Politecnico di Torino

Laurea Specialistica in Ingegneria Elettronica

A.A. 2004/2005

Corso di

Sistemi Programmabili per le Telecomunicazioni (*01GSNCI*)

Descrizione in VHDL dell'algoritmo CORDIC per il calcolo di $\sin x$ e $\cos x$, con numero di iterazioni parametrico ed architettura ripiegata

Barbiero	Daniele	118931
Caule	Mauro	118381
Ceroni	Marco	119027
Cistaro	Pierluca	120729

Indice

Capitolo 1

L'algoritmo *CORDIC*

Con questa relazione ci proponiamo di descrivere la realizzazione di un blocco hardware, mediante linguaggio VHDL, che realizzi le funzioni trigonometriche di $\sin\theta$ e $\cos\theta$; il valore in ingresso a tale blocco rappresenta l'angolo θ. Tale dispositivo trova una sua efficiente realizzazione mediante l'algoritmo CORDIC, che utilizza le sole funzioni logiche di *shift* e *add*, affiancate da una *Look-Up Table*.

1.1 Funzioni calcolabili mediante CORDIC

CORDIC è l'acronimo di *CO*ordinate *R*otation *DI*gital *C*omputer ed indica l'algoritmo che si serve di rotazioni vettoriali onde calcolare svariate funzioni trigonometriche. Mediante CORDIC è infatti possibile calcolare, oltre ai già citati seno e coseno, le seguenti funzioni:

- $\arctan\theta$
- $\arcsin\theta$
- $\arccos\theta$
- $\sinh\theta$
- $\cosh\theta$

Da queste funzioni è poi possibile ricavarne altre mediante uguaglianze matematiche, come ad esempio :

$$\tan\theta = \frac{\sin\theta}{\cos\theta}$$

oppure come

$$\tanh\theta = \frac{\sinh\theta}{\cosh\theta}$$

Mediante CORDIC è inoltre possibile ricavare altre funzioni che sfruttano l'operazione vettoriale compiuta dall'algoritmo, tra queste:

- Trasformazione di coordinate da polari a rettangolari
- Trasformazione di coordinate da cartesiane a polari
- Rotazioni vettoriali

A seconda delle necessità, ovvero della funzione da calcolare, il CORDIC può operare in modalità "rotazionale" oppure in modalità "vettoriale". Il nostro scopo è però ricavare un blocco che realizzi le funzioni seno e coseno; ci limiteremo dunque all'utilizzo della modalità "rotazionale" dell'algoritmo.

1.2 Il calcolo di *sin* e *cos*

Innanzitutto le equazioni che costituiscono la modalità "rotazionale" sono le seguenti :

$$x_{i+1} = x_i - y_i \cdot d_i \cdot 2^{-i} \tag{1.1}$$
$$y_{i+1} = y_i + x_i \cdot d_i \cdot 2^{-i} \tag{1.2}$$
$$z_{i+1} = z_i - d_i \cdot \arctan 2^{-i} \tag{1.3}$$

con

$$d_i = -1 \text{ se } z_i < 0 \ , \ +1 \text{ altrimenti}$$

Da queste equazioni iterative si ottengono le seguenti uguaglianze:

$$x_n = A_n[x_0 \cos z_0 - y_0 \sin z_0] \tag{1.4}$$
$$y_n = A_n[y_0 \cos z_0 + x_0 \sin z_0] \tag{1.5}$$
$$z_n = 0$$
$$A_n = \prod_n \sqrt{1 + 2^{-2i}} \tag{1.6}$$

A_n prende il nome di guadagno del CORDIC, o "CORDIC gain", ed è l'accumulo di modulo, dovuto al fatto che ad ogni operazione il vettore che

viene usato per la somma non ha modulo unitario, ma aumenta con il numero di iterazioni fino ad arrivare ad una valore di "saturazione" pari a circa 1.647.

Naturalmente il calcolo dell'arcotangente nella (1.3) non viene svolto dal punto di vista matematico poiché se così fosse l'utilizzo di tale algoritmo non porterebbe ad alcun vantaggio. Perciò abbiamo utilizzato una *Look-Up Table*, in cui sono stati memorizzati i valori dell'arcotangente per tutti i possibili valori della variabile "i", ovvero tanti quanto indicato dal numero di iterazioni "n".

Il calcolo vero e proprio di seno e coseno, avviene impostando il valore iniziale di y_i nelle (1.2) e (1.3) a zero, cosicché si ottiene il coseno dalla (1.4) ed il seno dalla (1.5). Al termine delle iterazioni, ovvero dopo n cicli, si perviene alle seguenti uguaglianze:

$$x_n = A_n \cdot x_0 \ \cos z_0 \tag{1.7}$$

$$y_n = A_n \cdot y_0 \ \sin z_0 \tag{1.8}$$

A tal punto è sufficiente moltiplicare il risultato per $\frac{1}{A_n}$ onde ottenere il risultato di seno e coseno esatti. Così facendo sarebbe però necessario l'utilizzo di un moltiplicatore, andando ad inificiare la bontà dell'algoritmo. La soluzione consiste nell'impostare il valore iniziale di x_0 pari all'inverso del CORDIC gain anziché ad uno.

Il CORDIC in questo modo funziona soltanto per angoli compresi tra $+\pi/2$ e $-\pi/2$; è facile estendere il funzionamento ad angoli esterni a quest'intervallo mediante cambio di segno oppure scambio di seno con coseno, in virtù delle simmetrie matematiche di codeste funzioni.

1.3 L'algoritmo: come funziona

L'algoritmo `CORDIC` è dunque ottimo allo scopo di risparmiare sull'hardware; per quanto riguarda invece la **precisione** dell'algoritmo, essa aumenta di pari passo con il numero di bit utilizzati nel bus interno. Per effettuare un paragone ed implementare velocemente l'algoritmo, è stato utilizzato il programma Matlab (©), mediante il quale abbiamo testato il calcolo di seno e coseno iterativo e ne abbiamo confrontato il risultato con i risultati noti. La prova è stata eseguita con un angolo in ingresso di 30°, il cui seno vale $\frac{1}{2}$ mentre il coseno è pari a $\frac{\sqrt{3}}{2}$.

Il *CORDIC* con Matlab

La funzione che chiama il programma si occupa di passare il valore iniziale, l'angolo (in radianti) ed il numero di iterazioni (i):

```
calc=cordic(0,0.523598775598299,i);
```

Mentre la funzione vera e propria è la seguente :

```
function out=cordic(y,z,n)
temp=1;
for i=0:1:n
    two_at_minus(i+1)=temp;
    temp=temp/2;
end
atan=[7.8539816339744800E-01
       .....
      5.4210108624275200E-20];
kc = [7.0710678118654700E-01
      .....
      6.0725293500888100E-01 ];
 x=kc(n); for k = 0:1:n
    x_temp = x;
    if ( z >= 0.0)
      x = x - y * two_at_minus(k+1);
      y = y + x_temp * two_at_minus(k+1);
      z = z - atan(k+1);
    else
      x = x + y * two_at_minus(k+1);
      y = y - x_temp * two_at_minus(k+1);
      z = z + atan(k+1);
    end
  out=[x y];
end
```

Grazie alle veloci funzioni grafiche di Matlab è stato dunque facile diagrammare i risultati ottenuti, al fine di facilitare la comprensione dell'algoritmo; di seguito si può vedere come esso converga all'aumentare del numero di iterazioni (figure 1.1 e 1.2).

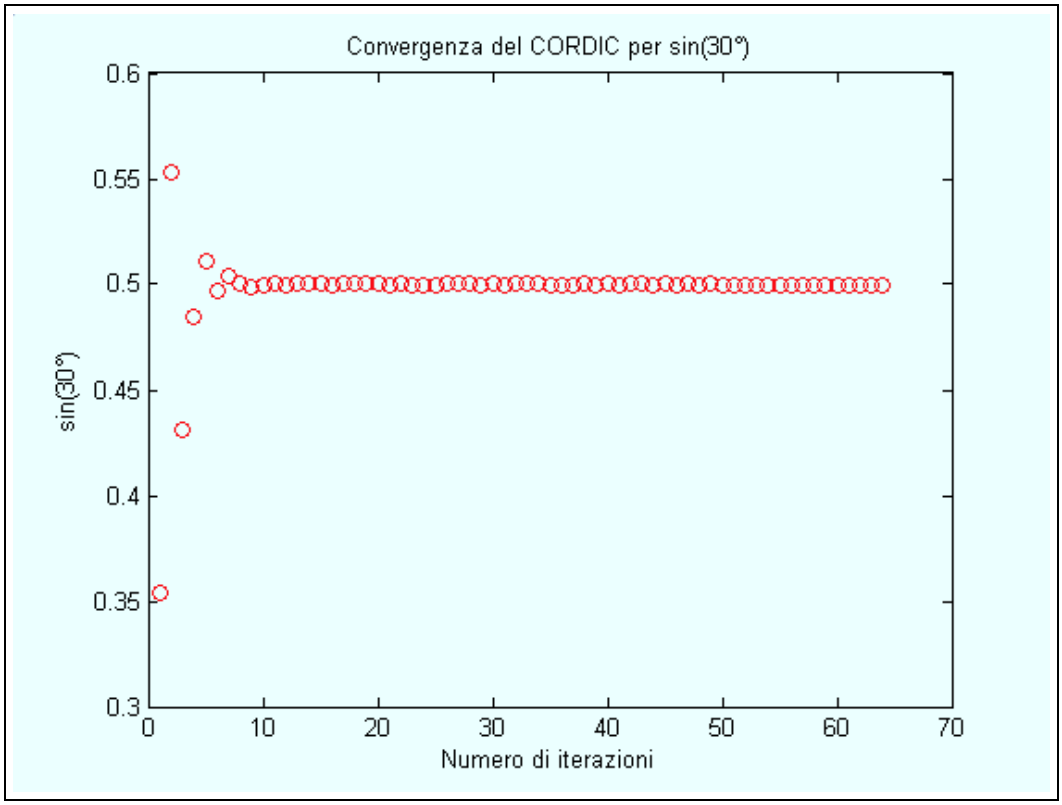

Figura 1.1: Convergenza sin

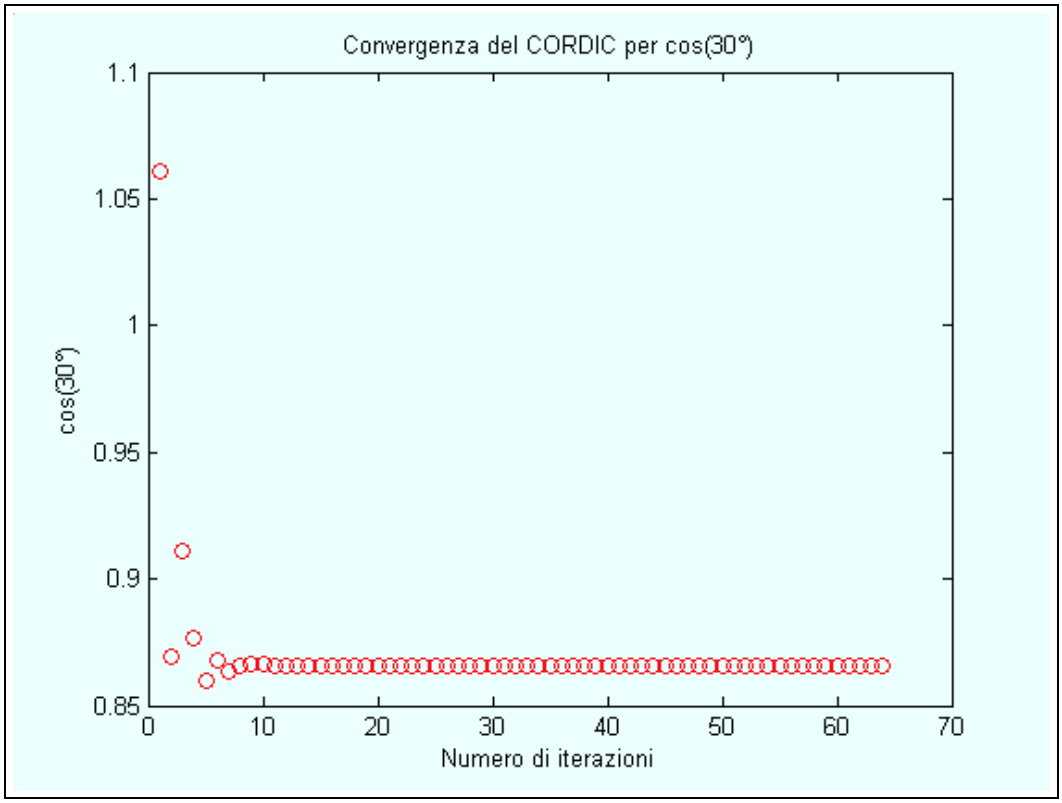

Figura 1.2: Convergenza cos

Invece le figure (1.3) e (1.4) mostrano come l'aggiunta di ogni passo iterativo aumenti la precisione del risultato, salvo poi saturare intorno a 10e-15 che non è causa dell'algoritmo ma del software utilizzato. Per il calcolo del seno si nota inoltre che attorno alla cinquantesima iterazione la curva sparisce; in realtà questi valori corrispondono esattamente al numero esatto, sempre a causa della limitatezza della precisione interna a Matlab.

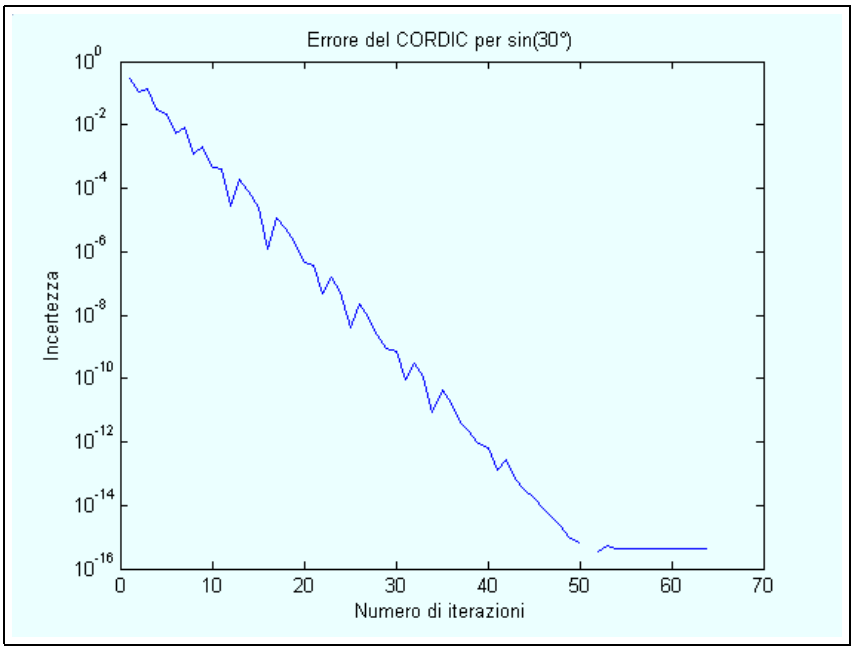

Figura 1.3: Errore sin

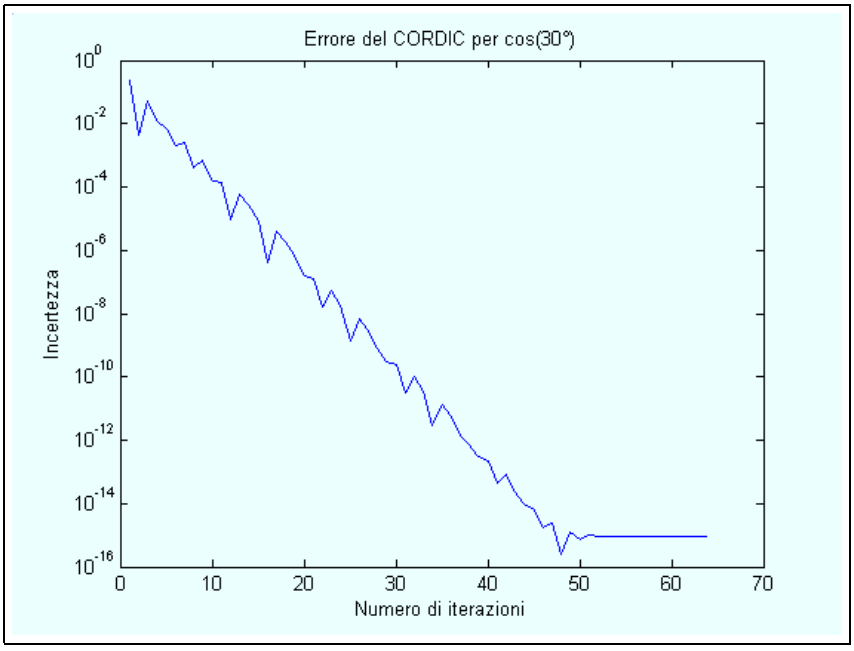

Figura 1.4: Errore cos

Capitolo 2

Diagrammi a blocchi

Una semplice struttura iterativa può essere sintetizzata direttamente a partire dalle equazioni (1.1),(1.2) e (1.3) mediante l'utilizzo di multiplexer, registri, bit-shifters e sommatori come mostrato nelle figure (2.1) e (2.2). In tali diagrammi la funzione di decisione sul segno del sommatore ($\pm d_i$) è ottenuta a partire dal bit più significativo all'interno del registro che contiene z_i, come da [1]. Tale configurazione presuppone dunque l'utilizzo del complemento a 2 nell'interpretazione e gestione dei bit.

Nel diagramma in figura (2.2) la ROM è il dispositivo che contiene i valori dell'arcotangente in funzione del passo iterativo.

Un'architettura di questo tipo è ottenibile direttamente dalle equazioni, come detto, ma implica l'interpretazione dei bit in ingresso come `std_logic_vector`, e dunque con notazioni senza virgola. Ciò è di difficile interpretazione e necessita di una conversione dei risultati ottenuti.

Una versione più facilmente "leggibile" fa uso del tipo `real`, che contempla cifre comprese entro il range 1e-308 $\div$ 1e+308 e si serve di 64 bit suddivisi in segno (1 bit), esponente (11 bit) e mantissa (52 bit) per interpretare il numero. Utilizzando numeri decimali il risultato si ottiene direttamente.

Il nostro gruppo di lavoro ha progettato e simulato perciò due versioni del *CORDIC*, di cui una facente uso del tipo `real` ed una invece "tradizionale", in cui l'interpretazione dei dati non è immediata, ma è sintetizzabile poiché fa uso dei soli tipi `std_logic_vector`.

Le figure (2.1) e (2.2) rappresentano dunque i diagrammi relativi alla prima implementazione.

La figura (2.3) indica come è stato realizzato il *CORDIC* nella seconda maniera.

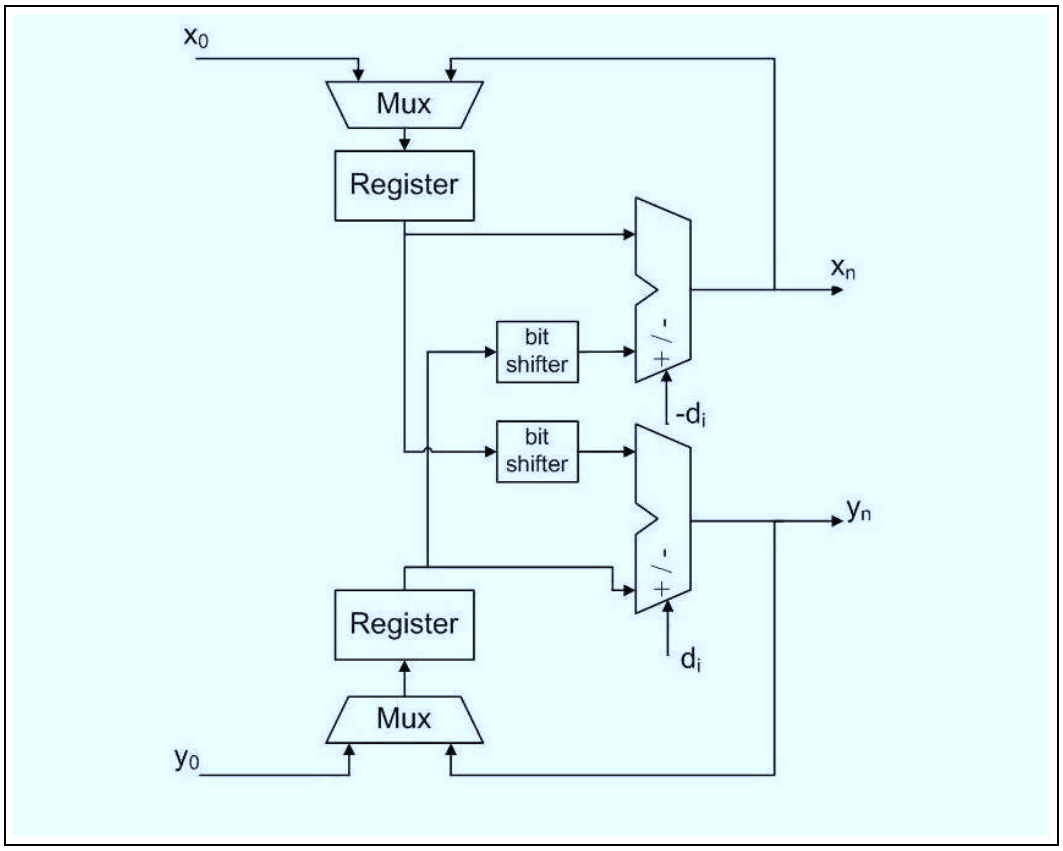

Figura 2.1: Diagramma a blocchi: calcolo di x_n e y_n

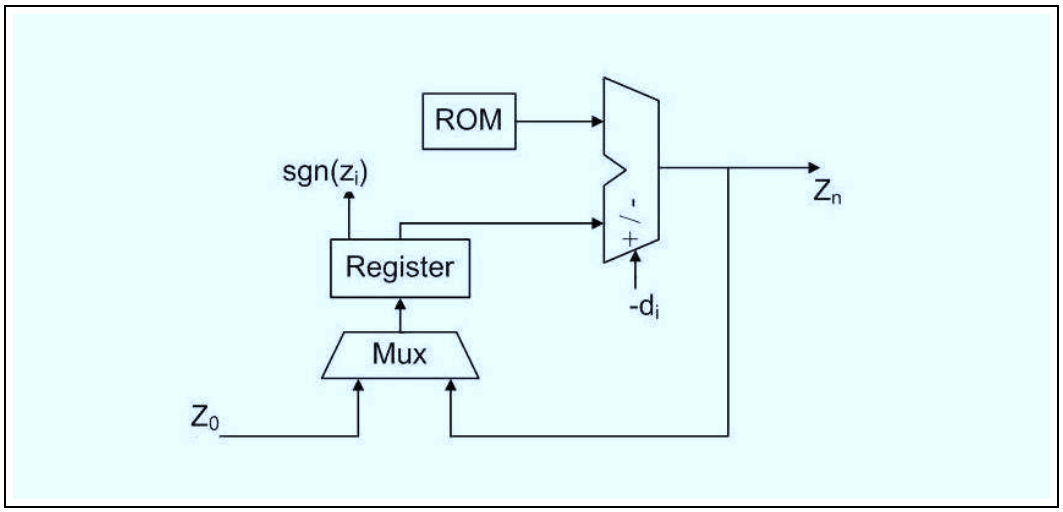

Figura 2.2: Diagramma a blocchi: calcolo di z_n

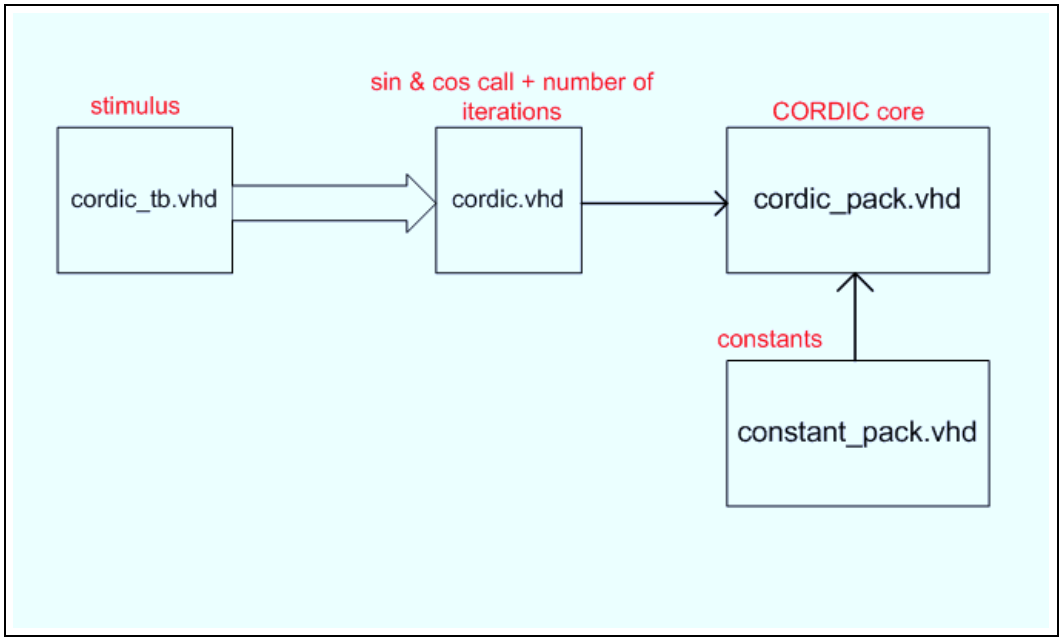

Figura 2.3: Diagramma a blocchi

Capitolo 3

Il *CORDIC* in VHDL

Entrambi i progetti sono stati eseguiti intendendo gli angoli in ingresso espressi in radianti; la conversione rad $\rightarrow$ deg e deg $\rightarrow$ rad coinvolge la semplice moltiplicazione e divisione per il fattore $\frac{\pi}{180^\circ}$, ma come intuibile richiederebbe un componente costoso dal punto di vista hardware e non è stato incluso.

Per convenzione chiameremo "CORDIC std_logic" il progetto sintetizzabile eseguito con il tipo `std_logic`, mentre il progetto realizzato con il tipo `real` verrà definito "CORDIC real".

Come già detto la profonda differenza tra i due sta nell'interpretazione del dato all'ingresso (ovvero l'angolo); infatti la precisione raggiungibile con la rappresentazione in virgola mobile è di gran lunga superiore a quella in virgola fissa.

Inoltre nel secondo caso abbiamo sfruttato la facilità di scrittura delle funzioni per inserire nella *Look-Up Table* i valori del cordic gain in funzione del numero di iterazioni. Abbiamo dunque collocato al suo interno i valori di tale guadagno per ogni passo iterativo. Notando però che da un certo numero di iterazioni (26 per l'esattezza) in poi il risultato non cambia, e come già detto satura, abbiamo risparmiato memoria caricando soltanto i primi 26 valori del guadagno anziché tutti e 64.
Ad essere precisi, non è proprio del tutto vero che essa satura, ma è dovuto al fatto che il calcolo di tali numeri è stato svolto con un numero limitato di cifre decimali (ovvero quelle messe a disposizione da Microsoft Excel ©), perciò si ha l'impressione che saturi. Con un sistema di calcolo più preciso ciò non avviene.

Cominciamo con il vedere l'algoritmo comprensibile "alla macchina"; di seguito descriveremo quello che non necessita di manipolazioni sui dati per essere comprensibili a noi.

3.1 Realizzazione tipo 1: std_logic

Innanzitutto la rappresentazione dei dati avviene su 32 bit, anche se è riconfigurabile mediante la key-word *"width"*. Di questi 32 bit i primi 2 vengono riservati per la parte intera mentre i rimanenti 30 esprimono la parte decimale. La notazione è in complemento a 2 ed è per questo che sono necessari 2 bit per la parte intera. Naturalmente il sistema funziona senza la virgola, mentre vanno riscalati di 30 posizioni i valori in ingresso ed in uscita per essere comprensibili a noi.

Il problema principale da noi incontrato per la realizzazione di questo componente è stato lo shifter, poiché non è agevole implementare un divisore modulo 2 che esegua lo shift velocemente. L'architettura che prevede il numero di iterazioni parametrico (come da specifica) ci impone l'utilizzo di uno shifter veloce, che presenti i valori per ogni passo iterativo; la nostra soluzione è stato dunque includere un clock separato rispetto al blocco principale, con frequenza 100 volte superiore per poter effettuare tali shift nel tempo stabilito.

Il CORDIC presenta l'inconveniente di non permettere la piena realizzazione parametrica del componente, e questo a causa della *Look-Up Table* in quanto i valori in essa caricati debbono essere almeno pari al numero di iterazioni, che corrispondono al numero di bit. In pratica il numero di arcotangenti caricate nella *Look-Up* è il limite superiore al numero di iterazioni.

Il progetto che fa uso del tipo std_logic per la gestione dei dati si compone di 6 files :

atan32.vhd

E' la *Look-Up Table* che contiene i valori dell'arcotangente; per comodità sono stati scritti in notazione esadecimale. Restituisce il valore di arcotangente (`ZData`, std_logic_vector) in base al puntatore che le viene passato in ingresso (`ZA`, integer):

```
process(ZA)
begin
  case ZA is
     when 0 => ZData <= X"3243f6a8";
     when 1 => ZData <= X"1dac6705";
     ...
  end case;
end process;
```

addsub.vhd

Un sommatore che si occupa semplicemente di sommare o sottrarre i valori in ingresso. Il bit che decide il segno (`as`, std_logic) viene usato per pilotare l'uscita (`obus`, std_logic_vector) in base agli ingressi (`abus` e `bbus`, std_logic_vector) nel seguente modo:

```
process(as,abus,bbus)
begin
  if as='1' then
    obus <= abus + bbus;
  else
    obus <= abus - bbus;
  end if;
end process;
```

fsm.vhd

La macchina a stati finiti gestisce i bit di controllo, che sono `init`, `load` e `done` (tutti std_logic). In ingresso vi sono naturalmente `rst` e `clk` (std_logic); `start` (std_logic) e `cnt` (integer) invece sono i segnali di partenza e di conteggio. La parte di codice che pilota le uscite è la seguente :

```
case state is
   when s0 =>done <= '0'; init <= '1'; load <= '0';
   when s1 =>done <= '0'; init <= '0'; load <= '1';
   when s2 =>done <= '1'; init <= '0'; load <= '0';
```

shiftn.vhd

Il bit shifter. Oltre a `clk_sh` e `rst` (std_logic) in ingresso contiene `ibus` (std_logic_vector) e `n` (integer). Esso costruisce un vettore di registri che contengono tanti shift quanti indicati dalla posizione di tale vettore (ad es. la posizione 3 contiene il dato in ingresso shiftato di 3 posizioni a destra); questo è il motivo per cui necessita di un clock più veloce. In uscita presenta la posizione a cui punta il passo iterativo (obus $\Leftarrow$ shifts(n)), che è il valore 'n' al suo ingresso. Ecco il codice interno al process :

```
shifts(0) <= ibus;
for i in 0 to width-2 loop
  shifts(i+1)(width-1)<= shifts(i)(width-1);
  shifts(i+1)(width-2)<= shifts(i)(width-1);
```

```
    for j in 0 to (width-2) loop
      shifts(i+1)(j) <= shifts(i)(j+1);
    end loop;
  end loop;
  obus <= shifts(n);
```

In pratica lo shift a destra, poiché tratta numeri in complemento a 2, deve ricopiare il bit più significativo nella posizione di peso più elevato ed in quella adiacente, per poi spostare i rimanenti bit verso destra di una posizione.

cordic.vhd

Cordic è il componente in cui sono mappati i files atan32, addsub, fsm e shiftn, la cui struttura segue il modello schematizzato nelle figure (2.1) e (2.2). Esso si occupa di "smistare" i compiti ai vari sottoblocchi, calcolando i valori di x,y e z iterando nel seguente modo (calcolo di x):

```
process (clk,newx_s)
 begin
   if (clk'event and clk='1') then
     if init_s='1' then
       xreg_s(WIDTH-1 downto 0) <= xinit_c(WIDTH-1 downto 0);
     elsif load_s='1' then
       xreg_s <= newx_s;
     end if;
   end if;
 end process;
```

Infine incrementa il contatore di iterazione ad ogni colpo di clock.

cordic_tb.vhd

Il test bench si occupa di generare i 2 clock, resetta il sistema e procura gli ingressi `angle` e `start`. Gli angoli utilizzare per testare la struttura sono 0°, 8°, 87°, -87° e 50° però espressi in radianti e convertiti in esadecimale. Un esempio del codice relativo allo stimolo di 8° è il seguente :

```
angle_s <= X"08efa351"; -- 8 deg
start_s <= '1';
wait for 300 ns;
start_s <= '0';
wait until done_s='1';
```

3.2 Realizzazione tipo 2: real

In figura (2.3) è schematizzata brevemente l'architettura globale del sistema; il linguaggio sintattico ricorda molto il "C", con le chiamate alle funzioni ed il passaggio di variabili. Eccone un esempio:

```
function SIN (X:real ; num_of_iters:NATURAL) return real;
```

In questo modo la funzione seno riceve due parametri, tra cui l'angolo (X) e restituisce un valore `real`. Le funzioni sin e cos vengono svolte mediante CORDIC all'interno del package, ma la chiamata alle funzioni non ne "vede" assolutamente lo svolgimento, che potrebbe essere fatto in qualunque altro modo.

In tale realizzazione il numero di iterazioni parametrico di per sè non è ottenibile, visto che il `real` utilizza 64 bit, perciò abbiamo inserito il parametro `num_of_iters` nella chiamata alla funzione, il quale può assumere qualunque valore tra 1 e 64.

I files utilizzati sono 4:

constant_pack.vhd

Il file **constant_pack.vhd** contiene le costanti, ovvero i valori di π (in `real`), $\frac{\pi}{2}$ (serve per stabilire il quadrante in cui si trova l'angolo), il numero max di iterazioni (`integer`), i valori dell'arcotangente in funzione del passo iterativo e del "cordic gain" (`real`).

cordic_pack.vhd

Esso contiene l'algoritmo vero e proprio, in cui vengono calcolati

- i valori di 2^{-i} (equazioni 1.1 e 1.2) ;
- il valore del cordic gain in funzione delle iterazioni (equazione 1.6);
- il quadrante in cui si trova l'angolo in ingresso (il cordic di per sè funziona solo da -π/2 a +π/2);
- le rotazioni del cordic.

Sono perciò contenute le seguenti funzioni :

```
function POWER_OF_2 (initial_value:REAL;number_of_values:
NATURAL) return REAL_VECTOR
```

Calcola il vettore delle potenze inverse di 2, ove `number_of_values` è il numero di iterazioni.

```
function CORDIC (x0:REAL;y0:REAL;z0:REAL;n:NATURAL)
return REAL_ARR_2
```

Restituisce un vettore (`REAL_ARR_2`) contenente x_n, y_n e y_n, con n pari al numero di iterazioni.

```
function CordicGain (num_iter:natural) return real
```

Seleziona il valore di cordic gain in funzione del numero di iterazioni.

```
function SIN (x:real;num_of_iters:natural) return real
```

Restituisce il seno di x dopo `num_of_iters` iterazioni.

```
function COS (x:real;num_of_iters:natural) return real
```

Restituisce il coseno di x dopo `num_of_iters` iterazioni.

cordic_call.vhd

Cordic_call è il file che si occupa di passare l'angolo ed il numero di iterazioni volute a cordic_pack.vhd. Al suo ingresso è dunque presente una variabile ('a') di tipo `real` che è l'angolo; in uscita 'b' e 'c' sono rispettivamente il seno ed il coseno.

cordic_call_tb.vhd

Il test bench stimola gli ingressi, fornendo valori a partire da 0 e incrementati di 15 gradi, corrispondenti a circa 0.26179 radianti :

```
a_i <= con;
con:=con+0.261799387799149;   -- adds 15 degs
wait for 10 ns;
```

Capitolo 4

Simulazioni

4.1 Simulazione CORDIC tipo 1: std_logic

4.1.1 Il bit shifter

Prima di esaminare il CORDIC di tipo 1 nella sua completezza, presentiamo la simulazione ricavata dal bit shifter; come già menzionato in cap. 3, lo shifter deve presentare in uscita il valore dell'ingresso shiftato di tante posizione quante definite dal puntatore al passo iterativo. Ciò corrisponde naturalmente ad effettuare l'operazione 2^{-i}, che in binario risulta molto agevole. La figura (4.1)[1] mostra come venga costruito il vettore degli shift; ad ogni colpo di clock viene creata una nuova posizione, mentre l'uscita ('b') è funzione di 'sel'.

[1] I grafici sono stati ruotati per facilitarne l'inserimento nel presente documento

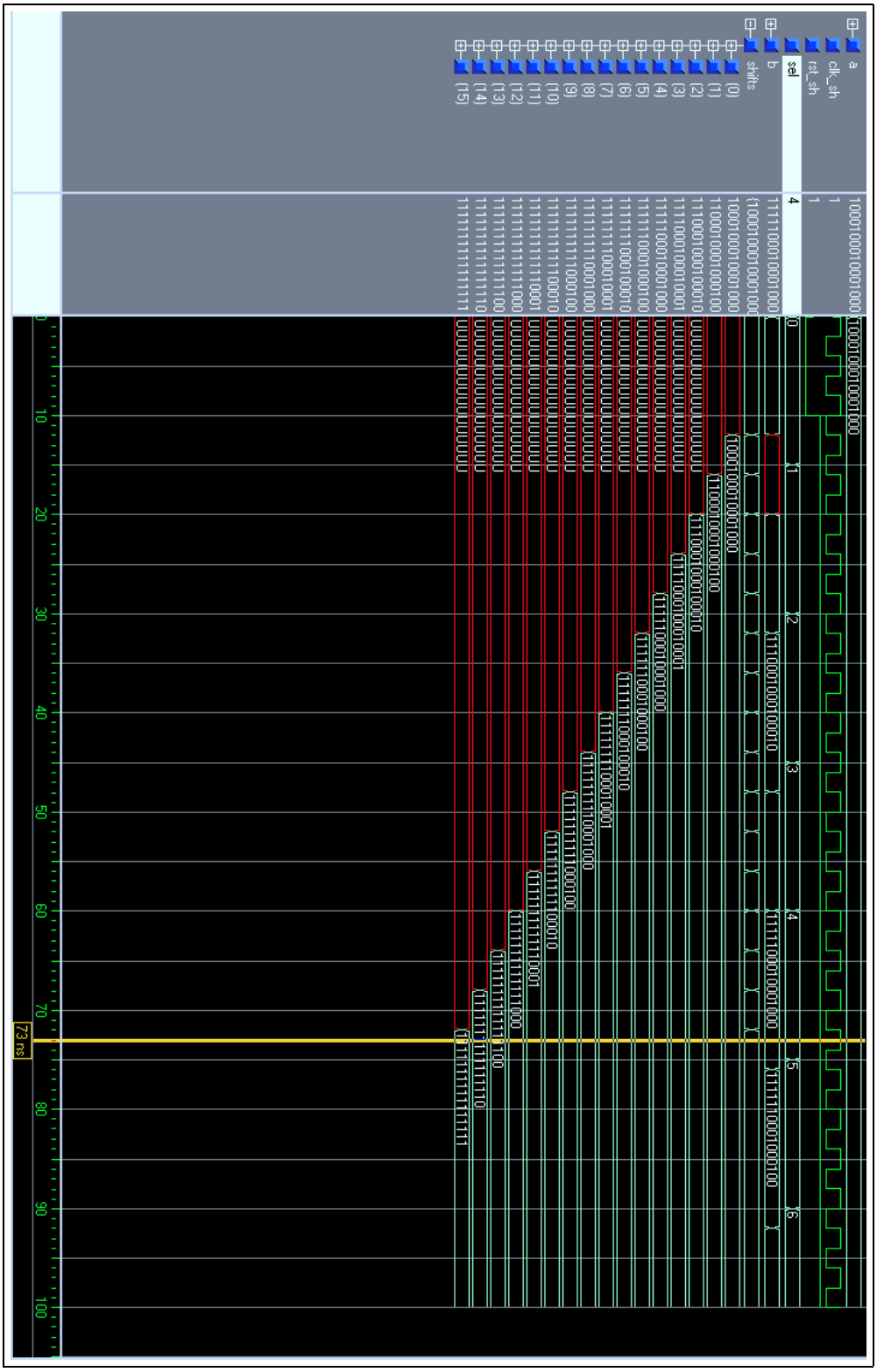

Figura 4.1: Simulazione del bit shifter

4.1.2 Il CORDIC completo

La simulazione completa del CORDIC di tipo 1, ovvero comprendente il tipo "std_logic", ha dato come risultato i grafici delle figure (4.2), (4.3) e (4.4). Da queste rappresentazioni si evince la correttezza del funzionamento poiché in risposta ad un angolo in ingresso di 0°, il seno vale "FFFFFFFF", ovvero il più piccolo numero negativo, che possiamo considerare zero. Il coseno invece vale 3FFFFFC8, ovvero, incertezza a parte, un numero che è circa uguale ad uno.

La figura (4.3) è uno zoom "temporale" della precedente, mentre la figura (4.4) mostra il funzionamento dell'algoritmo per un altro valore dell'angolo in ingresso. In tal caso esso vale 8° ed il risultato del CORDIC dà 0.1391730932 a fronte del valore esatto 0.1391731009. Lo scostamento dal valor vero è 5.6e-8. Il coseno invece risulta 0.9902680050, mentre il risultato esatto è 0.9902680687, con un errore di circa 6.4e-8.

Una piccola miglioria al componente potrebbe essere resettare i registri quando viene passato un angolo nuovo, ovvero dopo che `done` è andato ad uno. Ciò al fine di evitare numerosi warning da parte del simulatore. Abbiamo però notato che il circuito funziona ugualmente anche se non si inizializzano i registri ad ogni cambio del dato d'ingresso.

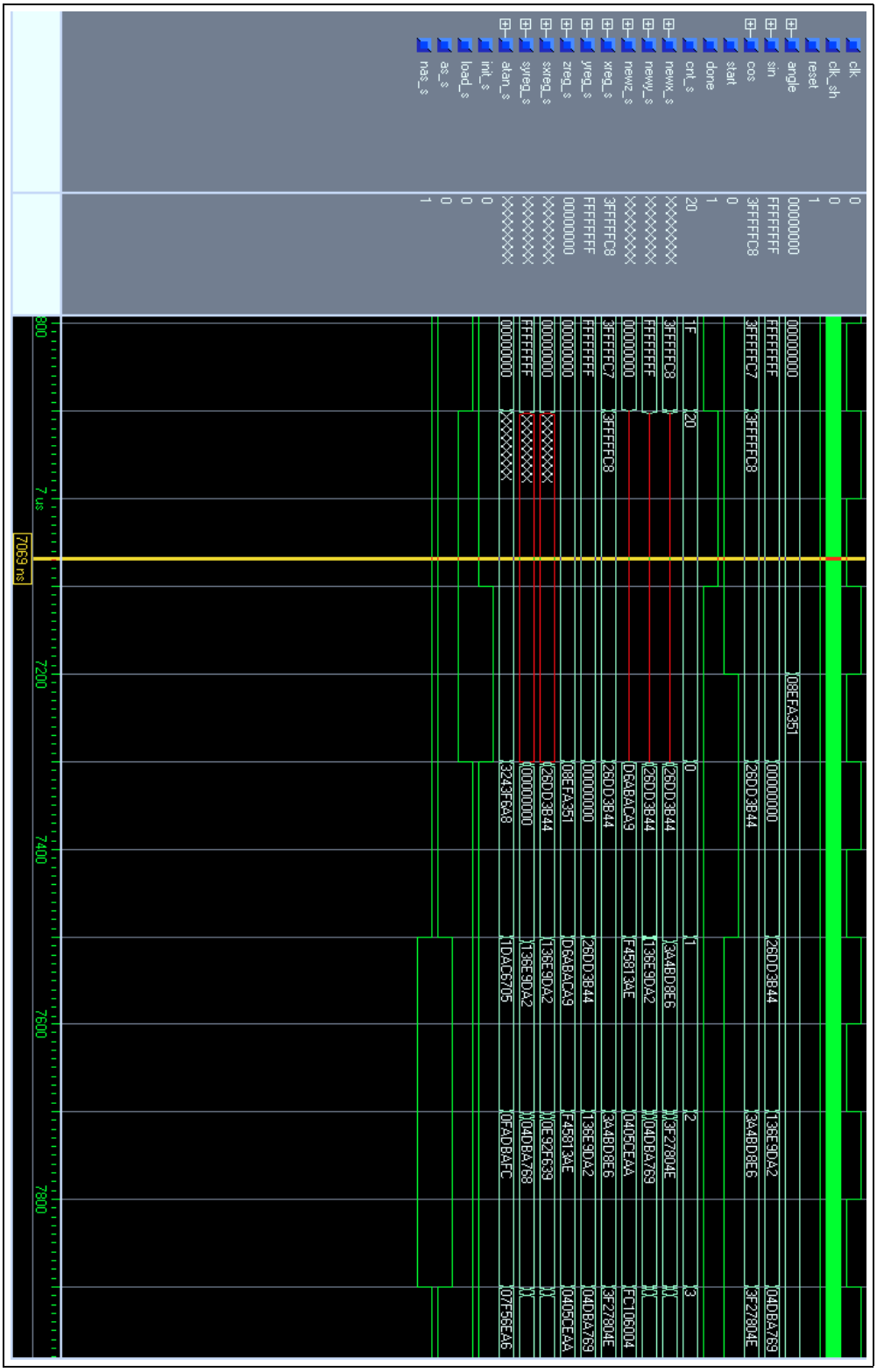

Figura 4.2: Simulazione del test bench di tipo 1

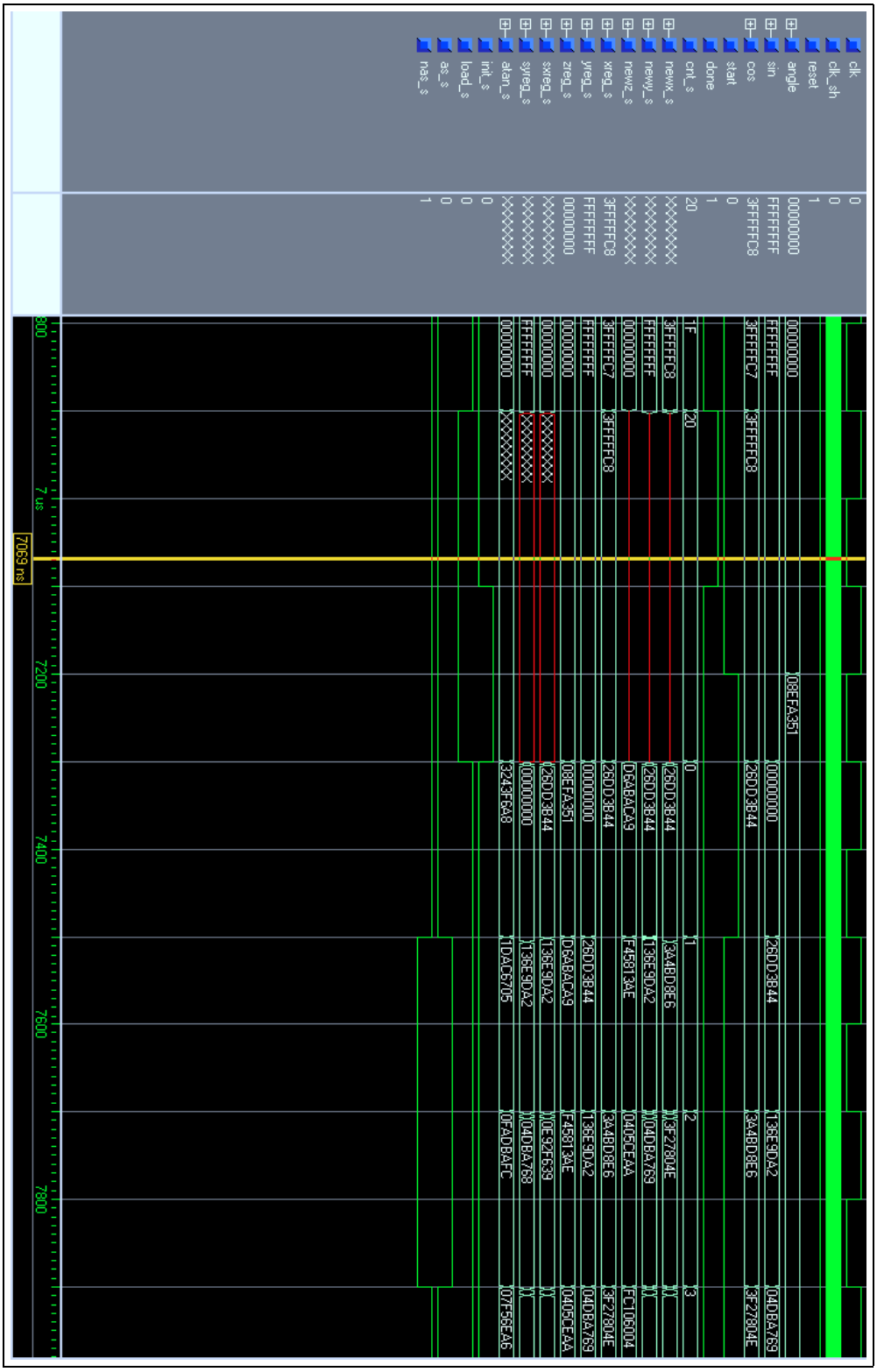

Figura 4.3: Ingrandimento della simulazione di tipo 1

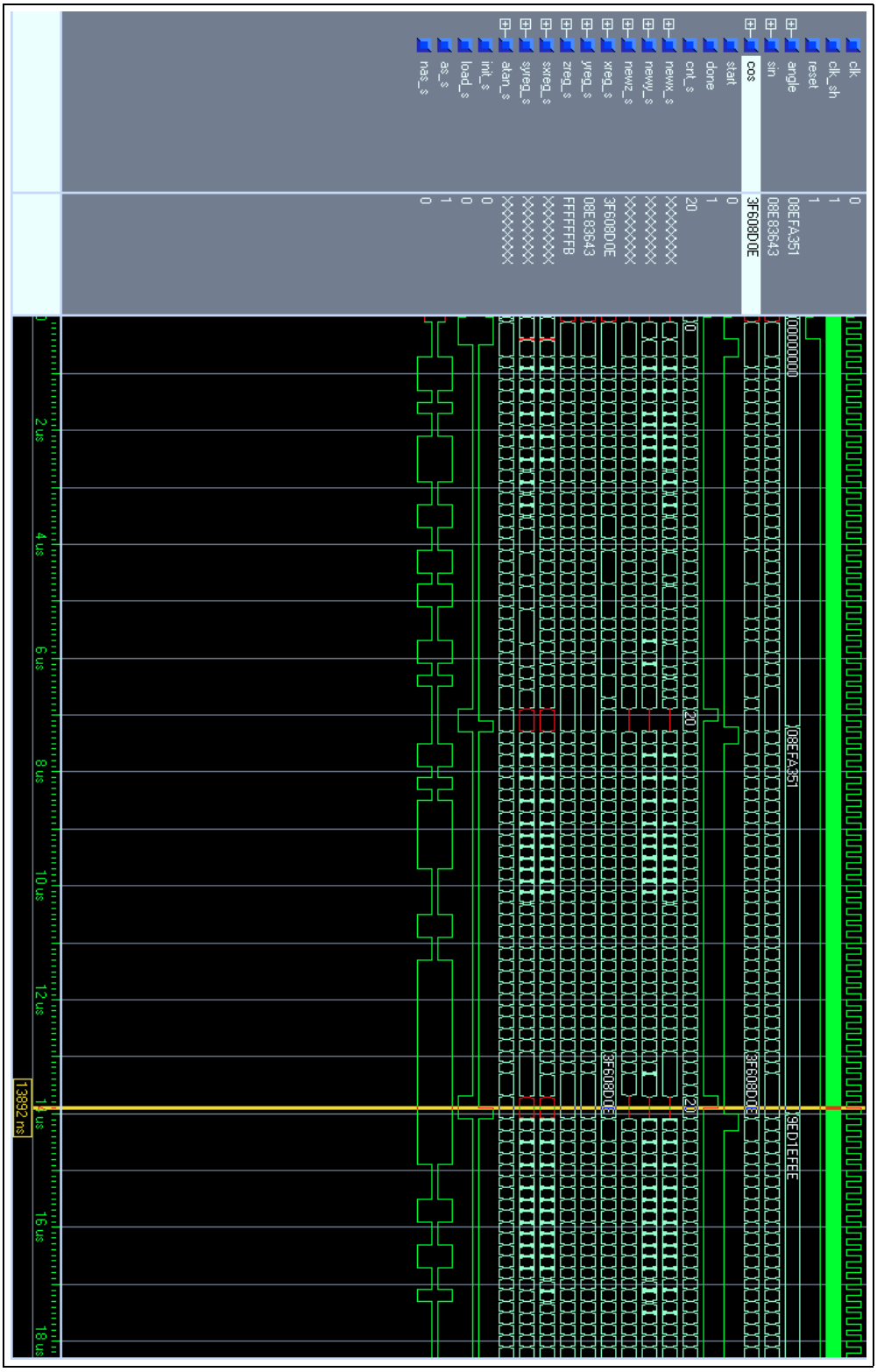

Figura 4.4: Estensione temporale della simulazione di tipo 1

4.2 Simulazione CORDIC tipo 2: real

L'algoritmo che fa uso del costrutto real di per sè non utilizza il clock e non ha ritardi; la risposta allo stimolo è stata ritardata di 5 ns utilizzando il comando `wait`.

In queste simulazioni è immediata la lettura sia del dato all'ingresso, che ricordiamo essere espresso in radianti, sia dei dati che rappresentano seno e coseno di tale dato.

Nella figura (4.5) si nota che fino a quando non vengono inizializzate, le variabili real assumono il massimo valor negativo, ovvero -1e308. Quando viene posto all'ingresso un angolo di zero radianti il coseno è correttamente pari ad uno, mentre il seno è un numero molto prossimo a zero, ma non è zero, bensì circa -4e-16. Ciò è dovuto al funzionamento interno del CORDIC, che fa uso di sottrazioni successive tra valori calcolati iterativamente e arcotangenti o potenze inverse di 2; la precisione di tali valori è naturalmente finita ed è ciò che influenza la precisione del valore calcolato. Il valore di 4e-16 è infatti la stessa incertezza massima che si ricava dai grafici di figg (1.3) e (1.4).

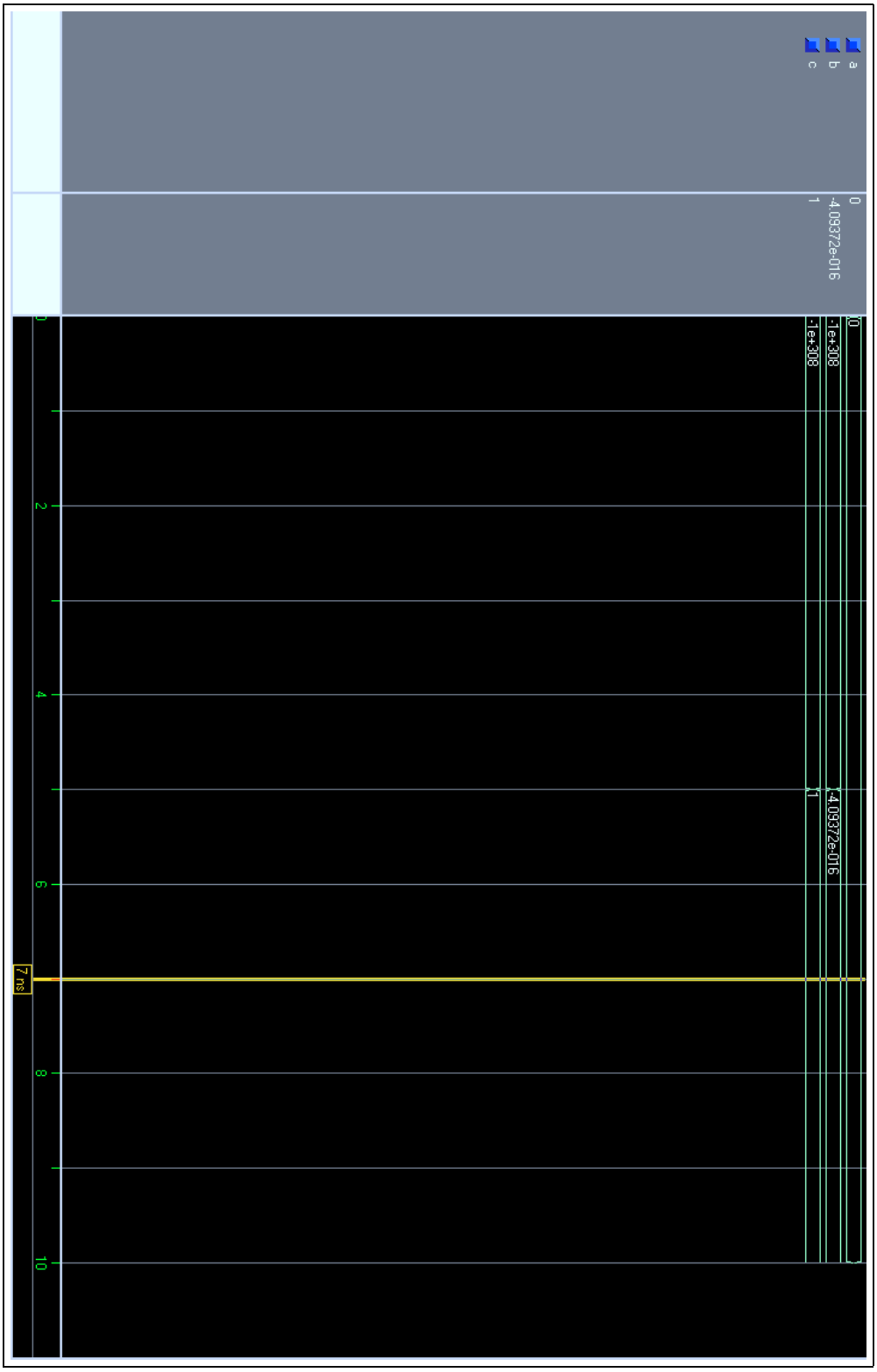

Figura 4.5: Simulazione del test bench di tipo 2

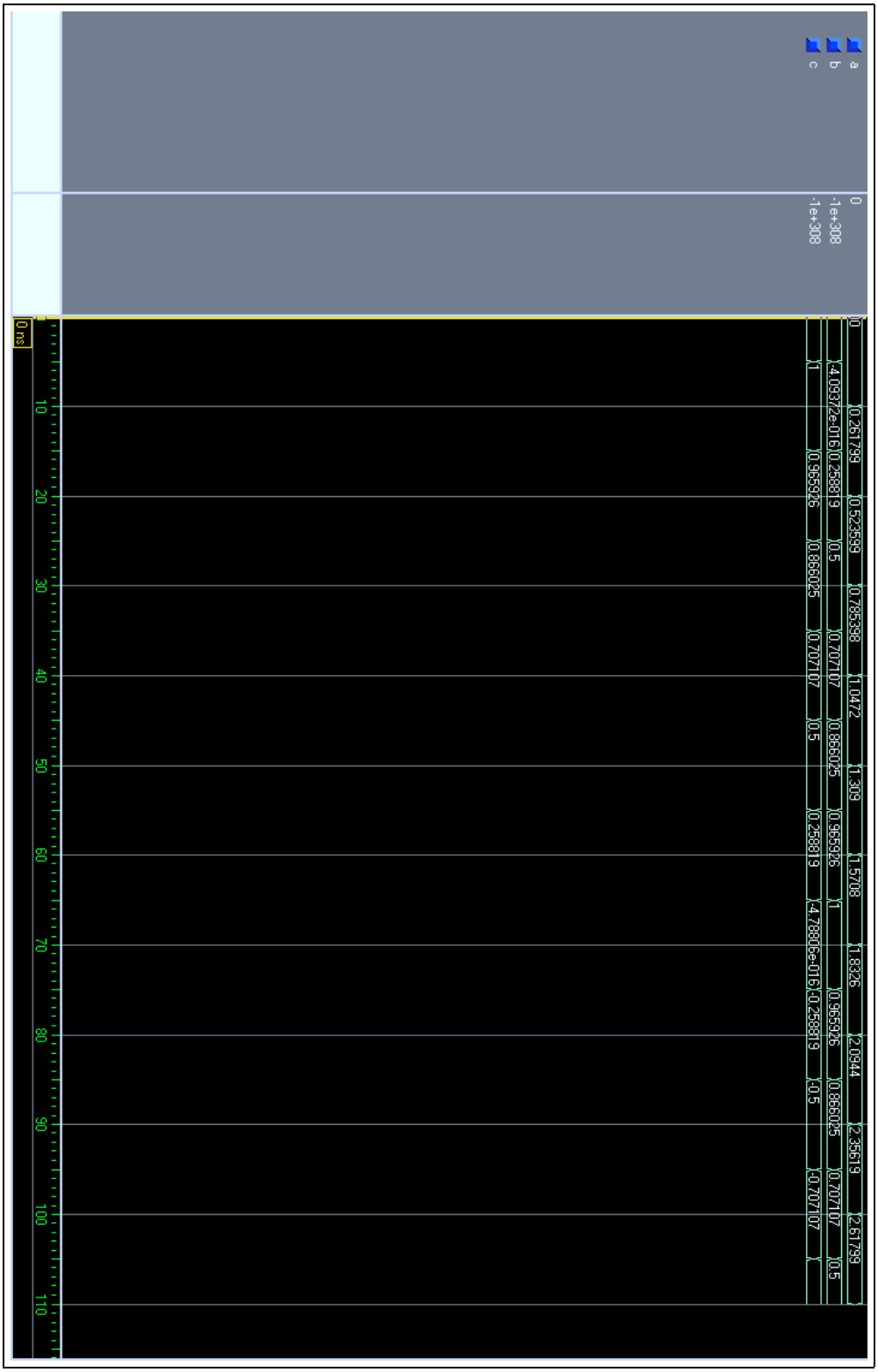

Figura 4.6: Estensione temporale della simulazione di tipo 2

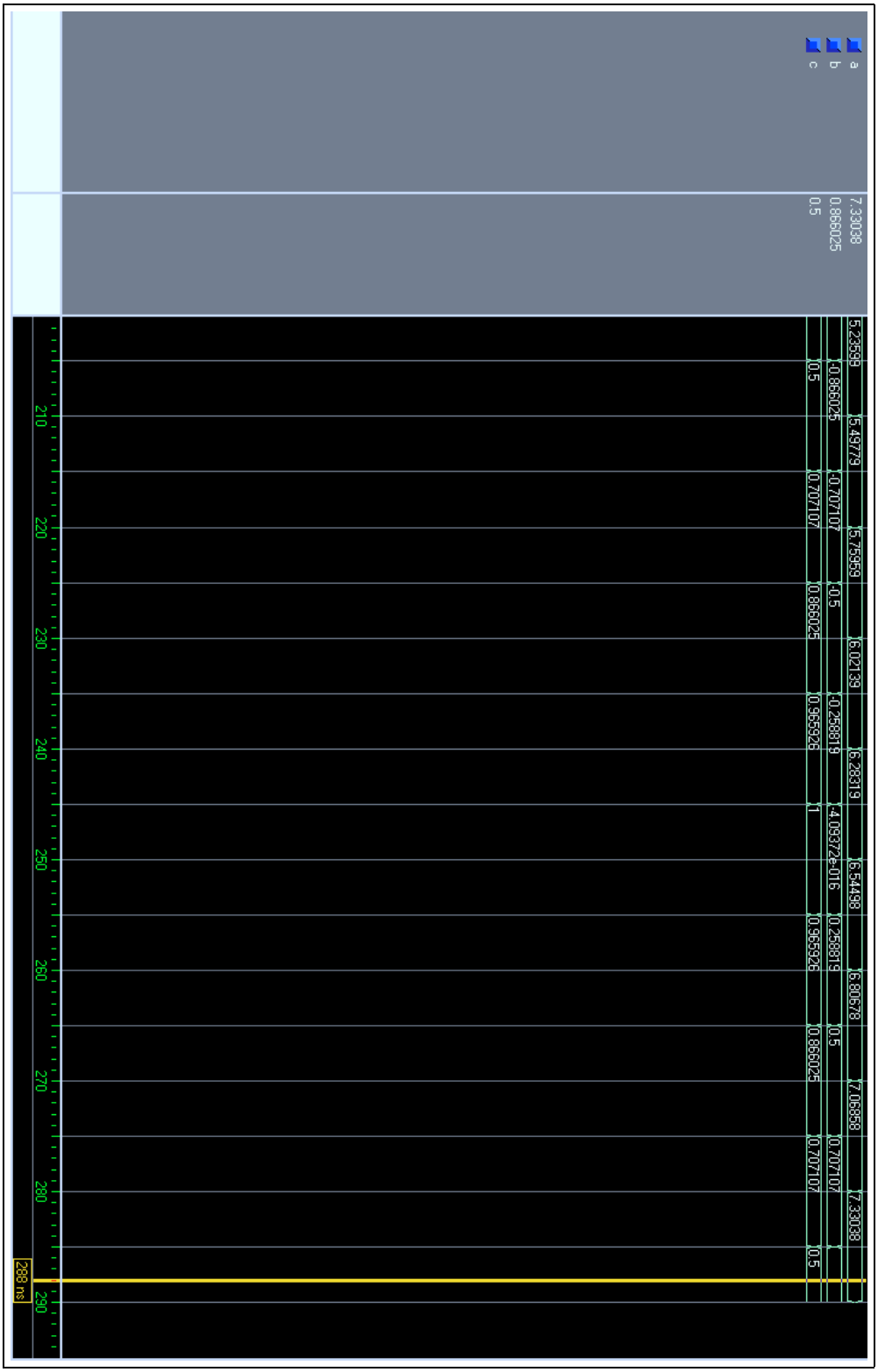

Figura 4.7: Estensione temporale della simulazione di tipo 2 a valori maggiori di 2π

Capitolo 5

Conclusioni

La realizzazione del CORDIC da parte del nostro gruppo è iniziata con il costrutto che fa uso del tipo `real`, poiché di più semplice ed immediata realizzazione. La sintetizzazione di tali files però ci è parsa fin da subito piuttosto dubbia, in quanto la gestione dei dati espressi in floating point non è immediata e difficilmente integrabile in un blocco che vuol implementare un algoritmo di per sè semplice ed efficiente.

Qualche difficoltà, come menzionato, l'ha creata il bit shifter nel CORDIC tipo std_logic_vector (shiftn.vhd), poiché le realizzazioni "guida" tratte dal web implementavano tale componente presupponendo noto il numero di bit, mentre il nostro gruppo ha optato per la realizzazione in parallelismo riconfigurabile anche di questo componente.

Nel complesso sarebbe davvero comodo poter realizzare circuiti con un linguaggio che somiglia molto al C, e che utilizza dei numeri in virgola mobile senza bisogno di complicazioni dovute all'interpretazione dei dati, ma siamo coscenti del fatto che quando scriviamo righe di codice del tipo:

```
for i in 0 to number_of_values loop
  v(i) := temp;
  temp := temp/2.0;
end loop;
```

il sintetizzatore difficilmente intenderà quello che intendiamo noi e dunque non implementerà un bit shifter.

Abbiamo dunque scelto di includere la versione numero 2 del CORDIC più per documentare il lavoro svolto che non per "funzionalità" del codice.

Bibliografia

[1] R.Andraka, *A survey for CORDIC algorithms for FPGA based computers*
FPGA 98 Monterey CA USA, 1998

[2] URL: http://tech-www.informatik.uni-hamburg.de/vhdl/packages/

[3] URL: http://www.dspguru.com/info/faqs/cordic.htm

[4] URL: http://my.execpc.com/∼embed/cordic.htm

[5] URL: http://www.ht-lab.com/freeutils/vhdlsort/vhdlsort.html

[6] M.Baudoin, *Impara* LaTeX! *(..e mettilo da parte)*
École Nationale Supérieure de Techniques Avancées, 1998

[7] T.O.Hubert I.Hyna E.Schlegl, *Una (mica tanto) breve introduzione a* LaTeX 2_ε
Free Software Foundation, 1999