UNIVERSITÀ DI PISA

VHDL Design Project

Electronics and communications Systems
A.A 2019/2020
Edoardo Casapieri, Andrea Di Donato

Indice

Capitolo 1

Introduzione

1.1 Descrizione algoritmo

CORDIC(COordinate Rotation DIgital Computer) è un algoritmo iterativo utilizzato per il calcolo di funzioni trigonometriche e iperboliche sviluppato da J.E. Volder nel 1959. Lo scopo di questo progetto è stato quello di implementare l'algoritmo per poter calcolare l'arcotangente di un rapporto di numeri interi. Dati quindi due numeri interi *num* e *den* il nostro circuito deve stimare il valore dell'arcotangente del loro rapporto, ossia:

$$ris = \arctan(\frac{num}{den})$$

Supponendo che *num* e *den* siano la parte reale e immaginaria di un numero complesso z, vale:

$$z = den + j \cdot num$$

l'angolo desiderato è proprio la fase di tale numero:

$$ris = \arctan(\frac{num}{den}) \ \ con \ \ den > 0$$

dove per fase si intende l'argomento del numero complesso z, ossia l'angolo tra il vettore z e il semiasse positivo del piano.

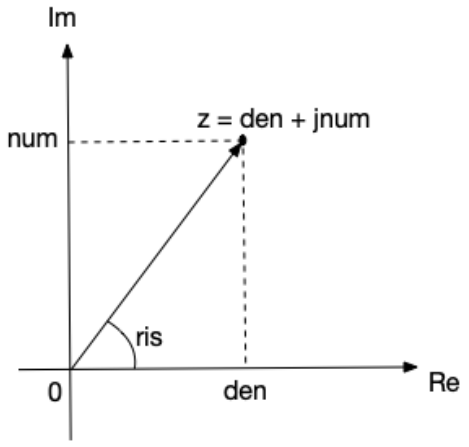

Figura 1.1: Fase del numero complesso Z

L'idea di base dell' algoritmo è quello di ruotare la fase del numero complesso z, moltiplicandolo per una successione di valori costanti. Tuttavia, le moltiplicazioni sono tutte per potenze di due, quindi in hardware possono essere implementate semplicemente mediante l'utilizzo di shifter e adder; il vantaggio del CORDIC è proprio questo ossia di non necessitare dell'utilizzo di moltiplicatori hardware.

Dato quindi un numero complesso z:

$$z = den + j \cdot num$$

verrà calcolata la sua rotazione

$$z' = den' + j \cdot num'$$

moltiplicandolo per un valore costante

$$r = I_r + j \cdot Q_r$$

Osserviamo, prima di passare alla descrizione dell'algoritmo, che quando si moltiplica una coppia di numeri complessi, la loro fase si somma mentre il loro modulo viene moltiplicato. Ugualmente, quando si moltiplica un numero complesso per il suo coniugato, la fase di quest'ultimo viene sottratta a quella del primo mentre i moduli vengono moltiplicati.

Pertanto:

- per sommare la fase di r alla fase di z:

$$z' = z \cdot r$$

$$den' = den \cdot I_{\mathrm{r}} - num \cdot Q_{\mathrm{r}}$$

$$num' = num \cdot I_{\mathrm{r}} + den \cdot Q_{\mathrm{r}}$$

- per sottrarre la fase di r alla fase di z:

$$z' = z \cdot r^*$$

$$den' = den \cdot I_{\mathrm{r}} + num \cdot Q_{\mathrm{r}}$$

$$num' = num \cdot I_{\mathrm{r}} - den \cdot Q_{\mathrm{r}}$$

Per ruotare quindi di +90 °, basta moltiplicare z per $r = 0 + j{\cdot}1$. Ugualmente, per rotare di -90 °, si moltiplica per $r = 0 - j{\cdot}1$. Nello specifico:

- moltiplicando per $r = 0 + j{\cdot}1$:

$$den' = -num$$

$$num' = den$$

- moltiplicando per *r = 0 - j·1*:

$$den' = num$$

$$num' = -den$$

Per ruotare invece di una fase minore di 90 °, si moltiplicherà il numero complesso z per un numero complesso della forma $r = 1 \pm jK$ con K = 2^{-i} dove i = 0,2,3..N-1; N è il numero di iterazioni dell'algoritmo scelte. Dato che la fase di un numero complesso Re + jIm è arctan($\frac{Im}{Re}$), la fase di 1 + j·K è arctan(K). Allo stesso modo, la fase di 1 - jK = arctan($-K$) = $-$ arctan(K). Per sommare le fasi si utilizza quindi un numero complesso r = 1 + j·K; per sottrarle utilizziamo r = 1 - j·K. Pertanto:

- per sommare le fasi, si moltiplica per r = 1 + j·K:

$$den' = den - num \cdot K = den - num \cdot 2^{-i}$$

$$num' = num + den \cdot K = num + den \cdot 2^{-i}$$

- per sottrarre le fasi, si moltiplica per r = 1 - j·K:

$$den' = den + num \cdot K = den + num \cdot 2^{-i}$$

$$num' = num - den \cdot K = num - den \cdot 2^{-i}$$

Di seguito si riporta la tabella dei valori costanti per i quali è ruotato il numero complesso z. Con i si indica il numero di iterazione dell'algoritmo:

i	K = 2^{-i}	r=1+j·K	Fase di r = atan(K)
0	1.0	1 + j1.0	45.00000°
1	0.5	1 + j0.5	26.56505°
2	0.25	1 + j0.25	14.03624°
3	0.125	1 + j0.125	7.12502°
4	0.0625	1 + j0.0625	3.57633°
5	0.03125	1 + j0.03125	1.78991°
6	0.015625	1 + j0.015625	0.89517°
7	0.007813	1 + j0.007813	0.44761°
...	...	...	...

Di qui è possibile capire il principio di funzionamento dell'algoritmo. Quello che in pratica viene fatto è ruotare il numero complesso di angoli sempre più piccoli cercando di farlo avvicinare all'asse reale positivo. Quando si è sufficientemente vicini si interrompe il ciclo, fornendo il risultato come sommatoria degli angolo di cui si è ruotato, cambiati di segno.

Nel seguente progetto si fanno al massimo 8 iterazioni, e ad ogni iterazione si va a moltiplicare per uno dei numeri complessi all'interno della tabella sopra riportata, a seconda del numero di iterazione corrente. Nello specifico, ad ogni iterazione si decide se moltiplicare per il complesso con parte immaginaria positiva oppure per il suo complesso coniugato con

lo scopo di avvicinare sempre di più all'asse reale il prodotto tra il numero complesso z e il numero complesso r.

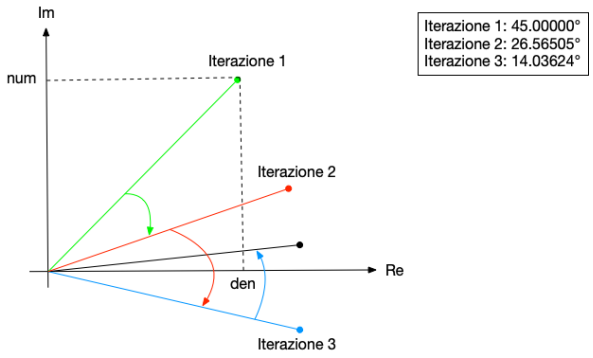

Figura 1.2: Esempio esecuzione algoritmo con i = 3

Al termine delle iterazioni si è ruotato il numero iniziale molteplici volte sempre di angoli più piccoli α_i. Detto *alpha* l'angolo iniziale, vale:

$$\alpha + \sum_{i=0}^{7} \alpha_i \cong 0$$

da cui:

$$\alpha = -\sum_{i=0}^{7} \alpha_i$$

L'algoritmo CORDIC in pseudocodice ha la seguente forma:

```
i = 0; //iterazione
ris = 0; // angolo risultante
num_i = num; //parte immaginaria del numero complesso
den_i = den; //parte reale del numero complesso
while(i < 8){
    if(num_i > 0){
        // si aggiunge l'angolo
        num_i+1 = num_i - den_i * 2^(-i);
        den_i+1 = den_i + num_i * 2^(-i);
        ris = ris + arctan(2^(-i));
    }
    else if(num_i < 0){
        // si sottrae l'angolo
        num_i+1 = num_i + den_i * 2^(-i);
        den_i+1 = den_i - num_i * 2^(-i);
        ris = ris - arctan(2^(-i));
    }else{
        // Termine dell'algoritmo a causa
        // del raggiungimento dell'asse reale
    }
    i = i + 1;
}
```

Naturalmente, maggiore è il numero di iterazioni che si fanno dell'algoritmo(nel nostro caso 8) e più preciso sarà il risultato in media. Si noti inoltre che quanto detto vale per le ipotesi di *den* > 0 e *num*$\neq 0$. Per le seguenti ipotesi l'algoritmo non è applicabile:

- se *den* < 0, la tangente cercata è la stessa che si ottiene sfruttando -*num* e -*den*, dunque si procede invertendo i segni di entrambi gli ingressi

- se $den = 0$, il numero complesso z giace sull'asse immaginario e dunque osservando il segno del num è possibile definire subito la tangente risultante:
  - se $num < 0$: l'angolo è pari a -90 °
  - se $num > 0$: l'angolo è pari a $+90°$
- se $num = 0$ si ha immediatamente che il numero complesso z giace sull'asse reale, dunque l'angolo è pari a 0

1.2 Possibili architetture

Sono riportate di seguito un insieme di possibili architetture per l'implementazione dell'algoritmo CORDIC valutando vantaggi e svantaggi per ciascuna di esse.

1.2.1 Architettura combinatoria CORDIC

L'architettura più semplice a cui possiamo pensare è una rete puramente combinatoria in cui dati i due ingressi den e num dell'algoritmo si provvede a sommare, oppure a sottrarre, ad ogni ingresso, l'ingresso opposto opportunamente shiftato, infatti:

$$num_{i+1} = num_i + den_i * 2^{-i}$$

$$den_{i+1} = den_i + num_i * 2^{-i}$$

dove i è il numero dell'iterazione(in particolare si shifta di un numero fisso di locazione ad ogni iterazione). La prima iterazione non effettua uno shift, la seconda effettua lo shift di una locazione, la terza di due e così via. Quindi, per ogni stadio i componenti elettronici saranno ripetuti e il risultato del precedente va in ingresso a quello successivo in una struttura a cascata. Tralasciamo per ora tutta la logica relativa al controllo e alla verifica della terminazione dell'algoritmo.

La semplicità di tale architettura si scontra però con alcuni svantaggi dovuti principalmente al fatto che si tratta di una rete puramente combinatoria. Questo significa che non consente ottimi valori di frequenza di pilotaggio qualora inserita in circuiti complessi, in quanto dà vita ad un percorso critico molto lungo; quindi, nelle prossime soluzioni cercheremo di ridurre la profondità e il ritardo massimo di tale circuito.

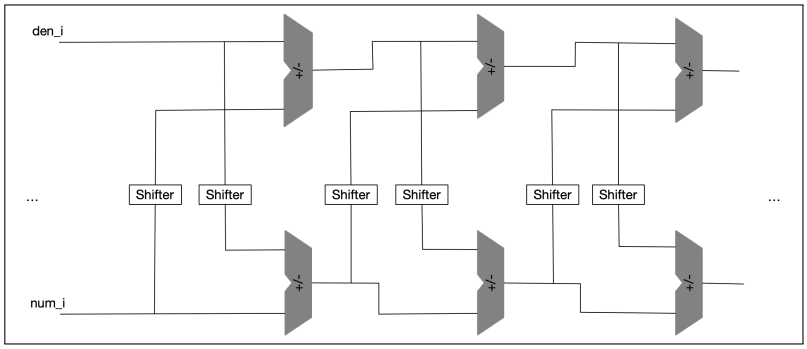

Figura 1.3: Architettura combinatoria

1.2.2 Architettura pipeline CORDIC

Riprendendo l'architettura precedente e cercando di risolvere il problema dei percorsi critici, una soluzione è quella di inserire in ogni stadio dei registri di pipeline(o stadi di pipeline) all'uscita del sommatore. In questo modo la frequenza di clock non risulta più un problema qualunque sia il numero di iterazioni dell'algoritmo, infatti, possiamo aggiungere stadi senza ridurre tale frequenza. Aggiungendo stadi, inoltre, aumenta il tempo di latenza ma aumenta anche sul throughput.

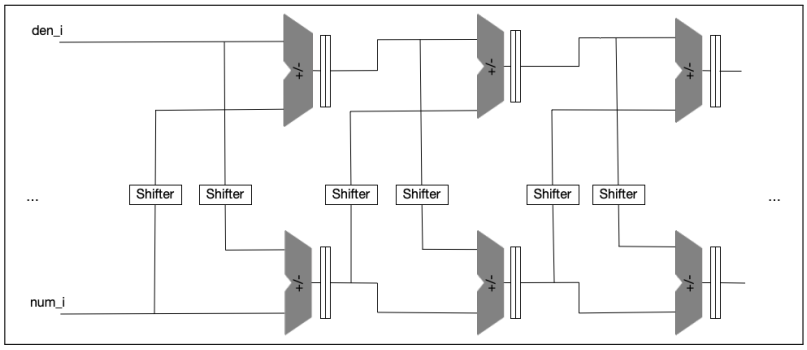

Figura 1.4: Architettura pipeline

1.2.3 Architettura CORDIC con reazione

In questa architettura si propone di fare un "avvolgimento" dell'architettura a pipeline, presa in considerazione nelle soluzioni precedenti, verso un'architettura meno complessa in termini di elementi circuitali. L'obiettivo è creare una reazione che si occupi di effettuare le giuste operazioni su parte reale ed immaginaria del numero complesso senza dover ripetere i componenti elettronici per ogni iterazione. I vantaggi di un 'architettura sono i seguenti:

- grande risparmio di area in quanto non si ripetono i componenti per ogni iterazione che si vuole eseguire.
- elevata configurabilità dell'architettura, che si può utilizzare anche per un numero variabile di passi dell'algoritmo CORDIC.
- la frequenza di clock massima consentita non subisce grandi variazioni rispetto al caso pipeline.

Uno svantaggio che si introduce è che il throughput rispetto all'architettura con pipeline si riduce, e il risultato finale è pronto solamente in un numero di cicli di clock pari al numero di iterazioni che si vuol far fare all'algoritmo. Per quanto riguarda gli shifter, potremmo utilizzare gli shifter tradizionali visti a lezione e modificarlo opportunamente così da prendere un ingresso a 12 bit e fornire il risultato sempre su 12 bit dopo un numero di cicli di clock configurato, poiché per come sono fatti, si ha uno shift all'arrivo di ogni fronte di salita. Il problema che si introduce è un pesante ritardo nell'operazione di shiftnrispetto al resto del circuito che effettua la maggior parte delle operazioni in un ciclo di clock. Possiamo quindi pensare di ottimizzare gli shifter sostituendoli con dei barrel shifter, ovvero reti puramente

combinatorie configurabili in base al numero di shift, quindi in base all'iterazione corrente, e capaci di completare l'operazione in un solo ciclo di clock.

Tale architettura presenta il miglior trade-off tra vantaggi e svantaggi; pertanto è stata selezionata questa per realizzare l'algoritmo CORDIC.

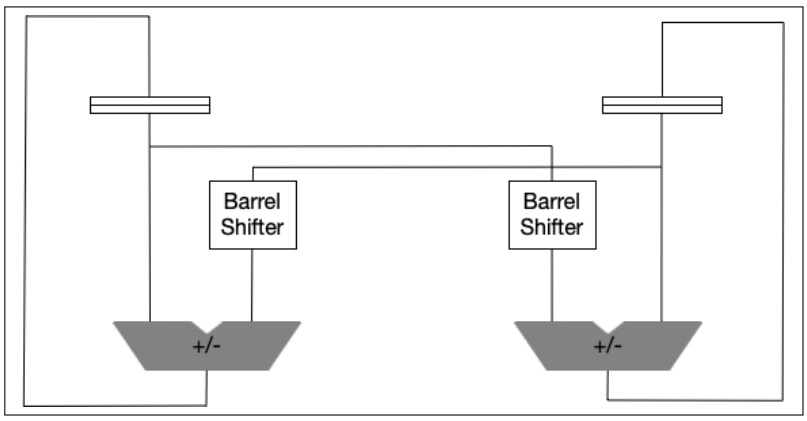

Figura 1.5: Architettura con reazione

Capitolo 2

Descrizione dell'architettura selezionata per la realizzazione

2.1 Diagramma a blocchi

Al livello più alto la struttura da utilizzare per il calcolo dell'arcotangente del rapporto di interi è la seguente:

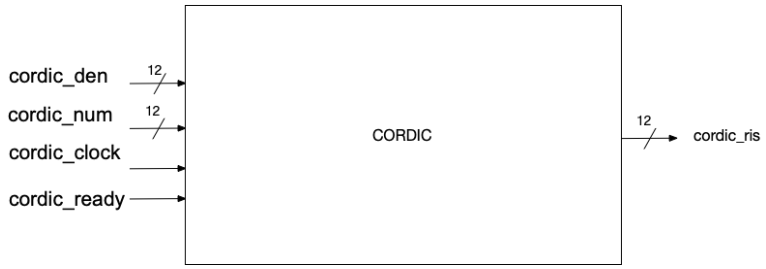

Figura 2.1: Cordic

dove:

- *cordic_num* è il numeratore della frazione di cui vogliamo calcolare l'arcotangente, supposto intero rappresentato in complemento a due su 12 bit
- *cordic_den* è il denominatore della frazione di cui vogliamo calcolare l'arcotangente, supposto intero rappresentato in complemento a due su 12 bit
- *cordic_ris* è il risultato, ossia l'angolo, ed è un numero intero rappresentato in complemento a due e in virgola fissa su 12 bit
- *cordic_ready* è una linea attiva alta, da portare ad 1 quando gli altri due ingressi sono pronti ad essere prelevati così da cominciare l'esecuzione dell'algoritmo

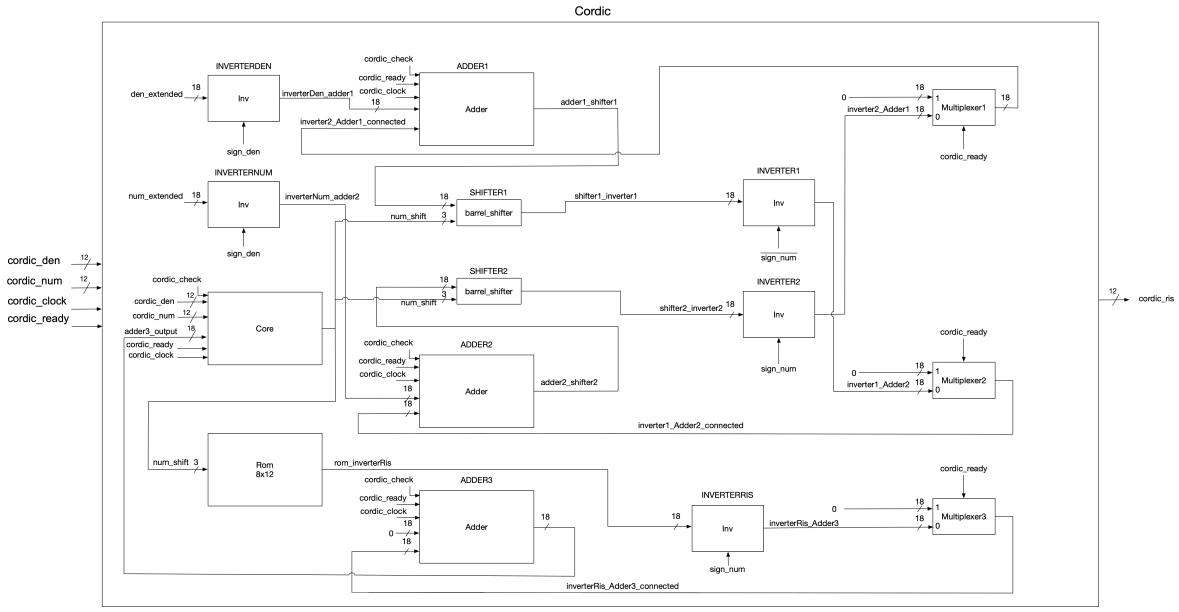

Figura 2.2: Cordic

Prima di passare alla descrizione di ciascun componente qui di seguito sono riportate le scelte di progetto effettuate:

- L'uscita è fornita su 12 bit in virgola fissa e in complemento a due. Notando inoltre che il massimo valore che dobbiamo essere in grado di rappresentare in uscita(sia positivo che negativo) è superiore a 64 in valore assoluto, vi devono essere sufficienti bit per la parte intera così da esprimere massimo il numero 90 (almeno 7 bit). È inoltre necessario un bit per il segno, e dunque rimangono solamente 4 bit per la parte frazionaria del numero in uscita. In questo caso l'uscita perde due bit di precisione rispetto ai valori memorizzati in ROM: uno per il segno e l'altro poiché la parte intera può essere più grande(nella ROM il massimo era 45). Al momento in cui sarà pronto il risultato saranno perciò scartati i due bit meno significativi.

- L'algoritmo CORDIC prevede di effettuare somme e sottrazioni di valori moltiplicati per una potenza negativa di 2; questo viene operativamente fatto tramite dei barrel shifter che effettuano la divisione per una potenza di 2 shiftando a destra il valore(shift in complemento a due) di un numero variabile di locazioni. Tuttavia, si può verificare una condizione di underflow(ossia come risultato dello shift si perdono alcuni bit meno significativi nella parte frazionaria) che può portare in certi casi a una prematura terminazione dell'algoritmo in quanto il valore di y ad una certa iterazione potrebbe essere confuso come lo zero. Il problema è risolvibile aggiungendo dei bit meno significativi al numero, e lavorando dunque con più di 12 bit nelle somme e negli shift. Nella fase di test plan saranno effettuate varie prove per capire il necessario numero di bit da aggiungere per ottenere un risultato soddisfacente. Nell'implementazione si sono aggiunti 4 bit meno significativi.

- Un altro aspetto di fondamentale importanza per il funzionamento del circuito è la necessità(una volta prelevati gli ingressi) di aggiungere dei bit più significativi nel sommatore

in modo tale da riuscire a contenere i valori sempre crescenti ad ogni iterazione; questo deve essere possibile anche nel caso in cui vengano forniti in ingresso i massimi numeri possibili ossia 2047 o -2048 in complemento a due su 12 bit. E' stato calcolato che nel peggiore dei casi, poiché la parte immaginaria tende invece a diminuire, la parte reale non raggiungerà mai 4 volte il valore originale. Perciò è necessario aggiungere anche 2 bit più significativi, quando si prelevano gli ingressi. Nell'architettura i componenti interni al circuito lavorano perciò su 18 bit e in media l'errore commesso sta intorno ad 1 grado.

- Per quanto riguarda lo stato iniziale del circuito sono stati inseriti dei multiplexer (implementati come *process* in VHDL) per l'imposizione di valori opportuni su alcuni collegamenti per la corretta partenza dell'algoritmo. Nel circuito lo stato iniziale dei registri utilizzati non è significativo in quanto viene sovrascritto appena arriva un fronte in salita del clock mentre *cordic_ready* è ad '1'; tuttavia prima di arrivare al registro vi è un sommatore dopo l'uscita del multiplexer: questo comporta la necessità di tenere fisso a zero il secondo ingresso ai blocchi adder quando *cordic_ready* è ad '1' e collegarlo al resto del circuito quando *cordic_ready* è a '0'. Anche il terzo blocco che si occupa di fare la cumulata dei valori deve inizialmente avere il secondo ingresso pari a 0 per evitare di cominciare la cumulata, al posto che da 0, dal primo valore in ROM.

2.1.1 Inv

Il componente è fatto come segue:

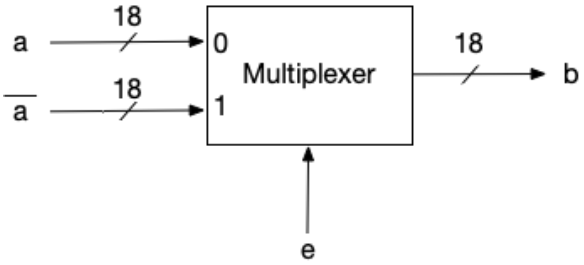

Figura 2.3: Inv

Il Multiplexer dati due bus di ingresso a 18 bit, decide quale dei due ingressi collegare all'unico bus di uscita a seconda del valore del bit di controllo e che viene fornito in ingresso. Lo scopo del componente è quello di invertire il segnale di ingresso a quando "e" assume il valore '1' altrimenti di lasciar passare "a" inalterato.

2.1.2 Barrel shifter

Il Barrel shifter a N bit(nel nostro caso 18) è una rete combinatoria nota in letteratura capace di shiftare la parola in ingresso di un numero variabile di bit a seconda del valore di un ingresso di controllo. Supponendo di voler realizzare un barrel shifter che shifta al massimo di 7 locazioni poichè si prevedono almeno 8 iterazioni per l'algoritmo, la parola di controllo deve essere su 3 bit e ogni bit di questa viene utilizzato per attivare o disattivare uno dei 3

shifter interni, i quali shiftano di un numero di bit fisso e diverso tra loro, rispettivamente di 1,2 e 4 locazioni.

2.1.3 Rom 8x12

All'interno dell'algoritmo si esegue la cumulata dei valori arctan(2^{-i}) facendo somme o differenze a seconda del segno della parte immaginaria del numero complesso associato all'iterazione corrente. La ROM, quindi ha un ingresso su 3 bit (in quanto i va da 0 a 7) corrispondente al numero dell'iterazione e che fungerà da indirizzo per così prelevare il corretto valore dell'angolo da utilizzare; questi valori sono già salvati nella ROM prima che l'algoritmo venga eseguito e rappresentati in virgola fissa con 6 bit riservati alla parte intera e 6 bit alla parte frazionaria. Di conseguenza viene introdotto un primo errore di troncamento dei valori immagazzianti in ROM dovuto alla rappresentazione su un numero finito di bit.

2.1.4 Adder

All'interno di questo componente si è deciso di utilizzare come sommatore il Ripple Carry Adder.Il Flip Flop positive edge triggered realizza un registro a 18 bit ed è in grado di memorizzare una parola di 18 bit ad ogni ciclo di clock. Si noti che si è scelto di aggiungere un ulteriore ingresso "*check*" di modo che il DFF campioni l'ingresso e quindi faccia variare l'uscita solo quando si ha sia il fronte di salita del clock sia "*check = 1*". Quest'ultimo, interno al modulo *Cordic*, viene a sua volta messo a 0 ogniqualvolta termina l'algoritmo e non è più necessario che il modulo adder effettui un ulteriore iterazione. In questo modo, non appena il risultato dell'algoritmo è disponibile, l'uscita del sommatore, e quindi il valore memorizzato nel DFF, rimarranno stabili per l'intera durata rimanente della simulazione.

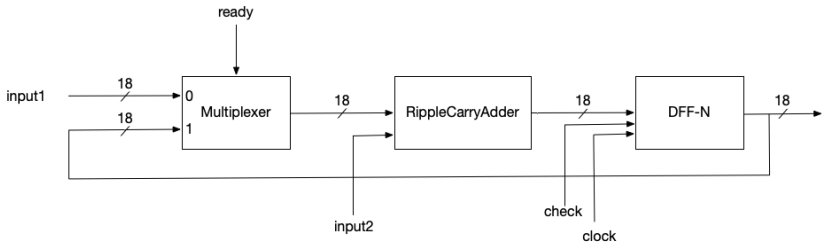

Figura 2.4: Adder

2.1.5 Core

Questo componente è di fondamentale importanza poiché racchiude tutta la logica di controllo necessaria alla corretta implementazione dell'algoritmo. Nel seguente schema a blocchi vengono evidenziati tutti i suoi ingressi e tutte le sue uscite:

Le sue funzioni principali sono:

- Gestione dei cicli e dell'incremento della variabile interna i(indice di iterazione) t, in ingresso verso la *ROM* e i *barrel shifter*.

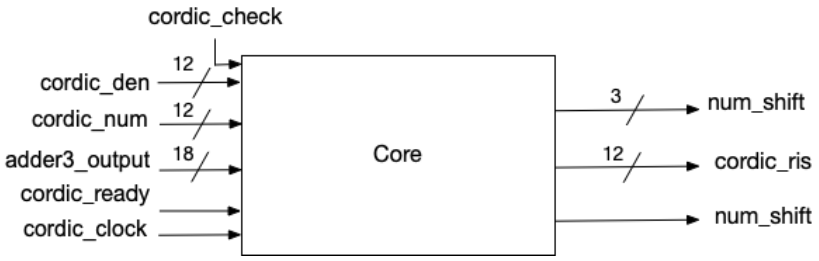

Figura 2.5: Core

- Imposizione di valori opportuni su alcuni collegamenti ai fini della corretta partenza dell'algoritmo quando *cordic_ready* sia è settato ad 1 prima dell'arrivo del fronte in salita del clock.
- Controllo della condizione di uscita precoce quando la parte immaginaria del numero complesso assume valore nullo.
- Gestione delle condizioni particolari presentate al circuito, ossia i casi in cui $num = 0$ o $den = 0$ (il caso in cui $den < 0$ è gestito automaticamente dal circuito grazie ai due *inv* collegati agl ingressi, ossia *INVERTERDEN* e *INVERTERNUM*).
- Gestione del risultato e suo allineamento ad un certo numero di bit. È proprio il *Core* a mandare in uscia *cordic_ris*.
- Gestione del segnale *cordic_check* per avviare o stoppare il calcolo della cumulata.

Capitolo 3

Codice VHDL

3.1 FullAdder.vhd

```
1  library IEEE;
2  use IEEE.std_logic_1164.all;
3
4  entity FullAdder is
5      port(
6          in_1      : in std_logic;
7          in_2      : in std_logic;
8          carry_in  : in std_logic;
9          sum       : out std_logic;
10         carry_out : out std_logic
11     );
12 end FullAdder;
13
14 architecture rtl of FullAdder is
15 begin
16     sum <= in_1 XOR in_2 XOR carry_in;
17     carry_out <= (in_1 AND in_2 ) OR (in_2 AND carry_in) OR (in_1 AND carry_in);
18 end rtl;
```

3.2 RippleCarryAdder.vhd

```
1  library IEEE;
2  use IEEE.std_logic_1164.all;
3
4  entity RippleCarryAdder is
5      generic ( Nbit : positive := 18);
6      port (
7          in_rca_1      : in std_logic_vector(Nbit-1 downto 0) ;
8          in_rca_2      : in std_logic_vector(Nbit-1 downto 0) ;
9          rca_carry_in  : in std_logic ;
10         rca_sum       : out std_logic_vector(Nbit-1 downto 0) ;
11         rca_carry_out : out std_logic
12     );
13 end RippleCarryAdder;
14
15 architecture rtl of RippleCarryAdder is
16     component FullAdder is
17         port(
18             in_1      : in std_logic;
19             in_2      : in std_logic;
20             carry_in  : in std_logic;
21             sum       : out std_logic;
22             carry_out : out std_logic
23         );
24     end component FullAdder;
25
26     signal cout_s : std_logic_vector (Nbit-1 downto 0)  ;
27
28 begin
29     -- generazione di N istanze del FullAdder
30     GEN: for i in 0 to Nbit-1 generate
```

```
31          FIRST: if i = 0 generate
32              FA1: FullAdder port map (in_rca_1(i), in_rca_2(i), rca_carry_in, rca_sum(i), cout_s(i));
33              end generate FIRST;
34
35          INTERNAL: if i > 0 and i < (Nbit-1) generate
36              FAI: FullAdder port map(in_rca_1(i), in_rca_2(i), cout_s(i-1), rca_sum(i), cout_s(i));
37              end generate INTERNAL;
38
39          LAST: if i = (Nbit-1) generate
40              FAN: FullAdder port map(in_rca_1(i), in_rca_2(i), cout_s(i-1), rca_sum(i), rca_carry_out);
41              end generate LAST;
42
43      end generate GEN;
44
45  end rtl;
```

3.3 DFF_N.vhd

```
1   library IEEE;
2   use IEEE.std_logic_1164.all;
3
4   entity DFF_N is
5       generic( Nbit : positive := 18);
6       port(
7           clk    : in std_logic ;
8           resetn : in std_logic ;
9           d      : in std_logic_vector (Nbit-1 downto 0) ;
10          q      : out std_logic_vector (Nbit-1 downto 0);
11          check  : in std_logic
12      );
13  end DFF_N;
14
15  architecture rtl of DFF_N is
16  begin
17      dff_n: process(resetn, clk,check)
18      begin
19          if resetn = '0' then
20              q <= (others => '0');
21          elsif (rising_edge(clk) and check = '1') then
22          -- se check = 1 vuol dire che l'algoritmo ancora non deve terminare e
23          -- dunque si svolgono ulteriori iterazioni.
24           q <= d;
25          end if;
26      end process dff_n;
27  end rtl;
```

3.4 adder.vhd

```
1   library IEEE;
2   use IEEE.std_logic_1164.all;
3
4   entity adder is
5       generic ( Nbit : positive := 18);
6       port (
7           add_in_1  : in std_logic_vector(Nbit-1 downto 0);
8           add_in_2  : in std_logic_vector(Nbit-1 downto 0);
9           add_out   : out std_logic_vector(Nbit-1 downto 0);
10          add_clk   : in std_logic;
11          add_ready : in std_logic;
12          add_rst   : in std_logic;
13          add_check : in std_logic
14      );
15  end adder;
16
17  architecture rtl of adder is
18
19      component RippleCarryAdder is
20      generic ( Nbit : positive := 18);
21      port (
22          in_rca_1      : in std_logic_vector(Nbit-1 downto 0) ;
23          in_rca_2      : in std_logic_vector(Nbit-1 downto 0) ;
24          rca_carry_in  : in std_logic ;
25          rca_sum       : out std_logic_vector(Nbit-1 downto 0) ;
```

```
        rca_carry_out : out std_logic
    );
    end component RippleCarryAdder;

    component DFF_N is
        port(
            clk    : in std_logic ;
            resetn : in std_logic ;
            d      : in std_logic_vector (Nbit-1 downto 0) ;
            q      : out std_logic_vector (Nbit-1 downto 0) ;
            check  : in std_logic
        );
    end component DFF_N;

    -- segnale prima del ripple carry adder
    signal before_rca: std_logic_vector(Nbit-1 downto 0);
    -- segnale dopo il ripple carry adder
    signal after_rca: std_logic_vector(Nbit-1 downto 0);
    -- segnale dopo il flip flop
    signal after_dff : std_logic_vector(Nbit-1 downto 0);

    begin

        DFF_ADDER: DFF_N
        port map(add_clk, add_rst, after_rca, after_dff, add_check);

        RCA: RippleCarryAdder port map(before_rca,add_in_2,'0',after_rca,open);

        MULTIPLEXER: process(add_in_1, after_dff, add_ready)
        begin
            if (add_ready = '1') then
                before_rca <= add_in_1;
            else
                before_rca <= after_dff;
            end if;
        end process MULTIPLEXER;


        add_out <= after_dff;

end rtl;
```

3.5 inv.vhd

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity inv is
    generic ( Nbit : positive := 18);
    port(
            xin  : in std_logic;
            yin  : in std_logic_vector(Nbit-1 downto 0);
            yout : out std_logic_vector(Nbit-1 downto 0)
    );
end inv;

architecture rtl of inv is
begin
    inv:process(xin,yin)
    begin
        if(xin = '1') then
            yout <= std_logic_vector (unsigned(not yin) + 1);
        else
            yout <= yin;
        end if;
    end process inv;
end rtl;
```

3.6 rom_8x12.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
```

```
3  use IEEE.numeric_std.all;
4
5  entity rom_8x12 is
6      port(
7          address : in STD_LOGIC_VECTOR(2 downto 0);
8          output : out STD_LOGIC_VECTOR(11 downto 0)
9          );
10 end rom_8x12;
11
12 architecture rtl of rom_8x12 is
13     signal addr_int : integer range 0 to 7;
14     type rom_t is array (0 to 7) of std_logic_vector (11 downto 0);
15      -- I valori sono rappresentati come numeri reali in virgola fissa.
16      -- 6 bit per la parte intera e 6 bit per la parte frazionaria
17     constant rom: rom_t :=
18     (
19         "101101000000",      -- 45
20         "011010100100",      -- 26.5625
21         "001110000010",      -- 14.03125
22         "000111001000",      -- 7.125
23         "000011100101",      -- 3.578125
24         "000001110011",      -- 1.796875
25         "000000111001",      -- 0.890625
26         "000000011101"       -- 0.453125
27     );
28     begin
29     addr_int <= TO_INTEGER(unsigned(address));
30     output <= rom(addr_int);
31 end rtl;
```

3.7 barrel_shifter.vhd

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.all;
3
4  entity barrel_shifter is
5      generic (Nbit : INTEGER:=18);
6      port(
7           input         : in  std_logic_vector (Nbit-1 downto 0);
8           shifted_input : out std_logic_vector (Nbit-1 downto 0);
9           N_loc         : in  std_logic_vector (2 downto 0)
10     );
11 end barrel_shifter;
12
13 architecture rtl of barrel_shifter is
14     signal result1, result2: STD_LOGIC_VECTOR (Nbit-1 downto 0);
15 begin
16     shift_one : process(input,N_loc(0))
17     begin
18         if(N_loc(0)='1') then                          -- Shifta di 1 ( il primo shifter viene attivato)
19             result1(Nbit-1) <= input(Nbit-1);
20             result1(Nbit-2 downto 0) <= input(Nbit-1 downto 1);
21         else
22             result1(Nbit-1 downto 0) <= input(Nbit-1 downto 0);    -- il primo shifter rimane disattivo
23         end if;
24     end process;
25
26     shift_two : process(result1,N_loc(1))
27     begin
28         if(N_loc(1)='1') then                          -- Shifta di 2 (il secondo shifter viene attivato)
29             result2(Nbit-1) <= result1(Nbit-1);
30             result2(Nbit-2) <= result1(Nbit-1);
31             result2(Nbit-3 downto 0) <= result1(Nbit-1 downto 2);
32         else
33             result2(Nbit-1 downto 0) <= result1(Nbit-1 downto 0);  -- il secondo shifter rimane disattivo
34         end if;
35     end process;
36
37     shift_four : process(result2, N_loc(2))
38     begin
39         if(N_loc(2)='1') then                          -- Shifta di 4 (il terzo shifter viene attivato)
40             shifted_input(Nbit-1) <= result2(Nbit-1);
41             shifted_input(Nbit-2) <= result2(Nbit-1);
42             shifted_input(Nbit-3) <= result2(Nbit-1);
43             shifted_input(Nbit-4) <= result2(Nbit-1);
44             shifted_input(Nbit-5 downto 0) <= result2(Nbit-1 downto 4);
45         else
46             shifted_input(Nbit-1 downto 0) <= result2(Nbit-1 downto 0); -- il terzo shifter rimane disattivo
```

```
        end if;
    end process;
end rtl;
```

3.8 Cordic.vhd

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE IEEE.numeric_std.all;

entity Cordic is
   generic ( N : positive := 12);
   port( cordic_den      :  in std_logic_vector  (N-1 downto 0);
         cordic_num      :  in std_logic_vector  (N-1 downto 0);
         cordic_ris      :  out std_logic_vector (N-1 downto 0);
         cordic_clock    :  in  std_logic;
         cordic_ready    :  in  std_logic
   );
end Cordic;

architecture rtl of CORDIC is

component inv is
    generic ( Nbit : positive := 18);
    port(
        xin  : in std_logic;
        yin  : in std_logic_vector(Nbit-1 downto 0);
        yout : out std_logic_vector(Nbit-1 downto 0)
    );
end component inv;

component adder is
    generic ( Nbit : positive := 18);
    port (
        add_in_1  : in std_logic_vector(Nbit-1 downto 0);
        add_in_2  : in std_logic_vector(Nbit-1 downto 0);
        add_out   : out std_logic_vector(Nbit-1 downto 0);
        add_clk   : in std_logic;
        add_ready : in std_logic;
        add_rst   : in std_logic;
        add_check : in std_logic
    );
end component adder;

component barrel_shifter is
    generic (Nbit : INTEGER:=18);
    port(
         input         : in  std_logic_vector (Nbit-1 downto 0);
         shifted_input : out std_logic_vector (Nbit-1 downto 0);
         N_loc         : in  std_logic_vector (2 downto 0)
    );
end component barrel_shifter;

component rom_8x12 is
    port(
        address : in STD_LOGIC_vector(2 downto 0);
        output  : out STD_LOGIC_vector(11 downto 0)
    );
end component rom_8x12;

-- utilizzato per avviare o terminare il calcolo della cumulata
signal cordic_check : std_logic := '1';
-- Segnale contenente il segno del denominatore
signal sign_den :std_logic;
-- Segnale contenente il segno del numeratore
signal sign_num :std_logic;
-- Segnale cotenente il complemento del segno del numeratore
signal notsign_num :std_logic;

-- Segnali utilizzati per estendere i due ingressi da 12 a 18 bit
signal den_extended: std_logic_vector (17 downto 0);
signal num_extended: std_logic_vector (17 downto 0);

-- Segnale utilizzato per indicare il numero di shit da effettuare
signal num_shift: std_logic_vector (2 downto 0);

-- Segnali dall'inverter del den o del num all'adder i-esimo
signal inverterDen_adder1: std_logic_vector (17 downto 0);
```

```
73  signal inverterNum_adder2: std_logic_vector (17 downto 0);
74
75  -- Segnali dall'inverter i-esimo all'adder i-esimo
76  signal inverter2_Adder1: std_logic_vector (17 downto 0);
77  signal inverter2_Adder1_connected: std_logic_vector (17 downto 0);
78  signal inverter1_Adder2: std_logic_vector (17 downto 0);
79  signal inverter1_Adder2_connected: std_logic_vector (17 downto 0);
80  signal inverterRis_Adder3: std_logic_vector (17 downto 0);
81  signal inverterRis_Adder3_connected: std_logic_vector (17 downto 0);
82
83  -- Segnali dall'adder i-esimo allo shifter i-esimo
84  signal adder1_shifter1: std_logic_vector (17 downto 0);
85  signal adder2_shifter2: std_logic_vector (17 downto 0);
86  signal adder3_output: std_logic_vector (17 downto 0);
87
88  -- Segnali dallo shifter all'inverter i-esimo
89  signal shifter1_inverter1: std_logic_vector (17 downto 0);
90  signal shifter2_inverter2: std_logic_vector (17 downto 0);
91
92  --Segnale dalla rom all'inverterRis
93  signal rom_inverterRis: std_logic_vector (17 downto 0);
94
95  begin
96
97      -- INVERTER
98
99      INVERTERDEN: inv
100         port map (
101                     xin => sign_den,
102                     yin => den_extended,
103                     yout => inverterDen_adder1
104         );
105
106     INVERTERNUM: inv
107         port map (
108                     xin => sign_den,
109                     yin => num_extended,
110                     yout => inverterNum_adder2
111         );
112
113     INVERTER1: inv
114         port map (
115                     xin => notsign_num,
116                     yin => shifter1_inverter1,
117                     yout => inverter1_Adder2
118         );
119
120     INVERTER2: inv
121         port map (
122                     xin => sign_num,
123                     yin => shifter2_inverter2,
124                     yout => inverter2_Adder1
125         );
126
127     INVERTERRIS: inv
128         port map (
129                     xin => sign_num,
130                     yin => rom_inverterRis,
131                     yout => inverterRis_Adder3
132         );
133
134     -- ADDER
135
136     -- Lo stato iniziale dei registri utilizzati non è significativo
137     -- in quanto viene sovrascritto appena arriva un fronte in salita del clock
138     -- mentre ready rimane ad '1'. Per tale motivo non vi sono segnali di reset.
139
140     ADDER1: adder
141         port map (
142                     add_in_1 => inverterDen_adder1,
143                     add_in_2 => inverter2_Adder1_connected,
144                     add_out => adder1_shifter1,
145                     add_clk => cordic_clock,
146                     add_ready => cordic_ready,
147                     add_rst => '1',
148                     add_check => cordic_check
149         );
150
151     ADDER2: adder
152         port map (
153                     add_in_1 => inverterNum_adder2,
154                     add_in_2 => inverter1_Adder2_connected,
```

```
155                     add_out => adder2_shifter2,
156                     add_clk => cordic_clock,
157                     add_ready => cordic_ready,
158                     add_rst => '1',
159                     add_check => cordic_check
160         );
161 
162     -- Il calcolo della cumulata deve partire da 0
163     ADDER3: adder
164         port map (
165                     add_in_1 => B"0000000000000000000",
166                     add_in_2 => inverterRis_Adder3_connected,
167                     add_out => adder3_output,
168                     add_clk => cordic_clock,
169                     add_ready => cordic_ready,
170                     add_rst => '1',
171                     add_check => cordic_check
172         );
173 
174 
175     -- SHIFTER
176 
177     SHIFTER1: barrel_shifter
178         port map (
179                     input => adder1_shifter1,
180                     shifted_input => shifter1_inverter1,
181                     N_loc => num_shift
182         );
183 
184     SHIFTER2: barrel_shifter
185         port map (
186                     input => adder2_shifter2,
187                     shifted_input => shifter2_inverter2,
188                     N_loc => num_shift
189         );
190 
191     -- ROM
192 
193     ROM: rom_8x12
194         port map(
195                     address => num_shift,
196                     output => rom_inverterRis(11 downto 0)
197         );
198 
199     -- MULTIPLEXER
200 
201     -- Multiplexer tra l'inverter2 e l'adder 1
202     MULTIPLEXER1:process(cordic_ready,inverter2_Adder1)
203     begin
204         if(cordic_ready = '1') then
205                 inverter2_Adder1_connected <= (others => '0');
206         else
207                 inverter2_Adder1_connected <= inverter2_Adder1;
208         end if;
209     end process MULTIPLEXER1;
210 
211     -- Multiplexer tra l'inverter1 e l'adder 2
212     MULTIPLEXER2:process(cordic_ready,inverter1_Adder2)
213     begin
214         if(cordic_ready = '1') then
215                 inverter1_Adder2_connected <= (others => '0');
216         else
217                 inverter1_Adder2_connected <= inverter1_Adder2;
218         end if;
219     end process MULTIPLEXER2;
220 
221     -- Multiplexer tra l'inverterRis e l'adder 3
222     MULTIPLEXER3:process(cordic_ready,inverterRis_Adder3)
223     begin
224         if(cordic_ready = '1') then
225                 inverterRis_Adder3_connected <= (others => '0');
226         else
227                 inverterRis_Adder3_connected <= inverterRis_Adder3;
228         end if;
229     end process MULTIPLEXER3;
230 
231     --CORE
232 
233     -- Core: all'interno si effettuano le iterazioni e viene stabilito quando
234     -- ritornare il risultato
235     CORE:process(cordic_clock, cordic_check)
236     -- core_start stabilisce quando devono cominciare le iterazioni
```

```
237     variable core_start:std_logic := '0';
238     -- core_counter viene utilizzata per tenere conto del numero di iterazioni
239     variable core_counter: integer range 0 to 15;
240     begin
241         if (rising_edge(cordic_clock) and cordic_check = '1') then
242             -- cordic_ready viene utilizzata per comunicare che l'utente
243             -- ha fornito in ingresso un valore valido
244             if (cordic_ready ='1') then
245                 -- Se il numeratore risulta uguale a 0 viene restituito 0
246                 if (cordic_num = B"000000000000") then
247                     core_start := '0';
248                     cordic_ris <= B"000000000000";
249                     cordic_check <= '0';
250                 -- Se il denominatore risulta uguale a 0 viene restituito +90 o -90 a
251                 -- seconda del segno del numeratore
252                 elsif (cordic_den = B"000000000000") then
253                         core_start := '0';
254                         if(cordic_num(11) = '0') then
255                             cordic_ris <= B"010110100000"; -- 1 bit per il segno
256                                                            -- 7 bit per la parte intera
257                                                            -- 4 bit per la parte frazionaria
258                         else
259                             cordic_ris <= B"101001100000";
260                         end if;
261                         cordic_check <= '0';
262                 else
263                         core_start := '1';
264                         core_counter := 0;
265                         -- Si inizializza num_shift che viene utilizzato anche
266                         -- per accedere alla rom
267                         num_shift <=std_logic_vector(to_unsigned(core_counter,3));
268                 end if;
269             elsif ( core_start = '1'  and core_counter < 8) then
270                     if ( adder2_shifter2 = B"0000000000000000000") then
271                         -- Il risultato è stato trovato poichè num vale 0
272                         core_start := '0';
273                         core_counter := 0;
274 
275                         cordic_ris (10 downto 0) <= adder3_output(12 downto 2);
276                         cordic_ris(11) <= adder3_output(17);
277                         cordic_check <= '0';
278                     else
279                         -- risultato ancora non trovato
280                         -- occorre effettuare una nuova iterazione
281                         core_counter := core_counter +1;
282                         num_shift <= std_logic_vector(to_unsigned(core_counter,3));
283                     end if;
284             elsif ( core_start = '1') then
285                     -- Risultato trovato poichè sono terminate le iterazioni
286                     core_start := '0';
287                     core_counter := 0;
288 
289                     cordic_ris (10 downto 0) <= adder3_output(12 downto 2);
290                     cordic_ris(11) <= adder3_output(17);
291                     cordic_check <= '0';
292 
293             end if;
294         end if;
295     end process CORE;
296 
297     -- Estrazione del segno del numeratore e del denominatore
298     -- Questi sono utilizzati come variabili di comando degli inverter
299     sign_den <=  cordic_den(11);
300     sign_num <=  adder2_shifter2(17);
301     notsign_num <= NOT sign_num;
302 
303     rom_inverterRis(17 downto 12) <= B"000000";
304 
305     -- Estensione del numeratore e del denominatore da 12 a 18 bit
306 
307     num_extended(15 downto 4) <= cordic_num;
308     num_extended(17) <= cordic_num(11);
309     num_extended(16) <= cordic_num(11);
310     num_extended(3) <= '0';
311     num_extended(2) <= '0';
312     num_extended(1) <= '0';
313     num_extended(0) <= '0';
314 
315     den_extended(15 downto 4) <= cordic_den;
316     den_extended(17) <= cordic_den(11);
317     den_extended(16) <= cordic_den(11);
318     den_extended(3) <= '0';
```

```
319         den_extended(2) <= '0';
320         den_extended(1) <= '0';
321         den_extended(0) <= '0';
322 
323 end rtl;
```

Capitolo 4

Test-Plan

Per verificare i requisiti funzionali ovvero la correttezza dei risultati forniti dal circuito, abbiamo scritto del codice VHDL che permettesse di simulare il pilotaggio del CORDIC mediante l'ambiente ModelSim. La verifica del corretto funzionamento è stata prima effettuata su ogni singolo componente interno al CORDIC, ma queste simulazioni saranno tralasciate. In tale ambiente vengono forniti in ingresso due numeri interi su 12 bit indicati come *ingr_x_tb* e *ingr_y_tb*; questi rappresentano rispettivamente il denominatore e il numeratore nell'architettura del circuito. Gli altri ingressi vengono opportunamente pilotati dando al clock un periodo di 12ns e facendo andare ad '1' il *ready* dopo 12ns dall'inizio della simulazione per poi riportarlo a '0' dopo 36ns, così da far partire correttamente il calcolo della cumulata. Il file di test bench realizzato permette inoltre di settare a piacimento i valori dei due numeri interi e di controllare pertanto le uscite restituite dal modulo CORDIC; in particolare l'uscita *ris_tb*, su 12 bit, dovrebbe contenere, a meno di errori, il risultato cercato.

Nella seguente tabella vengono riportati alcuni risultati ottenuti nella fase di testing del circuito. In particolare in tabella si osservano i valori di *den* e *num* in input, e l'output *ris* corrispondente ottenuto, inoltre viene riportato anche l'errore commesso dal modulo rispetto al valore dell'arcotangente atteso; si noti che i seguenti valori sono stati scelti in modo da coprire approssimativamente tutti i possibili casi d'uso. Sono state inoltre fatte varie prove variando il numero di bit aggiunti al sommatore e osservando come varia l'errore, così da poter individuare il numero di bit che permetta di ottenere un valore dell'arcotangente più vicino possibile a quello atteso.

4.1 Risultati della fase di testing

Den	Num	Ris(gradi)	Errore	Num Bit Utilizzati
0	2	90	0	-
0	-1	-90	0	-
2	0	0	0	-
-1	0	0	0	-
2	-2	-45	0	-
3	3	45	0	-

Den	Num	Ris(gradi)	Errore	Num Bit Utilizzati
2	1	21.756	3.809	14
2	1	24.455	2.11	16
2	1	25.672	0.893	17
2	1	25.812	0.753	18
2	1	25.976	0.589	19
2	1	25.3132	1.2513	20
12	13	47.25	0.04	18
2	-1	-25.75	0.815	18
3	1	18.437	0.002	18
3	-1	-18.437	0.002	18
3	2	32.875	0.815	18
15	23	55.825	1.063	17
15	23	56.125	0.763	18
15	-23	-56.125	0.763	18
15	23	56.275	0.613	19
-15	23	-56.125	0.763	18
1	2	64.187	0.753	18
35	512	86	0.089	18
-35	-512	86	0.089	18
460	121	14.337	0.390	17
460	121	14.562	0.175	18
460	-121	-14.562	0.175	18
3	512	89.562	0.101	18
2047	1	0.375	0.347	18
-2047	1	-0.437	0.409	18
1	2047	89.56	0.412	18
0	0	0	0	-
60	1	1.3125	0.357	18
-60	1	-1.3125	0.357	18
9	34	75.312	0.139	18
9	-34	-75.312	0.139	18
1	50	88.575	0.279	18
78	125	58.345	0.309	18
-78	-125	58.345	0.309	18
1	25	87.540	0.169	18
25	1	2.187	0.103	18
34	56	58.560	0.176	18
34	56	58.670	0.066	19
567	1078	62.062	0.194	18
-567	-1078	62.062	0.194	18
898	560	31.670	0.277	18

4.2 Cordic_tb.vhd

```
1  library IEEE;
2  use IEEE.std_logic_1164.all;
3
4  entity Cordic_tb is
5  end entity Cordic_tb;
6
7  architecture test of Cordic_tb is
8
9  constant T_CLK: time := 12 ns;
10 constant N_tb : integer := 12;
11
12 signal clk_tb : std_logic := '0';
13 signal end_sim : std_logic := '0';
14
15 signal ready_tb : std_logic := '0';
16
17 signal ingr_x_tb : std_logic_vector (11 downto 0) := "000000000010"; -- Inserire qui il valore del denominatore
18 signal ingr_y_tb : std_logic_vector (11 downto 0) := "111111111111"; -- Inserire qui il valore del numeratore
19 signal ris_tb : std_logic_vector(N_tb-1 downto 0);
20
21 component Cordic is
22 generic ( N : positive := 12);
23    port(
24          cordic_den     :  in std_logic_VECTOR (11 downto 0);
25          cordic_num     :  in std_logic_VECTOR (11 downto 0);
26          cordic_ris       :  out std_logic_VECTOR (11 downto 0);
27          cordic_clock          :  in  std_logic;
28          cordic_ready       :  in  std_logic
29     );
30 end component Cordic;
31
32 begin
33
34     clk_tb <= not(clk_tb) or end_sim after T_CLK/2;
35     ready_tb <= '1' after T_CLK, '0' after 3*T_CLK;
36     end_sim <= '1' after 100*T_CLK;
37
38     dut: CORDIC
39     generic map (N => N_tb)
40     port map(
41                 cordic_den => ingr_x_tb,
42                 cordic_num => ingr_y_tb,
43                 cordic_ris => ris_tb,
44                 cordic_clock => clk_tb,
45                 cordic_ready => ready_tb
46     );
47
48 end architecture test;
```

Capitolo 5

Sintesi e Implementazione

Dato che durante la fase di testing in ambiente *ModelSim* si sono ottenuti risultati soddisfacenti si è passati alla fase di sintesi e implementazione. Questa è stata svolta utilizzando Xilinx Vivado 2019.2.

Come primo passo si è svolta l'RTL analysis. Qui viene analizzato il codice .vhd a livello puramente funzionale, quindi non si parla di silicio ma di blocco logico, quindi di funzione; viene estratto uno schematico che si chiama per l'appunto RTL ossia blocchi *register transfer level* che permette di capire cosa è stato dedotto dall'ambiente da quello che è stato scritto nel codice. Osservando lo schematico risultante da questa analisi e tenendo conto dei warning è possibile verificare se quello creato dall'ambiente corrisponde con quello desiderato. Questa fase è utile sopratutto per avere un feedback a livello visivo. Qui di seguito si riporta la struttura interna dell' *INVERTER1* e dello *SHIFTER1* risultante dall'analisi RTL; al termine di questa non sono stati riportati messaggi di warning.

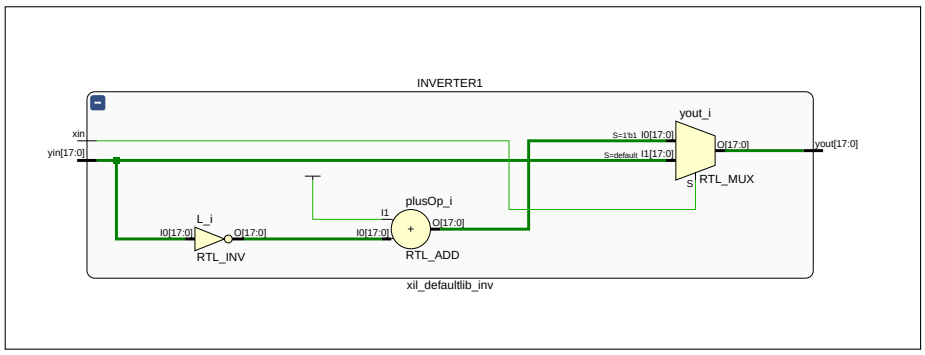

Figura 5.1: INVERTER1

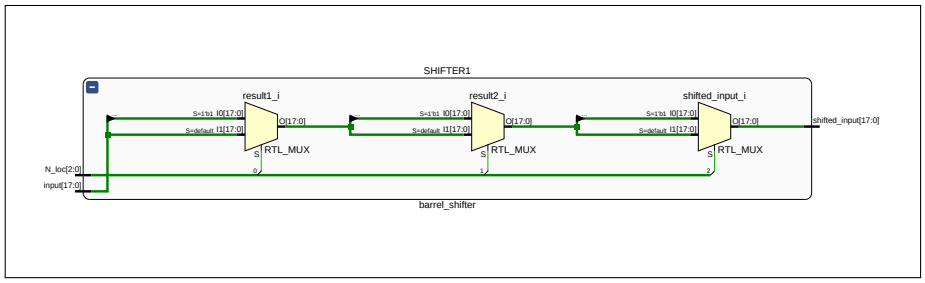

Figura 5.2: SHIFTER1

5.1 Sintesi e vincoli temporali

Dopo che è stato prodotto lo schematico RTL, si è avviata la fase di sintesi. Come development board si è utilizzato la *ZYNQ XC7Z010-1CLG400C*; l'obiettivo di questa architettura è di utilizzare una frequenza di clock pari a 83.3 MHz(12ns). Questo è l'unico vincolo inserito nel file *Cordic.xdc*.

```
1 create_clock -period 12.000 -name clk83 -waveform {0.000 6.000} [get_ports cordic_clock]
```

All'interno di Vivado il vincolo è stato rispettato con successo, con i seguenti risultati:

Tipo	Worst Slack
Setup	2.564 ns
Hold	0.142 ns

Come esempio si riporta di seguito il path classificato come più lungo:

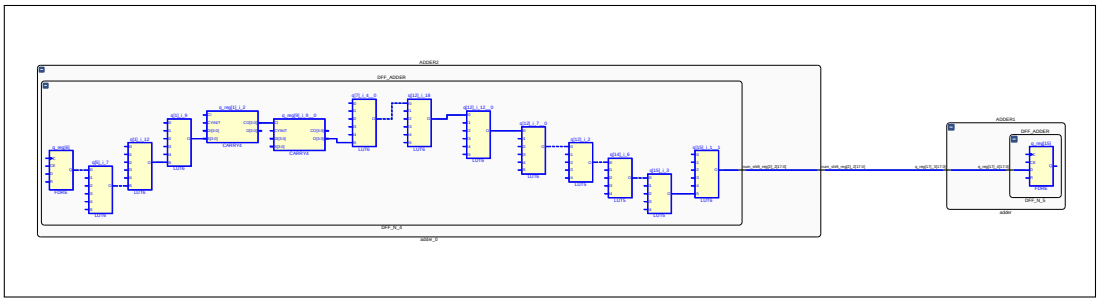

Figura 5.3

questo collega il DFF_ADDER all'interno dell'ADDER1 con il DFF_ADDER all'interno dell'ADDER2: il suo ritardo di propagazione è pari a 9.285ns.

5.2 Implementazione

È proprio in questa fase che viene implementato il design su FPGA; nella fase di sintesi infatti viene fatta una previsione grossolana dei componenti del design e della loro implementazione. Effettuata quindi l'implementazione in *Vivado IDE* si ottengono i seguenti nuovi valori:

Tipo	Worst Slack
Setup	2.310 ns
Hold	0.163 ns

e lo stesso path critico evidenziato precedentemente risulta avere adesso un ritardo di propagazione pari a 9.597 ns. L'On-Chip Power totale è pari a 0.095W mentre le risorse utilizzate sono:

Resource	Utilization	Available	Utilization %
LUT	313	17600	1.78
FF	75	35200	0.21
IO	38	100	38.00

Sotto l'assunzione che non si possa ottenere un'architettura migliore dal punto di vista delle performance dell'architettura proposta, allora la massima frequenza di clock è la seguente:

$$f_{\text{clk}} = \frac{1}{T_{\text{clk}} - T_{\text{slack}}} = 103.1MHz$$

Capitolo 6

Conclusioni

L'algoritmo CORDIC è spesso implementato a livello hardware per la sua semplicità dato che il calcolo di funzioni trigonometriche ed iperboliche richiede solo l'uso di sommatori e shifter. Come si è potuto notare dai risultati ottenuti nella fase di testing, questa semplicità implica che i risultati forniti hanno un certo grado di approssimazione, che nell'architettura qui proposta non è troppo alto perciò è possibile ritenere i risultati soddisfacenti.