

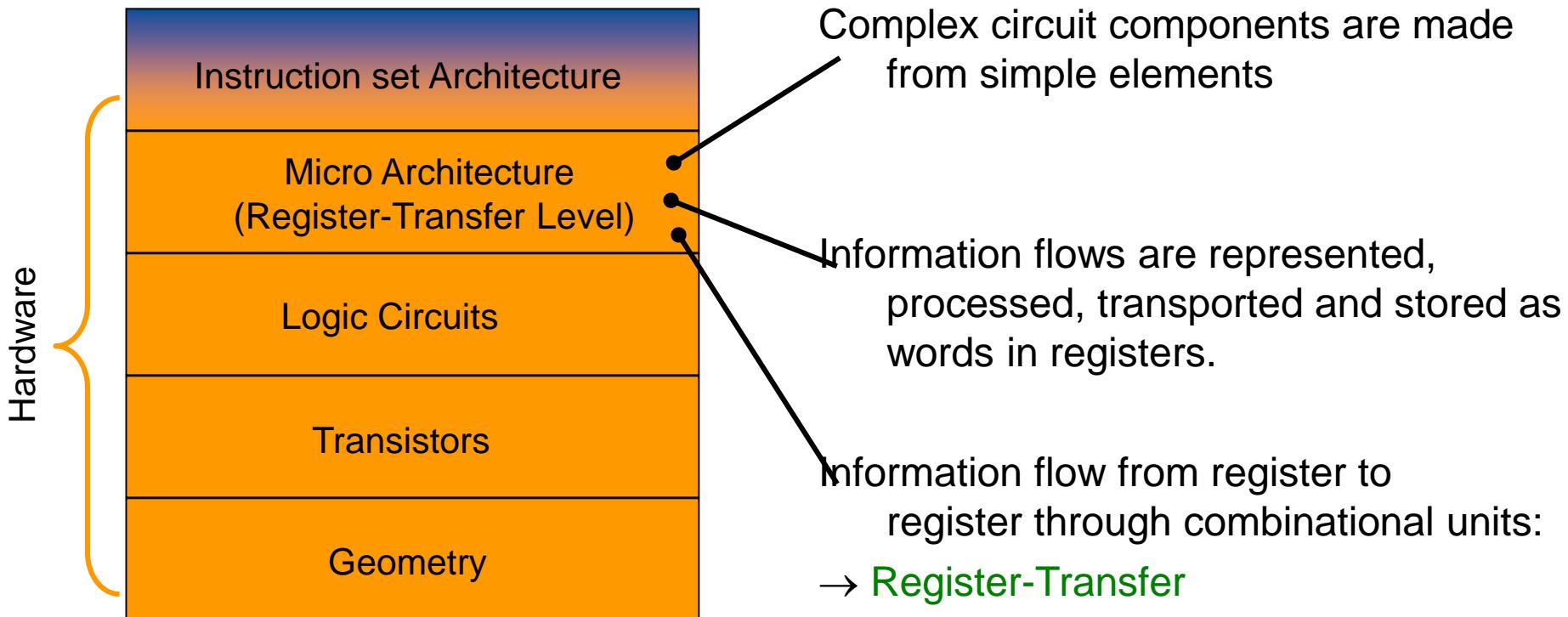
Department of Electrical & Computer Engineering

# Digital Logic And Computing Systems

## Chapter 06 – RTL Components

Dr. Christophe Bobda

# Register-Transfer Level (RTL)



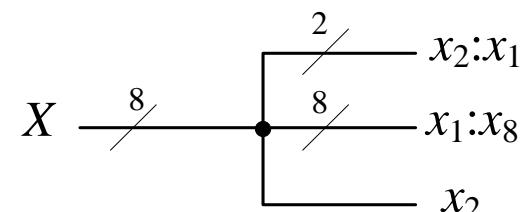
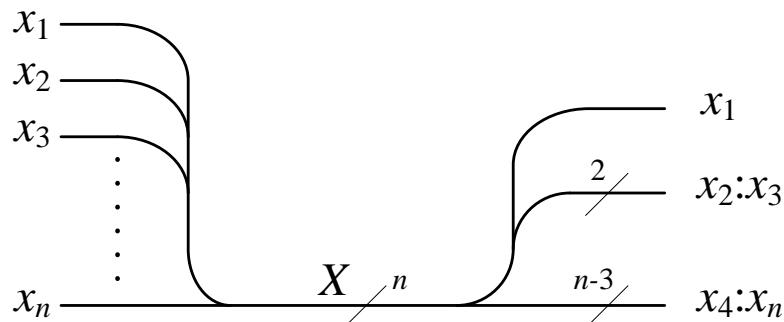
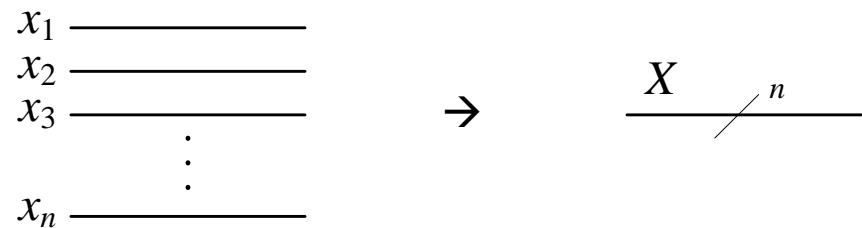
# Agenda

- ❑ Buses
- ❑ Word (bundle)gates
- ❑ Multiplexer / Demultiplexer
- ❑ Encoder / Decoder
- ❑ Memory
- ❑ Arithmetic Circuits

# Bus

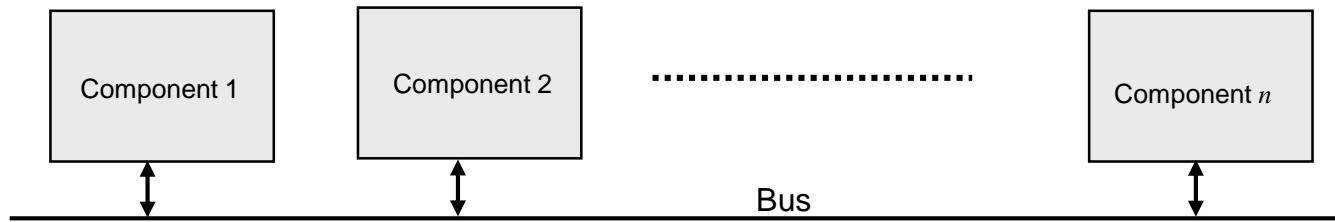
- Group of  $n \geq 1$  signals
  - $n$  is the width of the bus in bit

Graphical Representation:



# Buses

- At RT-Level, buses are used to connect system components.

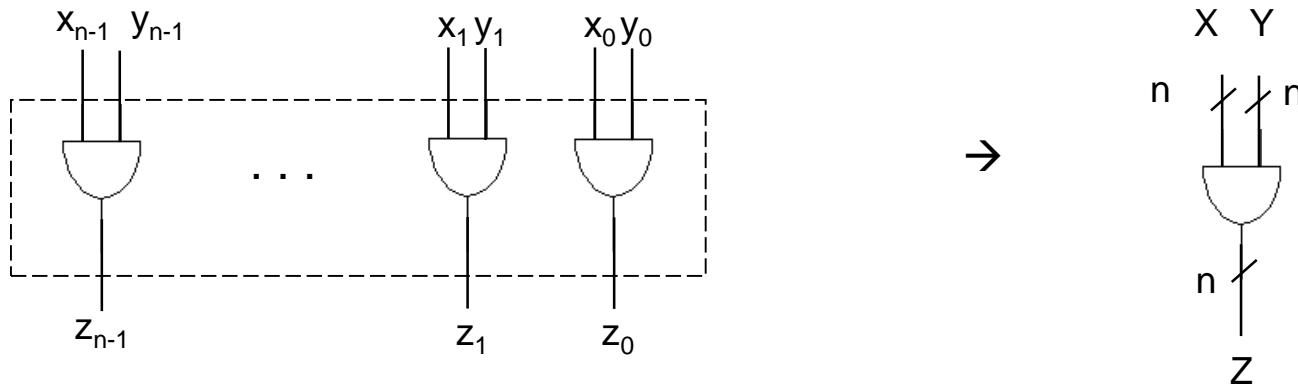


- Advantages compared to point-to-point:
  - Low hardware overhead: all components used the same bus.
  - scalability: easy to add new components without the need to redesign.
  - Easy "broadcasting": one component writes, all others read.
- Drawbacks compared to point-to-point:
  - Sequential communication.
  - Single point of failure.
- Technology must allow multiple outputs on single line ("Tri-State").

# N-bit (Word) gate

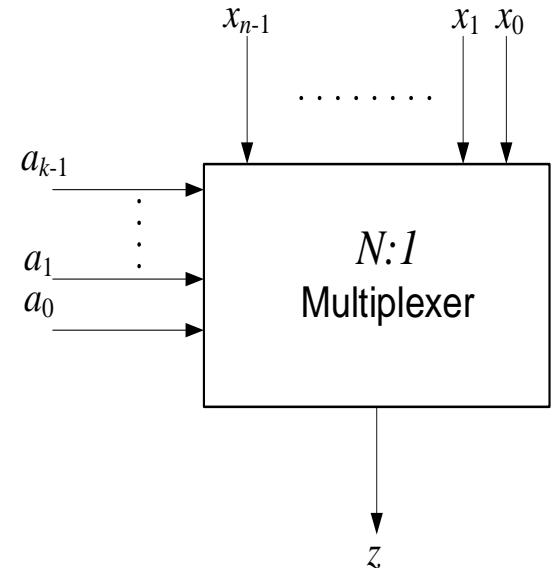
- A n-bit (word) gate is a grouping of  $n \geq 1$  independent gates
  - Inputs and outputs are  $n$ -bit buses

Graphical representation:



# Multiplexer

- A Multiplexer ( $n : 1$  MUX) connects one of  $n$  inputs to a single output at a time.
  - Also known as data selector
    - $n$  data inputs  $x_{n-1}, \dots, x_0$ , 1 data output  $z$
    - $k$  select input (address inputs)  $a_{k-1}, \dots, a_0$ ,  $n \leq 2^k$  (usually:  $n = 2^k$ )
  - Logic function: 
$$z = \sum_{j=0}^{n-1} m_j(a_{k-1}, \dots, a_0) \cdot x_j$$
    - $m_j = m_j(a_{k-1}, \dots, a_0)$  is the  $j$ -th minterm of  $k$  variables.
    - $a_{k-1}, \dots, a_0$  "addresses" the input data:  
 $\text{address} = i \rightarrow z = x_i$ .

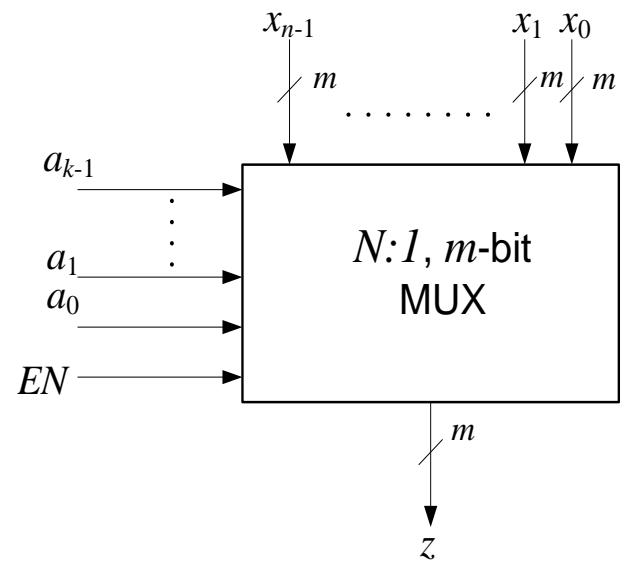


# Multiplexer

## □ Extension

- $m$ -bit input and output  $\rightarrow m \text{ times } n:1 \text{ MUX.}$
- Enable-Input  $EN$ ;
  - $EN=0 \rightarrow z = 0$  regardless of the data and select input.

$$z_l = EN \cdot \sum_{j=0}^{n-1} m_j \cdot x_{j,l} \quad \forall l : 0 \dots m-1$$



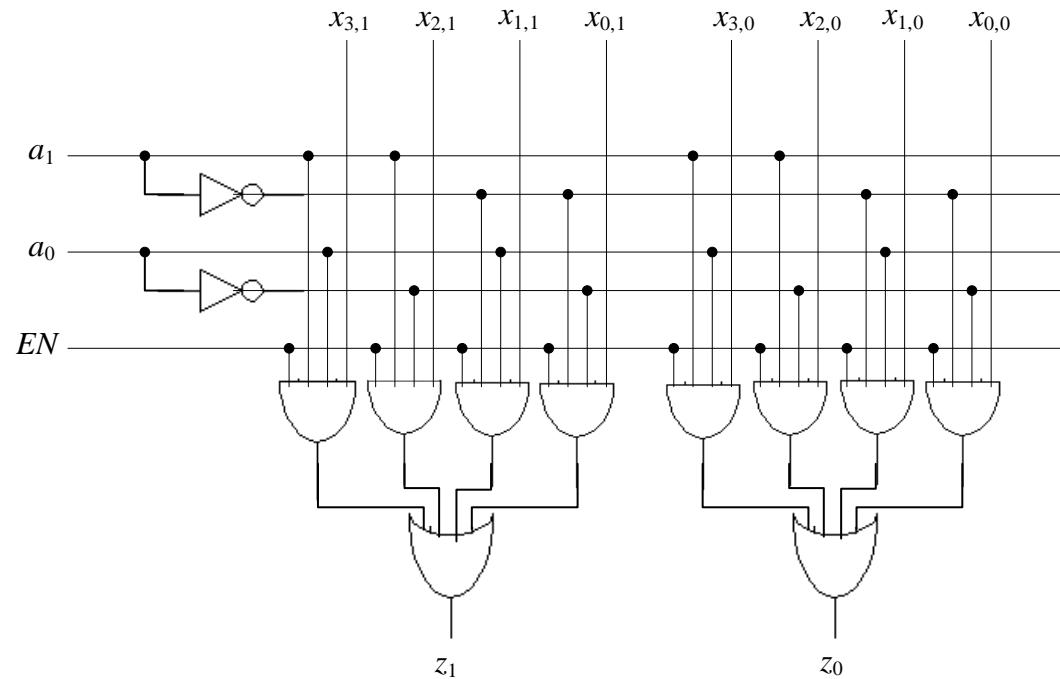
# Multiplexer

## □ Example:

- 4:1, 2-bit MUX with enable input.

$$z_1 = EN \cdot \sum_{j=0}^3 m_j \cdot x_{j,1} = EN \cdot (\bar{a}_1 \bar{a}_0 x_{0,1} + \bar{a}_1 a_0 x_{1,1} + a_1 \bar{a}_0 x_{2,1} + a_1 a_0 x_{3,1})$$

$z_0$  similar to  $z_1$

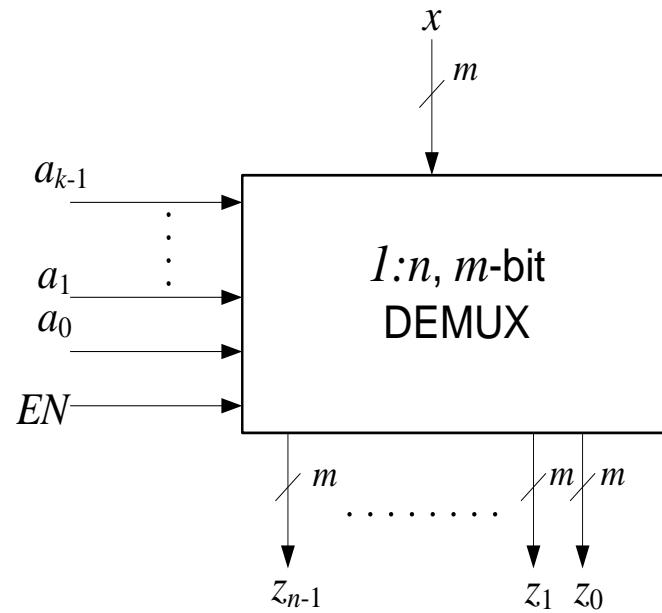


# Demultiplexer

- A demultiplexer ( $1:n$  DEMUX) connects 1 single input to one of  $n$  outputs at a time.
  - A DEMUX is the “Inverse” of a MUX.
  - Logic function and circuit.

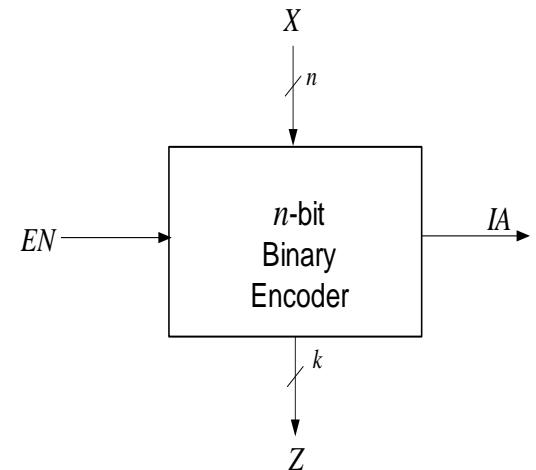
$$z_j = EN \bullet m_j(a_{k-1}, \dots, a_0) \bullet x$$

$$z_{j,l} = EN \bullet m_j(a_{k-1}, \dots, a_0) \bullet x_l$$



# Encoder

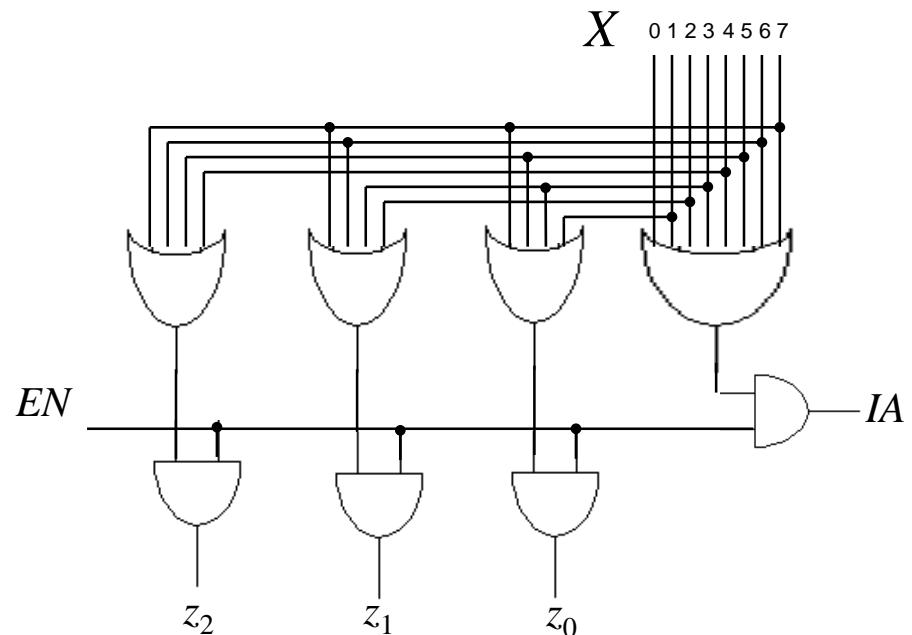
- A  $n$ -bit Binary-Encoder has ...
  - a  $n$ -bit word  $X$  input, of which maximum one bit  $x_i$  can be 1.
  - a  $k$ -bit Word  $Z$  output, which corresponds to index  $i$  binary coded;  
 $k \geq \text{ld}(n)$
  - Encoder have an additional output  $IA$  (Input Active), used to recognized cases where all input are 0.
    - If all  $x_i = 0 \rightarrow IA = 0$ , otherwise  $IA = 1$ .
  - $EN=0 \rightarrow Z=0..0$ , regardless of  $X$ .



# Encoder

- Example: 8-bit Binary-Encoder.

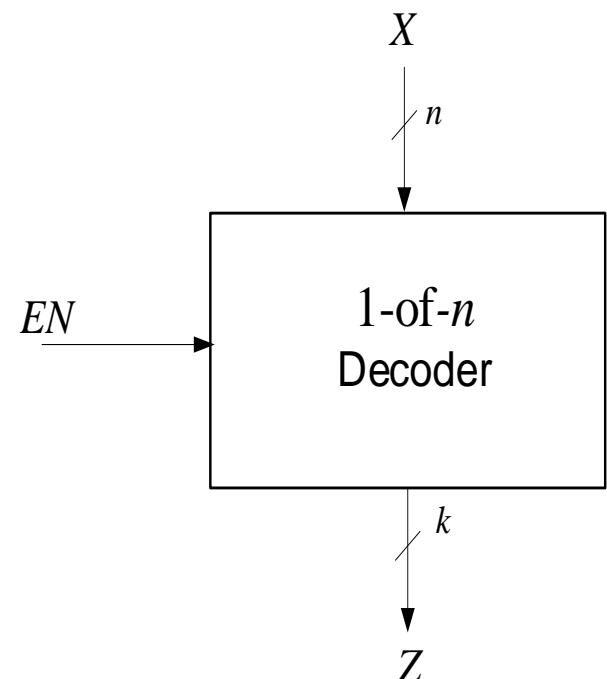
$X_0$	$X_1$	$X_2$	$X_3$	$X_4$	$X_5$	$X_6$	$X_7$	$z_2$	$z_1$	$z_0$	IA
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	1
0	1	0	0	0	0	0	0	0	0	1	1
0	0	1	0	0	0	0	0	0	1	0	1
0	0	0	1	0	0	0	0	0	1	1	1
0	0	0	0	1	0	0	0	1	0	0	1
0	0	0	0	0	0	1	0	0	1	0	1
0	0	0	0	0	0	1	0	1	0	1	1
0	0	0	0	0	0	1	1	1	1	1	1



- Priority-Encoder: many inputs  $X_i$  can be at 1 simultaneously.
  - The output is the binary code of the biggest Index  $i$  for  $X_i = 1$ ,
    - i.e inputs have “priorities”:  $x_{n-1}$  has the highest priority and  $x_0$  the lowest.

# Decoder

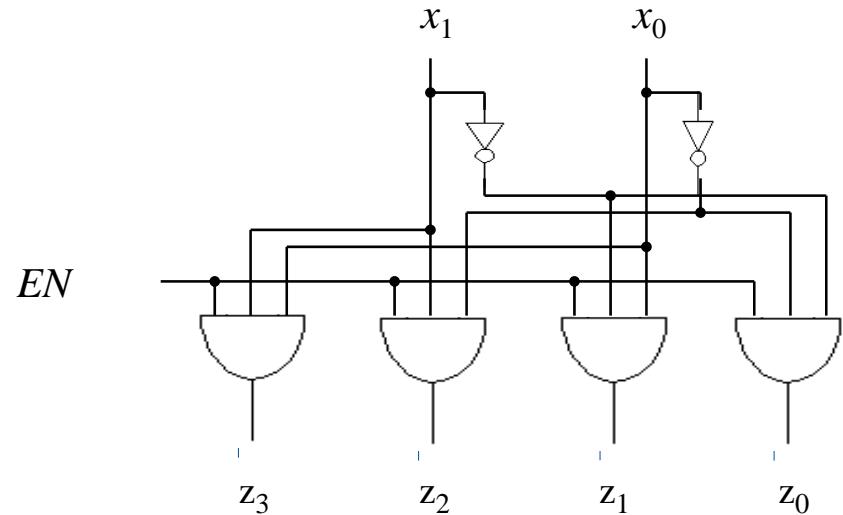
- A 1-of- $n$  decoder has ...
  - a binary-coded  $n$ -bit word  $X$  as input.
  - a  $k$ -bit word  $Z$ ,  $k = 2^n$ , as output,
    - $z_i = 1$ , if and only if  $X$  represents the number  $i$
  - $EN=0 \rightarrow Z=0..0$ .



# Decoder

- Example: 1-of-4 decoder.

$X_0$	$X_1$	$z_3$	$z_2$	$z_1$	$z_0$
0	0	0	0	1	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0



- In general: En/Decoder are used to change codes.
  - Conversion to binary → Binary encoder.
  - Conversion from binary → Binary decoder.

# Arithmetic Components

## □ Agenda

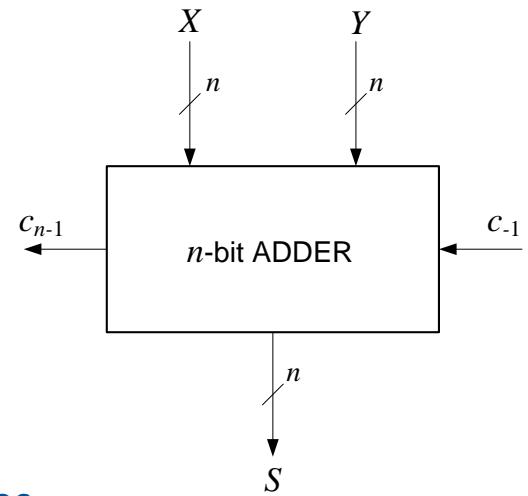
1. Fixpoint Adder/Subtractor
2. Fixpoint multiplier
3. Fixpoint divider

# Adder / Subtractor

- Fixpoint adders

- 2-level normal form (SOP)

- Vector function with  $n+1$  function of  $2n+1$  variables
  - truth table with  $2^{(2n+1)}$  entries
- Hardware overhead: exponential (!) with  $n$ 
  - Not feasible, even for very small  $n$
- Delay:  $2t_{pd}$  ( $t_{pd}$  ... Gate propagation time)
  - Fastest possible adder



# Full Adder

- *n*-bit Adder can be built as cascade of 1-bit adders
  - The basic element, a 1-bit adder, is called full adder

Truth Table

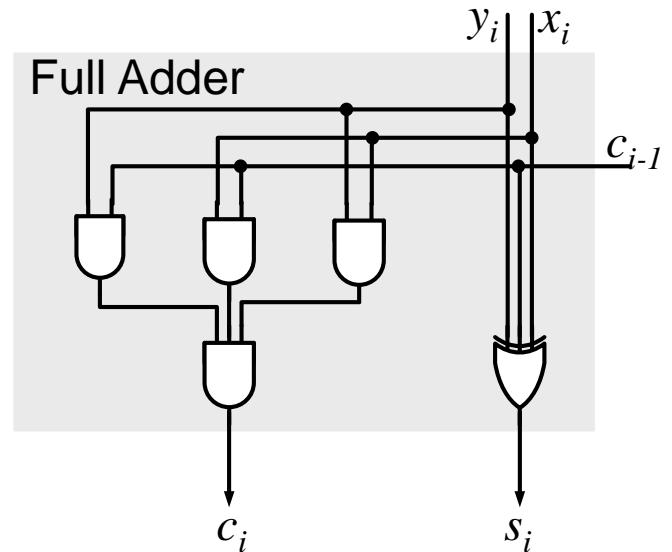
$x_i$	$y_i$	$c_{i-1}$	$s_i$	$c_i$
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Logic functions

$$s_i = x_i \oplus y_i \oplus c_{i-1}$$

$$c_i = x_i y_i + x_i c_{i-1} + y_i c_{i-1}$$

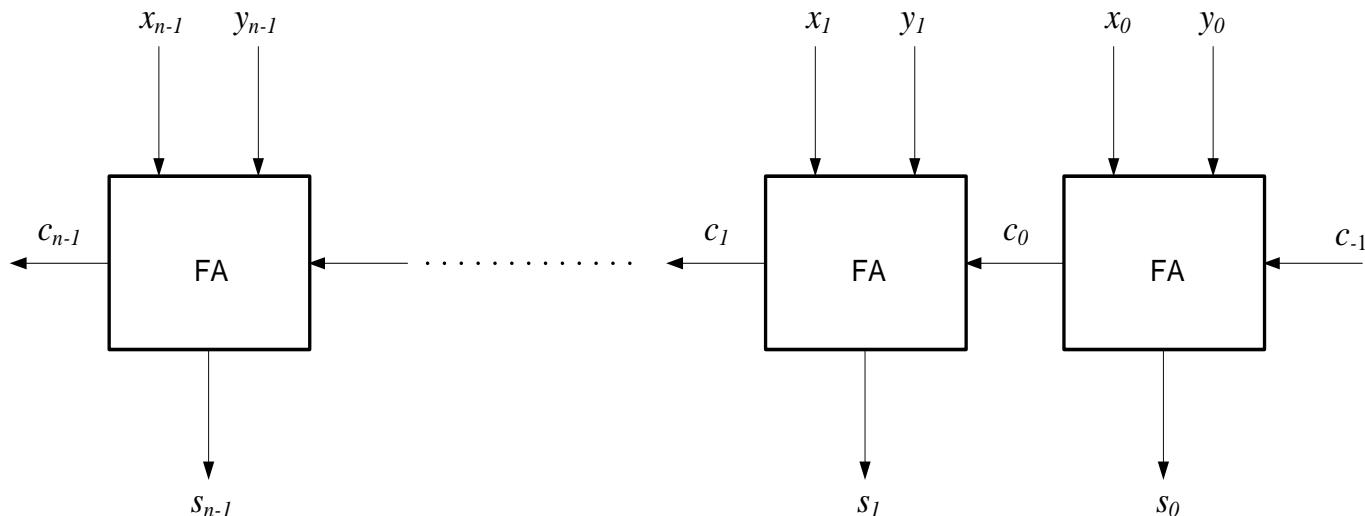
2-level logic



Hardware overhead: 5 gates. Delay:  $2t_{pd}$

# Ripple-Carry Adder

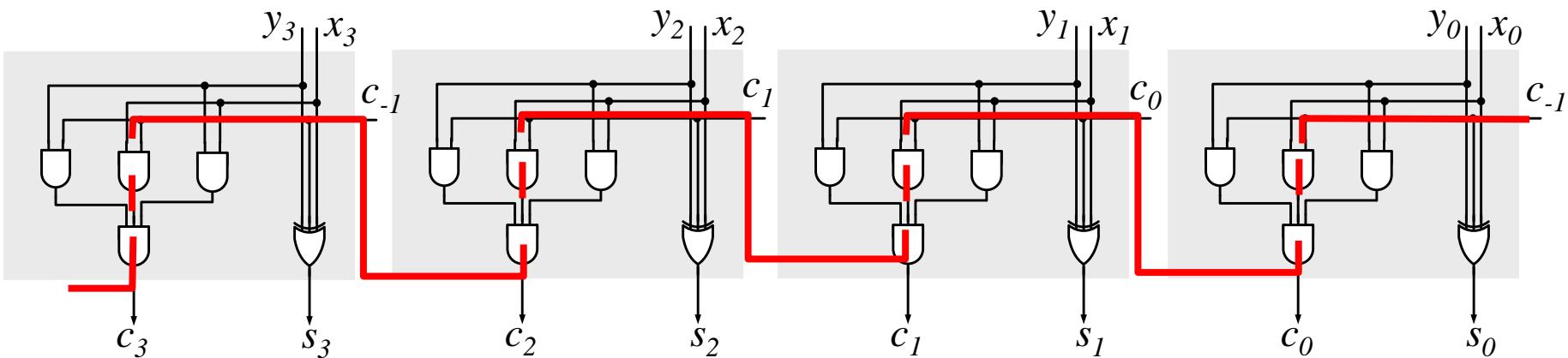
- A Ripple-Carry Adder is built as cascade of  $n$  full adders
  - Hardware overhead:  $5n$  gates, very cheap
  - Delay:  $2nt_{pd}$ , very slow



The carry-signal  $c_{n-1}$  has the highest delay, since it must go through all addition stages

# Ripple-Carry Adder

- Example: 4-bit Ripple-Carry Adder



# Ripple Carry Adder in VHDL

```
architecture ripple_carry of add is

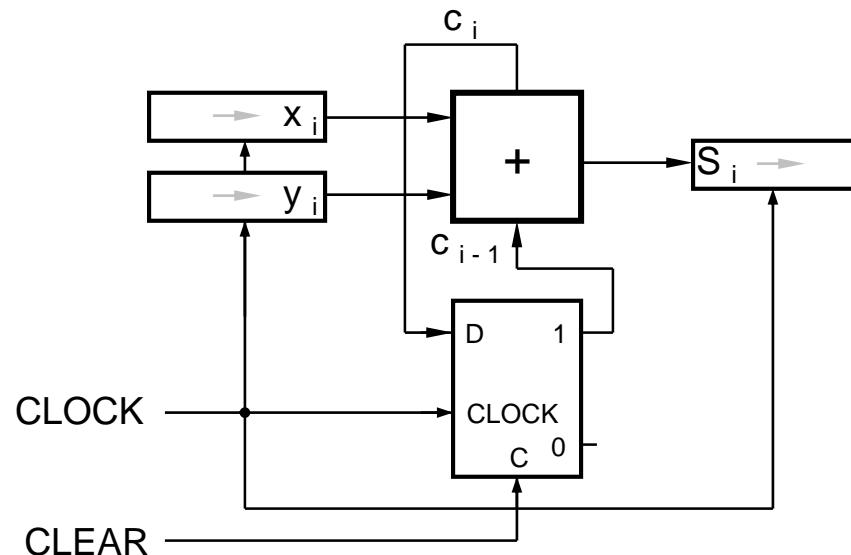
constant n : integer := ... ;
signal X, Y, S, C : std_logic_vector (n-1 downto 0);
signal OVL, CARRY_IN : Std_logic;

begin
for i in 1 to n-1 loop
    S(i) <= X(i) xor Y(i) xor C(i-1);
    C(i) <= (X(i) and Y(i)) or (X(i) and C (i-1)) or (Y(i) and C(i-1));
end loop ;
S(0) <= X(0) xor Y(0) xor CARRY_IN;
C(0) <= (X(0) and Y(0)) or (X(0) and CARRY_IN) or (Y(0) and CARRY_IN);
OVL <= C(n-1);

end ripple_carry;
```

# Sequential Adder (Bit-serial)

- ❑ A single full adder is used
  - The addition is done stepwise
  - The carry is saved in a flip flop
  - Parallel to serial converter is needed at the inputs
  - Serial to parallel converter at the output

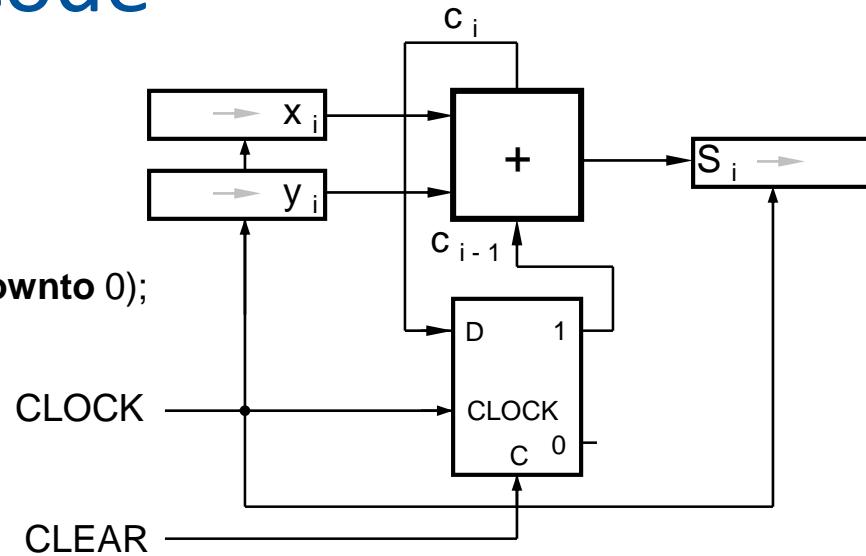


# Bit-Serial Adder: VHDL-Code

```

architecture bit_serial of add is
constant n : integer := ... ;
signal XREG, YREG, SREG : std_logic_vector (n-1 downto 0);
signal X, Y, S, CI, CO : std_logic_vector;
begin
    X <= XREG(0);  Y <= YREG(0);
    S <= X xor Y xor CI;
    CO <= (X and Y) or (X and CI) or (Y and CI);
    sch_r: process (CLK)
        if (CLK'event & CLK = '1') then
            CI <= CO;
            XREG <= '0' & XREG (n-1 downto 1); -- shift left
            YREG <= '0' & YREG (n-1 downto 1);
            SREG <= S & SREG (n-1 downto 1);
        end if ;
    end process ;
end bit_serial ;

```



# Carry-Lookahead Adder

- How can we speed up the slow carry signal?

$$\begin{aligned}g_i &= x_i \cdot y_i \\p_i &= x_i + y_i\end{aligned}$$

- Introduction of two help functions

- $g_i$  ... "generate carry":  $g_i = 1$  a carry will be generated in position  $i$  ( $c_i = 1$ ), regardless of  $c_{i-1}$
- $p_i$  ... "propagate carry":  $p_i = 1$ ,  $i$  will be propagated ( $c_i = 1$ ); otherwise, the carry will be absorbed

- Full adder-equations

$$s_i = x_i \oplus y_i \oplus c_{i-1}$$

$$c_i = x_i y_i + x_i c_{i-1} + y_i c_{i-1}$$



$$s_i = g_i \oplus p_i \oplus c_{i-1}$$

$$c_i = g_i + p_i c_{i-1}$$

Advantage? → the right expression for  $c_i$  is simple

# Carry-Lookahead Adder

- Carry-logic for ripple-carry adder

$$c_0 = x_0y_0 + x_0c_{-1} + y_0c_{-1}$$

$$s_i = g_i \oplus p_i \oplus c_{i-1}$$

$$c_1 = x_1y_1 + [x_1x_0y_0 + x_1x_0c_{-1} + x_1y_0c_{-1}] + [y_1x_0y_0 + y_1x_0c_{-1} + y_1y_0c_{-1}]$$

$$c_i = g_i + p_i c_{i-1}$$

⋮

- Carry-logic for carry-lookahead Adder

$$c_0 = g_0 + p_0c_{-1}$$

$$c_1 = g_1 + p_1c_0 = g_1 + p_1g_0 + p_1p_0c_{-1}$$

$$c_2 = g_2 + p_2c_1 = g_2 + p_2g_1 + p_2p_1g_0 + p_2p_1p_0c_{-1}$$

⋮

A 2-level circuit that generates all carry simultaneously is called **Carry-Lookahead Generator**

$$g_i = x_i \cdot y_i$$

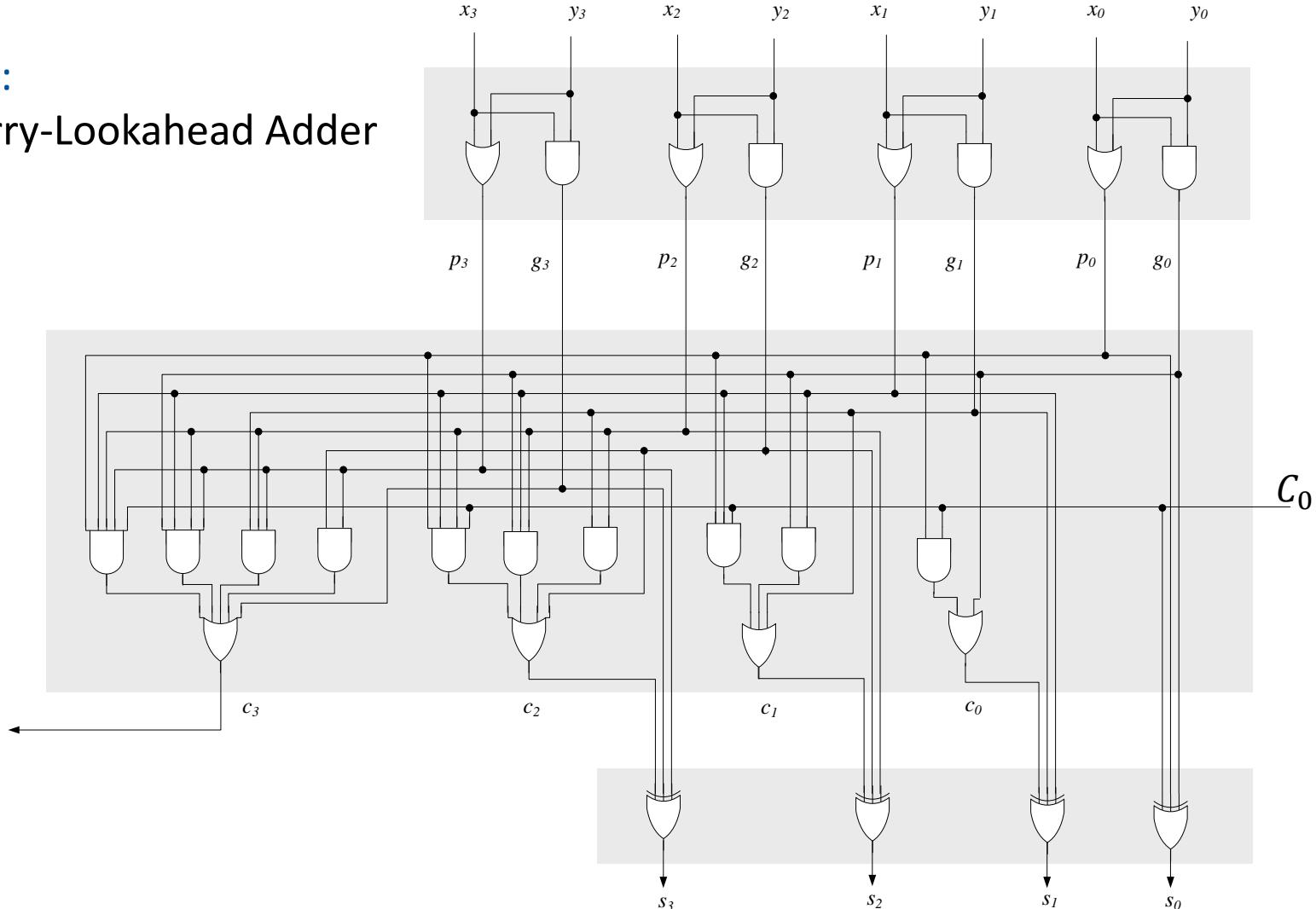
$$p_i = x_i + y_i$$

$$s_i = g_i \oplus p_i \oplus c_{i-1}$$

$$c_i = g_i + p_i c_{i-1}$$

# Carry-Lookahead Adder

- Example:  
4-bit Carry-Lookahead Adder



# Carry-Lookahead Adder

- Hardware overhead
  - Carry-Lookahead Generator
    - logic function for  $c_i$  has  $i+2$  product terms, with one literal. It can therefore be realized with  $i+2$  gates.
    - Sum of all product terms  $c_i$ :  
$$\sum_{i=0}^{n-1} (i+2) = \frac{n^2}{2} + \frac{3n}{2}$$
  - Generation of help functions  $p_i$  and  $g_i$ :  $2n$  gates
  - Computing the sums  $s_i$ :  $n$  gates
  - Total hardware overhead:  
$$\frac{n^2}{2} + \frac{9n}{2}$$
- Delay:  $4t_{pd}$

# Combinational Adder

- Side-by-side comparison of combinational  $n$ -bit adder

	Ripple-Carry	Carry-Lookahead	2-Level Logic
Hardware overhead [#gates]	$5n$	$\frac{n^2}{2} + \frac{9n}{2}$	$O(2^{n+1})$
Delay [ $t_{pd}$ ]	$2n$	4	2

- Example

16-bit adder,  $t_{pd} = 0.1$  ns

Ripple-Carry: 80 gates, 3.2 ns delay

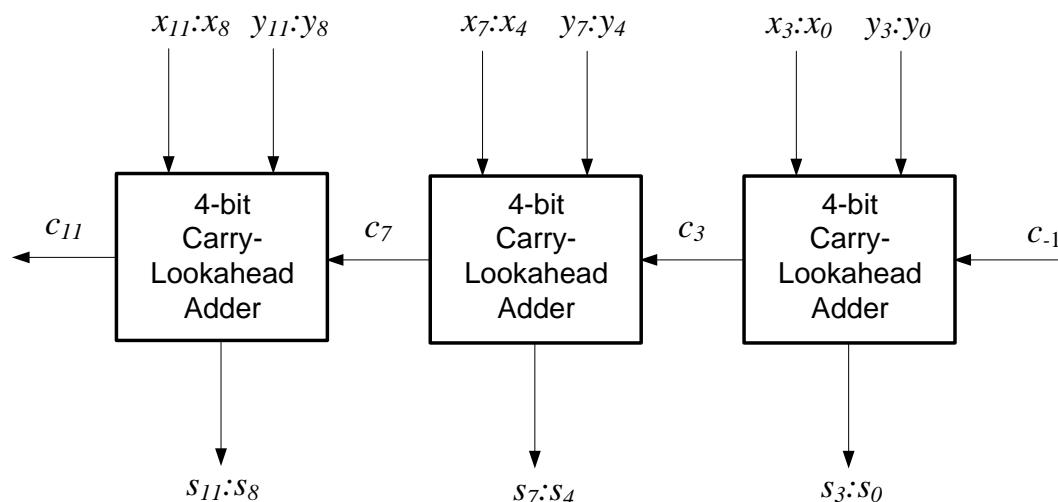
Carry-Lookahead: 200 gates, 0.4 ns delay

2-level logic:  $\geq 130000$  gates, 0.2 ns delay

# Combinational Adder

- ❑ Adder for large bit width
  - Carry-lookahead adders are expensive: quadratic hardware overhead for large  $n$  is not practical
  - Alternative:
    - Realize carry-lookahead generator in multiple level logic
    - Hybrid adder combines various adders such as ripple-carry and carry-lookahead.

- ❑ Example:  
4-bit 3-level logic adder  
with carry-lookahead adders



# Subtractors

- ❑ Subtractors are built in the same way as adders
  - 1-bit full subtractor:

$$d_i = x_i \oplus y_i \oplus b_{i-1}$$

$$b_i = \bar{x}_i y_i + \bar{x}_i b_{i-1} + y_i b_{i-1}$$

- Like adders, there is ripple-borrow and borrow-lookahead subtractors

# Adder / Subtractor

- Adder/Subtractor for 2-complement numbers
  - In 2's complement representation subtraction is done through addition
- Example

Adder/Subtraction circuit

with following operations:

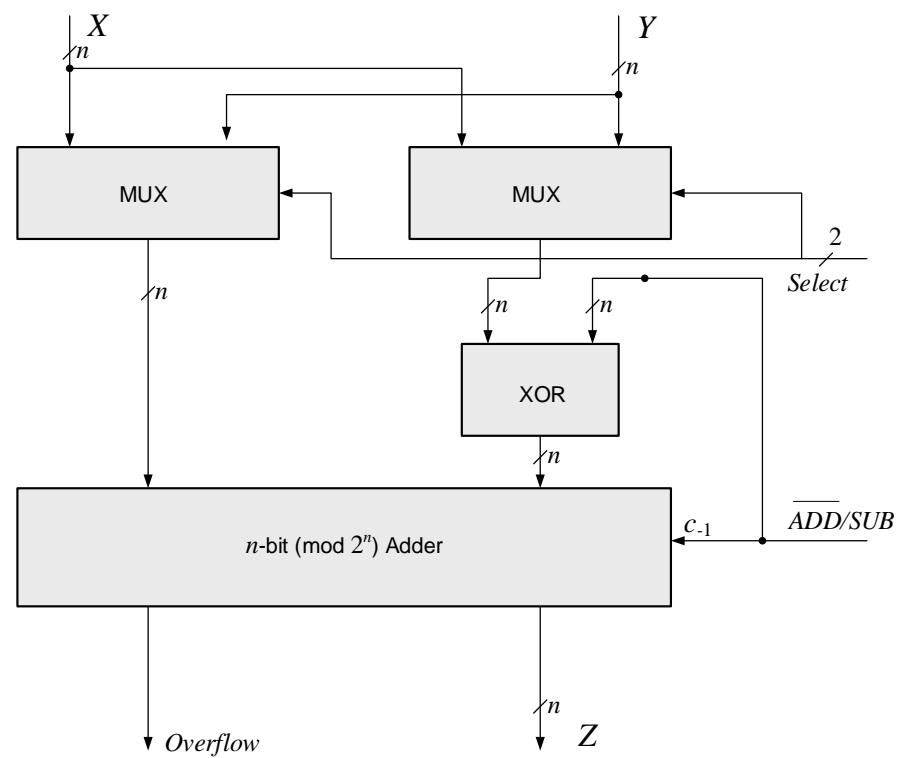
$$X+Y$$

$$X-Y$$

$$Y-X$$

$$2X$$

$$2Y$$



# Multiplier

- Array-Multipliers: combinational circuit to perform multiplication
  - Principle “shift and add”: result is the sum of partial shifted sums

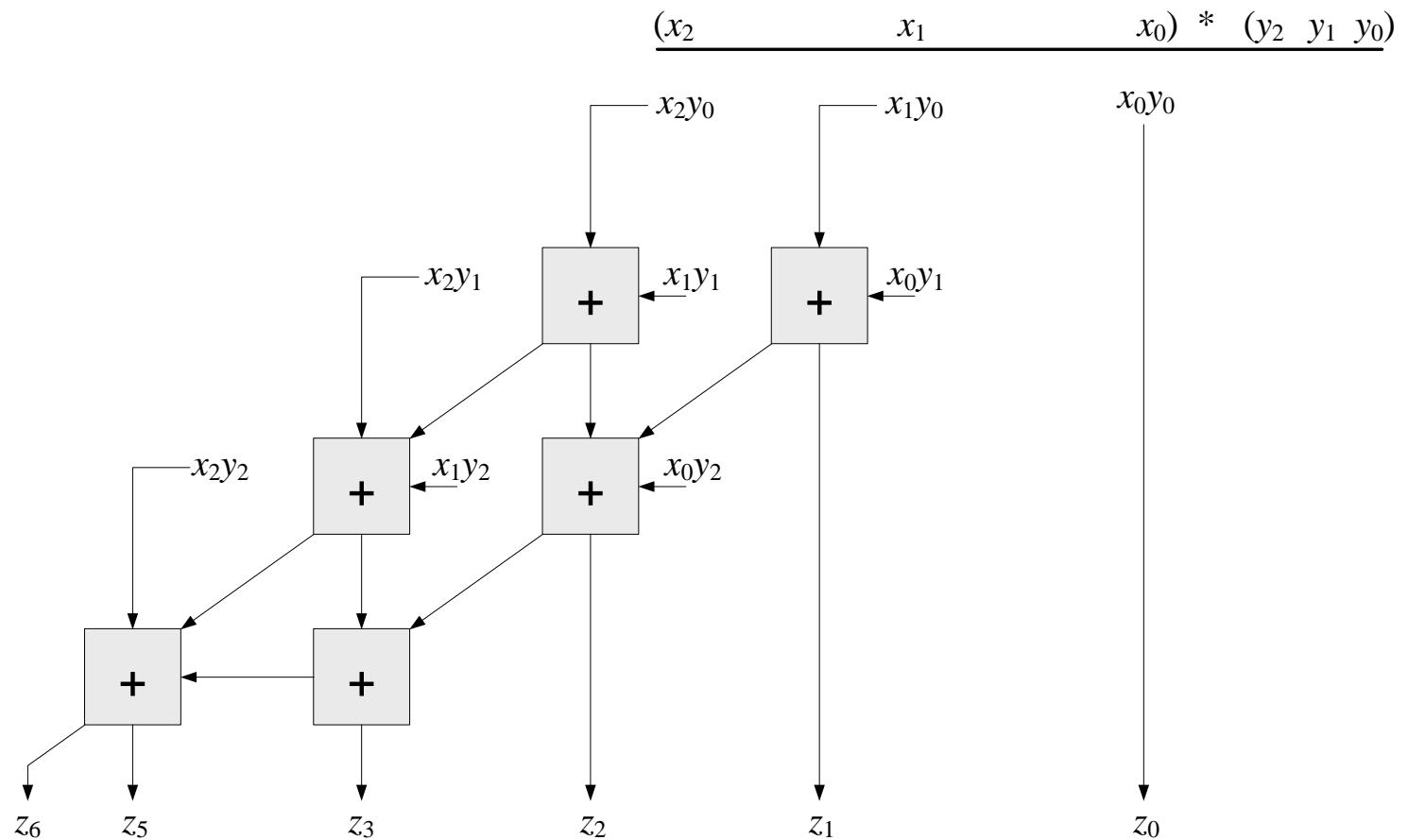
$$X \geq 0; Y \geq 0$$

$$X * Y = \left( \sum_{i=0}^{n-1} x_i 2^i \right) * \left( \sum_{j=0}^{n-1} y_j 2^j \right) = \sum_{i=0}^{n-1} \left( 2^i \cdot \sum_{j=0}^{n-1} x_i y_j 2^j \right)$$

- Start with the most significant bit and shift right, or
  - Start with the least significant bit and shift left
- Hardware overhead:  $O(n^2)$  gates
  - $n^2$  2-bit multiplications:  $n^2$  2-AND gates
  - $n$   $n$ -bit additions:  $n(n-1)$  full adder
- Delay:  $O(n)$

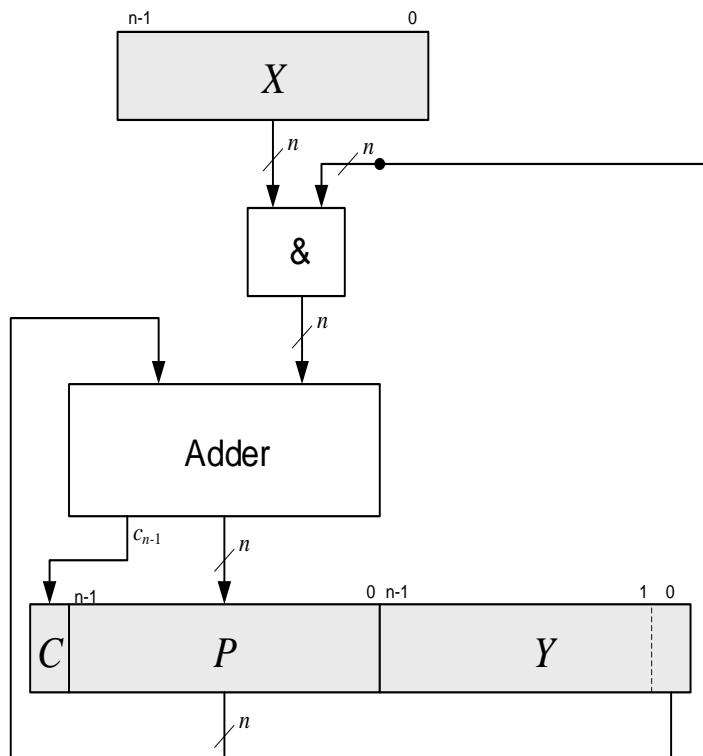
# Array-Multiplier

- Example: 3 x 3 bit array-multiplication  $Z=X \cdot Y$



# Sequential Multiplier

- Partial sums are accumulated stepwise



## Shift & Add Algorithm

1. load  $X, Y$
2.  $C \leftarrow 0, P \leftarrow 0$
3. Iterate  $n$ -times:
  - if  $Y(0)=1$   
then  $P \leftarrow P+X$
  - else  $P \leftarrow P+0$
  - shift  $(C, P, Y)$  1 digit left
4. Result is in  $(P, Y)$

## Remarks

- $Y$  is stored in a partial sum register  $\rightarrow$  save 1 register
- “logic shift”, i.e. fill right with 0s

# Sequential Multiplier: Shift & Add

Example:  $n=4$ ,  $X = 14_{10} = 1110$ ,  $Y = 7_{10} = 0111$

<i>C</i>	<i>P</i>	<i>Y</i>	next step	Operation
0	0000	0111		Iteration 1: $Y(0) = 1 \rightarrow$ add $X$
+ 1110				
0	1110	0111	Shift right	
0	0111	0011		Iteration 2: $Y(0) = 1 \rightarrow$ add $X$
+ 1110				
1	0101	0011	Shift right	
0	1010	1001		Iteration 3: $Y(0) = 1 \rightarrow$ add $X$
+ 1110				
1	1000	1001	Shift right	
0	1100	0100		Iteration 4: $Y(0) = 0 \rightarrow$ add 0
+ 0000				
0	1100	0100	Shift right	
0	0110	0010		Result $01100010 = 98_{10}$

# Sequential Multiplier– VHDL Code

```
constant n : integer := . . .
variable X, Y : std_logic_vector (n-1 downto 0);
variable P : std_logic_vector (n-1 downto 0);
variable C : std_logic;

begin
    P := "0 ... 0"; C := '0'
    for i in 0 to n-1 loop
        if Y(0) = '1' then
            P := (P(n-1 downto 0) + X);
        else NULL;
        end if;
        P := C & P(n-1 downto 1);
        C := '0'
        Y := P(0) & Y (n-1 downto 1);
    end loop;
end;
```

# Multiplication with Signed Numbers

## □ Case 1: sign and magnitude representation

1. Compute magnitude and sign

$$X = (s_x, |X|), \quad Y = (s_y, |Y|)$$

2. Multiply the magnitudes  $|Z| = |X| * |Y|$

3. Compute the sign of the result  $s_z = s_x \oplus s_y$

4. Negate the result if negative

# Multiplication with Signed Numbers

- Case 2: sequential multiplication (shift & add) with 2-complement operands
  - If  $X < 0$ :
    - “Arithmetic shift” (signed shift, sign extension) : shift left with sign instead of 0
    - Sign in position  $p_{n-1}$ ; on overflow in  $c_{n-1} \rightarrow$  shift  $p_{n-1} + c_{n-1}$
  - If  $Y < 0$ :
    - Then  $(2^n - |Y|) \cdot X$  is what was computed
    - We need to subtract  $2^n X$  through  $P \leftarrow P - X$  to get the correct result

# Sequential Multiplier

- Example 1:  $n=4$ ,  $X = -7_{10} = 1001$ ,  $Y = 5_{10} = 0101$

$C$	$P$	$Y$	Iteration	Operation
0	0000	0101		
+ 1001			Iteration 1: $Y(0) = 1 \rightarrow \text{Add } X$	
	0 1001	0101		shift right
	0 1100	1010	Iteration 2: $Y(0) = 0 \rightarrow \text{Add } 0$	
+ 0000				
	0 1100	1010		shift right
	0 1110	0101	Iteration 3: $Y(0) = 1 \rightarrow \text{Add } X$	
+ 1001				
	1 0111	0101		shift right
	0 1011	1010	Iteration 4: $Y(0) = 0 \rightarrow \text{Add } 0$	
+ 0000				
	0 1011	1010		shift right
	0 1101	1101	Result: $11011101 = -35_{10}$	

# Multiplication with Signed Numbers- VHDL Code

```
constant n : integer := ...;
variable P, X, Y : std_logic_vector (n-1 downto 0);
variable YS, C : std_logic;
begin
    P := 0; C := 0; YS := Y (n-1);
    for i in 0 to n-1 loop
        if Y (0) = '1' then
            C & P := (P(n-1 downto n) + X);
        else null;
        end if;
        P & Y := (C or P (n-1)) & P & Y (n-1 downto 1);
    end loop;
    if YS = '1' then P := P - X; else null; end if;
end;
```

# Multiplication with Signed Numbers

## Example 2

X(-3)  
1101

P Y(+5) 0101

$$\begin{array}{r} 0000 \\ +1101 \\ \hline \end{array} \quad \text{Iteration 1: } Y(0)=1, \text{ Add X}$$

$$\begin{array}{r} 1\ 1101 \\ \hline \end{array} \quad \text{Shift 1}$$

$$\begin{array}{r} 1110 \\ +0000 \\ \hline \end{array} \quad \text{Iteration 2: } Y(0)=0, \text{ Add 0}$$

$$\begin{array}{r} 1110 \\ \hline \end{array} \quad \text{Shift 2}$$

$$\begin{array}{r} 1111 \\ +1101 \\ \hline \end{array} \quad \text{Iteration 3: } Y(0)=1, \text{ Add X}$$

$$\begin{array}{r} 1\ 1100 \\ \hline \end{array} \quad \text{Shift 3}$$

$$\begin{array}{r} 1110 \\ +0010 \\ \hline \end{array} \quad \text{Iteration 4: } Y(0)=0, \text{ shift 4 (add 0 + shift)}$$

$$\begin{array}{r} 1111 \\ +0001 \\ \hline \end{array} \quad -15$$

$$\begin{array}{r} 1111 \\ 0000 \\ 0000 \\ 0000 \\ \hline \end{array} \quad \begin{array}{l} |0001 \\ |1110 \text{ (1s compl)} \\ |1111 \text{ (2s compl)} \end{array}$$

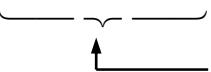
# Multiplication with Signed Numbers

## □ Example 3

X(+3)	P	Y(-5)	1011
0011	0000   1011		Iteration 1: Y(0)=1, Add X
	+0011		
	<hr/>		
	0011   1011		Shift 1
	0001   1101		Iteration 2: Y(0)=1, Add X
	+0011		
	<hr/>		
	0100   1101		Shift 2
	0010   0110		Iteration 3: Y(0)=0, shift 3
	0001   0011		Iteration 4: Y(0)=1, Add X
	+0011		
	<hr/>		
	0100   0011		Shift 4
	0010   0001		Y < 0, correction: subtract $2^nX$
- X :	+ 1101		
	1111   0001		
	<hr/>		
		↑	-15

# Multiplication with Signed Numbers

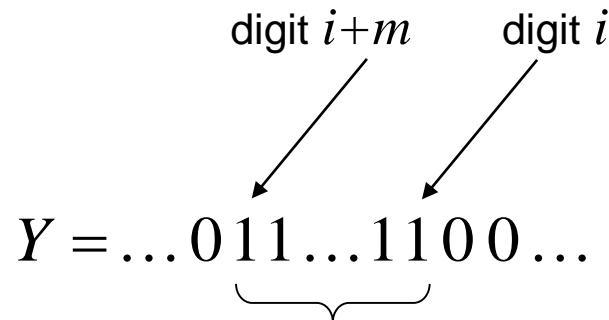
## □ Example 4

X(-3)	P	Y(-5)	1011
1101	0000   1011		Iteration 1: Y(0)=1, Add X
	+1101		
	<u>1 1101</u>   1011		Shift 1
	1110   1101		Iteration 2: Y(0)=1, Add X
	+1101		
	<u>1 1011</u>   1101		Shift 2
	1101   1110		Iteration 3: Y(0)=0, Shift 3
	1110   1111		Iteration 4: Y(0)=1, Add X
	+1101		
	<u>1 1011</u>   1111		Shift 4
	1101   1111		Y < 0, correction: subtract $2^nX$
- X : + 0011			
	0000   1111		Final Result
			15

# Booth Multiplier

## □ Observations

- No need to add when  $Y$  is equal '0'. Shift only!
- It is not necessary to perform  $(m+1)$  additions when there is a chain of  $(m+1)$  1's in  $Y$ .
- We just need to subtract  $X$  at the begin of the chain and add  $X$  back at the end of the chain:



$$X \cdot \sum_{j=i}^{i+m} 2^j = X \cdot \left( \sum_{j=0}^{i+m} 2^j - \sum_{l=0}^{i-1} 2^l \right) = X \left( 2^{i+m+1} - 1 - 2^i + 1 \right) = X \cdot (2^{i+m+1} - 2^i)$$

# Booth Multiplier

## ❑ Booth Recoding: $Y$ is translated into $Y'$

0101010101010  
0111011101110

1.  $Y$  is extended by  $y_{-1} = 0$

2.  $Y'$  is built as follows  $y'_i = y_{i-1} - y_i$

◦ Meaning of  $y'$ :

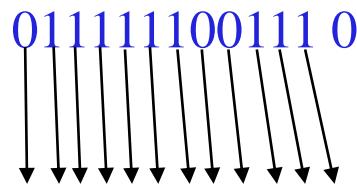
$y' = 0$  : shift right

$y' = -1$  (rsp. "S") :  $P \leftarrow P - X$ , right shift (Begin of a chain of 1s)

$y' = 1$  (rsp. "A") :  $P \leftarrow P + X$ , right shift (End of a chain of 1s)

$$Y = 01111100111$$

—————> Needs 9 additions



$$Y' = A00000S0A00S$$

—————> needs 4 additions/subtraction

## ❑ Example

# Booth Multiplier

- Example:  $n=5$ ,  $X = 11_{10} = 01011$ ,  $Y = 15_{10} = 01111 \rightarrow Y' = A000S$

<i>C</i>	<i>P</i>	
0	00000 00000	Iteration 1: $Y'(0) = S \rightarrow$ subtract $X$
-	01011	
1	10101 00000	right shift
0	11010 10000	Iteration 2: $Y'(1) = 0 \rightarrow$ right shift
0	11101 01000	Iteration 3: $Y'(2) = 0 \rightarrow$ right shift
0	11110 10100	Iteration 4: $Y'(3) = 0 \rightarrow$ right shift
0	11111 01010	Iteration 5: $Y'(4) = A \rightarrow$ add $X$
+	01011	
1	01010 01010	right shift
0	00101 00101	Result: $0010100101 = 165_{10}$

Make sure the shift (arithmetic/logic) is done correctly  
for 2-complement numbers

# Divider

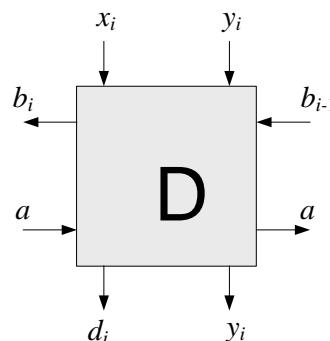
- Division ( $X = Q \cdot Y + R$ ) after "Restoring Algorithm"
  - Classic method, assuming always  $y_i=1$   
Then subtract the divisor.
    - If the result is positive then move to the next digit
    - If the result is negative, then set  $y_i=0$  and add the divisor back (i.e. cancel subtraction)
- Example:  $n=3$ ,  $X=3_{10}$ ,  $Y=2_{10}$

$$\begin{array}{r} 00011 \\ - 010 \\ \hline 1110 \\ 0001 \\ - 010 \\ \hline 1111 \\ 0011 \\ - 010 \\ \hline 001 \end{array} / 010 = \begin{array}{r} \cancel{1} \cancel{0} \cancel{1} 01 \\ \underbrace{\phantom{0}}_{Q=001=1_{10}} \end{array}$$

$R=001=1_{10}$

# Divider

- An Array-Divider performs the division as combinational circuit
  - 1 subtractor (Ex. ripple-borrow subtractor) is needed per row
  - How to perform correction in case of negative result?
    - The result is negative when the borrow (left most bit) is equal '1'
    - Idea: feed the borrow back (right) as signal  $a$ .
      - $a=0$ , the subtraction was correct, no need to correct
      - $a=1$ , instead of the difference, use the minuend
- Special full-subtractor "D-Box":

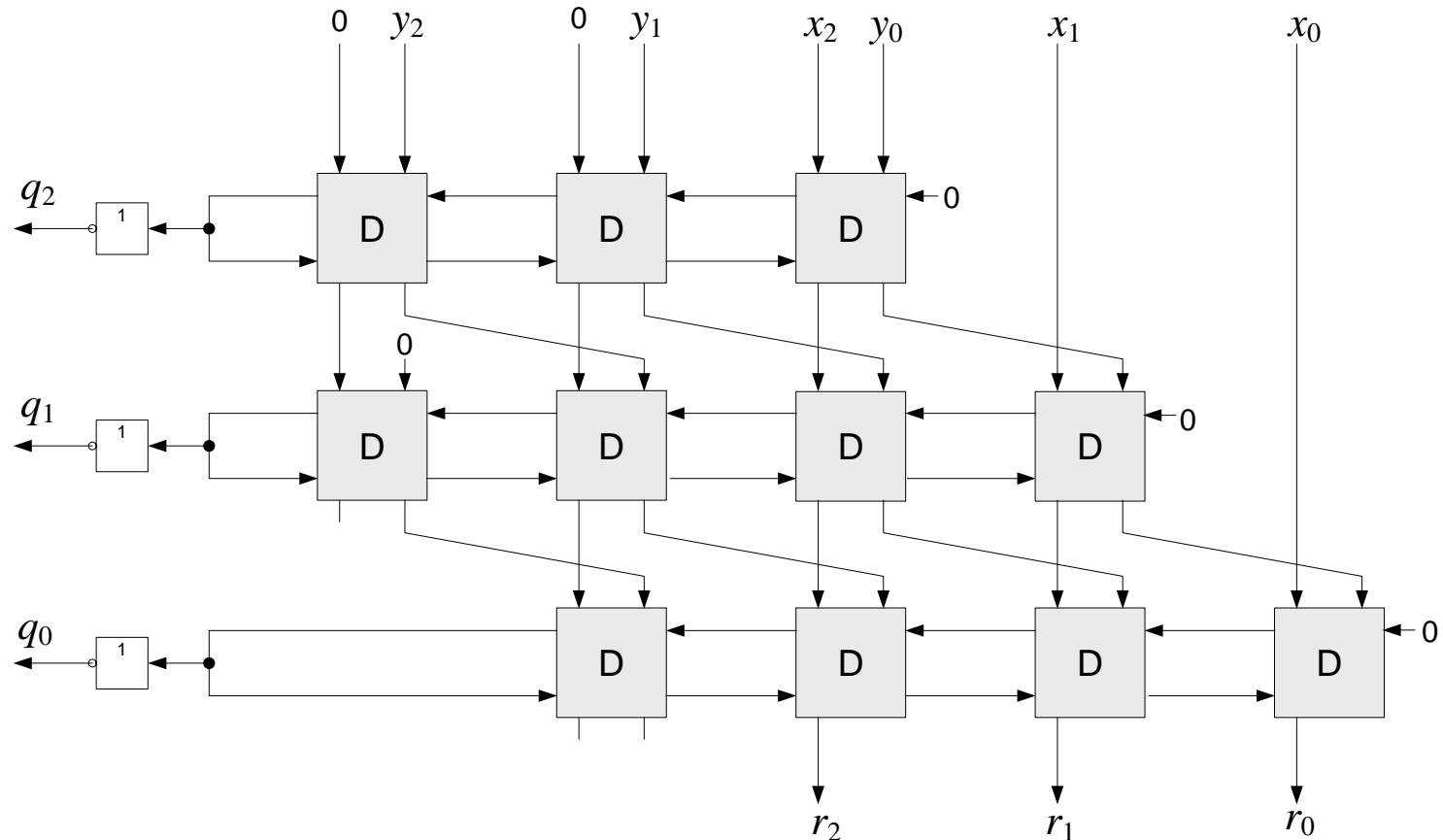


$$d_i = ax_i + \bar{a}(x_i \oplus y_i \oplus b_{i-1})$$

$$b_i = \bar{x}_i y_i + \bar{x}_i b_{i-1} + y_i b_{i-1}$$

# Array-Divider

- Example:  $n=3: (x_2, x_1, x_0) / (y_2, y_1, y_0) = (q_2, q_1, q_0) + (r_2, r_1, r_0) / (y_2, y_1, y_0)$

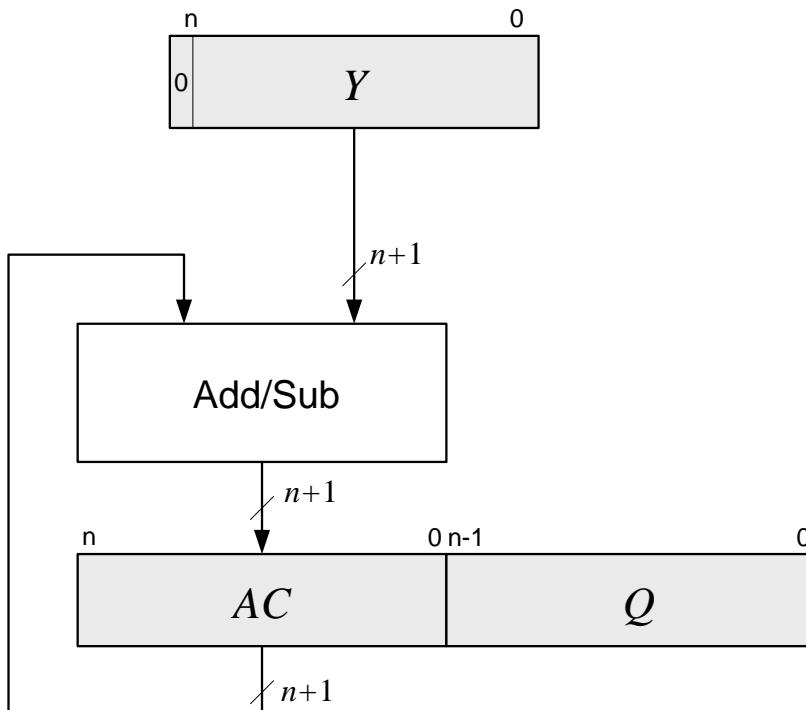


# Sequential Divider – Restoring Algorithm

- Quotient is computed stepwise

## Algorithm

- Load operand register  $Y$
- $AC \leftarrow 0, Q \leftarrow X$
- Iterate  $n$ -times:
  - shift  $(AC, Q)$  one position left
  - $AC \leftarrow AC - Y$
  - if  $AC < 0$ 
    - then  $AC \leftarrow AC + Y$
    - else  $Q(0) \leftarrow 1$
- Quotient is in  $Q$ , Rest in  $AC$



## Remarks

- “logic shift” → Fill left with 0
- $AC$  and  $Y$  are  $(n+1)$ -bit numbers in 2-complement

# Sequential Divider – Restoring Algorithm

## Example:

$n=4$ ,

$X = 14_{10} = 1110$ ,

$Y = 6_{10} = 0110$

Operation
$\begin{array}{r} AC \quad Q \\ \hline 00000 \quad 1110 \\ - 00001 \quad 1100 \\ \hline 00110 \\ + 00110 \\ \hline 00001 \quad 1100 \end{array}$
Iteration 1: Left shift Subtract Y
$\begin{array}{r} \\ - 11011 \quad 1100 \\ \hline 00110 \\ + 00110 \\ \hline 00011 \quad 1000 \end{array}$
add Y back
$\begin{array}{r} \\ - 00110 \\ \hline 11101 \quad 1000 \\ + 00110 \\ \hline 00011 \quad 1000 \end{array}$
Iteration 2: Left shift Subtract Y
$\begin{array}{r} \\ - 00110 \\ \hline 11101 \quad 1000 \\ + 00110 \\ \hline 00011 \quad 0000 \end{array}$
add Y back
$\begin{array}{r} \\ - 00110 \\ \hline 00001 \quad 0000 \\ - 00001 \quad 0001 \\ \hline 00010 \quad 0010 \\ - 00110 \\ \hline 11100 \quad 0010 \\ + 00110 \\ \hline 00010 \quad 0010 \end{array}$
Iteration 3: Left shift Subtract Y
$Q(0) \leftarrow 1$
Iteration 4: Left shift Subtract Y
add Y back

# Restoring Algorithm: VHDL-Code

**Computes  $Q = X/Y$ , Quotient in Q, Rest in AC**

```
constant n : integer := . . .;
signal X, Y, AC, Q : std_logic_vector (n-1 downto 0);
begin
    AC <= 0; Q <= X;
    for i in 1 to n loop
        AC & Q <= (AC & Q) sll 1;    -- sll: shift logic left
        AC <= AC -Y;
        if AC < 0 then
            AC <= AC + Y ;
        else Q(0) <= '1' ;
        end if ;
    end loop ;
end ;
```



Herbert Wertheim  
College of Engineering  
UNIVERSITY *of* FLORIDA