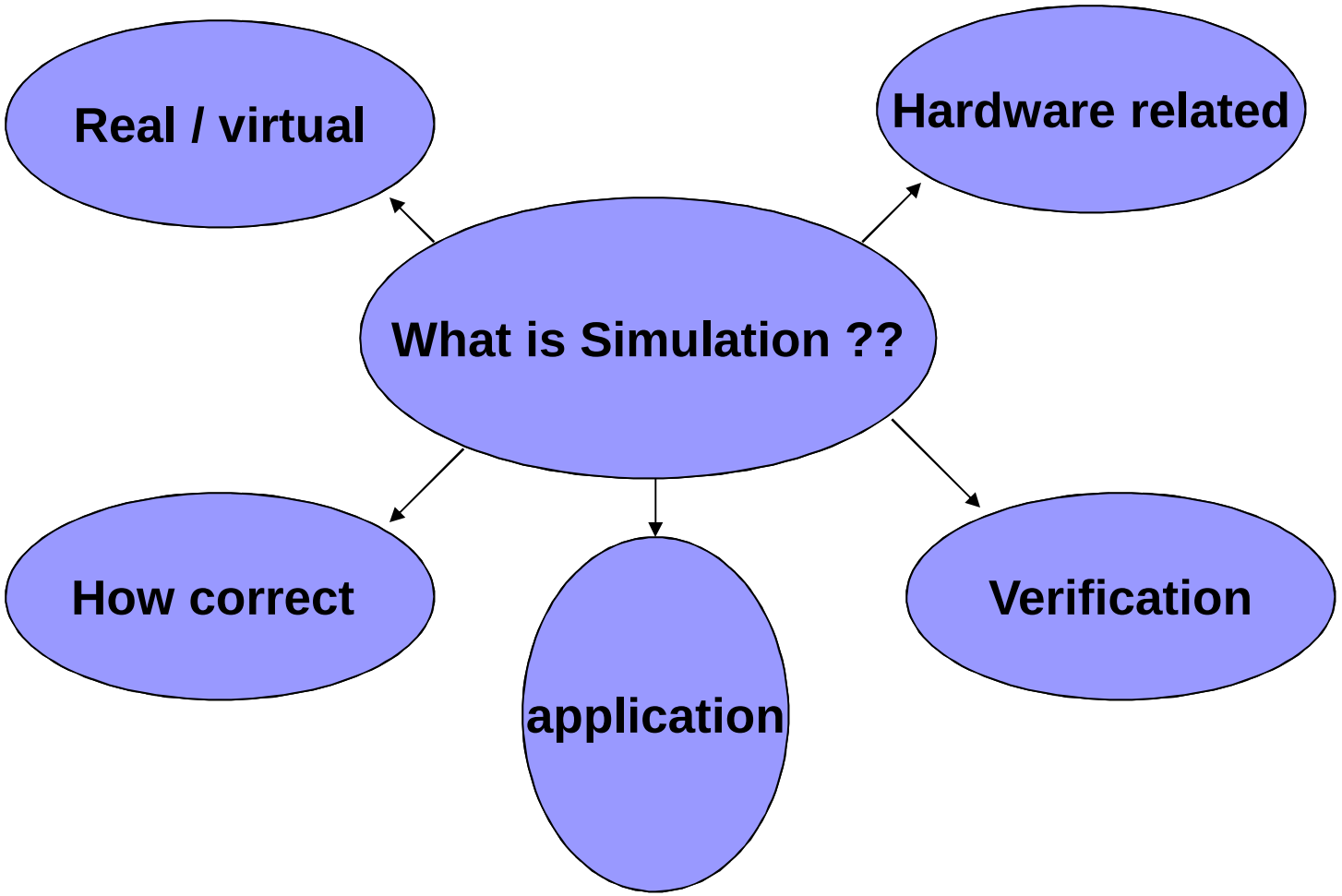
VHDL- Lecture Class-1

Dr. Sayan Chatterjee

Associate Professor

Dept.ETCE, Jadavpur University

Simulation


Real / virtual
Hardware related
What is Simulation ??
How correct
application
Verification

What does HDL stand for?

HDL is short for Hardware Description Language

(**VHDL** – **V**HSIC **H**ardware **D**escription **L**anguage)

(Very High Speed Integrated Circuit)

The purpose of this programming language is to assist circuit designers to describe the characteristics of circuit.

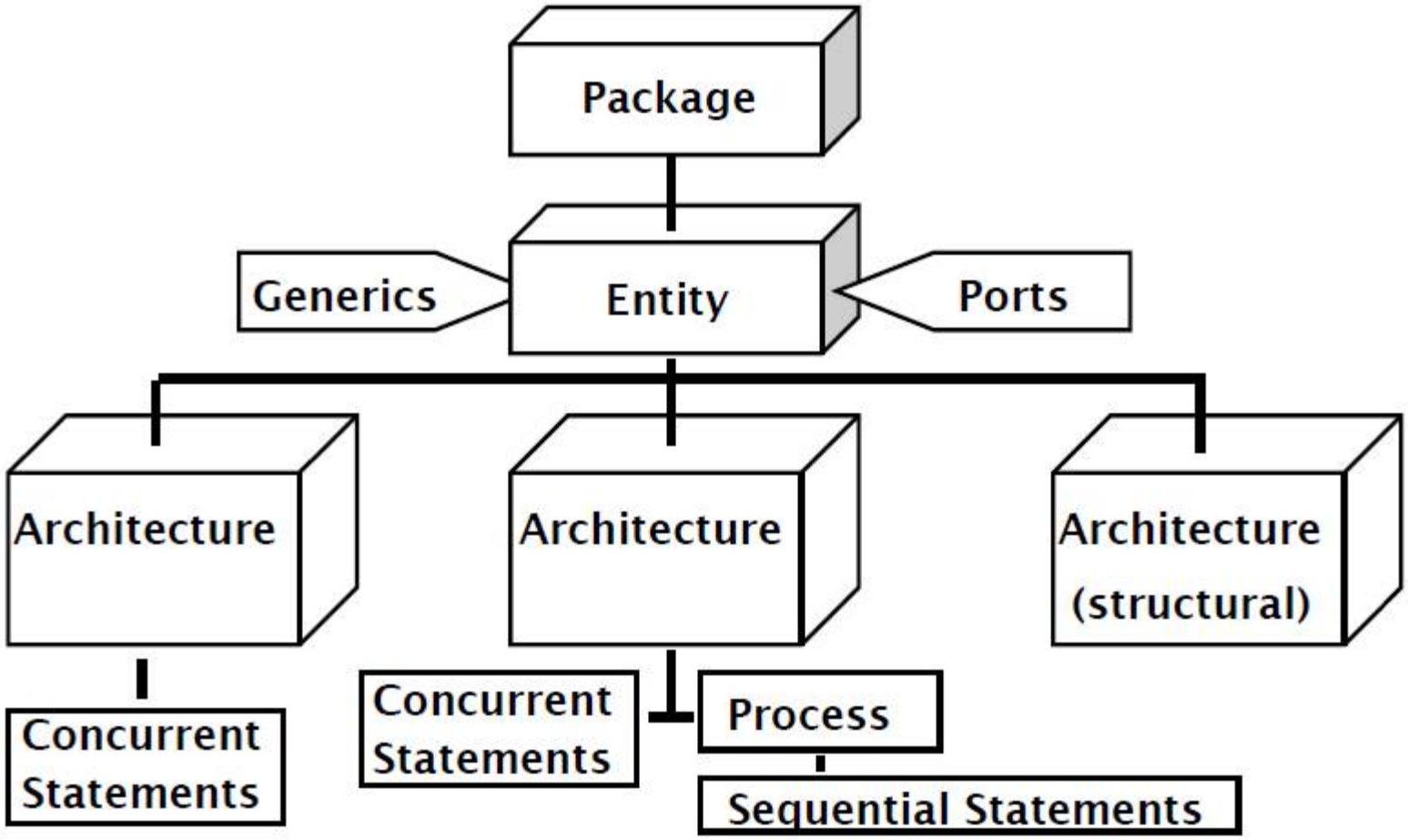
History

- In 1981 under VHSIC programme in US, a number of company is involved for DoD.
- Problem:- Most of the company uses different HDL for their IC. Different code – No matching between them.
- IBM, TI, Intermetrics were first contracted by DoD to develop a single version of language in 1983.
- Version 7.2 VHDl developed in 1985.
- Industry standardization under IEEE in 1986.
- First standard version IEEE Std 1076-1987.

Inside VHDL

- Each section of VHDL code is broken down into two parts
  - The entity
    - which describes the interface and how the component or circuit interacts with the outside world
  - The architecture
    - which describes the function of the component or circuit

VHDL Hierarchy



Library units (also known as design units) are the main components of a VHDL description. They consist of the following kinds of declarations:

- Package (optional)
- Entity
- Architecture
- Configuration (optional)

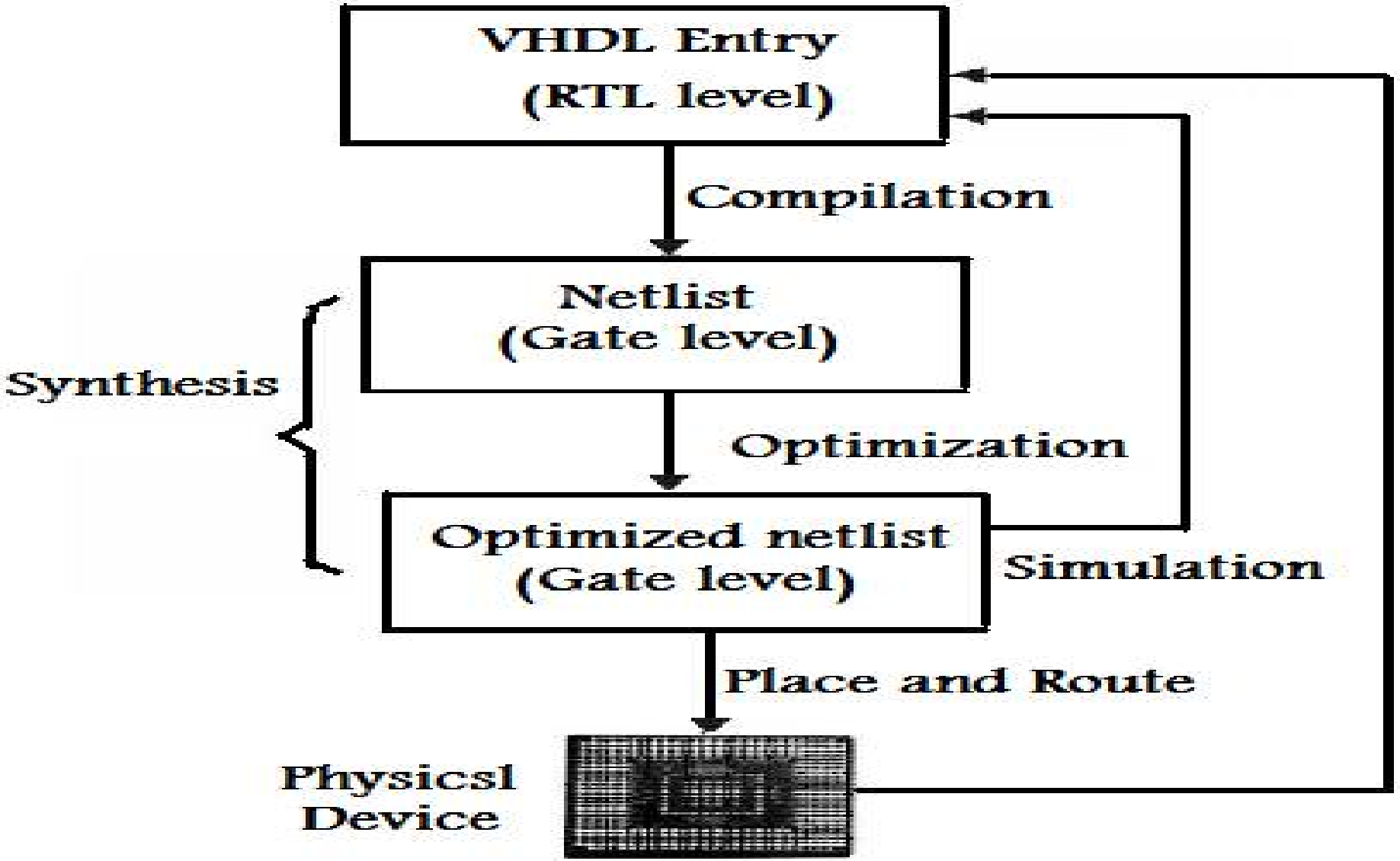
Design Library

A design library is a storage area for previously compiled design units. A design library is completely implementation dependent, meaning that it depends on synthesis tool. Typically, you specify the areas to store frequently used design units such as packages and package bodies for future reference. If you don't do so, the design units will be compiled into a default library called **work**

Most of the synthesis and simulation tools follow the standard definition in VHDL that defines the work library to be only those design units that are included in the source file currently being compiled.

- When you write VHDL descriptions, you write them in a *design file*, then invoke a compiler to analyse them and insert them into a *design library*.
- A number of VHDL constructs may be separately analysed for inclusion in a design library. These constructs are called *library units*.
- The *primary* library units are entity declarations, package declarations and configuration declarations.
- The *secondary* library units are architecture bodies and package bodies.

Desian Flow


VHDL Entry
(RTL level)
Compilation
Netlist
(Gate level)
Synthesis
Optimization
Optimized netlist
(Gate level)
Simulation
Place and Route
Physicsl
Device

VHDL – Header File

(Library / package)

- Include library

  library IEEE;

  - ***Define the library package used***

  use IEEE.STD_LOGIC_1164.all;

  - ***Define the library file used***
  - ***For example, STD_LOGIC_1164 defines '1' as logic high and '0' as logic low***
    - ***output <= '1'; --Assign logic high to output***

Use Statement

To use a design unit from within a library or other design units, a **use** statement must be used specifying the design unit. A **use** statement makes the referenced design unit visible to the working environment.

use ieee.std_logic_1164.**all**;
makes all components in std_logic_1164 library from ieee library visible

use work.func_package.**all**;
makes all components in func_package library from work library visible

use ieee.std_logic-arith.**all**;
makes all components in std_logic_arith library from ieee library visible

use ieee.std_logic_unsigned.**all**;
makes all components in std_logic_unsigned library from ieee library visible

use my_lib.reg_pack.jk;
makes jk flip flop component in reg_pack package from my_lib library visible

use my_lib.data_pack.**all**;
makes all components in data_pack package from my_lib library visible

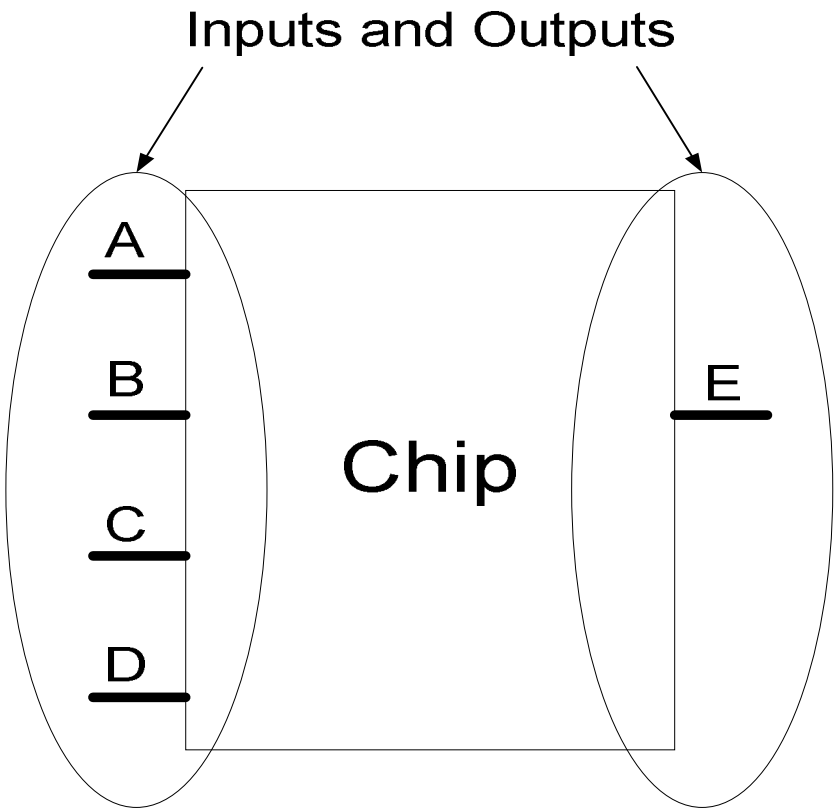
use my_lib.**all**;
makes all components in my_lib library visible

use work.**all**;
makes all components in **work** library visible

Entity

- Define inputs and outputs
- Example:

```
Entity test is
Port( A,B,C,D: in std_logic;
        E: out std_logic);
End test;
```



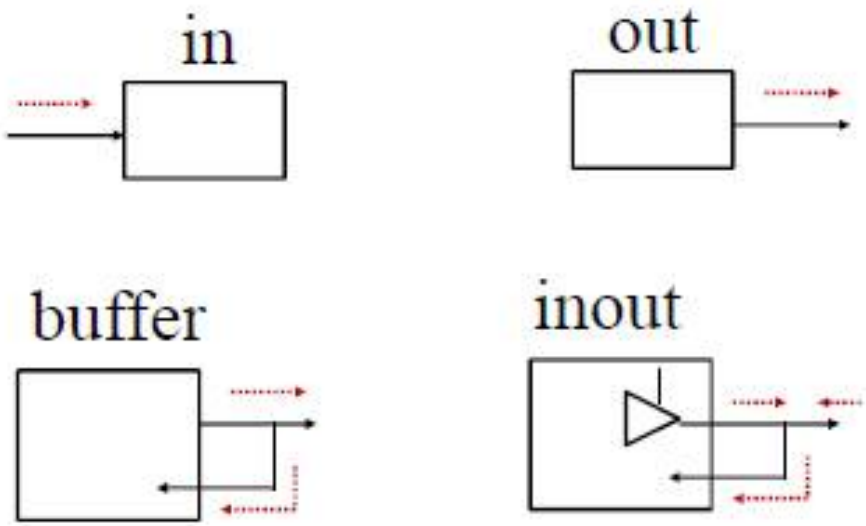
Port declaration

- PORT declaration establishes the interface of the object to the outside world.
- Three parts of the PORT declaration
  - Name
    - Any identifier that is not a reserved word.
  - Mode
    - In, Out, Inout, Buffer
  - Data type
    - Any declared or predefined datatype.
- Sample PORT declaration syntax:

```
ENTITY test IS
    PORT( name : mode data_type);
END test;
```

Port Attributes

bit - '0', '1'
std_logic – '0', '1', 'Z', …

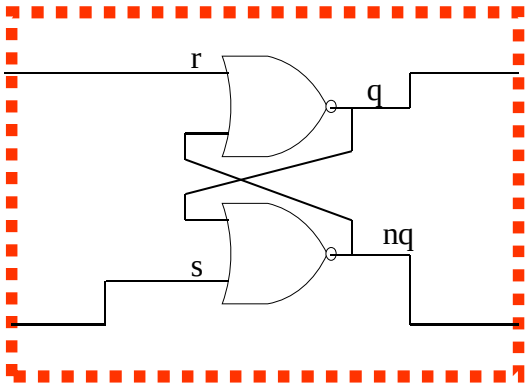


buffer – has restrictions, we use signals (wires) for local feedback

The Architecture

Entity name is “latch”

The architecture syntax



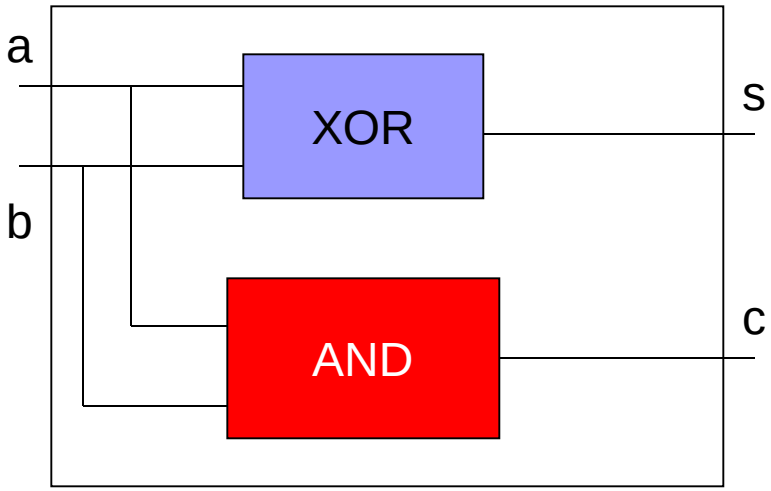
Description of the architecture (what it is)

Entity name architecture belongs to

```
architecture structure of latch is
    begin
        q<= r nor nq;
        nq<= s nor q;
end structure;
```

Description of circuit

A Full Programme- Half Adder




```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity HA is
port (a,b: in BIT; s,c: out BIT);
end HA;

architecture HA_S of HA is
begin
s<= a xor b;
c<= a and b;
end HA_S
```

Description types

- VHDL can be structured in three different ways.
  - Structural
  - Data flow
  - Behavioral
- Usually, a mixture of all three methods are used in the design of the circuit

Data Flow

- Data flow describes how the data flows through the circuit, from input to output
- It uses built in functions to describe the flow of data
- All commands in Data flow are concurrent (occur at the same time)
- Data flow operates in discrete time, when changes occur on the input, it immediately affects the output of the circuit
- This method is like the more traditional way of designing a circuit using gates
- For some traditional hardware designers, it is easier to use the data flow method, since it deals with the traditional method of designing circuits.

Data Flow Syntax

```
entity latch is
    port (s,r: in bit;
          q, nq: out bit);
end latch;
architecture dataflow of latch is
    begin
        q<= r nor nq;
        nq<= s nor q;
end dataflow;
```

Architecture description

describes the architecture is of the method data flow description and belongs to the entity latch

Logical Data Assignment

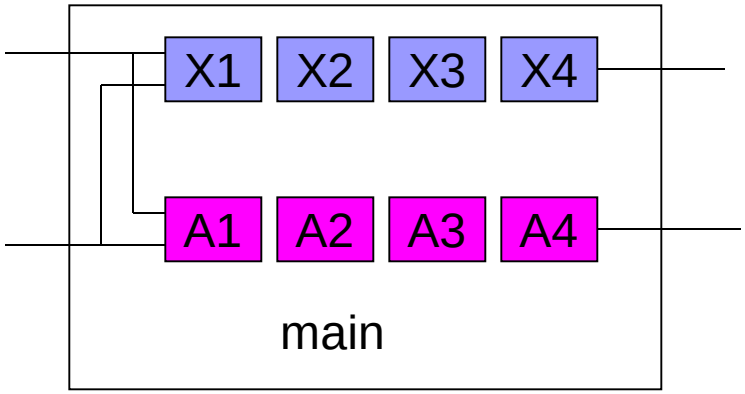
Shows that the value of the output pins are derived from a logical function of the input pins

"<=" is used for a signal assignment, which describes how the data on the right hand side of the operator to the left hand side.

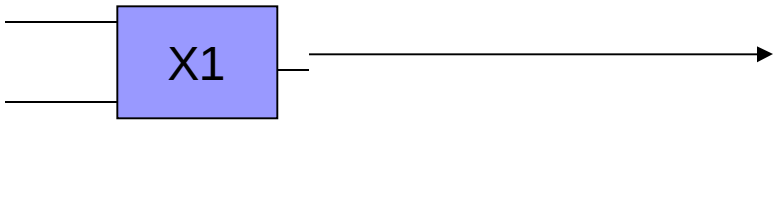
Structural Description

- Structural description uses text to show how components of a circuit are put together
  - similar to a schematic capture approach to designing a circuit
  - It is to combine smaller blocks or predefined components into a larger circuit by describing the way that the blocks interact
- The structural method is similar to a block diagram
  - smaller components are used to make a circuit without knowing what is happening in the block
- It can be thought of as a netlist
  - A netlist is used to describes how components are connected together form a circuit
- Write all parts in separate page and compile each one separately with different name.

Structural Description- Diagram



Header Entity (main) Architecture



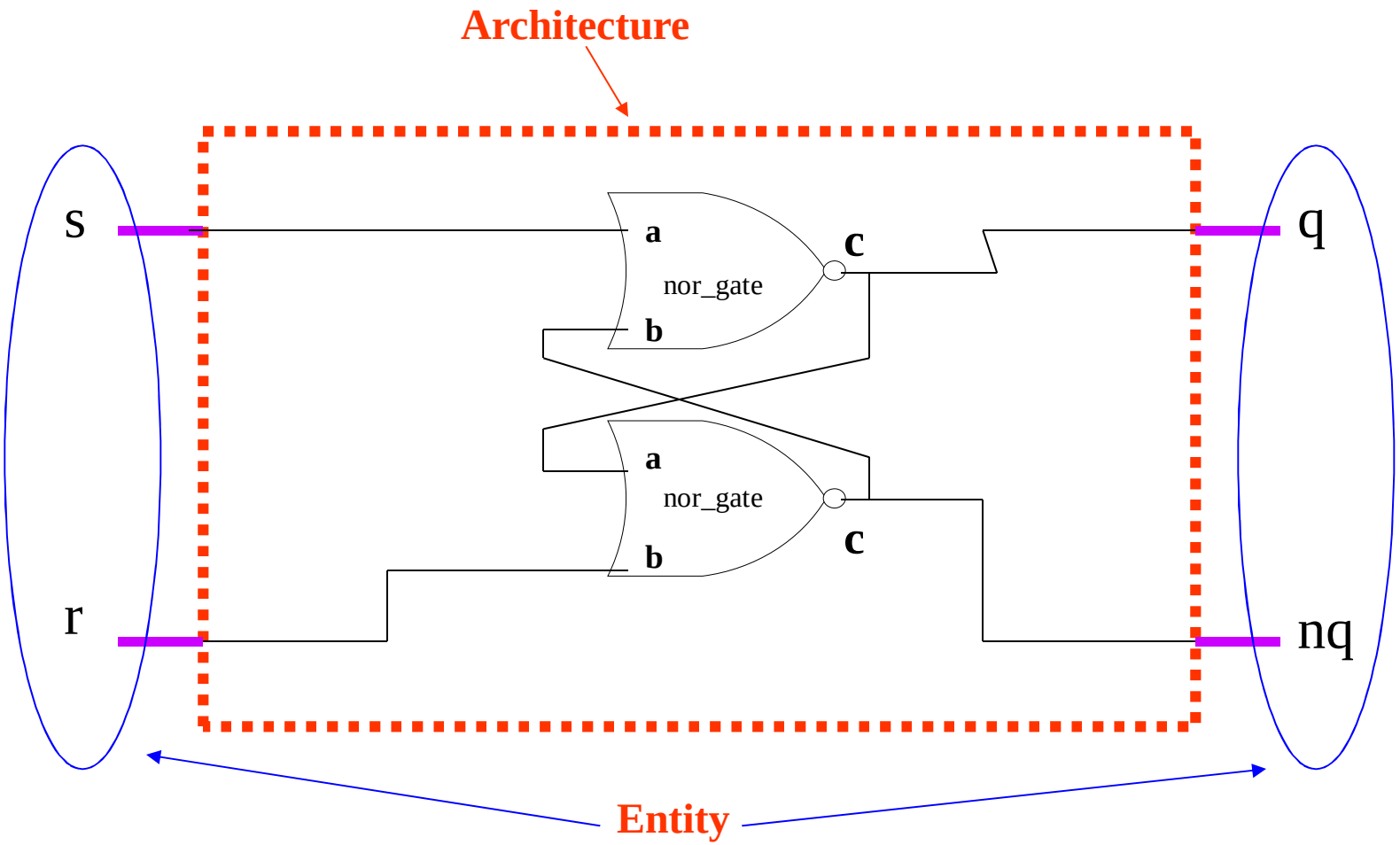
Header Entity (X1) Architecture
Write in separate page

Header Entity (A1) Architecture
Write in separate page

Link between main X1, A1

Port map = 1:1
Port map between input-output port of the A1, X1 to main program

Description Representation



Structural Syntax

```
entity latch is
      port (s,r: in bit;
              q, nq: inout bit);
end latch;
architecture structure of latch is
      component nor_gate
        port (a,b: in bit;
            c: out bit);
      end component;
      begin
        n1: nor_gate port map (s, nq, q);
        n2: nor_gate port map (r, q, nq);
end structure;
```

Component Pin Specifications

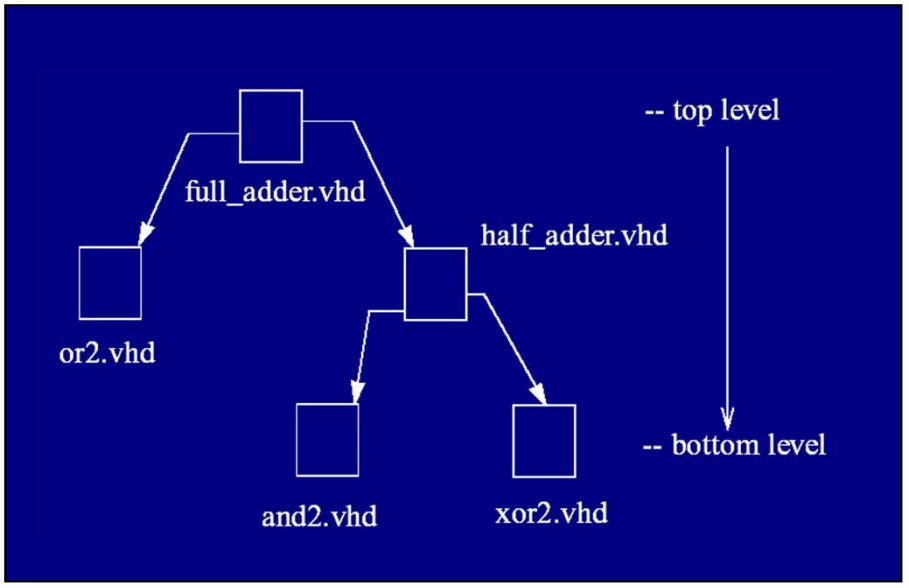
These are used to describe the input and output pins of the component nor_gate that will be used in the architecture section

Mapping pins to component

The command port map is used to show how the input and output pins are connected to the component nor_gate

Advantages of Structural description

- Hierarchy
  - allows for the simplification of the design
- Component Reusability
  - allows the re-use of specific components of the design (Latch, Flip-flops, half-adders, etc)
- Design Independent
  - allows for replacing and testing components without redesigning the circuit



Component Instantiation

•A component instantiation statement defines a sub-component of the entity in which it appears. It associates the signals in the entity with the ports of that sub-component.

•A format of a component instantiation statement**:**

```
Component-Label: Component-Name PORT MAP (association-list);
```

- The ***Component-Label*** can be any legal identifier and can be considered as the name of the instance.
- The ***Component-Name*** must be the name of a component declared earlier using a component declaration.
- The ***association-list,*** associates signals in the entity, called **actuals**, with the ports of a component, called **formals**.

Actuals and Formals

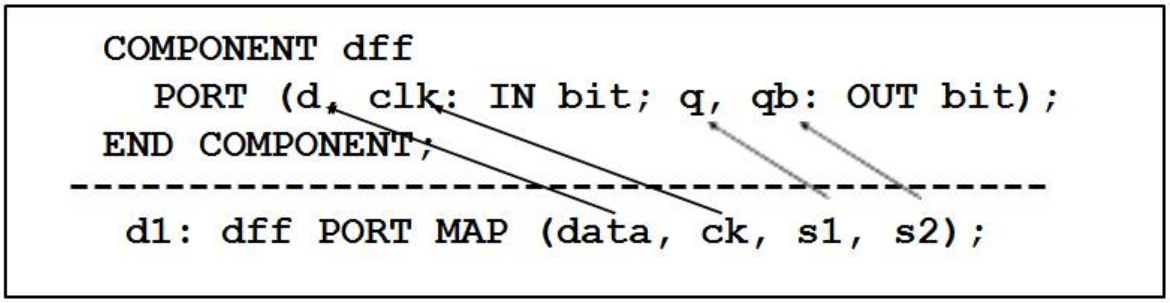
•An actual may be a signal. An actual for an input port may also be an expression.

•An actual may also be the keyword **open** to indicate a port that is not connected**.**

•There are two ways to perform the association of formals with actuals:

1. **Positional association**
2. **Named association**

•In positional association, each actual in the component instantiation is mapped by position with each port in the component declaration. That is, the first port in the component declaration corresponds to the first actual in the component instantiation, the second with the second, and so on.



Actuals and Formals

- If a port in a component instantiation is not connected to any signal, the keyword **OPEN** can be used to signify that the port is not connected.
- For example:

```
  COMPONENT dff
    PORT (d, clk: IN bit:='0'; q, qb: OUT bit);
  END COMPONENT;
-----------------------------------------
  d1: dff  PORT MAP (OPEN, ck, s1, OPEN);
```

- The second input port of the **dff** component is not connected to any signal. An **input** port may be left open only if its declaration specifies an initial value. For the previous component instantiation statement to be legal, port **d** of the component declaration for **dff** must have an initial value expression, while the **output** port **qb** not.
- A port of any other mode may be left unconnected as long as it is not an unconstrained array.

Actuals and Formals

- In named association, an association-list is of the form:
  - $formal_1 => actual_1, formal_2 => actual_2, \ldots formal_n => actual_n$
- For example:

```
  COMPONENT dff
    PORT (d, clk: IN bit; q, qb: OUT bit);
  END COMPONENT;
-----------------------------------------
  d1: dff PORT MAP (clk => ck, d => data, qb => s2, q => s1);
```

- In named association, the ordering of the associations is not important since the mapping between the actuals and formals is explicitly specified.
- An important point to note is that the scope of the formals is restricted to be within the port map part of the instantiation for that component.

Behavioral Descriptions

- Unlike the other two methods for describing the architecture, behavioral description is like a black box approach to modeling a circuit
  - It is designed to do a specific task, how it does it is irrelevant
- It is used to model complex components which are hard to model using basic design elements
- Behavioral is often more powerful and allows for easy implementation of the design
- Most texts they combine both data flow and behavioral descriptions into one

Behavioral description

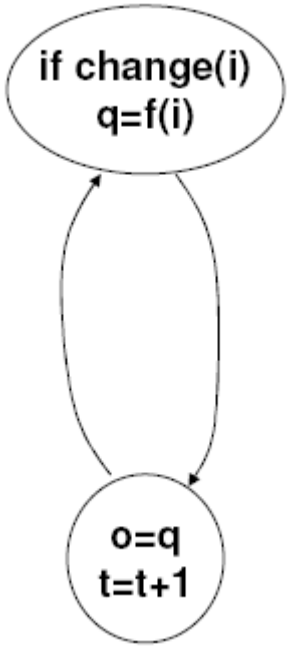
- Behavioral descriptions are supported inside a process statement
- A process is used to describe complex behaviors of the circuit
- The contents of a process can include sequential statements
- These sequential statements are similar to commands in conventional programming languages (it, for, etc) which can only be used in the body of a process statement
- Although, inside a process is sequential, the process itself is concurrent, all processes in a architecture begin execution at the same time
- The process statement is declared in the body of the architecture in the same way as signal assignments in data flow

Cycle-based simulators

Go through all functions using current inputs and compute next output

Update outputs & increase time with 1 delay unit

Event-based Simulators

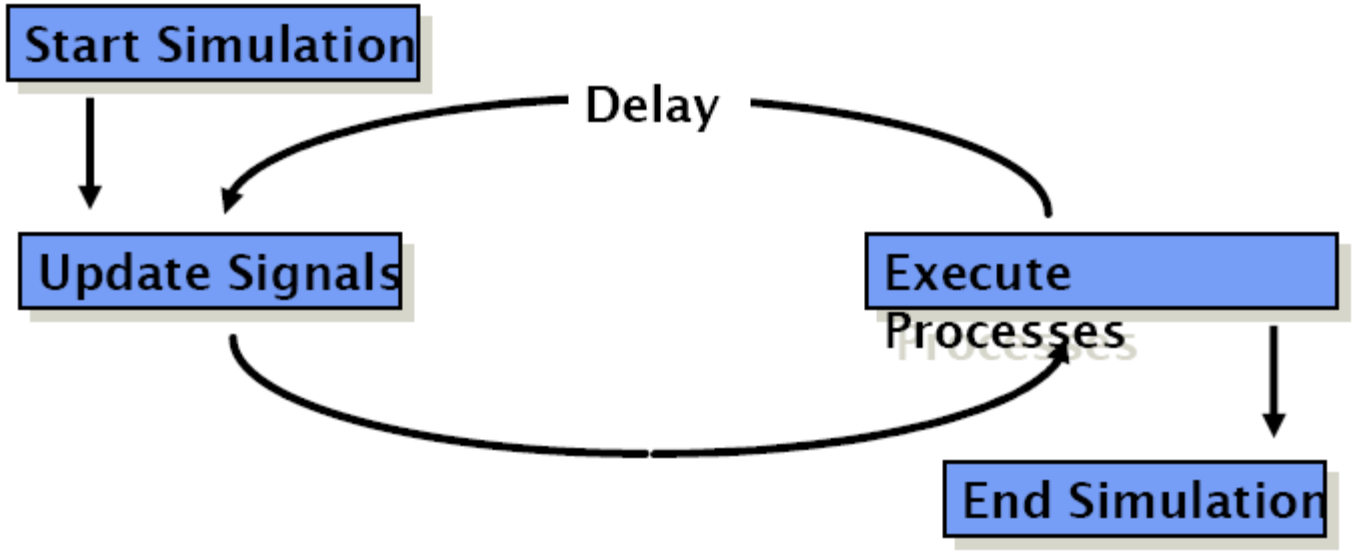


Go through all functions whose inputs has changed and compute next output

Update outputs & increase time with 1 delay unit

VHDL Simulation Cycle

- VHDL uses a simulation cycle to model the stimulus and response nature of digital hardware.



Elements of a Process

- Processes can have a list of signals that they depend on, a sensitivity list, or they can use wait signals to make the process wait for a event to occur (not both)
- They are only execute if the signals in the sensitivity list change
- This makes it critical to ensure that the signals that the process depends on are in the sensitivity list
- Each process is executed once upon power up of the system
- Wait statements are similar to sensitivity lists, but have the advantage of forcing a process to wait at any point within the process, not just the beginning.

Process syntax of a counter

Process label (optional)

Process is dependent only on variable "x"

```
count: process (x)
  variable cnt : integer := -1;
begin
  cnt:=cnt + 1;
end process
```

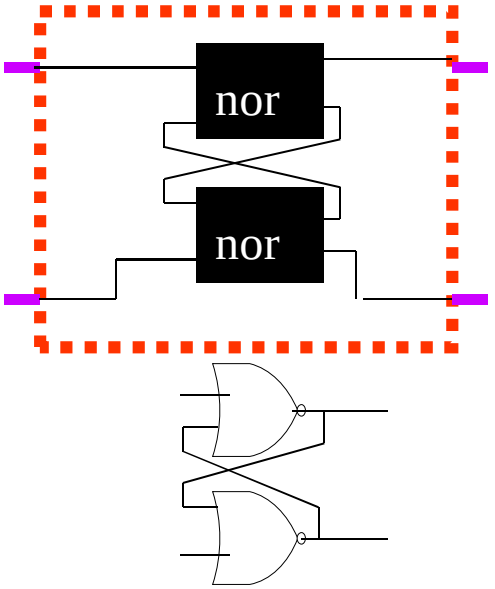
Variable is set to a value of -1 since the process is run once at startup to bring the value to 0

Increments the variable "cnt" every time the signal of "x" changes

Behavioral vs. Structural vs. Data flow

Structural

How components are put together



Data Flow

Describes how data flows from input to output

Behavioral

Describes the behavior of the circuit

within a process

When an event occur on sensitivity list (r,s) then process statement activated to do work under begin and end process.

```
process (r,s)
begin
  if (r nor nq) then
     q <= '1';
  else
     q <= '0';
  endif
  end process
```

Sensitivity-lists vs Wait-on - statement

```
Summation:
  PROCESS( A, B, Cin)
  BEGIN
    Sum <= A xor B xor Cin;
END PROCESS Summation;
```

=

```
Summation:  PROCESS
  BEGIN
    Sum <= A xor B xor Cin;
       WAIT ON A, B, Cin;
END PROCESS Summation;
```

if you put a sensitivity list in a process, you can't have a wait statement!

if you put a wait statement in a process, you can't have a sensitivity list!

For a process to model combinational logic, it must contain in the sensitivity list all the signals that are inputs of the process. In other words, the process must be reevaluated every time one of the inputs to the circuit it models changes. In this way combinational logic is correctly modeled.

If a process is not sensitive to all its inputs and it is not a sequential process, then it cannot be synthesized, since there is no hardware equivalent of such a process. Not all synthesis tools enforce such a rule, so great care should be taken in the design of combinational processes in order not to introduce errors in the design. Such errors will cause subtle differences between the simulated model and the circuit obtained by synthesis, because a non-combinational process is interpreted by the synthesizer as a combinational circuit.

WAIT – statement Syntax

- The wait statement causes the suspension of a process statement or a procedure.
- wait [sensitivity_clause] [condition_clause] [timeout_clause];
  - Sensitivity_clause ::= on signal_name

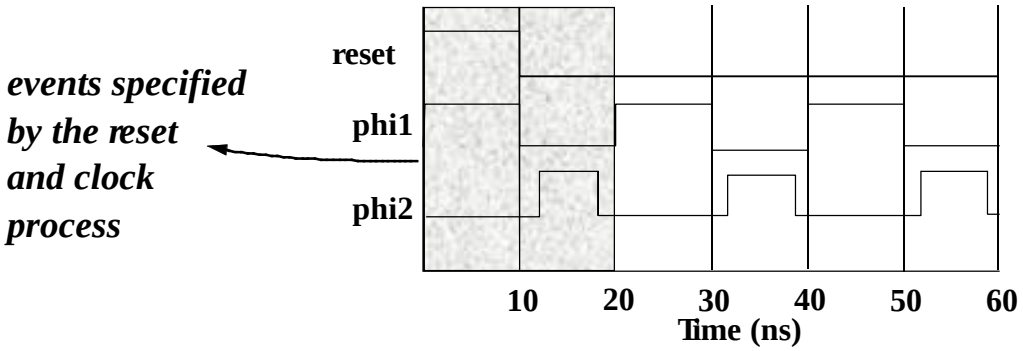
    ```
    wait  on  CLOCK;
    ```
  - Condition_clause ::= until boolean_expression

    ```
    wait  until  Clock = '1';
    ```
  - Timeout_clause ::= for time_expression

    ```
    wait  for  150 ns;
    ```

The Wait Statement: Waveform Generation

```
library IEEE;
use IEEE.std_logic_1164.all;
entity two_phase is
port(phi1, phi2, reset: out std_logic);
end entity two_phase;
architecture behavioral of two_phase is
begin
rproc: reset <= '1', '0' after 10 ns;
clock_process: process
begin
phi1 <= '1', '0' after 10 ns;
phi2 <= '0', '1' after 12 ns, '0' after
18 ns;
wait for 20 ns;
end process clock_process;
end architecture behavioral;
```



- Note the "perpetual" behavior of processes

Process statement use

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
-- entity
entity my_xor is
port ( A,B : in  std_logic;
       F   : out std_logic);
end my_xor;
-- architecture
architecture dataflow of my_xor is
begin
    F <= A XOR B;
end dataflow;
--
--
--
```

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
-- entity
entity my_xor is
port ( A,B : in  std_logic;
       F   : out std_logic);
end my_xor;
-- architecture
architecture behav of my_xor is
begin
  xor_proc: process(A,B) is
  begin
    F <= A XOR B;
  end process xor_proc;
end behav;
```

Checkpoints
On A , B

Data Objects

- There are three types of data objects:
  - Signals
    - Can be considered as wires in a schematic.
    - Can have current value and future values.
  - Variables and Constants
    - Used to model the behavior of a circuit.
    - Used in processes, procedures and functions.

Constant Declaration

- A constant can have a single value of a given type.
- A constant's value cannot be changed during the simulation.
- Constants declared at the start of an architecture can be used anywhere in the architecture.
- Constants declared in a process can only be used inside the specific process.

CONSTANT ***constant_name*** **:** ***type_name*** **[: = value];**

```
CONSTANT rise_fall_time : TIME : = 2 ns;
CONSTANT data_bus : INTEGER : = 16;
```

Variable Declaration

- Variables are used for local storage of data.
- Variables are generally not available to multiple components or processes.
- All variable assignments take place immediately.
- Variables are more convenient than signals for the storage of (temporary) data.
- Variables can only be declared and used inside process

```
VARIABLE variable_name : type_name [:=value];

VARIABLE opcode : BIT_VECTOR(3 DOWNTO 0) := "0000";
VARIABLE freq : INTEGER;
```

Signal Declaration

- Signals are used for communication between components.
- Signals are declared outside the process.
- Signals can be seen as real, physical signals.
- Some delay must be incurred in a signal assignment.

```
SIGNAL signal_name : type_name [:=value];

SIGNAL brdy : BIT;
SIGNAL output : INTEGER := 2;
```

Key Difference

```
ARCHITECTURE signals OF test IS
   SIGNAL a, b, c, out_1, out_2: BIT;
BEGIN
   out_1 <= a NAND b;
   out_2 <= out_1 XOR c;
END signals;
```

Time	a	b	c	out_1	out_2
0	0	1	1	1	0
1	1	1	1	1	0
1+d	1	1	1	0	0
1+2d	1	1	1	0	1

```
ARCHITECTURE variables OF test IS
BEGIN
   PROCESS (a, b, c)
   VARIABLE a,b,c,out_3,out_4: BIT;
   BEGIN
     out_3 := a NAND b;
     out_4 := out_3 XOR c;
   END PROCESS;
END example;
```

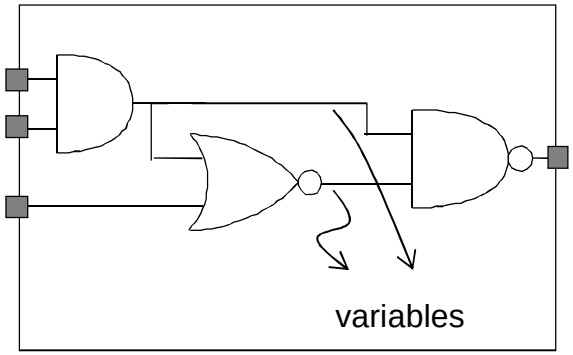
Time	a	b	c	out_3	out_4
0	0	1	1	1	0
1	1	1	1	0	1

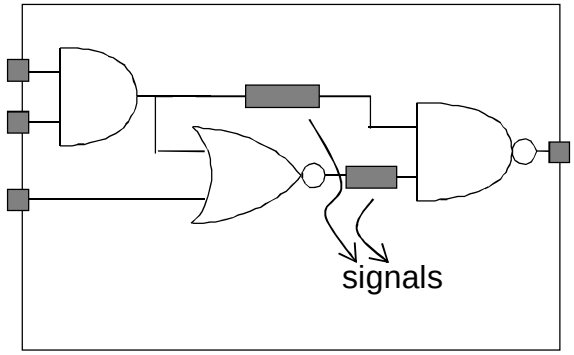
1. **Variable = instantaneous , signal = delayed.**
2. **Many cases variables are internal to architecture which can not be accessible from outside in a single chip.**

Variables vs. Signals: Example

```
proc1: process (x, y, z) is -- Process 1
variable var_s1, var_s2: std_logic;
begin
L1: var_s1 := x and y;
L2: var_s2 := var_s1 xor z;
L3: res1 <= var_s1 nand var_s2;
end process;
```

```
proc2: process (x, y, z) -- Process 2
begin
L1: sig_s1 <= x and y;
L2: sig_s2 <= sig_s1 xor z;
L3: res2 <= sig_s1 nand sig_s2;
end process;
```





- **Distinction between the use of variables vs. signals**
  - **Computing values vs. computing time-value pairs**
  - **Remember event ordering and delta delays!**

Signal Scope

- F3 = (L/ M/ N) + LM


```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
-- entity
entity my_ckt_f3 is
port ( L,M,N : in  std_logic;
       F3    : out std_logic);
end my_ckt_f3;
-- architecture
architecture f3_2 of my_ckt_f3 is
begin
F3<=((NOT L)AND(NOT M)AND N)OR(L AND M);
end f3_2;
```

Intermediate signal

- An alternative solution to above problem is provided in next figure. This example represents a massively important concept in VHDL. The special statements are used to provide what is often referred to as intermediate results. This approach is equivalent to declaring extra variables
- The intermediate signals must be declared within the body of the architecture because they have no link to the outside world and thus do not appear in the entity declaration. Note that the intermediate signals are declared in the architecture body but appear before the begin statement.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
-- entity
entity my_ckt_f3 is
port ( L,M,N : in  std_logic;
       F3    : out std_logic);
end my_ckt_f3;
-- architecture
architecture f3_1 of my_ckt_f3 is
    signal A1, A2 : std_logic; -- intermediate signals
begin
    A1 <= ((NOT L) AND (NOT M) AND N);
    A2 <= L AND M;
    F3 <= A1 OR A2;
end f3_1;
```

- Although the approach of using intermediate signals is not mandatory for this example, their use brings up some good points.
- As circuits become more complex, there are many occasions in which intermediate signals must be used.
- However both the cases all the statements are concurrent signal assignment statements.

Conditional Signal Assignment

- The term conditional signal assignment is used to describe statements that have only one target but can have more than one associated expression assigned to the target.
- Each of the expressions is associated with a certain condition. The individual conditions are evaluated sequentially in the conditional signal assignment statement until the first condition evaluates as true.
- Only one assignment is applied per assignment statement.

```
<target> <= <expression> when <condition> else
            <expression> when <condition> else
            <expression>;
```

```
architecture f3_3 of my_ckt_f3 is
begin
F3 <= '1' when (L= '0' AND M = '0' AND N = '1') else
      '1' when (L= '1' AND M = '1') else
      '0';
end f3_3;
```

Signal Assignment with time

Each signal has associated with it a *projected output waveform*, which is a list of transactions giving future values for the signal. A signal assignment adds transactions to this waveform. So, for example, the signal assignment:

```
s <= '0' after 10 ns;
```

Suppose then at time 16 ns, the assignment:

```
s <= '1' after 4 ns, '0' after 20 ns;
```

were executed. The two new transactions are added to the projected output waveform:

Data Types

- Every data object in VHDL can hold a value that belongs to a set of values- Type declaration.
- A type is a name that has associated with its set of values and set of operations.
- For example Integer is a predefined type.
- All the predefined type are contained in the package – Standard.

Integer declaration

- Integer
  - Minimum range for any implementation as defined by standard: -2,147,483,647 to 2,147,483,647
  - Integer assignment example

```
ARCHITECTURE test_int OF test IS
BEGIN
    PROCESS (X)
      VARIABLE a: INTEGER;
    BEGIN
      a := 1;   -- OK
      a := -1;   -- OK
      a := 1.0;   -- bad
    END PROCESS;
END TEST;
```

Real declaration

- Real
  - Minimum range for any implementation as defined by standard: -1.0E38 to 1.0E38
  - Real assignment example

```
ARCHITECTURE test_real OF test IS
BEGIN
    PROCESS (X)
      VARIABLE a: REAL;
    BEGIN
      a := 1.3;   -- OK
      a := -7.5;   -- OK
      a := 1;   -- bad
      a := 1.7E13;   --OK
      a := 5.3 ns;   -- bad
    END PROCESS;
END TEST;
```

Sub Types

- It is a type with constraint
- Constraints specify the subset of values for subtypes.
- Type == base type

```
subtype my_int is integer range 48 to 156.
```

Enumerated Type declaration

type M_ex is(‘U’, ‘O’, ‘1’, ‘Z’);

type micro_ex is(‘LO’, ‘ST’, ‘ADD’, ‘SUB’,’MUL’);
subtype arith is micro_ex range ADD to MUL;

M_ex is a enumeration type that has the set of ordered values.

Arith is a subtype of base type micro_ex. The range of subtype is specified.

type std_ulogic is ('U','X','1','0','Z','W','H','L','-');

Value	Interpretation
U	Uninitialized
X	Forcing Unknown
0	Forcing 0
1	Forcing 1
Z	High Impedance
W	Weak Unknown
L	Weak 0
H	Weak 1
-	Don’t Care

The Std_ulogic and Std_logic Data Types

Std_ulogic (which is the base type of the more-commonly used resolved type std_logic) is a data type defined by IEEE standard 1164, and defined in the file ieee.vhd. Std_ulogic is an enumerated type, and has the following definition (from ieee.vhd):

```
type std_ulogic is (
'U',                -- Uninitialized
'X',                -- Forcing Unknown
'0',                -- Forcing 0
'1',                -- Forcing 1
'Z'                 -- High Impedance
'W'                 -- Weak Unknown
'L'                 -- Weak 0
'H'                 -- Weak 1
'-'                 -- Don't care
);
```

The std_ulogic (or std_logic) data type is very important for both simulation and synthesis. Std_logic includes values that allow you to accurately simulate such circuit conditions as unknowns and high-impedance states. For synthesis purposes, the high-impedance and don't-care values provide a convenient and easily recognizable way to represent three-state enables and don't-care logic. For synthesis, only the values 0 , 1 , Z , and - have meaning and are supported. The version of std_logic_1164 defined in the file ieee.vhd includes an enum_encoding attribute that results in each object of type std_ulogic or std_logic being synthesized into a single wire.

Physical Type/ subtype

- Physical
  - Can be user defined range
  - Physical type example

```
TYPE resistance IS RANGE 0 to 1000000

UNITS
    ohm;   -- ohm
    Kohm = 1000 ohm;   -- 1 KΩ
    Mohm = 1000 kohm;   -- 1 MΩ
END UNITS;
```

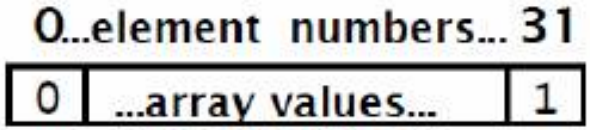
- Time units are the only predefined physical type in VHDL.
- Subtype example:

subtype resist is resistance range 10Kohm to 1Mohm;

Array Type

- Array
  - Used to collect one or more elements of a similar type in a single construct.
  - Elements can be any VHDL data type.

```
TYPE data_bus IS ARRAY (0 TO 31) OF BIT;
```




```
VARIABLE X:  data_bus;
VARIABLE Y:  BIT

Y := X(12);  -- Y gets value of 12th element
```

```
TYPE register IS ARRAY (15 DOWNTO 0) OF BIT;
```

ATTRIBUTES

- Attributes are a feature of VHDL that allow you to extract additional information about an object (such as a signal, variable or type).
- Attributes also allow you to assign additional information (such as data related to synthesis) to objects in your design description.
- An attribute is a value, function, type, range, signal, or a constant.

There are two classes of attributes:

- **Predefined Attributes** : Those that are predefined as a part of the 1076 standard.
- **User-Defined Attributes**: Those that have been introduced outside of the standard.

Predefined Attributes

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

Value Attributes

Types and objects declared in a VHDL description can have additional information, called attributes, associated with them. There are a number of standard pre-defined attributes, and some of those for types and arrays

thing'attr

Firstly, for any scalar type or subtype T, the following attributes can be used:

Attribute	Result
T'left	Left bound of T
T'right	Right bound of T
T'low	Lower bound of T
T'high	Upper bound of T

For an ascending range, T'left = T'low, and T'right = T'high. For a descending range, T'left = T'high, and T'right = T'low.

Secondly, for any discrete or physical type or subtype T, X a member of T, and N an integer, the following attributes can be used:

Attribute	Result
T'pos(X)	Position number of X in T
T'val(N)	Value at position N in T
T'leftof(X)	Value in T which is one position left from X
T'rightof(X)	Value in T which is one position right from X
T'pred(X)	Value in T which is one position lower than X
T'succ(X)	Value in T which is one position higher than X

For an ascending range, T'leftof(X) = T'pred(X), and T'rightof(X) = T'succ(X). For a descending range, T'leftof(X) = T'succ(X), and T'rightof(X) = T'pred(X).

Thirdly, for any array type or object A, and N an integer between 1 and the number of dimensions of A, the following attributes can be used:

Attribute	Result
A'left(N)	Left bound of index range of dim'n N of A
A'right(N)	Right bound of index range of dim'n N of A

type ALLOWED_VALUE is range 31 downto 0;

Then the following equivalence relations are true:

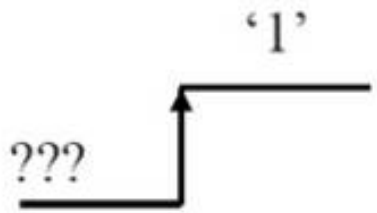
- ALLOWED_VALUE'LEFT = 31
- ALLOWED_VALUE'HIGH = 31
- ALLOWED_VALUE'RIGHT **= 0**
- ALLOWED_VALUE'LOW **=** ALLOWED_VALUE'RIGHT

type STATUS is (SILENT, SEND, RECEIVE);
subtype DELAY_TIME is TIME range 10 ns to 50 ns;

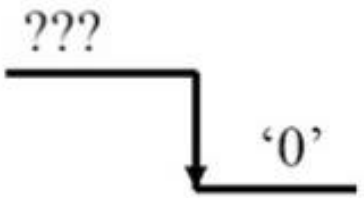
Then

- STATUS'POS(SEND) = 1
- DELAY_TIME'SUCC(21 ns) = 22 ns
- DELAY_TIME'PRED(10 ns) is an error

Examples of Using Attributes: Detecting Edges



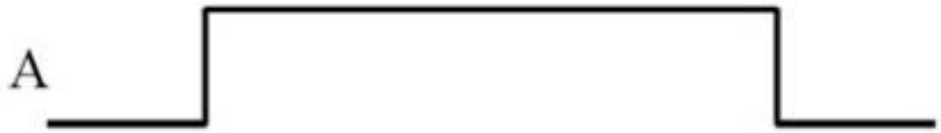
if (A'event and A = '1') then ....
-- rising edge



if (A'event and A = '0') then ....
-- falling edge

We use attribute
event

Measuring Pulse Width



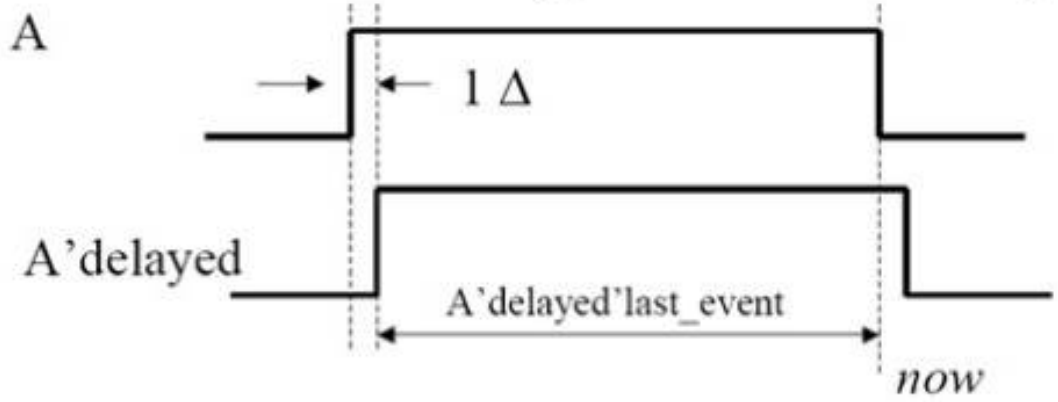

```
process (A)
variable  pw :time;
begin
  if (A = '0') then
     pw :=  A'last_event;
  ...........
```

Returns 0!!!!

'last_event returns elapsed time from last event. The last event triggered this process!!!

Measuring Pulse Width (cont)

Measuring Pulse Width (cont)




```
process (A)
variable  pw :time;
begin|
  if (A = '0') then
     pw :=  A'delayed'last_event;
  ..........
```

Returns pulse width.
'*delayed* returns *A* delayed by 1 delta.

'*last_event* returns elapsed time between now and last event.

Built-In Operators

- Logic operators
  - AND, OR, NAND, NOR, XOR, XNOR (XNOR in VHDL'93 only!!)
  - nand and nor are not associative. (a nand b nand c is not possible)
- Relational operators
  - =, /=, <, <=, >, >=
  - All operators are predefined Boolean type)
- Addition operators
  - +, -, &
- Multiplication operators
  - *, /, mod, rem
- Shift operators
  - **Sll, srl, sla, sra, rol, ror.**

Generic

- Provides certain types of information to be added inside entity.
  - Rise and fall delays
  - Size of interface ports
- Declares a constant object type
- Constant value can be declared at runtime

A Generic NAND Gate

```
entity NAND_GATE is
  generic (N: Natural := 2;
           D: Time := 10 ns);
  port (A: in Bit_Vector (1 to N);
        Z: out Bit);
end NAND_GATE;
```

Generic Map

```
component MY_NAND_GATE
   generic (N: Natural := 2; D: Time := 10 ns);
   port (A: in Bit_Vector (1 to N);  Z: out Bit);
 end component;
-------------------------------------------------------------------
G1: MY_NAND_GATE
   port map (A(1) => A, A(2) => B, Z => T1);
G2: MY_NAND_GATE
   generic map (3, 15 ns)
   port map (A(1) => C, A(2) => D, A(3) => E, Z => T2);
G3: MY_NAND_GATE
   generic map (D => 20 ns, N => 2)
   port map (A(1) => T1, A(2) => T2, Z => Z  OUT);
```

Note the lack of a ";" after the generic map!

- **Generic information is static--it can't be changed during the simulation**
  - **I.e. You can't have the simulation calculate the value that is going to be passed to the generic...**
  - **Specified at compile time.**

- **Generic value is instance-specific**
  - **Different instances of the same component can have different values.**

Example (n input and gate)

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
entity andgate is
        generic (n: natural)
        port (a: in BIT_VECTOR(1 to n); z: out BIT);
end andgate;
  architecture generaicand of andgate is
  begin
        process (a)
                variable and_out:BIT;
                begin
                and_out:='1';
                for k in 1 to n loop
                        and_out:=and_out and a(k);
                        exit when and_out='0';
                end loop;
                z<= and_out;
        end process;
end generaicand
```

In this program size of input port is constant type and modeled as generic. Hence an entire class of andgate is modeled having variable no of input.

Generic (cont.)

Declarion in entity

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
entity andgate is
        generic (n: integer := 2)
        port (a: in BIT_VECTOR(1 to n); z: out BIT);
end andgate;
```

Declarion in component

```
component andgate
        generic (n: integer := 2)
        port (a: in BIT_VECTOR(1 to n); z: out BIT);
end component;
```

Null Transaction

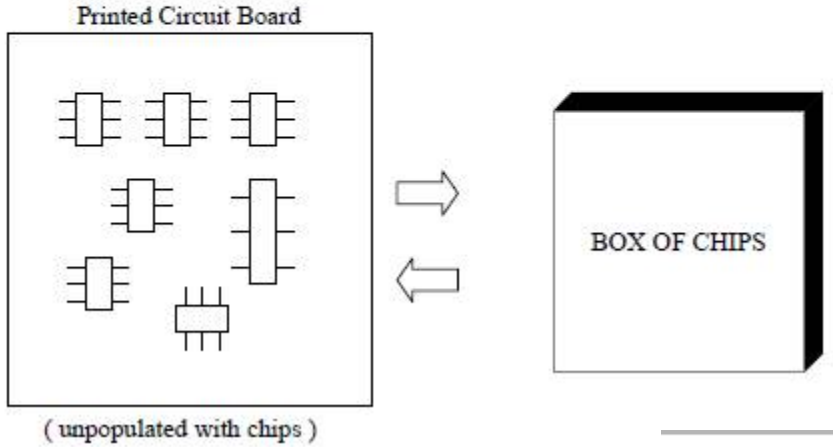
- VHDL provides a facility to model outputs which may be turned off.
- A signal assignment may specify that no value is to be assigned to a resolved signal, that is, that the driver should be disconnected.

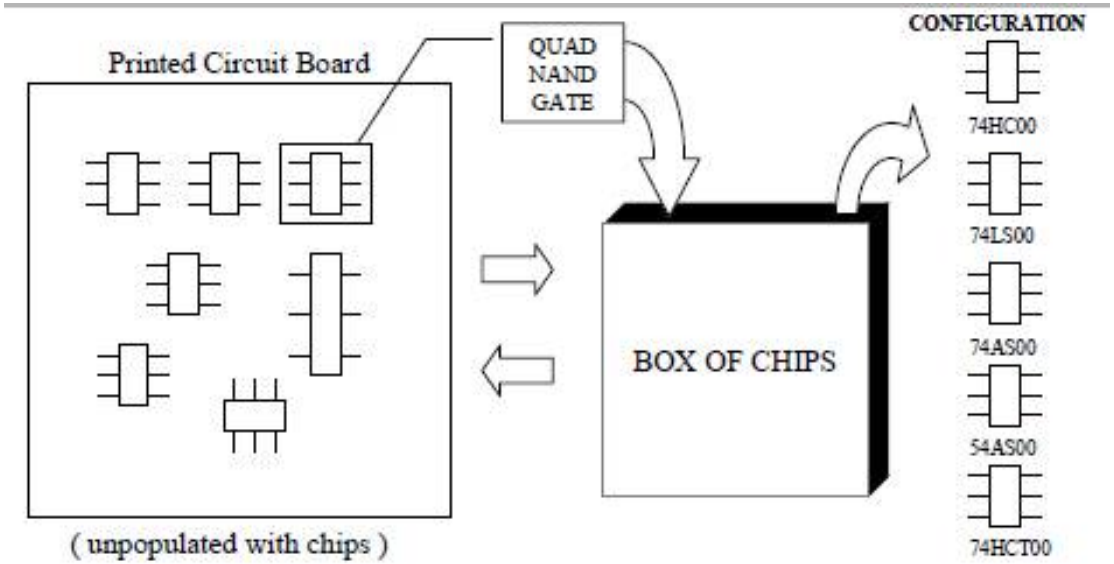
waveform_element ::=
*value*_expression [**after** *time*_expression]
| **null** [**after** *time*_expression]

So an example of such a signal assignment is:

```
d_out <= null after Toz;
```

Analogy


Printed Circuit Board
BOX OF CHIPS
(unpopulated with chips)


Printed Circuit Board
QUAD
NAND
GATE
CONFIGURATION
74HC00
74LS00
74AS00
54AS00
74HCT00
BOX OF CHIPS
(unpopulated with chips)

Configurations

- Configurations
  - specify which architectures to use for a particular component
  - specify which parameter values to use for a particular component
- Two basic forms
  - configuration specifications
  - configuration declarations

VHDL Binding

- Definition:
  - Associating an architectural description with a component in a structural model.
- Configurations bind all component declarations

Choosing a System