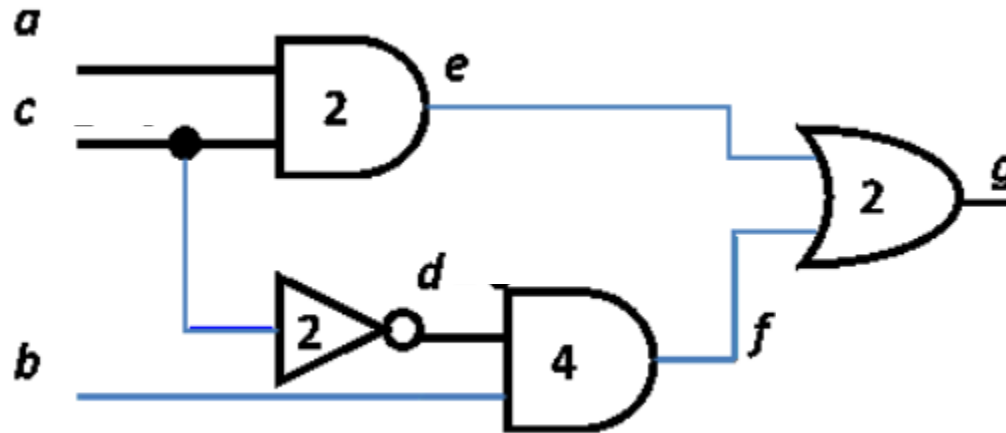


ECSE-323

Digital System Design

VHDL Lecture #2

An example: 2:1 multiplexer circuit



$$G = AC + B\overline{C}$$

```
entity mux is
    port( A, B, C: in std_logic;
          G: out std_logic);
end mux;

architecture implementation1 of mux is
    signal D,E,F : std_logic;
begin
    D <= not C;
    F <= D and B;
    E <= A and C;
    G <= E or F;
end implementation1;
```

We can simplify the description by combining the four assignment statements into a single statement.

```
entity mux is
    port( A, B, C: in std_logic;
          G: out std_logic);
end mux;

architecture implementation1 of mux is
    signal D,E,F : std_logic;
begin
    G <= (A and C) or (B and not C);
end implementation1;
```

Let's look at some other types of concurrent signal assignment statements:

- *Selected Signal* Assignment
- *Conditional Signal* Assignment
- *Component* Instantiation

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;
```

**Order is not
Important!**

Note: *All* conditions *must* be defined in a selected signal assignment.

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

Implementation of the 2:1 Multiplexer using Selected Assignment

```
entity mux is
    port( A, B, C: in std_logic;
          G: out std_logic);
end mux;

architecture implementation2 of mux is
begin
    with C select
        G <= A when '0',
            B when others;
end implementation2;
```


Conditional Signal 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

C when DSel = "10" else

D when others;

**Order is
Important!**

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";
```

Implementation of the 2:1 Multiplexer using Conditional Assignment

```
entity mux is
    port( A, B, C: in std_logic;
          G: out std_logic);
end mux;

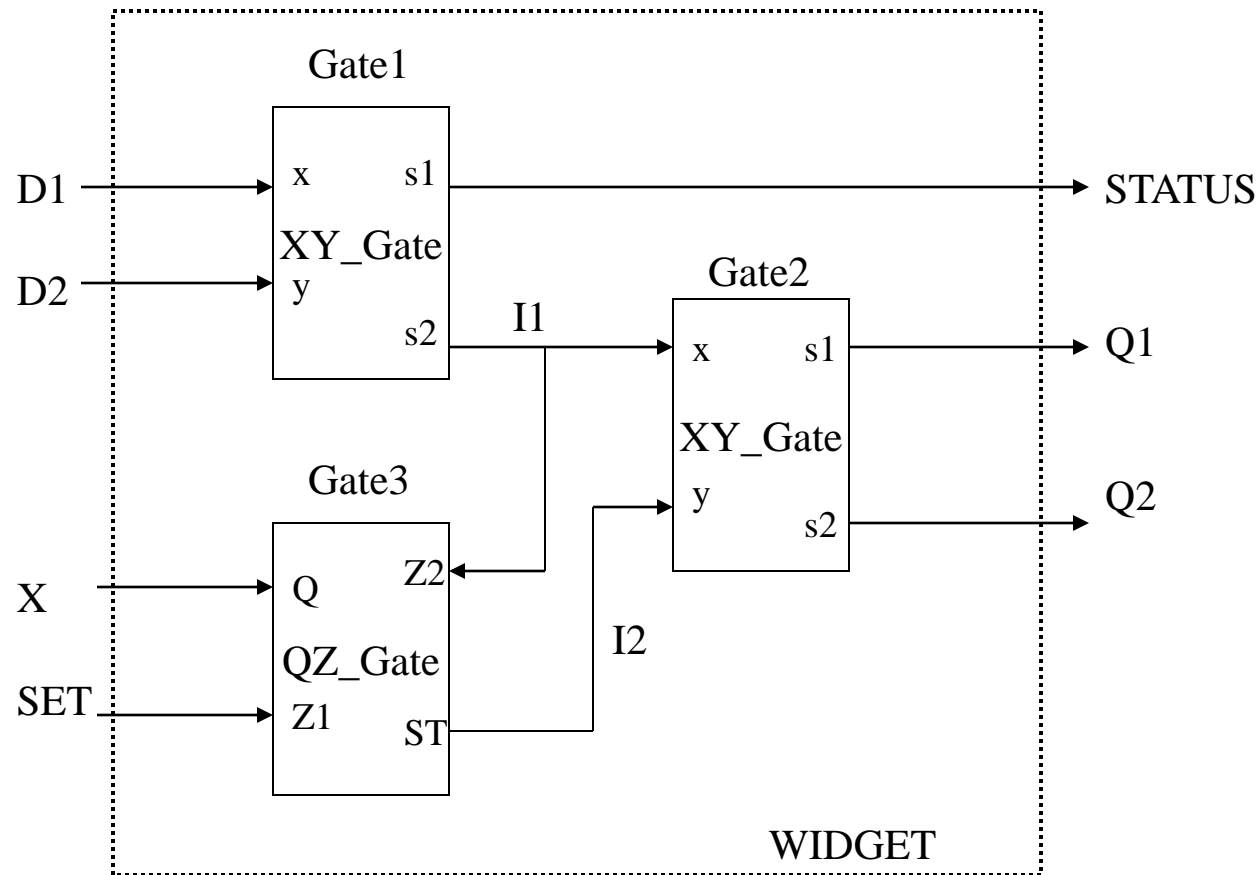
architecture implementation23 of mux is
begin
    G <= A when C = '0' else B;
end implementation3;
```

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.

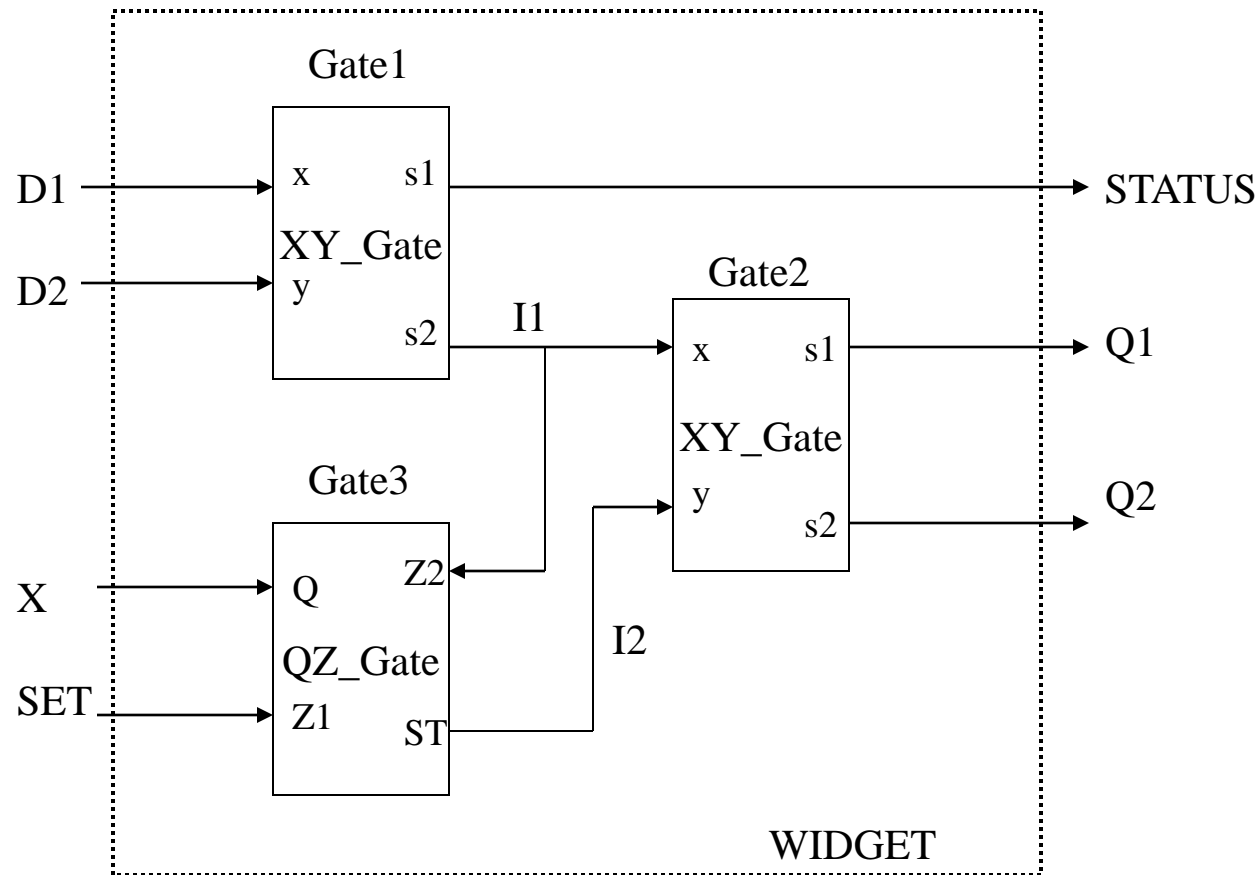
Components are a good method for implementing *design partitioning*.



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;
```

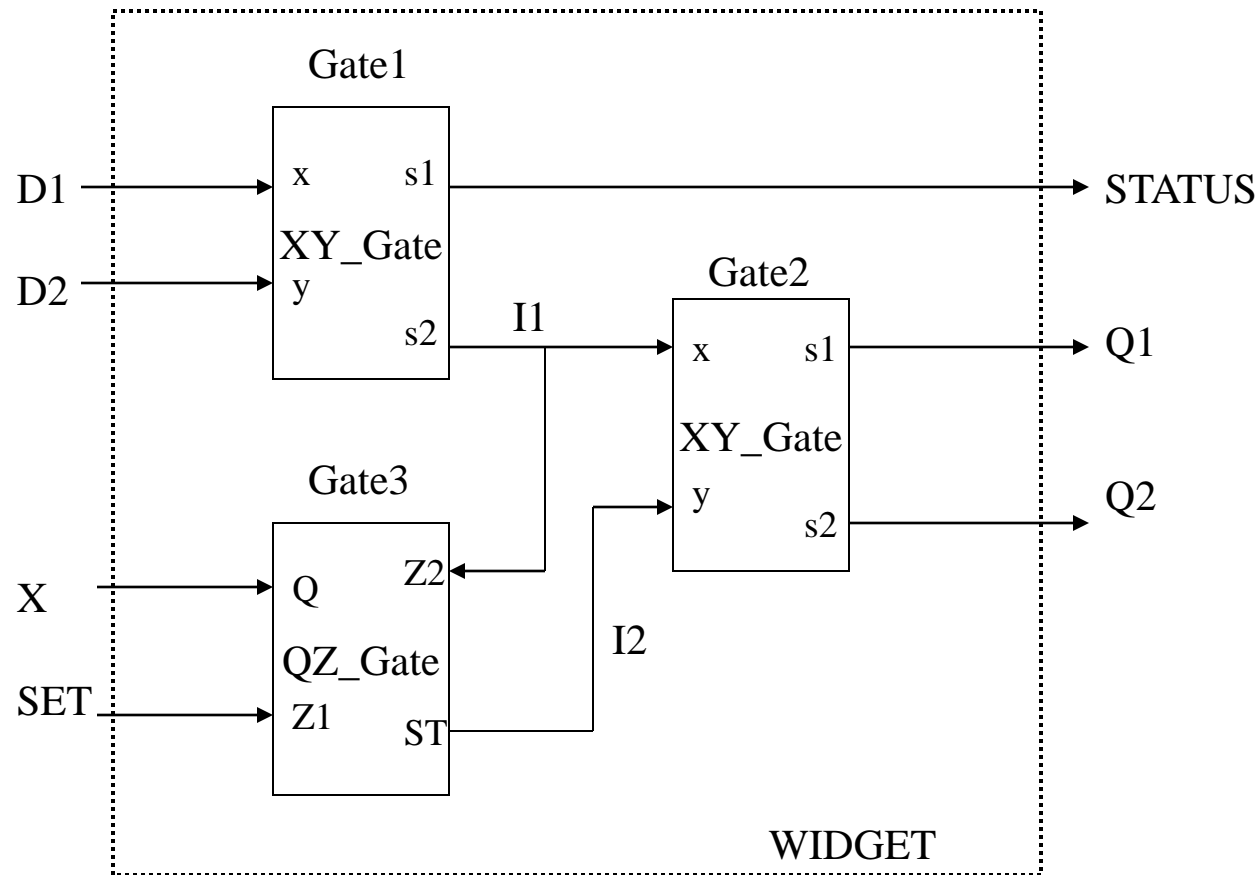
```
component QZ_gate  
    port(Z1, Z2 : in std_logic; ST: out std_logic;  
        Q : in std_logic_vector(7 downto 0));  
end component;
```

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;
```

Component Libraries

Often Libraries are compiled consisting of collections of useful components. One example is the Altera LPM library.

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
```

```
.....
```

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;
```

Instead of instantiating a component, we can use a ***process block*** to describe a specific function.

Process blocks allow you to describe a circuit using *sequential statements*, which are similar to those used in standard programming languages.

The disadvantage of process blocks over component instantiations is that the process blocks are local and cannot be re-used in other design entities.

Process block Syntax

```
process_label : process (sensitivity list)  
-- declarations  
begin  
  -- sequential statements  
end process;
```

The first line of the process statement includes a *label*, the *keyword process*, and an optional *list of signals*, known as the *sensitivity list*.

```
process_label : process (sensitivity list)
```

A process with a *sensitivity list* is evaluated during simulation whenever an event occurs on *any* of the signals in the sensitivity list (*and only then*).

If a process has *no sensitivity list* then it is evaluated when an event occurs on *any* signal.

In a typical circuit application, a process will include in its sensitivity list –

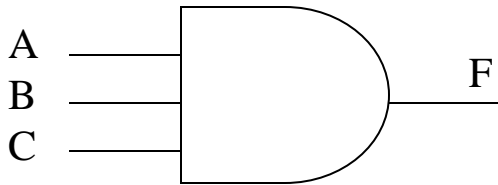
*all inputs that have **asynchronous** behavior.*

These asynchronous inputs may include

- **clock signal(s)**
- **reset signals**
- **inputs to blocks of combinational logic**

Example of a sensitivity list: 3-input AND gate

AND3_1 : process (A,B,C)



In purely combinational circuits, all inputs are considered to be asynchronous and should therefore appear in the sensitivity list.

Note: Outputs do not show up in sensitivity lists.

The second part of a process block is a set of *variable* declarations and initializations.

```
process_label: process (sensitivity list)  
-- variable declarations
```

We will talk about variables later in the course.

The main part of the process block involves a set of *Sequential Statements*, delimited by **begin** and **end**

```
process_label: process (sensitivity list)
-- variable declarations
begin
-- sequential statements
end process;
```

Sequential statements differ from concurrent statements in that they have order dependency.

A process block is *concurrent* from the point of view of the outside world, but *internally* it is *sequential*.

Statements inside a process block are evaluated in sequence from first to last.

Types of Sequential Statements

Types of sequential statements that can be used in process blocks:

- simple signal assignment statements
- if/then/else
- case
- for loop
- while loop

*Note: the last 4 structures can be used **only inside process blocks***



NOTE!!!

You *cannot* have

- *Component instantiations*
- *Selected Signal assignments*
- *Conditional Signal assignments*

in process blocks!

If/Then/Else statements:

```
if first_condition then
    --statements
elsif second_condition then --note spelling of elsif!
    --statements
else
    --statements
end if;  -- note space between end and if
```

The elsif and else are optional.

If/Then/Else statements:

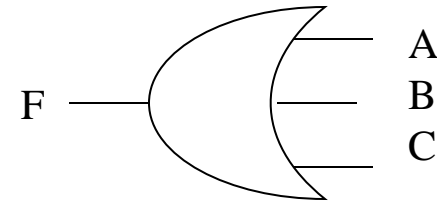
The *conditions* specified in an if-then-else construct must evaluate to a *Boolean* signal type.

If/Then/Else statements:

The *statement* parts of an if-then-else construct can contain any *sequential* VHDL statements, including other if-then-else statement constructs.

➔ you can *nest* if-then-else statements.

Example: 3-input OR gate



```
OR3_1 : process(A,B,C)
```

```
begin
```

```
    F <= '0'; -- provides a default value
```

```
    if A = '1' then F <= '1';
```

```
        elsif B = '1' then F <= '1';
```

```
        elsif C = '1' then F <= '1';
```

```
    end if;
```

```
end process;
```

(Of course, this is not a very efficient way to make an OR gate)

Notice that there are *multiple assignments* to the signal F . This is allowed! Only the final one takes effect.

```
OR3_1 : process(A,B,C)
```

```
begin
```

```
  F <= '0';
```

```
  if A = '1' then F <= '1';
```

```
    elsif B = '1' then F <= '1';
```

```
    elsif C = '1' then F <= '1';
```

```
  end if;
```

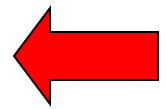
```
end process;
```

multiple signal assignments

A diagram with a red text label 'multiple signal assignments' at the top right. Two black arrows originate from this label. One arrow points to the assignment 'F <= '0';' in the first code line. The other arrow points to the assignment 'F <= '1';' in the 'if A = '1' then' branch of the code.

Multiple signal assignments are permitted
within a process block
(and *only* within a process block)

The *last assignment* to be evaluated in the sequential evaluation of the statements in the process block is the one that takes effect.



The first statement in the process block of the example is there to prevent the occurrence of *implied memory*.

```
OR3_1 : process(A,B,C)
begin
```

```
  F <= '0';
```

```
    if A = '1' then F <= '1';
```


```
        elsif B = '1' then F <= '1';
```

```
        elsif C = '1' then F <= '1';
```

```
    end if;
```

```
end process;
```

if not present, the circuit would hold the value of F when A=B=C=0



It is better to provide *default* values for combinational outputs rather than using an ending *else* clause as it is easier to read and less prone to error.

```
OR3_1 : process(A,B,C)
```

```
begin
```

```
    if A = '1' then F <= '1';
```

```
        elsif B = '1' then F <= '1';
```

```
        elsif C = '1' then F <= '1';
```

```
        else F <= '0';
```

```
    end if;
```

```
end process;
```

NOT A GOOD APPROACH!

You should keep things **simple** in a process block!

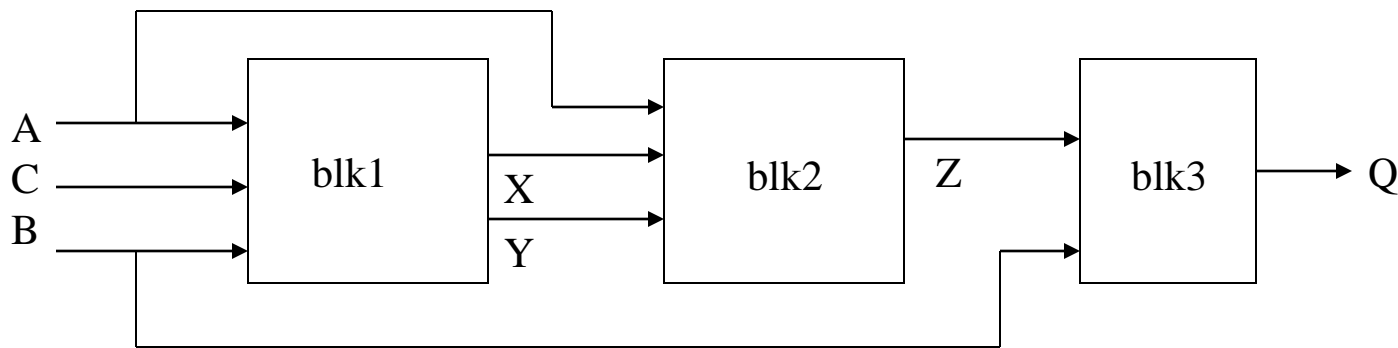
Don't try to do too much in one process block.

Avoid **spaghetti code** or **Byzantine** structures...



You can have *multiple* process blocks, so -

Divide your circuit into a number of functional blocks (like components) and use a *separate process block* to describe each one.



```
...  
blk1 : process (A,B,C) -- 2 in or and gates  
begin  
    X <= A or B;  
    Y <= C and B;  
end process;  
  
...  
blk2 : process (X,Y,A) -- 3 input OR gate  
begin  
    Z <= X or Y or A;  
end process;  
  
...  
blk3 : process (Z) -- implements a latch  
begin  
    if Z = '1' then Q <= B; end if;  
end process;
```

The following bit of VHDL code is
incorrect. **WHY?**

```
architecture EX1 of DSD is
  signal A,B,X,Y,Z : std_logic;
begin

  process (X,Y,A)
  begin
    Z <= X or Y or A;
    Y <= X and A;
  end process;

  process (Z,B)
  begin
    if Z = '1' then Y <= B; end if;
  end process;
```

The code on the previous slide is in *error* because *there is more than 1 assignment* to the signal *Y outside* of a process block!

```
architecture EX1 of DSD is
  signal A,B,X,Y,Z : std_logic;
begin
```

```
  process (X,Y,A)
  begin
```

```
    Z <= X or Y or A;
```

```
    Y <= X and A;
```

```
  end process;
```

```
  process (Z,B)
```

```
  begin
```

```
    if Z = '1' then Y <= B; end if;
```

```
  end process;
```

multiple signal assignments



You can have multiple assignments to a signal *within* a process block, since only one assignment actually takes effect.

This is not true outside of a process block!

In effect, the *multiple* assignments in a process count as a *single* concurrent assignment statement.

Case Statements

Case statements are a type of control statement that can be used as alternatives to if-then-else constructs.

```
case control_expression is  
when test_expression1 =>  
    --statements  
when test_expression2 =>  
    --statements  
when others =>  
    --statements  
end case;
```

The case statement is much like the selected signal assignment statement.

Their syntax is different however, so be careful to use the appropriate one.

Inside process block - *case statements*

Outside process block - *selected assignment*

The *test expressions* of a case statement *must be mutually exclusive*, meaning that no two test expressions are allowed to be true at the same time.

Case statements *must also include all possible conditions* of the control expression.

(The **others** expression can be used to guarantee that all conditions are covered.)

Simulation of Digital Systems

- Digital *simulation* is *event-driven*
- An event is a *transition* from one value to another
- A signal in VHDL can be thought of as a time-stamped *list* of events
- Digital simulation proceeds by looking at the list of events, finding the next event in time, and using *signal assignment statements* to generate new events caused by this event

How are propagation delays represented in VHDL?

- ***delta delay*** – signal assignment takes place an infinitesimal (delta) time after the events causing the assignment – used when no explicit delay is provided
- **after** clause – an explicit delay time is provided

Example: **A <= B or C *after* 2.2ns;**

Usually, you would not use the after clause in designing circuits, since you do not know how the hardware synthesis process will implement the circuit.

At this point only a functional simulation can be done, using delta delays.

After the synthesis is done, however, one can then get the timing information needed to do a timing simulation.

*OK, that's all the VHDL for now.
More to come in a few weeks...*

In the meantime, just remember...

H is for *Hardware!*

