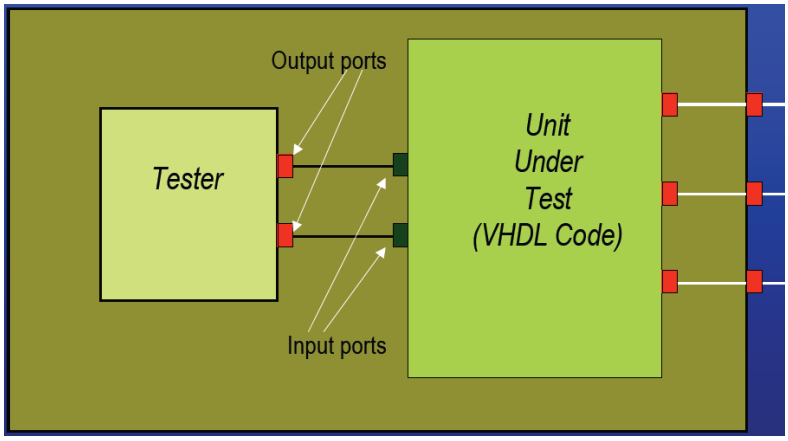
Zastosowanie

- Testowanie układu na „okoliczność” wystąpienia błędów na każdym etapie projektowania
  - Poprawność projektu sprawdzana poprzez symulację za pomocą oprogramowania, metody formalne lub emulację sprzętową
- Weryfikacja poprzez symulację jest ciągle najbardziej popularną metodą dzięki możliwości symulacji kodu VHDL
- Kod w VHDL poddawany symulacji za pomocą konstrukcji testbench

Zastosowanie

- Testbench – nadrzędna jednostka projektowa w hierarchii projektu
- Składa się z dwóch bloków:
- Nadrzędnego modułu projektowanego bloku VHDL
- jednostki projektowej wykorzystywanej do generowania wymuszeń (łącznie z zegarem), odczytu wektorów testowych z pliku zewnętrznego, prezentacji wyników w oknie symulatora lub zapis do pliku

Testbench



Główne zastosowania konstrukcji testbench

- **Generowanie wymuszeń do celów symulacji**
- **Doprowadzenie wektorów testowych do układu podawanego testowaniu (CUT – circuit under test) oraz pobieranie z niego odpowiedzi**
- **Zadanie opcjonalne: porównanie otrzymanych wyników z wartościami oczekiwanymi**

Konstrukcja testbench

- fragment typowej konstrukcji testbench

```
entity test is
end;

architecture testuj of test is
       component uklad_testowany
              port (lista portów w projekcie);
       end component;
       deklaracje sygnałów;
begin
       generuj_wymuszenia(vector);
       dostarcz_do_ukladu_testowanego;

       U0:uklad_testowany port map (przypisanie portów);

       opcjonalnie_monitoruj_wyniki;
end testuj;
```


Testbench

- Wektor podawany jest automatycznie do odpowiednich portów CUT poprzez konstrukcję mapowania portów polegającą na wewnętrznym przypisaniu sygnałów

- Potrzeba generowania dwóch typów sygnałów:
- Zegar – sygnał powtarzalny, okresowy
- sekwencyjne przypisanie sygnałów – wektory testowe CUT

Zegar

- Zegar można wygenerować używając współbieżnego przypisania sygnałów

```
zegar <= not zegar after 100 ns;
```

- przykład zegara niesymetrycznego

```
process is
        constant wylaczony:time:= 10 ns;
        constant wlaczony:time:= 20 ns;
begin
        wait for wylaczony;
        zegar <= '1';
        wait for wlaczony;
        zegar <= '0';
end process;
```


Zatrzymanie symulacji

- Zegar jest generowany w sposób nieskończony (kończy się, gdy zostaje przekroczona maksymalna wartość dla typu time – time'high)
- Konieczność określenia przedziału generowania zegara w inny sposób
- Składnia assert + określenie czasu

```
assert koniec_zegara <= 500 ns;
report "Koniec symulacji"
severity error;
```

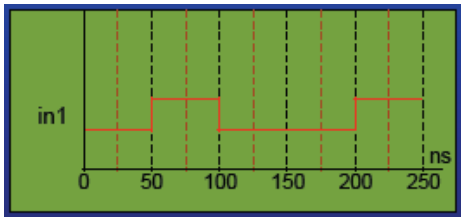
- instrukcja wait

```
if koniec_zegara > 500 ns then;
        wait; -- nieokreślone zawieszenie procesu
end if;
```

Sekwencyjne generowanie sygnałów

- wykorzystanie współbieżnego przypisania sygnałów

```
in1 <= '0' after 50 ns, '1' after 50 ns, '0' after 100 ns,
'1' after 200 ns;
```



Przykład

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

entity tester is;
      port (generuj_adresy: out std_logic_vector (2 downto 0))
end tester;

architecture wymuszenia of tester is
begin
wymuszaj:process
begin
      generuj_adresy <= '000'
      wait for 10ns;
      generuj_adresy <= '001'
      wait for 10ns;
      generuj_adresy <= '100'
      wait for 10ns;
end process;
end wymuszenia;
```

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

entity testbench is
port(test_output : out std_logic_vector(7 downto 0)); -- top
                                                      --level output
end testbench;

architecture lets_see_if_this_thing_works of testbench is

-- declare which components we will be instantatiating

component tester_component
port( generated_address : out std_logic_vector(2 downto 0) );
end component;

component rom_chip_component
port( address      : in std_logic_vector(2 downto 0);
      data_out     : out std_logic_vector(7 downto 0));
end component;

-- configuration specification of the components

for U1:tester_component   use entity WORK.tester(stimulus);
for U2:rom_chip_component use entity WORK.rom_chip(rom);

-- declare an internal signal to hook up the two components

signal s1 : std_logic_vector(2 downto 0); -- used to connect
                                          -- the address lines

begin -- let's test!

U1: tester_component   port map(generated_address => s1);
U2: rom_chip_component port map(address => s1, data_out =>
test_output);

end lets_see_if_this_thing_works;
```


Sekwencyjne generowanie sygnałów

- Wektory testowe mogą być wczytywane z plików tekstowych za pomocą standardowej procedury READ
  - Możliwe typy danych
    - bit, character, boolean, integer, real, bit_vector, time, string
- Dane z plików wczytywane są do zmiennych

```
PROCESS
  --   variable for the data to be read form the input file
  VARIABLE a_temp : INTEGER;
  VARIABLE b_temp: INTEGER;
  VARIABLE c_in_temp : INTEGER;
  VARIABLE in_line: line;


  --   variable for the data to be written to the output file
  VARIABLE c_out_temp: INTEGER;
  VARIABLE s_temp : INTEGER;


  FILE infile: TEXT IS IN
    "ripple_16bit.inp";
  FILE outfile: TEXT IS OUT
    "ripple_16bit.out";


  BEGIN

    IF ENDFILE(infile) THEN
              ASSERT FALSE
                REPORT "<<WARNING>>-END OF FILE REACHED"
                  SEVERITY ERROR;
    ELSE
       WAIT UNTIL CLK'event AND CLK = '1';
       READLINE(infile, in_line);
       READ(in_line, a_temp);
       a_tb <= CONV_UNSIGNED(a_temp, 16);
       READ(in_line, b_temp);
       b_tb <= CONV_UNSIGNED(b_temp, 16);
       READ(in_line, c_in_temp);
       c_in_tb <= CONV_UNSIGNED(c_in_temp, 1);

       -- s_temp := s_tb;


       WAIT UNTIL CLK'EVENT AND CLK = '0';

    END IF;

END PROCESS;
```

13

Przykład projektu Xilinx ISE – prezentacja narzędzia

14