# PROVA FINALE
# DI RETI LOGICHE
## (BOOTH MULTIPLIER)

**LUCA PUGNETTI**

**Matricola: 962859 - Codice Persona: 10712818**

# INDICE

# INTRODUZIONE

## 1. *Scopo del Progetto*

Lo scopo del progetto è quello di realizzare un moltiplicatore intero - puramente combinatorio - in grado di calcolare il prodotto di due operandi di 16 con segno mediante la codifica di Booth Radix-2. Il moltiplicatore produce in uscita un risultato su 32 bit, sempre in complemento a due.

Siano MULTIPLICAND il moltiplicando fornito, MULTIPLIER il moltiplicatore fornito, PRODUCT il prodotto calcolato. Le variabili di supporto utilizzate sono MPLC (il moltiplicando di supporto), MPLR (il moltiplicatore di supporto), PARTIAL_P (il prodotto parziale calcolato ad ogni iterazione).

## 2. *Rappresentazione informale*

La prima rappresentazione dell'algoritmo è fornita per punti, in linguaggio informale:

1. Inizializzazione del PARTIAL_PRODUCT a 0;

2. Si aggiunge un bit '0' in coda al LSB del MULTIPLIER;

3. Analizzando i due bit meno significativi del MULTIPLIER si procede come segue:

   - Se 00 o 11, si procede al passo successivo;
   - Se 01, si aggiunge il MULTIPLICAND al PARTIAL_PRODUCT;
   - se 10, si sottrae il MULTIPLICAND dal PARTIAL_PRODUCT;

4. Si moltiplica il MULTIPLICAND per 2;

5. Si ritorna al punto 3 con l'analisi dei due bit successivi del MULTIPLIER, a meno che i bit del MULTIPLIER da analizzare siano esauriti;

6. Quando i bit del MULTIPLIER sono stati tutti analizzati, il PARTIAL_PRODUCT sarà di fatto il prodotto finale, PRODUCT.

## 3. *Rappresentazione in pseudo-codice*

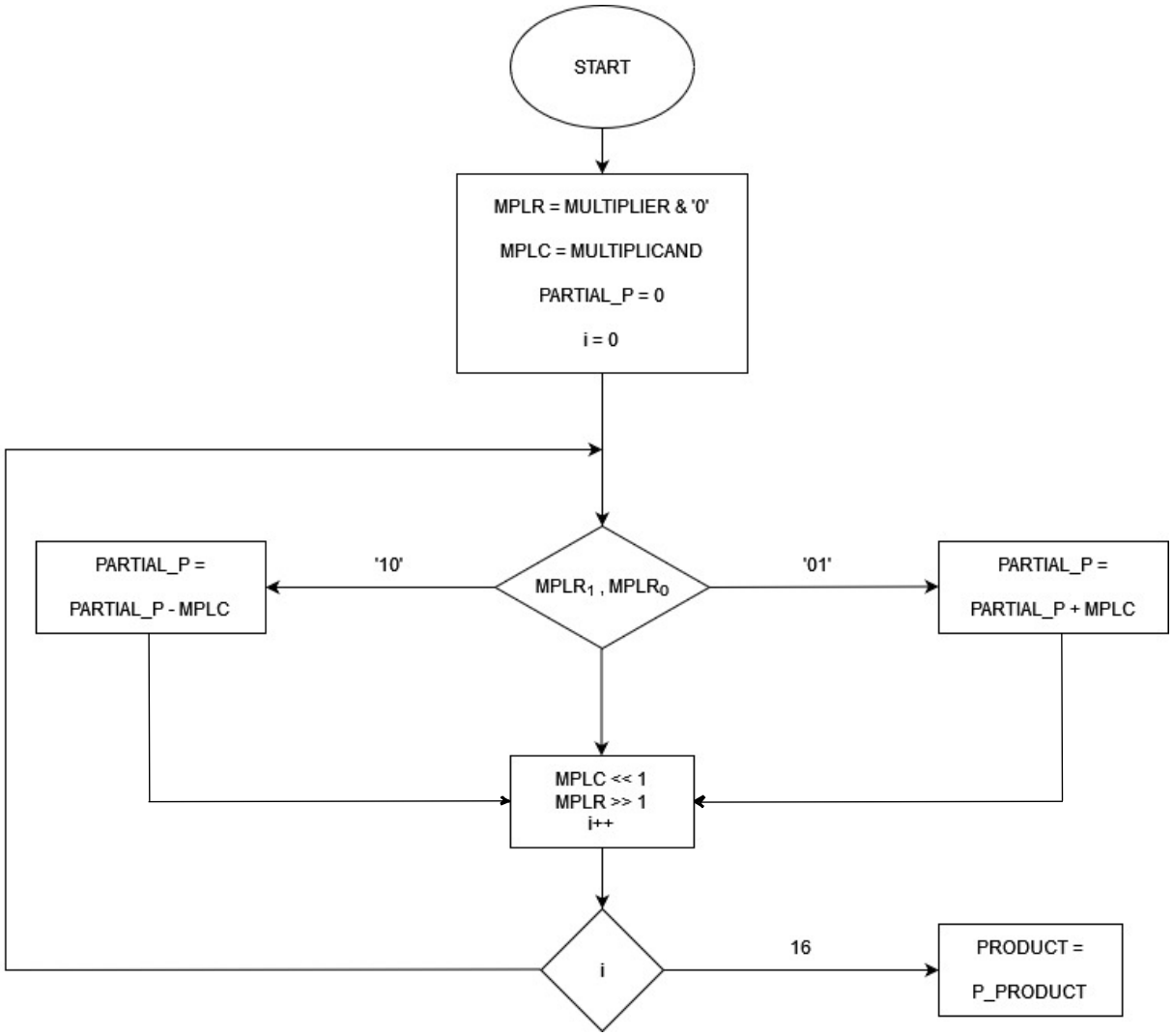
La seconda rappresentazione dell'algoritmo è fornita in pseudo-codice. In questo contesto l'analisi dei 2 LSB del MPLR (la variabile di supporto a cui viene assegnato il valore del MULTIPLIER) è più rigorosa, poiché al termine dell'analisi di ogni coppia di bit, è esplicitato uno shift logico del MPLR verso destra di 1 bit. Ciò conduce a una maggiore precisione nella formalizzazione dell'algoritmo e all'eliminazione di ambiguità dovute alla mera rappresentazione informale.

La moltiplicazione del MPLC (la variabile di supporto a cui viene assegnato il valore del MULTIPLICAND) per 2 è già mostrata mediante shift logico verso sinistra di 1 bit.

```
MPLC = MULTIPLICAND;

MPLR = MULTIPLIER & '0';

PARTIAL_P = 0;

for(i = 0; i < 16; i++ ) {

     if ( (MPLR1, MPLR0) = "10")

          PARTIAL_P = PARTIAL_P – MPLC;

     else if ( (MPLR1, MPLR0) = "01")

          PARTIAL_P = PARTIAL_P + MPLC;

     MPLC << 1;

     MPLR >> 1;

}

PRODUCT = PARTIAL_P;
```

## 4. *Rappresentazione mediante schema a blocchi*

Nella seguente immagine è fornita una rappresentazione dell'algoritmo mediante schema a blocchi, in cui la struttura dell'algoritmo, il nome e l'utilizzo delle variabili rimangono tutti invariati.




## 5. *Esempio di applicazione dell'algoritmo*

A titolo di esempio, per favorire la comprensione del funzionamento dell'algoritmo è di seguito calcolato il prodotto tra due operandi, ognuno di 4 bit. Siccome il prodotto sarà su 8 bit e le somme parziali interessano di fatto soltanto PARTIAL_P e MPLC, quest'ultimi vengono estesi su 8 bit in partenza.

MULTIPLIER = 0011 (3) | MULTIPLICAND = 0111 (7)

1.
- MPLR = 00110 | MPLC = 00000111 | PARTIAL_P = 00000000
- Guardo 2 LSB del MPLR: 00110 (differenza)
- PARTIAL_P = PARTIAL_P – MPLC = 00000000 + 11111001 = 11111001
- MPLC << 1 → MPLC = 00001110 | MPLR >> 1 → MPLR = 00011

2.
- Guardo 2 LSB del MPLR: 00011(procedere)
- MPLC << 1 → MPLC = 00011100 | MPLR >> 1 → MPLR = 00001

3.
- Guardo 2 LSB del MPLR: 00001 (somma)
- PARTIAL_P = PARTIAL_P + MPLC = 11111001 + 00011100 = 00010101
- MPLC << 1 → MPLC = 00111000 | MPLR >> 1 → MPLR = 00000

4.
- Guardo 2 LSB del MPLR: 00000 (procedere)
- MPLC << 1 → MPLC = 01110000 | MPLR >> 1 → MPLR = 00000
- Sono terminati i bit del MPLR da analizzare → PRODUCT = PARTIAL_P

PRODUCT = 00010101 (21)

Dopo il preambolo introduttivo, si procede all'analisi formale dell'architettura in questione.

# ANALISI ARCHITETTURALE

1. *Interfaccia del componente top level*

*BOOTH_MULTIPLIER*


Il *Booth_Multiplier* presenta 2 segnali in ingresso e 1 segnali in uscita.

Analisi degli ingressi:

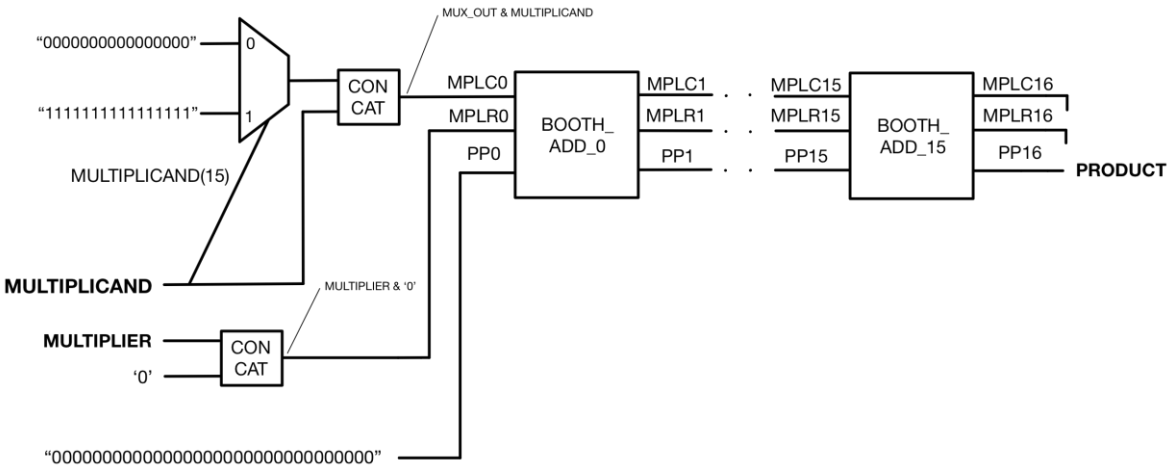
- **MULTIPLICAND** rappresenta il moltiplicando a 16 bit;
- **MULTIPLIER** rappresenta il moltiplicatore a 16 bit;

Analisi dell'uscita:

- **PRODUCT** rappresenta il prodotto a 32 bit della moltiplicazione;

**ARCHITETTURA**:

Nel *Booth_Multiplier* viene istanziato un *Multiplexer* (a 2 ingressi) per la corretta estensione del segnale MULTIPLICAND su 32 bit mantenendo la codifica in complemento 2. Inoltre vengono fatte 16 istanze del componente



*Booth_Add* e vengono inizializzati i segnali necessari al corretto funzionamento dell'architettura.

**LEGENDA**

- MPLRn corrisponde all'n-esimo ingresso dell' n-esima istanza di *Booth_Add* e MPLRn+1 corrisponde alla relativa uscita di tale istanza (contenente il valore parziale di MULTIPLIER);

- MPLCn corrisponde all'n-esimo ingresso dell' n-esima istanza di *Booth_Add* e MPLCn+1 corrisponde alla relativa uscita di tale istanza (contenente il valore parziale di MULTIPLICAND);

- MPLCn corrisponde all'n-esimo ingresso dell' n-esima istanza di *Booth_Add* e MPLCn+1 corrisponde alla relativa uscita di tale istanza (contenente il valore parziale di PRODUCT);

**LOGICA:**

Gli ingressi del componente *Booth Multiplier* sono **MULTIPLICAND** e **MULTIPLIER**. Quello di seguito riportato è il funzionamento del componente *Booth Multiplier* che si occuperà del progressivo calcolo del **PRODUCT**.

- Come prima cosa si concatena un bit '0' in coda al LSB del **MULTIPLIER**, ecco perché è stato introdotto il modulo *Concat*;

- Siccome il risultato atteso **PRODUCT** è frutto di somme parziali tra **MULTIPLICAND** e PARTIAL_PRODUCT c'è utilità ad estendere il segnale MULTIPLICAND, originariamente di 16 bit, su 32 bit. Questo comporta che si debbano aggiungere 16 bit '1' oppure 16 bit '0' in testa al **MULTIPLICAND** in base al valore del bit più significativo del **MULTIPLICAND.** Questo è il motivo per cui viene utilizzato un multiplexer a 2 ingressi (*MUX_2TO1*), il bit

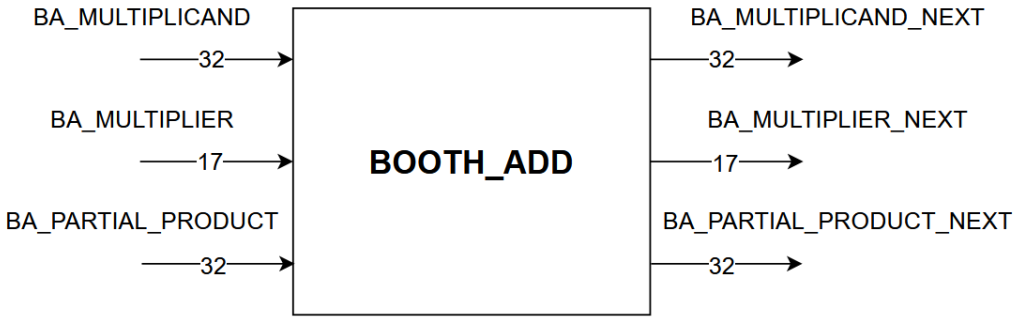
di selezione è perciò il MSB dell'ingresso **MULTIPLICAND.** L'uscita del *MUX_2TO1* viene concatenata (tramite il modulo *Concat*) al **MULTIPLICAND**;

- Dopo l'inizializzazione del PARTIAL_PRODUCT a 0 su 32 bit è possibile iniziare a effettuare le somme parziali che porteranno al progressivo calcolo del **PRODUCT**. Ecco perché vengono istanziati 16 *Booth_Add* che si occuperanno di ogni somma parziale e della preparazione degli operandi per ogni somma parziale successiva.

L'uscita relativa al prodotto parziale calcolata dall'ultima istanza del componente *Booth Add* (BOOTH_ADD_15) è il prodotto di mio interesse, **PRODUCT**.

## 2. *Interfaccia dei sottocomponenti*

### *BOOTH_ADD*




Il *Booth_Add* è il componente che si occupa di ogni somma parziale. Il componente, inoltre, si occupa della preparazione del moltiplicando e del Moltiplicatore per la somma successiva.

Analisi degli ingressi:

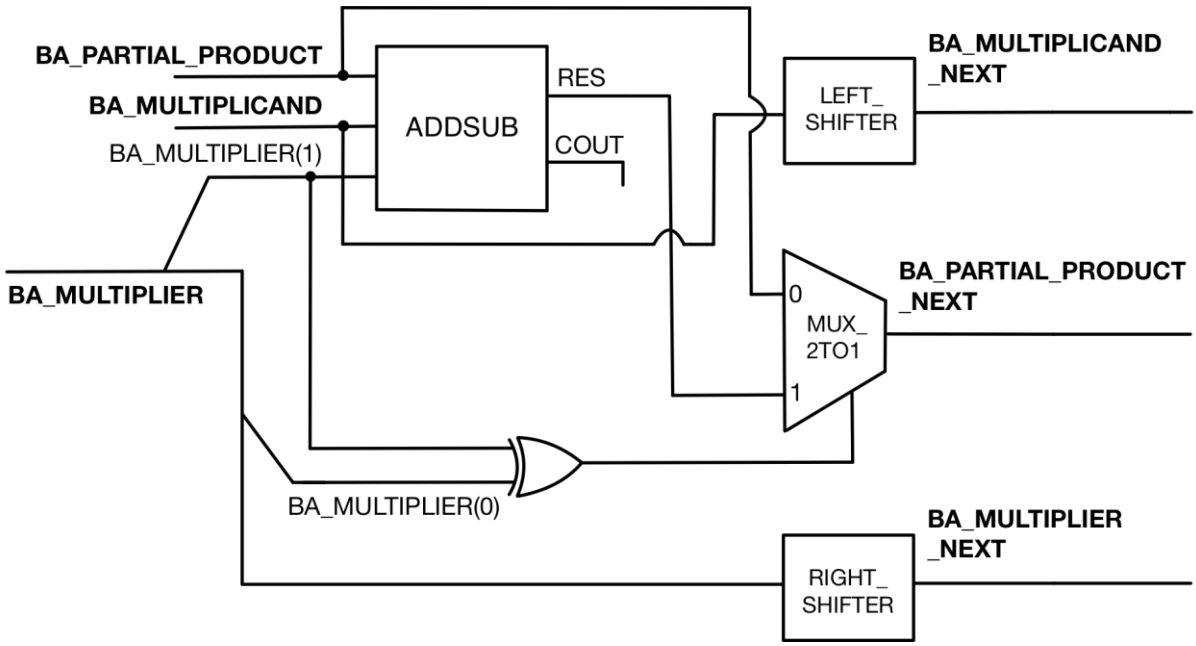
- **BA_MULTIPLICAND** rappresenta il moltiplicando a 32 bit;
- **BA_MULTIPLIER** rappresenta il moltiplicatore a 17 bit;
- **BA_PARTIAL_PRODUCT** rappresenta il prodotto parziale frutto della somma parziale precedente a 32 bit;

Analisi delle uscite:

- **BA_MULTIPLICAND_NEXT** rappresenta il moltiplicando che verrà utilizzato per la somma parziale successiva a 32 bit;
- **BA_MULTIPLIER_NEXT** rappresenta il moltiplicatore che verrà utilizzato per la somma parziale successiva a 17 bit;
- **BA_PARTIAL_PRODUCT_NEXT** rappresenta il prodotto parziale relativo alla somma parziale corrente a 32 bit;

**ARCHITETTURA:**

Il *Booth_Add* si occupa sia del calcolo della somma parziale che della preparazione degli operandi per il calcolo successivo. Perciò instanzia sia un componente dedito alla somma o sottrazione (*AddSub)* sia due shifter per preparare sia il moltiplicando (*Left_Shifter)* che il moltiplicatore (*Right_Shifter)* alla somma parziale successiva. Inoltre viene istanziato un multiplexer (*MUX_2TO1*) per la scelta relativa al valore del prodotto parziale in uscita.




**LEGENDA:**

- BA_MULTIPLIER(n) corrisponde all'n-esimo bit dell'ingresso **BA_MULTIPLIER;**

- RES corrisponde all'uscita prodotta dal componente ADDSUB;
- COUT corrisponde al riporto finale dell'operazione di somma calcolata dal componente ADDSUB. Tale riporto risulta inutilizzato.

**LOGICA:**

Il componente *Booth_Add* è fondamentale sia per il calcolo di ogni somma parziale che per la preparazione degli operandi per la somma parziale successiva. Perciò, dall'analisi del suo funzionamento, si può notare che:

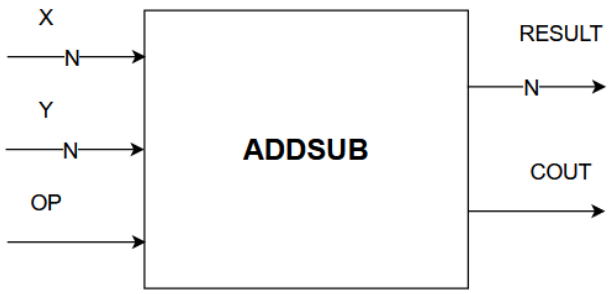
- L'*AddSub* è un componente che, in base a un bit di selezione esegue una somma o una differenza tra i due operandi in ingresso. In questo caso il bit di selezione è BA_MULTIPLIER(1) e i due operandi sono gli ingressi **BA_PARTIAL_PRODUCT** e **BA_MULTIPLICAND**. Il motivo della scelta di tale bit di selezione è dato dal fatto che,

come sottolineato dall'algoritmo, se gli ultimi 2 bit del MULTIPLIER sono '01' l'operazione da eseguire è la somma tra PARTIAL_PRODUCT e MULTIPLICAND altrimenti, se '10', l'operazione da eseguire è la differenza. L'uscita di tale componente è il segnale RES;

- Una volta calcolata la somma o la differenza tra **BA_PARTIAL_PRODUCT** e **BA_MULTIPLICAND** devo analizzare se effettivamente sono in uno dei due casi citati ('01','10') oppure se sono nel caso in cui non debba essere effettuata alcuna operazione, se gli ultimi due bit del MULTIPLICAND sono uguali ('00','11'). Tale analisi è effettuata tramite una porta XOR e successivamente l'uscita di tale porta funge da bit di selezione per il multiplexer MUX_2TO1 in cui, se gli ultimi due bit di **BA_MULTIPLIER** coincidono l'uscita del *Booth_Add* **BA_PARTIAL_PRODUCT_NEXT** corrisponde all'ingresso del *Booth_Add* **BA_PARTIAL_PRODUCT**, senò l'uscita del *Booth_Add* **BA_PARTIAL_PRODUCT_NEXT** corrisponde a RES;

- Per la preparazione degli operandi alla somma parziale successiva bisogna shiftare di una posizione verso sinistra il **BA_MULTIPLICAND** (moltiplicazione per 2) operazione effettuata dal *Left_Shifter* che produce in uscita il **BA_MULTIPLICAND_NEXT**. Invece il **BA_MULTIPLIER** deve essere shiftato di una posizione verso destra di 1 posizione (così alla prossima somma parziale verranno analizzati gli ultimi due bit successivi del MULTIPLIER), tale operazione è svolta dal Right_Shifte che produce in uscita il **BA_MUTLIPLIER_NEXT**.

*ADDSUB*




Il componente *AddSub* (Adder-Subtractor) è il componente che si occupa del calcolo effettivo di ogni somma parziale (distinguendo tra somma o differenza in base al valore del bit OP). Essendo un modulo indipendente e non creato appositamente per essere utilizzato nel moltiplicatore di Booth (a differenza del componente *Booth_Add*) ho optato per l'utilizzo di ingressi e uscite aventi dimensione non fissata e personalizzabile al momento dell'istanza.

Analisi degli ingressi:

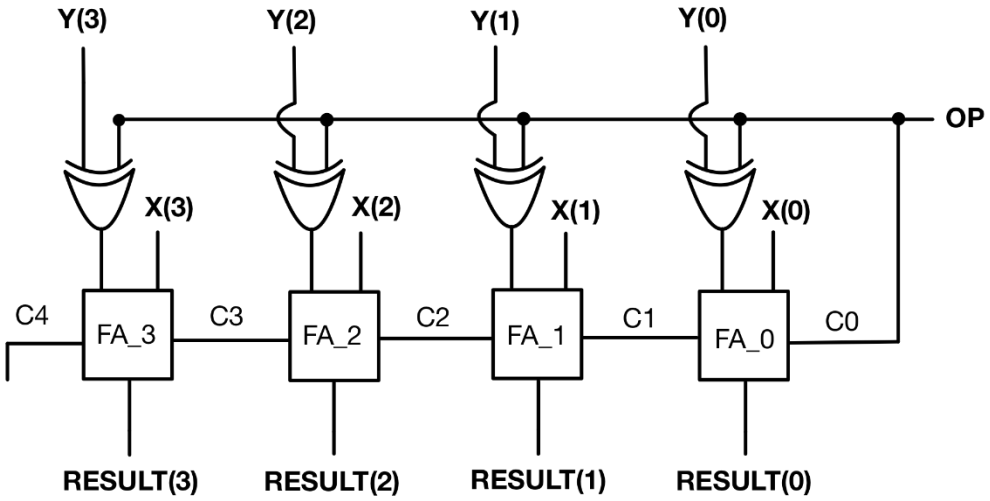
- **X** rappresenta il primo operando a N bit;
- **Y** rappresenta il secondo operando a N bit;
- **OP** rappresenta il bit relativo alla scelta dell'operazione da effettuare;

Analisi delle uscite:

- **RESULT** rappresenta il risultato della somma/differenza tra X e Y, a N bit;
- **COUT** rappresenta il bit di riporto dell'operazione di somma o differenza calcolata tra X e Y.

**ARCHITETTURA**

L'architettura raffigurata è quella di un generico *AddSub* avente segnali di ingresso a 4 bit, segnale di **OP** da 1 bit ed infine uscita a 4 bit. Vengono istanziati tanti Full_Adder quant'è il numero di bit dei segnali.

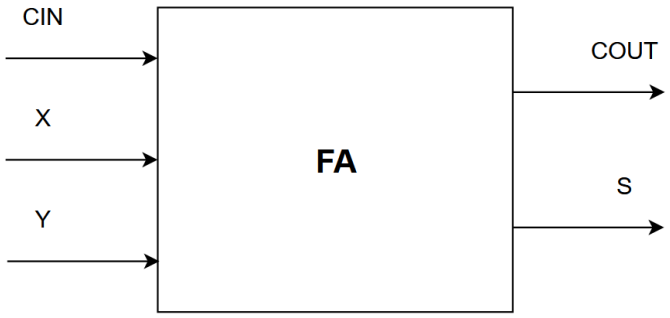



**LOGICA:**

- Addizione (OP=0): Quando OP è impostato a 0, il circuito esegue un'operazione di addizione. In questo caso, i bit di input X e Y vengono sommati direttamente. Ogni Full Adder (FA) somma i bit corrispondenti di X e Y, insieme al carry in entrata dallo stadio precedente (C0 per FA_0, l'output di FA_0 per FA_1, e così via). Il risultato di questa somma è un bit di somma (che forma parte del risultato finale) e un bit di carry, che viene passato allo stadio successivo. Questo processo si ripete per tutti e quattro gli stadi, da FA_0 a FA_3, permettendo la somma di numeri binari a 4 bit.

- Sottrazione (OP=1): Quando OP è impostato a 1, il circuito esegue un'operazione di sottrazione. In questo caso, i bit di input Y vengono invertiti (tutti gli 0 diventano 1 e viceversa) a causa del collegamento con le porte XOR. Il valore di **OP** (1) viene sommato al complemento a uno di Y e a X nel Full Adder FA_0, completando così il calcolo del complemento a due e avviando la sottrazione. Come nell'addizione, il bit di somma diventa parte del risultato finale e il bit di carry viene passato allo stadio successivo. Questo processo si ripete per tutti e quattro gli stadi, permettendo la sottrazione di numeri binari a 4 bit.

*FULL_ADDER*




Il *FA* è utilizzato per eseguire l'addizione binaria di tre bit: due bit di input e un bit di carry in entrata e produce in uscita un bit di risultato dell'operazione di somma e un bit di carry.

Analisi degli ingressi:

- **X** rappresenta il primo bit che deve essere sommato;
- **Y** rappresenta il secondo bit che deve essere sommato;

- **CIN** rappresenta il bit di carry in entrata.

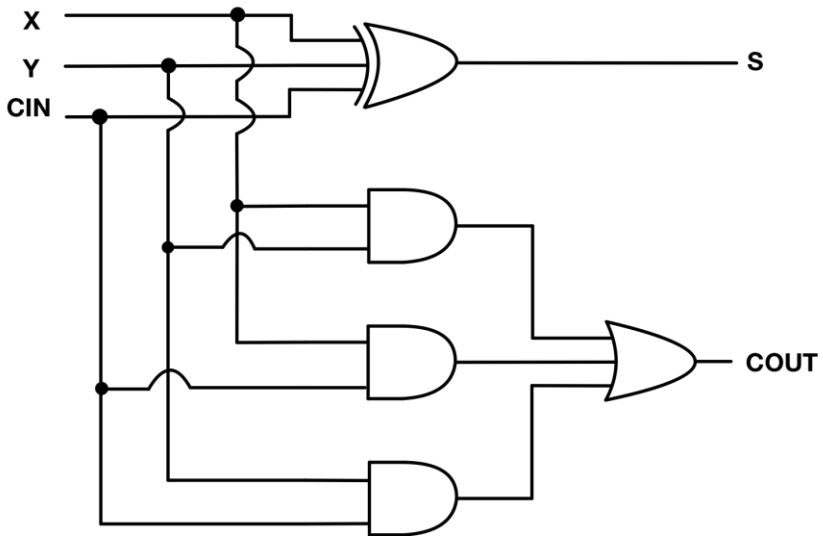
Analisi delle uscite:

- **S** rappresenta il risultato dell'addizione dei due bit di input e del bit di carry in entrata;

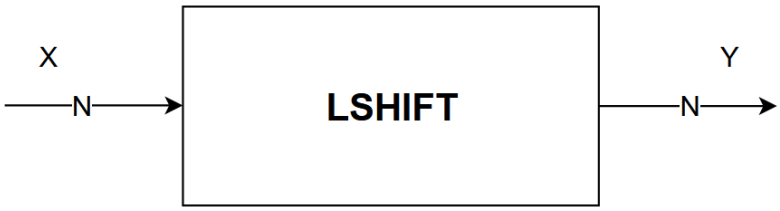
- **COUT** rappresenta il bit di carry in uscita.

**ARCHITETTURA:**

Un *Full_Adder* è composto da 1 porta XOR a 3 ingressi per il bit di somma **S**, tre porte AND e una porta OR a tre ingressi per il bit di riporto.


X
Y
CIN
S
COUT

*LEFT_SHIFTER*



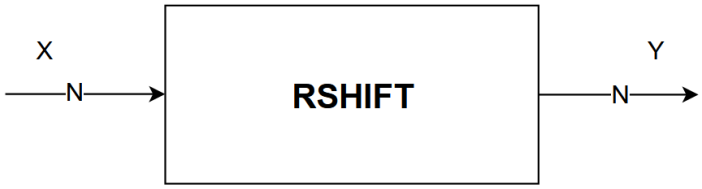

Analisi degli ingressi:

- **X** rappresenta l'ingresso a N bit;

Analisi delle uscite:

- **Y** rappresenta l'uscita a N bit;

*RIGHT_SHIFTER*



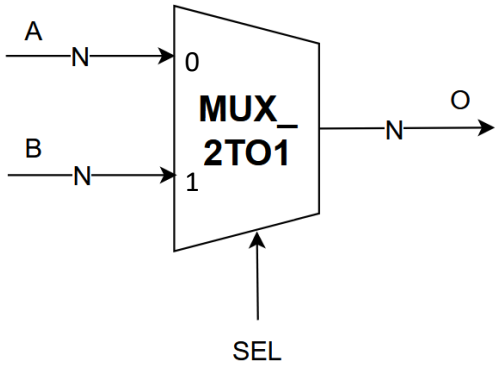

Analisi degli ingressi:

- **X** rappresenta l'ingresso a N bit;

Analisi delle uscite:

- **Y** rappresenta l'uscita a N bit;

*Mux_2TO1*




Analisi degli ingressi:

- **A** rappresenta il primo segnale in input a N bit;
- **B** rappresenta il secondo segnale in input a N bit;
- **S** rappresenta il bit di selezione;

Analisi delle uscite:

- **O** rappresenta l'uscita a N bit;

# VERIFICA

*TEST_BENCH RELATIVI AL BOOTH_MULTIPLIER*

**CASI D'USO**

- Moltiplicando positivo e moltiplicatore negativo

  - Ingressi:

**MULTIPLICAND** = "0111111100011111" (32.543)

**MULTIPLIER** = "1000000001000100" (-32.700)

  - Uscita attesa:

**PRODUCT** = "10111011010011110001000101101100" (-1.064.156.100)

- Moltiplicando positivo e moltiplicatore positivo

  - Ingressi:

**MULTIPLICAND** = "0000000000001111" (15)

**MULTIPLIER** = "0000000011000000" (192)

  - Uscita attesa:

**PRODUCT** = "00000000000000000000101101000000" (2.880)

- Moltiplicando nullo e moltiplicatore positivo

  - Ingressi:

**MULTIPLICAND** = "0000000000000000" (0)

**MULTIPLIER** = "0000000011100000" (224)

- Uscita attesa:

**PRODUCT** = “00000000000000000000000000000000” (0)

- Moltiplicando tutto a “1” e moltiplicatore tutto a “1”

  - Ingressi:

**MULTIPLICAND** = “1111111111111111” (-1)

**MULTIPLIER** = “1111111111111111” (-1)

  - Uscita attesa:

**PRODUCT** = “00000000000000000000000000000001” (1)

- Moltiplicando al massimo valore negativo e moltiplicatore al massimo valore positivo

  - Ingressi:

**MULTIPLICAND** = “1000000000000000” (-32768)

**MULTIPLIER** = “0111111111111111” (32767)

  - Uscita attesa:

**PRODUCT** = “11000000000000001000000000000000” (-1.073.709.056)