Corso di laurea in Ingegneria Informatica
Politecnico di Milano

Progetto di reti logiche

Moltiplicatore floating-point IEEE754

Mattia Brianti
Alex Hathaway

Anno Accademico 2023/2024

Indice

1 Introduzione

Il presente progetto intende sviluppare un moltiplicatore per numeri in formato floating-point conformi allo standard IEEE754 in virgola mobile a pre-

cisione singola. L'obiettivo principale è sviluppare un sistema in grado di gestire operandi normalizzati, denormalizzati e valori speciali, come NaN (Not a Number), infinito e zero, integrando al contempo un'architettura pipelined per ottimizzare l'efficienza complessiva del progetto. Il sistema dovrà eseguire come da specifica, i seguenti controlli: riconoscere che gli input siano uguali o diversi da zero, verificare l'overflow o l'underflow degli esponenti ed infine la normalizzazione e l'arrotondamento del risultato. Lo standard IEEE754 definisce la rappresentazione dei numeri in virgola mobile mediante la seguente formula:

$$(-1)^s \times 2^{e-\text{bias}} \times 1.m$$

dove s rappresenta il bit di segno, e indica l'esponente (dal quale viene sottratto il *bias*, un valore fisso pari a 127 per la precisione singola) e, infine, m rappresenta la mantissa, con un 1 implicito davanti poiché il numero è normalizzato.
Nello standard IEEE754, esistono due classi di numeri rappresentabili: i numeri normalizzati e quelli denormalizzati. Questa distinzione si basa sul range dei valori che ciascuna classe può rappresentare. I numeri normalizzati possono rappresentare i valori che vanno da:

- $1,17 \times 10^{-38}$ a $3,4 \times 10^{38}$
- $-3,4 \times 10^{38}$ a $-1,17 \times 10^{-38}$.

Mentre i numeri denormalizzati corrispondono ai numeri da:

- $1,4 \times 10^{-45}$ a $1,17 \times 10^{-38}$
- $-1,17 \times 10^{-38}$ a $-1,4 \times 10^{-45}$

La distinzione tra queste due classi può essere effettuata osservando gli 8 bit dell'esponente: nei numeri denormalizzati, infatti, l'esponente sarà costituito da tutti zeri.
Il processo di moltiplicazione tra due numeri floating-point può essere suddiviso nei seguenti passaggi fondamentali:

1. **Controllo degli operandi**: in questo passaggio si verifica se gli operandi assumono valori speciali come 0, NaN o infinito. In tali casi, viene gestito ogni caso specifico; altrimenti, se gli operandi sono normalizzati o denormalizzati, si procede con il calcolo della moltiplicazione.
2. **Calcolo del segno del risultato**: qui viene determinato il segno del numero risultante dalla moltiplicazione.

3. **Somma degli esponenti e sottrazione del bias**: in questo passaggio viene eseguita la somma degli esponenti dei due operandi, seguita dalla sottrazione del bias (127).

4. **Controllo di overflow e underflow dell'esponente**: Verifica se l'esponente rientra nel range dei numeri normalizzati, denormalizzati oppure se si é in presenza di un caso speciale come overflow o underflow.

5. **Moltiplicazione delle mantisse**: in questo caso si esegue la moltiplicazione delle mantisse dei due operandi, includendo l'1 implicito per i numeri normalizzati o lo 0 implicito per quelli denormalizzati.

6. **Normalizzazione della mantissa**: Regolazione della mantissa risultante e adeguamento dell'esponente di conseguenza.

7. **Scrittura del risultato**: Composizione finale del risultato in base ai calcoli eseguiti.

Lo standard IEEE754 definisce valori speciali per 0, Infinity e NaN:

- **0**: `00000000000000000000000000000000`
- **Infinity**: `S11111111000000000000000000000000`
- **NaN**: `S11111111-.....1.........-`

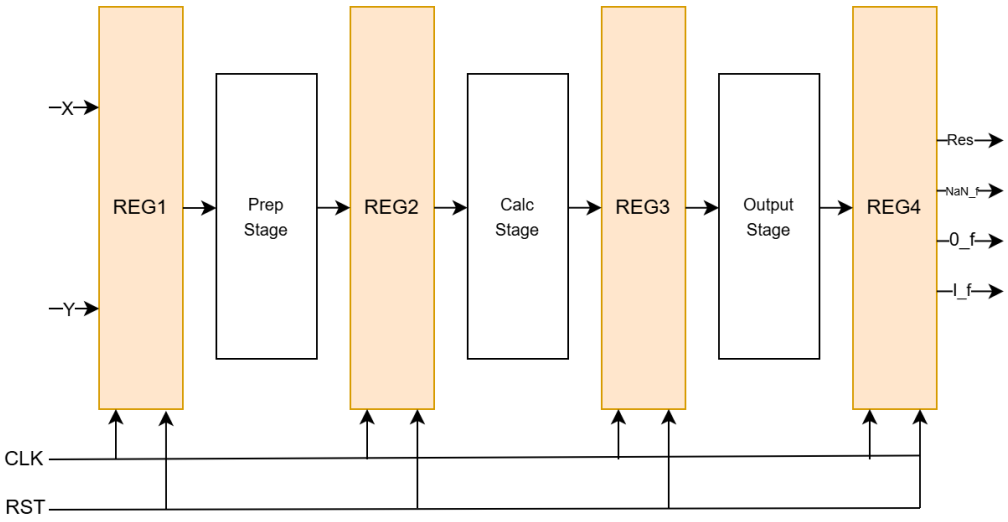
Nel contesto della codifica dei valori NaN (Not a Number), un valore è considerato tale quando l'esponente è composto interamente da bit a 1 e almeno un bit della mantissa assume valore 1. La scelta della rappresentazione di NaN adottata in questo progetto è stata effettuata in conformità con le specifiche dell'ARM Architecture Reference Manual, il quale prevede due modalità di codifica:

- **Signaling NaN (SNaN)**: caratterizzato da un bit più significativo (MSB) della mantissa impostato a 0.
- **Quiet NaN (QNaN)**: caratterizzato da un MSB della mantissa pari a 1.

Nel nostro caso, è stata adottata la seconda modalità, ovvero il QNaN.
Quando uno dei due operandi presenta valori come 0 o infinito, il moltiplicatore imposterà il risultato ai valori corrispondenti riportati sopra, 0 se si moltiplica per 0 ed infinito se si moltiplica per infinito. Per la moltiplicazione tra 0 e infinito, il risultato sará impostato a NaN.
Per la gestione delle condizioni di overflow e underflow, si è adottata la strategia di restituire i valori corrispondenti agli estremi dell'intervallo di rappresentazione:

- In caso di overflow, il risultato sarà rappresentato dal valore infinito.
- In caso di underflow, è necessaria un'ulteriore distinzione: se l'esponente risultante si trova nell'intervallo compreso tra 0 e -23, l'underflow è considerato recuperabile, poiché il valore rientra nell'insieme dei numeri denormalizzati. Se l'esponente è inferiore a -23, il valore non è più rappresentabile e il risultato verrà impostato a 0.

2 Specifica



L'architettura del sistema prevede l'ingresso di due numeri a 32 bit in codifica IEEE754 e un segnale di clock insieme ad un segnale di reset. Gli input numerici vengono posizionati in appositi registri di ingresso, mentre il risultato della moltiplicazione viene reso disponibile su un registro di uscita, insieme ad altre tre flag che ci segnalano la presenza di un risultato speciale (NaN, Infinito o Zero). Per poter rendere l'architettura pipelined, e permettere al sistema di elaborare sequenzialmente più richieste, riducendo il tempo di latenza per i risultati successivi, si é deciso di dividere il sistema nei seguenti stadi, tra i quali sono stati posti dei registri:

1. **Prep Stage**: Analizza esponente e mantissa degli input per riconoscere eventuali edge cases (Zero, NaN, Infinity), estende la mantissa a 24 bit aggiungendo l'1 implicito per i numeri normalizzati o lo 0 per i denormalizzati e calcola il segno della moltiplicazione.

2. **Calc Stage**: Esegue le operazioni aritmetiche necessarie, come la moltiplicazione delle mantisse estese e la somma degli esponenti, alla quale successivamente viene sottratto il bias.

3. **Output Stage**: Normalizza la mantissa a 23 bit e regola l'esponente in base allo shift fatto su di essa. Infine compone il risultato in uscita a 32 bit e segnala eventuali casi limite in uscita.

Nel primo modulo vengono eseguite operazioni preliminari necessarie alla moltiplicazione tra numeri in virgola mobile. Nello specifico, il modulo si occupa delle seguenti funzionalità principali:

- **Gestione dei casi particolari**: il modulo verifica la presenza di condizioni particolari, come valori infinito, NaN, zero o la condizione in cui entrambi i numeri siano denormalizzati, analizzando opportunamente esponenti e mantisse degli operandi.

- **Normalizzazione della mantissa**: il modulo determina se i numeri in ingresso sono normalizzati o denormalizzati, aggiungendo un '1' come MSB della mantissa per i numeri normalizzati o uno '0' nel caso di numeri denormalizzati.

- **Calcolo del segno della moltiplicazione**

Nel secondo modulo invece vengono svolte le operazioni aritmetiche necessarie al calcolo del risultato finale usando i seguenti componenti:

- **EXP_ADDER** : il quale somma gli esponenti tramite un CLA_8.

- **BIAS_SUBTRACTOR** : il quale sottrae alla somma fatta in EXP_ADDER il bias (127).

- **MANTIX_MULTIPLIER** : il quale moltiplica le mantisse estese andando a sommare i risultati parziali mediante un CSA_24.

Il terzo ed ultimo modulo integra le operazioni di arrotondamento, correzione dell'esponente, gestione di casi particolari e ricombinazione del risultato utilizzando quattro componenti principali:

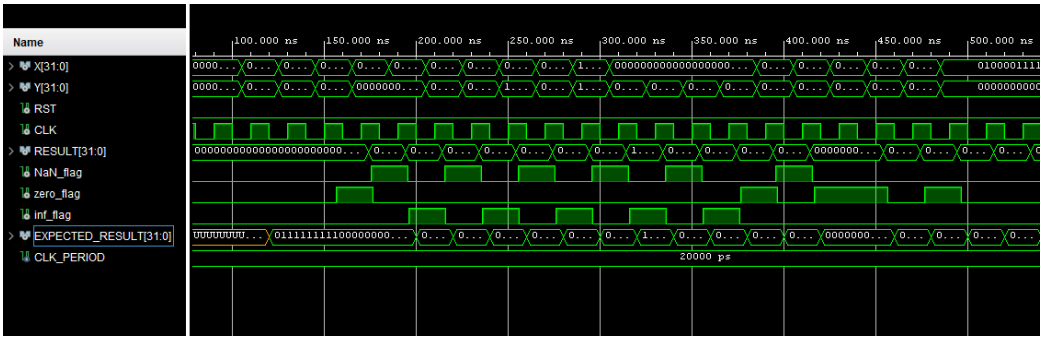
- **ROUND**: il quale normalizza la mantissa trovando il primo '1' e applicando lo slice corretto. Inoltre segnala se l'esponente in uscita deve essere modificato.

- **FINAL_EXP_CALC** : il quale calcola il valore corretto dell'esponente finale dopo l'arrotondamento.

- **RES_FIX** : Gestisce i casi di overflow e underflow, effettuando il recupero dei risultati denormalizzati quando possibile.
- **FLAG_DETECTOR** : il quale rileva NaN, zero e infinito tramite un flag di ingresso e restituisce il risultato appropriato, impostando i relativi flag e fornendo il risultato finale corretto.

2.1 Interfaccia del sistema

L'interfaccia progettata riceve in ingresso due numeri a 32 bit codificati secondo lo standard IEEE754, oltre a un segnale di clock e un segnale di reset, che permettono di implementare un'architettura pipelined. La computazione effettuata dall'interfaccia genera un risultato, anch'esso codificato in formato IEEE754, che viene reso disponibile su un registro di uscita a 32 bit, e rappresenta il prodotto dei due operandi in ingresso. Il risultato della moltiplicazione sarà disponibile al terzo ciclo di clock dall'inserimento dei due valori di ingresso. L'architettura utilizzata per implementare questo processo è una pipeline suddivisa in tre stadi, che consente di ottenere parallelismo nel calcolo. Questo significa che, mentre il primo risultato sarà disponibile dopo 3 cicli di clock dall'inserimento degli operandi, i risultati delle coppie successive saranno disponibili con una latenza di 1 ciclo di clock ciascuno. Inoltre in uscita é stato scelto di implementare tre flag diverse (NaN_flag, Zero_flag e Inf_flag) che segnalano la presenza di operandi speciali in uscita, come da prassi in alcuni microprocessori. Il disegno seguente rappresenta i segnali d'ingresso e uscita, con i rispettivi ritardi, dovuti alla simulazione post implementation:

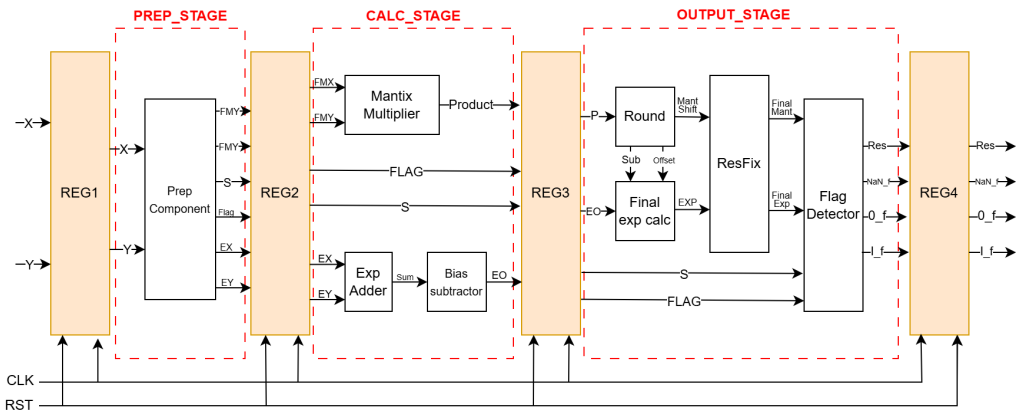


La frequenza di clock utilizzata é pari a 50 Mhz (20ns), la quale é stata scelta utilizzando i constraints di Vivado per comprendere i ritardi con i quali i vari stadi producevano i propri output. Un riassunto dei timing ottenuti é rappresentato nella seguente tabella:

Stage	Estimated Delay
PREP_STAGE	3.984ns
CALC_STAGE	12.966ns
OUTPUT_STAGE	3.737ns

Come facilmente prevedibile lo stage piú lento é quello contenente il moltiplicatore delle mantisse, che rappresenta il componente piú rilevante del progetto.

2.2 Architettura del MULTIPLIER_IEEE754



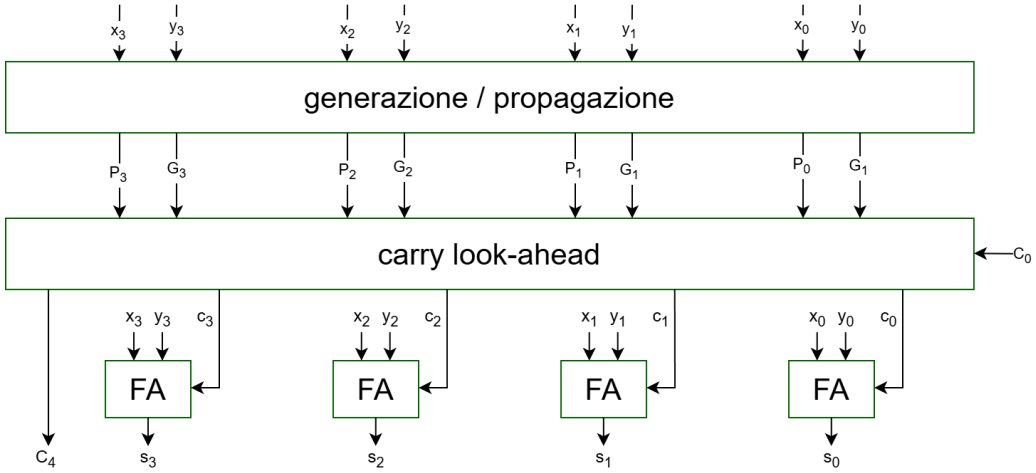
La struttura del sistema è suddivisa in stadi separati, con registri intermedi che consentono di preservare i dati durante le operazioni concorrenti. Tutti i registri sono sincronizzati con un segnale di reset e un clock condiviso. La frequenza del clock è stata scelta in base al modulo più lento del sistema, che è stato ottimizzato fino a raggiungere il valore di 20 ns. La divisione in stadi è stata effettuata tenendo conto delle dipendenze tra le diverse operazioni intermedie:

- Il **primo stadio**, si occupa delle operazioni preliminari sugli ingressi, come il controllo dei valori speciali, il calcolo del segno e la preparazione della mantissa per la moltiplicazione, aggiungendo il bit implicito (1 o 0).
- Il **secondo stadio**, dedicato al calcolo, include la moltiplicazione delle mantisse tramite un moltiplicatore, la somma degli esponenti e la sottrazione del bias.

- L'**ultimo stadio** è responsabile del controllo finale sull'esponente e del rounding della mantissa, che deve essere correttamente normalizzata e riportata a 23 bit.

2.3 CLA

Per svolgere le operazioni aritmetiche all'interno del progetto, é stato scelto di utilizzare sommatori CLA. Ne sono stati realizzati tre tipi: CLA_2, CLA_4 e CLA_8. Questi, sono stati posti in configurazione ripple carry per creare sommatori in grado di gestire piú bit come ad esempio un CLA_10 (CLA_8 + CLA_2) oppure il CLA_54 (5*CLA_10 + CLA_4).



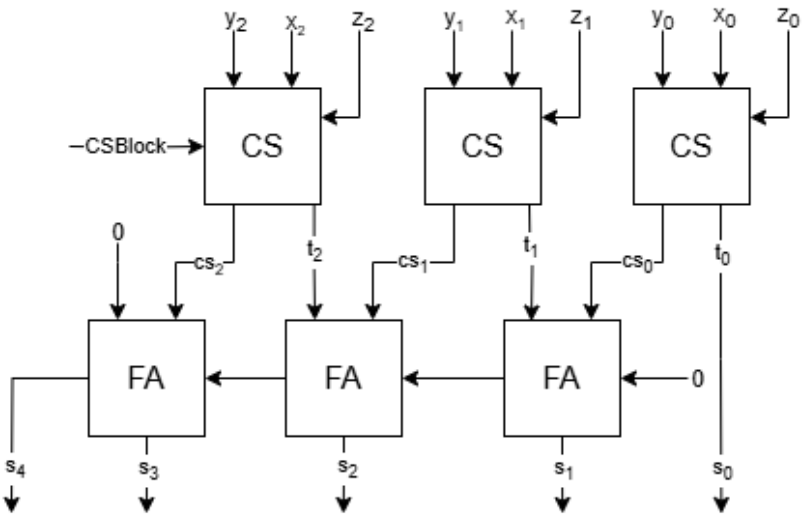
2.4 CSA_24

Per eseguire la somma dei prodotti parziali legati alla moltiplicazione delle mantisse é stato utilizzato un CSA_24 in quanto architettura particolarmente adatta alla somma di piú operandi, in questo caso 24 prodotti parziali da 47 bit ciascuno.

2.4.1 CSBlock

Il CSBlock é un componente fondamentale per la realizzazione di un CSA in quanto é un blocco generic che permette l'istanziazione di un numero di full adder pari al numero di bit degli operandi da sommare. Il singolo Full Adder (FA) somma tre bit alla volta producendo una somma parziale e un carry out, senza propagare il riporto, accelerando così il calcolo delle somme

multiple. Di seguito viene mostrato l'esempio di un Carry Save che somma tre operandi a tre bit:



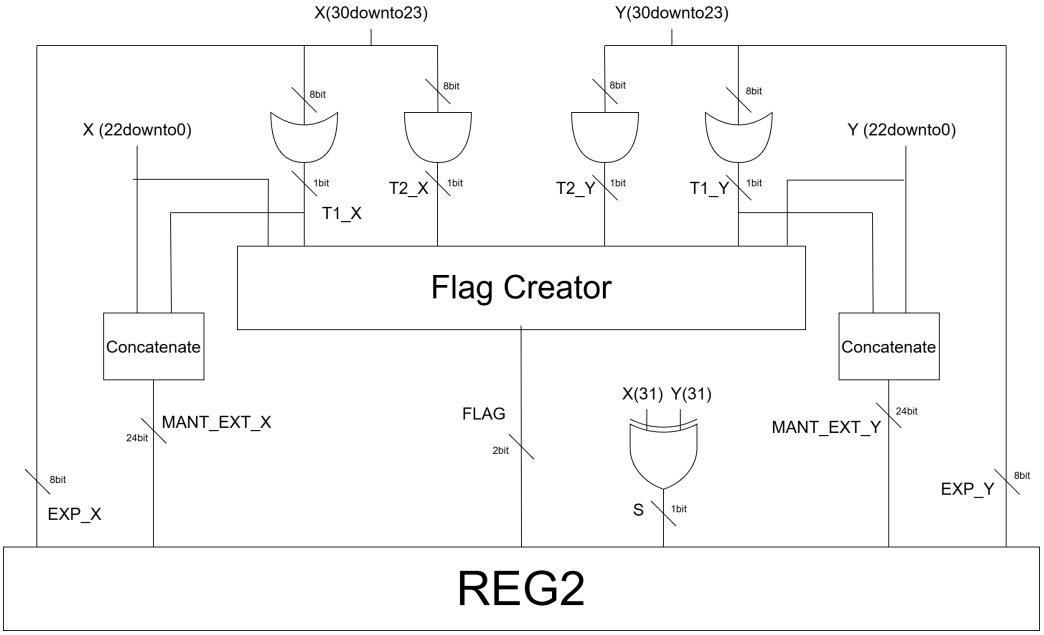
2.5 Encoder Offset

Questo componente prende in input un valore a 24 bit e ha la funzione di restituire in uscita la posizione dell'uno piú significativo presente sulle linee di ingresso. Il funzionamento é quello di un priority encoder inverso e con un offset di 1.
Esempio: se in input venisse passato 000001000000000000000000 l'offset encoder restituirá 00110 ovvero 6 (5+1).

2.6 Denormalizer

Questo componente si occupa dello shift della mantissa. Questo essendo un Barrel Shifter, riceve in ingresso un valore ed in base ad esso compie lo shift dell'input, in questo caso verso destra, andando ad esplicitare l'1.
Esempio: Se venisse passato in input il valore 10011 (19, il valore assoluto della somma tra 22 e EXP), il componente shifterá la mantissa verso destra di 3 bit esplicitando l'uno nel LSB e dando in uscita 0001 & MANTIX(22 downto 4).

2.7 PREP_STAGE



Il Prep_Component è un modulo che si occupa della preparazione dei due operandi prima della fase di calcolo, suddividendoli nelle loro componenti fondamentali e gestendo i casi speciali. Il modulo prende in ingresso due numeri in codifica IEEE 754 e restituisce in uscita il segno della moltiplicazione, gli esponenti e le mantisse correttamente estese. Come segno dell'operando viene usato il 32esimo bit, mentre i bit dell'esponente vanno dal 31esimo al 24esimo e quelli della mantissa dal 23esimo al primo. Le operazioni svolte sono 3:

- **Calcolo del segno risultante dalla moltiplicazione**: questo viene fatto utilizzando una porta XOR che prende in ingresso i MSB degli operandi in ingresso.
- **Estensione della mantissa**: questo avviene aggiungendo un bit implicito alla mantissa originale, necessario per rappresentare correttamente sia i numeri normalizzati che quelli denormalizzati. Per determinare se un numero è normalizzato o denormalizzato, il modulo verifica il valore dell'esponente. Se almeno un bit dell'esponente è 1, allora il numero è normalizzato e il bit implicito sarà 1. Se invece l'esponente è tutto 0, il numero è denormalizzato e il bit implicito sarà 0. Questa operazione viene fatta tramite una semplice OR dei bit dell'esponente.

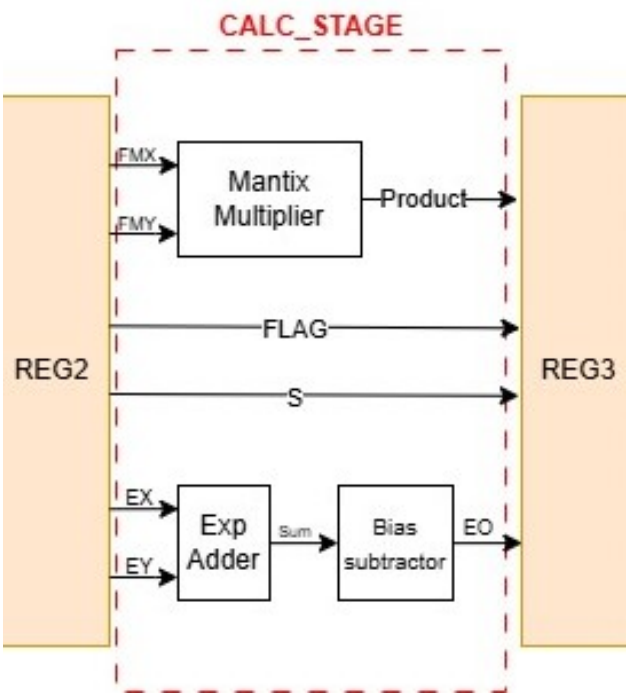
Dopo questa verifica, la mantissa viene estesa concatenando il bit implicito ottenuto con i 23 bit della mantissa originale. Il risultato è una mantissa a 24 bit, dove il primo bit indica se il numero è normalizzato (1) o denormalizzato (0), seguito dai 23 bit originali della mantissa.

- **Innalzamento della flag per i casi speciali**: Il valore di FLAG viene determinato analizzando i bit dell'esponente e della mantissa dei due operandi in ingresso. Il modulo utilizza due segnali intermedi per ogni numero:
  - T1_X e T1_Y che se a 0 indicano l'esponente è formato da tutti zeri.
  - T2_X e T2_Y che se a 1 indicano l'esponente è formato solo da 1.

  A seconda dei valori di T1 e T2, il modulo innalza la FLAG a uno dei seguenti stati:
  - **"10"**: se il risultato della moltiplicazione é NaN, ció avviene quando almeno uno dei due numeri ha esponente uguale a 255 (T2 = 1) e la mantissa diversa da 0, il che rappresenta la codifica di NaN (Not a Number). Inoltre se uno dei numeri è Infinito (T2 = 1 e mantissa = 0) e l'altro è zero (T1 = 0 e mantissa = 0), l'operazione è indeterminata e segnalata in uscita con la codifica di NaN.
  - **"01"**: se il risultato della moltiplicazione é infinito, ció avviene se uno dei due numeri é infinito (T2 = 1 e mantissa = 0) e l'altro numero é diverso da zero o da NaN.
  - **"11"**: se il risultato della moltiplicazione é zero, ció avviene quando uno dei numeri é zero (T1 = 0 e mantissa = 0) e l'altro numero é diverso da infinito o NaN. Se entrambi i numeri sono denormalizzati (ovvero entrambi hanno esponente = 0), il risultato della moltiplicazione è trattato come zero.

  Se gli operandi non presentano nessuna delle casistiche sopra elencate, ció significa che i numeri sono normali e possono essere moltiplicati senza particolari eccezioni.

2.8 CALC_STAGE



2.8.1 Exponent adder

Questo componente riceve in ingresso gli esponenti da 8bit degli operandi e ne restituisce in uscita la somma (Sum) su 9bit, poichè i due numeri sommati potrebbero eccedere il valore massimo rappresentabile da 8 bit. Nel caso in cui in ingresso ci fosse un numero denormalizzato, il componente imposta l'esponente di quel numero a "00000001", ovvero -126 nella codifica dello standard. La somma viene effettuata utilizzando un Carry Look Ahead da 8 bit con C_{in} fissato a 0 e con il riporto d'uscita che diventa il MSB della somma degli esponenti. Il numero SUM viene trattato come numero senza segno perché vengono sommati esponenti che includono già un bias positivo. La somma (E1 + E2 + bias + bias) è sempre positiva, quindi non servono bit di segno. Il nono bit gestisce il riporto nel Carry Look Ahead da 8 bit.

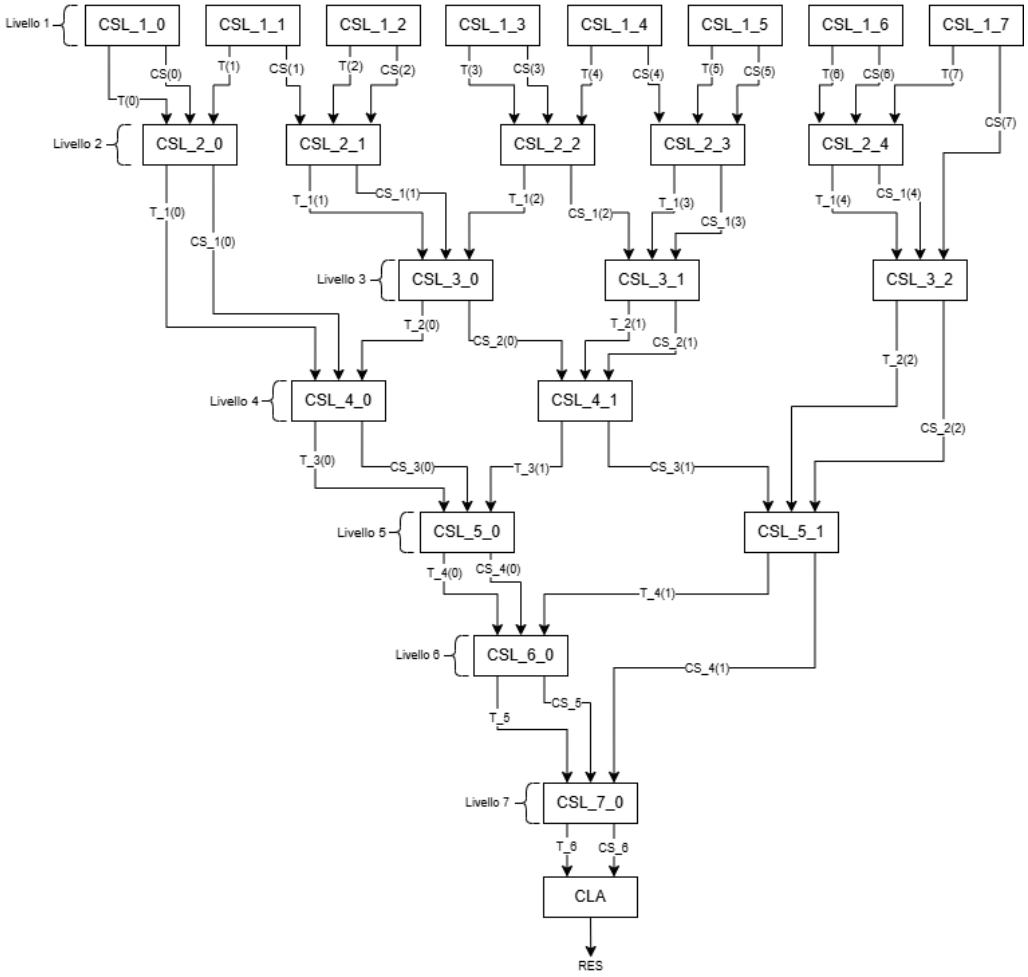
2.8.2 Bias subtractor

Questo componente prende in ingresso la somma degli esponenti rappresentata da 9bit, considerata come un numero unsigned e restituisce

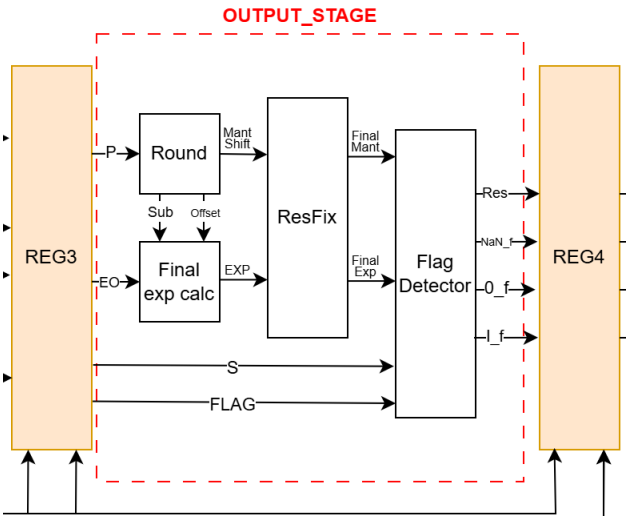
in uscita un numero in complemento a 2 da 10bit signed, in quanto sarà necessario il segno per effettuare un controllo in caso di numeri denormalizzati o per un eventuale underflow. Il componente, in particolare, ha l'obiettivo di rimuovere il bias in eccesso, essendo partiti da due esponenti aventi entrambi un bias iniziale. Questa sottrazione verrá effettuata con un Carry look ahead a 10bit, composto come annunciato prima da un CLA_2 e un CLA_8, dove in ingresso si avrá la somma degli esponenti (con uno '0' come MSB poichè unsigned) e "1110000001" (-127 in complemento a 2)..

2.8.3 Mantix multiplier

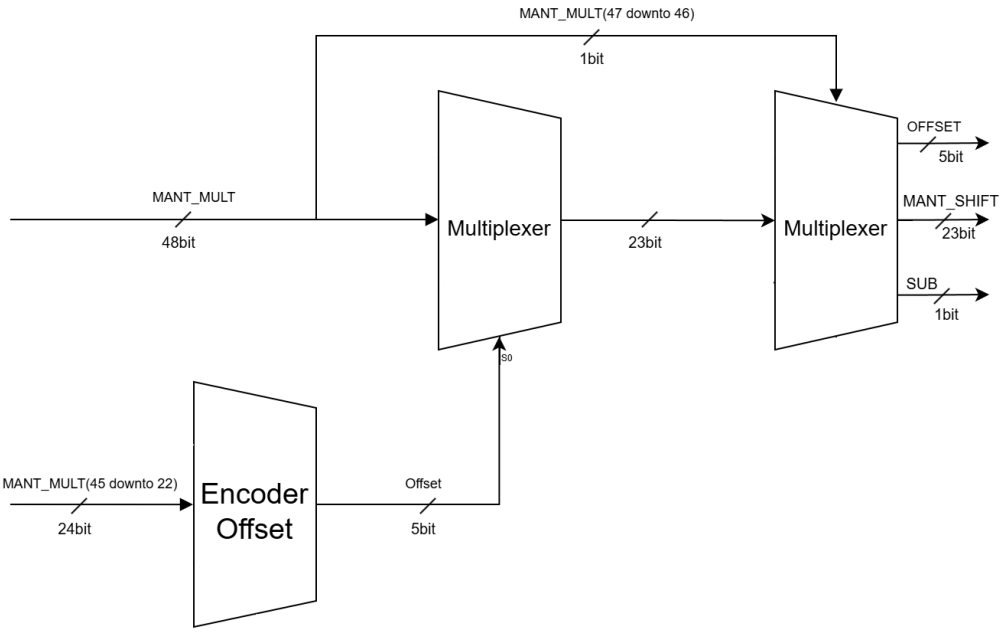
Il componente esegue la moltiplicazione delle mantisse dei due operandi iterando sui bit del secondo termine. Per ogni bit a '1', si genera un prodotto parziale che corrisponde al primo operando, mentre per un bit a '0', il prodotto parziale sará pari ad una stringa di 24 zeri. Per facilitare la somma dei prodotti parziali, questi vengono estesi a 47 bit mediante l'aggiunta di zeri sia a sinistra che a destra del risultato ottenuto in precedenza. Per la precisione, ogni prodotto parziale viene shiftato a sinistra in base alla posizione del bit corrispondente nel secondo operando e successivamente vengono concatenati alla sinistra del risultato un numero di zeri tali da raggiungere la lunghezza di 47 bit. Questa estensione assicura che i prodotti parziali siano della stessa dimensione e allineati correttamente per la somma, che viene eseguita utilizzando un CSA a 24 operandi il quale sfrutta un albero di carry save suddiviso su 8 livelli (contando il CLA finale), come descritto dal seguente schema:


Livello 1
CSL_1_0
CSL_1_1
CSL_1_2
CSL_1_3
CSL_1_4
CSL_1_5
CSL_1_6
CSL_1_7
CS(0)
T(1)
CS(1)
T(2)
CS(2)
T(3)
CS(3)
T(4)
CS(4)
T(5)
CS(5)
T(6)
CS(6)
T(7)
T(0)
CS(7)
Livello 2
CSL_2_0
CSL_2_1
CSL_2_2
CSL_2_3
CSL_2_4
T_1(0)
CS_1(0)
T_1(1)
CS_1(1)
T_1(2)
CS_1(2)
T_1(3)
CS_1(3)
T_1(4)
CS_1(4)
Livello 3
CSL_3_0
CSL_3_1
CSL_3_2
T_2(0)
CS_2(0)
T_2(1)
CS_2(1)
T_2(2)
CS_2(2)
Livello 4
CSL_4_0
CSL_4_1
T_3(0)
CS_3(0)
T_3(1)
CS_3(1)
Livello 5
CSL_5_0
CSL_5_1
T_4(0)
CS_4(0)
T_4(1)
Livello 6
CSL_6_0
CS_4(1)
CS_5
T_5
Livello 7
CSL_7_0
T_6
CS_6
CLA
RES

2.9 OUTPUT_STAGE



2.9.1 Round



L'intero componente si occupa di normalizzare la mantissa risultante dalla moltiplicazione svolta nello stadio precedente, determinando un

valore di OFFSET attraverso il priority encoder sopracitato, che permette il corretto slicing della mantissa. Inoltre il componente permette l'eventuale innalzamento di un segnale di sottrazione (SUB) da usare per il calcolo dell'esponente corretto. Le operazioni svolte dal Round sono le seguenti:

- Se viene trovato un uno nel MSB, verrá posto l'OFFSET a 1 e la mantissa SHIFT verrá settata ai bit che vanno dal 47esimo al 24esimo bit della mantissa d'ingresso. Inoltre verrá posto SUB a zero in quanto si dovrá sommare uno al valore dell'esponente uscente dal CALC_STAGE.
  Se invece il MSB é a zero e il bit successivo é pari ad uno, il valore di OFFSET viene impostato a 0. In questo caso, la mantissa SHIFT assume il valore dei bit che vanno dal 46esimo al 23esimo bit della mantissa di ingresso, e SUB viene posto a 0.

Tutto ció é necessario in quanto i due bit piú significativi della moltiplicazione sono valori appartenenti alla parte intera del risultato, per cui vanno aggiunti all'esponente.

- Se nel caso precedente non viene riscontrato alcun 1, OFFSET viene determinato attraverso l'utilizzo di un priority encoder che esaminando i bit compresi tra il 45esimo e il 22esimo bit della mantissa in ingresso, mi restituisce la posizione del primo uno partendo da sinistra. La flag SUB viene impostata a 1 e la mantissa verrá poi settata in base al valore di OFFSET. Se non viene trovato alcun 1 nei bit compresi tra il 45esimo e il 22esimo, allora SHIFT assume il valore degli ultimi 22 bit della mantissa ai quali viene concatenato uno zero (questo per distinguere il caso in cui OFFSET sia 23), SUB viene impostato a 1 e OFFSET al suo valore massimo (11000).

2.9.2 Final_exp_calc

Questo componente prende in ingresso l'esponente con segno, in uscita da BIAS_SUBTRACTOR, ed OFFSET e SUB in uscita da ROUND e restituisce in output il valore dell'esponente corretto, ovvero l'esponente con sommato o sottratto (in base alla flag sub) il valore di offset. Inizialmente il modulo estende l'OFFSET ricevuto in input a 10 bit e nel caso in cui SUB sia 1 l'OFFSET verrá complementato a 1 attraverso uno xor che lascerá l'OFFSET al suo valore originale nel caso in cui

SUB sia 0. Infine mediante un CLA_10 il componente esegue la somma tra l'esponente in ingresso e il risultato dello xor.

2.9.3 Res_fix

Il componente RES_FIX riceve in ingresso la mantissa shiftata da ROUND da 23bit e l'esponente con segno in uscita da FINAL_EXP_CALC da 10 bit. Il modulo si occupa di analizzare l'esponente con segno, applicando eventuali correzioni sia alla mantissa sia all'esponente in base alle seguenti condizioni:

- **Esponente superiore a 254 (Overflow)**: L'esponente in uscita viene impostato al massimo valore possibile (tutti 1), mentre la mantissa viene azzerata, ottendendo così la rappresentazione di infinito.
- **Esponente compreso tra -22 e 0 (Denormalizzato recuperabile)**: Possiamo ottenere un denormalizzato recuperabile, quindi la mantissa viene shifata di un numero di posizioni pari a EXP + 22.
- **Esponente inferiore a -22 (Underflow non recuperabile)**: Entrambi i valori, esponente e mantissa, vengono azzerati, ottendendo così la codifica di 0
- **Esponente in un intervallo valido ($0 \leq$ EXP ≤ 254)**: Il valore dell'esponente è già corretto, quindi in uscita vengono mantenuti la mantissa ricevuta in ingresso e gli ultimi 8 bit dell'esponente senza ulteriori modifiche

Per calcolare EXP+22, viene impiegato un CLA a 10 bit, mentre lo shift della mantissa viene effettuato dal modulo DENORMALIZER. Quest'ultimo riceve in ingresso il risultato della somma e shifta la mantissa, utilizzando un Barrel Shifter, di un numero di posizioni corrispondente a 22 - il risultato del CLA_10, rendendo esplicito l'1 implicito

2.9.4 Flag detector

Il modulo FLAG_DETECTOR ha il compito di determinare e assegnare il valore corretto all'output finale in base ai segnali di stato ricevuti in ingresso. Questo componente controlla specifiche condizioni, come la presenza di valori zero, infinito o NaN (Not a Number), e alza i corrispondenti segnali in uscita. Quando viene rilevata una condizione

particolare tra quelle previste, il modulo non solo imposta il valore d'uscita di conseguenza, ma attiva anche una flag specifica per indicare lo stato dell'operazione:

- **Nan_flag** viene attivata ('1') quando il valore in ingresso corrisponde a un numero non valido (NaN).
- **zero_flag** viene attivata ('1') se il risultato è zero.
- **inf_flag** viene attivata ('1') in caso di valore infinito.

Per quanto riguarda la gestione dei NaN, è importante notare che esistono diverse possibili rappresentazioni, in quanto un numero viene considerato NaN se l'esponente è composto interamente da '1' e almeno un bit della mantissa è pari a '1'. Tuttavia in questo progetto si è scelto di rappresentare i NaN secondo la codifica del Quiet NaN (QNaN), caratterizzata dall'esponente impostato su tutti '1' con il MSB della mantissa pari a '1', mentre tutti gli altri bit della mantissa sono impostati a '0'. Se nessuna di queste condizioni è verificata, il modulo considera valido il valore in ingresso e lo trasferisce direttamente all'uscita, garantendo una gestione affidabile delle situazioni eccezionali e una segnalazione adeguata delle anomalie.

3 Verifica

3.1 Test-bench

Per la verifica é stato scelto di testare il componente finale stimolando gli ingressi con numeri a 32 bit e controllando che le uscite fossero uguali ai risultati aspettati. Per aiutarci abbiamo posto un segnale ausiliario contenente l'expected result, ritardandolo di tre cicli di clock rispetto all'inserimento dei primi operandi, in modo che fosse il piú in linea possibile con l'effettivo risultato da comparare.

3.2 Casi d'uso

Nel corso delle nostre attività di testing, abbiamo esaminato meticolosamente ogni scenario e caso limite. I test effettuati hanno dimostrato che il nostro moltiplicatore è in grado di gestire con successo tutte le situazioni previste.

Come punto di partenza per i test, siamo partiti dagli Edges Cases noti:

3.2.1 Risultato atteso: NaN

Il risultato atteso è (S 11111111 10000000000000000000)

- $Nan \times Nan$
  $(01111111101100000000000010000000) \times (01111111101100000011000010000000)$
- $Nan \times Norm$
  $(01111111101100000000000010000000) \times (01111010101100000011000010000000)$
- $Nan \times Denorm$
  $(01111111101100000000000010000000) \times (00000000001100000011000010000000)$
- $Nan \times 0$
  $(01111111101100000000000010000000) \times (00000000000000000000000000000000)$
- $Nan \times Inf$
  $(01111111101100000000000010000000) \times (01111111100000000000000000000000)$
- $0 \times Inf$
  $(00000000000000000000000000000000) \times (01111111100000000000000000000000)$

3.2.2 Risultato "infinito con segno"

- $+Inf \times +Inf$
  $(01111111100000000000000000000000) \times (01111111100000000000000000000000)$
- $-Inf \times -Inf$
  $(11111111100000000000000000000000) \times (11111111100000000000000000000000)$
- $+Inf \times Denorm$
  $(01111111100000000000000000000000) \times (00000000001100000011000010000000)$
- $+Inf \times Norm$
  $(01111111100000000000000000000000) \times (01111010101100000011000010000000)$
- $+Inf \times -Inf = -Inf$
  $(01111111100000000000000000000000) \times (11111111100000000000000000000000)$

3.2.3 Risultato zero

- 0×0
  $(00000000000000000000000000000000) \times (00000000000000000000000000000000)$

- $0 \times Norm$
  $(00000000000000000000000000000000) \times (01111010101100000011000010000000)$
- $0 \times Denorm$
  $(00000000000000000000000000000000) \times (00000000001100000011000010000000)$

Dopo aver verificato i principali edge cases, ci siamo focalizzati sulle altre casistiche possibili per verificare al meglio il corretto funzionamento del nostro moltiplicatore andando a stimolarne tutti i moduli:

3.2.4 $Norm \times Norm = Norm$

$X = 01000000100100000000000000000000 = 4.5$ in decimale
$Y = 01000000011100000000000000000000 = 3.75$ in decimale

$EXP = 01000001100001110000000000000000 = 16.8750$ in decimale

3.2.5 $Denorm \times Denorm = Underflow\ (0)$

$X = 00000000011111111111111111111111 = 1.1754942e-38$ in decimale
$Y = 00000000011111111111111111111111 = 1.1754942e-38$ in decimale

$EXP = 00000000000000000000000000000000 = 0$ in decimale

3.2.6 $Norm \times Norm = Overflow\ (infinito)$

$X = 01111111011111111111111111111111 = 3.4028235e38$ in decimale
$Y = 01111111011111111111111111111111 = 3.4028235e38$ in decimale

$EXP = 01111111100000000000000000000000 = Inf$ in decimale

3.2.7 $Norm \times Denorm = Denorm$

$X = 01000000100000000000000110000100 = 4,00018500E+00$ in decimale
$Y = 00000000000010000010000110000100 = 7,46707E-40$ in decimale

$EXP = 00000000001000001000011001110010 = 2,986966E-39$ in decimale

3.2.8 *Norm* $\times$ *Denorm* = *Underflow (0)*

$X = 00000000100000100100011111111111 = 1.1964442e - 38$ in decimale
$Y = 00000000000000100100011111111111 = 2.09498e - 40$ in decimale

$EXP = 00000000000000000000000000000000 = 0$ in decimale

3.2.9 *Norm* $\times$ *Denorm* = *Norm*

$X = 01000011111000010000000000000000 = 4.500E + 02$ in decimale
$Y = 00000000000001011000110110101100 = 5.1000E - 40$ in decimale

$EXP = 00000010100111000011000010000101 = 2.295000E - 37$ in decimale