Combinational Logic

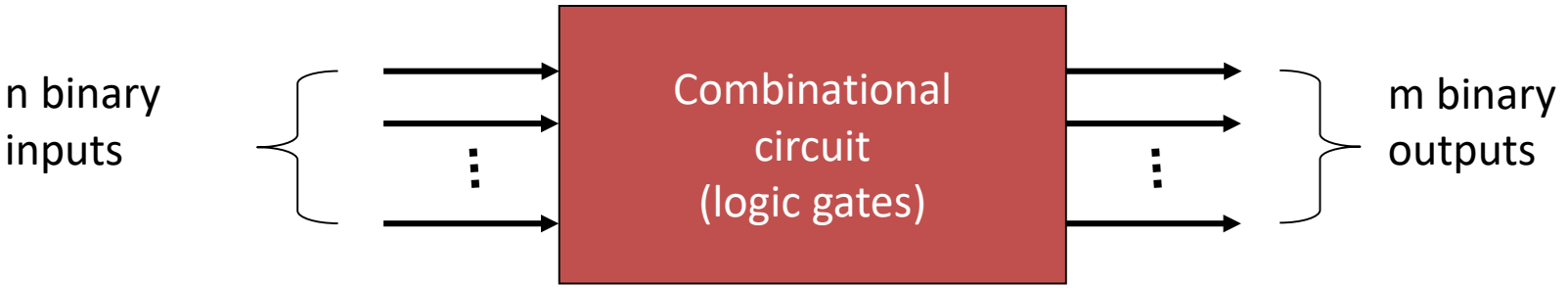
Mantıksal Tasarım – BBM231

section instructor:
Ufuk Çelikcan

Classification

1. Combinational
   - no memory
   - outputs depend only on 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 inputs and current state
   - state is a function of previous inputs
   - >> So outputs not only depend on 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, multiplexers.

Analysis & Design of Combinational Logic

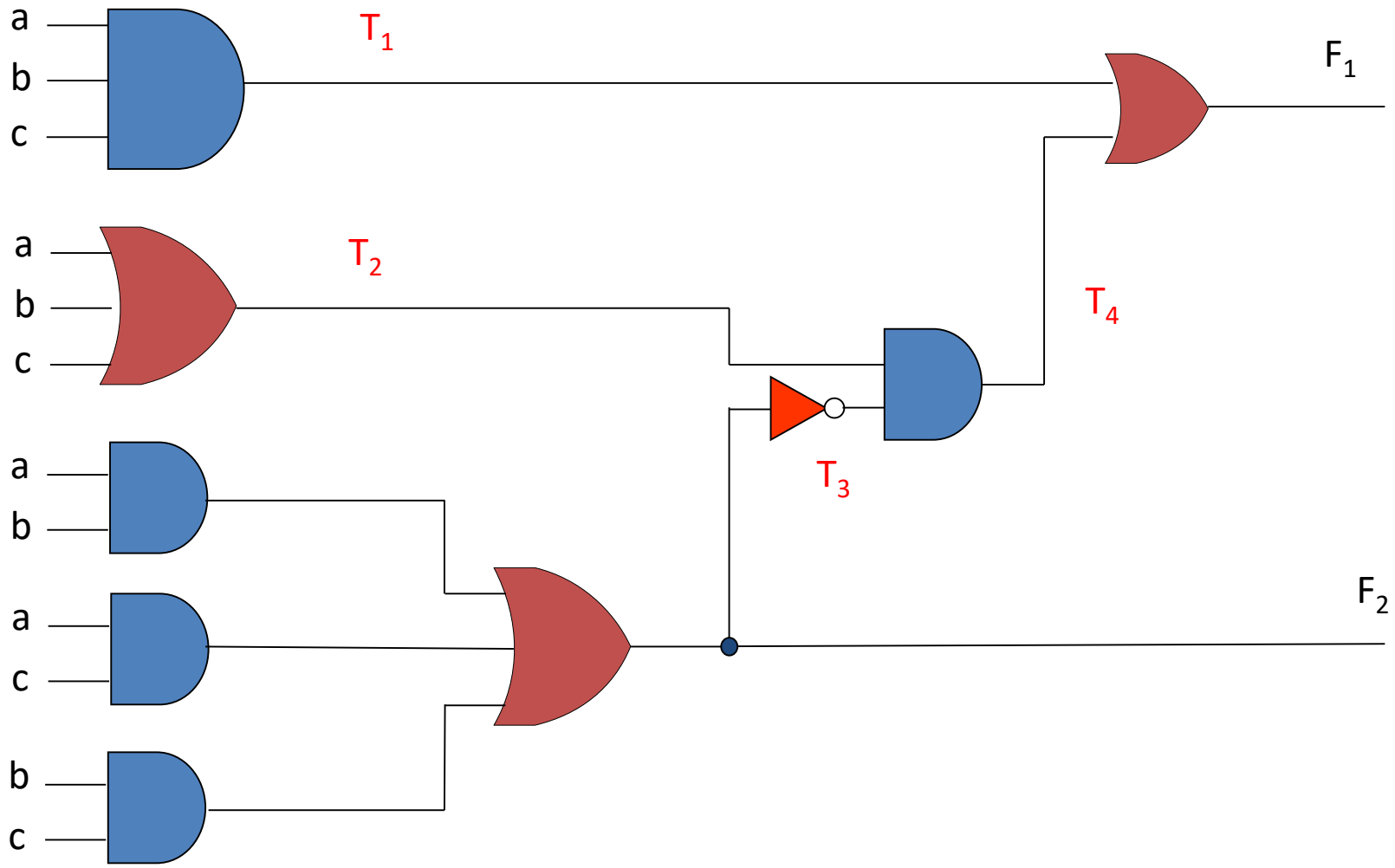
- Analysis: to find out the function that a given circuit implements
  - We are given a logic circuit and
  - we are expected to find out
    1. Boolean function(s)
    2. Truth table
    3. A possible explanation of the circuit operation (i.e. **what it does**)

Analysis of Combinational Logic

- First, make sure that the given circuit is, indeed, combinational.
  - Verifying the circuit is combinational
    - ✓No memory elements
    - ✓No feedback paths (connections)
- Second, obtain a Boolean function for each output or the truth table
- Finally, interpret the operation of the circuit from the derived Boolean functions or truth table
  - What is it the circuit doing?
    - Addition, subtraction, multiplication, comparison etc.

Obtaining Boolean Function

Example



Example: Obtaining Boolean Function

- Boolean expressions for named wires
  - $T_1 = abc$
  - $T_2 = a + b + c$
  - $F_2 =$
  - $T_3 =$
  - $T_4 = T_3 T_2$
  - $F_1 = T_1 + T_4$

    $=$

    $=$

    $=$

    $=$

Example: Obtaining Boolean Function

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

Example: Obtaining Boolean Function

- Boolean expressions for outputs
  - $F_2 = ab + ac + bc$
  - $F_1 =$
  - $F_1 =$
  - $F_1 =$
  - $F_1 =$

Example: Obtaining Boolean Function

- Boolean expressions for outputs
  - $F_2 = ab + ac + bc$
  - $F_1 = abc + (a'b' + a'c' + b'c')\,(a + b + c)$
  - $F_1 = abc + a'b'c + a'bc' + ab'c'$
  - $F_1 = a(bc + b'c') + a'(b'c + bc')$
  - $F_1 = a(b \oplus c)' + a'(b \oplus c)$
  - $F_1 = (a \oplus b \oplus c)$: Odd Function

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	1	0	1	1	1		
0	1	0	0	1	1	1		
0	1	1	0	1	0	0		
1	0	0	0	1	1	1		
1	0	1	0	1	0	0		
1	1	0	0	1	0	0		
1	1	1	1	1	0	0		

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 even faulty)
    - Wrong interpretations can result in incorrect circuit
  - We are expected to find
    1. 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 simplified 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, all realizing the same function.
- 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	1				
0	0	1	0				
0	0	1	1				
0	1	0	0				
0	1	0	1				
0	1	1	0				
0	1	1	1				
1	0	0	0				
1	0	0	1				

Binary coded decimal (**BCD)** is a system of writing numerals that assigns a four-digit binary code to each digit 0 through 9 in a decimal (base-10) numeral. The four-bit BCD code for any particular single base-10 digit is its representation in binary notation, as follows:

0 = 0000
1 = 0001
2 = 0010
3 = 0011
4 = 0100
5 = 0101
6 = 0110
7 = 0111
8 = 1000
9 = 1001

Numbers larger than 9, having two or more digits in the decimal system, are expressed digit by digit. For example, the BCD rendition of the base-10 number **1895** is

0001 1000 1001 0101

2421 Code

- a **weighted** code.
  - The weights assigned to the four digits are 2, 4, 2, and 1.

- The 2421 code is the same as that in BCD from 0 to 4; however, it differs from 5 to 9.
- For example, in this case the bit combination 0100 represents decimal 4; whereas the bit combination 1101 is interpreted as the decimal 7, as obtained from $2 \times 1 + 1 \times 4 + 0 \times 2 + 1 \times 1 = 7$.

- This is also a **self-complementary** code,
  - that is, the 9's complement of the decimal number is obtained by changing the 1s to 0s and 0s to 1s.

BCD-to-2421 Converter

- Truth Table

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

BCD-to-2421 Converter

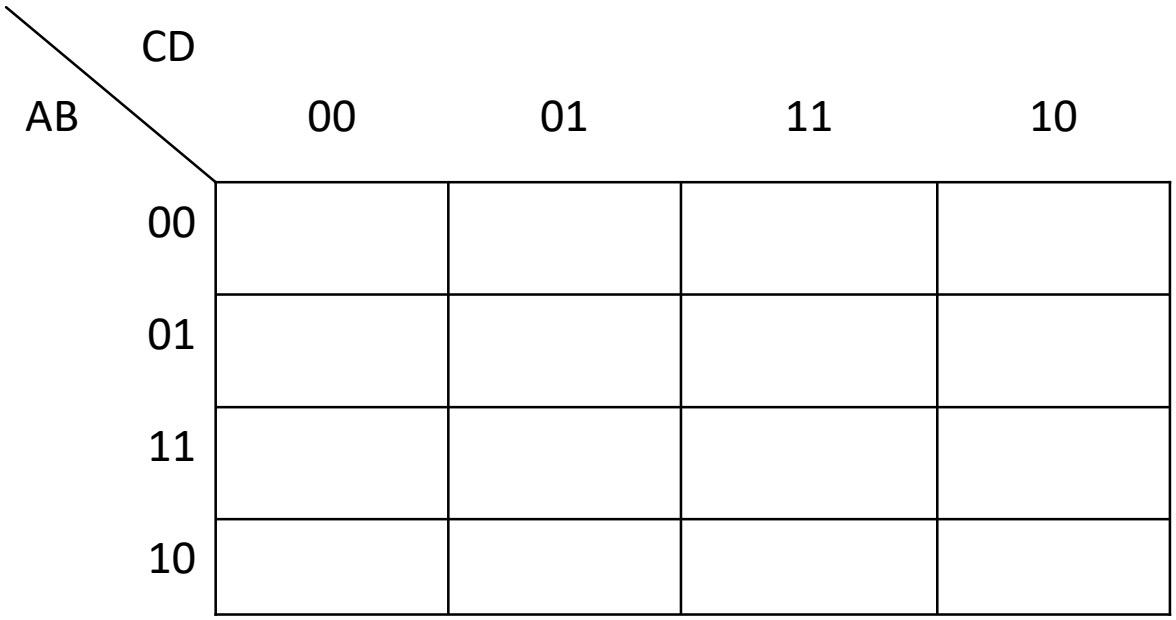
- Truth Table

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**:

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 =

BCD-to-2421 Converter

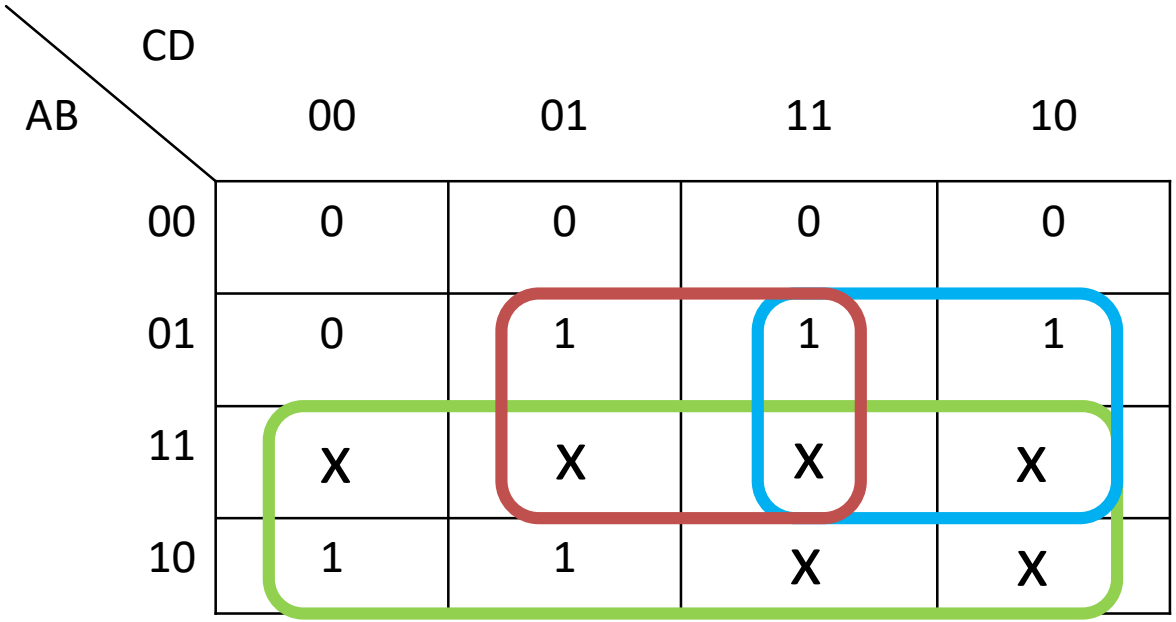
- Step 3: Obtain simplified Boolean expression for each output
- Output **x**:

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

BCD-to-2421 Converter

- Step 3: Obtain simplified Boolean expression for each output
- Output 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				
01				
11				
10				

- Output z:

AB \ CD	00	01	11	10
00				
01				
11				
10				

A1	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

y =

z =

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

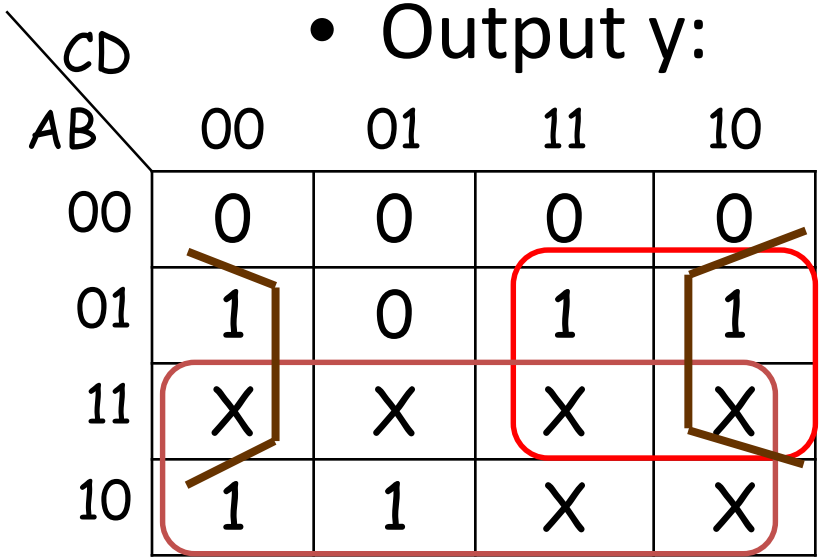
A	B	C	D	y	z
0	0	0	0	0	0
0	0	0	1	0	0
0	0	1	0	0	1
0	0	1	1	0	1
0	1	0	0	1	0
0	1	0	1	0	1
0	1	1	0	1	0
0	1	1	1	1	0
1	0	0	0	1	1
1	0	0	1	1	1
The rest				X	X

y =

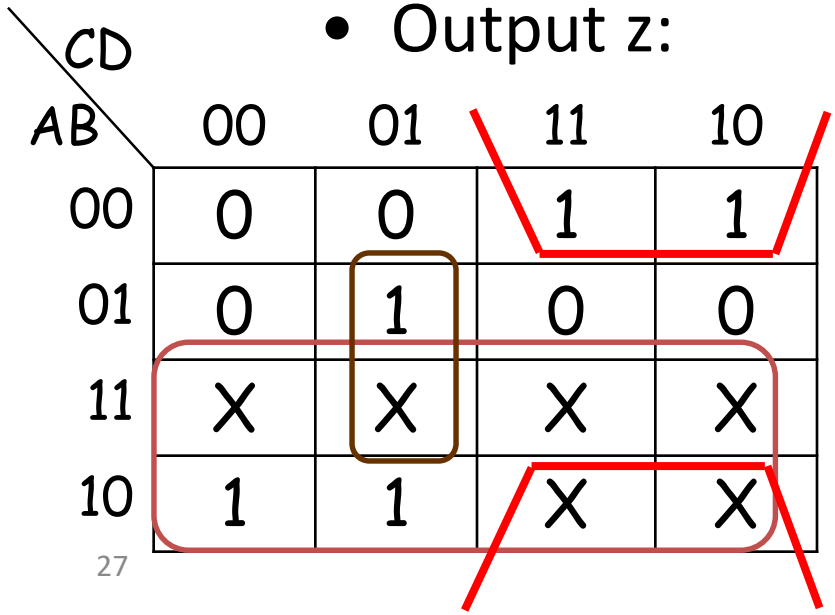
z =

Boolean Expressions for Outputs

- Output y:



- Output z:



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

$y = A + BD' + BC$

$z = A + B'C + BC'D$

Boolean Expressions for Outputs

- Output t:

CD / AB	00	01	11	10
00				
01				
11				
10				

t =

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

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 =

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

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

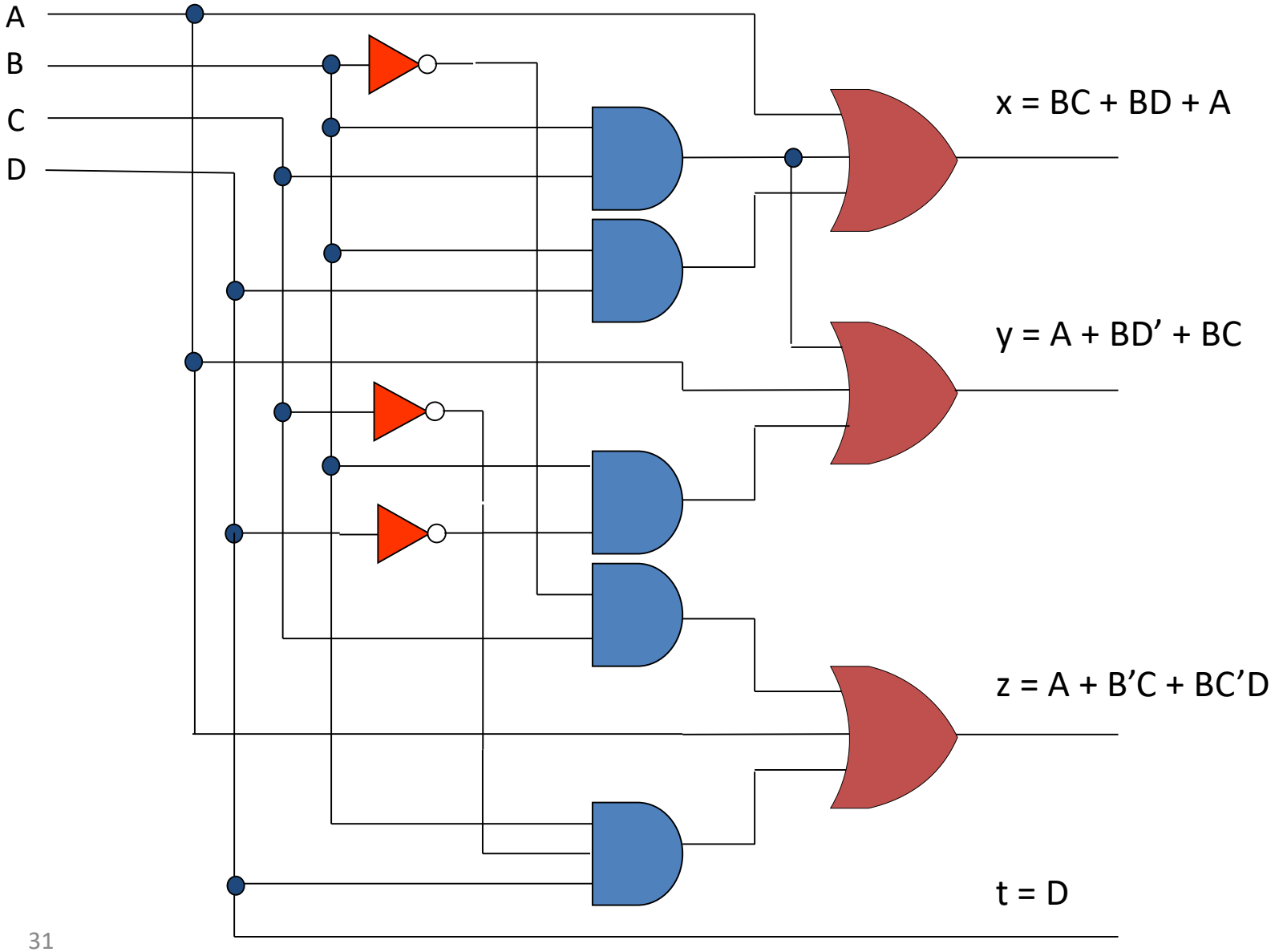
- Step 4: Draw the logic diagram

x = BC + BD + A

y = A + BD’ + BC

z = A + B’C + BC’D

Example: Logic Diagram



Example: Verification

- Step 5: Check the functional correctness of the logic circuit
- Apply all possible input combinations
- And check if the circuit generates the correct outputs for each input combinations
- 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

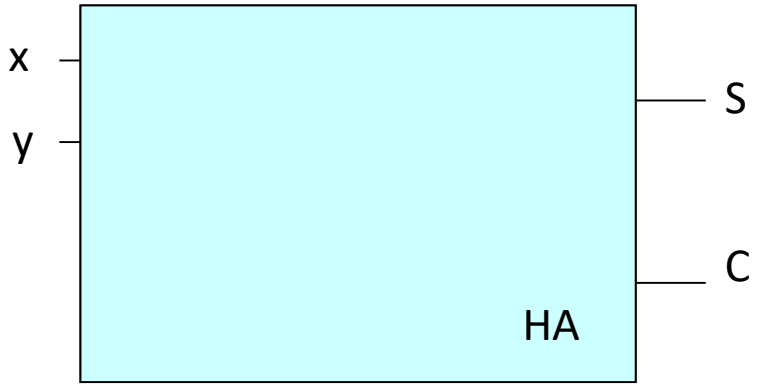
Half Adder

- Truth table

x	y	c	s
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

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

$C = xy$



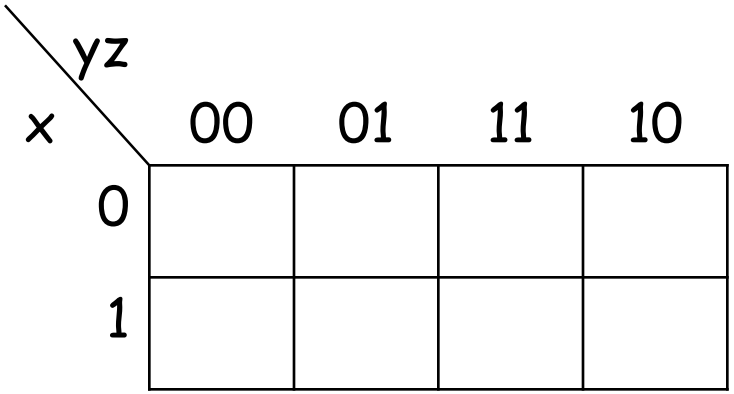
Full Adder 1/2

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

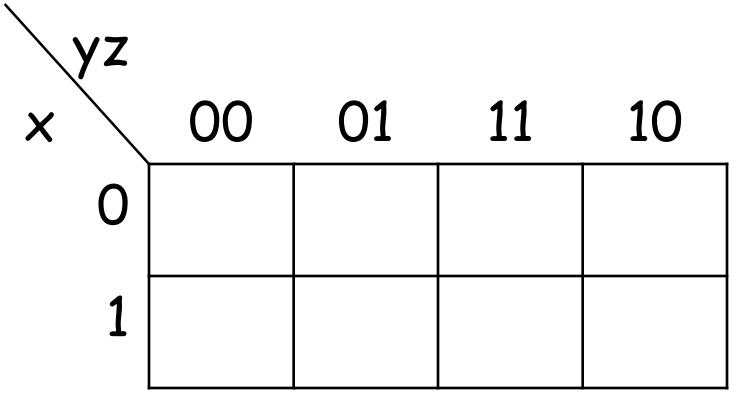
x	y	z	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Full Adder 2/2

- Karnaugh Maps



S = ?



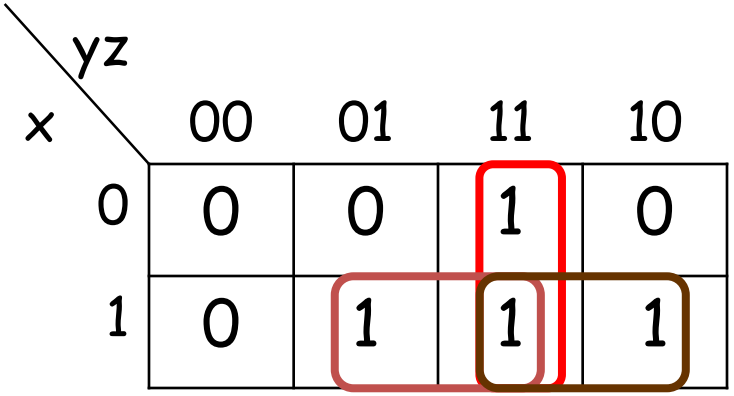
C = ?

Two level implementation
1st level: three AND gates
2nd level: One OR gate

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

C = xy + xz + yz

= xyz' + xyz + xy'z + x'yz

Two level implementation

1st level: three AND gates

2nd level: One OR gate

- S is an odd function of the 3 input bits

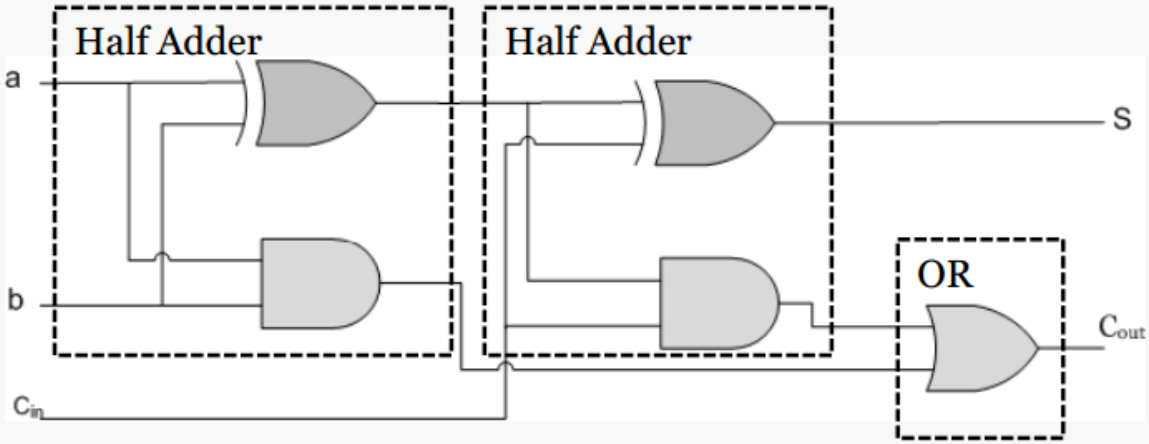
➔ $S = a \oplus b \oplus C_{in}$

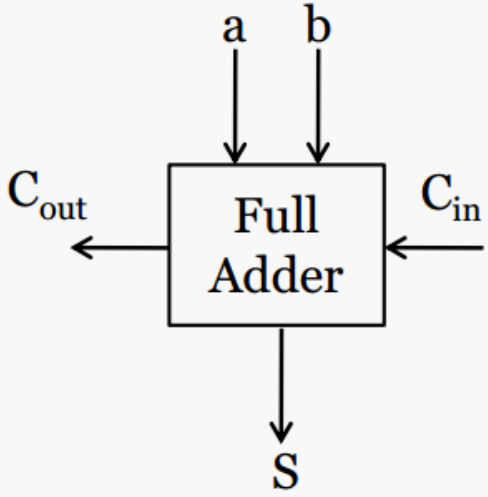
- $C_{out} = ab\,C_{in} + ab\,C_{in}' + ab'\,C_{in} + a'b\,C_{in}$

$= ab + C_{in}(ab' + a'b)$

$= ab + C_{in}(a \oplus b)$

a	b	C_{in}	C_{out}	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

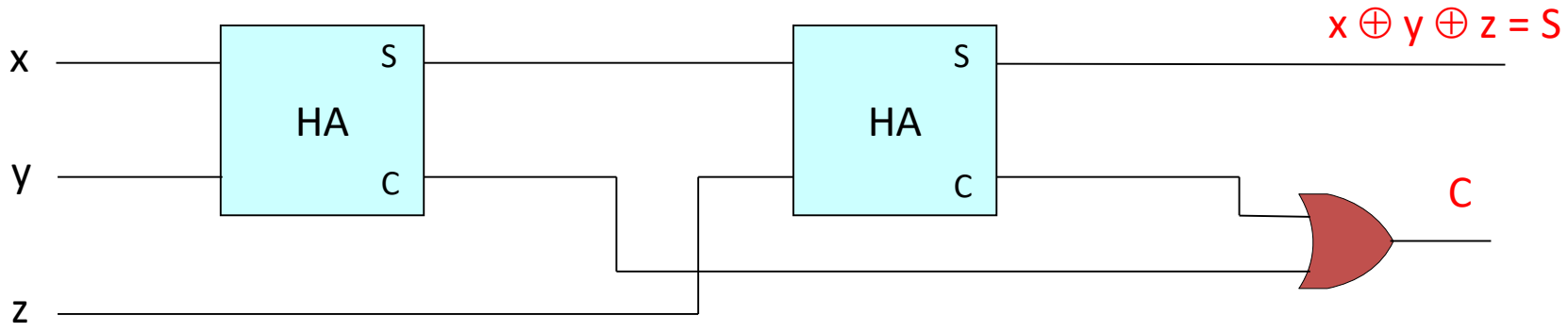




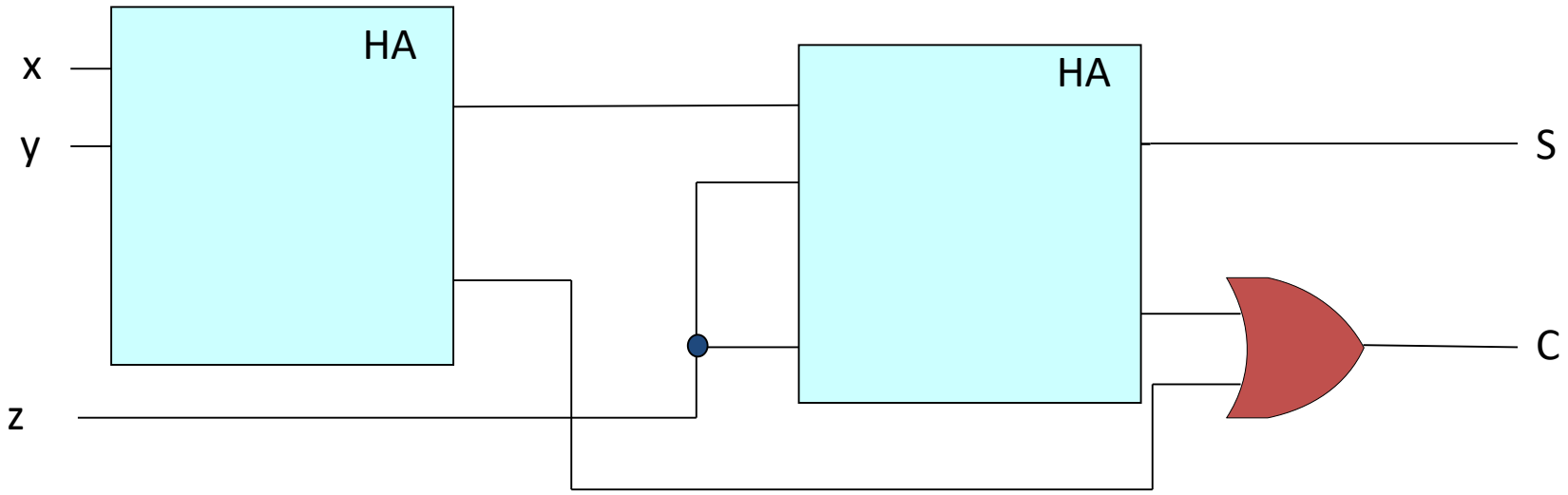
- Full Adder ⇔ 2 Half Adders and 1 OR gate

Full Adder: Hierarchical Realization

- Sum
  - $S = x \oplus y \oplus z = (x \oplus y) \oplus z$
- Carry
  - $C = xyz' + xyz + xy'z + x'yz$
    $= xy + (xy' + x'y)\, z$
    $= xy + (x \oplus y)\, z$
- This allows us to implement a full-adder using two half adders.

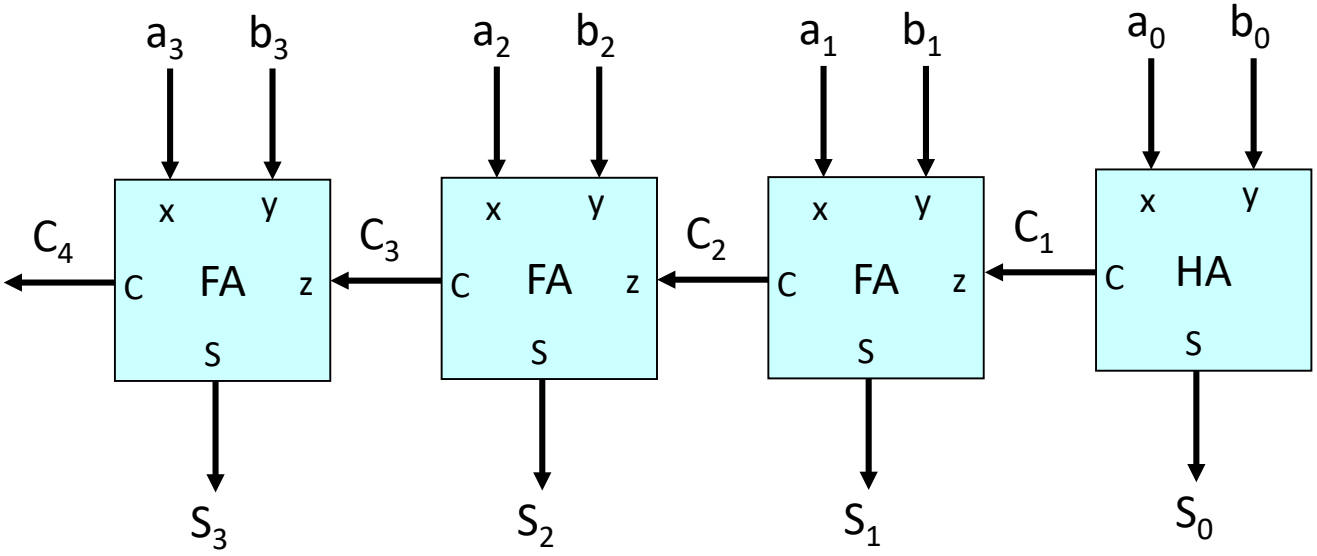


Full Adder Using Half Adders

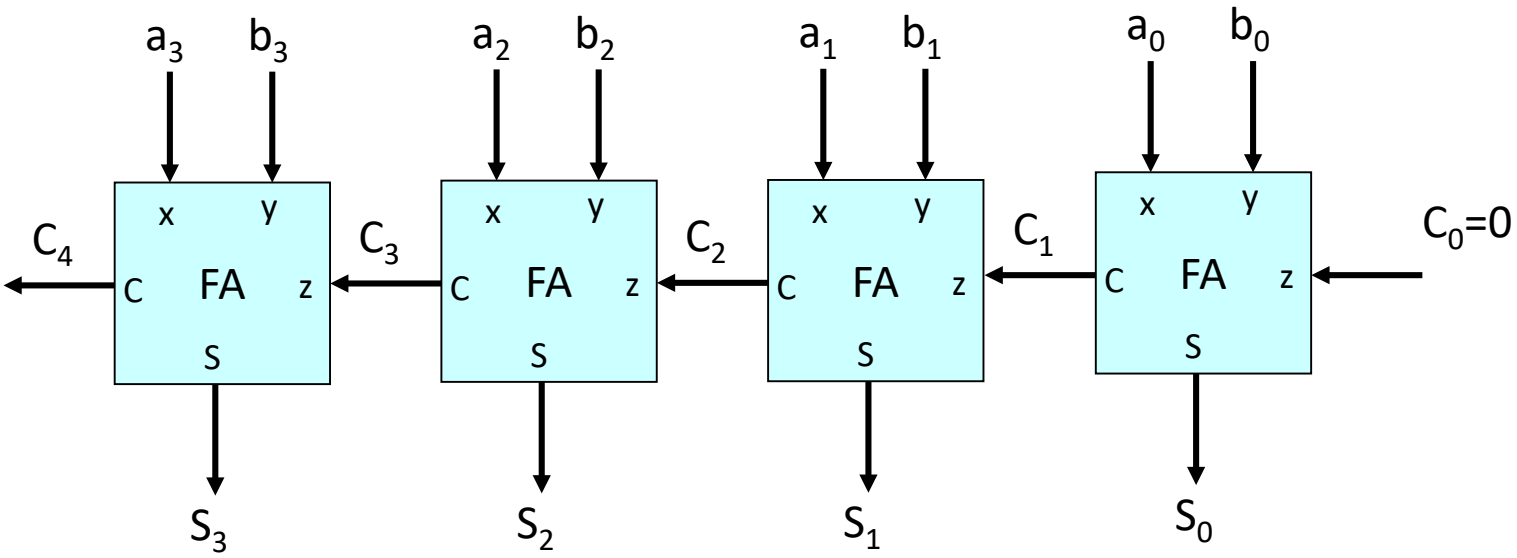


Integer Addition 1/2

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



Integer Addition 2/2

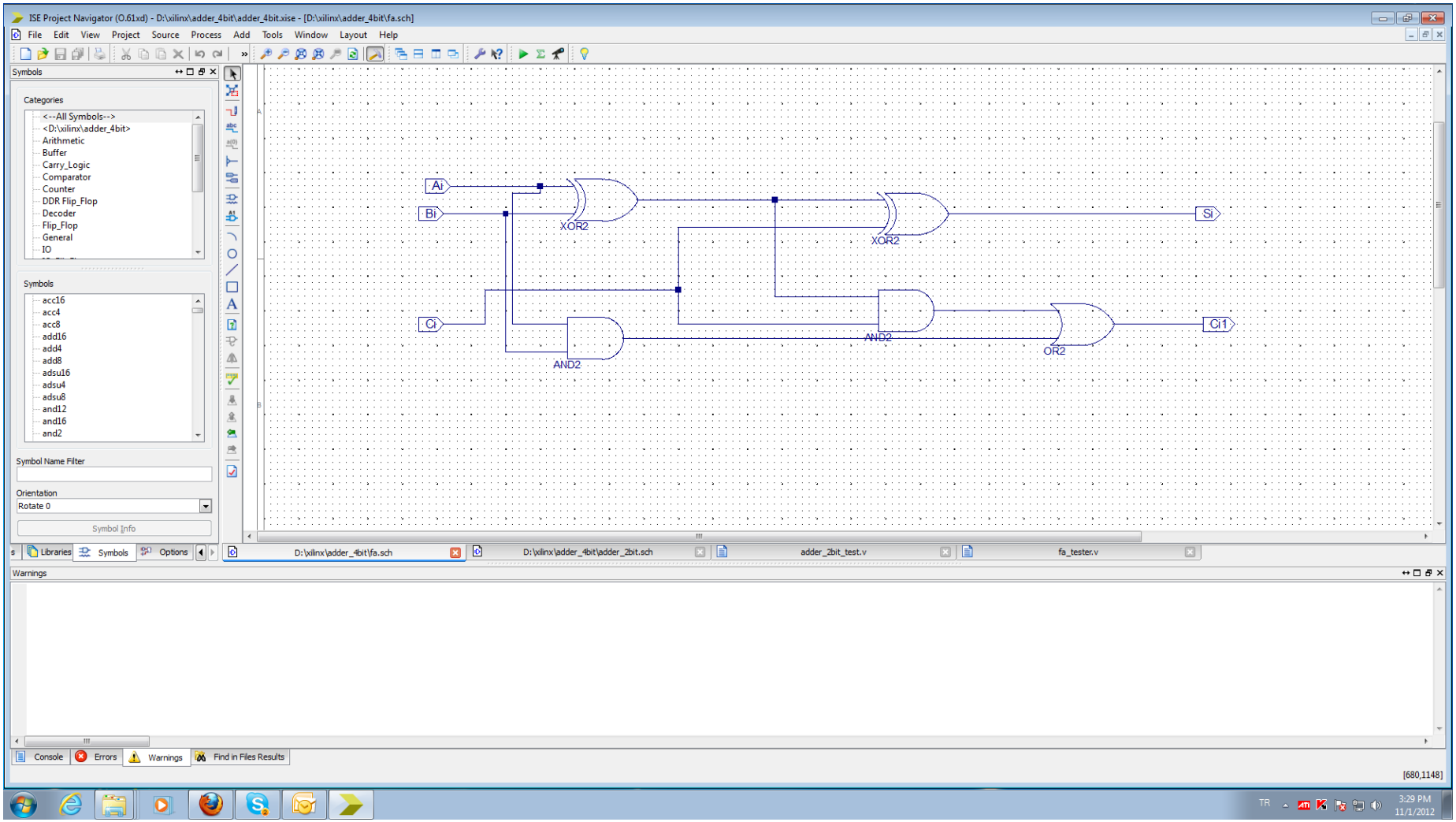


Ripple-carry adder

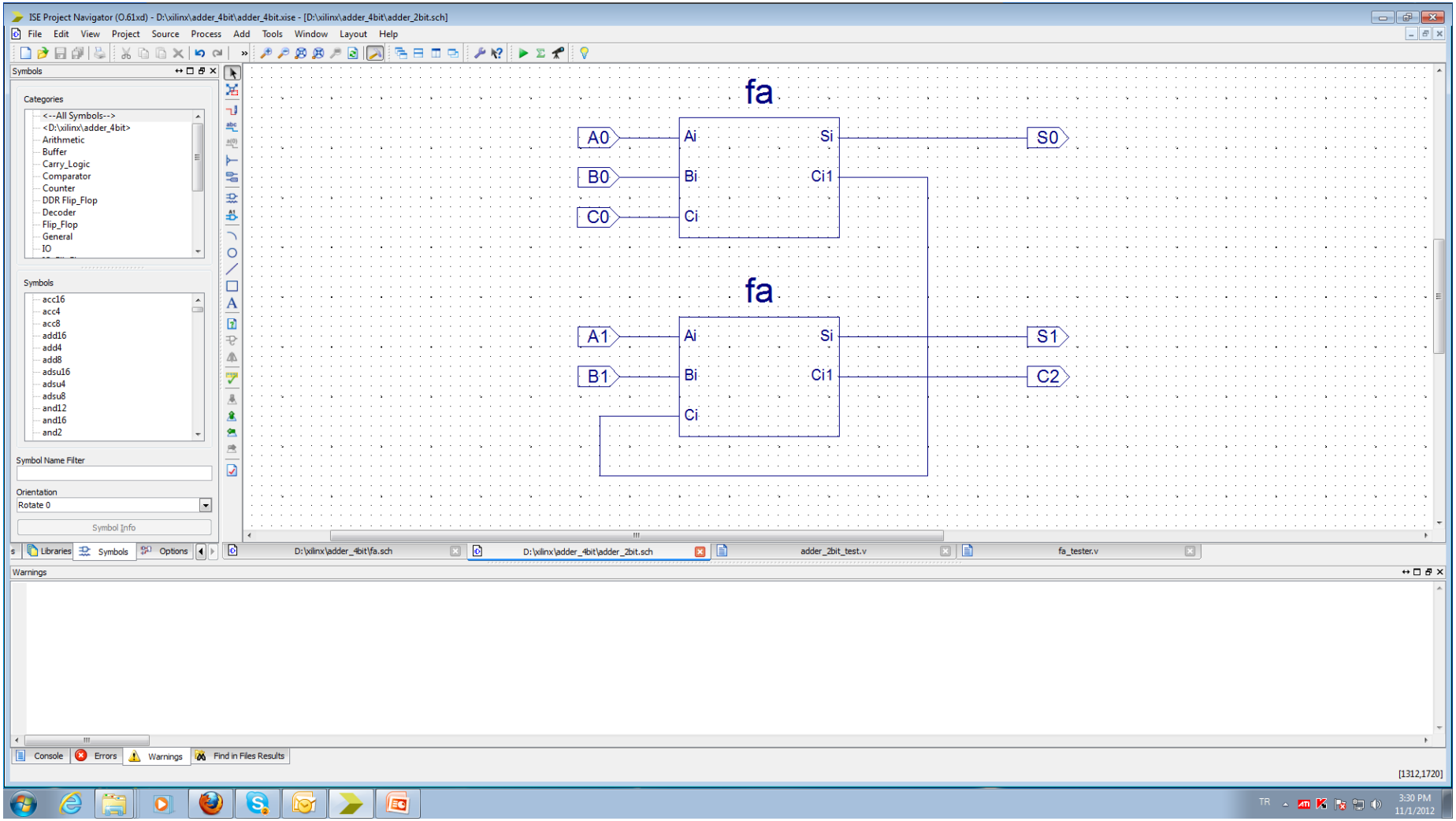
Hierarchical Design Methodology

- 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

Full Adder in Xilinx



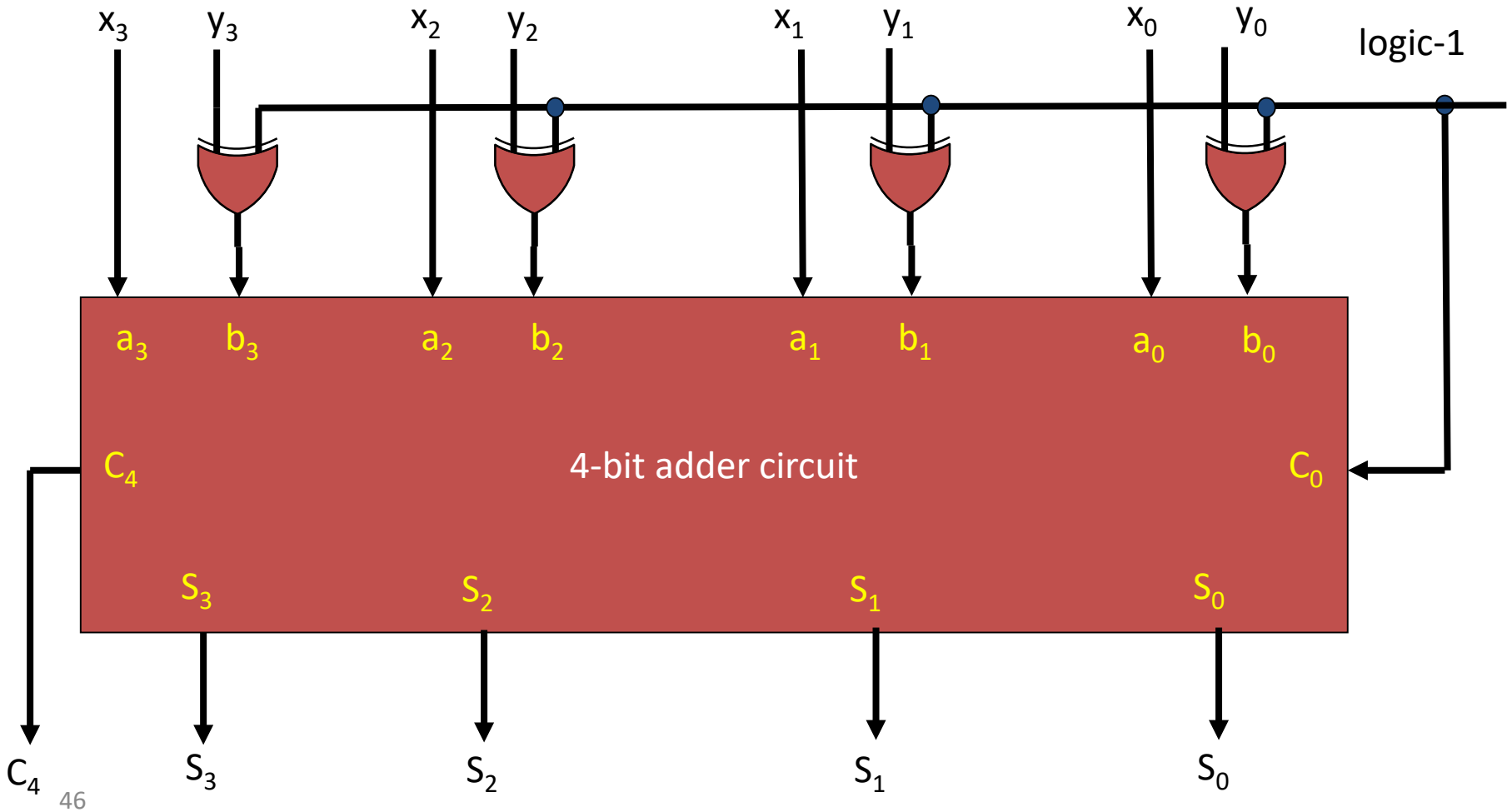
Two-bit Adder in Xilinx



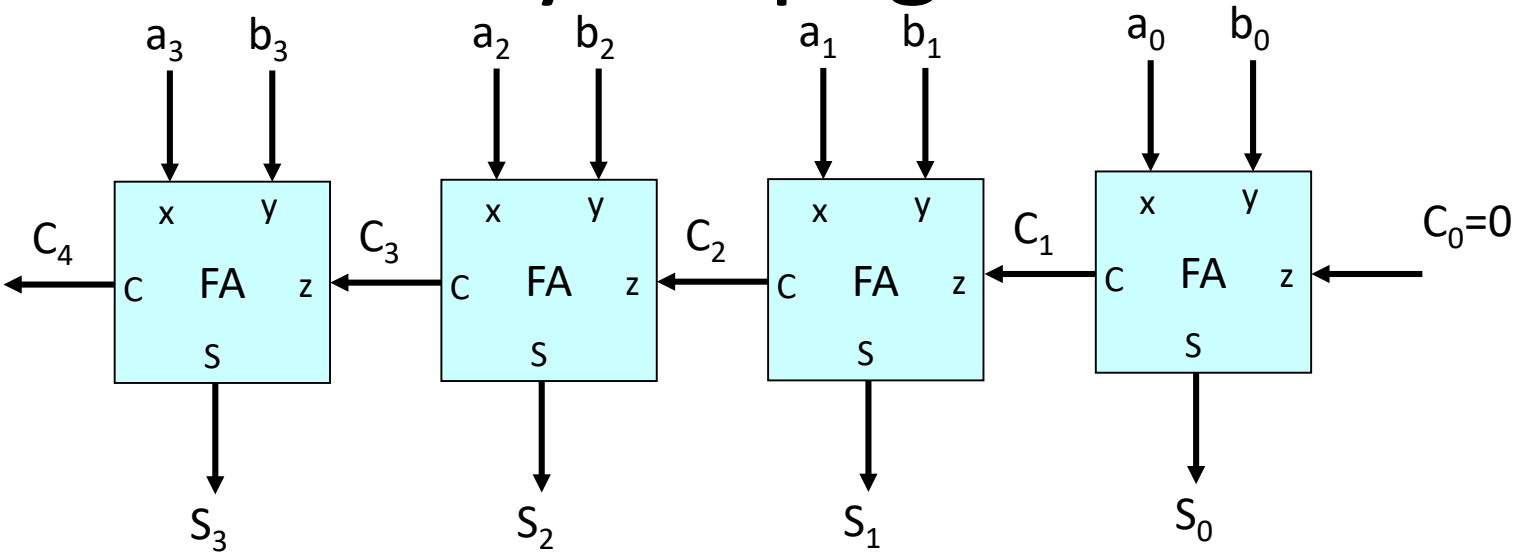
Subtractor

- Recall how we do subtraction (2's complement)

$$X - Y = X + (2^n - Y) = X + \sim Y + 1$$



Carry Propagation

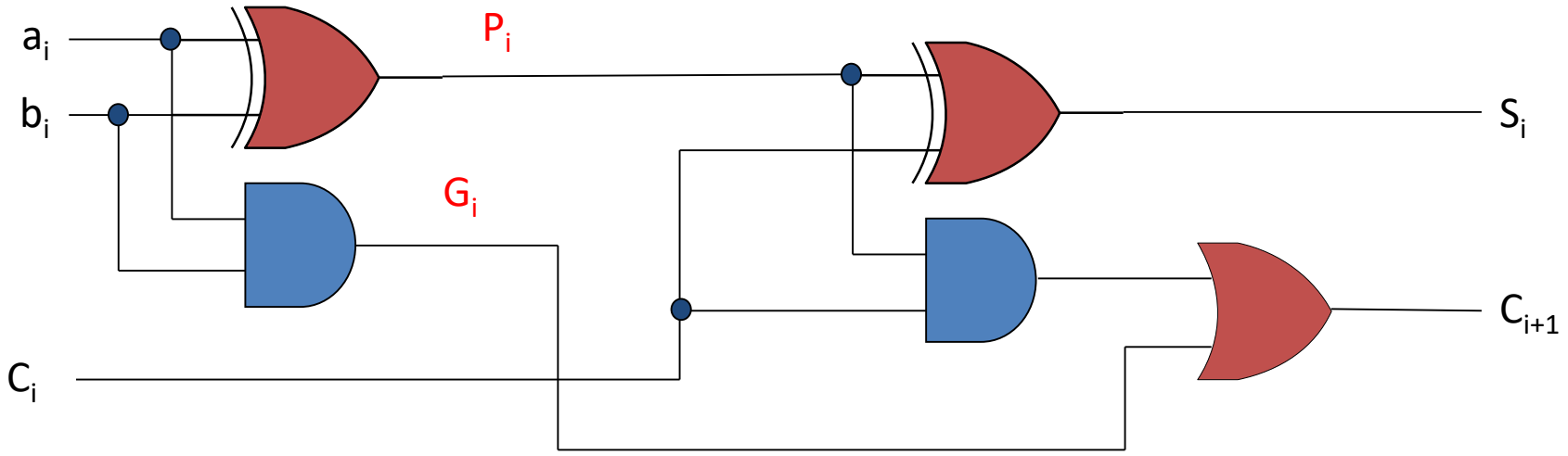


Ripple-carry adder

- 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: ??

Carry Propagation

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



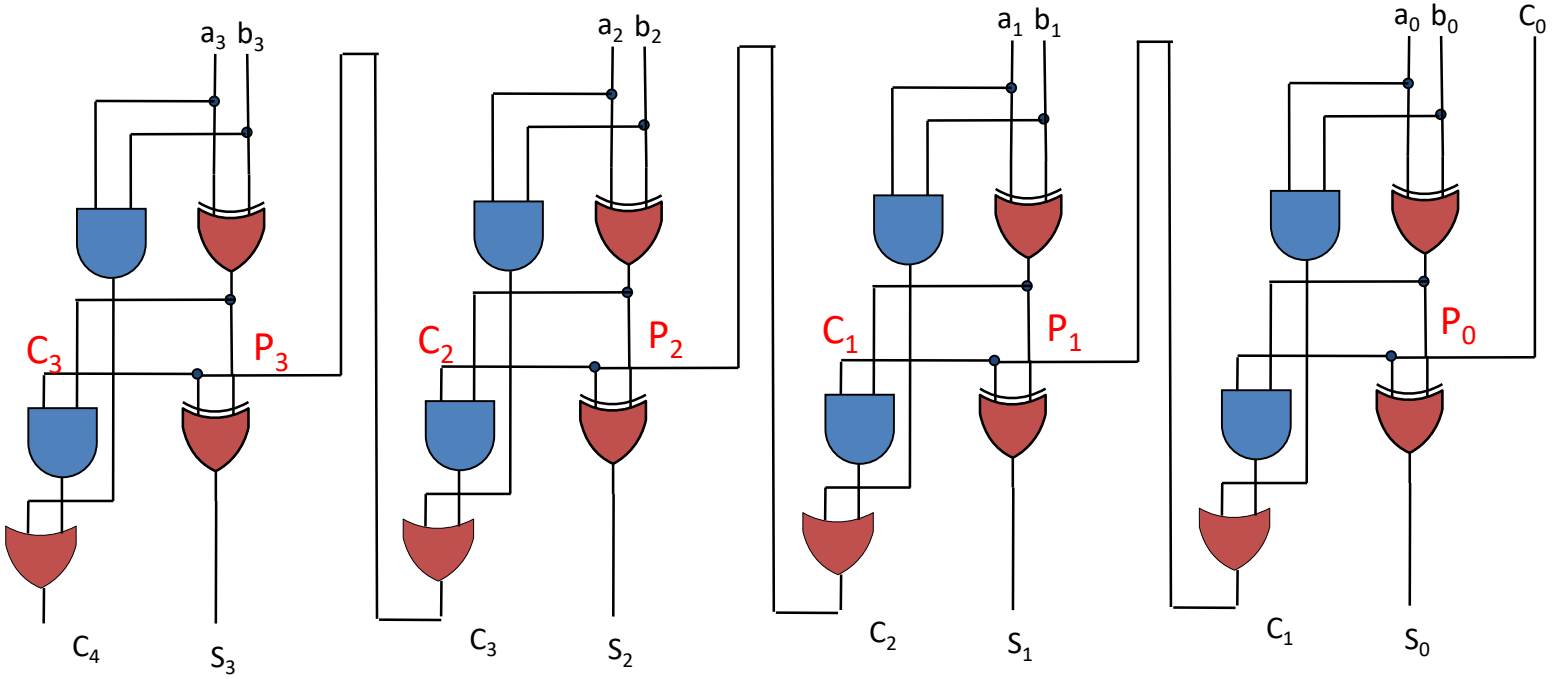
carry-**P**ropagate: $P_i = A_i \oplus B_i$

carry-**G**enerate: $G_i = A_i B_i$

$$S_i = P_i \oplus C_i$$

$$C_{i+1} = G_i + P_i C_i$$

Carry Propagation



- Carry-lookahead design

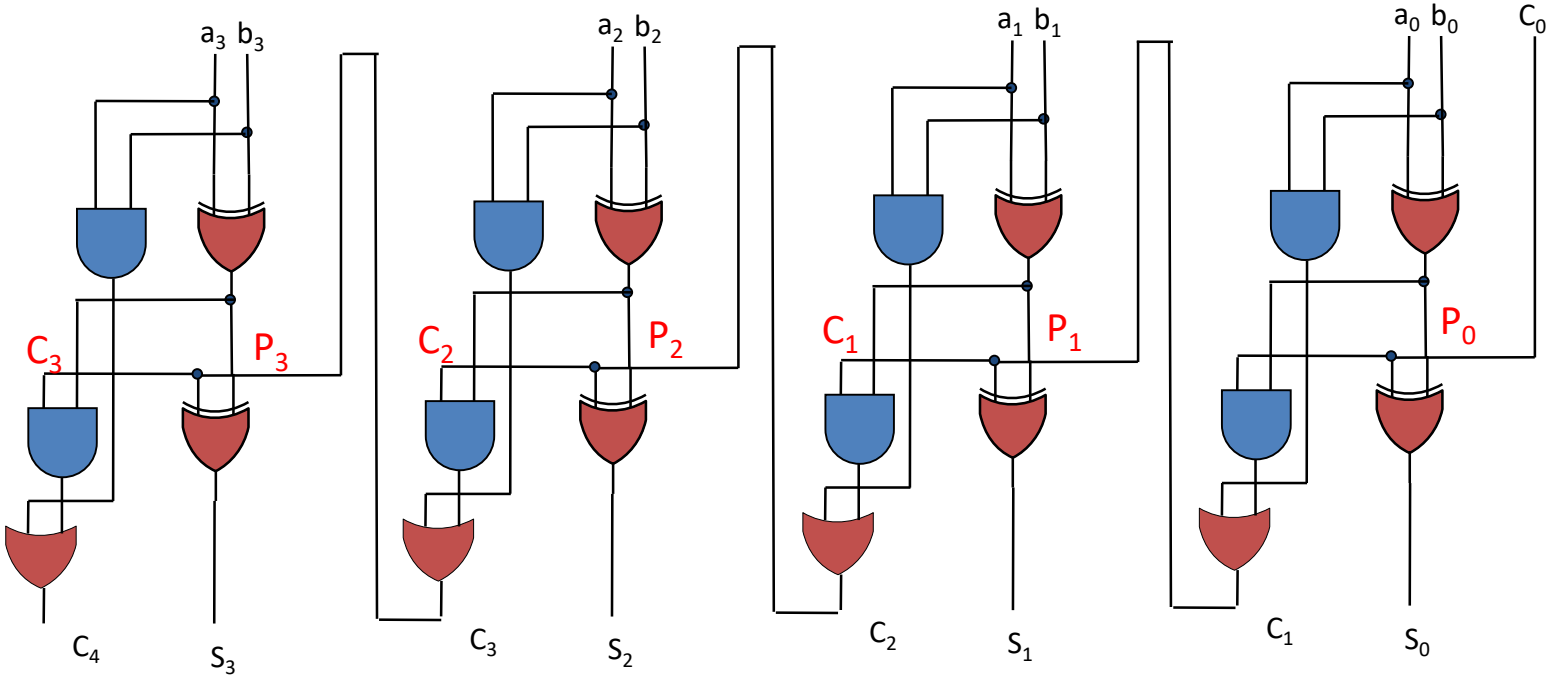
$C_0 = \text{input carry}$

$C_1 = G_0 + P_0C_0$

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

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

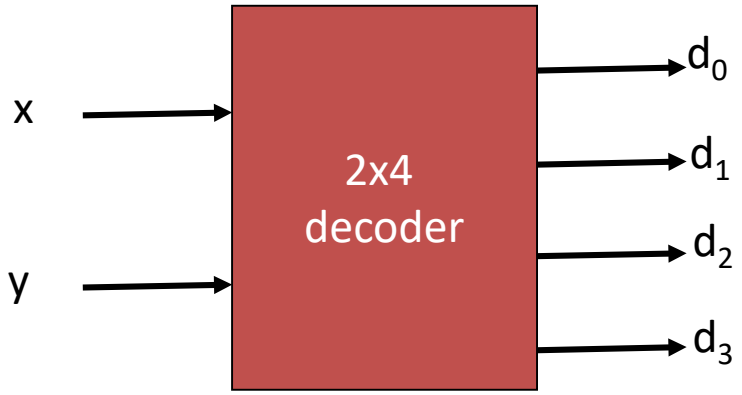
Carry Propagation



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

Decoders

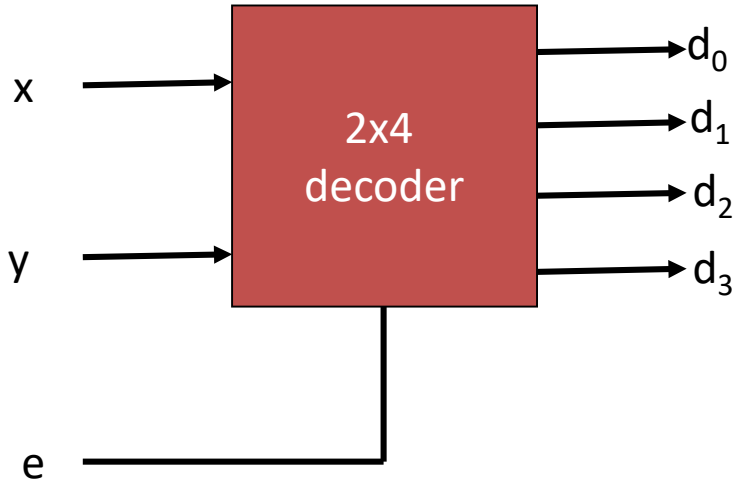
- A binary code of **n** bits
  - capable of representing $\mathbf{2^n}$ distinct elements of coded information
  - A decoder is a combinational circuit that converts binary information from n binary inputs to a maximum of 2^n unique output lines



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 =
- d_1 =
- d_2 =
- d_3 =

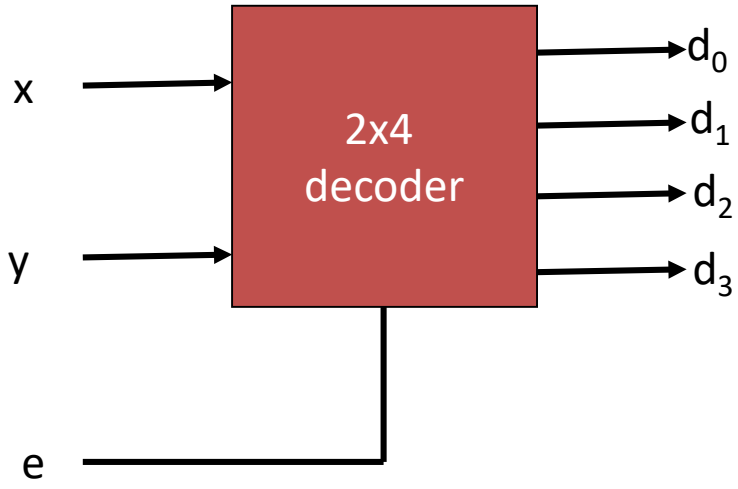
Decoder with Enable Input

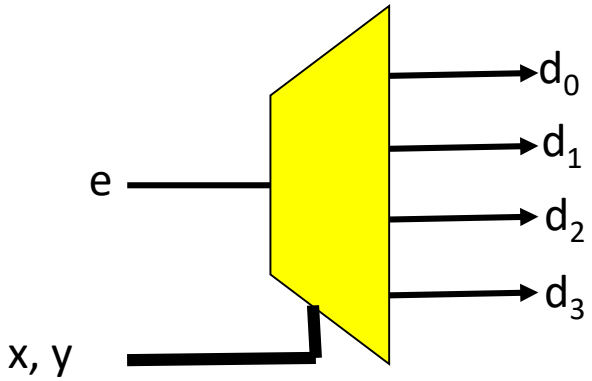


e	x	y	d_0	d_1	d_2	d_3
0	X	X	0	0	0	0
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1

Demultiplexer

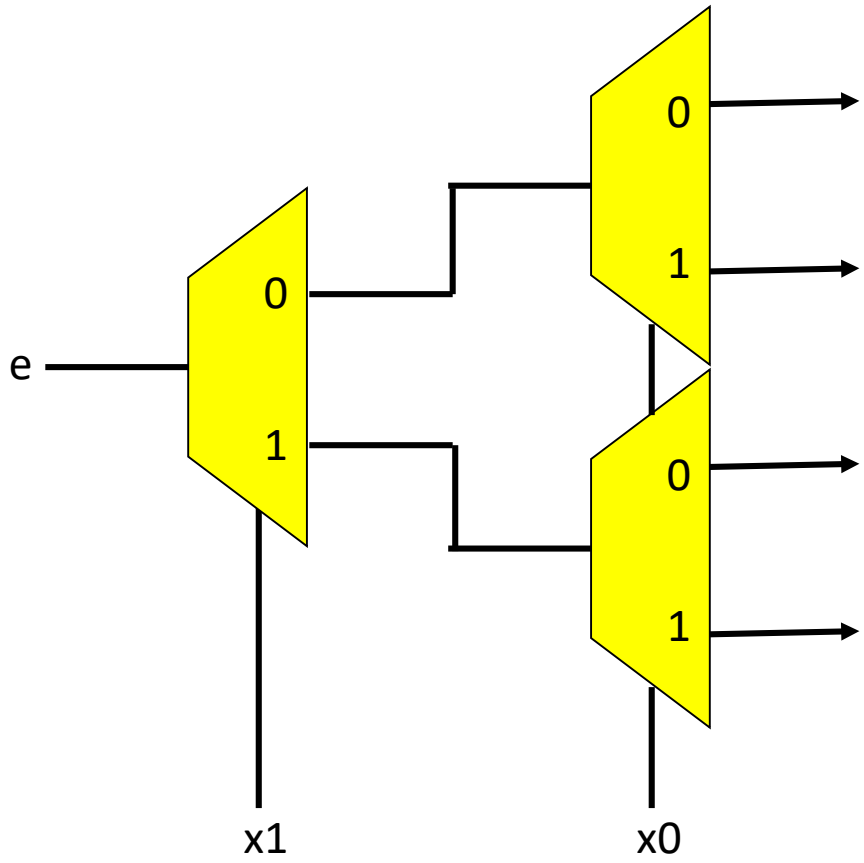
- A demultiplexer is a combinational circuit
  - it receives information from a single input line and directs it one of 2^n output lines
  - It has n selection lines as to which output will get the input



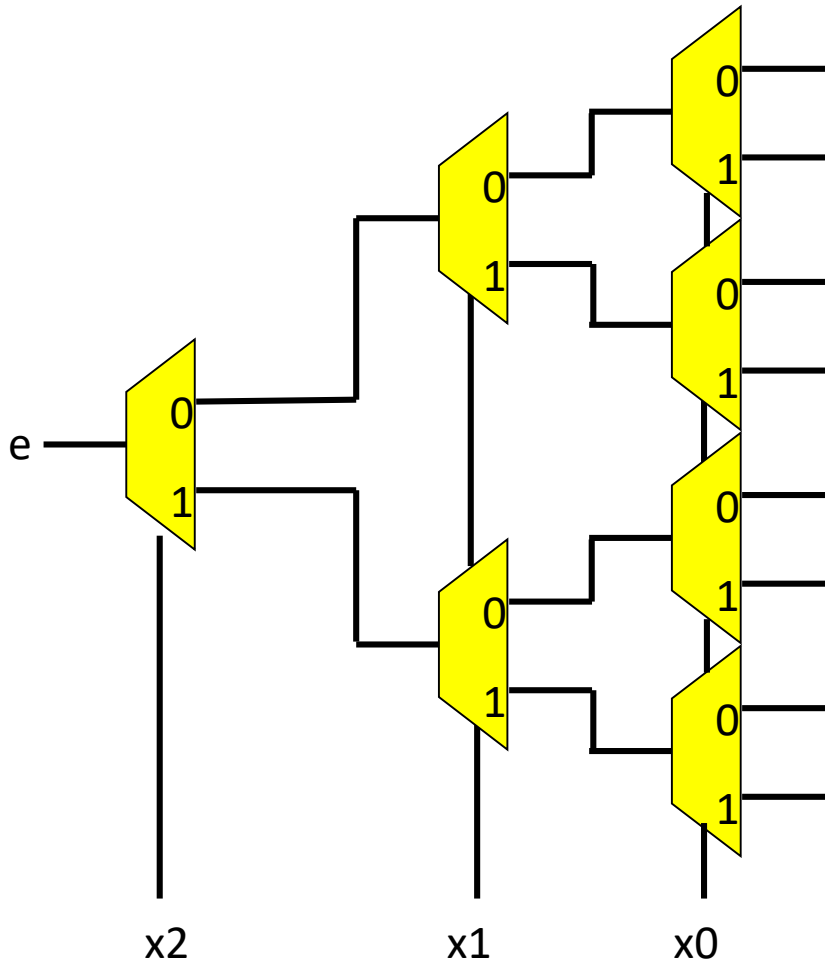


d_0 = e when x = 0 and y = 0
d_1 = e when x = 0 and y = 1
d_2 = e when x = 1 and y = 0
d_3 = e when x = 1 and y = 1

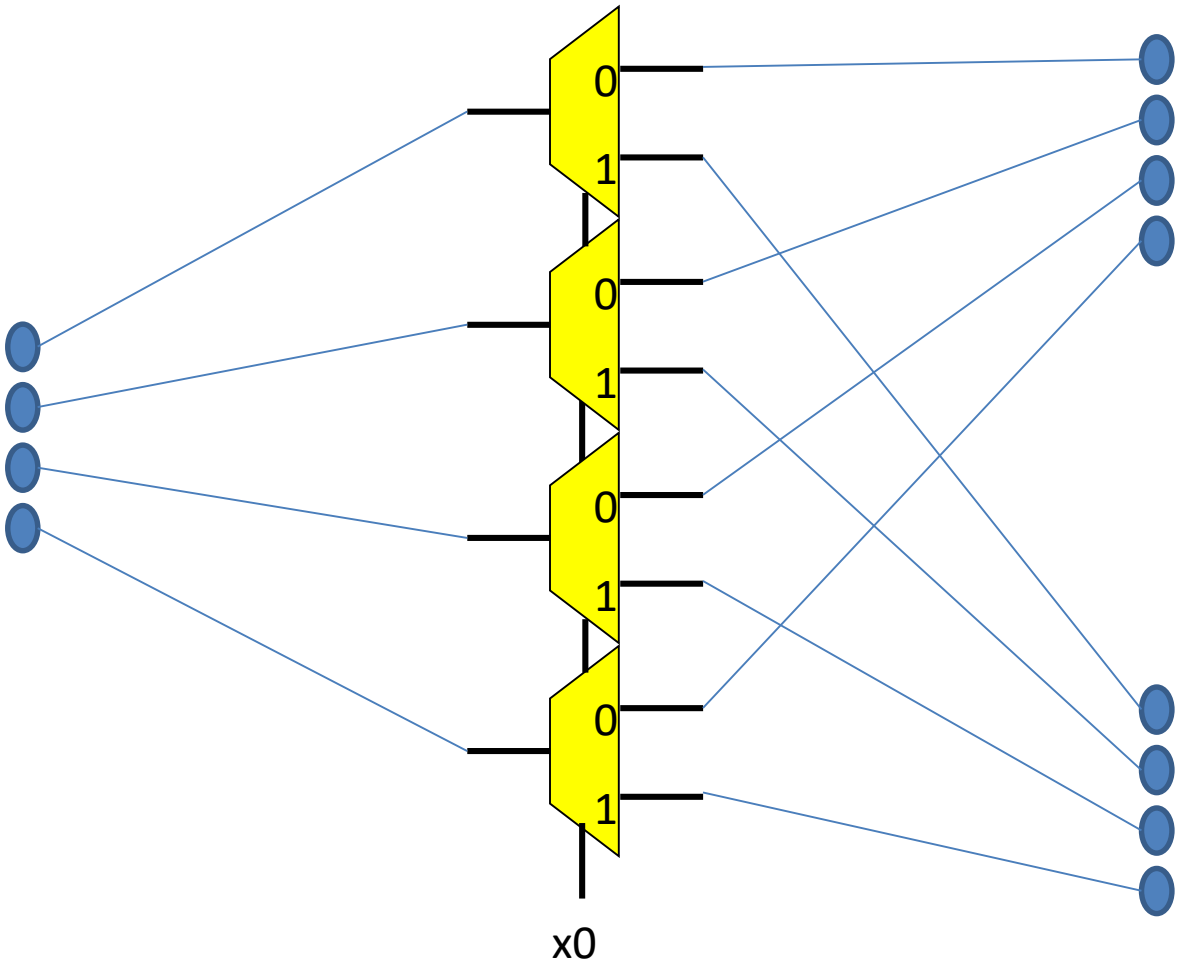
Demultiplexer



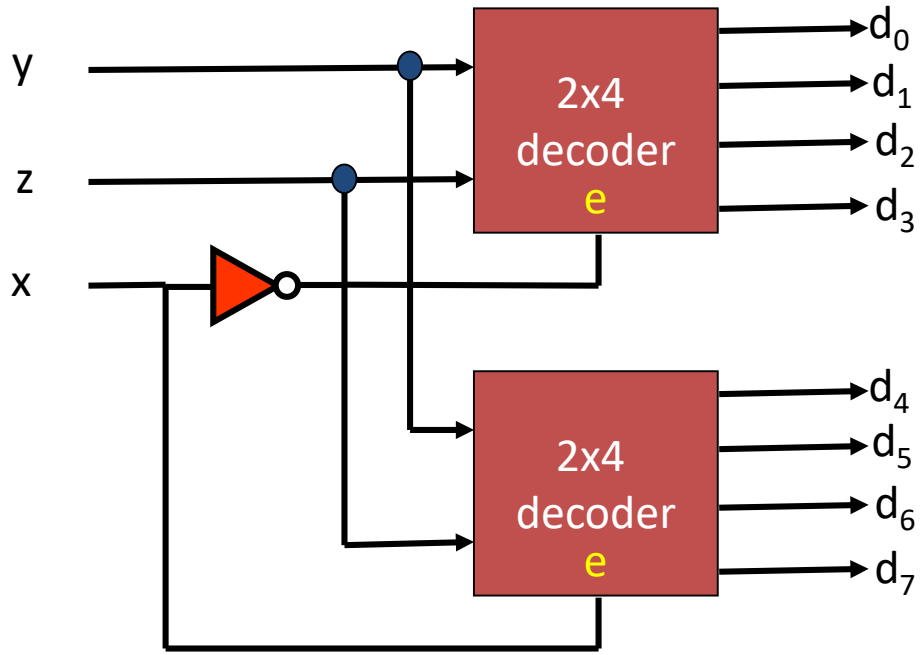
Demultiplexer



Demultiplexer



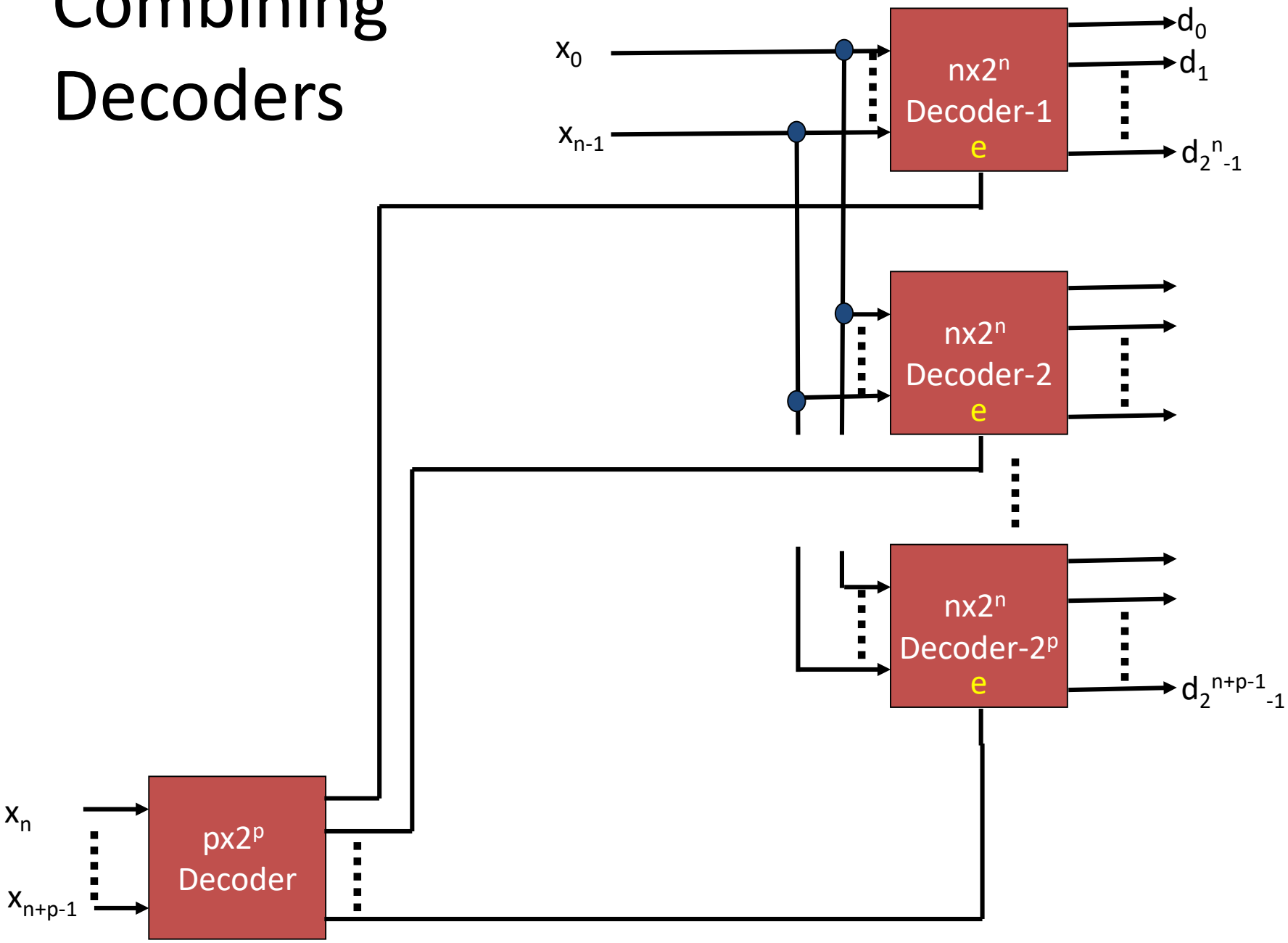
Combining Decoders



x	y	z	active output
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

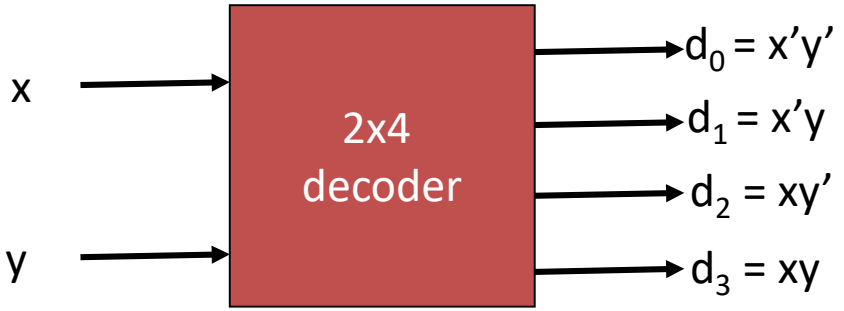
- Decoders with enable inputs can be connected together to form a larger decoder circuit.
- Figure shows two 2-to-4-line decoders with enable inputs connected to form a 3-to-8-line decoder.
- When *x*=0, the top decoder is enabled and the other is disabled. The bottom decoder outputs are all 0's, and the top four outputs generate minterms 000 to 011.
- When *x*=1, the enable conditions are reversed: The bottom decoder outputs generate minterms 100 to 111, while the outputs of the top decoder are all 0's.
- This example demonstrates the usefulness of enable inputs in decoders

Combining Decoders



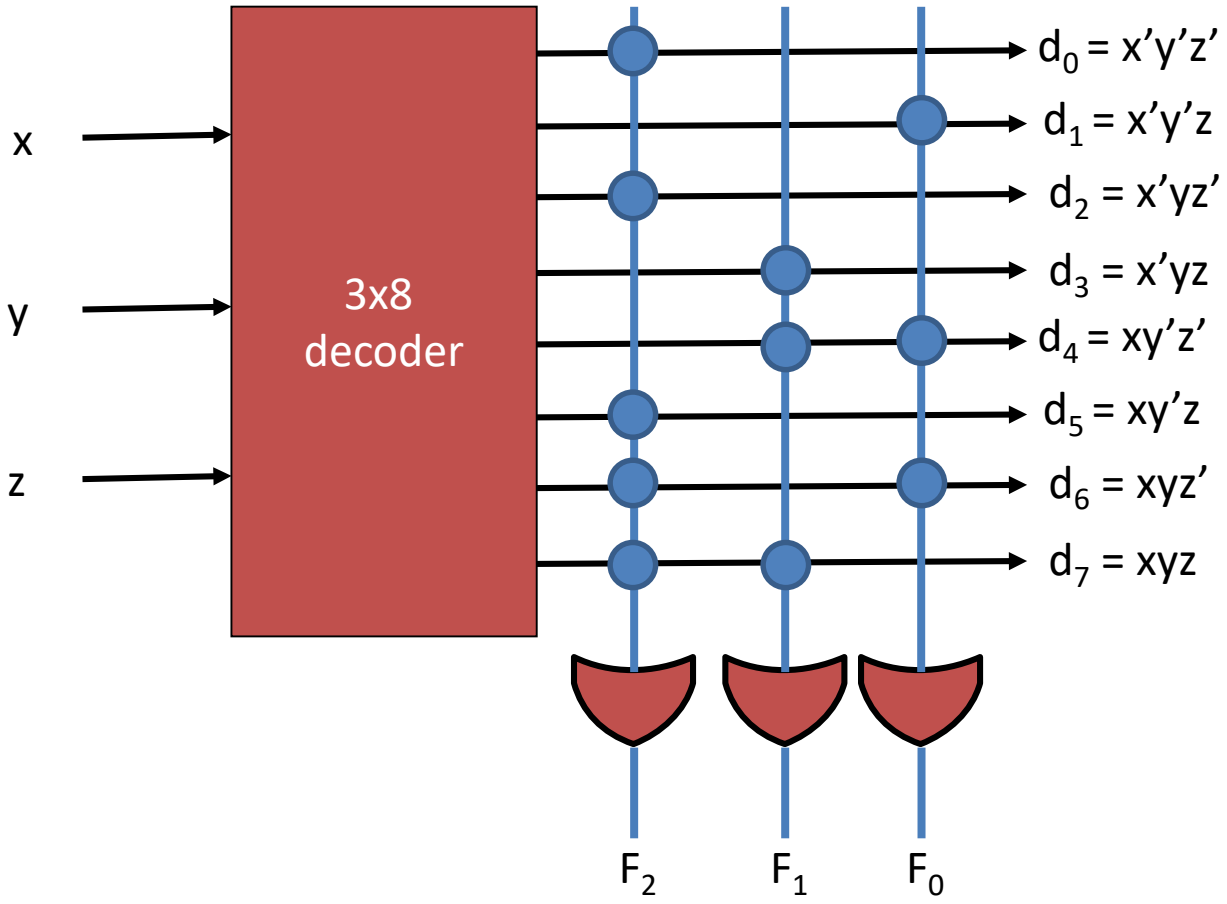
Decoder as a Building Block

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

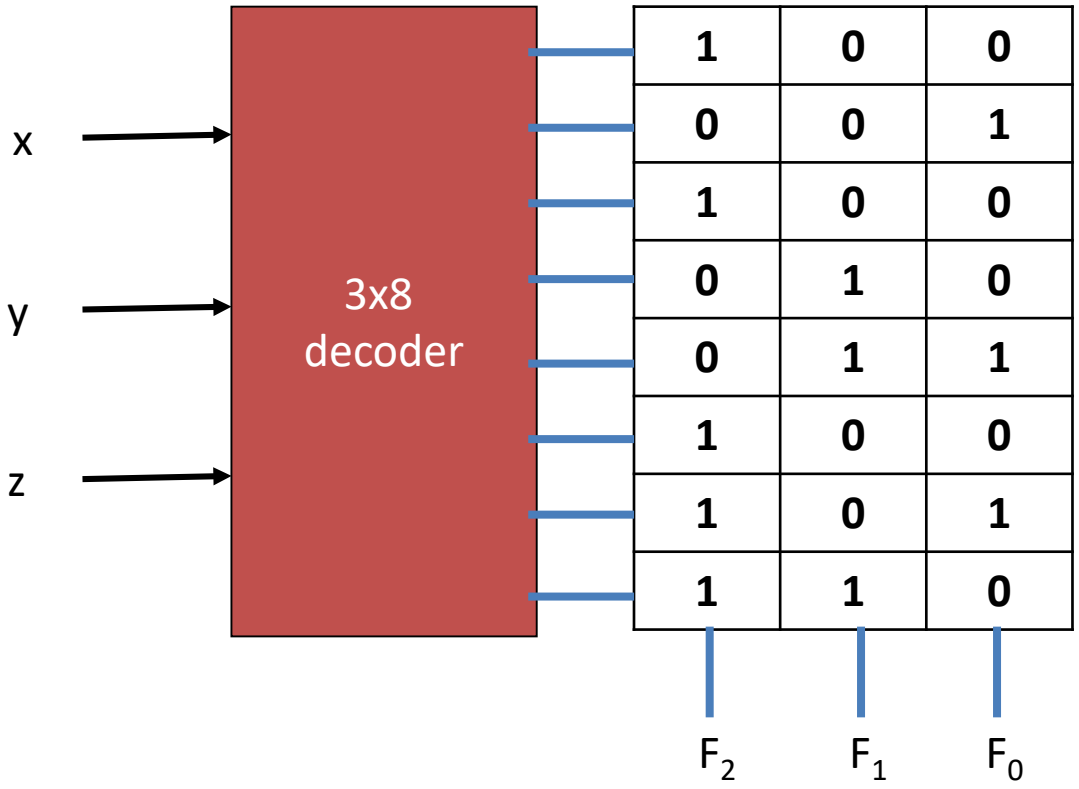


- We can use a decoder and OR gates to realize any Boolean function expressed as sum of minterms
  - Any combinational circuit with **n inputs** and **m outputs** can be realized using
    - an **n-to-2^n line decoder**
    - and **m OR gates**.

Decoder as a Building Block-ROM

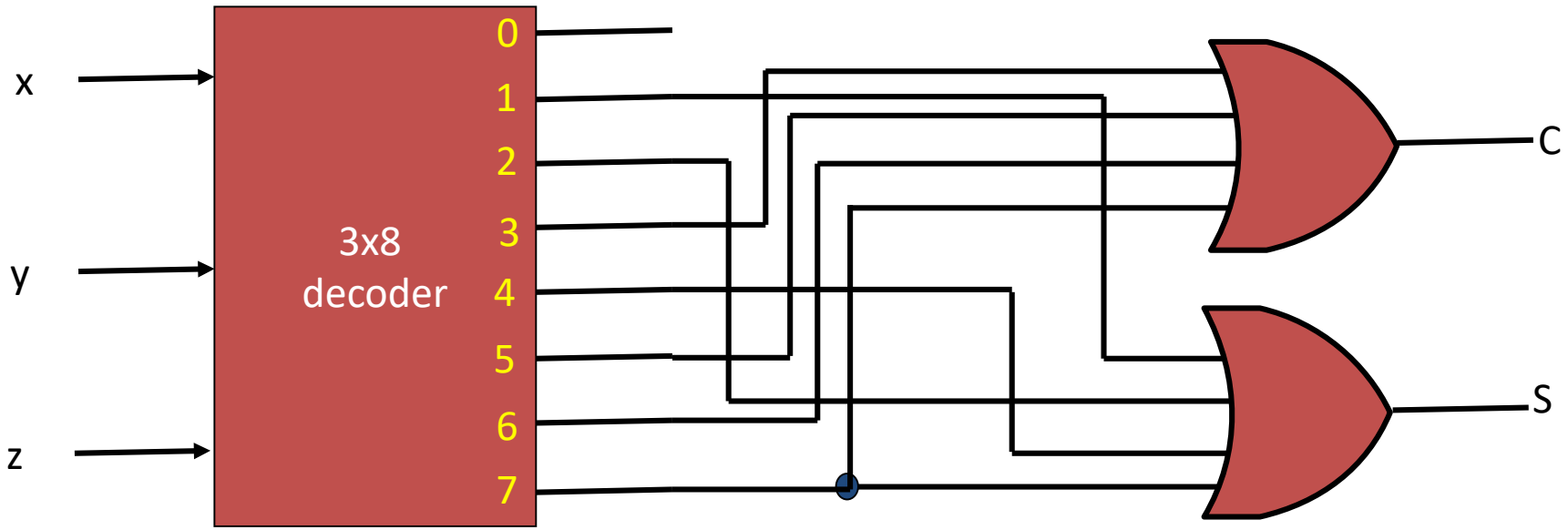


Decoder as a Building Block-ROM



Example: Decoder as a Building Block

- Full adder
  - C = xy + xz + yz
  - S = x ⊕ y ⊕ z



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	a	b	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

- **if two or more inputs are equal to 1 at the same time, the input having the highest priority will take precedence.**
- In addition to the two outputs *a* and *b* , the circuit has a third output designated by *V* ; this is a *valid* bit indicator that is set to 1 when one or more inputs are equal to 1. If all inputs are 0, there is no valid input and *V* is equal to 0. The other two outputs are not inspected when V equals 0 and are specified as don't-care conditions.

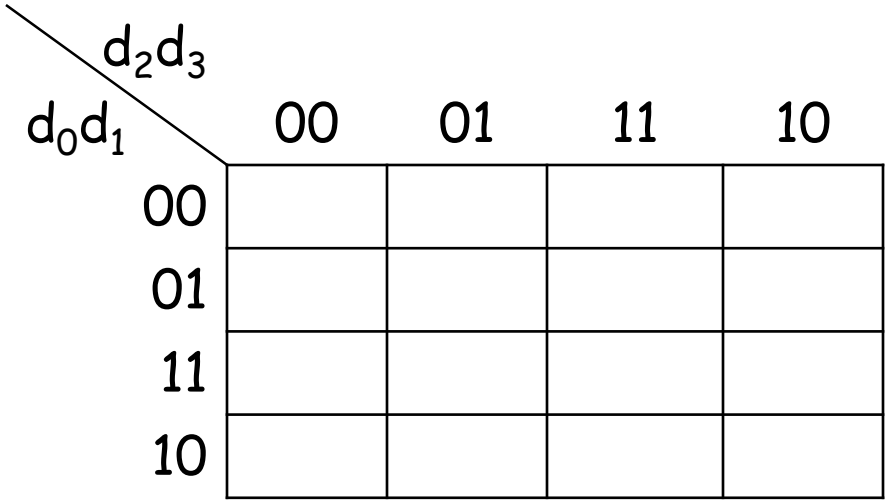
- Priority encoder:

d_0	d_1	d_2	d_3	a	b	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

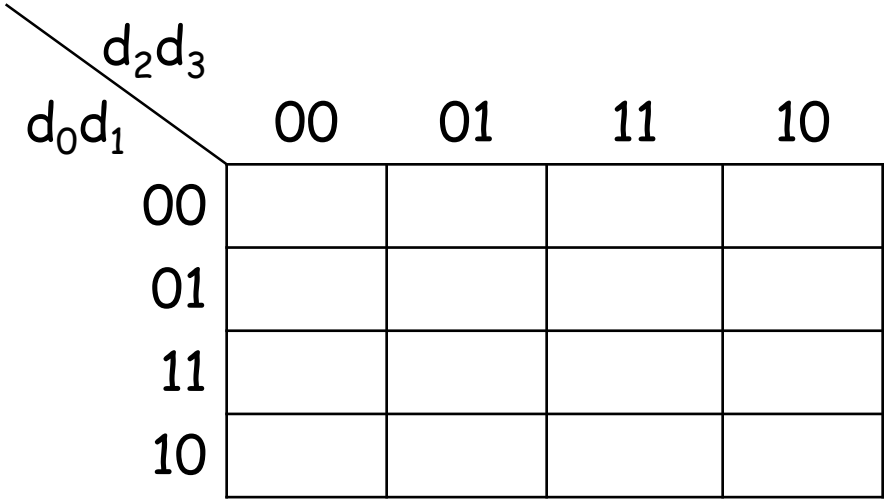
4-bit Priority Encoder

- 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 = ?
  - The condition for output *V* is an OR function of all the input variables.

Maps for 4-bit Priority Encoder



a =



b =

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

a =

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

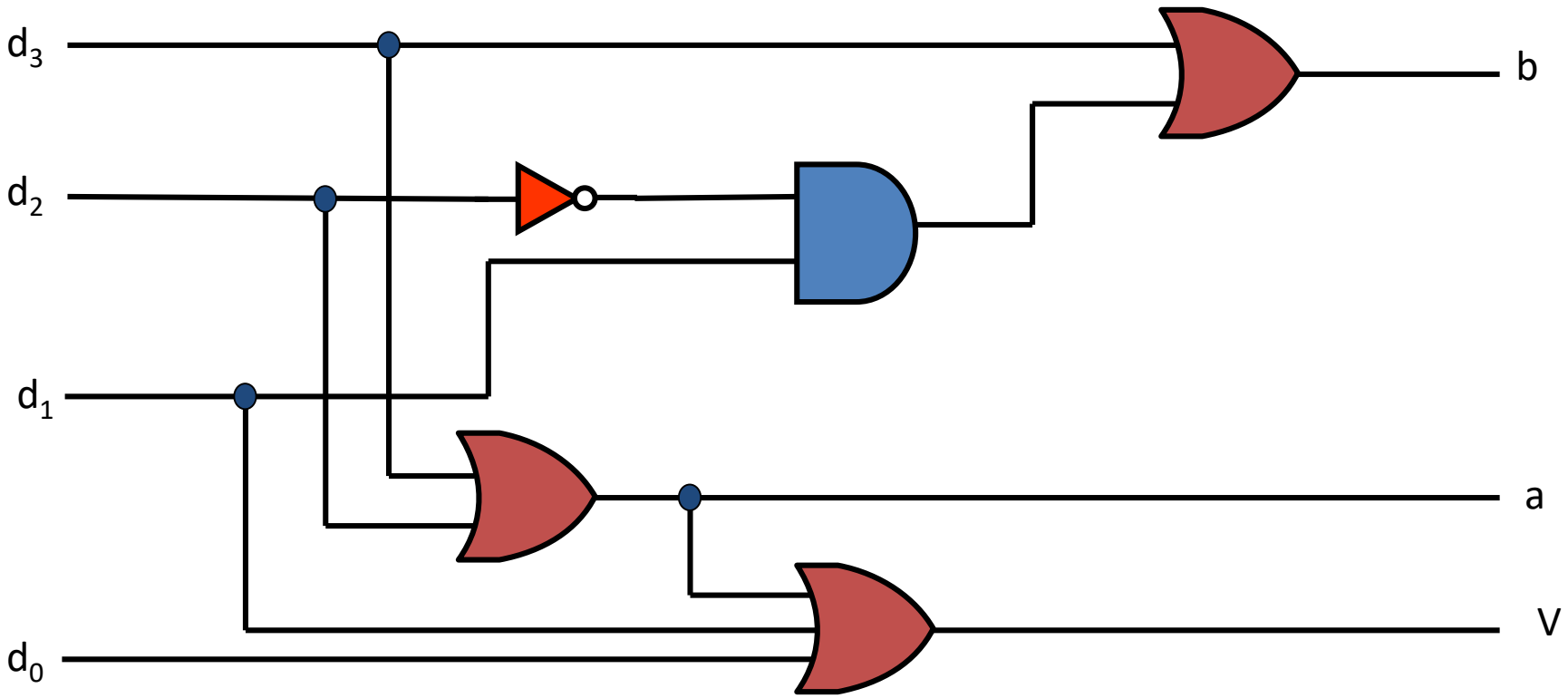
b =

4-bit Priority Encoder: Circuit

$a = d_2 + d_3$

$b = d_1 d_2' + d_3$

$V = d_0 + d_1 + d_2 + d_3$



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

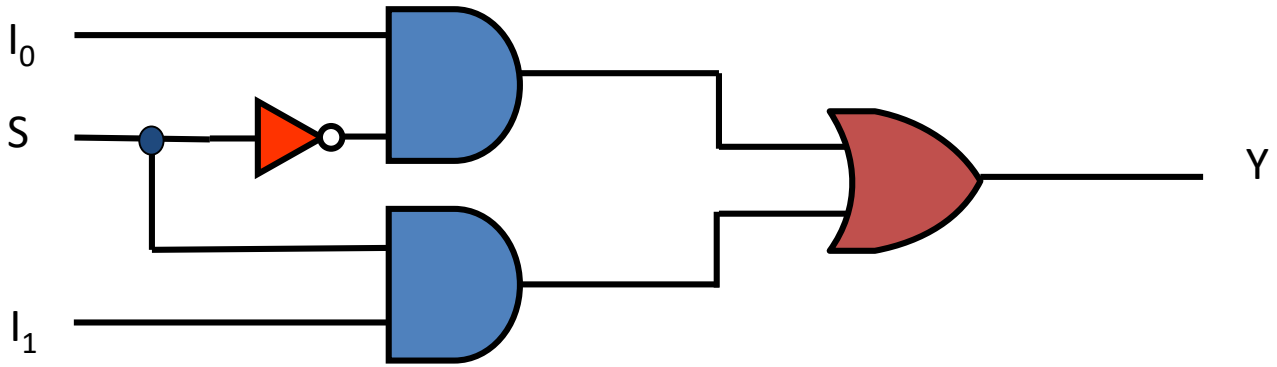
Y = ?

S	Y
0	I_0
1	I_1

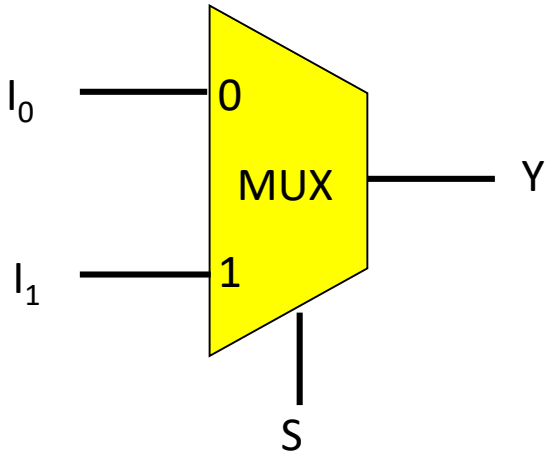
Function Table

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.

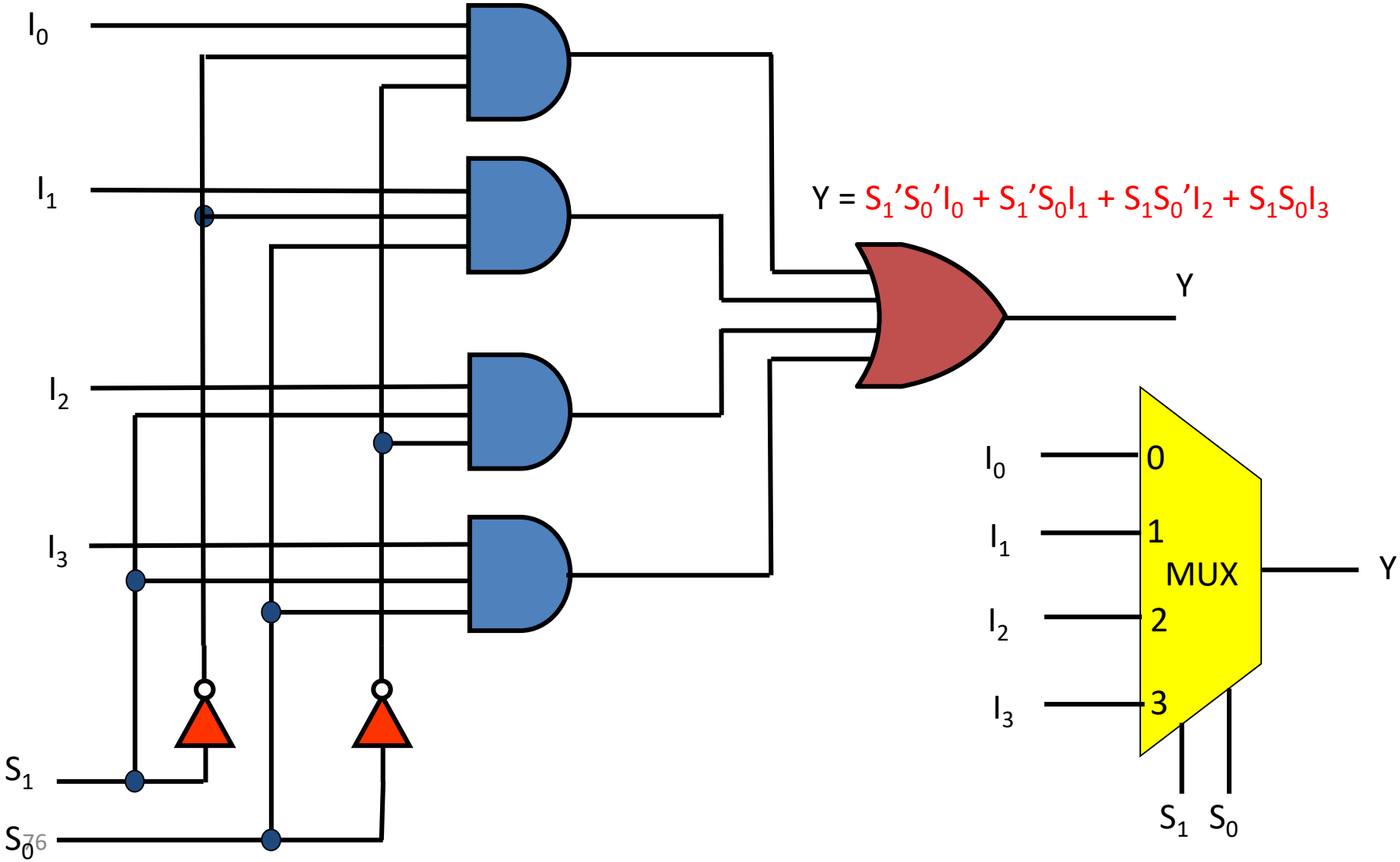
S_1	S_0	Y
0	0	
0	1	
1	0	
1	1	

Y = ?

Interpretation:

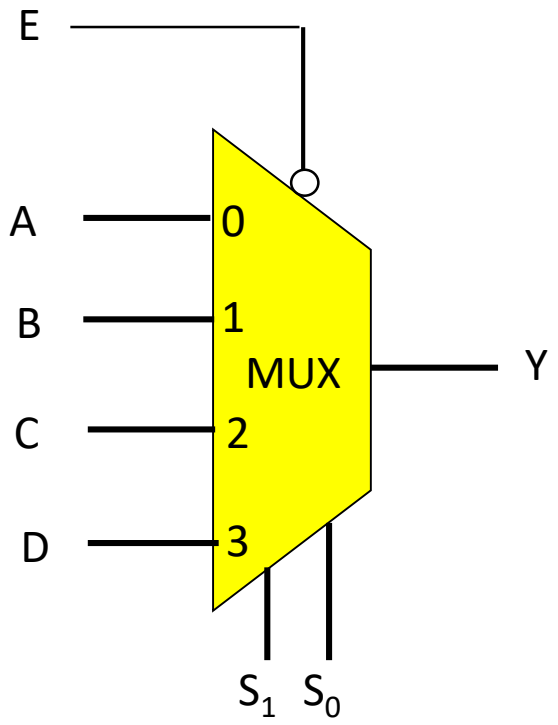
- In case $S_1 = 0$ and $S_0 = 0$, Y selects I_0
- In case $S_1 = 0$ and $S_0 = 1$, Y selects I_1
- In case $S_1 = 1$ and $S_0 = 0$, Y selects I_2
- In case $S_1 = 1$ and $S_0 = 1$, Y selects I_3

4-to-1-Line Multiplexer: Circuit



Multiplexer with Enable Input

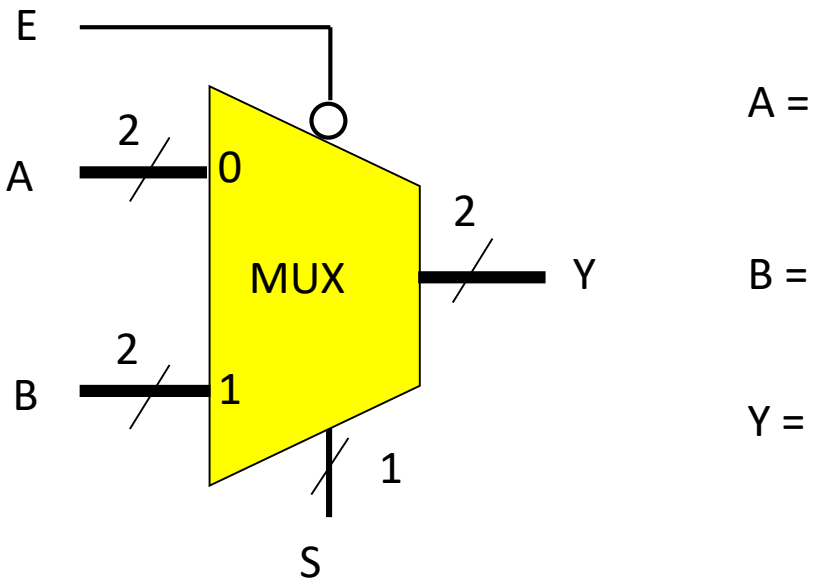
- To select a certain building block we use enable inputs



E	S	Y
1	XX	
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

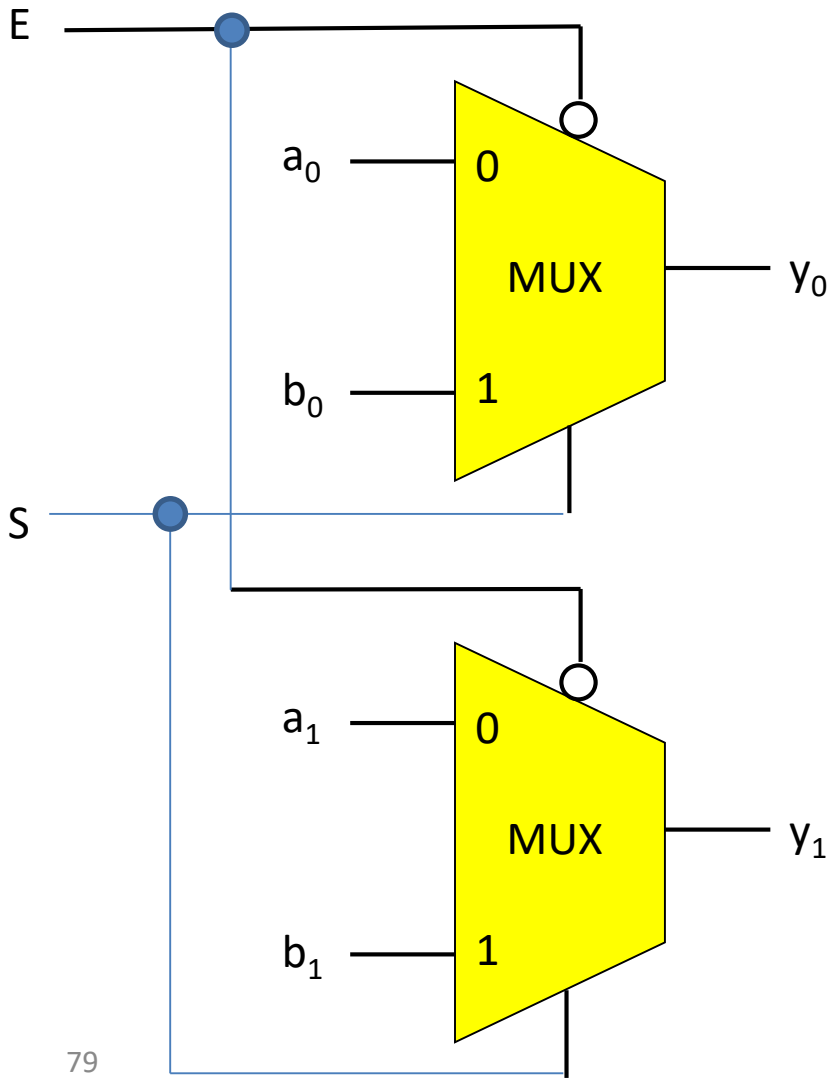


A =

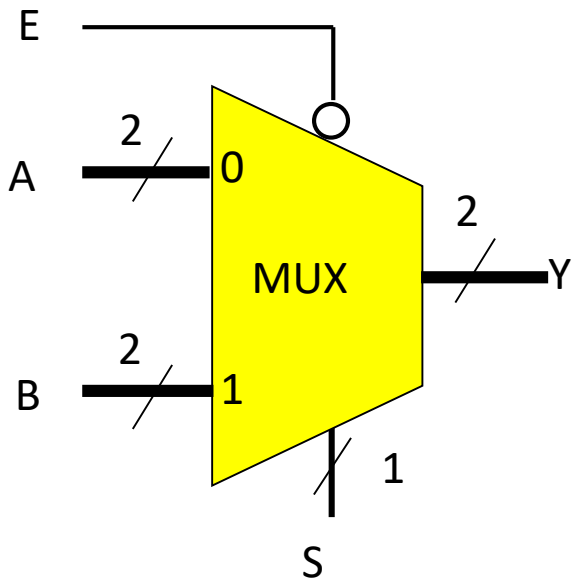
B =

Y =

Multiple-bit Selection Logic 2/2

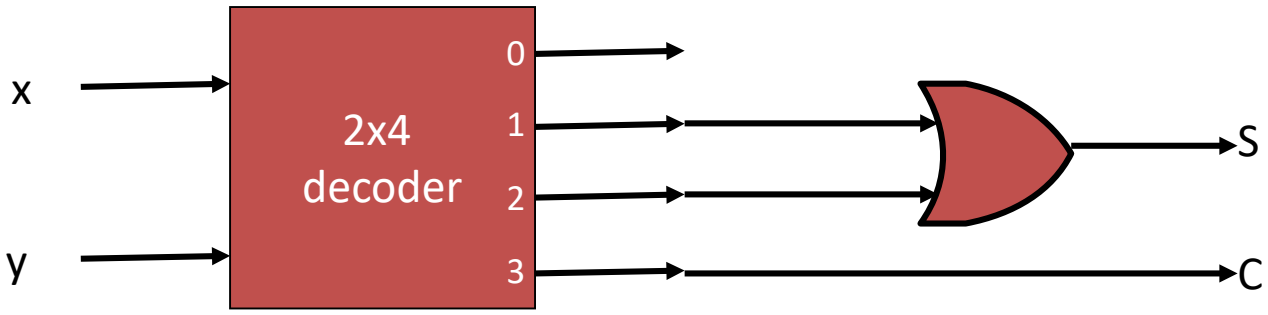


E	S	Y
1	X	all 0's
0	0	A
0	1	B



Design with Multiplexers 1/2

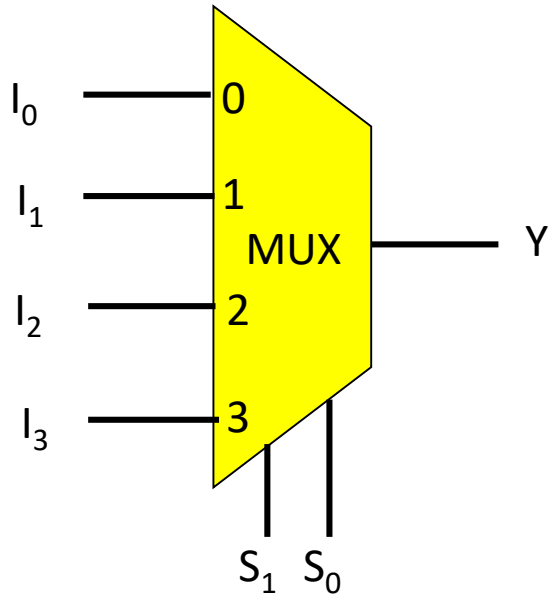
- Reminder: design with decoders
- Half adder
  - $C = xy = \sum(3)$
  - $S = x \oplus y = x'y + xy' = \sum(1, 2)$



- 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

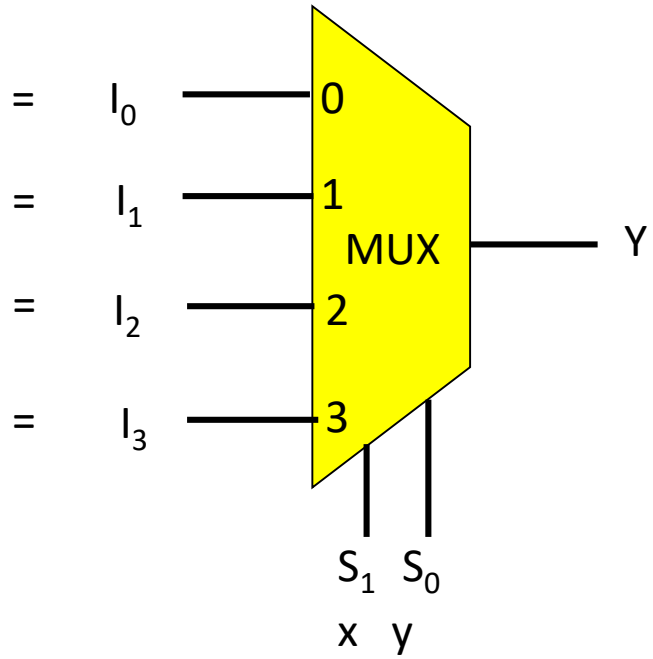


- $S_1 \rightarrow x$
- $S_0 \rightarrow y$
- $S_1'S_0' = x'y'$,
- $S_1'S_0 = x'y$,
- $S_1S_0' = xy'$,
- $S_1S_0 = xy$

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

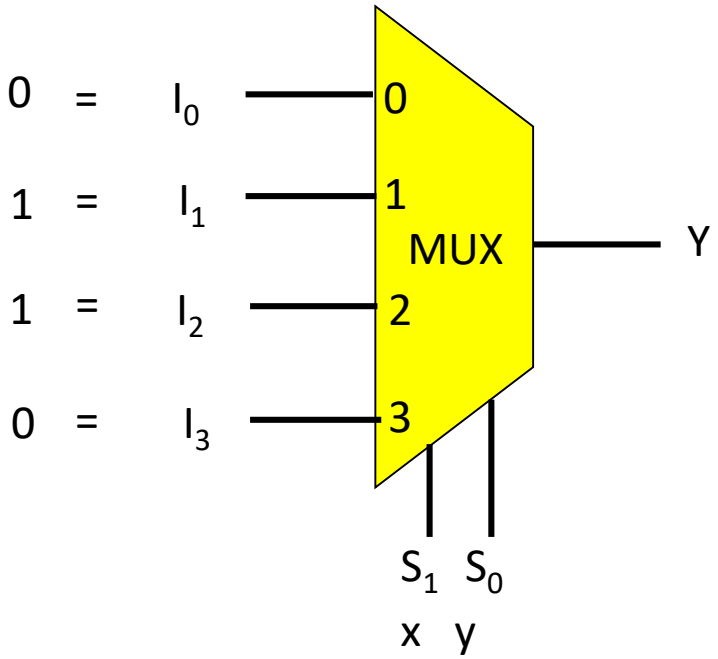
Example: Design with Multiplexers

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



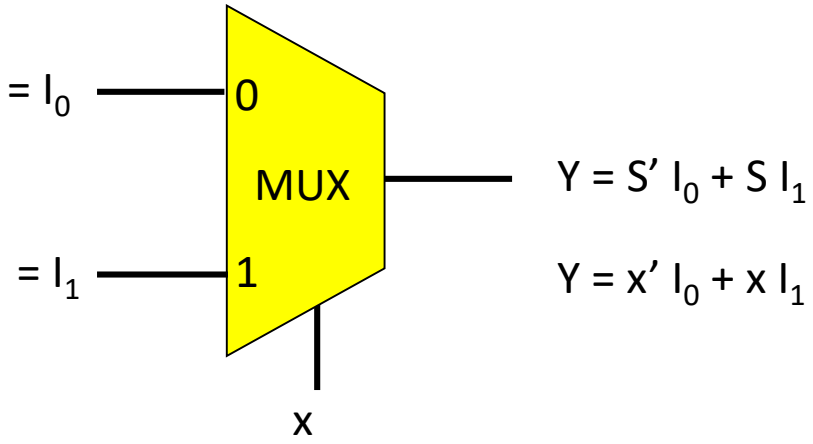
Example: Design with Multiplexers

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



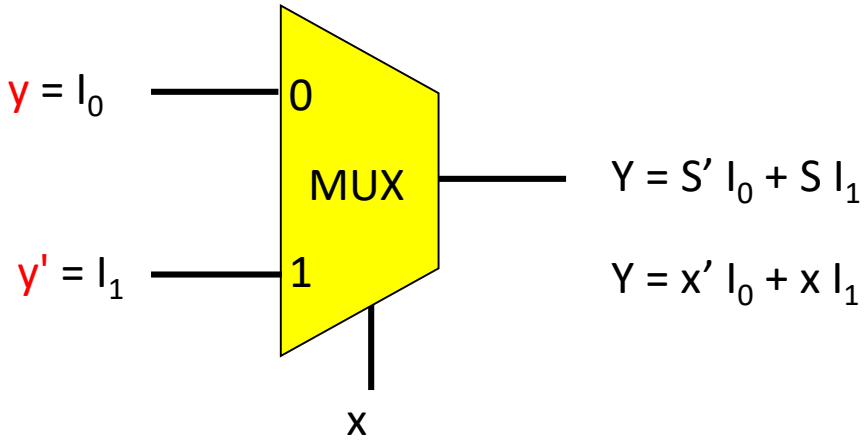
Design with Multiplexers Efficiently

- 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: $Y = \Sigma(1, 2)$



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: $Y = \Sigma(1, 2)$



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.

Example: Design with Multiplexers

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

x	y	z	F
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

F =

F =

F =

F =

Example: Design with Multiplexers

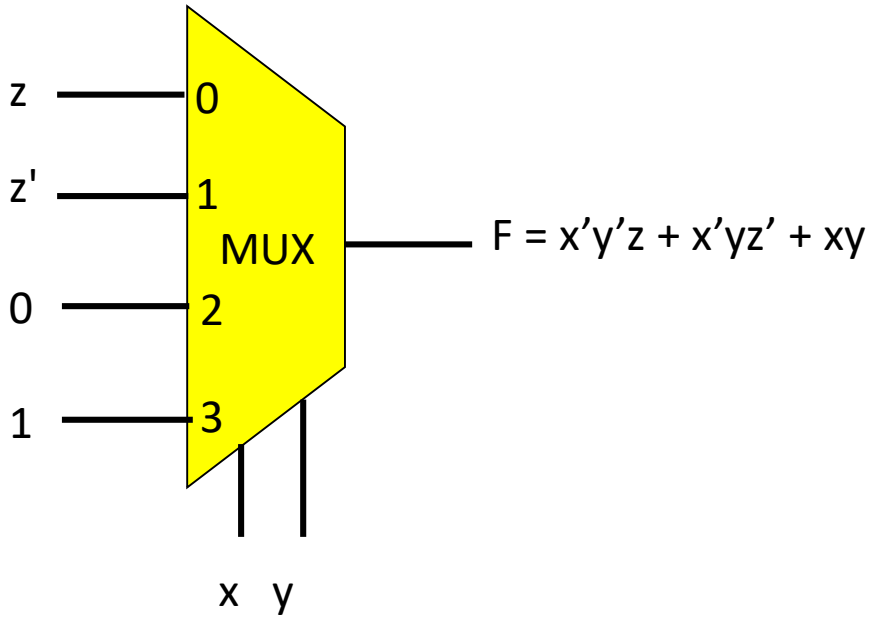
- $F(x, y, z) = \Sigma(1, 2, 6, 7)$
  - $F = x'y'z + x'yz' + xyz' + xyz$
  - $Y = S_1'S_0'\, I_0 + S_1'S_0\, I_1 + S_1S_0'\, I_2 + S_1S_0\, I_3$
  - $I_0 = z \quad I_1 = z' \quad I_2 = 0 \quad I_3 = 1$

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	

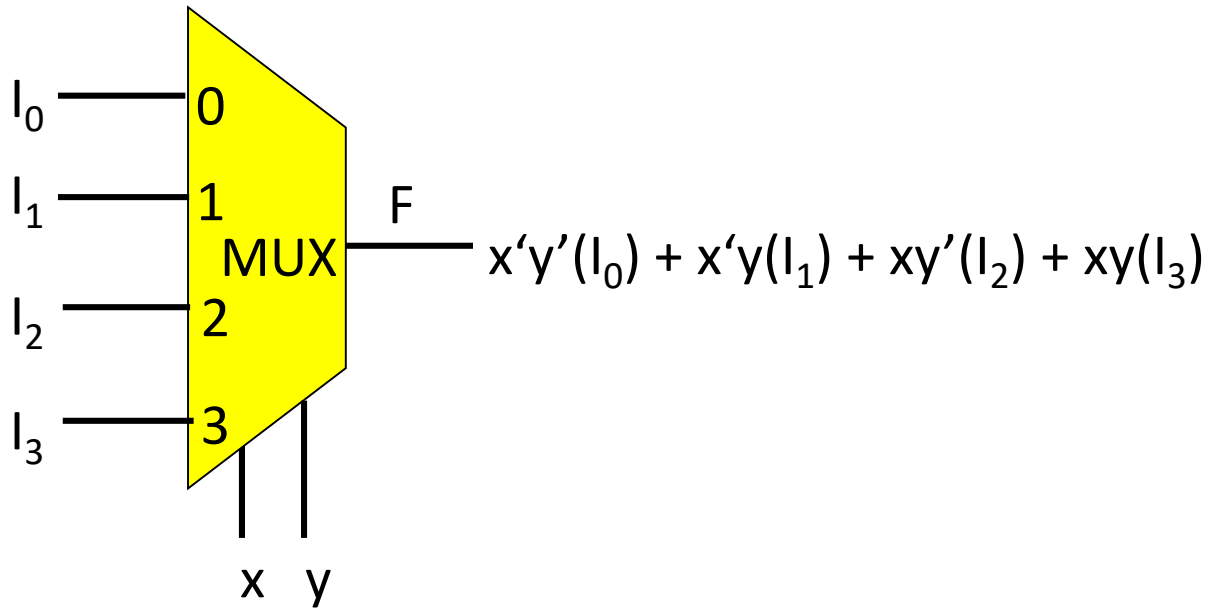
Example: Design with Multiplexers

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

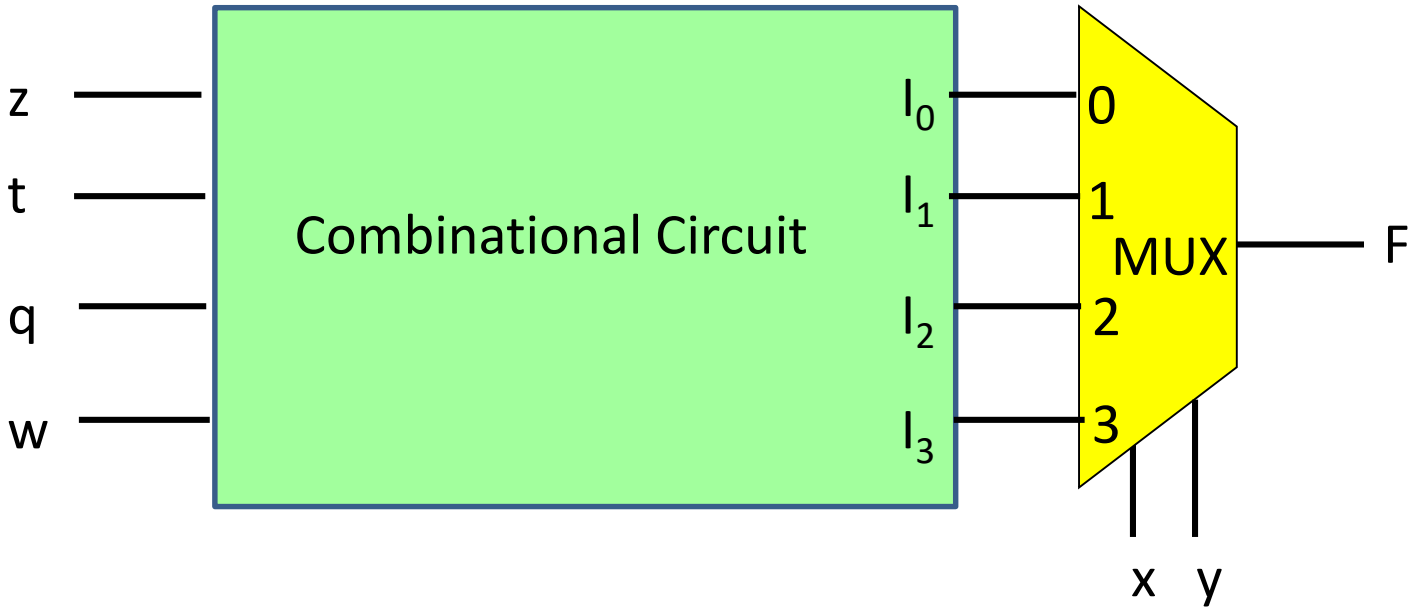


$F=x'y'(0) + x'y(1) + xy'(1) + xy(0)$

$F=x'y'(z) + x'y(z') + xy'(0) + xy(1)$

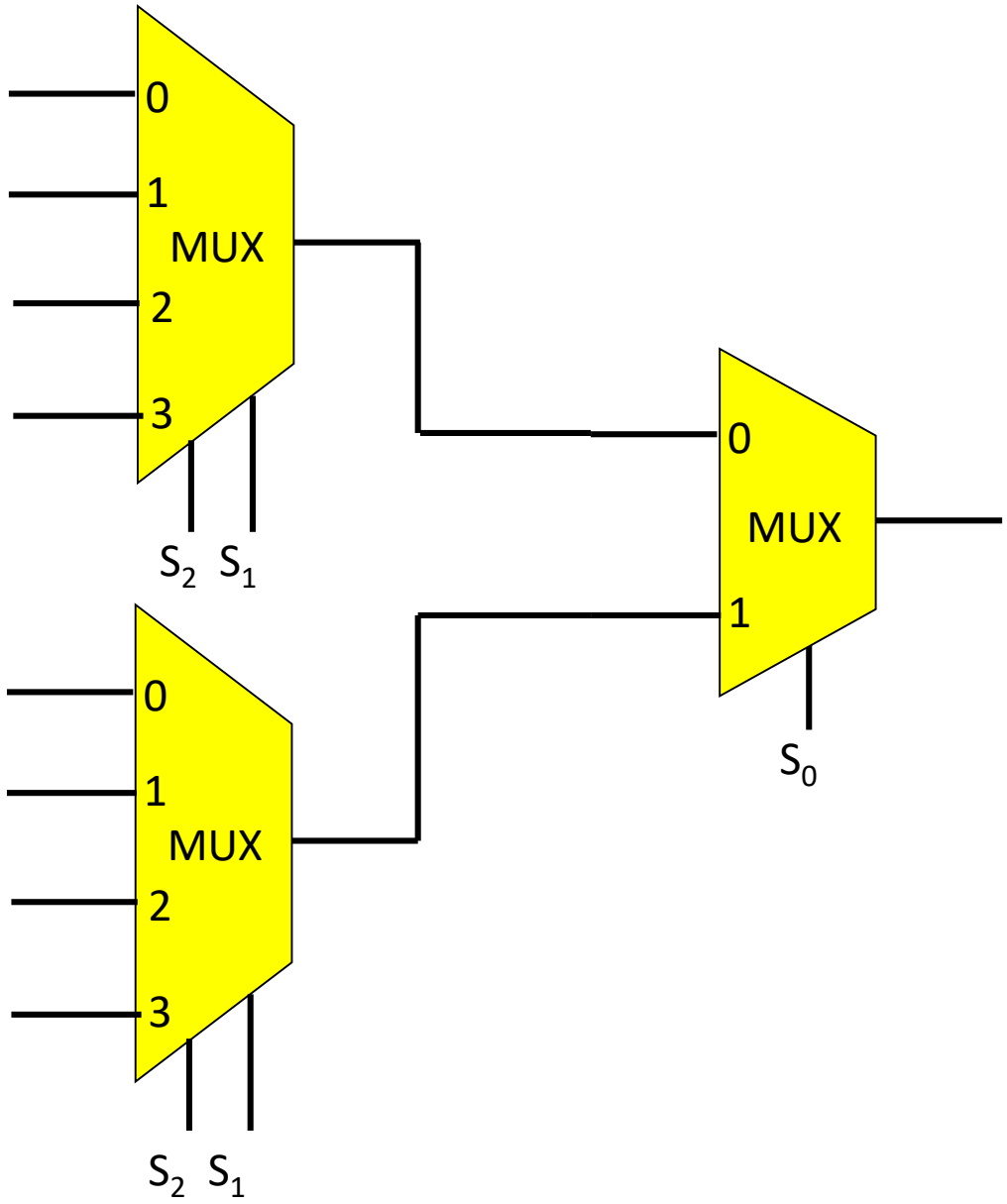
$F=x'y'(P(z,t,q,w)) + x'y(Q(z,t,q,w)) + xy'(R(z,t,q,w)) + xy(S(z,t,q,w))$

Design with Multiplexers

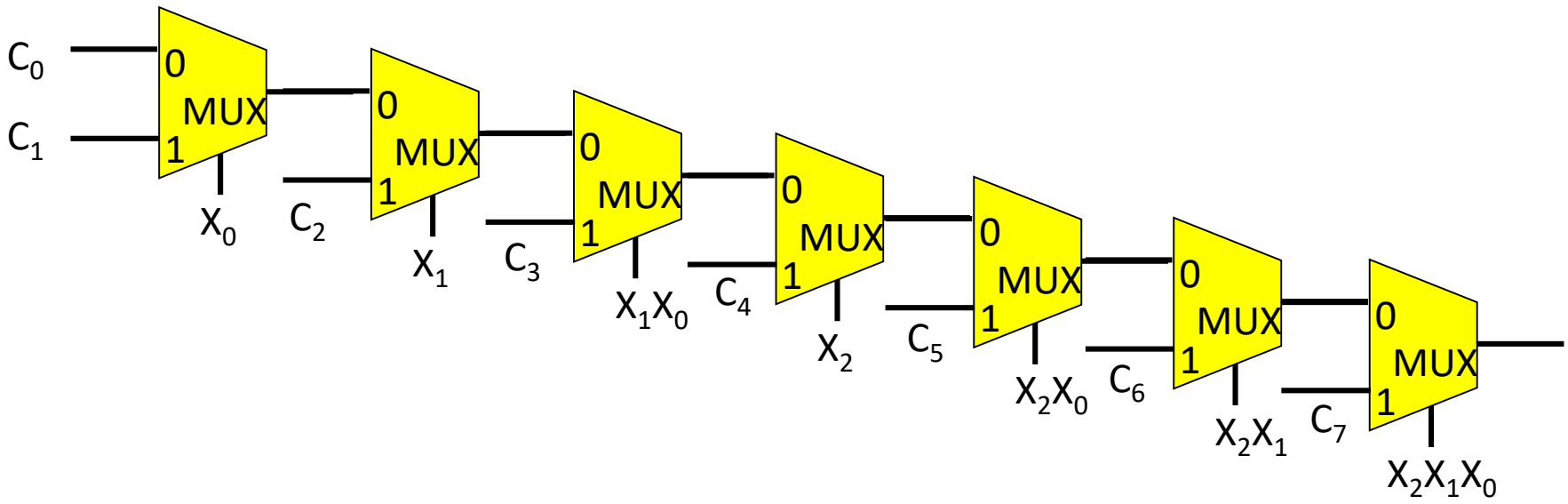


F=x‘y’(P(z,t,q,w)) + x‘y(Q(z,t,q,w)) + xy’(R(z,t,q,w)) + xy(S(z,t,q,w))

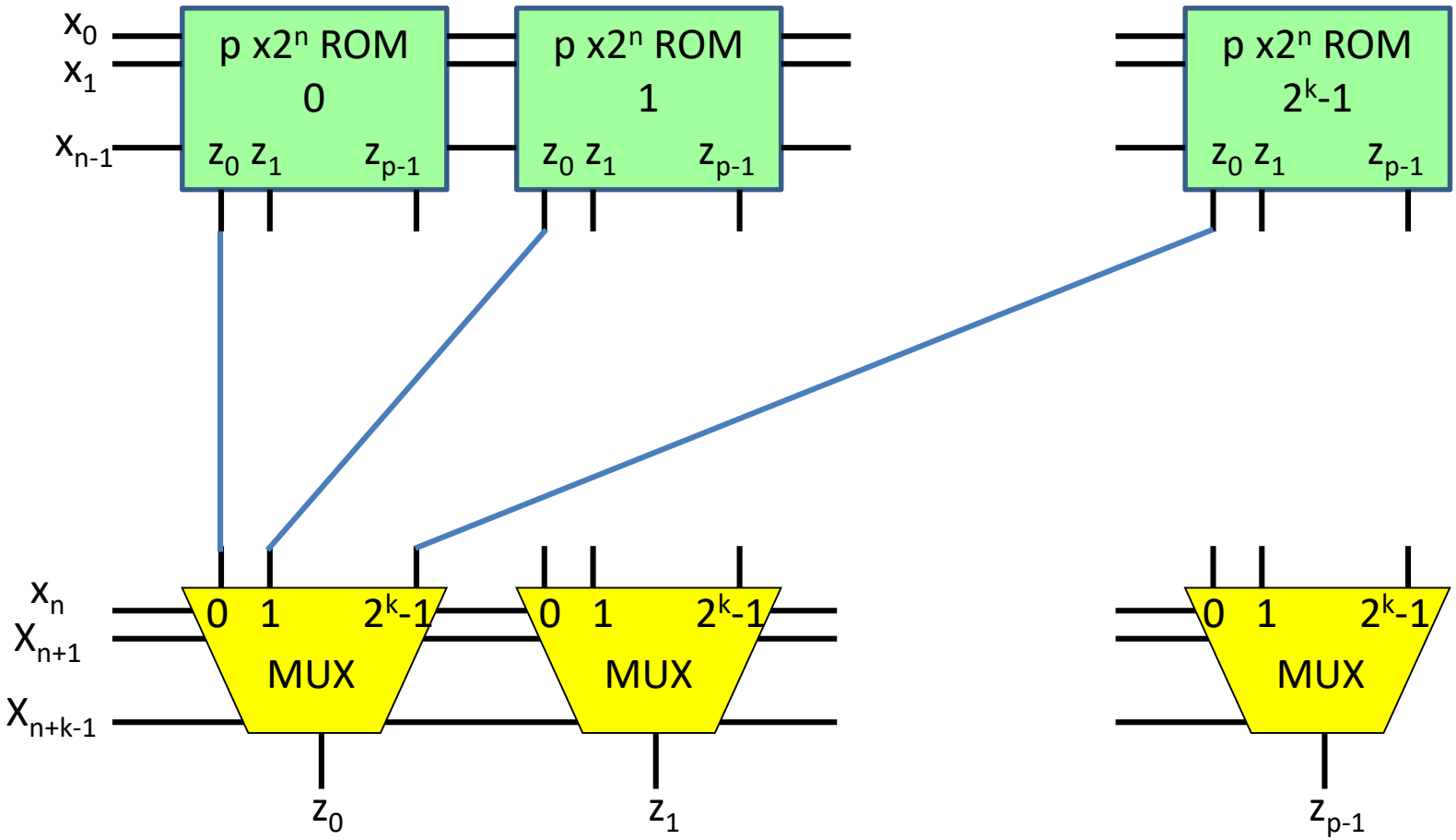
Combining Multiplexers



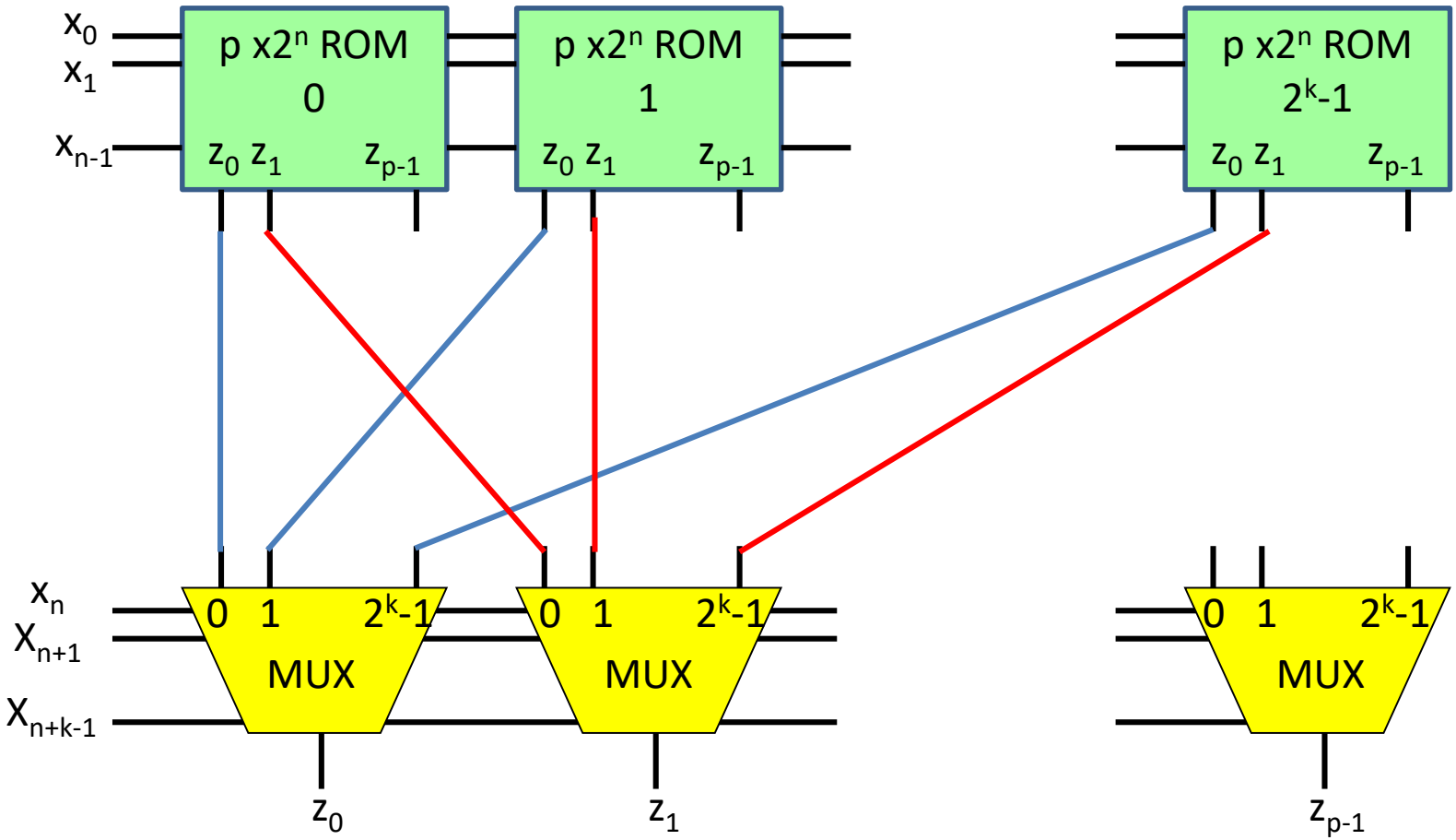
Example


C0
C1
0
MUX
1
X0
C2
0
MUX
1
X1
C3
0
MUX
1
X1X0
C4
0
MUX
1
X2
C5
0
MUX
1
X2X0
C6
0
MUX
1
X2X1
C7
0
MUX
1
X2X1X0

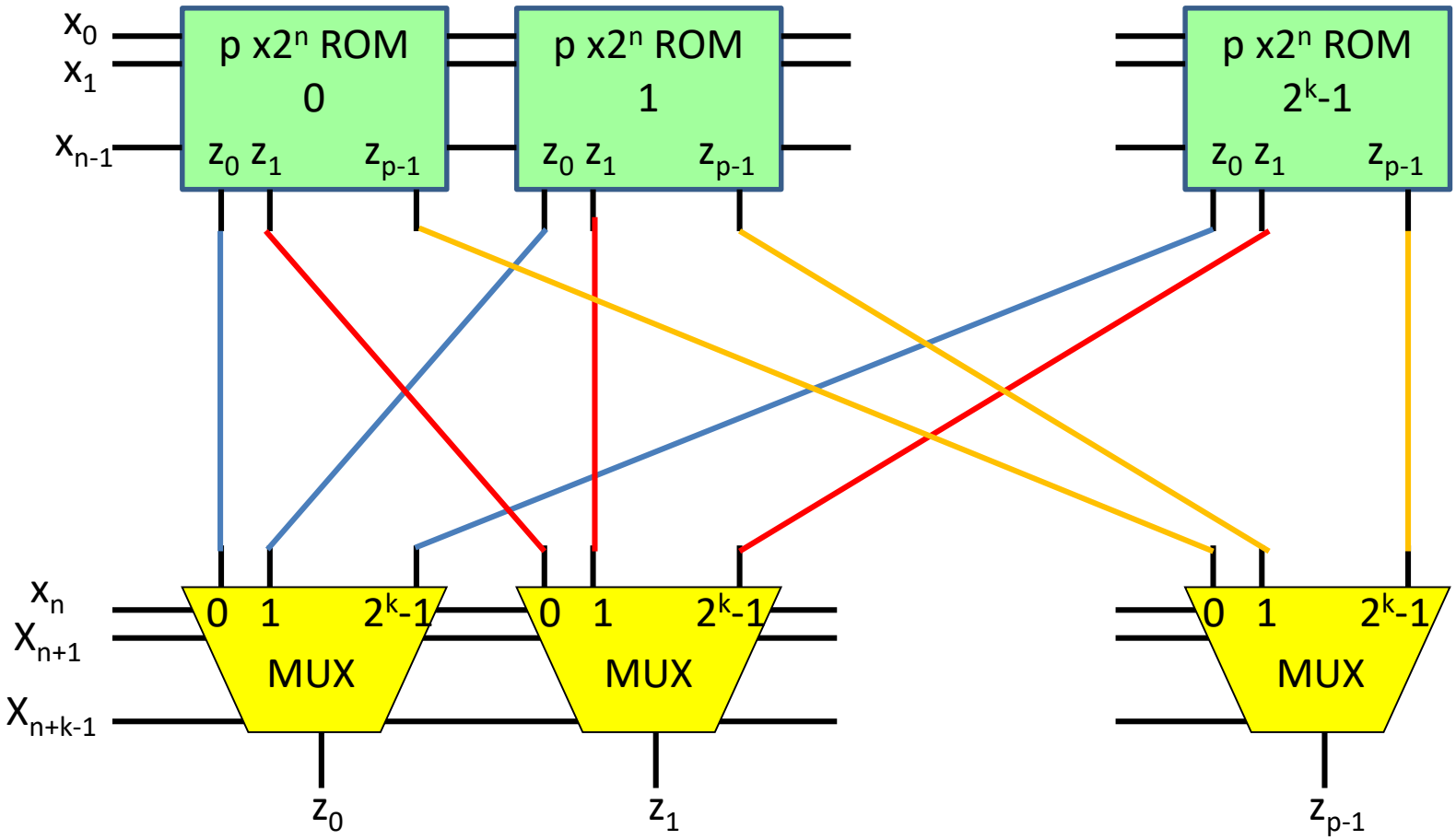
Example- 2 Level ROM



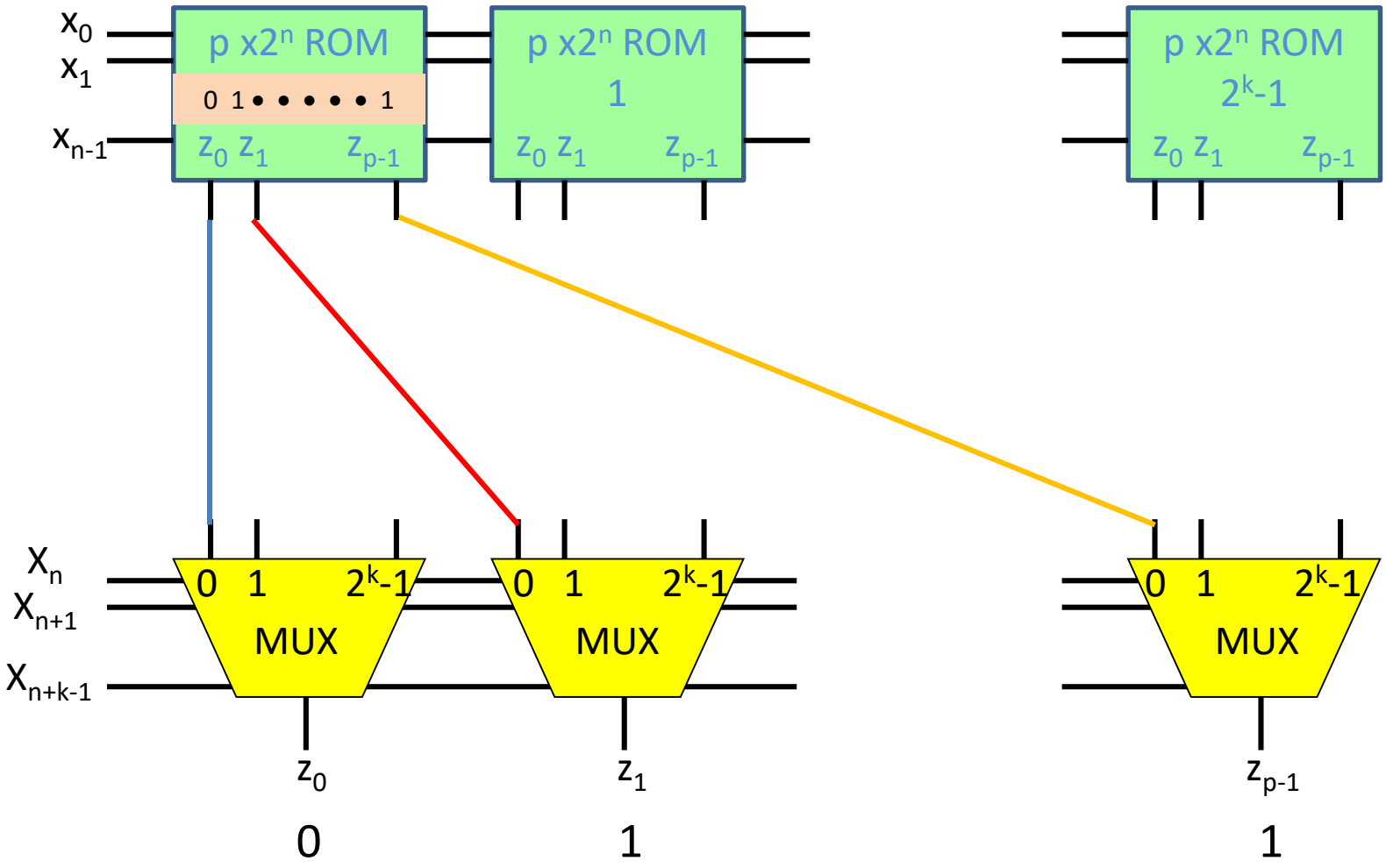
Example- 2 Level ROM



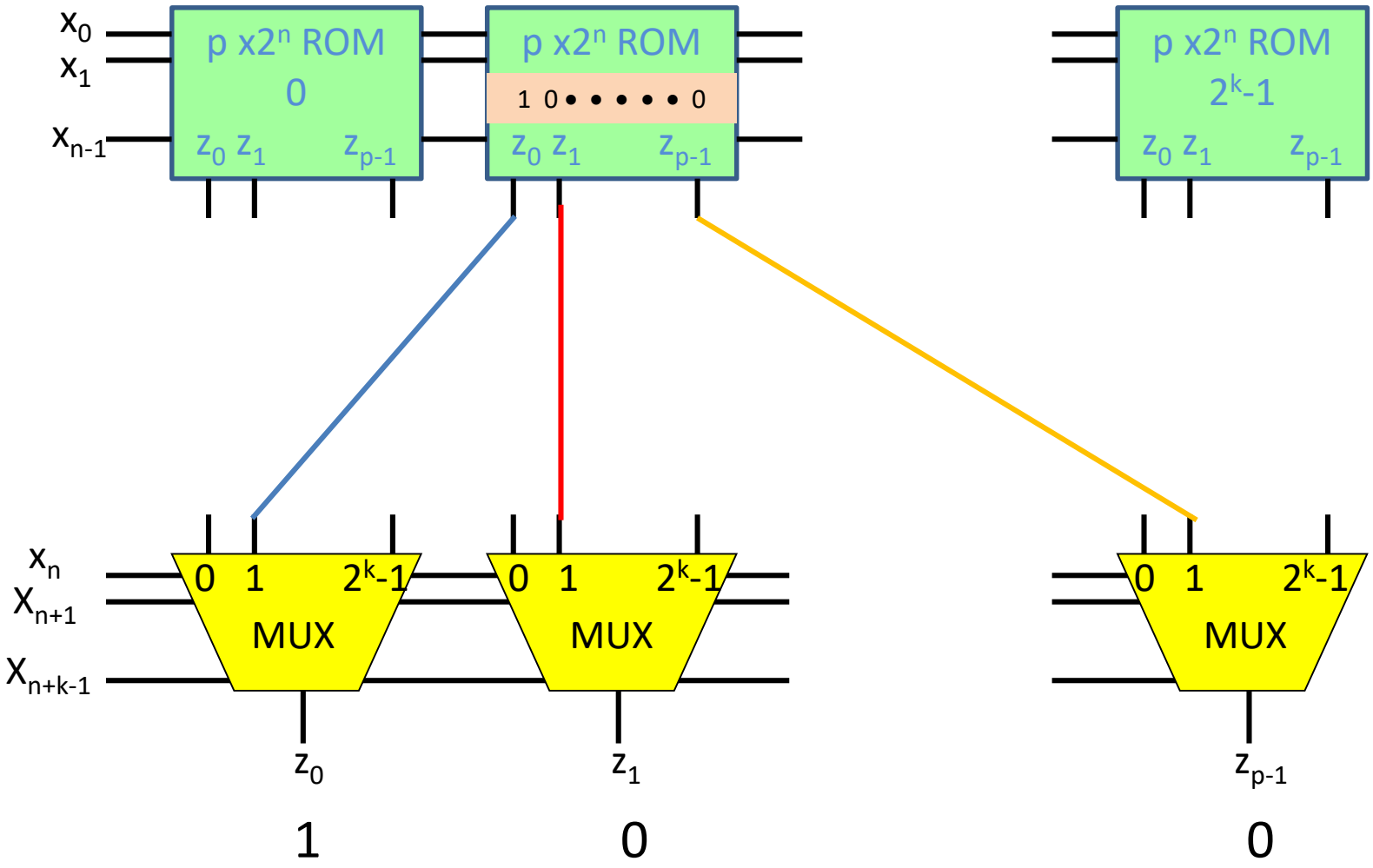
Example- 2 Level ROM



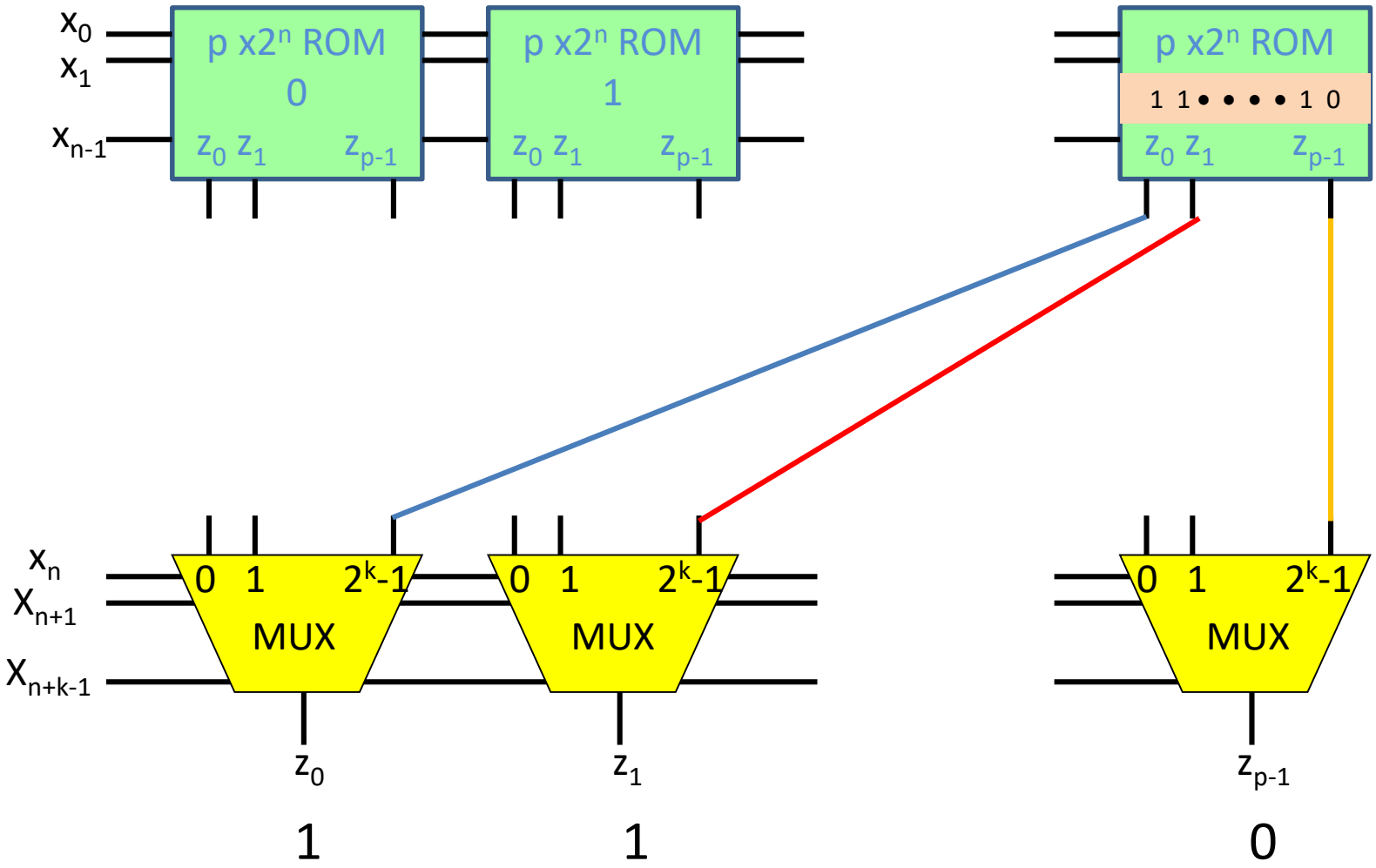
Example- 2 Level ROM



Example- 2 Level ROM

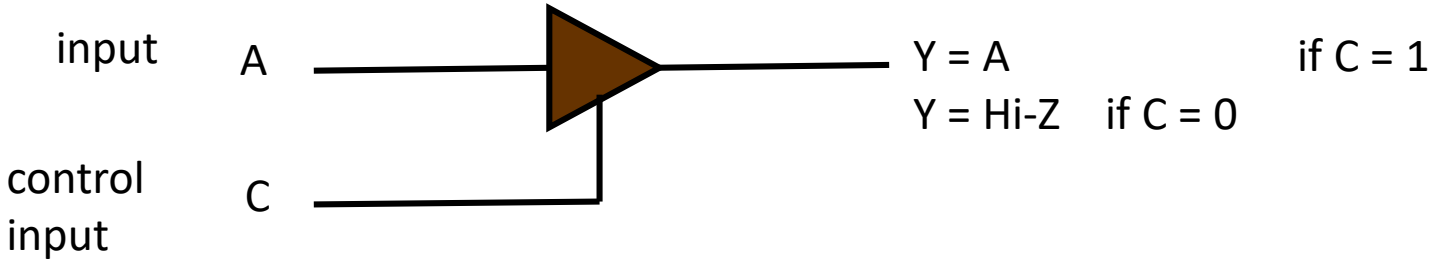


Example- 2 Level ROM



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)

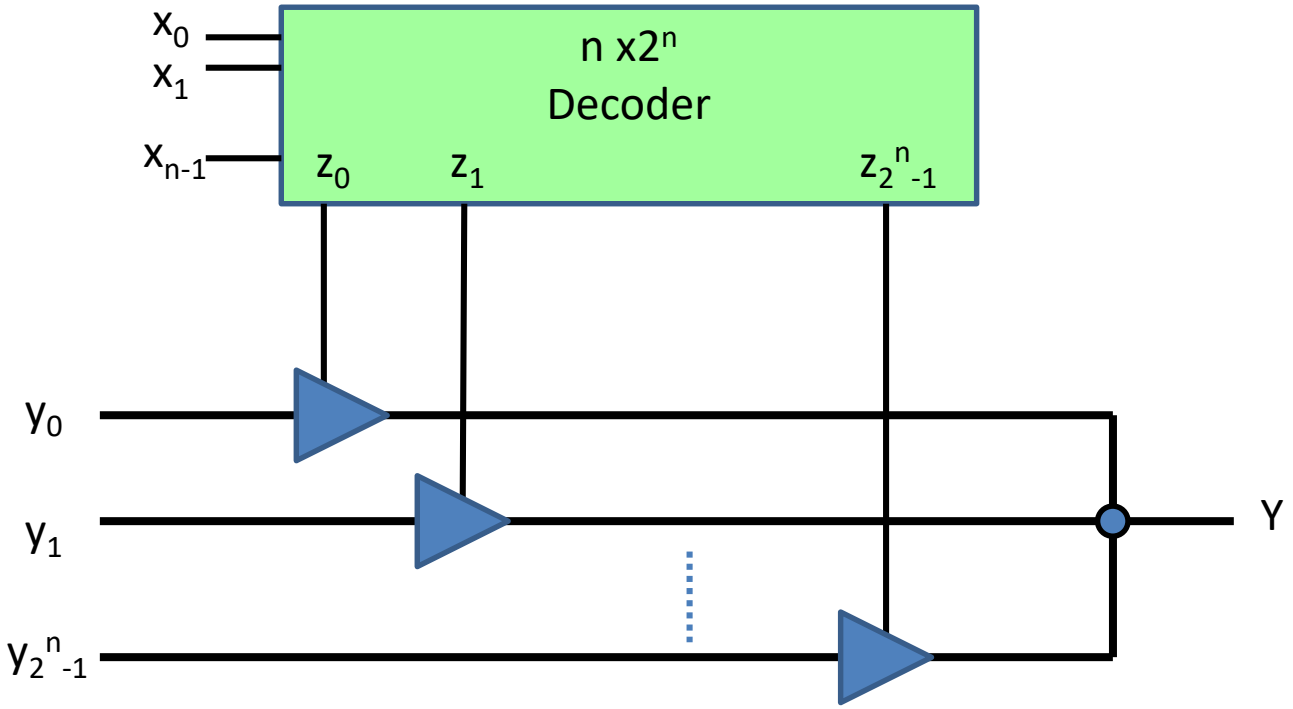


3-State Buffers

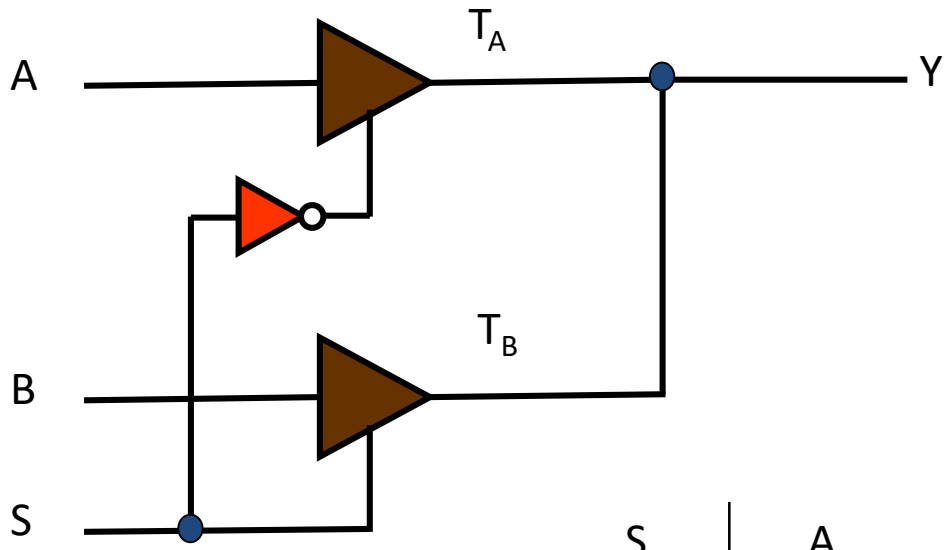
- Remember that we cannot connect the outputs of other logic gates.
- We can connect the outputs of three-state buffers
  - provided that no two three-state buffers drive the same wire to opposite 0 and 1 values at the same time.

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

Example- 3 State MUX



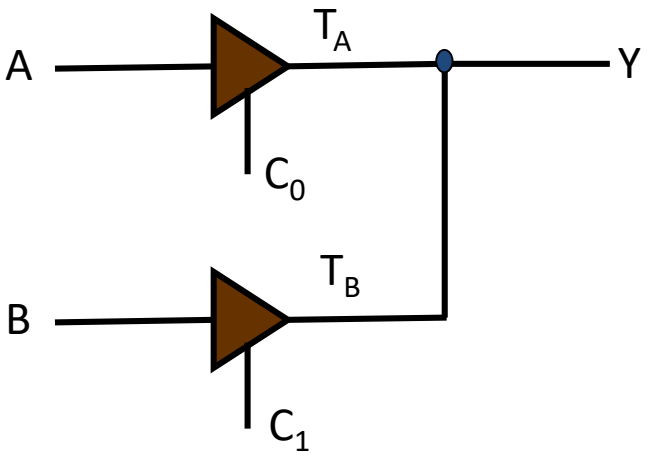
Multiplexing with 3-State Buffers



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

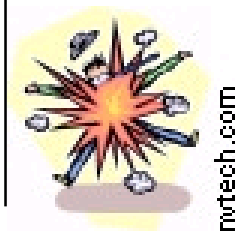
It is, in fact, a
2-to-1-line MUX

Two Active Outputs - 1



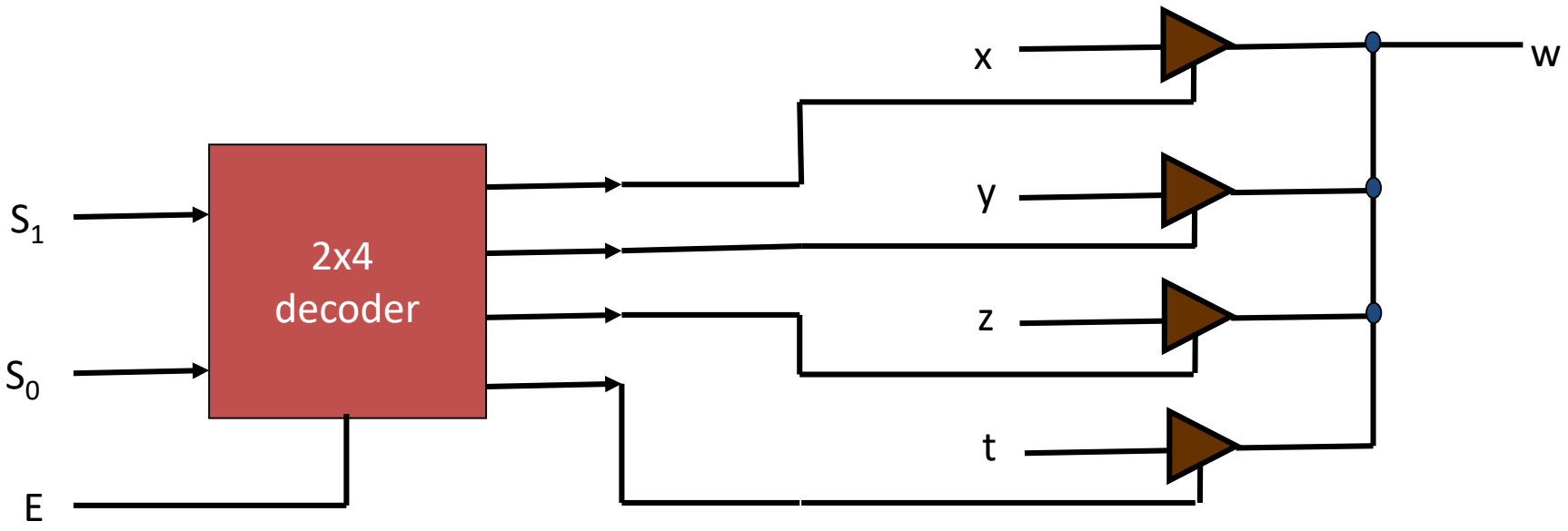
What will happen
if $C_1 = C_0 = 1$?

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	

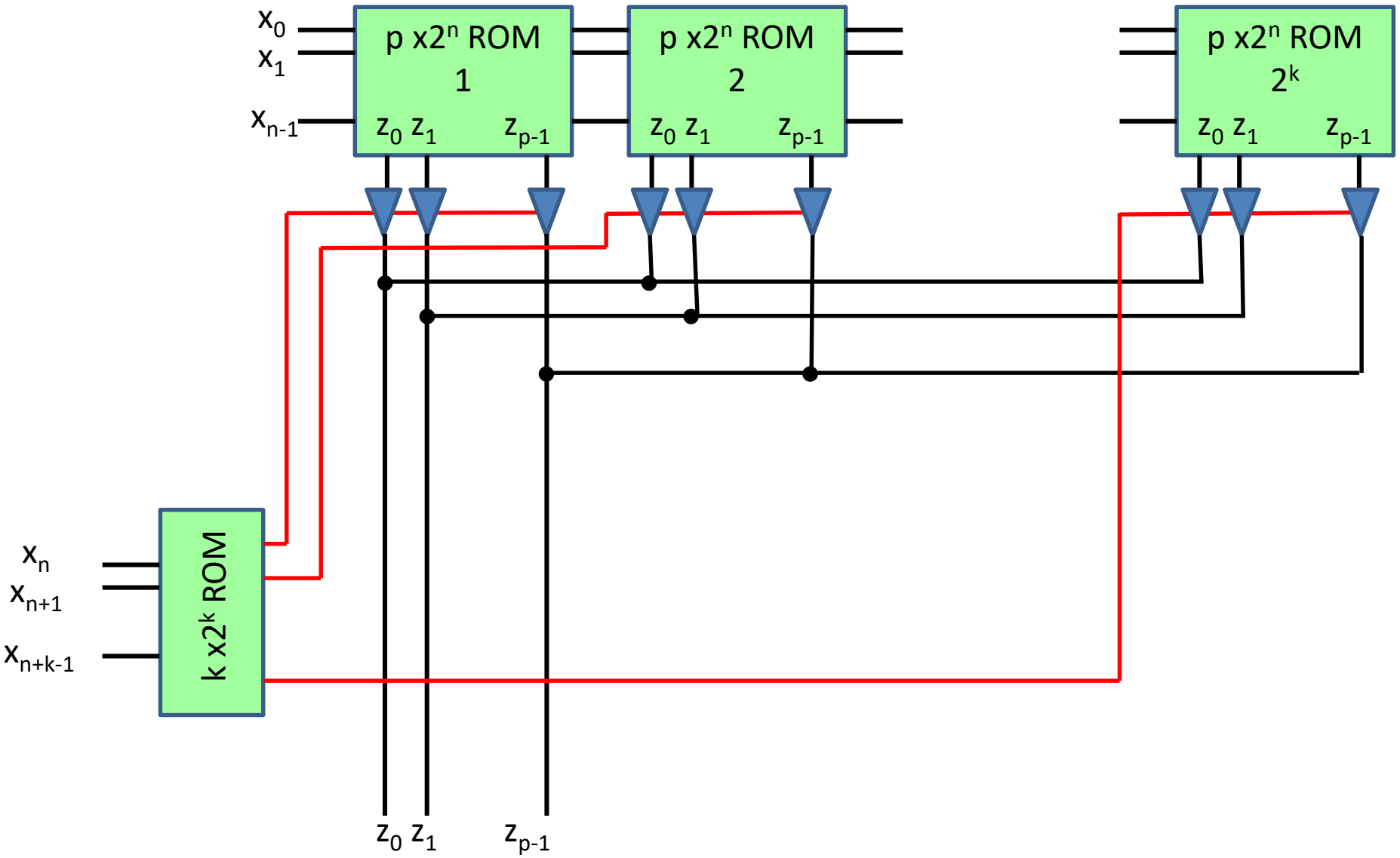


Design Principle with 3-State Buffers

- 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.

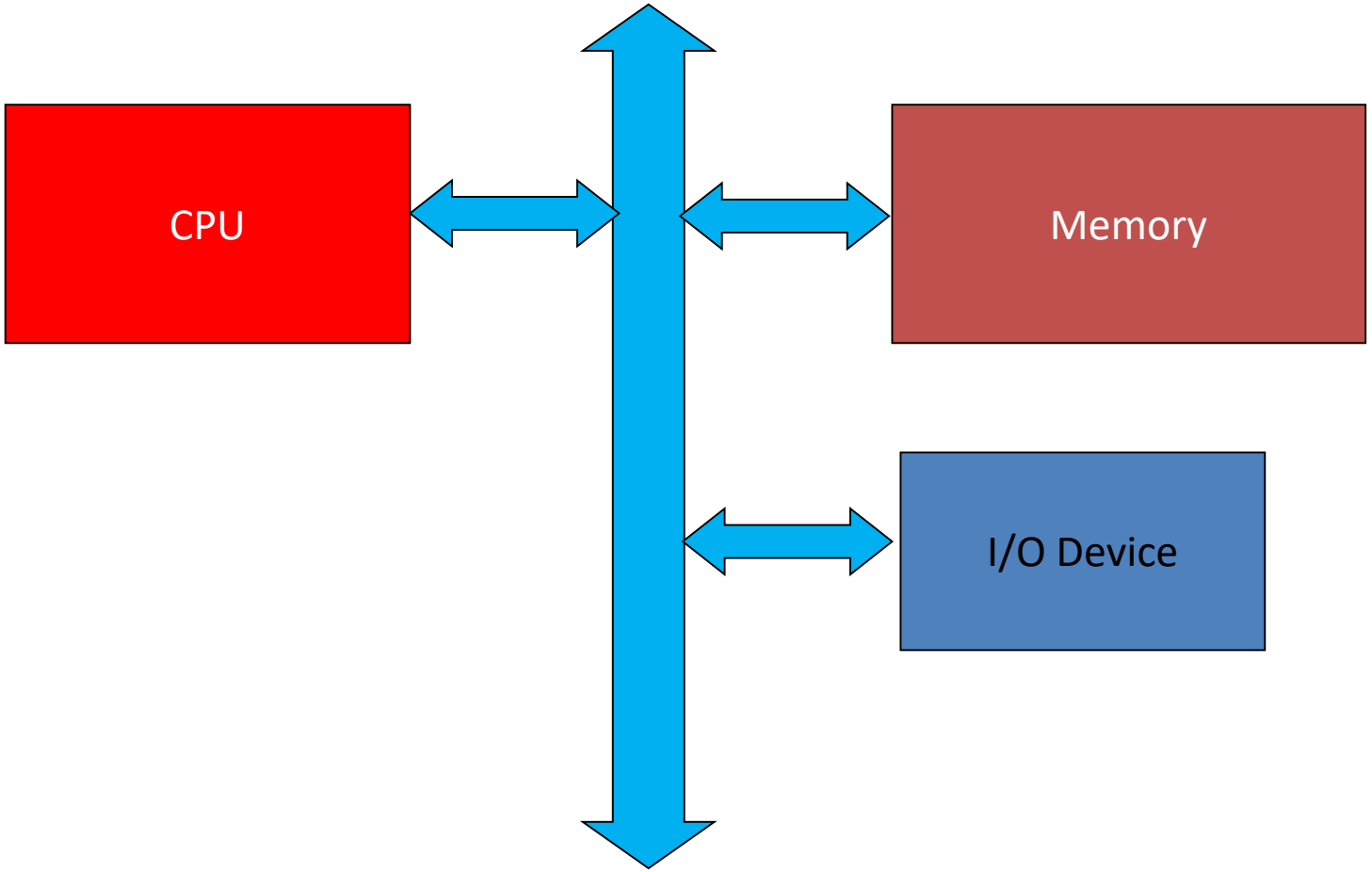


Example- 3 State Buffer & ROM



Busses with 3-State Buffers

- There are important uses of three-state buffers



Design with Verilog

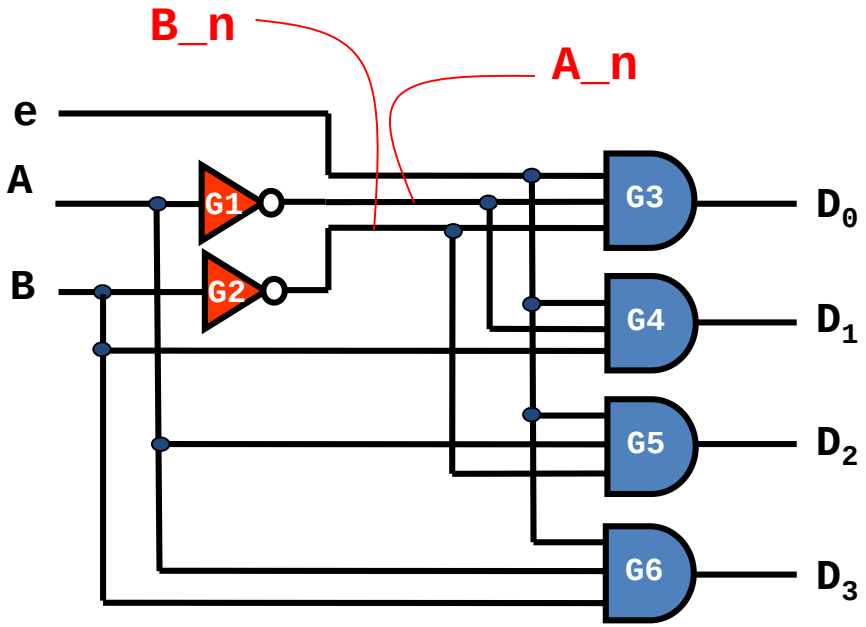
- Gate Level Design

```
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;
```

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

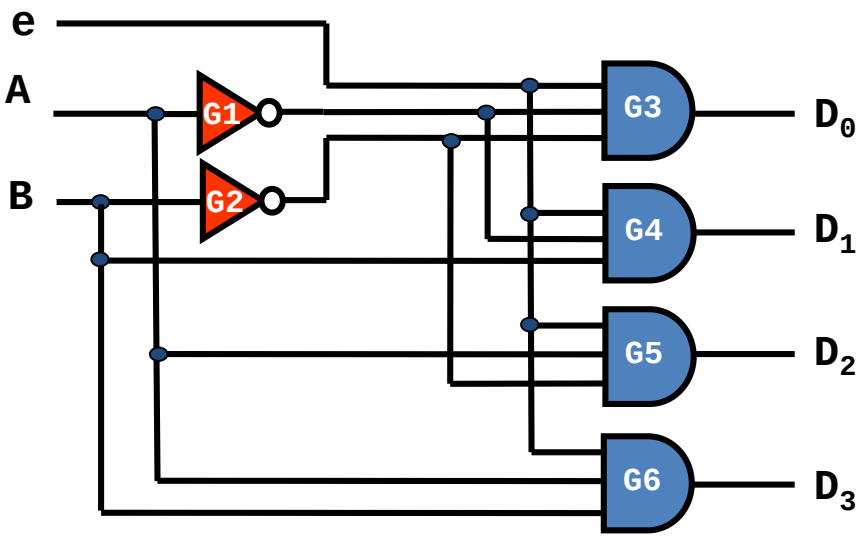


Dataflow Modeling

- Dataflow modeling uses a number of operators that act on operands
  - About 30 different operators

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

```
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;
```