Introduction to VHDL

Antoine Lavault[1,2]

[1]Apeira Technologies

[2]UMR CNRS 9912 STMS, IRCAM, Sorbonne Université

January 19, 2024

Contents

Contents

What is VHDL

- VHDL = Very-High Speed Integrated Circuit High-Level Description Language. Not Verilog HDL.
- IEEE-approved standard for hardware description language
- High-level description language for both simulation and synthesis

Some definitions

Some terms we will use during the presentation :

- HDL - Hardware Description Language : a programming language used to describe a piece of hardware (duh)

To model a piece of hardware in HDL, two ways :

- Behavior modeling : A component is described by its input/output response
- Structural modeling : a component is described by interconnecting different components or primitives

Some more definitions

- Register Transfer Level (RTL): a type of behavioural modeling for synthesis
- Synthesis : transposing the HDL description into an circuit, then optimizing the circuit.
- Process : a basic unit of execution in VHDL

Behavior modeling

- Only the functionality of the circuit is described, not the structure
- No specific hardware intent

Example

Think of a bit left shifter.
Input : $i_1, \cdots i_n$, output : $o_1, \cdots o_n$ If activated: for ii from n to 1: shift(i) := shift(i-1)
output <=shift

Structural Modeling

- Describe the functionality and the structure of the circuit
- Call out the specific hardware

RTL Synthesis

TODO

Typical development flow

Two step process with feedback paths : synthesis and simulation.
This is basically Test Driven Development

Things about VHDL

- Two types of constructs :
  - Simulation only
  - Synthesis and simulation
- VHDL language is (mostly) not case-sensitive
- A statement in VHDL ends with a ";" (semi-colon)
- VHDL is white-space insensitive (i.e indentation doesn't matter expect for the reader)
- Comments are added with "–" (two dashes)

All in all

- VHDL is a description language for complex digital circuits
- It is portable to any programmable digital platform
- High-level behavioural description from the specifications

Contents

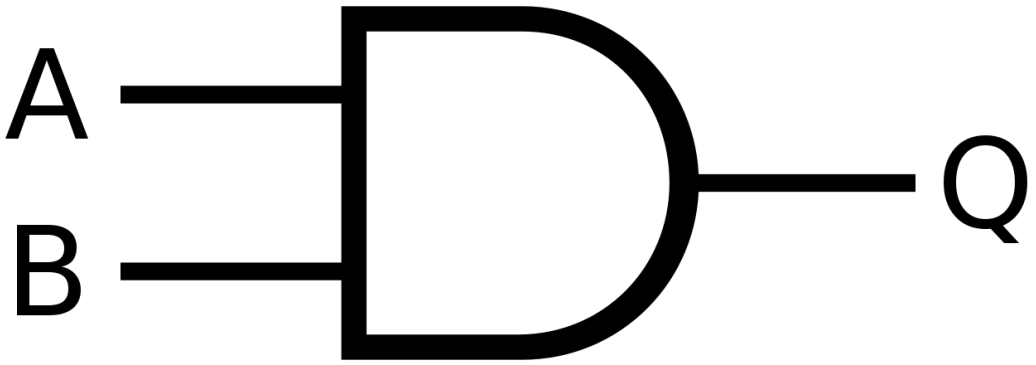

Figure: And Gate

Way. Too. Hard.

Inside and Out

The AND gate can be described by what is "outside" and what is "inside":
On the outside, we have:

- two inputs: A and B
- one output: Q

On the inside, we have some magic powder made of...
Just kidding. It's an AND gate: $Q = AB$

VHDL, at last

```
-- VHDL Code for AND gate
-- Header file declaration
library IEEE;
use IEEE.std_logic_1164.all;
-- Entity declaration
entity andGate is

    port(A : in STD_LOGIC;       -- AND gate input
         B : in STD_LOGIC;       -- AND gate input
         Q : out STD_LOGIC);     -- AND gate output
end andGate;
-- Architecture definition
architecture andLogic of andGate is
 begin
    Q <= A AND B;
end andLogic;
```

Library

- We have libraries in VHDL, and a library is made of packages.
- In the example above, the library is `IEEE`, and we use the package `std_logic_1164` (and especially, we import all of its content).
- To use the package, the keyword `use` is used, and `.all` shows we want to load all of the library's content.
- Another useful library : `STD_NUMERIC`

Entity, or the outside

The syntax for the entity is shown below:

```
-- Entity declaration
entity entityName is
    port(list of ports);
end entityName;
```

A mode and a type define a port. The 4 modes available are:

- in: for inputs
- out: for outputs
- inout: for bidirectional signals
- buffer: for outputs made to be fed back

As for the types, they are in general, taken from the `IEEE 1164` package:

- `STD_LOGIC` for a scalar i.e. a bit
- `STD_LOGIC_VECTOR` for a vector

Architecture, or the inside

- High-level portable description
  - Behavioral: data-flow, sequential process
  - Hierarchical: use of VHDL blocks and their connections
- Low-level hardware-specific description
  - Uses proprietary hardware-specific primitives
  - Can be used to force to use of some part of the logic blocks on an FPGA.

The syntax for the entity is shown below:

```
-- Architecture declaration
architecture architectureName of entityName is
    -- internal signals and constants
    -- internal functions
begin
    -- dataflow description
    -- sequential description
end architectureName;
```

Signals

Definition

A signal is an internal physical link between two elements.
Inputs and outputs are external signals.

You should think of a signal as something you can look at with an oscilloscope.
A signal can be instantiated between the keywords `Architecture` and `begin`:

```
signal signalName : signalType := initialValue;
```

Remark

All signals are strongly typed, like in C or Java!
This means conversion operations will happen often, between SIGNED and UNSIGNED especially.

Types in VHDL - Scalars

- `bit`: 0 or 1. Not really used in practice.
- `boolean`: True or False. Reference type for conditional structures
- `STD_LOGIC`: 9 values:
  - 3 of them have a physical sense: 0,1 and Z
  - 6 others are used for simulation
    - U for uninitialized
    - X for an unknown result
    - L for a signal that is probably 0
    - H for a signal that is probably 1
    - W for a signal non-quantifiable by 0 or 1
    - - for "Do Not Care"

Types in VHDL - Vectors

A vector can be thought of as an array of scalars.

- `STD_LOGIC_VECTOR`: is a vector of elements of type `STD_LOGIC`

```
a: STD_LOGIC_VECTOR(3 downto 0);
b: STD_LOGIC_VECTOR(1 to 4);
```

- For VHDL vectors, the leftmost bit is the MSB. In this example, $a(3)$ and $b(1)$ are the MSB and $a(0)$ and $b(4)$ are the LSB.
- `SIGNED` and `UNSIGNED`: behave like `STD_LOGIC_VECTOR` but have arithemetic operations defined in `NUMERIC_STD`. Conversion functions can be found in the `NUMERIC_STD` package.

Types in VHDL - Other useful types

- Mathematical types :
  - `INTEGER`: 32 bit integer. Conversion functions exist to go from `INTEGER` to `SIGNED` or `UNSIGNED`. It can also be set to a certain range :

```
a: integer range 0 to 20; -- an integer in [0,20]
```

  - `NATURAL`: integers greater or equal to 0.
  - `POSITIVE`! integers greater or equal to 1
  - `REAL`: float number, `IEEE-754` standard.
- Characters and Strings
  - Very useful during simulation!!!
  - `CHAR`: a character
  - `STRING`: a string of characters.

Syntax example

- Signals:

```
architecture Behavioural of syntaxExample is
    signal A: STD_LOGIC;
    signal B,C,D,E,F,G: STD_LOGIC;
    signal carry: STD_LOGIC := '1';
    signal HCounter: STD_LOGIC_VECTOR(9 downto 0);
    signal VCounter: STD_LOGIC_VECTOR(9 downto 0):=(others=>'0');
    signal op1, op2: STD_LOGIC_VECTOR(3 downto 0):="0000";
    signal seg7: STD_LOGIC_VECTOR(1 to 7);
    signal X,Y: INTEGER range 0 to 1023;
begin
    ...
```

- Constants :

```
architecture Behavioural of syntaxExample is
    signal seg7: STD_LOGIC_VECTOR(1 to 7);
    constant number_1: STD_LOGIC_VECTOR(1to7):="0110000";
    constant number_2: STD_LOGIC_VECTOR(1to7):="1101101";
begin
    ...
```

Operators in VHDL

- Assignment : `signal_name <= expression;`

If defined for the type in use, these operators also exist:

- Logic operators: and, or, not, nand, xor.
- Arithmetic operators (INTEGER, SIGNED, UNSIGNED): +,-,*,/,mod (modulo), rem(remainder).
- Concatenation: for `STD_LOGIC`,`STD_LOGIC_VECTOR`, CHAR and STRING.
- Comparison: <,>,<=,>=,=,/=

Contents

Example

We want to build a component that takes two inputs, *a* and *b*, and transforms them through an AND gate that is attached to a D flip-flop with clock-enable and asynchronous reset.
We found two interesting components in a vendor-specific package :

- AND2 : two-inputs AND gate
- FDCE : D flip-flop with clock-enable and asynchronous reset

Let's import the package containing these components.

```
library vendor;
use vendor.specific.all;
```

Entity description

We want to describe the outside of our component as such:

- *a* and *b* are inputs, *c* is an input
- A clock input with a clock enable pin
- An reset signal

A valid description in VHDL would be:

```
entity myComponent is
    port(a: in std_logic;
         b: in std_logic;
         c: out std_logic;
         clear: in std_logic;
         clk: in std_logic;
         clkEn: in std_logic);
end myComponent;
```

Internal signal

First, let's draw the circuit. If it hasn't been done before.
A clear place to put a signal is between the AND2 and the FDCE.

```
architecture behavioural of myComponent is
    signal D_internal: std_logic;
```

To use the AND2 and FDCE components, we add the following to the architecture before `begin`.

```
architecture behavioural of myComponent is
    signal D_internal: std_logic;
    component AND2
        port(i0: in std_logic;
        i1: in std_logic;
        o: out std_logic);
    end component;
    component FDCE
        port(...)
    end component;
```

Using the library components, for real.

We only described the component to our architecture, to use them effectively, we write:

```
begin
    CMP1: AND2
        port map(i0=>a;
        i1=>b;
        o=>D_internal);
    CMP2: FDCE
        port map(
        C=>clk;
        CE=>clkEn;
        CLR=>clr;
        D=>D_internal;
        Q=>c);
end behavioural;
```

Contents

Concurrency

Between `begin` and `end` lies the concurrent domain.
This means operations written in this domain are executed in parallel when possible.
There are three main components in the concurrent domain :

- Dataflow elements
  - Simple assignments
  - Conditional assignments : `when ... else ...`
  - Selected signal assignments : `with ... select ...`
- Sequential processes

An example - AND gate

This is how you write a `when ... else ...`:

```
output <= '1' when in1='1' and in2='1'
    else '0' when in1='1' and in2='0'
    else '0' when in1='0' and in2='1'
    else '0' when in1='0' and in2='0'
    else '0';
```

Why `else 0;` at the end? Keep in mind that `STD_LOGIC` elements have 9 values!

An example - AND gate

The following is also valid:

```
output <= '1' when in1='1' and in2='1' else '0';
```

and this one as well:

```
signal inputs: STD_LOGIC_VECTOR(1 to 2);
begin
    inputs <= in1 & in2;
    output <= '1' when inputs = "11" else '0';
end architecture;
```

Going further-ALU

Let's design an ALU with 8-bit inputs/output and two operations ADD and SUBSTRACT.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL
use IEEE.NUMERIC_STD;
entity alu is
    port(ina; std_logic_vector(7 downto 0);
        inb: std_logic_vector(7 downto 0);
        operation: std_logic;
        output: std_logic_vector(7 downto 0));
end alu;
architecture behavioural of alu is
begin
    output <= STD_LOGIC_VECTOR(SIGNED(ina) + SIGNED(inb))
        when operation = '0'
    else STD_LOGIC_VECTOR(SIGNED(ina) - SIGNED(inb))
        when operation = '1'
    else "00000000";
end behavioural;
```

Selective assignement

Back to the AND gate example...

```
signal inputs: STD_LOGIC_VECTOR(1 to 2);
begin
    inputs <= in1 & in2;
    output <= '1' when inputs = "11" else '0';
end architecture;
```

can be transformed into:

```
signal inputs: STD_LOGIC_VECTOR(1 to 2);
begin
    with inputs select
        sortie <= '1' when '11',
                  '0' when others;
end architecture;
```

Warning

Always put `when others` to handle cases not described.

With/select statement-ALU

The ALU example with the "with/select" statement would be:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL
use IEEE.NUMERIC_STD;
entity alu is
    port(ina; std_logic_vector(7 downto 0);
        inb: std_logic_vector(7 downto 0);
        operation: std_logic;
        output: std_logic_vector(7 downto 0));
end alu;
architecture behavioural of alu is
begin
    with operation select
        output<=STD_LOGIC_VECTOR(SIGNED(ina) + SIGNED(inb))
                    when operation = '0'
                STD_LOGIC_VECTOR(SIGNED(ina) - SIGNED(inb))
                    when operation = '1'
                (others=>0) when others; -- returns an adequatly sized vector
end behavioural;
```

Testing ?

Well, we better have a way to test these snippets of code...
Later. Too late to write a lecture.

Contents

Sequential domain

We have seen not so long ago that the concurrent domain allows the execution of sequential processes. In this domain, we will be able to :

- use C-like structures
- describe combinational logic blocks in a different way
- easily handle synchronous phenomenon i.e. clock ticks.
- test-benches

How to do that? Describe a process:

```
processName : process(sensitivity_list)
```

- Sensitivity list: list of signals whose change of state launches the process
- No sensitivity? Remove the parenthesis executed unconditionally.
- Naming the process is not necessary.

Process syntax

```
processName : process(sensitivity_list)
    -- declare internal variables
    -- declare constants
begin
    -- sequential process <-> line order matters
end processName;
```

Variables and signals

- The syntax is awfully similar...
- But they have totally different meanings:
  - A signal physically exists in the component (unless some optimization are applied at synthesis time)
  - A variable only exists in the process (limited scope) and is just a helper when writing the equations in the process
- And different assignation behaviors:
  - In a process, a signal is assigned with <=, and the assignation will be effective only when the process is exited !
  - In a process, a variable is assigned with := and it's instantaneous.

Programming structure

- Simple assignations : <= for signals, := for variables
- Conditional/selective assignments : ok in VHDL-2008. In general, not the default version of VHDL.
- Conditional structures `if.then.elsif.else.endif;`

Case statement

```
case var is
    when val => action1; -- action if var = val
    when vald to valf => action2; -- if var is between vald and valf
    when val1|val2|valn => action3; -- if var is val1, val2 or valn
    when others => action4; -- deal with the rest
end case;
```

Loops

For loop:

```
for index in start_value to end_value loop
-- index etc... are integers or subtype of integers
    -- actions to do in the loop
    -- etc...
end loop;
```

While loop:

```
while condition loop
    -- actions to do in the loop
end loop;
```

Ensure the condition becomes false at some point...

Contents

Testing a VHDL module

Let's take the AND gate we have seen some pages ago.
We can test it by:

- Playing with a simulator
- Writing tests in VHDL $\rightarrow$ exhaustive and with much more controls.

Test-bench as a component

A test-bench can be described as a VHDL component with:

- No inputs or outputs (a sealed box)
- Internal signals to test the unit under test
- a declaration of the unit under test
- After `begin`, instantiate the unit under test
- Test signals (generated within the test-bench)

Let's write this !

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL
-- No inputs or outputs
entity andGate_tb is
end andGate_tb;
architecture behavioural of andGate_tb is
    -- internal signals, used as stimuli
    signal A,B,Q: STD_LOGIC;
    -- uut declaration
    component andGate is
    port(A : in STD_LOGIC;
         B : in STD_LOGIC;
         Q : out STD_LOGIC);
    end component;
begin
---
```

Let's write this ! Cont.

```
begin
--- instantiate the unit under test
    uut: andGate port map(
    A=>A,
    B=>B,
    Q=>Q);
    A_stimuli: process
    begin
        A <= '0'; wait for 5ns;
        A <= '1'; wait for 5ns;
    end process A_stimuli;
    B_stimuli: process
    begin
        B <= '0'; wait for 10ns;
        B <= '1';
    end process B_stimuli;
end behavioural;
```

Contents

Type conversion

- A `STD_LOGIC_VECTOR` is a vector of `STD_LOGIC`. And that's it. Is it an integer, a float, a banana? Can I add it, can I multiply it, can I peel it?
- This is why we use `SIGNED` and `UNSIGNED`. The underlying data is still `STD_LOGIC` but we now know how to handle it.
- Conversion function: `TO_SIGNED`, `TO_UNSIGNED`, `TO_INTEGER` (in `NUMERIC_STD`)
- Operations : they are defined in `NUMERIC_STD` !

Signal Attributes

- An attribute in VHDL is a meta property that's attached to a type or object.
- It's useful to know a signal behaves → clock edges
- Some are used in simulation only e.g `last_event`

We also define what is a transaction:

- A signal is assigned a value
- The assignment does not necessarily change the value.

Signal attributes

- Bound to the signal declaration:
  - later
- Bound to the signal evolution
  - `event` if the signal got its value changed when entering a process
  - `event` if the signal got a transaction during the current process
  - `last_event`(simulation only) elapsed time since last event
  - `last_value` value before the last event
  - `last_active`(simulation only) elapsed time since last transaction

Usage: find a rising edge on the clock signal:

```
clk'event' and clk'last_value'=0 and clk=1
```

There are also the functions `rising_edge` and `falling_edge`from the 1164 package:

```
rising_edge(clk)
falling_edge(clk)
```