Sommatore Floating Point IEEE 754

Prova Finale Reti Logiche

Realizzato da:

KEVIN GUGLIELMETTI
RAFAEL MOSCA

Laurea in Ingegneria Informatica
POLITECNICO DI MILANO

Progetto realizzato sotto la supervisione del
PROF. CARLO BRANDOLESE.

LUGLIO 2018

INDICE

Page

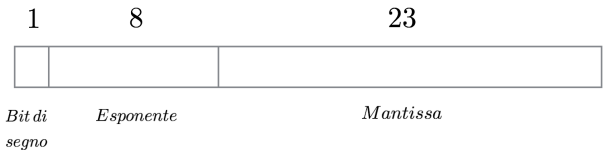
CAPITOLO 1

LO STANDARD IEEE 754

Lo standard IEEE per il calcolo in virgola mobile, chiamato altresì IEEE 754, è ad oggi lo standard più diffuso per la rappresentazione dei numeri in virgola mobile, o *floating point*. Questo standard ammette diversi formati per la rappresentazione dei numeri in virgola mobile, precisione singola (32 bit), precisione doppia (64 bit), e le corrispettive versioni estese ($\geq 43bit$), e ($\geq 79bit$).
Su questo documento ci concentriamo sulla rappresentazione a *precisione singola* a 32 bit che consente di rappresentare numeri nell'intervallo che va da $1,18 \cdot 10^{-38}$ a $3,40 \cdot 10^{38}$ e i casi speciali ± 0, $\pm\infty$, NaN ("not a number") e i numeri denormalizzati.

1.1 Rappresentazione a precisione singola

Un numero nello standard IEEE è rappresentato, indipendentemente dal tipo di precisione, con tre campi: bit di segno, esponente e mantissa. La precisione aumenta il range dei numeri rappresentabili aumentando il numero di bit dei campi esponente e mantissa. Per la precisione singola un numero è composto nel seguente modo:



① Il primo bit che è il bit di segno s, assume il valore 0 se il numero rappresentato è positivo, 1 se negativo.

② Il secondo campo è l'esponente che è codificato nel seguente modo: sia E il vero esponente e sia e l'esponente rappresentato negli 8 bit, allora:

$$E = e - bias$$

dove $bias = 127$ per numeri a precisione singola. Questa codifica permette che il valore vero dell'esponente vari nell'intervallo $[-126, 127]$.

③ Il terzo campo è la mantissa, molte volte riferito come significando e rappresentata successivamente con M. Un numero (normalizzato) può essere espresso come

$$(-1)^s \cdot 1.M \cdot 2^E$$

Si noti che l'uno davanti la mantissa M è implicito e dunque non viene rappresentato nella codifica IEEE754.

1.1.1 I numeri e casi speciali a precisione singola

Com'era stato accennato prima, lo standard prevede numeri e casi speciali nella rappresentazione[1], questi casi sono rappresentati nella tabella sottostante.

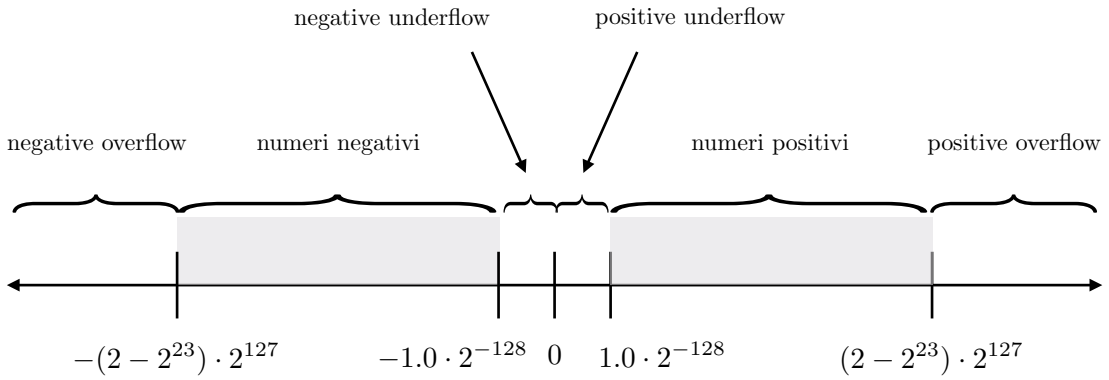
Categoria	Esponente	Mantissa
Zeri	0_{10}	0_{10}
Numeri denormalizzati	0_{10}	non zero
Numeri normalizzati	$1_{10} - 254_{10}$	qualunque
Infiniti	255_{10}	0_{10}
NaN	255_{10}	non zero

1.1.2 Arrotondamenti

Lo standard IEEE754 riesce a rappresentare un numero finito di numeri, per questo motivo prevede degli arrotondamenti nei casi in cui un numero non è rappresentabile dallo standard:

Nome arrotondamento	Comportamento
Arrotondamento a $+\infty$	si arrotonda al numero più piccolo che è $\geq$ al numero.
Arrotondamento a $-\infty$	si arrotonda al numero più grande che è $\leq$ al numero.
Arrotondamento a 0	Arrotondamento a $-\infty$ se $x \geq 0$ e a $+\infty$ se $x \leq 0$.
Arrotondamento al più vicino	Numero standard più vicino (pari in caso di uguale distanza).

1.1.3 Range di valori



In questa figura si evidenziano il range di valori che copre lo standard così come i range non coperti. Da questa figura risulta molto facile capire l'arrotondamento di *default* ovvero l'arrotondamento al più vicino, ed è riassunto di seguito:

$$\begin{aligned}
&\text{Negative overflow} && \rightarrow && -\infty \\
&\text{Positive overflow} && \rightarrow && +\infty \\
&\text{Negative underflow} && \rightarrow && -denormalizzato && \rightarrow && 0 \\
&\text{Positive underflow} && \rightarrow && +denormalizzato && \rightarrow && 0
\end{aligned}$$

1.2 Somma di numeri a precisione singola

L'algoritmo per sommare due numeri in virgola mobile risulta molto semplice ed è lo stesso per precisione singola che per precisione doppia. Supponendo di avere due numeri normalizzati in ingresso X e Y, i passi da seguire sono:

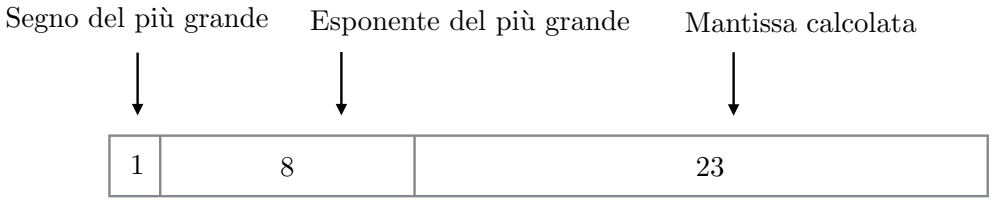
1. Realizzare controlli preventivi in modo che se un operando è zero, allora di conseguenza il risultato è l'altro operando.
2. Cambiare di segno il secondo operando se l'operazione è di sottrazione.
3. Identificare il numero più grande.
    a) Effettuare (se necessario) uno *swap* tra gli operandi per portare il numero più grande come primo operando.
4. Shiftare a destra la mantissa del numero più piccolo un numero di posizioni pari a la differenza tra l'esponente più grande e l'esponente più piccolo.

5. Realizzare la somma/sottrazione di mantisse a seconda dell'operazione immessa.

   a) Se nell'operazione precedente si verifica overflow della mantissa, si realizza uno shift a destra e si aumenta l'esponente di uno.

   b) Se all'aumentare di uno l'esponente si verifica un overflow dell'esponente, allora il risultato è infinito.

6. Calcolo del risultato preliminare come:



   a) Se il risultato non è normalizzato si realizza uno shift a sinistra e si decrementa l'esponente di uno finché non diventa normalizzato.

   b) Se al decrementare l'esponente si verifica un underflow dell'esponente, allora il risultato è subnormale.

7. Arrotondamento del risultato.

1.2.1 Gestione dei casi speciali

In caso di somma lo standard definisce i seguenti casi speciali (si considerano speciali nella tabella Zero, Infinito, e NaN):

Operando 1	Operando 2		Risultato
Non speciale	Zero	=	Operando 1
Zero	Non speciale	=	Operando 2
NaN	−	=	NaN
−	NaN	=	NaN
± Infinito	∓ Infinito	=	NaN
± Infinito	± Infinito	=	± Infinito
± Infinito	Non speciale	=	± Infinito
Non speciale	± Infinito	=	± Infinito

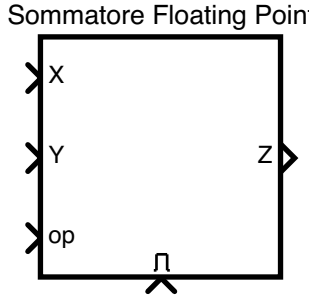
CAPITOLO 2

STRUTTURA DEL SOMMATORE

In questa sezione verranno descritti a grandi linee i moduli con i cui è realizzato il sommatore. Si è deciso di realizzare il sommatore floating point in modalità pipelined a tre stadi: fase pre-somma, fase somma e fase correttiva e anche in modalità puramente combinatoria. Entrambe le implementazioni usano gli stessi moduli e hanno come unica differenza che nell'implementazione combinatoria il sommatore è il collegamento a cascata dei moduli della pipeline, mentre nella pipeline sono questi stessi moduli separati da opportuni registri. Nelle seguenti pagine verranno presentate le fasi e i diversi moduli che le compongono così come le corrispettive *entity* in VHDL. A titolo di esempio, nel caso del modulo che identifica i casi, verrà mostrata in aggiunta la struttura con cui si è pensato il modulo e la relativa implementazione in VHDL.

2.1 Top level

Il sommatore a top level prevede un operando X e un operando Y (si suppongono i numeri già normalizzati se ammettono una tale rappresentazione), l'operazione da realizzare (0 somma, 1 sottrazione), e il clock per gestire la pipeline.




```
entity FPAdder is
   port(
      X    : in     std_logic_vector (31 downto 0);
      Y    : in     std_logic_vector (31 downto 0);
      op   : in     std_logic;
      clk  : in     std_logic;
      Z    : out    std_logic_vector (31 downto 0)
   );
end FPAdder ;
```

2.2 Fase pre-somma

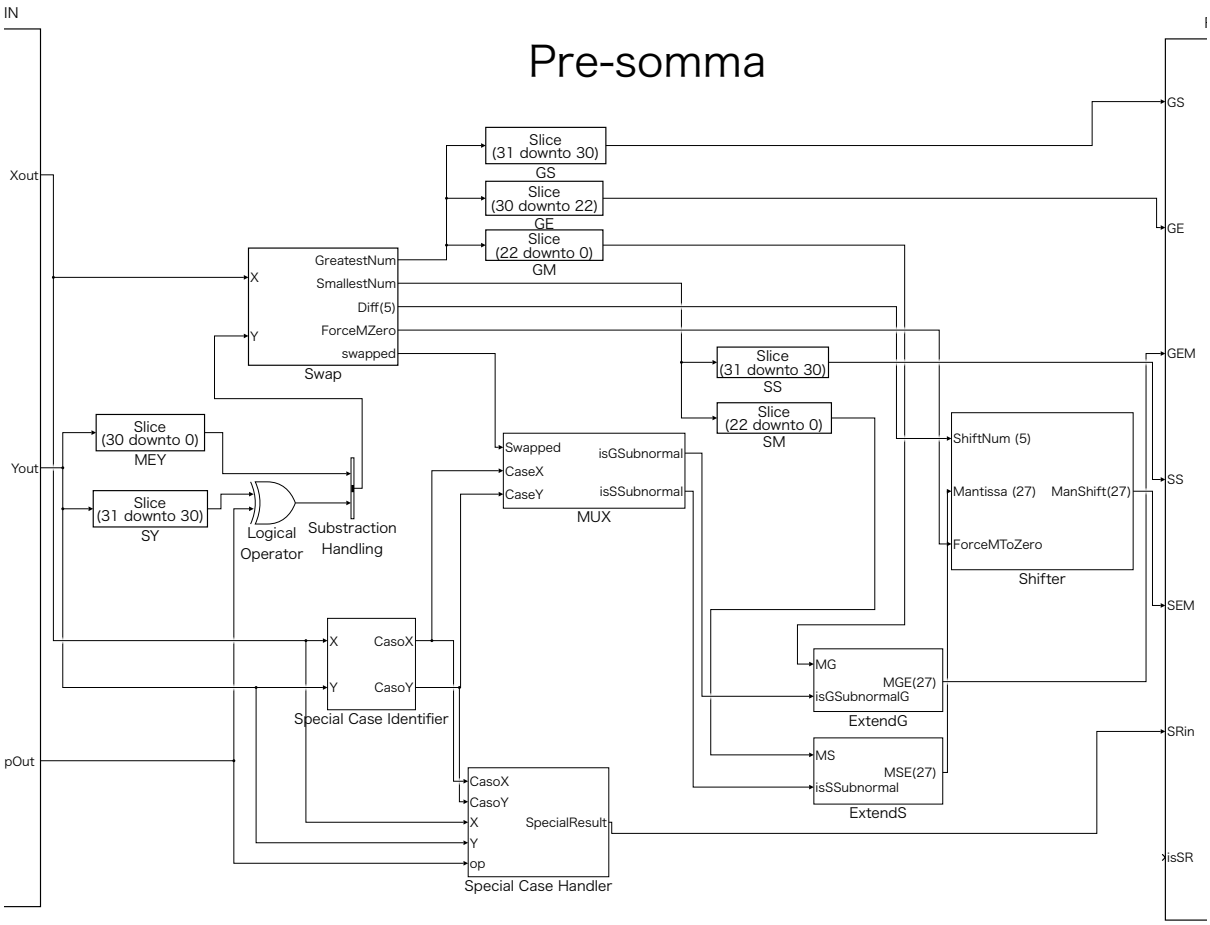
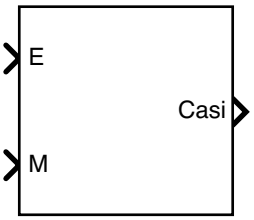


Figura 2.1: E possibile zoomare digitalmente per osservare i singoli moduli e i singoli segnali

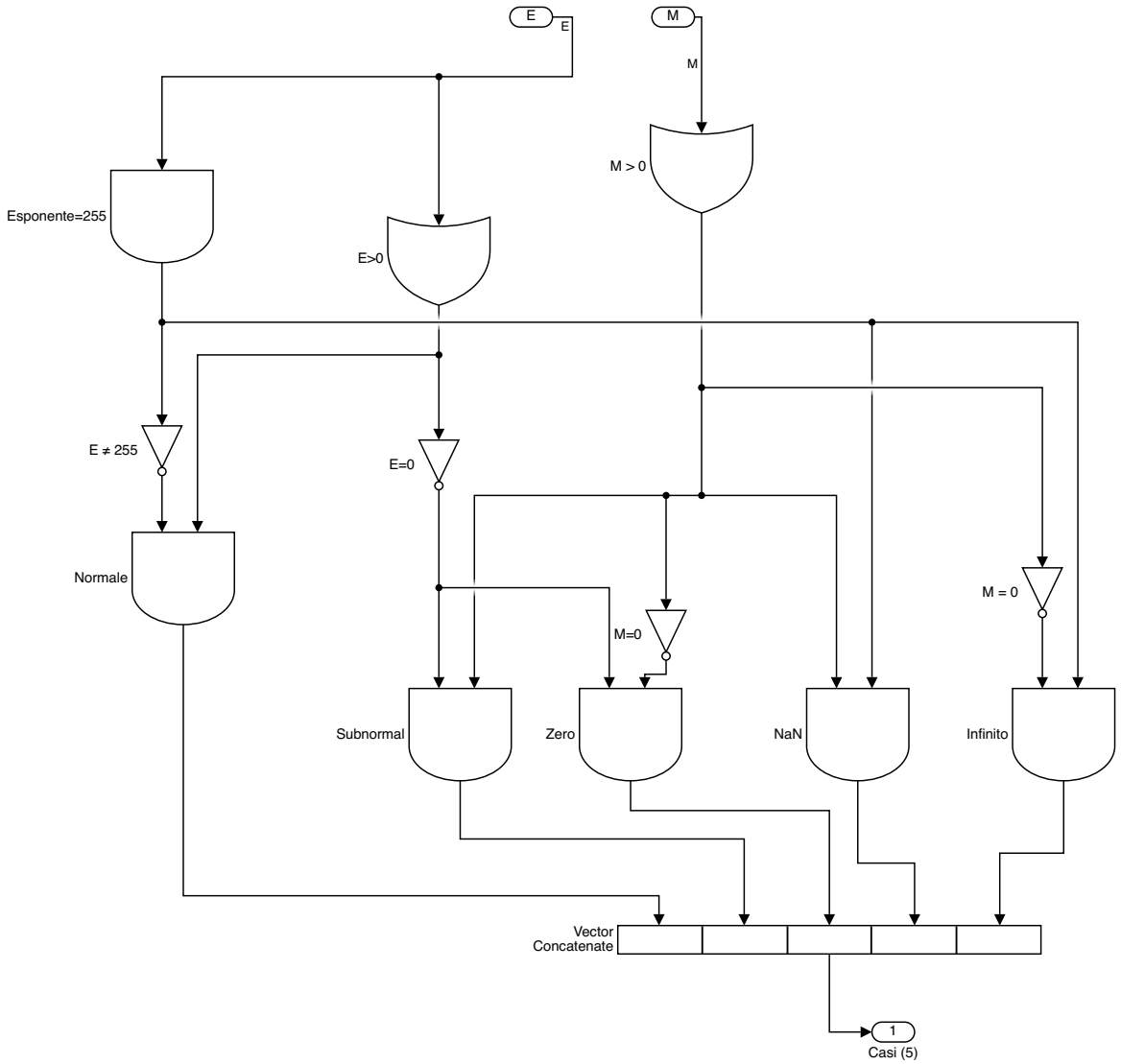
La fase pre-somma è costituita da 2 moduli che identificano il caso di ogni singolo operando, un modulo che gestisce i risultati nei casi speciali, un modulo che scambia gli operandi in modo tale da avere quello più grande come primo operando, un modulo che estende gli operandi con il bit implicito e i bit di servizio per l'arrotondamento e uno shifter combinatorio a destra per la mantissa più piccola.

2.2.1 Identificazione casi



Identificazione casi

```
entity SCIdentifier is
   port(
      E    : in     std_logic_vector ( 7 downto 0);
      M    : in     std_logic_vector (22 downto 0);
      Casi : out    std_logic_vector ( 4 downto 0)
   );
end SCIdentifier ;
```



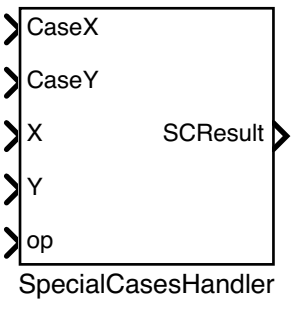
```
architecture structural of SCIdentifier is
    signal  eAlmenoUno  : std_logic;
    signal  eTuttiUno   : std_logic;
    signal  eZero       : std_logic;
    signal  mAlmenoUno  : std_logic;

begin
    mAlmenoUno <= '0' when M = "00000000000000000000000" else '1';
    eAlmenoUno <= '0' when E = "00000000" else '1';
    eTuttiUno  <= '1' when E = "11111111" else '0';
    Casi(4)    <= eAlmenoUno and not(eTuttiUno);         --isNormal
    Casi(3)    <= mAlmenoUno and not(eAlmenoUno);        --isSubnormal
    Casi(2)    <= not(mAlmenoUno) and not(eAlmenoUno);   --isZero
    Casi(1)    <= mAlmenoUno and eTuttiUno;              --isNaN
    Casi(0)    <= not(mAlmenoUno) and eTuttiUno;         --isInfinity
end structural;
```

Come si può apprezzare dallo schema e dal codice, si è deciso di codificare i casi in codifica One-Hot; le associazioni di ogni caso alla codifica sono riassunte nella tabella accanto.

Normale	10000
Subnormale	01000
Zero	00100
NaN	00010
Infinito	00001

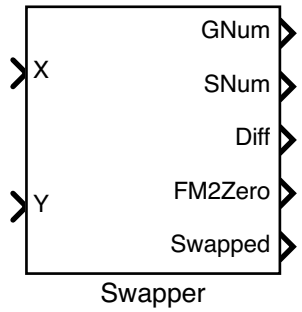
2.2.2 Gestore di casi speciali




```
entity SCHandler is
   port(
      op       : in   std_logic;
      CaseX    : in   std_logic_vector (2 downto 0);
      CaseY    : in   std_logic_vector (2 downto 0);
      X        : in   std_logic_vector (31 downto 0);
      Y        : in   std_logic_vector (31 downto 0);
      SCResult : out  std_logic_vector (31 downto 0)
   );
end SCHandler ;
```

Il gestore dei casi speciali riceve lo slice dei tre bit meno significativi del vettore in output al modulo che identifica i casi per ogni singolo operando e, di conseguenza, riceve tutti zero se si tratta un caso normale/subnormale oppure un uno in posizione variabile a seconda del caso speciale. I risultato *SCResult* è calcolato conforme allo standard specificato in sezione 1.2.1.

2.2.3 Swapper




```
entity Swapper is
   port(
      X       : in    std_logic_vector (31 downto 0);
      Y       : in    std_logic_vector (31 downto 0);
      GNum    : out   std_logic_vector (31 downto 0);
      SNum    : out   std_logic_vector (31 downto 0);
      Diff    : out   std_logic_vector ( 4 downto 0);
      FM2Zero : out   std_logic;
      swapped : out   std_logic
   );
end Swapper ;
```

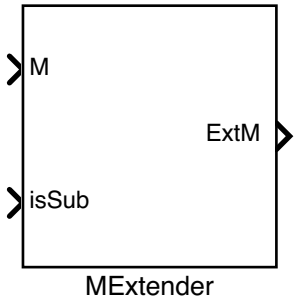
Lo swapper è un modulo che riceve entrambi gli operandi, effettua confronti di mantisse e di esponenti, e realizza o meno uno *swap*, ovvero uno scambio tra gli operandi, in modo da poter gestire come primo operando sempre il numero più grande. Se uno scambio ha luogo, il modulo segnala l'avvenimento di tale scambio attraverso il segnale *swapped* in modo che i moduli successivi possano risalire ai casi che erano stati identificati in X e Y.
I comparatori vengono realizzati come dei sottrattori che calcolano la differenza dei due operandi. Nel caso del comparatore di esponenti, esso ha un duplice scopo: oltre a dire se l'esponente del primo operando è maggiore o meno del secondo e viceversa, il comparatore da come risultato la differenza degli esponenti, sulla quale verrà fatto uno slice a 5 bit che determinerà di quanti bit si dovrà shiftare la mantissa del numero più piccolo. Prendendo uno slice di 5 bit si perdono però delle informazioni, infatti è possibile che la differenza di esponenti sia ad esempio 64 e prendendo gli ultimi 5 bit venga data l'indicazione allo shifter di non shiftare la mantissa; è per questo motivo che il modulo prevede un'altra uscita chiamata *FM2Zero (Force Mantissa to Zero)* che indicherà ai moduli successivi di imporre la mantissa del numero più piccolo a zero, dato che oltre i 27 bit (cfr.2.2.5) di shift la mantissa originale viene persa e si hanno tutti zero.

Si noti che dato che non si sa ancora se il primo operando è maggiore o meno del secondo, la differenza è corretta solo nei casi in cui il primo operando è maggiore del secondo (primo bit della somma a 0). Per chiarire il concetto viene riportato di seguito un esempio.

15	0 1111	15	0 1111	4	0 0100
4	0 0100	4_{C2}	1 1100	15_{C2}	1 0001
			0 1011		1 0101

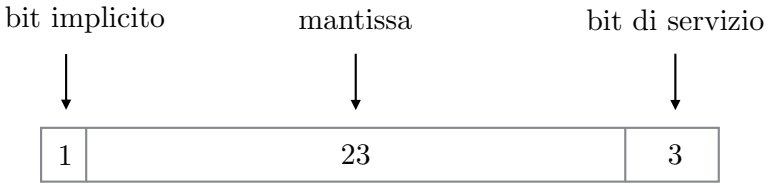
Nei casi in cui il secondo operando è maggiore del primo (primo bit della somma a 1) per ottenere la differenza in modo giusto basta fare il complemento a due del risultato della somma.

2.2.4 Estensione Mantissa




```
entity MExtender is
   port(
      isSub   : in   std_logic;
      M       : in   std_logic_vector (22 downto 0);
      ExtM    : out  std_logic_vector (26 downto 0)
   );
end MExtender ;
```

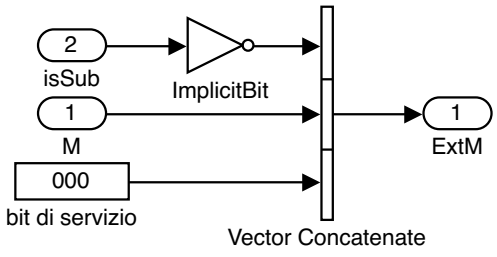
Questo banalissimo modulo ha come scopo trasformare la mantissa di 23 bit in una mantissa estesa a 27 bit.



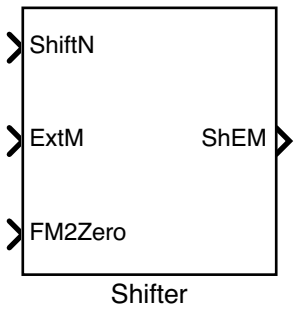
I 27 bit sono divisi in tre sezioni:

- Il *bit implicito* che è pari a 1 se il numero in ingresso è normalizzato mentre è 0 se il numero è denormalizzato.
- La *mantissa* che è quella in ingresso.
- I tre *bit di servizio* chiamati G (*guard bit*), R (*round bit*) e S (*sticky bit*) che servono a gestire gli arrotondamenti a valle della somma, e valgono "000" di default.

```
architecture RTL of MExtender is
begin
    ExtM <= not(isSub) & M & "000";
end structural;
```



2.2.5 Shifter




```
entity Shifter is
   port(
      FM2Zero : in  std_logic;
      ShiftN  : in  std_logic_vector ( 4 downto 0);
      ExtM    : in  std_logic_vector (26 downto 0);
      ShEM    : out std_logic_vector (26 downto 0)
   );
end Shifter ;
```

Per realizzare lo shifting si è deciso di utilizzare un *barrel shifter* a 5 livelli di multiplexer, in quanto, essendo la dimensione della mantissa EXTM uguale a $N = 27$ bit, questa tipologia di shifter richiede $\lceil \log_2 N \rceil$ livelli di multiplexer con N multiplexer 2-1 ad ogni livello. Com'era stato accennato precedentemente, se la differenza tra gli esponenti è maggiore di 27, ovvero se il flag FM2Zero è asserito, la mantissa sarà forzata tutta a zero, altrimenti, la mantissa verrà shiftata di ShiftN posizioni a destra.
Per capire meglio il funzionamento e il design fisico di un barrel shifter, è di seguito riportata, per semplicità di rappresentazione, una possibile architettura a 4 bit che realizza lo shift a destra. Si noti che se il bit che pilota il livello è asserito, al livello k, i segnali selezionati ad ogni multiplexer sono quelli che si trovano 2^k posizioni prima (caso shift a destra) e nel caso non ci siano segnali 2^k posizioni prima viene preso `'0'`[2].

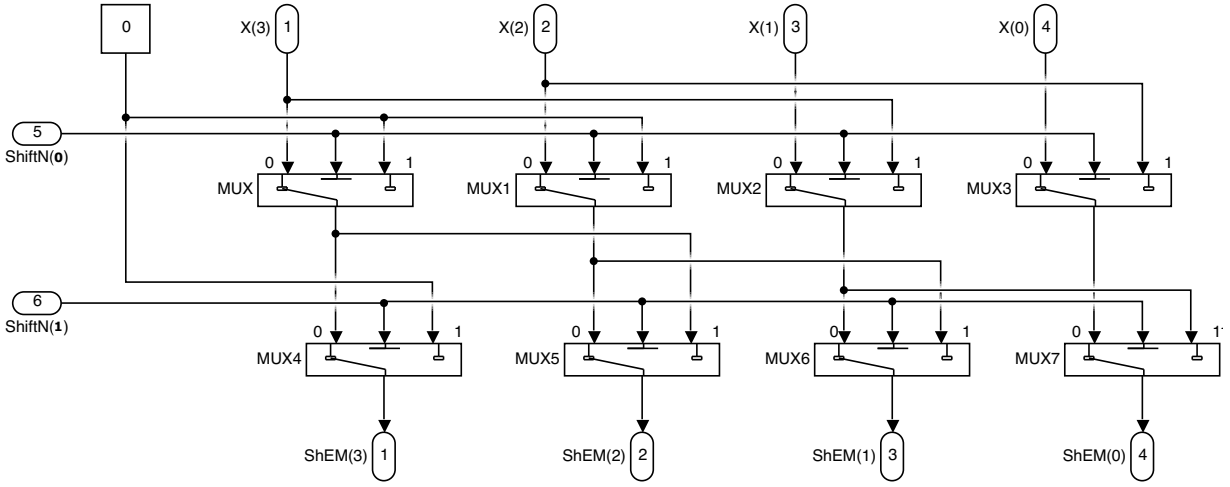


Figura 2.2: Come si può notare, nel caso di vettore in ingresso di 4 bit, sono necessari 2 livelli di multiplexer, ognuno composto 4 mux.

Un modulo duale a questo verrà utilizzato per realizzare lo shifting a sinistra nella normalizzazione.

L'implementazione in VHDL riflette molto bene la struttura pensata.

```
primoLivello     <= '0'& ExtM(26 downto 1) &  when
                    ShiftN(0) = '1' else ExtM;
secondoLivello   <= "00" & primoLivello(26 downto 2) when
                    ShiftN(1) = '1' else primoLivello;
terzoLivello     <="0000" & secondoLivello(26 downto 4) when
                    ShiftN(2) = '1' else secondoLivello;
quartoLivello    <="00000000" & terzoLivello(26 downto 8) when
                    ShiftN(3) = '1' else terzoLivello;
quintoLivello    <="0000000000000000" & quartoLivello(26 downto 16)  when
                    ShiftN(4) = '1' else quartoLivello;
ShEM             <= quintoLivello when FM2Zero = '0' else
                    "00000000000000000000000000000";
```

2.3 Fase di somma

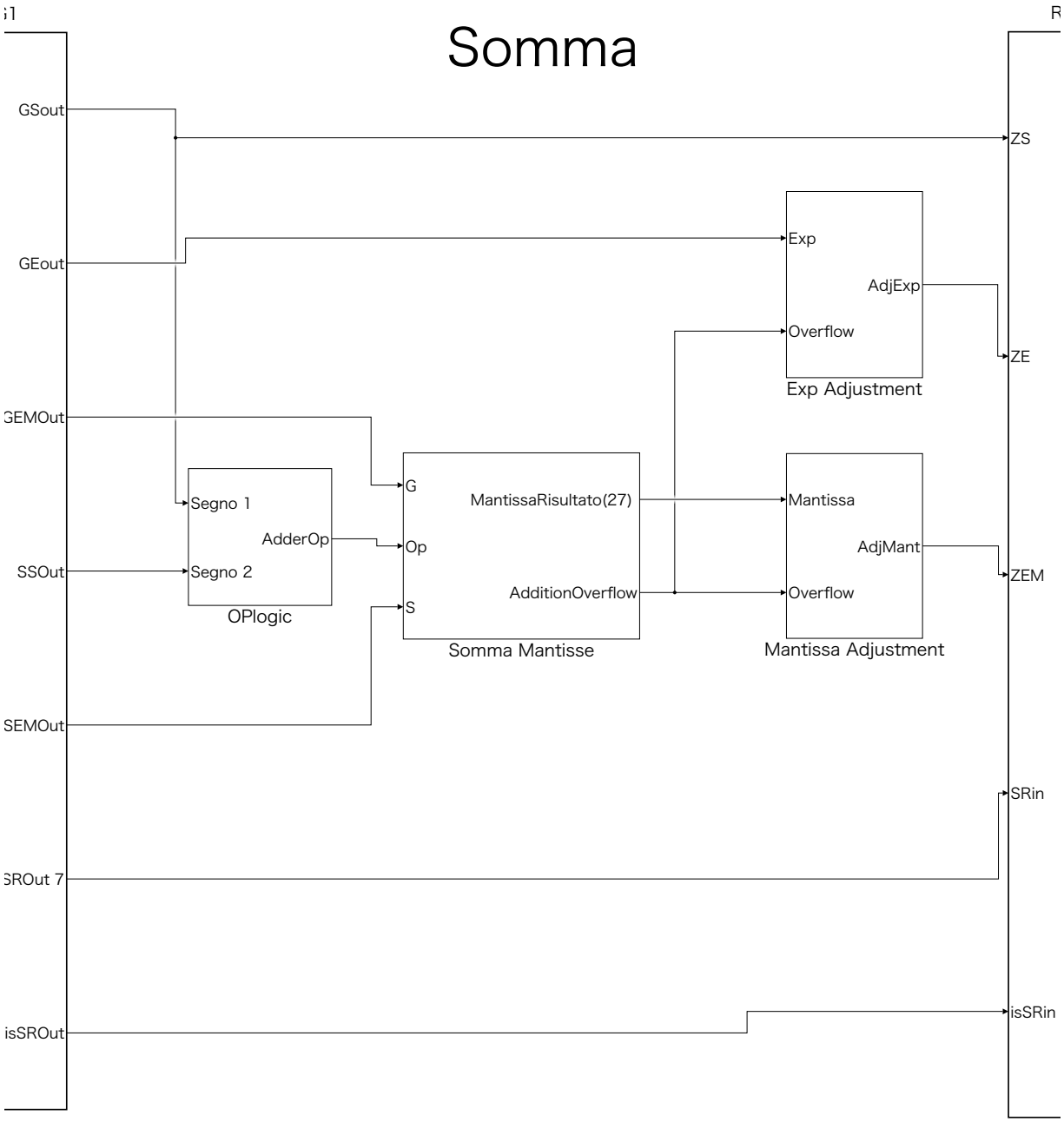
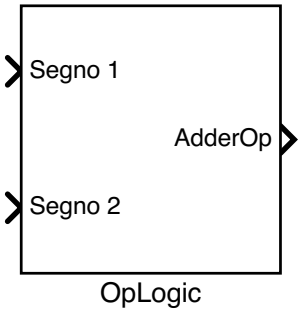


Figura 2.3: È possibile zoomare digitalmente per osservare i singoli moduli e i singoli segnali

2.3.1 Operation logic

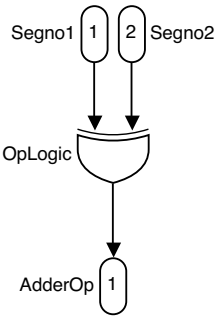



```
entity OpLogic is
   port(
      Segno1  : in   std_logic;
      Segno2  : in   std_logic;
      AdderOp : out  std_logic
   );
end OpLogic ;
```

Questo modulo calcola semplicemente l'operazione da svolgere nel sommatore, riceve in ingresso i segni degli operandi interni *Segno1* e *Segno2* (quindi eventualmente cambiati di segno e/o scambiati), e produrrà in uscita *AdderOp* che assumerà `0` se l'operazione da effettuare nel sommatore è la somma, `1` se la differenza. Si noti che questo modulo non utilizza in alcun modo l'operazione iniziale, poiché l'informazione dell'operazione è utilizzata per cambiare di segno il secondo operando.
È importante notare che questo modulo indica soltanto l'operazione che deve realizzare il sommatore, e che il sommmatore di mantisse posizionato a valle si preoccupa soltanto del calcolo del risultato in modulo, ovvero il segno del risultato è determinato dall'operando maggiore[1].

Il segnale in uscita *AdderOp* è calcolato secondo la seguente tabella della verità che può essere implementata con una semplice porta XOR.

Segno 1	Segno 2	AdderOp
+	+	+
+	-	-
-	+	-
-	-	+



L'implementazione di questo modulo risulta molto semplice grazie al fatto che nella prima fase si sfrutta il cambiamento di segno in caso di operazione di sottrazione, ovvero si sfrutta il fatto che:

```
X-Y = X+(-Y)  e  X-(-Y) = X+Y
```

[1]Si ricordi che se l'operazione da svolgere è la sottrazione, il segno del secondo operando viene cambiato ma il calcolo dell'operando maggiore avviene a valle di questo cambiamento di segno.

Per chiarire ulteriormente il fatto che non è necessaria l'operazione in questo modulo si riportano di seguito un paio di esempi:

Si supponga di avere in input al top level i numeri `-23.17` e `-57.0` codificati in virgola mobile di cui si vuole fare la differenza, ovvero `-23.17-(-57.0)`.
Come prima cosa la fase di pre-somma cambierà il segno del secondo operando dato che l'operazione da realizzare è una sottrazione (quindi `-57.0` diventa `+57.0`), successivamente si realizza uno swap per portare come primo operando quello maggiore e per porre il segno del risultato uguale al segno di questo operando (in questo caso il maggiore è ora `57.0` e di conseguenza il segno del risultato sarà +). Una volta entrati nella seconda fase, avremo come primo operando `57.0` e come secondo `-23.17`, dalla tabella della verità mostrata si ottiene che l'operazione che deve realizzare il sommatore è una differenza.

Si consideri ora il caso `-57.0-(-23.17)`, rifacendo lo stesso ragionamento di prima si noti che il modulo *OpLogic* riceve in ingresso `-57.0` e `23.17` e indica al sommatore di realizzare la differenza, si faccia attenzione al fatto che il sommatore prende il valore assoluto degli operandi e non considera in alcun modo il segno, che come si ricorda è fissato a priori dall'operando più grande.

2.3.2 Sommatore

Il sommatore di mantisse si è descritto in VHDL come un *CLA (Carry Look Ahead)*. Com'è ben noto, i moduli CLA si basano sul principio di anticipazione del riporto, ovvero il calcolo di ogni singolo riporto con due funzioni di supporto:

$$\begin{array}{ll} \text{Propagazione} & P_i = x_i + y_i \\ \text{Generazione} & G_i = x_i y_i \end{array}$$

Queste funzioni non dipendendo da alcuna propagazione, possono essere calcolate in un primo stadio e realizzate tutte in parallelo, e, una volte fatta questa fase, può essere calcolato il riporto $i+1-esimo$ come:

$$c_{i+1} = G_i + P_i c_i$$

Si è deciso di fare il sommatore in questo modo principalmente perché non volendo fare inferenziazione aritmetica e dovendo esplicitare la struttura interna, è la struttura più semplice che consente l'uso dei generic.

Un macro schema di funzionamento di un CLA a 4 bit è riportato nella figura sottostante.

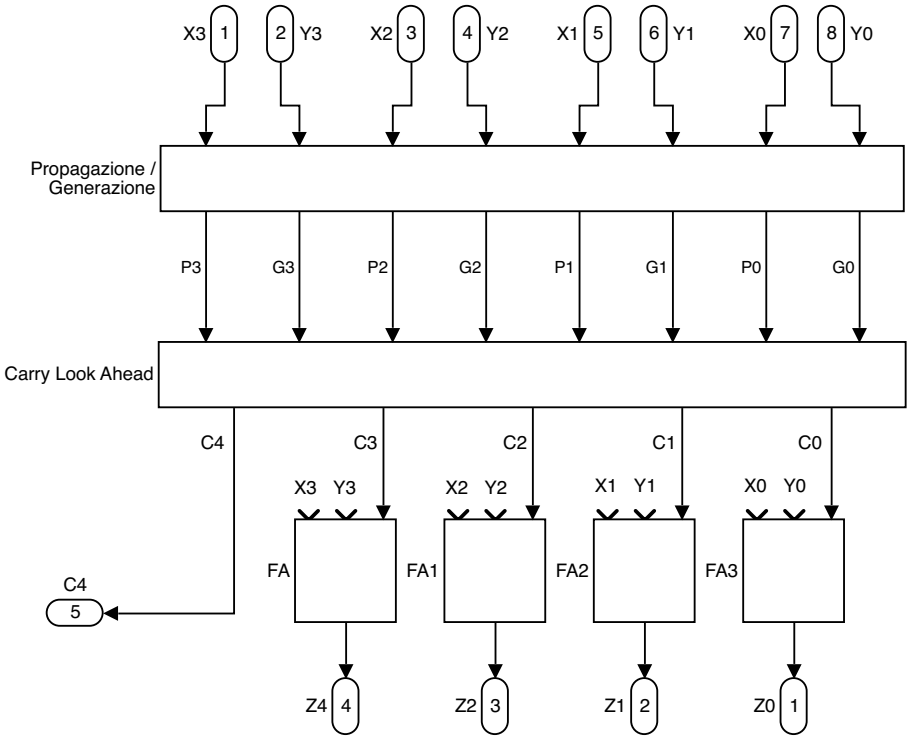


Figura 2.4: CLA 4 bit

Un modo corretto per la descrizione di questo modulo CLA richiede di esplicitare ogni singola funzione che calcola il riporto $i-esimo$ ad esempio:

$$c_4 = G_3 + P_3G_2 + P_2P_1G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0c_0$$

che ahimè per 27 bit diventano funzioni combinatorie lunghissime ...
Possiamo descrivere il comportamento con un ciclo for generate[3] che porti alla sintesi di un sommatore simile al CLA senza inserire esplicitamente tutte le equazioni:

```
CarryLoop: for I in 0 to N-1 generate
c_temp(I+1) <= g(I) or ( p(I) and c_temp(I) );
end generate CarryLoop;
```

Il problema di questa scelta è che non è perfettamente corretta nel senso che quello che stiamo descrivendo non è proprio il CLA (i segnali di riporto non vengono calcolati allo stesso tempo ma vengono calcolati in cascata, dunque il ritardo sarà molto alto) comunque non è da ritenersi sbagliato in quanto il codice descrive soltanto il funzionamento ed è ISE a sintetizzare una architettura fisica che produca lo stesso comportamento, se poi si inseriscono dei time-constraints

sicuramente si perviene alla struttura che si intendeva descrivere, in effetti Altera una delle aziende produttrice di FPGA più grandi al mondo (ora parte di Intel), ha nel suo sito una struttura di un CLA molto simile a ciò che è stato fatto. [2]

Ci sono altri modi di fare un sommatore generico, ad esempio facendo un ciclo for generate in cui si istanziano in modo generico dei full-adder in Ripple Carry ma si sono realizzate delle prove e i tempi risultano molto simili, ma i tempi restano comunque inferiori al simil-CLA.

```
FaCascade: for I in 0 to N generate
    FA_i: FullAdder
    PORT MAP(
        a       => y_c(I),
        b       => x_temp(I),
        cin     => temp(I),
        s       => z_temp(I),
        cout    => temp(I+1)
    );
end generate FaCascade;
```

Nella fase di somma ci sono due sommatori, uno che realizza la somma di mantisse e uno che realizza la somma dell'overflow della somma di mantisse con l'esponente più grande che è stato individuato nelle fasi precedenti. Per l'implementazione in VHDL si tratta dello stesso modulo ma diverse istanziazioni con parametro generic diverso.

È importante notare che quando si verifica un overflow il risultato è `1` concatenato ai 26 bit della somma di mantisse.

[2] cfr. `https://www.altera.com/support/support-resources/design-examples/design-software/vhdl/v_cl_addr.html`

2.4 Fase correttiva

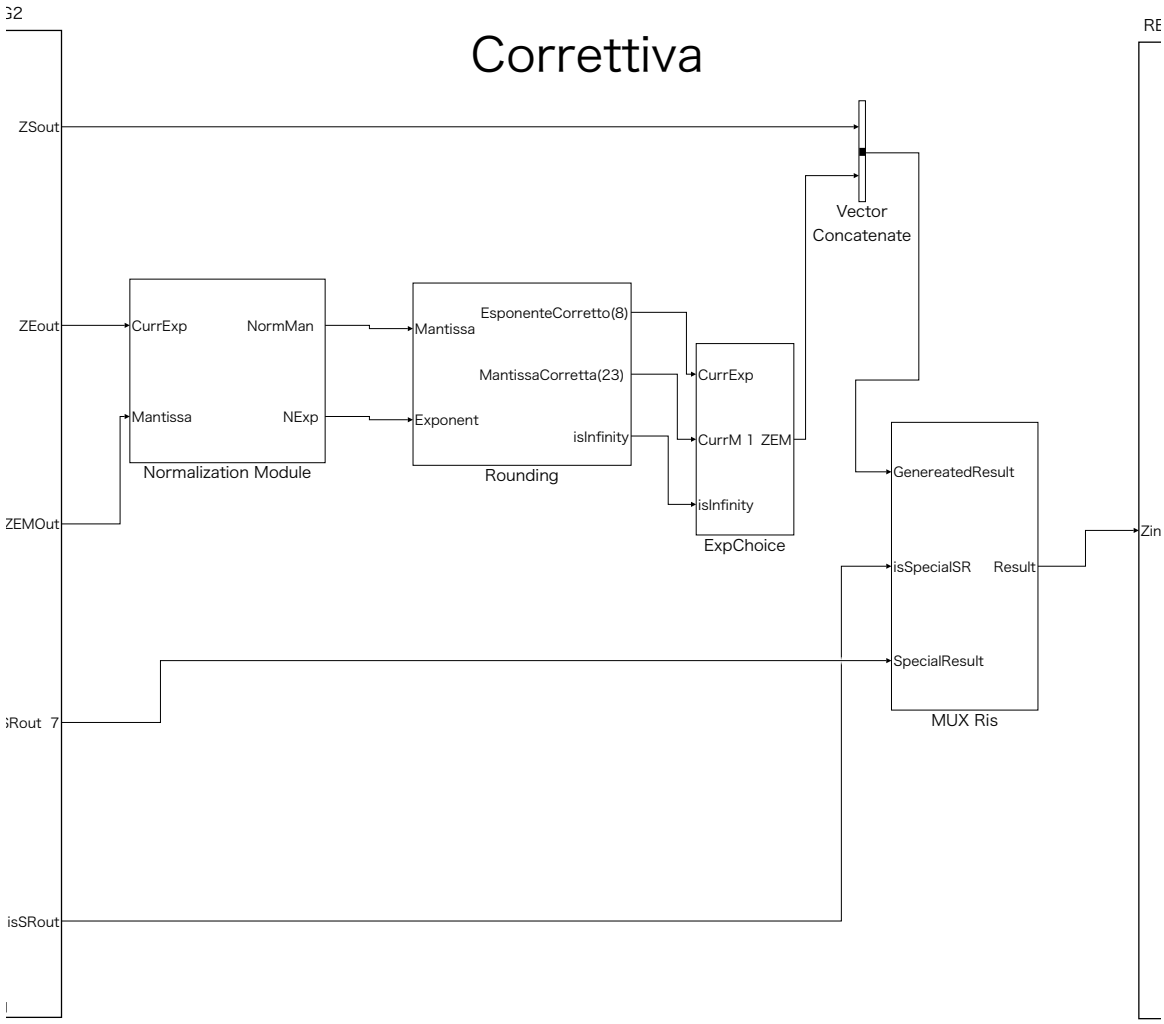
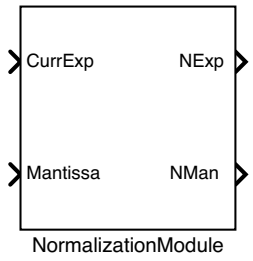


Figura 2.5: È possibile zoomare digitalmente per osservare i singoli moduli e i singoli segnali

2.4.1 Normalizzazione




```
entity NormalizationModule is
   port(
      CurrExp : in  std_logic_vector (7 downto 0);
      CurrMan : in  std_logic_vector (26 downto 0);
      NormExp : out std_logic_vector (7 downto 0);
      NormMan : out std_logic_vector (26 downto 0)
   );
end NormalizationModule;
```

Questo modulo prende in ingresso l'esponente e la mantissa corrente $CurrExp$ e $CurrMan$. Dentro al modulo si agisce con il seguente algoritmo:

1. Si calcolano le posizioni shiftabili (Exp in ingresso -1), questo perché l'esponente DEC 1 è logicamente uguale all'esponente DEC 0. Bisogna fare attenzione che se l'esponente in ingresso è zero le posizioni shiftabili sono zero (Exp in ingresso -1 ci da il numero massimo rappresentabile perché si va in underflow).

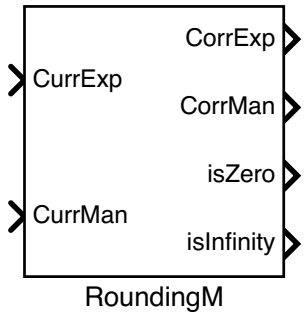
2. Il modulo calcola internamente il numero di posizioni che si deve shiftare la mantissa in modo simile a un priority encoder.

$$RequiredShift = 27 - PriorityEncoderResult$$

3. Con un comparatore si calcola la differenza $ShiftablePos - RequiredShift$, e si valuta se $ShiftablePos < RequiredShift$, nel caso quest'espressione sia vera, si asserisce un segnale interno chiamato $ShiftNotFeasible$.

4. Se $ShiftNotFeasible =$ `'1'`, l'esponente viene posto a zero in modo che il numero in uscita sia denormalizzato, la mantissa viene in questo caso shiftata a destra di $ShiftablePos$. Altrimenti se $ShiftNotFeasible =$ `'0'`, l'esponente viene posto a $CurrExp - RequiredShift$ e la mantissa è shiftata a destra di $RequiredShift$.

2.4.2 Arrotondamento




```
entity RoundingM is
   port(
      CurrExp : in  std_logic_vector (7 downto 0);
      CurrMan : in  std_logic_vector (27 downto 0);
      CorrExp : out std_logic_vector (7 downto 0);
      CorrMan : out std_logic_vector (22 downto 0);
      isZero  : out std_logic;
      isInfty : out std_logic
   );
end RoundingM ;
```

Il modulo prende in input *CurrExp* e *CurrMan* che sono rispettivamente l'esponente e la mantissa dopo la normalizzazione, e produce in uscita l'esponente e mantissa finale che verranno posteriormente concatenati al segno del risultato previamente calcolato. Si noti che la dimensione della mantissa in uscita è 23 bit mentre in ingresso è 27 bit, poiché nel modulo si perdono i 3 bit di servizio e il bit implicito.

Per gestire l'arrotondamento bisogna ricordare cosa dice lo standard. Lo standard prevede di default l'arrotondamento al numero più vicino, quindi se il primo dei bit di servizio (G - *guard bit*) è uguale a zero, ciò vuol dire che il numero più vicino è il numero corrente troncato dei bit di servizio.

Nel caso in cui il *guard bit* sia uguale a 1 bisogna vedere in che caso ci si trova: se ci si trova nel caso "100", ovvero quello "a metà", bisogna vedere qual è il valore del LSB poiché lo standard prevede che in caso di uguale distanza, l'arrotondamento deve avvenire a un numero pari, cioè si deve fare in modo che LSB diventi 0.

Negli altri casi, cioè quando i bit di servizio assumono un valore in binario ">100" si deve arrotondare al numero successivo quindi bisogna sommare un 1 al LSB.

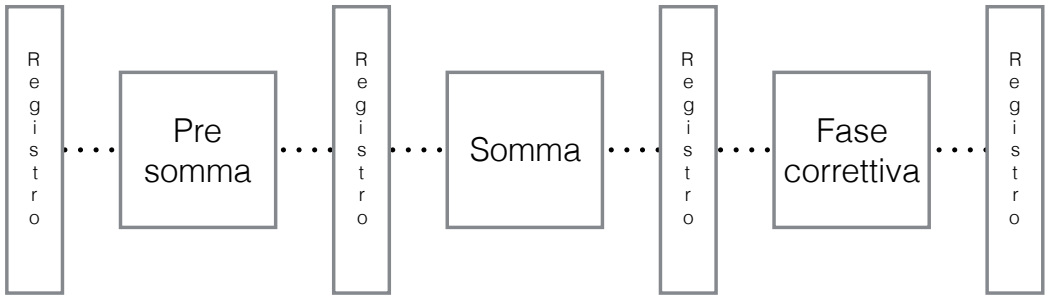
LSB	G	R	S	incremento
0	0	-	-	0
0	1	0	0	0
0	1	0	1	1
0	1	1	-	1
1	0	-	-	0
1	1	-	-	1

Tabella 2.1: Tabella di arrotondamento

Questo modulo segnala che il risultato dev'essere zero con *isZero* o che il risultato dev'essere infinito con *isInfty* quando si verifica rispettivamente, $E = 0$ oppure $E = 255$.

2.5 Pipeline

Il pipelining è una tecnica per migliorare le prestazioni basata sulla sovrapposizione dell'esecuzione di più istruzioni appartenenti ad un flusso di esecuzione sequenziale. Nel nostro caso, si è suddiviso il sommatore floating point in 3 fasi, separate mediante l'utilizzo di registri (flip-flop).



Il pipelining produce, una volta raggiunto il regime (pipeline piena), un risultato ogni ciclo di clock, e consente un notevolmente miglioramento del throughput (ovvero il numero di istruzioni completate per unità di tempo) rispetto al caso puramente combinatorio. Infatti con questa tecnica si dimunisce il periodo di clock richiesto e dunque si riesce ad aumentare la frequenza di funzionamento.

A titolo di esempio, viene mostrato come sono stati inferenziati i registri sensibile al fronte di salita del clock in VHDL[4]:

```
if( CLK'event and CLK = '1' ) then
...
end if;
```

Innanzitutto, è importante notare che nella sensitivity list del process è presente soltanto il segnale di clock, e quindi lo stato del registro non è influenzato in alcun modo con le transizioni degli altri segnali. Per riusucire a inferenziare un registro si deve omettere intenzionalmente il ramo else, giacché in assenza di un fronte di salita sul segnale di clock il dispositivo non deve realizzare nessun cambio nel segnale e dunque deve mantenere il proprio stato.

Nell'implementazione è presente anche un segnale di RESET sincrono rispetto al clock che non fa altro che resettare lo stato del flip-flop.

```
if( RESET = '1' ) then
... -- set to zero
else
... -- assign signals
end if;
```

Gli stadi sono risultati abbastanza bilanciati

	Source Pad	Destination Pad	Delay
1	inY<0>	SMantE<1>	24.543
2	inY<1>	SMantE<1>	24.413
3	inY<0>	SMantE<2>	24.168
4	inY<1>	SMantE<2>	24.038
5	inY<0>	SMantE<0>	23.897
6	inY<0>	SMantE<6>	23.794
7	inY<1>	SMantE<0>	23.767

Figura 2.6: Ritardo stadio di pre-somma

	Source Pad	Destination Pad	Delay
1	GSign_in	SResul...ut<30>	29.129
2	GSign_in	SResul...ut<25>	28.715
3	GSign_in	SResul...ut<26>	28.698
4	GSign_in	SResul...ut<31>	28.580
5	GSign_in	SResul...ut<29>	28.503
6	GSign_in	isSpecR_out	28.451
7	GSign_in	SResul...ut<27>	28.413

Figura 2.7: Ritardo stadio di somma

	Source Pad	Destination Pad	Delay
1	ZMantE_in<13>	Z<18>	29.798
2	ZMantE_in<13>	Z<17>	29.774
3	ZMantE_in<15>	Z<18>	29.576
4	ZMantE_in<18>	Z<18>	29.571
5	ZMantE_in<15>	Z<17>	29.552
6	ZMantE_in<18>	Z<17>	29.547
7	ZMantE_in<13>	Z<15>	29.545

Figura 2.8: Ritardo stadio correttivo

Come clock si è scelto un clock a 40ns

CAPITOLO 3

TESTBENCH

Come è noto, fare *testing* non è un compito banale; dimostrare la correttezza risulta molto difficile a meno di non realizzare dei test di tipo esaustivo che realizzano esecuzioni per tutti i possibili ingressi. In questo progetto si è deciso di adottare l'approccio *black-box o funzionale* in cui i casi di test sono determinati dalla specifica e non dal codice scritto. Nella maggior parte dei test a seguire quello che è stato fatto è un partizionamento sistematico del dominio di input in modo tale da testare almeno un elemento rappresentativo di ogni partizione, in alcuni casi si è deciso di testare anche i valori limiti, ovvero i valori agli estremi di ogni partizione[5] (ad esempio numeri normalizzati con mantissa uguale a zero, numeri denormalizzati con mantissa di tutti uni, ecc.).

Nelle seguenti pagine verranno mostrati i casi di test pensati per i modi più significativi di ogni fase. Anche in questo caso così come per la sezione precedente, a titolo di esempio, verrà mostrato l'implementazione in VHDL del testbench per il modulo che identifica i casi.

3.1 Testbench identificazione casi

Come detto in precedenza, a mero titolo esemplificativo verrà mostrato per questo modulo non soltanto come si intende realizzare il testbench ma anche come si realizza in pratica. In questo caso i casi considerati sono:

1. Normale. Esponente > 0 e Mantissa = 0
2. Normale 2. Esponente > 0 e Mantissa $\neq$ 0
3. Subnormale. Esponente = 0 e Mantissa $\neq$ 0
4. Zero. Esponente = 0 e Mantissa = 0
5. NaN. Esponente = 255 e Mantissa $\neq$ 0
6. Infinito. Esponente = 255 e Mantissa = 0

```
architecture behavior of tbSCIdentifier is

    -- Component Declaration for the Unit Under Test (UUT)
    component SCIdentifier
    port(
         E     : IN  std_logic_vector(7 downto 0);
         M     : IN  std_logic_vector(22 downto 0);
         Casi  : OUT std_logic_vector(4 downto 0)
        );
    end component;

   --Inputs
   signal E : std_logic_vector(7 downto 0) := (others => '0');
   signal M : std_logic_vector(22 downto 0) := (others => '0');

   signal clk: std_logic := '0';

  --Outputs
   signal Casi : std_logic_vector(4 downto 0);

   --Constants
   constant clk_period : time := 10 ns;
```

```
BEGIN
   -- Instantiation of the UUT (Unit Under Test)
   uut: SCIdentifier port map (
           E => E,
           M => M,
           Casi => Casi
        );
   -- Processes
   clk_process :process
   begin
      clk <= '0';
      wait for clk_period/2;
      clk <= '1';
      wait for clk_period/2;
   end process;
   stim_proc: process
   begin
      wait for 100 ns;
            e <= "00000001";
            M <= "00000000000000000000000"; --normale
      wait for 100 ns;
            e <= "00000001";
            M <= "10000000000000000000000"; --normale 2
      wait for 100 ns;
            e <= "00000000";
            M <= "10000000000000000000000"; --subnormale
      wait for 100 ns;
            e <= "00000000";
            M <= "00000000000000000000000"; --zero
      wait for 100 ns;
            e <= "11111111";
            M <= "00000000000000000000001"; --NaN
      wait for 100 ns;
            e <= "11111111";
            M <= "00000000000000000000000"; --infinito
      wait;
   end process;
END;
```

3.2 Gestioni dei casi speciali

Saranno testati tutti i casi in tabella 1.2.1.

3.3 Testbench swapper

1. Operando 1 > Operando 2
2. Operando 1 < Operando 2
3. Operando 1 = Operando 2

3.4 Testbench shifter

1. Shift <23, FM2Zero = 1
2. Shift 23<x<26, FM2Zero = 1
3. Shift >26, FM2Zero = 1
4. Shift <23, FM2Zero = 0
5. Shift 23<x<26, FM2Zero = 0
6. Shift >26, FM2Zero = 0

3.5 Testbench normalizzazione

1. overflow=1, Exp =255, Mantissa="11..."
2. overflow=1, 0< Exp <255, Mantissa="- -...",
3. overflow=0, 0< Exp <255, Mantissa="- -..."
4. overflow=0, Exp=0, Mantissa="0 -..."
5. overflow=0, Exp=k, Mantissa=" $\underbrace{00...}_{\leq k\, volte}$
6. overflow=0, Exp=k, Mantissa=" $\underbrace{00...}_{>k\, volte}$

3.6 Testbench arrotondamento

1. normale → $+\infty$
2. normale → $-\infty$
3. normale → normale
4. subnormale → normale
5. subnormale → 0

3.7 Testbench Top

```
-- Normali, esponenti uguali, somma, Op1<Op2
X   <= "00010000100000000000000000000000";  -- DEC 5.0487098 E-29 HEX 0x10800000
Y   <= "00010000100000000000000000000001";  -- DEC 5.0487104 E-29 HEX 0x10800001
op  <= '0';                                 -- Segno +
-- Risultato Atteso
-- BIN  00010001000000000000000000000000    -- DEC 1.00974196E-28 HEX 0x11000000

-- Normali, esponenti uguali, somma, Op1>Op2
X   <= "00010000100000000000000000000001";  -- DEC 5.0487098 E-29 HEX 0x10800000
Y   <= "00010000100000000000000000000000";  -- DEC 5.0487104 E-29 HEX 0x10800001
op  <= '0';                                 -- Segno +
-- Risultato Atteso
-- BIN  00010001000000000000000000000000    -- DEC 1.00974196E-28 HEX 0x11000000

-- Normali, esponenti uguali, sottrazione, Op1<Op2
X   <= "00010000100000000000000000000000";  -- DEC 5.0487098 E-29 HEX 0x10800000
Y   <= "00010000100000000000000000000001";  -- DEC 5.0487104 E-29 HEX 0x10800001
op  <= '1';                                 -- Segno -
-- Risultato Atteso
-- BIN  00000101000000000000000000000000    -- DEC 6.018531  E-36 HEX 0x05000000

-- Normali, esponenti uguali, sottrazione, Op1>Op2
X   <= "00010000100000000000000000000001";  -- DEC 5.0487098 E-29 HEX 0x10800000
Y   <= "00010000100000000000000000000000";  -- DEC 5.0487104 E-29 HEX 0x10800001
op  <= '1';                                 -- Segno -
-- Risultato Atteso
-- BIN 10000101000000000000000000000000     -- DEC-6.018531  E-36 HEX 0x85000000
```

```
--------------------------------Esponenti diversi-------------------------------


-- Normali, esponenti diversi , somma, Op1>Op2
X   <= "00010010100000000000000000000000";  -- DEC 8.0779357 E-28 HEX 0x12800000
Y   <= "00010000100000000000000000000001";  -- DEC 5.0487104 E-29 HEX 0x10800001
op  <= '0';                                 -- Segno +
-- Risultato Atteso
-- BIN  00010010100010000000000000000000    -- DEC 8.58280664E-28 HEX 0x12880000


-- Normali, esponenti diversi , somma, Op1<Op2
X   <= "00010000100000000000000000000001";  -- DEC 5.0487104 E-29 HEX 0x10800001
Y   <= "00010010100000000000000000000000";  -- DEC 8.0779357 E-28 HEX 0x12800000
op  <= '0';                                 -- Segno +
-- Risultato Atteso
-- BIN  00010010100010000000000000000000    -- DEC 8.5828066 E-28 HEX 0x12880000


-- Normali, esponenti diversi , sottrazione, Op1>Op2
X   <= "00010010100000000000000000000001";  -- DEC 8.077937  E-38 HEX 0x12800001
Y   <= "00010000100000000000000000000000";  -- DEC 5.0487098 E-29 HEX 0x10800000
op  <= '1';                                 -- Segno -
-- Risultato Atteso
-- BIN  00010010011100000000000000000010    -- DEC 7.5730657E-28 HEX 0x12700002


-- Normali, esponenti diversi , sottrazione, Op1<Op2
X   <= "00010000100000000000000000000000";  -- DEC 5.0487098 E-29 HEX 0x10800000
Y   <= "00010010100000000000000000000001";  -- DEC 8.077937  E-28 HEX 0x12800001
op  <= '1';                                 -- Segno -
-- Risultato Atteso
-- BIN  10010010011100000000000000000010    -- DEC-7.5730657E-28 HEX 0x92700002
```

```
---------------------------------Operandi misti-------------------------------


-- Normale e subnormale, somma, Op1>Op2
 X  <= "00000000100000001000000000000000";   -- DEC 1.1800861 E-38 HEX 0x00808000
 Y  <= "00000000000000001000000000000000";   -- DEC 9.18355   E-41 HEX 0x00010000
 op <= '0';                                  -- Segno +
-- Risultato Atteso
-- BIN  00000000100000011000000000000000    -- DEC 1.189269  E-38 HEX 0x00818000


-- Normale e subnormale, somma, Op1<Op2
 X  <= "00000000000000001000000000000000";   -- DEC 9.18355.  E-41 HEX 0x00010000
 Y  <= "00000000100000001000000000000000";   -- DEC 1.1800861 E-38 HEX 0x00808000
 op <= '0';                                  -- Segno +
-- Risultato Atteso
-- BIN 00000000100000011000000000000000     -- DEC 1.189269  E-38 HEX 0x00818000


-- Normale e subnormale, sottrazione, Op1>Op2
 X  <= "00000000100000001000000000000000";   -- DEC 1.1800861 E-38 HEX 0x00808000
 Y  <= "00000000000000001000000000000000";   -- DEC 9.18355.  E-41 HEX 0x00010000
 op <= '1';                                  -- Segno -
-- Risultato Atteso
-- BIN 00000000111111111000000000000000     -- DEC 1.1663108E-38 HEX 0x007f8000


-- Normale e subnormale, sottrazione, Op1<Op2
 X  <= "00000000000000001000000000000000";   -- DEC 9.18355.  E-41 HEX 0x00010000
 Y  <= "00000000100000001000000000000000";   -- DEC 1.1800861 E-38 HEX 0x00808000
 op <= '1';                                  -- Segno -
-- Risultato Atteso
-- BIN 10000000111111111000000000000000     -- DEC-1.1663108 E-38 HEX 0x807f8000
```

```
--------------------------Operandi entrambi subnormali-------------------------


-- Entrambi subnormali, somma, Op1>Op2
X  <= "00000000000000010000000000000000";   -- DEC 9.18355.  E-41 HEX 0x00010000
Y  <= "00000000000000000000000000000001";   -- DEC 1.4       E-45 HEX 0x00000001
op <= '0';                                  -- Segno +
-- Risultato Atteso
-- BIN  00000000000000010000000000000001    -- DEC 9.1837    E-41 HEX 0x00010001

-- Entrambi subnormali, somma, Op1<Op2
X  <= "00000000000000000000000000000001";   -- DEC 1.4       E-45 HEX 0x00000001
Y  <= "00000000000000010000000000000000";   -- DEC 9.18355   E-41 HEX 0x00010000
op <= '0';                                  -- Segno +
-- Risultato Atteso
-- BIN  00000000000000010000000000000001    -- DEC 9.1837    E-41 HEX 0x00010001

-- Entrambi subnormali, sottrazione, Op1>Op2
X  <= "00000000000000010000000000000000";   -- DEC 9.18355   E-41 HEX 0x00010000
Y  <= "00000000000000000000000000000001";   -- DEC 1.4       E-45 HEX 0x00000001
op <= '1';                                  -- Segno -
-- Risultato Atteso
-- BIN 00000000000000000111111111111111     -- DEC 9.1834    E-41 HEX 0x0000ffff

-- Entrambi subnormali, sottrazione, Op1<Op2
X  <= "00000000000000000000000000000001";   -- DEC 1.4       E-45 HEX 0x00000001
Y  <= "00000000000000010000000000000000";   -- DEC 9.18355   E-41 HEX 0x00010000
op <= '1';                                  -- Segno -
-- Risultato Atteso
-- BIN 10000000000000000111111111111111     -- DEC -9.1834   E-41 HEX 0x8000ffff

--Subnormal special
X  <= "00000000000000000000000011111000";   -- DEC 3.48      E-43 HEX 0x000000f8
Y  <= "00000000000000000000000001110000";   -- DEC 1.9       E-43 HEX 0x00000088
op <= '1';                                  -- Segno -
-- Risultato Atteso
-- BIN 00000000000000000000000010001000     -- DEC  1.9.   E-43 HEX 0x80000088
```

```
---------------------------------Casi particolari------------------------------


-- NaN 1
X  <= "01111111110000000000000000000001";   -- DEC NaN            HEX 0x7fc00001
Y  <= "00010000100000000000000000000000";   -- DEC 5.048709  E-29 HEX 0x10800000
op <= '1';                                 -- Segno -
-- Risultato Atteso
-- BIN 01111111111111111111111111111111     -- DEC NaN            HEX 0x7fffffff

-- NaN 2
X  <= "01111111100000000000000000000000";   -- DEC +Infinity      HEX 0x7f800000
Y  <= "01111111100000000000000000000000";   -- DEC +Infinity      HEX 0x7f800000
op <= '1';                                  -- Segno -
-- Risultato Atteso
-- BIN 01111111111111111111111111111111     -- DEC NaN            HEX 0x7fffffff

-- NaN 3
X  <= "01111111100000000000000000000000";   -- DEC +Infinity      HEX 0x7f800000
Y  <= "11111111100000000000000000000000";   -- DEC -Infinity      HEX 0xff800000
op <= '0';                                  -- Segno +
-- Risultato Atteso
-- BIN 01111111111111111111111111111111     -- DEC NaN            HEX 0x7fffffff

-- Infinity 1
X  <= "01111111011111111111111111111111";   -- DEC 3.402823  E 38 HEX 0x7f7fffff
Y  <= "01111111000000000000000000000011";   -- DEC 4.2.      E-45 HEX 0x00000003
op <= '0';                                  -- Segno +
-- Risultato Atteso
-- BIN 01111111100000000000000000000000     -- DEC +Infinity      HEX 0x7f800000

-- Infinity 2
X  <= "11111111000000000000000000000011";   -- DEC-4.2.      E-45 HEX 0x80000003
Y  <= "11111111011111111111111111111111";   -- DEC-3.402823  E 38 HEX 0xff7fffff
op <= '0';                                  -- Segno +
-- Risultato Atteso
-- BIN 11111111100000000000000000000000     -- DEC -Infinity      HEX 0xff800000
```

```
-- Operando 1 Zero
X  <= "00000000000000000000000000000000";   -- DEC 0              HEX 0x00000000
Y  <= "01110111000000000000000000011111";   -- DEC 2.596158 E 33  HEX 0x7700001f
op <= '0';                                  -- Segno +

-- BIN 01110111000000000000000000011111     -- DEC 2.596158 E 33 HEX 0x7700001f


-- Operando 2 Zero
X  <= "01110111000000000000000000011111";   -- DEC 2.596158 E 33  HEX 0x7700001f
Y  <= "00000000000000000000000000000000";   -- DEC 0              HEX 0x00000000
op <= '1';                                    -- Segno -
-- BIN 11110111000000000000000000011111     -- DEC-2.596158 E 33  HEX 0xf700001f
```

```
----------------------------Possibili Casi problematici-------------------------

-- Subnormali che si comportano come zero per la grande differenza di esponenti

--(1)
X  <= "00010000100000000000000000000001";   -- DEC 5.0487104 E-29 HEX 0x10800001
Y  <= "00000000000000000000000000000001";   -- DEC 1.4       E-45 HEX 0x00000001
op <= '0';                                  -- Segno +
-- Risultato Atteso
-- BIN 00010000100000000000000000000001     -- DEC 5.0487104 E-29 HEX 0x10800001

-- (2)
X  <= "00000000000000000000000000000001";   -- DEC 1.4       E-45 HEX 0x00000001
Y  <= "00010000100000000000000000000001";   -- DEC 5.0487104 E-29 HEX 0x10800001
op <= '0';                                  -- Segno +
-- Risultato Atteso
-- BIN 00010000100000000000000000000001     -- DEC 5.0487104 E-29 HEX 0x10800001

-- (3)
X  <= "00010000100000000000000000000001";   -- DEC 5.0487104 E-29 HEX 0x10800001
Y  <= "00000000000000000000000000000001";   -- DEC 1.4       E-45 HEX 0x00000001
op <= '1';                                  -- Segno -
-- Risultato Atteso
-- BIN 00010000100000000000000000000001     -- DEC 5.0487104 E-29 HEX 0x10800001

-- (4)
X  <= "00000000000000000000000000000001";   -- DEC 1.4       E-45 HEX 0x00000001
Y  <= "00010000100000000000000000000001";   -- DEC 5.0487104 E-29 HEX 0x10800001
op <= '1';                                 -- Segno -
-- Risultato Atteso
-- BIN 10010000100000000000000000000001     -- DEC-5.0487104E-29 HEX 0x90800001
```

```
-- Subnormali come risultato di operazione 1
X  <= "00000000100000000000000001110000";   -- DEC 1.17551  E-38 HEX 0x00800070
Y  <= "00000000100000000000000011111000";   -- DEC 1.1755291E-38 HEX 0x008000f8
op <= '1';                                  -- Segno -
-- Risultato Atteso
-- BIN 10000000000000000000000010001000     -- DEC -1.9     E-43 HEX 0x80000088


-- Subnormali come risultato di operazione 2
X  <= "00000011000000000000000001110000";   -- DEC 1.17551  E-38 HEX 0x03000070
Y  <= "00000011000000000000000011111000";   -- DEC 1.1755291E-38 HEX 0x030000f8
op <= '1';                                  -- Segno +
-- Risultato Atteso
-- BIN 00000000000000000000000000000000     -- DEC Zero.         HEX 0x80001100


-- Denormalizzati = Normalizzato (come somma)
X  <= "00000000011111111111111111111111";   -- DEC 1.175494 E-38  HEX 0x007fffff
Y  <= "00000000011111111111111111111111";   -- DEC 1.175494 E-38  HEX 0x007fffff
op <= '0';                                  -- Segno +
-- Risultato Atteso
-- BIN 00000000111111111111111111111110     -- DEC 2.596158 E 33 HEX 0x00fffffe


-- Denormalizzati = Normalizzato  (come sottrazione)
X  <= "00000000011111111111111111111111";   -- DEC 1.175494 E-38  HEX 0x007fffff
Y  <= "10000000011111111111111111111111";   -- DEC-1.175494 E-38  HEX 0x807fffff
op <= '1';                                  -- Segno -
-- Risultato Atteso
-- BIN 00000000111111111111111111111110     -- DEC 2.596158 E 33  HEX 0x00fffffe


-- (Arrotondamento a infinito)
X  <= "01111111011111111111111111111111";   -- DEC 3.4028235 E 38 HEX 0x7f7fffff
Y  <= "01111110000000000000000000000011";   -- DEC 4.253531  E 37 HEX 0x7e000003
op <= '0';                                  -- Segno +
-- Risultato Atteso
-- BIN 01111111100000000000000000000000     -- DEC +Infinity      HEX 0x7f800000
```

```
-- Not Really Infinity 1
X  <= "01111111011111111111111111111111";   -- DEC 3.402823 E 38  HEX 0x7f7fffff
Y  <= "00000000000000000000000000000011";   -- DEC 4.2.     E-45  HEX 0x00000003
op <= '0';                                  -- Segno +
-- Risultato Atteso
-- BIN 01111111011111111111111111111111     -- DEC 3.402823 E 38  HEX 0x7f7fffff


-- Not Really Infinity 2
X  <= "10000000000000000000000000000011";   -- DEC-4.2.     E-45  HEX 0x80000003
Y  <= "11111111011111111111111111111111";   -- DEC-3.402823 E 38  HEX 0xff7fffff
ops <= '0';                                 -- Segno +
-- Risultato Atteso
-- BIN 11111111011111111111111111111111     -- DEC-3.402823 E 38  HEX 0xff7fffff


-- Zero (Denormalizzati)
X  <= "00000000011111111111111111111111";   -- DEC 1.175494 E-38  HEX 0x007fffff
Y  <= "10000000011111111111111111111111";   -- DEC-1.175494 E-38  HEX 0x807fffff
op <= '0';                                  -- Segno +
-- Risultato Atteso
-- BIN 00000000000000000000000000000000     -- DEC Zero           HEX 0x00000000
```

BIBLIOGRAFIA

[1] William Kahan.
IEEE standard 754 for binary floating-point arithmetic.
Lecture Notes on the Status of IEEE, 754(94720-1776):11, 1996.

[2] Matthew R Pillmeier, Michael J Schulte, and Eugene George Walters.
Design alternatives for barrel shifters.
pages 3–4. Computer Science and Engineering Department Lehigh University, 2004.

[3] Charles H Roth Jr and Lizy K John.
Digital Systems Design Using VHDL.
Thomson-Engineering, 2007.

[4] Carlo Brandolese.
Introduzione al linguaggio VHDL.
pages 63–64. Politecnico di Milano.

[5] Luciano Baresi and Mauro Pezzè.
An introduction to software testing.
volume 148, pages 98–101. Electronic Notes in Theoretical Computer Science, Elsevier, 2006.