OF THE
DEPARTMENT OF
INFORMATION ENGINEERING

UNIVERSITÀ
DEGLI STUDI
DI PADOVA

Digital Systems

Binary Adders and Subtractors

Marta Bagatin, marta.bagatin@unipd.it

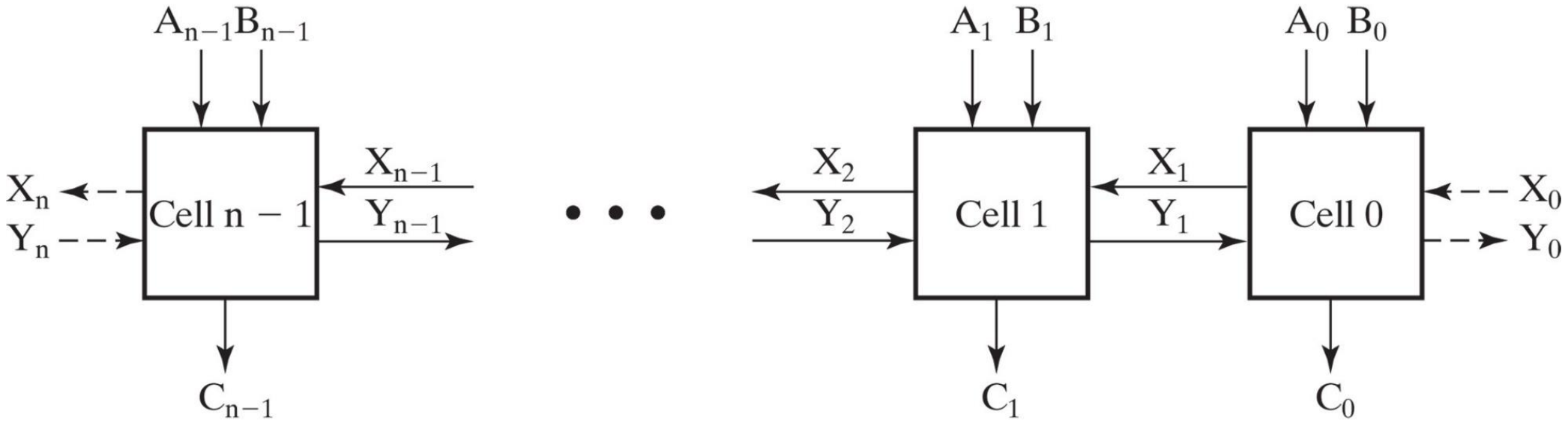
Degree Course in Information Engineering

Academic Year 2023-2024

Purpose of the Lesson

- Study the circuits for n-bit binary numbers addition and subtraction
  - Half adder
  - Full adder
  - Ripple carry adder
  - Subtraction of unsigned numbers
  - Representation of signed numbers
  - Adding and subtracting signed numbers
  - Overflow
  - VHDL representations of binary adder

Iterative Combinational Circuits



- They are an example of a **hierarchical structure** and are useful for functions operating on multiple bits
- The same subfunction is applied to each bit position. The circuit is created by combining several (identical or similar) blocks, which **iteratively pass the values to the adjacent subblock**. These intermediate values are only used internally and they are not present at the circuit output

Binary Arithmetic Circuits

- A binary arithmetic circuit is a **combinational circuit that performs arithmetic operations** (addition, subtraction, ...) on binary numbers
- As we will see, it is very convenient to use **iterative hierarchical designs** to realize arithmetic circuits

Binary Adder

Sum of two Binary Digits

X	Y	X + Y
0	0	0
0	1	1
1	0	1
1	1	10

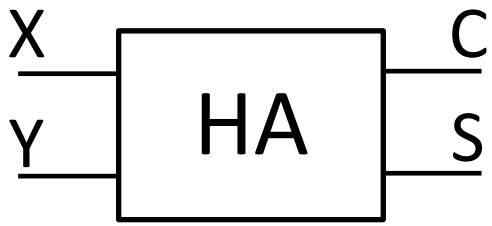
If both the augend and addend are equal to '1', we need 2 bits for the result!

Half Adder (HA)

Combinational circuit that performs the **sum between 2 bits**

Inputs		Outputs	
X	Y	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Truth table of the half adder



X: augend

Y: addend

C: carry

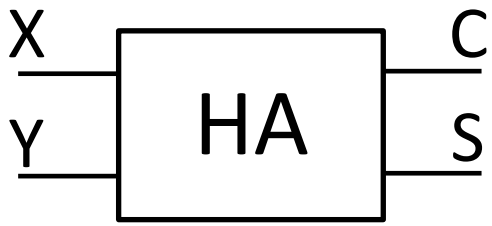
S: LSB of the sum

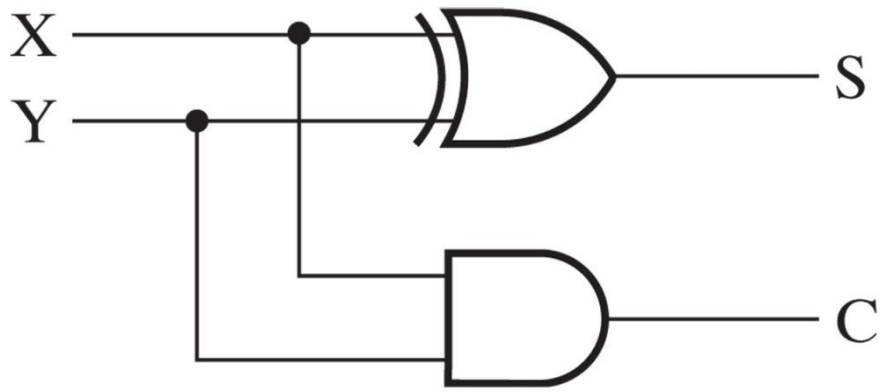
Half Adder (HA)

Inputs		Outputs	
X	Y	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

$$S = X \oplus Y$$

$$C = X\,Y$$





X: augend

Y: addend

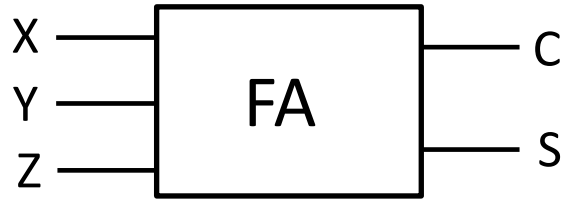
C: carry

S: LSB of the sum

Full Adder (FA)

Combinational circuit that performs the **sum of 3 input bits**: 2 bits to be added and 1 previous carry bit

Inputs			Outputs	
X	Y	Z	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

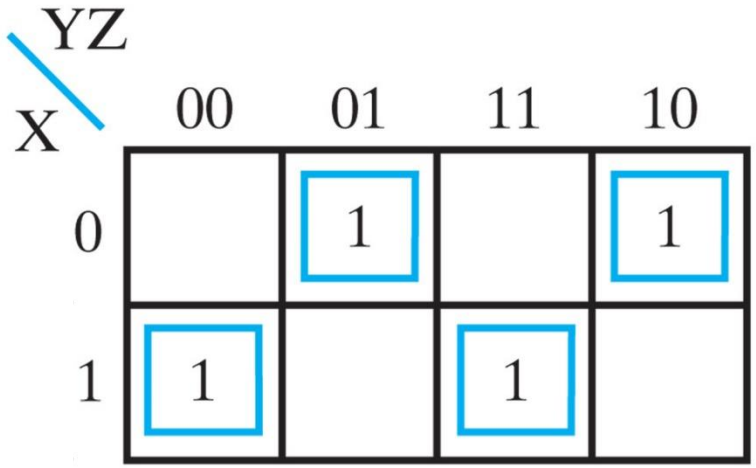


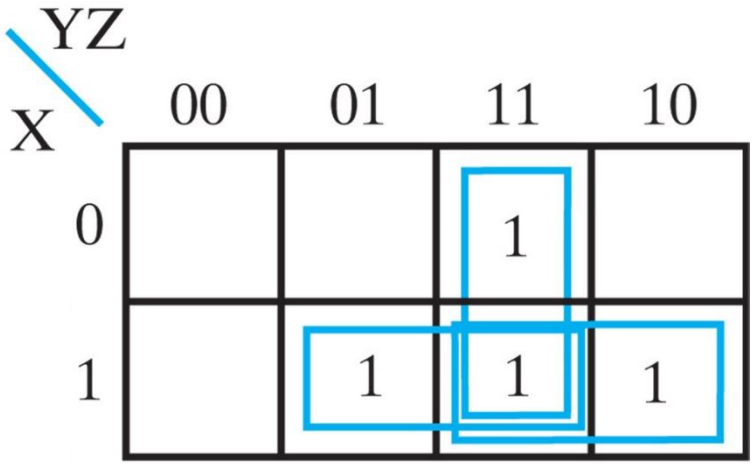
X: augend
Y: addend
Z: previous carry
S: LSB of the sum
C: carry

$$S(X, Y, Z) = \sum m(1, 2, 4, 7)$$

$$C(X, Y, Z) = \sum m(3, 5, 6, 7)$$

Full Adder (FA)





$$S = \bar{X}\,\bar{Y}\,Z + \bar{X}\,Y\,\bar{Z} + X\,\bar{Y}\,\bar{Z} + X\,Y\,Z$$
$$= X \oplus Y \oplus Z$$

XOR of 3 variables (odd function: equal to '1' if the number of '1' inputs is odd)

$$C = XY + XZ + YZ$$
$$= XY + Z(X + Y)$$
$$= XY + Z(X \oplus Y + XY)$$
$$= XY + ZXY + Z\,(X \oplus Y)$$
$$= XY + Z\,(X \oplus Y)$$

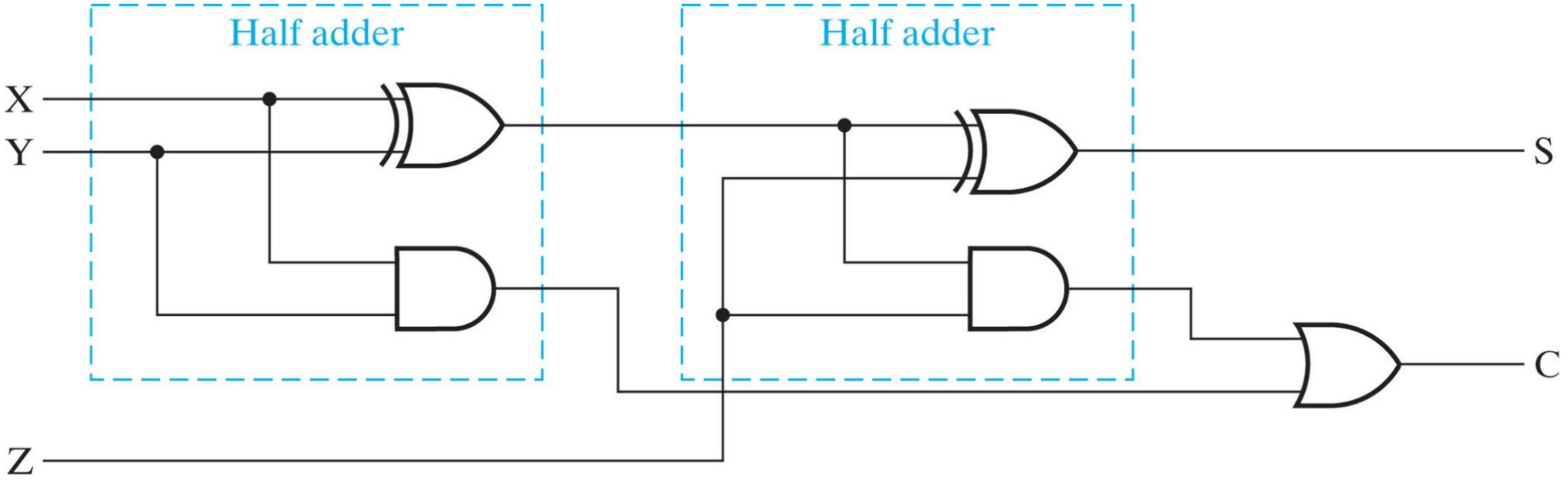
We express C as to include a XOR gate

Full Adder (FA)

The full adder can be implemented with **two half adders and a 2-input OR gate**:

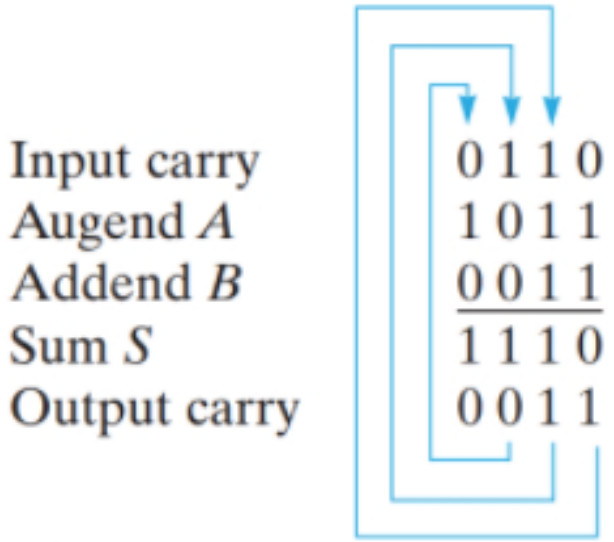
$$S = (X \oplus Y) \oplus Z$$

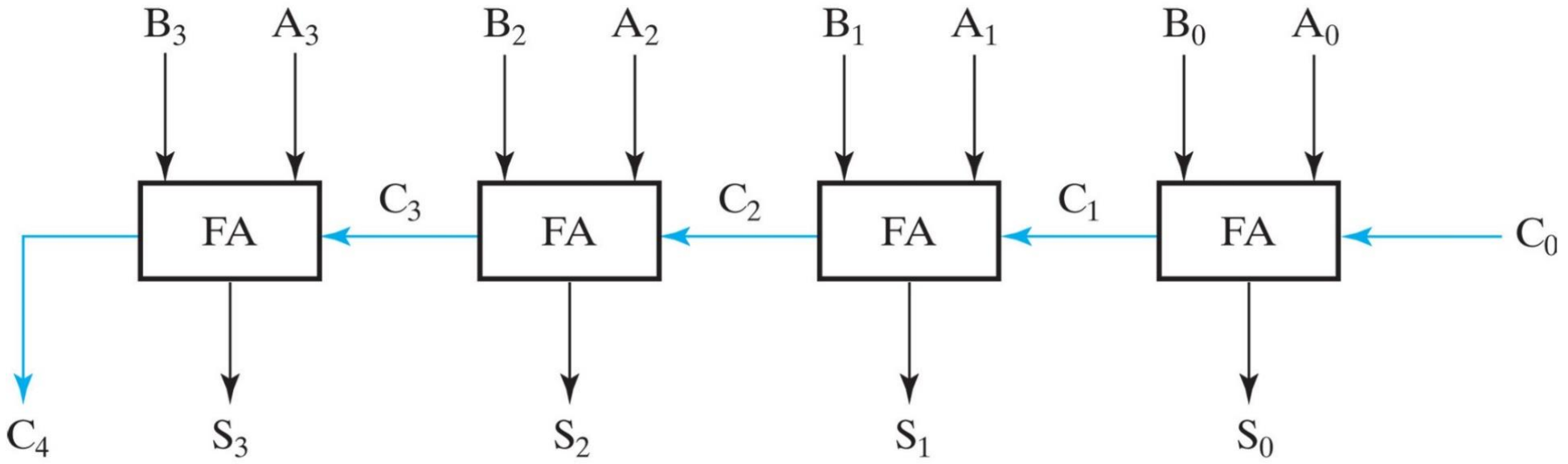
$$C = XY + Z\,(X \oplus Y)$$



Ripple Carry Adder

- The **sum of two n-bit binary numbers** is performed bitwise, with **n full adders connected (in chain) one after the other**
  - The output carry of each FA is the input carry of the next FA, starting from the LSB and moving to the MSB (ripple = propagate)
  - This simulates the procedure we use when we perform addition in column





Example of iterative circuit: the system is built reusing simple blocks (truth table has $2^9 = 512$ lines!)

Binary Subtraction

Unsigned Numbers

- So far we have considered unsigned numbers: the binary representation does not include any information about the sign of the number

Binary Subtraction with M > N

- In a previous lecture, we saw the subtraction of two n-bit binary numbers. When the minuend (M) is larger than the subtrahend (N), the subtraction returns a **correct** value and there is **no borrow in the most significant position**

  – **Example with M ≥ N**

Borrow:	0011		-> No borrow in the most significant position
Minuend:	11110	30_{10}	
Subtrahend:	10011	19_{10}	
Difference:	01011	11_{10}	-> Difference is positive and correct!

Binary Subtraction with M < N

- On the contrary, if the minuend (M) is smaller than the subtrahend (N), the procedure returns an incorrect value

  – **Example with M < N**

Borrow:	11100	
Minuend:	10011	19_{10}
Subtrahend:	11110	30_{10}
Difference: (not correct)	10101	21_{10}
(correct)	- 01011	-11_{10}

-> Borrow in the most significant position is 1 !!

We mistakenly added 2^n to the result (borrow in the most significant position)

-> Absolute value of the difference is not correct: we get M-N+2^n instead of the correct value M-N

Binary Subtraction

- There are two alternative ways to perform the subtraction of **unsigned numbers** correctly both with M ≥ N and with M <N

  a) If M < N, exchange M with N and invert the sign of the result: inefficient and costly circuitry!

  b) If M < N, don't swap M with N, subtract 2^n to the 'incorrect' result and add a minus

Binary Subtraction

- Method b) to subtract a number N (n bits) from M (n bits)

  1) Perform M-N subtraction

  2) If the borrow in the most significant position is 0, the result is positive and correct ($M \geq N$)

  3) If the borrow is 1 ($M < N$), the result is negative and its absolute value is 2^n minus the result obtained: as we will see shortly, this is called the 2s complement of the result obtained ($M - N + 2^n$)

  Example: perform the subtraction 01100100 - 10010110 (n = 8)

Binary Subtraction

- Method b) to subtract a number N (n bits) from M (n bits)

  1) Perform M-N subtraction

  2) If the borrow in the most significant position is 0, the result is positive and correct ($M \geq N$)

  3) If the borrow is 1 ($M < N$), the result is negative and its absolute value is 2^n minus the result obtained: as we will see shortly, this is called the 2s complement of the result obtained ($M - N + 2^n$)

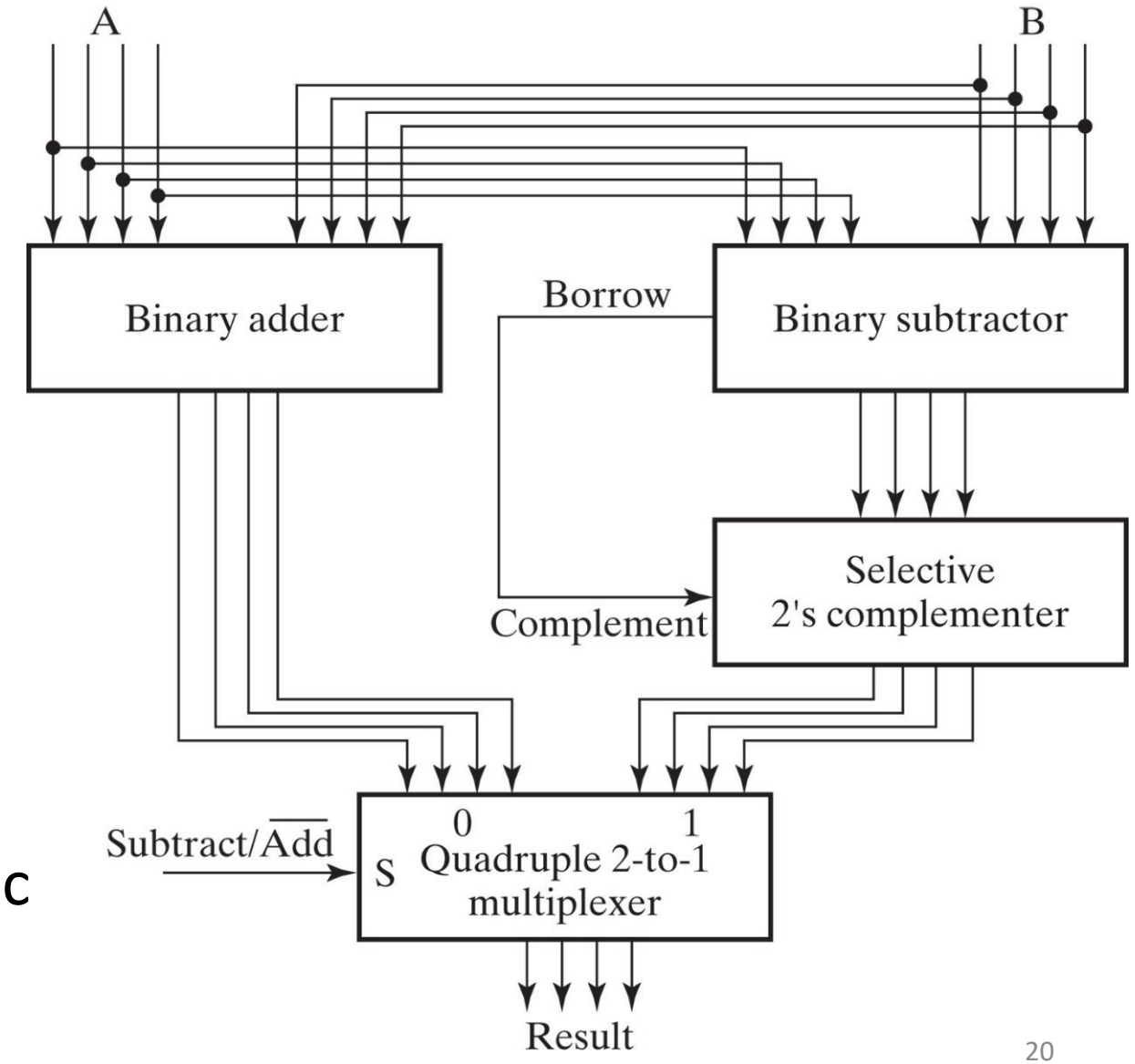
Example: perform the subtraction 01100100 - 10010110 (n = 8)

Borrow:	10011110		-> Borrow in the most significant position is 1 (M < N)
Minuend:	01100100	100_{10}	
Subtrahend:	10010110	150_{10}	
Difference: (not correct)	11001110	206_{10}	-> the absolute value of the correct difference is 2^n minus the obtained result : 256–206 = 50
(correct)	-0110010	-50_{10}	

Binary Adder-Subtractor

Circuit diagram that realizes the addition and subtraction with method b) requires

- A binary subtractor
- A binary adder
- A circuit that performs the 2s complement (in a selective way: only if the borrow in the most significant position is 1)
- a mux that selects the operation to be performed



This circuit is complicated! →
Let's try to simplify it, sharing the logic between the adder and subtractor….

Complements

- Given a **binary number N** to **n bits**, we define two types of complements
  - **1s complement**: $(2^n - 1) - N$ → bitwise complement of N, i.e. each bit is complemented
  - **2s complement**: $(2^n - 1) - N + 1$ → (1s complement) + 1

    $= 2^n - N$

N	1s complement	2s complement
1011001	0100110	0100111
0001111	1110000	1110001
101100	010011	010100

Note: the complement of the complement of N restores the number to its original value

Binary Adder-Subtractor of Signed Numbers

Representation of Signed Numbers

- We will see two types of representations of signed numbers
  - Signed-magnitude representation
  - 2s complement representation

Signed-Magnitude Representation

- The binary number is preceded by the sign bit : **'0' for positive (or null) numbers, '1' for negative numbers**
  - A string of n bits can represent different numbers, depending on whether it is considered as an unsigned or signed number (signed-magnitude representation)

N binary	Unsigned representation	Sign-magnitude representation
01001	9	+9
11001	25	-9

  - In arithmetic operations between numbers in signed-magnitude representation, the magnitude bits are processed separately (treated as unsigned numbers), so the subtraction requires a correction of the sign

2s Complement Representation

- The 2s complement representation is adopted to avoid the need of correcting the sign for subtraction
  - **Positive** numbers are indicated **by the sign bit '0' (MSB) followed by the binary representation** of the number
  - **Negative** numbers are indicated with the **2s complement representation of the corresponding positive number** (2^n - N)

  $\Rightarrow$ All positive numbers have '0' in the MSB position, all negative numbers have '1' as MSB!

Example: 8-bit binary representation (n = 8)

N decimal	Sign-magnitude representation	2s complement representation
+9	00001001	00001001
-9	10001001	11110111

Signed Numbers

- Positive numbers have the same representation in 2s complement and signed-magnitude notation
- In both representations, the MSB is always '0' for positive numbers, always '1' for negative numbers
- Signed-magnitude representation: 7 positive numbers, 7 negative numbers, 2 zeros (positive 0 and negative 0)
- 2s complement representation: 7 positive numbers, 8 negative numbers, 1 zero (positive)
- Signed-magnitude is used in ordinary arithmetics, 2s complement is used in computers, allowing simpler implementations for adder/subtractor

Signed Binary Numbers

Decimal	Signed 2s Complement	Signed Magnitude
+ 7	0111	0111
+ 6	0110	0110
+ 5	0101	0101
+ 4	0100	0100
+ 3	0011	0011
+ 2	0010	0010
+ 1	0001	0001
+ 0	0000	0000
− 0	—	1000
− 1	1111	1001
− 2	1110	1010
− 3	1101	1011
− 4	1100	1100
− 5	1011	1101
− 6	1010	1110
− 7	1001	1111
− 8	1000	—

Addition of Signed Numbers

- The sum of signed binary numbers in 2s complement representation is done **adding the numbers (including the sign bit)** and **discarding any carry-out of the sign bit position**
- We obtain the **result represented in 2s complement**
- Examples:

00000110	$+6_{10}$	11111010	-6_{10}
00001101	$+13_{10}$	00001101	$+13_{10}$
00010011	$+19_{10}$	100000111	$+7_{10}$
00000110	$+6_{10}$	11111010	-6_{10}
11110011	-13_{10}	11110011	-13_{10}
11111001	-7_{10}	111101101	-19_{10}

Subtraction of Signed Numbers

- The subtraction of signed binary numbers in 2s complement representation is done **adding (including the sign bit) the 2s complement of the subtrahend to the minuend, and discarding any carry-out of the sign bit position**
- Equivalent to the algebraic sum:

$$(\pm A) - (+B) = (\pm A) + (-B)$$
$$(\pm A) - (-B) = (\pm A) + (+B)$$

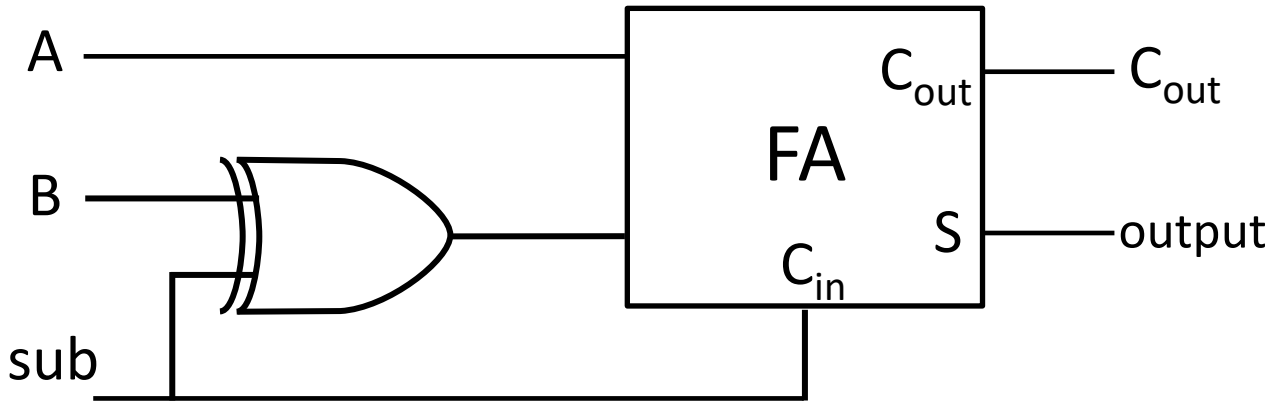
- This allows a computer to have a common hardware for addition and subtraction
- The result is obtained in 2s complement representation
- Examples:

11111010	-6_{10}		00000110	$+6_{10}$	
00001101	$-(-13)_{10}$		00001101	$-(-13)_{10}$	
100000111	$+7_{10}$		00010011	$+19_{10}$	

Binary Adder-Subtractor

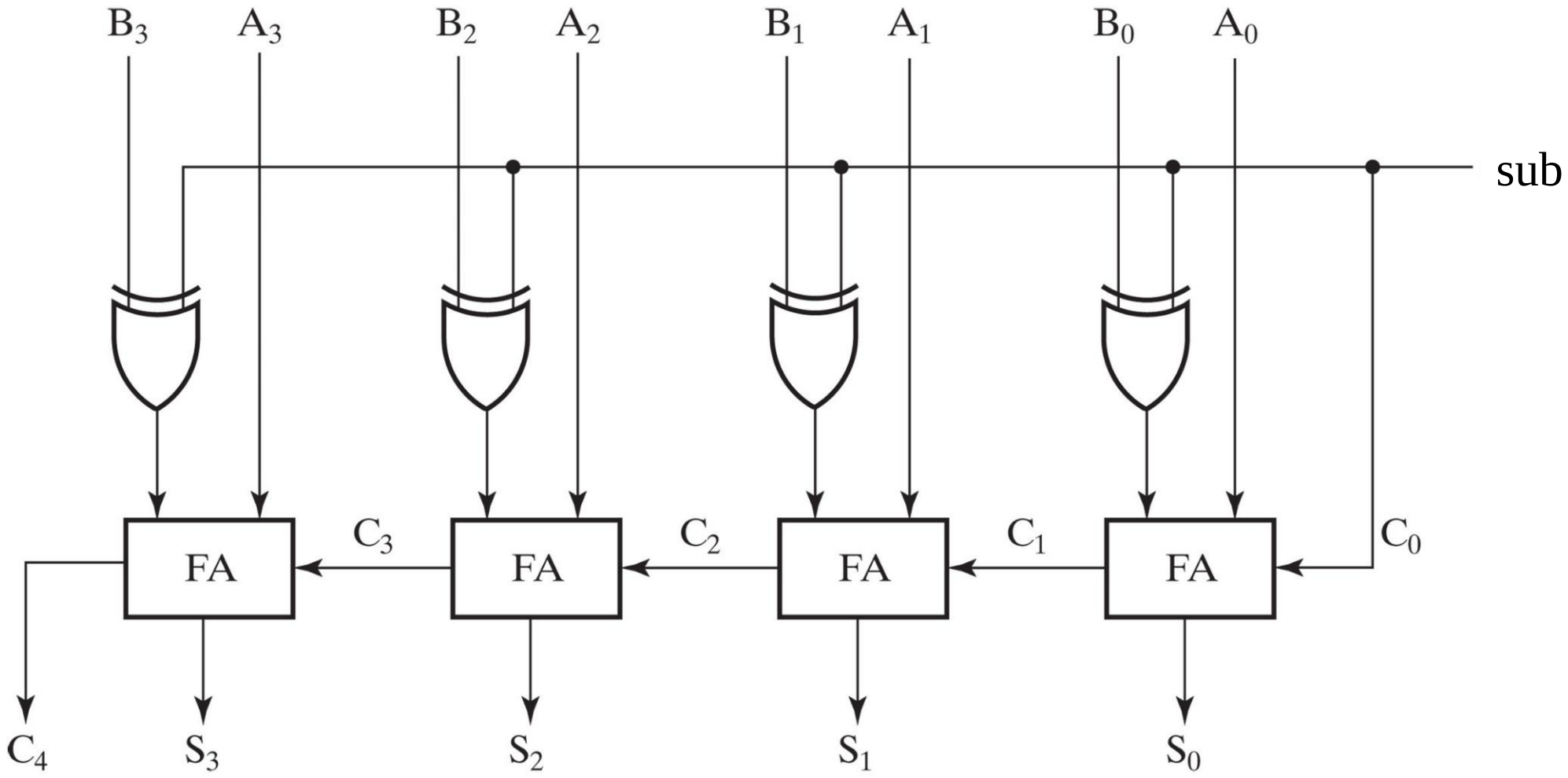
- Thanks to the 2s complement, we can combine addition and subtraction in a common adder (Full Adder, FA) and we introduce a XOR gate, with the purpose to selectively supply the 1s complement of addend B (to which 1 is then added), in case of subtraction

Inputs: A, B, sub
Outputs: output, C_{out}



- **if sub = 0, the circuit performs the sum**: the XOR gate has B as an output and the FA executes A + B (with 0 carry input)
- **if sub = 1 , the circuit performs the subtraction**: the XOR gate generates the complement of B, and the FA calculates A + NOT (B) + 1 (sum between A and the 2s complement of B)

Binary Adder-Subtractor: n = 4



Overflow

The Overflow Issue

- Overflow happens when the **result does not have a sufficient number of bits to represent the value** of the sum/difference
  - Example: we add two 4-bit numbers and we get a 5-bit result
- Overflow is a problem in a computer, where the number of bits allocated for the representation of numbers is fixed and a result exceeding the number of bits cannot be accomodated
- **Computers must be able to detect the occurrence of an overflow** and to signal that the result is incorrect with the number of available bits

Overflow Detection

- Overflow detection differs, depending on whether we are considering unsigned or signed numbers
- **Unsigned numbers**
  - Addition: there is an overflow if there is an output carry in the most significant position
  - Subtraction: there cannot be overflow (the result is always smaller than or equal to the larger of the 2 original numbers)
- **Signed numbers**
  - For both addition and subtraction [A - B = A + (-B)]: there is no overflow if the numbers have a different sign (the absolute value of the result is always smaller than or equal to the larger). There can be overflow only if the two numbers have the same sign, both positive or both negative (and the result has an opposite sign)

Overflow Detection

Example: sum of signed numbers in 2s complement representation (n = 8 bits)

a)

Carries:	01
+ 70	01000110
+ 80	01010000
+ 150	10010110

b)

Carries:	10
- 70	10111010
- 80	10110000
- 150	01101010

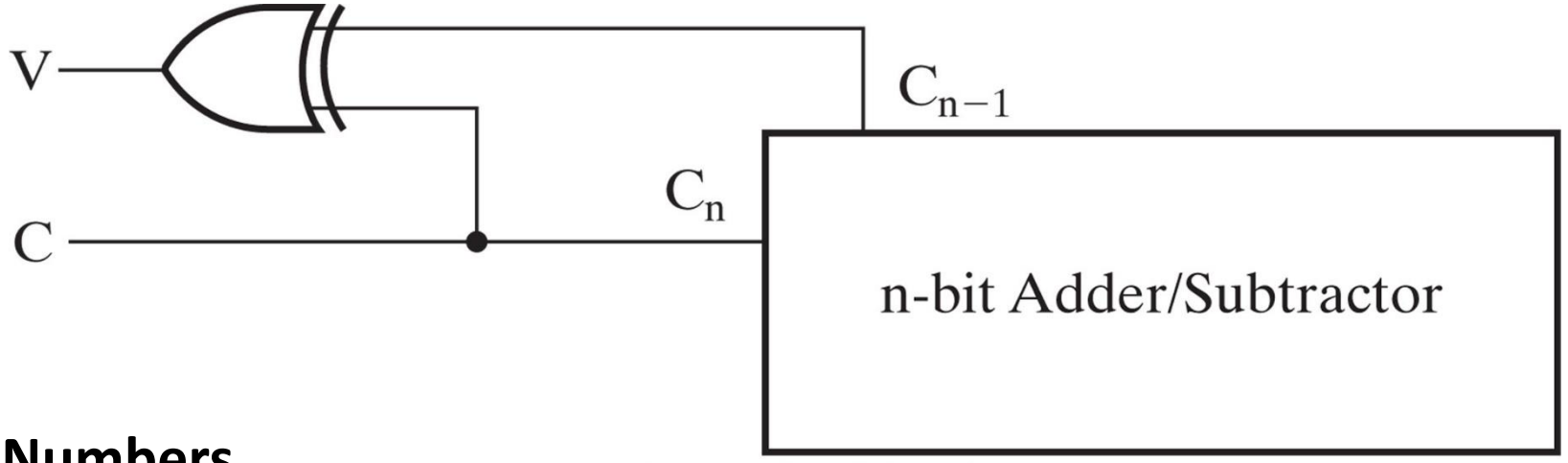
a) The result that should be positive is negative (sign bit is 1 due to the carry)

b) The result that should be negative is positive

=> The result would be correct if the carry out of the sign bit position was taken as the sign bit of the result

- There is **overflow** **if the carry bit in the sign position and the carry out bit are different**: if these two bits are applied to an XOR gate, an overflow can be detected!

Circuit for Overflow Detection

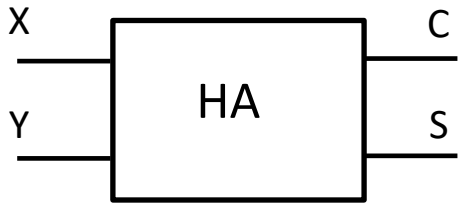


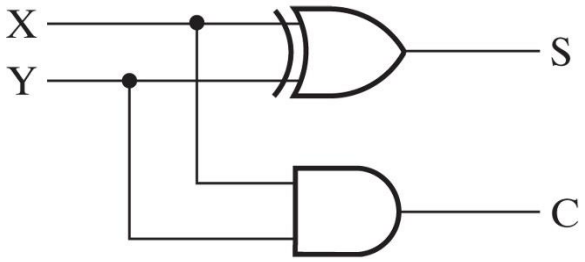
- **Unsigned Numbers**
  - C = 0 indicates that there is no overflow in the sum (output carry is null), while correction is needed in the sign for subtraction
  - C = 1 indicates that there is overflow in the sum (carry out in the last bit) and that the sign is correct for the subtraction
- **Signed Numbers**
  - V = 0 indicates that the result is correct
  - V = 1 indicates overflow (occurs when the carry bit in the sign position and the carry out of the sign bit position are different)

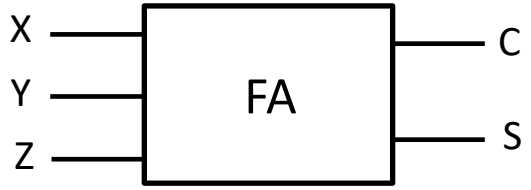
VHDL Representations of Adders

4-bit Adder: Hierarchical VHDL (1/3)

```
-- 4-bit Adder: Hierarchical Dataflow/Structural
-- (See Figures 3-42 and 3-43 for logic diagrams)
library ieee;
use ieee.std_logic_1164.all;
entity half_adder is
   port (x, y : in std_logic;
         s, c : out std_logic);
end half_adder;

architecture dataflow_3 of half_adder is
   begin
      s <= x xor y;
      c <= x and y;
end dataflow_3;

library ieee;
use ieee.std_logic_1164.all;
entity full_adder is
   port (x, y, z : in std_logic;
         s, c : out std_logic);
end full_adder;
```

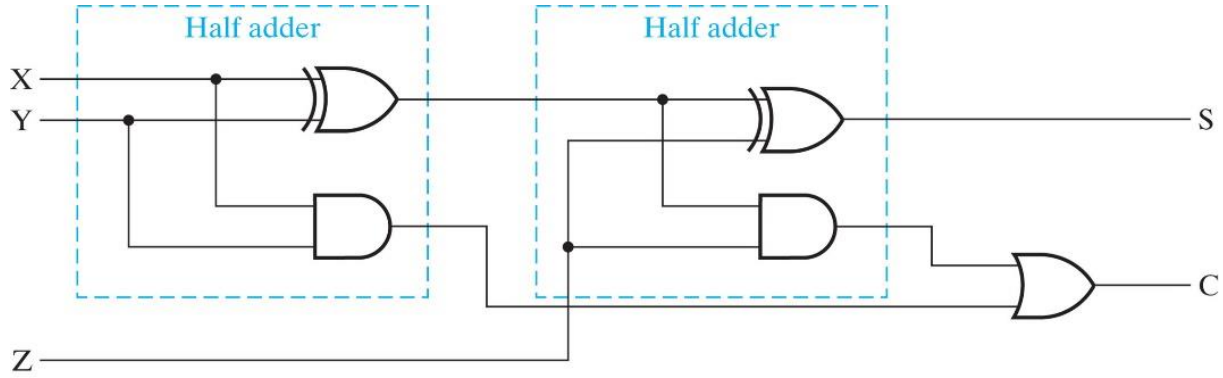


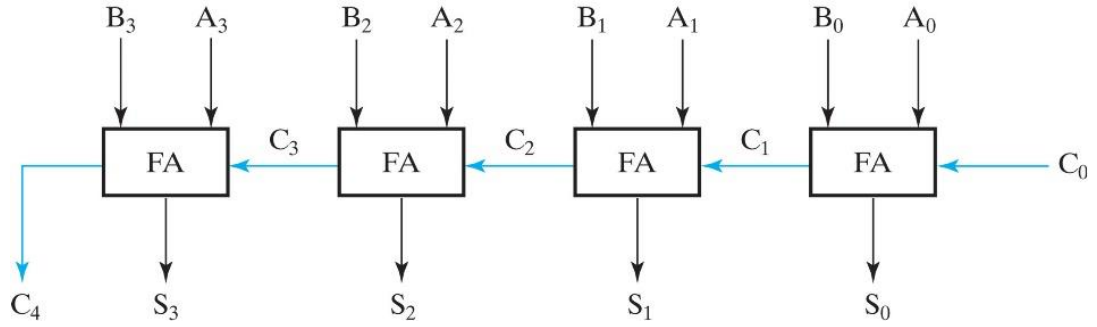




4-bit Adder: Hierarchical VHDL (2/3)

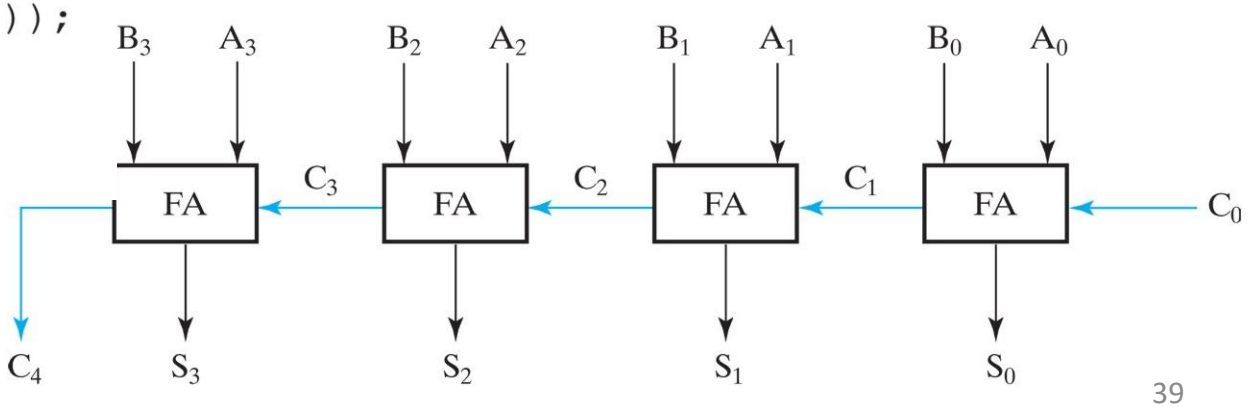
```
architecture struc_dataflow_3 of full_adder is
   component half_adder
      port (x, y : in std_logic;
            s, c : out std_logic);
   end component;
   signal hs, hc, tc: std_logic;
   begin
      HA1: half_adder
        port map  (x, y, hs, hc);
      HA2: half_adder
        port map (hs, z, s, tc);
      c <= tc or hc;
end struc_dataflow_3;

library ieee;
use ieee.std_logic_1164.all;
entity adder_4 is
   port(B, A : in std_logic_vector(3 downto 0);
        C0 : in std_logic;
        S : out std_logic_vector(3 downto 0);
        C4: out std_logic);
end adder_4;
```





4-bit Adder: Hierarchical VHDL (3/3)

```
architecture structural_4 of adder_4 is
   component full_adder
      port(x, y, z : in std_logic;
           s, c: out std_logic);
   end component;
   signal C: std_logic_vector (4 downto 0);
   begin
      Bit0: full_adder
       port map (B(0), A(0), C(0), S(0), C(1));
      Bit1: full_adder
       port map (B(1), A(1), C(1), S(1), C(2));
      Bit2: full_adder
       port map (B(2), A(2), C(2), S(2), C(3));
      Bit3: full_adder
       port map (B(3), A(3), C(3), S(3), C(4));
      C(0) <= C0;
      C4 <= C(4);
end structural 4;
```



4-bit Adder: Behavioral VHDL

```
-- 4-bit Adder: Behavioral Description
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity adder_4_b is
   port(B, A : in std_logic_vector(3 downto 0);
        C0 : in std_logic;
        S : out std_logic_vector(3 downto 0);
        C4: out std_logic);
end adder_4_b;

architecture behavioral of adder_4_b is
signal sum: std_logic_vector (4 downto 0);
begin
   sum <= ('0' & A) + ('0' & B) + ("0000" & C0);
   C4 <= sum(4);
   S <= sum(3 downto 0);
end behavioral;
```

4-bit Adder: Behavioral VHDL

```
-- 4-bit Adder: Behavioral Description
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity adder_4_b is
   port(B, A : in std_logic_vector(3 downto 0);
        C0 : in std_logic;
        S : out std_logic_vector(3 downto 0);
        C4: out std_logic);
end adder_4_b;

architecture behavioral of adder_4_b is
signal sum: std_logic_vector (4 downto 0);
begin
   sum <= ('0' & A) + ('0' & B) + ("0000" & C0);
   C4 <= sum(4);
   S <= sum(3 downto 0);
end behavioral;
```

Note: Addition cannot be performed on std_logic type, we need an additional package (std_logic_unsigned)

The circuit is not described in detail. Only its behavior is defined and the choice of how to implement the hardware is left to the simulator/ synthesizer

The adder is described as the sum of 3 binary numbers of 5 bits, obtained by concatenating the 4-bit augend and addend with an initial ‘0’, while the carry-in bit is concatenated with 4 initial ‘0s': the carry-out is given by the MSB, the final sum is given by the remaining 4 least significant bits

Disclaimer

Figures from*Logic and Computer Design Fundamentals*, Fifth Edition, GE Mano |Kime| Martin