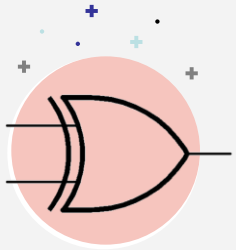
V1 (@2025)



Digital Circuits

Lecture 5:

Combinational Circuit Design

By: M.Razeghizadeh

Start now!

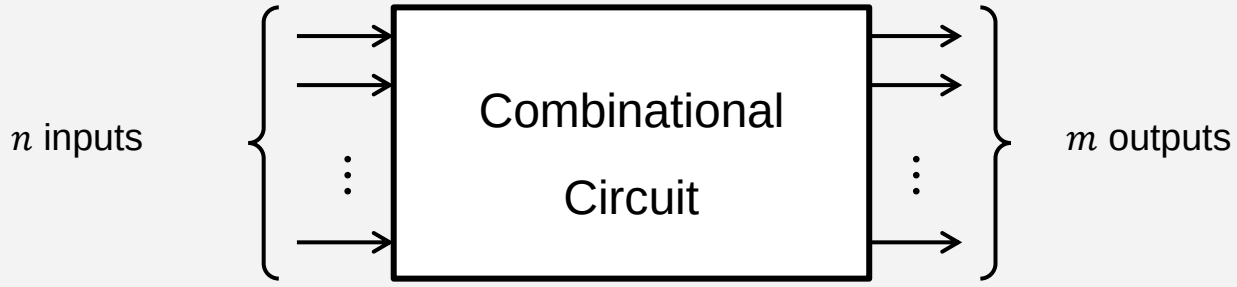
Boolean Function Minimization

- A combinational circuit is a block of logic gates having:

  n inputs: $x_1, x_2, \ldots, x_n$

  m outputs: $f_1, f_2, \ldots, f_m$

- Each output is a function of the input variables
- Each output is determined from **present combination** of inputs
- Combination circuit performs operation specified by logic gates



How to Design a Combinational Circuit

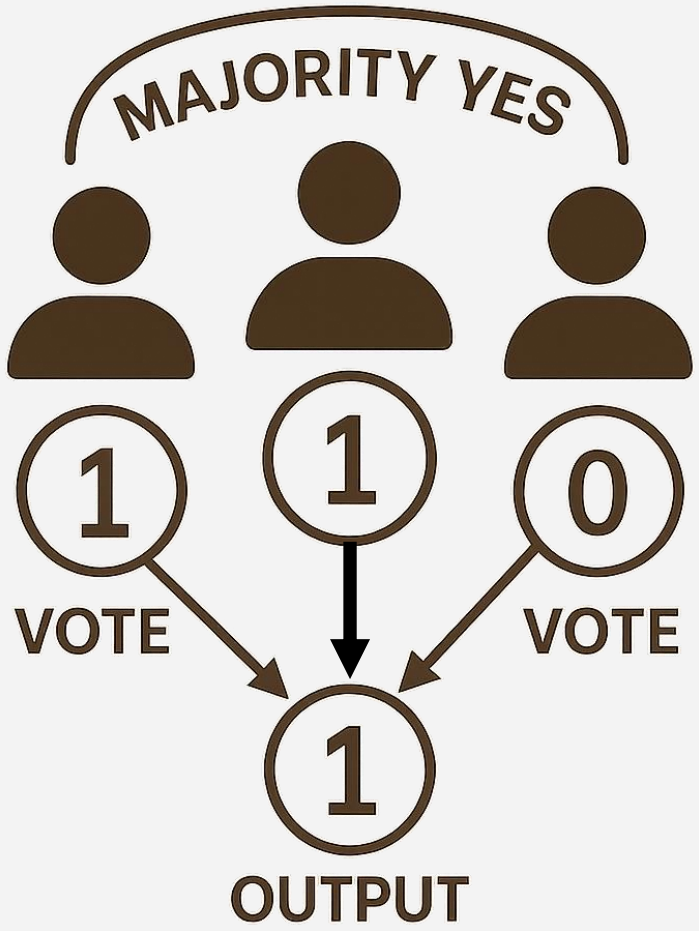
1. **Specification**
    1. Specify the inputs, outputs, and what the circuit should do
2. **Formulation**
    1. Convert the specification into truth tables or logic expressions for outputs
3. **Logic Minimization**
    1. Minimize the output functions using K-map or Boolean algebra
4. **Technology Mapping**
    1. Draw a logic diagram using ANDs, ORs, and inverters
    2. Map the logic diagram into the selected technology
    3. Considerations: cost, delays, fan-in, fan-out
5. **Verification**
    1. Verify the correctness of the design, either manually or using simulation

Designing a Decision-making Circuit

Problem Statement:

Three people vote on a single binary decision (yes = 1, no = 0).

The output should be 1 (majority yes) if at least two out of the three votes are 1.

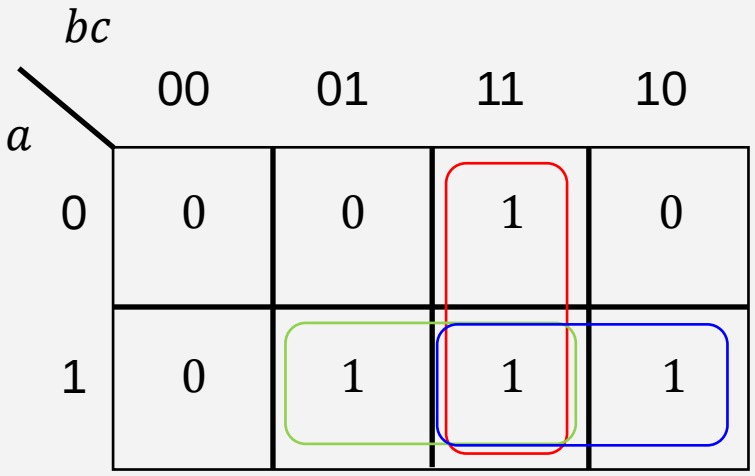


Designing a BCD to Excess-3 Code Converter

1. **Specification**
   1. The output should be 1 (majority yes) if at least two out of the three votes are 1.
   2. Input: Each represents one person's vote (1 = yes, 0 = no)
   3. Output: 1 if at least two votes are yes, otherwise 0
2. **Formulation**
   1. Done easily with a truth table
   2. Inputs: a, b, c
   3. Output: y

Inputs a b c	Outputs y
0 0 0	0
0 0 1	0
0 1 0	0
0 1 1	1
1 0 0	0
1 0 1	1
1 1 0	1
1 1 1	1

3. Logic Minimization of y using K-maps



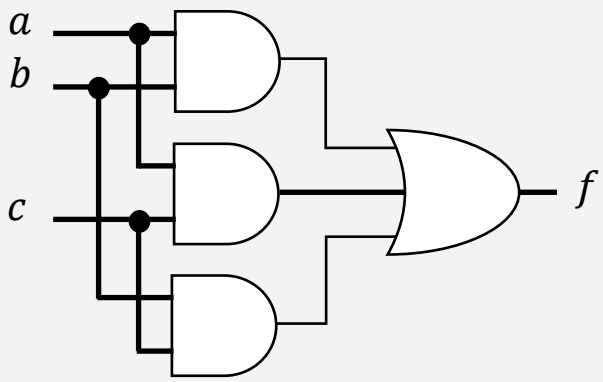
Minimal Sum-of-Product expressions:

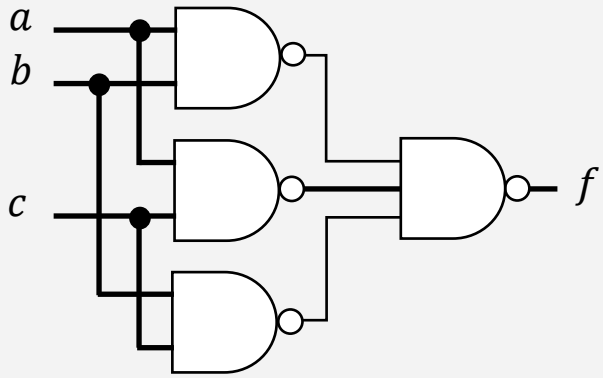
$y = bc + ac + ab$

4. Technology Mapping

- Draw a logic diagram using ANDs, ORs, and inverters
- Other gates can be used, such as NAND, NOR, and XOR

$$y = bc + ac + ab$$





Designing a BCD to Excess-3 Code Converter

5. Verification

Can be done manually

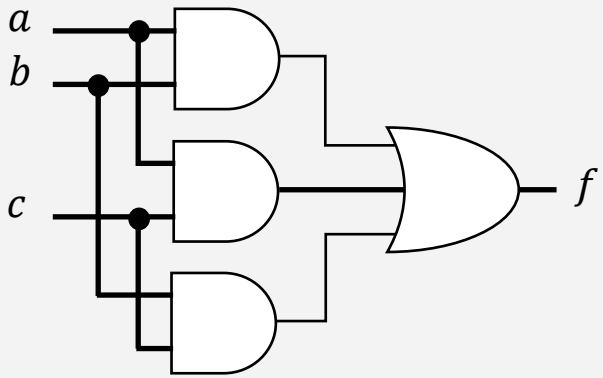
Extract output functions from circuit diagram

Find the truth table of the circuit diagram

Match it against the specification truth table

Verification process can be automated

Using a simulator for complex designs



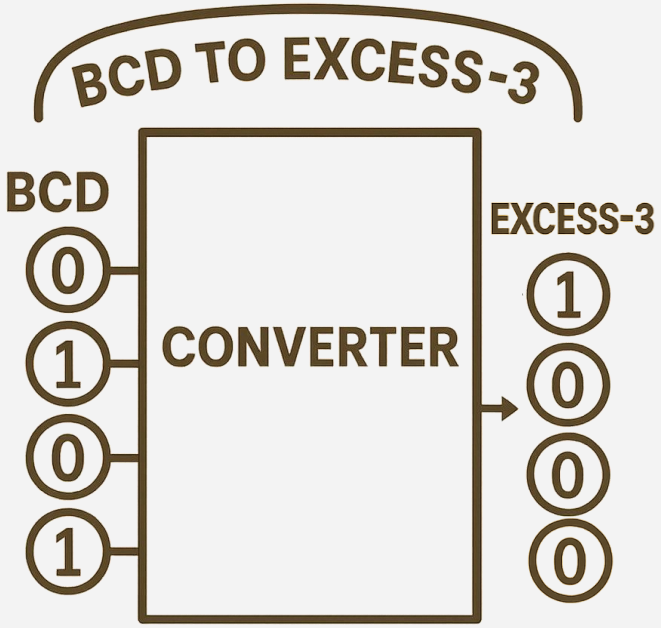
Truth Table of the Circuit Diagram

Inputs			Outputs
a	b	c	y
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Designing a BCD to Excess-3 Code Converter

Problem Statement:

Design a combinational circuit that converts a 4-bit Binary Coded Decimal (BCD) input into its corresponding Excess-3 (XS-3) code.



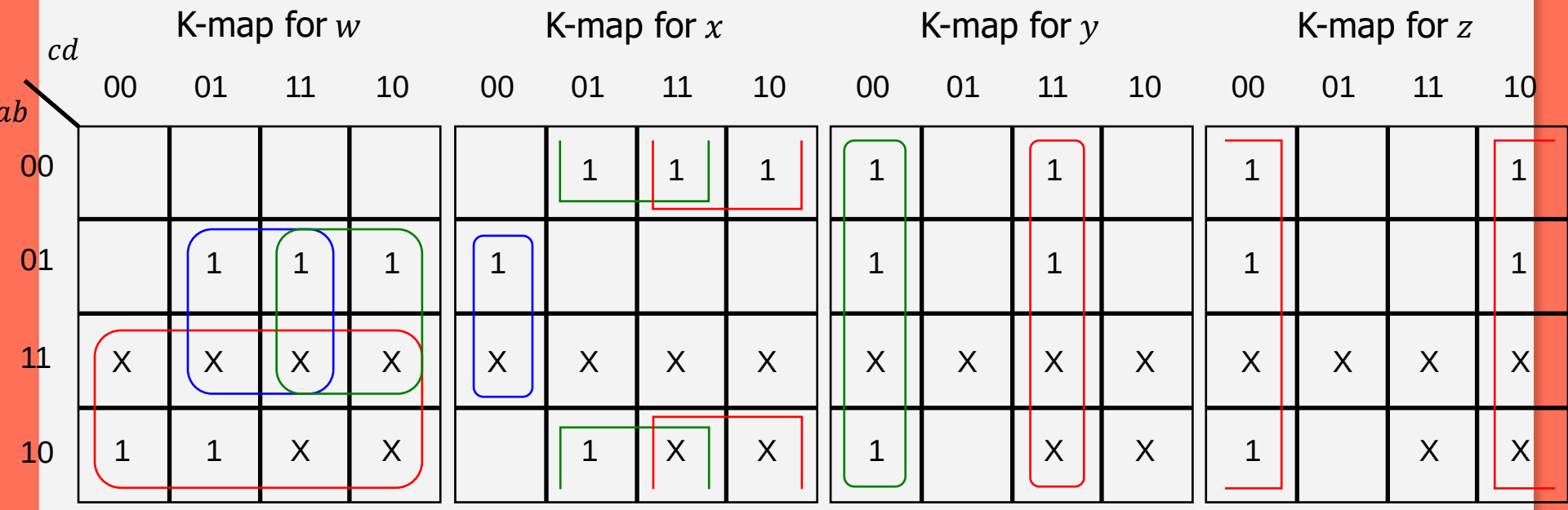
Designing a BCD to Excess-3 Code Converter

1. **Specification**
   1. Convert BCD code to Excess-3 code
   2. Input: BCD code for decimal digits 0 to 9
   3. Output: Excess-3 code for digits 0 to 9
2. **Formulation**
   1. Done easily with a truth table
   2. BCD input: a, b, c, d
   3. Excess-3 output: w, x, y, z
   4. Output is don't care for 1010 to 1111

BCD	Excess-3
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
1010 to 1111	X X X X

Designing a BCD to Excess-3 Code Converter

3. Logic Minimization using K-maps



Minimal Sum-of-Product expressions:

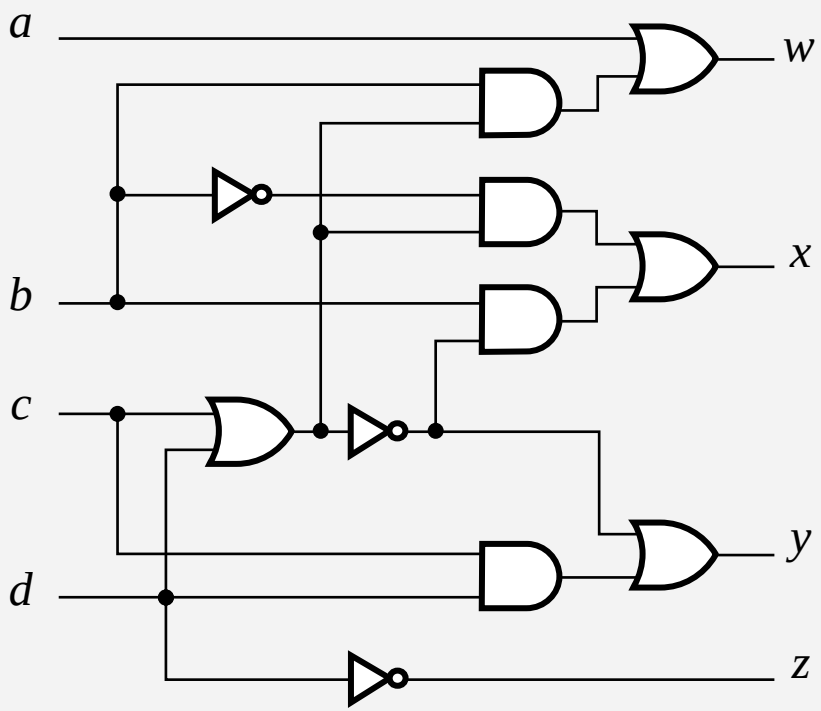
$w = a + bc + bd$, $x = b'c + b'd + bc'd'$, $y = cd + c'd'$, $z = d'$

Additional 3-Level Optimizations: extract common term $(c + d)$

$w = a + b(c + d)$, $x = b'(c + d) + b(c + d)'$, $y = cd + (c + d)'$

4. Technology Mapping

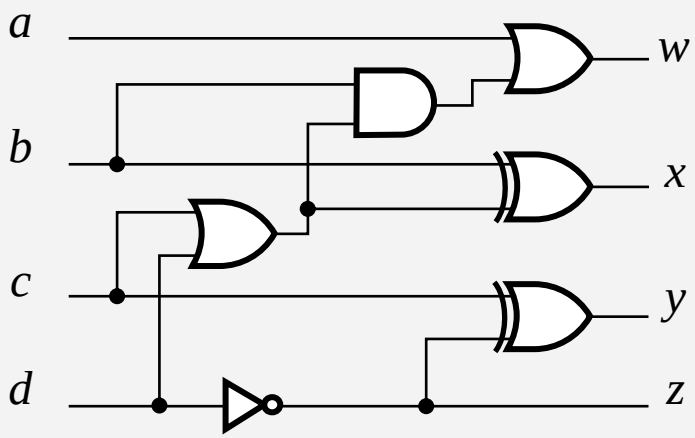
- Draw a logic diagram using ANDs, ORs, and inverters
- Other gates can be used, such as NAND, NOR, and XOR



Using XOR gates

$$x = b'(c + d) + b(c + d)' = b \oplus (c + d)$$

$$y = cd + c'd' = (c \oplus d)' = c \oplus d'$$



Designing a BCD to Excess-3 Code Converter

5. **Verification**

Can be done manually

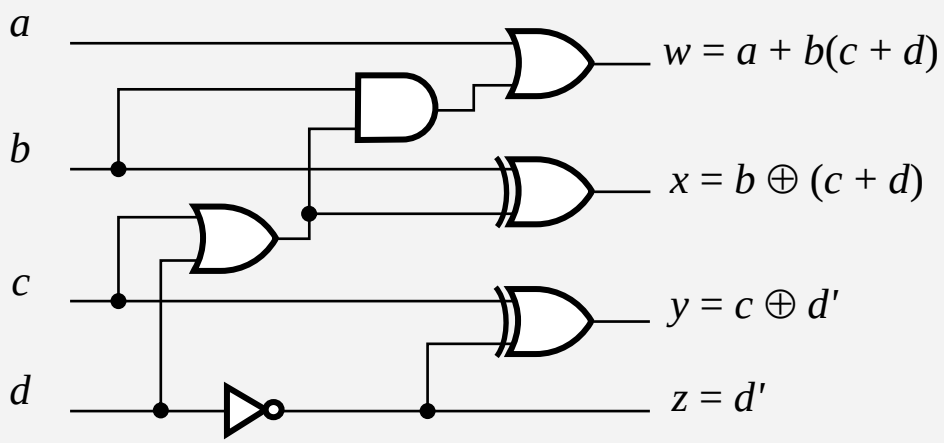
Extract output functions from circuit diagram

Find the truth table of the circuit diagram

Match it against the specification truth table

Verification process can be automated

Using a simulator for complex designs

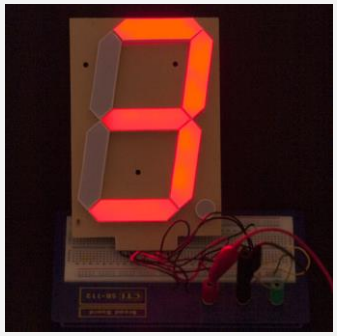


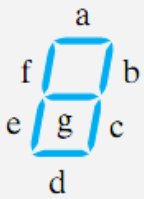
Truth Table of the Circuit Diagram

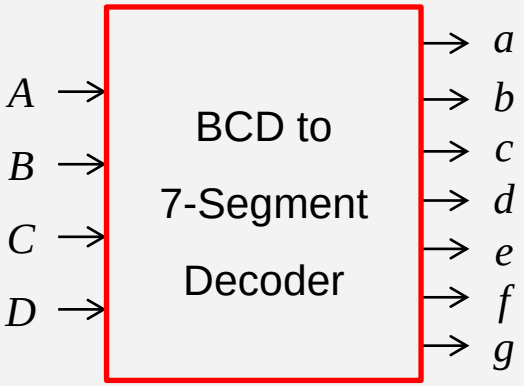
BCD a b c d	c+d	b(c+d)	Excess-3 w x y z
0 0 0 0	0	0	0 0 1 1
0 0 0 1	1	0	0 1 0 0
0 0 1 0	1	0	0 1 0 1
0 0 1 1	1	0	0 1 1 0
0 1 0 0	0	0	0 1 1 1
0 1 0 1	1	1	1 0 0 0
0 1 1 0	1	1	1 0 0 1
0 1 1 1	1	1	1 0 1 0
1 0 0 0	0	0	1 0 1 1
1 0 0 1	1	0	1 1 0 0

BCD to 7-Segment Decoder

- Seven-Segment Display:
  - Made of Seven segments: light-emitting diodes (LED)
  - Found in electronic devices: such as clocks, calculators, etc.







- ❖ BCD to 7-Segment Decoder
  - ✧ Accepts as input a BCD decimal digit (0 to 9)
  - ✧ Generates output to the seven LED segments to display the BCD digit
  - ✧ Each segment can be turned on or off separately

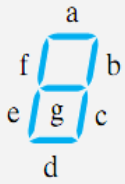
Designing a BCD to 7-Segment Decoder

1. Specification:

- Input: 4-bit BCD (A, B, C, D)
- Output: 7-bit (a, b, c, d, e, f, g)
- Display should be OFF for
- Non-BCD input codes

2. Formulation

- Done with a truth table
- Output is zero for 1010 to 1111

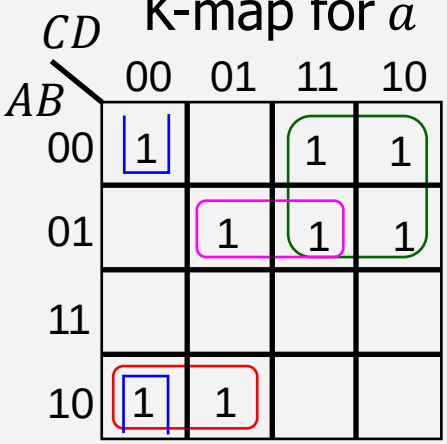


Truth Table

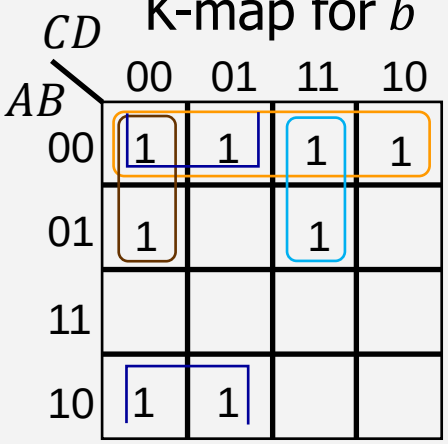
BCD input A B C D	7-Segment decoder a b c d e f g
0 0 0 0	1 1 1 1 1 1 0
0 0 0 1	0 1 1 0 0 0 0
0 0 1 0	1 1 0 1 1 0 1
0 0 1 1	1 1 1 1 0 0 1
0 1 0 0	0 1 1 0 0 1 1
0 1 0 1	1 0 1 1 0 1 1
0 1 1 0	1 0 1 1 1 1 1
0 1 1 1	1 1 1 0 0 0 0
1 0 0 0	1 1 1 1 1 1 1
1 0 0 1	1 1 1 1 0 1 1
1010 to 1111	0 0 0 0 0 0 0

3. Logic Minimization Using K-Maps

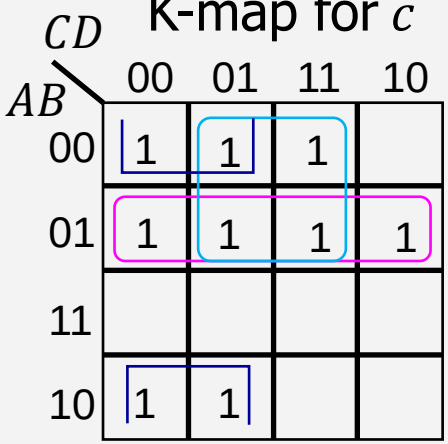
K-map for a



K-map for b



K-map for c



$$a = A'C + A'BD + AB'C' + B'C'D'$$

$$b = A'B' + B'C' + A'C'D' + A'CD$$

$$c = A'B + B'C' + A'D$$

Extracting common terms

Let $T_1 = A'B$, $T_2 = B'C'$, $T_3 = A'D$

Optimized Logic Expressions

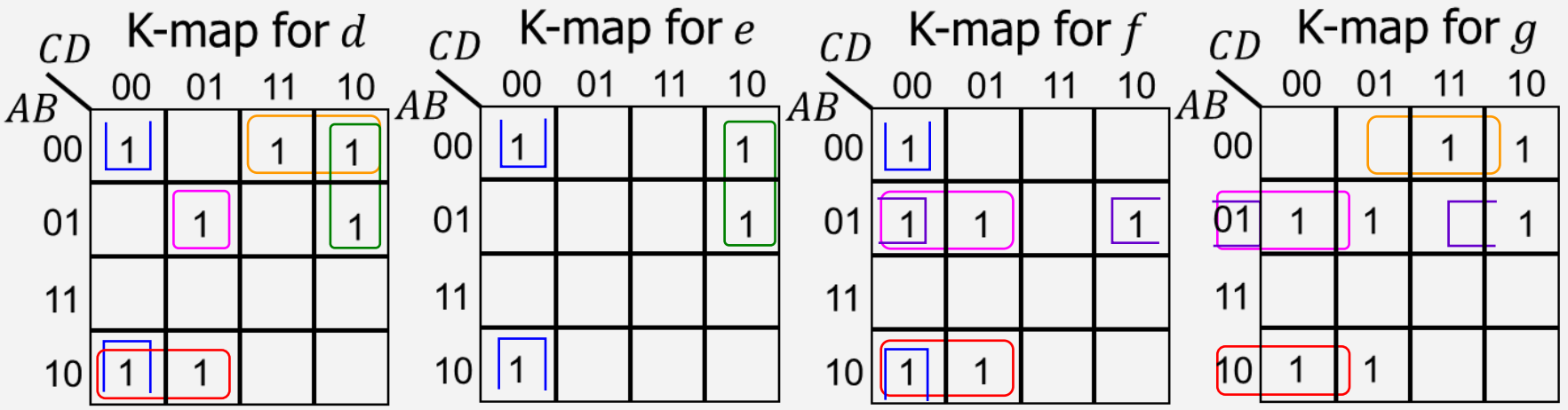
$$a = A'C + T_1\,D + T_2\,A + T_2\,D'$$

$$b = A'B' + T_2 + A'C'D' + T_3C$$

$$c = T_1 + T_2 + T_3$$

T_1, T_2, T_3 are **shared gates**

3. Logic Minimization Using K-Maps



Common AND Terms

➔ Shared Gates

$T_4 = AB'C'$, $T_5 = B'C'D'$

$T_6 = A'B'C$, $T_7 = A'CD'$

$T_8 = A'BC'$, $T_9 = A'BD'$

Optimized Logic Expressions

$d = T_4 + T_5 + T_6 + T_7 + T_8\,D$

$e = T_5 + T_7$

$f = T_4 + T_5 + T_8 + T_9$

$g = T_4 + T_6 + T_8 + T_9$

4. Technology Mapping

Many Common AND terms: T_0 thru T_9

$T_0 = A'C$, $T_1 = A'B$, $T_2 = B'C'$

$T_3 = A'D$, $T_4 = AB'C'$, $T_5 = B'C'D'$

$T_6 = A'B'C$, $T_7 = A'CD'$

$T_8 = A'BC'$, $T_9 = A'BD'$

Optimized Logic Expressions

$a = T_0 + T_1\,D + T_4 + T_5$

$b = A'B' + T_2 + A'C'D' + T_3C$

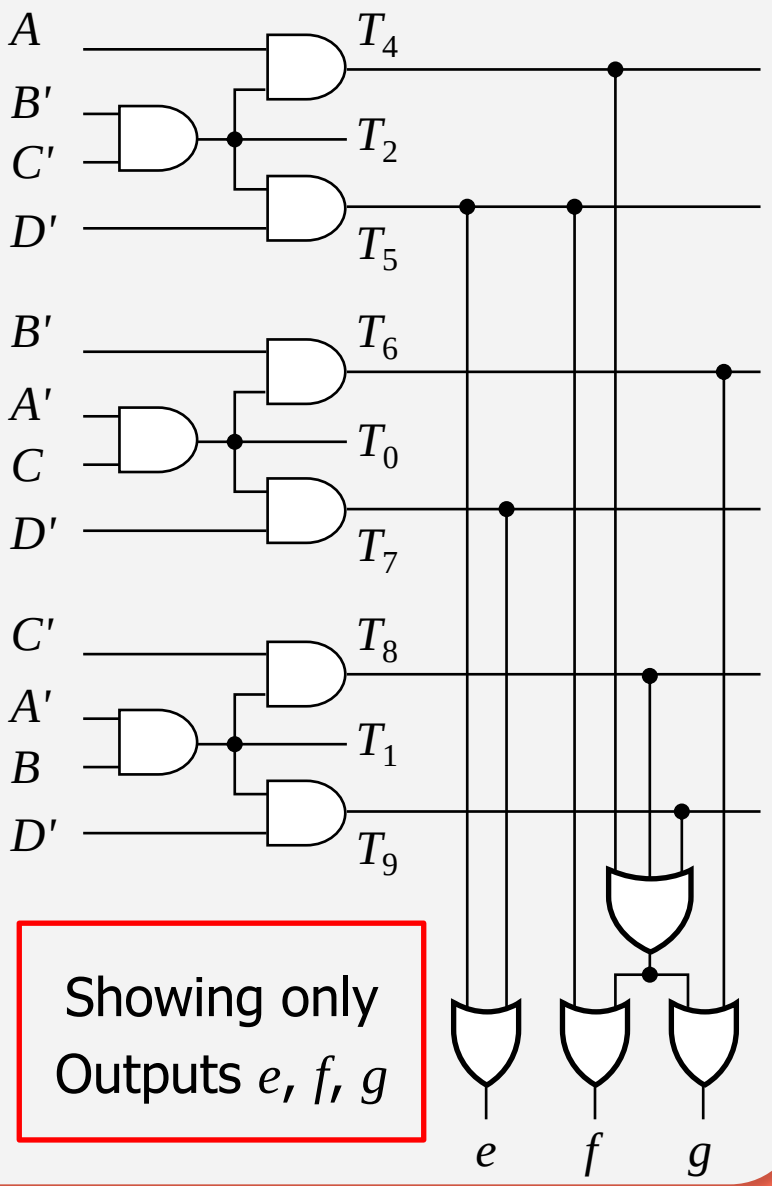
$c = T_1 + T_2 + T_3$

$d = T_4 + T_5 + T_6 + T_7 + T_8\,D$

$e = T_5 + T_7$

$f = T_4 + T_5 + T_8 + T_9$

$g = T_4 + T_6 + T_8 + T_9$



Showing only Outputs e, f, g

Designing a Half Adder Circuit

Problem Statement:

Design a combinational logic circuit that performs the addition of two single-bit binary numbers without considering any carry input.

Designing a Half Adder Circuit

1. **Specification**
   1. Input: Two 1-bit binary inputs
   2. Output:
      - Sum – The 1-bit sum output
      - Carry – The carry-out resulting from the addition

Inputs		Outputs	
a	b	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Formulation

1. Done easily without a truth table
2. Inputs: a, b
3. Output: s, c

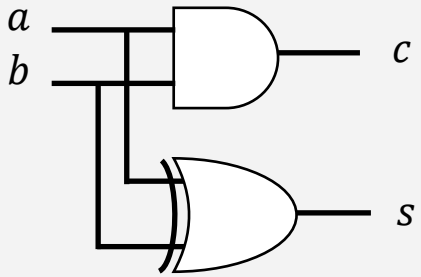
3. Logic Minimization of S , C

Minimal Sum-of-Product expressions:

$$s = ab' + a'b = a \oplus b$$
$$c = a \,.\, b$$

Inputs a b	Outputs S C
0 0	0 0
0 1	1 0
1 0	1 0
1 1	0 1

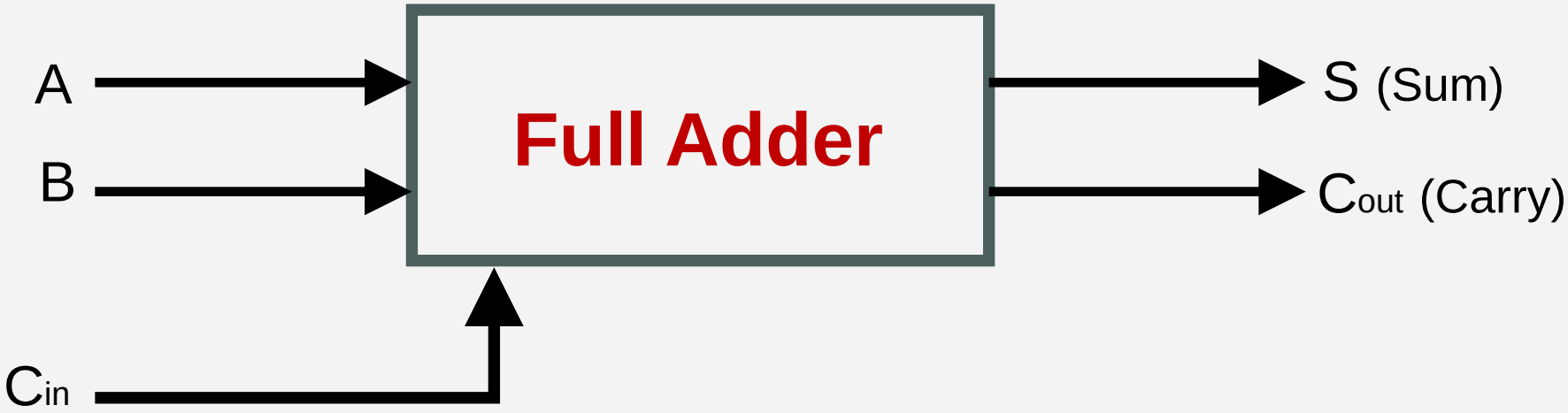
4. Technology Mapping



Designing a Full Adder Circuit

Problem Statement:

Design a combinational logic circuit called a Full Adder that performs the addition of three 1-bit binary numbers: two input bits and a carry-in bit. The circuit should produce a 1-bit sum and a 1-bit carry-out.



Designing a Full Adder Circuit

1. **Specification**
   1. Input: two input bits and a carry-in bit
   2. Output:
      - Sum – The least significant bit of the result
      - Carry – Carry-out to the next stage

Formulation

1. Done easily with a truth table
2. Inputs: a, b, c_{in}
3. Output: s, c_{out}

Inputs			Outputs	
c_{in}	a	b	S	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

3. Logic Minimization of S , Cout using K-maps

S

c_{in} \ ab	00	01	11	10
0	0	1	0	1
1	1	0	1	0

Cout

c_{in} \ ab	00	01	11	10
0	0	0	1	0
1	0	1	1	1

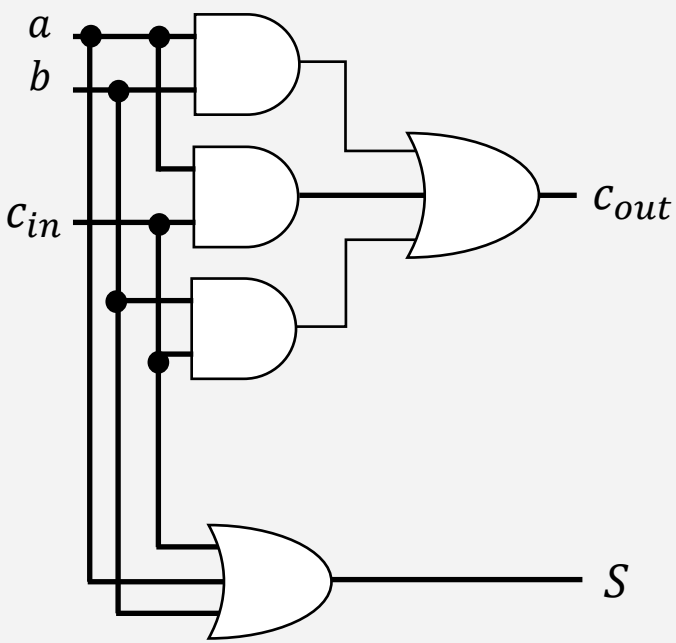
$c_{out} = ab + c_{in}\mathrm{b} + c_{in}\mathrm{a}$

$S = c_{in}a'b' + c_{in}'a'b + c_{in}ab + c_{in}ab'$

Minimal Sum-of-Product expressions:

$S = c_{in}a'b' + c_{in}'a'b + c_{in}ab + c_{in}'ab' = c_{in}(a'b' + ab) + c_{in}'(a'b + ab') = c_{in} \oplus a \oplus b$

4. Technology Mapping



Inputs			Outputs	
c_{in}	a	b	S	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

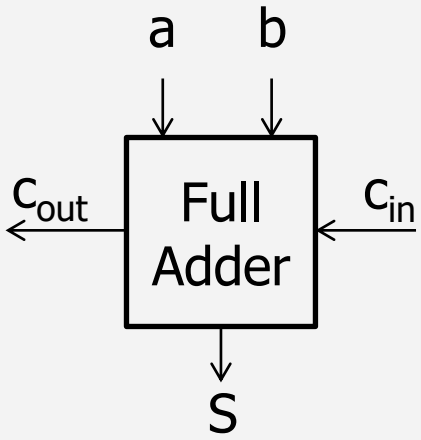
4. Technology Mapping

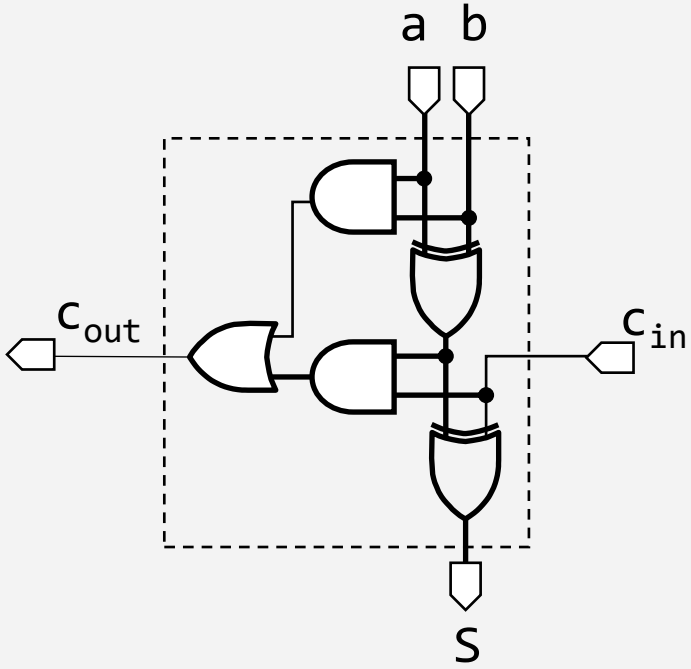
$$c_{out} = a'\ b\ \ c_{in} + a\ \ b'\ c_{in} + a\ \ b\ \ c'_{in} + a\ \ b\ \ c_{in}$$

$$c_{out} = (a'\ b\ \ + a\ \ b'\ )\ c_{in} + a\ \ b\ \ \ (c'_{in} + c_{in})$$

$$c_{out} = (\boldsymbol{a}\ \ \oplus \boldsymbol{b}\ \ )\ \boldsymbol{c_{in}} + \boldsymbol{a}\ \ \boldsymbol{b}$$

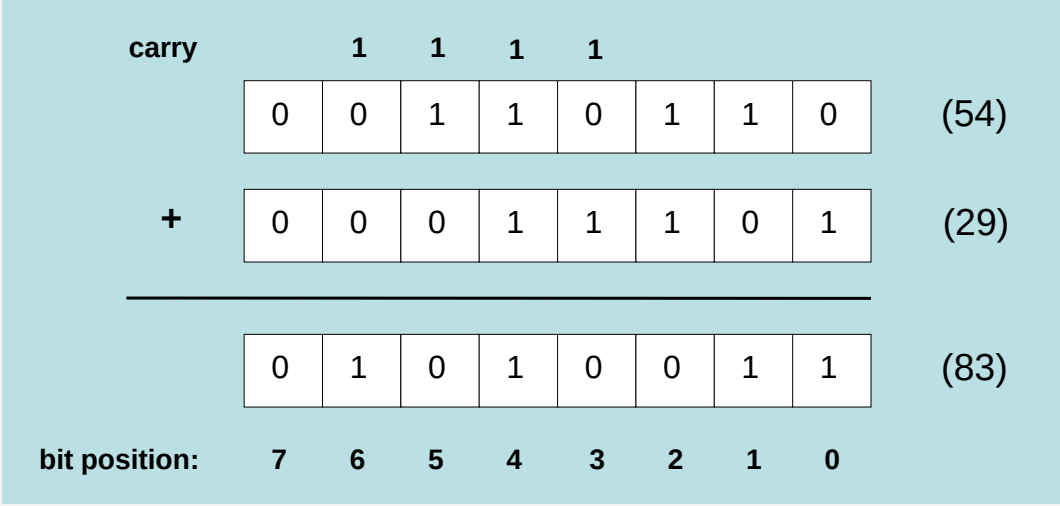
$$S = c_{in}a'b' + c_{in}'a'b + c_{in}ab + c_{in}'ab' = c_{in}(a'b' + ab) + c_{in}'(a'b + ab') = (\boldsymbol{a} \oplus \boldsymbol{b}) \oplus \boldsymbol{c_{in}}$$





Binary Addition

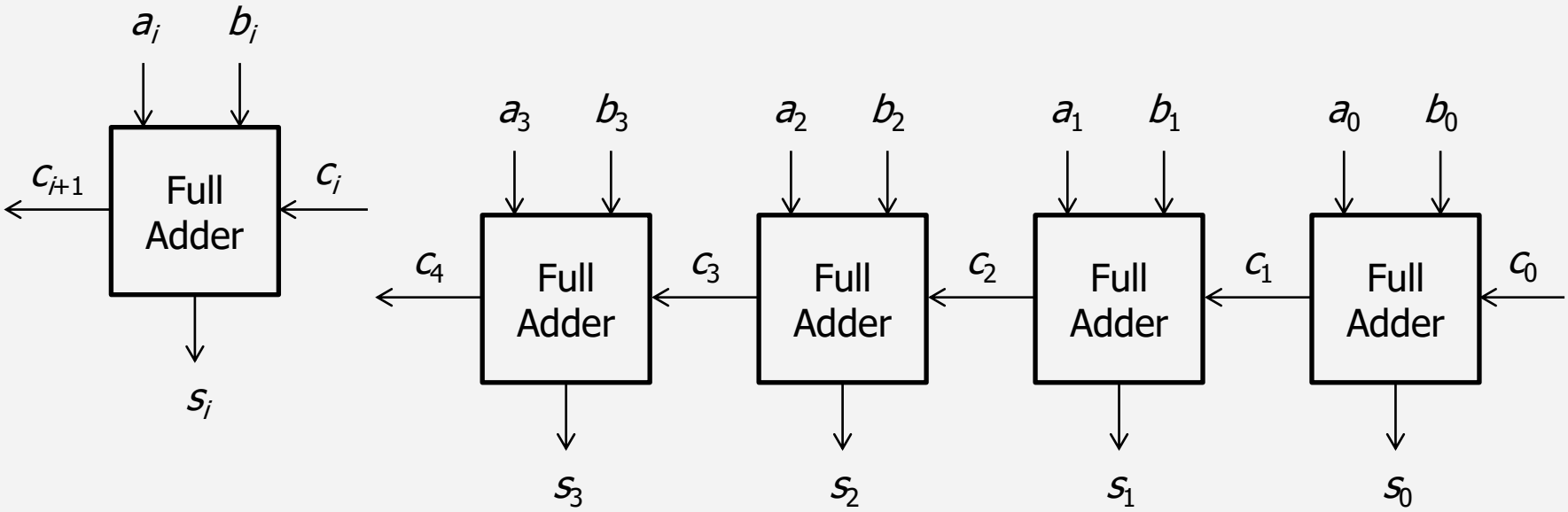
- Start with the least significant bit (rightmost bit)
- Add each pair of bits
- Include the carry in the addition



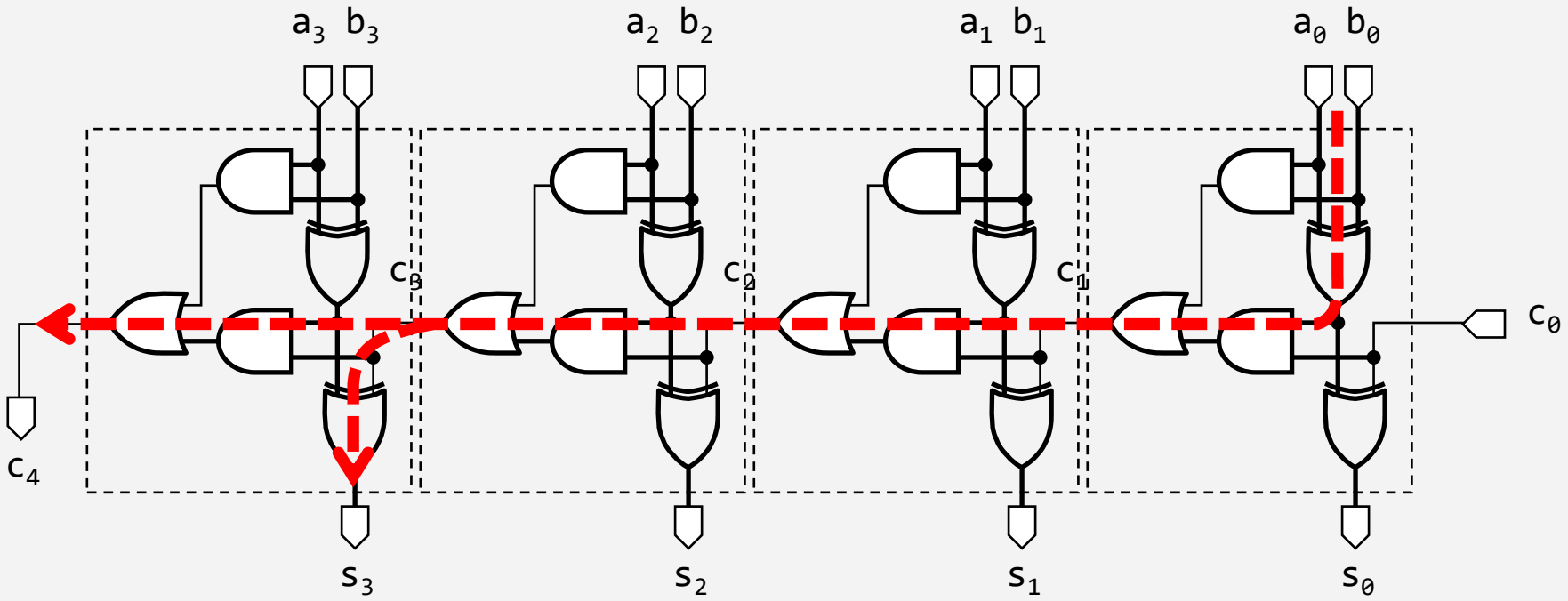
Iterative Design: Ripple Carry Adder

- Uses **identical copies** of a full adder to build a large adder
- Simple to implement: can be extended to add any number of bits
- The **cell** (iterative block) is a **full adder**

  Adds 3 bits: a_i b_i c_i Computes: Sum s_i and Carry-out c_{i+1}
- Carry-out of cell i becomes carry-in to cell (i +1)

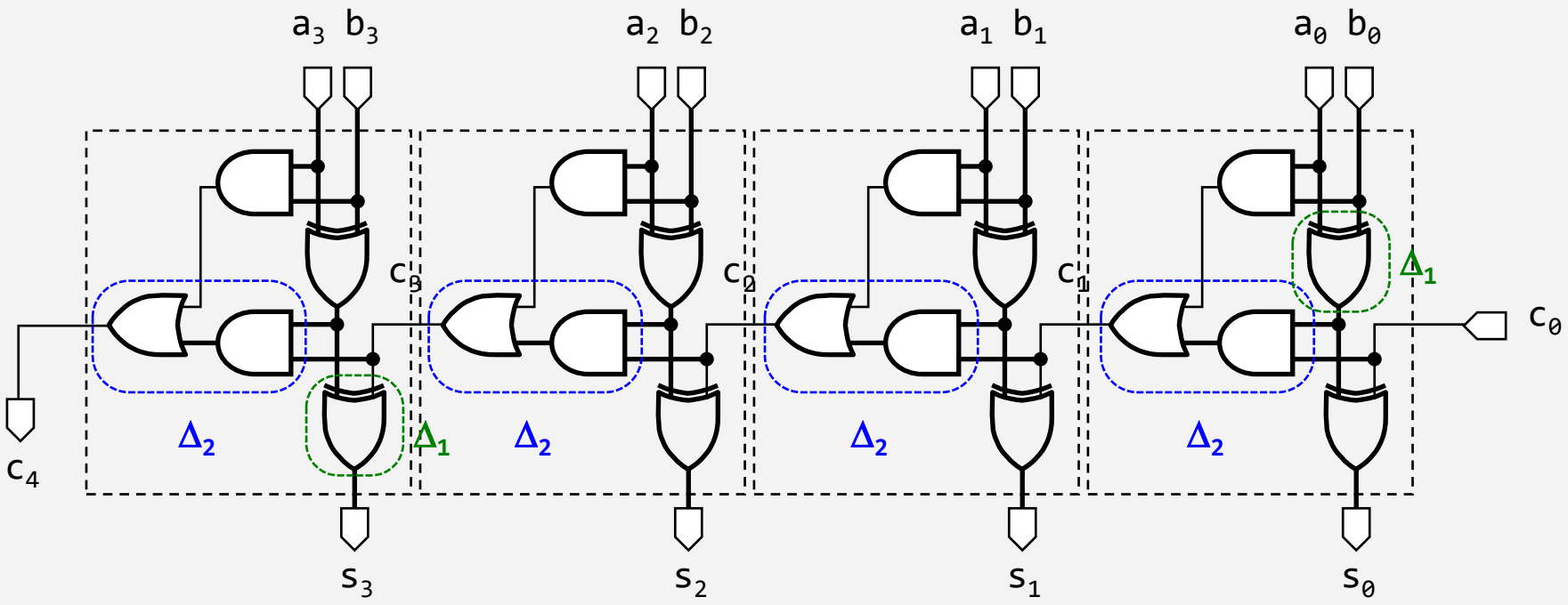


Carry Propagation



- Major drawback of ripple-carry adder is the **carry propagation**
- The carries are connected in a chain through the full adders
- This is why it is called a **ripple-carry adder**
- The **carry ripples** (propagates) through all the full adders

Longest Delay Analysis



Suppose the XOR delay is Δ_1 and AND-OR delay is Δ_2

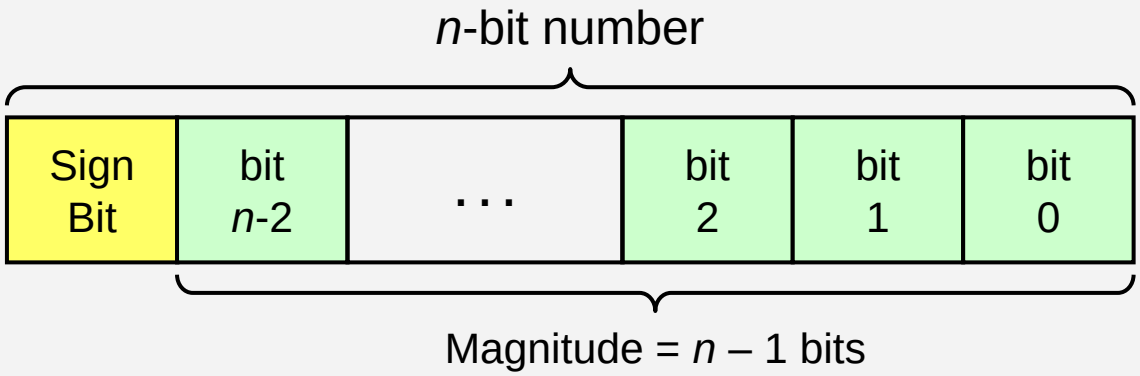
For an *N*-bit ripple-carry adder, if all inputs are present at once:

1. Most-significant sum-bit delay = $2\Delta_1 + (N - 1)\,\Delta_2$
2. Final Carry-out delay = $\Delta_1 + N\,\Delta_2$

Signed Numbers

- Several ways to represent a signed number
  - Sign-Magnitude
  - 1's complement
  - 2's complement
- Divide the range of values into two parts
  - First part corresponds to the positive numbers (≥ 0)
  - Second part correspond to the negative numbers (< 0)
- The 2's complement representation is widely used
  - Has many advantages over other representations

Sign-Magnitude Representation



- Independent representation of the sign and magnitude
- Leftmost bit is the sign bit: 0 is positive and 1 is negative
- Using n bits, largest represented magnitude = $2^{n-1} - 1$

Sign-magnitude
8-bit representation of +45

0	0	1	0	1	1	0	1

Sign-magnitude
8-bit representation of -45

1	0	1	0	1	1	0	1

Properties of Sign-Magnitude

- Symmetric range of represented values:

  For n-bit register, range is from $-(2^{n-1} - 1)$ to $+(2^{n-1} - 1)$

  For example, if $n = 8$ bits then range is -127 to +127

- Two representations for zero: +0 and -0 **NOT Good!**
- Two circuits are needed for addition & subtraction **NOT Good!**
  - In addition to an adder, a second circuit is needed for subtraction
  - Sign and magnitude parts should be processed independently
  - Sign bit should be examined to determine addition or subtraction
  - Addition of numbers of different signs is converted into subtraction
  - Increases the cost of the add/subtract circuit

Sign-Magnitude Addition / Subtraction

Eight cases for Sign-Magnitude Addition / Subtraction

Operation	ADD Magnitudes	Subtract Magnitudes	
		A >= B	A < B
(+A) + (+B)	+(A+B)		
(+A) + (-B)		+(A–B)	-(B–A)
(-A) + (+B)		-(A–B)	+(B–A)
(-A) + (-B)	-(A+B)		
(+A) – (+B)		+(A–B)	-(B–A)
(+A) – (-B)	+(A+B)		
(-A) – (+B)	-(A+B)		
(-A) – (-B)		-(A–B)	+(B–A)

1's Complement Representation

- Given a binary number *A*

  The 1's complement of *A* is obtained by inverting each bit in *A*

- Example: 1's complement of $(01101001)_2 = (10010110)_2$

- If *A* consists of *n* bits then:

  $A + (1\text{'s complement of } A) = (2^n - 1) = (1\ldots111)_2$ (all bits are 1's)

- Range of values is $-(2^{n-1} - 1)$ to $+(2^{n-1} - 1)$

  For example, if $n = 8$ bits, range is -127 to +127

- Two representations for zero: +0 and -0 **NOT Good!**

  1's complement of $(0\ldots000)_2 = (1\ldots111)_2 = 2^n - 1$

  $-0 = (1\ldots111)_2$ **NOT Good!**

2's Complement Representation

- Standard way to represent signed integers in computers
- A simple definition for 2's complement:

  Given a binary number A

  The 2's complement of A = (1's complement of A) + 1
- Example: 2's complement of $(01101001)_2$ =

  $(10010110)_2 + 1 = (10010111)_2$
- If A consists of n bits then

  A + (2's complement of A) = 2^n

  2's complement of $A = 2^n - A$

Computing the 2's Complement

starting value	00100100_2 = +36
step1: Invert the bits (1's complement)	11011011_2
step 2: Add 1 to the value from step 1	+ 1_2
sum = 2's complement representation	11011100_2 = -36

2’s complement of 11011100_2 (-36) = 00100011_2 + 1 = 00100100_2 = +36

The 2’s complement of the 2’s complement of *A* is equal to *A*

Properties of the 2's Complement

- Range of represented values: -2^{n-1} to $+(2^{n-1} - 1)$

  For example, if $n = 8$ bits then range is -128 to +127

- There is only **one zero** = $(0\ldots000)_2$ (all bits are zeros)
- The 2's complement of *A* is the **negative of *A***
- The sum of *A* + (2's complement of *A*) **must be zero**

  The final carry is ignored

- Consider the 8-bit number $A = 00101100_2 = +44$

  2's complement of $A = 11010100_2 = -44$

  $00101100_2 + 11010100_2 = \mathbf{1}\ 00000000_2$ (8-bit sum is 0)

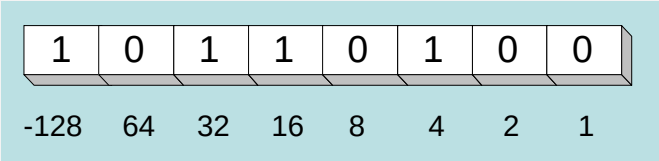
  Ignore final carry = 2^8

Values of Different Representations

8-bit Binary Representation	Unsigned Value	Sign Magnitude Value	1's Complement Value	2's Complement Value
00000000	0	+0	+0	0
00000001	1	+1	+1	+1
00000010	2	+2	+2	+2
. . .	. . .	. . .	. . .	. . .
01111101	125	+125	+125	+125
01111110	126	+126	+126	+126
01111111	127	+127	+127	+127
10000000	128	-0	-127	-128
10000001	129	-1	-126	-127
10000010	130	-2	-125	-126
. . .	. . .	. . .	. . .	. . .
11111101	253	-125	-2	-3
11111110	254	-126	-1	-2
11111111	255	-127	-0	-1

Binary Addition

- Positive numbers (sign-bit = 0)
  - Signed value = Unsigned value
- Negative numbers (sign-bit = 1)
  - Signed value = Unsigned value – 2^n
  - n = number of bits
- Negative weight for sign bit
  - The 2's complement representation assigns a negative weight to the sign bit (most-significant bit)



-128 + 32 + 16 + 4 = -76

8-bit Binary	Unsigned Value	Signed Value
00000000	0	0
00000001	1	+1
00000010	2	+2
. . .	. . .	. . .
01111101	125	+125
01111110	126	+126
01111111	127	+127
10000000	128	-128
10000001	129	-127
10000010	130	-126
. . .	. . .	. . .
11111101	253	-3
11111110	254	-2
11111111	255	-1

Converting Subtraction into Addition

- When computing **A – B**, convert **B** to its 2's complement

  A – B = A + (2's complement of B)

- **Same adder** is used for **both addition and subtraction**

  This is the biggest advantage of 2's complement

borrow:		-1	-1		-1			**carry:**	**1**	1			1	1	
		0	**1**	**0**	**0**	**1**	**1**	**0**	**1**			**0**	**1**	**0**	**0**	**1**	**1**	**0**	**1**	
–		**0**	**0**	**1**	**1**	**1**	**0**	**1**	**0**	⇨	**+**	**1**	**1**	**0**	**0**	**0**	**1**	**1**	**0**	(2's complement)
		0	**0**	**0**	**1**	**0**	**0**	**1**	**1**			**0**	**0**	**0**	**1**	**0**	**0**	**1**	**1**	(same result)

- Final carry is **ignored**, because

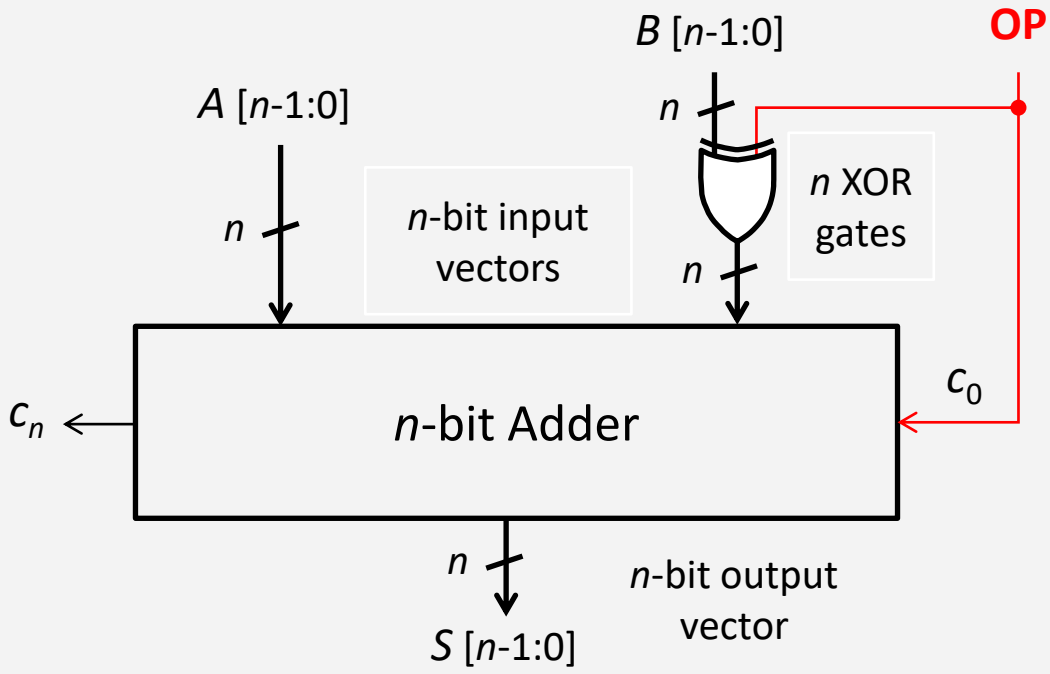
  A + (2's complement of B) = A + (2^n – B) = (A – B) + 2^n

  Final carry = 2^n, for *n*-bit numbers

Adder/Subtractor for 2's Complement

- Same adder is used to compute: (A + B) or (A – B)
- Subtraction (A – B) is computed as: A + (2's complement of B)

  2's complement of B = (1's complement of B) + 1
- Two operations: **OP = 0 (ADD)**, **OP = 1 (SUBTRACT)**



OP = 0 (ADD)

B XOR 0 = B

S = A + B + 0 = A + B

OP = 1 (SUBTRACT)

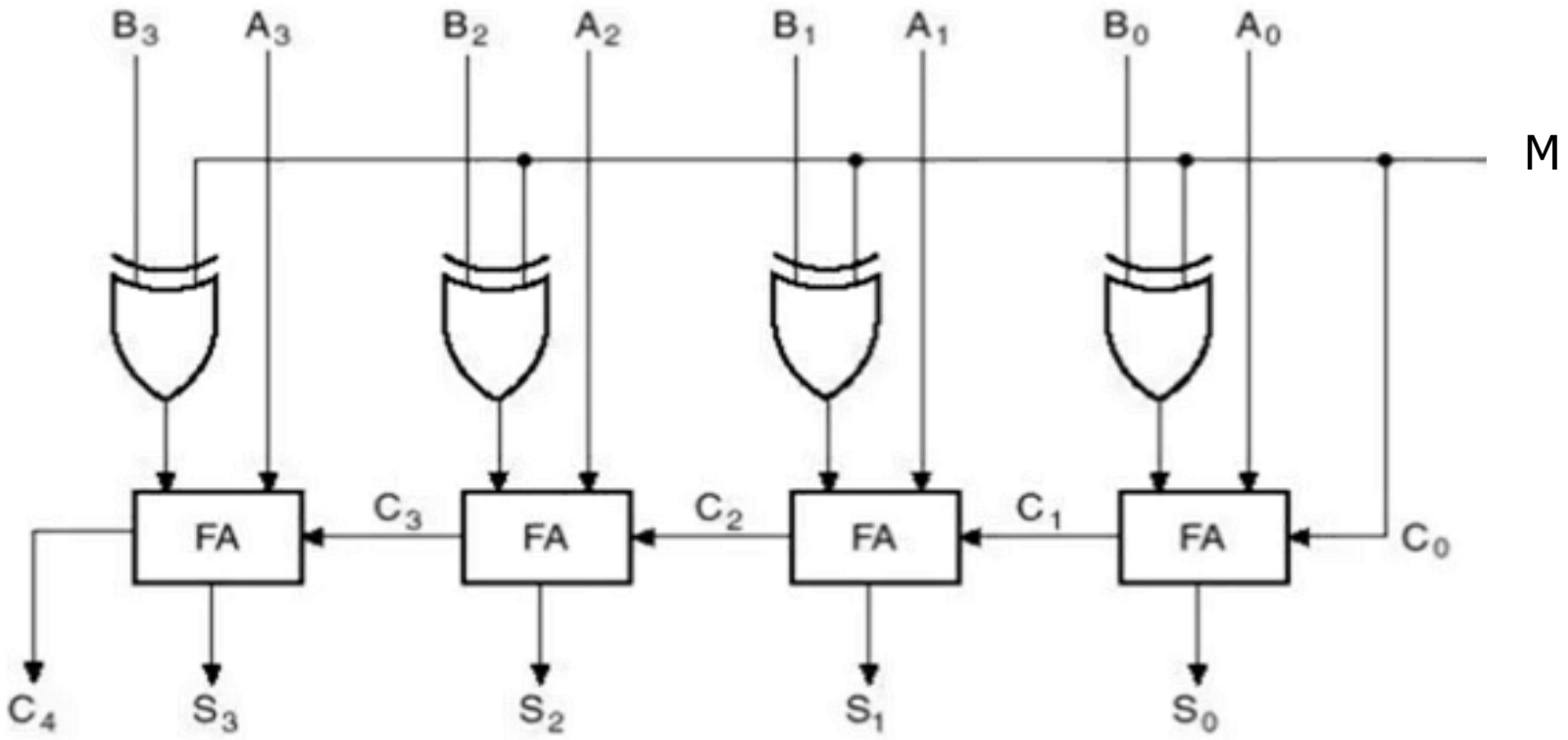
B XOR 1 = 1's complement of B

S = A + (1's complement of B) + 1

S = A + (2's complement of B)

S = A – B

Adder/Subtractor for 2's Complement



Carry versus Overflow

- Carry is important when …
  - Adding **unsigned integers**
  - Indicates that the **unsigned sum** is out of range
  - Sum > maximum unsigned *n*-bit value
- Overflow is important when …
  - Adding or subtracting **signed integers**
  - Indicates that the **signed sum** is out of range
- Overflow occurs when …
  - Adding two positive numbers and the sum is negative
  - Adding two negative numbers and the sum is positive
- Simplest way to detect Overflow: $V = C_{n-1} \oplus C_n$
  - C_{n-1} and C_n are the carry-in and carry-out of the most-significant bit

Carry and Overflow Examples

- We can have carry without overflow and vice-versa
- Four cases are possible (Examples on 8-bit numbers)

				1					
	0	0	0	0	1	1	1	1	15
+	0	0	0	0	1	0	0	0	8
	0	0	0	1	0	1	1	1	23

Carry = 0 Overflow = 0

	1	1	1	1	1				
	0	0	0	0	1	1	1	1	15
+	1	1	1	1	1	0	0	0	248 (-8)
	0	0	0	0	0	1	1	1	7

Carry = 1 Overflow = 0

		1							
	0	1	0	0	1	1	1	1	79
+	0	1	0	0	0	0	0	0	64
	1	0	0	0	1	1	1	1	143 (-113)

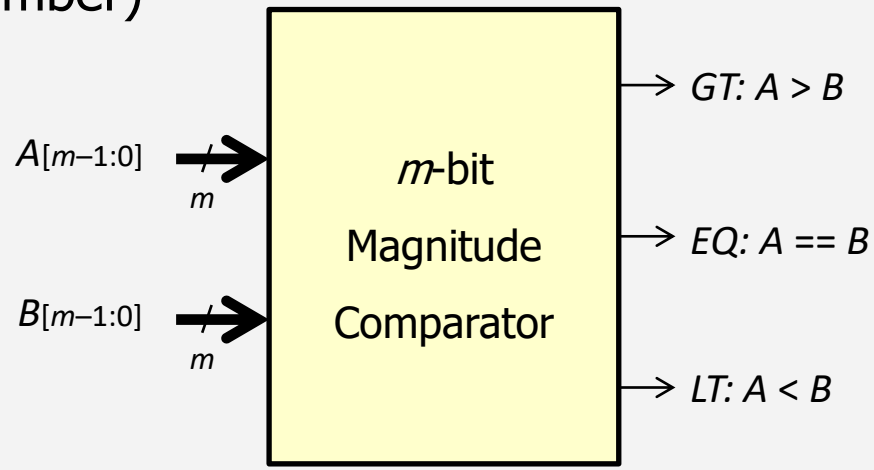
Carry = 0 Overflow = 1

	1			1	1				
	1	1	0	1	1	0	1	0	218 (-38)
+	1	0	0	1	1	1	0	1	157 (-99)
	0	1	1	1	0	1	1	1	119

Carry = 1 Overflow = 1

Magnitude Comparator

- A combinational circuit that compares two unsigned integers
- Two Inputs:
  - Unsigned integer *A* (*m*-bit number)
  - Unsigned integer *B* (*m*-bit number)
- Three outputs:
  - A > B (GT output)
  - A == B (EQ output)
  - A < B (LT output)



- Exactly one of the three outputs must be equal to 1
- While the remaining two outputs must be equal to 0

2-bit Magnitude Comparator

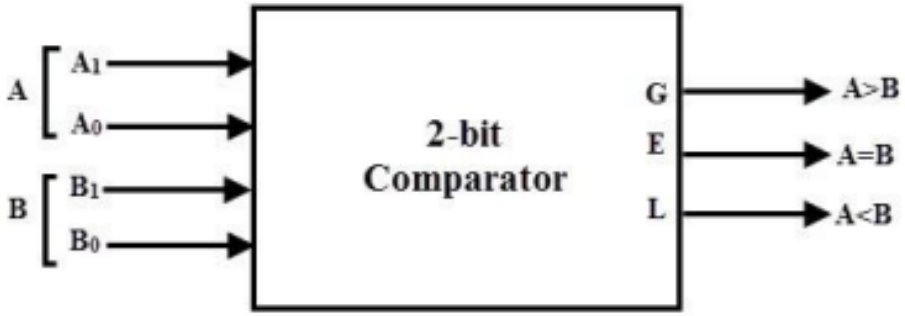
- ✓ Compares two binary numbers
- ✓ Each of two bits
- ✓ Produces result such as -

One number is Equal to / Greater than / Less than the other

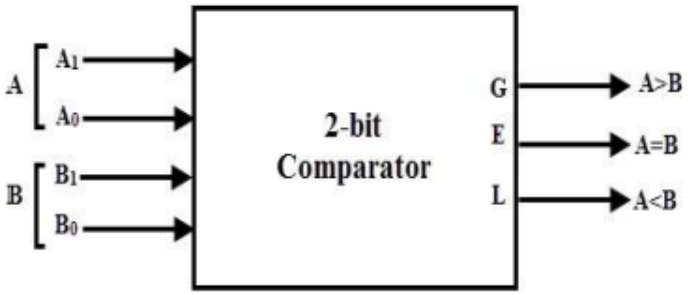
Block Diagram (Two-bit comparator) :

Number A = A1 A0
and
Number B = B1 B0

Produces 3 outputs as -
G -- (G = 1 if A>B)
E -- (E = 1, if A = B)
L -- (L = 1 if A<B)



2-bit Magnitude Comparator



INPUTS				OUTPUTS		
A1	A0	B1	B0	A>B	A=B	A<B
0	0	0	0	0	1	0
0	0	0	1	0	0	1
0	0	1	0	0	0	1
0	0	1	1	0	0	1
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	1
1	0	0	0	1	0	0
1	0	0	1	1	0	0
1	0	1	0	0	1	0
1	0	1	1	0	0	1
1	1	0	0	1	0	0
1	1	0	1	1	0	0
1	1	1	0	1	0	0
1	1	1	1	0	1	0

2-bit Magnitude Comparator

A1A0 \ B1B0	00	01	11	10
00	1	0	0	0
01	0	1	0	0
11	0	0	1	0
10	0	0	0	1

$$A=B: E=\overline{A1}\ \overline{A0}\ \overline{B1}\ \overline{B0}+\overline{A1}\ A0\ \overline{B1}\ B0+A1\ A0\ B1\ B0+A1\ \overline{A0}\ B1\ \overline{B0}$$

$$=\overline{A1}\ \overline{B1}\ (\overline{A0}\ \overline{B0}+A0\ B0)+A1\ B1\ (A0\ B0+\overline{A0}\ \overline{B0})$$

$$=(A0\ B0+\overline{A0}\ \overline{B0})\ (A1\ B1+\overline{A1}\ \overline{B1})$$

$$=(A0 \text{ Ex-NOR } B0)\ (A1 \text{ Ex-NOR } B1)$$

$$(\overline{A0 \oplus B0})\quad (\overline{A1 \oplus B1})$$

INPUTS				O/P
A1	A0	B1	B0	A=B
0	0	0	0	1
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	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	1
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

2-bit Magnitude Comparator

B1B0 / A1A0	00	01	11	10
00	0	0	0	0
01	1	0	0	0
11	1	1	0	1
10	1	1	0	0

$A>B: G = A0\,\overline{B1}\;\overline{B0} + A1\,\overline{B1} + A1\,A0\,\overline{B0}$

INPUTS				O/P
A1	A0	B1	B0	A>B
0	0	0	0	**0**
0	0	0	1	**0**
0	0	1	0	**0**
0	0	1	1	**0**
0	1	0	0	**1**
0	1	0	1	**0**
0	1	1	0	**0**
0	1	1	1	**0**
1	0	0	0	**1**
1	0	0	1	**1**
1	0	1	0	**0**
1	0	1	1	**0**
1	1	0	0	**1**
1	1	0	1	**1**
1	1	1	0	**1**
1	1	1	1	**0**

2-bit Magnitude Comparator

B1B0 / A1A0	00	01	11	10
00	0	1	1	1
01	0	0	1	1
11	0	0	0	0
10	0	0	1	0

$$A < B: L = \overline{A1}\ B1 + \overline{A0}\ B1\ B0 + \overline{A1}\ \overline{A0}\ B0$$

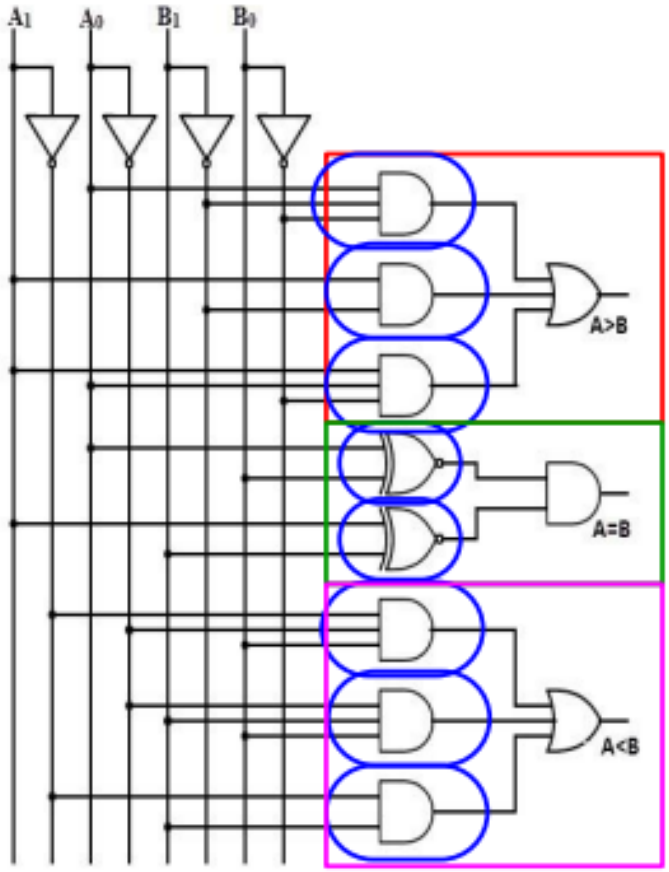
INPUTS				O/P
A1	A0	B1	B0	A<B
0	0	0	0	**0**
0	0	0	1	**1**
0	0	1	0	**1**
0	0	1	1	**1**
0	1	0	0	**0**
0	1	0	1	**0**
0	1	1	0	**1**
0	1	1	1	**1**
1	0	0	0	**0**
1	0	0	1	**0**
1	0	1	0	**0**
1	0	1	1	**1**
1	1	0	0	**0**
1	1	0	1	**0**
1	1	1	0	**0**
1	1	1	1	**0**

2-bit Magnitude Comparator

$$A>B:\ G = A0\,\overline{B1}\ \overline{B0} + A1\,\overline{B1} + A1\ A0\,\overline{B0}$$

$$A=B:\ E = \overline{(A0 \oplus B0)}\ \overline{(A1 \oplus B1)}$$

$$A<B:\ L = \overline{A1}\ B1 + \overline{A0}\,B1\,B0 + \overline{A1}\ \overline{A0}\,B0$$



4-bit Magnitude Comparator

- Inputs:
  - $A = A_3A_2A_1A_0$
  - $B = B_3B_2B_1B_0$
  - 8 bits in total ➔ 256 possible combinations
  - Not simple to design using conventional K-map techniques
- The magnitude comparator can be designed at a higher level
- Let us implement first the EQ output (A is equal to B)
  - $EQ = 1 \leftrightarrow A_3 == B_3$, $A_2 == B_2$, $A_1 == B_1$, and $A_0 == B_0$
  - Define: $E_i = (A_i == B_i) = A_iB_i + A_i'B_i'$
  - Therefore, $EQ = (A == B) = E_3E_2E_1E_0$

The Greater Than Output

Given the 4-bit input numbers: A and B

1. If $A_3 > B_3$ then $GT = 1$, irrespective of the lower bits of A and B

   Define: $G_3 = A_3 B_3'$ ($A_3 == 1$ and $B_3 == 0$)

2. If $A_3 == B_3$ ($E_3 == 1$), we compare A_2 with B_2

   Define: $G_2 = A_2 B_2'$ ($A_2 == 1$ and $B_2 == 0$)

3. If $A_3 == B_3$ and $A_2 == B_2$, we compare A_1 with B_1

   Define: $G_1 = A_1 B_1'$ ($A_1 == 1$ and $B_1 == 0$)

4. If $A_3 == B_3$ and $A_2 == B_2$ and $A_1 == B_1$, we compare A_0 with B_0

   Define: $G_0 = A_0 B_0'$ ($A_0 == 1$ and $B_0 == 0$)

Therefore, $GT = G_3 + E_3 G_2 + E_3 E_2 G_1 + E_3 E_2 E_1 G_0$

The Less Than Output

We can derive the expression for the LT output, similar to GT

Given the 4-bit input numbers: A and B

1. If $A_3 < B_3$ then $LT = 1$, irrespective of the lower bits of A and B

   Define: $L_3 = A_3' B_3$ $\quad (A_3 == 0 \text{ and } B_3 == 1)$

2. If $A_3 = B_3$ $(E_3 == 1)$, we compare A_2 with B_2

   Define: $L_2 = A_2' B_2$ $\quad (A_2 == 0 \text{ and } B_2 == 1)$

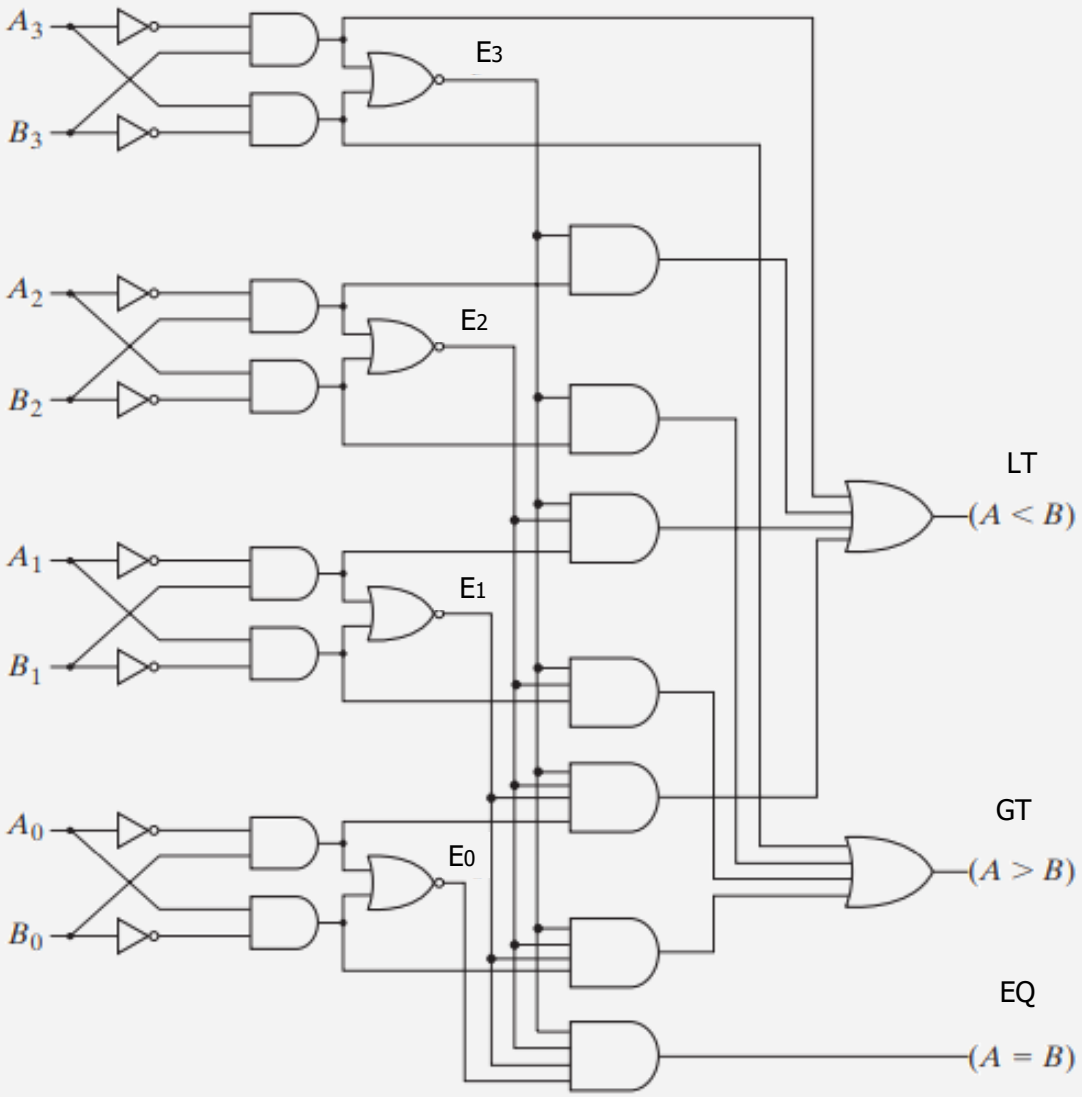
3. Define: $L_1 = A_1' B_1$ $\quad (A_1 == 0 \text{ and } B_1 == 1)$

4. Define: $L_0 = A_0' B_0$ $\quad (A_0 == 0 \text{ and } B_0 == 1)$

Therefore, $LT = L_3 + E_3 L_2 + E_3 E_2 L_1 + E_3 E_2 E_1 L_0$

Knowing GT and EQ, we can also derive $LT = (GT + EQ)'$

The Greater Than Output



Binary Multiplication

- Binary Multiplication is simple:

`0×0=0,  0×1=0,  1×0=0,  1×1=1`

Multiplicand $1100_2 = 12$

Multiplier $\times\ 1101_2 = 13$

```
      1100
     0000
    1100
   1100
--------
10011100
```

Binary multiplication

0 × multiplicand = 0

1 × multiplicand = multiplicand

Product $10011100_2 = 156$

- *n*-bit multiplicand × *n*-bit multiplier = 2*n*-bit product
- Accomplished via shifting and addition

2-bit Binary Multiplication

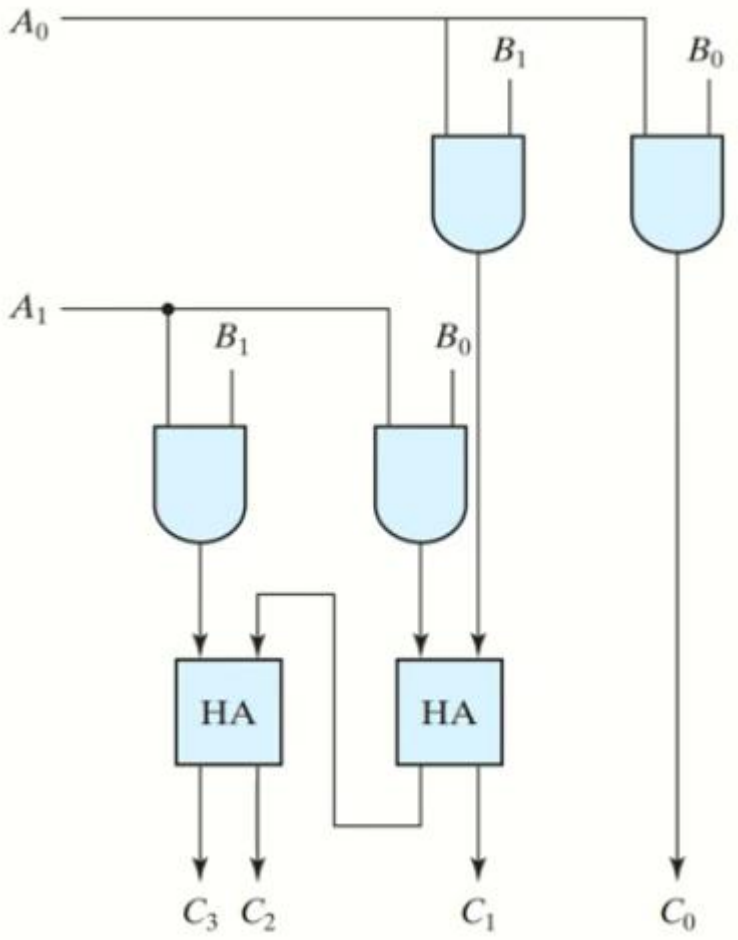
Multiplicand $B_1 \quad B_0$

Multiplier $A_1 \quad A_0$

$A_0B_1 \quad A_0B_0$

$A_1B_1 \quad A_1B_0$

$C_3 \quad C_2 \quad C_1 \quad C_0$



razeghizade@gmail.com

Razeghizade.pudica.ir

CREADITS: This presentation was created by M.Razeghizadeh
Please keep this slide for attribution