Introduction to Digital Logic

Assist. Prof. Dr. Hamza Osman ILHAN

hoilhan@yildiz.edu.tr

Course Outline

1. Digital Computers, Number Systems, Arithmetic Operations, Decimal, Alphanumeric, and Gray Codes
2. Binary Logic, Gates, Boolean Algebra, Standard Forms
3. Circuit Optimization, Two-Level Optimization, Map Manipulation, Multi-Level Circuit Optimization
4. Additional Gates and Circuits, Other Gate Types, Exclusive-OR Operator and Gates, High-Impedance Outputs
5. Implementation Technology and Logic Design, Design Concepts and Automation, The Design Space, Design Procedure, The major design steps
6. Programmable Implementation Technologies: Read-Only Memories, Programmable Logic Arrays, Programmable Array Logic,Technology mapping to programmable logic devices
7. Combinational Circuits
8. Arithmetic Functions and Circuits
9. Sequential Circuits Storage Elements and Sequential Circuit Analysis
10. Sequential Circuits, Sequential Circuit Design State Diagrams, State Tables
11. Counters, register cells, buses, & serial operations
12. Sequencing and Control, Datapath and Control, Algorithmic State Machines (ASM)
13. Memory Basics

Introduction to Digital Logic

Lecture 7

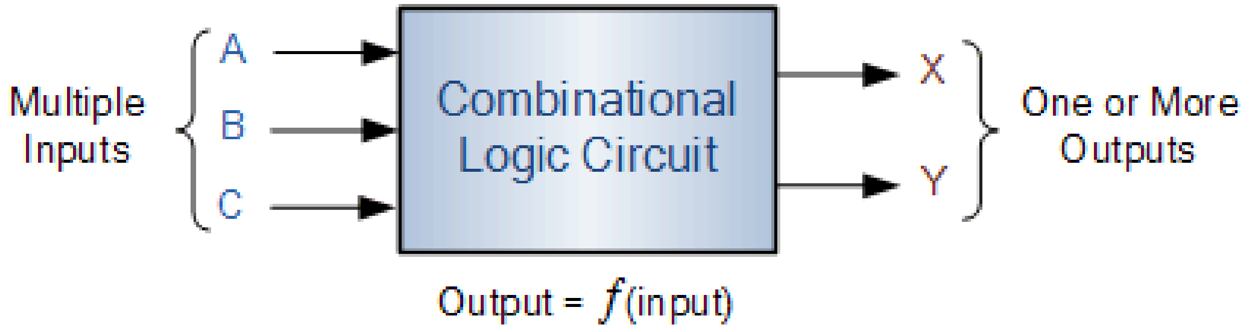
Combinational Circuits

Overview

- Combinational Circuit
- Decoding
- Encoding
- Selecting
- Implementing Combinational Functions Using:
  - The Adder and Subtractors
  - Decoders and Encoders
  - Multiplexers and Demultiplexer
  - ROMs
  - PLAs
  - PALs
  - Lookup Tables

Combinational Circuit

- **Combinational Circuits (CC)** are circuits made up of different types of logic gates. A logic gate is a basic building block of any electronic circuit. The output of the combinational circuit depends on the values at the input at any given time. The circuits do not make use of any memory or storage device.



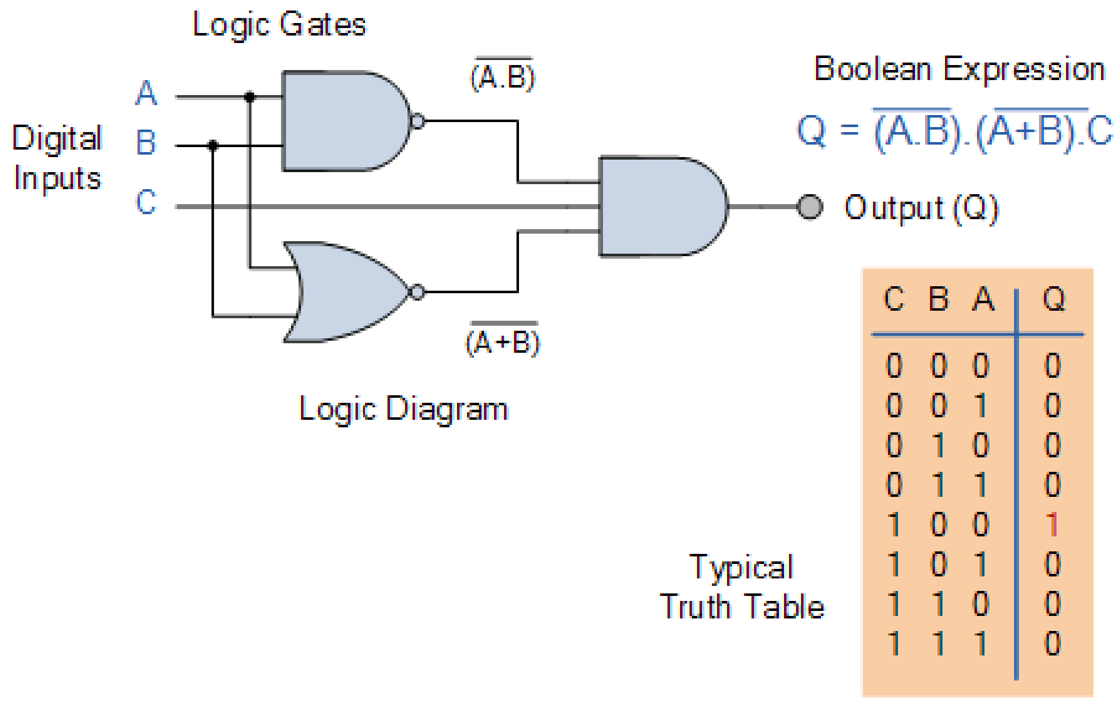
- Combinational Logic Circuits are made up from basic logic NAND, NOR or NOT gates that are “combined” or connected together to produce more complicated switching circuits. These logic gates are the building blocks of combinational logic circuits.

Combinational Circuit

- Combinational logic circuits can be very simple or very complicated and any combinational circuit can be implemented with only NAND and NOR gates as these are classed as “universal” gates.
- The three main ways of specifying the function of a combinational logic circuit are:
  - Boolean Algebra – This forms the algebraic expression showing the operation of the logic circuit for each input variable either True or False that results in a logic “1” output.
  - Truth Table – A truth table defines the function of a logic gate by providing a concise list that shows all the output states in tabular form for each possible combination of input variable that the gate could encounter.
  - Logic Diagram – This is a graphical representation of a logic circuit that shows the wiring and connections of each individual logic gate, represented by a specific graphical symbol, that implements the logic circuit.

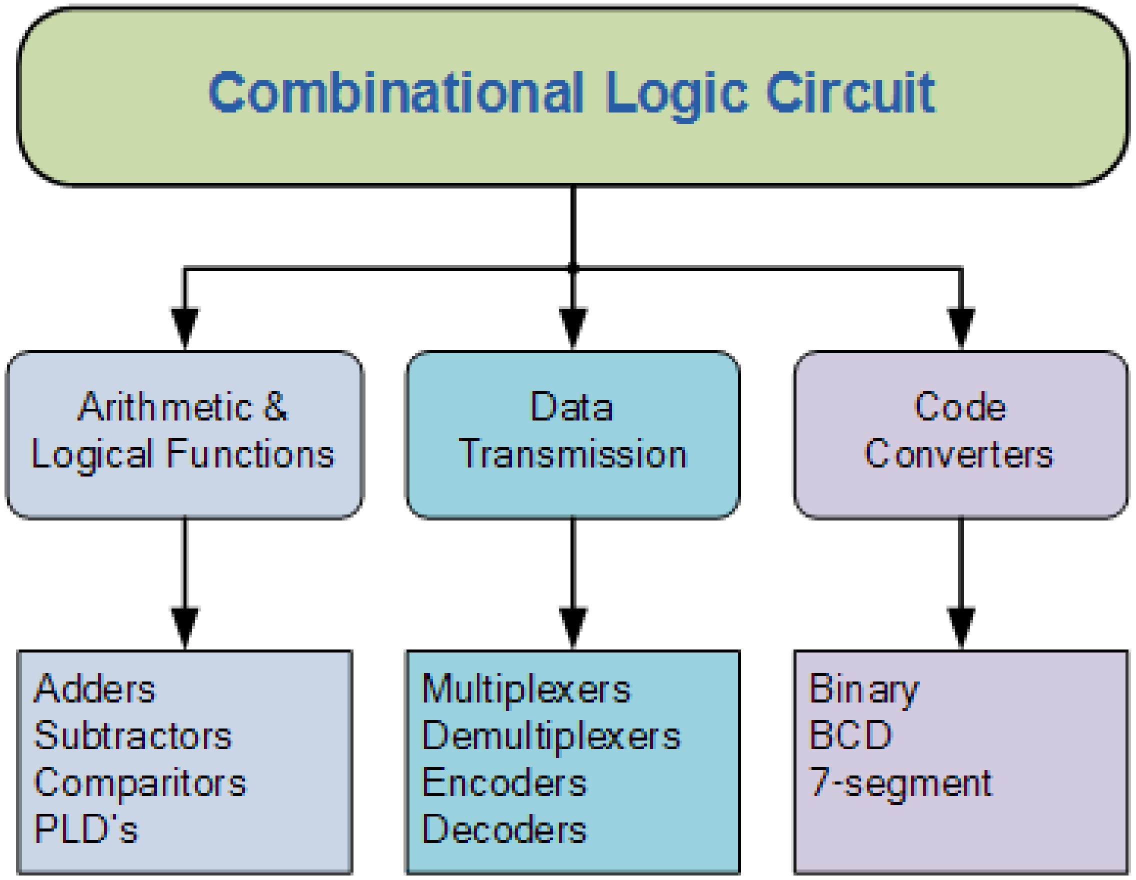
Combinational Circuit

- As combinational logic circuits are made up from individual logic gates only, they can also be considered as “decision making circuits” and combinational logic is about combining logic gates together to process two or more signals in order to produce at least one output signal according to the logical function of each logic gate. Common combinational circuits made up from individual logic gates that carry out a desired application include Multiplexers, De-multiplexers, Encoders, Decoders, Full and Half Adders etc.



C	B	A	Q
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0

Combinational Circuit

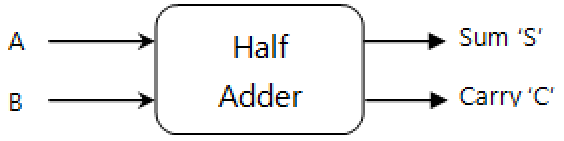


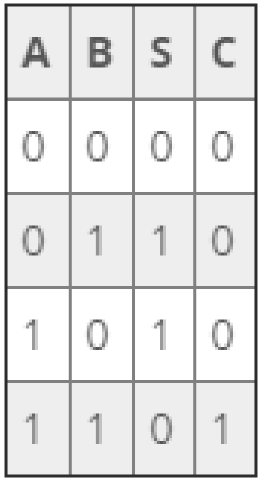
Adder

- An adder is a digital circuit that is used to perform the addition of numeric values. It is one of the most basic circuits and is found in arithmetic logic units of computing devices.
- There are two types of adders. Half adders compute single digit numbers, while full adders compute larger numbers.

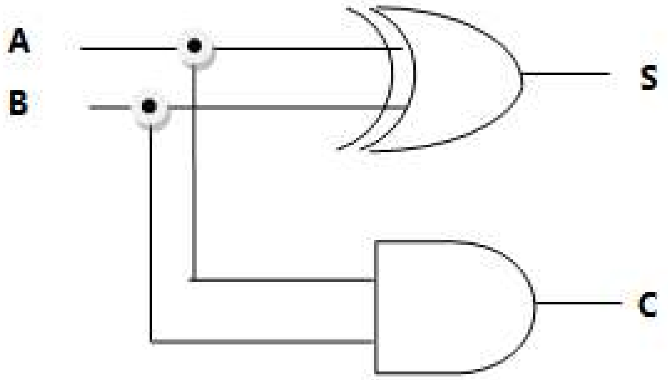
Half Adder

- The half adder adds two single digit binary numbers and forms the foundation for all addition operations in computing.
- If we have two single binary digits, A and B, then the half adder adds them with the circuit carrying two outputs, the sum and the carry.
- The carry represents any overflow from the addition of the two numbers.





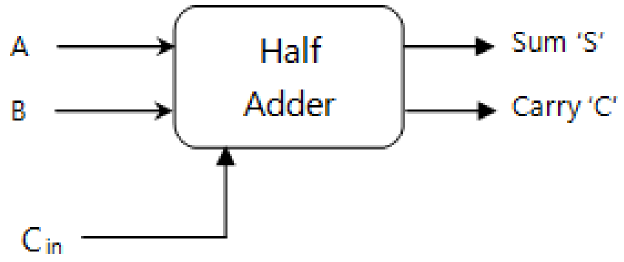
A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

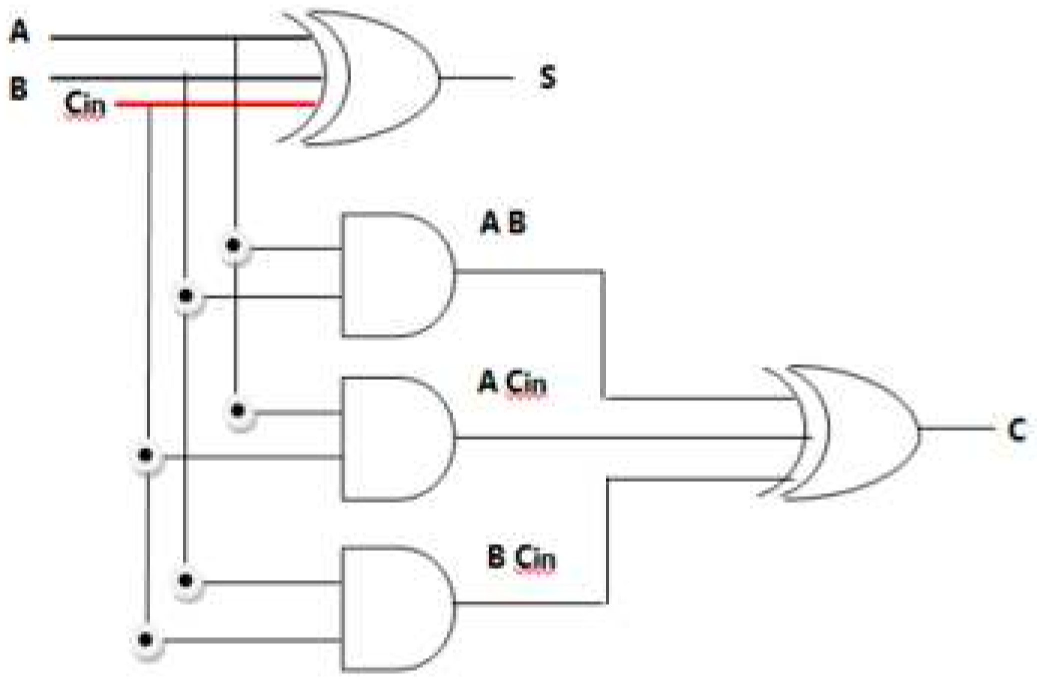


Adder

Full Adder

- The full adder overcomes the disadvantages of the half adder in that it can add two single bit numbers in addition to the carry digit at its input as seen in this figure:



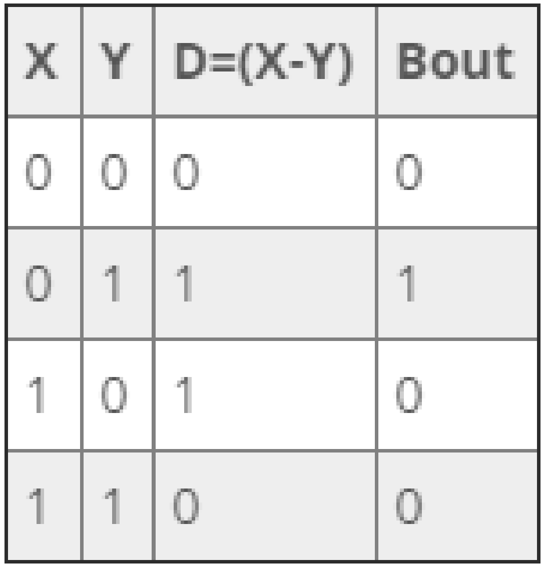


A	B	Cin	Co	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	0	1
1	1	1	1	1

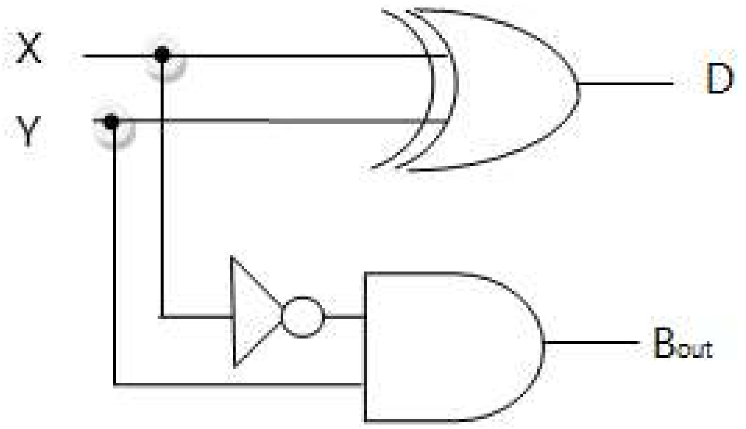
Subtractor

- A subtractor is used to subtract one number from another.
- Because we are dealing with binary digits, the 1s complement and 2s complement of the numbers are used to achieve this.
- Three bits are involved in performing the basic subtraction: the minuend (X), the subtrahend (Y) and the borrow (Bi), which is input from the previous bit. The outputs are the difference (D) and the borrow bit (Bout).

Half Subtractor



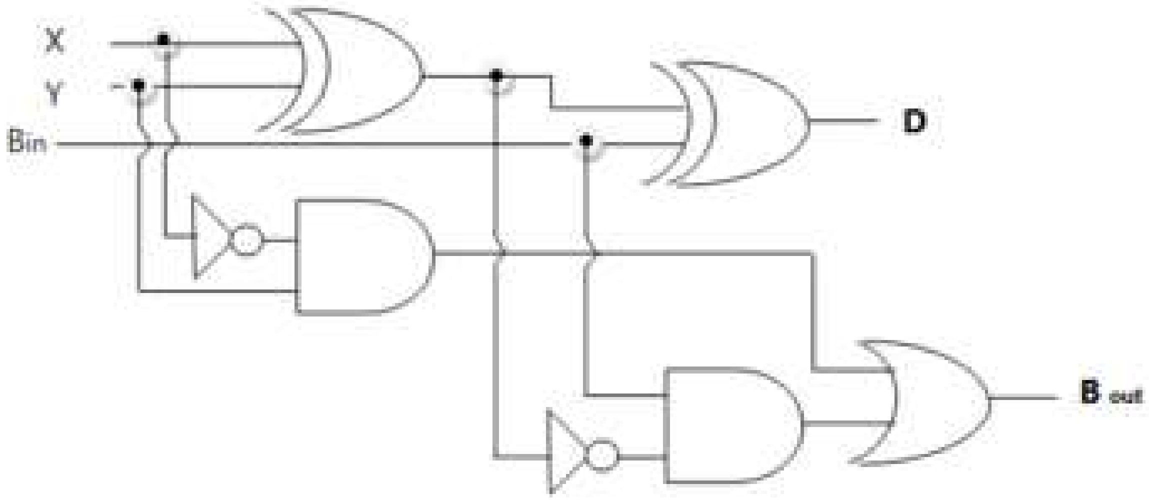
X	Y	D=(X-Y)	Bout
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0



Subtractor

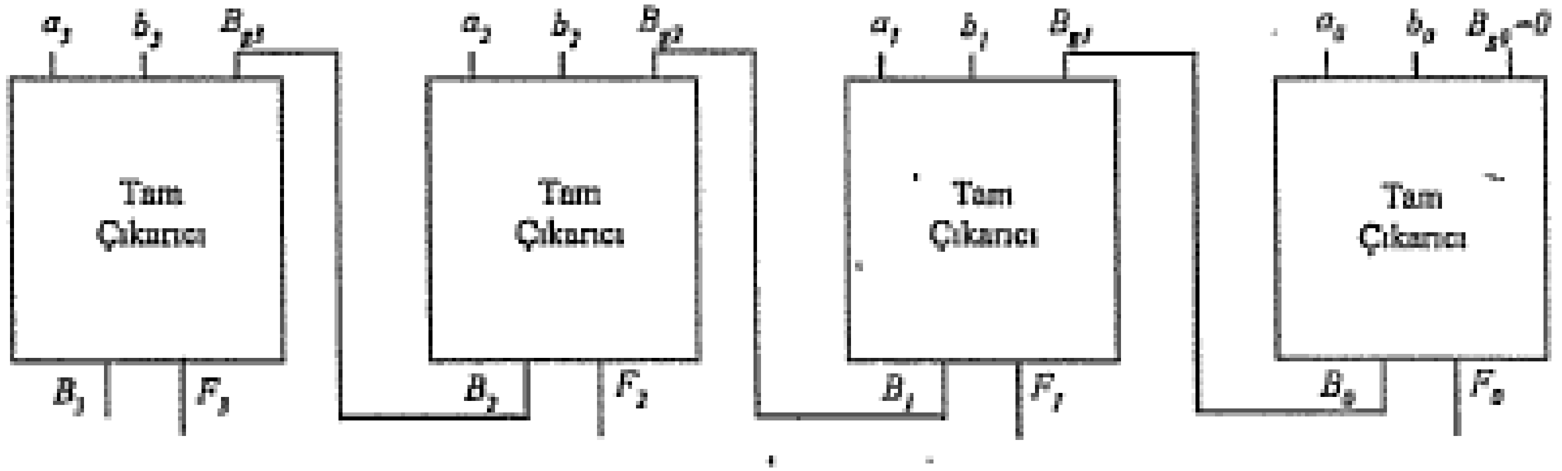
Full Subtractor

X	Y	Bin	D=X-Y-Bin	Bout
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

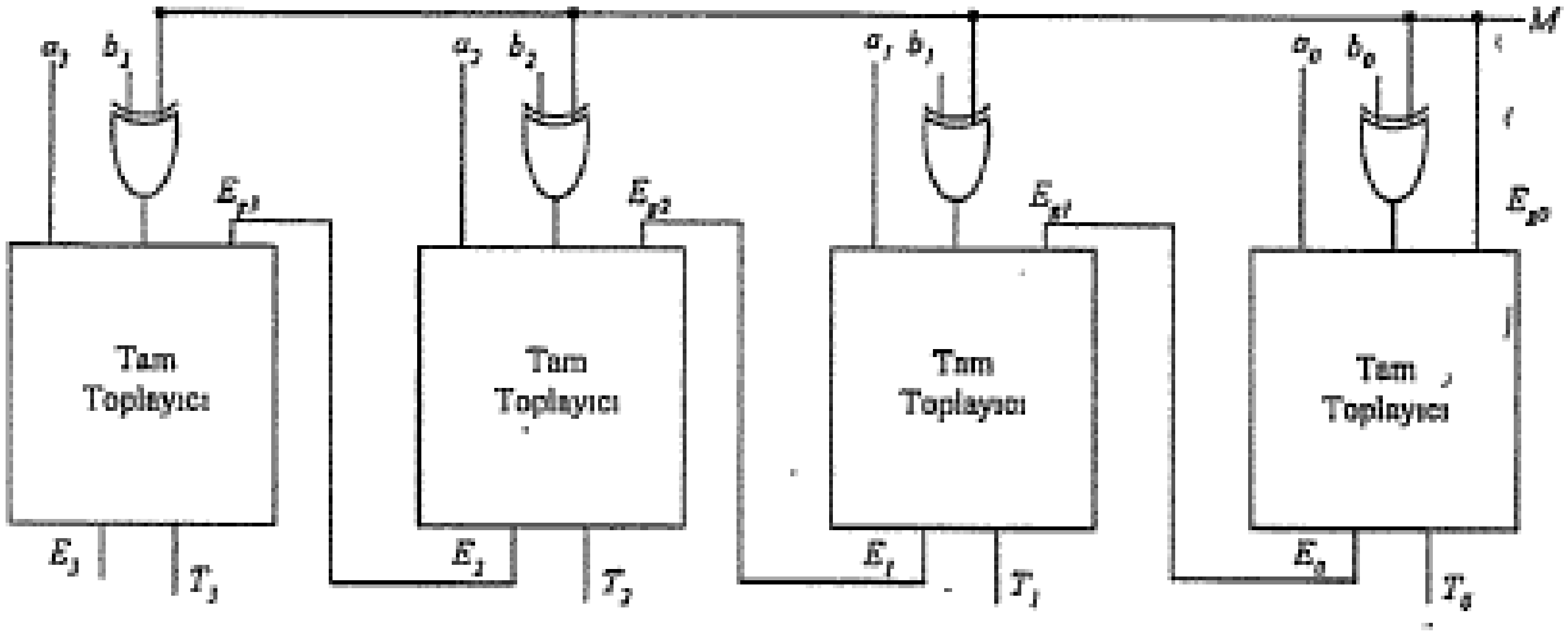


Subtractor

Paralel Full Subtractor

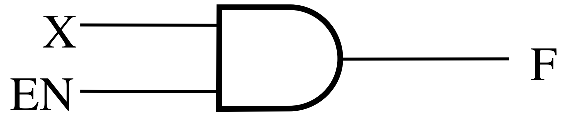


Subtractor / Adder Combine Circuit

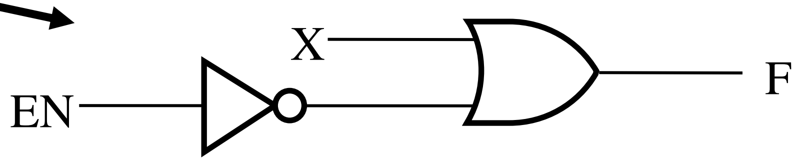


Enabling Function

- *Enabling* permits an input signal to pass through to an output
- *Disabling* blocks an input signal from passing through to an output, replacing it with a fixed value
- The value on the output when it is disable can be Hi-Z (as for three-state buffers and transmission gates), 0 , or 1
- When disabled, 0 output
- When disabled, 1 output
- Enabling applications?



(a)



(b)

Decoding

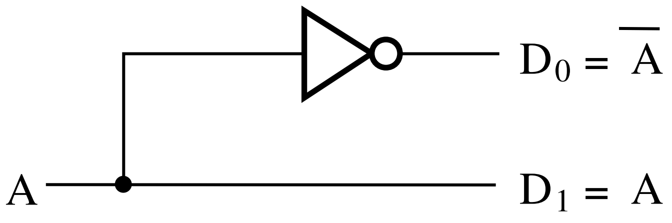
- Decoding - the conversion of an n-bit input code to an m-bit output code with $n \leq m \leq 2^n$ such that each valid code word produces a unique output code
- Circuits that perform decoding are called *decoders*
- Here, functional blocks for decoding are
  - called n-to-m line decoders, where $m \leq 2^n$, and
  - generate 2^n (or fewer) minterms for the n input variables

Decoder Examples

- 1-to-2-Line Decoder

A	D_0	D_1
0	1	0
1	0	1

(a)

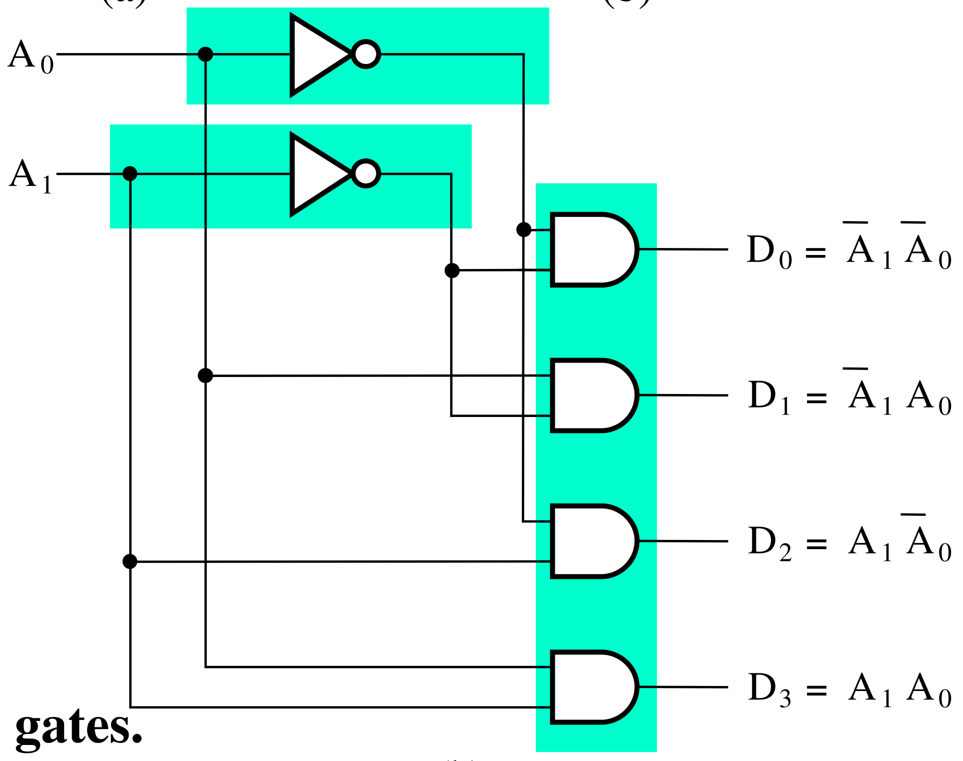


(b)

- 2-to-4-Line Decoder

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

(a)



A_0
A_1
$D_0 = \overline{A}_1 \overline{A}_0$
$D_1 = \overline{A}_1 A_0$
$D_2 = A_1 \overline{A}_0$
$D_3 = A_1 A_0$

(b)

- **Note that the 2-4-line made up of 2 1-to-2-line decoders and 4 AND gates.**

Decoder Expansion

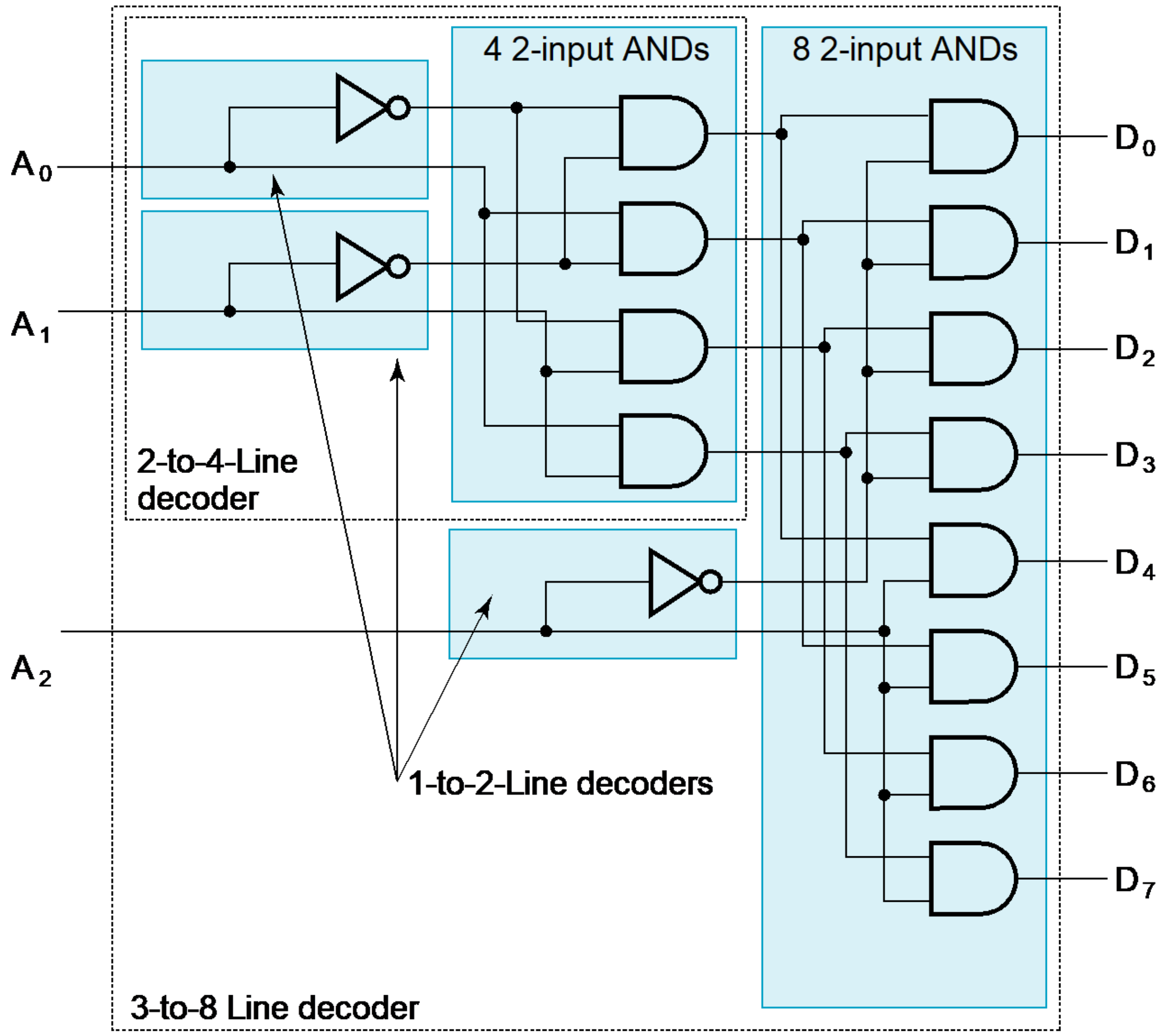
- General procedure given in book for any decoder with n inputs and 2^n outputs.
- This procedure builds a decoder backward from the outputs.
- The output AND gates are driven by two decoders with their numbers of inputs either equal or differing by 1.
- These decoders are then designed using the same procedure until 2-to-1-line decoders are reached.
- The procedure can be modified to apply to decoders with the number of outputs $\neq 2^n$

Decoder Expansion - Example 1

- 3-to-8-line decoder
  - Number of output ANDs = 8
  - Number of inputs to decoders driving output ANDs = 3
  - Closest possible split to equal
    - 2-to-4-line decoder
    - 1-to-2-line decoder
  - 2-to-4-line decoder
    - Number of output ANDs = 4
    - Number of inputs to decoders driving output ANDs = 2
    - Closest possible split to equal
      - Two 1-to-2-line decoders

Decoder Expansion - Example 1

- Result

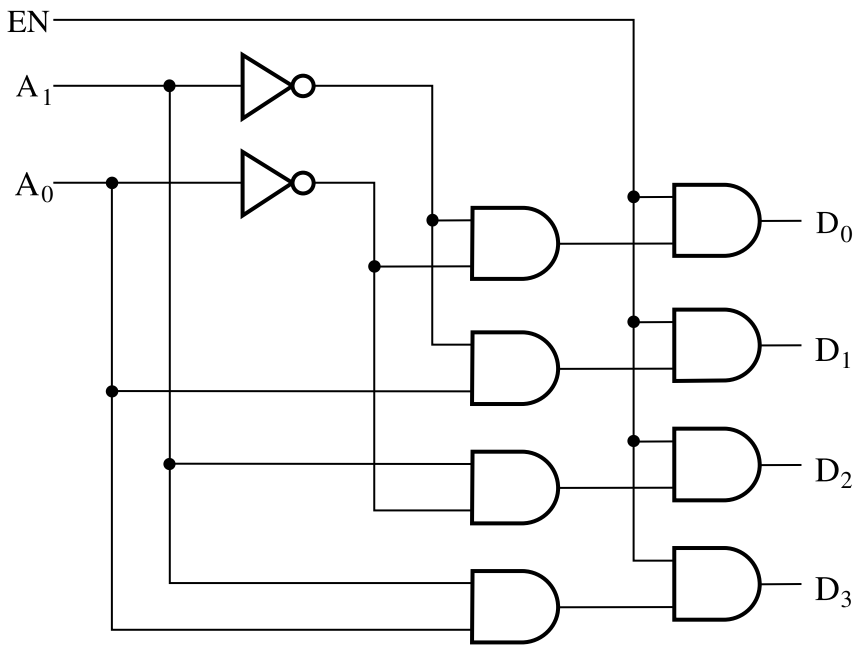


Decoder with Enable

- In general, attach *m*-enabling circuits to the outputs
- Truth table for the function
  - Note use of X's to denote both 0 and 1
  - Combination containing two X's represent four binary combinations
- Alternatively, can be viewed as distributing value of signal EN to 1 of 4 outputs
- In this case, called a *demultiplexer*

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

(a)



(b)

Encoding

- Encoding - the opposite of decoding - the conversion of an m-bit input code to a n-bit output code with $n \leq m \leq 2^n$ such that each valid code word produces a unique output code
- Circuits that perform encoding are called *encoders*
- An encoder has 2^n (or fewer) input lines and n output lines which generate the binary code corresponding to the input values
- Typically, an encoder converts a code containing exactly one bit that is 1 to a binary code corresponding to the position in which the 1 appears.

Encoder Example

- A decimal-to-BCD encoder
  - Inputs: 10 bits corresponding to decimal digits 0 through 9, (D_0, …, D_9)
  - Outputs: 4 bits with BCD codes
  - Function: If input bit D_i is a 1, then the output (A_3, A_2, A_1, A_0) is the BCD code for i,
- The truth table could be formed, but alternatively, the equations for each of the four outputs can be obtained directly.

Encoder Example (continued)

- Input D_i is a term in equation A_j if bit A_j is 1 in the binary value for i.
- Equations:

$$A_3 = D_8 + D_9$$

$$A_2 = D_4 + D_5 + D_6 + D_7$$

$$A_1 = D_2 + D_3 + D_6 + D_7$$

$$A_0 = D_1 + D_3 + D_5 + D_7 + D_9$$

- $F_1 = D_6 + D_7$ can be extracted from A_2 and A_1

Priority Encoder

- If more than one input value is 1, then the encoder just designed does not work.
- One encoder that can accept all possible combinations of input values and produce a meaningful result is a *priority encoder*.
- Among the 1s that appear, it selects the most significant input position (or the least significant input position) containing a 1 and responds with the corresponding binary code for that position.

Priority Encoder Example

- Priority encoder with 5 inputs (D_4, D_3, D_2, D_1, D_0) - highest priority to most significant 1 present - Code outputs A2, A1, A0 and V where V indicates at least one 1 present.

No. of Min-terms/Row	Inputs					Outputs			
	D4	D3	D2	D1	D0	A2	A1	A0	V
0	0	0	0	0	0	X	X	X	0
1	0	0	0	0	1	0	0	0	1
2	0	0	0	1	X	0	0	1	1
4	0	0	1	X	X	0	1	0	1
8	0	1	X	X	X	0	1	1	1
16	1	X	X	X	X	1	0	0	1

- Xs in input part of table represent 0 or 1; thus table entries correspond to product terms instead of minterms. The column on the left shows that all 32 minterms are present in the product terms in the table

Priority Encoder Example (continued)

- Could use a K-map to get equations, but can be read directly from table and manually optimized if careful:

$$\mathbf{A_2 = D_4}$$

$$\mathbf{A_1 = \overline{D}_4 D_3 + \overline{D}_4 \overline{D}_3 D_2 = \overline{D}_4 F_1,\ \ F_1 = (D_3 + D_2)}$$

$$\mathbf{A_0 = \overline{D}_4 D_3 + \overline{D}_4 \overline{D}_3 \overline{D}_2 D_1 = \overline{D}_4 (D_3 + \overline{D}_2 D1)}$$

$$\mathbf{V = D_4 + F_1 + D_1 + D_0}$$

Selecting

- Selecting of data or information is a critical function in digital systems and computers
- Circuits that perform selecting have:
  - A set of information inputs from which the selection is made
  - A single output
  - A set of control lines for making the selection
- Logic circuits that perform selecting are called *multiplexers*
- Selecting can also be done by three-state logic or transmission gates

Multiplexers

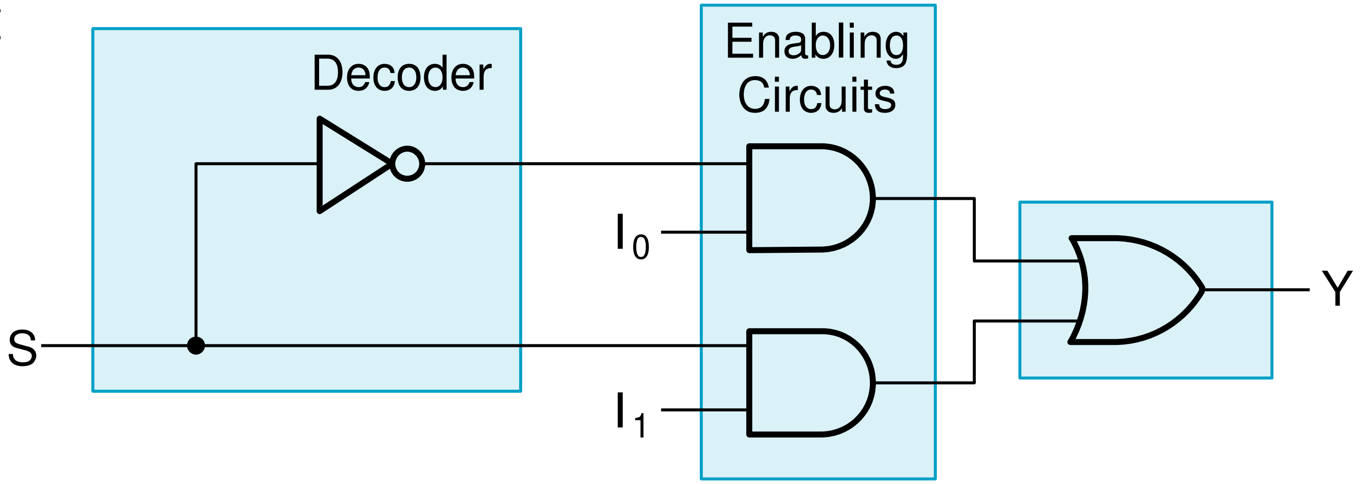
- A multiplexer selects information from an input line and directs the information to an output line
- A typical multiplexer has *n* control inputs (S_{n-1}, … S_0) called *selection inputs*, 2^n information inputs ($I_{2^n - 1}$, … I_0), and one output Y
- A multiplexer can be designed to have *m* information inputs with $m < 2^n$ as well as *n* selection inputs

2-to-1-Line Multiplexer

- Since $2 = 2^1$, $n = 1$
- The single selection variable S has two values:
  - $S = 0$ selects input I_0
  - $S = 1$ selects input I_1
- The equation:

$$Y = \overline{S}I_0 + SI_1$$

- The circuit:

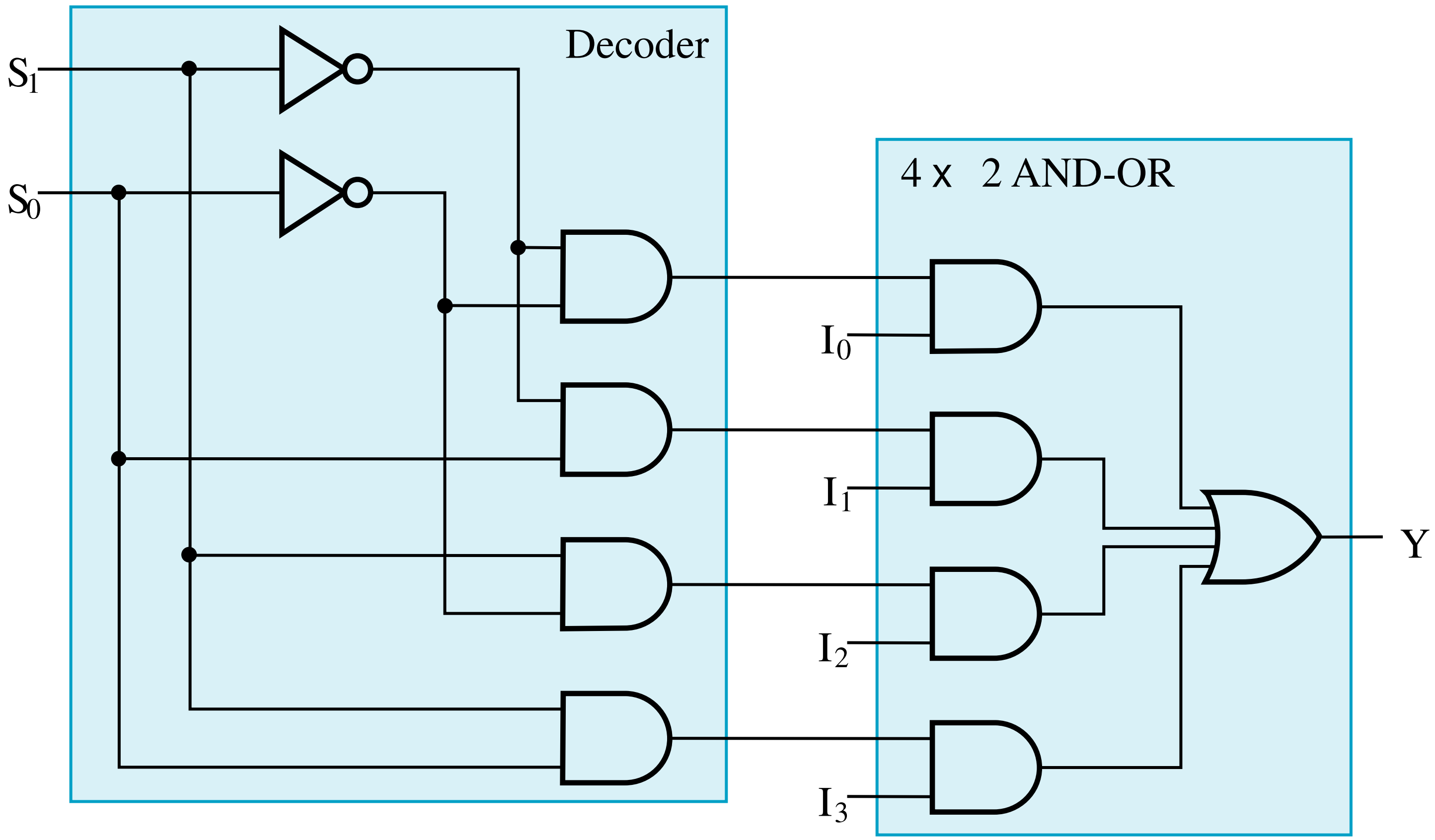


2-to-1-Line Multiplexer (continued)

- Note the regions of the multiplexer circuit shown:
  - 1-to-2-line Decoder
  - 2 Enabling circuits
  - 2-input OR gate
- To obtain a basis for multiplexer expansion, we combine the Enabling circuits and OR gate into a 2 × 2 AND-OR circuit:
  - 1-to-2-line decoder
  - 2 × 2 AND-OR
- In general, for an 2^n-to-1-line multiplexer:
  - n-to-2^n-line decoder
  - $2^n \times 2$ AND-OR

Example: 4-to-1-line Multiplexer

- 2-to-2^2-line decoder
- $2^2 \times$ 2 AND-OR



Example: Gray to Binary Code

- Design a circuit to convert a 3-bit Gray code to a binary code
- The formulation gives the truth table on the right
- It is obvious from this table that X = C and the Y and Z are more complex

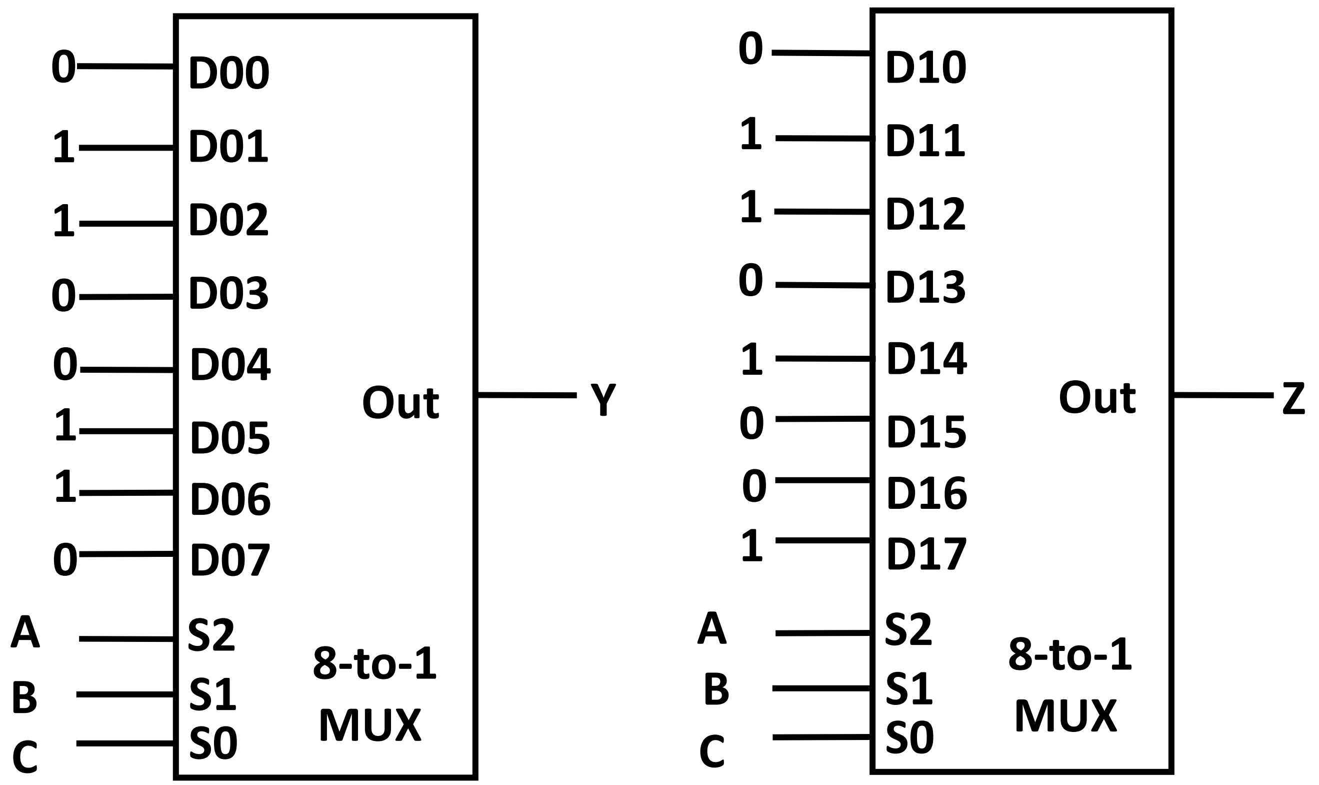
Gray **A B C**	**Binary** **x y z**
0 0 0	0 0 0
1 0 0	0 0 1
1 1 0	0 1 0
0 1 0	0 1 1
0 1 1	1 0 0
1 1 1	1 0 1
1 0 1	1 1 0
0 0 1	1 1 1

Gray to Binary (continued)

- Rearrange the table so that the input combinations are in counting order
- Functions y and z can be implemented using a dual 8-to-1-line multiplexer by:
  - connecting A, B, and C to the multiplexer select inputs
  - placing y and z on the two multiplexer outputs
  - connecting their respective truth table values to the inputs

Gray	**Binary**
A B C	**x y z**
0 0 0	0 0 0
0 0 1	1 1 1
0 1 0	0 1 1
0 1 1	1 0 0
1 0 0	0 0 1
1 0 1	1 1 0
1 1 0	0 1 0
1 1 1	1 0 1

Gray to Binary (continued)



- Note that the multiplexer with fixed inputs is identical to a ROM with 3-bit addresses and 2-bit data!

Multiplexer Approach 2

- Implement any *m* functions of *n* + 1 variables by using:
  - An m-wide 2^n-to-1-line multiplexer
  - A single inverter
- Design:
  - Find the truth table for the functions.
  - Based on the values of the first *n* variables, separate the truth table rows into pairs
  - For each pair and output, define a rudimentary function of the final variable (0, 1, X, $\overline{X}$)
  - Using the first *n* variables as the index, value-fix the information inputs to the multiplexer with the corresponding rudimentary functions
  - Use the inverter to generate the rudimentary function $\overline{X}$

Example: Gray to Binary Code

- Design a circuit to convert a 3-bit Gray code to a binary code
- The formulation gives the truth table on the right
- It is obvious from this table that x = C and the Y and Z are more complex

Gray **A B C**	**Binary** **x y z**
0 0 0	0 0 0
1 0 0	0 0 1
1 1 0	0 1 0
0 1 0	0 1 1
0 1 1	1 0 0
1 1 1	1 0 1
1 0 1	1 1 0
0 0 1	1 1 1

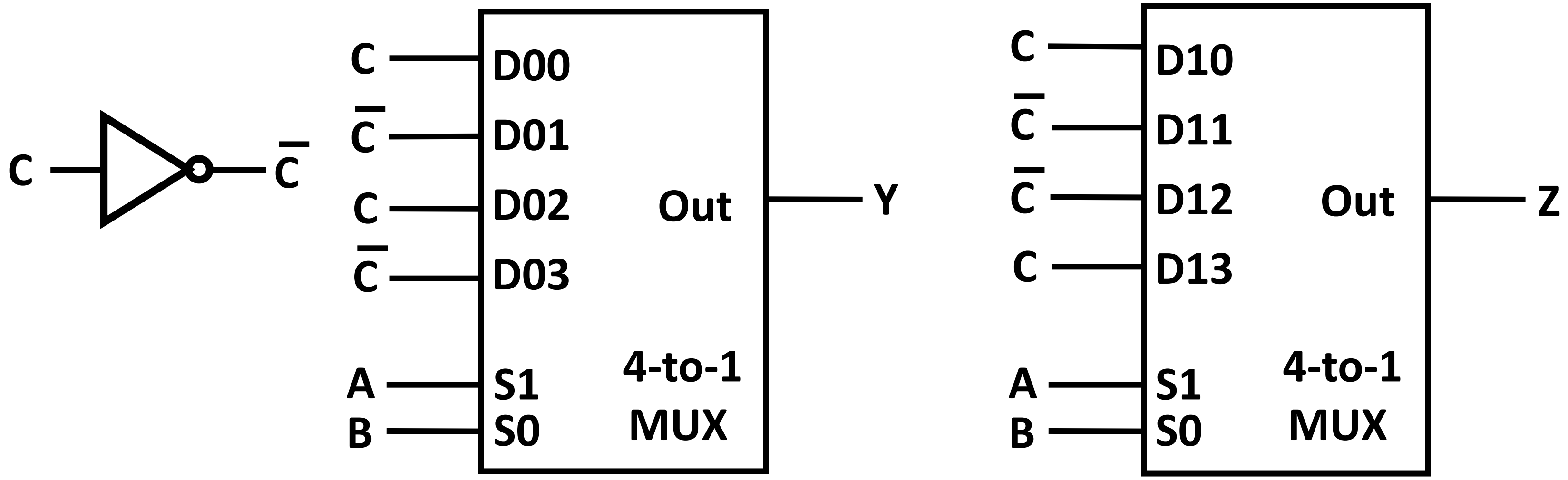
Gray to Binary (continued)

- Rearrange the table so that the input combinations are in counting order, pair rows, and find rudimentary functions

Gray A B C	Binary x y z	Rudimentary Functions of C for y	Rudimentary Functions of C for z
0 0 0	0 0 0	**F = C**	**F = C**
0 0 1	1 1 1		
0 1 0	0 1 1	**F = $\overline{C}$**	**F = $\overline{C}$**
0 1 1	1 0 0		
1 0 0	0 0 1	**F = C**	**F = $\overline{C}$**
1 0 1	1 1 0		
1 1 0	0 1 0	**F = $\overline{C}$**	**F = C**
1 1 1	1 0 1		

Gray to Binary (continued)

- Assign the variables and functions to the multiplexer inputs:

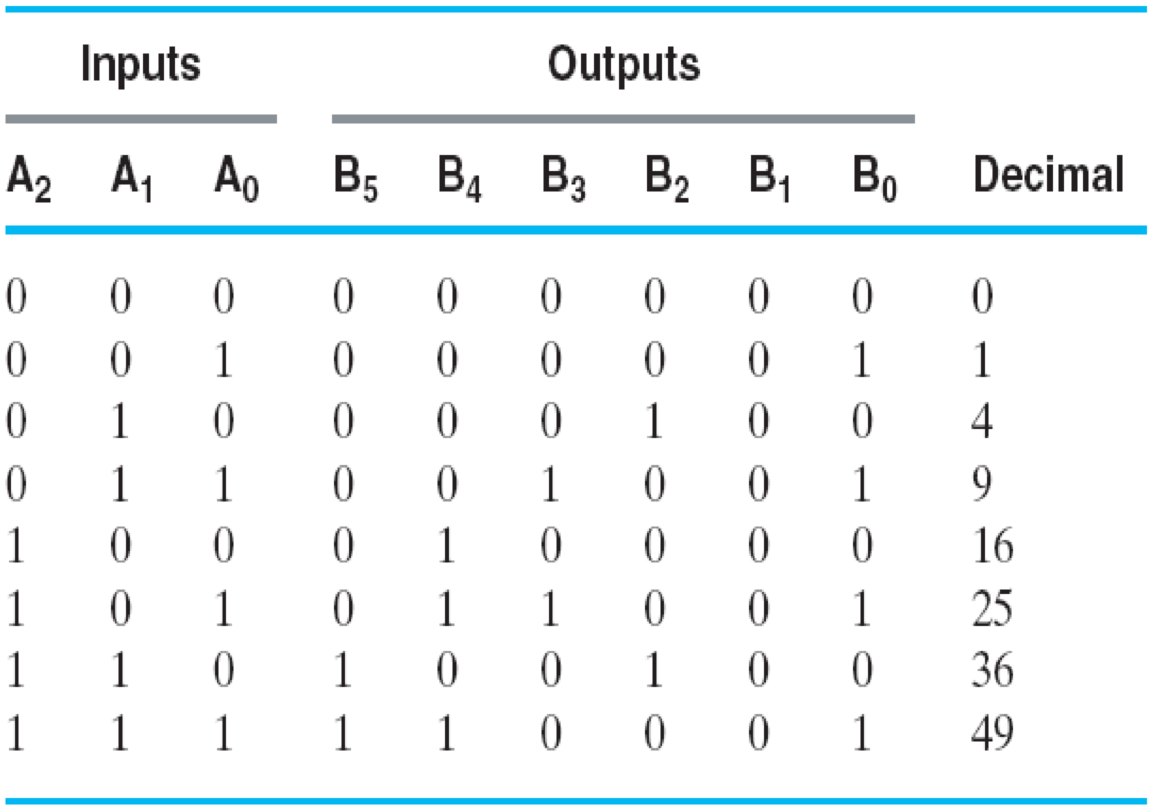


- Note that this approach (Approach 2) reduces the cost by almost half compared to Approach 1.
- This result is no longer ROM-like
- Extending, a function of more than *n* variables is decomposed into several sub-functions defined on a subset of the variables. The multiplexer then selects among these sub-functions.

Read Only Memory

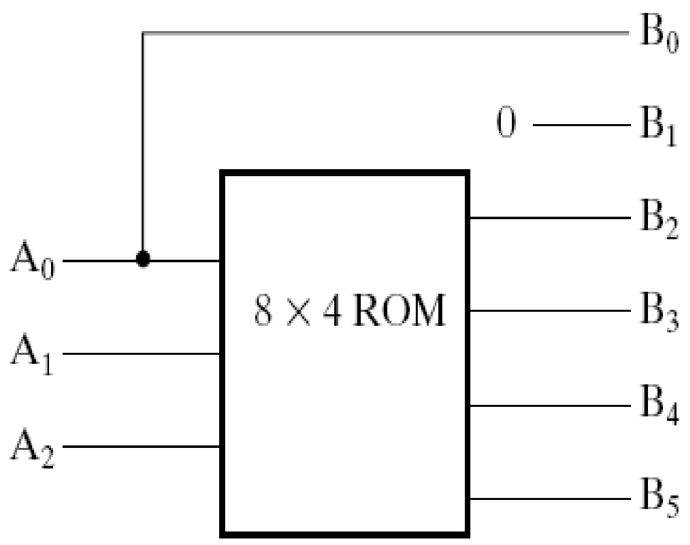
- Functions are implemented by storing the truth table
- Other representations such as equations more convenient
- Generation of programming information from equations usually done by software
- Text Example 4-10 : Design a combinational circuit using a ROM. The circuit accepts a 3 bit number and generates an output binary number equal to the square of the input number

Truth table



Inputs			Outputs						
A_2	A_1	A_0	B_5	B_4	B_3	B_2	B_1	B_0	Decimal
0	0	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	1	1
0	1	0	0	0	0	1	0	0	4
0	1	1	0	0	1	0	0	1	9
1	0	0	0	1	0	0	0	0	16
1	0	1	0	1	1	0	0	1	25
1	1	0	1	0	0	1	0	0	36
1	1	1	1	1	0	0	0	1	49

Implementation



(a) Block diagram

A_2	A_1	A_0	B_5	B_4	B_3	B_2
0	0	0	0	0	0	0
0	0	1	0	0	0	0
0	1	0	0	0	0	1
0	1	1	0	0	1	0
1	0	0	0	1	0	0
1	0	1	0	1	1	0
1	1	0	1	0	0	1
1	1	1	1	1	0	0

(b) ROM truth table

Two outputs are generated outside of the ROM
In the implementation of the system, these two functions are "hardwired" and even if the ROM is reprogrammable or removable, cannot be corrected or updated

Programmable Array Logic

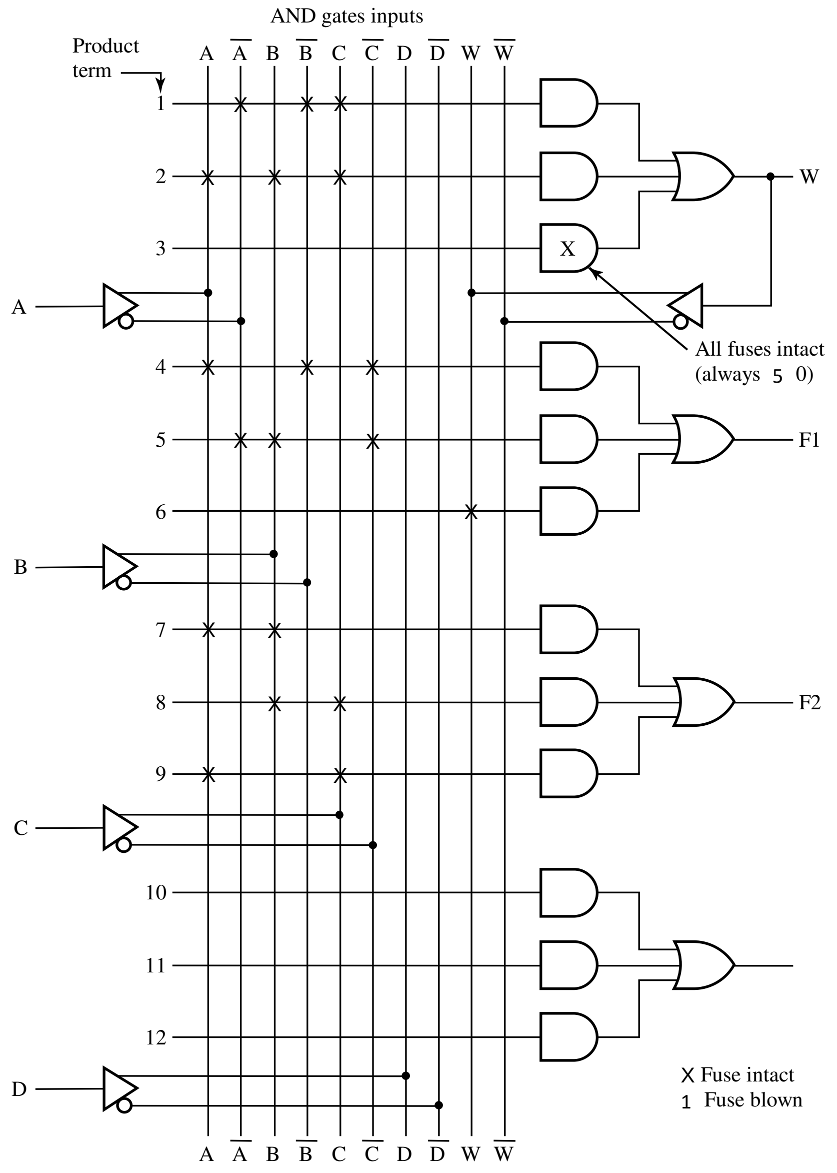
- There is no sharing of AND gates as in the ROM and PLA
- Design requires fitting functions within the limited number of ANDs per OR gate
- Single function optimization is the first step to fitting
- Otherwise, if the number of terms in a function is greater than the number of ANDs per OR gate, then factoring is necessary

Programmable Array Logic Example

- Equations: $\mathbf{F1} = A\overline{B}\,\overline{C} + \overline{A}B\overline{C} + \overline{A}\,\overline{B}C + ABC$
  $\mathbf{F2} = AB + BC + AC$
- F1 must be factored since four terms
- Factor out last two terms as W

Product term	AND Inputs A	B	C	D	W	Outputs
1	0	0	1	—	—	$W = \overline{A}\,\overline{B}C$
2	1	1	1	—	—	$+ ABC$
3	—	—	—	—	—	
4	1	0	0	—	—	$F1 = X = A\overline{B}\,\overline{C}$
5	0	1	0	—	—	$+ \overline{A}B\overline{C} + W$
6	—	—	—	—	1	
7	1	1	—	—	—	$F2 = Y$
8	—	1	1	—	—	$= AB + BC + AC$
9	1	—	1	—	—	
10	—	—	—	—	—	
11	—	—	—	—	—	
12	—	—	—	—	—	

Programmable Array Logic Example

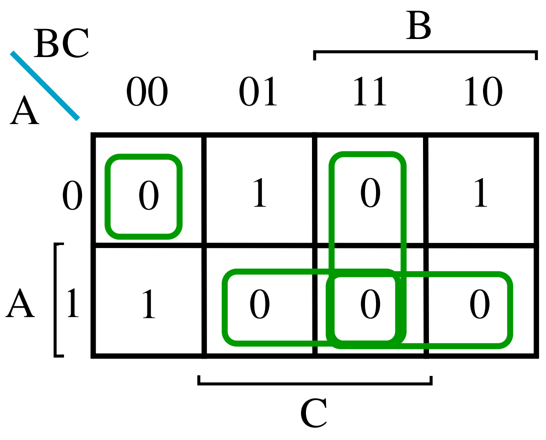


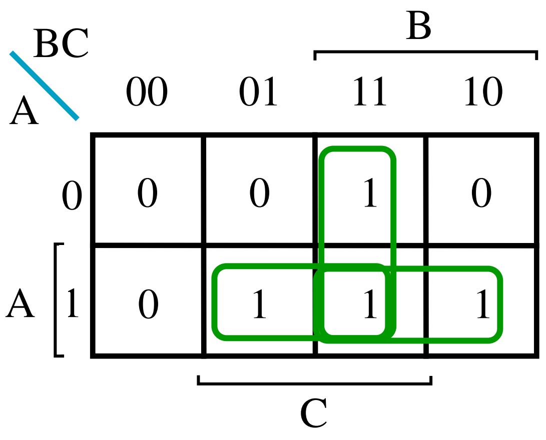
Programmable Logic Array

- The set of functions to be implemented must fit the available number of product terms
- The number of literals per term is less important in fitting
- The best approach to fitting is multiple-output, two-level optimization (which has not been discussed)
- Since output inversion is available, terms can implement either a function or its complement
- For small circuits, K-maps can be used to visualize product term sharing and use of complements
- For larger circuits, software is used to do the optimization including use of complemented functions

Programmable Logic Array Example

- K-map specification
- How can this be implemented with four terms?
- Complete the programming table





$F_1 = \overline{A}\,\overline{B}C + \overline{A}B\overline{C} + A\overline{B}\,\overline{C}$

$\overline{F_1} = AB + AC + BC + \overline{A}\,\overline{B}\,\overline{C}$

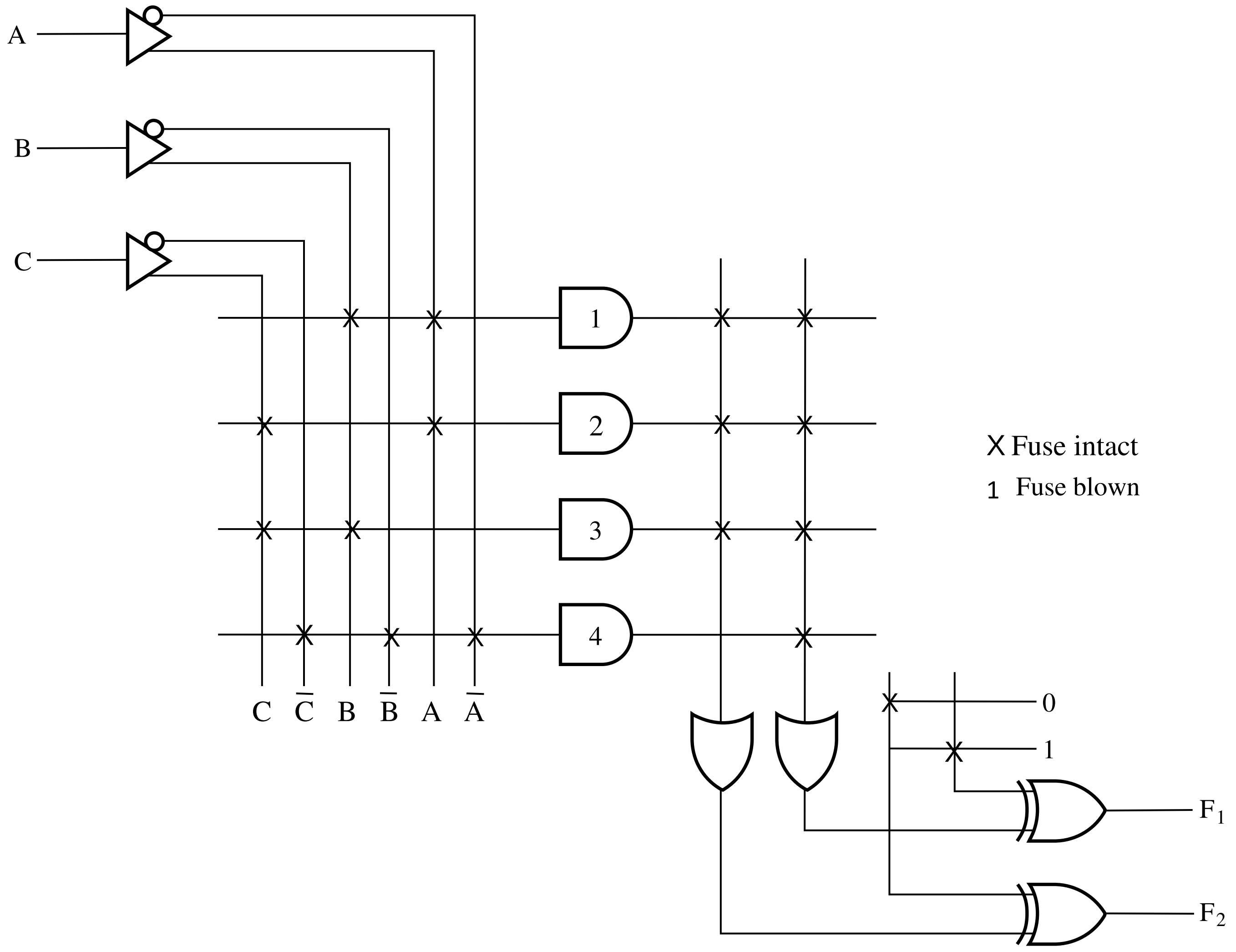
$F_2 = AB + AC + BC$

$\overline{F_2} = \overline{A}\,\overline{C} + \overline{A}\,\overline{B} + \overline{B}\,\overline{C}$

PLA programming table

	Product term	Inputs			Outputs	
		A	B	C	F_1	(T) F_2
AB	1	1	1	–		1
AC	2	1	–	1		1
BC	3	–	1	1		1
	4					–

Programmable Logic Array Example



Lookup Tables

- Lookup tables are used for implementing logic in Field-Programmable Gate Arrays (FPGAs) and Complex Logic Devices (CPLDs)
- Lookup tables are typically small, often with four inputs, one output, and 16 entries
- Since lookup tables store truth tables, it is possible to implement any 4-input function
- Thus, the design problem is how to optimally decompose a set of given functions into a set of 4-input two- level functions.
- We will illustrate this by a manual attempt

Lookup Table Example

- Equations to be implemented:
  $F_1(A,B,C,D,E) = A\,D\,E + B\,D\,E + C\,D\,E$
  $F_2(A,B,D,E,F) = A\,D\,E + B\,D\,E + F\,D\,E$
- Extract 4-input function:
  $F_3(A,B,D,E) = A\,D\,E + B\,D\,E$
  $F_1(C,D,E,F_3) = F_3 + C\,D\,E$
  $F_2(D,E,F,F_3) = F_3 + F\,D\,E$
- The cost of the solution is 3 lookup tables