Concurrent Signal Assignment Statements

From VHDL Essentials I, we discussed the **architecture** statement as one of the three components of a VHDL file

```
architecture arch_name of entity_name is
  declarations
  begin
  concurrent_stmt;
  concurrent_stmt;
end arch_name;
```

So far, we looked at only ***simple signal assignment*** statements as an example of a valid 'concurrent_stmt' within the architecture body

```
architecture beh of demo is
  signal ab: std_logic;
  begin
  ab <= a and b;
  x <= ab;
  y <= not ab;
end ok_arch;
```

Concurrent Signal Assignment Statements

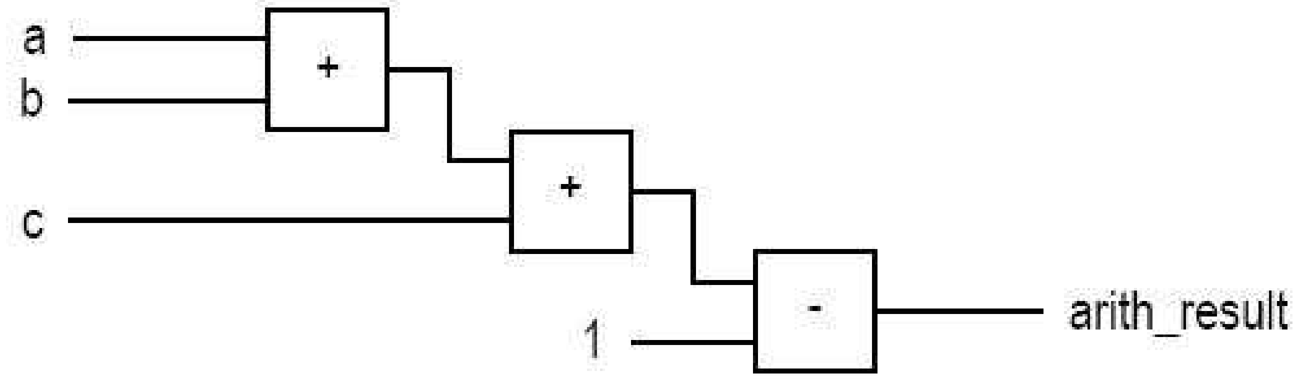
In this lecture, we consider two other types of 'concurrent_stmt', in particular, **conditional signal assignment** and **selected signal assignment**

Remember that all concurrent signal assignment statements describe hardware components that operate in **parallel**

This will be true when we discuss the **process** statement as well

For example, the following signal assignment is implemented as shown, and continuously re-computes *arith_out* as values on the wires labeled *a*, *b* and *c* change

```
arith_out <= a + b + c - 1;
```



Other signal assignments, if included, would operate **in parallel** with this circuit

Concurrent Signal Assignment Statements

One final note on *simple signal assignment*

Although it is syntactically correct to use a signal name on **both** sides of an assignment statement, NEVER do it!

```
q <=  (d and en) or ((not q) and (not en));
```

This describes a circuit which assigns to *q* the value of *d* when *en* is ’1’, otherwise it assigns the *inverse* of itself *not q*

This statement forms a *combinational feedback loop* and will not be synthesized and behave as you expect

You will encounter plenty of instances where you want to do this type of assignment, however, this is not the correct way to do it

We will discuss how to properly describe a circuit with this type of assignment later when we discuss D flip-flops

Selected Signal Assignment Statements

Selected signal assignment describes a circuit that implements a ***case statement*** in a programming language

```
with select_expression select
   signal_name <= value_expr_1 when choice_1,
                  value_expr_2 when choice_2,
                  value_expr_3 when choice_3,
                  ...
                  value_expr_n when others;
```

The *select_expression* is usually of type *std_logic* or *std_logic_vector*

choice_x are usually constants such as "00", "01", "10" and "11"

The *choices_x* must be **mutually exclusive** and **all inclusive**, i.e., always use **others** as the condition in the last clause to ensure this

The *selected signal assignment* statement is VERY COMMONLY used to describe a MUX-based circuit

Selected Signal Assignment Statements

```
architecture sel_arch of mux4 is
   begin
   with s select
     x <= a when "00",
          b when "01",
          c when "10",
          d when others;
end sel_arch;
```

Describing a **binary decoder** (n-to-2^n) is another common usage scenario

```
architecture sel_arch of decoder4 is
   begin
   with s select
     x <= "0001" when "00",
          "0010" when "01",
          "0100" when "10",
          "1000" when others;
end sel_arch;
```

Selected Signal Assignment Statements

Selected signal assignment can also be used to describe a circuit which uses a **truth table** as the source

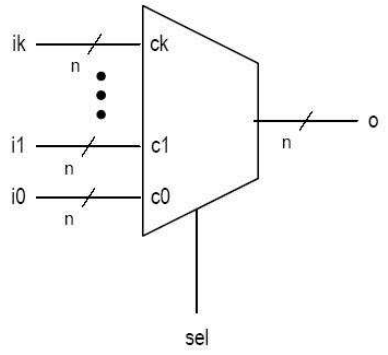
input a b	output y
0 0	0
0 1	1
1 0	1
1 1	1

```
library ieee;
use ieee.std_logic_1164.all;
entity OR_gate_truth_table is
   port(
      a, b: in std_logic;
      y: out std_logic
   );
end OR_gate_truth_table;
```

Selected Signal Assignment Statements

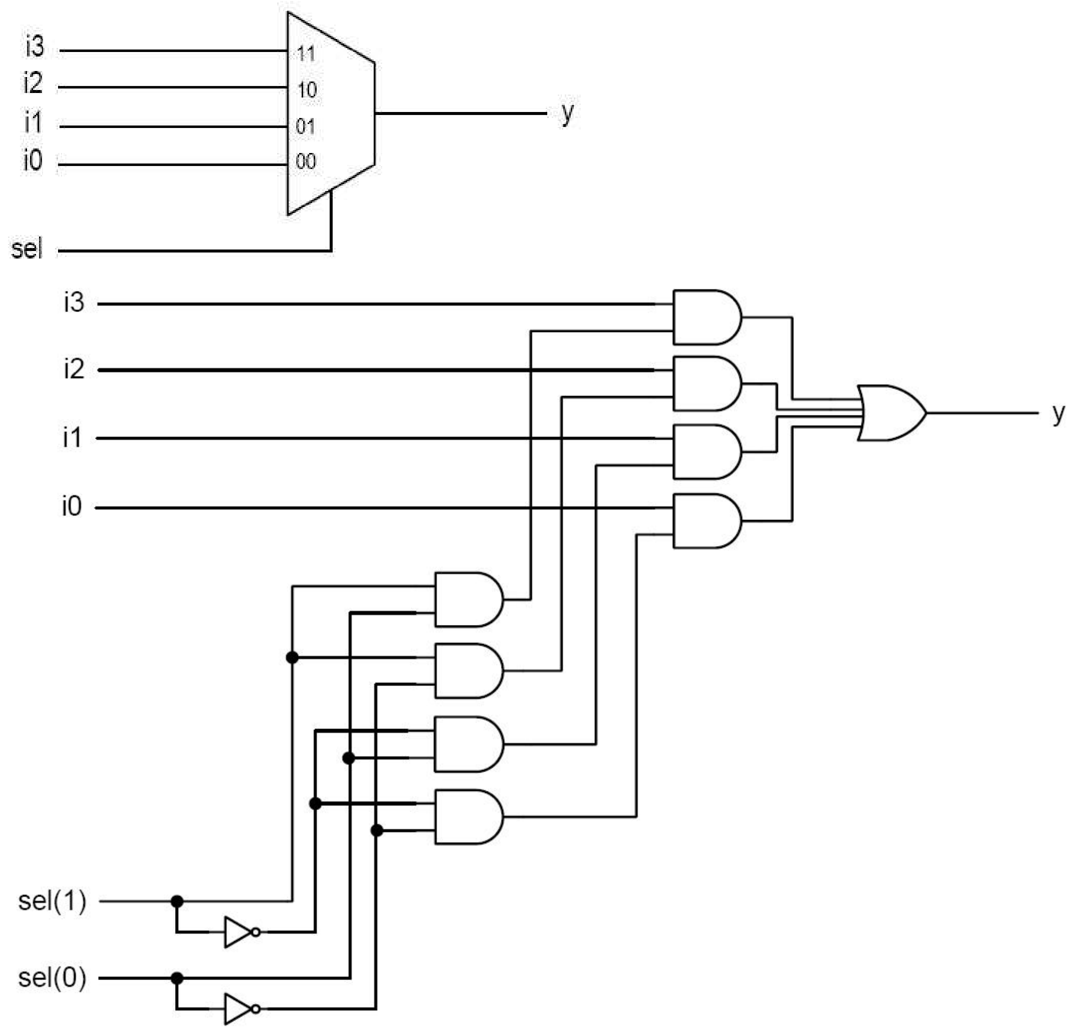
```
architecture beh of OR_gate_truth_table is
   signal tmp: std_logic_vector(1 downto 0);
   begin
   tmp <= a & b; -- concatenate a and b
   with tmp select
      y <= '0' when "00", -- rows of the truth table
           '1' when "01",
           '1' when "10",
           '1' when others;
end beh;
```

The conceptual implementation of a *selected signal assignment* is simply a **MUX**



Selected Signal Assignment Statements

When synthesized, a 4-to-1 MUX can be mapped into logic gates

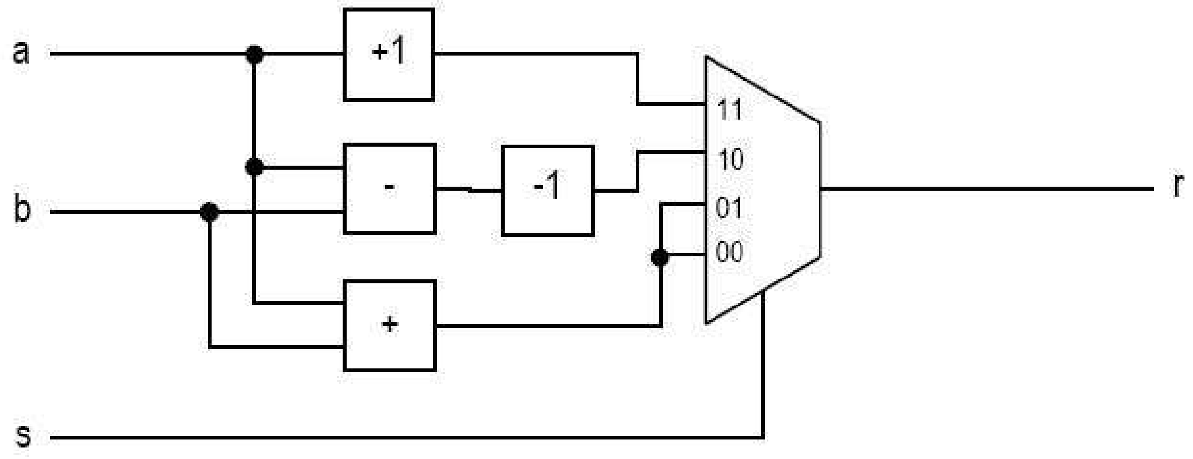


This circuit selects either *i0*, *i1*, *i2* or *i3* depending on the values of *sel(0)* and *sel(1)*

Selected Signal Assignment Statements

More complex MUX-based multi-bit arithmetic circuits can also be described

```
signal a, b, r: unsigned(7 downto 0);
signal s: std_logic_vector(1 downto 0);
...
with s select
   r <= a+1 when "11",
        a-b-1 when "10",
        a+b when others;
```



Conditional Signal Assignment Statements

Conditional signal assignment describes a circuit that implements an ***if stmt*** in a programming language

```
sig_name <=
        value_expr_1 when boolean_expr_1 else
        value_expr_2 when boolean_expr_2 else
        value_expr_3 when boolean_expr_3 else
        ...
        value_expr_n
```

Similar to *if stmts*, each of the *boolean_expr_x* generate **1** or **0** with the first one that generates a 1 causing the *value_expr_x* to be assigned to the *sig_name* signal

Only use *conditional signal assignment* when it is NOT possible to use *selected signal assignment*

The synthesis tool will generally use more logic gates to implement *conditional signal assignment* because of the **priority** that exists among the choices

boolean_expr_1 takes priority over *boolean_expr_2*

Conditional Signal Assignment Statements

A **priority encoder** is a good example of when you should use *conditional signal assignment*

A priority encoder checks the input requests and generates the code for the request of *highest priority*

input	**output**	
r	code	active
1 – – –	11	1
0 1 – –	10	1
0 0 1 –	01	1
0 0 0 1	00	1
0 0 0 0	00	0

There are four input requests, *r(3)*, ..., *r(0)*

The outputs include a 2-bit signal (*code*), which is the binary code of the highest priority request and a 1-bit signal *active* that indicates if there is an active request

Here, *r(3)* has the highest priority, i.e., when asserted, the other three requests are ignored and the *code* signal becomes "11"

When *r(3)* is **not** asserted, the second highest request, *r(2)* is given priority

The *active* signal is used to distinguish between the cases when *r(0)* is asserted and the case in which NO request is asserted

Conditional Signal Assignment Statements

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

entity prio_encoder42 is
  port(
       r: in std_logic_vector(3 downto 0);
       code: out std_logic_vector(1 downto 0);
       active: out std_logic);
end prio_encoder42;

architecture beh of prio_encoder42 is
   begin
   code <= "11" when (r(3)='1') else
           "10" when (r(2)='1') else
           "01" when (r(1)='1') else
           "00";
   active <= r(3) or r(2) or r(1) or r(0);
end beh;
```

Conditional Signal Assignment Statements

The *conditional signal assignment* statement implements a priority structure and requires additional logic to implement the *priority routing network*

Selected signal assignment requires the first two of the following circuits to be implemented while *conditional signal assignment* requires all three

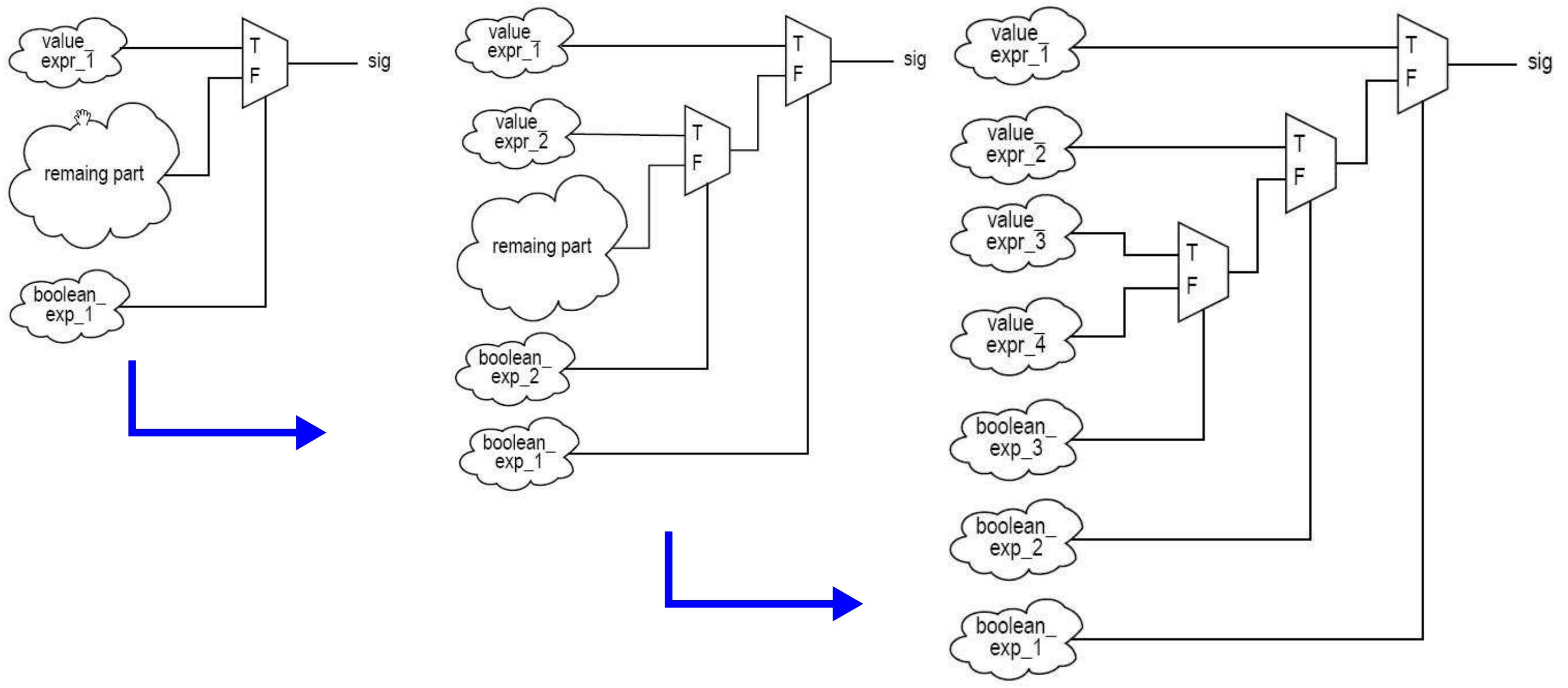
- Value expression circuits
- Boolean expression circuits
- Priority routing network

A *layered* sequence the MUXes are used to implement the *priority routing network* which determines which *value expression* circuit is connected to the output

The outputs of circuits that implement the *Boolean* expressions are used to drive the *select* inputs to the layered sequence of MUXes

Conditional Signal Assignment Statements

```
    sig <= value_expr_1 when boolean_expr_1 else
           value_expr_2 when boolean_expr_2 else
           value_expr_3 when boolean_expr_3 else
           value_expr_4;
```



Adding **when** clauses increases the overall combinational delay of the circuit, so you will be limited in how many **when** clauses you can use