VHDL for FPGA Simulation

Rodrigo A. Melo

Oct | 2021

Joint ICTP, SAIFR and UNESP School on Systems-on-Chip, Embedded Microcontrollers and their Applications in Research and Industry | (smr 3557)

Outline

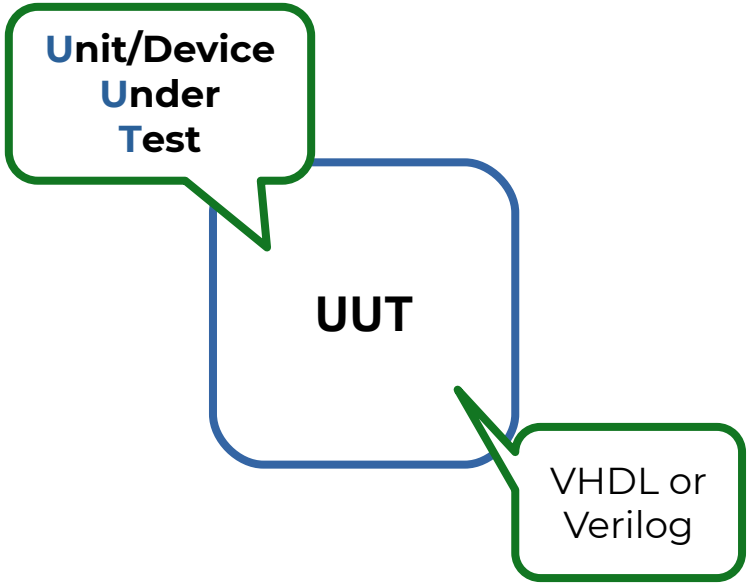

1
Introduction
2
VHDL for Simulation
3
Simulators
4
Conclusion

Introduction

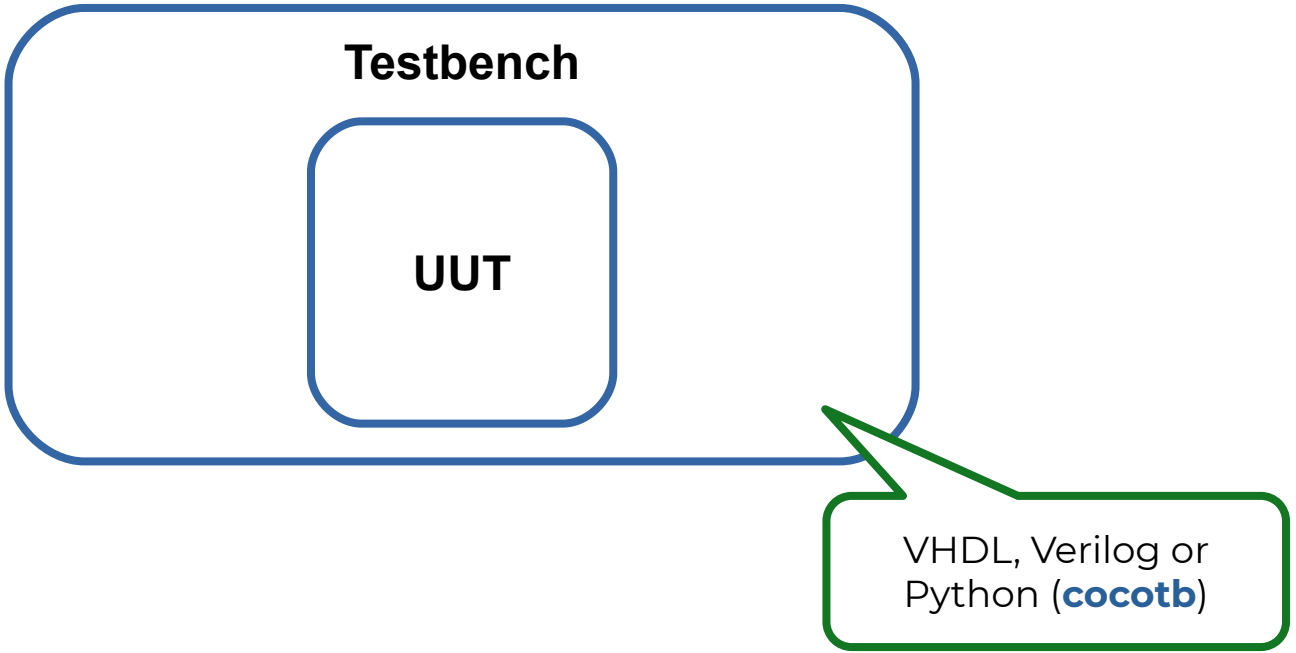
Verification

- **Verification:** to check if a design works as expected.
- **In-Circuit Debug:**
  - Performed in a test bench, with a Scope (simple design with a few signals), Logic Analyzer (**$$$**) or Embedded Logic Analyzer (modify your design).
- **Functional verification:**
  - Stimulus are provided and the output compared with expected values, according to a specification.
  - It implies a testbench (generally an HDL) and a simulator
- **Formal verification:**
  - Generally speaking, you need to specify formal properties and a tool checks your design against formal methods (mathematical proof). Ex: **SymbiYosys** (FOSS).

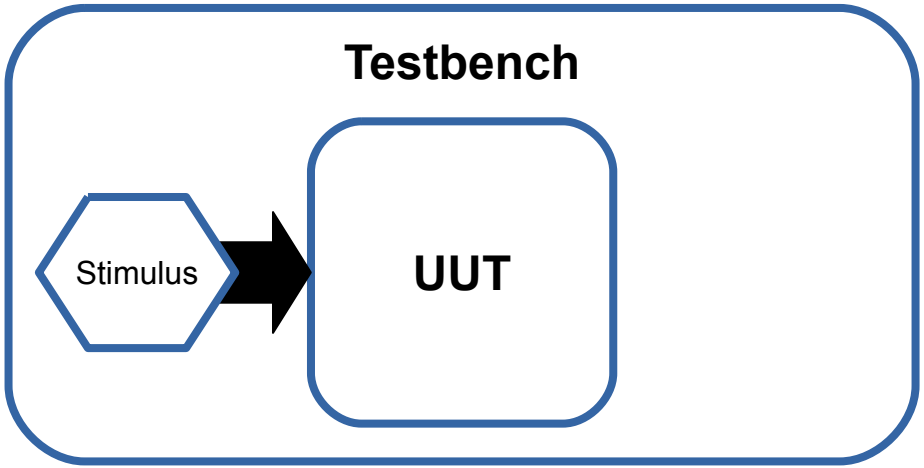
Testbench

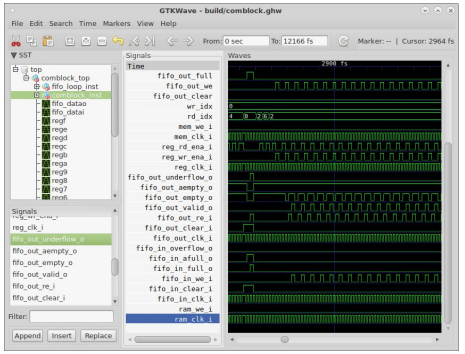


Testbench

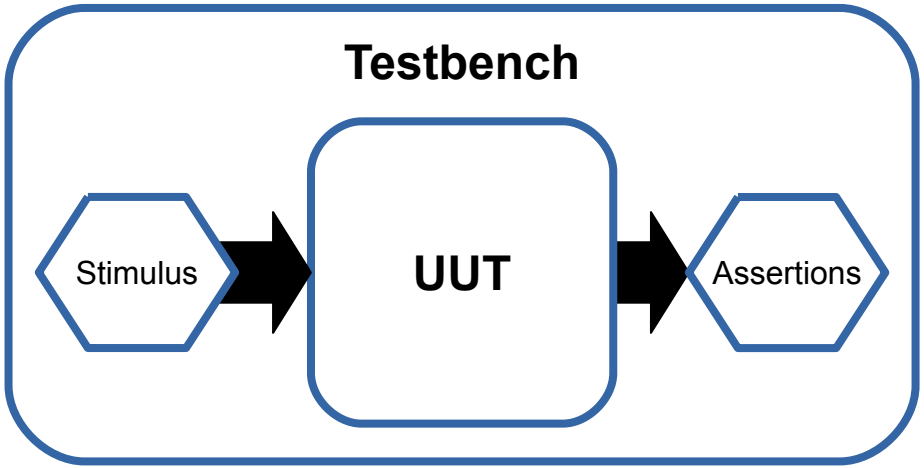


Testbench

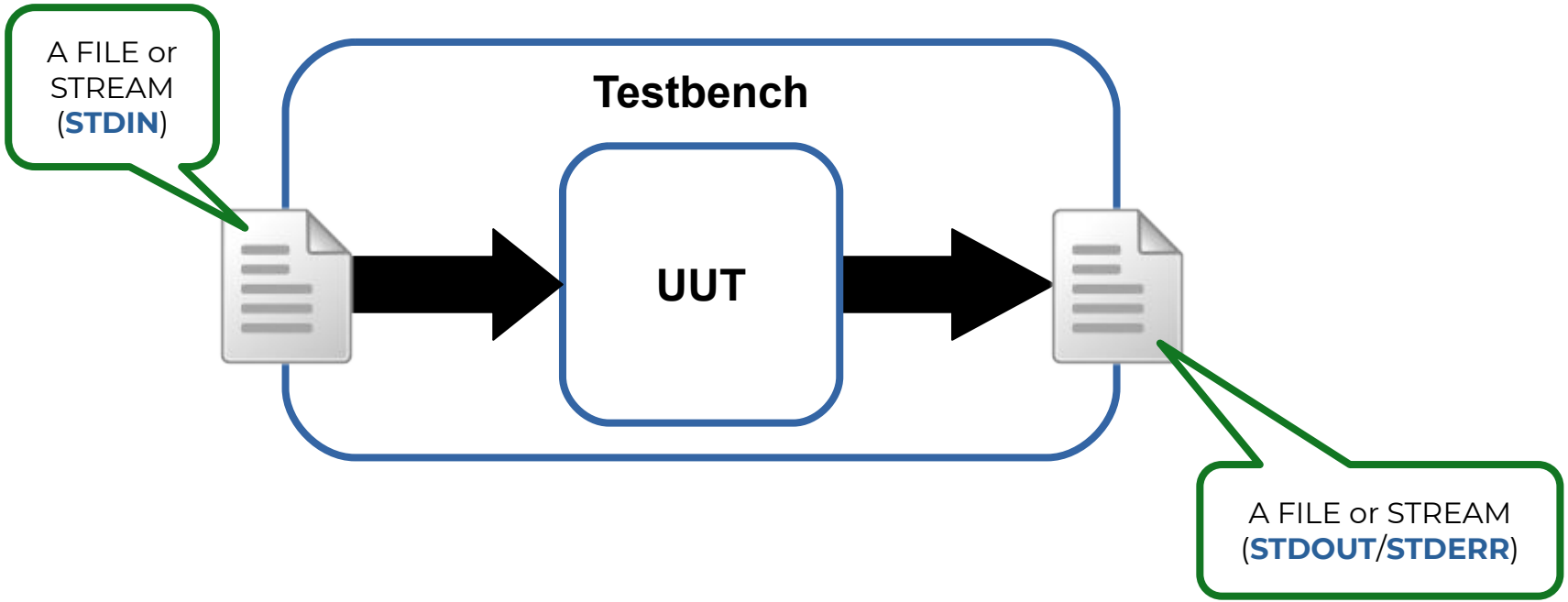

Testbench
Stimulus
UUT


GTKWave - build/comblock.ghw
File Edit Search Time Markers View Help
SST
top
comblock_top
fifo_loop_inst
fifo_datao
fifo_datai
regf
rege
regd
regc
regb
rega
reg9
reg8
reg7
reg6
Signals
reg_clk_i
fifo_out_underflow_o
fifo_out_aempty_o
fifo_out_empty_o
fifo_out_valid_o
fifo_out_re_i
fifo_out_clear_i
Filter:
Append
Insert
Replace
Time
fifo_out_full
fifo_out_we
fifo_out_clear
wr_idx
rd_idx
mem_we_i
mem_clk_i
reg_rd_ena_i
reg_wr_ena_i
reg_clk_i
fifo_out_underflow_o
fifo_out_aempty_o
fifo_out_empty_o
fifo_out_valid_o
fifo_out_re_i
fifo_out_clear_i
fifo_out_clk_i
fifo_in_overflow_o
fifo_in_afull_o
fifo_in_full_o
fifo_in_we_i
fifo_in_clear_i
fifo_in_clk_i
ram_we_i
ram_clk_i
Waves
2900 fs

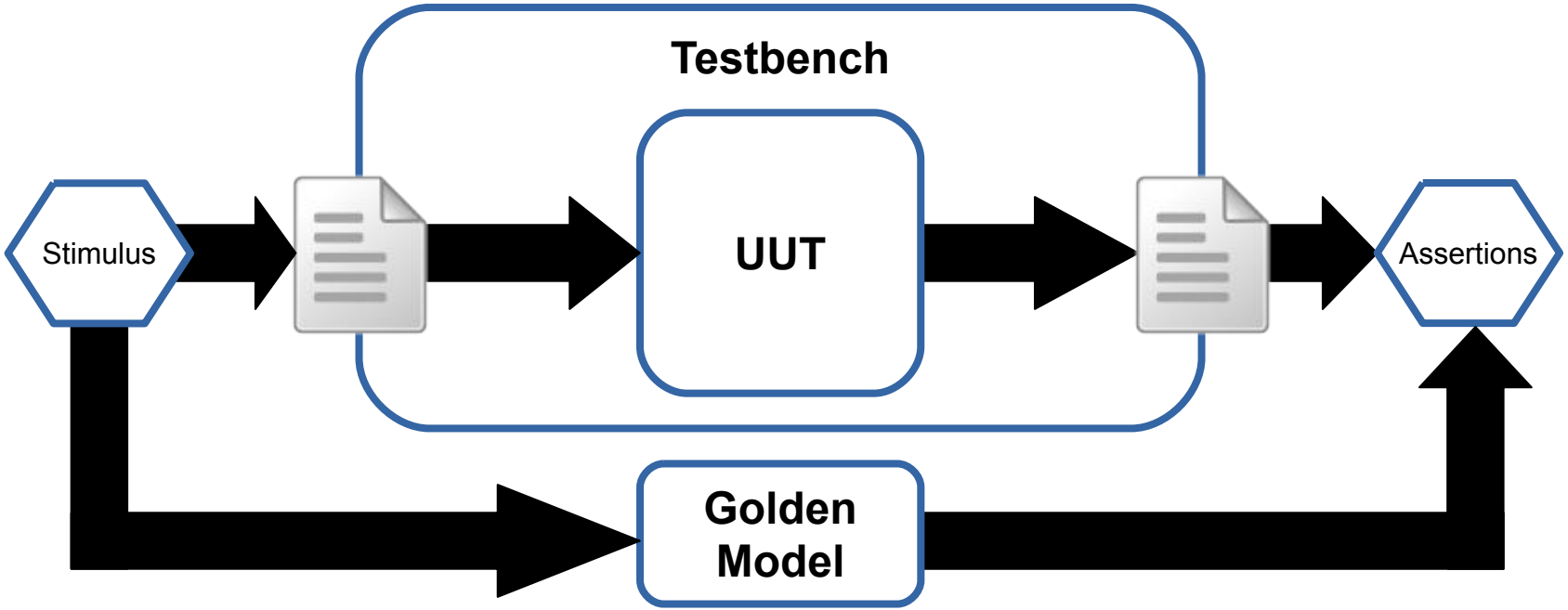
Testbench



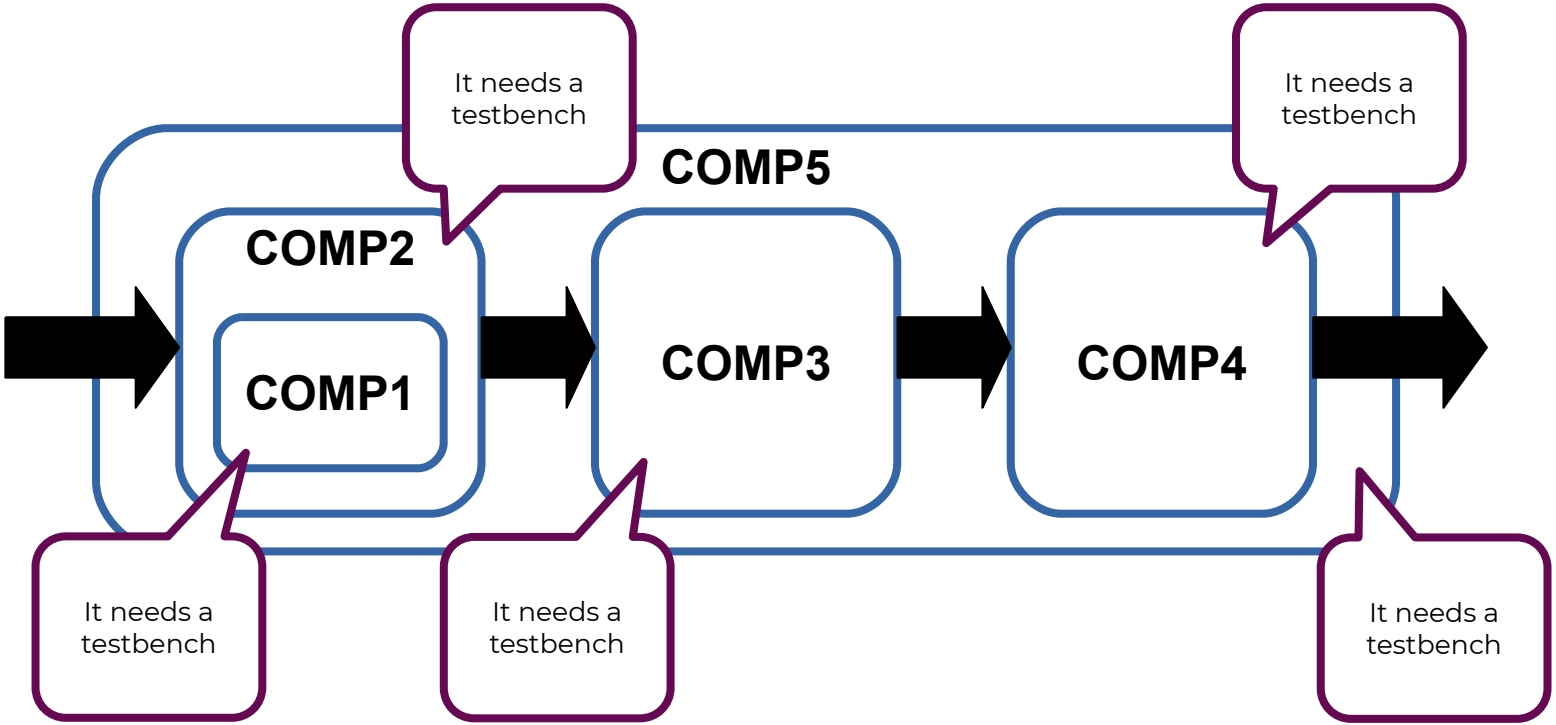
Testbench


A FILE or STREAM (STDIN)
Testbench
UUT
A FILE or STREAM (STDOUT/STDERR)

Testbench



To be clear


It needs a testbench
It needs a testbench
COMP5
COMP2
COMP1
COMP3
COMP4
It needs a testbench
It needs a testbench
It needs a testbench

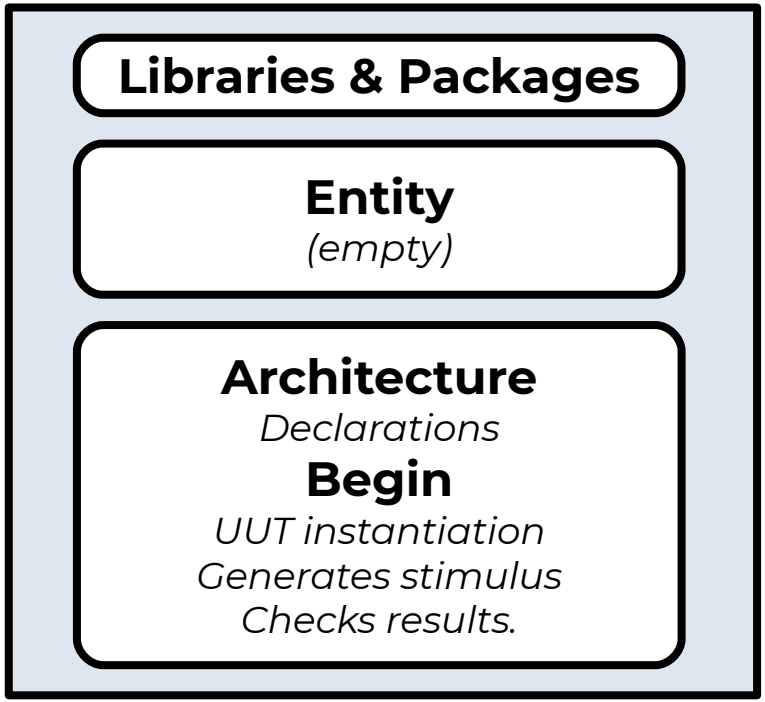
Frameworks and methodologies

- **UVM:** Universal Verification Methodology
  - Open Verification Methodology
  - Functional verification using SystemVerilog
  - Supported by a few very expensive simulators (and Vivado?)
- **FOSS:**
  - **OSVVM:** Open Source VHDL Verification Methodology
  - **UVVM:** Universal VHDL Verification Methodology
  - **VUnit:** unit testing framework for VHDL/SystemVerilog
  - **SVUnit:** unit testing framework for Verilog/SystemVerilog
  - **Cocotb:** Python based testbenches

But we will learn how to create a basic VHDL testbench from scratch.

VHDL for simulation

Structure of a VHDL testbench



- The **Entity** is employed only to define the name of the testbench.
- Only one **Architecture**.

Clock and reset generation

```
architecture Simul of counter_tb is
   constant PERIOD : time := 20 ns; -- 50 MHz
   signal clk      : std_logic:='1';
   signal rst      : std_logic;
   ...
   signal stop     : boolean := FALSE;
begin

   do_clock: process
   begin
      while not stop loop
        wait for PERIOD/2;
        clk <= not clk;
      end loop;
      wait; -- Event Starvation
   end process do_clock;

   rst <= '1', '0' after 3*PERIOD;
```

- time is a pre-defined physical type. It allows you to specify fs, ps, ns, us, ms, sec, min and hr.
- You need to produce **Event Starvation** (no more events) to finish the simulation. It is achieved by a **wait** without options.

Wait

```
wait [on signals] [until condition] [for time];
```

```
-- wait on signals;
wait on s1, s2; -- wait for an event
-- wait until condition;
wait until clk_i = '1'; -- wait for an event
-- wait for time;
wait for 10 ns;
-- wait;
wait; -- wait forever (event starvation)
```

- Is a sequential statement.
- Used by processes **without** a sensitivity list.

Assert and report

```
report <message_string> [severity <severity_level>];
assert <condition> [severity <severity_level>];
assert <condition> report <message_string> [severity <severity_level>];
```

- Report is a sequential statement (only inside of a process).
- The <severity_level> can be note (default for report), warning, error (default for assert) or failure.
- Assert can be either, a sequential or a concurrent statement.
- <condition> is a boolean value which must be TRUE to avoid the report.
- You can only report a string. The value of a **signal** or **variable** is not a string. You need to know the data **type** and use the image attribute:

```
report "unexpected value. i = " & integer'image(i);
```

Example - Component

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

entity counter is
    port (
        clk_i : in  std_logic;
        rst_i : in  std_logic;
        cnt_o : out std_logic_vector(3 downto 0)
    );
end entity counter;
```

```
architecture RTL of counter is
    constant MODULE : positive := 12;
    signal cnt      : unsigned(3 downto 0);
begin
    do_counter : process (clk_i)
    begin
        if rising_edge(clk_i) then
            if rst_i = '1' then
                cnt <= (others => '0');
            else
                if cnt < MODULE-1 then
                    cnt <= cnt + 1;
                else
                    cnt <= (others => '0');
                end if;
            end if;
        end if;
    end process do_counter;

    cnt_o <= std_logic_vector(cnt);
end architecture RTL;
```

Example - Testbench (part 1)

```vhdl
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
library std; -- Standard library of VHDL
use std.textio.all; -- we will write to STDOUT
library COUNTER_LIB; -- Our LIBRARY
use COUNTER_LIB.COUNTER_PKG.all; -- Our PACKAGE

entity counter_tb is
end entity counter_tb;

architecture Simul of counter_tb is
    constant PERIOD : time := 20 ns; -- 50 MHz
    signal clk      : std_logic:='1';
    signal rst      : std_logic;
    signal cnt      : std_logic_vector(3 downto 0);
    signal stop     : boolean := FALSE;
begin
```

```vhdl
    do_clock: process
    begin
        while not stop loop
            wait for PERIOD/2;
            clk <= not clk;
        end loop;
        wait; -- Event Starvation
    end process do_clock;

    rst <= '1', '0' after 3*PERIOD;

    -- UUT instantiation
    uut : counter
    port map(
        clk_i => clk,
        rst_i => rst,
        cnt_o => cnt
    );
```

Example - Testbench (part 2)

```
testing: process
    variable L : LINE; -- buffer
begin
    -- Print to STDOUT
    write(L,NOW); -- Current simulation time
    write(L,STRING'(" --> Start of test"));
    writeline(output,L); -- Write to STDOUT
    -- Test of the initial value
    wait until rising_edge(clk);
    wait until rst='0';
    assert unsigned(cnt)=0
        report "Error! not 0"
            severity failure;
    -- Test of the intermediate values
    for I in 0 to 11 loop
        wait until rising_edge(clk);
        assert unsigned(cnt)=I
            report "Error! cnt = ("&integer'image(to_integer(unsigned(cnt)))&")"
                severity failure;
    end loop;
    wait until rising_edge(clk);
```

Example - Testbench (part 3)

```
        -- Test cicle restart
        assert unsigned(cnt)=0
            report "Error! Mod-12 ("&integer'image(to_integer(unsigned(cnt)))&")"
                severity failure;
        -- Print to STDOUT
        write(L,NOW); -- Current simulation time
        write(l,string'("-> End of test"));
        writeline(output,L); -- Write to STDOUT
        -- Clock stop
        stop <= true;
        wait;
    end process testing;

end architecture Simul;
```

Files – std.textio

- Defines read and write procedures to work with FILES.
- The procedures supports the types bit, bit_vector, boolean, character (ex: ‘A’), string (ex: “ICTP”, defined from 1 **to** 4), integer, real and time.

```
procedure READLINE(file F: TEXT; L: out LINE);
procedure READ(L:inout LINE; VALUE: out <type>);
procedure READ(L:inout LINE; VALUE: out <type>; GOOD : out BOOLEAN);

procedure WRITE(
   L :inout LINE; VALUE : in <type>;
   JUSTIFIED: in SIDE := right;
   FIELD: in WIDTH := 0
);
procedure WRITELINE(file F : TEXT; L : inout LINE);
```

Files – read

```
stimulus: process
    file F       : TEXT open READ_MODE is "input.dat";
    variable L    : LINE;
    variable tag  : string(1 to 3);
    variable int  : integer;
    variable ok   : boolean;
begin
    while not endfile(F) loop
        readline(F, L); -- F can be replaced by input (read from STDIN)
        read(L, tag, ok);
        assert ok report "Read ERROR!" severity failure;
        -- Do something with tag (the read value)
        read(L, int, ok);
        assert ok report "Read ERROR!" severity failure;
        -- Do something with int (the read value)
    end loop;
    wait; -- event starvation
end process stimulus;
```

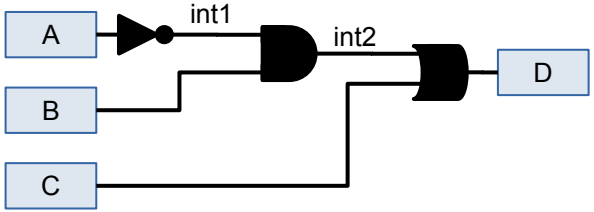
Files - write

```
checks: process
    file F: TEXT open WRITE_MODE is "output.dat";
    variable L: LINE;
begin
    ...
    WRITE(L, NOW);
    WRITE(L, STRING'("Your string")); -- This cast is needed for strings
    WRITELINE(F,L); -- F can be replaced by output (print to STDOUT)
    ...
    wait; -- event starvation
end process checks;
```

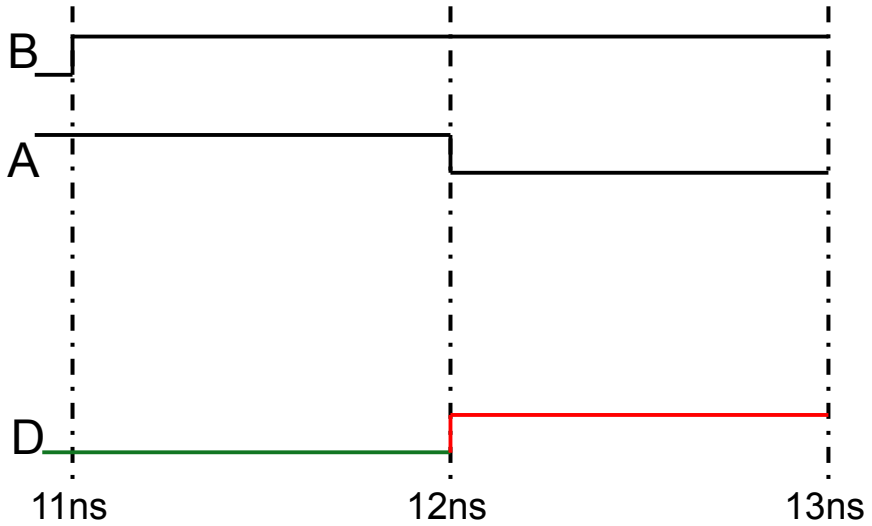
You can read from **STDIN** with a file called **input** and to write to **STDOUT** with a file called **output** (these files are automatically opened when you include **textio**).

Simulators

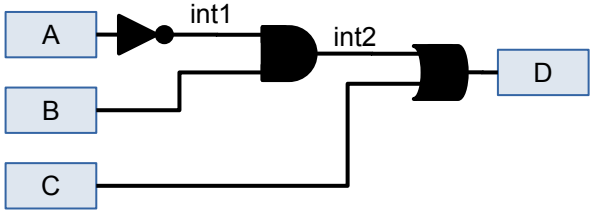
How a simulator works?



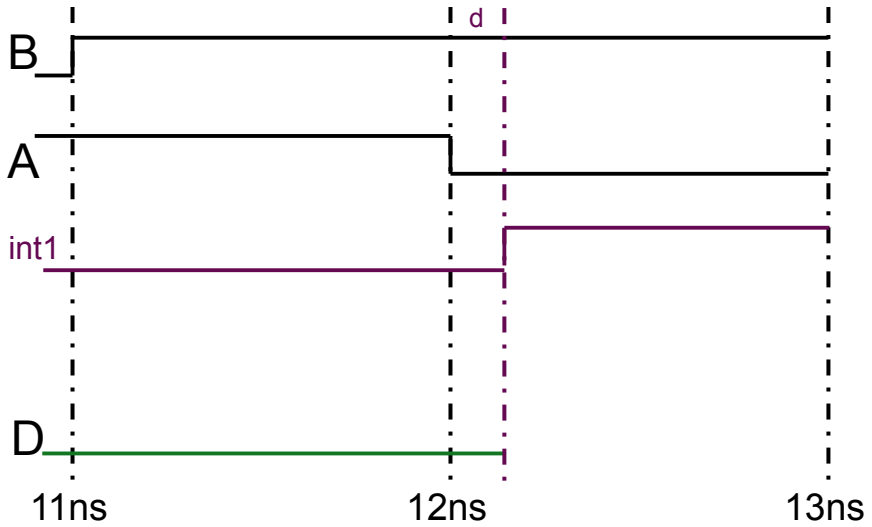

```
architecture rtl of example is
    signal int1, int2 : std_logic;
begin
    int1 <= not A;
    int2 <= int1 and B;
    D    <= int2 or C;
end architecture rtl;
```



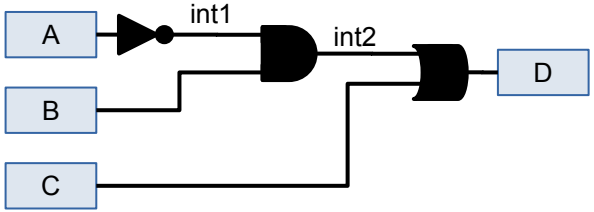
How a simulator works?



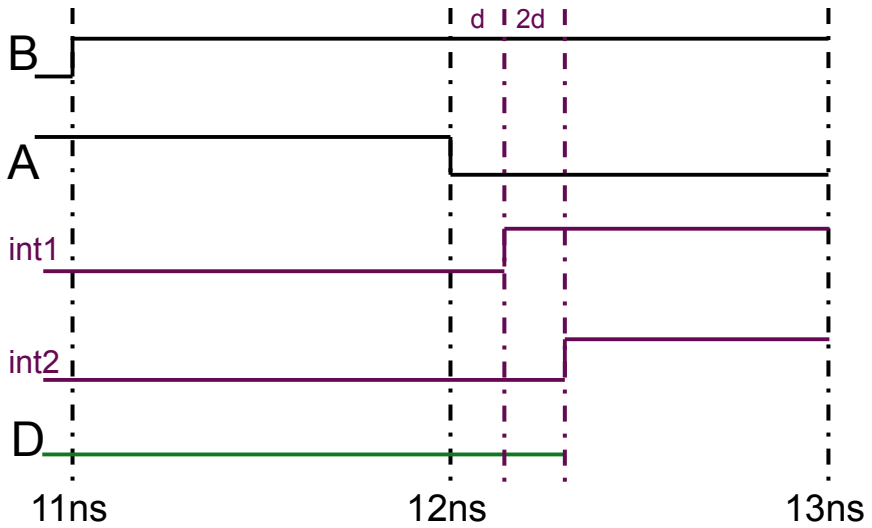

```
architecture rtl of example is
   signal int1, int2 : std_logic;
begin
   int1 <= not A;
   int2 <= int1 and B;
   D    <= int2 or C;
end architecture rtl;
```



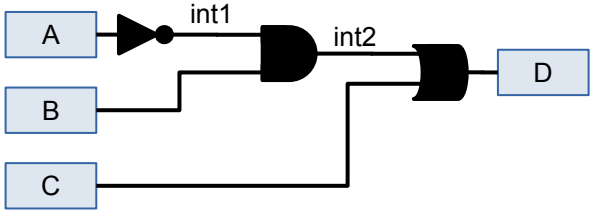
How a simulator works?



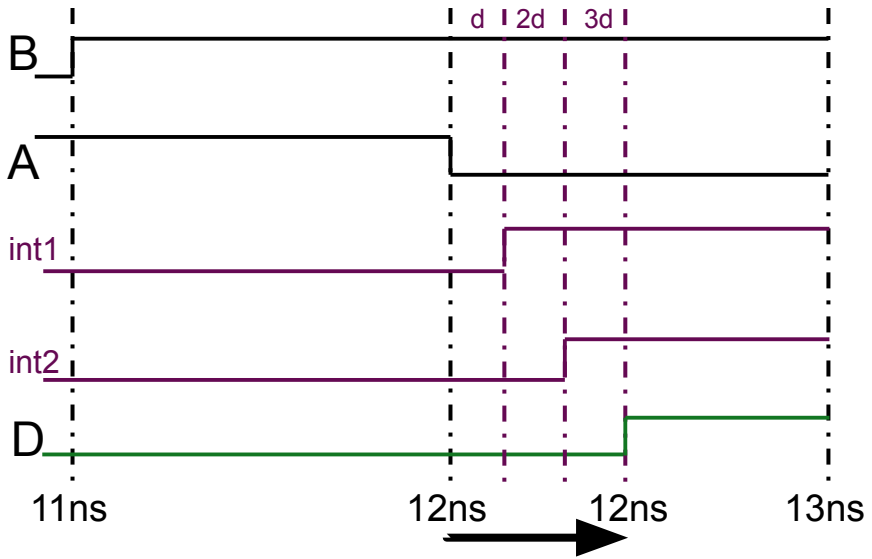

```
architecture rtl of example is
    signal int1, int2 : std_logic;
begin
    int1 <= not A;
    int2 <= int1 and B;
    D    <= int2 or C;
end architecture rtl;
```



How a simulator works?

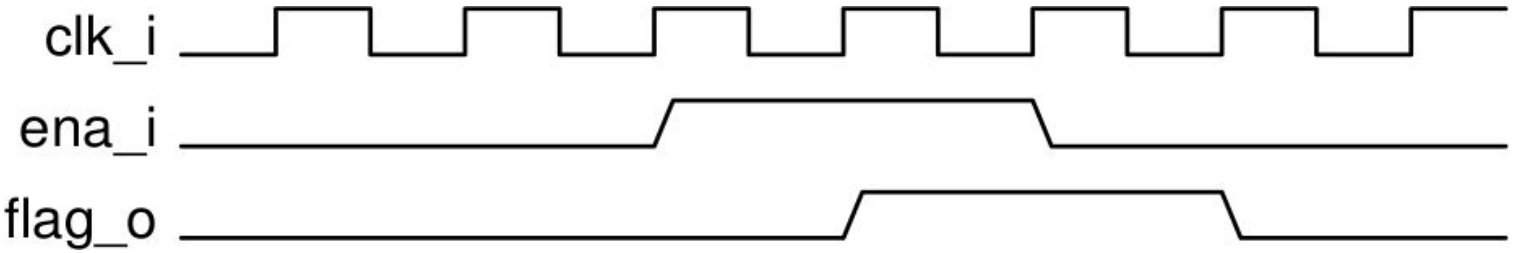



```
architecture rtl of example is
   signal int1, int2 : std_logic;
begin
   int1 <= not A;
   int2 <= int1 and B;
   D    <= int2 or C;
end architecture rtl;
```



Waveforms interpretation

```
if rising_edge(clk_i) then
    flag_o <= '0';
    if ena_i = '1' then
        flag_o <= '1';
    end if;
end if;
```



Example

- counter.vhdl: entity/component
- counter_pkg.vhdl: package which contains the component
  - The package name is COUNTER_PKG (name defined into the VHDL file)
- counter_tb.vhdl: testbench of the component
  - The library name is COUNTER_LIB (name defined by the tool)

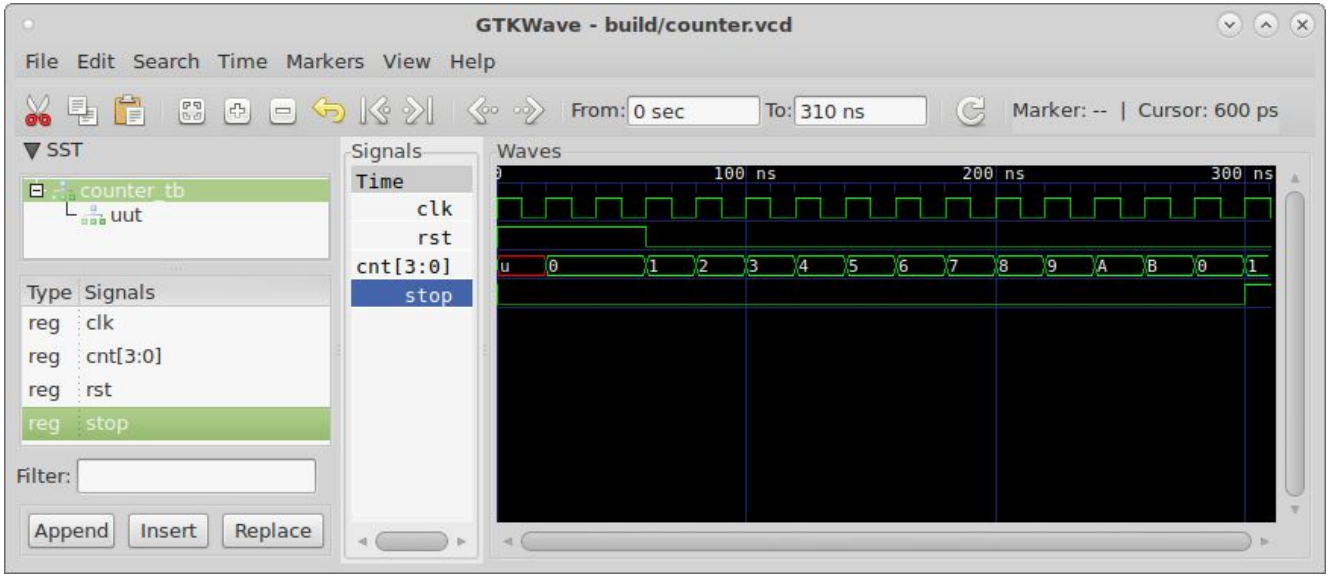
GHDL simulation

```
$ mkdir -p build
$ ghdl -a --workdir=build --work=COUNTER_LIB counter.vhdl counter_pkg.vhdl
$ ghdl -a --workdir=build -Pbuild counter_tb.vhdl
$ ghdl -e --workdir=build -Pbuild counter_tb
$ ghdl -r --workdir=build -Pbuild counter_tb --vcd=build/counter.vcd
```

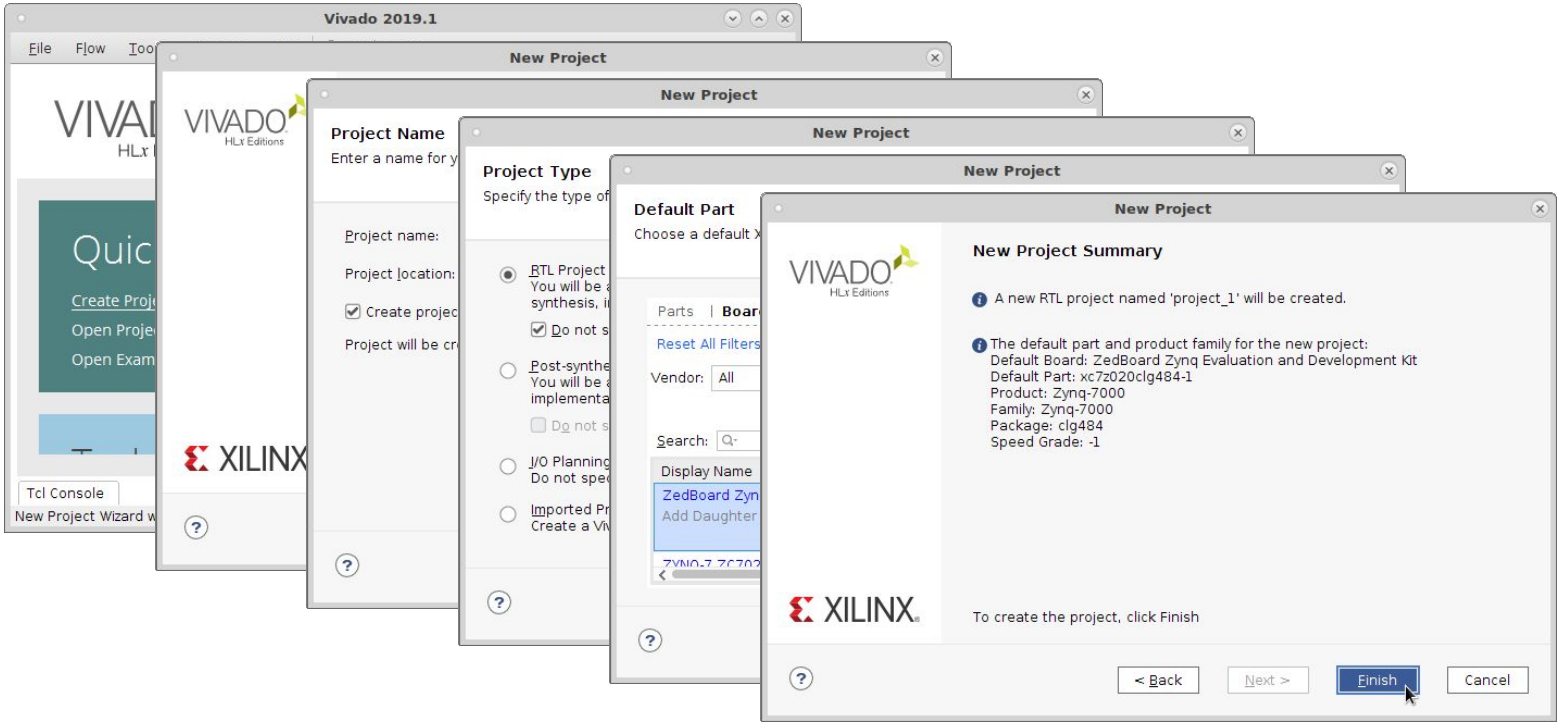
- GHDL analyze (-a), elaborate (-e) and run (-r) our simulation.
- Use --workdir to specify where to put generated files (build directory).
- Use --work to specify the LIBRARY NAME (COUNTER_LIB).
- Use -P to specify where to find libraries (no space between P and the directory).
- Use --vcd or --wave (.ghw), which are runtime options, to specify where to dump waveforms.

GTKwave waveform viewer

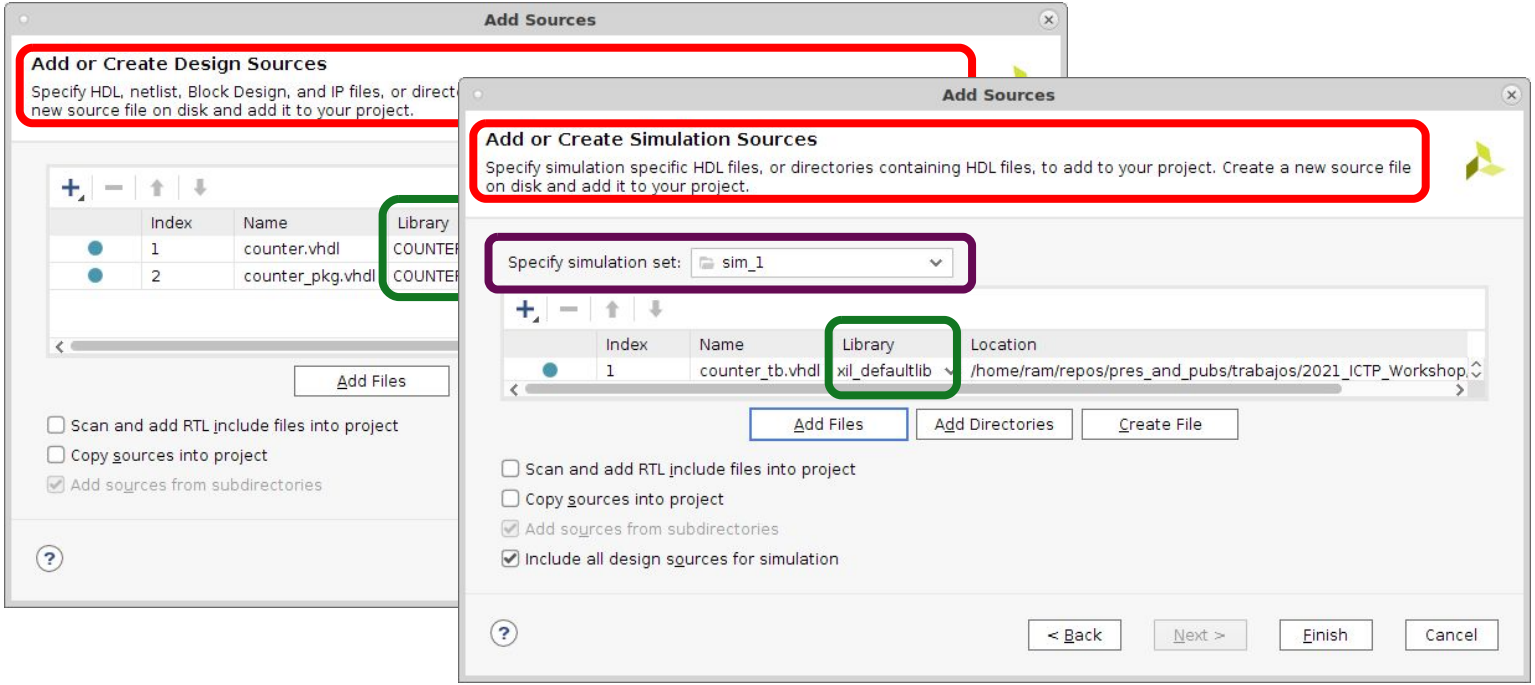
```
$ gtkwave build/counter.vcd
```



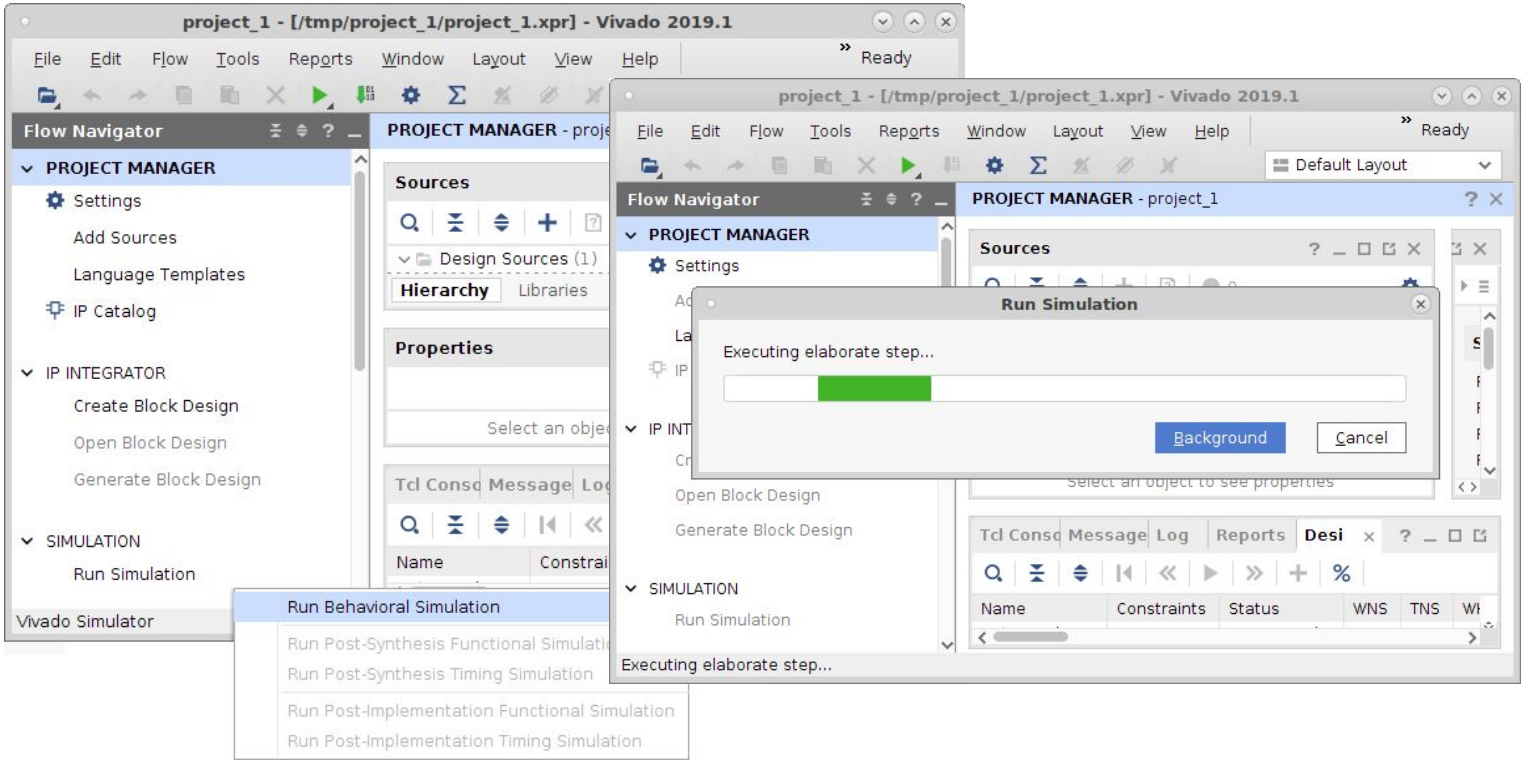
Vivado simulation - create project



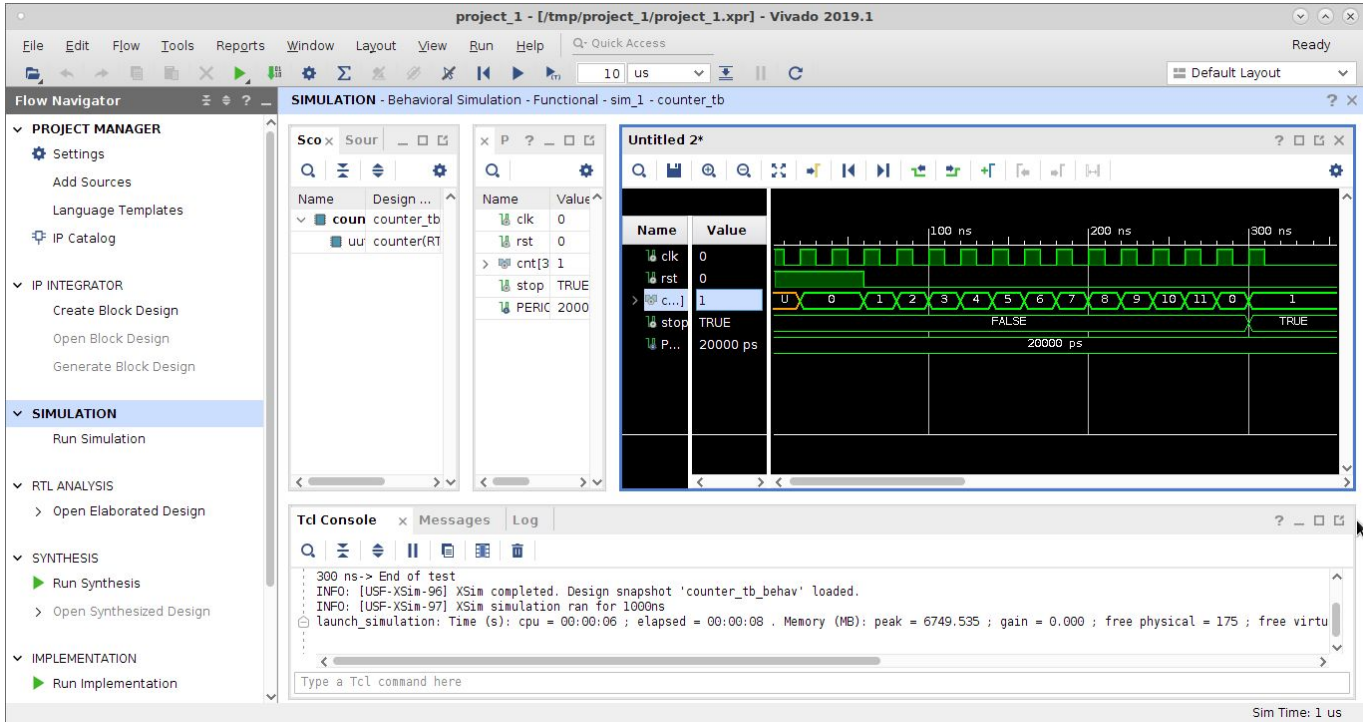
Vivado simulation - add sources



Vivado simulation – launch simulation



Vivado simulation – see waveforms



Conclusion

Final remarks

- Do not perform a testbench can be only allowed for very basic descriptions (a counter, a ROM) included in another simulated description. **Professional advice.**
- What we saw today is enough to develop a small testbench with stimulus and assertions.
- Also, you should be capable of read/write files.

rodrigomelo9@gmail.com

rodrigoalejandromelo

@rodrigomelo9ok

rodrigomelo9

rodrigomelo9