Progetto di reti Logiche

Signed Booth multiplier

Massimo Pavoni

Introduzione

Il progetto scelto prevede la progettazione, tramite Xilinx ISE[1], di un moltiplicatore combinatorio per numeri interi. Gli operandi in ingresso sono due vettori di 8 bit, rappresentati in complemento a 2, mentre l'unica uscita è il prodotto, sempre in complemento a 2, su 16 bit.

Internamente al componente è richiesto l'uso della codifica di Booth[2] (in $fig.1$, l'algoritmo citato in versione sequenziale, con codifica radix-2), per risolvere il problema della moltiplicazione con numeri anche negativi.

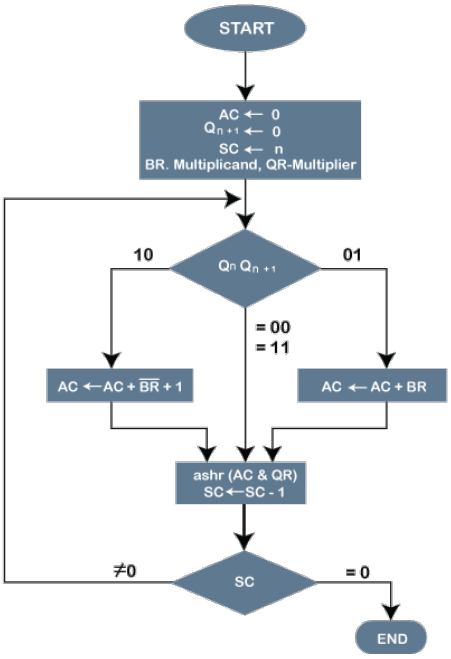
Si ipotizza di usare la codifica radix-4 ($tab.1$), piuttosto che radix-2 (intuibile in $fig.1$), al fine di ridurre sempre il numero di prodotti parziali da sommare.

I file di progetto sono disponibili in una repository su GitHub[3], intitolata "*RL22-PF-BoothMultiplier*"[4].

Il componente moltiplicatore finale è realizzato con un approccio generico, per poter non solo usare ingressi e uscita rispettivamente a 8 e 16 bit, ma anche a 12/24, 16/32, 24/48, 32/64, e così via (tuttavia limitando le dimensioni a numeri pari, a causa del modo in cui la codifica radix-4 deve essere usata).

Se la codifica è effettuata coerentemente, la somma dei prodotti parziali, spostati l'uno rispetto all'altro in modo opportuno, restituisce il risultato corretto della moltiplicazione fra i due numeri interi, in ognuno dei casi: positivo per positivo, negativo per negativo, positivo per negativo e viceversa.

$Fig.1$ – *Algoritmo di Booth (radix-2)*



$Tab.1$ – *Radix-4*

B_{i+1}	B_i	B_{i-1}	Prodotto Parziale
0	0	0	+0
0	0	1	+A
0	1	0	+A
0	1	1	+2A
1	0	0	-2A
1	0	1	-A
1	1	0	-A
1	1	1	0

1 Integrated Synthesis Environment, strumento fuori produzione per la sintesi e l'analisi di design HDL (Hardware Description Language), sviluppato da Xilinx

2 Basata sull'algoritmo ideato da Andrew Donald Booth

3 Servizio di hosting per progetti software gestiti con controllo di versione (Git)

4 Reti Logiche 2022 – Prova Finale – Booth Multiplier

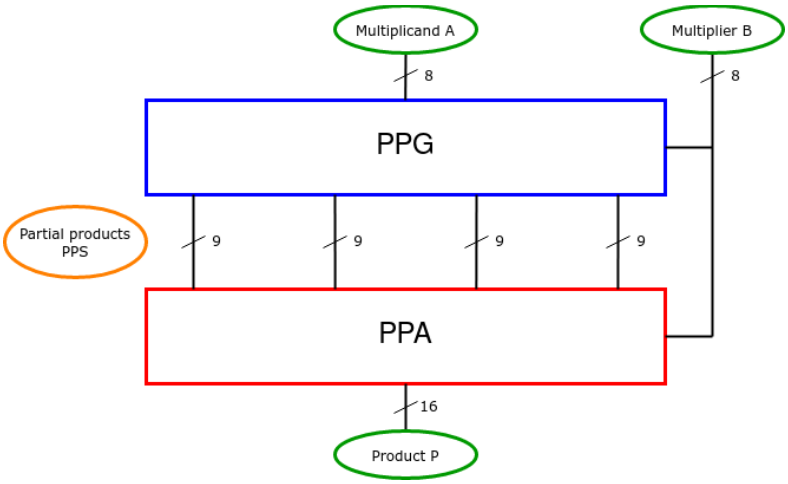
Specifica

Nel descrivere l'architettura, si considera il caso richiesto con ingressi a 8 bit e uscita a 16 bit.

Il componente, chiamato Radix-4 Booth Multiplier ($RX4BOOTHMULT$), è diviso in due parti principali:

1. PPG[5], volta a ricavare i prodotti parziali della moltiplicazione.
2. PPA[6], la quale effettua la somma dei prodotti parziali.

$Fig.2$ – Architettura ad alto livello



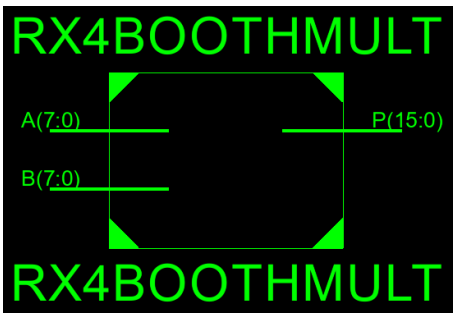
Interfaccia del sistema

I segnali in ingresso sono due numeri interi a 8 bit, chiamati moltiplicando (numero A) e moltiplicatore (numero B), codificati già in complemento a 2.

L'unico segnale di uscita è un numero intero a 2N bit, 16 in questo caso, chiamato prodotto (numero P), sempre in complemento a 2, risultato della moltiplicazione tra i due numeri interi in ingresso.

Essendo il componente puramente combinatorio, l'uscita inizierà a cambiare non appena i segnali d'ingresso, anch'essi modificati, avranno causato la propagazione dei segnali interni al componente, fino a raggiungere le porte di output.

$Fig.3$ – Interfaccia del componente finale



5 Partial Products Generation

6 Partial Products Addition

Architettura del sistema

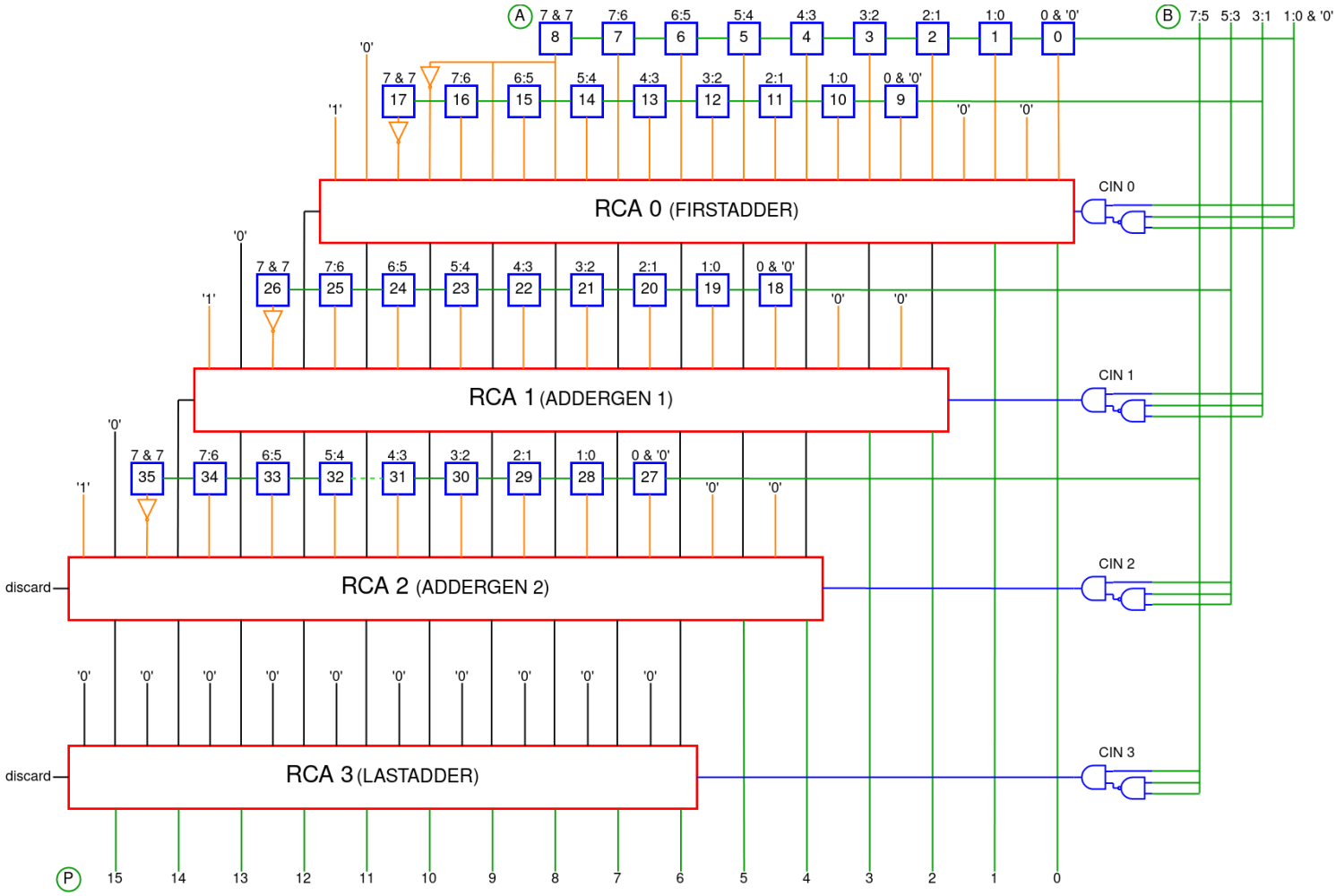
La parte di PPG è composta dai moduli:

- $MBED$[7], che effettuano la codifica/decodifica di Booth radix-4.
- Porte dedicate al calcolo dell'eventuale riporto in entrata per i prodotti parziali.

La parte di PPA è composta solamente da moduli RCA[8], i cui ingressi sono i vari prodotti parziali (modificati per avere il padding della moltiplicazione corrente e l'estensione del segno) e le somme intermedie degli adder precedenti, tranne per il sommatore finale, il quale ha lo scopo di propagare l'ultimo eventuale riporto in ingresso.

Il numero massimo di livelli strutturali innestati è pari a 3 (considerando la sequenza FA[9], RCA e $RX4BOOTHMULT$). In $fig.4$ è presentata l'architettura dettagliata (con moduli $MBED$ evidenziati dal colore blu), al fine di osservare il numero di moduli citati e le connessioni fra questi, sempre per il caso a 8 bit.

$Fig.4$ – Architettura RTL del moltiplicatore



7 Modified Booth Encoder/Decoder

8 Ripple Carry Adder

9 Full Adder

Essendo il moltiplicatore pensato in modo generico, per poter sintetizzare l'architettura con diverse dimensioni (sempre pari) per i segnali d'ingresso (l'uscita sarà sempre il doppio), molti dei segnali intermedi interni al componente sono pensati e usati a tale scopo.

Sono definite le costanti:

- OP[10], dimensione delle parole di bit che rappresentano gli operandi in ingresso, usato anche come unico valore generico che definisce l'architettura.
- PPN[11] $= OP/2$, numero di prodotti parziali in ingresso alla PPA.
- $ADDERW$[12] $= OP + 4$, dimensione degli ingressi dei moduli RCA.

Sono usati i seguenti segnali interni:

- MA[13], vettore di dimensione $OP + 2$, costituente il moltiplicando modificato per poter essere usato come ingresso per i moduli $MBED$.
- MB[14], vettore di dimensione $OP + 1$, usato come moltiplicatore modificato per la sovrapposizione dei tre bit di ingresso dei moduli $MBED$.
- PPS[15], array di PPN vettori di dimensione $OP + 1$, usati per i prodotti parziali in uscita dalla PPG.
- AXS e AYS[16], array di $PPN - 1$ vettori di dimensione $ADDERW$, usati per mappare i PPS e le AS agli ingressi dei moduli RCA.
- ACS[17], vettore di dimensione PPN, rappresentante i riporti in ingresso ai RCA.
- AS[18], array di $PPN - 1$ vettori di dimensione $ADDERW + 1$, usati per mappare le somme intermedie dei RCA assieme al corrispondente riporto in uscita.

10 Operand

11 Partial Products Number

12 Adders' Width

13 Modified A (modified multiplicand)

14 Modified B (modified multiplier)

15 Partial Products

16 Adders' Xs, Adders' Ys

17 Adders' Carries

18 Adders' Sums

Modulo MBED

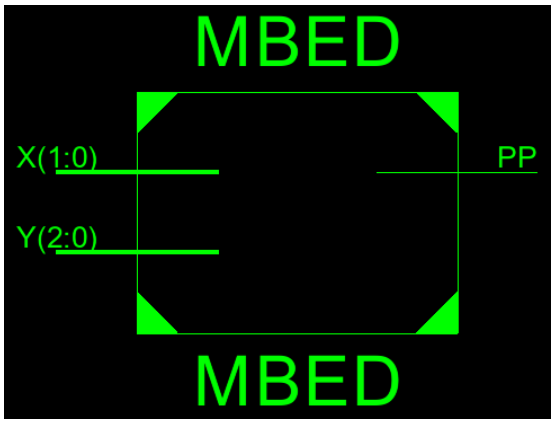
Il modulo per la codifica e decodifica radix-4 di Booth è implementazione del comportamento rappresentato dalla $tab.1$ mostrata nell'introduzione, a pagina 1. Tale tabella fornisce una descrizione precisa di come tre bit del moltiplicatore forniscano il prodotto parziale specifico; il singolo modulo $MBED$ fornirà invece l'i-esimo bit del prodotto parziale, grazie all'interfaccia descritta in $fig.5$.

I segnali di ingresso sono:

- Y, vettore di 3 bit del moltiplicatore, tali da codificare l'i-esimo bit del moltiplicando.
- X, vettore di 2 bit del moltiplicando, in cui il bit 1 rappresenta l'i-esimo bit da codificare e il bit 0 rappresenta il bit precedente, necessario per le operazioni di shift nei casi 011 e 100.

L'unico segnale d'uscita è l'i-esimo bit del prodotto parziale corrente, secondo la tabella di verità sotto riportata (semplificata, senza tutte le combinazioni di ingressi, al fine di non presentare 32 righe).

$Fig.5$ – *Interfaccia* $MBED$



$Tab.2$ – *Tabella di verità per* $MBED$

Y_2	Y_1	Y_0	PP
0	0	0	0
0	0	1	X_1
0	1	0	X_1
0	1	1	X_0
1	0	0	$\overline{X_0}$
1	0	1	$\overline{X_1}$
1	1	0	$\overline{X_1}$
1	1	1	0

Per poter usare questo modulo in parallelo per i vari prodotti parziali, occorre mappare ai due ingressi A e B del moltiplicatore i segnali corrispondenti MA e MB, composti come di seguito.

$Fig.6$ – *Mappa per gli ingressi del moltiplicatore verso la* PPG

MA (moltiplicando)

$$\begin{array}{ccccccccccc} \mathrm{PP}_8 & PP_7 & PP_6 & PP_5 & PP_4 & PP_3 & PP_2 & PP_1 & PP_0 & & \\ \mathrm{A}_7 & A_7 & A_6 & A_5 & A_4 & A_3 & A_2 & A_1 & A_0 & 0 & \end{array}$$

MB (moltiplicatore)

$$\begin{array}{cccccccccc} \mathrm{PPS}_3 & & PPS_2 & & PPS_1 & & PPS_0 & & & \\ \mathrm{B}_7 & B_6 & B_5 & B_4 & B_3 & B_2 & B_1 & B_0 & 0 & \end{array}$$

Tale operazione permette di avere sufficienti gruppi di 3 bit per il moltiplicatore e di avere i necessari bit per le eventuali operazioni di shift per quanto riguarda il moltiplicando.

La ripetizione del MSB[19] di A è utilizzata, assieme al nono e ultimo modulo $MBED$ per l'n-esimo prodotto parziale, per il calcolo del bit di segno (PP_8): si può verificare intuitivamente sulla $tab.2$ che la ripetizione di tale bit fornisce un segno coerente con l'operazione individuata dall'ingresso Y.

Quest'ultimo modulo è essenziale per non perdere l'informazione del bit di segno soprattutto quando è necessario effettuare uno shift del moltiplicando.

RCA CIN

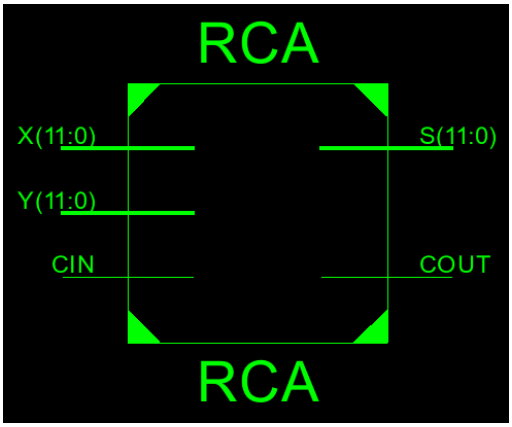
Anche se non propriamente moduli, le due porte, $NAND$ e AND, collegate al riporto in ingresso per ogni RCA, sono necessarie per completare la codifica radix-4, in quanto i moduli $MBED$ possono solamente creare un primo complemento a 1 (con shift nel caso $-2A$).

Si fa infatti uso dei CIN[20] per sommare l'eventuale bit a 1 per il complemento a 2 nei casi 100, 101 e 110. Poiché il caso 111 non deve avere il CIN a 1, ecco che non è sufficiente usare il segnale Y_2 del $MBED$ corrente: l'espressione booleana $\overline{(Y_0 \cdot Y_1)} \cdot Y_2$ ha come on-set i tre casi appena citati.

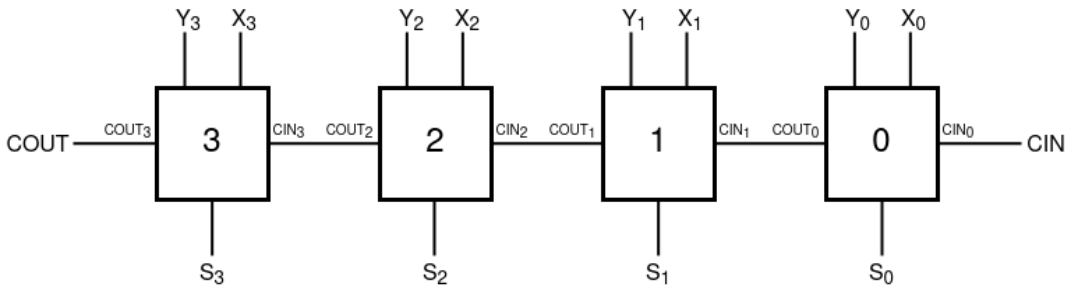
RCA

Classico modulo sommatore ripple carry, vettore di N moduli FA (i blocchi numerati in $fig.8$), utile a sommare numeri a N bit.

$Fig.7$ – Interfaccia RCA (caso a 12 bit)



$Fig.8$ – Architettura RTL del sommatore (caso a 4 bit)



19 Most Significant Bit

20 Carry Input

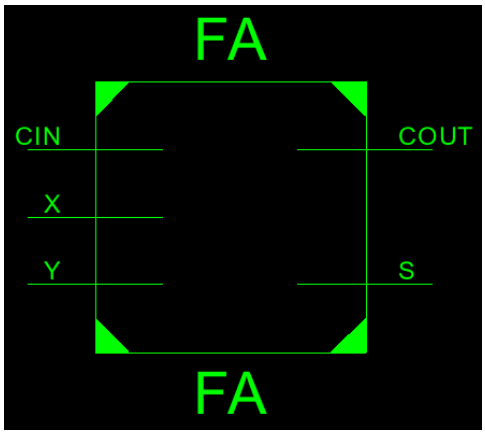
FA

Il modulo più piccolo e semplice, usato in tutte le operazioni che coinvolgono la somma di due o più bit.

Le equazioni che identificano le due uscite sono:

- S[21] $= X \oplus Y \oplus CIN$
- $COUT$[22] $= (X \cdot Y) + (X \cdot CIN) + (Y \cdot CIN)$

$Fig.9$ – Interfaccia FA



$Tab.3$ – Tabella di verità per FA

X	Y	CIN	S	COUT
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Sintesi

Il report della sintesi del modulo top level $RX4BOOTHMULT$ comunica che il massimo tempo di propagazione combinatorio è pari a circa 30.329 nanosecondi, su 19 livelli di logica.

Sono poi presentati due warning riguardanti due segnali che, come si osserva in $fig.4$, a pagina 3, vengono scartati:

- Il segnale $AS(2)(12)$, collegato all'ultimo $COUT$ del terzo RCA.
- Il $COUT$ del quarto e ultimo RCA, esplicitamente lasciato "*open*" anche nel codice del moltiplicatore.

Verifica

Ad ogni test-bench del progetto corrisponde uno script in Python per generare dei file di testo con molti test case, contenenti gli input e output previsti.

Quasi tutti i moduli sono stati testati con gli abbinamenti tra i valori possibili dei rispettivi ingressi, ricavati tramite semplice prodotto cartesiano. Solamente nel caso del RCA è stato testato un sottoinsieme degli ingressi, scelto casualmente, per via della decisione di stimolare il modulo nella sua versione a 16 bit (il numero di test case per tutte le combinazioni sarebbe stato nell'ordine dei miliardi, invece che delle decine o centinaia di migliaia, come per gli altri moduli, e questo avrebbe richiesto un tempo di simulazione non indifferente ogni volta).

21 Sum

22 Carry Output

Test-bench

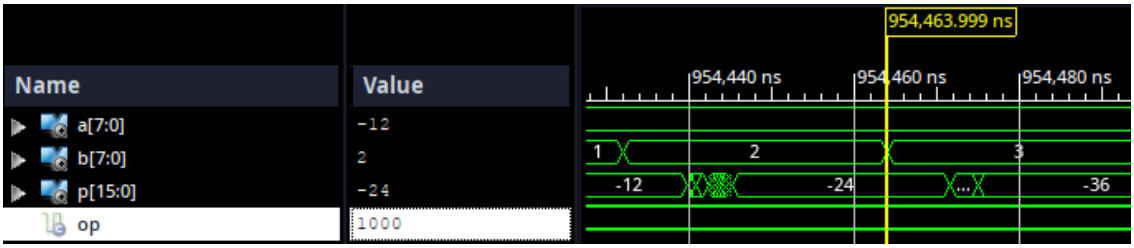
Sia per il $RX4BOOTHMULT$ che per gli altri moduli, il test-bench è definito con la UUT[23], i segnali d'ingresso e uscita del componente e due variabili di tipo "*file*" per le operazioni di IO[24] con i sopra citati file di testo generati: per poter aprire, leggere e scrivere con quest'ultimi vengono quindi usati i package "*ieee.std_logic_textio*" e "*std.textio*".

Il processo di stimolo è definito con variabili di lettura e scrittura delle linee di testo da e su file, assieme alle variabili per memorizzare i valori degli ingressi, poi inseriti nei segnali già dichiarati.

Il tempo di attesa per la propagazione puramente combinatoria dei segnali è fissato a 32 nanosecondi per il $RX4BOOTHMULT$, e il risultato ottenuto viene poi scritto in un file di testo contenente tutti gli output della simulazione. È presente un ciclo che legge e scrive una riga per volta, e viene ripetuto finché non si arriva alla fine del file con gli ingressi.

I risultati possono essere verificati in parte manualmente, grazie allo strumento di simulazione integrato in Xilinx ISE, oppure chiamando tra i due file di output (uno generato dallo script Python, l'altro creato con la simulazione) il comando "*diff*", presente in qualsiasi distribuzione Linux (la realizzazione del progetto è stata portata a termine su sistema Arch Linux, con ISE installato dalla AUR[25] e file del software concessi da Xilinx).

$Fig.10$ – *Parte di simulazione, con segnali in decimale, signed (complemento a 2)*



$Fig.11$ – *Verifica della correttezza degli output tramite il comando "diff"*

```
[21:13:25 CEST+0200] ~/Documents/Repositories/RL22-PF-BoothMultiplier/BoothMultiplier/test_gen
[damax@archie]$ diff rx4boothmult_out.txt rx4boothmult_sim.txt
[21:16:13 CEST+0200] ~/Documents/Repositories/RL22-PF-BoothMultiplier/BoothMultiplier/test_gen
[damax@archie]$
```

Casi d'uso

Il test-bench per il componente finale simula tutte le combinazioni possibili ($2^8 \cdot 2^8 = 2^{16}$ test case) in circa 2.097152 millisecondi (prendendo come tempo di delay i 32 nanosecondi sopra citati), indicando una frequenza stimata di circa 31.25MHz.

In realtà, il sistema potrebbe non supportare tale frequenza, dato che il moltiplicatore andrebbe inserito in una rete sequenziale che possa mantenere gli ingressi e osservare le uscite grazie a vari registri e un opportuno clock, il quale dovrebbe probabilmente essere scelto con un margine maggiore rispetto a quello considerato (pari a 1.671 nanosecondi in più rispetto al delay comunicato nel report della sintesi).

23 Unit Under Test

24 Input-Output

25 Arch User Repository