7 Arithmetic combinational circuits

7.1 Introductory concepts

- Arithmetic combinational circuits are the most typical example of **structured or array** combinational circuits
- Typically an n-bit arithmetic circuit can be decomposed into n 1-bit circuits connected in an appropriate way.
- Most typical example is an n-bit adder that can be thought of as a 1-dimensional array of 1-bit adders.
- Examples of arithmetic circuits that form 2-dimensional arrays of 1-bit cells include fast multiplication circuits and vector rotators.
- Even simple arithmetic circuits cannot be implemented in an unstructured way:

  Consider a 16-bit adder adding two 16-bit numbers.

  It is equivalent to a combinational circuit with 32 inputs and 16 outputs.

  The truth table of such a circuit has $2^{32} = 4,294,967,296$ rows and 16 columns.
- Unstructured implementation of such a big circuit is rather impossible.

7.2 An Incrementer

- An incrementer performs operation $y <= a + 1$ which can be implemented as a 1-dimensional array of 1-bit incrementers
- A **1-bit incrementer** has a 1-bit input a and an **input carry** c, and generate 1-bit output y and an **output carry** d.
- The signals are related by the following arithmetic equation

$$2 \cdot d + y = a + c \quad \text{or} \quad \frac{a+c}{2} = d + \frac{y}{2}$$

- It says that the result of 1-bit incrementation, y, and an output carry d are remainder and the quotient, respectively, from division of $a + c$ by 2.
- The truth table can now be easily created.

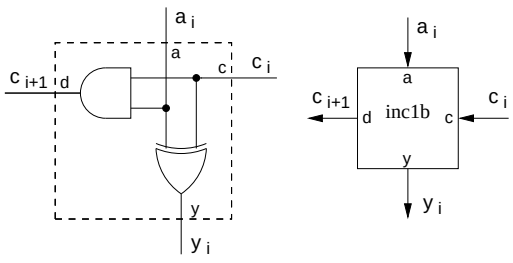
Truth table:

1	1	2	1
c	a	d	y
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

- From the truth table it is easy to write equations for the output signal y and the output carry d:

$$y = a \oplus c, \quad d = a \cdot c$$

The equations can be implemented as follows:



- The presence of the XOR gate is characteristic to all arithmetic circuits.

An n-bit Incrementer

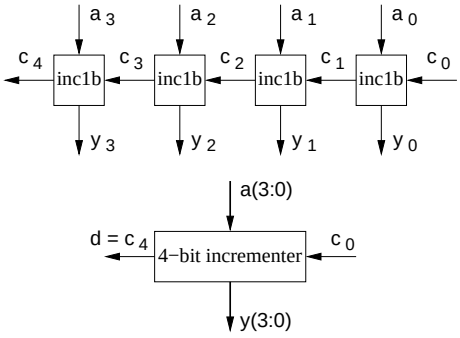
- In order to obtain an n-bit incrementer, we arrange n 1-bit incrementers in a 1-D array, connecting their output carry ports to respective input carry ports.
- If A and Y are numbers represented by the n-bit binary words, $a(n-1:0)$ and $y(n-1:0)$, respectively, then the n-bit incrementer performs the operation

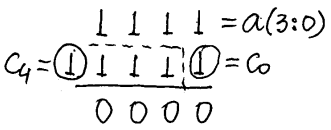
$$2^n d + Y = A + c_0$$

where $d = c_n$ is the 1-bit output carry, and c_0 is the 1-bit input carry.

- Note that when $c_0 = 0$ then $Y = A$, that is, no increment is performed.
- Note also that the output carry $c_n = 1$ **if and only if** all a_i and c_0 are 1.

A 4-bit incrementer





7.3 Adders

7.3.1 1-bit adder

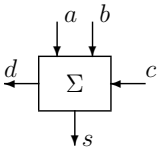
- Adders are fundamental building blocks of all arithmetic circuits.
- Following considerations of the previous section we conclude than an n-bit adder can be built using an array of 1-bit adders.
- **A 1-bit adder** has three inputs, a, b, c, and two outputs, d, s, known as the **output carry** and the **sum**, respectively.
- The 1-bit adder counts the number of ones at its three inputs and represents the result as a two-bit binary number.
- Hence, the defining arithmetic relationship between inputs and outputs can be written as:

$$a + b + c = (d, s)_2 = 2 \cdot d + s \quad \text{or} \quad \frac{a+b+c}{2} = d + \frac{s}{2}$$

- All three inputs are equivalent, but normally c is called the input carry.
- The **arithmetic equation** can be converted into a **truth table** which describes the relationship between three adder inputs c, b, a and two adder outputs, d, s, and then into the **logic equations**:

$$s = a \oplus b \oplus c$$
$$d = a \cdot b + b \cdot c + c \cdot a$$

A 1-bit adder:



Truth table:

c	b	a	1's	d	s
0	0	0	0	0	0
0	0	1	1	0	1
0	1	0	1	0	1
0	1	1	2	1	0
1	0	0	1	0	1
1	0	1	2	1	0
1	1	0	2	1	0
1	1	1	3	1	1

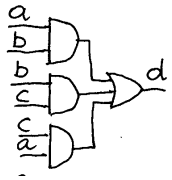
1-bit adder

Karnaugh Maps: **Logic equations:** **Generic implementation:**

$d = \sum(3, 5, 6, 7)$

c \ a	0	0	1	0
0	0	1	1	1
	b			

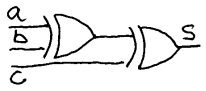
$$\begin{aligned} d &= a \cdot b + b \cdot c + a \cdot c \\ &= a \cdot b + c \cdot (a + b) \end{aligned}$$



$s = \sum(1, 2, 4, 7)$

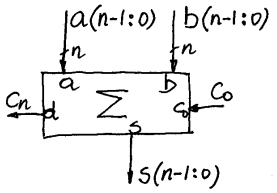
c \ a	0	1	0	1
1	1	0	1	0
	b			

$$\begin{aligned} s &= \bar{c} \cdot a \cdot \bar{b} + \bar{c} \cdot \bar{a} \cdot b + c \cdot \bar{a} \cdot \bar{b} + c \cdot a \cdot b \\ &= \bar{c} \cdot (a \cdot \bar{b} + \bar{a} \cdot b) + c \cdot (\bar{a} \cdot \bar{b} + a \cdot b) \\ &= \bar{c} \cdot (a \oplus b) + c \cdot (\overline{a \oplus b}) \\ &= a \oplus b \oplus c \end{aligned}$$



7.3.2 An n-bit adder

- An n-bit adder adds two n-bit binary numbers $a = (a_{n-1} \dots a_0)$ and $b = (b_{n-1} \dots b_0)$ and a 1-bit input carry c_0 and produces an n-bit sum $b = (s_{n-1} \dots s_0)$ and a 1-bit output carry d.



- This can be formally described in the following way:

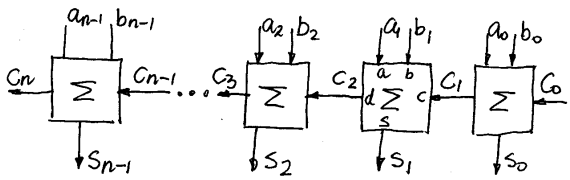
$$s = a + b + c_0 = \sum_{i=0}^{n-1} a_i 2^i + \sum_{i=0}^{n-1} b_i 2^i + c_0 = \sum_{i=0}^{n-1} (a_i + b_i) 2^i + c_0$$

Starting from the least significant position $(i = 0)$ we can convert

$$a_0 + b_0 + c_0 = 2c_1 + s_0 \quad \text{or in general for } i = 0, \dots, n-1: \quad a_i + b_i + c_i = 2c_{i+1} + s_i$$

Substituting we have $$s = \sum_{i=0}^{n-1} (a_i + b_i) 2^i + c_0 = c_n 2^n + \sum_{i=0}^{n-1} s_i 2^i$$

- Ripple-carry implementation of an n-bit adder built from 1-bit adders:
- Time taken for the carry to propagate from c_0 to c_n is proportional to n: $t_n = n \cdot t_1$



7.4 2's complement representation of numbers

- The 2's complement number system is an extension of a binary system to representation of also the negative numbers
- In the 2's complement system the most significant weight is negative, or alternatively the most significant digit (the sign digit) takes values $a_{n-1} \in (-1, 0)$
- Hence, an n-bit numeral $\mathbf{a} = (a_{n-1}, a_{n-2} \dots a_0)$ represents the number:

$$a = -a_{n-1}2^{n-1} + \sum_{i=0}^{n-2} a_i 2^i$$

All 3-bit 2's complement numbers:

-4 2 1	a
0 0 0	0
0 0 1	1
0 1 0	2
0 1 1	3
1 0 0	-4
1 0 1	-3
1 1 0	-2
1 1 1	-1

- Example:

$$a = [\underbrace{\bar{1}0110}_{A_{4:0}}] \begin{bmatrix} 2^4 \\ 2^3 \\ 2^2 \\ 2^1 \\ 2^0 \end{bmatrix} = -2^4 + 2^2 + 2^1 = -(10)_{10}$$

- The range of numbers represented is from $(10 \dots 0)_{\bar{2}} = -2^{n-1}$ to $(01 \dots 1)_{\bar{2}} = 2^{n-1} - 1$

7.5 Changing sign of a 2's complement number

- Complementing every digit of a 2'complement number: (Note that $\bar{a}_i = 1 - a_i$)

$$\bar{a} = -(1 - a_{n-1})2^{n-1} + \sum_{i=0}^{n-2}(1 - a_i)2^i = -2^{n-1} + \sum_{i=0}^{n-2} 2^i + a_{n-1}2^{n-1} - \sum_{i=0}^{n-2} a_i 2^i$$

Re-grouping the terms and noting that $\sum_{i=0}^{n-2} 2^i = 2^{n-1} - 1$, we have

$$\bar{a} = -1 - (-a_{n-1}2^{n-1} + \sum_{i=0}^{n-2} a_i 2^i) = -a - 1$$

or

$$-a = \bar{a} + 1$$

- Hence, to change the sign of a 2's complement number we **complement each digit and add 1**:

-32 16 8 4 2 1
a = 1 0 1 1 0 0 = -32+12=-20
ā = 0 1 0 0 1 1
+1 1
-a = 0 1 0 1 0 0 = +20

- **Sign extension:**

Note that increasing the number of bits we have extend to the left the bit sign.
For example: $(10011)_{\bar{2}} = (1111110011)_{\bar{2}}$ and $(010011)_{\bar{2}} = (0000010011)_{\bar{2}}$

7.6 Adding 2's complement numbers

- 2's complement n-bit numbers can be added using a standard binary adder. The n-bit result will be correct provided that the **overflow** does nor occur.
- Formal proof:

$$s = a+b+c_0 = -a_{n-1}2^{n-1}+\sum_{i=0}^{n-2} a_i 2^i - b_{n-1}2^{n-1}+\sum_{i=0}^{n-2} b_i 2^i + c_0 = -(a_{n-1}+b_{n-1})2^{n-1}+\sum_{i=0}^{n-2}(a_i+b_i)2^i+c_0$$

Substituting the 1-bit addition law

$$a_i + b_i + c_i = 2c_{i+1} + s_i, \qquad \text{we have}$$

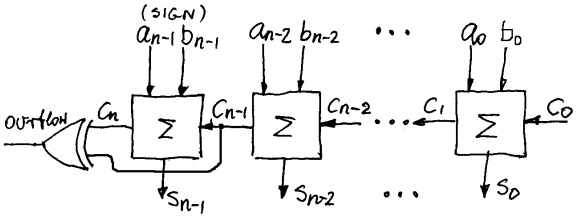
$$s = -(a_{n-1}+b_{n-1})2^{n-1} + 2c_{n-1}2^{n-2} + \sum_{i=0}^{n-2} s_i 2^i = c_{n-1}2^n - (a_{n-1}+b_{n-1}+c_{n-1})2^{n-1} + \sum_{i=0}^{n-2} s_i 2^i$$

Finally, we have

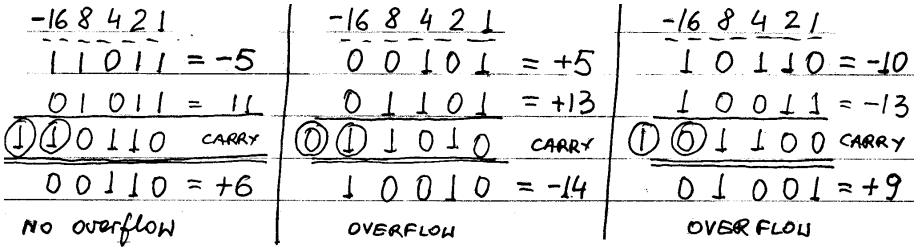
$$s = a + b + c_0 = (c_{n-1} - c_n)2^n - s_{n-1}2^{n-1} + \sum_{i=0}^{n-2} s_i 2^i$$

- When $c_{n-1} = c_n$ the above expression gives the proper 2's complement sum of a and b (and c_0)
- When $c_{n-1} \oplus c_n = 1$ **overflow** occurs and the result is "incorrect", that is, $\pm 2^n$ must be added for proper the interpretation of the result

Ripple-carry implementation of an n-bit 2'c complement adder:



Examples:



Note that:

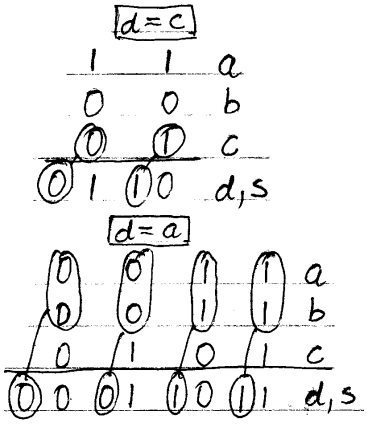
- An overflow can only occur when we are adding numbers of **the same sign**.
- In this case the c_n carry is equal to the sign bit but the c_{n-1} carry can be both 0 or 1.
- Adding numbers of opposite sign the c_{n-1} carry propagates through the sign position and $c_n = c_{n-1}$

7.7 Carry propagation and generation

- Designing adders it is useful to formulate the following carry propagation/generation conditions.
- Consider again a 1-bit adder in which input and output variables are related by the following equation:

$$2d + s = a + b + c$$

- **Carry propagation:** when $a \oplus b = 1$ (a and b are **different**) $d = c$ output carry is equal to the input carry.
- We say that the carry $c = d$ **is propagated through** this position of the adder.



- **Carry generation:** when $a \oplus b = 0$ (a and b are **identical**) $d = a,\ s = c$ output carry is equal to the addend bit $a = b$ and is independent of the input carry c.
- We say that the carry $d = a$ **is generated** at this position of the adder.

- Using two intermediate signals:

$g = a \cdot b$ — carry "1" generate
$p = a \oplus b$ — carry propagate

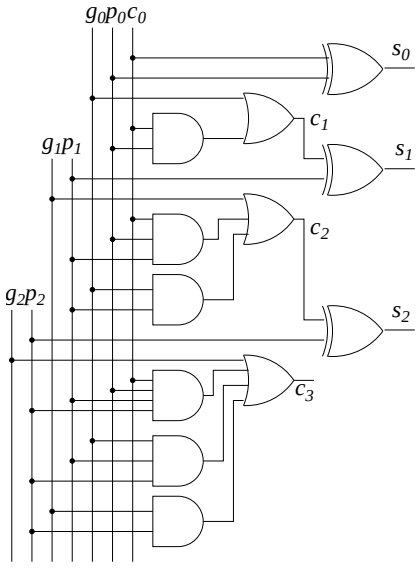
- the logic **equations for the 1-bit adder** can be written as:

$s = p \oplus c$ — the sum
$d = g + p \cdot c$ — the output carry

7.8 Carry Lookahead adder

- The ripple-carry n-bit adder is relatively slow, because the initial carry c_0 must travel through all n 1-bit adders.
- The can be avoided if we unfold the recursive way of calculating carry.
- This can be conveniently done using carry generate/propagete signals:

$$\begin{aligned} c_1 &= g_0 + p_0 \cdot c_0 \\ c_2 &= g_1 + p_1 \cdot c_1 = g_1 + p_1 \cdot g_0 + p_1 \cdot p_0 \cdot c_0 \\ c_3 &= g_2 + p_2 \cdot c_2 = g_2 + p_2 \cdot g_1 + p_2 \cdot p_1 \cdot g_0 + p_2 \cdot p_1 \cdot p_0 \cdot c_0 \\ &\ldots \end{aligned}$$



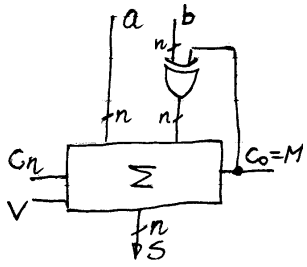
- Note from the logic diagram that the number of gates to produce the carry signal for the given position and their number of inputs grows with the adder position number
- Such an adder implementation is called a carry look-ahead adder an is the typical way of speeding up the adder operations.

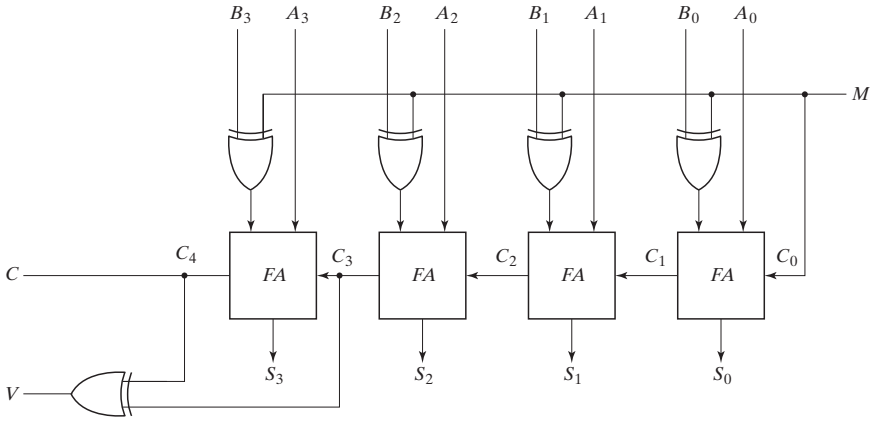
7.9 Subtractors

- Subtractors are typically used in the 2's complement system.
- Implementations of a subtractor involves the change of sign of the subtrahend through the complementation of its bits and an increment, according to the formula:

$$s = a - b = a + \bar{b} + 1$$

- the resulting block/logic diagrams:



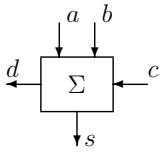


- It is also possible to build a (i-bit) subtractor according to the formula: $-2d + s = a - b - c$
- Note that the weight associated with carry is negative.
  Give the truth table and logic equation for such a subtractor. Compare it with a 1-bit adder.

7.10 VHDL specification of a 1-bit adder

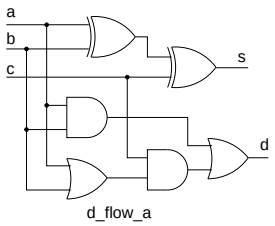
The 1-bit adder entity specifies input output ports:

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
ENTITY add_1b IS
  PORT (a, b, c : IN   bit ;
        s, d    : OUT  bit ) ;
END add_1b ;
```

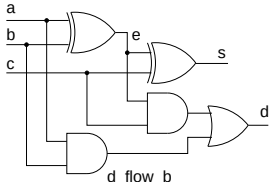


Many architectures are possible. Consider the following two:

```
-- dataflow architecture for  add_1b
ARCHITECTURE  d_flow_a  OF  add_1b  IS
BEGIN
  s <= a XOR b XOR c ;  -- a SIGNAL assignment
  d <= (a AND b) OR ((a OR b) AND c) ;
END  d_flow_a ;
```




```
-- another dataflow architecture for  add_1b
ARCHITECTURE  d_flow_b  OF  add_1b  IS
  SIGNAL e : std_logic ; -- internal signal declaration
BEGIN
  e <= a XOR b ;
  s <= e XOR c ;
  d <= (a AND b) OR (e AND c) ;
END  d_flow_b ;
```



- In the `d_flow_b` architecture we use an internal **signal** `e` which is specified as being of the type std_logic.
- The internal signals are always bi-directional and are used to simplify the description of the circuit.

Two more architectures:

In this architecture we specify the 1-bit adder in the form of its truth table. The truth table can be specified as a **constant array of binary words**:

```
ARCHITECTURE  ttbl  OF  add1_b IS
   TYPE arr_vec IS ARRAY (natural range <>)
               OF std_logic_vector(1 downto 0);
   CONSTANT add1bit : arr_vec(0 to 7) := (
   -- d s      cba
   ---------------- the truth table of a 1-bit adder
     "00",  -- 0  0
     "01",  -- 1  1
     "01",  -- 2  1
     "10",  -- 3  2
     "01",  -- 4  1
     "10",  -- 5  2
     "10",  -- 6  2
     "11"); -- 7  3
   SIGNAL  cba : std_logic_vector (2 downto 0) ;
   SIGNAL   ds : std_logic_vector (1 downto 0) ;
BEGIN
-- concatenation of three signals into one 3-bit word
  cba <= c & b & a ;
  --    reading from the truth table
  ds <= add1bit(conv_integer(unsigned(cba)));
   d <= ds(1) ;
   s <= ds(0) ;
END ttbl ;
```

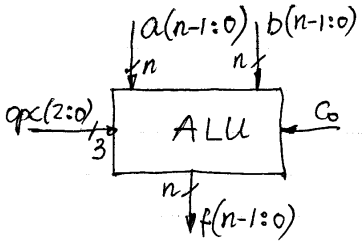
The 1-bit adder can be also specified arithmetically, leaving all the design/synthesis problems to the CAD tools:

```
ARCHITECTURE  cnt1  OF  add_1b IS
  SIGNAL   ds : std_logic_vector (1 downto 0) ;
BEGIN
 ds <=  unsigned('0' & a)
      + unsigned('0' & b)
      + unsigned('0' & c) ;
  d <= ds(1) ;
  s <= ds(0) ;
END cnt1 ;
```

- Note the various type conversion functions: `unsigned` and `conv_integer` .
- Type conversion informs the tools about desired method of conversion of binary vectors into numbers.

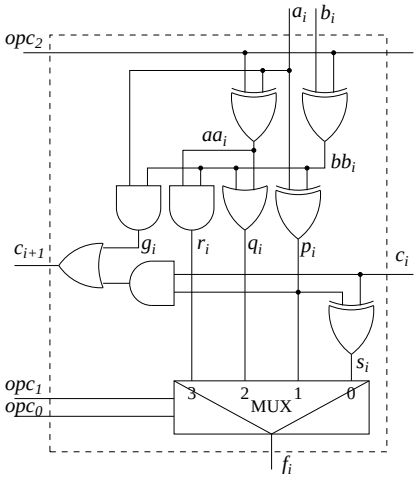
7.11 Arithmetic-Logic Units

- In practical applications adders and subtractors are group together with logic functions performed on n-bit binary words.



- As an example we consider an ALU performing eight different arithmetic and logic operations selected by a 3-bit operation code, opc(2:0)

- The i-th bit of the ALU can have the following logic structure:



- The operations performed are described by the following table:

opc	function
0 0 0	$a+b+c_0$
0 0 1	$a \oplus b$
0 1 0	a OR b
0 1 1	$a \cdot b$
1 0 0	$a-b-c_0$
1 0 1	$\overline{a \oplus b}$
1 1 0	$\overline{a \cdot b}$
1 1 1	$\overline{a \text{ OR } b}$

7.12 VHDL implementation of n-bit arithmetic circuits. The "generate" statement.

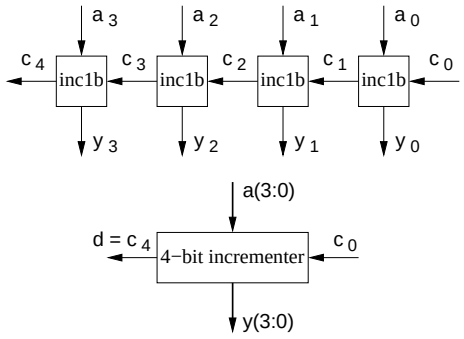
- In VHDL 1-bit arithmetic circuits are replicated to form a n-bit circuit using the **generate** statement of the form:

for ... generate

- The **generate** statement is a loop which replicates the **circuitry** specified by its body.
- Consider again a 4-bit incrementer as an illustrative example
- The **generate** loop will be repeated 4 times, and its body will describe the 1-bit incrementer in the following way:

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all;
ENTITY  incr4b IS
  GENERIC (N : natural := 3) ;
  PORT (
     a : IN  std_logic_vector (N downto 0) ;
    c0 : IN  std_logic ;
     y : OUT std_logic_vector (N downto 0) ;
     d : OUT std_logic ) ;
END  incr4b ;

ARCHITECTURE  GnrtStt  OF  incr4b  IS
  SIGNAL c : std_logic_vector (N+1 DOWNTO 0) ;
BEGIN
  c(0) <= c0 ;
  gnrt: FOR  i  IN  0 TO  N  GENERATE
      y(i) <= a(i) XOR c(i) ;
    c(i+1) <= a(i) AND c(i) ;
  END GENERATE gnrt ;
  d <= c(N+1) ;
END  GnrtStt ;
```



1-bit incrementer:

$$y_i = a_i \oplus c_i \ , \quad c_{i+1} = a_i \cdot c_i$$

- Note that the architecture consists of three **concurrent** statements (two assignment statements and one generate statements) that can be written in any order.
- Similarly, two assignment statements inside the generate statement can be also written in any order.

7.13 Structural specification of digital circuits

- In the previous examples VHDL statements described signal flow inside a component (logic circuit).
- It is possible to describe a digital circuit as interconnection of other components.
- Each constituent component is a black box with an unspecified, at this stage, function or behaviour, but with precisely defined ports.
- In the declarative part of the architecture we specify input-output ports of all components used in the architecture body in a way identical to the respective entity declarations for these components.
- The components may already exist in libraries, or can be specified later.
- Such structural specification of digital circuits is made in VHDL with the **Component Instantiation Statement** of the general form:

port map (...)

We use the n-bit incrementer to clarify the concept of structural specification.

We start with specification of a 1-bit incrementer as a separate component:

```
ENTITY  inc1b IS
  PORT ( a, c : IN  std_logic ;
         d, y : OUT std_logic  ) ;
END  inc1b ;

ARCHITECTURE  arch1  OF  inc1b  IS
BEGIN
      y <= a XOR c ;
     co <= a AND c ;
END  arch1 ;
```

The 4-bit incrementer instantiate the 1-bit component in the following way:

```
ARCHITECTURE  strctrl  OF  incr4b  IS

  COMPONENT inc1b
    PORT ( a, c : IN  std_logic ;
           d, y : OUT std_logic ) ;
    END COMPONENT inc1b ;

  SIGNAL c : std_logic_vector (N+1 downto 0) ;
BEGIN
  c(0) <= c0 ;
  gnrt : FOR  i  IN  a'RANGE  GENERATE
    u1 : inc1b PORT MAP ( a(i), c(i), c(i+1), y(i) );
  END GENERATE strctrl ;
  d <= c(4) ;
END  strcrl ;
```

Note that

- In the architecture body the library components are instantiated as many times as specified by the schematic describing the architecture using a port map component instantiation statement.
- Each component instantiation statement is labeled as its schematic equivalent. In the example, the 1-bit component is labeled `u1`
- Interconnections between components are specified by the port map statement. For it to work, every net in the schematic, that is, all external and internal signals, must be assigned a name.
- Every port map statement is associated by positions with the respective component declaration.

Note also

- In the **generate** statement we used expression " a'**range**" to describe the scope of the generate loop. Ut is an example of an **attribute** that we will study in some depth latter. Here we simply have:

$$\text{a'}\textbf{range} \equiv 3\ \textbf{downto}\ 0$$

- In the **port map** statement every signal is associated with the respective formal component port by position, in this case according to the following table

component:	a	c	d	y
port map:	a(i)	c(i)	c(i+1)	y(i)

- There is another, more explicit form of the **port map** statement where association of the formal component ports and the instantiated component signals is by names, not by position.

  We can write

```
PORT MAP (a => a(i), c => c(i), d => c(i+1), y => y(i));
```