# VHDL Presentation

Alex Boswell

## What is a Hardware Description Language?

The short and sweet:

- Hardware Description Languages are a textual description of a digital circuit
- Hardware Description Languages are also used to simulate a digital circuit
- Hardware Description Languages borrow syntax and style from procedural programming languages (C, C++, Ada).
- Programming languages are just high level abstractions of instructions for a CPU to execute.
- Hardware Description Languages can ultimately be used to design a CPU
- Traditionally HDLs were used to describe the behavior of Application Specific Integrated Circuits (ASICs) and provide simulations for said Integrated Circuits.
- Since the mid 1990s, Hardware Description Languages can use synthesis software to create a digital circuit inside of a programmable logic device (PLD, CPLD, FPGA).
- The two dominant Hardware Description languages are SystemVerilog and VHDL.

## Basic Structure of a VHDL file

A typical VHDL design file has the following regions

- library declaration
- package imports
- entity declaration
- architecture declarative region
- architecture behavioral region

```vhdl
library libraryname; --library declare
use libraryname.packagename.all; --package import

entity entity_name is --entity declaration
    port(
        identifier : mode type;
        identifier : mode type
    );
end entity;

architecture arch_name of entity_name is
    signal identifier : type; --architecture declarative region
    constant identifier : type := initialvalue;
    function name is blah blah blah;
    procedure name is blah blah blah;
    type type_name is (definition);
begin
    behavioral stuff --architecture behavioral region
end architecture;
```

Describe the ports, signals, constants, functions, procedures, types to the class.

Common packages:

- std_logic_1164
- numeric_std
- math_real

## Basic Operators

Inside of the architecture behavioral region you can assign logic and/or values to signals and ports using <=.

The basic operators in VHDL are **AND, OR, XOR, NAND, NOR, XNOR, NOT**.

Given a VHDL file with input ports $a$,$b$ and output port $f$:

```
f <= a xor b;
```

Because this is not a programming language, two things must be made clear:

1. Given multiple outputs, the order of assignment does not matter since they represent individual gates
2. Because this order does not matter, you cannot reassign a signal or port. VHDL sees this as a conflicting driver rather than a reassignment.

When it comes to the order of operations, there is no set precedent, except for **NOT** (this always goes first, so **ALWAYS** use parenthesis to denote the logic grouping.

```
f <= ((not a and b) or (c nand d)) xor e;
```

This is the equivalent of

$$\left(\left(\overline{A} \cdot B\right) + \left(\overline{C \cdot D}\right)\right) \oplus E$$

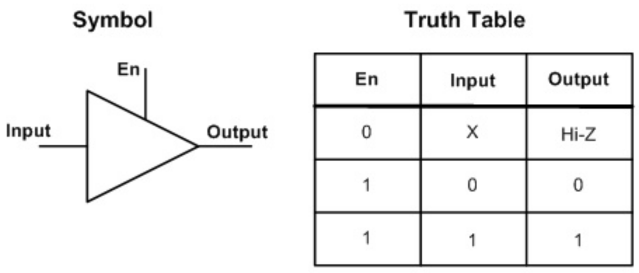
## The Fundamental Type

While there is a built-in type **bit**, the most used type due to its versatility is **std_logic**. This type is imported with the **ieee.std_logic_1164** package.

While bit has two values, either '1' or '0', std_logic has a total of nine values for superior simulation capabilities. Of those nine, only four can be used for synthesis.

- '1' A logic level high (Synthesizable)
- '0' A logic level low (Synthesizable)
- 'Z' A high impedance output (Synthesizable)
- '-' A don't care input (Synthesizable)
- 'U' Uninitialized
- 'X' Unknown
- 'W' Weak Unknown
- 'L' Weak logic level low
- 'H' Weak logic level high

The '1' and '0' are, hopefully, self explanatory in what they do. The 'Z' operator forces an output to be the high impedance value. Neither a HIGH or a LOW. These are commonly found in tri-state buffers. Tri-state buffers are mainly used in busses that connect multiple devices, in order to ensure only one device is transmitting on the bus.



| En | Input | Output |
|---|---|---|
| 0 | X | Hi-Z |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

Figure 1: Tri-state buffer and truth table

The '-' is used to simplify repeated inputs and will be explored later.

## Standard Logic Vector

The **std_logic_vector** is an array of the type **std_logic**. Unlike traditional programming languages, you must define the constraints of the array. The overwhelming majority of the time you want to have the vector is the big endian format, thus you declare it as a downto.

```
a: in std_logic_vector(7 downto 0); --a(7)...a(0)
b: in std_logic_vector(2 to 5); --b(2)...b(5)
```

The way operators work on vectors is different than that of scalar values. If two vectors of equal length are operated on, the result is a vector of the same size in which each element is operated with the corresponding element. If a vector is operated on by a scalar, the result is a vector of equal length in which each element has been operated on by the scalar. If an operator is before the vector, every element in the vector is operated together, making a scalar. This is a mouthful, so here is an example:

```
1 signal a: std_logic_vector(3 downto 0) := "1011";
2 signal b: std_logic_vector(3 downto 0) := "1101";
3 signal c: std_logic := '1';
4 signal f: std_logic_vector(3 downto 0);
5 signal g: std_logic_vector(3 downto 0);
6 signal h: std_logic;
7 f <= a and b; -- "1001";
8 g <= a xor c; -- "0100";
9 h <= and b; -- b(3) and b(2) and b(1) and b(0) = '0'
```

As can be seen in the comments for line 9, given a vector ”b”, you can index vector b with b(i) where i is an integer within the specified range of the vector.

Two ways to join elements into a vector are aggregation and concatenation. For aggregation, you can define specific indexes of a vector to be something, and then mass assign the remainder to be whatever. This is particularly helpful if you have a massive vector. Concatenation is mainly used to join vectors together.

```
signal s1: std_logic_vector(3 downto 0) := "1011";
signal s2: std_logic_vector(3 downto 0) := "1101";
signal b1: std_logic_vector(7 downto 0);
signal ag1: std_logic_vector(3 downto 0);
signal ag2: std_logic_vector(7 downto 0);
b1 <= s1 & s2; -- "10111101"
ag1 <= (3|1 => '1', others => '0'); -- "1010"
ag2 <= (7 downto 5 => 'Z', 4|2|0 => '-', others => s2(2));
-- "ZZZ-1-1-"
```

## Signals as wires

Signals are constructs in VHDL that can act as either wires or registers. Right now, only the wire usage shall be explored. The declaration for a signal occurs in the architecture declarative body.

```vhdl
architecture rtl of entity_name is
    signal identifier : std_logic := '0';
begin

end architecture;
```

Here is an example of a Full Adder using signals as wires and the schematic equivalent.

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

entity fullAdder is
    port(
        a,b,c: in std_logic; --input/output declaration
        sum,carry: out std_logic
    );
end fullAdder;

architecture behavior of fullAdder is
    signal w1,w2,w3: std_logic; --intermediate variables
begin

  w1 <= a xor b; --logic
  w2 <= w1 and c;
  w3 <= a and b;
  sum <= w1 xor c;
  carry <= w2 or w3;

end behavior;
```

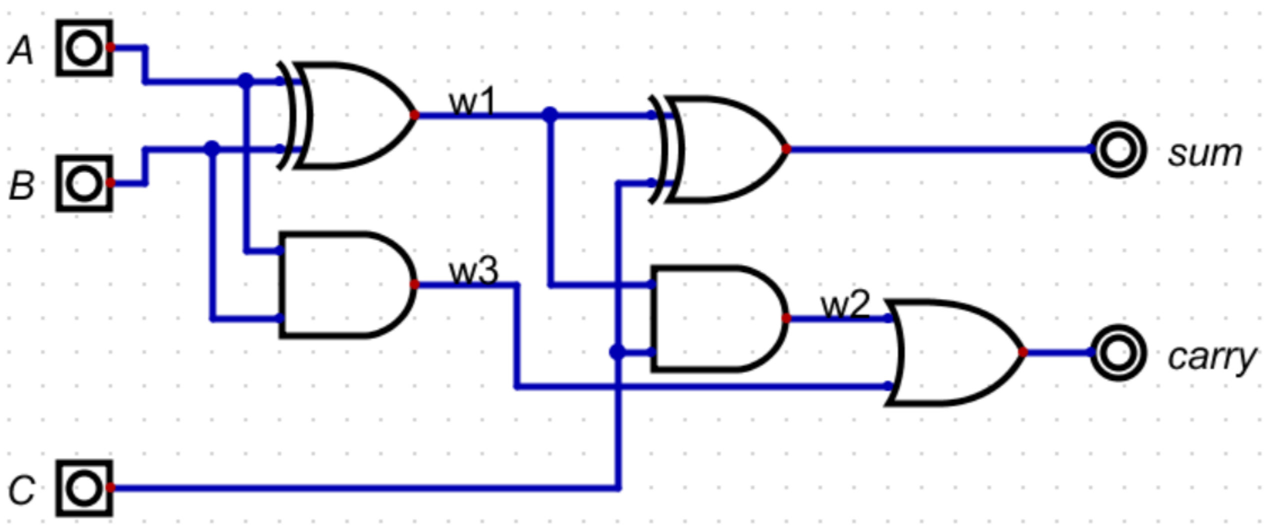


Figure 2: Full Adder

## Select Statement

Select is the first of the concurrent statements we will go over. Select is analogous to a switch statement in C++. The main purpose of this is to make truth tables and should be the primary usage of this. The main Syntax for it is:

```vhdl
with expression select
    target <= value when choice,
              value when choice,
              value when choice,
              value when others;
-- when others is recommended for simualation purposes.
```

Example Multiplexer:

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

entity mux is
    port(
        D       : in  std_logic_vector(3 downto 0);
        sel     : in  std_logic_vector(1 downto 0);
        mux_out : out std_logic
    );
end mux;

architecture rtl of mux is

begin
with SEL select
    mux_out <= D(3) when "11",
               D(2) when "10",
               D(1) when "01",
               D(0) when "00",
               '0' when others;
end architecture;
```

## When Statement

The when statement is analogous to an if, else if, else construct. As such, it has more freedom for the equality matching, just like an if statement vs a switch in a real programming language.

Syntax:

```vhdl
target <= value when condition1 else
          value when condition2 else
          value when condition3 else
          value;
```

What makes it more powerful than the select statement is that it can use equality operators and any signal to achieve the desired output for the target.

Here is an example of a comparator:

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

entity comparator is
    generic(
        n : integer := 8
    );
    port(
        a, b : in std_logic_vector(n-1 downto 0);
        eq, neq, lt, lte, gt, gte : out std_logic
    );
end;

architecture synth of comparator is

begin
    eq  <= '1' when (a = b)  else '0'; --equal to
    neq <= '1' when (a /= b) else '0'; --not equal to
    lt  <= '1' when (a < b)  else '0'; --less than
    lte <= '1' when (a <= b) else '0'; --less than equal to
    gt  <= '1' when (a > b)  else '0'; --greater than
    gte <= '1' when (a >= b) else '0'; --greater than equal to
end architecture;
```

With logical operators like **AND** and **OR**, you can add significantly more conditions to be tested.

When statements have an inherent priority to them since they act like if, else if statements, so one can create priority encoders using the when statement as well. **The select statement does not have this property**. Here is an 8-3 line encoder.

```vhdl
outdata <= "111" when data(7) = '1' else
           "110" when data(6) = '1' else
           "101" when data(5) = '1' else
           "100" when data(4) = '1' else
           "011" when data(3) = '1' else
           "010" when data(2) = '1' else
           "001" when data(1) = '1' else
           "000" when data(0) = '1' else
           "000";
```

## Generate statement

The generate statement is a shortcut to duplicate combinational assignments. It works by printing the statements generated by the loop multiple times. A piece of hardware is inferred with every loop, hence why generate was chosen as the name

Syntax:

```
label:for i in range generate
    --statements
end generate;
```

Here are a few examples:

```
--reverse a vector
reverse: for i in a'range generate
        b(i) <= a(a'high-i);
end generate;
--xor LSB of a vector with MSB of another vector
gen: for i in x'range generate
    x(i) <= a(i) xor b(b'left-i);
end generate;
--3-8 line decoder
entity combogen is
    port(
        input: in std_logic_vector(2 downto 0);
        output: out std_logic_vector(7 downto 0)
    );
end combogen;

architecture rtl of combogen is
    signal counter: integer := 0;
begin
    gen: for i in 0 to 7 generate
        output(i) <= '1' when counter = i else '0';
        end generate;
    counter <= to_integer(unsigned(input));
end architecture;
```

## Functions and attributes

As you have have noticed in the previous code block, there are two concepts not covered. The 'left, 'high, 'range and the function to_integer.

The one with the ticks are called attributes. Attributes can access information about the properties of either a signal, type, or entity.

For example, 'range is short hand for specifying the range of a vector (0 to 4), (5 downto 3), whatever.

Here are a list of some noteworthy attributes in VHDL: T means type, A means array, S means scalar.

```
attribute ascending        : boolean; -- T'ASCENDING  is boolean true if range of T defined with to .
                                      -- A'ASCENDING  is boolean true if range of A defined with to .
attribute event            : boolean; -- S'EVENT      is true if signal S has had an event this simulation cycle.
attribute high             : integer; -- T'HIGH       is the highest value of type T.
                                      -- A'HIGH
attribute left             : integer; -- T'LEFT       is the leftmost value of type T. (Largest if downto)
                                      -- A'LEFT
attribute length           : integer; -- A'LENGTH     is the integer value of the number of elements in array A.
attribute low              : integer; -- T'LOW        is the lowest value of type T.
                                      -- A'LOW
attribute range            : string;  -- A'RANGE      is the range  A'LEFT to A'RIGHT  or  A'LEFT downto A'RIGHT .
attribute reverse_range    : string;  -- A'REVERSE_RANGE  is the range of A with to and downto reversed.
attribute stable           : string;  -- S'STABLE     is true if no event is occurring on signal S.
                                      -- S'STABLE(t)  is true if no even has occurred on signal S for t time.
```

The function to_integer is defined in the package numeric_std. It takes an signed or unsigned binary input and converts it to its respective integer representation. This is why we see input being casted to an unsigned by doing *unsigned(input)*.

If you want to convert an integer to unsigned binary and then to a std_logic_vector, you would do:

*std_logic_vector(to_unsigned(int,n))*, where n is the bit length of the vector.

There are a bunch of useful functions in the various packages for you to explore.

## Sequential Logic: The process

A process in VHDL is an extremely difficult concept to understand for someone coming from a programming background. The typical syntax of a process is:

```
architecture
begin
    process(sensitivity_list)
        --declarative region
    begin
        --statements
    end process;
end architecture;
```

If you research online, you may hear that the process is a region in which you can do sequential statements. This is misleading. While it is true you can do sequential constructs, signal assignments are still concurrent. You CANNOT do a loop 5 times and expect a integer signal to go up by 5 in a single process reevaluation.

Every time a signal in the sensitivity list experiences a change, or an event, the process in reevaluated. Typically when we do sequential statements, the process in only sensitive to the clock. But we never want to imply a latch, so we use a nifty function called *rising_edge()* in order for the language to imply an edge-triggered flip-flop rather than a latch. Here is an example of a D Flip-Flop:

```
architecture rtl of combogen is
    signal clk,d,q,qb : std_logic;
begin
    process(clk)
    begin
        if rising_edge(clk) then
            q  <= d;
            qb <= not d;
        end if;
    end process;
end architecture;
```

Why not have qb be not q? Well, the way assignment happens with processes is that at the very end of the processes, the signal assignments are applied all at the same time. Meaning that even though q is now d, qb is going to be whatever q was last event, not what it is now. PULL UP THE DIGITAL SIMULATION AS A REFERENCE.

## The if statement

The if statement is a processed version of the when statement, though as of VHDL-2008 whens can be used in processes. But we don't care right now.

An example can of course be seen on the previous page. Syntax:

```
if condition1 = 4 or condition2 /= "hello" then
    --statements
elsif condition3 < 27 and condition3 /= "meow" then
    ---statements
else
    report "I love VHDL so much :)))";
end if;
```

## The case statement

The case statement is analogous to the the select statement in conditional space. However, case statements are capable of nesting if statements within them. An example of this can be seen in the skeleton code. The generic syntax is:

```
case signal is
    when (state) =>
        -- statements
    when (state2) | (state3) => -- either state2 or state3
        if condition then --thing
        else -- thing
        end if;
    when others =>
        null --do nothing
end case;
```

Case statements are the best choice for doing state machines due to their flexible control structure.

## The loop statement

The loop statement is practically identical to the generate statement (it does not require a label), however it gains two keywords for better control flow over the loop. **exit** and **next**.

During loops, using the exit statement, of course, exits the loop. This is useful if you want to make a gigantic if elsif construct without having to write an absurd amount of lines. Next is mainly used to make patterns, such as excluding even numbers during a loop.

```vhdl
entity priorityencode is
    port(
        a1: in std_logic_vector(7 downto 0);
        b1: out std_logic_vector(2 downto 0)
    );
end priorityencode;

architecture rtl of priorityencode is
    function priority_encode(input: std_logic_vector) return std_logic_vector is
        variable result: std_logic_vector(integer(ceil(log2(real(input'length)))) - 1 downto 0) := (others => '0');
    begin
        for i in input'range loop
            if input(i) = '1' then
                result := std_logic_vector(to_unsigned(i,result'length));
                exit;
            end if;
        end loop;
        return result;
    end function priority_encode;

begin
    b1 <= priority_encode(a1);
end architecture;
```

```vhdl
 process(clk, reset)
    begin
        if reset = '1' then
            reg <= (others => '0');
        elsif rising_edge(clk) then
            loadt <= load;
            rbtnt <= rbtn;
            lbtnt <= lbtn;
            if load = '1' and loadt = '0' then
                reg <= swts;
            elsif lbtn = '1' and lbtnt = '0' then
                reg <= reg(6 downto 0) & '0';
            elsif rbtn = '1' and rbtnt = '0' then
                for i in 0 to 7 loop --EXAMPLE OF NEXT STATEMENT!
                    if i mod 2 = 0 then --switch 0 to 1 if you want odds
                        next; --If the index is divisible by 2, skip the loop.
                    end if;
                    reg(i) <= '0';
                end loop;
            end if;
        end if;
end process;
```

## Counters

Here is an example of a 4 bit counter with down up capabilities:

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

entity counter is
    port(
        clk : in  std_logic;
        du  : in  std_logic;
        q   : out std_logic_vector(3 downto 0)
    );
end counter;

architecture rtl of counter is
    signal count : unsigned(3 downto 0) := (others => '0');
begin
    process(clk)
    begin
        if rising_edge(clk) then
            if du = '1' then
                count <= count - 1;
            elsif du = '0' then
                count <= count + 1;
            end if;
        end if;
    end process;
    q <= std_logic_vector(count);
end architecture;
```

Another way of doing it:

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

entity counter is
    port(
        clk: in std_logic;
        du: in std_logic;
        q: out std_logic_vector(3 downto 0)
    );
end counter;

architecture rtl of counter is
    signal count: std_logic_vector(3 downto 0) := "0000";
begin
    process(clk)
    begin
        if rising_edge(clk) then
            if du = '0' then
                count <= std_logic_vector(unsigned(count) + 1);
            elsif du = '1' then
                count <= std_logic_vector(unsigned(count) - 1);
            end if;
        end if;
    end process;
    q <= count;
end architecture;
```

## Shift Registers

Shift registers can be made via the concatenation operator &.

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

entity SIPO is
    port(
        clk: in std_logic;
        din: in std_logic;
        rw: in std_logic;
        dout: out std_logic_vector(7 downto 0)
    );
end SIPO;

architecture rtl of SIPO is
    signal datastore: std_logic_vector(7 downto 0) := (others => '0');
begin
    process(clk)
    begin
        if rising_edge(clk) and rw = '0' then
            datastore <= datastore(6 downto 0) & din;
        end if;
    end process;

    dout <= datastore when rw = '1' else (others => 'Z');
end architecture;
```

## Enumerated Types

One of the core prerequisites to making a state machine is using custom types.

VHDL is extremely versatile when it comes to types. This is because of its basis in the Ada programming language.

Enumerated types can be created with the following syntax.

```vhdl
architecture rtl of SIPO is
    type enumerated_type is (first, second, third, fourth, fifth);
    signal thing : enumerated_type := second;
begin

end architecture;
```

Enumerated types are primarily used in state machines rather than binary. The reason is that the synthesis tools will optimize for the flip flop allocation. You can also manually control it, and that will be explained later. For now let us continue to examples of state machines.

## State Machines

State machines are just clocked processes with a case and conditional branching. An example of a state machine can be seen in lab 6. Here are some other random state machines.

A different version of lab 6 for FPGAs:

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

entity mymachine is
    port(
        clk: in std_logic;
        areset: in std_logic;
        sreset: in std_logic;
        change: in std_logic;
        sseg: out std_logic_vector(6 downto 0)
    );
end mymachine;

architecture rtl of mymachine is
    type stateType is (B, A, D, E); --creates a new type
    signal next_state, state: stateType := B; --assigns signals to type
begin
    process(clk,areset)
    begin
        if areset = '1' then
            state <= B;
        elsif rising_edge(clk) then
            if sreset = '1' and areset = '0' then
                state <= A;
            else
                state <= next_state;
            end if;
        end if;
    end process;

    process(state,change)
    begin
        case state is
            when B =>
                if change = '1' then
                    next_state <= A;
                else
                    next_state <= E;
                end if;
            when A =>
                next_state <= D;
            when D =>
                if change = '1' then
                    next_state <= E;
                else
                    next_state <= B;
                end if;
            when E =>
                if change = '1' then
                    next_state <= B;
                else
                    next_state <= A;
                end if;
        end case;
    end process;

    sseg <= "1100000" when state = B else
            "0110000" when state = E else
            "0001000" when state = A else
            "1000010" when state = D else "0000000";
end architecture;
```

As can be seen, the machine is split up into two processes. One for the clock and the other for selecting the next state. In FPGAs this is the most ideal method for describing state machines.

Manual encoding:

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

entity async is
    port(
        clk: in std_logic;
        areset: in std_logic;
        input: in std_logic;
        output: out std_logic
    );
end async;

architecture rtl of async is
    constant A: std_logic_vector (1 downto 0) := "00";
    constant B: std_logic_vector (1 downto 0) := "01";
    constant C: std_logic_vector (1 downto 0) := "10";
    constant D: std_logic_vector (1 downto 0) := "11";
    signal state: std_logic_vector (1 downto 0);
    signal next_state: std_logic_vector (1 downto 0);
begin
    clockBehavior: process(clk,areset)
    begin
        if areset = '1' then
            state <= A;
        elsif rising_edge(clk) then
            state <= next_state;
        else
            state <= state;
        end if;
    end process clockBehavior;

    stateSwitching: process(state, input)
    begin
        case state is
            when A =>
                if (input = '1') then
                    next_state <= B;
                elsif (input = '0') then
                     next_state <= A;
                end if;
            when B =>
                if (input = '1') then
                    next_state <= C;
                elsif (input = '0') then
                    next_state <= B;
                end if;
            when C =>
                if (input = '1') then
                    next_state <= D;
                elsif (input = '0') then
                    next_state <= C;
                end if;
            when D =>
                if (input = '1') then
                    next_state <= A;
                elsif (input = '0') then
                    next_state <= D;
                end if;
            when others =>
                next_state <= A;
        end case;
    end process stateSwitching;
    output <= '1' when state = D else '0';
end architecture;
```

State machine theory is extensive. The best I have seen is in chapter 15 of the book:

*Circuit Design with VHDL 3rd ed.* by Volnei A. Pedroni

## Synthesis Attributes

Creating a custom attribute for synthesis is easy. The syntax is:

```
attribute attribute_name : attribute_type;
attribute attribute_name of entity_name : entity_class is expression;
```

Each synthesis tool has a wide bredth of synthesis attributes to work with. For the most part all we care about are

```
attribute LOC : string;
attribute syn_encoding : string;
```

LOC is declared in the entity and the string value is ”P##” where the numbers represent the pin numbers. For example ”P01” would be pin 1.

syn_encoding can be one of the following:

- sequential
- gray
- onehot

This attribute tells the synthesizer what to do with the registers when an enumerated state type is used.

Example:

```
signal state : state_type := First;
attribute syn_encoding of state : signal is "onehot";
```

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.math_real.all;
use work.UART2.all;

entity BASYS_UART_RX is
    port(
        clock         : in  std_logic;
        serial_input  : in  std_logic;
        stop_bit      : out std_logic;
        data_byte     : out std_logic_vector(UART_BITS-1 downto 0)
    );
end entity;

architecture rtl of BASYS_UART_RX is
    type UART_RX_SM is (RX_Idle, RX_Start, RX_Data, RX_Stop, RX_Cleanup);
    signal rx_state       : UART_RX_SM := RX_Idle;
    signal clock_count    : integer range 0 to UART_CLKB-1 := 0;
    signal bit_index      : integer range 0 to UART_BITS-1 := 0;
    signal register_byte  : std_logic_vector(UART_BITS-1 downto 0) := (others => '0');
    signal wire_stop_bit  : std_logic := '0';
begin
    statemachine:process(clock)
    begin
        if rising_edge(clock) then
            case rx_state is
                when RX_Idle =>
                    wire_stop_bit <= '0';
                    clock_count   <=  0;
                    bit_index     <=  0;

                    if serial_input = '1' then
                        rx_state <= RX_Start;
                    else
                        rx_state <= RX_Idle;
                    end if;
                when RX_Start =>
                    if clock_count = (UART_CLKB-1) / 2 then
                        if serial_input = '0' then
                            clock_count <= 0;
                            rx_state    <= RX_Data;
                        else
                            rx_state    <= RX_Start;
                        end if;
                    else
                        clock_count <= clock_count + 1;
                        rx_state    <= RX_Start;
                    end if;
                when RX_Data =>
                    if clock_count < UART_CLKB - 1 then
                        clock_count <= clock_count + 1;
                        rx_state    <= RX_Data;
                    else
                        clock_count              <= 0;
                        register_byte(bit_index) <= serial_input;
                        if bit_index < UART_BITS - 1 then
                            bit_index <= bit_index + 1;
                            rx_state  <= RX_Data;
                        else
                            bit_index <= 0;
                            rx_state  <= RX_Stop;
                        end if;
                    end if;
                when RX_Stop =>
                    if clock_count < UART_CLKB - 1 then
                        clock_count <= clock_count + 1;
                        rx_state    <= RX_Stop;
                    else
                        wire_stop_bit <= '1';
                        clock_count <= 0;
                        rx_state    <= RX_Cleanup;
                    end if;
                when RX_Cleanup =>
                    rx_state      <= RX_Idle;
                    wire_stop_bit <= '0';
                when others =>
                    rx_state <= RX_Idle;
            end case;
        end if;
    end process;

    stop_bit  <= wire_stop_bit;
    data_byte <= register_byte;
end architecture;
```

## Verification

A testbench is code that applies time stimulus to a design, observes the response, and verifies it with expected behavior. It is basically a software version of a logic analyzer.

Verification is the most important process in digital design. There are substantially more verification engineers than design engineers, they made exorbitant amounts of money, and verification is around 80% of effort in the design process.

There are several ways to write a testbench. We shall explore them as they ramp up in complexity.

## Extremely basic testbench

Given a full adder design from page 7, we can write a simple test bench to stimulate the inputs and then report the outputs to the terminal:

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

entity fa_tb is
end entity;

architecture test of fa_tb is
    signal ain,bin,cin,summation,cout : std_logic := '0';
    constant DELAY : time := 10 ns;
    constant CHECK_DELAY : time := 5 ns;
begin
    fa_inst: entity work.fa
    port map(
        a     => ain,
        b     => bin,
        c     => cin,
        sum   => summation,
        carry => cout
    );

    process
    begin
        ain <= '0';
        bin <= '0';
        cin <= '0';
        wait for CHECK_DELAY;
        report "Sum: " & to_string(summation);
        report "Carry: " & to_string(cout);
        wait for DELAY;
        -- etc etc etc

        wait; --makes process wait forever to end testbench
    end process;
end architecture;
```

Very nice. As can be seen the testbench stimulates the inputs and then reports the inputs.

Just like processes in synthesis the signals do not get assigned until the process suspends. A process with a sensitivity list is like having a "wait on " statement at the end of a process.

The wait statement is when it suspends. That is why we have a CHECK_DELAY in order to accurately report the Sum and Carry values.

## Data structures for Testbenches

Writing out repetitive code for a testbench can be annoying. Thankfully using a **record** data structure type, we can automate this process with a loop.

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

entity fa_tb is
end entity;

architecture test of fa_tb is
    type pattern_type is record
        a, b, c : std_logic;
        o, cot  : std_logic;
    end record;
    type pattern_array is array (natural range <>) of pattern_type;
    signal ain,bin,cin,summation,cout : std_logic := '0';
    constant DELAY : time := 10 ns;
    constant CHECK_DELAY : time := 5 ns;

    constant PATTERNS: pattern_array :=
            (('0', '0', '0', '0', '0'),
             ('0', '0', '1', '1', '0'),
             ('0', '1', '0', '1', '0'),
             ('0', '1', '1', '0', '1'),
             ('1', '0', '0', '1', '0'),
             ('1', '0', '1', '0', '1'),
             ('1', '1', '0', '0', '1'),
             ('1', '1', '1', '1', '1'));
begin
    fa_inst: entity work.fa
    port map(
        a     => ain,
        b     => bin,
        c     => cin,
        sum   => summation,
        carry => cout
    );

    process
    begin
        for i in PATTERNS'range loop
            ain <= PATTERNS(i).a;
            bin <= PATTERNS(i).b;
            cin <= PATTERNS(i).c;
            wait for CHECK_DELAY;
            assert summation = patterns(i).o
                report "Mismatch on sum"
            severity error;
            assert cout = patterns(i).cot
                report "Mismatch on carry"
            severity error;
            wait for DELAY;
        end loop;
        report "End of test!";
        wait; --makes process wait forever to end testbench
    end process;
end architecture;
```

The keyword assert makes it so that if the summation signal does not equal the value of our test vector, it reports an error at the timestamp. If nothing gets reported, everything worked fine!