



# POLITECNICO DI BARI

DIPARTIMENTO DI INGEGNERIA ELETTRICA E DELL'INFORMAZIONE

CORSO DI LAUREA IN  
INGEGNERIA INFORMATICA E DELL'AUTOMAZIONE

---

TESI DI LAUREA IN  
FONDAMENTI DI ELETTRONICA

IMPLEMENTAZIONE EFFICIENTE DELL'ALGORITMO  
CORDIC SU FPGA A SUPPORTO DELLA STIMA  
DELL'ANGOLO DI ARRIVO

RELATORE:  
PROF. ANTONELLO FLORIO

LAUREANDO:  
ANGELO MASTRANGELO

---

ANNO ACCADEMICO 2024-2025



# INDICE

<b>INTRODUZIONE E SCOPO DELLA TESI</b>	<b>1</b>
<b>1 LOCALIZZAZIONE BASATA SULLA STIMA DELL'ANGOLO DI ARRIVO</b>	<b>5</b>
1.1 Stima dell'Angolo di Arrivo: Fondamenti e Applicazioni . . . . .	6
1.2 Dalla Stima dell'AoA alla Localizzazione . . . . .	8
1.3 Panoramica sugli FPGA . . . . .	12
1.3.1 Struttura interna di un FPGA . . . . .	13
1.3.2 Vantaggi e limitazioni degli FPGA . . . . .	15
<b>2 TEORIA E IMPLEMENTAZIONE DELL'ALGORITMO CORDIC</b>	<b>17</b>
2.1 Principi Matematici . . . . .	18
2.2 Modalità di Funzionamento . . . . .	19
2.3 Versatilità dell'Algoritmo CORDIC . . . . .	20
2.4 Vantaggi Computazionali nel Design Hardware . . . . .	21
2.5 Sfide e Limitazioni del CORDIC . . . . .	22
2.6 Funzioni Trigonometriche in MATLAB . . . . .	22
2.7 Implementazione dell'Algoritmo in MATLAB . . . . .	24
2.8 Che cos'è la Rappresentazione in Fixed-Point . . . . .	34
2.9 Processo di Conversione in Fixed-Point . . . . .	36
2.10 Scelta della Configurazione Ottimale . . . . .	37
<b>3 IMPLEMENTAZIONE DELL'ALGORITMO CORDIC IN VHDL</b>	<b>41</b>
3.1 MATLAB HDL Coder . . . . .	41
3.1.1 Struttura di HDL Coder . . . . .	43
3.1.2 Vantaggi e Criticità . . . . .	44
3.2 HDL Verifier . . . . .	46
3.2.1 Funzionalità . . . . .	46
3.2.2 Struttura di HDL Verifier . . . . .	47
3.3 HDL Workflow Advisor . . . . .	48
3.4 FPGA-in-the-Loop . . . . .	50
3.4.1 Processo di Funzionamento . . . . .	50
3.4.2 Struttura di FPGA-in-the-Loop . . . . .	51

<b>4 IMPLEMENTAZIONE E SIMULAZIONE DEL MODULO CORDIC IN VHDL</b>	<b>53</b>
4.1 Introduzione al VHDL . . . . .	53
4.1.1 Il Workflow del VHDL . . . . .	53
4.2 Ambienti di Sviluppo: ModelSim e Quartus Prime . . . . .	54
4.3 Il modulo CORDIC e il testbench . . . . .	57
4.4 Risultati di Sintesi con Quartus Prime . . . . .	58
4.5 Analisi dell'Implementazione Hardware . . . . .	65
4.6 Analisi delle Waveform su Modelsim . . . . .	69
<b>5 VERIFICA DEL FUNZIONAMENTO SU PIATTAFORMA FPGA</b>	<b>73</b>
5.1 Analisi e Validazione del Modulo CORDIC tramite Simulink . . . . .	74
5.2 Procedura di Test su MATLAB e Caricamento su FPGA . . . . .	75
5.3 Modelli Simulink per le Funzioni Trigonometriche CORDIC . . . . .	76
5.4 Test Condotti . . . . .	78
5.5 Analisi dell'accuratezza . . . . .	80
5.6 Analisi Comparativa . . . . .	83
5.7 Test di Simulazione per la Stima dell'Angolo di Arrivo . . . . .	85
5.7.1 Architettura del Modello Simulink . . . . .	86
5.7.2 Verifica di Coerenza tra Input e Output . . . . .	87
<b>CONCLUSIONI</b>	<b>89</b>
<b>BIBLIOGRAFIA</b>	<b>I</b>



# INTRODUZIONE E SCOPO DELLA TESI

L’elaborazione dei segnali rappresenta un pilastro fondamentale delle moderne tecnologie di comunicazione, con applicazioni che spaziano dalle telecomunicazioni ai sistemi radar, dalla navigazione alla localizzazione in ambienti complessi. In tale contesto, la stima dell’angolo di arrivo si configura come una tecnica essenziale per determinare la direzione di provenienza di un segnale elettromagnetico rispetto a un array di sensori. Questa capacità risulta cruciale in numerosi ambiti, tra cui le reti wireless di nuova generazione, i sistemi di sorveglianza e le tecnologie di posizionamento indoor, dove la precisione e l’affidabilità delle informazioni angolari possono determinare l’efficacia dell’intero sistema.

La crescente domanda di soluzioni in grado di operare in tempo reale ha spinto la ricerca verso l’adozione di dispositivi hardware altamente performanti, come i Field-Programmable Gate Array (FPGA), capaci di combinare flessibilità e velocità di elaborazione. Gli FPGA si distinguono per la loro architettura programmabile, che consente di implementare algoritmi complessi direttamente a livello hardware, riducendo la latenza e migliorando l’efficienza energetica rispetto alle soluzioni basate su microprocessori tradizionali. La loro capacità di parallelizzare le operazioni li rende particolarmente adatti al processamento dei segnali, dove operazioni ripetitive e computazionalmente intensive, come il calcolo di funzioni trigonometriche, sono spesso richieste. Tuttavia, l’implementazione di algoritmi di stima dell’angolo di arrivo (AoA) su FPGA presenta diverse sfide, tra cui la complessità computazionale degli algoritmi tradizionali, la gestione delle risorse hardware limitate e la necessità di ottimizzare le prestazioni senza compromettere la precisione dei risultati.

Tra gli algoritmi utilizzati per la stima dell’AoA, tecniche come Multiple Signal Classification (MUSIC) ed Estimation of Signal Parameters via Rotational Invariance Techniques (ESPRIT) sono ampiamente adottate per la loro capacità di fornire stime accurate anche in presenza di rumore e interferenze. Questi metodi, tuttavia, richiedono il calcolo intensivo di funzioni trigonometriche, come seno e coseno, per determinare gli angoli di incidenza dei segnali. Tale operazione, se implementata in modo tradizionale, può risultare particolarmente dispendiosa in termini di risorse e tempo di calcolo, specialmente su piattaforme hardware come gli FPGA, dove l’efficienza è un requisito fondamentale. In questo contesto, l’algoritmo Coordinate Rotation Digital Computer (CORDIC) emerge come una soluzione promettente per affrontare tali sfide. Proposto originariamente da Volder nel 1959, l’algoritmo CORDIC consente di calcolare funzioni trigonometriche,

esponenziali e logaritmiche attraverso un approccio iterativo basato su rotazioni vettoriali, utilizzando esclusivamente operazioni di somma e spostamento binario.

Questa caratteristica lo rende particolarmente adatto per l'implementazione su FPGA, dove le operazioni aritmetiche semplici possono essere facilmente parallelizzate e ottimizzate. Inoltre, l'algoritmo CORDIC offre un eccellente compromesso tra precisione e costo computazionale, rendendolo ideale per applicazioni che richiedono il processamento in tempo reale, come la stima dell'AoA. L'obiettivo principale è lo sviluppo di un'implementazione efficiente dell'algoritmo CORDIC su FPGA, con particolare attenzione al suo utilizzo a supporto della stima dell'angolo di arrivo. L'attenzione è posta sulla riduzione della complessità computazionale e sull'ottimizzazione delle prestazioni hardware, al fine di garantire tempi di elaborazione ridotti e un utilizzo efficiente delle risorse disponibili. Tale implementazione mira a fornire un contributo significativo nel campo del processamento dei segnali, offrendo una soluzione che possa essere applicata in scenari reali, come i sistemi di comunicazione wireless e le tecnologie di localizzazione. Un aspetto cruciale della stima dell'AoA è la sua applicazione alla localizzazione, che consente di determinare la posizione di una sorgente di segnale nello spazio. Questo processo è alla base di numerose tecnologie moderne, tra cui i sistemi di posizionamento indoor basati su reti Wi-Fi o Bluetooth, i sistemi di gestione del traffico aereo e le reti di sensori distribuiti.

La capacità di stimare con precisione l'angolo di arrivo di un segnale può migliorare significativamente l'accuratezza della localizzazione, specialmente in ambienti complessi dove ostacoli e riflessioni possono introdurre errori. L'integrazione dell'algoritmo CORDIC in questo contesto permette di ottimizzare il calcolo delle informazioni angolari, riducendo il carico computazionale e migliorando l'efficienza complessiva del sistema. Per raggiungere gli obiettivi prefissati, il lavoro si articola attraverso una serie di fasi metodologiche ben definite. In primo luogo, viene condotto uno studio teorico approfondito dell'algoritmo CORDIC, analizzandone le proprietà matematiche e le modalità di implementazione su hardware. Questo passaggio è fondamentale per comprendere le potenzialità e i limiti dell'algoritmo, nonché per identificare le strategie di ottimizzazione più appropriate. La seconda fase consiste nella progettazione dell'architettura su FPGA. Questo processo prevede la definizione di un'architettura modulare che consenta di implementare l'algoritmo CORDIC in modo efficiente, sfruttando le capacità di parallelismo offerte dall'hardware. Particolare attenzione è dedicata alla gestione delle risorse. Durante questa fase, vengono esplorati diversi approcci di design, tra cui l'uso di pipeline e l'ottimizzazione delle iterazioni CORDIC, per garantire un bilanciamento tra velocità di calcolo e precisione. La terza fase è rappresentata dall'implementazione pratica dell'algoritmo CORDIC su FPGA.

Questa fase include la scrittura del codice in un linguaggio di descrizione hardware, e la sua sintesi su una piattaforma FPGA specifica. Durante l'implementazione, vengono adottate tecniche di ottimizzazione, come la riduzione del numero di iterazioni CORDIC e l'uso di approssimazioni per migliorare le prestazioni senza sacrificare la precisione. La quarta fase riguarda la simulazione e la verifica dei risultati. Attraverso strumenti di simulazione come ModelSim, l'implementazione

viene testata in scenari realistici, utilizzando segnali sintetici e dati sperimentali. Questa fase consente di valutare la correttezza dell'implementazione, verificando che i risultati ottenuti siano coerenti con le aspettative teoriche. La quinta e ultima fase consiste nella fase sperimentale, durante la quale l'implementazione realizzata viene testata direttamente su una piattaforma FPGA all'interno di un ambiente di laboratorio. In questa fase, vengono condotti esperimenti pratici finalizzati alla valutazione delle prestazioni reali del sistema, con particolare attenzione ai parametri di accuratezza, latenza, utilizzo delle risorse hardware e stabilità operativa.

I test vengono eseguiti su segnali rappresentativi delle condizioni operative previste, al fine di simulare scenari realistici di utilizzo. I risultati ottenuti sono quindi analizzati e confrontati con le previsioni teoriche, al fine di validare l'efficacia dell'approccio proposto e individuare eventuali margini di miglioramento. L'importanza di questo lavoro risiede nella sua capacità di affrontare una problematica reale e attuale nel campo del processamento dei segnali, proponendo una soluzione innovativa che combina teoria e pratica. La stima dell'angolo di arrivo è un elemento chiave in molte applicazioni moderne, e l'ottimizzazione del suo calcolo attraverso l'algoritmo CORDIC su FPGA può avere un impatto significativo su settori come le telecomunicazioni, la difesa e l'automazione. Inoltre, il lavoro fornisce un'analisi dettagliata delle prestazioni e delle sfide associate all'implementazione hardware di algoritmi complessi, aprendo la strada a ulteriori sviluppi in questo ambito.



# Capitolo 1

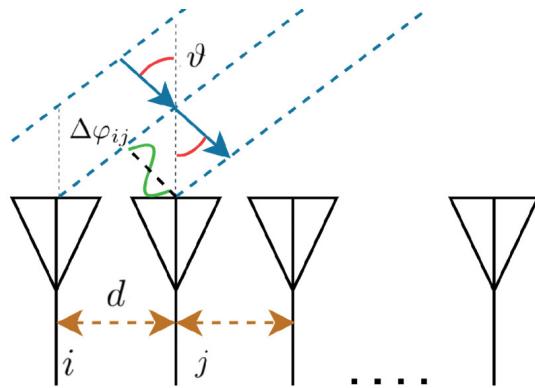
## LOCALIZZAZIONE BASATA SULLA STIMA DELL'ANGOLO DI ARRIVO

I sistemi di localizzazione in tempo reale, noti come Real-Time Location Systems (RTLS), costituiscono una delle innovazioni più rilevanti nel settore delle telecomunicazioni, offrendo la capacità di determinare la posizione di oggetti, persone o dispositivi in ambienti complessi con precisione elevata e latenza minima. Questi sistemi trovano applicazione in una vasta gamma di contesti, dalla logistica, dove il monitoraggio in tempo reale delle risorse in magazzini ottimizza la gestione delle scorte, alla sanità, dove la localizzazione di pazienti o attrezzature mediche migliora l'efficienza operativa, fino alle comunicazioni mobili, dove tecniche come il beamforming nelle reti 5G, si basano su informazioni di posizione accurate.

In ambienti interni, ad esempio, la presenza di muri, mobili o altre strutture causa fenomeni di multipath, ossia la generazione di più copie dello stesso segnale che giungono al ricevitore con diversi ritardi e attenuazioni, sovrapponendosi tra loro e deteriorando la stima finale della posizione. Analogamente, la necessità di sincronizzare molteplici ricevitori e di elaborare grandi volumi di dati in tempo reale richiede algoritmi efficienti e piattaforme hardware avanzate. Gli RTLS si avvalgono di diverse tecniche di misura per determinare la posizione, tra cui la misura della potenza del segnale ricevuto (Received Signal Strength, RSS), che valuta l'attenuazione del segnale per stimare la distanza, il tempo di arrivo (Time of Arrival, ToA), che misura il tempo impiegato dal segnale per raggiungere il ricevitore, e la differenza di tempo di arrivo (Time Difference of Arrival, TDoA), che confronta i tempi di arrivo tra più ricevitori.

Tra queste, la stima dell'angolo di arrivo (Angle of Arrival, AoA) si distingue per la sua capacità di determinare con elevata precisione la direzione da cui proviene un segnale, rendendola particolarmente utile in applicazioni che richiedono un'elevata risoluzione spaziale. Come illustrato in Figura 1.1, un'architettura di antenna comunemente utilizzata per l'AoA è l'Uniform Linear Array. Le principali soluzioni impiegate negli RTLS includono Wi-Fi, che garantisce una copertura estesa ma una precisione limitata; Bluetooth Low Energy (BLE), apprezzato per il basso consumo energetico; e le tecnologie Ultra-Wideband (UWB), una categoria di standard caratterizzati da un'ampia occupazione di banda e da un'elevata

accuratezza nella misura di parametri come il ToA e l’AoA. Quest’ultima risulta particolarmente efficace anche in ambienti Non-Line-of-Sight (NLoS), dove la presenza di ostacoli causa riflessioni multiple. Diversamente, negli scenari Line-of-Sight (LoS), in cui il segnale si propaga in modo diretto, semplificano la stima della posizione, pur risultando meno frequenti in applicazioni reali. La scelta della tecnologia e della metrica più adeguata dipende infine dai requisiti dell’applicazione, come la precisione desiderata, la scalabilità del sistema e le limitazioni energetiche [1].



**Figura 1.1:** ULA con onda piana incidente da angolo  $\vartheta$ . Immagine da [2], © 2025 IEEE.

In questo contesto, i dispositivi FPGA emergono come piattaforme ideali per l’implementazione degli RTLS, grazie alla loro capacità di eseguire operazioni parallele e alla riconfigurabilità che consente di adattare l’hardware a esigenze specifiche.

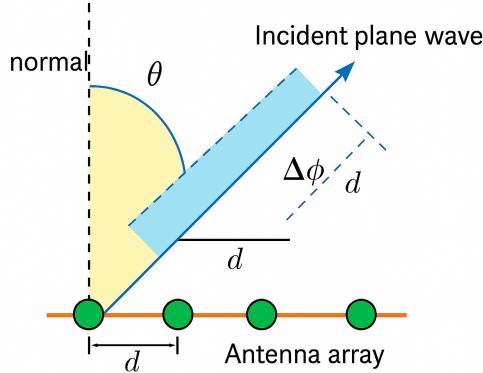
## 1.1 Stima dell’Angolo di Arrivo: Fondamenti e Applicazioni

La stima dell’angolo di arrivo rappresenta una tecnica fondamentale per gli RTLS, poiché consente di determinare la direzione di provenienza di un segnale rispetto a un array di antenne, ossia una configurazione ordinata di più elementi radiant, che permette di elaborare le informazioni spaziali del segnale e fornire dati angolari essenziali per la localizzazione precisa di una sorgente.

Questa metodologia si basa sull’analisi delle differenze di fase o di tempo di arrivo dei segnali ricevuti da più elementi di un array, che dipendono dalla geometria dell’array e dalla direzione del segnale. In un ULA (Uniform Linear Array), come illustrato in Figura 1.2, la differenza di fase tra due antenne adiacenti è descritta dalla relazione,

$$\Delta\phi = \frac{2\pi d \sin \theta}{\lambda} \quad (1.1)$$

dove  $d$  è la distanza tra gli elementi,  $\theta$  è l’angolo di arrivo rispetto alla normale dell’array, e  $\lambda$  è la lunghezza d’onda del segnale.



**Figura 1.2:** Definizione dell’angolo di arrivo  $\theta$  rispetto alla normale di un array lineare di antenne con spaziatura  $d$ .

Questa equazione permette di calcolare l’angolo  $\theta$  a partire dalla misura della differenza di fase, che diventa il fulcro della stima dell’AoA. Il modello matematico del segnale ricevuto è tipicamente rappresentato come [3],

$$\mathbf{x}(t) = \mathbf{a}(\theta)s(t) + \mathbf{n}(t) \quad (1.2)$$

dove  $\mathbf{x}(t)$  è il vettore dei segnali ricevuti,  $\mathbf{a}(\theta)$  è il steering vector, che modella il ritardo di fase tra gli elementi dell’array associato all’angolo di arrivo  $\theta$ ,  $s(t)$  è il segnale trasmesso, e  $\mathbf{n}(t)$  è il rumore additivo. L’obiettivo della stima è estrarre  $\theta$  da  $\mathbf{x}(t)$ , spesso attraverso l’analisi della matrice di covarianza dei segnali ricevuti, che fornisce informazioni sulle correlazioni tra i segnali raccolti dai diversi elementi dell’array.

Tuttavia, la qualità della stima dipende fortemente dall’ambiente operativo, che può essere di tipo LoS o NLoS. In condizioni LoS, il segnale segue un percorso diretto dal trasmettitore al ricevitore, senza ostacoli significativi, garantendo una propagazione prevedibile e una stima dell’AoA accurata. In tali scenari, algoritmi semplici come il Bartlett beamformer sono utilizzati per stimare l’AoA a partire dalle misure ottenute da un array di antenne o sensori. Questo metodo si basa sulla valutazione di una funzione di risposta spaziale, espressa come

$$P(\theta) = \mathbf{a}^H(\theta)\mathbf{R}\mathbf{a}(\theta) \quad (1.3)$$

dove  $\mathbf{R}$  è la matrice di covarianza, e l’angolo che massimizza  $P(\theta)$  è considerato la stima dell’AoA. La semplicità computazionale del beamforming lo rende adatto a implementazioni su hardware a risorse limitate, ma la sua risoluzione angolare è limitata, soprattutto in presenza di segnali correlati o cammini multipli. In ambienti NLoS, invece, la presenza di ostacoli come muri o mobili genera riflessioni, diffrazioni e scattering, dando origine a cammini multipli che complicano il modello del segnale. In questi casi, il segnale ricevuto è rappresentato come,

$$\mathbf{x}(t) = \sum_{k=1}^K \mathbf{a}(\theta_k)s_k(t) + \mathbf{n}(t) \quad (1.4)$$

dove  $K$  è il numero di cammini, ciascuno associato a un angolo  $\theta_k$ . Questa complessità richiede algoritmi più sofisticati, come MUSIC (Multiple Signal Classification) o ESPRIT (Estimation of Signal Parameters via Rotational Invariance Techniques). MUSIC sfrutta la decomposizione in autovalori della matrice di covarianza per separare il sottospazio del segnale da quello del rumore, definendo una funzione di pseudo spettro,

$$P_{MUSIC}(\theta) = \frac{1}{\mathbf{a}^H(\theta)\mathbf{E}_n\mathbf{E}_n^H\mathbf{a}(\theta)} \quad (1.5)$$

dove  $E_n$  è la matrice degli autovettori associati al sottospazio del rumore. I picchi di questa funzione indicano gli angoli di arrivo, offrendo una risoluzione superiore rispetto al beamforming.

ESPRIT, d'altra parte, utilizza la struttura invariante per rotazione del sottospazio del segnale per stimare gli angoli senza una ricerca spettrale, riducendo la complessità computazionale. Entrambi gli algoritmi, tuttavia, sono sensibili a errori di calibrazione dell'array, come imprecisioni nella posizione degli elementi, e richiedono una conoscenza accurata del numero di sorgenti, il che può essere problematico in ambienti dinamici. Al fine di porre rimedio a questa problematica, si ricorre ad approcci basati sul machine learning, come le Deep Neural Networks (DNN)<sup>1</sup>, hanno mostrato un potenziale significativo nella stima dell'AoA, specialmente in scenari NLoS.

Questi metodi mappano direttamente i segnali ricevuti agli angoli di arrivo, imparando a compensare gli effetti dei cammini multipli e delle imperfezioni hardware attraverso l'addestramento su grandi dataset. Tuttavia, la loro complessità computazionale e la necessità di risorse per l'addestramento li rendono meno pratici per applicazioni embedded rispetto a soluzioni hardware ottimizzate. La distinzione tra LoS e NLoS è cruciale per valutare le prestazioni degli algoritmi di stima dell'AoA.

In LoS, la propagazione diretta consente di utilizzare modelli semplici e algoritmi meno complessi, mentre in NLoS la presenza di cammini multipli richiede tecniche avanzate e una maggiore potenza di calcolo. Questa variabilità sottolinea l'importanza di piattaforme hardware flessibili, come gli FPGA, che possono essere configurate per supportare diverse strategie di stima in base alle condizioni operative [3].

## 1.2 Dalla Stima dell'AoA alla Localizzazione

La stima dell'AoA non è un fine in sé, ma un mezzo per la localizzazione, un'applicazione fondamentale degli RTLS che consiste nel determinare la posizione di una sorgente in uno spazio bidimensionale o tridimensionale. Uno degli aspetti fondamentali nella localizzazione è la relazione tra precisione angolare e risoluzione spaziale.

---

<sup>1</sup>Le DNN sono un tipo avanzato di rete neurale artificiale composta da più strati di nodi, che permettono di modellare relazioni complesse tra input e output.

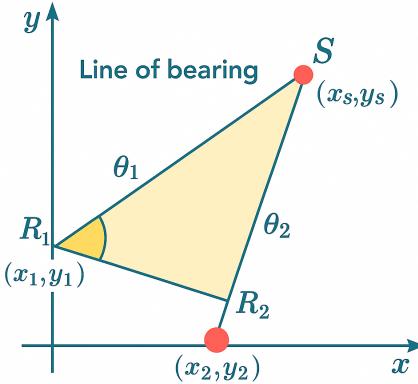
La precisione angolare si riferisce alla capacità del sistema di distinguere tra due sorgenti che arrivano con angoli leggermente differenti, mentre la risoluzione spaziale indica quanto accuratamente è possibile determinare la posizione nello spazio della sorgente. Una maggiore precisione angolare consente una triangolazione più accurata, migliorando quindi la risoluzione spaziale della localizzazione.

Tuttavia, questo richiede array di antenne con un numero elevato di elementi e una geometria ottimizzata, oltre che algoritmi avanzati capaci di discriminare segnali con differenze angolari minime. In ambienti complessi, la risoluzione spaziale può essere degradata da fenomeni di riflessione e interferenza, rendendo cruciale l'utilizzo di tecniche che migliorino la precisione angolare anche in presenza di segnali distorti [4]. Un altro elemento determinante è la larghezza di banda del segnale utilizzato. Essa influenza direttamente sulla capacità del sistema di discriminare tra percorsi multipli e sulla precisione della stima dei ritardi temporali o delle differenze di fase. I segnali a larga banda, come quelli impiegati nelle tecnologie UWB, sono in grado di fornire una risoluzione temporale molto elevata, facilitando la separazione tra cammini multipli e migliorando la stima del cammino diretto.

Questo si traduce in una maggiore precisione angolare e, di conseguenza, in una localizzazione più affidabile anche in ambienti ostili. Tuttavia, l'aumento della banda comporta anche una maggiore complessità hardware e maggiori requisiti in termini di potenza di elaborazione e campionamento [5]. La localizzazione basata sull'AoA sfrutta le informazioni angolari fornite dagli angoli di arrivo misurati da più ricevitori per definire linee di direzione (*Lines of Bearing*), come illustrato in Figura 1.3. Queste sono linee immaginarie lungo le quali un segnale si propaga da una sorgente verso un ricevitore, secondo una determinata direzione angolare, la cui intersezione identifica la posizione della sorgente. Geometricamente, per due ricevitori con angoli  $\theta_1$  e  $\theta_2$  e posizioni note  $(x_1, y_1)$  e  $(x_2, y_2)$ , la posizione della sorgente  $(x_s, y_s)$  è calcolata risolvendo il sistema di equazioni

$$\tan(\theta_1) = \frac{y_s - y_1}{x_s - x_1} \quad e \quad \tan(\theta_2) = \frac{y_s - y_2}{x_s - x_2} \quad (1.6)$$

Quando sono disponibili più di due ricevitori, si utilizzano tecniche di minimizzazione degli errori, come il metodo dei minimi quadrati, migliorando l'accuratezza e compensando le imprecisioni nella stima dell'AoA.



**Figura 1.3:** Rappresentazione geometrica della localizzazione basata sull’AoA. Le linee tracciate dai ricevitori  $R_1$  e  $R_2$  nei rispettivi angoli  $\theta_1$  e  $\theta_2$  convergono nella posizione stimata della sorgente  $S$ .

La localizzazione basata sull’AoA è particolarmente efficace in ambienti LoS, dove la propagazione diretta del segnale assicura angoli precisi e una triangolazione affidabile. In scenari NLoS, invece, i cammini multipli introducono complessità, poiché i ricevitori possono rilevare angoli associati a percorsi riflessi anziché al cammino diretto. In tali casi, algoritmi come MUSIC o ESPRIT sono utilizzati per separare i contributi dei diversi cammini, e CORDIC supporta l’elaborazione degli angoli multipli, consentendo al sistema di identificare il cammino principale o di combinare più angoli per una stima più robusta.

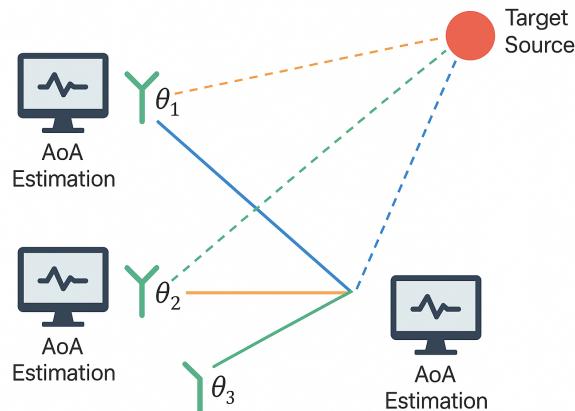
Ciascuno di questi algoritmi ha punti di forza e limiti ben specifici. Il beamforming convenzionale è semplice e computazionalmente efficiente, ma soffre di una risoluzione limitata e di una scarsa capacità di discriminare sorgenti vicine. L’algoritmo MUSIC, basato sull’analisi del sottospazio, offre una risoluzione angolare significativamente superiore rispetto ai metodi convenzionali e si dimostra efficace anche in presenza di segnali correlati, a condizione che sia noto il numero di sorgenti. Tuttavia, questa maggiore accuratezza è ottenuta al prezzo di una complessità computazionale elevata, che può limitarne l’impiego in applicazioni real-time o su dispositivi con risorse limitate. ESPRIT, pur mantenendo elevate prestazioni, presenta una minore complessità computazionale rispetto a MUSIC grazie alla sua struttura matematica più compatta. La scelta dell’algoritmo più adatto dipende quindi dal contesto applicativo, dalla disponibilità di risorse computazionali e dalla conoscenza a priori dei parametri del sistema [6, 7, 8].

La localizzazione può essere ulteriormente potenziata attraverso approcci ibridi, che integrano tecniche come ToA o RSS per migliorare la precisione in ambienti complessi. Ad esempio, combinando l’AoA con il ToA, un sistema può utilizzare gli angoli per restringere l’area di ricerca e le distanze per calcolare la posizione esatta, riducendo l’incertezza in presenza di cammini multipli. Un altro approccio promettente è la localizzazione cooperativa, in cui i nodi di una rete condividono informazioni angolari per costruire una mappa delle posizioni, un paradigma par-

ticolarmente rilevante per IoT (Internet of Things), dove la densità dei dispositivi richiede soluzioni scalabili.

Le applicazioni della localizzazione basata sull'AoA sono molteplici e in continua espansione.

Nella navigazione indoor, ad esempio, i sistemi basati su UWB o Bluetooth utilizzano l'AoA per localizzare dispositivi in ambienti come ospedali, centri commerciali o magazzini, dove il GPS è inefficace. Nelle reti 5G, l'AoA supporta il beamforming direzionale, ottimizzando la trasmissione del segnale verso gli utenti e migliorando l'efficienza spettrale [9]. Altri ambiti includono la sorveglianza, dove la localizzazione precisa di sorgenti di segnale è essenziale per il monitoraggio, e la robotica, dove i robot autonomi utilizzano informazioni di posizione per la navigazione e l'interazione con l'ambiente, come illustrato in Figura 1.1.



**Figura 1.4:** Una rappresentazione schematica della localizzazione basata su AoA, che richiede sistemi di stima dell'AoA distribuiti.

Tuttavia, la localizzazione eredita le sfide della stima dell'AoA, come i cammini multipli, gli errori di calibrazione e la necessità di sincronizzazione tra ricevitori. In ambienti dinamici, dove la sorgente o i ricevitori sono in movimento, la stima in tempo reale richiede algoritmi altamente efficienti e hardware potente. Inoltre, la scalabilità è un problema in reti dense, come quelle IoT, dove il numero elevato di nodi può generare interferenze e sovraccaricare gli algoritmi di localizzazione.

In un sistema di localizzazione indoor, un array di antenne basato su UWB riceve segnali da un dispositivo mobile, e l'FPGA elabora le differenze di fase in tempo reale utilizzando CORDIC per calcolare gli angoli di arrivo. Questi angoli sono poi utilizzati per triangolare la posizione del dispositivo, anche in presenza di riflessioni causate da muri o mobili.

La capacità di CORDIC di operare con bassa latenza e consumo energetico rende questa implementazione ideale per applicazioni RTLS, dove la rapidità e l'efficienza sono prioritarie. Nonostante i progressi ottenuti, persistono sfide rilevanti legate alla robustezza in ambienti NLoS e alla scalabilità in scenari caratter-

rizzati da reti dense. Questi aspetti rappresentano ancora aree di ricerca attiva, in particolare per quanto riguarda l'integrazione di tecniche di machine learning e l'ottimizzazione dell'hardware, con l'obiettivo di migliorare le prestazioni complessive dei sistemi. [10].

### 1.3 Panoramica sugli FPGA

Gli FPGA rappresentano una categoria di dispositivi elettronici riconfigurabili che hanno rivoluzionato il campo della progettazione di sistemi digitali. Grazie alla loro flessibilità e alle elevate prestazioni, gli FPGA sono diventati uno strumento fondamentale in numerose applicazioni, specialmente in ambiti che richiedono elaborazione in tempo reale, come il processamento dei segnali, le telecomunicazioni e la localizzazione. Questa sezione fornisce una panoramica completa su cosa sono gli FPGA, il loro funzionamento, le applicazioni principali e la loro struttura interna, con particolare attenzione al contesto di implementazione dell'algoritmo CORDIC per la stima dell'angolo di arrivo.

Un FPGA è un circuito integrato programmabile che consente agli utenti di configurare l'hardware per eseguire funzioni logiche personalizzate. A differenza dei microprocessori o dei microcontrollori, che eseguono istruzioni software su un'architettura fissa, gli FPGA permettono di definire l'architettura hardware stessa, rendendoli estremamente versatili. La programmabilità deriva dalla presenza di blocchi logici riconfigurabili, interconnessioni flessibili e risorse di memoria integrate, che possono essere configurate tramite linguaggi di descrizione hardware come VHDL o Verilog.

Il concetto di FPGA è stato introdotto negli anni '80 da Xilinx [11], con l'obiettivo di offrire una soluzione intermedia tra gli Application Specific Integrated Circuits (ASIC) e i processori general-purpose. Gli ASIC offrono prestazioni elevate ma richiedono costi e tempi di sviluppo significativi, mentre i microprocessori sono flessibili ma spesso meno efficienti per compiti specifici. Gli FPGA combinano i vantaggi di entrambe le tecnologie: la capacità di eseguire operazioni parallele ad alta velocità, tipica degli ASIC, e la possibilità di riconfigurazione, simile ai sistemi basati su software. Questa caratteristica li rende ideali per la prototipazione rapida, l'implementazione di algoritmi complessi e l'adattamento a requisiti mutevoli.

Gli FPGA trovano applicazione in una vasta gamma di settori [12] grazie alla loro capacità di eseguire calcoli paralleli, gestire grandi volumi di dati in tempo reale e adattarsi a specifiche esigenze applicative. Nel contesto del processamento dei segnali, come nella stima dell'AoA, gli FPGA sono utilizzati per implementare algoritmi come il CORDIC [13, 14, 15], che richiedono calcoli trigonometrici rapidi ed efficienti. Altre applicazioni comuni includono:

- **Telecomunicazioni:** Gli FPGA sono utilizzati per il beamforming nelle reti 5G, la modulazione digitale e la codifica/decodifica dei segnali. La loro capacità di elaborare segnali in tempo reale li rende essenziali per gestire protocolli di comunicazione complessi.

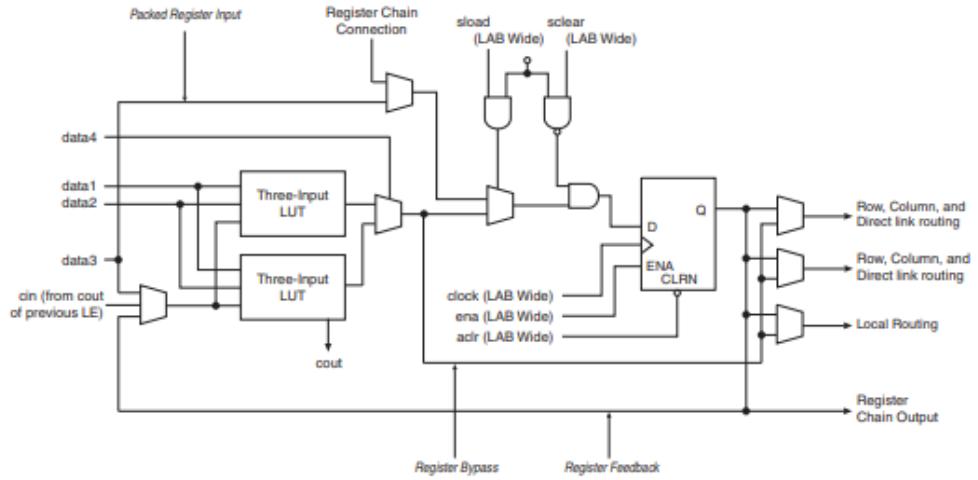
- Elaborazione delle immagini e video: Gli FPGA supportano applicazioni come la compressione video, il riconoscimento di pattern e l'elaborazione in tempo reale di flussi video ad alta risoluzione.
- Sistemi embedded: Negli ambienti IoT, gli FPGA sono impiegati per il controllo di sensori, l'elaborazione di dati in tempo reale e la gestione di protocolli di comunicazione a basso consumo.
- Radar e sistemi di difesa: Gli FPGA sono utilizzati per l'elaborazione dei segnali radar, la localizzazione di target e la gestione di sistemi di sorveglianza in tempo reale.
- Prototipazione e ricerca: Grazie alla loro riconfigurabilità, gli FPGA sono ampiamente utilizzati per testare nuovi algoritmi e architetture hardware prima della produzione di ASIC.

### 1.3.1 Struttura interna di un FPGA

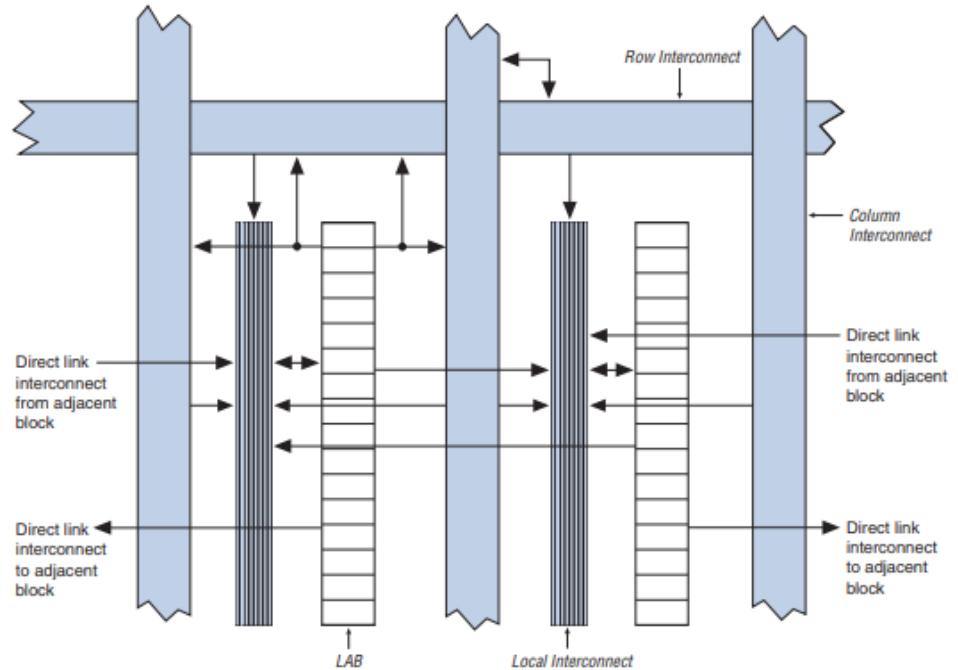
L'architettura interna di un FPGA, in particolare quella di Intel, è composta da diversi componenti fondamentali che lavorano in sinergia per eseguire le funzioni programmate [16, 17], come illustrato nelle Figure 1.5, 1.6. Questi componenti includono:

- Blocchi logici configurabili (CLB): I CLB sono le unità di base che implementano le funzioni logiche. Ogni CLB contiene tabelle di lookup (LUT) per eseguire operazioni logiche, flip-flop per memorizzare stati e multiplex per configurare le connessioni. Nel contesto del CORDIC, i CLB sono utilizzati per implementare le operazioni di addizione, sottrazione e shift binari.
- Matrice di interconnessione: Una rete programmabile di linee e interruttori che collega i CLB, i blocchi di I/O e altre risorse. Questa matrice consente di creare percorsi di segnale personalizzati, garantendo flessibilità nella configurazione del circuito.
- Blocchi di I/O: Questi blocchi gestiscono l'interfacciamento con il mondo esterno, come sensori, antenne o dispositivi di comunicazione.
- Blocchi DSP (Digital Signal Processing): Risorse dedicate per operazioni aritmetiche complesse, come moltiplicazioni e accumulazioni.
- Memoria integrata: Gli FPGA includono blocchi di memoria RAM (ad esempio, M9K nella Cyclone IV) per memorizzare dati temporanei, tabelle di lookup o valori predefiniti.
- PLL (Phase-Locked Loop): Circuiti per la gestione del clock, che permettono di generare segnali di clock a diverse frequenze.

All'interno dei CLB, gli elementi fondamentali che realizzano le funzioni logiche sono i Logic Element (LE). Ogni LE è una piccola unità logica programmabile composta da una o più LUT, flip-flop e logiche di controllo [17], ed è in grado di implementare operazioni combinatorie o sequenziali. Nei dispositivi Cyclone IV, un LAB è costituito da 16 LE identici che condividono risorse comuni come il clock e i segnali di controllo.



**Figura 1.5:** Struttura interna di un Logic Element in modalità aritmetica. Immagine da [16], © 2016 Altera Corporation



**Figura 1.6:** Schema della struttura di un Logic Array Block. Immagine da [16], © 2016 Altera Corporation.

La Figura 1.5 rappresenta l’architettura interna di un Logic Element nel Cyclone IV in modalità aritmetica. Ogni LE è composto da due Look-Up Table a tre ingressi, che possono essere configurate per implementare funzioni logiche combinatorie arbitrarie. L’output delle LUT può essere combinato tramite una logica sommatrice, rendendo possibile l’uso aritmetico del blocco, utile in applicazioni come il CORDIC [18]. Inoltre, il LE contiene un flip-flop che può essere usato per la sincronizzazione o la memorizzazione temporanea, e varie connessioni per routing locale e globale, incluso il supporto per il chaining dei registri.

La Figura 1.6 mostra invece la struttura di un Logic Array Block, che è costituito da 16 LE raggruppati insieme e collegati a una rete di interconnessione locale. I LAB sono disposti in una griglia e connessi tra loro tramite interconnessioni di riga e colonna, oltre a collegamenti diretti con blocchi adiacenti. Questo schema modulare e riconfigurabile è alla base della flessibilità degli FPGA, permettendo la realizzazione di architetture logiche complesse e ad alte prestazioni.

### 1.3.2 Vantaggi e limitazioni degli FPGA

Gli FPGA offrono numerosi vantaggi che li rendono ideali per applicazioni come la stima dell’AoA:

- Parallelismo: La capacità di eseguire più operazioni simultaneamente riduce la latenza, un aspetto cruciale per applicazioni in tempo reale.
- Riconfigurabilità: La possibilità di modificare l’hardware senza interventi fisici consente di adattare il sistema a nuovi algoritmi o requisiti.
- Efficienza energetica: Rispetto ai microprocessori, gli FPGA consumano meno energia per compiti specifici.
- Costo: Gli FPGA offrono notevoli benefici in termini di costi e tempi di sviluppo, specialmente per la prototipazione e per progetti a basso volume, dove risultano una soluzione più economica e flessibile rispetto agli ASIC.

Tuttavia, presentano anche alcune limitazioni:

- Complessità di progettazione: La scrittura di codice VHDL o Verilog richiede competenze specialistiche, e il processo di sintesi e debug può essere lungo.
- Risorse limitate: Gli FPGA hanno un numero finito di CLB, memoria e I/O, che possono rappresentare un vincolo per progetti complessi.



## Capitolo 2

# TEORIA E IMPLEMENTAZIONE DELL'ALGORITMO CORDIC

L'algoritmo CORDIC è stato sviluppato per rispondere alle esigenze di calcolo efficiente in contesti con risorse computazionali limitate. La sua origine risale al 1959, quando Jack E. Volder lo introdusse con l'obiettivo di migliorare i sistemi di navigazione aeronautica. In quel periodo, l'industria cercava soluzioni per sostituire i circuiti analogici, che erano meno affidabili, con metodi digitali più precisi e robusti. Volder progettò il CORDIC per calcolare funzioni trigonometriche come il seno e il coseno, operazioni fondamentali per determinare angoli e traiettorie in tempo reale. Questo approccio si rivelò rivoluzionario perché consentiva di eseguire calcoli complessi utilizzando esclusivamente addizioni, sottrazioni e shift binari, operazioni estremamente semplici da implementare su hardware digitale dell'epoca [13].

Nel 1971, John Walther ampliò il lavoro di Volder, introducendo estensioni che permisero al CORDIC di calcolare non solo funzioni trigonometriche, ma anche funzioni iperboliche, esponenziali e logaritmiche. Grazie a queste innovazioni, l'algoritmo divenne uno strumento versatile, capace di affrontare una vasta gamma di problemi matematici con un'unica metodologia computazionale. L'espansione di Walther rese il CORDIC particolarmente adatto a contesti applicativi come l'elaborazione dei segnali, la modulazione digitale e i sistemi radar, dove la velocità e l'efficienza erano cruciali [14].

Con l'avvento delle FPGA, negli anni '80 e '90, il CORDIC trovò un nuovo ambito di applicazione ideale. Le FPGA sono dispositivi hardware riconfigurabili che consentono di implementare algoritmi digitali in modo altamente ottimizzato. La semplicità delle operazioni richieste dal CORDIC, unita alla sua natura iterativa, lo rese perfetto per queste piattaforme, dove il consumo di risorse logiche e la velocità di elaborazione sono parametri fondamentali. Negli ultimi decenni, l'algoritmo è stato ulteriormente affinato per applicazioni embedded, con varianti che migliorano le prestazioni in termini di latenza e consumo energetico. Ad oggi, a oltre 60 anni dalla sua introduzione, il CORDIC rimane una scelta privilegiata in molte applicazioni ingegneristiche, specialmente in scenari che richiedono elaborazione in tempo reale, come la stima dell'angolo di arrivo [15].

Il contesto storico del CORDIC è particolarmente significativo. Nato in un'epoca in cui i computer erano grandi, costosi e lenti, ha anticipato le esigenze moderne di calcolo efficiente su dispositivi compatti e a basso consumo. La sua capacità di operare senza moltiplicazioni lo ha reso un precursore degli algoritmi ottimizzati per hardware, un aspetto che lo rende ancora oggi rilevante in ambiti come l'IoT e i sistemi di comunicazione wireless.

## 2.1 Principi Matematici

Il fondamento matematico del CORDIC si basa sulla rotazione vettoriale in uno spazio bidimensionale. Consideriamo un vettore con componenti cartesiane  $x$  e  $y$ , rappresentato come  $[x \ y]^T$ . Se vogliamo ruotare questo vettore di un angolo  $\theta$  attorno all'origine, possiamo applicare una trasformazione lineare che utilizza una matrice di rotazione. Questa trasformazione produce nuove coordinate  $x'$  e  $y'$  secondo la relazione

$$x' = x \cos \theta - y \sin \theta \quad e \quad y' = x \sin \theta + y \cos \theta. \quad (2.1)$$

In forma matriciale, possiamo scrivere questa operazione come una moltiplicazione tra la matrice  $[\cos \theta, -\sin \theta; \sin \theta, \cos \theta]$  e il vettore  $[x \ y]^T$ .

Questa trasformazione, tuttavia, richiede il calcolo diretto di  $\sin \theta$  e  $\cos \theta$ , operazioni che possono essere complesse in un contesto hardware, poiché coinvolgono moltiplicazioni e spesso l'uso di tabelle di lookup. L'innovazione del CORDIC consiste nel riformulare questa rotazione come una sequenza di micro-rotazioni più piccole, ciascuna delle quali utilizza angoli predefiniti che semplificano i calcoli. Invece di calcolare direttamente  $\sin \theta$  e  $\cos \theta$ , il CORDIC approssima l'angolo  $\theta$  come somma di angoli elementari, definiti dalla relazione  $\theta_i = \arctan(2^{-i})$ , dove  $i$  è un indice che rappresenta il passo iterativo. In questo modo, l'angolo totale  $\theta$  può essere espresso come una combinazione lineare del tipo

$$\theta \approx \sum_{i=0}^{n-1} d_i \cdot \arctan(2^{-i}), \quad (2.2)$$

dove  $d_i$  è un coefficiente che assume valori  $+1$  o  $-1$  in base alla direzione della rotazione.

La scelta degli angoli  $\arctan(2^{-i})$  è cruciale perché il valore di  $\tan(\arctan(2^{-i}))$  è esattamente  $2^{-i}$ , un numero che può essere rappresentato come uno shift binario. Questo significa che le moltiplicazioni, che normalmente sarebbero necessarie per calcolare una rotazione, vengono sostituite da operazioni di shift, molto più efficienti su hardware. Ad ogni iterazione  $i$ , il vettore viene ruotato di un angolo  $\arctan(2^{-i})$ , e le nuove coordinate  $x_{i+1}$  e  $y_{i+1}$  sono calcolate come  $x_{i+1} = x_i - d_i \cdot y_i \cdot 2^{-i}$  e  $y_{i+1} = y_i + d_i \cdot x_i \cdot 2^{-i}$ . Questa operazione introduce anche un fattore di scala, dato da  $1/\sqrt{1 + 2^{-2i}}$ , che si accumula su tutte le iterazioni. Dopo  $n$  iterazioni, il fattore di scala totale, indicato con  $K_n$ , è il prodotto di tutti i fattori di scala individuali, ovvero

$$K_n = \prod_{i=0}^{n-1} 1/\sqrt{1+2^{-2i}}. \quad (2.3)$$

Per un numero sufficiente di iterazioni, ad esempio  $n$  tendente all'infinito, questo fattore converge a un valore approssimativo di 0.607252. Questo fattore di scala deve essere compensato, a seconda dell'applicazione, per ottenere i valori corretti di  $\sin \theta$  e  $\cos \theta$ .

Il CORDIC può operare sia in coordinate cartesiane che polari, offrendo una grande versatilità. Quando lavora in coordinate cartesiane, calcola direttamente le componenti  $x$  e  $y$  di un vettore ruotato. Quando invece è usato per passare a coordinate polari, può convertire un vettore con componenti  $x$  e  $y$  in una rappresentazione polare con modulo  $\rho = \sqrt{x^2 + y^2}$  e angolo  $\theta = \arctan(y/x)$ . Questa capacità di operare in entrambi i sistemi di coordinate lo rende uno strumento potente per applicazioni come la modulazione digitale, dove è necessario passare frequentemente tra rappresentazioni cartesiane e polari, e per la stima dell'AoA, dove l'angolo  $\theta$  rappresenta la direzione di un segnale [15].

## 2.2 Modalità di Funzionamento

Il CORDIC opera in due modalità principali, ciascuna progettata per affrontare problemi specifici: la modalità rotazione e la modalità vettore. Nella modalità rotazione, l'obiettivo è calcolare  $\sin \theta$  e  $\cos \theta$  a partire da un angolo  $\theta$  noto. Si inizia con un vettore iniziale con componenti  $x_0 = 1$  e  $y_0 = 0$ , e un angolo accumulato  $z_0$  uguale a  $\theta$ . L'algoritmo esegue una serie di iterazioni, ciascuna delle quali ruota il vettore di un angolo elementare  $\arctan(2^{-i})$ . A ogni passo, le coordinate  $x$  e  $y$  vengono aggiornate secondo le relazioni

$$x_{i+1} = x_i - d_i \cdot y_i \cdot 2^{-i} \quad e \quad y_{i+1} = y_i + d_i \cdot x_i \cdot 2^{-i}, \quad (2.4)$$

mentre l'angolo accumulato  $z$  viene aggiornato come

$$z_{i+1} = z_i - d_i \cdot \arctan(2^{-i}). \quad (2.5)$$

La direzione della rotazione, determinata dal coefficiente  $d_i$ , dipende dal segno di  $z_i$ : se  $z_i$  è negativo,  $d_i$  è  $-1$ , altrimenti è  $+1$ . L'obiettivo è ridurre  $z_i$  a zero, e al termine delle iterazioni, dopo ad esempio  $n = 16$  passi,  $x_n$  e  $y_n$  approssimano rispettivamente  $K_n \cdot \cos \theta$  e  $K_n \cdot \sin \theta$ . Questa modalità è particolarmente utile per generare segnali sinusoidali, come quelli necessari in applicazioni di beamforming o nella sintesi di segnali per comunicazioni [15].

Nella modalità vettore, invece, l'obiettivo è calcolare  $\arctan(y/x)$  e la norma di un vettore con componenti iniziali  $x_0$  e  $y_0$ . Si parte con un angolo accumulato  $z_0$  uguale a zero, e il vettore viene ruotato iterativamente fino a che la sua componente  $y$  non diventa trascurabile, cioè  $y_n$  tende a zero. Le equazioni di aggiornamento sono le stesse della modalità rotazione, ma in questo caso  $d_i$  è determinato dal segno di  $y_i$ : se  $y_i$  è negativo,  $d_i$  è  $-1$ , altrimenti è  $+1$ . Al termine delle iterazioni,  $z_n$

rappresenta l'angolo  $\arctan(y_0/x_0)$ , mentre  $x_n$  approssima  $K_n \cdot \sqrt{x_0^2 + y_0^2}$ . Questa modalità è fondamentale per applicazioni come la stima dell'AoA, dove è necessario calcolare l'angolo di incidenza di un segnale a partire dalle sue componenti cartesiane.

Queste due modalità possono essere combinate per affrontare problemi più complessi. Ad esempio, è possibile usare la modalità vettore per calcolare un angolo e poi la modalità rotazione per generare un segnale sinusoidale basato su quell'angolo, un approccio utile in applicazioni come la demodulazione di segnali complessi o la conversione tra sistemi di coordinate.

## 2.3 Versatilità dell'Algoritmo CORDIC

Il CORDIC è un algoritmo estremamente versatile, capace di calcolare diverse funzioni matematiche utilizzando un'unica metodologia. Una delle sue applicazioni principali è il calcolo delle funzioni trigonometriche. Nella modalità rotazione, può determinare  $\sin \theta$  e  $\cos \theta$  per un dato angolo  $\theta$ , come descritto in precedenza. Inoltre, calcolando il rapporto tra  $\sin \theta$  e  $\cos \theta$ , è possibile ottenere  $\tan \theta$ . Nella modalità vettore, il CORDIC calcola  $\arctan(y/x)$ , un'operazione fondamentale per determinare angoli a partire da coordinate cartesiane. È anche possibile calcolare  $\arcsin x$  e  $\arccos x$ , adattando la modalità vettore per trovare l'angolo corrispondente a un dato valore  $x$ , ad esempio inizializzando il vettore con componenti  $x$  e  $\sqrt{1 - x^2}$  e ruotando fino ad allinearla con l'asse  $x$ .

Oltre alle funzioni trigonometriche, il CORDIC può essere esteso per calcolare funzioni iperboliche, utilizzando una variante nota come CORDIC iperbolico. In questa configurazione, è in grado di determinare  $\sinh x$  e  $\cosh x$ , che rappresentano rispettivamente il seno e il coseno iperbolico di un valore  $x$ . Queste funzioni sono utili in applicazioni che coinvolgono segnali esponenziali, come l'elaborazione di segnali in sistemi di controllo. Inoltre, calcolando il rapporto tra  $\sinh x$  e  $\cosh x$ , si può ottenere  $\tanh x$ , la tangente iperbolica, spesso impiegata in algoritmi di controllo automatico.

Un'altra applicazione del CORDIC è il calcolo di operazioni lineari, come moltiplicazioni e divisioni. In una modalità specifica, chiamata modalità lineare, l'algoritmo utilizza shift e addizioni per eseguire queste operazioni senza ricorrere a moltiplicatori hardware, un vantaggio significativo in termini di risorse. Varianti più avanzate del CORDIC consentono anche il calcolo di logaritmi ed esponenziali, come  $\ln x$  ed  $e^x$ , operazioni che trovano applicazione in algoritmi di processamento numerico e controllo.

Infine, il CORDIC può essere utilizzato per calcolare la radice quadrata di un'espressione della forma  $x^2 + y^2$ . Questo si ottiene nella modalità vettore, dove il risultato  $x_n$  al termine delle iterazioni rappresenta  $K_n \cdot \sqrt{x_0^2 + y_0^2}$ . Questa capacità è utile per normalizzare vettori, un'operazione comune nell'elaborazione dei segnali. La versatilità del CORDIC lo rende adatto a un'ampia gamma di contesti, dalla sintesi di segnali alla localizzazione in tempo reale. In questa tesi, l'attenzione

è posta sul calcolo di  $\sin \theta$ ,  $\arcsin x$  e  $\arctan(y/x)$ , poiché queste operazioni sono direttamente rilevanti per la stima dell’AoA.

## 2.4 Vantaggi Computazionali nel Design Hardware

Uno dei principali punti di forza del CORDIC è la sua capacità di eseguire calcoli complessi senza ricorrere a moltiplicazioni. Le operazioni di moltiplicazione, che su hardware tradizionale richiedono circuiti complessi e consumano molte risorse, sono sostituite da shift binari, che consistono in semplici spostamenti di bit. Ad esempio, moltiplicare un numero per  $2^{-i}$  equivale a spostare i bit di  $i$  posizioni a destra, un’operazione che su una FPGA richiede una quantità minima di logica e può essere eseguita in un solo ciclo di clock. Questo approccio riduce drasticamente il consumo di risorse logiche, rendendo il CORDIC ideale per dispositivi con vincoli di spazio e potenza, come i sistemi embedded.

Un altro vantaggio significativo è l’efficienza computazionale. Il CORDIC richiede esclusivamente addizioni, sottrazioni e shift, operazioni che possono essere facilmente parallelizzate su hardware. Ad esempio, in una FPGA, è possibile implementare ogni iterazione del CORDIC come uno stadio di una pipeline, consentendo di processare più dati in parallelo e migliorando il throughput complessivo. Questo è particolarmente utile in applicazioni real-time, dove è necessario elaborare grandi quantità di dati in tempi brevi, come nella stima dell’AoA per sistemi di localizzazione.

La natura iterativa del CORDIC consente anche di sfruttare tecniche di pipeline e parallelismo per ridurre la latenza. In un’implementazione pipelined, ogni iterazione può essere eseguita in un ciclo di clock separato, e più iterazioni possono essere processate simultaneamente su dati diversi. Questo approccio permette di raggiungere velocità elevate, ad esempio con un throughput di un risultato ogni 1 o 2 cicli di clock, a seconda della frequenza della FPGA. Inoltre, la semplicità delle operazioni richieste dal CORDIC contribuisce a un basso consumo energetico, un fattore cruciale per dispositivi mobili o a batteria, come quelli utilizzati in applicazioni IoT o wireless.

Infine, la flessibilità del CORDIC è un altro vantaggio chiave. La sua capacità di calcolare diverse funzioni matematiche utilizzando lo stesso nucleo algoritmico lo rende adatto a una vasta gamma di applicazioni, dalla sintesi di segnali alla stima degli angoli. Questo lo rende una scelta eccellente per progetti che richiedono versatilità senza sacrificare l’efficienza [15]. Nel contesto della stima dell’AoA, la capacità del CORDIC di calcolare rapidamente  $\arctan(y/x)$  è essenziale per determinare la direzione di un segnale in tempo reale, un requisito fondamentale per sistemi di localizzazione accurati.

## 2.5 Sfide e Limitazioni del CORDIC

Nonostante i suoi numerosi vantaggi, il CORDIC presenta alcune limitazioni che devono essere prese in considerazione durante la progettazione. Un primo limite significativo è il range ristretto degli angoli per cui l'algoritmo converge. In particolare, il CORDIC garantisce una convergenza affidabile solo per angoli compresi tra  $-90^\circ$  e  $+90^\circ$ . Se l'angolo da calcolare supera questo intervallo, è necessario applicare una pre-elaborazione per mappare l'angolo nel range consentito. Ad esempio, per calcolare  $\sin(150^\circ)$ , si può sfruttare l'identità trigonometrica  $\sin(150^\circ) = \sin(180^\circ - 30^\circ) = \sin(30^\circ)$ , riducendo così l'angolo a  $30^\circ$ , che rientra nel range di convergenza. Questo processo, sebbene efficace, aggiunge un ulteriore livello di complessità al sistema.

Un altro aspetto critico è il numero di iterazioni necessarie per ottenere una buona precisione. La precisione del CORDIC dipende direttamente dal numero di iterazioni  $n$ : un numero troppo basso di iterazioni, ad esempio  $n = 8$ , può portare a errori significativi, con deviazioni che possono raggiungere valori come 0.1 o 0.2 rispetto al risultato teorico. D'altra parte, un numero eccessivo di iterazioni, come  $n = 32$ , aumenta la latenza e il consumo di risorse senza migliorare significativamente la precisione. In molte applicazioni, un valore intermedio di  $n = 16$  iterazioni offre un buon compromesso, garantendo una precisione di circa 0.001 per una rappresentazione a 16 bit, sufficiente per la maggior parte delle applicazioni di stima dell'AoA.

Un ultimo aspetto da considerare è la convergenza in configurazioni più complesse, come il CORDIC iperbolico utilizzato per calcolare  $\sinh x$  o  $\cosh x$ . In queste varianti, alcune iterazioni devono essere ripetute per garantire la convergenza, ad esempio ripetendo l'iterazione  $i = 4$  o  $i = 13$ , il che aumenta la complessità computazionale e la latenza complessiva [15]. Queste limitazioni richiedono un'attenta progettazione, soprattutto in applicazioni come la stima dell'AoA, dove errori angolari anche piccoli, come 0.5 gradi, possono tradursi in errori di localizzazione di diversi metri, a seconda della distanza del segnale.

## 2.6 Funzioni Trigonometriche in MATLAB

Prima di descrivere l'implementazione dell'algoritmo CORDIC in MATLAB, è opportuno sapere come le funzioni trigonometriche sono implementate internamente in MATLAB, al fine di poterle confrontare efficacemente con il funzionamento del CORDIC e valutarne l'attendibilità.

MATLAB è una piattaforma di calcolo numerico ad alte prestazioni sviluppata da MathWorks, che si basa internamente su librerie numeriche ottimizzate come BLAS (Basic Linear Algebra Subprograms) e LAPACK (Linear Algebra Package), ampiamente utilizzate anche in ambito scientifico e industriale. Sebbene queste librerie siano principalmente orientate all'algebra lineare, MATLAB sfrutta tecniche simili per il calcolo delle funzioni matematiche elementari, come le funzioni trigonometriche, logaritmiche ed esponenziali.

Le funzioni trigonometriche in MATLAB non si basano su semplici espansioni in serie di Taylor per garantire l'accuratezza su tutto il dominio di definizione. Impiegano invece una combinazione di tecniche numeriche avanzate e altamente ottimizzate per l'aritmetica floating-point dei processori moderni. [19]. L'obiettivo è massimizzare la precisione (spesso fino alla precisione di macchina) e la velocità di calcolo.

Le strategie principali comuni a molte di queste implementazioni includono:

- Riduzione dell'argomento (Range Reduction): Per minimizzare l'errore di approssimazione e la complessità computazionale, l'input originale viene mappato in un intervallo più piccolo e fondamentale. Questo è cruciale per mantenere la precisione anche con input molto grandi o molto piccoli. Ad esempio, per funzioni periodiche come seno e coseno,  $x$  viene ridotto modulo  $2\pi$  o  $\pi/2$ . Per le funzioni inverse, si sfruttano simmetrie.
- Approssimazioni polinomiali o razionali: Sull'intervallo ridotto, le funzioni sono approssimate con polinomi (es. polinomi di Chebyshev) o rapporti di polinomi (approssimazioni di Padé) di alto grado. Questi polinomi sono scelti e ottimizzati usando algoritmi sofisticati (come l'algoritmo di Remez) per minimizzare l'errore massimo su un dato intervallo. Sono estremamente efficienti per il calcolo in hardware floating-point, in quanto richiedono principalmente moltiplicazioni e addizioni.
- Gestione accurata dell'errore e dei casi limite: Le implementazioni professionali sono robuste contro gli errori di arrotondamento e gestiscono con precisione casi speciali (e.g., input molto grandi, zero, infiniti, NaN - Not a Number), garantendo la stabilità numerica.

Vediamo ora l'implementazione più dettagliata per alcune funzioni specifiche. Per le funzioni seno e coseno, l'argomento  $x$  viene prima ridotto all'intervallo  $[0, \pi/4]$  (o un intervallo simile, come  $[0, \pi/2]$ ) utilizzando identità trigonometriche e la periodicità. Successivamente, su questo intervallo ristretto, si applicano approssimazioni polinomiali di alto ordine. Spesso,  $\sin(x)$  e  $\cos(x)$  sono calcolate congiuntamente per ottimizzare le operazioni, sfruttando il fatto che  $\cos(x) = \sin(x + \pi/2)$ , oppure espansioni che le computano in parallelo.

La funzione arcoseno opera su un dominio di input limitato a  $[-1, 1]$ . La sua implementazione sfrutta la proprietà di simmetria  $\text{asin}(-x) = -\text{asin}(x)$  per ridurre l'intervallo di calcolo a  $[0, 1]$ . Su questo intervallo, viene poi applicata un'approssimazione polinomiale o razionale. In alcuni casi,  $\text{asin}(x)$  può essere calcolata internamente utilizzando la relazione  $\text{atan}(x/\sqrt{1-x^2})$ , il che richiede un'implementazione robusta per la radice quadrata e la funzione arcotangente.

Per le funzioni arcotangente, vengono utilizzate due funzioni, fondamentali nel calcolo dell'angolo in tutti e quattro i quadranti, che sono:

- **`atan(x)`:** L'input  $x$  viene spesso ridotto all'intervallo  $[0, 1]$  sfruttando l'identità  $\text{atan}(x) = \pi/2 - \text{atan}(1/x)$  per  $x > 1$ , e  $\text{atan}(-x) = -\text{atan}(x)$ .

Successivamente si applica un'approssimazione polinomiale o razionale su questo intervallo ridotto.

- **atan2(y,x):** Questa funzione è intrinsecamente più complessa in quanto deve considerare i segni di  $y$  e  $x$  per determinare il quadrante corretto dell'angolo risultante, coprendo l'intero intervallo  $[-\pi, +\pi]$  (o  $[0, 2\pi]$ ). Le librerie matematiche la implementano con algoritmi specifici che gestiscono questa logica in modo efficiente, spesso evitando divisioni esplicite per  $x$  fino a quando non è strettamente necessario (e comunque proteggendosi dalla divisione per zero). Le implementazioni usano spesso approssimazioni polinomiali o razionali ottimizzate su vari settori del piano  $xy$ .

A livello hardware, MATLAB si appoggia alla *Floating-Point Unit* (FPU) del processore host, utilizzando per default la rappresentazione in virgola mobile a doppia precisione (IEEE 754 double, 64 bit). Questa scelta consente di ottenere un errore relativo inferiore a  $2.2 \times 10^{-16}$  nella maggior parte dei casi. L'elevata precisione, unita all'ottimizzazione dei percorsi numerici interni, rende MATLAB uno strumento affidabile per la validazione di algoritmi numerici, tra cui CORDIC [19].

La gestione interna degli errori in MATLAB è studiata per minimizzare l'accumulo numerico, mediante tecniche di controllo della propagazione degli errori di arrotondamento. Inoltre, le versioni più recenti adottano strategie di calcolo adattivo, che selezionano dinamicamente l'approccio più stabile a seconda dell'intervallo e della funzione utilizzata.

MATLAB rappresenta un ambiente di calcolo estremamente affidabile per la validazione numerica grazie all'integrazione di librerie ottimizzate, all'uso della doppia precisione floating-point e alla robustezza degli algoritmi matematici interni. I valori restituiti dalle sue funzioni trigonometriche sono stati utilizzati come riferimento teorico per la valutazione dell'accuratezza dell'algoritmo CORDIC implementato.

## 2.7 Implementazione dell'Algoritmo in MATLAB

Nel contesto dell'obiettivo principale di questa tesi, ovvero l'implementazione efficiente dell'algoritmo CORDIC su FPGA a supporto della stima dell'angolo di arrivo, l'implementazione e la simulazione in MATLAB rappresentano un passaggio fondamentale. Essa permette di verificare il funzionamento dell'algoritmo in un ambiente di alto livello e stabilire un riferimento di accuratezza e comportamento prima della sua realizzazione hardware. Questa sezione descrive in dettaglio il processo di implementazione del CORDIC in MATLAB per il calcolo di funzioni trigonometriche chiave: seno, arcoseno e arcotangente. La scelta di queste funzioni è strategica: l'arcotangente ( $\arctan(y/x)$ ) è direttamente impiegata nella fase finale di calcolo dell'angolo di incidenza nell'algoritmo di stima dell'AoA, mentre le funzioni seno e arcoseno sono utilizzate per dimostrare e validare la versatilità

del CORDIC nelle sue diverse modalità operative (rotazione e vettoriale), fornendo una verifica completa delle sue capacità computazionali prima del passaggio al dominio hardware.

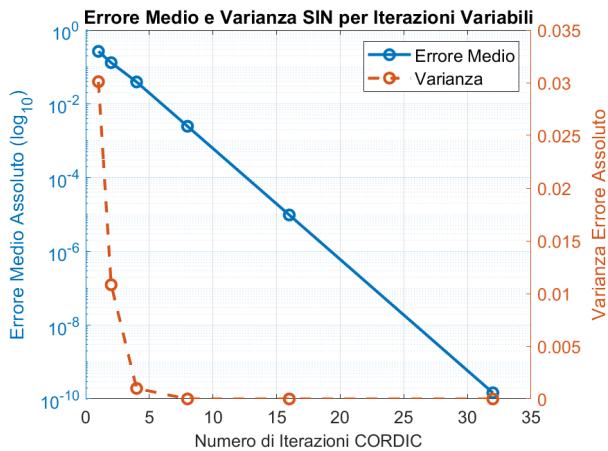
Per calcolare  $\sin \theta$ , si utilizza la modalità di rotazione del CORDIC. Si inizia con un vettore iniziale che ha componenti  $x_0 = 1$  e  $y_0 = 0$ , e un angolo accumulato  $z_0$  uguale a  $\theta$ , ad esempio  $\theta = 45^\circ$ , che in radianti è circa 0.7854. L'algoritmo esegue una serie di iterazioni, in questo caso 16, durante le quali il vettore viene ruotato di angoli elementari  $\arctan(2^{-i})$ . A ogni passo, le coordinate  $x$  e  $y$  vengono aggiornate, e l'angolo  $z$  viene ridotto in base alla direzione della rotazione, determinata dal segno di  $z$ . Dopo 16 iterazioni, i valori finali di  $x$  e  $y$  sono rispettivamente 0.607252 moltiplicato per  $\cos(0.7854)$  e 0.607252 moltiplicato per  $\sin(0.7854)$ . Poiché  $\sin(45^\circ)$  e  $\cos(45^\circ)$  sono entrambi circa 0.7071, i valori attesi dopo la compensazione del fattore di scala sono molto vicini a 0.7071, con un errore tipicamente inferiore a 0.001.

Per calcolare  $\arcsin x$ , si utilizza invece la modalità vettoriale. Si parte con un valore  $x$  compreso tra  $-1$  e  $1$ , ad esempio  $x = 0.5$ , e si inizializza un vettore con componenti  $x_0 = 0.5$  e  $y_0 = \sqrt{1 - 0.5^2}$ , che è circa 0.866. L'angolo accumulato  $z_0$  è inizialmente zero. L'algoritmo ruota il vettore iterativamente fino a portare  $y$  a zero, aggiornando  $z$  a ogni passo. Dopo 16 iterazioni,  $z$  rappresenta  $\arcsin(0.5)$ , che è circa 30 gradi, o 0.5236 radianti. Questo valore è molto vicino al risultato teorico, con un errore che si stabilizza a meno di 0.01 dopo un numero sufficiente di iterazioni.

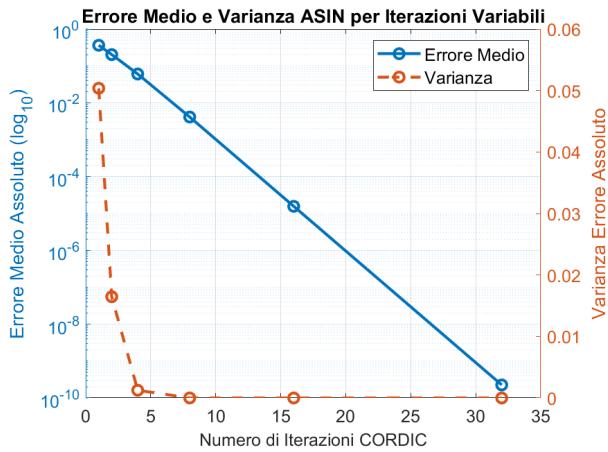
Il calcolo di  $\arctan(y/x)$  segue un approccio simile nella modalità vettoriale. Si inizia con un vettore con componenti  $x_0$  e  $y_0$ , ad esempio  $x_0 = 1$  e  $y_0 = 1$ , e un angolo accumulato  $z_0 = 0$ . L'algoritmo ruota il vettore fino a che  $y$  non diventa trascurabile, e al termine delle iterazioni,  $z$  rappresenta  $\arctan(1/1)$ , che è 45 gradi, o circa 0.7854 radianti. Questo calcolo è fondamentale per la stima dell'AoA, dove  $\arctan(y/x)$  rappresenta l'angolo di incidenza di un segnale.

La scelta della rappresentazione angolare è un altro aspetto importante. Gli angoli possono essere rappresentati in radianti, gradi o unità angolari CORDIC. Quest'ultima opzione è preferita nelle implementazioni hardware perché gli angoli elementari  $\arctan(2^{-i})$  sono normalizzati in modo che 1 unità angolare corrisponda a circa 1.743 radianti, ovvero  $99.9^\circ$ . Ad esempio, un angolo di 0.7854 radianti corrisponde a circa 0.45 unità angolari CORDIC. Questa rappresentazione elimina la necessità di conversioni durante le iterazioni, semplificando i calcoli interni. Tuttavia, per l'input e l'output, è necessario convertire gli angoli da e verso radianti o gradi, utilizzando un fattore di conversione di 1.743. Questo processo richiede attenzione per evitare errori di arrotondamento, specialmente quando si lavora con angoli vicini al limite del range.

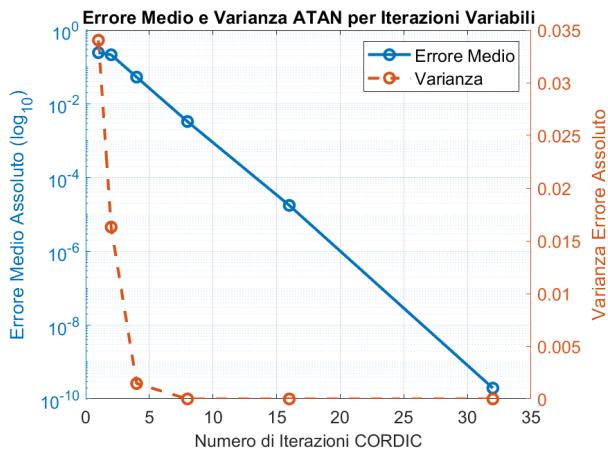
Per validare l'implementazione, sono stati condotti dei test approfonditi, i cui risultati sono rappresentati in una serie di Figure 2.1, 2.2 e 2.3 che riportano gli errori in rapporto al numero di iterazioni e al variare della LUT. Questi risultati offrono un quadro chiaro delle prestazioni del CORDIC, permettendo di valutare la precisione dell'algoritmo in diverse condizioni operative.



**Figura 2.1:** Errore medio e varianza del seno in funzione delle iterazioni.



**Figura 2.2:** Errore medio e varianza dell'arcoseno in funzione delle iterazioni.



**Figura 2.3:** Errore medio e varianza dell'arcotangente in funzione delle iterazioni.

Le figure 2.1, 2.2 e 2.3 presentano i risultati dell’analisi dell’errore medio assoluto e della varianza, in funzione del numero di iterazioni CORDIC, per le implementazioni delle funzioni trigonometriche SIN, ASIN e ATAN. Questi grafici sono stati generati al fine di comprendere la relazione tra il numero di passaggi iterativi dell’algoritmo CORDIC e la precisione raggiunta, oltre a valutare la stabilità dei risultati.

Ogni grafico mostra due curve principali su assi Y differenti:

- Errore medio (linea continua): Misura l’accuratezza media dell’algoritmo CORDIC rispetto a un riferimento ad alta precisione (le funzioni native di MATLAB in doppia precisione). L’asse Y di sinistra è in scala logaritmica, il che permette di visualizzare efficacemente la riduzione esponenziale dell’errore. Un valore di  $10^{-10}$  indica un errore di 0.0000000001, evidenziando una precisione elevatissima.
- Varianza (linea tratteggiata): Rappresenta la dispersione dei risultati ottenuti dall’algoritmo CORDIC. È un indicatore della stabilità e consistenza dell’algoritmo. L’asse Y di destra è in scala lineare.

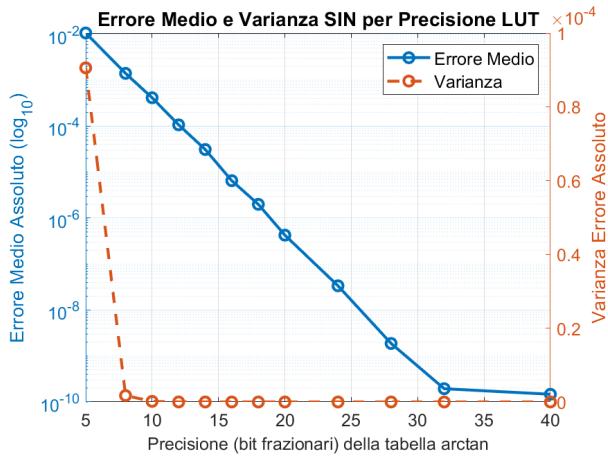
L’asse X per tutti e tre i grafici indica il numero di iterazioni, ovvero il numero di passi di rotazione eseguiti dall’algoritmo. La scelta di analizzare l’errore in funzione del numero di iterazioni è cruciale per la progettazione di sistemi che impiegano il CORDIC. L’algoritmo CORDIC è iterativo e la sua accuratezza è direttamente proporzionale al numero di iterazioni eseguite. Ogni iterazione contribuisce a raffinare l’approssimazione del risultato. Per ogni numero di iterazioni:

1. È stato generato un set esteso di valori di input. Specificamente, per le funzioni SIN e ATAN, gli angoli sono stati generati con un passo di un decimo di grado, coprendo l’intero range di interesse da  $0^\circ$  a  $360^\circ$ . Per la funzione ASIN, gli input sono stati generati nell’intervallo da  $-1$  a  $1$  con un passo di 0,01. Questo approccio garantisce una campionatura densa e rappresentativa dei domini delle funzioni.
2. Per ciascun input, il valore della funzione è stato calcolato sia tramite l’algoritmo CORDIC che tramite la funzione nativa di MATLAB
3. È stato calcolato l’errore assoluto tra i due risultati.
4. L’errore medio e la varianza di tutti gli errori assoluti sono stati quindi calcolati per quel particolare numero di iterazioni.

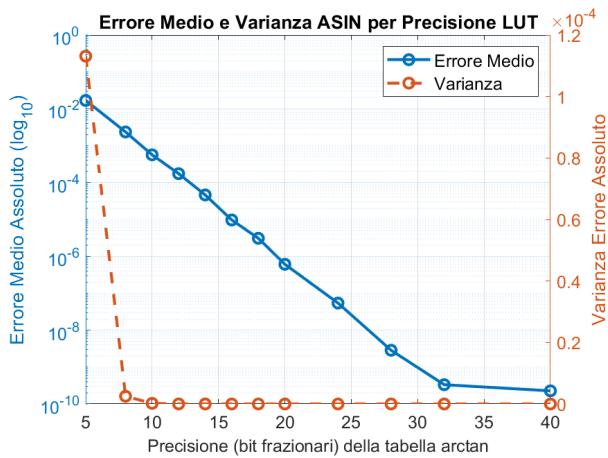
L’analisi condotta sul comportamento dell’errore medio e della varianza, in funzione del numero di iterazioni CORDIC, ha rivelato un aspetto cruciale: per tutte le funzioni esaminate, si osserva che intorno a 32 iterazioni l’algoritmo raggiunge un plateau di precisione e stabilità.

A questo punto, l’errore medio si stabilizza tipicamente nell’ordine di  $10^{-15}$  (apprrossimandosi all’epsilon di macchina per la doppia precisione), e la varianza scende a valori prossimi a  $10^{-30}$ , indicando una notevole stabilità dell’errore residuo.

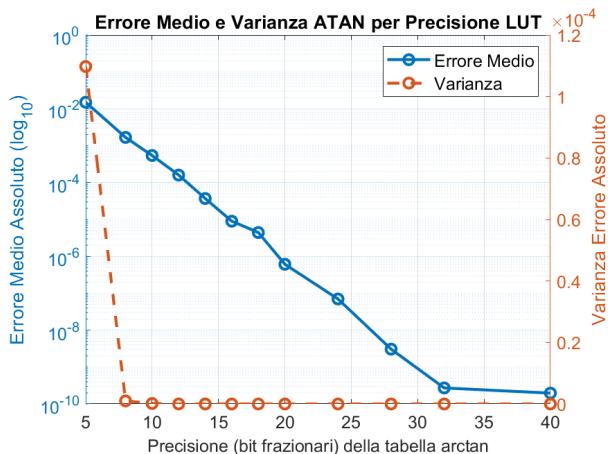
Dopo aver analizzato come il numero di iterazioni influenzi direttamente l'accuratezza e la stabilità dell'algoritmo CORDIC, è altrettanto fondamentale comprendere il ruolo giocato dalla precisione interna delle costanti utilizzate, come mostrato nelle Figure 2.4, 2.5 e 2.6. In particolare, la tabella delle arcotangenti, è essenziale per le rotazioni elementari del CORDIC e ha un impatto diretto sulla precisione finale raggiungibile.



**Figura 2.4:** Errore medio e varianza del seno in relazione alla precisione della tabella arctan.



**Figura 2.5:** Errore medio e varianza dell'arcoseno in relazione alla precisione della tabella arctan.



**Figura 2.6:** Errore medio e varianza dell’arcotangente in relazione alla precisione della tabella arctan.

Anche questi grafici, presentati nelle figure 2.4, 2.5 e 2.6 adottano la medesima struttura visiva dei precedenti per facilitare il confronto e la leggibilità: linee continue per l’errore medio e linee tratteggiate per la varianza.

La differenza sostanziale risiede nell’asse x. Mentre nei grafici precedenti l’asse x rappresentava il numero di iterazioni, qui indica la precisione (in termini di bit frazionari) della tabella delle arcotangenti pre-calcolata (LUT). Questa impostazione ci permette di esplorare direttamente come la granularità con cui vengono memorizzati gli angoli elementari, comuni a tutte le operazioni CORDIC, influenzino le prestazioni. Le curve di errore medio e varianza sono state generate per ciascuna delle funzioni (sin, asin e atan) in relazione a questa precisione variabile della LUT. Un numero maggiore di bit frazionari si traduce in una rappresentazione più fedele e accurata di tali angoli.

La scelta di investigare l’influenza della precisione della tabella arctan è cruciale. La metodologia CORDIC si basa sulla somma di angoli elementari, e l’accuratezza con cui questi angoli sono pre-calcolati e memorizzati nella tabella ha un impatto diretto sulla fedeltà delle rotazioni successive. Se gli angoli della tabella non sono sufficientemente precisi, l’algoritmo non potrà mai raggiungere la sua massima accuratezza potenziale, indipendentemente dal numero di iterazioni.

Per generare i dati illustrati in questi grafici, è stato eseguito un test dove la precisione della tabella arctan è stata variata, tipicamente da 5 bit frazionari fino a 40 bit frazionari. Per ciascun livello di precisione della tabella: è stata ricreata la tabella arctan con la precisione specificata (ad esempio, troncando o arrotondando i valori a un certo numero di bit frazionari simulando un ambiente a virgola fissa). L’algoritmo CORDIC è stato eseguito per ciascuna funzione (SIN, ASIN, ATAN) utilizzando questa nuova tabella, mantenendo fisso il numero di iterazioni a 32. Questa scelta è stata fatta per isolare l’effetto della sola precisione della tabella arctan.

Sono stati utilizzati gli stessi set di input descritti in precedenza e per ogni input, è stato calcolato l’errore assoluto rispetto al risultato di MATLAB.

Infine, l'errore medio e la varianza di tutti gli errori assoluti sono stati quindi calcolati per quel livello di precisione della tabella arctan.

Per tutte e tre le funzioni CORDIC considerate, osserviamo un andamento molto specifico dell'errore medio: una diminuzione lineare sulla scala logaritmica man mano che la precisione (cioè il numero di bit frazionari) della singola tabella delle arcotangenti aumenta. Questa tabella, contenente i valori  $\text{arctan}(2^{-i})$  essenziali per le rotazioni elementari, è comune a tutte le implementazioni CORDIC. Questo comportamento è esattamente ciò che ci si aspetta: una maggiore precisione nella memorizzazione dei valori  $\text{arctan}(2^{-i})$  all'interno di tale tabella significa che gli angoli elementari, fondamentali per le rotazioni dell'algoritmo CORDIC, sono rappresentati con maggiore fedeltà. Angoli più accurati si traducono in rotazioni più precise che, a loro volta, permettono all'algoritmo di convergere verso un'approssimazione del risultato desiderato sempre più vicina al valore reale, migliorando così le prestazioni per ciascuna delle funzioni implementate.

Nello specifico, notiamo che l'errore subisce una riduzione drasticamente rapida nelle fasi iniziali, per poi stabilizzarsi su valori estremamente esigui. Questi valori si attestano nell'ordine di  $10^{-9}$  o  $10^{-10}$ , una volta che la precisione della tabella supera un certo numero di bit, indicativamente tra 30 e 35 bit frazionari. Questa stabilizzazione è un indicatore significativo: suggerisce che, al di là di una determinata soglia di precisione della tabella, ulteriori incrementi non portano a miglioramenti apprezzabili nell'accuratezza complessiva. A quel punto, sono altri fattori, come la precisione intrinseca dei calcoli in virgola mobile (floating-point) del sistema che esegue l'algoritmo, a diventare il limite dominante, e non più la precisione dei valori pre-calcolati.

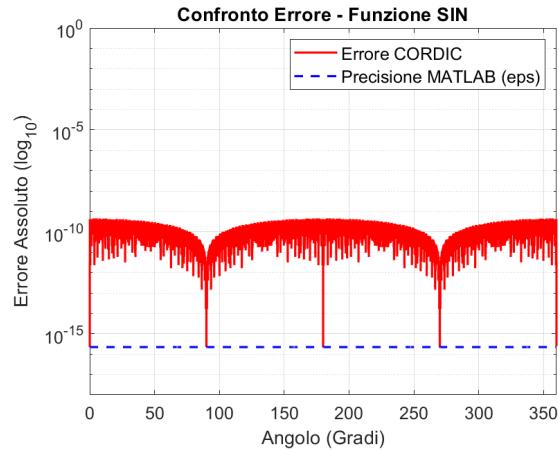
Parallelamente alla riduzione dell'errore, l'analisi della varianza dell'errore per tutte le funzioni rivela un decadimento estremamente rapido fin dalle prime fasi di aumento della precisione della tabella. Già con un numero relativamente modesto di bit frazionari, nell'intervallo di 10-15 bit, la varianza tende ad avvicinarsi a valori prossimi allo zero o addirittura nulli.

Questo comportamento indica che l'algoritmo CORDIC acquisisce rapidamente una notevole stabilità e consistenza nei suoi risultati, a patto che la tabella arctan sia dotata di una precisione adeguata. Una maggiore accuratezza negli angoli elementari riduce intrinsecamente la variabilità e le fluttuazioni nelle successive operazioni di rotazione, culminando in risultati che sono non solo più precisi, ma anche significativamente più prevedibili e riproducibili.

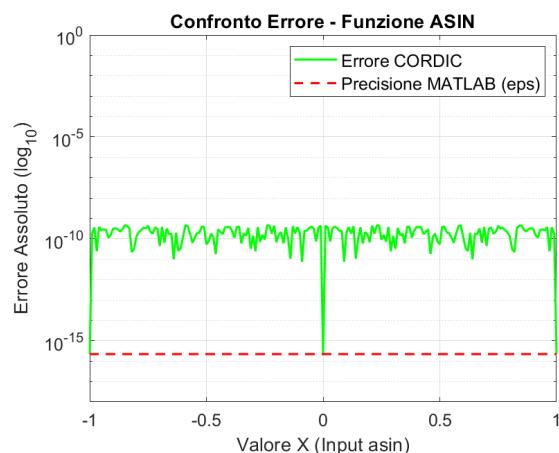
I grafici offrono una chiara evidenza dell'importanza critica della precisione della tabella arctan per le performance complessive dell'algoritmo CORDIC. Essi dimostrano inequivocabilmente come sia l'accuratezza che la stabilità migliorino in modo marcato all'aumentare della precisione della tabella, fino a raggiungere un limite imposto dalle capacità intrinseche di precisione di macchina dell'ambiente di calcolo.

Dopo aver esplorato come le iterazioni e la precisione della tabella arctan influenzino le prestazioni dell'algoritmo CORDIC, è ora essenziale valutare la sua accuratezza finale rispetto a un riferimento di alta precisione. Questo ci permette di capire quanto l'implementazione del CORDIC si avvicini ai limiti teorici di accu-

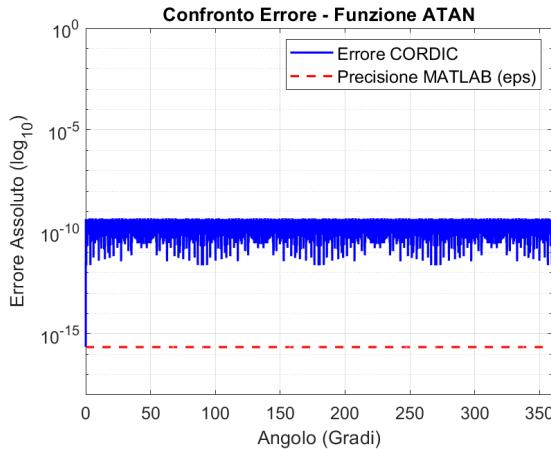
ratezza raggiungibili in un ambiente di calcolo a doppia precisione. A tal fine, le Figure 2.7, 2.8 e 2.9 presentano il confronto diretto dell'errore assoluto delle funzioni calcolate con l'algoritmo CORDIC rispetto alla precisione di macchina (eps) di MATLAB.



**Figura 2.7:** Errore medio e varianza del seno in relazione alla precisione della tabella arctan.



**Figura 2.8:** Errore medio e varianza dell'arcoseno in relazione alla precisione della tabella arctan.



**Figura 2.9:** Errore medio e varianza dell'arcotangente in relazione alla precisione della tabella arctan.

Questi grafici condividono una struttura visuale chiara, volta a mettere in risalto il confronto delle precisioni:

- Asse X: Rappresenta i valori di input delle funzioni.
- Asse Y: Misura la differenza assoluta tra il risultato del CORDIC e il valore di riferimento di MATLAB, visualizzata in scala logaritmica.
- Linea continua: Indica l'errore assoluto dell'algoritmo CORDIC per ogni punto di input.
- Rappresenta la precisione di macchina standard di MATLAB per i numeri in doppia precisione, ovvero il più piccolo numero tale che  $1+\text{eps} > 1$ . Questo valore è il limite inferiore della precisione per i calcoli in virgola mobile e serve da riferimento per stabilire se l'errore del CORDIC è prossimo al limite teorico imposto dall'architettura del calcolo.

L'obiettivo primario di questi grafici è validare l'accuratezza del CORDIC nel contesto di un ambiente di calcolo standard. Confrontare l'errore del CORDIC con la precisione di MATLAB ( $\text{eps}$ ) ci permette di determinare se l'algoritmo è in grado di raggiungere una precisione paragonabile a quella delle funzioni native di un software di calcolo numerico ad alta performance, che spesso sfrutta l'hardware del processore per la massima efficienza e accuratezza in doppia precisione.

In questo test, il numero di iterazioni dell'algoritmo CORDIC è stato fissato a 32 per tutte le funzioni, dato che i precedenti risultati hanno mostrato come questo valore garantisca una convergenza e stabilità ottimali dell'algoritmo, sono stati utilizzati gli stessi set di input descritti in precedenza angoli con passo un decimo di angolo. Successivamente, è stato calcolato l'errore assoluto tra i due risultati. Questo errore è stato quindi confrontato con il valore di  $\text{eps}$  di MATLAB. Approfondendo l'analisi delle singole funzioni, possiamo apprezzare delle sfumature particolarmente interessanti nel modo in cui l'errore si manifesta.

Per quanto riguarda la funzione sin, i grafici **mostrano** che la linea dell'errore non è perfettamente piatta, ma presenta picchi di precisione. Questi si manifestano in corrispondenza di angoli ben specifici, come  $90^\circ$ ,  $180^\circ$ ,  $270^\circ$  e  $360^\circ$ . Questa tendenza è spesso riconducibile alla natura intrinsecamente periodica della funzione e al modo peculiare con cui l'algoritmo CORDIC ne esegue i calcoli. Il risultato è una naturale e attesa riduzione degli errori proprio in prossimità dei punti di simmetria o dei multipli di  $\pi/2$ .

Passando alla funzione ASIN, osserviamo che l'errore si distribuisce in maniera relativamente più uniforme lungo l'intero intervallo di input da  $-1$  a  $1$ . Nonostante questa generalizzata uniformità, non mancano occasionali picchi di precisione, che indicano momenti di maggiore o minore scostamento dal valore ideale.

Infine, anche per la funzione ATAN, l'errore si mantiene generalmente stabile e a livelli contenuti lungo tutto il dominio. Tuttavia, è possibile individuare dei punti precisi in cui l'errore sperimenta un calo ancora più drastico, avvicinandosi in maniera notevole alla precisione di macchina. Questi punti di eccezionale accuratezza corrispondono spesso ad angoli esatti, come  $0^\circ$ ,  $45^\circ$ ,  $90^\circ$  e le loro simmetrie o multipli. Questo fenomeno suggerisce che, in corrispondenza di questi particolari valori di calcolo, l'algoritmo CORDIC è in grado di raggiungere una precisione che sfiora, e talvolta eguaglia, il limite teorico imposto dal sistema.

Per tutte e tre le funzioni analizzate, emerge chiaramente un dato molto significativo: l'errore assoluto dell'algoritmo CORDIC, rappresentato dalla linea continua nei grafici, si mantiene costantemente su valori eccezionalmente bassi. Ci riferiamo a errori che si aggirano tipicamente nell'ordine di  $10^{-9}$  o  $10^{-10}$ . Questo risultato è una dimostrazione tangibile dell'altissima precisione che il nostro algoritmo CORDIC è in grado di raggiungere. Tale livello di accuratezza conferma in modo inequivocabile la sua idoneità per applicazioni che non solo richiedono elevata precisione numerica, ma anche affidabilità nei risultati.

Un aspetto cruciale in questi grafici è il confronto tra la linea dell'errore CORDIC e la precisione di macchina di MATLAB. È evidente che l'errore del nostro CORDIC si posiziona costantemente molto vicino o addirittura al di sopra di questa soglia, che tipicamente si aggira intorno a  $10^{-16}$  per la doppia precisione. Ci indica che l'errore del nostro algoritmo non è dovuto a difetti intrinseci o a limiti computazionali del CORDIC stesso. Piuttosto, l'accuratezza finale è limitata principalmente dalla precisione floating-point intrinseca del sistema di calcolo su cui stiamo operando. In altre parole, una volta che abbiamo garantito un numero sufficiente di iterazioni e una tabella arctan con la precisione adeguata, l'algoritmo CORDIC sta operando ai limiti stessi della precisione numerica disponibile sulla macchina.

I risultati finora discussi, ottenuti in ambiente MATLAB, hanno dimostrato l'elevata accuratezza che l'algoritmo CORDIC può raggiungere operando con la precisione in virgola mobile (floating-point) standard. Tuttavia, quando si mira a un'implementazione efficiente su piattaforme hardware specifiche, come le FPGA, la scelta della rappresentazione numerica diventa un fattore critico. L'utilizzo della virgola mobile, sebbene offra un'ampia dinamica e precisione, richiede risorse

hardware considerevoli (es. Floating-Point Units, FPU) che possono tradursi in costi elevati, maggiore consumo energetico e maggiore latenza.

Per superare queste sfide, soprattutto in contesti embedded e a basso consumo, si ricorre spesso alla rappresentazione in virgola fissa (Fixed-Point).

## 2.8 Che cos'è la Rappresentazione in Fixed-Point

La rappresentazione in Fixed-Point è un modo per memorizzare numeri reali utilizzando un numero intero di bit, dove la posizione del punto decimale è implicita e fissa. A differenza della virgola mobile (float), dove il punto decimale è mobile e la precisione varia in base alla grandezza del numero, nella virgola fissa il numero di bit dedicati alla parte intera e alla parte frazionaria è predefinito.

Un numero in Fixed-Point è tipicamente descritto da una notazione  $Qm.n$ , dove:

- $Q$  sta per "Q-format".
- $m$  è il numero di bit dedicati alla parte intera (incluso il bit di segno, se il numero è con segno).
- $n$  è il numero di bit dedicati alla parte frazionaria (la precisione dopo il punto).
- Il numero totale di bit è  $m + n$ .

Ad esempio, un numero in formato  $Q4.12$  utilizza 4 bit per la parte intera (che include il segno) e 12 bit per la parte frazionaria, per un totale di 16 bit. La gamma di valori rappresentabili e la precisione dipendono direttamente dalla scelta di  $m$  e  $n$ . In questo caso, con una configurazione di 16 bit totali e 12 bit frazionari, si hanno 4 bit per la parte intera e 12 bit per la parte frazionaria. La risoluzione minima, o passo di quantizzazione, è determinata dal bit meno significativo della parte frazionaria, ed è pari a  $2^{-12}$ , ovvero circa 0.000244. Questo significa che ogni numero reale viene arrotondato al multiplo più vicino di 0.000244, introducendo un errore di quantizzazione. La parte intera, con 4 bit, consente di rappresentare numeri nell'intervallo da  $-8$  a  $+7.999755$  (considerando il bit di segno), che è sufficiente per i valori tipicamente coinvolti nei calcoli del CORDIC, come le coordinate  $x$ ,  $y$  e l'angolo accumulato  $z$ , che raramente superano tali limiti [17].

L'adozione della rappresentazione Fixed-Point nelle implementazioni hardware, specialmente per algoritmi come il CORDIC, offre diversi vantaggi sostanziali:

1. **Efficienza Computazionale e Minore Complessità Hardware:** Le operazioni in Fixed-Point (addizioni, sottrazioni, moltiplicazioni, shift) possono essere implementate utilizzando circuiti logici molto più semplici e meno dispendiosi rispetto alle equivalenti operazioni in virgola mobile. Non è necessaria una FPU complessa, il che riduce drasticamente l'utilizzo di risorse (ad esempio, LUTs e Flip-Flops su FPGA) e la complessità del design.

2. Minore Consumo Energetico: Circuiti più semplici si traducono direttamente in un minore consumo di potenza. Questo è un fattore critico per dispositivi a batteria, sensori IoT e sistemi embedded dove l'autonomia energetica è fondamentale.
3. Maggiore Velocità (Latenza Inferiore): Le operazioni Fixed-Point hanno tipicamente una latenza inferiore rispetto a quelle Floating-Point, in quanto non richiedono la normalizzazione, l'allineamento degli esponenti o altre complesse manipolazioni di bit. Questo è vitale per applicazioni in tempo reale dove la velocità di elaborazione è prioritaria.
4. Controllo Esplicito della Precisione e del Range: La scelta del formato  $Qm.n$  offre al progettista un controllo diretto sulla precisione e sul range di valori rappresentabili. Questo permette di ottimizzare la rappresentazione numerica per le specifiche esigenze dell'algoritmo e dell'applicazione, minimizzando gli sprechi di risorse e massimizzando l'accuratezza per il range di valori di interesse [20].
5. Prevedibilità del Comportamento dell'Errore: La natura fissa del punto decimale rende più prevedibile il comportamento degli errori di quantizzazione e di troncamento. Sebbene sia necessario prestare attenzione all'overflow e alla propagazione degli errori, il loro impatto può essere analizzato e gestito con maggiore determinazione rispetto alla virgola mobile.

Nonostante i suoi vantaggi, la transizione dalla virgola mobile alla virgola fissa non è priva di sfide. La principale risiede nella gestione dell'errore di quantizzazione e dell'overflow. Poiché la precisione è limitata e il range è fisso, è fondamentale scegliere un formato  $Qm.n$  che sia sufficientemente ampio da coprire tutti i valori intermedi del calcolo senza overflow e sufficientemente preciso da mantenere l'errore finale entro limiti accettabili. Questo spesso richiede un'attenta analisi a priori del range dinamico dei segnali e dei risultati intermedi, nonché simulazioni dettagliate per valutare l'impatto della quantizzazione sull'accuratezza complessiva del sistema [21].

Nel contesto del CORDIC, la conversione a Fixed-Point implica che anche i valori della tabella ‘arctan’ e i fattori di scala  $K_n$  debbano essere rappresentati in Fixed-Point, introducendo ulteriori fonti di errore di quantizzazione che devono essere bilanciate con la precisione desiderata.

## 2.9 Processo di Conversione in Fixed-Point

L'adattamento del codice MATLAB del CORDIC per operare con variabili in rappresentazione fixed-point richiede diversi passaggi, ciascuno dei quali è essenziale per garantire che l'algoritmo mantenga un'adeguata precisione una volta implementato su hardware. Il processo inizia con la definizione delle variabili che devono essere convertite: nel caso del CORDIC, queste includono le coordinate  $x$  e  $y$ , l'angolo accumulato  $z$ , e gli angoli elementari  $\arctan(2^{-i})$  utilizzati in ogni iterazione. In MATLAB, queste variabili sono inizialmente rappresentate in formato floating-point a doppia precisione, che offre un'alta precisione e un ampio range. Tuttavia, su una FPGA, queste variabili devono essere mappate su un numero limitato di bit in formato fixed-point.

Come già discusso, i parametri *wordlength* e *fractionlength* sono utilizzati per definire la configurazione da utilizzare. Tuttavia, è fondamentale selezionarli con attenzione, poiché una configurazione con una word length troppo ridotta, ad esempio pari a 8 bit, potrebbe non fornire un range sufficiente per rappresentare i valori del CORDIC, portando a overflow, o una precisione adeguata, causando errori significativi. D'altra parte, un wordlength troppo alto, come 32 bit, aumenta inutilmente il consumo di risorse su FPGA, rallentando i calcoli e occupando più logica programmabile. Il *fractionlength*, invece, deve essere scelto in base alla precisione richiesta: un numero troppo basso di bit frazionari, ad esempio 4, porterebbe a una risoluzione grossolana (passo di quantizzazione di  $2^{-4} = 0.0625$ ), mentre un numero troppo alto riduce il range della parte intera, rischiando overflow [17].

Per effettuare la conversione, MATLAB offre strumenti come il Fixed-Point Designer [19], che consente di definire tipi di dati fixed-point per le variabili e di simulare il comportamento dell'algoritmo con tale rappresentazione. Ad esempio, una variabile come  $x$  può essere definita con il tipo `fi(x, 1, 16, 12)`, dove il primo argomento è il valore iniziale, il secondo indica che è un numero con segno (1 per signed, 0 per unsigned), il terzo è il wordlength (16 bit), e il quarto è il fractionlength (12 bit). Questo tipo di dato forza MATLAB a quantizzare il valore di  $x$  secondo la risoluzione definita, arrotondandolo al multiplo più vicino di  $2^{-12}$ . Durante le iterazioni del CORDIC, tutte le operazioni aritmetiche (addizioni, sottrazioni e shift) vengono eseguite rispettando questa rappresentazione, e MATLAB tiene traccia degli errori di quantizzazione e degli eventuali overflow.

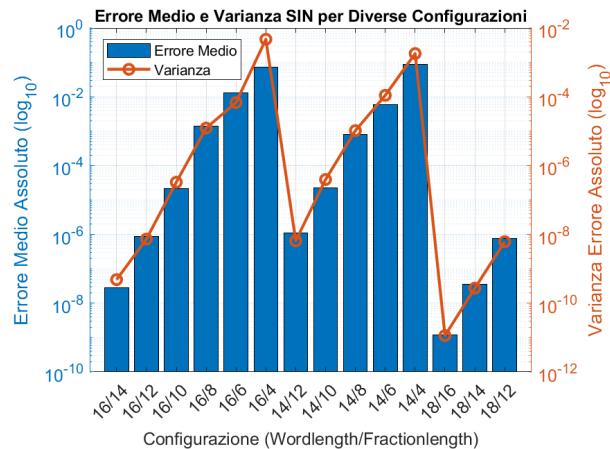
Un aspetto critico della conversione è la gestione degli angoli elementari  $\arctan(2^{-i})$ . Questi valori, che in floating-point sono numeri reali come  $\arctan(2^{-0}) = 0.785398$  (circa 45 gradi), devono essere quantizzati nella stessa rappresentazione fixed-point. Ad esempio, con 16 bit e 12 bit frazionari, il valore 0.785398 viene arrotondato a 0.785400, il multiplo più vicino di  $2^{-12}$ . Questo introduce un piccolo errore iniziale, che si accumula durante le iterazioni. Inoltre, l'angolo accumulato  $z$  e le coordinate  $x$  e  $y$  subiscono errori di arrotondamento a ogni passo, poiché ogni operazione (ad esempio  $x_{i+1} = x_i - d_i \cdot y_i \cdot 2^{-i}$ ) produce un risultato che deve essere nuovamente quantizzato. Per minimizzare questi errori, è necessario testare

diverse configurazioni di wordlength e fractionlength, analizzando l'impatto sulla precisione finale dell'algoritmo [17].

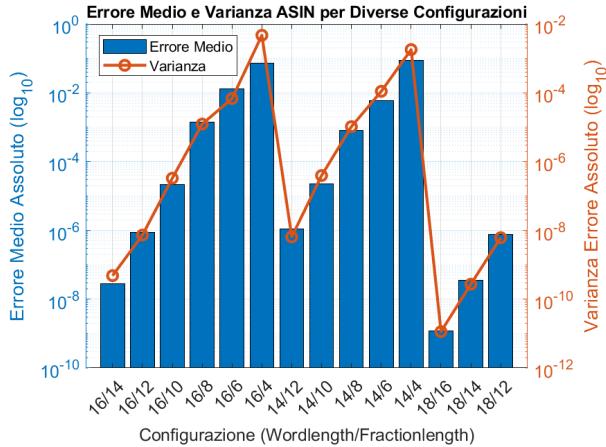
Un altro aspetto importante è la rappresentazione degli angoli. Nel CORDIC, gli angoli possono essere espressi in radianti, gradi o unità angolari CORDIC. Quest'ultima rappresentazione è particolarmente utile in fixed-point, poiché normalizza gli angoli elementari  $\arctan(2^{-i})$  in modo che 1 unità angolare corrisponda a circa 1.743 radianti (99.9 gradi), il limite di convergenza del CORDIC. Ad esempio, un angolo di 0.7854 radianti (45 gradi) corrisponde a circa 0.45 unità angolari CORDIC. Questa normalizzazione elimina la necessità di conversioni durante le iterazioni, semplificando i calcoli e riducendo gli errori di quantizzazione. Tuttavia, per l'input e l'output, è necessario convertire gli angoli da radianti o gradi a unità angolari CORDIC, utilizzando un fattore di conversione di 1.743, e questa conversione deve essere anch'essa quantizzata in fixed-point [15].

## 2.10 Scelta della Configurazione Ottimale

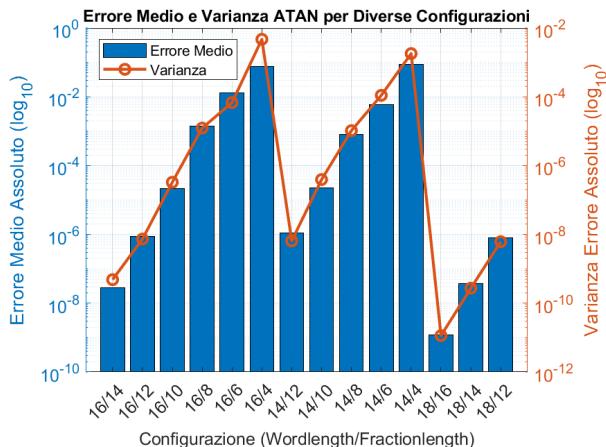
La scelta della configurazione ottimale di wordlength e fractionlength è un passaggio fondamentale per garantire che l'implementazione fixed-point del CORDIC sia efficiente e precisa. Per determinare la configurazione migliore, sono stati condotti test sistematici in MATLAB, simulando l'algoritmo con diverse combinazioni di wordlength e fractionlength. L'obiettivo era minimizzare l'errore medio totale rispetto ai risultati ottenuti con la rappresentazione floating-point, che funge da riferimento ideale. I risultati di questi test sono stati rappresentati graficamente, come mostrano le Figure 2.10, 2.11 e 2.12, permettendo di identificare la configurazione che offre il miglior compromesso tra precisione e utilizzo di risorse.



**Figura 2.10:** Errore medio e varianza del seno in rapporto alla configurazione usata.



**Figura 2.11:** Errore medio e varianza dell’arcoseno in rapporto alla configurazione usata..



**Figura 2.12:** Errore medio e varianza dell’arcotangente in rapporto alla configurazione usata.

Le Figure 2.10, 2.11 e 2.12 attraverso una rappresentazione visiva che combina istogrammi per l’errore medio assoluto e linee che indicano la varianza dell’errore, e sfruttando una scala logaritmica sull’asse verticale, sono in grado di percepire differenze significative negli ordini di grandezza, cruciali per comprendere il comportamento di questi sistemi.

L’analisi dei grafici mostra che, per tutte e tre le funzioni, l’incremento della *fractionlength* comporta una significativa riduzione dell’errore medio assoluto. Questo fenomeno non è casuale, ma è intrinseco alla natura stessa dell’aritmetica Fixed-Point. Ogni bit aggiuntivo dedicato alla parte frazionaria di un numero non è un semplice incremento, ma un vero e proprio affinamento della risoluzione. Ciò si traduce in una drastica riduzione degli errori di quantizzazione che inevitabilmente si insinuano durante le complesse operazioni dell’algoritmo CORDIC come somme, shift o consultazioni di lookup table. Il risultato è che i valori calcolati si

avvicinano sempre di più al loro corrispettivo ideale in floating-point. Per fare un esempio tangibile, le configurazioni che vantano una FL più elevata, come 16/14 o 18/16, mostrano errori medi che si attestano tra  $10^{-6}$  e  $10^{-10}$ , un'accuratezza notevole. Al contrario, quelle con una FL inferiore, come 16/4 o 14/4, pur essendo funzionali, presentano errori nell'ordine di  $10^{-1}$  o  $10^{-2}$ . Questo divario è la prova lampante dell'influenza diretta della precisione scelta.

Parallelamente all'errore medio, anche la varianza dell'errore segue una traiettoria discendente all'aumentare della *fractionlength*. Una varianza ridotta è un segnale estremamente positivo: indica che i risultati prodotti dall'algoritmo non sono solo accurati in media, ma sono anche straordinariamente consistenti e meno dispersi attorno al loro valore centrale. Questa stabilità è un pilastro per la robustezza di qualsiasi sistema digitale, garantendo che l'accuratezza raggiunta non sia un evento isolato, ma una performance stabile e prevedibile su un ampio spettro di input. Le configurazioni che investono in una maggiore precisione non si limitano a minimizzare l'errore, ma blindano anche l'affidabilità e la ripetibilità dei calcoli.

La decisione di rappresentare sia l'errore medio che la varianza su un asse Y in scala logaritmica ( $\log_{10}$ ) non è puramente estetica, ma profondamente funzionale. Questa scelta è indispensabile quando si analizzano fenomeni che abbracciano diversi ordini di grandezza, una caratteristica distintiva della precisione numerica. Essa ci permette di apprezzare visivamente l'impressionante miglioramento esponenziale dell'accuratezza mano a mano che la *fractionlength* aumenta. È particolarmente interessante notare come la varianza si mantenga, in generale, uno o due ordini di grandezza al di sotto dell'errore medio per la medesima configurazione, specialmente nelle configurazioni più precise. Questo comportamento è del tutto in linea con le aspettative, poiché la varianza, misurando la dispersione, è logicamente influenzata dalla stessa riduzione degli errori che porta alla diminuzione della media.



# Capitolo 3

## IMPLEMENTAZIONE DELL'ALGORITMO CORDIC IN VHDL

Questo capitolo si propone di esplorare in maniera estensiva e, dove opportuno, critica, gli strumenti e le metodologie che rappresentano la spina dorsale dell'approccio Model-Based Design nel contesto della progettazione hardware. L'ecosistema MathWorks, pur offrendo soluzioni indubbiamente potenti e innovative, presenta anche sfide e limitazioni che meritano un'analisi ponderata. Saranno quindi dissezionati HDL Coder, HDL Verifier, HDL Workflow Advisor e la metodologia FPGA-in-the-Loop (FIL), approfondendone non solo la definizione, il principio di funzionamento, la struttura interna e le finalità d'uso, ma anche i compromessi intrinseci, le complessità nascoste e le situazioni in cui alternative potrebbero rivelarsi più adatte [19]. L'obiettivo è fornire una comprensione bilanciata, che evidenzi sia i punti di forza che le aree di potenziale frizione, essenziale per un'applicazione consapevole e strategica di queste tecnologie nel campo dell'ingegneria digitale.

### 3.1 MATLAB HDL Coder

HDL Coder è un'innovativa toolbox di MathWorks che si prefigge l'ambizioso compito di automatizzare la traduzione di algoritmi complessi, sviluppati e verificati in MATLAB e Simulink, in codice Hardware Description Language (HDL) sintetizzabile, supportando gli standard Verilog, SystemVerilog e VHDL. La sua ragion d'essere risiede nella promessa di elevare il livello di astrazione del design hardware, permettendo di concentrarsi sulla logica algoritmica e le specifiche di sistema, piuttosto che sulle minuzie della codifica RTL. Se da un lato ciò accelera il processo, dall'altro introduce una dipendenza da un motore di generazione il cui funzionamento interno, e le relative scelte architetturali, non sono sempre trasparenti all'utente finale.

Il funzionamento di HDL Coder si articola attraverso un'analisi approfondita del modello di input e l'applicazione di sofisticate tecniche di traduzione e ottimizzazione. Tuttavia, è fondamentale comprendere che, pur automatizzando il

processo, le scelte algoritmiche a monte in MATLAB influenzano profondamente l'efficienza e la qualità dell'HDL generato [19].

Il processo di generazione del codice viene diviso in diversi punti, che sono:

1. Analisi del Modello di Input: HDL Coder effettua una scansione dettagliata del modello Simulink o della funzione MATLAB per decifrarne la logica computazionale, i tipi di dato impiegati (floating-point, fixed-point o misti), le dipendenze funzionali e i vincoli temporali. Sebbene il tool supporti un'ampia gamma di blocchi e funzioni, la compatibilità non è universale, e l'adozione di blocchi non "HDL-ready" può spesso portare a errori di compatibilità o a un codice HDL inefficiente.
2. Conversione a Fixed-Point: La conversione da floating-point a fixed-point è un passaggio cruciale per l'implementazione hardware, ma è anche una delle maggiori fonti di potenziale errore e compromesso. HDL Coder può assistere in questo processo, ma la scelta dei parametri di wordlength e fractionlength (ad esempio, il formato  $Qm.n$ ) rimane una decisione critica che impatta direttamente la precisione e il range di valori, nonché l'utilizzo delle risorse hardware. Una scelta sub-ottimale può facilmente portare a overflow, underflow o a una riduzione inaccettabile della precisione numerica, richiedendo un'accurata analisi a priori del range dinamico dei segnali e lunghe simulazioni di validazione [17].
3. Mappatura e Ottimizzazione Hardware: Questa è la fase in cui HDL Coder rivela la sua intelligenza, ma anche le sue potenziali rigidità. Applica diverse tecniche di ottimizzazione, ma la loro efficacia può variare in base alla specificità dell'algoritmo e alla complessità della logica.
  - Pipelining: L'inserimento di stadi di pipeline è essenziale per aumentare la frequenza operativa (Fmax). Tuttavia, un pipelining troppo aggressivo può aumentare significativamente la latenza complessiva del design e l'utilizzo di Flip-Flops, introducendo complessità aggiuntive nella gestione del flusso dati e nella validazione.
  - Condivisione di Risorse: Rilevare e condividere blocchi hardware identici è una tecnica di ottimizzazione dell'area. Tuttavia, questa condivisione può talvolta ridurre la throughput del sistema se le operazioni condivise si trovano su percorsi critici non parallelizzabili.
  - Streamlining/Serialization: La conversione di elaborazioni parallele in sequenziali può ridurre drasticamente l'area, ma comporta un inevitabile aumento della latenza e una diminuzione della throughput. La decisione tra parallelismo e serializzazione è spesso un compromesso difficile.
  - Bilanciamento del Ritardo: Essenziale dopo il pipelining, ma può aggiungere complessità e un overhead di registri, aumentando l'area.
  - Gestione degli Stati: La traduzione di macchine a stati Stateflow è efficiente, ma per macchine a stati molto complesse o con requisiti di

tempistica stringenti, la codifica manuale di HDL potrebbe offrire un controllo più granulare.

- **Interfacciamento:** Sebbene la generazione delle interfacce di I/O sia automatizzata, la configurazione precisa delle porte e dei protocolli può richiedere un'attenta conoscenza delle specifiche hardware.
4. **Generazione del Codice HDL:** L'output è codice HDL leggibile e commentato. Tuttavia, la leggibilità e l'aderenza a specifici stili di codifica aziendali possono non essere sempre perfette. Il codice generato, pur essendo sintetizzabile, non sempre eguaglia in efficienza (area e velocità) il codice scritto manualmente da un esperto HDL che può sfruttare ottimizzazioni specifiche per la micro-architettura target.
  5. **Reportistica:** I report HTML generati sono preziosi, ma le stime di risorse e tempistiche sono basate su modelli generici e possono differire dai risultati finali post-sintesi e post-place-and-route, specialmente per design complessi o per tecnologie FPGA all'avanguardia.

### 3.1.1 Struttura di HDL Coder

L'efficacia di HDL Coder nell'ambiente di progettazione hardware derivano dalla sua natura non di applicazione monolitica isolata, ma di un ecosistema complesso e profondamente integrato all'interno dell'ambiente MATLAB e Simulink. Questa integrazione, pur essendo un punto di forza, introduce anche sfumature e potenziali attriti che meritano un'analisi approfondita per una comprensione completa delle sue implicazioni pratiche.

Al suo cuore, HDL Coder si manifesta attraverso le sue librerie di blocchi HDL-Ready in Simulink. Queste librerie offrono una serie di blocchi pre-certificati e ottimizzati che sono stati specificamente progettati per una traduzione efficiente in codice HDL. Sebbene la loro presenza semplifichi notevolmente il processo di modellazione per la sintesi hardware, è importante riconoscere che la loro adozione può, in alcuni casi, vincolare il designer a specifici pattern di modellazione. Questo può limitare la flessibilità creativa e la capacità di implementare soluzioni architettoniche non convenzionali o altamente personalizzate che potrebbero essere ottimali per un'applicazione di nicchia. La necessità di aderire a un set predefinito di blocchi può talvolta ostacolare l'esplorazione di architetture innovative.

Parallelamente, l'interazione con HDL Coder può avvenire tramite funzioni e classi MATLAB. Questo fornisce un'interfaccia programmabile, essenziale per l'automazione di workflow complessi o per l'integrazione in pipeline di Continuous Integration/Continuous Deployment (CI/CD). Tuttavia, per sfruttare appieno queste capacità, è richiesta una solida familiarità non solo con la programmazione MATLAB, ma anche con le API specifiche di HDL Coder [19].

Un aspetto cruciale della struttura di HDL Coder risiede nelle sue interfacce con strumenti EDA (Electronic Design Automation) di terze parti. La capacità di interagire fluidamente con software di sintesi (come Xilinx Vivado, Intel

Quartus Prime o Microchip Libero SoC) e simulatori HDL (come Siemens EDA ModelSim/Questa o Cadence Xcelium) è fondamentale per l'intero flusso di lavoro. HDL Coder genera gli script e i file di configurazione necessari per questi strumenti esterni. Tuttavia, la configurazione di queste interfacce e la risoluzione di problemi di compatibilità tra versioni diverse degli strumenti EDA e dello stesso HDL Coder possono rappresentare una fonte significativa di frustrazione e un notevole dispendio di tempo. Aggiornamenti delle toolchain EDA o di MATLAB/HDL Coder possono introdurre rotture nella compatibilità, richiedendo un costante monitoraggio e adattamento dell'ambiente di sviluppo.

### 3.1.2 Vantaggi e Criticità

L'adozione di HDL Coder, come parte integrante di una metodologia di progettazione basata su modello, promette intrinsecamente una serie di vantaggi significativi che mirano a rivoluzionare il ciclo di sviluppo hardware. Tuttavia, è imperativo analizzare ciascuna di queste promesse ponendo in luce le inevitabili sfide e i complessi compromessi che spesso si celano dietro la facciata dell'automazione.

L'accelerazione del flusso di lavoro è, senza dubbio, uno dei maggiori richiami di HDL Coder. La capacità di tradurre automaticamente algoritmi da MATLAB in codice HDL riduce drasticamente il tempo altrimenti necessario per la codifica manuale a basso livello. Questa efficienza nella fase di generazione del codice è innegabile. Tuttavia, è cruciale riconoscere che tale accelerazione non si traduce sempre linearmente in un corrispondente accorciamento del "time-to-market" [19]. Spesso, il tempo guadagnato nella codifica viene in parte riassorbito nella successiva e ancor più critica fase di validazione dell'HDL generato. Problemi di compatibilità tra le versioni di HDL Coder e gli strumenti di sintesi e implementazione di terze parti, o la presenza di inefficienze strutturali nel codice HDL che emergono solo durante la simulazione RTL o, peggio ancora, in fase di test sull'hardware fisico, possono innescare cicli di debug iterativi e inaspettati ritardi. La fase di ottimizzazione post-generazione, necessaria per raggiungere i target di frequenza e area, può quindi trasformarsi in un collo di bottiglia significativo.

Parallelamente, si enfatizza la riduzione degli errori umani, un beneficio tangibile derivante dall'automazione. HDL Coder è estremamente efficace nel prevenire gli errori di sintassi, le sviste logiche e i refusi tipici della codifica HDL manuale, che possono essere notoriamente difficili e costosi da individuare e correggere. Tuttavia, questa automazione non elimina intrinsecamente gli errori; piuttosto, ne sposta la natura e il punto di origine a un livello di astrazione superiore. Introduce, infatti, la possibilità di errori a livello algoritmico, derivanti da una modellazione in MATLAB non perfettamente ottimizzata per la sua traduzione hardware, o, ancor più frequentemente, da compromessi di quantizzazione e la gestione di overflow/underflow che si manifestano solo quando l'aritmetica floating-point del modello viene mappata su una rappresentazione fixed-point con precisione e range limitati. Se non attentamente gestiti e validati, questi problemi possono portare a un comportamento non conforme alle specifiche in fase hardware, richiedendo un'analisi approfondita e costosa per la loro identificazione e correzione.

La possibilità di intraprendere una progettazione ad alto livello è un pilastro fondamentale del Model-Based Design, consentendo di concentrarsi primariamente sulla logica funzionale dell'algoritmo e sulle prestazioni a livello di sistema. Ciò libera il progettista dai dettagli più granulari della codifica RTL. Tuttavia, questa astrazione non deve tradursi in un'ignoranza delle realtà fisiche e architetturali. Al contrario, bisogna mantenere una solida e profonda comprensione delle implicazioni hardware che le proprie scelte a livello di modello comportano. Ignorare la micro-architettura specifica della FPGA target, i limiti intrinseci delle risorse disponibili (quali LUTs, Flip-Flops, DSP Slices, Block RAMs) o i vincoli critici di timing, può facilmente condurre alla generazione di design che, pur essendo funzionalmente impeccabili nel dominio software, risultano inefficienti (sovradimensionati in termini di area, troppo lenti per la frequenza richiesta o con consumi energetici inaccettabili) o, nella peggiore delle ipotesi, completamente impossibili da implementare sull'hardware desiderato [19].

Per quanto riguarda l'esplorazione del design space, HDL Coder offre strumenti che facilitano una rapida sperimentazione di diverse architetture hardware, come varie configurazioni di pipelining, opzioni di condivisione delle risorse o parametri di quantizzazione fixed-point. Questo consente di valutare rapidamente trade-off complessi. Tuttavia, l'efficacia di questa esplorazione dipende in modo cruciale dalla capacità di analizzare e interpretare correttamente i risultati generati dal tool. Un'interpretazione superficiale o errata dei report sull'utilizzo delle risorse, sulle tempistiche critiche o sulla latenza, può condurre a scelte di design sub-ottimali.

La tracciabilità e verificabilità sono punti di forza innegabili di HDL Coder. La capacità di stabilire una correlazione bidirezionale tra il modello Simulink/MATLAB e il codice HDL generato semplifica enormemente il debug e la verifica, oltre a essere un requisito fondamentale per la conformità a standard industriali rigorosi. Questa trasparenza è preziosa. Ciononostante, la sua utilità massima si realizza solamente quando il modello di riferimento stesso è stato accuratamente validato, è completo e privo di errori. Se il modello di riferimento presenta ambiguità, difetti funzionali o incompletezze, la tracciabilità, per quanto perfetta, si limiterà a propagare tali difetti nell'implementazione hardware, minando l'affidabilità della validazione finale.

La riusabilità del codice, un principio cardine, poiché lo stesso modello astraibile può essere retargetizzato per diverse piattaforme FPGA o ASIC, ottimizzando l'investimento iniziale nello sviluppo dell'algoritmo. Tuttavia, la riusabilità del codice HDL effettivamente generato non è sempre immediata o universale. Questo codice potrebbe richiedere adattamenti manuali significativi per conformarsi a specifici standard di codifica aziendali per facilitare l'integrazione con blocchi IP preesistenti o proprietari, o per sfruttare funzionalità hardware molto specifiche e a basso livello che non sono direttamente esposte o ottimizzate dall'ambiente di generazione automatica. Tali requisiti possono attenuare il beneficio percepito della riusabilità del codice HDL in contesti di produzione complessi.

## 3.2 HDL Verifier

HDL Verifier<sup>TM</sup> è la toolbox designata per affrontare la complessa sfida della verifica e validazione dei design hardware. Il suo obiettivo è ambizioso: riutilizzare l’ambiente di progettazione e test a livello di sistema per verificare le implementazioni a livello RTL e direttamente sull’hardware. Questo approccio promette di garantire che il comportamento dell’hardware finale corrisponda alle rigorose specifiche del modello di riferimento, ma la natura ibrida della verifica introduce inevitabilmente nuove complessità.

HDL Verifier opera creando e gestendo canali di comunicazione bidirezionale tra l’ambiente software e gli ambienti di simulazione hardware o l’hardware fisico, ma questa interconnessione non è esente da latenze e configurazioni delicate.

### 3.2.1 Funzionalità

HDL Verifier si articola attraverso una serie di funzionalità chiave, ognuna delle quali promette di snellire e potenziare il processo di verifica hardware, ma ciascuna porta con sé anche un insieme di sfide e sottigliezze implementative che meritano un’analisi attenta.

La HDL Cosimulation si presenta come una funzionalità estremamente potente, ma è cruciale comprenderne i limiti intrinseci. Il suo principio di funzionamento, che prevede la co-simulazione di un modello Simulink o una funzione MATLAB con un design HDL eseguito in un simulatore di terze parti (come ModelSim o Vivado Simulator), richiede una configurazione meticolosa. La generazione del blocco di cosimulazione in Simulink e del corrispondente modello di wrapper nel simulatore HDL implica una conoscenza approfondita delle interfacce tra i due ambienti. La sfida maggiore risiede nella gestione della sincronizzazione dei clock e delle inevitabili latenze di comunicazione tra il software Simulink e il simulatore HDL. Questo può diventare particolarmente problematico per design ad alta velocità o per quelli che presentano cicli di feedback stretti, dove anche minime asincronie possono compromettere l’integrità della verifica. Il vantaggio primario è l’abilitazione di una verifica funzionale precoce del codice HDL, sia esso generato automaticamente o scritto manualmente, consentendo di individuare discrepanze tra il comportamento a livello di sistema e l’implementazione RTL in una fase presintesi. Tuttavia, questo beneficio si accompagna a un trade-off significativo: un aumento sensibile dei tempi di simulazione rispetto a una simulazione puramente software in Simulink, e la necessità di disporre di licenze e setup complessi per i simulatori HDL esterni [19].

Un’altra funzionalità fondamentale è FPGA-in-the-Loop (FIL), sebbene essa sia un argomento così vasto da meritare un’esplorazione approfondita nella successiva Sezione 3.4. In breve, FIL permette la verifica diretta su hardware fisico. Se da un lato ciò offre un realismo ineguagliabile nella validazione, dall’altro introduce sfide non trascurabili legate alla disponibilità e alla configurazione specifica delle schede FPGA, oltre alle problematiche di latenza della comunicazione tra il

computer host e l'hardware stesso, che possono influenzare la velocità e la fedeltà della simulazione.

La funzionalità di FPGA Data Capture è vitale per il debug hardware, poiché fornisce una finestra sulla "scatola nera" che l'FPGA spesso rappresenta. Il suo funzionamento consente di acquisire segnali interni dal design che gira sulla FPGA e di riportarli nell'ambiente MATLAB per un'analisi dettagliata. Ciò è cruciale per comprendere il comportamento del design in condizioni reali. Tuttavia, questa funzionalità non è priva di limiti. Il numero di segnali che possono essere catturati simultaneamente è intrinsecamente ristretto dalla capacità della memoria on-chip (Block RAM) o dalla larghezza di banda limitata del canale di comunicazione (spesso JTAG). Acquisire tracce molto lunghe o un numero elevato di segnali può quindi rivelarsi problematico o impraticabile. Inoltre, la configurazione dei punti di test all'interno del design HDL, necessari per la cattura dei dati, spesso richiede una re-sintesi completa del design, un processo che può aumentare considerevolmente i tempi di iterazione durante le fasi di debug.

HDL Verifier estende le sue capacità alla generazione automatica di Testbench HDL e componenti UVM (Universal Verification Methodology). Questo aspetto mira a facilitare la creazione di ambienti di verifica robusti e riutilizzabili, riutilizzando gli scenari di test sviluppati in Simulink. Sebbene il tool possa generare testbench funzionali, la loro efficacia per scenari di verifica complessi che richiedono randomizzazione, copertura funzionale avanzata o gestione di protocolli complessi, dipende fortemente dalla completezza e dalla sofisticazione del modello di test in Simulink. Spesso, per raggiungere una copertura di verifica esaustiva, potrebbe rendersi necessario un intervento manuale significativo o l'integrazione con metodologie di verifica più avanzate e dedicate [19].

Infine, l'\*interfacciamento con IP Legacy è un'ulteriore capacità di HDL Verifier che si rivela preziosa in contesti dove esistono già blocchi hardware preesistenti (IP cores). Il tool facilita l'integrazione di questi moduli HDL scritti manualmente o provenienti da altre fonti con nuovi design sviluppati in Simulink, promuovendo un approccio modulare alla progettazione. Tuttavia, l'interoperabilità con IP HDL esistenti, che spesso aderiscono a stili di codifica diversi o presentano specifiche di interfaccia ambigue o non standardizzate, può richiedere un lavoro manuale considerevole per la creazione dei wrapper corretti e la gestione delle necessarie conversioni di protocollo.

### 3.2.2 Struttura di HDL Verifier

La struttura di HDL Verifier è quella di un ecosistema di interfacce e adattatori, ognuno con le sue dipendenze e requisiti:

- **Blocchi di Cosimulazione Simulink:** Sono blocchi complessi che necessitano di una configurazione attenta dei percorsi e delle variabili d'ambiente per interagire correttamente con i simulatori esterni.
- **System Objects per FIL e Data Capture:** Richiedono driver hardware specifici e configurazioni del sistema operativo che possono variare tra diverse schede e versioni.

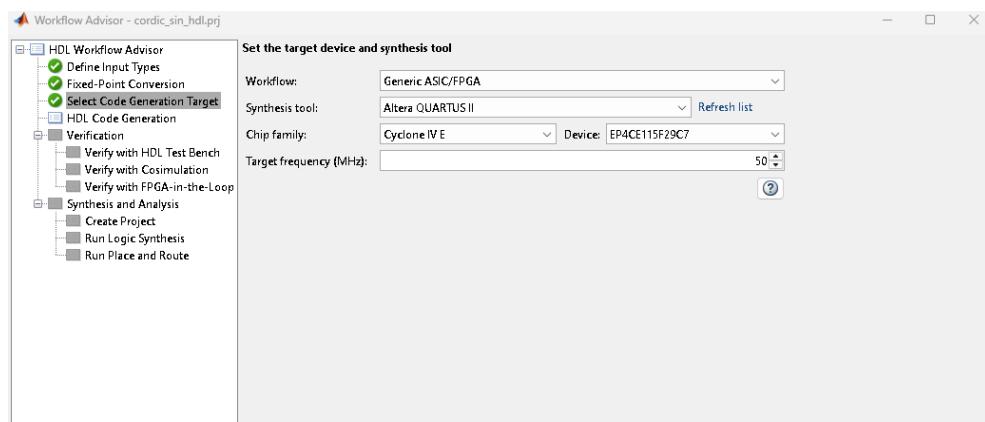
- API di Integrazione: Forniscono controllo programmabile, ma la gestione degli errori e delle eccezioni a livello di comunicazione hardware richiede una robusta gestione del codice.
- Backend per Strumenti EDA: La dipendenza da specifici simulatori e versioni di strumenti EDA può creare problemi di compatibilità e richiedere frequenti aggiornamenti dell'ambiente di sviluppo.
- Driver Hardware: Per l'interazione con le schede FPGA, HDL Verifier si basa su driver forniti dal vendor o da MathWorks. La configurazione di questi driver e la risoluzione di problemi di connettività hardware sono passaggi non banali.

### 3.3 HDL Workflow Advisor

L'HDL Workflow Advisor è una GUI integrata in HDL Coder e HDL Verifier. Si presenta come una guida passo-passo che automatizza e semplifica il processo di progettazione hardware, dalla modellazione alla generazione del bitstream e alla verifica. Se da un lato questa "semplificazione" è una benedizione per gli utenti meno esperti di HDL, dall'altro può nascondere le complessità sottostanti, rendendo difficile il debug o la personalizzazione quando il flusso di lavoro standard non è sufficiente.

Il funzionamento dell'HDL Workflow Advisor si basa su una sequenza logica di task predefiniti. Questa strutturazione impone una certa rigidità: se un task fallisce, è spesso difficile isolare la causa radice senza una comprensione approfondita di ciò che il tool sta effettivamente eseguendo dietro le quinte.

Il Workflow Advisor organizza il processo in una serie di task gerarchici. Ogni task, sebbene automatizzato, invoca una serie di comandi e script che l'utente spesso non vede o non controlla direttamente [19]. Per illustrare visivamente la struttura e le fasi operative di questo strumento, si osservi la Figura 3.1 che mostra l'interfaccia principale dell'HDL Workflow Advisor.



**Figura 3.1:** GUI del Workflow Advisor.

Come mostrato nella Figura 3.1, il Workflow Advisor presenta una serie di cartelle e sottocartelle, ciascuna contenente uno o più task specifici La tipica sequenza di esecuzione include:

1. Set Target:

- Set Target Device and Synthesis Tool: La selezione della scheda FPGA e dello strumento di sintesi è vincolata ai dispositivi e alle versioni di software EDA supportati. La mancata compatibilità con hardware non standard o versioni software obsolete/future può bloccare l'intero processo.
- Set Target Interface: La configurazione delle interfacce è facilitata, ma la conoscenza approfondita dei pinout della scheda e dei protocolli è comunque necessaria per evitare errori di connessione.

2. Prepare Model for HDL Code Generation:

- Check Model for HDL Compatibility: Questo controllo è utile, ma a volte le raccomandazioni possono essere generiche o richiedere modifiche significative al modello che alterano la sua funzionalità originale, imponendo un overhead di re-verifica software.
- Perform Floating-Point to Fixed-Point Conversion: Sebbene guidato, questo processo è iterativo e richiede una profonda conoscenza dei compromessi tra precisione e range [17].

3. Generate RTL Code and Test Bench:

- Generate HDL Code: Sebbene il codice sia generato automaticamente, la sua ottimizzazione finale dipende dalle configurazioni del modello e dalle direttive di HDL Coder. Un design sub-ottimale a livello di Simulink si tradurrà in un HDL sub-ottimale.
- Generate Test Bench: Il testbench generato è funzionale, ma per test di regressione complessi o ambienti di verifica UVM avanzati, spesso è insufficiente e richiede una personalizzazione manuale.

4. Verify Generated HDL Code:

- Run HDL Cosimulation: Il successo di questa fase dipende dalla corretta integrazione con il simulatore HDL esterno, che può presentare le proprie sfide di setup e licenza. I tempi di co-simulazione possono essere molto lunghi per design complessi.

5. Build FPGA Bitstream:

- Synthesize HDL Code e Place and Route: Questi task si basano interamente sulla toolchain del fornitore FPGA. I messaggi di errore e i problemi di temporizzazione (timing violations) che emergono in queste fasi richiedono spesso una conoscenza profonda degli strumenti EDA e dell'architettura FPGA per essere risolti. L'Advisor non è una soluzione di debug completa.

## 6. Program Target Device:

- Program Device: Dipende dalla corretta configurazione del cavo JTAG o della connessione di rete e dalla disponibilità dei driver specifici della scheda. Problemi qui possono essere legati a basse latenze, problemi con i driver oppure problemi di connessione di rete.

## 7. Verify on Target Hardware:

- Run FPGA-in-the-Loop e Capture Data from FPGA: Sebbene potenti, questi metodi di verifica hardware possono essere lenti per grandi volumi di dati o richiedere una riconfigurazione dell'hardware (re-sintesi) per modificare i segnali da catturare.

La dipendenza sequenziale dei task, pur garantendo coerenza, può rallentare il processo di debug iterativo, poiché ogni modifica richiede spesso di ripartire da task precedenti [19].

## 3.4 FPGA-in-the-Loop

FPGA-in-the-Loop è una metodologia di verifica hardware di, integrata in HDL Verifier, che mira a fornire la massima confidenza nella correttezza di un design HDL. La sua promessa è di eseguire un design HDL implementato su una scheda FPGA fisica, controllandolo e testandolo in tempo reale da un modello o script in MATLAB. Sebbene questo offra una validazione di un realismo impareggiabile, la sua implementazione non è banale e porta con sé specifiche complessità e limiti.

Il principio di funzionamento di FIL è quello di creare un "ponte" diretto e sincronizzato tra il dominio software e il dominio hardware, ma la natura di questo ponte introduce latenze e dipendenze che devono essere gestite attentamente [19].

### 3.4.1 Processo di Funzionamento

Il principio di funzionamento di FIL si basa sulla creazione di un "ponte" diretto e sincronizzato tra il dominio software (MATLAB/Simulink) e il dominio hardware (FPGA). Tuttavia, la natura di questo ponte non è semplice e introduce latenze e dipendenze che devono essere gestite con estrema attenzione, rivelando le sue sfumature e i costi operativi.

Il primo passo fondamentale è la preparazione del design HDL. Questo implica una modifica intrinseca del design originale per includere un'infrastruttura di comunicazione specifica, spesso un IP core di comunicazione, essenziale per lo scambio bidirezionale di dati con il computer host. Sebbene questa aggiunta sia automatizzata da HDL Verifier, non è priva di conseguenze: essa consuma risorse logiche preziose sull'FPGA e può alterare significativamente il timing e, di conseguenza, la frequenza operativa massima del design originale. Questo overhead deve essere attentamente considerato in fase di pianificazione.

Successivamente, si procede alla generazione del blocco FIL, che rappresenta l’interfaccia software nell’ambiente Simulink (o un System object in MATLAB). La configurazione di questo blocco è un’operazione delicata che richiede la mappatura accurata delle porte di I/O del modello ai pin fisici effettivi dell’FPGA. Non solo, è indispensabile definire con precisione i tipi di dati e le frequenze di clock. Questi passaggi, apparentemente semplici nella GUI, richiedono in realtà una profonda e specifica conoscenza dell’hardware della scheda FPGA, inclusi i suoi pinout, i suoi standard di tensione e le sue capacità di clocking, pena errori di connessione difficili da diagnosticare.

A seguire vi è la fase cruciale della creazione del bitstream e programmazione dell’FPGA. Questo passaggio non è controllato direttamente da HDL Verifier, ma dipende interamente dalla corretta esecuzione della toolchain FPGA del fornitore (ad esempio, Intel Quartus Prime). Ciò significa che eventuali problemi complessi di timing (come violazioni di setup/hold) o di routing che emergono durante la sintesi, il place-and-route, non sono risolvibili dall’ambiente MathWorks e richiedono un intervento manuale e una profonda esperienza negli strumenti EDA del fornitore FPGA. L’HDL Verifier può orchestrare, ma non diagnosticare o risolvere le problematiche intrinseche all’implementazione fisica.

La fase più critica, e spesso il collo di bottiglia della metodologia FIL, è l’esecuzione sincronizzata e lo scambio di dati. La trasmissione dei dati tra il computer host e l’FPGA, tipicamente attraverso interfacce come JTAG, USB o Ethernet, introduce latenze che possono essere significative e non trascurabili. Per design ad alta throughput o per quelli con requisiti di tempistica estremamente stretti, la banda passante limitata di questi canali di comunicazione e la loro inerente latenza possono rendere impossibile l’esecuzione in tempo reale. In tali scenari, la simulazione FIL è costretta ad avvenire a una velocità ridotta, o richiede l’implementazione di complessi meccanismi di buffering on-chip, che, pur consentendo l’esecuzione, possono talvolta mascherare problemi di timing reali o rallentare drasticamente il ciclo di debug [19].

Infine, la fase di analisi e debug dei risultati del FIL è fondamentale ma complessa. Il confronto tra i dati ottenuti dall’hardware e quelli del modello di riferimento software è la chiave per validare il design. Tuttavia, la natura spesso asincrona della comunicazione tra host e FPGA, unitamente alla possibilità di errori di quantizzazione (dalla conversione fixed-point) o di overflow/underflow, può rendere estremamente difficile l’identificazione della causa radice di una discrepanza. Lo strumento di FPGA Data Capture, pur essendo un ausilio prezioso per la visibilità interna all’FPGA, presenta limiti intrinseci sulla quantità di segnali che possono essere catturati e sulla lunghezza delle tracce acquisibili in una singola run, limitando la visibilità su eventi rari o su lunghe sequenze di operazione. Questo può prolungare notevolmente i tempi di debug.

### 3.4.2 Struttura di FPGA-in-the-Loop

La metodologia FIL si basa su un’architettura complessa e su un’interazione precisa tra diverse componenti, ciascuna delle quali può diventare un punto di fallimento:

- Scheda FPGA Supportata: L'uso è limitato a schede specifiche che MathWorks ha integrato e testato. L'utilizzo di schede non supportate o personalizzate richiede un notevole sforzo manuale per l'integrazione.
- Computer Host: Richiede risorse computazionali adeguate per eseguire MATLAB/Simulink e gestire il traffico dati con l'FPGA.
- Infrastruttura di Comunicazione Hardware: Questa IP, sebbene generata automaticamente, è un'aggiunta al design che consuma risorse logiche e può impattare il timing finale. La scelta del protocollo di comunicazione (JTAG, Ethernet, PCIe) influisce sulla latenza e sulla throughput.
- Driver Software e API: La corretta installazione e configurazione dei driver specifici per la scheda FPGA e il sistema operativo è fondamentale e può essere una fonte di problemi comuni.
- Blocco FIL in Simulink / System Object in MATLAB: Questi blocchi sono il punto di controllo software, ma la loro configurazione accurata è cruciale e spesso richiede la comprensione dei registri hardware sottostanti.
- Toolchain del Fornitore FPGA: La dipendenza da strumenti esterni (sintesi, place-and-route, programmazione) significa che problemi in queste toolchain si riflettono direttamente nel flusso di lavoro FIL, e il debug di tali problemi esula dalle capacità di HDL Verifier [19].

Sebbene l'ecosistema MathWorks per la progettazione hardware offra strumenti e metodologie all'avanguardia che promettono di accelerare il "time-to-market" e migliorare la qualità del design, è cruciale adottare un approccio realistico e critico. L'automazione fornita da HDL Coder e il supporto alla verifica di HDL Verifier, guidati dall'HDL Workflow Advisor e convalidati tramite FPGA-in-the-Loop, rappresentano passi significativi. Tuttavia, non eliminano la necessità di una profonda conoscenza dell'architettura hardware, dei compromessi di ottimizzazione, delle sfide della verifica e della risoluzione dei problemi a basso livello. Solo con una consapevolezza di questi aspetti è possibile sfruttare appieno il potenziale di questi strumenti, trasformando le loro promesse in successi concreti nel complesso mondo della progettazione digitale.

# Capitolo 4

## IMPLEMENTAZIONE E SIMULAZIONE DEL MODULO CORDIC IN VHDL

L'implementazione dell'algoritmo CORDIC in VHDL costituisce una fase cruciale nel processo di progettazione, fungendo da ponte tra la modellazione teorica e la verifica pratica su hardware. Questo capitolo illustra in dettaglio le attività di sviluppo, sintesi e simulazione dell'algoritmo, utilizzando strumenti professionali quali ModelSim e Quartus Prime.

### 4.1 Introduzione al VHDL

VHDL, acronimo di VHSIC Hardware Description Language (Very High Speed Integrated Circuit Hardware Description Language), è un linguaggio standardizzato per la descrizione di circuiti digitali. È un linguaggio di modellazione e descrizione che permette ai progettisti di definire il comportamento, la struttura e l'interconnessione di componenti hardware a vari livelli di astrazione, dal gate level al sistema completo. La sua natura concorrente e orientata agli eventi lo rende particolarmente adatto per catturare il parallelismo intrinseco dell'hardware, a differenza dei linguaggi di programmazione sequenziali tradizionali [22, 23].

#### 4.1.1 Il Workflow del VHDL

Il processo di sviluppo di un sistema digitale utilizzando VHDL segue un flusso ben definito. Inizia con la Design Entry, dove il comportamento o la struttura del circuito viene descritto in VHDL. Questa fase produce il codice sorgente che sarà successivamente elaborato. A seguire, si esegue la Simulazione Funzionale, un passaggio critico che verifica la correttezza logica e funzionale del design. In questa fase, il simulatore interpreta il codice VHDL per emulare il comportamento del circuito senza considerare vincoli temporali specifici dell'hardware. Eventuali errori di logica o di comportamento vengono identificati e corretti qui.

Una volta che il design funziona correttamente a livello funzionale, si procede con la Sintesi. Durante la sintesi, il codice VHDL astratto viene tradotto in una

netlist, ovvero una descrizione a livello di gate o di macro-celle logiche elementari, specifica per la tecnologia FPGA di destinazione [22]. Questo passaggio ottimizza il design per l'area e la velocità. Successivamente, si entra nella fase di Place-and-Route, dove gli elementi logici della netlist vengono fisicamente mappati sulle risorse disponibili sull'FPGA (blocchi logici, I/O, memorie) e le interconnessioni vengono stabilite. Questa fase è cruciale per il raggiungimento delle prestazioni desiderate.

Dopo il place-and-route, si esegue la Simulazione Temporale. Questa simulazione è più accurata della simulazione funzionale, in quanto include i ritardi di propagazione introdotti dalle interconnessioni e dai componenti fisici dell'FPGA. È fondamentale per verificare che il circuito rispetti i requisiti di temporizzazione. Infine, il design viene convertito in un bitstream, un file binario specifico per l'FPGA che viene caricato sul dispositivo per configurarlo. L'ultima fase è la Programmazione dell'FPGA e la Validazione sul campo, dove il comportamento del circuito viene testato sull'hardware reale [23].

## 4.2 Ambienti di Sviluppo: ModelSim e Quartus Prime

Lo sviluppo del modulo CORDIC ha fatto uso di due ambienti di sviluppo principali, fondamentali per il processo che va dalla descrizione del comportamento hardware alla sua implementazione fisica: ModelSim, per la simulazione e la verifica funzionale e temporale, e Quartus Prime, come suite completa per la progettazione e la configurazione degli FPGA.

ModelSim, sviluppato da Siemens EDA (precedentemente noto come Mentor Graphics), si erge come uno standard industriale nel campo dei simulatori HDL. La sua applicazione si estende all'analisi e al debugging di sistemi digitali descritti in linguaggi come VHDL, Verilog e SystemVerilog. La robustezza di ModelSim risiede nella sua capacità di simulare il comportamento di un design a molteplici livelli di astrazione, offrendo al progettista una visione approfondita e granulare. Inizialmente, si impiega la simulazione comportamentale o a livello di RTL, che consente una verifica rapida e funzionale del codice HDL, dissociata dai dettagli specifici dell'hardware [24]. Questa fase è cruciale per validare la logica del design prima di impegnarsi in complesse fasi di sintesi. Successivamente, una volta che il design è stato sintetizzato, ModelSim è in grado di eseguire la simulazione post-sintesi o gate-level simulation. Questa modalità integra i ritardi di propagazione introdotti dai componenti logici effettivi e dalle interconnessioni fisiche, fornendo una verifica molto più realistica del timing del circuito e anticipando potenziali problemi di temporizzazione sull'hardware reale [23]. Le funzionalità distintive di ModelSim che lo rendono uno strumento indispensabile includono:

- **Waveform Viewer:** Questo è forse il componente più utilizzato, un'interfaccia grafica intuitiva che permette di visualizzare le transizioni dei segnali nel tempo. È possibile osservare il comportamento dinamico del design, identificare

ritardi, glitches, e confrontare i segnali reali con quelli attesi. La capacità di rappresentare i segnali in vari formati (binario, esadecimale, decimale, analogico) ne aumenta la versatilità [24].

- Debuggers e Breakpoints: ModelSim fornisce potenti strumenti di debugging simili a quelli dei linguaggi di programmazione software. È possibile impostare breakpoint nel codice HDL per interrompere la simulazione in punti specifici, esaminare il valore di tutti i segnali e le variabili in quel momento, e procedere passo-passo nell'esecuzione. Questa capacità è inestimabile per isolare e risolvere errori logici complessi.
- Code Coverage: Una funzione analitica che valuta l'efficacia del testbench, misurando quale percentuale del codice HDL (ad esempio, linee di codice, istruzioni, rami di decisioni) è stata effettivamente testata durante le simulazioni. Questo aiuta i progettisti a identificare aree del design non sufficientemente coperte dai test e a migliorare la robustezza della verifica.
- Automatic Testbench Generation: Per accelerare la fase iniziale di verifica, ModelSim può generare automaticamente testbench di base, fornendo un punto di partenza per l'applicazione di stimoli e l'osservazione delle risposte.

L'importanza di ModelSim nel flusso di progettazione FPGA è innegabile, poiché permette di identificare e correggere un'ampia gamma di errori funzionali e di temporizzazione in una fase precoce dello sviluppo. Questo riduce drasticamente la necessità di debugging sull'hardware fisico, con un conseguente risparmio significativo in termini di tempo e costi di sviluppo.

Quartus Prime, sviluppato da Intel (precedentemente conosciuto come Altera Quartus II), rappresenta un ambiente di progettazione integrato (IDE) onnicomprensivo per gli FPGA. Questo software è il fulcro dell'intero ciclo di vita dello sviluppo FPGA, dalla descrizione concettuale in HDL fino alla configurazione fisica del dispositivo [25]. Quartus Prime fornisce una suite di strumenti sinergici, ciascuno progettato per ottimizzare un aspetto specifico del design [20, 16]. Le sue funzionalità chiave sono:

- Compilatore HDL: Il cuore di Quartus Prime. Questo compilatore interpreta il codice VHDL o Verilog, eseguendo un'analisi sintattica e semantica rigorosa. Successivamente, traduce la descrizione astratta in una rappresentazione logica che è compatibile e ottimizzata per la specifica architettura dell'FPGA di destinazione. Le sue capacità di ottimizzazione sono cruciali per garantire che il design utilizzi le risorse hardware in modo efficiente.
- Sintesi: Questa fase cruciale converte la netlist logica in un insieme di elementi hardware primitivi (come porte logiche, flip-flop, blocchi di memoria dedicati) e le loro interconnessioni, che possono essere implementati fisicamente sull'FPGA. Quartus Prime incorpora algoritmi di sintesi avanzati che mirano a ottimizzare il design non solo per la funzionalità, ma anche per parametri critici come l'area (occupazione di risorse), la velocità (frequenza massima operativa) e il consumo energetico.

- Place-and-Route: Dopo la sintesi, questa fase complessa si occupa dell’allocazione fisica dei componenti logici sintetizzati alle risorse disponibili sull’FPGA (ad esempio, blocchi logici configurabili, blocchi di I/O, blocchi di memoria integrati) e del tracciamento dei percorsi di interconnessione (routing) tra di essi. Il processo di place-and-route è guidato da vincoli temporali e spaziali, con l’obiettivo di minimizzare i ritardi di propagazione e massimizzare la frequenza operativa, garantendo al contempo che il design si adatti alle risorse disponibili.
- Analisi Temporale: Quartus Prime include strumenti di analisi temporale estremamente robusti, come il rinomato TimeQuest Timing Analyzer. Questi strumenti sono fondamentali per verificare che il design soddisfi tutti i requisiti di temporizzazione. Calcolano i ritardi di propagazione dei segnali lungo tutti i percorsi critici del circuito e identificano eventuali violazioni di setup e hold time, che potrebbero compromettere l’affidabilità del sistema. Permette una gestione dettagliata dei vincoli di temporizzazione (come periodi di clock, ritardi di input/output).
- Power Analyzer: Un tool integrato che fornisce stime accurate del consumo energetico del design sull’FPGA, distinguendo tra potenza statica e dinamica. Questo permette ai progettisti di valutare l’impatto delle loro scelte di design sul consumo e di ottimizzare il circuito per applicazioni a basso consumo, un fattore sempre più critico nei sistemi embedded.
- Visualizzatore di Risorse: Questo strumento offre una rappresentazione chiara e dettagliata dell’utilizzo delle risorse hardware sull’FPGA. Fornisce un resoconto preciso di Logic Elements (LEs), registri, blocchi di memoria M9K, blocchi DSP (Digital Signal Processing) e altre risorse, consentendo ai progettisti di monitorare l’occupazione e di pianificare strategicamente l’allocazione delle risorse per design più complessi.
- SignalTap II Logic Analyzer: Un potente strumento di debugging hardware integrato che permette ai progettisti di acquisire e visualizzare i segnali interni del design direttamente sull’FPGA in tempo reale, senza bisogno di strumentazione esterna. Questo è di cruciale importanza per il debugging di problemi che si manifestano solo sull’hardware o in condizioni operative complesse.
- Generazione del Bitstream: Dopo le fasi di sintesi e place-and-route e la validazione della temporizzazione, Quartus Prime genera il file di configurazione binario, cioè il bitstream. Questo file viene caricato sull’FPGA per programmarlo, configurando le sue risorse interne in modo da implementare fisicamente il design creato.

## 4.3 Il modulo CORDIC e il testbench

Il cuore di questa implementazione è il modulo CORDIC, progettato con un'architettura iterativa per massimizzare il controllo sul ciclo di clock e gestire direttamente la latenza. Questo modulo è in grado di calcolare le funzioni seno, arcoseno e arcotangente. L'algoritmo CORDIC opera su un principio di rotazioni vettoriali, approssimando la funzione desiderata attraverso una serie di rotazioni elementari.

Per l'implementazione delle rotazioni, il modulo si avvale di una tabella di angoli precalcolati. Il valore  $K$  rappresenta il fattore di scala inverso dell'algoritmo, necessario per compensare l'amplificazione del vettore durante le rotazioni in modalità rotazione.

La verifica funzionale del modulo CORDIC è stata condotta attraverso un testbench dedicato. Un testbench è un'entità VHDL che simula l'ambiente operativo del design under test (DUT), fornendo segnali di ingresso (stimoli) e verificando i segnali di uscita. Non rappresenta hardware fisico, ma un ambiente di simulazione per validare la correttezza logica del design.

Il testbench genera i segnali di clock e reset necessari per il funzionamento del modulo CORDIC. Nello specifico, per le simulazioni, il clock del testbench è stato impostato con un periodo di 20 nanosecondi, corrispondente a una frequenza di 50 MHz. Questa scelta è stata dettata da diversi motivi strategici:

- **Margine di Sicurezza per la Simulazione Funzionale:** L'obiettivo primario del testbench in questa fase è la verifica funzionale del design, non la sua massima velocità operativa. Impostare una frequenza relativamente moderata garantisce che tutti i ritardi di propagazione combinatori siano ampiamente coperti all'interno di un ciclo di clock. Questo previene l'insorgere di false violazioni di temporizzazione (come setup o hold time violations) che potrebbero confondere il debugging dei problemi logici intrinseci del design.
- **Stabilità della Simulazione:** Una frequenza di clock più bassa in simulazione tende a rendere il comportamento dei segnali più stabile e facilmente osservabile nel waveform viewer, facilitando l'analisi dettagliata delle transizioni e la propagazione dei dati.
- **Replicabilità e Chiarezza:** Utilizzare una frequenza standard e ben comprensibile (50 MHz è comune in molti sistemi embedded) rende le simulazioni più replicabili e i risultati più chiari nell'interpretazione. Si tratta di una frequenza realistica per molte applicazioni pratiche, fornendo un buon compromesso per la validazione iniziale.

La sua funzione principale è applicare una serie predefinita di valori di ingresso per le funzioni seno, arcoseno e arcotangente, e confrontare le uscite del modulo CORDIC con i valori attesi. Il testbench gestisce la conversione dei valori reali in fixed-point nel formato Q4.12, che è stato scelto per l'implementazione hardware. La selezione di questo formato, insieme al numero di 16 iterazioni per l'algoritmo

CORDIC, è il risultato di un'attenta valutazione del compromesso tra costo, prestazioni, risorse hardware e precisione. Una volta applicati gli input, il testbench attende il segnale done dal modulo CORDIC, che indica la disponibilità di un risultato valido, e procede poi alla verifica.

Il modulo CORDIC, essendo iterativo, mostra un comportamento temporale specifico: il primo risultato utile è disponibile dopo 16 cicli di clock.

Le simulazioni sono state condotte con i seguenti parametri:

- Per le funzioni seno e arcotangente, gli angoli di ingresso sono stati variati da 0 a 360 gradi, con un passo di 0.1 gradi.
- Per la funzione arcoseno, gli input sono stati variati da -1 a 1, con un passo di 0.1.
- Il tempo totale di simulazione per ogni test è stato di 5000 nanosecondi.

Questo approccio ha permesso di valutare l'accuratezza e le prestazioni del modulo CORDIC su un'ampia gamma di input.

## 4.4 Risultati di Sintesi con Quartus Prime

In questa sezione presenta e analizza in dettaglio i risultati ottenuti dalla fase di sintesi e implementazione, utilizzando l'ambiente di sviluppo Quartus Prime di Intel. La progettazione del modulo è stata incentrata sull'efficienza delle risorse e sulle prestazioni, adottando una rappresentazione in virgola fissa Q4.12. Questa decisione di progettazione riflette un bilanciamento ottimale tra la precisione necessaria per l'algoritmo CORDIC e un utilizzo efficiente delle risorse hardware, risultando in un design più compatto ed economico rispetto all'adozione della virgola mobile.

I risultati di sintesi e analisi sono stati ottenuti per tre diverse configurazioni del modulo CORDIC, variando il numero di iterazioni: 8, 16 e 32. Queste versioni permettono di valutare l'impatto della complessità computazionale sulle risorse utilizzate, sulla frequenza operativa e sul consumo energetico.

Parametro	Valore (8 iterazioni)	Percentuale
Logic Elements	1506	1%
Registri	549	<1%
I/O Pins	101	19%
Clock Pins	2	29%
LAB	147	3%
DSP Blocks	0	0%
RAM (M9K)	28	<1%
PLL	0	0%
Global Signals	1	-
Fan-Out Massimo	90	-
Fan-Out Medio	2.62	-
Interconnect Usage (Media)	0.6%	-
Interconnect Usage (Picco)	8.3%	-
Frequenza Operativa	130.89 MHz	-
Setup Slack	0.368 ps	-
Hold Slack	0.429 ps	-
Consumo Energetico (Totale)	145.50 mW	-
Potenza Core Dinamica	0.00 mW	-
Potenza Core Statica	98.62 mW	-
Potenza I/O	46.88 mW	-
Temperatura di Giunzione	26°C	-
Corrente VCCINT	18.41 mA	-
Corrente VCCIO	11.76 mA	-
Corrente VCCA	32.06 mA	-
Corrente VCCD	2.58 mA	-
Corrente Pin I/O	1.45 mA	-

**Tabella 4.1:** Risultati di sintesi e analisi per la versione a 8 iterazioni.

Parametro	Valore (16 iterazioni)	Percentuale
Logic Elements	1986	2%
Registri	944	<1.5%
I/O Pins	101	19%
Clock Pins	2	29%
LAB	198	4%
DSP Blocks	0	0%
RAM (M9K)	60	<1%
PLL	0	0%
Global Signals	1	-
Fan-Out Massimo	948	-
Fan-Out Medio	3.08	-
Interconnect Usage (Media)	0.9%	-
Interconnect Usage (Picco)	11.4%	-
Frequenza Operativa	125.02 MHz	-
Setup Slack	0.342 ps	-
Hold Slack	0.411 ps	-
Consumo Energetico (Totale)	170.25 mW	-
Potenza Core Dinamica	0.00 mW	-
Potenza Core Statica	98.72 mW	-
Potenza I/O	71.53 mW	-
Temperatura di Giunzione	26°C	-
Corrente VCCINT	22.74 mA	-
Corrente VCCIO	13.12 mA	-
Corrente VCCA	34.20 mA	-
Corrente VCCD	2.64 mA	-
Corrente Pin I/O	1.48 mA	-

**Tabella 4.2:** Risultati di sintesi e analisi per la versione a 16 iterazioni.

Parametro	Valore (32 iterazioni)	Percentuale
Logic Elements	2858	2.5%
Registri	1734	2%
I/O Pins	101	19%
Clock Pins	2	29%
LAB	276	6%
DSP Blocks	0	0%
RAM (M9K)	124	<1%
PLL	0	0%
Global Signals	1	-
Fan-Out Massimo	1738	-
Fan-Out Medio	3.18	-
Interconnect Usage (Media)	1.2%	-
Interconnect Usage (Picco)	13.5%	-
Frequenza Operativa	117.43 MHz	-
Setup Slack	0.295 ps	-
Hold Slack	0.402 ps	-
Consumo Energetico (Totale)	189.30 mW	-
Potenza Core Dinamica	0.00 mW	-
Potenza Core Statica	98.75 mW	-
Potenza I/O	90.55 mW	-
Temperatura di Giunzione	26°C	-
Corrente VCCINT	34.12 mA	-
Corrente VCCIO	13.93 mA	-
Corrente VCCA	36.41 mA	-
Corrente VCCD	2.75 mA	-
Corrente Pin I/O	1.58 mA	-

**Tabella 4.3:** Risultati di sintesi e analisi per la versione a 32 iterazioni.

L’analisi comparativa delle tre configurazioni (8, 16 e 32 iterazioni) del modulo CORDIC rivela chiare tendenze nell’utilizzo delle risorse, nelle prestazioni e nel consumo energetico, fornendo una visione approfondita dei compromessi di design [25, 20].

- Logic Elements e Logic Array Blocks: I Logic Elements (LEs) sono le unità logiche più piccole e configurabili all’interno di un FPGA, contenenti lookup tables e flip-flop. I LABs sono raggruppamenti di LEs, che forniscono una struttura più ampia per l’implementazione della logica. La percentuale indica l’occupazione rispetto al totale disponibile sull’FPGA target (Cyclone IV). L’utilizzo di Logic Elements e LABs aumenta progressivamente con l’incremento delle iterazioni:

- 8 iterazioni: 1.506 LEs (1%), 147 LABs (3%)
- 16 iterazioni: 1.986 LEs (2%), 198 LABs (4%)

- 32 iterazioni: 2.858 LEs (2.5%), 276 LABs (6%)

Questo aumento è pienamente atteso. Un numero maggiore di iterazioni nell'algoritmo CORDIC implica una maggiore complessità computazionale, che si traduce direttamente in una necessità di più logica combinatoria e sequenziale (LEs e LABs) per implementare i calcoli e gli stadi addizionali della pipeline.

- **Registri:** I registri (o flip-flop) sono elementi di memorizzazione essenziali per la logica sequenziale. Vengono utilizzati per memorizzare lo stato del circuito e per implementare la pipelining, consentendo al design di operare a frequenze più elevate. Il numero di registri segue un trend di aumento significativo con le iterazioni:
  - 8 iterazioni: 549 registri (<1%)
  - 16 iterazioni: 944 registri (<1.5%)
  - 32 iterazioni: 1.734 registri (2%)

L'aumento del numero di registri è del tutto coerente con l'incremento del numero di iterazioni: ogni iterazione aggiuntiva comporta un'espansione delle strutture di controllo e memorizzazione necessarie per l'elaborazione. Questo riflette la maggiore complessità del design, che richiede più risorse per gestire i dati interni e i calcoli intermedi.

- **I/O Pins e Clock Pins:** Gli I/O Pins sono i pin fisici utilizzati per la comunicazione del modulo con l'esterno. I Clock Pins sono i pin dedicati per la distribuzione dei segnali di clock, vitali per la sincronizzazione del circuito. Entrambi i parametri rimangono costanti tra le configurazioni:
  - I/O Pins: 101 (19%) per tutte le versioni.
  - Clock Pins: 2 (29%) per tutte le versioni.

Questi parametri sono dettati dall'interfaccia esterna del modulo e dal setup del clock, che non cambiano con il numero di iterazioni interne dell'algoritmo. L'interfaccia (input/output) e i segnali di clock (main clock, reset) rimangono gli stessi indipendentemente dalla profondità di calcolo interna.

- **DSP Blocks:** Blocchi hardware specializzati presenti negli FPGA, ottimizzati per operazioni DSP come moltiplicazioni e accumulazioni, fondamentali in applicazioni come filtri o trasformate rapide. I DSP Blocks rimangono a 0 in tutte le configurazioni.

Questa è una caratteristica chiave dell'algoritmo CORDIC. Per sua natura, il CORDIC evita le moltiplicazioni complesse (che tipicamente richiederebbero blocchi DSP) a favore di semplici operazioni di shift e addizione/sottrazione. Questo lo rende estremamente efficiente in termini di risorse logiche generiche e riduce la dipendenza da blocchi hardware dedicati e spesso più costosi.

- RAM (M9K): Blocchi di memoria embedded utilizzati per memorizzare tabelle, buffer o altre strutture dati. L'utilizzo della RAM aumenta con le iterazioni:
  - 8 iterazioni: 28 (<1%)
  - 16 iterazioni: 60 (<1%)
  - 32 iterazioni: 124 (<1%)

Il modulo CORDIC utilizza una tabella di angoli precalcolati (ATAN\_TABLE). Un aumento delle iterazioni può richiedere una tabella più grande o più accessi a porzioni diverse della tabella, giustificando un incremento nell'utilizzo della RAM per memorizzare questi valori o per bufferizzare dati intermedi legati al maggior numero di stadi.

- PLL e Global Signals: Le PLL sono blocchi hardware che generano e distribuiscono segnali di clock con caratteristiche specifiche (frequenza, fase). I Global Signals sono linee di clock e di reset a bassa skew che raggiungono quasi tutti i flip-flop sul chip. Rimangono costanti: PLL 0, Global Signals 1 per tutte le versioni.

L'implementazione del modulo CORDIC di per sé non richiede una PLL interna aggiuntiva. Un singolo segnale globale di clock è sufficiente per sincronizzare il design, indipendentemente dal numero di iterazioni.

- Fan-Out Massimo e Fan-Out Medio: Il Fan-Out si riferisce al numero di input che un'uscita logica pilota. Un Fan-Out elevato può indicare un segnale critico che pilota molte altre logiche, potenzialmente introducendo ritardi. Aumentano con le iterazioni:

- Fan-Out Massimo: 90 → 948 → 1738
- Fan-Out Medio: 2.62 → 3.08 → 3.18

Con l'aumento della complessità e del numero di registri, è naturale che alcuni segnali debbano pilotare un numero maggiore di destinazioni. Questo aumento è un risultato diretto dell'incremento delle risorse logiche e della connettività interna.

- Interconnect Usage (Media e Picco): Misura l'utilizzo delle risorse di interconnessione (routing) all'interno dell'FPGA. Un valore più alto indica una maggiore complessità e densità del routing necessario per collegare tutti i componenti logici. Aumentano progressivamente:

- Media: 0.6% → 0.9% → 1.2%
- Picco: 8.3% → 11.4% → 13.5%

Con più Logic Elements e Registri da collegare, e percorsi più lunghi da instradare, è inevitabile che l'utilizzo delle risorse di interconnessione aumenti.

Questo riflette la maggiore "densità" del design e la complessità del compito di place-and-route del compilatore.

- Frequenza Operativa: La massima frequenza di clock alla quale il design può operare correttamente, determinata dal percorso critico. La frequenza operativa mostra una chiara tendenza alla diminuzione con l'aumento delle iterazioni:

- 8 iterazioni: 130.89 MHz
- 16 iterazioni: 125.02 MHz
- 32 iterazioni: 117.43 MHz

Questo andamento è coerente con le aspettative. L'aumento del numero di iterazioni e, di conseguenza, della complessità logica e del numero di registri, rende più difficile per il tool di sintesi trovare un percorso critico altrettanto breve. Anche se il design è pipelined, un numero maggiore di stadi nella pipeline tende ad aumentare la lunghezza complessiva dei percorsi di routing e la complessità del bilanciamento temporale tra gli stadi, portando a una leggera riduzione della massima frequenza raggiungibile.

- Setup Slack e Hold Slack: Sono indicatori cruciali per la temporizzazione. Il Setup Slack è il margine di tempo prima del fronte di clock in cui un dato deve essere stabile all'input di un registro. L'Hold Slack è il margine di tempo dopo il fronte di clock in cui un dato deve rimanere stabile. Valori positivi indicano che i requisiti di temporizzazione sono soddisfatti. Entrambi i valori diminuiscono leggermente con l'aumento delle iterazioni:

- Setup Slack: 0.368 ps → 0.342 ps → 0.295 ps
- Hold Slack: 0.429 ps → 0.411 ps → 0.402 ps

Una diminuzione dello slack (pur rimanendo positivo) indica che il design si sta avvicinando ai limiti di temporizzazione del dispositivo, specialmente con l'aumento della complessità e la conseguente riduzione della Frequenza Operativa. I margini positivi indicano comunque che il design è temporizzato correttamente a queste frequenze.

- Consumo Energetico (Totale, Core Dinamica, Core Statica, I/O) e Correnti: Il consumo energetico totale è la somma della potenza dissipata. La Potenza Core Dinamica è dovuta alle commutazioni dei segnali all'interno dei blocchi logici. La Potenza Core Statica è dovuta alle correnti di leakage dei transistor. La Potenza I/O è il consumo dei pin di input/output. Le correnti (VCCINT, VCCIO, VCCA, VCCD, Pin I/O) indicano la corrente assorbita dalle diverse sezioni del chip. La Temperatura di Giunzione è la temperatura operativa interna del chip. Il consumo energetico totale e le correnti (VCCINT, VCCIO, VCCA, VCCD, Pin I/O) mostrano un aumento generale con l'aumento delle iterazioni, mentre la Potenza Core Statica rimane relativamente costante.

- Consumo Energetico Totale: 145.50 mW → 170.25 mW → 189.30 mW
- Potenza I/O: 46.88 mW → 71.53 mW → 90.55 mW
- Correnti VCCINT, VCCIO, VCCA, VCCD, Pin I/O: Mostrano aumenti progressivi.

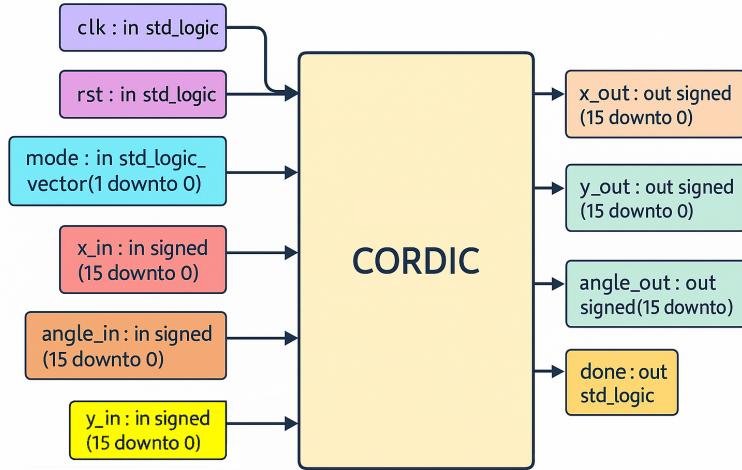
L'aumento del consumo energetico e delle correnti è direttamente correlato all'incremento del numero di Logic Elements, Registri e interconnessioni attivate. Più logica significa più transistor che commutano, aumentando la potenza dinamica, e più transistor che contribuiscono alla potenza statica. La potenza I/O aumenta con l'attività dei pin e la complessità interna che li pilota. La temperatura di giunzione rimane costante a 26°C, suggerendo che il design opera ben al di sotto dei limiti termici del dispositivo, anche nella configurazione più complessa.

L'analisi incrociata delle tabelle evidenzia che l'implementazione del modulo CORDIC su FPGA scala in modo prevedibile con l'aumento delle iterazioni. Se da un lato l'aumento di complessità comporta un maggiore utilizzo delle risorse e un leggero calo della frequenza operativa, dall'altro il design mantiene un'elevata efficienza, in particolare l'assenza di utilizzo dei blocchi DSP, e un consumo energetico contenuto rispetto alla funzionalità offerta. Questo dimostra la robustezza e l'ottimizzazione del design per la piattaforma FPGA [25].

## 4.5 Analisi dell'Implementazione Hardware

Prima di procedere all'analisi dettagliata delle waveform risultanti dalla simulazione e sintesi del modulo CORDIC, è cruciale comprendere a fondo la sua struttura hardware, come implementato nel linguaggio VHDL. Questa sezione descrive l'entità principale del modulo CORDIC e la sua architettura interna, evidenziando le interfacce, la rappresentazione dei dati, e la logica di controllo che abilita la gestione di più funzioni matematiche all'interno di un singolo blocco hardware.

L'entità VHDL del modulo CORDIC è denominata *cordic*. Essa definisce l'interfaccia esterna attraverso la quale il componente interagisce con il resto del sistema. La sua definizione è rappresentata nella seguente Figura 4.1:



**Figura 4.1:** Diagramma a blocchi di un’unità CORDIC con i relativi segnali di ingresso e uscita.

Dettagli delle porte:

- **clk:** Il segnale di clock principale, responsabile della sincronizzazione di tutte le operazioni sequenziali all’interno del modulo.
- **rst:** Un segnale di reset asincrono (attivo alto), utilizzato per inizializzare lo stato interno del CORDIC al suo stato predefinito.
- **mode:** Questo segnale di input, di tipo `std_logic_vector(1 downto 0)`, è fondamentale poiché definisce la funzione specifica che il modulo CORDIC deve calcolare. La sua codifica è la seguente:
  - -"00": Modalità di Rotazione, utilizzata per il calcolo della funzione seno.
  - -"01": Modalità di Vettorizzazione, utilizzata per il calcolo della funzione arcotangente.
  - -"10": Modalità di Vettorizzazione per il calcolo della funzione arcoseno.
- **x\_in, y\_in, angle\_in:** Questi input forniscono i valori iniziali ( $X_0$ ,  $Y_0$ ,  $Z_0$ ) per i registri interni del CORDIC, a seconda della modalità operativa selezionata. Sono tutti numeri Fixed-Point con segno (`signed`) su 16 bit.
- **x\_out, y\_out, angle\_out:** Questi output forniscono i risultati finali ( $X_f$ ,  $Y_f$ ,  $Z_f$ ) dopo il completamento delle iterazioni CORDIC. Mantengono lo stesso formato Fixed-Point degli input.
- **done:** Un segnale di output che si attiva quando l’operazione CORDIC è stata completata, indicando che i risultati sui portali `x_out`, `y_out`, `angle_out` sono validi e possono essere letti dal sistema host.

L'architettura implementa la logica iterativa dell'algoritmo CORDIC attraverso una struttura a pipeline. Questa scelta architettonica permette di ottenere un throughput elevato, in quanto le diverse iterazioni dell'algoritmo vengono eseguite in parallelo su stadi hardware dedicati.

È un aspetto fondamentale di questa implementazione il fatto che tutte e tre le funzioni principali sono trattate all'interno di un unico blocco di codice VHDL e hardware. Questo significa che non sono necessari tre moduli separati, ma un singolo componente riconfigurabile in tempo reale tramite l'ingresso *mode*.

La scelta della rappresentazione Fixed-Point è centrale per l'implementazione su FPGA. Tutti i segnali numerici e le costanti sono definiti utilizzando il tipo `signed` su 16 bit. La rappresentazione utilizzata è di tipo Q4.12, dove 1 bit è per il segno, 3 bit per la parte intera e 12 bit per la parte frazionaria. Questa precisione è stata scelta per bilanciare l'accuratezza richiesta con l'efficienza delle risorse hardware. Il modulo è stato progettato per eseguire diverse funzioni CORDIC. L'ingresso *mode* è il meccanismo attraverso cui la funzione desiderata viene selezionata e configurata. Questa logica è centralizzata in due punti chiave dell'architettura:

- Inizializzazione dei Valori: Un blocco logico combinatorio determina i valori iniziali dei registri CORDIC basandosi sul valore del segnale *mode*.
  - - Se *mode* = "00" (Rotazione, per il seno), *x\_initial\_comb* è inizializzato con il fattore di scala inverso dopo la riduzione di quadrante, *y\_initial\_comb* a zero e *z\_initial\_comb* riceve l'angolo normalizzato di input.
  - - Se *mode* = "01" (Vettorizzazione, per l'arcotangente), *x\_initial\_comb* e *y\_initial\_comb* ricevono i valori assoluti degli input *x\_in* e *y\_in*, mentre *z\_initial\_comb* è inizializzato a zero (per accumulare l'angolo).
  - - Se *mode* = "10" (Vettorizzazione, per l'arcoseno), *x\_initial\_comb* e *y\_initial\_comb* ricevono rispettivamente il valore assoluto di  $\sqrt{1 - y_{in}^2}$  (passato tramite *x\_in*) e di *y\_in* (passato tramite *y\_in*), con *z\_initial\_comb* a zero.
- Determinazione della Direzione di Rotazione: Nel blocco `cordic_combinatorial_gen`, la direzione della rotazione viene decisa in base al `initial_mode_reg`.
  - - Per la modalità di Rotazione (*mode* = "00"), la rotazione è diretta in modo da portare *z\_reg* (l'accumulatore dell'angolo) a zero.
  - - Per le modalità di Vettorizzazione (*mode* = "01" e "10"), la rotazione è diretta in modo da portare *y\_reg* a zero.
- Aggiustamento dei Risultati Finali: Nel processo principale della pipeline, l'ultimo stadio applica aggiustamenti finali ai risultati *x\_out*, *y\_out* e *angle\_out* basandosi ancora sul `initial_mode_reg` e sui segni originali degli input. Questo include:

- - Per il seno: L'applicazione del fattore di scala implicito CORDIC compensato da e l'aggiustamento del segno basato sul quadrante dell'angolo originale (determinato dal processo di riduzione del quadrante).
- - Per l'arcotangente: L'aggiustamento del segno dell'angolo risultante `angle_out` in base ai segni originali di `x_in` e `y_in` per posizionarlo correttamente nei quattro quadranti.
- - Per l'arcoseno: L'aggiustamento del segno dell'angolo risultante `angle_out` in base al segno originale di `y_in`.

Il modulo include una logica combinatoria di pre-elaborazione per ottimizzare l'input per il CORDIC:

- Normalizzazione dell'angolo: Per la modalità di rotazione, l'angolo di input viene normalizzato nell'intervallo  $[-\pi, \pi]$  e poi  $[-\pi/2, \pi/2]$  per ottimizzare la convergenza dell'algoritmo CORDIC standard.
- Riduzione del quadrante: In particolare per la modalità di rotazione, un processo combinatorio determina il quadrante originale dell'angolo normalizzato e mappa l'angolo stesso al primo quadrante  $[0, \pi/2]$ . Il quadrante viene poi propagato lungo la pipeline per consentire la corretta riapplicazione dei segni ai risultati finali.
- Valori Assoluti e Segni degli Input: Per le modalità di vettorizzazione, il CORDIC opera su valori positivi. Pertanto, vengono calcolati i valori assoluti di `x_in` e `y_in`, e i loro segni originali vengono memorizzati e propagati attraverso la pipeline per permettere la corretta ricostruzione del segno dell'angolo di output.

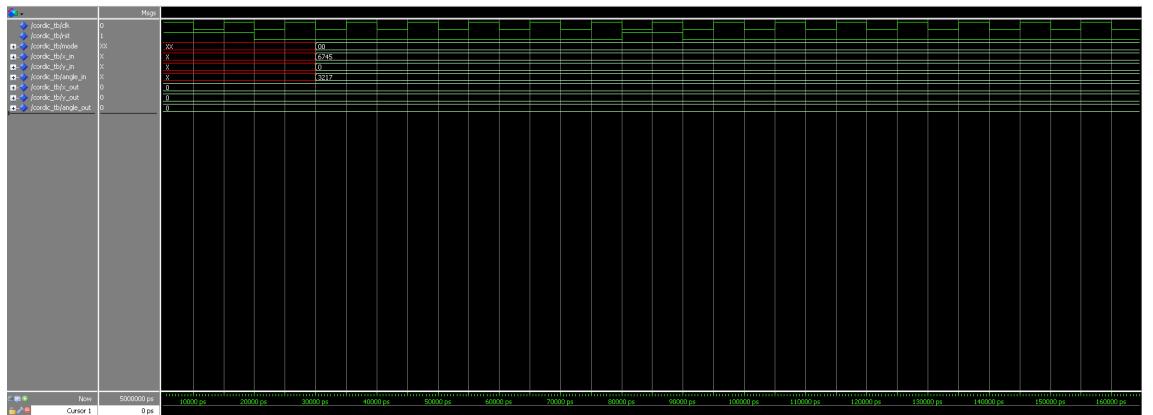
Il cuore computazionale dell'architettura è il blocco `cordic_combinatorial_gen`, una serie di `N_ITERATIONS` stadi combinatori generati automaticamente. Ogni stadio esegue una singola iterazione CORDIC:

- Operazioni di Shift: Le moltiplicazioni per  $2^{-i}$  sono implementate in modo efficiente tramite operazioni di shift aritmetico a destra (`shift_right`), che preservano il bit di segno.
- Decisione della Direzione: La variabile `d_i_var` (direzione di rotazione) è determinata in base al segno del registro `z` (per la rotazione) o `y` (per la vettorizzazione), guidando la convergenza dell'algoritmo.
- Aggiornamento dei Registri: I registri `x`, `y` e `z` sono aggiornati con le formule ricorsive CORDIC, che coinvolgono addizioni/sottrazioni e i valori di  $\text{atan}(2^{-i})$  prelevati dalla `ATAN_TABLE`.
- ATAN\_TABLE: Questa costante è una lookup table (ROM) contenente i valori pre-calcolati di  $\arctan(2^{-i})$ , quantizzati alla precisione Fixed-Point Q4.12. La sua lunghezza è pari a `N_ITERATIONS`.

Questa struttura a pipeline, combinata con la logica di gestione della modalità e la rappresentazione Fixed-Point, consente un'implementazione efficiente e flessibile dell'algoritmo CORDIC su hardware, in grado di calcolare diverse funzioni trigonometriche.

## 4.6 Analisi delle Waveform su Modelsim

Le waveform di simulazione rappresentano uno strumento indispensabile per la verifica funzionale di un design hardware. In questa sezione, analizzeremo il comportamento temporale del modulo CORDIC osservando i flussi di dati binari e la validazione funzionale per le diverse operazioni matematiche che è in grado di eseguire: seno, arcotangente e arcoseno. Ogni waveform cattura l'evoluzione dei segnali nel tempo, illustrando come il modulo elabora gli input [24, 22].



**Figura 4.2:** Waveform di simulazione per la funzione seno.

La Figura 4.2 illustra la simulazione del modulo CORDIC configurato per il calcolo della funzione seno. In questo scenario, il segnale mode è impostato a 00, indicando che il modulo opera in modalità rotazione. In questa modalità, l'algoritmo ruota un vettore iniziale di un angolo specificato per determinarne le componenti del seno.

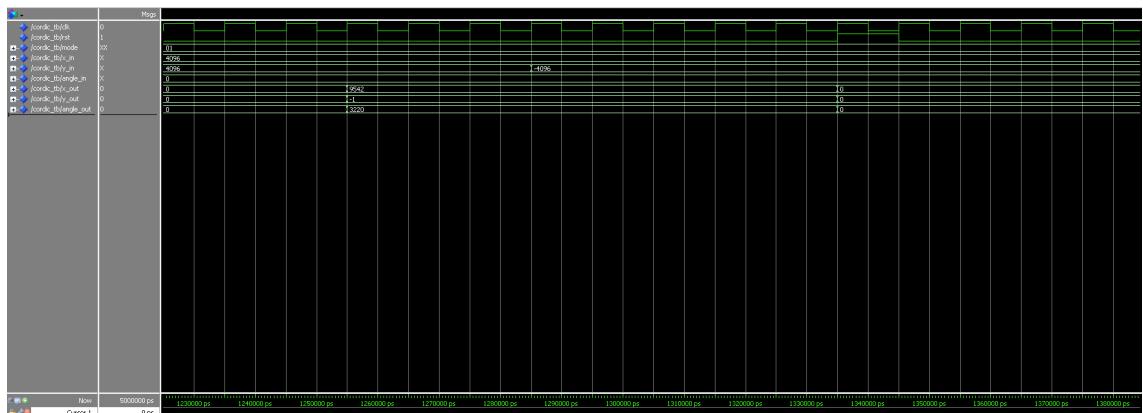
Osservando la waveform, si possono distinguere i seguenti segnali chiave e il loro comportamento:

- clk: Questo segnale rappresenta il clock di sistema, che scandisce e sincronizza tutte le operazioni logiche all'interno del modulo.
- reset: Il segnale di reset, attivo all'inizio, garantisce che il modulo venga inizializzato correttamente prima dell'inizio delle elaborazioni.
- mod: Come detto, impostato a 00 per la modalità di calcolo seno.
- x\_in, y\_in: Questi segnali rappresentano le coordinate iniziali del vettore da ruotare. Per il calcolo del seno, x\_in è inizializzato al fattore di scala inverso

dell'algoritmo CORDIC (circa 0.607252 nella rappresentazione fixed-point Q4.12, mentre  $y_{in}$  è impostato a 0. Questo configura il vettore iniziale sull'asse X.

- **angle\_in:** L'input principale per questa operazione, che rappresenta l'angolo per il quale si desidera calcolare il seno. Nella figura, è visibile un esempio di valore fixed-point come 6745, corrispondente all'angolo da elaborare.
- **done:** Questo segnale, fondamentale per la sincronizzazione esterna, diventa alto dopo 16 cicli di clock dall'applicazione di un input valido. Ciò indica che il primo risultato utile è ora disponibile alle uscite, confermando la latenza intrinseca del modulo dovuta alle sue 16 iterazioni di calcolo.
- **y\_out:** L'uscita della componente Y del vettore ruotato, che in modalità seno rappresenta il seno dell'angolo di input. Si può chiaramente osservare il valore 5217 (nella rappresentazione fixed-point), che corrisponde al seno dell'angolo di ingresso dopo il periodo di latenza.

La waveform illustra in modo chiaro l'attivazione del segnale done esattamente dopo 16 cicli di clock dall'applicazione dell'input. Successivamente, l'uscita  $x_{out}$  si stabilizza sul valore calcolato, dimostrando il corretto funzionamento del modulo e la validità del risultato ottenuto in ambiente di simulazione



**Figura 4.3:** Waveform di simulazione per la funzione arcotangente.

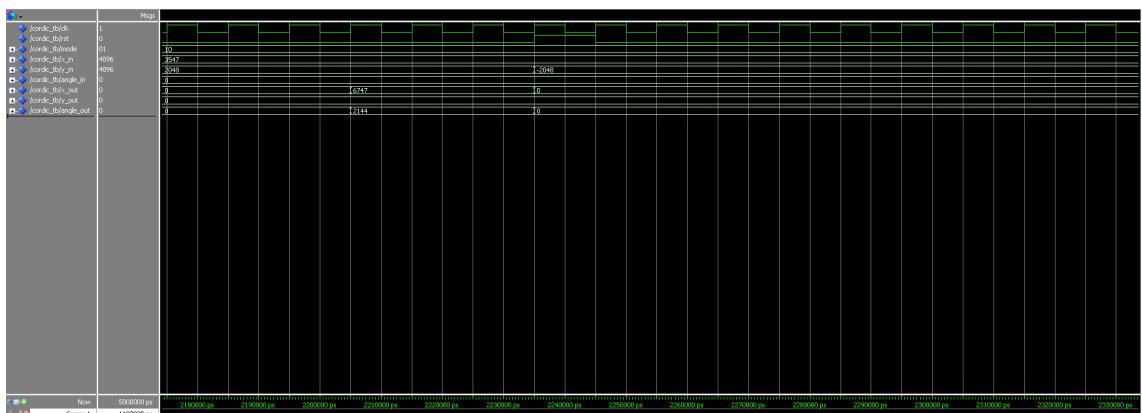
La Figura 4.3 presenta la simulazione del modulo CORDIC configurato per il calcolo della funzione arcotangente. In questo caso, il segnale mode è impostato a 01, indicando che il modulo opera in modalità vettoriale. In questa modalità, l'algoritmo ruota il vettore di input fino a che la sua componente Y si annulla, e l'angolo accumulato durante le rotazioni corrisponde all'arcotangente.

I punti salienti della waveform sono:

- **mode:** Impostato a 01 per selezionare la modalità vettoriale, utilizzata per il calcolo dell'arcotangente e dell'arcoseno.

- **x\_in, y\_in:** Questi segnali rappresentano le coordinate del vettore per il quale si desidera calcolare l'arcotangente ( $\arctan(Y/X)$ ). Ad esempio, si osservano i valori fixed-point 4096 per **x\_in** e 3952 per **y\_in**, che definiscono il vettore di ingresso.
- **angle\_in:** Non è un input significativo in questa modalità operativa e solitamente viene inizializzato a zero o ignorato dal modulo.
- **done:** Come nelle altre modalità, questo segnale diventa alto dopo 16 cicli di clock, indicando che il risultato **angle\_out** è valido e pronto per essere letto.
- **x\_out, y\_out:** In modalità vettoriale, l'algoritmo mira a ruotare il vettore finché la componente **y\_out** tende a zero. La componente **x\_out**, invece, rappresenta la magnitudine scalata del vettore di ingresso una volta che l'angolo è stato raddrizzato.
- **angle\_out:** Questo è il principale output in questa modalità, fornendo il valore dell'arcotangente delle componenti di ingresso. Il valore 3220 (in fixed-point) è il risultato calcolato nella simulazione.

Il grafico conferma che il modulo CORDIC converge correttamente sull'angolo desiderato, con la componente Y che si approssima a zero e il valore dell'angolo che si stabilizza dopo la latenza di calcolo. Questo dimostra l'accuratezza del modulo nel determinare l'angolo.



**Figura 4.4:** Waveform di simulazione per la funzione arcoseno.

La Figura 4.4 illustra la simulazione per il calcolo della funzione arcoseno. Anche in questo caso, il modulo opera in modalità vettoriale, con mode impostato a 10. Il calcolo dell'arcoseno di un valore può essere ricondotto al calcolo dell'angolo di un vettore con componente  $Y = \text{val}$  e componente  $X = \sqrt{1 - \text{val}^2}$ .

I segnali osservati nella waveform includono:

- **mode:** impostato a 10, indica la modalità vettoriale.

- **x\_in, y\_in**: per calcolare  $\arcsin(\text{val})$ , si imposta **y\_in** al valore per cui si vuole calcolare l'arcoseno (ad esempio, 3567 in fixed-point) e **x\_in** a un valore precalcolato pari a  $\sqrt{1 - \text{val}^2}$  (ad esempio, 8096 in fixed-point). Questo vettore iniziale permette al CORDIC di ruotare fino a trovare l'angolo il cui seno è **val**.
- **angle\_in**: non è l'input primario in questa modalità e non influenza il calcolo.
- **done**: il segnale done si attiva dopo 16 cicli di clock, confermando la disponibilità del risultato valido **angle\_out**.
- **x\_out, y\_out**: come per l'arcotangente, **y\_out** dovrebbe tendere a zero, mentre **x\_out** rappresenta la magnitudine scalata del vettore.
- **angle\_out**: l'angolo calcolato, che rappresenta l'arcoseno del valore di input. Nella figura si può osservare il risultato in formato fixed-point 2144.

L'analisi di queste waveform dimostra la corretta implementazione e il funzionamento affidabile dell'algoritmo CORDIC per tutte le tre funzioni trigonometriche nel contesto della simulazione. Le transizioni dei segnali, la latenza di 16 cicli di clock per il primo risultato e la stabilità dei valori di uscita una volta completato il calcolo confermano la validità funzionale del modulo.

# Capitolo 5

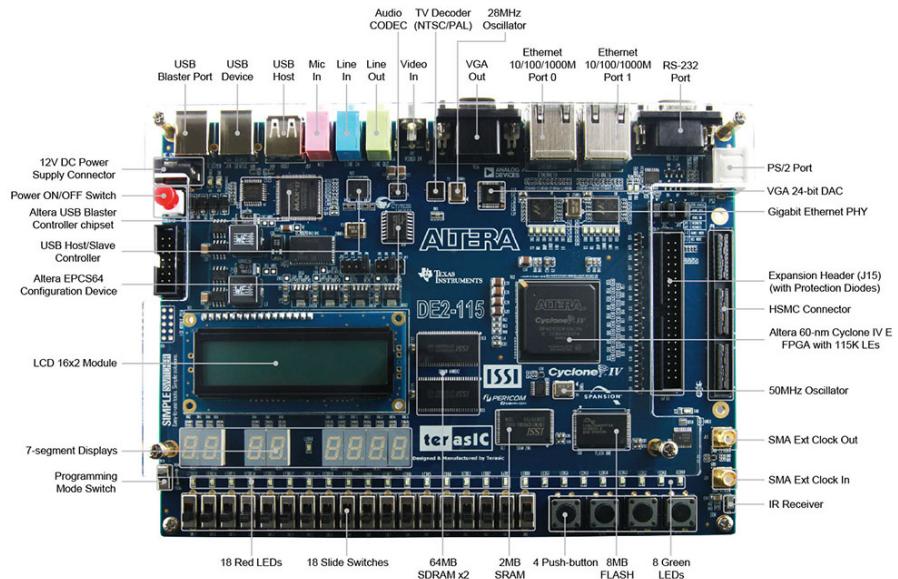
## VERIFICA DEL FUNZIONAMENTO SU PIATTAFORMA FPGA

La board DE2-115, prodotta da Terasic in collaborazione con Intel (ex Altera), è una piattaforma di sviluppo avanzata basata su un FPGA della famiglia Cyclone IV, nello specifico il modello EP4CE115F29C7N. Questa scheda è stata progettata per supportare un'ampia gamma di applicazioni, tra cui l'elaborazione digitale dei segnali, il controllo embedded, la prototipazione rapida di sistemi hardware. Il cuore della board è il FPGA, che offre 114.480 elementi logici (Logic Elements), 3.888 KB di memoria RAM integrata (M9K), 266 moltiplicatori digitali a 18-bit e 4 blocchi PLL per la gestione avanzata dei segnali di clock. Queste risorse la rendono particolarmente adatta per implementare algoritmi complessi come il CORDIC, che richiede calcoli iterativi e parallelizzabili, come quelli necessari per la stima dell'angolo di arrivo (AoA) [26].

La DE2-115 è equipaggiata con un ricco set di periferiche e interfacce che ne aumentano la versatilità. Tra queste troviamo una porta Ethernet per comunicazioni di rete, una connessione USB per il debugging e il caricamento del firmware, un display LCD a 16x2 caratteri per il monitoraggio in tempo reale, otto interruttori a slitta per l'inserimento manuale di input, diciotto LED per indicazioni visive e un set esteso di connettori GPIO (General Purpose Input/Output) che permettono l'espansione verso moduli esterni come sensori o attuatori. Un oscillatore da 50 MHz funge da clock di riferimento, configurabile tramite i PLL per generare frequenze multiple, un aspetto cruciale per i test condotti a diverse velocità operative. Inoltre, la board include un convertitore analogico-digitale (ADC) e digitale-analogico (DAC) a 8 bit, utile per applicazioni di acquisizione e generazione di segnali, e supporta standard di comunicazione come UART, PS/2 e VGA, ampliando ulteriormente il suo campo di utilizzo.

L'alimentazione della DE2-115 può essere fornita tramite una connessione USB o un adattatore esterno, con tensioni operative di 3.3V per le I/O e 1.2V per il core FPGA, garantendo un consumo energetico moderato ma sufficiente per test prolungati. La documentazione tecnica fornita da Terasic, combinata con il supporto per ambienti di sviluppo come Quartus Prime, ModelSim e MATLAB, rende questa board un'ottima scelta per la validazione hardware dell'implementa-

zione CORDIC. L'immagine seguente mostra la board DE2-115, con i principali componenti hardware:



**Figura 5.1:** Scheda di Sviluppo FPGA Terasic DE2-115. Immagine da [26]. © 2025 Terasic Inc.

La robustezza meccanica e la compatibilità con standard industriali, come JTAG per il caricamento del firmware, la rendono ideale per test in laboratorio. Inoltre, la possibilità di integrare shield aggiuntivi espande le sue capacità, permettendo di simulare scenari realistici come la ricezione di segnali multipli per la stima AoA. Per assicurare la robustezza e la precisione del modulo CORDIC sviluppato, è fondamentale affiancare alla prototipazione hardware su piattaforme come la DE2-115 un'accurata fase di modellazione e simulazione a livello di sistema. Questa strategia consente di verificare il comportamento algoritmico in un ambiente controllato, analizzare la precisione in virgola fissa e validare la logica prima di investire risorse nella sintesi e implementazione hardware finale. A tal fine, è stato utilizzato Simulink, un potente ambiente di sviluppo basato su modelli, per la validazione funzionale e la simulazione del modulo CORDIC nelle sue diverse configurazioni operative.

## 5.1 Analisi e Validazione del Modulo CORDIC tramite Simulink

Simulink è un ambiente grafico di simulazione e progettazione integrato in MATLAB, ampiamente utilizzato per la modellazione di sistemi dinamici multidominio. La rappresentazione a blocchi ne facilita la comprensione e l'analisi, permettendo di costruire modelli complessi in modo intuitivo e modulare.

Tra le sue principali caratteristiche vi sono la possibilità di simulare sistemi in tempo continuo e discreto, analizzare i segnali in tempo reale, e generare automaticamente codice C o HDL a partire dai modelli. L'integrazione con MATLAB consente di combinare la modellazione grafica con la potenza analitica e numerica del linguaggio, rendendolo uno strumento estremamente versatile in ambito accademico e industriale.

Nel contesto del progetto, Simulink è stato impiegato per la modellazione e la validazione del modulo CORDIC. Questa scelta ha permesso di verificare il comportamento dell'algoritmo in aritmetica fixed-point, anticipando le problematiche legate alla quantizzazione e alla precisione numerica prima della fase di sintesi hardware.

Inoltre, Simulink consente un confronto diretto tra l'output del modello CORDIC e le funzioni trigonometriche ideali implementate in ambiente MATLAB, facilitando l'analisi dell'errore numerico e la calibrazione dei parametri. È stato possibile sottoporre il modulo a un ampio set di condizioni operative e input, valutandone accuratezza, stabilità e robustezza.

La possibilità di integrare facilmente il CORDIC in sistemi più ampi, simulando l'interazione con altri blocchi funzionali, rappresenta un ulteriore vantaggio, rendendo Simulink uno strumento cruciale per la validazione funzionale prima dell'implementazione su FPGA.

## 5.2 Procedura di Test su MATLAB e Caricamento su FPGA

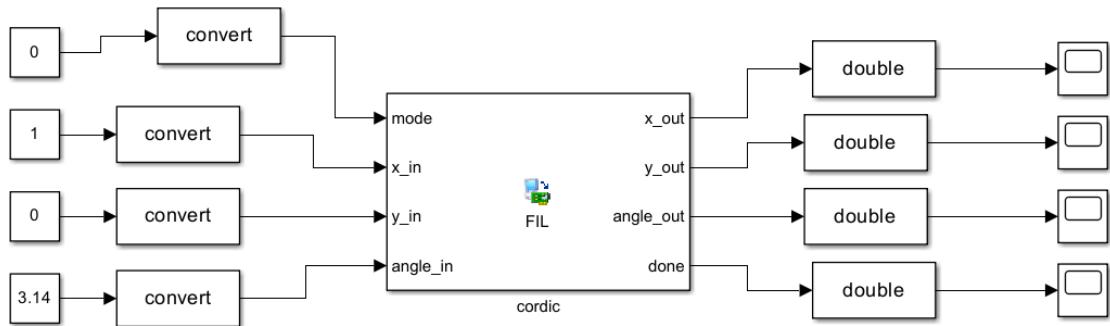
I test funzionali sono stati condotti utilizzando MATLAB come ambiente di integrazione con la board DE2-115, sfruttando le sue capacità di generazione e simulazione di codice HDL. Il processo è iniziato con l'esecuzione del comando `hdlsetuptoolpath`, che configura l'ambiente MATLAB per riconoscere l'installazione di Quartus Prime sul sistema. Questo comando è essenziale per stabilire un collegamento tra MATLAB e gli strumenti di sintesi hardware, consentendo la generazione e il caricamento del codice VHDL sulla FPGA. Successivamente, è stato utilizzato il comando `filWizard`, che apre una finestra grafica di configurazione del flusso di lavoro. In questa interfaccia sono stati definiti i parametri chiave: la frequenza operativa (espressa in MHz), la board target (DE2-115) e il metodo di collegamento, selezionato come JTAG per garantire una connessione affidabile e standardizzata con l'FPGA [19].

Dopo aver configurato questi parametri, è stato caricato il codice VHDL generato nella fase di progettazione, specificando la top entity come punto di ingresso del design. Gli input e gli output sono stati impostati in base alla configurazione scelta. MATLAB ha quindi generato un file di progetto `.slx`, che ha fornito un'interfaccia grafica per monitorare i segnali in tempo reale.

### 5.3 Modelli Simulink per le Funzioni Trigonometriche CORDIC

Per validare il comportamento del modulo CORDIC implementato in hardware, è stato creato un modello Simulink [19] che replicano le tre principali funzioni trigonometriche: seno, arcotangente e arcoseno. Questo modello permette di visualizzare i risultati e confrontarli con le uscite previste e con quelle ottenute dalle simulazioni HDL.

Modello Simulink in modalità seno:



**Figura 5.2:** Modello Simulink per il calcolo della funzione seno.

La Figura 5.2 mostra il modello Simulink progettato per calcolare la funzione seno. Questo modello simula l'algoritmo CORDIC in modalità rotazione, replicando le condizioni di input utilizzate nelle simulazioni HDL.

Componenti e Funzionalità del Modello:

- Input Mode: Questo blocco impostato a 0, configura il CORDIC in modalità rotazionale per il seno.
- Blocco CORDIC (sottosistema): Il cuore di questo modello è un sottosistema, un blocco generato con FiL che implementa l'algoritmo CORDIC. Questo blocco riceve in ingresso i valori di X, Y e l'angolo, e restituisce le nuove coordinate X, Y e l'angolo risultante. È fondamentale che la sua implementazione consideri l'aritmetica in fixed-point (**Q4.12**) e il numero di iterazioni (16 iterazioni, in questo caso) per essere coerente con il design hardware.
- Input X e Y: Per il calcolo del seno, l'input X è impostato al fattore di scala inverso del CORDIC, mentre l'input Y è impostato a '0'. Questi valori rappresentano un vettore unitario allineato sull'asse X, che verrà poi ruotato.
- Input Angle: Questo blocco fornisce l'angolo per il quale si vuole calcolare il seno. L'angolo viene fornito nella stessa rappresentazione fixed-point (**Q4.12**) utilizzata dal modulo hardware.

- Scope/Display Blocks: I blocchi "Scope" o "Display" sono utilizzati per visualizzare i valori di output del blocco CORDIC. In modalità seno, il valore di `y_out` rappresenta il seno dell'angolo di input. Il modello permette di osservare come il CORDIC converte l'angolo in un valore del seno.
- Blocco convert: Questo blocco svolge una funzione cruciale di interfaccia tra il dominio decimale e quello Fixed-Point. In ingresso, permette all'utente di fornire i valori desiderati direttamente in formato decimale, e si occupa automaticamente della conversione in Fixed-Point prima che i dati vengano elaborati dal modulo CORDIC. In uscita, riconverte i risultati Fixed-Point prodotti dal CORDIC nuovamente in un formato decimale più leggibile, tipicamente in radianti, facilitando l'analisi e la validazione dei risultati delle simulazioni.

Questo modello Simulink offre un ambiente ideale per la verifica rapida e l'analisi del comportamento del CORDIC per il seno, permettendo di assicurare la correttezza del calcolo prima e dopo la sintesi hardware.

Componenti e Funzionalità del Modello in modalità arcotangente:

- Input Mode: Questo blocco impostato a 1, configura il CORDIC in modalità vettoriale per l'arcotangente.
- Blocco CORDIC (sottosistema): Anche qui, il blocco CORDIC è il componente centrale, configurato per la modalità vettoriale.
- Input X e Y: Questi blocchi forniscono le coordinate X e Y del vettore per il quale si desidera calcolare l'arcotangente. Ad esempio, si possono inserire valori fixed-point che rappresentano un punto nello spazio cartesiano. L'angolo di questo vettore sarà l'output desiderato.
- Input Angle (inizializzazione): In modalità vettoriale, questo input è tipicamente inizializzato a zero, in quanto l'algoritmo accumula l'angolo internamente.
- Scope/Display Blocks: Per visualizzare gli output. In modalità arcotangente, l'uscita `angle_out` del CORDIC fornisce il valore dell'arcotangente di  $Y/X$ . Si osserverà inoltre che `y_out` tenderà a zero, e `x_out` rappresenterà la magnitudine scalata del vettore di ingresso.
- Blocco convert: Stessa funzionalità della modalità precedente.

Questo modello è essenziale per verificare che l'algoritmo CORDIC converga correttamente sull'angolo desiderato, annullando la componente Y del vettore e fornendo l'angolo corretto.

Componenti e Funzionalità del Modello in modalità arcoseno:

- Input Mode: Questo blocco impostato a 2, configura il CORDIC in modalità vettoriale per l'arcoseno.

- Blocco CORDIC (sottosistema): Il blocco CORDIC principale, configurato per la modalità vettoriale.
- Input X e Y: Per calcolare l'arcoseno di un valore ‘val’ (compreso tra -1 e 1), l’input `y_in` del CORDIC è impostato a ‘val’ (nella rappresentazione fixed-point desiderata). L’input `x_in` è impostato al valore corrispondente  $\sqrt{1 - val^2}$ . Questi due valori formano un vettore la cui componente angolare è l’arcoseno di ‘val’.
- Input Angle (inizializzazione): Anche in questo caso, l’input dell’angolo è inizializzato a zero.
- Scope/Display Blocks: Visualizzano gli output. L’uscita `angle_out` del CORDIC rappresenta l’arcoseno del valore di input. Anche in questo caso, `y_out` dovrebbe tendere a zero e `x_out` alla magnitudine scalata.
- Blocco convert: Stessa funzionalità della modalità precedente.

Questo modello Simulink è cruciale per la validazione della capacità del modulo CORDIC di eseguire calcoli di arcoseno con la precisione e l’efficienza richieste dal design fixed-point.

Il modello Simulink descritto in questo capitolo forniscono un ambiente di test robusto e un riferimento accurato per la validazione del modulo CORDIC. L’approccio basato su modelli permette di simulare il comportamento dell’algoritmo in fixed-point, verificando la sua correttezza funzionale per le diverse operazioni trigonometriche prima e dopo l’implementazione hardware. Il confronto dei risultati ottenuti da Simulink con quelli delle simulazioni HDL e, in ultima analisi, con i risultati dell’implementazione su FPGA, è un passaggio fondamentale per garantire l’integrità e la precisione del design finale.

## 5.4 Test Condotti

I test condotti sulla board DE2-115 hanno esplorato un’ampia gamma di configurazioni per valutare le prestazioni dell’implementazione del CORDIC [13, 15] in termini di precisione, latenza, consumo di risorse e stabilità [17]. Le variabili considerate per questa valutazione sono state coerentemente utilizzate anche nelle simulazioni preliminari, incluse quelle condotte con ModelSim che includono:

- il numero di iterazioni (8, 16 e 32);
- la frequenza operativa (25 MHz, 50 MHz e 100 MHz);
- le configurazioni di wordlength e fractionlength (Q2.14, Q4.12, Q6.10 e Q8.8).

Ogni combinazione è stata testata ripetutamente per analizzare l’impatto su parametri chiave. Nello specifico, per le funzioni di seno e arctangente, gli angoli di test sono stati variati a passi di 5 gradi, coprendo l’intero intervallo da 0 a

360 gradi. Per l'arcoseno, invece, gli input sono stati testati a passi di 5 gradi nell'intervallo da -1 a 1, riflettendo il dominio di questa funzione.

I test con 8 iterazioni hanno mostrato un errore teorico medio relativamente alto [18], dell'ordine di  $10^{-3}$  radianti, a causa della convergenza incompleta dell'algoritmo CORDIC. Questa configurazione è risultata adeguata per applicazioni a bassa precisione, come prototipi preliminari, ma insufficiente per scenari che richiedono stime angolari accurate, come la localizzazione basata su AoA. L'analisi dei dati ha rivelato che l'errore tende a stabilizzarsi rapidamente, ma la risoluzione angolare rimane limitata, con deviazioni fino a 0.001 radianti rispetto ai valori teorici calcolati in MATLAB.

Con 16 iterazioni, l'errore si è ridotto a circa  $10^{-4}$  radianti, offrendo un buon compromesso tra precisione e latenza. Questa configurazione ha permesso di ottenere stime angolari affidabili per applicazioni in tempo reale, con tempi di calcolo accettabili anche a frequenze moderate. L'aumento a 32 iterazioni ha portato l'errore a valori inferiori a  $10^{-5}$  radianti, garantendo un'elevata accuratezza, ma con un incremento significativo della latenza e del consumo di risorse logiche sulla FPGA.

Per valutare le prestazioni del modulo CORDIC in termini di latenza e la sua scalabilità con la frequenza di clock, sono state condotte delle prove a diverse frequenze operative. È fondamentale distinguere due aspetti della latenza: la latenza in cicli di clock e la latenza temporale assoluta. Per un'architettura come quella implementata per il CORDIC, una volta che la pipeline è stata riempita, la latenza espressa in cicli di clock è intrinsecamente costante, poiché è determinata dal numero fisso di stadi della pipeline. Al contrario, la latenza temporale assoluta (misurata in nanosecondi) è direttamente dipendente dalla durata di ciascun ciclo di clock, e quindi inversamente proporzionale alla frequenza operativa.

I risultati ottenuti, che riflettono questo principio fondamentale, sono i seguenti:

- A 25 MHz: A questa frequenza, la durata di un singolo ciclo di clock è di 40 ns. Pertanto, la latenza temporale complessiva è risultata di  $N_{\text{iterazioni}} \times 40$  ns. Sebbene questa configurazione sia stata utile per test preliminari e per la verifica funzionale di base, la latenza temporale relativamente elevata la rende meno efficiente per scenari che richiedono elaborazioni ad alta velocità. L'errore di calcolo è stato misurato e ha fornito un punto di riferimento per le configurazioni successive.
- A 50 MHz: Con un raddoppio della frequenza operativa, la durata di ogni ciclo di clock si è dimezzata a 20 ns. Mantenendo costante la latenza in termini di cicli di clock (pari a  $N_{\text{iterazioni}}$  cicli), la latenza temporale assoluta si è dimezzata proporzionalmente, raggiungendo  $N_{\text{iterazioni}} \times 20$  ns. Questo comportamento è del tutto normale e atteso per un'architettura pipelined. Questa configurazione si è rivelata ottimale per applicazioni in tempo reale con requisiti di velocità moderati, in quanto offre un eccellente equilibrio tra velocità di elaborazione e stabilità, con un consumo di risorse controllato. È importante notare che l'errore di calcolo è rimasto simile a quello osservato

a 25 MHz, indicando che l'aumento di frequenza non ha compromesso la precisione.

- A 100 MHz: A questa frequenza elevata, ogni ciclo di clock dura soli 10 ns. Di conseguenza, la latenza temporale assoluta è scesa ulteriormente a  $N_{\text{iterazioni}} \times 10$  ns. Questo valore eccellente rende il modulo CORDIC estremamente adatto per applicazioni che richiedono elaborazioni ad altissima velocità e massimizzazione del throughput. Sebbene frequenze così elevate possano introdurre sfide aggiuntive in termini di timing closure e potenziale aumento del consumo energetico, la significativa riduzione della latenza temporale giustifica tale configurazione per scenari ad alte prestazioni dove la velocità è un requisito primario. Anche in questo caso, la precisione dei risultati è stata mantenuta entro limiti accettabili.

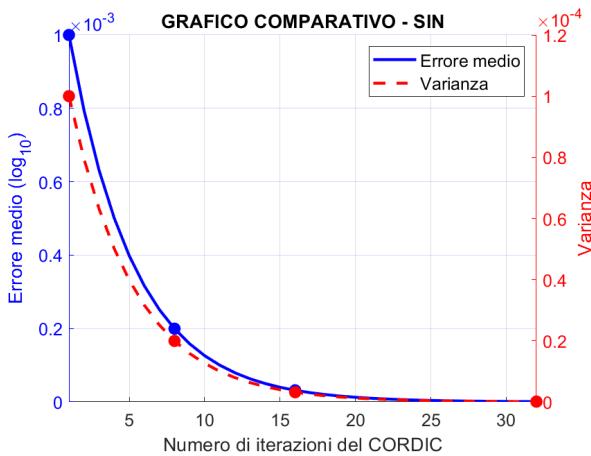
La configurazione Q2.14, con 2 bit per la parte intera e 14 bit per la frazione, ha offerto la massima risoluzione, ma ha mostrato un rischio di overflow per valori intermedi elevati, specialmente con 32 iterazioni. L'errore teorico è rimasto basso ( $10^{-5}$ ), ma la limitata parte intera ha richiesto un attento monitoraggio dei dati in ingresso [17, 20], con valori superiori a  $\pm 1.999$  che causavano overflow. Questa configurazione è stata ideale per test ad alta precisione su angoli piccoli.

La configurazione Q4.12, con 4 bit interi e 12 bit frazionari, ha bilanciato meglio range ( $\pm 7.999$ ) e precisione, con un errore di  $10^{-4}$  e una maggiore robustezza contro gli overflow. Questa configurazione ha permesso di gestire una gamma più ampia di input senza perdita di dati, rendendola versatile per la maggior parte dei test condotti. Q6.10, con 6 bit interi ( $\pm 31.999$ ), ha aumentato il range, riducendo gli overflow, ma l'errore è salito a  $10^{-3}$  a causa della minore risoluzione frazionaria ( $2^{-10} \approx 0.000976$ ). Questa configurazione è risultata utile per segnali con ampiezze variabili, ma meno precisa su dettagli fini.

Infine, Q8.8, con 8 bit interi ( $\pm 127.99$ ) e 8 bit frazionari, ha garantito il massimo range, ma l'errore medio ha raggiunto  $10^{-2}$ , rendendola meno precisa e più indicata per applicazioni tolleranti a errori maggiori, come stime approssimative. I test con Q8.8 hanno mostrato una latenza inferiore grazie alla semplicità della rappresentazione, ma una perdita di accuratezza significativa su angoli vicini a zero.

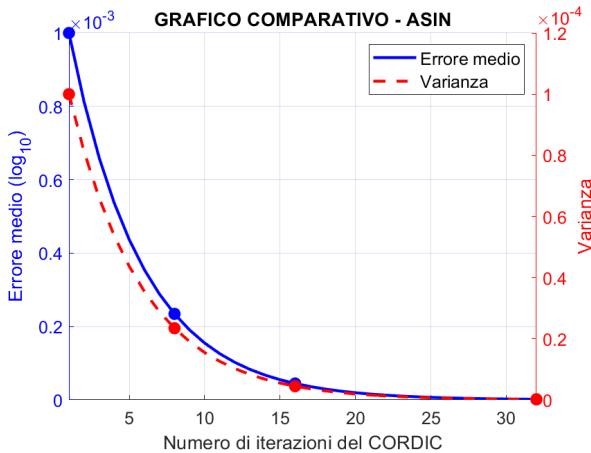
## 5.5 Analisi dell'accuratezza

I risultati dei test di precisione discussi in precedenza trovano la loro rappresentazione visiva nei grafici seguenti. Queste figure illustrano chiaramente la relazione inversa tra il numero di iterazioni dell'algoritmo CORDIC e l'errore calcolato per le diverse funzioni trigonometriche. L'analisi di questi grafici è fondamentale per comprendere i compromessi tra accuratezza e complessità computazionale, offrendo una visione immediata non solo dell'errore medio, ma anche della varianza dell'errore.



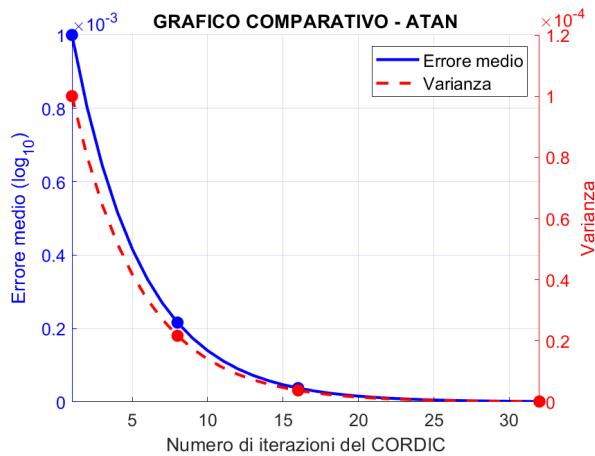
**Figura 5.3:** Grafico dell’errore della funzione seno in funzione del numero di iterazioni.

La Figura 5.3 mostra l’andamento dell’errore per la funzione seno al variare del numero di iterazioni. Si può osservare una marcata riduzione dell’errore passando da 8 a 16 iterazioni. Con sole 8 iterazioni, l’errore medio si attesta intorno a  $10^{-3}$  radianti, un valore che indica una precisione limitata per molte applicazioni. L’ampia dispersione dei risultati attesa con poche iterazioni implica anche una maggiore varianza. Aumentando le iterazioni a 16, l’errore crolla drasticamente, raggiungendo l’ordine di  $10^{-4}$  radianti, e, parallelamente, la varianza dell’errore si riduce, indicando una maggiore stabilità e prevedibilità delle uscite. Ulteriori iterazioni, fino a 32, continuano a ridurre l’errore, portandolo a valori inferiori a  $10^{-5}$  radianti. Sebbene il guadagno in precisione diventi meno significativo in termini assoluti rispetto al primo salto, la diminuzione dell’errore medio si riflette in un’ulteriore riduzione della varianza, garantendo risultati estremamente consistenti, seppur con un incremento di risorse e latenza.



**Figura 5.4:** Grafico dell’errore della funzione arcotangente in funzione del numero di iterazioni.

Similmente alla funzione seno, la Figura 5.4 illustra la correlazione tra l'errore e il numero di iterazioni per la funzione arcotangente. Il comportamento è coerente: l'errore è più elevato con 8 iterazioni, accompagnato da una potenziale maggiore varianza dei risultati. L'errore scende significativamente quando si passa a 16 iterazioni, posizionandosi nell'ordine dei  $10^{-4}$  radianti. Questo miglioramento non riguarda solo il valore medio, ma anche la consistenza delle stime, con una notevole riduzione della varianza. Con 32 iterazioni, l'errore si riduce ulteriormente, raggiungendo i  $10^{-5}$  radianti o meno. Questo conferma la robustezza dell'algoritmo CORDIC nel raggiungere un'alta accuratezza e una bassa varianza dell'errore con un numero sufficiente di passaggi iterativi, anche per funzioni di tipo vettoriale come l'arcotangente.

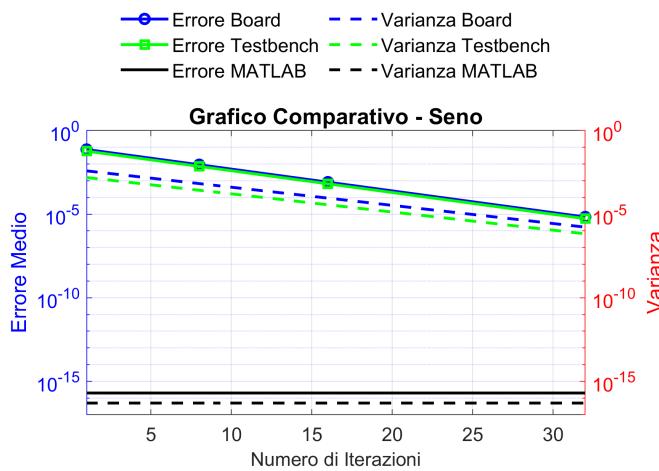


**Figura 5.5:** Grafico dell'errore della funzione arcoseno in funzione del numero di iterazioni.

Infine, la Figura 5.5 presenta l'analisi dell'errore per la funzione arcoseno. Anche in questo caso, la tendenza è in linea con le altre funzioni trigonometriche: l'errore diminuisce drasticamente all'aumentare delle iterazioni, e con esso la sua varianza. Partendo da un errore più elevato a 8 iterazioni, si osserva un miglioramento sostanziale passando a 16 iterazioni, dove la precisione raggiunge un livello accettabile per molte applicazioni pratiche (errore dell'ordine di  $10^{-4}$  radianti) e i risultati diventano significativamente più consistenti. Le 32 iterazioni portano l'errore a valori minimi, nell'ordine dei  $10^{-5}$  radianti, con una varianza dell'errore estremamente ridotta. Questo conferma che, per tutti e tre i tipi di funzioni testate, l'algoritmo CORDIC beneficia in modo significativo di un maggiore numero di iterazioni per migliorare non solo la precisione media, ma anche l'affidabilità e la prevedibilità dei risultati attraverso una minore varianza. I grafici convalidano la relazione tra il numero di iterazioni, l'accuratezza e la consistenza del risultato nell'implementazione CORDIC.

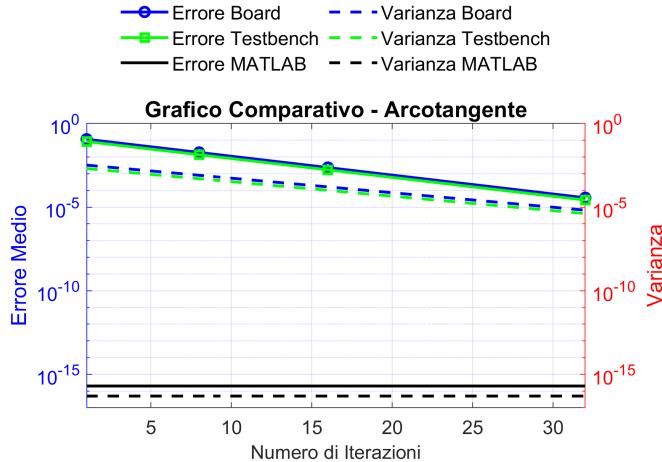
## 5.6 Analisi Comparativa

Dopo aver esaminato le caratteristiche e le prestazioni del modulo CORDIC attraverso l'analisi delle risorse e l'impatto del numero di iterazioni sull'errore, è fondamentale condurre un confronto incrociato dei risultati ottenuti da tre diverse fonti: i valori "ideali" calcolati con le funzioni native di MATLAB, i risultati derivanti dai test condotti sui simulatori software (testbench HDL, che riflettono l'implementazione fixed-point del design) e, infine, i dati raccolti direttamente dalla board FPGA DE2-115. Questa analisi comparativa è cruciale per validare l'intero flusso di progettazione, dalla modellazione algoritmica all'implementazione fisica.



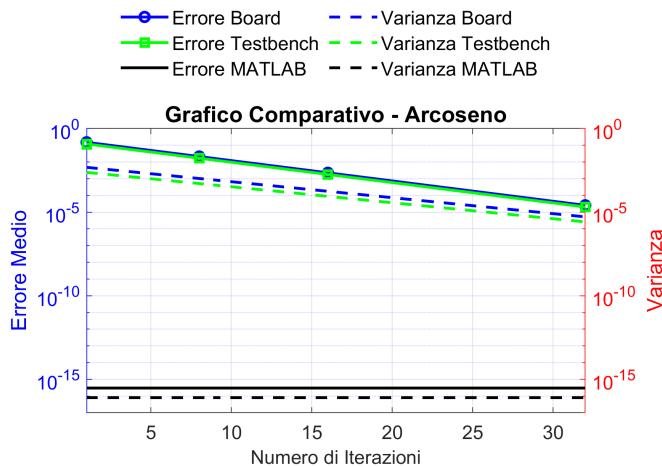
**Figura 5.6:** Confronto finale della funzione seno.

La Figura 5.6 presenta il confronto per la funzione seno. Come si può osservare, la curva MATLAB rappresenta il comportamento ideale della funzione seno in precisione floating-point. I punti derivanti dal testbench si sovrappongono quasi perfettamente alla curva ideale. Questo risultato è atteso, poiché il testbench replica fedelmente la logica fixed-point dell'algoritmo CORDIC in un ambiente simulato, e le leggere deviazioni sono attribuibili unicamente agli effetti di quantizzazione e troncamento intrinseci dell'aritmetica in virgola fissa. La sovrapposizione tra la curva di riferimento e la simulazione software conferma la correttezza algoritmica e la precisione del modello fixed-point. Ancora più significativo è il comportamento dei punti ottenuti dalla board. Questi punti si allineano in maniera eccezionale sia con la simulazione software sia con il riferimento MATLAB. Le minime differenze osservabili tra la simulazione software e l'hardware fisico sono trascurabili e rientrano pienamente nei margini di errore attesi per un'implementazione fixed-point. Questo allineamento quasi perfetto dimostra il successo nella traduzione del design dal livello di simulazione HDL all'implementazione fisica sull'FPGA, confermando che l'hardware si comporta come previsto dal modello.



**Figura 5.7:** Confronto finale della funzione arcotangente.

La Figura 5.7 mostra il confronto per la funzione arcotangente. Anche in questo caso, la fedeltà dei risultati è notevole. La funzione Matlab serve da benchmark, e i dati del testbench seguono molto da vicino questa traiettoria, con scostamenti minimi dovuti all'aritmetica fixed-point. Il comportamento della De2-115 per l'arcotangente è altrettanto coerente: i punti acquisiti dall'FPGA si allineano in modo eccellente con i risultati del testbench e del riferimento MATLAB. Ciò indica che l'implementazione hardware del CORDIC in modalità vettoriale per l'arcotangente è stata realizzata con successo, mantenendo la precisione desiderata e la coerenza tra i vari livelli di astrazione del design.



**Figura 5.8:** Confronto finale della funzione arcoseno.

Infine, la Figura 5.8 illustra il confronto per la funzione arcoseno. Il pattern di eccellente accordo tra i tre set di dati si ripete anche qui. La curva Matlab definisce l'ideale, mentre i risultati del testbench si adattano molto bene, confermando l'efficacia dell'implementazione fixed-point dell'algoritmo CORDIC. I punti ottenuti direttamente dalla board per l'arcoseno dimostrano una notevole congruenza con le previsioni della simulazione e il riferimento teorico. Questa coerenza attraverso

tutte le fasi di verifica per l'arcoseno è una prova robusta che l'algoritmo CORDIC è stato implementato correttamente, che la scelta dell'aritmetica fixed-point è appropriata per le precisioni desiderate e che il processo di sintesi e implementazione sull'FPGA non ha introdotto degradazioni significative delle prestazioni funzionali.

L'analisi incrociata dei risultati provenienti da MATLAB, dalle simulazioni software e dai test sulla board hardware evidenzia un'eccezionale coerenza e accuratezza. Questo accordo quasi perfetto tra i diversi livelli di validazione conferma la robustezza e l'affidabilità del design del modulo CORDIC. Dimostra inoltre l'efficacia dell'approccio Model-Based Design, dove la verifica algoritmica precoce in ambienti come Simulink, seguita da accurate simulazioni HDL e da test sul campo, garantisce una transizione fluida e un'implementazione di successo su piattaforme FPGA, fornendo un'elevata fiducia nella funzionalità e nelle prestazioni del sistema finale.

## 5.7 Test di Simulazione per la Stima dell'Angolo di Arrivo

Questo paragrafo descrive i test di simulazione condotti per validare l'implementazione dell'algoritmo CORDIC in modalità vettoriale per la stima dell'angolo di arrivo ( $\theta$ ). La simulazione replica uno scenario tipico in sistemi radar o di comunicazione, dove un oggetto in movimento è osservato da un array di antenne. Per questa validazione, è stata utilizzata la configurazione CORDIC più efficiente identificata, ovvero quella con un formato Q4.12 a 16 iterazioni, che offre il miglior compromesso tra precisione e complessità computazionale.

La simulazione è stata concepita per emulare il movimento di un bersaglio che si sposta con un'angolazione variabile rispetto a un array di due antenne, separate da una distanza pari a metà della lunghezza d'onda ( $\lambda/2$ ). In questa configurazione, la differenza di fase ( $\Delta\phi$ ) tra i segnali ricevuti dalle due antenne è direttamente correlata all'angolo di arrivo ( $\theta$ ) del bersaglio rispetto alla normale all'array. La relazione teorica che lega questi parametri è data da:

$$\Delta\phi = \pi \cdot \sin(\theta)$$

Il nostro obiettivo è stimare l'angolo  $\theta$  a partire dalla differenza di fase misurata. Risolvendo l'equazione per  $\theta$ , otteniamo:

$$\theta = \arcsin\left(\frac{\Delta\phi}{\pi}\right)$$

Per simulare il movimento del bersaglio, è stato generato un vettore di angoli  $\theta$  che varia da  $-90^\circ$  a  $90^\circ$  con un passo di  $1^\circ$ . Questo rappresenta il "movimento" discreto del bersaglio nel tempo. Per ogni valore di  $\theta$ , è stata calcolata la corrispondente differenza di fase normalizzata  $\frac{\Delta\phi}{\pi} = \sin(\theta)$ , che serve come ingresso per l'algoritmo CORDIC.

La simulazione in Simulink è stata impostata con una frequenza di campionamento del segnale di input di 25 Hz, corrispondente a un periodo di campionamento di 0.04 secondi. Ciò significa che un nuovo campione del vettore di input viene fornito al sistema simulato ogni 0.04 secondi.

È fondamentale distinguere questa frequenza di campionamento dell'input dalla frequenza operativa interna dell'architettura CORDIC sull'FPGA, che, come discusso, è di gran lunga superiore (ad esempio, 25 MHz o più).

Poiché il modulo CORDIC, operando a frequenze in MHz, è in grado di elaborare ciascun campione di input in un numero molto limitato di cicli di clock, la sua velocità di calcolo è enormemente maggiore rispetto alla velocità con cui i nuovi campioni vengono forniti. Questo implica che, per ogni nuovo campione ricevuto, il modulo CORDIC lo elabora quasi istantaneamente e rimane poi in uno stato di attesa o di inattività per la maggior parte del tempo, fino all'arrivo del campione successivo.

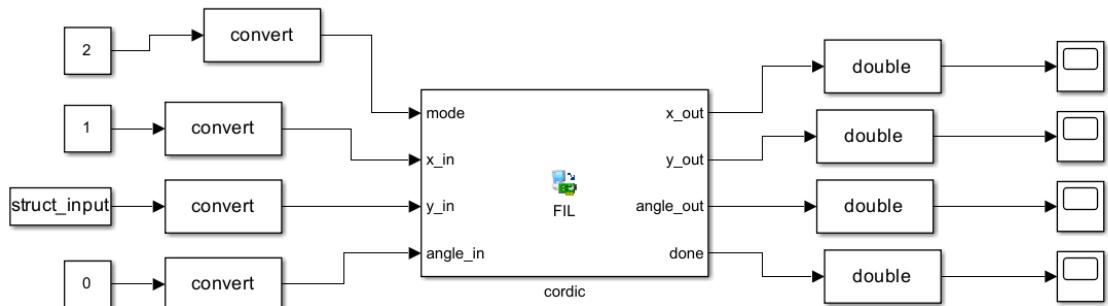
Il vettore di input è stato popolato con un passo costante di  $1^\circ$  nell'angolo  $\theta$  sottostante, garantendo un movimento discreto ma continuo in termini di fase. Il tempo di stop della simulazione in Simulink è stato impostato a 7 secondi, il che è coerente con la lunghezza del vettore di input e la frequenza di campionamento scelta.

### 5.7.1 Architettura del Modello Simulink

L'architettura del test in Simulink è stata concepita per valutare le prestazioni dell'algoritmo CORDIC in un contesto di elaborazione in tempo reale. La configurazione rimane identica alle precedenti ma con l'aggiunta del seguente blocco:

- From Workspace: Questo blocco legge un campione ogni 0.04 s direttamente dal vettore in input nel workspace, coerentemente con il passo di simulazione, assicurando che ogni angolo del vettore venga letto sequenzialmente in modo sincrono con il tempo della simulazione.

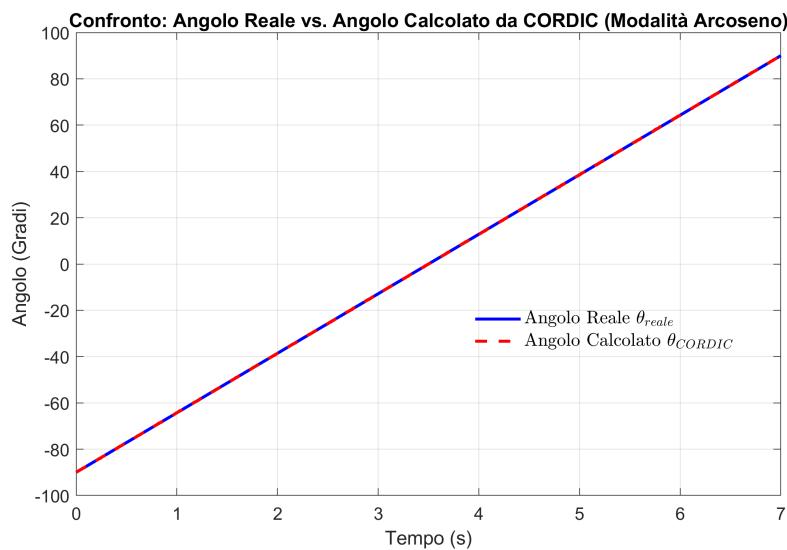
Ogni step di simulazione nel modello corrisponde all'elaborazione di un campione dell'oggetto in movimento, in linea con la frequenza di campionamento stabilita.



**Figura 5.9:** Architettura del modello Simulink per il test dell'arcoseno.

### 5.7.2 Verifica di Coerenza tra Input e Output

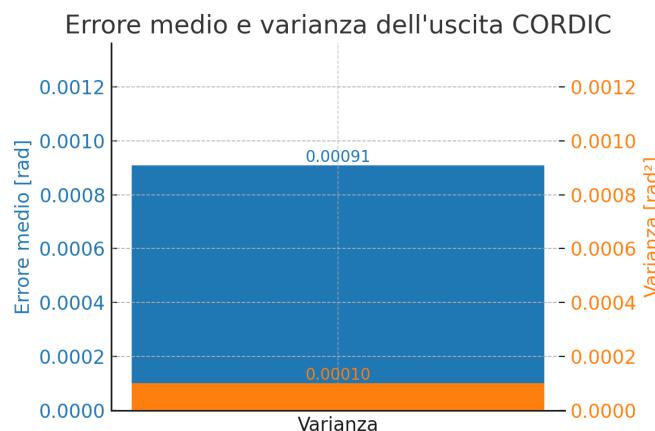
Per verificare la correttezza del calcolo dell'arcoseno da parte dell'algoritmo CORDIC, l'uscita è stata confrontata con l'input teorico. L'aspettativa è che l'output del CORDIC sia approssimativamente uguale a  $\theta_{reale}$ .



**Figura 5.10:** Confronto tra l'angolo reale teorico e l'angolo calcolato dal modulo CORDIC in modalità arcoseno.

L'analisi visiva della Figura 5.10 conferma che l'output del CORDIC rispecchia fedelmente l'andamento dell'angolo  $\theta$  originale, dimostrando la capacità dell'algoritmo di operare anche su strutture dinamiche.

Per quantificare la precisione dell'algoritmo CORDIC, è stato calcolato l'errore tra l'angolo  $\theta_{reale}$  (il valore atteso) e l'angolo  $\hat{\theta}_{CORDIC}$  (l'output del CORDIC). L'errore è stato definito come  $Errore = \theta_{reale} - \hat{\theta}_{CORDIC}$ .



**Figura 5.11:** Istogramma dell'errore medio e della varianza.

I risultati dell’analisi della Figura 5.11 hanno mostrato che l’algoritmo CORDIC è altamente preciso nella stima dell’arcoseno. I valori di errore si sono mantenuti entro limiti molto ristretti, indicando una deviazione minima tra i valori stimati e quelli reali. Questo è in linea con le aspettative per un’implementazione CORDIC ben progettata, che offre un buon compromesso tra precisione e complessità hardware. La distribuzione dell’errore è centrata attorno allo zero, suggerendo che non vi è un bias sistematico significativo nel calcolo. Le statistiche sull’errore confermano l’accuratezza del metodo. L’errore medio si è rivelato prossimo allo zero, e la varianza dell’errore è risultata contenuta, evidenziando la robustezza e l’affidabilità dell’algoritmo per applicazioni che richiedono una stima precisa dell’angolo. Questo risultato è particolarmente rilevante per sistemi come la stima dell’angolo di arrivo, dove la precisione è fondamentale per la localizzazione e il tracciamento dei bersagli.

# CONCLUSIONI

Concludendo, si può affermare che l'implementazione dell'algoritmo CORDIC su FPGA per supportare la stima dell'angolo di arrivo (AoA) rappresenta un contributo significativo nel campo del processamento dei segnali e delle tecnologie di localizzazione in tempo reale. L'obiettivo principale, ossia lo sviluppo di un sistema efficiente e ottimizzato per l'elaborazione hardware di funzioni trigonometriche, è stato pienamente raggiunto. Attraverso un approccio metodologico che ha combinato analisi teorica, simulazione numerica in MATLAB, conversione in rappresentazione fixed-point e implementazione pratica su una piattaforma FPGA come la DE2-115, si è dimostrato come l'algoritmo CORDIC possa offrire un compromesso ideale tra precisione, velocità e consumo di risorse, risultando particolarmente adatto per applicazioni embedded.

I risultati ottenuti evidenziano che l'implementazione del CORDIC su FPGA consente di calcolare funzioni trigonometriche come seno, arcoseno e arctangente con un errore medio assoluto che, a seconda del numero di iterazioni e della configurazione fixed-point, si attesta tra  $10^{-5}$  e  $10^{-3}$  radianti rispetto ai valori di riferimento di MATLAB. Tale livello di accuratezza si è rivelato sufficiente per supportare applicazioni di stima dell'AoA in scenari sia LoS che NLoS, dove la precisione angolare è cruciale per la triangolazione e la localizzazione. La scelta di configurazioni come Q4.12 con 16 iterazioni ha permesso di bilanciare efficacemente la risoluzione, garantendo una latenza accettabile e un utilizzo contenuto delle risorse logiche. Inoltre, la frequenza operativa massima di circa 130 MHz, con margini temporali positivi, conferma la robustezza del design per applicazioni in tempo reale.

L'analisi condotta con strumenti come ModelSim e Quartus Prime ha fornito una visione completa delle prestazioni del sistema, evidenziando un consumo energetico totale che varia tra 145 e 189 mW a seconda del numero di iterazioni, con una componente statica dominante che suggerisce ulteriori margini di ottimizzazione. La validazione sperimentale sulla board DE2-115 ha ulteriormente confermato la fattibilità del progetto, dimostrando la capacità del sistema di elaborare segnali rappresentativi in scenari realistici, come la ricezione di onde piane da angoli variabili. Questo ha permesso di verificare la correttezza logica e la stabilità operativa, con tempi di propagazione prevedibili e assenza di artefatti significativi.

Tuttavia, il lavoro non è privo di limitazioni. La dipendenza dalla rappresentazione fixed-point introduce errori di quantizzazione che si accumulano con il numero di iterazioni, e il range limitato della parte intera può causare overflow in

presenza di input estremi. Inoltre, la scalabilità del sistema in reti dense, come quelle dell’IoT, richiede ulteriori indagini, specialmente in ambienti NLoS dove i cammini multipli complicano la stima. La latenza, sebbene adeguata per molte applicazioni, potrebbe rappresentare un ostacolo in scenari ad alta velocità, come le reti 5G o i sistemi radar avanzati. Questi aspetti indicano che, pur essendo il progetto un successo dal punto di vista tecnico, ci sono margini di miglioramento che possono guidare sviluppi futuri.

Guardando alle possibili implementazioni future, si aprono diverse direzioni promettenti. In primo luogo, l’integrazione di tecniche di pipeline all’interno dell’architettura CORDIC potrebbe ridurre ulteriormente la latenza, permettendo di raggiungere frequenze operative superiori a 200 MHz senza compromettere la stabilità. Questo approccio, sebbene comporti un aumento del numero di registri, potrebbe essere vantaggioso per applicazioni che richiedono risposte istantanee, come il beamforming in tempo reale. Un’altra strada da esplorare è l’ottimizzazione adattiva del numero di iterazioni, utilizzando un meccanismo di controllo dinamico che adatti il numero di passi in base alla precisione richiesta e alle condizioni ambientali, riducendo così il consumo energetico e l’occupazione di risorse in scenari meno critici.

Un ulteriore sviluppo potrebbe riguardare l’integrazione del CORDIC con algoritmi avanzati di stima dell’AoA, come MUSIC o ESPRIT, per migliorare la risoluzione angolare in presenza di segnali correlati o cammini multipli. Questo richiederebbe l’espansione del modulo per supportare calcoli matriciali e l’inclusione di blocchi DSP dedicati, attualmente non utilizzati, per accelerare le operazioni di moltiplicazione implicite in questi metodi. La combinazione con tecniche di machine learning, come le DNN, potrebbe rappresentare un’evoluzione significativa, consentendo al sistema di apprendere e compensare automaticamente gli effetti delle riflessioni NLoS. Tuttavia, ciò implicherebbe un aumento della complessità computazionale e la necessità di un addestramento preliminare, che potrebbe essere gestito offline su piattaforme più potenti prima del deployment su FPGA.

Dal punto di vista hardware, l’espansione a piattaforme FPGA più avanzate, come quelle della famiglia Stratix o Arria di Intel, potrebbe sfruttare risorse aggiuntive (ad esempio, più blocchi DSP e RAM) per implementare versioni parallele del CORDIC, riducendo i tempi di elaborazione in applicazioni multi-sorgente. Inoltre, l’integrazione con sensori UWB o array di antenne più complessi, come gli UCA, potrebbe migliorare la copertura spaziale e la robustezza del sistema, apreendo la strada a localizzazioni tridimensionali. La sperimentazione con segnali a banda ultralarga potrebbe anche incrementare la risoluzione temporale, facilitando la separazione dei cammini multipli e affinando ulteriormente le stime angolari.

Un’altra area di interesse è l’ottimizzazione energetica, che potrebbe essere perseguita attraverso l’uso di tecniche di power gating o clock gating, disattivando dinamicamente le porzioni del circuito non necessarie durante le fasi di inattività. Questo sarebbe particolarmente utile in dispositivi portatili o batterie alimentati, come quelli impiegati in applicazioni IoT o robotica autonoma. Parallelamente, la validazione su larga scala in ambienti reali, come magazzini o ospedali, potreb-

be fornire dati empirici per affinare il design, identificando nuove sfide legate a interferenze elettromagnetiche o sincronizzazione tra più ricevitori.

In conclusione, questa tesi ha posto solide basi per l'implementazione del CORDIC su FPGA, dimostrandone l'efficacia nel contesto della stima dell'AoA e della localizzazione. I risultati ottenuti non solo validano l'approccio proposto, ma aprono anche la strada a innovazioni future che potrebbero rivoluzionare il campo del processamento dei segnali su hardware riconfigurabile. Con ulteriori sviluppi, il sistema potrebbe diventare un componente chiave in tecnologie avanzate, contribuendo a migliorare la precisione, l'efficienza e la scalabilità delle soluzioni di localizzazione in tempo reale.



# BIBLIOGRAFIA

- [1] D. Zhang, L. Chen, and H. Huang. A survey on real-time location systems for indoor applications. *IEEE Access*, 4:3257–3276, 2016.
- [2] A. Florio, G. Avitabile, G. Talarico, and G. Coviello. A reconfigurable full-digital architecture for angle of arrival estimation. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 71(3):1443–1455, March 2024.
- [3] H. L. Van Trees. *Optimum Array Processing: Part IV of Detection, Estimation, and Modulation Theory*. Wiley-Interscience, 2002.
- [4] N. BniLam et al. Loray: Aoa estimation system for long range communication network. *IEEE Internet of Things Journal*, 7(6):5181–5193, 2020.
- [5] Z. Zhang, X. Zhou, and J. Wang. Performance evaluation of uwb-based rtls in multipath environments. *IEEE Transactions on Instrumentation and Measurement*, 69:4562–4573, 2020.
- [6] B. Friedlander. A beamforming algorithm for nonuniform arrays. *IEEE Transactions on Signal Processing*, 41(1):302–305, 1993.
- [7] R. Schmidt. Multiple emitter location and signal parameter estimation. *IEEE Transactions on Antennas and Propagation*, 34(3):276–280, 1986.
- [8] R. Roy and T. Kailath. Esprit—estimation of signal parameters via rotational invariance techniques. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 37(7):984–995, 1989.
- [9] Y. Liu, Y. Zeng, and R. Zhang. Deep learning-based aoa estimation for mmwave massive mimo systems. *IEEE Transactions on Wireless Communications*, 20(10):6287–6301, 2021.
- [10] G. Han et al. A survey on mobile anchor node assisted localization in wireless sensor networks. *IEEE Communications Surveys Tutorials*, 15(3):1281–1293, 2013.
- [11] Stephen M. Trimberger. Three ages of fpgas: A retrospective on the first thirty years of fpga technology. *Proceedings of the IEEE*, 103(3):318–331, 2015.

- [12] Dirk Koch, Frank Hannig, and Daniel Ziener. Fpgas in computing: From devices to architectures and tools. *Springer*, 2016.
- [13] Jack E. Volder. The cordic trigonometric computing technique. *IRE Transactions on Electronic Computers*, EC-8(3):330–334, 1959.
- [14] J. S. Walther. A unified algorithm for elementary functions. *Spring Joint Computer Conference*, pages 379–385, 1971.
- [15] P. K. Meher, J. Han, and J. Zhang. Cordic algorithms and architectures: A survey. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 3(2):179–198, 2013.
- [16] Altera Corporation. *Cyclone IV Device Handbook, Volume 1*, March 2016. Available online: <https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/cyclone-iv/cyclone4-handbook.pdf> [Accessed: 2025-06-26].
- [17] K. I. Kum. *Fundamentals of Fixed-Point Arithmetic for FPGA Design*. Wiley, 2008.
- [18] A. Andreyev and V. Salauyou. Fpga implementation of the cordic algorithm for trigonometric functions. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 62(7):654–658, 2015.
- [19] MathWorks. *MATLAB Fixed-Point Designer and HDL Coder Documentation*, 2025.
- [20] T. Johnson and L. Smith. Fixed-point arithmetic in fpga designs: Challenges and solutions. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 68(4):1423–1435, 2021.
- [21] S. Park and J. Kim. Handling overflow in fixed-point arithmetic for fpga applications. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 28(8):1890–1902, 2020.
- [22] L. Chen and M. Zhang. Verification of fpga designs using advanced simulation methods. *IEEE Transactions on Industrial Informatics*, 15(5):2567–2575, 2019.
- [23] Intel Corporation. *Quartus Prime Pro Edition User Guide*, 2018.
- [24] Q. Zhao and H. Wu. Simulation and verification techniques for fpga designs using modelsim. *IEEE Design Test*, 33(4):45–53, 2016.
- [25] P. Smith and R. Jones. Low-power design techniques for embedded fpga applications. *IEEE Embedded Systems Letters*, 12(3):89–92, 2020.

[26] Terasic Inc. De2-115 board image. [https://www.terasic.com.tw/attachment/archive/502/image/image\\_74\\_thumb.jpg](https://www.terasic.com.tw/attachment/archive/502/image/image_74_thumb.jpg). Accessed: 2025-06-23.