

CAPITOLO 6: STILI DI CODIFICA VHDL PER LOGICHE PROGRAMMABILI

1. VHDL

Il VHDL (VHSIC Hardware Description Language) è un linguaggio di descrizione dell'hardware che consente la progettazione di circuiti integrati. È un derivato del programma Very High Speed Integrated Circuit (VHSIC) ideato dal Dipartimento della Difesa Statunitense tra la fine degli anni 70 e l'inizio degli anni 80. L'obiettivo di questo programma era la produzione della successiva generazione di circuiti integrati. I partecipanti al programma erano esortati a portare al limite questa tecnologia allo studio in ogni fase progettuale. I risultati stabilirono che i tool usati per la progettazione gate level erano inadeguati per la creazione di progetti di centinaia di migliaia di gate e occorreva un cambio di metodologia. Questo cambio avvenne nel 1981 con il VHDL. Gli obiettivi di questo linguaggio erano duplici: stabilire uno standard di interfacciamento tra i vari progettisti; avere un linguaggio con istruzioni orientate alla descrizione circuitale che, senza successo, si era cercato di fare con il programma VHSIC. Nel 1986 il VHDL fu proposto come standard IEEE.

Una completa conoscenza del VHDL può risultare difficoltosa per chi si avvicina da profano al linguaggio. Tuttavia, un subset delle sue istruzioni consente la creazione di qualsiasi tipo di progetto.

L'insieme delle istruzioni che descrivono un progetto producono una modellizzazione della realtà a cui il progettista fa riferimento. La simulazione è la barriera oltre la quale non è possibile andare con un linguaggio di descrizione dell'hardware. Per avere dei gate reali occorre tradurre queste istruzioni in logica combinatoria e logica sequenziale. Questo processo è affidato ai sintetizzatori di logica che, a partire dai livelli di astrazione tipici del linguaggio, producono una netlist composta da elementi della libreria tecnologica target. Poiché il VHDL non è stato ideato come linguaggio sintetizzabile, alcuni costrutti non sono processati dai tool di sintesi. In fase di progettazione andrà quindi posta attenzione al tipo di istruzioni usate.

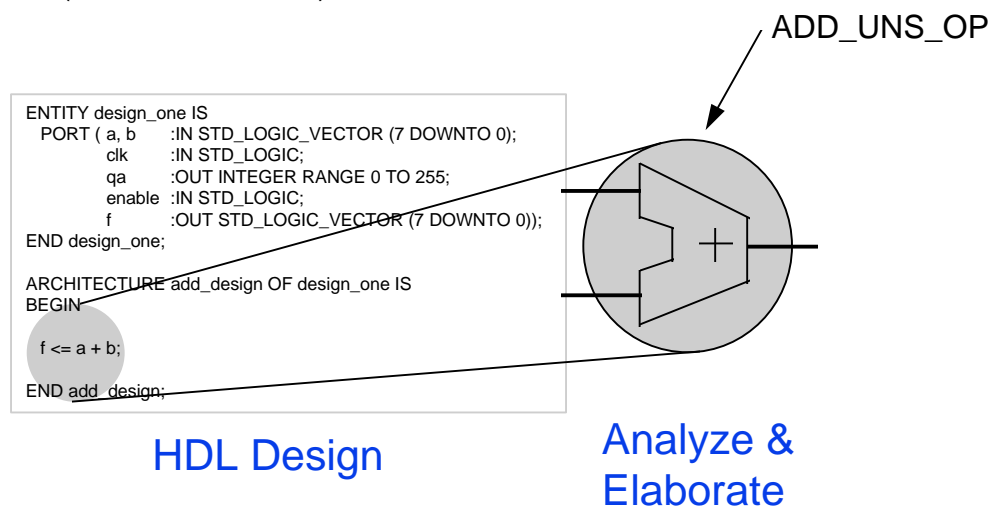
Demandando la sintesi a tool automatici, si ha la possibilità di creare progetti sempre più complicati e grandi nelle dimensioni in un tempo molto più contenuto di quanto avveniva nella progettazione gate level. Il rischio che si corre è quello di produrre una funzionalità logica con un'area maggiore del necessario e questo, per una logica programmabile, produce uno spreco di risorse preziose. Per tal motivo, un corretto uso del linguaggio e una buona conoscenza dell'architettura che ospiterà la logica garantiranno buoni risultati in termini di occupazione delle risorse e prestazioni timing. Questo si riflette in una ottimizzazione dei costi del silicio e in una riduzione dei tempi di sviluppo che, a sua volta, produce un abbattimento dei costi di sviluppo.

Nel seguito si riportano le principali considerazioni sull'utilizzo del VHDL mirato alla progettazione di una logica programmabile. Si utilizzerà e solo a scopo di esempio, come tool di riferimento per la sintesi Synopsys e come librerie tecnologiche sia quelle di Altera che di Xilinx.

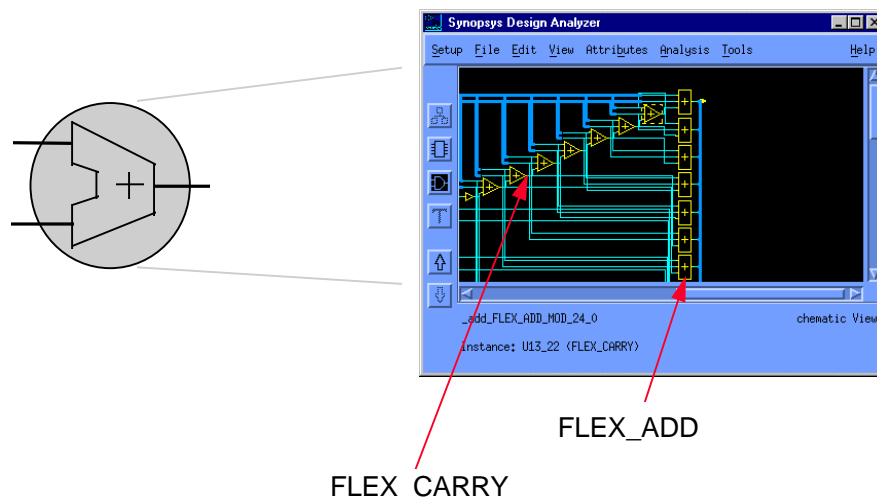
2. CONCETTI BASE SULLA SINTESI

La sintesi produce una netlist a partire da un codice e da una libreria. Questa, contenente modelli circuitali parametrici denominati DesignWare, può essere composta di due sezioni: una che il vendor di logica programmabile compila per il tool di sintesi (Synopsys DesignWare), un'altra a cui il tool di sintesi può fare riferimento per prelevare moduli caratteristici delle famiglie di componenti (Synthetic Library contenente Vendor DesignWare). L'utilizzo di questa seconda libreria non produce tipicamente buoni risultati in termini di area e di velocità Solo una verifica caso per caso, in funzione della libreria scelta e del componente selezionato, può smentire quest'affermazione.

Si parla di Inference quando il tool di sintesi, in base alle istruzioni presenti nel codice VHDL, si propone di trovare un appropriato modello gate level all'interno delle DesignWare messe a sua disposizione. A partire dagli operatori VHDL presenti nel codice (+, -, +1, <, >, =, ...) il comando di elaborate, incluso nella procedura di sintesi, compie la trasformazione di questi elementi in operatori sintetici (ADD, SUB, INC, ...).



Il successivo comando di compilazione trasforma questi operatori sintetici nei moduli messi a disposizione dalle DesignWare, scegliendone il più appropriato, tra un certo numero di modelli di differenti prestazioni ma stesse caratteristiche. Un operatore sintetico ad esempio, in base ai vincoli di sintesi, timing o area, può essere trasformato in una architettura cla (carry look-ahead) o in una rpl (ripple).



Si parla di Instantiation quando nel codice è espressamente dichiarato il modello circuitale che si vuole utilizzare presente in libreria.

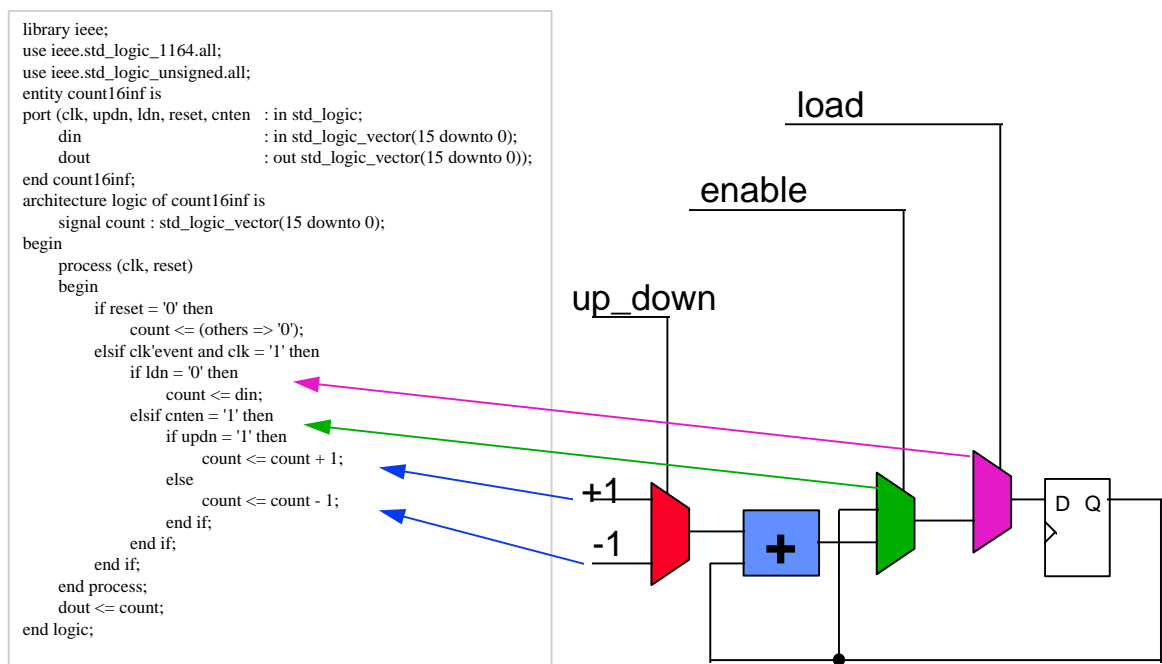
L'Inference, che non è molto efficiente per operatori complessi (x, +1, -1), ha il vantaggio di essere indipendente dalla tecnologia e di facile uso. Infatti non occorre scrivere un codice, avendo presente i vari componenti che la libreria mette a disposizione. L'Instantiation, per contro, non è portabile perchè è dipendente dalla tecnologia ed è una modalità di progettazione molto noiosa se usata in maniera diffusa. I codici risultanti, inoltre, non sono di facile leggibilità. Il vantaggio si vede soprattutto nell'utilizzo degli operatori complessi.

Al fine di produrre un codice efficiente e comprensibile, l'uso dell'Instantiation viene riservato a elementi di libreria non riconosciuti dal sintetizzatore come memorie, blocchi di startup, buffer di tipo globale, oppure a moduli in cui l'obiettivo, in termini di area o velocità non è raggiungibile con una procedura di Inference.

I tool di sintesi sono programmi dotati di enormi potenzialità ma possono anche ingenerare confusione. Avendo a disposizione un vasto scenario di strategie di sintesi e diversi modelli dello stesso operatore, è possibile ottenere un numero imprecisato di varianti sintetizzate dello stesso codice. Onde evitare divagazioni, occorre tenere ben presente i risultati che si vogliono raggiungere e i mezzi necessari ad ottenerli. La procedura di default di sintesi è un buon punto di partenza. Nel caso non si rispettino le specifiche, è possibile agire sulla sintesi, sul codice o su entrambi. Lo sforzo compiuto nel creare un codice, in cui sia quasi tutto istanziato, deve essere giustificato da reali necessità. Allo stesso modo il costo di una sintesi, in termini di calcolo, deve essere calibrato alle reali difficoltà.

3. INFERENCE E INSTANTIATION DI UN CONTATORE

Una volta creato un codice VHDL, ad esempio di un contatore, il processo di Inference utilizzerà i componenti di libreria che ritiene più appropriati per la sua realizzazione. Il risultato ottenuto, anche se sono state utilizzate catene di carry, non è sempre ottimale.



Le prestazioni possono crescere se si ricorre all'istanziamento del contatore desiderato, se presente nella libreria DesignWare.

```
LIBRARY ieee, DW03;
USE ieee.std_logic_1164.all;
USE DW03.DW03_components.all;

ENTITY updn_32 is
  PORT ( d      : in std_logic_vector(32-1 downto 0);
        up_dn, ld, ce, clk, rst : in std_logic;
        tercnt  : out std_logic;
        q      : out std_logic_vector(32-1 downto 0));
END updn_32;

ARCHITECTURE structure OF updn_32 IS
BEGIN
  u1: DW03_updn_ctr
  GENERIC MAP (width => 32)
  PORT MAP (data => d, clk => clk, reset => rst, up_dn => up_dn,
           load => ld, tercnt => tercnt, cen => ce, count => q);
END structure;
```

Un modo alternativo è quello di istanziare lo stesso contatore con l'utilizzo delle LPM. Il contatore viene posto nel codice come una black box. Il tool di sintesi riceve dei vincoli che impediscono la sintesi di questo modulo. La netlist risultante viene letta dal Fitter insieme alla netlist del contatore prodotta dal programma che crea l'LPM.

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

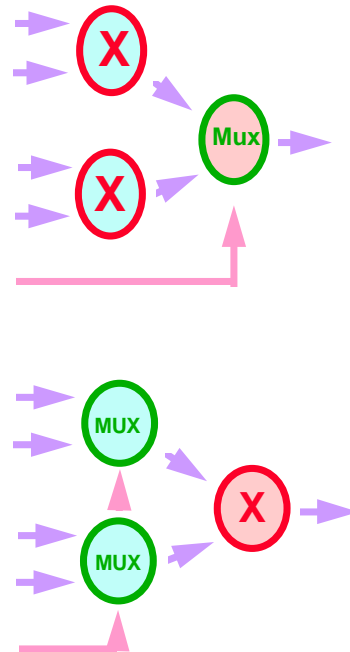
entity count16 is
port (clk, updn, ld, reset, cnten: in std_logic;
      din  : in std_logic_vector(15 downto 0);
      dout : out std_logic_vector(15 downto 0));
end count16;

architecture logic of count16 is
component cnt16
  PORT
  (
    clock : in std_logic;
    q     : out std_logic_vector (15 downto 0);
    updown: in std_logic;
    cnt_en: in std_logic;
    sload : in std_logic;
    aclr  : in std_logic;
    data  : in std_logic_vector (15 downto 0)
  );
end component;
begin
ucnt16: cnt16 port map(clk, dout, updn, cnten, ld, reset, din);
end logic;
```

4. RESOURCE SHARING

La resource sharing è una tecnica di ottimizzazione, adottata dal tool di sintesi, al fine di usare un singolo blocco funzionale, come comparatore, moltiplicatore o addizionatore, per implementare diversi operatori presenti all'interno di uno stesso processo del codice VHDL. Questa tecnica consente di ridurre l'area occupata dal processo e di ridurre la congestione del routing.

```
process(sel, in1, in2, in3, in4)
begin
  if sel='1' then
    q <= in1 * in2;
  else
    q <= in3 * in4;
  end if;
end process;
```



Le performance, in termini di velocità sono penalizzate dal fatto che viene aggiunto un livello di logica per la selezione di ingressi provenienti da più parti per realizzare più di una funzione.

Poichè la resource sharing è automaticamente evocata dal processo di sintesi, essa va disabilitata, quando occorre, con il comando *Hdl_resource_allocation=none*. Un modo alternativo di procedere è quello di creare processi separati, di numero coincidente alle repliche dello stesso operatore presenti all'interno del processo iniziale.

```
PARTE_A:process(in1, in2) begin
  qa <= in1 * in2;
end process PARTE_A;

PARTE_B:process(in3, in4) begin
  qb <= in3 * in4;
end process PARTE_B;

q <= qa when (SEL='1') else qb;
```

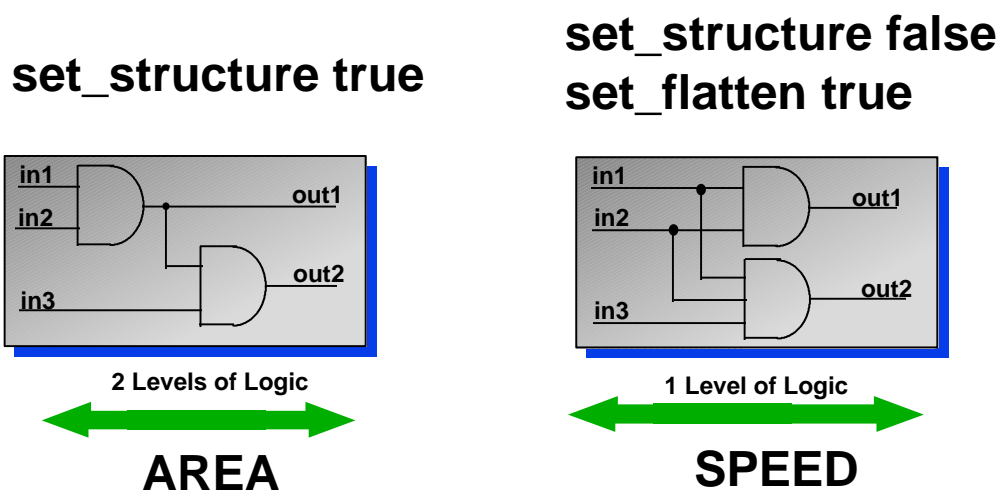
5. STRUCTURE E FLATTEN

La sintesi del codice, detta anche compilazione o ottimizzazione, viene eseguita in base ad opportune direttive. Queste influenzano il tipo di risultato ottenibile. Una sintesi può essere timing driven oppure orientata all'ottimizzazione booleana, può costruire la logica privilegiando dei percorsi (structure) o essere priva di priorità (flatten).

Una sintesi con direttiva structure privilegia l'area minima ottenibile. Una sintesi con direttiva flatten è mirata ad avere buone performance in termini di velocità

Nella sintesi basata sulle look-up table, non sempre la direttiva flatten produce un numero di livelli di logica inferiore a quello ottenibile con direttiva structure; anzi, in certi casi, si ottiene lo stesso numero di livelli di logica con un numero di look-up table superiore. Questo produce una netlist che occupa maggiori risorse nella logica programmabile sia in termini di celle logiche che in termini di risorse di routing.

Nella sintesi basata sulle porte logiche, la direttiva flatten produce un insieme di equazioni basate sulla somma di prodotti. Può essere usata, con le dovute attenzioni all'area prodotta dalla sintesi, per la compilazione di progetti orientati a PLD.



6. SEGNALI E VARIABILI

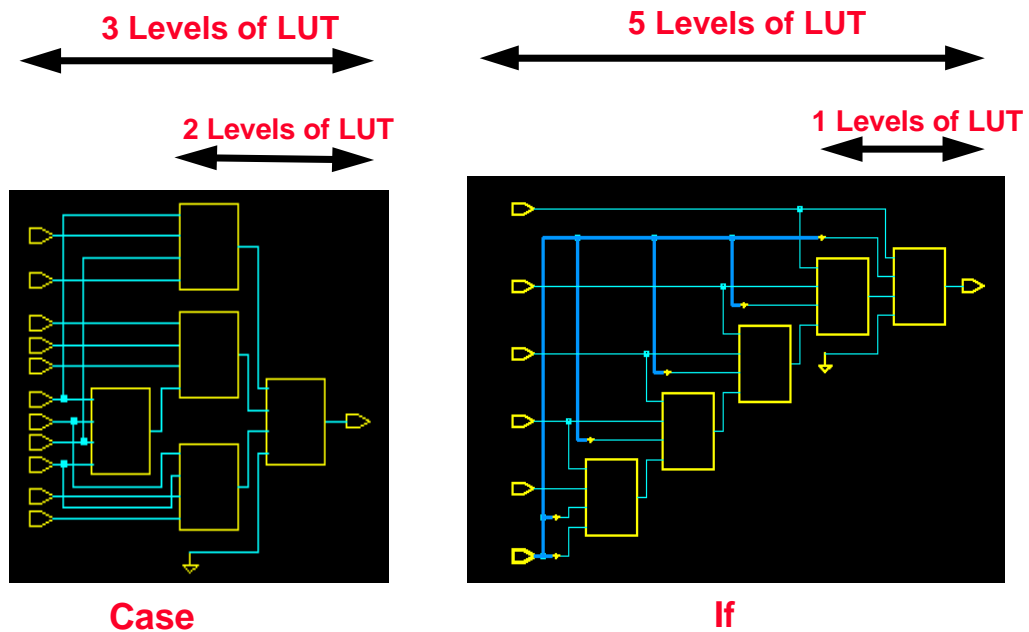
L'utilizzo sia dei segnali che delle variabili nei processi non ha particolari limitazioni. I segnali sono molto vicini all'hardware e sono aggiornati solo alla fine dei processi. Le variabili, invece, sono aggiornate immediatamente. Esse però possono mascherare dei glitch in simulazione che a loro volta possono alterare il funzionamento della logica. L'uso dei segnali permette una sintesi sicura, quello delle variabili una simulazione veloce.

7. IF E CASE

Il case e l'if sono due costrutti messi a disposizione dal linguaggio VHDL. Il primo produce una logica con priorità sui segnali coinvolti, il secondo crea una logica parallela. Un if può contenere un insieme di differenti espressioni ed è destinato alla realizzazione di percorsi prioritari. Il case è considerato in base ad una comune espressione e viene usato per decodifiche complesse. A parità di espressione da analizzare, il case richiede un minor numero di livelli di logica rispetto ad un if annidato.

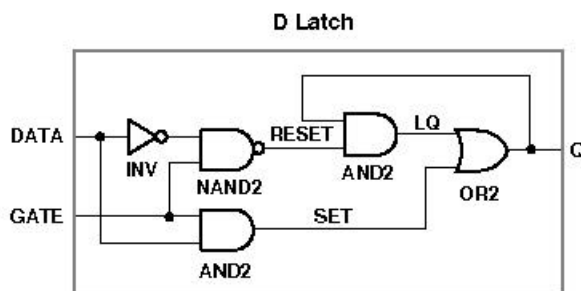
```
process(sel, d)
begin
  case sel is
    when "00001" => y <= d(0);
    when "00010" => y <= d(1);
    when "00100" => y <= d(2);
    when "01000" => y <= d(3);
    when "10000" => y <= d(4);
    when others => y <= d(5);
  end case;
end process;
```

```
process(sel, d)
begin
  if sel(0)='1' then
    y <= d(0);
  elsif sel(1)='1' then
    y <= d(1);
  elsif sel(2)='1' then
    y <= d(2);
  elsif sel(3)='1' then
    y <= d(3);
  elsif sel(4)='1' then
    y <= d(4);
  else
    y <= d(5);
  end if;
end process;
```

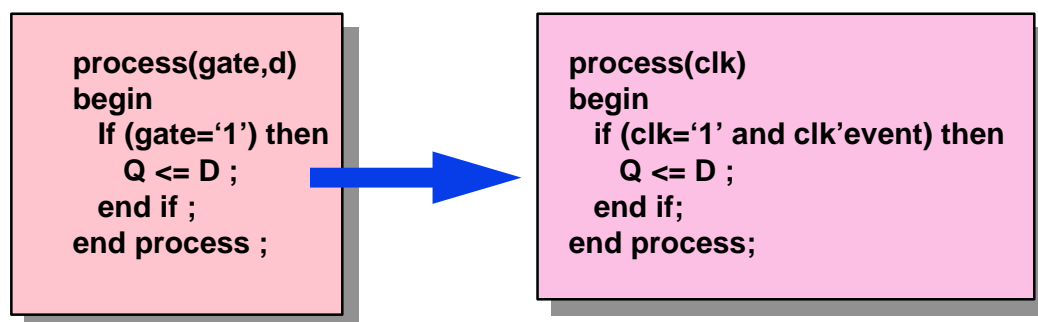


8. IMPLEMENTAZIONE DI UN LATCH

Il compilatore, se non incontra un latch istanziato, può crearne uno con logica sparsa. Questo produce un loop di logica combinatoria e può condurre alla perdita di identità del latch nella sintesi, essendo inglobato in altra logica dal processo di ottimizzazione gerarchica. Utilizzando il comando `hdlin_check_no_latch="TRUE"` è possibile stabilire se nella sintesi sono stati creati latch, magari non voluti, per un' incompleta descrizione di un costrutto di tipo if.



Nelle logiche programmabili, ove il latch non sia supportato come componente, è opportuno usare in sua sostituzione un flip flop.



9. INSERIMENTO DI BUFFER GLOBALI

Le logiche programmabili sono dotate di linee globali preposte al trasporto di segnali con elevato carico quali reset, clock o enable. Il tool di sintesi inserisce automaticamente i buffer di clock. Per l'utilizzo di una linea globale destinata al trasporto del segnale di reset e quello di enable, quando né il sintetizzatore né il Fitter vi siano riusciti, occorre istanziare il buffer opportuno nel codice.

Esistono degli FPGA che hanno a disposizione diversi tipi di buffer di clock. Il sintetizzatore in questo caso sceglierà quello che ritiene più opportuno. Volendo imporre il tipo di buffer da utilizzare, è possibile agire in due modi equivalenti: istanziarlo nel codice VHDL oppure controllare il processo di inserzione dei pad prima della sintesi.

Nel primo caso sono richiesti, prima della direttiva di compilazione, i seguenti comandi:

```
set_port_is_pad "*"
```

```
remove_attribute (tipo_di_buffer_istanziato) port_is_pad
```

insert_pads

Nel secondo caso sono richiesti, prima della direttiva di compilazione, i seguenti comandi:

```
set_port_is_pad "*"
```

```
set_pad_type -exact tipo_di_buffer (port_list)
```

insert_pads

Nel codice che segue si evidenzia l'uso di un buffer globale per diramare un segnale di enable e di un buffer specifico, per diramare un segnale di clock. Inoltre si fa uso di un componente per gestire il segnale di reset.

In generale, l'utilizzo di una risorsa di tipo globale per la diramazione del segnale di reset consente di ridurre la congestione del routing e di incrementare le prestazioni totali del componente. In alcuni FPGA, il Fitter collega automaticamente tutti gli elementi sequenziali alla linea precostituita di reset globale; in altri, l'uso del global reset avviene istanziando nel codice un componente che pilota tale linea e che prende il nome di STARTUP o STARTBUF.

Il componente STARTUP viene riconosciuto dal Fitter ma non dal simulatore, per cui se si vuole compiere una simulazione, occorre inserire una seconda net di reset collegata a tutti i processi e poi disconnetterla alla fine del processo di sintesi e prima della scrittura della netlist con il comando *disconnect_net RESET -all*.

Il componente STARTBUF, inserendo nel codice il link alla libreria unisim per le simulazioni funzionali o alla libreria simprim per quelle gate level, permette sia di simulare il codice che di essere riconosciuto dal Fitter. Infatti, a differenza del componente STARTUP non dotato di uscite, STARTBUF ha un'uscita che può essere collegata al reset di tutti i processi del codice in esame. L'ingresso di questo componente è il reset che viene osservato dall'esterno del progetto implementato. Anche in questo caso è necessario il comando *disconnect_net RESET -all* al fine di evitare la generazione di una duplice linea di reset (globale e di utente) da parte del Fitter.

```
Library IEEE;
```

```
use IEEE.std_logic_1164.all;  
use IEEE.std_logic_misc.all;  
use IEEE.std_logic_arith.all;  
use IEEE.std_logic_components.all;  
use IEEE.std_logic_unsigned.all;
```

```
entity SYNC_BLOCK is  
Port ( CLK : In std_logic;  
      CLKEXT: In std_logic;  
      EXT_RES : In std_logic;  
      SYNC : In std_logic;  
      SYNC_AUS : In std_logic;  
      EN : Out std_logic_vector (15 downto 0));  
end SYNC_BLOCK;
```

```
architecture BEHAVIORAL of SYNC_BLOCK is  
signal COUNT,SUB_COUNT: std_logic_vector (3 downto 0);  
signal INV_RESET : std_logic;  
signal ENABLE,CLKAUS : std_logic;  
signal VCC: std_logic;
```

```
component BUFGP_F  
port( I : in std_logic;
```

```

        O : out std_logic );
end component;

component BUFG
port( I : in std_logic;
      O : out std_logic );
end component;

component STARTUP
port(GSR: in std_logic);
end component;

component INV
port( I : in std_logic;
      O : out std_logic);
end component;

begin

U0: INV port map (I=>EXT_RES, O=>INV_RESET);

U1: STARTUP port map(GSR=>INV_RESET);

MYBUFG : BUFG port map( I=>SYNC, O=>ENABLE );

AUS_BUF : BUFGP_F port map( I=>CLKEXT, O=>CLKAUS );

CONTATORE_AUSILIARIO:process(CLK) begin
if (CLK'event and CLK='1') then
    if (ENABLE='1') then
        SUB_COUNT<="1110";
    elsif (SUB_COUNT=0) then
        SUB_COUNT<= "1111";
    else
        SUB_COUNT<= SUB_COUNT-1;
    end if;
end if;
end process CONTATORE_AUSILIARIO;

ABILITAZIONI_EXT: process(CLK) begin
if (CLK'event and CLK='1') then
    for I in 0 to 15 loop
        if (SUB_COUNT=I) then
            EN(I)<= '0';
        else
            EN(I)<= '1';
        end if;
    end loop;
end if;
end process ABILITAZIONI_EXT;

TEMPORIZZAZIONE: process(CLKAUS) begin
if (CLKAUS'event and CLKAUS='1') then
    COUNT<= COUNT+1;
    if (COUNT=1) then
        SYNC_AUS<= '1';
    else
        SYNC_AUS<= '0';
    end if;
end if;
end process TEMPORIZZAZIONE;

end BEHAVIORAL;

```

Nel successivo esempio si mostra l'utilizzo del segnale di reset globale prodotto dal blocco simulabile STARTBUF.

```
Library IEEE;
  use IEEE.std_logic_1164.all;
  use IEEE.std_logic_misc.all;
  use IEEE.std_logic_arith.all;
  use IEEE.std_logic_components.all;
  use IEEE.std_logic_unsigned.all;
-- pragma translate_off
Library UNISIM;
  use UNISIM.all;
-- pragma translate_on

entity SYNC_BLOCK_GR is
Port (  CLK : In  std_logic;
      EXT_RES : In  std_logic;
      SUB_COUNT : BUFFER  std_logic_vector (5 downto 0));
end SYNC_BLOCK_GR

architecture BEHAVIORAL of SYNC_BLOCK_GR is
signal  INV_RESET,RESET : std_logic;

  component STARTBUF
    port(GSRIN: In  std_logic;
        GSROUT: Out std_logic);
  end component;

  component INV
    port( I : in std_logic;
        O : out std_logic);
  end component;

begin

  U0: INV port map (I=>EXT_RES, O=>INV_RESET);

  U1: STARTBUF port map(GSRIN=>INV_RESET, GSROUT=> RESET);

  CONTATORE_AUSILIARIO:process(CLK,RESET) begin
  if (RESET='0') then
    SUB_COUNT<= (others=>'0');
  elsif (CLK'event and CLK='1') then
    SUB_COUNT<= SUB_COUNT+1;
  end if;
end process CONTATORE_AUSILIARIO;
```

Il blocco STARTBUF, così come STARTUP, non deve essere ottimizzato. Per cui, al nome del componente istanziato e prima del comando di sintesi, va aggiunto l'attributo *set_dont_touch cell_instance_name* (nei due esempi precedenti *set_dont_touch U1*).

10. IMPLEMENTAZIONE DI UN MUX CON BUFFER TRISTATE

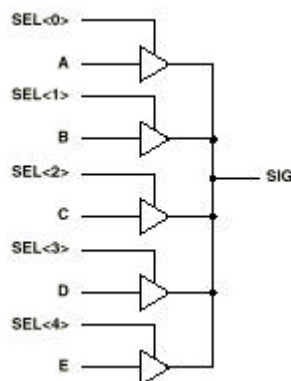
Un multiplexer 2 a 1 è implementato in maniera efficiente in una look-up table, uno 4 a 1 in un elemento logico del tipo CLB. Quando le dimensioni iniziano a crescere, il numero delle celle logiche cresce e con esse il numero dei livelli di logica che incontrano i segnali nella loro propagazione. Ad esempio, se si vuole ottenere un mux 16 a 1, occorreranno due livelli di logica e cinque CLB.

Esistono delle famiglie di FPGA o PLD in cui è possibile avere buffer tristate interni al componente. Questi buffer pilotano, a loro volta, una linea che può essere globale. Ciascun elemento del mux da realizzare alimenta l'ingresso di un buffer tristate. Ciascuna decodifica dei segnali di selezione pilota uno ed un solo ingresso di enable di buffer.

La realizzazione di un mux in questo modo ha il vantaggio di avere un impatto minimo al variare delle dimensioni del mux da realizzare, sia in termini di area richiesta che in termini di ritardo di propagazione. I limiti al numero di ingressi sono rappresentati dal numero di buffer tristate che possono pilotare una singola linea ad alto fan-out. L'assenza di contese è invece garantita da una separata decodifica di abilitazione compiuta buffer per buffer.

```
Library IEEE;  
  use IEEE.std_logic_1164.all;  
  use IEEE.std_logic_arith.all;  
  
entity MUX_TBUF is  
  port( SEL: In std_logic_vector (4 downto 0);  
        A,B,C,D,E: In std_logic;  
        SiG : Out std_logic);  
end MUX_TBUF;
```

```
architecture BEHAVIORAL of MUX_TBUF is  
begin  
  SIG<= A when (SEL(0)='1') else 'Z';  
  SIG<= B when (SEL(1)='1') else 'Z';  
  SIG<= C when (SEL(2)='1') else 'Z';  
  SIG<= D when (SEL(3)='1') else 'Z';  
  SIG<= E when (SEL(4)='1') else 'Z';  
end BEHAVIORAL;
```



11. ISTANZIAZIONE DI UN UNBONDED I/O

In alcuni FPGA, non tutti i pad sono associati ai pin del package. Questi pad non connessi possono comunque essere sfruttati dalla logica di utente riutilizzandone i due flip flop, uno di uscita e uno di ingresso. I sintetizzatori di logica non inferenziano flip flop di unbonded I/O, per cui l'unica via per usarli è di istanziarli.

Nell'esempio seguente si mostra l'utilizzo di tali componenti per creare un elemento di ritardo di due colpi di clock.

```
Library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_misc.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_components.all;
use IEEE.std_logic_unsigned.all;

entity BISHIFT_UN is
port(
    Q : out std_logic_vector(7 downto 0); -- dati ritardati
    C : in std_logic; -- segnale di clock
    CE : in std_logic; -- enable del ritardo
    D : in std_logic_vector(7 downto 0)); -- dati in ingresso
end BISHIFT_UN;

architecture BEHAVIORAL of BISHIFT_UN is
signal INT: std_logic_vector(7 downto 0);

component IFDX_U -- flip flop unbonded di input
port(
    Q : out std_logic;
    C : in std_logic;
    CE : in std_logic;
    D : in std_logic);
end component;

component OFDX_FU -- flip flop unbonded di output
port(
    Q : out std_logic;
    C : in std_logic;
    CE : in std_logic;
    D : in std_logic);
end component;

begin

FLIP_0: OFDX_FU port map (Q=>INT(7),C=>C,CE=>CE,D=>D(7));
FLOP_0: IFDX_U port map (Q=>Q(7),C=>C,CE=>CE,D=>INT(7));
FLIP_1: OFDX_FU port map (Q=>INT(6),C=>C,CE=>CE,D=>D(6));
FLOP_1: IFDX_U port map (Q=>Q(6),C=>C,CE=>CE,D=>INT(6));
FLIP_2: OFDX_FU port map (Q=>INT(5),C=>C,CE=>CE,D=>D(5));
FLOP_2: IFDX_U port map (Q=>Q(5),C=>C,CE=>CE,D=>INT(5));
FLIP_3: OFDX_FU port map (Q=>INT(4),C=>C,CE=>CE,D=>D(4));
FLOP_3: IFDX_U port map (Q=>Q(4),C=>C,CE=>CE,D=>INT(4));
FLIP_4: OFDX_FU port map (Q=>INT(3),C=>C,CE=>CE,D=>D(3));
FLOP_4: IFDX_U port map (Q=>Q(3),C=>C,CE=>CE,D=>INT(3));
FLIP_5: OFDX_FU port map (Q=>INT(2),C=>C,CE=>CE,D=>D(2));
FLOP_5: IFDX_U port map (Q=>Q(2),C=>C,CE=>CE,D=>INT(2));
FLIP_6: OFDX_FU port map (Q=>INT(1),C=>C,CE=>CE,D=>D(1));
FLOP_6: IFDX_U port map (Q=>Q(1),C=>C,CE=>CE,D=>INT(1));
FLIP_7: OFDX_FU port map (Q=>INT(0),C=>C,CE=>CE,D=>D(0));
FLOP_7: IFDX_U port map (Q=>Q(0),C=>C,CE=>CE,D=>INT(0));

end BEHAVIORAL;
```

12. ISTANZIAZIONE DI UNA MEMORIA

In una logica programmabile la memoria può essere concentrata o distribuita. Può essere usata per implementare registri di stato, memorizzazione dello stato di contatori, shift register, LIFO stack, FIFO buffer. In ogni caso va istanziata. Se si parla di ram distribuita 16x1 (single o dual port) o 32x1 (single port) esistono in libreria moduli istanziabili. Se si parla di ram distribuita di dimensione generica, occorrerà istanziare il componente prodotto da programmi dedicati allo scopo. Nel caso di ram a blocchi (la cui dimensione per blocco è tipicamente 4096x1, 2048x2, 1024x4, 512x8, 256x16 oppure 2048x1, 1024x2, 512x4, 256x8, 128x16) è possibile sia istanziare la memoria di dimensione opportuna (in questo caso esisterà un nome identificativo del tipo e delle dimensioni), sia istanziarne una prodotta da un programma specifico.

Nel caso di istanziazione di memoria di tipo non preconstituito in libreria, il componente che la caratterizza può essere ricavato da programmi orientati alla generazione di memorie o da programmi del tipo Core generation o LPM.

Nel seguente esempio si produce con il programma genmem e poi si istanzia nel codice `ram_test` una memoria denominata `syn_ram_256x16_irou`. L'utility genmem crea tre tipi di file: un file di estensione `cmp` che contiene il template del componente da istanziare nel codice VHDL che richiama la memoria, un file di estensione `vhd` che contiene il modello simulabile da un punto di vista funzionale, un file di estensione `lib` che contiene il modello per la simulazione timing.

```
genmem memory_type memory_size [-vhd] [-verilog]
```

ove:

```
memory_type: ASYNRAM   Asynchronous RAM
              ASYNROM   Asynchronous ROM
              SYNRAM    Synchronous RAM
              SYNROM    Synchronous ROM
              ASYNDRAM  Asynchronous Dual_port RAM
              SYNDPRAM  Synchronous Dual_port RAM
              CSDPRAM   Cycle-Shared Dual_port RAM
              CSFIFO    Cycle-Shared FIFO
              SCFIFO    Single-clock FIFO
              DCFIFO    Dual-clock FIFO
```

memory_size: Depth x Width (ad esempio 256 x 8)

[-vhd] [-verilog]: Switch tra i formati di generazione

```
architecture ram_test of ram_test is
component syn_ram_256x16_irou          -- Dichiarazione del componente prelevata dal file
.cmp
-- pragma translate_off
  generic (LPM_FILE : string);
-- pragma translate_on
  port ( data   : in std_logic_vector (15 downto 0);
        address : in std_logic_vector (7  downto 0);
        we     : in std_logic;
        q      : out std_logic_vector (15 downto 0);
        inclock : in std_logic
        );
end component;
begin
  u1: syn_ram_256x16_irou          -- Istanziamento del modulo di memoria
-- pragma translate_off
  generic map (LPM_FILE => "UNUSED") -- Specificazione del file di inizializzazione
```

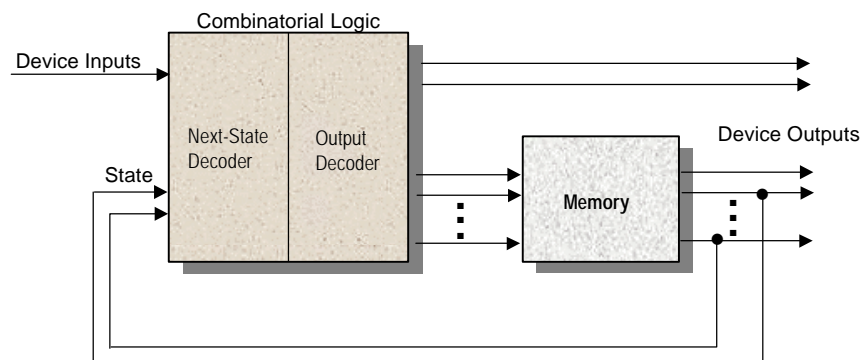
```

-- pragma translate_on                                -- (Intel .hex format)
    port map (address => addr, data => din, we => we , q => dout, inclock => clk);
end ram_test;

```

13. PROGETTO DI MACCHINE A STATI

Una macchina a stati è un dispositivo digitale che evolve attraverso una determinata sequenza di stati. Viene comunemente usata in applicazioni di tipo monitoraggio di eventi, risoluzioni di contese, generazione di segnali di controllo, test di eventi multipli. È realizzata in due sezioni di cui una implementa logica combinatoria e l'altra logica sequenziale costituita, a sua volta, da registri di stato ed eventualmente da registri di uscita. Lo stato in cui la macchina a stati può trovarsi è identificato dai valori memorizzati nei registri di stato e, in certi casi, anche nei registri di uscita. La sezione combinatoria può essere a sua volta distinta in due blocchi funzionali denominati rispettivamente decodificatore dello stato successivo e decodificatore di uscita. Il primo blocco, denominato anche transition function, determina il successivo stato in cui deve evolvere la macchina a stati. Il secondo, noto anche come function, genera le uscite opportune.



Queste due operazioni si basano sul valore degli ingressi (operazione definita anche branching) e dello stato presente. La transizione da uno stato all'altro è invece chiamata sequenza di controllo. L'uso o meno dei segnali di ingresso nella produzione dei segnali di uscita classifica le macchine a stati in due categorie. Se le uscite sono funzioni del solo stato presente si parla di macchina di Moore. Se le uscite sono realizzate anche con il contributo degli ingressi si parla di macchina di Mealy. Esistono anche delle macchine a stati di tipologia ibrida in cui alcune uscite dipendono dagli ingressi e altre no.

La rappresentazione del modello evolutivo di una macchina a stati può avvenire in tre modi: mediante il diagramma di stato detto anche diagramma di transizione di stato, mediante una tabella delle transizioni di stato, mediante un flowchart.

Nella descrizione VHDL di una macchina a stati non occorre realizzare, come avviene con le metodologie tradizionali, l'assegnamento degli stati, la generazione della tabella di transizione, la determinazione delle equazioni dello stato successivo basate sul tipo di flip flop disponibili, la determinazione del tipo di uscite (se di Mealy o di Moore).

Il progetto può essere diviso in due processi: uno, di tipo combinatorio, che calcola il prossimo stato, l'altro che funge da elemento di memoria per quello che nello stato seguente sarà considerato lo stato presente. Le uscite saranno a loro volta incluse nella parte combinatoria, in quella sequenziale o in entrambi. Il processo preposto al calcolo dello stato successivo è basato su un costrutto di tipo case-when. La transizione tra uno stato e il successivo è basata su un costrutto di tipo if-then-else.

È possibile adottare un processo unico che non produce alterazioni funzionali ma richiede attenzione nella descrizione delle uscite in ogni ramificazione delle condizioni di if e una complicazione nella comprensione del codice. Inoltre può generare confusione nell'interpretazione dei segnali di uscita che devono essere definiti per il successivo stato e non per il corrente.

Nel progetto di una macchina a stati è opportuno prendere delle precauzioni al fine di evitare un funzionamento improprio. È opportuno sia evitare che ci siano stati raggiungibili da cui non è possibile uscire, sia avere uno stato di partenza definito al power-up, sia esplicitare il comportamento della macchina per tutti i possibili stati non codificati che il sintetizzatore VHDL potrebbe produrre. Nel caso in cui sia previsto un reset che intervenga in fase di inizializzazione o per un malfunzionamento, è opportuno che sia asincrono. Questo tipo di reset, tra le altre cose citate, riduce la logica combinatoria che occorrerebbe per l'implementazione del reset sincrono, quest'ultimo usato quando è parte integrante dell'evoluzione della macchina.

Area e velocità sono direttive in contrasto tra loro. Alcune implementazioni possono richiedere più registri che logica combinatoria (generalmente migliori per gli FPGA), altre possono richiedere più product term che registri (generalmente migliori per PLD).

La struttura basata su due processi, di cui uno incorpora i registri e l'altro la parte combinatoria di dimensioni non trascurabili, è ben sfruttata da una PLD, la cui architettura ha pochi registri ma sezioni di logica combinatoria di potenzialità superiori a quelle ottenibili con una struttura basata sulle look-up table. Ciò non vieta l'uso di una simile struttura anche in FPGA ove si può ricorrere a tre diverse metodologie di codifica: binary, enumerated, one_hot.

Nella codifica binaria si utilizza un attributo per esplicitare la codifica dello stato. Nella codifica di tipo enumerativo il tipo di codifica non viene esplicitato. Sarà compito del sintetizzatore, attraverso il comando Finite State Machine extraction (FSM extraction), realizzare la minimizzazione degli stati e l'ottimizzazione della macchina. Questa estrazione può essere a sua volta di tipo one_hot, case implemented o if implemented. Nella codifica one_hot l'attributo specifica l'utilizzo di un solo registro per ogni stato.

La codifica one_hot, in cui ad ogni stato viene associato un registro, permette di creare implementazioni efficienti per il tipo di architettura considerata anche se introduce un elevato numero di potenziali stati illegali. È il tipo di codifica preferito per macchine con un elevato numero di stati. Per un numero di stati contenuto (al massimo 8) la codifica binaria può essere più efficiente della codifica one_hot. Quando si codifica descrivendo un elevato numero di stati, ad esempio superiore a 32, al fine di aumentare le prestazioni è possibile dividere la macchina a stati in macchine più piccole in cui, a seconda dei casi, si applica la codifica più opportuna.

```
Library IEEE;
  use IEEE.std_logic_1164.all;
  use IEEE.std_logic_misc.all;
  use IEEE.std_logic_arith.all;
  use IEEE.std_logic_components.all;
  use IEEE.std_logic_unsigned.all;
```

```
entity STATE_MACHINE is
  Port ( READY : In std_logic;
        CLK : In std_logic;
        RESET : In std_logic;
        RW : In std_logic;
```

```

        WE : Out std_logic;
        OE : Out std_logic );
end STATE_MACHINE;

architecture BEHAVIORAL of STATE_MACHINE is
-- for binary encoding
type STATE_TYPE is (S1,S2,S3,S4);
attribute ENUM_ENCODING: STRING;
attribute ENUM_ENCODING of STATE_TYPE : type is "00 01 10 11";

-- for enumerated encoding
--type STATE_TYPE is (S1,S2,S3,S4);

-- for one_hot encoding
--type STATE_TYPE is (S1,S2,S3,S4);
--attribute ENUM_ENCODING: STRING;
--attribute ENUM_ENCODING of STATE_TYPE : type is "0001 0010 0100 1000";

-- for all styles
signal PRESENT_STATE,NEXT_STATE: STATE_TYPE;
begin

LOGICA_COMBINATORIA: process (PRESENT_STATE,READY,RW) begin
    case PRESENT_STATE is
        when S2 =>
            OE<= '0';
            WE<= '0';
            if (RW='1') then
                NEXT_STATE<= S4;
            else
                NEXT_STATE<= S3;
            end if;
        when S3 =>
            OE<= '0';
            WE<= '1';
            if (READY='1') then
                NEXT_STATE<= S1;
            else
                NEXT_STATE<= S3;
            end if;
        when S4 =>
            OE<= '1';
            WE<= '0';
            if (READY='1') then
                NEXT_STATE<= S1;
            else
                NEXT_STATE<= S4;
            end if;
        when others =>
            OE<= '0';
            WE<= '0';
            if (READY='1') then
                NEXT_STATE<= S2;
            else
                NEXT_STATE<= S1;
            end if;
    end case;
end process LOGICA_COMBINATORIA;

LOGICA_SEQUENZIALE: process (RESET,CLK) begin
    if (RESET = '0') then
        PRESENT_STATE<= S1;
    elsif (CLK'event and CLK= '1') then
        PRESENT_STATE<= NEXT_STATE;
    end if;
end process LOGICA_SEQUENZIALE;

```

end BEHAVIORAL;

Un confronto tra i tre metodi porta a concludere che la codifica one_hot produce un miglior risultato in termini di velocità rispetto alla codifica binaria, a discapito del numero di registri impiegati. L'utilizzo del comando FSM extraction, opportunamente indirizzato verso la codifica one_hot, riduce le risorse occupate rispetto alla pura codifica one_hot, anche se produce un numero superiore di registri rispetto alla codifica binaria. Questo accade perchè il sintetizzatore riduce ogni stato ridondante e ottimizza la macchina dopo l'operazione di estrazione.

Non sempre l'obiettivo del progetto è ottenere una implementazione ottimale. La velocità di descrizione del codice e la sua leggibilità sono superiori, a volte, alle necessità di efficienza in termini di area o velocità. Nei casi in cui si voglia ricorrere ad ulteriori accorgimenti per aumentare le prestazioni, è opportuno rileggere i segnali di uscita e, al fine di ridurre la logica addizionale coinvolta nella produzione delle uscite, decodificarli prima che i bit di stato siano riletti.

```
Library IEEE;
  use IEEE.std_logic_1164.all;
  use IEEE.std_logic_misc.all;
  use IEEE.std_logic_arith.all;
  use IEEE.std_logic_components.all;
  use IEEE.std_logic_unsigned.all;

entity STATE_MACHINE is
  Port ( READY : In  std_logic;
        CLK : In  std_logic;
        RESET : In  std_logic;
        RW : In  std_logic;
        WE : Out  std_logic;
        OE : Out  std_logic );
end STATE_MACHINE;

architecture BEHAVIORAL of STATE_MACHINE is
-- for binary encoding
type STATE_TYPE is (S1,S2,S3,S4);
attribute ENUM_ENCODING: STRING;
attribute ENUM_ENCODING of STATE_TYPE : type is "00 01 10 11";

-- for enumerated encoding
--type STATE_TYPE is (S1,S2,S3,S4);

-- for one_hot encoding
--type STATE_TYPE is (S1,S2,S3,S4);
--attribute ENUM_ENCODING: STRING;
--attribute ENUM_ENCODING of STATE_TYPE : type is "0001 0010 0100 1000";

-- for all styles
signal PRESENT_STATE,NEXT_STATE: STATE_TYPE;

-- for output registered
signal OE_INT,WE_INT: std_logic;
begin

LOGICA_COMBINATORIA: process (PRESENT_STATE,READY,RW) begin
  case PRESENT_STATE is
    when S2 =>
      if (RW='1') then
        NEXT_STATE<= S4;
      else
        NEXT_STATE<= S3;
      end if;
  end case;
end process;
```

```

        when S3 =>
            if (READY='1') then
                NEXT_STATE<= S1;
            else
                NEXT_STATE<= S3;
            end if;
        when S4 =>
            if (READY='1') then
                NEXT_STATE<= S1;
            else
                NEXT_STATE<= S4;
            end if;
        when others =>
            if (READY='1') then
                NEXT_STATE<= S2;
            else
                NEXT_STATE<= S1;
            end if;
    end case;
end process LOGICA_COMBINATORIA;

OE_INT<= '1' when (NEXT_STATE=S3) else '0';
WE_INT<= '1' when (NEXT_STATE=S4) else '0';

LOGICA_SEQUENZIALE: process (RESET,CLK) begin
    if (RESET = '0') then
        PRESENT_STATE<= S1;
        OE<= '0';
        WE<= '0';
    elsif (CLK'event and CLK= '1') then
        PRESENT_STATE<= NEXT_STATE;
        OE<= OE_INT;
        WE<= WE_INT;
    end if;
end process LOGICA_SEQUENZIALE;

end BEHAVIORAL;

```

Nelle PLD, ove il numero di registri è una risorsa preziosa, è possibile considerare dei registri di uscita, opportunamente codificati, come registri di stato.

```

Library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_misc.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_components.all;
use IEEE.std_logic_unsigned.all;

entity STATE_MACHINE is
    Port (
        READY : In  std_logic;
        CLK : In  std_logic;
        RESET : In  std_logic;
        RW : In  std_logic;
        WE : Out  std_logic;
        OE : Out  std_logic );
end STATE_MACHINE;

architecture BEHAVIORAL of STATE_MACHINE is
    constant S1: std_logic_vector (2 downto 0):="000";
    constant S2: std_logic_vector (2 downto 0):="001";
    constant S3: std_logic_vector (2 downto 0):="010";
    constant S4: std_logic_vector (2 downto 0):="100";
    signal NEXT_STATE,PRESENT_STATE: std_logic_vector(2 downto 0);
begin

```

```

LOGICA_COMBINATORIA: process (PRESENT_STATE,READY,RW) begin
  case PRESENT_STATE is
    when S2 =>
      if (RW='1') then
        NEXT_STATE<= S4;
      else
        NEXT_STATE<= S3;
      end if;
    when S3 =>
      if (READY='1') then
        NEXT_STATE<= S1;
      else
        NEXT_STATE<= S3;
      end if;
    when S4 =>
      if (READY='1') then
        NEXT_STATE<= S1;
      else
        NEXT_STATE<= S4;
      end if;
    when others =>
      if (READY='1') then
        NEXT_STATE<= S2;
      else
        NEXT_STATE<= S1;
      end if;
  end case;
end process LOGICA_COMBINATORIA;

LOGICA_SEQUENZIALE: process (RESET,CLK) begin
  if (RESET = '0') then
    PRESENT_STATE<= S1;
  elsif (CLK'event and CLK= '1') then
    PRESENT_STATE<= NEXT_STATE;
  end if;
end process LOGICA_SEQUENZIALE;

WE<= PRESENT_STATE(1);
OE<= PRESENT_STATE(2);

end BEHAVIORAL;

```

Nell'esempio si evidenziano due registri che fungono anche da uscite e uno che funge da stato. In questo modo si sono prodotte due uscite sincronizzate al clock e contemporaneamente si sono risparmiati due registri per la decodifica dello stato.

In tutti i casi si è usata una struttura del case in cui, mediante la condizione others, si impone il passaggio allo stato S1. Questo riduce i potenziali banchi per salti a stati non previsti per effetto di rumore o eventi imprevedibili, ma aumenta la logica combinatoria necessaria a decodificare tutti gli stati che potrebbero originarsi dal numero di bit scelti per implementare le variabili di stato. Implementando una condizione di don't care nell'others e una specifica condizione per l'evoluzione dallo stato S1, si ottiene una riduzione della logica combinatoria utilizzata nella macchina a stati a discapito della sua tolleranza ad eventi di guasto.

14. GERARCHIA DI PROGETTO

Un progetto compiuto in VHDL può essere sintetizzato sia mantenendo integra la gerarchia dei blocchi scelta dal progettista, sia compiendo una sintesi con direttiva

flatten. Una corretta suddivisione in blocchi consente una maggiore velocità di simulazione e una più facile ripartizione del lavoro sia tra team di progettisti che in termini di funzionalità implementate. Creando blocchi di dimensioni non superiori ai 500 elementi logici, interfacce sincronizzate da un segnale di clock che, auspicabilmente, dovrebbe essere unico per ogni blocco e, raccogliendo la logica combinatoria di un percorso tutta in un singolo blocco, si riesce ad avere ragionevoli tempi di sintesi e buoni livelli di ottimizzazione rispetto quella flatten. Ulteriori cure vanno poste nella strategia di sintesi dei blocchi critici che vanno compilati entro confini ben precisi e con un costo di sintesi non giustificabile per il resto del progetto.

15. RIFERIMENTI BIBLIOGRAFICI

VHDL for Programmable Logic, Kevin Skahill, Addison-Wesley.

Programmable Logic Handbook, Ashok K. Sharma, McGraw-Hill.

Synopsys Design Compiler, Reference Manual, Synopsys.

Xilinx Synopsys Interface, Xilinx.

VHDL for Synopsys, Altera.