



Politecnico di Milano – Polo di Como

Facoltà di Ingegneria dell'Informazione
Dipartimento di Elettronica e Informazione

VHDL Test Benching

Politecnico di Milano
Polo di Como

Sommario



- Generalità
- Basi di *testbenching*
- Struttura di un *testbench*

Generalità



- Un aspetto rilevante della realizzazione è verificare che la descrizione rispetti le specifiche
- La verifica (*testing*) di un progetto viene svolta generando una sequenza di stimoli di ingresso e verificando la validità della sequenza risultante
 - Generazione degli stimoli
 - Realizzazione manuale del file degli stimoli
 - Spesso impraticabile
 - Attraverso uno o più *Test Bench*
 - Verifica
 - La verifica può avvenire mediante una analisi visuale del risultato prodotto (grafica)
 - Oppure mediante un processo che verifica la correttezza delle operazioni

Generalità

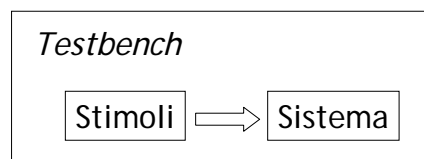


- La verifica del progetto è completa quando tutte le sequenze applicate coprono l'intera funzionalità
 - La copertura della specifica deve essere del 100%
 - “*Ogni aspetto non verificato sicuramente non andrà bene*”, Legge di Marphy
- Tre problemi
 - Come realizzare il test bench
 - Come svolgere la verifica
 - Come realizzare il test bench che garantisca la copertura totale

Basi di *Testbenching*



- Il *testbench* è un modulo VHDL, non sintetizzabile, costituito da
 - Una *entity* priva di port
 - Il *test bench* rappresenta l'ambiente
 - Il componente che deve essere verificato
 - I costrutti e/o processi che generano gli stimoli



Basi di *Testbenching*



➤ Esempio

```
entity TestAdder is
end entity TestAdder;

architecture TB of TestAdder is
    signal A, B, Sum: std_logic_vector(1 downto 0);
    signal Cin, Cout: std_logic;
begin
    S0: entity work.adder(mio)
        port map(A, B, Cin, Sum, Cout);

    Cin <= '0', '1' after 10 ns, '0' after 25 ns;
    A <= "00", "01" after 5ns, "11" after 10 ns;
    B <= "00", "11" after 15 ns;
end;
```

Basi di *Testbenching*



➤ Esempio

```
entity TestAdder is
end entity TestAdder;

architecture TB of TestAdder is
    signal A, B, Sum: std_logic_vector(1 downto 0);
    signal Cin, Cout: std_logic;
begin
    S0: entity work.adder(mio)
        port map(A, B, Cin, Sum, Cout);
    process is
    begin
        Cin <= '0'; A <= "00"; B <="00"; wait for 5 ns;
        A <= "01"; wait for 10 ns;
        A <= "11"; wait for 15 ns;
        B <= "11"; wait;
    end process;
end;
```

Ferma il processo definitivamente

Basi di *Testbenching*



- Per comodità di interpretazione, è possibile generare dei valori interi al posto dei bit e vettori di bit
- Il VHDL è fortemente tipizzato; la disomogeneità di tipo deve essere risolta esplicitamente
 - Cast esplicito mediante funzioni
 - `std_logic_vector(unsigned)`
 - `to_unsigned(integer, val)`
 - `unsigned(std_logic_vector)`
 - `to_integer(unsigned)`

Basi di *Testbenching*



➤ Esempio

```
architecture TB of TestAdder is
    signal AInt, BInt, SumInt: natural;
    signal A, B, Sum: std_logic_vector(1 downto 0);
    signal Cin, Cout: std_logic;
begin
    S0: entity work.adder(mio)
        port map(A, B, Cin, Sum, Cout);
    A <= std_logic_vector(to_unsigned(AInt, 2));
    B <= std_logic_vector(to_unsigned(BInt, 2));
    SumInt <= to_integer(unsigned(Cout&Sum));
    process is
    begin
        Cin <= '0'; AInt <= 0; BInt <= 0;
        wait for 5 ns;
        AInt <= 1;
        ...
    end process;
end TB;
```

Valori interi

Test Bench

© Fabio Salice

9

Basi di *Testbenching*



- Per generare segnali per i quali è indicata una periodicità (modulo 2^n) è conveniente l'uso dei *process*
 - Esempio: segnali di ingresso a reti combinatorie

```
signal A: std_logic_vector(n+1 downto 0);

SignA: process is
begin
    wait for 10 ns;
    A <= A + 1;
end process;
```

Test Bench

© Fabio Salice

10

Basi di *Testbenching*



- In modo analogo è possibile realizzare segnali scalari periodici

- Esempio: clock

```
signal clock : std_logic := '0';
```

```
begin
```

```
  clk: process is
```

```
  begin
```

```
    clock <= not clock after 10 ns;
```

```
  end process;
```

```
  ...
```

```
end;
```

```
clock <= not clock after PERIOD/2;
```

Basi di *Testbenching*



- Per generare segnali periodici con *duty-cycle* diverso da 50% si esplicitano le durate degli intervalli

```
clk: process is
```

```
begin
```

```
  clock <= '0';
```

```
  wait for 15 ns;
```

```
  clock <= '1';
```

```
  wait for 5 ns;
```

```
end process;
```

1. Meno overhead rispetto al processo
2. Il clock deve essere inizializzato

```
clock <= '0' after 3*PERIOD/4 when clock='1' else  
  '1' after PERIOD/4  when clock='0' else  
  '0';
```

```
// clock con duty-cycle al 50%
```

```
clock <= not clock after PERIOD/2;
```

Basi di *Testbenching*



- Per generare segnali non periodici mediante un *process* si utilizza il costrutto *wait* (mantiene sospeso il processo indefinitamente)

- Esempio: reset, clear

```
rst: process is
begin
    reset <= '0'; -- attivo basso
    wait for 20 ns;
    reset <= '1';
    wait;
end process;

reset <= '1',
        '0' after 20 ns,
        '1' after 40 ns;
```

Meno overhead
rispetto al processo

Basi di *Testbenching*



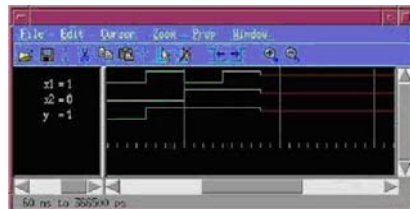
- Infine, per generare segnali di ingresso sincronizzati con altri si utilizza un *process* che ha tali segnali nella una *sensitivity list*

```
SegnA_sinc: process (clock) is
Begin
    if (clock=1 and clock'event)
        A <= A+1 after 10 ns;
    end if;
end process;
```

Basi di *Testbenching*



- Realizzato il *testbench*, la simulazione produce una forma d'onda che deve essere analizzata dal progettista che verifica la correttezza del progetto



- Se la forma d'onda è particolarmente lunga e coinvolge molti segnali, la verifica può risultare complessa, molto dispendiosa e potenzialmente soggetta ad errori

Basi di *Testbenching*



- Un modo per risolvere il problema relativo alla onerosità della verifica consiste nell'analizzare delle condizioni di funzionamento specifiche sui segnali coinvolti
 - NOTA: Non esistono problemi di sincronizzazione poiché i segnali provengono dalla descrizione in analisi
- Si utilizzano il costrutto `assert`
 - Assert *condizione*
Report *messaggio*
Severity *gravità errore*
 - Il costrutto `assert` verifica che la condizione sia vera. Se la condizione è falsa, genera il messaggio a cui è concatenato il livello di gravità dell'errore

Basi di Testbenching



► Esempio:

```
wait for 10 ns;  
x1 <= '1'; x2 <= '1';  
assert y = (x1 xor x2)  
  report "E@TB: failure at:" &  
    "x1=" & str(x1) &  
    " x2=" & str(x2)  
  severity Error;  
wait for 10 ns;
```

str()
dal package
txt_util.vhd



```
# ** Error: E@TB: failure at: x1=1 x2=1  
# Time: 40 ns  Iteration: 0 Instance: /tb1
```

Basi di Testbenching



- Per il messaggio d'errore si utilizza formato "standard" per facilitare l'identificazione della posizione dell'errore
- Lo standard suggerito è
 - Una lettera che indica la gravità
 - I=Information, W=Warning, E=Error, F=Failure
 - @ o at
 - Il nome della *entity* che genera il messaggio

```
# ** Error: E@TB: failure at: x1=1 x2=1  
# Time: 40 ns  Iteration: 0 Instance: /tb1
```

Basi di *Testbenching*



- Per generare risposte dinamiche sulla base del comportamento del DUT è possibile utilizzare la seguente strategia
 - Solo per casi semplici

```
process(Data_Bus)
begin
  case Data_Bus is
    when 3      => response <= '1' after 10 ns;
    when others => response <= '0' after 10 ns;
  end case;
end process;
```

Struttura di un *Testbench*



- In generale, anche il codice per la verifica (*testbench code*) viene strutturato in modo modulare
 - Il *testbench* ha una complessità simile a quella del dispositivo da verificare
 - La probabilità di riuso di un *testbench* è ragionevolmente alta
- I *testbench* possono includere tre tipi di componenti di supporto
 - Modelli
 - Transactor
 - Bus Functional Model (*BFM*)

Struttura di un *Testbench*



➤ Modello

- È una descrizione del dispositivo che si comporta nello stesso modo e il cui il comportamento può essere controllato solo dagli stimoli in ingresso.
 - Es: RAM

➤ Transactor

- È un modello nel quale sono aggiunti dei meccanismi di controllo come, ad esempio, la lettura dei dati di controllo da file.

➤ Bus Functional Model (*BFM*)

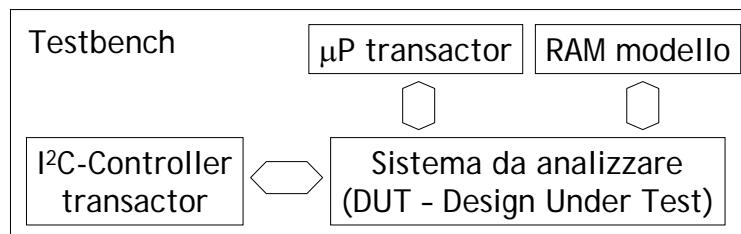
- Un *BFM* (modello funzionale di un BUS) è il modello di un dispositivo che si comporta come un BUS.
 - ES: un processore che contiene solo dei processi di lettura e scrittura senza ALU, registri...

Struttura di un *Testbench*



➤ Struttura tipica di un *testbench*

- Esempio
 - La *top-entity* (testbench) istanzia quattro componenti: Il processore e il I²C-Controller (transactor), la RAM (modello) e il sistema che deve essere analizzato (DUT)
 - Ogni componente del *testbench* produce gli stimoli e verifica la risposta proveniente dal DUT



Transactor e BFM



- La realizzazione di *Transactor* e *BFM* richiede l'uso di file di supporto su cui scrivere e, principalmente, leggere dati e comandi.
- Per le operazioni di gestione dei file (e delle stringhe) è necessario includere un package

Use STD.textio.all

- Come per altri linguaggi, i file devono essere dichiarati aperti
 - Dichiarazione e apertura: **file** *FP* : **text is in** "nome";
 - Modo: in o out

Transactor e BFM



- Processo per la generazione degli stimoli mediante file

```
constant PERIOD : time := 10 ns;
begin
...
STIMULI : process
  variable L_IN : line;
  variable DATA : std_logic_vector(7 downto 0);
  file STIM_IN : text is in "stim_in.txt";
begin
  W_VALUE <= (others => '0');
  wait for PERIOD;
  while not endfile (STIM_IN) loop
    readline (STIM_IN, L_IN); // legge una riga da file
    hread (L_IN, DATA); // legge da L_IN un valore esadecimale e lo
                        // traduce in std_logic_vector
    W_VALUE <= DATA;
    wait for PERIOD;
  end loop;
  wait;
end process STIMULI;
```

Transactor e BFM



► Esempio ulteriore

```
File da leggere
FF FF
FF A1
A0 FF
...

STIMULI : process
  variable L_IN : line;
  variable DATA1 : std_logic_vector(7 downto 0);
  variable DATA2 : std_logic_vector(7 downto 0);
  variable CHAR : character;
  file STIM_IN : text is in "stim_in.txt";
begin
  while not endfile (STIM_IN) loop
    readline(STIM_IN, L_IN);
    hread(L_IN, DATA1);
    read(L_IN, CHAR);
    hread(L_IN, DATA2);
    W_VALUE <= DATA1&DATA2;
    wait for PERIOD;
  end loop;
  wait;
end process STIMULI;
```

Transactor e BFM



► Processo per la generazione degli stimoli mediante file

```
RESPONSE : process(W_RESULT)
  variable L_OUT : line;
  variable CHAR_SPACE : character := ' ';
  file STIM_OUT : text is out "stim_out.txt";
begin
  write (L_OUT, now);
  write (L_OUT, CHAR_SPACE);
  write (L_OUT, W_RESULT);
  write (L_OUT, CHAR_SPACE);
  hwrite (L_OUT, W_RESULT); // esadecimale
  write (L_OUT, CHAR_SPACE);
  write (L_OUT, W_OVERFLOW);
  writeline (STIM_OUT, L_OUT); // scrive nel file
end process RESPONSE;
```

Appendice



- **str()**: trasforma un `std_logic` in una stringa (in `txt_util.vhd`)
- **hstr()**: trasforma un `std_logic_vector` in una stringa in formato esadecimale (in `txt_util.vhd`)

Regression Testing



- Test that a refinement of a design is correct
 - that lower-level structural model does the same as a behavioral model
- Test bench includes two instances of design under test
 - behavioral and lower-level structural
 - stimulates both with same inputs
 - compares outputs for equality
- Need to take account of timing differences

Regression Test Example



```
architecture regression of test_bench is
  signal d0, d1, d2, d3, en, clk : bit;
  signal q0a, q1a, q2a, q3a, q0b, q1b, q2b, q3b : bit;
begin
  dut_a : entity work.reg4(struct)
    port map ( d0, d1, d2, d3, en, clk, q0a, q1a, q2a, q3a );
  dut_b : entity work.reg4(behav)
    port map ( d0, d1, d2, d3, en, clk, q0b, q1b, q2b, q3b );
  stimulus : process is
  begin
    d0 <= '1'; d1 <= '1'; d2 <= '1'; d3 <= '1'; wait for 20 ns;
    en <= '0'; clk <= '0'; wait for 20 ns;
    en <= '1'; wait for 20 ns;
    clk <= '1'; wait for 20 ns;
    ...
    wait;
  end process stimulus;
  ...
```

Test Bench

© Fabio Salice

29

Regression Test Example



```
...
verify : process is
begin
  wait for 10 ns;
  assert q0a = q0b and q1a = q1b and q2a = q2b and q3a = q3b
    report "implementations have different outputs"
    severity error;
  wait on d0, d1, d2, d3, en, clk;
end process verify;
end architecture regression;
```

Test Bench

© Fabio Salice

30