POLITECNICO DI TORINO

Collegio di Ingegneria Elettronica, delle Telecomunicazioni e Fisica

Corso di Laurea Magistrale
in Ingegneria Elettronica (Electronic Engineering)

Tesi di Laurea Magistrale

Approximate Computing for Floating Point Arithmetic Operators. Design and Implementation

Relatore

Prof. Guido Masera

Correlatore

Ing. Giuseppe Notarangelo

Ing. Francesco Pappalardo

Candidato

Luca Francesco Perroni

Anno accademico 2017-2018

Indice

Indice Formule

1. Introduzione

I metodi di rappresentazione di un numero reale in un calcolatore sono oggetto di studio fin dagli albori dell'era digitale. La rappresentazione di un numero reale, potenzialmente infinito, è limitata e dipendente dal numero di bit che si intende usare. Si tratta di stabilire a priori un numero di bit tali da poter rappresentare il più efficacemente possibile il dato. I due metodi di rappresentazione di numeri reali, attualmente più in uso, sono: *Fixed-point* e *Floating-point.*

La rappresentazione *fixed-point* è di tipo *posizionale*, ovvero, le cifre a sinistra della virgola vengono moltiplicate per potenze positive della base, e quelle a destra per potenze negative. Come esempio, si supponga di voler rappresentare un numero reale avendo a disposizione 5 bit, di cui 3 per la parte intera e 2 per la parte frazionaria. Il numero 4.25 viene convertito come segue:

1. *Conversione parte intera*: $4 : 2 = 2 + resto\ 0$
   $2 : 2 = 1 + resto\ 0$
   $1 : 2 = 0 + resto\ 1$ ↑

   La parte intera in rappresentazione binaria viene ottenuta dai resti di ogni divisione in ordine contrario di computazione: $4_{(10)} = 100_{(2)} = 2^2 \cdot 1 + 2^1 \cdot 0 + 2^0 \cdot 0$

2. *Conversione parte frazionaria*: $0.25 \cdot 2 = 0.5$
   $0.50 \cdot 2 = 1{,}0$ ↓

   La parte frazionaria viene ottenuta dalla parte intera del risultato della moltiplicazione: $0.25_{(10)} = 0.01_{(2)} = 2^{-1} \cdot 0 + 2^{-2} \cdot 1$

In conclusione, unendo la parte intera con quella frazionaria si ottiene:

$$4.25_{(10)} = 100.01_{(2)} = 2^2 \cdot 1 + 2^1 \cdot 0 + 2^0 \cdot 0 + 2^{-1} \cdot 0 + 2^{-2} \cdot 1$$

Con la rappresentazione *floating-point*, a parità di bit della *fixed-point*, è possibile estendere l'intervallo di rappresentazione, lasciando invariato il numero totale di numeri esprimibili. Inoltre, mentre nell'aritmetica *fixed-point* due numeri successivi sono uniformemente spaziati, nell'aritmetica *floating-point* la densità non è uniforme, avendo distanza maggiore per numeri molto grandi e minore per quelli molto piccoli. Questo si traduce in aumento di precisione, come mostrato in Figura 1.1.

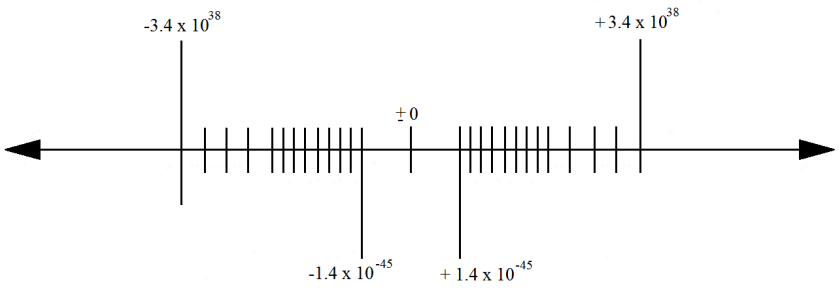


Figura 1.1: Intervallo e densità dei numeri floating-point su 32 bit

Di contro, l'implementazione hardware di questa aritmetica, rispetto alla *fixed-point*, comporta maggior consumo di area e potenza.

Il lavoro di tesi mira ad analizzare e ad applicare tecniche di *approximate computing* ai sommatori/sottrattori *floating-point*, con l'obiettivo di ridurre i consumi di area, potenza e velocità di computazione. Dopo aver approfondito lo stato dell'arte dei sommatori *floating-point* approssimati esistenti e lo studio delle varie tecniche di approssimazione dei sommatori *fixed-point*, si procede con la progettazione di un sommatore/sottrattore *floating-point* senza errore, così da avere un punto di riferimento con cui confrontare i sommatori/sottrattori approssimati che verranno implementati.

Ai fini di ottenere una stima dell'errore quanto più precisa possibile, e soprattutto strettamente legata all'hardware progettato, si è deciso di sintetizzare il tutto su *FPGA* o *Field Programmable Gate Array*, progettando un *system on-chip* in cui sia presente un processore *custom* (*Nios II,* [1]*,* [2]) e l'unità *floating-point*. In questo modo, grazie all'ausilio di un *firmware* [3], sarà semplice e veloce inviare all'unità *floating-point* dei dati di ingresso al fine di verificare il dato in uscita [4].

La prima fase di progettazione si esegue su *ModelSim*, di *Mentor Graphics*, ambiente di simulazione di *HDL* o *hardware description language*: si verificano con appositi *testbench* il funzionamento di ogni blocco *RTL* (*Register Transfer Level*) progettato in *VHDL,* o *VHSIC* (*Very High Speed Integrated Circuits*) *Hardware Description Language*.

Si passa, quindi, allo studio dell'errore del sommatore/sottrattore progettato, implementando un *system on-chip* sul software *Quartus* e sintetizzando il tutto su *FPGA*. Si procede alla stima dell'errore utilizzando i due parametri principali presenti in letteratura, detti, *MED* (*mean error distance*) e *MRED* (*mean relative error distance*).

La seconda fase di studio riguarda la progettazione di sommatori/sottrattori *floating-point* approssimati. Si tenta di replicare il sommatore presente in [21], in cui viene applicata la tecnica di approssimazione *LOA-k*, *Lower-part OR Adder*, con il fine di effettuare un confronto in termini di area e potenza dissipata. Inoltre, si progetta ed applica ad un sommatore classico, nello specifico un *RCA*, o *Ripple-Carry Adder* [5], una nuova tecnica di approssimazione (originale). Si utilizza, anche in questo caso, lo stesso metodo di studio dell'errore descritto in precedenza, quindi, valutando i due parametri, *MED* e *MRED*.

In ultima analisi, si effettua la sintesi dei modelli *HDL* utilizzando il software *Synopsys Design Compiler*, così da ottenere stime di potenza, area e frequenza massima di lavoro. Sarà necessario in quest'ultima fase applicare tecniche di ottimizzazione per evitare delle violazioni di timing. Le sintesi su tecnologia *BCD8SP* delle unità aritmetiche *floating-point* sono state realizzate dall'Ing. Giuseppe Notarangelo di STMicroelectronics.

Infine, si procede al confronto dei risultati, traendo le conclusioni finali del lavoro.

Nella prima parte dell'elaborato, Capitolo 2, dopo un breve riepilogo dello *standard IEEE 754*, si analizza l'algoritmo di somma e di sottrazione *floating-point*, includendo anche l'implementazione hardware. Nel Capitolo 3, si passa in rassegna lo stato dell'arte dei sommatori *floating-point* approssimati, avendo come obiettivo lo studio di metodi di ottimizzazione di area e potenza. Si focalizza l'attenzione sullo studio di caratterizzazione dell'errore nei circuiti aritmetici e sulle tecniche di *approximate computing* per sommatori

fixed-point, in quanto forniranno valide opportunità di ottimizzazione anche per i sommatori *floating-point*.

Il Capitolo 4 ha come oggetto l'implementazione *VHDL* dei sommatori/sottrattori *floating-point*: la prima fase di sviluppo riguarda lo studio e la progettazione di un'unità *floating-point* esatta, mostrando l'architettura e il funzionamento. Si passa in seguito al *design* delle unità aritmetiche approssimate: si metteranno in risalto le differenze, le gerarchie e le varie tecniche di *approximate computing* usate.

Nel Capitolo 5 si ha una breve spiegazione del *system-on chip* progettato e del *firmware* di funzionamento per il test delle unità aritmetiche. Si presentano, quindi, i risultati della sintesi su *FPGA*, confrontando le varie unità *floating-point* approssimate con quella di riferimento (esatta). Si effettua, pertanto, lo studio dell'errore tramite i parametri *MED* e *MRED*.

Successivamente, nel Capitolo 6, si mostrano i risultati della sintesi sul software S*ynopsys Design Compiler*, le ottimizzazioni attuate ed, infine, si effettua un confronto in termini di area e potenza.

Nel capitolo finale, Capitolo 7, si espongono le conclusioni finali del lavoro.

2. Numeri Floating-Point

2.1 Standard IEEE 754

Nello standard *IEEE Standard for Binary Floating-Point Arithmetic* (*ANSI/IEEE Std 754-1985)* sono definiti il formato per la rappresentazione dei numeri *floating-point*, le operazioni effettuabili e quattro metodi di arrotondamento. I tre formati di rappresentazione dei numeri *floating-point*, sono i seguenti:

- Precisione singola (32 bit);
- Precisione doppia (64 bit);
- Precisione quadrupla (128).

Tutti i numeri *floating-point* sono costituiti da quattro campi:

- Segno, *s*;
- Esponente, *E*;
- Mantissa, *M*;
- Base, *b*.

Il valore del numero rappresentato sarà calcolabile secondo la (2.1):

$$N = (-1)^s \cdot b^E \cdot M \qquad 2.1$$

Il campo segno, *s*, a cui viene riservato un bit, indica un numero positivo se assume valore 0, e negativo se assume valore 1. La base *b*, ha solitamente valore 2, ma esistono in commercio sistemi con base 8 o 16.

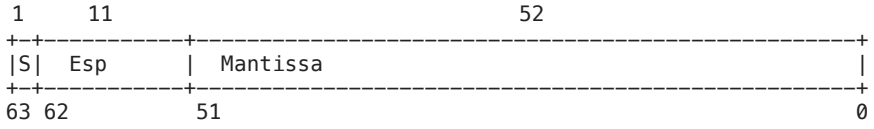
Per ognuno dei tre formati, si riserva un numero di bit di esponente e mantissa predefinito:

- Precisione singola: 8 bit di esponente e 23 di mantissa;
- Precisione doppia: 11 bit di esponente e 52 di mantissa;
- Precisione quadrupla: 15 bit di esponente e 112 di mantissa.

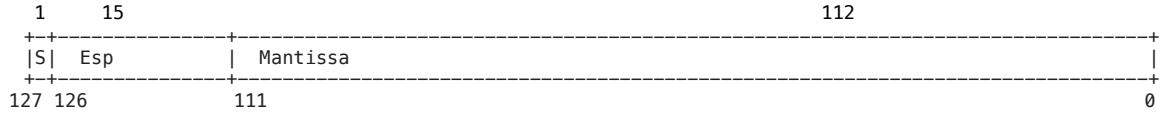
Si avranno, per ogni formato, le seguenti schematizzazioni:

- Precisione singola:

- Precisione doppia:



- Precisione quadrupla:



2.1.1 Precisione singola

Il formato a precisione singola, a 32 bit, è la rappresentazione più in uso tra quelle disponibili nello standard. Il bit più significativo è il bit di segno, successivamente vi sono gli 8 bit di esponente e, infine, i 23 di mantissa.

L'esponente si rappresenta in forma polarizzata, con polarizzazione pari a 127. Essendo di 8 bit, esso potrà assumere valori inclusi nell'intervallo 0-255. Durante l'esecuzione dei diversi algoritmi di calcolo, questa rappresentazione semplifica le operazioni di confronto tra esponenti. L'Equazione (2.2) di polarizzazione dell'esponente è:

$$b_{exp} = u_{exp} + 127 \qquad 2.2$$

Il simbolo b_{exp} sta per *biased exponent* e u_{exp} per *unbiased exponent*. Quindi, supponendo di voler rappresentare un numero *floating-point*, il cui esponente sia uguale a 7, si ha:

$$b_{exp} = 7 + 127 = 134_{(10)} = 10000110_{(2)}$$

La mantissa è in forma normalizzata se il campo E (esponente) è diverso da 0: in questa situazione l'*hidden* bit, ovvero il 24-esimo bit della mantissa, è sempre pari a 1. Viceversa, se il campo E è pari a 0, allora la mantissa sarà in forma denormalizzata e l'*hidden* bit sarà pari a 0. L'*hidden* bit, nella rappresentazione del numero, viene sempre omesso.

A seconda dei valori assunti dalla mantissa e dall'esponente, il numero appartiene alle categorie mostrate nella Tabella 2.1.1:

Tabella 2.1.1: Categorie di appartenenza dei numeri floating-point

Categoria	Esponente	Mantissa	Valore
Zeri	0	0	± 0
Numeri demoralizzati	0	$\neq 0$	$\pm(0.m) \cdot 2^{-126}$
Numeri normalizzati	$[1,254]$	$\forall$	$\pm(1.m) \cdot 2^{e-127}$
Infiniti	255	0	$\pm\infty$
NaN (Not a number)	255	$\neq 0$	NaN

Infine, la Formula (2.3) che permette la conversione da numero *floating-point* a numero decimale è:

$$value = (-1)^s \cdot 2^{b_{(exp)}-127} \cdot \left(1 + \sum_{i=1}^{23} b_{23-i} \cdot 2^{-i}\right) \qquad 2.3$$

Il simbolo b_i rappresenta l'i-esimo bit della mantissa.

Di seguito l'intervallo di rappresentabilità, Figura 2.1.1:

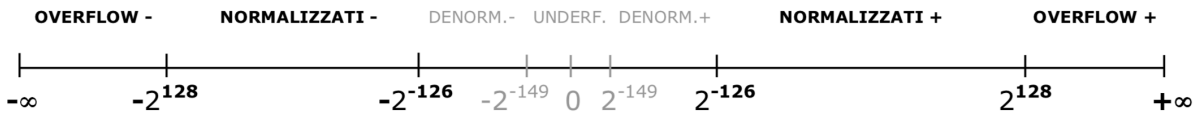


Figura 2.1.1: Intervallo di rappresentabilità.

In ultima analisi, lo standard prevede quattro tipi diversi di arrotondamento o *rounding* [6]: *rounding to the nearest even, rounding to* $-\infty$*, rounding to* $+\infty$ e *rounding to zero.*

- *Rounding to the nearest even*: arrotonda il numero al valore più vicino. Quando si è a metà dell'intervallo si arrotonda al valore più vicino pari;
- *Rounding to* $-\infty$: arrotonda il numero verso l'infinito positivo;
- *Rounding to* $+\infty$: arrotonda il numero verso l'infinito negativo;
- *Rounding to zero*: conosciuto anche troncamento, se il numero è positivo si arrotonda verso l'*infinito negativo*, viceversa, se il numero è negativo verso l'*infinito positivo.*

Il primo metodo è quello che garantisce maggiore precisione e stabilità numerica. Durante l'esecuzione degli algoritmi di somma/sottrazione vengono effettuati numerosi *shift* verso destra. Questi bit in 'eccesso', detti *guard bit*, vengono salvati nella parte meno significativa della mantissa. In particolare, il bit successivo al *LSB* (Least Significant Bit) della mantissa,

prende il nome di *round bit* (*R*) e l'*OR* logico di tutti gli altri bit rimanenti, viene chiamato *sticky bit* (*S*). L'algoritmo di calcolo del *rounding to the nearest even* esegue operazioni differenti a seconda del valore assunto da *R* e *S*. Si avranno così i seguenti casi:

1. *R=0* e *S*= ∀: troncamento di *R* ed *S*;
2. *R=1* e *S=0*: se il bit precedente a *R*, ovvero *LSB* della mantissa, è pari a 0 allora *R* ed *S* verrano troncati (caso 2.1), viceversa si sommerà 1 alla mantissa (caso 2.2);
3. *R=1* e *S=1*: viene sommato 1 alla mantissa.

In Tabella 2.1.2 si schematizzano i casi descritti:

Tabella 2.1.2: Regole per il rounding

Caso	LSB	R	S	Round
1	∀	1	1	1
2.1	0	1	0	0
2.2	1	1	0	1
3	∀	0	∀	0

In un'unità *floating-point*, il circuito combinatorio di *rounding* è tra i più dispendiosi in termini di potenza dissipata, secondo solo al sommatore della mantissa [23].

2.1.2 Algoritmo di somma

L'algoritmo di somma *floating-point* è costituito da quattro step fondamentali, ognuno dei quali riconducibile al *flow chart* di Figura 2.1.2.1.

1. *Compare and shift*: si applicano degli *shift* a destra alla mantissa con esponente minore di una quantità pari alla differenza tra i due esponenti.
2. *Addition*: a seguito dell'aggiornamento dell'esponente della mantissa, a cui è stata applicata l'operazione di *shift* al punto 1, si procede con la somma delle mantisse.
3. Normalization: se il risultato della somma è in condizione di *overflow*, cioè il 25-esimo bit della mantissa uguale ad '1', si esegue lo *shift* della mantissa a destra di una posizione, così da tornare ad avere l'*hidden* bit pari a 1. Si aggiorna l'esponente del risultato, incrementando il suo valore di 1.
4. *Rounding*: si arrotonda il risultato seguendo il metodo *rounding to the nearest even.* In caso di *overflow* si torna al punto 3.

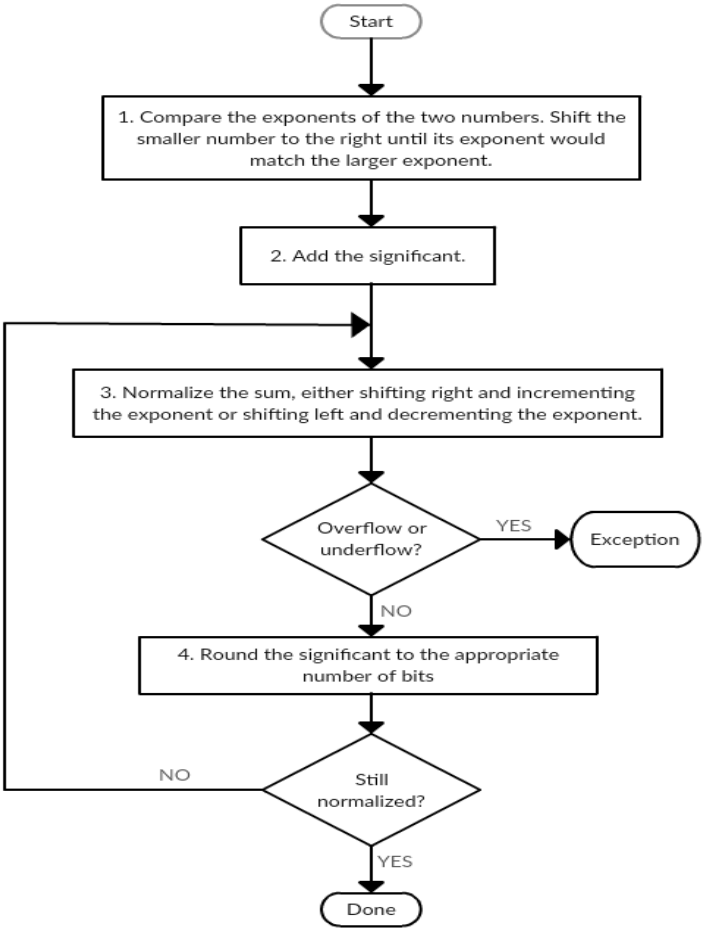


Figura 2.1.2.1: Flow chart algoritmo somma floating-point

Esempio: si supponga di voler effettuare la somma dei seguenti numeri in formato *floating-point,* $A = +2^8 \cdot 1.11100101110000100101110$ e $B = +2^5 \cdot 1.11010100000101000010101$.

1. *Compare and shift*: in questo step si sottrae l'esponente di *B* all'esponente di *A*. Il valore ottenuto corrisponde al numero di *shift* da effettuare alla mantissa di *B*.

$$B = +2^5 \cdot 1.11010100000101000010101 = +2^8 \cdot 0.00111010100000101000010\ 101$$

2. *Addition*: si sommano le due mantisse.

$$\begin{array}{lll} A = +2^8 \cdot & 1.11100101110000100101110 & + \\ B = +2^8 \cdot & 0.00111010100000101000010\ 101 & = \\ \hline C = +2^8 \cdot & 10.00100000100010011100000\ 101 & \end{array}$$

3. *Normalization*: in caso di *overflow*, si esegue lo *shift* a destra di una posizione e si aggiorna l'esponente.

$$C = +2^{8} \cdot 10.00100000010001001110000\ 101 = +2^{9} \cdot 1.00010000001000100111000\ 0\ 101$$

4. *Rounding*: si calcolano i bit di *round* e di *sticky.* Nell'esempio si ha *R*=0, che corrisponde al bit successivo al *LSB* della mantissa, e *S*=1, cioè l'OR logico dei restanti bit. In accordo con l'algoritmo *rounding to the nearest even*, si troncano i due bit *R* e *S.*

$$C = +2^{9} \cdot 1.00010000001000100111000$$

5. In caso di l'*overflow* dopo l'operazione di *rounding*, normalizzare nuovamente la mantissa.

2.1.3 Algoritmo di sottrazione

L'algoritmo di sottrazione *floating-point* è costituito di 7 step, Figura 2.1.3.1:

1. *Compare and shit*;
2. *Two's complement*: si calcola il complemento a due del sottraendo;
3. *Addition;*
4. *Check result:* se il risultato della somma è negativo si complementa a due il risultato;
5. Si eseguono degli *shift* a sinistra fino ad ottenere un *'1'* nel *MSB* (*most significant bit*) *della mantissa;*
6. *Rounding*;
7. *Normalization.*

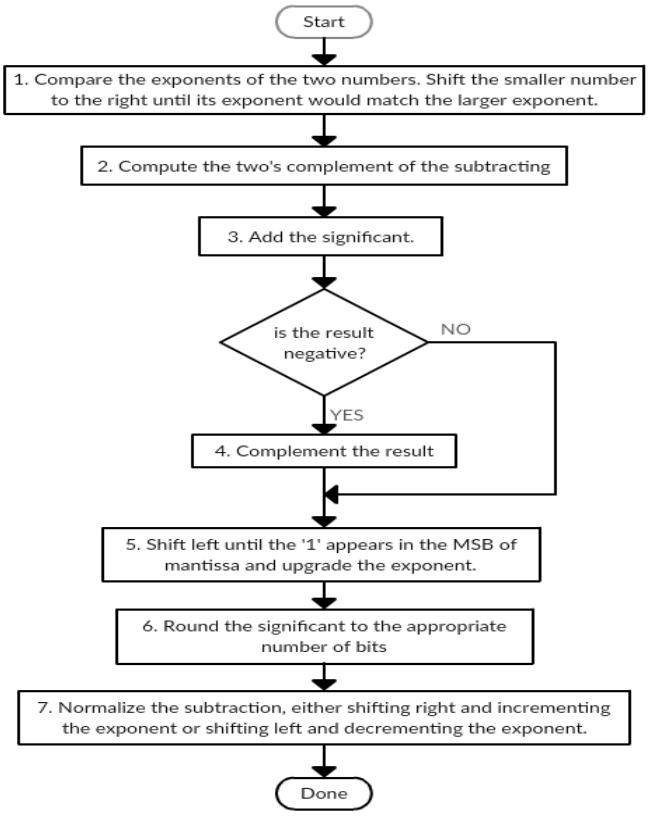


Figura 2.1.3.1: Flow chart algoritmo sottrazione floating-point

Esempio: si supponga di voler la somma algebrica, in questo caso la sottrazione, dei seguenti numeri, $A = +2^5 \cdot 1.00101001000010101111000$ e $B = -2^6 \cdot 1.00011101011100000001000$.

1. *Compare and shift*: si sottrae l'esponente di A all'esponente di B. Il valore ottenuto corrisponde al numero di shift.

$$A = +2^5 \cdot 1.00101001000010101111000 = +2^6 \cdot 0.10010100100001010111100\ 0$$

2. *Two's complement*: si calcola il two's complement del sottraendo.

$$B = -2^6 \cdot 1.00011101011100000001000 = -2^6 \cdot 0.11100010100011111111000$$

3. *Addition*: si sommano le mantisse dei due operandi.

$$\begin{array}{l} A = 0\ 0.10010100100001010111100\ 0 \cdot 2^6 \\ B = 1\ 0.11100010100011111111000\ \ \cdot 2^6 \\ \hline C = 1\ 1.01110111000101010110100\ 0 \cdot 2^6 \end{array}$$

4. *Check result*: se il risultato è negativo, '*1*' nel *MSB*, si effettua il *two's complement* del risultato.

$$C = 1\ 1.01110111000101010110100\ 0 \cdot 2^6 = 1\ 0.10001000111010101001100\ 0 \cdot 2^6$$

5. Si esegue uno *shift* a sinistra, e si aggiorna l'esponente, fino ad ottenere un '*1*' nell'*hidden* bit. Nell'esempio basta effettuare un solo *shift*.

$$C = 1\ 1.00010001110101010011000 \cdot 2^5 = -1.00010001110101010011000 \cdot 2^5$$

6. *Rounding*: si esegue l'algoritmo *rounding to the nearest even*. In questo caso si ha *S*=0 ed *R*=0, quindi non si effettua alcuna operazione sul risultato.
7. *Normalization*: in caso di *overflow* si normalizza. Nel caso mostrato come esempio non si effettua la normalizzazione, in quanto non si ha *overflow*.

Quindi il risultato finale è:

$$C = -1.00010001110101010011000 \cdot 2^5$$

2.2 Architettura di un sommatore/sottrattore

L'architettura in grado di sommare o sottrarre due numeri *floating-point* viene mostrata in Figura 2.2.1.

Il sottrattore differisce da un sommatore *floating-point* nella fase antecedente alla somma e nella fase di normalizzazione. Infatti, è necessario applicare il complemento a due alla mantissa dell'operando di segno negativo e, successivamente, dopo aver effettuato la somma delle mantisse, se il numero ottenuto è negativo, riapplicare il complemento a due. Tali operazioni si effettuano solo quando i segni dei due addendi sono discordi, in caso contrario l'algoritmo che verrà eseguito sarà quello della somma. Inoltre, quest'architettura è in grado di gestire qualsiasi tipo di numero *floating-point*, ovvero è valida sia per numeri normalizzati che per quelli denormalizzati.

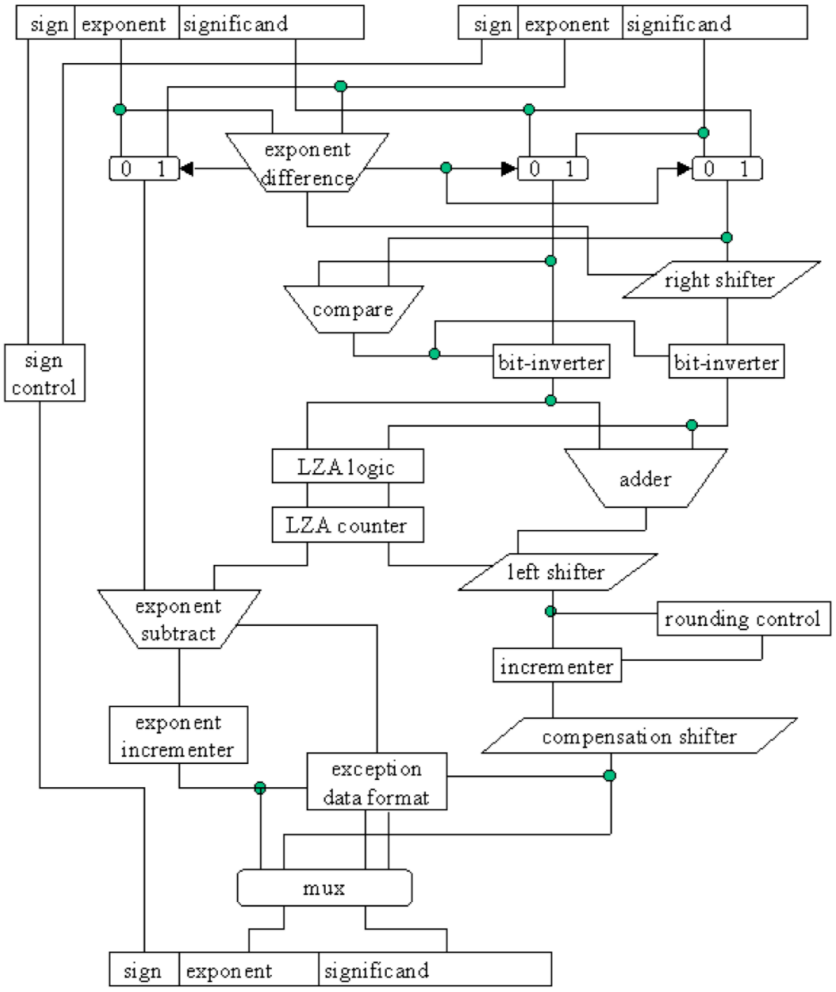


Figura 2.2.1: Architettura di un sommatore/sottrattore floating-point

L'architettura di Figura 2.2.1, rispecchia i vari step dei due algoritmi sin qui esposti: il risultato del blocco *exponent difference*, che esegue la differenza dei due esponenti, è collegato al blocco *right shifter*, il quale applica *shift* a destra alla mantissa con esponente minore, di un quantità pari al numero ricevuto dal blocco precedente. Successivamente, i blocchi *bit inverter* applicano il complemento a due, nel caso in cui i due addendi hanno segno discorde. Il sommatore, *adder*, riceve come *input*, le due mantisse, opportunamente processate dai blocchi precedenti, e fornisce il suo *output*, al blocco *left shifter*. Quest'ultimo riceve il valore per cui si intende eseguire l'operazione di *shift*, dai due blocchi *LZA* (*leading zero Anticipation*) *logic* e *counter*. Questi, processano il valore derivante dal blocco *exponent difference*, al fine di determinare in anticipo il numero di *shift* da effettuare al risultato. Infine, si applica il rounding (*rounding control*) al risultato precedentemente ottenuto, e, in caso di overflow, si procede con degli *shift* a sinistra, grazie al blocco *compensation shifter*. Il blocco *exception format*, controlla se il risultato sia un *NaN* (*not a number*).

3. Stato dell'arte dei sommatori Floating-Point

L'avanzamento tecnologico dei circuiti digitali integrati ha portato ad un aumento sostanziale della potenza dissipata. Per questo motivo, il consumo di potenza è diventato uno dei parametri più importanti in fase di progetto. Esistono alcune applicazioni in cui la precisone nei calcoli non deve essere molto accurata ed è proprio per questo tipo di applicazioni che l'approccio del *calcolo approssimato* è molto importante per ridurre i consumi di potenza e di area.

Si presentano in questo capitolo i parametri di caratterizzazione degli errori dei sommatori standard, e, dopo un'analisi delle tecniche di *approximate computing* applicate di quest'ultimi, si passa in rassegna lo stato dell'arte dei sommatori *floating-point* approssimati, analizzando le tecniche usate e i rispettivi risultati ottenuti. Infatti, le tecniche di *approximate computing* applicate ai sommatori *floating-point* riguardano solo la parte riguardante il sommatore di mantisse.

3.1 Caratterizzazione dell'errore

L'affidabilità dei sommatori approssimati viene valutata in accordo con le nuove metriche attualmente in uso [7]. L'*ED* (3.1), o *error distance*, si definisce come la distanza aritmetica tra il risultato approssimato, M^i, e quello esatto, M:

$$ED = |M^i - M| \tag{3.1}$$

La stima dell'errore relativo si ottiene dal parametro detto *RED* nella (3.2), ovvero *relative error distance*:

$$RED = \frac{ED}{M} \tag{3.2}$$

Infine, dal calcolo delle medie dei parametri *ED* e *RED* si ottengono i due valori di riferimento più importanti, che prendono il nome rispettivamente di *MED*, o *mean error distance*, e *MRED*, *mean relative error distance*. Il calcolo delle due medie si ottiene tramite la simulazione *Monte Carlo*, che consiste nella generazione di input del tutto casuali.

3.2 Tecniche di approssimazione sommatori fixed-point

Da [8] si ricavano le seguenti tipologie di sommatori approssimati:

1. *Speculative adders*: questa famiglia è costituita dai sommatori *ACA* [9], *Almost Correct Adder*. In un sommatore n-bit *ACA*, *k LSBs* sono usati per predire il *Carry-in* di ogni bit di somma ($n > k$). Il *delay* del *critical path* si riduce di $O(log(k))$, mentre il consumo di area aumenta di $O((n-k)k \cdot log(k))$;
2. *Segment adders*: la peculiarità di questa famiglia è la suddivisione del sommatore in blocchi che operano in modo concorrenziale. La propagazione del *carry* viene troncata in piccoli *segmenti*. I *segment adders* si suddividono in *ESA*, *Equal*

Segmentation Adder, *ETA*, *Error Tolerant Adder* , *ACAA*, *Accuracy Configurable Approximate Adder*.

2.1. *ESA*: come spiegato in [10], questo tipo di *adder* è ottenuto dal tecnica *DSEC*, o *Dynamic Segmentation with Error Compensation,* spiegata in [11], ma escludendo la compensazione dell'errore. Il *delay* è $O(log(k))$ e la complessità del circuito $O(nlog(k))$;

2.2. *ETA*: esistono due implementazione, chiamate *ETA I* ed *ETA II*. Data la scarsa efficienza del sommatore *ETA I* nel calcolo di numeri molto piccoli, si presenta la seconda tipologia, *ETA II*, nell'articolo [12]. Il *delay* è $O(log(k))$ e la complessità del circuito $O(nlog(k))$;

2.3. *ACAA*: come presentato in [13], questi sommatori possono cambiare il loro grado di accuratezza in *runtime*, al fine di trovare il giusto *trade-off* tra accuratezza, performance e consumi di potenza. Il delay è $O(log(k))$ e la complessità del circuito $O((n-k)log(k))$.

3. *Carry select adders*: i sommatori di questa tipologia hanno la caratteristica di avere tanti segnali all'interno della loro architettura per la computazione del risultato [9]. Le tipologie attualmente in letteratura sono: *SCSA* (*Specultive Carry Select Adder*), *CSA* (*Carry Skip Adder*), *CSPA* (*Carry Speculative Adder*), *CCA* (consistent carry approximate adder) e *GCSA* (*Generate Carry Speculation Adder*).

3.1. *SCSA*: questa tecnica è usata per il design di *speculative adder* con bassi tassi di *error rate* e di consumo di area aggiuntivo ad alte performance, [14]. Il *delay* del *critical path* ed il consumo di area, da [15], sono $t_{adder} + t_{mux}$ e $A_{adder} + A_{mux}$, con t_{adder} pari alla (3.3) e A_{adder} alla (3.4), e t_{mux} , A_{mux} rispettivamente delay ed area del multiplexer.

$$t_{adder} = O(log(k)) \tag{3.3}$$

$$A_{adder} = O(nlog(k)) \tag{3.4}$$

3.2. *CSA*: presentato in [16], l'architettura è simile al *SCSA*, ma ogni blocco *k* è composto da generatori di *sub-carry* e *sub-adder*. Da [15] si ricavano il delay, $O(log(k))$, e l'area, $A_{adder} + A_{carry}$, con A_{adder} uguale alle (3.4) e A_{carry} , area del circuito di generazione del *carry*.

3.3. *CSPA*: in [17] viene presentata questa variante del sommatore *SCSA*, in cui viene ridotto il *critical path* e il consumo di area. Da [15] si ottiene il delay, pari a $t_{adder} + t_{mux}$ e $A_{adder} + A_{mux} + A_{carry}$

3.4. *CCA*: questo sommatore è una variante del *SCSA* [8]. Il delay e l'area occupata, estratti da [15], sono, $t_{adder} + t_{mux}$ e $A_{adder} + A_{carry}$.

3.5. *GCSA*: l'architettura è simile a quella dei sommatori *CSA*, ma questa struttura permette il controllo dell'errore relativo massimo [8]. Il delay è pari a $O(log(k))$ e l'area $O(nlog(k))$, da [15].

4. *Approximate full adders*: fanno parte di questa famiglia i sommatori *LOA*, *Lower-part-OR Adder*. L'architettura prevede la sostituzione dei classici *Full-Adder* con *OR-gate* nella parte meno significativa (*LSB*) del sommatore. In Figura 3.2.2 i dettagli dell'architettura. Il delay e l'area sono $O(log(n-l))$ e $A_{LOA} + (l \cdot A_{OR})$, con A_{LOA} pari alla (3.5).

$$A_{LOA} = O((n-l)log(n-l)) \quad 3.5$$

5. *Truncated Adders*: l'architettura dei sommatori *truncated*, Figura 3.1.1, prevede il troncamento di *k LSB* di un sommatore *RCA*.

I risultati ottenuti da [8] sono riferiti a sommatori *16-bit*, sintetizzati usando la tecnologia STM CMOS *28nm*. Per ogni tipologia di sommatore approssimato è stata implementata una funzione *MATLAB* che effettua la somma *fixed-point* generando una distribuzione random di 10 milioni di campioni uniformemente distribuiti.

La Tabella 3.2.1 riassume i valori di area, potenza, ritardo, *error rate*, *MRED* e *PDP (product delay power)* dei vari tipi di sommatori approssimati nell'articolo [8]. Inoltre, tenendo presente che uno degli obiettivi principali dell'elaborato è ridurre il consumo di area, in questa fase di studio i valori sono stati ordinati avendo come parametro di riferimento l'area e il primo elemento della tabella sarà l'elemento con area minore.

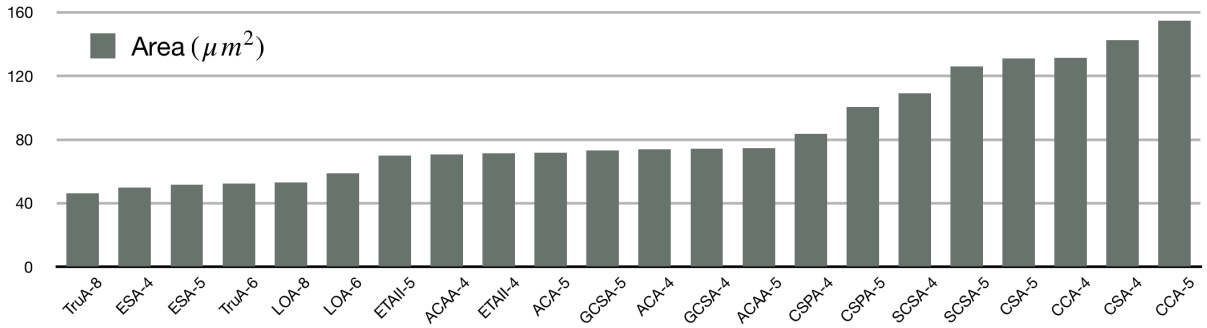
Tabella 3.2.1: Simulation Results for the Approximate Adders

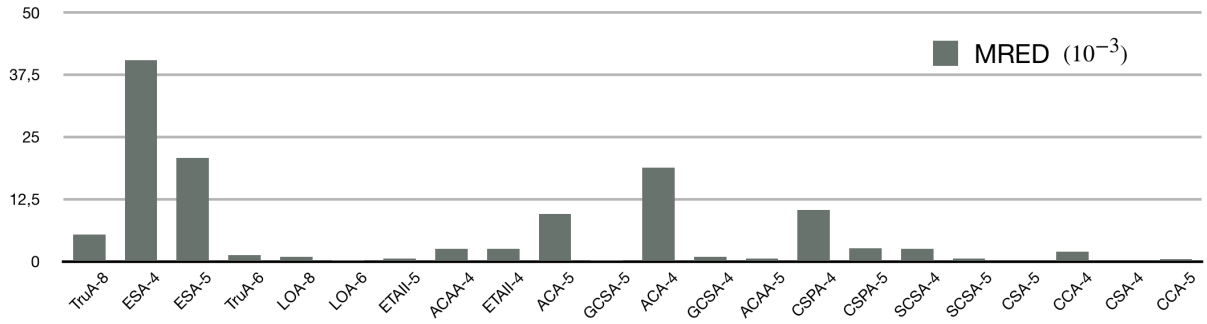
Adder type	**Area** (μm^2)	**Power** (μW)	**Delay** (ps)	**Error rate** (%)	**MRED** (10^{-3})	**PDP** (fJ)
TruA-8	46,2	64,2	350	100	5,4	22,5
ESA-4	49,9	47	260	85,07	40,4	12,2
ESA-5	51,7	50,6	310	80,3	20,8	15,7
TruA-6	52,4	67,9	390	99,98	1,3	26,5
LOA-8	53,2	66,9	390	89,99	1	26,1
LOA-6	58,8	75,1	440	82,19	0,25	33,0
ETAII-5	70,2	78,5	670	2,28	0,65	52,6
ACAA-4	70,8	80,9	550	5,85	2,60	44,5
ETAII-4	71,6	80,6	550	5,85	2,60	44,3

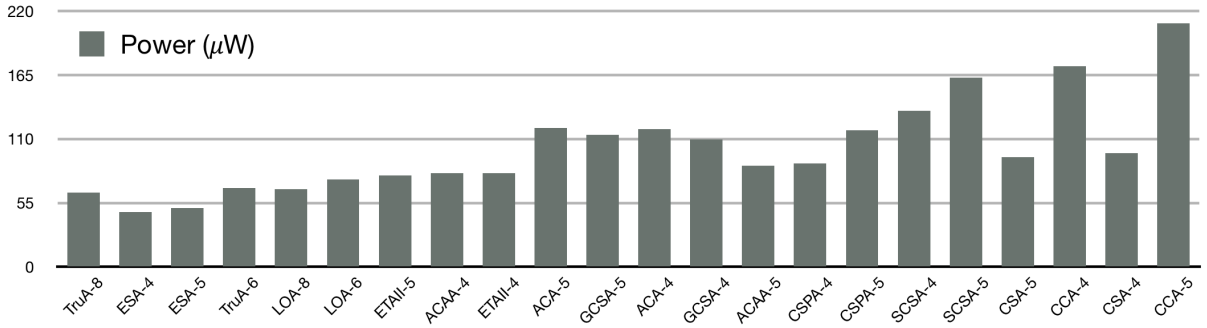
Adder type	Area (μm^2)	Power (μW)	Delay (ps)	Error rate (%)	MRED (10^{-3})	PDP (fJ)
ACA-5	71,8	119,4	270	7,76	9,60	32,2
GCSA-5	73,3	113,6	460	1,52	0,25	52,3
ACA-4	73,8	118,4	250	16,66	18,90	29,6
GCSA-4	74,3	109,7	380	4,26	0,98	41,7
ACAA-5	74,6	87,3	650	2,29	0,65	56,8
CSPA-4	83,7	89,2	300	29,82	10,40	26,8
CSPA-5	100,7	117,6	370	11,31	2,70	43,5
SCSA-4	109,2	134,5	320	5,85	2,60	43,0
SCSA-5	126,2	163	400	2,28	0,65	65,2
CSA-5	131,2	94,3	420	0,02	0,01	39,6
CCA-4	131,4	172,6	320	8,71	2,00	55,2
CSA-4	142,5	97,8	390	0,18	0,15	38,1
CCA-5	155	209,5	420	3,78	0,49	88,0

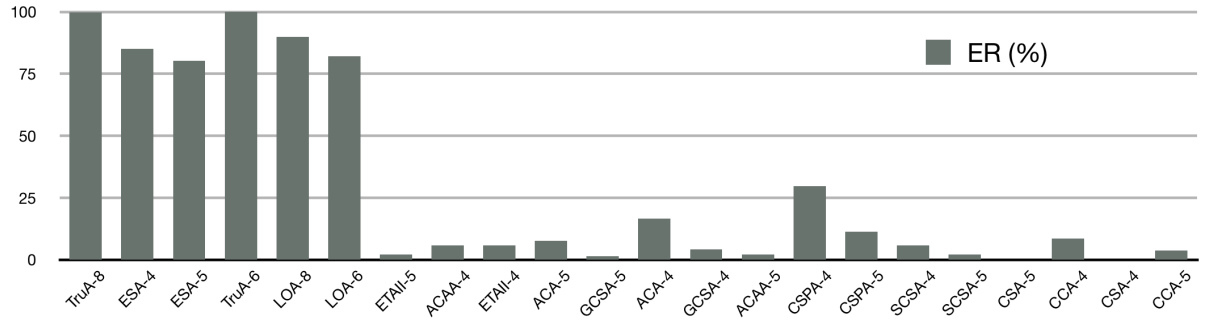
In questa fase di progetto, si pone l'attenzione sui valori di area e di *MRED*. Infatti, l'obiettivo principale è ridurre quanto più possibile l'area ed ottenere al contempo valori di *MRED* bassi. Da notare che un sommatore può avere valori molto bassi di *MRED* ma al contempo valori alti di *error rate*. Infatti, se da una parte l'*error rate* fa una stima della probabilità che si verifichi un errore, l'*MRED* dà informazioni sulla differenza tra valore atteso e valore ottenuto.

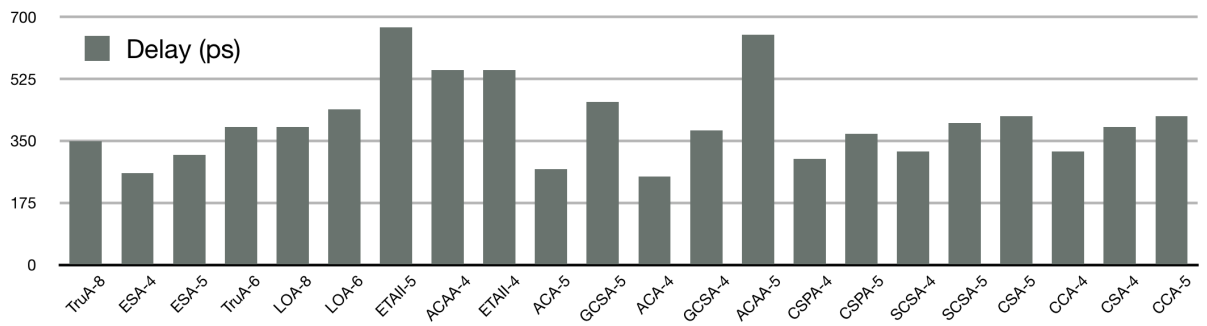
Per agevolare il confronto tra le varie tecniche di approssimazione, dalla Tabella 3.2.1 sono stati ricavati i grafici a seguire.

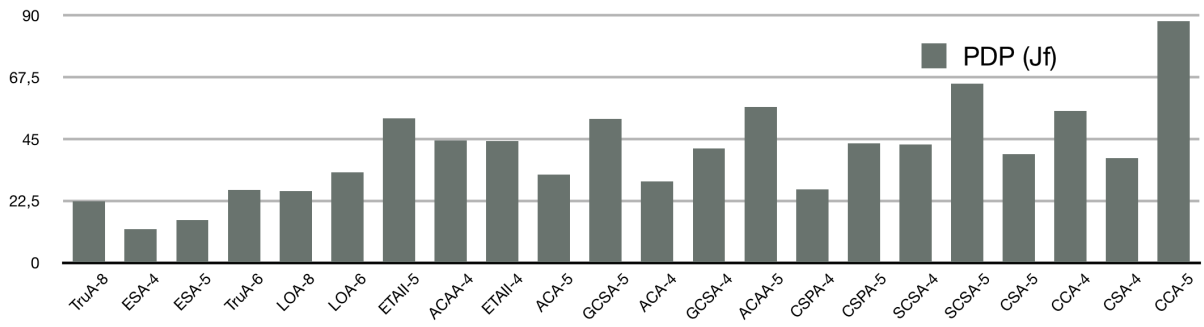



50
37,5
25
12,5
0
MRED (10^{-3})
TruA-8
ESA-4
ESA-5
TruA-6
LOA-8
LOA-6
ETAII-5
ACAA-4
ETAII-4
ACA-5
GCSA-5
ACA-4
GCSA-4
ACAA-5
CSPA-4
CSPA-5
SCSA-4
SCSA-5
CSA-5
CCA-4
CSA-4
CCA-5


220
165
110
55
0
Power (μW)
TruA-8
ESA-4
ESA-5
TruA-6
LOA-8
LOA-6
ETAII-5
ACAA-4
ETAII-4
ACA-5
GCSA-5
ACA-4
GCSA-4
ACAA-5
CSPA-4
CSPA-5
SCSA-4
SCSA-5
CSA-5
CCA-4
CSA-4
CCA-5


100
75
50
25
0
ER (%)
TruA-8
ESA-4
ESA-5
TruA-6
LOA-8
LOA-6
ETAII-5
ACAA-4
ETAII-4
ACA-5
GCSA-5
ACA-4
GCSA-4
ACAA-5
CSPA-4
CSPA-5
SCSA-4
SCSA-5
CSA-5
CCA-4
CSA-4
CCA-5


700
525
350
175
0
Delay (ps)
TruA-8
ESA-4
ESA-5
TruA-6
LOA-8
LOA-6
ETAII-5
ACAA-4
ETAII-4
ACA-5
GCSA-5
ACA-4
GCSA-4
ACAA-5
CSPA-4
CSPA-5
SCSA-4
SCSA-5
CSA-5
CCA-4
CSA-4
CCA-5



Tra i sommatori più efficienti in termini di area e potenza vi sono il *TrunA-k* (*Truncated*) e il *LOA-k* (*lower-part or adder*). Entrambi, anche se non tra i più veloci in assoluto, hanno un ritardo paragonabile ai più performanti. Per quanto riguarda il *LOA-k*, si nota che nonostante l'*error rate* sia molto alto, presenta un *MRED* molto basso. Differente invece il caso del *TruA-k*, che all'aumentare di *k*, cioè dei bit troncati, si hanno valori di *MRED* molto alti. Da notare che con $k = 5$ il parametro *MRED* è molto vicino al sommatore *LOA-8*.

Altri sommatori che permettono di ridurre area e potenza sono i sommatori *ESA-k*. Quest'ultimi però hanno valori elevati di *ER* ed *MRED*.

Il sommatore *GCSA-k*, *Generate signals for carry speculative adder*, a fronte di un amento considerevole di area e potenza, rispetto a sommatori *LOA-k* o *TrunA-k*, presenta valori molto bassi di *error rate* e *MRED*. In fase di progetto, si decide di scartare tale tipologia, in quanto il consumo di area e potenza è molto elevato, praticamente confrontabile con uno normale *Ripple-Carry adder*.

La scelta finale sui metodi di approssimazione dei sommatori in *fixed-point* da applicare è ricaduta sui sommatori *TrunA-k* e *LOA-k*, in quanto sono gli unici ad avere risparmi considerevoli di area e potenza mantenendo basso il parametro *MRED*. La tecnica *TrunA-k* viene implementata troncando gli ultimi *k-bit* di un *Ripple-Carry adder*, o *RCA*, come mostrato in Figura 3.2.1.

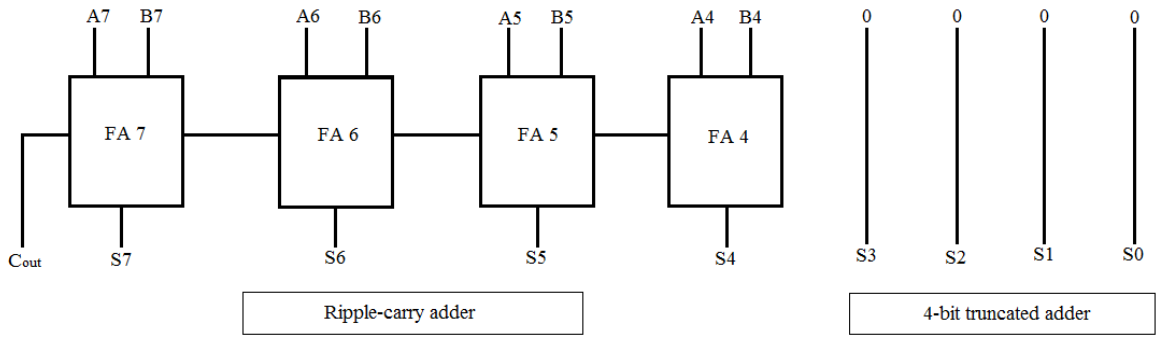


Figura 3.2.1: Architettura di un sommatore TrunA-4.

L'architettura del sommatore *LOA-k* (in Figura 3.2.2, estratta da [8]), viene implementata sostituendo ad ogni *full-adder* una *OR-gate*. Si noti che il valore del *carry-in* del *RCA* viene fornito dal sommatore *LOA*. Questo tipo di architettura, rispetto alla *Truncated*, a fronte di un aumento di area, è più performante in termini di *MRED.*

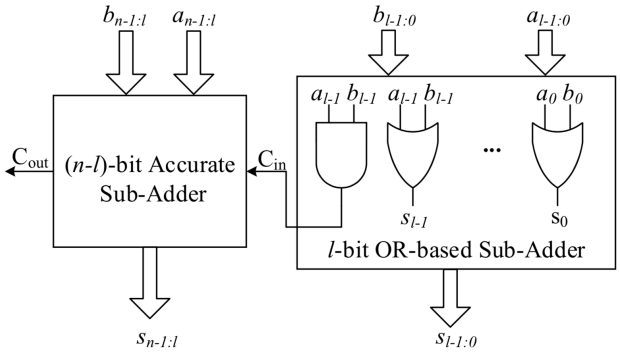


Figura 3.2.2: Architettura di un sommatore LOA-l

In conclusione, queste due tecniche agiscono sui bit meno significativi di un sommatore accurato, come il *Ripple-Carry adder.* Esse garantiscono risparmi significativi di area, potenza dissipata e ritardo, con tassi di *MRED* e *MED* non troppo elevati. Di contro, la percentuale di errore, *ER*, è quasi pari al 100% per la tecnica *Truncated*, implicando l'inesattezza del risultato nella totalità dei casi, mentre, è di circa l'80% per la tecnica *LOA-k*.

3.3 Stato dell'arte sommatori Floating-Point approssimati

Nell'articolo [21] viene presentata l'architettura di un sommatore *floating-point* inesatto, Figura 3.3.1.

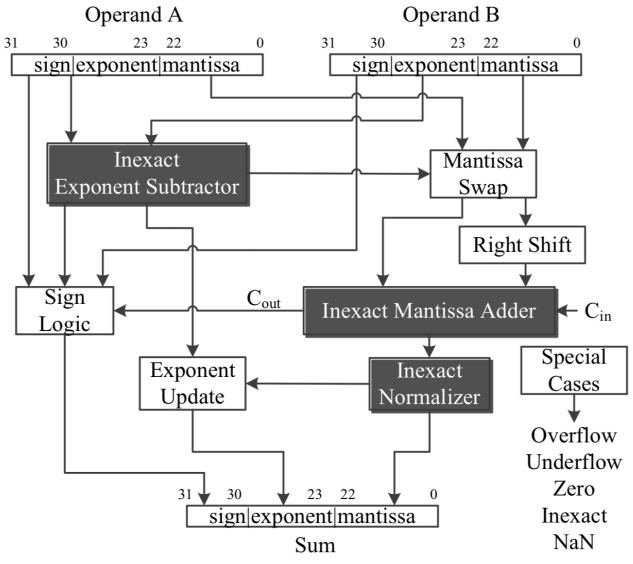


Figura 3.3.1: Architettura di un sommatore floating-point approssimato

La tecnica di *approximate computing*, *LOA-k*, viene impiegata nel sommatore delle mantisse e nel sottrattore degli esponenti [21]. Nell'articolo in questione, vengono presentati due tipi differenti di implementazioni: la prima sostituisce il sommatore esatto di mantisse con un sommatore *LOA-k* (con *K* numero di bit approssimati) che nell'esempio sono tutti i bit della mantissa, cioè 23, lasciando il sottrattore di esponenti invariato, ovvero senza approssimazioni. La seconda implementazione utilizza un sottrattore *LOA-1* per gli esponenti e lascia invariato il sommatore di mantisse. Il *rounding* in entrambe le implementazioni non è contemplato.

La scelta dell'omissione del processo di *rounding* verrà esaminata nelle fasi conclusive dell'elaborato, con un'attenta analisi sui benefici apportati da tale blocco in un sommatore approssimato. Si vedrà in seguito che il *rounding* non aumenta la precisione del risultato finale nel calcolo approssimato di somme/sottrazioni.

Di seguito i risultati ottenuti dalla sintesi sintetizzando su *Synopsys Design Compiler* con tecnologia *Faraday 65 nm*, *Tabella 3.3.1 e Tabella 3.3.2*:

Tabella 3.3.1: Comparison of exact and first inexact floating-point adders

FP Adders	Power (mW)	Area (μm^2)	Delay (ns)	PDP(pJ)
Exact FP Adder	0,3169	8131,20	3,20	1,01
Inexact FP Adder I	0,2219	5679,36	2,74	0,61
Improvement	29,98%	30,15%	14,38%	39,60%

Tabella 3.3.2: Comparison of exact and second inexact floating-point adders

FP Adders	Power (mW)	Area (μm^2)	Delay (ns)	PDP(pJ)
Exact FP Adder	0,3169	8131,20	3,20	1,01
Inexact FP Adder II	0,2555	6422,72	3,15	0,80
Improvement	19,38%	21,01%	1,56%	20,79%

L'articolo in questione [21] pone l'attenzione sull'importanza del calcolo dell'esponente in termini di precisione, infatti quest'ultimo è dominante e determina il range dinamico. Una piccola approssimazione può causare un errore di calcolo, nel risultato finale, molto elevato.

Si sottolinea, inoltre, l'importanza in termini di area e potenza dell'operazione di *rounding*. Infatti, la seconda implementazione, nonostante usi un sommatore di mantisse senza

approssimazioni, presenta risparmi considerevoli in termini di area e potenza di circa il 20%. Ciò non può essere giustificato dall'approssimazione fatta nel sottrattore, visto l'uso della tecnica *LOA* solo al primo *LSB* del sottrattore, con un risparmio di al massimo 4 *gate*, rispetto ad una configurazione senza errore. Quindi, risulta chiaro come l'eliminazione del *rounding* dia dei benefici in termini di area e potenza, mentre è quasi trascurabile in termini di ritardo. Il *delay*, dipendendo dal *critical path* del sommatore di mantissa, può essere migliorato solo ottimizzando quest'ultimo.

Nell'articolo [22] si presentano altri due tipi di implementazioni di sommatori *floating-point* approssimati: la prima è una tecnica *CAD*, che prende il nome di *Gate-Level Pruning*, mentre la seconda usa il sommatore di tipo *speculative*, ma approssimato. Nella Tabella 3.3.3 vengono esposti i risultati ottenuti di potenza dissipata, area occupata e del prodotto area-potenza. Inoltre sono stati calcolati i valori percentuali di risparmi di area e di potenza dissipata.

Tabella 3.3.3: Power, area and power-area product of the FPUs

FPU	Power (mW)	Area $(\mu m^2)(\%)$	Power-area product $(W \cdot \mu m^2)$
Exact FP Adder	2,81	13200	37,1
Pruned	2,4 (15%)	11850 (11%)	28,4 (23,45%)
Speculative	2,48 (12%)	10070 (14%)	25 (36,61%)
Mixed	2,07 (27%)	8550 (36%)	17,7 (53%)

I risultati ottenuti mostrano che si possono ottenere risparmi di area e potenza rispettivamente del 27% e del 36% mixando le due tecniche usate nell'articolo. Da notare che l'uso della sola tecnica *speculative* al sommatore di mantisse produce un risparmio del 14% in area e del 12% in potenza.

Tabella 3.3.4: Error characteristics of approximate floating-point

Adder/Subtractor	RE_{max}	RE_{RMS}
Pruned	1	$1{,}15E^{-3}$
Speculative	$5{,}69E^{-3}$	$2{,}36E^{-6}$
Mixed	1	$2{,}27E^{-4}$

La caratterizzazione dell'errore viene mostrata nella Tabella 3.3.4. I parametri considerati sono il *relative error* massimo, equivalente al *RED* discusso nel Paragrafo 3.1, e il *relative*

error RMS, parametro proporzionale al *SNR* (*signal noise ratio*), usato principalmente in applicazioni di *multimedia processing*.

In conclusione, si possono ottenere miglioramenti in termini di area e potenza dissipata di un sommatore *floating-point*, introducendo però una certa quantità di errore, applicando tecniche di *approximate computing*. Tutto dipende dal tipo di applicazione, per esempio, si prestano molto bene ad ospitare una *floating-point unit* approssimate, applicazioni di *image processing*, in cui non è richiesta grande precisione.

4. Implementazione VHDL dei sommatori/sottrattori Floating-Point

In questa fase di progetto si procede all'implementazione hardware in *VHDL* dei sommatori/ sottrattori *floating-point*. In primo luogo, si sviluppa un sommatore/sottrattore *floating-point* esatto, quindi senza approssimazioni, e solo successivamente si passerà all'implementazione dei sommatori/sottrattori *floating-point* approssimati. Questa metodologia di *design* è stata scelta perché risulta tecnicamente più precisa durante la valutazione dell'errore. Infatti, avendo come obiettivo la riduzione dell'errore relativo, derivante da ambienti di sviluppo differenti, la decisione di usare un unico ambiente di sviluppo per il calcolo dell'errore risulta determinante. Le motivazioni sono da ricercarsi nelle differenze tra un'unità *floating-point* e l'altra dal punto di vista del *rounding*, oltre alle differenze che potrebbero sussistere tra gli algoritmi di conversione da un numero *floating-point* in rappresentazione binaria, a quello in rappresentazione decimale.

La prima fase di sviluppo è focalizzata sullo studio degli algoritmi, puntando l'attenzione sulle differenze tra l'algoritmo di somma e quello di sottrazione. Come visto in precedenza, i due algoritmi sono simili e differiscono l'uno dall'altro, per gli step riguardanti il complemento a due e la normalizzazione.

La stesura del codice è stata pensata per ottenere un'architettura modulare gerarchica, così da semplificare la fase di test e, quindi, per la correzione di eventuali errori, oltre che alla lettura dei report post-sintesi su *Synopsys Design Compiler*. L'unità *floating-point* esatta è stata concepita per eseguire il calcolo in 5 *clock cycle*, che rispecchiano i vari step dell'algoritmo. Inoltre, con lo scopo di ridurre la complessità di progettazione dei sommatori approssimati, la *entity* che gestisce la somma e la normalizzazione del risultato è stata divisa in due sotto-moduli. In questo modo, la progettazione dei sommatori viene semplificata notevolmente, perché basterà modificare una sola *entity*, per ottenere un componente che applica la somma approssimata.

Ogni unità *floating-point* è provvista di una *finite state machine*, o *FSM*, che gestisce i moduli implementati. L'analisi in termini di precisione, cioè della caratterizzazione degli errori, di sommatori/sottrattori approssimati con *rounding* rispetto ad altri in cui il *rounding* non è contemplato, avviene implementando due tipi diversi di unità *floating-point*. La differenza consiste nella presenza, o meno, dello stato di *rounding* nella *FSM*, e del componente hardware dedicato. Da notare che l'assenza del *rounding* ridurrà l'esecuzione del calcolo di un *clock cycle.*

Un'unità *floating-point* approssimata differisce da un'unità esatta principalmente per il sommatore di mantisse. Le tecniche di *approximate computing* usate sono di tre tipi:

- *LOA-k*;
- *TrunA-k*;
- *RcHA-k*.

Queste tre tecniche verranno *mixate* tra loro, così da verificare la bontà di un sommatore approssimato *ibrido*.

La scelta di *design* effettuata per lo sviluppo del codice *VHDL* è stata di tipo comportamentale, o *behavioral*, per la parte riguardante l'esecuzione dell'algoritmo, mentre di tipo *RTL* per l'implementazione dei sommatori approssimati.

Nei paragrafi 4.1 e 4.2, si presentano tutte le architetture, fornendo dettagli riguardanti il funzionamento e la gerarchia sia del sommatore/sottrattore di riferimento, cioè quello esatto, che dei sommatori/sottrattori approssimati. La tecnica *RcHA-k*, o *Ripple-Carry Half-Adder*, si approfondisce nel Paragrafo 4.3, in quanto concepita durante il lavoro di tesi.

4.1 Sommatore/Sottrattore Floating-Point esatto

Il sommatore/sottrattore *floating-point* esatto viene implementato seguendo i vari step dell'algoritmo di somma e sottrazione. In Figura 4.1.1 viene mostrata l'architettura del sommatore/sottrattore implementato. Le due mantisse, inizialmente di 23 bit, durante l'esecuzione dell'algoritmo, aumenteranno il loro numero di bit, fino ad arrivare a 29. Infatti, è necessario aggiungere 1 bit per effettuare la somma dei segni, 2 bit per sommare gli *hidden* bit, tenendo conto della possibilità di *overflow*, 23 bit di mantissa, ed infine 3 bit aggiuntivi per effettuare il *rounding*. Quindi il sommatore *fixed-point* di mantisse, sarà di 29 bit.

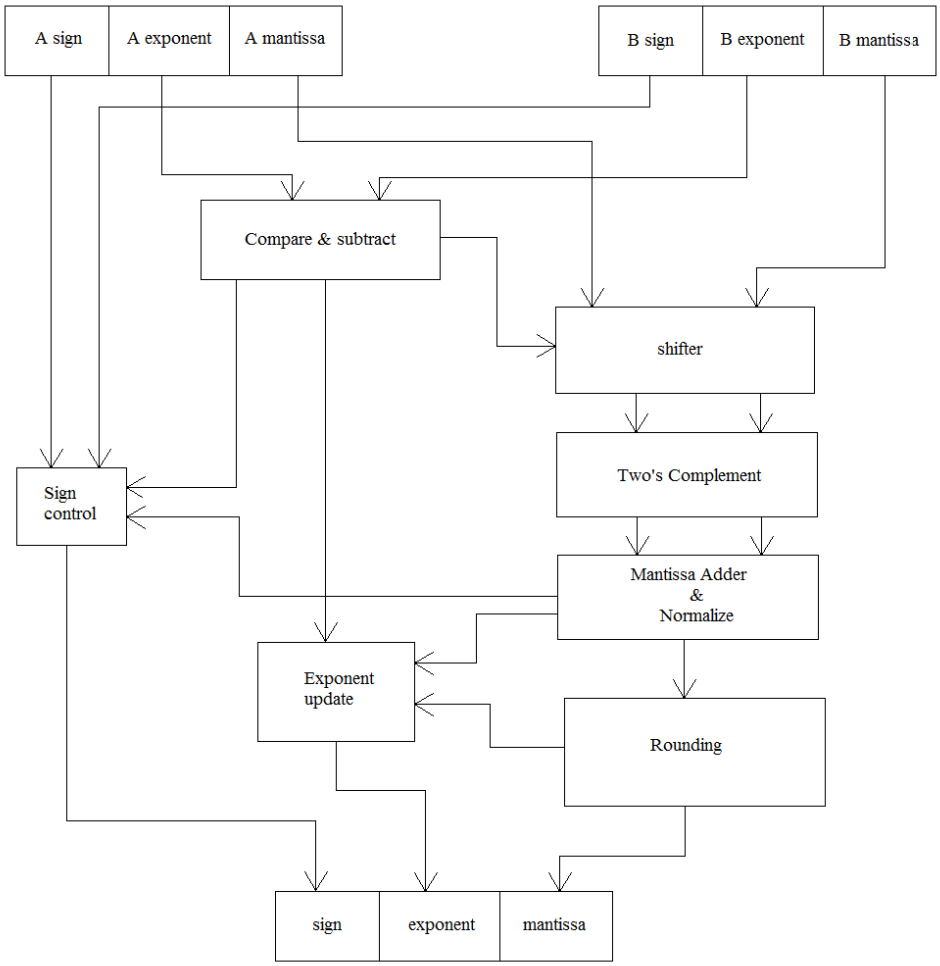


Figura 4.1.1: Architettura sommatore/sottrattore floating-point esatto

L'algoritmo di calcolo della somma algebrica *floating-point* è composto da 5 step principali, nominati come segue:

1. *Compare & subtract*: in questo modulo si esegue il confronto tra gli esponenti, si sottrae l'esponente con valore minore a quello con valore maggiore e il risultato viene inviato al modulo successivo, chiamato *shifter*;
2. *Shifter*: il blocco ha come *input* le due mantisse e il numero di *shift* da effettuare. All'interno vi è un segnale di controllo che decide quale delle due mantisse è quella a cui verranno applicati gli *shift*. Gli *output* di questo modulo saranno le due mantisse, di cui una delle due opportunamente modificata.
3. *Two's complement*: nel caso in cui i segni dei due numeri da sommare siano discordi, entra in gioco il seguente modulo. Infatti, per numeri con segno opposto, quindi nel caso in cui si voglia effettuare una sottrazione, bisogna applicare il complemento a due del numero con segno negativo. Questo blocco, nel caso in cui i numeri siano concordi, restituisce come *output* le due mantisse senza apportare alcuna modifica.
4. *Mantissa adder & normalize*: la somma viene eseguita in questo blocco funzionale. Inoltre, in caso di *overflow* o di sottrazione, si provvede alla normalizzazione del risultato. In questa fase viene aggiornato l'esponente.
5. *Rounding*: l'algoritmo si conclude con l'esecuzione del modulo di *rounding*. Il metodo di *rounding* usato è il *rounding to the nearest even*, che tra tutti è quello che garantisce maggiore precisione.

I blocchi di *Sign control* e *Exponent update,* di Figura 4.1.1, non rappresentano dei veri e propri moduli, ma possono essere visti come delle istruzioni all'interno dei moduli *two's complement*, *mantissa adder & normalize* e *rouding*, che gestiscono il valore del segno e dell'esponente.

Tutti i moduli vengono gestiti da una *FSM* di *Moore*. In Figura 4.1.2 si ha il diagramma degli stati.

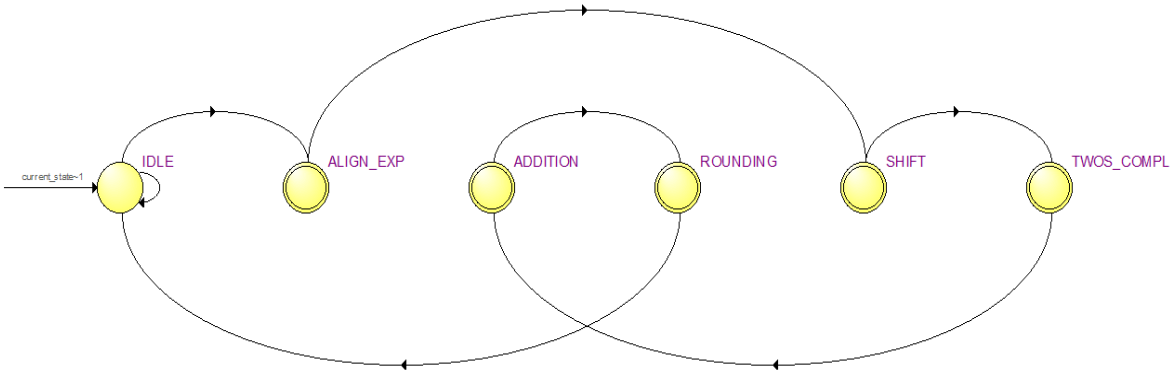


Figura 4.1.2: Diagramma degli stati della *FSM* di gestione del sommatore/sottrattore *floating-point* esatto.

Lo stato iniziale, chiamato *IDLE*, dipende dall'unico ingresso della *FSM*, nominato *start.* Fintanto che il segnale di *start* è al valore logico '0', ad ogni *clock cycle,* la *FSM* rimane sullo stato di *IDLE.* Il passaggio allo stato successivo, che nello specifico corrisponde allo stato *ALIGN_EXP*, si ha non appena il segnale *start* passa allo stato logico '1' nel fronte di salita del *clock.* La macchina stati, così, va di volta in volta allo stato successivo, ogni *clock cycle*, sul fronte di salita del *clock*, modificando i suoi *output*, attivando i processi di ogni blocco funzionale. Lo stato finale, *rounding*, fa tornare la *FSM* al suo stato d'origine, ovvero l'*IDLE* e rimane in questo stato fintanto che il segnale *start* è allo stato logico '0'. I segnali di uscita della *FSM* sono di controllo per i vari moduli, infatti, essi sono collegati ai blocchi funzionali precedentemente descritti. Ogni modulo è così provvisto di un *process,* in cui nella *sensitivity list* sono inseriti i segnali di controllo a cui si è appena fatto riferimento.

La struttura gerarchica è composta da 3 livelli. Nel primo livello, o *Top Level Entity*, si istanziano le entity e si effettuano i collegamenti di tutti i blocchi funzionali quali *Compare & subtract*, *Shifter*, *Two's complement*, *Mantissa adder & normalize* e *Rounding.*

Il livello successivo è composto dalle architetture dei moduli che costituiscono il sommatore/sottrattore *floating-point*. In questo caso è costituito da 6 file *VHDL.*

Nel livello 3, si inserisce il sommatore di mantisse. Nel caso del sommatore algebrico *floating-point* esatto, il sommatore *fixed-point* di mantisse usato è quello della libreria *IEEE*, in grado di applicare la somma a due segnali *std_logic_vector.*

La Tabella 4.1.1 riassume quanto detto.

Tabella 4.1.1: Gerarchia sommatore/sottrattore floating-point esatto

Gerarchia	Entity
Livello 1	fp_unit
Livello 2	FSM_fp_unit, compare_and_sub, shift, twoS_complement, adder_external, rounding
Livello 3	adder_internal

La *Top Level Entity* finale dell'unità *floating-point,* in grado di calcolare somme o differenze, è rappresentata nella Figura 4.1.3.

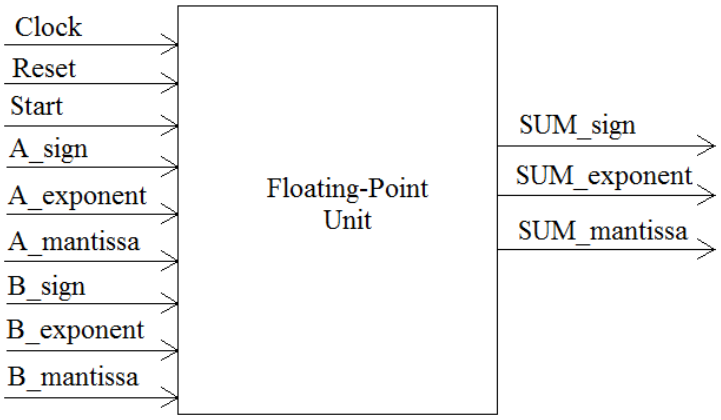


Figura 4.1.3: Top Level Entity dell'unità floating-point

4.1.1 Simulazione della Floating-Point unit

A seguito dell'implementazione *VHDL* dell'unità *floating-point* si procede con la fase di test in *ModelSim*, per verificare il corretto funzionamento dell'architettura. Le simulazioni sono state effettuate per ogni componente implementato, verificando anche i casi particolari oltre che i numeri denormalizzati. Di seguito, Figura 4.1.4 e Figura 4.1.5, due immagini estratte da *ModelSim* durante la fase di test della *Top Level Entity*. Come descritto in precedenza, il calcolo inizia non appena il segnale di *start*, chiamato *start_tb*, passa allo stato logico '1', anche solo per un fronte di salita del *clock*. Successivamente, per i prossimi 5 *clock cycle*, la *FSM* passerà tra i vari stati, elaborando il dato finale, che sarà visibile in uscita al quinto ciclo di *clock.* Unificando i tre segnali di uscita, chiamati rispettivamente *SUM_sign_tb*, *SUM_exp_upd_ro_tb* e *SUM_mantissa_ro_tb* si otterrà il risultato della somma algebrica *floating-point* in rappresentazione binaria, su 32 bit.

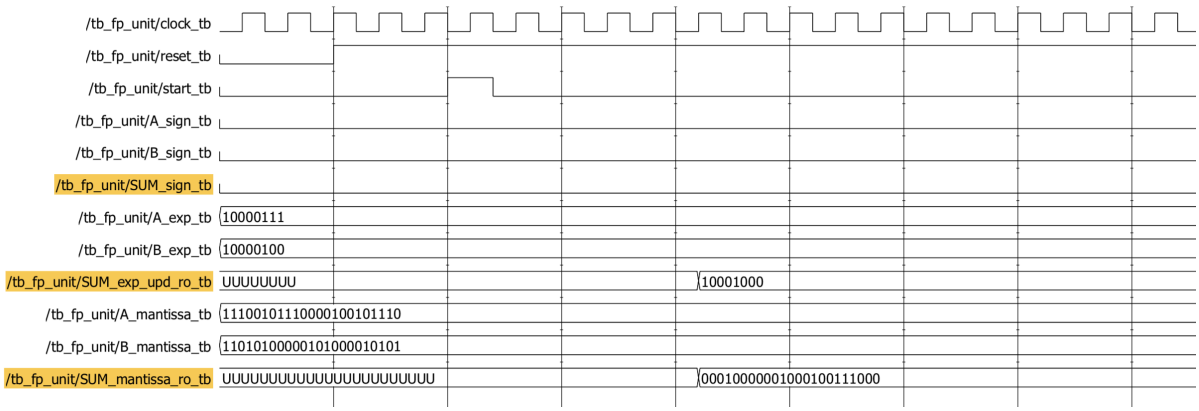


Figura 4.1.4: Simulazione del sommatore/sottrattore *floating-point* esatto su ModelSim. I numeri sommati sono gli stessi usati in fase di spiegazione dell'algoritmo di somma Paragrafo 2.1.2.

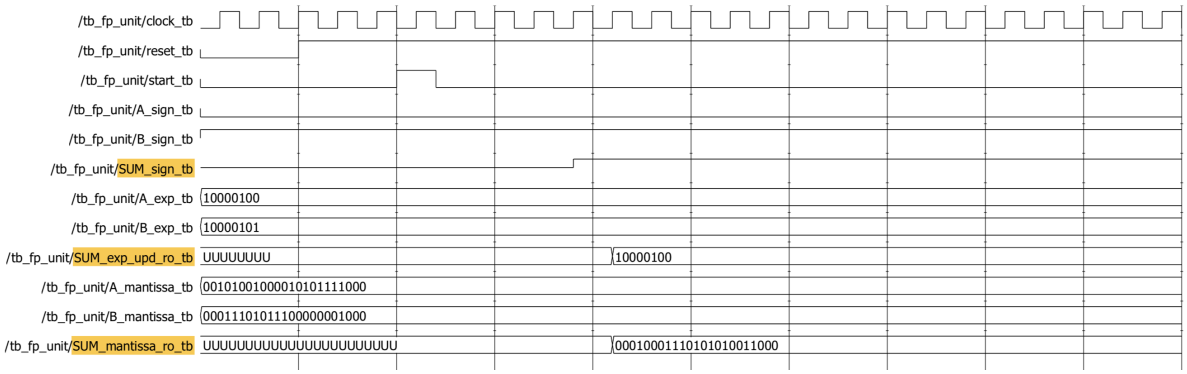


Figura 4.1.5: Simulazione del sommatore/sottrattore *floating-point* esatto su ModelSim. I numeri sommati sono gli stessi usati in fase di spiegazione dell'algoritmo di sottrazione del Paragrafo 2.1.3.

4.2 Implementazione sommatori/sottrattori Floating-Point approssimati

Nel Paragrafo in questione, si passa all'implementazione di vari tipi di unità *floating-point* approssimate usando tre principali tecniche di approssimazione: *LOA-k*, *TrunA-k* e *RcHA-k*.

Si implementano due grandi famiglie di unità *floating-point*: unità approssimate che includono il processo di *rounding* ed unità in cui il *rounding* viene omesso. In questo modo, sarà possibile approfondire l'utilità del *rounding* in un'unità *floating-point* approssimata.

La metodologia di *design* applicata consiste nel modificare il terzo livello della gerarchia, cioè *adder_internal* del Paragrafo 4.1, sostituendo la somma effettuata dalla libreria *IEEE* con un sommatore *fixed-point* approssimato appositamente progettato.

Oltre ai sommatori che usano una singola tecnica di *approximate computing,* si implementano anche sommatori *ibridi*, cioè sommatori in cui all'interno sono presenti due o più tecniche di approssimazione.

Di seguito la lista delle dieci unità *floating-point* approssimate, ognuna delle quali avrà un sommatore *23-bit fixed-point* approssimato:

- fp_12rca11loa;
- fp_12rca11trun;
- fp_12rca11rcha;
- fp_11rca12trun;
- fp_6rca5rcha12loa;
- fp_5rca5rcha5loa8trun;
- fp_5rca5loa13trun;
- fp_loafull;
- fp_13loa10trun;
- fp_8loa15trun.

Si intuisce dal nome delle unità che i numeri prima del nome del sommatore approssimato, indicano il quantitativo di bit approssimati: per esempio, l'unità *fp_12rca11loa* indica che la somma dei primi 12 bit di ingresso, cioè i più significativi, verranno ottenuti da un classico *Ripple-Carry adder* ed i successivi 11, con la tecnica di approssimazione *LOA-k*, con *k* il numero di bit a cui viene applicata l'approssimazione.

L'unità *fp_loafull* è stata implementata usando la tecnica *LOA-k* a tutta la somma della mantissa, in contrasto con le altre in cui l'approssimazione viene applicata solo ai bit meno significativi. L'obiettivo è replicare l'architettura dell'articolo [21] per valutare il risparmio di area e potenza e confrontarlo con le altre unità approssimate.

4.2.1 Gerarchie delle unità Floating-Point approssimate

Come spiegato in precedenza, si implementando due famiglie di unità *floating-point*, con e senza rounding. In Tabella 4.2.1.1, 4.2.1.2 e 4.2.1.3 vengono mostrate le gerarchie delle varie unità approssimate in cui è presente anche l'operazione di *rounding*. Nelle successive tre tabelle, cioè 4.2.1.4, 4.2.1.5 e 4.2.1.6, sono visibili le gerarchie delle unità *floating-point* approssimate ma sprovviste di *rounding*. In quest'ultime, oltre a mancare l'entity *rounding*, la macchina a stati finiti viene modificata appositamente, perdendo uno dei 6 suoi stati.

Tabella 4.2.1.1: Gerarchie unità floating-point approssimate con rounding

Gerarchia\Entity	fp_12rca11loa	fp_12rca11trun	fp_12rca11rcha
Livello 1	fp_unit	fp_unit	fp_unit
Livello 2	FSM_fp_unit, compare_and_sub, shift, twoS_complement, adder_external, rounding	FSM_fp_unit, compare_and_sub, shift, twoS_complement, adder_external, rounding	FSM_fp_unit, compare_and_sub, shift, twoS_complement, adder_external, rounding
Livello 3	add_RCA12_LOA11_tot	add_RCA12_trun11_tot	add_RCA12_rcha11_tot
Livello 4	add_RCA12_LOA11	add_RCA12_trun11	add_RCA12_rcha11
Livello 5	full_adder	full_adder	rcHA11
Livello 6	-	-	full_adder half_adder

Tabella 4.2.1.2: Gerarchie unità floating-point approssimate con rounding

Gerarchia\Entity	fp_11rca12trun	fp_6rca5rcha12loa	fp_5rca5rcha5loa8trun
Livello 1	fp_unit	fp_unit	fp_unit
Livello 2	FSM_fp_unit, compare_and_sub, shift, twoS_complement, adder_external, rounding	FSM_fp_unit, compare_and_sub, shift, twoS_complement, adder_external, rounding	FSM_fp_unit, compare_and_sub, shift, twoS_complement, adder_external, rounding
Livello 3	add_rca11Trun11_tot	add_rca6rcha5loa12_tot	add_rca5rcha5loa5trun8_tot
Livello 4	add_RCA11_Trun11	add_rca6_rcha5_loa12	add_rca5rcha5loa5trun8
Livello 5	full_adder	full_adder half_adder	rcHA5
Livello 6	-	-	full_adder half_adder

Tabella 4.2.1.3: Gerarchie unità floating-point approssimate con rounding

Gerarchia\Entity	fp_5rca5loa13trun	fp_loafull	fp_13loa10trun	fp_8loa15trun
Livello 1	fp_unit	fp_unit	fp_unit	fp_unit
Livello 2	FSM_fp_unit, compare_and_sub, shift, twoS_complement, adder_external, rounding	FSM_fp_unit, compare_and_sub, shift, twoS_complement, adder_external, rounding	FSM_fp_unit, compare_and_sub, shift, twoS_complement, adder_external, rounding	FSM_fp_unit, compare_and_sub, shift, twoS_complement, adder_external, rounding
Livello 3	add_5rca5loa13trun_tot	add_loafull_tot	add_13loa10trun_tot	add_8loa15trun_tot
Livello 4	add_5rca5loa13trun	add_loafull	add_13loa10trun	add_8loa15trun
Livello 5	full_adder	full_adder	full_adder	full_adder

Tabella 4.2.1.4: Gerarchie unità floating-point approssimate senza rounding

Gerarchia\Entity	fp_12rca11loa	fp_12rca11trun	fp_12rca11rcha
Livello 1	fp_unit	fp_unit	fp_unit
Livello 2	FSM_fp_unit, compare_and_sub, shift, twoS_complement, adder_external	FSM_fp_unit, compare_and_sub, shift, twoS_complement, adder_external	FSM_fp_unit, compare_and_sub, shift, twoS_complement, adder_external
Livello 3	add_RCA12_LOA11_tot	add_RCA12_trun11_tot	add_RCA12_rcha11_tot
Livello 4	add_RCA12_LOA11	add_RCA12_trun11	add_RCA12_rcha11
Livello 5	full_adder	full_adder	rcHA11
Livello 6	-	-	full_adder half_adder

Tabella 4.2.1.5: Gerarchie unità floating-point approssimate senza rounding

Gerarchia\Entity	fp_11rca12trun	fp_6rca5rcha12loa	fp_5rca5rcha5loa8trun
Livello 1	fp_unit	fp_unit	fp_unit
Livello 2	FSM_fp_unit, compare_and_sub, shift, twoS_complement, adder_external	FSM_fp_unit, compare_and_sub, shift, twoS_complement, adder_external,	FSM_fp_unit, compare_and_sub, shift, twoS_complement, adder_external,
Livello 3	add_rca11Trun11_tot	add_rca6rcha5loa12_tot	add_rca5rcha5loa5trun8 _tot
Livello 4	add_RCA11_Trun11	add_rca6_rcha5_loa12	add_rca5rcha5loa5trun8
Livello 5	full_adder	full_adder half_adder	rcHA5
Livello 6	-	-	full_adder half_adder

Tabella 4.2.1.6: Gerarchie unità floating-point approssimate senza rounding

Gerarchia\ Entity	fp_5rca5loa13trun	fp_loafull	fp_13loa10trun	fp_8loa15trun
Livello 1	fp_unit	fp_unit	fp_unit	fp_unit
Livello 2	FSM_fp_unit, compare_and_sub, shift, twoS_complemen, adder_external	FSM_fp_unit, compare_and_sub, shift, twoS_complemen, adder_external	FSM_fp_unit, compare_and_sub, shift, twoS_complement, adder_external	FSM_fp_unit, compare_and_sub, shift, twoS_complemen, adder_external
Livello 3	add_5rca5loa13trun_tot	add_loafull_tot	add_13loa10trun_tot	add_8loa15trun_tot
Livello 4	add_5rca5loa13trun	add_loafull	add_13loa10trun	add_8loa15trun
Livello 5	full_adder	full_adder	full_adder	full_adder

Anche in questa fase di progetto, tutte le unità *floating-point* vengono testate su *ModelSim* per valutare il corretto funzionamento, prima di effettuare la sintesi.

4.2.2 Architettura delle unità Floating-Point approssimate senza rounding

Ai fini del corretto funzionamento delle unità approssimate senza *rounding*, si effettuano delle modifiche architetturali al sommatore *floating-point* esatto: le nuove unità approssimate non disporranno più del blocco di *rounding*, dispendioso in termini di area e potenza, e poco utile alla precisione. Inoltre, l'unità di controllo, *FSM*, verrà modificata eliminando uno dei suoi 6 stati. In Figura 4.2.2.1 e 4.2.2.2 si illustra la nuova architettura e la nuova macchina stati dei sommatori approssimati senza *rounding*.

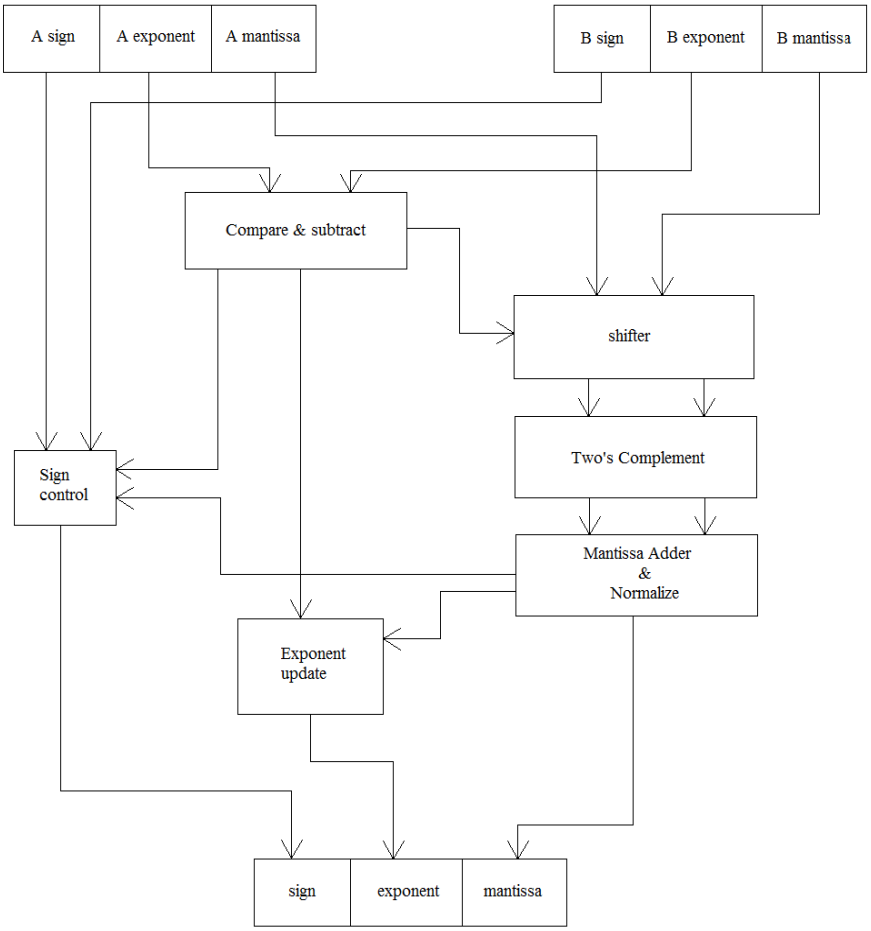


Figura 4.2.2.1: Architettura unità floating-point senza rounding

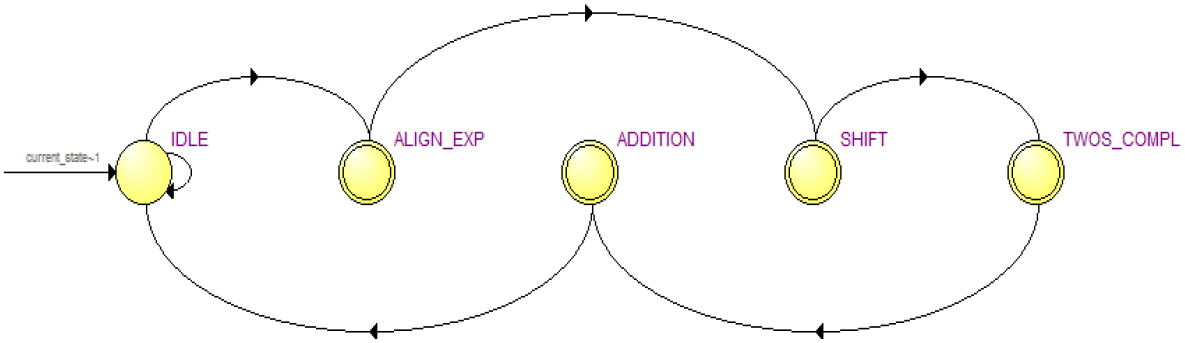


Figura 4.2.2.2: FSM dei sommatori/sottrattori senza rounding

Questi sommatori, saranno sicuramente più performanti in termini di area, potenza e *delay* rispetto a quelli provvisti di *rounding*. Le differenze in termini di precisione saranno discusse nei capitoli successivi, ma si anticipa che l'assenza del *rounding* non influenza i parametri di errore che verranno successivamente calcolati.

Prima di procedere con la sintesi su *Quartus*, si effettuano delle simulazioni su *ModelSim*, per verificare il corretto funzionamento. In Figura 4.2.2.3, il risultato di una simulazione svolta sul sommatore/sottrattore *floating-point* approssimato, senza *rounding*, di nome *fp_loafull*.

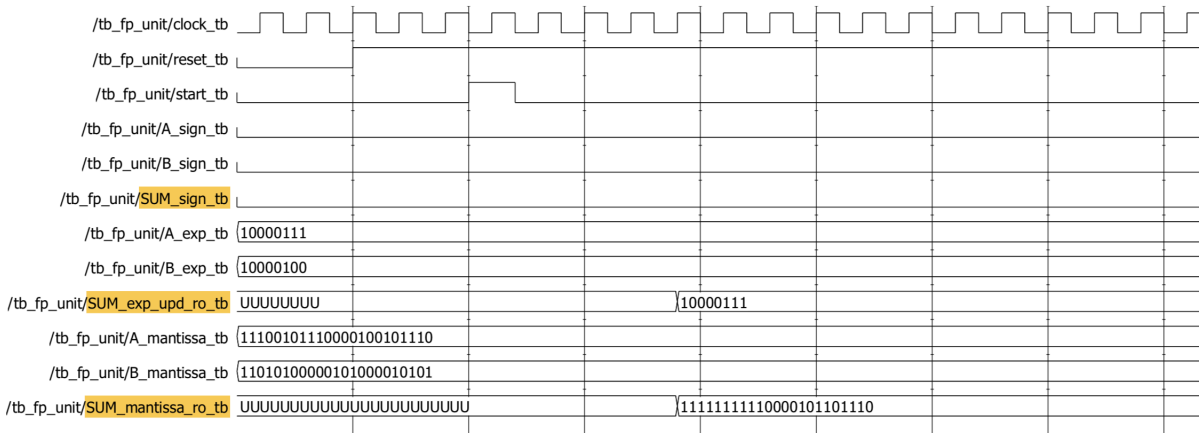


Figura 4.2.2.3: Simulazione *ModelSim* dell'unità *floating-point fp_loafull* senza *rounding*.

Come previsto, il sommatore approssimato senza *rounding*, dopo aver ricevuto il segnale di *start* a livello logico '1', fornirà il risultato finale in 4 *clock cycle*. Tutti i sommatori approssimati senza *rounding* avranno lo stesso comportamento del sommatore appena esaminato.

4.3 Sviluppo della tecnica Ripple-Carry Half Adder

Oltre alle tecniche di *approximate computing* esistenti in letteratura, durante l'attività di tesi in *STMicroelectronics* di Catania, si è deciso di cimentarsi nell'implementazione di una tecnica di approssimazione originale.

Prima di procedere con la trattazione del sommatore approssimato implementato, si definiscono i vari step che hanno portato al risultato finale:

1. Studio delle tabelle di verità di *Full Adder*, *Half Adder* e *Ripple-Carry Adder*;
2. Studio delle architetture;
3. Simulazione in *C* di più architetture;
4. Design hardware in *VHDL*;
5. Simulazione Hardware su *ModelSim*;
6. Applicazione al sommatore algebrico *floating-point* approssimato.

L'obiettivo iniziale è ridurre il numero di porte di logiche, e di conseguenza l'area, di un *Full Adder*. Si effettua un confronto tra la tabella di verità dell'*Half Adder* e quella del *Full Adder*, Tabella 4.3.1 e 4.3.2, e si nota che aggiungendo una porta logica *OR* tra due *Half Adder* si

ottengono risultati vicini a quelli di un *Ripple-Carry Adder* classico, formato da due *Full adder* consecutivi.

Tabella 4.3.1: tabella verità Half-Adder

A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Tabella 4.3.1: tabella verità Full-Adder

A_n	B_n	C_{n-1}	S_n	C_n
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Il risparmio di area è significativo: due *Full Adder* sono costituiti da 10 porte logiche, mentre, due *Half adder* e una OR-gate da 5 *gate*. Quindi, le porte logiche usate saranno esattamente la metà di un classico *RCA*. L'architettura del *RCA* due bit e del nuovo componente, che è stato rinominato *Ripple-Carry Half Adder* due bit, sono presenti in Figura 4.3.1 e 4.3.2.

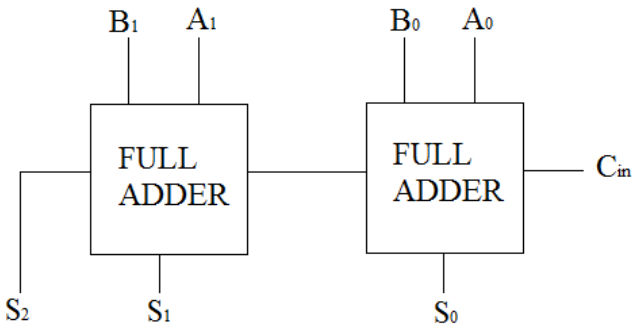


Figura 4.3.1: Architettura *Ripple-Carry Adder* a due bit, con numero totale di porte logiche è pari a 10.

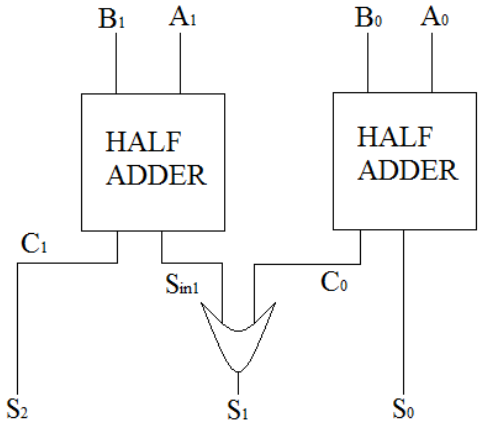


Figura 4.3.2: Architettura *Ripple-Carry Half adder* con 5 porte logiche totali.

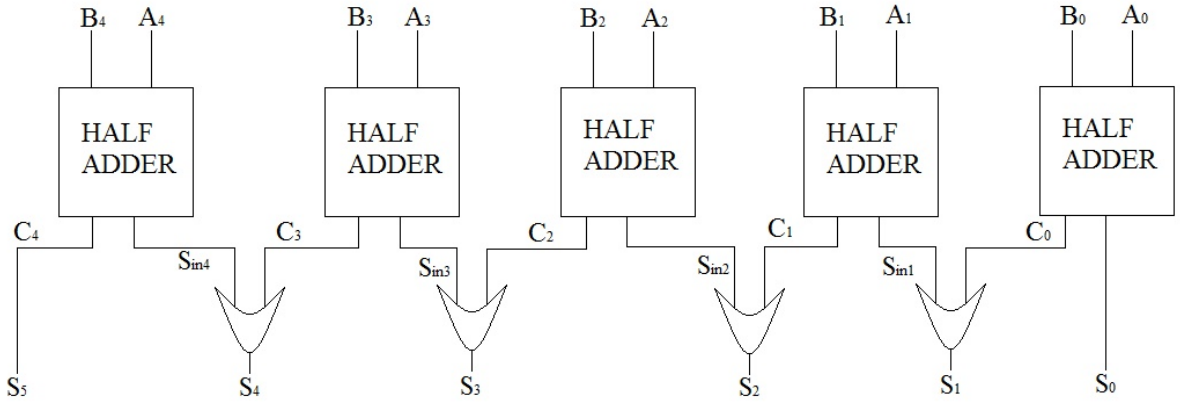


Figura 4.3.3: Architettura sommatore 5 bit *Ripple-Carry Half Adder*

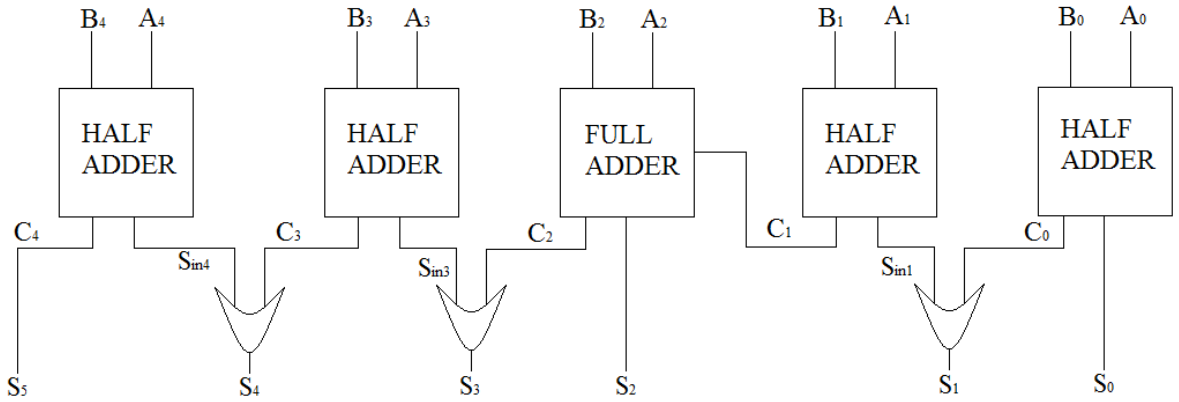


Figura 4.3.4: Architettura sommatore composto da 4 *Half Adder* e *1 Full Adder*

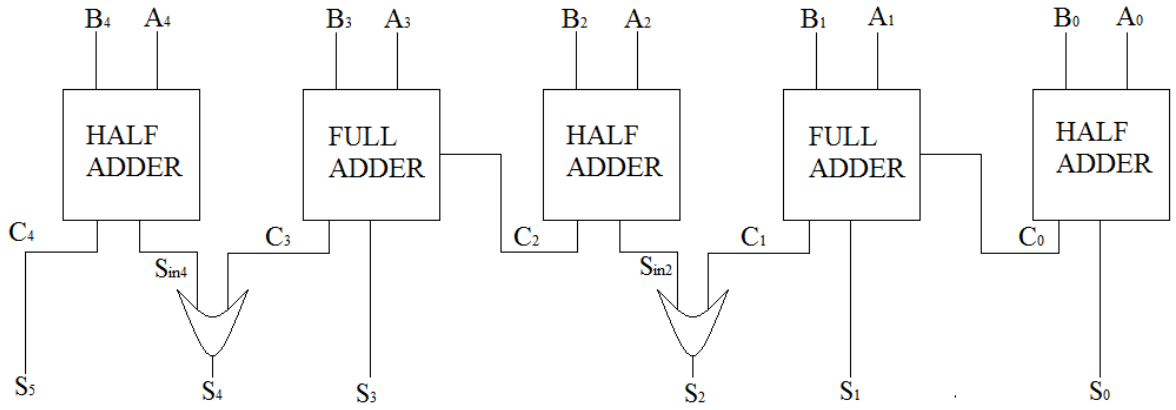


Figura 4.3.5: Architettura sommatore composto da 3 *Half Adder* e 2 *Full Adder*

Oltre alla sequenza di *Half Adder*, collegati come spiegato in precedenza, sono stati implementati altri due tipi di sommatori, che seguono lo stesso principio descritto fino ad ora. Per stabilire la migliore configurazione da usare, si procede con la simulazione di questi componenti hardware, descrivendo il loro comportamento in *C*, tramite l'uso delle operazioni di *bitwise*. Per semplicità si decide di implementare sommatori approssimati di 5 bit.

Le combinazioni progettate di sommatori *5-bit* sono:

- *Ripple-Carry Half Adder*, Figura 4.3.3;
- Sommatore composto da 4 *Half adder* ed 1 *Full adder*, Figura 4.3.4;
- Sommatore composto da 3 *Half adder* e 2 *Full adder*, Figura 4.3.5.

Prima di passare alla progettazione hardware, tutti e 3 i sommatori *5-bit*, vengono implementati in *C* e testati mettendo in ingresso tutte le possibili combinazioni. L'obiettivo è analizzare l'errore dovuto all'approssimazione fatta, calcolando i due parametri *MRED* e *MED* e solo dopo scegliere la configurazione più efficiente.

Tabella 4.3.1: Caratterizzazione dell'errore

5 Bit Adders	MED	MRED
RcHa	3,75	0,11341412
4 Half Adder & 1 Full Adder	3,75	0,112077541
3 Half Adder & 2 Full Adder	3,75	0,109123807

Dai risultati, estraibili dalla Tabella 4.3.1, si nota che le differenze in termini di errore relativo, *MRED*, sono abbastanza piccole. Confrontando, la prima implementazione, cioè *RcHa5*, con l'implementazione con 3 *Half Adder* e 2 *Full Adder*, si giunge alla conclusione che il parametro *MRED*, nonostante la presenza di ben 6 gate in più, non è così basso tale da essere preferito al sommatore *RcHa5*. Da qui la scelta di usare l'architettura *RcHa-k* in fase di progettazione di alcune unità *floating-point* approssimate.

Si passa così all'implementazione in *VHDL*, ed infine, prima di inserire il nuovo componente nell'unità *floating-point*, si usa il software di simulazione *ModelSim* per testarne il funzionamento.

5. Sintesi su fpga e simulazione hardware per lo studio dell'errore

Le validazioni delle unità progettate e, quindi, la simulazione hardware per testare il corretto funzionamento e per la caratterizzazione dell'errore, passa dalla sintesi su *FPGA*. Come spiegato nel Paragrafo 3.1, per la valutazione dei parametri *MRED* e *MED* è necessario avere un set molto ampio di campioni da testare. Per questo motivo, la valutazione dei due parametri avviene tramite la sintesi di ogni unità *floating-point* su un *FPGA* di *Altera*. Il sistema progettato sarà un *system-on chip*, formato da un processore *custom* e l'unità *floating-point* di cui si vuole effettuare il test. La *board* di sviluppo usata è la *DE0-nano* di *Altera* il cui *FPGA* è della famiglia *Cyclone-IV E.*

Il sistema finale si sviluppa con l'uso dei seguenti software:

- *Quartus II 13.1*: usato per la sintesi e per la programmazione del sistema sulla *board DE0-nano*;
- *Qsys 13.1*: tool di *Quartus*, utile alla creazione del *system-on chip*. In esso sono disponibili componenti come *timer*, *PIO*, *RAM* e *microprocessore*, utili per il corretto funzionamento del sistema;
- *Nios II 13.1 Software Build Tools:* usato per lo sviluppo del *firmware* di gestione del sistema finale, oltre che per effettuare le simulazioni.

Lo scopo ultimo di questa fase di progetto è da una parte automatizzare la fase di test, per ottenere, in tempi abbastanza rapidi, i risultati della somma/sottrazione *floating-point* e in secondo luogo avvicinarsi il più possibile al test dell'implementazione fisica dell'unità.

Per fare ciò, ci si serve di un apposito *firmware*, che dal microprocessore, anch'esso sintetizzato e programmato su *FPGA*, genera dei valori *random* da inviare all'unità *floating-point*. Quest'ultima dopo un certo *delay*, restituirà il valore del suo calcolo al microprocessore, che verrà infine visualizzato nella console dell'ambiente di sviluppo.

Il *system-on chip* pensato e successivamente progettato è visibile in Figura 5.1.

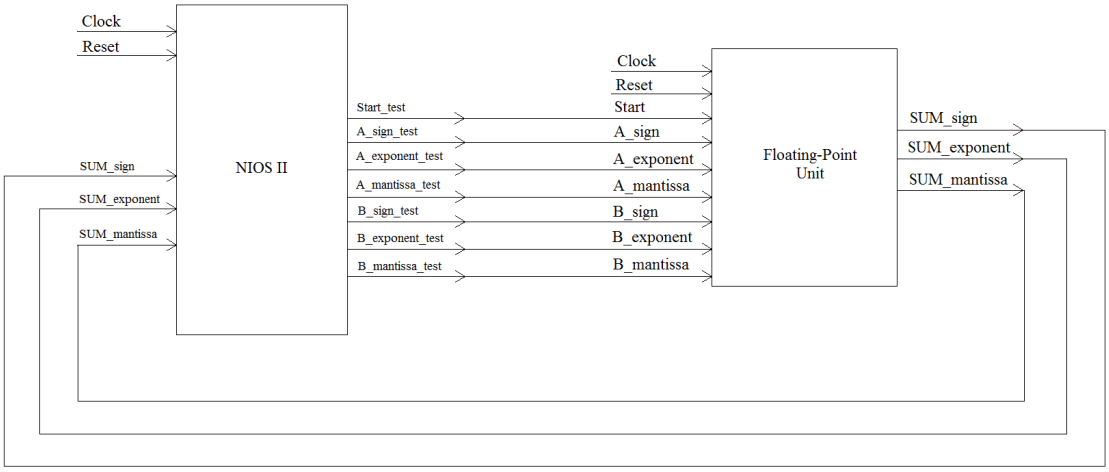


Figura 5.1: system-on chip finale, per effettuare i test.

5.1 Progettazione system-on chip

Il sistema finale si implementa, sintetizza e, infine, programma su un *FPGA* con l'uso del sintetizzatore *Quartus 13.1*. I passaggi che seguiranno, sono stati fatti per tutte e dieci le tipologie di unità *floating-point* approssimate, oltre che per il sommatore/sottrattore esatto:

1. *Impostazioni iniziali*: in questa fase si setta l'ambiente di sviluppo *Quartus*, fornendo informazioni riguardanti il nome del progetto, della directory ed il modello della *board* di sviluppo. Infine, si aggiungono i file *VHDL* dell'unità *floating-point* da testare.
2. *Implementazione di un sistema custom in Qsys 13.1*: il blocco, che nella Figura 5.1 è stato chiamato *NIOS II*, viene implementato con il tool *Qsys 13.1*. Il blocco è composto da più componenti, di seguito la lista completa: microprocessore (*NIOS II*), *jtag*, *ram*, *timer* e tutta una serie di *PIO*. Il primo step consiste nella configurazione del microprocessore: nella libreria dei componenti si sceglie *Nios II Processor* e nella finestra successiva il modello *Nios II/e*, come mostrato in Figura 5.1.1.

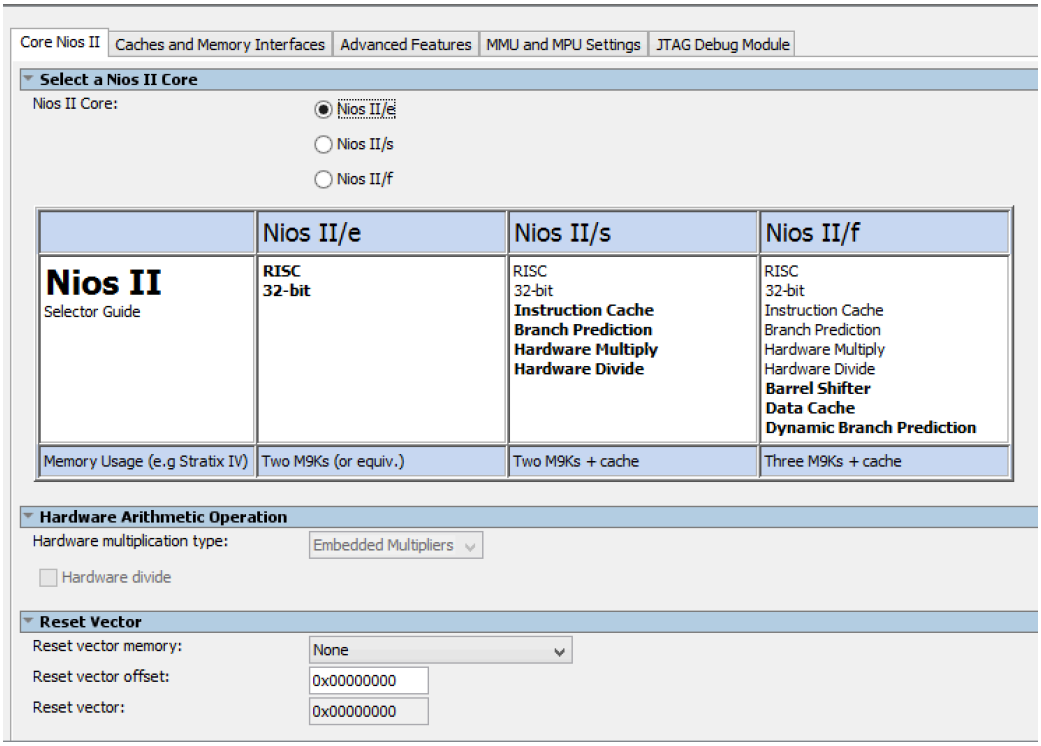


Figura 5.1.1: Si aggiunge il microprocessore al sistema

Successivamente si aggiungono i componenti chiamati *Jtag*, per la programmazione del *NIOS II*, *On-chip Memory* (*RAM*), scelta di 4 *MB*, ed un *timer* con risoluzione di 1 μs.

Infine, si inseriscono tutta una serie di *PIO* per la gestione via software dell'unità *floating-point*, di seguito la lista completa:

- *start*: *Width*=1 e *Direction=Output*;
- *A_sign_test*: *Width*=1 e *Direction=Output*;
- *B_sign_test*: *Width*=1 e *Direction=Output*;

- *A_exponent_test*: *Width*=8 e *Direction=Output*;
- *B_exponent_test*: *Width*=8 e *Direction=Output*;
- *A_mantissa_test*: *Width*=23 e *Direction=Output*;
- *B_mantissa_test*: *Width*=23 e *Direction=Output*;
- *SUM_sign*: *Width*=1 e *Direction=Input*;
- *SUM_exponent*: *Width*=8 e *Direction=Input*;
- *SUM_mantissa*: *Width*=23 e *Direction=Input*;

Queste periferiche sono dei registri configurabili il quale vengono collegati al microprocessore *NIOS II* e resi visibili dall'esterno del sistema implementato in Qsys, abilitando l'opzione *external_connection* nel campo *Export.* In questo modo è possibile effettuare il collegamento tra microprocessore ed unità *floating-point*, In Figura 5.1.2 il riassunto di quanto detto:

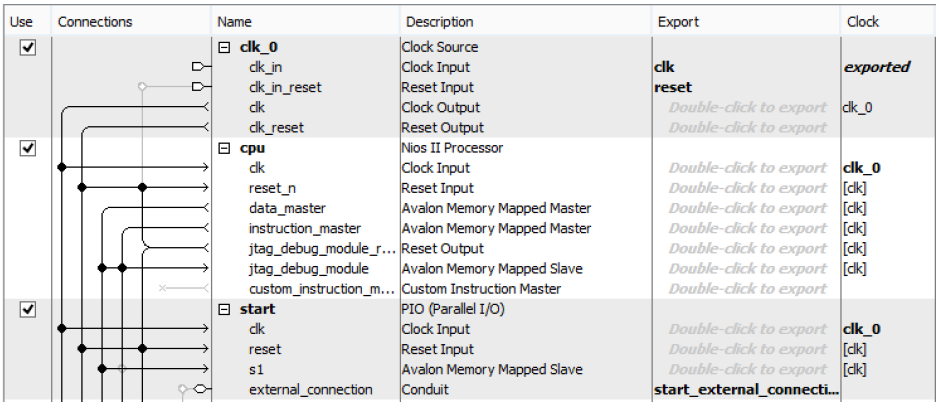


Figura 5.1.2: Abilitazione dell'opzione *external_connection* nel campo *Export* della periferica *PIO*, rinominata *start.*

Gli ultimi step, prima di generare il sistema sono:

- Selezione del settaggio *onchip_memory2.s1,* nel componente *Nios II Processor*, nel campo *reset vector memory* e nel campo *exception vector memory*;
- Si assegnano automaticamente i B*ase Addresses* e gli *Interrupt*.

Di seguito, in Figura 5.1.3, l'*Address Map* e la entity, Figura 5.1.4, del sistema finale generato da *Qsys*:

	cpu.data_master	cpu.instruction_master
cpu.jtag_debug_module	0x0002_0800 - 0x0002_0fff	0x0002_0800 - 0x0002_0fff
onchip_memory2.s1	0x0001_0000 - 0x0001_9fff	0x0001_0000 - 0x0001_9fff
timer.s1	0x0002_1000 - 0x0002_101f	
jtag_uart.avalon_jtag_slave	0x0002_10c0 - 0x0002_10c7	
start.s1	0x0002_10b0 - 0x0002_10bf	
A_sign.s1	0x0002_10a0 - 0x0002_10af	
B_sign.s1	0x0002_1090 - 0x0002_109f	
A_exp.s1	0x0002_1080 - 0x0002_108f	
B_exp.s1	0x0002_1070 - 0x0002_107f	
A_mantissa.s1	0x0002_1060 - 0x0002_106f	
B_mantissa.s1	0x0002_1050 - 0x0002_105f	
SUM_sign.s1	0x0002_1040 - 0x0002_104f	
SUM_exp.s1	0x0002_1030 - 0x0002_103f	
SUM_mantissa.s1	0x0002_1020 - 0x0002_102f	

Figura 5.1.3: Address map del sistema

```
component nios_fp_exact is
    port (
        clk_clk                              : in  std_logic                     := 'X';             -- clk
        reset_reset_n                        : in  std_logic                     := 'X';             -- reset_n
        start_external_connection_export     : out std_logic;                                        -- export
        a_sign_external_connection_export    : out std_logic;                                        -- export
        b_sign_external_connection_export    : out std_logic;                                        -- export
        a_exp_external_connection_export     : out std_logic_vector(7 downto 0);                     -- export
        b_exp_external_connection_export     : out std_logic_vector(7 downto 0);                     -- export
        a_mantissa_external_connection_export : out std_logic_vector(22 downto 0);                   -- export
        b_mantissa_external_connection_export : out std_logic_vector(22 downto 0);                   -- export
        sum_sign_external_connection_export  : in  std_logic                     := 'X';             -- export
        sum_exp_external_connection_export   : in  std_logic_vector(7 downto 0)  := (others => 'X'); -- export
        sum_mantissa_external_connection_export : in  std_logic_vector(22 downto 0) := (others => 'X')  -- export
    );
end component nios_fp_exact;
```

Figura 5.1.4: Entity del componente generato da Qsys

Le *PIO* aggiunte al sistema vengono implementate in *VHDL* con i tipi std_logic o std_logic_vector, semplificando in questo modo il processo di collegamento tra i due macro-blocchi.

Si passa in ultima analisi alla generazione del componente: il risultato della generazione sarà un unico file di formato *.qsys*. Basterà, quindi, aggiungere il file al progetto *Quartus*, per poter sintetizzare il sistema appena descritto.

Inoltre, il sistema generato da *Qsys* potrà essere collegato alla *floating-point unit* con il classico *port map( )*, come ogni qualsiasi altro componente progettato in *VHDL*.

3. *Interconnessioni e Top Level Entity*: quest'ultima fase riguarda il collegamento del componente *Nios II* all'unità *floating-point.* I due macro-blocchi vengono istanziati e collegati nella *Top Level Entity.* Le *PIO*, come visto in precedenza al punto 2, vengono implementate come *std_logic* o *std_logic_vector*, di *input* o *output*, a seconda dalla

funzione svolta. Nell'appendice, punto A, viene mostrata la *Top Level entity* del *system-on chip* per effettuare il test del sommatore/sottrattore *floating-point* esatto.

5.2 Sintesi del sistema e programmazione su FPGA

Dopo aver implementato la *Top Level Entity* si è pronti a sintetizzare il sistema e programmarlo sulla *board DE0-nano*. In Tabella 5.2.1 la gerarchia del sistema finale.

Tabella 5.2.1: Gerarchia system-on chip

Gerarchia	Entity
Livello 1	soc_fp_exact
Livello 2	nios_fp_exact, fp_unit
Livello 3	FSM_fp_unit, compare_and_sub, shift, twoS_complement, adder_external, rounding
Livello 4	adder_internal

La Figura 5.2.1, mostra la vista *RTL* del sistema finale, post-sintesi, estratta da *Quartus 13.1*. Si nota che il sistema inizialmente pensato e presentato all'inizio di questo capitolo, Figura 5.1, è esattamente uguale a quello ottenuto durante la sintesi.

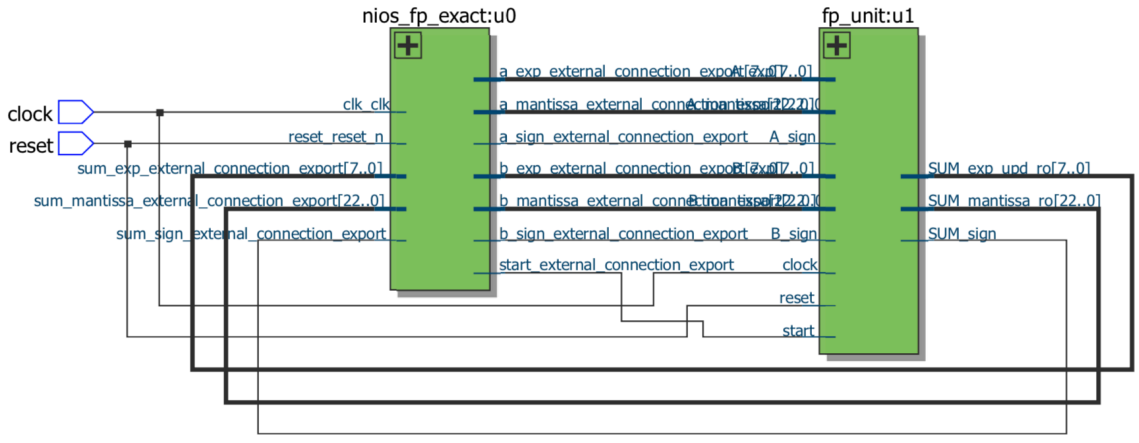


Figura 5.2.1: System-on chip, vista RTL. Immagine estrapolata da *Quartus 13.1, post-sintesi.*

L'ultima fase, prima della programmazione sulla *board*, riguarda il collegamento dei due *PIN*, *clock e reset*, sulla *board.*

In Figura 5.2.2, si mostra il settaggio del *Pin Planner* della *board*: il primo *pin* di ingresso del sistema, *clock*, viene collegato al *pin R8*, e l'*input reset* al *pin E1*. Il segnale di *clock* sarà così fornito da un oscillatore integrato nella *board* con frequenza di lavoro pari a 50 MHz. Il collegamento al pin E1 è riferito ad uno dei due *push-button* della *board*.

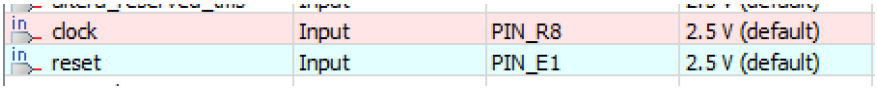


Figura 5.2.2: Pin planner del sistema.

Effettuato quest'ultimo step, si programma l'*FPGA* con il sistema sin qui descritto.

L'approccio scelto ha due grossi vantaggi:

- Permette di testare in modo diretto l'hardware implementato, avvicinandosi il più possibile al risultato che potrà essere ottenuto su silicio.
- Si automatizza la fase di testing: si implementa un *firmware* in grado di generare numeri ordinati o casuali, inviarli all'unità *floating-point* e, infine, acquisirli per poi visualizzarli in console.

Si sottolinea, nuovamente, che ogni unità *floating-point* progettata ed elencata nel Paragrafo 4.2, viene sintetizzata su *Quartus* e poi testata come fin qui detto.

5.3 Progettazione Firmware di gestione del sistema

Il codice di gestione del sistema viene implementato sul software *NIOS II 13.1 Software Build Tools for Eclipse* fornito da *Altera.* Prima di iniziare con la stesura del codice, si setta l'ambiente di sviluppo: si inizia aggiungendo il file d'estensione *.sopcinfo*, in cui vi sono presenti tutte le varie informazioni per generare i *BSP* di sistema. Questo file viene generato dal sintetizzatore *Quartus II 13.1*.

L'applicativo viene implementato servendosi delle funzioni presenti nelle varie librerie di sistema fornite da *Altera*, come la libreria *alt_timestamp.h*, che gestisce il *timer* inserito nel progetto e *altera_avalon_pio_regs.h*, il quale fornisce alcune funzioni di lettura e scrittura delle *PIO*.

Inoltre, in mancanza della funzione di stampa dei numeri *floating-point*, si implementa una libreria in grado di convertire un numero *floating-point* in rappresentazione binaria, in uno in rappresentazione decimale, e successivamente di stamparlo.

Nella libreria *system.h*, sono presenti i *Base Address* delle varie periferiche. Questi vengono usati nelle funzioni di lettura e scrittura delle periferiche *PIO*.

5.3.1 Implementazione delle librerie di gestione del sistema

La progettazione dell'applicativo ha inizio con lo studio di alcune funzioni presenti nelle librerie. La libreria *altera_avalon_pio_reg.h* contiene le seguenti funzioni di gestione dei registri *PIO*:

- *IOWR_ALTERA_AVALON_PIO_DATA(base, data),* permette la scrittura di un valore nella linea *PIO*;
- *IORD_ALTERA_AVALON_PIO_DATA(base),* permette la lettura di un valore nella linea *PIO.*

La prima funzione serve ad inviare all'unità *floating-point* i due numeri da sommare, mentre la seconda è usata per acquisire il risultato. È chiaro che ogni linea *PIO* è pilotata usando la propria *Base Address*, quindi, da come è stato concepito il sistema, la funzione di scrittura è chiamata per ben sei volte, quando si vogliono inviare due numeri A e B. Questo è dovuto al fatto che si sono usate sei linee diverse per mandare il dato, ovvero 2 linee di un bit per il segno, *A_sign* e *B_sign*, due linee di 8 bit per inviare gli esponenti, *A_exponent* e *B_exponent*, e altre due linee da 23 bit per le due mantisse, *A_mantissa* e *B_mantissa*. Inoltre, per semplicità, si implementano tre funzioni visibili nell'appendice al punto B, aventi come argomento, il valore da inviare all'unità *floating-point*.

La stessa situazione si ripropone in fase di acquisizione del valore derivante dall'unità *floating-point.* Infatti, la funzione di lettura, *IORD_ALTERA_AVALON_PIO_DATA(base)*, è chiamata per ben tre volte, per permettere la lettura delle tre linee *SUM_sign*, *SUM_exponent* e *SUM_mantissa*.

L'insieme delle funzioni utili alla lettura e alla scrittura delle *PIO,* si raggruppano in una libreria, nominata *general_function.h.*

Si implementa una seconda libreria, nominata *delay_func.h*, utile alla temporarizzazione dell'esecuzione dell'applicativo: *alt_timestamp.h* è la libreria usata per gestire il componente *timer.* Le due funzioni di libreria usate sono *alt_timestamp_start()* e *alt_timestamp()* e svolgono le seguenti operazioni:

- *void alt_timestamp_start()*, *reset* del timer al valore iniziale '0';
- *int alt_timestamp()*, ha come valore di ritorno il numero di *clock ticks* intercorsi dall'ultima chiamata di *alt_timestamp_start*().

Usando un ciclo *while*, è possibile bloccare l'esecuzione del programma per il tempo voluto. Nel caso specifico, si implementata una funzione che stoppa il programma, nei punti desiderati, per 7 ms. Questo parametro è stato scelto in modo casuale e quindi nulla vieta di aumentare o abbassare questo valore.

Infine, l'ultima libreria implementata, riguarda la stampa di numeri *floating-point*, nominata *printf_float.h.* L'implementazione di questa libreria è dovuta al fatto che la libreria nativa di stampa presente nel sistema *NIOS II*, è una versione ridotta della *stdio.h*, che prende il nome di *alt_stdio.h*, ed in essa non è presente la gestione dei tipi di dato *float.* Quindi, si implementa, in primis, una funzione di conversione da un numero binario in rappresentazione *floating-point*, a decimale, e, successivamente, un algoritmo di stampa del dato. In questa fase di progettazione, vengono gestite anche le eccezioni: infatti, nel caso in cui il numero generato sia un NaN, oppure la somma di due numeri restituisca un NaN, l'algoritmo di stampa mostrerà questo valore in console.

5.3.2 Sviluppo delle librerie di testing

Per testare efficacemente le unità *floating-point* si decide di utilizzare, per ogni tipo di simulazione, un set di 2500 valori. Si effettuano principalmente due tipi di test:

- Si generano numeri casuali in un determinato range di valori;
- Si applicano in ingresso delle rampe, crescenti o decrescenti, mantenendo un operando fisso, e variando l'altro.

Riguardo la generazione di numeri casuali, da un'attenta analisi, si giunge alla conclusione che per il *testing* dell'unità della somma algebrica *floating-point,* generare un set di campioni in tutto il range di numeri può rilevarsi inutile, al contrario di quanto potrebbe accadere nella moltiplicazione *floating-point*. Infatti, se la differenza tra due esponenti è maggiore di 23, non viene effettuata alcune operazione, ed il risultato è semplicemente il numero con l'esponente maggiore. La spiegazione di quanto detto è da ricercarsi sia nella struttura della rappresentazione dei numeri *floating-point* che nell'algoritmo di somma/sottrazione: infatti, se la differenza è maggiore di 23, l'operazione di *shift* della mantissa con esponente minore, porta ad avere una sequenza di zeri in essa e, quindi, il sommatore è come se non effettuasse alcuna operazione. Per questo motivo, la libreria che genera i numeri casuali di test, viene implementata tenendo conto di quanto detto, evitando di generare numeri la cui differenza di esponenti possa superare il numero 23.

Lo scopo ultimo di questi test è valutare la bontà di ogni sommatore/sottrattore implementato, valutando i due parametri di caratterizzazione dell'errore, *MED* e *MRED.* I due parametri hanno bisogno di un riferimento per essere calcolati, cioè di un'unità che esegua i calcoli correttamente. Di conseguenza tutti i sommatori approssimati verranno confrontati con il sommatore/sottrattore *floating-point* esatto implementato e discusso in precedenza.

Di seguito l'elenco dei test effettuati:

- Test per valutare il comportamento e il peso del *rounding* delle unità *floating-point* con numeri maggiori di 1. Si usa un set di valori con le seguenti caratteristiche:
  - Segno e mantissa dei due numeri viene generato casualmente;
  - Esponente, di valore causale, ma compreso tra 2^2 e 2^4.
- Test per valutare il comportamento dell'unità con numeri minori di zero. Sono stati valutati tre intervalli di valori:
  - Da 2^1 a 2^{-23}, con segno sempre positivo e mantissa qualsiasi;
  - Da 2^1 a 2^{-23}, con segno sempre negativo e mantissa qualsiasi;
  - Da 2^1 a 2^{-23}, con segno e mantissa qualsiasi.
- Test usando una rampa crescente o decrescente: si mantiene l'operando A costante e si varia B, sommando o sottraendo a quest'ultimo un certo ΔB.

Si implementano, quindi, due librerie diverse:

1. *genereta_value.h*: gestisce la generazione di numeri casuali;
2. *test_ramp.h*: manda in ingresso dell'operando B una rampa crescente o decrescente.

5.4 Test delle unità e caratterizzazione dell'errore

Il test delle unità di calcolo viene eseguito in due step principali:

1. Generazione dei valori, invio all'unità *floating-point* e stampa su console.
2. Acquisizione dei risultati delle somme/sottrazioni effettuate e stampa su console.

Tutti i valori stampati su console vengono estratti e successivamente copiati in un foglio di calcolo, per la valutazione dell'errore.

5.4.1 Valutazione dei parametri di caratterizzazione dell'errore e rounding

I primi test effettuati sono atti a caratterizzare gli errori dei sommatori/sottrattori implementati. I parametri da calcolare sono rispettivamente il *Mean Error Distance*, o *MED*, e il *Mean Relative Distance Error*, o *MRED.*

Come accennato nei capitoli precedenti, si effettuano dei test per valutare se il processo di *rounding*, dispendioso dal punto di vista di area e potenza, sia necessario in caso di unità approssimate o meno.

Si effettuano i test a tutti i sommatori/sottrattori implementati, di cui si è fatto ampiamente cenno nei capitoli precedenti (Paragrafo 4.2.1), generando un set di 2500 valori, con segno e mantissa casuali e esponente compreso tra 2^2 e 2^4. Il range di questi valori varia, quindi tra 4 e 32.

In Tabella 5.4.1.1 i risultati ottenuti:

Tabella 5.4.1.1: Risultati simulazioni con A e B casuali e compresi tra 4 e 32

Adder	MED	MED NO ROUNDING	MRED	MRED NO ROUNDING
Exact	0	/	0	/
RCA12-RcHA11	0,000366664	0,000367086	7,14369E-05	7,14569E-05
RCA12-LOA 11	0,000550863	0,000551218	0,000133226	0,000133234
RCA12-Trun 11	0,002945293	0,002945098	0,000678431	0,000678413
RCA11-Trun12	0,005921074	0,005920879	0,001407979	0,001407961
RCA6-RcHA5-LOA12	0,023028803	0,023029422	0,006603445	0,006603459
RCA5-RcHA5-LOA5-Trun8	0,045238421	0,045238252	0,011075284	0,011075268
RCA5-LOA5-Trun13	0,073700367	0,073700279	0,015473968	0,015473959
LOA-23	2,428125805	2,428125715	0,283024639	0,283024624
LOA13-Trun10	2,428650695	2,428650483	0,283241600	0,283241576
LOA8-Trun15	2,444679833	2,444679621	0,289474406	0,289474382

I risultati, ordinati in ordine crescente, forniscono informazioni riguardanti la precisione dei sommatori approssimati. Il sommatore che garantisce prestazioni migliori è l'*RCA12-RcHA11*, cioè un sommatore in cui i 12 bit più significativi vengono sommati con un *Ripple-Carry Adder*, e gli ultimi 11 con un *Ripple-Carry Half Adder*. Si sottolinea inoltre, che tra i primi tre sommatori, tutti con i primi 12 *MSB* esatti, e i sommatori *LOA-23*, *LOA13-Trun10* e *LOA8-Trun15*, si ha una differenza di circa 4 ordini di grandezza. Interessanti anche le piccole differenze tra i sommatori *LOA-23* e *LOA13-Trun10*, in cui a scapito di un piccolo aumento di *MRED* e *MED*, si ha un risparmio circa del 40% delle porte logiche del sommatore/sottrattore *fixed-point* di mantisse.

Dai valori di *MRED* e *MED*, con e senza *rounding*, risulta evidente che la differenza tra i due parametri dello stesso tipo è molto piccola. Per questo motivo, si arriva alla conclusione che il *rounding*, quando si ha un sommatore/sottrattore *floating-point*, può essere omesso, così come è stato fatto nell'articolo [21]. Per questo motivo i successivi test sono attuati alle unità approssimate senza *rounding.*

5.4.2 Valutazione dei sommatori con numeri minori di zero

In questa parte di simulazioni, si vuole studiare il comportamento delle unità *floating-point* quando si hanno in ingresso valori minori di zero ai due operandi *A* e *B*. Le motivazioni di questa analisi, vanno ricercate nella caratteristica intrinseca della rappresentazione *floating-point*. Infatti, la distanza tra due numeri consecutivi minore di zero è molto piccola, al contrario della distanza che intercorre tra due numeri consecutivi molto grandi, e si vuole, quindi, analizzare il comportamento delle unità progettate in questo specifico range.

In queste simulazioni si analizza anche la percentuale di errore per eccesso o per difetto delle unità sotto analisi. Per questo motivo, per ogni range di valori vengono calcolati due tipi diversi di *MRED* e *MED*. Ricordando le formule di base, che sono (3.1) e (3.2), si avranno due tipi di parametri:

1. Considerando il valore assoluto nel calcolo dell'*errore distance;*
2. Eliminando il valore assoluto.

Le modifiche alle formule dei parametri di caratterizzazione dell'errore sono necessarie, in quanto senza l'omissione del valore assoluto, sarebbe impossibile capire se l'errore commesso dalle unità approssimate sia per eccesso o per difetto. Si ha un errore per eccesso se $ED = M^i - M > 0$, mentre per difetto se $ED = M^i - M < 0$.

Si effettuano tre simulazioni differenti, cambiando il range di valori come segue:

1. Da 2^1 a 2^{-23}, con segno sempre positivo e mantissa qualsiasi;
2. Da 2^1 a 2^{-23}, con segno sempre negativo e mantissa qualsiasi;
3. Da 2^1 a 2^{-23}, con segno e mantissa qualsiasi.

Con i primi due range di valori si valuta la percentuale di errore per eccesso o difetto. Per avere coerenza nei risultati, si sceglie di effettuare solo somme, fissando il valore del segno costante, o sempre positivo, o sempre negativo.

Infine, nell'ultima simulazione, si prendono in considerazione sia numeri positivi che negativi, e quindi verranno calcolate somme e differenze, così da caratterizzare il comportamento dei sommatori/sottrattori anche in caso di sottrazione, avendo numeri minori di zero. In questo caso non si valuta la percentuale di errore per eccesso o difetto.

Come detto in precedenza, il primo set di valori viene scelto tra 2^{1} e 2^{-23}, con segno positivo e mantissa qualsiasi. La Tabella 5.4.2.1 espone i risultati ottenuti di *MED* e *MRED* applicando il valore assoluto, mentre non è applicato, per il calcolo dei due parametri, in Tabella 5.4.2.2.

Tabella 5.4.2.1: MED e MRED calcolati usando il valore assoluto

Adder	MED NO ROUNDING	MRED NO ROUNDING
Exact	/	/
RCA12-RcHA11	3,76245E-06	1,81029E-05
RCA12-LOA 11	5,26701E-06	2,57324E-05
RCA12-Trun 11	3,31492E-05	0,000145347
RCA11-Trun12	6,50077E-05	0,000283755
RCA6-RcHA5-LOA12	0,000140726	0,000810869
RCA5-RcHA5-LOA5-Trun8	0,000251380	0,001461357
RCA5-LOA5-Trun13	0,000454338	0,002412533
LOA-23	0,005112785	0,027843041
LOA13-Trun10	0,005124778	0,027894795
LOA8-Trun15	0,005475414	0,029423866

Tabella 5.4.2.2: MED e MRED calcolati senza l'uso del valore assoluto

Adder	MED NO ROUNDING	MRED NO ROUNDING
Exact	/	/
RCA12-RcHA11	3,76245E-06	1,81029E-05
RCA12-LOA 11	7,54557E-08	6,86551E-07
RCA12-Trun 11	3,31492E-05	0,000145347
RCA11-Trun12	6,50077E-05	0,000283755
RCA6-RcHA5-LOA12	0,000140726	0,000810869
RCA5-RcHA5-LOA5-Trun8	0,000248294	0,001443102
RCA5-LOA5-Trun13	0,000114675	0,000672202

Adder	MED NO ROUNDING	MRED NO ROUNDING
LOA-23	0,001055246	0,010567497
LOA13-Trun10	0,001068328	0,010623146
LOA8-Trun15	0,001445800	0,012275918

La Tabella 5.4.2.3 riassume i valori percentuali dell'errore per eccesso o per difetto. Dai risultati ottenuti, si può affermare che per quasi tutte le tipologie di sommatori approssimati l'errore sarà per difetto, superando abbondantemente l'80% dei campioni presi in esame. I sommatori *RCA12-Trun11* e *RCA11-Trun12* hanno il 100% dei campioni con errore per difetto. Solo il sommatore *RCA12-LOA11* ha percentuali più basse, circa il 70% dei campioni, e ciò influisce positivamente nel parametro *MRED* calcolato, però, senza valore assoluto.

Tabella 5.4.2.3: Valori percentuali di errore per eccesso o difetto dei campioni valutati

Adder	% difetto	(campioni)	% eccesso	(campioni)
RCA12-RcHA11	94.5%	(2368)	5.5%	(132)
RCA12-LOA 11	69.9%	(1910)	30.1%	(590)
RCA12-Trun 11	100%	(2500)	0%	(0)
RCA11-Trun12	100%	(2500)	0%	(0)
RCA6-RcHA5-LOA12	99.36%	(2484)	0.64%	(16)
RCA5-RcHA5-LOA5-Trun8	95.8%	(2398)	4.2%	(102)
RCA5-LOA5-Trun13	84.5%	(2163)	15.5%	(337)
LOA-23	93.9%	(2355)	6.1%	(145)
LOA13-Trun10	94%	(2358)	6.0%	(142)
LOA8-Trun15	94%	(2358)	6.0%	(142)

La seconda parte di simulazioni viene effettuata generando gli stessi valori casuali della simulazione esaminata in precedenza, ma con segno sempre negativo. Anche in questo caso vengono analizzati i due parametri di caratterizzazione dell'errore con e senza valore assoluto.

In Tabella 5.4.2.4 si hanno i risultati delle simulazioni, tenendo conto del valore assoluto, e nella tabella successiva, Tabella 5.4.2.5, i risultati senza valore assoluto. Infine, i valori percentuali dei campioni calcolati per eccesso o per difetto, Tabella 5.4.2.6.

Tabella 5.4.2.4: Risultati simulazioni. MED e MRED calcolati usando il valore assoluto

Adder	MED NO ROUNDING	MRED NO ROUNDING
Exact	/	/
RCA12-RcHA11	3,76245E-06	1,81029E-05
RCA12-LOA 11	5,26701E-06	2,57324E-05
RCA12-Trun 11	3,31492E-05	0,000145347
RCA11-Trun12	6,50077E-05	0,000283755
RCA6-RcHA5-LOA12	0,000140726	0,000810869
RCA5-RcHA5-LOA5-Trun8	0,00025138	0,001461357
RCA5-LOA5-Trun13	0,000454338	0,002412533
LOA-23	0,005112785	0,027843041
LOA13-Trun10	0,005124778	0,027894795
LOA8-Trun15	0,005475414	0,029423866

Tabella 5.4.2.5: MED e MRED calcolati senza l'uso del valore assoluto

Adder	MED NO ROUNDING	MRED NO ROUNDING
Exact	/	/
RCA12-RcHA11	-3,76245E-06	1,81029E-05
RCA12-LOA 11	-7,54557E-08	6,86551E-07
RCA12-Trun 11	-3,31492E-05	0,000145347
RCA11-Trun12	-6,50077E-05	0,000283755
RCA6-RcHA5-LOA12	-0,000140726	0,000810869
RCA5-RcHA5-LOA5-Trun8	-0,000248294	0,001443102
RCA5-LOA5-Trun13	-0,000114675	0,000672202
LOA-23	-0,001055246	0,010567497
LOA13-Trun10	-0,001068328	0,010623146
LOA8-Trun15	-0,001445800	0,012275918

Tabella 5.4.2.6: Valori percentuali di errore per eccesso o difetto

Adder	% difetto	(campioni)	% eccesso	(campioni)
RCA12-RcHA11	94.5%	(2368)	5.5%	(132)
RCA12-LOA 11	69.9%	(1910)	30.1%	(590)
RCA12-Trun 11	100%	(2500)	0%	(0)
RCA11-Trun12	100%	(2500)	0%	(0)
RCA6-RcHA5-LOA12	99.36%	(2484)	0.64%	(16)
RCA5-RcHA5-LOA5-Trun8	95.8%	(2398)	4.2%	(102)
RCA5-LOA5-Trun13	84.5%	(2163)	15.5%	(337)
LOA-23	93.9%	(2355)	6.1%	(145)
LOA13-Trun10	94%	(2358)	6.0%	(142)
LOA8-Trun15	94%	(2358)	6.0%	(142)

Analizzando i risultati si conclude, affermando che le unità *floating-point* approssimate hanno lo stesso comportamento nel range di valori minori di zero, positivi o negativi. Infatti, hanno gli stessi valori percentuali di errore per eccesso o difetto e gli stessi parametri di caratterizzazione degli errori, *MRED* e *MED*. Si nota, inoltre, che i sommatori in cui viene usata la tecnica *LOA-k* hanno percentuali di errore per eccesso o difetto più alte rispetto al resto delle unità *floating-point* approssimate.

L'ultima parte di simulazione, anch'essa avendo in esame campioni minori di zero, viene effettuata generando valori casuali, ma questa volta con segno casuale, così da effettuare sia somme che sottrazioni. I risultati sono esposti in Tabella 5.4.2.7.

Tabella 5.4.2.7: Risultati simulazioni.

Adder	MED NO ROUNDING	MRED NO ROUNDING
Exact	/	/
RCA12-RcHA11	3,8845E-06	2,7716E-05
RCA12-LOA 11	4,64278E-06	3,52165E-05
RCA12-Trun 11	3,07623E-05	0,000213457
RCA11-Trun12	6,30923E-05	0,000426633
RCA6-RcHA5-LOA12	0,000171534	0,001448155
RCA5-RcHA5-LOA5-Trun8	0,000334901	0,003419812
RCA5-LOA5-Trun13	0,000506722	0,004814583

Adder	MED NO ROUNDING	MRED NO ROUNDING
LOA-23	0,010382219	0,087260932
LOA13-Trun10	0,010391814	0,087322391
LOA8-Trun15	0,010677246	0,089024881

Questi risultati sono linea con i valori ottenuti nelle simulazioni precedenti, esposti in Tabella 5.4.2.1 e 5.4.2.4.

5.4.3 Test delle unità avendo come input una rampa crescente o decrescente

In questa sezione, si analizza il comportamento dei sommatori/sottrattori *floating-point*, ponendo un valore costante come ingresso di un operando, e una rampa crescente o decrescente come *input* del secondo operando. L'obiettivo, anche in questo caso, è caratterizzare l'errore.

In queste simulazioni vengono analizzati solo 4 tipi di sommatori, che sono: *RCA12-rcHA11*, *RCA12-LOA11*, *RCA12-Trun11* e *LOA-23*.

Il numero *floating-point* $+2^9 \cdot 10101010101010101010101$ viene posto in ingresso dell'operando A, mentre, $+2^9 \cdot 10000000000000000000000$, è il valore di partenza dell'operando B. Nella libreria *test_ramp.h* viene implementata una funzione che esegue un ciclo *for* di 2500 iterazioni, in cui viene sommata o sottratta alla mantissa dell'operando B, una certa quantità ΔB, pari a, 00000000000000010000000.

Nella Tabella 5.4.3.1, si hanno i risultati delle simulazioni, avendo come ingresso di B una rampa crescente. Quindi si avrà:

- $A = +2^9 \cdot 10101010101010101010101$;
- $B = +2^9 \cdot 10000000000000000000000$;
- $\Delta B = 00000000000000010000000$.

Tabella 5.4.3.1: Risultati simulazioni con rampa crescente in B

Adder	MED NO ROUNDING	MRED NO ROUNDING
RCA12-RcHA11	0,007800148	4,78219E-06
RCA12-LOA 11	0,03121246	1,9136E-05
RCA12-Trun 11	0,141767889	8,69166E-05
LOA-23	250,817	0,153782874

I valori di *MRED* sono confrontabili con i valori ottenuti nelle simulazioni precedenti. Il valore di *MED* del sommatore *LOA-23* è molto alto, ma non essendo un parametro relativo, non è rilevante.

La stessa analisi viene fatta per analizzare il comportamento dei sommatori/sottrattori avendo in ingresso una rampa decrescente. Gli *input* usati sono gli stessi usati in precedenza, ma se nella simulazione precedente, alla seconda iterazione, si aveva $B = B + \Delta B$, adesso si ha $B = B - \Delta B$. In Tabella 5.4.3.2 i risultati ottenuti.

Tabella 5.4.3.2: Risultati simulazioni con rampa decrescente in B

Adder	MED NO ROUNDING	MRED NO ROUNDING
RCA12-RcHA11	0,007825148	4,85566E-06
RCA12-LOA 11	0,031262459	1,93988E-05
RCA12-Trun 11	0,141880389	8,80394E-05
LOA-23	80,22018857	0,049768846

Anche in questo caso, non si riscontrano casi particolari su cui investigare, in quanto i valori di *MED* ed *MRED* sono in linea con quelli ottenuti in precedenza, avendo gli stessi ordini di grandezza.

6. Sintesi, ottimizzazione e confronto dei modelli

Quest'ultima parte di elaborato è incentrata sulla sintesi dei sommatori/sottrattori *floating-point* su ambiente *Synopsys Design Compiler*. La tecnologia usata in fase di sintesi è la *BCD8SP* (180 nm) di *STMicroelectronics* con celle *low leakage* e *high speed.*

I sommatori algebrici sintetizzati sono i seguenti:

- *fp_exact_with_round*;
- *fp_12rca11rcha_without_round*;
- *fp_12rca11loa_without_round*;
- *fp_12rca11trun_without_round*;
- *fp_11rca12trun_without_round*;
- *fp_6rca5rcha12loa_without_round*;
- *fp_5rca5rcha5loa8trun_without_round*;
- *fp_5rca5loa13trun_without_round*;
- *fp_loafull_without_round*;
- *fp_13loa10trun_without_round*;
- *fp_8loa15trun_without_round*.

L'obiettivo è valutare i risparmi in termini di area occupata e potenza dissipata di ogni unità approssimata rispetto all'unità *floating-point* esatta.

6.1 Ottimizzazioni per superare le violazioni sul timing

La prima sintesi effettuata riguarda l'unità aritmetica esatta, presentata Paragrafo 4.1. Il report *timing*, usando 100 MHz come frequenza di sistema, restituisce delle violazioni di *timing* nei seguenti path:

```
ist_addition/A_mantissa_comp_reg_1x/CP  (FD4QHDX4) ist_round_1/SUM_exp_upd_ro_reg_7x/D   (FD4SQHDX2)   -0.827
ist_addition/A_mantissa_comp_reg_1x/CP  (FD4QHDX4) ist_round_1/SUM_exp_upd_ro_reg_6x/D   (FD4SQHDX2)   -0.827
ist_addition/A_mantissa_comp_reg_1x/CP  (FD4QHDX4) ist_round_1/SUM_mantissa_ro_reg_21x/D (FD4SQHDX1)   -0.825
ist_addition/A_mantissa_comp_reg_1x/CP  (FD4QHDX4) ist_round_1/SUM_mantissa_ro_reg_18x/D (FD4SQHDX1)   -0.822
ist_addition/A_mantissa_comp_reg_1x/CP  (FD4QHDX4) ist_round_1/SUM_mantissa_ro_reg_17x/D (FD4SQHDX1)   -0.821
ist_addition/A_mantissa_comp_reg_1x/CP  (FD4QHDX4) ist_round_1/SUM_mantissa_ro_reg_16x/D (FD4SQHDX1)   -0.819
ist_addition/A_mantissa_comp_reg_20x/CP (FD4QHDX4) ist_round_1/SUM_exp_upd_ro_reg_4x/D   (FD4SQHDX2)   -0.806
ist_addition/A_mantissa_comp_reg_1x/CP  (FD4QHDX4) ist_round_1/SUM_mantissa_ro_reg_13x/D (FD4SQHDX1)   -0.806
ist_addition/A_mantissa_comp_reg_1x/CP  (FD4QHDX4) ist_round_1/SUM_mantissa_ro_reg_14x/D (FD4SQHDX1)   -0.805
ist_addition/A_mantissa_comp_reg_1x/CP  (FD4QHDX4) ist_round_1/SUM_mantissa_ro_reg_19x/D (FD4SQHDX1)   -0.805
ist_addition/A_mantissa_comp_reg_1x/CP  (FD4QHDX4) ist_round_1/SUM_mantissa_ro_reg_15x/D (FD4SQHDX1)   -0.803
ist_addition/A_mantissa_comp_reg_20x/CP (FD4QHDX4) ist_round_1/SUM_exp_upd_ro_reg_3x/D   (FD4SQHDX2)   -0.796
ist_addition/A_mantissa_comp_reg_1x/CP  (FD4QHDX4) ist_round_1/SUM_exp_upd_ro_reg_5x/D   (FD4SQHDX2)   -0.795
ist_addition/A_mantissa_comp_reg_1x/CP  (FD4QHDX4) ist_round_1/SUM_mantissa_ro_reg_22x/D (FD4SQHDX1)   -0.794
ist_addition/A_mantissa_comp_reg_20x/CP (FD4QHDX4) ist_round_1/SUM_mantissa_ro_reg_7x/D  (FD4SQHDX1)   -0.784
ist_addition/A_mantissa_comp_reg_20x/CP (FD4QHDX4) ist_round_1/SUM_exp_upd_ro_reg_2x/D   (FD4SQHDX2)   -0.783
ist_addition/A_mantissa_comp_reg_20x/CP (FD4QHDX4) ist_round_1/SUM_mantissa_ro_reg_8x/D  (FD4SQHDX1)   -0.783
ist_addition/A_mantissa_comp_reg_20x/CP (FD4QHDX4) ist_round_1/SUM_mantissa_ro_reg_6x/D  (FD4SQHDX1)   -0.778
ist_addition/A_mantissa_comp_reg_20x/CP (FD4QHDX4) ist_round_1/SUM_mantissa_ro_reg_5x/D  (FD4SQHDX1)   -0.778
ist_addition/A_mantissa_comp_reg_20x/CP (FD4QHDX4) ist_round_1/SUM_mantissa_ro_reg_11x/D (FD4SQHDX1)   -0.762
```

Prima modificare l'architettura dell'unità sintetizzata, si prova a superare queste violazioni, scalando la frequenza di funzionamento, portandola a 50 MHz. Purtroppo le violazioni dei path precedenti continuano a persistere. Inoltre, si trovano le stesse violazioni anche nel secondo operando, ovvero *B_mantissa_comp_reg*.

In Figura 6.1.1, sono messi in risalto i path in cui si hanno violazioni. Queste sono dovute al tipo di architettura usata: infatti, in fase di progettazione, si decide di effettuare l'operazione di somma e normalizzazione in un unico *clock cycle.* I problemi di *timing* sono riconducibili a questa scelta, in quanto non è possibile effettuare queste due operazioni in un così breve arco temporale. Si decide di applicare la tecnica di ottimizzazione *pipelining*, aggiungendo un registro tra le due operazioni, per risolvere queste violazioni.

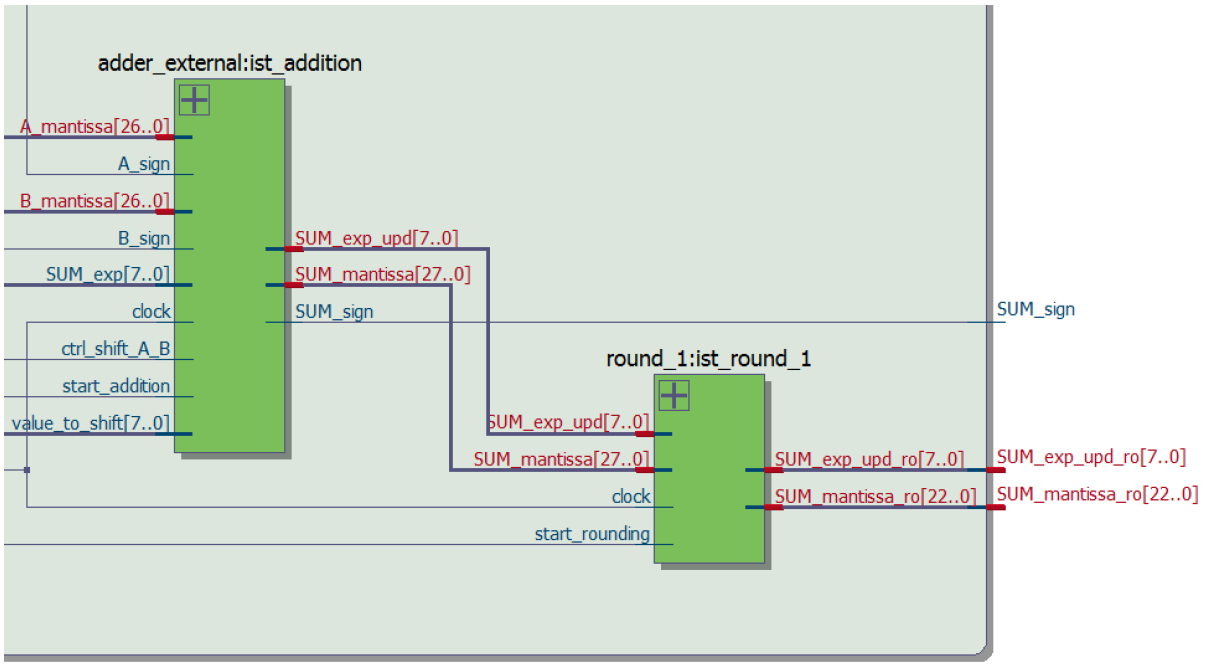


Figura 6.1.1: critical path dell'unità floating-point esatta

L'applicazione di questa tecnica di ottimizzazione avviene mediante l'aggiunta di uno stato alla *FSM*, chiamato *Normalization.* Il blocco funzionale *adder_external*, viene quindi diviso in due moduli:

1. Il primo effettua la somma delle mantisse, al quale si lascia invariato il nome *adder_external*;
2. Il secondo modulo normalizza il risultato della somma effettuata in precedenza e prende il nome di *normalization*.

Il blocco *normalization*, viene provvisto di un segnale di *start*, attivo al livello logico '1', che viene collegato alla macchina a stati finiti. Anche in questo caso, il segnale di *start* viene inserito nella *sensitivity list* del modulo.

Con queste modifiche, l'unità *floating-point* esatta fornisce in uscita il valore della somma/ sottrazione di due numeri al 6° *clock cycle*, e non più al 5°, come in precedenza.

In Tabella 6.1.1 vi è la struttura gerarchica della nuova unità floating point e in Figura 6.1.2 la *FSM* aggiornata e fornita dello stato di normalizzazione.

Tabella 6.1.1: Gerarchia sommatore/sottrattore floating-point esatto

Gerarchia	Entity
Livello 1	fp_unit
Livello 2	FSM_fp_unit, compare_and_sub, shift, twoS_complement, adder_external, normalization, rounding
Livello 3	adder_internal

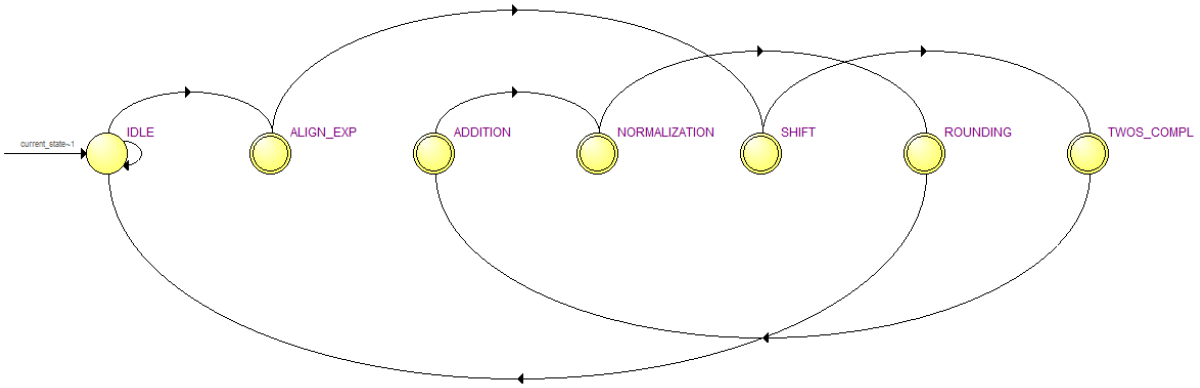


Figura 6.1.2: FSM della unità floating-point esatta provvista dello stato di normalizzazione

L'esecuzione algoritmica della nuova unità aritmetica *floating-point* è esattamente uguale alle versioni analizzate nei capitoli precedenti e per questo motivo non vengono ricalcolati i parametri di caratterizzazione dell'errore, quali *MRED* e *MED*. In Figura 6.1.3, una simulazione effettuata su ModelSim, nel quale viene sottolineata la presenza di un *clock cycle* aggiuntivo, che svolge la normalizzazione.

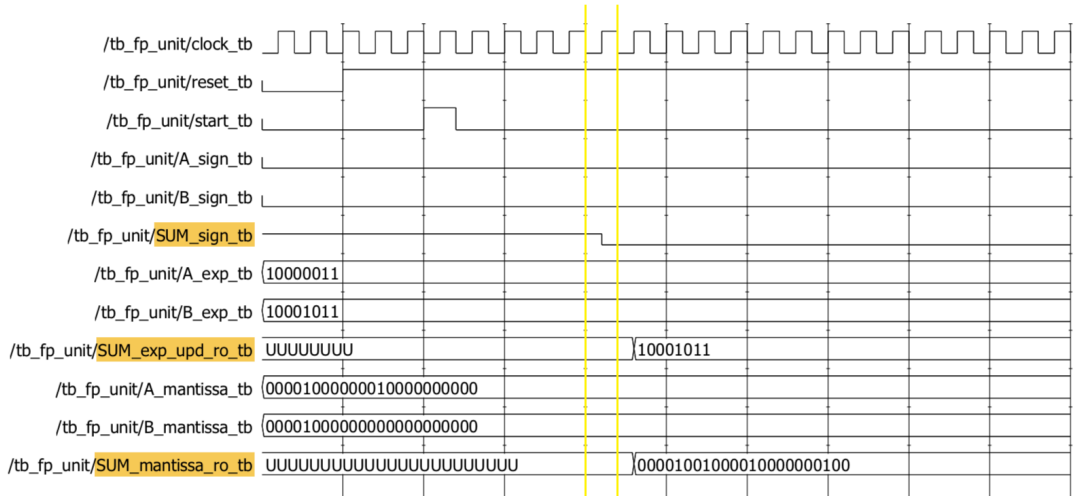


Figura 6.1.3: Simulazione ModelSim del sommatore/sottrattore ‘pipelinato’

Infine, in Figura 6.1.4, si mostra la vista *RTL*, ricavata dal sintetizzatore *Quartus*, in cui si sottolinea il nuovo critica path.

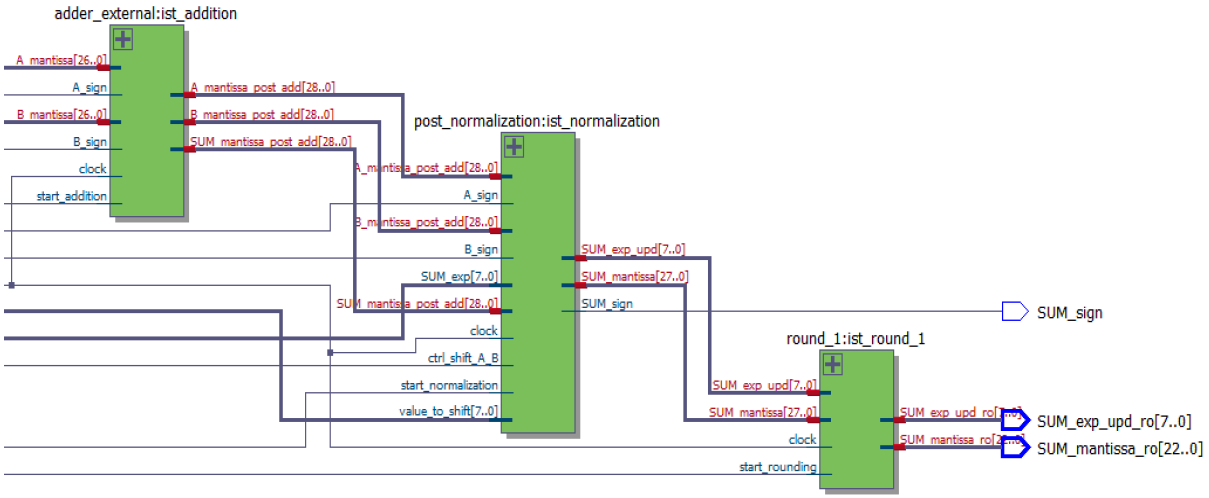


Figura 6.1.4: critical path dell’unità ottimizzata.

In conclusione, il problema delle violazioni di *timing*, grazie all’ausilio di questa tecnica di ottimizzazione, è bypassato.

In tutti i sommatori/sottrattori approssimati, elencati in precedenza, si applica questa tecnica di ottimizzazione, ma in questi, l’output del blocco *normalization* è il risultato finale dell’operazione svolta, in quanto essi non hanno il blocco *round.* Anche in questo caso, il risultato finale si ottiene con un *clock cycle* aggiuntivo, con la conseguente modifica della macchina stati finiti, mostrata in Figura 6.1.5.

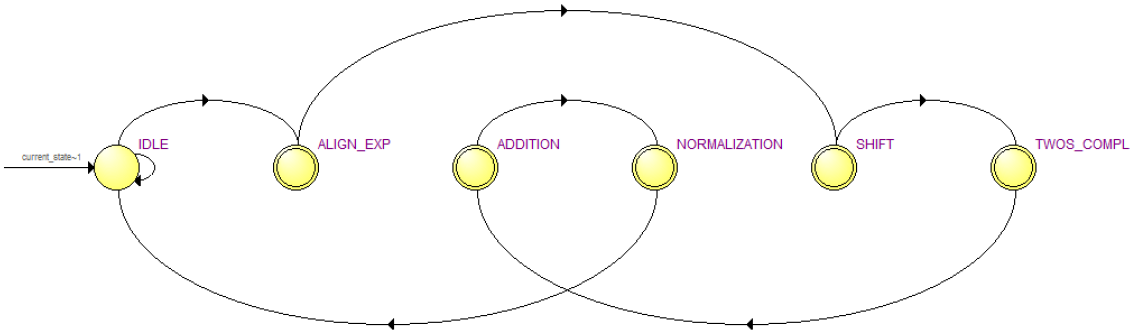


Figura 6.1.5: FSM dei sommatori/sottrattori approssimati

In Figura 6.1.5 l'architettura finale delle unità di somma algebrica *floating-point.*

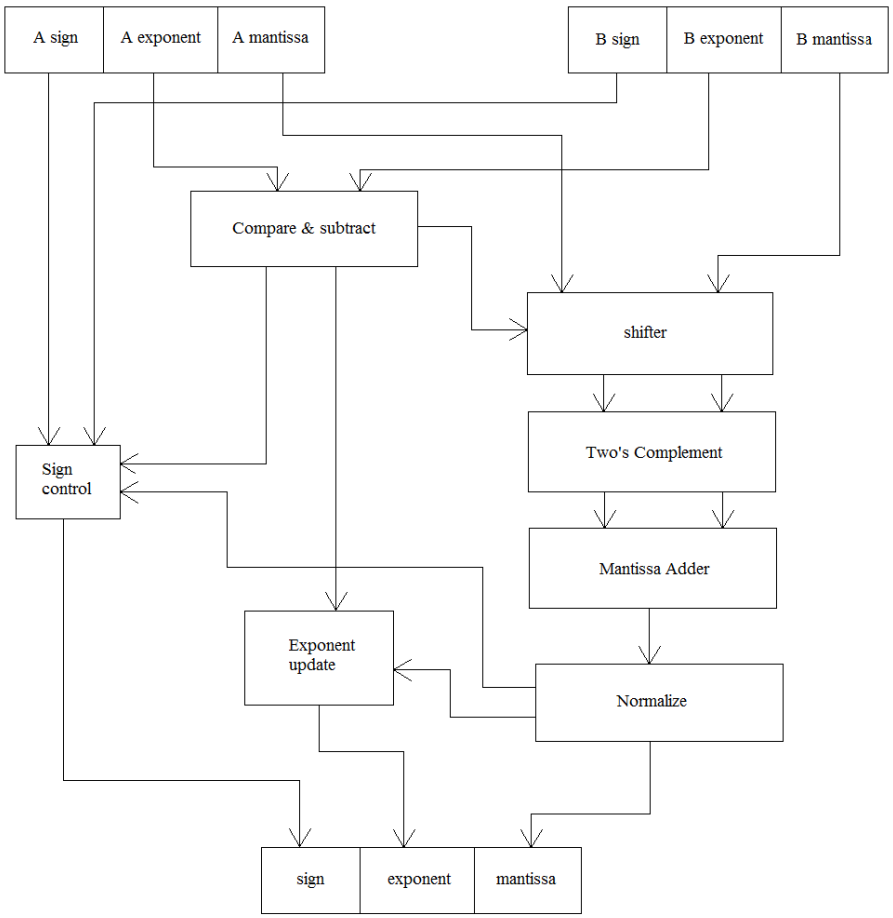


Figura 6.1.5: Architettura unità floating-point approssimate

6.2 Risultati sintesi

A seguito dell'ottimizzazione effettuata, si procede alla sintesi di tutte le unità *floating-point* implementate ed elencate nel passaggio introduttivo. La frequenza massima per l'unità floating-point esatta è di 70 MHz, e tale frequenza si imposta per tutte le unità *floating-point* approssimate.

6.2.1 Report Area e Power

In Tabella 6.2.1.1 si espongono i risultati di area occupata e le rispettive potenze dissipate. L'ordine crescente, è uguale a quello stabilito durante l'analisi dell'errore.

La *Total Power* è il risultato della somma di tre contributi: *Net Switching Power*, *Cell Internal Power* e *Cell Leakage Power*. La *Switching Power* si ottiene mediante una stima probabilistica di tre fattori:

1. *power_default_toggle_rate*: descrive il numero di transizioni della net da '0' a '1' e da '1' a '0' per ns. Il valore di default consigliato da *Synopsys* è 0.1, il quale significa che il nodo *toggla* una volta ogni 10 *clock cycle* relativo a quella net. Questi risultati si ottengono nelle condizioni peggiori, cioè con fattore 0.5.
2. *power_default_static_probability*: rappresenta la percentuale del tempo in cui la net è allo stato logico 1. Il valore di default, e con cui sono stati ottenuti i risultati è 0.5.
3. *power_default_toggle_rate_reference_clock*: indica a quale *clock* si riferiscono le prime due variabili. Si possono associare al *clock* più veloce o al *clock relativo* alla net sotto osservazione. Nei report è *relativo*.

Tabella 6.2.1.1: Area e Power

Unità floating-point	Total Area (μm^2)	Total Power (W)
fp_unit_exact	46127	9.613e-04
RCA12-RcHA11	36298	9.188e-04
RCA12-LOA 11	35911	9.323e-04
RCA12-Trun 11	28099	7.949e-04
RCA11-Trun12	27435	8.046e-04
RCA6-RcHA5-LOA12	34568	8.922e-04
RCA5-RcHA5-LOA5-Trun8	29749	8.308e-04
RCA5-LOA5-Trun13	26293	7.711e-04
LOA-23	34537	9.110e-04
LOA13-Trun10	27995	7.941e-04
LOA8-Trun15	24951	7.529e-04

In Tabella 6.2.1.2, vengono presentati i risparmi percentuali delle unità approssimate rispetto all'unità di riferimento, *fp_unit_exact*:

Tabella 6.2.1.2: Risparmi percentuali di area e power

Unità floating-point	Area	Power
fp_unit_exact	/	/
RCA12-RcHA11	21,3%	4,42%
RCA12-LOA11	22,1%	3,01%
RCA12-Trun11	39,0%	17,3%
RCA11-Trun12	40,5%	16,3%
RCA6-RcHA5-LOA12	25,0%	7,18%
RCA5-RcHA5-LOA5-Trun8	35,5%	13,5%
RCA5-LOA5-Trun13	42,9%	19,8%
LOA-23	25,1%	5,23%
LOA13-Trun10	39,3%	17,4%
LOA8-Trun15	45,9%	21,7%

Come previsto, tutti i sommatori hanno risparmi considerevoli di area. Tra i più performanti vi sono tutti i sommatori in cui è stata applicata la tecnica *Truncated*, ottenendo risparmi quasi del 40%. Da notare, inoltre, le piccole differenze riguardanti i sommatori *RCA12-RcHA11* e *RCA12-LOA11*: sia l'area occupata che la potenza dissipata sono molto simili. Significativo invece, i risultati ottenuti del sommatore *LOA-23*, che presenta risparmi rispettivamente del 25,1% e del 5,23%.

In ultima analisi, si hanno delle piccole discrepanze nelle stime di area e potenza dei sommatori *RCA12-TRUN11* ed *RCA11-TRUN12*: sebbene, il primo sommatore occupi meno area, i risultati di potenza mostrano consumi più bassi. La spiegazione è da ricercare nel metodo usato per fare le analisi di potenza, che, essendo di tipo probabilistico, può portare a risultati in apparenza discordanti tra loro.

Il sintetizzatore, inoltre, fornisce nel *report_area* l'area occupata da ciascun modulo. In Tabella 6.2.1.3, i valori di area occupata dei due moduli più significativi del lavoro svolto.

Tabella 6.2.1.3: Consumi di area dei moduli Addition e Normalization

Area dei moduli (μm^2)	Addition	Normalization
fp_unit_exact	6542.103664	11229.861660
RCA12-RcHA11	4494.020433	8261.524851
RCA12-LOA11	4023.514832	8403.368452
RCA12-Trun11	2414.800833	5047.556429
RCA11-Trun12	2259.118832	5075.233231
RCA6-RcHA5-LOA12	3701.772058	7465.816851
RCA5-RcHA5-LOA5-Trun8	2726.164843	5940.133244
RCA5-LOA5-Trun13	1888.941628	4494.020443
LOA-23	3238.185658	7756.423250
LOA13-Trun10	2027.325634	5140.965645
LOA8-Trun15	1421.895622	3874.752029

Si nota che, nonostante il modulo *Normalization* sia identico in tutti i sommatori approssimati, esso viene sintetizzato per ogni unità approssimata in modo differente, ottimizzando i consumi di area.

Per completezza, in Tabella 6.2.1.4, i valori percentuali di ogni modulo implementato. Tutti i moduli elencati sono uguali tra loro, ad eccezione di *Addition* ed *FSM*, quest'ultima diversa solo nel caso di *fp_unit_exact*.

Tabella 6.2.1.4: Consumi di area percentuali dei moduli implementati

Unità aritmetiche	Comp&sub (%)	Shift (%)	2's Comp. (%)	Addition (%)	Normaliz. (%)	Round (%)	FSM (%)
fp_unit_exact	8.1	24.6	15.7	14.2	24.3	9.7	0.9
RCA12-RcHA11	9.6	31.8	17.8	12.4	22.8	/	1.0
RCA12-LOA11	9.9	31.9	17.9	11.2	23.4	/	1.1
RCA12-Trun11	12.5	41.2	12.8	8.6	18.0	/	1.3
RCA11-Trun12	12.8	41.5	12.2	8.2	18.5	/	1.4
RCA6-RcHA5-LOA12	10.1	33.2	18.5	10.7	21.6	/	1.1
RCA5-RcHA5-LOA5-Trun8	12.0	38.0	14.5	9.2	20.0	/	1.3
RCA5-LOA5-Trun13	13.4	43.3	11.8	7.2	17.1	/	1.4
LOA-23	10.2	33.4	18.7	9.4	22.5	/	1.1

Unità aritmetiche	Comp&sub (%)	Shift (%)	2's Comp. (%)	Addition (%)	Normaliz. (%)	Round (%)	FSM (%)
LOA13-Trun10	12.6	41.4	13.7	7.2	18.4	/	1.3
LOA8-Trun15	14.1	46.0	10.6	5.7	15.5	/	1.5

Il modulo *shift* è quello con consumi di area percentuale maggiore. Considerando il fatto che per i sommatori in cui è stata adottata la tecnica *Truncated*, è possibile ridurre il numero di bit delle mantisse degli operandi di ingresso, *A* e *B*, è plausibile aspettarsi ulteriori riduzioni di area e potenza dall'ottimizzazione di questo modulo.

Anche nei *report_power* sono presenti i consumi percentuali di potenza di ogni blocco implementato. Di seguito, in Tabella 6.2.1.5, i risultati ottenuti.

Tabella 6.2.1.5: Consumi di potenza percentuali dei moduli implementati

Unità aritmetiche	Comp&sub	Shift	2's Comp.	Addition	Normaliz.	Round	FSM
fp_unit_exact	31.8	37.2	6.7	2.8	15.9	0.6	2.4
RCA12-RcHA11	29.1	39.3	6.2	2.2	16.8	/	2.4
RCA12-LOA11	29.2	38.8	6.1	2.1	17.6	/	2.3
RCA12-Trun11	33.3	44.8	3.5	1.6	10.3	/	2.7
RCA11-Trun12	33.6	44.2	3.3	1.5	11.3	/	2.8
RCA6-RcHA5-LOA12	30.7	40.8	6.3	2.2	13.7	/	2.4
RCA5-RcHA5-LOA5-Trun8	33.0	42.3	4.0	1.8	12.8	/	2.6
RCA5-LOA5-Trun13	34.6	45.7	3.1	1.5	9.1	/	2.8
LOA-23	30.7	39.7	6.4	2.1	14.9	/	2.4
LOA13-Trun10	33.8	44.9	3.8	1.6	9.7	/	2.7
LOA8-Trun15	34.8	46.2	2.7	1.3	8.8	/	2.9

I risultati mostrano che i moduli con consumi di potenza maggiore sono *compare_and_sub* e *shift*. Inoltre, il modulo *Round*, presente solo nell'unità aritmetica esatta, ha consumi molto bassi di potenza, apparentemente in contrasto con quanto appreso in letteratura [23]. Infatti, nel valore percentuale in tabella, non viene considerato l'hardware aggiuntivo necessario per eseguire il rounding presente nella maggioranza dei moduli. Di fatto, dal blocco funzionale *shift*, l'implementazione dell'unità di calcolo è pensata per effettuare l'arrotondamento su *3-bit*. Questo significa che ogni blocco necessita di flip-flop aggiuntivi, per la memorizzazione dei dati ed, inoltre, lo stesso sommatore *fixed-point* di mantisse ha 3 bit aggiuntivi da sommare. Inoltre, bisogna considerare anche il numero di bit a cui viene applicato l'arrotondamento: il sommatore/sottrattore esatto è sviluppato, considerando *3-bit* di

arrotondamento, mentre il *rounder* dell'unità analizzata in [23], un moltiplicatore, effettua il *rounding* di *24-bit*.

6.3 Confronto

In ultima analisi, si procede con i confronti tra le varie unità *floating-point* implementate, confrontando i risultati ottenuti dalla sintesi su *Synopsys* e quelli dalla caratterizzazione degli errori su *FPGA*, come visto nel Capitolo 5.

Si confrontano, inoltre, due sommatori con la stessa tecnica di approssimazione, cioè i sommatori approssimati *loa-k*: il primo è il sommatore presentato nell'articolo [21] ed il secondo è quello implementato e discusso in questo elaborato.

I risultati di *MRED* e *MED,* riportati in Tabella 6.3.1, si ottengono ponendo in ingresso dei due operandi numeri casuali *float* compresi tra 2 e 32, come visto nel Paragrafo 5.4.1. I sommatori/sottrattori approssimati, riferiti ai risultati in tabella, sono quelli senza il modulo *rounder*. L'area occupata e la potenza dissipata si calcolano in tecnologia *BCD8SP* (180 nm) di *STMicroelectronics* con frequenza di lavoro impostata a 70 MHz. Come spiegato in precedenza, la potenza si ottiene mediante una stima probabilistica. Per avere migliore leggibilità, in Tabella 6.3.1 vengono riportati i valori percentuali di risparmio di area e potenza, rispetto al riferimento *fp_unit_exact*.

Tabella 6.3.1: Riassunto valori delle unità calcolati a 70MHz

Unità aritmetiche	MED NO ROUND	MRED NO ROUND	AREA (μm^2)	POWER (mW)
fp_unit_exact	0	0	46127	0.9613
RCA12-RcHA11	0,000367086	7,14569E-05	-21,3%	-4,42%
RCA12-LOA11	0,000551218	1,33234E-04	-22,1%	-3,01%
RCA12-Trun11	0,002945098	6,78413E-04	-39,0%	-17,3%
RCA11-Trun12	0,005920879	1,40796E-03	-40,5%	-16,3%
RCA6-RcHA5-LOA12	0,023029422	6,60345E-03	-25,0%	-7,18%
RCA5-RcHA5-LOA5-Trun8	0,045238252	1,10752E-02	-35,5%	-13,5%
RCA5-LOA5-Trun13	0,073700279	1,54739E-02	-42,9%	-19,8%
LOA-23	2,428125715	2,83024E-01	-25,1%	-5,23%
LOA13-Trun10	2,428650483	2,83241E-01	-39,3%	-17,4%
LOA8-Trun15	2,444679621	2,89474E-01	-45,9%	-21,7%

Tra i sommatori più performanti in termini di errore relativo si mettono in risalto i sommatori in cui viene applicata la tecnica *Truncated*: infatti, hanno un tasso molto basso di *MRED*, confrontabile con il migliore, cioè *RCA12-RcHA11*, e al contempo risparmi di area e potenza

nettamente superiori. Infatti, se l'unità *RCA12-RcHA11* produce un risparmio di area del 21,3% e di potenza del 4,42%, il sommatore/sottrattore *RCA12-Trun11* ha risparmi rispettivamente del 39% e del 17,3%, quindi, quasi del doppio in area e più del quadruplo in potenza. Inoltre, confrontando questa unità con quelle meno performanti, si notano consumi di area e potenza simili, o, in alcuni casi migliori, e tassi di *MRED* nettamente più bassi, anche di tre ordini di grandezza, come ad esempio con le unità *LOA-k*.

Le unità *ibride*, cioè quelle in cui sono applicate più tecniche di approssimazione contemporaneamente, possono essere considerate come un buon *trade-off* tra precisione e consumi di area-potenza. Infatti, hanno tassi di *MRED* intermedi: circa due ordini di grandezza inferiori ai più performanti e altrettanti dai meno performanti.

Infine, in Tabella 6.3.2, il confronto, solo in termini percentuali, tra l'unità *LOA-k* sviluppata nell'elaborato e quella implementata nell'articolo [21]. Da notare che l'unità aritmetica implementata è in grado di calcolare somme e differenze, al contrario di quanto accade nell'articolo [21], che elabora solo somme.

Tabella 6.3.2: Confronto tra l'unità implementata LOA-23 e l'unità presente nell'articolo [3]

FP Adders	Power (mW)	Area (μm^2)
LOA-23 Articolo [21]	29,98%	30,15%
LOA-23	5,23%	25,1%

I dati percentuali, in tabella, sono i risparmi in termini di area e potenza rispetto alle proprie unità *floating-point* esatte di appartenenza. Infatti, il sommatore dell'articolo è sintetizzato su tecnologia Faraday 65nm, mentre, il sommatore del lavoro di tesi, in tecnologia *BCD8SP* 180nm. Se i risultati in termini di risparmio di area sono dello stesso ordine e confrontabili, differente è la situazione riguardo la potenza dissipata. Le cause, anche in questo caso, sono riconducibili al tipo di design implementato: l'unità progettata, infatti, è anche in grado di sottrarre due numeri, e questo si traduce in hardware aggiuntivo principalmente nei blocchi chiamati *two's_complement*, assente nel sommatore dell'articolo [21], e *normalization*, che risulterà nettamente più grande in termini di area e di consumo di potenza del sommatore di [21]. Infatti, il blocco *normalization*, oltre ad eseguire i classici *shift* post-somma per normalizzare il numero, nel caso in cui il risultato derivante dalla somma delle mantisse sia negativo, esegue nuovamente l'operazione di complemento a due, consumando potenza e occupando area.

Un altro fattore, da non sottovalutare durante l'analisi della potenza, è l'implementazione del *rounder*: infatti, il rounder del sommatore/sottrattore esatto implementato durante il lavoro di tesi è pensato per agire solo su 3 bit, e quindi la predisposizione nei vari blocchi funzionali è progettata per tale scopo. Ciò si traduce in un risparmio più lieve di potenza dissipata, in quanto si hanno già delle differenze significative nel sommatore/sottrattore esatto.

7. Conclusioni

I dati raccolti evidenziano risparmi considerevoli di area e potenza dissipata. In particolare, per la totalità delle unità aritmetiche implementate, i risparmi di area sono superiori al 20%, arrivando in alcuni casi al 40%. I consumi di potenza, anch'essi ridotti per la totalità delle unità progettate, hanno valori percentuali di risparmio meno accentuati. Il sommatore/ sottrattore *floating-point* con maggiore potenza dissipata è *RCA12-LOA11*, con *-3,01%*, viceversa, l'unità *LOA8-TRUN15* è quella con minore dissipazione, -21,7%.

Uno dei risultati più rilevanti della ricerca, riguarda le piccole differenze in termini di area e potenza tra i sommatori implementati con la tecnica *RcHA*, rispetto ai sommatori *LOA*, a parità di bit: questo risultato esalta le proprietà del sommatore *RcHA*, infatti, a scapito di un aumento di area del 0.8%, con potenza dissipata pressoché invariata, ha tassi di *MRED* più bassi del sommatore *LOA* del 86%. Ciò lascia presagire che l'applicazione della tecnica *RcHA*, può rappresentare una valida alternativa alla tecnica *LOA.* A supporto di questa tesi, il confronto tra i risultati dell'unità *RCA12-RCHA11* e *LOA-23*: sebbene le due unità sono confrontabili in termini di area e potenza, l'unità *RCA12-RCHA11* è nettamente migliore in termini di precisione di calcolo, avendo l'*MRED* quattro ordini di grandezza inferiore a quello del *LOA-23*.

Hanno grande rilevanza i risultati dell'implementazione delle tecnica *TrunA*. Infatti, tra tutte le tecniche è quella che riesce a garantire risparmi di area e potenza migliori, mantenendo tassi di *MRED* accettabili, come per il sommatore *RCA12-TRUN11* che è nell'ordine di 10^{-4}. Questi risultati evidenziano l'importanza che hanno gli *MSB* nel calcolo del risultato finale in termini di precisione.

I sommatori algebrici *ibridi* hanno un degrado del risultato finale intermedio, infatti l'*MRED* è dell'ordine di 10^{-3} o 10^{-2}. I consumi di area e potenza, sono dipendenti dalla tecniche di *approximate computing* attuate: si hanno consumi ridotti per i sommatori *RCA5-LOA5-Trun13,* con il -42,9% di area e -19,8% di potenza, e *RCA5-RcHA5-LOA5-Trun8*, con -35,5% e 13,5%, mentre sono più accentuati per *RCA6-RcHA5-LOA12*, con il -25% e 7,18%.

Al fine di ottenere un valore assoluto di *MRED*, che caratterizzi ogni sommatore/sottrattore *floating-point* implementato, viene calcolata la media aritmetica dei vari *MRED*, Tabella 7.1, ottenuti dai test effettuati (Capitolo 5). Si ricordano per completezza i tre tipi di test effettuati:

- Test per valutare il comportamento e il peso del *rounding* delle unità *floating-point* con numeri maggiori di 1;
- Test per valutare il comportamento dell'unità con numeri minori di zero;
- Test usando una rampa crescente o decrescente .

Tabella 7.1: Valori assoluti MRED

Unità aritmetiche	MRED
RCA12-RcHA11	2,4169E-05
RCA12-LOA11	4,3075E-05
RCA12-Trun11	2,2625E-04
RCA11-Trun12	6,0053E-04
RCA6-RcHA5-LOA12	2,4183E-03
RCA5-RcHA5-LOA5-Trun8	4,3544E-03
RCA5-LOA5-Trun13	6,2784E-03
LOA-23	1,0492E-01
LOA13-Trun10	1,0659E-01
LOA8-Trun15	1,0934E-01

Infine, hanno grande rilevanza i dati percentuali relativi ai consumi di area e potenza dissipata ottenuti su tecnologia *BCD8SP* in fase di sintesi, presentati nel Capitolo 6, Tabelle 6.2.1.4 e 6.2.1.5, in quanto offrono possibili spunti di ottimizzazione: infatti, nelle unità approssimate hanno un grande impatto i blocchi gerarchici *shifter* e *compare_&_subtract*. Il blocco funzionale *shift* supera, in tutte le unità approssimate, il 30% dell'area totale occupata e il 38% della potenza dissipata totale. Il blocco *compare_&_subtract* ha alti consumi di potenza, superiori in molti casi al 30%, ma di contro, ha tassi piuttosto bassi di area occupata, non superiori al 14%. I dati sono molto interessanti perché i due blocchi sono pensati e progettati per l'unità *floating-point* esatta e riutilizzati, senza alcuna modifica, per le unità approssimate. Quindi, questi hanno la predisposizione a supportare l'operazione di arrotondamento.

Un altro tipo di ottimizzazione riguarda la natura delle unità progettate: si ricorda, che le unità *floating-point* implementate sono in grado di calcolare sia somme che differenze. Anche questa scelta di *design* impatta negativamente nei consumi di area e potenza: di fatto, il blocco chiamato *two's_complement*, può essere eliminato e grandi semplificazioni possono essere fatte nel blocco *normalization.* Il blocco *two's_complement* ha consumi percentuali di area che superano il 10%, arrivando in alcuni casi anche al 18%, e di potenza tra il 2,7% e il 6,4%. I consumi di area del blocco *normalization* sono compresi tra il 15,5% e il 24.3%, e di potenza, tra 8,8% e 17,6%. Riassumendo, sarà possibile ottimizzare l'area e il consumo di potenza a scapito di perdere la funzione di sottrazione.

Bibliografia

[1] *Altera, Embedded Design Handbook, Altera Co.*

[2] *Altera, Nios II Processor Reference Guide, Altera Co.*

[3] *Altera, Nios II Gen 2 Software Developer's Handobook, Altera Co.*

[4] *Altera, Embedded Peripherals IP User Guide, Altera Co.*

[5] *Chandrahash Patel. 2014. Ripple Carry Adder Design Using Universal Logic Gates. Research Journal of Engineering Sciences, Vol. 3(11), 1-5, November 2014.*

[6] *IEEE Computer Society, IEEE Standard for Floating-Point Arithmetic. IEEE (New York), (Revision of IEEE Std 754-1985).*

[7] *Jinghang Liang, Jie Han, and F. Lombardi. 2013. New Metrics for the Reliability of Approximate and Probabilistic Adders. IEEE Trans. Comput. 62, 9 (September 2013), 1760–1771.*

[8] *H. Jiang, C. Liu, L. Liu, F. Lombardi, J. Han. A Review, Classification and Comparative Evaluation of Approximate Arithmetic Circuits. ACM Journal on Emerging Technologies in Computing Systems, Vol. 13, No. 4, Article 60, Pub. date: July 2017.*

[9] *K. Verma, P. Brisk, and P. Ienne. Variable latency speculative addition: A new paradigm for arithmetic circuit design. In DATE, pages 1250-1255, 2008.*

[10] *G. Naveen Balaji, S. Deni Johnson, S. Giridharan and R. Aswanth. Approximate Adders for Digital Signal Processing (DSP) Applicatons - A Relative Study. IEEE International Conference on Science, Technology, Engineering and Management (ICSTEM'17). March 2017.*

[11] *D. Mohapatra, V. K. Chippa, A. Raghunathan and K. Roy. Design of voltage-scalable meta-functions for approximate computing. 2011 Design, Automation \& Test in Europe. 14-18 March 2011.*

[12] *N. Zhu, W. L. Goh, and K. S. Yeo. An Enhanced Low-Power High-Speed Adder for Error-Tolerant Application. Proceedings of the 2009 12th International Symposium on Integrated Circuits. 14-16 Dec. 2009.*

[13] *B. Kahng and S. Kang. Accuracy-Configurable Adder for Approximate Arithmetic Designs. In DAC, pages 820-825, June 2012.*

[14] K. Du, P. Varman and K. Mohanram. High performance reliable variable latency carry select addition. 2012 Design, Automation \& Test in Europe Conference \& Exhibition. Pages 1257-1262. 12-16 March 2012.

[15] *Honglan Jiang, , Cong Liu, Naman Maheshwari, Pilani, Rajasthan, Jie Han. "A Comparative Study of Approximate Adders and Multipliers"*

[16] *Y. Kim, Y. Zhang, and P. Li. An energyefficient approximate adder with carry skip for error resilient neuromorphicvlsi systems. In ICCAD, pages 130-137, 2013.*

[17] *I. C. Lin, Y. M. Yang and C. C. Lin. High-Performance Low-Power Carry Speculative Addition With Variable Latency. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, pages 1591-1603, Sept. 2015.*

[18] *Li Li and Hai Zhou. On Error Modeling and Analysis of Approximate Adders. In ICCAD. 511–518, 2014.*

[19] *J. Hu and W. Qian. A New Approximate Adder with Low Relative Error and Correct Sign. In DATE, pages 1449–1454, 2015.*

[20] *H. R. Mahdiani, A. Ahmadi, S. M. Fakhraie, and C. Lucas. Bio-Inspired Imprecise Computational Blocks for Efficient VLSI Implementation of Soft - Computing Applications. IEEE Trans. Circuits Syst., Vol. 57, NO. 4, pages: 850-862, April 2010.*

[21] *W. Liu, L. Chen, C. Wang, M. O'Neill, F. Lombardi. Inexact Floating-Point Adder for Dynamic Image Processing. 14th IEEE International Conference on Nanotechnology, Aug. 2014, pp.239-243 [Rivista Peer Reviewed].*

[22] *V. Camus, J. Schlachter, C. Enz, M.Gautschi, F. K. Gurkaynak. Approximate 32-bit Floating-point Unit Design with 53% Power-area Product Reduction. IEEE (New York), 2016.*

[23] *J. Y. Tong, D. Nagle, and R. Rutenbar, "Reducing Power by Optimizing the Necessary Precision/Range of Floating-Point Arithmetic", IEEE Trans. Very Large Scale Integr. (VLSI) Syst., vol. 8, pp. 273-286, 2000.*

Appendice

A. soc_fp_exact.vhd

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

entity soc_fp_exact is
        port(
        clock: IN std_logic;
        reset: IN std_logic
        );
end entity soc_fp_exact;

architecture struct of soc_fp_exact is
   component fp_unit is
        port(
          clock, reset            : IN  std_logic;
          start                   : IN  std_logic;
          A_exp, B_exp            : IN  std_logic_vector(7 downto 0);
          A_sign,B_sign           : IN  std_logic;
          A_mantissa,B_mantissa   : IN  std_logic_vector(22 downto 0);
          SUM_exp_upd_ro          : OUT std_logic_vector(7 downto 0);
          SUM_mantissa_ro         : OUT std_logic_vector(22 downto 0);
          SUM_sign                : OUT std_logic
        );
   end component fp_unit;

   component nios_fp_exact is
       port(
         clk_clk                                : IN  std_logic                     := 'X';
         reset_reset_n                          : IN  std_logic                     := 'X';
         start_external_connection_export       : OUT std_logic;
         a_sign_external_connection_export      : OUT std_logic;
         b_sign_external_connection_export      : OUT std_logic;
         a_exp_external_connection_export       : OUT std_logic_vector(7 downto 0);
         b_exp_external_connection_export       : OUT std_logic_vector(7 downto 0);
         a_mantissa_external_connection_export  : OUT std_logic_vector(22 downto 0);
         b_mantissa_external_connection_export  : OUT std_logic_vector(22 downto 0);
         sum_sign_external_connection_export    : IN  std_logic     := 'X';
         sum_exp_external_connection_export: IN std_logic_vector(7 downto 0):=(others => 'X');
         sum_mantissa_external_connection_export: IN  std_logic_vector(22 downto 0):=
(others=>'X')
      );
    end component nios_fp_exact;

    signal A_exp_in, B_exp_in, SUM_exp_in            : std_logic_vector(7 downto 0);
    signal A_mantissa_in,B_mantissa_in,SUM_mantissa_in: std_logic_vector(22 downto 0);
    signal A_sign_in, B_sign_in, SUM_sign_in, start_in: std_logic;
    begin

    u0 : component nios_fp_exact
        port map(
            clk_clk                                => clock,
            reset_reset_n                          => reset,
            start_external_connection_export       => start_in,
            a_sign_external_connection_export      => A_sign_in,
            b_sign_external_connection_export      => B_sign_in,
            a_exp_external_connection_export       => A_exp_in,
            b_exp_external_connection_export       => B_exp_in,
            a_mantissa_external_connection_export  => A_mantissa_in,
            b_mantissa_external_connection_export  => B_mantissa_in,
            sum_sign_external_connection_export    => SUM_sign_in,
            sum_exp_external_connection_export     => SUM_exp_in,
            sum_mantissa_external_connection_export => SUM_mantissa_in
        );
     u1 : component fp_unit
        port map(
clock, reset, start_in, A_exp_in, B_exp_in, A_Sign_in, B_sign_in, A_mantissa_in,
B_mantissa_in, SUM_exp_in, SUM_mantissa_in, SUM_sign_in);

end architecture struct;
```

B. Funzioni di invio dei valori A e B all'unità *floating-point*

```
void send_sign_AB (alt_u8 A_sign, alt_u8 B_sign){
       IOWR_ALTERA_AVALON_PIO_DATA(A_SIGN_BASE, A_sign);
       IOWR_ALTERA_AVALON_PIO_DATA(B_SIGN_BASE, B_sign);
}

void send_exp_AB (alt_u8 A_exp, alt_u8 B_exp){
       IOWR_ALTERA_AVALON_PIO_DATA(A_EXP_BASE,A_exp);
       IOWR_ALTERA_AVALON_PIO_DATA(B_EXP_BASE,B_exp);
}

void send_mantissa_AB(alt_u32 A_mantissa, alt_u32 B_mantissa){
       IOWR_ALTERA_AVALON_PIO_DATA(A_MANTISSA_BASE,A_mantissa);
       IOWR_ALTERA_AVALON_PIO_DATA(B_MANTISSA_BASE,B_mantissa);
}
```