

---

# EEL 4783: Hardware/Software Co-design with FPGAs

## Lecture 9: Short Introduction to VHDL\*

Prof. Mingjie Lin



---

\* Beased on notes of Turfts lecture

# What does HDL stand for?

---

HDL is short for Hardware Description Language

(**VHDL** – VHSIC Hardware **D**escription **L**anguage)  
(Very High Speed Integrated Circuit)

# Why use an HDL?

---

## **Question:**

How do we know that we have not made a mistake when we manually draw a schematic and connect components to implement a function?

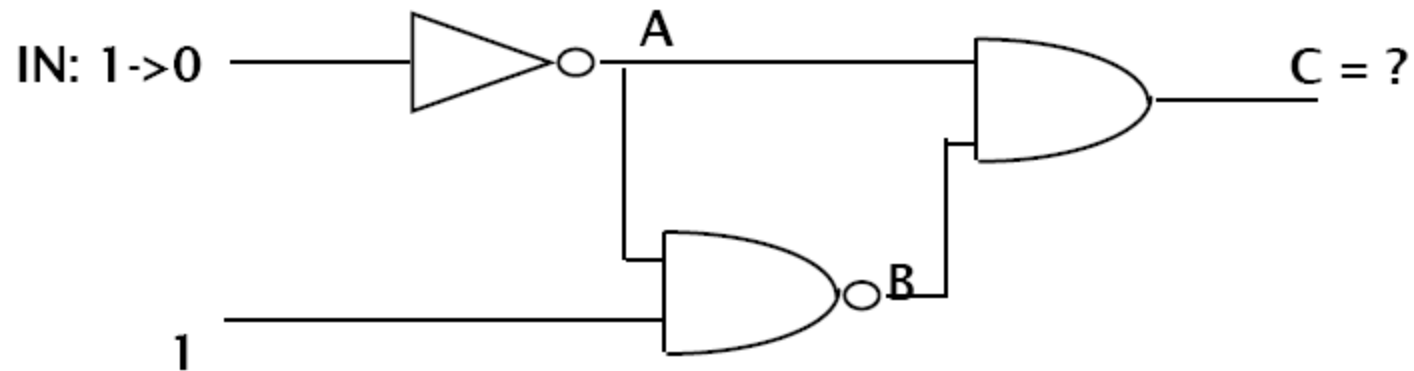
## **Answer:**

By describing the design in a high-level (=easy to understand) language, we can simulate our design before we manufacture it. This allows us to catch design errors, i.e., that the design does not work as we thought it would.

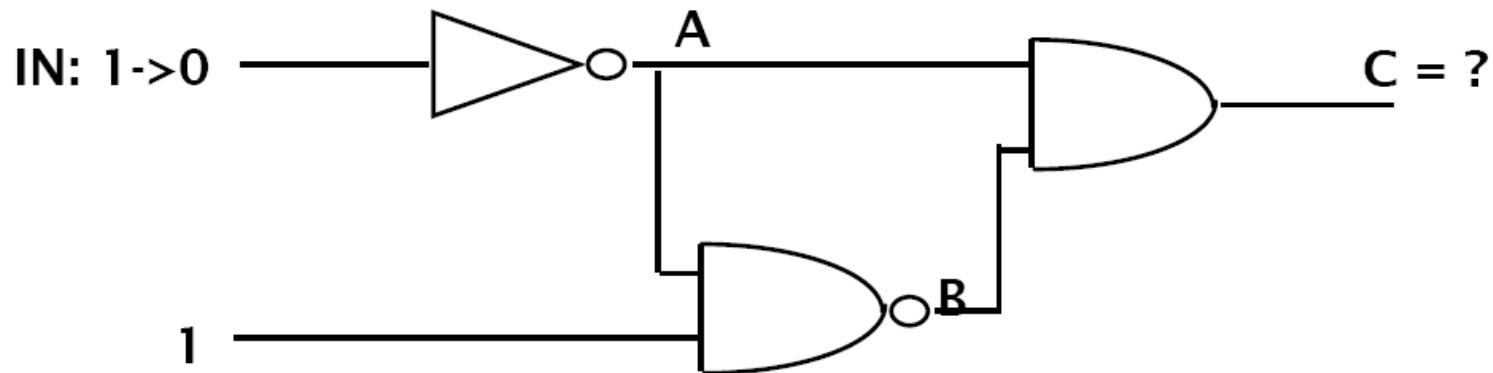
- Simulation guarantees that the design behaves as it should.

## How does the simulation work?

---



## What is the output of C?



**NAND gate evaluated first:**

IN: 1->0

A: 0->1

B: 1->0

C: 0->0

**AND gate evaluated first:**

IN: 1->0

A: 0->1

C: 0->1

B: 1->0

C: 1->0

# The two-phase simulation cycle

---

- 1) Go through all functions. Compute the next value to appear on the output using current input values and store it in a local data area (a value table inside the function).
- 2) Go through all functions. Transfer the new value from the local table inside to the data area holding the values of the outputs (=inputs to the next circuit)

# Cycle-based simulators

---

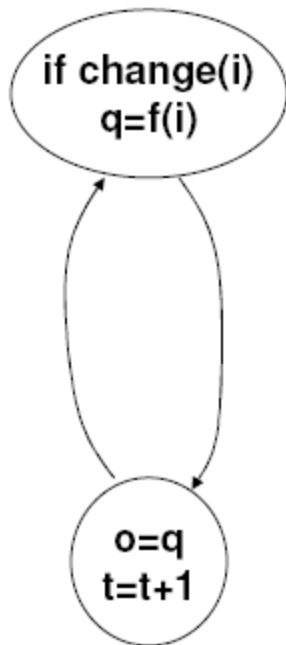


Go through all functions using current inputs and compute next output

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

# Event-based simulators with event queues

---



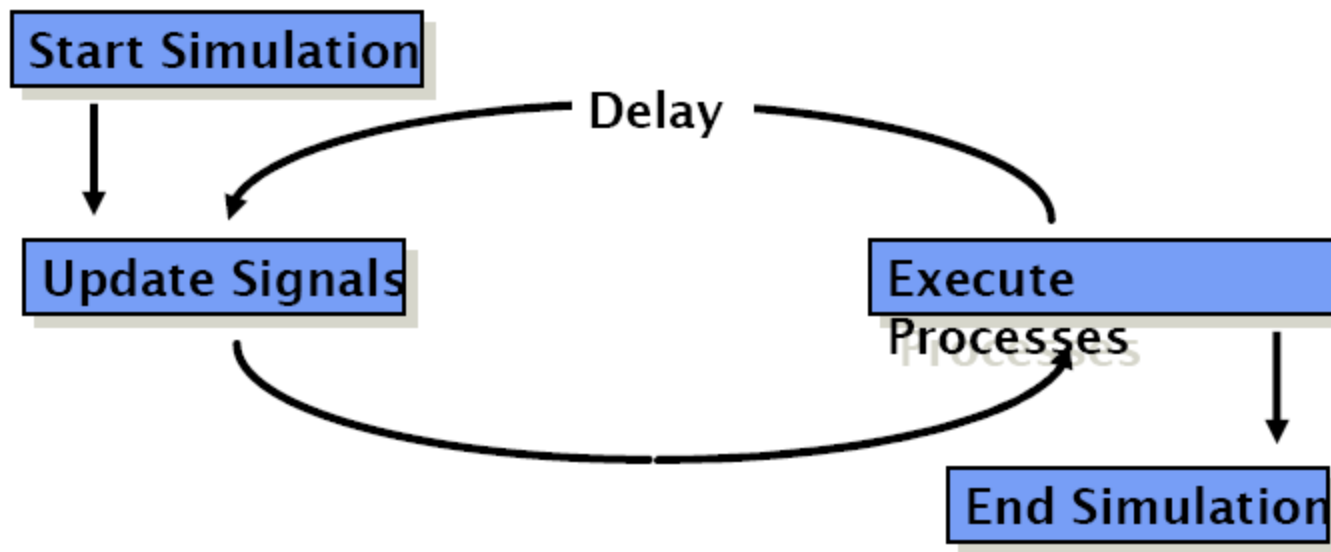
Go through all functions whose inputs has changed and compute value and time for next output change

Increase time to first scheduled event & update signals

# VHDL Simulation Cycle

---

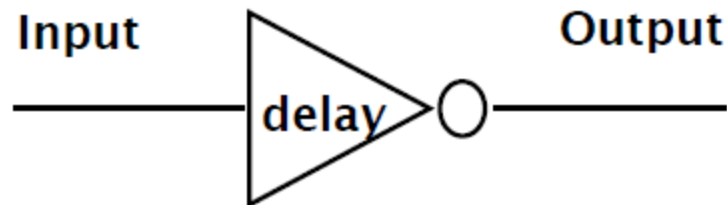
- VHDL uses a simulation cycle to model the stimulus and response nature of digital hardware.



# VHDL Delay Models

---

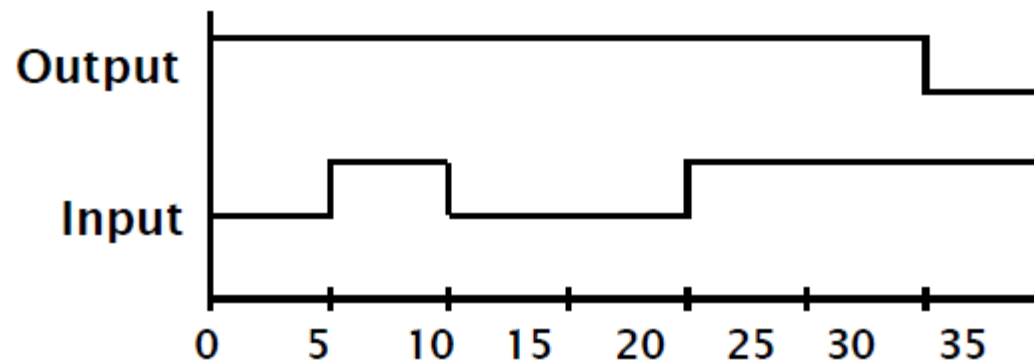
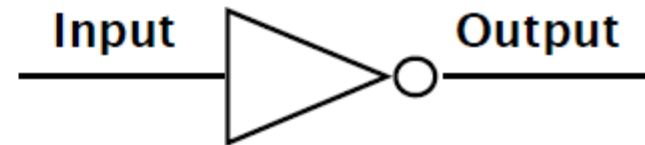
- Delay is created by scheduling a signal assignment for a future time.
- Delay in a VHDL cycle can be of several types
  - Inertial
  - Transport
  - Delta



# Inertial Delay

- Default delay type
- Allows for user specified delay
- Absorbs pulses of shorter duration than the specified delay

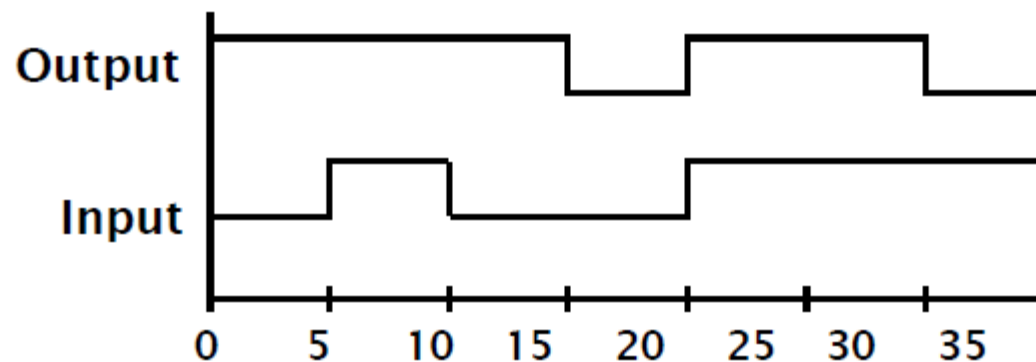
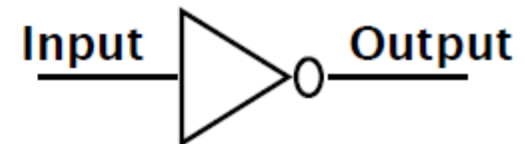
```
-- Inertial is the default  
Output <= NOT Input AFTER 10 ns;
```



# Transport Delay

- Must be explicitly specified by user
- Allows for user specified delay
- Passes all input transitions with delay

```
-- TRANSPORT must be specified  
Output <= TRANSPORT NOT Input AFTER 10 ns;
```

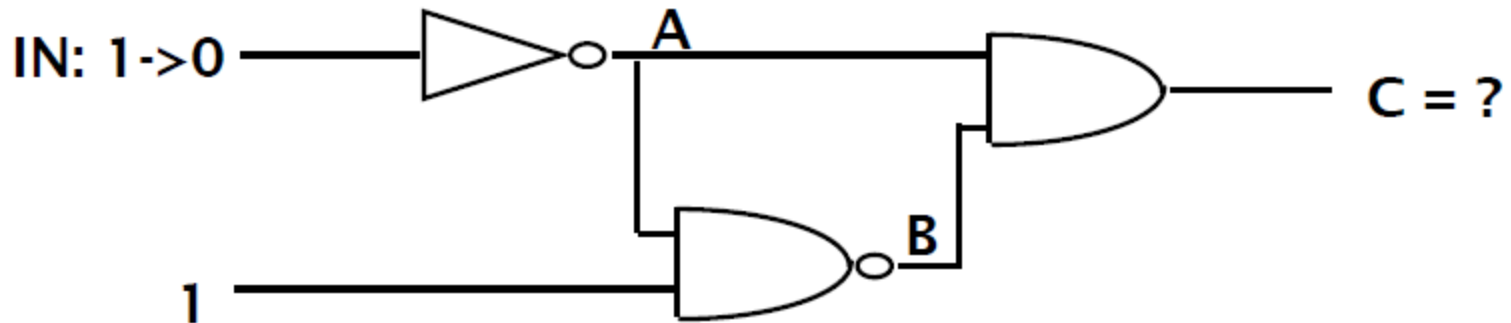


# Delta Delay

---

- 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

## Example – Delta Delay

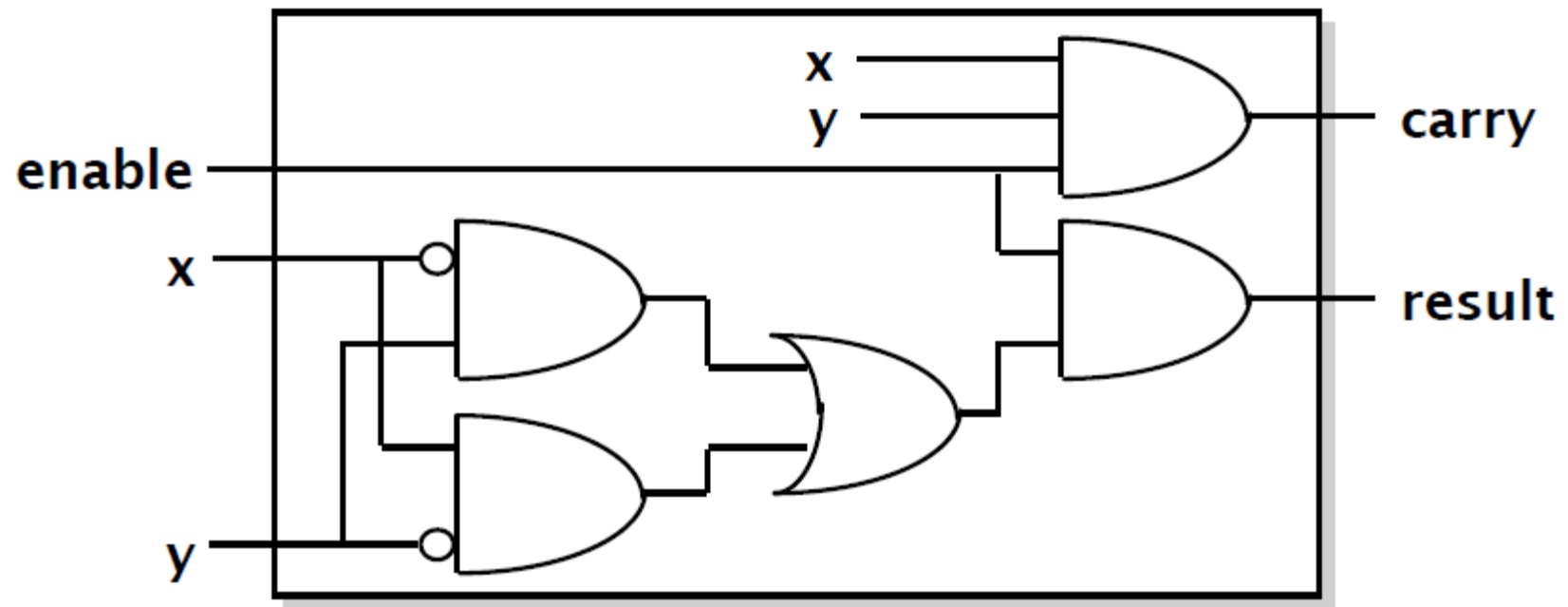


### Using delta delay scheduling

<u>Time</u>	<u>Delta</u>	<u>Event</u>
0 ns	1	IN: 1->0 eval inverter
<hr/>		
	2	A: 0->1 eval NAND, AND
<hr/>		
	3	B: 1->0 C: 0->1 eval AND
<hr/>		
	4	C: 1->0
<hr/>		
1 ns		

## How do we write code?

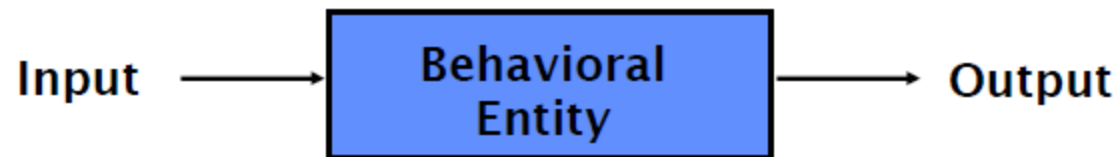
---



# Basic Form of VHDL Code

---

- Every VHDL design description consists of at least one entity / architecture pair, or one entity with multiple architectures.
- The entity section is used to declare I/O ports of the circuit. The architecture portion describes the circuit's behavior.
- A behavioral model is similar to a “black box”.
- Standardized design libraries are included before entity declaration.



# Standard Libraries

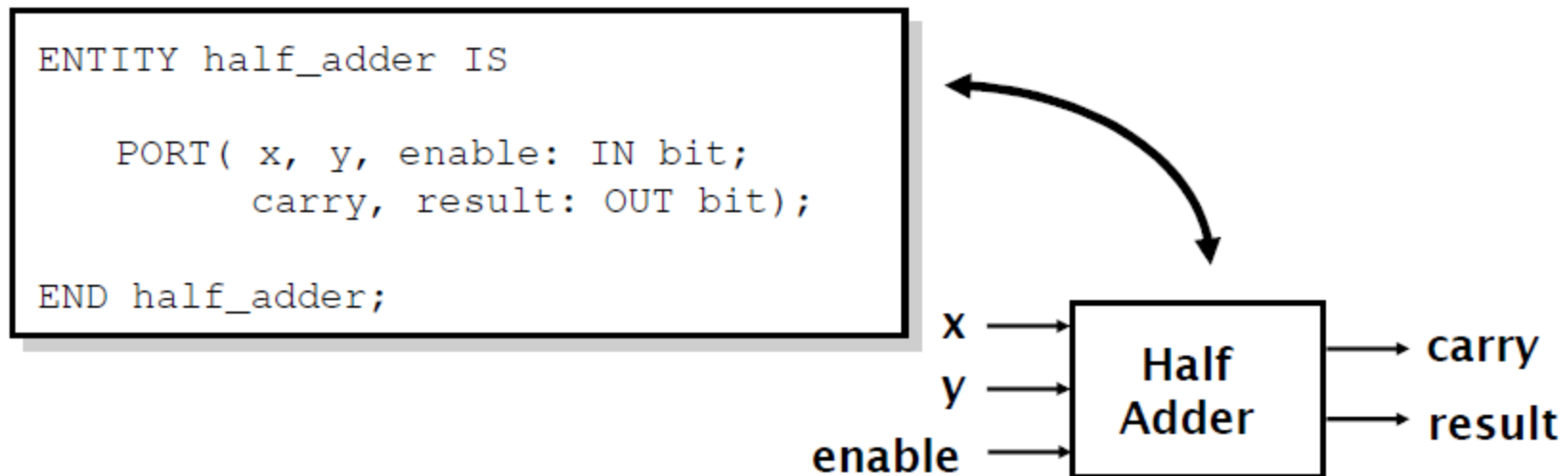
---

- Include *library ieee;* before entity declaration.
- `ieee.std_logic_1164` defines a standard for designers to use in describing interconnection data types used in VHDL modeling.
- `ieee.std_logic_arith` provides a set of arithmetic, conversion, comparison functions for signed, unsigned, `std_ulogic`, `std_logic`, `std_logic_vector`.
- `ieee.std_logic_unsigned` provides a set of unsigned arithmetic, conversion, and comparison functions for `std_logic_vector`.
- See all available packages at <http://www.cs.umbc.edu/portal/help/VHDL/stdpkg.html>

# Entity Declaration

---

- An entity declaration describes the interface of the component. Avoid using Altera's primitive names which can be found at [c:/altera/91/quartus/common/help/webhelp/master.htm#](http://c:/altera/91/quartus/common/help/webhelp/master.htm#)
- PORT clause indicates input and output ports.
- An entity can be thought of as a symbol for a component.



# Port Declaration

---

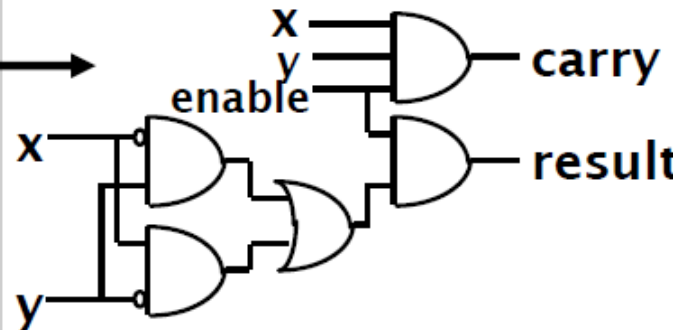
- PORT declaration establishes the interface of the object to the outside world.
- Three parts of the PORT declaration
  - Name
    - Any identifier that is not a reserved word.
  - Mode
    - In, Out, Inout, Buffer
  - Data type
    - Any declared or predefined datatype.
- Sample PORT declaration syntax:

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

# Architecture Declaration

- Architecture declarations describe the operation of the component.
- Many architectures may exist for one entity, but only one may be active at a time.
- An architecture is similar to a schematic of the component.

```
ARCHITECTURE behave OF half_adder IS
BEGIN
  PROCESS (enable, x, y)
  BEGIN
    IF (enable = '1') THEN
      result <= x XOR y;
      carry  <= x AND y;
    ELSE
      carry  <= '0';
      result <= '0';
    END IF;
  END PROCESS;
END behave;
```



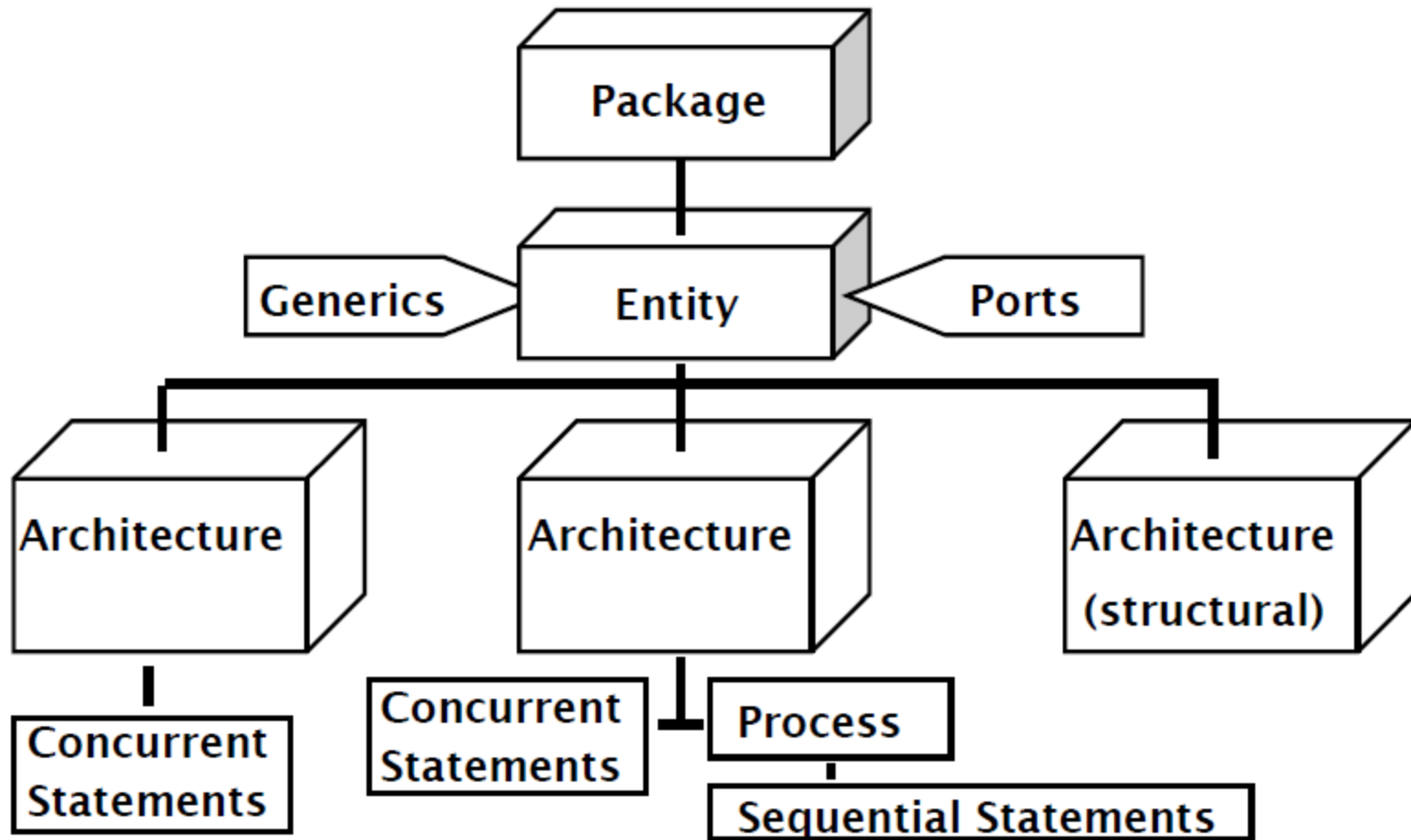
# Modeling Styles

---

- There are three modeling styles:
  - Behavioral (Sequential)
  - Data flow
  - Structural

# VHDL Hierarchy

---



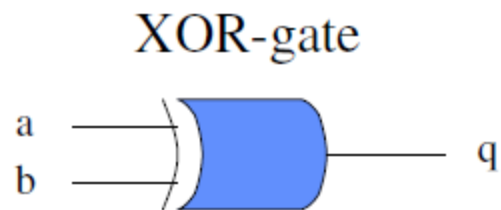
# Sequential vs Concurrent Statements

---

- VHDL provides two different types of execution: sequential and concurrent.
- Different types of execution are useful for modeling of real hardware.
  - Supports various levels of abstraction.
- Sequential statements view hardware from a “programmer” approach.
- Concurrent statements are order-independent and asynchronous.

# Sequential Style

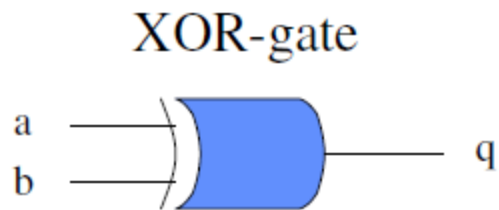
---



```
process(a,b)
begin
    if (a/=b) then
        q <= '1';
    else
        q <= '0';
    end if;
end process;
```

# Data flow Style

---



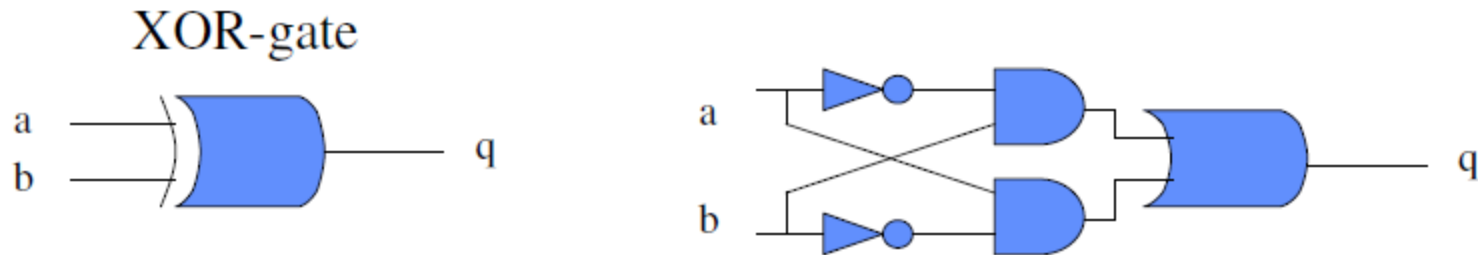
```
q <= a xor b;
```

Or in behavioral data flow style:

```
q <= '1' when a/=b else '0';
```

# Structural Style

---




```
u1: inverter port map (a, ai);  
u2: inverter port map (b, bi);  
u3: and_gate port map (ai, b, t3);  
u4: and_gate port map (bi, a, t4);  
u5: or_gate port map (t3, t4, q);
```

# Sequential Style Syntax

---

```
[ process_label : ] PROCESS  
[ ( sensitivity_list ) ]  
  
    process_declarations  
  
BEGIN  
  
    process_statements  
  
END PROCESS [ process_label ] ;
```

**NO  
SIGNAL  
DECLARATIONS!**



- Assignments are executed sequentially inside processes.

# Sequential Statements

---

- {Signal, Variable} assignments
- Flow control
  - if <condition> then <statments>  
[elseif <condition> then <statments>]  
else <statements>  
end if;
  - for <range> loop <statments> end loop;
  - while <condition> loop <statments> end loop;
  - case <condition> is
    - when <value> => <statements>;
    - when <value> => <statements>;
    - when others => <statements>;
- Wait on <signal> until <expression> for <time>;

# 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.

```
VARIABLE variable_name : type_name [:=value];  
  
VARIABLE opcode : BIT_VECTOR(3 DOWNT0 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;
```

# Signal Assignment

---

- A key difference between variables and signals is the assignment delay.

```
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

# Variable Assignment

```
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

# IF – vs CASE – statement Syntax

---

```
if (a='1') then
    q <= '1';
elsif (b='1') then
    q <= '1';
else
    q <= '0';
end if;
```

```
case (a&b) is
    when "00" =>
        q <= '0';
    when others =>
        q <= '1';
end case;
```

# FOR – vs WHILE – statement Syntax

---

```
for i in 0 to 9 loop
    q(i) <= a(i) and b(i);
end loop;
```

For is considered to be a combinational circuit by some synthesis tools. Thus, it cannot have a wait statement to be synthesized.

---

```
i:=0;
while (i<9) loop
    q <= a(i) and b(i);
    WAIT ON clk UNTIL clk='1';
end loop;
```

While is considered to be an FSM by some synthesis tools. Thus, it needs a wait statement to be synthesized.

# 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;`

# 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!

# Concurrent Process Equivalents

---

- All concurrent statements correspond to a process equivalent.

U0: `q <= a xor b after 5 ns;`

is short hand notation for

U0: `process`

`begin`

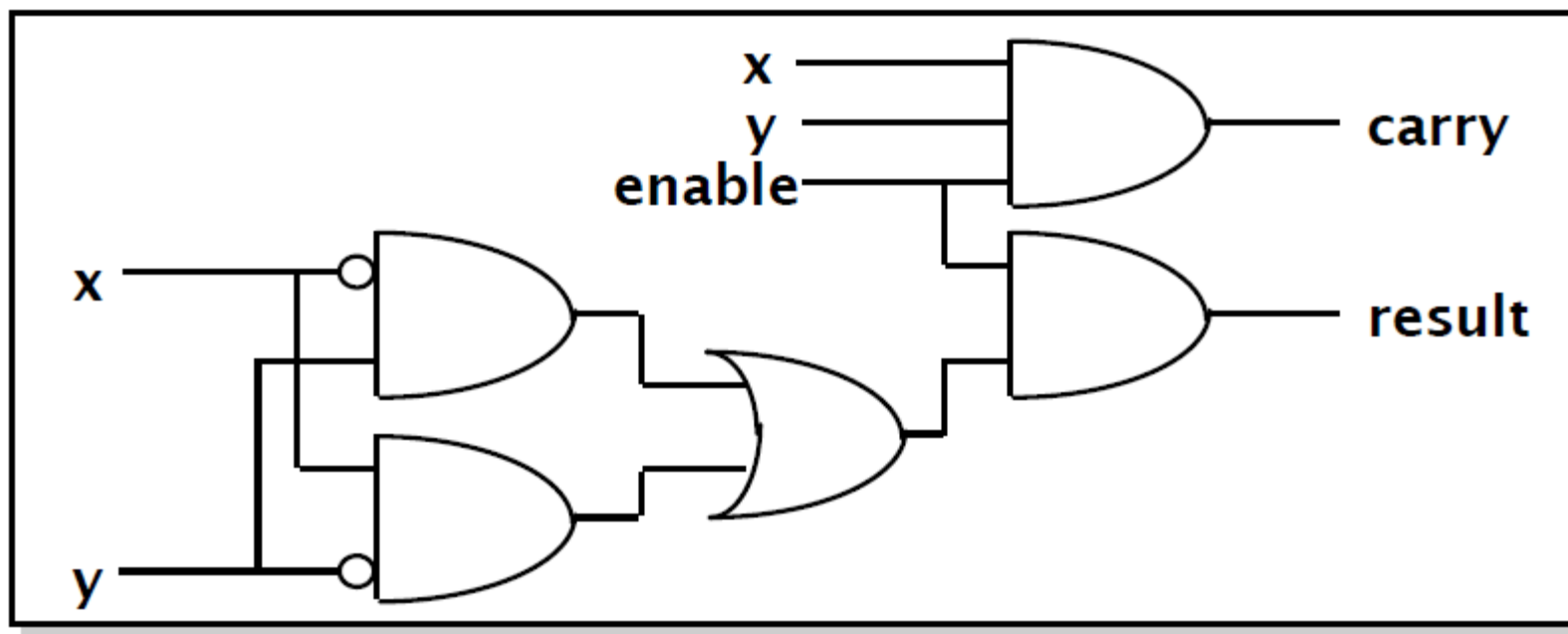
`q <= a xor b after 5 ns;`

`wait on a, b;`

`end process;`

# Structural Style

- Circuits can be described like a netlist.
- Components can be customized.
- Large, regular circuits can be created.



# Structural Statements

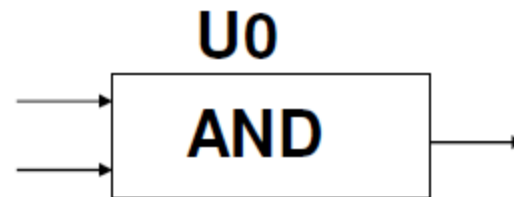
---

- Structural VHDL describes the arrangement and interconnection of components.
  - Behavioral descriptions, on the other hand, define responses to signals.
- Structural descriptions can show a more concrete relation between code and physical hardware.
- Structural descriptions show interconnects at any level of abstraction.

# Structural Statements

---

- The component instantiation is one of the building blocks of structural descriptions.
- The component instantiation process requires component declarations and component instantiation statements.
- Component instantiation declares the interface of the components used in the architecture.
- At instantiation, only the interface is visible.
  - The internals of the component are hidden.



# Component Declaration

---

- The component declaration declares the interface of the component to the architecture.
- Necessary if the component interface is not declared elsewhere (package, library).

```
ARCHITECTURE test OF test_entity
  COMPONENT and_gate
    PORT ( in1, in2 : IN BIT;
           out1 : OUT BIT);
  END COMPONENT;
... more statements ...
```

# Component Instantiation

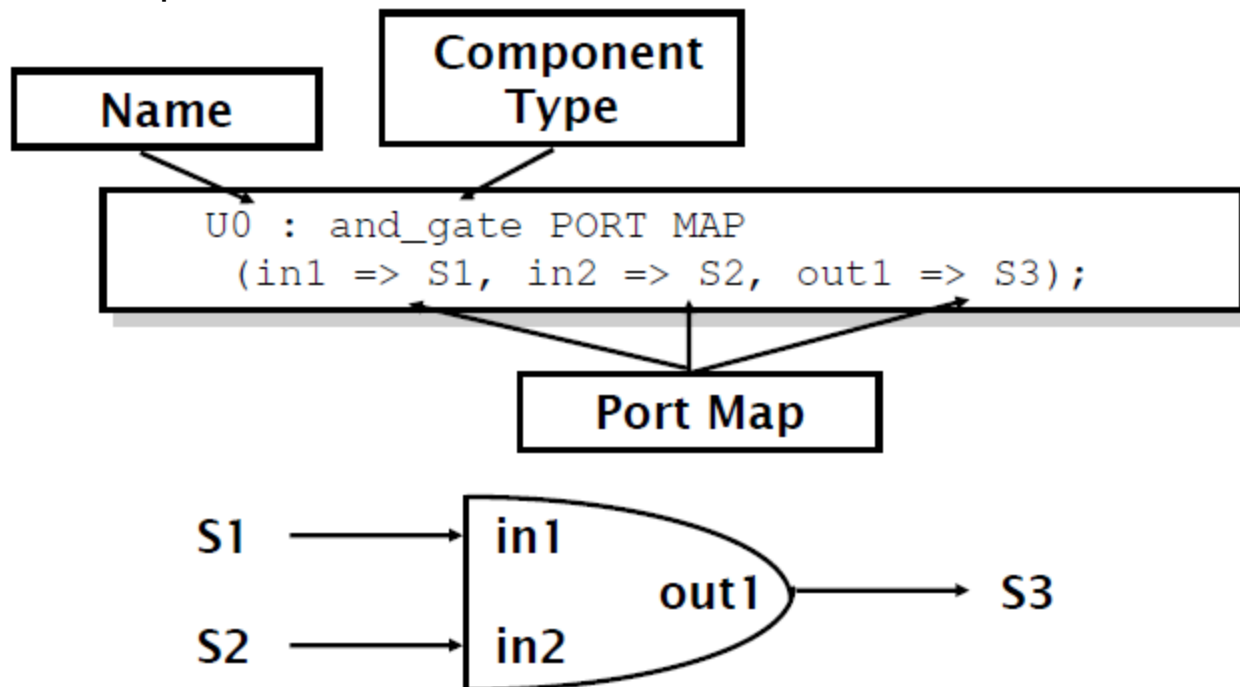
---

- The instantiation statement maps the interface of the component to other objects in the architecture.

```
ARCHITECTURE test OF test_entity
  COMPONENT and_gate
    PORT ( in1, in2 : IN BIT;
           out1 : OUT BIT);
  END COMPONENT;
  SIGNAL S1, S2, S3 : BIT;
BEGIN
  U0 : and_gate PORT MAP (in1 => S1,
                          in2 => S2, out1 => S3);
END test;
```

# Component Instantiation Syntax

- The instantiation has 3 key parts
  - Name
  - Component type
  - Port map



# Component Libraries

---

- Component declarations may be made inside packages.
  - Components do not have to be declared in the architecture body

```
PACKAGE my_stuff IS
    COMPONENT and_gate
        PORT ( in1, in2 : IN BIT;
              out1 : OUT BIT);
    END COMPONENT;
END my_stuff;

USE Work.my_stuff.ALL;

ARCHITECTURE test OF test_entity
    SIGNAL S1, S2, S3 : BIT;
BEGIN
    Gate1 : and_gate
        PORT MAP (S1, S2, S3);
END test;
```

entity ADDER is generic(n: natural :=2); port( A: in std\_logic\_vector(n-1 downto 0); B: in std\_logic\_vector(n-1 downto 0); carry: out std\_logic; sum: out std\_logic\_vector(n-1

# Generics

---

- Generics allow the component to be customized upon instantiation.
- Generics pass information from the entity to the architecture.
- Common uses of generics
  - Customize timing
  - Alter range of subtypes
  - Change size of arrays

```
ENTITY adder IS
  GENERIC(n: natural :=2);
  PORT(
    A: IN STD_LOGIC_VECTOR(n-1 DOWNT0 0);
    B: IN STD_LOGIC_VECTOR(n-1 DOWNT0 0);
    C: OUT STD_LOGIC;
    SUM: OUT STD_LOGIC_VECTOR(n-1 DOWNT0
0) );
END adder;
```

# Technology Modeling

---

- One use of generics is to alter the timing of a certain component.
- It is possible to indicate a generic timing delay and then specify the exact delay at instantiation.

```
COMPONENT inv IS  
  PORT ( in1 : IN BIT;  
         out1 : OUT BIT);  
  GENERIC (tplh, tphl : TIME);  
END COMPONENT;
```

- The example above declares the interface to a component named *inv*.
- The propagation time for high-to-low and low-to-high transitions can be specified later.

# Structural Statements

---

- The GENERIC MAP is similar to the PORT MAP in that it maps specific values to generics declared in the component.

```
PACKAGE my_stuff IS
  COMPONENT and_gate
    GENERIC ( tplh, tphl : time);
    PORT ( in1, in2 : IN BIT; out1 : OUT BIT);
  END COMPONENT;
END my_stuff;

USE Work.my_stuff.ALL;
ARCHITECTURE test OF test_entity
  SIGNAL S1, S2, S3 : BIT;
BEGIN
  Gate1 : my_stuff.and_gate
    GENERIC MAP (2 ns, 3 ns)
    PORT MAP (S1, S2, S3);
END test;
```

# Generate Statement

---

- Structural for-loops: The GENERATE statement
  - Some structures in digital hardware are repetitive in nature. (RAM, ROM, registers, adders, multipliers, ...)
  - VHDL provides the GENERATE statement to automatically create regular hardware.
  - Any VHDL concurrent statement may be included in a GENERATE statement, including another GENERATE statement.

# Generate Statement Syntax

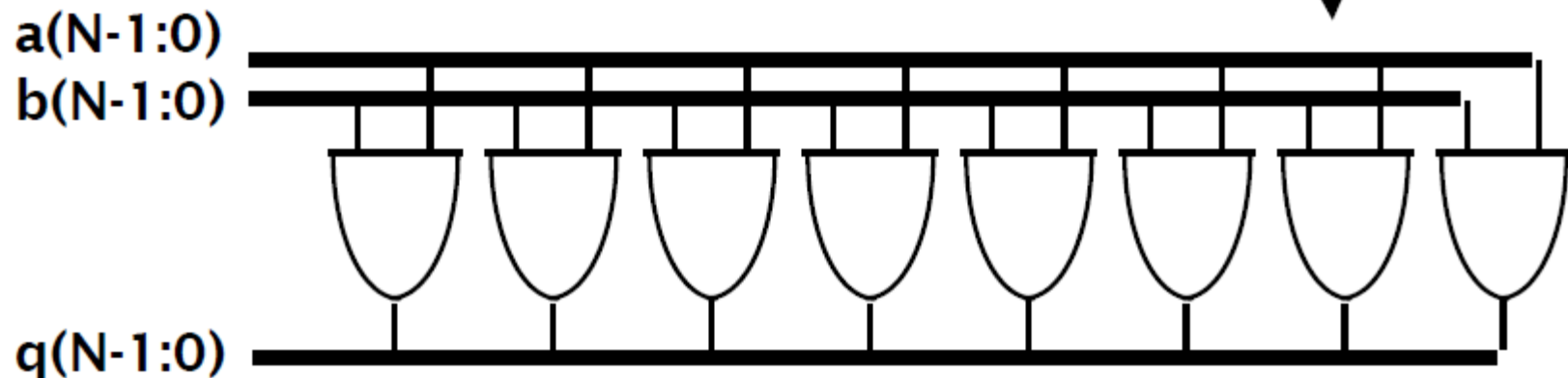
---

- All objects created are similar.
- The GENERATE parameter must be discrete and is undefined outside the GENERATE statement.

```
name : FOR N IN 1 TO 8 GENERATE  
      concurrent-statements  
END GENERATE name;
```

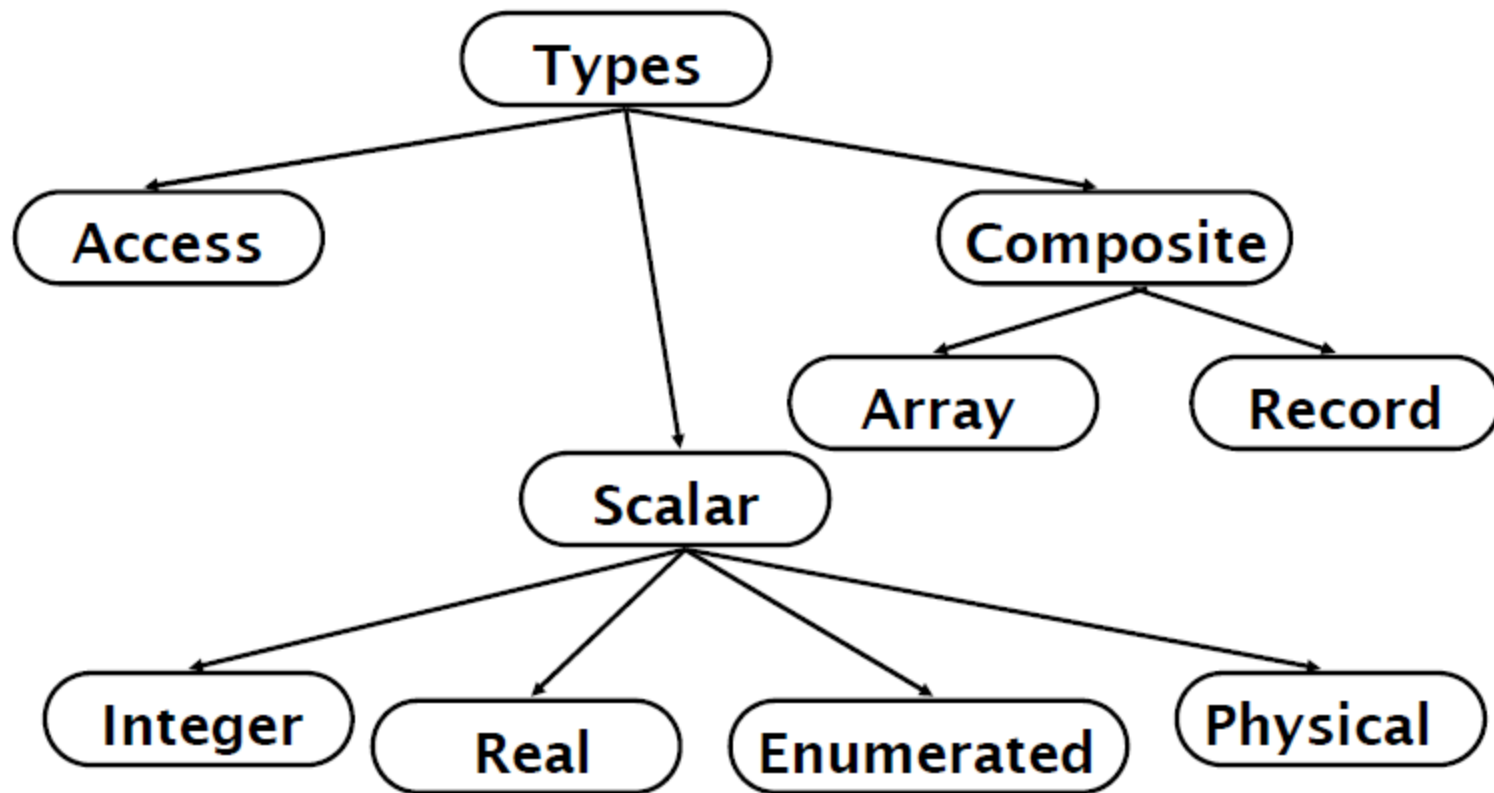
# Example: Array of AND-gates

```
USE work.my_gates.all;  
ARCHITECTURE structural OF and_bit_vector IS  
BEGIN  
  G1 : FOR i IN N-1 DOWNTO 0 GENERATE  
    and_array : and_gate  
      GENERIC MAP (2 ns, 3 ns)  
      PORT MAP (i1=>a(i), i2=>b(i), q=>q(i));  
    END GENERATE G1;  
END structural;
```



# VHDL Data Types

---



# Predefined Data Types

---

- bit ('0' or '1')
- bit\_vector (array of bits)
- integer
- real
- time (physical data type)

# Integer

---

- 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

---

- 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;
```

# Enumerated

---

- Enumerated
  - User defined range
  - Enumerated example

```
TYPE binary IS ( ON, OFF );  
... some statements ...  
ARCHITECTURE test_enum OF test IS  
BEGIN  
    PROCESS (X)  
        VARIABLE a: binary;  
    BEGIN  
        a := ON;  -- OK  
        ... more statements ...  
        a := off; -- OK  
        ... more statements ...  
    END PROCESS;  
END TEST;
```

# Physical

---

- 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.

# Array

- 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;
```

**0...element numbers... 31**

<b>0</b>	<b>...array values...</b>	<b>1</b>
----------	---------------------------	----------

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

```
TYPE register IS ARRAY (15 DOWNT0 0) OF BIT;
```

# Record

---

- Record
  - Used to collect one or more elements of different types in a single construct.
  - Elements can be any VHDL data type.
  - Elements are accessed through field name.

```
TYPE binary IS ( ON, OFF );
TYPE switch_info IS
    RECORD
        status : binary;
        IDnumber : integer;
    END RECORD;

VARIABLE switch : switch_info;

switch.status := on;  -- status of the switch
switch.IDnumber := 30;  -- number of the switch
```

# Subtype

---

- Subtype
  - Allows for user defined constraints on a data type.
  - May include entire range of base type.
  - Assignments that are out of the subtype range result in error.
  - Subtype example

```
SUBTYPE name IS base_type RANGE <user range>;  
SUBTYPE first_ten IS INTEGER RANGE 0 to 9;
```

# Natural and Positive Integers

---

- Integer subtypes:
  - Subtype Natural is integer range 0 to integer'high;
  - Subtype Positive is integer range 1 to integer'high;

# Boolean, Bit and Bit\_vector

---

- type Boolean is (false, true);
- type Bit is ('0', '1');
- type Bit\_vector is array (integer range <>) of bit;

# Char and String

---

- type Char is (NUL, SOH, ..., DEL);
  - 128 chars in VHDL'87
  - 256 chars in VHDL'93
- type String is array (positive range <>) of Char;

# IEEE Predefined data types

---

- type Std\_ulogic is ('U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '-');
  - 'U' -- Uninitialized
  - 'X' -- Forcing unknown
  - '0' -- Forcing zero
  - '1' -- Forcing one
  - 'Z' -- High impedance
  - 'W' -- Weak Unknown
  - 'L' -- Weak Low
  - 'H' -- Weak High
  - '-' -- Don't care
- type std\_logic is resolved std\_ulogic;
- type std\_logic\_vector is array (integer range <>) of std\_logic;

# Assignments

---

- `constant a: integer := 523;`
- `signal b: bit_vector(11 downto 0);`

`b <= "000000010010";`

`b <= B"000000010010";`

`b <= B"0000_0001_0010";`

`b <= X"012";`

`b <= O"0022";`

# Vector & Array assignments

---

- subtype instruction: bit\_vector(31 downto 0);
- signal regs: array(0 to 15) of instruction;

regs(2) <= regs(0) + regs(1);

regs(1)(7 downto 0) <= regs(0)(11 downto 4);

# Alias Statement

---

- Signal instruction: `bit_vector(31 downto 0);`
- Alias op1: `bit_vector(3 downto 0) is instruction(23 downto 20);`
- Alias op2: `bit_vector(3 downto 0) is instruction(19 downto 16);`
- Alias op3: `bit_vector(3 downto 0) is instruction(15 downto 12);`
  - `Op1 <= "0000";`
  - `Op2 <= "0001";`
  - `Op3 <= "0010";`
  - `Regs(bit2int(op3)) <= regs(bit2int(op1)) + regs(bit2int(op2));`

# Type Conversion (Similar Base)

---

- Similar but not the same base type:
  - signal i: integer;
  - signal r: real;
  - i <= integer(r);
  - r <= real(i);

# Type Conversion (Same Base)

---

- Same base type:

```
type a_type is array(0 to 4) of bit;  
signal a:a_type;  
signal s:bit_vector(0 to 4);
```

`a<="00101"` -- Error, is RHS a `bit_vector` or an `a_type`?

`a<=a_type'("00101");` -- type qualifier

`a<=a_type(s);` -- type conversion

# Type Conversion (Different Base)

---

- Different base types:

Function `int2bits(value:integer;ret_size:integer)` return `bit_vector`;

Function `bits2int(value:bit_vector)` return integer:

```
signal i:integer;
```

```
signal b:bit_vector(3 downto 0)
```

```
i<=bits2int(b);
```

```
b<=int2bits(i,4);
```

# Built-In Operators

---

- Logic operators
  - AND, OR, NAND, NOR, XOR, XNOR (XNOR in VHDL'93 only!!)
- Relational operators
  - =, /=, <, <=, >, >=
- Addition operators
  - +, -, &
- Multiplication operators
  - \*, /, mod, rem
- Miscellaneous operators
  - \*\*, abs, not

# Simulate Design using Quartus II

---

- Altera's Quartus II is a PLD design software suitable for high-density FPGA designs.
- Schematic Editor, VHDL/Verilog Editor, Waveform Simulator.

# Final issues

---

- Come by my office hours (right after class)
- Any questions or concerns?