Combinational Logic

Hsi-Pin Ma 馬席彬

https://eeclass.nthu.edu.tw/course/3452

Department of Electrical Engineering

National Tsing Hua University

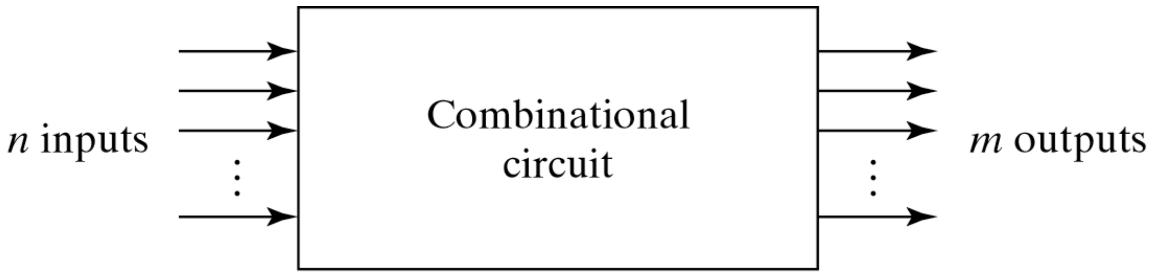
Outline

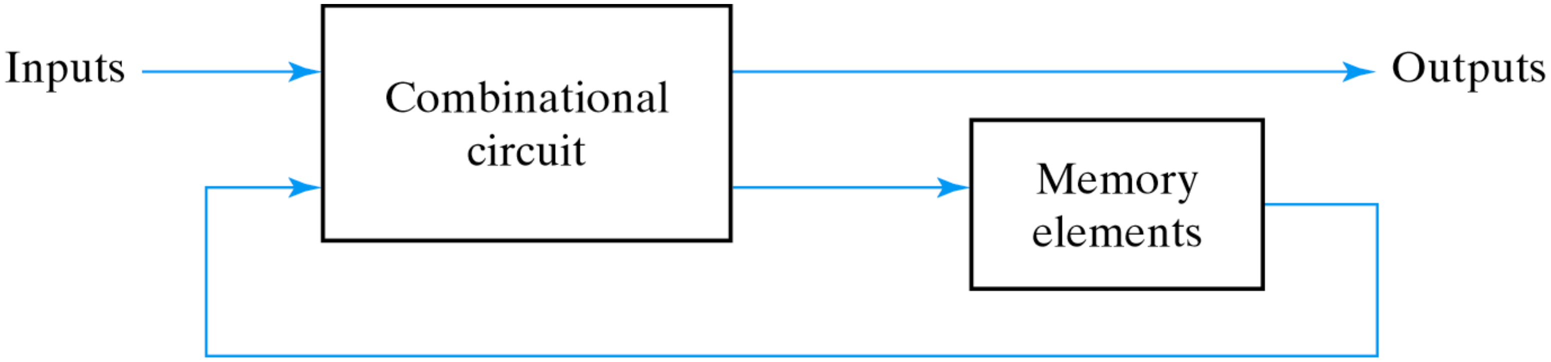
- Combinational Circuits
- Analysis of Combinational Circuits
- Design Procedure
- Binary Adder-Subtractor
- Decimal Adder
- Binary Multiplier
- Magnitude Comparator
- Decoder
- Encoders
- Arbiters
- Multiplexers
- Shifters

Combinational Circuits

Logic Circuits for the Digital System

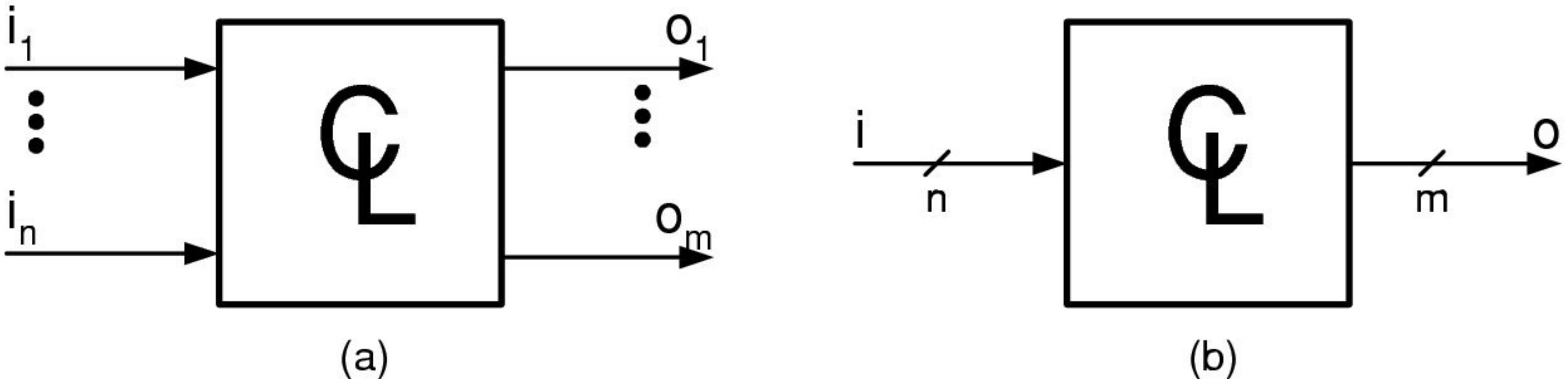
- Combinational circuits
  - Logic circuits whose outputs at any time are determined ***directly*** and ***only*** from the present ***input combination***.
- Sequential circuits
  - Circuits that employ memory elements + (combinational) logic gates
  - Outputs are determined from the present input combination as well as the state of the memory cells.





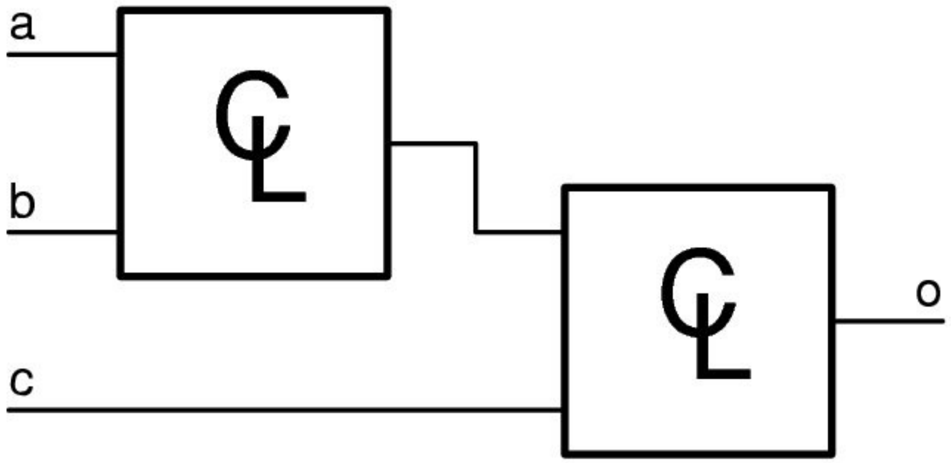
Combinational Logic Circuits

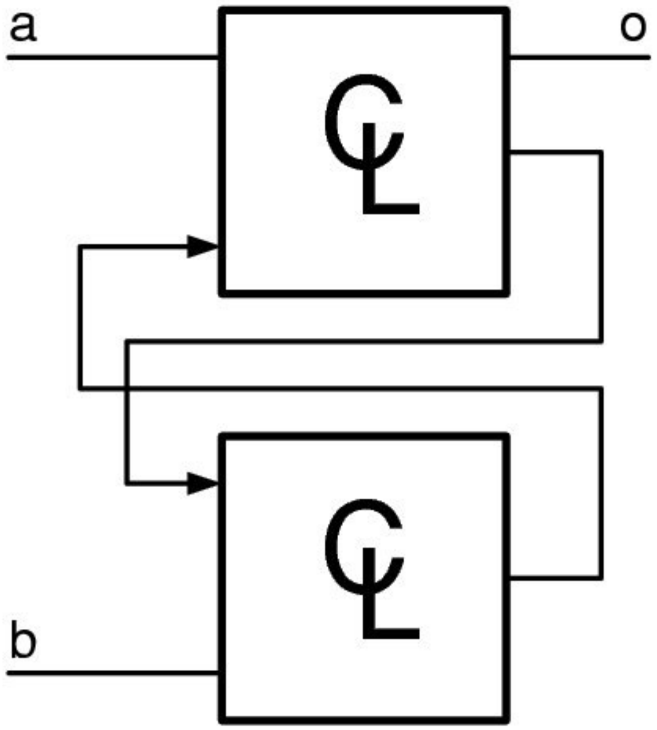
- Memoryless: o=f(i)
  - Used for control, arithmetic, and data steering.



Closure

- Combinational logic circuits are closed under *acyclic* composition
- Cyclic composition of two combinational logic circuits
  - The ***feedback*** variable can remember the ***history*** of the circuits
  - Sequential logic circuit





Analysis of Combinational Circuits

Analysis Procedure

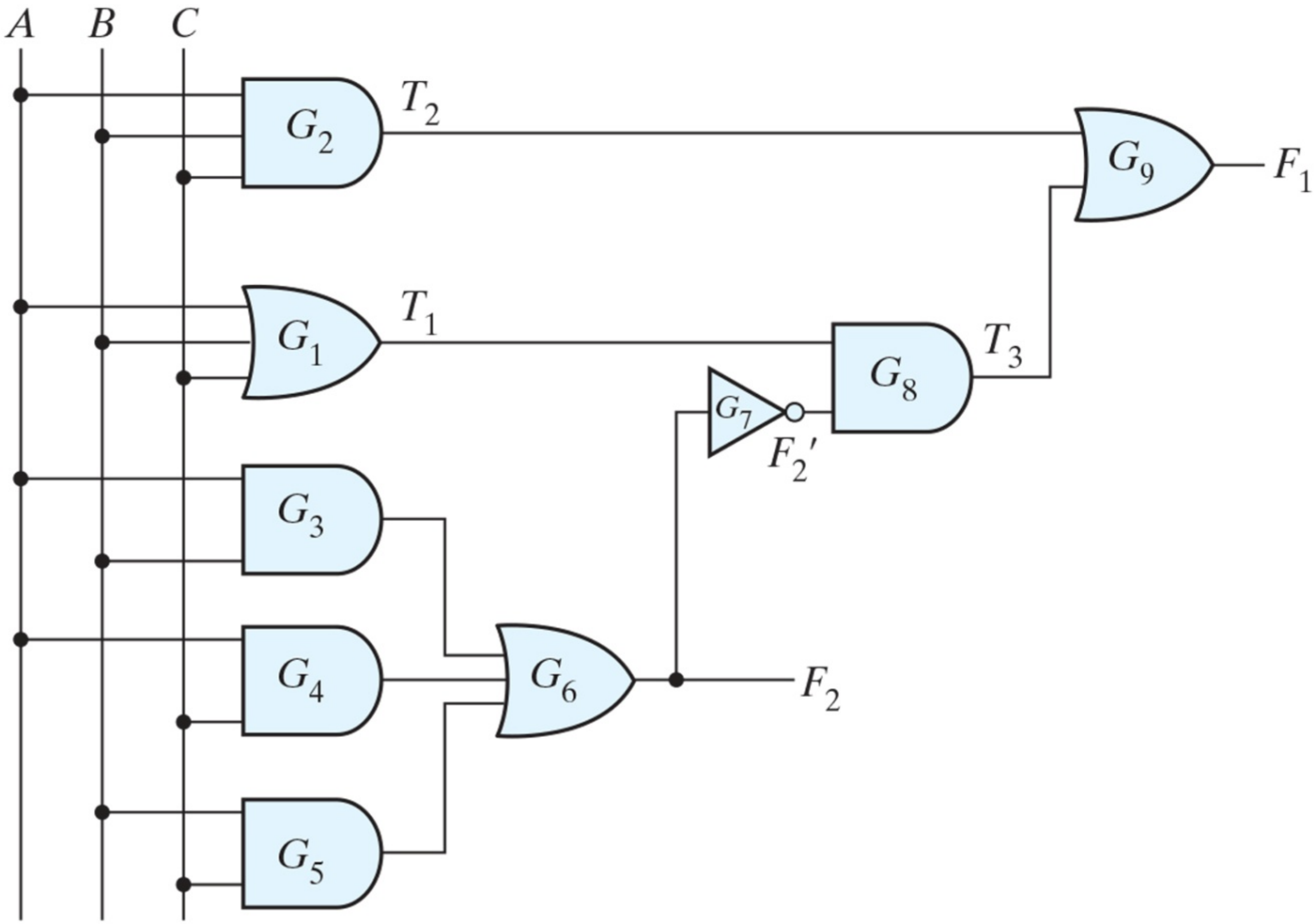
- Analysis for an available logic diagram
  - Make sure the given circuit is combinational
    - No ***feedback path*** or ***memory element***
  - Derive the corresponding ***Boolean functions***
  - Derive the corresponding ***truth table***
  - Verify and analyze the design
    - Logic simulation (waveforms)
  - Explain the function

Derivation of Boolean Functions (1/2)

- Label all gate outputs that are functions of the input variables only. Determine the functions.
- Label all gate outputs that are functions of the input variables and previously labeled gate outputs, and find the functions.
- Repeat previous step until all the primary outputs are obtained.

Derivation of Boolean Functions (2/2)

- Example
  - List all functions
    - $F_2=AB+AC+BC$
    - $T_1=A+B+C$
    - $T_2=ABC$
    - $T_3=F_2'T_1$
    - $F_1=T_3+T_2$



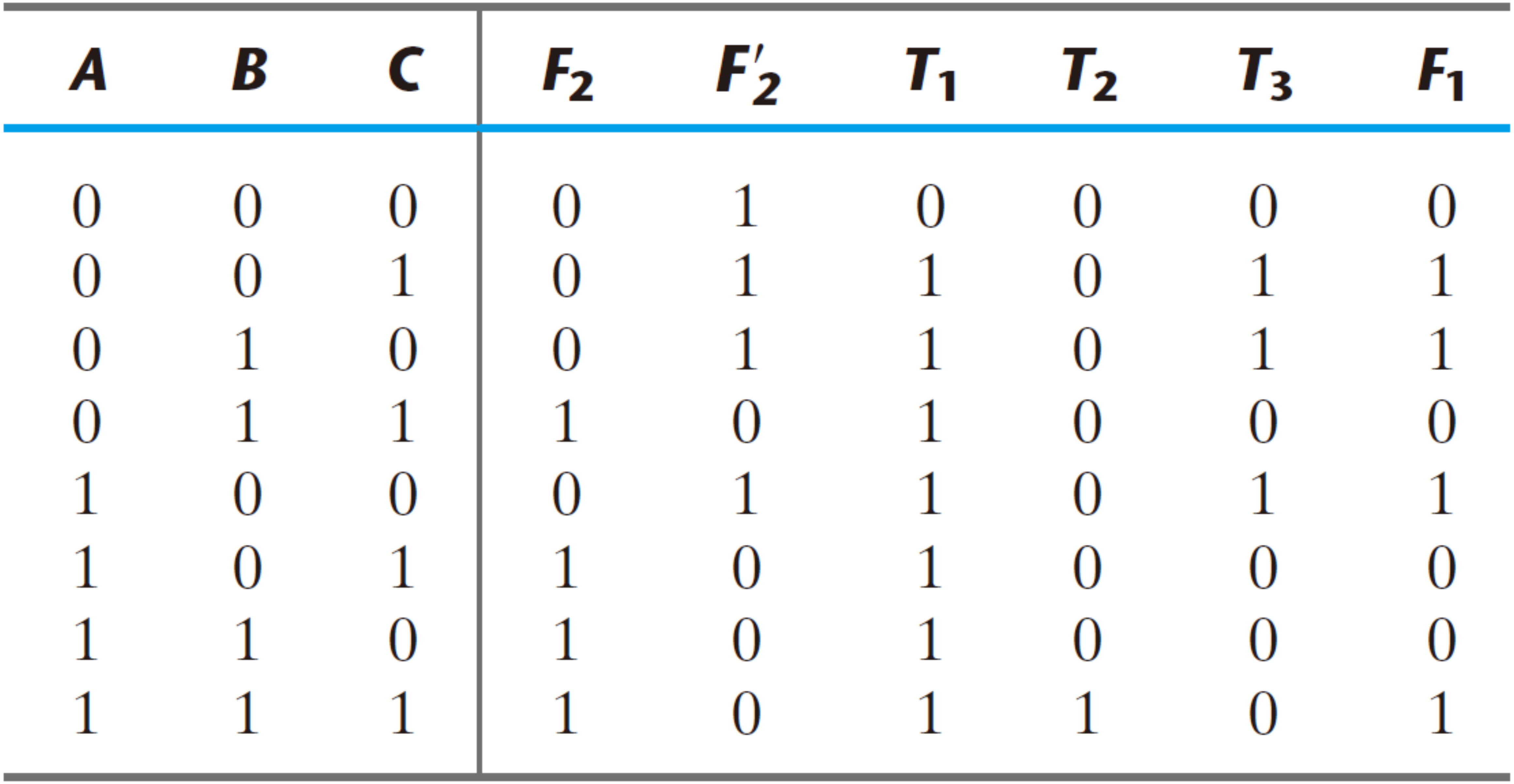
  - $F_1=T_3+T_2=F_2'T_1+ABC=(AB+AC+BC)'(A+B+C)+ABC$
    $=A'BC'+A'B'C+AB'C'+ABC$
  - Full adder (F_1: sum, F_2: carry)

Derivation of Truth Table (1/2)

- For *n* input variables
  - List all the 2^n input combinations from 0 to 2^n-1.
  - Partition the circuit into small single-output blocks and label the output of each block.
  - Obtain the truth table of the blocks depending on the input variables only.
  - Proceed to obtain the truth tables for other blocks that depend on previously defined truth tables.

Derivation of Truth Tables (2/2)

- Example



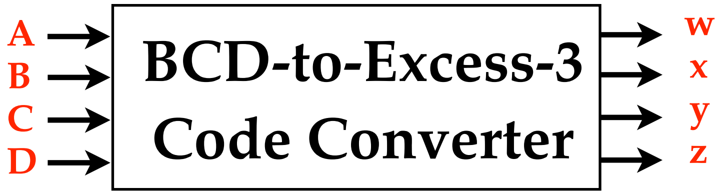
A	B	C	F_2	F'_2	T_1	T_2	T_3	F_1
0	0	0	0	1	0	0	0	0
0	0	1	0	1	1	0	1	1
0	1	0	0	1	1	0	1	1
0	1	1	1	0	1	0	0	0
1	0	0	0	1	1	0	1	1
1	0	1	1	0	1	0	0	0
1	1	0	1	0	1	0	0	0
1	1	1	1	0	1	1	0	1

Design Procedure

Design Procedure

- 1 **Specification**: From the *specifications*, determine the inputs, outputs, and their symbols.
- 2 **Formulation**: Derive the ***truth table*** (***functions***) from the relationship between the inputs and outputs
- 3 **Optimization**: Derive the ***simplified Boolean functions*** for each output function. Draw a logic diagram or provide a netlist for the resulting circuits using AND, OR, and inverters.
- 4 **Technology Mapping**: Transform the logic diagram or netlist to a new diagram or netlist using the available implementation technology.
- **Verification**: Verify the design.

A BCD-to-Excess-3 Code Converter (1/3)



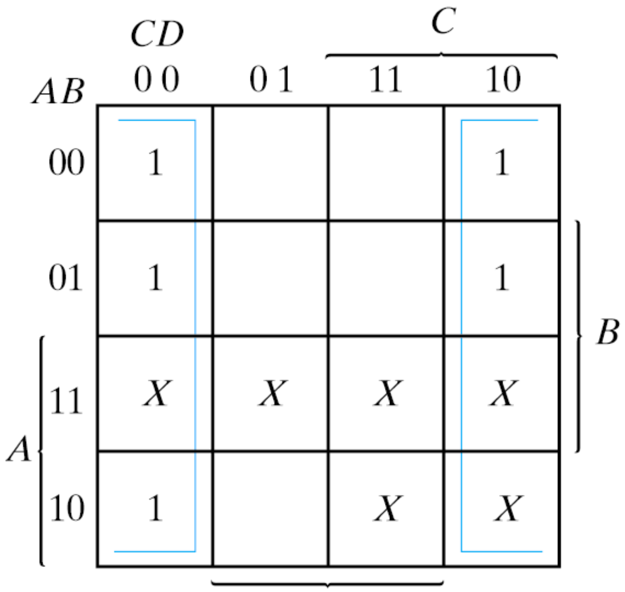
- Spec 1
  - input (ABCD), output (wxyz) (MSB to LSB)
  - ABCD: 0000 ~ 1001 (0~9)
- Formulation 2
  - wxyz = ABCD+0011

Input BCD				Output Excess-3 Code			
A	B	C	D	w	x	y	z
0	0	0	0	0	0	1	1
0	0	0	1	0	1	0	0
0	0	1	0	0	1	0	1
0	0	1	1	0	1	1	0
0	1	0	0	0	1	1	1
0	1	0	1	1	0	0	0
0	1	1	0	1	0	0	1
0	1	1	1	1	0	1	0
1	0	0	0	1	0	1	1
1	0	0	1	1	1	0	0
1	0	1	0	x	x	x	x
1	0	1	1	x	x	x	x
1	1	0	0	x	x	x	x
1	1	0	1	x	x	x	x
1	1	1	0	x	x	x	x
1	1	1	1	x	x	x	x

don't care

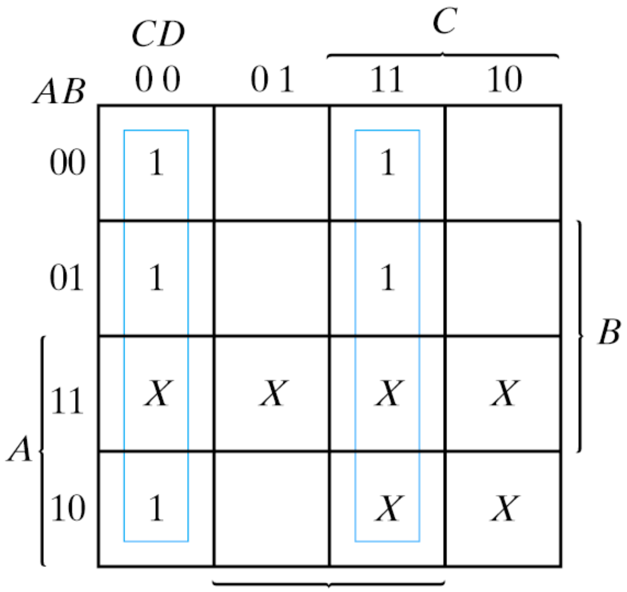
A BCD-to-Excess-3 Code Converter (2/3)

- Optimization 3



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

$z = D'$



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

$y = CD + C'D'$

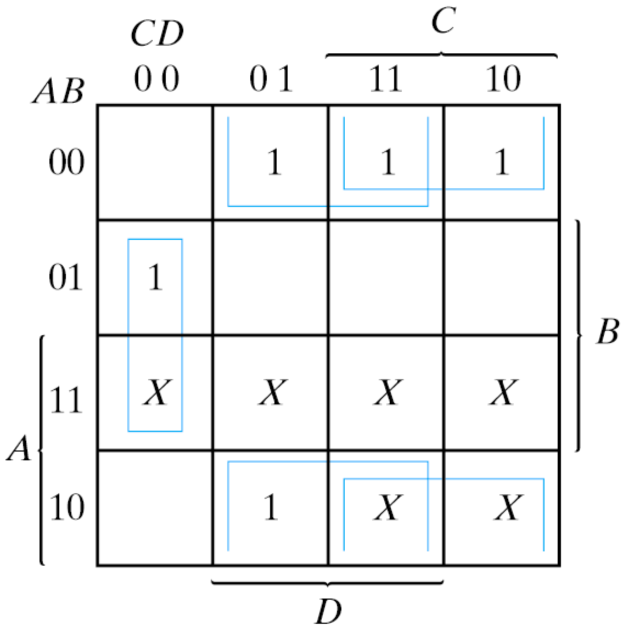
z=D′

y=CD+C′D′

x=B′C+B′D+BC′D′

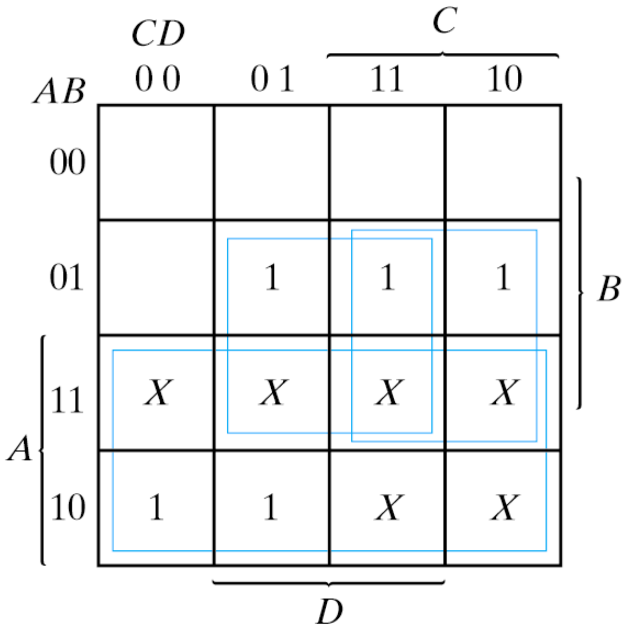
w=A+BC+BD

from K-map



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

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



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

$w = A + BC + BD$

z=D′

y=CD+(C+D)′

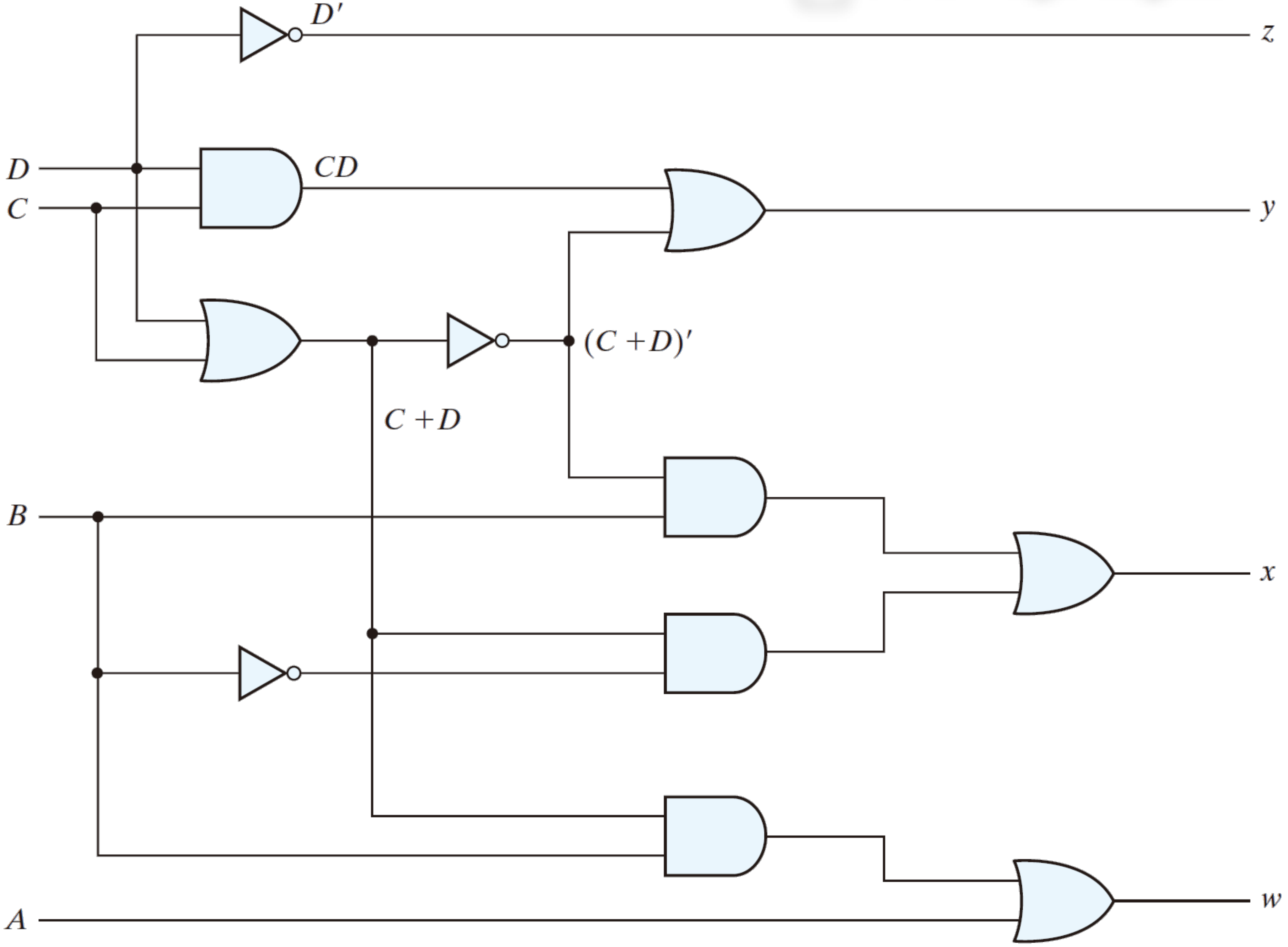
x=B′(C+D)+BC′D′

w=A+B(C+D)

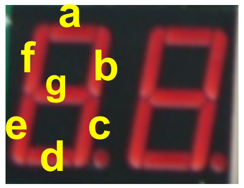
reduce gate numbers

A BCD-to-Excess-3 Code Converter (3/3)

4 4. Draw logic diagram



A BCD-to-Seven-Segment Display Decoder (1/2)



- Spec 1
  - input (ABCD), output (abcdefg) (MSB to LSB)
  - ABCD: 0000 ~ 1001 (0~9)
- Formulation 2

BCD Input				Seven-Segment Decoder						
A	B	C	D	a	b	c	d	e	f	g
0	0	0	0	1	1	1	1	1	1	0
0	0	0	1	0	1	1	0	0	0	0
0	0	1	0	1	1	0	1	1	0	1
0	0	1	1	1	1	1	1	0	0	1
0	1	0	0	0	1	1	0	0	1	1
0	1	0	1	1	0	1	1	0	1	1
0	1	1	0	1	0	1	1	1	1	1
0	1	1	1	1	1	1	0	0	0	0
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	1	0	1	1
All other inputs				0	0	0	0	0	0	0

A BCD-to-Seven-Segment Decoder (2/2)

- Optimization 3
  - 7x K-Map simplification
  - a=A’C+A’BD+B’C’D’+A’B’C’
  - b=A’B’+A’C’D’+A’CD+AB’C’
  - c=A’B+A’D+B’C’D’+AB’C’
  - d=A’CD’+A’B’C+B’C’D’+AB’C’+A’BC’D
  - e=A’CD’+B’C’D’
  - f=A’BC’+A’C’D’+A’BD’+AB’C’
  - f=A’CD’+A’B’C+A’BC’+AB’C’
- Technology Mapping 4

Binary Adder-Subtractor

Binary Half Adder & Full Adder (1/3)

- Half adder
  - Inputs: x, y
  - Outputs: C (carry), S(sum)

1

2

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

3

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

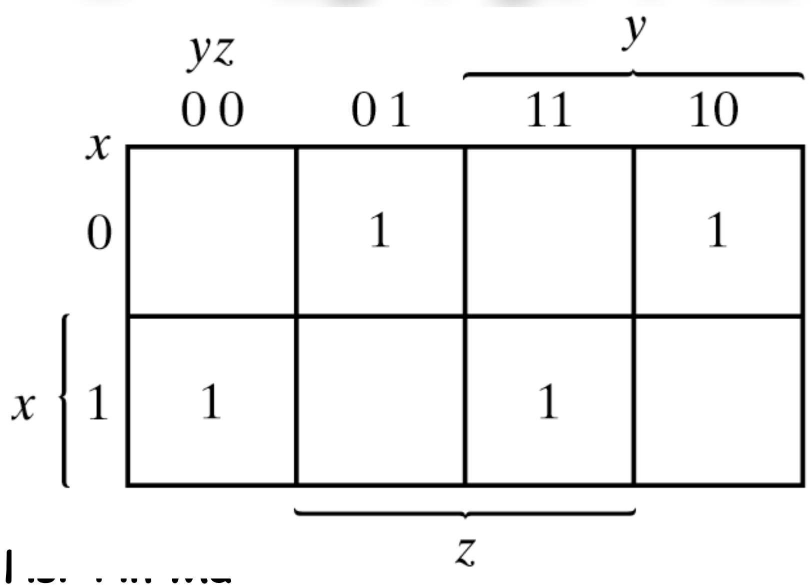
$$C = xy$$

- Full adder
  - Inputs: x, y, z(carry from previous lower significant bit)
  - Outputs: C(carry), S(sum)

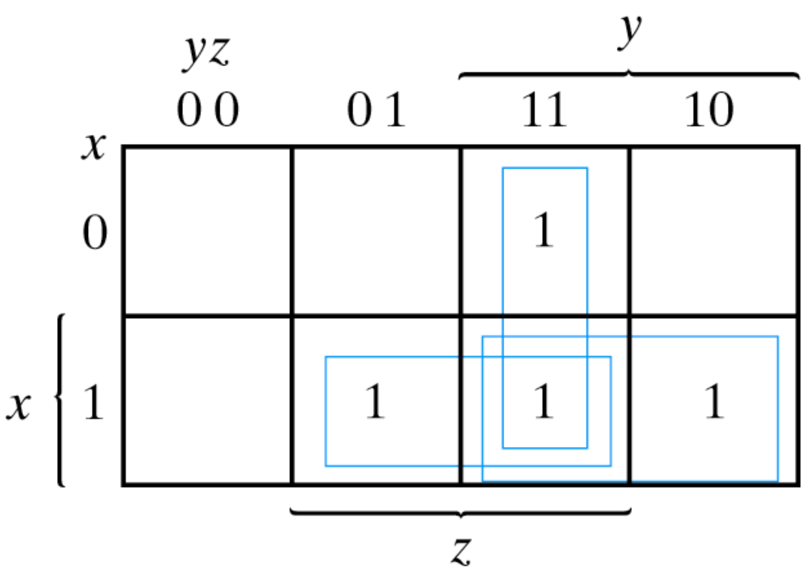
1

$$S = x'y'z + x'yz' + xy'z' + xyz = x \oplus y \oplus z$$

$$C = xy + yz + zx$$



3



2

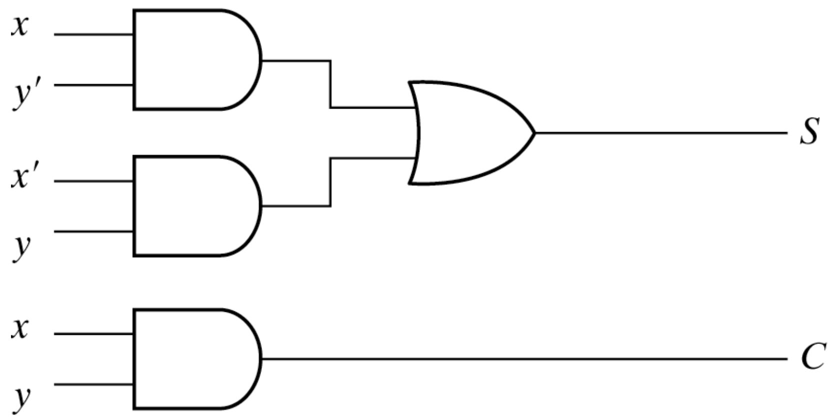
x	y	z	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

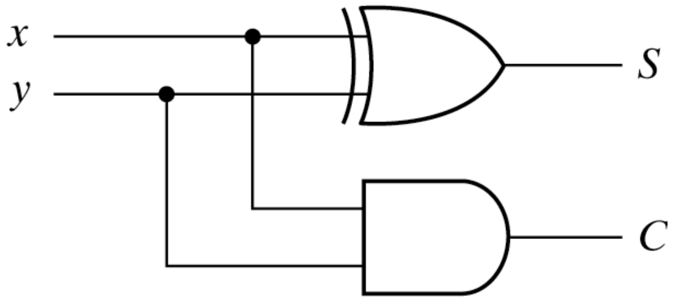
Binary Half Adder & Full Adder (2/3)

- Logic diagram $\boxed{4}$

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

$$C = xy$$

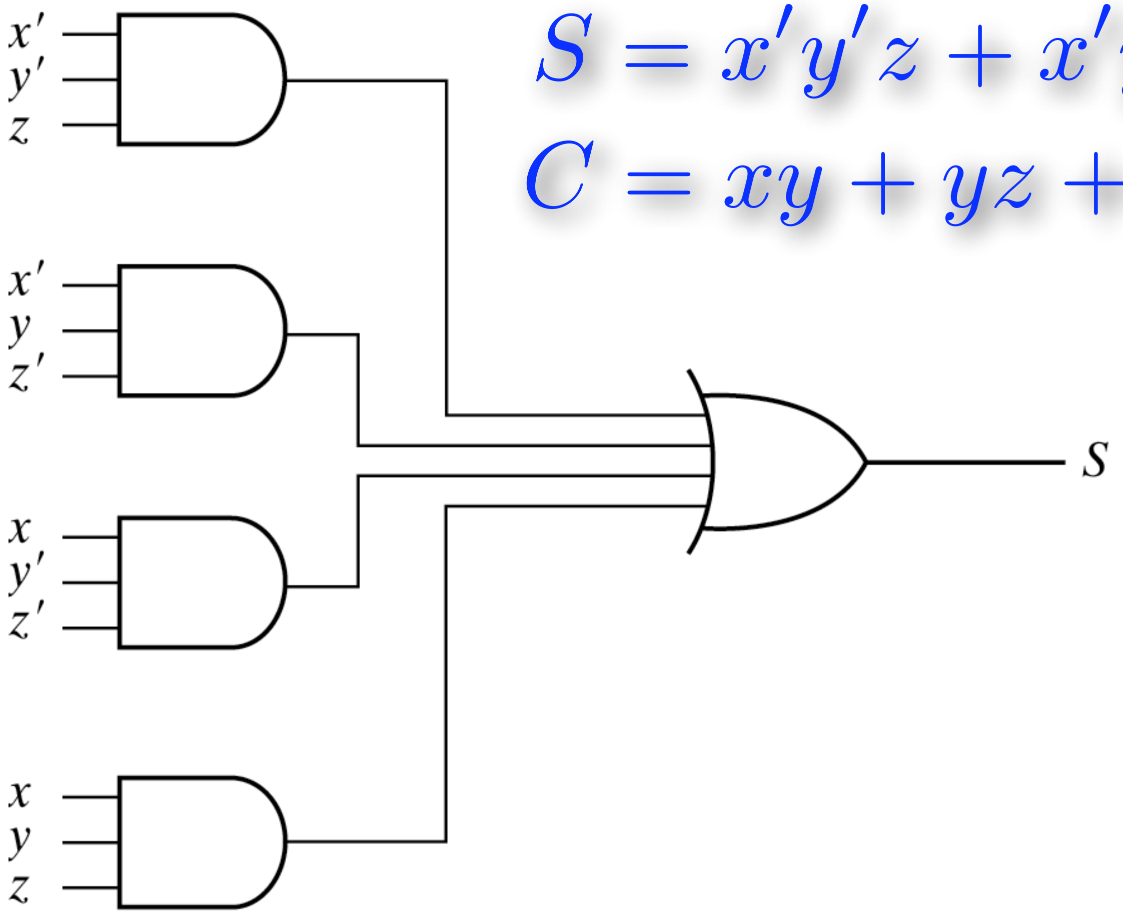


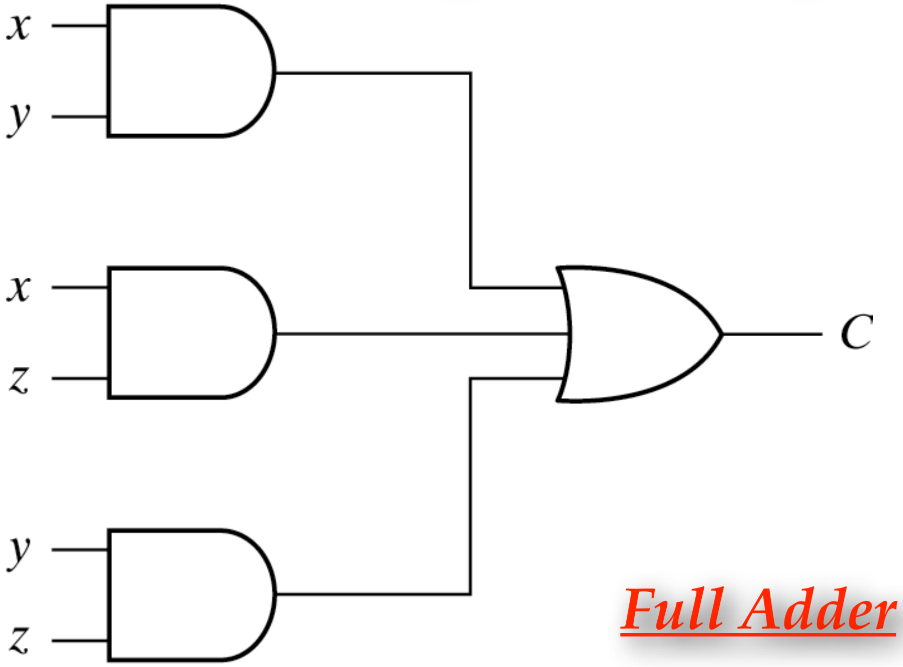


Half Adder

$$S = x'y'z + x'yz' + xy'z' + xyz = x \oplus y \oplus z$$

$$C = xy + yz + zx$$





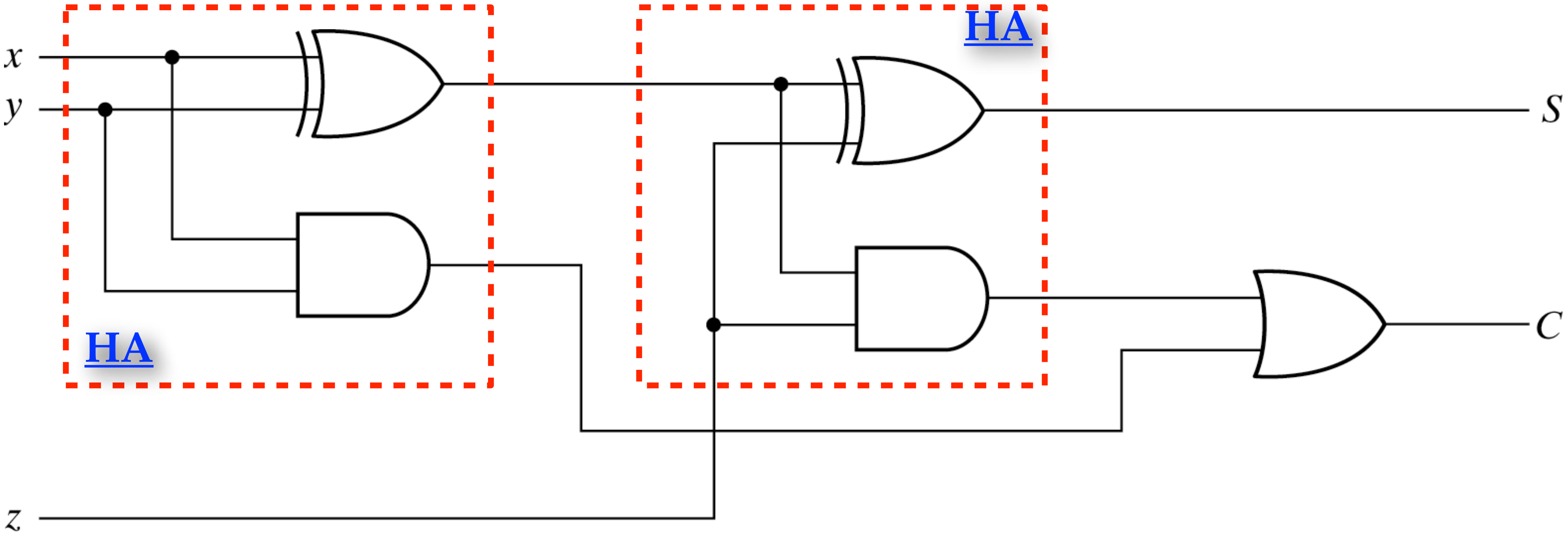
Full Adder

Binary Half Adder & Full Adder (3/3)

- Full adder implemented with half adders
  - Two half adders and one OR gate

$$S = z \oplus (x \oplus y)$$

$$C = z(xy' + x'y) + xy$$

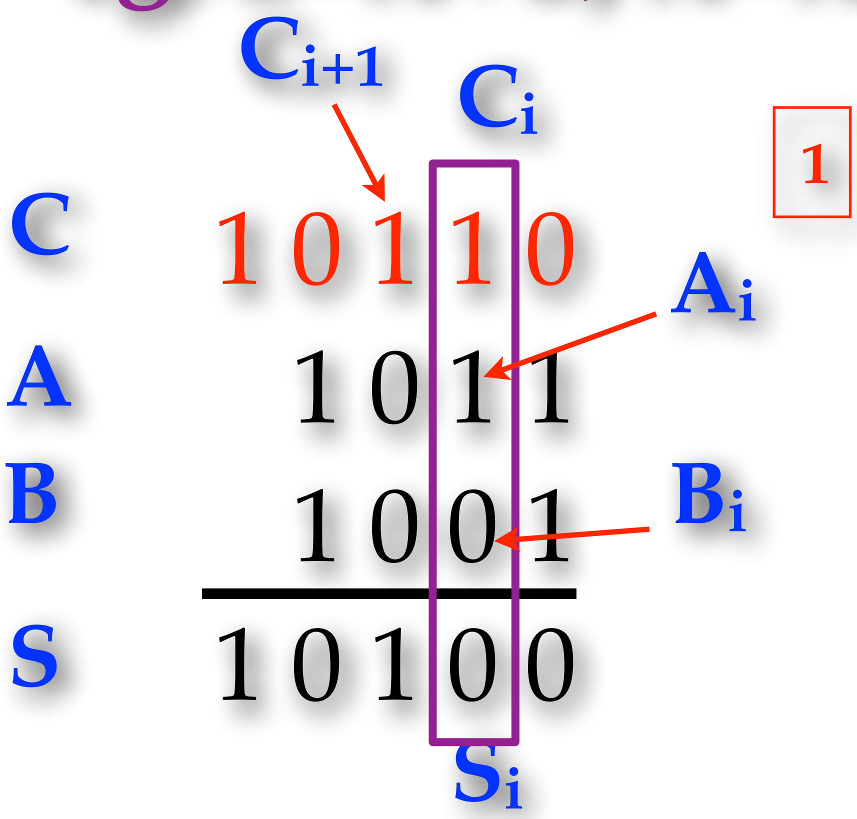


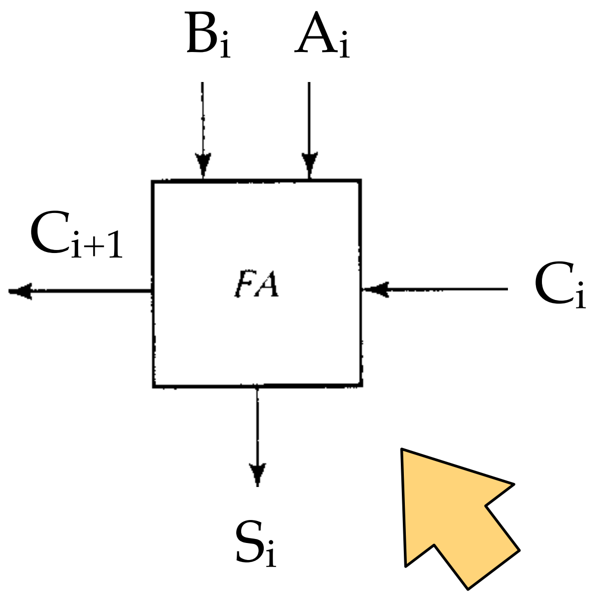
Ripple-Carry Adder (1/4)

unsigned addition

$$(C_{n+1}S_nS_{n-1}\ldots S_1)=(A_nA_{n-1}\ldots A_1)+(B_nB_{n-1}\ldots B_1)$$

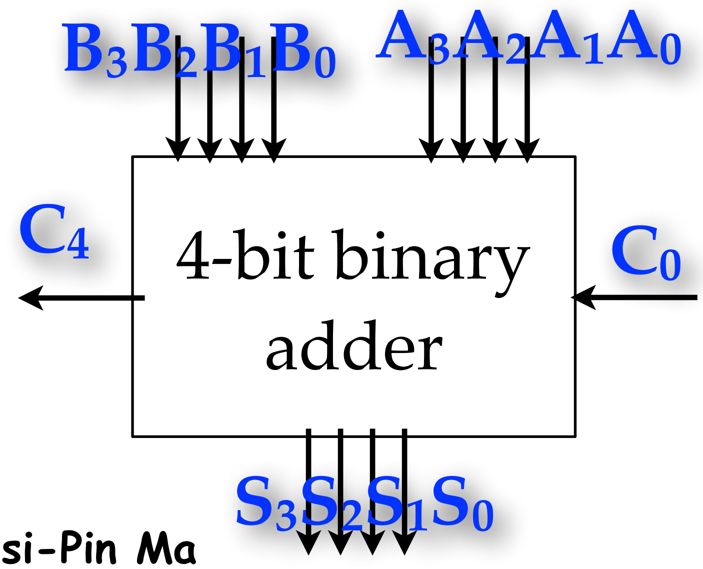
eg. $S=A+B$, $A=A_3A_2A_1A_0$, $B=B_3B_2B_1B_0$, $S=S_3S_2S_1S_0$

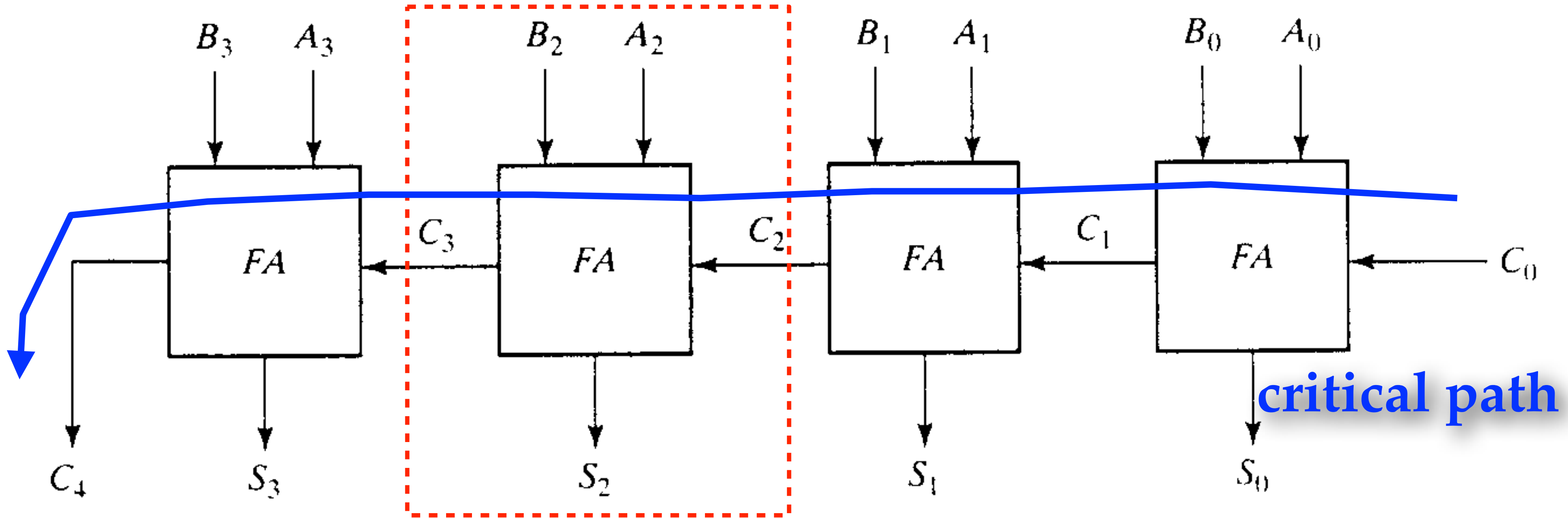




The computation time of a ripple-carry adder grows linearly with word length *n*

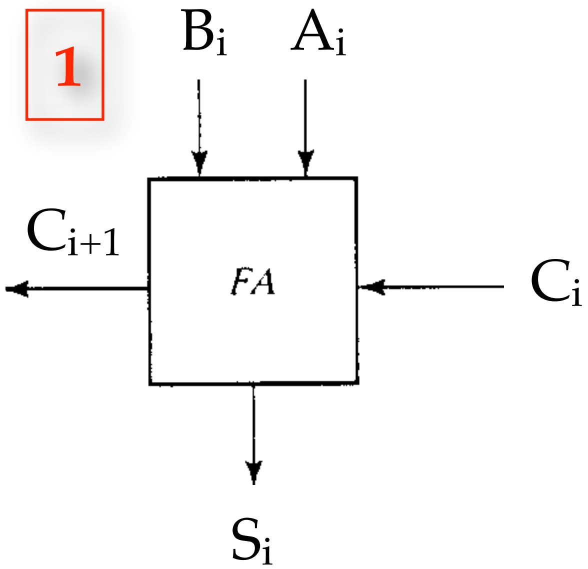
T=O(*n*) due to carry chain





Ripple-Carry Adder (2/4)

1



$C_i \backslash A_iB_i$	00	01	11	10
0	0	1 **1**	3	2 **1**
1	4 **1**	5	7 **1**	6

$$S_i = A_i \oplus B_i \oplus C_i$$

3

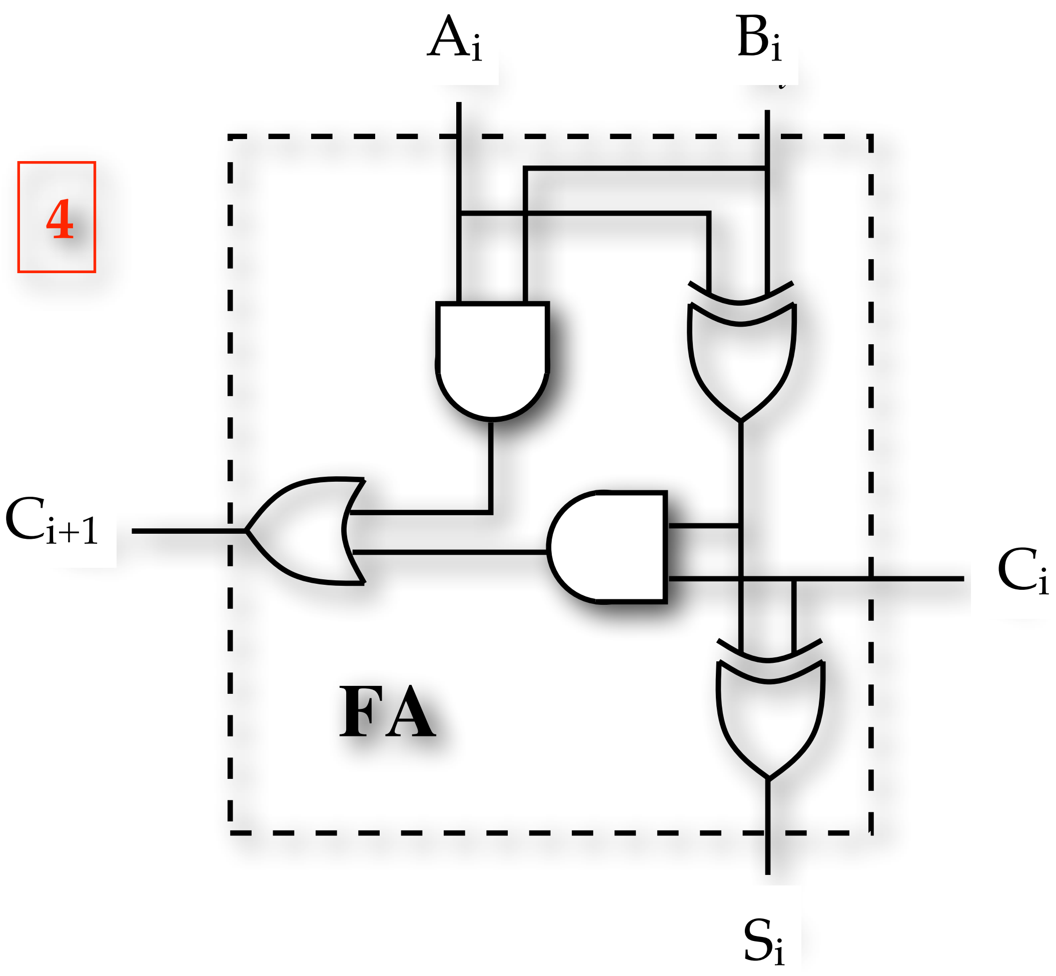
$C_i \backslash A_iB_i$	00	01	11	10
0	0	1	3 **1**	2
1	4	5 **1**	7 **1**	6 **1**

$$C_{i+1} = A_iB_i + C_i(A_i \oplus B_i)$$

2

A_i	B_i	C_i	C_{i+1}	S_i
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

4



Ripple-Carry Adder (3/4)

define

$$S_i = f(A_i, B_i, C_i) = A_i \oplus B_i \oplus C_i$$
$$C_{i+1} = g(A_i, B_i, C_i) = A_i \cdot B_i + B_i \cdot C_i + C_i \cdot A_i$$

$$S_0 = f(A_0, B_0, C_0)$$
$$C_1 = g(A_0, B_0, C_0)$$

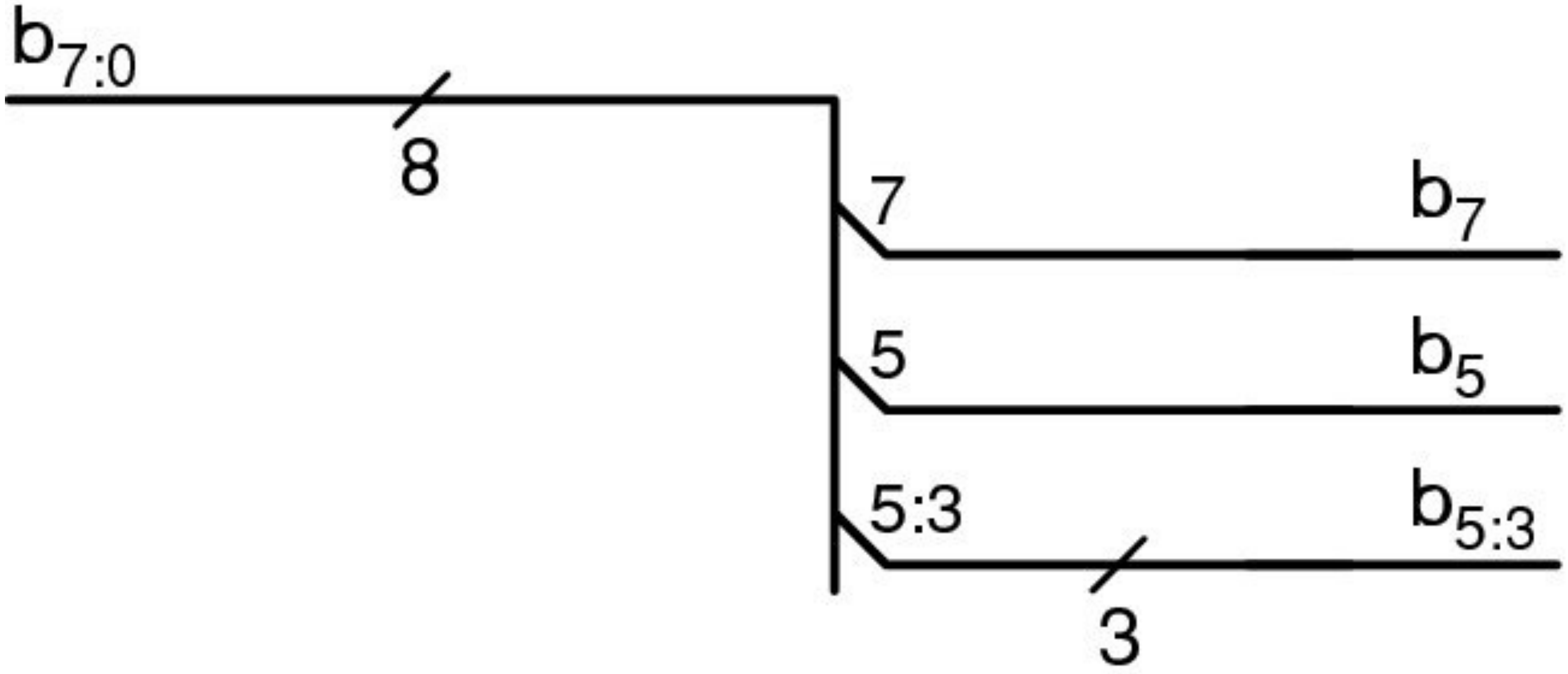
$$S_1 = f(A_1, B_1, C_1)$$
$$C_2 = g(A_1, B_1, C_1)$$

$$S_2 = f(A_2, B_2, C_2)$$
$$C_3 = g(A_2, B_2, C_2)$$

$$S_3 = f(A_3, B_3, C_3)$$
$$C_4 = g(A_3, B_3, C_3)$$

Multi-bit Notation

- Multi-bit signal or a bus



- Verilog bit-select (bit-slice) or part-select
  - b[7:0]
  - b[7]
  - b[5:3]

Carry Lookahead Adder (1/3)

- For a full adder, define what happens to carry

1

- Carry-generate: C_{out}=1 independent of C_{in}
  - $G_i = A_i \cdot B_i$
- Carry-propagate: C_{out}=C_{in}
  - $P_i = A_i \oplus B_i$
- Carry-kill: C_{out}=0 independent of C_{in}
  - $K_i = A_i' \cdot B_i'$

2

A_i	B_i	G_i	P_i	K_i
0	0	0	0	1
0	1	0	1	0
1	0	0	1	0
1	1	1	0	0

- Use the above info

3

- $C_{i+1} = A_iB_i + B_iC_i + A_iC_i = A_iB_i + (A_i + B_i)C_i = \underline{G_i + P_iC_i}$
- $S_i = A_i \oplus B_i \oplus C_i = \underline{P_i \oplus C_i}$

Carry Lookahead Adder (2/3)

- Do not have to wait for C_i to compute C_{i+1}
  - $C_{i+1} = G_i + P_i C_i$
  - $C_{i+2} = G_{i+1} + P_{i+1} C_{i+1} = G_{i+1} + P_{i+1} G_i + P_{i+1} P_i C_i$
  - $C_{i+3} = G_{i+2} + P_{i+2} C_{i+2} = G_{i+2} + P_{i+2} G_{i+1} + P_{i+2} P_{i+1} G_i + P_{i+2} P_{i+1} P_i C_i$

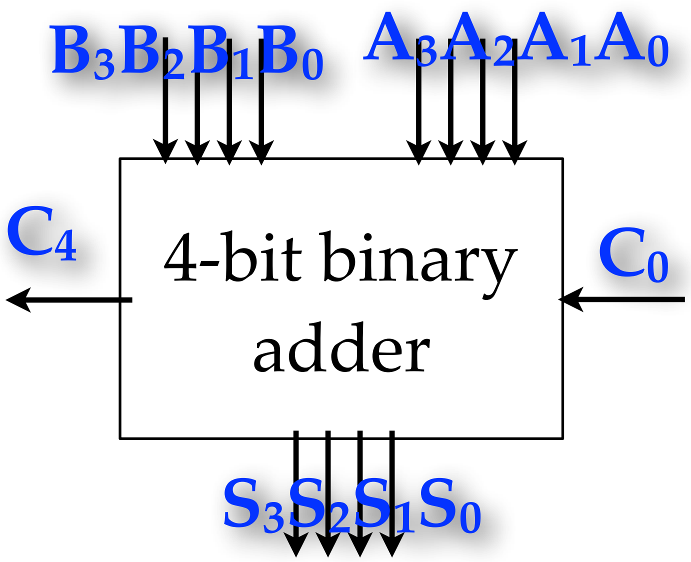
3

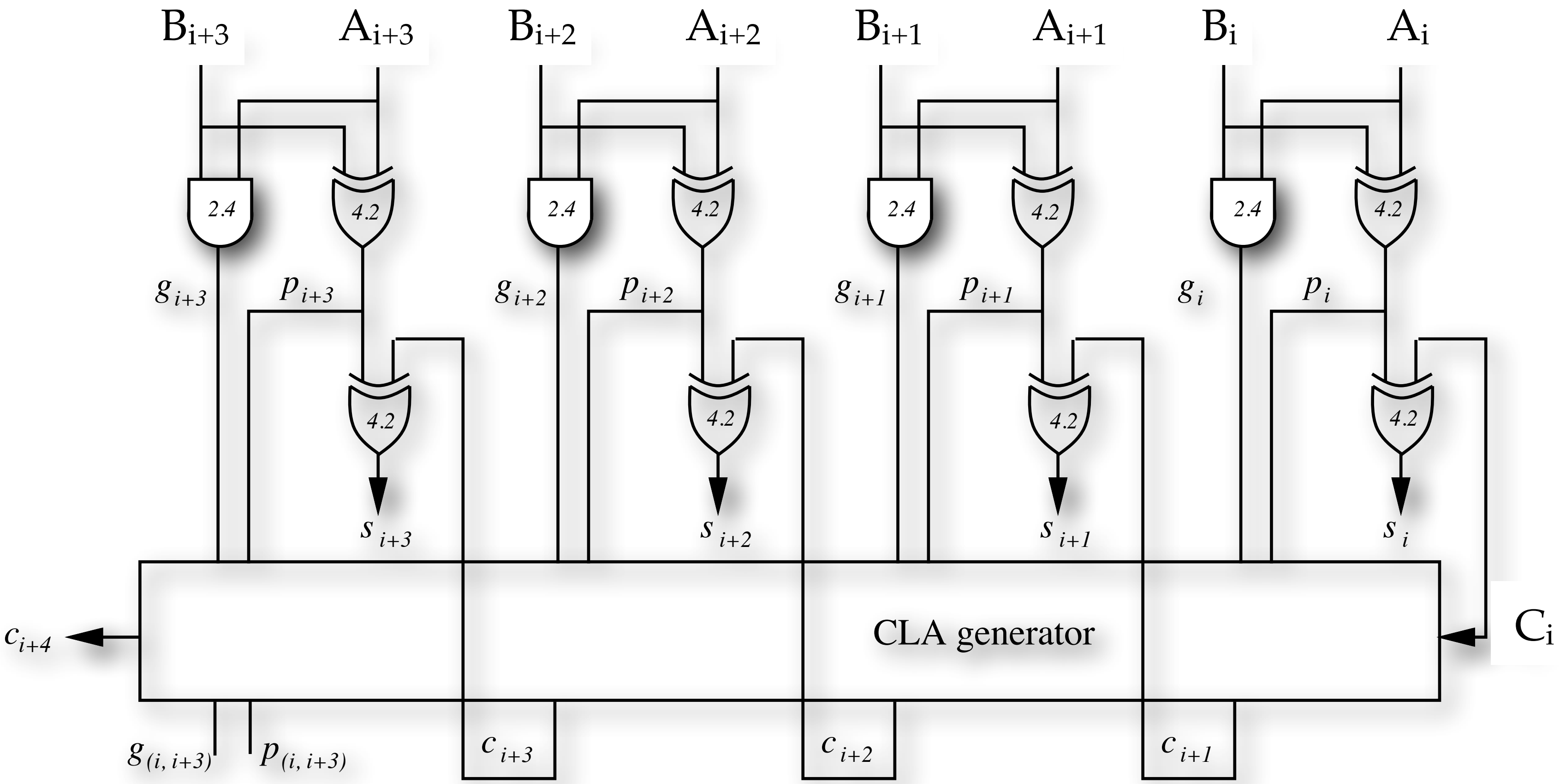
$$C_{i+4} = G_{i+3} + P_{i+3} C_{i+3} = G_{i+3} + P_{i+3} G_{i+2} + P_{i+3} P_{i+2} G_{i+1} + P_{i+3} P_{i+2} P_{i+1} G_i + P_{i+3} P_{i+2} P_{i+1} P_i C_i$$

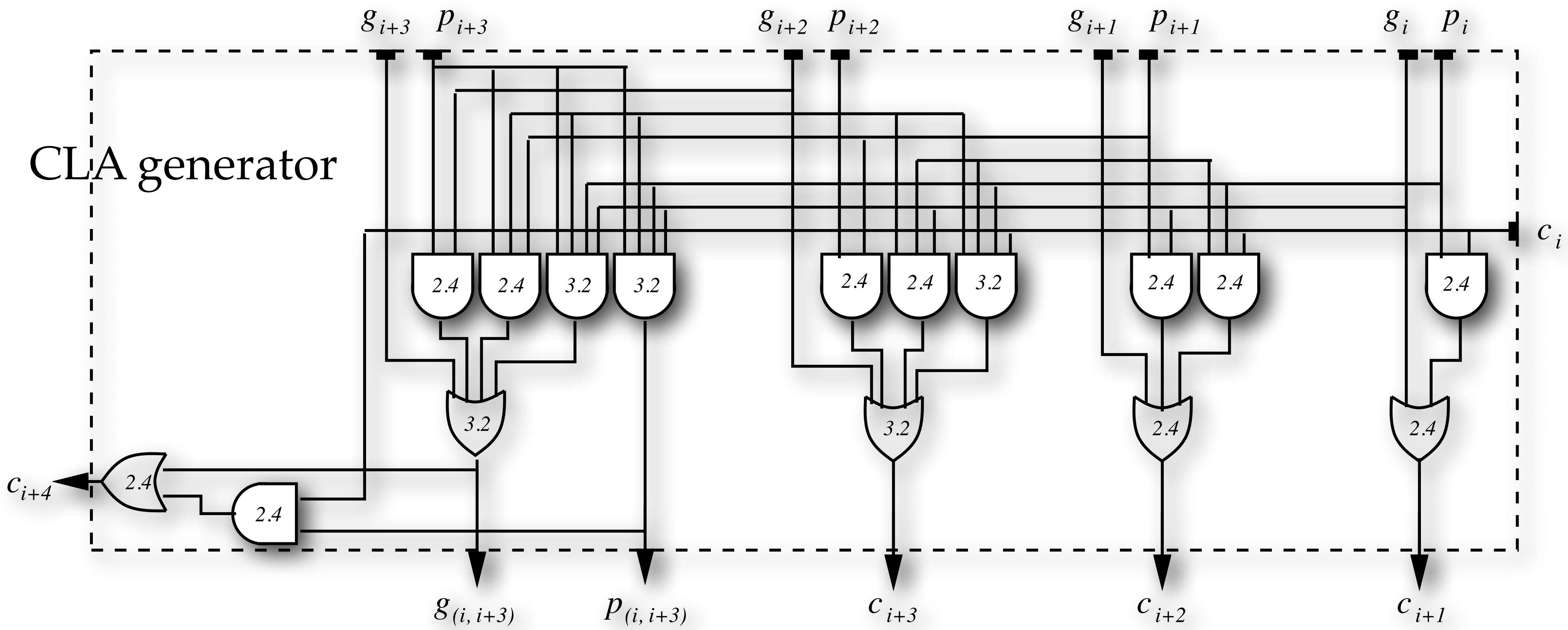
- Fixed delay time for each carry (but not the same for every gate!)
- Fanout of G_i & P_i also affect the overall delay => usually be limited to 4 bits

Carry Lookahead Adder (3/3)

4







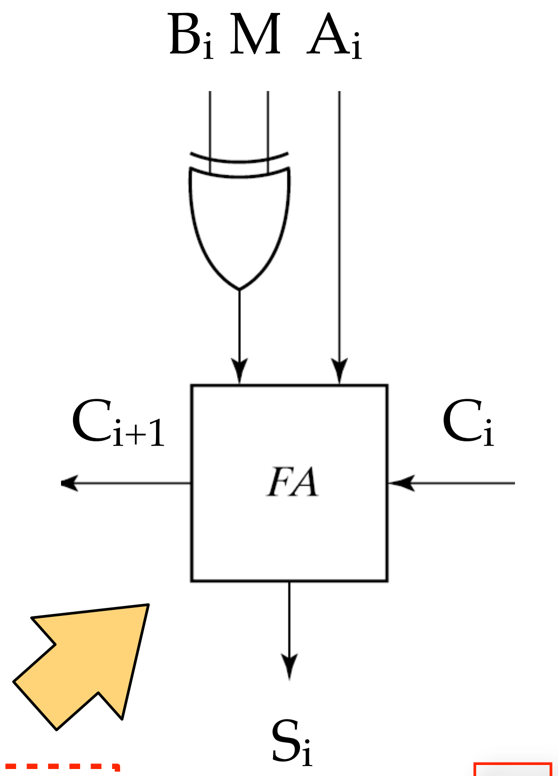
Binary Adders/Subtractors

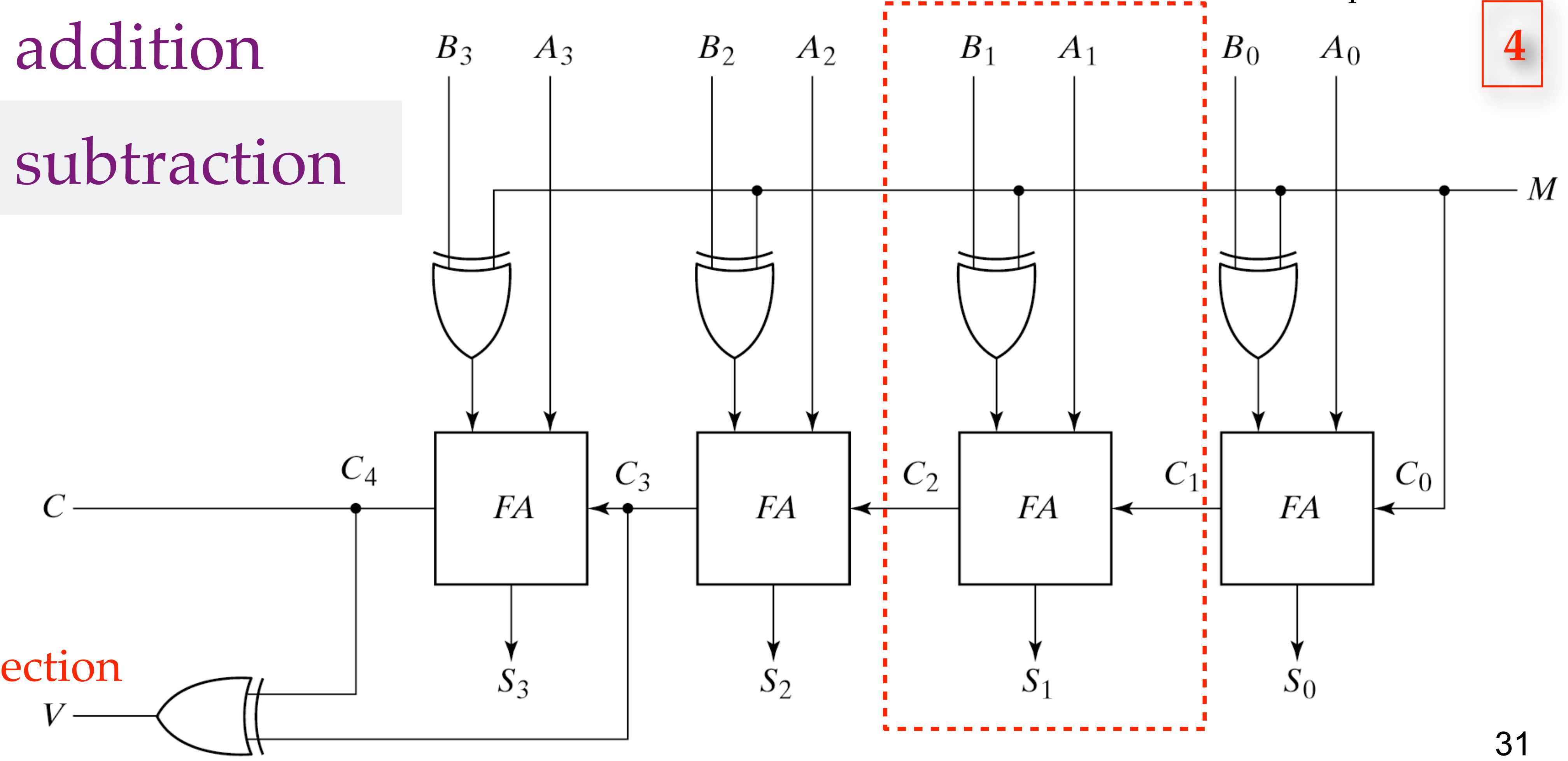
1

- Binary subtraction normally is performed by adding the minuend to the 2's complement of the subtrahend.

M	Function	Comments
0	S=A+B	addition
1	S=A+B'+1	subtraction

2 3





overflow detection

Decimal Adder

Decimal Adders (1/3)

- Addition of 2 decimal digits in BCD
  - $\{C_{out},S\}=A+B+C_{in}$
    - 1 $S=S_8S_4S_2S_1$, $A=A_8A_4A_2A_1$, $B=B_8B_4B_2B_1$
  - A digit in BCD cannot exceed 9, add 6 (0110) for final correction.

2 3

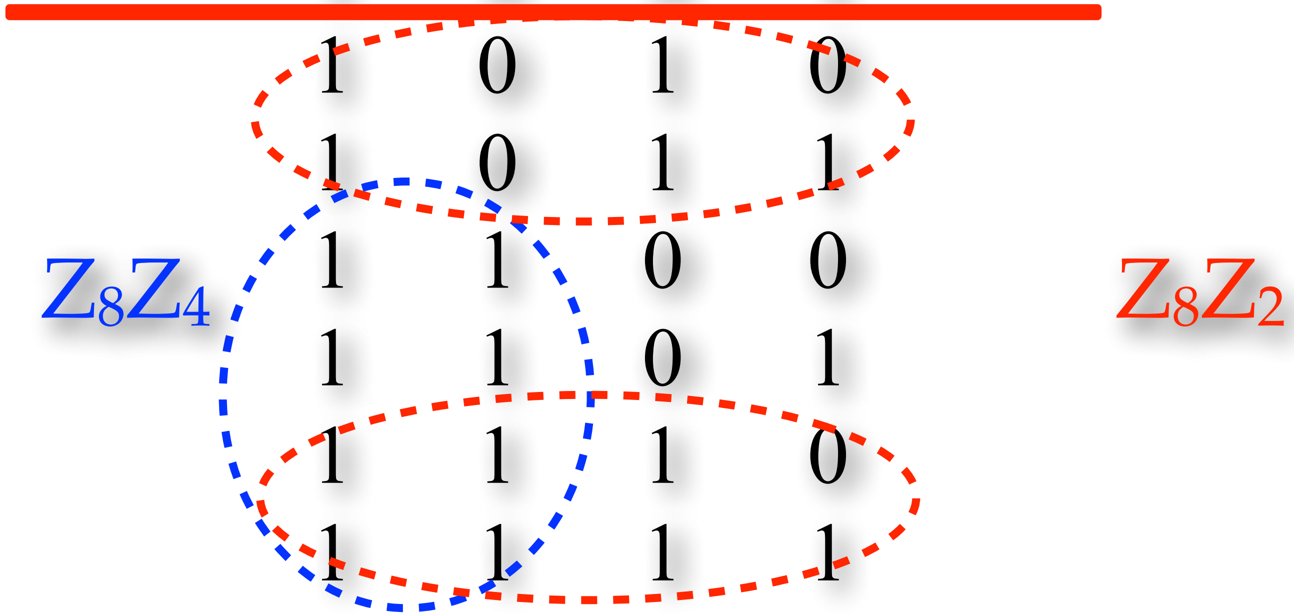
1 0		1 0 0 0 0	
8_{10}	A	$1\,0\,0\,0_2$	
9_{10}	B	$1\,0\,0\,1_2$	
$1\,7_{10}$	KZ	$1\,0\,0\,0\,1_2$	binary coded results
		$0\,1\,1\,0_2$	if >9, add 6
		$0\,0\,0\,1\,0\,1\,1\,1_2$	BCD coded results

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

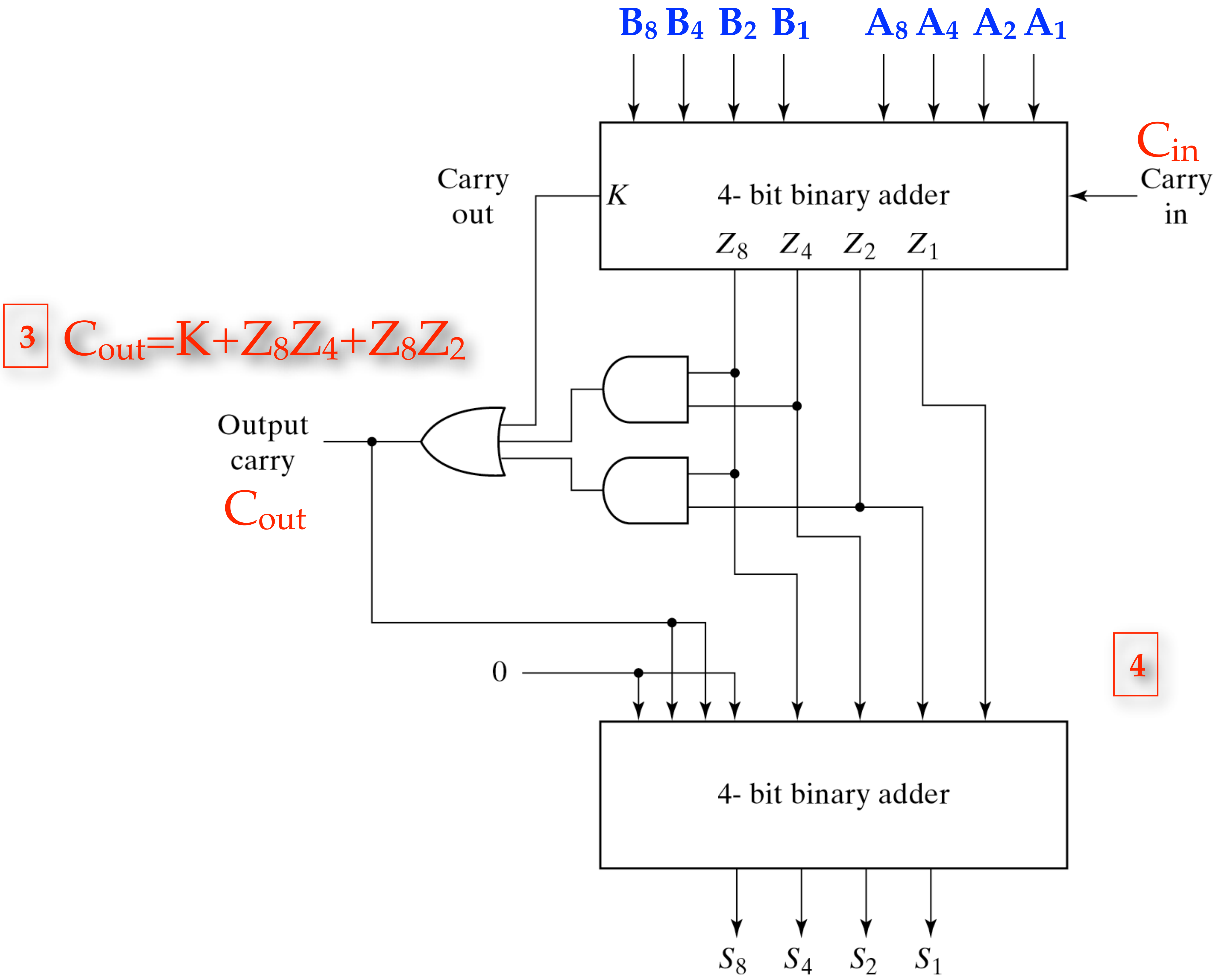
Decimal Adders (2/3)

2 3

Z_8	Z_4	Z_2	Z_1
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
1	0	1	0
1	0	1	1
1	1	0	0
1	1	0	1
1	1	1	0
1	1	1	1



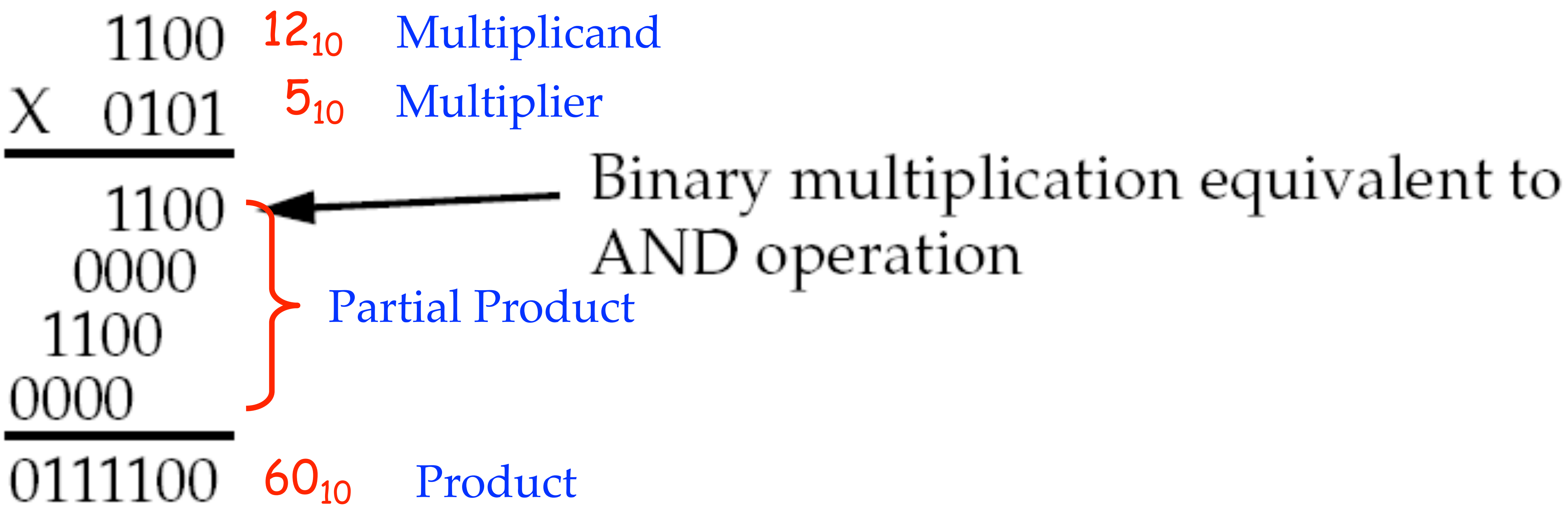
Decimal Adders (3/3)



Binary Multiplier

Multiplication

- Multiplication consists of
  - Generation of partial products
  - Accumulation of shifted partial products



1100 12_{10} Multiplicand

X 0101 5_{10} Multiplier

1100

0000

1100

0000

0111100 60_{10} Product

Partial Product

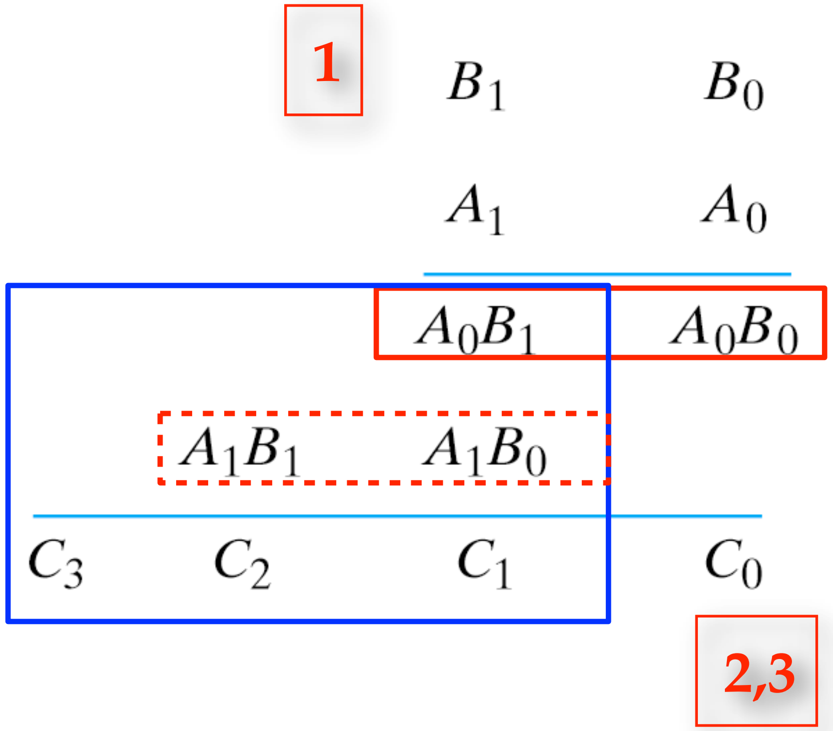
Binary multiplication equivalent to AND operation

M-bit x N-bit Multiplication

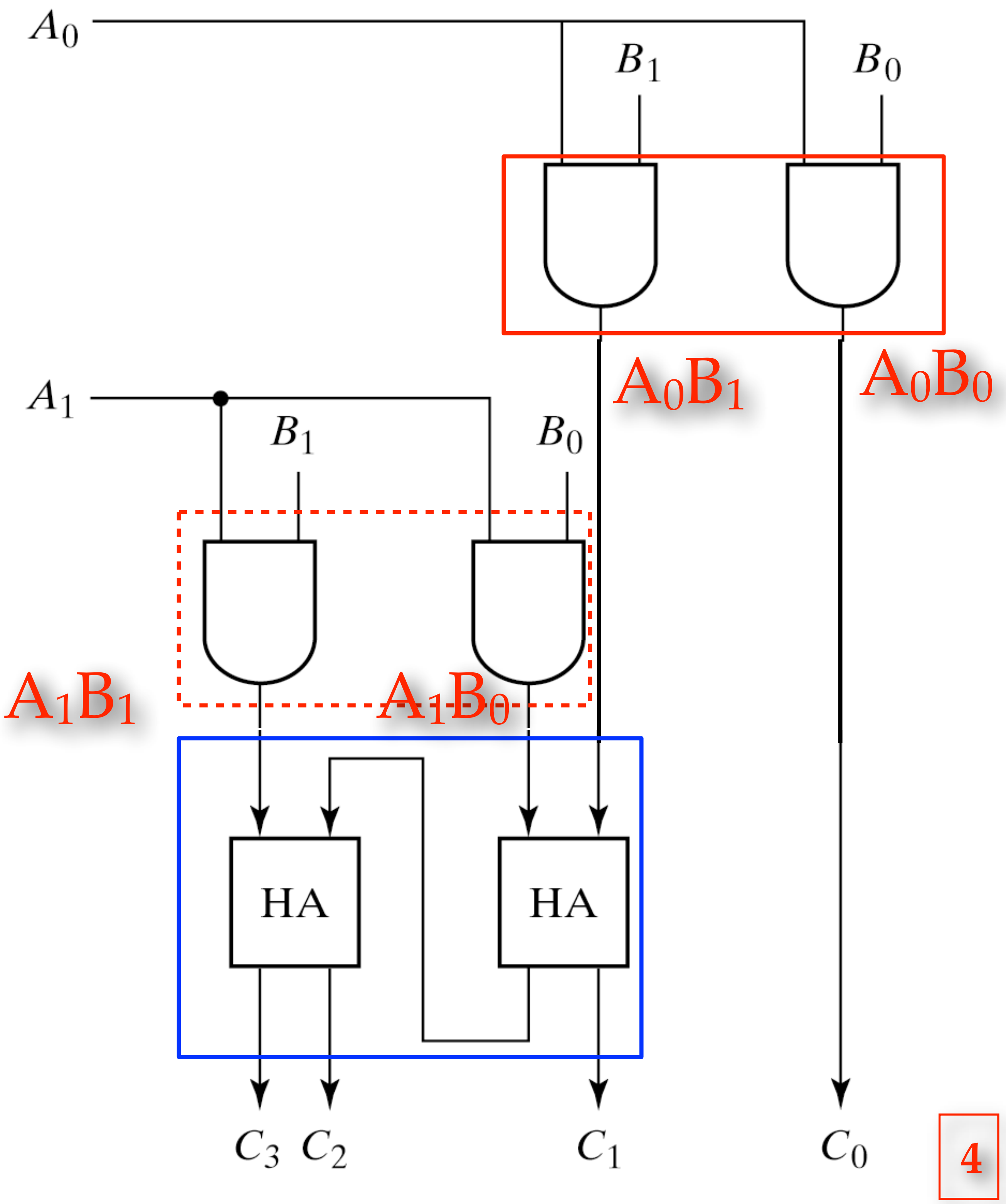
$$P = (\sum_{j=0}^{M-1} y_j 2^j)(\sum_{i=0}^{N-1} x_i 2^i) = \sum_{i=0}^{N-1} \sum_{j=0}^{M-1} x_i y_j 2^{i+j}$$

						y_5	y_4	y_3	y_2	y_1	y_0	Multiplicand
						x_5	x_4	x_3	x_2	x_1	x_0	Multiplier
						x_0y_5	x_0y_4	x_0y_3	x_0y_2	x_0y_1	x_0y_0	Partial Products
					x_1y_5	x_1y_4	x_1y_3	x_1y_2	x_1y_1	x_1y_0		
				x_2y_5	x_2y_4	x_2y_3	x_2y_2	x_2y_1	x_2y_0			
			x_3y_5	x_3y_4	x_3y_3	x_3y_2	x_3y_1	x_3y_0				
		x_4y_5	x_4y_4	x_4y_3	x_4y_2	x_4y_1	x_4y_0					
	x_5y_5	x_5y_4	x_5y_3	x_5y_2	x_5y_1	x_5y_0						
p_{11}	p_{10}	p_9	p_8	p_7	p_6	p_5	p_4	p_3	p_2	p_1	p_0	Product

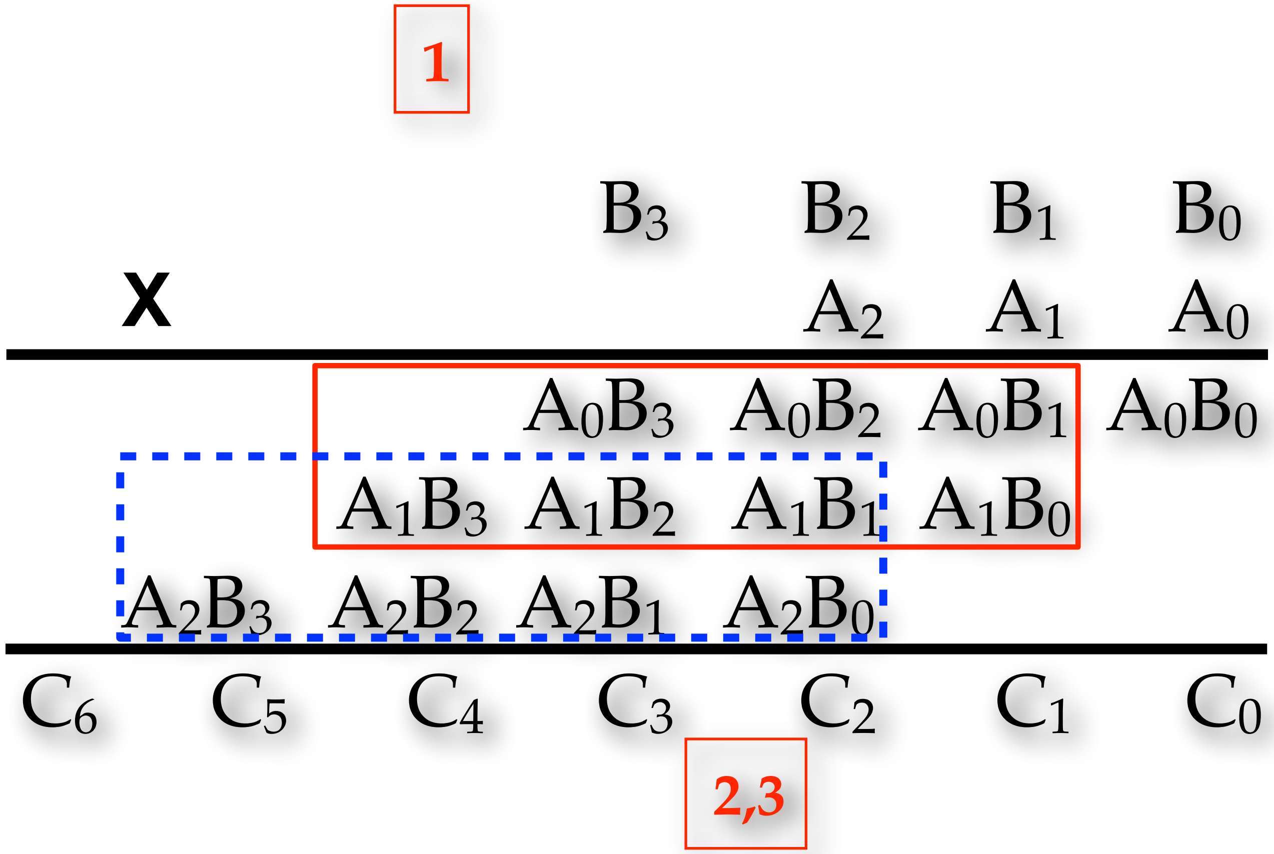
2-bit x 2-bit Binary Multiplier



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



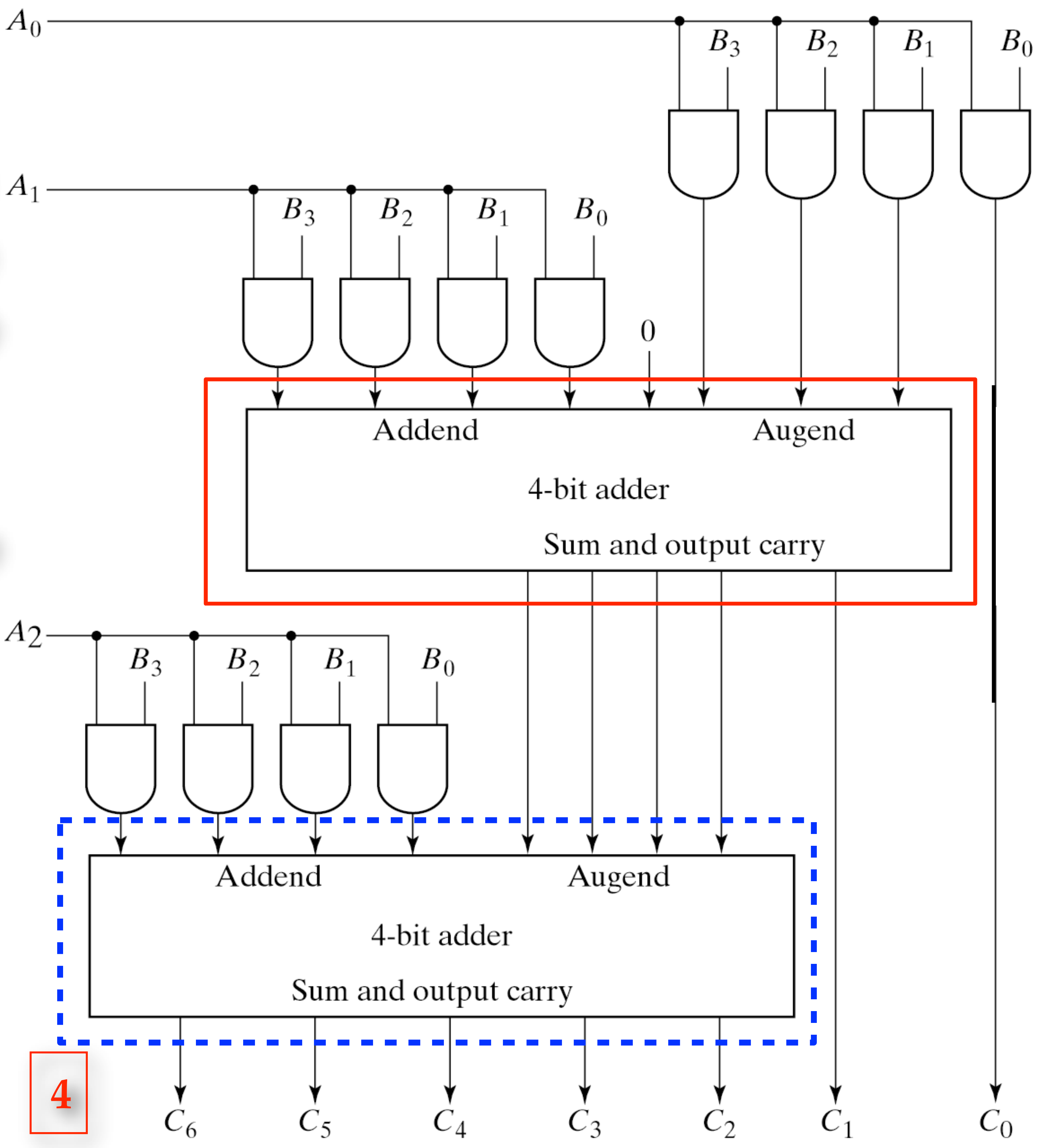
4-bit x 3-bit Binary Multiplier



1

			B_3	B_2	B_1	B_0
X				A_2	A_1	A_0
			A_0B_3	A_0B_2	A_0B_1	A_0B_0
		A_1B_3	A_1B_2	A_1B_1	A_1B_0	
	A_2B_3	A_2B_2	A_2B_1	A_2B_0		
C_6	C_5	C_4	C_3	C_2	C_1	C_0

2,3



Other Arithmetic Functions

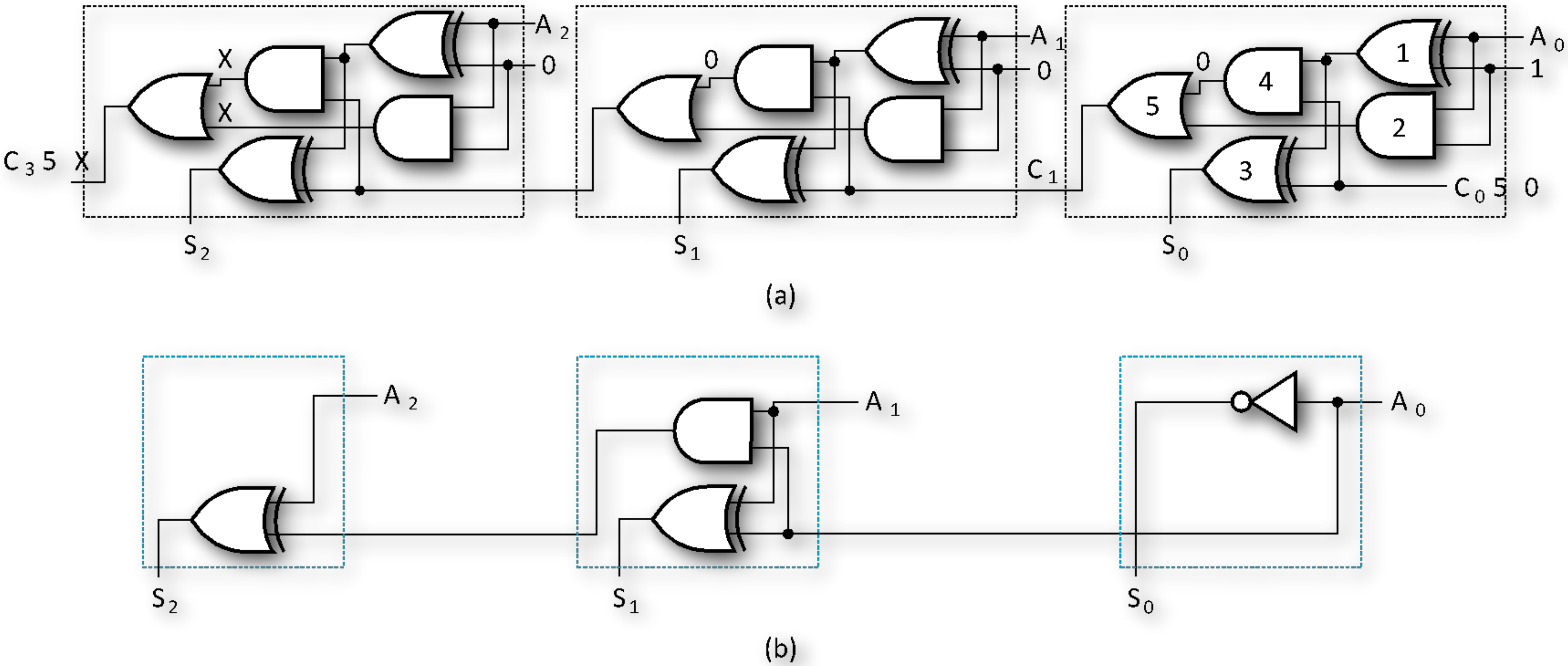
- It is convenient to design the functional blocks by ***contraction***
  - Removal of redundancy from circuit to which input fixing has been applied
- Functions
  - Increment
  - Decrement
  - Multiplication by constant
  - Division by constant
  - Zero fill and extension

Design by Contraction

- Simplify the logic in a functional block to implement a different function
  - The new function must be realizable from the original function by applying rudimentary functions to its inputs
  - Contraction is treated here only for application of 0s and 1s (not for X and X′).
  - After application of 0s and 1s, equations or the logic diagram are simplified

Design by Contraction Example

- Contraction of a ripple carry adder to incrementer for n=1 (Set B=001)



(a)

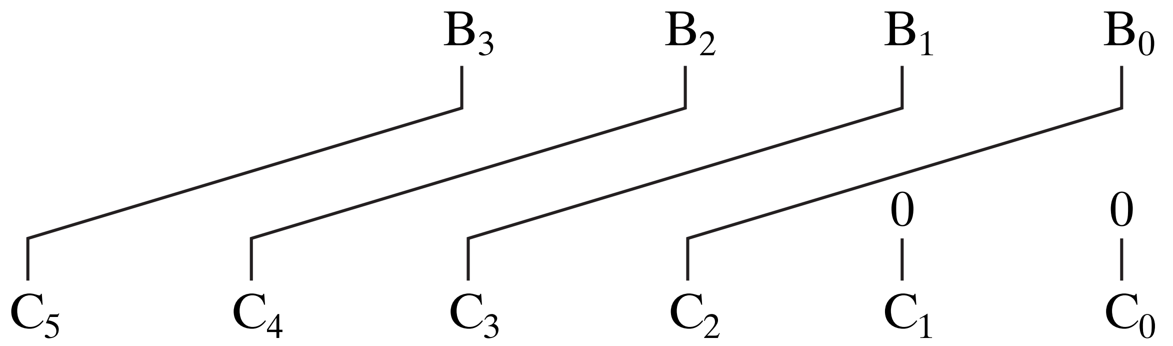
(b)

Incrementing and Decrementing

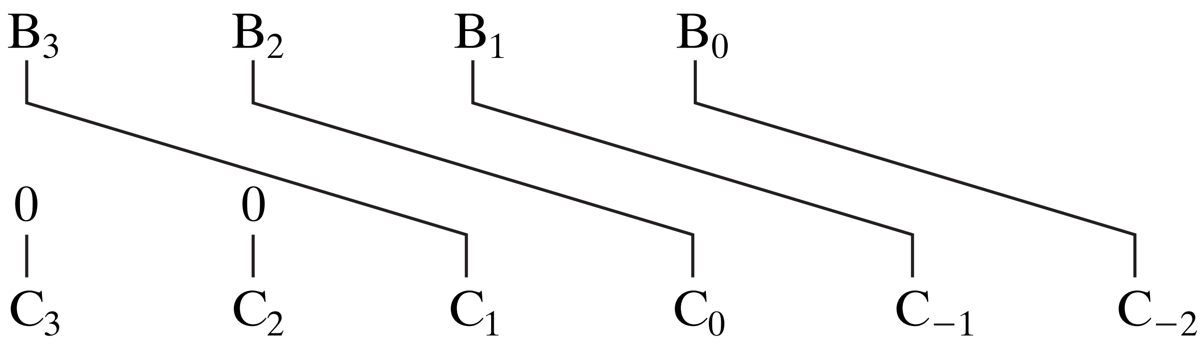
- Incrementing
  - Add a fixed value to an arithmetic variable
  - Fixed value is often 1, called counting up
    - A+1, B+4
  - Functional block is called incrementer
- Decrementing
  - Subtracting a fixed value from an arithmetic variable
  - Fixed value is often 1, called counting down
    - A-1, B-4
  - Functional block is called decrementer

Multiplication / Division by 2^n

- Shift left (multiplication) or right (division)

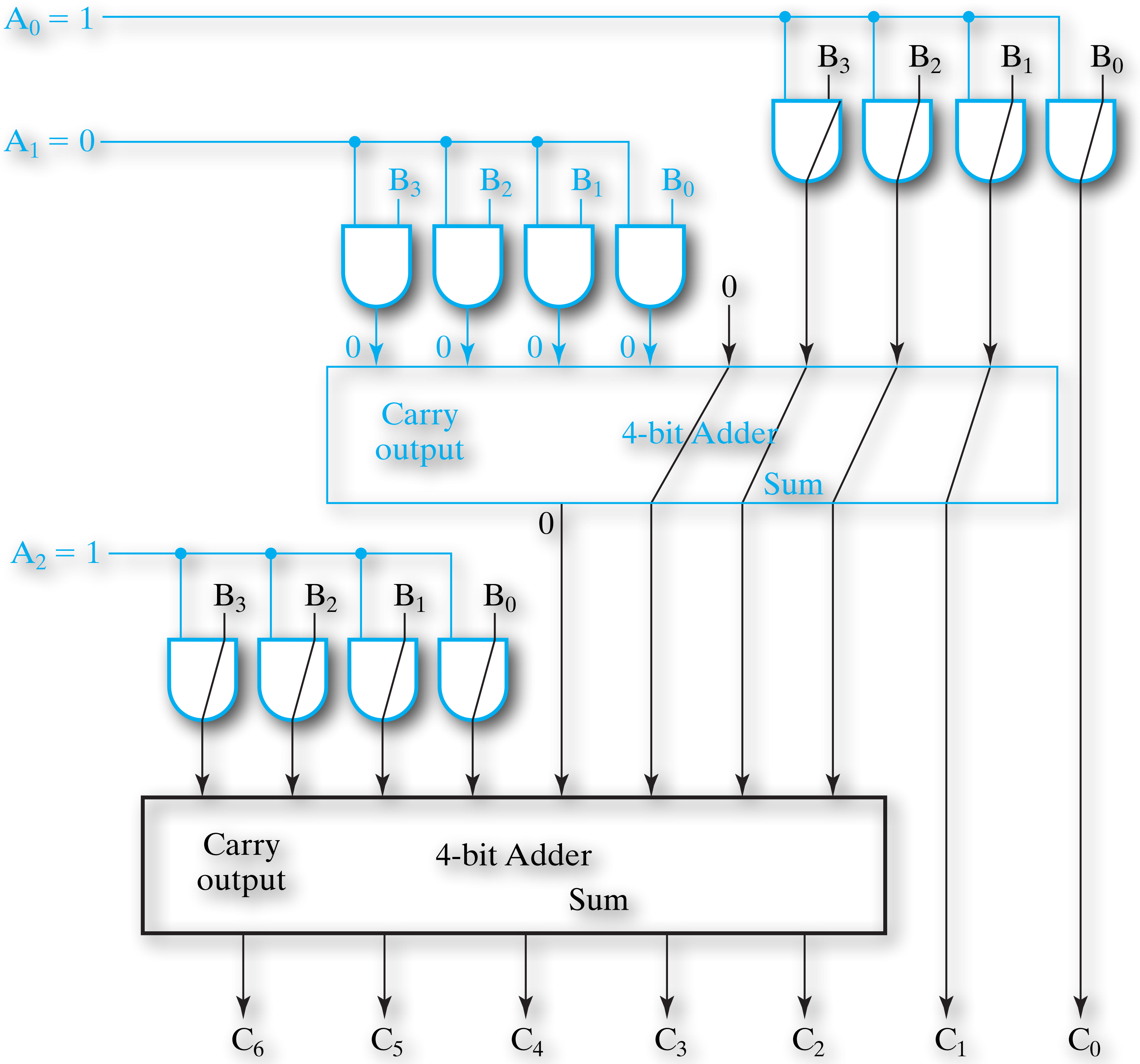


shift left by 2



shift right by 2

Multiplication by a Constant



Zero Fill

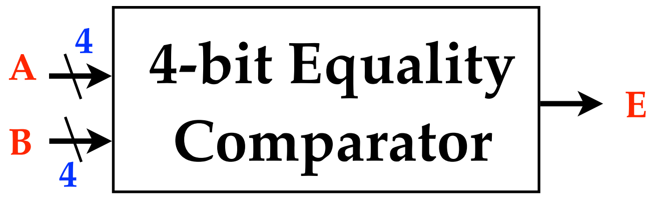
- Fill an *m*-bit operand with 0s to become an *n*-bit operand with *n* > *m*.
- Filling usually is applied to the MSB end of the operand, but can also be done on the LSB end.
- 11110101 filled to 16 bits
  - MSB end: 0000000011110101 {{8{0}}11110101}
  - LSB end: 1111010100000000 {11110101{8{0}}}

Extension

- Increase in the number of bits at the MSB end of an operand by using a complement representation
  - Copies the MSB of the operand into the new positions
  - 01110101 extended to 16 bits
    - **0000000001110101** $\{\{8\{a_7\}\}a_71110101\}$
  - 11110101 extended to 16 bits
    - **1111111111110101**

Magnitude Comparator

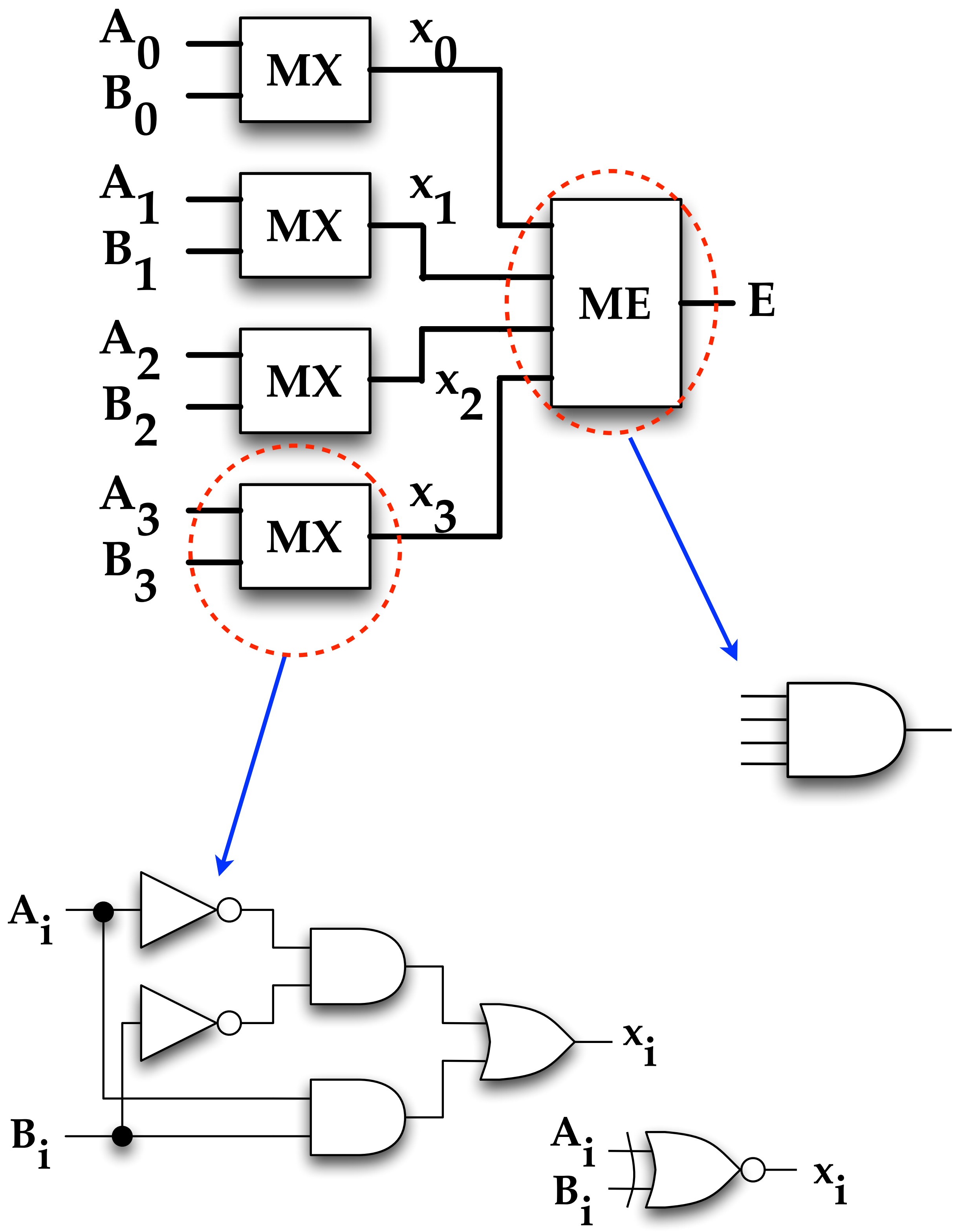
A 4-bit Equality Comparator



- Spec $\boxed{1}$
  - input A(3:0), B(3:0); output E (1/0 for equal/unequal)
- Formulation $\boxed{2}$
  - Bypass the truth table approach due to its size (8 inputs)
  - By algorithm to build a regular circuit
    - $A=A_3A_2A_1A_0$, $B=B_3B_2B_1B_0$
    - A==B, if $(A_3==B_3)$ AND $(A_2==B_2)$ AND $(A_1==B_1)$ AND $(A_0==B_0)$
      - bit equality $x_i=A_iB_i+A_i'B_i'$, $(A==B) = x_3x_2x_1x_0$

A 4-bit Equality Comparator

- Optimization 3
  - Regularity 4
  - Reuse



Magnitude Comparator

1

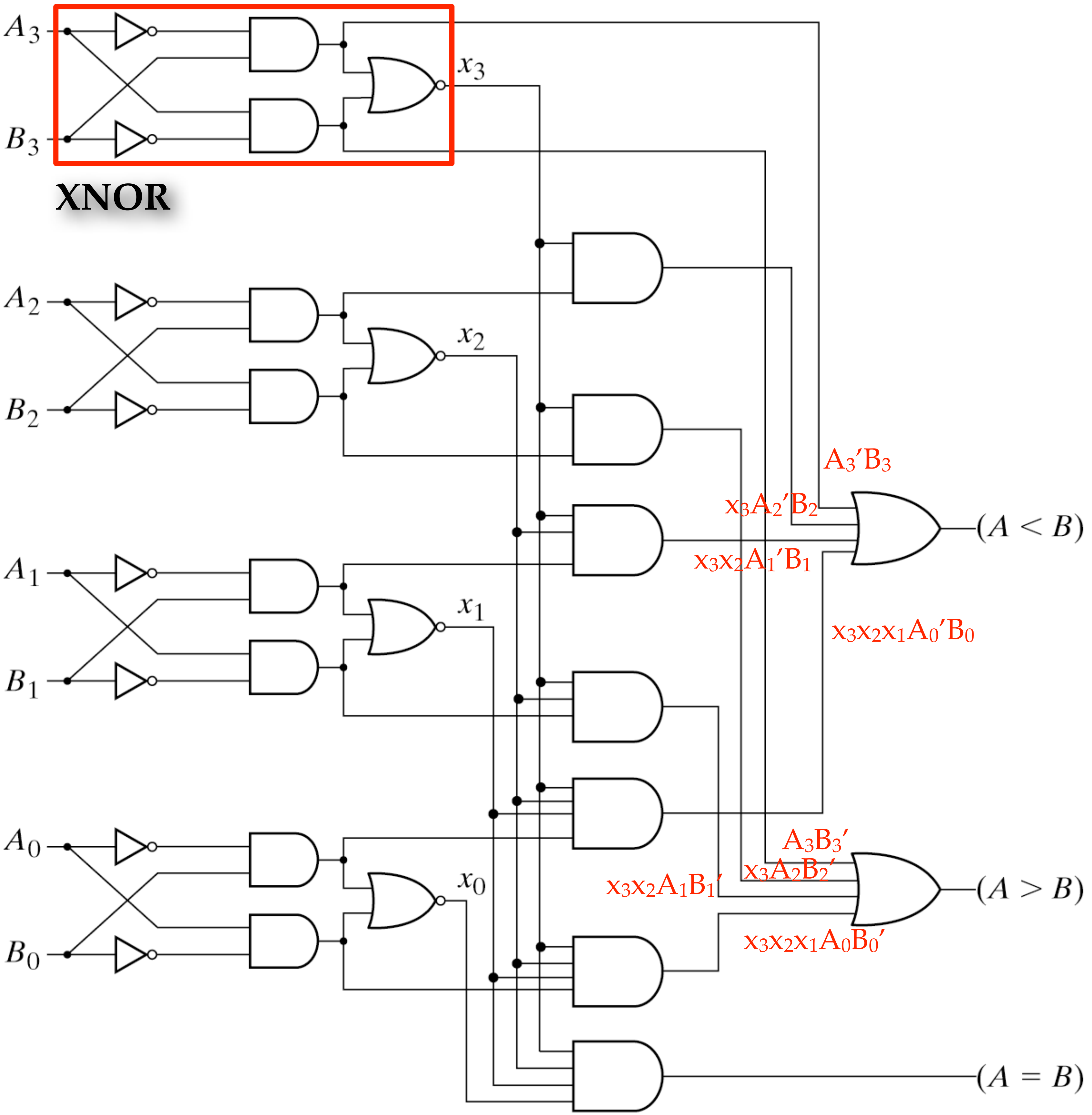
- Comparison of two numbers, three possible results (A>B, A=B, A<B)
- Design approaches (for *n*-bit numbers)
  - By truth table: 2^{2n} rows => not practicable 2 x
  - By algorithm to build a regular circuit

3

    - $A=A_3A_2A_1A_0$, $B=B_3B_2B_1B_0$
    - $A==B$, if $(A_3==B_3)$ AND $(A_2==B_2)$ AND $(A_1==B_1)$ AND $(A_0==B_0)$
      - equality $x_i=A_iB_i+A_i'B_i'$, $(A=B) = x_3x_2x_1x_0$
    - $(A>B) = A_3B_3'+x_3A_2B_2'+x_3x_2A_1B_1' +x_3x_2x_1A_0B_0'$
    - $(A<B) = A_3'B_3+x_3A_2'B_2+x_3x_2A_1'B_1 +x_3x_2x_1A_0'B_0$

Magnitude Comparator

4



XNOR

$A_3'B_3$

$x_3A_2'B_2$

$x_3x_2A_1'B_1$

$x_3x_2x_1A_0'B_0$

A_3B_3'

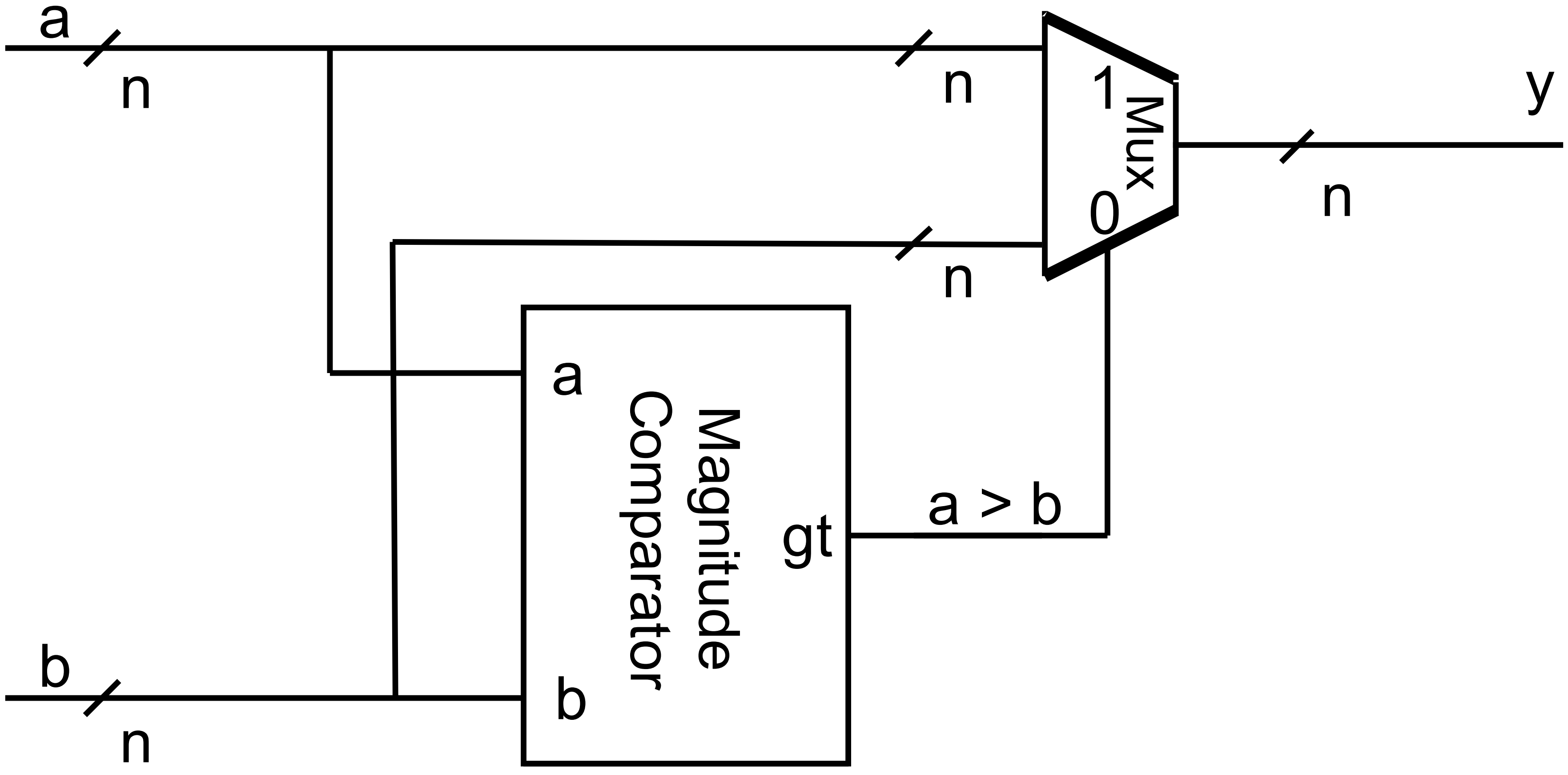
$x_3A_2B_2'$

$x_3x_2A_1B_1'$

$x_3x_2x_1A_0B_0'$

Maximun Unit

$$y = \max\{a, b\}$$



Decoders

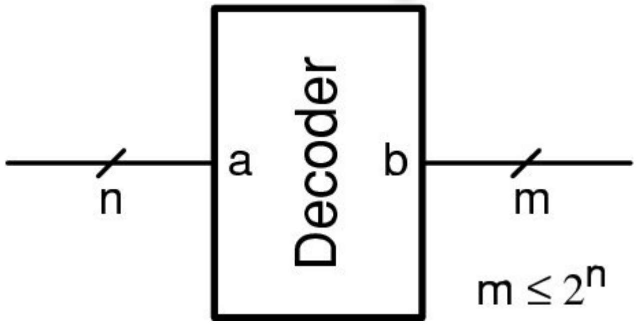
One-hot Representation

- Represent a set of N elements with N bits
- Exactly one bit is set

Binary	One-hot
000	00000001
001	00000010
010	00000100
011	00001000
100	00010000
101	00100000
110	01000000
111	10000000

Decoder

- A decoder is a combinational circuit that converts binary information from *n* input lines to *m* (maximum of 2^n) ***unique*** output lines
  - –*n*-to-*m*-line decoder



- A binary one-hot decoder converts a symbol from binary code to a one-hot code
  - Output variables are ***mutually exclusive*** because only one output can be equal to 1 at any time (the very 1-minterm)
  - Example
    - binary input ***a*** to one-hot output ***b***

$$b[i] = 1 \text{ if } a = i \qquad \text{or} \qquad b = 1 << a$$

1-to-2-Line Decoder

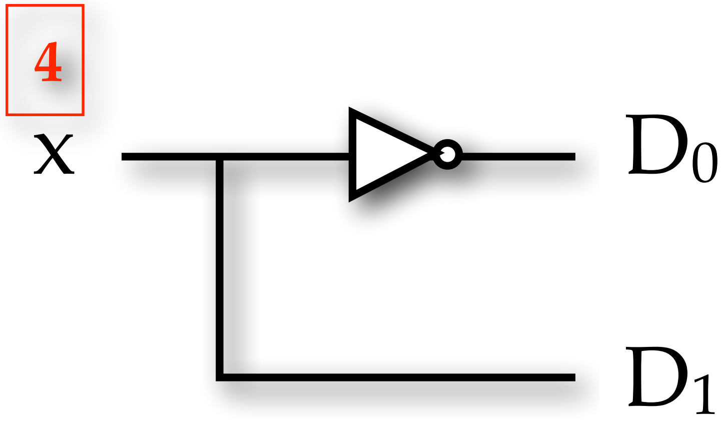
1 2

x	D_1	D_0
0	0	1
1	1	0

3

$D_0 = x'$

$D_1 = x$



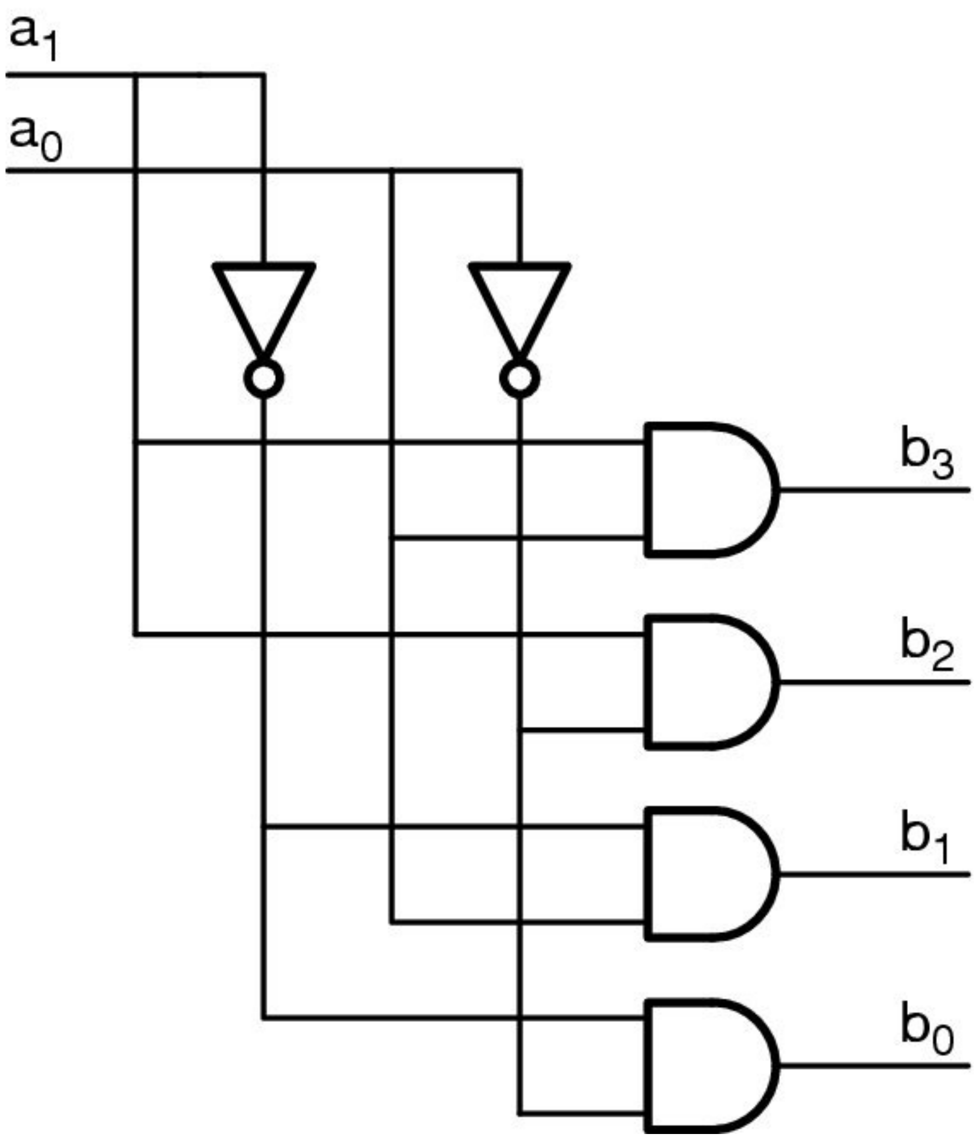
2-to-4-Line Decoder

1 2

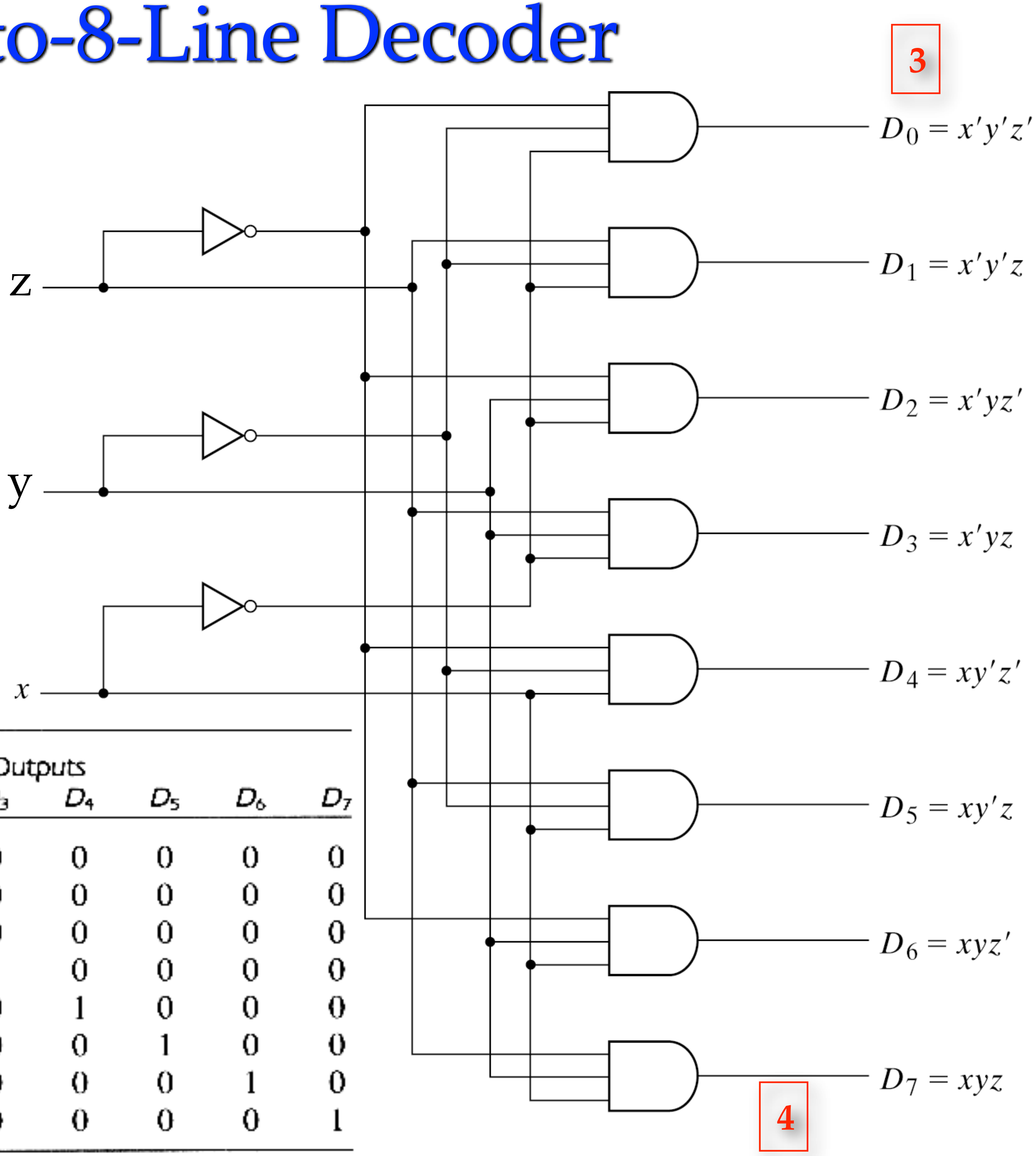
a_1	a_0	b_3	b_2	b_1	b_0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

3

$$b_3 = a_1 a_0$$
$$b_2 = a_1 a_0'$$
$$b_1 = a_1' a_0$$
$$b_0 = a_1' a_0'$$



3-to-8-Line Decoder



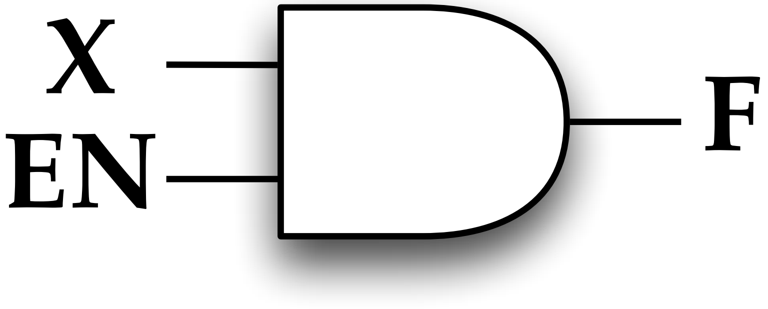
1

2

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

Enabling

- Enabling permits an input signal to pass through to an output.



$$F = \mathrm{EN}{\cdot}X$$

EN	X	F
0	0	0
0	1	0
1	0	0
1	1	1

Decoder with Enable Input (1/3)

- Line decoder with *enable* control (E)
- Also called demultiplexer (DMUX, DEMUX)

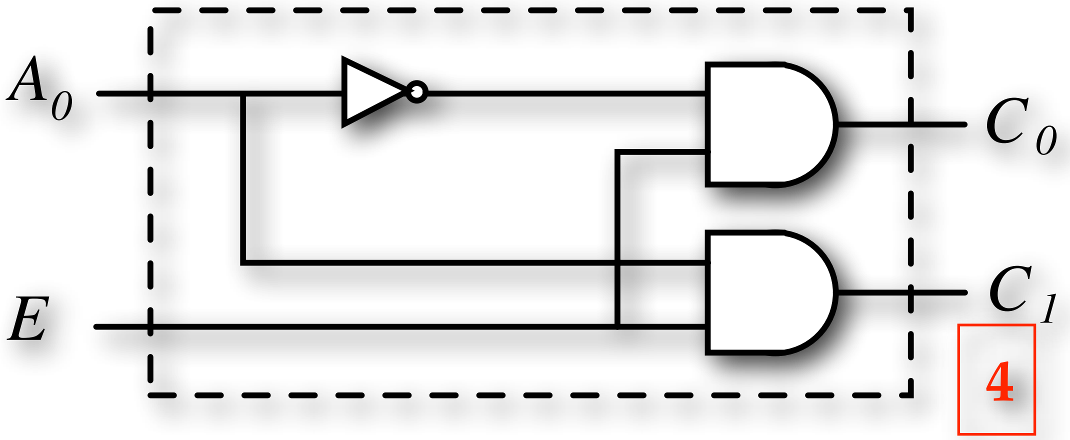
1 2

E	A_0	C_1	C_0
1	0	0	1
1	1	1	0
0	x	0	0

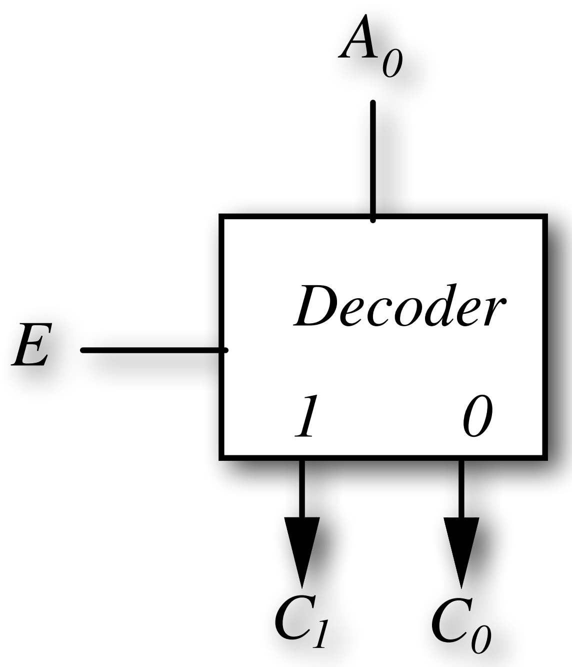
3

$C_0 = EA_0'$

$C_1 = EA_0$



4



graphic symbol/ block diagram

Decoder with Enable Input (2/3)

- Constructed with NAND gates
  - decoder minterms in their complemented form (more economical)

1 2

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

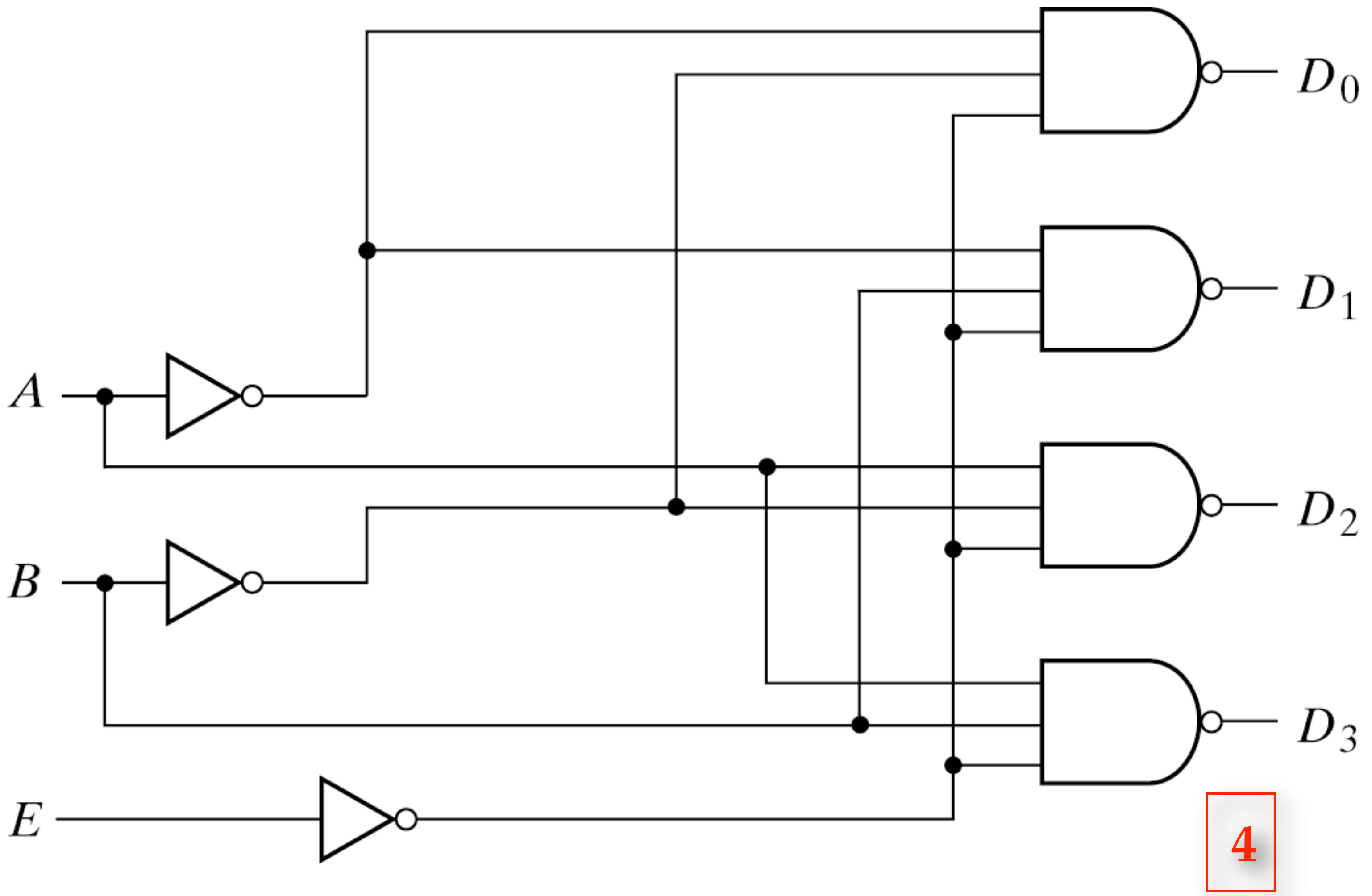
3

$D_0=(E'A'B')'$

$D_1=(E'A'B)'$

$D_2=(E'AB')'$

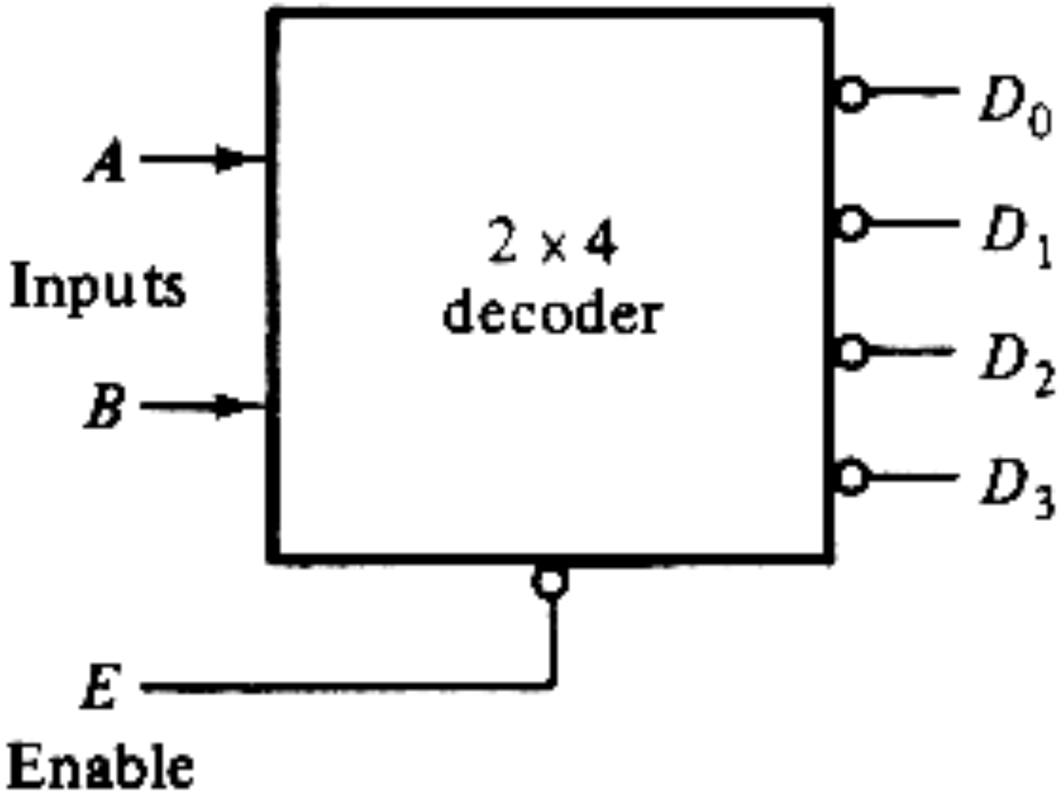
$D_3=(E'AB)'$

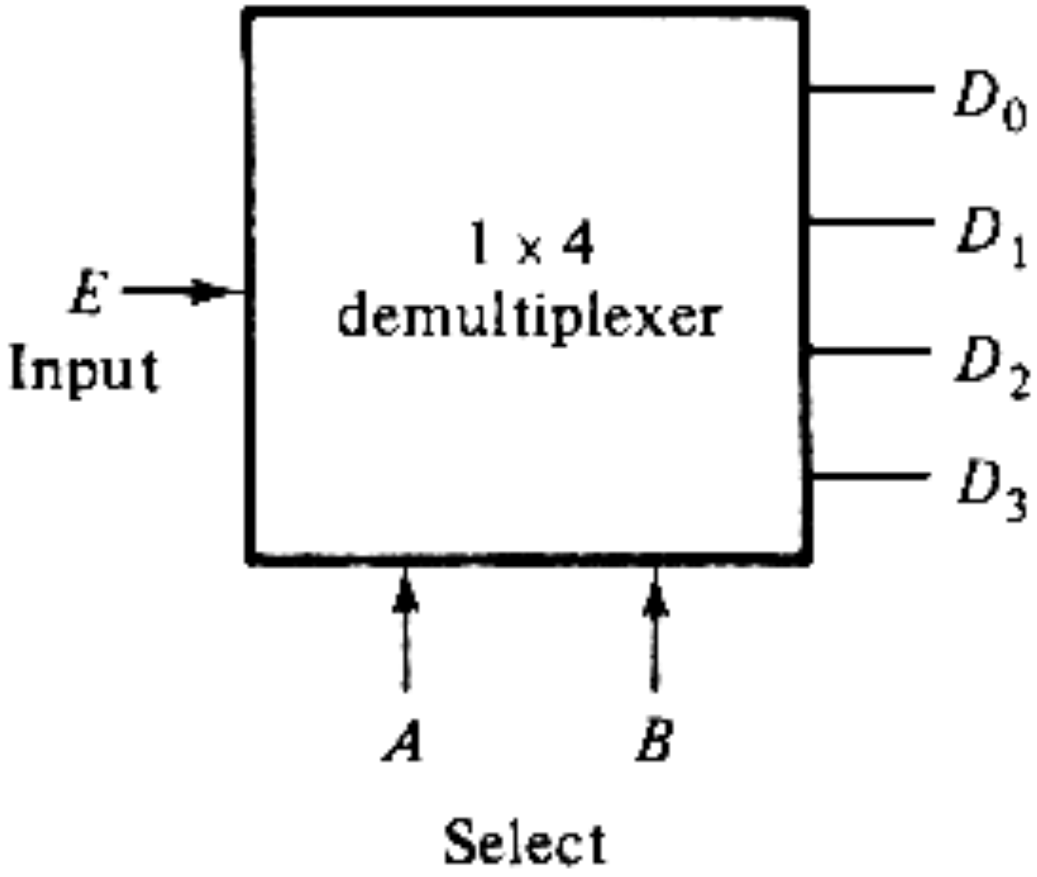


4

Decoder with Enable Input (3/3)

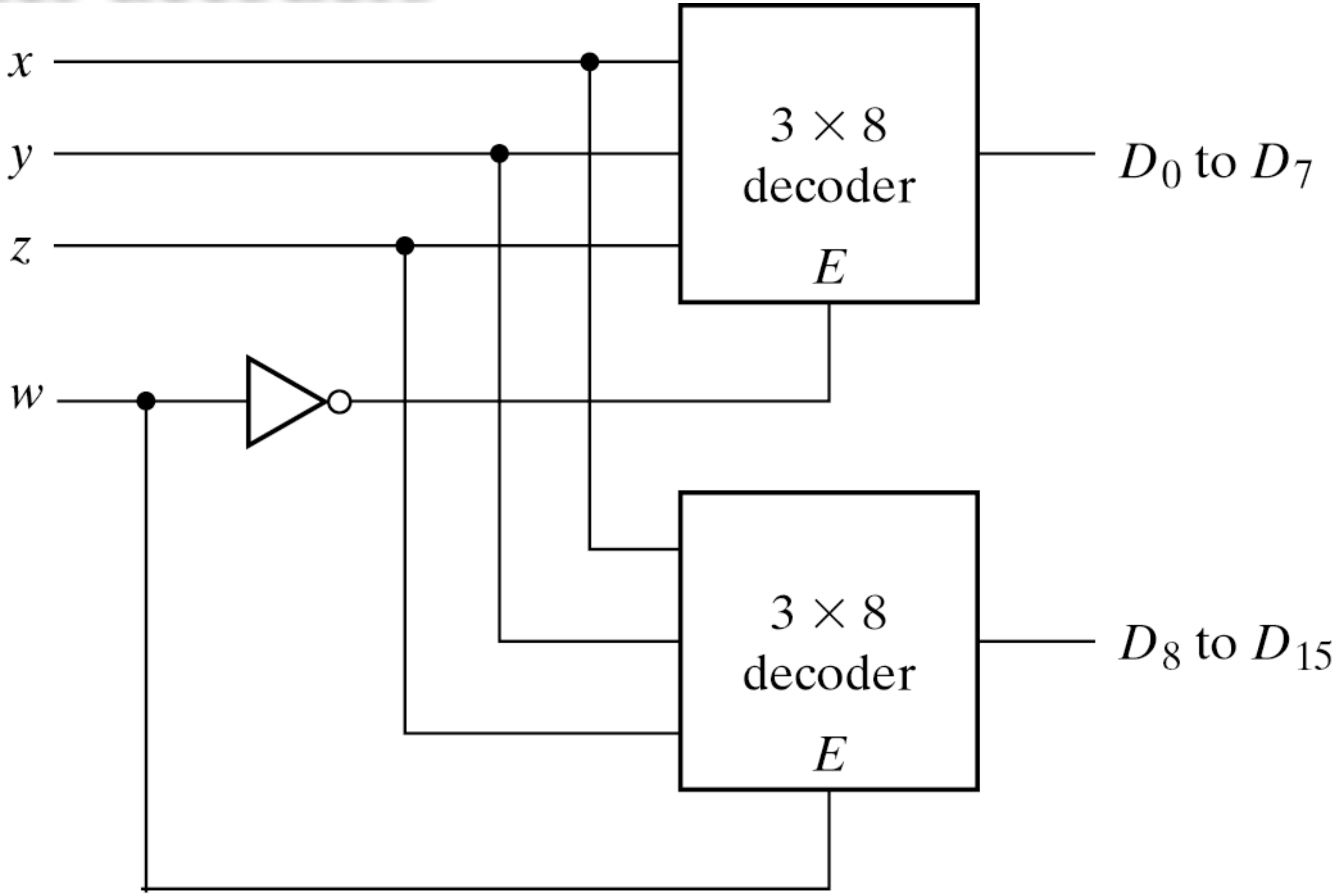
- decoder with enable vs. demultiplexer





Decoder Expansion

- Larger decoders can be implemented with smaller decoders



A 4-to-16-line decoder from two 3-to-8-line decoders

Combinational Logic Implementation with Decoders

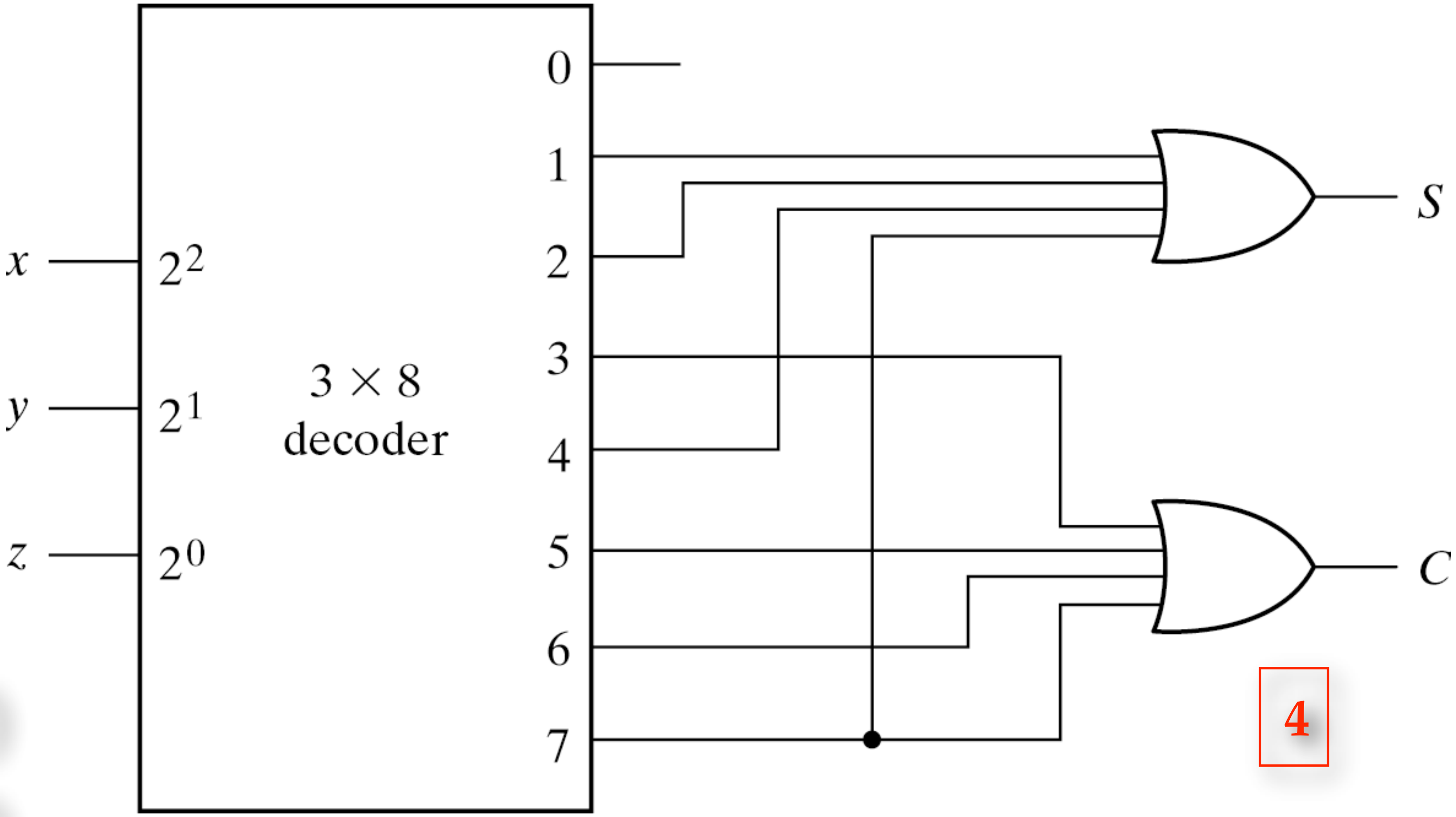
- Any combinational circuit with *n* inputs and *m* outputs can be implemented with an *n*-to-2^n decoder in conjunction with *m* external OR gates

1 2

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

3

$$S(x, y, z) = \sum(1, 2, 4, 7)$$
$$C(x, y, z) = \sum(3, 5, 6, 7)$$

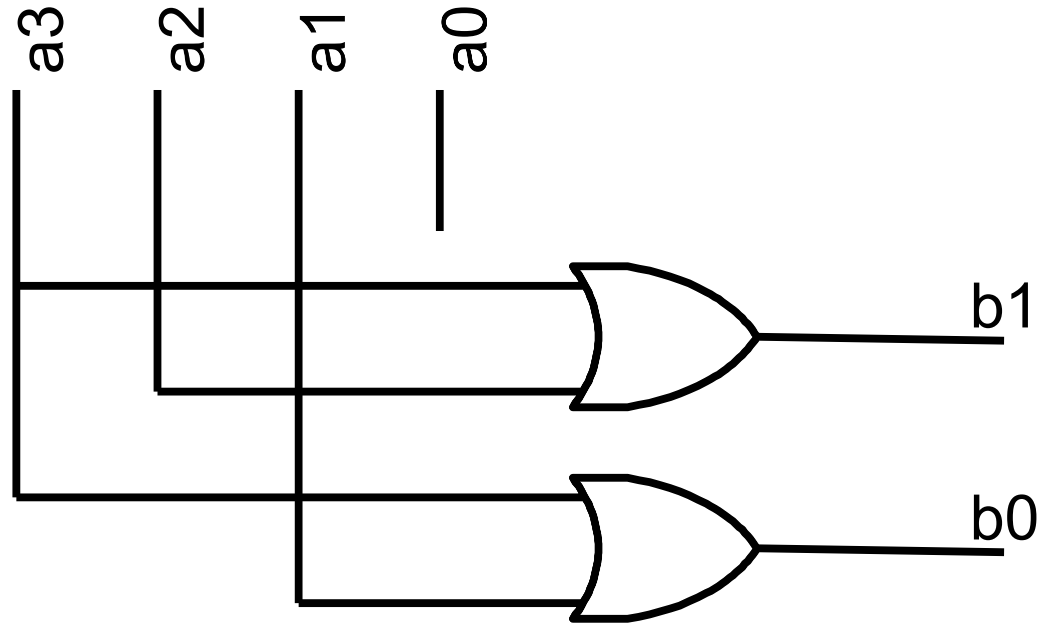


Encoders

Encoder

- An encoder is an inverse of a decoder.
- Encoder is a logic module that converts a *one-hot* input signal to a binary-encoded output signal
- Other input patterns are ***forbidden*** in the truth table.
- Example: a 4->2 encoder

a3	a2	a1	a0	b1	b0
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1

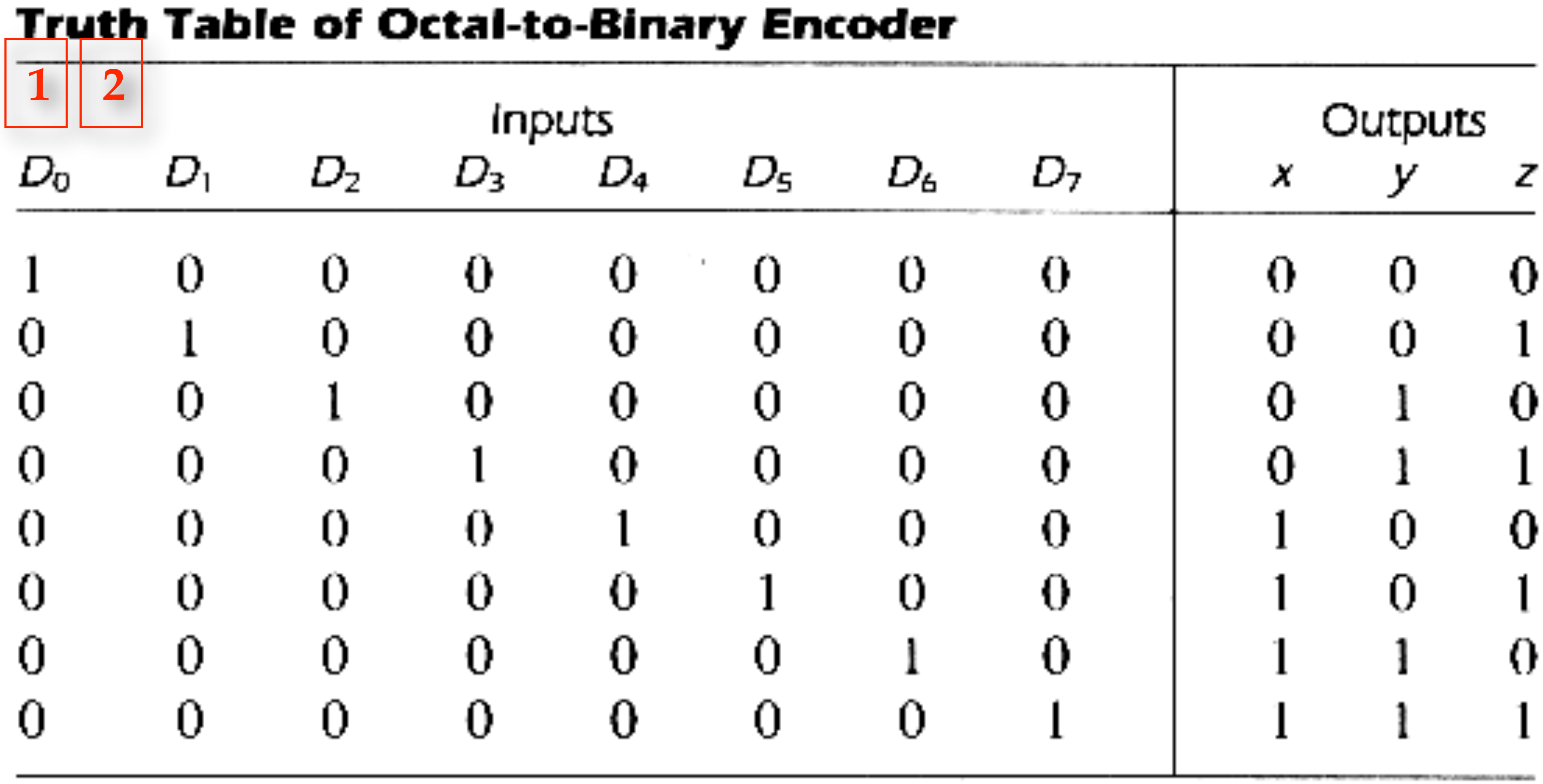


$$b_0 = a_3 + a_1$$

$$b_1 = a_3 + a_2$$

Encoder (1/2)

- A combinational logic that performs the inverse operation of a decoder
  - Only one input has value 1 at any given time
  - Can be implemented with OR gates



Truth Table of Octal-to-Binary Encoder

1 2

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

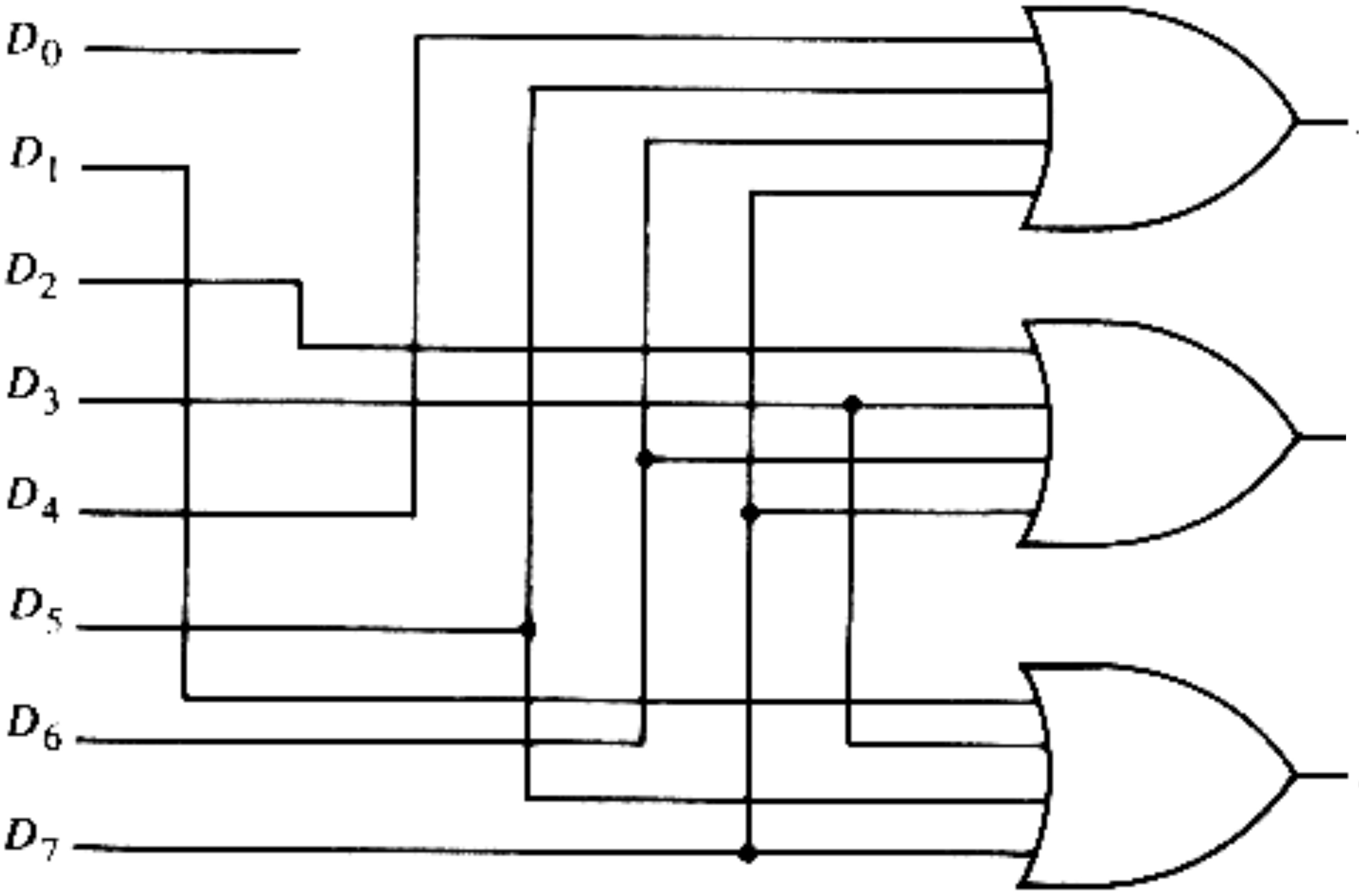
3

$x=D_4+D_5+D_6+D_7$

$y=D_2+D_3+D_6+D_7$

$z=D_1+D_3+D_5+D_7$

Encoder (2/2)



$x=D_4+D_5+D_6+D_7$

$y=D_2+D_3+D_6+D_7$

$z=D_1+D_3+D_5+D_7$

4

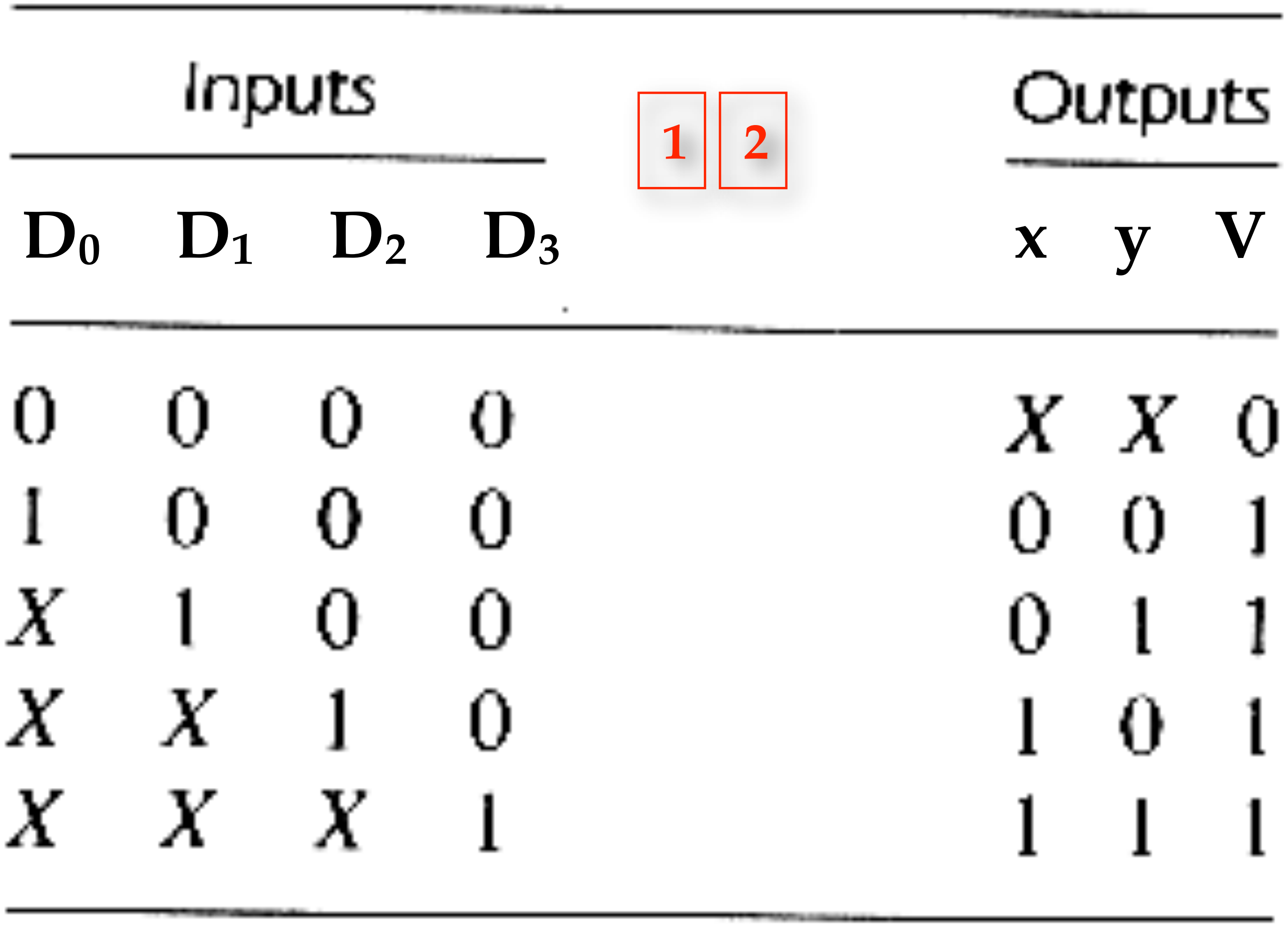
However, when both D3 and D6 goes 1, the output will be 111 (ambiguity)!!!

illegal inputs

Use priority encoder!

Priority Encoder (1/2)

- Ensure only one of the input is encoded
- D_3 has the highest priority, while D_0 has the lowest priority.
- X is the don't care conditions, V is the valid output indicator.



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

1 2

3

$V=D_0+D_1+D_2+D_3$

Priority Encoder (2/2)

D_2

	00	01	11	10
00	X	1	1	1
01		1	1	1
11		1	1	1
10		1	1	1

D_1

D_0

D_3

3

$x = D_2 + D_3$

$y = D_3 + D_1 D_2'$

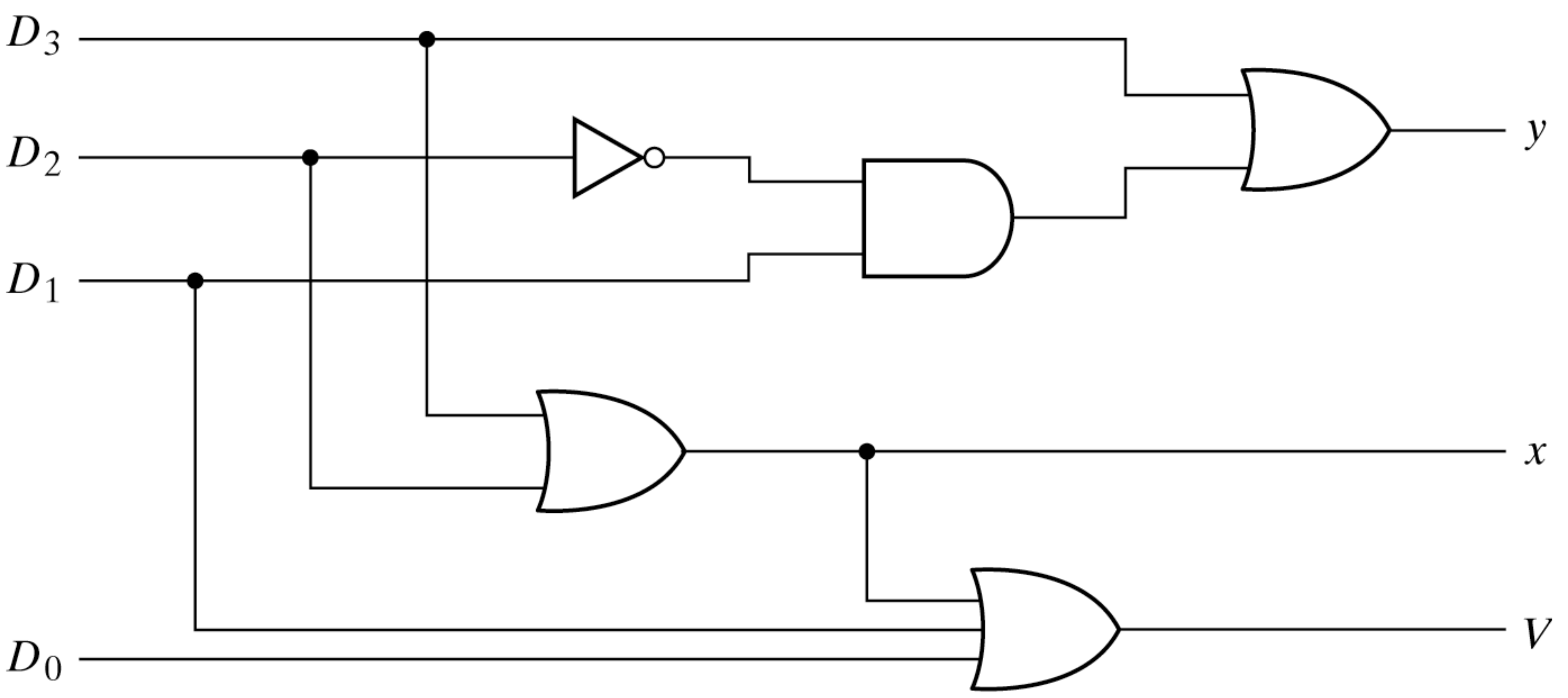
D_2

	00	01	11	10
00	X	1	1	
01	1	1	1	
11	1	1	1	
10		1	1	

D_1

D_0

D_3

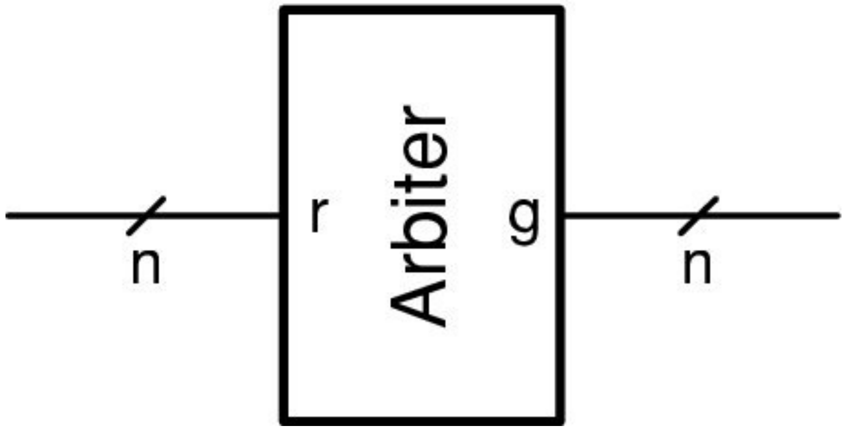


4

Arbiters and Priority Encoders

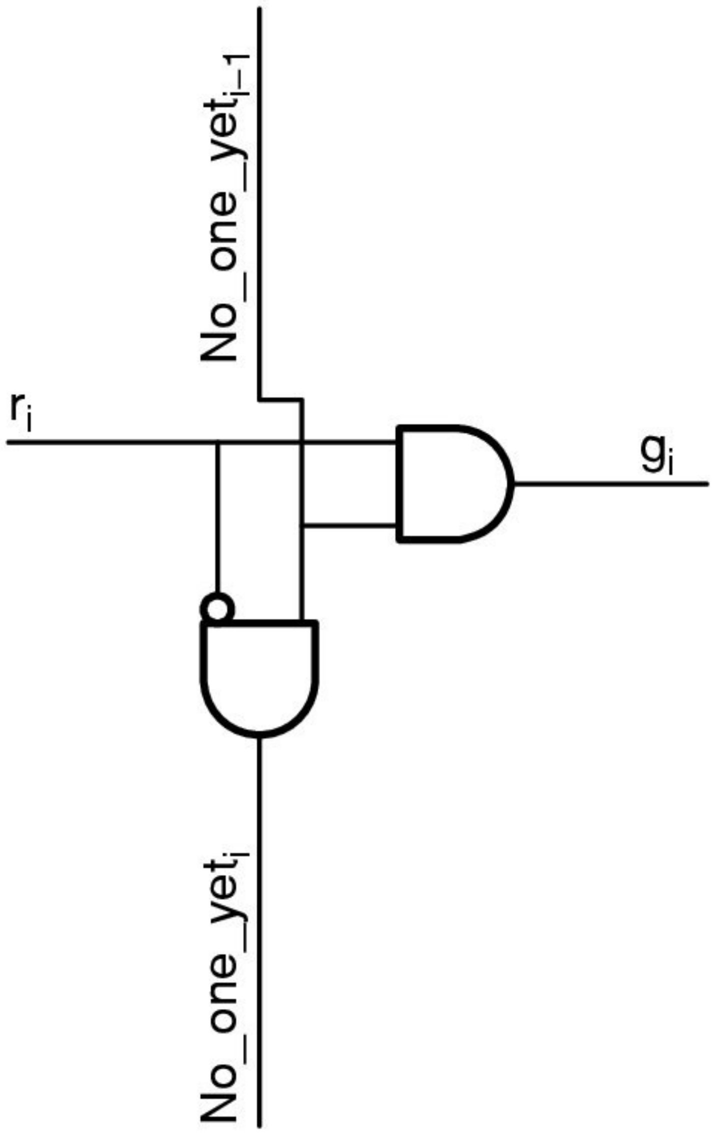
Arbiters

- Arbiter handles requests from multiple devices to use a single resource
  - Also called ***find-first-one*** (FF1) unit
  - Accepts an arbitrary input signal (r), and outputs a one-hot signal (g) to indicate the least significant 1 (or the most significant 1) of the input
  - Example: input: 01011100
    - output: 00000100 (least significant 1)
    - output: 01000000 (most significant 1)

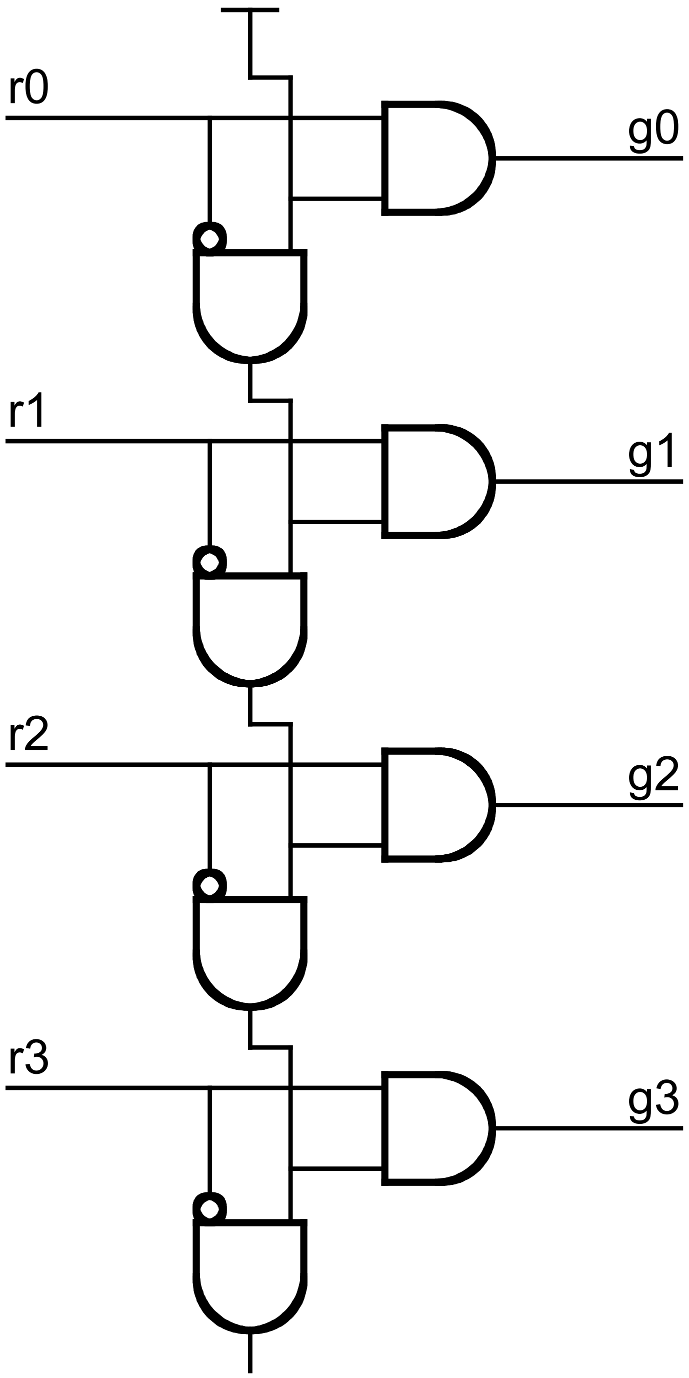


Finds the first "1" bit in r
g[i] = 1 if r[i] = 1 and r[j] = 0 for j < i
(for the least significant 1)

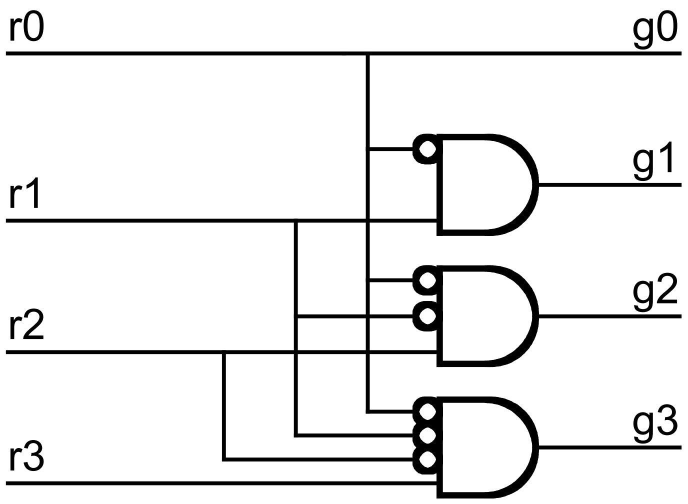
Implementation of Arbiters



1 bit cell of arbiter



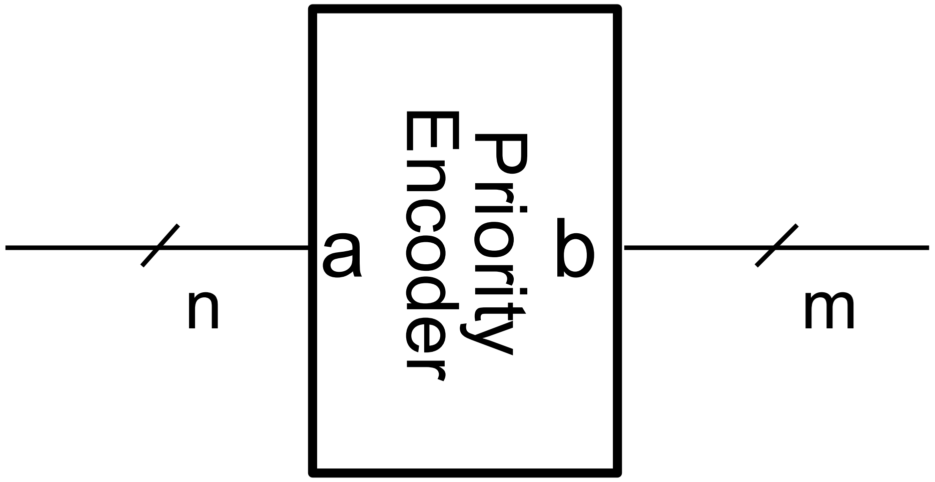
Using bit cell



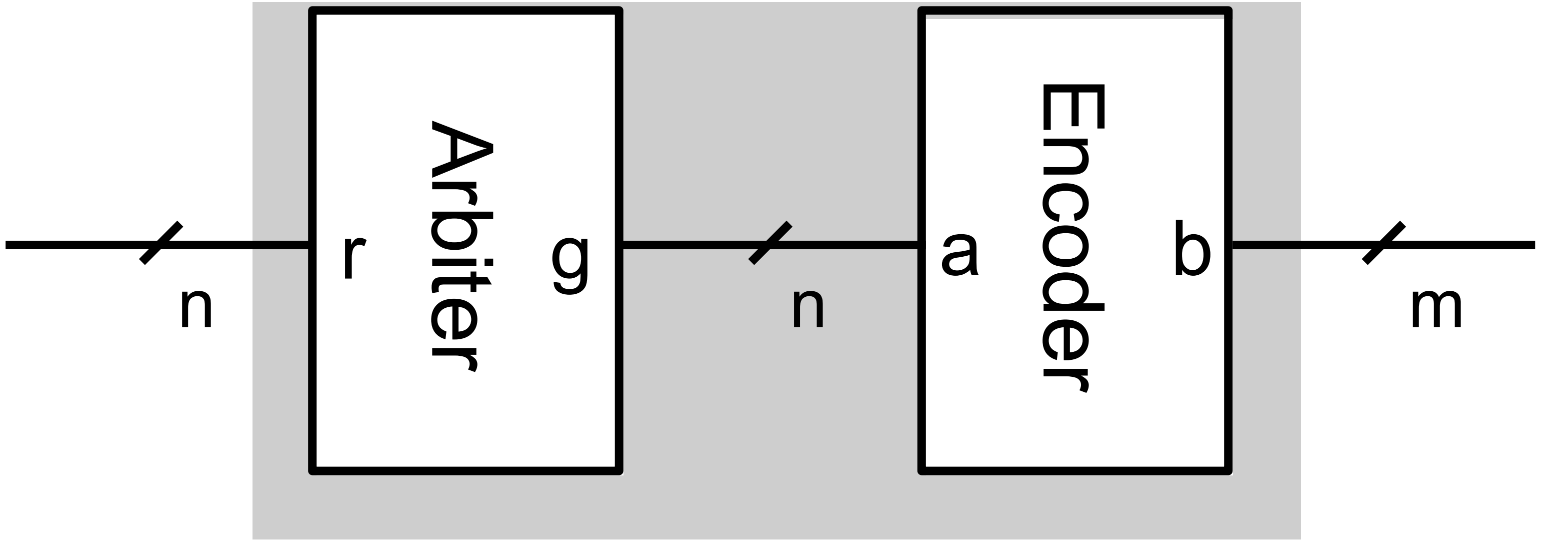
Using look ahead

Priority Encoder

- n-bit one-hot input signal a
- m-bit output signal b
  - b indicates the position of the first 1 bit in a



$$m = \lceil \log_2 n \rceil$$

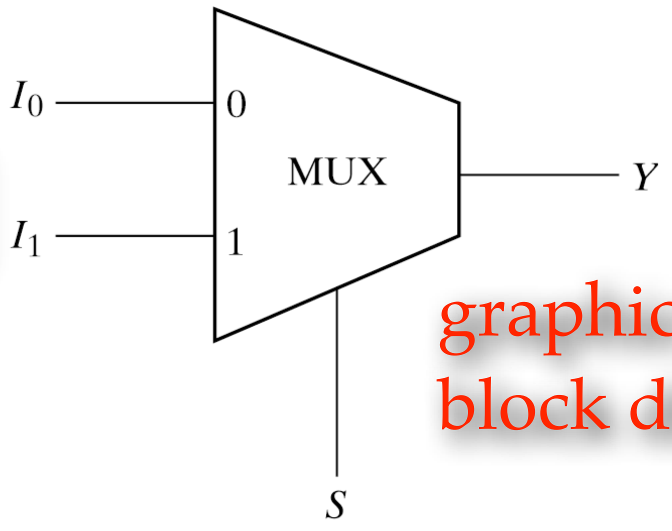


Multiplexers

Multiplexers / Selectors

- A Multiplexer selects (usually by *n* select lines) binary information from one of many (usually 2^n) input lines and directs it to a single output line.

1



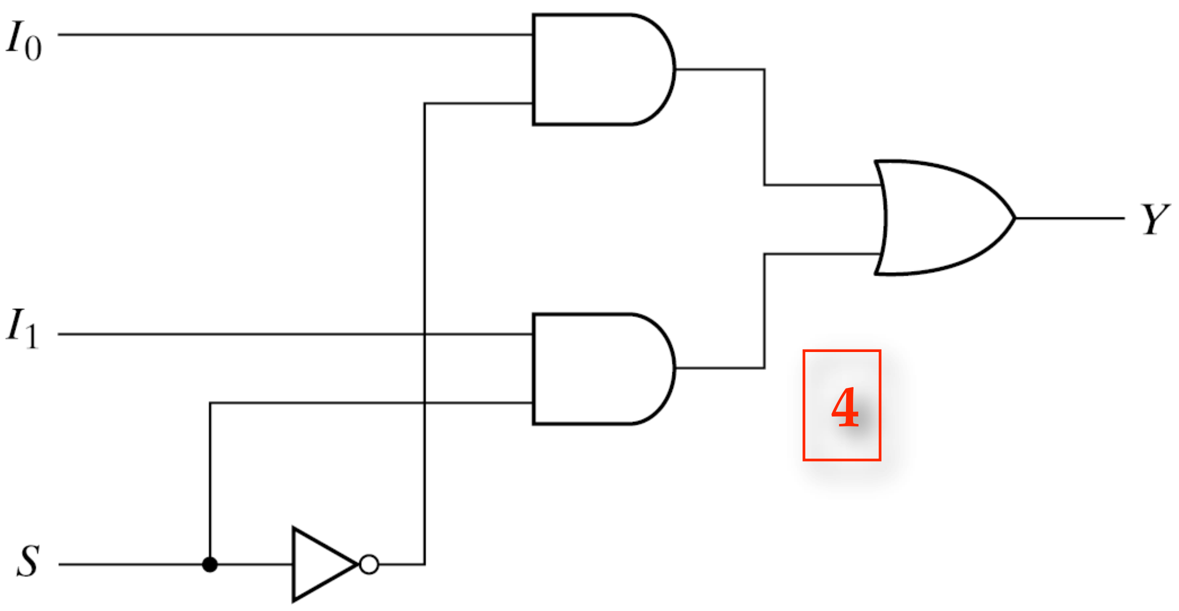
graphic symbol/ block diagram

2:1 multiplexer

2

S	Y
0	I_0
1	I_1

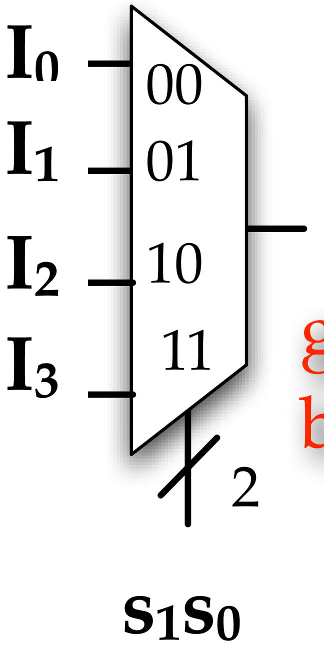
3 $Y=S'I_0+SI_1$



4

4:1 MUX

1



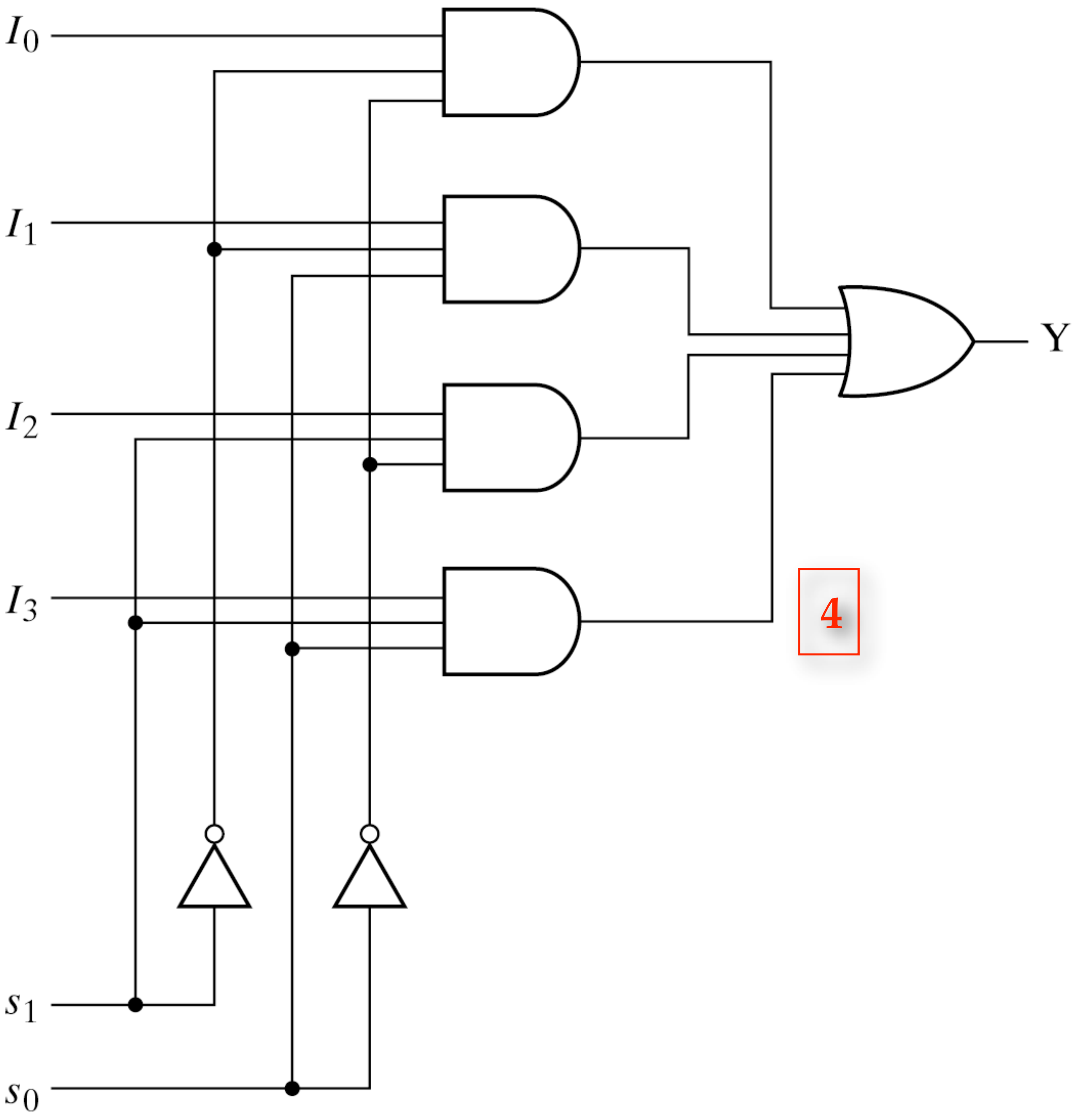
graphic symbol/ block diagram

2

s_1	s_0	Y
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

$$Y=s_0's_1'I_0+s_0s_1'I_1+s_0's_1I_2+s_0s_1I_3$$

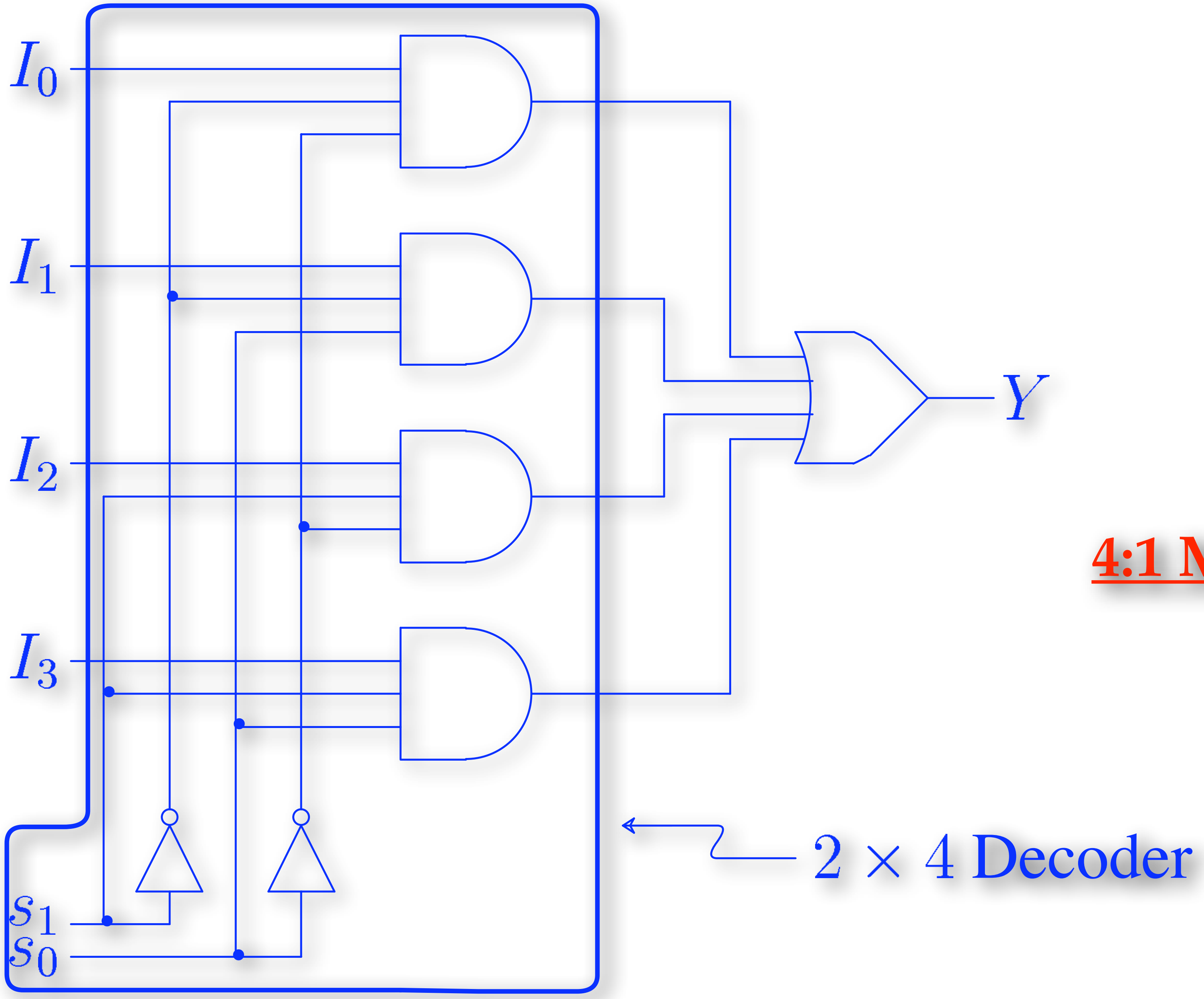
3



4

MUX as a Decoder

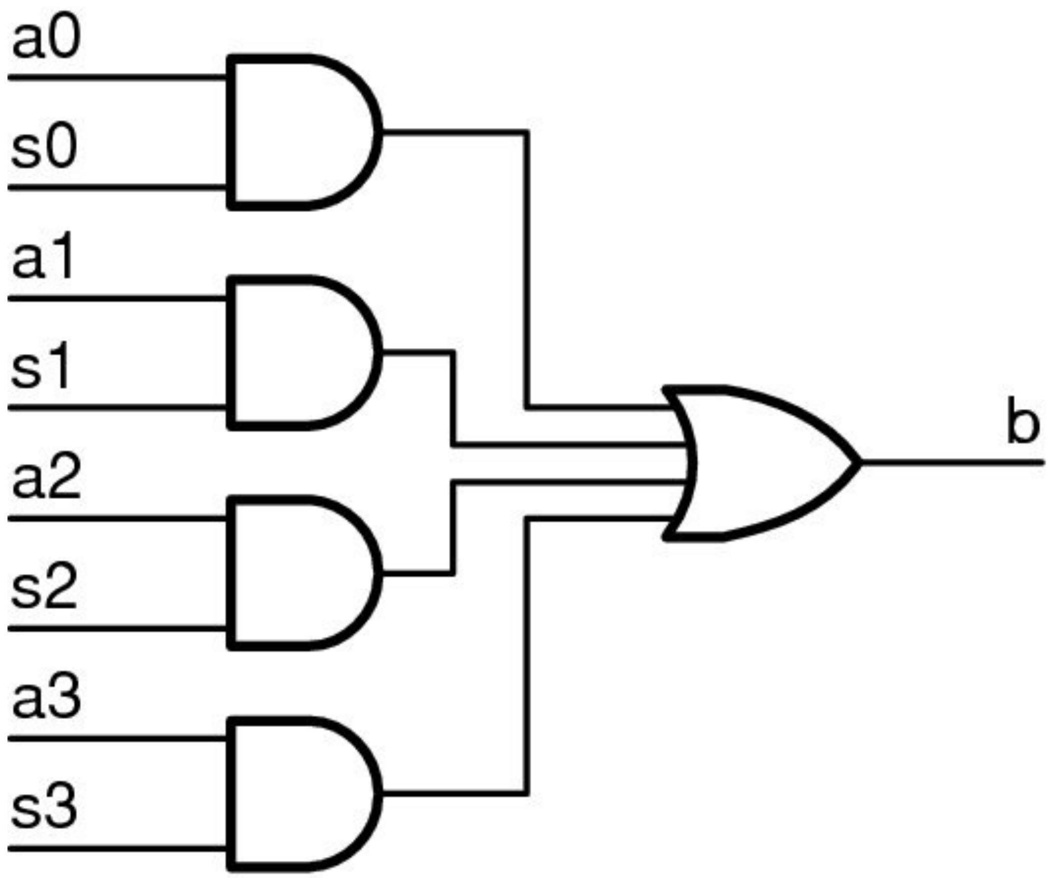
- MUX = decoder + OR gate + enable (optional)



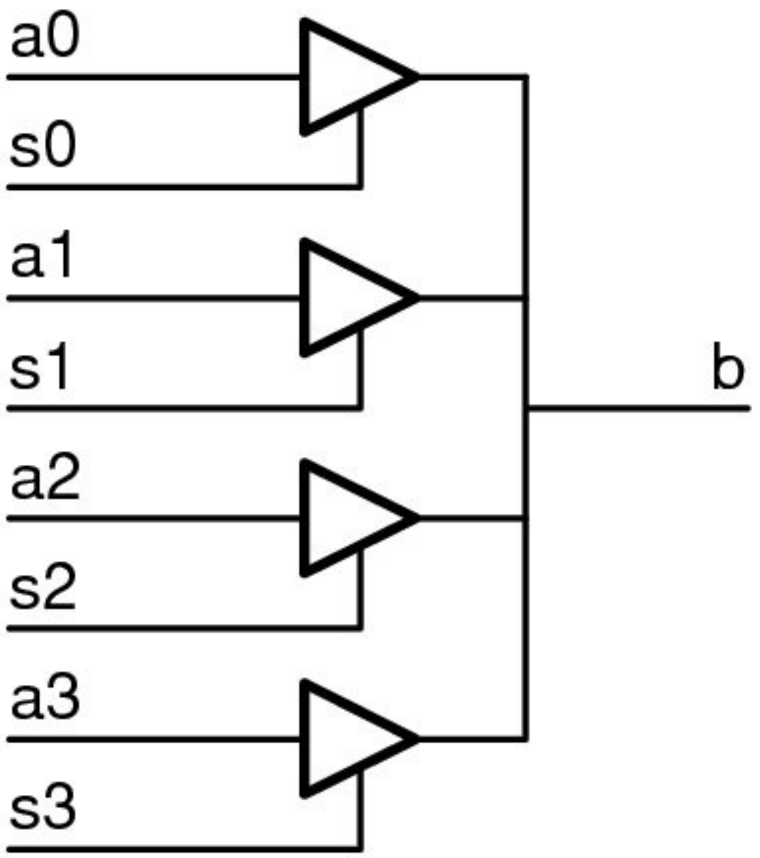
4:1 MUX

Multiplexer Implementation

- One-bit 4:1 multiplexer



Using AND-OR circuit

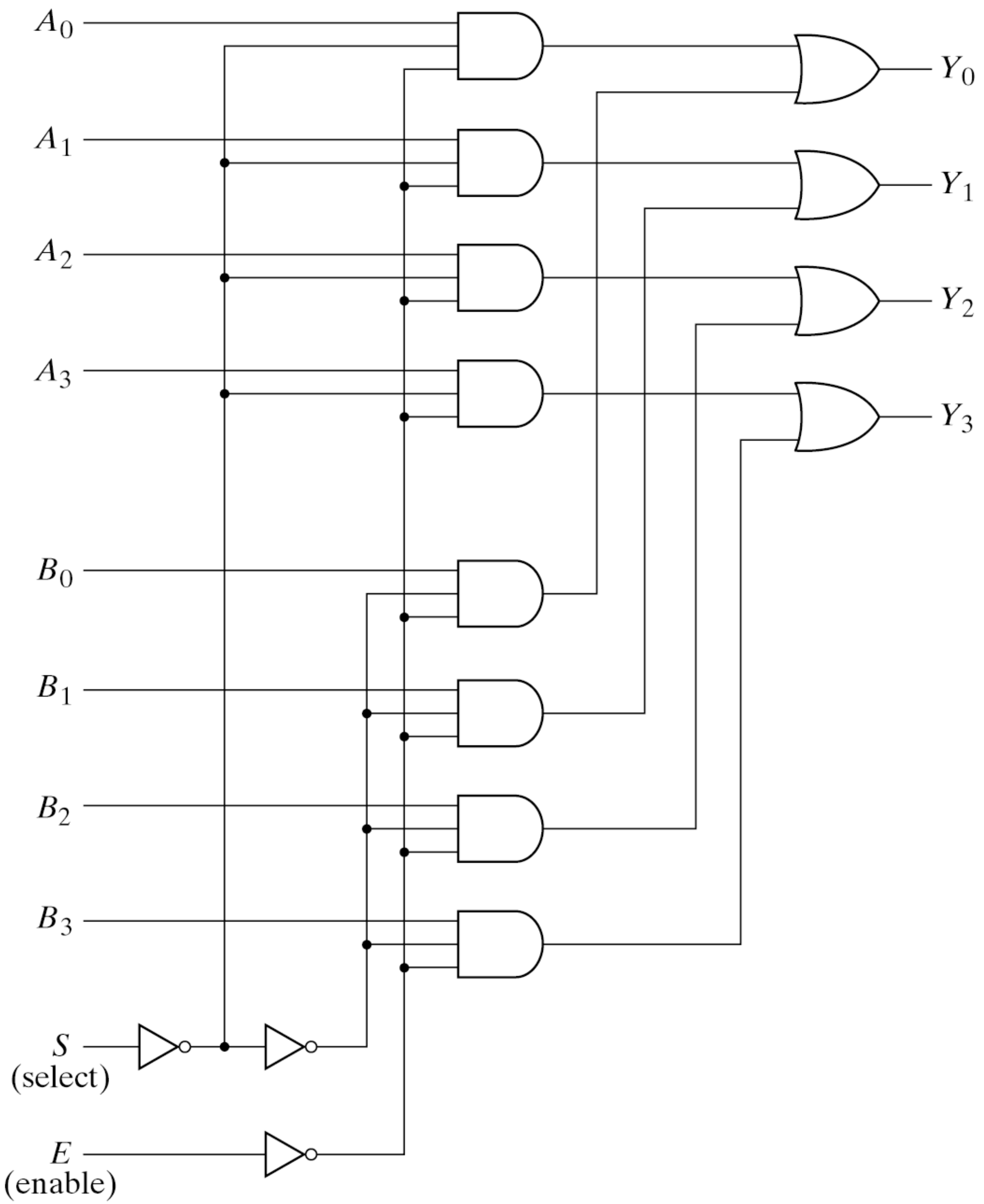


Using Tri-state buffer

Quadruple 2:1 MUX (4-bit 2:1 MUX)

Function table

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

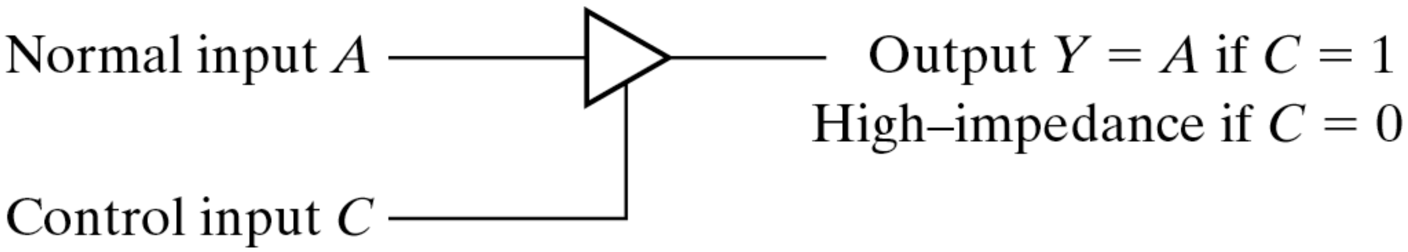


four 2:1 MUX with enable

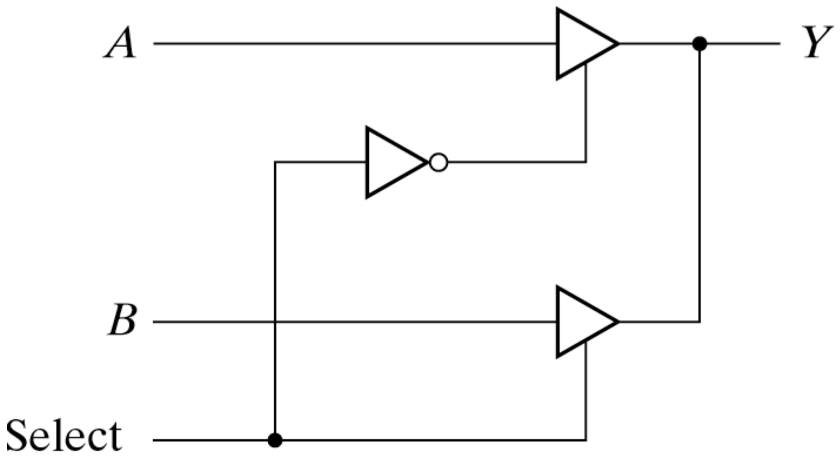
Bus

- Bus is a common communication channel which is routed around modules on a microchip or PCB.
- To construct a bus, we use a component, ***tristate driver (buffer)***, which has three possible output ***states***: 0, 1, Z (high impedance).
- Functionally, a bus is equivalent to a selector. It has many inputs but allow only one data on the bus at a time.

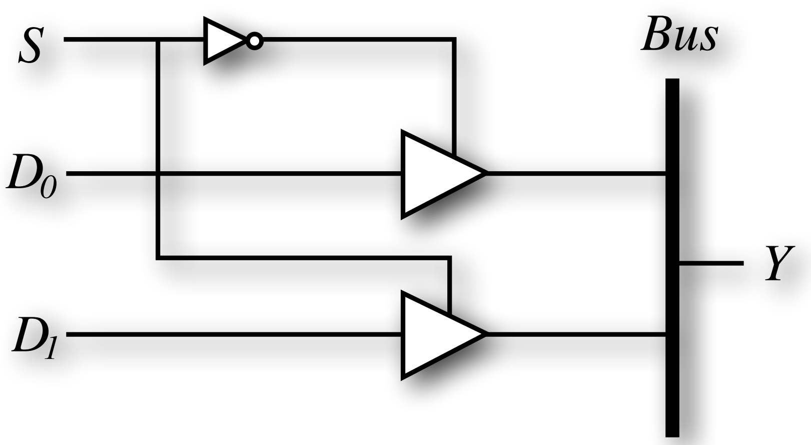
C	Y
0	Z
1	A

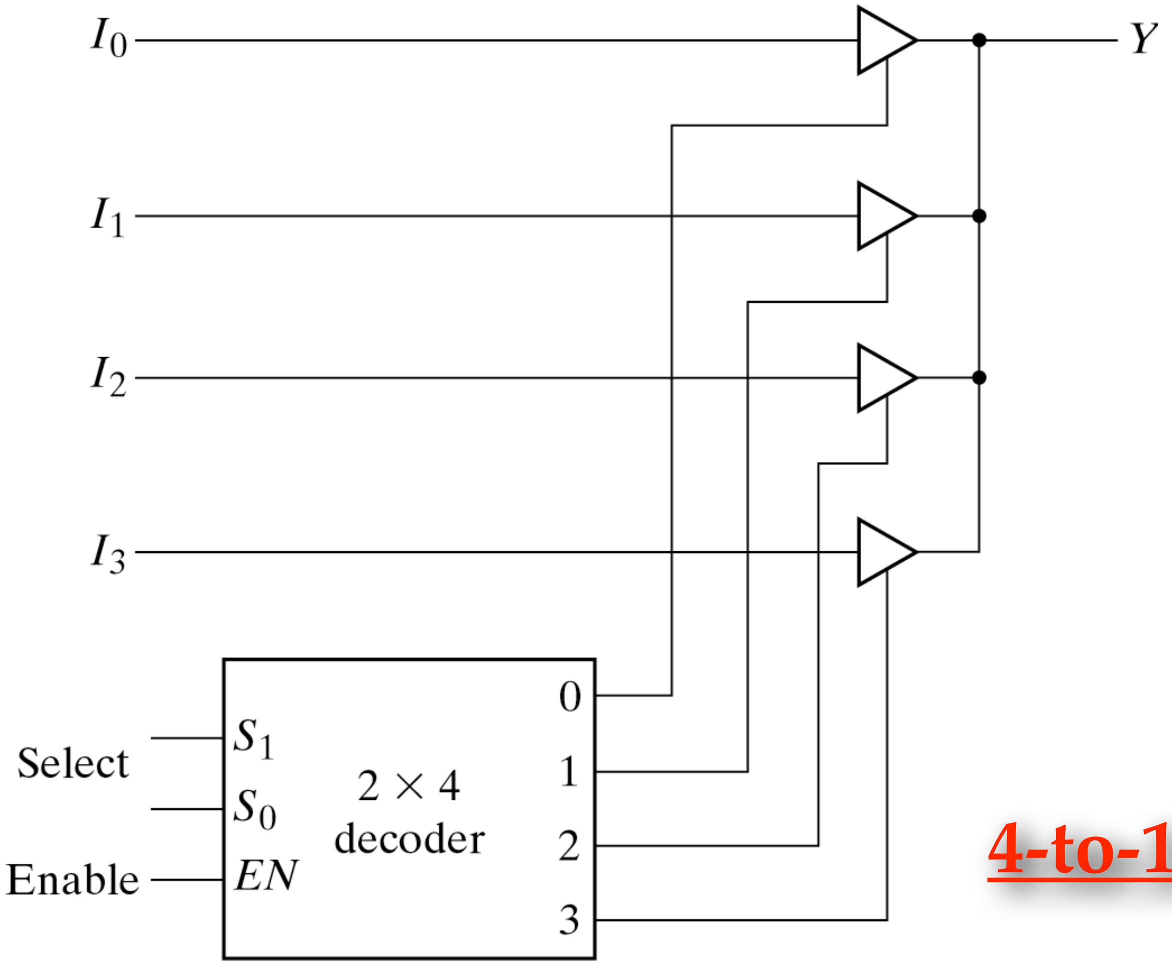


MUX with Three-State Gates

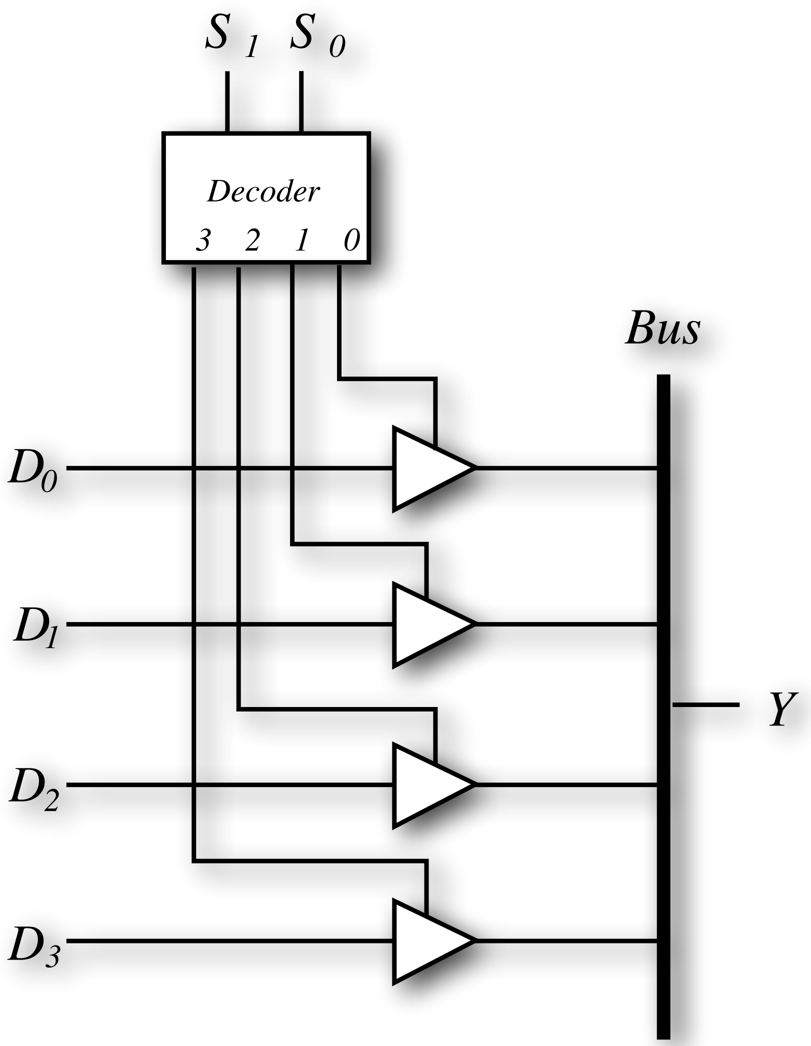


2-to-1 MUX





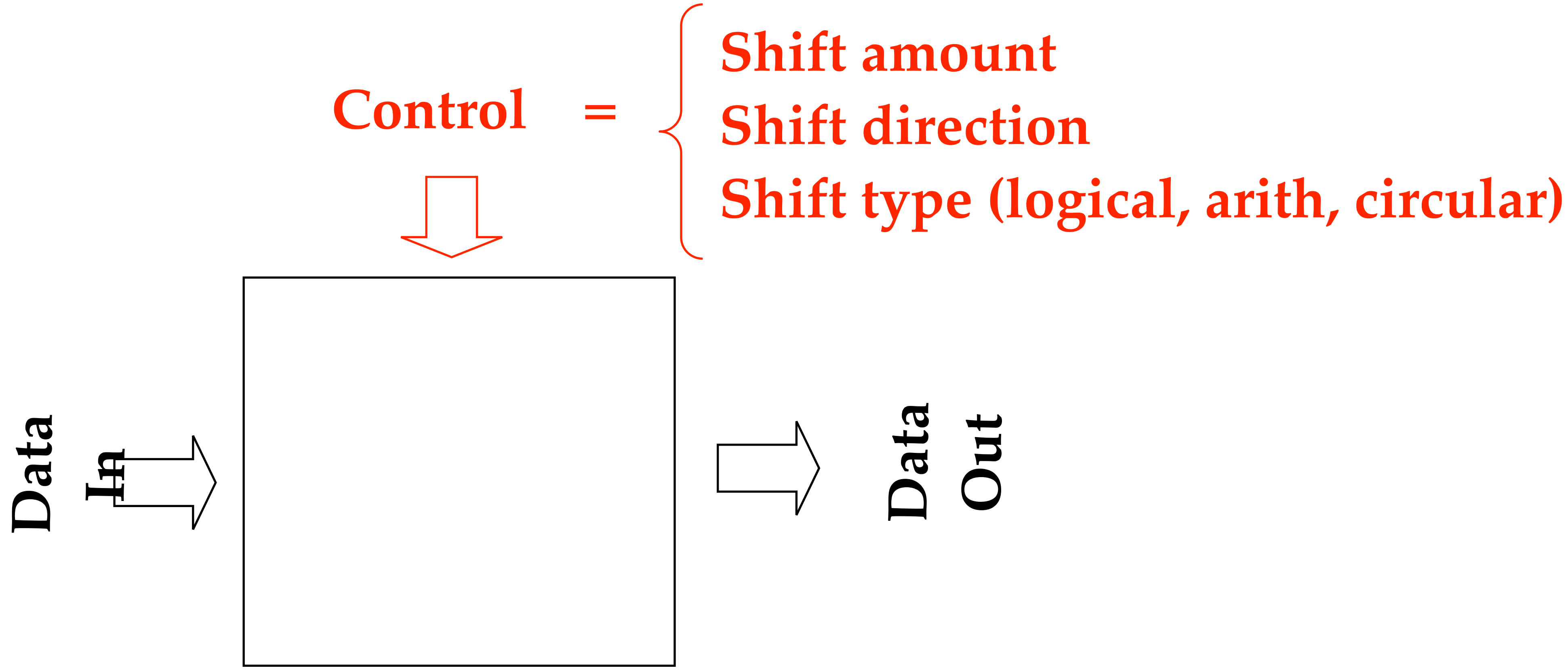
4-to-1 MUX



Shifter

Shifter

- A shifter shifts one bit position of its content to the left or right at a time, taking the input bit from the right or left when it shifts.



Control = Shift amount
Shift direction
Shift type (logical, arith, circular)

Data In

Data Out

Shifter Types

- Logical shifter
  - Shift the number to the left or right and fills empty spots with ***0's***
  - Ex: 1101, LSR 1=0110, LSL 1=1010
- Arithmetic shifter
  - Same as logical shifter but on right shift fills empty the ***MSBs*** with the sign bit (sign extension)
  - Ex: 1101, ASR 1=1110, ASL 1=1010
- Barrel shifter (rotator, cyclic shift)
  - Rotate numbers in a circle such that empty spots are filled with ***bits shifted off*** the other end
  - Ex: 1101, RSR 1=1110, RSL 1=1011