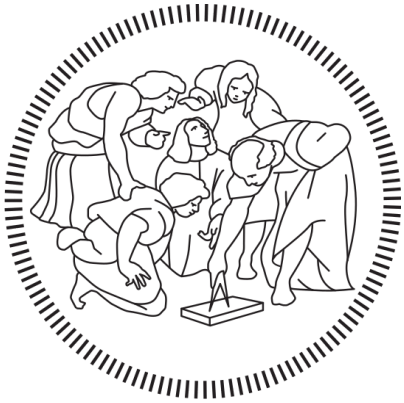
Sommatore Floating Point IEEE754

Prova Finale di Reti Logiche

Realizzato da:

Tirabassi Maurizio

Trotta Alessandro



Laurea in Ingegneria Informatica

POLITECNICO DI MILANO

Indice

1. Introduzione

1.1 Lo standard

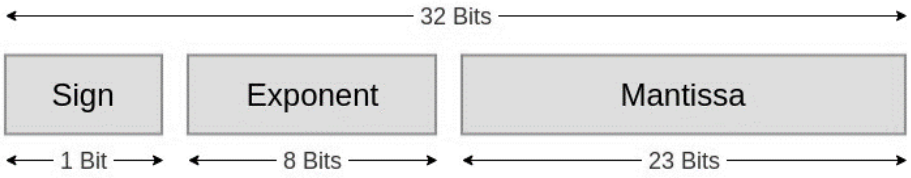


Figura 1.1: Suddivisione della parola di 32 bit secondo lo standard IEEE754

Il numero rappresentato in standard IEE754 è calcolabile come

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

dove $E = e - k$ con valore di bias $k = 127$.

Il valore di bias permette di rappresentare numeri positivi e negativi con esponente compreso tra –126 e 127.

La mantissa assume valore $M = 1.m$ quando l'hidden bit '1' è esplicitato.

1.2 Moduli notevoli

A seguire un'esposizione di moduli notevoli adottati, il loro comportamento ed implementazione.

1.2.1 Comparatore a 4 bit

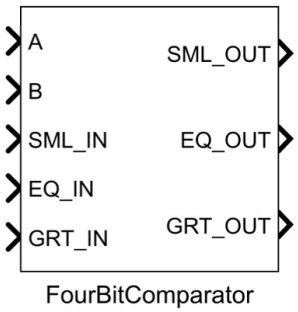


Figura 1.2: FourBitComparator

Il modulo *FourBitComparator* ha il ruolo di confrontare tra loro due parole da 4 bit, segnalando il risultato attraverso tre flag:

- $SML\_OUT = 1$ se il primo input è il minore tra i due
- $EQ\_OUT = 1$ se entrambi gli input sono uguali
- $GRT\_OUT = 1$ se il primo input è il maggiore tra i due

Il confronto avviene paragonando le due parole bit a bit.

I tre segnali di ingresso *SML_IN*, *EQ_IN* e *GRT_IN* indicano il risultato propagato da eventuali moduli che paragonano bit meno significativi.

Se usato in singola istanza, questi ultimi vengono imposti in ingresso in maniera seguente:

- $SML\_IN = 0$
- $EQ\_IN = 1$
- $GRT\_IN = 0$.

1.2.2 Comparatore a 8 bit

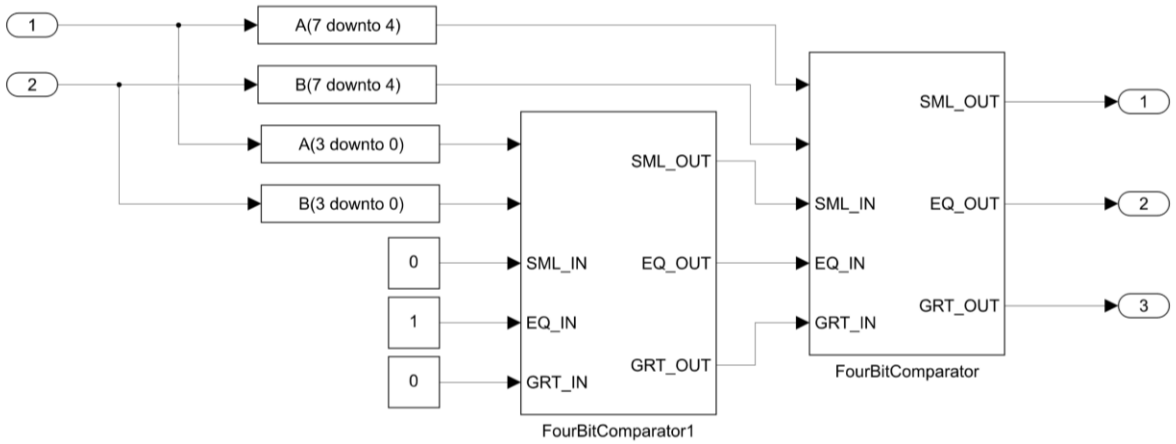


Figura 1.3: EightBitComparator

Il modulo *EightBitComparator* ha il ruolo di confrontare tra loro due parole da 8 bit, segnalando il risultato attraverso tre flag:

- $SML\_OUT = 1$ se il primo input è il minore tra i due
- $\mathrm{EQ\_OUT} = 1$ se entrambi gli input sono uguali
- $\mathrm{GRT\_OUT} = 1$ se il primo input è il maggiore tra i due

Il calcolo del risultato avviene attraverso l'utilizzo di un primo *FourBitComparator* per i 4 bit più significativi delle parole ed un secondo per i 4 bit meno significativi.

Se il *FourBitComparator* che paragona i 4 bit più significativi indica uguaglianza, il risultato dell'*EightBitComparator* coinciderà con quello del *FourBitComparator* che paragona i 4 bit meno significativi.

1.2.3 Comparatore a 23 bit

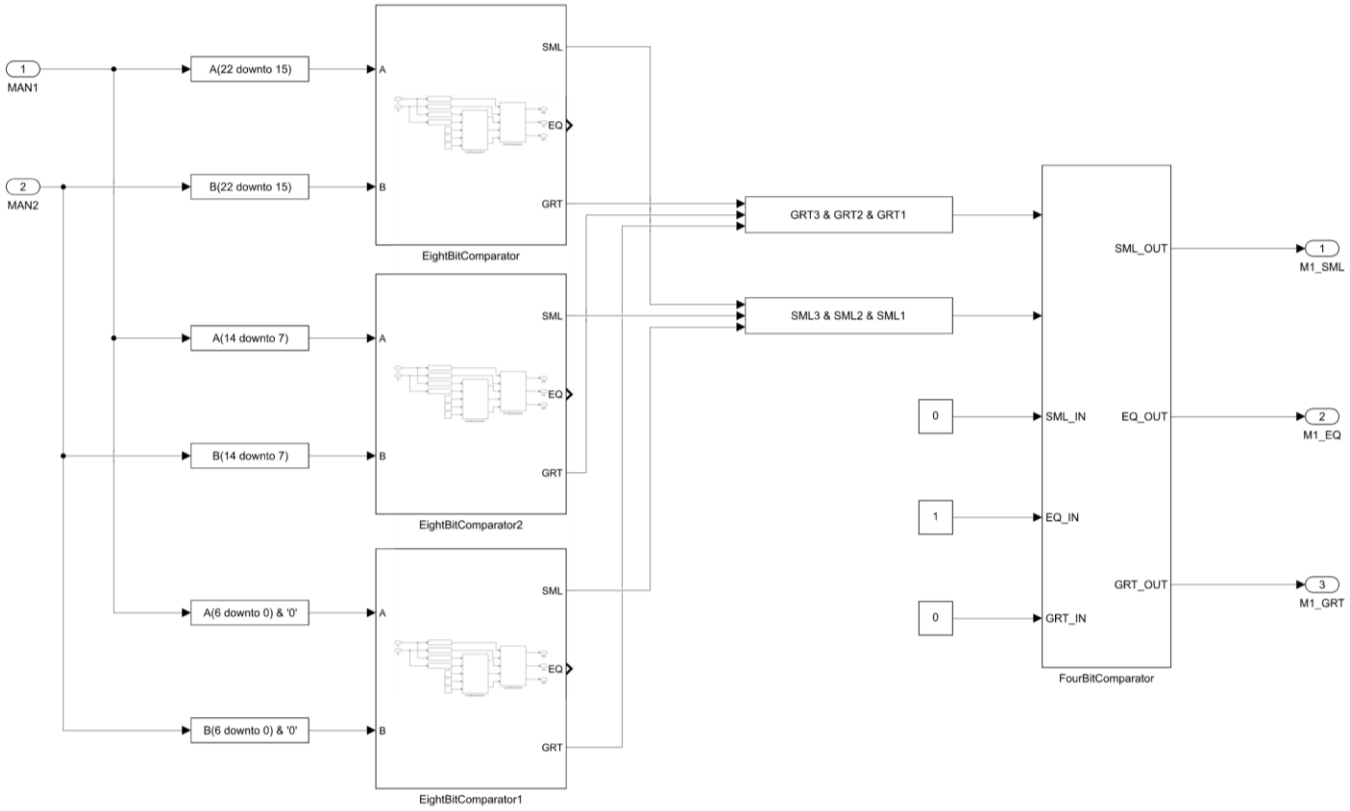


Figura 1.4: MantissaComparator

Il modulo *MantissaComparator* ha il ruolo di confrontare tra loro due parole da 23 bit, segnalando il risultato attraverso tre flag:

- $SML\_OUT = 1$ se il primo input è il minore tra i due
- $\mathrm{EQ\_OUT} = 1$ se entrambi gli input sono uguali
- $\mathrm{GRT\_OUT} = 1$ se il primo input è il maggiore tra i due

Il calcolo del risultato avviene tramite l'uso del modulo *EightBitComparator* istanziato tre volte e di un *FourBitComparator*.

Il terzo *EightBitComparator* riceve gli ultimi 7 bit delle mantisse da comparare e impone l'ottavo bit a '0'.

Il *FourBitComparator* calcola il risultato finale ricevendo in input dai tre *EightBitComparator* i rispettivi risultati in ordine di significatività e imponendo il quarto bit '0'.

1.2.4 Ripple Carry Adder (RCA)

Il modulo *RippleCarryAdder* ha il ruolo di sommare o sottrarre due numeri da N bit.

È composto da N moduli *FullAdder* che sommano i due numeri bit a bit.

In caso di sottrazione il secondo numero è complementato a due attraverso l'utilizzo di una porta XOR.

Si sceglie di ignorare un'eventuale overflow nel caso di sottrazione.

2. Il sommatore

2.1 Stadio di paragone

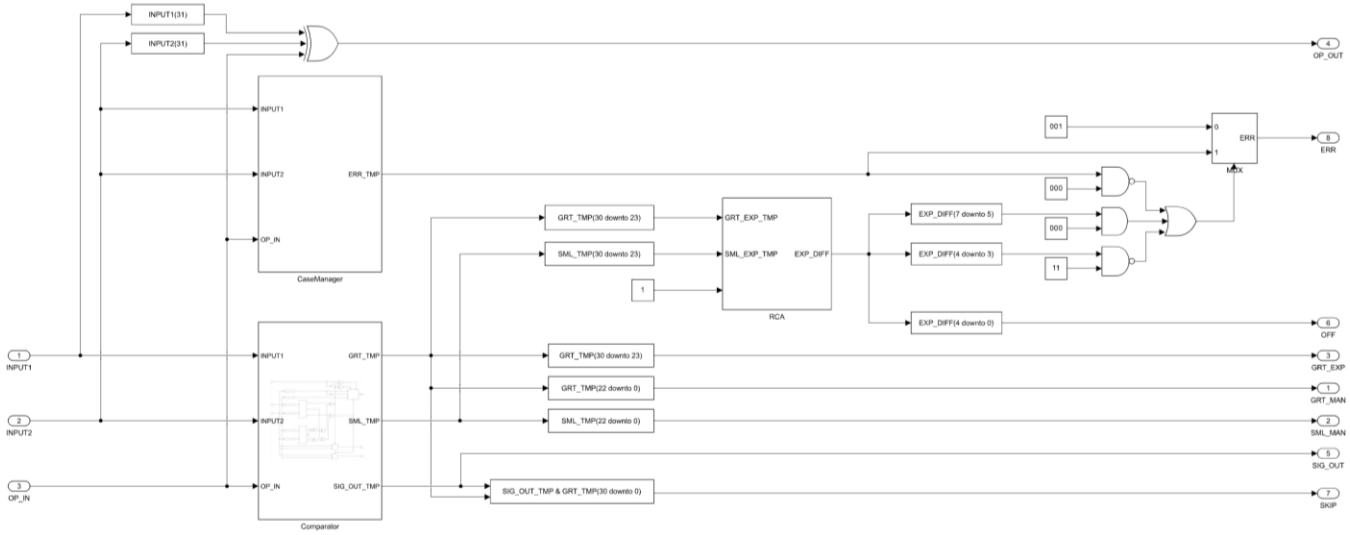


Figura 2.1: ComparingStage

Il primo stadio della pipeline si occupa di determinare l'operazione effettiva tra le due somme, paragonare i due numeri, identificare eventuali casi particolari dipendenti dai tipi di segnale in ingresso e verificare se uno dei due numeri è "molto più grande dell'altro".

Si è ricavata la logica da implementare per determinare l'operazione che sarà effettivamente eseguita tra i due numeri attraverso una sintesi con il metodo delle mappe di Karnaugh.

SIG1	SIG2	OP_IN	OP_OUT
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

	00	01	11	10
0	0	1	0	1
1	1	0	1	0

Figura 2.2: Tavola della verità (sinistra) e mappa di Karnaugh (destra)

Viene assegnato al segnale di *SKIP* il numero più grande in valore assoluto con segno corretto relativamente all'ordine degli operandi e all'operazione. Questo è il segnale che dovrà essere propagato come risultato finale del sommatore nel caso in cui uno dei due operandi sia nullo oppure la diferenza tra i loro esponenti ecceda 23.

2.1.1 CaseManager

Il modulo *CaseManager* si occupa di identificare eventuali casi particolari all'inizio della computazione, ovvero quei casi che dipendono dal tipo specifico di segnale in ingresso al sommatore.

Per identificare l'output particolare che il sommatore dovrà generare nell'eventualità si verifichi un'anormalità nel corso della computazione si è scelto di adottare una codifica a 3 bit assegnata al segnale *ERR*.

La codifica è la seguente:

- 000: Nessuna anormalità. Se alla fine della computazione il segnale di *ERR* mantiene tale codifica, il sommatore propagherà in output il risultato della somma canonica.
- 001: Propagare come risultato l'input che tra i due viene assegnato al segnale *SKIP*.
- 010: Generare $ZERO$.
- 011: Generare NaN.
- 100: Generare $+\infty$.
- 101: Generare $-\infty$.

Il modulo *CaseManager* esegue il parsing degli input isolando segno, esponente e mantissa ed identifica le eventuali occorrenze di casi particolari attraverso appositi segnali logici.

2.1.2 Comparator

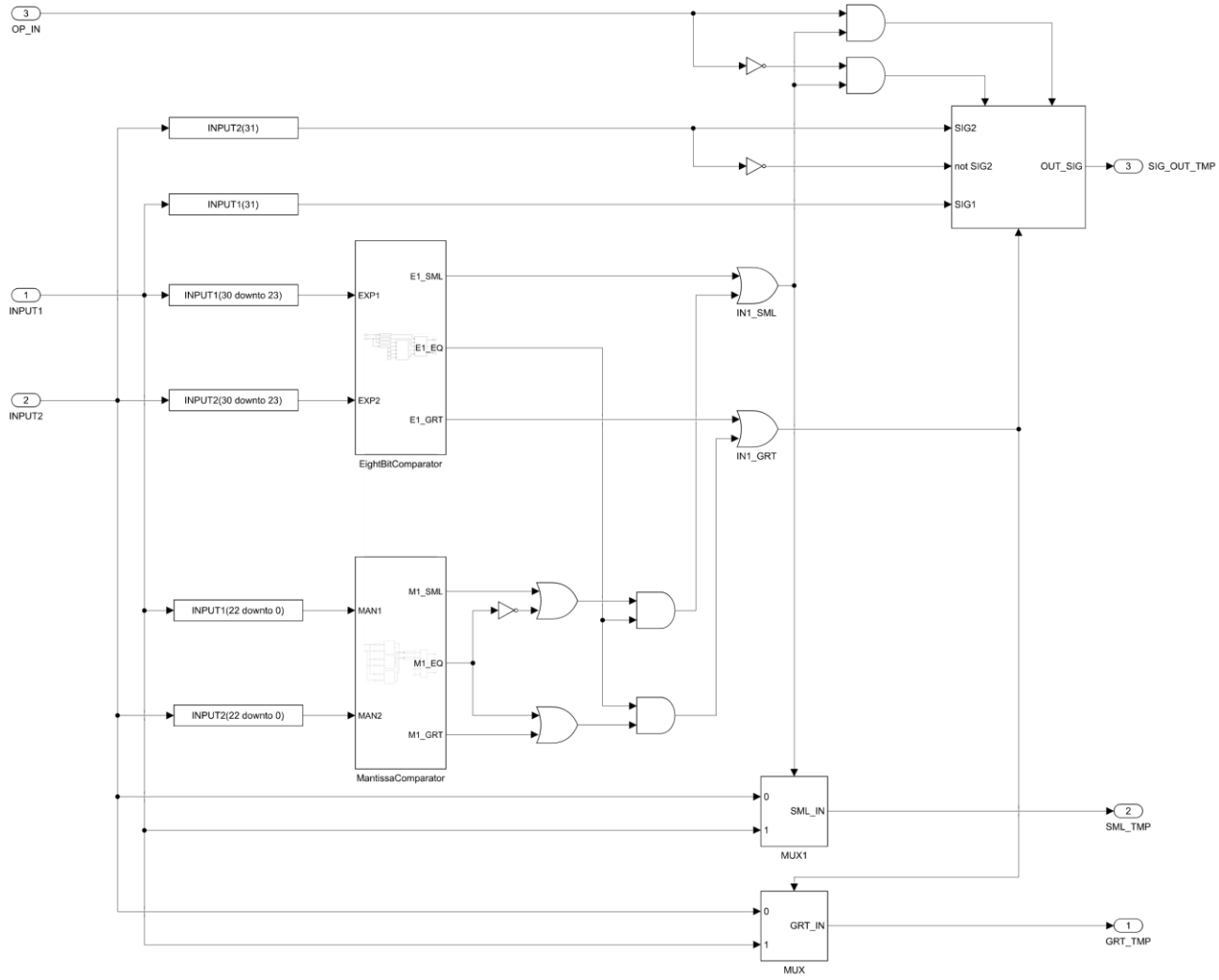


Figura 2.3: Comparator

Il modulo *Comparator* si occupa di paragonare i due numeri e determinare il segno che avrà il risultato finale del sommatore.

Viene eseguito il parsing degli input isolando segno, esponente e mantissa. Tale suddivisione consente di eseguire un paragone tra i due numeri in valore assoluto seguendo tale logica:

```
IN1_GRT  <= '1' when (E1_GRT or (E1_EQ and (M1_GRT or M1_EQ)))             = '1' else
      '0';
IN1_SML  <= '1' when (E1_SML or (E1_EQ and (M1_SML or (not M1_EQ)))) = '1' else '0';
```

Il primo operando risulta essere il più grande tra i due nel caso in cui avesse l'esponente maggiore oppure, a parità di esponente, mantissa maggiore. Risulta, invece, essere il più piccolo nel caso in cui avesse l'esponente minore oppure, a parità di esponente, mantissa minore.

Convenzionalmente si è deciso di considerare il primo numero come maggiore anche nel caso in cui i due risultino effettivamente uguali.

Il segno del risultato finale viene calcolato seguendo tale logica:

```
OUT_SIG  <=     SIG1          when IN1_GRT = '1'
         else   (not SIG2)    when (IN1_SML and OP_IN) = '1'
         else   SIG2          when (IN1_SML and (not OP_IN)) = '1';
```

- Coincide con il segno del primo operando, indipendentemente dall'operazione da effettuare, se questo è il più grande tra i due.
- Coincide con l'inverso del segno del secondo operando se questo è il più grande tra i due e l'operazione da eseguire è una differenza.
- Coincide con il segno del secondo operando se questo è il più grande tra i due e l'operazione da eseguire è una somma.

2.2 Stadio di somma

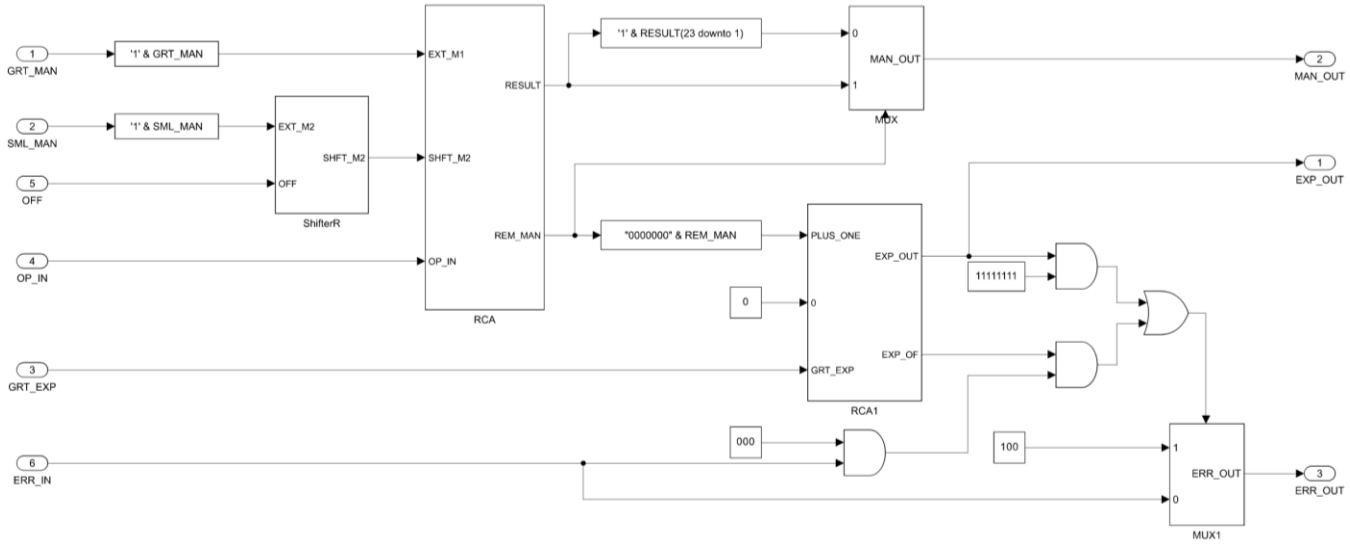


Figura 2.4: SummingStage

Nel secondo stadio della pipeline avviene la traslazione della mantissa corrispondente al numero più piccolo affiché essa risulti allineata a quella del più grande, l'operazione tra le due mantisse e l'eventuale identificazione dell'overflow dell'esponente.

Per entrambe le mantisse viene esplicitato l'hidden bit '1' prima di eseguire alcuna operazione su di esse.

La mantissa corrispondente al numero più piccolo viene traslata di un numero di posizioni coincidente con la differenza tra i due esponenti calcolata nello stadio di pipeline precedente.

Nel caso in cui si verificasse un overflow, la mantissa risultato dell'operazione viene traslata di un posto a destra e l'esponente viene corretto.

Se durante la correzione dell'esponente quest'ultimo assume valore "11111111" si segnala l'avvenuta del caso particolare per il quale il risultato finale del sommatore viene forzato a $+\infty$ (solo se precedentemente non sia già stato segnalata un'anormalità).

2.3 Stadio di correzione

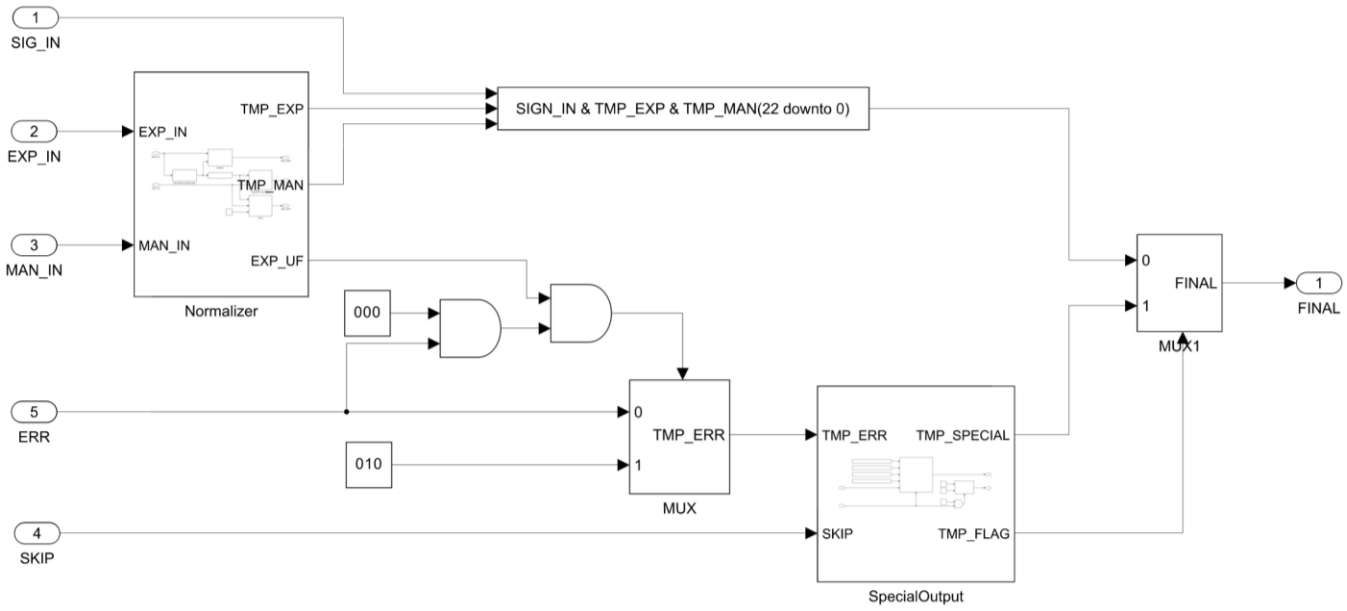


Figura 2.5: AdjustingStage

Nel terzo stadio della pipeline avviene la normalizzazione della mantissa risultato dell'operazione, la verifica dell'avvenimento di un underflow e la decodifica del segnale di casi particolari per la produzione del risultato finale del sommatore.

2.3.1 Normalizer

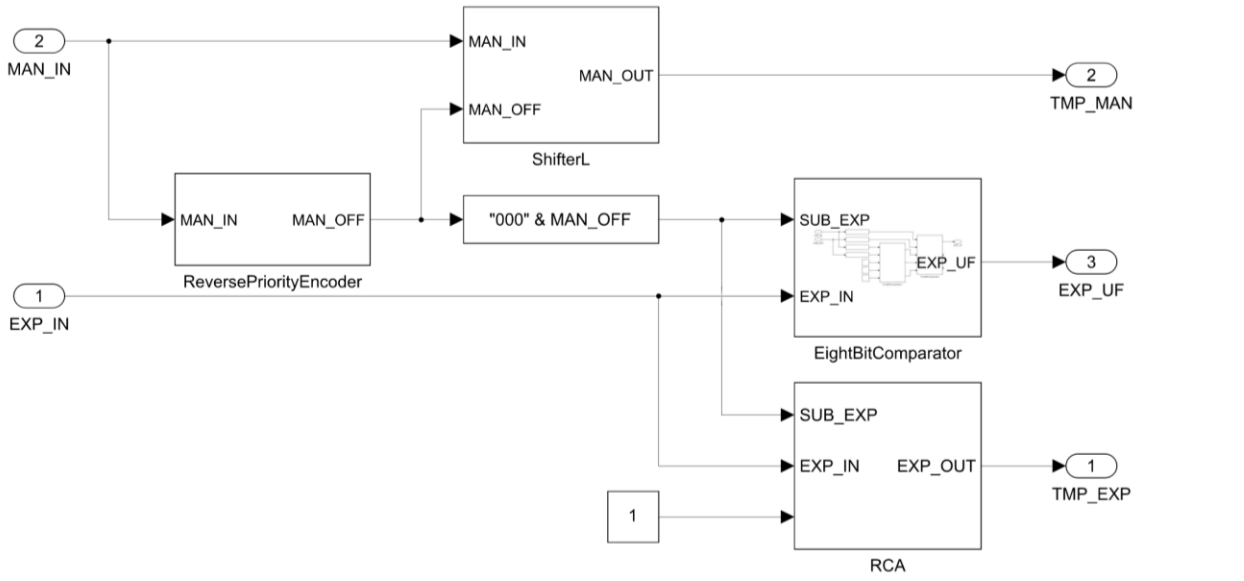


Figura 2.6: Normalizer

Il modulo *Normalizer* ha il ruolo di portare la mantissa nuovamente in notazione scientifica traslandola a sinistra fino a che il suo MSB non sia un "1" ed eventualmente correggere l'esponente. Il modulo prende in ingresso la mantissa risultato dell'operazione avvenuta nello stadio di pipeline precedente *MAN_IN* e l'esponente ad essa associato *EXP_IN*.

La computazione avviene nei seguenti moduli:

- Il modulo ***ReversePriorityEncoder*** si occupa di calcolare il numero di posizioni che allontanano il MSB della mantissa dal suo "1" più significativo generando il segnale di 5 bit *MAN_OFF* che indica di quanto dovrà essere traslata.
- Il modulo ***ShifterL*** si occupa di traslare la mantissa a sinistra.
- Si verifica che la traslazione sia effettivamente realizzabile paragonando con un ***EightBitComparator*** l'esponente *EXP_IN* con il segnale di offset *MAN_OFF* (esteso ad 8 bit diventando il segnale *SUB_EXP*). Nel caso in cui l'offset risulti maggiore dell'esponente viene alzato il segnale *EXP_UF*, output del modulo, che indica l'avvenuta di un underflow.
- Si corregge l'esponente *EXP_IN* sottraendogli il segnale di offset *SUB_EXP* attraverso un ***RCA***.

2.3.2 SpecialOutput

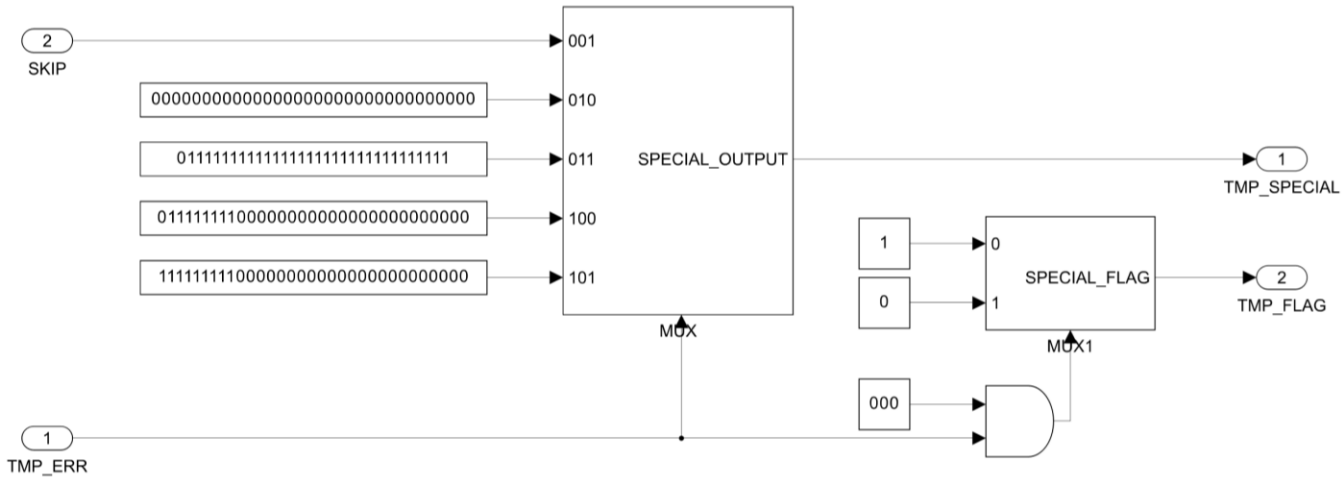


Figura 2.7: SpecialOutput

Il modulo *SpecialOutput* si occupa, in ultima istanza, di decodificare il segnale *ERR* generando in uscita il corrispondente segnale speciale *SPECIAL_OUTPUT*.

Nel caso in cui *ERR* valga "000", ovvero nel caso in cui non sia avvenuta alcuna anormalità nel corso della computazione, si segnala di ignorare l'output di questo modulo ponendo a "0" il segnale di uscita *SPECIAL_FLAG*. In tale situazione il segnale *SPECIAL_OUTPUT* perde rilevanza.

In caso contrario, *SPECIAL_FLAG* viene posto a "1", segnalando di forzare il risultato finale del sommatore al segnale *SPECIAL_OUTPUT*.

3. Pipeline

3.1 Struttura della pipeline

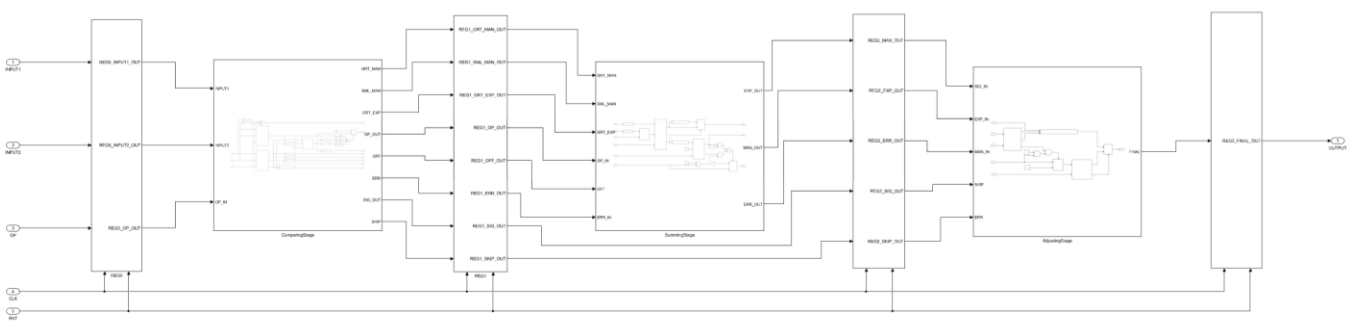


Figura 3.1: Pipeline

La pipeline è composta da tre stadi e quattro registri.

Si tiene conto di eventuali anormalità avvenute durante la computazione attraverso il segnale *ERR*, propagato attraverso tutti gli stadi e i registri rimanenti.

Il primo registro ha il compito di acquisire i dati in ingresso e trasmetterli alla prima fase.

Il secondo registro riceve i segnali generati dalla logica del primo stadio e trasmette al secondo stadio i segnali necessari alla somma dei due numeri. I segnali relativi al segno del risultato finale e al valore di *SKIP* sono direttamente propagati al terzo registro.

Il terzo registro trasmette all'ultimo stadio il risultato dell'operazione affinché questo venga normalizzato. In caso di avvenuta anormalità il segnale finale viene forzato secondo la codifica dettata da *ERR*.

Il quarto registro riceve il segnale corrispondente al risultato finale e lo propaga in uscita.

I registri sono sensibili al fronte di salità del segnale di clock.

Il segnale di reset, sincrono al clock, se attivo, assegna ai segnali di uscita dei registri il valore zero.

Poiché il periodo minimo osservato per ogni stadio ricade nel range $(8 \div 10)ns$ si è deciso di dimensionare il periodo di clock a $40ns$.

3.2 Bilanciamento della pipeline

Il carico risulta abbastanza equidistribuito tra gli stadi di pipeline.

3.2.1 Primo stadio

Timing Summary:

Minimum period: 8.800ns (Maximum Frequency: 113.632MHz)
Minimum input arrival time before clock: 3.607ns
Maximum output required time after clock: 3.634ns

3.2.2 Secondo stadio

Timing Summary:

Minimum period: 9.614ns (Maximum Frequency: 104.016MHz)
Minimum input arrival time before clock: 3.539ns
Maximum output required time after clock: 3.597ns

3.2.3 Terzo stadio

Timing Summary:

Minimum period: 8.906ns (Maximum Frequency: 112.278MHz)
Minimum input arrival time before clock: 3.573ns
Maximum output required time after clock: 3.597ns

4. Test bench

4.1 Casi particolari

- TEST: $+\infty + ZERO$
  Expected binary: "0 11111111 00000000000000000000000"
  Expected decimal: $+\infty$
  INPUT1 <= "0 00000000 00000000000000000000000" $ZERO$
  INPUT2 <= "0 11111111 00000000000000000000000" $+\infty$

- TEST: $+\infty + -\infty$
  Expected binary: "0 11111111 11111111111111111111111"
  Expected decimal: NaN
  INPUT1 <= "0 11111111 00000000000000000000000" $+\infty$
  INPUT2 <= "1 11111111 00000000000000000000000" $-\infty$

- TEST: $ZERO + ZERO$
  Expected binary: "0 00000000 00000000000000000000000"
  Expected decimal: $ZERO$
  INPUT1 <= "0 00000000 00000000000000000000000" $ZERO$
  INPUT2 <= "0 00000000 00000000000000000000000" $ZERO$

- TEST: $NaN + 17.2\,E0$
  Expected binary: "0 11111111 11111111111111111111111"
  Expected decimal: NaN
  INPUT1 <= "0 11111111 11111111011111111110000" NaN
  INPUT2 <= "0 10000011 00010011001100110011010" $17.2\,E0$

- TEST: $+\infty - +\infty$
  Expected binary: "0 11111111 11111111111111111111111"
  Expected decimal: NaN
  INPUT1 <= "0 11111111 00000000000000000000000" $+\infty$
  INPUT2 <= "0 11111111 00000000000000000000000" $+\infty$
- TEST: $+\infty - -\infty$

  Expected binary: "0 11111111 00000000000000000000000"
  Expected decimal: $+\infty$
  INPUT1 <= "01111111100000000000000000000000" $+\infty$
  INPUT2 <= "11111111100000000000000000000000" $-\infty$

- TEST: $17.2 - +\infty$
  Expected binary: "1 11111111 00000000000000000000000"
  Expected decimal: $-\infty$
  INPUT1 <= "0 10000011 00010011001100110011010" 17.2
  INPUT2 <= "0 11111111 00000000000000000000000" $+\infty$

4.2 Casi normali

- TEST: A < B
  Expected binary: "0 10000011 01000000000000000000000"
  Expected decimal: 20
  INPUT1: <= "0 10000000 01100110011001100110011" --2.8
  INPUT2: <="0 10000011 00010011001100110011010" –17.2

- TEST: ZERO + B
  Expected binary: "1 10000100 00011001110101110000101"
  Expected decimal: -35.23
  INPUT1 <= "0 00000000 00000000000000000000000"; -- ZERO
  INPUT2 <= "0 10000100 00011001110101110000101"; -- 35.23

- TEST: ZERO – 3.4028235 E38
  Expected binary: "1 11111110 11111111111111111111111"
  Expected decimal: -3.4028235 E38
  INPUT1 <= "0 00000000 00000000000000000000000"; -- ZERO
  INPUT2 <= "0 111111101 11111111111111111111111"; -- 3.4028235 E38

- TEST: ZERO – 1.17549435082 E-38
  Expected binary: "1 00000001 00000000000000000000000"
  Expected decimal: -1.17549435082 E-38
  INPUT1 <= "0 00000000 00000000000000000000000"; -- ZERO
  INPUT2 <= "0 00000001 00000000000000000000000"; -- 1.1754943508 E-38

- TEST: 3.4028235 E38 + 17.2
  Expected binary: "0 11111110 11111111111111111111111"
  Expected decimal: 3.4028235 E38
  INPUT1 <= "0 11111110 11111111111111111111111"; -- 3.4028235 E38
  INPUT2 <= "0 10000011 00010011001100110011010"; -- 17.2

- TEST: 3.4028235 E38 + 3.4028235 E38
  Expected binary: "0 11111111 00000000000000000000000"
  Expected decimal: +INF
  INPUT1 <= "0 11111110 11111111111111111111111"; -- 3.4028235 E38
  INPUT2 <= "0 11111110 11111111111111111111111"; -- 3.4028235 E38

- TEST: 2.11897634797 E37 + 2.5260167 E30
  Test della somma tra due numeri in cui il risultato non tiene conto del rounding del numero più piccolo.

  Expected binary: "0 11111010 11111110001000000000001"
  Expected decimal: 2.11897647473 E37
  INPUT1 <= "0 11111010 11111110001000000000000"; -- 2.11897634797 E37
  INPUT2 <= "0 11100011 11111110001000000000000"; -- 2.5260167 E30

- TEST: 2.11897634797 E37 - 2.5260167 E30
  Test della sottrazione tra due numeri in cui il risultato non tiene conto del rounding del numero più piccolo.
  Expected binary: "0 11111010 11111110000111111111111"

Expected decimal: 2.1189762212 E37
INPUT1 <= "0 11111010 11111110001000000000000"; -- 2.11897634797 E37
INPUT2 <= "0 11100011 11111110001000000000000"; -- 2.5260167 E30

- TEST: 2.0571154 E38 – 2.0571155 E38
  Test del caso di underflow.

Expected binary: "0 11111010 11111110000111111111111"
Expected decimal: 2.1189762212 E37
INPUT1 <= "0 00000001 11000000000000000000010"; -- 2.0571154 E38
INPUT2 <= "0 00000001 11000000000000000000011"; -- 2.0571155 E38