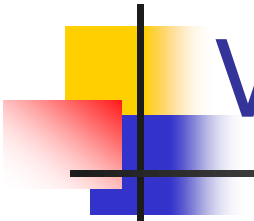
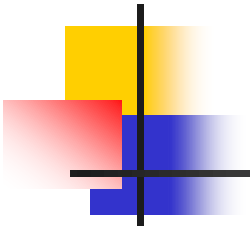


VHDL Testbenches

- Testing is important
  - Pentium FDIV bug
  - More test engineers than design engineers
- Lecture objective: writing VHDL testbenches to apply test vectors

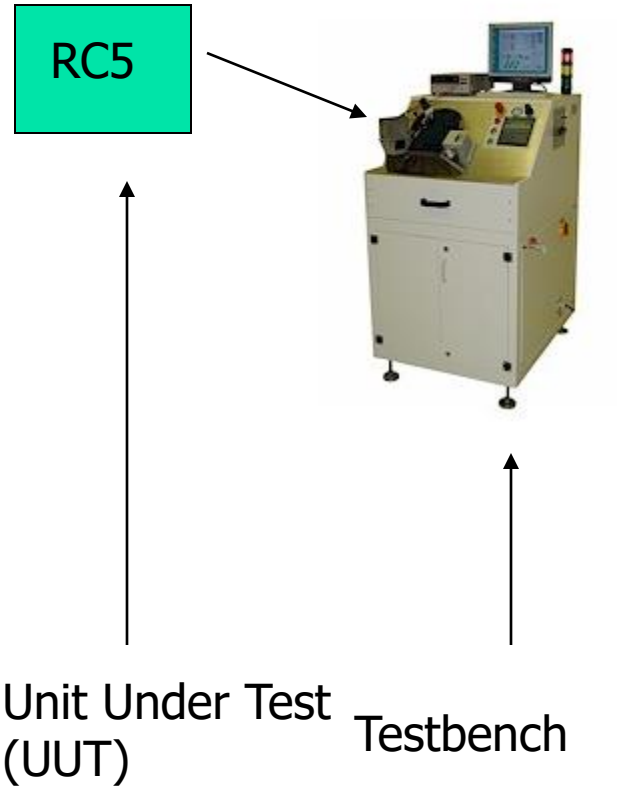


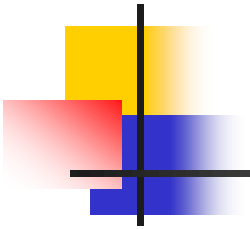
Introduction to Testbenches

- As logic designs become complex, comprehensive, up front verification becomes critical to the success of the design project.
- When simulation is used right at the start of the project, you will have a much easier time with synthesis, and you will spend far less time re-running time-intensive processes, such as FPGA place-and-route tools and other synthesis related software.

Introduction to Testbenches

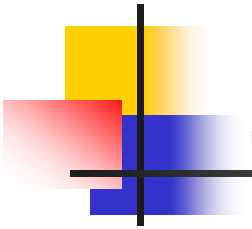
- To simulate your project, you will need to develop an additional VHDL program called a test bench.
- Testbenches emulate a hardware breadboard into which you will "install" your synthesizable design description for the purpose of verification.
- A Testbench can be thought of as a "Virtual Tester" into which you plug your design for verification.





Introduction to Testbenches

- Can be simple – applying a sequence of inputs to the circuit over time.
- Can be complex – reading test data from a disk file and writing test results to a screen and to a report file.



Successful Testbench

- A successful Testbench should be able to automatically:
  - Read input stimulus (test vectors) from an input file and apply it to the unit under test.
  - Compare the outputs from the unit under test with a file that contains the expected results.
  - Display to the user all errors so that they can be fixed.
- We will be using the TextIO package along with assert statements to accomplish this function.

assert statement

- VHDL's **assert** statement provides a quick and easy way to check expected values and display messages from your testbench.
- An **assert** statement has the following general format:

  assert *condition_expression*

  report *test_string*

  severity *severity_level*
- When analyzed, the condition expression is evaluated. As in an **if** statement, the condition expression of an **assert** statement must evaluate to a boolean value (**true** or **false**).
  - If the condition expression is **false** (assertion failed), the text that you have specified in the optional **report** statement is displayed in your simulators transcript window. The **severity** statement clause then indicates to the simulator what action (if any) should be taken in response to the assertion failure.
  - The severity level can be specified using one of the following predefined severity levels: **NOTE, WARNING, ERROR,** or **FAILURE.** The actions that result from the use of these severity levels will depend on the simulator you are using. (Modelsim has options that you can specify on what actions to take for each of these severity levels)

Draw the system diagram for this description

```
library ieee;
use ieee.std_logic_1164.all;
entity testnand is
end testnand
architecture do_it of testnand is
 component nand
   port(A,B : in std_logic;  y  : out std_logic);
 end component
signal A,B, Y: std_logic;
begin
UUT: nand port map (A=>A, B=>B, Y=>Y);

Process
…
end process;
end do_it;
```

A Simple Testbench

```
library ieee;
use ieee.std_logic_1164.all;
entity testnand is
end testnand
architecture do_it of testnand is
 component nand
   port(A,B : in std_logic;  y  : out std_logic);
 end component
signal A,B, Y: std_logic;
begin
UUT: nandgate port map (A=>A, B=>B,
Y=>Y);
```

```
Process
  constant PERIOD: time:= 40ns;
Begin
  A<='1'; B<='1';
 wait for PERIOD;
 assert (Y='0')  report "Test failed!" severity
ERROR;
  A <= '1';  B <= '0';
  wait for PERIOD;
   assert (Y='1')  report "Test failed!" severity
ERROR;
  A <= '0';  B <= '1';
  wait for PERIOD;
    assert (Y='1')  report "Test failed!" severity
ERROR;
  A <= '0';  B <= '0';
  wait for PERIOD;
    assert (Y='1')  report "Test failed!" severity
ERROR;
 wait;
end process;
end do_it;
```

A Simple Testbench

Reading from the top of the testbench we see:

Library and **Use** statements making standard logic package available for use.

Entity declaration for testbench. Note: testbenches do not include an interface (port) list; they are highest-level design unit when simulated.

Architecture declaration:

A component declaration corresponding to unit under test.

Signal declarations for **A, B, Y.** These signals will be used to

(1) apply inputs to the unit under test, and

(2) observe behavior or output during simulation.

A component instantiation statement and corresponding port map statement that associates top-level signals **A, B,** and **Y** with their equivalent ports in the lower-level entity. Component name used (**UUT**) is not significant; any valid component name could have been chosen.

A **process** statement describing inputs to the circuit over time. This process has been written without a sensitivity list.

uses **wait** statements to provide a specific amount of delay (defined using constant **PERIOD**) between each new combination of inputs.

uses **Assert** statements to verify that ckt is operating correctly for each combination of inputs.

Uses wait statement w/o any condition expression to suspend simulation indefinitely after desired inputs have been applied. (In the absence of the final **wait** statement, the process would repeat forever, or for as long as the simulator has been instructed to run.)

Mux: Testbench (example 1)

(adapted from http://esd.cs.ucr.edu/labs/tutorial/tb_mux.vhd)

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
entity Mux_TB is                                -- empty entity
end Mux_TB;
architecture TB of Mux_TB is
    -- initialize declared signals
    signal Test_I3: std_logic_vector(1 downto 0):="00";
    signal Test_I2: std_logic_vector(1 downto 0):="00";
    signal Test_I1: std_logic_vector(1 downto 0):="00";
    signal Test_I0: std_logic_vector(1 downto 0):="00";
    signal Test_O: std_logic_vector(1 downto 0);
    signal Test_S: std_logic_vector(1 downto 0);
component Mux
    port(      I3: in std_logic_vector(1 downto 0);
               I2: in std_logic_vector(1 downto 0);
               I1: in std_logic_vector(1 downto 0);
               I0: in std_logic_vector(1 downto 0);
               S:  in std_logic_vector(1 downto 0);
               O:  out std_logic_vector(1 downto 0));
    end component;
begin
  TestMux: Mux port map (I3=>Test_I3, I2=>Test_I2, I1=>Test_I1, I0=>Test_I0, S=>Test_S, O=>Test_O);
```

Mux: Testbench (example 1)

(adapted from http://esd.cs.ucr.edu/labs/tutorial/tb_mux.vhd)

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity Mux_TB is                                -- empty entity
end Mux_TB;

architecture TB of Mux_TB is
    -- initialize declared signals
    signal Test_I3: std_logic_vector(1 downto 0):="00";
    signal Test_I2: std_logic_vector(1 downto 0):="00";
    signal Test_I1: std_logic_vector(1 downto 0):="00";
    signal Test_I0: std_logic_vector(1 downto 0):="00";
    signal Test_O: std_logic_vector(1 downto 0);
    signal Test_S: std_logic_vector(1 downto 0);
```

Mux: Testbench

```
component Mux
    port( I3: in std_logic_vector(1 downto 0);
          I2: in std_logic_vector(1 downto 0);
          I1: in std_logic_vector(1 downto 0);
          I0: in std_logic_vector(1 downto 0);
          S:  in std_logic_vector(1 downto 0);
          O:  out std_logic_vector(1 downto 0));
    end component;
begin
    TestMux: Mux port map (I3=>Test_I3, I2=>Test_I2, I1=>Test_I1,
I0=>Test_I0, S=>Test_S, O=>Test_O);
```

Mux: Testbench

```
process
begin
        Test_I3 <= "01";
        Test_I2 <= "10";
        Test_I1 <= “00";
        Test_I0 <= "11";
        -- case select equal "00"
        wait for 10 ns;
        Test_S <= "00";
        wait for 1 ns;
        assert (Test_O="11") report "Error Case 0" severity error;
        -- case select equal "01"
        wait for 10 ns;
        Test_S <= "01";
        wait for 1 ns;
        assert (Test_O=“00") report "Error Case 1" severity error;
```

Mux: Testbench

```
        -- case select equal "10"
        wait for 10 ns;
        Test_S <= "10";
        wait for 1 ns;
        assert (Test_O="10") report "Error Case 2" severity error;
        -- case select equal "10"
        wait for 10 ns;
        T_S <= "11";
        wait for 1 ns;
        assert (T_O="01") report "Error Case 3" severity error;
        wait;
    end process;
end TB;
```

Reading and Writing files with Text I/O

- One of the predefined packages supplied with VHDL is the Textual Input and Output (TextIO) package.
- TextIO package contains procedures and functions that give the designer the ability to read from and write to formatted text files (ASCII files of any format).
- TextIO treats these ASCII files as files of lines, where a line is a string, terminated by a carriage return. There are procedures to read a line and write a line and a function that checks for end of file.
- Type **line** is declared in the TextIO package and is used to hold a line to write to a file or a line that has just been read from the file. The line structure is the basic unit upon which all TextIO operations are performed.
- For instance, when reading from a file, the first step is to read in a line from the file into a structure of type **line**. Line structure is processed field by field.
- The opposite is true when writing to a file. First, the line structure is built field by field in a temporary line data structure, then the line is written to the file.

Use of textio package

Typical textio functions

```
file cmdfile: TEXT;  -- Define the file 'handle'
FILE_OPEN(cmdfile,"TST_ADD.DAT",READ_MODE);

variable L: Line;        -- Define the line buffer
variable good: boolean; --status of the read operation

readline(cmdfile,L);           -- Read the line
read(L,CI,good);  --Read the CI argument from the line
-- good is a boolean value that will be false on read error
```

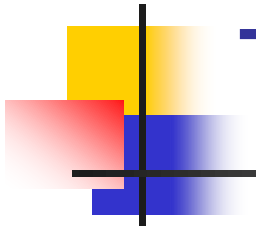
Reading and Writing files with Text I/O

Simple example of TextIO behavior:

- Example shows how to read a single integer value from a line, square the value, and write the squared value to another file.
- Process loops until an end of file condition occurs on the input file **infile**.
- **READLINE** statement reads a line from the file and places the line in the variable **my_line**.
- Next executable line has a **READ** procedure call that reads a single integer value from **my_line** into variable **int_val**.
- Variable is squared and written to another variable of type **line**, called **out_line**.
- Last TextIO procedure call is the **WRITELINE**. This procedure writes out the line variable **out_line** to the output file **outfile.**

```
LIBRARY IEEE;
USE STD.TEXTIO.ALL;
USE IEEE.STD_LOGIC_TEXTIO.ALL;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY square IS
  PORT( go : IN std_logic);
END square;

ARCHITECTURE simple OF square IS
BEGIN
  PROCESS(go)
    FILE infile : TEXT IS IN "/doug/test/example1";
    FILE outfile : TEXT IS OUT "/doug/test/outfile1";

    VARIABLE out_line, my_line : LINE;
    VARIABLE int_val : INTEGER;
  BEGIN
    WHILE NOT( ENDFILE(infile)) LOOP
      -- read a line from the input file
      READLINE( infile, my_line);
      -- read a value from the line
      READ( my_line, int_val);
      -- square the value
      int_val := int_val **2;
      -- write the squared value to the line
      WRITE( out_line, int_val);
      -- write the line to the output file
      WRITELINE( outfile, out_line);
    END LOOP;
  END PROCESS;
END simple
```

Testbench - adder



- The circuit we will be testing is a 32-bit adder subtractor unit.
- Testbench we want to write will read information from a file, in the form of hexadecimal values, and will include inputs and expected outputs for the circuit.
- We will use hread procedure. Hread accepts same arguments as the standard read procedure, but allows values to be expressed in hexadecimal format.
- The test file called tst_add.dat is shown below:

```
0 00000001 00000001 00000002 0
0 00000002 00000002 00000004 0
0 00000004 00000004 00000008 0
1 00000001 00000001 00000000 0
```

- The file format is CI A B SUM CO, with A, B and SUM expressed as hex values.
- Ideally this test file would be very large and be generated by a C++ program.

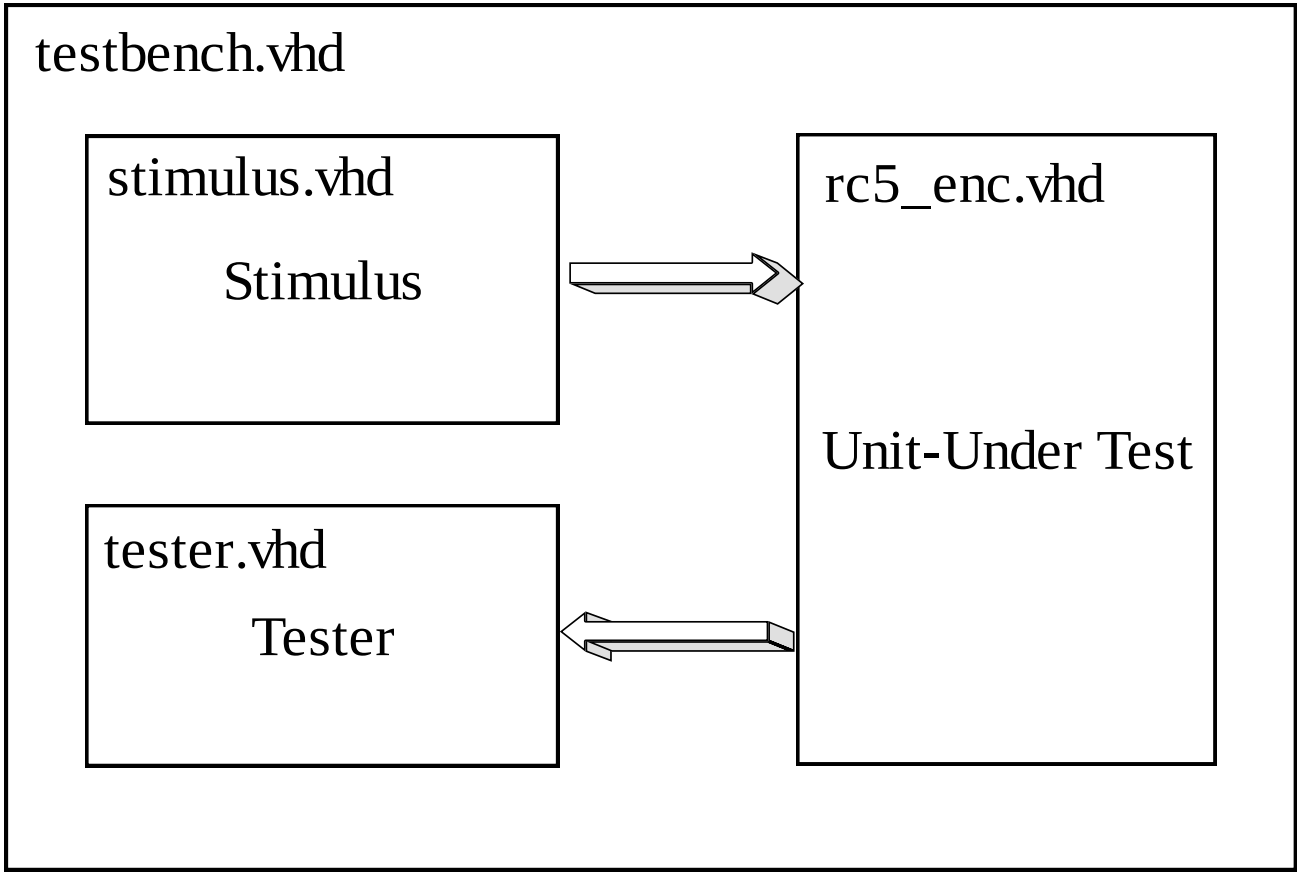
Complete Testbench - Adder

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

library textutil;
use textutil.std_logic_textio.all;

ENTITY tst_add IS
END tst_add;

ARCHITECTURE do_it OF tst_add IS
  COMPONENT adder32 PORT (cin : IN std_logic;
                          a, b : IN std_logic_VECTOR(31 DOWNTO 0);
                          sum : OUT std_logic_VECTOR(31 DOWNTO 0);
                          cout : OUT std_logic);
  END COMPONENT;

  SIGNAL Clk: std_logic;
  SIGNAL x, y : std_logic_vector(31 DOWNTO 0);
  SIGNAL sum : std_logic_vector(31 DOWNTO 0);
  SIGNAL cin, cout : std_logic;

  constant PERIOD: time := 200 NS;

BEGIN
  UUT : adder32 PORT MAP (cin, x, y, sum, cout);

  -- generate the clock...
  clk_gen: process
      begin
         loop
            Clk <= '0';
            wait for PERIOD / 2;
            Clk <= '1';
            wait for PERIOD / 2;
         end loop;
      end process;
```

```
readcmd: process

      -- This process loops through a file and reads one line
      -- at a time, parsing the line to get the values and
      -- expected result.

      file cmdfile: TEXT;       -- Define the file 'handle'
      variable L: Line;         -- Define the line buffer
      variable good: boolean; --status of the read operation

      variable CI, CO: std_logic;
     variable A,B: std_logic_vector(31 downto 0);
      variable S: std_logic_vector(31 downto 0);

   begin

      -- Open the command file...

      FILE_OPEN(cmdfile,"TST_ADD.DAT",READ_MODE);

      loop

         if endfile(cmdfile) then  -- Check EOF
            assert false
               report "End of file encountered; exiting."
               severity NOTE;
            exit;
         end if;

         readline(cmdfile,L);           -- Read the line
         next when L'length = 0;  -- Skip empty lines
       read(L,CI,good);     -- Read the A argument as hex value
         assert good
            report "Text I/O read error"
            severity ERROR;
```

```
         hread(L,A,good);     -- Read the A argument as hex value
         assert good
            report "Text I/O read error"
            severity ERROR;

         hread(L,B,good);     -- Read the B argument
         assert good
            report "Text I/O read error"
            severity ERROR;

         hread(L,S,good);     -- Read the Sum expected resulted
         assert good
            report "Text I/O read error"
            severity ERROR;

       read(L,CO,good);     -- Read the CO expected resulted
         assert good
            report "Text I/O read error"
            severity ERROR;

         cin <= CI;
         x <= A;
         y <= B;

         wait for PERIOD;

         assert (sum = S)
            report "Check failed!"
               severity ERROR;

          assert (cout = CO)
            report "Check failed!"
               severity ERROR;
      end loop;

      wait;

   end process;

end do_it;
```

RC5 testbench



Testbench.vhd

- An empty entity used to instantiate all modules
  - no I/Os! –why ??
  - Like a testbench on which you put your module-under-test, oscilloscope, logic analyzer, etc.

```
-- testbench.vhd
LIBRARY IEEE;
USE       IEEE.STD_LOGIC_1164.ALL;
USE       IEEE.STD_LOGIC_TEXTIO.ALL;
USE       STD.TEXTIO.ALL;

ENTITY testbench IS          -- No I/O signals defined
END testbench;

ARCHITECTURE tb_arch OF testbench IS
```

Testbench.vhd

```
   -- component declarations
      -- rc5_enc
      -- stimulus
      -- tester
   -- signal declarations, used to wire the components
      -- clr, clk, din, dout, etc
   CONSTANT   ClkPeriod    : Time:= 100 ns;
BEGIN
   -- component instantiations
            sti: stimulus PORT MAP(clr, clk, din);
            uut: rc5_enc PORT MAP(clr, clk, din, dout);
            tst: tester PORT MAP(dout);
END tb_arch;
```

Stimulus.vhd

- Applies input signals

```
-- stimulus.vhd
LIBRARY IEEE;
USE       IEEE.STD_LOGIC_1164.ALL;
USE       IEEE.STD_LOGIC_TEXTIO.ALL;
ENTITY stimulus IS
   PORT   (
      clr    : OUT     STD_LOGIC;
      clk    : OUT     STD_LOGIC;
      din    : OUT     STD_LOGIC_VECTOR(63 DOWNTO 0));
END stimulus;
ARCHITECTURE sti_arch OF stimulus IS
   SIGNAL       clk_tmp : STD_LOGIC;
   CONSTANT  ClkPeriod       : Time := 100 ns;
BEGIN
```

Apply clear signal

```
sti_clr: PROCESS
   BEGIN
      clr<='0';
      WAIT FOR 200 ns;
        clr<='1';
      WAIT;
   END PROCESS;
```

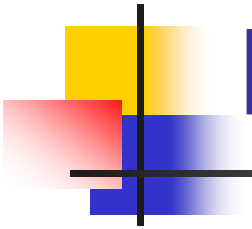
Generate Clock Signal

```
sti_clk: PROCESS
  BEGIN
     clk_tmp<='0';
     LOOP
        WAIT FOR (ClkPeriod/2);    clk_tmp<=NOT clk_tmp;
     END LOOP;
  END PROCESS;
  clk<=clk_tmp;
```

Apply Data Input signal

- Apply Data Input signal

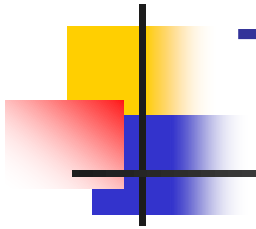
```
 sti_din: PROCESS
   BEGIN
      din<=X"FEDCBA9876543210";
      WAIT;
   END PROCESS;

END sti_arch;
```



How to apply input signals?

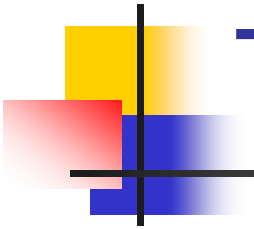
- Apply Other signals

```
start<='1', '0' AFTER 300 ns, '1' AFTER 350 ns, '0' AFTER 850 ns;
```



Tester.vhd

```
-- tester.vhd
LIBRARY IEEE;
USE       IEEE.STD_LOGIC_1164.ALL;
USE       IEEE.STD_LOGIC_TEXTIO.ALL;
USE       STD.TEXTIO.ALL;
ENTITY tester IS
   PORT(
            dout       : IN       STD_LOGIC_VECTOR(63 DOWNTO 0));
END tester;
```



Tester.vhd

```
ARCHITECTURE tst_arch OF tester IS
BEGIN
   chk_result: PROCESS
      VARIABLE tmp_dout: STD_LOGIC_VECTOR(63 DOWNTO 0);
      VARIABLE l: LINE;
      VARIABLE good_time, good_val, errordet: BOOLEAN;
      VARIABLE r : REAL;
      VARIABLE vector_time: TIME;
      VARIABLE space: CHARACTER;
```

Tester.vhd

```
      FILE vector_file: TEXT IS IN "values.txt";
BEGIN
   WHILE NOT ENDFILE(vector_file) LOOP
      READLINE(vector_file, l);
      READ(l, r, good_time);
      NEXT WHEN NOT good_time;
      vector_time:=r * 1 ns;
      IF(NOW < vector_time) THEN
         WAIT FOR vector_time-NOW;
      END IF;
      READ(l, space);  READ(l, tmp_dout, good_val);
      ASSERT good_val REPORT "Bad vector value";
      WAIT FOR 10 ns; -- Wait until the signal becomes stable
      ASSERT (tmp_dout=dout) REPORT "Output mismatch";
   END LOOP;
   WAIT;
   END PROCESS;
END tst_arch;
```

Usage of textio

- How to open a text file?

```
FILE vector_file: TEXT IS IN "values.txt";
```

- Typical textio functions

```
READLINE(vector_file, l); -- read a line from the text file
READ(l, r, good_time);    -- read the values
```

Test-vector file (values.txt)

Format:
Time value
Time value
Time value
……

1500 18C7C99A97F52AE0 (change it to binary value please)

What is inside a DO file?

```
cd {c:/ee4313/proj2}
vlib work
vcom rc5_enc.vhd -93                          -- enable VHDL'93 support
vsim rc5_enc
view wave
add wave clr clk din dout a a_reg b b_reg     -- you can view internal signals
force clr 0 0, 1 200                          -- Asynchronous reset
force clk 0 0, 1 50 –repeat 100               -- clock signal
force din 16#FEDCBA9876543210 0
run 1500
```

Functional simulation

- In ModelSim console window, type in following commands in sequence

```
cd {c:/ee4313/proj6}
vlib work
vcom rc5_enc.vhd stimulus.vhd -93
vcom tester.vhd testbench.vhd
vsim testbench
view wave
add wave clr clk din dout
run 1800
```

Timing Simulation

- In ModelSim console window, type in following commands in sequence

```
cd {c:/ee4313/proj6}
vlib work
vmap simprim c:/modeltech_5.6d/xilinx/simprim
vcom rc5_sim.vhd
vcom stimulus.vhd
vcom tester.vhd
vcom testbench.vhd
vsim -sdftyp /testbench/uut=c:/ee4313/proj6/rc5_sim.sdf work.testbench(tb_arch)
view wave
add wave clr clk din dout
run 1800
```

Real World – High Level Design Flow

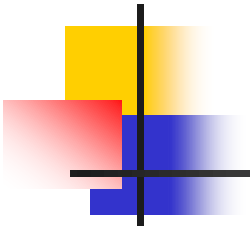
1. Design Specification –
   - Specifying the behavior expected of the final design.
   - Designer puts enough detail into the specification so that the design can be built.
   - Usually closely coordinated with software and other devices that communicate with the chip.
2. HDL Capture –
   - Designer enters the VHDL code for entities of the design and check them for correct syntax.
3. RTL Simulation –
   - Verify the correctness of the RTL VHDL description
   - VHDL simulator reads the VHDL description, compiles it into an internal format, and executes the compiled format using test vectors.
   - Designer can look at the output of the simulation and determine whether or not the design is initially working properly.
   - Designer then loads the VHDL into an automatic testbench that thoroughly applies stimulus and checks the results.

You are here →

   - Designer runs simulation for as long as needed to generate enough output data to determine if the design is correct. At the beginning of the design process, this may be only a few vectors to make sure the design resets properly. But later, more and more of the vectors are run as the design starts to function properly.

Real World – High Level Design Flow

4. RTL Synthesis –
   - The VHDL synthesis tools convert the VHDL description into a netlist in the target FPGA or ASIC technology.
   - The VHDL synthesis step is to create a design that implements the required functionality and matches the designer's constraints in speed, area, or power.
   - The synthesis tools generate a timing report and area report
     - Timing report – shows the timing of the critical paths of the design. The designer examines the timing of the critical paths closely because these paths ultimately determine how fast the design can run.
     - Area report – Shows the designer how much of the resources of the chip the design has consumed. The designer can tell if the design is too big for a particular chip and the designer needs to target a larger chip.



Real World – High Level Design Flow

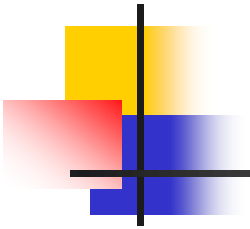
5. Functional Gate Simulation

- Some designer want to do a quick check on the output of the synthesis tool to make sure that the synthesis tool produced a design that is functionally correct.
- If proper design rules are followed for the input VHDL description, the synthesis tool should never generate an output that is functionally different from the RTL VHDL input, unless the tool has a bug.
- To do this, the designer reads the output VHDL netlist from the synthesis tool, plus a library of the synthesis primitives into the VHDL simulator and runs the simulation using the RTL verification vectors.
- For a completely functional simulation no timing is back annotated.

Real World – High Level Design Flow

6. Place and Route

- Place and route tools are used to take the design netlist and implement the design in the target technology device.
- The place and route tools place each primitive (gate, LUT, etc.) from the netlist into an appropriate location on the target device and then route signals between the primitives to connect the devices according to the netlist.
- Place and route tools are typically very architecture and device dependant. FPGA vendors provide these tools because the differences in architectures are large enough that writing a common tool for all architectures would be very difficult.
- Inputs to the place and route tools are the netlist (EDIF), timing constraints and placement constraints.
  - Timing constraints – Gives the place and route tools an indication of which signals have critical timing associated with them and to route these nets in the most timing efficient manner.
  - Placement constraints – Some place and route tools allow the designer to specify the placement of large parts of the design. This process is also known as floorplanning. Floorplanning allows the designer to pick locations on the chip for large blocks of the design so that routing wires are as short as possible.
- The other output from the place and route tools is a file used to generate the timing file. This file describes the actual timing of the programmed FPGA device of the final ASIC device. The most common format is SDF (standard delay format).



Real World – High Level Design Flow

7. Post Layout Timing Simulation
   - After the place and route process has completed, the designer will want to verify the results of the place and route process.
   - This simulation combines the netlist used for place and route with the timing file (SDF) from the place and route process into a simulation that checks both the functionality and timing of the design.
   - Porst route gate-level simulation, if done properly, also uses the same simulator as the RTL simulation. (Same testbench)
8. Static Timing
   - For designs of 10k-100k gates, post route timing simulation can be a good method of verifying design functionality and timing. However, as designs gets larger, or if the designer does not have test vectors, the designer can use static timing analysis to make sure the design meets timing requirements.
   - A static timing analyzer traces each path in the design and keeps track of the timing from a clock edge or an input. A timing report is then generated.

Real World – High Level Design Flow

9. On chip Debug –
   - This technique provides the designers with the ability to debug their design in the target system, at target speed, at the VHDL RTL level.
   - The VHDL for the device is read into a tool that automatically creates and inserts a small debug core into the device that probes internal signals.
   - The debug core is created based on information from the designer about what signals are to be probed.
   - The debug core communicates through the JTAG port on the device to an HDL debugger executing on a host platform.
   - The HDL debugger sends and receives data from the debug core and displays this data in context with an HDL for the design.
   - Waveforms of the internal device can also be displayed, providing the ability to trace down problems in the design.