



دانشگاه اصفهان

دانشکده مهندسی کامپیوتر

درس طراحی کامپیوتری سیستمهای دیجیتال

دستورات Sequential در VHDL

Sequential code

- Code written within the following statements execute sequentially.
 - Process
 - Functions
 - Procedures

Process

- A process is a sequential section of VHDL code.
- It is characterized by the presence of following statements:
 - If Statements
 - Case Statements
 - Null Statements
 - Loop Statements
 - Exit Statements
 - Next Statements
 - While loops
 - For loops

Process (cont..)

- A Process must be installed in the main code, and is executed every time a signal in the sensitivity list changes (or the condition related to WAIT is fulfilled).
- Syntax:

```
[label:] PROCESS [(sensitivity_list)] [IS]  
    [declarative_part]  
BEGIN  
    sequential_statements_part  
END PROCESS [label];
```

Process Statements

Label: **process** (sensitivity_signal_list)

-- constant_declaration

-- variable_declaration

--Subprogram declaration

-- signal declaration are not permitted

here

Begin

--Sequential statements

End process LABEL;

- Used in architectures
- Every process statement within an architecture is executed once at the beginning of simulation, and thereafter, only when a signal in its sensitivity list changes value (i.e., when there is an event on one or more of the signal in the sensitivity list).
- The statements within processes must be sequential statements.

- Variables declared within processes are static. They are initialized only once at the beginning of simulation and retain their values between process activations.
- The other form of process statements as no sensitivity list.

```
PROCESS (clk, rst)
    VARIABLE a, b: INTEGER RANGE 0 TO 255;
    VARIABLE c: BIT_VECTOR(7 DOWNT0 0) := "00001111";
BEGIN
    ...
END PROCESS;
```

If Statements

```
[label:] IF conditions THEN
    assignments;
ELSIF conditions THEN
    assignments;
...
ELSE
    assignments;
END IF [label];
```

```
IF (x<y) THEN
    temp:= "00001111";
ELSIF (x=y AND w='0') THEN
    temp:= "11110000";
ELSE
    temp:=(OTHERS => '0');
END IF;
```

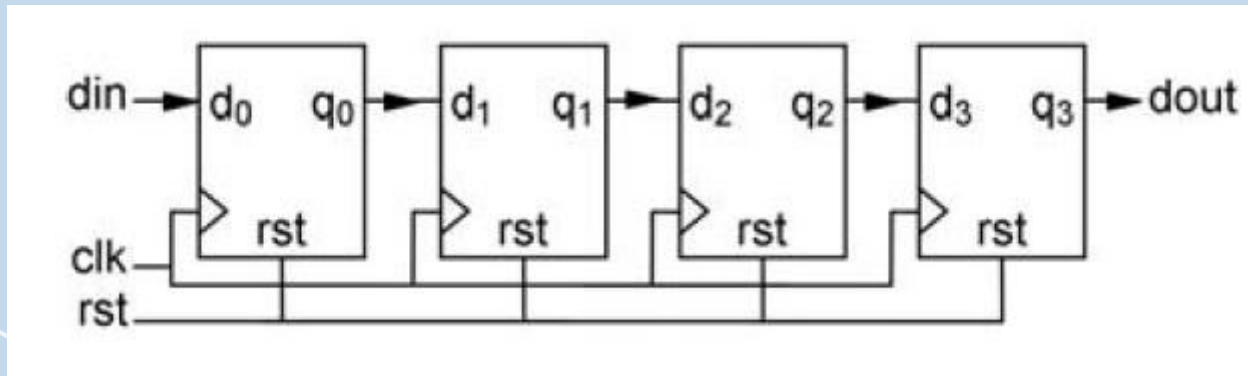

Caution for using if.....else statement

- Avoid using more than three levels of if...else.....end if statements.
- If more than three levels are required, encapsulate the inner nested levels with procedure calls.
- Indent each level of if statement.
- When defining the condition, use parentheses levels of operations on the condition. Group the operations in a logical and readable order.

Basic Counter

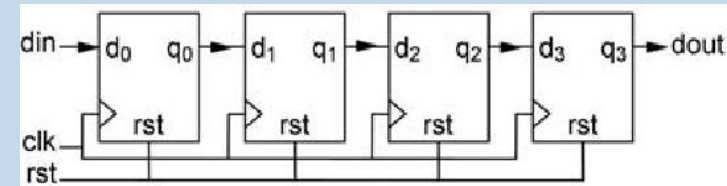
```
1  -----
2  ENTITY counter IS
3      PORT (clk: IN BIT;
4              count: OUT INTEGER RANGE 0 TO 9);
5  END ENTITY;
6  -----
7  ARCHITECTURE counter OF counter IS
8  BEGIN
9      PROCESS(clk)
10         VARIABLE temp: INTEGER RANGE 0 TO 10;
11     BEGIN
12         IF (clk'EVENT AND clk='1') THEN
13             temp := temp + 1;
14             IF (temp=10) THEN
15                 temp := 0;
16             END IF;
17         END IF;
18         count <= temp;
19     END PROCESS;
20 END ARCHITECTURE;
21 -----
```

Shift register

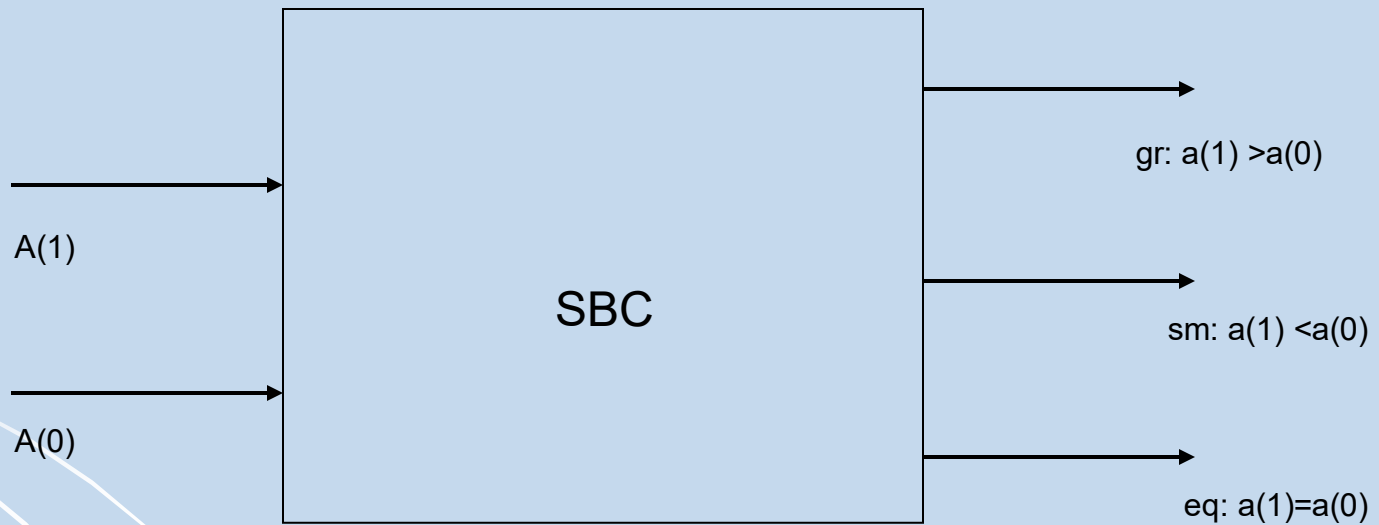


Shift Register

```
1  -----
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  -----
5  ENTITY shift_register IS
6      GENERIC (N: INTEGER := 4); --number of stages
7      PORT (din, clk, rst: IN STD_LOGIC;
8            dout: OUT STD_LOGIC);
9  END ENTITY;
10 -----
11 ARCHITECTURE shift_register OF shift_register IS
12 BEGIN
13     PROCESS (clk, rst)
14         VARIABLE q: STD_LOGIC_VECTOR(0 TO N-1);
15     BEGIN
16         IF (rst='1') THEN
17             q := (OTHERS => '0');
18         ELSIF (clk'EVENT AND clk='1') THEN
19             q := din & q(0 TO N-2);
20         END IF;
21         dout <= q(N-1);
22     END PROCESS;
23 END ARCHITECTURE;
24 -----
```



Single bit comparator



```
1  entity singlebitcomparator is
2      Port ( a : in std_logic_vector(1 downto 0);
3            en: in std_logic;
4            gt : out std_logic;
5            sm : out std_logic;
6            eq : out std_logic);
7  end singlebitcomparator;
8
9  architecture Behavioral of singlebitcomparator is
10 begin
11     process (en,a)
12     begin
13         if (a(1)>a(0)) then
14             gt <= '1'; sm <= '0'; eq <= '0';
15         elsif (a(1) < a(0)) then
16             gt <= '0'; sm <= '1'; eq <= '0';
17         else
18             gt <= '0'; sm <= '0'; eq <= '1';
19         end if;
20     end process;
21 end Behavioral;
```

4x1 Multiplexer

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity mux4_1_if is
5  port (    a: in std_logic_vector(3 downto 0);
6          s: in std_logic_vector(1 downto 0);
7          y: out std_logic
8  );
9  end mux4_1_if;
10
11 architecture mux_behave of mux4_1_if is
12 begin
13 process (s,a)
14     begin
15         if s = "00" then
16             y <= a(0);
17         elsif s = "01" then
18             y <= a(1);
19         elsif s = "10" then
20             y <= a(2);
21         else
22             y <= a(3);
23         end if;
24 end process;
25 end mux_behave;
```

Case statements

A case statement selects for execution one of a number of alternative sequences of statements.

The syntax for a case statement is as follows:

```
[label:] CASE expression IS  
    WHEN value => assignments;  
    WHEN value => assignments;  
    ...  
END CASE;
```


Case Example

```
CASE control IS
```

```
  WHEN "000" => x<=a; y<=b;
```

```
  WHEN "000" | "111" => x<=b; y<= '0';
```

```
  WHEN OTHERS => x<='0'; y<='1';
```

```
END CASE;
```

Like SELECT, CASE too allows the use of multiple values

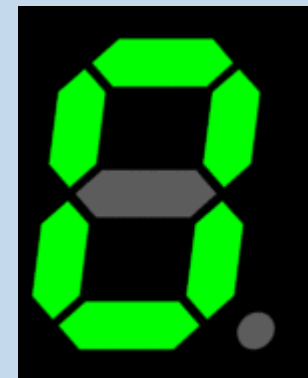
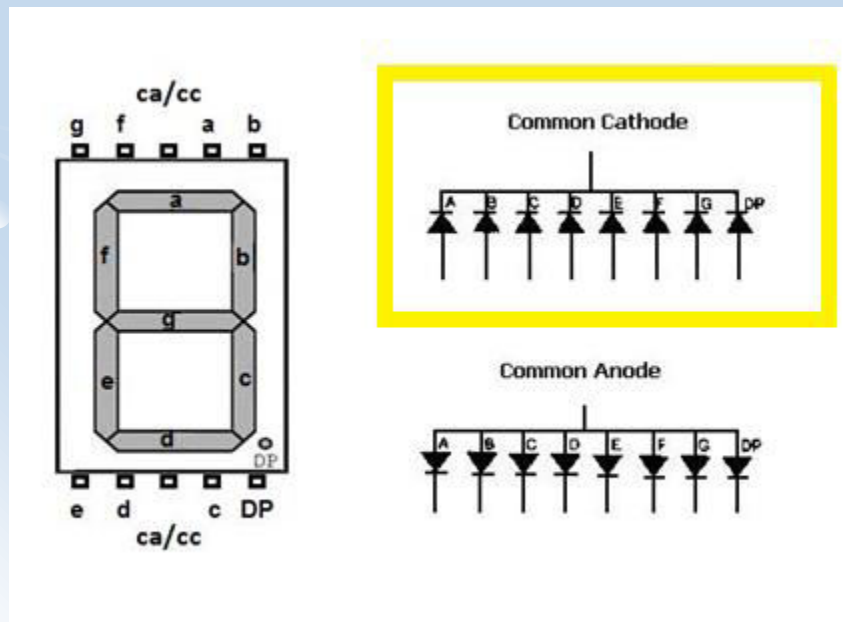
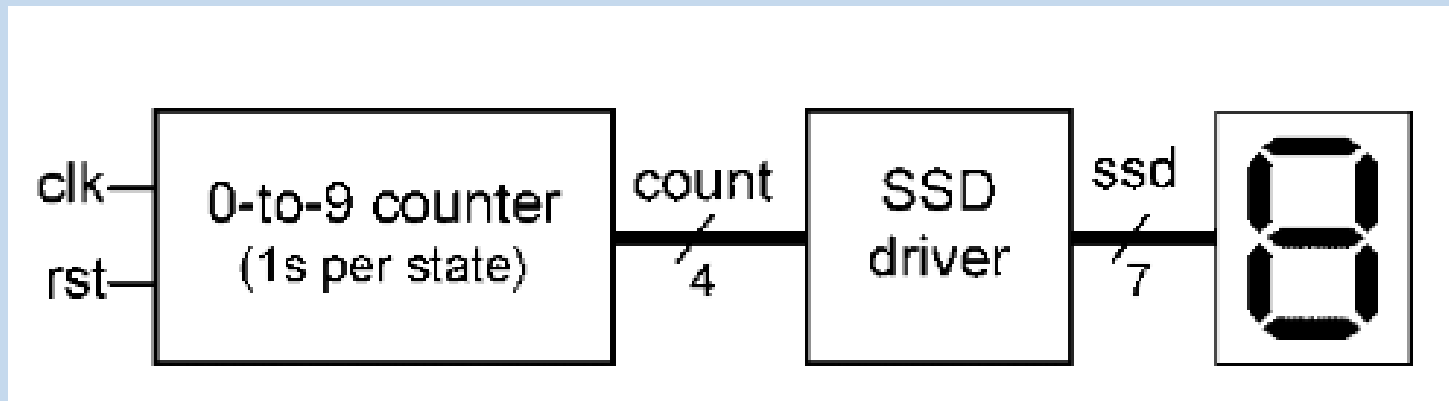
```
WHEN value1 | value2 | ...      --value1 or value2 or ...
```

```
WHEN value1 TO value2          --range (for enumerated types only)
```

Example 4:1 Mux using Case statement

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity mux4_1_case is
5  port  (  a: in std_logic_vector(3 downto 0);
6          s: in std_logic_vector(1 downto 0);
7          y: out std_logic
8  );
9  end mux4_1_case;
10
11 architecture mux_behave of mux4_1_case is
12 begin
13 process (s)
14 begin
15     case s is
16         when "00"      => y <= a(0);
17         when "01"      => y <= a(1);
18         when "10"      => y <= a(2);
19         when others    => y <= a(3);
20     end case;
21 end process;
22 end mux_behave;
```

Slow 0-to-9 Counter with 7Seg



```

1  -----
2  ENTITY slow_counter IS
3      GENERIC (fclk: INTEGER := 50_000_000); --50MHz
4      PORT (clk, rst: IN BIT;
5              ssd: OUT BIT_VECTOR(6 DOWNT0 0));
6  END ENTITY;
7  -----
8  ARCHITECTURE counter OF slow_counter IS
9  BEGIN
10     PROCESS (clk, rst)
11         VARIABLE counter1: NATURAL RANGE 0 TO fclk := 0;
12         VARIABLE counter2: NATURAL RANGE 0 TO 10 := 0;
13     BEGIN
14         -----counter:-----
15         IF (rst='1') THEN
16             counter1 := 0;
17             counter2 := 0;
18         ELSIF (clk'EVENT AND clk='1') THEN
19             counter1 := counter1 + 1;
20             IF (counter1=fclk) THEN
21                 counter1 := 0;
22                 counter2 := counter2 + 1;

```

```

23         IF (counter2=10) THEN
24             counter2 := 0;
25         END IF;
26     END IF;
27 END IF;
28 -----SSD driver:-----
29 CASE counter2 IS

```

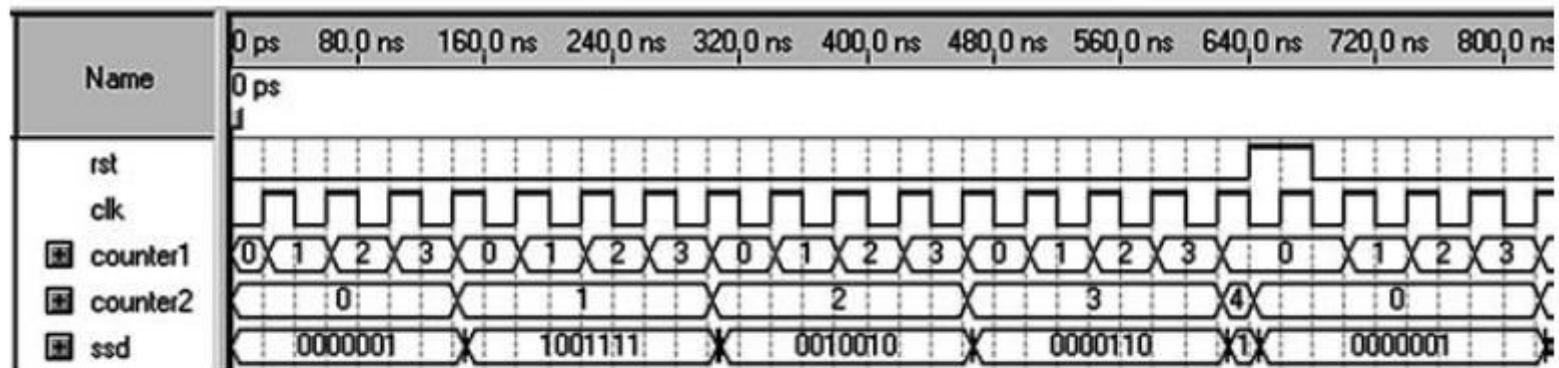


Figure 6.10

Simulation results from the counter of example 6.6.

```

38         WHEN 8 => ssd<="00000000";  --"8" on SSD
39         WHEN 9 => ssd<="0000100";  --"9" on SSD
40         WHEN OTHERS => ssd<="0110000";  --"E"rror
41     END CASE;
42 END PROCESS;
43 END ARCHITECTURE;
44 -----

```

Wait Statement

- WAIT is another sequential statement.
- It is available in three forms:

```
[label:] WAIT UNTIL condition;
```

```
[label:] WAIT ON sensitivity_list;
```

```
[label:] WAIT FOR time_expression;
```

When WAIT is employed, the PROCESS cannot have a sensitivity list.

WAIT UNTIL

- This statement causes the process to hold until the expressed condition is fulfilled.
- two equivalent processes for a DFF with synchronous clear

```
---DFF process with IF:-----  
PROCESS (clk)  
BEGIN  
    IF (clk'EVENT AND clk='1') THEN  
        IF (clr='1') THEN  
            q <= '0';  
        ELSE  
            q <= d;  
        END IF;  
    END IF;  
END PROCESS;
```

```
---DFF process with WAIT UNTIL:-----  
PROCESS  
BEGIN  
    WAIT UNTIL (clk'EVENT AND clk='1');  
    IF (clr='1') THEN  
        q <= '0';  
    ELSE  
        q <= d;  
    END IF;  
END PROCESS;
```

Wait On

- This statement causes the process to hold until any listed signal changes.
- Two equivalent processes for a DFF with synchronous clear

```
---DFF process with IF:-----  
PROCESS (clk)  
BEGIN  
    IF (clk'EVENT AND clk='1') THEN  
        IF (clr='1') THEN  
            q <= '0';  
        ELSE  
            q <= d;  
        END IF;  
    END IF;  
END PROCESS;
```

```
---DFF process with WAIT ON:-----  
PROCESS  
BEGIN  
    IF (clk'EVENT AND clk='1') THEN  
        IF (clr='1') THEN  
            q <= '0';  
        ELSE  
            q <= d;  
        END IF;  
    END IF;  
    WAIT ON clk;  
END PROCESS;
```


WAIT FOR

- This statement is for simulations
- The declaration below creates a clock waveform with period 80 ns


```
WAIT FOR 40ns;  
clk <= NOT clk;
```

Loop

- Loop statements are used to execute a sequence of statements zero or more times.
- Loop is intended exclusively for sequential code.
- For/loop : The loop is repeated a fixed number of times.

```
[label:] FOR identifier IN range LOOP  
    sequential_statements  
END LOOP [label];
```

Loop Statement

- Three forms of loop statements:
 - The simple loop
 - The while loop
 - The for loop
- 

Unconditional LOOP:

```
[label:] LOOP  
    sequential_statements  
END LOOP [label];
```

```
LOOP  
    WAIT UNTIL clk='1';  
    count := count + 1;  
END LOOP;
```

The while loop

```
[label:] WHILE condition LOOP  
    sequential_statements  
END LOOP [label];
```

```
WHILE (i<10) LOOP  
    WAIT UNTIL clk'EVENT AND clk='1';  
    ...  
END LOOP;
```

The for loop

- The loop parameter type for the **for** iteration loop scheme is the base type of the discrete range, and is not explicitly defined as a type. The type is implicitly defined from the range.
- Syntax:

```
[label:] FOR identifier IN range LOOP  
    sequential_statements  
END LOOP [label];
```

Example of For/loop

```
FOR i IN 0 TO 5 LOOP  
    x(i) <= a(i) AND b(5-i);  
    y(0, i) <= c(i);  
END LOOP;
```

NOTE: Range must be static.

Exit Statement

Exit statement: is a sequential statement that can be used only inside a loop

Syntax:

exit [*loop-label*] [**when condition**];

Example:

SUM:= 1; J := 0;

L3: **loop** J:= J + 21; SUM := SUM * 10;

if SUM > 100 **then exit** L3;

--only **exit** is also possible

--if no loop label is specified, the innermost loop is

end if;

end loop L3;

exited

Loop with Exit Statement

```
[loop_label:] [FOR identifier IN range] LOOP  
    ...  
    [exit_label:] EXIT [loop_label] [WHEN condition];  
    ...  
END LOOP [loop_label];
```

Next Statement

Next Statement: skipping the remaining statements in the current iteration of the specified loop;

- execution resumes with the first statement in the next iteration of this loop, if one exists
- can be used only inside a loop
- sequential statement
- if no loop label is specified, the innermost loop is assumed

Syntax

next [*loop-label*] [**When** *condition*];

Loop with Next Statement

```
[loop_label:] [FOR identifier IN range] LOOP  
    ...  
    [next_label:] NEXT [loop_label] [WHEN condition];  
    ...  
END LOOP [loop_label];
```

Next Statement (Example-1)

Next statement can also cause an inner loop to be exited

example:

```
L4: for K in 10 downto 1 loop
```

```
.....
```

```
  L5: loop .....
```

```
    next L4 when WR_Done := '1';
```

```
.....
```

```
  end loop L5;
```

```
.....
```

```
end loop L4;
```

Example (exit and Next)

```
FOR i IN data'RANGE LOOP
  CASE data(i) IS
    WHEN '0' => count:=count+1;
    WHEN OTHERS => EXIT;
  END CASE;
END LOOP;
```

Example (next)

```
FOR i IN 0 TO 15 LOOP  
    NEXT WHEN i=skip;  
    ...  
END LOOP;
```

Carry-Ripple Adder

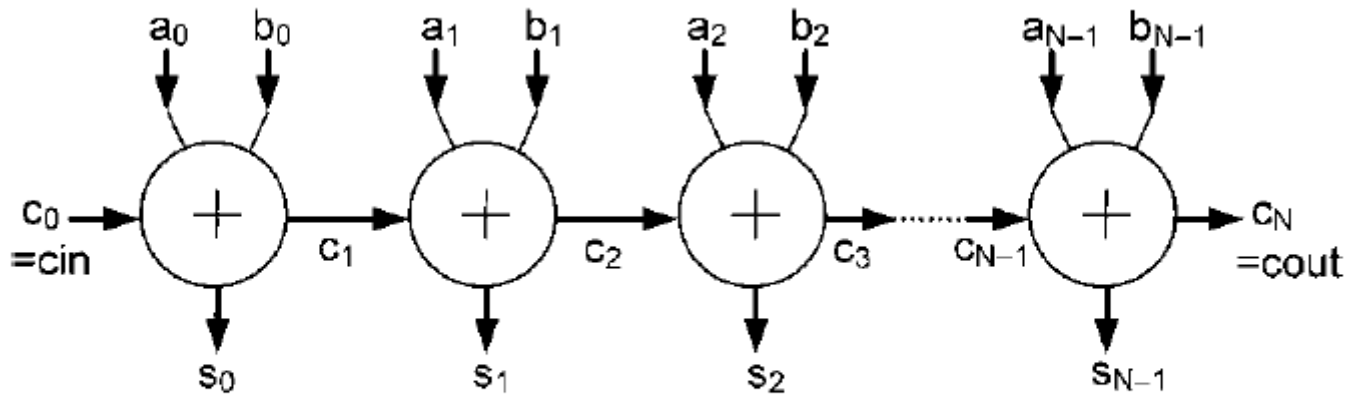


Figure 6.6
Carry-ripple adder.

```

1  -----
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  -----
5  ENTITY carry_ripple_adder IS
6      GENERIC (N : INTEGER := 8); --number of bits
7      PORT (a, b: IN STD_LOGIC_VECTOR(N-1 DOWNT0 0);
8           cin: IN STD_LOGIC;
9           s: OUT STD LOGIC VECTOR(N-1 DOWNT0 0));

```

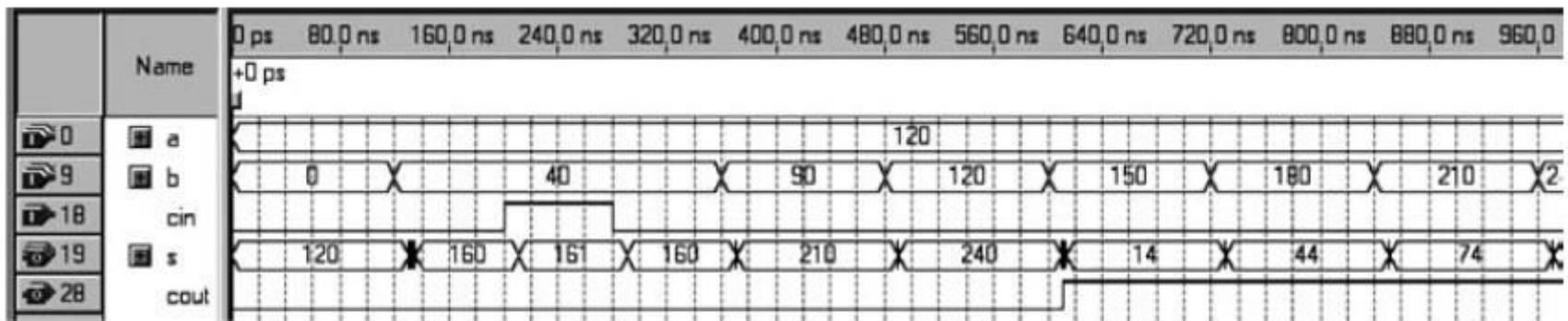


Figure 6.7

Simulation results from the carry-ripple adder of example 6.4.

```

19      FOR i IN 0 TO N-1 LOOP
20          s(i) <= a(i) XOR b(i) XOR c(i);
21          c(i+1) := (a(i) AND b(i)) OR (a(i) AND c(i)) OR
22                  (b(i) AND c(i));
23      END LOOP;
24      cout <= c(N);
25  END PROCESS;
26 END ARCHITECTURE;
27 -----

```


Leading Zeros

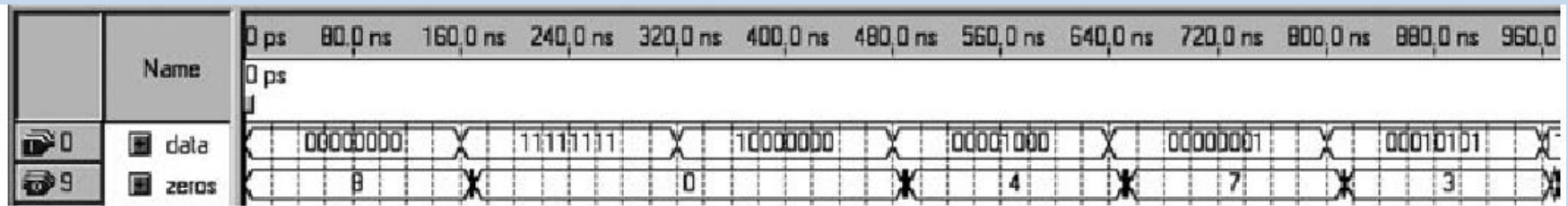


Figure 6.8
Simulation results from the leading-zeros counter of example 6.5.

```

1  -----
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  -----
5  ENTITY leading_zeros IS
6      PORT (data: IN STD_LOGIC_VECTOR (7 DOWNT0 0);
7            zeros: OUT INTEGER RANGE 0 TO 8);
8  END ENTITY;
9  -----
10 ARCHITECTURE behavior OF leading_zeros IS
11 BEGIN
12     PROCESS (data)
13         VARIABLE count: INTEGER RANGE 0 TO 8;
14     BEGIN
15         count := 0;
16         FOR i IN data'RANGE LOOP
17             CASE data(i) IS
18                 WHEN '0' => count := count + 1;
19                 WHEN OTHERS => EXIT;
20             END CASE;
21         END LOOP;
22         zeros <= count;
23     END PROCESS;
24 END ARCHITECTURE;
25 -----

```