INTRODUCTION TO FPGA PROGRAMMING

Lesson 06: VHDL Simulation

Dr. Davide Cieri[1]

[1]Max-Planck-Institut für Physik, Munich

August 2024

MAX-PLANCK-INSTITUT FÜR PHYSIK

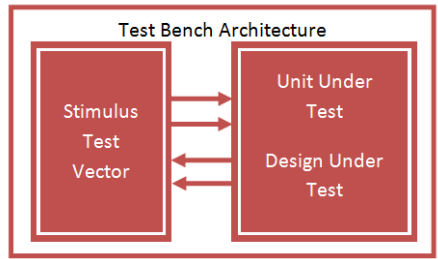
VHDL SIMULATION

- In the past labs, you already interfaced with some VHDL simulations
- Simulation is the process of testing a VHDL module to ensure it behaves as expected
- Simulation is fundamental to test your code, before actually implementing it on an FPGA
  - Once the FPGA is programmed, you can only change the input signals and check the outputs
  - No control on what is happening inside the FPGA[1]
- Simulation or Verification is a fundamental part of the FPGA design workflow
  - Large companies have separated teams, only dedicated to the verification of RTL designs.

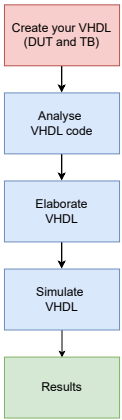
[1] FPGA vendors offer special modules to "spy" inside the FPGA, more in another lecture.

VHDL TESTBENCHES

- A testbench is a VHDL code that applies stimuli to the design/unit under test (DUT/UUT) and checks the responses.
- It is not synthesized into hardware but used solely for simulation purposes.
- Helps automate the testing process.



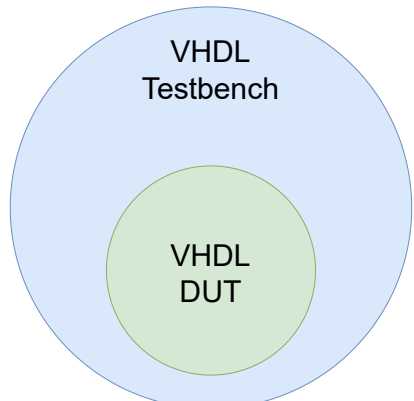
VHDL SIMULATION FLOW



1. *Write your code.* Write your VHDL design and test-bench using your favorite text editor.
2. *Analyse.* Compiles the code to find syntax errors, using a VHDL simulator.
3. *Elaborate.* Advances the simulation time to 0. It can be merged with the simulation stage
4. *Simulate.* Run the test bench for a specific period or until no further activity.
5. *Results.* Check the results of your simulation with a waveform viewer

BASIC TESTBENCH STRUCTURE

- Purely behavioural code (no ports)
- Instantiate the RTL design
- Optionally, defines clocks
- Stimulate the design
- Optionally, check the responses from the design
  - Pass/Fail reports, timeouts



EXAMPLE: A SIMPLE TESTBENCH

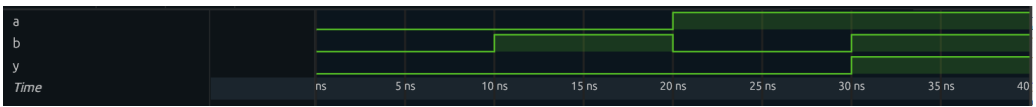
```vhdl
entity and_gate is
port(
  A : in std_logic;
  B : in std_logic;
  Y : out std_logic
);
end and_gate;

-- AND Gate Architecture
architecture behavior of and_gate is
begin
    Y <= A and B;
end behavior;
```

```vhdl
entity tb_and_gate is                    -- Testbench Entity (empty)
end tb_and_gate;
architecture test of tb_and_gate is      -- Testbench Architecture
    signal A, B, Y : std_logic;
begin
    UUT: entity work.and_gate port map(A => A, B => B, Y => Y);
    stim_proc: process                   -- Stimulus process
    begin
      A <= '0'; B <= '0';                -- Test case 1
      wait for 10 ns;
      A <= '0'; B <= '1';                -- Test case 2
      wait for 10 ns;
      A <= '1'; B <= '0';                -- Test case 3
      wait for 10 ns;
      A <= '1'; B <= '1';                -- Test case 4
      wait for 10 ns;
      wait;
    end process;
end test;
```

EXAMPLE: WAVEFORM

- Simulation Waveforms show changes to signal values as function of simulation time
  - You can show both internal and interface signals



DISCRETE EVENT SIMULATION

- VHDL simulation is discrete
  - Multiple of the time resolution of the simulation software
  - Simulation advances from event to event
  - Every time a signal changes, it is an event
- Event scheduling
  - VHDL is a parallel programming language, but CPUs work sequentially
  - Simulators schedules transactions to signals, when seeing an assignment in the code
  - Transactions are updated in the next *delta cycle*
    - A delta cycle is a zero-time timestamp

DELTA CYCLE EXAMPLE

```
process
begin
  A <= '1';          -- Event 1: A
      updated to '1'
  wait for 10 ns;    -- Wait 10 ns
  B <= A;            -- Event 2: B
      updated to '1'
  C <= B;            -- Event 3: C
      remains '0' initially, will
      update in the next delta cycle
  wait;
end process;
```

- **Initial State:** A = '0', B = '0', C = '0'
- **Delta Cycle 1:** A updated to '1'
- **Delta Cycle 2:** B updated to '1' (due to the wait for 10 ns)
- **Delta Cycle 3:** C updated to '1' (immediately after B updates)

SIMULATING THE CLOCK SIGNAL

- Very simple to define in your testbench
- It can be free-running (loops indefinitely) or running for a finite number of clock cycles
- Make uses of constants for better readability

```
architecture behavior of testbench is
  signal clk : std_logic := '0';
begin
  clk <= not clk after 5 ns;
end behavior;
```

```
architecture behavior of testbench is
  signal clk : std_logic := '0';
  constant CLK_PERIOD : time := 10 ns;
begin
  clk <= not clk after CLK_PERIOD/2;
end behavior;
```

```
architecture behavior of testbench is
signal clk : std_logic := '0';
constant NCYCLES : integer := 100;
begin
  clk_proc : process
  begin
    for I in 0 to NCYCLES-1 loop
      clk <= not clk;
      wait for 5 ns;
      clk <= not clk;
      wait for 5 ns;
    end loop;
    wait;
  end process;
end behavior;
```

WAIT STATEMENTS

- Test benches use delayes to sequence inputs with `wait` statements (not synthetisable)
- `wait for <time>`.
  - E.g. `wait for 10 ns;`
- `wait on <signal>`. Waiting for an event (change of state in a signal).
  - E.g. `wait on clk`.
- `wait until <boolean expression>`. Wait for a specific signal value.
  - E.g. `wait until clk = '1';`

SYNCHRONOUS RESET SIGNAL

- To avoid potential race condition, set the rst signal always after the rising edge of the clock in the testbench (one delta cycle later)

```
-- Example
stimulus: process
begin
  wait until rising_edge(clk);
  rst <= '1';
  wait until rising_edge(clk);
  wait until rising_edge(clk);
  rst <= '0';
end process;
```

STIMULATING DATA

- In a simulated environment, the precise timing of signals is ideal and deterministic.
- In actual hardware, there are propagation delays, clock skew, and other non-ideal factors that make timing more variable.
- Applying data changes at the clock edge might work in simulation but fail in real hardware due to these variabilities.
  - Avoid applying data on the active clock edge

```
stimulus: process
begin
  -- Avoid this
  wait until rising_edge(clk);
  data <= "010";
  -- Better specifying a delay
  wait until rising_edge(clk);
  data <= "111" after 1 ns;
  -- or applying on the inactive edge
  wait until falling_edge(clk);
  data <= "110";
  wait;
end process;
```

PROCEDURES

- A procedure is a subprogram that performs a specific task.
- It can contain multiple sequential statements.
- Procedures can have input, output, and inout parameters.
  - Allowed parameters are constants, variables, signals or files.

```
procedure <procedure name> <optional parameter list> is
  <declarations>; -- variable/constants/types local to the procedure
begin
  <statements>;   -- Sequential statements. Explicit return statement stops the
      procedure
end procedure;
```

PROCEDURE EXAMPLES

Example without parameters

```
architecture behavior of testbench is
  signal clk : std_logic := '0';
  constant NCYCLES : integer := 100;

  procedure run_clk(signal clk : out std_logic) is
    constant CLK_PERIOD : time := 10 ns;
  begin
    clk <= not clk;
    wait for CLK_PERIOD/2;
    clk <= not clk;
    wait for CLK_PERIOD/2;
  end procedure;
begin
  process
  begin
  for I in 0 to NCYCLES-1 loop
    run_clk(clk);
  end loop;
  wait;
end process;
end behavior;
```

```
architecture behavior of testbench is
  signal a : integer := 0;

  procedure write_data (value : in integer; data :
       out integer ) is
  begin
    data <= value;
    wait for 10 ns;
  end procedure;
begin
  process
  begin
    write_data(5, a);
    write_data(10, a);
    write_data(12, a);
  wait;
end process;
end behavior;
```

FUNCTION

- A `function` is a group of statements for computing a result of a certain data type
- It can have only input parameters (allowed are constants, signals or files).
- It must return a `return` statement
- An `impure` function can modify and read external signals not in the parameter list

```
[pure/impure] function <name> [ <parameters> ]
return <type> is
  <declarations>;
begin
  <sequential statement>;
return some_value; -- of type <type?
end function;
```

FUNCTION EXAMPLE

```
-- Calculate the number of clock cycles in minutes/seconds
function CounterVal(Minutes : integer := 0;
                    Seconds : integer := 0) return integer is
    variable TotalSeconds : integer;
begin
    TotalSeconds := Seconds + Minutes * 60;
    return TotalSeconds * ClockFrequencyHz -1;
end function;
```

PRINTING, WRITING AND ASSERTING

- Checking the waveform is a direct way to debug your code. However:
  - Not easy to determine functional correctness from waveforms
  - Sampling a lot of signals in waveforms increase simulation runtime
  - It's a manual work. Prone to human errors
- For more complex designs, it is preferable to automatise the testbench
- This can achieved by printing, writing and asserting signals
- VHDL provides standard way thanks to the `std.textio.all` package

PRINTING SIGNAL VALUES IN VHDL

- Printing information to the console in VHDL is done with the `report` function

```
-- Syntax
report <message_string> [severity <severity_level>];
-- Example
report "this is a message;
report "this is a serious message" severity warning;
```

- Possible severity levels are: `note` (default), `warning`, `error`, `failure`
- To report the value of a signal that is not a string use the `'image` attribute
- Strings can be concatenated using the `&` operator

```
-- Syntax
<type>'image(<signal_name>)
-- Example
report "unexpected value. i = " & integer'image(i);
```

WRITING TO A FILE

- Often is more convenient to write into a file
- Done with the `writeline` and `write` functions

```
-- procedure WRITE(L : inout LINE; VALUE : in integer; JUSTIFIED: in SIDE := right; FIELD: in WIDTH := 0);
use std.textio.all;
...
process
  file fp : text;
  variable lp : line;
begin
  file_open(fp, "filename.txt", write_mode);
  write(lp, "a string"); -- write a string into line lp
  writeline(fp, lp); -- write the line into the file fp
end process;
```

- `write` gets also `string, boolean, real, time` in input
- `owrite, hwrite, swrite, bwrite` are aliases to write octal, hexadecimal, string and binary values

READING FROM A FILE

- In a similar way, you can read from a file, maybe to read some data to be injected into your DUT

```
-- procedure READ(L:inout LINE; VALUE: out integer);
use std.textio.all;
...
process
  file fp : text;
  variable lp : line;
  variable my_int : integer;
begin
  file_open(fp, "filename.txt", read_mode);
  readline(fp,lp);   -- read the file into a line
  read(lp, my_int);  -- read the line and assign the value to my_int
  dut_int <= my_int; -- Assign the dut_int signal to my_int
end process;
```

ASSERTING

- In VHDL you can use the `assert` function, to check signal values against some expectation
- Assertions can be concurrent or sequential
  - Concurrent defined in entities or architectures, continuously monitor the DUT
  - Sequentials are activated ony when reaching the statement
- Assert returns always a boolean value. Default severity is `error`
- The values to assert can be extracted from a reference file

```
-- Syntax
[<label>:] assert <condition to check> [report <message>] [severity <level>]
-- Example
assert a = '0' report "a is not 0" severity failure;
```

TERMINATING A SIMULATION

- Report statements with `failure` severity stop the simulation
- Stopping all stimuli, `wait;` at the end of the process
- Using the `stop` and `finish` procedures
  - `stop` doesn't actually finish the simulation, but it pauses it, and gets back to the Tcl shell (kind of a breakpoint.)

```
use std.env.all; -- Include this package for the stop/finish procedures
...
report "This is the end, my only friend, the end." severity failure;
wait;
finish(<status>); -- 0: print nothing, 1: print simulation time and location, 2: print
    simtime, location and statistics
stop(<status>); -- Same as finish
```

TIMEOUT

- It is always a good idea to put a simulation timeout process, to avoid unexpected infinite runs

```
timeout_proc : process
begin
  wait for 10 ms; -- stops after 10ms
  report "Reached the timeout of the simulation!" severity failure;
end process;
```

VHDL SIMULATORS

- In this course, we are using two VHDL simulators
  - Vivado Simulator (Xsim). Integrated in Vivado. Manual here
  - GHDL: Open-source simulator https://ghdl.github.io/ghdl/
    - GHDL does not provide a waveform viewer, but can save the output into a waveform file format (`.vcd`, `.ghw`)
    - GTKWave is an open-source wave viewer that can be used in combination with GHDL

BASIC SIMULATION WITH XSIM AND GHDL

Vivado XSIM Example workflow

```
# Analyse the required simulation files
    with VHDL2008 (default 93)
xvhdl --2008 dut.vhd
xvhdl --2008 tb.vhd
# Elaborate the design (tb is the name
    od the testbench module to run)
xelab tb -s my_sim --debug typical
# Running the simulation in the GUI
xsim my_sim -gui
# Running the simulation in batch mode
xsim my_sim -R
```

GHDL Example workflow

```
# Analyse the required simulation files
    with VHDL2008 (default 93)
ghdl -a --std=08 dut.vhd
ghdl -a --std=08 tb.vhd
# Elaborate the design (tb is the name
    od the testbench module to run)
ghdl -e tb
# Running the simulation (tb is the name
     od the testbench module to run)
ghdl -r --std=08 tb --wave=mywave.ghw
# Optionally, open the waveform with
    gtkwave
gtkwave mywave.ghw &
```

LAB 10: TESTBENCH CODING

The figures in these slides are taken from:

- Digital Design: Principles and Practices, Fourth Edition, John F. Wakerly, ISBN 0-13- 186389-4.

- allaboutfpga.com
- nandland.com
- docs.amd.com
- https://www.symmetryelectronics.com/
- https://www.edn.com/