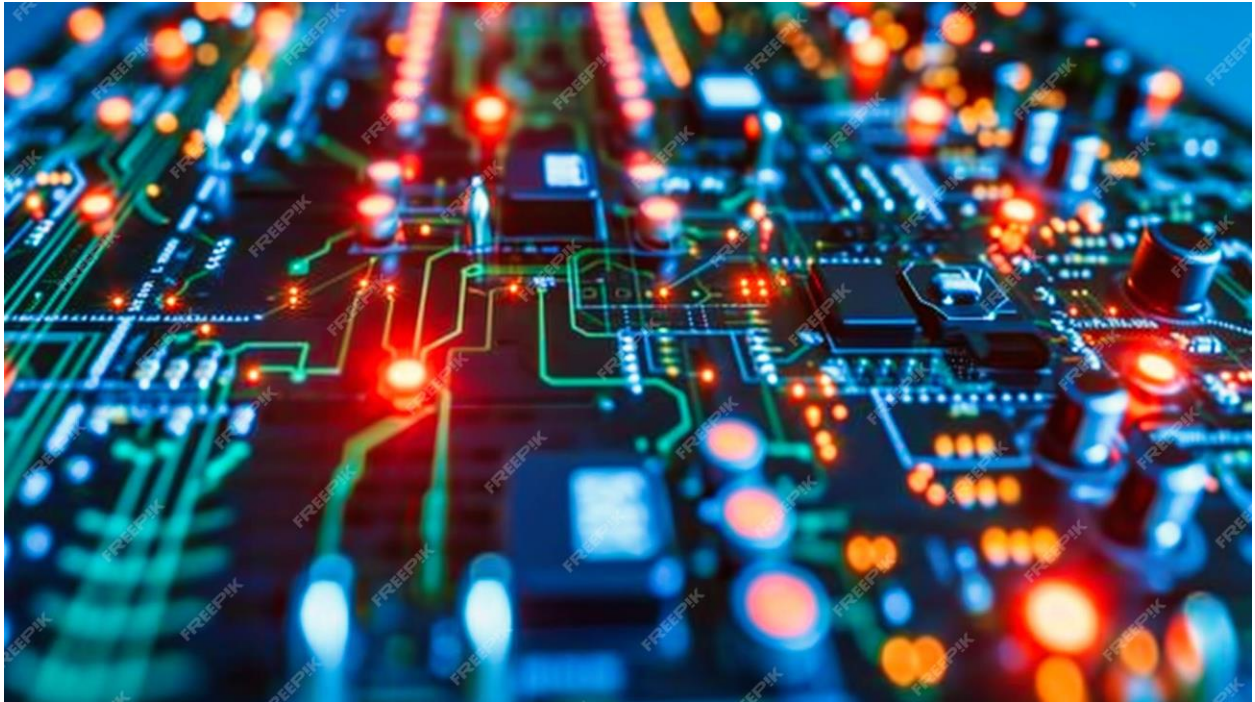


VHDL TEXTIO



PRECISAZIONI:

Lo studio del materiale contenuto in questo documento non e' richiesto per l'esame. All'esame, almeno per quest'anno accademico, NON verra' richiesto l'utilizzo delle funzioni VHDL dedicate a I/O su file in fase di simulazione.

Il materiale fornito ha il solo obiettivo di mostrare come, mediante l'utilizzo di librerie VHDL disponibili nell'installazione standard (e quindi anche nel vostro sistema di esercitazione) sia possibile effettuare l'analisi d'onda anche mediante un file di output di tipo testuale.

Per dare un esempio di utilizzo di TEXTIO modificheremo dei file che avete gia' visto nel corso di L7(FSM) : seq_det.vhdl e seq_det_tb.vhdl (che diventera' seq_det_texttb.vhdl).

Package TEXTIO e STD_LOGIC_TEXTIO:

I package textio e std_logic_textio fanno parte, rispettivamente, delle librerie ieee e std. Essi danno la possibilita' di leggere/scrivere da e verso file di input/output in formato testuale. Vengono spesso utilizzati insieme poiche', per leggere/scrivere il valore di segnali da e su file esterni sono necessarie funzioni specifiche per i diversi tipi di dato e la libreria ieee non copre tutti i tipi di dati comunemente utilizzati in descrizioni/testbench VHDL. In particolare e' necessario utilizzare i package di entrambe le librerie se vogliamo essere in grado di scrivere sia i dati di tipo std_logic che i dati di tipo std_logic_vector.

In questo esempio di modifica di due file visti insieme in classe non ci occuperemo della parte di input (input di valori da file di testo esterno per impostare i segnali nel testbench in fase di simulazione) ma solo della parte di scrittura su file esterno.

VHDL utilizza la visibilita' dei segnali per rendere inaccessibili (o quasi) i segnali interni di UUT all'esterno (ad esempio nel testbench). Questo e' necessario poiche' UUT e testbench hanno vincoli diversi. Primo fra tutti quello che UUT **deve** essere sintetizzabile mentre testbench no. Limitare fortemente l'accessibilita' ai segnali interni dei componenti e' la via piu' sicura da seguire. Specialmente quando utilizziamo design di tipo gerarchia puo' essere composta da molti livelli. Per quanto sensato questo approccio rende piu' difficile avere un unico file di testo nel quale siano riportati i valori di segnali interni a componenti appartenenti a diversi livelli della gerarchia. Questo esempio mostra come aggirare il problema.

Partiamo da una generica descrizione di textio e di std_logic_textio. Per poterli utilizzare dobbiamo includere quanto segue all'inizio dei sorgenti VHDL:

```
library std;
use std.textio.all;

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

ATTENZIONE: per un qualche strano motivo la libreria STD **non** e' considerata parte dello standard ieee VHDL (la libreria dello standard ieee e' ieee!) anche se il nome ricorda molto la parola 'standard' . Di conseguenza, se vogliamo utilizzare funzioni provenienti da packages contenuti in STD, siamo **obbligati ad utilizzare SIA IN FASE DI ANALISI CHE IN FASE DI ELABORAZIONE un'opzione aggiuntiva :**

-fsynopsys

che va scritta **DOPO** -a o -e . Quindi analisi diventa ghdl -a -fsynopsys sorgente.vhdl mentre elaborazione diventa ghdl -e -fsynopsys sorgente .

Textio definisce DUE NUOVI tipi di dato che vengono utilizzati per interfacciarsi con file di testo:

1) file

```
file file_handle : <file_type> open <file_mode> is <"filename">;
```

Il file_handle e' semplicemente un nome di variabile da utilizzare per leggere/scrivere da e su file. file_type puo' assumere due valori : TEXT se vogliamo usare il file come semplice file di testo o INTF se vogliamo leggere/scrivere con un vincolo di formato, potremo leggere/scrivere solo interi con segno espressi su 32 bit. Il caso di utilizzo piu' comune e' il formato TEXT ed esistono molte funzioni in grado di convertire i tipi VHDL bit/bit_vector, std_logic/std_logic_vector in stringhe di caratteri che vengono scritte sul file di testo e possono essere utilizzate da altri programmi. Noi vedremo solo l'utilizzo di file in formato TEXT. file_mode specifica la modalita' di interazione con il file e puo' assumere due valori: WRITE_MODE (per la scrittura) e READ_MODE (per la lettura).

La definizione di un riferimento a file in VHDL puo' essere effettuata come nei seguenti esempi:

```
file Fout: TEXT open WRITE_MODE is "output_file.txt";
```

```
file Fin: TEXT open READ_MODE is "input_file.txt";
```

Questi comandi creano automaticamente i file nel caso in cui essi non siano presenti nella directory corrente. Se sono presenti il loro contenuto viene sovrascritto.

Prima di passare al secondo nuovo tipo di dato fornito da textio e' necessario precisare che la definizione esplicita di un file handle in VHDL NON E' OBBLIGATORIA. Di fatto e' SEMPRE disponibile un file handle di sistema, un "file" sul quale, normalmente, vengono scritte tutte le informazioni quando usiamo delle funzioni come print e non specifichiamo dove scrivere. Questo file e' lo schermo ... e ad esso e' associato un canale di output predefinito che si chiama STANDARD OUTPUT, a volte abbreviato in STDOUT. In VHDL anche se non abbiamo definito un file di testo su cui scrivere, possiamo scrivere OUTPUT come file handle e, in questo caso, tutto verra' scritto sullo schermo (nel terminale). In

2) line

```
variable <line_variable_name> : line;
```

L'accesso al file in lettura/scrittura avviene mediante una variabile di tipo line che rappresenta la riga corrente del file sul quale stiamo lavorando. Questo tipo di dato rappresenta uno spazio temporaneo sul quale e' possibile scrivere (da qui in poi parleremo solo di scrittura ma tramite line e' possibile anche leggere valori) informazioni. UNA volta che la linea corrente ha il formato ed i dati che vogliamo scrivere e' possibile utilizzarla per aggiungere una riga di testo al file.

Ci sono due procedure che permettono di trasferire informazioni tra una variabile di tipo line in VHDL e la riga corrente di un file specificato m

```
readline(<file_handle>, <line_variable_name>);  
writeline(<file_handle>, <line_variable_name>);
```

Il trasferimento di informazione avviene usando **l'intero contenuto della variabile di tipo line** e, di conseguenza, il modo normale di agire e' il seguente :

- Modificare il contenuto della variabile di tipo line fino ad ottenere il risultato
- Scrivere il contenuto di line come riga corrente del file di testo (aggiungendo un carattere per andare a capo).

In questo modo viene letta/scritta un'intera linea di testo dal/nel file.

Tipi di dati definiti dall'utente e attributi definiti dall'utente:

Abbiamo già visto, in `seq_det.vhdl`, come definire un tipo di dato da utilizzare come **insieme degli stati** della FSM. Questo avviene mediante la parola chiave **type** e permette di evitare di utilizzare un `std_logic_vector` che verrebbe sintetizzato come il minimo insieme di bit di memoria necessario ad identificare in modo univoco tutti gli stati presenti in STG della FSM. In questo modo, ogni volta che nella descrizione VHDL è necessario riferirci ad uno stato possiamo evitare di scrivere, ad esempio, "000" (o il codice binario dello stato a cui siamo interessati in quel momento). Possiamo semplicemente utilizzare il nome dello stato (ad esempio `start`).

Per definire l'insieme degli stati in `seq_det.vhdl` avevamo scritto quanto segue:

```
type state is (start, d0_is_1, d0_not_1, d1_is_1, d1_not_1);
```

E poi avevamo dichiarato dei segnali di tipo `state`.

Il normale modo di operare di VHDL fa sì che, ogni volta che nel sorgente si trova un'occorrenza, ad esempio, di `start`, essa venga sostituita da uno `std_logic_vector` di ampiezza 3 bit contenente il valore associato a `start`. Per come è stato definito il tipo di dato `state` è un'enumerazione e **NON** fornisce in modo automatico la conversione dei valori in esso contenuti in **stringa**, cosa richiesta per poterlo scrivere in linea e, di seguito, sul `file_handle`.

Dovremo aggiungere al tipo `state` un **attributo di tipo stringa** prima di poter scrivere i valori dei segnali di tipo `state` su un file mediante `textio`. Per riuscirci dovremo scrivere quanto segue:

```
attribute enum_encoding : string; -- user defined

type state is (start, d0_is_1, d0_not_1, d1_is_1, d1_not_1);

attribute enum_encoding of state : type is "start d0_is_1 d0_not_1 d1_is_1 d1_not_1";
```

Creiamo un attributo definito dall'utente di tipo `string` e poi lo applichiamo a `state` inserendo una stringa con i nomi degli stati separati da spazi e rigorosamente nello stesso ordine in cui compaiono nella riga che inizia con `type state is`. D'ora in poi quando avremo bisogno di

estrarre la rappresentazione in formato stringa del valore di un segnale di tipo `state` potremo fare così:

```
state'image(nomesegnaleditipostate)
```

Il che ci permette di stampare il valore sul `file_handle` mediante `textio`.

Utilizzo di `textio` in descrizione VHDL / testbench :

Equipaggiati con la spiegazione di base di `textio` possiamo procedere con la descrizione del suo utilizzo in descrizione VHDL di UUT e nel testbench (per ognuno di essi avremo una sottosezione dedicata). Vedremo in fine come effettuare analisi/elaborazione ed una simulazione che produca un file di testo.

Textio in descrizione VHDL di UUT (modifica di `seq_det.vhdl`) :

La descrizione della entity `seq_det` è la seguente:

```
library STD;
use STD.textio.all;

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_textio.all;

entity seq_det is
port(clk, reset : in std_logic;
      din : in std_logic;
      err : out std_logic);
end entity;
```

L'architettura e' piu' complessa. Non cambia molto rispetto a quello che abbiamo visto insieme in classe in occasione di L7 (FSM) ma dovremo dichiarare la variabile di tipo line ed aggiungere diverse linee di codice per la stampa su OUTPUT. Come potrete notare la complessita' non e' alta ... piu' che altro si tratta di aggiungere spesso codice sempre uguale e ripetitivo. Il problema diventera' piu' evidente (molte linee di codice per stampare su OUTPUT) nel testbench. Quello che vogliamo ottenere e' che dei segnali interni di seq_det (current_state e next_state) di tipo state vengano stampati su OUTPUT OGNI volta che current_state o next_state cambiano.

Per ottenere questo risultato dovremo aggiungere le parti di codice relative a textio nel processo state_memory (quello che permette l'aggiornamento dello stato) e nel processo next_state_logic (quello che determina il valore dello stato prossimo). Il processo di output non e' coinvolto.

Il sorgente della architecture (per i soli due processi da modificare) di seq_det e' riportato di seguito :

```
-- PROCESS 1 : state memory :

state_memory : process(clk, reset)

begin

    variable smpline : line;

    if(reset = '0') then current_state <= start;

    write(smpline, string'("(R)current state: ") &
state'image(current_state));

    writeline(OUTPUT, smpline);

    elsif(clk'event and clk='1') then current_state <= next_state;

    write(smpline, string'(" current state: ") &
state'image(current_state)); writeline(OUTPUT, smpline);

    writeline(OUTPUT, smpline);

    end if;

end process;
```

Le linee evidenziate in giallo sembrano andare a capo ma e' colpa di Word ... in realta' sono righe con il carattere a capo solo dopo il ; .

Ora passiamo al secondo processo da modificare ... quello che si occupa della logica di stato prossimo.

```
-- PROCESS 2 : next state logic :
next_state_logic : process(current_state, din) is
    variable nspline : line;
begin case(current_state) is
    when start => if(din = '1') then
        next_state <= d0_is_1;
    else
        next_state <= d0_not_1;
    end if;
    when d0_is_1 => if(din = '1') then
        next_state <= d1_is_1;
    else
        next_state <= d1_not_1;
    end if;
    when d1_is_1 => next_state <= start;
    when d0_not_1 => next_state <= d1_not_1;
    when d1_not_1 => next_state <= start;
    when others => next_state <= start;
end case; -- NB: a seconda del timing degli eventi e' possibile che --
il valore di next_state sia scritto due volte (vedere sensitivity list)
write(nspline, string'(" next state: ") & state'image(next_state) & "
from event cs/din: ");
```



```

-- event originating the message (per capire quale evento genera
l'esecuzione della writeline):

if(current_state'event) then
    write(nspline, string("T/"));
else
    write(nspline, string("F/"));
end if;

if(din'event) then
    write(nspline, string("T"));
else
    write(nspline, string("F"));
end if;

writeline(OUTPUT, nspline);

end process;

```

La parte di scrittura in questo caso e' molto complessa. Viene completato il processo (cosa che assegna un valore al segnale next_state) e viene stampato il valore del segnale next_state su file_handle (riga in giallo). Poi, dato che questo processo ha in sensitivity list 2 segnali, current_state e din) cerchiamo di determinare quale segnale ha innescato l'esecuzione del processo perche' ... ogni volta che viene eseguito viene stampato il valore di next_state (quindi vediamo il valore, nel file di testo, ma non sappiamo quae segnale ha innescato la sua stampa). Costruiamo quindi una scringa nel formato X/Y in cui sia X che Y possono assumere solamente i valori T (true) e F (false). Non si tratta di variabili booleane ... solo dei caratteri T e F. Nel testo scritto su OUTPUT se al posto di X troviamo T vuol dire che l'innescato del processo arriva da un cambiamento in current_state, mentre se troviamo T al posto di Y vuol dire che l'innescato del processo arriva da un cambiamento in din.

Accodata anche questa stringa a line usiamo writeline per stamparla su OUTPUT.

Alcuni di voi si chiederanno come mai non ripuliamo mai il contenuto della variabile di tipo line. Essa e' dichiarata all'inizio del processo. Ogni volta che il processo viene avviato essa e' dichiarata e inizializzata a stringa vuota.

TEXTIO in testbench (seq_det_texttb.vhdl) :

In testbench (post generazione con testbench generator) sono necessarie alcune modifiche.

1) In primo luogo e' necessaria una precisazione. Nella sezione (definita mediante commento) signal to map to component i segnali sono dichiarati senza valori di inizializzazione.

E' necessario impostare almeno il valore iniziale del segnale di clock (clk). Il motivo e' che il processo di clock determina il ciclo del segnale clk negandolo. Se il valore iniziale di clk e' 'U' la sua negazione produrra' sempre 'U' (il che equivale ad avere un segnale di clock fisso che non serve a nulla).

E' necessario, in questa, sezione cambiare almeno

```
signal clk : std_logic;
```

```
in
```

```
Signal clk : std_logic := '1';
```

in questo modo il valore iniziale di clk e' impostato a '1' (la prima esecuzione del processo di clock, con negazione, lo portera' a '0'. E' per questo che scegliamo il valore iniziale '1').

2) Il processo di definizione degli stimoli da inviare a UUT dovra' contenere, nella parte delle dichiarazioni, una variabile di tipo line :

```
stimulus: process is
-- TEXTIO variables : current line
variable current_line : line;
begin
```

Come prima cosa scriviamo su OUTPUT alcune righe di intestazione :

```
begin

-- write your test hereE

-- Report header:

-- writing the content of current_line

write(current_line, string'("-----"));

writeline(OUTPUT, current_line);

write(current_line, string'("Beginning test: in (reset, din) out (err)"));

-- write current_line in the output file (Fout)

writeline(OUTPUT, current_line);

write(current_line, string'("-----"));

writeline(OUTPUT, current_line);
```

Nella parte di impostazione degli stimoli, invece, dovremo PER OGNI VOLTA CHE IMPOSTAMO GLI STIMOLI E USIAMO WAIT, una parte FISSA in cui utilizziamo textio per stampare input / reset / output (ERR) di UUT.

Riporto solamente l'esempio della prima impostazione degli stimoli. La parte textio, costante, va aggiunta per ogni impostazione degli stimoli. Il file di esempio che ricevete (seq_dt_texttb.vhdl) e' completo.

```
-- Begin stimuli test

reset <= '0'; wait for 5 ns;

write(current_line, now, UNIT => ns);

write(current_line, string'(" reset="));

write(current_line, reset);

write(current_line, string'(" din="));

write(current_line, din);

write(current_line, string'(" err="));
```

```
write(current_line, err);  
writeline(OUTPUT, current_line);
```

Il file testbench va completato aggiungendo la parte textio a tutte le impostazioni degli stimoli.

Analisi, elaborazione ed esecuzione della simulazione :

Di seguito vengono riportati i comandi necessari per effettuare la simulazione con textio. Nei sistemi della famiglia Unix (tra cui Linux, il sistema operativo del vostro sistema di esercitazione) lo standard output può essere rediretto (scritto) su file.

La logica complessiva del test effettuato è la seguente : mediante textio stampiamo su OUTPUT dei segnali interni a UUT (current_state e next_state) che sarebbero difficili da ottenere e leggere in testbench a causa dei vincoli di visibilità di VHDL.

Sempre attraverso l'utilizzo di textio in testbench stampiamo segnali di UUT a cui abbiamo accesso (quelli specificati mediante port map) ossia i segnali di input (din) e di output (ERR).

In fase di simulazione utilizziamo sul terminale Linux il simbolo di redirezione > che scrive tutto quello che dovrebbe essere scritto su STDOUT in un file al quale possiamo assegnare un nome a nostra scelta. In questo modo, oltre a utilizzare textio, riusciamo anche ad aggirare i vincoli di visibilità dei segnali interni imposti da VHDL. I comandi da utilizzare sono i seguenti:

```
ghdl -a -fsynopsys seq_det.vhdl
```

```
ghdl -a -fsynopsys seq_det_texttb.vhdl
```

```
ghdl -e -fsynopsys seq_det_texttb
```

```
./seq_det_texttb --stop-time=120ns --wave=seq_det_texttb.ghw > test.txt
```

```
vim test.txt
```

Il file in cui scriviamo quello che dovrebbe essere scritto su OUTPUT (per Linux STDOUT) lo chiamiamo test.txt e, usando vim, ne ispezioniamo il contenuto. La simulazione produce

non solo il file di testo test.txt ma anche il canonico file per l'analisi d'onda (seq_det_texttb.gwh) che si può aprire normalmente in gtkwave.

Output prodotto in test.txt :

Di seguito parte del contenuto di test.txt (ispezionate l'intero file nell'ambiente di esercitazione):

```
-----  
Beginning test: in (reset, din) out (err)  
-----
```

```
    next state: start from event cs/din: F/F
```

```
(R)current state: start
```

```
5 ns reset=0 din=0 err=0
```

```
    current state: start
```

```
    next state: d0_not_1 from event cs/din: T/F
```

```
10 ns reset=1 din=0 err=0
```

```
    next state: d1_not_1 from event cs/din: F/T
```

```
    current state: d0_not_1
```

```
    next state: d1_not_1 from event cs/din: T/F
```

```
20 ns reset=1 din=1 err=0
```

```
    next state: start from event cs/din: F/T
```

```
    current state: d1_not_1
```

```
    next state: start from event cs/din: T/F
```

```
30 ns reset=1 din=0 err=0
```

```
    current state: start
```

```
    next state: d0_not_1 from event cs/din: T/F
```

```
40 ns reset=1 din=0 err=0
```

```
    next state: d1_not_1 from event cs/din: F/T
```

```
    current state: d0_not_1
```

```
    next state: d1_not_1 from event cs/din: T/F
```

...

In grassetto le parti stampate via textio dall'interno del testbench. Le righe non in grassetto, invece, sono quelle prodotte dall'interno di UUT.