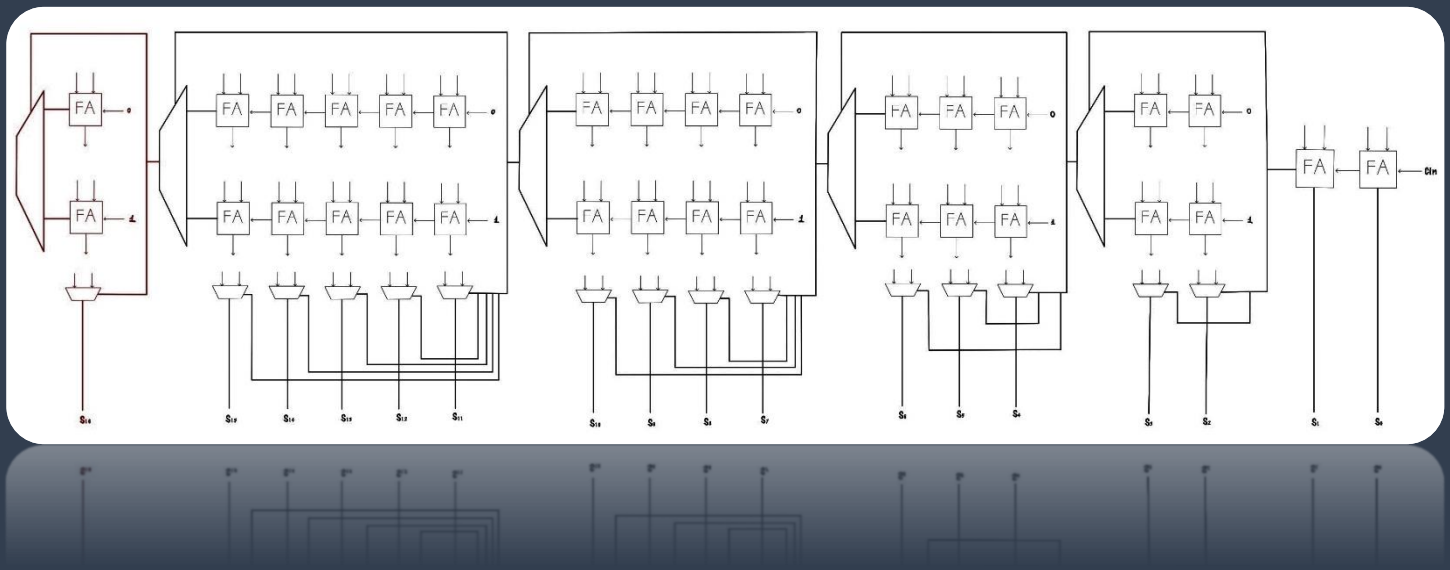
UNIVERSITÀ DELLA CALABRIA

ELETTRONICA DIGITALE A.A. 20-21

RELAZIONE PROGETTO

Marco Greco | Ma200901 | CdL Ingegneria Informatica

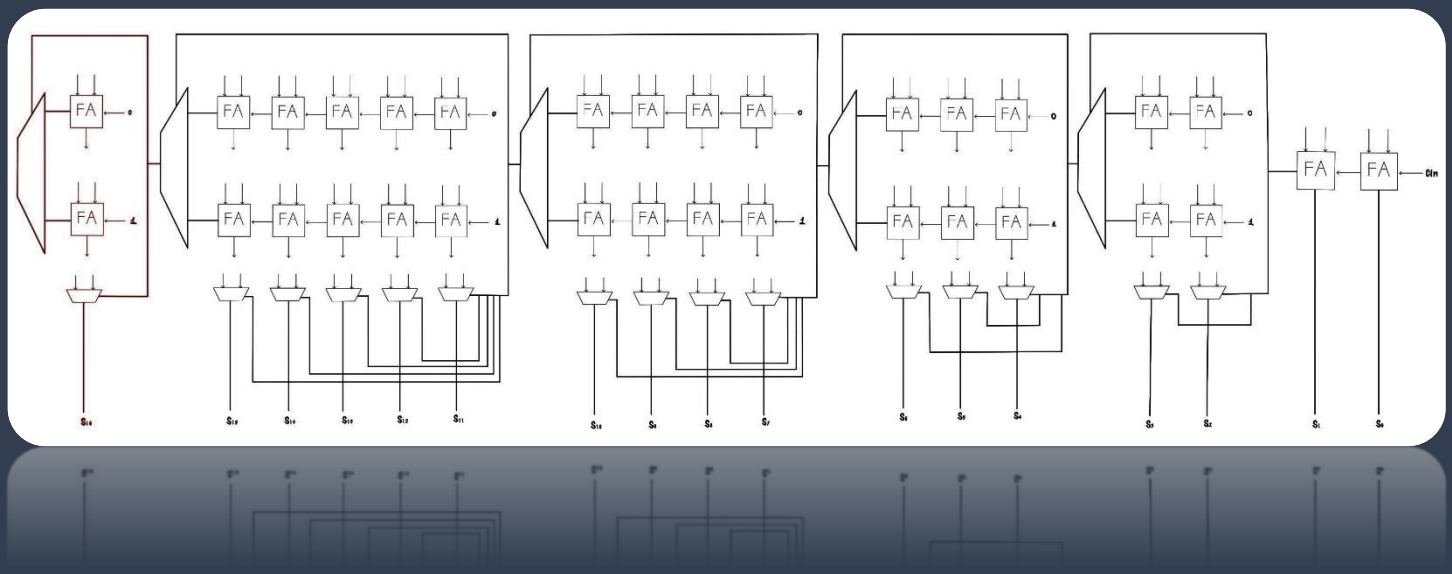


CARRY SELECT ADDER A 16 BIT

INDICE

Introduzione

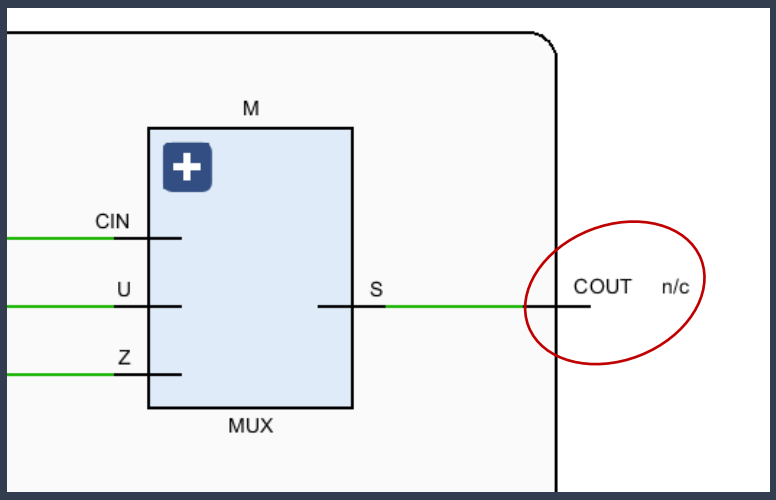
Il Carry Select Adder a 16 bit che ho deciso di costruire ha la seguente configurazione (illustrazione da me disegnata):



Inizialmente presenta un semplice ripple carry a due full adder. Successivamente ho inserito cinque blocchi (l'ultimo per la gestione dell'estensione del segno), ognuno dei quali presenta:

- Due ripple carry (RC_Z e RC_U) che lavorano in contemporanea, in cui il primo si suppone che riceve come carry in il valore logico 0, mentre il valore 1 il secondo.
- Ogni ripple carry di un singolo blocco presenta lo stesso numero di full adder (ovvio) che però non è lo stesso considerando blocchi diversi. Infatti, ho scelto di incrementare di uno la quantità di FA in ogni RC a mano a mano che passiamo da un blocco all'altro ad esclusione dell'ultimo (in quanto si occupa solo di gestire l'estensione del segno).
- Le uscite di ogni coppia di full adder ($FA°_i$, FA'_i) di ogni blocco (il primo riguarda il RC_Z e il secondo è relativo invece al RC_U), sono messe come ingresso di un multiplexer, il cui selettore è il riporto in uscita del blocco precedente o del ripple carry precedente se stiamo considerando il primo blocco.
- Il riporto in uscita di ogni blocco è dato dall'uscita del multiplexer che ha come ingressi i due riporti in uscita dei ripple carry RC_Z e RC_U e come selettore il riporto in uscita del blocco precedente.

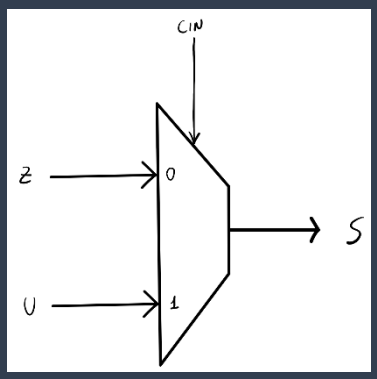
- Come ultimo blocco come già anticipato, gestisco l'estensione del segno. Al suo interno i due full adders in "parallelo" presentano come ingressi a_{15} e b_{15} (gli stessi ingressi dei full adders $FA°_{15}$ e FA'_{15}. Le uscite S16° e S16' ovviamente sono messe nel multiplexer corrispondente e verrà selezionata in base al selettore (che assume il valore del riporto in uscita del blocco precedente). Ovviamente ci è utile solo se il selettore è 1 (abbiamo un riporto), altrimenti abbiamo solo ricopiato l'ultimo bit (S_{15}) che comunque non altera il risultato.
- Ho fatto la scelta di non inserire il riporto in uscita di tutto il carry select adder poiché ai fini del risultato è inutile (forse potrebbe servire al committente per usufruire del segno del risultato, ma basterebbe prendere il bit più significativo..). Infatti questo riporto viene calcolato dall'ultimo multiplexer ma non l'ho collegato a niente, come si vede dallo schematic (che vedremo nel dettaglio in seguito):



Ma procediamo con ordine.

Ho come prima cosa sviluppato i vari componenti, quali il multiplexer, il full adder, il ripple carry e l'intero blocco spiegato in precedenza.

Multiplexer 2:1 (MUX)



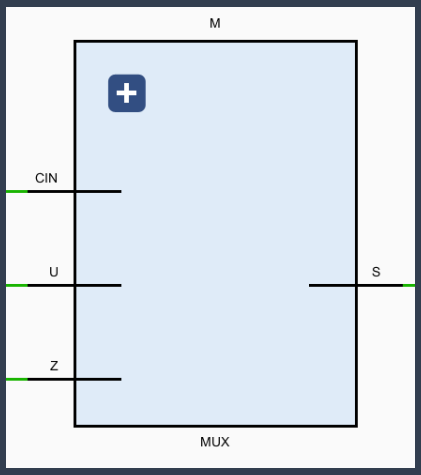
Il multiplexer 2:1 come ben sappiamo riceve in ingresso il selettore (che qui ho chiamato CIN, poi vedremo perché), altri due ingressi (Z e U) e un output.

Il selettore è capace di selezionare uno tra i due segnali Z e U e mandarlo in output:

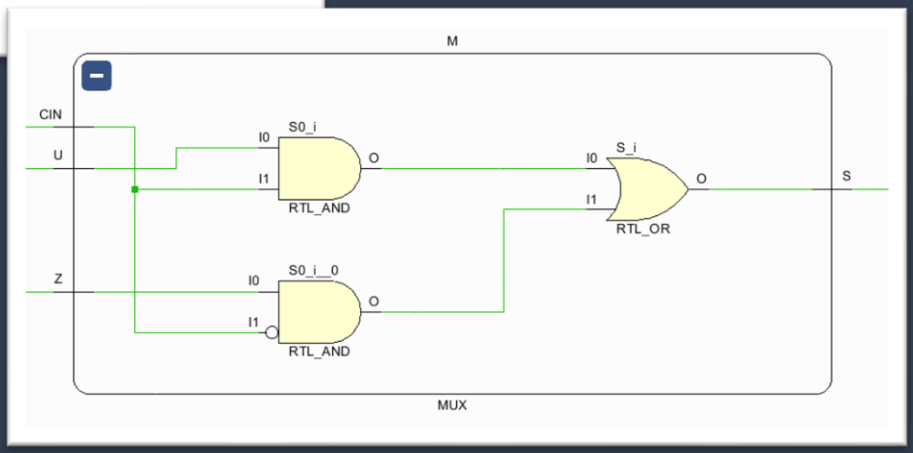
In questo caso se il selettore assumerà il valore alto passerà U, altrimenti Z.

Ho descritto quindi la sua interfaccia con la entity e nell'architecture ho riempito solo la parte di definizione descrivendo la sua funzione.

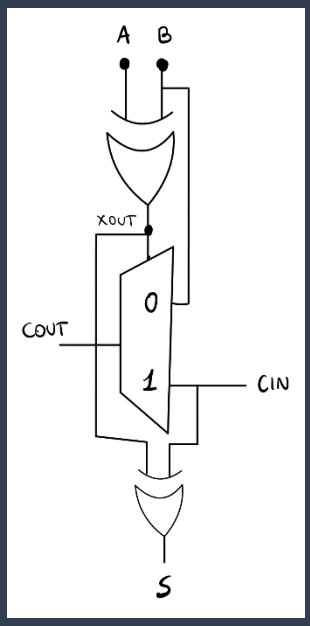
```
entity MUX is
    Port (
            Z : in STD_LOGIC;
            U : in STD_LOGIC;
            CIN : in STD_LOGIC;
            S : out STD_LOGIC);
end MUX;
```




```
architecture Behavioral of MUX is

begin

S <= ((U and CIN) or (Z and (not CIN)));

end Behavioral;
```



Full Adder (FA)



La configurazione del full adder che ho deciso di utilizzare è quella vista durante il corso in cui vi è la presenza del multiplexer.

Nella entity ho descritto i segnali di ingresso A , B e CIN (riporto in ingresso) e quelli di uscita S e COUT (carry out).

Nell'architecture ho dovuto dichiarare XOUT che è un segnale intermedio e il component relativo al multiplexer (MUX).

Come ho rappresentato in figura l'uscita XOUT deriva dalla xor tra A e B, il quale sarà messo in xor con l'ingresso CIN, la cui uscita sarà S.

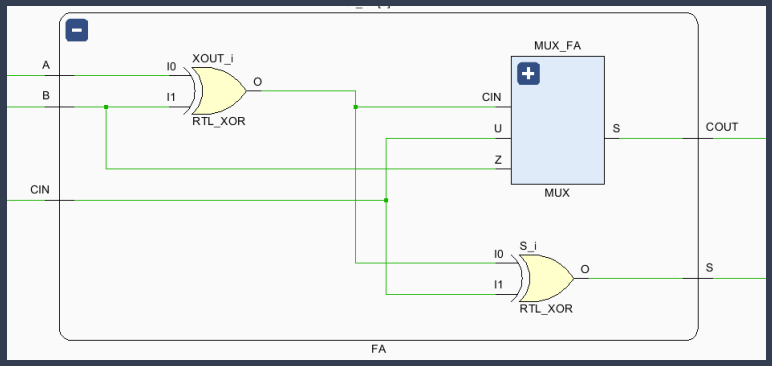
Il riporto in uscita (COUT), invece, è ricavato con l'ausilio del multiplexer che farà assumere il valore del segnale B se XOUT è 0, altrimenti assume il valore del riporto in ingresso.

Vediamo quanto detto in VHDL:

```
entity FA is
    Port (
            A : in STD_LOGIC;
            B : in STD_LOGIC;
            CIN : in STD_LOGIC;
            COUT : out STD_LOGIC;
            S : out STD_LOGIC);
end FA;
```

```
component MUX is
    Port (
            Z : in STD_LOGIC;
            U : in STD_LOGIC;
            CIN : in STD_LOGIC;
            S : out STD_LOGIC);
end component;
```

```
begin

XOUT <= A xor B;
S <= XOUT xor CIN;
MUX_FA : MUX port map(B,CIN,XOUT,COUT);

end Behavioral;
```



Ripple Carry (RC)

Il ripple carry è composto da k full adders disposti in serie, in cui ogni riporto in ingresso corrisponde al riporto in uscita del precedente.

Faremo uso del ripple carry (B*2)+1 volte, dove B è il numero di blocchi del carry select :

- Il ripple carry singolo iniziale composto da due FA.
- Il RC_Z in cui si suppone che il carry_in ha valore logico 0.
- Il RC_U in cui si suppone che il carry_in assume il valore alto.

Gli ultimi due sono presenti in ogni blocco.

Nel mio carry select adder ogni coppia (RC_Z , RC_U) possiede una coppia di FA in più quando passiamo da un blocco di destra a quello di sinistra, ad eccezione dell'ultimo (estensione del segno).

Nel definire l'interfaccia del RC con la entity, ho utilizzato un generic, che mi consentirà di assegnargli poi il numero di full adders che deve contenere. Inoltre ho descritto gli ingressi : A , B , CIN ; e le uscite: COUT e S :

```
entity RippleCarry is
generic(
        N:integer);
    Port (
           A : in STD_LOGIC_VECTOR (N-1 downto 0);
           B : in STD_LOGIC_VECTOR (N-1 downto 0);
           CIN : in STD_LOGIC;
           COUT : out STD_LOGIC;
           S : out STD_LOGIC_VECTOR (N-1 downto 0));
end RippleCarry;
```

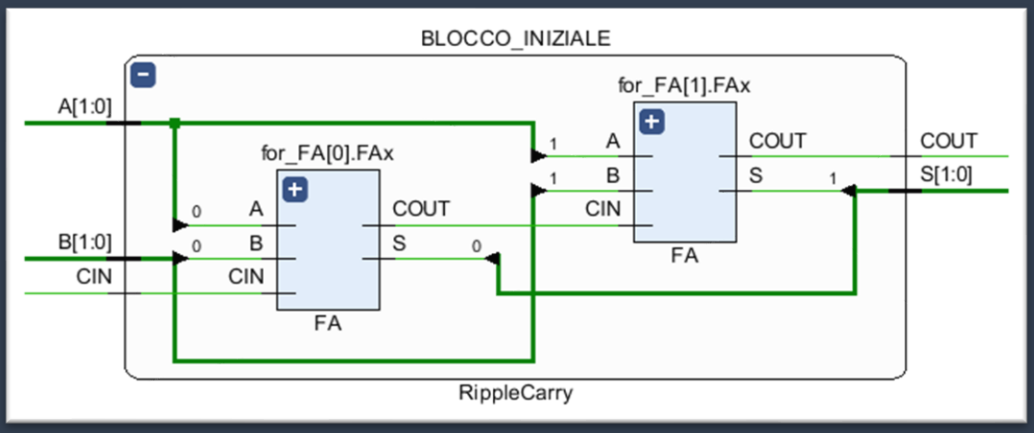
Nell'architecture ho dichiarato il vettore dei riporti in uscita (intermedi al circuito) di ogni FA e ovviamente il component Full Adder :

```
signal C : STD_LOGIC_VECTOR (N downto 0);

component FA is
    Port (
           A : in STD_LOGIC;
           B : in STD_LOGIC;
           CIN : in STD_LOGIC;
           COUT : out STD_LOGIC;
           S : out STD_LOGIC);
end component;
```

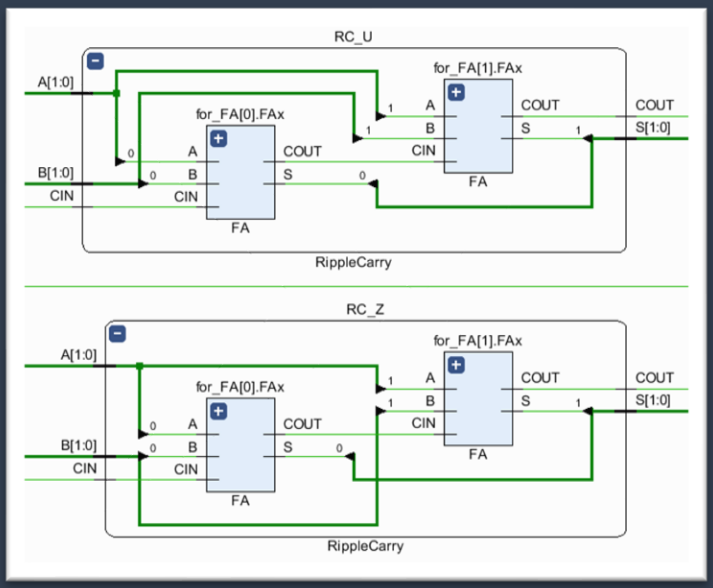
Nella zona di definizione dell'architecture, con un for generate creo gli N Full Adders, stando attento a passare i segnali nel port map con un ordine ben preciso, legato al port del component FA. Inserendo infine il valore CIN in posizione C(0) del vettore dei riporti in uscita del RC e in COUT il valore dell'ultimo riporto (in posizione N) del vettore C.

```
for_FA: for i in 0 to N-1 generate
            FAx : FA port map(A(i),B(i),C(i),C(i+1),S(i));
        end generate for_FA;

        C(0) <= CIN;
        COUT <= C(N);

end Behavioral;
```

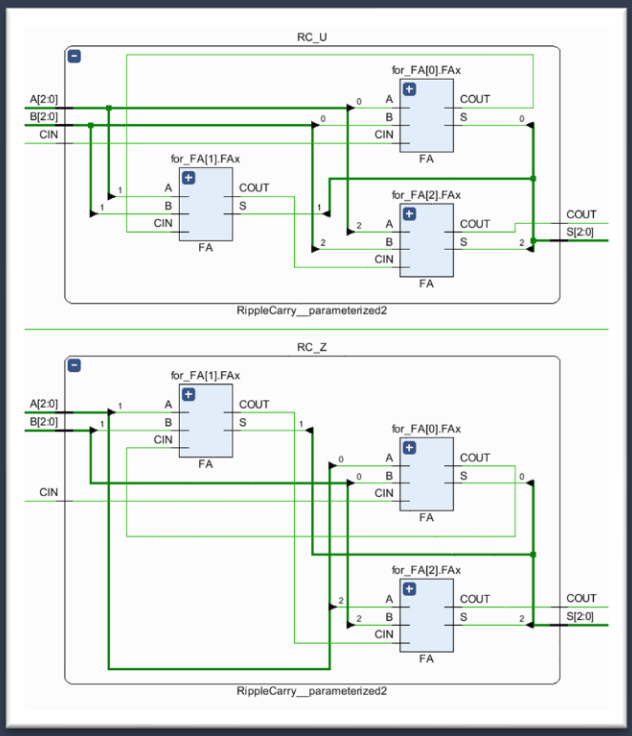
Qui il ripple carry iniziale :



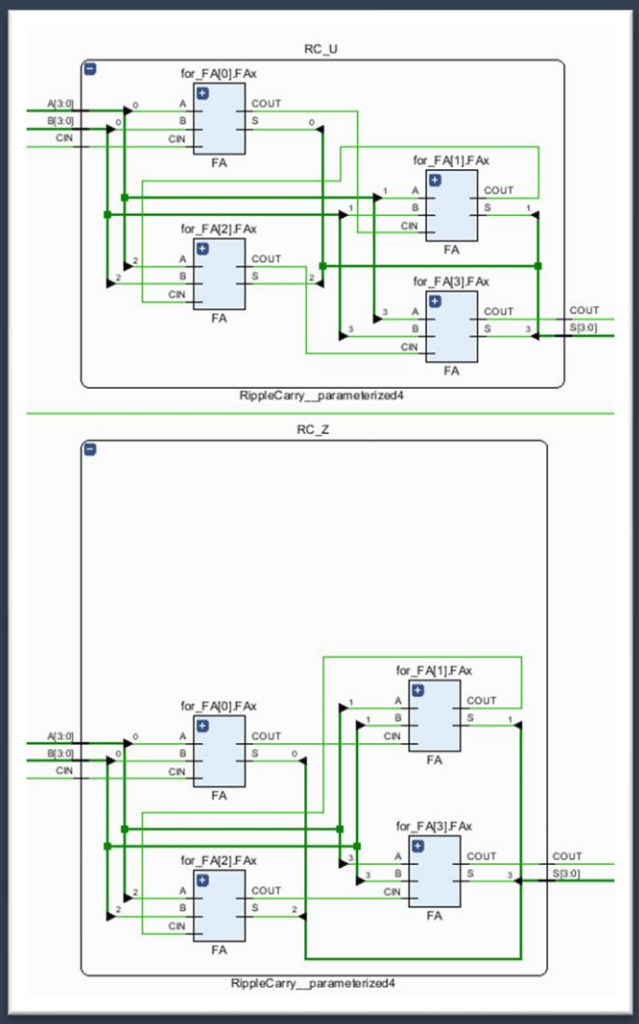
I ripple carry del blocco 1 :



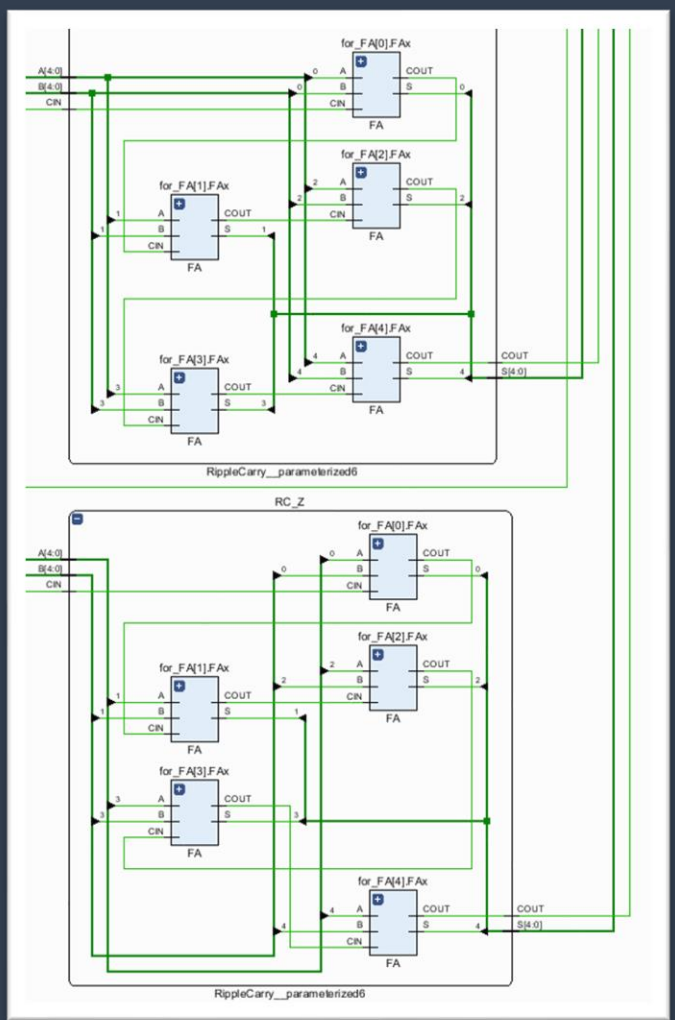
I ripple carry del blocco 2 :



I ripple carry del blocco 3 :



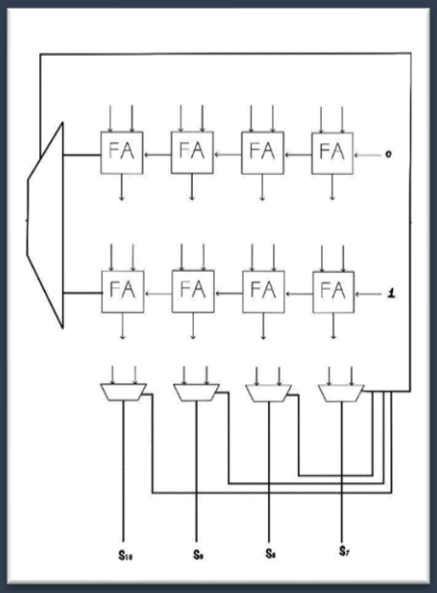
I ripple carry del blocco 4 :



Il Blocco

Come già anticipato ogni blocco presenta :

- Due ripple carry : RC_Z e RC_U
- N multiplexer che gestiscono le uscite di ogni coppia di full adder ($FA°_i$, FA'_i)
- Il multiplexer che genera il riporto in uscita del blocco



Come con il ripple carry nella entity ho aggiunto un generic, che mi consentirà poi nella definizione di decidere il numero di full adders che deve avere ogni ripple carry del blocco in questione.

E come al solito ho descritto gli ingressi e le uscite :

```
entity BLOCCO_RC is
    generic(
            NB:integer );
    Port (
            A : in STD_LOGIC_VECTOR (NB-1 downto 0);
            B : in STD_LOGIC_VECTOR (NB-1 downto 0);
            CIN : in STD_LOGIC;
            COUT : out STD_LOGIC;
            S : out STD_LOGIC_VECTOR (NB-1 downto 0));
end BLOCCO_RC;
```

Nell'architecture ho inizialmente dichiarato due vettori. Il primo rappresenterà la soluzione del ripple carry RC_U e il secondo relativa al RC_Z. Ed inoltre i due riporti in uscita dei due RC, che saranno poi mandati in ingresso in un multiplexer che ha come selettore il carry_out del blocco precedente, determinando così COUT (il riporto del blocco).

Successivamente non ho potuto che dichiarare i due component RippleCarry e MUX.

Dichiarazione nell'architecture :

```
architecture Behavioral of BLOCCO_RC is

signal S_U , S_Z : STD_LOGIC_VECTOR(NB-1 downto 0);
signal C_U , C_Z : STD_LOGIC; --rispettivamente il c
```

```
component RippleCarry is
generic(
        N:integer);
    Port (
           A : in STD_LOGIC_VECTOR (N-1 downto 0);
           B : in STD_LOGIC_VECTOR (N-1 downto 0);
           CIN : in STD_LOGIC;
           COUT : out STD_LOGIC;
           S : out STD_LOGIC_VECTOR (N-1 downto 0));
end component;
```

```
component MUX is
    Port (
           Z : in STD_LOGIC;
           U : in STD_LOGIC;
           CIN : in STD_LOGIC;
           S : out STD_LOGIC);
end component;
```

Come già anticipato il generic NB del blocco identificherà il numero di FA in ogni Ripple Carry RC_U e RC_Z. Pertanto siccome il generic N del component RippleCarry ha lo stesso scopo, nella definizione del RC (in particolare nel generic map) gli passo proprio il valore NB !

Come possiamo vedere in RC_U inserisco come CIN il valore logico 1, viceversa per il RC_Z.

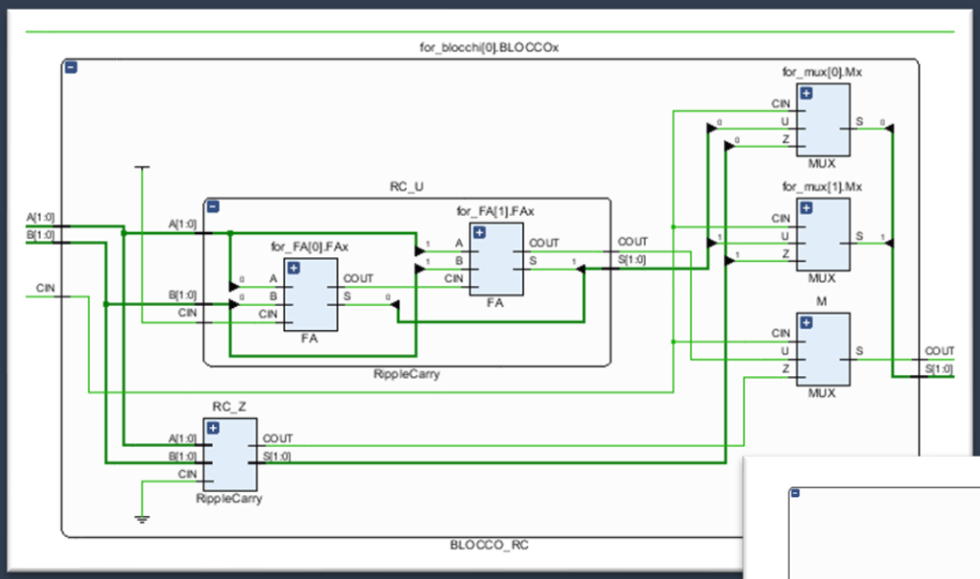
Assegno quindi gli output C_U e S_U al RC_U e C_Z e S_Z al RC_Z.

```
begin

RC_U : RippleCarry generic map( N => NB)
                   port map(A,B,'1',C_U,S_U);
RC_Z : RippleCarry generic map( N => NB)
                   port map(A,B,'0',C_Z,S_Z);
```

Infine non ci resta che definire i multiplexer !

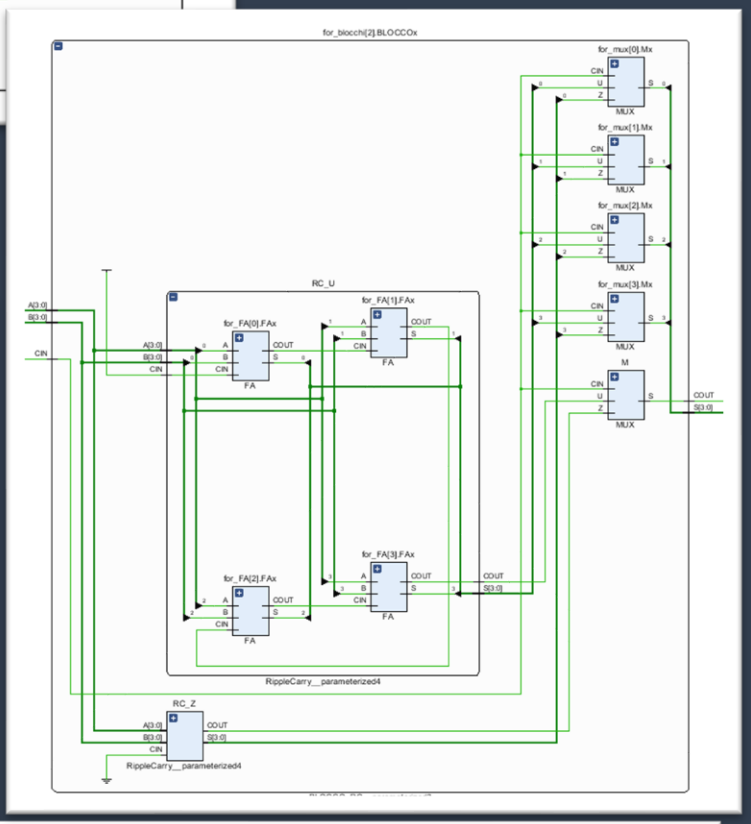
Prima con un for generate genero quelli relativi all'output dei FA in parallelo e infine il MUX che gestisce il riporto in uscita del blocco :

```
for_mux : for i in 0 to NB-1 generate
            Mx: MUX port map(S_Z(i),S_U(i),CIN,S(i));
end generate for_mux;

-- Gestione MUX del riporto in uscita

M:MUX port map(C_Z,C_U,CIN,COUT);

end Behavioral;
```

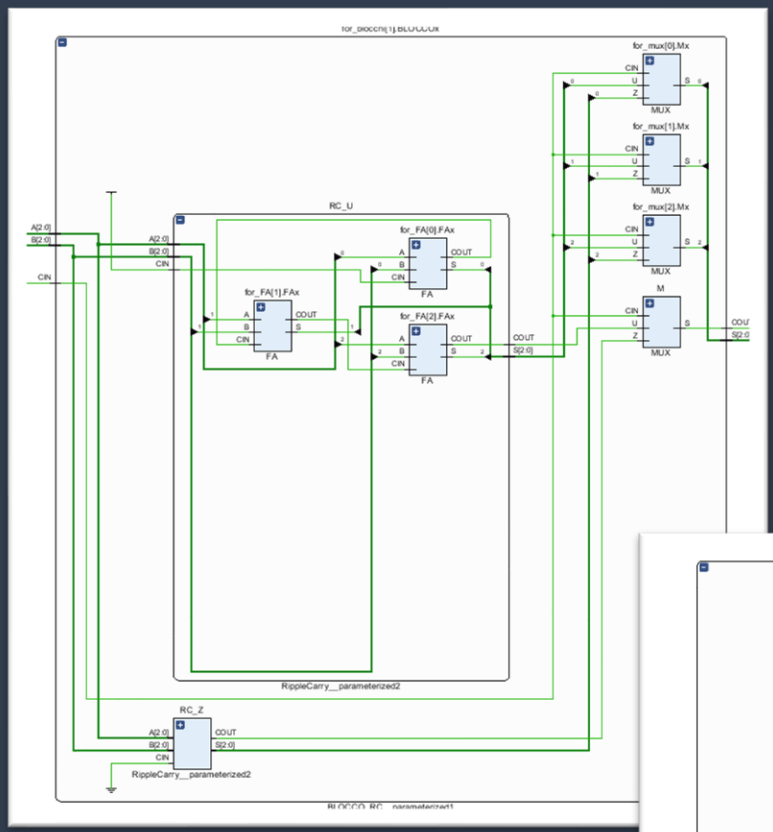


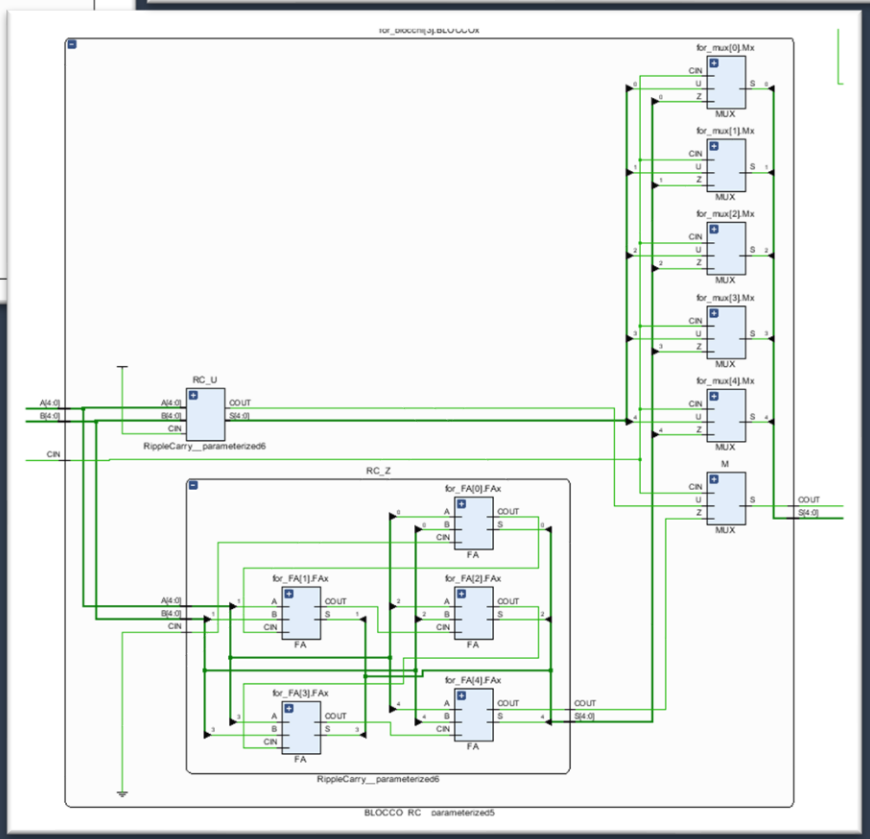
BLOCCO 1

BLOCCO 3



BLOCCO 2





BLOCCHI

BLOCCO 4

Carry Select Adder

Ho sviluppato inizialmente (dopo aver importato le opportune librerie) la entity del CarrySelect, che descrive la sua interfaccia (i segnali di ingresso/uscita o opportuni altri parametri):

```
entity CarrySelect is

    Port (
            A,B : in STD_LOGIC_VECTOR (15 downto 0);
            C_IN: in STD_LOGIC;
            S : out STD_LOGIC_VECTOR (16 downto 0));

end CarrySelect;
```

Nell'architecture ho dichiarato il vettore dei riporti di ogni blocco (compreso quello del ripple carry iniziale che occuperà la posizione 0) e due component:

- Il ripple carry.
- Il blocco che comprende i due ripple carry RC_Z e RC_U e i relativi mux (Blocco_RC).

```
signal C_BLOCCHI : STD_LOGIC_VECTOR(5 downto 0);
```

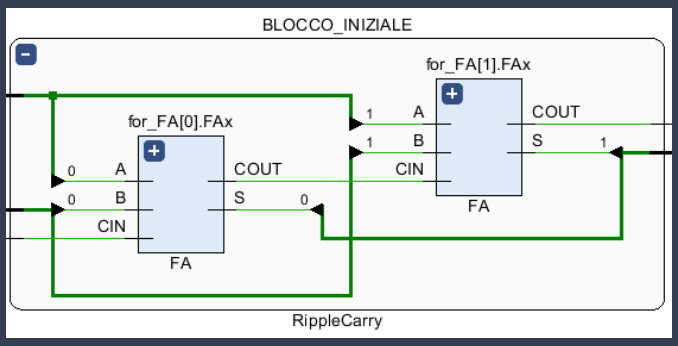
```
component RippleCarry is
generic(
        N:integer);
    Port (
            A : in STD_LOGIC_VECTOR (N-1 downto 0);
            B : in STD_LOGIC_VECTOR (N-1 downto 0);
            CIN : in STD_LOGIC;
            COUT : out STD_LOGIC;
            S : out STD_LOGIC_VECTOR (N-1 downto 0));
end component;
```

```
component BLOCCO_RC is
    generic(
            NB:integer );
    Port (
            A : in STD_LOGIC_VECTOR (NB-1 downto 0);
            B : in STD_LOGIC_VECTOR (NB-1 downto 0);
            CIN : in STD_LOGIC;
            COUT : out STD_LOGIC;
            S : out STD_LOGIC_VECTOR (NB-1 downto 0));
end component;
```

Sempre nell'architecture ma nella zona della definizione (begin) ho descritto il funzionamento del sistema, istanziando i component.

Dapprima il RippleCarry singolo composto da due FA:

```
--creo inizialmente il blocco iniziale con un solo ripplecarry:

BLOCCO_INIZIALE : RippleCarry generic map(2)
                  port map(A(1 downto 0),B(1 downto 0),C_IN,C_BLOCCHI(0),S(1 downto 0));
```



Ora definisco i 4 blocchi considerando che a partire dal primo (in cui ogni RC ha 2 FA), il numero di FA all'interno di ogni RC aumenti di uno fino al quarto blocco.

Per fare questo ho utilizzato un for generate (i = 0,1,2,3), definendo ogni blocco, indicando nel corrispettivo generic map il numero di FA da generare per ogni coppia di RC presente nei blocchi. Ho impostato il valore ad i + 2, in modo tale da ottenere la sequenza 2,3,4,5.

Quindi :

- Per i = 0 i RC avranno 2 FA
- Per i = 1 i RC avranno 3 FA
- Per i = 2 i RC avranno 4 FA
- Per i = 3 i RC avranno 5 FA

Ora però manca da gestire la parte un po' più complicata, ovvero la scelta di un indice generico che valga per ogni sottovettore di ingressi e uscite. Mi spiego meglio :

Ad ogni iterazione del for (quindi ad ogni blocco), nel port map dobbiamo passargli i vettori degli ingressi A [t downto s] , B [t downto s] e dell'uscita S [t downto s], in cui gli indici variano al variare del blocco di cui si sta considerando.

Dalla figura possiamo notare che la scelta degli indici t e s deve essere la seguente :

- Per i = 0 --> t = 3 , s = 2
- Per i = 1 --> t = 6 , s = 4
- Per i = 2 --> t = 10 , s = 7
- Per i = 3 --> t = 15 , s = 11

Possiamo subito notare che il valore S è facilmente ottenibile :

S = T - (1 + i) (LSB)

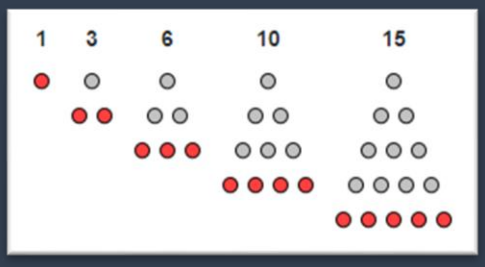
Non ci resta che ricavare l'indice T relativo all'MSB di ogni vettore.

Si può fare ricorso alla formula di Gauss legato ai *numeri triangolari*.

Un numero triangolare è un numero poligonale rappresentabile in forma di triangolo, ovvero, preso un insieme con una cardinalità pari al numero in oggetto. È possibile disporre i suoi elementi in modo da formare un triangolo equilatero o un triangolo isoscele.

(fonte: Wikipedia)

Ovvero :



e così via..

L' n-simo numero triangolare si può ottenere con la formula di Gauss :

$$T_n = \frac{n(n+1)}{2}$$

A noi però interessa la sequenza 3 , 6 , 10 , 15 e non 1 , 3 , 6 , 10 , 15 :

Per questo motivo (e considerando che i parte da 0), scelgo n = i + 2.

Ho quindi ricavato l'indice MSB che vale per ogni iterazione del for :

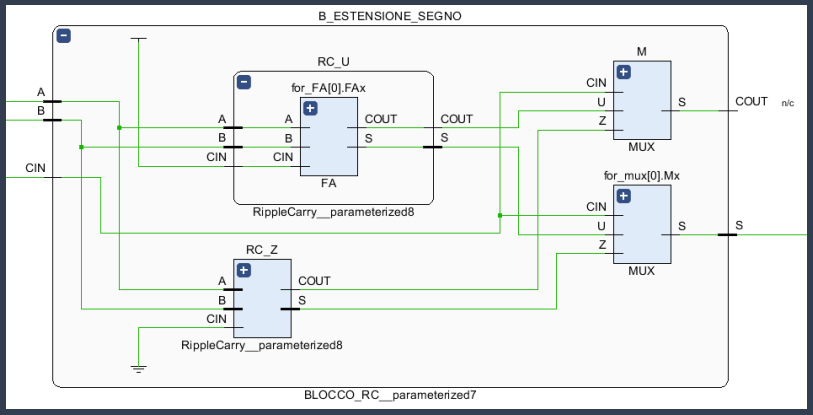
$$T = ((i+2)*((i+2)+1))/2$$

Posso quindi realizzare il port map nel seguente modo :

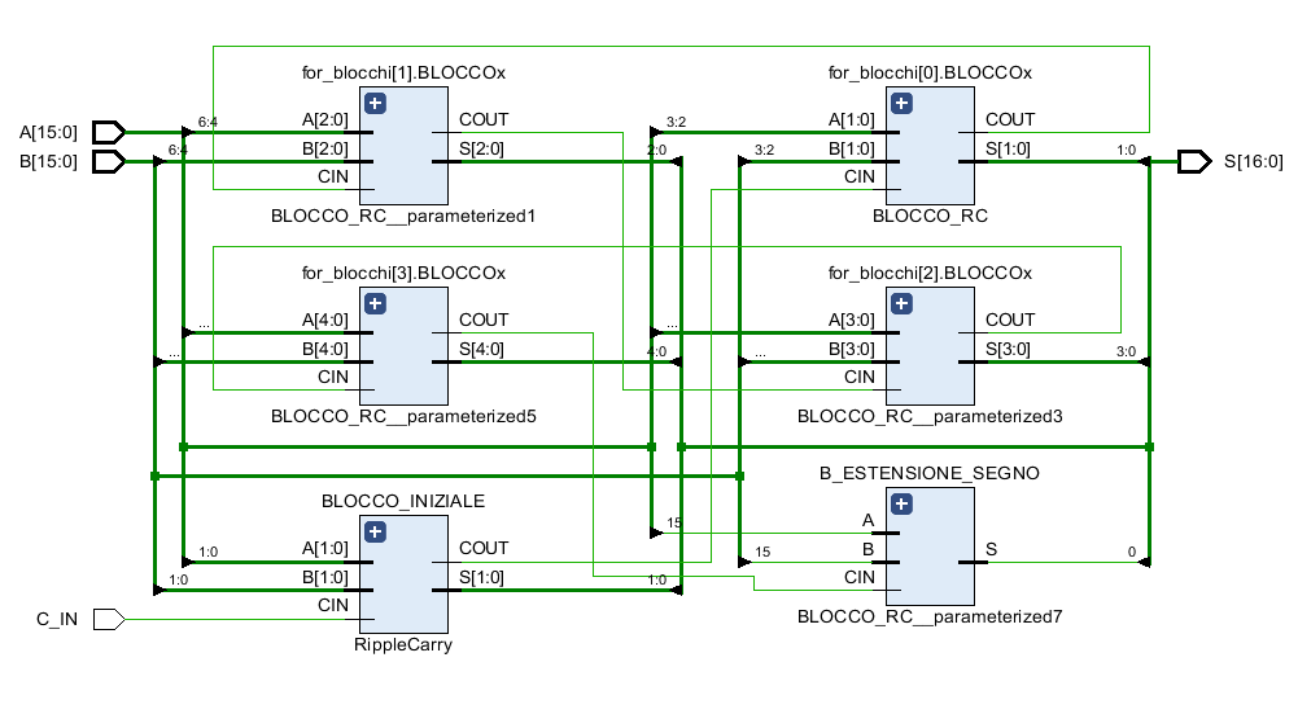
```
for_blocchi: for i in 0 to 3 generate

        BLOCCOx: BLOCCO_RC generic map(i+2)
        -- per gli indici MSB posso sfruttare la formula di gauss che ci consente di ricavare la
                        port map(
                                A => A(((i+2)*(i+2+1))/2 downto (((i+2)*(i+2+1))/2)-(1+i) ),
                                B => B(((i+2)*(i+2+1))/2 downto (((i+2)*(i+2+1))/2)-(1+i)),
                                CIN => C_BLOCCHI(i),
                                COUT => C_BLOCCHI(i+1),
                                S => S(((i+2)*(i+2+1))/2 downto (((i+2)*(i+2+1))/2)-(1+i))
                                );
end generate for_blocchi;
```

Infine il blocco dedicato all'estensione del segno :

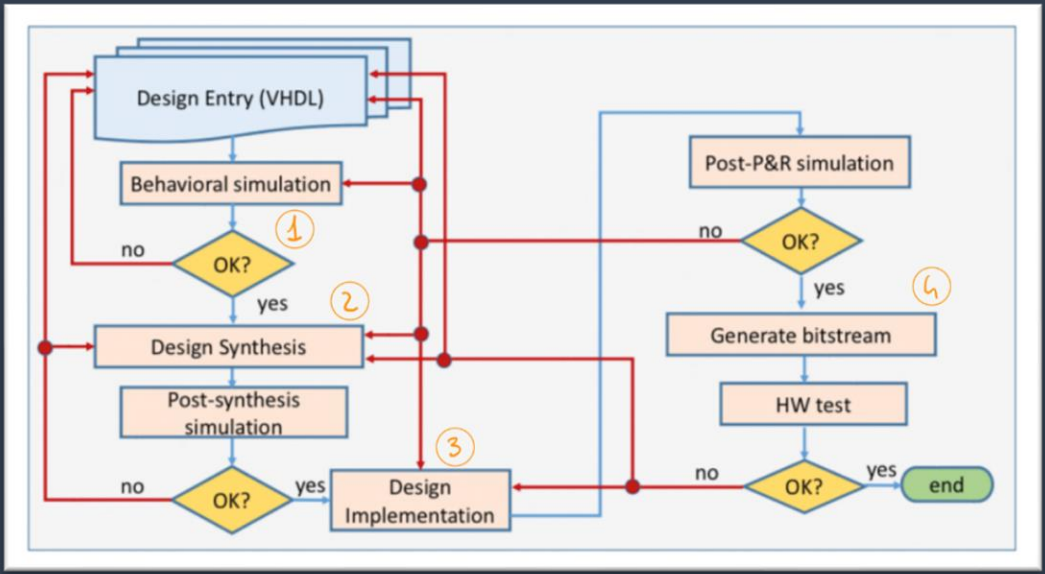
```
--gestione estensione del segno

B_ESTENSIONE_SEGNO : BLOCCO_RC generic map(1)
                port map(A(15 downto 15),B(15 downto 15),C_BLOCCHI(4),C_BLOCCHI(5),S(16 downto 16));
```



Schematic


for_blocchi[1].BLOCCOx
A[2:0]
B[2:0]
CIN
COUT
S[2:0]
BLOCCO_RC__parameterized1
for_blocchi[0].BLOCCOx
A[1:0]
B[1:0]
CIN
COUT
S[1:0]
BLOCCO_RC
A[15:0]
B[15:0]
S[16:0]
for_blocchi[3].BLOCCOx
A[4:0]
B[4:0]
CIN
COUT
S[4:0]
BLOCCO_RC__parameterized5
for_blocchi[2].BLOCCOx
A[3:0]
B[3:0]
CIN
COUT
S[3:0]
BLOCCO_RC__parameterized3
BLOCCO_INIZIALE
A[1:0]
B[1:0]
CIN
COUT
S[1:0]
RippleCarry
C_IN
B_ESTENSIONE_SEGNO
A
B
CIN
S
BLOCCO_RC__parameterized7

Flusso di Progettazione dell'RTL



1) Nell'Design Entry si descrive la funzionalità del circuito (facendo uso in questo caso di un linguaggio di alto livello VHDL). Qui avviene la verifica della sintassi fatta automaticamente da Vivado, ma noi abbiamo bisogno anche di una verifica dal punto di vista funzionale.
   Per questo motivo si esegue una simulazione comportamentale (Behavioral simulation) fatta mediante l'uso di forme d'onda. Forniamo in ingresso del nostro circuito dei particolari valori dei dati da processare e andiamo a verificare se si ottiene il risultato corretto. Se così non fosse possiamo tornare indietro a correggere eventuali errori.

2) Sintesi : prende il codice e la traduce in un circuito, ovvero un insieme di porte logiche (schematic) che siano collegate tra di loro nella maniera appropriata perché tutto funzioni come vogliamo.
   Bisogna però essere certi che la mappatura del nostro codice in elementi reali non abbia modificato il funzionamento atteso, pertanto è opportuno lanciare una simulazione post sintesi. Una volta fatto ciò possiamo ritornare all'inizio perché l'utente inizialmente ha potuto aggiungere delle direttive (può volere ad esempio che si utilizzi solo porte and) e magari queste sono troppo stringenti e le devo modificare.

3) Nel design implementation introduco l'interfaccia, la quale mi permette di comunicare con l'esterno (altri componenti reali che ancora una volta possono andare a modificare nel tempo il comportamento del circuito).
   Ogni volta che introduco qualcosa che non è prevista nella descrizione VHDL devo andare a ripetere il test, in questo caso post implementazione.

4) Se tutto va bene si passa alla realizzazione del chip fisico.
   Se utilizzo un FPGA si genererà un file di 0 e di 1 (bitstream) che corrisponde alla descrizione complessiva del circuito.
   Se invece stessi realizzando un chip custom bisognerebbe rivolgersi ad una foundry esterna. Questa fase richiede un tempo piuttosto lungo (qualche mese) alla fine del quale la foundry ci restituisce il chip.
   Una volta ricevuto dobbiamo verificarne il suo funzionamento eseguendo un test hardware svolto in laboratorio.

Simulazione Comportamentale

Tutto ciò che abbiamo visto prima di descrivere il flusso di progettazione (Carry Select, Fa, Rc, Mux , il blocco) si trattava del Deisgn Entry in cui ho descritto le funzionalità del circuito.

Ora è necessario eseguire una simulazione comportamentale, per verificare se effettivamente il circuito svolge la sua funzione in modo corretto.

Ho creato un nuovo file vhd SimCarrySelect in cui la entity resta vuota.

Processo 256 combinazioni di ingresso diversi come richiesto dalla traccia del progetto e andiamo a verificare se si ottiene il risultato corretto. Se così non fosse posso tornare indietro a correggere eventuali errori.

Posso quindi descrivere direttamente l'architecture in cui dichiaro il componente che voglio simulare.

Per prima cosa dichiaro i 3 signal IA, IB, OSum , i primi due saranno gli ingressi e l'ultimo l'uscita del nostro circuito.

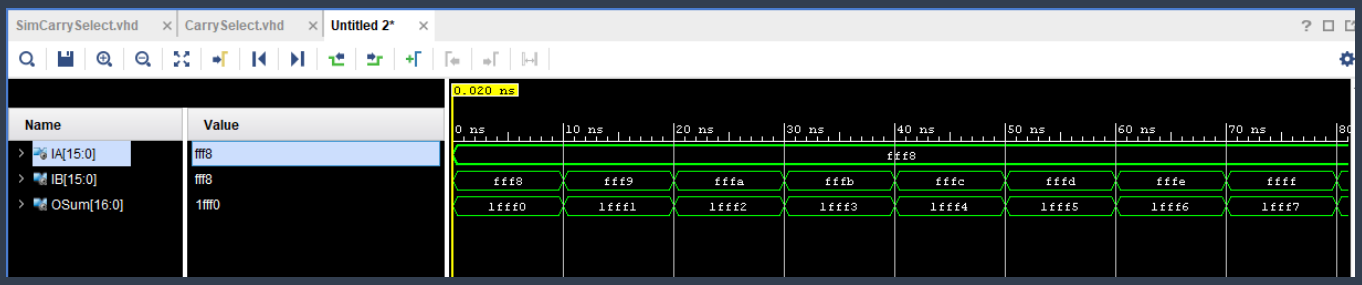
E subito dopo ovviamente il component Carry Select :

```
signal IA,IB : STD_LOGIC_VECTOR (15 downto 0);
signal OSum : STD_LOGIC_VECTOR (16 downto 0);

component CarrySelect is

    Port (
           A : in STD_LOGIC_VECTOR (15 downto 0);
           B : in STD_LOGIC_VECTOR (15 downto 0);
           C_IN: in STD_LOGIC;
           S : out STD_LOGIC_VECTOR (16 downto 0));

end component;
```

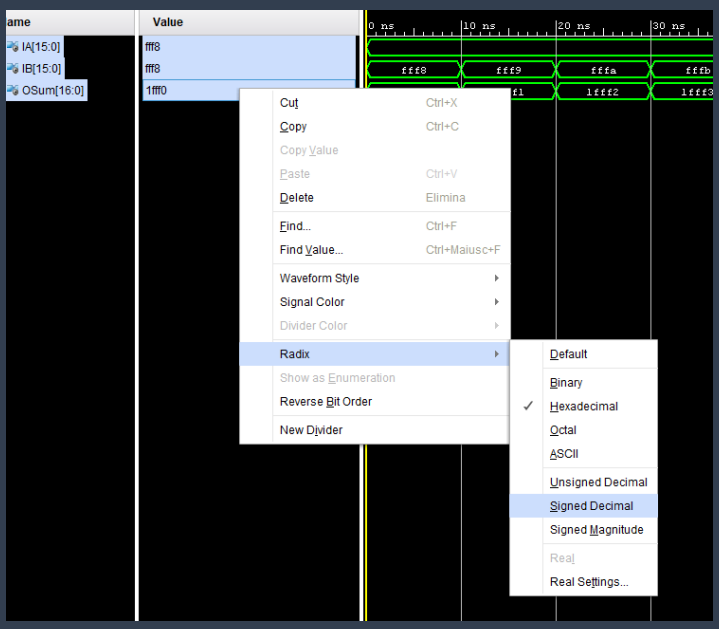
Nel begin inizialmente collego il Carry select con i segnali di ingresso e uscita (port map) e in seguito all'interno del process effettuo 256 combinazioni diverse di ingressi con due for loop innestati.

```
        for va in -8 to 7 loop
            IA <= conv_std_logic_vector(va,16);
            for vb in -8 to 7 loop
                IB <= conv_std_logic_vector(vb,16);
        wait for 10 ns;
            end loop;
        end loop;
end process;


end Behavioral;
```

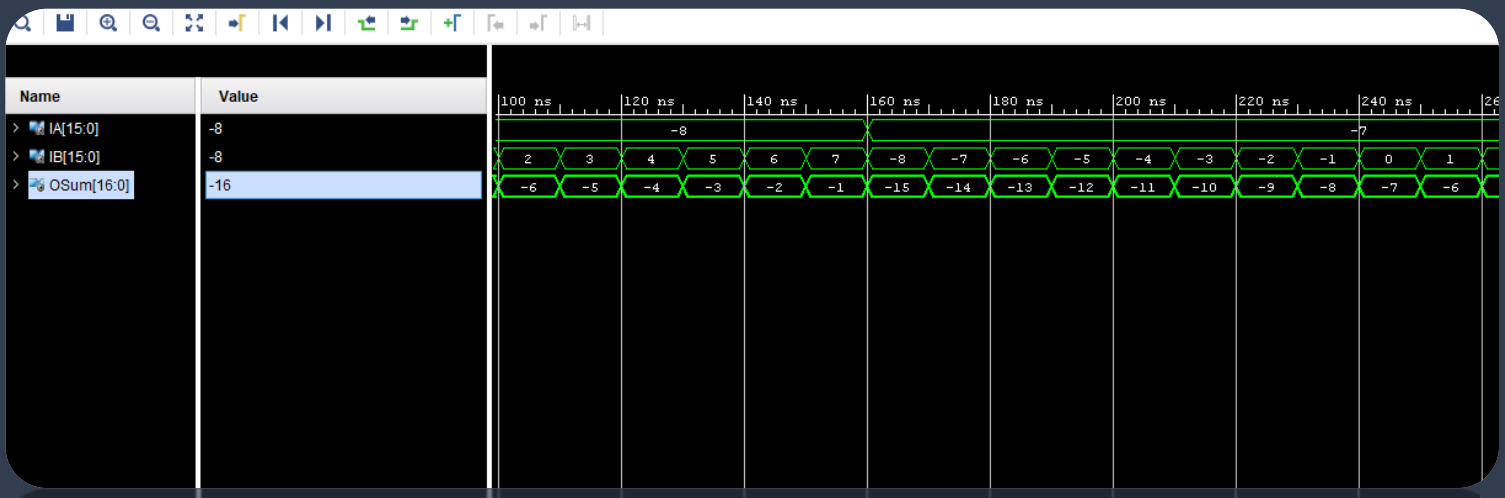
Lanciata la behavioral simulation ci troviamo di fronte questa schermata :



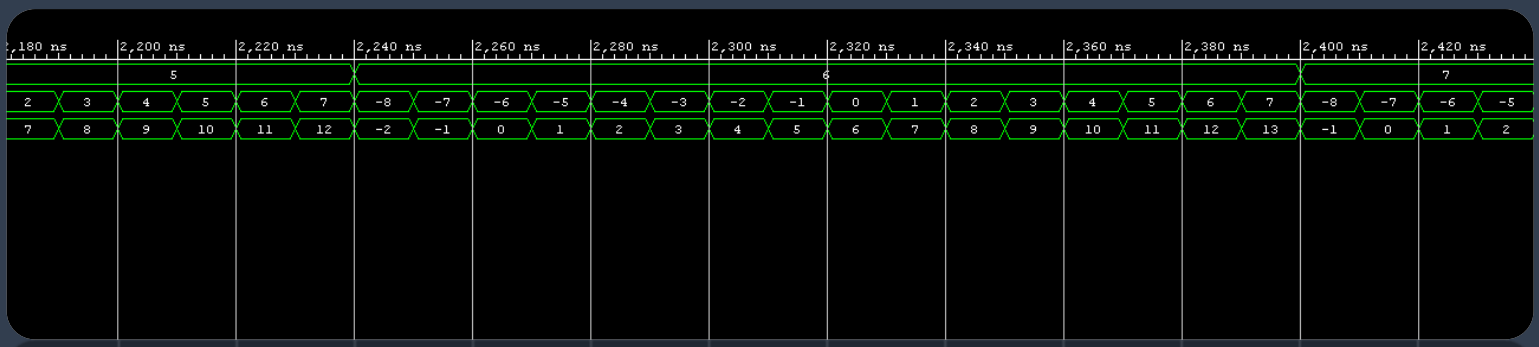
Ma i valori sono in esadecimale.. Possiamo modificarli premendo su radix e poi su decimale con segno :



Ecco ora siamo pronti a verificare se il nostro circuito esegue correttamente la somma tra due numeri a 16 bit (rappresento solo una porzione della simulazione per mancanza di spazio) :

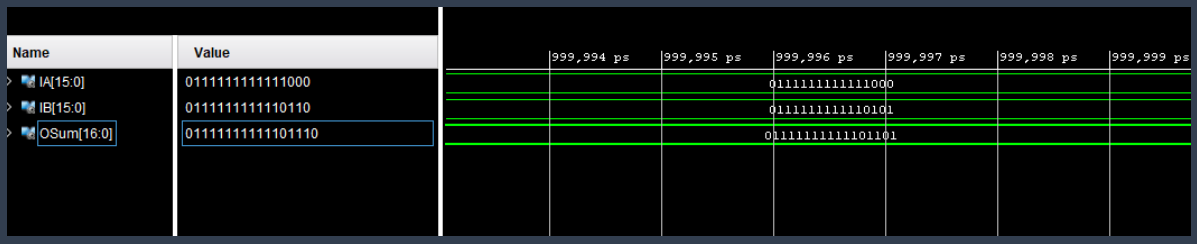


Tutto corretto !!



Siccome il range di un numero a 16 bit in complemento a due è [-2^{15} , $2^{15} + 1$] provo a verificare se ho gestito bene l'estensione del segno considerando 256 combinazioni lavorando con il più grande valore a 16 bit esprimibile in complemento a due :

```
    for va in 32754 to 32769 loop
        IA <= conv_std_logic_vector(va,16);
        for vb in 32754 to 32769 loop
            IB <= conv_std_logic_vector(vb,16);
    wait for 10 ns;
        end loop;
    end loop;
nd process;
```

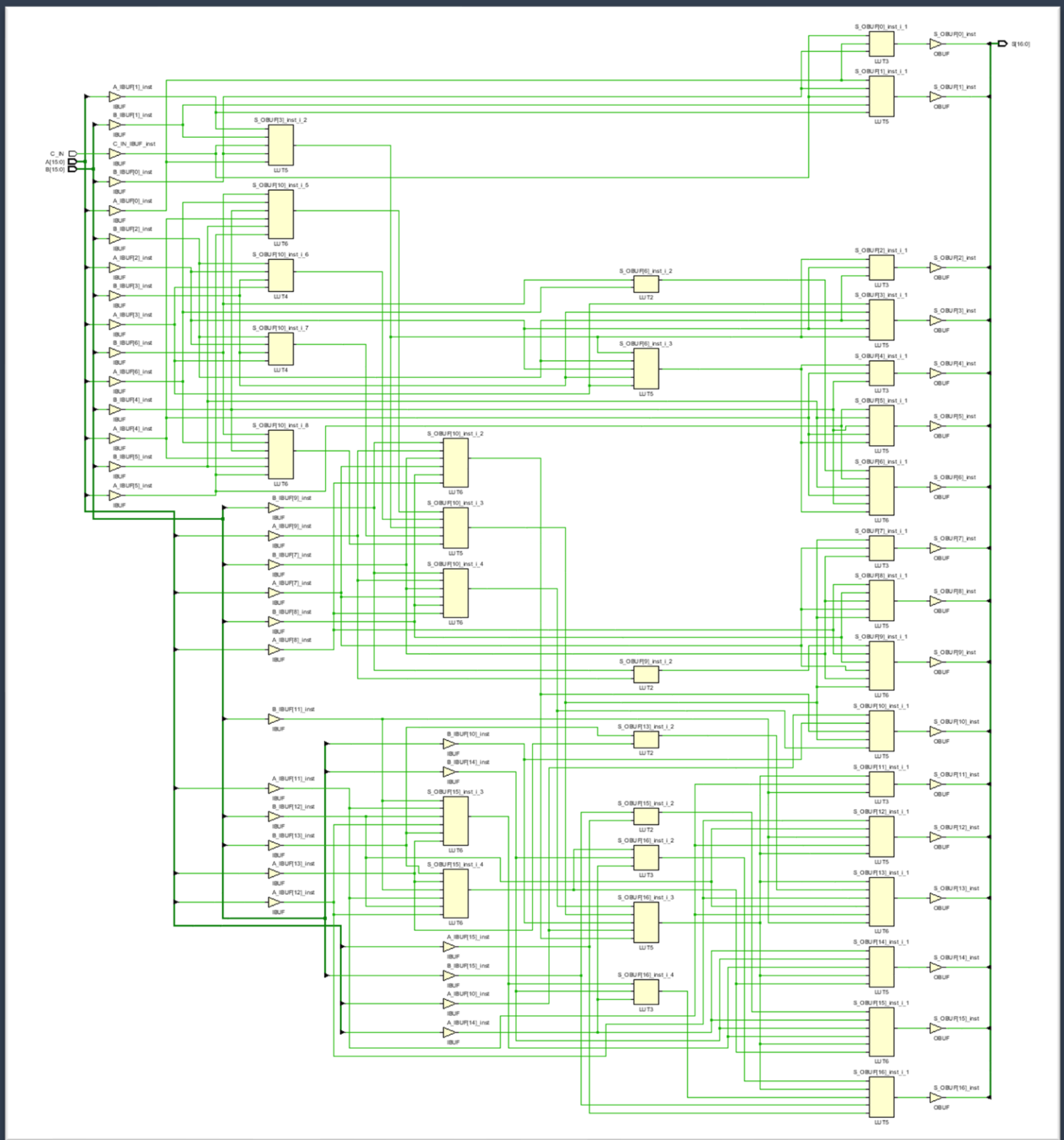


Come possiamo vedere l'estensione del segno ha funzionato ! Il risultato è un numero a 17 bit. Se non avessimo gestito ciò ci saremmo fermati al 16esimo bit ed il risultato 1111111111101110 in decimale sarebbe stato -18.. (32760 + 32758 = 65518 e non -18 !!).

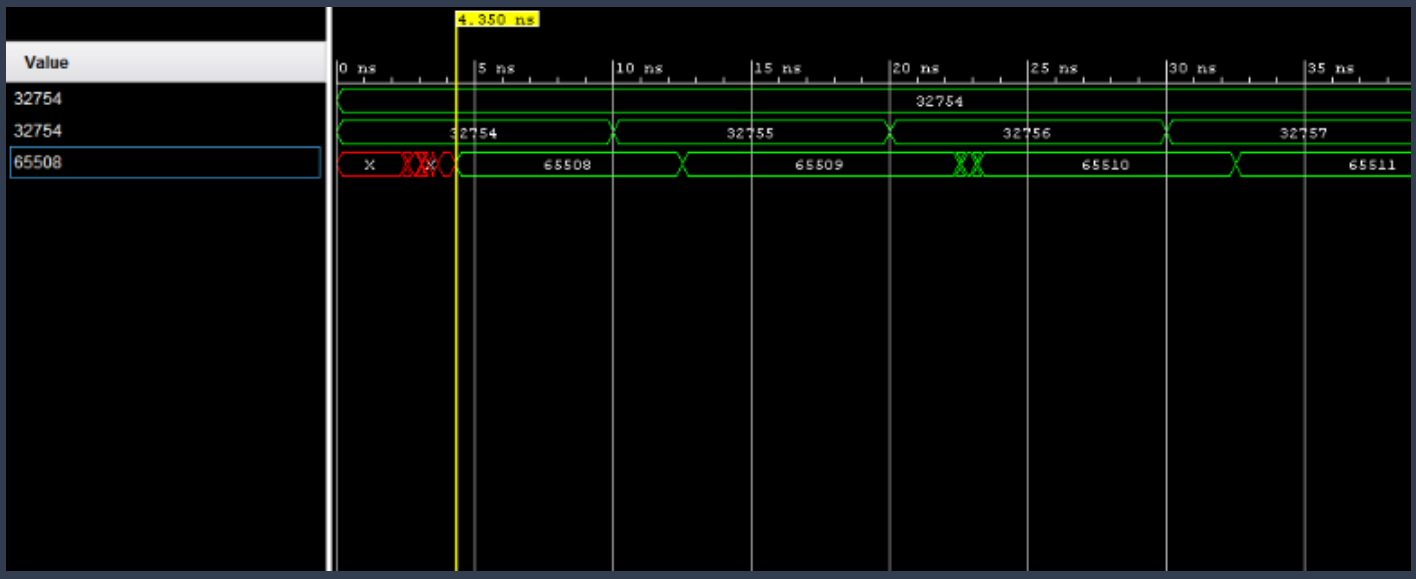
(in tutti i casi il risultato sarà a 17 bit. Ma in tutti quelli in cui il valore del risultato è esprimibile a 16 bit (appartiene al range espresso poc'anzi), sarà sostanzialmente ricopiato il segno. Quindi in questo caso (come l'esempio del for che va da -8 a 7) se avessimo considerato i primi 16 bit il risultato sarebbe stato corretto ugualmente)

Sintesi e Simulazione Post Sintesi

La sintesi come già anticipato traduce il codice in un circuito, ovvero un insieme di porte logiche (vediamo in figura delle LUT) che siano collegate tra di loro nella maniera appropriata perché tutto funzioni come vogliamo :

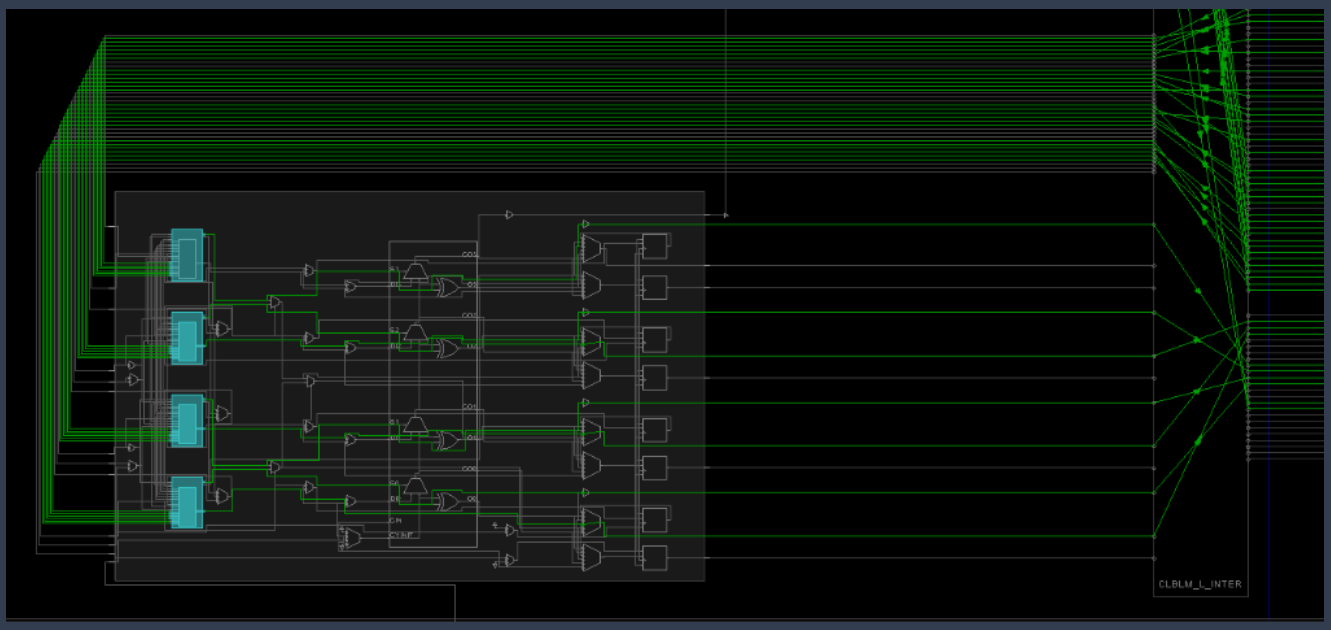


Bisogna però essere certi che la mappatura del nostro codice in elementi reali non abbia modificato il funzionamento atteso, pertanto è opportuno lanciare una simulazione post sintesi.

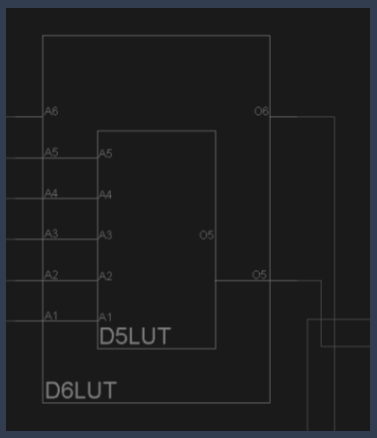


Oltre ad aver verificato il corretto funzionamento, possiamo notare che in questa simulazione c'è un ritardo iniziale nell'output (circa pari a 4.350 ns), dovuto al fatto che stiamo simulando l'effettivo comportamento delle porte logiche o delle LUT che generano appunto dei ritardi reali.

Implementazione e Simulazione Post Implementazione



Questa vista mi consente di entrare proprio all'interno del chip. Qui dentro ci sono una serie di componenti, insieme ai collegamenti elettrici, che sono messi a disposizione dalla tecnologia che stiamo usando, e siamo noi a decidere quali utilizzare.



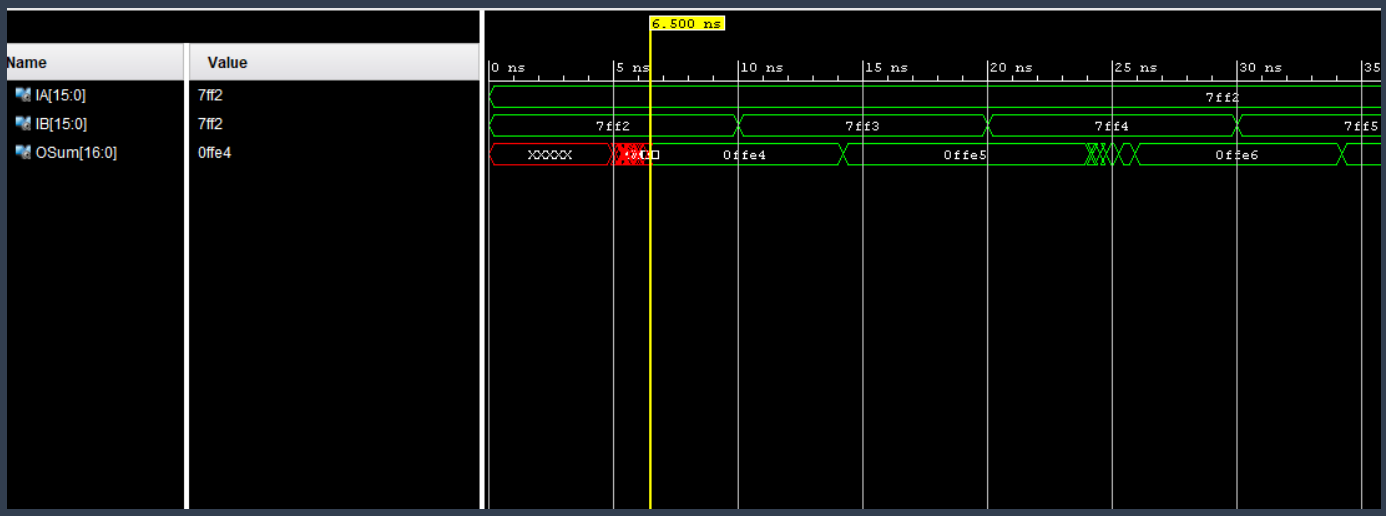
Le LUT che ci mette a disposizione sono delle LUT che possono essere configurate per svolgere due funzioni contemporaneamente "inglobando" ad una LUT a sei ingressi e due uscite, una LUT a cinque ingressi e una uscita (quindi non le possiamo considerare delle funzioni elementari).

Il codice VHDL che abbiamo scritto, va a stabilire che cosa deve fare ognuna di queste risorse non elementari, in base a quello che è il risultato della sintesi.

Quindi lo scopo del sintetizzatore è quello di coniugare le risorse messe a disposizione con il codice VHDL realizzato nello step dedicato al design entry.

Le LUT colorate in azzurro sono quelle che effettivamente abbiamo utilizzato e le loro informazioni le avevamo già avute dalla sintesi, ma invece con l'implementazione stiamo recuperando informazioni relative ai segnali. Utilizzando l'apposita sezione "truth table" possiamo fare un controllo sulle funzionalità delle singole LUT :

In seguito alla implementazione, bisogna tenere conto oltre che dai ritardi degli elementi circuitali quali le LUT, anche i ritardi dovuti ai collegamenti elettrici, che sono rappresentati in verde. Infatti dalla simulazione post implementazione notiamo un ritardo iniziale dell'output nettamente maggiore (6.500 ns) di quello ottenuto nella simulazione post sintesi:



La realizzazione del chip fisico non è richiesta dalla traccia del progetto.

Grazie dell’attenzione !