Combinational Logic

2019. 4. 8.

K-S. Sohn

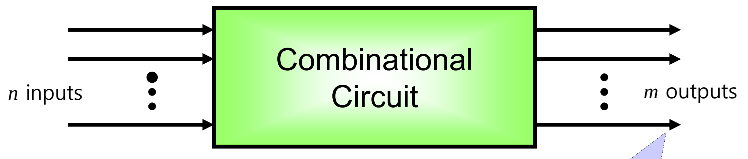
Contents

Introduction

- Logic circuits for digital systems may be combinational or sequential.
- A combinational circuit consists of logic gates whose outputs at any time are determined from only the present combination of inputs.

Definition of Combinational Circuit

- Block diagram



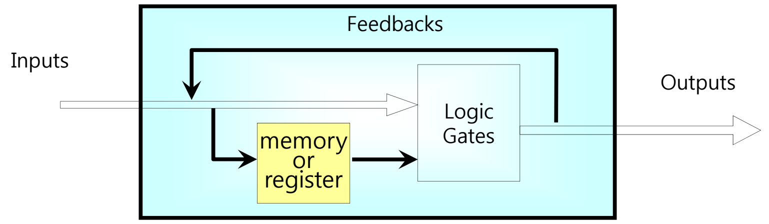
- Inputs
  - 2^n possible combinations of input values
- Outputs

OUTPUTs are determined from the PRESENT COMBINATION of INPUTs

- Examples
  - Adders, subtractor, comparator, multiplier, decoder, encoder, multiplexer, etc.

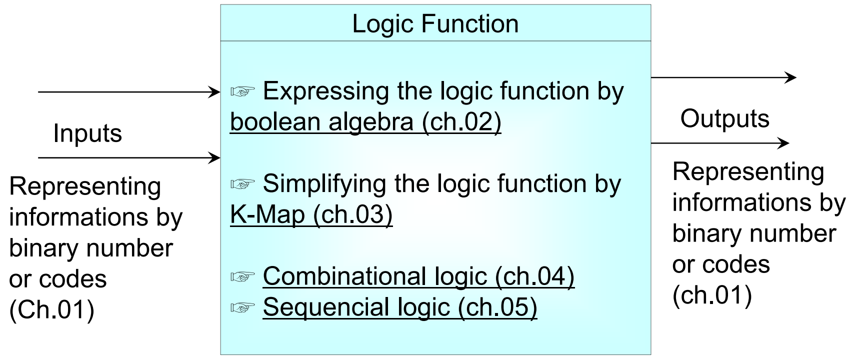
Difference from S.C.

- Definition of sequencial circuit (S.C.)
  - Outputs depends not only on the present input values but also on the past input values
- Essential components for S.C.
  - Memory, register, feedback
- Block diagram of S.C.



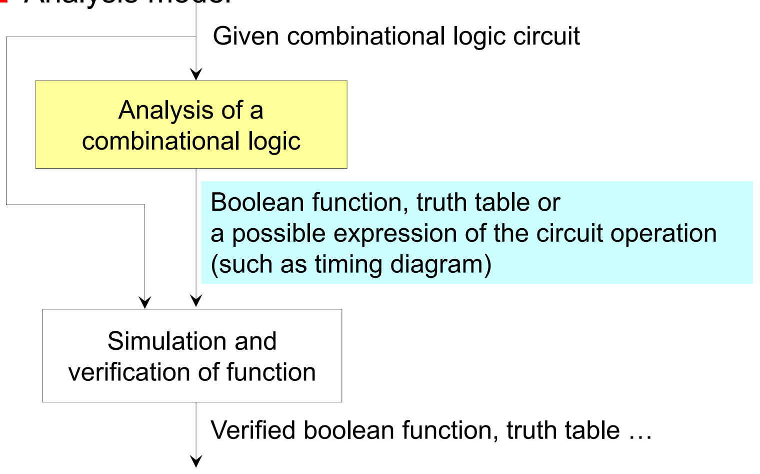
Knowledge of Digital System

- Knowledge map in a digital system


Logic Function
☞ Expressing the logic function by boolean algebra (ch.02)
☞ Simplifying the logic function by K-Map (ch.03)
☞ Combinational logic (ch.04)
☞ Sequencial logic (ch.05)
Inputs
Representing informations by binary number or codes (Ch.01)
Outputs
Representing informations by binary number or codes (ch.01)

Analysis of A Combinational Logic

- Analysis model


Given combinational logic circuit
Analysis of a combinational logic
Boolean function, truth table or a possible expression of the circuit operation (such as timing diagram)
Simulation and verification of function
Verified boolean function, truth table …

Analysis Steps

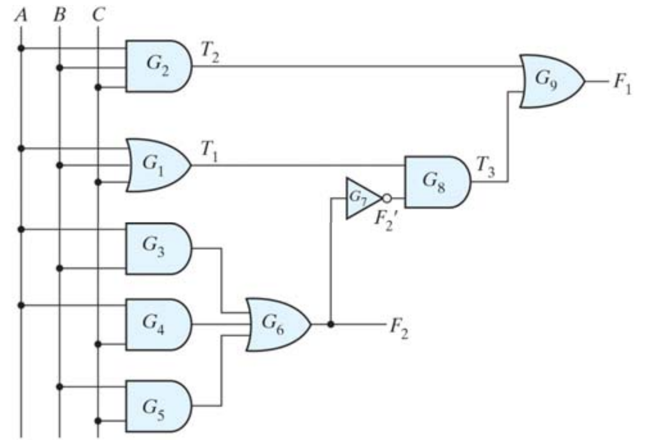
- Checking if it is a COMBINATIONAL CIRCUIT
  - No feedback paths
  - No memory or register elements
- Obtaining Boolean functions and truth table
  - Labeling, Boolean function at each gate level, ...
- Investigating the function of the circuit using:
  - Boolean function
  - Truth table

Obtaining Boolean Function

- Step-1. Labeling
  - all (inputs and outputs of the given circuit and) gate outputs
- Step-2. Boolean functions of gate outputs of the 1st stage
  - Boolean functions of input variables
- Step-3. Boolean functions of intermediate gate outputs
  - Function of input variables and the outputs from the previous stage
- Step-4. Repeat Step-3.
  - Until all output variables are determined.
- Step-5. Substitution of Boolean functions
  - Obtain Boolean functions of output variables in terms of input variables

Obtaining Boolean Function

- Given combinational circuit



Obtaining Boolean Function

- F_1 and F_2

  Step-1. Labeling
  -. input variables : A,B,C
  -. circuit outputs : F_1, F_2
  -. gate outputs : T_1, T_2, T_3
  Step-2. Boolean function of gate outputs of the 1st stage
  -. $T_1 = ABC$
  -. $T_2 = A+B+C$
  -. $F_2 = AB + AC + BC$
  Step-3.,4. Boolean function of gate outputs of intermediate stages
  -. $T_3 = F'_2T_2$
  -. $F_1 = T_3 + T_1$
  Step-5.
  -. $F_1 = F'_2 + T_2 + T_1 = (AB+AC+BC)'(A+B+C) + ABC$
  $= A'BC' + A'B'C + AB'C' + ABC$
  -. $F_2 = AB + AC + BC$

Obtaining Truth Table

- Obtaining the truth table from the combinational circuit
  - Step-1. Determine the number of input variables
  - Step-2. Label the outputs of selected gates
  - Step-3. Obtain the truth table for the outputs that are a function of input variables only
  - Setp-4. Proceed to obtain the truth table for the outputs that are a function of the previously defined values

Obtaining Truth Table

step-2

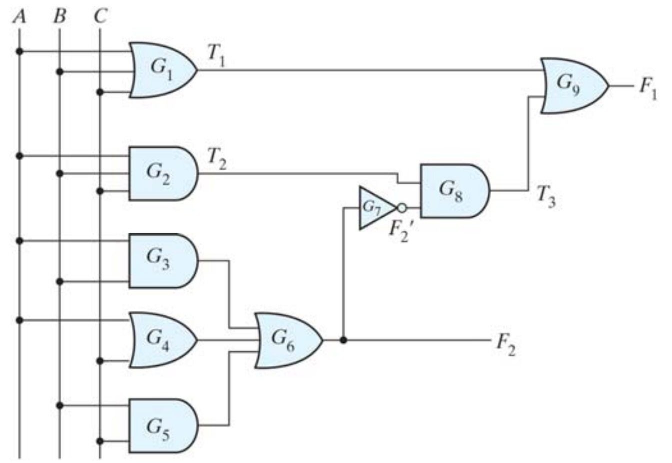
A	B	C	F_2	F'_2	T_1	T_2	T_3	F_1
0	0	0	0	1	0	0	0	0
0	0	1	0	1	1	0	1	1
0	1	0	0	1	1	0	1	1
0	1	1	1	0	1	0	0	0
1	0	0	0	1	1	0	1	1
1	0	1	1	0	1	0	0	0
1	1	0	1	0	1	0	0	0
1	1	1	1	0	1	1	0	1

step-4

step-1 step-3

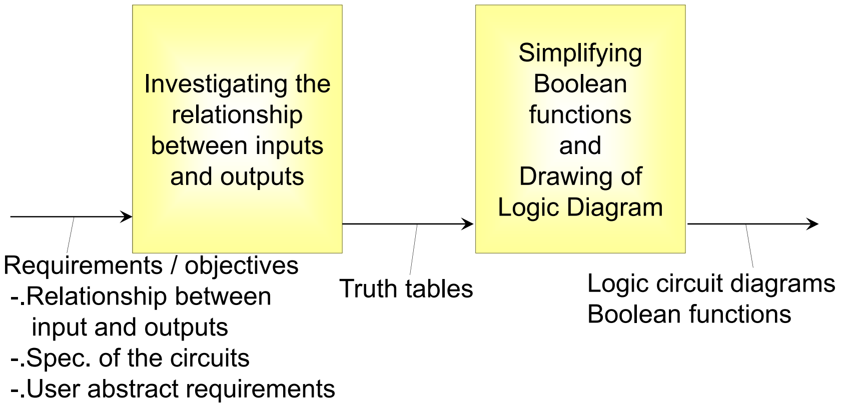
Obtaining Boolean Function (P.E.)

- Find the Boolean expressions for F_1 and F_2



Design Procedure of Combinational Circuits

- Overview



Investigating the relationship between inputs and outputs
Simplifying Boolean functions and Drawing of Logic Diagram
Requirements / objectives
-.Relationship between input and outputs
-.Spec. of the circuits
-.User abstract requirements
Truth tables
Logic circuit diagrams
Boolean functions

Design Steps

- Step-1. Determine the number of input and output variables
  - the input and output variable are assigned symbols(or labels)
- Step-2. Derive the truth table
- Step-3. Obtain the Boolean functions
  - Simplification with algebraic manipulation, K-Map,…
- Step-4. Draw the logic circuit diagram
  - verify the correctness

Design Methods

- Truth table

n-input variables	outputs
2^n binary numbers obtained from the combinations of inputs	Determined by the stated specifications and described as the canonical formed function of inputs

Design Methods

- Functional description
  - Boolean function
  - trueth table
  - HDL
  - logic diagram
- Logic minimization objectives
  - number of gates
  - number of inputs to a gate
  - propagation delay
  - number of interconnection
  - limitations of the driving capabilities

Code Converter

- Needs of code converter
  - Representing the same information with different codes at different digital systems (e.g., BCD, Excess-3, 2421, 84-2-1)
  - Code conversion between systems that use different codes for compatibility and information exchange.

Design BCD to Excess-3 Code Converter

- Step-1. Determination of input and output variables
  - Input variables : BCD code bits (A,B,C,D)
  - Output variables : Excess-3 code bits (w,x,y,z)
- Step-2. Obtaining truth table
  - Mapping BCD code values to Excess-3 code values
  - 6 don't care combinations
- Step-3. Obtaining Boolean functions
  - K-map simplification for each output functions
- Step-4. Drawing logic circuit diagram
  - Manipulating Boolean functions for the purpose of using common gates for more than 2 outputs

Design BCD to Excess-3 Code Converter

- Truth table specifying the objectives of the code converter

Input BCD				Output Excess-3 Code				Decimal
A	**B**	**C**	**D**	**w**	**x**	**y**	**z**	
0	0	0	0	0	0	1	1	0
0	0	0	1	0	1	0	0	1
0	0	1	0	0	1	0	1	2
0	0	1	1	0	1	1	0	3
0	1	0	0	0	1	1	1	4
0	1	0	1	1	0	0	0	5
0	1	1	0	1	0	0	1	6
0	1	1	1	1	0	1	0	7
1	0	0	0	1	0	1	1	8
1	0	0	1	1	1	0	0	9

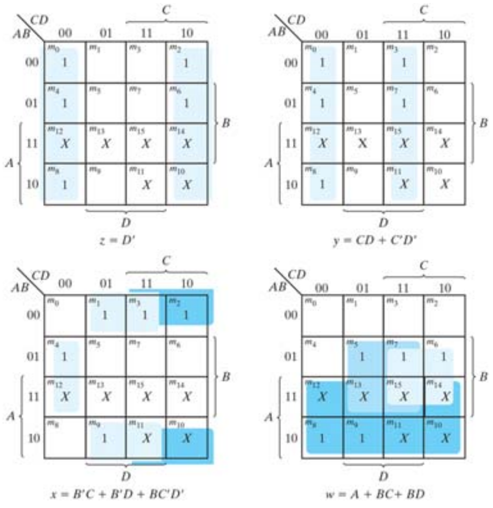
Remind

- Ch.01 binary codes for decimal digits

Decimal Digit	BCD 8421	2421	Excess-3	8, 4, −2, −1
0	0000	0000	0011	0000
1	0001	0001	0100	0111
2	0010	0010	0101	0110
3	0011	0011	0110	0101
4	0100	0100	0111	0100
5	0101	1011	1000	1011
6	0110	1100	1001	1010
7	0111	1101	1010	1001
8	1000	1110	1011	1000
9	1001	1111	1100	1111
	1010	0101	0000	0001
Unused bit combi-nations	1011	0110	0001	0010
	1100	0111	0010	0011
	1101	1000	1101	1100
	1110	1001	1110	1101
	1111	1010	1111	1110

Design BCD to Excess-3 Code Converter

- Maps for BCD-to-excess-3 code converter



$$w = A + BC + BD$$

$$x = B'C + B'D + BC'D'$$

$$y = CD + C'D'$$

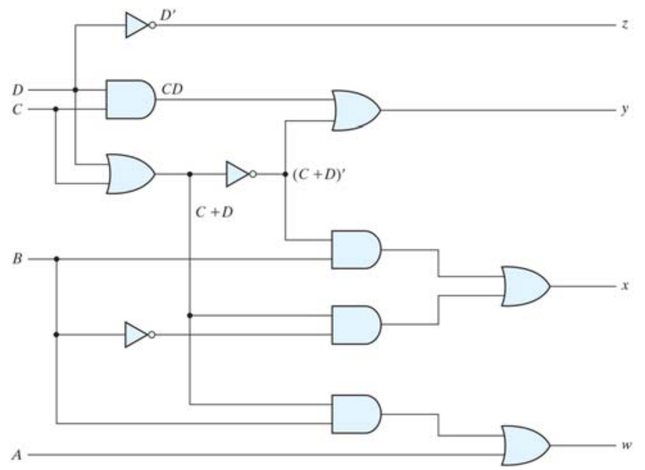
$$z = D'$$

Design BCD to Excess-3 Code Converter

- The simplified functions (standard form, 2-level circuit)
  - z = D'
  - y = CD +C'D'
  - x = B'C + B'D+BC'D'
  - w = A+BC+BD
- Another implementation (multilevel circuit)
  - z = D'
  - y = CD +C'D' = CD + (C+D)'
  - x = B'C + B'D+BC'D'= B'(C+D) +B(C+D)'
  - w = A+BC+BD

Design BCD to Excess-3 Code Converter

- Logic diagram for BCD-to-excess-3 code converter

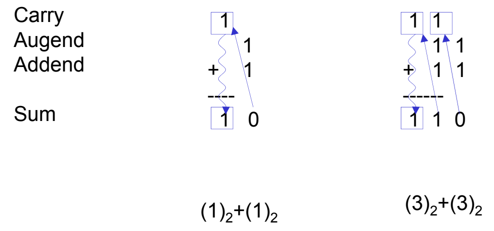


Binary Adder

- Addition of 1 bit numbers
  - 0+0=0, 0+1=1, 1+0=1, 1+1=10 ⇒ 1+1=0 with a carry
- Half adder
  - Addition of only 2 significant bits
- Full adder
  - Addition of 2 significant bits and a carry
- Addition of multi-bit numbers
  - Add bits of the same order
  - Carry is added to addition of the next higher order bits

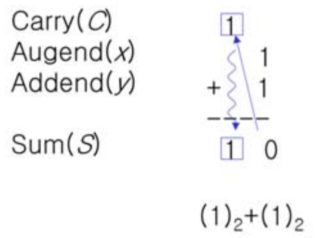
Binary Adder

- Bit addition examples



Design of Half Adder

- Requirement analysis

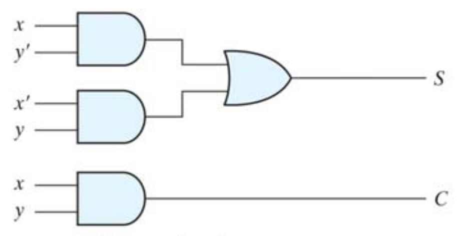


  - two input variable: x, y
  - two output variables: C, S
  - truth table ⇒

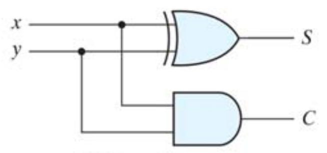
x	y	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Design of Half Adder

- Boolean function of outputs
  - S = x'y + xy' = x⊕y
  - C = xy
- Logic diagrams



(a) $S = xy' + x'y$
$C = xy$



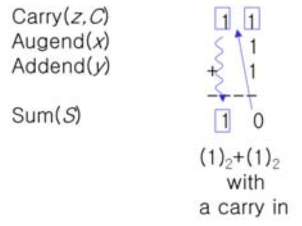
(b) $S = x \oplus y$
$C = xy$

Design of Half Adder

- for implementation flexibility
  - S = (x+y)(x'+y')
  - S' = xy + x'y'
  - S = (xy + x'y')' = (C + x'y')'
  - C = xy = (x' + y')'

Design of Full Adder

- Requirements

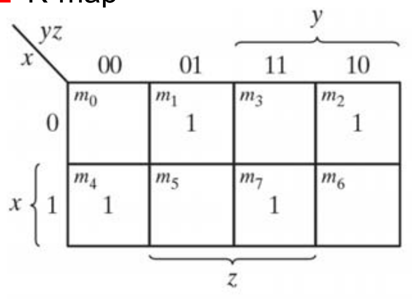


x	y	z	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

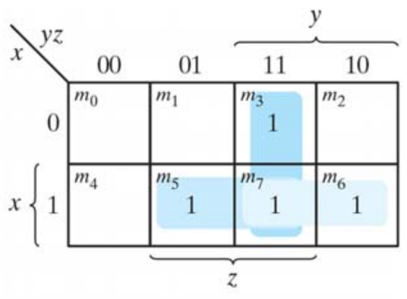
  - three input variables: x, y, and z(carry)
  - two outputs: C, S
  - truth table ⇒

Design of Full Adder

- K-map



(a) $S = x'y'z + x'yz' + xy'z' + xyz$

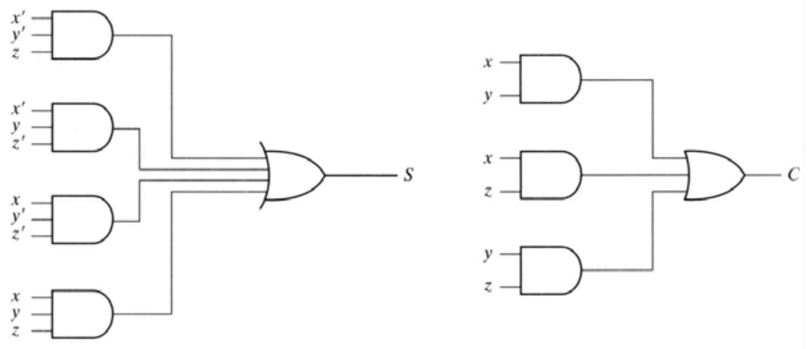


(b) $C = xy + xz + yz$

- Boolean functions
  - S=x'y'z+x'yz'+xy'z'+xyz
  - C=xy+xz+yz

Design of Full Adder

- 2-level implementation (sum-of-product form)



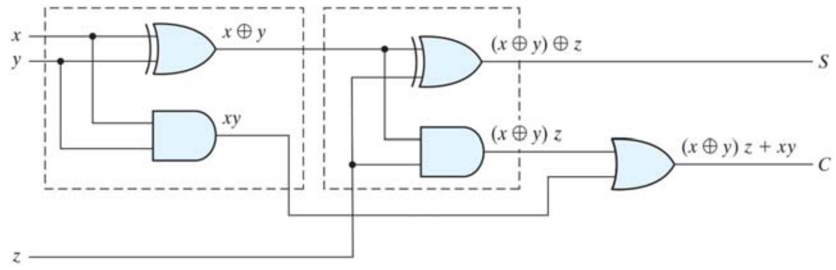
Design of Full Adder

- Modifying Boolean function

$$S = xy'z' + x'yz' + xyz + x'y'z$$
$$= z'(x'y + xy') + z(x'y + xy')'$$
$$= z'(x \oplus y) + z(x \oplus y)'$$
$$= z \oplus (x \oplus y)$$
$$C = xy + xz + yz$$
$$= xy + (y + y')xz + (x + x')yz = xy + x'yz + xy'z$$
$$= z(x \oplus y) + xy$$

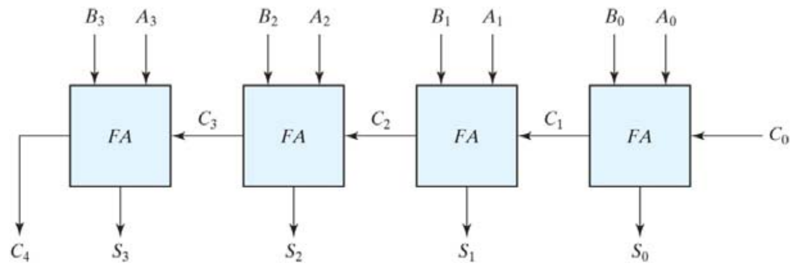
Design of Full Adder

- Implementation with two half adders and an OR gates



Binary Adder

- Example: 4-bit adder
  - connection of 4 full adders



  - or connection of 3 full adders and a half adder

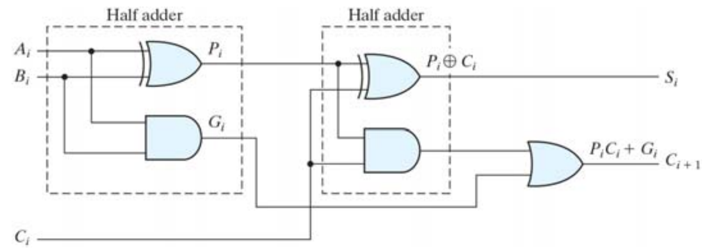
Design of n-Bit Binary Adder

- Example: 4-bit adder
  - Calcluate 11 + 3

Subscript *i*:	**3**	**2**	**1**	**0**	
Input carry	0	1	1	0	C_i
Augend	1	0	1	1	A_i
Addend	0	0	1	1	B_i
Sum	1	1	1	0	S_i
Output carry	0	0	1	1	C_{i+1}

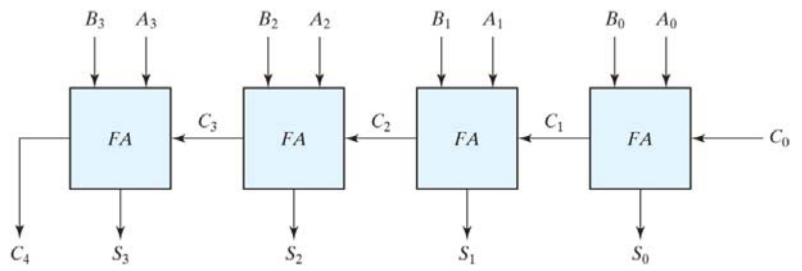
Carry Propagation Problems

- Carry propagation delay in the binary adder
  - The longest propagation delay time in an adder is the time it takes the carry to propagate through the full adders
- Number of gate levels for the carry propagation
  - # of gate levels for each bit = 2 gate-levels
  - n-bits binary adder ⇒ 2n gate-levels

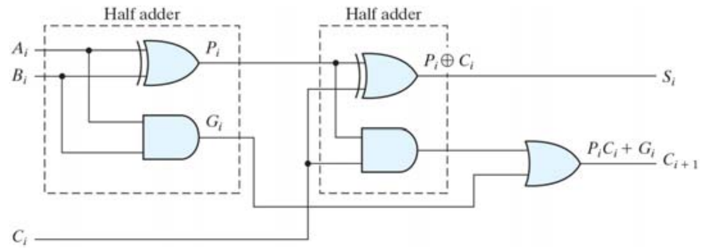


Carry Propagation Problem

- Carry propagation in 4-bit adder
  - when the correct outputs are available
  - the critical path counts (the worst case)
  - $(A_0,B_0,C_0) > C_1 > C_2 > C_3 > (C_4,S_3)$
  - > 8 gate levels



Propagation in an Full Adder



- Boolean expression
  - The 1st H.A. stage $P_i = A_i \oplus B_i, \quad G_i = A_iB_i$
  - The 2nd H.A. stage $S_i = P_i \oplus C_i, \quad C_{i+1} = G_i + P_iC_i$
- Characteristics of intermediate outputs
  - G_i carry generate - produces a carry of 1 when both A_i and B_i are 1.
  - P_i carry propagate - controls the propagation of the carry from C_i to C_{i+1}

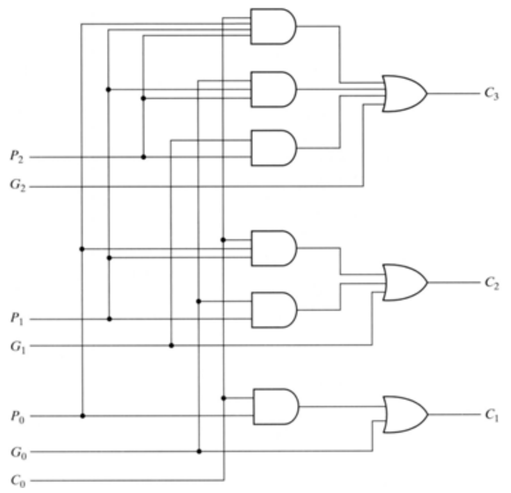
Analysis on 4-bit Binary Adder

- Boolean functions for each carry

$$C_0 = \text{input carry}$$
$$C_1 = G_0 + P_0C_0$$
$$C_2 = G_1 + P_1C_1 = G_1 + P_1(G_0 + P_0C_0) = G_1 + P_1G_0 + P_1P_0C_0$$
$$C_3 = G_2 + P_2C_2 = G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0C_0$$

  - augend : A0, A1, A2, A3
  - addend : B0, B1, B2, B3
  - initial carry :C0
  - ⇒ Each carry can be obtained independently of the carries from the previous digits ⇒ carry lookahead

Carry Lookahead Generator



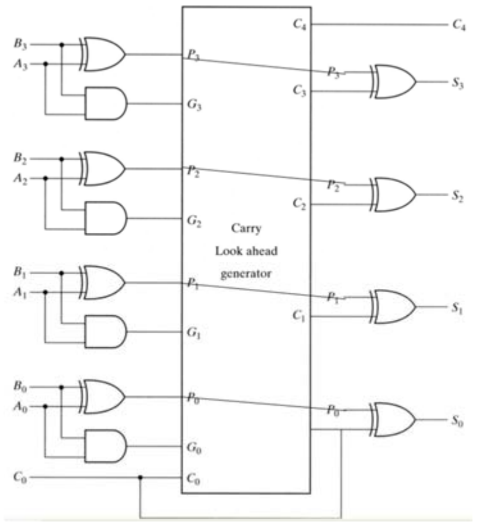
$$C_3 = G_2 + P_2C_2$$
$$= G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0C_0$$

$$C_2 = G_1 + P_1C_1$$
$$= G_1 + P_1(G_0 + P_0C_0)$$
$$= G_1 + P_1G_0 + P_1P_0C_0$$

$$C_1 = G_0 + P_0C_0$$

$$C_0 = \text{input carry}$$

Binary Adder witn Carry Lookahead



Constant 2 gate-level delay regardless to the number of bits of augend and addend

Binary Subtractor

- Requirements
  - minuend + subtrahend(2's complement)
- 1's complement by a XOR gate
  - 1 y = y' → inverter
  - 0 y = y → buffer
- 2's complement = 1's complement + 1
- Subtraction
  - 1's complement by XOR gates (M=1) and plus by a input carrry

Overflow

- Why overflow?
  - The storage is limited
  - Limit of the ability to represent the number in digital computer. ⇒ Limit of the number of digits (in terms of bits).
- What overflow?
  - The result of calculation is the number of (n+1)-bits while the computer has only n-bit registers to hold the number.
- When overflow?
  - Positive number + positive number ⇒ negative number
  - Negative number + negative number ⇒ positive number

Overflow

- How to detect the overflow?
  - The carry into the sign bit position and the carry out of the sign bit position are different. ⇒ Overflow condition

Overflow

- Example
  - Positive + positive ⇒ Inversed sign bit by the carry from the MSB

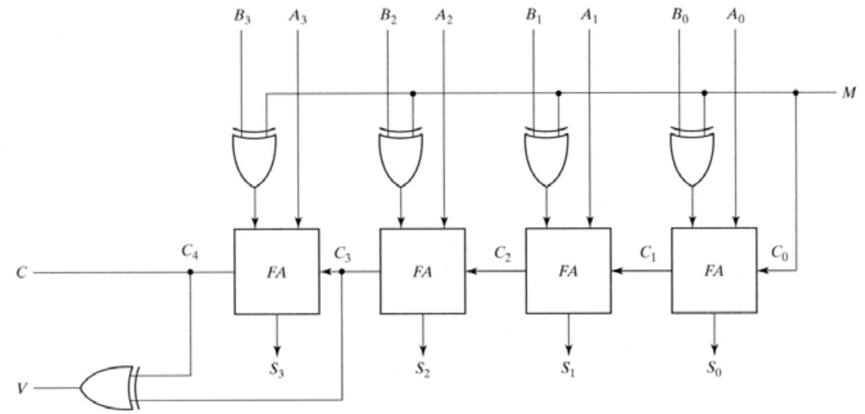
carries:	0 1
+70	0 1000110
+80	0 1010000
+150	1 0010110

  - Negative + negative ⇒ Inversed sign bit by absence of carry from the MSB

carries:	1 0
−70	1 0111010
−80	1 0110000
−150	0 1101010

4-Bit Adder-Subtractor

- M=1 ⇒ subtractor(A+B'+1), M=0 ⇒ adder(A+B)

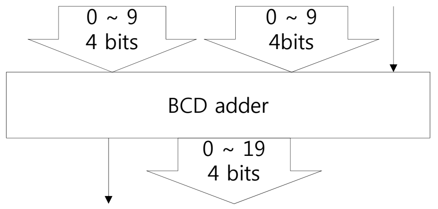


Decimal Adder

- Error between real life and binary world
  - 1/10 = 0.1
  - 0.099999 due to conversion from binary to decimal
  - Will you accept $0.09999 or $0.1 (i.e., 10 cents)?
- Many calculator (HW) or money dealing SW use decimal numbers in binary coded form

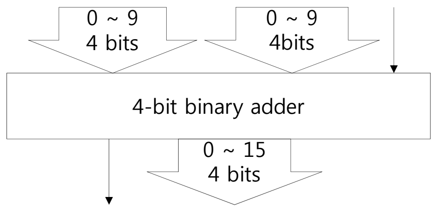
1-Digit BCD Adder

- One BCD digit + one BCD digit
  - 9 bit inputs: two BCD digits and one carry-in
  - each input BDC digit ≤ 9
  - 5 bit outputs: one BCD digit and one carry-out
  - output BCD digit ≤ 19 = 9 + 9 + 1
- One digit BCD adder

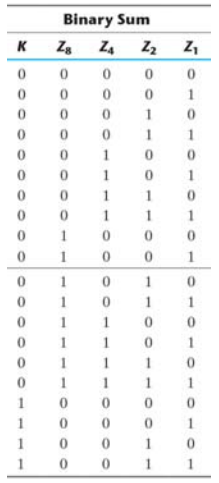


BCD Adder Based on Binary Adder

- Implement with 4bit binary adders
  - Single layered binary adder

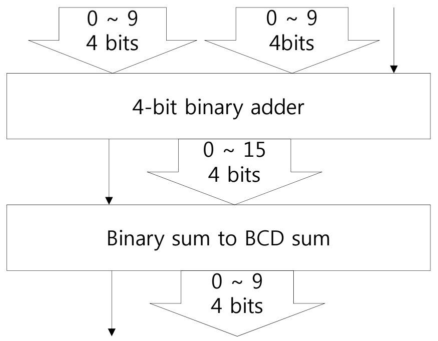


  - Check the binary (not BCD) sum output >>

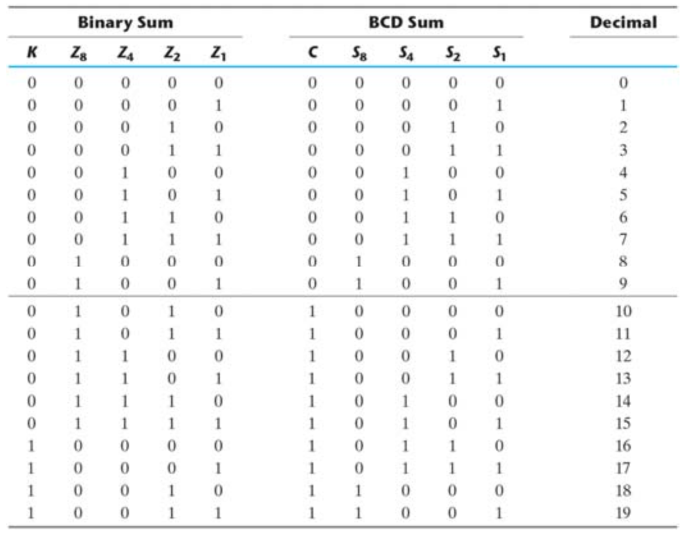


Binary Sum				
K	Z_8	Z_4	Z_2	Z_1
0	0	0	0	0
0	0	0	0	1
0	0	0	1	0
0	0	0	1	1
0	0	1	0	0
0	0	1	0	1
0	0	1	1	0
0	0	1	1	1
0	1	0	0	0
0	1	0	0	1
0	1	0	1	0
0	1	0	1	1
0	1	1	0	0
0	1	1	0	1
0	1	1	1	0
0	1	1	1	1
1	0	0	0	0
1	0	0	0	1
1	0	0	1	0
1	0	0	1	1

Binary sum to BCD sum



Binary vs. BCD

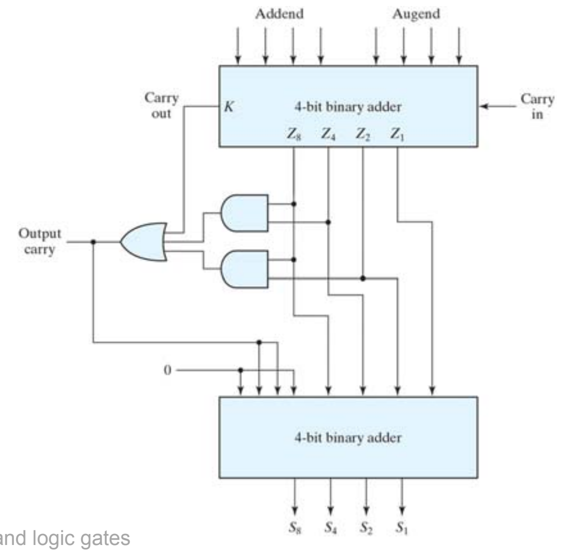


Binary Sum					BCD Sum					Decimal
K	Z_8	Z_4	Z_2	Z_1	C	S_8	S_4	S_2	S_1	
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	2
0	0	0	1	1	0	0	0	1	1	3
0	0	1	0	0	0	0	1	0	0	4
0	0	1	0	1	0	0	1	0	1	5
0	0	1	1	0	0	0	1	1	0	6
0	0	1	1	1	0	0	1	1	1	7
0	1	0	0	0	0	1	0	0	0	8
0	1	0	0	1	0	1	0	0	1	9
0	1	0	1	0	1	0	0	0	0	10
0	1	0	1	1	1	0	0	0	1	11
0	1	1	0	0	1	0	0	1	0	12
0	1	1	0	1	1	0	0	1	1	13
0	1	1	1	0	1	0	1	0	0	14
0	1	1	1	1	1	0	1	0	1	15
1	0	0	0	0	1	0	1	1	0	16
1	0	0	0	1	1	0	1	1	1	17
1	0	0	1	0	1	1	0	0	0	18
1	0	0	1	1	1	1	0	0	1	19

Binary vs. BCD

- Conceptually
  - 0 ~ 9: <binary sum> = <BCD sum>
  - 10 ~ 19: <binary sum> + 6 = <BCD sum> with a carry-out
- How to detect that the <binary sum> exceeds 9
- Look at the truth table
  - IF K==1 OR Z_8 == 1 AND (Z_4 == 1 OR Z_2 == 1) THEN C = 1
  - That is, $C = K + Z_8Z_4 + Z_8Z_2$
  - IF C == 1 THEN <BCD sum> = <binary sum> + 6

1-Digit BCD Adder(Impl'ted)



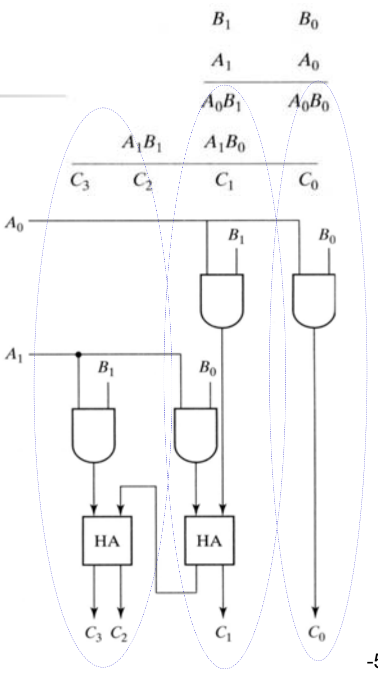
Binary Multiplier

- Human's binary multiplication
  - Simple case of 2-bit multiplying: C = B × A

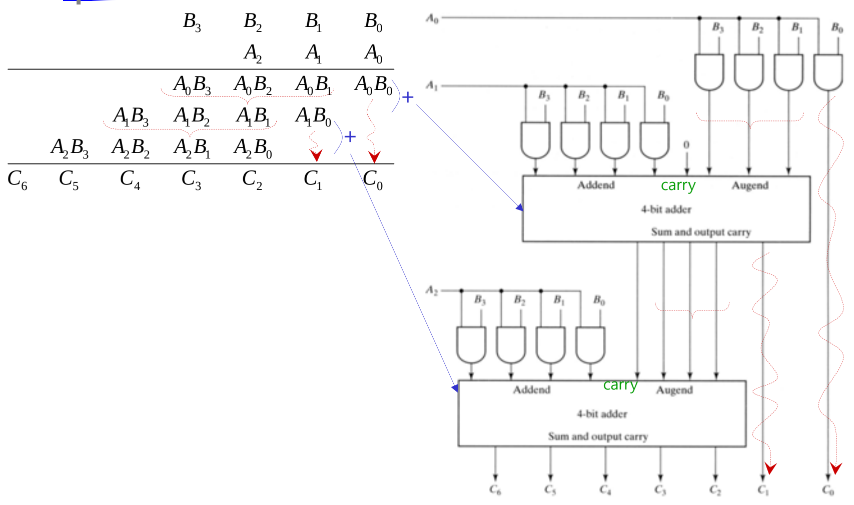
$$\begin{array}{cccc} & & B_1 & B_0 \\ & & A_1 & A_0 \\ \hline & & A_0B_1 & A_0B_0 \\ & A_1B_1 & A_1B_0 & \\ \hline C_3 & C_2 & C_1 & C_0 \end{array}$$

Binary Multiplier

- Mimicing with digital logic



4-bit by 3-bit Binary Multiplier



B_3 B_2 B_1 B_0
A_2 A_1 A_0
A_0B_3 A_0B_2 A_0B_1 A_0B_0
A_1B_3 A_1B_2 A_1B_1 A_1B_0
A_2B_3 A_2B_2 A_2B_1 A_2B_0
C_6 C_5 C_4 C_3 C_2 C_1 C_0
+
Addend
carry
Augend
4-bit adder
Sum and output carry

K-bit by J-bit Binary Multiplier

- K-bit multiplicand × J-bit multiplier
  - J*K AND gates
  - (J-1) K-bit adders
  - (J+K)-bit product
- In the example of K=4, J=3
  - 12 AND gates
  - Two 4-bit adders
  - 7-bit product

Magnitude Comparator

- The comparison of two numbers
  - outputs: A>B, A=B, A<B
- Design Approaches
  - the truth table
    - 2^{2n} entries - too cumbersome for large n
  - use inherent regularity of the problem
    - reduce design efforts
    - reduce human errors

Magnitude Comparator

- Algorithm

```
for (i=n-1; i>=0; i--) {
    if ( a[i] == b[i] ) {
        AeqB[i] = 1;  // equal significant bit list
    elseif ( a[i] ==1 ) { // a[i] ≠ b[i], a[i] ==1, b[i]==0
        AgtB = 1; AeqB[i] = 0; break;
    else // a[i] ≠ b[i], a[i] ==1, b[i]==0
        AgtB = 0; AeqB[i] = 0; break;
}
// we can know the break even point by checking
// AeqB[] and AgtB
```

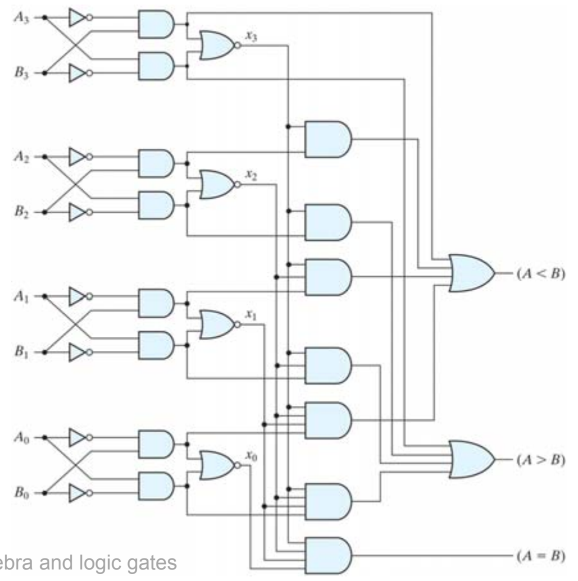
Magnitude Comparator

- Comparing 2 n-bit binary numbers : A, B
  - $A=(A_0,A_1,\ldots,A_{n-1})$, $B=(B_0,B_1,\ldots,B_{n-1})$
- Bit-by-bit comparation
  - Input binary numbers (n-bits): A, B
  - Equality (A=B) : $A_i=B_i$ for i=0,1,…, n-1
  - Inequality
    - Compare bits from MSB bit position
    - Compare until two bits are not equal
    - If $A_i \neq B_i$, $A_i=1$, and $B_i=0$ then A>B, where i=0,1,…, or n-1
    - if $A_i \neq B_i$, $A_i=0$, $B_i=1$ then A<B, where i=0,1, …, or n-1

4-bit Magnitude Comparator

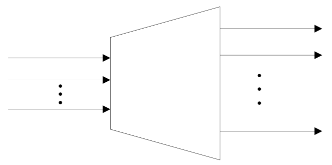
- Algorithm to logic
  - $A = A_3A_2A_1A_0$; $B = B_3B_2B_1B_0$
  - A=B if $A_3=B_3$, $A_2=B_2$, $A_1=B_1$and $A_1=B_1$
  - equality: $x_i= A_iB_i+A_i'B_i'$, for i = 0, 1, 2, 3
  - $(A=B) = x_3x_2x_1x_0$
  - $(A>B) = A_3B_3'+x_3A_2B_2'+x_3x_2A_1B_1'+x_3x_2x_1A_0B_0'$
  - $(A>B) = A_3'B_3+x_3A_2'B_2+x_3x_2A_1'B_1+x_3x_2x_1A_0'B_0$
- Implementation
  - $xi = (A_iB_i'+A_i'B_i)'$

4-bit Magnitude Comparator



Decoders

- Definition
  - Converts a given input code to a unique output line's assertion
  - Value of n input lines ⇒ One of 2^n output lines
  - Each output represents one of 2^n minterms



- Use of decoders
  - Switch, demultiplexer
  - Implementation of Boolean functions: $F = \sum_i(m_i)$

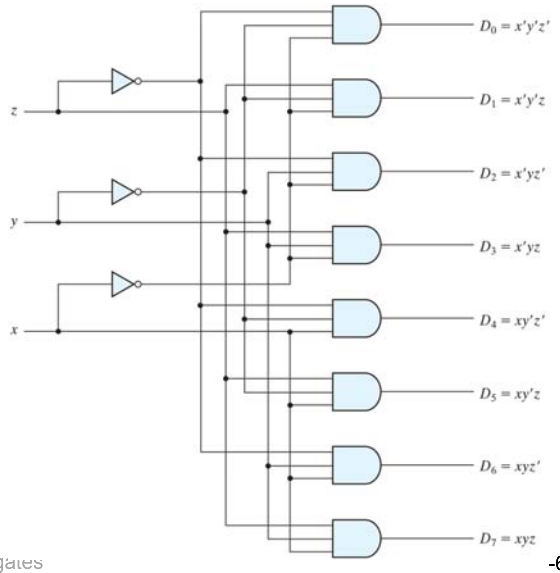
3-to-8 Line Decoder

- Truth table

Inputs			Outputs							
x	y	z	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

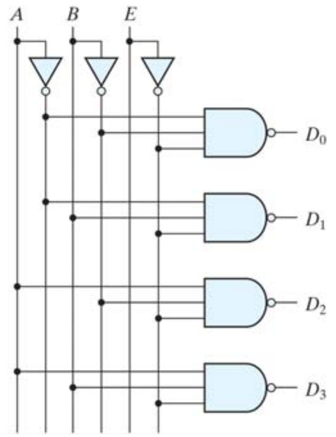
3-to-8 Line Decoder

- Implementation



2-to-4 Decoder NAND Implementation

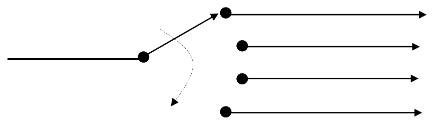
- Decoding minterms in complement form



E	A	B	D_0	D_1	D_2	D_3
1	X	X	1	1	1	1
0	0	0	0	1	1	1
0	0	1	1	0	1	1
0	1	0	1	1	0	1
0	1	1	1	1	1	0

Demultiplexer from Decoder

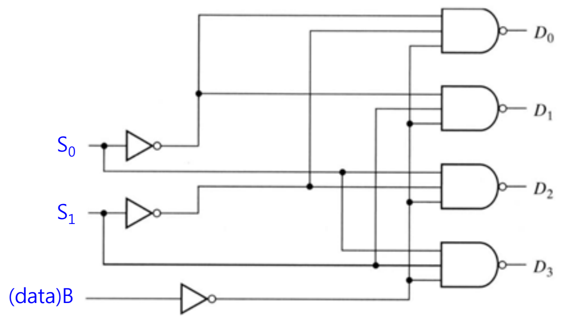
- Definition
  - Distribute information from a single line to one of 2^n possible output lines



- Decoder with enable $(E) \Rightarrow$ Demultiplexer
  - Input lines $\Rightarrow$ Selection lines
  - Enable line $\Rightarrow$ Input information line

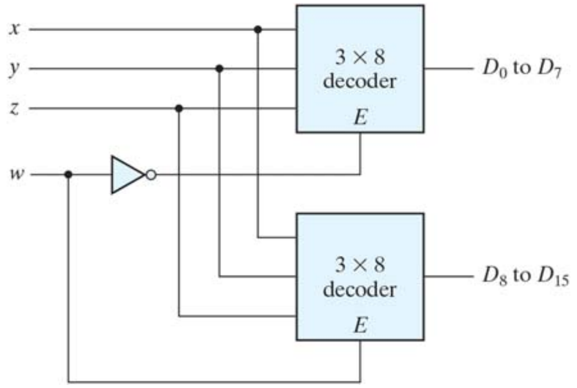
Demultiplexer

- 1×4 DeMux
  - a decoder with selection lines and a data line



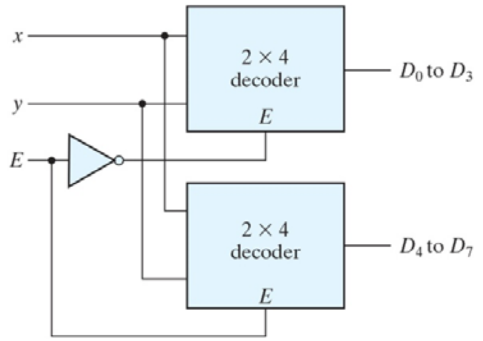
Decoder Expansion

- Construct 4×16 decoder with two 3×8 decoders



Decoder Expansion

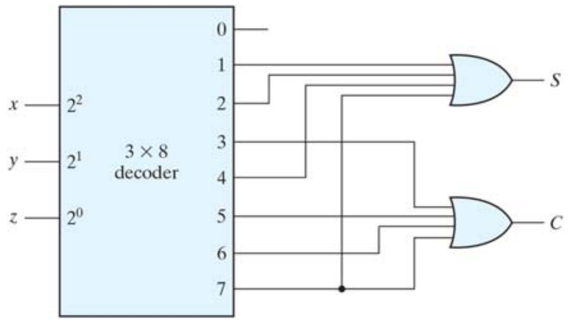
- P.E.
  - Verification with a smaller case



E	x	y	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

Combinational Logic Implementation

- Sum of minterm
  - use a decoder and an external OR gate to implement any Boolean function of n-input variables
- Ex. full-adder
  - S(x,y,z) = ∑(1,2,4,7)
  - C(x,y,z) = ∑(3,5,6,7)



Combinational Logic Implementation

- Two possible approaches using decoder
  - OR(minterms of F): k inputs
  - NOR(minterms of F'): 2^n-k inputs
- In general,
  - it is not a practical implementation

Encoders

- Definition
  - 2^n input lines and n output lines
  - The inverse function of a decoder
- Example : Octal-to-binary encoder

Inputs								**Outputs**		
D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7	x	y	z
1	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	0	0	1	0	0	0	0	0	1	1
0	0	0	0	1	0	0	0	1	0	0
0	0	0	0	0	1	0	0	1	0	1
0	0	0	0	0	0	1	0	1	1	0
0	0	0	0	0	0	0	1	1	1	1

Encoders

- Octal-to-binary encoder
  - Output Boolean functions

$$z = D_1 + D_3 + D_5 + D_7$$
$$y = D_2 + D_3 + D_6 + D_7$$
$$x = D_4 + D_5 + D_5 + D_7$$

- Ambiguity in the encoder
  - Multiple active inputs
    - Ex.: If D3 and D6 are active simultaneously, output is 111
    - Solution: establish priority among inputs
  - Multiple identical outputs
    - All inputs are 0 ⇒ All outputs are 0
    - Solution: all-zero-indicator

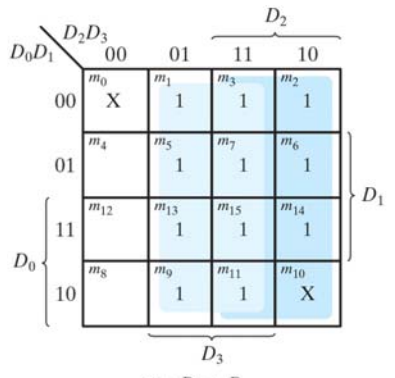
Priority Encoder

- Requirements
  - resolve the ambiguity of illegal inputs
  - only one of the input is encoded
- Design
  - Priority (H to L): D_3, D_2, D_1, D_0

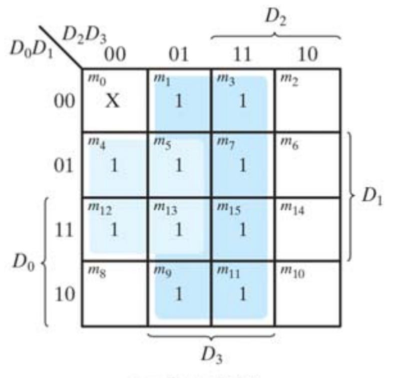
Inputs				Outputs		
D_0	D_1	D_2	D_3	x	y	V
0	0	0	0	X	X	0
1	0	0	0	0	0	1
X	1	0	0	0	1	1
X	X	1	0	1	0	1
X	X	X	1	1	1	1

Priority Encoder

- The maps for x and y (how to get these?)



$x = D_2 + D_3$



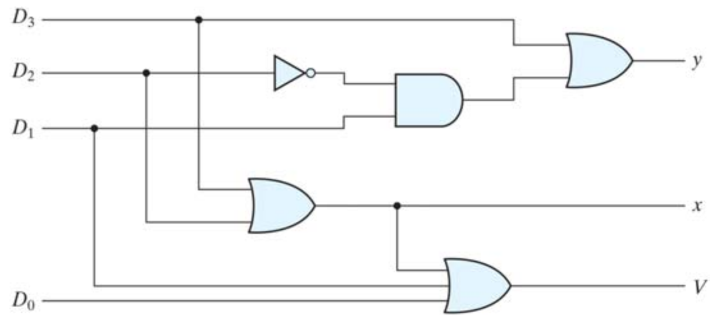
$y = D_3 + D_1D'_2$

Priority Encoder

- Boolean expressions

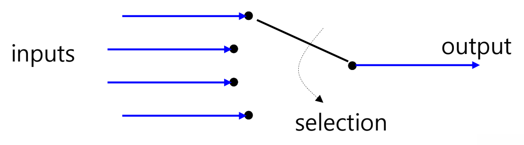
$$x = D_2 + D_3$$
$$y = D_3 + D_1D_2'$$
$$V = D_1 + D_2 + D_3 + D_4$$

- Implementation

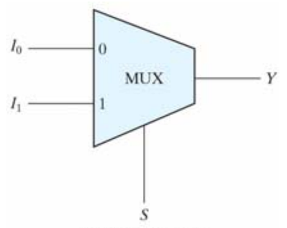


Multiplexers

- Definition
  - select binary information from one of many input lines and direct it to a single output line



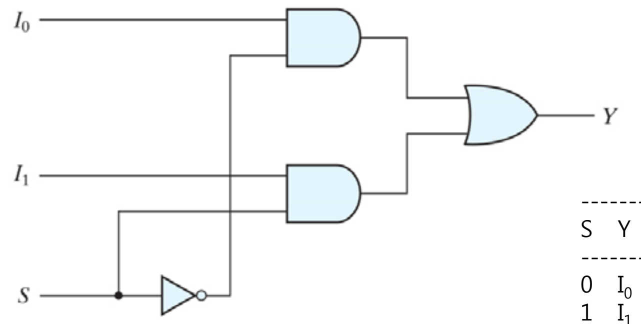
- 2^n input lines,
- n selection lines
- and one output line



(b) Block diagram

Multiplexers

- 2-to-1 multiplexer



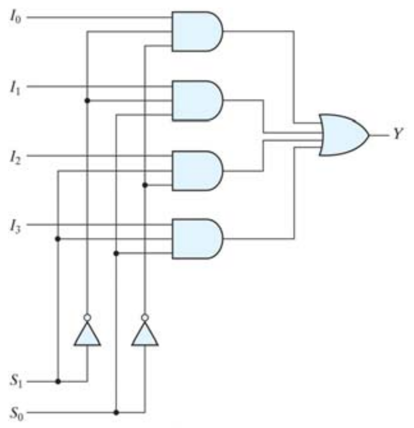
(a) Logic diagram

S	Y
0	I_0
1	I_1

Function table

Multiplexers

- 4-to-1 multiplexer



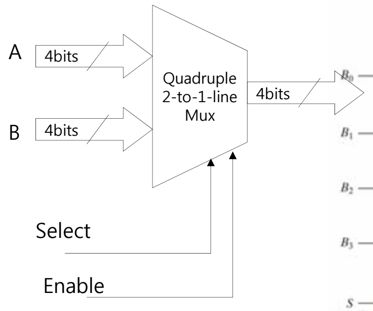
(a) Logic diagram

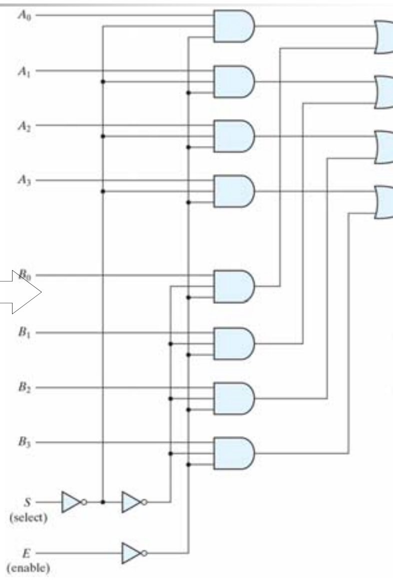
S_1	S_0	Y
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

(b) Function table

Multiplexers

- Quadruple 2-to-1 multiplexer



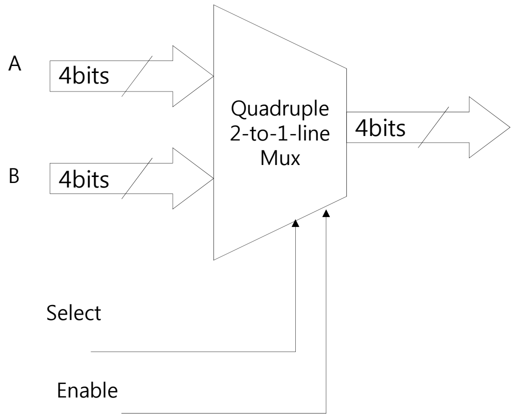


E	S	Output Y
1	X	all 0's
0	0	select A
0	1	select B

Function table

Multiplexers

- Quadruple 2-to-1 multiplexer



Boolean Function Implementation

- Analogy Mux and Decoder
  - A multiplexer is a decoder including an OR gate.
- 2^n-to-1 MUX
  - can implement any Boolean function of n input variable
  - however needs external gates
- $2^{(n+1)}$-to-1 MUX
  - a better solution: implement any Boolean function of n+1 input variable
  - n of these input: act as selection lines
  - remaining 1 input: the variable of the function

Boolean Function Implementation

- n-Variable function
  - Lookup the truth table
    - Input variables
      - n-1 pivot input variables
      - Remaining 1 designated input variable
    - Output variable (the boolean function)
      - Designated variable
      - Complement of the designated variable
      - Constant value 0
      - Constant value 1

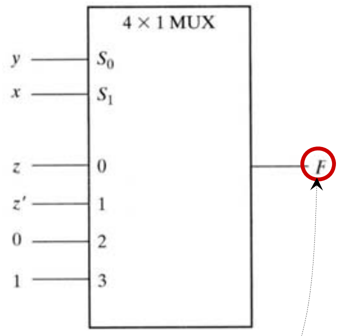
v_0	v_1	…	v_{n-2}	v_{n-1}	F
0	0	…	0	0	
0	0	…	0	1	
0	0	…	1	0	
0	0	…	1	1	
0	0	…			
		…			
1	1	…	1	0	
1	1	…	1	1	

  - Implement the function using 2^{n-1}-to-1-line multiplexer
    - Input variable ⇒ selection input
    - Remaining 1 input variable ⇒ the 0-th input data position

Boolean Function Implementation

- F(x,y,z)=Σ(1,2,6,7)
  - Using a 4x1 multiplexer

x	y	z	F	
0	0	0	0	$F = z$
0	0	1	1	
0	1	0	1	$F = z'$
0	1	1	0	
1	0	0	0	$F = 0$
1	0	1	0	
1	1	0	1	$F = 1$
1	1	1	1	

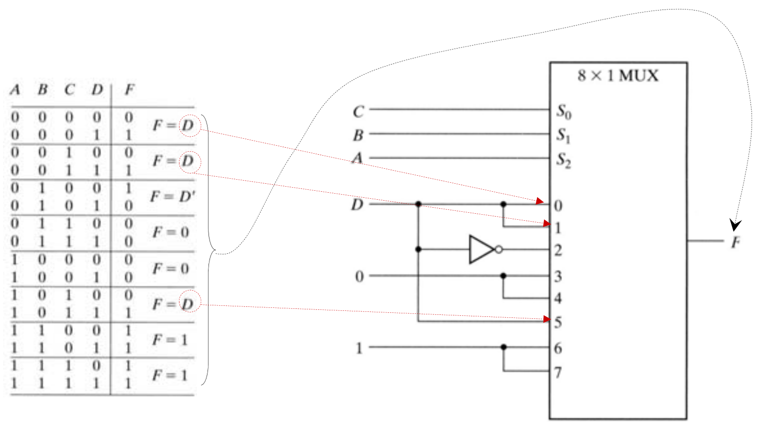


Boolean Function Implementation

- Procedure
  - assign an ordering sequence of the input variable
  - the rightmost variable (D) will be used for the input lines
  - assign the remaining n-1 variables to the selection lines w.r.t. their corresponding sequence
  - construct the truth table
  - consider a pair of consecutive minterms starting from m_0
  - determine the input lines

Boolean Function Implementation

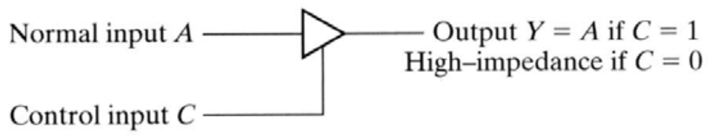
- F(A,B,C,D) = ∑(1,3,4,11,12,13,14,15) using 8x1 MUX



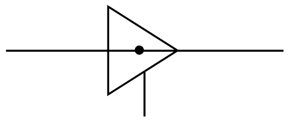
A	B	C	D	F	
0	0	0	0	0	$F = D$
0	0	0	1	1	
0	0	1	0	0	$F = D$
0	0	1	1	1	
0	1	0	0	1	$F = D'$
0	1	0	1	0	
0	1	1	0	0	$F = 0$
0	1	1	1	0	
1	0	0	0	0	$F = 0$
1	0	0	1	0	
1	0	1	0	0	$F = D$
1	0	1	1	1	
1	1	0	0	1	$F = 1$
1	1	0	1	1	
1	1	1	0	1	$F = 1$
1	1	1	1	1	

Three-State Gates

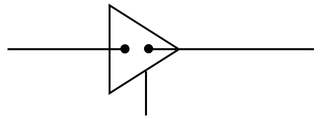
- Definition



- Buffer

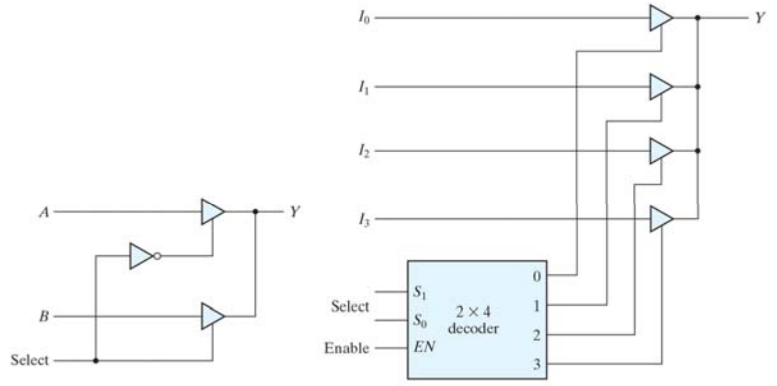


- High impledence: Open circuit (R = ∞)



Three-State Gates

- Multiplexers with 3-state gates



(a) 2-to-1-line mux

(b) 4-to-1-line mux

Discussion~~~