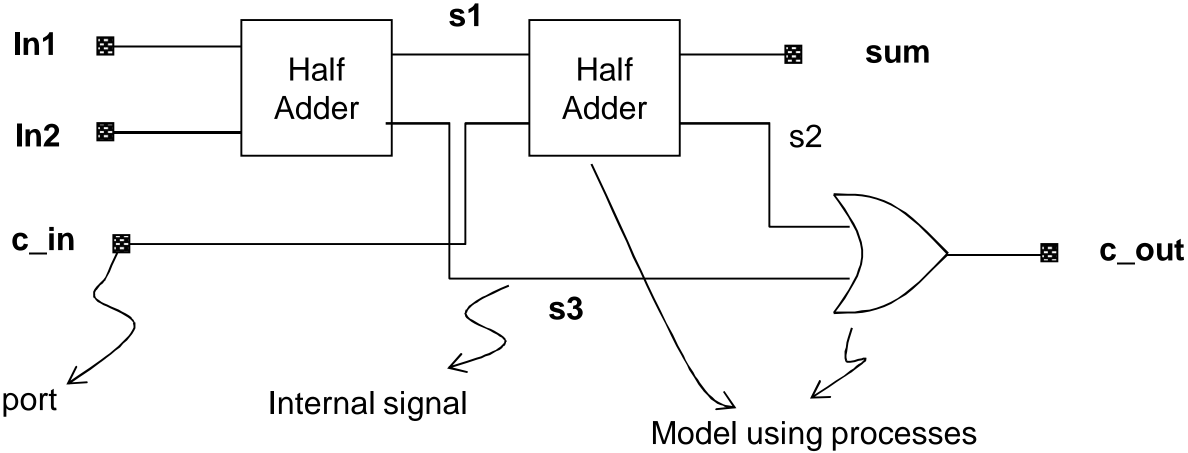
Multiple process execution



- Each of the components of the full adder can be modeled using a process
- Processes execute concurrently
  - In this sense they behave exactly like concurrent signal assignment statements
- Processes communicate via signals

Multiple process execution (Cont.)

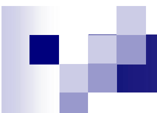
```
library IEEE;
use IEEE.std_logic_1164.all;

entity full_adder is
port (In1, c_in, In2: in std_logic;
        sum, c_out: out std_logic);
end entity full_adder;

architecture behavioral of full_adder is
signal s1, s2, s3: std_logic;
constant delay:Time:= 5 ns;
begin

HA1: process (In1, In2) is
begin
s1 <= (In1 xor In2) after delay;
s3 <= (In1 and In2) after delay;
end process HA1;

HA2: process(s1,c_in) is
begin
sum <= (s1 xor c_in) after delay;
s2 <= (s1 and c_in) after delay;
end process HA2;

OR1: process (s2, s3) -- process
describing the two-input OR gate
begin
c_out <= (s2 or s3) after delay;
end process OR1;

end architecture behavioral;
```

Conditional Statement in Process

The keywords used for conditional assignments and selected assignments differ from those used within a process:

Outside Processes	Inside Processes
WHEN..ELSE	IF..ELSIF..ELSE..END IF
WITH..SELECT..WHEN	CASE..WHEN..END CASE

Processes can be used for combinational logic but most often, processes encapsulate sequential logic



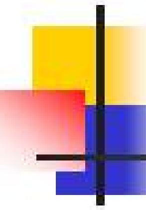
Process Model

```
architecture EXAMPLE of COMMUNICATE is
        signal DONE_FIRST : Bit := '0';
begin
        FIRST: process
        begin
                ......          -- do first steps......
                DONE_FIRST <= '1';
                wait for 10 ns;
        end process;
        NEXT: process
        begin
                wait until DONE_FIRST = '1';
                .......         -- do the other steps
        end process;
end EXAMPLE;
```

Declaration of the signal

Assignment

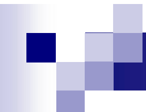
Other process waits until '1' is assigned to the signal



IF statement

- **IF** statements
  - Represent hardware decoders in both abstract and detailed hardware models
  - Select for execution one or more of the enclosed sequential statements
    (if more than one sequential statement, use ';' at the end of each statement).
  - Can be nested.

```
if CONDITION1 then Sequential_Statement(s);
            {elsif CONDITION2 then Sequential_Statement(s);}
            {elsif CONDITION3 then Sequential_Statement(s);}
            {elsif CONDITION4 then Sequential_Statement(s);}
            {......
            }
                        [else Sequential_Statement(s);]
end if;
```



```
if CONDITION then
        .....           -- Sequential_Statement(s)
end if;
```

```
if CONDITION then
        .....           -- Sequential_Statement(s)
        else
        .....           -- Sequential_Statement(s)
end if;
```

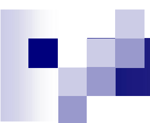
```
if CONDITION1 then
        .....           -- Sequential_Statements
        elsif CONDITION2 then
                .....           -- Sequential_Statement(s)
        elsif CONDITION3 then
                .....           -- Sequential_Statement(s)
        else
                .....           -- Sequential_Statement(s)
end if;
```

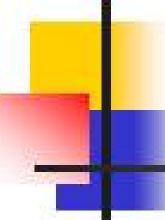
```
SIGNAL reset, clock, d, q          :std_logic;

PROCESS (reset, clock)
-- reset and clock are in the sensitivity list to
-- indicate that they are important inputs to the process
BEGIN
   -- IF keyword is only valid in a process
   IF (reset = '0') THEN
        q <= 0;
   ELSIF (clock'EVENT AND clock = '1') THEN
```

The `EVENT` attribute is true if an edge has been detected on the corresponding signal.

D Flipflop

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY dffe IS
   PORT( rst, clk, ena, d : IN     std_logic;
         q                : OUT    std_logic );
END dffe;

ARCHITECTURE synthesis1 OF dffe IS
BEGIN
   PROCESS (rst, clk)
   BEGIN
         IF (rst = '1') THEN
               q <= '0';
         ELSIF (clk'EVENT) AND (clk = '1') THEN
               IF (ena = '1') THEN
                     q <= d;
               END IF;
         END IF;
   END PROCESS;
END synthesis1;
```

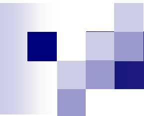




CASE statement

- CASE statements
  - useful to describe decoding of busses and other codes
  - Select for execution one of a number of alternative sequential statements (if more than one sequential statement, use ';' at the end of each statement).

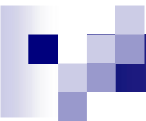
```
case (EXPRESSION) is
        when CHOISE1 => Sequential_Statement(s);
        when CHOISE2 | CHOISE3 | CHOISE4 => Sequential_Statement(s);
        when value1 to value2 => Sequential_Statement(s);
        when others => Sequential_Statement(s);
end case;
```

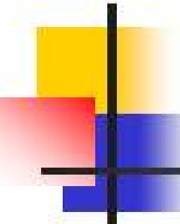


CASE statement

Example

```
library IEEE;
use IEEE.Std_Logic_1164.all;

entity CASE_STATEMENT is
port (      U, W, X, Y: in integer range 0 to 15;
            Z           : out integer range 0 to 15);
end CASE_STATEMENT;
architecture CASE_ARC of CASE_STATEMENT is
begin
            process (U, W, X, Y)
            begin
                        case Y is
                                    when 0 to 9  => Z <= X;
                                    when 13      => Z <= W;
                                    when 11 | 15=> Z <= U;
                                    when others => Z <= 0;
                        end case;
            end process;
end IF_ARC;
```

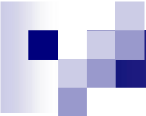




LOOP statement

- LOOP Statements
  - Provide a convenient way to describe bit-sliced logic or iterative circuit behavior.
  - Include sequential statements to execute repeatedly, zero or more times.
  - Can be nested.

```
            [LOOP_LABEL]: while CONDITION1 loop
                    .....          -- Sequential_Statement(s)
            end loop[LOOP_LABEL];
OR
            [LOOP_LABEL]: for IDENTIFIER in DISCRETE_RANGE loop
                    .....          -- Sequential_Statement(s)
            end loop[LOOP_LABEL];
```



LOOP statements

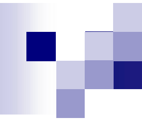
Example

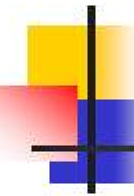
FOR LOOP

```
LABEL_ONE:  for I in 1 to 20 loop
                Sequential_Statement(s);
end loop LABEL_ONE
```

WHILE LOOP

```
INITIAL := 0;
LABEL_TWO:  while (INITIAL < 10) loop
                Sequential_Statement(s);
                INITIAL := INITIAL + 1;
end loop LABEL_TWO
```

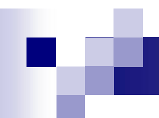


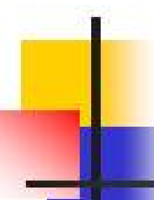


EXIT statement

- The **EXIT** statement completes the execution of an enclosing LOOP statement.
- The LOOP label in the EXIT statement identifies the particular loop to be exited.

```
exit [LABEL] [when CONDITION];
```



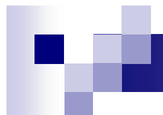


EXIT statement

Example

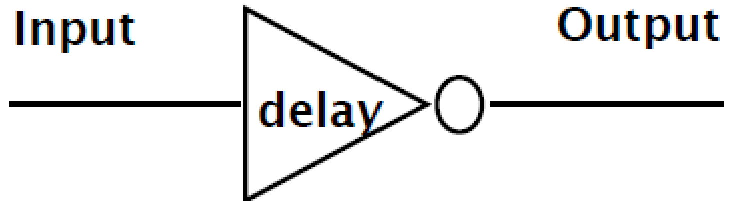
```
for I in 1 to 30 loop
        if (VALUE(I) = 0) then exit;
        end if;
OUTPUT_VALUE(I) := VALUE(I);
end loop;
```

The **for** statement has a range for **loop**. When the I indexed variable has a value of zero, **exit** causes execution to exit the loop entirely.

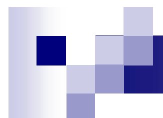


VHDL Delay Models

- Delay is created by scheduling a signal assignment for a future time.
- Delta delay
  - Inertial
  - Transport



- Inertial delay
  - Default delay model
  - Suitable for modeling delays through devices with inertia (e.g. logic gates)
  - Pulses shorter than a device's delay are not propagated to its output
- Transport delay
  - Model delays through elements with no inertia, e.g., wires
  - no inertia = all input events are propagated to output signals
  - Any pulse, no matter how short, is propagated
- Delta delay
  - What about models where no propagation delays are specified?
  - Infinitesimally small delay is automatically inserted (after 0ns) by the simulator to preserve correct ordering of events
  - The delta delay is a special case of inertial delay with delay infinitesimally small

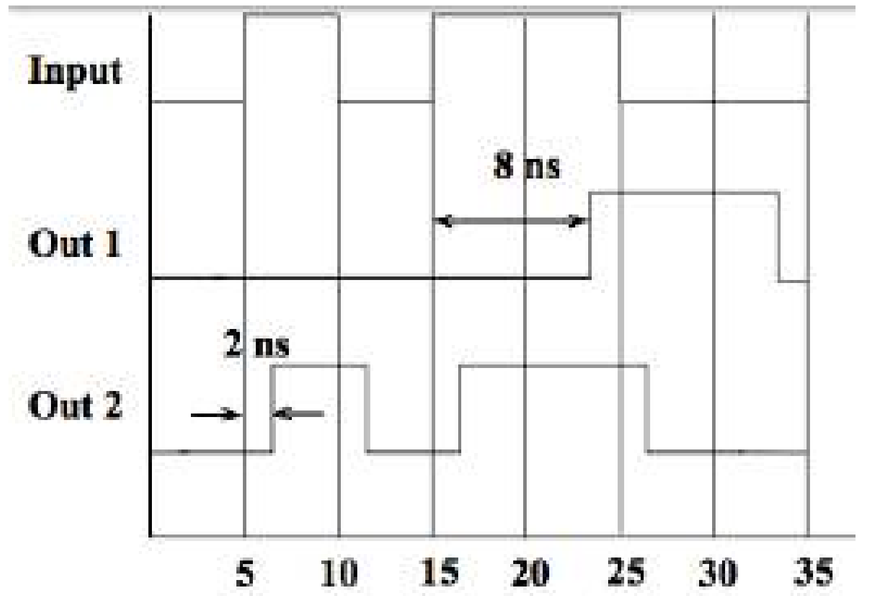


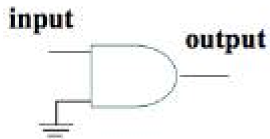
Inertial Delay (for gate)

Signal <= [[reject *pulse limit*] inertial] *expression* after *inertial delay value*;

If no pulse limit specified then default pulse rejection limit is the inertial delay value.

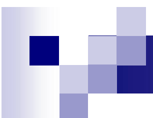
```
Example:
Out_1 <= inertial Input after 8 ns;
Out_2 <= inertial Input after 2 ns;
```





Out_2 <= Input after 2 ns;

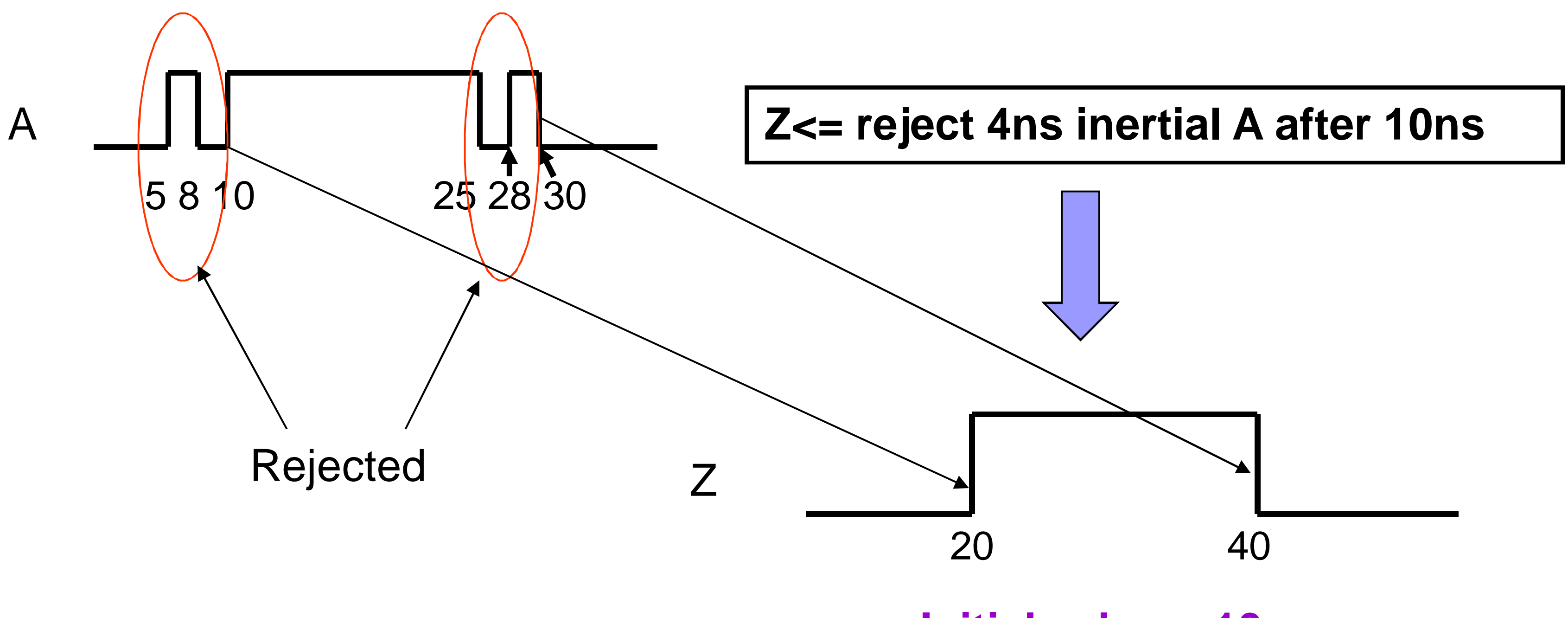
Out_2 <= inertial Input after 2 ns;

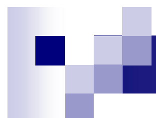


Inertial Delay

Signal <= [[reject *pulse limit*] inertial] *expression* after *inertial delay value*;

If no pulse limit specified then default pulse rejection limit is the inertial delay value.



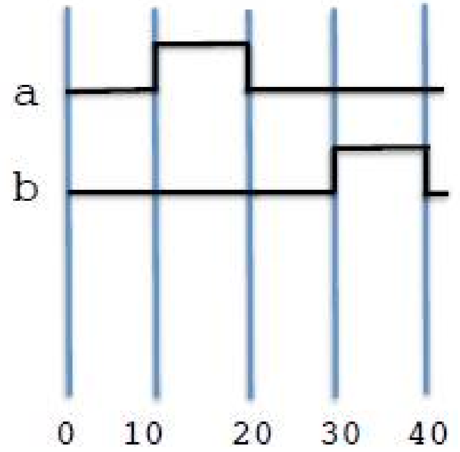


Transport Delay (for wire)

Signal <= transport *signal name* after *transport delay value*;

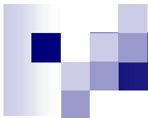
It is nothing but propagation delay.

```
ARCHITECTURE beh OF delay_line
BEGIN
  b <= TRANSPORT a AFTER 20 ns;
END beh;
```



What will happen if I write
b<= a after 20ns;

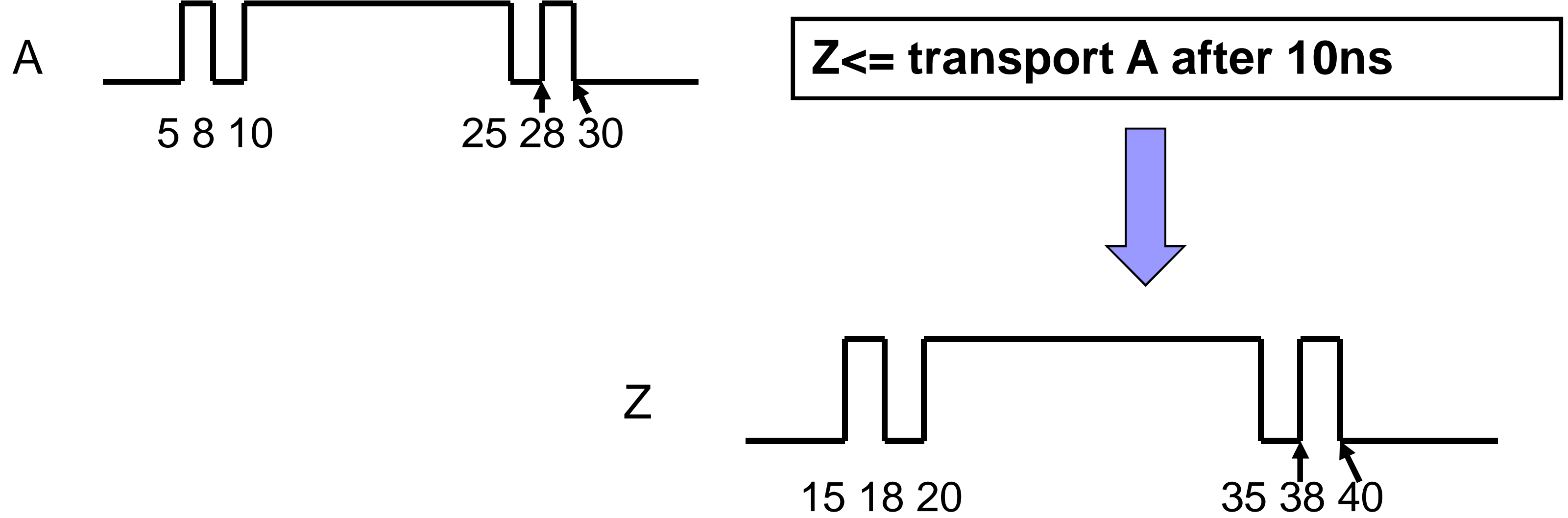
Inertial delay === 20ns is pulse rejection limit

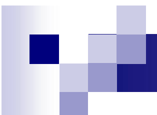


Transport Delay

Signal <= transport *signal name* after *transport delay value*;

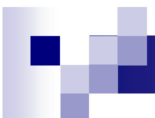
It is nothing but propagation delay.





Delta Delay

- Delta delay needed to provide support for concurrent operations with zero delay
  - The order of execution for components with zero delay is not clear
- Scheduling of zero delay devices requires the delta delay
  - A delta delay is necessary if no other delay is specified
  - A delta delay does *not advance simulator time*
  - One delta delay is an infinitesimal amount of time
  - The delta is a scheduling device to ensure repeatability



Delta Delay

- Δ does not have to be given a value, but is **used within the simulator** to **order events**.
- The simulator organizes and processes events in time order of occurrence. Events that occur Δ seconds later are followed by events that occur 2 Δ seconds later, etc.
- **Delta delays** are used to **enforce dependencies between events** and thereby **ensure correct simulation.**

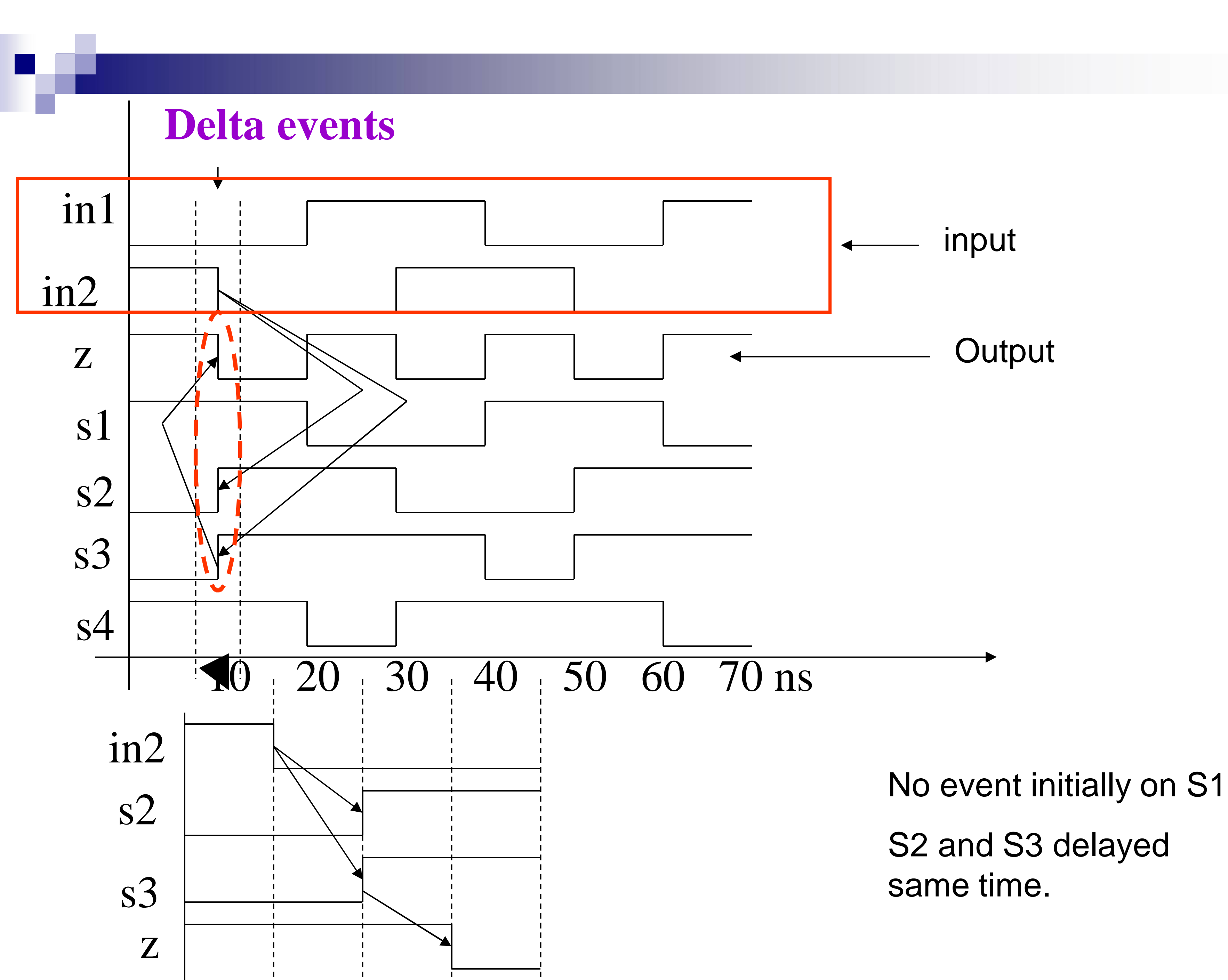
- The statement sum <= (a **xor** b)**;** is given *implicitly* a time expression "**after** 0 **ns**" after the value expression.

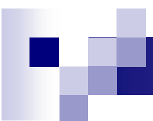
So the component is *effectively* given a delay value of Δ.

Delta Delay

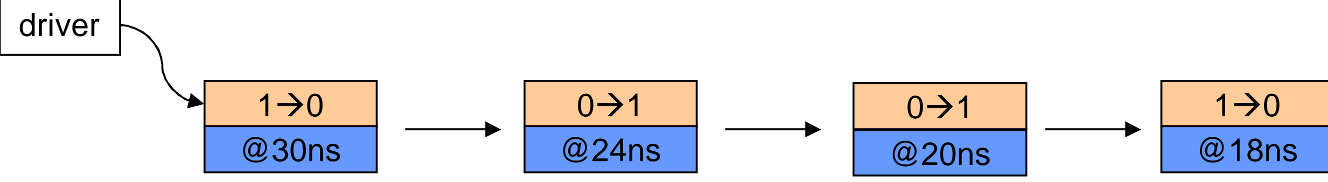
in1 NOT s1 NAND s3
NAND z
in2 NOT s2 NAND s4

```
library IEEE;
use IEEE.std_logic_1164.all;
entity combinational is
port(in1, in2:in std_logic;
        z: out std_logic);
end combinational

architecture behavior of combinational is
signal s1, s2, s3, s4:std_logic:= ‘0’;
begin
s1 <= not in1;
s2 <= not in2;
s3 <= not (s1 and in2);
s4 <= not (s2 and in1);
z <= not (s3 and s4);
end behavior;
```


Delta events
in1
in2
z
s1
s2
s3
s4
input
Output
10
20
30
40
50
60
70 ns
in2
s2
s3
z
10
Δ
2Δ
3Δ
4Δ
No event initially on S1.
S2 and S3 delayed
same time.

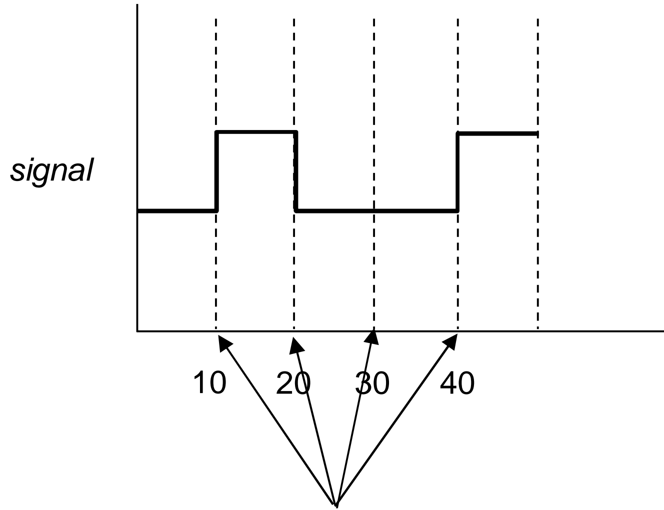


Implementation of Signals



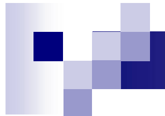
- *Driver* is set of future signal values: current signal value is provided by the transaction at the head of the list
- We can specify multiple waveform elements in a single assignment statement
  - Specifying multiple future values for a signal
- Rules for maintaining the driver
  - Conflicting transactions

Example: Waveform Generation



signal <= ‘0’,‘1’ **after** 10 **ns**,‘0’ **after** 20 **ns**,‘1’ **after** 40 **ns**;

- Multiple waveform elements can be specified in a single signal assignment statement
- Describe the signal transitions at future point in time
  - Each transition is specified as a waveform element



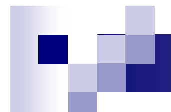
Signal Driver

```
process
begin
………
        r<= 3 after 5ns,21 after 10ns, 14 after 17ns;
end process;
```

r ←

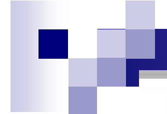
Current	3@T+5ns	21@T+10ns	14@T+17ns

time →

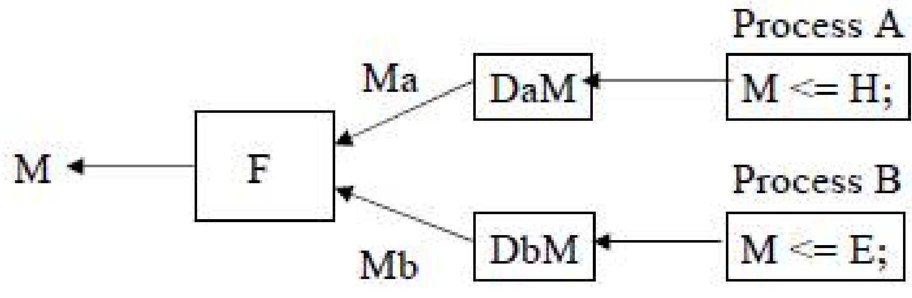


Cont

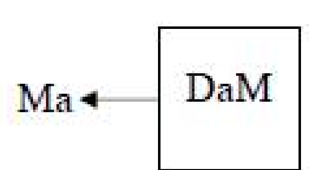
- **Each process that assigns a value to a signal creates a separate waveform driver for the signal.**
- **A resolution function resolves conflicts among the various drivers for the signal.**



- Each signal driver consists of a waveform of (time,value) pairs.
- Current value of a driver
  - There is exactly one transaction in the driver whose time value is less than or equal to the current simulation time.
  - The signal value of the above transaction is the current value of the driver.



If, as a result of simulation time advance, the current simulation time becomes equal to the time component of the next transaction, then the first transaction is deleted from the projected output waveform and the next transaction becomes the current value of the driver.

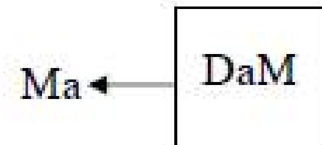


CV

5 ns	10 ns	15 ns	22 ns	31 ns	42 ns
'0'	'1'	'0'	'0'	'1'	'0'

State of driver Ma at simulation time 8 ns. CV of signal is '0'.

CV

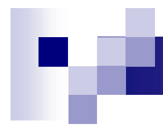


5 ns	10 ns	15 ns	22 ns	31 ns	42 ns	
'0'	'1'	'0'	'0'	'1'	'0'	

When simulation time advances to 10 ns, the driver is updated, and the current value of the signal becomes '1'.

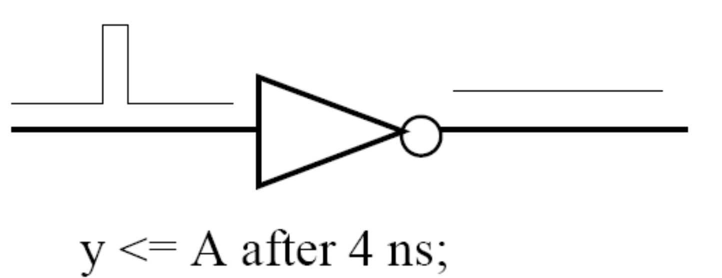
CV

10 ns	15 ns	22 ns	31 ns	42 ns		
'1'	'0'	'0'	'1'	'0'		



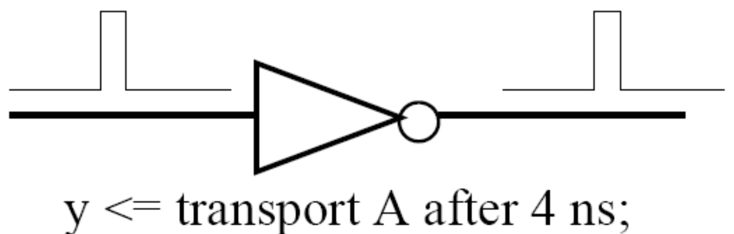
A problem with inertial delay

Inertial delay is intended to model physical device that have inertia and reject spikes. Unfortunately, the model assumes that the propagation delay and inertia delay are the same.



y <= A after 4 ns;

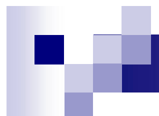
Inertial model can be used to reject glitch, but it also defines prop delay. Output has prop delay of 4 ns, rejects glitches < 4 ns.



y <= transport A after 4 ns;

Output delayed from input by 4 ns. No glitch rejection.

Output has propagation delay of 4ns



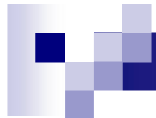
Effect of transport delay on drivers

```
signal rx_data : natural;
        .......
process
        begin
                ....
        rx_data<= trasport 11 after 10ns;
        rx_data<= trasport 20 after 22ns;
        rx_data<= trasport 35 after 18ns;
        end process
```

Current	11@T+10ns	20@T+22ns

Current	11@T+10ns	35@T+18ns

20@T+22ns



Effect of Inertial Delay on Driver

```
process
begin
data <= 11 after 10ns;
data <= reject 15ns inertial 22 after 20ns;
data <= 33 after 15ns;
wait;
end process;
```

Current	11@10ns

Calculate the window:
F and F- pulse rejection limit
If the value within window is different with previous value then
Discard the previous value

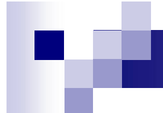
F= 20 and F-15 = 20-15= 5

Value within this is 11@10 and 22 @20
(between 5 to 20ns)
So previous 11@10 be removed

Current	22@20ns

Finally

Current	33@15



Cont..

```
process
begin
data <= 1 after 5ns,21 after 9ns, 6 after 10ns, 12 after 19ns;
data <= reject 4ns inertial 6 after 12ns, 20 after 19ns;
wait;
end process;
```

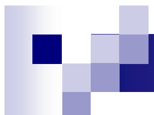
Current	1@5	21@9	6@10	12@19

Second statement 6 after 12ns < 12 after 19ns : **delete 12 after 19ns;**

F= 12 and F-4 = 12-4= 8 (my range is 8 to 12)

Value within this is 21@9, 6 @10, 6 @12, So previous 21@9 be removed

Current	1@5	6@10	6@12	20@19



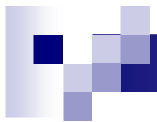
Unconstrained array

A constrained array's index range is explicitly defined; for example, the integer range `(1 to 4)`. When you declare a variable or signal of this type, it has the same index range.

The syntax of an unconstrained array type definition is

```
type array_type_name is
    array (range_type_name range <>)
        of element_type_name ;
```

```
type BIT_VECTOR is array(INTEGER range <>) of BIT;
  -- An unconstrained array definition
. . .
variable MY_VECTOR : BIT_VECTOR(5 downto -5);
```

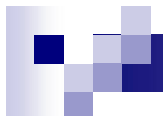


Array Attributes

VHDL Compiler supports the following predefined VHDL attributes for use with arrays:

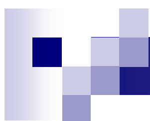
- left
- right
- high
- low
- length
- range
- reverse_range

MY_VECTOR'left	5
MY_VECTOR'right	–5
MY_VECTOR'high	5
MY_VECTOR'low	–5
MY_VECTOR'length	11
MY_VECTOR'range	(5 down to –5)
MY_VECTOR'reverse_range	(–5 to 5)

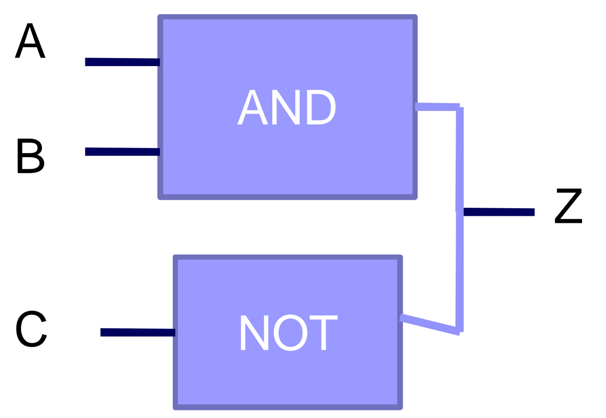


Difference

- A constrained array is an array where the index is specified (and hence the number of components is specified), we say that the bounds are static, hence constrained arrays are sometimes referred to as *static* arrays.
- When an unconstrained array is declared the index type is a specified but there is no need to state the limit of the index.



Multiple Driver (Concurrent)

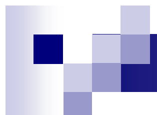



```
architecture xx of legal is
begin
Z<= A and B after 10ns;
Z<=not C after 5ns;
end xx;
```

```
architecture xx of legal is
begin
Z<= '1' after 2n,'0' after 5ns, '1' after 10ns;
Z<=  '0' after 4ns, '1' after 5ns, '0' after 20ns;
Z<= '1' after 10ns, '0' after 20ns;
end xx
```

In this example, there are two gates driving the output signal Z. How is the value of Z determined?

Resolution function



Resolved Function o/p

Current	1@10	0@5	1@2

Driver -1

Current	0@20	1@5	0@4

Driver -2

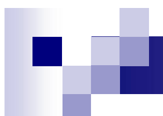
Current	0@20	1@10

Driver -3

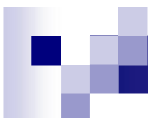
Resolution Function

Z

T	2ns	4ns	5ns	10ns	20ns
Z	1	1	1	1	1
	1	1	0	1	1
	0	0	1	1	0
	0	0	0	1	0



- The value of each driver is an input to the resolution function and based on the computation performed within the resolution function, the value returned by this function becomes the resolved value for the signal. The resolution function is user-written and it may perform any function.
- A signal with more than one driver must have a resolution function associated with it, otherwise, it is an error. Such a signal is called a *resolved* signal. A resolution function is associated with a signal by specifying its name in the signal declaration.



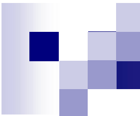
Sub Program- Function

A subprogram defines a sequential algorithm that performs a certain computation and executes in zero simulation time.
A subprogram is defined using a *subprogram body.* The typical format for a subprogram body is

```
subprogram-specification is
          subprogram-item-declarations
begin
          subprogram-statements     -- Same as sequential-statements.
end [ subprogram-name ];
```

The *subprogram-specification* specifies the name of a subprogram and defines its interface, that is, it defines the formal parameter names, their class (i.e., signal, variable, or constant), their type, and their mode (whether they are in, out, or inout).

A return statement, which is also a sequential statement, is a special statement that is allowed only within subprograms. The format of a return statement is
return *{ expression]*;
The return statement causes the subprogram to terminate and control is returned back to the calling object.



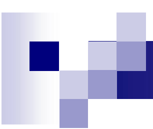
Function

- Functions are used to describe frequently used sequential algorithms that return a single value. This value is returned to the calling program using a return statement. Some of their common uses are as resolution functions

```
function LARGEST (TOTAL_NO: INTEGER; SET: PATTERN)
          return REAL is
          -- PATTERN is defined to be atype of 1-D array of
          -- floating-point values, elsewhere.
```

function-name (parameter-list) **return** *return-type*

```
function VRISE (signal CLOCK_NAME: BIT) return BOOLEAN is
begin
          return (CLOCK_NAME = '1') and CLOCK_NAME'EVENT;
end VRISE;
```

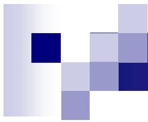


There are five classes of predefined attributes.

1. Value attributes: these return a constant value
2. Function attributes: calls a function that returns a value
3. Signal attributes: creates a new signal
4. Type attributes: returns a type name
5. Range attributes: returns a range

Function Attributes

- T'POS(V): returns the position number of the value V in the ordered list of values of T.
- T'VAUP): returns the value of the type that corresponds to position P.
- T'SUCC(V): returns the value of the parameter whose position is one larger than the position of value V in T.
- T'PRED(V): returns the value of the parameter whose position is one less than the position of value V in type T.
- T'LEFTOF(V): returns the value of th^parameter that is to the left of value V in type T.
- T'RIGHTOF(V): returns the value of the parameter that is to the right of value V in type T.



Resolution function for driver

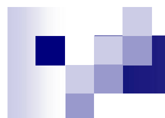
```
signal BUSY: WIRED_OR BIT;
```

```
function WIRED_OR (INPUTS: BIT_VECTOR) return BIT is
begin
        for J in INPUTS'RANGE loop
                if INPUTS(J)='1' then
                        return '1';
                end if;
        end loop;
        return '0';
end WIRED_OR;
```

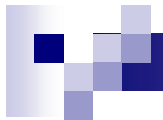
```
port (A,B, C: in BIT; Z: out WIRED_OR BIT);
```

```
subtype RESOLVED_BIT is WIRED_OR BIT;
signal BUSY: RESOLVED_BIT;
```

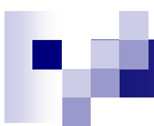
No arguments need be specified, since by default, the arguments for the function are the current values of all the drivers for that signal.



- Whenever an event occurs on a resolved signal, the resolution function associated with that signal is called with the values of all its drivers.
- The return value from the resolution function becomes the value for the resolved signal.
- The resolution function has only one input parameter, which is a one dimensional unconstrained array. The input parameter type and the return type are the same type as the signal.



- A function is recognized as a resolution function if it is associated with a signal in the signal declaration.
- we introduced a predefined attribute of an array object called RANGE. This attribute returns the range of the number of elements of the specified array object.
- For example, if there are four drivers when the WIRED_OR function is called, INPUTS'RANGE returns the range "0 to 3".

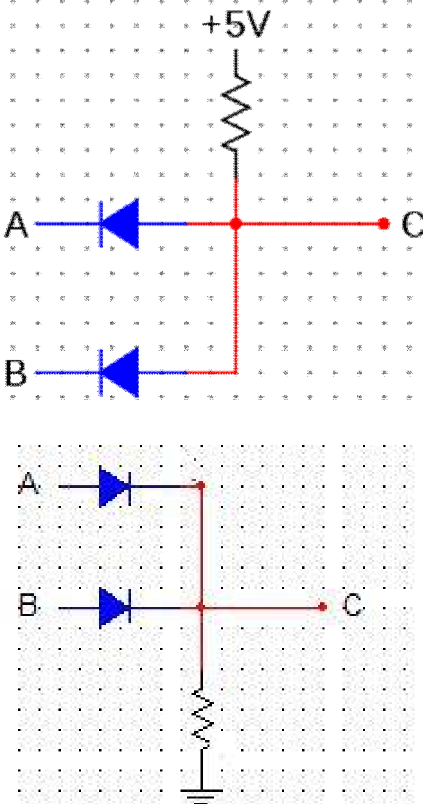


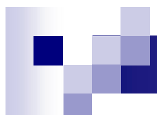
Wired logic

A **wired logic connection** is a logic gate that implements boolean algebra (logic) using only active and passive components such as diodes and resistors. A wired logic connection can create an AND or an OR gate.

The wired AND connection is a form of AND gate. It uses a pull up resistor and one diode per input to create this function.

The wired OR connection electrically performs the Boolean logic operation of an OR gate, using a pull down resistor and one diode per input.





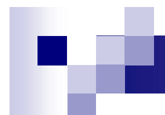
Package

PACKAGE → compile → LIBRARY

A LIBRARY may contains several packages

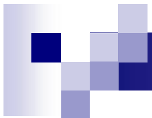
PACKAGE
COMPONENT
FUNCTION
PROCEDURE
TYPE
CONSTANT
SIGNAL
etc.

Main code
LIBRARY declarations
ENTITY
ARCHITECTURE
CONFIGURATION



Package declaration

- Used to store a common declaration
  - Components, types, procedure, function
- Can be imported as a single unit with **use** clause.
- Behavior of function does not appear in package.

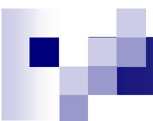


A package declaration contains a set of declarations that may possibly be shared by many design units. It defines the interface to the package, that is, it defines items that can be made visible to other design units,

package *package-name* **is**
package-item-declarations "> These may be:
- subprogram declarations ~ type declarations
- subtype declarations
- constant declarations
- signal declarations
- file declarations
- alias declarations
- component declarations
- attribute declarations
- attribute specifications
- disconnection specifications
- use clauses

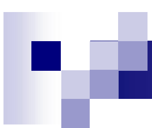
end [*package-name*] ;

Items declared in a package declaration can be accessed by other design units by using the library and use context clauses.



```
package SYNTH_PACK is
        constant LOW2HIGH: TIME := 20ns:
        type ALU_OP is (ADD, SUB, MUL, DIV, EQL);
        attribute PIPELINE: BOOLEAN;
        type MVL is ('U', '0', '1', 'Z');
        type MVL_VECTOR is array (NATURAL range <>) of MVL;
        subtype MY_ALU_OP is ALU_OP range ADD to DIV;
        component NAND2
                port (A, B: in MVL; C: out MVL);
        end component;
end SYNTH_PACK;
```

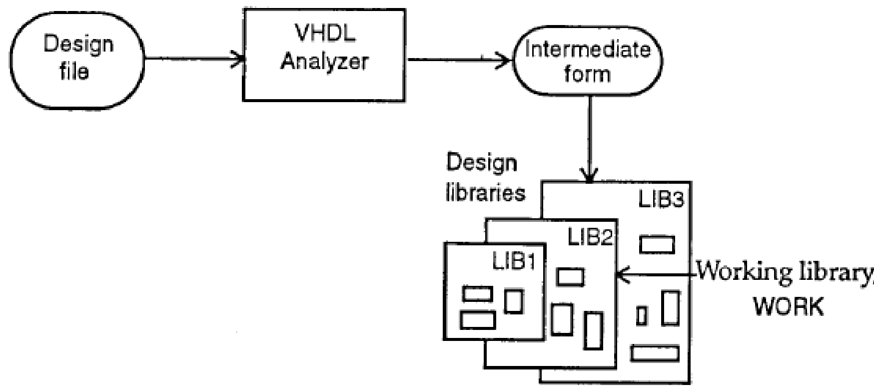
A package body primarily contains the behavior of the subprograms and the values of the deferred constants declared in a package declaration. It may contain other declarations as well, as shown by the following syntax of a package body.



package body *package-name* **is**
package-body-item-daclarations "> These are:
- subprogram bodies -- complete constant declarations
- subprogram declarations
- type and subtype declarations
- file and alias declarations
- use clauses

end [*package-name*];

A compiled VHDL description is stored in a design library. A design library is an area of storage in the file system of the host environment. The format of this storage is not defined by the language. Typically, a design library is implemented on a host system as a file directory and the compiled descriptions are stored as files in this directory. The management of the design libraries is also not defined by the language and is again tool implementation-specific.



The design file is an ASCII file containing the VHDL source. It can contain one or more design units, where a design unit is one of the following:

- entity declaration,
- architecture body,
- configuration declaration,
- package declaration,
- package body.

A design library consists of a number of compiled design units. Design units are further classified as

1. ***Primary* units**: These units allow items to be exported out of the design unit. They are

a. ***entity declaration****:* The items declared in an entity declaration are implicitly visible within the associated architecture bodies.

b. ***package declaration****:* Items declared within a package declaration can be exported to other design units using context clauses.

Secondary units: These units do not allow items declared within them to be exported out of the design unit, that is, these items cannot be referenced in other design units. These are

a. ***architecture body****:* A signal declared in an architecture body, for example, cannot be referenced in other design units.

b. ***package body.***

Package example

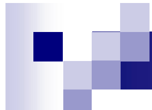
```
library design_lib;

use design_lib. my_package.all;

entity……
```

my_package is a package which is compiled under design_lib library.

Package :

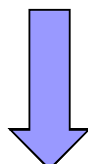
```
Package my_package is

        type sum is(‘May’, ‘June’, ‘July’, Áug’);
        component d_ff

          port (d, clk: in bit; q, out bit);

        end component;
        constant pin_delay: time:=125ns;

End my_package
```

All types combined in a single package body.



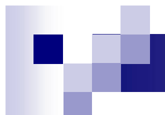
Package example (cont.)

```
library design_lib;
use design_lib. my_package.all;
use design_lib. my_package.d_ff;
use design_lib. my_package.pin_delay;
entity……
```



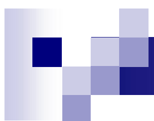
Header file : - a package

```
library IEEE;
use IEEE.std_logic_1164.all;
```



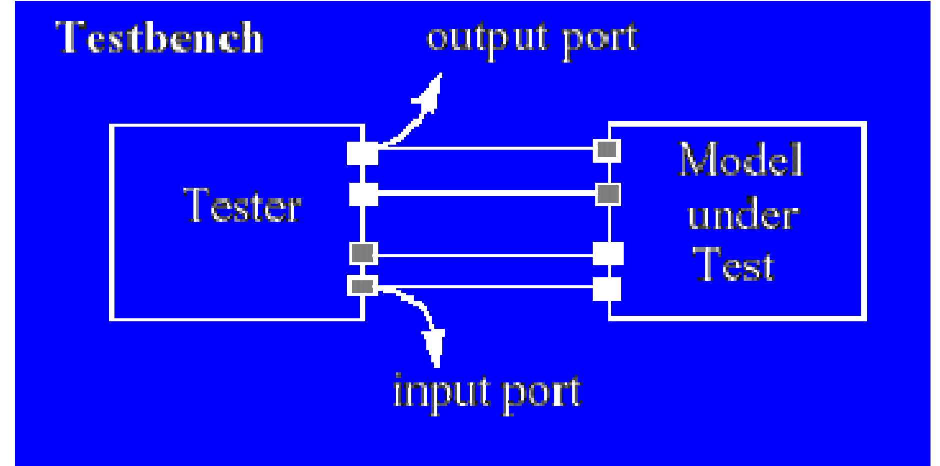
Auto Package

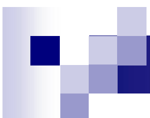
- **The package STANDARD is automatically made visible to all entities.**
- **STANDARD defines:**
  - **Bit**
  - **Bit_Vector**
  - **Boolean**
  - **Integer**
  - **Real**
  - **Character**
  - **String**
  - **Time**



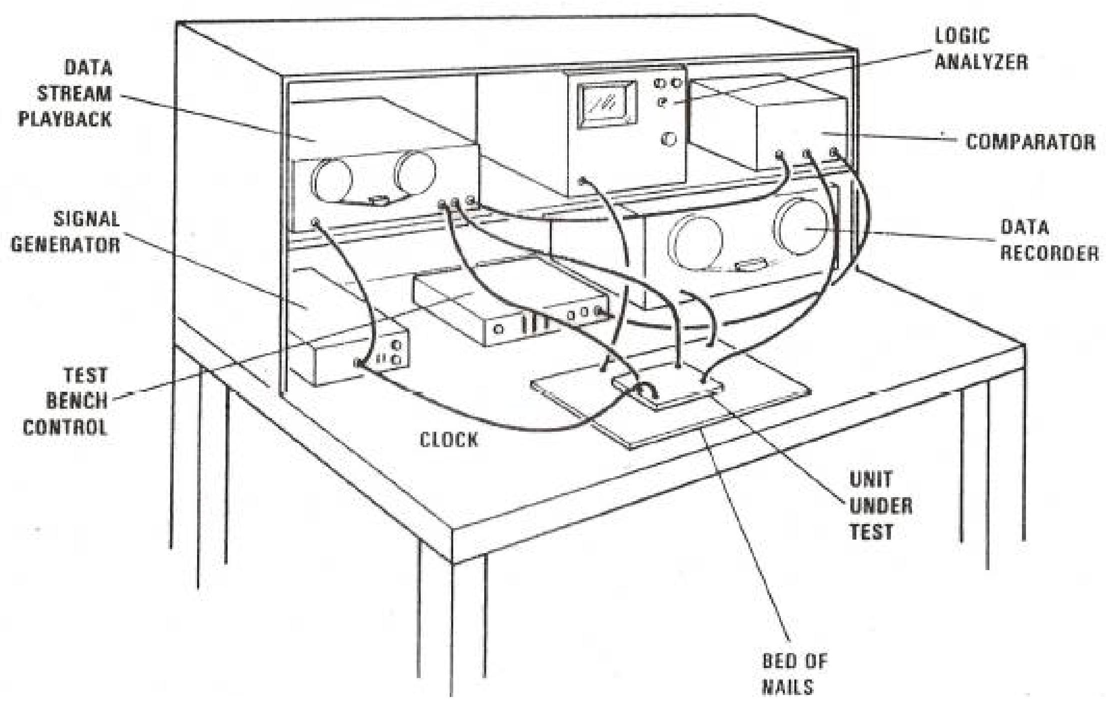
VHDL Test Bench

- VHDL test bench is VHDL code that produces stimuli to test your design correctness
- It can automatically verify accuracy of the VHDL code
  - Given a known input, does the system generate the expected output
- Verifies that the VHDL code meets the circuits specifications
- Test benches should be easily modified, allowing for future use with other code
- Should be Easy to understand the behavior of the test bench

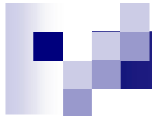




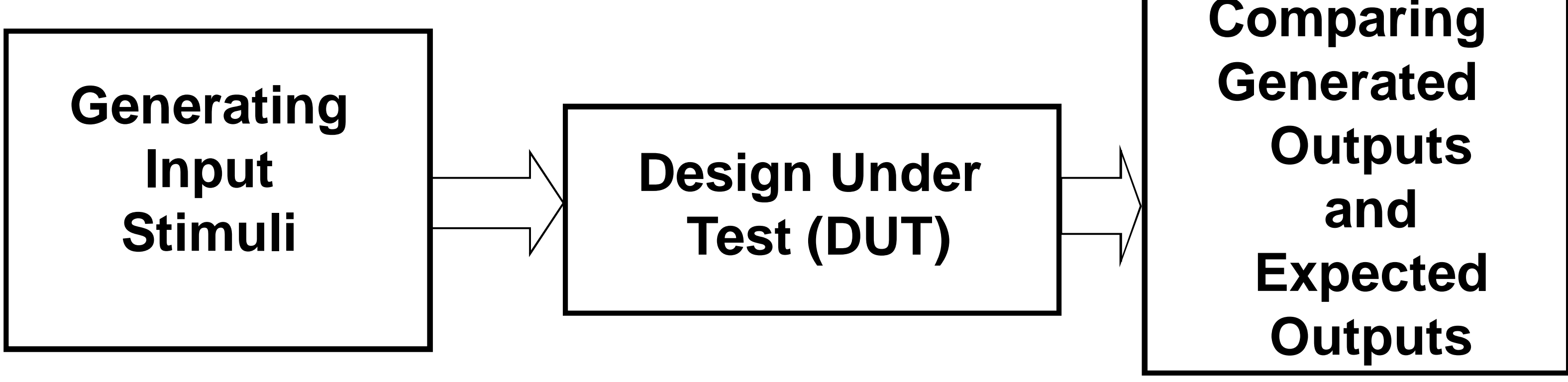
Cont

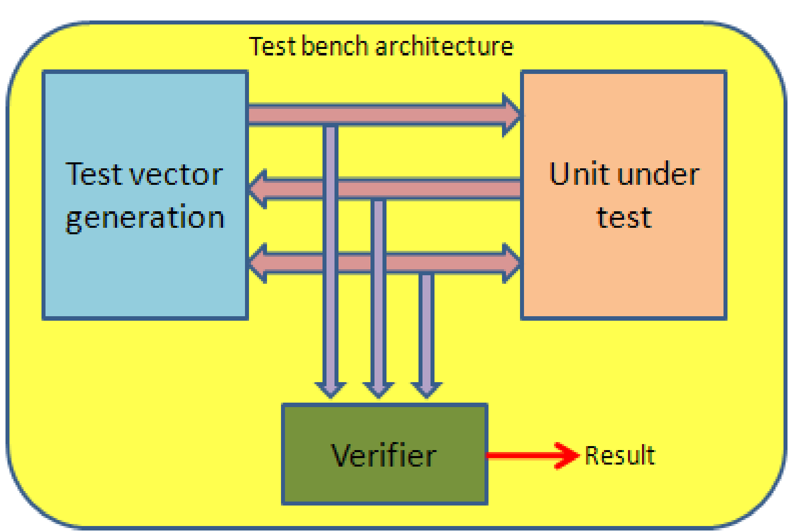


- Test Bench is a program that verifies the functional correctness of the hardware design.
- The test bench program checks whether the hardware model does what it is supposed to do and is not doing what it is not supposed to do.



Cont.





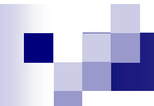
- **Generate stimulus for testing the hardware block.**
- **Apply the stimulus.**
- **Compare the generated outputs against the expected outputs.**

A typical test bench

```
entity entity-name is
end entity-name;
architecture testbench-architecture-
name  of entity-name is
begin
        UUT: UUT instantiation;
        stimulus-gen: process
        begin
            test vector generation;
        end process;
end testbench-architecture-name;
```

Main programme

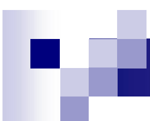
```
library ieee;
use ieee.std_logic_1164.all;

--ENTITY DECLARATION: name, inputs,
outputs
entity andGate is
   port( A, B : in std_logic;
            F : out std_logic);
end andGate;

--FUNCTIONAL DESCRIPTION: how the
AND Gate works
architecture func of andGate is
begin
  F <= A and B;
end func;
```



Test Bench

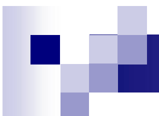
```
library ieee;
use ieee.std_logic_1164.all;

--ENTITY DECLARATION: no inputs, no
outputs
entity andGate_tb is
end andGate_tb;

-- Describe how to test the AND Gate
architecture tb of andGate_tb is
   component andGate is
   port( A, B : in std_logic;
            F : out std_logic);
   end component;

   signal  inA, inB, outF : std_logic;
begin
   --map the testbench signals to the ports
of the andGate
   mapping: andGate port map(inA, inB,
outF);
```

```
Process
      --TEST 1
begin
      inA <= '0';
      inB <= '0';
      wait for 15 ns;

      --TEST 2
      inA <= '0';
      inB <= '1';
      wait for 15 ns;

      --TEST 3
      inA <= '1';
      inB <= '1';
      wait for 15 ns;
end process;

end tb;
```



```vhdl
--include ieee library
library ieee;
--use package std_logic_1164 which contains type declerations for 'std_logic'
use ieee.std_logic_1164.all;
--entity specifies interface to the module
entity gates is
 port(a,b: in std_logic;o_and,o_or,o_xor,o_xnor,o_nor,o_nand: out std_logic);
end gates;
--architecture defines the implementation of entity
architecture gates_arch of gates is
begin
 o_and<=a and b;
 o_or<=a or b;
 o_nor<=a nor b;
 o_nand<=a nand b;
 o_xor<=a xor b;
 o_xnor<=a xnor b;
end gates_arch;
```

```vhdl
 1 library ieee;
 2 use ieee.std_logic_1164.all;
 3 --entity for a test bench should not conatin any port declerations
 4 entity gates_tb is
 5 end gates_tb;
 6 --test bench
 7 architecture gates_tb_arch of gates_tb is
 8 --component declaration of unit under test
 9 component gates is
10  port(a,b: in std_logic;o_and,o_or,o_xor,o_xnor,o_nor,o_nand: out std_logic);
11 end component;
12 --interface signals to the unit under test
13 signal a,b,o_and,o_or,o_nand,o_nor,o_xor,o_xnor:std_logic;
14 begin
15 --instantiate unit under test and map the signals to uut
16  uut:gates port map(a,b,o_and,o_or,o_xor,o_xnor,o_nor,o_nand);
17 --apply different logic values to the input signals
18  tb:process
19  begin
20   a<='0';b<='0';
21   wait for 10 ns;
22   b<='1';
23   wait for 10 ns;
24   b<='0';a<='1';
25   wait for 10 ns;
26   b<='1';
27   wait for 10 ns;
28   wait;--waits for ever
29  end process;
30 end gates_tb_arch;
```



Variations with version

VHDL 93

```
entity flipflop is
    generic (Tprop:delay_length);
    port (clk, d: in bit; q: out bit);
end entity flipflop;
```

VHDL 87

```
entity flipflop
        generic (Tprop: delay_length);
        port (clk, d: in bit; q: out bit);
end flipflop;
```

VHDL 93

architecture name **of** entity-name
is
    (declarations)
begin   (concurrent statements)
end architecture name;

VHDL 87

architecture name **of** entity-name
is
    (declarations)
begin   (concurrent statements)
end architecture name;