# Module 5: Structural Style Programming In VHDL

- Structural Style, Port Mapping, and Generic Map
- VHDL Modularity
- Array and Types in VHDL

# Structural Style, Port Mapping, and Generic Map

## Structural Style Programming

Structural Style Programming in VHDL allows designers to build digital circuits using basic components connected to form a more complex system. In this approach, circuits are represented as collections of entities connected in a specific way to achieve the desired function.

## Port Mapping

Port mapping is the process of associating the ports of a VHDL component (entity) with signals in the architecture. This allows entities to be connected with the actual circuit in the design.

### Important Points

- **Entity Definition**: An entity must be defined first, which includes its ports and data types.
- **Port-Map List**: Ports of the entity are mapped to corresponding signals in the architecture.
- **Port Mapping Order**: Mapping must follow the order defined in the entity.
- **Signal Declaration**: Signals used in port mapping must be declared in the architecture.

### Example

```
entity AND2 is
    port (
        A, B: in std_logic;
        Y: out std_logic
    );
end entity;

-- Port mapping
architecture RTL of AND2 is
begin
```

```
    Y <= A and B;
end architecture;

-- Using the entity with port mapping
D1: AND2 port map (
    A => input_signal_A,
    B => input_signal_B,
    Y => output_signal
);
```

# Generic Map

Generic map is the process of mapping generic values in an entity to corresponding values in the architecture. Generics are parameters that can be set for an entity to configure the behavior or characteristics of a component.

## Important Points

- **Generic**: Parameters used to modify characteristics of an entity.
- **Generic Map**: Defines the values for generics when instantiating an entity.
- **Default Value**: Generics often have default values, but they can be overwritten during instantiation.

## Example

```
entity Counter is
    generic (
        WIDTH: positive := 8; -- Default value for WIDTH is 8
        ENABLED: boolean := true -- Default value for ENABLED is true
    );
    port (
        clk: in std_logic;
        reset: in std_logic;
        count: out std_logic_vector(WIDTH-1 downto 0)
    );
end entity;

-- Generic map when instantiating the entity
architecture RTL of MyDesign is
    signal my_counter_output: std_logic_vector(7 downto 0);
```

```
begin
    my_counter_inst: Counter
        generic map (
            WIDTH => 8, -- Generic value WIDTH is reset to 8
            ENABLED => true -- Generic value ENABLED is reset to true
        )
        port map (
            clk => system_clock,
            reset => reset_signal,
            count => my_counter_output
        );
end architecture;
```

# VHDL Modularity

We will build a 4-bit Ripple Carry Adder using 4 Full Adders in Structural Style Programming. Each Full Adder's carry-out serves as the carry-in for the next Full Adder, creating a ripple effect in addition.

## Step 1 - Full Adder Entity

Inside full adder entity, we will declare the input and output ports. The input ports are A, B, and Cin, and the output ports are Sum and Cout.

```
entity full_adder is
    port(
        A, B, Cin : in std_logic;
        Sum, Cout : out std_logic
    );
end entity full_adder;

architecture structural of full_adder is
begin
    Sum <= A xor B xor Cin;
    Cout <= (A and B) or (B and Cin) or (A and Cin);
end architecture structural;
```

## Step 2 - Ripple Carry Adder Architecture

Next step is to create the architecture for the 4-bit Ripple Carry Adder. We will instantiate 4 Full Adders and connect them in a way that the carry-out of one Full Adder is connected to the carry-in of the next Full Adder.

```
entity ripple_carry_adder is
    port(
        A, B : in std_logic_vector(3 downto 0);
        Sum : out std_logic_vector(3 downto 0);
        Cout : out std_logic
    );
```

```
end entity ripple_carry_adder;

architecture structural of ripple_carry_adder is
    component full_adder
        port(
            A, B, Cin : in std_logic;
            Sum, Cout : out std_logic
        );
    end component full_adder;

    signal C : std_logic_vector(3 downto 0);
begin
    FA0 : full_adder port map(A(0), B(0), '0', Sum(0), C(0));
    FA1 : full_adder port map(A(1), B(1), C(0), Sum(1), C(1));
    FA2 : full_adder port map(A(2), B(2), C(1), Sum(2), C(2));
    FA3 : full_adder port map(A(3), B(3), C(2), Sum(3), Cout);
end architecture structural;
```

Based on the above code, we can see that the carry-out of each Full Adder is connected to the carry-in of the next Full Adder. This creates a ripple effect in addition. Therefore the port map explanation is as follows:

# FA0 - Full Adder 0

- A(0) and B(0) are the input bits for the first Full Adder.
- '0' is the carry-in for the first Full Adder.
- Sum(0) is the output sum of the first Full Adder.
- C(0) is the carry-out of the first Full Adder.

# FA1 - Full Adder 1

- A(1) and B(1) are the input bits for the second Full Adder.
- C(0) is the carry-in for the second Full Adder.
- Sum(1) is the output sum of the second Full Adder.
- C(1) is the carry-out of the second Full Adder.

# FA2 - Full Adder 2

- A(2) and B(2) are the input bits for the third Full Adder.
- C(1) is the carry-in for the third Full Adder.
- Sum(2) is the output sum of the third Full Adder.
- C(2) is the carry-out of the third Full Adder.

## FA3 - Full Adder 3

- A(3) and B(3) are the input bits for the fourth Full Adder.
- C(2) is the carry-in for the fourth Full Adder.
- Sum(3) is the output sum of the fourth Full Adder.
- Cout is the carry-out of the fourth Full Adder.

With this architecture, we have successfully implemented a 4-bit Ripple Carry Adder using 4 Full Adders in Structural Style Programming.

# Step 3 - Testbench

To test the functionality of the 4-bit Ripple Carry Adder, we will create a testbench that provides input values to the adder and checks the output values.

```
entity tb_ripple_carry_adder is
end entity tb_ripple_carry_adder;

architecture testbench of tb_ripple_carry_adder is
    signal A, B : std_logic_vector(3 downto 0);
    signal Sum : std_logic_vector(3 downto 0);
    signal Cout : std_logic;
    signal clk : std_logic := '0';

    component ripple_carry_adder
        port(
            A, B : in std_logic_vector(3 downto 0);
            Sum : out std_logic_vector(3 downto 0);
            Cout : out std_logic
        );
    end component ripple_carry_adder;

begin
    dut : ripple_carry_adder port map(A, B, Sum, Cout);

    process
    begin
        A <= "0000"; B <= "0000"; wait for 10 ns;
        A <= "0001"; B <= "0001"; wait for 10 ns;
        A <= "0010"; B <= "0010"; wait for 10 ns;
        A <= "0011"; B <= "0011"; wait for 10 ns;
```

```
        A <= "0100"; B <= "0100"; wait for 10 ns;
        A <= "0101"; B <= "0101"; wait for 10 ns;
        A <= "0110"; B <= "0110"; wait for 10 ns;
        A <= "0111"; B <= "0111"; wait for 10 ns;
        A <= "1000"; B <= "1000"; wait for 10 ns;
        A <= "1001"; B <= "1001"; wait for 10 ns;
        A <= "1010"; B <= "1010"; wait for 10 ns;
        A <= "1011"; B <= "1011"; wait for 10 ns;
        A <= "1100"; B <= "1100"; wait for 10 ns;
        A <= "1101"; B <= "1101"; wait for 10 ns;
        A <= "1110"; B <= "1110"; wait for 10 ns;
        A <= "1111"; B <= "1111"; wait for 10 ns;
        wait;
    end process;

end architecture testbench;
```

In the testbench, we provide different input values to the 4-bit Ripple Carry Adder and observe the output values. The testbench will simulate the addition process and verify the correctness of the adder. The output values can be checked against the expected results to ensure the adder is functioning correctly

# Array and Types in VHDL

## Array

In VHDL, an array is a collection of elements that have the same data type. You can think of an array as a variable that has many elements with the same data type, and these elements are indexed for access. The index can be a number or another indexable type, such as integer, natural, or std_logic_vector. Arrays can have one dimension (one-dimensional array) or more (two-dimensional array, three-dimensional array, and so on). Two-dimensional arrays are often used to represent tables or matrices.

## Types

In VHDL, a type is a definition used to declare a new data type. Types can be used to define complex data types, such as arrays or records, or as types used to declare variables, ports, or signals. Types can also be used to describe the properties and structure of data. VHDL has predefined data types, such as std_logic, std_logic_vector, integer, and others, but we can also create our own custom data types. Types that are predefined or embedded in the VHDL library are called "built-in types," while types that we define ourselves are called "derived types."

### Example

```
-- Custom type definition
type color is (red, green, blue, yellow, black, white);

-- Variable declaration using custom type
signal primary_color: color;

-- Array declaration using custom type
type color_array is array (natural range <>) of color;

-- Array instantiation
signal color_table: color_array(0 to 3);
```

In the example above, we define a custom type `color` with specific values. We then declare a signal `primary_color` using this custom type. We also define an array type `color_array` that can hold elements of type `color`, and we instantiate an array `color_table` with a range of 0 to 3 using this array type.