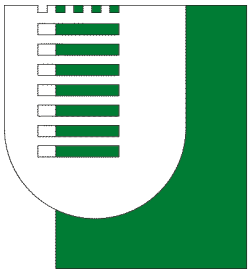


TOR VERGATA

UNIVERSITÀ DEGLI STUDI DI ROMA

Progetto per il corso di

Architetture e Sistemi VLSI per il DSP

Design di un Floating Point Fused Multiply-Add

Simone Catenacci - Matricola: 0336840

Marco Salvatori - Matricola: 0338570

Indice

Introduzione

Il progetto si propone di analizzare l'architettura del fused multiply-add (FMA), un'operazione fondamentale nel calcolo floating point che combina moltiplicazione e somma in un unico passaggio, migliorando precisione ed efficienza rispetto all'esecuzione separata delle due operazioni.

L'FMA trova applicazione in numerosi contesti che richiedono un calcolo *floating point* avanzato, grazie alla sua capacità di ridurre gli errori di arrotondamento e di migliorare l'efficienza computazionale. Gli acceleratori hardware progettati per il calcolo floating point, inclusi quelli conformi allo standard IEEE 754, sfruttano l'FMA per eseguire calcoli intensivi con prestazioni elevate, riducendo il numero di cicli e il consumo energetico.

L'architettura dell'FMA si articola in diversi blocchi funzionali, tra cui il calcolo della mantissa, l'aggiornamento dell'esponente e la gestione dell'arrotondamento. Inoltre, i blocchi dedicati alla gestione di valori speciali garantiscono un comportamento affidabile anche in presenza di *corner case*.

Un contesto particolarmente interessante per l'integrazione dell'FMA è quello delle architetture RISC-V, dove le estensioni floating point (F e D) forniscono un supporto standardizzato per il calcolo IEEE 754. RISC-V, grazie alla sua modularità e flessibilità, offre un'implementazione scalabile dell'FMA sia in sistemi ad alte prestazioni sia in dispositivi a basso consumo.

1 Floating Point

Il formato *floating point* è un sistema di rappresentazione numerica utilizzato per approssimare numeri reali nei sistemi di calcolo, particolarmente utile per i calcoli scientifici e ingegneristici dove i valori possono variare su intervalli estremamente ampi. Lo standard più utilizzato per la rappresentazione dei numeri in floating point è l'IEEE 754. Questa rappresentazione permette di superare i limiti del formato *fixed point*, che risulterebbe troppo limitata per calcoli che richiedono grande precisione o che comportano valori estremamente grandi o estremamente piccoli. Il floating point è oggi un elemento fondamentale nell'elaborazione numerica, utilizzato in ambiti che spaziano dalle simulazioni scientifiche al machine learning.

1.1 Standard IEEE 754

Nello standard IEEE 754, ogni numero è rappresentato da tre componenti (nella figura 1.1 viene mostrato il caso *single precision*):

- Segno: se il bit di segno è 0, il numero è positivo, altrimenti il numero è negativo;

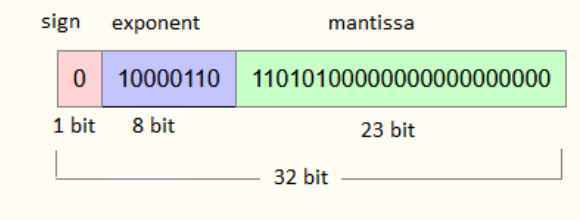


Figura 1.1: Rappresentazione *floating point* a 32 bit

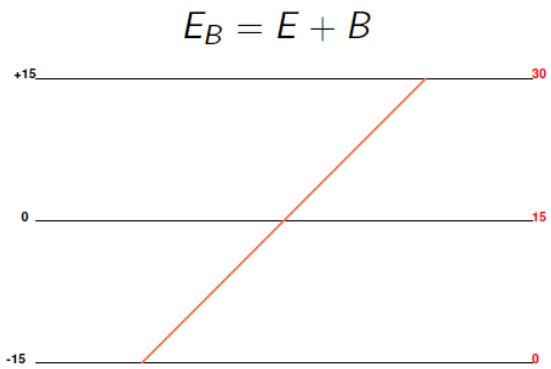


Figura 1.2: Esponente *biased*

- Esponente: specifica la scala del numero, cioè la potenza di 2 a cui moltiplicare la mantissa. L'esponente è *biased*, ovvero si somma un valore costante per garantire solo valori positivi, facilitando il confronto di numeri e la gestione degli estremi (Figura 1.2);
- Mantissa: rappresenta la parte frazionaria significativa del numero. Grazie alla normalizzazione, il bit più significativo della mantissa è implicito, consentendo di utilizzare lo spazio con maggiore efficienza;

La formula che rappresenta un numero in formato *floating point* è data da, dove b è la base (solitamente $b = 2$):

$$x = (-1)^{S_x} \cdot M_x \cdot b^{E_x}$$

I numeri *floating point* possono avere diverse precisioni e *range* diversi, in base alla quantità di bit utilizzata. Una tabella riassuntiva dei vari formati è rappresentata nella figura 1.3

Format	16	32	64	128
Storage (bits)	16	32	64	128
Precision $f = 1$ (bits)	11	24	53	113
Total exponent length (bits)	5	8	11	15
E_{max}	15	127	1023	16383
bias	15	127	1023	16383
Trailing significand f (bits)	10	23	52	112

Figura 1.3: Formati *floating point*

1.2 Rounding

Le modalità di *rounding*, sono meccanismi utilizzati per gestire l'inevitabile perdita di precisione quando un numero reale non può essere rappresentato esattamente nel formato floating point. Lo standard IEEE 754 ne prevede diverse e sono:

- Round to Nearest (Round Tie to Even): è la modalità di arrotondamento predefinita in IEEE 754 e consiste nell'arrotondare al numero più vicino. In caso di parità (quando il numero si trova esattamente a metà tra due rappresentabili);
- Round Toward Zero: questa modalità tronca semplicemente la parte decimale, arrotondando il valore verso lo zero. È un'approssimazione diretta e introduce meno oscillazioni negli errori, ma può produrre risultati sfavorevoli quando si opera con valori negativi, poiché tende a sottostimare i numeri.
- Round Toward Positive Infinity: in questa modalità, il valore viene sempre arrotondato verso l'infinito positivo. È spesso utilizzata nelle applicazioni dove è fondamentale non sottostimare un valore, come nelle operazioni finanziarie o nei calcoli di sicurezza che richiedono margini di sicurezza elevati.

- Round Toward Negative Infinity: simile alla modalità precedente, ma arrotonda sempre verso l'infinito negativo. Questa modalità è utile in contesti dove non è ammessa una sovrastima del risultato, come nei calcoli di tolleranza dei sistemi ingegneristici.

Il *rounding* comporta l'esistenza dell'errore di arrotondamento, che è uno dei principali limiti del formato floating point. A causa del numero limitato di bit disponibili per rappresentare un numero, alcuni numeri reali non possono essere rappresentati esattamente e devono essere approssimati. Questo può portare a piccoli errori che, sommati in operazioni successive, possono produrre risultati significativamente imprecisi.

2 Implementazione del sistema

L'operazione *Fused Multiply-Add* (FMA) rappresenta un'ottimizzazione nell'aritmetica floating point, in particolare per quelle applicazioni che richiedono calcoli ad alta precisione e rapidità. L'FMA permette di eseguire, in un'unica istruzione, la moltiplicazione di due numeri e l'addizione di un terzo numero:

$$z = x \cdot y + w$$

In questo lavoro, si è presa ad esempio l'architettura contenuta in [1] e quindi, come si evince dalla figura 2.1, si è andati a realizzare una struttura composta essenzialamente da tre moduli, ognuno dei quali calcola una componente del formato floating point. Per l'implementazione effettiva dei moduli e delle varie operazioni effettuate in essa, ci si è basati sulle architetture presenti in [4] e [8], andando, però, a fare alcune modifiche. Così facendo si è ottenuta l'architettura finale presente in figura 2.2. A questo, è dovereso procedere con un analisi approfondita dei vari blocchi che compongono l'architettura implementata.

2.1 Mantissa

Nell'FMA, il calcolo della mantissa, ovvero la parte significativa della rappresentazione *floating point*, viene calcolata combinando la moltiplicazione delle mantisse degli operandi x e y con la somma del terzo operando w. Questa sequenza è realizzata

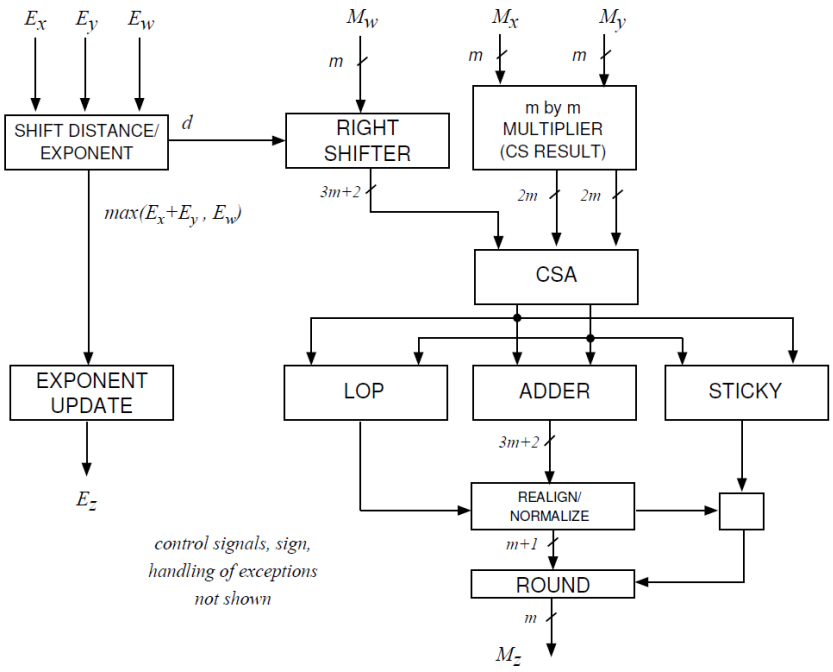


Figura 2.1: FMA da *Digital Arithmetic*

tramite un'unità aritmetica dedicata alla gestione delle mantisse, in cui un moltiplicatore esegue il prodotto tra x e y, generando un risultato esteso per evitare perdite di informazione. Successivamente, questo valore viene sommato alla mantissa del terzo operando w in un accumulatore, il quale mantiene la completezza del risultato fino alla fase finale di rounding.

2.1.1 Multiplier

La moltiplicazione tra le mantisse degli operandi x e y, viene eseguita attraverso un moltiplicatore con *carry-save result*. Quest'ultimo è stato implementato andando a realizzare un *Unsigned Multipler Array* (Figure 2.3), ma "eliminando" il *Carry Propagate Adder* presente in uscita.

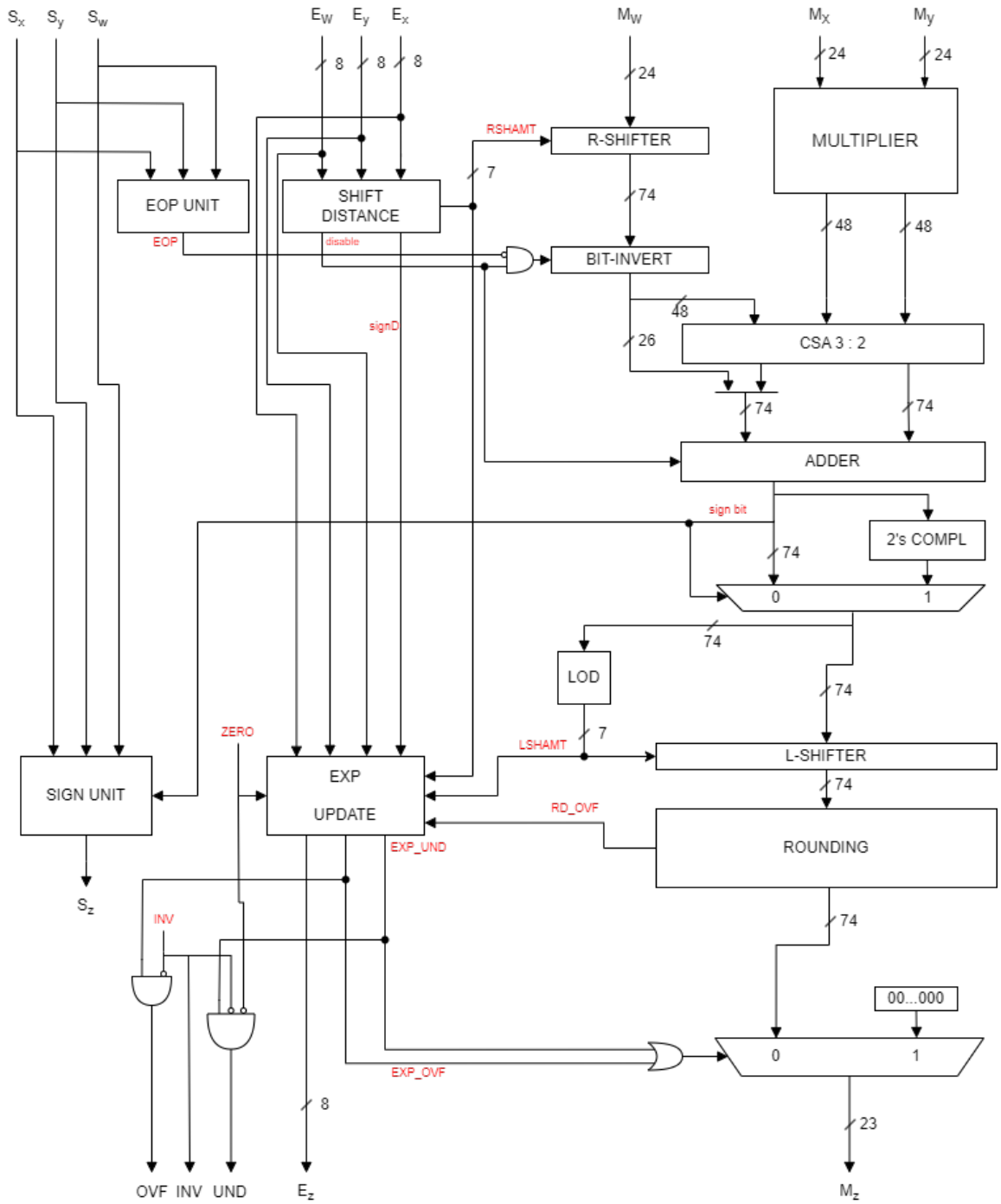


Figura 2.2: Architettura FMA

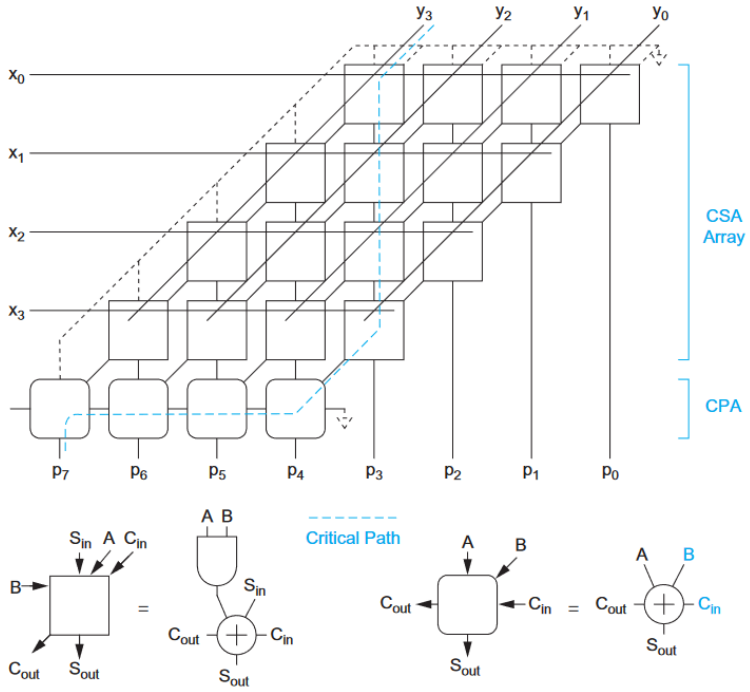


Figura 2.3: Unsigned Array Multipler (presente in [11])

2.1.2 Adders & Alignment Shifter

A questo punto, per effettuare l'operazione di addizione con il terzo operando, è necessario *shiftare* verso destra la mantissa M_w. La quantità di cui bisogna *shiftare* M_w è data dalla seguente formula (dove B è il bias e m è il numero di bit della mantissa più l'hidden bit):

$$RSHAMT = (E_x + E_y - B) - E_w + m + 3$$

La figura 2.4 rende più chiara l'operazione di allineamento effettuata dal blocco **R-SHIFTER** (se l'operazione effetiva è la sottrazione viene effettuata anche un'operazione di inversione su M_w, ma di questo se ne parlerà più avanti nella trattazione).

Adesso non rimane che sommare il terzo operando al prodotto tra x e y, e ciò viene effettuato combinando un *Carry-Save Adder* 3:2 (Figura 2.5) e un *Adder One's Com-*

Initial position:

```
                ------ m ------  -------- 2m ------------
Product x*y:                  00xxxxxxxx....xxxxxxxxxxx
Addend:         1.xxxxxxxxxxxx
                 |--- m-1+4 -------|
                                  |--- sticky ------------|
                                       region
```

Alignment when $E_{xy} = E_w$:

```
                ------ m ------  -------- 2m --------------
Product x*y:                  00xx.xxxxx....xxxxxxxxxxxxxx
Addend:                          1.xxxxxxxxxxxx
                                                  |-sticky|
Shift distance: |--- m-1+4 -------|
```

Figura 2.4: Allineamneto del terzo operando

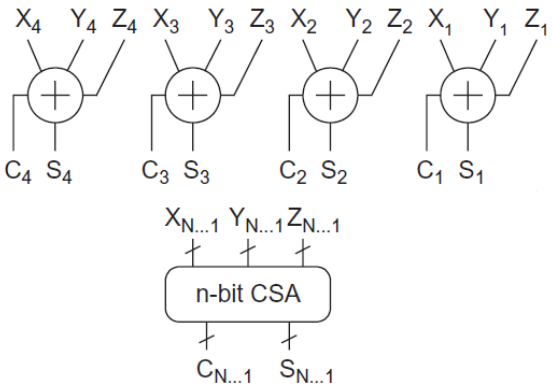


Figura 2.5: N-bit Carry-Save Adder (presente in [11])

plement. Quest'ultimo è stato implementato con un semplice *adder* in complemento a due, che applica il complemento a due sul risultato se esso è negativo.

2.1.3 Leading Ones Detector

Dopo aver svolto l'addizione tra gli operandi, bisogna, se necessario, normalizzare il risultato ottenuto. Questa operazione viene realizzata tramite un *Leading Ones Detector* (Nello schema in Figura 2.2, il blocco **LOD**), che va semplicemente a calcolare il numero di zeri dopo la virgola presenti nel risultato. Quindi, per effettuare

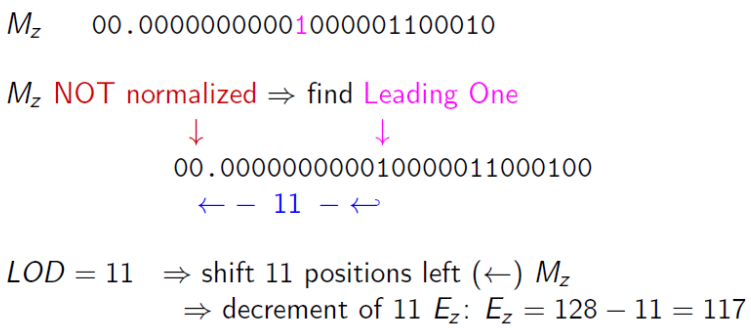


Figura 2.6: Operazione di normalizzaione

la normalizzaione, il risultato viene *shiftato* verso sinistra della quantità calcolata dal *Leading Ones Detector*. Questa operazione viene mostrata in Figura 2.6.

2.1.4 Rounding

L'ultimo blocco del modulo relativo al calcolo della mantissa M_z è quello dedicato all'arrotondamento. Esistono varie modalità di arrotondamento, per questa implementazione è stata scelta la modalità *Round to the Nearest*. In questo modello di arrotondamento, tutti i numeri vengono arrotondati al valore rappresentabile più vicino; se ci si trova esattamente a metà tra due rappresentazioni, il risultato viene arrotondato al numero pari più vicino. Per questa modalità di arrotondamento, è necessario conoscere il round-bit, il guard-bit, lo sticky-bit e il bit meno significativo (LSB) del risultato normalizzato.

Il guard-bit è il primo bit dopo l'LSB del risultato. Se questo bit è 1, i bit scartati (incluso il guard-bit stesso) hanno almeno metà del peso dell'LSB (cioè, un'unità di ultima posizione, o ULP). In caso contrario, il numero rappresentabile più vicino è sempre il risultato normalizzato troncato. In tutti gli altri casi, è necessario esaminare il round e lo sticky-bit per determinare se selezionare il numero più vicino o il numero pari più vicino. Se il round-bit o lo sticky-bit è 1 in combinazione con il guard-bit

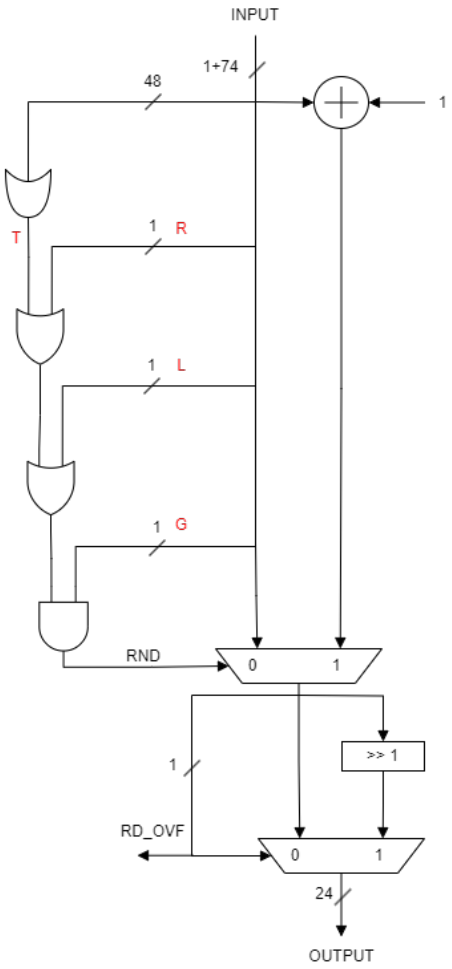


Figura 2.7: Implementazione RTNE

uguale a 1, i bit scartati hanno un peso superiore a metà di un ULP e il risultato dovrà essere incrementato. Se nessuno dei casi sopra è vero, l'algoritmo deve trovare il numero pari più vicino.

In Figura 2.7 viene mostrata l'implementazione dell'operazione spiegata sopra. Inoltre, sebbene non vengano mostrati i loro schemi logici, sono state implementate anche altre modalità di rounding, ovvero: *Rounding to Zero*, *Rounding to Plus* ∞ e *Rounding to Minus* ∞.

2.2 Esponente

Il modulo che calcola l'esponente del risultato finale è diviso, essenzialmente, in due blocchi: **SHIFT DISTANCE** e **EXPONENT UPDATE**.

2.2.1 Shift Distance

Il blocco **SHIFT DISTANCE** (Figura 2.8) calcola la differenza tra gli esponenti dei due valori, determinando di quanti bit occorre spostare la mantissa verso destra per ottenere un allineamento preciso. Questo spostamento verso destra è cruciale per mantenere la precisione dell'operazione e minimizzare la perdita di informazioni significative nei bit meno significativi. Questa quantità è data dalla formula:

$$D = (E_x + E_y - B) - E_w \longrightarrow RSHAMT = D + m + 3$$

2.2.2 Exponent Update

Il blocco **EXPONENT UPDATE** (Figura 2.9) è responsabile dell'aggiornamento dell'esponente del risultato finale. Dopo l'allineamento delle mantisse e l'esecuzione dell'operazione di moltiplicazione e somma, il blocco exponent update verifica se il risultato ottenuto necessita di un aggiustamento dell'esponente, specialmente nel caso di valori che richiedono una normalizzazione. Se il risultato supera la gamma rappresentabile o si posiziona al di sotto dei limiti di precisione, il blocco aggiorna l'esponente in modo da riflettere il corretto ordine di grandezza del numero finale. Inoltre, **EXPONENT UPDATE** tiene conto di eventuali condizioni attivando meccanismi di gestione per adattare il risultato finale secondo lo standard IEEE 754.
Il blocco effettua il seguente algoritmo sulla base dei valori delle esponenti dei singoli

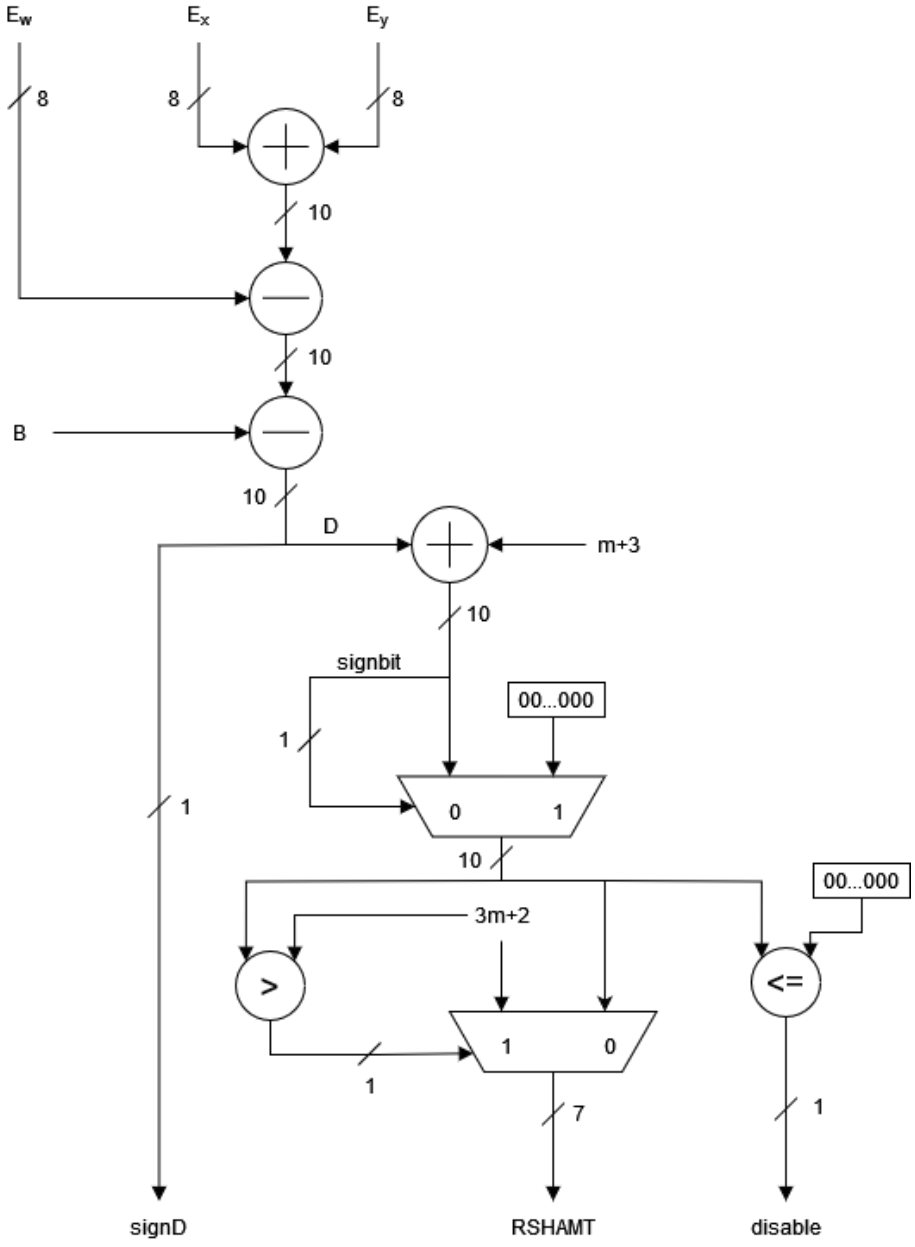


Figura 2.8: Blocco **SHIFT DISTANCE**

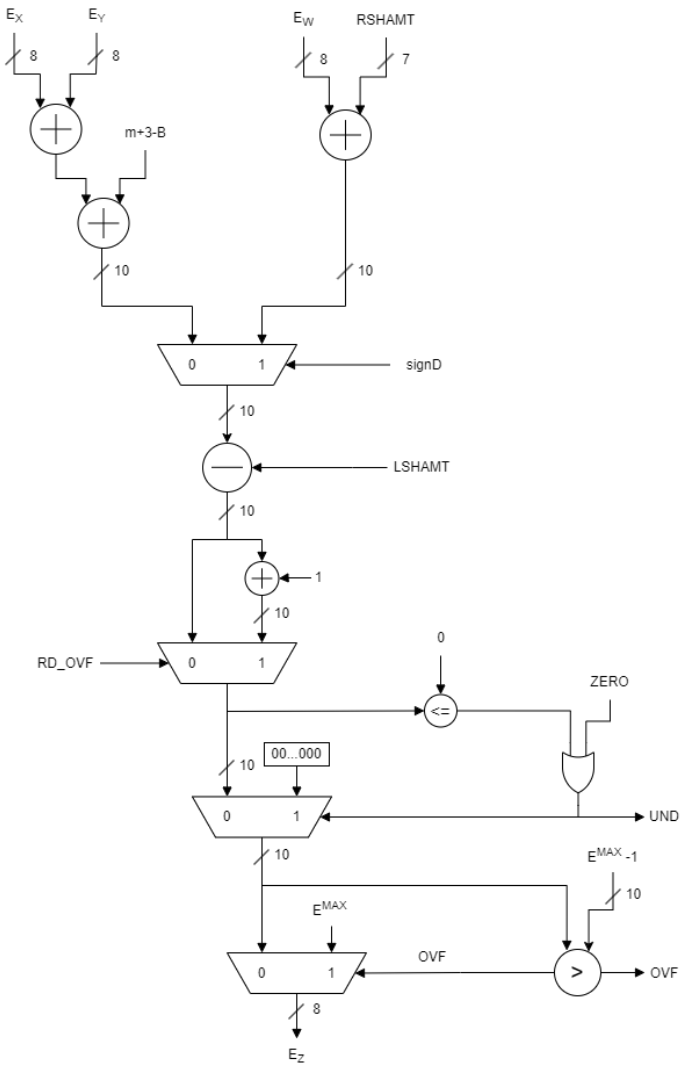


Figura 2.9: Blocco **EXPONENT UPDATE**

operandi:

$$E_x + E_y > E_w \longrightarrow E_z = E_x + E_y - B - [LSHAMT - (m+3)]$$

$$E_x + E_y < E_w \longrightarrow E_z = E_w + RSHAMT - LSHAMT$$

Inoltre, l'esponente viene aumentato di 1 se vi è *rounding overflow*, evento che viene segnalato dal blocco **ROUNDING**.

Per quanto riguarda i le condizioni speciali, il sistema rivela se il risultato finale rientra tra alcuni degli *special values*, ovvero: $\pm\infty$, attivando il segnale di *overflow*, o se il numero non è rappresentabile o pari a zero, attivando il segnale di *underflow* (questo segnale include anche il caso di un numero *subnormal*). Se uno di questi segnali è attivo la mantissa (tramite un multiplexer) viene *flushato* a zero.

2.3 Segno & Effective Operation

Le unità dedicate al calcolo del segno e alla determinazione dell'*effective operation* giocano un ruolo centrale nella gestione corretta delle operazioni aritmetiche, poiché stabiliscono come combinare il risultato della moltiplicazione e il valore dell'addendo. L'operazione FMA calcola $(x \cdot y) + w$, ma la specifica natura dell'operazione additiva tra il prodotto $x \cdot y$ e l'addendo w può variare in base ai segni degli operandi e ai rispettivi valori assoluti. Di fatto, l'*effective operation* identifica le condizioni in cui il risultato finale si ottiene sommando o sottraendo il valore del prodotto e quello dell'addendo, mantenendo così la correttezza dell'operazione. Se $|x| \cdot |y| > |w|$, allora il segno del risultato è quello del prodotto; se $|x| \cdot |y| < |w|$, il segno finale cambierà. La tabella 2.1 riassume la logica dietro quello appena descritto.

Operazione	**EOP e Moduli**	**Segno Risultante**
$2+1=3$	addizione	positivo
$-2+-1=-3$	addizione	negativo
$2+-1=1$	sottrazione ($\|x \cdot y\| > \|w\|$)	positivo
$-2+1=-1$	sottrazione ($\|x \cdot y\| > \|w\|$)	negativo
$2+-3=-1$	sottrazione ($\|x \cdot y\| < \|w\|$)	negativo
$-2+3=1$	sottrazione ($\|x \cdot y\| < \|w\|$)	positivo

Tabella 2.1: Tabella per la logica del segno

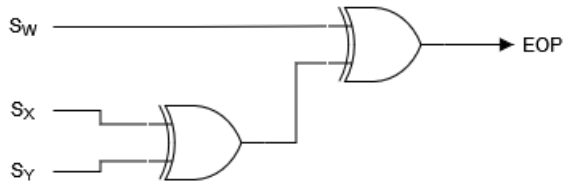


Figura 2.10: Implementazione **EOP Unit**

L'*effective operation* viene calcolata in modo semplice nel seguente modo:

$$S_z = (S_x \oplus S_y) \oplus S_w$$

Che corrisponde, a livello implementativo a due semplici porte logiche *XOR* (Figura 2.10).

Il blocco **SIGN UNIT** è stato implementato attraverso questo semplice circuito rappresentato nella figura 2.11 (presente in [8]), dove *sign sum* è il bit di segno della somma risultante fornita dall'*Adder One's Complement*.

2.4 Special Values

Il blocco **SPECIAL VALUES** ha il compito di monitorare e rispondere a situazioni in cui uno o più operandi coinvolti nell'FMA risultano essere valori speciali, come zero, NaN o infinito. Questi casi, se non trattati correttamente, potrebbero generare risultati imprevisti o errori nel calcolo. Questo blocco, nello specifico, calcola (attraverso il circuito combinatorio presente in Figura 2.12 i segnali **ZERO**, che rileva se

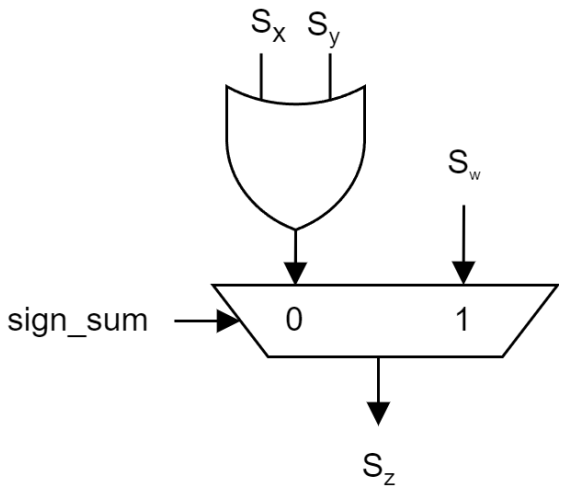


Figura 2.11: Implementazione **Sign Unit**

uno o entrambi gli input x e y sono zero e l'input w è zero, e **INV**, che rivela il caso $+\infty - \infty$ oppure se uno degli input è NaN. I segnali prodotti vengono successivamente elaborati per produrre i vari *flag* in uscita e per gestire le operazioni riguardanti esponente e mantissa nel caso di eventi speciali.

2.5 Pipelining

Dopo aver implementato il sistema è stata effettuata un'operazione di *pipelining*, sostanzialmente dividendo il circuito in 3 *stage* come mostrato in figura 2.13, al fine di migliorare le perfomance a livello di timing e di equalizzare i segnali paralleli, in modo da evitare possibili glitch in uscita.

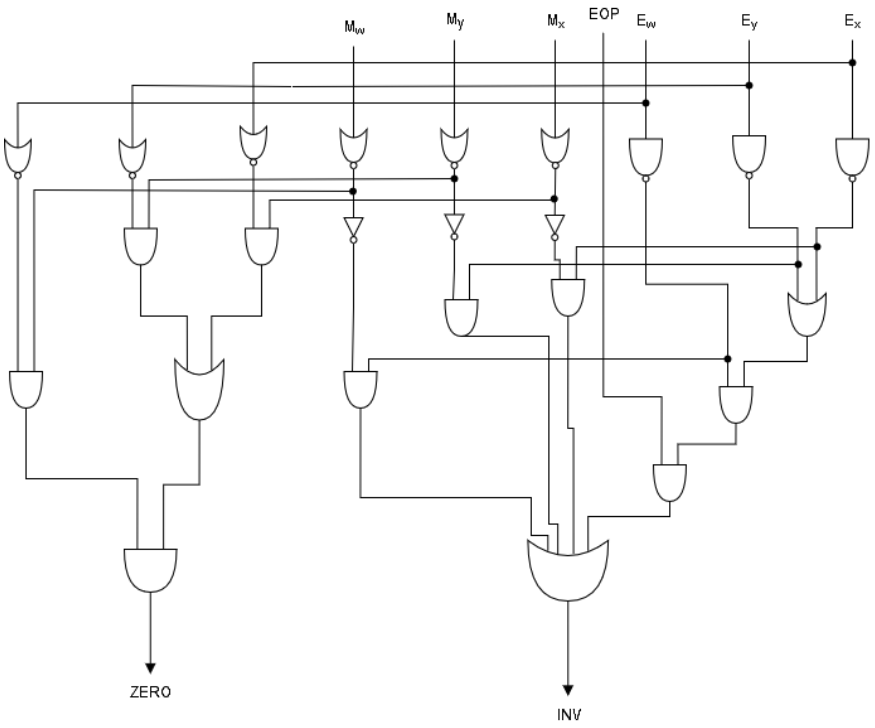


Figura 2.12: Implementazione **SPECIAL VALUES**

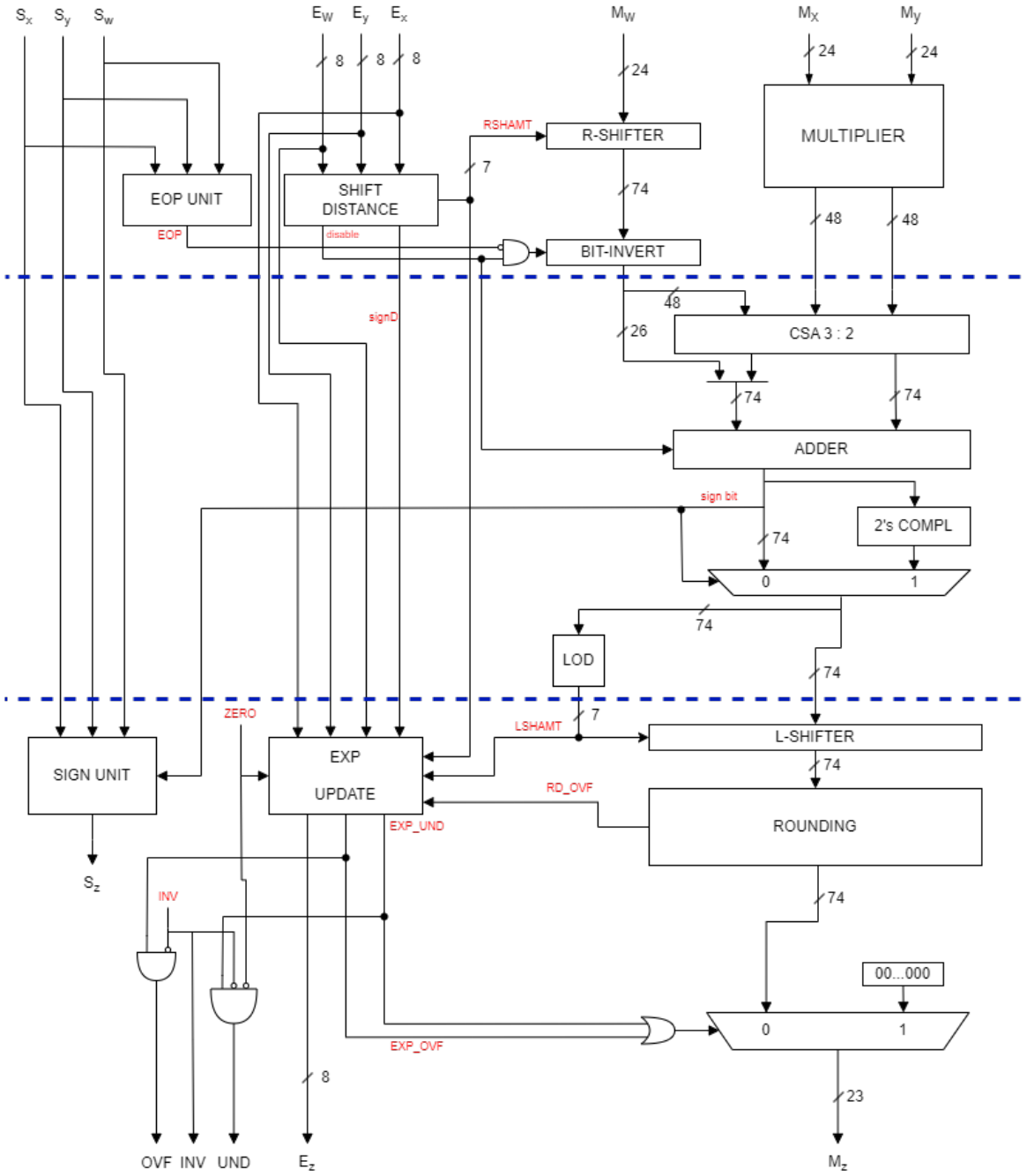


Figura 2.13: Architettura FMA con *pipeline*

3 Risultati simulativi e implementativi

3.1 Simulazione Randomica

Per testare il corretto comportamento del FMA, è stata una eseguita simulazione in cui vengono immessi degli input randomici e l'output ottenuto viene comparato degli output desiderati generati via software, come mostrato in Figura 3.1.

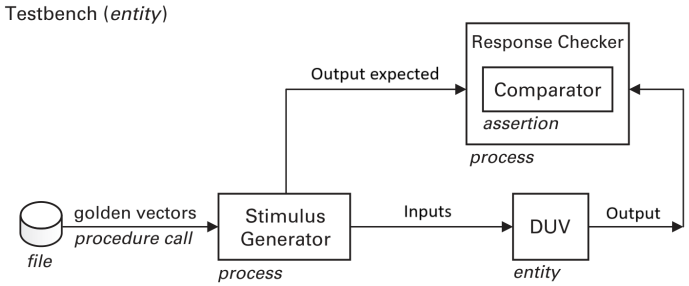
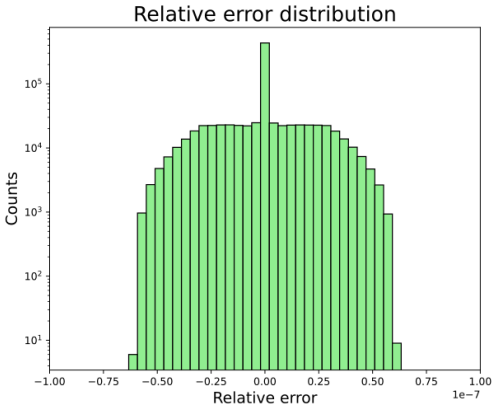


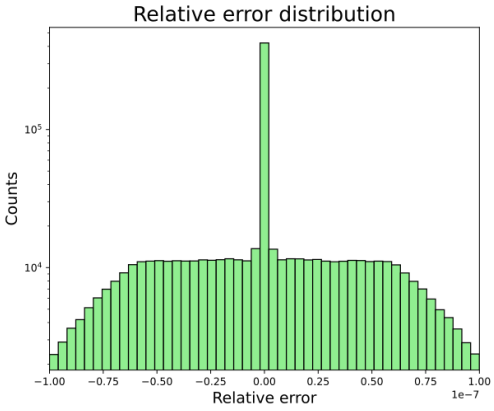
Figura 3.1: Testbench per simulazione randomica

Le specifiche per la simulazione randomica sono:

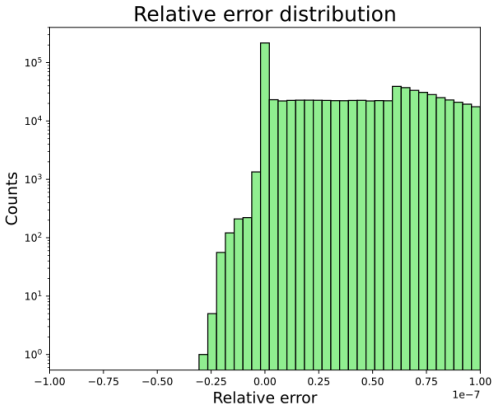
- N° di vettori: 1.000.000
- Distribuzione inputs: Uniforme
- Inputs esatti in float32, ovvero non ci sono arrotondamenti eseguiti via software per passare da float64 a float32



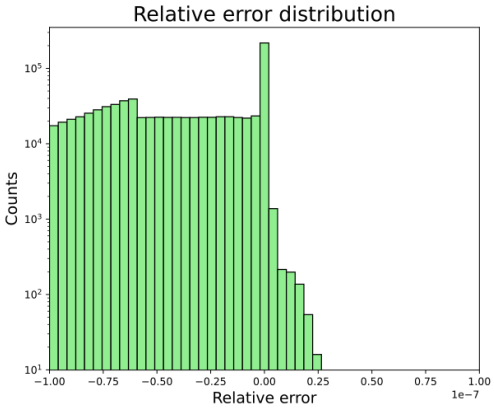
(a) PDF RTNE



(b) PDF RTZ



(c) PDF RT+∞



(d) PDF RT-∞

- Outputs desiderati calcolati in float128, per ottenere la massima precisione possibile
- 0 errori, esclusi i casi con output subnormal che non son supportati dalla nostra architettura

È stato anche stimata la probability density function dell'errore relativo, attraverso l'uso di un istogramma con 100 bin, e ciò è stato fatto per tutte le modalità di rounding Figura 3.2a-3.2d .

Inoltre, è stata calcolata la distibuzione dei bit errati in relazione alla loro posizione all'interno dell'intera *word* Figura 3.2.

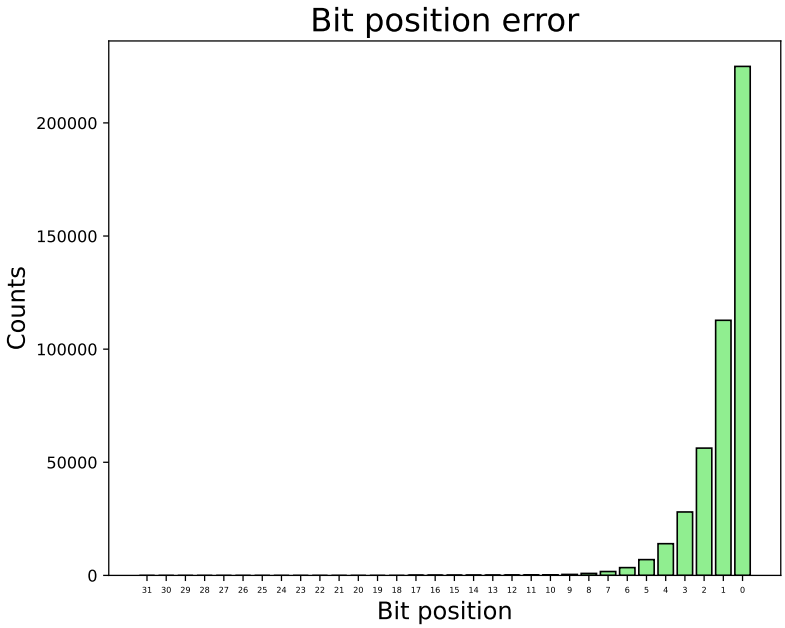


Figura 3.2: PDF posizione dei bit (caso RTZ)

In particolare, per ovvi motivi, sono state escluse tutte le uscite con valori non finiti.

L'errore relativo è stato calcolato usando la seguente formula:

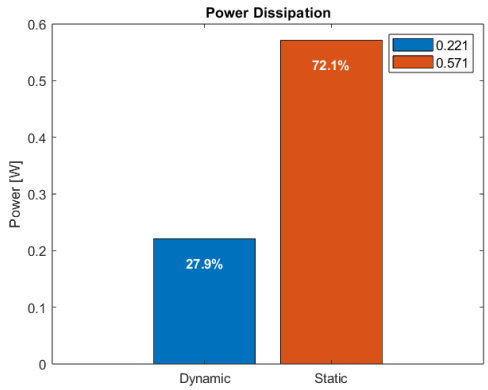
$$Error_{rel} = \frac{OUT_{exp} - OUT}{|OUT_{exp}|}$$

3.2 Simulazione Corner Cases

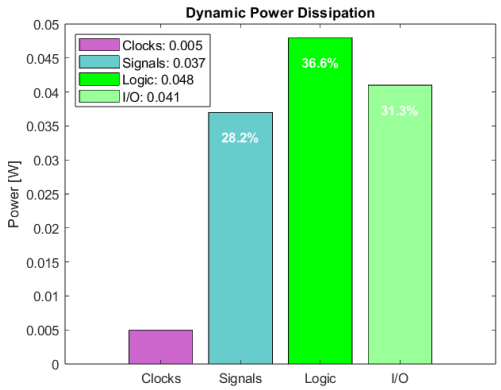
È stata anche eseguita una simulazione con degli input speciali oppure che possono portare ad output particolari, nella Tabella 3.2 sono presentati i vari casi particolari testati.

X	Y	W	OUT	OVF	UND	INV
0	0	0	0	0	0	0
-0	0	-0	-0	0	0	0
2.5	0	0	0	0	0	0
3.403e38	3.403e38	3.403e38	∞	1	0	0
1.175e-38	1.175e-38	0	0	0	1	0
∞	2.5	2.5	∞	1	0	0
$-\infty$	2.5	2.5	$-\infty$	1	0	0
∞	∞	∞	∞	1	0	0
$-\infty$	$-\infty$	∞	∞	1	0	0
∞	2.5	$-\infty$	NaN	0	0	1
∞	$-\infty$	∞	NaN	0	0	1
NaN	2.5	2.5	NaN	0	0	1
NaN	NaN	NaN	NaN	0	0	1

Tabella 3.2: Tabella di input output per i corner cases del FMA



(a) Dissipazione di potenza



(b) Dissipazione di potenza dinamica

Infine, è stato implementato il progetto su una FPGA prodotta dalla *Xilinx*: Zynq UltraScale+ MPSoCs. Sono state ottenute le dissipazioni di potenza mostrate in Figura 3.3a e 3.3b ed è stata ottenuta una frequenza massima di clock pari a circa 333 MHz.

4 Implementazione Caso Subnormal

Nel contesto dell'aritmetica *floating point*, i numeri *subnormal* rivestono un ruolo fondamentale nel garantire una rappresentazione continua dei valori molto vicini a zero. Introdotti dallo standard IEEE 754, i numeri *subnormal* permettono di estendere la gamma rappresentabile oltre il limite minimo dei numeri normalizzati, evitando un comportamento noto *underflow*.

4.1 Subnormals

I numeri *subnormal* si differenziano dai numeri normalizzati in quanto:

- L'esponente è fissato zero (senza il bias);
- La mantissa non include l'*hidden bit* (il bit implicito che nei numeri normalizzati vale sempre 1);

La formula per i numeri *subnormal* diventa quindi:

$$subn = (-1)^S \cdot M \cdot b^{B-(E-1)}$$

L'implementazione dei numeri *subnormal* richiede circuiti aggiuntivi per gestire la rilevazione e l'elaborazione della mantissa senza l'*hidden bit*. Questo aumenta la complessità del design dei processori e introduce:

- Maggiore latenza nelle operazioni aritmetiche;
- Maggiori costi in termini di area dei circuiti;

Le operazioni con numeri *subnormal* sono generalmente più lente rispetto a quelle con numeri normalizzati, a causa della logica addizionale necessaria per eseguire calcoli corretti. Per questo motivo, alcune architetture (specialmente nei sistemi embedded) disabilitano la gestione completa dei numeri *subnormal*, impostando i risultati a zero (modalità *flush-to-zero*).
Inoltre, la manipolazione dei numeri *subnormal* è più dispendiosa in termini di energia. In applicazioni ad alte prestazioni, come l'elaborazione grafica e l'intelligenza artificiale, ciò può rappresentare un ostacolo significativo.

4.2 Architettura Subnormals

Siccome la gestione accurata dei *subnormals*, che rappresentano valori più vicini a zero rispetto al più piccolo valore in *floating point* normalizzato, rimane una sfida nelle implementazioni FMA tradizionali. In [12] viene proposto un design FMA innovativo che mantiene l'efficienza supportando al contempo il calcolo *subnormal* completo, garantendo una maggiore accuratezza e conformità con lo standard IEEE 754.
Il paper presenta un'unità Fused Multiply-Add (Figura 4.1)ottimizzata che opera in 4 cicli in *pipeline*, supporta operazioni per *single precision* e *double precision*, e gestisce tutti i quattro modi di arrotondamento. A differenza delle unità floating-point tradizionali, che necessitano di gestori microcodice per i numeri *subnormal* (causando ritardi significativi), la struttura proposta elimina completamente tale necessità, supportando i numeri *subnormal* senza penalità di ritardo.
Per raggiungere questa efficienza, sono state implementate diverse ottimizzazioni:

1. **Allineamento mantissa unidirezionale**:eseguito parallelamente alla moltiplicazione, elimina la necessità di operazioni aggiuntive (CSA) e velocizza la logica di arrotondamento;

2. **Codifica Booth-Radix 16**: riduce il numero di prodotti parziali rispetto al radix-4, diminuendo area e consumo energetico senza impattare la latenza;

3. **Gestione del J-bit**: ottimizza la gestione dei numeri *subnormal* senza ritardi, combinando correzioni direttamente nel moltiplicatore;

4. **Leading Zero Anticipator**: velocizza la normalizzazione e gestisce l'underflow, riducendo la necessità di operazioni di denormalizzazione;

5. **Rilevamento sticky bits e all-ones in parallelo**: accelera la logica di arrotondamento grazie a decisioni anticipate;

6. **Integrazione del complemento a due**: il complemento per l'addizionatore principale è fuso nella logica di arrotondamento, eliminando l'uso di multiplexer aggiuntivi;

4.2.1 Exponent Difference e Alignment

Nel FMA, il primo passo è il calcolo della differenza tra l'esponente del prodotto e_M e l'esponente dell'addendo e_C (Figura 4.2. Questo valore determina lo spostamento del significando necessario per allineare gli operandi e viene determinato essenzialmente come spiegato nel Capitolo 2.
La loro gestione dei *subnormals* richiede specifici accorgimenti:

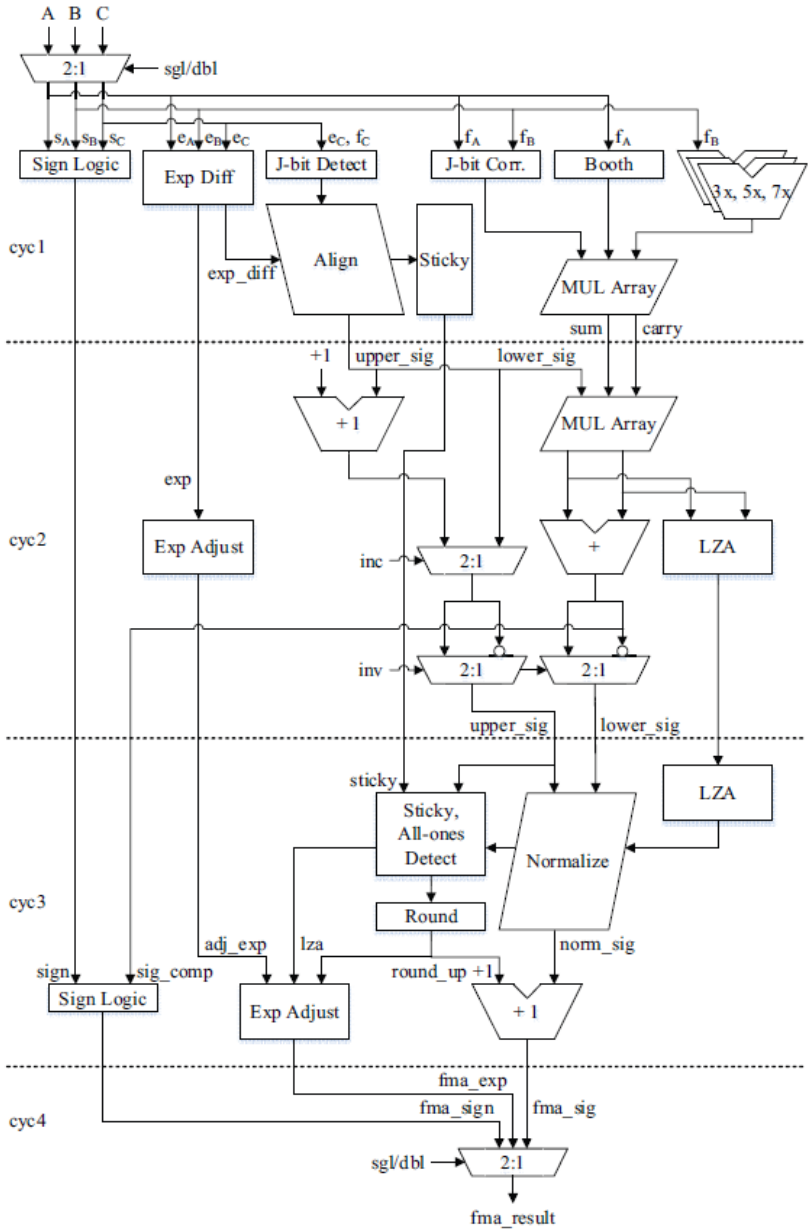


Figura 4.1: FMA con la gestione dei *subnormal*

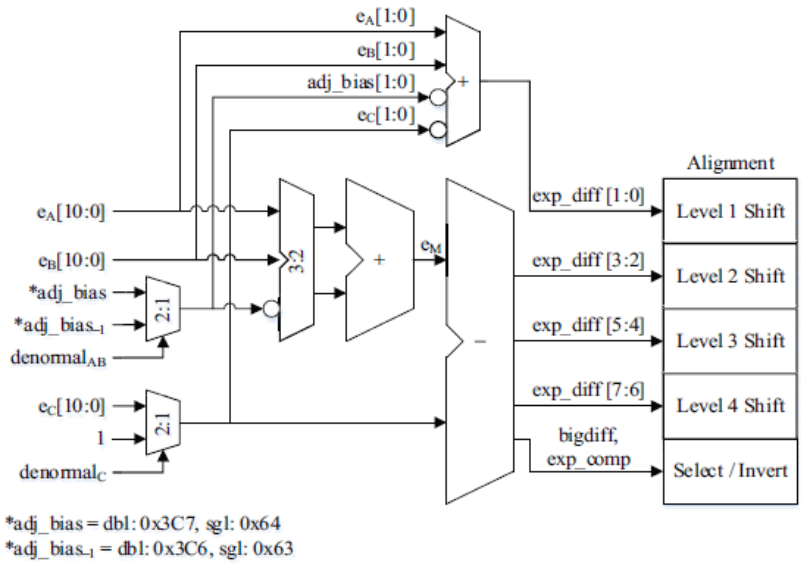


Figura 4.2: Exponent Difference

- Il J-bit, normalmente implicito e pari a 1 per i numeri normalizzati, è trattato come 0 per i numeri *subnormal*. Questo implica modifiche nel calcolo della differenza di esponente e nell'allineamento del significando;
- La differenza di esponente tiene conto di una correzione aggiuntiva per gestire i bias specifici dei numeri *subnormals*, senza introdurre ritardi nel processo;

Una volta calcolata la differenza di esponente, la mantissa dell'addendo viene spostato a destra in base alla quantità stabilita. Questo allineamento avviene attraverso un sistema di shifter multipli (Figura 4.3) organizzati su più livelli :

1. Primo livello: shift di 0, 1, 2 o 3 bit;
2. Secondo livello: shift di 0, 4, 8 o 12 bit;
3. Terzo livello: shift di 0, 16, 32 o 48 bit;
4. Quarto livello: shift di 0, 64 o 128 bit;

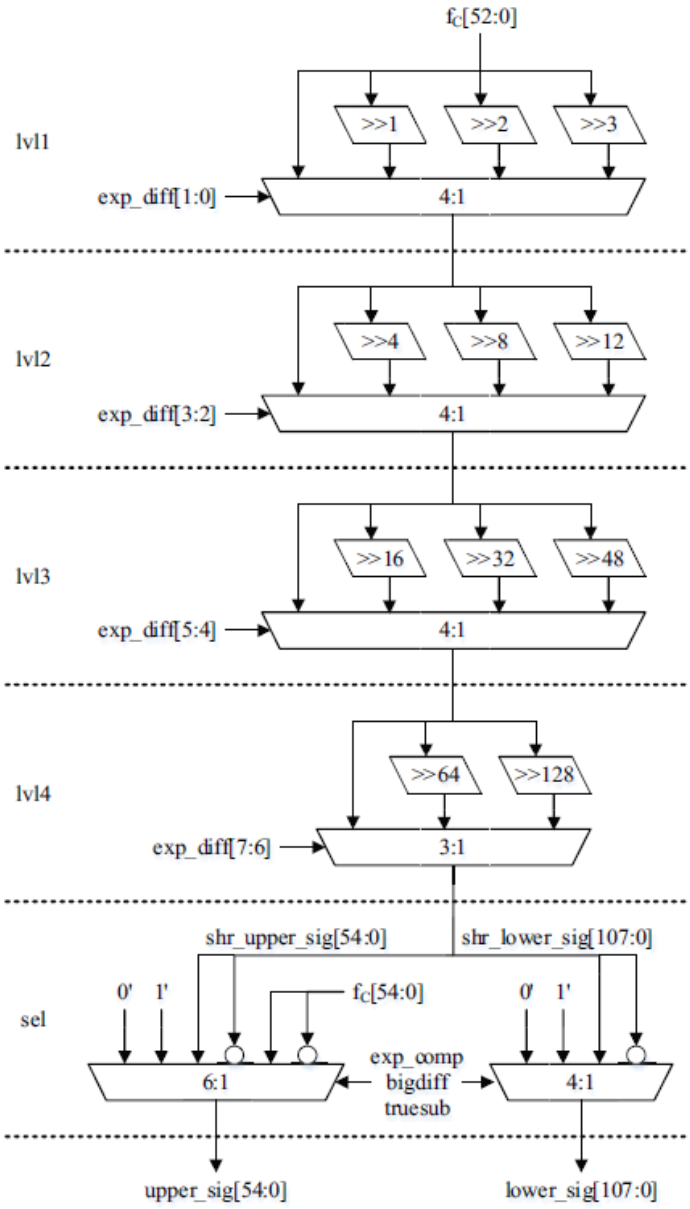


Figura 4.3: Alignment Logic

4.2.2 Multiplier e J-bit Correction

Il moltiplicatore riceve direttamente le mantisse degli operandi, bypassando la rilevazione del J-bit per evitare *delay* nel *critical path*. L'approccio adottato prevede:

- **Radix-16 Booth Encoding**: per ridurre l'area e il consumo di potenza del moltiplicatore. Questo metodo dimezza i prodotti parziali generati rispetto alla tradizionale codifica Radix-4 (14 contro 27 prodotti), eliminando due livelli di sommatore (carry-save adders, CSA) nella matrice di moltiplicazione;
- **Pre-calcolo delle moltiplicazioni**: i multipli $1x$, $2x$, fino a $8x$ delle mantisse vengono precalcolati utilizzando tre addizionatori paralleli;

Il J-bit, implicito nei numeri normalizzati, assume valore zero per i numeri *subnormal*. Per gestire questa condizione senza *delay*:

- Si assume che entrambi i J-bit siano 1, e si introduce una linea di correzione del J-bit nella matrice di moltiplicazione;
- **Linea di correzione del bit J**:
  - Se un operando è *subnormal*, la sua mantissa viene sottratta all'altro operando nella matrice;
  - Questo richiede una linea di prodotto parziale aggiuntiva e pochi bit per il complemento a due, ma tali operazioni sono integrate senza introdurre ritardi;

I prodotti parziali generati sono compressi attraverso sei livelli di alberi CSA (3:2), ottimizzati per ridurre la complessità e bilanciare il carico computazionale:

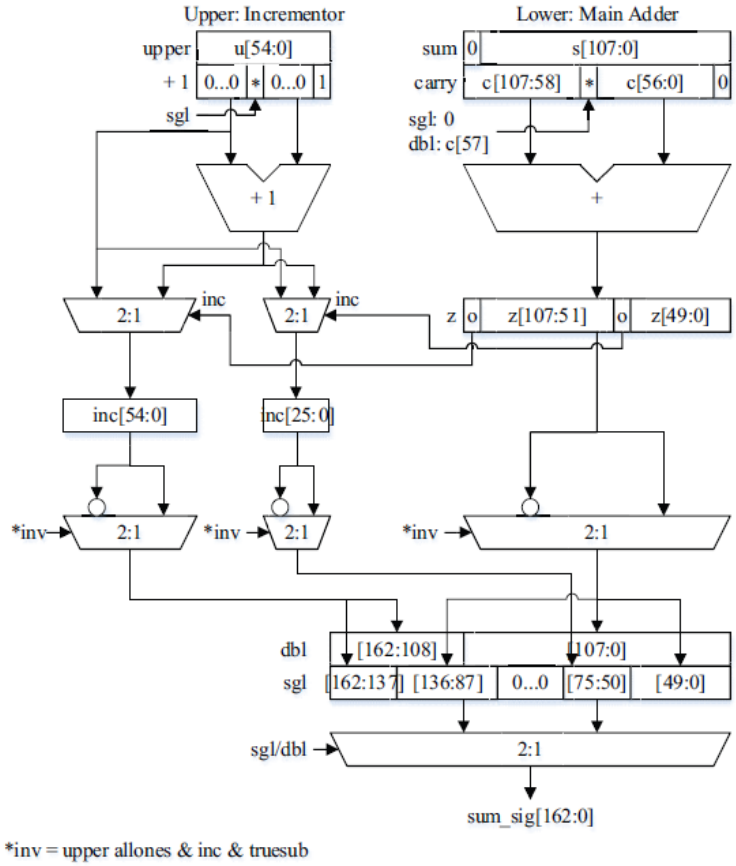


Figura 4.4: Main Adder e Incrementor

- I primi quattro livelli sono completati nel primo ciclo, mentre i restanti due livelli nel secondo ciclo;
- Le linee di correzione per il bit J e i significandi allineati vengono aggiunte al sistema CSA senza richiedere livelli aggiuntivi;

4.2.3 Main Adder and Incrementetor

L'*addder* principale (Figura 4.4) riceve i valori di somma e *carry* generati dalla matrice di moltiplicazione e calcola la mantissa risultante. L'output dell'*adder* rappresenta la mantissa preliminare, che può richiedere ulteriori correzioni o aggiustamenti, come la gestione del complemento a due.

L'incrementatore (Figura 4.4) è utilizzato per correggere eventuali *carry-out* generati dall'*adder* principale effetuando la seguente operazione:

- Aggiunge 1 al significando superiore quando il *carry-out* è presente;
- Integra il complemento a due per convertire i risultati negativi in rappresentazioni corrette;

Le ottimizzazioni implementate per l'*adder* principale e l'incrementatore apportano miglioramenti significativi sia in termini di prestazioni che di gestione dell'hardware. La riduzione della latenza è ottenuta integrando l'incrementatore e il complemento a due direttamente nella logica di arrotondamento. Questa fusione consente di eliminare componenti aggiuntivi, come i multiplexer, lungo il *critical path*, riducendo i ritardi complessivi. Inoltre, il design ottimizzato utilizza meno area del circuito, rendendolo più compatto e meno dispendioso in termini di risorse.
Questi *subnormals* sono trattati in parallelo con l'elaborazione principale. Lo *sticky bit*, che raccoglie informazioni sui bit meno significativi durante gli shift, è calcolato simultaneamente con le altre operazioni, garantendo una rappresentazione accurata dei risultati denormalizzati.

4.2.4 LZA e Normalization

Durante questa fase, la mantissa viene adattata per garantire che i risultati rientrino nel formato corretto, con il bit più significativo nella posizione appropriata.
Per velocizzare la normalizzazione, è stato implementato una Leading Zero Anticipation (LZA) modificato, che opera in parallelo con l'*adder* principale. Questa tecnica prevede la stima del numero di *leading zeros* da rimuovere dalla mantissa, consentendo di calcolare in anticipo gli shift richiesti (Figura 4.5). Inoltre, lo LZA è stata adattato

per gestire i casi di *underflow*, interrompendo il processo di normalizzazione quando l'esponente diventa negativo, evitando così una successiva denormalizzazione. Questo approccio riduce significativamente la latenza e previene ritardi aggiuntivi.

La rilevazione dello *sticky bit*, necessaria per le operazioni di arrotondamento, avviene in parallelo con la normalizzazione. Durante il processo, i bit meno significativi, che vengono eliminati durante gli shift, vengono aggregati tramite operazioni logiche OR. Lo *sticky bit* così calcolato è utilizzato direttamente nella logica di arrotondamento, riducendo ulteriormente i tempi di elaborazione.

L'arrotondamento è l'ultimo passo prima che il risultato venga memorizzato. Esso tiene conto della mantissa normalizzata, del bit di guardia, dello *sticky bit* e del modo di arrotondamento specificato.

Nel design proposto, l'arrotondamento è stato semplificato grazie alla fusione del complemento a due con la logica di rounding. Questa integrazione permette di rilevare rapidamente se il risultato deve essere incrementato, eliminando la necessità di calcoli aggiuntivi. Inoltre, il processo gestisce in parallelo eventuali overflow del significando, garantendo che il risultato rimanga entro i limiti del formato IEEE 754.

4.2.5 Exponent e Sign Logic

l calcolo dell'esponente nell'FMA comporta l'elaborazione di due valori principali: e_M, derivato alla somma degli operandi di moltiplicazione, e e_C, utilizzato per l'allineamento della mantissa.

L'esponente risultante viene determinato mediante una combinazione di operazioni logiche e matematiche (Figura 4.6):

1. **Selezione dell'esponente maggiore**: Durante il calcolo della differenza di esponenti, il valore e_M o e_C viene selezionato come base per il risultato, a seconda

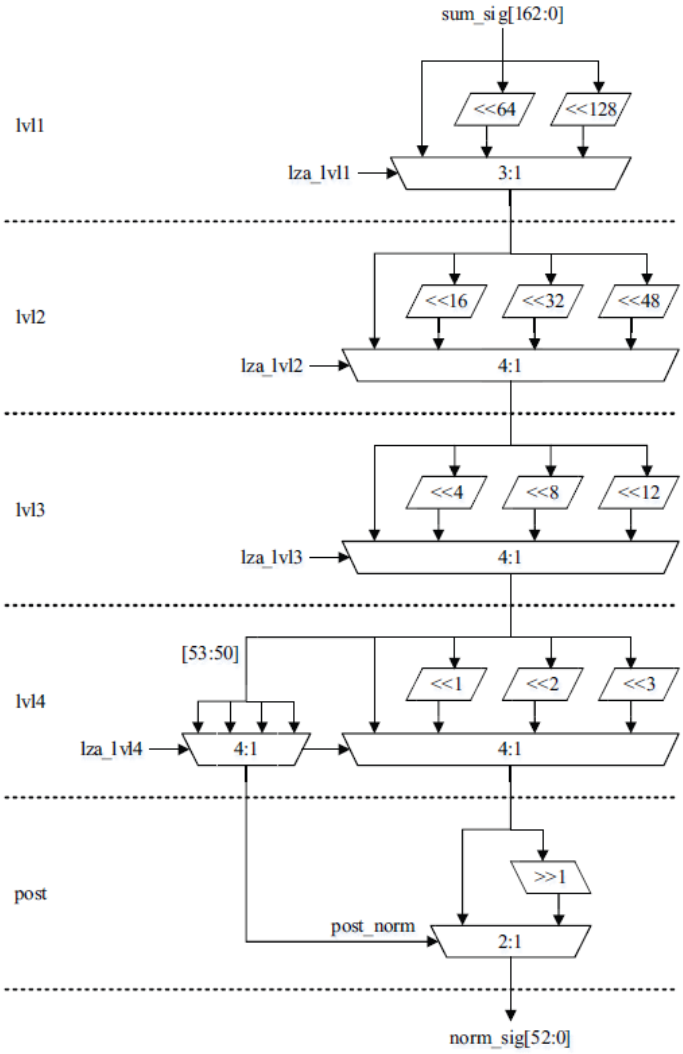


Figura 4.5: Normalization Logic

della logica di confronto;

2. **Regolazione dell'esponente**:

   - L'esponente è corretto sottraendo la quantità di shift applicata durante la normalizzazione;
   - Ulteriori aggiustamenti sono effettuati in base al rilevamento di overflow o al comportamento post-arrotondamento, come indicato dalla logica dello (LZA);

Questo processo è progettato per gestire con precisione i numeri normalizzati e denormalizzati, riducendo la latenza associata.
Invece, il segno è calcolato combinando i segni degli operandi (S_A, S_B, S_C) con il risultato delle operazioni aritmetiche:

1. $S_M = S_A \bigoplus S_B$
2. Il segno dell'addendo è preso in considerazione tramite un confronto tra e_M e e_C e l'applicazione di operazioni XOR con il segno del prodotto;
3. La logica tiene conto di eventuali inversioni della mantissa (causate dal complemento a due) e delle modalità di arrotondamento per determinare il segno finale

Il risultato finale del segno è stabilito nel terzo ciclo dell'operazione FMA, garantendo che sia accuratamente sincronizzato con il calcolo dell'esponente.

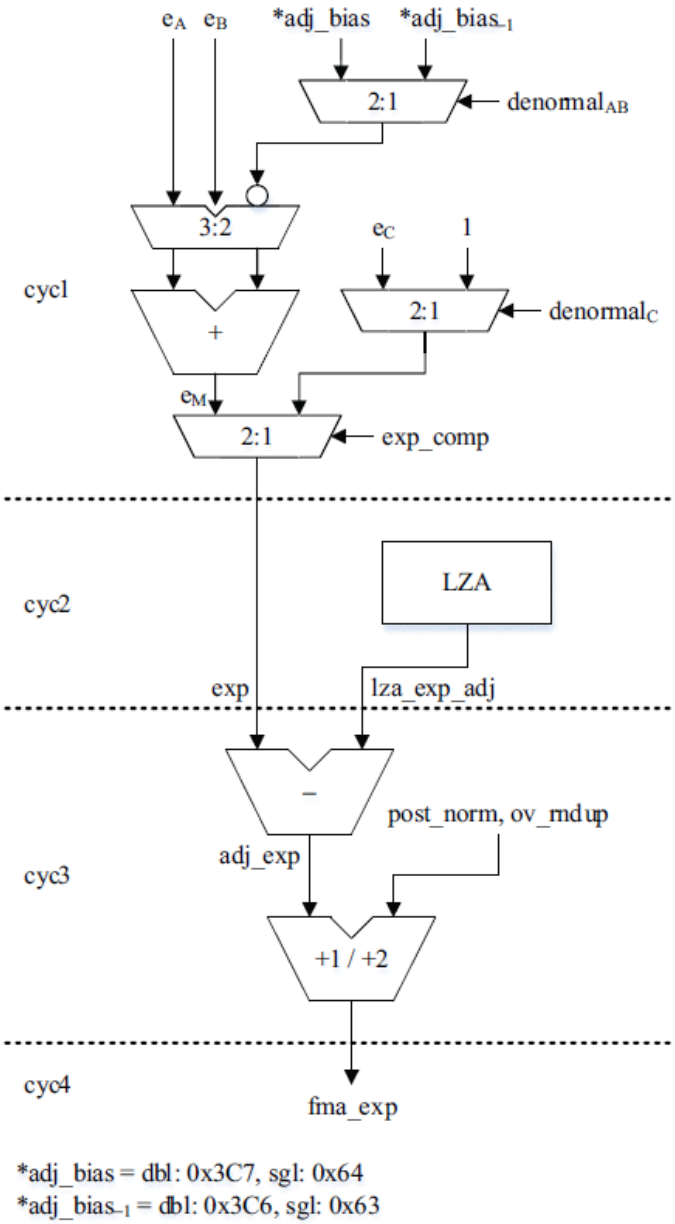


Figura 4.6: Exponent Logic

5 Conclusioni e Sviluppi Futuri

Il formato *floating point* rappresenta una soluzione versatile per la gestione dei numeri reali nei calcolatori, con un'ampia gamma di applicazioni in vari settori della scienza, dell'ingegneria e della tecnologia. Nonostante i suoi limiti, come l'errore di arrotondamento e i problemi di stabilità numerica, il *floating point* rimane un componente essenziale per molte applicazioni di calcolo avanzato. Con l'evoluzione delle architetture hardware e delle tecniche numeriche, gli algoritmi e le strutture che utilizzano il formato *floating point* continuano a migliorare, riducendo gli errori e ampliando le possibilità di elaborazione numerica.

All'interno di questo progetto è stata realizzata con successo un'architettura *Fused-Multiply Add* in formato *floating point* e sono stati analizzati tutti i blocchi computazioni che la compongono, descrivendone dettagliamente le funzioni al loro interno e le operazioni che effettuano. Successivamente, il sistema è stato simulato in modo tale da analizzare anche tutti i *corner case* del caso. Inoltre, il sistema è stato implementato su FPGA e sono state analizzate le risorse utilizzate ed i tempi ottenuti.

Infine, è stato effettuato un'analisi di un'architettura FMA con supporto completo per i *subnormals*, presente [12].

Per quanto riguarda i possibili sviluppi futuri di questo lavoro, questi sono:

- Un interessante sviluppo futuro consiste nel confronto della modalità di *rounding* utilizzata con altre configurazioni, per valutarne l'impatto su precisione e

prestazioni in specifiche applicazioni;

- Un ulteriore approfondimento consisterebbe nel confrontare l'architettura sviluppata con altre unità FMA implementate in processori concorrenti;

- Un altro sviluppo possibile riguarda l'implementazione di tecniche ancora più efficienti per la gestione dei numeri subnormal, come la riduzione dell'area dedicata al calcolo o l'integrazione di algoritmi predittivi per ottimizzare ulteriormente il flusso dei dati;

Elenco delle figure

Elenco delle tabelle

Bibliografia

[1] Milos D. Ercegovac, Tomas Lang, *"Digital Arithmetic"*, Morgan Kaufmann Publishers, 2004

[2] Tom M. Bruintjes, *"Design of Fused Multiply-Add Floating-Point and Integer Path"*, 2011

[3] *"Handbook of Floating-Point Arithmetic"*, Birkhauser

[4] Alberto Nannarelli, *"Fused Multiply-Add for Variable Precision FLoating Point"*, 2019

[5] Eric Quinnel, Earl E. Swartzlander Jr., Carl Lemonds, *"Floating-Point Fused Multiply-Add Architectures"*, 2007

[6] Sylvie Boldo, Guillaume Melquiond, *"Emulation of FMA and Correctly Rounded Sums: Proved Algorithms Using Rounding to Odd"*, 2008

[7] Bjorn Liebig, Jens Huthmann, Andreas Koch, *"Architecture Exploration of High-Performance Floating-Point Fused Multiply-Add Units and their Automatic Use in High-Level Synthesis"*,

[8] Ahmed Youssef, *"Single-Precision Floating-Point Multiply-Add-Fused for Field Programmable Gate Arrays"*, 2014

[9] Vojin G. Oklobdzija, *"An Implementation Algorithm and Design of a Novel Leading Zero Detector Circuit"*, 1992

[10] Hu He, Zheng Li, Yihe Sun, *"Multiply-Add Fused Float Point Unit with On-fly Denormalized Number Processing"*, 2005

[11] Neil H. Weste, David Money Harris, *CMOS VLSI Design: A Circuits and Systems Perspective*

[12] J. Sohn, D. K. Dean, E. Quintana, W. S. Wong, Enhanced Floating-Point Multiply-Add with Full Denormal Support