ECSE-323 Digital System Design

VHDL Lecture #2

Simple Concurrent Assignment Statements

```
signal  <=  expression;
```

Examples:

```
A_Out <= not ( A_In or B_in ) and Enable;

Data <= X nor D2 xor (flag_A and flag_B);
```

•The "not" operator has the highest precedence.
•Operators in parentheses are evaluated first.
•All binary operators have equal precedence.
•Operators with the same precedence are evaluated left-to-right.

Other Types of Assignment Statements

The descriptions of complex circuits can be **tedious** to write using just simple assignment statements

So, VHDL gives additional types of signal assignment statements.

These other types of statements can also better direct synthesis programs in generating hardware

Forms of signal assignment statements:

- ***Simple Concurrent*** Assignment
- ***Selected*** Assignment
- ***Conditional*** Assignment
- ***Component*** Instantiation
- ***Generate*** Statements
- ***Process*** Blocks

Selected Signal Assignment Statements

A selected signal assignment is a means of conveniently describing ***multiplexer*** structures

```
with Dsel select
      Y <=
            A when "00",
            B when "01",
            C when "10",
            D when others;
```

$\left.\begin{matrix} \\ \\ \\ \end{matrix}\right\}$ **Order is not Important!**

Note: ***All*** conditions ***must*** be defined in a selected signal assignment, otherwise your code will not conform to the VHDL standard and you will probably get an error from whatever software happens to be reading your code.

Coverage of all conditions can be tricky to ensure – which is why the "**`when others`**" clause is useful

An example of possible problems -

```
with Dsel select
   Y <=
         A when “00”,
         B when “01”,
         C when “10”,
         D when “11”;
```

This assignment statement will give an ***error*** if **Dsel** is of type **STD_LOGIC_VECTOR** (but not if **Dsel** is of type **BIT_VECTOR**).

Why?

Because the *type* of *signal* **Dsel** has more possible values than just 0 and 1 (e.g. a high-impedance value – Z) so that not all possibilities are covered.

Conditional Assignment Statements

Conditional signal assignment is similar to selected signal assignment. It is often used for circuits which implement some sort of ***priority***.

```
    Y <=
          A when DSel = "00" else
          B when DSel = "01" else     Order is
          C when DSel = "10" else     Important!
          D;
```

Conditional Assignment Statements

Unlike in selected assignment it is *not absolutely required that all conditions be covered.*

```
Y <=
      A when DSel = "00" else
      B when DSel = "01" else
      C when DSel = "10";
```

If all conditions are *not* accounted for in a conditional assignment statement, a ***storage element*** (e.g. a *flipflop*) will be created, a phenomenon known as

implied memory.

Implied Memory

Consider the following conditional statement:

```
Y <= A when D = '1';
```

This statement does not say what happens when D is 0. Therefore ***no events can be created on Y*** when D is 0, no matter what events occur on other signals. Thus the value of Y is held (memory) when D is 0.

Implied Memory

To remove the implied memory we can write:

```
   Y <= A  when D = '1' else
         '0';
```

This describes a purely combinational circuit, with no storage elements.

Conditional vs. Selected Assignment

What is the difference between a conditional assignment and a selected assignment?

From a functional point of view, not much – they both can implement the same function.

From a synthesis point of view, quite a bit! – they both generate different forms of hardware.

Selected assignment ⇒ two-level, fast, high gate count and longer VHDL descriptions.

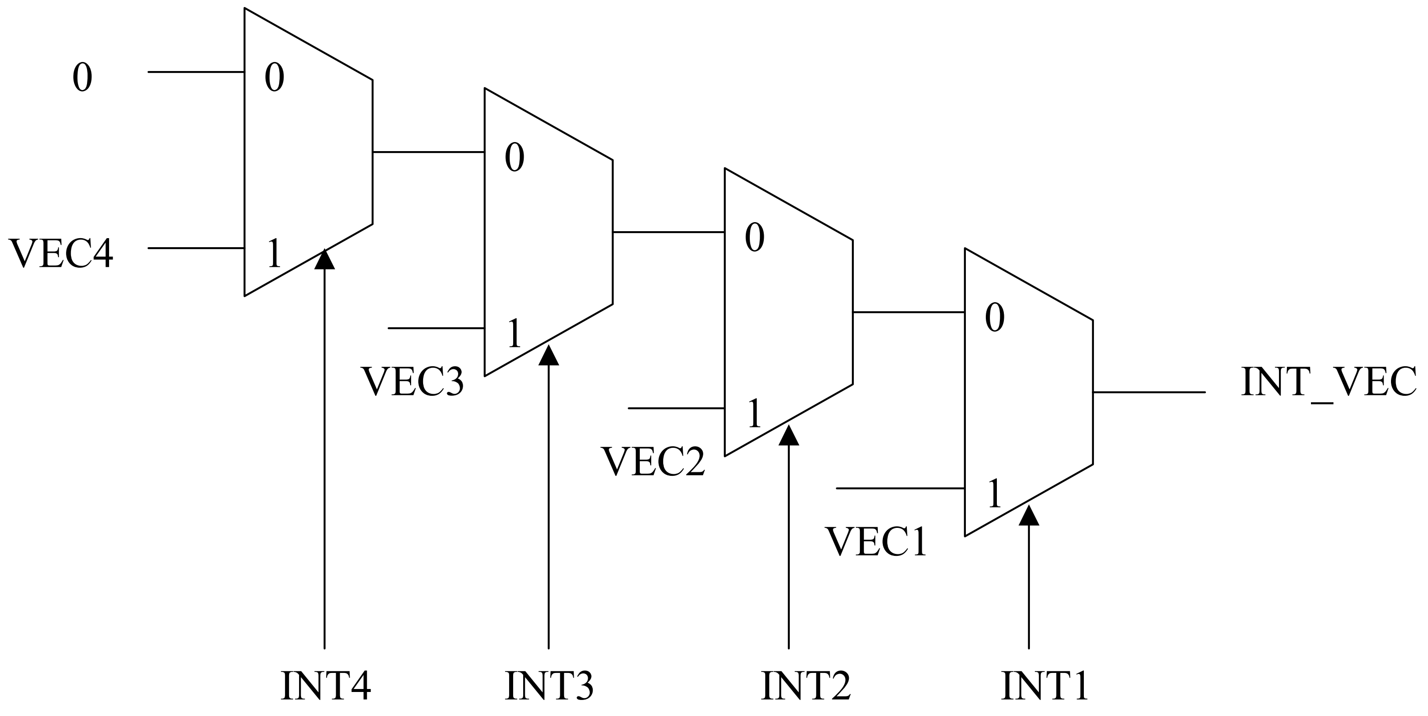
Conditional assignment ⇒ multi-level, slow (worst-case), low gate count and shorter VHDL descriptions.

Use conditional assignments if you want to implement some sort of ***priority*** scheme.

For example, priority encoders are often used to handle multiple ***interrupt*** lines.

```
INT_VEC <=
   VEC1 when Int1 = '1' else
   VEC2 when Int2 = '1' else
   VEC3 when Int3 = '1' else
   VEC4 when Int4 = '1' else
   '0';
```

Priority Interrupt Encoder - as described with conditional signal assignment.



Maximum delay is 4 MUX gate delays

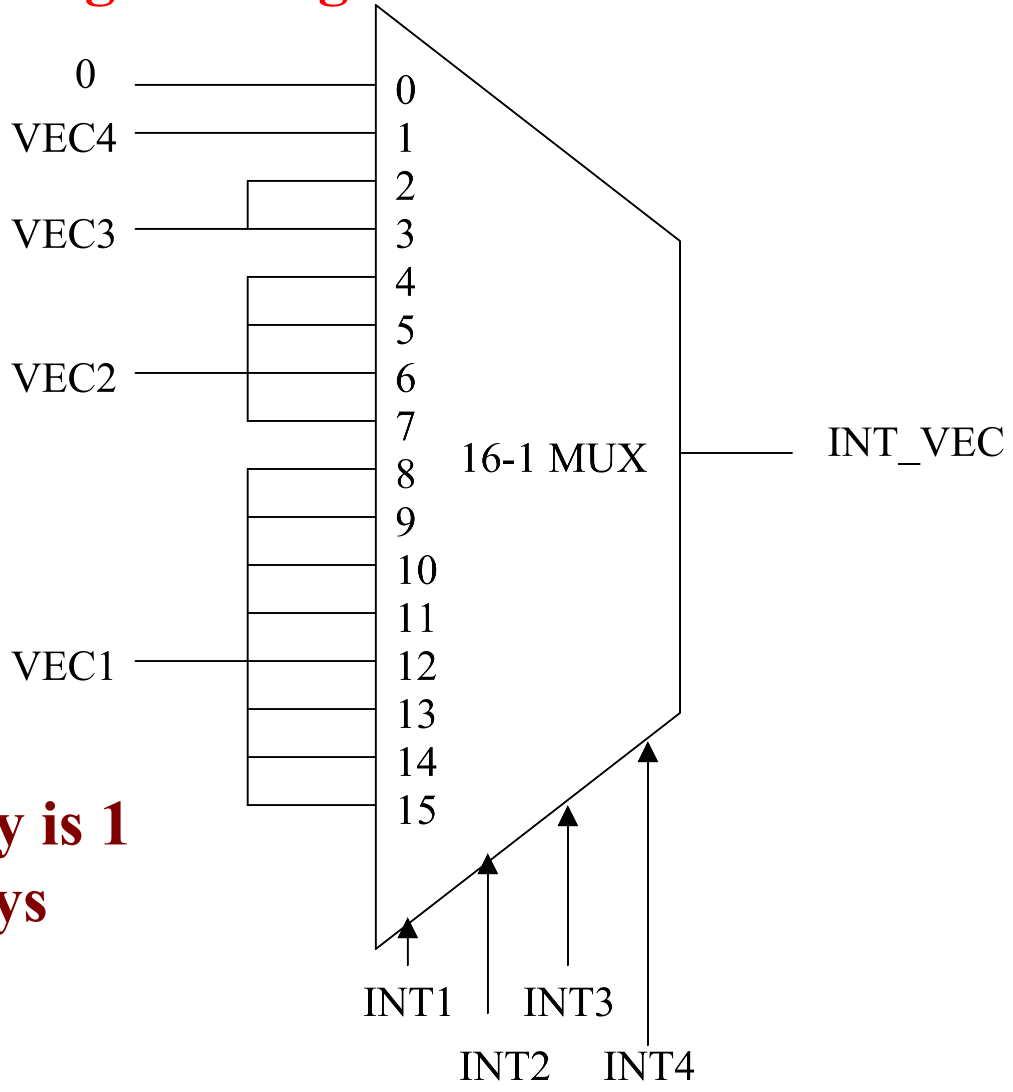
The equivalent circuit description using **Selected Signal assignment** is longer:

```
with IntSel select
   IntVec <=
      Vec1 when “1000”, Vec1 when “1001”,
      Vec1 when “1010”, Vec1 when “1011”,
      Vec1 when “1100”, Vec1 when “1101”,
      Vec1 when “1110”, Vec1 when “1111”,
      Vec2 when “0100”, Vec2 when “0101”,
      Vec2 when “0110”, Vec2 when “0111”,
      Vec3 when “0010”, Vec3 when “0011”,
      Vec4 when “0001”, ‘0’  when others;
```

To make the selected assignment statement as short as possible, always group the output value that has the most cases (for this example it is **Vec1**) under the "**when others**" clause:

```
with IntSel select
   IntVec <=
      Vec2 when “0100”, Vec2 when “0101”,
      Vec2 when “0110”, Vec2 when “0111”,
      Vec3 when “0010”, Vec3 when “0011”,
      Vec4 when “0001”, '0'  when “0000”,
      Vec1 when others;
```

Priority Interrupt Encoder - as described with selected signal assignment.



Maximum delay is 1 MUX gate delays

Usually, selected signal assignment results in ***faster, but larger***, circuits than using conditional signal assignment statements.

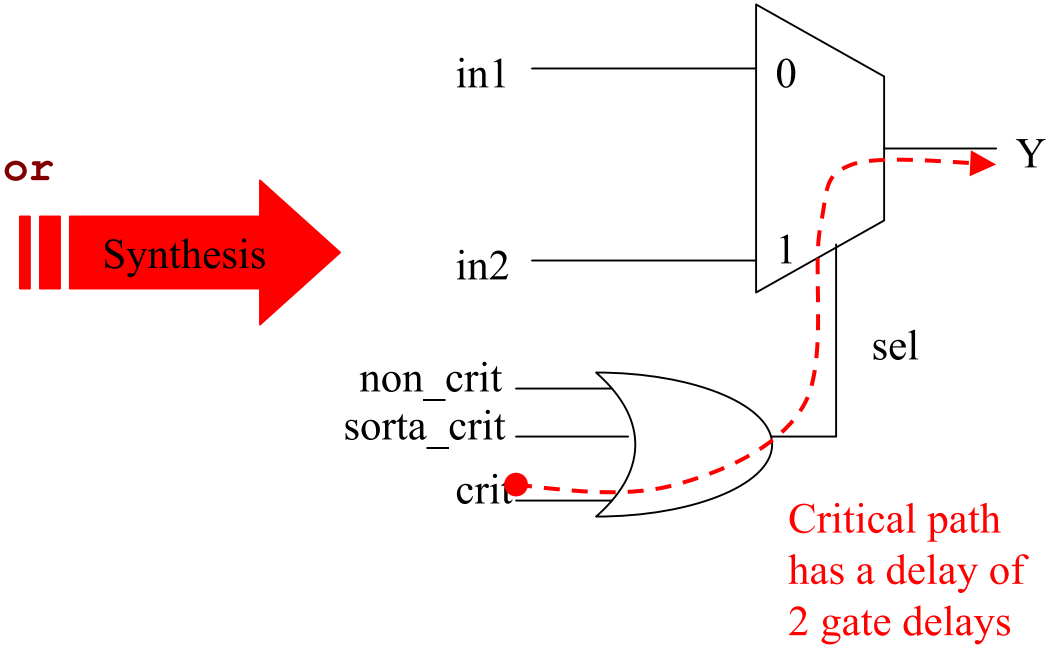
This is true only for the *worst-case* speed. The priority tree structures generated by conditional signal assignment have differing delays for different signal paths.

Some paths can have very short delays!

Conditional assignment statements are often used to ***reduce delay*** for ***critical path*** signals.

Consider the following VHDL description that uses selected signal assignment:

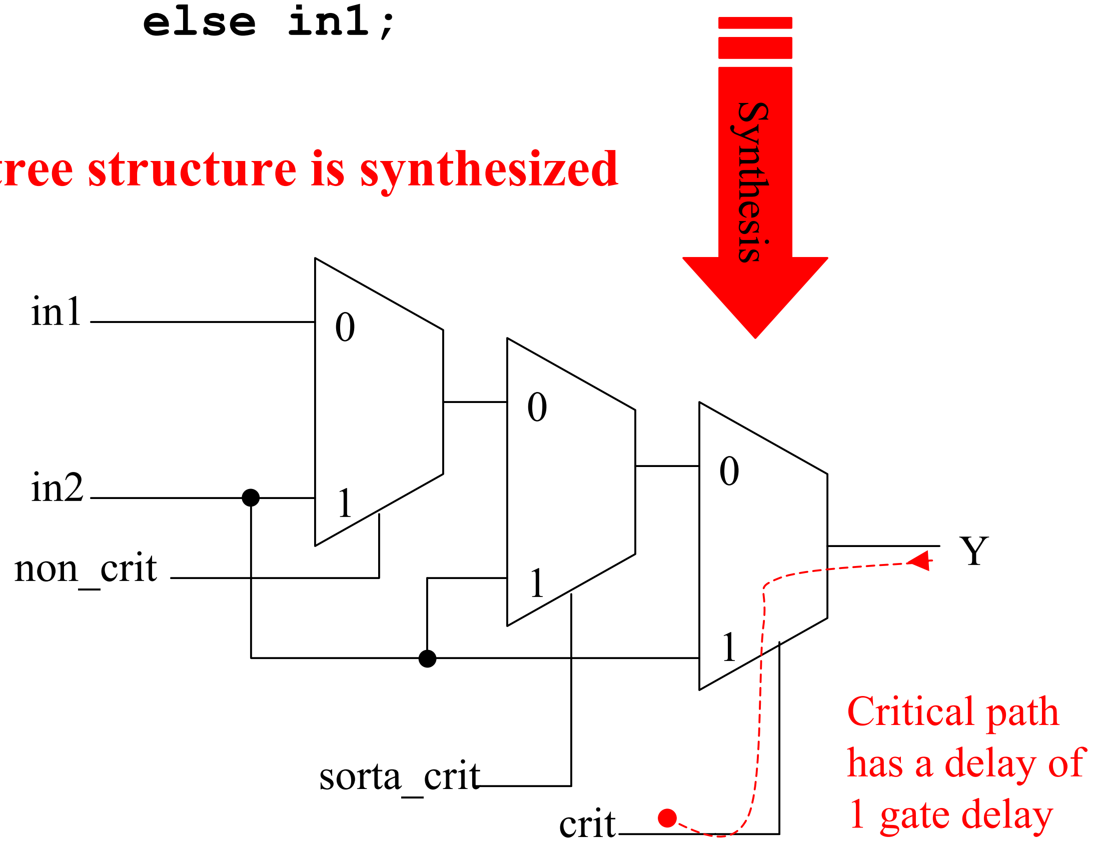
```
sel <= crit or sorta_crit or
non_crit;

with sel select

   Y <=
         in1 when '0',
         in2 when '1',
         in2 when others;
```



Re-design with *conditional assignment statements:*

```
Y <= in2 when crit = '1'
        else in2 when sorta_crit = '1'
        else in2 when non_crit = '1'
        else in1;
```

A priority tree structure is synthesized



Design Rule:

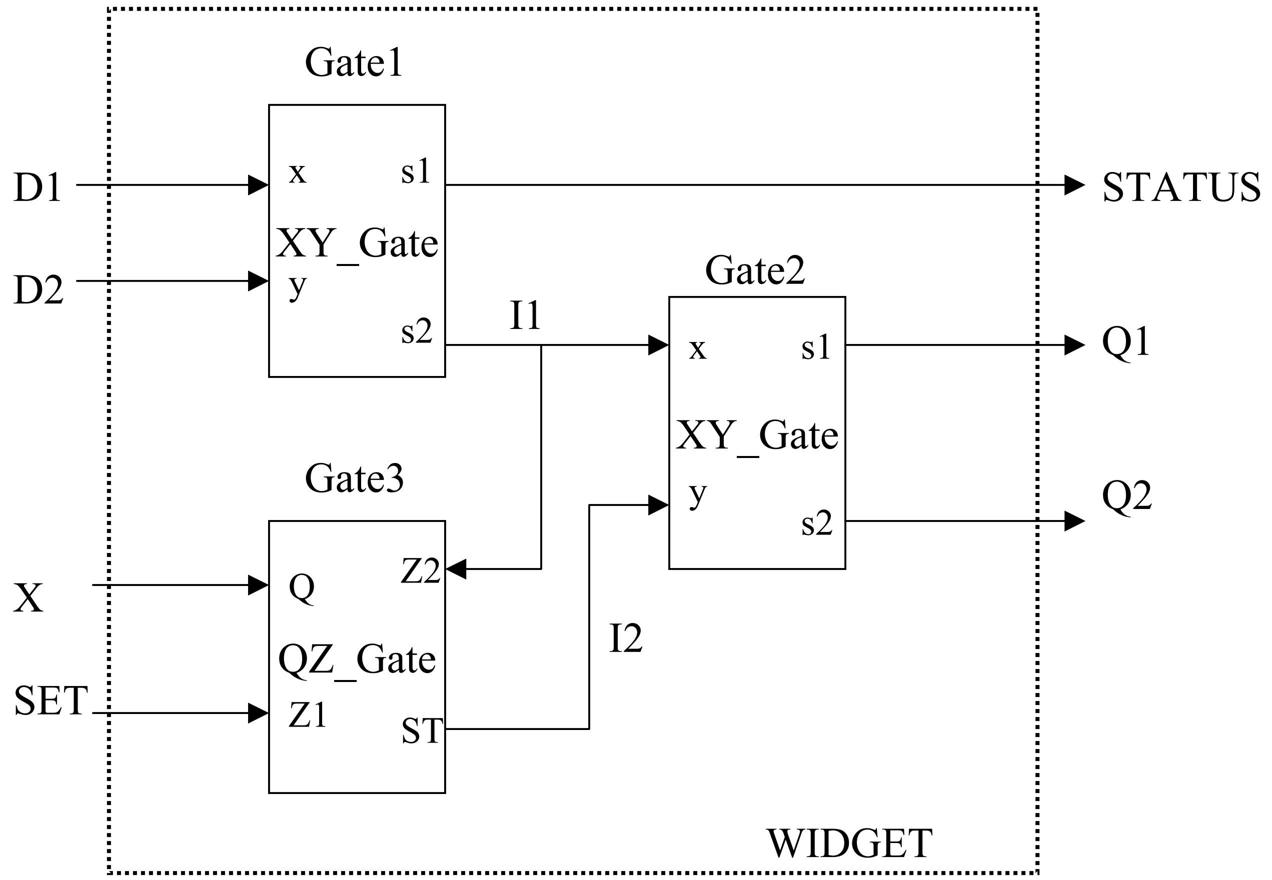
Put time-critical signals at the ***top levels*** of conditional assignment trees.

After circuit synthesis these signals will usually have the shortest path to the output.

Components

Components are used to connect multiple VHDL design units (entity/architecture pairs) together to form a larger, hierarchical design.

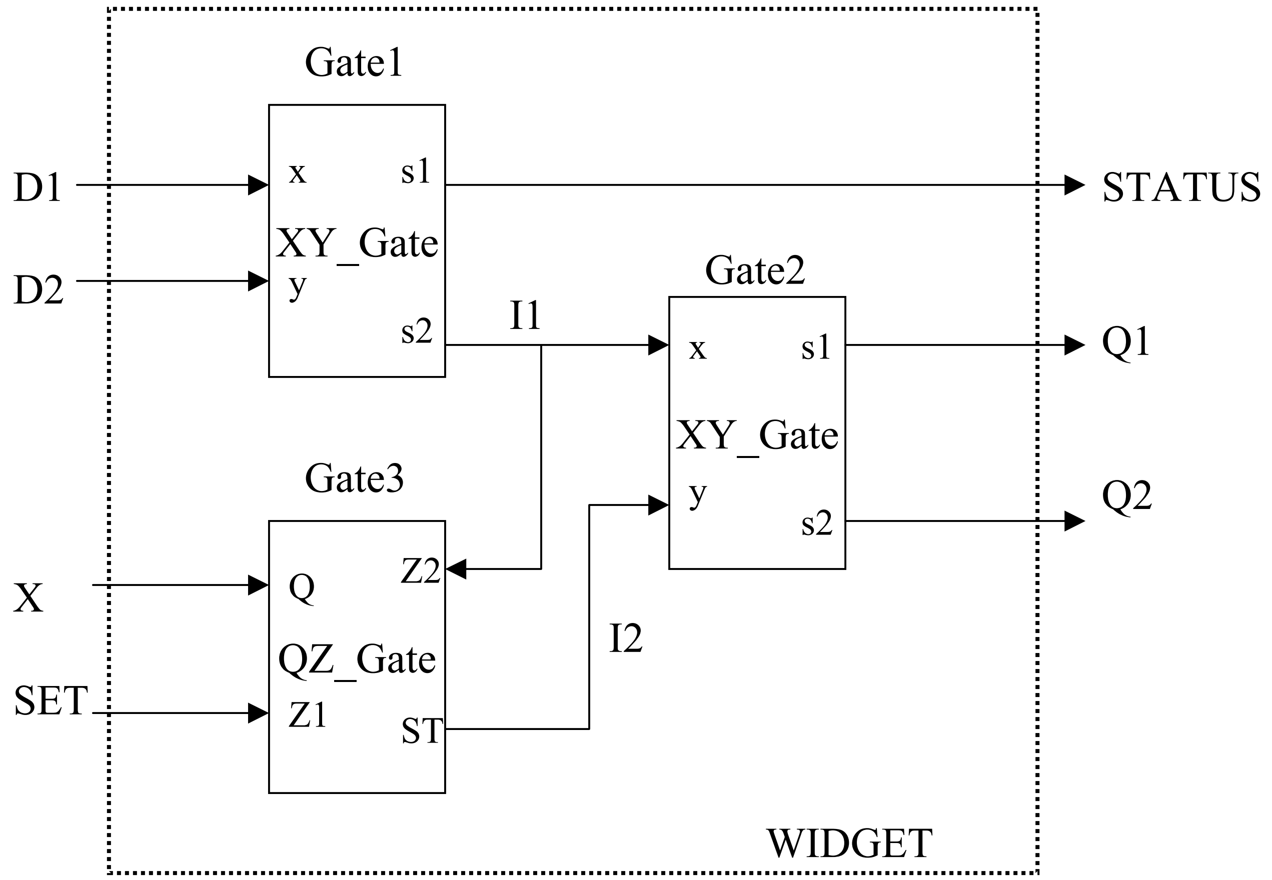
Hierarchy can dramatically simplify your design description and can make it much easier to re-use portions of the design in other projects.



Example of Hierarchical Design – The design entity "WIDGET" contains *instantiations* of other design entities.

A component must be *declared*, in the *declarations section* of the architecture body, before it is *instantiated* in the concurrent statements area.

```
component component_name
   port list (...);
end component;
```



There are 3 component instances, but only 2 different types of components. ***Thus we need 2 component declarations.***

The component declarations for our example:

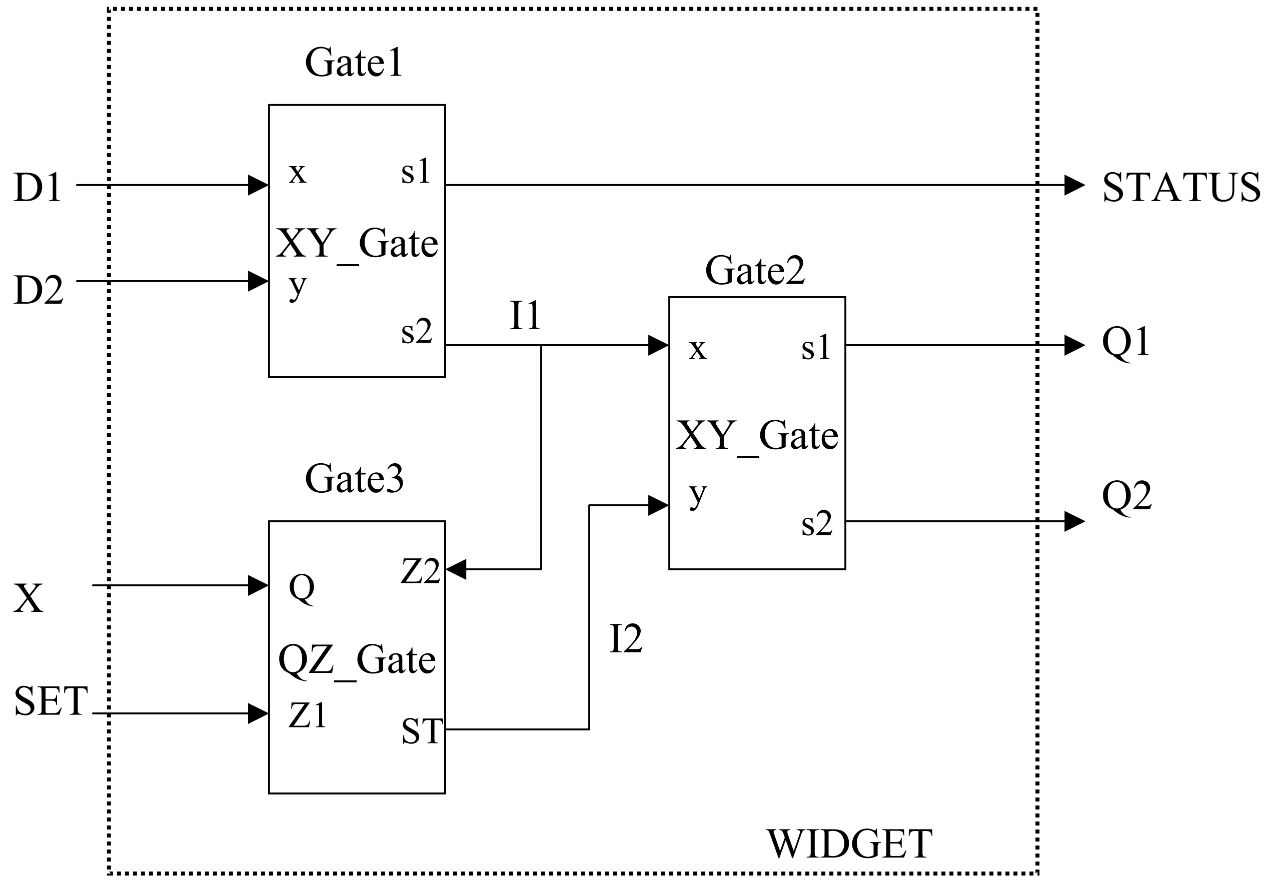
```
component XY_gate
   port(X, Y: in std_logic; s1, s2: out std_logic);
end component;

component QZ_gate
   port(Z1, Z2 : in std_logic; ST: out std_logic;
         Q : in std_logic_vector(7 downto 0));
end component;
```

Note: The *port list* should match the port list of the component's entity declaration.

To actually use a component it must be *instantiated*, in the *concurrent statements section* of the architecture body.

The basic template for component instantiation is:

```
instance_label: component_name
      port map (
          component_port1 => signal1,
          component_port2 => signal2,
...
          component_portn => signaln);
```



There are 3 components in total of all kinds, so we need 3 instantiation statements.

The component instantiations for our example:

```
Gate1: XY_Gate   port map (
         x=>D1, y=>D2, s2=>I1, s1=>STATUS);
Gate2: XY_Gate   port map (
         x=>I1, s2=>Q2, y=>I2, s1=>Q1);
Gate3: QZ_Gate   port map (
         Q=>X, Z2=>I1, Z1=>SET, ST=>I2);
```

Now, let’s put it all together…

```
entity widget is
        port ( D1, D2, SET : in std_logic;
                         X : in std_logic_vector(7 downto 0);
                Q1, Q2, STATUS : out std_logic);
end widget;


architecture A of widget is
signal I1, I2 : std_logic;

component XY_gate
        port(X, Y: in std_logic; s1, s2: out std_logic);
end component;

component QZ_gate
        port(Z1, Z2 : in std_logic; ST: out std_logic;
        Q : in std_logic_vector(7 downto 0));
end component;

begin
Gate1: XY_Gate port map (x=>D1, y=>D2, s2=>I1, s1=>STATUS);
Gate2: XY_Gate port map (x=>I1, s2=>Q2, y=>I2, s1=>Q1);
Gate3: QZ_Gate   port map (Q=>X, Z2=>I1, Z1=>SET, ST=>I2);
end A;
```

VHDL Libraries

Libraries are collections of signal definitions, and design entity definitions (for use as components).

Always declare all used libraries before the design entity

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

   entity my_design is
...........
```

IEEE STD_LOGIC Library

This is a set of standard packages defining commonly used data types and operations

Std_Logic_1164 Multivalue Logic System

Defines types and operations to deal with strong, weak and high-impedance strengths, and unknown values

– Std_logic
– Std_logic_vector

Always include this library in your VHDL designs

Example of *std_logic* package usage:

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

entity nandgate is
   port (A, B, OE: in std_logic; Y: out
std_logic);
end nandgate;

architecture arch1 of nandgate is
signal n: std_logic;
begin
   n <= not (A and B);
   Y <= n when OE = ‘1' else 'Z';
end arch1;
```

Altera LPM Library

This is a set of design entities which implement various logic blocks, ranging from simple (and gates, nor gates, etc) to complex (adders, multipliers, counters).

LPM stands for ***Library of Parametrized Modules***.

Various parameters of these modules (such as data_bus widths) can be adjusted (see the online MaxPlusII documentation for information on these parameters) using **GENERICs**.

You can use these as components in your VHDL designs.

Example of *lpm* package usage:

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

architecture arch1 of widget is
component lpm_inv
       generic (LPM_WIDTH: POSITIVE;
PORT (data : IN std_logic_vector(LPM_WIDTH-1 downto 0);
       result : OUT std_logic_vector(LPM_WIDTH-1 downto 0));

signal int1, int2 : std _logic_vector(3 downto 0));

begin
   inv1 : lpm_inv generic map (LPM_WIDTH => 4)
       port map( result => int1, data => int2);
end arch1;
```

OK, that's all the VHDL for now.
More to come next month…

In the meantime, just remember…

H is for *Hardware!*