VHDL Simulation: Test bench

- Introduction
- Test Bench Structures
- Modelsim-Altera simulation
- Advanced Test Bench

What is a VHDL Test Bench?

- VHDL Testbench is a piece of code, which purpose is to verify the functional correctness of HDL model.
- The main objectives are:
  - Instantiate the Design Under Test (DUT) or Unit Under Test (UUT)
  - Generate stimulus waveforms for it
  - Generate reference outputs and compare them with the outputs of DUT
  - Automatically provide a pass or fail indication
- So, the Test bench is a part of the circuits specification

Stimulus and response

- Test bench can generate stimulus:
  - Generating them "on the fly"
  - Reading vectors stored as constants in an array
  - Reading vectors stored in a separate system file
- Response is produced in the test bench.
- Response can be stored into file for further processing.
- Example:
  - Stimulus can be generated with Matlab and TB feeds it into DUT.
  - DUT generates the response and TB stores it into file.
  - Result can be compared to Matlab simulations.

- Introduction
- Test Bench Structures
- Modelsim-Altera simulation
- Advanced Test Bench

Test Bench characteristics

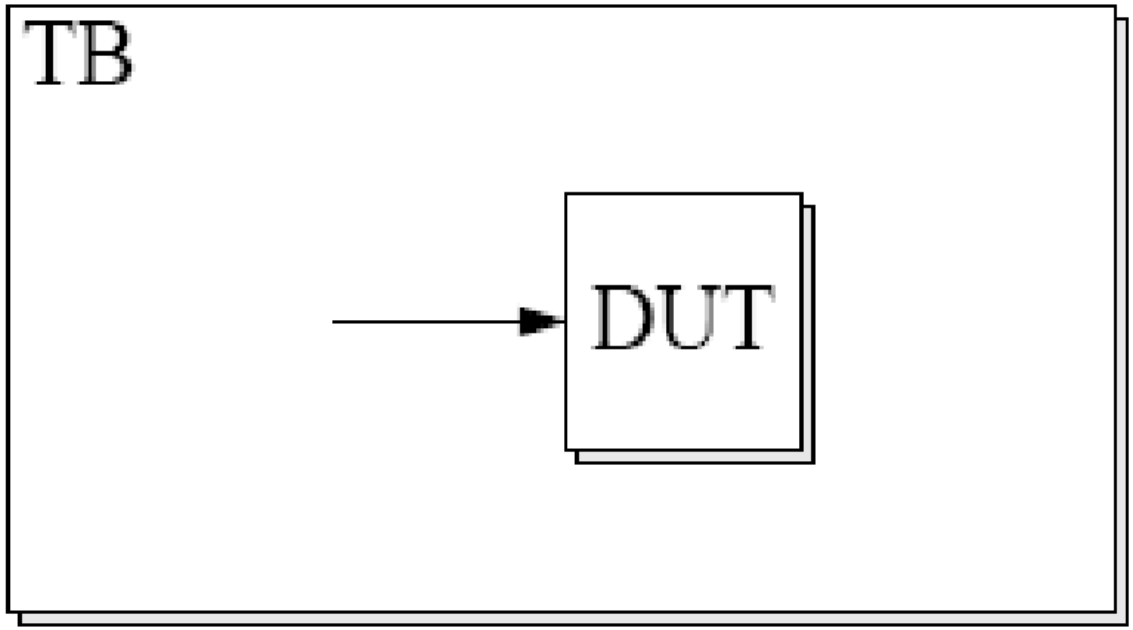
- Test bench should be reusable without difficult modifications.
- The structure of the TB should be simple enough so that other people understand its behaviour.
- Good test bench propagates all the generics and constants into DUT.

Question:

How to verify that the function of the test bench is correct?

Simple Test Bench

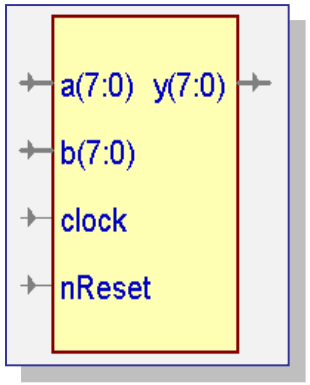
- Only DUT is instantiated into the testbench
- Stimulus is generated inside the testbench
- Poor reusability
- Suitable only for relatively simple designs



Simple Test Bench Example:

- DUT: Synchronous adder

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity adder_reg is
    port(
        clock   : in std_logic;
        nReset  : in std_logic;
        a, b    : in std_logic_vector(7 downto 0);
        y       : out std_logic_vector(7 downto 0)
        );
end adder_reg;

architecture bhv of adder_reg is
begin
    process (clock, nReset)
    begin
        if nReset='0' then
            y <= (others => '0');
        elsif clock'event and clock='1' then
            y <= a + b;
        end if;
    end process;
end bhv;
```



Simple Test Bench Example:

- TB for synchronous adder (template from Aldec's Active-HDL)

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

    -- Add your library and packages declaration here ...

entity adder_reg_tb is
end adder_reg_tb;

architecture TB_ARCHITECTURE of adder_reg_tb is
    -- Component declaration of the tested unit
    component adder_reg
    port(
        clock : in std_logic;
        nReset : in std_logic;
        a : in std_logic_vector(7 downto 0);
        b : in std_logic_vector(7 downto 0);
        y : out std_logic_vector(7 downto 0) );
    end component;

    -- Stimulus signals - signals mapped to the input
    signal clock : std_logic;
    signal nReset : std_logic;
    signal a : std_logic_vector(7 downto 0);
    signal b : std_logic_vector(7 downto 0);
    -- Observed signals - signals mapped to the output
    signal y : std_logic_vector(7 downto 0);

    -- Add your code here ...
```

```
begin
    -- Unit Under Test port map
    UUT : adder_reg
        port map (
            clock => clock,
            nReset => nReset,
            a => a,
            b => b,
            y => y
        );

    -- Add your stimulus here ...

end TB_ARCHITECTURE;

configuration TESTBENCH_FOR_adder_reg of adder_reg_tb is
    for TB_ARCHITECTURE
        for UUT : adder_reg
            use entity work.adder_reg(bhv);
        end for;
    end for;
end TESTBENCH_FOR_adder_reg;
```

Simple Test Bench Example:

Signal declaration

```
-- Add your code here ...
constant PERIOD : time := 50 ns;
signal clock : std_logic := '0';
```

Clock generation

```
-- Add your stimulus here ...
gen_clock : process (clock)
begin
    clock <= not clock after PERIOD/2;
end process;
```

Stimuli generation

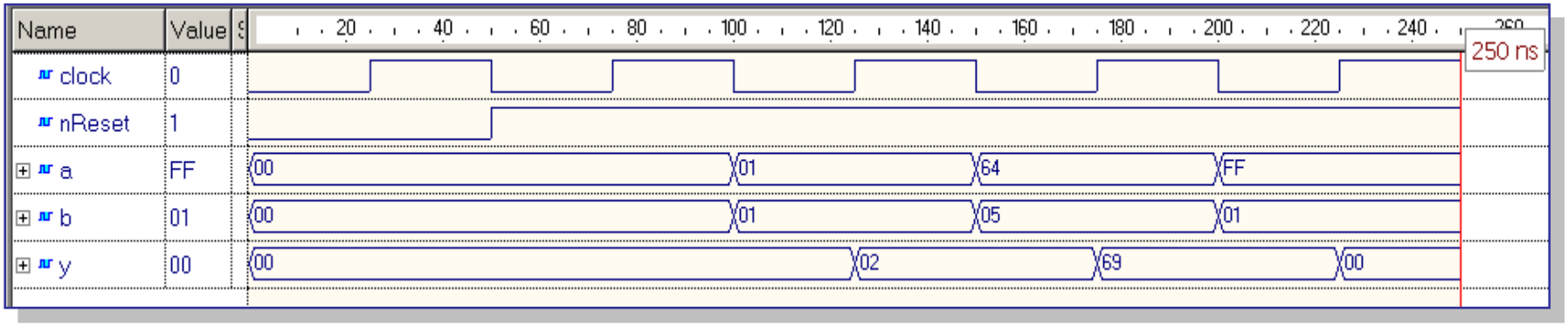
- Sequential (process):

```
process
begin
    nReset <= '0';
    wait for PERIOD;

    nReset <= '1';
    a    <=  "00000000";
    b    <=  "00000000";
    wait for PERIOD;

    a    <=  "00000001";
    b    <=  "00000001";
    wait for PERIOD;

    a    <=  "01100100";
    b    <=  "00000101";
    wait for PERIOD;

    a    <=  "11111111";
    b    <=  "00000001";
    wait for PERIOD;
end process;
```

- Concurrent:

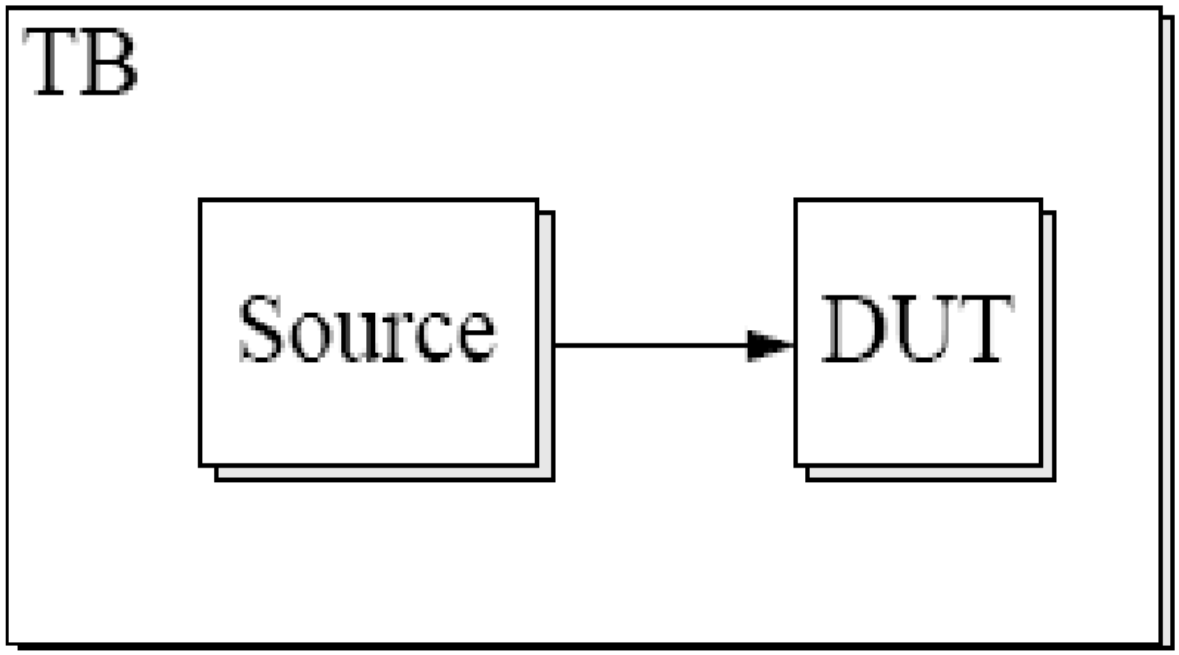
```
nReset <= '0', '1' after PERIOD;
a    <=  "00000000",
         "00000001" after 2*PERIOD,
         "01100100" after 3*PERIOD,
         "11111111" after 4*PERIOD;
b    <=  "00000000",
         "00000001" after 2*PERIOD,
         "00000101" after 3*PERIOD,
         "00000001" after 4*PERIOD;
```

Simulation



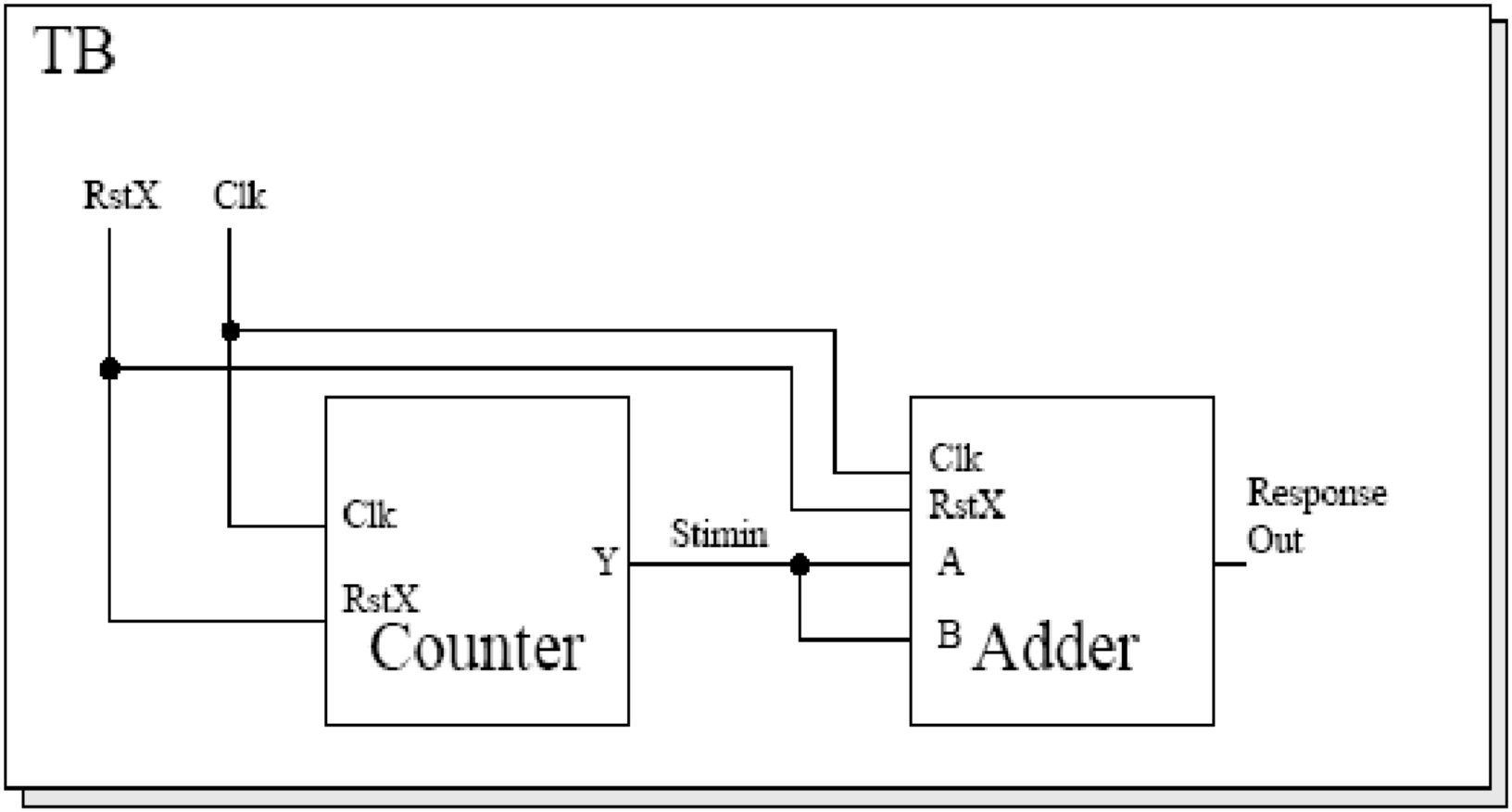
Test Bench with Separate Source

- Source and DUT instantiated into Test Bench.
- For designs with complex inputs and simple output.
- Source can be for instance an entity or a process.



Test Bench with Separate Source Example:

- Input stimuli for adder is generated in a separate entity counter

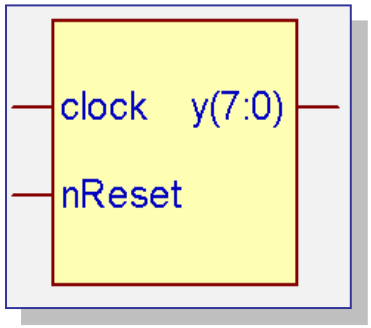


- Source is a clock triggered up-counter.

Test Bench with Separate Source

- Source component

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

entity tb_counter is
    port(
        clock   : in std_logic;
        nReset  : in std_logic;
        y       : out std_logic_vector(7 downto 0)
        );
end tb_counter;

architecture bhv of tb_counter is
begin
    process (clock, nReset)
        variable count: std_logic_vector(7 downto 0);
    begin
        if nReset='0' then
            count := (others => '0');
            y <= (others => '0');
        elsif clock'event and clock='1' then
            count := count + 1;
            y <= count;
        end if;
    end process;
end bhv;
```



Test Bench with Separate Source

Test Bench

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

    -- Add your library and packages declaration here ...

entity adder_reg_ss_tb is
end adder_reg_ss_tb;

architecture TB_ARCHITECTURE of adder_reg_ss_tb is
    -- Component declaration of the tested unit
    component adder_reg
    port(
        clock : in std_logic;
        nReset : in std_logic;
        a : in std_logic_vector(7 downto 0);
        b : in std_logic_vector(7 downto 0);
        y : out std_logic_vector(7 downto 0) );
    end component;

    -- Component declaration of the "tb_counter(bhv)" unit
    component tb_counter
    port(
        clock : in std_logic;
        nReset : in std_logic;
        y : out std_logic_vector(7 downto 0));
    end component;
```

```
    -- Stimulus signals - signals mapped to the input and
    -- inout ports of tested entity
    signal nReset : std_logic;
    -- Observed signals - signals mapped to the output
    -- ports of tested entity
    signal y : std_logic_vector(7 downto 0);

    -- Add your code here ...
    constant PERIOD : time := 50 ns;
    signal clock : std_logic := '0';
    signal stim_in : std_logic_vector(7 downto 0);

begin

    -- Unit Under Test port map
    UUT : adder_reg
        port map (
            clock => clock,
            nReset => nReset,
            a => stim_in,
            b => stim_in,
            y => y
        );

    stim_gen : tb_counter
    port map(
        clock => clock,
        nReset => nReset,
        y => stim_in
    );
```

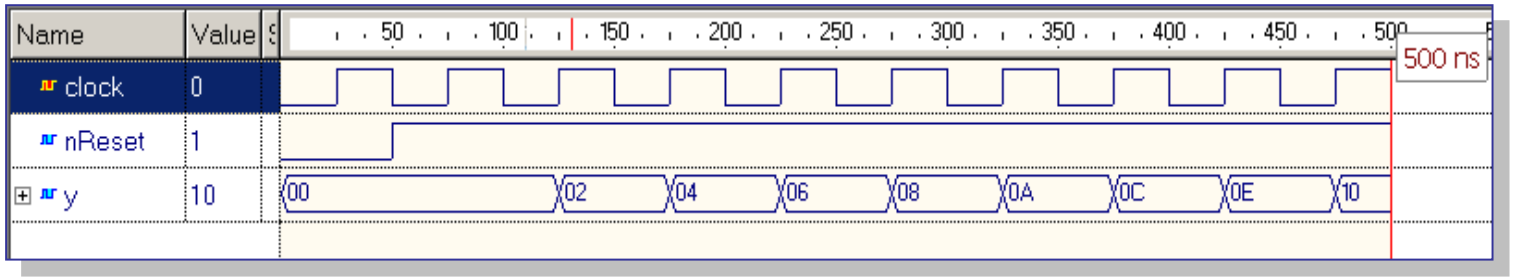
Test Bench with Separate Source

Test Bench

```
    -- Add your stimulus here ...
    gen_clock : process (clock)
    begin
        clock <= not clock after PERIOD/2;
    end process;

    nReset <= '0', '1' after PERIOD;

end TB_ARCHITECTURE;
```

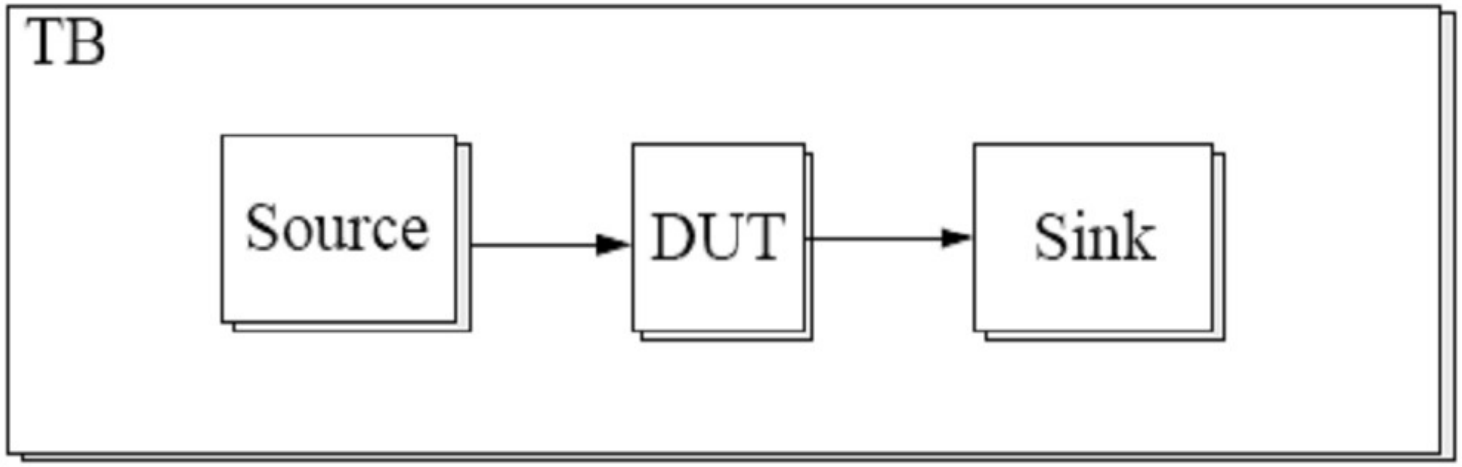
```
configuration TESTBENCH_FOR_adder_reg of adder_reg_ss_tb is
    for TB_ARCHITECTURE
        for UUT : adder_reg
            use entity work.adder_reg(bhv);
        end for;
        for stim_gen : tb_counter
            use entity work.tb_counter(bhv);
        end for;
    end for;
end TESTBENCH_FOR_adder_reg;
```

Simulation



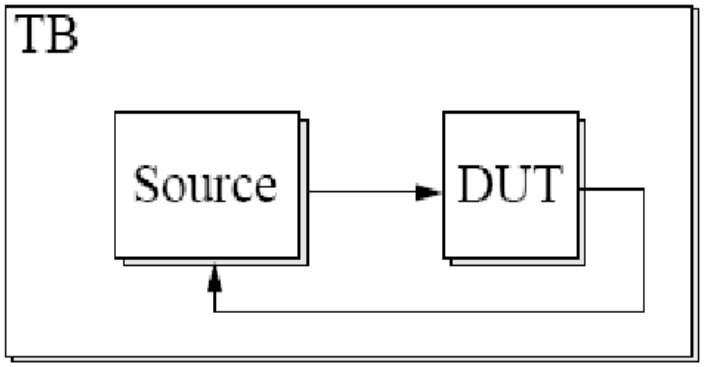
Test Bench with a Separate Source and Sink

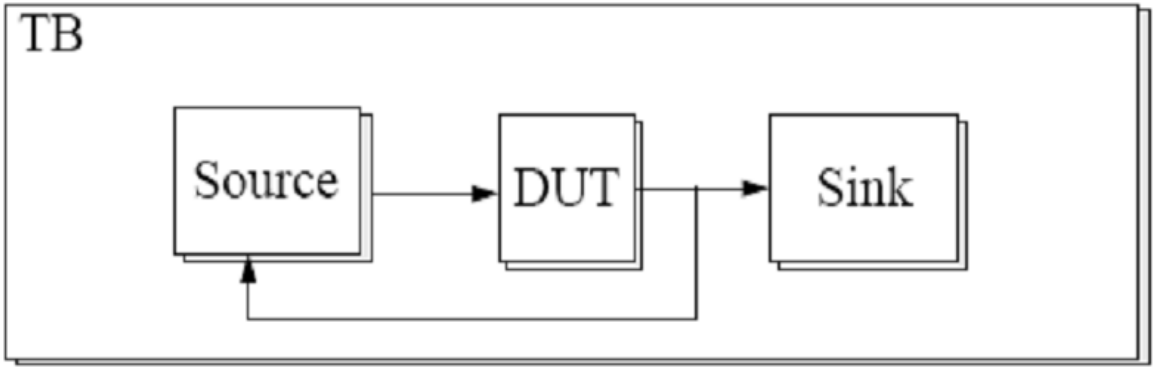
- Both the stimulus source and sink are in separate instances.
- Complex source and sink without response-source interaction.



Smart Test Bench

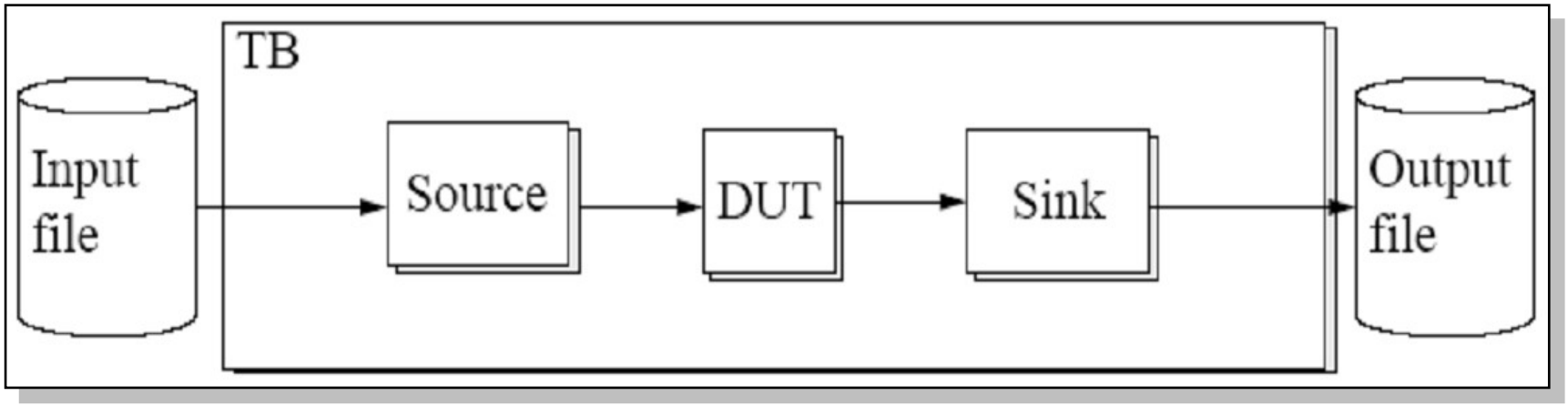
- When circuit's response affects further stimulus.





Test Bench with Text-IO

- Stimulus for DUT is read from an input file and the data is modified in the source.
- The response modified in the sink is written in the output file.



Test Bench with Text-IO

Test Bench

```
library ieee;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_1164.all;
library std;
use std.textio.all;
use ieee.std_logic_textio.all;

entity adder_reg_textio_tb is
end adder_reg_textio_tb;

architecture TB_ARCHITECTURE of adder_reg_textio_tb is
    -- Component declaration of the tested unit
    component adder_reg
    port(
        clock : in std_logic;
        nReset : in std_logic;
        a : in std_logic_vector(7 downto 0);
        b : in std_logic_vector(7 downto 0);
        y : out std_logic_vector(7 downto 0) );
    end component;

    -- Stimulus signals - signals mapped to the input
    -- and inout ports of tested entity
    signal nReset : std_logic;
    signal a : std_logic_vector(7 downto 0);
    signal b : std_logic_vector(7 downto 0);
    -- Observed signals - signals mapped to the output
    -- ports of tested entity
    signal y : std_logic_vector(7 downto 0);
    -- Add your code here ...
    constant PERIOD : time := 50 ns;
    signal clock : std_logic := '0';

begin

    -- Unit Under Test port map
    UUT : adder_reg
        port map (
            clock   => clock,
            nReset  => nReset,
            a       => a,
            b       => b,
            y       => y
        );

    -- Add your stimulus here ...
    gen_clock : process (clock)
    begin
        clock <= not clock after PERIOD/2;
    end process;

    nReset <= '0', '1' after PERIOD;
```

Test Bench with Text-IO

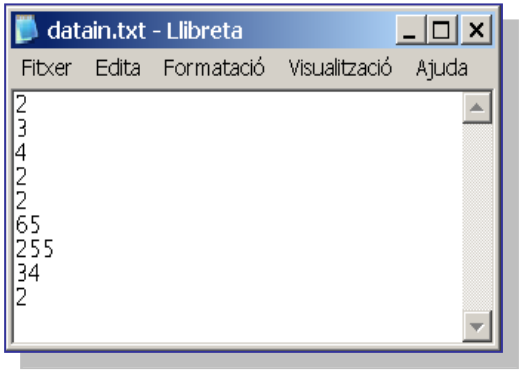
Test Bench

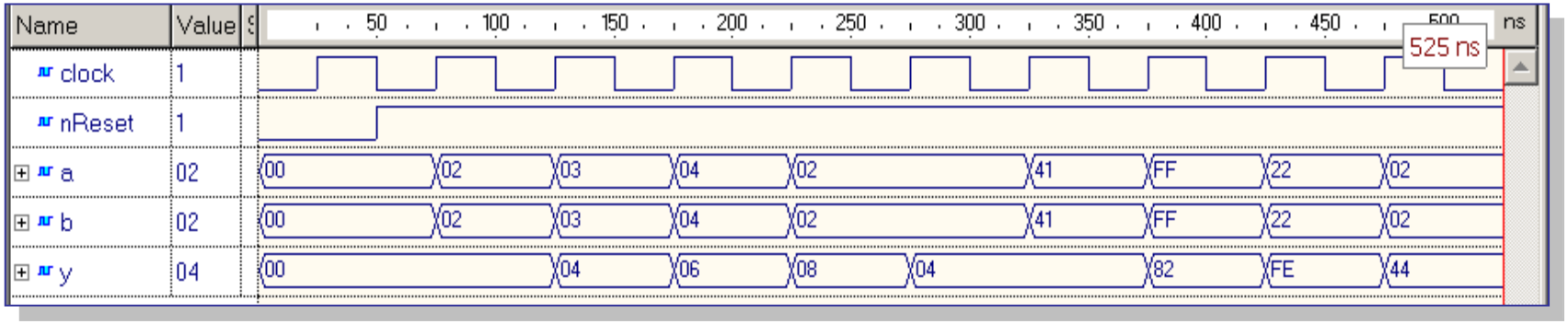
```
    process(clock, nReset)
        file file_in : text is in "datain.txt";
        file file_out : text is out "dataout.txt";
        variable line_in : line;
        variable line_out : line;
        variable input_tmp : integer;
        variable output_tmp : integer;
    begin
        if nReset='0' then
            a <= (OTHERS => '0');
            B <= (OTHERS => '0');
        elsif clock'event and clock='1' then
            -- Read one line from the input file
            -- and move data to "input_tmp"
            if not (endfile(file_in)) then
                readline(file_in, line_in);
                read(line_in, input_tmp);

                a <= CONV_STD_LOGIC_VECTOR(input_tmp, 8);
                b <= CONV_STD_LOGIC_VECTOR(input_tmp, 8);

                output_tmp := CONV_INTEGER(y);
                write(line_out, output_tmp);
                writeline(file_out, line_out);
            else
                assert false
                    report "****** End of file ! ******"
                severity note;
            end if;
        end if;
    end process;
end TB_ARCHITECTURE;
```

```
configuration TESTBENCH_FOR_adder_reg of adder_reg_textio_tb is
    for TB_ARCHITECTURE
        for UUT : adder_reg
            use entity work.adder_reg(bhv);
        end for;
    end for;
end TESTBENCH_FOR_adder_reg;
```

Test Bench with Text-IO

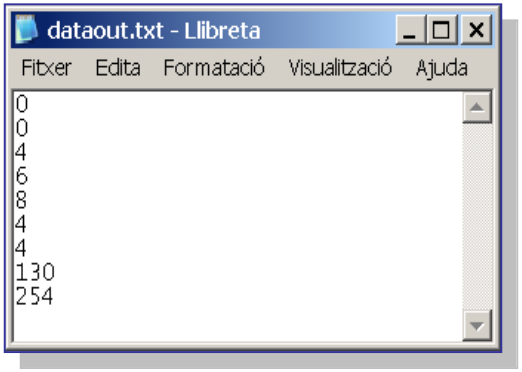
- Simulation





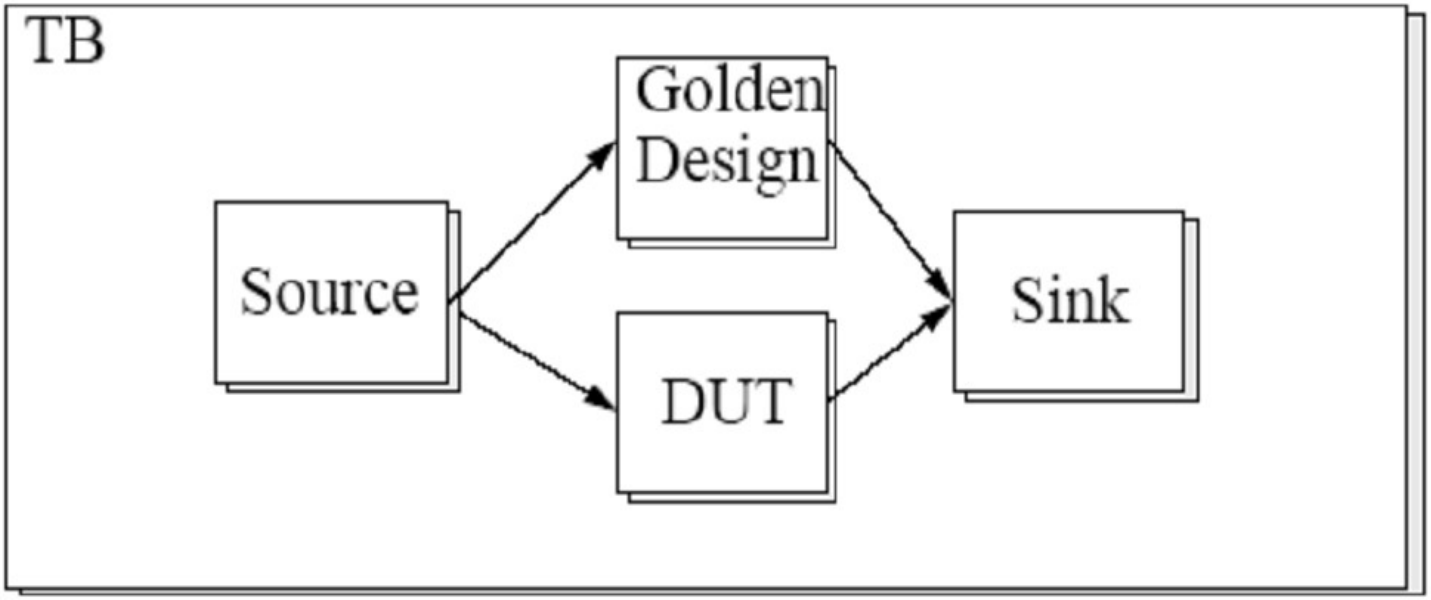

```
#  Simulation has been initialized
#  Selected Top-Level: testbench_for_adder_reg
run @525ns
# EXECUTION:: NOTE    : ****** End of file ! ******
# EXECUTION:: Time: 525 ns,  Iteration: 0,  TOP instance,  Process: line__76.
# KERNEL: stopped at time: 525 ns
>
```

Console



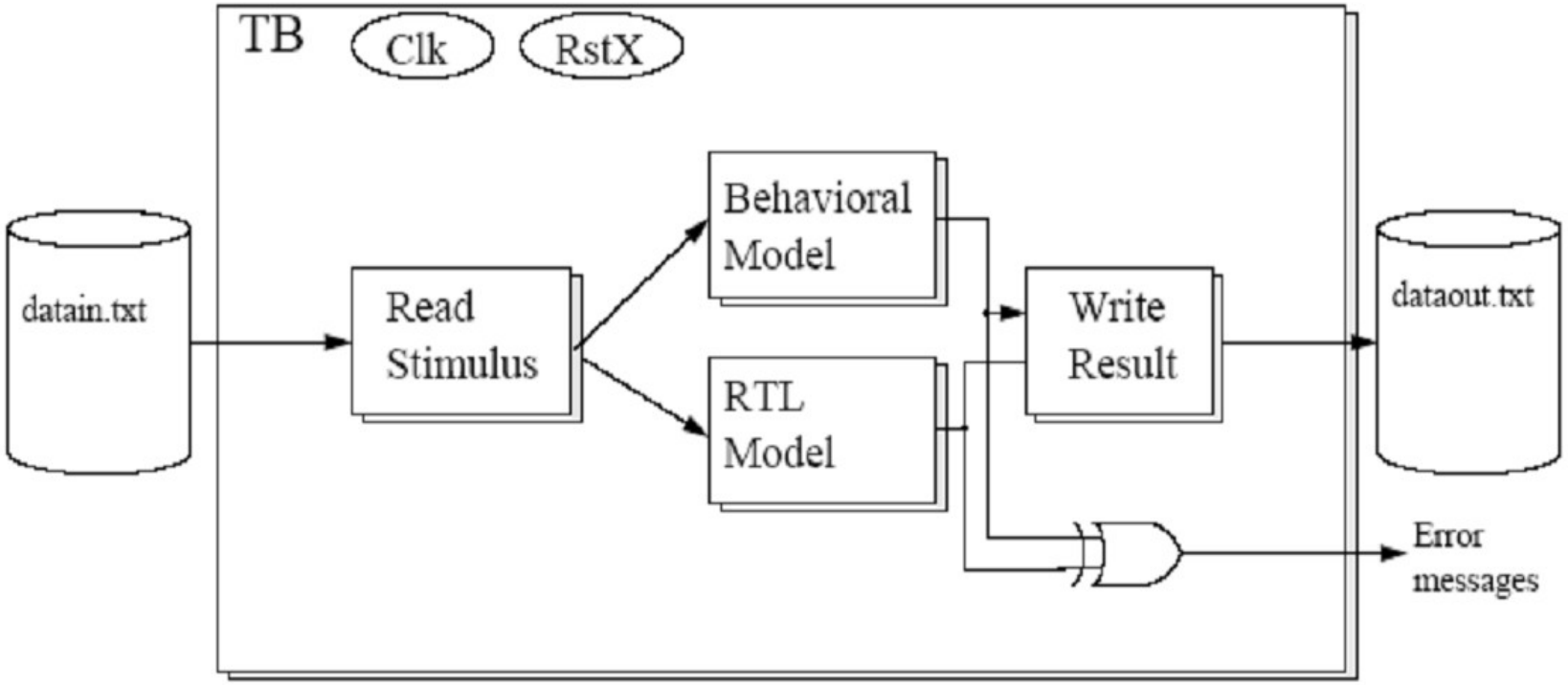
Test Bench using Golden Design

- DUT is compared to the specification i.e. the golden design.



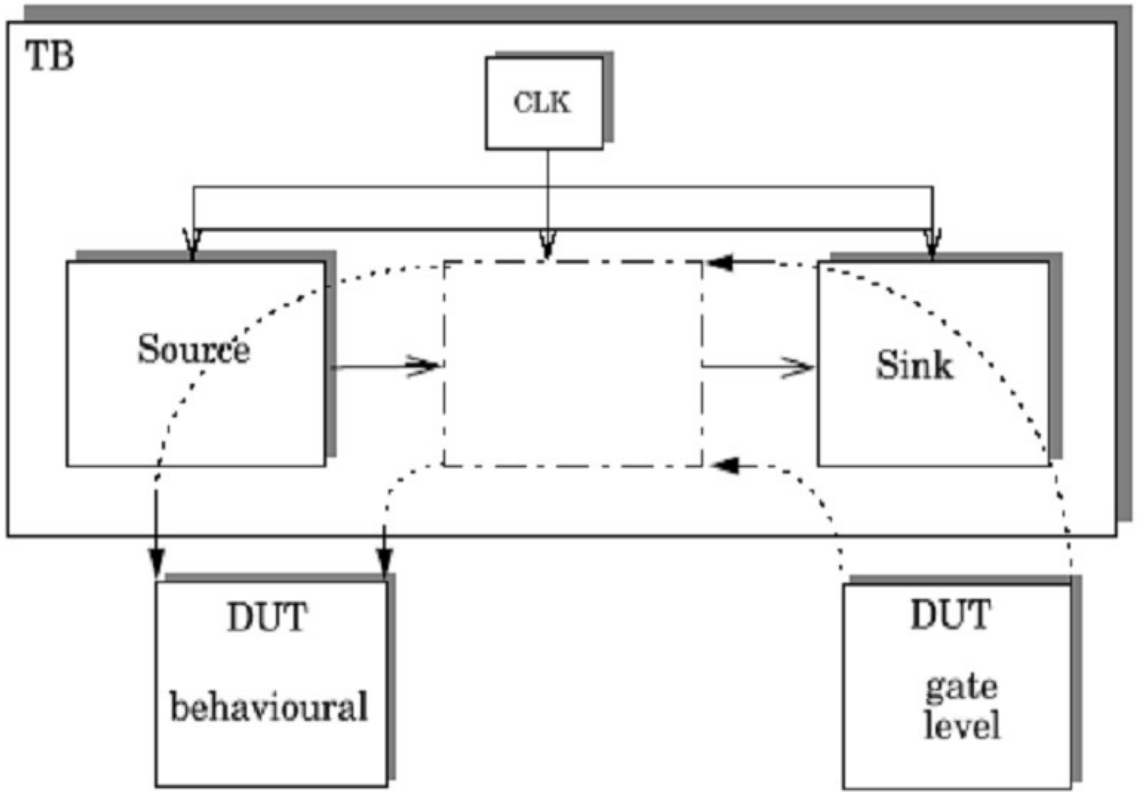
Test Bench using Golden Design

- Simulation with a Golden Design



Same Test Bench for different Designs (or architectures)

- Architecture of the DUT can be changed using the configuration.



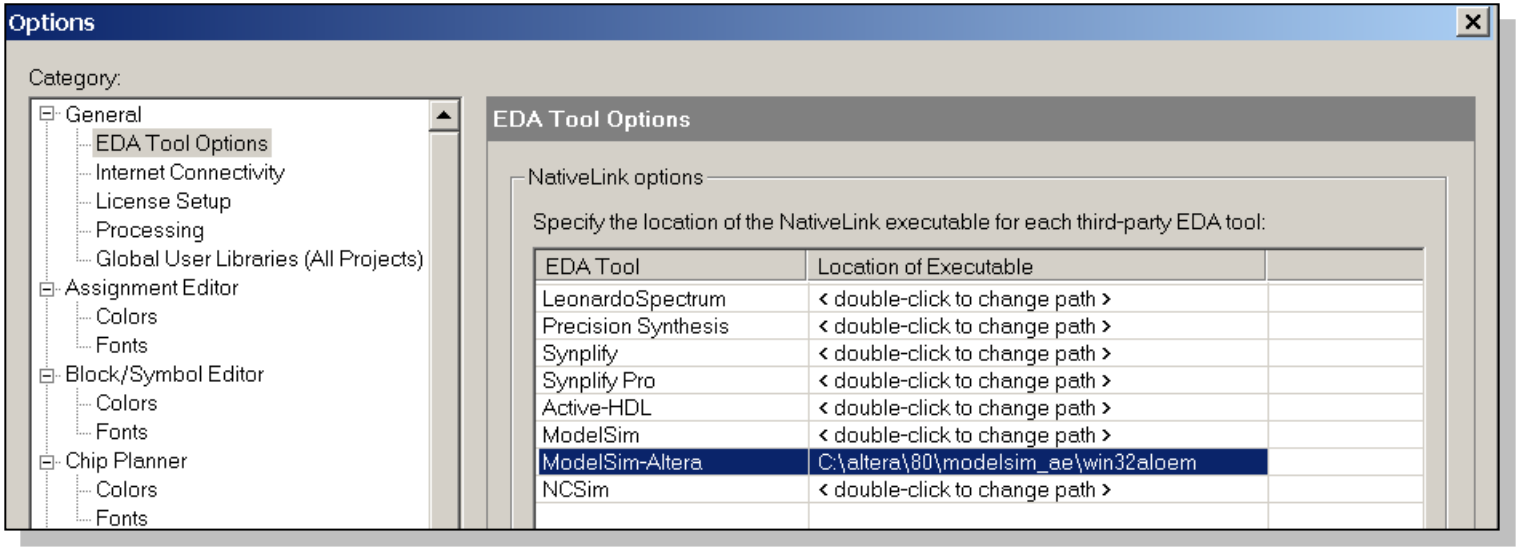
- Introduction
- Test Bench Structures
- **Modelsim-Altera simulation**
- Advanced Test Bench

Modelsim-Altera simulator

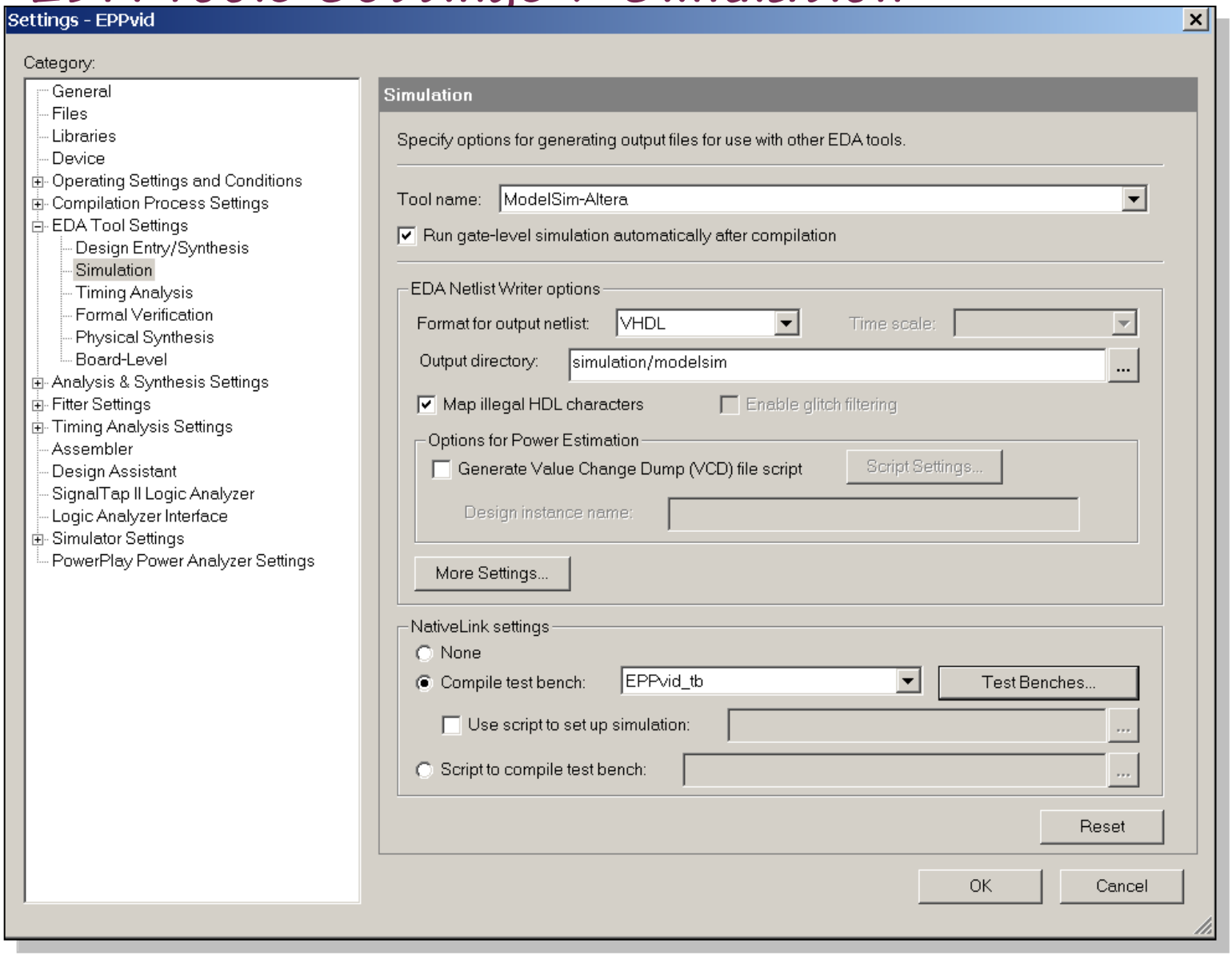
- Invoked directly from Altera's Quartus II tool.
- Needed to use test benches (Quartus simulator only allows waveform entry).

Setup path:

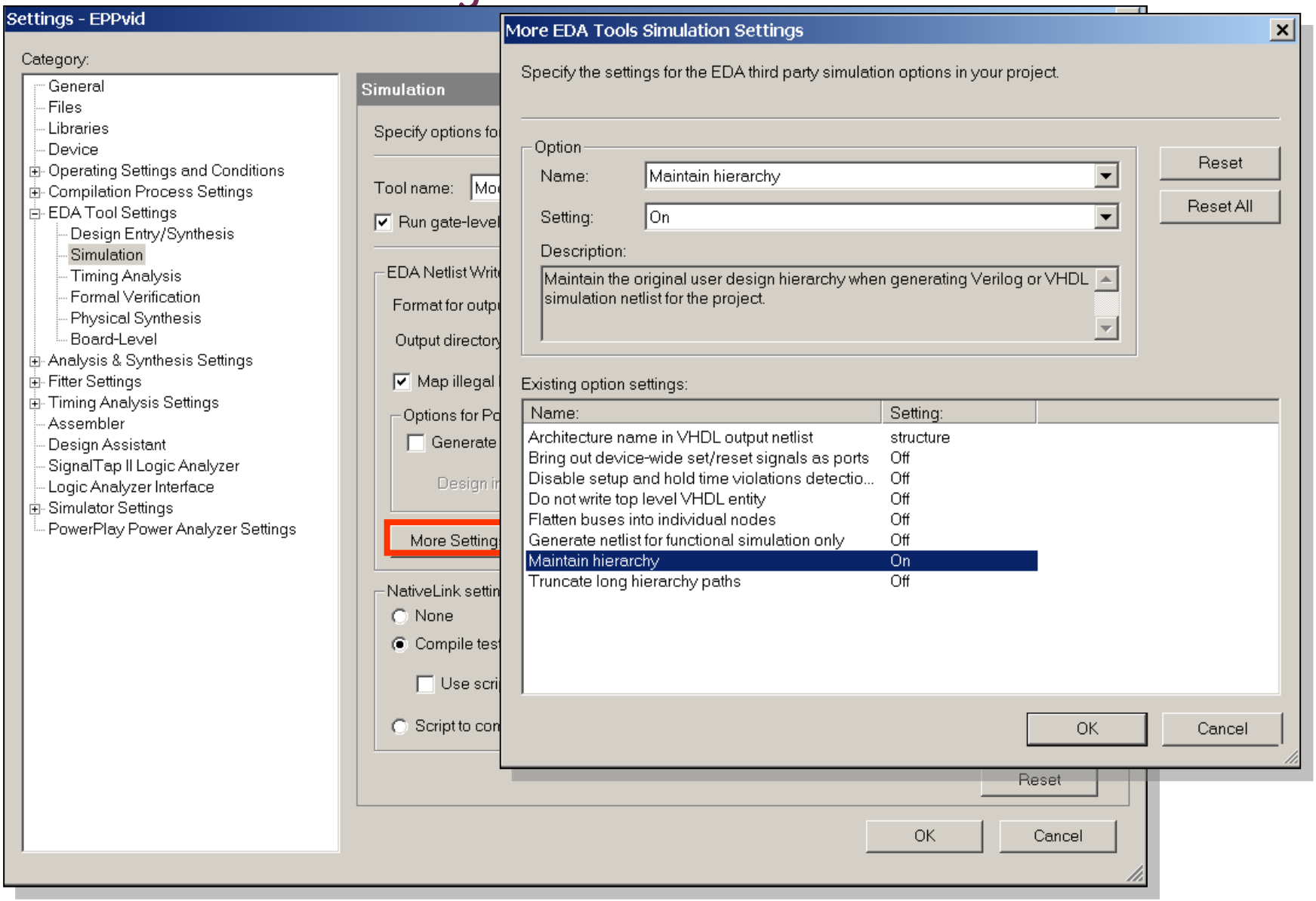
- Quartus II > Tools > Options



EDA Tools Settings > Simulation

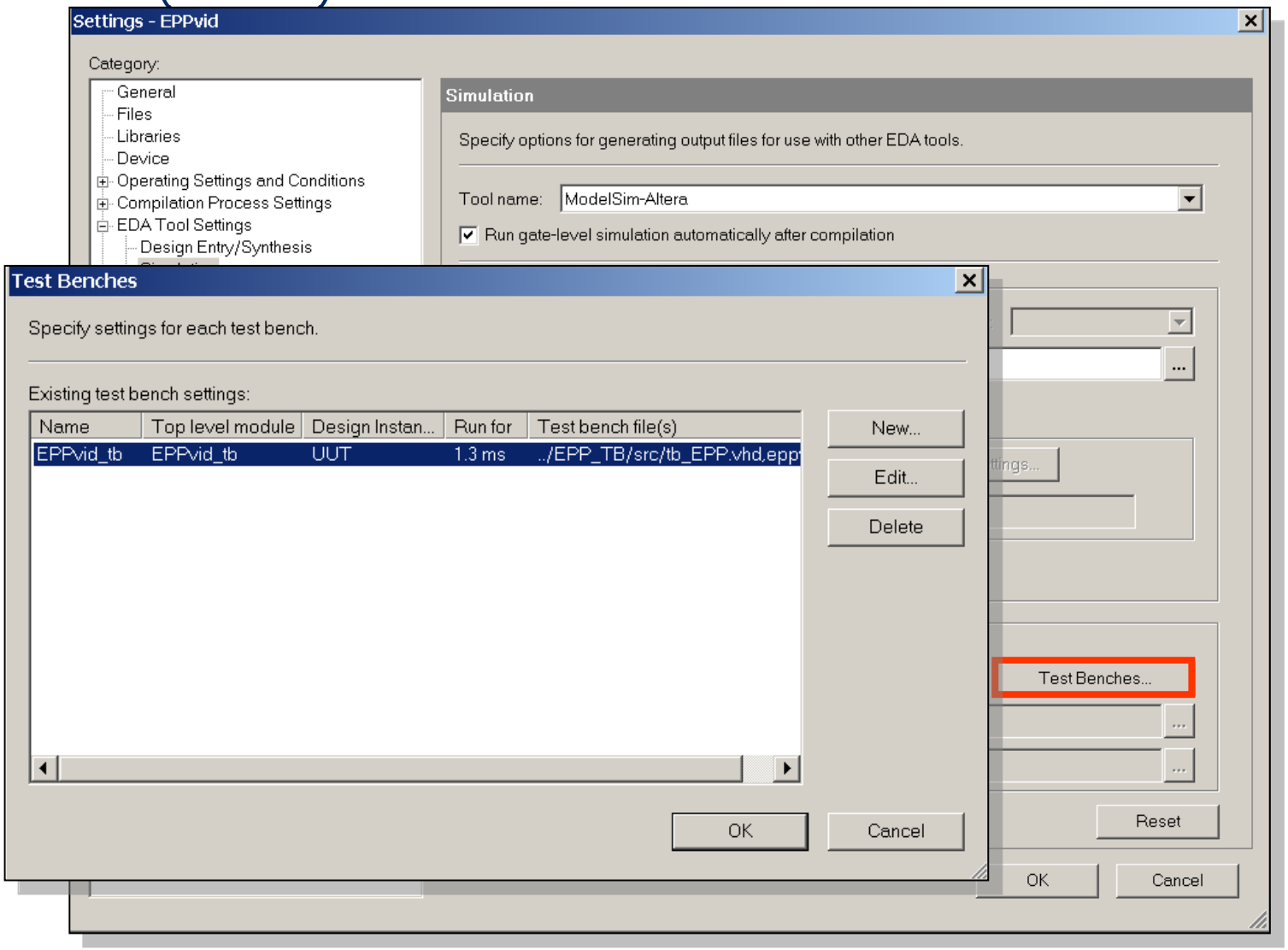


EDA Tools Settings > Simulation



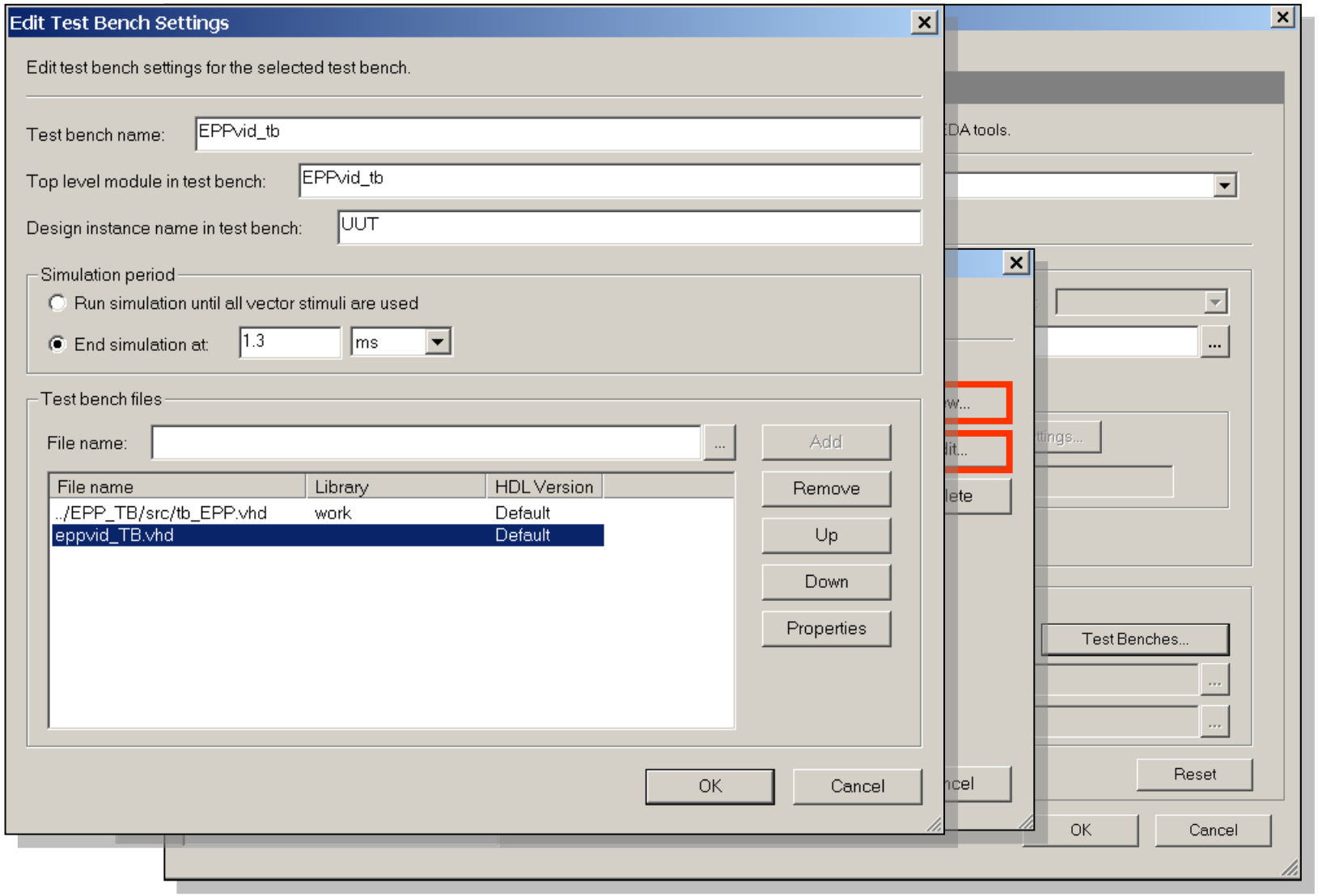
EDA Tools Settings > Simulation > Test Benches

- New (or Edit)



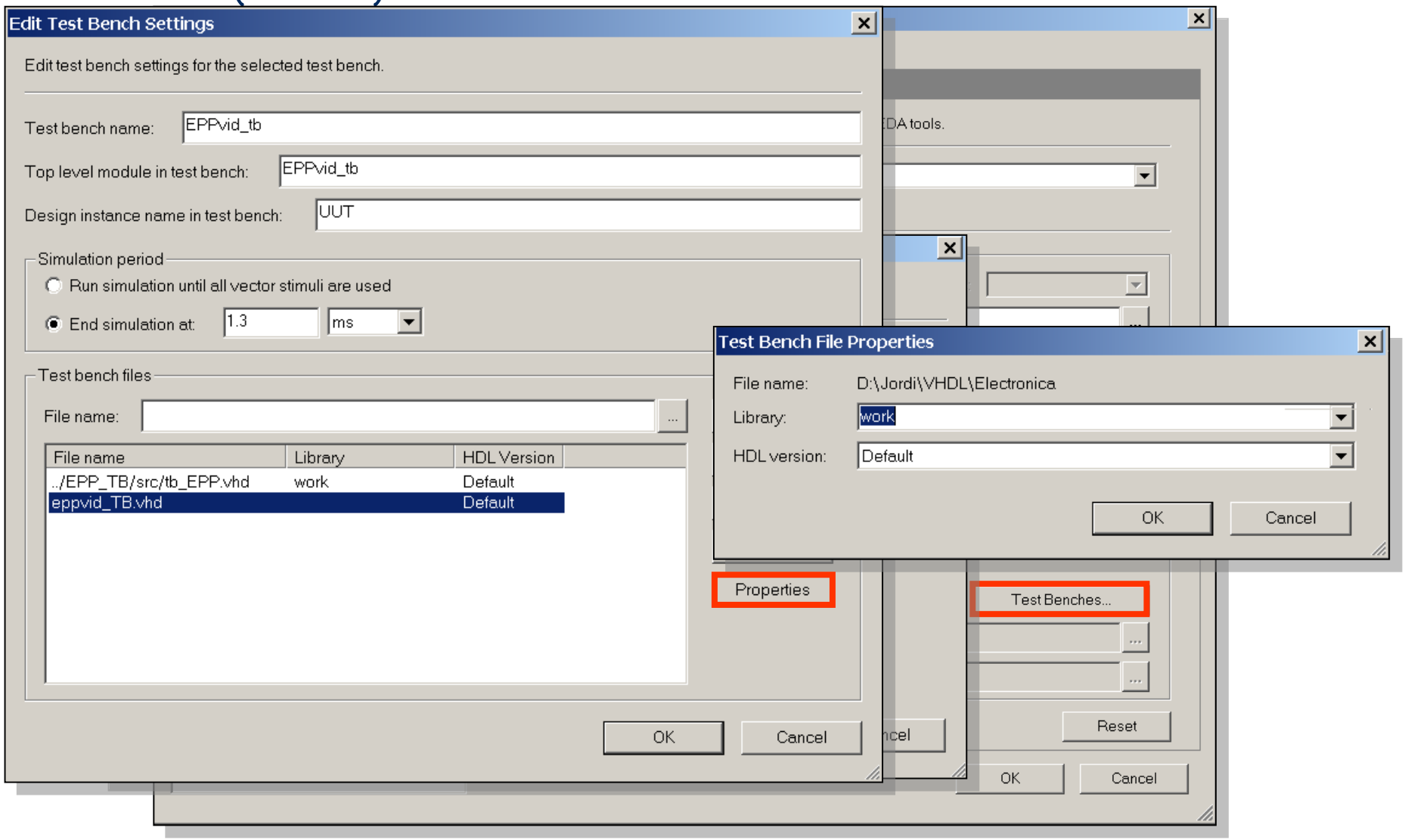
EDA Tools Settings > Simulation > Test Benches

- New (or Edit)

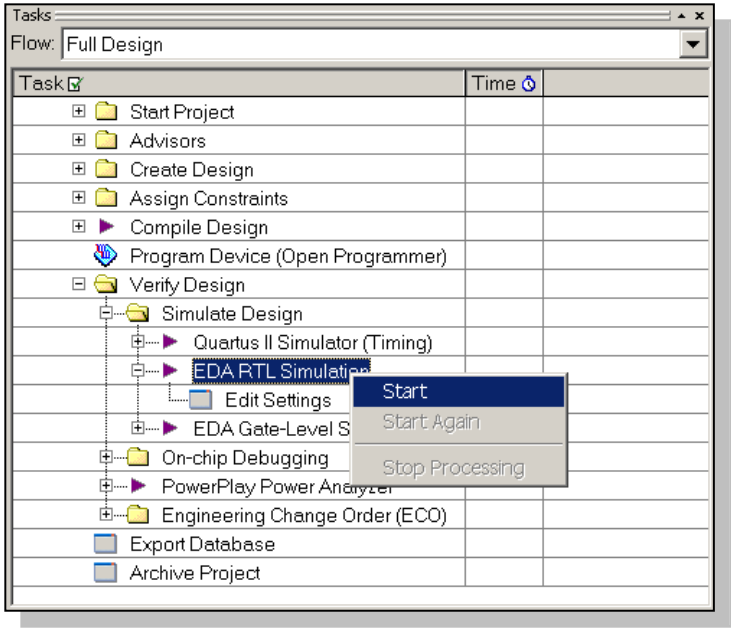


EDA Tools Settings > Simulation > Test Benches

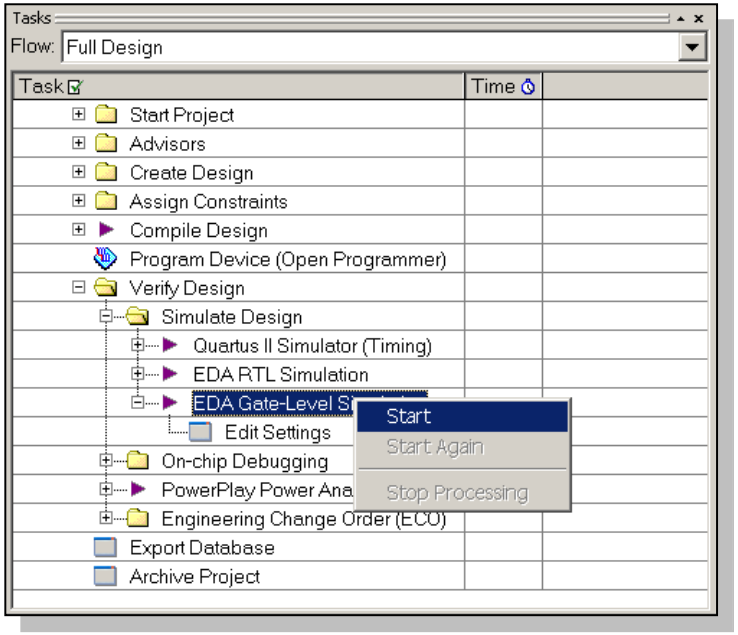
- New (or Edit)



- **Functional Simulation**
  - Fast, but no timing information.

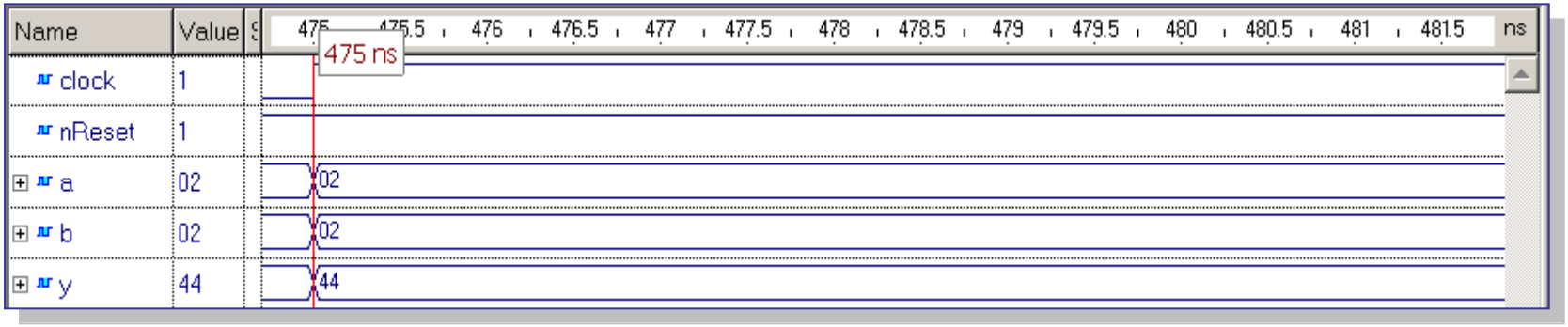


- **Post-layout (Gate-Level) Simulation**
  - Slower, but detailed timing information available.

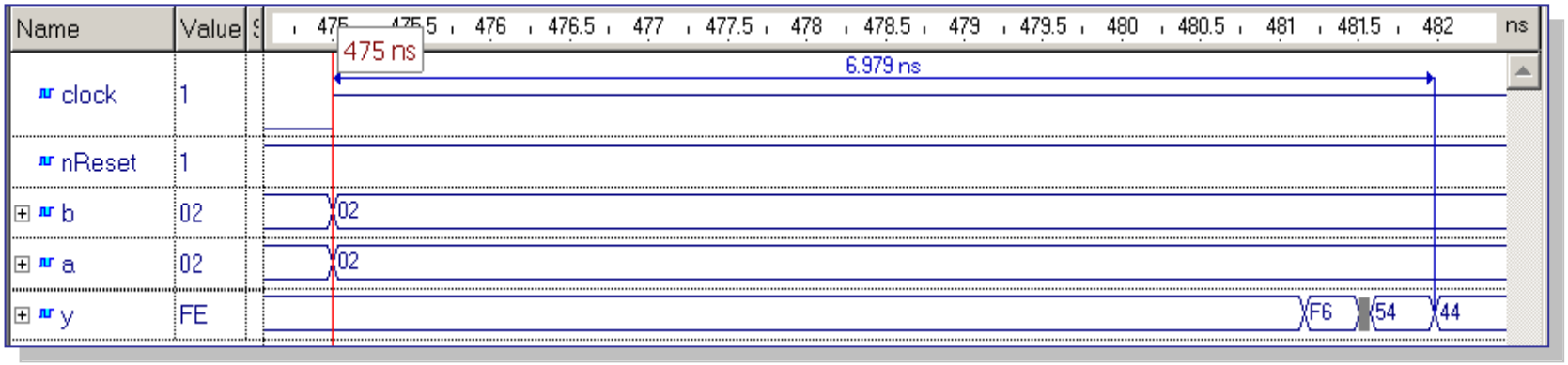


Test Bench: functional vs. timing simulation

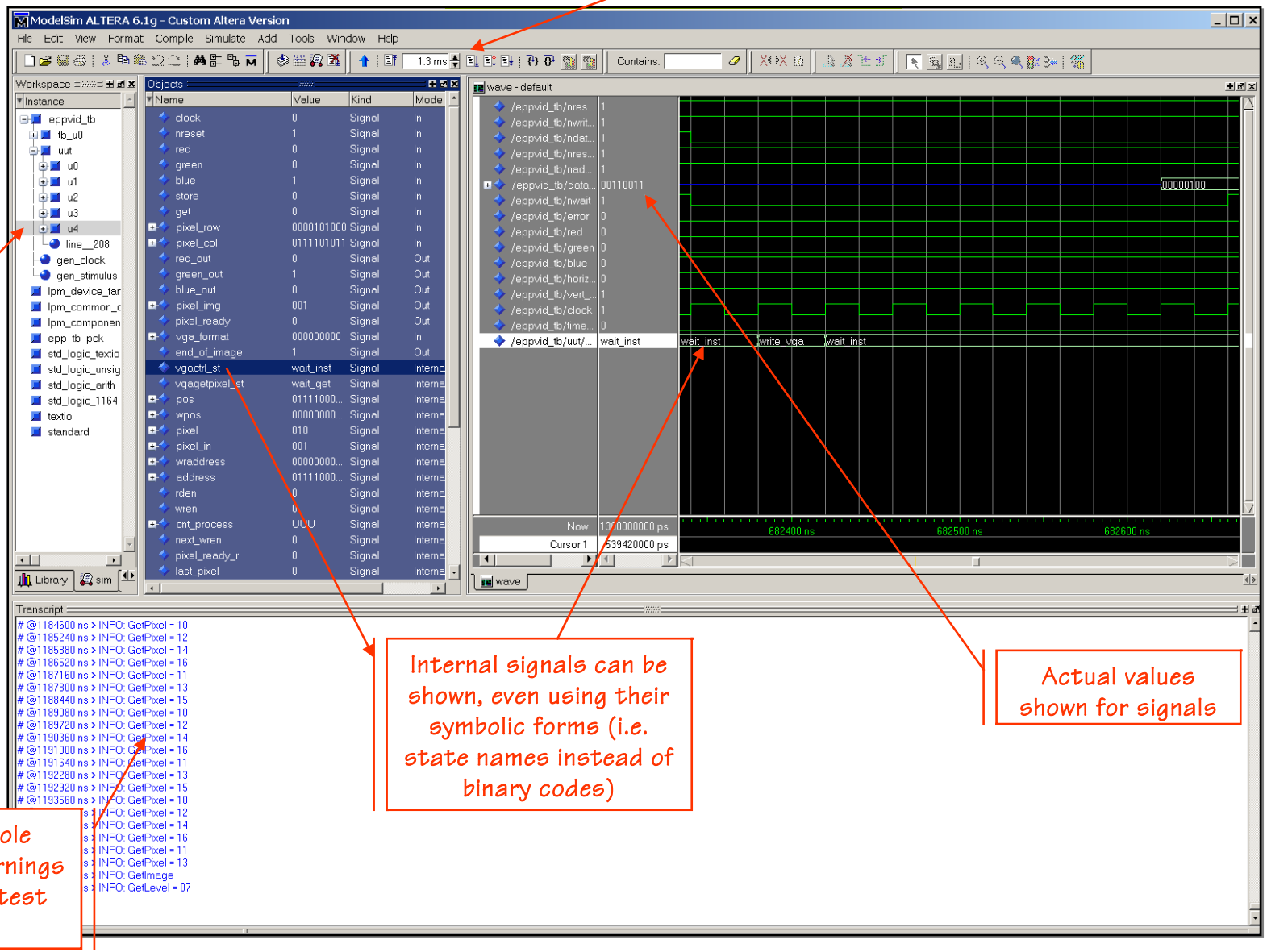
- Functional:
  - Ideal, "instantaneous" response.



- Timing (Gate-Level simulation):
  - Notice delay and glitches.



Modelsim Execution



- Introduction
- Test Bench Structures
- Modelsim-Altera simulation
- Advanced Test Bench

Using Records to store vectors:

```
    -- Add your code here ...
    constant PERIOD : time := 50 ns;
    signal clock : std_logic := '0';
    signal error_sim : std_logic := '0';

    type test_record is record
        nReset  : std_logic;     -- Inputs
        a       : std_logic_vector(7 downto 0);
        b       : std_logic_vector(7 downto 0);
        y       : std_logic_vector(7 downto 0); -- Expected result
    end record;

    type test_array is array(positive range <>) of test_record;

    constant test_vectors : test_array := (
    -- nReset   a   b               y
    ('0',   CONV_STD_LOGIC_VECTOR(3, 8),    CONV_STD_LOGIC_VECTOR(4, 8),    CONV_STD_LOGIC_VECTOR(0, 8)),
    ('1',   CONV_STD_LOGIC_VECTOR(1, 8),    CONV_STD_LOGIC_VECTOR(1, 8),    CONV_STD_LOGIC_VECTOR(2, 8)),
    ('1',   CONV_STD_LOGIC_VECTOR(65, 8),   CONV_STD_LOGIC_VECTOR(4, 8),    CONV_STD_LOGIC_VECTOR(69, 8)),
    ('1',   X"CD",                          "00010011",                     CONV_STD_LOGIC_VECTOR(69, 8)),
    ('1',   CONV_STD_LOGIC_VECTOR(255, 8),  CONV_STD_LOGIC_VECTOR(1, 8),    CONV_STD_LOGIC_VECTOR(0, 8))
    );

begin

    -- Unit Under Test port map
    UUT : adder_reg
        port map (
            clock => clock,
            nReset => nReset,
            a => a,
            b => b,
            y => y
        );

    -- Add your stimulus here ...
    gen_clock : process (clock)
    begin
        clock <= not clock after PERIOD/2;
    end process;
```

Using Records to store vectors:

- Allows stimulus application:
- … and response analysis:

```
    test_run: process
        variable vector : test_record;
        variable last_y : std_logic_vector(7 downto 0);
    begin
        -- Initialization
        last_y  := test_vectors(1).y;

        for i in test_vectors'range loop
            -- Apply stimulus
            vector := test_vectors(i);
            nReset  <= vector.nReset;
            a       <= vector.a;
            b       <= vector.b;
            wait for PERIOD/2;

            -- Verify results from previous cycle
            assert last_y = y
                report "******* Output mismatch! *******"
                severity WARNING;
            if (last_y = y) then
                error_sim <= '0';
            else
                error_sim <= '1';
            end if;

            -- Set result to verify on next cycle
            last_y := vector.y;

            -- Operate
            wait for PERIOD/2;
        end loop;

        wait for PERIOD;

        assert false
            report "****** End of stimulus ! ******"
            severity note;
    end process;
end TB_ARCHITECTURE;
```

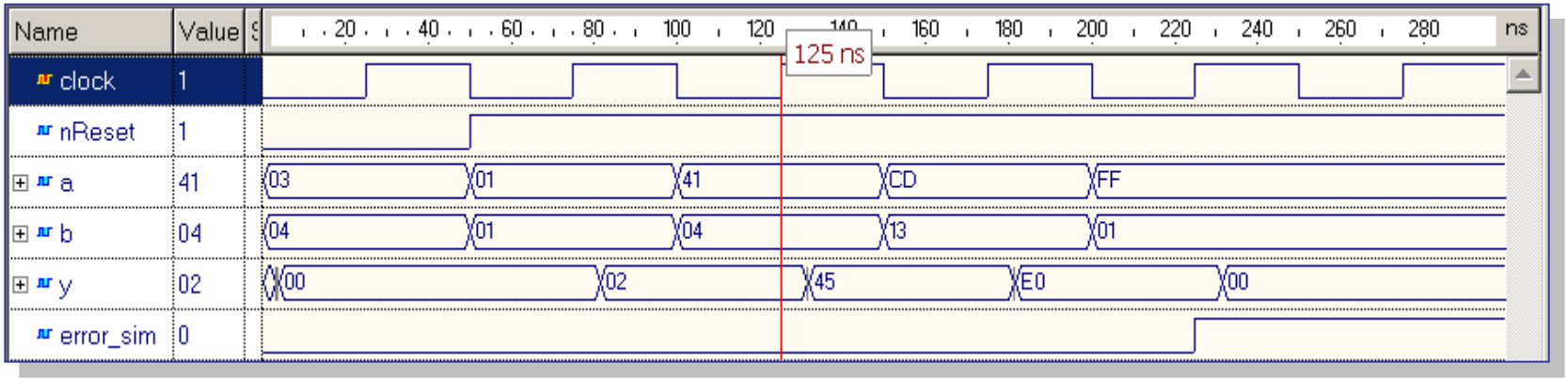
Using Records to store vectors:

- Allows easy designer notification in case of simulation errors:
  - "Assert" command allows console notification

```
run 300ns
# EXECUTION:: WARNING: ****** Output mismatch! ******
# EXECUTION:: Time: 225 ns,  Iteration: 0,  TOP instance,  Process: test_run.
# EXECUTION:: NOTE    : ****** End of stimulus ! ******
# EXECUTION:: Time: 300 ns,  Iteration: 0,  TOP instance,  Process: test_run.
# KERNEL: stopped at time: 300 ns
>
```

Console

  - Timing simulation using "error_sim "signal for error notification:



Complex I/O on text files can produce similar results:

```
read_cmd: process
    file cmdfile : TEXT;     -- Define file "handle"
    variable line_in : line;
    variable line_out : line;
    variable isOk : boolean;

    variable vector : test_record;
    variable last_y : std_logic_vector(7 downto 0);
begin

    FILE_OPEN(cmdfile, "cmdfile.txt", READ_MODE);
    last_y := CONV_STD_LOGIC_VECTOR(0, 8);

    loop
        if endfile(cmdfile) then
            assert false report "****** EOF ******" severity NOTE;
            exit;
        end if;

        readline(cmdfile, line_in);
        next when line_in'length=0; -- Skip empty lines

        read(line_in, vector.nReset, isOk);
        assert isOk report "Text I/O read error" severity ERROR;

        hread(line_in, vector.a, isOk);
        assert isOk report "Text I/O read error" severity ERROR;

        hread(line_in, vector.b, isOk);
        assert isOk report "Text I/O read error" severity ERROR;

        hread(line_in, vector.y, isOk);
        assert isOk report "Text I/O read error" severity ERROR;

        -- Aply stimulus
        nReset  <= vector.nReset;
        a       <= vector.a;
        b       <= vector.b;

        wait for PERIOD/2;

        -- Check outputs
        if (last_y = y) then
            error_sim <= '0';
            write(line_out, "Ok  : ");
        else
            write(line_out, "FAIL :");
            error_sim <= '1';
        end if;

        -- Show output
        hwrite(line_out, a, RIGHT, 2);
        write(line_out, " + ");
        hwrite(line_out, b, RIGHT, 2);
        write(line_out, " = ");
        hwrite(line_out, y, RIGHT, 2);
        writeline(OUTPUT, line_out);

        last_y := vector.y;

        wait for PERIOD/2;

    end loop;

    wait;

end process;
```

Complex I/O on text files (Results):

- Stimulus file:

cmdfile.txt - Llibreta

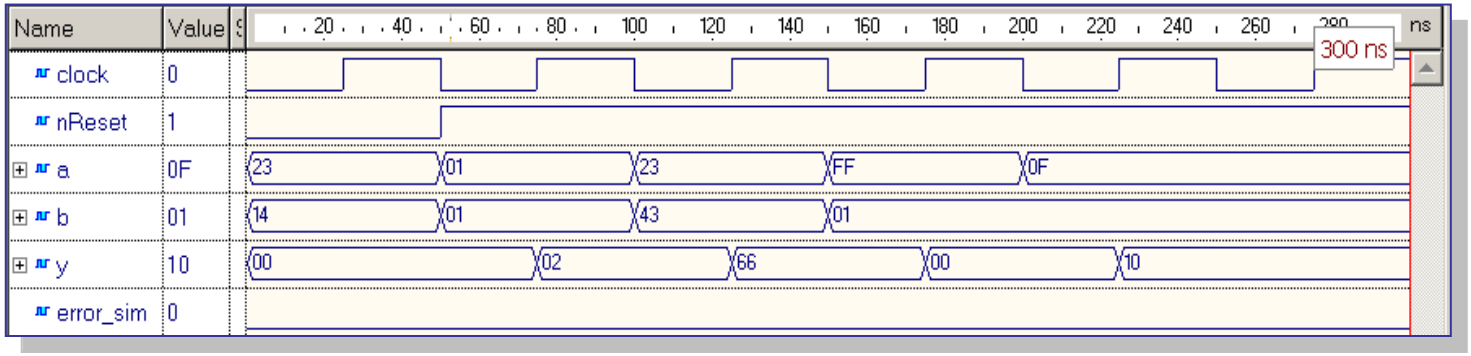
Fitxer Edita Formatació Visualització Ajuda

```
0 23 14 00

1 01 01 02
1 23 43 66
1 ff 01 00
1 0f 01 10
```

- Simulation Results:

```
run 300ns;
# KERNEL: Ok   : 23 + 14 = 00
# KERNEL: Ok   : 01 + 01 = 00
# KERNEL: Ok   : 23 + 43 = 02
# KERNEL: Ok   : FF + 01 = 66
# KERNEL: Ok   : 0F + 01 = 00
# EXECUTION:: NOTE    : ****** EOF ******
# EXECUTION:: Time: 250 ns,  Iteration: 0,  TOP instance,  Process: read_cmd.
# KERNEL: stopped at time: 300 ns
>
```

Console



Random patterns:

- Use math_real package:

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_unsigned.ALL;
use ieee.math_real.all;
use ieee.std_logic_textio.all;
library std;
use std.textio.all;
```

- Procedure:
  - Initialize seeds
    - Each pair of values produce the same result (each time the simulation is executed on the same simulator; they are pseudo-random numbers).
  - Use ***uniform()*** function and scale its result.

```
read_cmd: process
    variable line_out : line;
    variable last_y : std_logic_vector(7 downto 0);
    variable isOk : boolean;
    variable seed1, seed2: positive;
    variable r: real;
begin
    -- Initialize seed with random values
    seed1 := 123;
    seed2 := 12131;

    nReset <= '0';
    last_y := CONV_STD_LOGIC_VECTOR(0, 8);
    wait for PERIOD;

    nReset <= '1';

    for i in 1 to 100 loop
        -- Apply stimulus
        uniform(seed1, seed2, r);
        a   <= CONV_STD_LOGIC_VECTOR(integer(r*256.0), 8);
        uniform(seed1, seed2, r);
        b   <= CONV_STD_LOGIC_VECTOR(integer(r*256.0), 8);
        wait for PERIOD/2;

        -- Check outputs
        if (last_y = y) then
            error_sim <= '0';
            write(line_out, string'("Ok  : "));
        else
            write(line_out, string'("FAIL :"));
            error_sim <= '1';
        end if;

        -- Show output
        hwrite(line_out, a, RIGHT, 2);
        write(line_out, string'(" + "));
        hwrite(line_out, b, RIGHT, 2);
        write(line_out, string'(" = "));
        hwrite(line_out, y, RIGHT, 2);
        writeline(OUTPUT, line_out);

        last_y := a+b;
        wait for PERIOD/2;
    end loop;

    assert false report "****** End Of Simulation ******" severity NOTE;
    wait;
end process;
```

Any positive integer is allowed

Uniform distribution: gives a real between 0 and 1

Result can be scaled and converted to integer

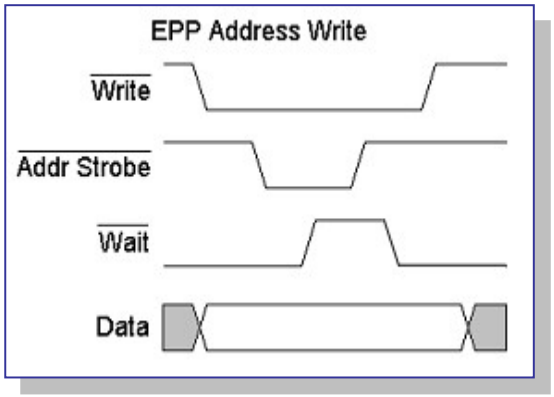
Qualification required on Modelsim

Limitations of vector-based Test Bench

- Raw vectors are adequate for testing small systems.
- However, as test suite size increases it becomes more difficult to maintain and write raw data.

Bus functional models (BFM) are the solution?

- BFM overcome the difficulties of creating and maintaining raw test data, because modular techniques can be used.
- BFM describes the behavior of the part at the interface-level without modeling the internal operation.
- BFM can be created directly from the interface information contained in data sheets.



Writing Bus Functional Models

- BFM only needs to model a small portion of the functionality:
  - None of the internal operation is modeled.
  - Only some aspects of external operation are needed.
  - However, the reactivity of the BFM makes it more difficult to check all possible behaviors and isolate errors in the test bench from the UUT.
- Therefore, first step is to decide what transactions need to be modeled to test the UUT (and obtain a description of those).
- Next step is to write HDL code for each transaction type.
  - Should be coded for reusability
- Seems clear that procedures or components should be used to model transactions.
  - Because these constructs are reusable and can contain sequential statements that allow for the passage of simulation time (so, VHDL functions can't be used).

Writing Bus Functional Models

- Procedures-based transactions are easier:
  - Less coding is required
  - Can be invoked by a simple procedure call (whereas extra control signals must be added to a component and triggered from the calling location to initiate a component's transaction).
  - Passing parameters can be done using variables instead of signals.
  - But… must be executed within the context of a single process.

- Component based-transactions
  - More powerful.
  - Can contain multiple internal processes. This allows modeling concurrently-executed sections within a transaction.
  - Can be triggered asynchronously from within an executing transaction without causing it to suspend.

Writing Bus Functional Models

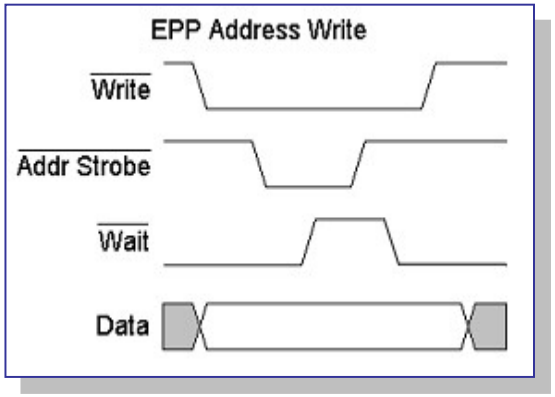
- VHDL Procedures
  - Contains:
    - Local declarations
    - Sequence of statements
  - Definition
    - Procedure declaration (optional)
      - » Name
      - » Parameters
    - Procedure body
      - » Any sequential statements are allowed (including ***wait***).
      - » However, wait can not be used in procedures which are called from a process with a sensibility list or from within a function.
  - Can be called in any place of the architecture
    - So it can be a sequential or concurrent statement.
  - Can be overloaded (different number or type of parameters).
  - Procedures can be nested.

Simplified Syntax:

```
procedure procedure_name ( formal_parameter_list )


procedure procedure_name ( formal_parameter_list ) is
        procedure_declarations
        begin
                sequential statements
        end procedure procedure_name;
```

Writing Bus Functional Models

- Example: EPP Address Write Cycle transaction
  - "Timing" diagram:



  - Procedure:
    - Relies on external variables or signals.
    - Not-reusable.
    - DIY Lab:
      - » Make it reusable.

```
procedure tb_EPP_CmdWrite(cmd : in std_logic_vector(7 downto 0)) is
begin
    nWrite_async        <= '1';
    nAddrStrobe_async   <= '1';
    if nWait='1' then
        wait until nWait='0';
    end if;

    nWrite_async        <= '0';
    nAddrStrobe_async   <= '0';
    data_port           <= cmd;
    wait until nWait='1';

    wait for PERIOD/2;
    nWrite_async        <= '1';
    nAddrStrobe_async   <= '1';
    data_port           <= (others => 'Z');
end procedure tb_EPP_CmdWrite;
```

  - Call: `tb_EPP_CmdWrite("00000000");`

- Introduction
- Test Bench Structures
- Models
- Advan

The End