VHDL, HOW IT WORKS

Simulating the passage of time in discrete steps is called discrete time simulation.
VHDL uses discrete time event driven simulation, that is if a signal value changes, that change is considered an event that has to be processed to find out the effect of this change on the other signals. Events occur at discrete times and signals are updated during next discrete time intervals.
Once an events occur then a list of events to be changed are updated accordingly and are changed in rounds of discrete time. So in each round the list that are newly scheduled events are processed and a new list of events are generated or scheduled. So simulation goes in rounds of discrete time until all lists contents are processed and there is no more scheduled events. Each signal assignment is processed once at the beginning of simulation.

CONCORDIA **Sequential VHDL**

Discrete Time Simulation and EVEVTs

- EVENTs are queued up.
- EVENTs are ordered in time
- Simulation moves from "EVENT" to "EVENT"
- The events queue is open ended as new events occur they are scheduled and old events are removed from the queue and saved in history file
- Addition and deletion of "EVENTs" is possible as a result of current event
- Each "SIGNAL" has a driver that maintains time and value of recent "EVENT"
- You may check the time, value and event time by their attributes

Signal_1'last_event
Signal_1'last value
Signal_1'last active

CONCORDIA

A δ delay is a small delay that separates events occurring in the same simulation cycle but within the simulation time to represent events occurring in 0 time.
Signals are data objects that can be assigned a time series of "value, time" for the data object.
Signal values are always scheduled in a future time.

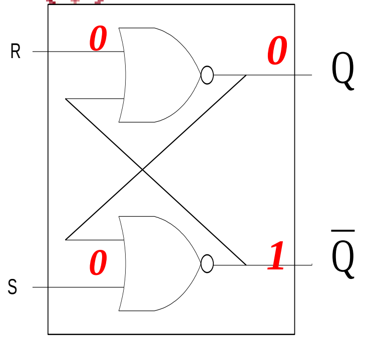
Time
Value

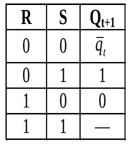
Signal_1 <= a and b after 5ns;

This statement directs the driver of Signal_1 to generate a "Value, Time" pair to be scheduled at 5 ns.

S2 <= '1' after 5 ns;
S3 <= not Signal_1 after 6ns;

The events queue keeps track of scheduled signal changes

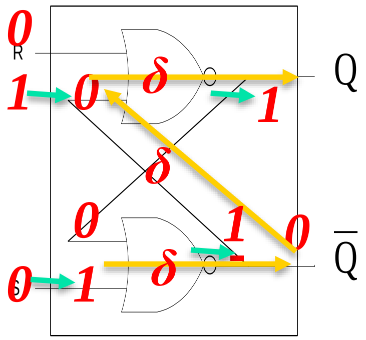


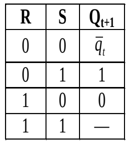


R	S	Q_{t+1}
0	0	$\bar{q}_t$
0	1	1
1	0	0
1	1	—

process
(R,S,Q,Q')
begin
Q <= R nor Q'
Q'<=S nor Q
end process

R=S=0 Q=0. Q'=1



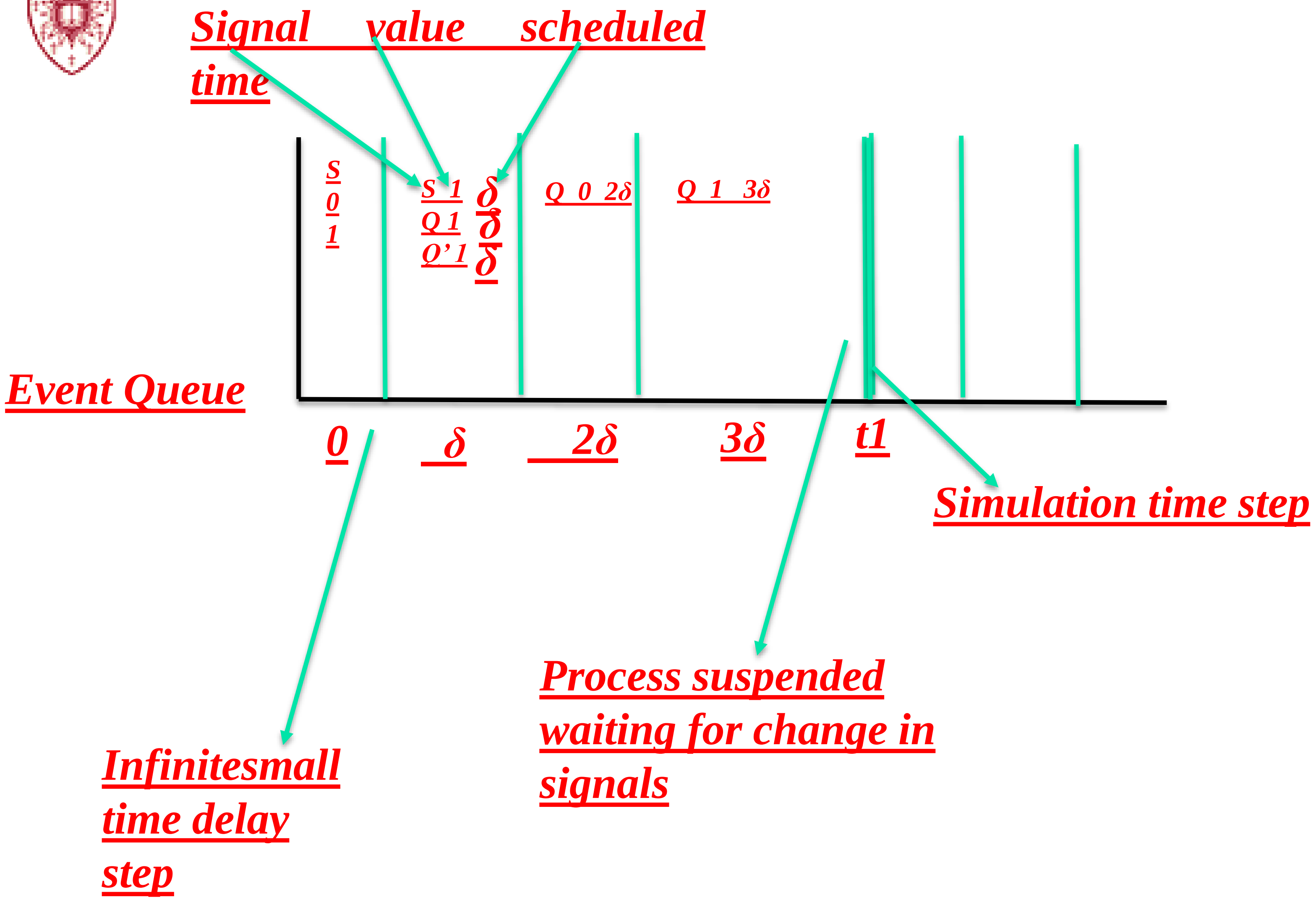


R	S	Q_{t+1}
0	0	$\bar{q}_t$
0	1	1
1	0	0
1	1	—

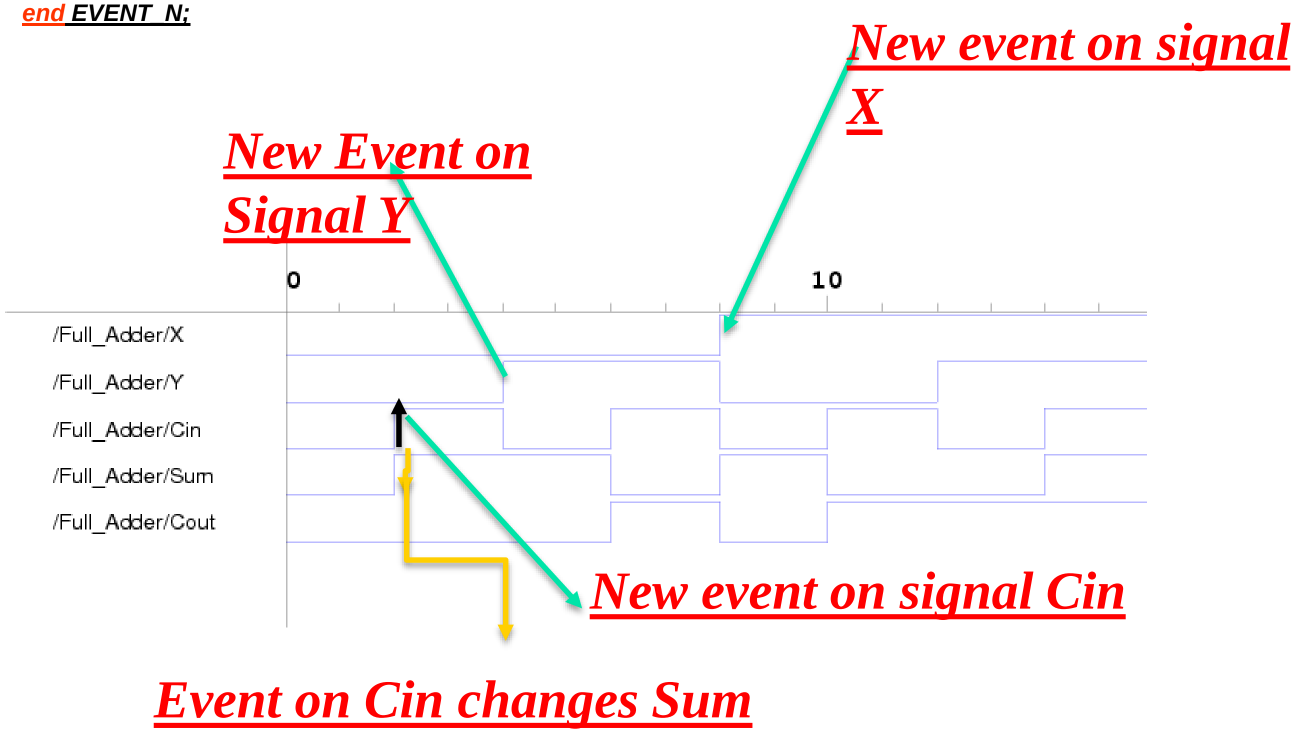
	0	δ	→2δ	3δ
R	0	0	0	0
S	0 → 1	1	1	1
Q	0	1	0	1
Q′	1	0	0	0

R=0 S=0 Q=0 Q'=1 Initial Values
ThenS is set to 1 ***With zero time delay***

CONCORDIA


Signal value scheduled time
S 0 1
S 1 δ
Q 1 δ
Q' 1 δ
Q 0 2δ
Q 1 3δ
Event Queue
0
δ
2δ
3δ
t1
Simulation time step
Infinitesmall time delay step
Process suspended waiting for change in signals

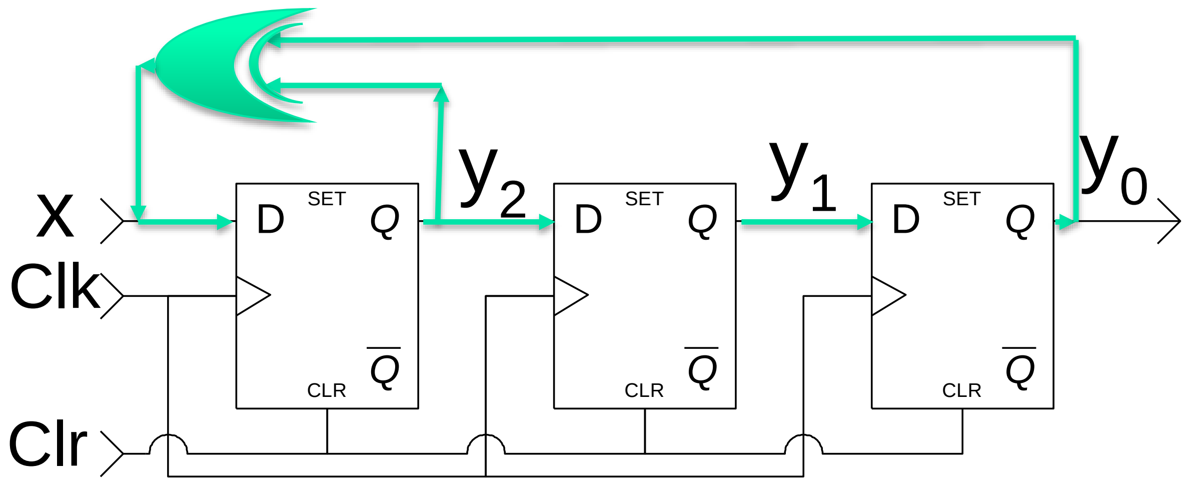
```
library ieee;
use ieee.std_logic_1164.all;
entity Full_Adder is
 -- generic (TS : TIME := 0.11 ns; TC : TIME := 0.1 ns);
        port (X, Y, Cin: in std_logic; Cout, Sum: out  std_logic);
 end Full_Adder;
 architecture EVENTS_N of Full_Adder is
begin
   Sum <=  X xor Y xor Cin after 0.11 ns ;
   Cout <= (X and Y) or (X and Cin) or (Y and Cin) after 0.1 ns;
end EVENT_N;
```



At the start, assume To=0 time, the processes that have a sensitivity list are activated, Each signal is assigned a driver and their value and their future time of activation is recorded.
when all processes are evaluated and suspended then the simulation time move to T1. The next time step.
Simulation stops when Tc, current time is equal TIME'HIGH.

During simulation each active signal is updated and new "value , time" is calculated. all other signals effected are also updated.
New updates will be then effected during the next simulation cycle.
Events occurring as a results of these changes are further affected in the next simulation cycle until no further events are scheduled or time has reached TIME'HIGH
(When using 64-bit signed integers the maximum value is 2^{63}-1 = 9223372036854775807 = 9.22x10^{18}. This indicates that time values within 1 femtosecond to 3 hours range can be covered, this is sufficient for majority of applications.

CONCORDIA



Events are <u>infinite,</u> maybe stopped by conditional statement statement or Tc =Time HIGH

Sequential VHDL

In this mode such as process clause, the assignments are carried out sequentially. This means that the assignments are executed in order of appearance one after the other. Therefore the order in which they appear is important.

What happens if new assignments are added when previous transactions are not yet carried out. In such situations the driver of the signal will look at all the transactions placed on the signal and decides if the new transaction that is scheduled as a new event overwrites other events or will it be queued up.

Example:

```
architecture Implementation of Sequential_event is
signal connect_1 STD-logic  <= 'U';
begin
   process
   begin
         connect_1 <= '1' after 14 ns;  -- line 1
         connect_1 <= '0' after 3 ns ;   --line 2
         wait;
   end process;
end Implementation;          -- In this case line 2 will overwrite line 1
```

CONCORDIA

Sequential VHDL

continued

Process, Procedures, Functions are sequential VHDL.
All time based behavior in VHDL is defined in terms of the process statement.
Process statements is made up of two parts

The Declarative Part

Functions

Procedures

Type, subtype

Constant, Variable

File,

Alias, Attribute,

Use clause

group

The Statement Part

Wait, if, case, loop,

Variable, & signal assignment

Exit, return, next, null

Procedure call,

Assertion

report

Process *continued*

- Process statement defines the timing behavior in VHDL.

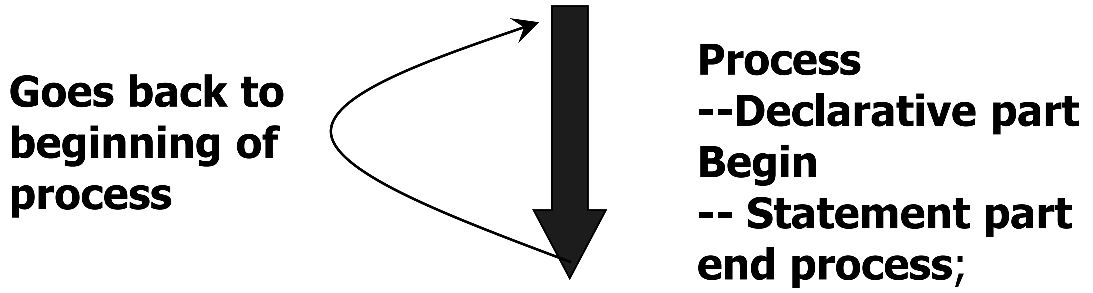
- A process is composed of two main parts:

  ➢ Declarative part:

  Procedure, function, type, subtype, constant, variable, file, alias, attribute, use clause, group.

  ➢ Statement part :

  Wait, variable / signal assignment, if, exit, procedure call / return, case, assertion, report, loop, next, null.



```
Process
--Declarative part
Begin
-- Statement part
end process;
```

Process…. notes

- The execution of the process follows the same pattern as a **hardware execution**. Process starts at the beginning of the simulation by executing series of signal transitions brought in by the series of inputs. The process stops itself at the end of the simulation.

- Process starts with the declaration part and then sets up the initial values. The style of the process execution is sequential i.e statements are executed top to bottom. After the execution of the last line of the process, the control shifts back to the first line in the process. So, the process is virtually like a infinite *do loop*.

- Process is **activated** through change in input and then the process **reacts** by producing an output. Process can also be activated by the change in its sensitivity list. Sensitivity list is a very useful statement for defining process activation or suspension based on the events occurring on signals on the sensitivity list.

CONCORDIA

Process NOTES

All Processes in the architecture of an entity run concurrently and are active at all times.

All assignments within the body of a process run sequentially.

A process gets executed when an event occurs on one of its signals on the right hand side of signal assignment (Sensitive to these changes).

A process begins with the reserved word *process* and ends with the reserved word *end process*.

Each process has a declarative and the statement part.

Only variables, files, or constant objects can be declared within the declarative part.

Signals and Constants declared in the architecture that contains the process are visible within the body of the process.

The statement part of the process is always active and is running at time zero unless suspended by a *wait* statement (implicit or explicit) , a process runs for ever.

Only sequential statement (if, Loop, case..) are allowed within the statement part of a process.

Process NOTES

The process execution is different from a procedure. With the procedure the execution stops once the statements are executed. With process it goes back to the beginning of the statements and repeats itself.

The process can be conditionally stopped or suspended by its sensitivity list.

The process is activated whenever an event occurs on its sensitivity list and whenever the last statement is executed then the process gets suspended (still alive) waiting for a change in one or more of the sensitivity list.

Each Process is activated at least once at the beginning of the process, independent of the sensitivity list.

Process

- Process example

Optional

Process_label : ***process*** (sensivity list…..) ***is***

-------------   Declaration Part
----

begin

--------------   Statement Part

Optional

end process Process_label;

Wait statements

***Process*:**

The most common and useful parts in a process is the **wait** statements:

[I] ***wait until*** clk = 1 ; -- waits for clk = 1

[II] ***process***(x,y) --Process with x, y in its sensitivity list, ***process****(clk,reset)*

[III] ***wait on*** x,y ; -- Sensitivity list in [II] can not be used with III .

[IV] ***wait for*** 10 ns ;

[IV] ***wait for*** 0 unit time ;

[V] ***wait ;***

The wait statement is used to model delays, handshaking and dependencies.

[I] --Suspends when condition is satisfied.

[II] --The process is suspended until an event on sensitivity list occurs.

[III] --process is suspended until an event on x,y occurs.

[IV] -- wait for the time period specified, when time is 0, then suspends process for δ

[V] Suspend the process for ever.

CONCORDIA

Process....

Example taken from ASIC, By J.S. Smith

All processes are executed at the same time / Concurrent

library to print

Delay of 10 ns

Stop after 500 ns

```
library STD;
use STD.TEXTIO.all ;
entity counter_8 is
--------------
end counter_8;
architecture behavior of
counter_8 is
signal clk: Bit := '0';
signal counter: Integer := 0 ;
begin
Proc1_clk: process
        begin
          wait  for 10 ns;
          clk <= not (clk);
        if (now > 500 ns) then
wait;
          end if;
            end process
Proc1_clk;
```

Counter increments on negative edges of clock and then resets back to 0

```
Proc2_counter: process
        begin
            wait until (clk ='0');
            if (counter =7) then
            count <= 0;
            else
            count <= count + 1;
            end if;
            end process Proc2_counter;
Proc3_print: process
          variable L: Line;
          begin
          write (L, now);
          write (L, string'("count= "));
            write ( count);
            writeline (output,L);
             wait for 1ns;
                end process Proc3_print;
```

Wait

Example taken from reference 3

Example 1

```
process (x)
begin
a1 <= not x ;
end process;
```

Example 2

```
process
begin
a2 <= not x;
wait on x;
end process;
```

Example 3

```
process
begin
wait on x
a3 <= not x ;
end process;
```

Example 4

```
process
begin
wait      until
x='1';
a4 <= not x;
end process;
```

Example 5

```
process
begin
a5 <= not x ;
wait until x='1' for
10 ns;
end process;
```

Wait

Example Ref. 3

Example 1

```
process (x)
begin
a1 <= not x
;
end process;
```

Example 2

```
process
begin
a2 <= not x;
wait on x;
end
process;
```

Example 3

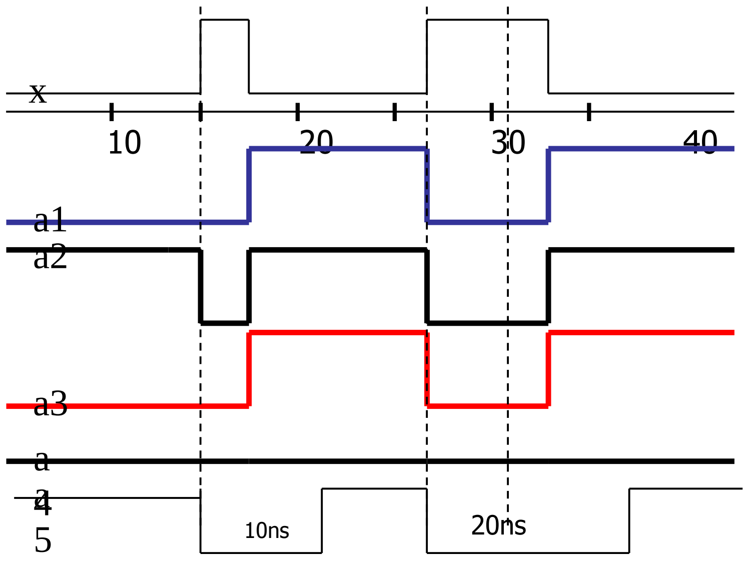
```
process
begin
wait on x
a3 <= not x ;
end process;
```

Example 4

```
process
begin
wait   until
x='1';
a4 <= not x;
end process;
```

Example

```
process
begin
a5 <= not x ;
wait  until   x='1'
for 10 ns;
end process;
```

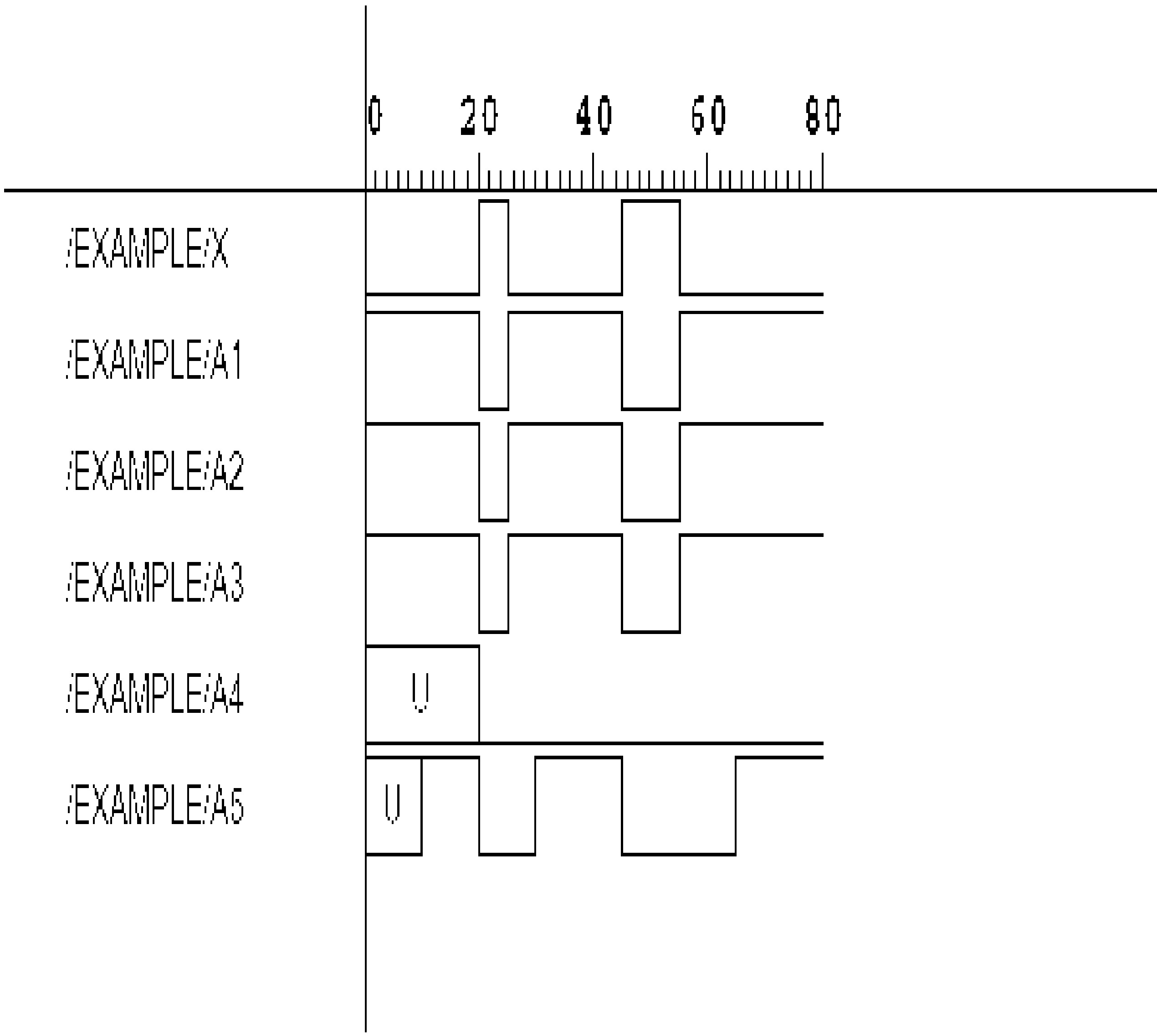


CONCORDIA

Example of Process

Taken from reference 3

```
EX1: process(X)
        begin
        A1<=not X;
end process;
EX2:  process
        begin
        A2<=not X;
        wait on X;
end process;
EX3:  process
        begin
        wait on X;
        A3<=not X;
end process;
EX4:  process
        begin
        wait until X='1
        A4<=not X;
end process;
EX5:  process
        begin
        A5<=not X;
        wait until X='1
for 10 ns;
end process;
end ALGORITHM;
```



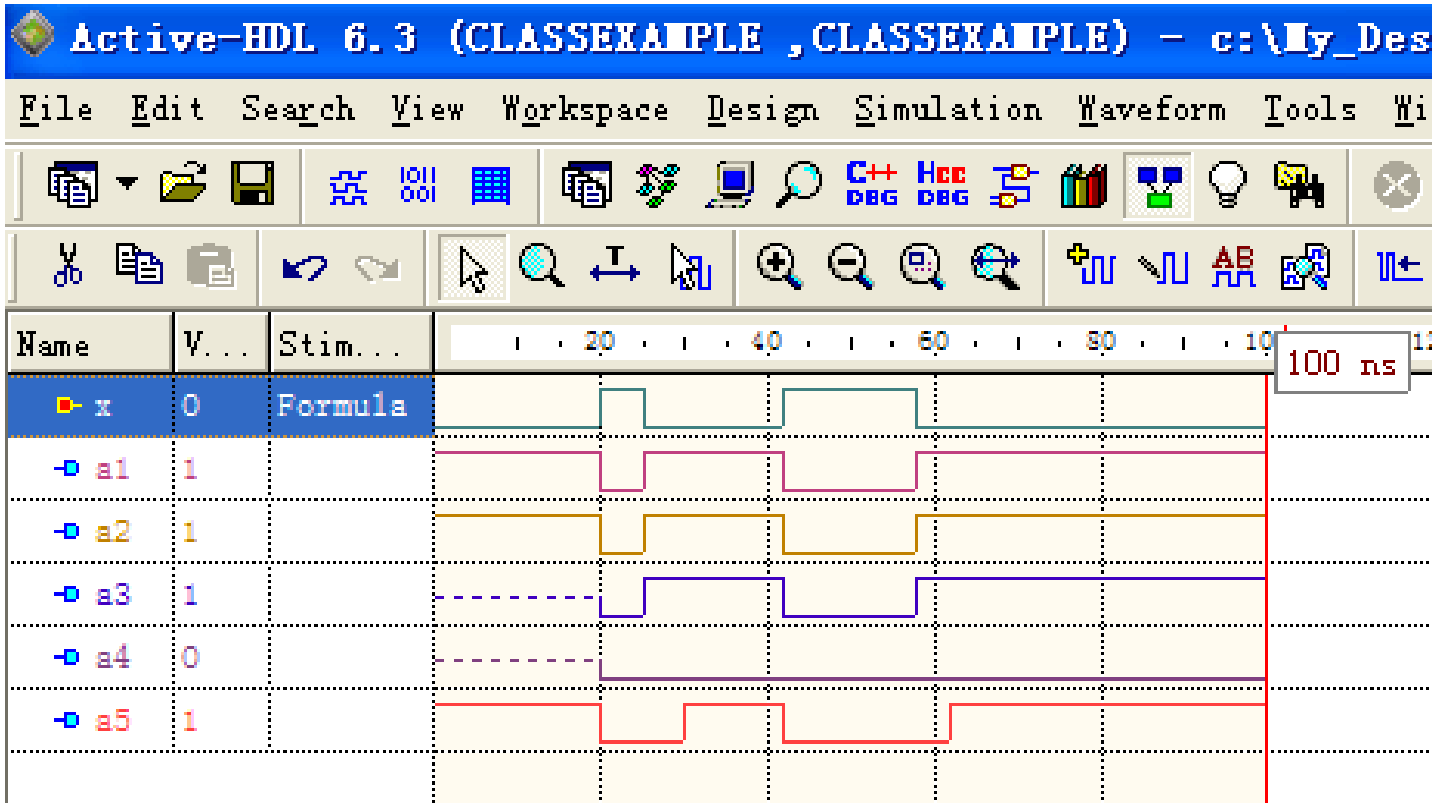
CONCORDIA

```
library IEEE;
 use IEEE.STD_LOGIC_1164.all;
 use IEEE.std_logic_unsigned.all;
 entity waitexample is
 port ( x:in std_logic;
a1,a2,a3,a4,a5: out std_logic);
 end waitexample;
```

```
. architecture behav of waitexample is
 begin
 p1:process(x)
 begin
 a1<= not x;
end process;

 p2: process
 begin
 a2<= not x;
 wait on x;
 end process;

 p3:process
 begin
 wait on x ;
a3<= not x;
 end process;
 p4:process
 begin
 wait until x='1';
 a4<= not x;
 end process;
 p5: process
 begin
 a5<= not x;
wait until x='1' for 10ns;
 end process;
 end behav;
```


Active-HDL 6.3 (CLASSEXAMPLE ,CLASSEXAMPLE) - c:\My_Des
File Edit Search View Workspace Design Simulation Waveform Tools Wi
Name V... Stim...
20 40 60 80 10
100 ns
x 0 Formula
a1 1
a2 1
a3 1
a4 0
a5 1

CONCORDIA

Example (wait for) Waveform Generator

```
library ieee;
use ieee.std_logic_1164.all;
entity example is
port(A: out std_logic);
end example;
architecture ALGORITHM of example is
signal X: std_logic;
begin
STIMULATOR : process
        begin
        X <= '0';
        wait for 20 ns;
        X <= '1';
        wait for 5 ns;
        X <= '0';
        wait for 20 ns;
        X <= '1';
        wait for 10 ns;
        X <= '0';
        wait for 20 ns;
end process;
```

CONCORDIA

Sequential VHDL….

Loop:

[I] **For loop:** *for* **j** *in* **0 to 6** *loop*

[II] **While loop:** *while* **j < 5** *loop*

Case:

```
[I] case S is
      when '0' => C <= X;
      when '1' => C <= Y
```

If statement:

```
 if    x= "01" then  y <="01" ;
elsif x="11" then    y <= "11" ; -- Can have numbers of elsif statements
else y <="10" ;
end if;
```

Generate:

for **generate:** *for* **j** *in* **1** *to* **n** *generate*

if **generate** *if* **j=1** *generate*

Generate Statement

- Generate statement example

for i ***in*** m ***downto*** n ***generate***
-------------	Statement Part
----
end generate ;

-- VHDL 93 should contain a declarative part and begin

for i ***in*** m ***downto*** n ***generate***
-------------	Declaration Part
Begin
-------------	Statement Part
end generate ;

Generate Statement….

The *Generate* statement as used in VHDL provides a powerful ability to describe a regular or slightly irregular structures by automatic component instantiation generation instead of manually writing each component instantiation.

There are two kinds of generate statements:
- Iteration
- Conditional

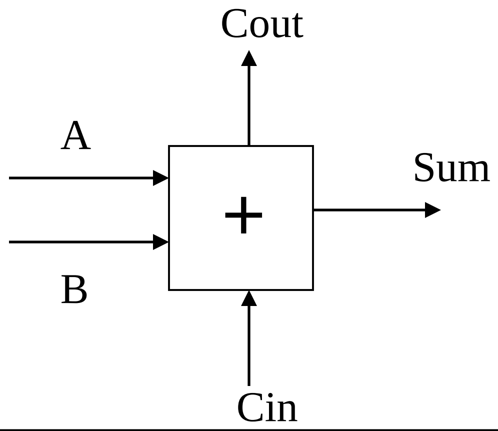
The iteration statement is also known as *for* generate statement.

The conditional statement is also known as the *if* generate statement.

Full Adder

```
entity full_adder is
generic (T1: time := 0.11 ns; T2 : time := 0.1 ns);
port (A, B, Cin : in BIT; Cout, Sum : out BIT);
end full_adder ;
architecture behave of full_adder is
begin
Sum  <=  A xor B xor Cin after T1;
Cout <= (A and B) or (A and Cin) or (B and Cin) after T2;
end behave;
```

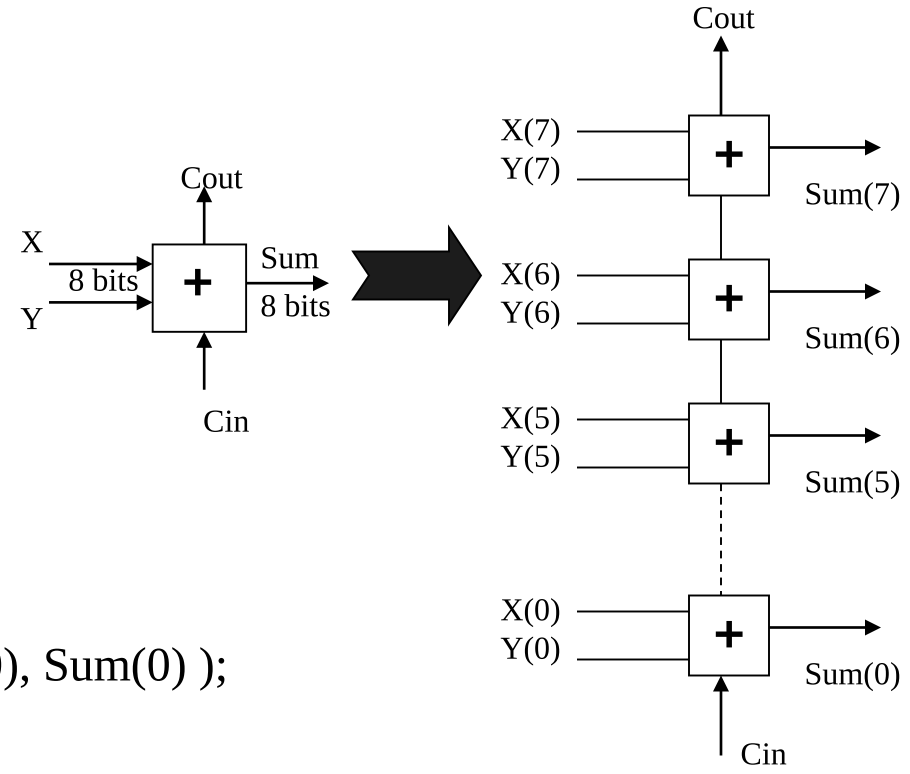
TIMINGS;
T1 (Input → Sum) = 0.11 ns
T2 (Input → Cout) = 0.1 ns



CONCORDIA

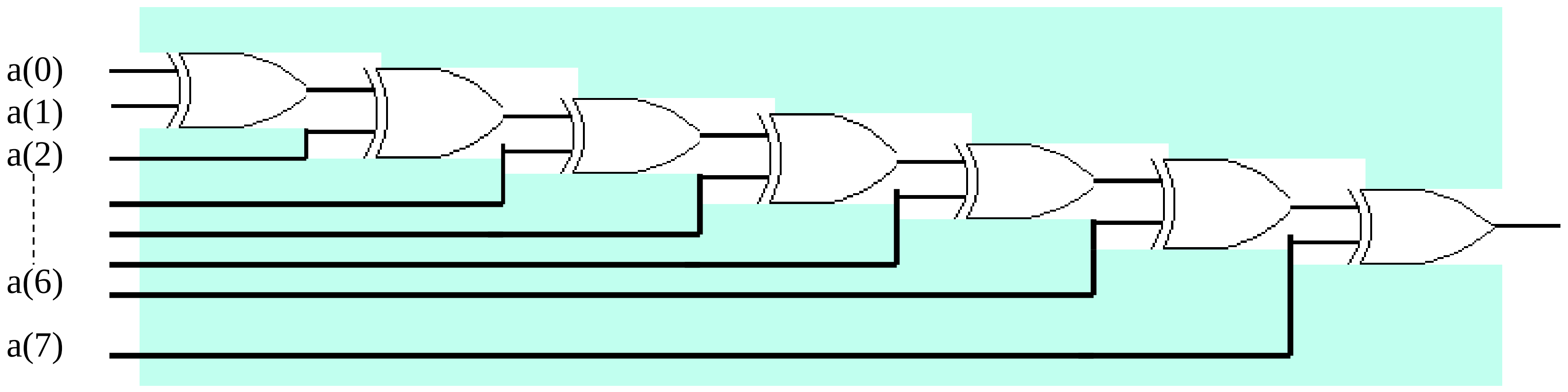
Full Adder...

```
entity full_adder8 is
port (X,Y : in BIT_VECTOR (7 downto 0);
        Cin: in BIT; Cout: out BIT;
        Sum : out BIT_VECTOR (7 downto 0) );
end full_adder8 ;
architecture structure of full_adder8 is
component full_adder
port (A, B, Cin : in BIT; Cout, Sum : out BIT);
end component ;
signal D: BIT_VECTOR (7 downto 0);
begin
  Levels: for i in 7 downto 0 generate
  Lowbit: if i=0 generate
  FA : full_adder port map (X(0), Y(0), Cin, D(0), Sum(0) );
  end generate ;
             Otherbits : if i /= 0 generate
             FA : full_adder port map (X(i), Y(i), D(i-1), D(i), Sum(i) );
             end generate ;
   end generate;
   end structure;
```



CONCORDIA

Parity Generator



```
entity even_parity is
port(a: in BIT_VECTOR (7 downto 0)
        out1: out BIT ) ;
end even_parity;
architecture structural of even_parity is
signal sig1: BIT_VECTOR (1 to 6);
begin
for i in 0  to 6 generate
if i=0 generate -- continued on the right
```

```
 sig1 <= a(i) xor a(i+1);
 end generate;   -- i=0 case
if ( i >=1 and i <= 5) generate
sig1(i+1) <= sig1(i) xor a(i+1);
end generate;   -- 1< i <5 case
if i=6 generate
out1 <= sig1(i) xor a (i+1);
end generate;   -- i=6 case
end generate;  end structural;
```

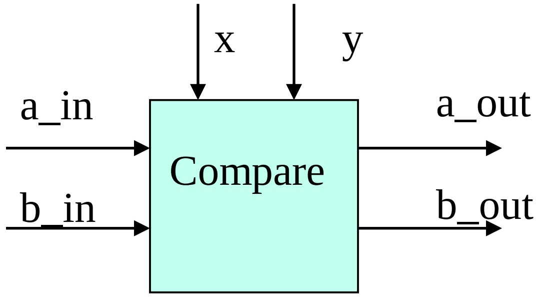
CONCORDIA

Comparator (bit by bit)

```
entity compare is
port(x,y, a_in, b_in: in BIT ;
        a_out, b_out: out BIT ) ;
end compare;
architecture behave of  compare is
begin
a_out <= ( ( (not y) and x  and b_in )  or a_in) ;   -- By K-maps Optimization
b_out <= ( (y and (not a) and (not a_in) ) or b_in) ;-- By K-maps Optimization
end behavior;
```

Truth Table (bit by bit)

a	b	
0	0	x=y
0	1	x<y
1	0	x>y
1	1	Don't Care



CONCORDIA

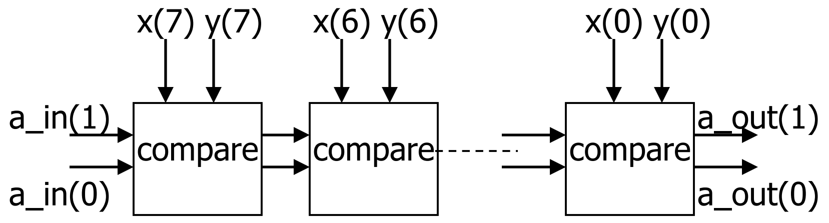
Comparator (8-bit)

```
entity compare_8 is
port(x,y: in BIT_VECTOR (7 downto 0) ;
a_in: in BIT_VECTOR (1 downto 0);
a_out: out BIT_VECTOR (1 downto 0) );
end compare_8;
architecture behave of  compare_8 is
begin
component compare is
port(x,y, a_in, b_in: in BIT ;
        a_out, b_out: out BIT ) ;
end component;
signal s1, s2: BIT_VECTOR(7 downto 1);
begin
for i in 7 downto 0 generate
if (i=7) generate
comp7: compare port map (x(i), y(i), a_in(1),
                                        a_in(0), s1(i), s2(i) );
end generate; -- First bit case
```

```
if (i<= 6 and i>= 1)  generate
compx: compare port map (x(i), y(i)
     s1(i+1), s2(i+1) s1(i), s2(i) );

if (i=0) generate
comp0: port map (x(i), y(i),s1(i+1)
             s2(i+1), a_out(1), a_out(0);

end generate; -- Normal Case
end generate; -- End Case
end behave;
```



CONCORDIA

Clk Functions

Style 1

```
           -- signal clk: std_logic :='0';

process (clk)
begin
if clk'event and clk ='1' then
a<=x;
end if ;  -- Not valid for asynchronous reset
end process;
```

Style 2

```
process
begin
if clk ='1' then
a <=x;
end if;
end process;
```

Style 3

Not supported by Synopsys

```
process (clk)
begin
if clk'event and clk ='1'
   and clk'last_value ='0' then
   a <=x;
end if ;
end process;
```

Activated by '0' to '1' event on clk

Style 4

```
process (clk)
begin
wait until  prising(clk);
a <=x;
end process;
```

Use rising_edge() and falling_edge() functions instead of (clk'event and clk='1') statements in your designs.

Prising (clk) and Pfalling(clk)

Signal vs Variable

Scenario 1

```
process
begin          -- a1,a2 defined as signals
wait for 10 ns;
a1<= a1 + 1;
a2<= a1 + 1;
end process;
```

Scenario 2

```
process
begin
variable a1, a2: integer ;
wait for 10 ns;
a1 := a1 + 1;
a2 := a1 + 1;
end process;
```

	signal		variable	
Time	**a1**	**a2**	**a1**	**a2**
0	**0**	**0**	**0**	**0**
10	**0**	**0**	**1**	**2**
10 + δ	**1**	**1**	**1**	**2**
20	**1**	**1**	**2**	**3**
20 + δ	**2**	**2**	**2**	**3**
30	**2**	**2**	**3**	**4**
30 + δ	**3**	**3**	**3**	**4**

Shift / Rotate Operators

– Predefined for one-dimensional array of type Bit or Boolean

- ***SLL*** , ← ***SRL.*** →

– Fill in type' LEFT or type'RIGHT ('0'or False)

- ***SLA*** ←

– Fill in right most bit

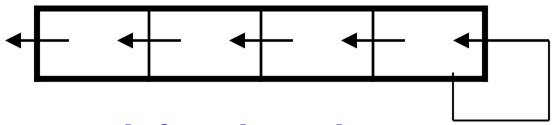
- ***SRA .*** →

– Sign extension

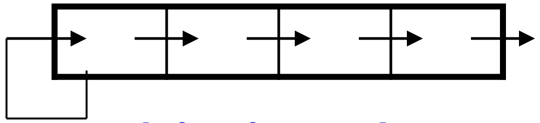
- ***ROL*** , ⤵ ***ROR*** ⤴

Shift / Rotate Operators....

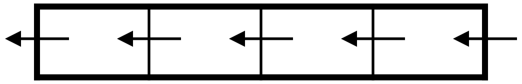
Shift Left Arithmetic



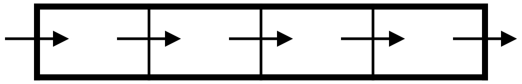
Shift Right Arithmetic



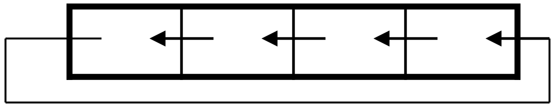
Shift Left Logical



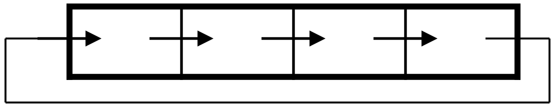
Shift Right Logical



Rotate Left Logical



Rotate Right Logical



Binary test values

"1110" ***sla*** 1 = "1100"
"0111" ***sla*** 1 = "1111"

"1100" ***sra*** 1 = "1110"
"1100" ***sra*** -1 = "1000"

"1100" ***sll*** 2 = "0000"
"1101" ***sll*** 3 = "1000"

"1100" ***srl*** 2 = "0011"
"1101" ***srl*** 3 = "0001"

"1100" ***rol*** 2 = "0011"
"1100" ***rol*** -1 = "0110"

"1100" ***ror*** 2 = "0011"
"1100" ***ror*** -1 = "1001"

Conditional Statements

If & Case $\Longrightarrow$ Sequential Statement

When & With $\Longrightarrow$ Concurrent Assignments

Case Statement

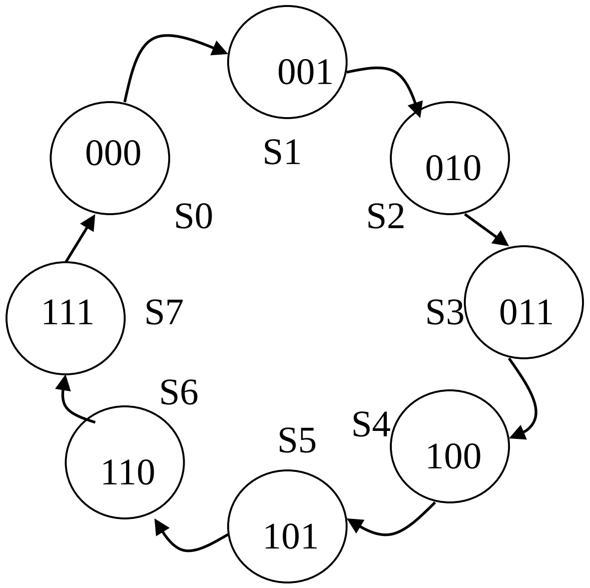
```
case name is
 when choice 1 => statement;
 when choice 2 => statement;
 when choice 3 => statement;
 when choice 4 => statement;
-----
-----
 when choice n => statement;
 end case ;
```

```
-- Case example
-- val, a, b, c, d are type integer
case val is
when 1 => a:= b;
when 2 => a:= 0;
when 3 => c:= d;
when others => null;
end case ;
```

CONCORDIA

Case Statement….

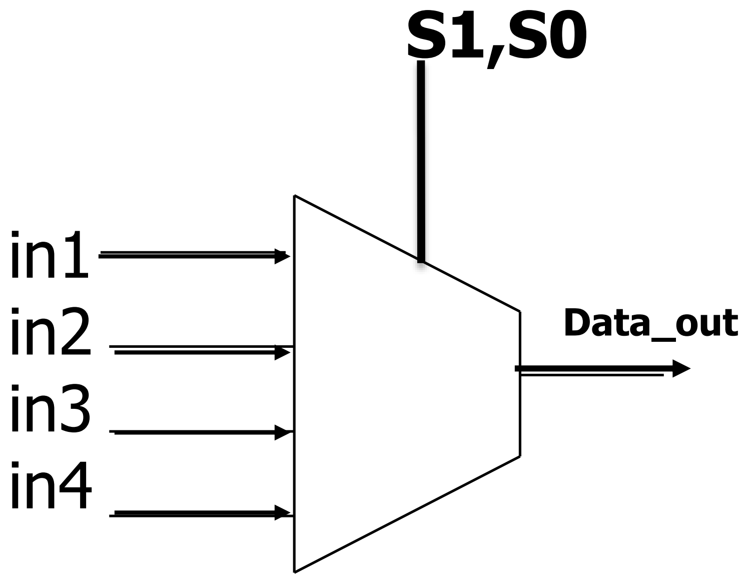
```
entity 3bit_counter is
port (clk: in BIT;
        state: out BIT_VECTOR(2 downto 0) );
end 3bit_counter;
architecture behave of 3bit_counter is
begin
process
variable current_state: BIT_VECTOR(2 downto 0) :=“111”;
begin
case current_state is
when “000” => current_state := “001”;
when “001” => current_state := “010”;
when “010” => current_state := “011”;
when “011” => current_state := “100”;
when “100” => current_state := “101”;
when “101” => current_state := “110”;
when “110” => current_state := “111”;
end case ;
state <= current_state after 10 ns ;
wait until (clk=‘1’) ;
end process;    end behave ;
```



CONCORDIA

With-Select Statement

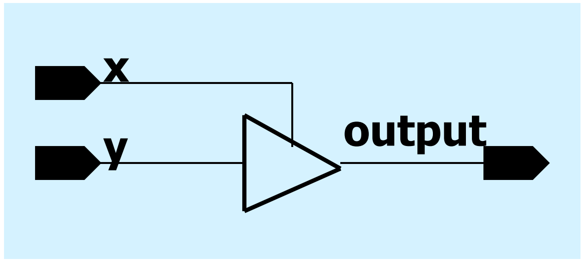
```
entity mux_4 is
port (in1, in2, in3, in4: in BIT;
          s: in BIT_VECTOR (1 downto 0);
         Data_out: out BIT );
end mux_4;
architecture behave of  mux_4 is
begin
with s select
Data_out <= in1 when “00”
         in2 when “01”
         in3 when “10”
         in4 when “ 11”;
end behave ;
```



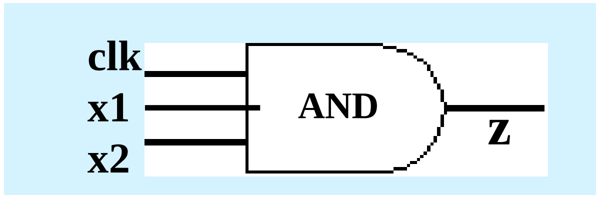
CONCORDIA

If Statement

```
-- Buffer Example
process (x,y)
begin
if x ='0' then
output <= 'Z'
-- High impedance state
else
output <= y;
end if;
end process;
```



```
-- "AND" example
process (clk,x1,x2)
begin
if clk ='1' and clk'event then
if x1 ='0' or x2='0' then
z <= '0'   else
z <= '1';
-- clocked "AND" gate example
end if;
end if;
end process;
```



CONCORDIA

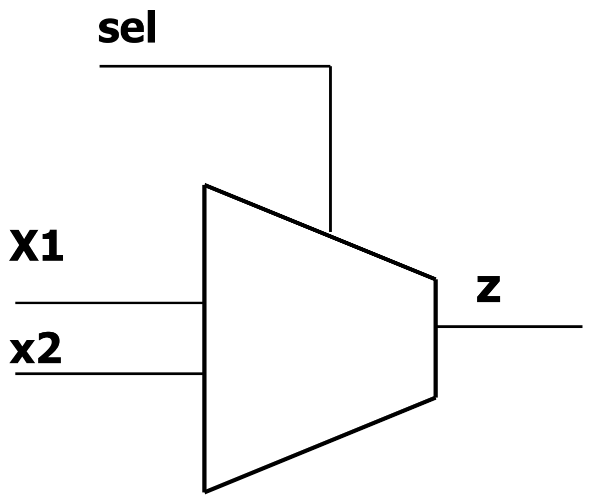
When-Else Statement

- With-else form a conditional concurrent assignment statement

```
-- Multiplexer Example

entity MUX is
port (x1, x2, sel: in std_logic;
        z : out std_logic);
end MUX;
architecture top of MUX is
begin

z <= x1 when sel = '1' else x2;
end top;
```

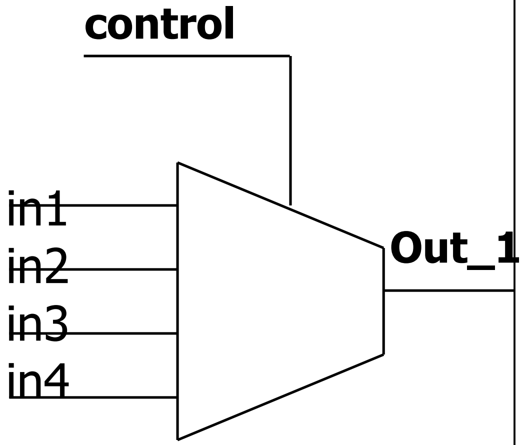


CONCORDIA

Multiplexer Example

```
-- Sequential Assignment

process ( control )
begin
case control is
when "00" => out_1 <= in1;
when "01" => out_1 <= in2;
when "10" => out_1 <= in3;
when "11" => out_1 <= in4;
end case;
end process;
```



```
-- Concurrent Assignment

with control select
Out_1 <=  in1 when "00"
          in2 when "01"
          in3 when "10"
          in4 when "11";
end behavior ;
```

Iterative Loops

-- There are 3 iterative loops in VHDL

- ➢ Simple Loop
- ➢ For Loop
- ➢ While Loop

-- The exit statement is a sequential statement which is associated with the loops

-- For the for loop the loop index is incremented
--and for
-- a while loop the condition is always checked

Simple Loops Example

```
a1: process
variable x: integer :=0;
variable y: integer :=0;
Begin
outerloop: loop
        x := x+1;
        y := 30;

innerloop: loop
        if y < = (4*x) then
        exit innerloop;
        end if;
        y:= y-x;
   end loop innerloop;
   exit loop outerloop when x>10;
   end loop outerloop;
   wait;
   end process;
```

Exit statement

Normal end Loop statement

While / For Loops

```
-- While/for Loop Example
a1: process
variable x: integer :=0;
begin
outerloop: for y in 1 to 20      loop
           x := 30;

innerloop: while x >= (5*y)  loop
           x := x-y ;
end loop innerloop;
end loop outerloop;
   wait;
   end process;
```

Iterative for loop

Constraint on while loop

Normal end Loop statement

Comparison of while and simple loop exit statement

```
-- While loop

i:=0;                          Less than
                               /equal

while ( (f(i) /='0') and (i<=100) )
loop
f(i) := (3 * i) ;
i := i + 1;

end loop;
```

```
-- Simple loop

i:=0;                          Less than
                               /equal

loop
    exit when f(i) /='0' or i<=100 loop
    f(i) := (3 * i) ;
    i := i + 1;

end loop;
```

Conditional statement is checked by the constraint on while and by exit on the simple loop

CONCORDIA

Next Statement

Next syntax: ***next*** **[loop label] [** ***when*** **condition] ;**

- Next statement is used in a loop to cause the next iteration.

- Without the label next statement applies to the innermost enclosing loop.

- Loop label is conditional.

```
-- EXAMPLE: Counting the number of zeros
number_zeros := 0 ;
for i in 0 to 31 loop
next when temp1(i) /= '0' ;
number_zeros:= number_zeros + 1 ;
end loop ;
```

CONCORDIA

Assertion Statement

Assert syntax: [label:] *assert* Boolean_condition [*report* string]
[sensitivity name] ;

- Assert statement is used by the programmer to encode constraints in the code.

- The constraints are checked during simulation and if the constraint conditions are not satisfied, message is sent to terminal. The severity of the message can be set by the programmer. Assert can also be used for debugging of the code. The report can give the programmer indication of the location of program error.

- Predefined sensitivity names are: *NOTE*, *WARNING*, *ERROR*, *FAILURE*. Default sensitivity for assert is *ERROR*

```
-- Assert Example
   assert (expected_output = actual_output )
   report " actual and expected outputs don't match "
   severity Error ;
```

Assertion Statement…

```
-- Assert example, door opens when z =‘1’

entity door_open is
port (key1, key2: in std_logic;
        z : out std_logic);
end door_open;
architecture top of door_open is
begin
if key1 = ‘1’ or key2 =‘1’ then
z <= ‘1’ ;
end if ;
assert not(key1=‘0’ and key2=‘0’)
report “ both keys are wrong, door remains closed ”
severity error ;
end top;
```

Null Statement

Null statement is used when there is nothing to do and hence its execution has no effect.

```
---------------------------
---------------------------
signal: z: BIT := '0';

case x is
            when 0 => z <='0' ;
            when 1 => z <='1' ;
            when others => Null;   -- Program does not do anything here
end case;
```

Generic

```
 -- Generic example, OR_ gate
entity OR_2 is
generic (prop_delay: time);
port (x, y: in std_logic;
        z : out std_logic);
end OR_2;
architecture top of or_2 is
begin
z <= x or y after prop_delay ;
end top;
          :
component and_2
generic (prop_delay : time);
port ( x,y: in BIT; z: out BIT);
end component ;
          :
o1: OR_2
generic map (prop_delay => 5 ns)
port map ( x =>x, y=>y, z=> z);
```

Propagation delay ***for entities can be written in a general manner and*** *exact value* ***put during their instantiation***

Global Package

- Functions and procedures can be declared globally, so that they are used throughout the design, or locally within the declarative region of an architecture, block, process, or another subprogram.

- For the subprogram that will be used throughout the design, the subprogram declaration in an external package will have the syntax:

```
package asim_package is
    function asim_global_function(...)
        return BIT;
end asim_package;

package body asim_package is
    function asim_global_function(...)
        return bit is
    begin
---------------
    end asim_global_function;
end asim_package;
---------------
```

```
use work.asim_package.asim_global_functio
entity asim_design is
begin
---------------------
End asim_design;
```

CONCORDIA

Synthesis Steps with VHDL

Analysis: *Static behavior, that checks for syntax and semantics*

Elaboration: *Creates ports, signals, architecture body, flattening the design (Can get a schematic). Eventually a flat collection of gates, FFs, processors, other units connected with signal nets.*

Simulation: *Discrete even driven simulation following the events to final steady state and final input to output transformation.*

Targeting: *Selecting an FPGA to be downloaded. Selection of optimization criterion*

Synthesis

- Type conversion functions are written using unconstrained integers. Therefore, cannot be synthesized. In a synthesizable design, an arbitrary width type should not be used. The solution is to use the conversion functions provided by the synthesis vendor or the IEEE 1076.3 signed or unsigned types.

- The wait statement is also not synthesizable.
- Floating Point numbers are usually not synthesizable.
-Time usually is ignored and placed with real values once device is tagetted

Attributes

- Is a VHDL feature that permits the extraction of additional information for an object such as signal, variable or type.

- Attributes also allow the access to additional information that may be needed in synthesis.

There are 2 classes of attributes:

- **Pre-defined (defined inside 1076 STANDARD)**
- **Introduced by the programmer or tool supplier**

Pre-defined Attributes:

Five kinds: Value, Function, Signal, Type or Range

Example:

wait until clk=‘1’ ***and*** clk’**event** ***and*** clk’ **last_value** =‘0’ ;

Not a reserved word BUT pre-defined in the 1076 package

Funtion Attributes

- Pos (value) – To return the position number of a type value

```
--Example
type state_type  is (Init, Hold, Strobe, Read, Idle) ;
variable P : INTEGER := state_type'pos (Read);
-- Value of P is 3
```

- Val (value) – To return the position number of a type value

```
--Example
variable X : state_type := state_type' Val (2);
-- X has the value of Strobe
```

- Succ (value) – Return the value to the position after the given type value

```
--Example
variable Y : state_type := state_type'succ (Init);
-- Y has the value of Hold
```

-- Other functions: **Pred (value)** **Leftof (value)** **Rightof (value)**

Value Attributes

- Left (value) – To return the leftmost element index of a given type

```
--Example
type BIT_ARRAY  is ARRAY (1 to 5) of BIT;
variable M: INTEGER := BIT_ARRAY' Left;
-- Value of M is 1
```

- Right (value) – To return the rightmost element index of a given type

- High (value) – Return the upper bound of a given scalar type

```
--Example
type BIT_ARRAY  is ARRAY (-15 to 15) of BIT;
variable M: INTEGER := BIT_ARRAY' High;
-- M has a value of  15
```

- Low (value) – Return the lower bound of a given scalar type

- Length (value) – Return the length of an array

```
type BIT_ARRAY  is ARRAY (0 to 31) of BIT;
variable N: INTEGER := BIT_ARRAY' length;
-- Value of N is 32
```

Value Attributes....

```
-- Example to show the value attributes in action

signal  sum : BIT_VECTOR (7 downto 0) ;

sum`Left = 7
sum`Right = 0
sum`High = 7
sum`Low = 0
sum`Range = 7 downto 0
sum`REVERSE_RANGE = 0 to 7
sum`Length = 8
```

CONCORDIA

Funtions Attributes....

- Event – Returns a true value if the signal had an event in current simulation time

```
--Example
process (Rst, clk)
begin
        if Rst ='1' then
        M <= '0';
        elsif clk = '1' and clk'event then          -- On look out for the clock rising edge
        M <= N;
        end if;
        end process ;
```

- Active – Returns true if any event (scheduled) occurs in current simulation

```
process (Rst, clk)
variable  A,E : BOOLEAN ;
begin
    M <= N after 10 ns
    A:= M'Active;     -- A = true
    E := M'Event;     -- E = false
    end process ;
```

CONCORDIA

Funtions Attributes….

- Last_event – Return the time elapsed since the previous event occurring

```
process
variable T : time;
begin
    P <= Q after 5 ns;
    wait 10 ns;
    M <= ‘0’;
    T := P’last_event;              -- T gets the value of 5 ns
    end process ;
```

- Last_value – Return the value of the signal prior to the last event

```
process
variable T2 : BIT;
begin
    P <= ‘1’ ;
    wait 10 ns;
    P <= ‘0’;
    wait 10 ns
    T2 : = P’last_value;     -- T2 gets a value of ‘1’
    end process;
```

CONCORDIA

Funtions Attributes....

- Last_active– Return the time elapsed since the last scheduled event of the signal

```
-- Example

process
variable T : time;
begin

    P <= Q after 30ns;
    wait 10 ns;
    T := P'last_active;           -- T gets the value of 10 ns
    ----------------------
    ----------------------
    end process ;
```

Funtions Attributes….

- *Delayed (time)*
– Creates a delayed signal that is identical in waveform to the attribute applied signal.

- *Stable (time)*
– Creates a signal of type BOOLEAN that is true when the signal is stable (without any events) for some period of time.

- *Quiet (time)*
– Creates a signal of type BOOLEAN that is true when the signal has no scheduled events for some period of time.

- *Transaction (time)*
– Creates a signal of type BIT that toggles its value when an actual event or transaction occurs on the signal.

- *PC based packages for VHDL*
- ActiveHDL
  http://www.aldec.com/products/active-hdl/

  Please visit this site for window based VHDL they have a demo that you can be downloadedThe tool is called ActiveHDL.
- Xilinx:
  www.xilinx.com/ise/logic_design_prod/webpack.htm
- VHDL Simili
  http://www.symphonyeda.com/products.htm. There's a free version for students, but you can only simulate 10 waveforms at the same time. There is also a 30 day trial for the standard/professional edition which does not have this limit. It is very good and
- Aldec's Active-HDL EDA tool and free educational resources
  http://www.aldec.com/downloads

CONCORDIA

Explanation of Delta Delay

In VHDL simulation when no delay is prescribed for the signal transformation then some time delay must finish before the signal assignment is carried out and the signal takes its new value. Generally delays are 3 types:
1)Transport delay (propagation delay), 2)Inertia delay(propagation + pulse width) and 3) Delta delay. Although the first two types are well defined the third is ambiguous.
In VHDL Simulation everything within a process happens simultaneously. However if two or more signals are assigned to the same target in a process then the final assignment takes precedence. To do this within the same simulation cycle, the value of the signal does not change immediately, rather is remembered and used in the next delta cycle. So the Delta delay is a fictions quantum delay for the purpose of simulation only. Within the same simulation time, when all the processes are completed then the signal changes value and the next delta cycle takes effect. When all signals are processed by the succeeding delta cycles then the simulation cycle advances. The delta delay is the default delay when no delay is specified.

Sub-programs

Functions and *procedures* in VHDL commonly referred to as **subprograms**,
are directly analogous to functions and procedures in a high-level programming language such as Pascal or C/C++.

Subprograms are very useful for separating segments of VHDL that are commonly used. They can be defined locally (e.g inside architecture), or they can be placed in a package and used globally anywhere in the design.

Subprograms are quite similar to processes in VHDL. Any statement that can be entered in a VHDL process can also be entered in a function or procedure, with the exception of a wait statement (since a subprogram executes once each time it is called and cannot be suspended while executing).

Procedures

- A procedure is a subprogram that has an argument list consisting of inputs and outputs, and no return value.

- It allows the programmer to control the scheduling of simulation without the overhead of defining several separate design entities.

Procedure Syntax:

procedure procedure_name (parameter_list) ***is***
[variable declaration]
[constant declaration]
[type declaration]
[use clause]
begin
sequential statements
end procedure_name;

Procedure Call: procedure_name (association list);

Procedures….

```
-- Procedure Example, and_ gate

procedure and_2 (a,b: in BIT; c: out BIT);  is
begin

if a=‘1’ and b=‘1’ then
c <=‘1’;
else
c<=‘0’;

end if;
end and_2 ;
```

*Procedure can have parameters of the mode **in**, **inout**, and **out**.*

Functions

- A function is a subprogram that has only inputs in its argument list, and has a return value.

- Can only take parameters of mode *in*. They are useful for modeling of combinational logic.

Function Syntax:

function function_name (parameter_list)
return type_name ***is***
[variable declaration]
[constant declaration]
[type declaration]
[use clause]
begin
[sequential statements]
return expression;
[sequential statements]
end function_name ;

Function Call: function_name (parameter);

Functions….

```
-- Function Example, and_ gate

function and_func (a,b: in BIT) return BIT  is
begin

if a='1' and b='1' then
return '1';
else
return '0';

end if;
end and_func ;
```

Function return type is specified here

CONCORDIA

Functions….

```
 -- Convert an integer to a unsigned STD_ULOGIC_VECTOR, from std_logic_arith.all

function CONV_UNSIGNED(ARG: INTEGER; SIZE: INTEGER) return UNSIGNED is
            variable result: UNSIGNED(SIZE-1 downto 0);
            variable temp: integer;
            Begin
            temp := ARG;
            for i in 0 to SIZE-1 loop
               if (temp mod 2) = 1 then
                        result(i) := '1';
               else
                        result(i) := '0';
               end if;
               if temp > 0 then
                        temp := temp / 2;
               else
                        temp := (temp - 1) / 2;
               end if;
            end loop;
            return result;
            end;
```