(x_{i-1} \oplus y_{i-1}) = G_{i-1} + c_{i-1}(P_{i-1})$$

i	G_i	P_i	c_i
0	$G_0 = x_0 y_0$	$P_0 = (x_0 \oplus y_0)$	c_0
1	$G_1 = x_1 y_1$	$P_1 = (x_1 \oplus y_1)$	$c_1 = G_0 + c_0 P_0$
2	$G_2 = x_2 y_2$	$P_2 = (x_2 \oplus y_2)$	$c_2 = G_1 + c_1 P_1 = G_1 + P_1(G_0 + c_0 P_0)$ $= G_1 + P_1 G_0 + c_0 P_0 P_1$
3	$G_3 = x_3 y_3$	$P_3 = (x_3 \oplus y_3)$	$c_3 = G_2 + c_2 P_2 = G_2 + P_2 G_1 + P_2 P_1 G_0 + c_0 P_0 P_1 P_2$

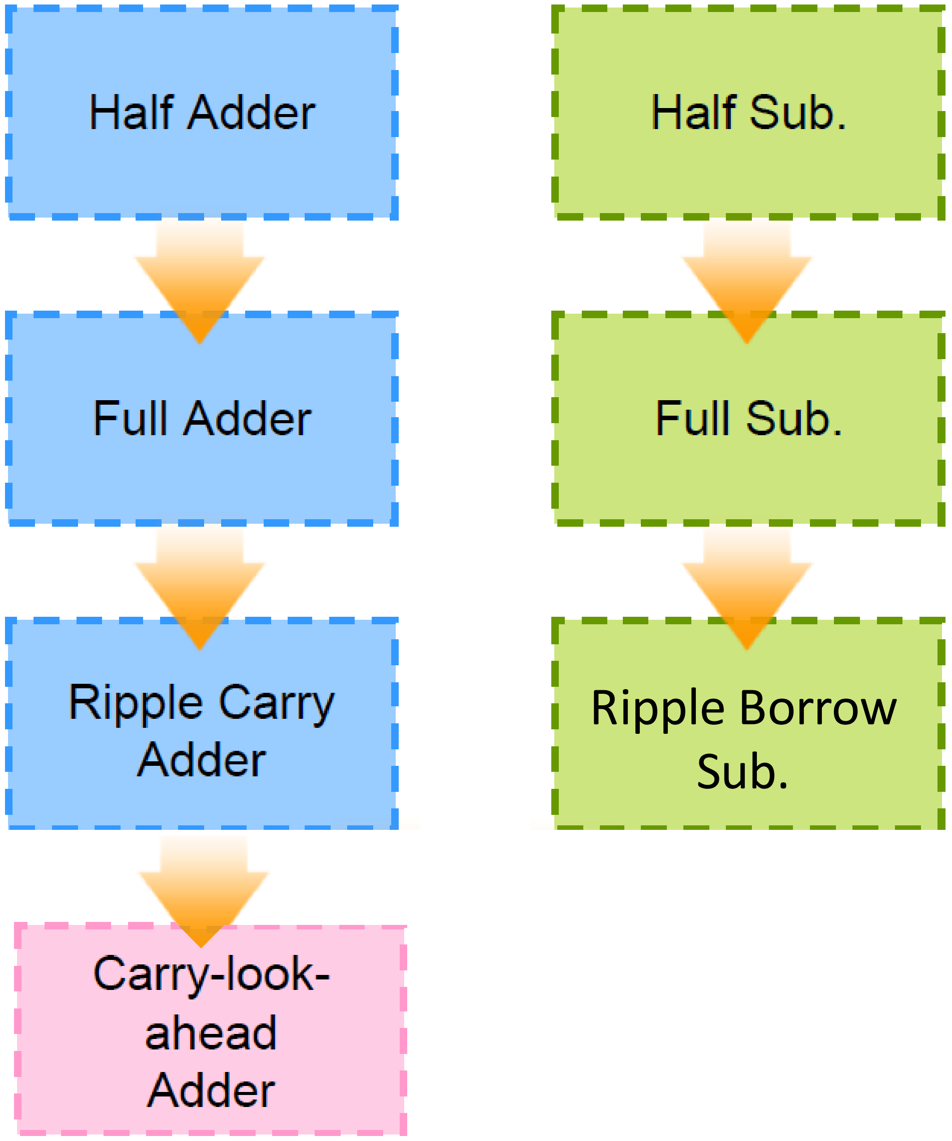
4-Bit Carry-Look-Ahead Adder



c_i
c_0
$G_0+c_0P_0$
$G_1+c_1P_1 = G_1+P_1(G_0+c_0P_0) = G_1+P_1G_0 + c_0P_0P_1$
$G_2+c_2P_2 = G_2+P_2G_1+P_2P_1G_0 + c_0P_0P_1P_2$

$$s = c_{in} \oplus (x \oplus y)$$

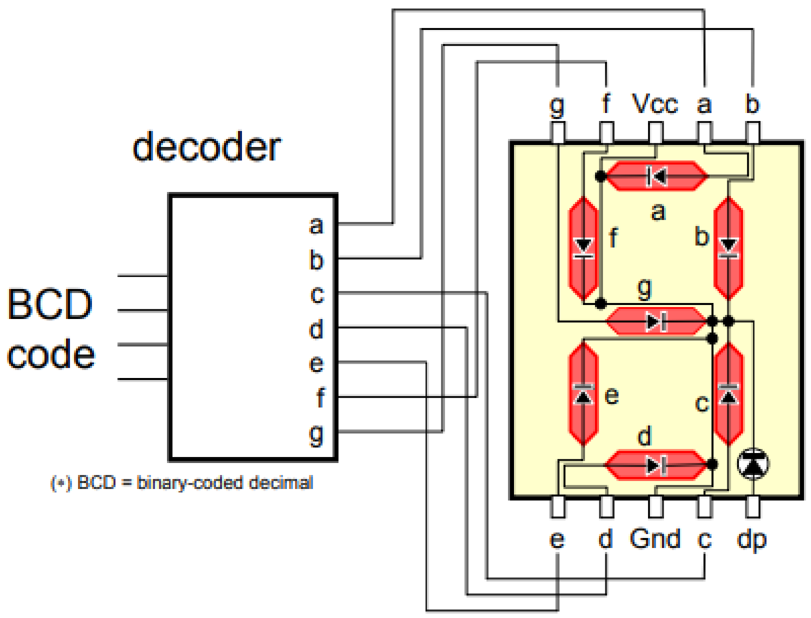
Summary


Half Adder
Full Adder
Ripple Carry Adder
Carry-look-ahead Adder
Half Sub.
Full Sub.
Ripple Borrow Sub.

5.3 Logical Functional Blocks

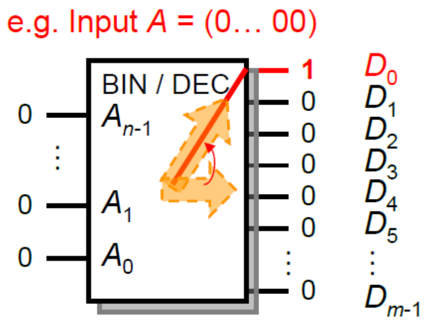
Decoder

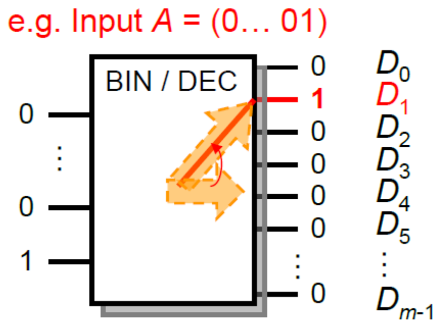
- A decoder is a combinational circuit that converts coded inputs into coded outputs.
- Each input produces a different output (1-to-1 mapping)



Binary Decoder

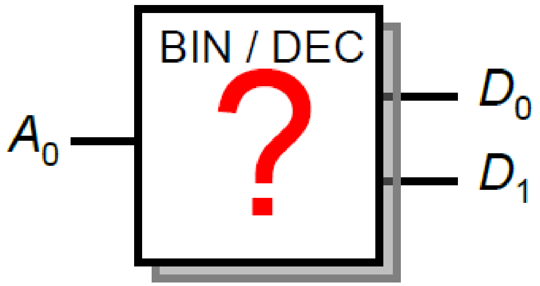
- Accept an ***n***-bit binary input code and activate only one of the 2^n outputs
- Outputs $\boldsymbol{m} \le 2^n$, but usually $\boldsymbol{m} = 2^n$
- A very important functional blocks.
  - Select different banks of memory
  - Select different devices
  - Enable different functional units
  - …





1-to-2 Decoder

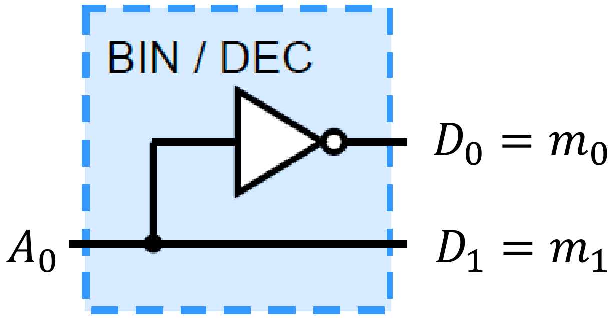
- Input: 1-bit (A_0)
- Output: 2-bit $(D_0 \,\&\, D_1)$



Input	Output	
A_0	D_1	D_0
0	0	1
1	1	0

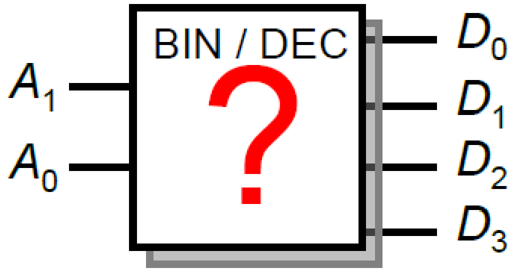
$$D_0 = A_0' = m_0$$

$$D_1 = A_0 = m_1$$

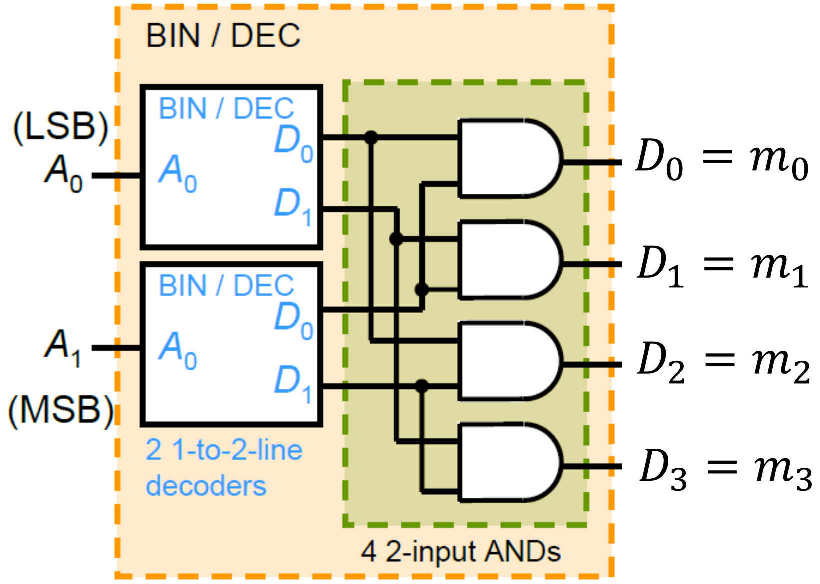


2-to-4 Decoder

- Input: 2-bit $(A_0\ \&\ A_1)$
- Output: 4-bit (D_0, D_1, D_2, D_3)



Input		Output			
A_1	A_0	D_3	D_2	D_1	D_0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0



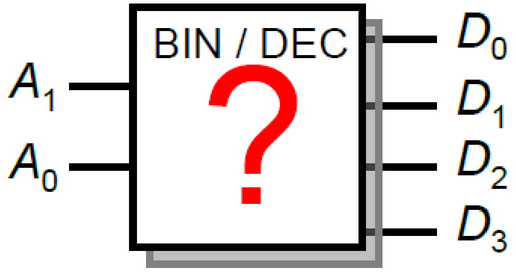
$D_0 = A_1' A_0' = m_0$ $\quad$ $D_2 = A_1 A_0' = m_2$

$D_1 = A_1' A_0 = m_1$ $\quad$ $D_3 = A_1\, A_0 = m_3$

Active high decoder!

2-to-4 Decoder (Active Low decoder)

- Input: 2-bit $(A_0 \,\&\, A_1)$
- Output: 4-bit (D_0, D_1, D_2, D_3)



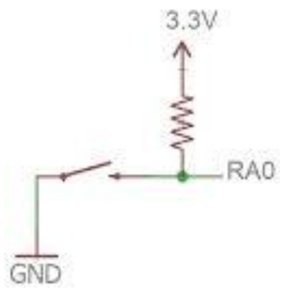
Input		Output			
A_1	A_0	D_3	D_2	D_1	D_0
0	0	1	1	1	0
0	1	1	1	0	1
1	0	1	0	1	1
1	1	0	1	1	1

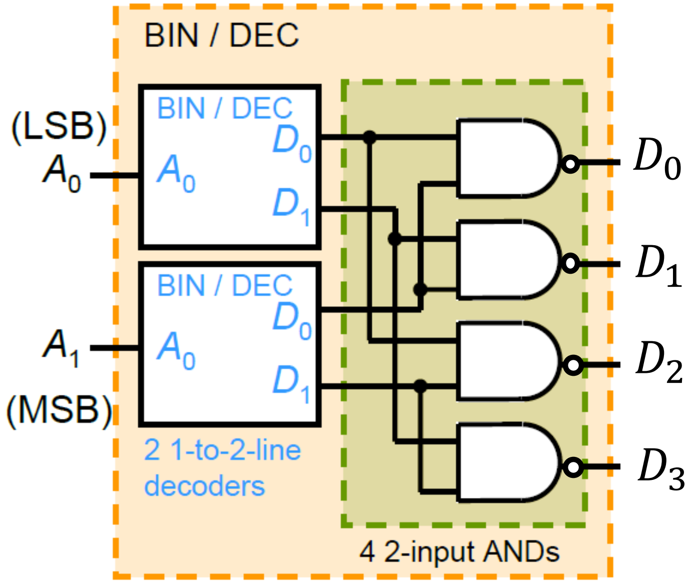
$$D_0 = (A_1' A_0')'$$

$$D_1 = (A_1' A_0)'$$

$$D_2 = (A_1 A_0')'$$

$$D_3 = (A_1 A_0)'$$



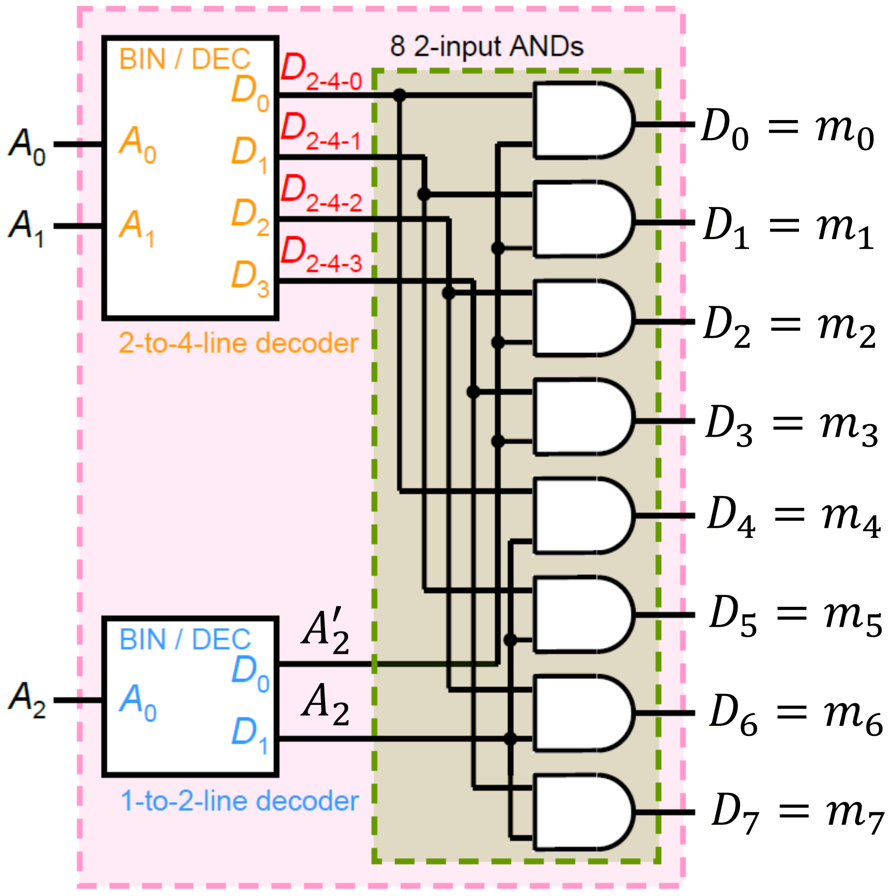


3-to-8 Decoder

Input			Output							
A_2	A_1	A_0	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1	0	0
0	1	1	0	0	0	0	1	0	0	0
1	0	0	0	0	0	1	0	0	0	0
1	0	1	0	0	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0

$D_{2-4-0} = A_1' A_0'$ $D_{2-4-2} = A_1 A_0'$

$D_{2-4-1} = A_1' A_0$ $D_{2-4-3} = A_1 A_0$



Use simpler block to build higher order decoder

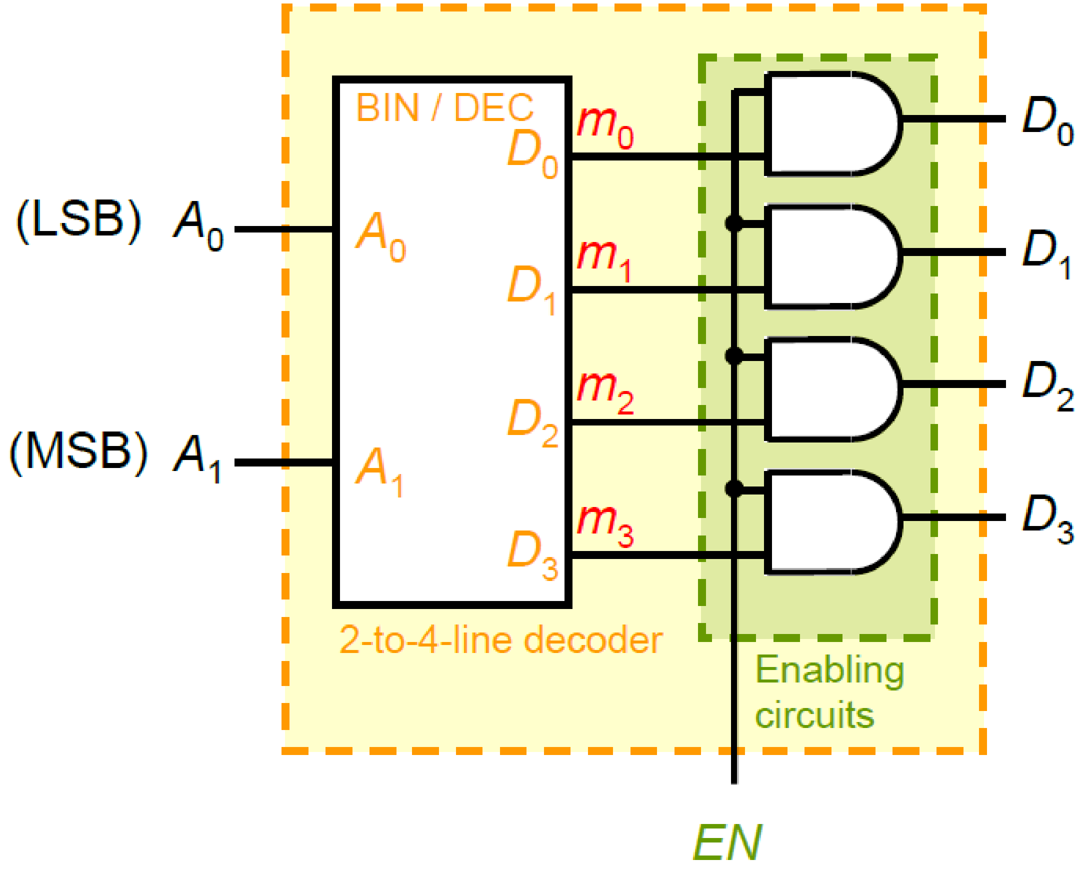
Decoder with Enable Input (2-to-4)

- Input: 3-bit (A_0, A_1, EN)
- Output: 4-bit (D_0, D_1, D_2, D_3)
- Decoder is activated only when EN = 1

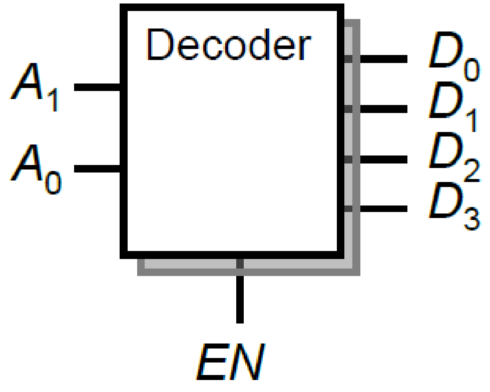
Input			Output			
EN	A_1	A_0	D_3	D_2	D_1	D_0
0	x	x	0	0	0	0
1	0	0	0	0	0	1
1	0	1	0	0	1	0
1	1	0	0	1	0	0
1	1	1	1	0	0	0

Decoder with Enable Input (2-to-4)

- $D_i = EN \cdot m_i$



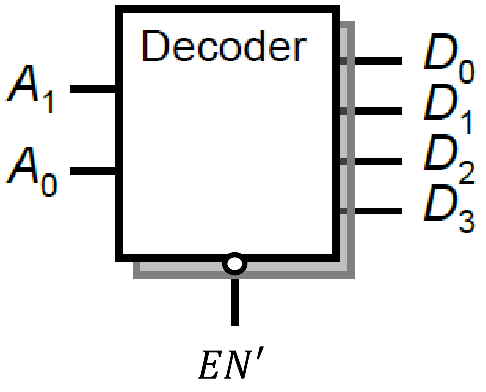
The corresponding symbol of 2-to-4-line decoder with enabling

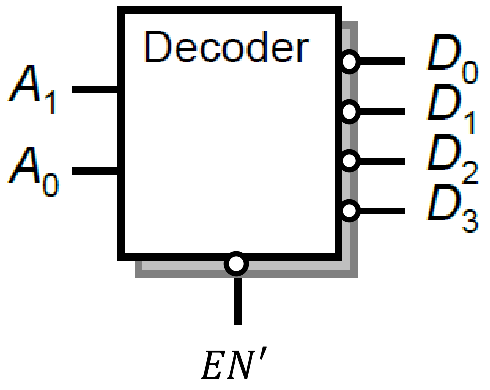


Decoder with Enable Input (2-to-4)

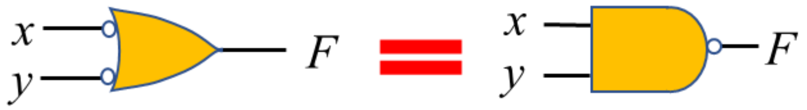
Input			Output			
EN'	A_1	A_0	D_3	D_2	D_1	D_0
1	x	x	0	0	0	0
0	0	0	0	0	0	1
0	0	1	0	0	1	0
0	1	0	0	1	0	0
0	1	1	1	0	0	0

Input			Output			
EN'	A_1	A_0	D_3	D_2	D_1	D_0
1	x	x	1	1	1	1
0	0	0	1	1	1	0
0	0	1	1	1	0	1
0	1	0	1	0	1	1
0	1	1	0	1	1	1





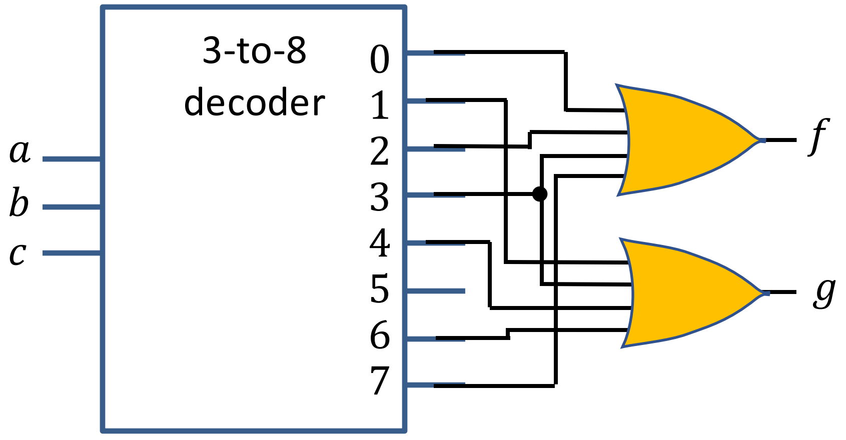
Example



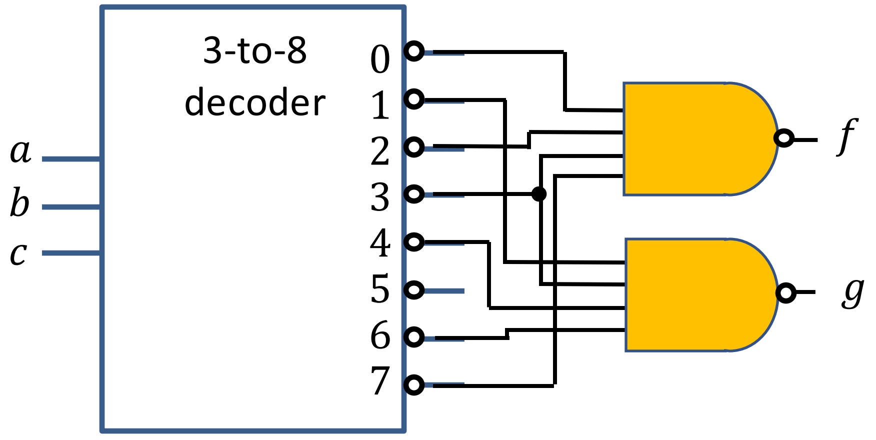
Given that

$f(a,b,c) = \sum m(0,2,3,7)$
$g(a,b,c) = \sum m(1,3,4,6)$

Use (a) active high, and (b) active low 3-to-8 decoder to realize the functions.



Active high

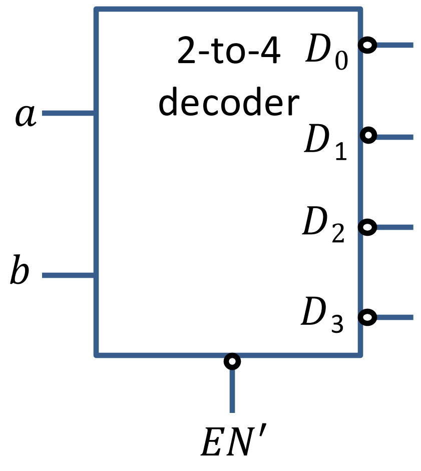


Active low

Example

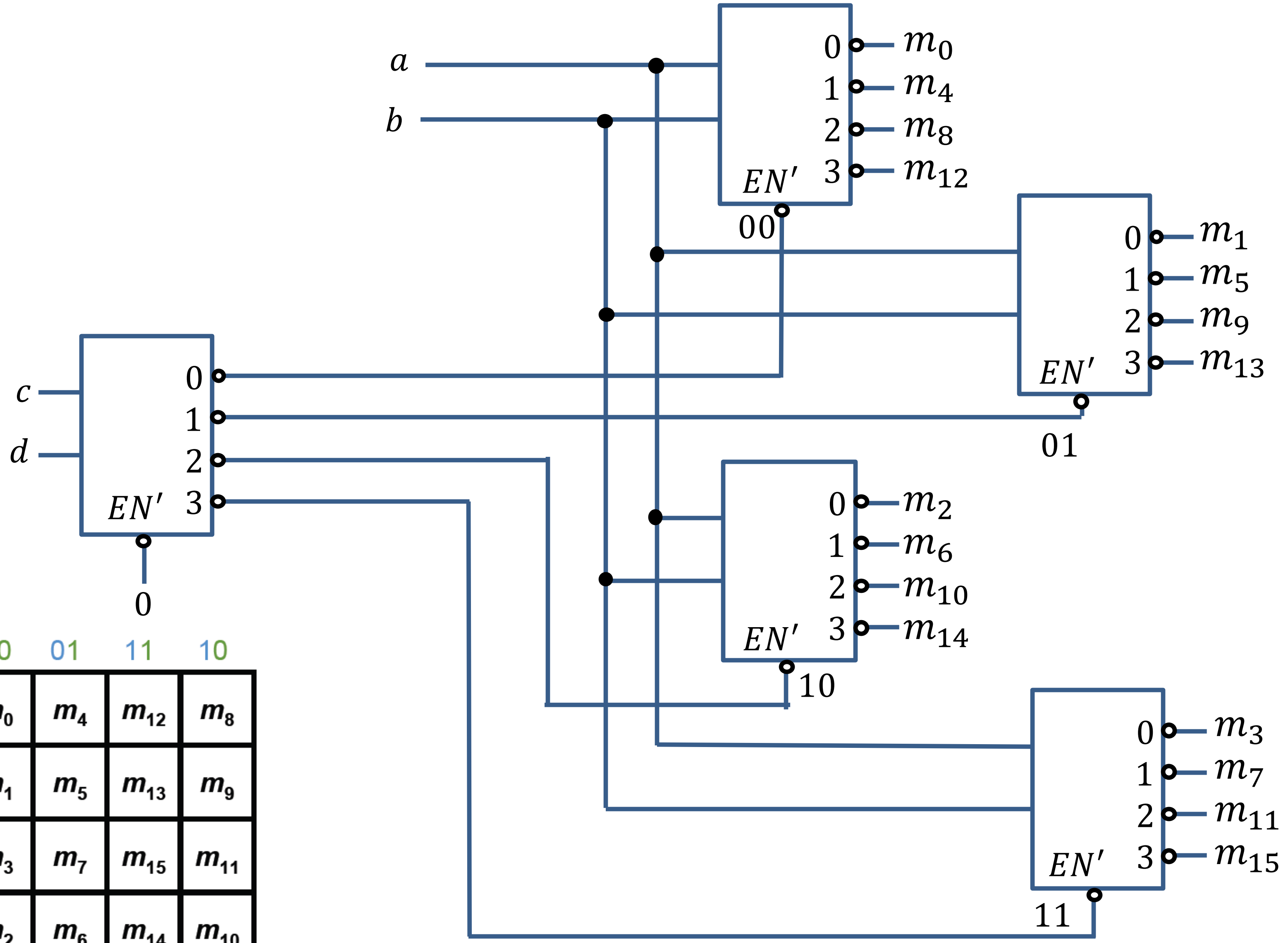
Build a 16-bit decoder using Active Low 2-to-4 decoders and mapped all the minterms.

- 16-bit = 16 outputs = 4 decoders
- Use 2 bits to control the 4 decoders via EN input



Input			Output			
EN'	a	b	D_3	D_2	D_1	D_0
1	x	x	1	1	1	1
0	0	0	1	1	1	0
0	0	1	1	1	0	1
0	1	0	1	0	1	1
0	1	1	0	1	1	1

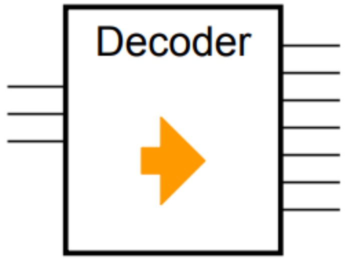
Example (4-variable K-map)

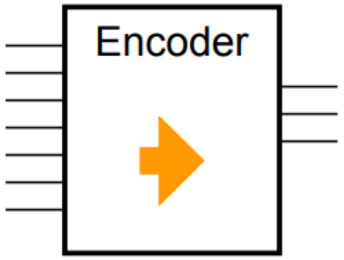


cd \ ab	00	01	11	10
00	m_0	m_4	m_{12}	m_8
01	m_1	m_5	m_{13}	m_9
11	m_3	m_7	m_{15}	m_{11}
10	m_2	m_6	m_{14}	m_{10}

Binary Encoder

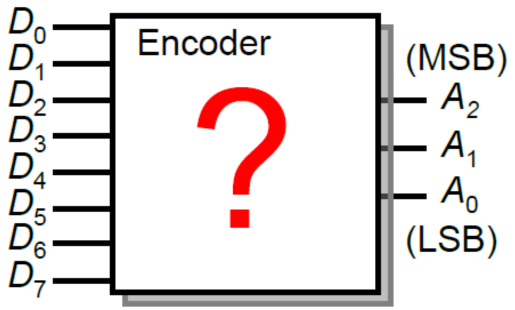
- A functional block that performs the inverse operation of a decoder
- ***m*** inputs and ***n*** outputs
- $\boldsymbol{m} \leq 2^{n}$, but usually $\boldsymbol{m} = 2^{n}$
- Only one input can be '1' at a time.





Example (Octal-to-Binary Encoder)

- 8 inputs and 3 outputs
- Only one input can be '1' at a time

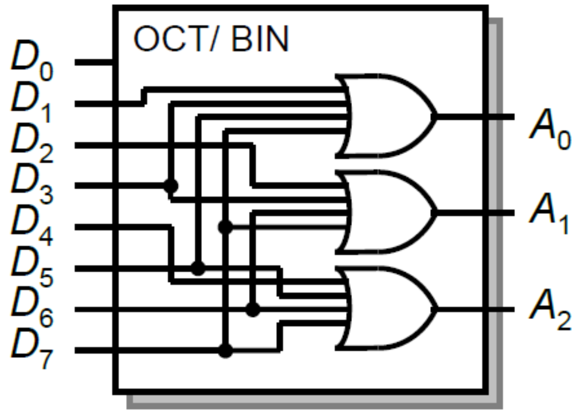


No	Input								Output		
	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	A_2	A_1	A_0
0	0	0	0	0	0	0	0	1	0	0	0
1	0	0	0	0	0	0	1	0	0	0	1
2	0	0	0	0	0	1	0	0	0	1	0
3	0	0	0	0	1	0	0	0	0	1	1
4	0	0	0	1	0	0	0	0	1	0	0
5	0	0	1	0	0	0	0	0	1	0	1
6	0	1	0	0	0	0	0	0	1	1	0
7	1	0	0	0	0	0	0	0	1	1	1

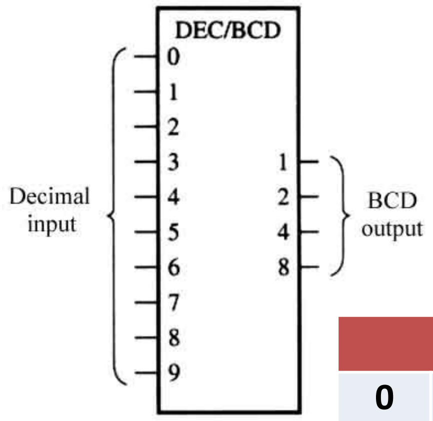
Example (Octal-to-Binary Encoder)

No	Input								Output		
	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	A_2	A_1	A_0
0	0	0	0	0	0	0	0	1	0	0	0
1	0	0	0	0	0	0	1	0	0	0	1
2	0	0	0	0	0	1	0	0	0	1	0
3	0	0	0	0	1	0	0	0	0	1	1
4	0	0	0	1	0	0	0	0	1	0	0
5	0	0	1	0	0	0	0	0	1	0	1
6	0	1	0	0	0	0	0	0	1	1	0
7	1	0	0	0	0	0	0	0	1	1	1

$$A_0 = D_1 + D_3 + D_5 + D_7$$
$$A_1 = D_2 + D_3 + D_6 + D_7$$
$$A_2 = D_4 + D_5 + D_6 + D_7$$



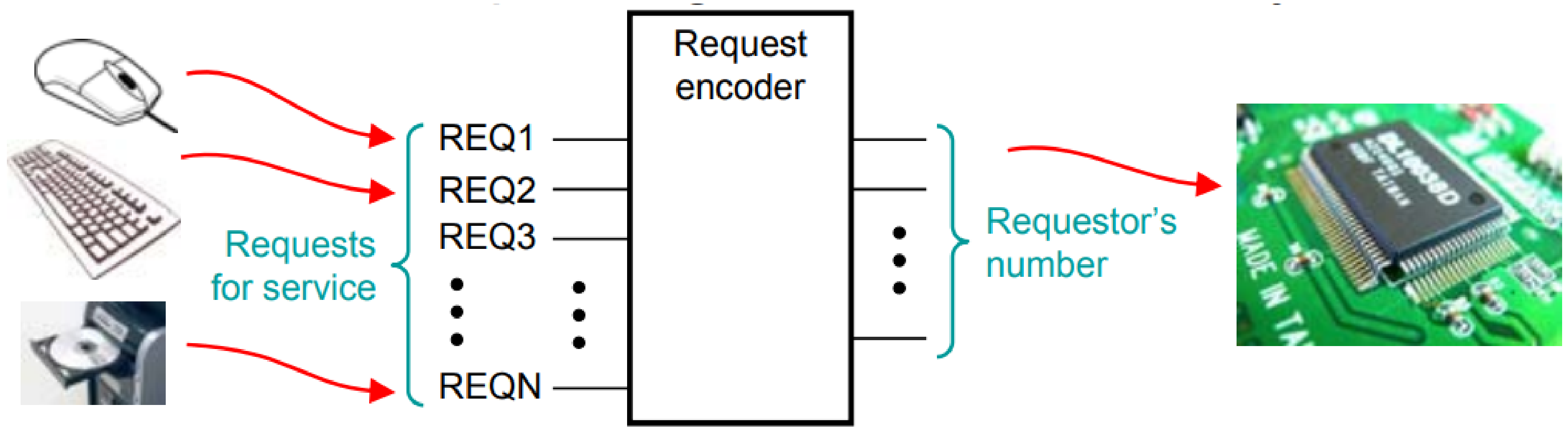
Exercise (Decimal-to-Binary Encoder)



Inputs										Outputs			
0	1	2	3	4	5	6	7	8	9	A_3	A_2	A_1	A_0

Limitation

- Only one input can be active (*i.e.*, 1)
- If two or more inputs active simultaneously, the output produces an incorrect combination



Solution – Priority Encoder

- Only one input can be active (*i.e.*, 1)
- If two or more inputs are active simultaneously, the output produces an incorrect combination
- Solution:
  - To resolve this ambiguity, introduce an input priority
  - Each input pin has a different priority
  - If two or more inputs are 1 at the same time, only consider input that has a higher priority
  - This kind of encoder is called a priority encoder

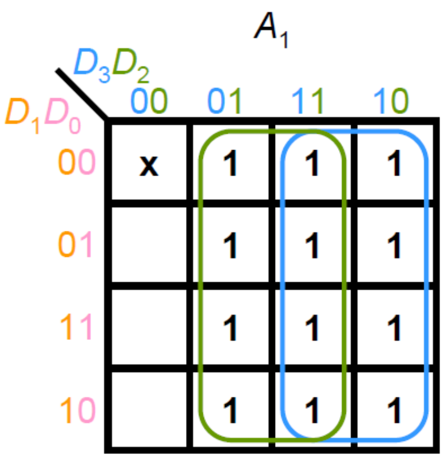
Example (4-input Priority Encoder)

- Design a 4-input priority encoder whereby inputs with higher subscript numbers has higher priority.
- Add an extra output ***V*** stands for valid output.
- If all inputs are 0, ***V*** is 0; else ***V*** is 1.

Input				Output		
D_3	D_2	D_1	D_0	A_1	A_0	V
0	0	0	0	X	X	0
0	0	0	1	0	0	1
0	0	1	X	0	1	1
0	1	X	X	1	0	1
1	X	X	X	1	1	1

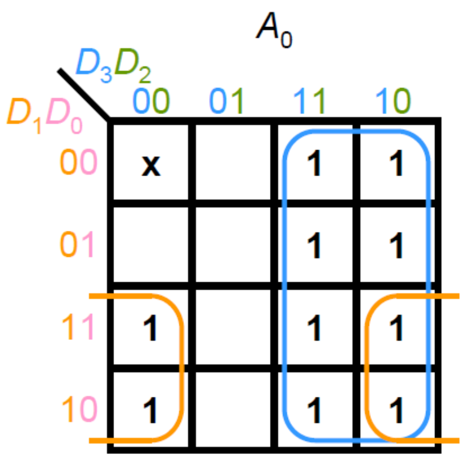
Example (4-input Priority Encoder)

Input				Output		
D_3	D_2	D_1	D_0	A_1	A_0	V
0	0	0	0	X	X	0
0	0	0	1	0	0	1
0	0	1	X	0	1	1
0	1	X	X	1	0	1
1	X	X	X	1	1	1



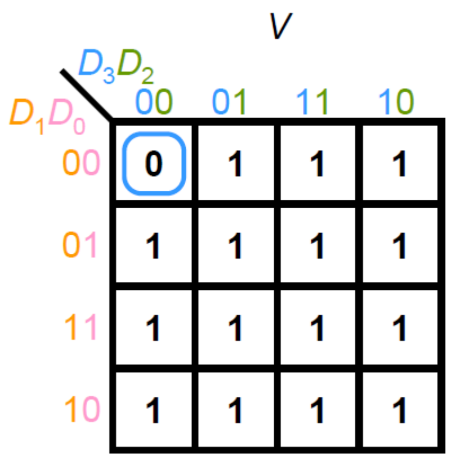
D_1D_0 \ D_3D_2	00	01	11	10
00	x	1	1	1
01		1	1	1
11		1	1	1
10		1	1	1

$A_1(D_3,D_2,D_1,D_0)$
$= D_3 + D_2$



D_1D_0 \ D_3D_2	00	01	11	10
00	x		1	1
01			1	1
11	1		1	1
10	1		1	1

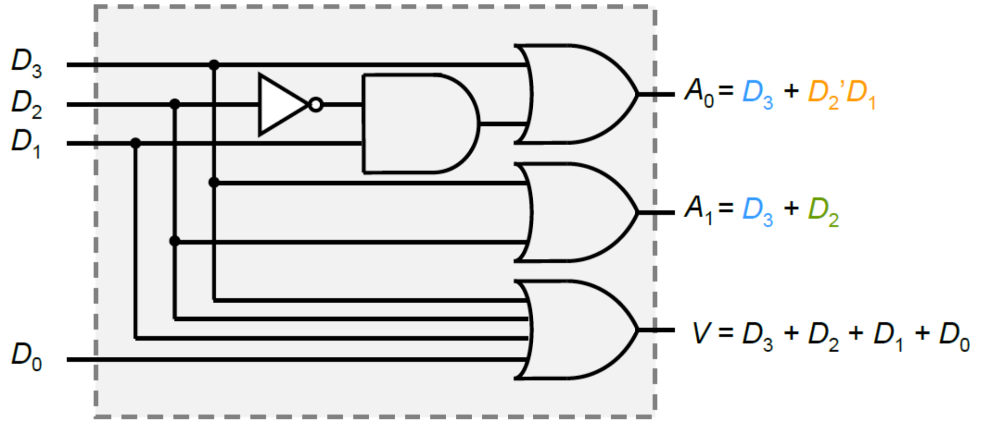
$A_0(D_3,D_2,D_1,D_0)$
$= D_3 + D_2'D_1$



D_1D_0 \ D_3D_2	00	01	11	10
00	0	1	1	1
01	1	1	1	1
11	1	1	1	1
10	1	1	1	1

$V' = D_3'D_2'D_1'D_0'$

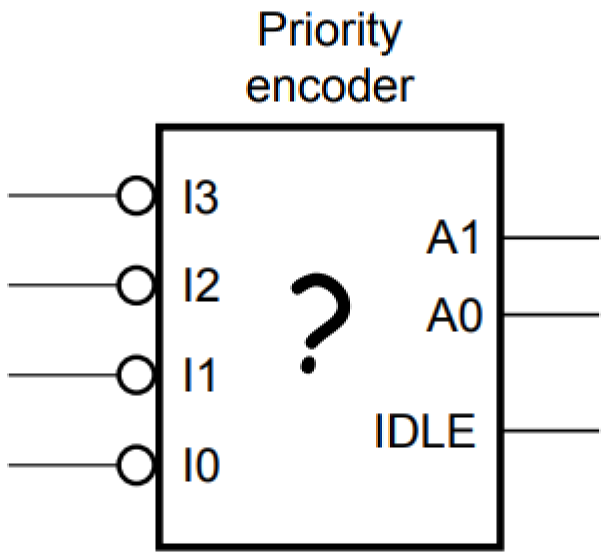
$V = D_3 + D_2 + D_1 + D_0$



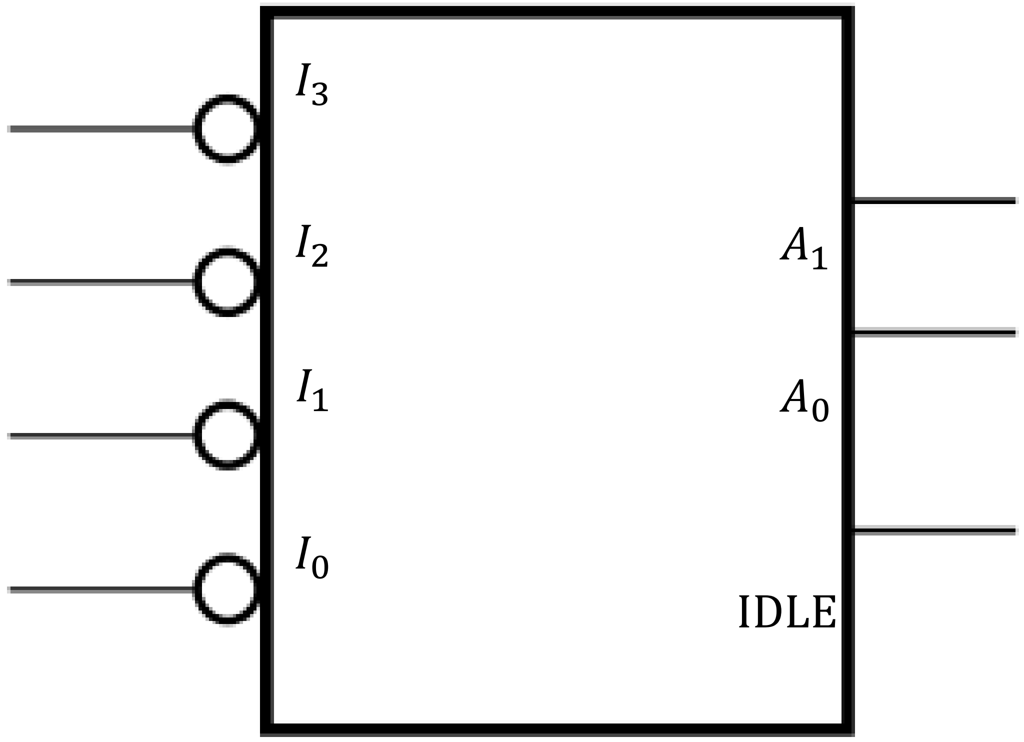
Exercise (Active Low)

- Design an Active Low 4-input priority encoder whereby inputs with higher subscript numbers has higher priority.
- Output IDLE is High when all inputs are high.

Input				Output		
I_3	I_2	I_1	I_0	A_1	A_0	IDLE

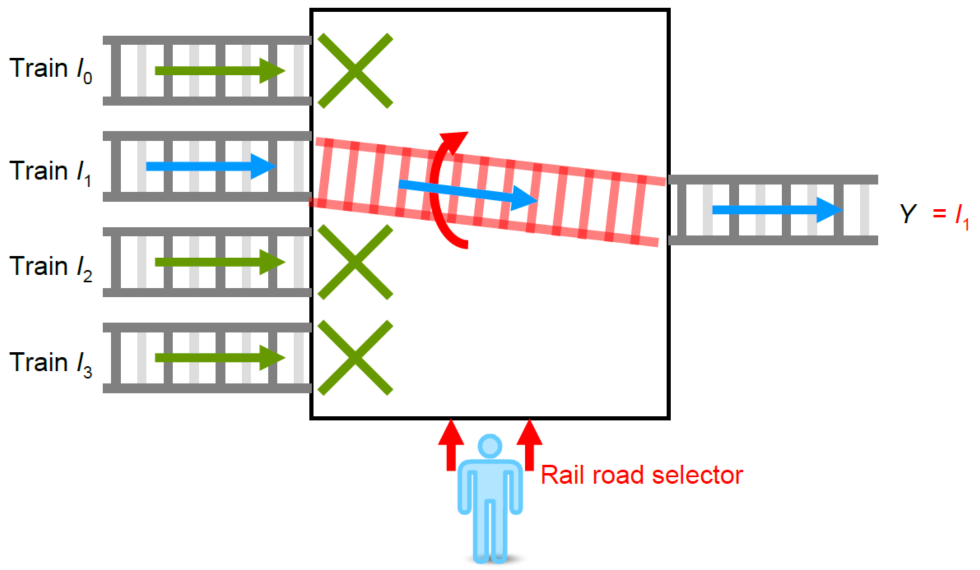


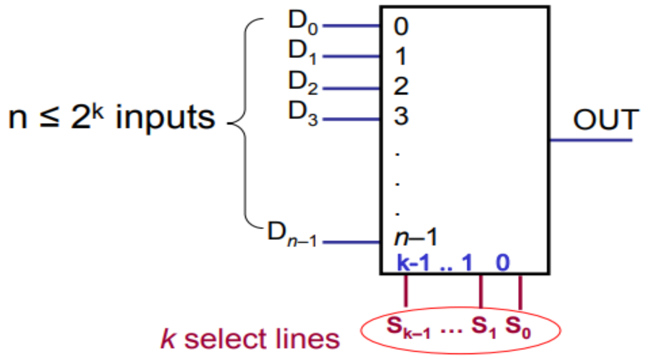
Exercise (Active Low)



Multiplexer (MUX)

- Basically a digital switch
- Pass one of the inputs to the output
- Selected by the control input.

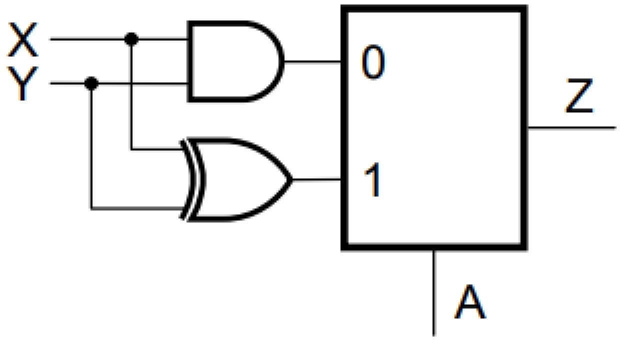


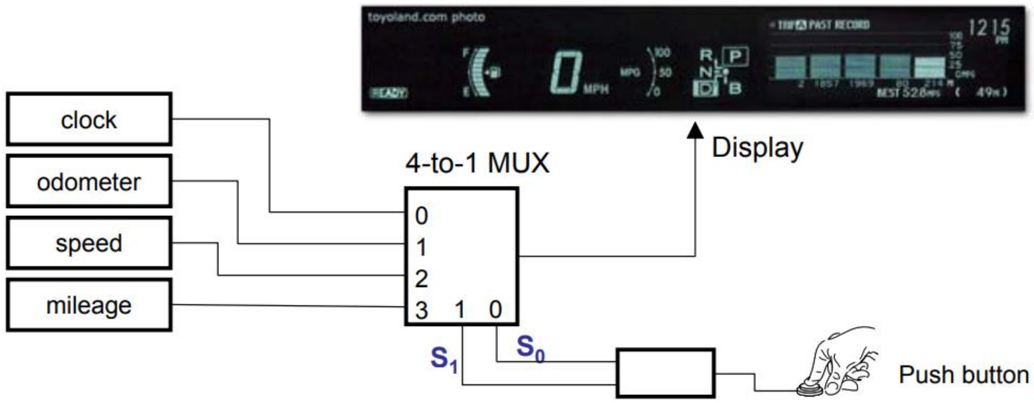


- ***k*** control inputs
- ***n*** data inputs, ***n*** $\le 2^k$
- 1 output

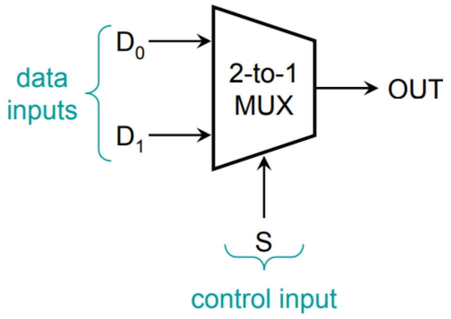
Applications

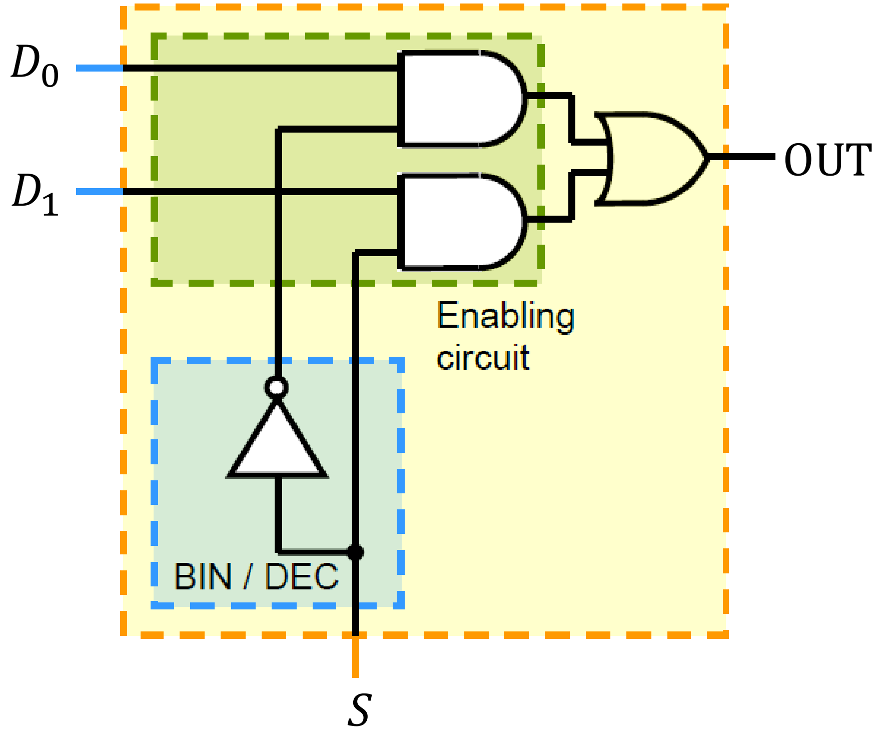
- In computers, select signals
- To implement different functions
- Trip controller in a car to choose different displays
- …





2-to-1 Multiplexer (MUX)





Input			Output
S	D_1	D_0	
0	x	x	D_0
1	x	x	D_1

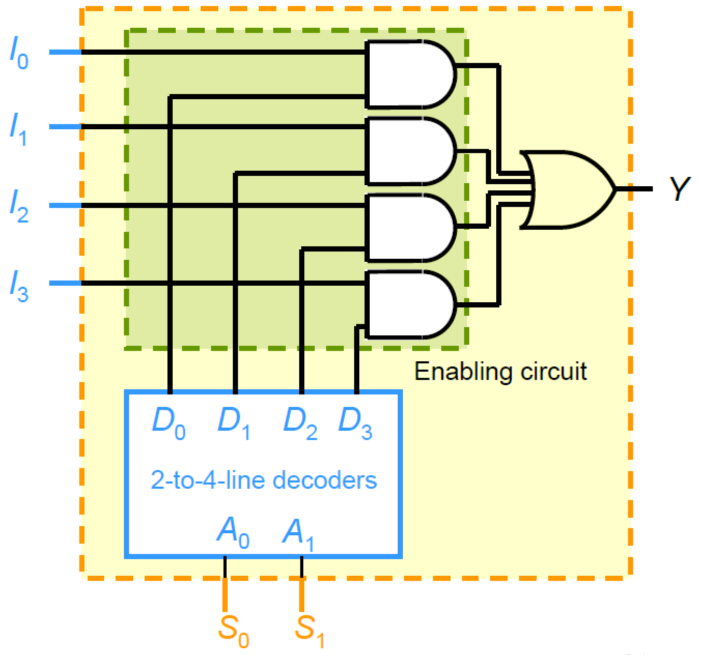
$$\mathrm{OUT} = S'D_0 + SD_1$$

4-to-1 Multiplexer (MUX)

- Specification:
  - $m = 4$
  - $n = \log_2 m = 2$
- Formulation:

Inputs						Output
I_0	I_1	I_2	I_3	S_1	S_0	Y
x	x	x	x	0	0	I_0
x	x	x	x	0	1	I_1
x	x	x	x	1	0	I_2
x	x	x	x	1	1	I_3

- Optimization:
  - $Y(I_0,I_1,I_2,I_3,S_1,S_0) = S_1'S_0'I0 + S_1'S_0I_1 + S_1S_0'I_2 + S_1S_0I_3$



Example

Realize the function $f(w, x, y, z) = \sum m(1, 2, 5, 7, 9, 11, 13)$ using a 4-to-1 MUX

STEP 1: Plot the K-map

wx \ yz	00	01	11	10
00		1		1
01		1	1	
11		1		
10		1	1	

Example

STEP 2: Since 4-to-1 MUX has 2 control inputs, choose two variables for these inputs

$$w = S_1 \quad x = S_0$$

wx \ yz	00	01	11	10
00		1		1
01		1	1	
11		1		
10		1	1	

STEP 3:

$$f(w = 0, x = 0) = y'z + yz'$$

$$f(w = 0, x = 1) = z$$

$$f(w = 1, x = 1) = y'z$$

$$f(w = 1, x = 0) = z$$

Example

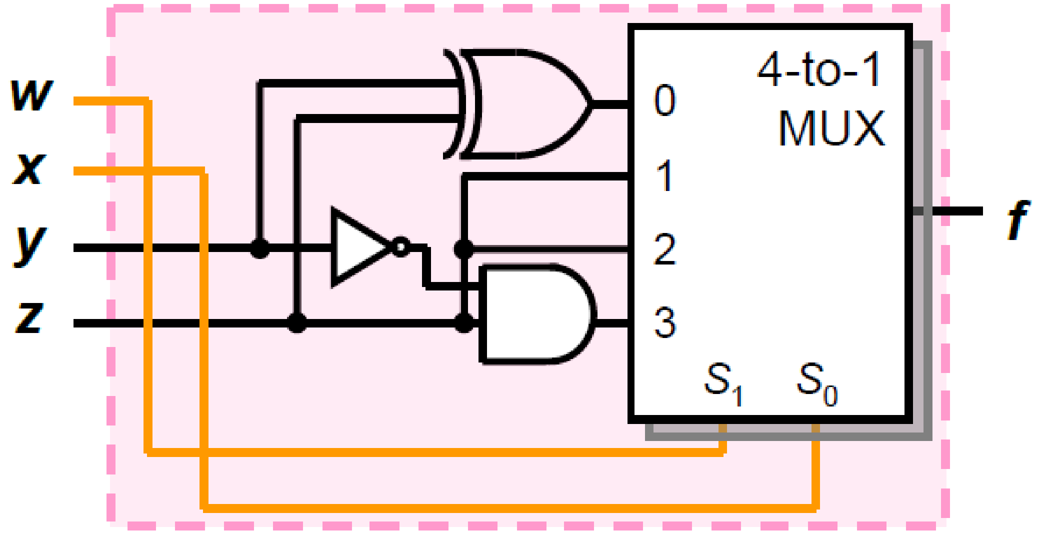
$w = S_1 \quad x = S_0$

$f(w = 0, x = 0) = y'z + yz'$

$f(w = 0, x = 1) = z$

$f(w = 1, x = 1) = y'z$

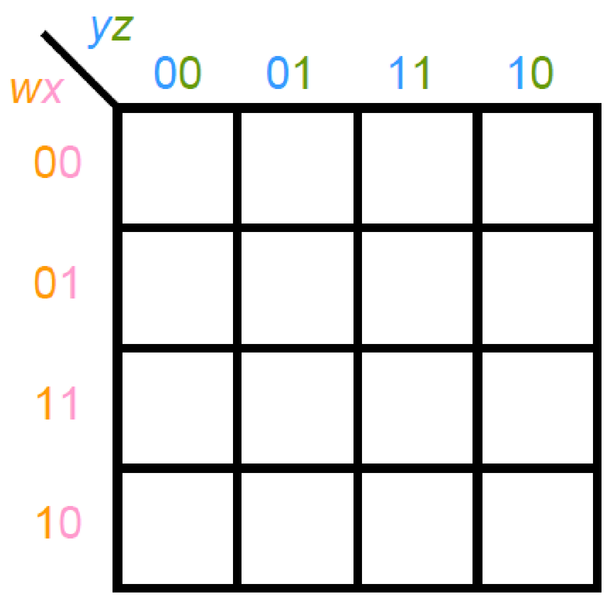
$f(w = 1, x = 0) = z$



Question: How can we realize the function using 2-to-1 muxs?

Exercise

Realize the function $f(w, x, y, z) = \sum m(1, 2, 5, 7, 9, 11, 13)$ using a 2-to-1 MUX



Exercise

Realize the function $f(w, x, y, z) = \sum m(1, 2, 5, 7, 9, 11, 13)$ using an 8-to-1 MUX

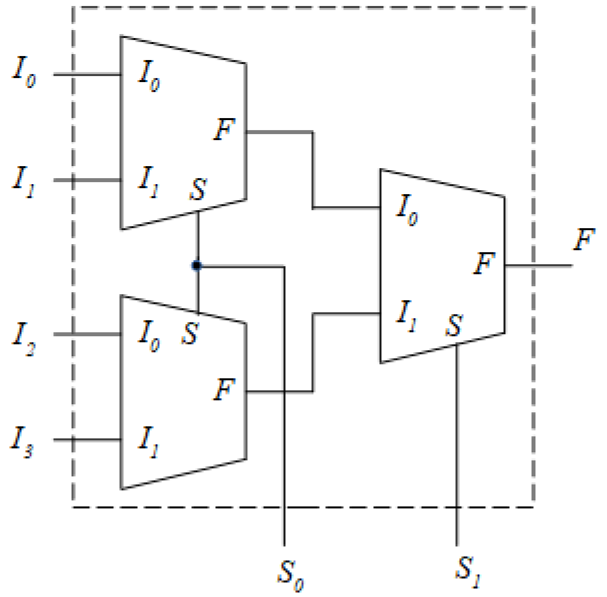
wxyz	*F*
0000	0
0001	1
0010	1
0011	0
0100	0
0101	1
0110	0
0111	1

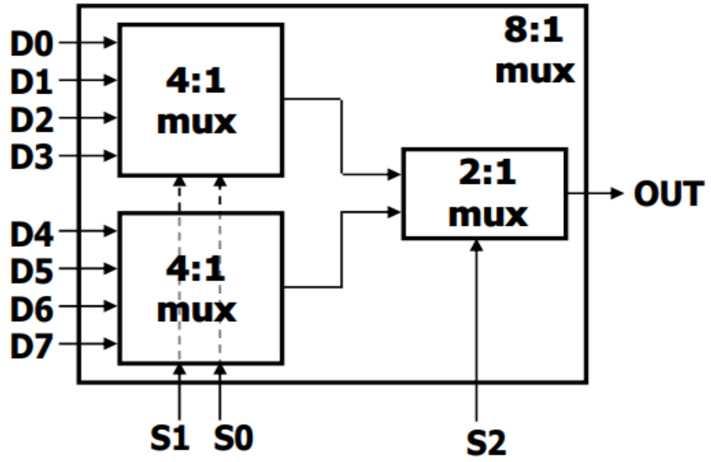
wxyz	*F*
1000	0
1001	1
1010	0
1011	1
1100	0
1101	1
1110	0
1111	0

Summary

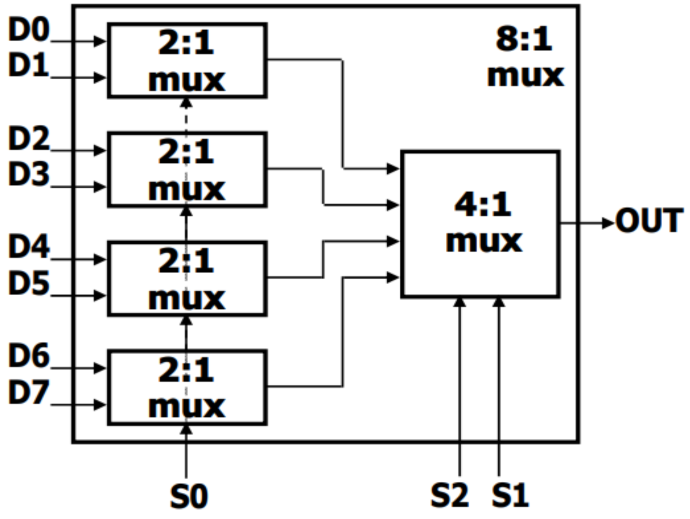
	With	2-to-1 Mux	4-to-1 Mux	8-to-1 Mux
wxyz	*F*	$S_0=w$	$S_1=w, S_0=x$	$S_2=w, S_1=x, S_0=y$
0000	0	$I_0=xz+y'z+x'yz'$	$I_0=y \oplus z$	$I_0=z$
0001	1			
0010	1			$I_1=z'$
0011	0			
0100	0		$I_1=z$	$I_2=z$
0101	1			
0110	0			$I_3=z$
0111	1			
1000	0	$I_1=x'z+y'z$	$I_2=z$	$I_4=z$
1001	1			
1010	0			$I_5=z$
1011	1			
1100	0		$I_3=y'z$	$I_6=z$
1101	1			
1110	0			$I_7=0$
1111	0			

Cascading Multiplexers



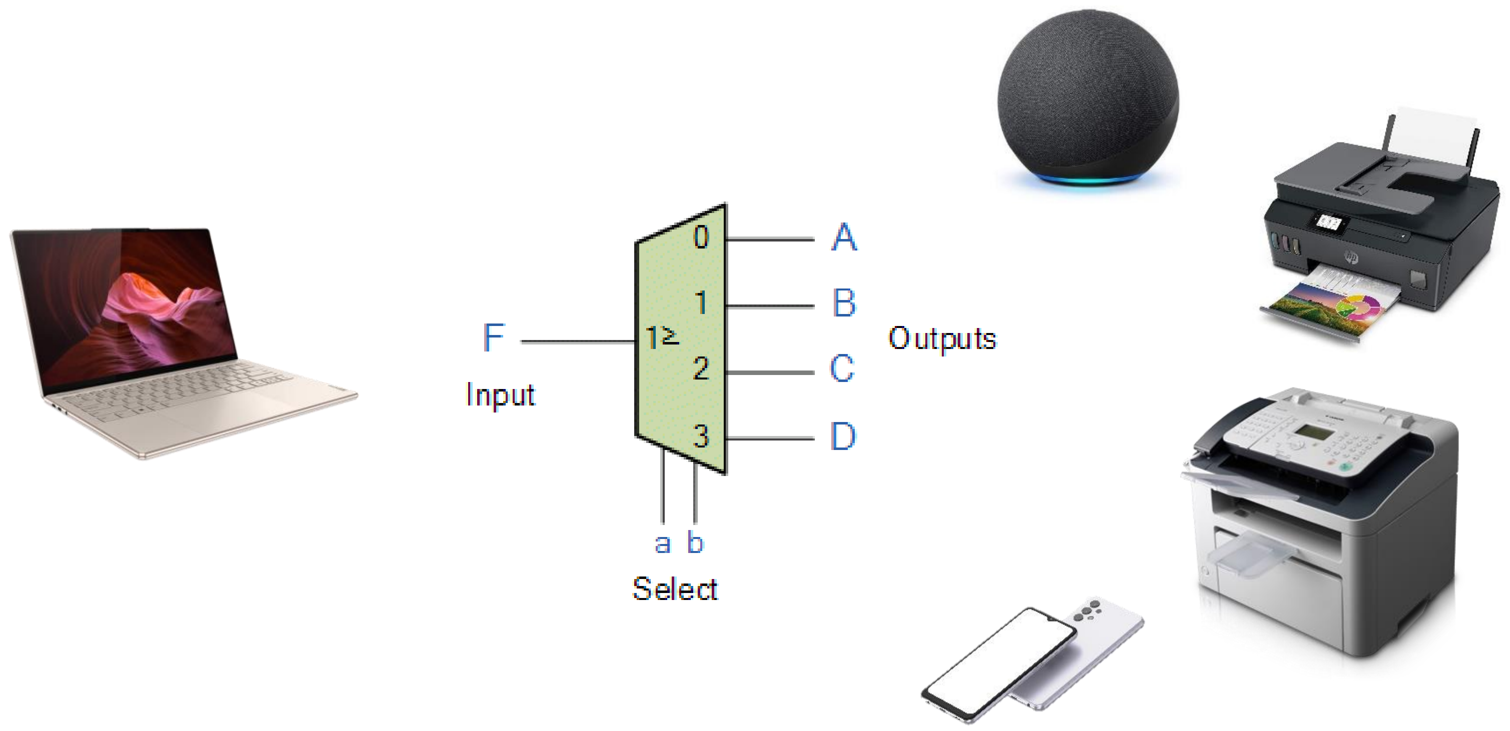


Inputs						Output
I_0	I_1	I_2	I_3	S_1	S_0	Y
x	x	x	x	0	0	I_0
x	x	x	x	0	1	I_1
x	x	x	x	1	0	I_2
x	x	x	x	1	1	I_3



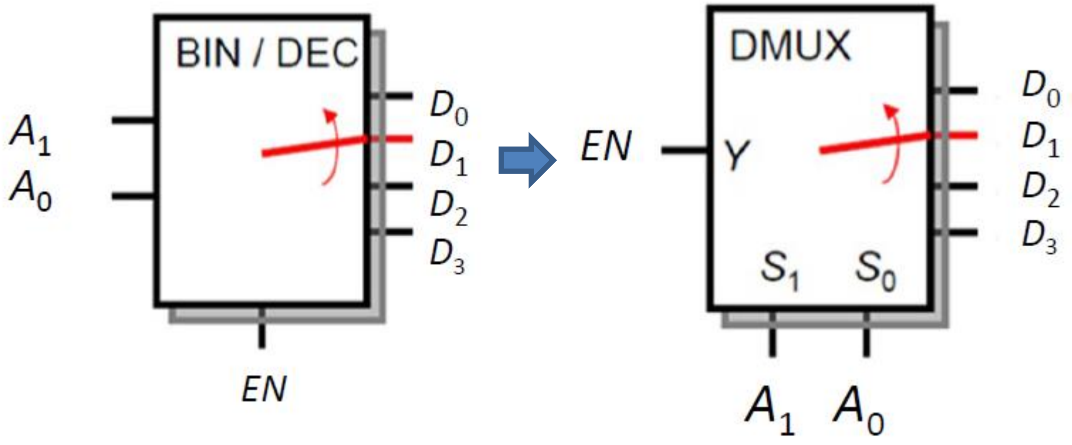
Demultiplexer (DMUX)

- Reverse the function of MUX
- Route a single input to one of the many outputs
- Selected by the control input.



Demultiplexer (DMUX)

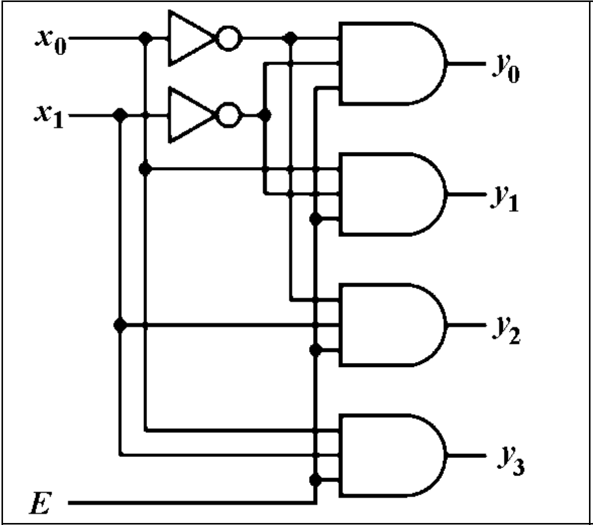
- Remember the decoder with Enable?
- The decoder can perform demultiplexer if we take EN as the input line, A_i (input lines of decoder) as the selection inputs



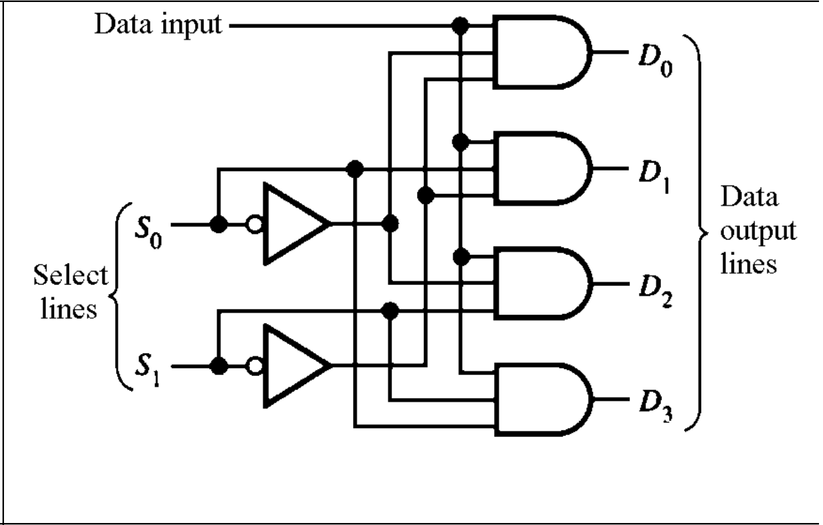
Input	Select lines		Output			
EN	A_1	A_0	D_3	D_2	D_1	D_0
1	0	0	0	0	0	1
1	0	1	0	0	1	0
1	1	0	0	1	0	0
1	1	1	1	0	0	0
0	X	X	0	0	0	0

Demultiplexer (DMUX)

- Can be realized using decoder with enable input
- Input lines -> Select/Control lines
- Enable -> Data input



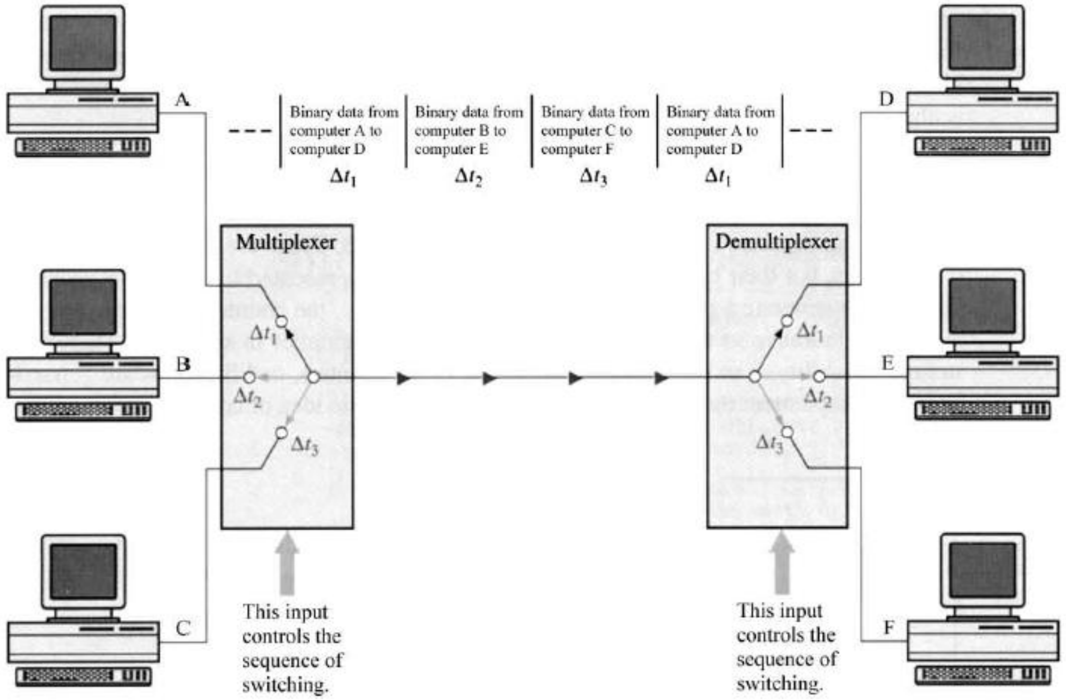
Decoder with enable



Demultiplexer

Example of MUX & DMUX Application

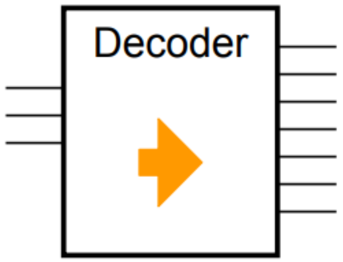
- A MUX allows digital information from several sources to be routed onto a single line for transmission over that line to a common destination
- A DMUX basically reverses the multiplexing function


A
B
C
D
E
F
Binary data from computer A to computer D Δt_1
Binary data from computer B to computer E Δt_2
Binary data from computer C to computer F Δt_3
Binary data from computer A to computer D Δt_1
Multiplexer
Demultiplexer
Δt_1
Δt_2
Δt_3
This input controls the sequence of switching.
This input controls the sequence of switching.

Summary

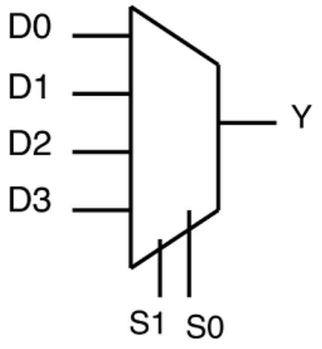
Only 1 output is ‘1’

n inputs



2^n outputs

Binary Decoder

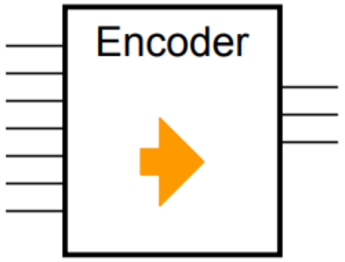


1 output

MUX

Only 1 input is ‘1’

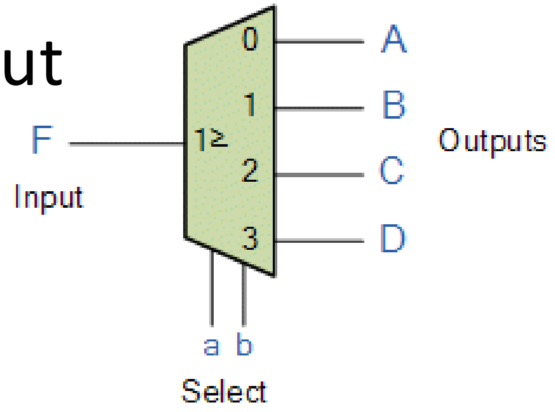
2^n inputs



n outputs

Binary Encoder

1 input



DMUX

Supplementary Information (For your own interest)

Signed Numbers

- In ordinary arithmetic, a plus(+)/minus(-) sign is used to represent positive or negative numbers (+4 or -4)
- In digital electronic circuits, everything is represented with a bit (0 or 1)
- There are several ways to represent the signed binary numbers using a bit
- **Sign magnitude representation**: The **MSB** is a sign bit

e.g.	01101 = +13	11101 = -13
	00000 = +0	10000 = -0

Disadvantages: (1) **2 patterns represent 0**,

(2) **handle sign bit separately**

One’s Complement

- Positive numbers and the corresponding negative numbers **complement** each other
- Complement of 0 is 1; 1 is 0

  e.g. 01101 = +13 10010 = -13

  00000 = +0 11111 = -0

- Advantage: **Symmetry and easy**
- Disadvantage: **2 patterns to represent 0**

Two's Complement

- Positive numbers and negative numbers are their corresponding **1's complement number + 1**

  e.g. 01101 = +13

  10010 + 1 = 10011 = -13

  00000 = +0

  11111 + 1 = 00000 = +0

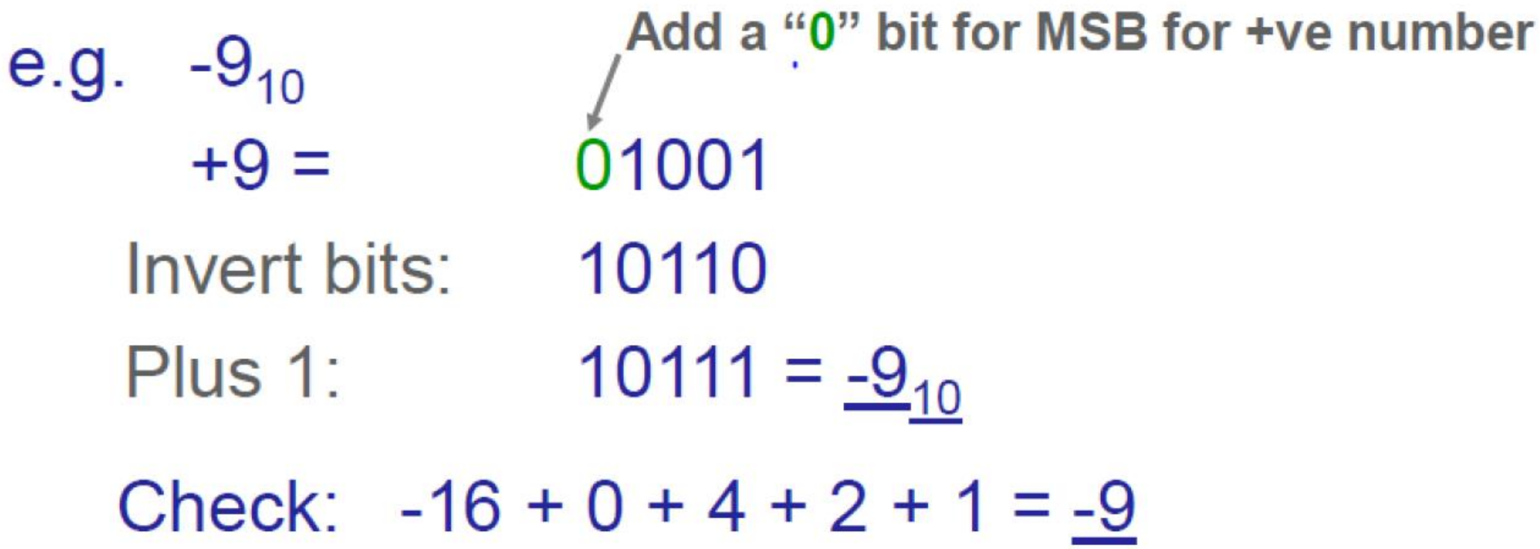
- Advantages: **(1) Only 1 pattern represents 0**

  (2) Handle sign bit as other bits

Value of a Two's Complement Number

- For an ***n***-bit Two's Complement number $a_{n-1}a_{n-2}\ldots a_1 a_0$, it has a value of

$$N = (-1)a_{n-1} \times 2^{n-1} + a_{n-2} \times 2^{n-2} + \cdots + a_1 \times 2^1 + a_0 \times 2^0$$



Add a "0" bit for MSB for +ve number

e.g. -9_{10}

+9 = 01001

Invert bits: 10110

Plus 1: 10111 = $\underline{-9_{10}}$

Check: -16 + 0 + 4 + 2 + 1 = $\underline{-9}$

Binary Addition

- Two's complement numbers can be added in ordinary binary addition
- Any carry beyond the MSB will be discarded/ignored

```
     +3        0011                -2        1110
 +   +4    +   0100            +   -6    +   1010
 ------    --------            ------    --------
     +7        0111                -8       11000

     +6        0110                +4        0100
 +   -3    +   1101            +   -7    +   1001
 ------    --------            ------    --------
     +3       10011                -3        1101
```

Range and Overflow

Unsigned	0000 → 1111	0 → 15
1's Complement	1000 → 1111	-7 → -0
	0000 → 0111	0 → 7
2's Complement	1000 → 1111	-8 → -1
	0000 → 0111	0 → 7

- For 4-bit binary, the range is
- Overflow occurs when the addition operation produces a result that exceed the range of the number system
- Overflow is detected when addition of 2 numbers with same signs produces a different sign

$$\begin{array}{r} -3 \\ +\ \ -6 \\ \hline -9 \end{array} \qquad \begin{array}{r} 1101 \\ +\ \ 1010 \\ \hline 10111 \end{array} = +7 \qquad\qquad \begin{array}{r} +5 \\ +\ \ +6 \\ \hline +11 \end{array} \qquad \begin{array}{r} 0101 \\ +\ \ 0110 \\ \hline 1011 \end{array} = -5$$

$$\begin{array}{r} -8 \\ +\ \ -8 \\ \hline -16 \end{array} \qquad \begin{array}{r} 1000 \\ +\ \ 1000 \\ \hline 10000 \end{array} = +0 \qquad\qquad \begin{array}{r} +7 \\ +\ \ +7 \\ \hline +14 \end{array} \qquad \begin{array}{r} 0111 \\ +\ \ 0111 \\ \hline 1110 \end{array} = -2$$

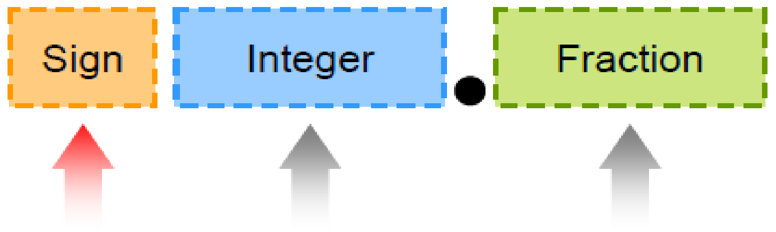
Ripple Carry Subtractor

- How can we build an ***n***-bit subtractor using ***n*** 1-bit Full Adder?

  $(X_3X_2X_1X_0) + (-Y_3Y_2Y_1Y_0)$

- How to represent negative binary number?

  1) Sign-magnitude Format



  2) Complement Format

Sign-Magnitude Binary

- The MSB is a sign bit
  - 0 means positive "+"
  - 1 means negative "-"

For example:

4-bit system (+5 = 0101; -5= 1101)

Last 3 bits used for magnitude/value

Problems

1) Incorrect result for the addition of negative number

    e.g. Compute the sum of +5 and -3

    +5 = 0101   -3 = 1011

    Sum = 10000 (???) not 0010

2) Two zeros (0000 and 1000)

Two-Complement (4-bit)

1) MSB (0 as positive and 1 as negative)
2) Negative number (-a) = $2^n - a = 16 - a$ for n-bit

Decimal	2's complement
0	0000
+1	0001
+2	0010
+3	0011
+4	0100
+5	0101
+6	0110
+7	0111

Decimal	2's complement
-1	1111
-2	1110
-3	1101
-4	1100
-5	1011
-6	1010
-7	1001
-8	1000

4-bit system
(+5 = 0101; -5= 1101) Sign-Magnitude Binary
(+5 = 0101; -5= 1011) Two's Complement Binary

Easier Way to Find Negative Binary

1) Find the binary equivalent magnitude
2) Complement each bit
3) Add 1

	-5	-0
Step 1)	$5_{10} = 0101_2$	$0_{10} = 0000_2$
Step 2)	1010	1111
Step 3)	1010 +) 1 1011	1111 +) 1 $\cancel{1}$0000
Answer =	1011_2	0000_2

The carry bit can be ignored.

2-Complement Binary to Decimals

1) Complement each bit
2) Add 1 and find the magnitude
3) Add – to represent negative number

	1011_2
Step 1)	0100_2
Step 2)	$\begin{array}{r} 0100 \\ +)\quad 1 \\ \hline 0101 \end{array}$
Step 3)	$0101_2 = 5_{10}$
Magnitude	5
Answer =	-5_{10}

Addition using 2-Complement

- (a) (-5) + 3, (b) (-5) + 7

	(a)		(b)	
Carries		*0 1 1*		*1 1 1*
Augend	- 5	1 0 1 1	- 5	1 0 1 1
Addend	+ 3	0 0 1 1	+ 7	0 1 1 1
Sum	- 2	1 1 1 0	2	$\cancel{1}$ 0 0 1 0

Carry out is ignored

The results are correct now!

Binary Subtraction using 2-Complement

- A – B = A + (-B) = A + (2’s complement of B)
  e.g. 5 – 7

Input: 5 = 0101 and 7 = 0111

Step 1: Complement each bit 7 = 1000

Step 2: Add 1 7 = 1001

Step 3: Perform addition 0101 + 1001 = 1110 (-2)

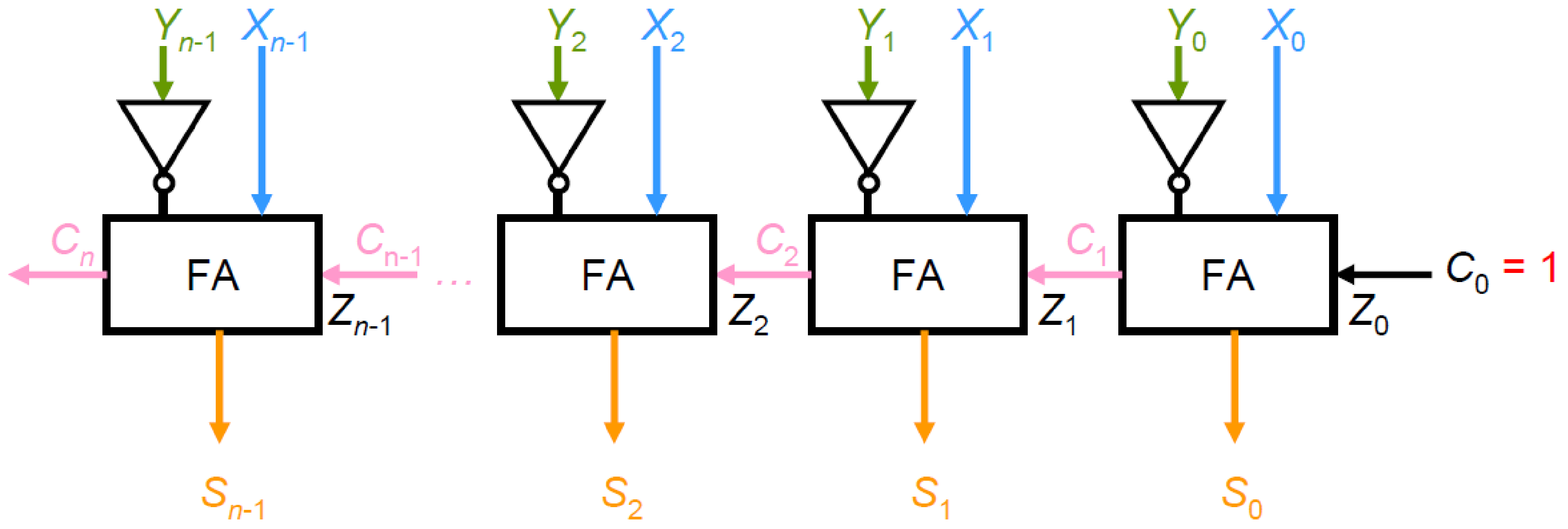
Ripple Carry Subtractor

- How can we build an ***n***-bit subtractor using ***n*** 1-bit Full Adder?

  $(X_3X_2X_1X_0) + (-Y_3Y_2Y_1Y_0)$

  Step1: Complement each bit

  Step 2: Add 1



Exercise

Build an ***n***-bit adder + subtractor using ***n*** 1-bit Full Adder, i.e. combine ripple carry adder and subtractor.

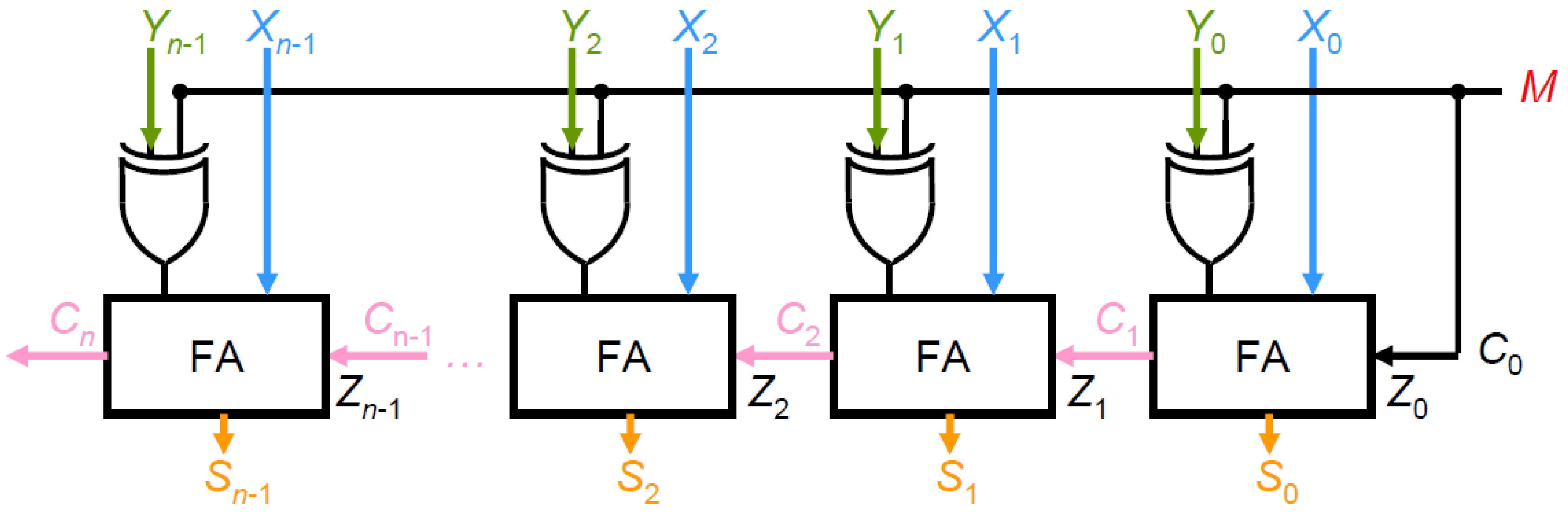
Exercise

For addition, $C_0 = 0$, $Y_i = Y_i$

For subtraction, $C_0 = 1$, $Y_i = Y_i'$

y_i	M	f
0	0	0
1	0	1
0	1	1
1	1	0

- *M* is the mode selection cable (0 means addition, 1 means subtraction)



- For addition ($M = 0$), $C_0 = 0$, $Y_i = Y_i \oplus 0 = Y_i$
- For subtraction ($M = 1$), $C_0 = 1$, $Y_i = Y_i \oplus 1 = Y_i'$