Part 4

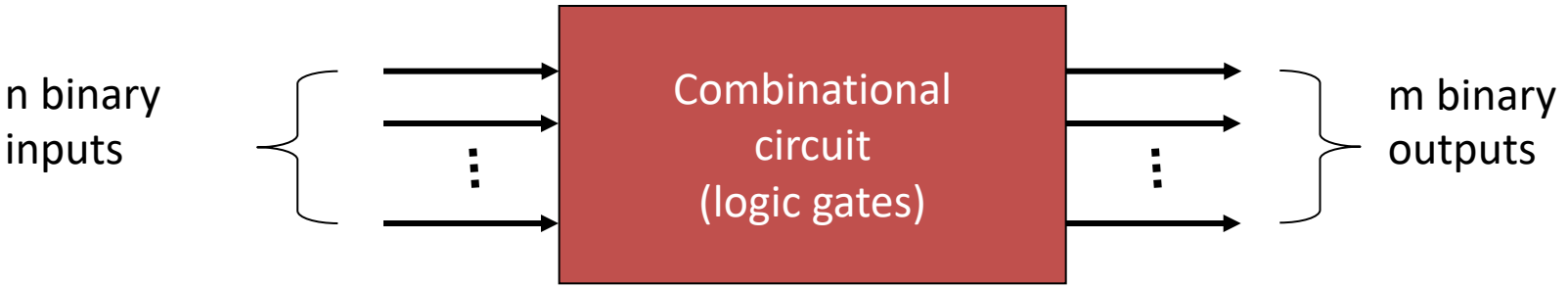
***Digital Design and Computer Architecture*, 2nd Edition**

David Money Harris and Sarah L. Harris

Classification of Digital Circuits

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

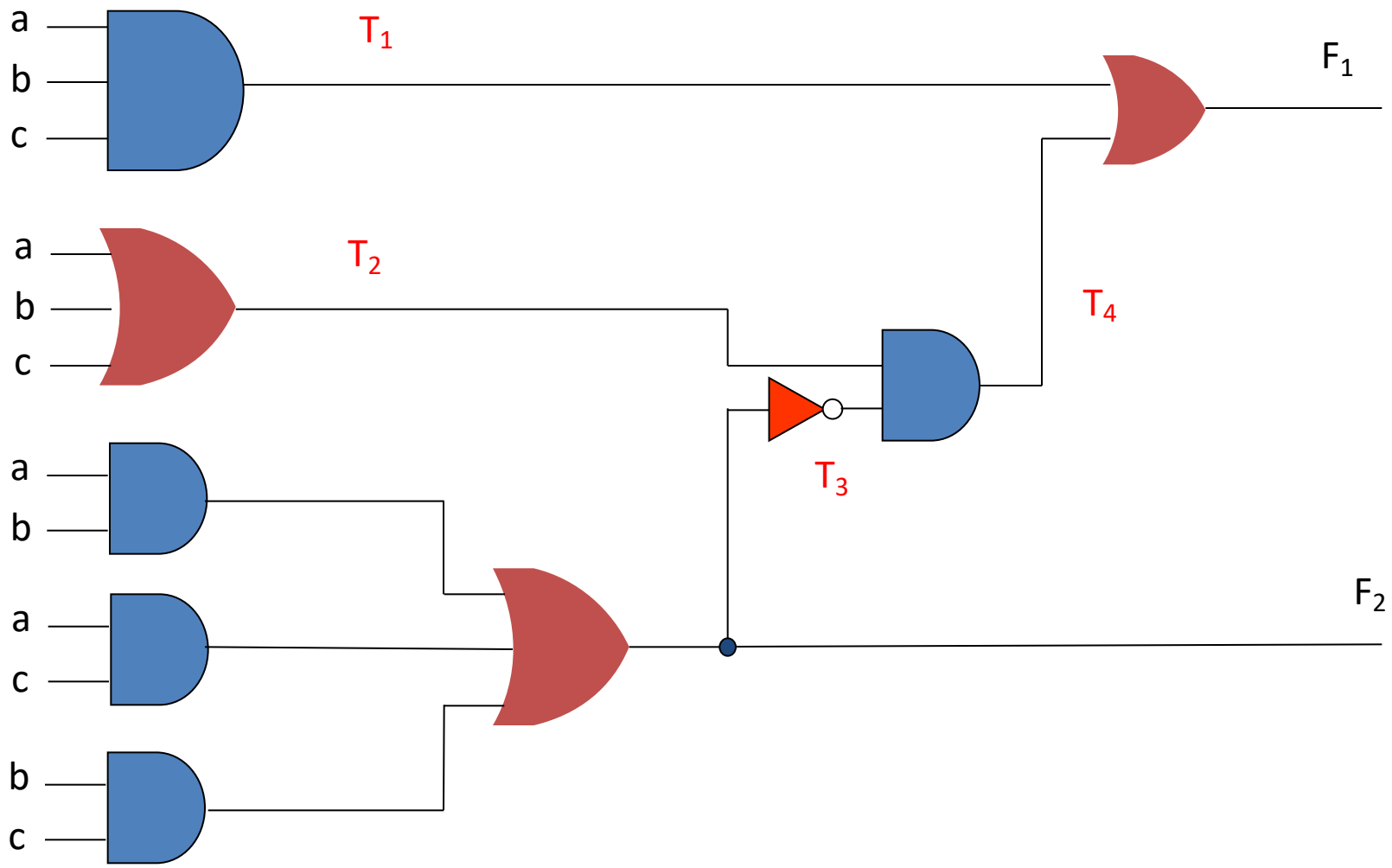
- Analysis: to find out the function that a given circuit implements
  - We are given a logic circuit and
  - we are expected to find out
    1. Boolean function(s)
    2. Truth table
    3. A possible explanation of the circuit operation (i.e. **what it does**)

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
  - $T_1 = abc$
  - $T_2 = a + b + c$
  - $F_2 =$
  - $T_3 =$
  - $T_4 = T_3T_2$
  - $F_1 = T_1 + T_4$

    $=$

    $=$

    $=$

    $=$

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

$F_1 = a \oplus b \oplus c$

$F_2 = ab + ac + bc$

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 (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
- Possible Design Constraints:
  - number of gates
  - propagation time of the signal 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 Binary Coded Decimal (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?
  - 4 inputs and 4 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				

The 4-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 ×1) + (1 ×4) + (0 ×2) + (1 ×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.
    - 0011 : 3, 1100 : 6
    - 0000 : 0, 1111 : 9

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

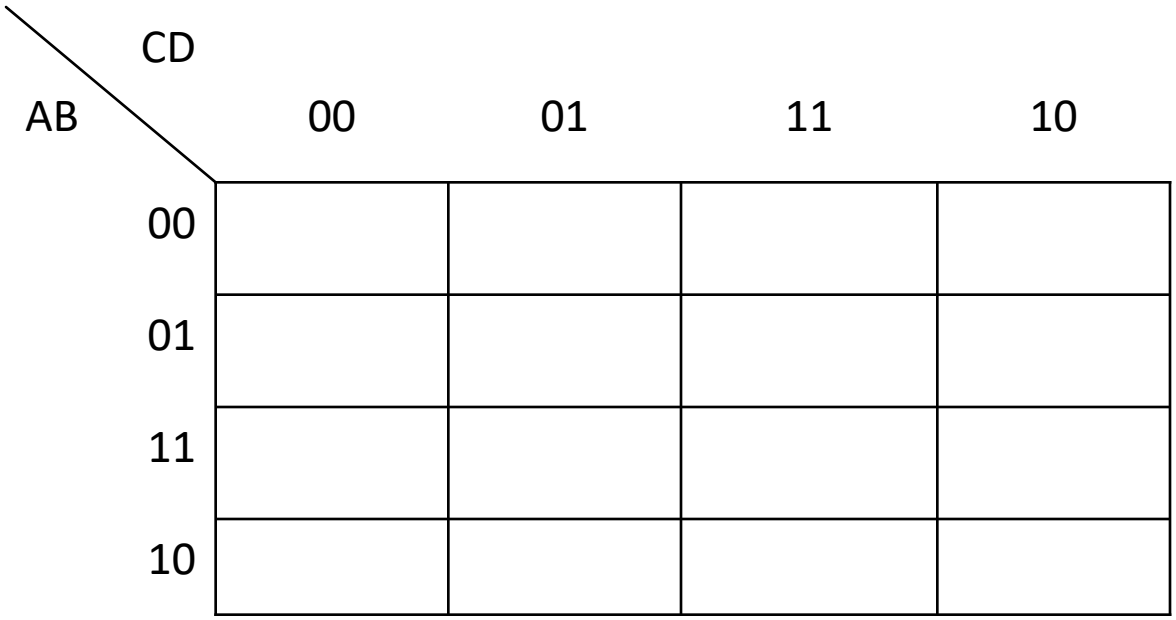
- Truth Table

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

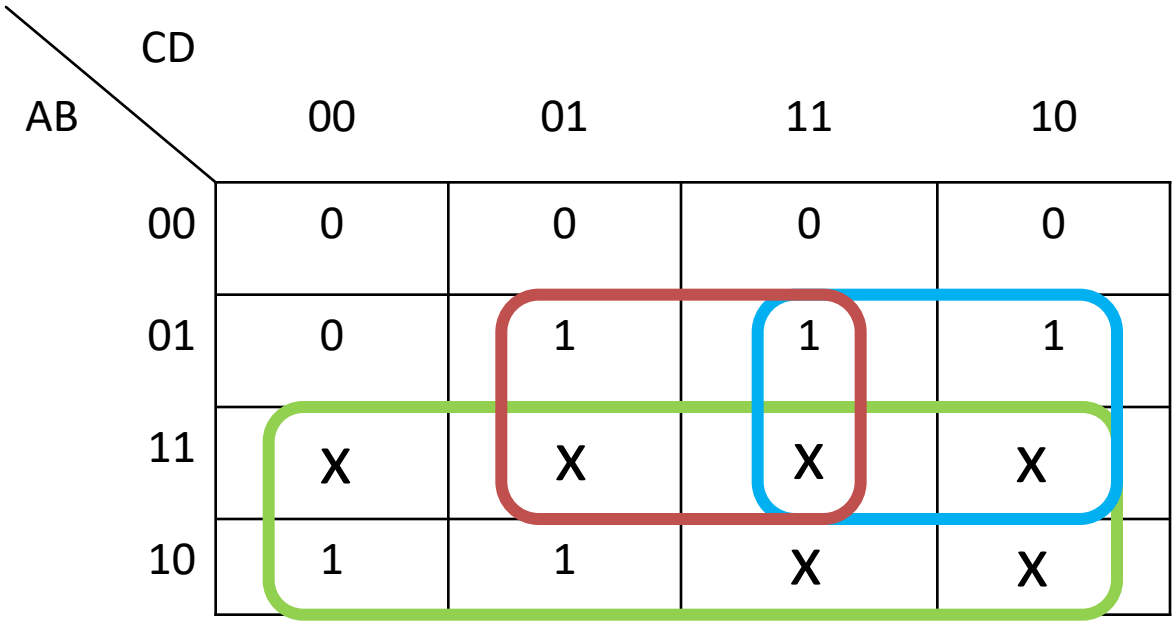
- Step 3: Obtain simplified Boolean expression for each output
- Output **x**:

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				

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:

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

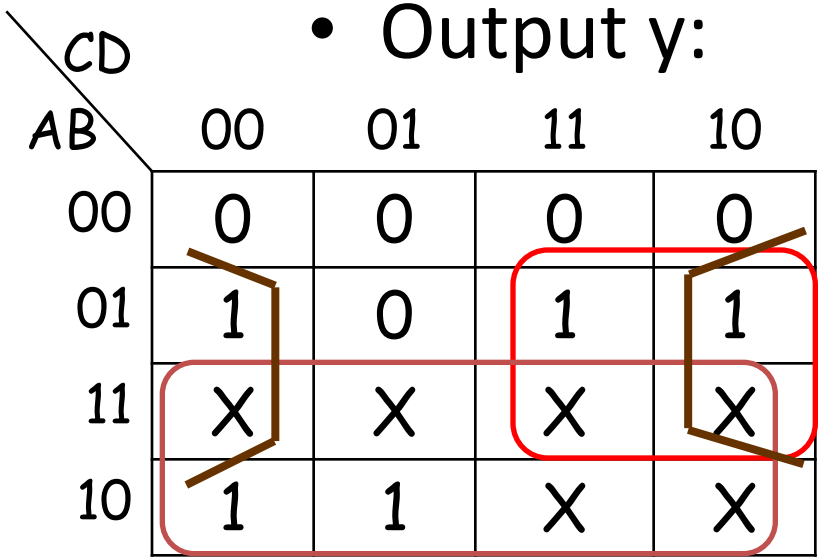
A	B	C	D	y	z
0	0	0	0	0	0
0	0	0	1	0	0
0	0	1	0	0	1
0	0	1	1	0	1
0	1	0	0	1	0
0	1	0	1	0	1
0	1	1	0	1	0
0	1	1	1	1	0
1	0	0	0	1	1
1	0	0	1	1	1
The rest				X	X

y =

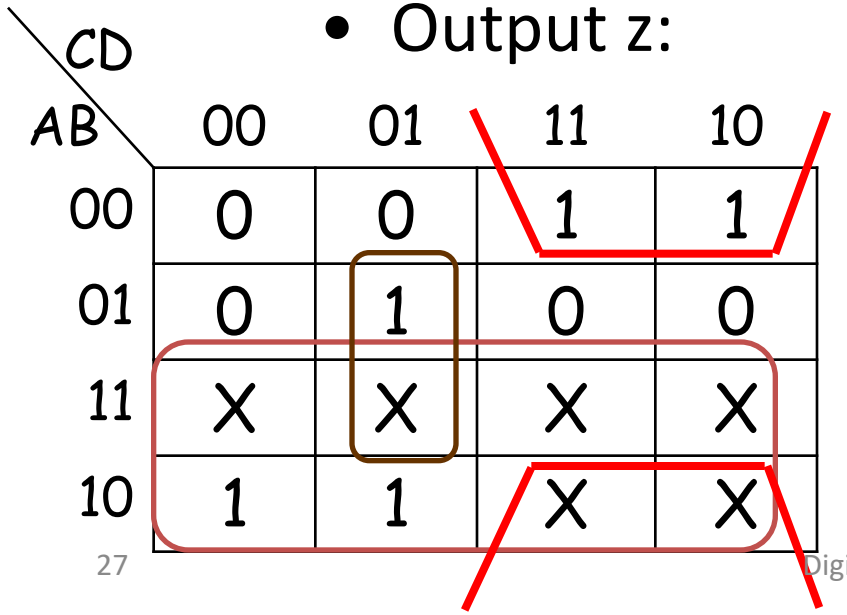
z =

Boolean Expressions for Outputs

- Output y:



- Output z:



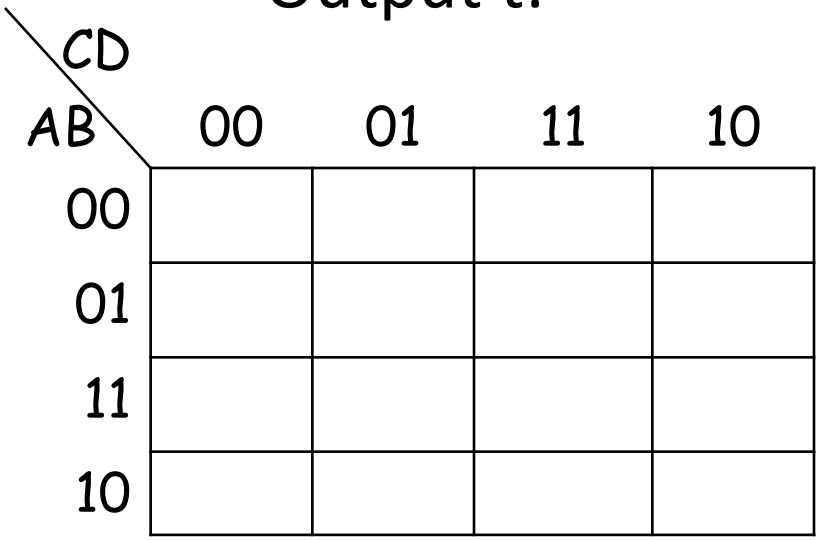
A	B	C	D	y	z
0	0	0	0	0	0
0	0	0	1	0	0
0	0	1	0	0	1
0	0	1	1	0	1
0	1	0	0	1	0
0	1	0	1	0	1
0	1	1	0	1	0
0	1	1	1	1	0
1	0	0	0	1	1
1	0	0	1	1	1
The rest				X	X

$y = A + BD' + BC$

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

Boolean Expressions for Outputs

- Output t:



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

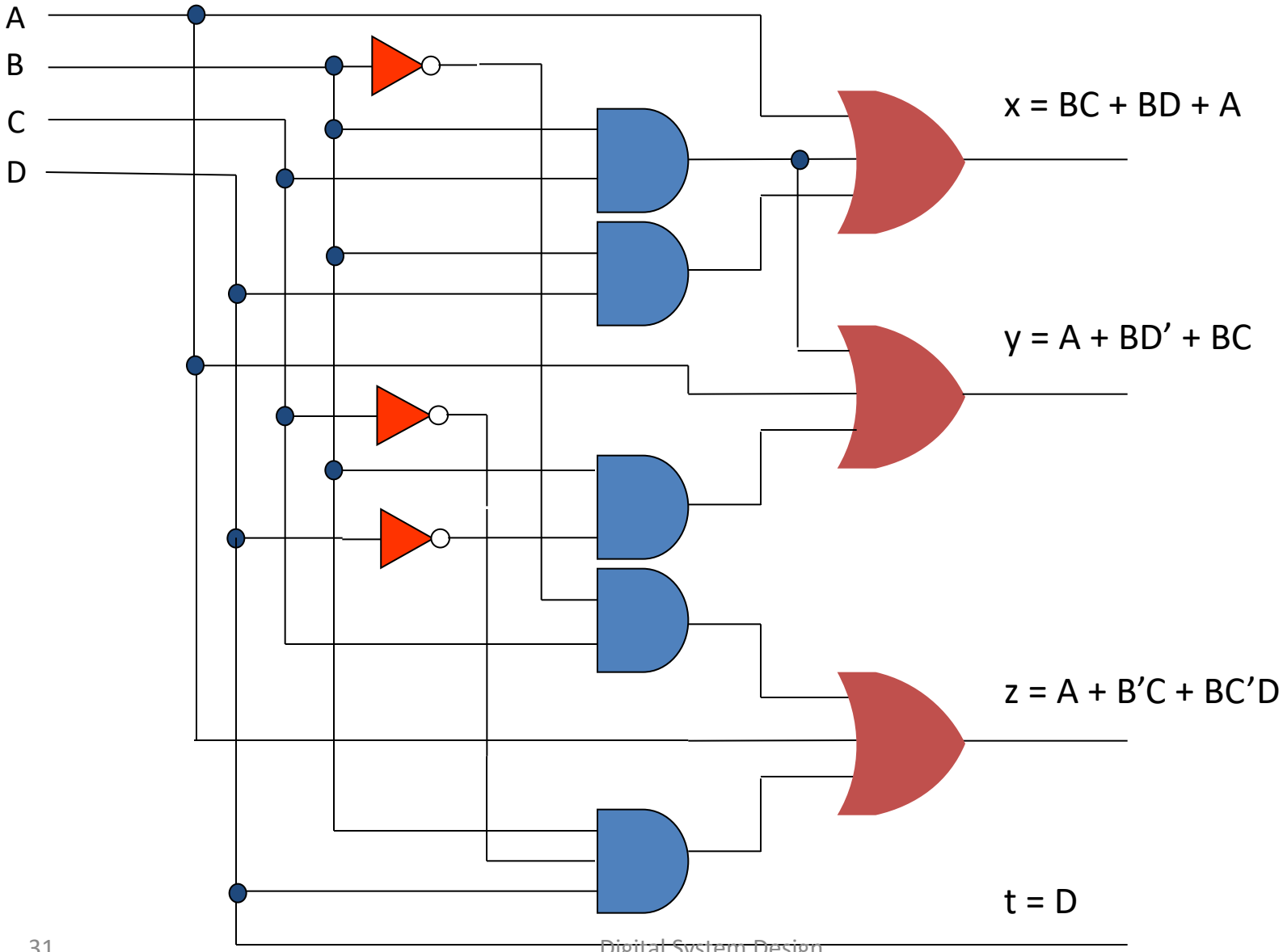
- Step 4: Draw the logic diagram

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 Adders

- Addition is a vital function in computer systems
- What does an adder do?
  - Add binary digits
  - Generate carry if necessary
  - Consider carry from previous digit
- Binary adders operate bit-wise
  - A 16-bit adder uses 16 one-bit adders
- Binary adders come in two flavors
  - Half adder : adds two bits, generates result and carry
  - Full adder : also considers carry input
  - 2 Half Adders + OR : make 1 Full Adder

Binary Half Adder

- Specification:
  - Design a circuit that adds two bits and generates the sum and a carry
- Outputs:
  - Two inputs: *x, y*
  - Two output: *S* (sum), *C* (carry)
  - 0+0=0 ; 0+1=1 ; 1+0=1 ; 1+1=10

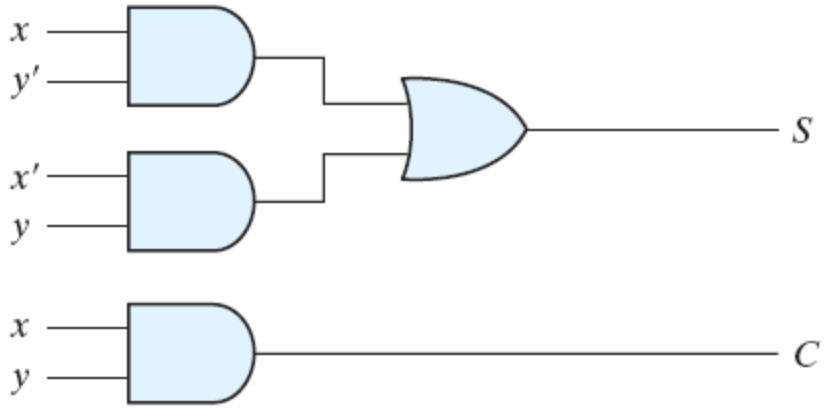
Table 4.3
Half Adder

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

- The S output represents the least significant bit of the sum.
- The C output represents the most significant bit of the sum or (a carry).

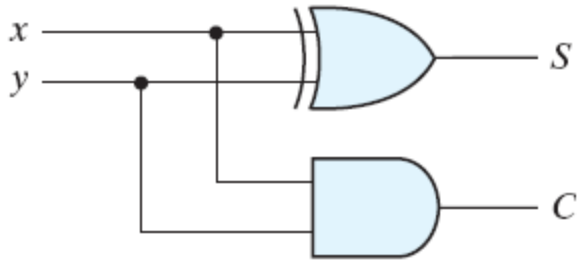
Implementation of Half Adder

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



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

- S = x'y+xy'
- C = xy



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

Full-Adder

- Specification:
  - A combinational circuit that forms the arithmetic sum of three bits and generates a sum and a carry
- Inputs:
  - Three inputs: *x,y,z*
  - Two outputs: *S, C*
- Truth table:

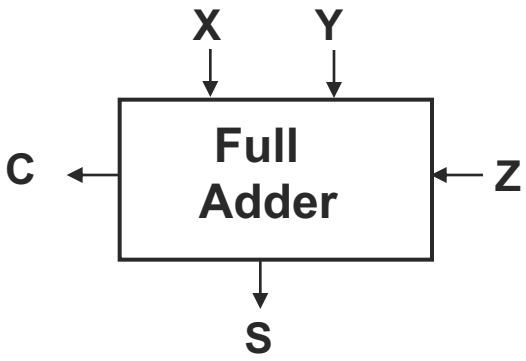
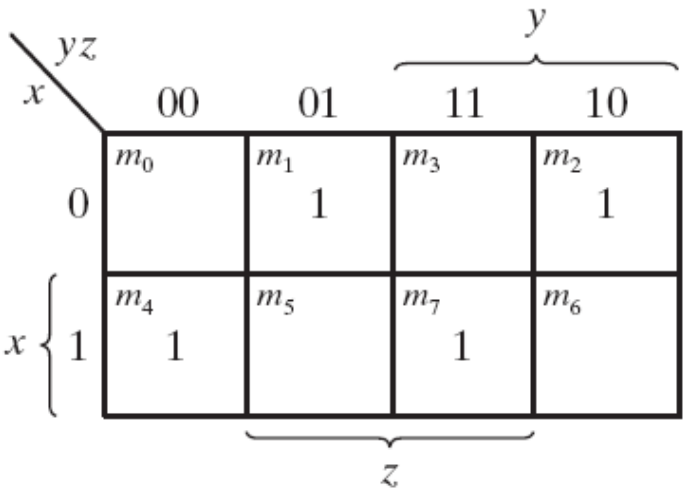
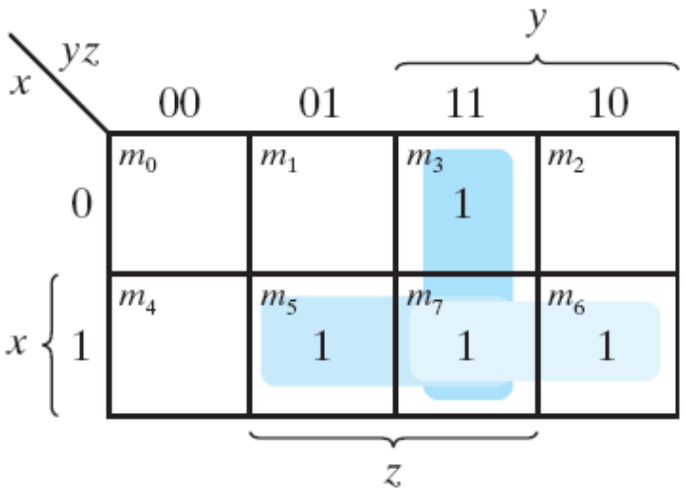


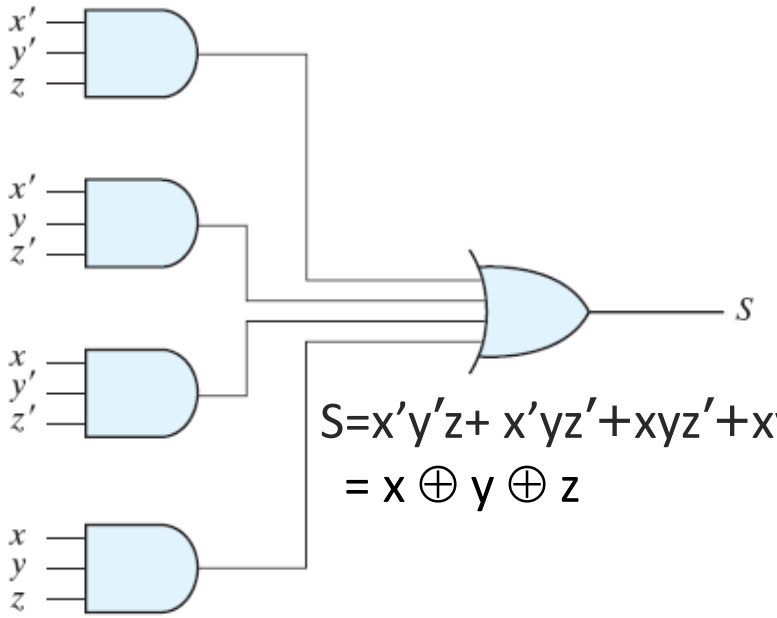
Table 4.4
Full Adder

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

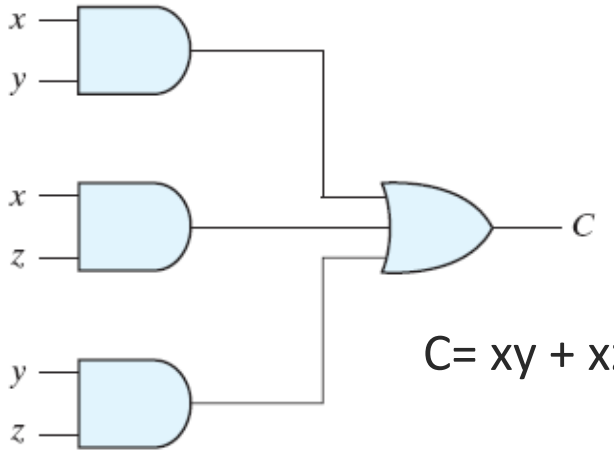
Implementation of Full Adder







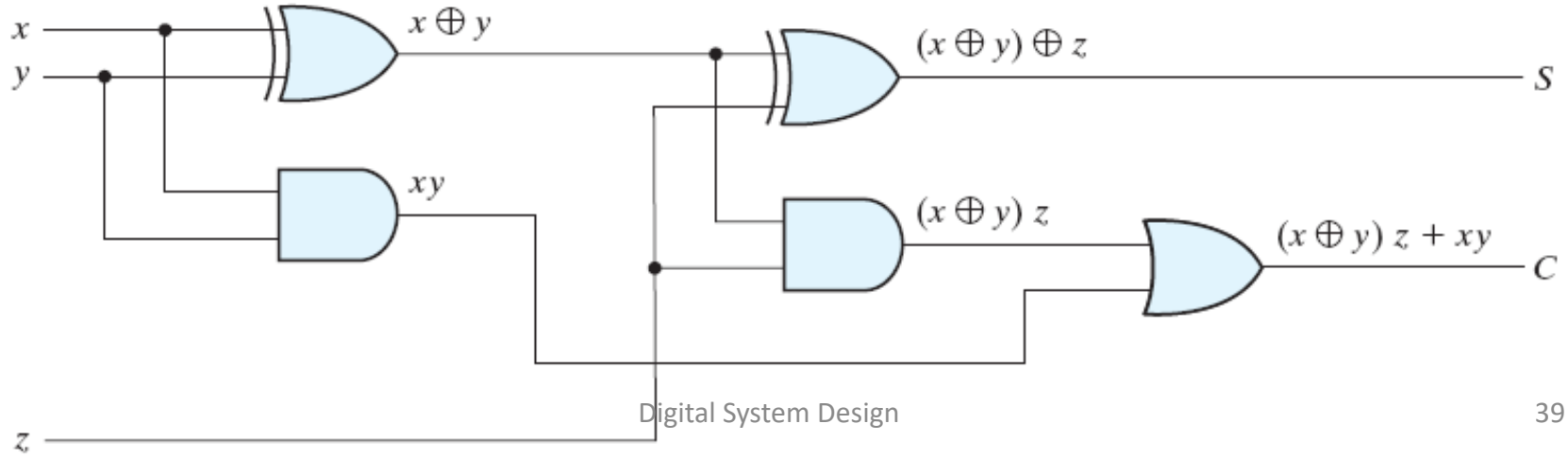
S=x'y'z+ x'yz'+xyz'+xyz
= x ⊕ y ⊕ z



C= xy + xz + yz

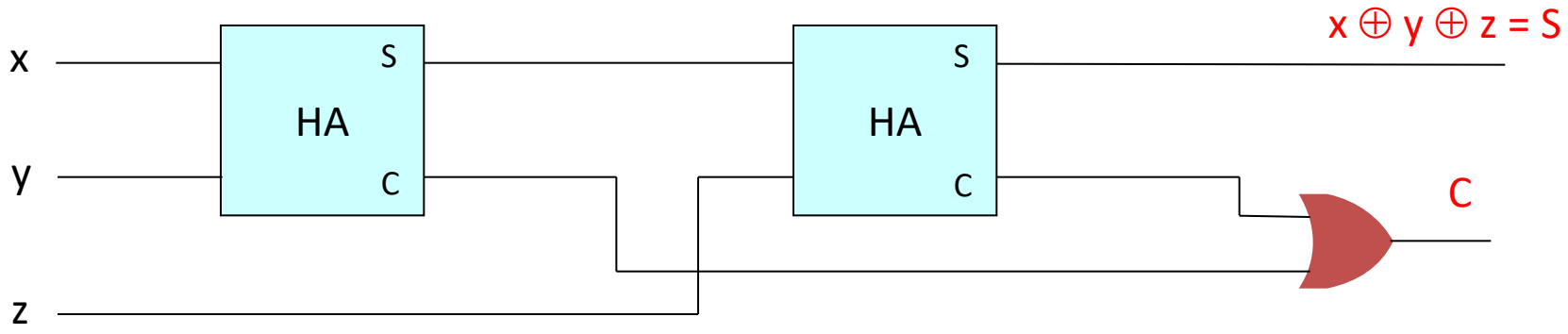
Full Adder: Hierarchical Realization

- 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 2 half adders and 1 OR gate.



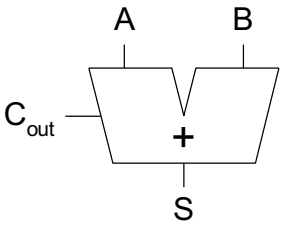
Full Adder: Hierarchical Realization

- 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 2 half adders and 1 OR gate.



Symbols for 1-Bit Adders

Half Adder

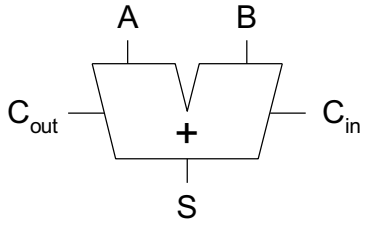


A	B	C_{out}	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

$S = A \oplus B$

$C_{out} = AB$

Full Adder



C_{in}	A	B	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

$S = A \oplus B \oplus C_{in}$

$C_{out} = AB + AC_{in} + BC_{in}$

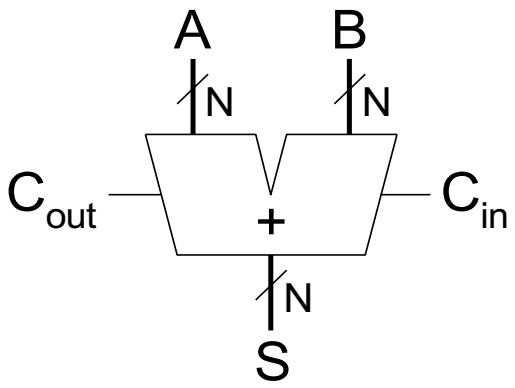
Binary Adder

- A binary adder is a digital circuit that produces the arithmetic sum of two binary numbers.
- A binary adder can be implemented using multiple full adders (FAs).

Multibit Adders (CPAs)

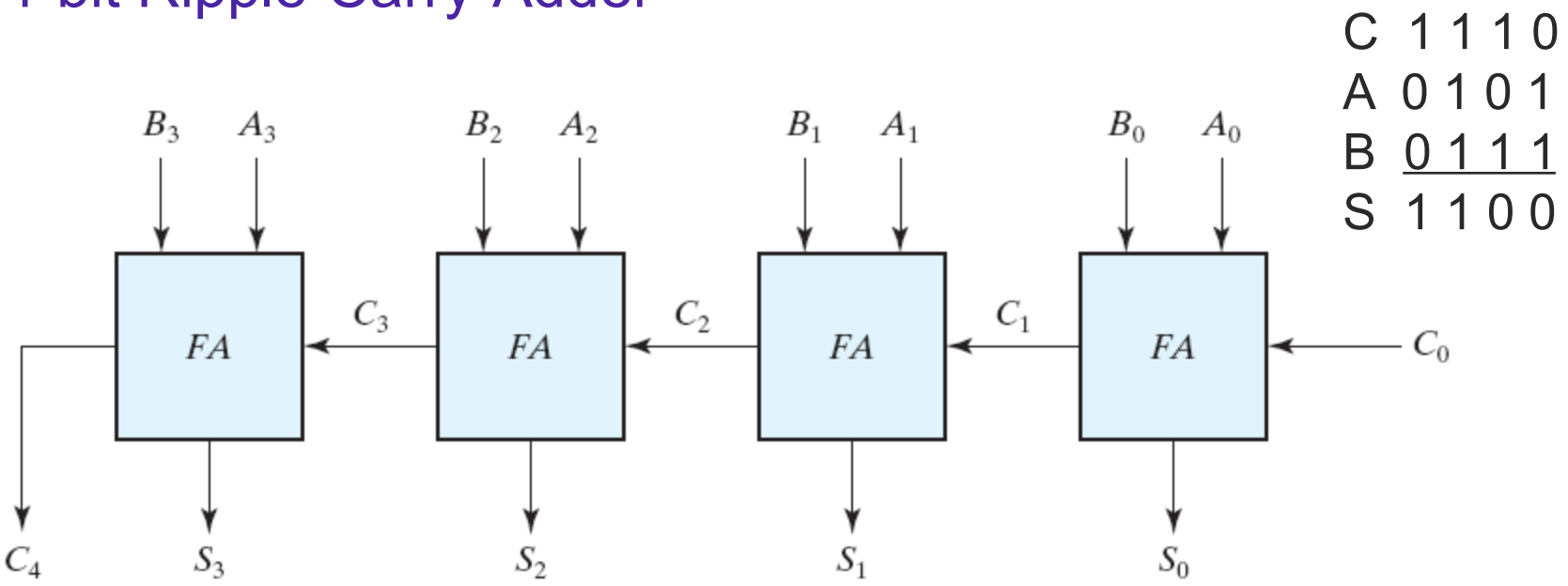
- Realized by carry propagate adders (CPAs)
- Types of CPAs
  - Ripple-carry (slow)
  - Carry-lookahead (fast)
  - Prefix (faster)
- Carry-lookahead and prefix adders faster for large adders but require more hardware

Symbol



Example:4-bit binary adder

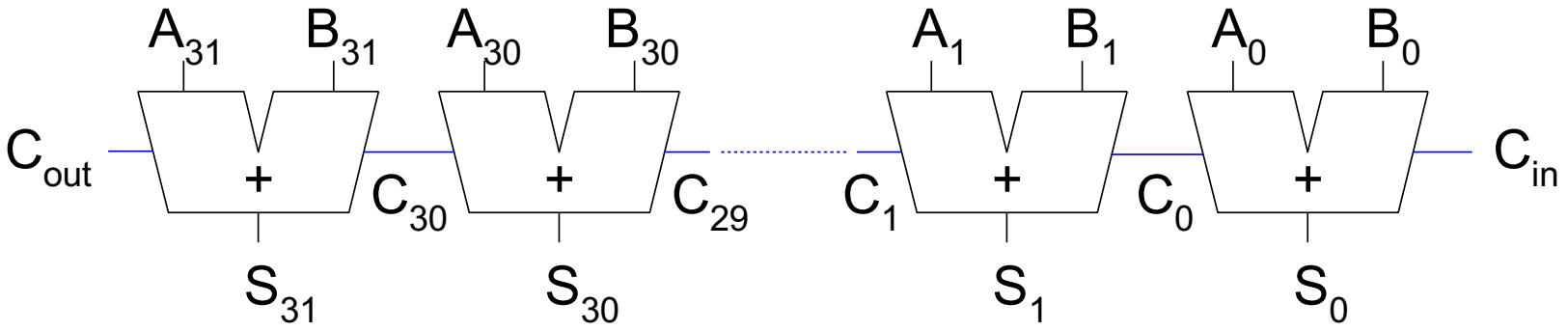
- 4-bit Ripple Carry Adder



- Classical example of standard components
  - Would require truth table with 2^9 entries!

Ripple-Carry Adder

- Chain 1-bit adders together
- Carry ripples through entire chain
- Disadvantage: **slow**



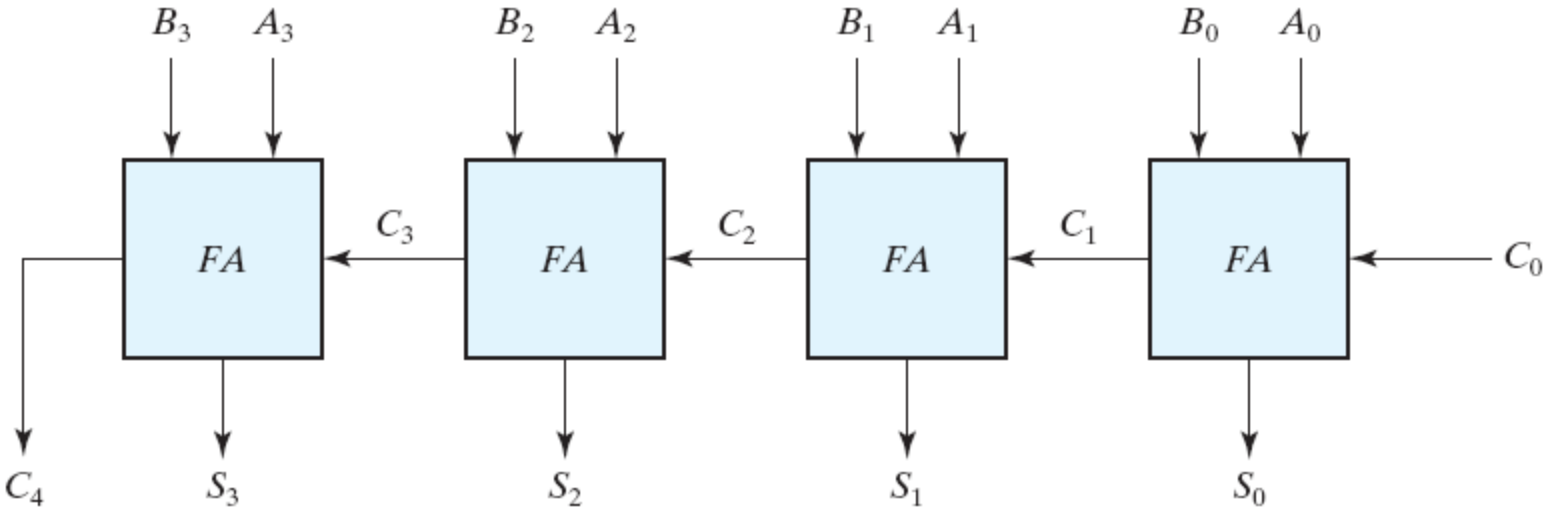
Hierarchical Design Methodology

- The design methodology we used to build ripple-carry adder is a 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 larger functionality

Carry Propagation

- In any combinational circuit, the signal must propagate through the gates before the correct output is available in the output terminals.
- Total propagation time =

  (the propagation delay of a typical gate) *x* (the number of gate levels)
- The longest propagation delay time in a binary adder is the time it takes the carry to propagate through the full adders. This is because each bit of the sum output depends on the value of the input carry. This makes the binary adder very slow.



n-bit Carry Ripple Adders

- In the expression of the sum C_j must be generated by the full adder at the lower position j-1.
- The propagation delay in each full adder to produce the carry is equal to two gate delays = $2\, t_{\text{AND/OR}} = t_{FA}$
- Since the generation of the sum requires the propagation of the carry from the lowest position to the highest position, the total propagation delay of the adder is approximately:

$$\text{Total Propagation delay} \sim 2n\, t_{\text{AND/OR}} = n\, t_{FA}$$

4-bit Carry Ripple Adder

Adds two 4-bit numbers:

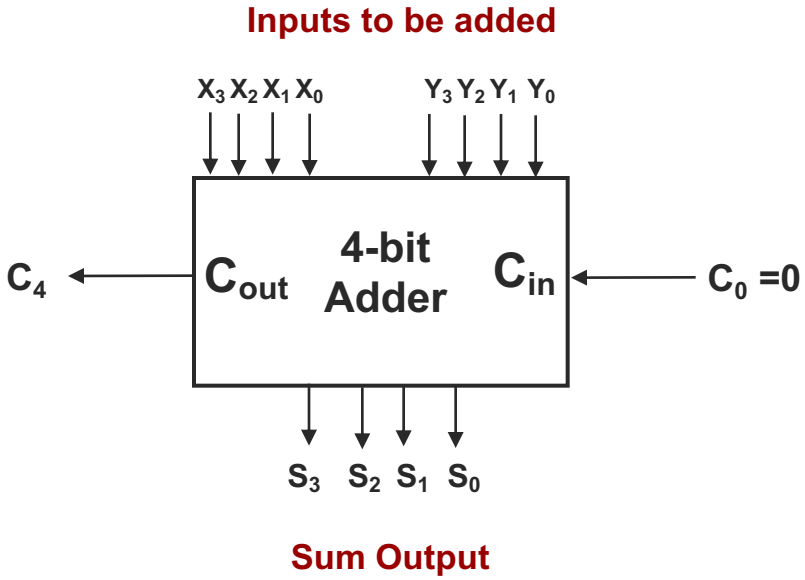
$X = X_3\ X_2\ X_1\ X_0$

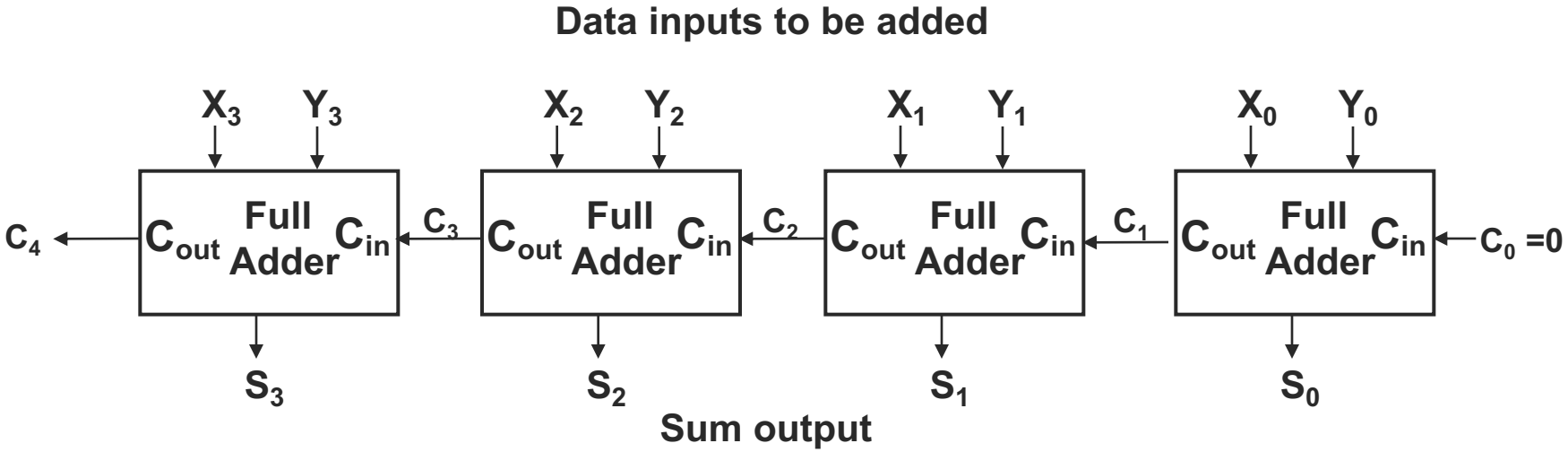
$Y = Y_3\ Y_2\ Y_1\ Y_0$

producing the sum $S = S_3\ S_2\ S_1\ S_0$, $C_{out} = C_4$ from the most significant position j=3

Total Propagation delay : 4 t_{FA}

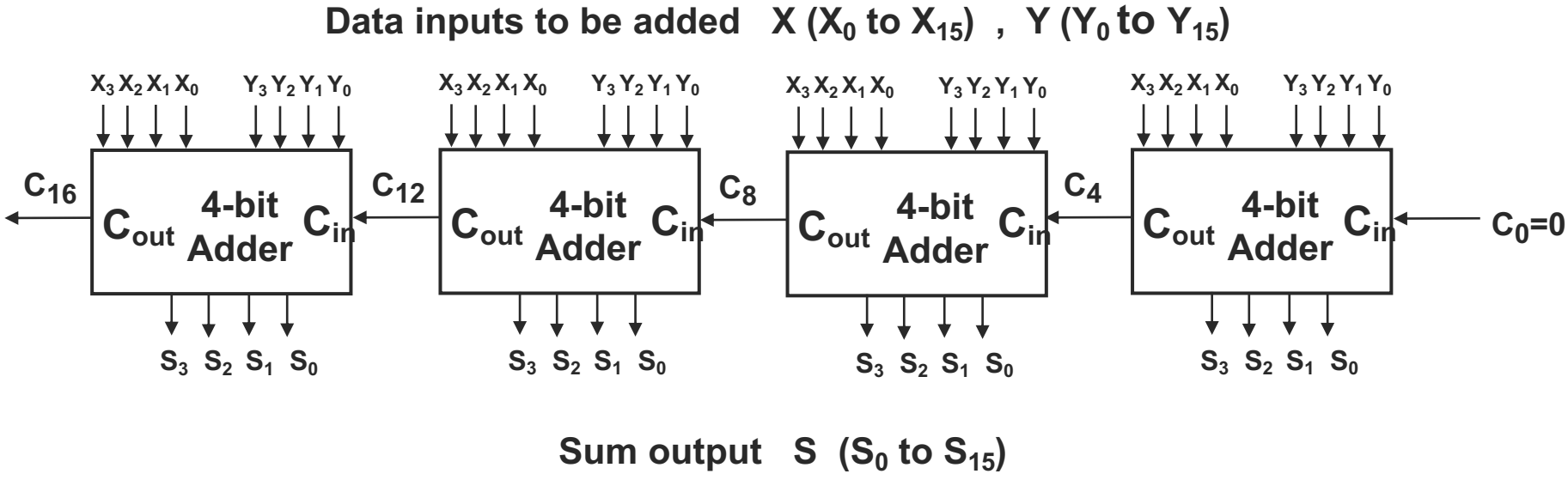
or 8 $t_{AND/OR}$ (8 gate delays)





Larger Adders

- Example: 16-bit adder using 4 * 4-bit adders
- Adds two 16-bit inputs X (bits X_0 to X_{15}), Y (bits Y_0 to Y_{15}) producing a 16-bit Sum S (bits S_0 to S_{15}) and a carry out C16 from most significant position.



Propagation delay for 16-bit adder = 4 x propagation delay of 4-bit adder
~ 16 t_{FA} or 32 gate delays

Carry-Lookahead Adder (CLA)

- Compute carry out (C_{out}) for k-bit blocks using *generate* and *propagate* signals
- **Some definitions:**
  - Column i produces a carry out by either *generating* a carry out or *propagating* a carry in to the carry out
  - Generate (G_i) and propagate (P_i) signals for each column:
    - Column i will generate a carry out if A_i AND B_i are both 1.

$$G_i = A_i B_i$$

    - Column i will propagate a carry in to the carry out if A_i XOR B_i is 1.

$$P_i = A_i \oplus B_i$$

    - The carry out of column i (C_i) is:

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

Carry-Lookahead Adder

- Full adder: $S_i = A_i \oplus B_i \oplus C_i$, $C_{i+1} = A_i B_i + (A_i \oplus B_i) C_i$
- Create *new signals:*
  - $G_i = A_i B_i$ "carry generate" for stage *i*
  - $P_i = A_i \oplus B_i$ "carry propagate" for stage *i*
- Full adder equations expressed in terms of G_i and P_i
  - $S_i = P_i \oplus C_i$
  - $C_{i+1} = G_i + P_i C_i$

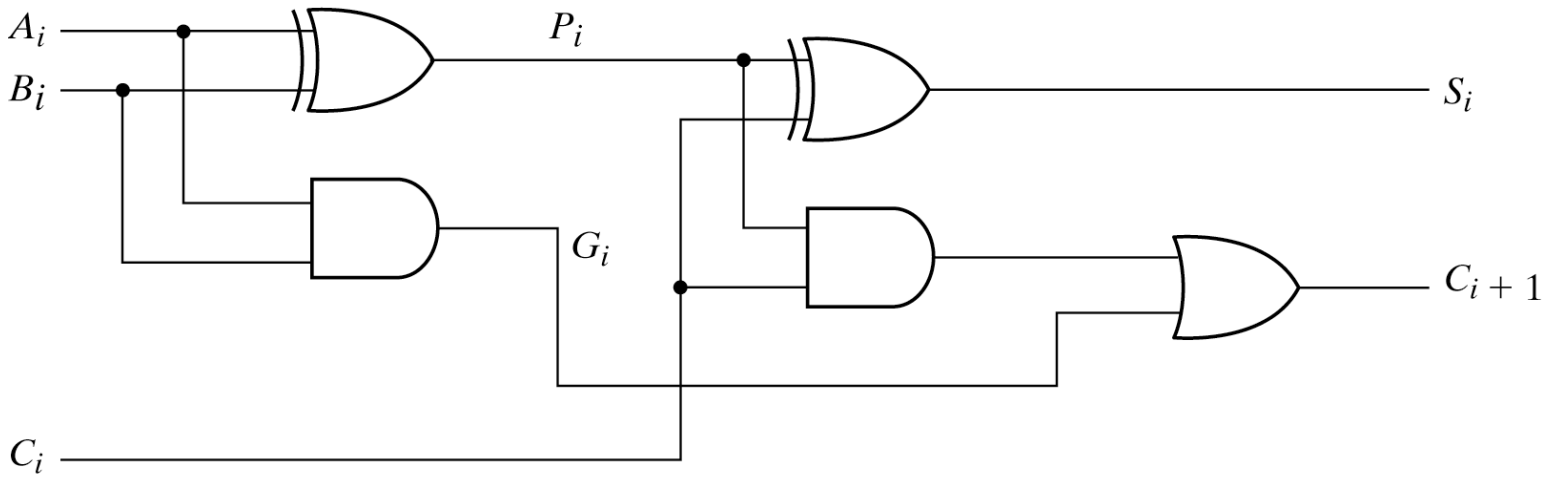


Fig. 4-10 Full Adder with P and G Shown

Carry Lookahead - Equations

- Full adder functionality can be expressed recursively
  - $S_i = P_i \oplus C_i$
  - $C_{i+1} = G_i + P_i C_i$
- Carry of each stage
  - $C_0 = input\ carry$
  - $C_1 = G_0 + P_0C_0$
  - $C_2 = G_1 + P_1C_1 = G_1 + P_1(G_0 + P_0C_0) = G_1 + P_1G_0 + P_1P_0C_0$
  - $C_3 = G_2 + P_2C_2 = \ldots = G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0C_0$
  - $C_4 = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0C_0$

Carry Lookahead - Circuit

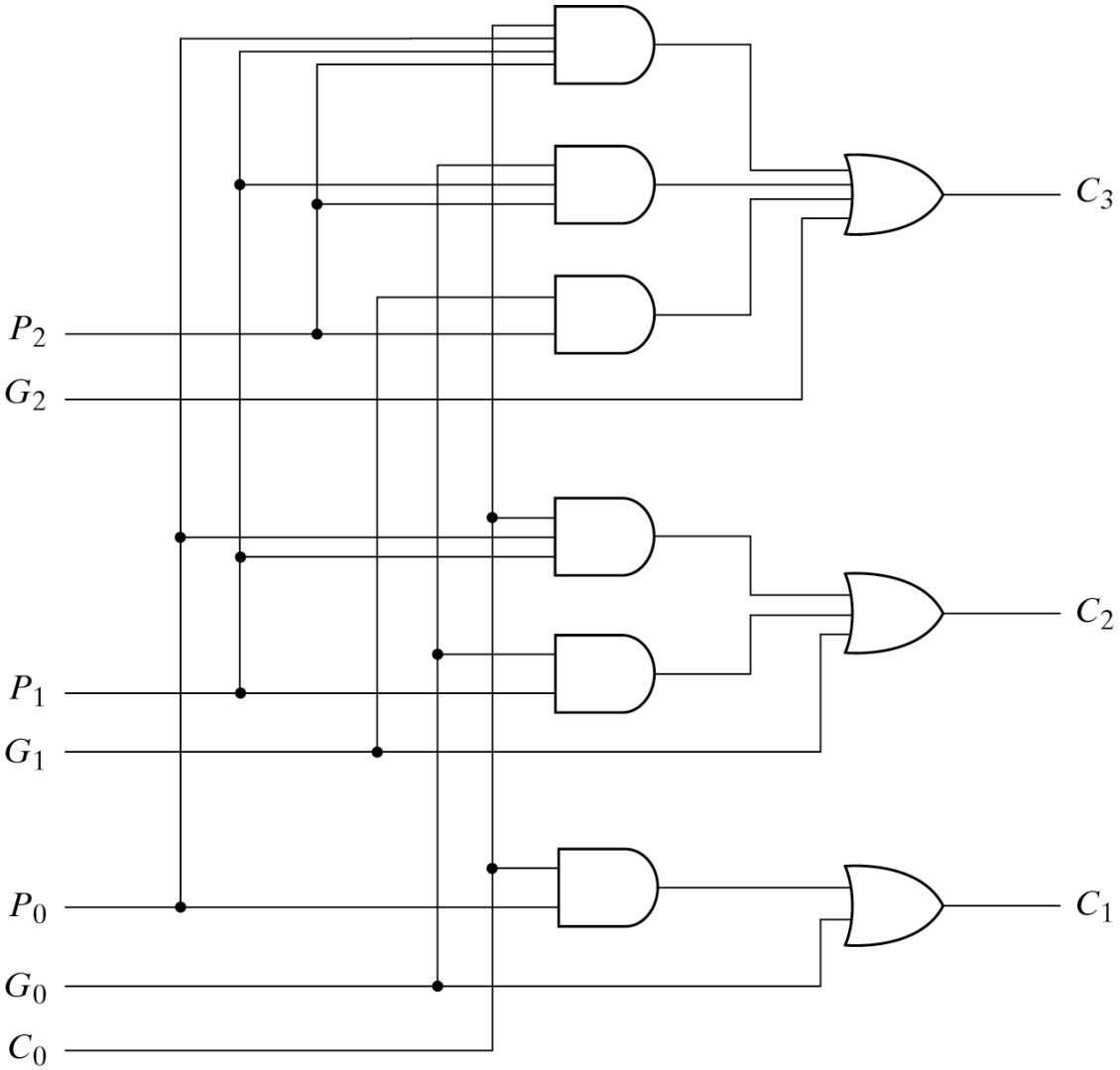
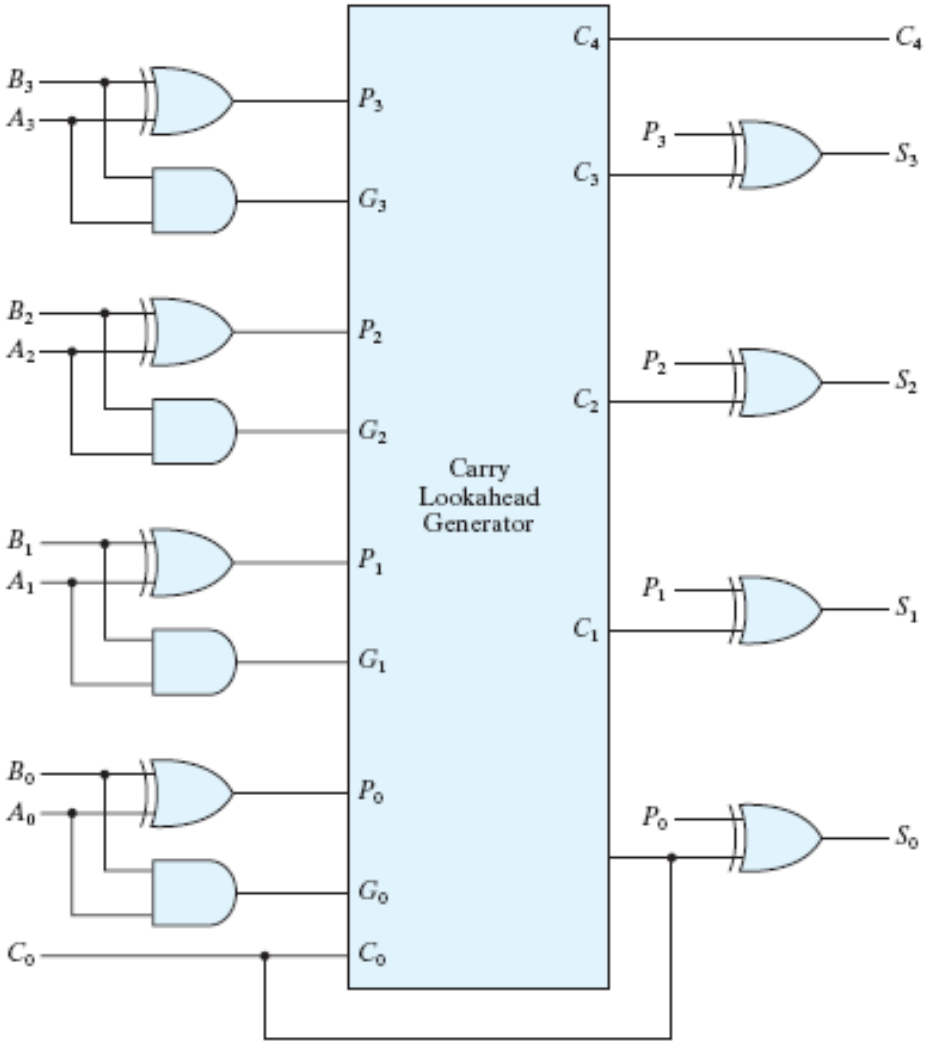


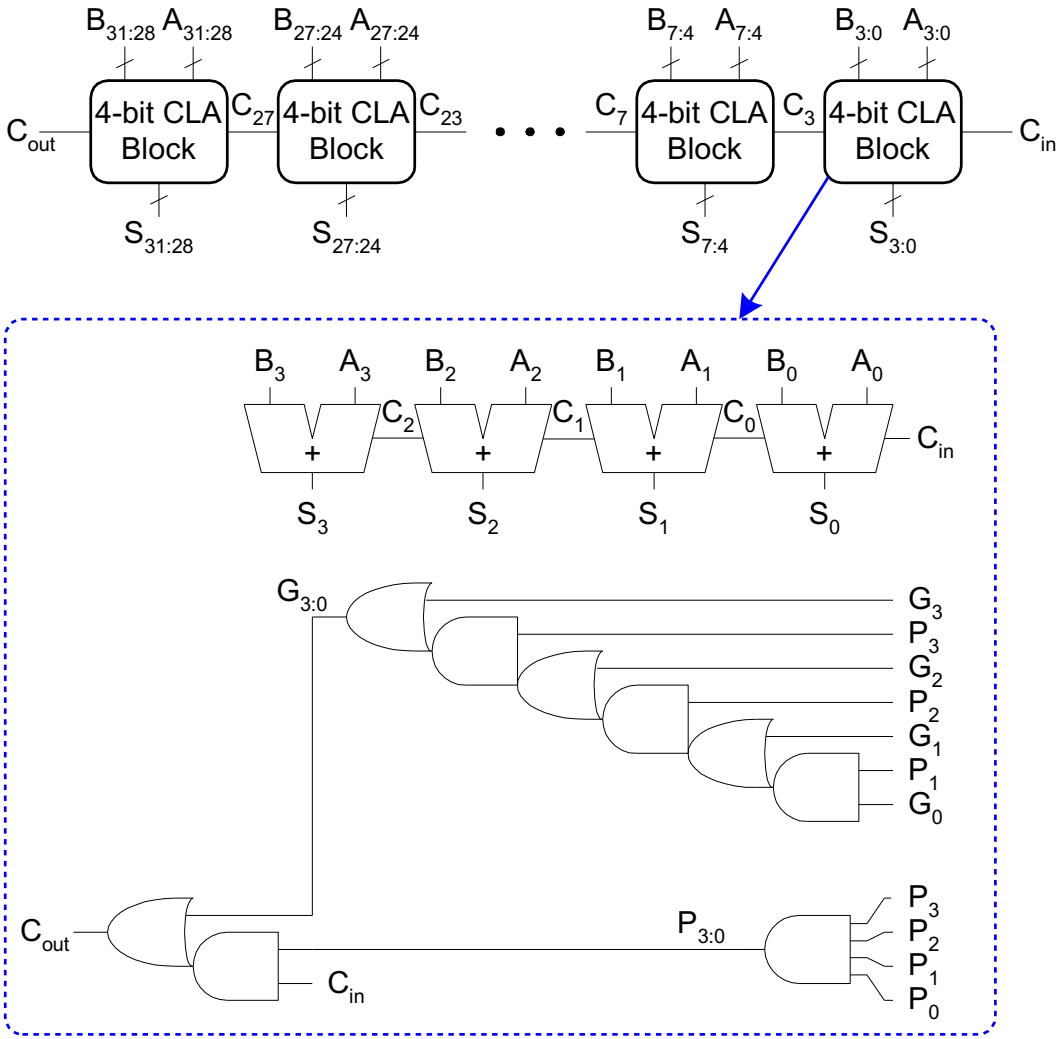
Fig. 4-11 Logic Diagram of Carry Lookahead Generator

4-bit Adder with Carry Lookahead

- Complete adder:
  - Same number of stages for each bit
- Drawback?
  - Increasing complexity of lookahead logic for more bits



32-bit CLA with 4-bit Blocks



Carry-Lookahead Adder Delay

For N-bit CLA with k-bit blocks:

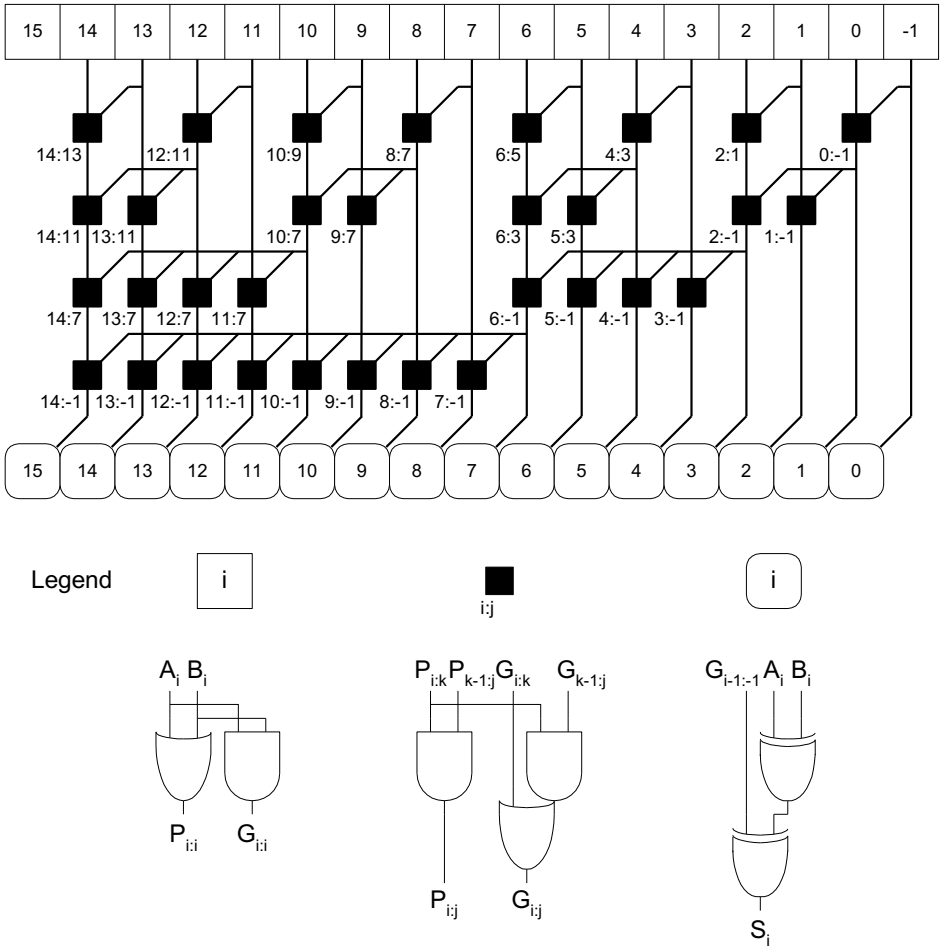
$$t_{CLA} = t_{pg} + t_{pg\_block} + (N/k - 1)t_{AND\_OR} + kt_{FA}$$

- t_{pg} : delay to generate all P_i, G_i
- t_{pg_block} : delay to generate all $P_{i:j}$, $G_{i:j}$
- t_{AND_OR} : delay from C_{in} to C_{out} of final AND/OR gate in k-bit CLA block

An N-bit carry-lookahead adder is generally much faster than a ripple-carry adder for $N > 16$

Prefix Adder Schematic

- A prefix adder works by dividing the input numbers into smaller chunks, typically 2 bits each, and then processes these chunks in parallel to compute the sum and carry outputs.
- It uses a tree-like structure, where each level of the tree performs a set of carry-lookahead calculations.
  - The carry signals are pre-computed and propagated through the tree structure, reducing the time needed to determine the final carry and sum outputs.
  - This parallel processing of carry signals and intermediate sums speeds up the addition process considerably.
- There are variations such as Brent-Kung and Kogge-Stone adders, each with its own trade-offs and optimizations.



Prefix Adder Delay

$$t_{PA} = t_{pg} + \log_2 N(t_{pg\_prefix}) + t_{XOR}$$

- t_{pg}: delay to produce P_i G_i (AND or OR gate)
- t_{pg_prefix}: delay of black prefix cell (AND-OR gate)

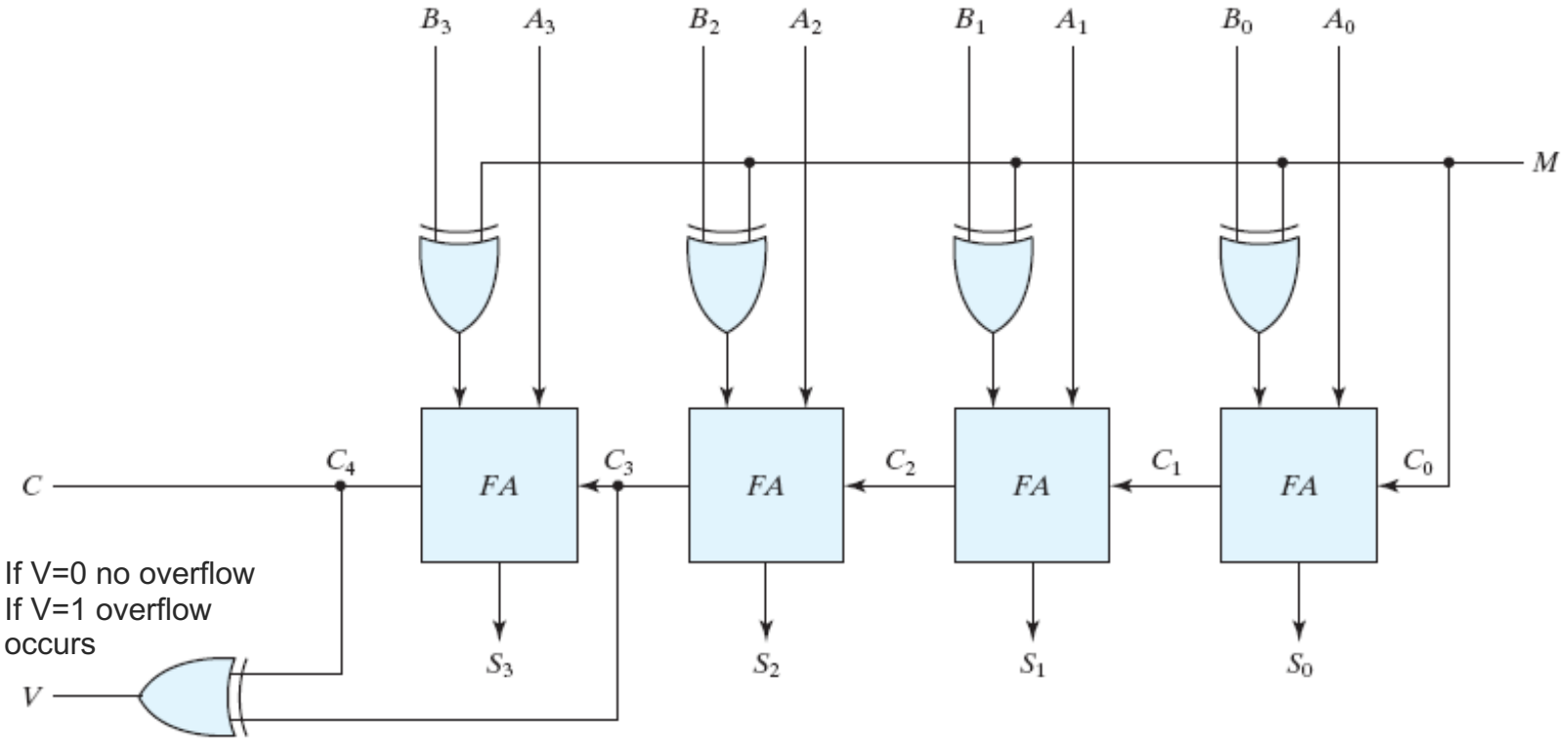
Adder Delay Comparisons

Compare delay of: 32-bit ripple-carry, carry-lookahead, and prefix adders

- CLA has 4-bit blocks
- 2-input gate delay = 100 ps; full adder delay = 300 ps

$$
\begin{aligned}
t_{\text{ripple}} &= Nt_{FA} = 32(300\text{ ps}) \\
&= \mathbf{9.6\text{ ns}} \\
t_{CLA} &= t_{pg} + t_{pg\_\text{block}} + (N/k - 1)t_{\text{AND\_OR}} + kt_{FA} \\
&= [100 + 600 + (7)200 + 4(300)]\text{ ps} \\
&= \mathbf{3.3\text{ ns}} \\
t_{PA} &= t_{pg} + \log_2 N(t_{pg\_\text{prefix}}) + t_{\text{XOR}} \\
&= [100 + \log_2 32(200) + 100]\text{ ps} \\
&= \mathbf{1.2\text{ ns}}
\end{aligned}
$$

Four-bit adder-subtractor



- Recall how we do subtraction: Using complements
  $X - Y = X + (2^n - Y) = X + \sim Y + 1$
- M sets *mode*: *M=0* addition and *M=1* subtraction
- *M* is a "control signal" (not "data") switching between Add and Subtract

Overflow Conditions

- Overflow conditions
  - There is no overflow if signs are different (*pos + neg*, or *neg + pos*)
  - Overflow can happen only when
    - both numbers have the same sign but carry *into* sign position and *out* of sign position differ
  - Example: 2's complement signed numbers wih $n = 4$ bits

+6	0 110	-6	1 010
+7	0 111	-7	1 001
+13	0 1 101	-13	1 0 011

  - Result would be correct with extra position
  - Detected by XOR gate (output =1 when inputs differ)
  - Can be used as input carry for next adder circuit

Overflow Conditions

INPUTS			OUTPUTS		
A_{sign}	B_{sign}	CARRY IN	CARRY OUT	SUM_{sign}	OVERFLOW
0	0	0	0	0	0
0	0	1	0	1	1
0	1	0	0	1	0
0	1	1	1	0	0
1	0	0	0	1	0
1	0	1	1	0	0
1	1	0	1	0	1
1	1	1	1	1	0

Notice that **overflow** occurs only when

$$CARRY_{in} \neq CARRY_{out}$$

or simply

$$V = C_{in} \text{ XOR } C_{out}$$

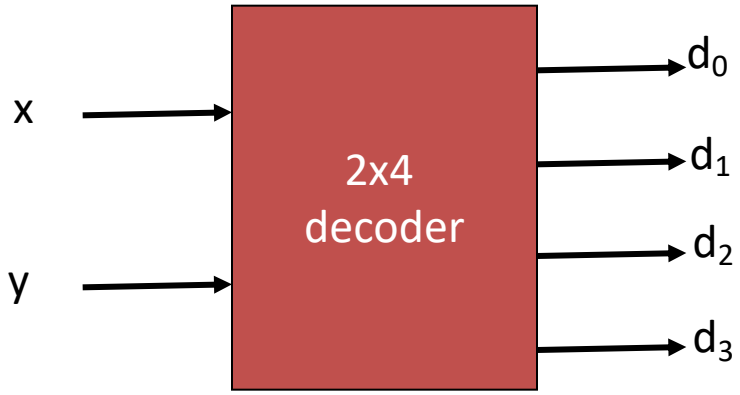
where **V** is the overflow signal.

Addition cases and overflow

00	01	11	10	00	11
0010	0011	1110	1101	0010	1110
0011	0110	1101	1010	1100	0100
0101	1001	1011	0111	1110	0010
2	3	-2	-3	2	-2
3	6	-3	-6	-4	4
5	-7	-5	7	-2	2
	OFL		OFL		

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

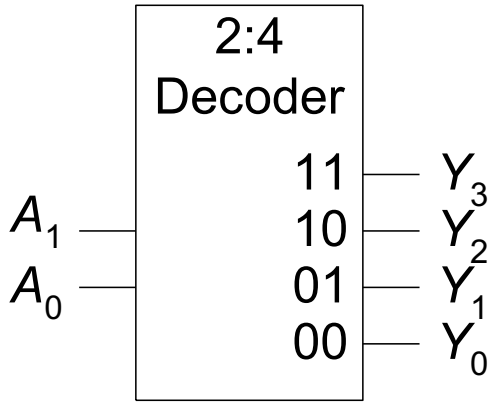


x	y	d_0	d_1	d_2	d_3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

- d_0 =
- d_1 =
- d_2 =
- d_3 =

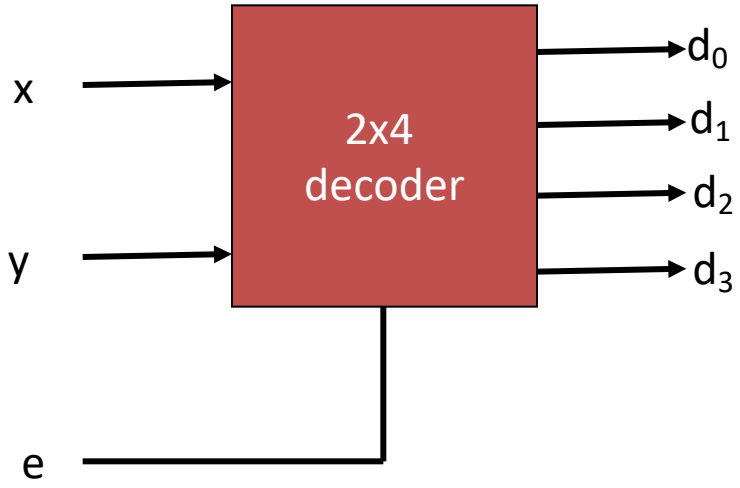
Decoders

- N inputs, 2^N outputs
- One-hot outputs: only one output HIGH at once



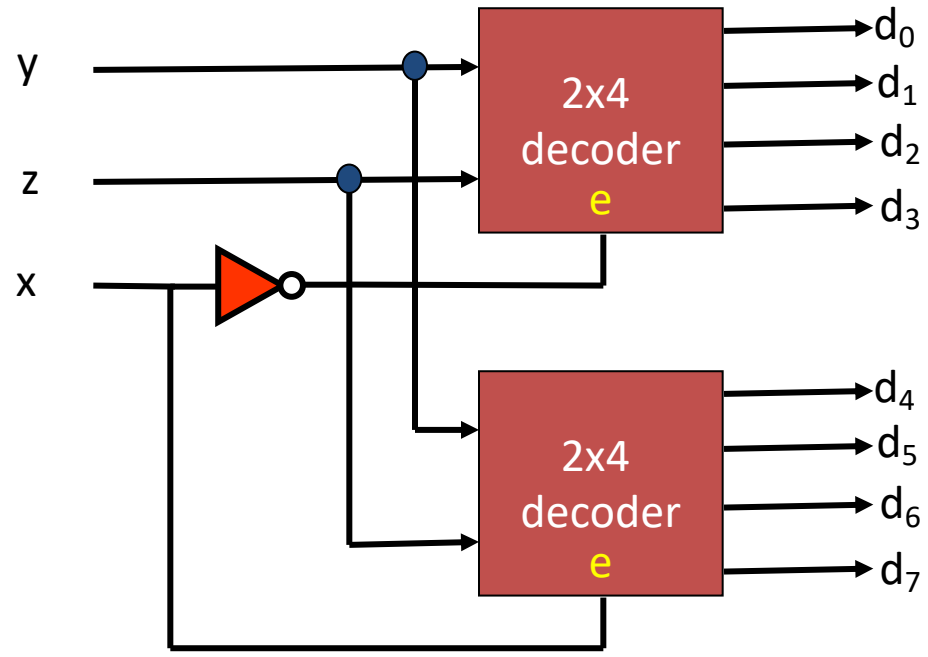
A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

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

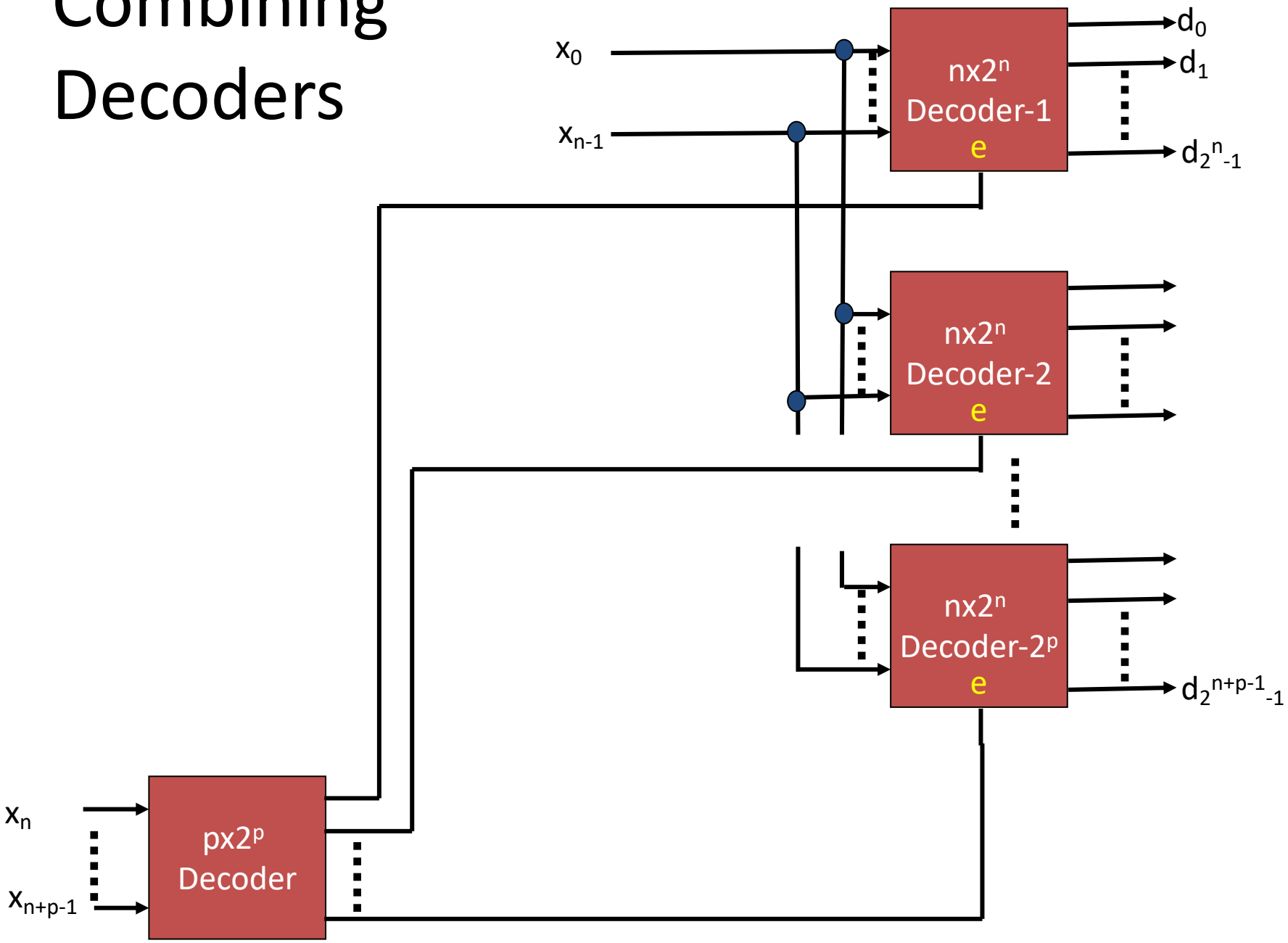
Combining Decoders



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	

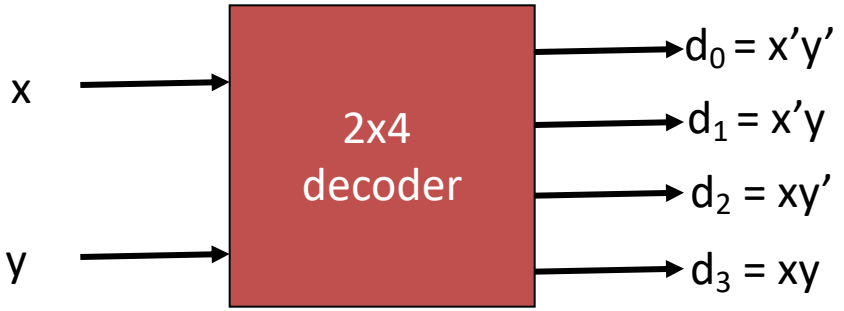
- Decoders with enable inputs can be connected together to form a larger decoder circuit.
- Figure shows two 2-to-4-line decoders with enable inputs connected to form a 3-to-8-line decoder.
- When *x*=0, the top decoder is enabled and the other is disabled. The bottom decoder outputs are all 0's, and 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

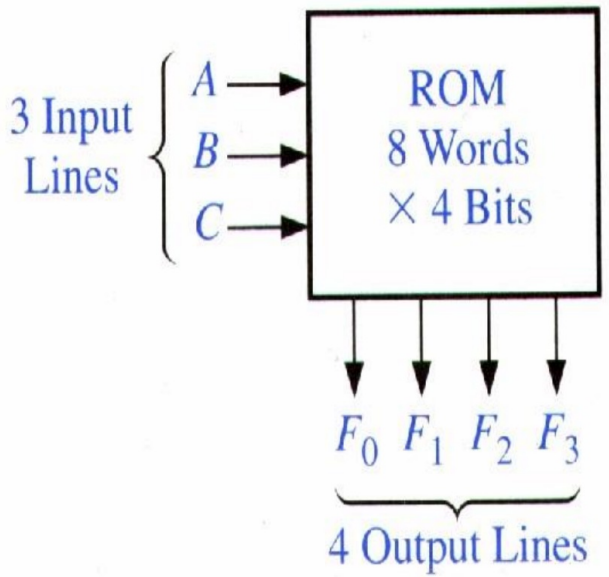
- A decoder provides the 2^n **minterms** of n input variables



- 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**.

ROM

ROM: Read-only memory: stored data can not be changed under normal operating conditions.



(a) block diagram

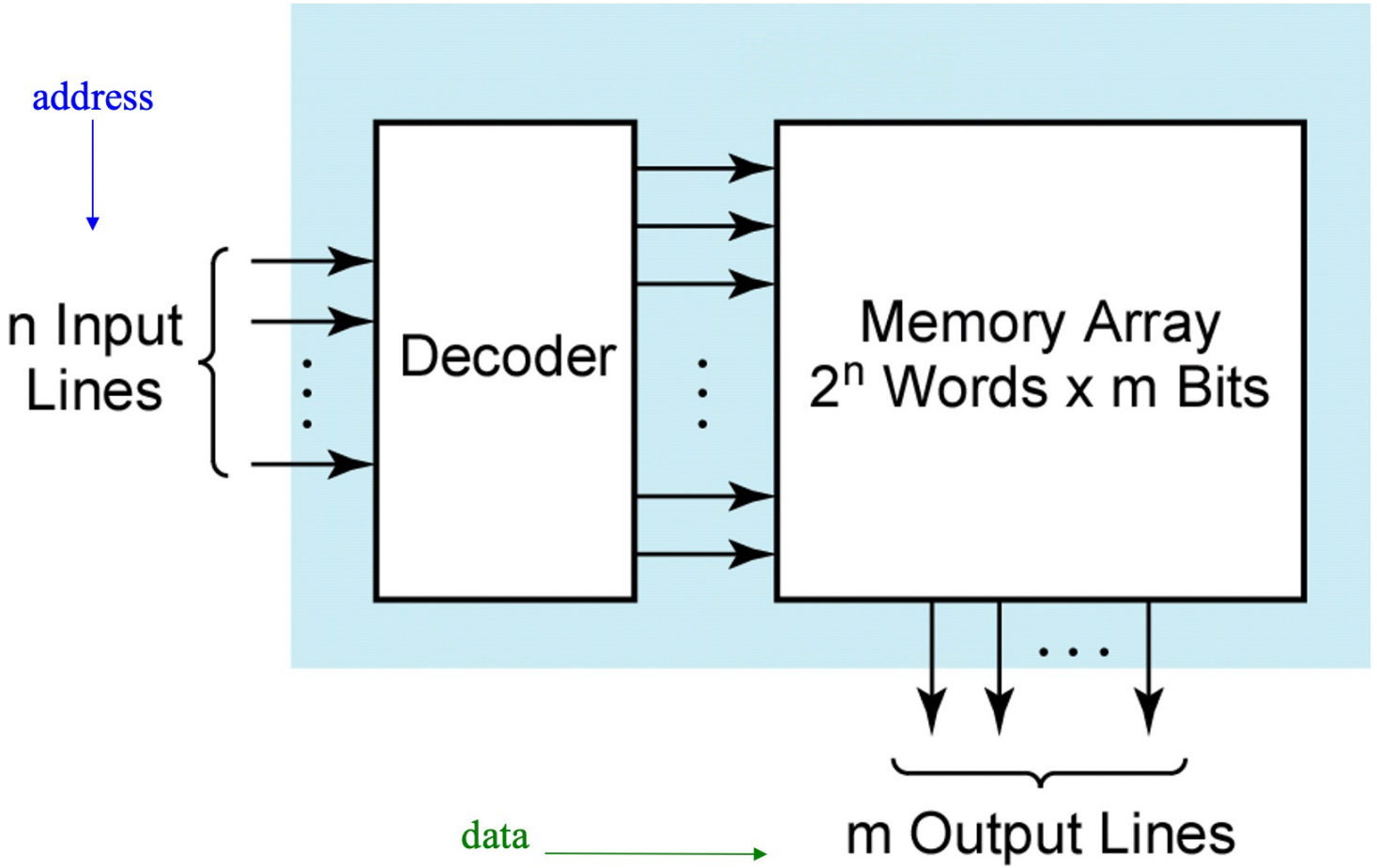
A B C	F0 F1 F2 F3
0 0 0	1 0 1 0
0 0 1	1 0 1 0
0 1 0	0 1 1 1
0 1 1	0 1 0 1
1 0 0	1 1 0 0
1 0 1	0 0 0 1
1 1 0	1 1 1 1
1 1 1	0 1 0 1

typical data stored in ROM (2^3 words of 4 bits each)

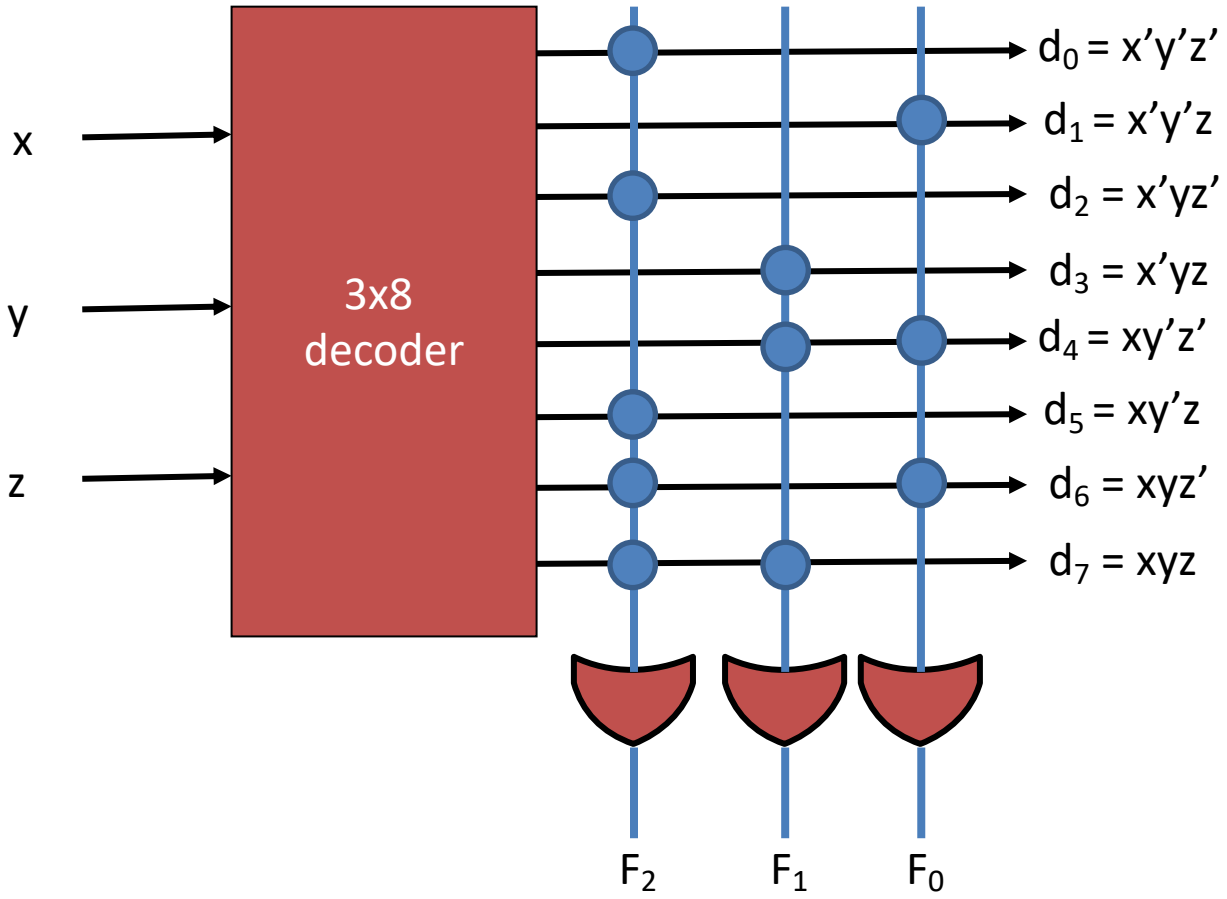
(b) truth table for ROM

ROM

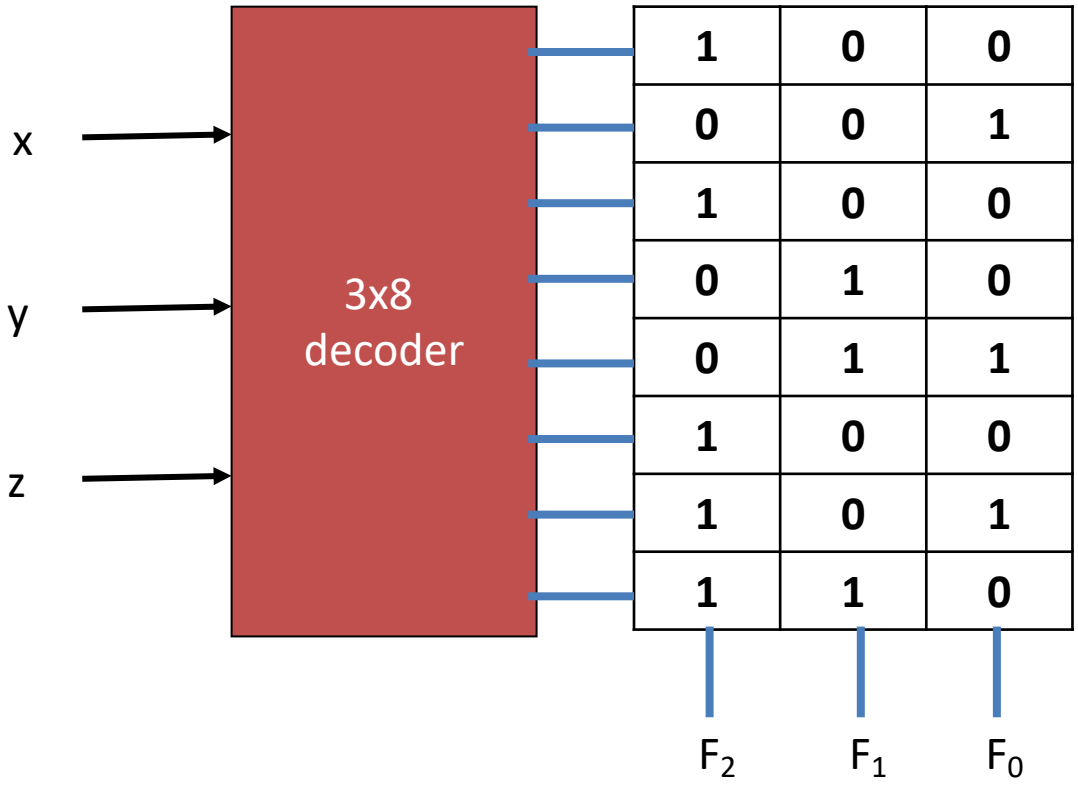
ROM: Read-only memory: stored data can not be changed under normal operating conditions.



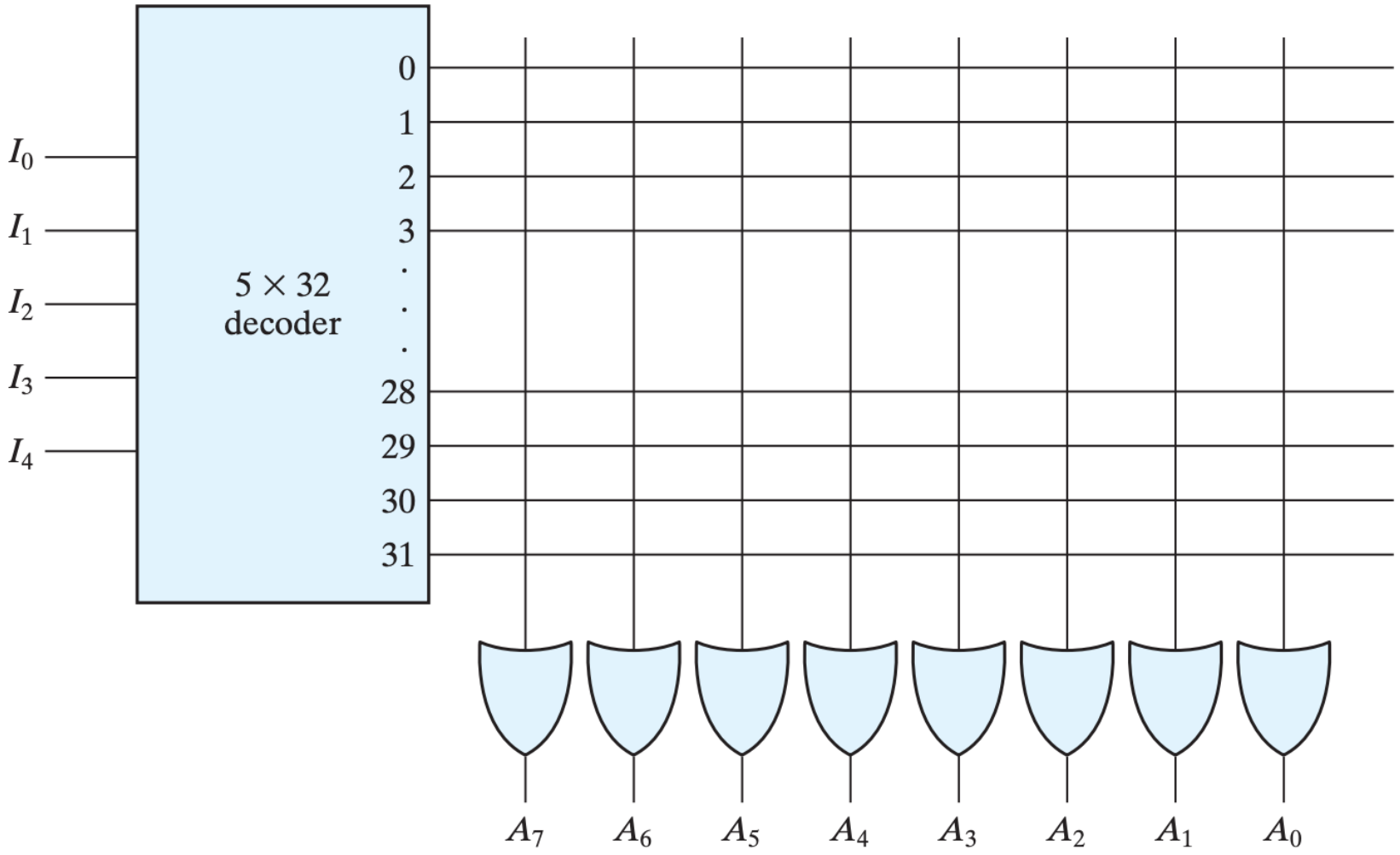
Decoder as a Building Block of ROM



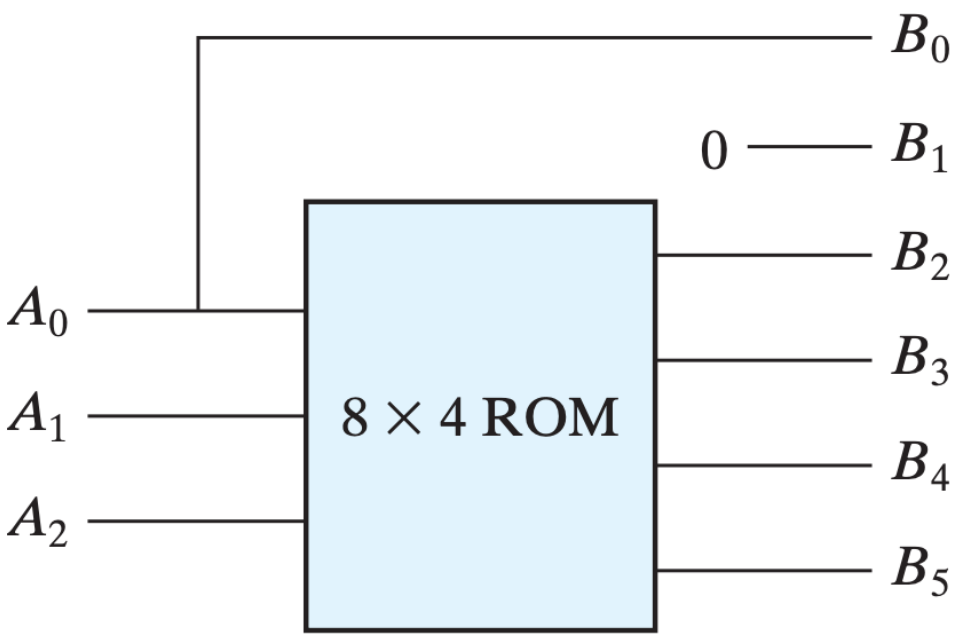
Decoder as a Building Block of ROM



32x8 ROM
(32 words of 8 bits each)



Example: ROM to give the square of a 3-bit input



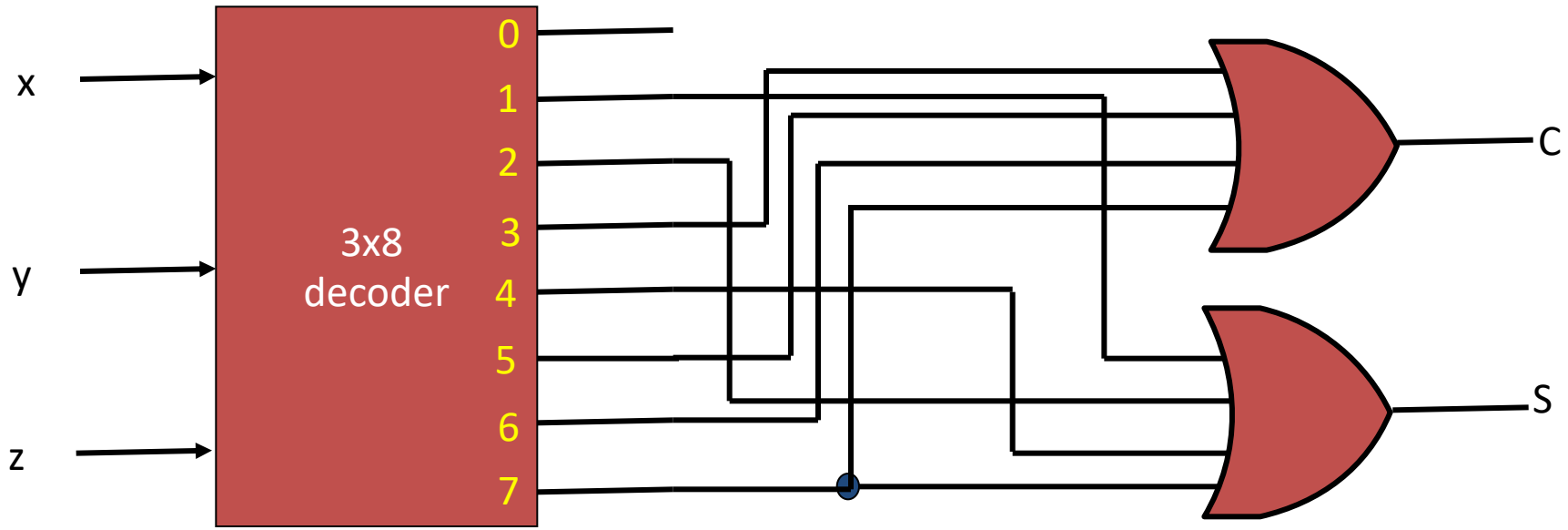
(a) Block diagram

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

(b) ROM truth table

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 an input variable 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

d_0d_1 \ d_2d_3	00	01	11	10
00				
01				
11				
10				

a =

d_0d_1 \ d_2d_3	00	01	11	10
00				
01				
11				
10				

b =

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

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

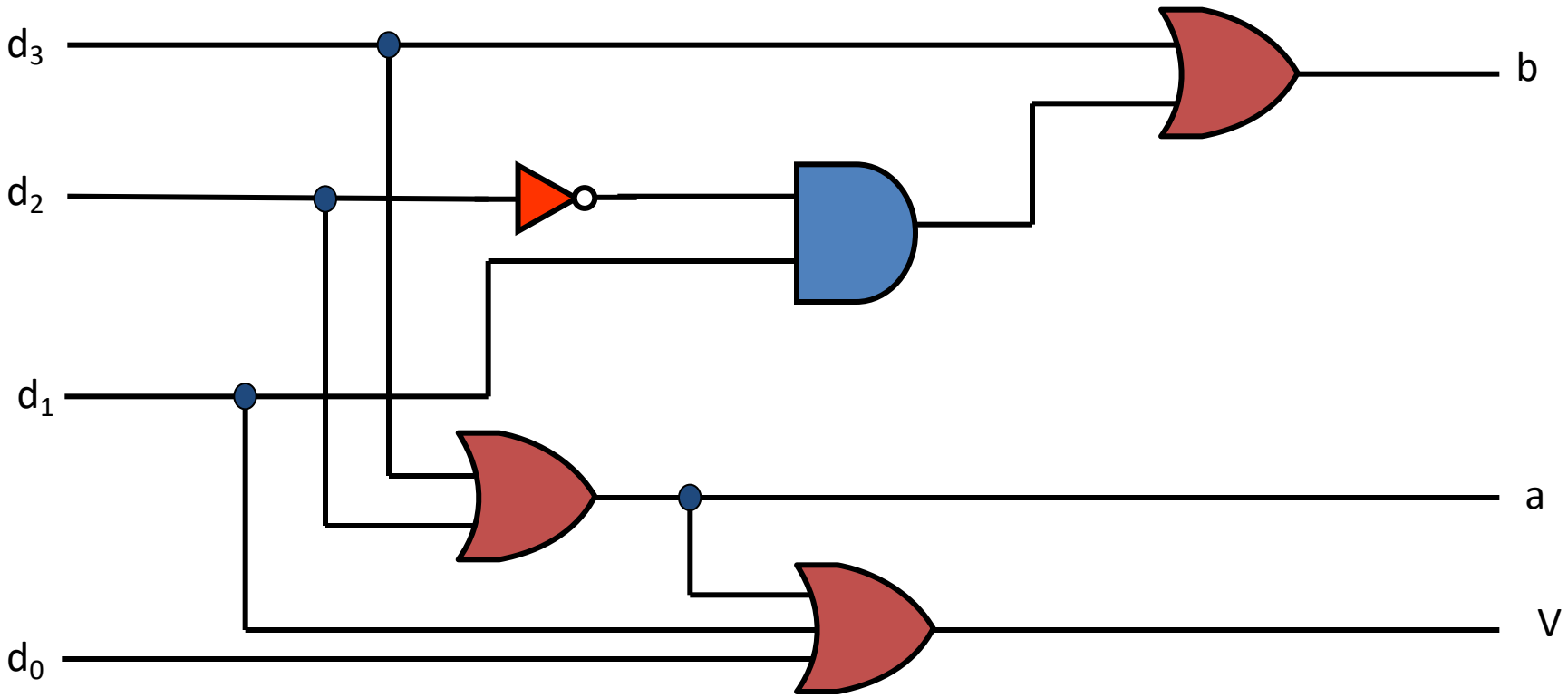
b =

4-bit Priority Encoder: Circuit

$a = d_2 + d_3$

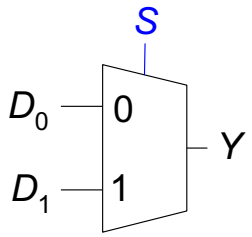
$b = d_1 d_2' + d_3$

$V = d_0 + d_1 + d_2 + d_3$



Multiplexer (Mux)

- Selects between one of *N* inputs and directs it to a single output.
- $\log_2 N$-bit select input – control input
- **Example:** **2:1 Mux**

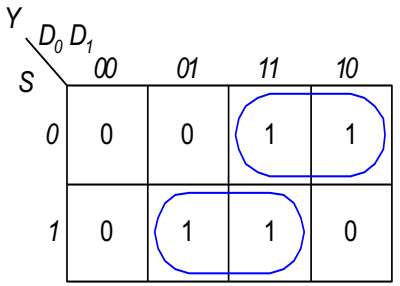


S	D_1	D_0	Y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

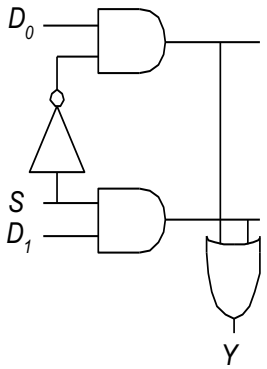
S	Y
0	D_0
1	D_1

Multiplexer Implementations

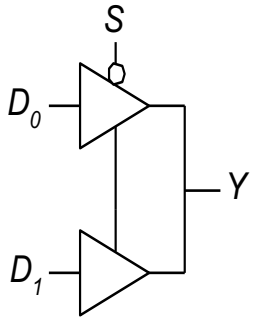
- **Logic gates**
  - Sum-of-products form



$Y = D_0\overline{S} + D_1S$



- **Tristates**
  - For an N-input mux, use N tristates
  - Turn on exactly one to select the appropriate input



4-to-1-Line Multiplexer

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

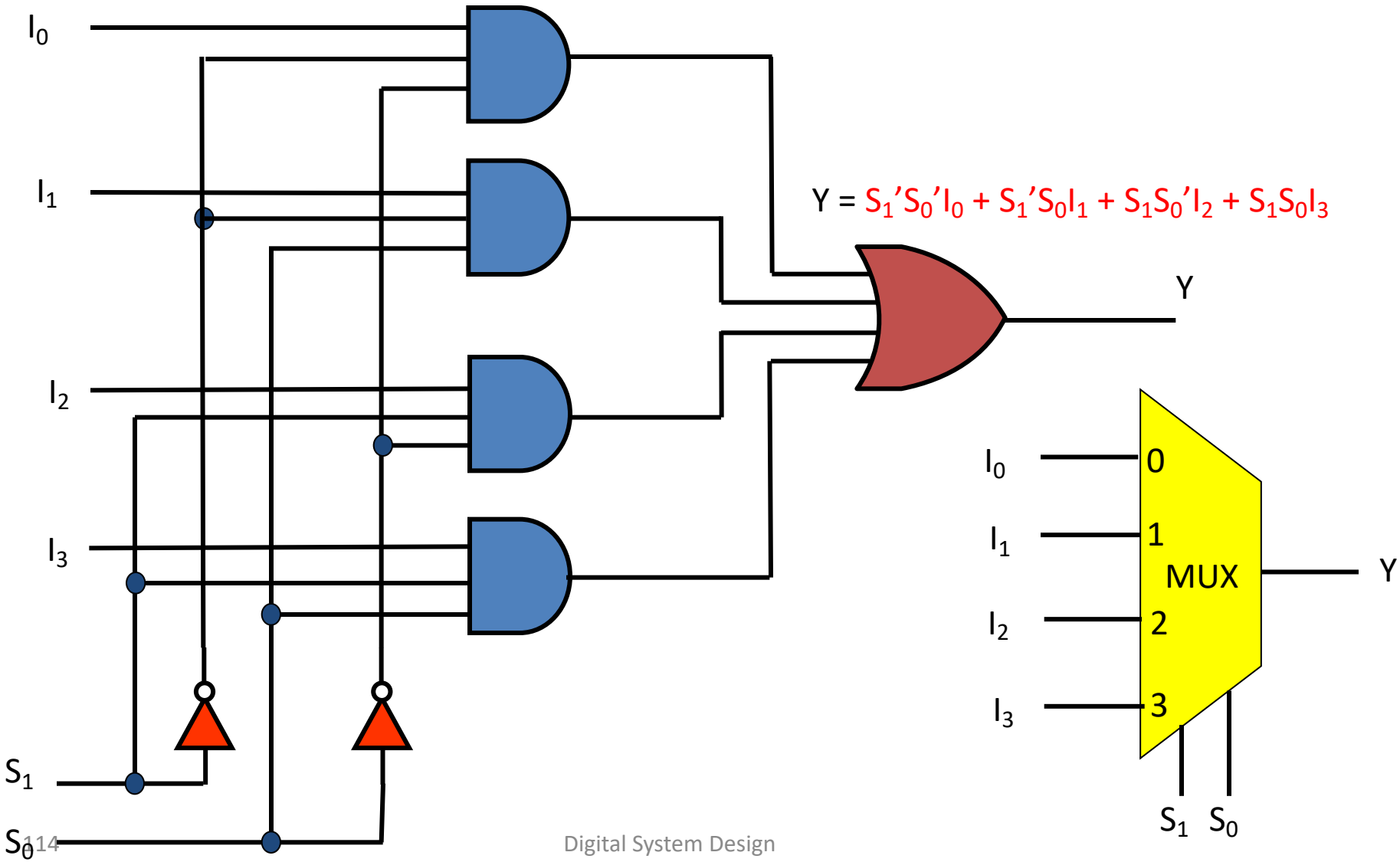
S_1	S_0	Y
0	0	
0	1	
1	0	
1	1	

Y = ?

Interpretation:

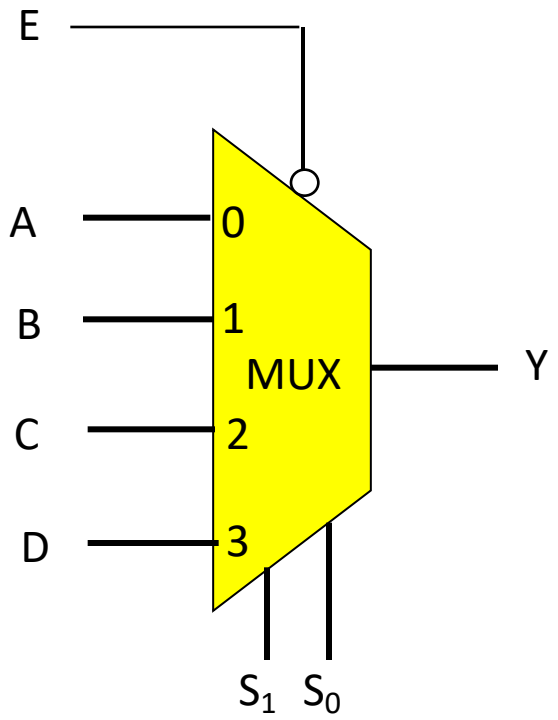
- In case $S_1 = 0$ and $S_0 = 0$, Y selects I_0
- In case $S_1 = 0$ and $S_0 = 1$, Y selects I_1
- In case $S_1 = 1$ and $S_0 = 0$, Y selects I_2
- In case $S_1 = 1$ and $S_0 = 1$, Y selects I_3

4-to-1-Line Multiplexer: Circuit



Multiplexer with Enable Input

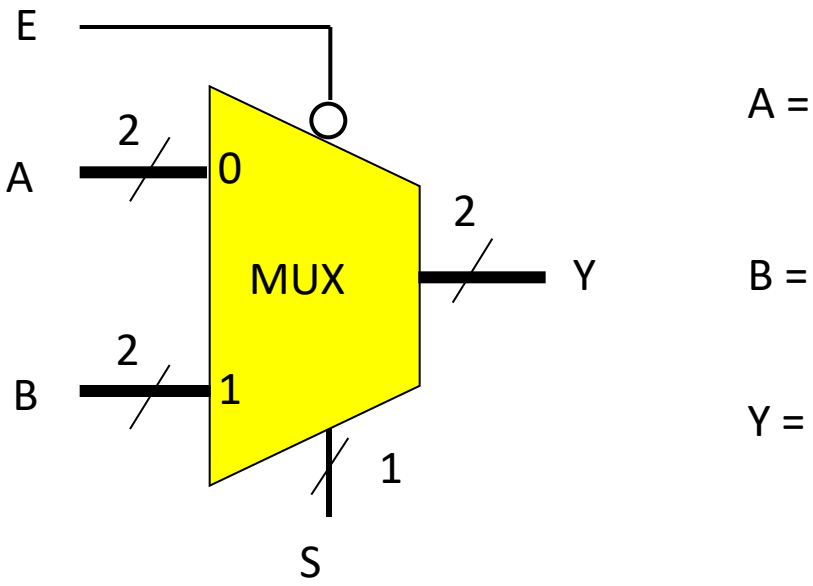
- To select a certain building block we use enable inputs



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

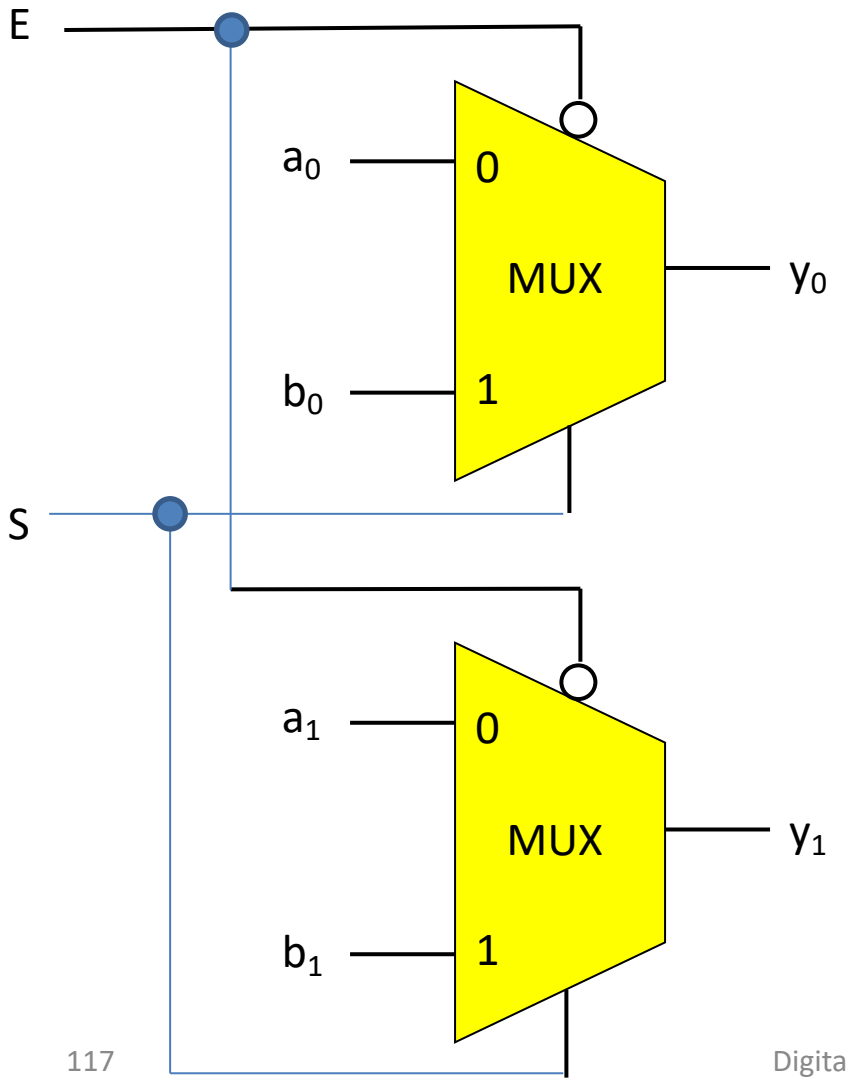


A =

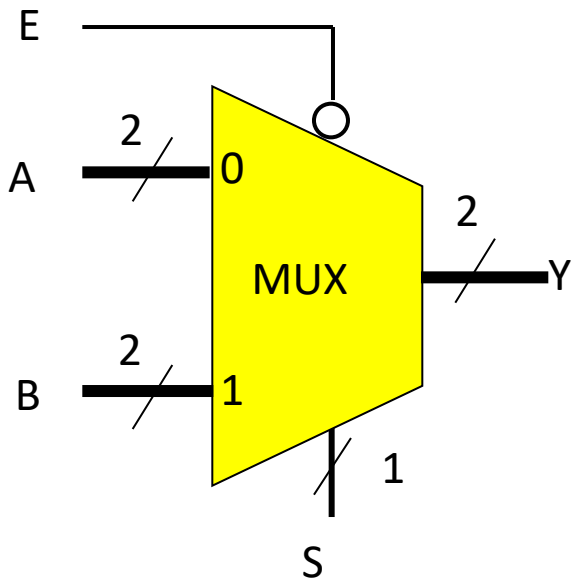
B =

Y =

Multiple-bit Selection Logic 2/2

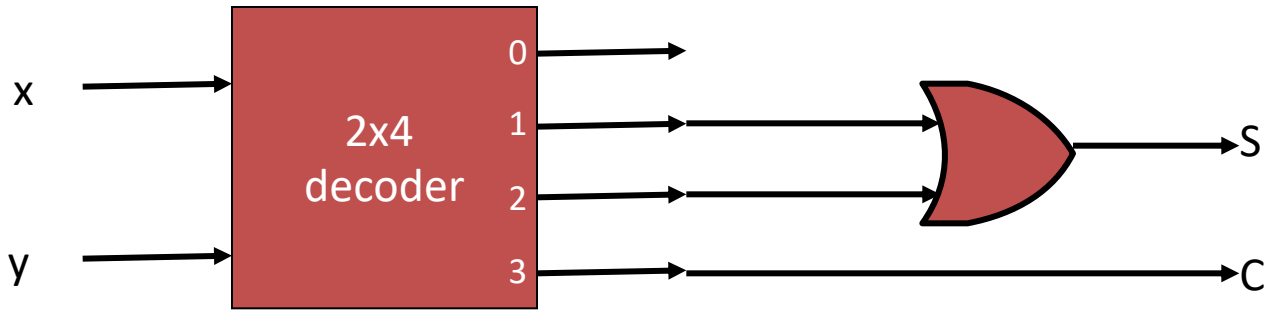


E	S	Y
1	X	all 0's
0	0	A
0	1	B



Design with Multiplexers 1/2

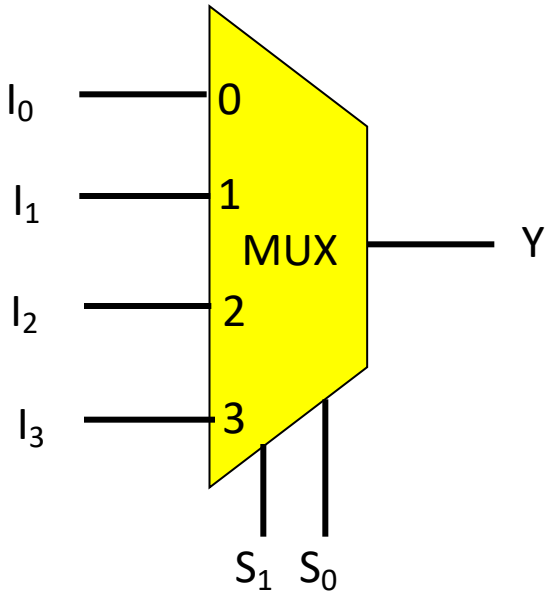
- Reminder: design with decoders
- Half adder
  - C = xy = $\sum(3)$
  - S = x $\oplus$ y = x'y + xy' = $\sum(1, 2)$



- 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

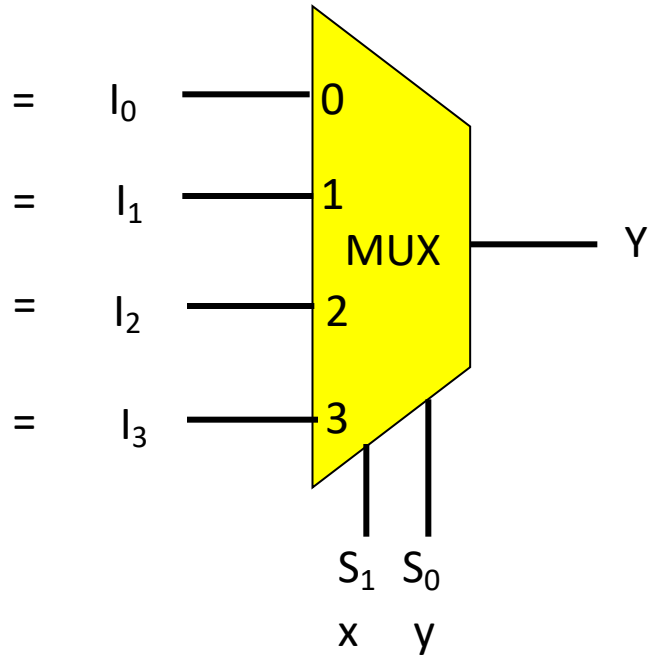


- $S_1 \rightarrow x$
- $S_0 \rightarrow y$
- $S_1'S_0' = x'y'$,
- $S_1'S_0 = x'y$,
- $S_1S_0' = xy'$,
- $S_1S_0 = xy$

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

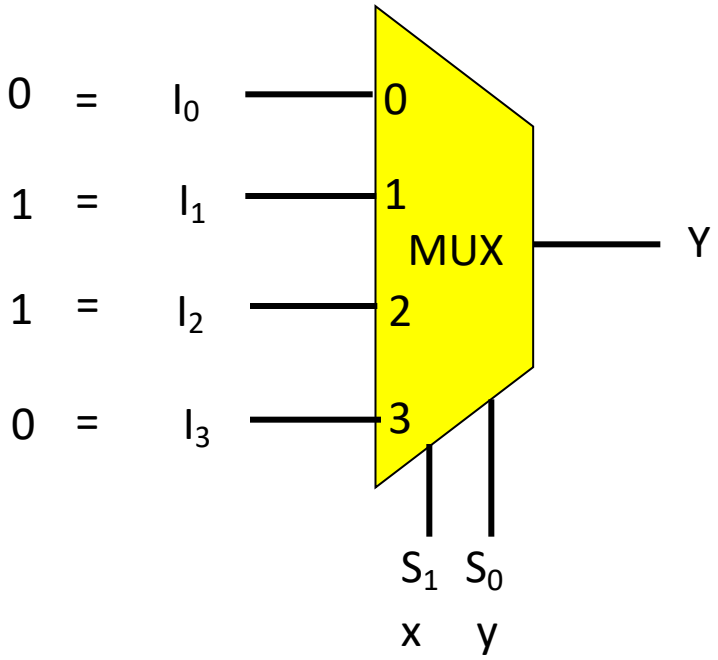
Example: Design with Multiplexers

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



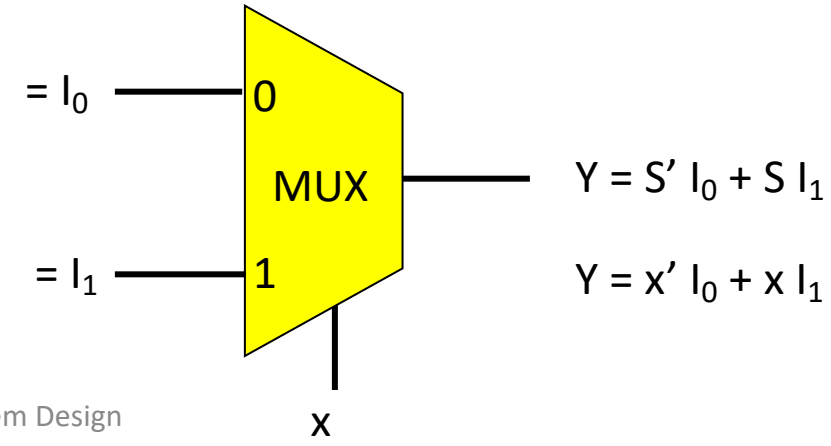
Example: Design with Multiplexers

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



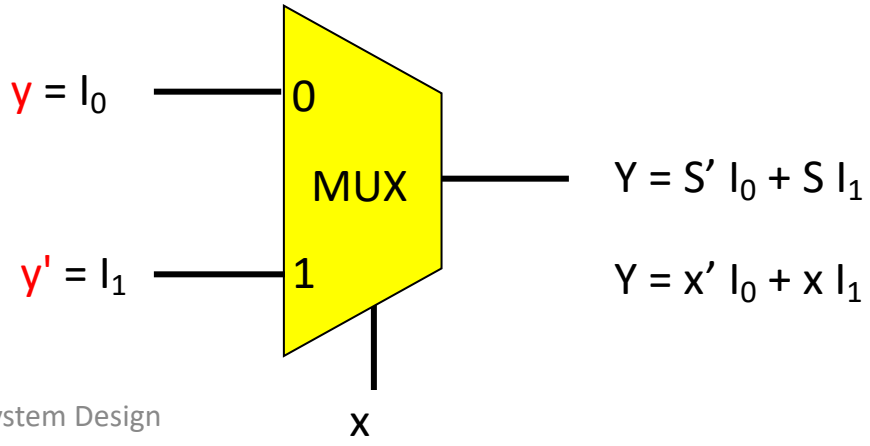
Design with Multiplexers Efficiently

- 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

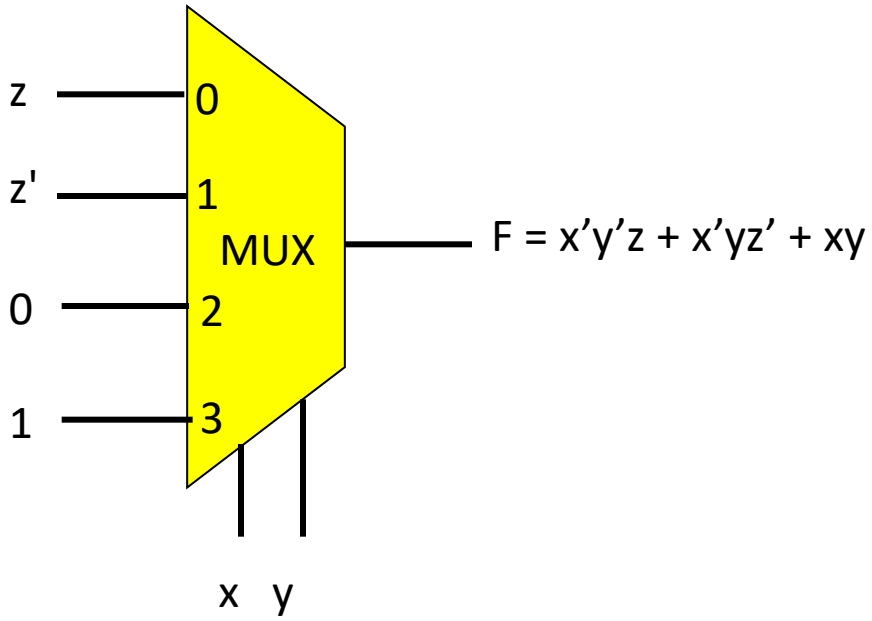
- $F(x, y, z) = \Sigma(1, 2, 6, 7)$
  - $F = x'y'z + x'yz' + xyz' + xyz$
  - $Y = S_1'S_0'\, I_0 + S_1'S_0\, I_1 + S_1S_0'\, I_2 + S_1S_0\, I_3$
  - $I_0 = z \quad I_1 = z' \quad I_2 = 0 \quad I_3 = 1$

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	

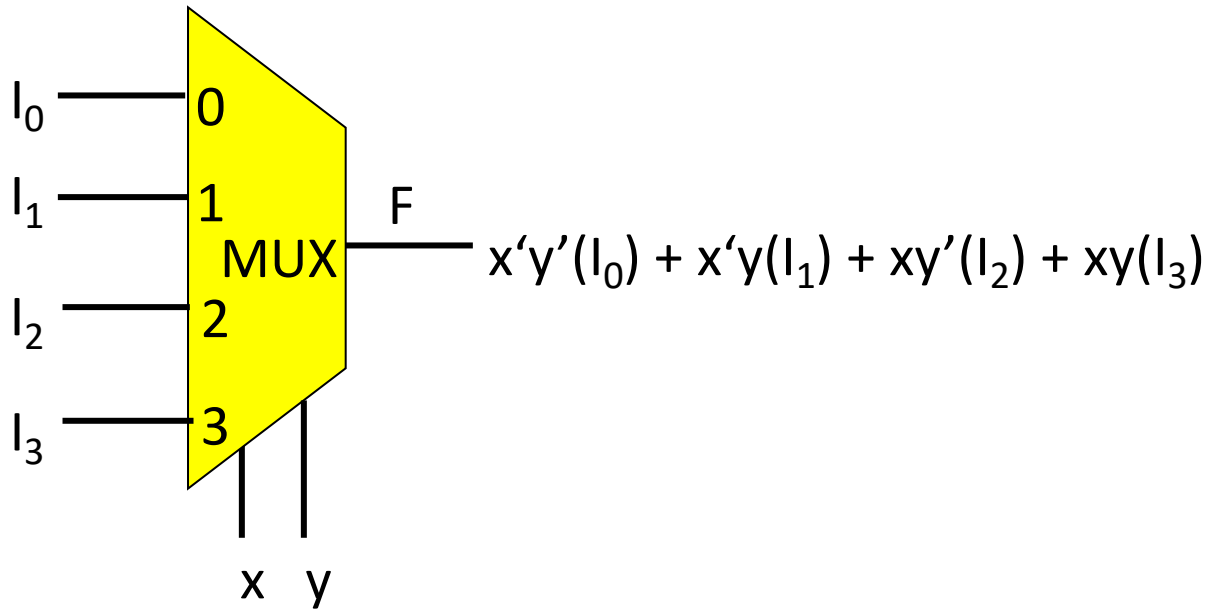
Example: Design with Multiplexers

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



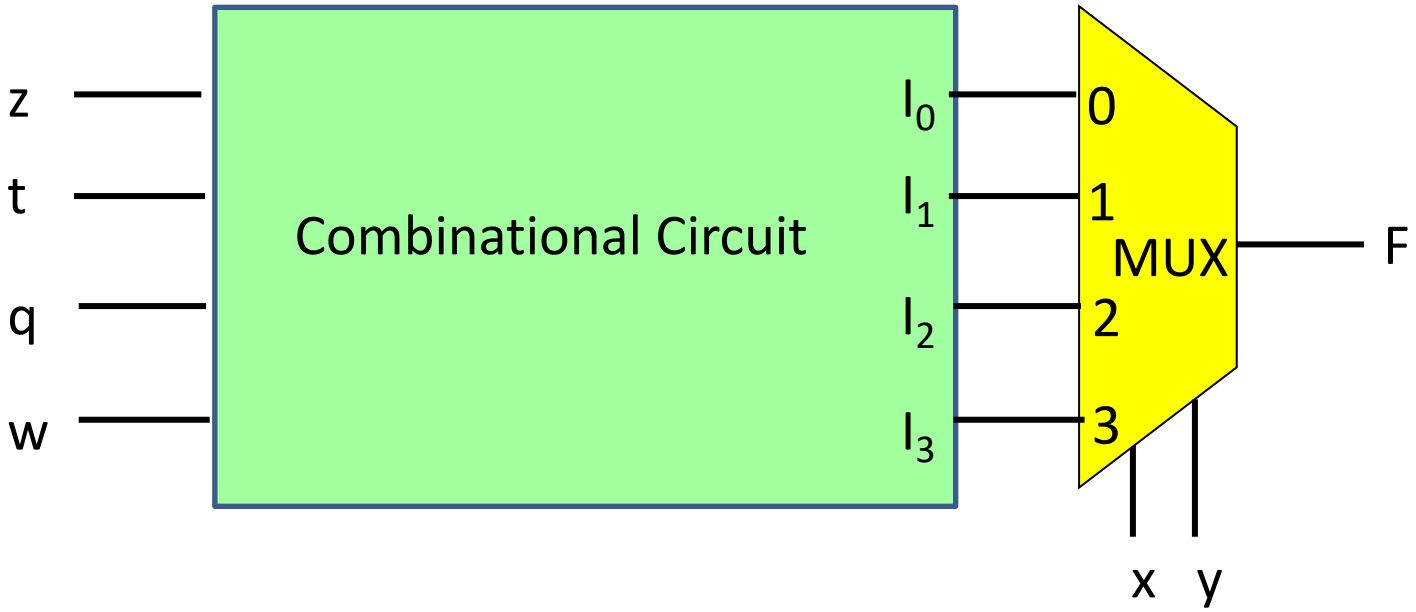
$F_1 = x'y'(0) + x'y(1) + xy'(1) + xy(0)$

$F_2 = x'y'(z) + x'y(z') + xy'(0) + xy(1)$

...

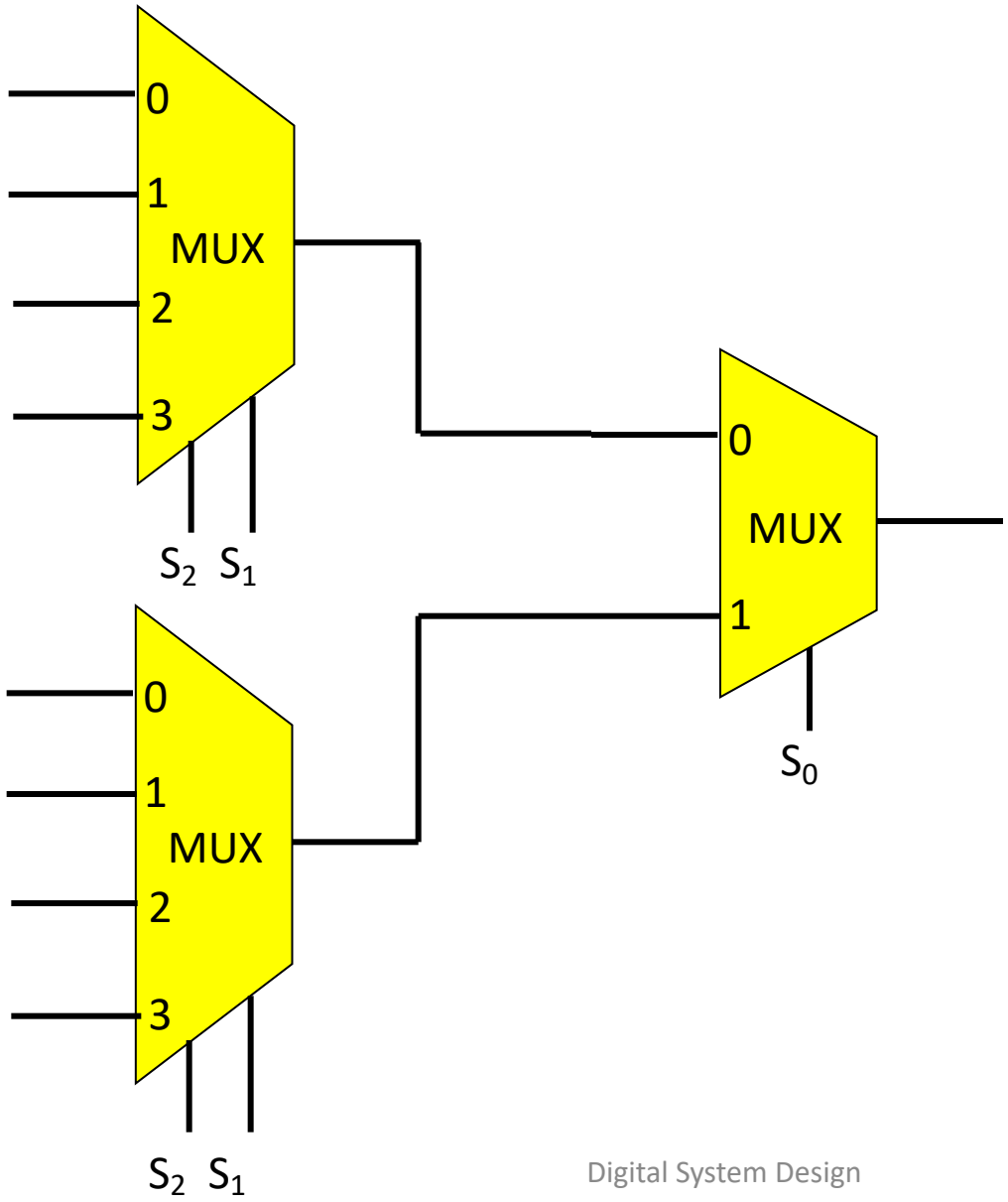
$F = x'y'(P(z,t,q,w)) + x'y(Q(z,t,q,w)) + xy'(R(z,t,q,w)) + xy(S(z,t,q,w))$

Design with Multiplexers



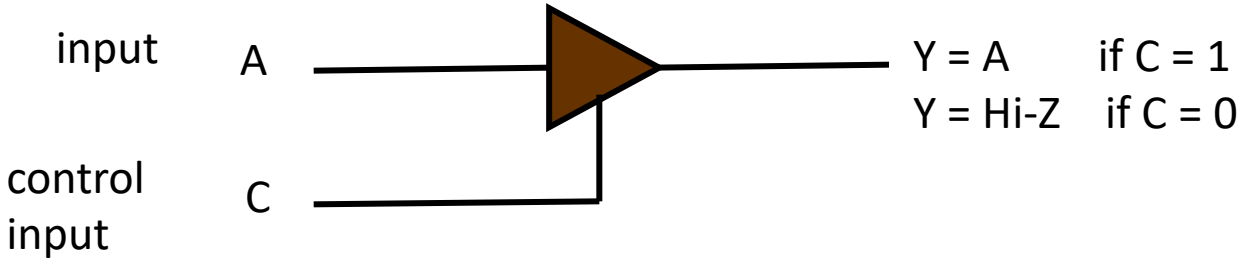
$$F=x'y'(P(z,t,q,w)) + x'y(Q(z,t,q,w)) + xy'(R(z,t,q,w)) + xy(S(z,t,q,w))$$

Combining Multiplexers



3-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

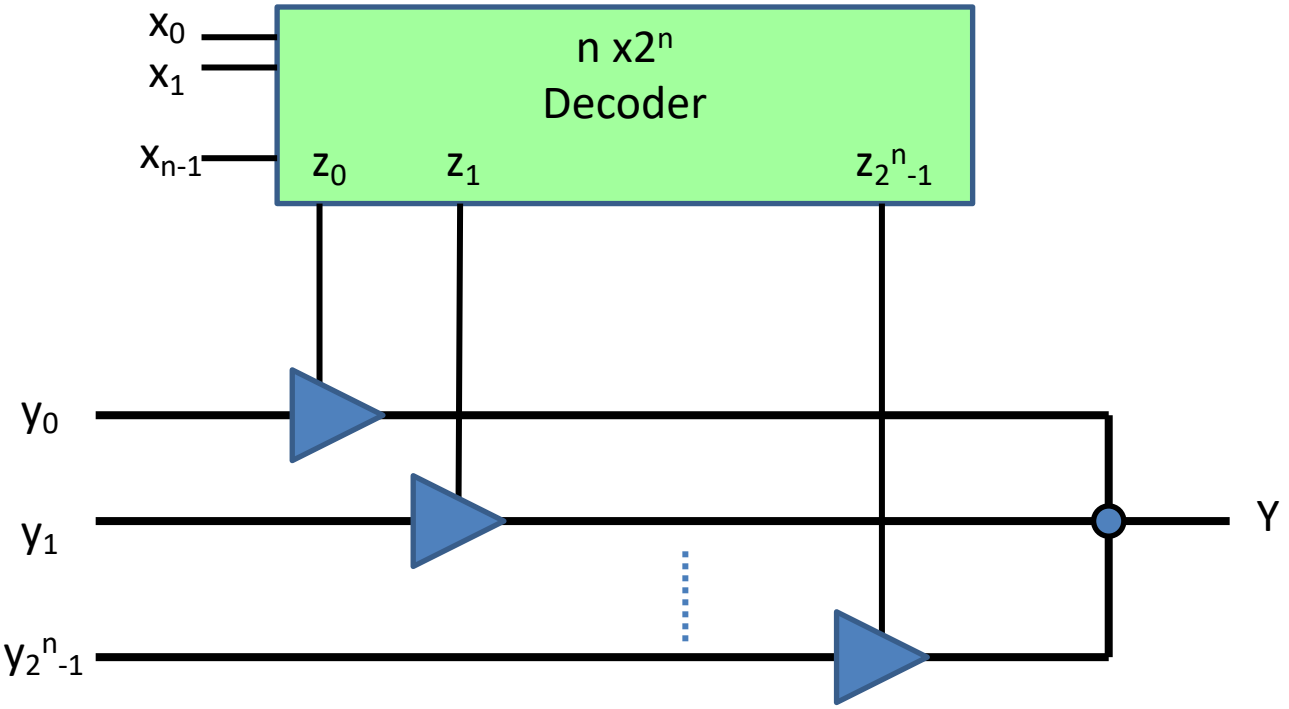


3-State Buffers

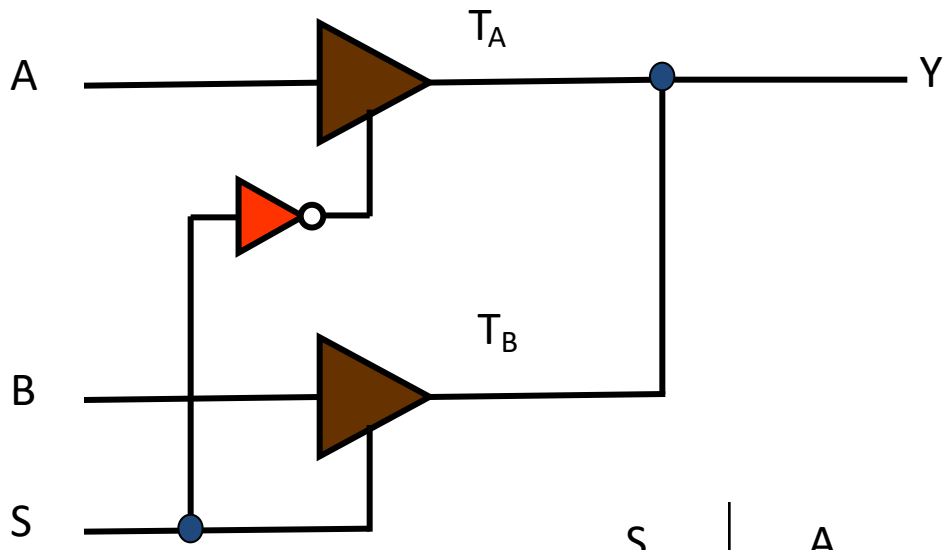
- Remember that we cannot connect the outputs of other logic gates directly.
- But we can connect the outputs of three-state buffers
  - provided that no two three-state buffers drive the same wire to opposite 0 and 1 values at the same time.

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

Example- 3 State MUX



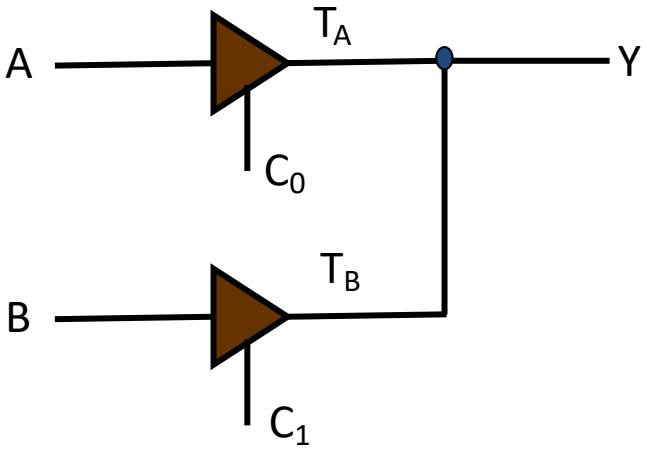
Multiplexing with 3-State Buffers



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

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

Two Active Outputs - 1

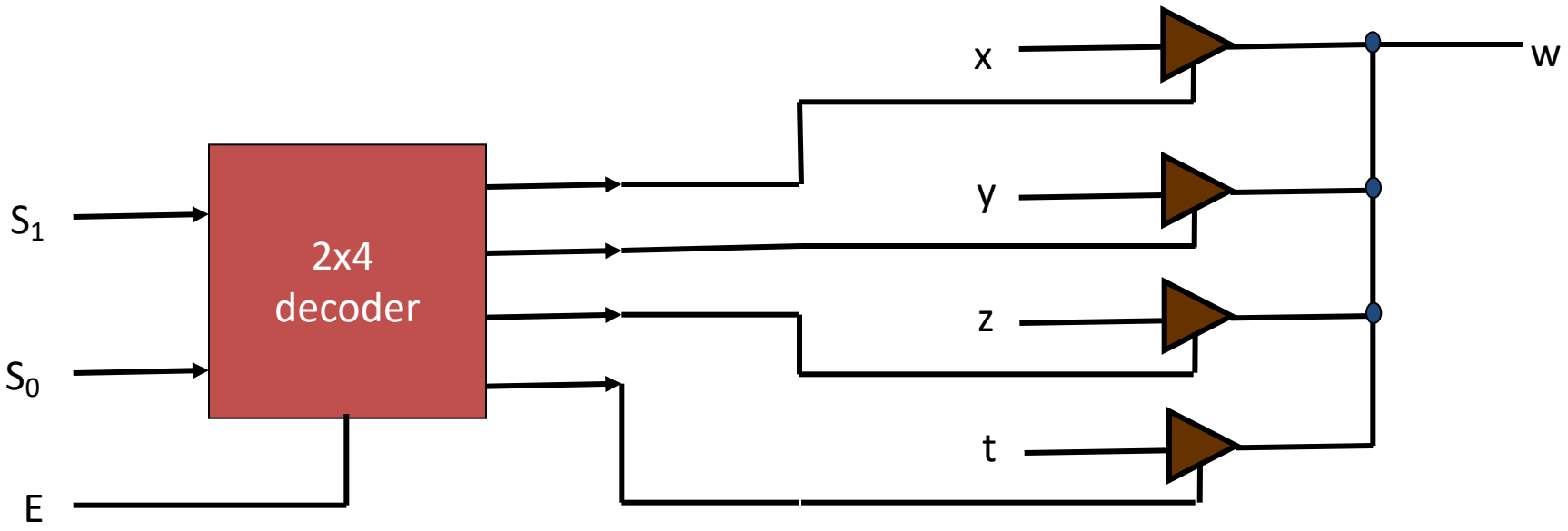


What will happen if $C_1 = C_0 = 1$?

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	

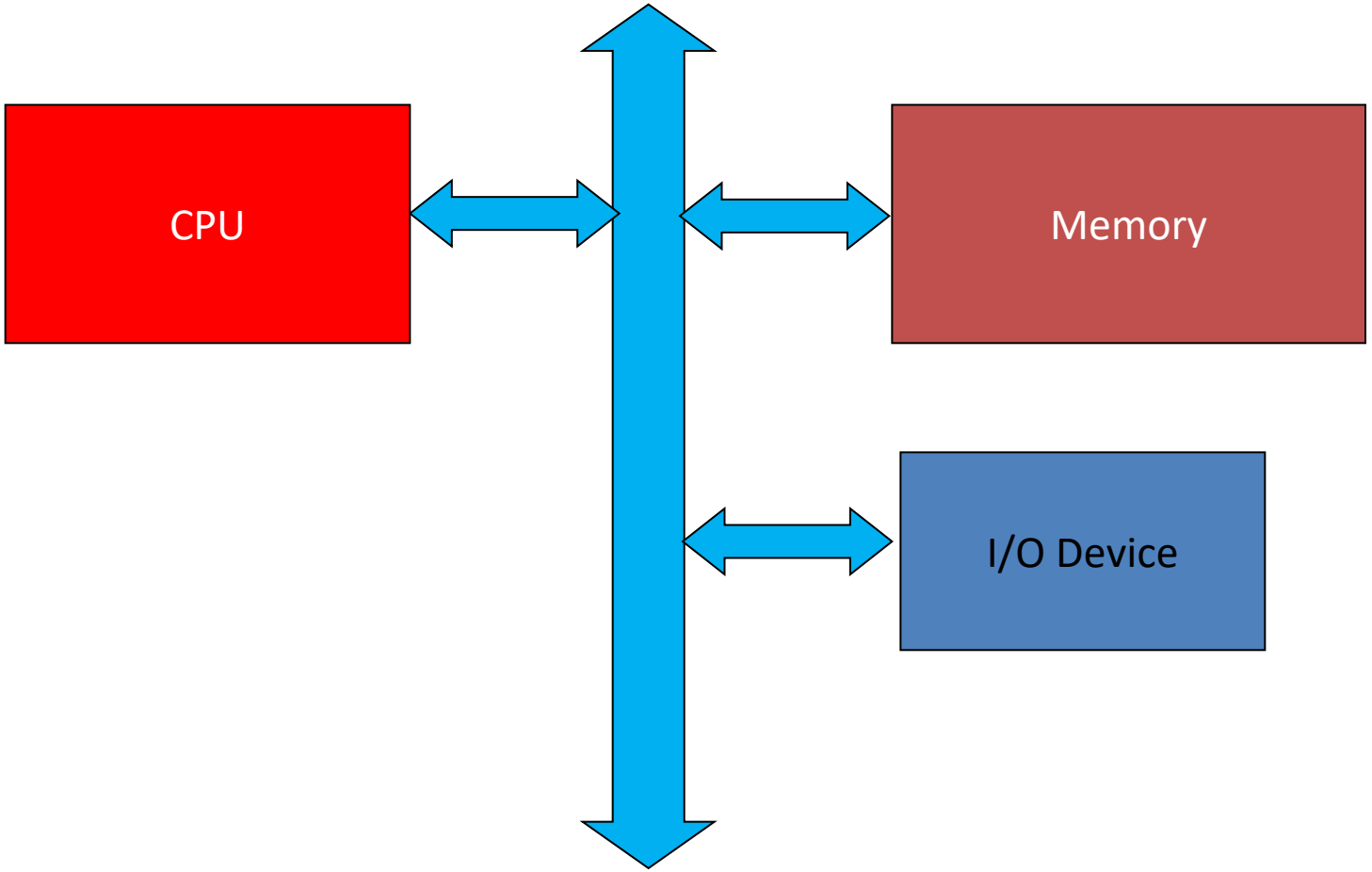
Design Principle with 3-State Buffers

- 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.



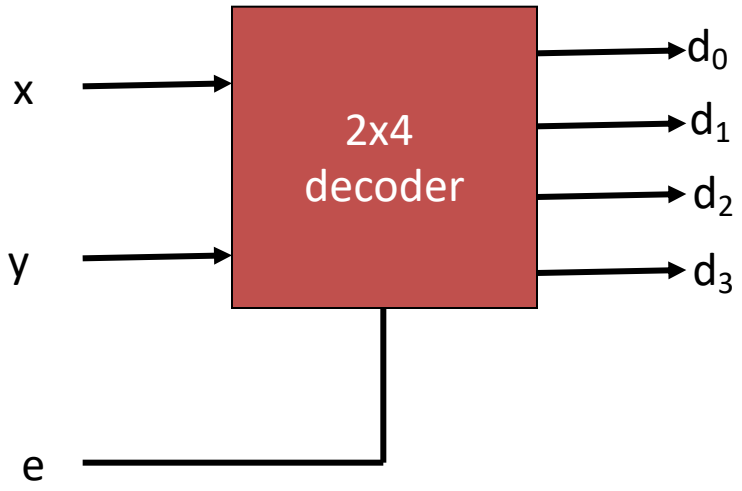
Busses with 3-State Buffers

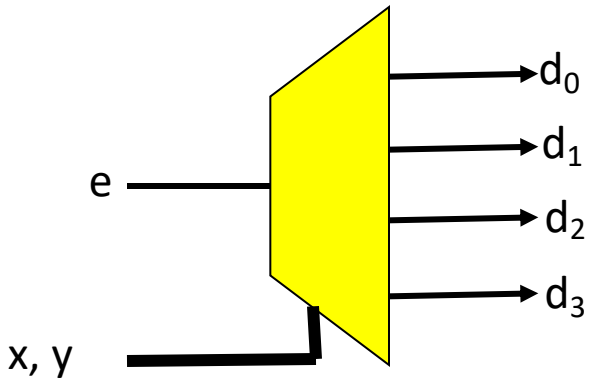
- There are important uses of three-state buffers



Demultiplexer

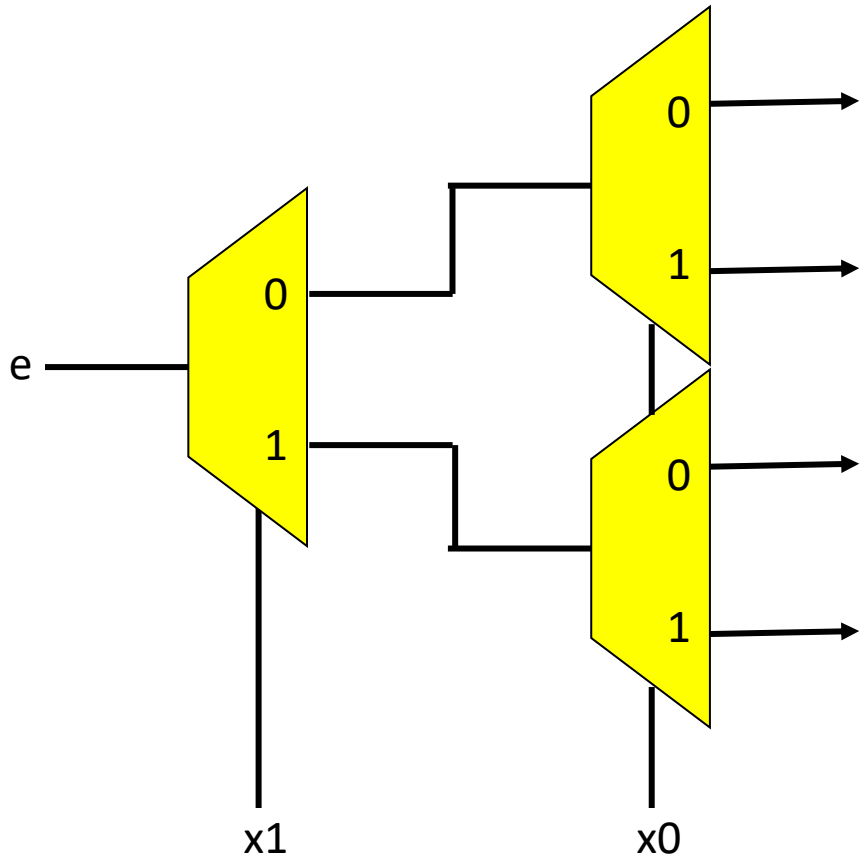
- A demultiplexer is a combinational circuit
  - it receives information from a single input line and directs it to one of 2^n output lines
  - It has n selection lines as to which output will get the input



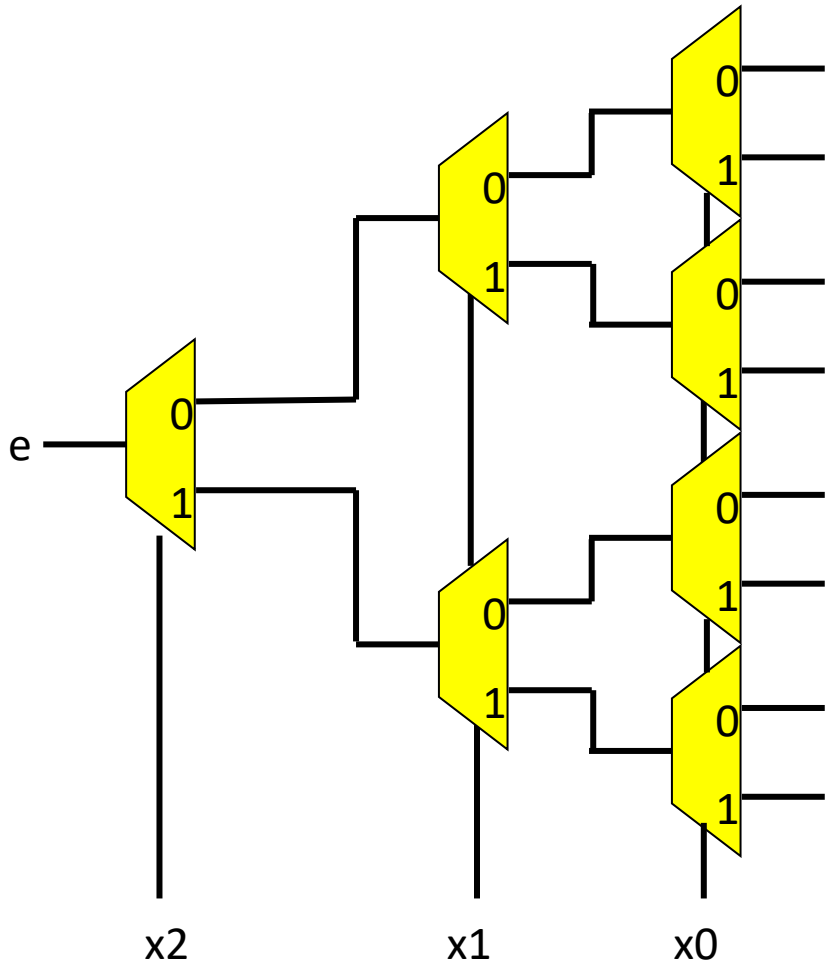


$d_0 = e$ when $x = 0$ and $y = 0$
$d_1 = e$ when $x = 0$ and $y = 1$
$d_2 = e$ when $x = 1$ and $y = 0$
$d_3 = e$ when $x = 1$ and $y = 1$

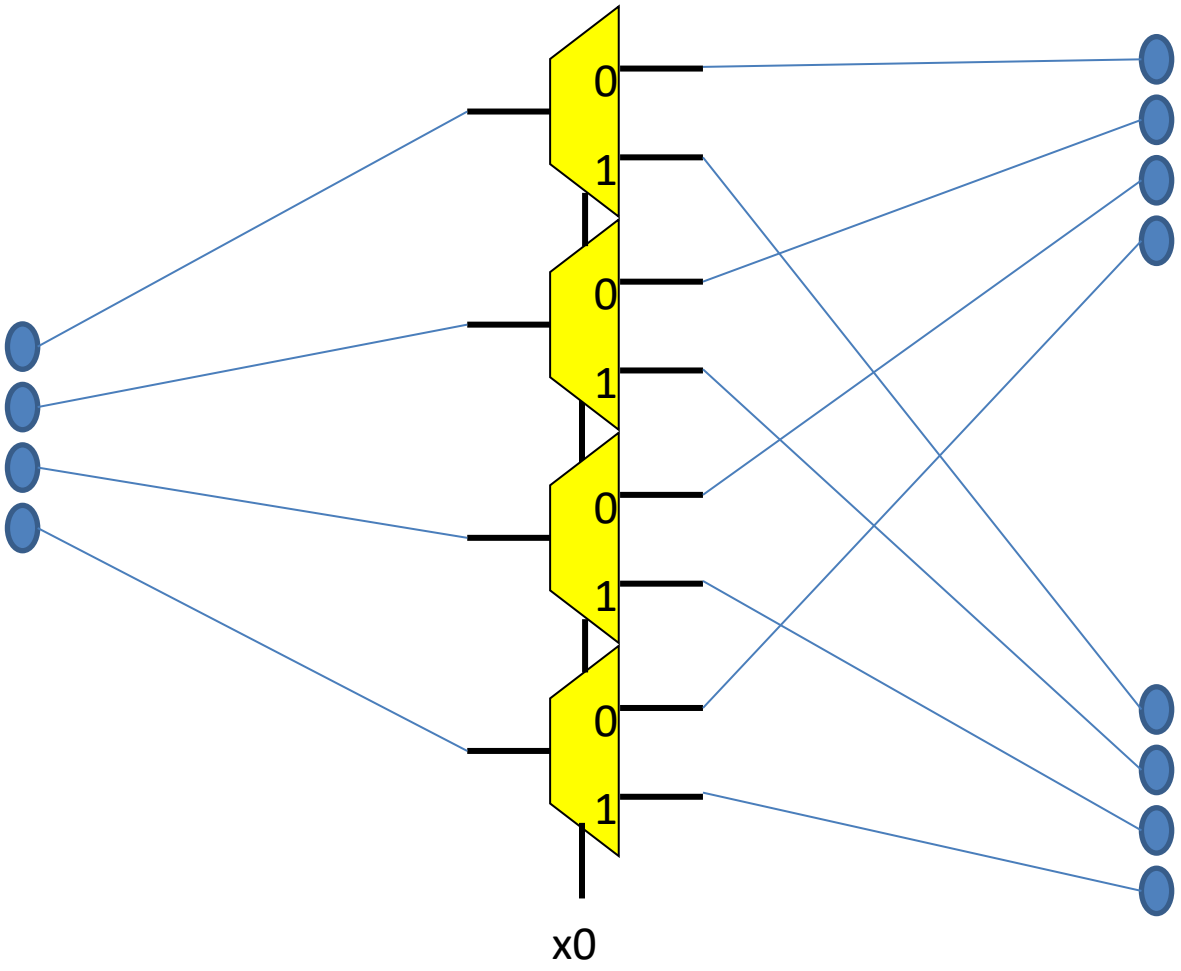
Demultiplexer



Demultiplexer

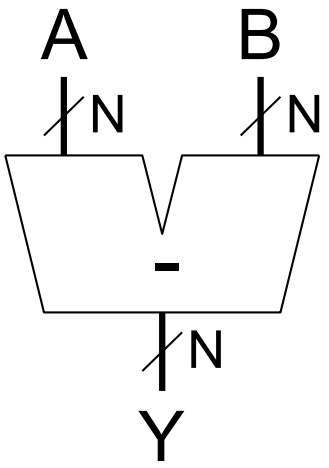


Demultiplexer

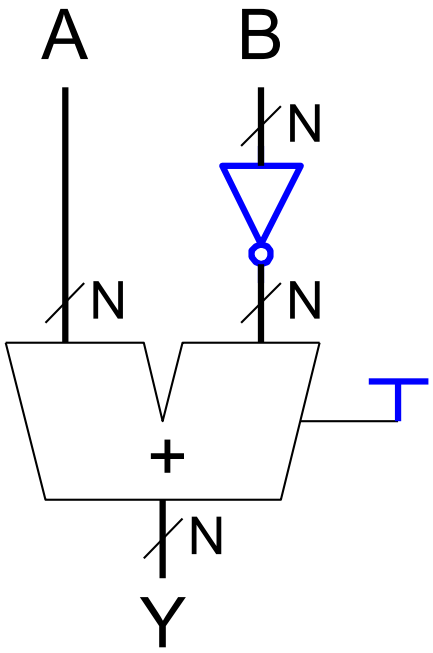


Symbols: Subtracter

Symbol

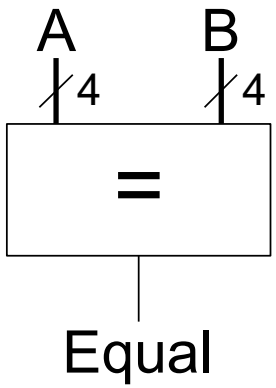


Implementation

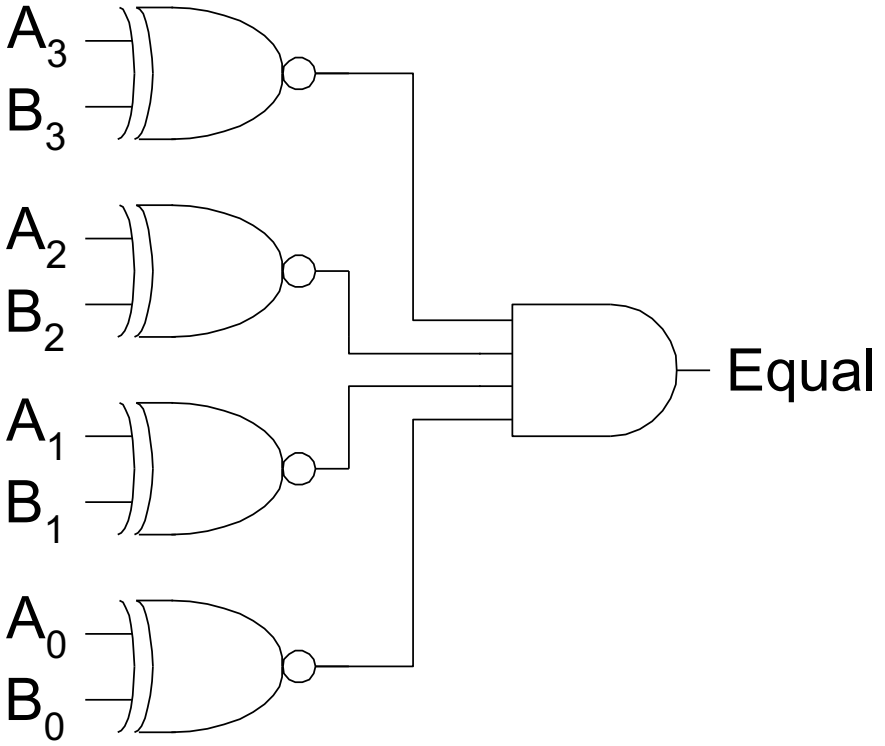


Symbols: Equality Comparator

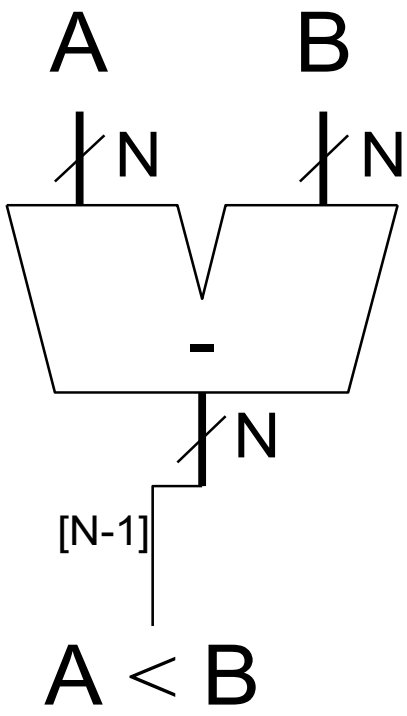
Symbol



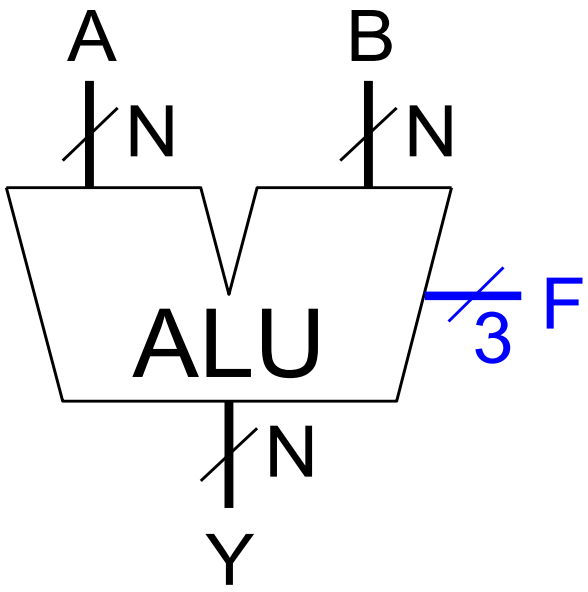
Implementation



Symbols: Less Than Comparator

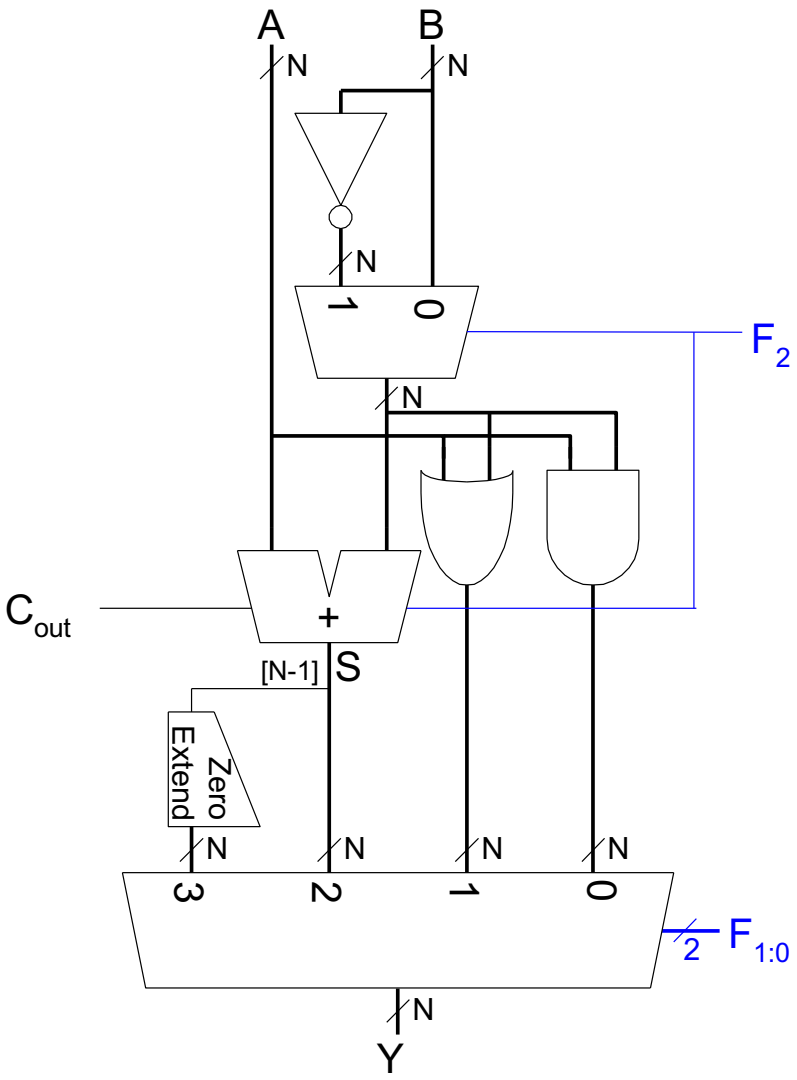


Arithmetic Logic Unit (ALU)



$F_{2:0}$	Function
000	A & B
001	A | B
010	A + B
011	not used
100	A & ~B
101	A | ~B
110	A - B
111	slt

ALU Design

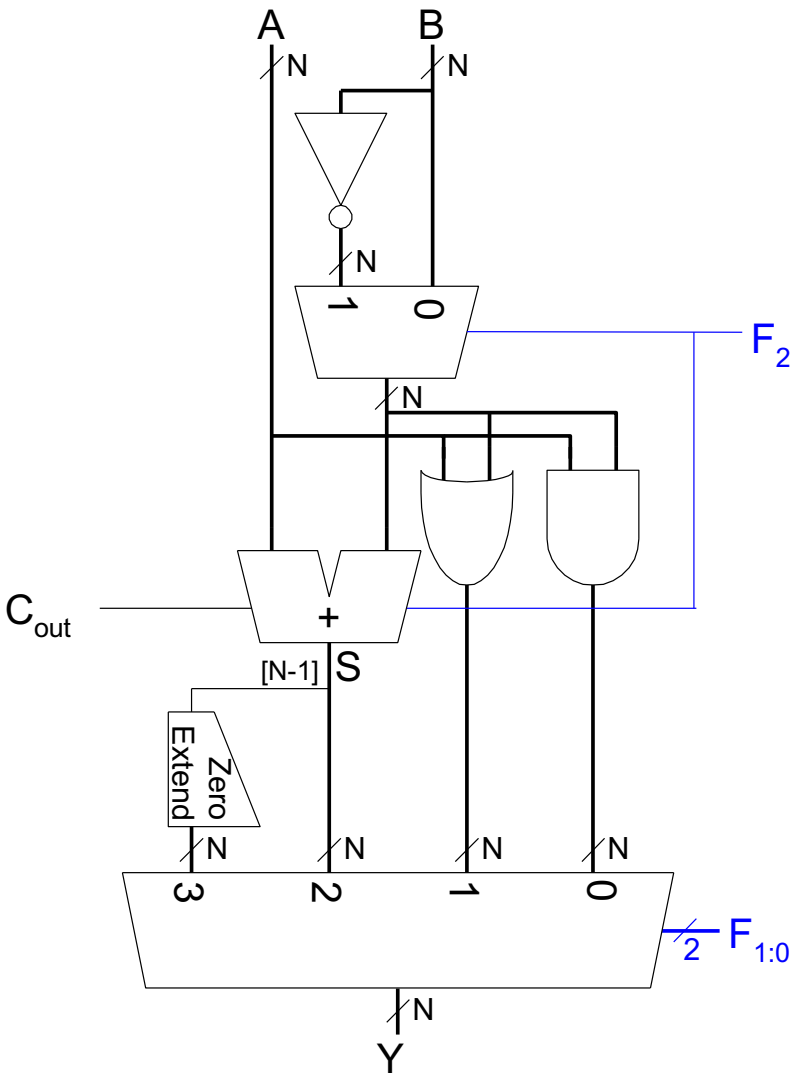


$F_{2:0}$	Function
000	A & B
001	A | B
010	A + B
011	not used
100	A & ~B
101	A | ~B
110	A - B
111	slt

ALU Design

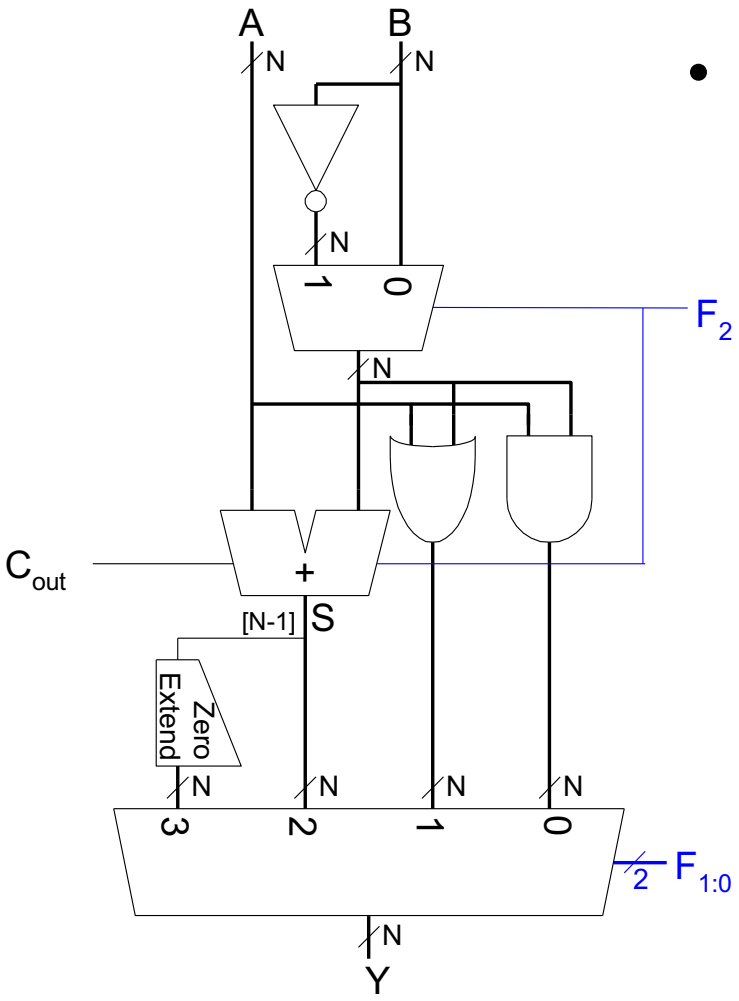
$$A \text{ slt } B = \begin{cases} 000 \ldots 001 & \text{if } A < B, \text{ i.e. if } A - B < 0 \\ 000 \ldots 000 & \text{if } A \geq B, \text{ i.e. if } A - B \geq 0 \end{cases}$$

ALU Design



$F_{2:0}$	Function
000	A & B
001	A | B
010	A + B
011	not used
100	A & ~B
101	A | ~B
110	A - B
111	slt

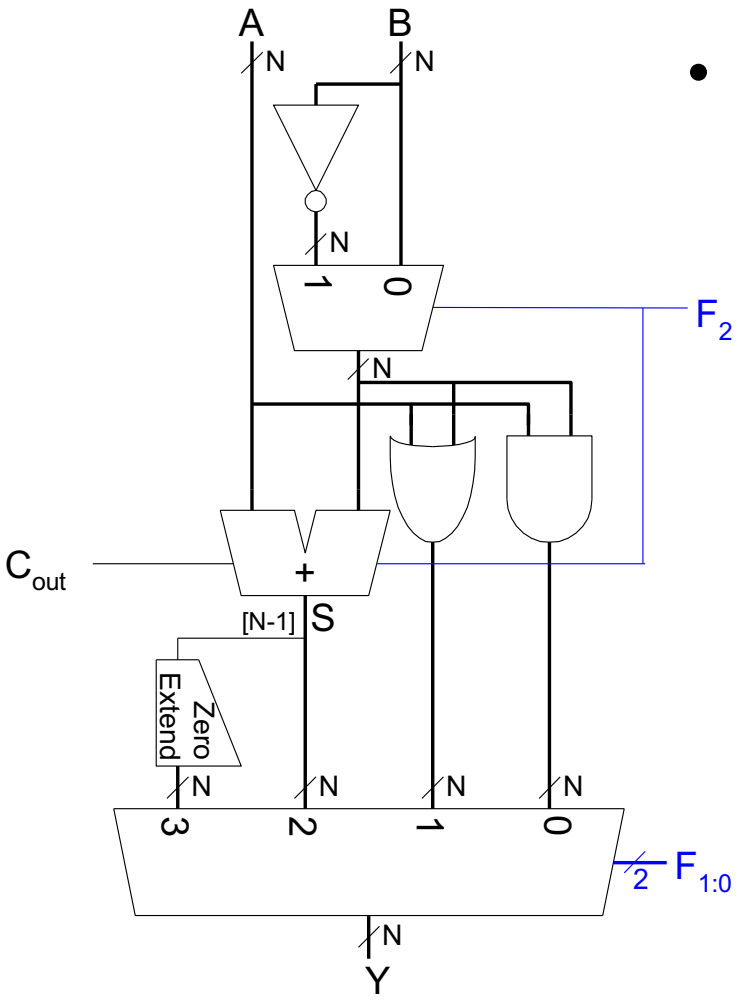
Set Less Than (SLT) Example



- Configure 32-bit ALU for slt operation: $A = 25$ and $B = 32$

<157>

Set Less Than (SLT) Example



- Configure 32-bit ALU for slt operation: $A = 25$ and $B = 32$
  - $A < B$, so Y should be 32-bit representation of 1 (0x00000001)
  - $F_{2:0} = 111$
    - $F_2 = 1$ (adder acts as subtracter), so 25 - 32 = -7
    - -7 has 1 in the most significant bit ($S_{31} = 1$)
    - $F_{1:0} = 11$ multiplexer selects $Y = S_{31}$ (zero extended) = 0x00000001.

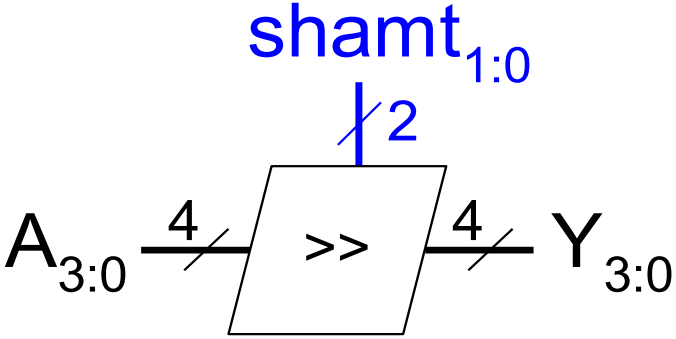
Shifters

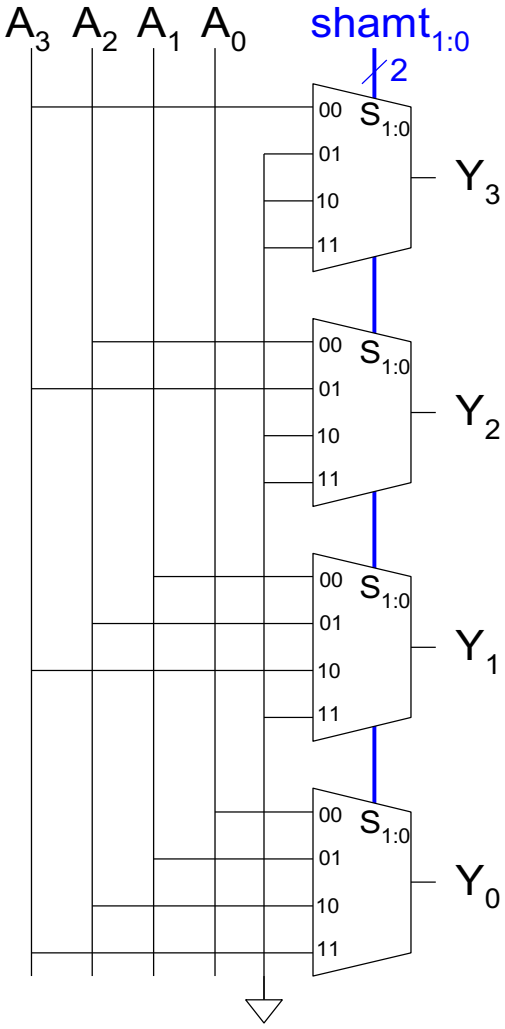
- **Logical shifter:** shifts value to left or right and fills empty spaces with 0's
  - Ex: 11001 >> 2 =
  - Ex: 11001 << 2 =
- **Arithmetic shifter:** same as logical shifter, but on right shift, fills empty spaces with the old most significant bit (msb).
  - Ex: 11001 >>> 2 =
  - Ex: 11001 <<< 2 =
- **Rotator:** rotates bits in a circle, such that bits shifted off one end are shifted into the other end
  - Ex: 11001 ROR 2 =
  - Ex: 11001 ROL 2 =

Shifters

- **Logical shifter:** shifts value to left or right and fills empty spaces with 0's
  - Ex: 11001 >> 2 = 00110
  - Ex: 11001 << 2 = 00100
- **Arithmetic shifter:** same as logical shifter, but on right shift, fills empty spaces with the old most significant bit (msb).
  - Ex: 11001 >>> 2 = **11**110
  - Ex: 11001 <<< 2 = 00100
- **Rotator:** rotates bits in a circle, such that bits shifted off one end are shifted into the other end
  - Ex: 11001 ROR 2 = 01110
  - Ex: 11001 ROL 2 = 00111

Shifter Design





Shifters as Multipliers, Dividers

- $A << N = A \times 2^N$
  - **Example:** 00001 << 2 = 00100 ($1 \times 2^2 = 4$)
  - **Example:** 11101 << 2 = 10100 ($-3 \times 2^2 = -12$)
- $A >>> N = A \div 2^N$
  - **Example:** 01000 >>> 2 = 00010 ($8 \div 2^2 = 2$)
  - **Example:** 10000 >>> 2 = 11100 ($-16 \div 2^2 = -4$)

Binary Multiplication

- Multiplication is achieved by adding a list of shifted multiplicands according to the digits of the multiplier.
- Ex. (unsigned)

```
    11                      1 0 1 1      multiplicand (4 bits)
  X 13                    X 1 1 0 1      multiplier  (4 bits)
--------              ------------------
    33                      1 0 1 1
   11                     0 0 0 0
 _____                   1 0 1 1
   143                  1 0 1 1
                     --------------------
                     1 0 0 0 1 1 1 1      Product (8 bits)
```

Binary Multiplication

- An n-bit *x* n-bit multiplier can be realized in combinational circuitry by using an array of n-1 n-bit adders where each adder is shifted by one position.
- For each adder one input is the multiplied by 0 or 1 (using AND gates) depending on the multiplier bit, the other input is n partial product bits.

				X_3	X_2	X_1	X_0
			x	Y_3	Y_2	Y_1	Y_0
				$X_3.Y_0$	$X_2.Y_0$	$X_1.Y_0$	$X_0.Y_0$
			$X_3.Y_1$	$X_2.Y_1$	$X_1.Y_1$	$X_0.Y_1$	
		$X_3.Y_2$	$X_2.Y_2$	$X_1.Y_2$	$X_0.Y_2$		
	$X_3.Y_3$	$X_2.Y_3$	$X_1.Y_3$	$X_0.Y_3$			
P_7	P_6	P_5	P_4	P_3	P_2	P_1	P_0

Multipliers

- **Partial products** are formed by multiplying a single digit of the multiplier with multiplicand
- **Shifted** partial products are **summed** to form result

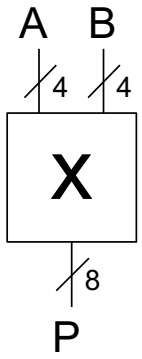
Decimal		Binary
230	multiplicand	0101
x 42	multiplier	x 0111
460	partial products	0101
+ 920		0101
9660		0101
		+ 0000
	result	0100011

230 x 42 = 9660

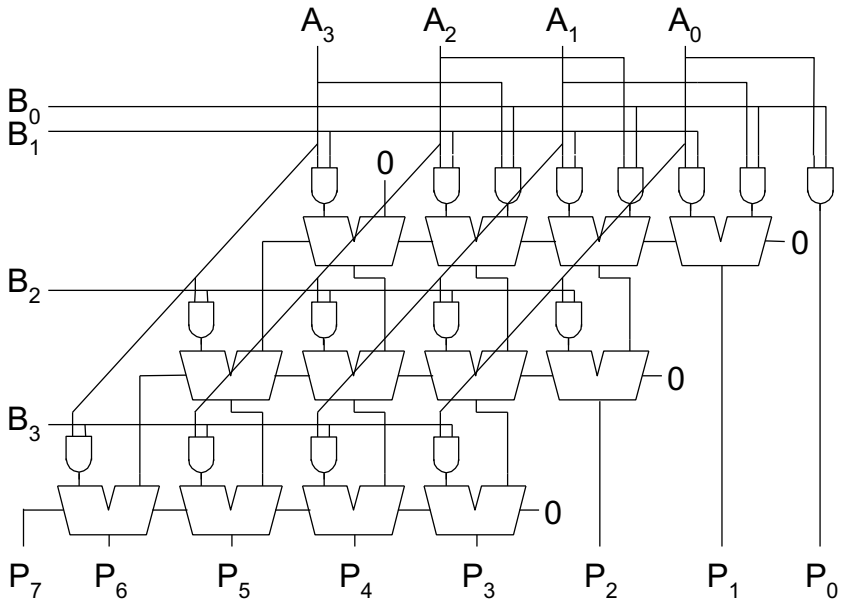
5 x 7 = 35

4 x 4 Multiplier

Symbol



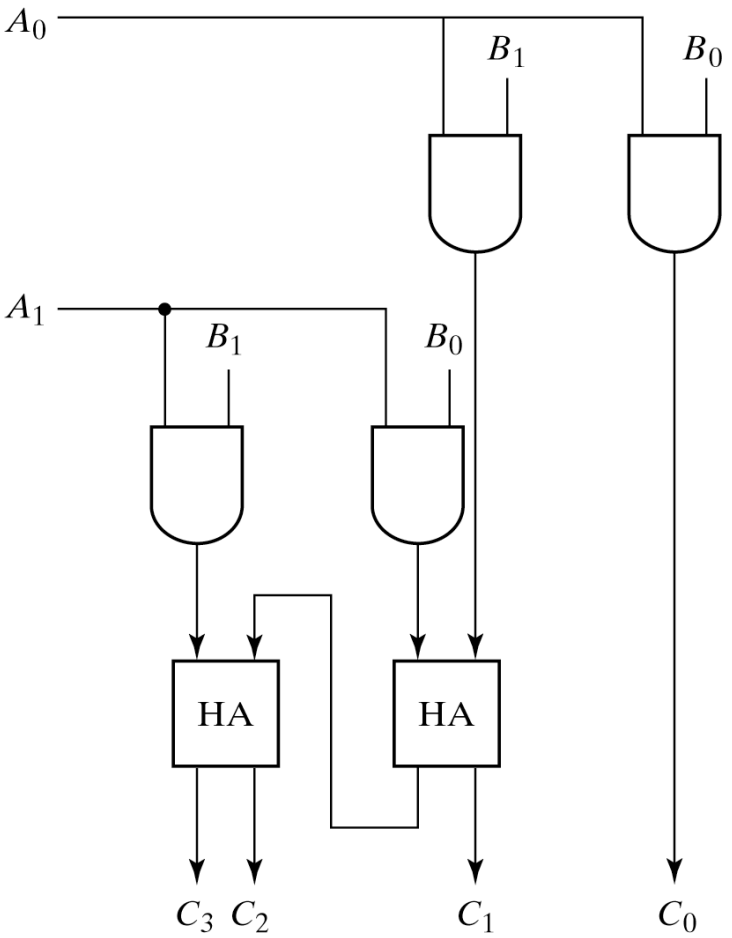
$$
\begin{array}{rcccccccc}
 & & & A_3 & A_2 & A_1 & A_0 & & \\
 & x & & B_3 & B_2 & B_1 & B_0 & & \\
\hline
 & & A_3B_0 & A_2B_0 & A_1B_0 & A_0B_0 & & & \\
 & A_3B_1 & A_2B_1 & A_1B_1 & A_0B_1 & & & & \\
A_3B_2 & A_2B_2 & A_1B_2 & A_0B_2 & & & & & \\
+ \quad A_3B_3 & A_2B_3 & A_1B_3 & A_0B_3 & & & & & \\
\hline
P_7 & P_6 & P_5 & P_4 & P_3 & P_2 & P_1 & P_0
\end{array}
$$



Binary Multiplier

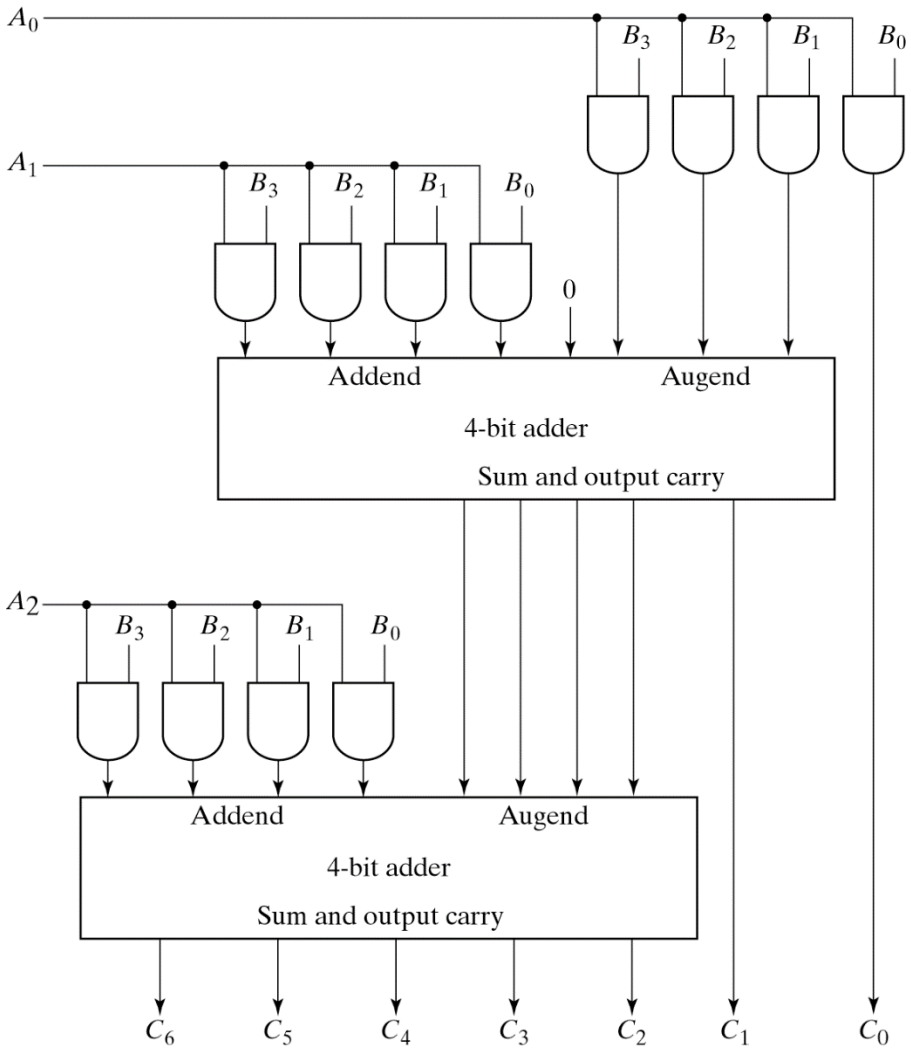
- Partial products – AND operations

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



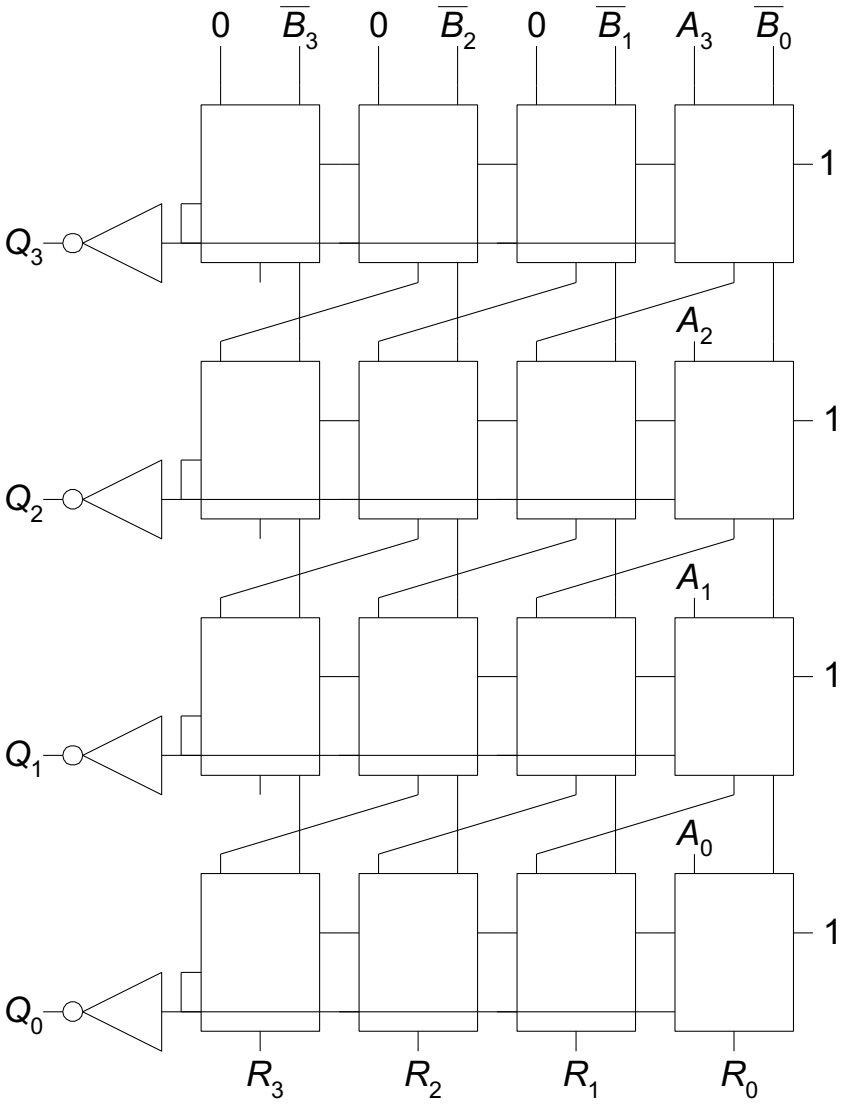
2-Bit by 2-Bit Binary Multiplier

4-bit by 3-bit binary multiplier

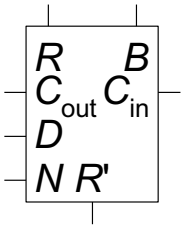


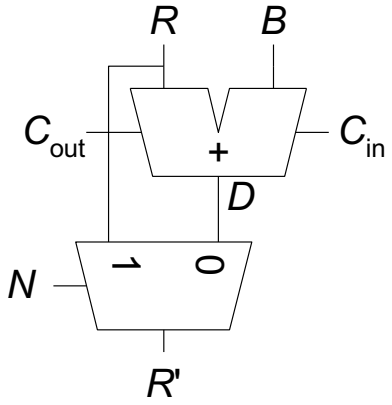
4-Bit by 3-Bit Binary Multiplier

4 x 4 Divider



Legend





$A/B = Q + R/B$ $\quad \frac{\text{Dividend}}{\text{Divisor}} = \text{Quotient} + \frac{\text{Remainder}}{\text{Divisor}}$

Algorithm:

$R' = 0$

for i = N-1 to 0

  $R = \{R' << 1. A_i\}$

  $D = R - B$

  if $D < 0$, Q_i=0, $R' = R$

  else Q_i=1, $R' = D$

$R' = R$