EE-334
Digital System Design

Custom Digital Circuits

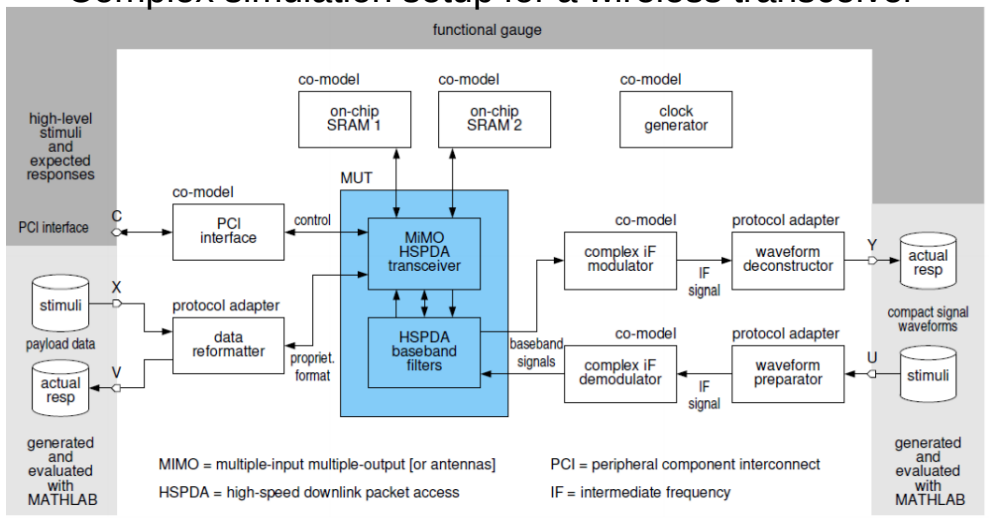
VHDL for Simulation –
Basic Constructs

Andreas Burg

Verification and Simulation

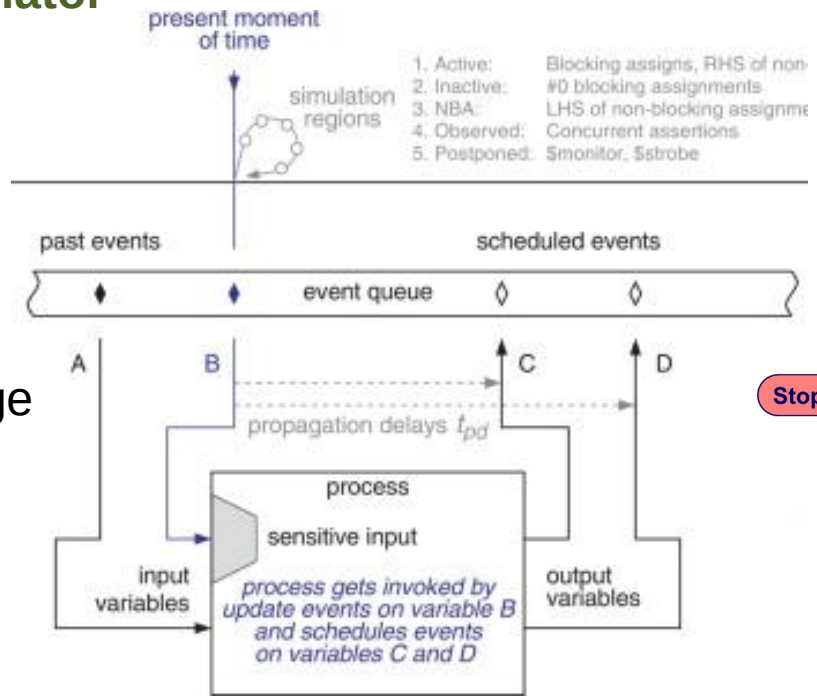
- HDL simulations mimic hardware behaviour
- We **simulate VHDL code to verify and debug** the code of **a Device Under Test**
- HDL **simulations involve three types of components**:
  - The **device under test (DUT)**
  - A **testbench** that
    - orchestrates the simulation
    - Provides stimuli to the DUT
    - Checks the correctness of the behaviour (outputs)
  - **Models of other system components** that interact with the DUT

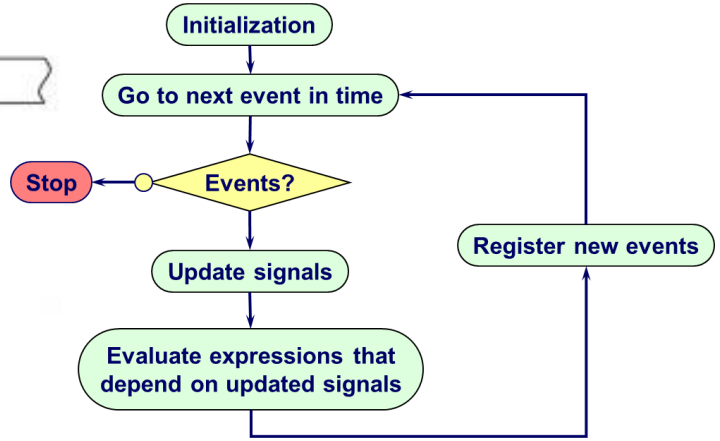
Complex simulation setup for a wireless transceiver



Event Based Simulation of VHDL

- How to simulate parallel hardware on a computer?
- **HDL simulation follows** an **event based execution** model
  - Key element is the **event queue: ordered list** of entries that mark **signal changes over time**
  - **Signal assignments** that lead to a change in a signal **deposit an event in the event queue**
    - **Signal changes only when the simulator proceeds to the scheduled event**
  - **Three stage execution:**
    1. **Advance simulation time** to the next event
    2. **Update the signal values** that change with the currently processed event
    3. **Process** (execute) **all statements that are sensitive to this event**





Event Based Simulation of VHDL

- **Simulation time advances while progressing through the event queue**
- **Often, VHDL assignments do not have any delay**
  - When to schedule the signal change in the event queue?

- **If no delay**/time **is specified** for an assignment, the **simulator assigns a δ-Delay**
  - A δ-Delay has no time duration, but allows to schedule events one-after-another

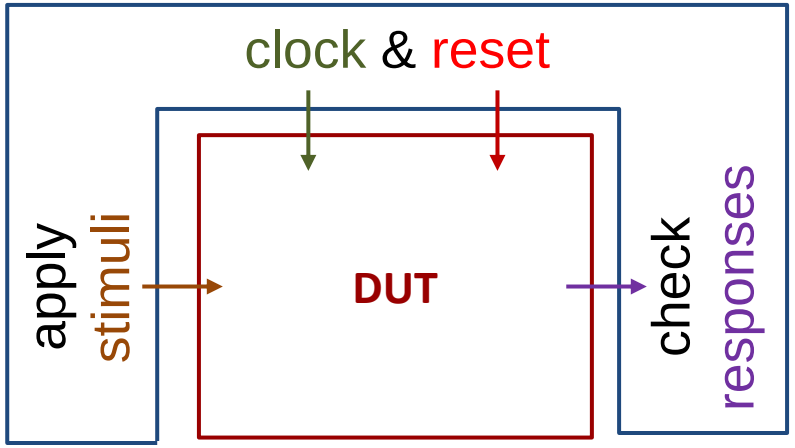
```
-- Async Reset Generation
p_rst : PROCESS (INxS, AxS)
BEGIN
  BxS <= AxS;
  AxS <= INxS;
  CxS <= AxS;
END PROCESS p_rst;
```

	T=0ns	T=10ns			T=20ns		
			$+1\delta$	$+2\delta$		$+1\delta$	$+2\delta$
INxS	0	**1**	1	1	**0**	0	0
AxS	0	0	**1**	1	1	**0**	0
BxS	0	0	0	**1**	1	1	**0**
CxS	0	0	0	**1**	1	1	**0**

What is a Testbench?

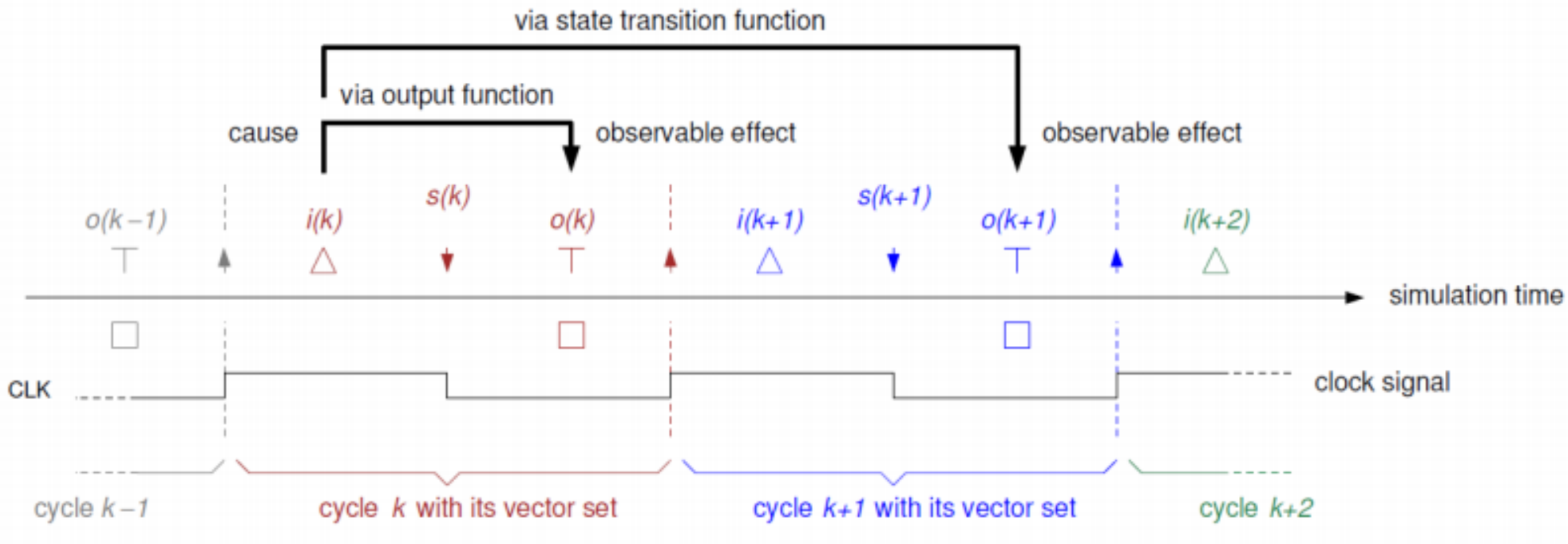
- **A testbench is a test harness** for the purpose of simulating a **DUT**
  - **Testbenches** themselves **are** often **written in an HDL** (e.g., VHDL)
  - The **testbench is the top-level entity for simulation**

- A **basic testbench** performs the following tasks:
  - Instantiation of the **DUT**
  - Generation of the Clock
  - Asynchronous (power-up) Reset
  - Application of stimuli (DUT inputs)
  - Checking of responses (DUT outputs)



The Simulation Schedule for Synchronous Circuits

- In (positive edge triggered) synchronous systems, **stimuli application and response acquisition follows a specific** cycle-by-cycle **schedule** and timing
  - Each cycle starts with the positive clock edge
  - **Stimuli are applied in the beginning of each cycle (Δ)**, after a fixed, short delay
  - **Responses are checked at the end of each cycle (T)**, a fixed, short delay before the next edge



Simulation Strategies: Cycle-by-Cycle

- **Define cycle accurate stimuli inputs and expected outputs** obtained
  - from a cycle-accurate golden model of the circuit
  - manually by interpreting the design specification
- Stimuli and expected outputs can be stored
  - As part of the code of the testbench OR
  - In files that are read by the testbench

```
p_tb : PROCESS
BEGIN
  -- CYCLE 1
  …

  -- CYCLE 2
  …
END PROCESS p_tb;
```

Cycle	INPUT	OUTPUT
1	0100010	01010
2	0101110	01001
3	0100011	01011
…		

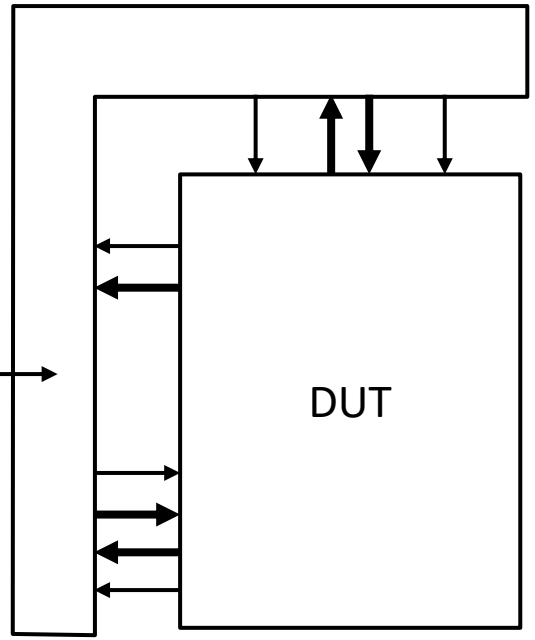
- **Very simple testbench proceeds cycle-by-cycle**
  - Applies the stimuli to the DUT (in the beginning of every cycle)
  - Checks the expected outputs (before the end of every cycle)

- This approach is **only convenient for very small blocks** since
  - ✘ Cycle accurate golden models are usually not available
  - ✘ Manual design of stimuli and expected responses is very tedious
  - ✘ Changes in the timing (not the function) of the design requires re-design of the stimuli/testbench

Cycle-Accurate vs. …

Cycle	INPUT	OUTPUT
1	0100010	01010
2	0101110	01001
3	0100011	01011
…		

```
p_ALL : PROCESS (  )
BEGIN
  -- APPLY STIMULI
  -- CHECK RESPONSES
  -- GO to NEXT CYCLE
END PROCESS p_ALL;
```



Simulation Strategies: Non-Cycle-Accurate

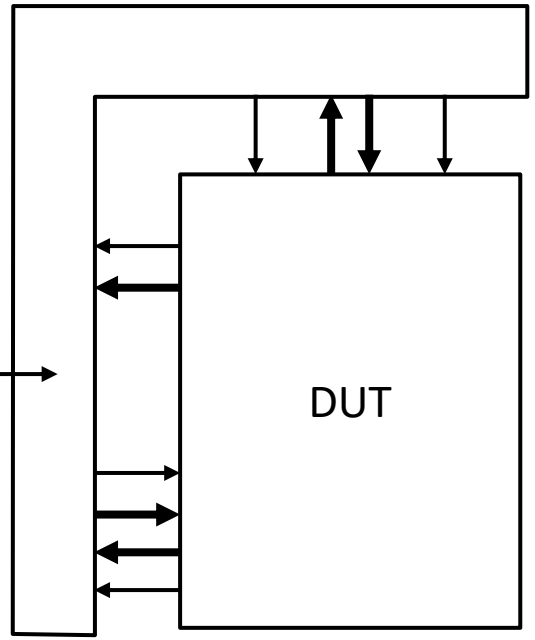
- **Define only the functional stimuli and responses**, but not their exact timing
  - Manually based on the specifications
  - Through a purely behavioural model
- Stimuli and expected outputs can be stored
  - As part of the code of the testbench OR
  - In files that are read by the testbench
- **Complex testbench handles the cycle-by-cycle interaction with the DUT**
  - Applies the stimuli to the and checks the expected outputs
  - Reacts to control outputs of the DUT and generates by itself all control inputs to the DUT

- This approach is **much more flexible and easy for complex DUTs**
  - Focuses on the functionality
  - Does not require anticipation of the exact timing of inputs/outputs

Cycle-Accurate vs. Non-Cycle-Accurate

Cycle	INPUT	OUTPUT
1	0100010	01010
2	0101110	01001
3	0100011	01011
…		

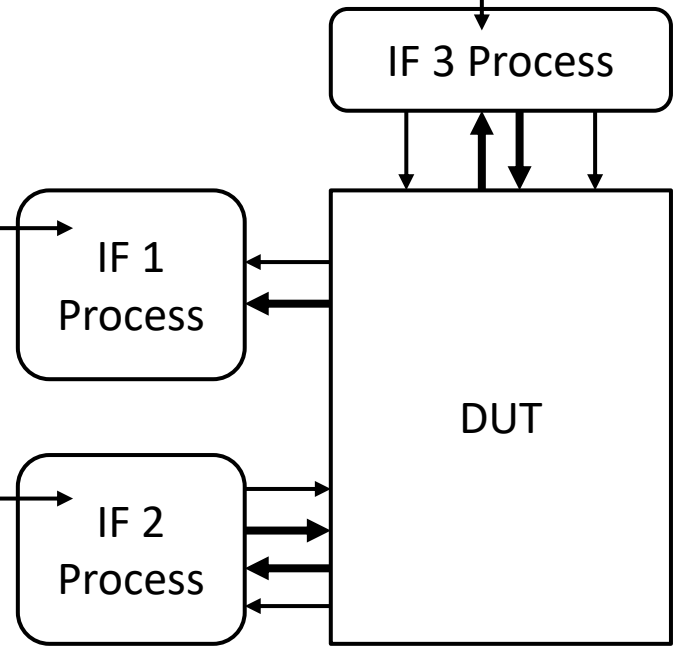
```
p_ALL : PROCESS (  )
BEGIN
  -- APPLY STIMULI
  -- CHECK RESPONSES
  -- GO to NEXT CYCLE
END PROCESS p_ALL;
```




```
p_IF1 : PROCESS (  )
BEGIN
  -- BEHAVIORAL CODE
END PROCESS p_IF1;
```

```
p_IF2 : PROCESS (  )
BEGIN
  -- BEHAVIORAL CODE
END PROCESS p_IF2;
```

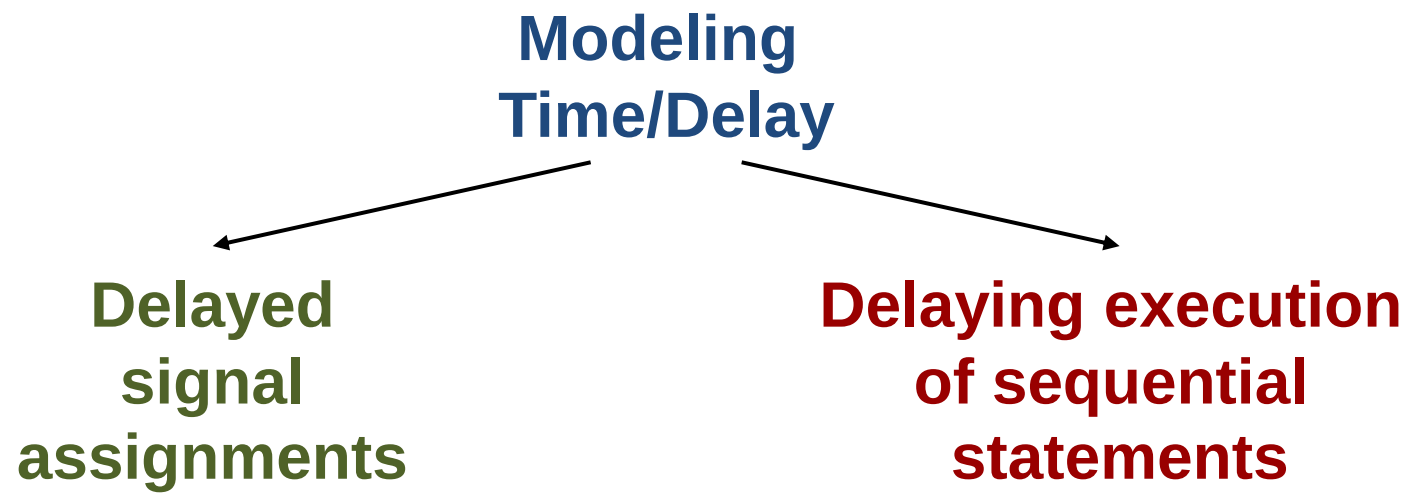
```
p_IF3 : PROCESS (  )
BEGIN
  -- BEHAVIORAL CODE
END PROCESS p_IF3;
```



VHDL for Simulations: Introducing Time

- RTL design descriptions for synthesis are discrete time models
  - Concept of time (in seconds) is not supported by synthesis
- However, **for simulation, we need to model time and delays**
  - **Examples:** generate a clock signal with a given period or model circuit delays

- **VHDL offers two main options to introduce time and delays**



Delayed Signal Assignments in VHDL

- Signal **assignments normally happen without a physical delay**

- **Assignments can be delayed** artificially using

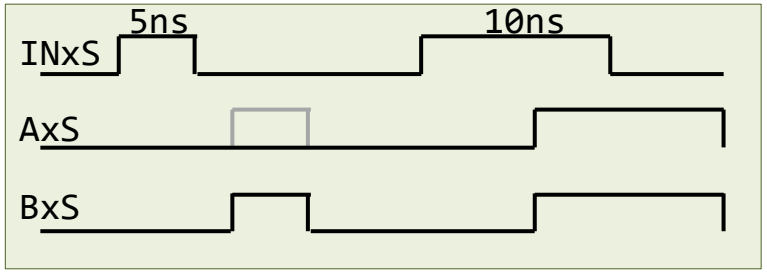
  - **An inertial delay model**: pulses shorter than the delay are swallowed
    - **Time** is specified in s, ms, us, ns, ps

```
signal <= expression AFTER time;
```

  - **A transport delay model**: all signal changes are retained and propagated after the given delay

```
signal <= TRANSPORT expression AFTER time;
```

```
AxS <= INxS AFTER 7.5ns;
BxS <= TRANSPORT INxS AFTER 7.5ns;
```



Delaying Sequential Statement Execution

- **Sequential statements** in a process are **carried out in zero** physical **time**

- **Execution of a process can be suspended** with a **WAIT-statement** to

  - Wait for any change in any of the signals in a list of signals

```
WAIT ON signal_1, signal_2, … ;
```

  - Wait for a given boolean condition to be fulfilled

```
WAIT UNTIL boolean_expression ;
```

  - Wait for a specified amount of time

```
WAIT FOR waiting_time ;
```

  - Wait forever

```
WAIT ;
```

Checking for Expected Responses

- **VHDL provides dedicated statements to check conditions and issue** different levels of warning or error **messages in the simulator console**
  - Used often to compare signals against expected responses
  - Can also be used to check complex conditions, computed within a testbench

- The **ASSERT-statement**

```
ASSERT boolean_expression REPORT string SEVERITY severity_level;
```

  - checks if a **boolean-expression and if** the condition evaluates to **FALSE**
  - outputs a **REPORT string** to the console of the simulator
  - with one of four **SEVERITY** levels**: NOTE, WARNING, ERROR or FAILURE**
    (FAILURE normally aborts the simulation)

Testbench: Creating Clock and Reset

- **CLOCK signal**: periodic and runs forever
  - Best practice: define clock parameters as constants
  - Realized with a process and wait statements
  - Exploits that once a process without sensitivity list ends it is called again
- **RESET signal**: asserted once in the beginning of the simulation
  - Realized with a process and wait statements
  - Reset removal aligned with the CLOCK

```
CONSTANT CLK_PERIOD  : time := 10ns;
CONSTANT CLK_HIGH    : time := CLK_PERIOD / 2;
CONSTANT CLK_LOW     : time := CLK_PERIOD / 2;
…
BEGIN -- architecture
          -- Clock Generation
          p_clk : PROCESS
          BEGIN
            CLKxC <= '0';
            WAIT FOR CLK_LOW;
            CLKxC <= '1';
            WAIT FOR CLK_HIGH;
          END PROCESS p_clk;
```

```
          -- Async Reset Generation
          p_rst : PROCESS
          BEGIN
            RSTxRB <= '0';
            WAIT UNTIL CLKxC'EVENT and CLKxC='1';
            WAIT UNTIL CLKxC'EVENT and CLKxC='1';
            WAIT FOR 1ns;
            RSTxRB <= '1';
            WAIT;
          END PROCESS p_rst;
```

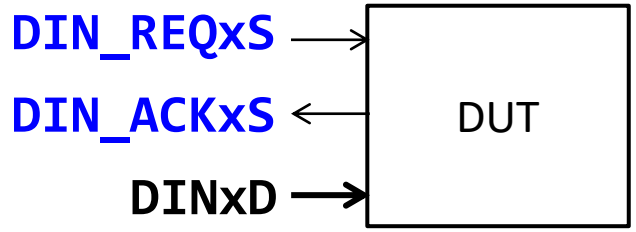
Applying Stimuli & Checking Responses

- Stimuli application and response acquisition can be done in
  - **a common process** if both steps are tightly related with short response latency
  - **independent processes** when latencies are large and timing of inputs and outputs is decoupled
- Example for hard-coded stimuli and responses:

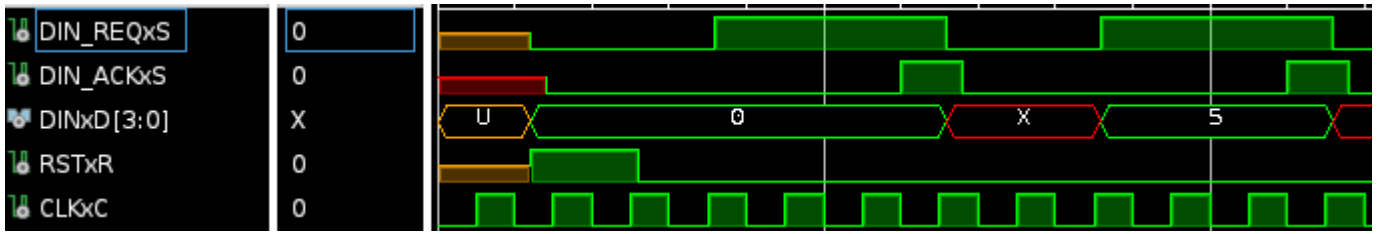
```
-- Stimuli Application
p_stim : PROCESS
BEGIN
  INPUTxS <= '0';  -- Initial Input during Reset

  WAIT UNTIL CLKxC'EVENT and CLKxC='1' and RSTxRB = '1';
  WAIT FOR STIM_APPL_DELAY;
  INPUTxS <= '1';  -- First cycle

  WAIT UNTIL CLKxC'EVENT and CLKxC='1';
  WAIT FOR STIM_APPL_DELAY;
  INPUTxS <= '0';  -- Second cycle

  WAIT;
END PROCESS p_stim;
```

```
-- Response Checking
p_resp : PROCESS
BEGIN
  WAIT UNTIL CLKxC'EVENT and CLKxC='1' and RSTxRB = '1';
  WAIT FOR RESP_CHK_DELAY;
  ASSERT OUTxS = exp_response_1 REPORT "Mismatch 1"
         SEVERITY FAILURE;

  WAIT UNTIL CLKxC'EVENT and CLKxC='1';
  WAIT FOR RESP_CHK_DELAY;
  ASSERT OUTxS = exp_response_2 REPORT "Mismatch 2"
         SEVERITY FAILURE;

  WAIT;
END PROCESS p_stim;
```

“Interactive” Stimuli Application



- Consider a simple REQ / ACK protocol at the DUT input
  - REQ signal (provided by the TB) indicates that input is ready
  - ACK signal from the DUT indicates that data was accepted

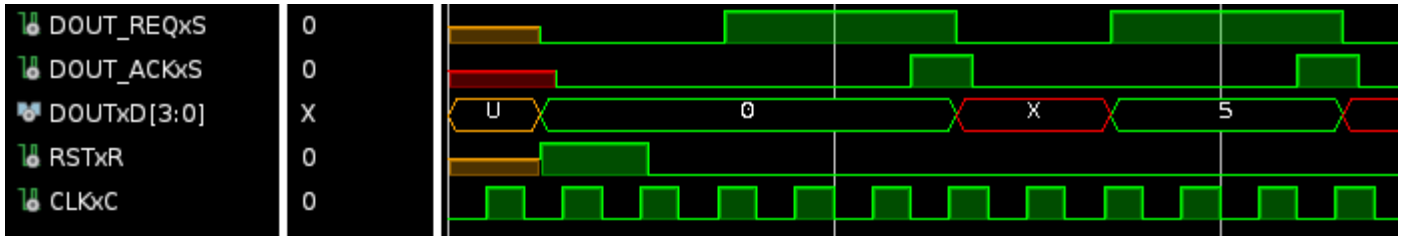



```
p_DIN: PROCESS IS
BEGIN  -- PROCESS p_DIN
    WAIT UNTIL RSTxR='1’;
    DINxD <= "0000";
    DIN_REQxS <= ‘0’;

    WAIT UNTIL CLKxC'event AND CLKxC=‘1’ AND RSTxR='0’;
    WAIT FOR CLK_STIM;
    DINxD <= "0000";
    DIN_REQxS <= ‘1’;

    WAIT UNTIL CLKxC'event AND CLKxC=‘1’ AND DIN_ACKxS='1’;
    WAIT FOR CLK_STIM;
    DIN_REQxS <= '0’;
    DINxD <= "XXXX";
```

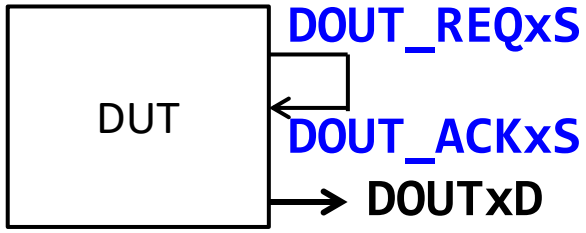
```
    WAIT UNTIL CLKxC'event AND CLKxC='1’;
    WAIT UNTIL CLKxC'event AND CLKxC='1’;
    WAIT FOR CLK_STIM;
    DIN_REQxS <= '1’;
    DINxD <= "0101";

    WAIT UNTIL CLKxC'event AND CLKxC='1’
         AND DIN_ACKxS='1’;
    WAIT FOR CLK_STIM;
    DIN_REQxS <= '0’;
    DINxD <= "XXXX";

    WAIT;
END PROCESS p_DIN;
```

“Interactive” Response Acquisition/Check

- Consider a simple REQ / ACK protocol at a DUT input
  - REQ signal (provided by the TB) indicates that input is ready
  - ACK signal from the DUT indicates that data was accepted




```
p_DOUT_ACK: PROCESS (DOUT_REQxS) IS
BEGIN  -- PROCESS p_DIN
    IF DOUT_REQxS = '1' THEN
        DOUT_ACKxS <= '1' AFTER 24ns;
    ELSIF DOUT_REQxS = '0' THEN
         DOUT_ACKxS <= '0' AFTER 2ns;
    ELSE
         DOUT_ACKxS <= 'X';
    END IF;
END PROCESS p_DOUT_ACK;
```




```
p_DOUT: PROCESS IS
BEGIN  -- PROCESS p_DIN
    WAIT UNTIL RSTxR='1';

    WAIT UNTIL CLKxC'event AND CLKxC='1' AND RSTxR='0';
    WAIT UNTIL CLKxC'event AND CLKxC='1' AND DOUT_REQxS='1';
    ASSERT DOUTxD = "0000" REPORT "OK" SEVERITY warning;

    WAIT UNTIL CLKxC'event AND CLKxC='1' AND DOUT_REQxS='1';
    ASSERT DOUTxD = "0000" REPORT "OK" SEVERITY warning;

    WAIT;
END PROCESS p_DOUT;
```

How to Debug VHDL Code

- **To debug the DUT, we look at** the evolution of signals over time as **waveforms**
  - Consider not only the inputs and outputs, but also the internal signals of your code

Interpreting “std_logic” Signal Waveforms

- The type **“std_logic” supports values other than just ‘0’ and ‘1’** which provide useful hints to potential “issues” or “intents” in the code
  - **‘X’** indicates a signal that has conflicting assignments
  - ‘U’ indicates a signal that was never (in the entire simulation) assigned any value (appear sometimes in the beginning of a simulation in signals from the testbench)
    - Within a design, all ‘U’ should have disappeared after the asynchronous reset or after the first clock edge
  - ‘-’ is a signal that was assigned as “DON’T CARE”, i.e., it could be either ‘0’ or ‘1’ (this is usually done on purpose to give the synthesis tool more freedom to optimize)