Select Carry Adder

Carmelo Gugliotta Mat:213477

Anno Accademico: 2021/2022

Contents

1 Introduzione

Per incrementare la velocità di un computer digitale, un progettista ha la necessità di utilizzare componenti e circuiti molto veloci o di utilizzare componenti e circuiti lenti, ma organizzati in maniera efficiente così da ottenere la velocità desiderata.

Nei calcolatori digitali, la velocità di calcolo è limitata dal tempo di propagazione del riporto. Il calcolo della somma, in adder elementari, avviene così come verrebbe calcolata a mano, in maniera sequenziale (a partire dal bit meno significativo) e solo dopo che i bit, in posizione i-esima, sono stati sommati, il riporto (se generato) è propagato alla posizione successiva.

Un addizionatore elementare ad n *bit* è rappresentato dal Ripple-Carry Adder (RCA), o addizionatore a propagazione del riporto. Questo circuito richiede n Full-Adder (FA) in cascata. La sua architettura è semplice, ma anche lenta. Il tempo di calcolo, infatti, dipende dal numero di bit, in quanto bisogna attendere che il riporto si propaghi dal primo all'ultimo FA per avere il risultato corretto.

È possibile implementare addizionatori che generano il riporto in maniera concorrente? Se si può fare, questo è il nostro obiettivo.

2 Carry Select Adder

2.1 Struttura

È possibile costruire un addizionatore più veloce del Ripple-Carry. Possiamo infatti, dividere l'addizionatore RCA (ad n *bit*) in x blocchi di m *bit*. La relazione che si andrà a ottenere è la seguente.

$$n = x \cdot m \tag{1}$$

Un primo blocco sarà costituito da un RCA ad m bit, e successivamente, i blocchi dal secondo al x-esimo saranno costituiti da due RCA ad m bit, uno con segnale $C_{in} = 0$ e l'altro con $C_{in} = 1$. Le uscite, saranno poi, selezionate da un Multiplexer che riceverà in ingresso come segnale di selezione, il riporto uscente dal blocco precedente.

Tramite tale struttura, è possibile notare che i calcoli eseguiti nei diversi blocchi, possono essere effettuati indipendentemente e ciò è garantito dal parallelismo introdotto all'interno del circuito. Chiaramente, questo vantaggio comporta un prezzo da pagare in termini di porte logiche usate.

Un addizionatore, così descritto, prende il nome di **Carry Select Adder** ed è mostrato in Figura 1, una sua versione basilare con n=16 e m=8:

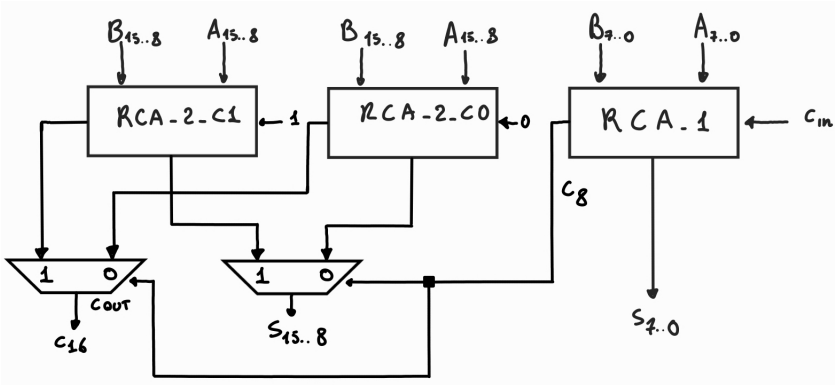


Figure 1: Addizionatore Carry Select

2.2 Complessità Temporale [Ritardo]

Analizziamo in dettaglio il ritardo. Osservando che, come riportato in precedenza, ogni blocco corrisponde a un RCA, i singoli blocchi avranno effettuato le loro somme in un tempo $T_{blocco} = m\tau_{FA}$. I primi m bit saranno disponibili dopo T_{blocco}. I successivi m bit saranno disponibili dopo un tempo $T_{blocco} + \tau_{mux}$ ove T_{mux} è il ritardo del multiplexer. Il tempo totale, con tale configurazione sarà quindi:

$$T_{CSA} = m \cdot \tau_{FA} + \tau_{mux} \cong (m+1) \cdot \tau_{FA} \tag{2}$$

A discapito della complessità circuitale, vi è un guadagno della complessità temporale.

2.3 Implementazione VHDL

2.3.1 Package

Generalmente è preferibile inserire tutte le costanti che definiranno il numero di bit in ingresso, e il numero di bit assegnato a ogni blocco, in unico file package, invece di copiare e incollare quest'ultimi ove necessario. In questo modo, se si vuole analizzare il comportamento del circuito al variare di un determinato parametro, è sufficiente aggiornare solo il file package.

```
1    package MyDef is
2        constant n:integer:=16;
3        constant m:integer:=4;
4    end package;
```

Listing 1: MyDef

2.3.2 Modulo Ripple Carry Adder

Andiamo a descrivere il primo componente: il **Il Ripple Carry Adder**. Quest'ultimo deve essere in grado di sommare 3 operandi, A e B, di m bit, e il riporto in ingresso c_{in} per generare in uscita il risultato S e il riporto c_{out}.

```
1    library IEEE;
2    library WORK;
3    use IEEE.STD_LOGIC_1164.ALL;
4    use WORK.MyDef.ALL;
5    entity RCA_v1 is
6        Port ( A : in STD_LOGIC_VECTOR (m-1 downto 0);
7               B : in STD_LOGIC_VECTOR (m-1 downto 0);
8               cin : in STD_LOGIC;
9               cout: out STD_LOGIC;
10              S : out STD_LOGIC_VECTOR (m-1 downto 0));
11   end RCA_v1;
12   Architecture behav of RCA_v1 is
13       signal carry: STD_LOGIC_VECTOR(m downto 0);
14       signal p,g: STD_LOGIC_VECTOR(m-1 downto 0);
15       begin
16           p <= (A xor B);
17           g <= (A and B);
18           carry(m downto 1) <= (p(m-1 downto 0) and
19                       carry(m-1 downto 0))or g(m-1 downto 0);
20           S<=p xor carry(m-1 downto 0);
21           carry(0) <= cin;
22           cout <= carry(m);
23   end behav;
```

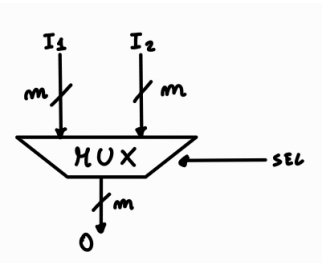
Listing 2: Modulo Ripple Carry Adder

2.3.3 Modulo Multiplexer

Se osserviamo la struttura descritta in Figura 1, notiamo un altro importantissimo componente : Il **Multiplexer**. Quest'ultimo è un dispositivo che può ricevere in ingresso 2^n segnali, e tramite n segnali di controllo, sintetizza un unico segnale in output tra quelli ricevuti in ingresso.

```
1   --Modulo Multiplexer ad M bit in ingresso ed M bit in uscita(2 ingressi 1
    uscita)
2   library IEEE;
3   library WORK;
4   use IEEE.STD_LOGIC_1164.ALL;
5   use WORK.MyDef.ALL;
6
7   entity MyMuxM_v1 is
8       port ( I0,I1: in STD_LOGIC_VECTOR (m-1 downto 0);
9              Sel: in STD_LOGIC;
10             O: out STD_LOGIC_VECTOR(m-1 downto 0));
11  end MyMuxM_v1;
12
13  Architecture behav of MyMuxM_v1 is
14  begin
15      with Sel select
16       O <=
17         I0 when '0',
18         I1 when '1',
19         (others=>'X') when others;
20  end behav;
21
22  --Modulo Multiplexer a 1 BIT
23  library IEEE;
24  library WORK;
25  use IEEE.STD_LOGIC_1164.ALL;
26  use WORK.MyDef.ALL;
27
28  entity MyMux2_v1 is
29      port ( I0,I1: in STD_LOGIC;
30             Sel: in STD_LOGIC;
31             O: out STD_LOGIC);
32  end MyMux2_v1;
33  Architecture behav of MyMux2_v1 is
34  begin
35          O<=I0 when Sel='0' else
36             I1 when Sel='1' else 'X';
37  end behav;
```

Listing 3: Modulo Multiplexer



Sel	O
0	I1
1	I2

Figure 2: Rappresentazione Multiplexer e Tabella di Verità associata

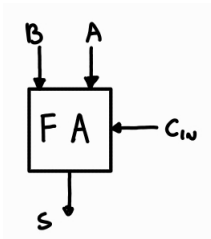
2.3.4 Modulo Full-Adder

Rimane da descrivere l'ultimo componente : Il **Full-Adder**. Quest'ultimo servirà per implementare l'estensione del bit del segno per consentire la somma di operandi espressi in complemento a 2. Il Full adder, permette di sommare 3 bit, l' operando A, l' operando B e il riporto in ingresso C_{in}, e trasmette in output 2 bit, il risultato della somma S, e il riporto in uscita c_{out}

```
    library IEEE;
    library WORK;
    use IEEE.STD_LOGIC_1164.ALL;
    use WORK.MyDef.ALL;

    entity FA is
        Port (  A : in STD_LOGIC;
                B : in STD_LOGIC;
                cin: in STD_LOGIC;
                S:  out STD_LOGIC);
    end FA;
    architecture behav of FA is
    signal cout,p,g : STD_LOGIC;
    begin
        S <= A xor B xor cin;
        p   <= A xor B;
        g   <= A and B;
        cout <= g or (p and cin);
    end behav;
```

Listing 4: Modulo Full-Adder



A	B	C_{in}	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Figure 3: Rappresentazione Full-Adder e Tabella di Verità associata

2.3.5 Implementazione Carry Select Adder e Schema Logico

Non ci resta che organizzare opportunamente le componenti precedentemente descritte, per ottenere il **Carry Select Adder**.

```
library IEEE;
library WORK;
use IEEE.STD_LOGIC_1164.ALL;
use WORK.MyDef.ALL;
entity CSAdd_nbit is
    Port ( A : in STD_LOGIC_VECTOR (n-1 downto 0);
           B : in STD_LOGIC_VECTOR (n-1 downto 0);
           cin : in STD_LOGIC;
           S : out STD_LOGIC_VECTOR (n downto 0));
end CSAdd_nbit;
```

```
11
12 Architecture Struct_v1 of CSAdd_nbit is
13
14 component RCA_v1 is
15     Port ( A : in STD_LOGIC_VECTOR (m-1 downto 0);
16            B : in STD_LOGIC_VECTOR (m-1 downto 0);
17            cin : in STD_LOGIC;
18            cout : out STD_LOGIC;
19            S : out STD_LOGIC_VECTOR (m-1 downto 0));
20 end component;
21
22 component FA is
23     Port (  A : in STD_LOGIC;
24             B : in STD_LOGIC;
25             cin: in STD_LOGIC;
26             S:  out STD_LOGIC);
27 end component;
28
29 component MyMuxM_v1 is
30 port ( I0,I1: in STD_LOGIC_VECTOR (m-1 downto 0);
31         Sel: in STD_LOGIC;
32         O: out STD_LOGIC_VECTOR(m-1 downto 0));
33 end component;
34
35 component MyMux2_v1 is
36     port ( I0,I1: in STD_LOGIC;
37         Sel: in STD_LOGIC;
38         O: out STD_LOGIC);
39 end component;
40
41 signal carry: STD_LOGIC_VECTOR(n/m downto 0);
42 signal carryalpha:STD_LOGIC_VECTOR(n/m downto 0);
43 signal carrybeta:STD_LOGIC_VECTOR(n/m downto 0);
44 signal Salpha: STD_LOGIC_VECTOR(n downto m);
45 signal Sbeta: STD_LOGIC_VECTOR(n downto m);
46 signal bit0: STD_LOGIC:='0';
47 signal bit1: STD_LOGIC:='1';
48 begin
49     myFOR: for i in 0 to (n/m)-1 generate -- devo generare le istanze dell'RCA
50         myIfFirst: if i=0 generate
51             RCA_First: RCA_v1
52                     port map (A(m-1 downto 0),
53                     B(m-1 downto 0),
54                     carry(0),carry(1),S(m-1 downto 0));
55         end generate;
56         myIfOthers: if i>0 generate
57             RCA_OthersC0: RCA_v1
58             port map(A((m*(i+1))-1 downto m*i ),
59                 B( (m*(i+1))-1 downto m*i ),
60                 bit0,carryalpha(i),
61                 Salpha((m*(i+1))-1 downto m*i));
62
63             RCA_OthersC1: RCA_v1
64             port map(A((m*(i+1))-1 downto m*i ),
65                 B( (m*(i+1))-1 downto m*i ),
66                 bit1,carrybeta(i),
67                 Sbeta((m*(i+1))-1 downto m*i));
68
69             MUX_SelectRES: MyMuxM_v1
70             port map(Salpha((m*(i+1))-1 downto m*i),
```

```
                Sbeta((m*(i+1))-1 downto m*i),
                carry(i),S((m*(i+1))-1 downto m*i));

            MUX_selectCout: MyMux2_v1
                port map(carryalpha(i),carrybeta(i),
                        carry(i),carry(i+1));
        end generate;
    end generate;
    myFAC0: FA port map(A(n-1),B(n-1),bit0,Salpha(n));
    myFAC1: FA port map(A(n-1),B(n-1),bit1,Sbeta(n));
    MUX_SelectLastBit: MyMux2_v1
            port map(Salpha(n), Sbeta(n), carry(n/m), S(n));
    carry(0)<= cin;
end Struct_v1;
```

Listing 5: Modulo Carry Select Adder

Lo schema logico corrispondente, definendo n pari a 16 e m pari a 4, è il seguente:

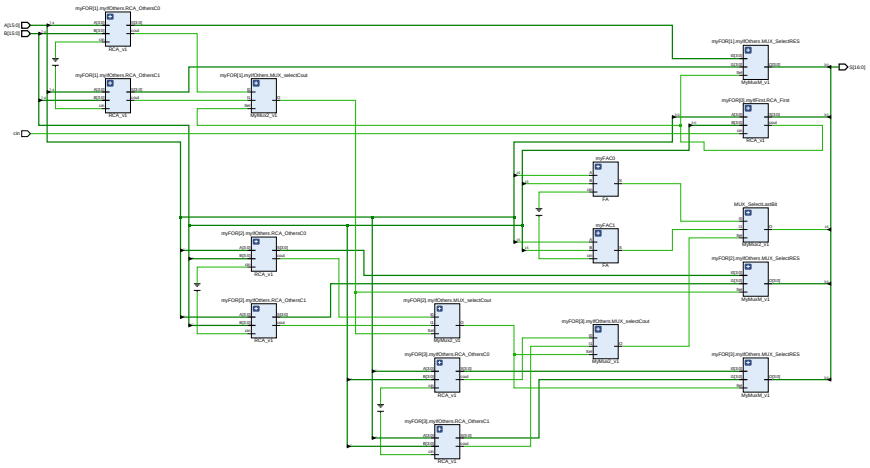


Figure 4: Schema Logico

2.4 Simulazione Comportamentale

Modelliamo ora gli aspetti salienti dell'ambiente in cui il circuito si troverà a operare, definendo un TestBench esaustivo, che permette di specificare gli stimoli in ingresso e di collezionare risultati, per la simulazione del circuito e per la verifica della correttezza delle operazioni.

```
library IEEE;
library WORK;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_1164.ALL;
use WORK.MyDef.ALL;

entity simCSAdd_nbit is
---port();
end simCSAdd_nbit;

architecture Behavioral of simCSAdd_nbit is
```

```
component CSAdd_nbit is
    Port ( A : in STD_LOGIC_VECTOR (n-1 downto 0);
        B : in STD_LOGIC_VECTOR (n-1 downto 0);
        cin : in STD_LOGIC;
        S : out STD_LOGIC_VECTOR (n downto 0));
end component;
signal IA : STD_LOGIC_VECTOR (n-1 downto 0);
signal IB : STD_LOGIC_VECTOR (n-1 downto 0);
signal Icin : STD_LOGIC;
signal OS : STD_LOGIC_VECTOR (n downto 0);
signal TrueRis,Error:Integer;
begin
    circ: CSAdd_nbit port map(IA,IB,Icin,OS);
    Icin<='0';
    process
    begin
        for va in -(2**(n-1)) to (2**(n-1)-1) loop
            for vb in -(2**(n-1)) to (2**(n-1)-1) loop
                IA<=CONV_STD_LOGIC_VECTOR(va,n);
                IB<=CONV_STD_LOGIC_VECTOR(vb,n);
                TrueRis<=va+vb;
                wait for 10ns;
            end loop;
        end loop;
    end process;
    Error<=TrueRis-CONV_INTEGER(SIGNED(OS));
end Behavioral
```

Listing 6: Test Bench

Si noti che i segnali A e B che immettiamo in ingresso hanno valori appartenenti all'intervallo $[-2^{n-1}, 2^{n-1}-1]$, nel nostro caso specifico $[-32768, +32767]$; C_{in} invece è fissato ad 0. Inoltre, per verificare la correttezza dei risultati ottenuti, si è introdotto un segnale intero *TrueRis*, pari a $va + vb$ e un altro segnale intero *Error* che è pari alla differenza tra il risultato atteso e il risultato calcolato (Valori diversi da 0 determinano un errore di calcolo).

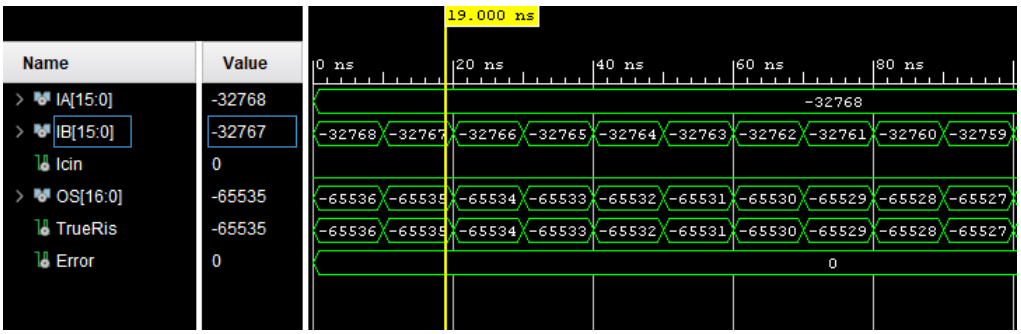
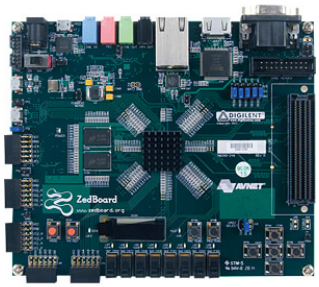


Figure 5: Grafico Simulazione Comportamentale

Dal grafico è possibile osservare, che ogni ingresso ricevuto viene elaborato con **ritardo nullo**. Questo non è possibile nella realtà, ed avviene in quanto si sta eseguendo una simulazione comportamentale.

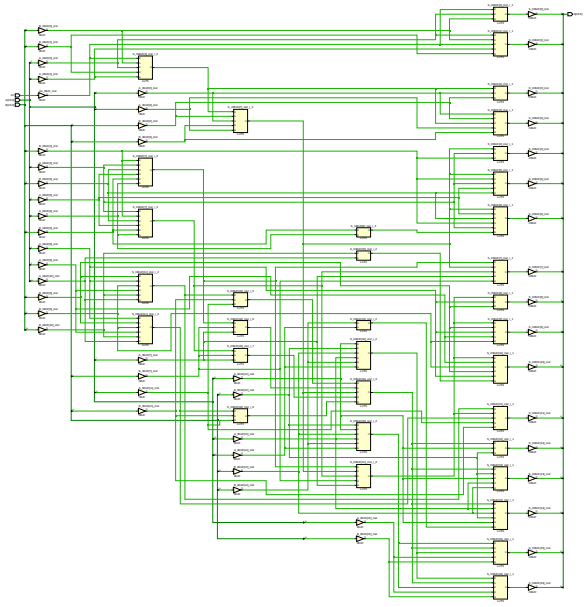
Per effettuare un' analisi circuitale più approfondita, è necessario effettuare la **sintesi del circuito** e successivamente la **simulazione con timing**.

2.5 Sintesi Circuitale



Le board sulle quali vengono sviluppati circuiti di tal tipo, sono dispositivi elettronici, formati da un circuito integrato le cui funzionalità logiche di elaborazione sono appositamente programmabili e modificabili tramite opportuni linguaggi di descrizione hardware, e vengono detti **Field Programmable Gate Aways** (solitamente abbreviato in **FPGA**). Per cui un chip FPGAs, mette a disposizione dei progettisti, una matrice di blocchi logici configurabili, connessi fra loro attraverso interconnessioni programmabili. Ogni blocco logico è un elemento di memoria all'interno della quale è possibile memorizzare una o più **Look-up Table** (LUT), le quali sono tabelle di verità, utilizzate per implementare funzioni logiche ad n ingressi. Lungo il perimetro della Look-up Table sono disposti buffer d'ingresso e di uscita, detti rispettivamente **IBUF** e **OBUF**, che permettono eventuali comunicazioni.

Generalmente i Look-Up Table sono caratterizzati da una dimensione fissata, e nello specifico nella ZedBoard (board in utilizzo) presentano al più 6 ingressi e 1 uscita; infine sono fortemente riutilizzabili, in quanto essendo composte da una memoria RAM (memoria volatile), la loro caratterizzazione può essere riprogrammata a ogni accensione.
Il risultato della sintesi è il seguente:



Site Type	Used	Fixed	Available	Util%
Slice LUTs	25	0	53200	0.05
LUT as Logic	25	0	53200	0.05
LUT as Memory	0	0	17400	0.00
Slice Registers	0	0	106400	0.00
Register as Flip Flop	0	0	106400	0.00
Register as Latch	0	0	106400	0.00
F7 Muxes	0	0	26600	0.00
F8 Muxes	0	0	13300	0.00

Figure 6: Sintesi Carry Select Adder

2.6 Analisi simulazione con Timing

Effettuiamo ora la simulazione, tenendo conto di tutti i comportamenti reali del circuito (Ritardo LUT e Ritardo Collegamenti), introducendo quindi una **non idealità totale**.

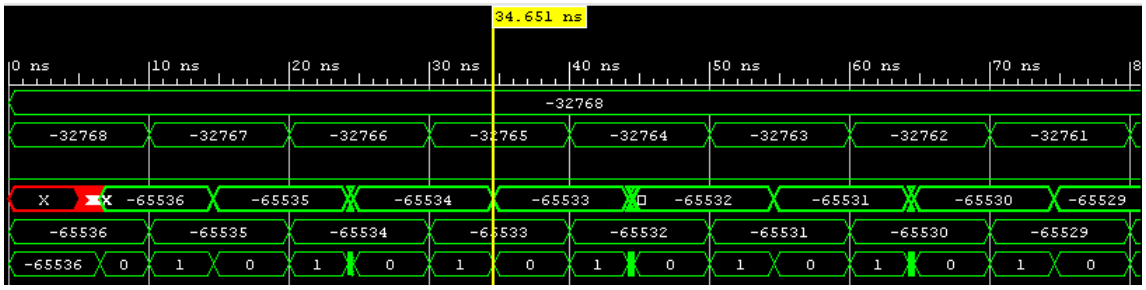


Figure 7: Grafico Simulazione con Timing

Osservando il grafico, si ha che dati due ingressi A e B con rappresentazione in complemento a due, il circuito, per calcolare la loro somma e restituire il risultato corretto in uscita impiega:

$$\tau_{CarrySelect} \cong 4,681ns$$

Inoltre precedentemente, si era introdotto un segnale *TrueRis*, che permetteva di verificare la correttezza del risultato ottenuto in uscita; si noti che in Figura 5, il valore di tale segnale era costante e pari a 0, invece in Figura 7, il valore assume valori diversi da 0, ma solo in alcuni intervalli di tempo, determinati dal **tempo di assestamento** del dispositivo; questa situazione si verifica, poiché gli n bit in uscita, vengono calcolati ognuno con un tempo differente e solo che dopo tutti i bit si assestano, l'uscita diventa valida.

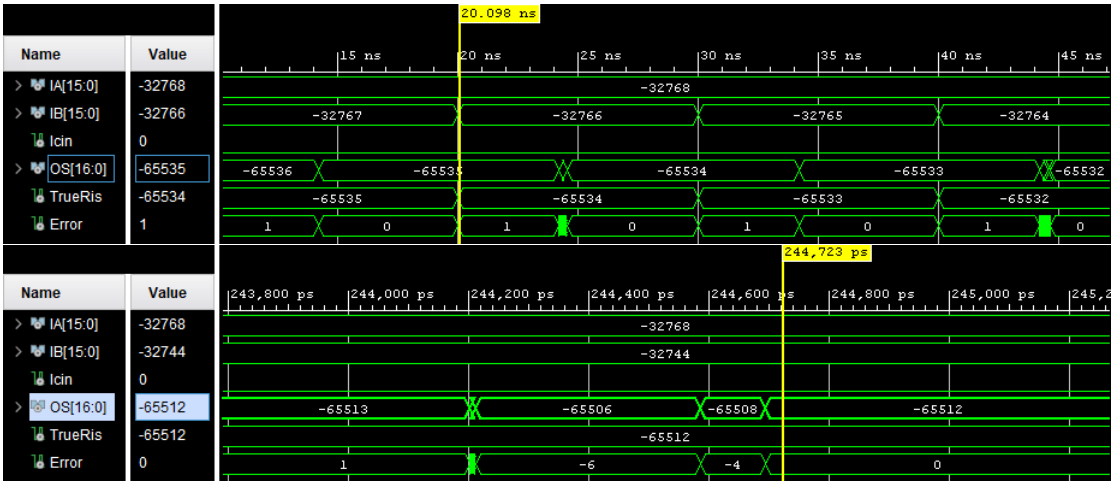


Figure 8: Valori di Assestamento

2.7 Osservazioni

Il sommatore Carry-Select a 16 bit, implementato e analizzato è costituito da sommatori Ripple-Carry a 4 bit. È possibile però, sintetizzare un Carry-Select a 16 bit costituito da Blocchi RCA con dimensione uniforme pari a 8 bit, visibile in Figura 1. Chiaramente, ci si aspetta, dal nuovo addizionatore ottenuto, un miglioramento in termini di porte logiche utilizzate, ma un peggioramento per quanto riguarda la Complessità Temporale (Ritardo).
Per verificare, con dettaglio, questi due aspetti, effettuiamo la Sintesi e l'implementazione con successiva Analisi con Timing. Si ottiene quindi:

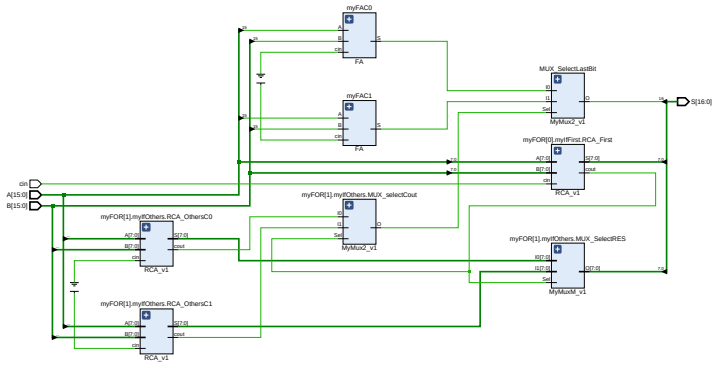
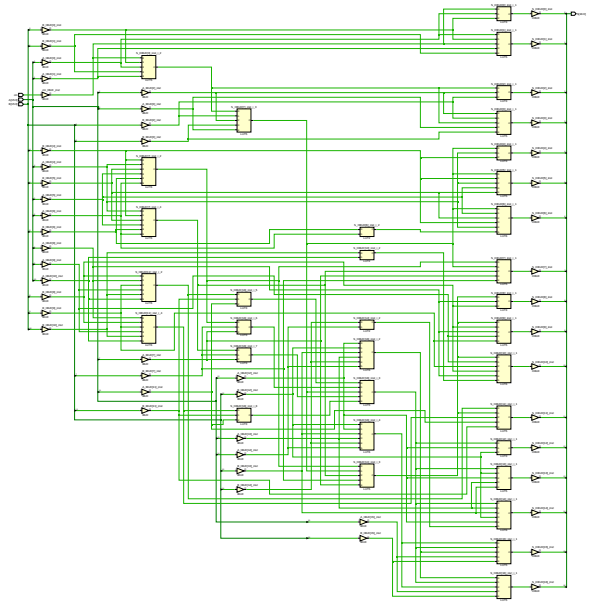


Figure 9: Schema Logico Carry-Select n=16 bit m=8 bit



Site Type	Used	Fixed	Available	Util%
Slice LUTs	27	0	53200	0.05
LUT as Logic	27	0	53200	0.05
LUT as Memory	0	0	17400	0.00
Slice Registers	0	0	106400	0.00
Register as Flip Flop	0	0	106400	0.00
Register as Latch	0	0	106400	0.00
F7 Muxes	0	0	26600	0.00
F8 Muxes	0	0	13300	0.00

Figure 10: Sintesi Carry-Select n=16 bit m=8 bit

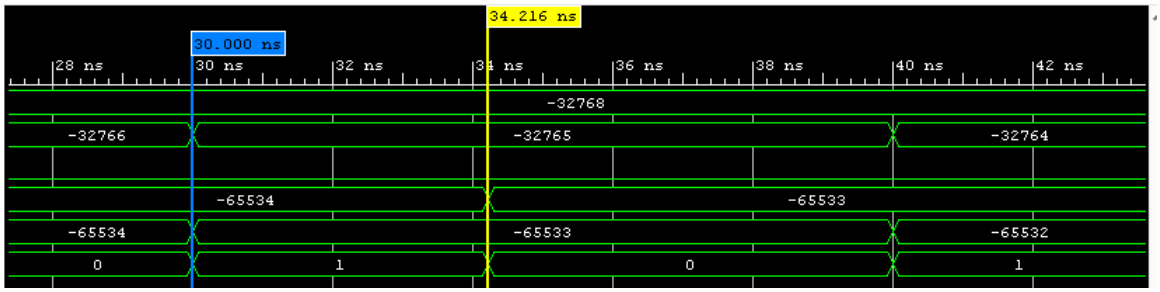


Figure 11: Simulazione con Timing

Il Carry-Select a 16 bit con sommatori Ripple-Carry a 8 bit (2 blocchi), dati due ingressi A e B con rappresentazione in complemento a due, per calcolare la loro somma e restituire il risultato corretto in uscita impiega:

$$\tau_{CarrySelect} \cong 4,216ns$$

Osservando le Figure 9,10,11 si ottengono risultati contrastanti a ciò che si era affermato in precedenza. Gli strumenti utilizzati, sono ottimi per analizzare cosa succede per una specifica operazione di somma, e non per studiare con dettaglio il massimo tempo di calcolo di un circuito digitale; difatti se comparassimo i dati ottenuti, con i parametri di un Ripple-Carry a 16 bit (l'addizionatore di partenza che si era proposto di migliorare), quest'ultimo presenterebbe un ritardo minore.
I risultati contrastanti sono una conseguenza del fatto, che le due implementazioni sono caratterizzate da collegamenti diversi, con differenti lunghezze e quindi contributi delle interconnessioni, sul ritardo, dissimili. In termini di LUT, invece, può succedere, come in questo caso, che il sintetizzatore riesca a semplificare in maniera differente le funzioni logiche, utilizzando così 25 LUT con $n = 16, m = 4$ e 27 LUT con $n = 16, m = 8$.

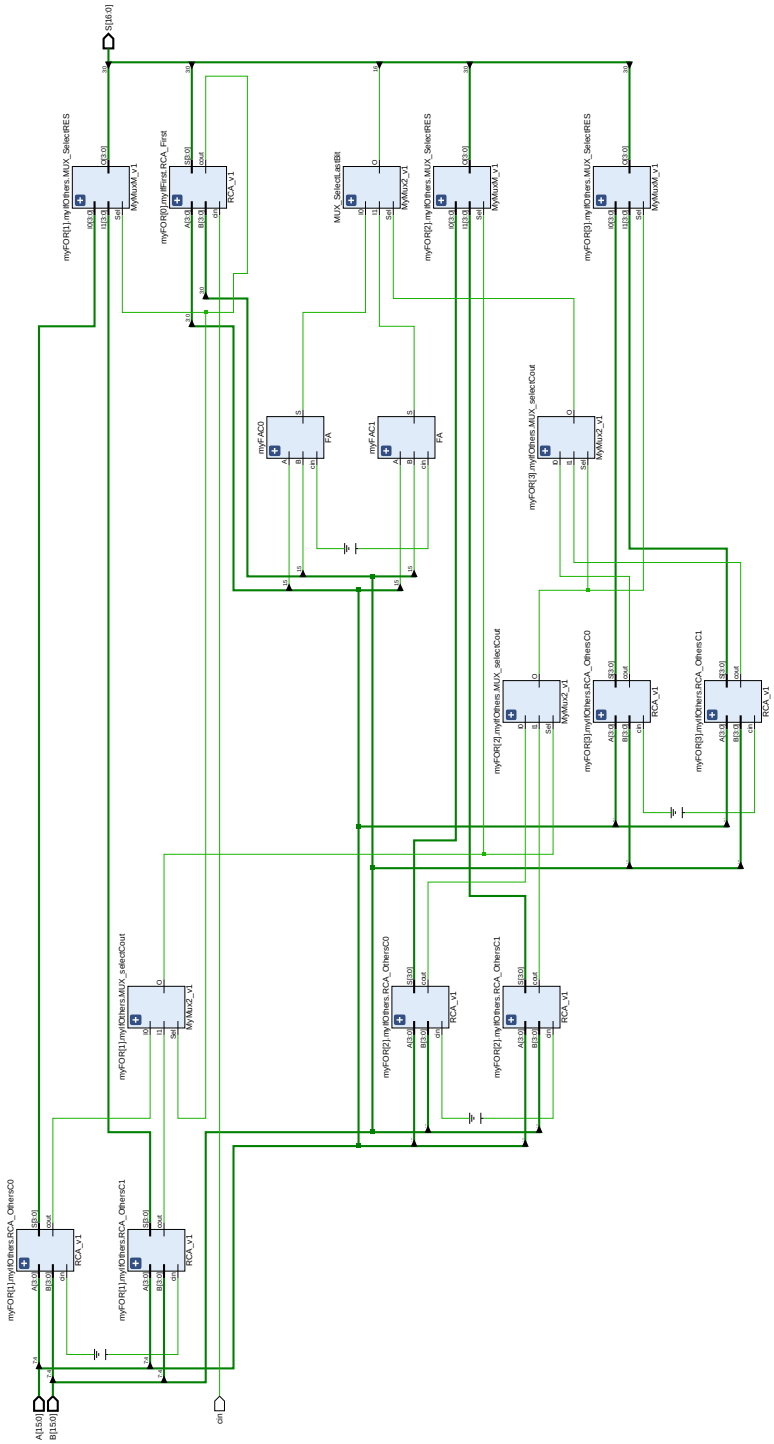


Figure 12: Ref. figura 4

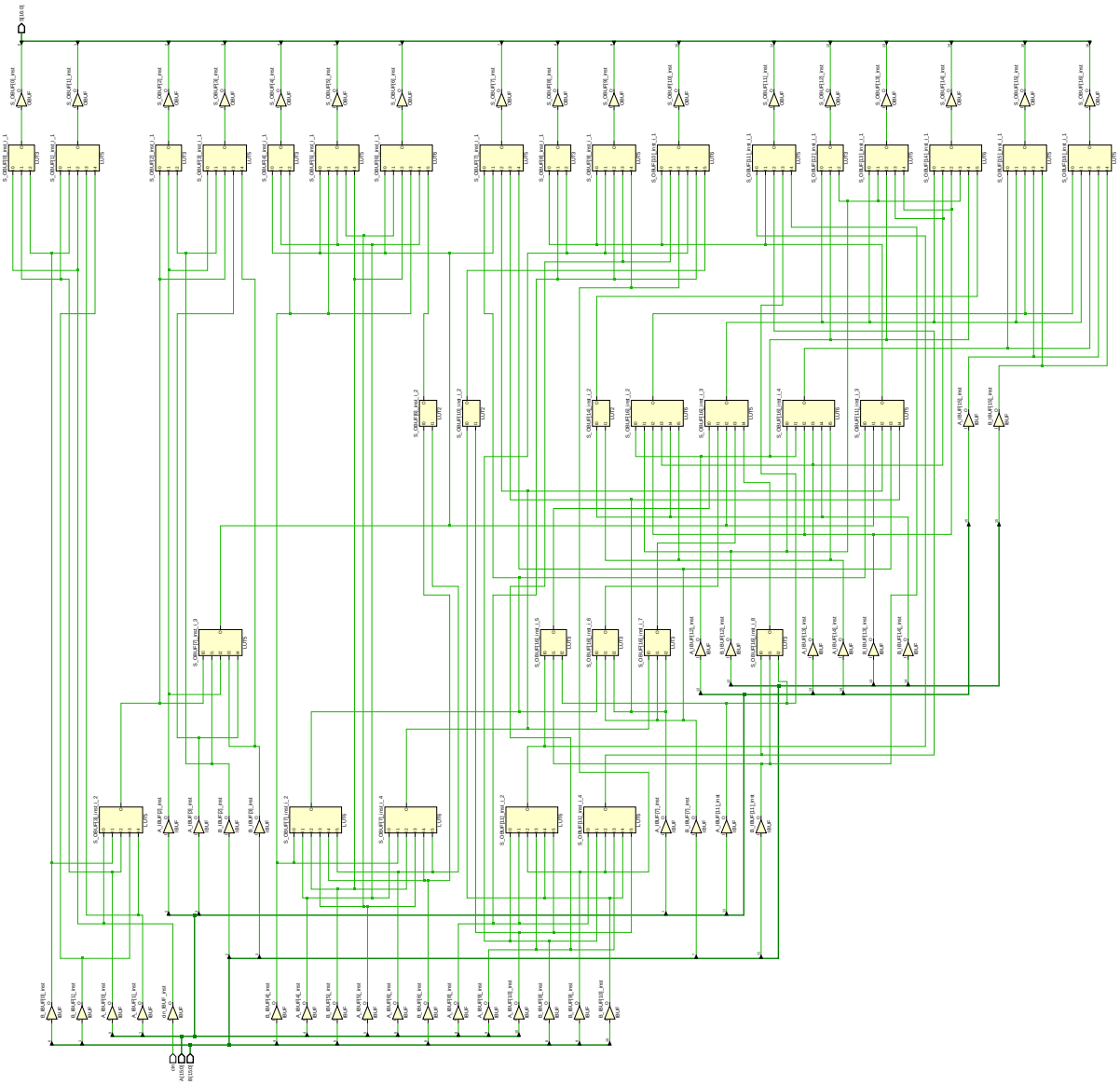


Figure 13: Ref. figura 6

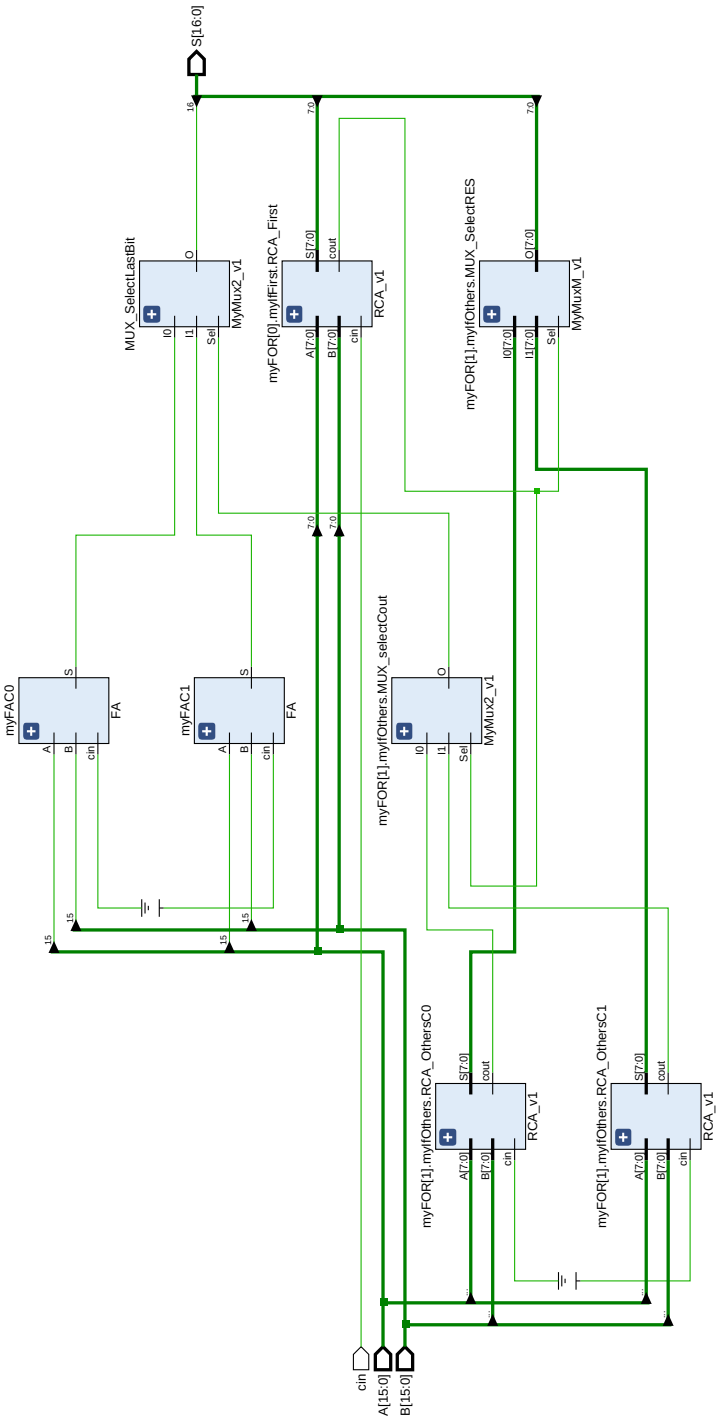


Figure 14: Ref. figura 9

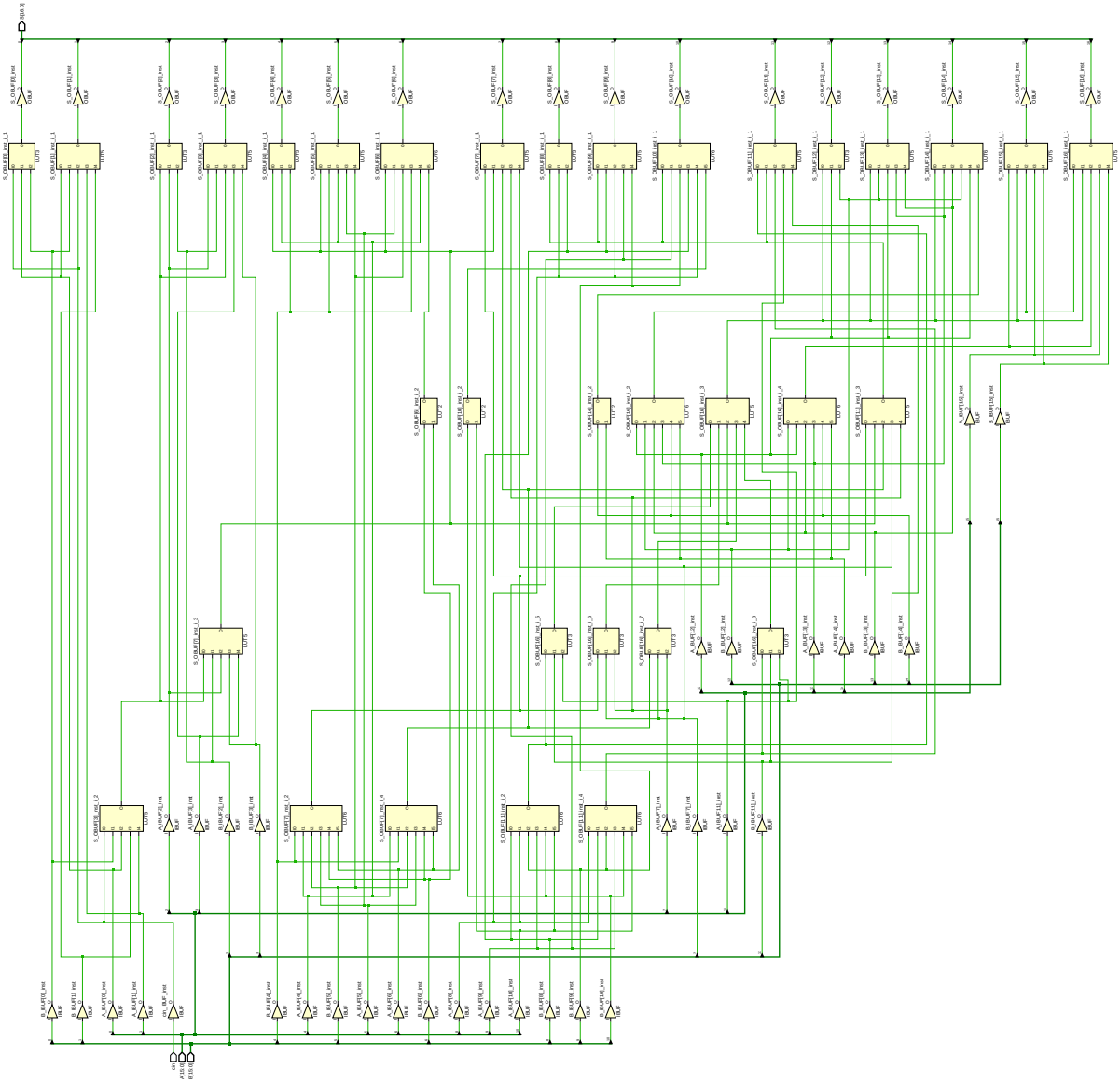


Figure 15: Ref. figura 10