

EE2000 Logic Circuit Design

Combinational Functional Blocks

Outline

- 5.1 Equality Comparator
- 5.2 Arithmetic functional blocks
 - Half adder, Full adder, Ripple carry adder
 - Half subtractor, Full subtractor, Ripple carry subtractor
 - Carry-look-ahead adder
- 5.3 Logical functional blocks
 - Decoder
 - Encoder
 - Multiplexer
 - Demultiplexer
- 5.4 Programmable logic devices
 - ROM, PLA, PAL

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 unequal

1-bit Equality Comparator

■ Formulation:

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

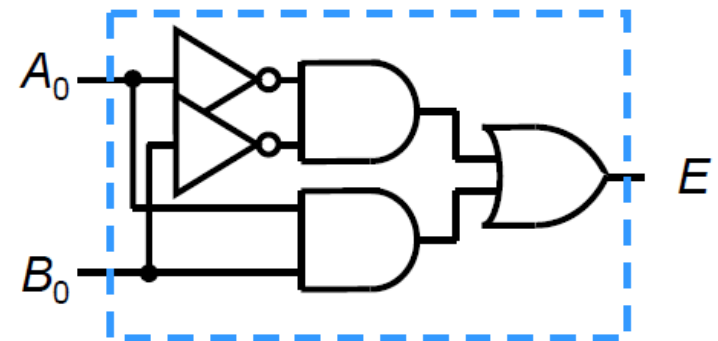
■ Optimization:

■ $E(A_0, B_0) = \sum m(0, 3)$

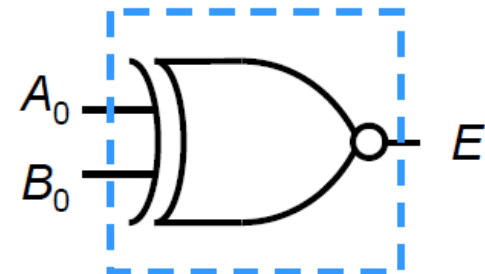
■ $= A_0'B_0' + A_0B_0$

■ $= A_0 \oplus B_0$

■ 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

■ Optimization:

■ $E(A_1, A_0, B_1, B_0)$

■ $= \sum m(0, 5, 10, 15)$

■ $= A_1'A_0'B_1'B_0' +$

■ $A_1'A_0B_1'B_0 + A_1A_0'B_1B_0'$

■ $+ A_1A_0B_1B_0$

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:

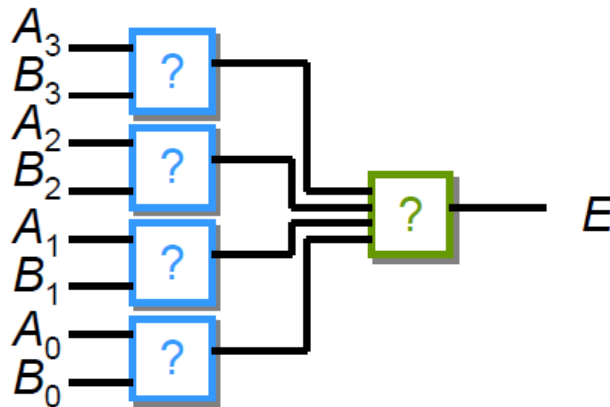
Modular design

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

4-bit Equality Comparator

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



Functional Blocks:

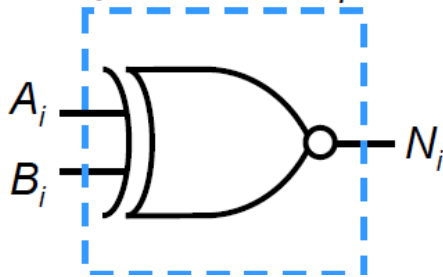
 1-bit Comparator Block

 Equality Block

4-bit Equality Comparator

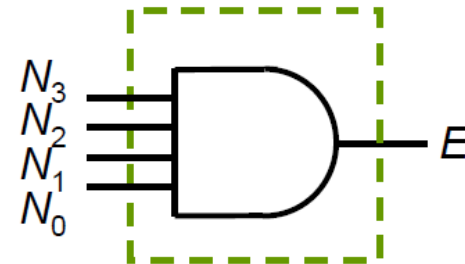
■ 1-bit Comparator Block

- 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 $N_i = A_i \oplus B_i$



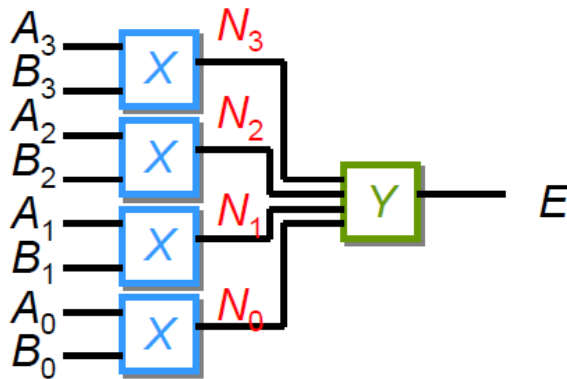
■ Equality Block

- 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
- $E = N_3 \cdot N_2 \cdot N_1 \cdot N_0$

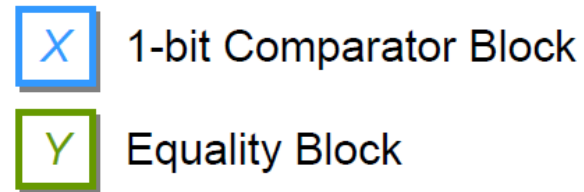


4-bit Equality Comparator

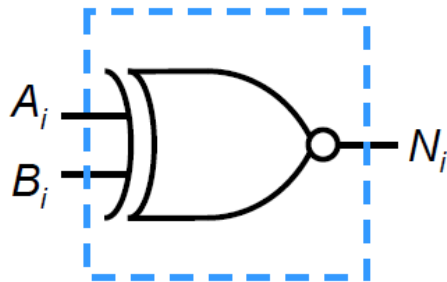
- Final logic diagram



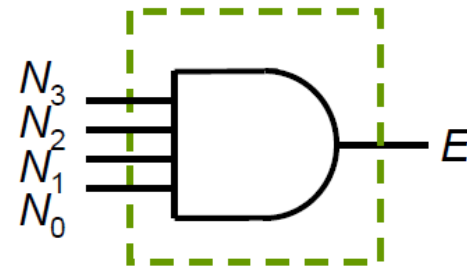
Functional Blocks:



Block X:



Block Y:



Summary of 5.1

- 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

1-bit Adder

■ Formulation:

Inputs		Outputs	
X	Y	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

■ Optimization:

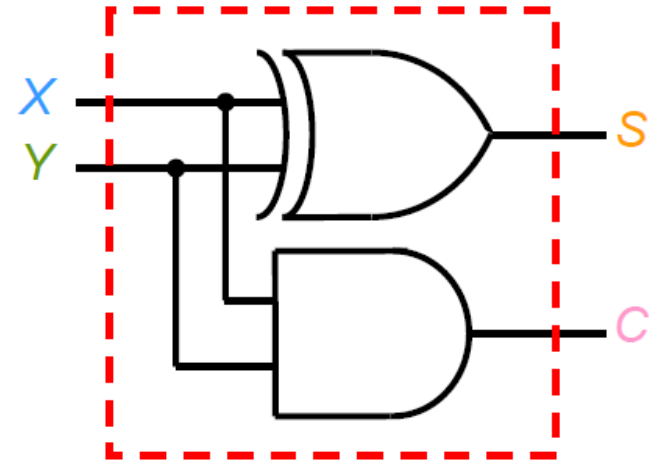
■ $C(X, Y) = m_3$

■ $= X \cdot Y$

■ $S(X, Y) = \sum m(1, 2)$

■ $= X \oplus Y$ (i.e. $X'Y + XY'$)

■ Final logic diagram:



The outputs are
S (sum bit), and
C (carry-out bit)

- As known as **half adder (HA)**

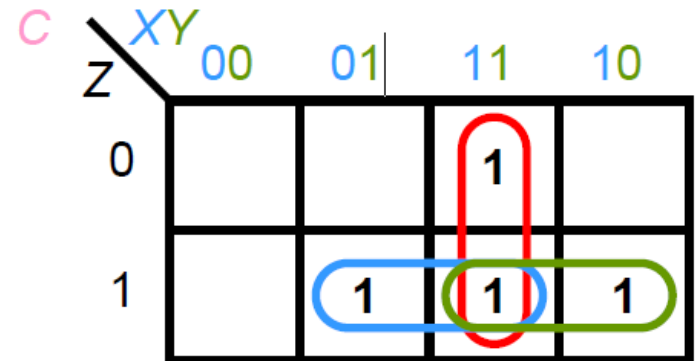
Full Adder

■ Formulation:

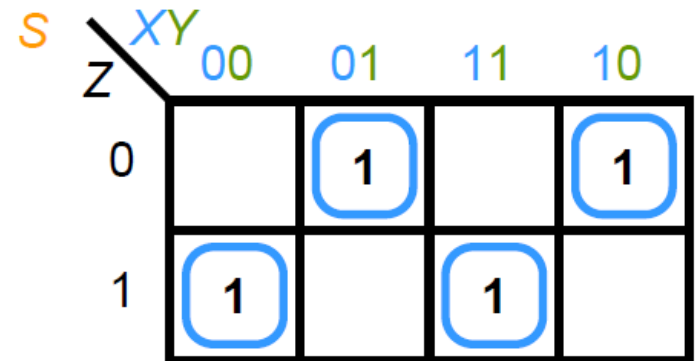
Inputs			Outputs	
X	Y	Z	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

■ Optimization:

■ Use K-map



$$\begin{aligned}
 C &= XY + XZ + YZ \\
 &= XY + Z(XY' + X'Y + XY) \\
 &= XY(1 + Z) + Z(XY' + X'Y) \\
 &= XY + Z(X \oplus Y)
 \end{aligned}$$

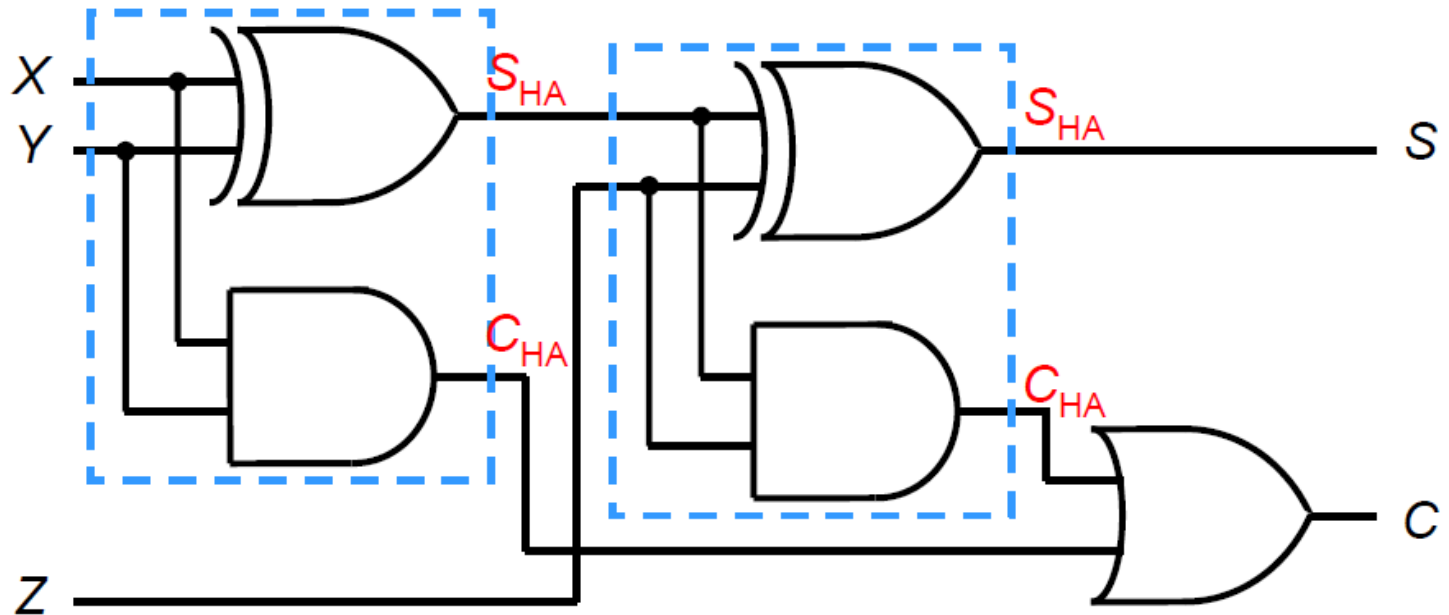


$$\begin{aligned}
 S &= X'Y'Z + X'YZ' + XY'Z' + XYZ \\
 &= X \oplus Y \oplus Z
 \end{aligned}$$

Full Adder

- Final Logic Diagram:

- $S = X \oplus Y \oplus Z$, $C = XY + Z(X \oplus Y)$

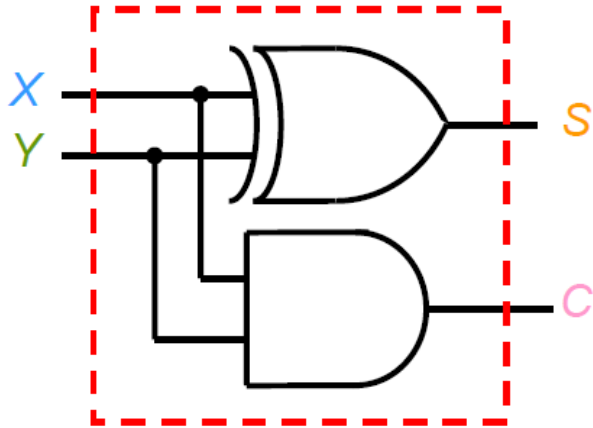


- Do you notice anything?

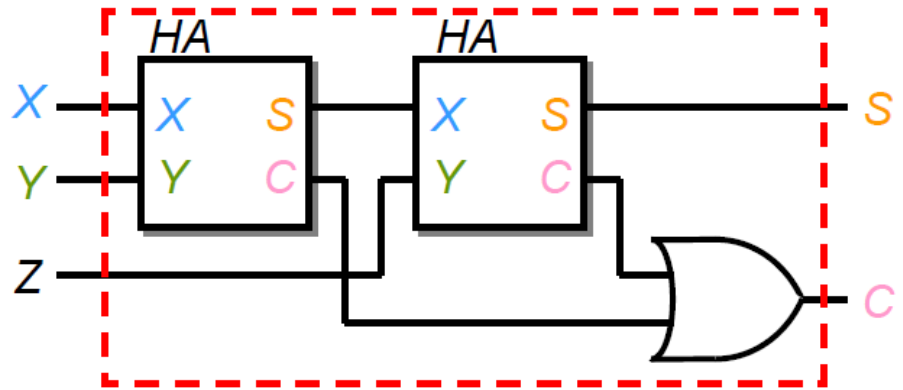
- Actually build up by two half adders

Half Adder and Full Adder

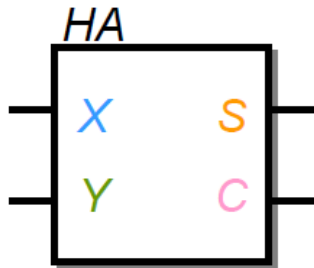
■ Logic circuit diagram



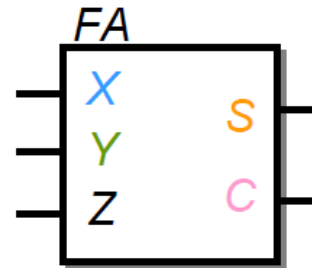
■ Logic circuit diagram



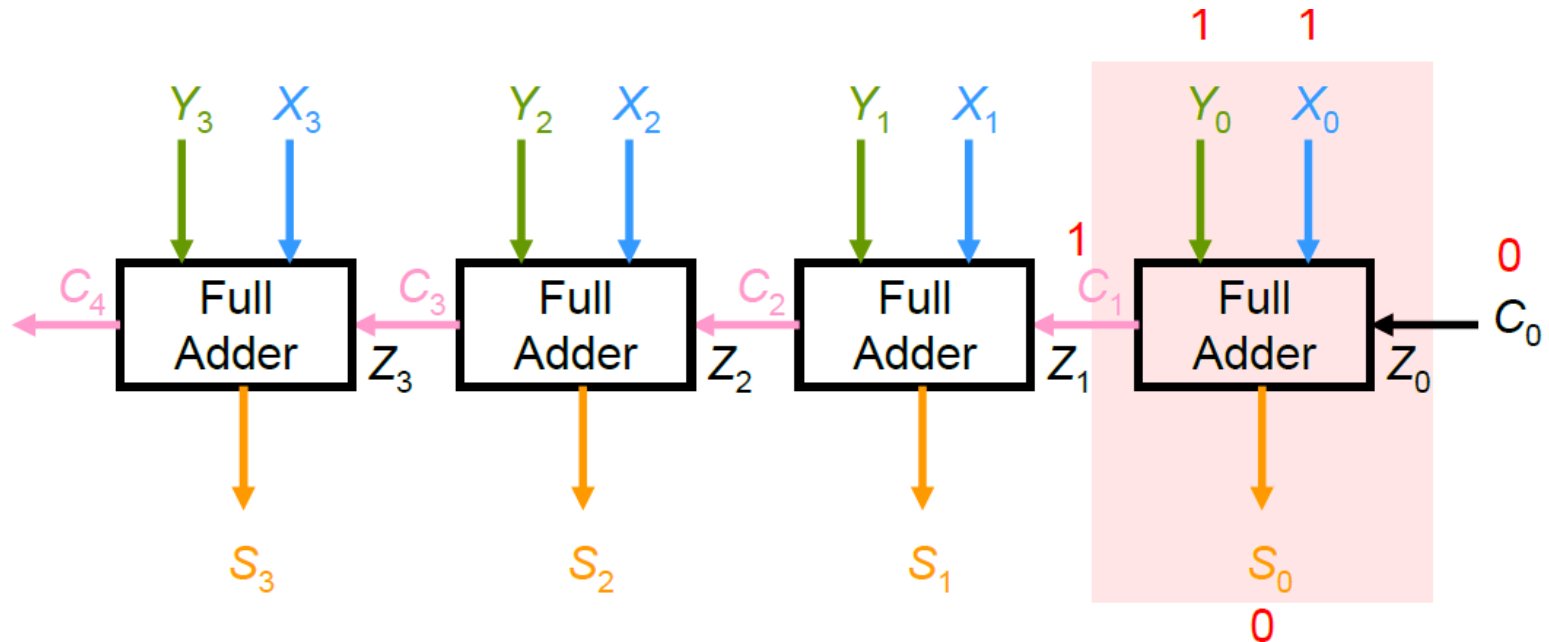
■ Symbol



■ Symbol



Ripple Carry Adder



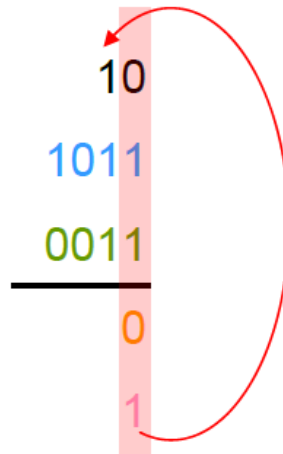
Carry-in (Z_i)

Augend (X_i)

Addend (Y_i)

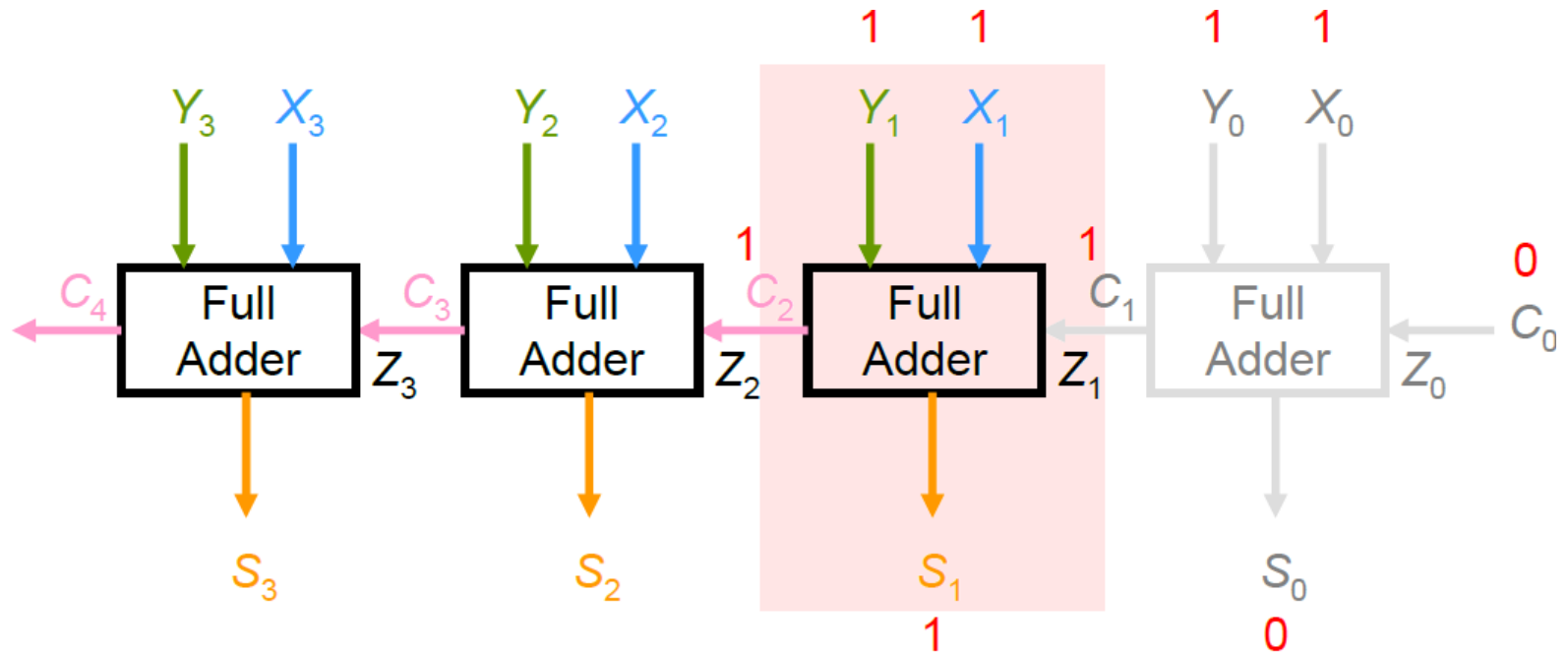
Sum (S_i)

Carry-out (C_i)



Carry out connect to the carry input of next full adder

Ripple Carry Adder



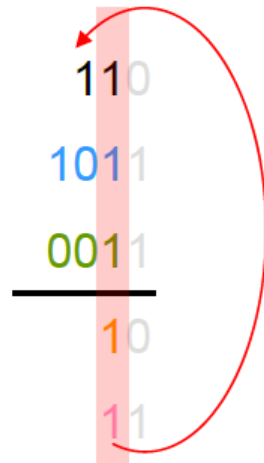
Carry-in (Z_i)

Augend (X_i)

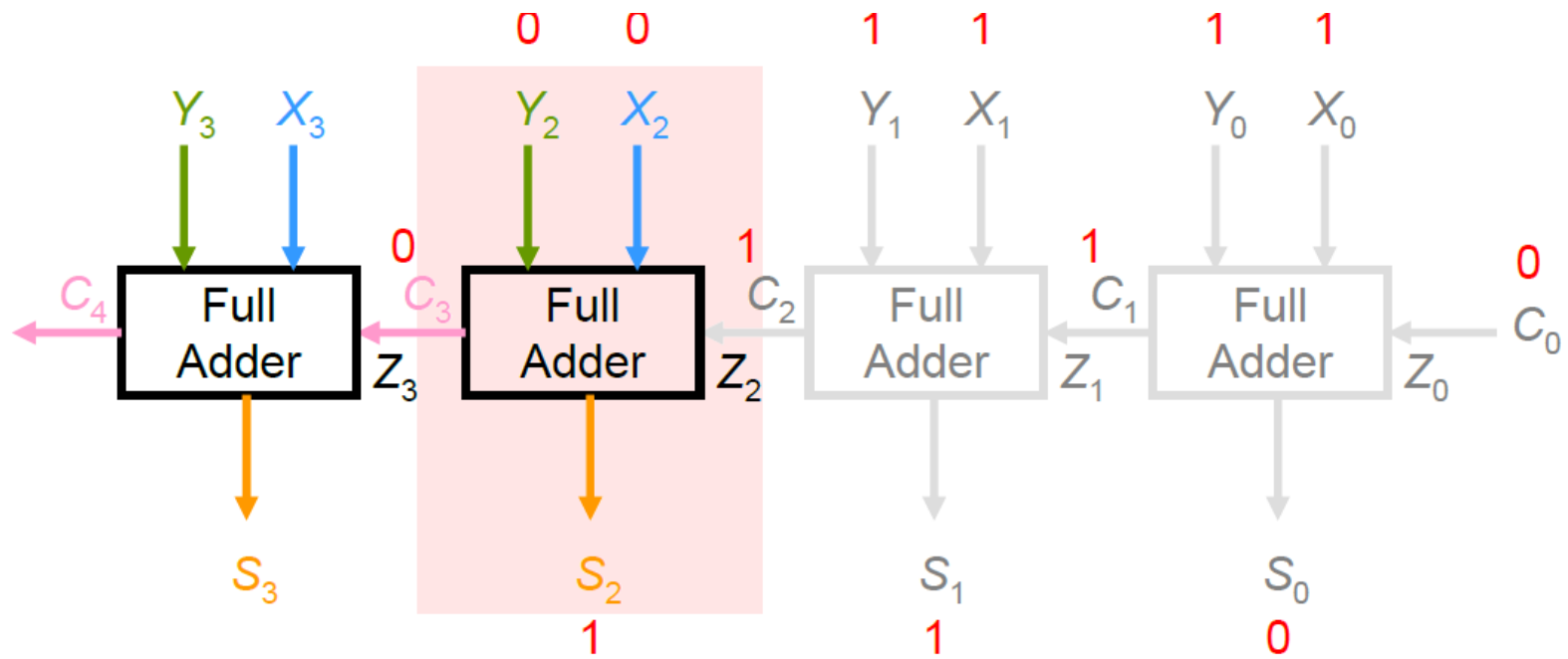
Addend (Y_i)

Sum (S_i)

Carry-out (C_i)



Ripple Carry Adder



Carry-in (Z_i)

Augend (X_i)

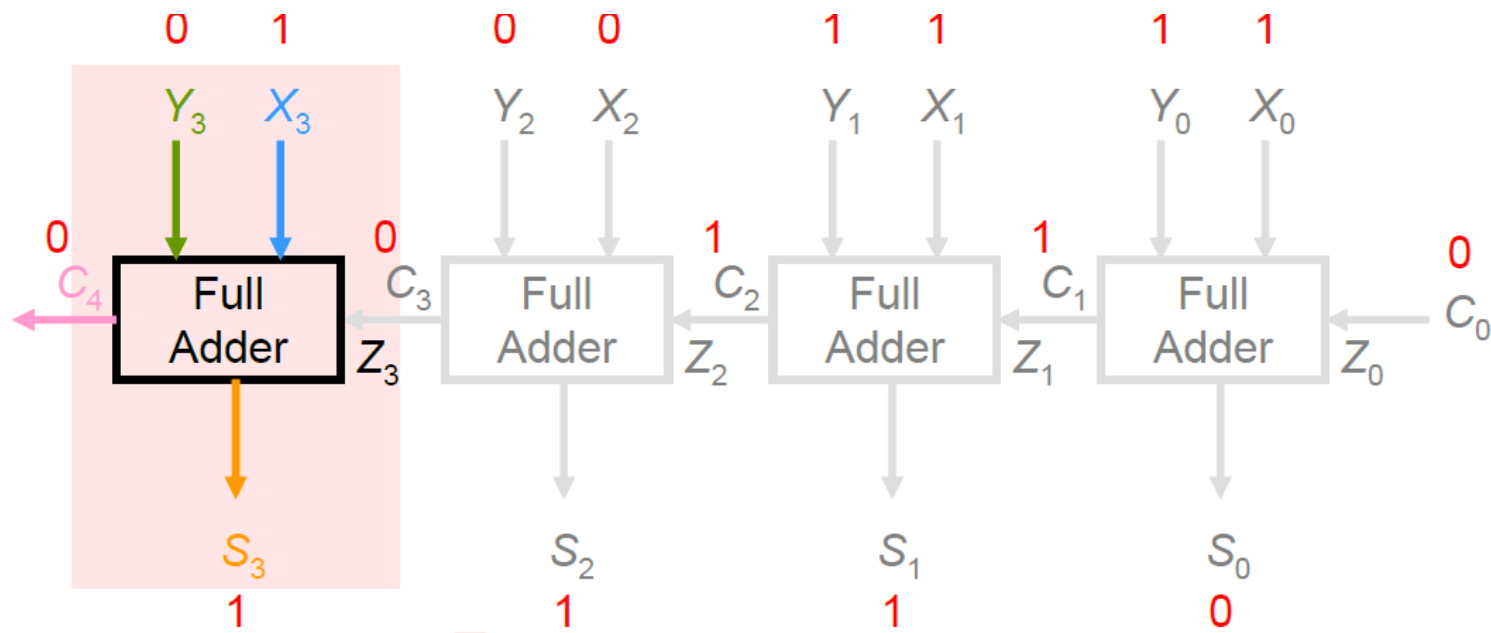
Addend (Y_i)

Sum (S_i)

Carry-out (C_i)

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	0	
Carry-out (C_i)	0	1	1	

Ripple Carry Adder



Carry-in (Z_i)

Augend (X_i)

Addend (Y_i)

Sum (S_i)

Carry-out (C_i)

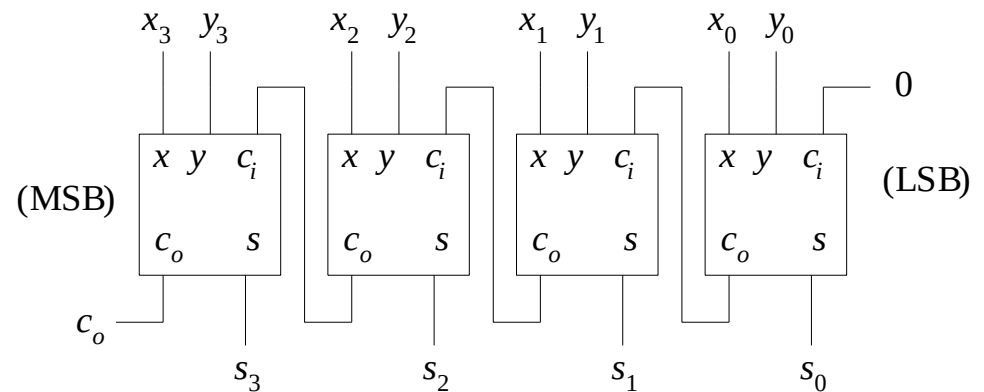
0110

1011

0011

1110

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

$$\begin{array}{r}
 \text{Minuend (X)} \quad 1 \\
 \text{Subtrahend (Y) } -) \quad 0 \\
 \hline
 \text{Difference (D)} \quad 1
 \end{array}$$

HS: performs simple two single-bit subtraction

$$\begin{array}{r}
 \text{Borrow-in (Z}_i\text{)} \quad 0 \\
 \text{Minuend (X}_i\text{)} \quad 1 \\
 \text{Subtrahend (Y}_i\text{) } -) \quad 1 \\
 \hline
 \text{Difference (D}_i\text{)} \quad 0
 \end{array}$$

FS: performs simple three single-bit subtraction

$$\begin{array}{r}
 \text{Borrow-in (Z}_i\text{)} \quad ? \ ? \ ? \ ? \\
 \text{Minuend (X}_i\text{)} \quad 1 \ 0 \ 1 \ 1 \\
 \text{Subtrahend (Y}_i\text{) } -) \quad 0 \ 0 \ 1 \ 1 \\
 \hline
 \text{Difference (D}_i\text{)} \quad ? \ ? \ ? \ ?
 \end{array}$$

RCS: performs real two n -bit subtraction

Half Subtractor

■ Formulation:

Inputs		Outputs	
X	Y	B	D
0	0	0	0
0	1	1	1
1	0	0	1
1	1	0	0

■ Optimization:

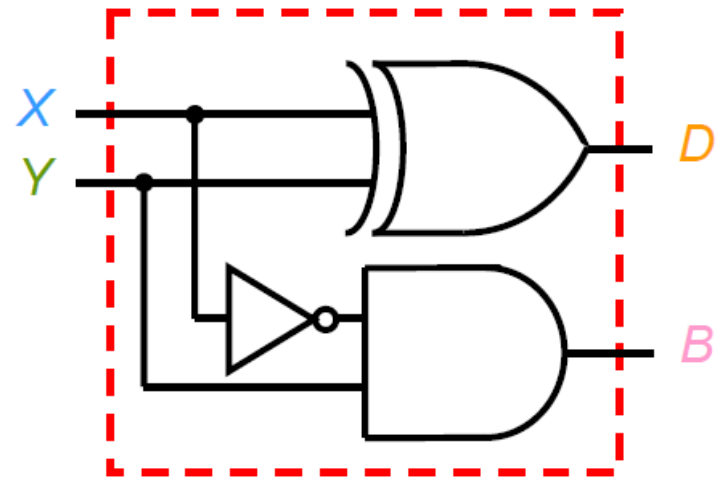
■ $B(X, Y) = m_1$

■ $= X' \cdot Y$

■ $D(X, Y) = \sum m(1, 2)$

■ $= X \oplus Y$ (i.e. $X'Y + XY'$)

■ Final logic diagram:



The outputs are
 D (difference bit) and B
(borrow out bit)

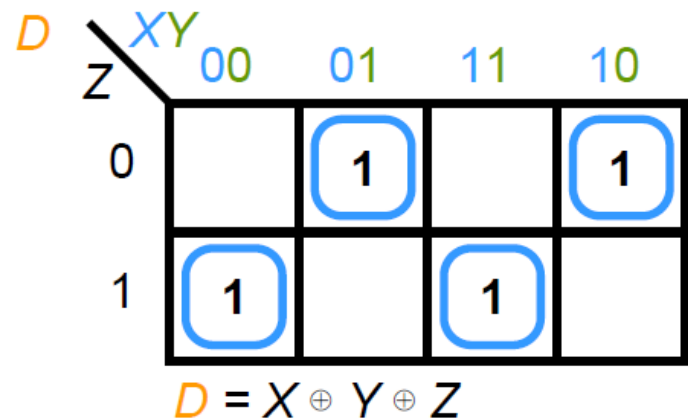
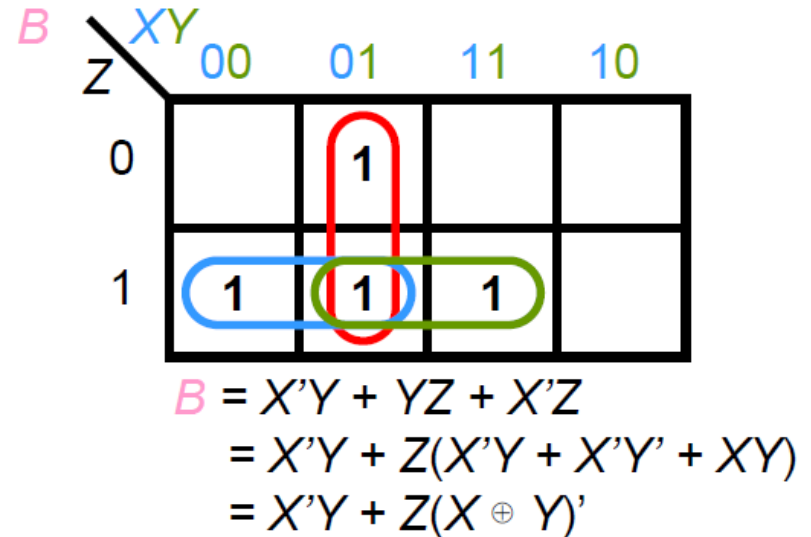
Full Subtractor

■ Formulation:

Inputs			Outputs	
X	Y	Z	B	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

■ Optimization:

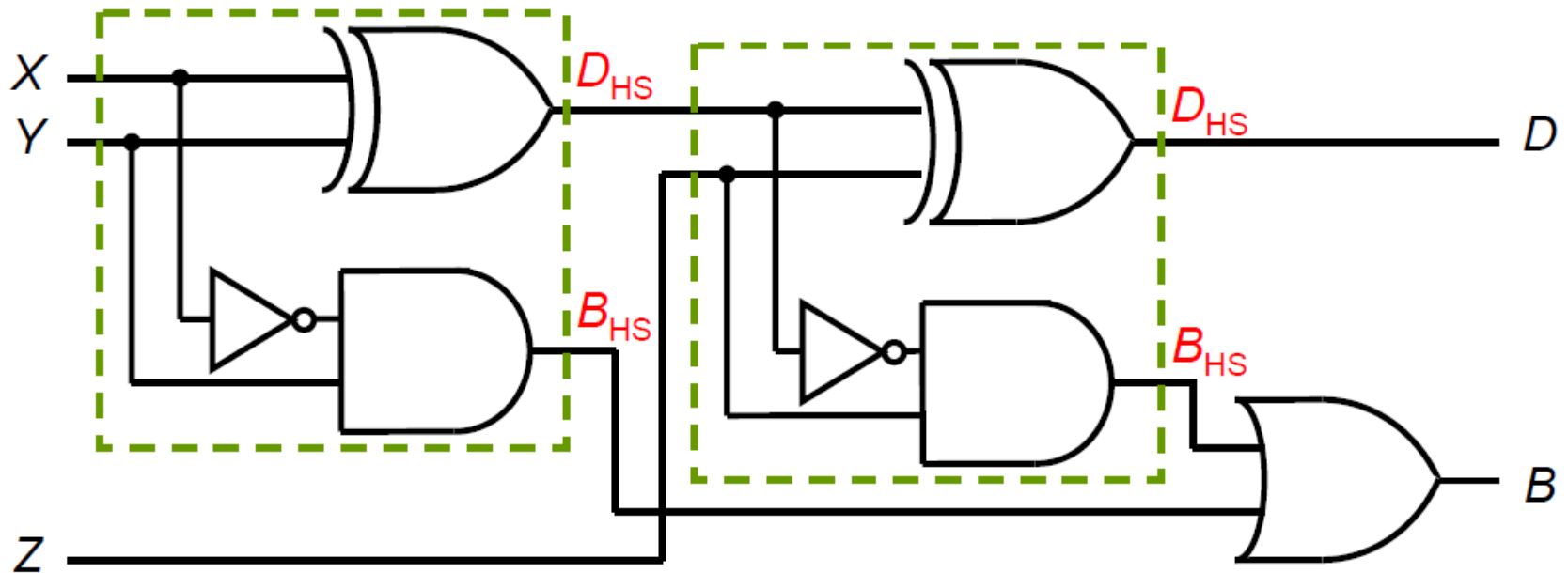
■ Use K-map



Full Subtractor

■ Final Logic Diagram:

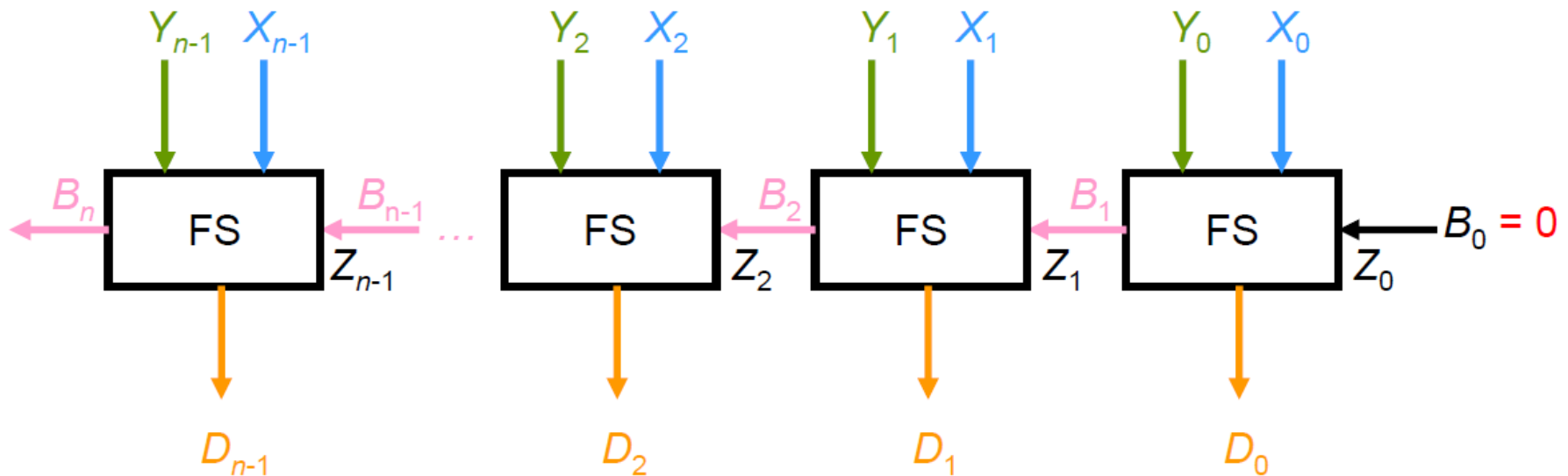
■ $D = X \oplus Y \oplus Z$, $B = X'Y + Z(X \oplus Y)$



■ Like FA, FS is built up by two HSs

Ripple Carry Subtractor

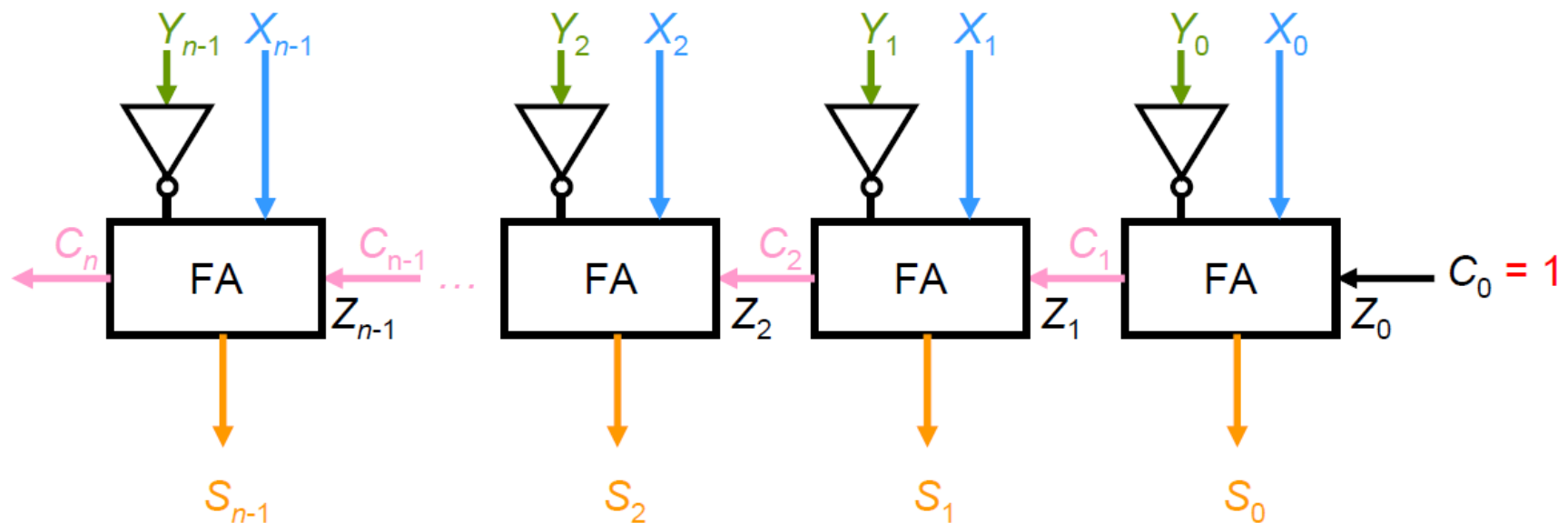
- In addition to ripple carry adder, there is ripple carry subtractor
- Connect n FSs in cascade



- May we reuse FA to implement it? Yes.

Ripple Carry Subtractor (Another version)

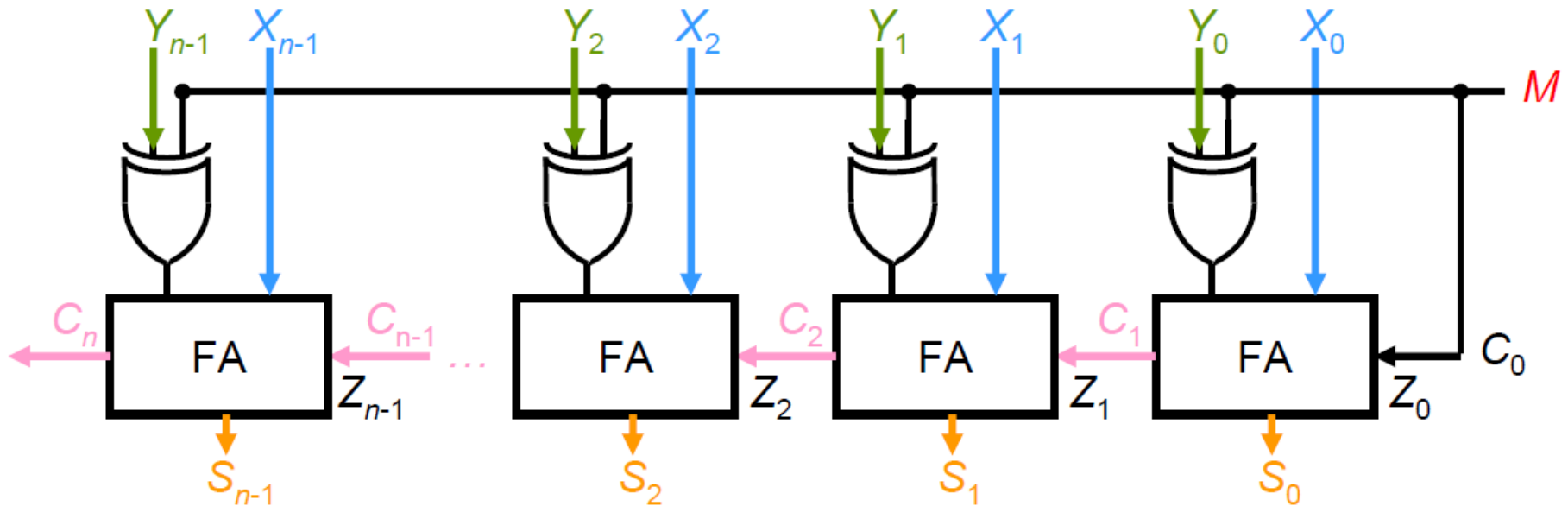
- $X - Y = X + (-Y)$
- Invert input Y and set carry input to 1 (why?)



- Can we combine adder and subtractor?

n -bit RC Adder/Subtractor

- 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 ($M = 1$), $C_0 = 1$, $Y_i = Y_i \oplus 1 = Y_i'$

Delay Problem of RCA

- The carry output of each full-adder stage is connected to the carry input of the next higher stage
- A time delay thus occurs because the sum and the carry output of each stage cannot be produced until the input carry appears
- **Assuming** that the delay for generating the carry output is **m nsec**
- For a N -bit adder, the total delay will be up to **Nm nsec**
- Serious delay problem if N is a large number
- **Solution: Calculate the carry bits beforehand, then construct carry-look-ahead adder**

Calculate Carry Bits Beforehand

- $C_{i+1} = X_i Y_i + C_i (X_i \oplus Y_i)$
 $= G_i + C_i P_i$

- Define

- G_i as **generate bit**: $X_i Y_i$

- P_i as **propagate bit**: $X_i + Y_i$ or $X_i \oplus Y_i$

- This produce a recursive definition

- $C_1 = G_0 + P_0 C_0$

- $C_2 = G_1 + P_1 C_1$

- $= G_1 + P_1 (G_0 + P_0 C_0)$

- $= G_1 + P_1 G_0 + P_1 P_0 C_0$ (distributivity)

- $C_3 = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_0$

General Form of the Carry Bits

- $C_{i+1} = G_i + P_i G_{i-1} + P_i P_{i-1} G_{i-2} + \dots + P_i P_{i-1} \dots P_1 G_0 + P_i P_{i-1} \dots P_1 P_0 C_0$

- C_{i+1} is now independent of C_i !

- Only depends on G_i , P_i and C_0

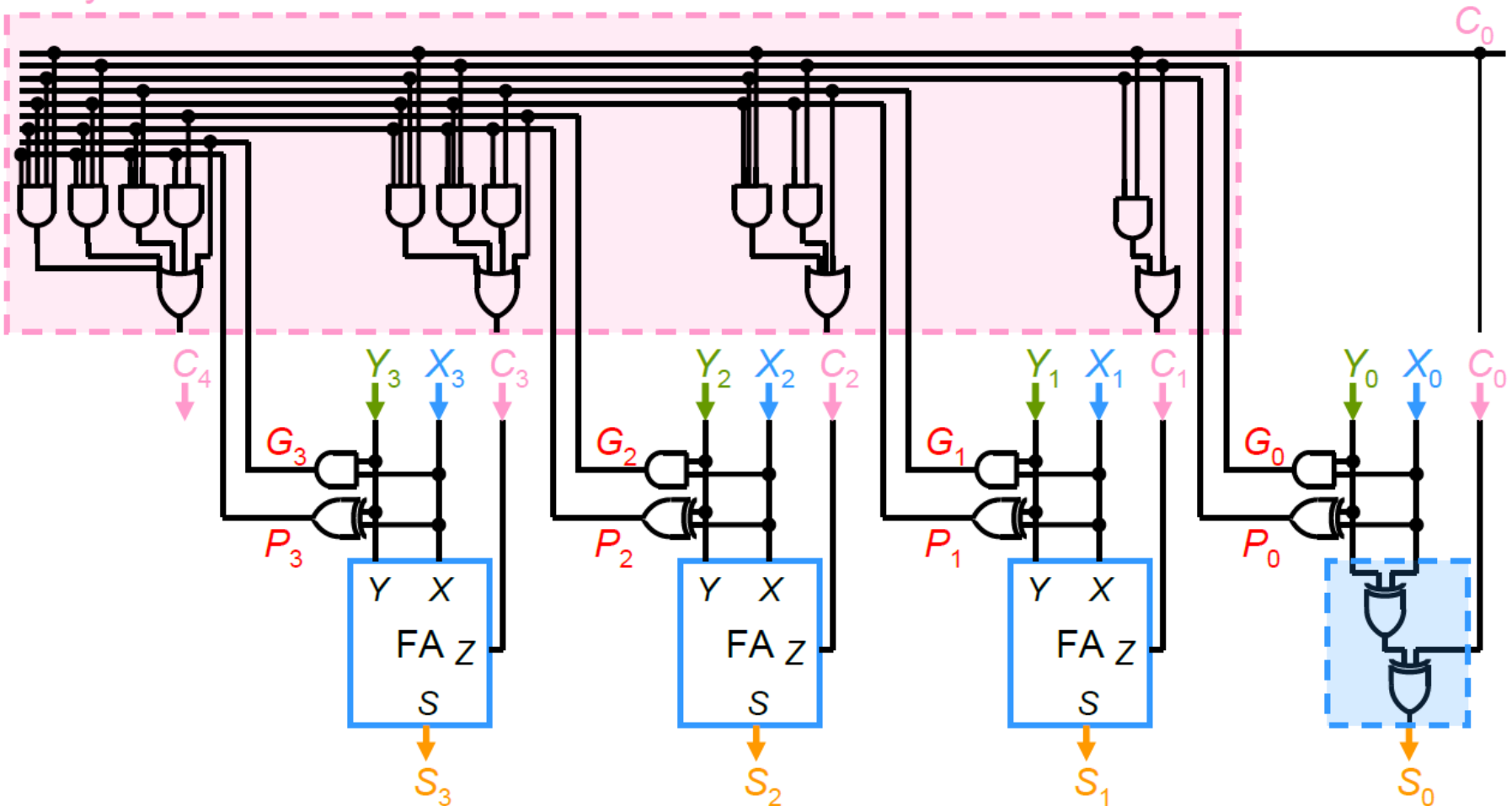
- i.e. depends on X_i and Y_i only

- The carry bits can now be computed independently

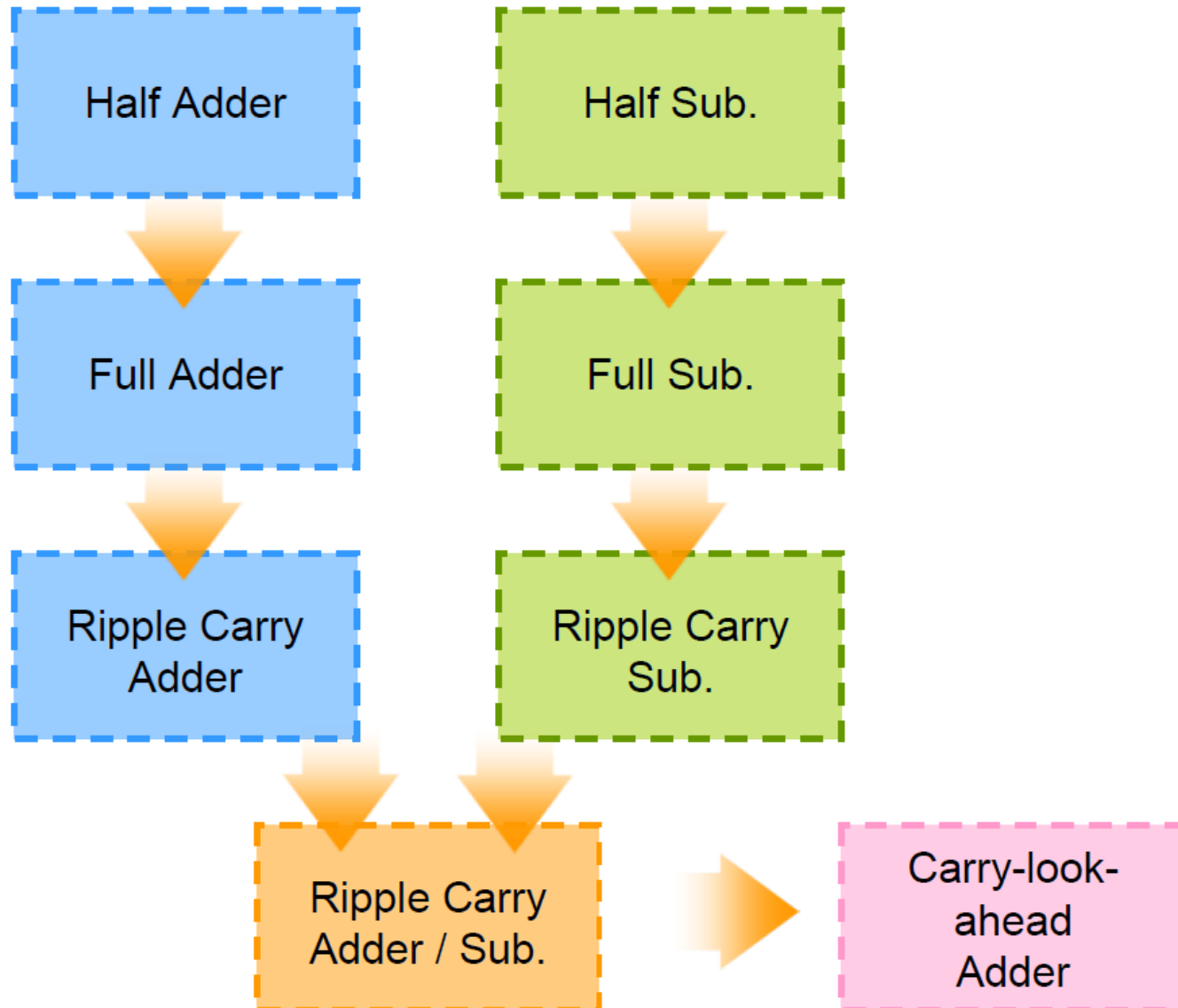
- Based on G_i , P_i and C_0

4-bit Carry-look-ahead Adder

Carry-look-ahead Generator



Summary of 5.2

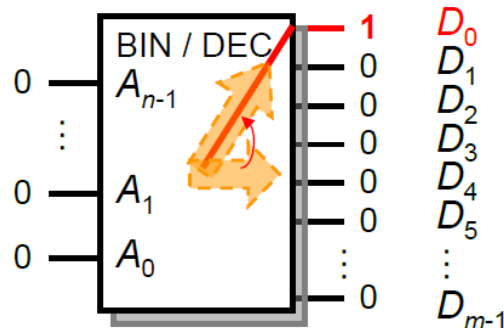


5.3 Logical functional blocks (Decoder)

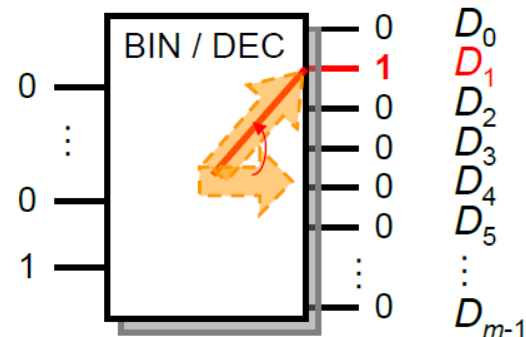
Decoder

- A decoder is a combinational circuit with n -input and m -output ($0 < n \leq m \leq 2n$, but usually $m = 2^n$)
- A very important functional blocks as it can be incorporated into many of the other functions

e.g. Input $A = (0 \dots 00)$



e.g. Input $A = (0 \dots 01)$



1-to-2-Line decoder

■ Specification:

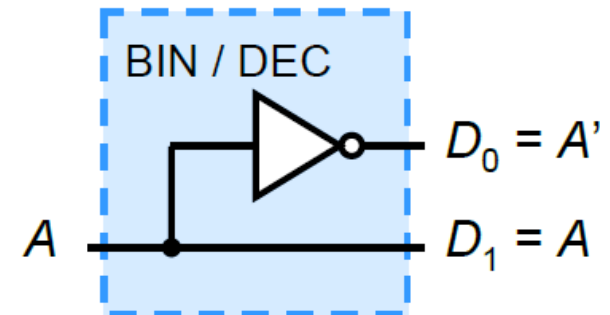
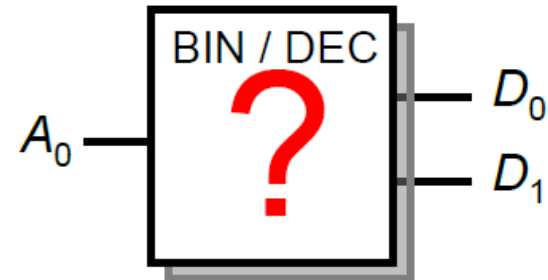
- Input: 1-bit (A_0)
- Outputs: 2-bit (D_0, D_1)

■ Formulation:

Input	Outputs	
A	D_0	D_1
0	1	0
1	0	1

■ Optimization:

- $D_0 = m_0 = A'$
- $D_1 = m_1 = A$

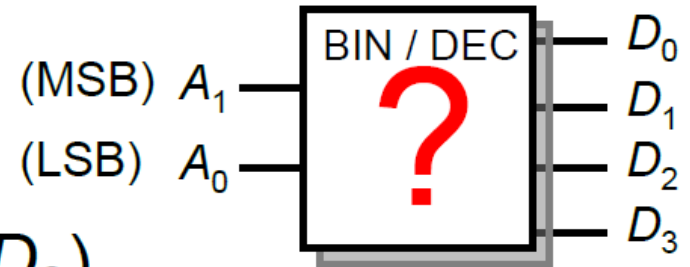


2-to-4-Line Decoder

■ Specification:

■ Inputs: 2-bit (A_1, A_0)

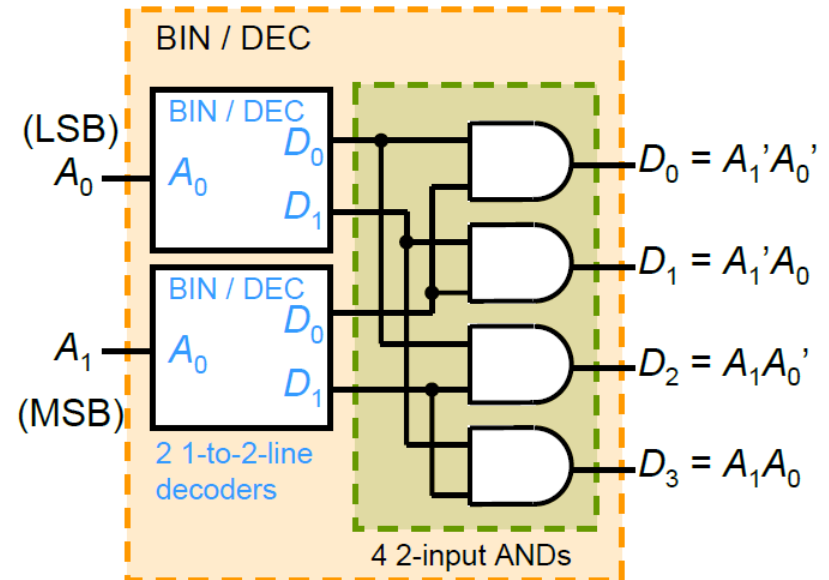
■ Outputs: 4-bit (D_0, D_1, D_2, D_3)



■ Formulation:

Inputs		Outputs			
A_1	A_0	D_0	D_1	D_2	D_3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

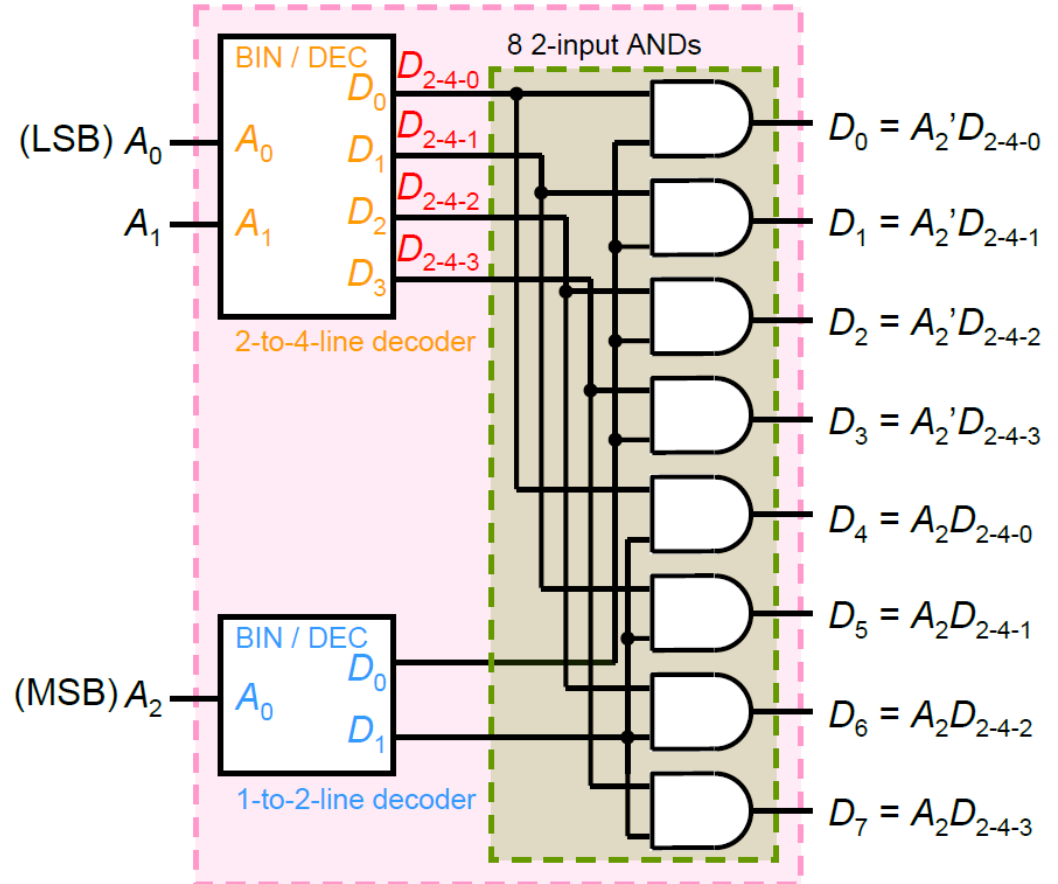
■ Final logic diagram:



3-to-8-Line Decoder

Inputs			Outputs							
A_2	A_1	A_0	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

- Build the simplest block:
 - 1-to-2-line decoder
 - 2-to-4-line decoder
 - 3-to-8-line decoder
 - n -to- 2^n -line decoder



Decoder with Enabling

- Specification:

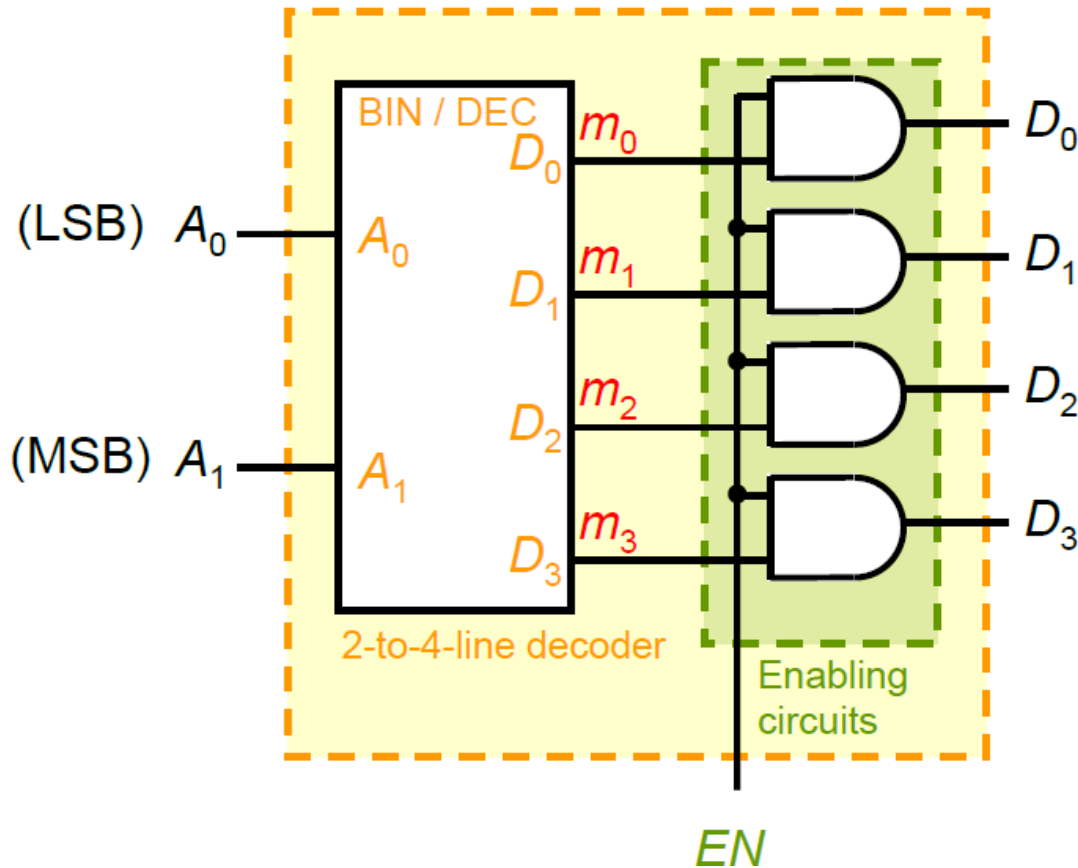
- A 2-to-4-line decoder with enable input

Inputs			Outputs			
EN	A_1	A_0	D_0	D_1	D_2	D_3
0	x	x	0	0	0	0
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1

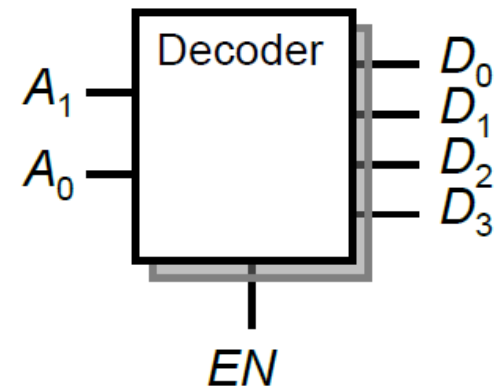
- If EN to 0 (**disabled**), all outputs are 0
 - If EN to 1 (**enabled**), the outputs are same as normal decoder (i.e. only the corresponding $D_i = 1$, and all others are 0)
 - **Note: the Xs in the above truth table are don't care inputs**

Decoder with Enabling

$$D_i = EN \cdot m_i$$



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



5.3 Logical functional blocks (Encoder)

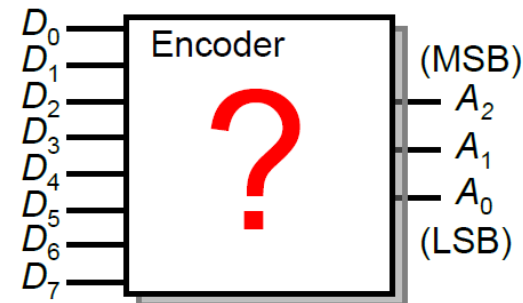
- An encoder is a functional block that performs the inverse operation of a decoder
- m inputs and n outputs
- $0 < n \leq m \leq 2n$, but usually $m = 2^n$

Example of Octal-to-Binary Encoder

■ $m = 8, n = 3$

■ i.e. inputs: 8, outputs: 3

■ Formulation:



Inputs								Outputs		
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	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
All other invalid inputs								x	x	x

Octal-to-binary Encoder

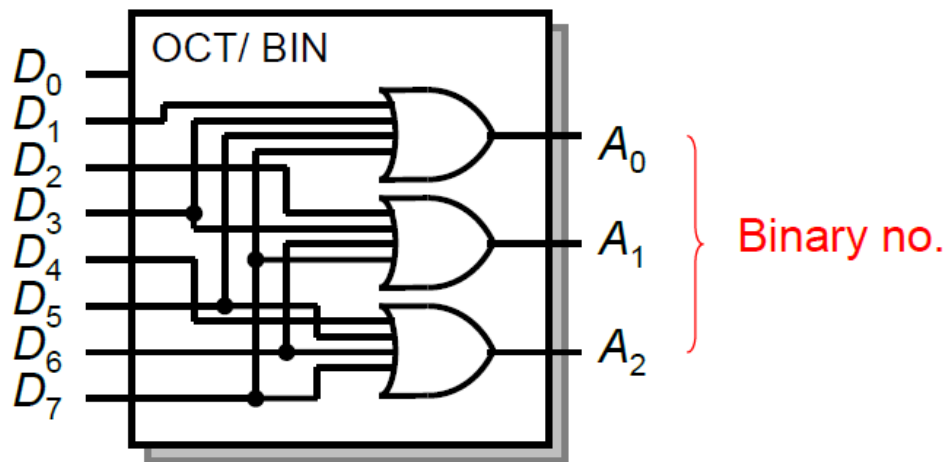
■ Optimization:

■ $A_0(D_0, D_1, D_2, D_3, D_4, D_5, D_6, D_7) = D_1 + D_3 + D_5 + D_7$

■ $A_1(D_0, D_1, D_2, D_3, D_4, D_5, D_6, D_7) = D_2 + D_3 + D_6 + D_7$

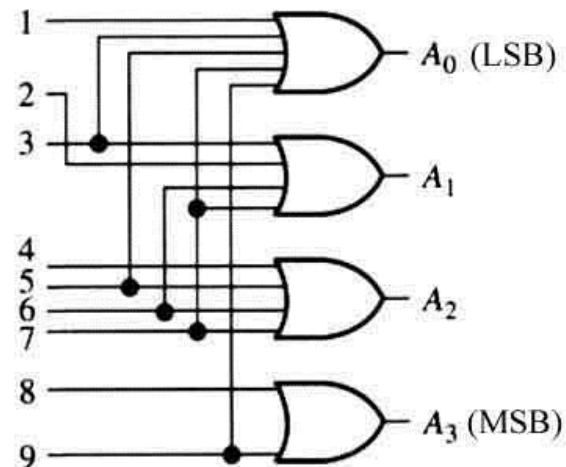
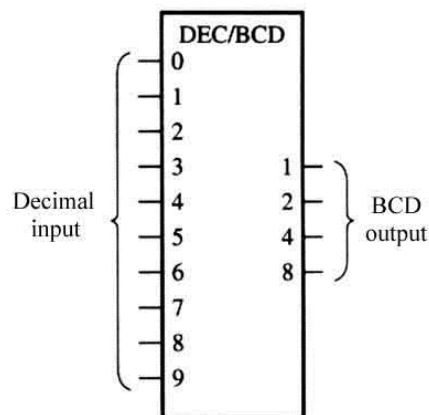
■ $A_2(D_0, D_1, D_2, D_3, D_4, D_5, D_6, D_7) = D_4 + D_5 + D_6 + D_7$

■ Final logic diagram:



Decimal-to-BCD encoder

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



Limitation of Encoders

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

Priority Encoders

■ Specification:

- Design a 4-input priority encoder
- Inputs with higher subscript numbers has higher priority

■ Formulation:

Inputs				Outputs		
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

Introduce one more output V

V stands for valid output

If all inputs are 0 (i.e. invalid input), V is 0. Otherwise V is 1

If D_3 and D_2 are both 1, ignore D_2 (as if D_2 is 0)

4-Input Priority Encoder

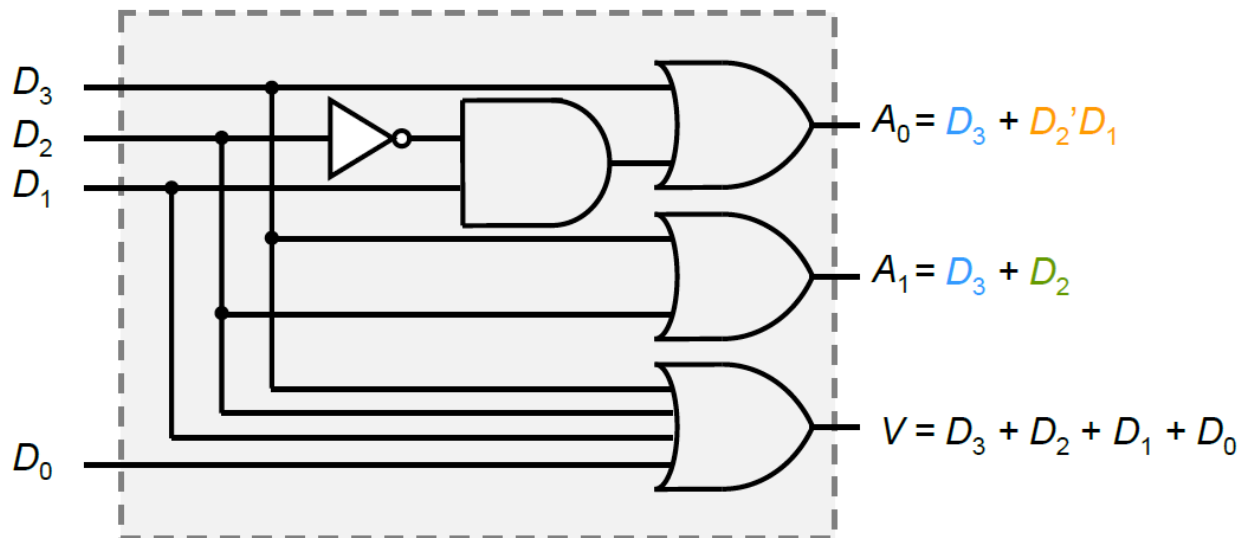
		A_1				A_0				V			
$D_1 D_0$	$D_3 D_2$	00	01	11	10	00	01	11	10	00	01	11	10
	00	x	1	1	1	x		1	1	0	1	1	1
01		1	1	1	1			1	1	1	1	1	1
11		1	1	1	1	1		1	1	1	1	1	1
10		1	1	1	1	1		1	1	1	1	1	1

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

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

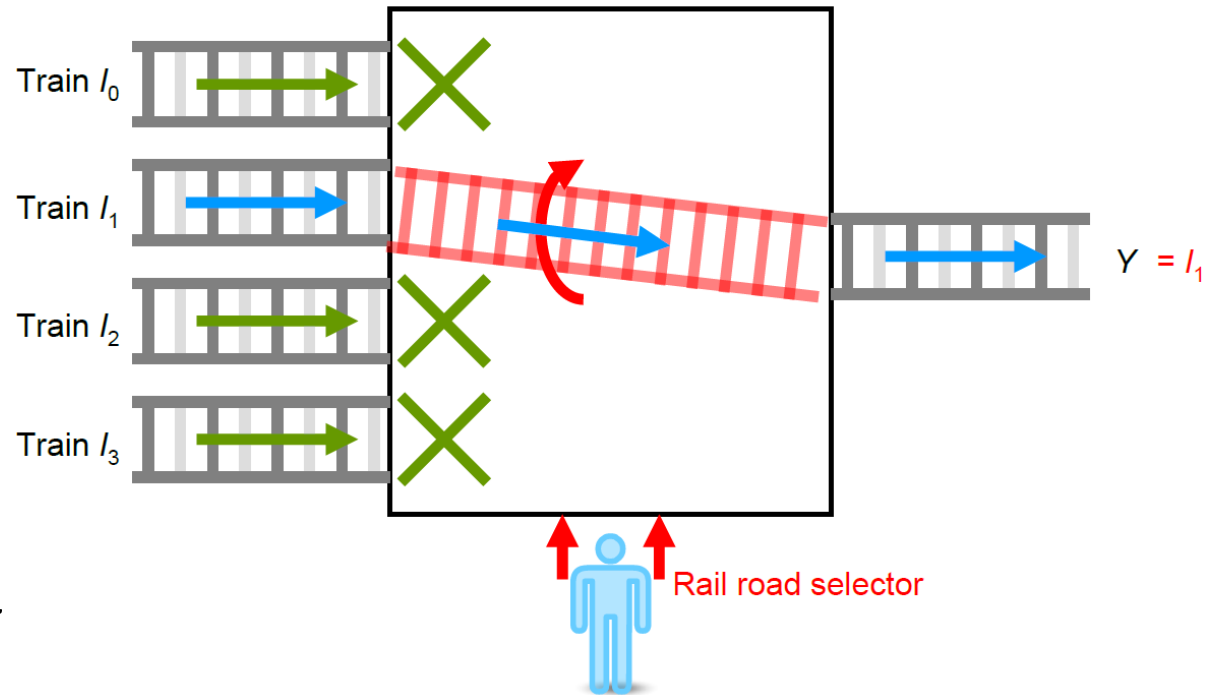
$$V' = D_3' D_2' D_1' D_0'$$

$$V = D_3 + D_2 + D_1 + D_0$$



5.3 Logical functional blocks (Multiplexer -- MUX)

- Combinational circuit performs selection
- To appear I_s on a **single output** (Y) from **many inputs** (I_i)
- A set of **selection input variables** $(S)_{10} = (S_{n-1} \dots S_1 S_0)_2$



2-to-1-line MUX

■ Specification:

■ $m = 2$

■ $n = \log_2 m = 1$

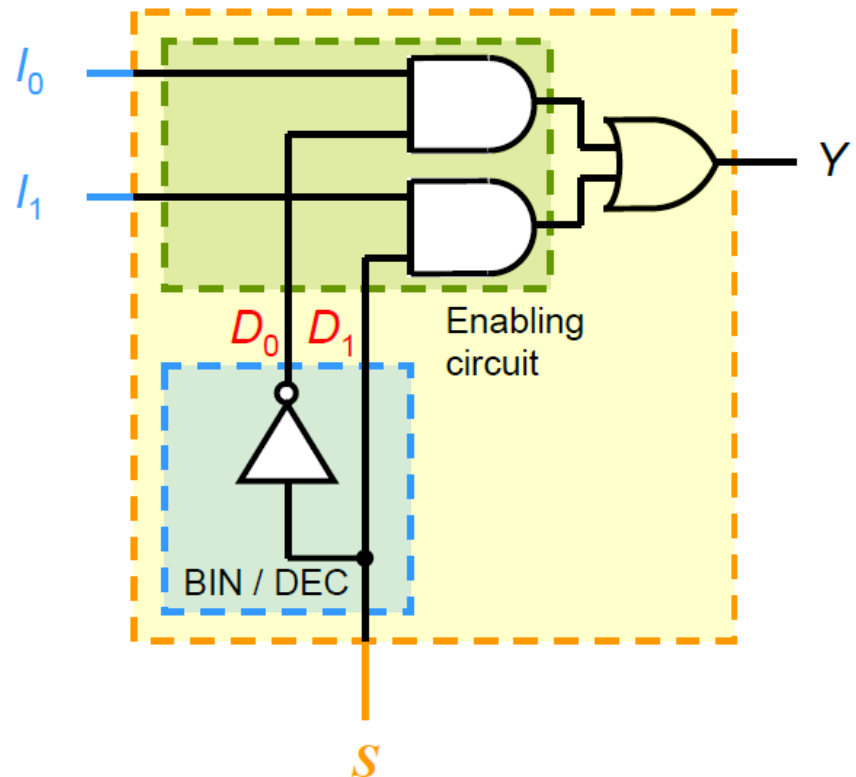
■ Formulation:

Inputs			Output
I_0	I_1	S_0	Y
x	x	0	I_0
x	x	1	I_1

■ Optimization:

■ $Y(I_0, I_1, S_0) = S_0' I_0 + S_0 I_1$

■ Final logic diagram



4-to-1-line MUX

■ Specification:

■ $m = 4$

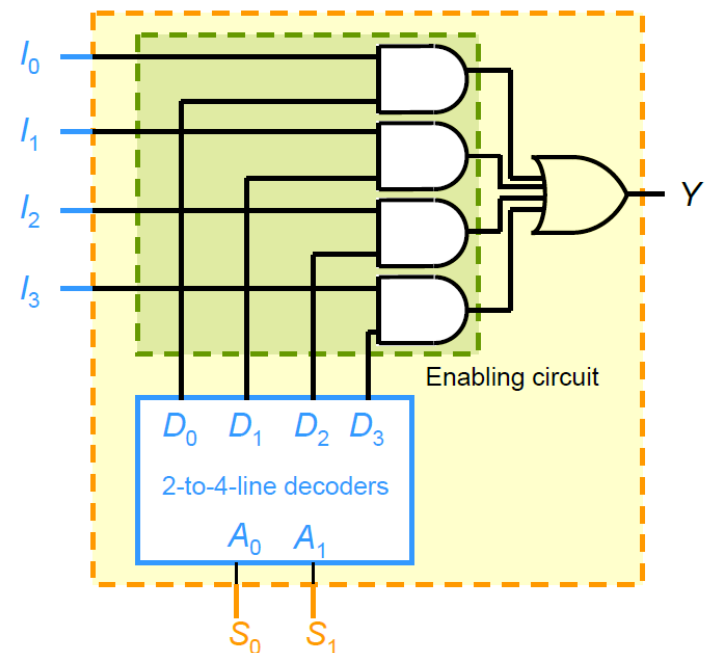
■ $n = \log_2 m = 2$

■ 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

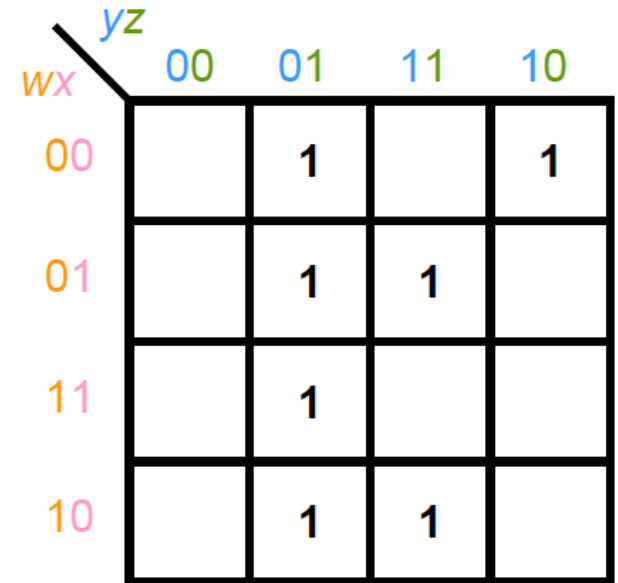
■ Optimization:

■ $Y(I_0, I_1, I_2, I_3, S_1, S_0) =$
 $S_1' S_0' I_0 + S_1' S_0 I_1 +$
 $S_1 S_0' I_2 + S_1 S_0 I_3$



4-to-1-line MUX (Example)

- Realize the function $f(w, x, y, z) = \sum m(1, 2, 5, 7, 9, 11, 13)$ using 4-to-1-line MUX
- Step 1) Plot the K-map
 - *No need to group the 1s!*
 - *Note the convention of var.*
- Step 2) Since $n = 2$,
 - Pick w, x as selection var.
 - w as S_1 , x as S_0
 - Remaining vars. are y, z



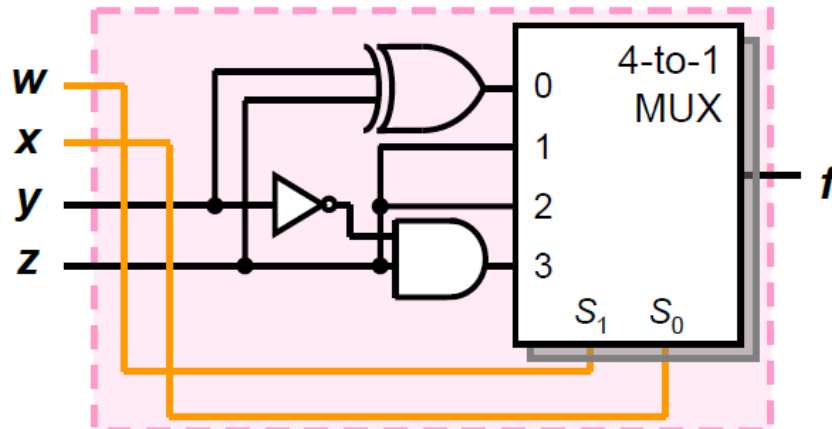
$wx \backslash yz$	00	01	11	10
00		1		1
01		1	1	
11		1		
10		1	1	

4-to-1-line MUX (Example)

■ Step 3)

	yz	00	01	11	10	
wx						
00			1		1	} When $wx = 00$, $f = y'z + yz' = y \oplus z$
01			1	1		} When $wx = 01$, $f = z$
11			1			} When $wx = 11$, $f = y'z$
10			1	1		} When $wx = 10$, $f = z$

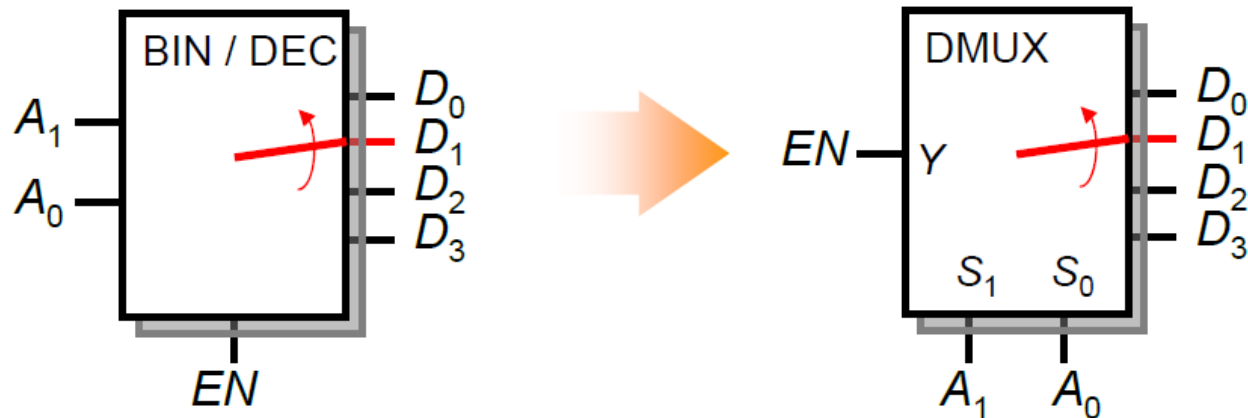
■ Step 4)



Exercise: instead of using 4-to-1-line MUX, can you implement this using 8-to-1-line MUX or 2-to-1-line MUX?

Demultiplexer (DMUX)

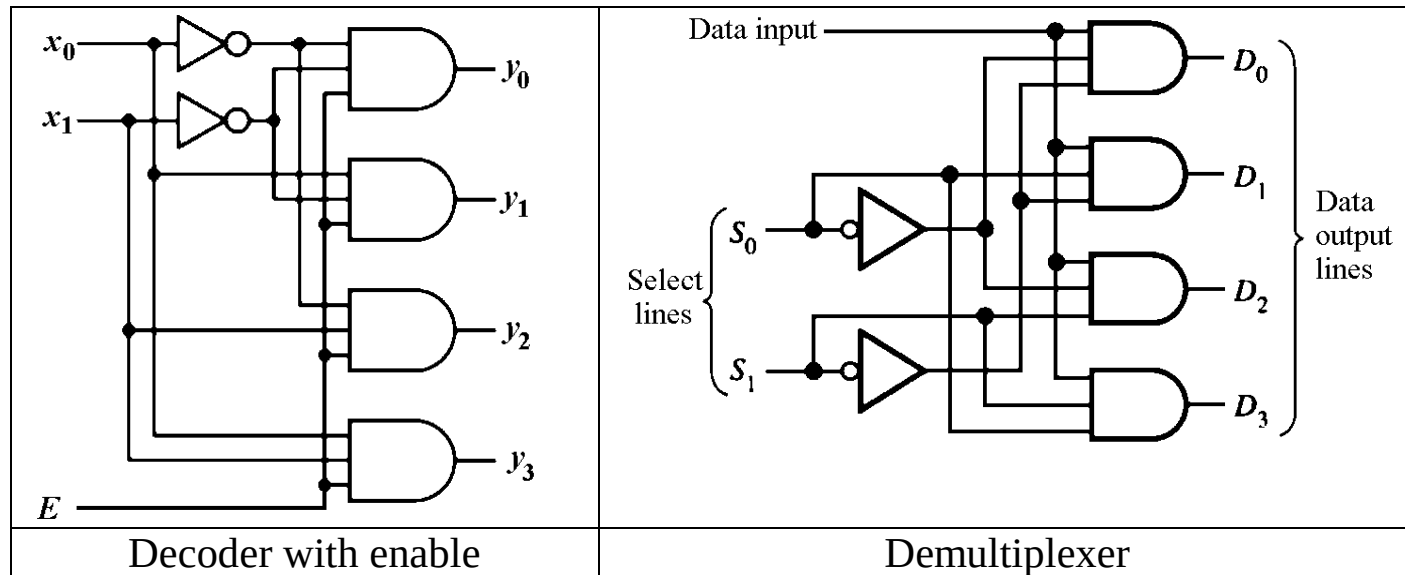
- Remember the decoder with enabling?



- The decoder can perform demultiplexing if we take EN as the input line, A_i (input lines of decoder) as the selection inputs

Demultiplexer (DMUX)

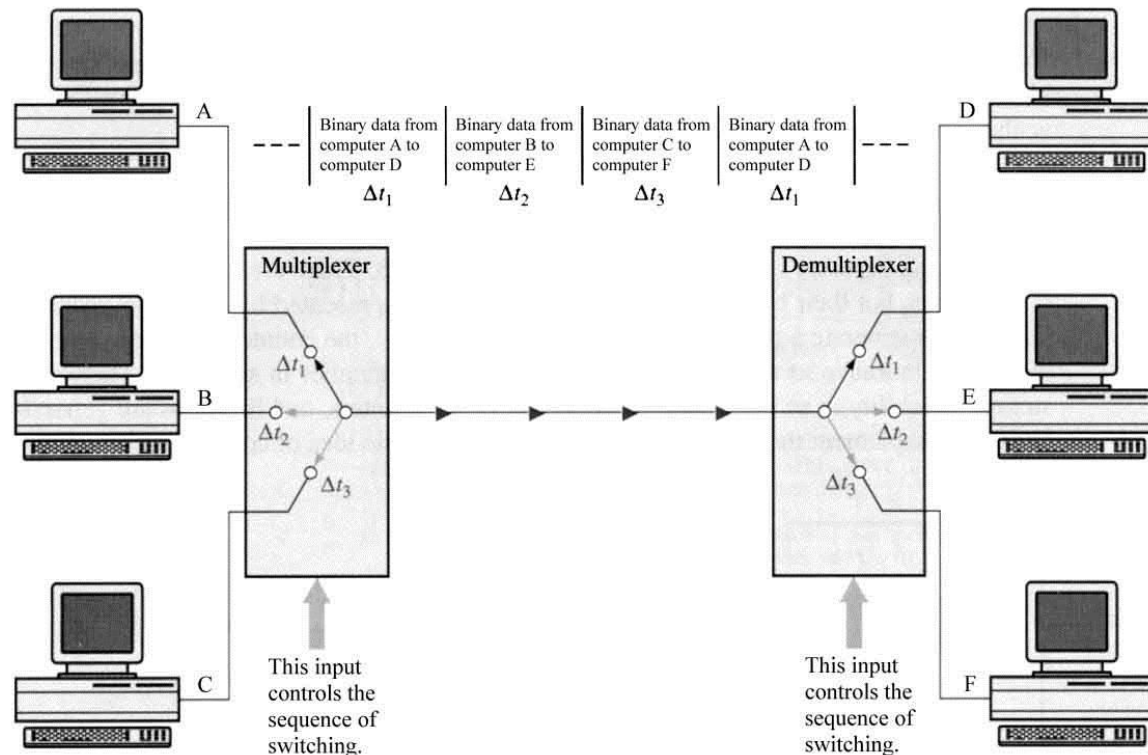
A demultiplexer (DMUX) basically reverses the multiplexing function. A DMUX is a circuit that receives information on a single line and transmits this information on one of 2^n possible output lines. The selection of a specific output line is controlled by the bit pattern of n select lines. For this reason, the demultiplexer is also known as a data distributor.



Decoder can function as demultiplexer if the E line is taken as a data input line and input lines taken as the selection lines.

Typical MUX and DMUX application

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



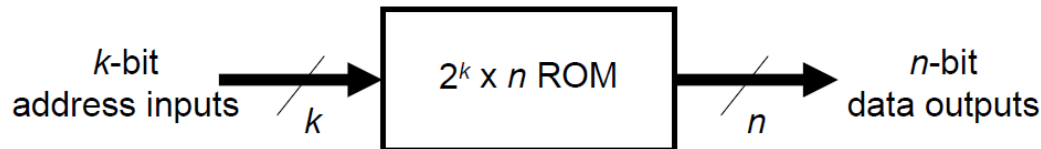
5.4 Programmable logic devices

- Implementation technologies introduced so far are not programmable
 - Fixed integrated circuits
- Programmable Logic Device (PLD)
 - Integrated circuit with internal logic gates connected together
 - These internal connections can be reconfigured (erased / programmed) to form a specific logic circuit
- Three types of basic PLDs
 - ROM, PLA, PAL

Read-Only Memory (ROM)

- An essential memory device in which **permanent** binary information is stored (even when the power is turned off, the information is still there).
- k -bit address inputs and n -bit data outputs
Inputs: address for the memory
Outputs: the word (n data bits) stored in ROM selected by the k -bit input address
- k address input can specify 2^k words

■ k address input can specify 2^k words



- Features:
 - advantage of simple and less connections
 - disadvantage of doubling the size with an additional input
 - cheap for mass production but expensive for development

Other ROMs

- ROM usually works as a lookup table
- No **data inputs** as ROM does not have a **write** operation (only read operation)

- ROMs that can be written / rewritten
PROM

EPROM (Erasable PROM)

EEPROM (Electrically Erasable PROM)

FEPRM (Field Erasable PROM)

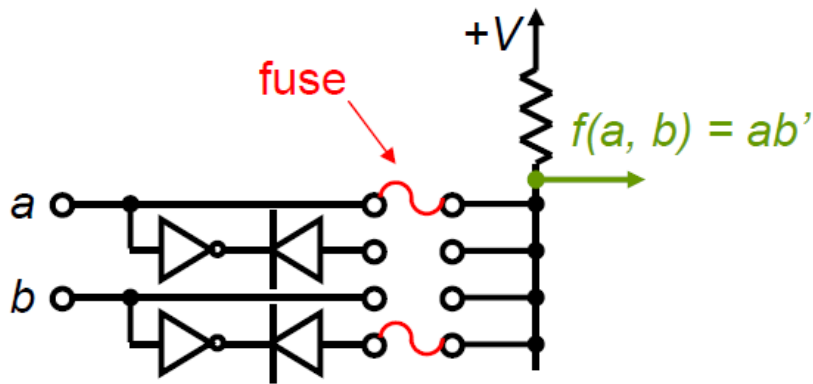
FLASH Memory

PROM

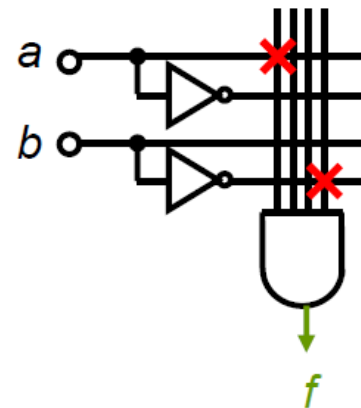
- Programmable Read-Only Memory
- Perform same functions as ROM
- Oldest programmable logic device
- Fuse or anti-fuse programmed
- One-time writing process has been deferred to the end-user
- Advantage: easy programming and inexpensive
- Disadvantage: non-reprogrammable

Fuse-Programmable Arrays

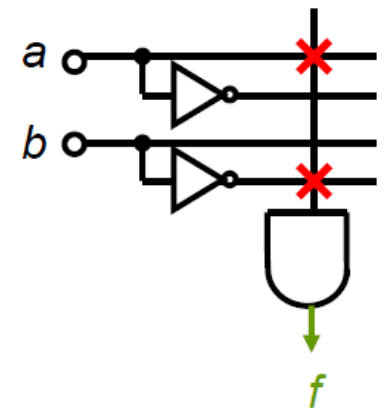
■ Programmable AND Arrays



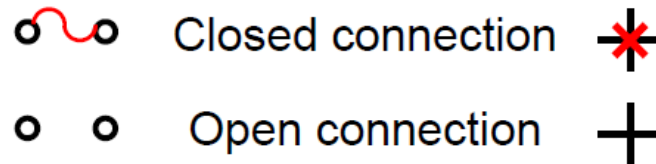
Fused-programmable
AND arrays



Conventional
symbol



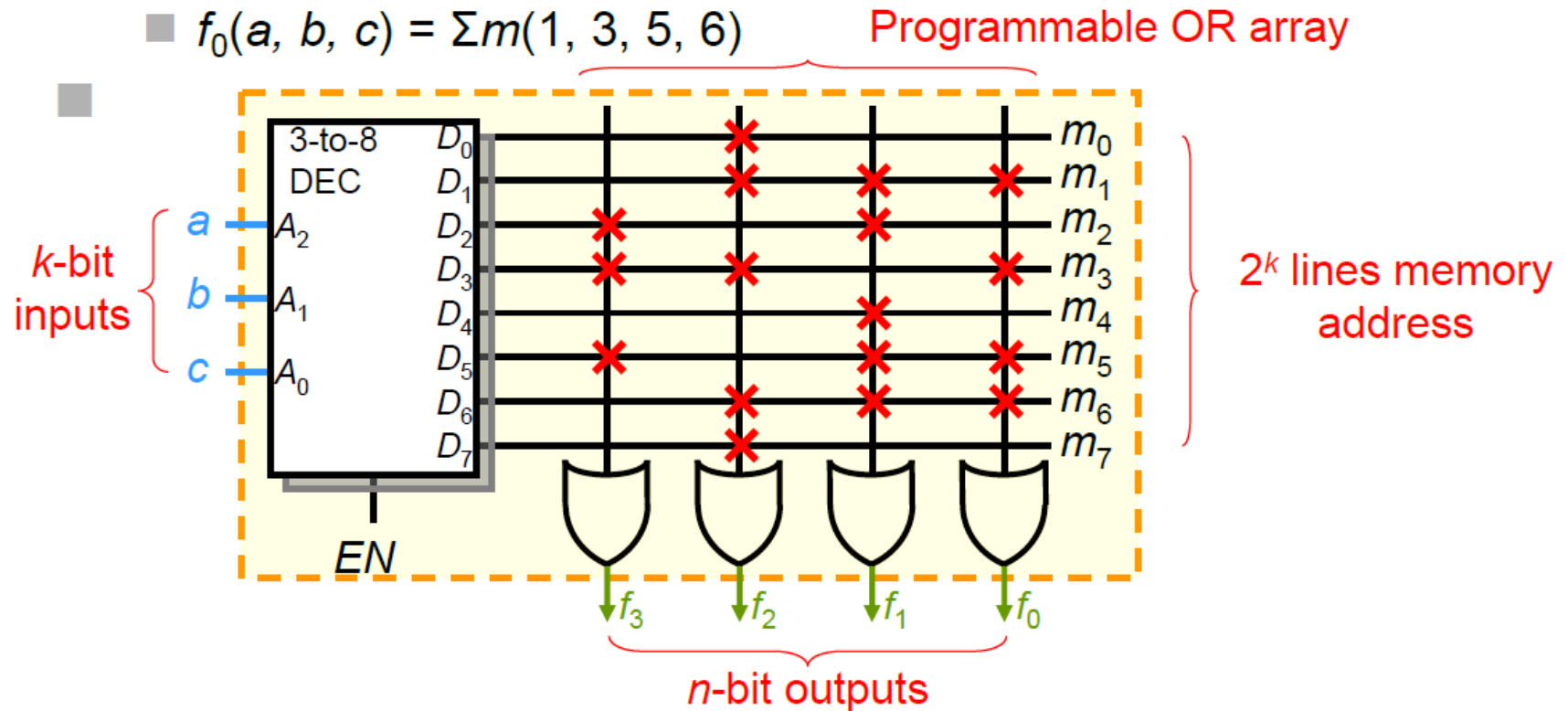
Array logic
symbol



Example: 8 x 4 PROM

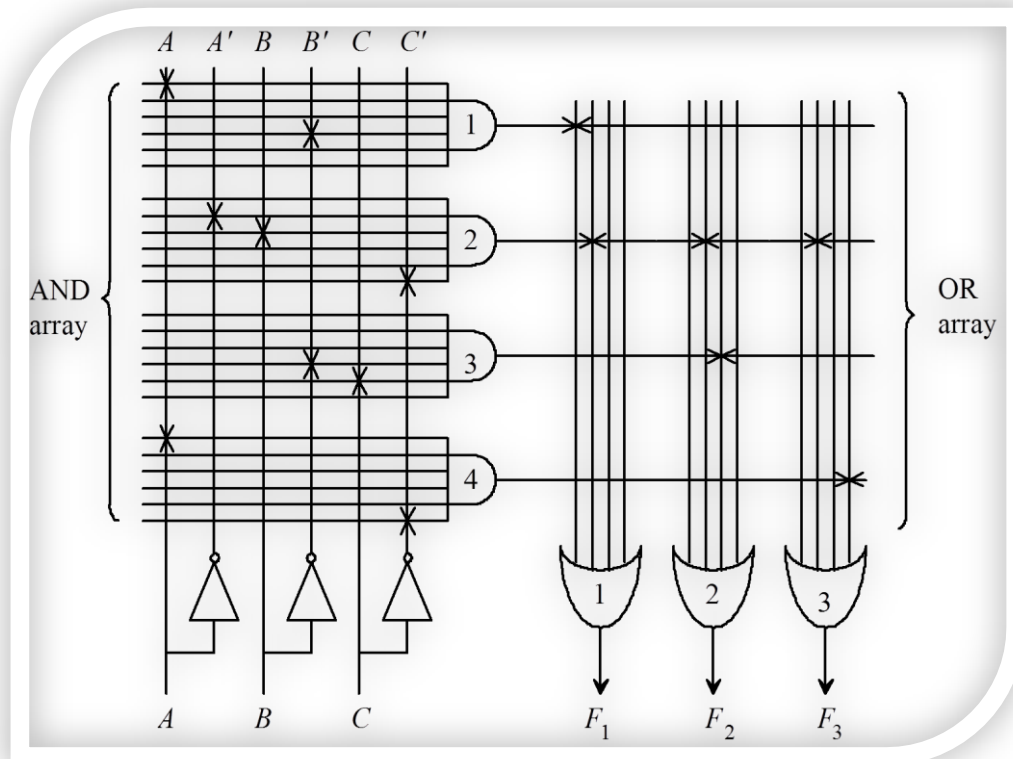
Realize the following functions with a 8 x 4 PROM

- $f_3(a, b, c) = \Sigma m(2, 3, 5)$
- $f_2(a, b, c) = \Sigma m(0, 1, 3, 6, 7)$
- $f_1(a, b, c) = \Sigma m(1, 2, 4, 5, 6)$
- $f_0(a, b, c) = \Sigma m(1, 3, 5, 6)$



Programming Logic Array (PLA)

- Programmable device similar to PROM but with different internal structure
- The device is used to implement logic function in the form of sum-of-product terms

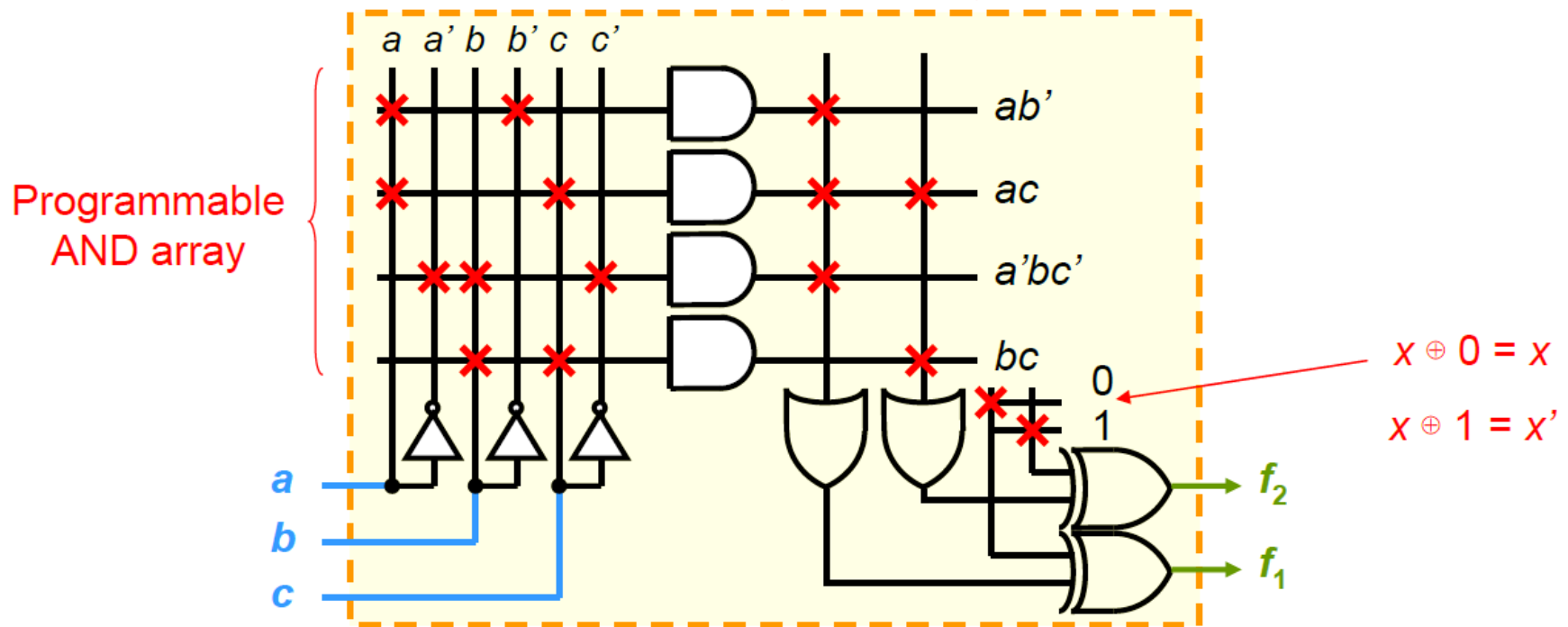


Example: PLA

■ Implement these 2 functions with a PLA

■ $f_1(a, b, c) = ab' + ac + a'bc'$

■ $f_2(a, b, c) = (ac + bc)'$ Programmable OR array



- How many product terms for f_1 and f_2 ?

Minimization in PLA

- Reduce the number of distinct product terms, so as to
Minimize the number of AND gates
Minimize the size of PLA
- Step 1) Simplify the Boolean functions, and also their **complementary functions** to minimum number of terms
- Step 2) Find the combination to produce the minimum number of distinct product terms (or maximum common terms)

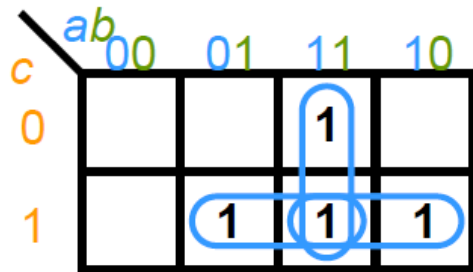
Minimization in PLA: Example

- Implement these 2 functions with a PLA

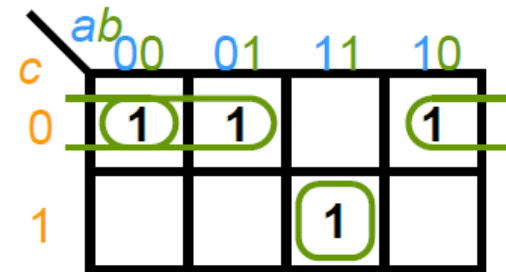
- $f_1(a, b, c) = \Sigma m(3, 5, 6, 7)$

- $f_2(a, b, c) = \Sigma m(0, 2, 4, 7)$

- Step 1) Simplify by K-map first:

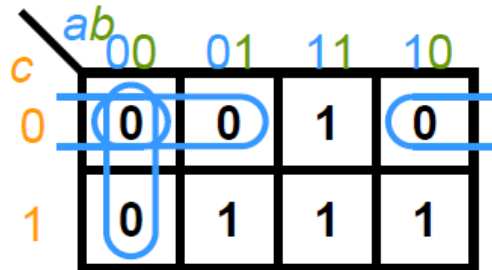


$$f_1 = ac + ab + bc$$

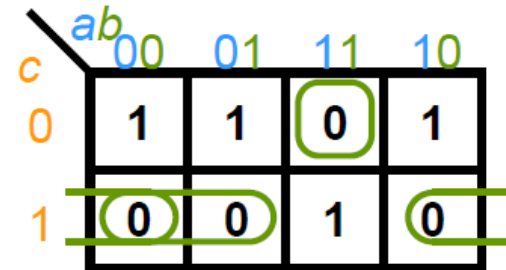


$$f_2 = \underline{a'c'} + \underline{b'c'} + abc$$

- Now find the simplified complementary functions



$$f_1' = \underline{a'c'} + \underline{a'b'} + \underline{b'c'}$$



$$f_2' = \underline{a'c} + \underline{b'c} + \underline{abc'}$$

Minimization in PLA: Example

- Since there are two functions f_1 and f_2
- There are 4 complemented and uncomplemented combinations (why?)
 - (i) f_1 and f_2
 - (ii) f_1 and f_2'
 - (iii) f_1' and f_2
 - (iv) f_1' and f_2'
- We may implement design (i), (ii), (iii), or (iv)
 - How to choose?
 - The design (iii) give the min. no. of distinct product terms (shared 2 common terms)

Minimization in PLA: Example

- Tabular the result of last step in the PLA programming table
 - Design (iii): f_1' and f_2

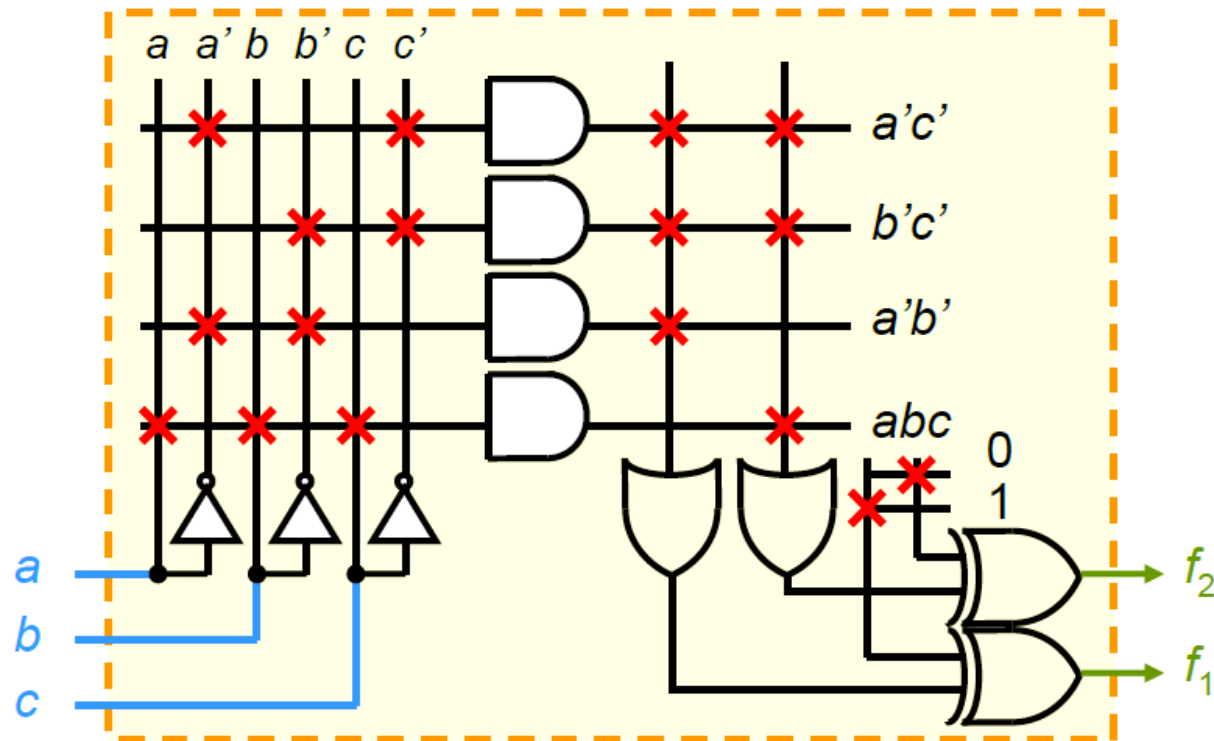
	Product Term	Inputs $a \ b \ c$	Outputs $f_1 \ f_2$
$a'c'$	1	0 – 0	1 1
$b'c'$	2	– 0 0	1 1
$a'b'$	3	0 0 –	1 –
abc	4	1 1 1	– 1
			C T

T: Uncomplemented output

C: Complemented output

Minimization in PLA: Example

- The final resulting PLA fuse map



Programmable Array Logic (PAL)

- PLA: both the AND and the OR arrays are programmable
- PAL: the AND array is programmable and the OR array is fixed
- PLA is a general design for implementing sum-of-product terms, whereas the PAL has fixed sum-of-product terms
- Design for PAL device is easier, but not as flexible as that for PLA

PAL Example

- Implement these functions with a PAL
 - $w(a, b, c, d) = \Sigma m(2, 12, 13)$
 - $x(a, b, c, d) = \Sigma m(7, 8, 9, 10, 11, 12, 13, 14, 15)$
 - $y(a, b, c, d) = \Sigma m(0, 2, 3, 4, 5, 6, 7, 8, 10, 11, 15)$
 - $z(a, b, c, d) = \Sigma m(1, 2, 8, 12, 13)$
- Analysis
 - 4 inputs (a, b, c, d), 4 outputs (w, x, y, z)

PAL Example

- Simplify the functions to minimum number of terms (steps omitted here)
 - $w(a, b, c, d) = abc' + a'b'cd'$
 - $x(a, b, c, d) = a + bcd$
 - $y(a, b, c, d) = a'b + cd + b'd'$
 - $z(a, b, c, d) = \underline{abc' + a'b'cd'} + ac'd' + a'b'c'd$
 - Note: z is sum of 4 product terms
- By considering output w as an input variable
 - $z(a, b, c, d, \underline{w}) = w + ac'd' + a'b'c'd$
 - Now z is SOP of 3 product terms!

PAL Example

- Tabular the result of last step in the PAL programming table

Product Term		AND Inputs <i>a b c d w</i>	Outputs
1	abc'	1 1 0 --	$w = abc' + a'b'cd'$
2	$a'b'cd'$	0 0 1 0 -	
3		-----	
4	a	1 -----	$x = a + bcd$
5	bcd	- 1 1 1 -	
6		-----	
7	$a'b$	0 1 ---	$y = a'b + cd + b'd'$
8	cd	-- 1 1 -	
9	$b'd'$	- 0 - 0 -	
10	w	---- 1	$z = w + ac'd' + a'b'c'd$
11	$ac'd'$	1 - 0 0 -	
12	$a'b'c'd$	0 0 0 1 -	

PAL Example

■ The final result

■ Reuse w

