

# Testbench Overview

- Testbenches can serve two purposes
  - Generate input stimulus
  - Check outputs for correct behavior (optional)

# Instantiate UUT/DUT

- The top level is a structural model that instantiates your design as a component
- You could have a separate component for the TB code (or two, one to generate stimulus and one to check outputs)
- Xilinx generates testbenches with TB code in the same architecture as the UUT

# Xilinx Support

- Create > Test Bench Waveform
  - Creates a VHDL representation of the waveform with the UUT instantiated (fid.vhw)
- Create > VHDL Test Bench
  - Creates VHDL Test Bench template with UUT instantiated, but no stimulus
- Can Generate a Self-Checking Test Bench
  - Simulates existing TB and generates a new TB

# VHDL Testbenches

- Uses constructs not normally used in coding for synthesis
  - Wait until <condition>
  - Wait for <time amount>
  - Delayed signal assignments
  - For loops
  - Assert statements
- Much closer to coding for behavior
- Example: Memory model

# Testbench Example

- Test receive portion of a UART (serial)
- Inputs
  - Clock that is 8x faster than symbol (Baud) rate
  - Serial input data (including start, parity, stop)
  - Synchronous reset
- Outputs
  - Received data
  - Valid (used as load enable to read data)
  - Error signal indicating parity or framing error

# Unit Under Test (UUT)

```
-- Component Declaration for the Unit Under Test
(UUT)
COMPONENT univ_recvr
PORT(
    clk : IN std_logic;
    rst : IN std_logic;
    rxd : IN std_logic;
    dout : OUT std_logic_vector(7 downto 0);
    valid : OUT std_logic;
    err : OUT std_logic
);
END COMPONENT;
```

# Constants

```
-- Constants
```

```
CONSTANT sclk : integer := 9600;  
CONSTANT scc  : time   := 1 sec/sclk;  
CONSTANT cc   : time   := scc/8;  
  
CONSTANT idle : std_logic := '0';  
CONSTANT start : std_logic := '0';  
CONSTANT stop  : std_logic := '1';
```

Symbol clock frequency (sclk) sets Baud rate.

Other times derived from symbol clock period (scc).

# clock & reset

```
--Inputs
```

```
SIGNAL clk : std_logic := '0';
```

```
SIGNAL rst : std_logic := '0';
```

```
-- Set up clock (endless simulation!!)
```

```
clk <= not clk after 0.5 * cc;
```

```
-- Set up rst
```

```
rst <= '1' after 4.75 * cc,
```

```
      '0' after 5.75 * cc;
```

# Generate Data Process

```
tb : PROCESS
variable data : std_logic_vector(7 downto 0);
variable parity : std_logic;
BEGIN

-- Wait 100 ns for global reset to finish
wait for 100 ns;

-- wait for reset to complete

wait until rising_edge(clk) and rst='1';
wait for 4*cc;

wait for 0.25*cc; -- 1/4 cycle after rising edge
```

# Send One Byte

```
data := X"37"; -- hexadecimal format  
parity := '0';
```

```
for i in 7 downto 0 loop  
    parity := parity xor data(i);  
end loop;
```

```
rxd <= start; wait for scc;
```

```
for i in 0 to 7 loop  
    rxd <= data(i); wait for scc;  
end loop;
```

```
rxd <= parity; wait for scc;  
rxd <= stop; wait for scc;
```

# End Simulation

```
data := X"5A";
parity := '0';
for i in 7 downto 0 loop
    parity := parity xor data(i);
end loop;
parity := not parity; -- induce parity error
rxd <= start; wait for scc;
for i in 0 to 7 loop
    rxd <= data(i); wait for scc;
end loop;
rxd <= parity; wait for scc;
rxd <= stop; wait for scc;

report "Sim done" severity FAILURE;

wait; -- will wait forever
END PROCESS;
```

# Check Data

```
dout0 : process
subtype word is integer range 0 to 255; -- constrained int
type dout_array is array (natural range <>) of word;
variable dout_vals : dout_array(0 to 4) :=
    (16#37#, 16#6D#, 16#E8#, 16#C2#, 16#5A#); -- hex
begin
for i in 0 to 4 loop

    wait until rising_edge(clk) and valid='1';

    assert to_integer(unsigned(dout)) = dout_vals(i)
        report "dout incorrect";

    wait until valid='0';

end loop;
end process;
```

# Check Error Signal

```
err0 : process
begin

wait until rising_edge(clk) and err='1';

report "error signal asserted";

wait until err='0';

end process;
```

Report statement will also print current simulation time

# Testbench Output

```
# vsim -do vsim.start -lib work -c -quiet -t lps
work.recvrv_tb_vhd
# do vsim.start
# hexadecimal
# ** Error: dout incorrect
# Time: 2402343873 ps Iteration: 0 Instance: /recvrv_tb_vhd
# ** Note: error signal asserted
# Time: 3509114763 ps Iteration: 0 Instance: /recvrv_tb_vhd
# ** Failure: Sim done
# Time: 3694661435 ps Iteration: 0 Process: /recvrv_tb_vhd/tb
File: recvrv_tb_new.vhd
# Break at recvrv_tb_new.vhd line 175
```

# Full Code

See `recvr_tb.vhd` under class handouts directory

# Simple Wait Variations

- wait on [signal list];  
waits until event on any signal in list
- wait until [signal condition];  
waits until condition satisfied
- wait for [time duration];

# Complex Wait Variations

- wait until [signal condition] and [condition]
- wait on [signal list] for [time duration];
- wait on [signal list] until [condition];
- wait on [sig list] until [cond] for [time];

With "wait on/until ... for" you don't know "why" the wait ended.

Must test with additional statements to determine.

# Wait Summary

STATEMENT	SENSITIVITY LIST	CONDITION	TIMEOUT
<b>wait;</b>	none**	true	time'high
<b>wait on S1, S2;</b>	S1, S2	true	time'high
<b>wait until Clk'event;</b>	Clk	Clk'event	time'high
<b>wait until Clk'event and Clk = '1';</b>	Clk	Clk'event and Clk = '1'	time'high
<b>wait on S1 until CLK = '1';</b>	S1	Clk = '1'	time'high
<b>wait on S2 for 100 ns;</b>	S2	true	100 ns
<b>wait on S1, S2 until Clk = '1' for 10 us;</b>	S1, S2	Clk = '1'	10 us
<b>wait on CLK for VAR * 1 ns;</b>	Clk	true	Var * 1 ns
<b>wait on S1 until VAR = 10;</b>	S1	VAR = 10	time'high
<b>wait until VAR = 10;</b>	none**	VAR = 10	time'high
<b>wait on S1'transaction;</b>	S1'transaction	true	time'high
<b>wait for 100 ns;</b>	none	true	100 ns

# Signal Attributes for Timing Checks

**Table 5.1 Summary of the VHDL Signal Attributes**

S'event	<p><b>Function</b> returning a <b>Boolean</b> that identifies if signal S has a new value assigned onto this signal (i.e., value is different that last value).  <b>if</b> Clk'event <b>then</b> ... -- if Clk just changed in value' then ...  <b>wait until</b> Clk'event <b>and</b> Clk = '1'; -- rising edge of clock</p>
S'active	<p><b>Function</b> returning a <b>Boolean</b> that identifies if signal S had a new assignment made onto it (whether the value of the assignment is the SAME or DIFFERENT).  <b>if</b> Data'active <b>then</b> ... -- New assignment of Data -- 😊  <b>wait on</b> Data'active; -- 💣  <b>wait until</b> Data'active; -- 😊</p>
S'transaction	<p><b>Implicit signal</b> of type <b>bit</b> created for signal S when S'transaction is used in the code. This implicit signal is NOT declared since it is implicitly defined. [1] This signal toggles in value (between '0' and '1') when signal S had a new assignment made onto it (whether the value of the assignment is the SAME or DIFFERENT. The user should NOT rely on its VALUE.  -- Process resumes when ReceiveData gets a new signal  -- assignment of same or different value.  <b>wait on</b> ReceivedData'transaction;</p>

# Attributes (cont)

S'delayed(T)

**Implicit signal** of the same **base type** as S. It represents the value of signal S delayed by a time Tn. Thus, [1] *the value of S'delayed(T) at time Tn is always equal to the value of S at time Tn - T*. For example, the value of S'delayed(5 ns) at time 1000 ns is the value of S at time 995 ns. If time is omitted, it defaults to 0 ns.

```
Subtype BV2_Typ is Bit_Vector(1 to 2);
```

```
....
```

```
wait on Data'transaction;
```

```
case BV2_Typ'(Data'Delayed & Data) is
```

```
when "X0" => ... -- from X to 0 transition
```

```
when "10:" => ... -- from 1 to 0 transition
```

```
when others => ... --
```

```
end case;
```

Data at last delta time &  
Data at current time



S'stable(T)

**Implicit signal** of **Boolean** type. [1] *This implicit signal has the value TRUE when an event (change in value) has NOT occurred on signal S for T time units, and the value FALSE otherwise.* If time is omitted, it defaults to 0 ns.

```
if Data'stable(40 ns) then -- met set up time
```

# Attributes (cont)

S'quiet(T)

**Implicit\_signal** of **Boolean** type. [1] *This implicit signal has the value TRUE when the signal has been quiet (i.e., no activity or signal assignment) for T time units, and the value FALSE otherwise. If time is omitted, it defaults to 0 ns.*

```
    if Data'quiet(40 ns) then -- Really quiet, not even an assignment of
        -- the same value during the last T time units
```

S'last\_event

**Function** returning the [1] amount of **time** that has elapsed since the last event (change in value) occurred on signal S. If there was no previous event, it returns Time'high (The maximum value for time).

```
    variable : TsinceLastEvent : time;
```

```
    -- ..
```

```
    TsinceLastEvent := Data'last_event;
```

S'last\_active

**Function** returning the [1] amount of **time** that has elapsed since the last activity (assignment) occurred on signal S. If there was no previous event, it returns Time'high.

```
    variable : TsinceLastEvent : time;
```

```
    -- ..
```

```
    TsinceLastEvent := Data'last_active;
```

# DFF Example

```
entity DFF is
  port (D, CK: in BIT; Q, NOTQ: out BIT);
end DFF;

architecture CHECK_TIMES of DFF is
  constant HOLD_TIME: TIME := 5 ns;
  constant SETUP_TIME: TIME := 3 ns;
begin
  process (D, CK)
    variable LastEventOnD, LastEventOnCk: TIME;
  begin
    -- Check for hold time:
    if D'EVENT then
      assert NOW = 0 ns or
        (NOW - LastEventOnCk) >= HOLD_TIME
        report "Hold time too short!"
        severity FAILURE;
      LastEventOnD := NOW;
    end if;
  end process;
end CHECK_TIMES;
```

```
-- Check for setup time:
if CK = '1' and CK'EVENT then
  assert NOW = 0 ns or
    (NOW - LastEventOnD) >= SETUP_TIME
    report "Setup time too short!"
    severity FAILURE;
  LastEventOnCk := NOW;
end if;

-- Behavior of FF:
if CK = '1' and CK'EVENT then
  Q <= D;
  NOTQ <= not D;
end if;
end process;
end CHECK_TIMES;
```