Capitolo 1

Esercizio: descrizione di un sommatore per interi in binario naturale

1.1 Introduzione

In queste note discuteremo alcuni possibili approcci per descrivere il comportamento di un sommatore per interi rappresentati in binario naturale. L'esercizio ha lo scopo di mettere in evidenza alcune funzionalità e peculiarità del linguaggio vhdl. Nella seconda parte dell'esercizio verranno impiegate funzioni per semplificare la descrizione all'interno delle architetture.

1.2 entità sum_nbit

La definizione dell'entità sommatore che vogliamo costruire è riportata nel listing 1.

```
entity sum_nbit is
        generic (ENNE:integer:=4; TP:time:=20 ns);
        port (
                x:in bit_vector(ENNE-1 downto 0);
                y:in bit_vector(ENNE-1 downto 0);
                s:out bit_vector(ENNE-1 downto 0);
                c:out bit
            );
end entity sum_nbit;
```

Listing 1: Dichiarazione dell'entità sommatore.

Con riferimento alla definizione di entità, supporremo che i vettori di bit `x` e `y` siano la rappresentazione in binario naturale su `N` bit di due numeri naturali. Supponiamo inoltre che la l'ordinamento dei bit all'interno dei vettori sia tale che il bit di indice `N-1` rappresenti la cifra più significativa e il bit di indice `0` rappresenti la cifra meno significativa [1].

[1] Si presti attenzione all'uso della locuzione "il bit di indice ... rappresenta la cifra ...". L'uso di questa locuzione serve ad evidenziare il fatto che i "bit" all'interno di un vettore sono oggetti di natura diversa dalle cifre binarie in una rappresentazione di un numero: '1' (elemento del tipo `bit`) *rappresenta* il valore

L'entità deve operare nel modo seguente: ogni volta che cambiano i valori sulle porte di ingresso, sull'uscita deve essere presentato il risultato della somma in binario naturale. Usando una espressione più corretta, in uscita deve essere presentata la *rappresentazione* in binario naturale della somma. Come dovrebbe essere ovvio, la somma fra due numeri natirali rappresentati su `N` bit non è sempre rappresentabile su `N` bit. La somma è però sempre rappresentabile su `N+1` bit. Gli `N+1` bit della rappresentazione della somma sono così riporati in uscita:

- la cifra più significativa (di peso 2^N) viene rappresentata sulla porta di uscita `c` (di tipo bit); in particolare, se la cifra è 1 `c='1'`, se la cifra è 0 `c='0'`;
- le rimanenti `N` cifre sono rappresentate sull'uscita (s); in particolare, la cifra di peso 2^{N-1} è rappresentata dal bit di `s` di indice `N-1`,..., la cifra di peso 2^0 è rappresentata dal bit di `s` di indice `0`.

Il modo di riprodurre il risultato della somma appena descritto, può essere anche interpretato nel modo seguente:

- se si ha come risultato `c='0'`, allora il vettore `s` è la rappresentazione corretta, su `N` bit, della somma fra i numeri di ingresso;
- se si ha come risultato `c='1'` allora la somma dei numeri in ingresso non è rappresentabile su `N` cifre. Il contenuto del vettore `s` è la rappresentazione del numero $(x + y)mod2^N$. Si noti che nell'espressione precedente x e y sono i numeri rappresentati dagli ingressi `x` e `x` e *mod* è l'operazione matematica di modulo.

La dimensione dei vettori di ingresso e di uscita è fissata grazie alla costante generica `TP` consente di stabilire il tempo di propagazione fra un cambiamento dell'ingresso e il conseguente cambiamento dell'uscita.

Occorre chiarire, come dovrebbe essere ovvio, che per il VHDL un vettore di bit è una collezione ordinata di bit senza alcun significato. Siamo noi che *interpretiamo* la stringa di bit, con l'ordine dei bit discusso in precedenza, come la rappresentazione di un intero. In altre parole, il VHDL non prevede [2] alcun meccanismo per ottenere in maniera automatica i bit che rappresentano la somma. Siamo noi a dover ottenere questi bit mediante un opportuno algoritmo che dovremo implementare come parte dell'architettura dell'entità. Gli approcci possibili sono molteplici. Noi ci limiteremo, allo scopo di familiarizzare con il VHDL, a discuterne alcuni.

1.3 Primo approccio

Un possibile approccio per l'implementazione dell'architettura è quello di sfruttare il fatto che in VHDL esiste il tipo predefinito `integer` sul quale è definita l'operazione di somma. Possiamo allora pensare di convertire le stringhe di bit in ingresso in numeri interi, eseguire la somma di questi numeri, e riconvertire il risultato (un intero) in una stringa di bit che costituisce il risultato. Bisogna tenere presente, come già evidenziato in precedenza, che la somma di due numeri naturali rappresentati su `N` bit richiede `N+1` bit per una corretta rappresentazione. Una possibile architettura che opera medinate la conversione in interi

1 (valore che appartiene all'insieme degli interi)

[2] Sarebbe meglio dire che il VHDL "non può prevedere" alcun modo di fare operazioni su stringhe di bit, visto che il significato della stringa stessa è "negli occhi di chi guarda."

dell'ingresso e la successiva conversione del risultato della somma fra interi in una stringa di bit è risportata nel listing 2.

```
1  architecture first of sum_nbit is
2  --converti stringhe di bi in interi, fai la somma e riconverti il numero ottenuto
3  begin
4          process(x,y) is
5          variable xi,yi,si:integer;
6          variable s_appoggio:bit_vector (ENNE downto 0);
7          begin
8          xi:=0;
9          yi:=0;
10         for i in ENNE-1 downto 0 loop
11          xi:=xi*2;
12          yi:=yi*2;
13          if (x(i)='1') then
14           xi:=xi+1;
15          end if;
16          if (y(i)='1') then
17           yi:=yi+1;
18          end if;
19         end loop;
20         --a questi punto xi è il valore di x, yi è il valore di y.
21         si:=xi+yi;
22         --dentro si c'e' la somma.
23         --ora bisogna tirare fuori la rappresentazione in binario naturale di si.
24         for i in 0 to ENNE loop
25                 if ((si mod 2)=1) then
26                         s_appoggio(i):='1';
27                 else
28                         s_appoggio(i):='0';
29                 end if;
30                 si:=si/2;
31         end loop;
32         --facciamo uscire i risultati.
33         s<=s_appoggio(ENNE-1 downto 0) after TP;
34         c<=s_appoggio(ENNE) after TP;
35         end process;
36 end architecture first;
```

Listing 2: Primo approccio.

All'interno dell'unico process contenuto nell'architettura (linea 4) si definiscono tre variabili di tipo intero: `xi,yi,si` (linea 5). Nelle linee da 8 a 19 non si fa altro che ricavare il valore dei numeri interi rappresentati in binario naturale dai vettori di bit `x` e `y`; al termine del ciclo (linea 19), le variabili intere `xi` e `yi` contengono questi valori. Dopo la linea 21, la variabile intera `si` contiene la somma e procediamo a ricavare la rappresentazione in binario (su N+1 bit) che conserviamo temporaneamente nella variabile `s_appoggio` (linee

da 24 a 31). A questo punto abbimo ricavato i valori dei bit da inviare sulle porte di uscita per cui programmiamo gli eventi corrispondenti su `s` e `c` nelle linee 33 3 34).

Possiamo a questo punto predisporre un testbench per procedetre alla simulazione del comporatmento dell'entità. Un sempio di possibile testbench è riportanto nel listing 3.

```
1  entity testbench is
2  end entity testbench;
3
4  architecture behav of testbench is
5  signal s_x,s_y,s_s:bit_vector (3 downto 0);
6  signal s_c:bit;
7  begin
8
9  the_system:entity work.sum_nbit(first)
10         generic map (ENNE=>4, TP=>10 ns)
11         port map (x=>s_x,y=>s_y,s=>s_s,c=>s_c);
12
13         process is
14         begin
15         wait for 100 ns;
16         s_x<="1100";
17         wait for 100 ns;
18         s_y<="0101";
19         wait for 100 ns;
20         s_x<="0011";
21         wait for 100 ns;
22         wait;
23         end process;
24 end architecture behav;
```

Listing 3: Esempio di testbench.

Il testbench nel listing 3 potrà essere usato anche per le successive architetture che verranno esaminate, modificando la linea 9 ed eventualmente modificando anche la sequenza di ingressi predisposta nell'esempio (linee da 13 a 23).

1.4 Secondo approccio

Nel secondo approccio, riportato nel listing 4 lavoriamo direttamente sulle rappresentazioni manipolando i bit di ingressi nello stesso modo in cui faremmo eseguendo la somma con carta e matita e ottenendo direttamente i bit che rappresentano la somma e il riporto. Prima di procedere all'esame del listing 4 si suggerisce di riflettere sull'algoritmo che si esegue con carta e penna per capire il senso delle condizioni logiche che fanno parte dell'algoritmo.

Nell'algoritmo di somma carta e matita otteniamo la cifra i-esima della somma a partire dai valori delle cifra i-esime degli addendi e dal riporto proveniente dall'algoritmo eseguito

sulle cifre (i-1). Ovviamente nel caso della prima cifra (`i=0`), non c'e' riporto, ma per rendere uniforme l'algoritmo su tutte le cifre usiamo lo stesso il riporto che è comunque messo a `'0'` prima di cominciare l'algoritmo di somma (linea 9). AL termine dell'algoritmo di somma (linee da 10 a 29), l'ultimo valore ottenuto per il riporto è la (n+1)-esima cifra della somma. Si noti che non sarebbe stato necessario, in questo caso, dimensionare la variabile `s_appoggio` in modo che essa possa contenere (n+1) bit. Si sarebbe potuto ottenere lo stesso risultato con `s_appoggio` "lungo" solo n, cancellando le linee 30 e 33 e nel listing 4 e utilizzando la linea 34 (ovvero rimuovendo il segno di commento).

1.5 Terzo approccio

Il terzo appoggio in realtà non è altro che una variante del secondo. Fino ad ora, per ottenere il valore da assegnare a un bit abbiamo fatto ricorso al costrutto "`if -esle-en if`". Tuttavia sul tipo predefinito bit sono definite, in vhdl, operazioni logiche che a partire da uno più bit producono un risultato di tipo bit. Per esempio è definita la funzione "`and`" fra bit (da non confondere con la funzione "`and`" fra boolean[3]) che restituisce un valore di tipo bit. Per esempio, con riferimento al listing 5, l'effetto della riga 20 è esattamente lo stesso del blocco di codice VHDL nelle linee da 25 a 29. SI ponga attenzione al fatto che la funzione "`and`" nella riga 20 è una funzione diversa dalla funzione "`and`" che compore nella riga 25: la prima opera su tipi `bit` e produce come risultato un valore del tipo `bit`; la seconda opera su `boolean` e produce valori del tipo `boolean`. E' evidente che, quando possibile, conviene usare direttamente le operazioni logiche predefinite sul tipo bit invece di passare attraverso strutture `if-then-esle`.

Nel terzo approccio, riportato nel listing **??** facciamo appunto uso degli operatori "`and`","`or`" e "`not`" definiti sul tipo bit. Si noti che poiché è anche definito l'operatore "`xor`" le linee da 12 a 16 potrebbero essere sostituite da quanto suggerito al commento nella linea 17.

[3]Il VHDL consente di usare funzioni con lo stesso nume purché queste si riferiscano a "contesti" diversi. Il VHDL discingue qual'è la funzione "and" da usare sulla base del tipo degli argomenti che compaiono a sinistra e a destra della parola chiave "and".

```
architecture second of sum_nbit is
--esegui algoritmo carta e matita
begin

        process(x,y) is
        variable s_appoggio:bit_vector (ENNE downto 0);
        variable r:bit;
        begin
        r:='0';
        for i in 0 to ENNE-1 loop
                if (
                ((x(i)='0')and(y(i)='0')and(r='1'))or
                ((x(i)='0')and(y(i)='1')and(r='0'))or
                ((x(i)='1')and(y(i)='0')and(r='0'))or
                ((x(i)='1')and(y(i)='1')and(r='1'))) then
                        s_appoggio(i):='1';
                else
                        s_appoggio(i):='0';
                end if;

                if (
                ((x(i)='1') and (y(i)='1')) or
                ((x(i)='1') and (r='1')) or
                ((r='1') and (y(i)='1'))) then
                        r:='1';
                else
                        r:='0';
                end if;
        end loop;
        s_appoggio(ENNE):=r;
        --facciamo uscire i risultati.
        s<=s_appoggio(ENNE-1 downto 0) after TP;
        c<=s_appoggio(ENNE) after TP;
        --c<=r after TP;
        end process;
end architecture second;
```

Listing 4: Secondo approccio.

```
.
.
10  variable a,b,c:bit;
.
.
20 c:=a AND b;
.
.
25 if ((a='1')AND(b='1')) then
26      c:='1';
27  else
28      c:='0';
29  end if;
.
.
```

Listing 5: Utilizzo della funzione AND fra bit e fra boolean.

```
 1 architecture third of sum_nbit is
 2 --eseguiamo ancora l'algoritmo carta e matita, ma stavolta
 3 --usiamo le funzioni definite sul tipo bit
 4 begin
 5
 6         process(x,y) is
 7         variable s_appoggio:bit_vector (ENNE downto 0);
 8         variable r:bit;
 9         begin
10         r:='0';
11         for i in 0 to ENNE-1 loop
12                 s_appoggio(i):=(
13                 ((not x(i))and(not y(i))and r) or
14                 ((not x(i))and y(i) and (not r))or
15                 (x(i) and (not y(i)) and (not r))or
16                 (x(i)and y(i) and r));
17                 -- oppure s_appoggio(i)=x(i) xor y(i) xor r;
18
19                 r:=((x(i) and y(i))or(x(i) and r) or (y(i) and r));
20         end loop;
21         s_appoggio(ENNE):=r;
22         --facciamo uscire i risultati.
23         s<=s_appoggio(ENNE-1 downto 0) after TP;
24         c<=s_appoggio(ENNE) after TP;
25         --c<=r after TP;
26         end process;
27 end architecture third;
```

Listing 6: Terzo approccio.