CS 303
Logic & Digital System Design

Ömer Ceylan

Sabancı Üniversitesi

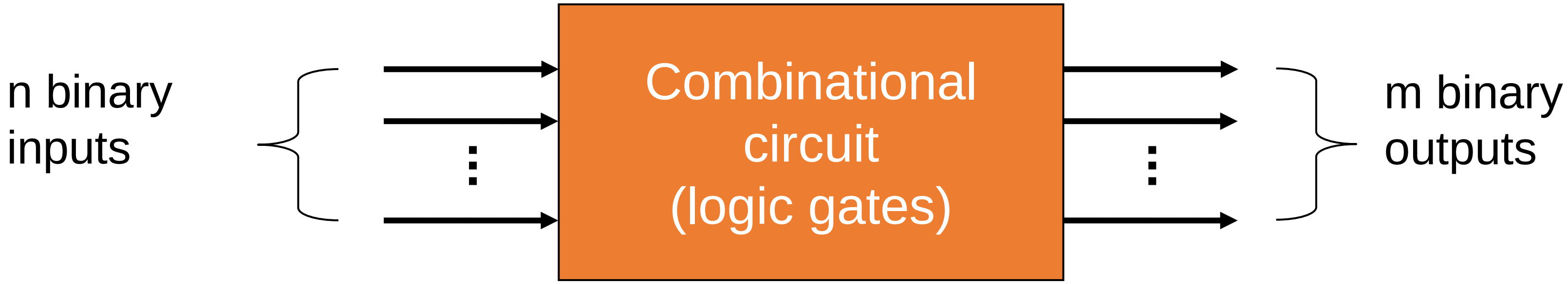
Chapter 4
Combinational Logic

Sabancı
Üniversitesi

Classification

1. Combinational
   - no memory
   - outputs depends on only the present inputs
   - expressed by Boolean functions
2. Sequential
   - storage elements + logic gates
   - the content of the storage elements define the state of the circuit
   - outputs are functions of both input and current state
   - state is a function of previous inputs
   - outputs not only depend the present inputs but also the past inputs

Combinational Circuits



- n input bits ➔ 2^n possible binary input combinations
- For each possible input combination, there is one possible output value
  - truth table
  - Boolean functions (with n input variables)
- Examples: adders, subtractors, comparators, decoders, encoders, and multiplexers.

Analysis & Design of Combinational Logic

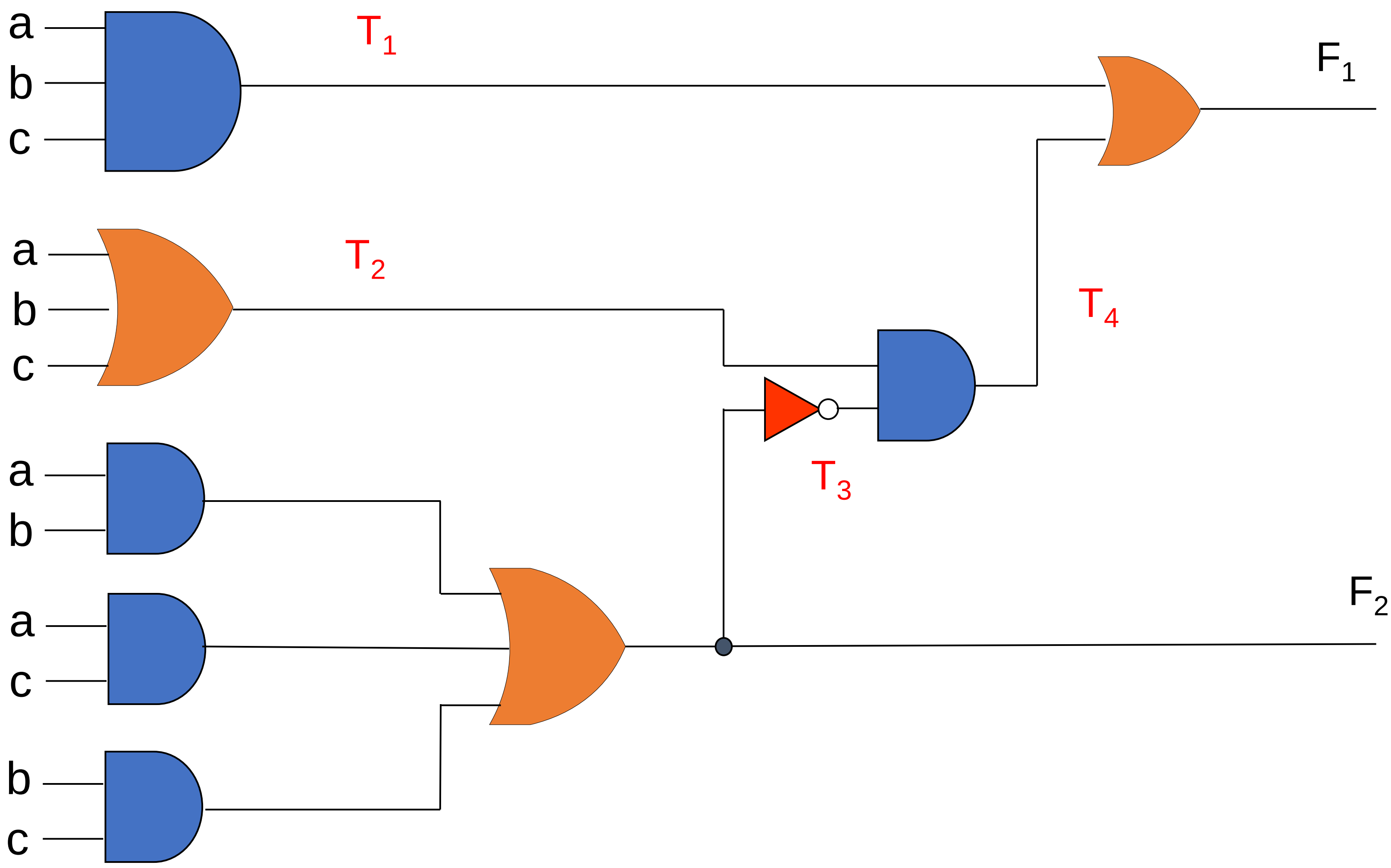
- Analysis: to find out the function that a given circuit implements
  - We are given a logic circuit and
  - we are expected to find out
    1. Boolean function(s)
    2. truth table
    3. A possible explanation of the circuit operation (i.e. what it does)
- Firstly, make sure that the given circuit is, indeed, combinational.

Analysis of Combinational Logic

  - Verifying the circuit is combinational
    - No memory elements
    - No feedback paths (connections)
- Secondly, obtain a Boolean function for each output or the truth table
- Lastly, interpret the operation of the circuit from the derived Boolean functions or truth table
  - What is it the circuit doing?
  - Addition, subtraction, multiplication, etc.

Obtaining Boolean Function

Example



Example: Obtaining Boolean Function

- Boolean expressions for named wires
  - $T_1 = abc$
  - $T_2 = a + b + c$
  - $F_2 = ab + ac + bc$
  - $T_3 = F_2' = (ab + ac + bc)'$
  - $T_4 = T_3T_2 = (ab + ac + bc)' (a + b + c)$
  - $F_1 = T_1 + T_4$

    $= abc + (ab + ac + bc)' (a + b + c)$

    $= abc + ((a' + b')(a' + c')(b' + c')) (a + b + c)$

    $= abc + ((a' + a'c' + a'b' + b'c')(b' + c')) (a + b + c)$

    $= abc + (a'b' + a'c' + a'b'c' + b'c') (a + b + c)$

Example: Obtaining Boolean Function

- Boolean expressions for outputs
  - F_2 = ab + ac + bc
  - F_1 = abc + (a’b’ + a’c’ + b’c’) (a + b + c)
  - F_1 = abc + a’b’c + a’bc’ + ab’c’
  - F_1 = a(bc + b’c’) + a’(b’c + bc’)
  - F_1 = a(b ⊕ c)’ + a’(b ⊕ c)
  - F_1 =

Example: Obtaining Truth Table

$F_1 = a \oplus b \oplus c$

$F_2 = ab + ac + bc$

carry sum

a	b	c	T_1	T_2	T_3	T_4	F_2	F_1
0	0	0	0	0	1	0	0	0
0	0	1	0	1	1	1	0	1
0	1	0	0	1	1	1	0	1
0	1	1	0	1	0	0	1	0
1	0	0	0	1	1	1	0	1
1	0	1	0	1	0	0	1	0
1	1	0	0	1	0	0	1	0
1	1	1	1	1	0	0	1	1

This is what we call full-adder (FA)

Design of Combinational Logic

- Design Procedure:
  - We start with the verbal specification about what the resulting circuit will do for us (i.e. which function it will implement)
    - Specifications are often verbal, and very likely incomplete and ambiguous (if not faulty)
    - Wrong interpretations can result in incorrect circuit
  - We are expected to find
    1. firstly, Boolean function(s) (or truth table) to realize the desired functionality
    2. Logic circuit implementing the Boolean function(s) (or the truth table)

Possible Design Steps

1. Find out the number of inputs and outputs
2. Derive the truth table that defines the required relationship between inputs and outputs
3. Obtain a <u>simplified</u> Boolean function for each output
4. Draw the logic diagram (enter your design into CAD)
5. Verify the correctness of the design

Design Constraints

- From the truth table, we can obtain a variety of simplified expressions
- Question: which one to choose?
- The design constraints may help in the selection process
- Constraints:
  - number of gates
  - propagation time of the signal all the way from the inputs to the outputs
  - number of inputs to a gate
  - number of interconnections
  - power consumption
  - driving capability of each gate

Example: Design Process

- BCD-to-2421 Converter
- Verbal specification:
  - Given a BCD digit (i.e. {0, 1, …, 9}), the circuit computes 2421 code equivalent of the decimal number
- Step 1: how many inputs and how many outputs?
  - four inputs and four outputs
- Step 2:
  - Obtain the truth table
  - `0000` → `0000`
  - `1001` → `1111`
  - etc.

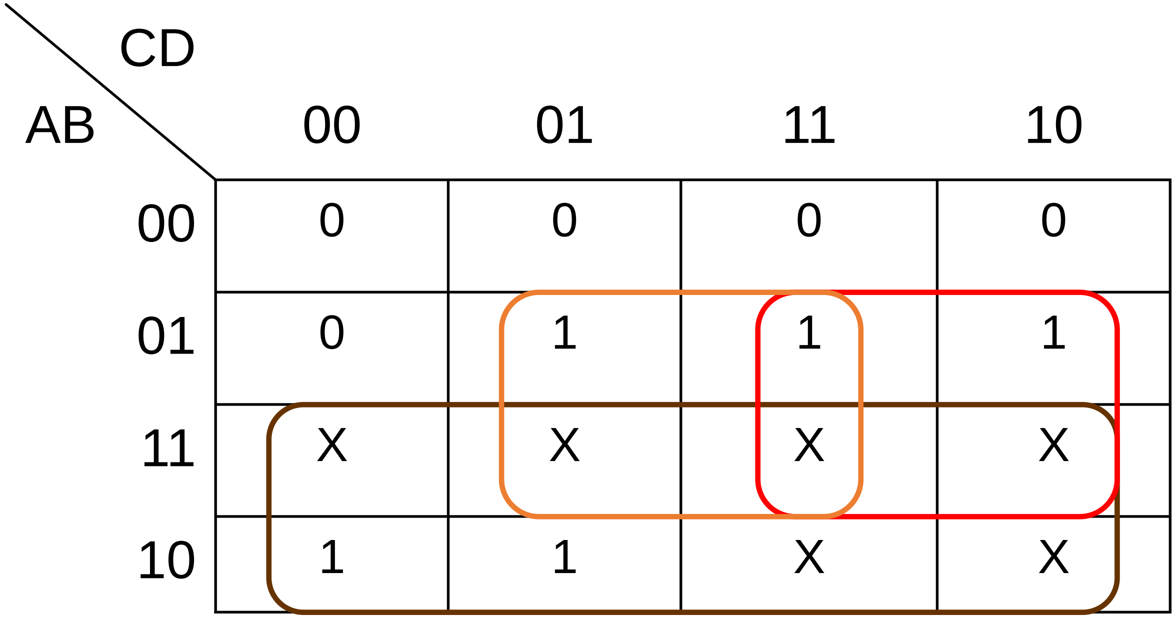
BCD-to-2421 Converter

- Truth Table

Inputs				Outputs			
A	B	C	D	x	y	z	t
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	0
0	0	1	1	0	0	1	1
0	1	0	0	0	1	0	0
0	1	0	1	1	0	1	1
0	1	1	0	1	1	0	0
0	1	1	1	1	1	0	1
1	0	0	0	1	1	1	0
1	0	0	1	1	1	1	1

BCD-to-2421 Converter

- Step 3: Obtain simplified Boolean expression for each output
- Output x:



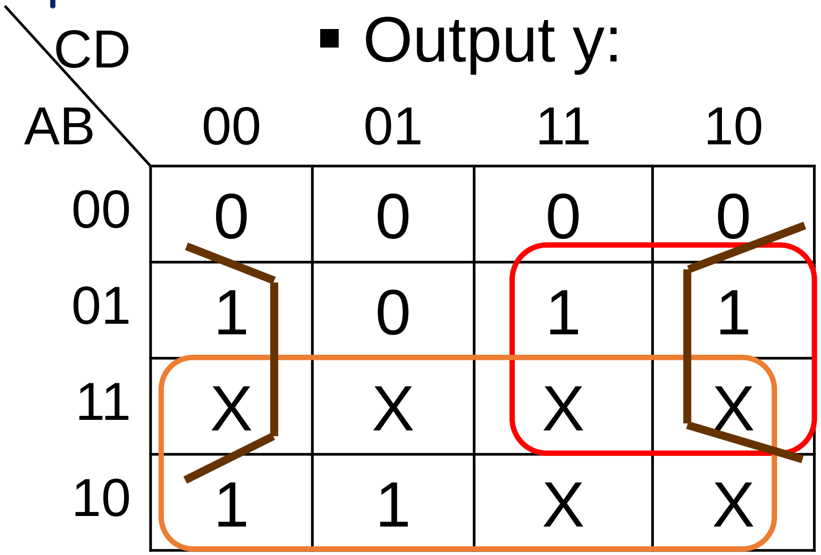
AB \ CD	00	01	11	10
00	0	0	0	0
01	0	1	1	1
11	X	X	X	X
10	1	1	X	X

A	B	C	D	x
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	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
The rest				X

x = BD + BC + A

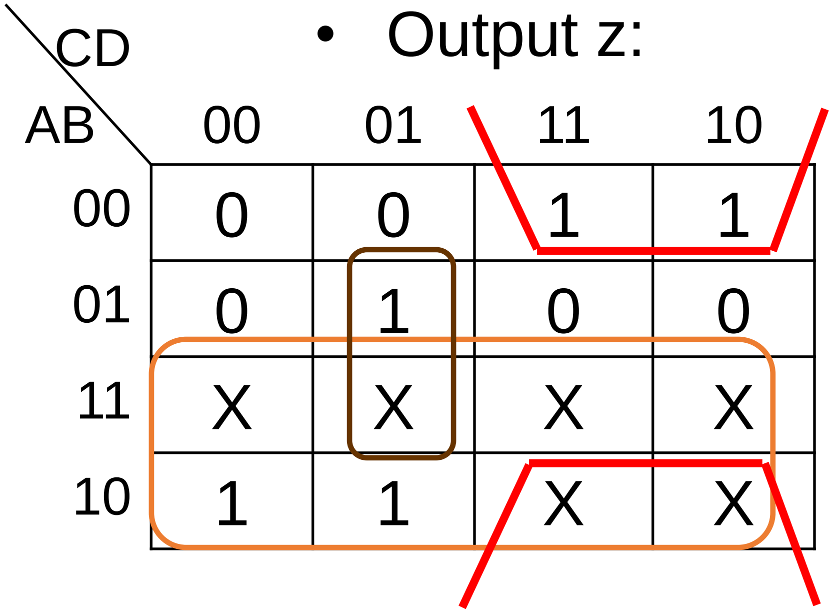
Boolean Expressions for Outputs

- Output y:



AB \ CD	00	01	11	10
00	0	0	0	0
01	1	0	1	1
11	X	X	X	X
10	1	1	X	X

- Output z:

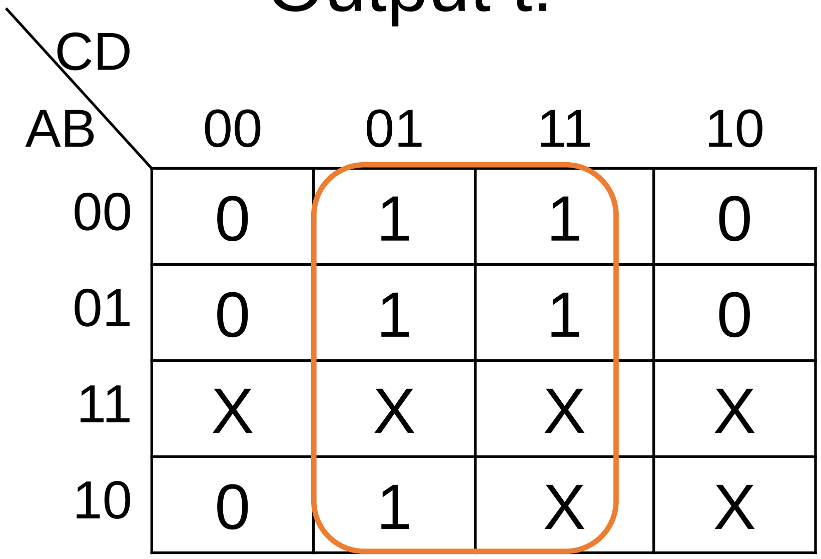


AB \ CD	00	01	11	10
00	0	0	1	1
01	0	1	0	0
11	X	X	X	X
10	1	1	X	X

A	B	C	D	y	z
0	0	0	0	0	0
0	0	0	1	0	0
0	0	1	0	0	1
0	0	1	1	0	1
0	1	0	0	1	0
0	1	0	1	0	1
0	1	1	0	1	0
0	1	1	1	1	0
1	0	0	0	1	1
1	0	0	1	1	1
The rest				X	X

Boolean Expressions for Outputs

- Output t:



AB \ CD	00	01	11	10
00	0	1	1	0
01	0	1	1	0
11	X	X	X	X
10	0	1	X	X

t = D

A	B	C	D	t
0	0	0	0	0
0	0	0	1	1
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	1
The rest				X

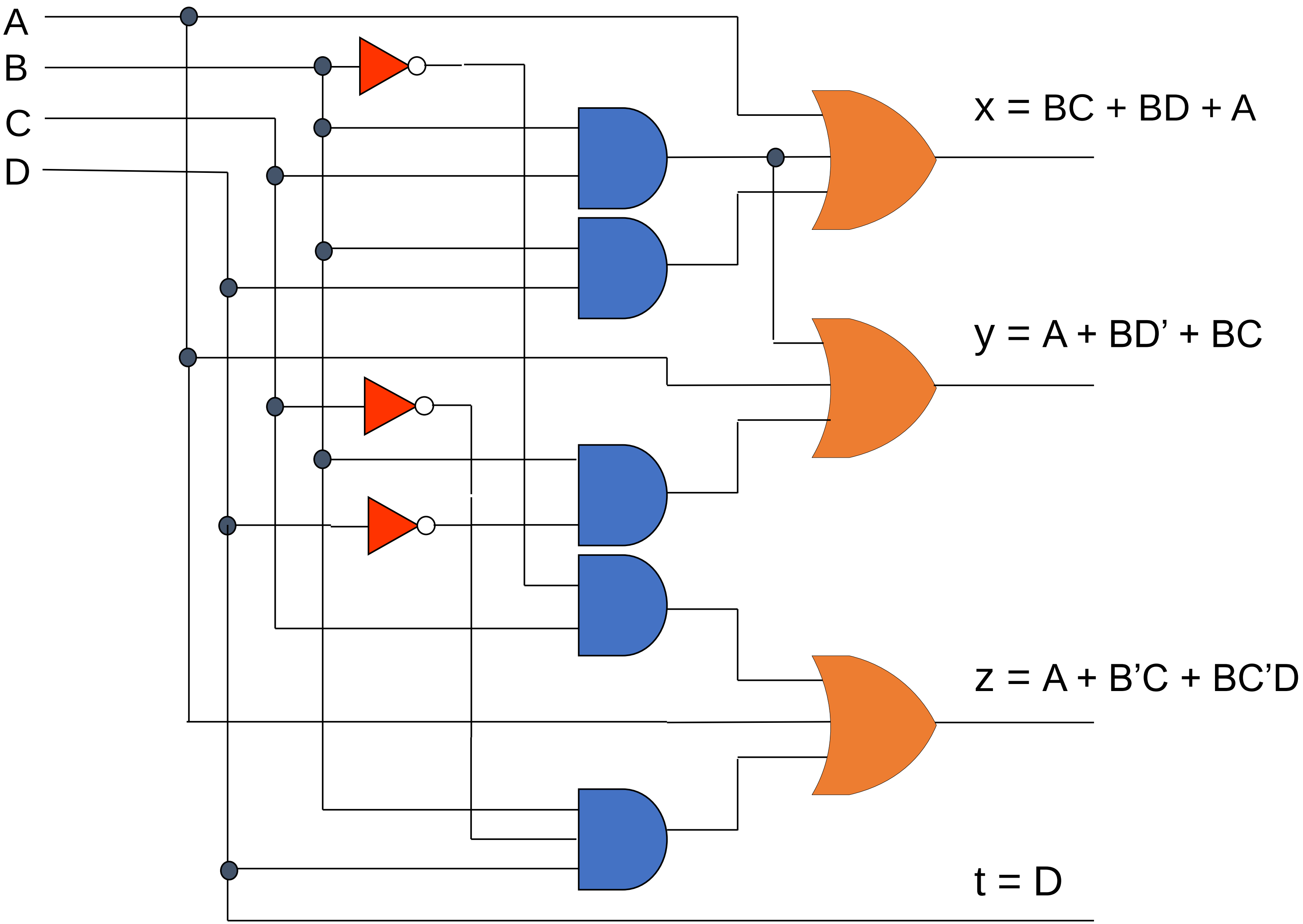
- Step 4: Draw the logic diagram

x = BC + BD + A

y = A + BD’ + BC

z = A + B’C + BC’D

Example: Logic Diagram



x = BC + BD + A

y = A + BD’ + BC

z = A + B’C + BC’D

t = D

Example: Test & Verification

- Step 5: Check the functional correctness of the logic circuit
- Apply all possible input combinations
- And check if the circuit generates the correct output for each input combination
- For large circuits with many input combinations, this may not be feasible.
- Statistical techniques may be used to verify the correctness of large circuits with many input combinations

Binary Adder/Subtractor

- (Arithmetic) Addition of two binary digits
  - 0 + 0 = 0, 0 + 1 = 1, 1 + 0 = 1, and 1 + 1 = 10
  - The result has two components
    - the sum (S)
    - the carry (C)
- (Arithmetic) Addition of three binary digits

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

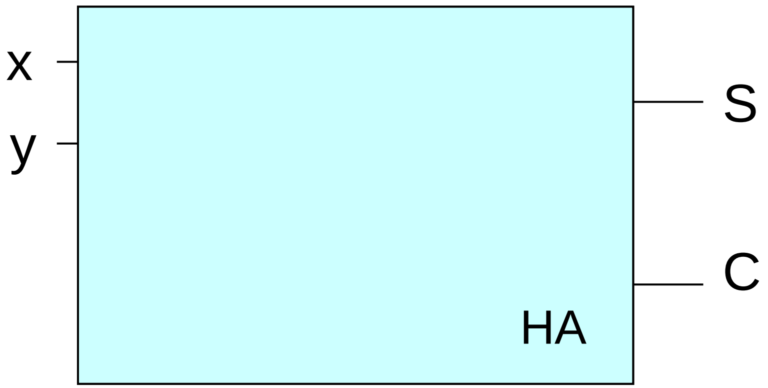
Half Adder

- Truth table

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

$S = x'y + xy' = x \oplus y$

$C = xy$



Full Adder 1/2

- A circuit that performs the arithmetic sum of three bits
  - Three inputs
  - the range of output is [0, 3]
  - Two binary 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

Full Adder 2/2

- Karnaugh Maps

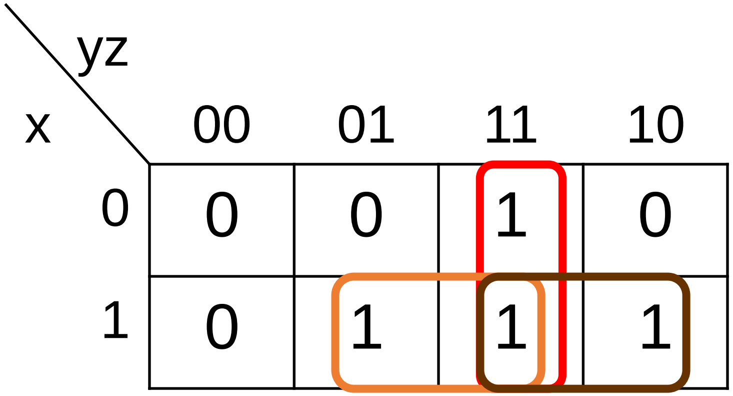
x \ yz	00	01	11	10
0	0	1	0	1
1	1	0	1	0

S = xy’z’ + x’y’z + xyz + x’yz’

= ...

= ...

= x ⊕ y ⊕ z



x \ yz	00	01	11	10
0	0	0	1	0
1	0	1	1	1

C = xy + xz + yz

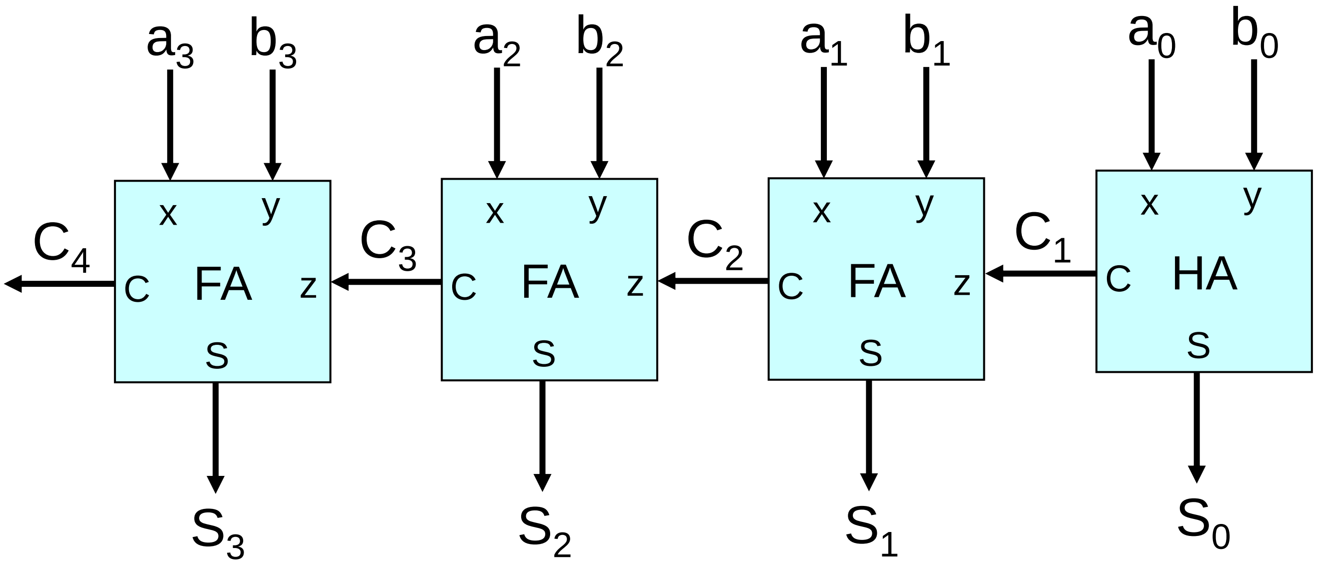
Two level implementation

1st level: three AND gates

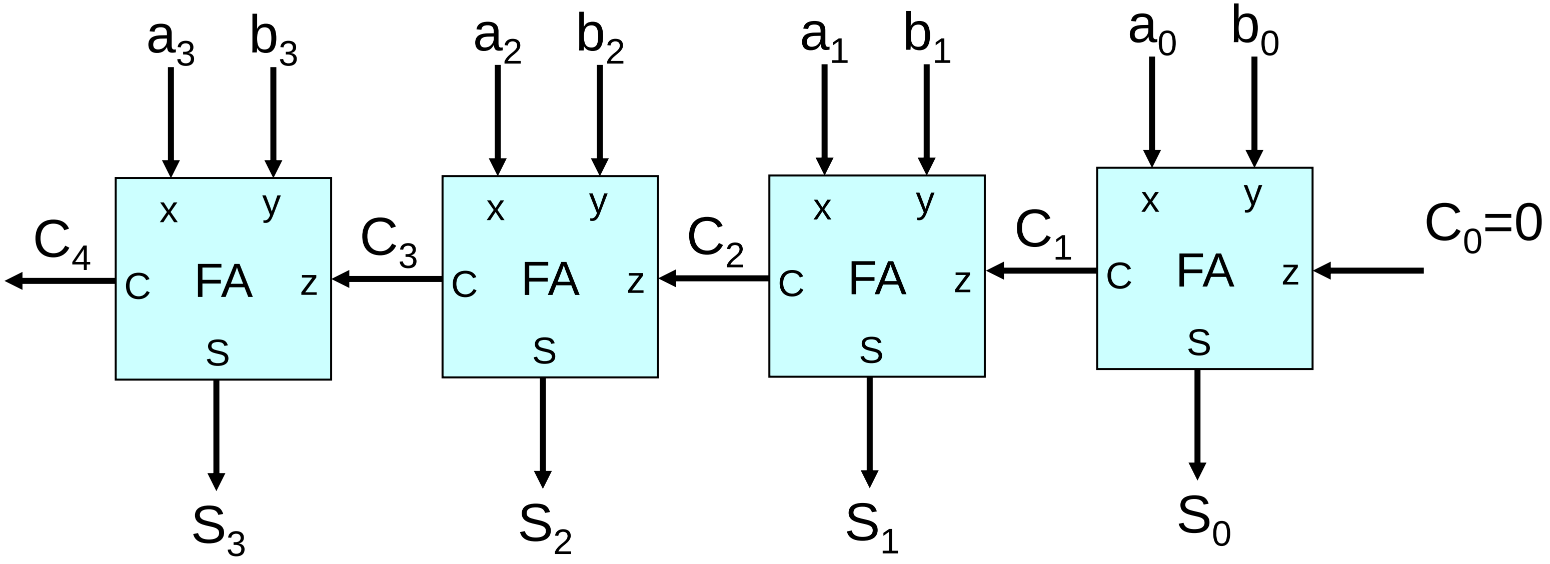
2nd level: One OR gate

Integer Addition 1/2

- Binary adder:
  - A digital circuit that produces the arithmetic sum of two binary numbers
  - $A = (a_{n-1}, a_{n-2}, \ldots, a_1, a_0)$
  - $B = (b_{n-1}, b_{n-2}, \ldots, b_1, b_0)$
- A simple case: 4-bit binary adder



Integer Addition 2/2



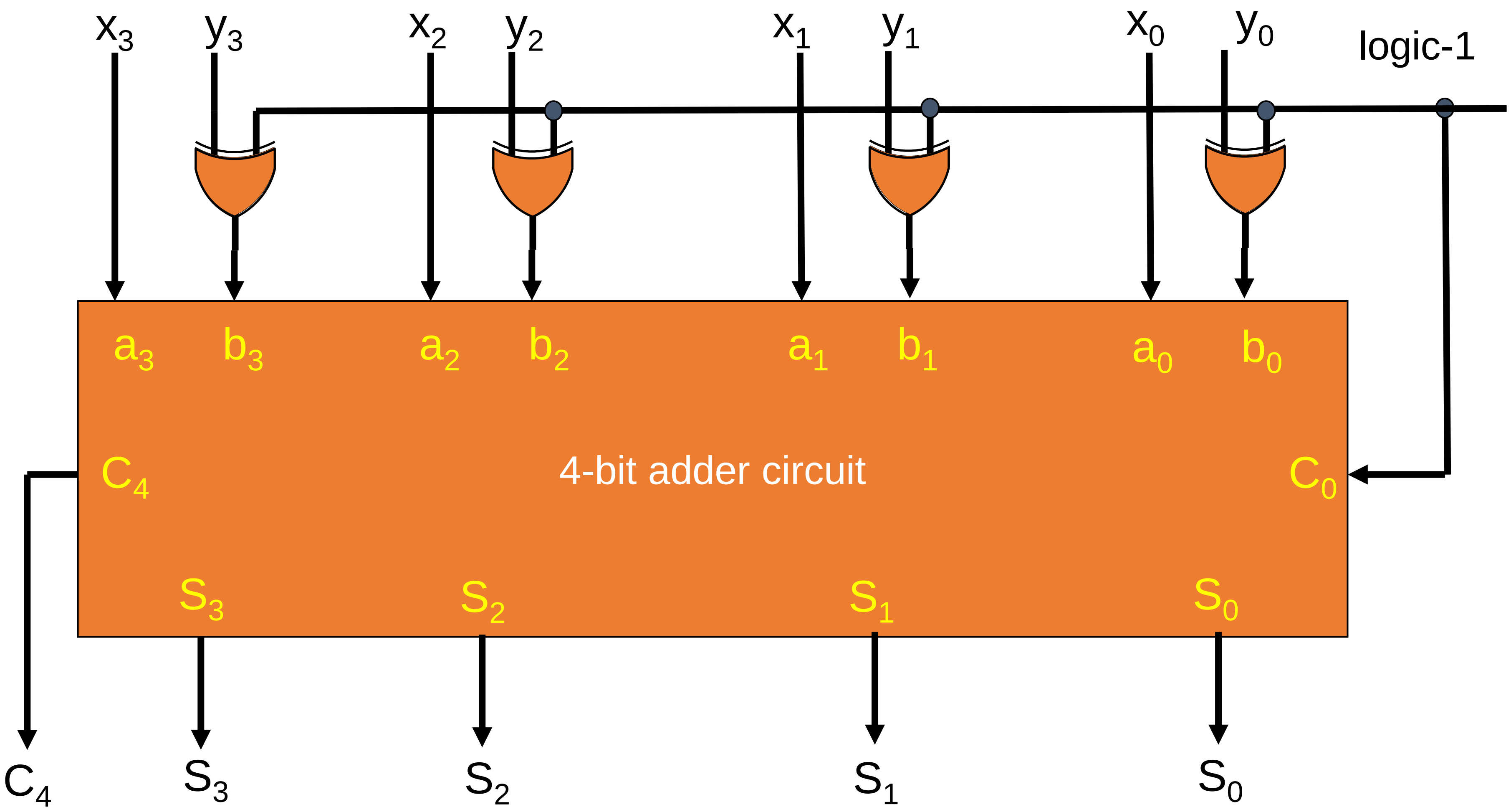
Ripple-carry adder

Hierarchical Design Methodology

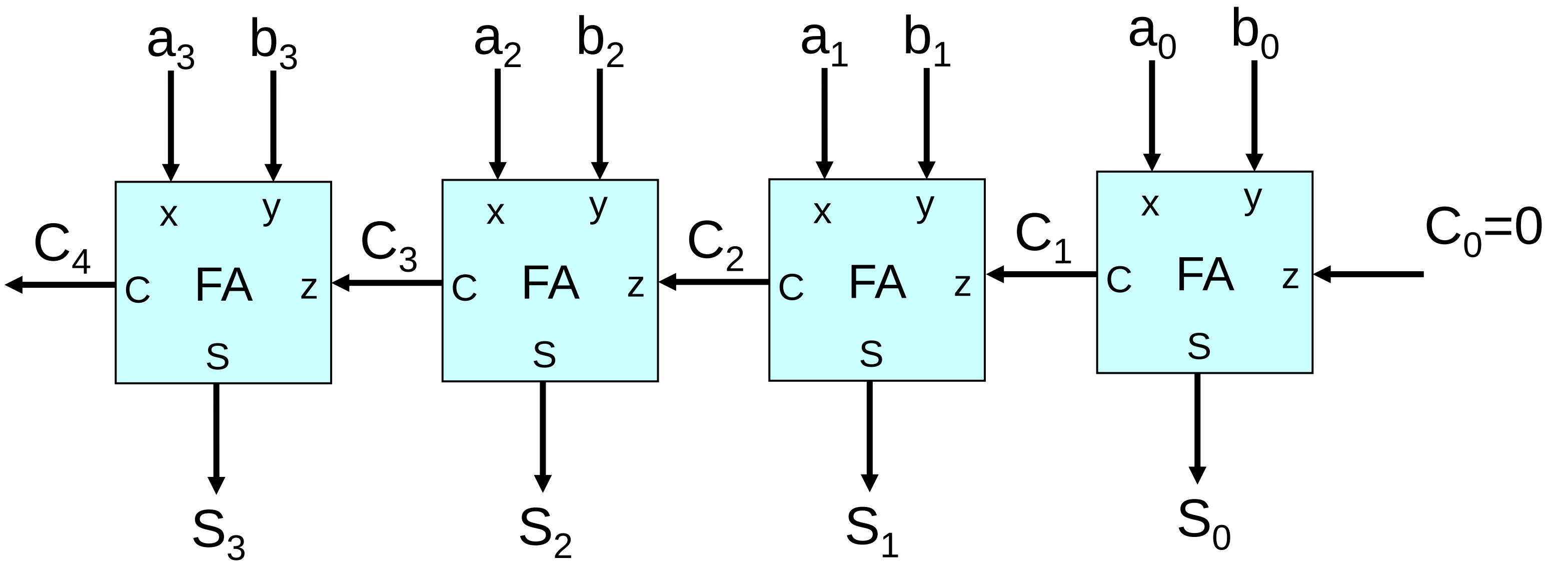
- The design methodology we used to build carry-ripple adder is what is referred as hierarchical design.
- In classical design, we have:
  - 9 inputs including C_0.
  - 5 outputs
  - Truth tables with 2^9 = 512 entries
  - We have to optimize five Boolean functions with 9 variables each.
- Hierarchical design
  - we divide our design into smaller functional blocks
  - connect functional units to produce the big functionality

Subtractor

- Recall how we do subtraction (2’s complement)
  - $X - Y = X + (2^n - Y) = X + \sim Y + 1$



Faster Adders

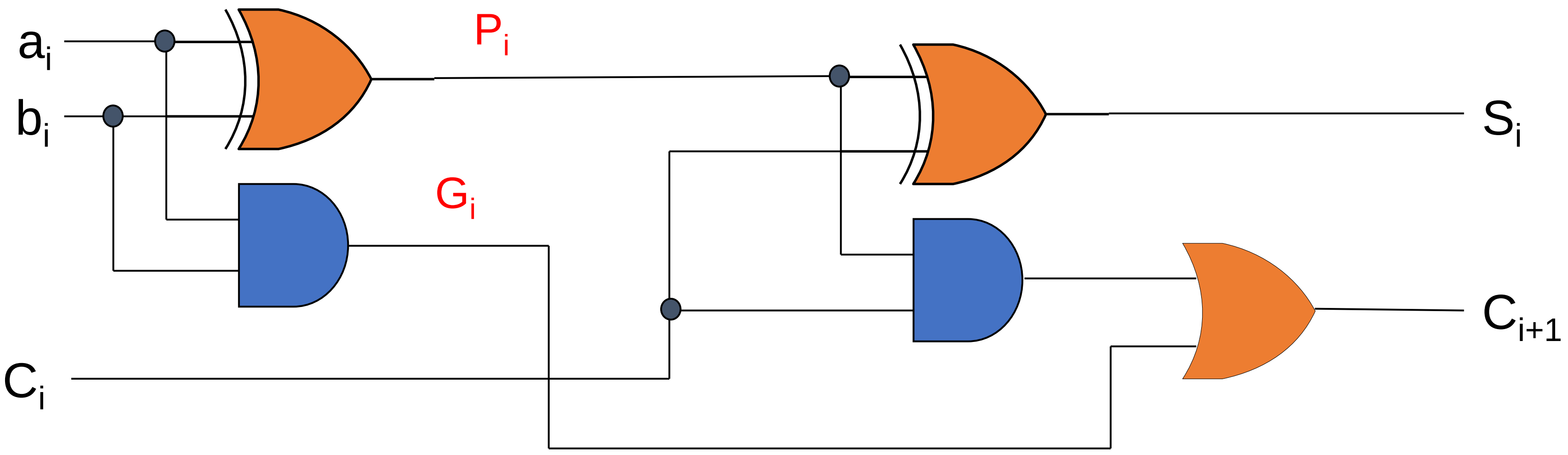


Ripple-carry adder

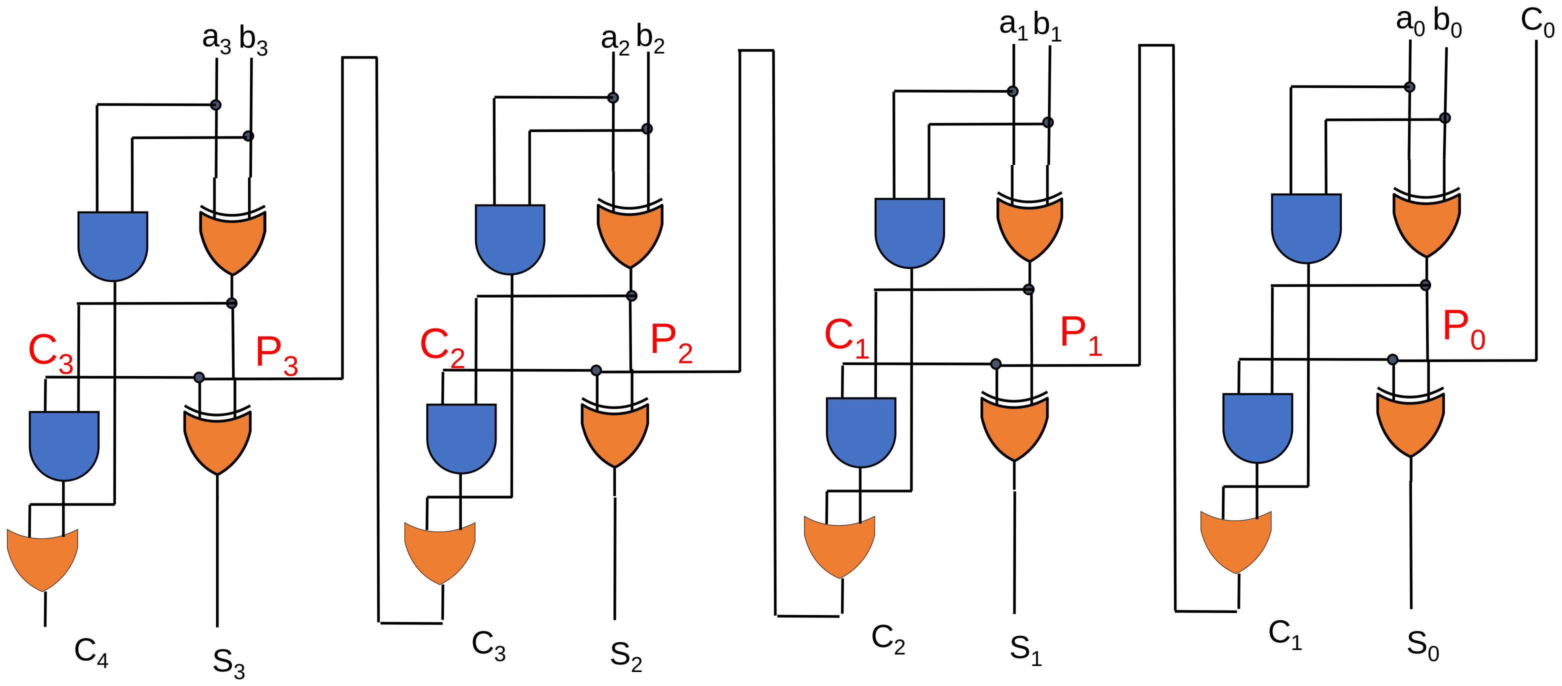
- What is the total propagation time of 4-bit ripple-carry adder?
  - τ_{FA}: propagation time of a single full adder.
  - We have four full adders connected in cascaded fashion
  - Total propagation time: $4\tau_{FA}$ (roughly)

Carry Propagation

- Propation time of a full adder
  - $\tau_{XOR} \approx 2\tau_{AND} = 2\tau_{OR}$
  - $\tau_{FA} \approx 2\tau_{XOR}$



Carry Propagation



- Delays
  - **P_0, P_1, P_2, P_3: $\tau_{XOR} \approx 2\tau_{AND}$**
  - **$C_1(S_0)$: $\approx 2\tau_{AND} + 2\tau_{AND} \approx 4\tau_{AND}$**
  - **$C_2(S_1)$: $\approx 4\tau_{AND} + 2\tau_{AND} \approx 6\tau_{AND}$**
  - **$C_3(S_2)$: $\approx 6\tau_{AND} + 2\tau_{AND} \approx 8\tau_{AND}$**
  - **$C_4(S_3)$: $\approx 8\tau_{AND} + 2\tau_{AND} \approx 10\tau_{AND}$**

Faster Adders

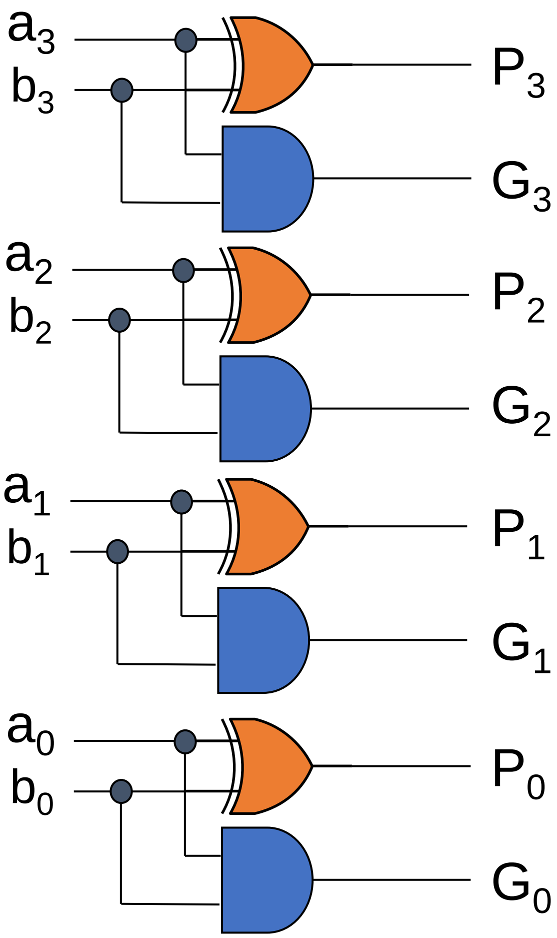
- The carry propagation technique is a limiting factor in the speed, with which two numbers are added.
- Two alternatives
  1. use faster gates with reduced delays
  2. Increase the circuit complexity (i.e. put more gates) in such a way that the carry delay time is reduced.
- An example for the latter type of solution is carry lookahead adders
  - Two binary variables:
    1. $P_i = a_i \oplus b_i$ – carry propagate
    2. $G_i = a_i b_i$ – carry generate

Carry Lookahead Adders

- Sum and carry can be expressed in terms of P_i and G_i:
  - $P_i = a_i \oplus b_i$; $G_i = a_ib_i$
  - $S_i = P_i \oplus C_i$
  - $C_{i+1} = G_i + P_iC_i$
- Why the names (carry propagate and generate)?
  - If $G_i = 1$ (both $a_i = b_i = 1$), then a “new” carry is generated
  - If $P_i = 1$ (either $a_i = 1$ or $b_i = 1$), then a carry coming from the previous lower bit position is propagated to the next higher bit position

Generating P_i and G_i Signals

- Example: 4-bit operands



4-bit Carry Lookahead Adder

- We can use the carry propagate and carry generate signals to compute carry bits used in addition operation
  - C_0 = input
  - $C_1 = G_0 + P_0C_0$
  - $C_2 = G_1 + P_1C_1$

    $= G_1 + P_1(G_0 + P_0C_0) = G_1 + P_1G_0 + P_1P_0C_0$
  - $C_3 = G_2 + P_2C_2 = G_2 + P_2(G_1 + P_1G_0 + P_1P_0C_0)$

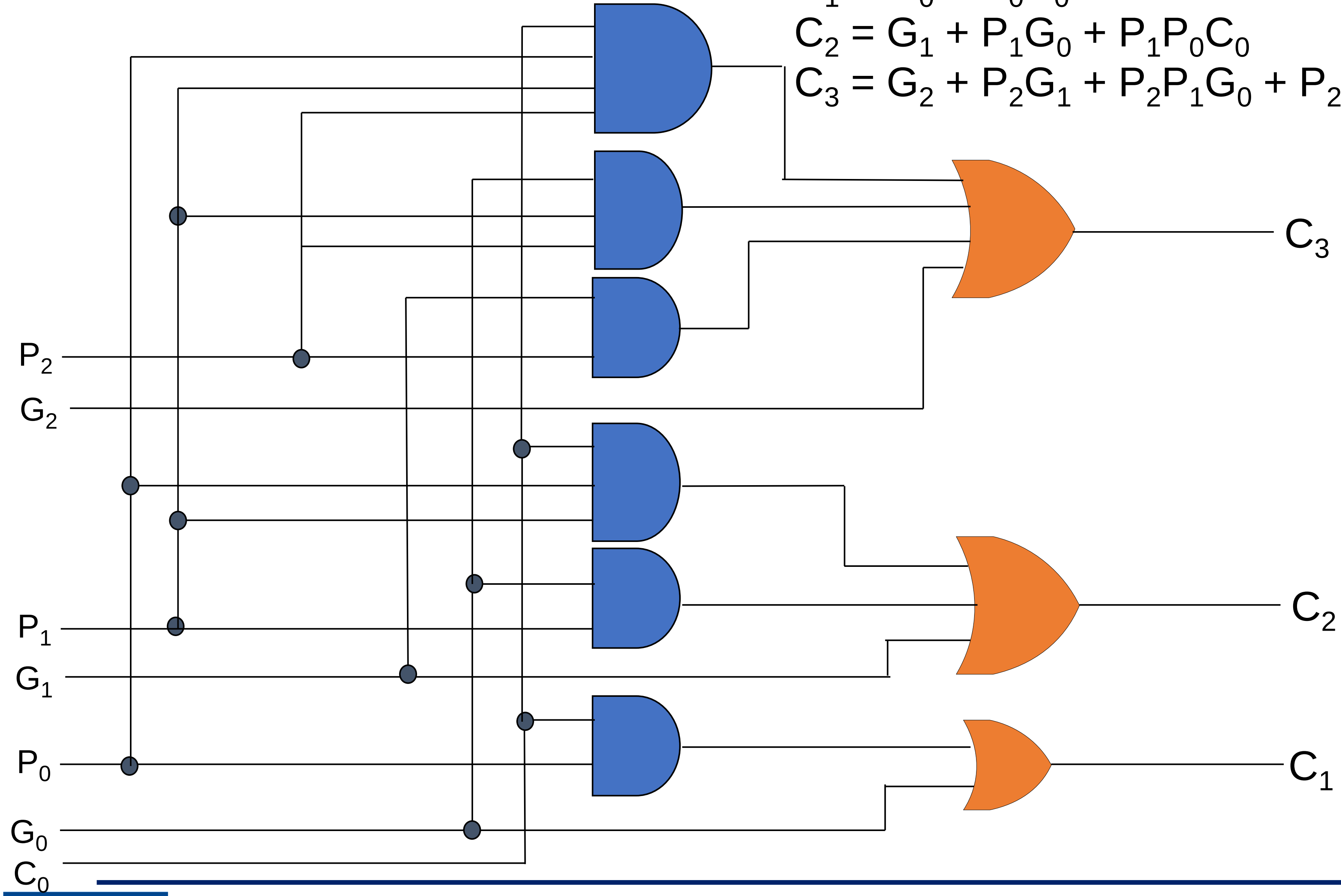
    $= G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0C_0$
  - $C_4 = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0C_0$

4-bit Carry Lookahead Circuit 1/3

$C_1 = G_0 + P_0C_0$

$C_2 = G_1 + P_1G_0 + P_1P_0C_0$

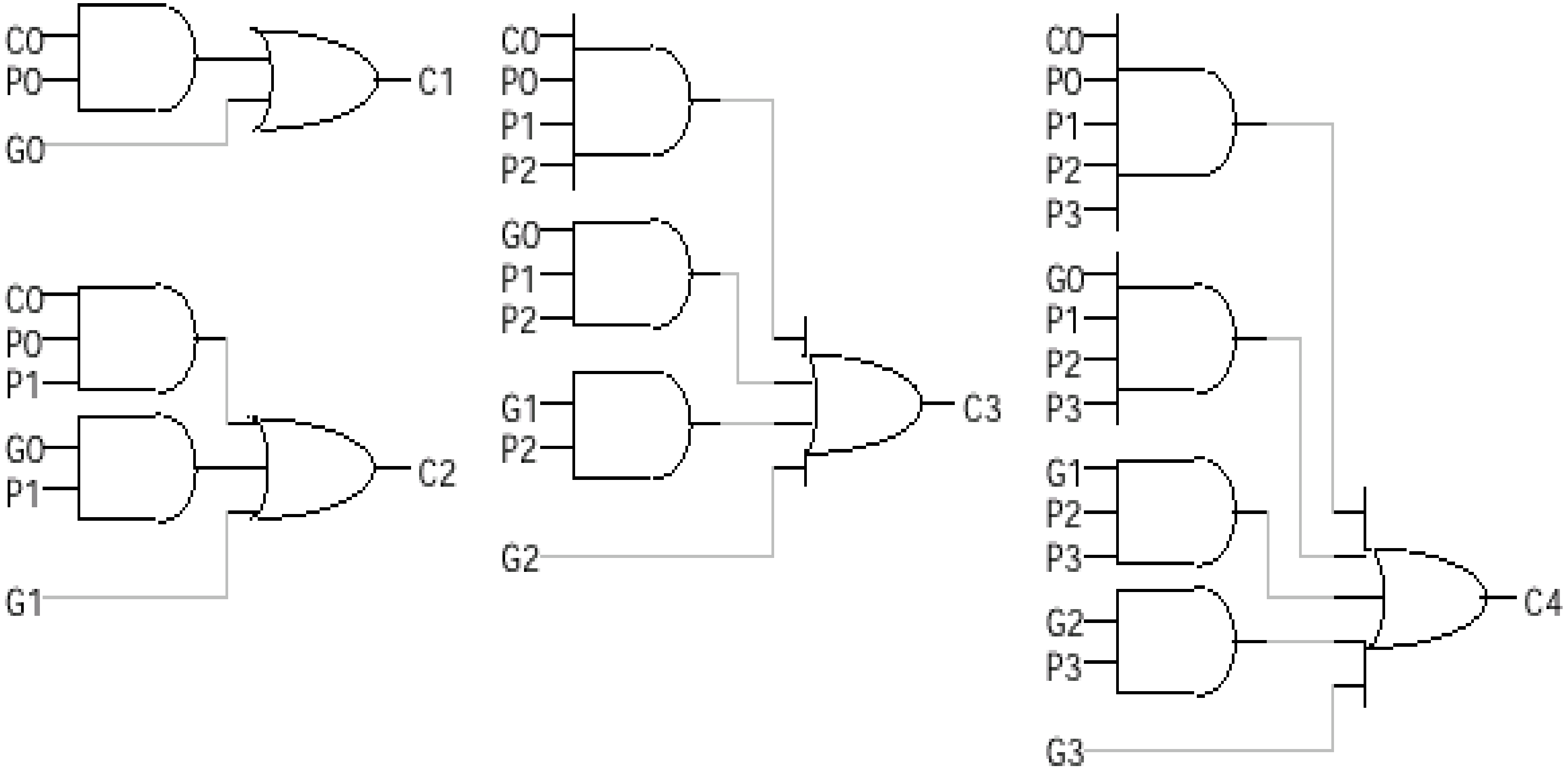
$C_3 = G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0C_0$



4-bit Carry Lookahead Circuit 2/3

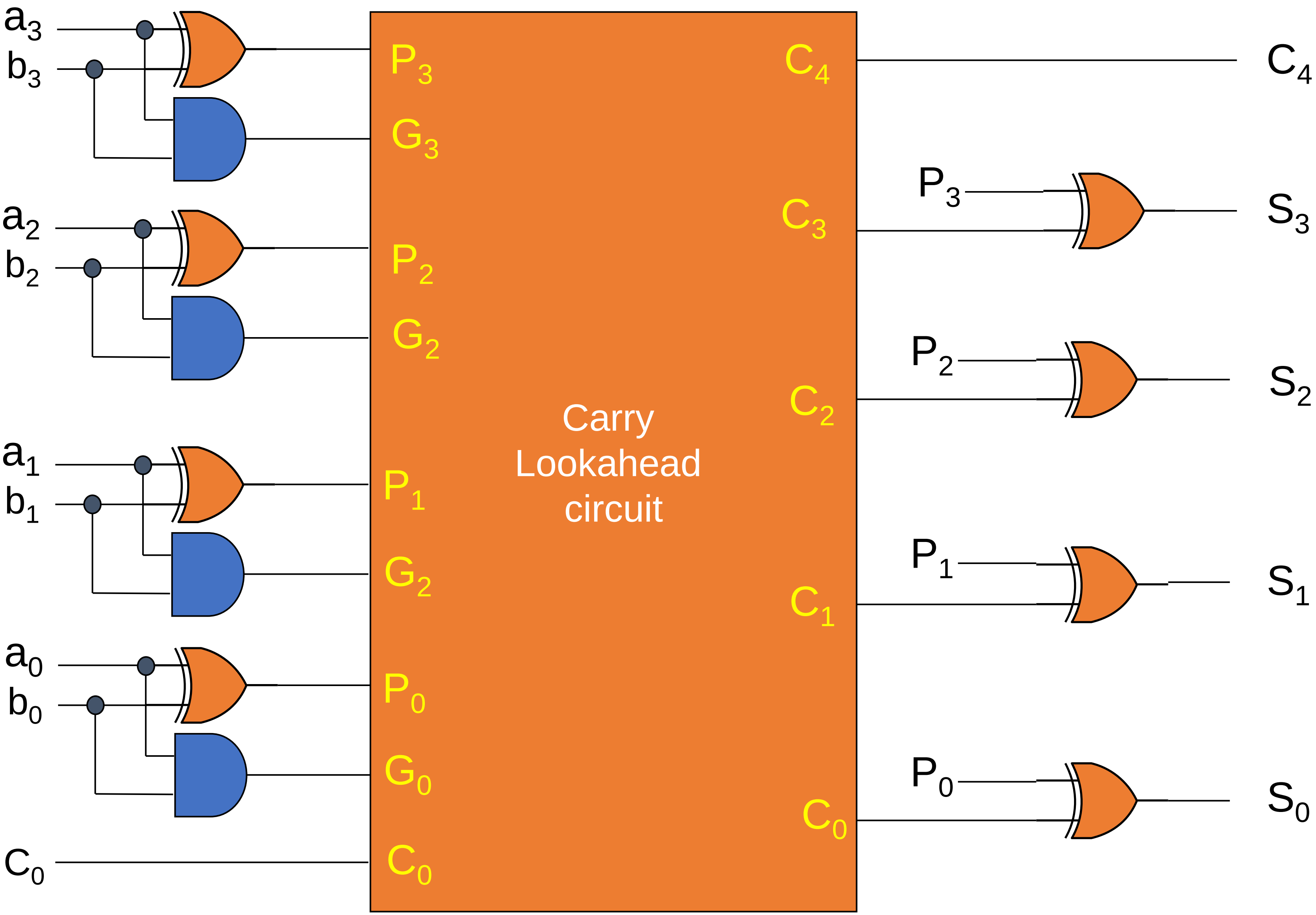
- All three carries (C_1, C_2, C_3) can be realized as two-level implementation (i.e. AND-OR)
- C_3 does not have to wait for C_2 and C_1 to propagate
- C_3 has its own circuit
- The propagations happen concurrently
- Certain parts are repeated.
  - This is the main reason why the faster adder is more complicated.

4-bit Carry Lookahead Circuit 3/3

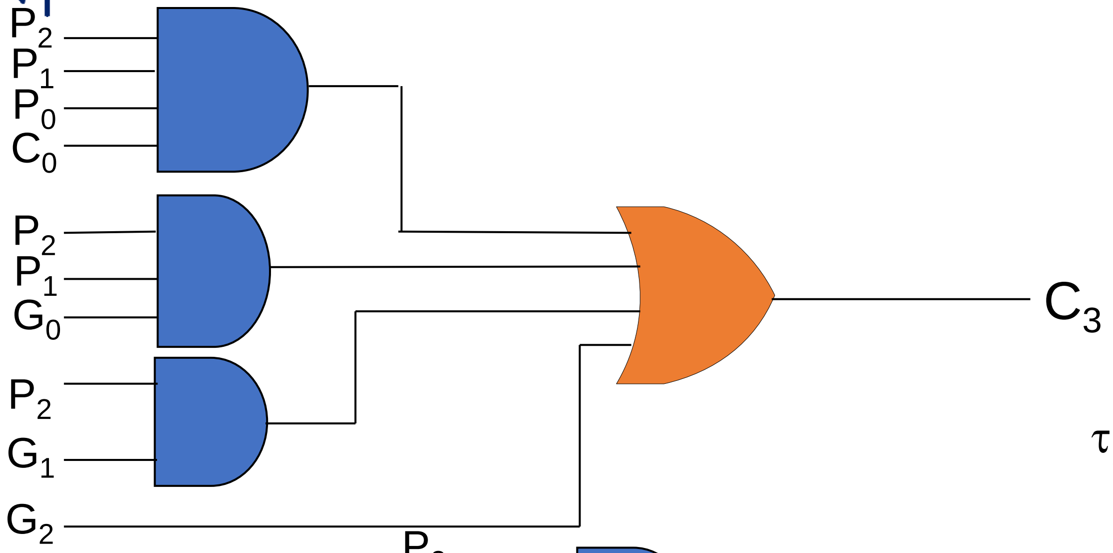


- Two levels of logic

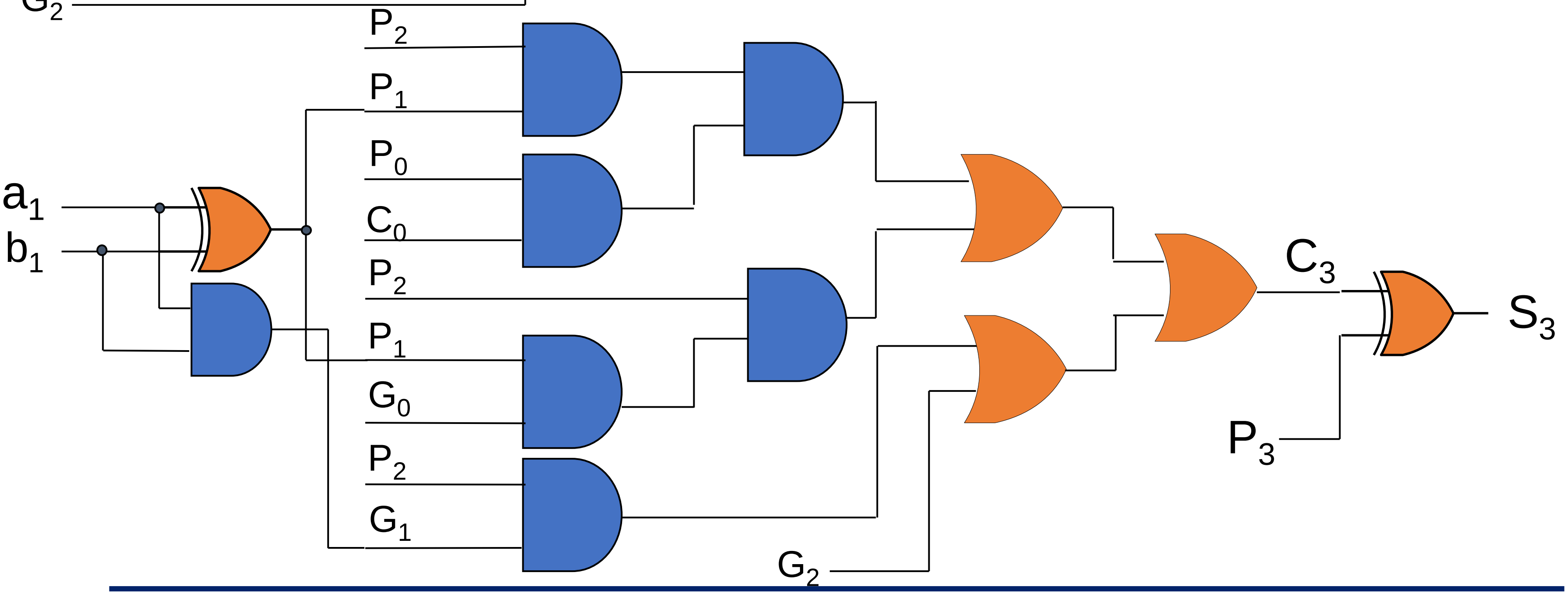
4-bit Carry Lookahead Adder



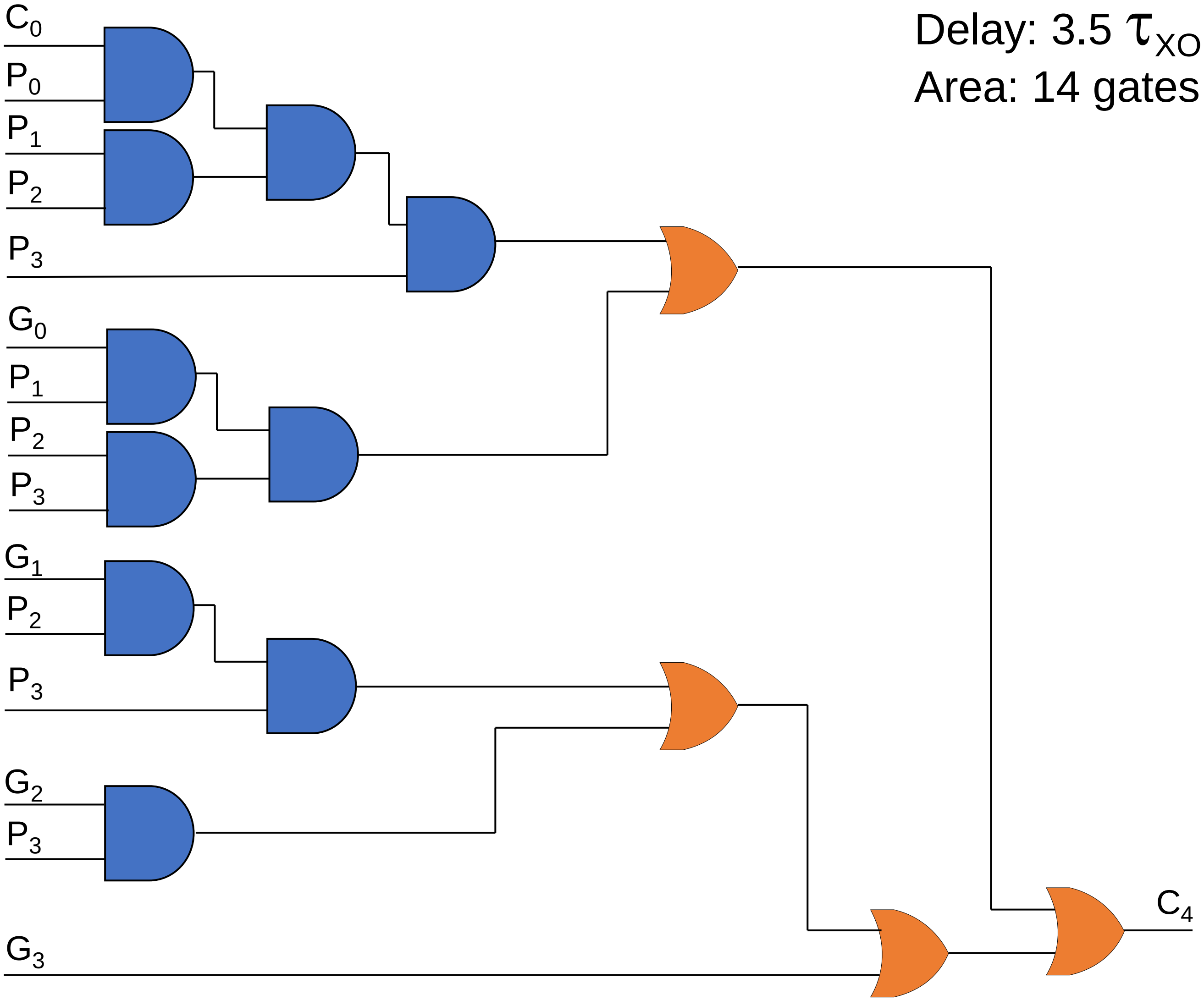
Propagation Time of Carry Lookahead Adders



$$\tau_{XOR} = 2\tau_{AND} = 2\tau_{OR}$$



Generating C_4



Delay: 3.5 τ_{XOR}

Area: 14 gates

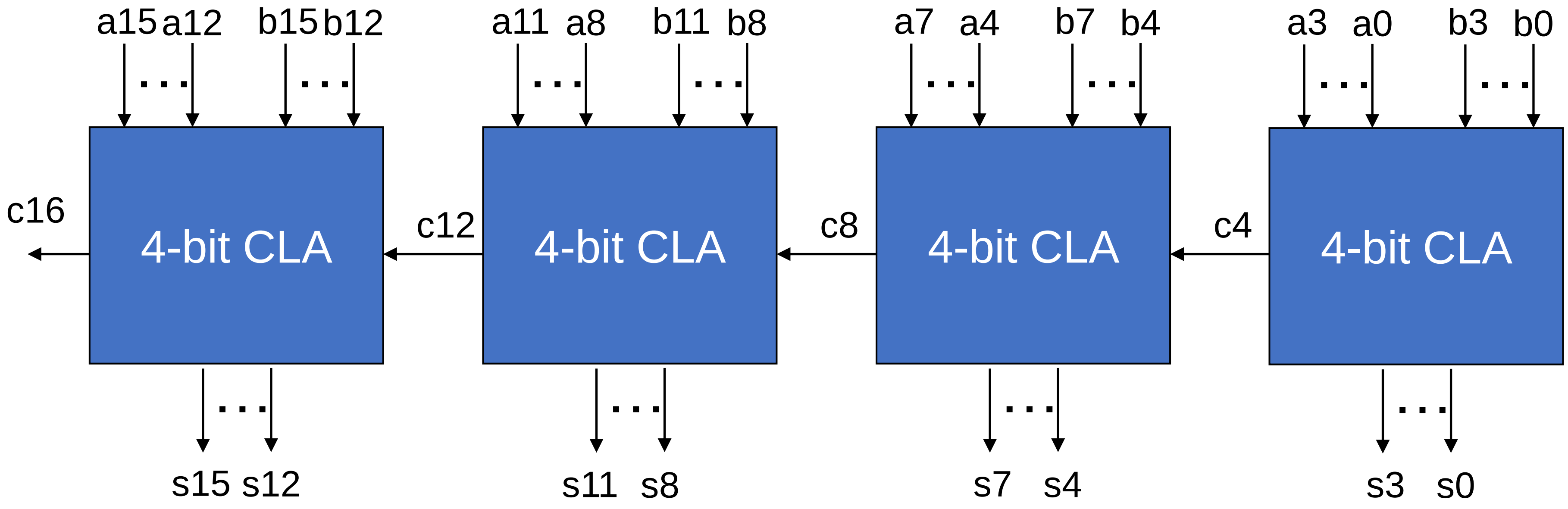
Summary

- Half adder → HA
- Full adder → FA
  - critical path delay
  - time complexity: $2\tau_{XOR}$ (assume $\tau_{XOR} = 2\tau_{AND} = 2\tau_{OR}$)
- 4-bit carry-ripple adder
  - 4 FAs in serial
    - carry propagation
  - time complexity: $10\tau_{AND}$
- Faster Adder
  - Separate carry generation circuits (more complex)
  - time complexity: $\approx 4\tau_{XOR} \approx 8\tau_{AND}$

Summary

- General formulae
- n-bit carry-ripple adder
  - time complexity: $\approx (2 + 2n)\tau_{AND}$
- n-bit CLA
  - Separate carry generation circuits (more complex)
  - time complexity: $\approx (4 + 2\lceil \log_2 n \rceil)\tau_{AND}$
  - n = 8
    - $C_7 = G_6 + P_6G_5 + P_6P_5G_4 + P_6P_5P_4G_3 + P_6P_5P_4P_3G_2 + P_6P_5P_4P_3P_2G_1 + P_6P_5P_4P_3P_2P_1G_0 + P_7P_6P_5P_4P_3P_2P_1C_0$

Hybrid Approach for 16-bit Adder



Overflow

- How to detect overflows:
  - two n-bit numbers
  - we add them, and result may be an (n+1)-bit number → overflow.
  - Unsigned numbers:
    - easy
    - check the carryout.
  - Signed numbers
    - more complicated
    - overflow can occur in addition, when the operands are of the same sign

Examples: Overflows

- Example 1: 8-bit signed numbers

…00	0	1	0	0	0	1	0	0	68
…00	0	1	0	1	1	0	1	1	91
…00	1	0	0	1	1	1	1	1	159

- Example 2: 8-bit signed numbers

…11	1	0	1	1	1	1	0	0	-68
…11	1	0	1	0	0	1	0	1	-91
…11	0	1	1	0	0	0	0	1	-159

How to Detect Overflows:

- First Method

1. If both operands are positive and the MSB of the result is 1.
2. If both operands are negative and the MSB of the result is 0.

a_{n-1}	b_{n-1}	s_{n-1}	V
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

Detecting Overflows: First Method

a_{n-1} \ $b_{n-1}S_{n-1}$	00	01	11	10
0	0	1	0	0
1	0	0	0	1

- $V = a_{n-1}' \, b_{n-1}' \, S_{n-1} + a_{n-1} \, b_{n-1} \, S_{n-1}'$

- Can we do it better?

a_{n-1}	b_{n-1}	S_{n-1}	V
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

Detecting Overflows

- Second method:
  - Remember we have other variables when adding:
    - Carries

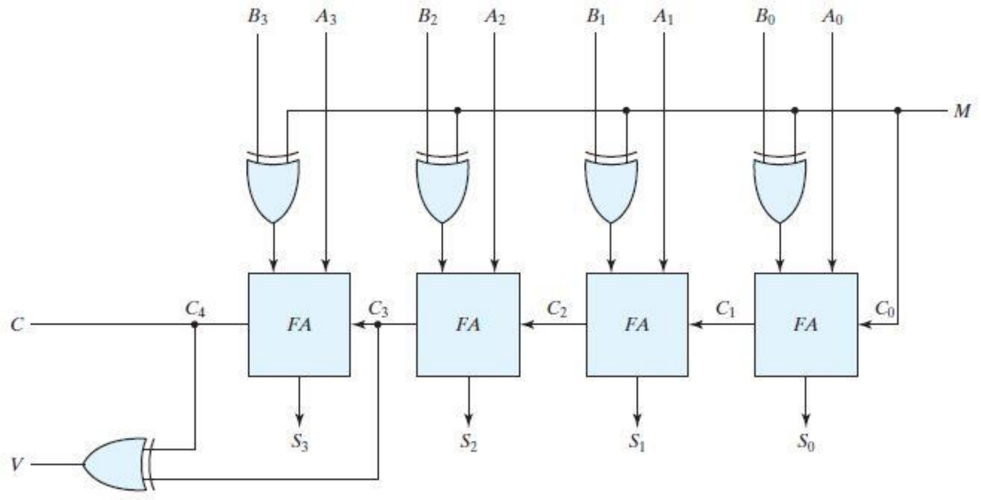
…00	0	1	0	0	0	1	0	0	A
…00	0	1	0	1	1	0	1	1	B
0	1	0	0	0	0	0	0	0	C
…00	1	0	0	1	1	1	1	1	S

Look at C_7 and C_8 in both cases

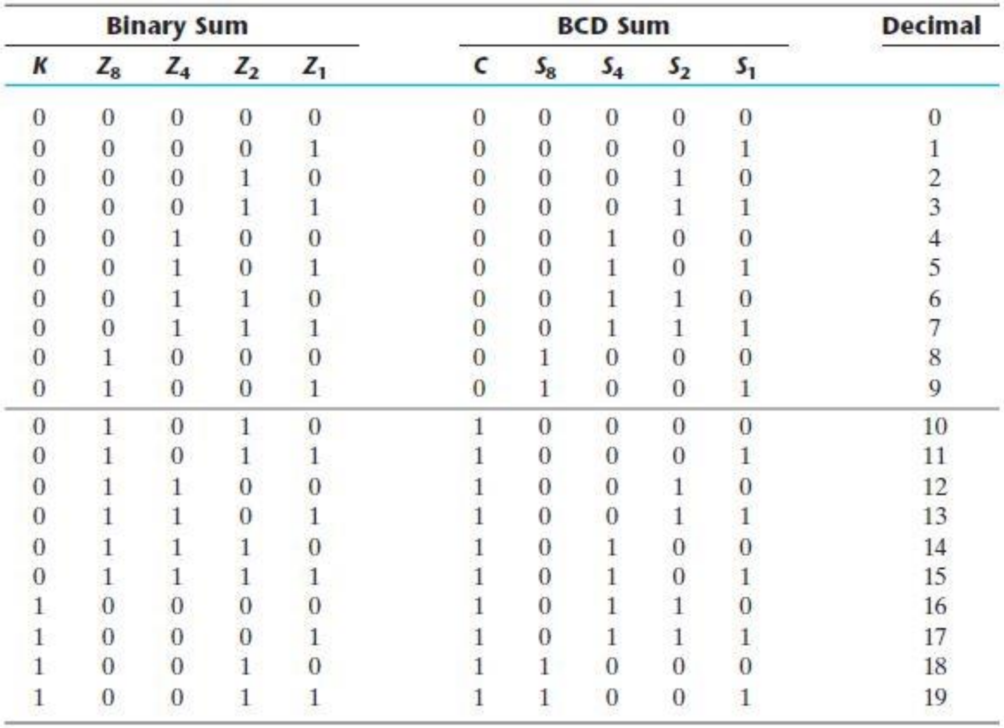
…11	1	0	1	1	1	1	0	0	A
…11	1	0	1	0	0	1	0	1	B
1	0	1	1	1	1	0	0	0	C
…11	0	1	1	0	0	0	0	1	S

Detecting Overflows: Second Method

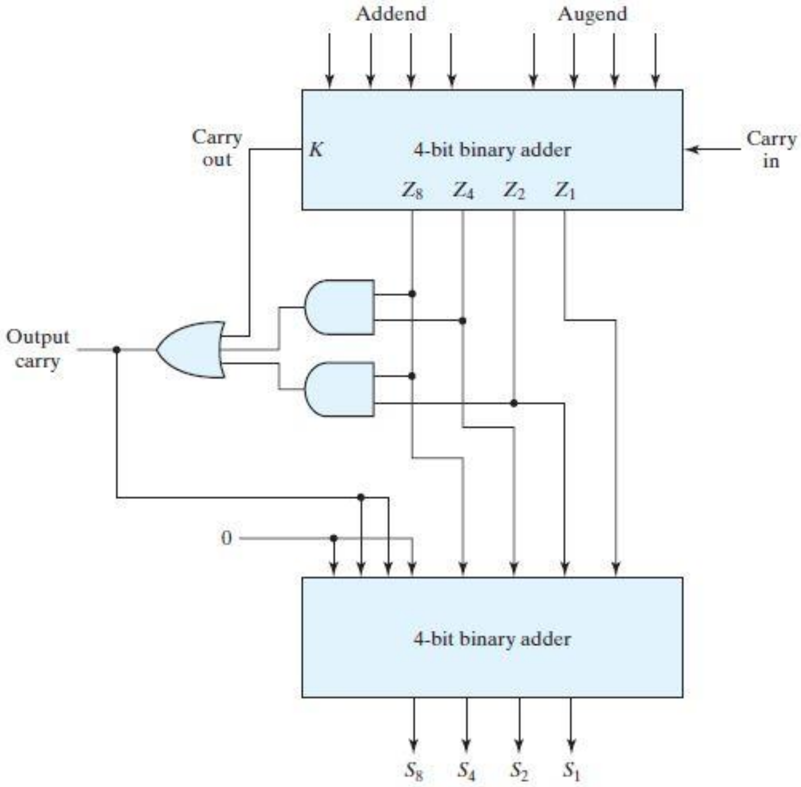
- Observations
  - Case 1: V = 1 when C_7 = 1 and $C_8 = 0$
  - Case 2: V = 1 when C_7 = 0 and $C_8 = 1$
  - $V = C_7 \oplus C_8 = 1$
  - Think about whether this could happen when the operands have different signs.
    - $C_7 = C_8$
- Overflow detection logic
  - Which one is simpler?
  - $V = C_7 \oplus C_8$
  - $V = a_7' b_7' S_7 + a_7 b_7 S_7'$



BCD Adder



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

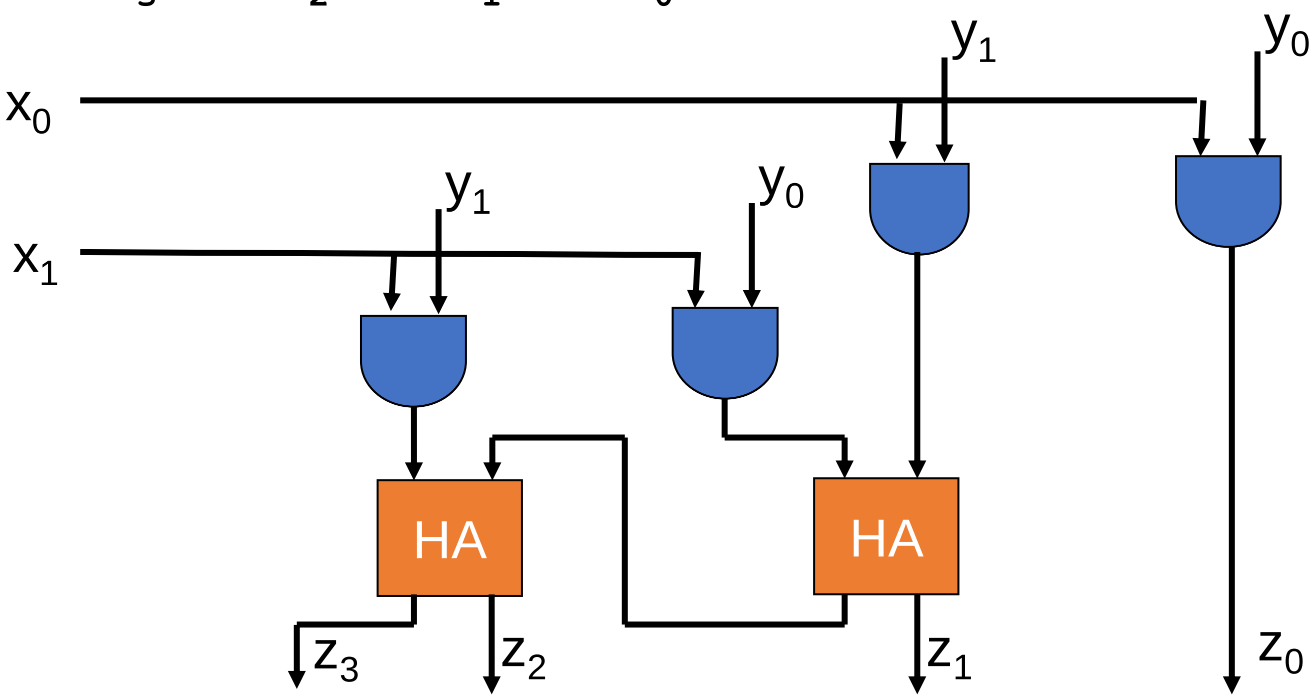


- $C = K + Z_8Z_4 + Z_8Z_2$
- Add 6 after 9

Binary Multipliers

- Two-bit multiplier

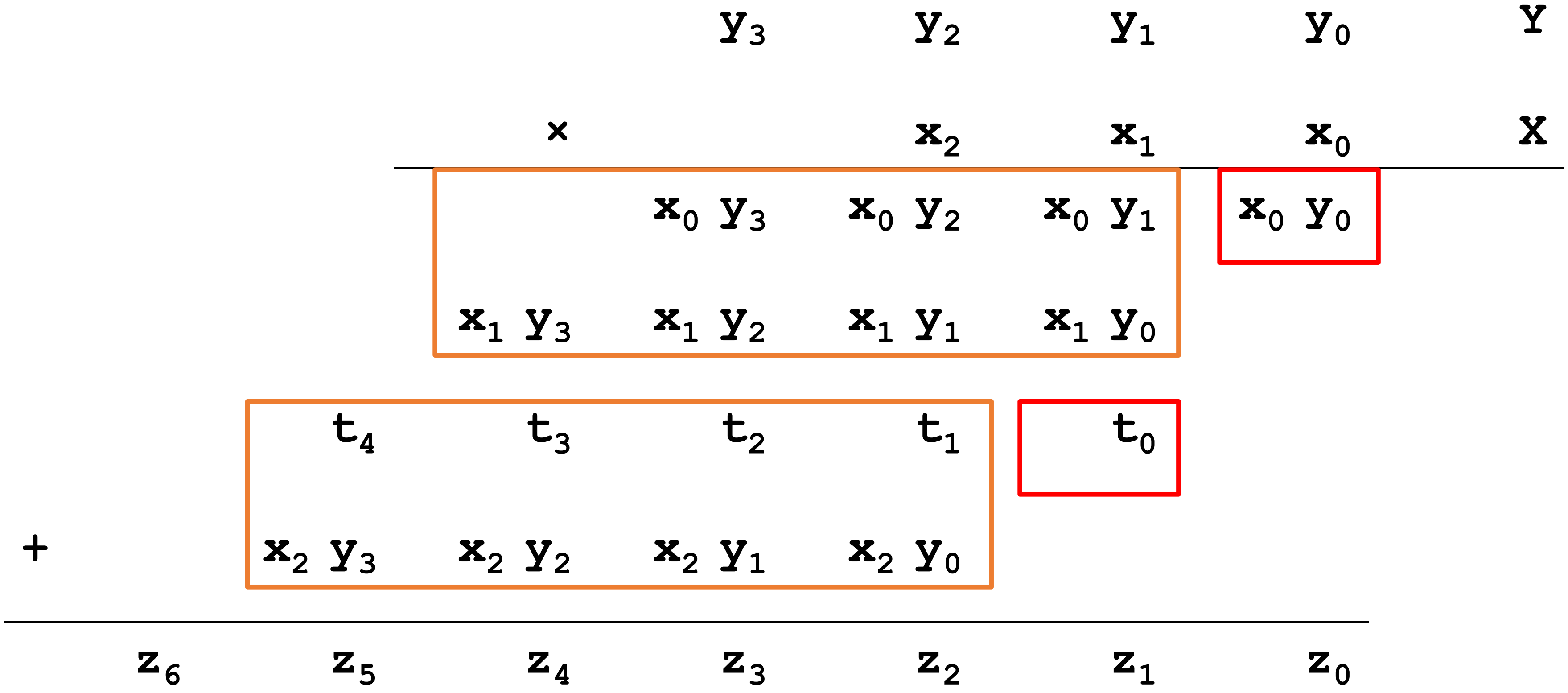
$$
\begin{array}{cccccc}
 & & & y_1 & y_0 & Y \\
 & & \times & x_1 & x_0 & X \\
\hline
 & & & x_0\,y_1 & x_0\,y_0 & \\
+ & & x_1\,y_1 & x_1\,y_0 & & \\
\hline
 & z_3 & z_2 & z_1 & z_0 & Z
\end{array}
$$



(3x4)-bit Multiplier: Method

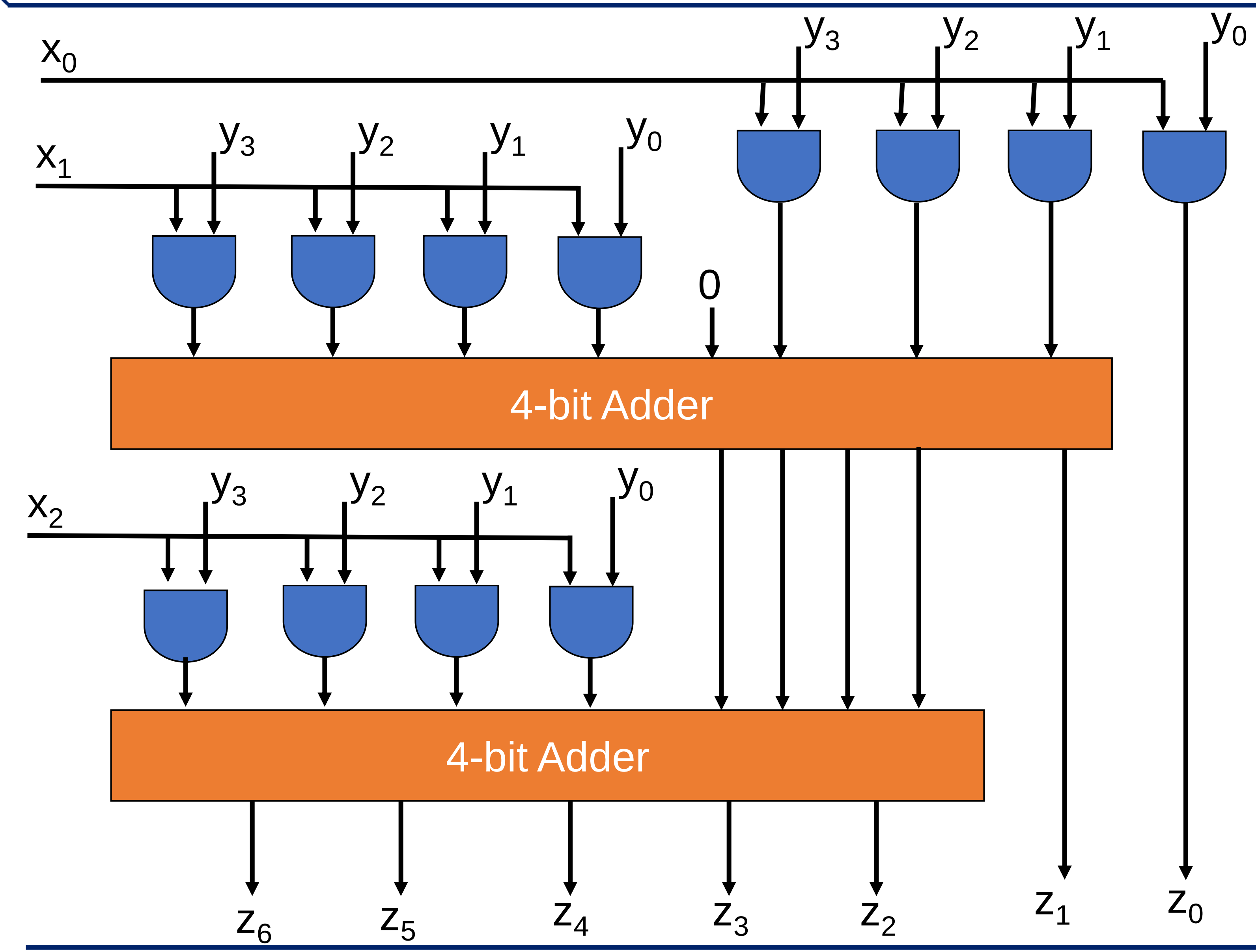
				y_3	y_2	y_1	y_0	Y
			×		x_2	x_1	x_0	X
				$x_0\ y_3$	$x_0\ y_2$	$x_0\ y_1$	$x_0\ y_0$	
			$x_1\ y_3$	$x_1\ y_2$	$x_1\ y_1$	$x_1\ y_0$		
+		$x_2\ y_3$	$x_2\ y_2$	$x_2\ y_1$	$x_2\ y_0$			
	z_6	z_5	z_4	z_3	z_2	z_1	z_0	

(3x4)-bit Multiplier: Method



$$
\begin{array}{ccccccccc}
 & & & & y_3 & y_2 & y_1 & y_0 & Y \\
 & & & \times & & x_2 & x_1 & x_0 & X \\
\hline
 & & & & x_0\,y_3 & x_0\,y_2 & x_0\,y_1 & x_0\,y_0 & \\
 & & & x_1\,y_3 & x_1\,y_2 & x_1\,y_1 & x_1\,y_0 & & \\
 & & t_4 & t_3 & t_2 & t_1 & t_0 & & \\
+ & & x_2\,y_3 & x_2\,y_2 & x_2\,y_1 & x_2\,y_0 & & & \\
\hline
 & z_6 & z_5 & z_4 & z_3 & z_2 & z_1 & z_0 &
\end{array}
$$

4-bit Multiplier: Circuit



mxn-bit Multipliers

- Generalization:
- multiplier: m-bit integer
- multiplicand: n-bit integers
- m×n AND gates
- (m-1) adders
    - each adder is n-bit

Magnitude Comparator

- Comparison of two integers: A and B.
  - $A > B \Rightarrow (1, 0, 0) = (x, y, z)$
  - $A = B \Rightarrow (0, 1, 0) = (x, y, z)$
  - $A < B \Rightarrow (0, 0, 1) = (x, y, z)$
- **Example**: 4-bit magnitude comparator
  - $A = (a_3, a_2, a_1, a_0)$ and $B = (b_3, b_2, b_1, b_0)$

1. (A = B) case
   - they are equal if and only if $a_i = b_i \quad 0 \leq i \leq 3$
   - $t_i = (a_i \oplus b_i)' \qquad 0 \leq i \leq 3$
   - $y = (A{=}B) = t_3\, t_2\, t_1\, t_0$

4-bit Magnitude Comparator

2. (A > B) and (A < B) cases
   - We compare the most significant bits of A and B first.
     - if (a_3 = 1 and b_3 = 0) ➔ A > B
     - else if (a_3 = 0 and b_3 = 1) ➔ A < B
     - else (i.e. $a_3 = b_3$) compare a_2 and b_2.

$y = (A=B) = t_3\, t_2\, t_1\, t_0$

$x = (A>B) = a_3\, b_3' + t_3\, a_2\, b_2' + t_3 t_2\, a_1\, b_1' + t_3 t_2 t_1\, a_0\, b_0'$

$z = (A<B) = a_3'\, b_3 + t_3\, a_2'\, b_2 + t_3 t_2\, a_1'\, b_1 + t_3 t_2 t_1\, a_0'\, b_0$

4-bit Magnitude Comparator: Circuit

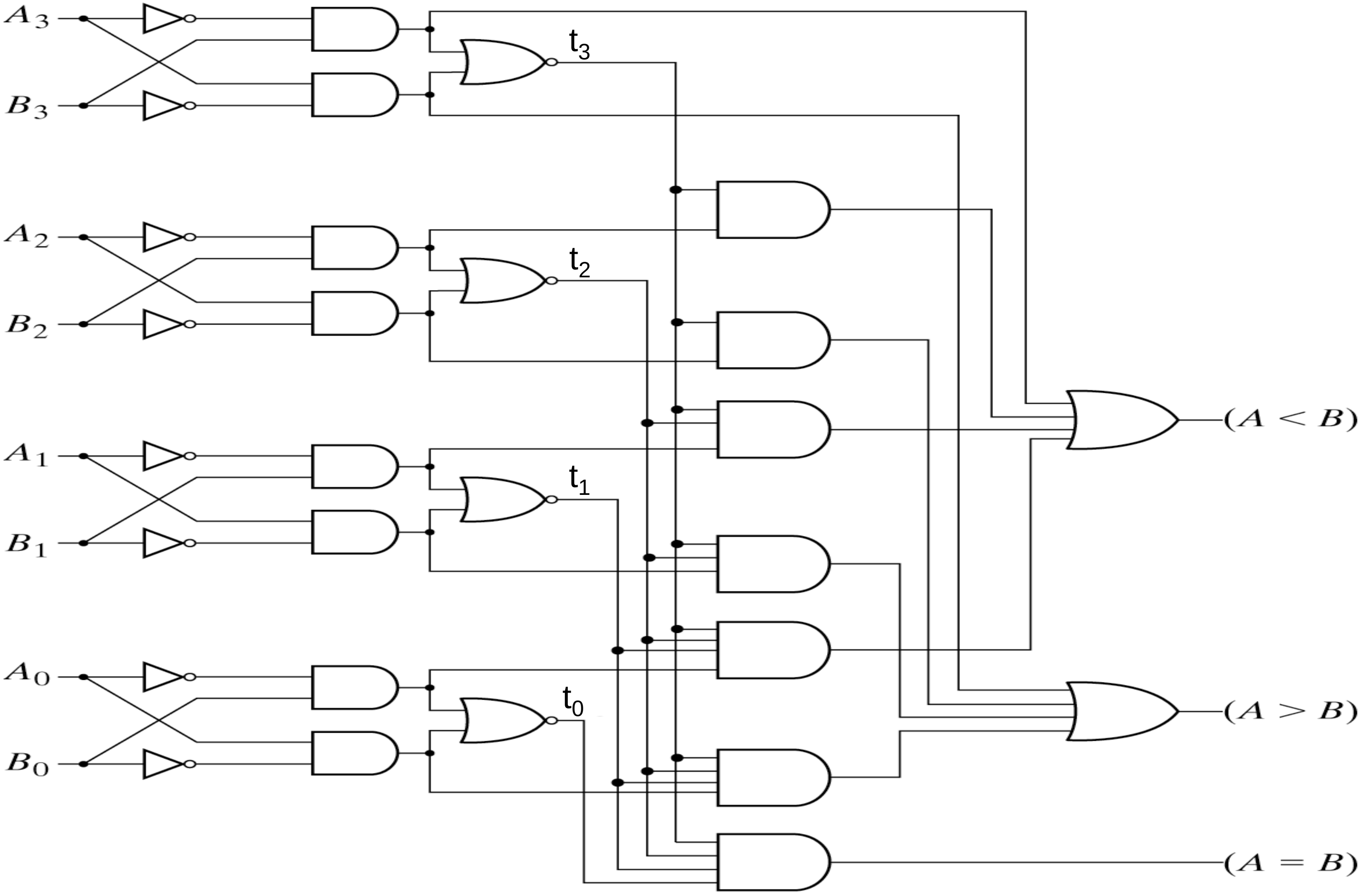
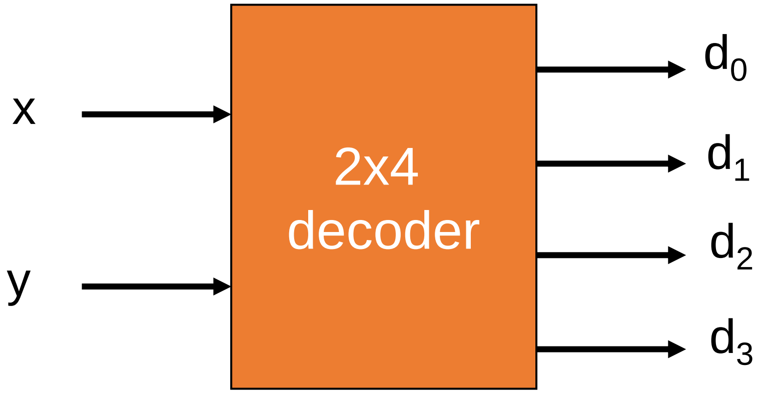


Fig. 4-17 4-Bit Magnitude Comparator

Decoders

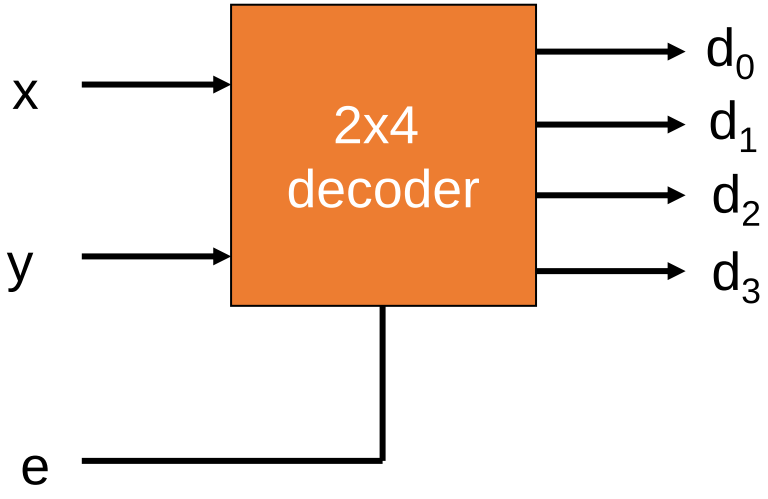
- A binary code of n bits
  - capable of representing 2^n distinct elements of coded information
  - A decoder is a combinational circuit that converts binary information from n binary inputs to a maximum of 2^n unique output lines



x	y	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

- $d_0 = x'y'$
- $d_1 = x'y$
- $d_2 = xy'$
- $d_3 = xy$

Decoder with Enable Input



e	x	y	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

2-to-4 Decoder

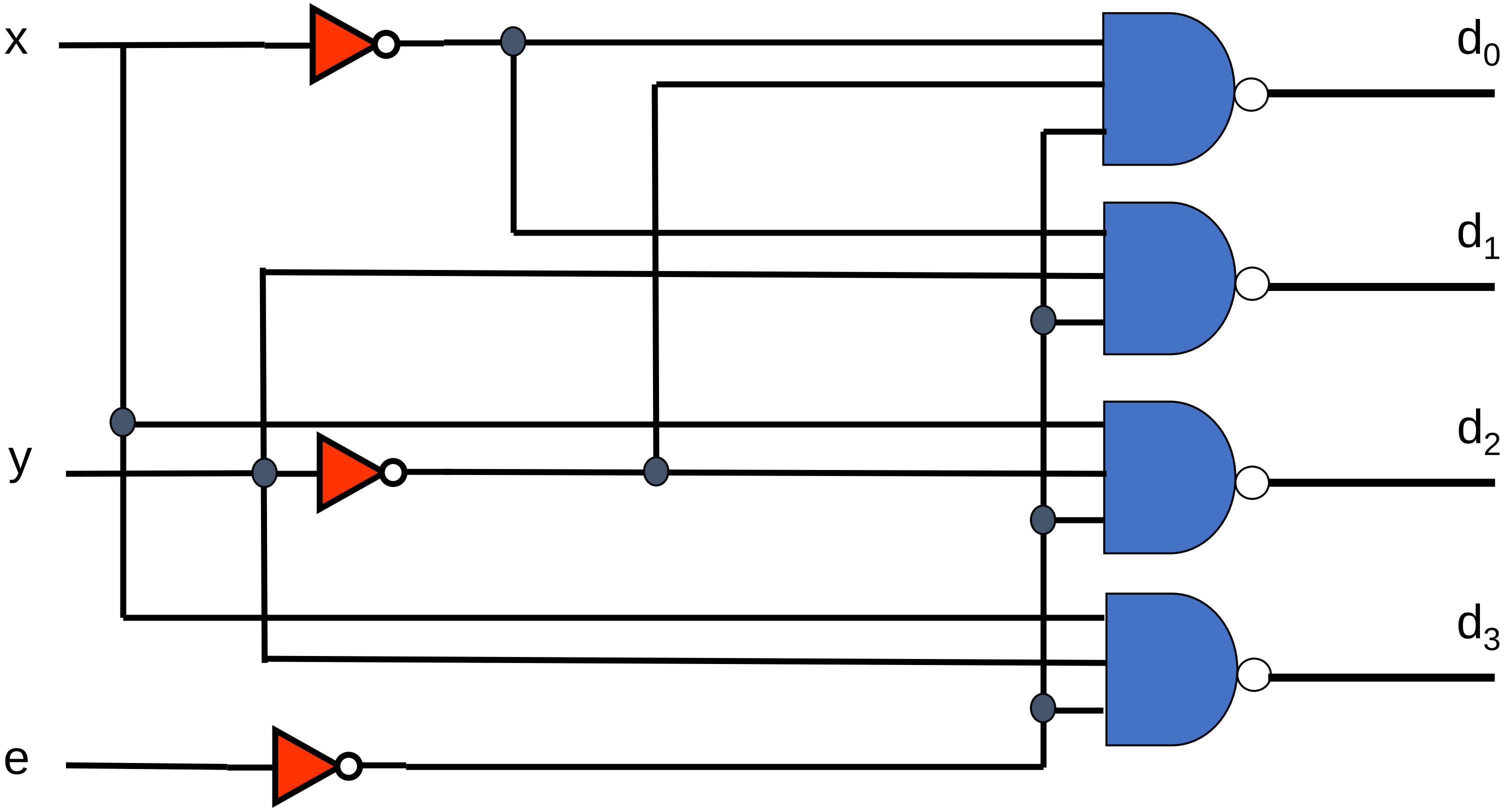
- Some decoders are constructed with NAND gates.
  - Thus, active output will be logic-0
  - They also include an "enable" input to control the circuit operation

e	x	y	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

- $d_0 = e + x + y$
- $d_1 = e + x + y'$
- $d_2 = e + x' + y$
- $d_3 = e + x' + y'$

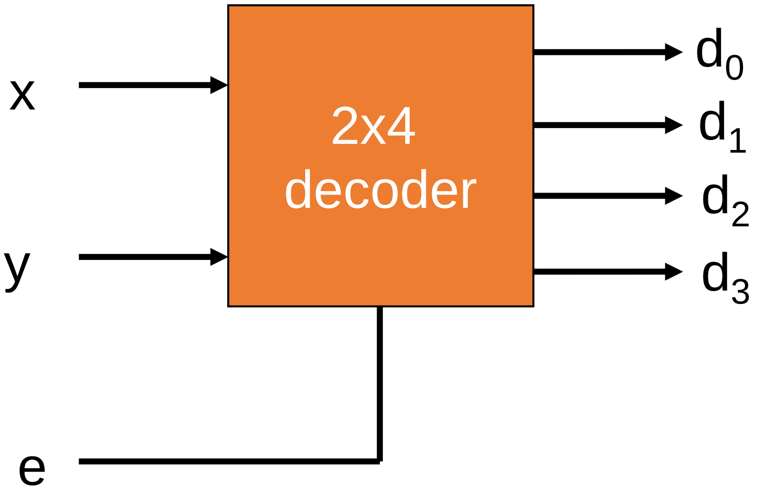
2-to-4-Line Decoder with Enable

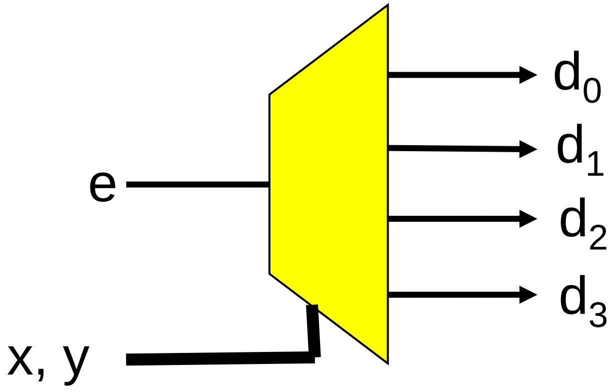
$d_0 = e + x + y =$
$d_1 = e + x + y' =$
$d_2 = e + x' + y =$
$d_3 = e + x' + y' =$



Demultiplexer

- A demultiplexer is a combinational circuit
  - it receives information from a single input line and directs it one of 2^n output lines
  - It has n selection lines as to which output will get the input





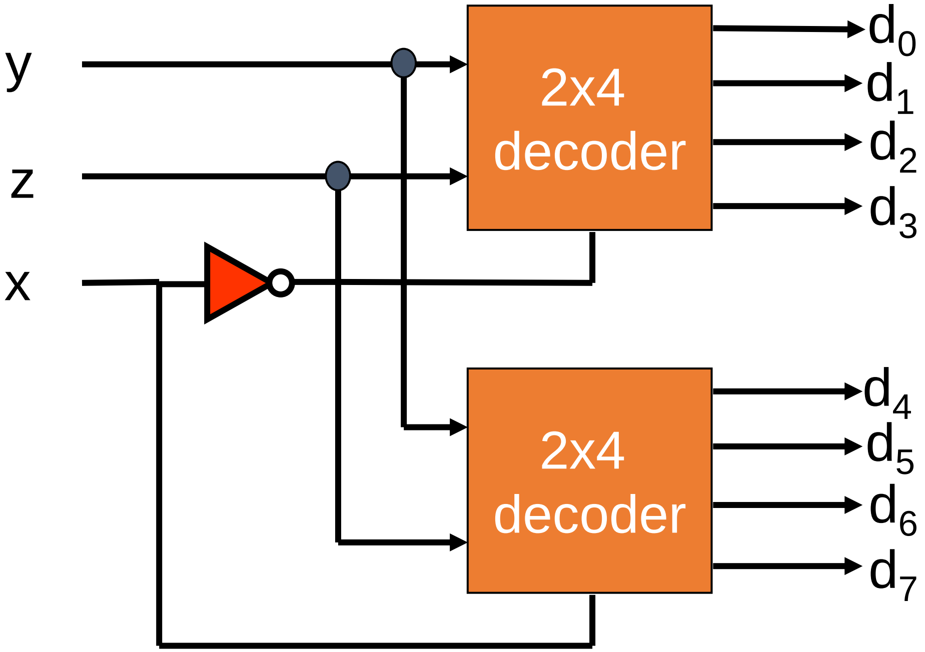
d_0 = e when x = 0 and y = 0

d_1 = e when x = 0 and y = 1

d_2 = e when x = 1 and y = 0

d_3 = e when x = 1 and y = 1

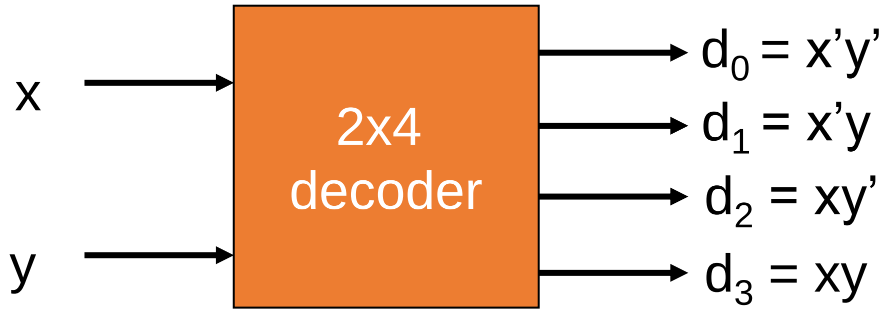
Combining Decoders



x	y	z	active output
0	0	0	d_0
0	0	1	d_1
0	1	0	d_2
0	1	1	d_3
1	0	0	d_4
1	0	1	d_5
1	1	0	d_6
1	1	1	d_7

Decoder as a Building Block

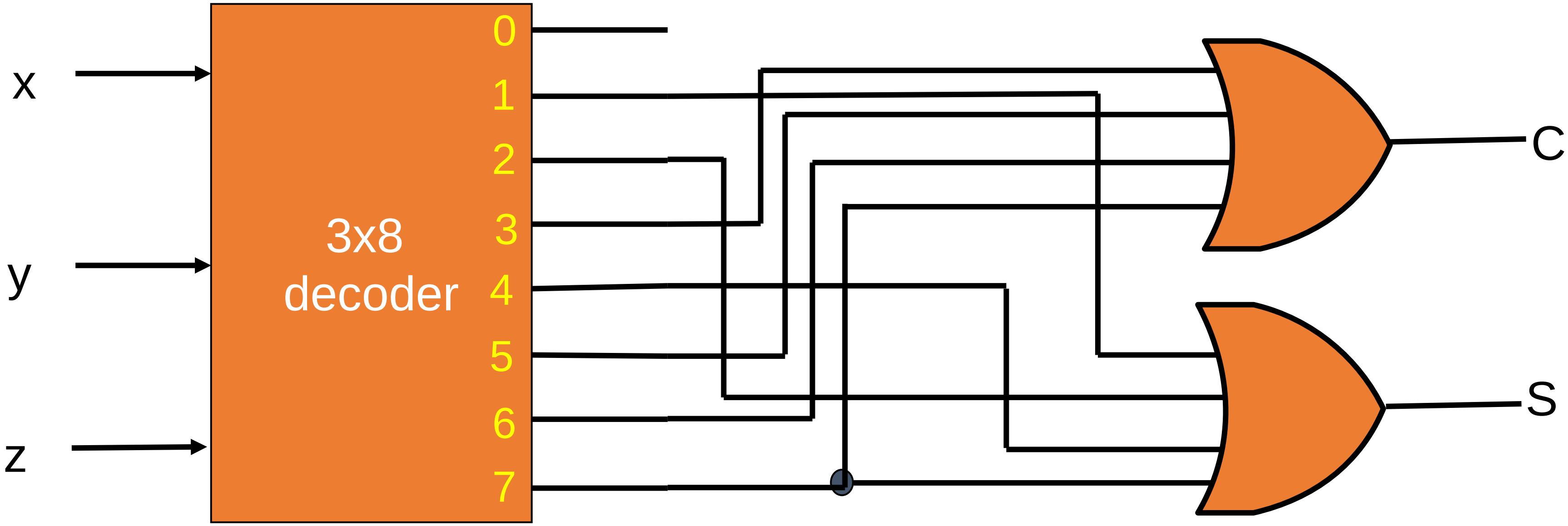
- A decoder provides the 2^n minterms of n input variable



- We can use a decoder and OR gates to realize any Boolean function expressed as sum of minterms
  - Any circuit with n inputs and m outputs can be realized using an n-to-2^n decoder and m OR gates.

Example: Decoder as a Building Block

- Full adder
  - C = xy + xz + yz = x’yz + xy’z + xyz + xyz’ = Σ(3, 5, 7, 6)
  - S = x ⊕ y ⊕ z = xy’z’ + x’y’z + xyz + x’yz’ = Σ(4, 1, 7, 2)



Encoders

- An encoder is a combinational circuit that performs the inverse operation of a decoder
  - number of inputs: 2^n
  - number of outputs: n
  - the output lines generate the binary code corresponding to the input value
- Example: n = 2

d_0	d_1	d_2	d_3	x	y
1	0	0	0	0	0
0	1	0	0	0	1
0	0	1	0	1	0
0	0	0	1	1	1

Priority Encoder

- Problem with a regular encoder:
  - only one input can be active at any given time
  - the output is undefined for the case when more than one input is active simultaneously.
- Priority encoder:
  - there is a priority among the inputs

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

4-bit Priority Encoder

- In the truth table
  - X for input variables represents both 0 and 1.
  - Good for condensing the truth table
  - Example: X100 → (0100, 1100)
    - This means d_1 has priority over d_0
    - d_3 has the highest priority
    - d_2 has the next
    - d_0 has the lowest priority
  - V = ?

Maps for 4-bit Priority Encoder

d_0d_1 \ d_2d_3	00	01	11	10
00	X	1	1	1
01	0	1	1	1
11	0	1	1	1
10	0	1	1	1

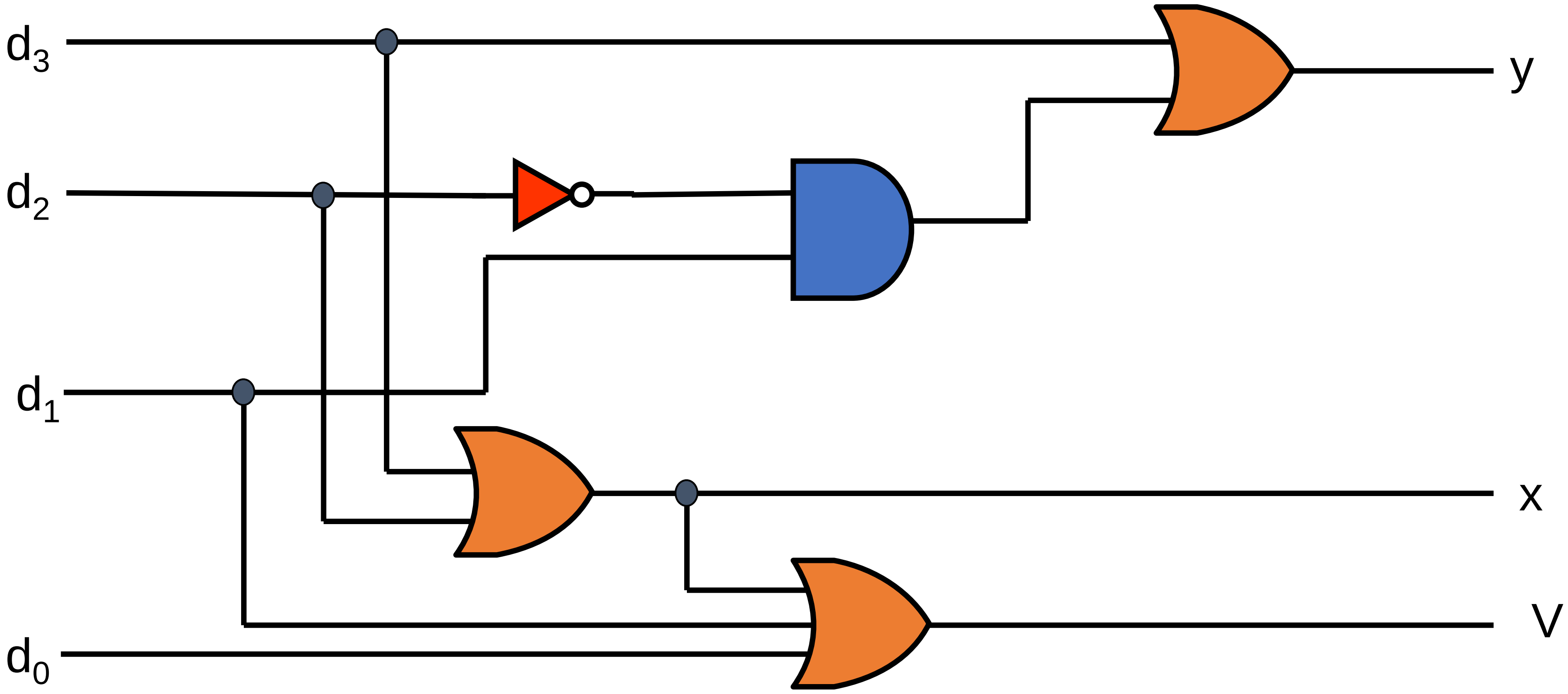
$x = d_2 + d_3$

d_0d_1 \ d_2d_3	00	01	11	10
00	X	1	1	0
01	1	1	1	0
11	1	1	1	0
10	0	1	1	0

$y = d_3 + d_1d_2'$

4-bit Priority Encoder: Circuit

- $x = d_2 + d_3$
- $y = d_1 d_2' + d_3$
- $V = d_0 + d_1 + d_2 + d_3$



Multiplexers

- A combinational circuit
  - It selects binary information from one of the many input lines and directs it to a single output line.
  - Many inputs – m
  - One output line
  - n selection lines → n = ?
- Example: 2-to-1-line multiplexer
  - 2 input lines I_0, I_1
  - 1 output line Y
  - 1 select line S

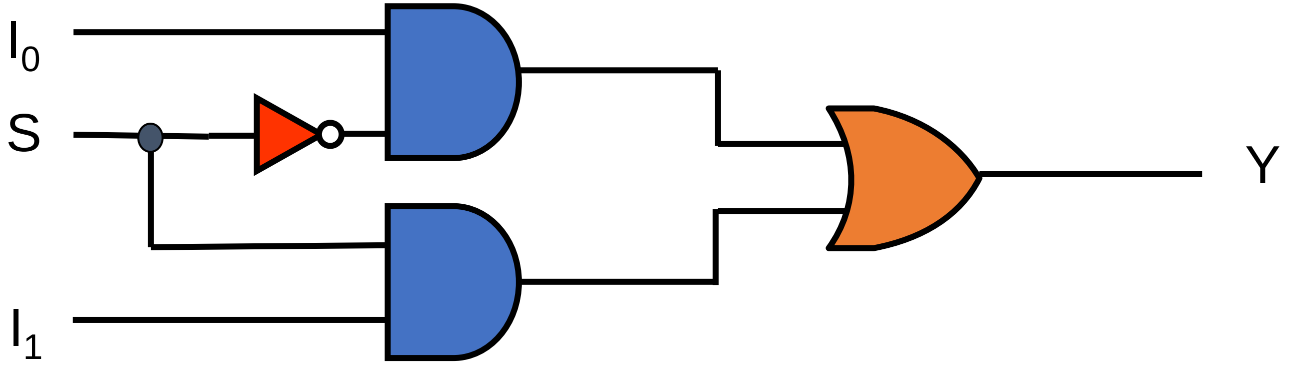
S	Y
0	I_0
1	I_1

Function Table

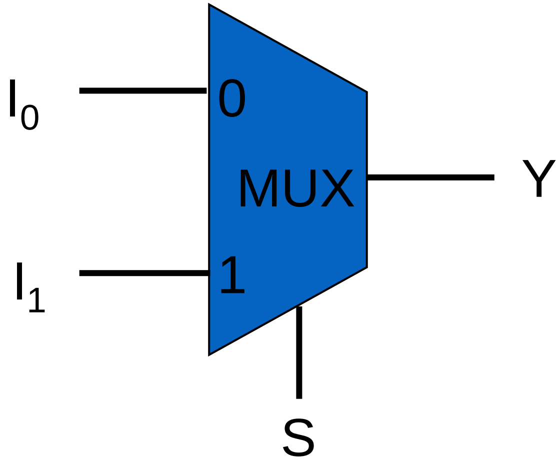
$Y = S'I_0 + SI_1$

2-to-1-Line Multiplexer

$Y = S'I_0 + SI_1$



- Special Symbol



4-to-1-Line Multiplexer

- 4 input lines: I_0, I_1, I_2, I_3
- 1 output line: Y
- 2 select lines: S_1, S_0.

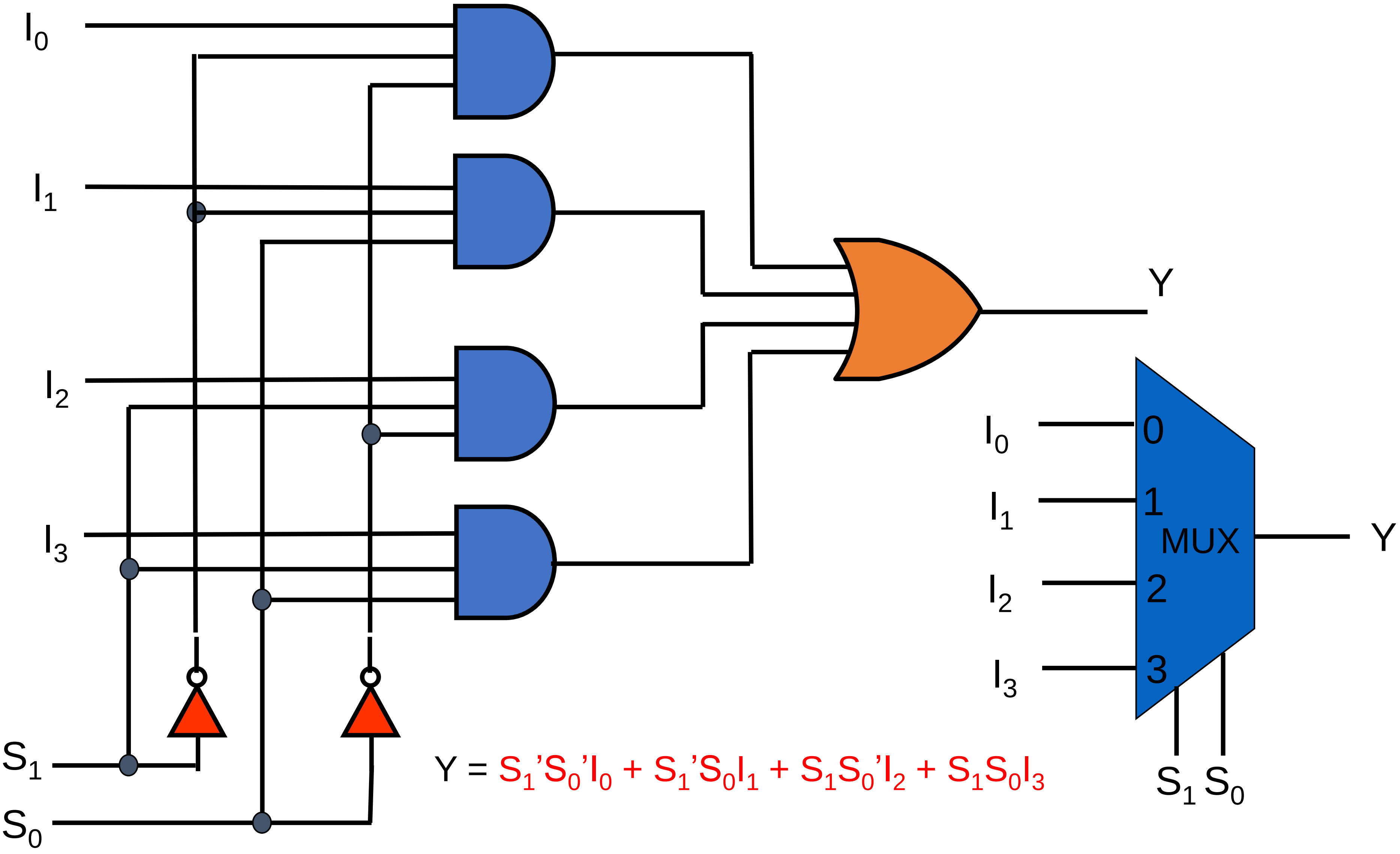
S_1	S_0	Y
0	0	
0	1	
1	0	
1	1	

$$Y = S_1'S_0'I_0 + S_1'S_0I_1 + S_1S_0'I_2 + S_1S_0I_3$$

Interpretation:

- In case $S_1 = 0$ and $S_0 = 0$, Y selects I_0
- In case $S_1 = 0$ and $S_0 = 1$, Y selects I_1
- In case $S_1 = 1$ and $S_0 = 0$, Y selects I_2
- In case $S_1 = 1$ and $S_0 = 1$, Y selects I_3

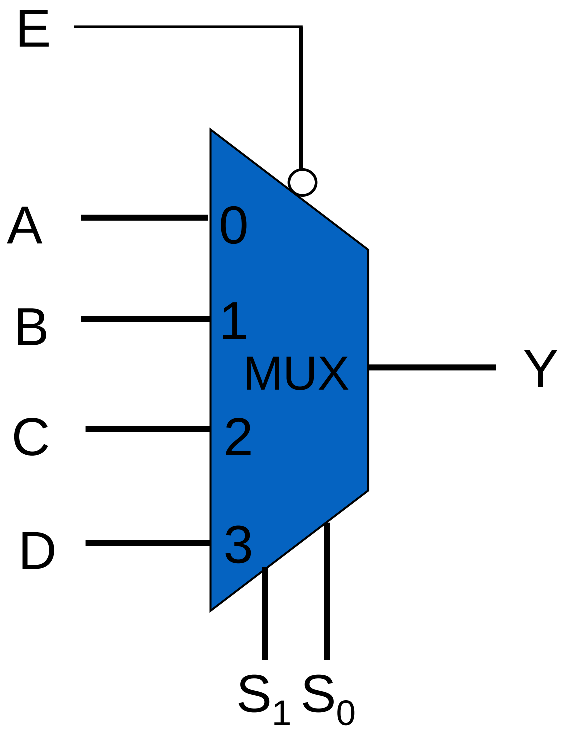
4-to-1-Line Multiplexer: Circuit



$Y = S_1'S_0'I_0 + S_1'S_0I_1 + S_1S_0'I_2 + S_1S_0I_3$

Multiplexer with Enable Input

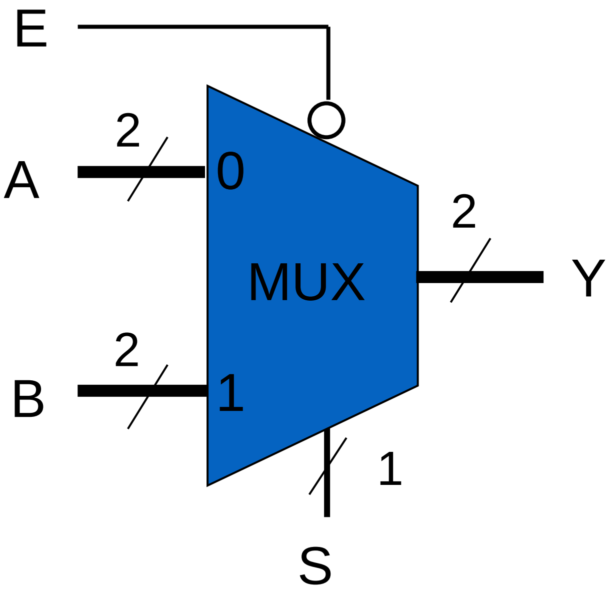
- To select a certain building block we use enable inputs



E	S	Y
1	XX	0
0	00	A
0	01	B
0	10	C
0	11	D

Multiple-Bit Selection Logic 1/2

- A multiplexer is also referred as a “data selector”
- A multiple-bit selection logic selects a group of bits



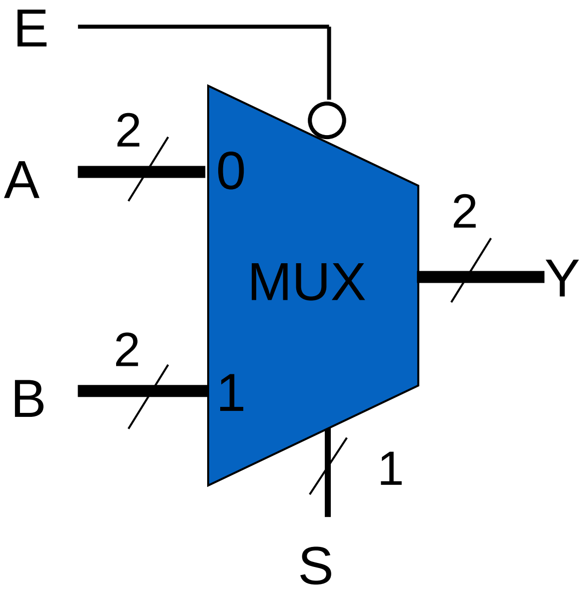
$A = a_1a_0$

$B = b_1b_0$

If S=0: $Y = a_1a_0$

If S=1: $Y = b_1b_0$

Multiple-bit Selection Logic 2/2

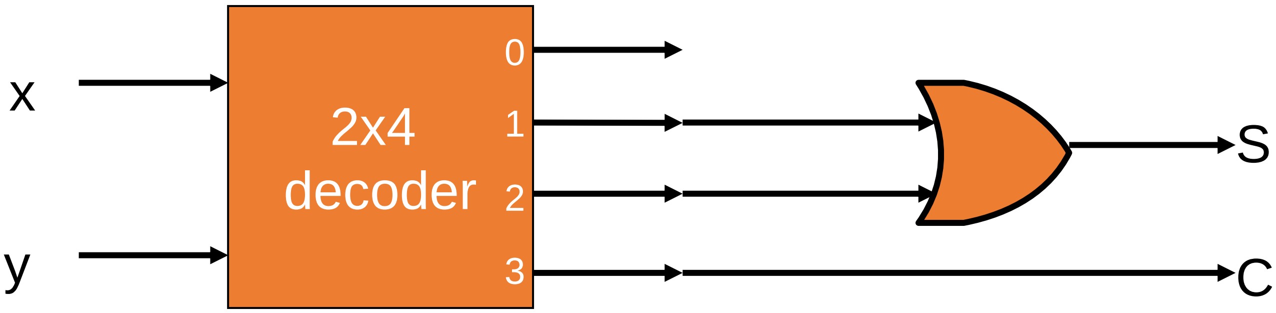


E	S	Y
1	X	all 0’s
0	0	A
0	1	B

Design with Multiplexers 1/2

- Reminder: design with decoders

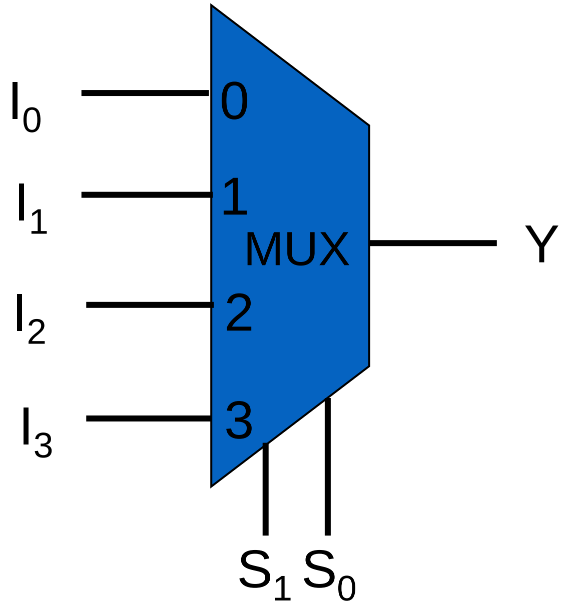
- Half adder
  - $C = xy = \Sigma(3)$
  - $S = x \oplus y = x'y + xy' = \Sigma(1, 2)$



- A closer look will reveal that a multiplexer is nothing but a decoder with OR gates

Design with Multiplexers 2/2

- 4-to-1-line multiplexer

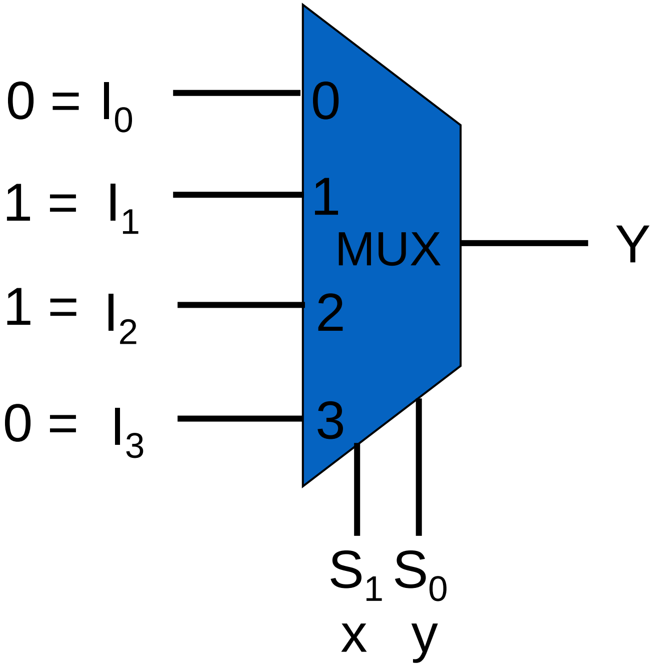


- $S_1 \rightarrow x$
- $S_0 \rightarrow y$
- $S_1'S_0' = x'y'$,
- $S_1'S_0 = x'y$,
- $S_1S_0' = xy'$,
- $S_1S_0 = xy$

- $Y = S_1'S_0' I_0 + S_1'S_0 I_1 + S_1S_0' I_2 + S_1S_0 I_3$.
- $Y = x'y' I_0 + x'y I_1 + xy' I_2 + xyI_3$
- $Y =$

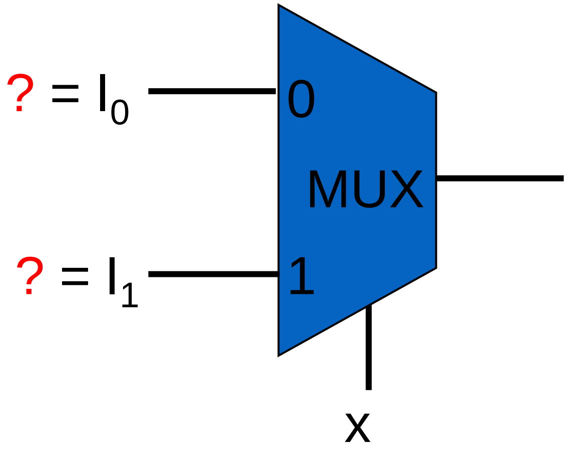
Example: Design with Multiplexers

- Example: S(x, y) = $\Sigma(1, 2)$



Design with Multiplexers Efficiently

- More efficient way to implement an n-variable Boolean function
  1. Use a multiplexer with n-1 selection inputs
  2. First (n-1) variables are connected to the selection inputs
  3. The remaining variable is connected to data inputs
- Example: $Y = \Sigma(1, 2) = x'y + xy'$



$Y = S' I_0 + S I_1$

$Y = x'I_0 + x I_1$

Example: Design with Multiplexers

- $F(x, y, z) = \Sigma(1, 2, 6, 7)$
  - $F = x'y'z + x'yz' + xyz' + xyz$
  - $Y = S_1'S_0' I_0 + S_1'S_0 I_1 + S_1S_0' I_2 + S_1S_0 I_3$
  - $I_0 = z$, $I_1 = z'$, $I_2 = 0$, $I_3 = ?$

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	

Example: Design with Multiplexers

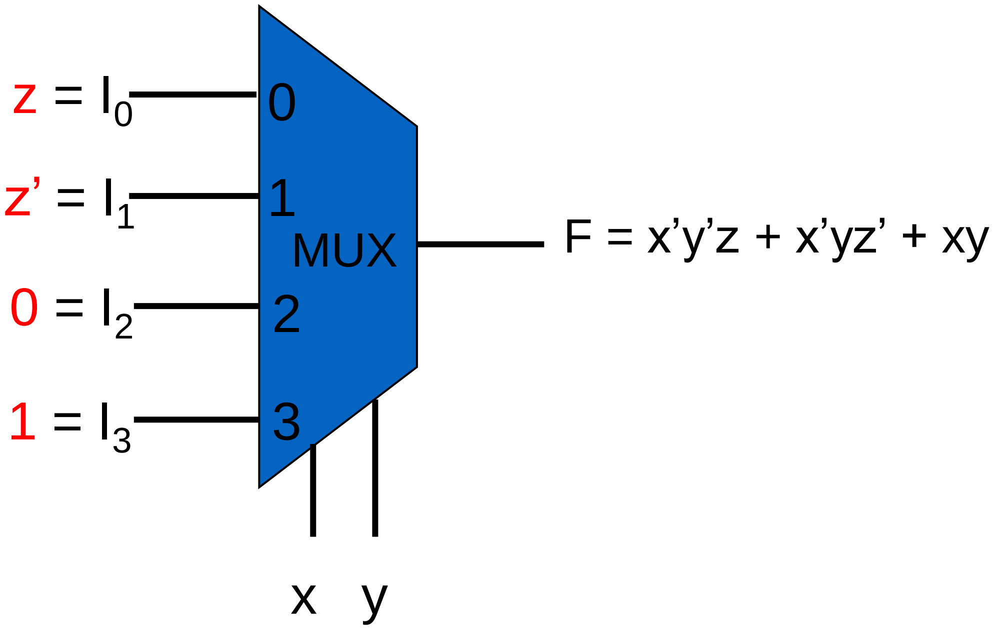
F = x'y'z + x'yz' + xyz' + xyz

F = z when x = 0 and y = 0

F = z' when x = 0 and y = 1

F = 0 when x = 1 and y = 0

F = 1 when x = 1 and y = 1

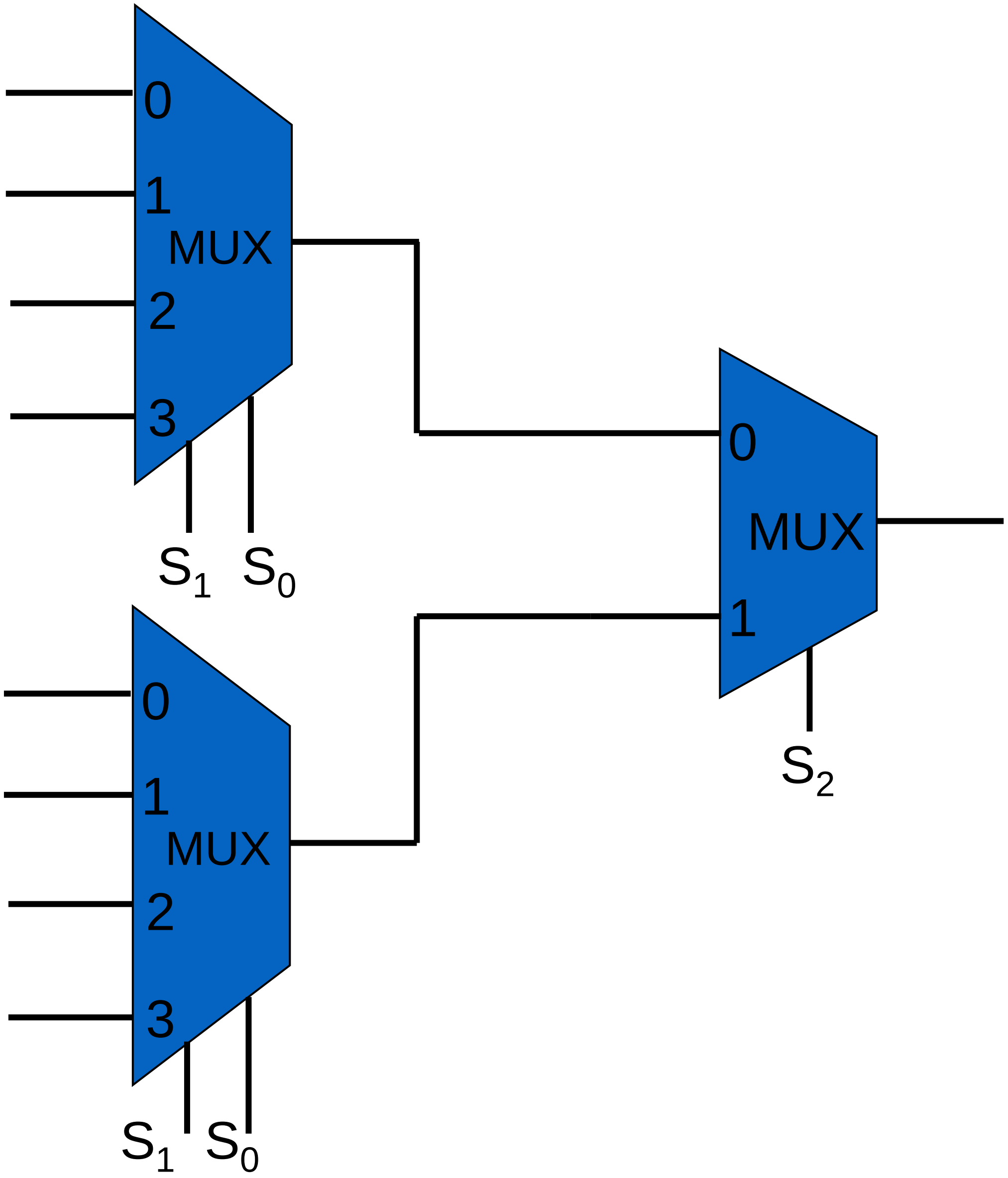


Design with Multiplexers

- General procedure for n-variable Boolean function
  - $F(x_1, x_2, ..., x_n)$

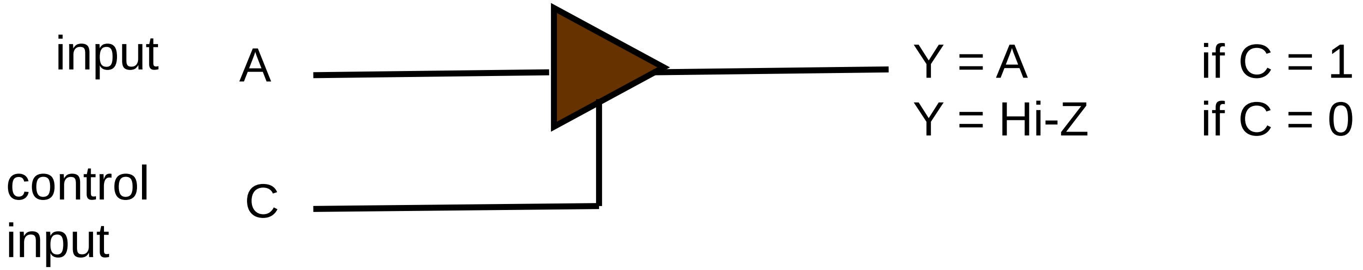
1. The Boolean function is expressed in a truth table
2. The first (n-1) variables are applied to the selection inputs of the multiplexer $(x_1, x_2, ..., x_{n-1})$
3. For each combination of these (n-1) variables, evaluate the value of the output as a function of the last variable, x_n.
   - $0, 1, x_n, x_n'$
4. These values are applied to the data inputs in the proper order.

Combining Multiplexers



Three-State Buffers

- A different type of logic element
  - Instead of two states (i.e. 0, 1), it exhibits three states (0, 1, Z)
  - Z (Hi-Z) is called high-impedance
  - When in Hi-Z state the circuit behaves like an open circuit (the output appears to be disconnected, and the circuit has no logic significance)

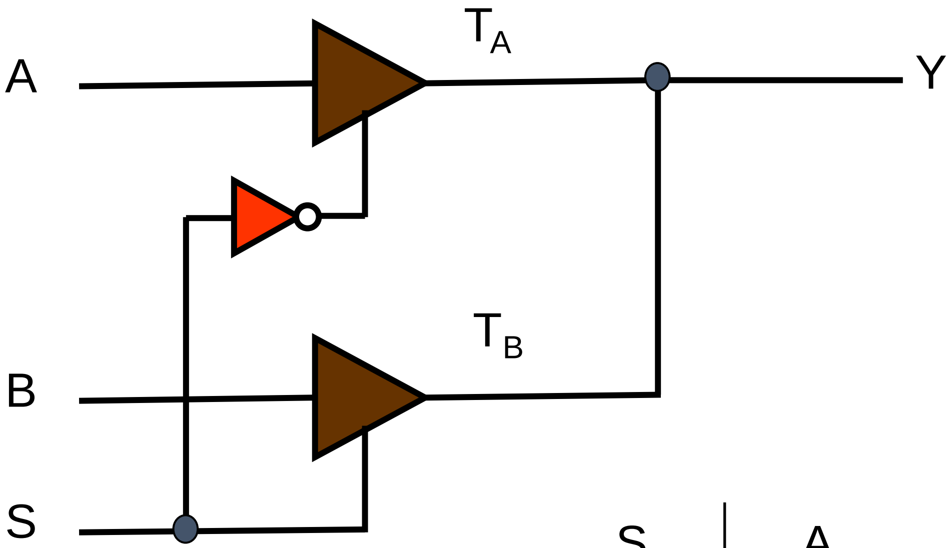


3-State Buffers

- Remember that we cannot connect the outputs of other logic gates.
- We can connect the outputs of three-state buffers
  - provided that no two three-state buffers drive the same wire to opposite 0 and 1 values at the same time.

C	A	Y
0	X	Hi-Z
1	0	0
1	1	1

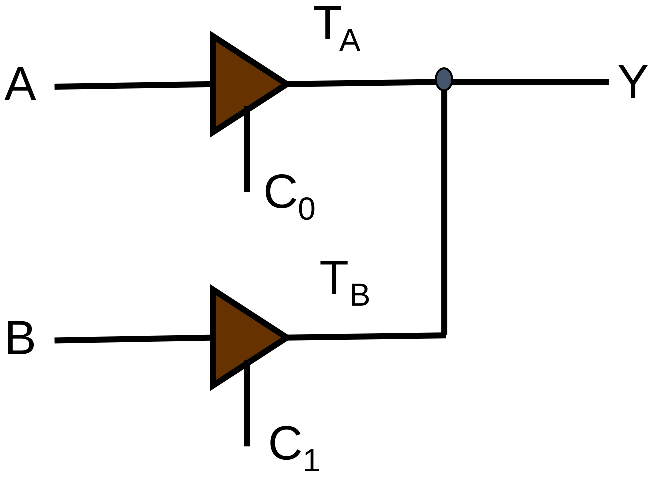
Multiplexing with 3-State Buffers



S	A	B	T_A	T_B	Y
0	0	X	0	Z	0
0	1	X	1	Z	1
1	X	0	Z	0	0
1	X	1	Z	1	1

It is, in fact, a
2-to-1-line MUX

Two Active Outputs - 1

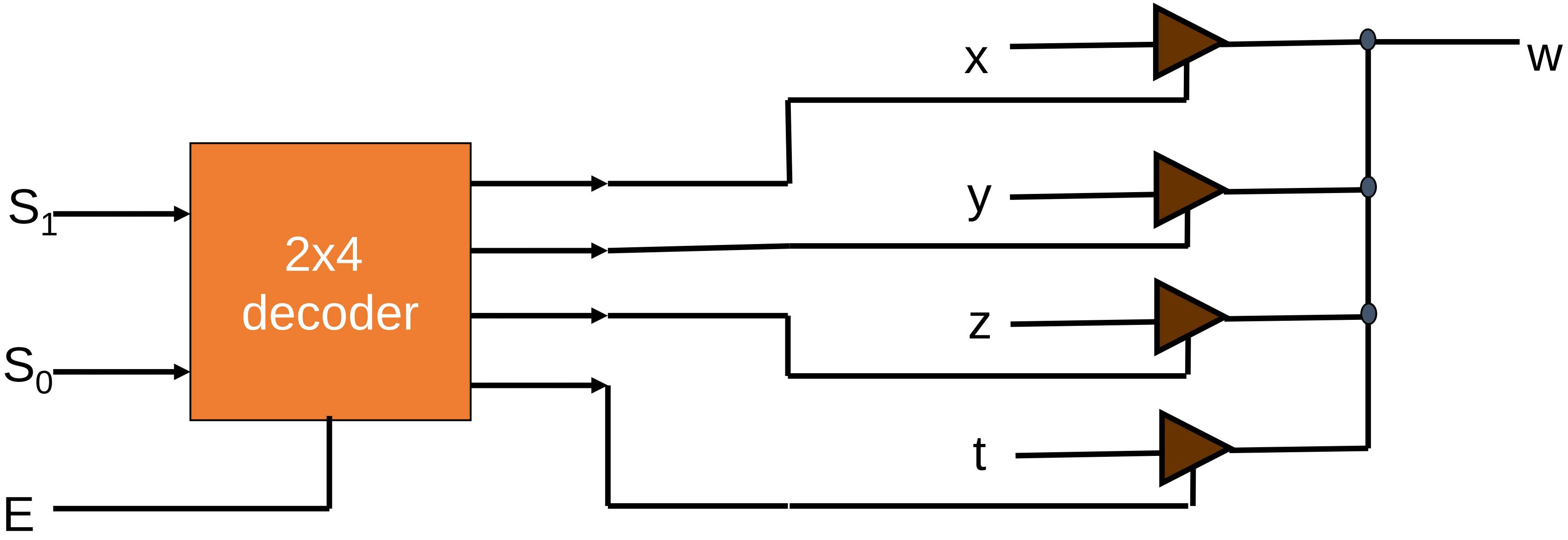


What will happen if $C_1 = C_0 = 1$?

C_1	C_0	A	B	Y
0	0	X	X	Z
0	1	0	X	0
0	1	1	X	1
1	0	X	0	0
1	0	X	1	1
1	1	0	0	0
1	1	1	1	1
1	1	0	1	
1	1	1	0	

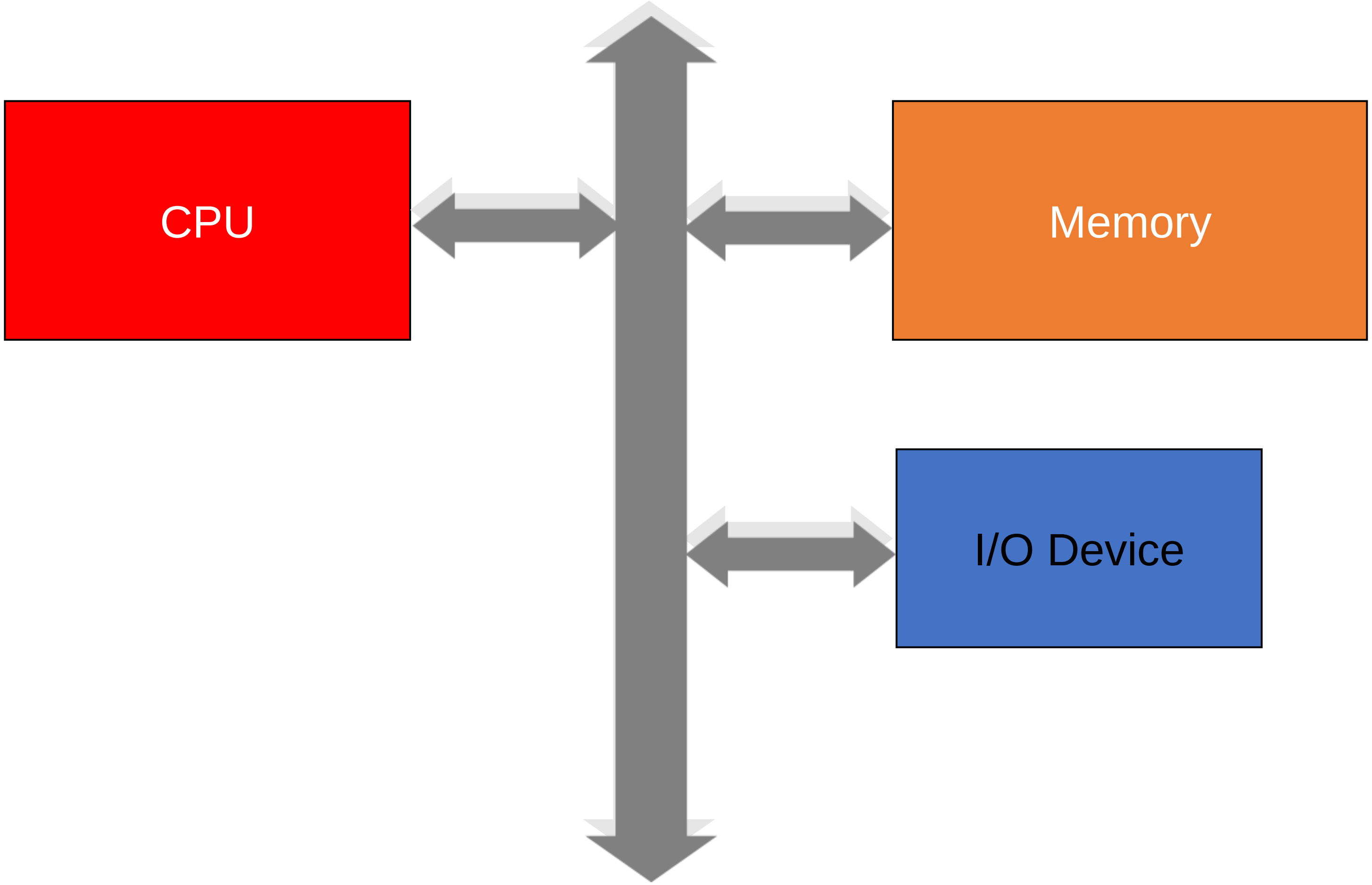
Design Principle with 3-State Buffers

- Designer must be sure that only one control input must be active at a time.
  - Otherwise, the circuit may be destroyed by the large amount of current flowing from the buffer output at logic-1 to the buffer output at logic-0.



Busses with 3-State Buffers

- There are important uses of three-state buffers



Design with Verilog

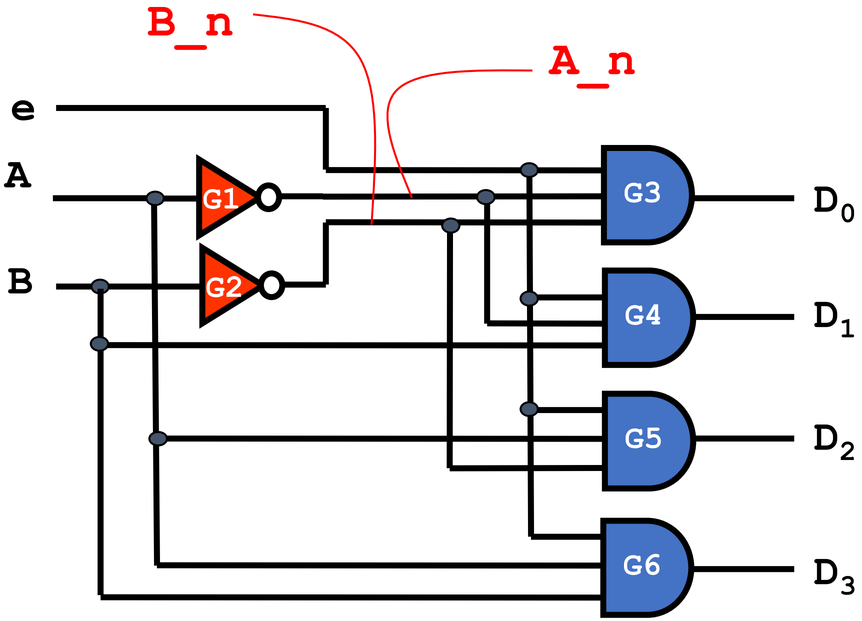
- Gate Level Design (Decoder with enable input)

D_0 = eA’B’
D_1 = eA’B
D_2 = eAB’
D_3 = eAB

```
module decoder_2x4_gates(D, A, B, e);
   output [0:3] D;
   input  A, B, e;
   wire A_n, B_n;

   not G1(A_n, A);
   not G2(B_n, B);

   and G3(D[0], e, A_n, B_n);
   and G4(D[1], e, A_n, B);
   and G5(D[2], e, A, B_n);
   and G6(D[3], e, A, B);
endmodule;
```

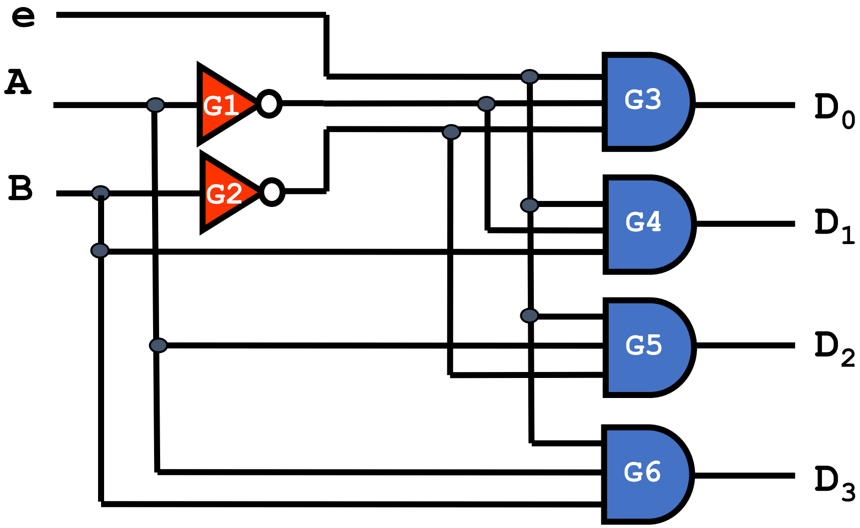


Dataflow Modeling

- Dataflow modeling uses a number of operators that act on operands
  - About 30 different operators:
    See the textbook for the operators
  - Describes combinational circuits by their functions rather than their gate structure

D_0 = eA'B'
D_1 = eA'B
D_2 = eAB'
D_3 = eAB

```
module decoder_2x4_dataflow(
    output [0:3] D,
    input   A, B,
        e);
   assign D[0] = e & ~A & ~B;
   assign D[1] = e & ~A & B;
   assign D[2] = e & A & ~B;
   assign D[3] = e & A & B;
endmodule;
```



Dataflow Modeling

- Data type “**`net`**”
  - Represents a physical connection between circuit elements
  - e.g., “**`wire`**”, “**`output`**”, “**`input`**”.
- Continuous assignment “**`assign`**”
  - A statement that assigns a value to a net
  - `e.g., assign D[0] = e & ~A & ~B;`
  - `e.g., assign A_lt_B = A < B;`
- Bus type
  - `wire [0:3] T;`
  - `T[0], T[3], T[1..2];`

Behavioral Modeling

- Represents digital circuits at a functional and algorithmic level
  - Mostly used to describe sequential circuits
  - Can also be used to describe combinational circuits

```
module mux_2x1_beh(m, A, B, S);
  output m;
  input A, B, S;
  reg m;
  always @(A or B or S);
   if(S == 1) m = A;
   else m = B;
endmodule;
```

Behavioral Modeling

- Output must be declared as “**reg**” data type
- “**always**” block
    - Procedural assignment statements are executed every time there is a change in any of the variables listed after the “@” symbol (i.e., sensitivity list).

```
module mux_4x1_beh(output reg m,
  input I0, I1, I2, I3;
  input [1:0] S);
  always @(I0 or I1 or I2 or I3 or S);
   case (S)
       2’b00:    m = I0;
       2’b01:    m = I1;
       2’b10:    m = I2;
       2’b11:    m = I3;
   endcase
endmodule;
```