CPEG 422/622
EMBEDDED SYSTEMS DESIGN

Chengmo Yang

chengmo@udel.edu

Evans 201C

LECTURE 5
TESTBENCH

OUTLINE

- Last lecture:
  - 24-bit Adder/Subtractor
  - Overflow, CLA
- This lecture:
  - Testbench
  - Sequential circuit in VHDL

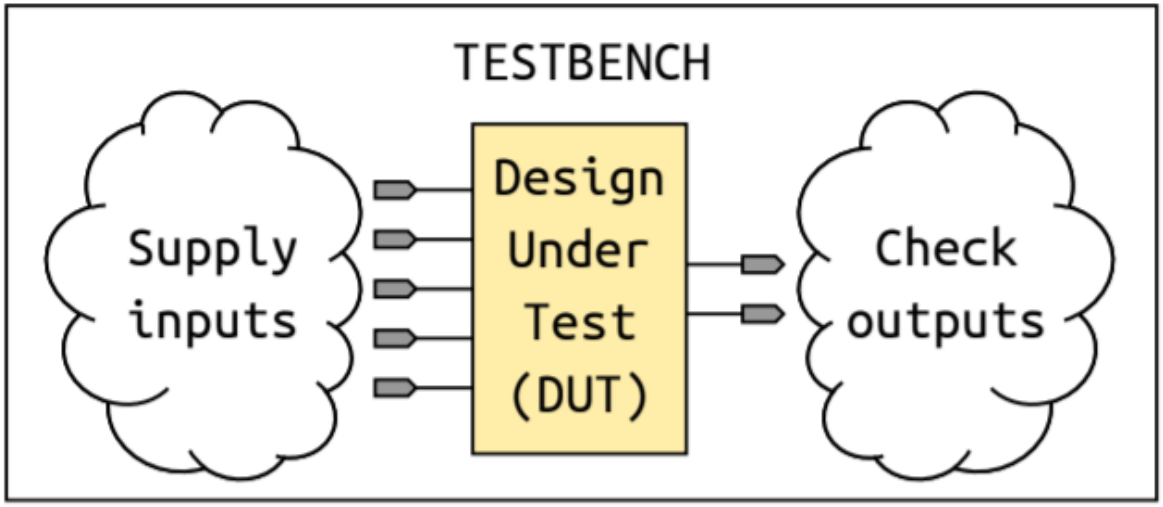
TESTBENCH

- Structure
- Testing
- Syntax

TESTBENCH

- Testbench: test code to verify your design correctness.
- Testbench has three parts:

1. The component we want to test, Design Under Test (DUT).
2. A mechanism for supplying inputs to the DUT.
3. A mechanism for checking the outputs of the DUT against expected outputs.

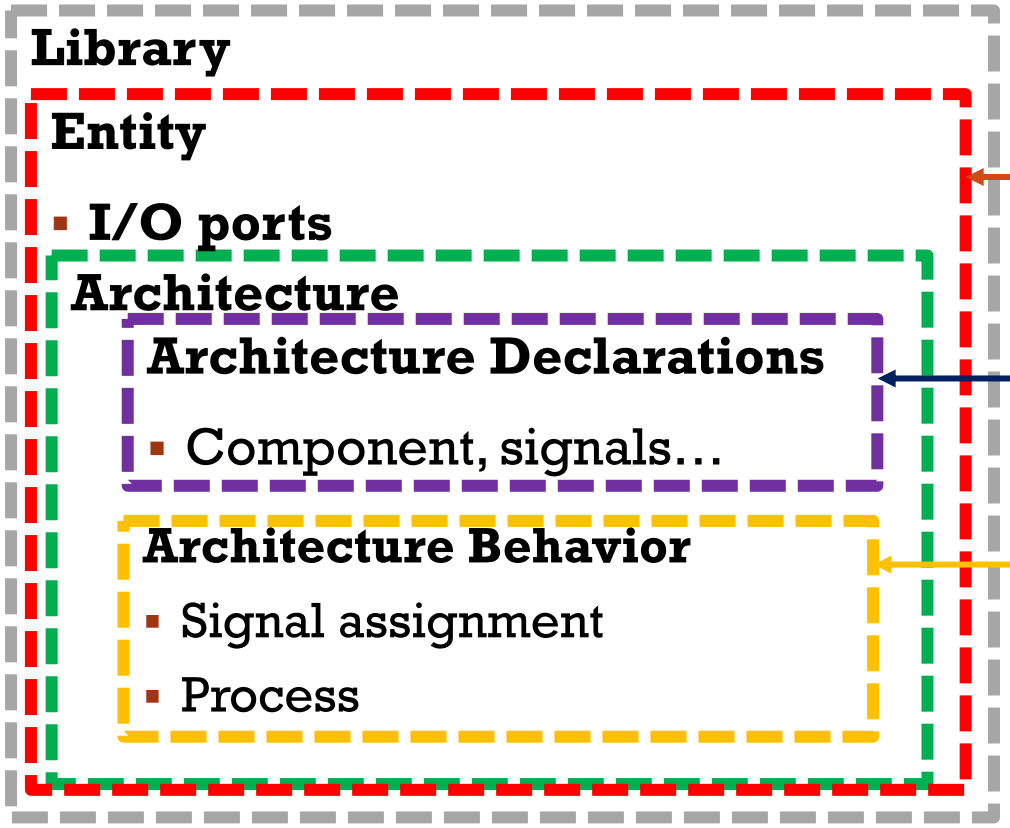


Treat DUT as a **black box,** give different combinations of inputs and observe the outputs.

TESTBENCH

Testbench structure is the same as design code, however:

Testbench differs in the following aspects:



Entity is empty, no need for IO

Component is DUT.

Behavior is for:

1. Set up and initialize DUT
2. Generate different input combinations.
3. Observe outputs.

TESTBENCH

Full adder testbench:

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

ENTITY Testbench_full_adder IS
END Testbench_full_adder;

ARCHITECTURE behavior OF Testbench_full_adder IS
 -- Component Declaration for the Unit Under Test (UUT)
  COMPONENT full_adder
  PORT(
       A : IN std_logic;
       B : IN std_logic;
       Cin : IN std_logic;
       S : OUT std_logic;
       Cout : OUT std_logic
   );
  END COMPONENT;
 --Inputs
  signal a,b,cin : std_logic := '0';
 --Outputs
  signal s,cout : std_logic;

BEGIN
 -- Instantiate the Unit Under Test (UUT)
  DUT: full_adder port map (a,b,cin,s,cout);

 -- Stimulus process
  stim_proc: process
  begin
  -- hold reset state for 100 ns.
  wait for 10 ns;
  -- insert stimulus here
  A <= '1';  B <= '0';Cin <= '0';
  wait for 10 ns;

  A <= '0'; B <= '1'; Cin <= '0';
  wait for 10 ns;

  A <= '1'; B <= '1'; Cin <= '0';
  wait for 10 ns;

  end process;
END behavior;
```

Entity is empty

Component full adder is DUT.

Declare and initialize signals, to be connected to DUT

Instantiate DUT

Simulation process, generate different test cases

VHDL SYNTAX RELATED TO TESTBENCH

- The previous testbench example has
  - Signal initialization
  - Process
  - Wait
- We will also introduce
  - Assert & Report
  - Loop

SIGNAL INITIALIZATION

- Initialize input signals when declaring them.
- **Syntax**: *signal <signal_name>: <type> := <value>;*
- **Example**:

```
signal A : std_logic_vector(3 downto 0) := (others => '0');
signal B : std_logic_vector(3 downto 0) := (others => '0');
signal Cin : std_logic := '0';
```

Std_logic_vector

Std_logic

- **Usage**: Initialize all inputs as 0.
- Signal initialization uses "**:=** " while signal assignment uses "**<=**".
- ***Q**: Where in the code structure is signal initialization and where is assignment?*

PROCESS

Syntax:

```
[<process_name>:] process (<sensitive signals>)
  variable declarations
  constant declarations
  ...
begin

  statements
  ...
end process;
```

- Process is used to describe event or signal triggered logic.
- Process has to include either a *sensitivity list* or *wait statements*, *but not both*.

Example 1: a 3-bit counter

Sensitivity list: this process is triggered when clock or reset changes its value.

```
architecture counter of counter is

signal temp: std_logic_vector(2 downto 0);

begin
   process(Clock,Reset)
   begin
      if Reset='1' then
         temp <= "000";
      elsif(rising_edge(Clock)) then
         if Enable='1' then
           if temp="111" then
              temp<="000";
           else
               temp <= temp + 1;
           end if;
         end if;
     end if;
   end process;

   Output <= temp;

end counter;
```

PROCESS

Syntax:

```
[<process_name>:] process (<sensitive signals>)
  variable declarations
  constant declarations
  ...
begin

  statements
  ...
end process;
```

- Process is used to describe event or signal triggered logic.
- Process has to include either a *sensitivity list* or *wait statements*, *but not both*.

Example 2: adder testbench

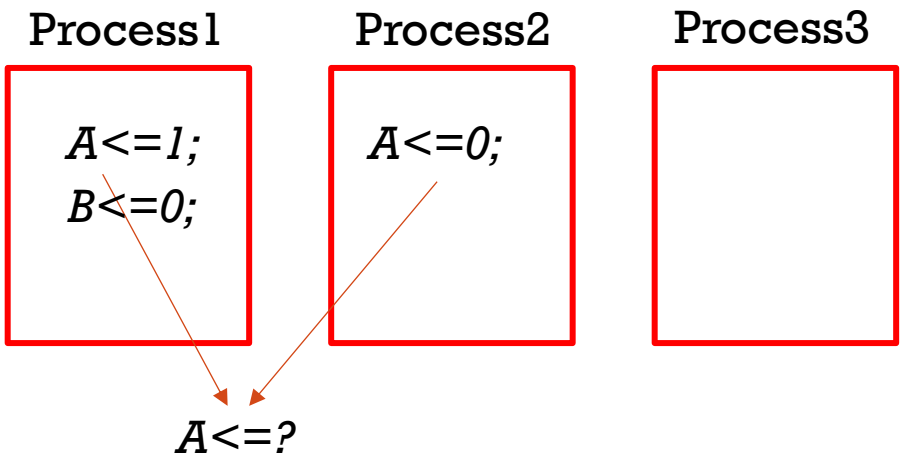
A process with wait statements.

```
BEGIN
  -- Instantiate the Unit Under Test (UUT)
  DUT: full_adder port map (a, b, cin, s, cout);

  -- Stimulus process
  stim_proc: process
  begin
  -- hold reset state for 100 ns.
  wait for 10 ns;
  -- insert stimulus here
  A <= '1';  B <= '0';Cin <= '0';
  wait for 10 ns;

  A <= '0'; B <= '1'; Cin <= '0';
  wait for 10 ns;

  A <= '1'; B <= '1'; Cin <= '0';
  wait for 10 ns;

  end process;
END behavior;
```

PROCESS

In one architecture, there can be multiple processes.

- Statements in a process are executed **sequentially**.
- All processes are executed **concurrently**.
- All assignments of one signal should occur in one process, otherwise the value is likely unstable.



WAIT

In a testbench, you probably want to assign different values to input signals.

Q:How to hold the input signals?

A: Use ***wait.***

Wait statement is used for timing control in process.

There are usually four types of usage:

- **wait**;
- **wait for** time;
- **wait on** signal_list;
- **wait until** condition;

WAIT

- **wait**;

  Wait for infinite time length, stuck at the point.

- **wait for** time;

  Wait for fixed time length.

Example:

```
clk: process
begin
    clock1 <= '0'; wait for 5ns;
    clock1 <= '1'; wait for 5ns;
end process;
```

Process will keep repeating

```
setup: process
   begin
  --setup
    reload1 <= '1';
    en1 <= '0';
    seed_a<="0000000000000001";
    seed_b<="0000000001010101";
    wait for 10ns;

    --working mode
    reload1 <= '0';
    en1 <= '1';
    wait;
end process;
```

Process only execute once

WAIT

- **wait on** signal_list;

Suspend on the wait statement until a change of signal in the list.

Example:

```
signal S1, S2 : Std_Logic;
. . .
process
  begin
    . . .
   wait on S1, S2;
end process;
```

Process will resume if S1 or S2 changes

- **wait until** condition;

Suspend on the wait statement until the condition becomes true.

Example:

```
wait until Enable = '1';
-- this is equivalent to
--   loop
--     wait on Enable;
--     exit when Enable = '1';
--   end loop;
```

Process will resume if enable=1

ASSERT & REPORT

Q: Once the input signal are set, how to check the correctness of outputs automatically?

A: Use ***assert*** and ***report***.

- **Assert** tests if the values match what we expect, in most cases used with the **report** statement to give the user an ERROR or WARNING.
- **Syntax**: ***assert*** *<condition>*

  report *<string> [**severity** type]*

- Predefined severity names are: NOTE, WARNING, ERROR, FAILURE
- Default severity for **assert** is ERROR
- Default severity for **report** (without assert) is NOTE

ASSERT & REPORT

Q: Once the input signal are set, how to check the correctness of outputs automatically?

A: Use ***assert*** and ***report***.

- **Example**:

```
A <= '0';
B <= '1';
wait for 10ns;
assert S = '1' report "Error, expected 1 for S" severity ERROR;
```

- **Usage**: check if S equals 1. If not, report an error on console.
- **More examples:**
  - `assert j<i report "internal error, tell someone";`
  - `assert clk='1' report "clock not up" severity WARNING;`
  - `report "Inconsistent data." severity FAILURE;`
  - `report "finished pass1";  -- default severity name is NOTE`

OVERFLOW ASSERT

Example: A, B are 4-bit vectors, check the first case.

Operation	Operand A	Operand B	Sum if overflow
A+B	>=0	>=0	<0
A+B	<0	<0	>=0
A-B	>=0	<0	<0
A-B	<0	>=0	>=0

```
if(A(3)='0' AND B(3)='0') then
        assert S(3)='0' report "error: sing bit is wrong, overflow" severity ERROR;
end if;
end process;
```

LOOP

Q: Suppose you have two 4-bit inputs, it is unrealistic to write all 256 test cases by hand. How to test them automatically?

A: Use a ***loop***.

- Same as other languages, a loop in VHDL is used to repeatedly execute a sequence of **sequential statements**.
- There are three types of loops:
  - basic **loop**
  - **while … loop**
  - **for … loop**
- All of them may contain
  - **next** – move to the next iteration
  - **exit** – terminate the loop

```
loop
  input_something;
  exit when end_file;
end loop;


for I in 1 to 10 loop
  AA(I) := 0;
end loop;


while not end_file loop
  input_something;
end loop;
```

FOR LOOP

- **Syntax**: *For <variable> in <starting> to <ending> loop*

  <statement>;

  End loop;

  use ieee.numeric_std.ALL; ***Library included***

- **Example:**

```
signal A : std_logic_vector(7 downto 0) := (others => '0');

for i in 0 to 255 loop
    WAIT FOR 1ns;
    A <= std_logic_vector(to_unsigned(i, A'length));
END LOOP;
```

- **Usage:** increase the value of A [0-255] by 1 every ns.
- **Note:** the index "i" is automatically declared by the loop, *no need to declare it separately*. The index can only be *read (not written!) inside the loop* and is not available outside its loop.

TYPE CONVERSION

Q: Why we need the following statement to assign a value to A?

```
A <= std_logic_vector(to_unsigned(i, A'length));
```

- VHDL is a **strongly typed language.**
- One has always to declare the type of every object , such as signals, constants, and variables.
- One cannot assign a value of one type to an object of another type.

  `A <= i;` -- this is wrong because A is a std_logic_vector but i is an integer .

- Assignment between different types of objects has to do type conversion: the syntax is as follows:

```
type_name (expression);
```

- More examples on type conversion can be found here: https://www.nandland.com/vhdl/tips/tip-convert-numeric-std-logic-vector-to-integer.html
- A'**length** is bit width of A. The attributes of different objects can be found here: https://www.csee.umbc.edu/portal/help/VHDL/attribute.html

NESTED LOOP

- For multiple input combinations, we use nested loop to go through all possible combinations.
- **Example: A,B are 4-bit signals.**

```
for i in 0 to 15 loop
        A <= std_logic_vector(to_unsigned(i, A'length));
    for j in 0 to 15 loop
                B <= std_logic_vector(to_unsigned(j, B'length));
                Cin<='0';
                wait for 10ns;
                assert S=(i+j)mod 16 report  "Error:S is wrong" severity ERROR;
                Cin<='1';
                wait for 10ns;
                assert S=(i+j+1)mod 16 report  "Error:S is wrong" severity ERROR;
               wait for 10ns;
    end loop;
end loop;
```

- **Usage:** Every time increase A by 1, B goes from 0 to 15.

  If Output S is not as expected, report an error.

NOTE

- WAIT, ASSERT, REPORT statements **can not be synthesized** in you design files, they can be only used in the testbench for simulation.

- If you observe UUUU (uninitialized) or XXXX (unknown) in the waveform for certain output or signal, check if:
  1. your input ports are all initialized.
  2. your design logic has any problem.

- Simulation results guarantee that the design is functionally correct at the behavior level. But sometimes the design behavior may be different when running on real board.

In the previous nested loop example, two level loops involve $2^4 \times 2^4$ iterations, each iteration takes 30ns.

Total test time is $2^4 \times 2^4 \times 30 = 7680$ ns

What if the A and B are 24-bit signals?

$2^{24} \times 2^{24} \times 30 \approx 8000{,}000$ s, too slow !

Q: How to generate test cases with a good coverage?

A: Use random test.

RANDOM NUMBER GENERATION

- Use uniform() function.
- **Example:**

use ieee.math_real.all

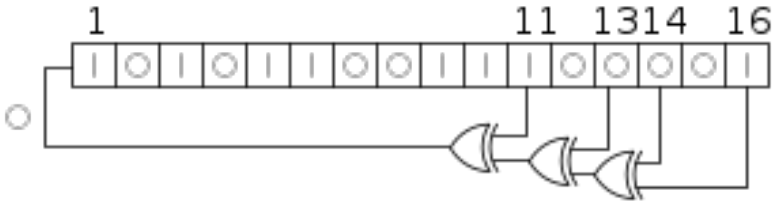
```
process
    variable seed1, seed2: positive;                -- seed values for random generator
    variable rand: real;    -- random real-number value in range 0 to 1.0
    variable range_of_rand : real := 1000.0;      -- the range of random values created will be 0 to +1000.
begin
    uniform(seed1, seed2, rand);    -- generate random number
    rand_num <= integer(rand*range_of_rand);  -- rescale to 0..1000, convert integer part
    wait for 10 ns;
end process;
```

- **Usage:** use seed1 and seed2 to generate random number between 0 to 1000.

- ***However*** *vivado does not support real library any more after 2015.* ***Cannot use uniform in vivado.***

RANDOM NUMBER GENERATION

- Currently a common approach for generating pseudo random numbers uses a **linear-feedback shift register** (**LFSR**) implementation in Vivado.
- It is a **sequential** circuit implementation. We will provide you its VHDL code for testing your 24-bit adders.



LFSR

LFSR CONTROL

```
entity LFSR is
    Port (
        clock :    in STD_LOGIC;
        reload:    in STD_LOGIC;
            D :    in STD_LOGIC_VECTOR (15 downto 0);
           en :    in STD_LOGIC;
            Q :    out STD_LOGIC_VECTOR (15 downto 0)
       );
end LFSR;
```

Two working modes:

- Initialization – this loads a seed to the LFSR;

```
reload <=1, en <=0, D <= a non-zero seed value;
```

- Running – the LFSR generates a new value every clock cycle;

```
reload <=0, en <=1;
```

Note that the LFSR needs a clock all the time!

A LITTLE MORE ABOUT TESTBENCH

- Add/subtractor are **combinational** logic:

1. Their logic circuits are "stateless": The output only depends on inputs.
2. But to test the add/subtractor automatically, we would like to change its inputs periodically, thus our testbench will be a sequential circuit driven by a clock signal.

For the adder/subtractor test, we will introduce:

- Clock generation in testbench
- Random number generation

TESTBENCH-CLOCK GENERATION

- To generate clock signal in testbench, you need a ***process***.

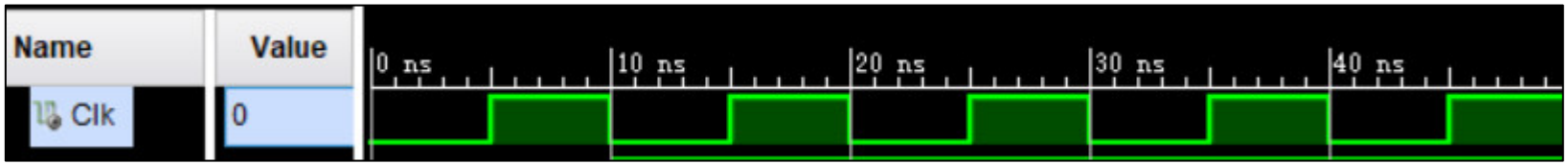
```
signal Clk : std_logic := '0';
constant clk_period : time := 10 ns;
```

1.Declare clock signal and period constant.

```
clk_process :process
   begin
        Clk <= '0';
        wait for clk_period/2;  --for 5 ns signal is '0'.
        Clk <= '1';
        wait for clk_period/2;  --for next 5 ns signal is '1'.
   end process;
```

2.Define a clock generation process.

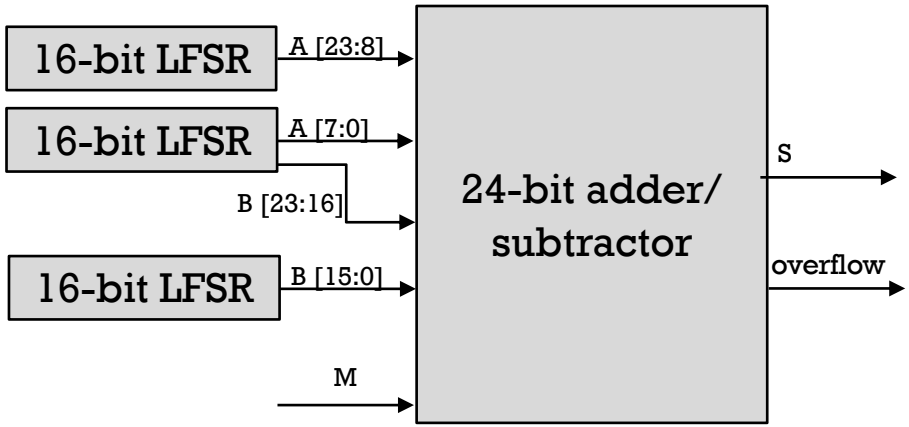
3. Get a clock signal with a period of 10ns



TESTBENCH-RANDOM NUMBER

- To generate random test cases for inputs A & B, a 16-bit linear-feedback shift register (LFSR) code is offered.

1. Add the LFSR file in your design sources.
2. Change the testbench as follows:
   - Add LFSR as a component
   - Instantiate three 16-bit LFSRs
     - One for A [23:8], one for A [7:0] and B [23:16], one for B [15:0]
   - Connect LFSR outputs to adder inputs



TESTBENCH-RANDOM NUMBER

- To generate random test cases for inputs A & B, a 16-bit linear-feedback shift register (LFSR) code is offered.

1. Add the LFSR file in your design sources.
2. Change the testbench as follows:
   - Add LFSR as a component
   - Instantiate three 16-bit LFSRs
   - Connect LFSR outputs to adder inputs
   - Add connection signals for LFSRs

```
signal reload,clock,en: std_logic;
signal seed_a : std_logic_vector(15 downto 0):= (others => '0');
signal seed_b : std_logic_vector(15 downto 0):= (others => '0') ;
signal seed_c : std_logic_vector(15 downto 0):= (others => '0') ;
```

TESTBENCH-RANDOM NUMBER

- To generate random test cases for inputs A & B, a 16-bit linear-feedback shift register (LFSR) code is offered.

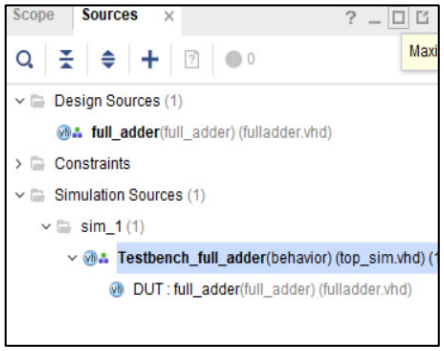
1. Add the LFSR file in your design sources.
2. Change the testbench as follows:
   - Add LFSR as a component
   - Instantiate three 16-bit LFSRs
   - Connect LFSR outputs to adder inputs
   - Add connection signals for LFSRs
   - Add a setup process, then A, B can be generated automatically

```
setup: process
   begin
  --setup
    reload <= '1';
    en <= '0';
    seed_a<=x"0505";
    seed_b<=x"7272";
    seed_c<=x"a1a1";
    wait for 10ns;

    --working mode
    reload <= '0';
    en <= '1';
    wait;    --infinite wait
  end process;
```

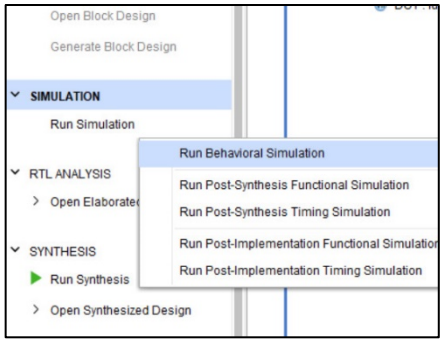
ADDER TESTBENCH STRUCTURE

- Your testbench should have three processes
  - One for generating clock signal
  - One for setting up the LFSR
  - One for checking sum and overflow
- Note that connection of LFSRs to adder and instantiation of LFSRs and Adder should be done outside any process

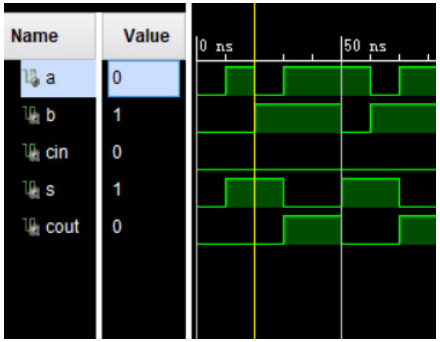
TO RUN THE TESTBENCH



1. Create a **simulation file** in your project and put your testbench inside.



2. Click run simulation on the control panel, such as choose "Run behavior simulation".



3. A waveform is automatically popped up, and you can check if the DUT performs as you expected.

NEXT LECTURE

- Generate design reports
- Read design reports