



Dipartimento di ingegneria dell'informazione ed elettrica e matematica applicata

IMPLEMENTAZIONE DELLA NUMBER THEORETIC TRANSFORM (NTT) SU FPGA PER GLI ALGORITMI IN NIST/FIPS 203

Supervisore:
Prof. Ettore Napoli

Autore:
Andrea Scala

Indice

1	Introduzione	2
2	Operazioni in FIPS-203	3
2.1	dominio modulare	3
2.2	moltiplicazione tra polinomi	3
2.3	convoluzione discreta	3
2.4	convoluzione negaciclica	3
3	Trasformate Numero-Teoretiche (NTT)	5
3.1	Radice n -esima primitiva d'unità	5
3.2	Convoluzione basata su NTT di tipo negativo	5
3.2.1	Radice $2n$ -esima primitiva d'unità	5
3.2.2	NTT basata su ψ	6
3.2.3	Esempio di NTT basata su ψ	6
3.3	Calcolo dell'inverso moltiplicativo in \mathbb{Z}_q	7
3.3.1	Inverse NTT (INTT) basata su ψ	7
4	Calcolo Efficiente della NTT e della INTT	8
4.1	Algoritmo di Cooley–Tukey per la NTT	8
4.1.1	Descrizione dell'algoritmo	8
4.1.2	Operazione farfalla	8
4.2	Implementazione iterativa	9
4.3	INTT mediante Gentleman–Sande (GS)	9
4.3.1	Panoramica	9
4.3.2	Butterfly GS per l'INTT	9
4.3.3	Sequenza dei twiddle (zeta) e bit-reversal	9
4.3.4	Pseudocodice (INTT – Gentleman–Sande)	9
4.3.5	Correttezza (intuitiva)	10
4.3.6	Esempio simbolico ($n = 4$, relazione con Esempio 3.8)	10
4.3.7	Osservazioni implementative	10
4.4	Numero di operazioni	11
4.5	Conclusione	11
5	Implementazione su FPGA	12
5.1	Hardware per aritmetica modulare	12
5.1.1	Addizione modulare	12
5.1.2	Sottrazione modulare	13
5.1.3	Moltiplicazione modulare	13
5.1.4	pipelining	14
5.2	Butterfly Unit	15
5.3	Modulo NTT	16
5.4	Modulo INTT	17
5.5	Modulo FULL NTT-INTT	18
5.5.1	Area utilizzata	18

Capitolo 1

Introduzione

In questa relazione si descrive il processo di progettazione e implementazione di un modulo hardware su FPGA per l'esecuzione della Number Theoretic Transform o NTT all'interno degli algoritmi facenti parte dello standard FIPS 203, ossia lo schema di cifratura [CRYSTALS-KYBER](#)[1]. Quest'operazione ha un carico computazionale molto elevato, e un'implementazione in hardware anziché in software riduce di vari ordini di grandezza il tempo necessario per la sua esecuzione. La versione corrente del modulo è una versione completamente parallela progettata per input più piccoli di quelli usati in Kyber, poiché una versione completamente parallela avrebbe un'area fin troppo grande e non entrerebbe nel nostro FPGA.

Capitolo 2

Operazioni in FIPS-203

Prima di dare un'occhiata alla NTT e la nostra implementazione in hardware, è necessario comprendere alcuni concetti matematici preliminari e come questi vengono applicati all'interno degli algoritmi usati in Kyber.

2.1 dominio modulare

Tutte le operazioni in Kyber operano in un anello polinomiale modulo q , rappresentato da:

$$R_q = \mathbb{Z}_q[x]/x^n + 1$$

dove \mathbb{Z}_q è il campo intero finito modulo q , e $x^n + 1$ è il modulo polinomiale, vale a dire che gli elementi di R_q possono avere un grado massimo di $n - 1$. In Kyber, $n = 256$ e $q = 3329$.

2.2 moltiplicazione tra polinomi

Si definisce la moltiplicazione tra polinomi nel seguente modo: *siano $G(x)$ e $H(x)$ polinomi di grado $n - 1$ nell'anello \mathbb{Z}_q dove $q \in \mathbb{Z}$ e x è la variabile polinomiale, la moltiplicazione di $G(x)$ e $H(x)$ corrisponde a:*

$$Y(x) = G(x) \cdot H(x) = \sum_{k=0}^{2(n-1)} y_k x^k$$

dove $y_k = \sum_{i=0}^k g_i h_{k-i} \bmod q$, e g e h sono i coefficienti rispettivamente di $G(x)$ e $H(x)$. Come mostrato in seguito, Kyber fa un uso molto frequente di moltiplicazioni in R_q , rendendo l'ottimizzazione di questa operazione necessaria per un'esecuzione efficiente degli algoritmi.

2.3 convoluzione discreta

La moltiplicazione tra polinomi è equivalente a una *convoluzione lineare discreta* tra i vettori di coefficienti g e h :

$$y[k] = (g \star h)[k] = \sum_{i=0}^k g[i]h[k-i]$$

Quest'operazione è alla base della NTT; più precisamente, poiché Kyber opera su $R_q = \mathbb{Z}_q[x]/x^n + 1$, la NTT in Kyber si basa su una *convoluzione negaciclica*.

2.4 convoluzione negaciclica

siano $G(x)$ e $H(x)$ polinomi di grado $n - 1$ nell'anello $\mathbb{Z}_q/(x^n + 1)$ dove $q \in \mathbb{Z}$. una convoluzione negaciclica o negative wrapped convolution, $NWC(x)$ è definita come:

$$NWC(x) = \sum_{k=0}^{n-1} c_k x^k$$

dove $c_k = \sum_{i=0}^k g_i h_{k-i} - \sum_{i=k+1}^{n-1} g_i h_{k+n-i} \bmod q$. Se $Y(x)$ è il risultato della loro convoluzione lineare nell'anello $\mathbb{Z}_q[x]$, si può anche definire come:

$$NWC(x) = Y(x) \bmod (x^n + 1)$$

Poiché Kyber opera su un anello polinomiale modulare questa è l'operazione desiderata al contrario della semplice convoluzione lineare.

Capitolo 3

Trasformate Numero-Teoretiche (NTT)

Qui descriviamo le convoluzioni basate sulla trasformata numero-teoretica o number-theoretic transform (NTT). l'algoritmo classico per la NTT ha una complessità computazionale di $O(n^2)$, mentre algoritmi ottimizzati simili alla trasformata veloce di Fourier o FFT hanno una complessità lineare, ossia $O(n \log n)$.

3.1 Radice n -esima primitiva d'unità

Sia \mathbb{Z}_q un anello di interi modulo q , e $n - 1$ il grado polinomiale di $G(x)$ e $H(x)$. Tali anelli hanno come identità moltiplicativa (unità) l'elemento 1. Si definisce ω come radice n -esima primitiva d'unità in \mathbb{Z}_q se e solo se:

$$\omega^n \equiv 1 \pmod{q}$$

e

$$\omega^k \not\equiv 1 \pmod{q}, \quad \text{per } k < n.$$

In generale, la radice n -esima primitiva dell'unità in un anello \mathbb{Z}_q può non essere unica. Ad esempio, per $q = 7681$ e $n = 4$, le radici quartiche d'unità che soddisfano $\omega^4 \equiv 1 \pmod{7681}$ sono $\{3383, 4298, 7680\}$. Tuttavia, 7680 non è primitiva, poiché esiste $k = 2 < n$ con $\omega^2 \equiv 1 \pmod{7681}$. Pertanto, $\omega = 3383$ o $\omega = 4298$ sono radici quartiche primitive in \mathbb{Z}_{7681} .

Il valore di ω è essenziale nel calcolo della NTT e delle convoluzioni negacicliche basate su radici $2n$ -esime di unità. Per moduli grandi, la ricerca di ω è complicata. Una possibile alternativa è usare librerie matematiche come Sympy, che offre la funzione `nthroot_mod` per calcolare ω .

3.2 Convoluzione basata su NTT di tipo negativo

Questa sezione introduce la definizione di Trasformata Numero-Teoretica (NTT) e della sua inversa (INTT) basata su radici $2n$ -esime d'unità, ψ , e spiega come utilizzarle per calcolare convoluzioni negate o negacicliche.

3.2.1 Radice $2n$ -esima primitiva d'unità

Sia \mathbb{Z}_q un anello di interi modulo q , con polinomi di grado $n - 1$ e ω la radice n -esima primitiva d'unità. Si definisce ψ come radice $2n$ -esima primitiva d'unità se e solo se:

$$\psi^2 \equiv \omega \pmod{q}, \quad \psi^n \equiv -1 \pmod{q}.$$

Ad esempio, per $q = 7681$ e $n = 4$, con $\omega = 3383$, i possibili valori di ψ sono 1925 o 5756, poiché $\psi^2 \equiv 3383 \pmod{7681}$ e $\psi^4 \equiv -1 \pmod{7681}$.

3.2.2 NTT basata su ψ

La Trasformata Numero-Teoretica negativa o negaciclica (NTT_ψ) di un vettore di coefficienti polinomiali a è definita come:

$$\hat{a}_j = \sum_{i=0}^{n-1} \psi^{2ij+i} a_i \pmod{q}, \quad j = 0, \dots, n-1.$$

Grazie alla proprietà $\psi^2 = \omega$, si ottiene una trasformata che permette il calcolo della convoluzione negaciclica, adatta agli schemi crittografici post-quantum e alla cifratura omomorfica.

3.2.3 Esempio di NTT basata su ψ

[Esempio 3.8 – calcolo completo] Sia $q = 7681$, $n = 4$, $\psi = 1925$ nell'anello \mathbb{Z}_{7681} . Prendiamo il vettore di coefficienti

$$g = [1, 2, 3, 4].$$

La definizione della NTT basata su ψ fornisce la seguente moltiplicazione matriciale

$$\hat{g} = \begin{bmatrix} \psi^{2(0 \cdot 0)+0} & \psi^{2(0 \cdot 1)+1} & \psi^{2(0 \cdot 2)+2} & \psi^{2(0 \cdot 3)+3} \\ \psi^{2(1 \cdot 0)+0} & \psi^{2(1 \cdot 1)+1} & \psi^{2(1 \cdot 2)+2} & \psi^{2(1 \cdot 3)+3} \\ \psi^{2(2 \cdot 0)+0} & \psi^{2(2 \cdot 1)+1} & \psi^{2(2 \cdot 2)+2} & \psi^{2(2 \cdot 3)+3} \\ \psi^{2(3 \cdot 0)+0} & \psi^{2(3 \cdot 1)+1} & \psi^{2(3 \cdot 2)+2} & \psi^{2(3 \cdot 3)+3} \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}.$$

Semplificando gli esponenti (ricordando che l'elemento in posizione (j, i) è $\psi^{i(2j+1)}$) otteniamo la forma con potenze esplicite:

$$\hat{g} = \begin{bmatrix} \psi^0 & \psi^1 & \psi^2 & \psi^3 \\ \psi^0 & \psi^3 & \psi^6 & \psi^9 \\ \psi^0 & \psi^5 & \psi^{10} & \psi^{15} \\ \psi^0 & \psi^7 & \psi^{14} & \psi^{21} \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}.$$

Ora sostituiamo le potenze di ψ con i valori numerici ridotti modulo 7681. Poiché $\psi^4 \equiv -1 \pmod{7681}$, vale $\psi^8 \equiv 1$, per cui le potenze si ripetono con periodo 8. Le potenze rilevanti sono:

esponente k	$\psi^k \pmod{7681}$
0	1
1	1925
2	3383
3	6468
5	5756
6	4298
7	1213

Sostituendo nella matrice otteniamo la matrice numerica (tutte le entità ridotte modulo 7681):

$$\hat{g} = \begin{bmatrix} 1 & 1925 & 3383 & 6468 \\ 1 & 6468 & 4298 & 1925 \\ 1 & 5756 & 3383 & 1213 \\ 1 & 1213 & 4298 & 5756 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}.$$

Calcoliamo ora ciascuna componente di \hat{g} riga per riga (mostriamo i prodotti parziali, la somma grezza e la riduzione modulo 7681):

$$\begin{aligned} \hat{g}_0 &= 1 \cdot 1 + 1925 \cdot 2 + 3383 \cdot 3 + 6468 \cdot 4 \\ &= 1 + 3850 + 10149 + 25872 = 39872 \equiv 1467 \pmod{7681}, \\ \hat{g}_1 &= 1 \cdot 1 + 6468 \cdot 2 + 4298 \cdot 3 + 1925 \cdot 4 \\ &= 1 + 12936 + 12894 + 7700 = 33531 \equiv 2807 \pmod{7681}, \\ \hat{g}_2 &= 1 \cdot 1 + 5756 \cdot 2 + 3383 \cdot 3 + 1213 \cdot 4 \\ &= 1 + 11512 + 10149 + 4852 = 26514 \equiv 3471 \pmod{7681}, \\ \hat{g}_3 &= 1 \cdot 1 + 1213 \cdot 2 + 4298 \cdot 3 + 5756 \cdot 4 \\ &= 1 + 2426 + 12894 + 23024 = 38345 \equiv 7621 \pmod{7681}. \end{aligned}$$

Quindi il risultato finale della NTT è

$$\hat{g} = [1467, 2807, 3471, 7621].$$

Sia $g = [1, 2, 3, 4]$, $n = 4$ e $\psi = 1925$ in \mathbb{Z}_{7681} . Calcolando la trasformata secondo la definizione, si ottiene $\hat{g} = [1467, 2807, 3471, 7621]$.

3.3 Calcolo dell'inverso moltiplicativo in \mathbb{Z}_q

Sia q un numero primo e sia $a \in \mathbb{Z}_q$ con $a \neq 0$. L'inverso moltiplicativo di a in \mathbb{Z}_q è definito come il numero a^{-1} tale che

$$a \cdot a^{-1} \equiv 1 \pmod{q}.$$

Un metodo efficiente per calcolare a^{-1} si basa sul piccolo teorema di Fermat, che afferma:

$$a^{q-1} \equiv 1 \pmod{q}, \quad \text{per ogni } a \in \mathbb{Z}_q^*.$$

Da questa relazione si ricava immediatamente che:

$$a^{q-2} \equiv a^{-1} \pmod{q}.$$

Pertanto, per ottenere l'inverso di a in \mathbb{Z}_q è sufficiente calcolare l'esponenziazione modulare:

$$a^{-1} = a^{q-2} \pmod{q}.$$

Esempio

Supponiamo $q = 17$ e $a = 5$. Vogliamo trovare $5^{-1} \pmod{17}$.

Applichiamo la formula:

$$5^{-1} \equiv 5^{17-2} = 5^{15} \pmod{17}.$$

Calcoliamo passo per passo:

$$5^2 = 25 \equiv 8 \pmod{17}, \quad 5^4 \equiv 8^2 = 64 \equiv 13 \pmod{17},$$

$$5^8 \equiv 13^2 = 169 \equiv 16 \pmod{17}, \quad 5^{15} = 5^8 \cdot 5^4 \cdot 5^2 \cdot 5 \equiv 16 \cdot 13 \cdot 8 \cdot 5 \pmod{17}.$$

Eseguendo i calcoli modulari:

$$16 \cdot 13 = 208 \equiv 4 \pmod{17}, \quad 4 \cdot 8 = 32 \equiv 15 \pmod{17}, \quad 15 \cdot 5 = 75 \equiv 7 \pmod{17}.$$

Quindi

$$5^{-1} \equiv 7 \pmod{17}.$$

Infatti, $5 \cdot 7 = 35 \equiv 1 \pmod{17}$.

3.3.1 Inverse NTT (INTT) basata su ψ

L'inversa della trasformata NTT basata su ψ si calcola utilizzando la stessa matrice di trasformazione, ma con esponenti negativi e un fattore di normalizzazione n^{-1} (l'inverso moltiplicativo di n in \mathbb{Z}_q).

$$a_i = n^{-1} \sum_{j=0}^{n-1} \psi^{-2ij-i} \hat{a}_j \pmod{q}.$$

Ad esempio, a partire da $\hat{g} = [1467, 2807, 3471, 7621]$, si ricava $g = [1, 2, 3, 4]$ dopo l'applicazione dell'INTT.

Capitolo 4

Calcolo Efficiente della NTT e della INTT

In questo capitolo, discuteremo metodi efficienti per il calcolo della Trasformata di Fourier Numero-teorica (NTT) e della sua inversa (INTT). Il calcolo diretto tramite moltiplicazione matriciale ha complessità quadratica $O(n^2)$, mentre applicando un approccio divide-et-impera simile alla FFT di Cooley–Tukey, possiamo ridurre la complessità a $O(n \log n)$.

4.1 Algoritmo di Cooley–Tukey per la NTT

Sia $n = 2^m$ la dimensione della trasformata, e ψ una radice $2n$ -esima primitiva d'unità in \mathbb{Z}_q . Allora $\omega = \psi^2$ è una radice n -esima primitiva d'unità. L'algoritmo divide il problema di una NTT di dimensione n in due NTT di dimensione $n/2$, ricombinando i risultati tramite le cosiddette *operazioni farfalla* (*butterfly operations*).

4.1.1 Descrizione dell'algoritmo

Dato un vettore $g = (g_0, g_1, \dots, g_{n-1})$, la sua NTT è definita come

$$\hat{g}_i = \sum_{j=0}^{n-1} g_j \omega^{ij} \pmod{q}, \quad 0 \leq i < n.$$

Si può riscrivere separando indici pari e dispari:

$$\hat{g}_i = \sum_{j=0}^{n/2-1} g_{2j} \omega^{i(2j)} + \sum_{j=0}^{n/2-1} g_{2j+1} \omega^{i(2j+1)}.$$

Definendo due sottopolinomi

$$g^{(0)}(x) = \sum_{j=0}^{n/2-1} g_{2j} x^j, \quad g^{(1)}(x) = \sum_{j=0}^{n/2-1} g_{2j+1} x^j,$$

si ottiene

$$\hat{g}_i = \hat{g}_i^{(0)} + \omega^i \hat{g}_i^{(1)},$$

dove $\hat{g}^{(0)}$ e $\hat{g}^{(1)}$ sono NTT di dimensione $n/2$.

4.1.2 Operazione farfalla

Il passo fondamentale di ricombinazione è l'operazione farfalla:

$$(u, v) \mapsto (u + \omega^k v, u - \omega^k v) \pmod{q}.$$

Esempio 4.1. Sia $g = [1, 2, 3, 4]$, $n = 4$, $\psi = 1925$ in \mathbb{Z}_{7681} . Con $\omega = \psi^2 = 1925^2 \equiv 3383 \pmod{7681}$, si esegue la NTT in due stadi di farfalle. Dopo il primo stadio si ottengono i valori intermedi

$$[1 + 3, 2 + 4, 1 - 3, (2 - 4) \cdot 3383] = [4, 6, 6, 7680].$$

Dopo il secondo stadio:

$$[4 + 6, 4 - 6, 6 + 7680, 6 - 7680] = [10, 7679, 5, 7686] \equiv [10, 7679, 5, 5] \pmod{7681}.$$

4.2 Implementazione iterativa

L'algoritmo può essere implementato iterativamente in $m = \log_2 n$ stadi. In ciascuno stadio s vengono eseguite $n/2$ operazioni farfalla, usando come fattori ω^k con k appropriati.

Esempio 4.2. Per $n = 8$, $\psi = 1925$ in \mathbb{Z}_{7681} , il calcolo richiede 3 stadi di farfalle. I twiddle factors usati in ciascuno stadio sono $\{1\}$, $\{1, \omega^2\}$, $\{1, \omega, \omega^2, \omega^3\}$.

4.3 INTT mediante Gentleman–Sande (GS)

4.3.1 Panoramica

L'algoritmo di Gentleman–Sande (GS) è la controparte “decimation in frequency” (DIF) che si usa comunemente per implementare l'*inverse* NTT (INTT) in modo efficiente. Mentre la variante di Cooley–Tukey (DIT) è spesso usata per la NTT diretta, la versione GS è conveniente per la INTT poiché opera in-place partendo dall'output della NTT (spesso in ordine bit-reversed) e ricostruisce i coefficienti nel corso di stadi che riducono progressivamente la struttura a coppie (butterfly).

4.3.2 Butterfly GS per l'INTT

La singola operazione “farfalla” (butterfly) usata in GS, in modalità inversa, prende due valori u e v e aggiorna:

$$\begin{aligned} t &\leftarrow u, \\ u' &\leftarrow u + v \pmod{q}, \\ v' &\leftarrow \zeta \cdot (u - v) \pmod{q}, \end{aligned}$$

dove ζ è il fattore (twiddle) appropriato per quel particolare stadio e posizione. Notare la differenza con la farfalla Cooley–Tukey: qui la combinazione lineare avviene prima sulla somma, mentre la differenza viene scalata dal twiddle.

4.3.3 Sequenza dei twiddle (zeta) e bit-reversal

I fattori ζ per le varie farfalle sono precomputati e applicati in una certa sequenza; in molte implementazioni (ad es. nella specifica FIPS/kyber) tali fattori vengono letti in un ordine ottenuto mediante una funzione di *bit-reversal* degli indici per garantire che la disposizione degli elementi a ogni stadio corrisponda alla struttura a coppie richiesta. In sostanza:

- la NTT (CT / DIT) richiede twiddle in un certo ordine crescente;
- la INTT (GS / DIF) usa twiddle nello “specchio” di tale ordine (spesso ottenuto tramite bit-reversal degli indici).

Dopo l'ultimo stadio, per ottenere i coefficienti originali è necessario moltiplicare ogni valore per n^{-1} (l'inverso moltiplicativo di n modulo q).

4.3.4 Pseudocodice (INTT – Gentleman–Sande)

Qui riportiamo una versione iterativa in-place della INTT con algoritmo GS (notazione simile alla FIPS):

```
INPUT: f_hat[0..n-1]      // vettore in ingresso (di solito in bit-reversed order)
OUTPUT: f[0..n-1]         // vettore dei coefficienti originali

f <- f_hat
i <- NUM_ZETAS-1           // indice per leggere i twiddle in ordine inverso (es. bit-reversed)

for len <- 2 to n step len *= 2:
  for start <- 0 to n-1 step 2*len:
    zeta <- ZETAS[i] mod q
    i <- i - 1
    for j <- start to start + len - 1:
      t <- f[j]
      f[j] <- (t + f[j + len]) mod q
```

```

        f[j + len] <- zeta * (t - f[j + len]) mod q
    end for
end for

// Alla fine: moltiplica per n^{-1} (se non hai distribuito la divisione prima)
for k <- 0 to n-1:
    f[k] <- f[k] * n^{-1} mod q
end for

return f

```

4.3.5 Correttezza (intuitiva)

La sequenza di operazioni in ogni stadio ricombina coppie di coefficienti in modo tale che, a livello matriciale, si va applicando la matrice inversa della trasformazione NTT per blocchi. L'uso dei twiddle opportuni (con la giusta permutazione) garantisce che le potenze di ψ usate siano quelle inverse rispetto alla NTT diretta, e la moltiplicazione finale per n^{-1} normalizza il risultato.

4.3.6 Esempio simbolico ($n = 4$, relazione con Esempio 3.8)

Riprendiamo il vettore già ottenuto con la NTT basata su ψ (vedi Esempio 3.8):

$$\hat{g} = [\hat{g}_0, \hat{g}_1, \hat{g}_2, \hat{g}_3] = [1467, 2807, 3471, 7621].$$

Applichiamo l'algoritmo GS in due stadi ($\text{len} = 2$, poi $\text{len} = 4$). Denotiamo la tabella dei twiddle ζ usati (gli indici e l'ordine dipendono dalla strategia di memorizzazione; qui li indichiamo simbolicamente come ζ_0, ζ_1, \dots).

Stadio 1 ($\text{len} = 2$). Per $\text{start} = 0$ (un unico blocco di lunghezza $2 \cdot \text{len} = 4$):

$$\begin{aligned} j = 0 : \quad & t = \hat{g}_0, \quad \hat{g}_0 \leftarrow t + \hat{g}_2, \quad \hat{g}_2 \leftarrow \zeta_{s1} \cdot (t - \hat{g}_2); \\ j = 1 : \quad & t = \hat{g}_1, \quad \hat{g}_1 \leftarrow t + \hat{g}_3, \quad \hat{g}_3 \leftarrow \zeta_{s2} \cdot (t - \hat{g}_3). \end{aligned}$$

Dopo questo stadio otteniamo due coppie ricombinate (valori espressi come combinazioni lineari delle \hat{g}_i , moltiplicate per i relativi ζ).

Stadio 2 ($\text{len} = 4$). Ora per $\text{start} = 0$ (unico blocco, $\text{len} = 4$):

$$j = 0 : \quad t = f[0], \quad f[0] \leftarrow t + f[4?]$$

qui $f[4?]$ è fuori per $n=4$, quindi la formula si adatta: si usa la coppia rimanente,

(la forma generale è quella mostrata nel pseudocodice: quando $\text{len} = n$ si ricombina l'ultimo stadio e si ottengono i coefficienti ricostruiti).

Normalizzazione. Alla fine di tutti gli stadi moltiplichiamo ogni elemento per n^{-1} (in \mathbb{Z}_q) per ottenere i coefficienti finali:

$$g_i = n^{-1} \cdot f_i \pmod{q}, \quad i = 0, \dots, n-1.$$

In particolare, nel caso dell'Esempio 3.8 (con $n = 4$) il risultato dell'INTT applicato a $\hat{g} = [1467, 2807, 3471, 7621]$ restituisce il vettore originale $g = [1, 2, 3, 4]$ (dopo la normale riduzione modulare e la moltiplicazione per n^{-1}).

4.3.7 Osservazioni implementative

- **Ordine degli input / output:** GS (DIF) tipicamente assume l'input in ordine naturale e produce l'output in ordine bit-reversed (o l'opposto a seconda di convenzioni). È importante allineare l'ordinamento con la NTT/INTT che si usa nel resto del design.
- **Twiddle storage:** i ζ sono precomputati e memorizzati in tabella; l'indirizzamento può richiedere una permutazione bit-reversed (o l'accesso in reverse-order) per ragioni di efficienza e per corrispondenza con le fasi dell'algoritmo.

- **Scaling:** la moltiplicazione finale per n^{-1} può essere fatta come singola passata oppure si può distribuire il fattore di scala nei singoli stadi (es. shift di 1 bit per stadio quando n è potenza di due). Attenzione: la strategia di distribuzione deve preservare l'aritmetica modulare e la precisione (troncamenti/arrotondamenti).
- **Pipelining e latenza:** in implementazioni hardware pipeline, ogni farfalla può essere pipelined (moltiplicazione modulare, addizione, sottrazione). Quando si concatenano stadi, è necessario sincronizzare adeguatamente i segnali di controllo (valid, delay) e i registri di valore affinché i twiddle siano applicati alle coppie corrette.

4.4 Numero di operazioni

La complessità è

$$\frac{n}{2} \log_2 n$$

moltiplicazioni modulari e

$$n \log_2 n$$

addizioni modulari, molto meno delle n^2 richieste dal metodo diretto.

Esempio 4.4. Per $n = 256$, la NTT richiede 1024 moltiplicazioni modulari e 2048 addizioni modulari.

4.5 Conclusione

Il calcolo della NTT e della sua inversa può essere effettuato in tempo $O(n \log n)$ usando l'algoritmo fast-NTT con operazioni farfalla. Questo rende la NTT un componente cruciale per schemi di crittografia basati su reticoli, dove moltiplicazioni polinomiali ad alta dimensione devono essere eseguite in modo molto efficiente.

Capitolo 5

Implementazione su FPGA

In questo capitolo verrà esposta la progettazione e implementazione proposta della NTT specificamente per CRYSTALS-KYBER 3.0, incluso l'hardware per l'aritmetica modulare. Viene presentata una versione full-parallel per polinomi di cardinalità 8 ($n = 8$), mentre servirà un'implementazione di tipo seriale per $n = 256$ poiché una versione full-parallel riempirebbe troppa area per il nostro FPGA. si è usata una scheda Zybo Z7-20, con l'FPGA "XC7Z020-1CLG400C"

5.1 Hardware per aritmetica modulare

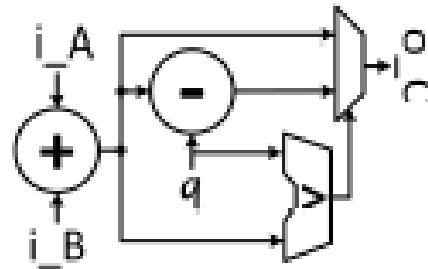
Cominciamo col descrivere i moduli hardware per le operazioni modulari, più precisamente si lavora nell'anello modulare \mathbb{Z}_q , con $q = 3329$ come da specifica per KYBER 3.0. In questa implementazione si useranno soltanto valori non-negativi (modulo classico), mentre alcune implementazioni usano anche valori negativi (modulo simmetrico), con valori possibili nell'intervallo $[-q/2 \leq n \leq (q/2) - 1]$. Per esempio, se il risultato di un'operazione nella nostra implementazione è uguale a 3328 ($q - 1$), allora usando il modulo simmetrico la stessa operazione ha come risultato -1 .

5.1.1 Addizione modulare

Il modulo per l'addizione modulare esegue un algoritmo molto semplice:

```
INPUT: A,B //ENTRAMBI A 12 BIT  
OUTPUT: SUM //12 BIT
```

```
1:   IF A+B >= 3329 THEN  
2:     SUM = (A+B) - 3329  
3:   ELSE  
4:     SUM = A+B
```



Il modulo usa soltanto un addizionatore, un sottrattore, un comparatore, e un multiplexer.

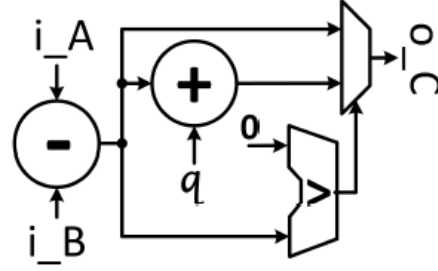
Figura 5.1: modulo per addizione modulare con $q = 3329$

5.1.2 Sottrazione modulare

Il modulo per la sottrazione modulare esegue lo stesso algoritmo dell'addizione, ma con ordine inverso:

INPUT: A,B //ENTRAMBI A 12 BIT
OUTPUT: SUB //12 BIT

```
1: IF A-B < 0 THEN
2:   SUB = (A-B) + 3329
3: ELSE
4:   SUB = A-B
```



Come prima, Il modulo usa soltanto un addizionatore, un sottrattore, un comparatore, e un multiplexer.

Figura 5.2: modulo per sottrazione modulare con $q = 3329$

5.1.3 Moltiplicazione modulare

Esistono vari algoritmi efficienti per la moltiplicazione modulare; il nostro modulo esegue una versione leggermente modificata dell'algoritmo di Barrett.

Algoritmo di Barrett per la moltiplicazione modulare

l'algoritmo di Barrett, chiamato anche *Barrett reduction*, per la moltiplicazione modulare ha il seguente aspetto generale: siano a e $n \in \mathbb{N}$, con $a < n^2$ si definisce la funzione $\text{mod } \lceil n : \mathbb{Z} \rightarrow (\mathbb{Z}/n\mathbb{Z})$ - dove \lceil è una funzione di approssimazione, nel nostro caso *round* - come:

$$a \text{ mod } \lceil n = a - \lfloor a/n \rfloor n.$$

Con n costante, è possibile evitare moltiplicazioni e divisioni, molto lente e costose in termini di area, rimpiazzandole con degli shift, completamente gratuiti in hardware e molto veloci.

implementazione hardware per KYBER

Si è deciso per l'implementazione proposta in [2], che esegue il seguente algoritmo basato su Barrett reduction:

Input: c , 24-bits unsigned integer; $q = 3329$ modulus

Output: x 12-bits unsigned, $x = c \text{ mod } q$

```
1:  $\hat{m} = \lceil \frac{c}{q} \rceil = \frac{c}{4096} (1 + \frac{1}{4} - \frac{1}{64} - \frac{1}{256}) = (c \gg 12) + (c \gg 14) - (c \gg 18) - (c \gg 20)$ 
2: correct = round(( $c[11 : 9] + c[13 : 11] - c[17 : 15] - c[19 : 17]$ ) >> 3)
3:  $m = \hat{m} + \text{correct}$ 
4:  $x = c - (q \times m)$ 
5: if  $x < 0$  then
6:    $x = x + q$ 
7: end if
8: return x
```

Descrizione dell'algoritmo

- 1: $\hat{m} = \lceil \frac{c}{q} \rceil = \frac{c}{4096} (1 + \frac{1}{4} - \frac{1}{64} - \frac{1}{256}) = (c \gg 12) + (c \gg 14) - (c \gg 18) - (c \gg 20)$
calcola il valore $\hat{m} = c/q$ usando un'approssimazione con potenze di 2 per $q = 3329$, infatti:

$$\frac{1}{4096} (1 + \frac{1}{4} - \frac{1}{64} - \frac{1}{256}) = (\frac{1}{2^{12}} + \frac{1}{2^{14}} - \frac{1}{2^{18}} - \frac{1}{2^{20}}) = 0.000300407 \approx 0.000300391 = \frac{1}{3329}$$

Poiché si usano soltanto potenze di 2, è possibile rimpiazzare la divisione con degli shift aritmetici.

- 2: correct = round(($c[11 : 9] + c[13 : 11] - c[17 : 15] - c[19 : 17]$) >> 3)
calcola un fattore di correzione necessario per via della perdita di informazione dovuta ai vari shift raggruppando i 3 bit meno significativi per ogni shift e dividendo il risultato per

8. Considerando i gruppi di bit come numeri con segno, *correct* può assumere un valore nell'intervallo $[-2, 2]$, arrotondando il risultato della divisione per eccesso.

- 3: $m = \hat{m} + \text{correct}$
Calcola il risultato della divisione con applicato il fattore di correzione.
- 4: $x = c - (q \times m)$
Calcola il risultato finale sottraendo al valore in ingresso originario $q * m$. Questo valore è garantito nell'intervallo $[-q, q]$
- 5: if $x < 0$ then
6: $x = x + q$
7: end if
8: return x

Infine si controlla se x è minore di 0, e nel caso viene aggiunto q come misura di compensazione per l'errore dovuto all'approssimazione nel fattore di correzione, restituendo x in uscita.

5.1.4 pipelining

Il diagramma a blocchi generale del modulo per la moltiplicazione modulare in kyber è il seguente:

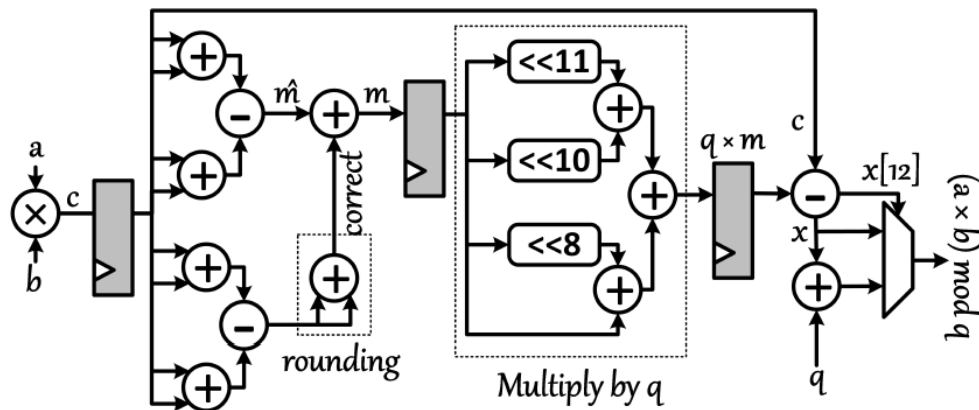


Figura 5.3: Schema circuitale per il modulo di moltiplicazione modulare

È possibile implementare una pipeline all'interno del modulo in 3 stadi, permettendo un throughput molto più alto utilizzando tutte le diverse parti del modulo che altrimenti rimarrebbero inutilizzate. Ci sono però alcuni segnali non rappresentati nello schema:

- il valore di c va anch'esso memorizzato tra i vari stadi di pipeline per sincronizzarsi con l'ultimo stadio, dove c è richiesto per l'operazione 4: $x = c - (qm)$.
- un segnale *valid_out* propagato per la pipeline che indica quando un valore un uscita è il risultato di un'operazione corretta, così da ignorare valori non validi tra un'attivazione del modulo e l'altra; È facilmente implementabile tramite un registro a scorrimento.

Il pipelining, mentre migliora di molto le prestazioni, introduce errori dovuti alla sincronizzazione tra ingressi e uscite, e per un corretto funzionamento del sistema è necessario gestire i segnali di controllo in modo attento così da ricevere valori corretti in uscita.

5.2 Butterfly Unit

Questo modulo implementa una singola unità che esegue le operazioni "butterfly", sia per la NTT (Cooley-Tukey) che per la INTT (gentleman-Sande), il cui schema ad alto livello è stato preso sempre da [2]. Il modulo esegue l'algoritmo esposto in precedenza nella sezione teorica:

```
IF (i_mode == 0) //NTT
    o_0 = (i_E * i_tf) + i_0 // u = u + v * twiddle
    o_E = (i_E * i_tf) - i_0 // v = u - v * twiddle
ELSE //INTT
    o_0 = (i_E + i_0) >> 1 // u = (u + v) / 2
    o_E = ((i_E - i_0) * twiddle) >> 1 // v = ((u - v) * twiddle) / 2
```

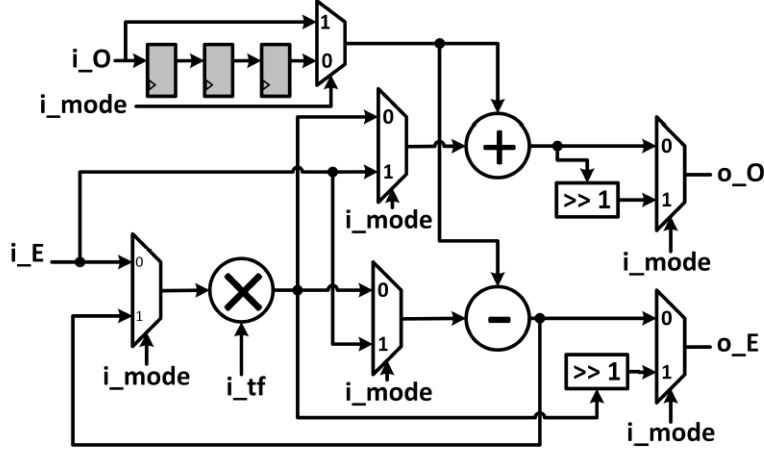


Figura 5.4: schema ad alto livello per la butterfly unit

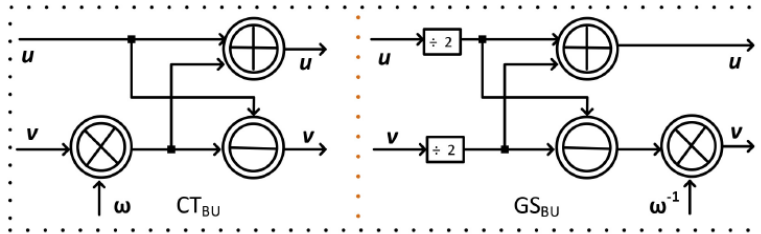


Figura 5.5: schemi logici per le butterfly CT e GS separate.

sono state fatte alcune modifiche:

- sono stati aggiunti 3 registri al secondo ingresso del multiplexer per o_0 per sincronizzarlo con gli altri valori negli stessi stadi della pipeline (poiché il moltiplicatore modulare è anch'esso pipelined con una latenza di 3 cicli di clock); il segnale "i_mode" è stato anch'esso pipelined aggiungendo 3 registri e usando l'ultimo come ingresso per i multiplexer;
- sono stati rimossi gli shift destri per la INTT poiché davano risultati errati, visto che dividere per $\frac{1}{n}$ non è possibile tramite una semplice divisione in un anello modulare; invece tutti i coefficienti passano per un moltiplicatore modulare che moltiplica i valori per l'inverso (nel caso di $n = 8$ e $q = 3329$, l'inverso risulta essere $8^{q-2} \bmod 3329 = 8^{3327} \bmod 3329 = 2913$, poiché $2913 * 8 \bmod 3329 = 23304 \bmod 3329 = 1$);
- è stato aggiunto un segnale "valid_in" in ingresso per indicare quando i valori in ingresso corrispondono a dei coefficienti su cui lavorare, e un segnale "valid_out" che indica quando l'uscita corrisponde ad un risultato di un'operazione NTT/INTT completata con la dovuta latenza, nel nostro caso 3 cicli di clock in totale.

5.3 Modulo NTT

Il modulo per la NTT è progettato per $n = 8$, ossia polinomi al massimo di grado 7, e quindi con 8 coefficienti. Non viene implementato nuovo hardware, ma consiste semplicemente di 12 butterfly unit collegate appropriatamente come mostrato nel grafico sottostante:

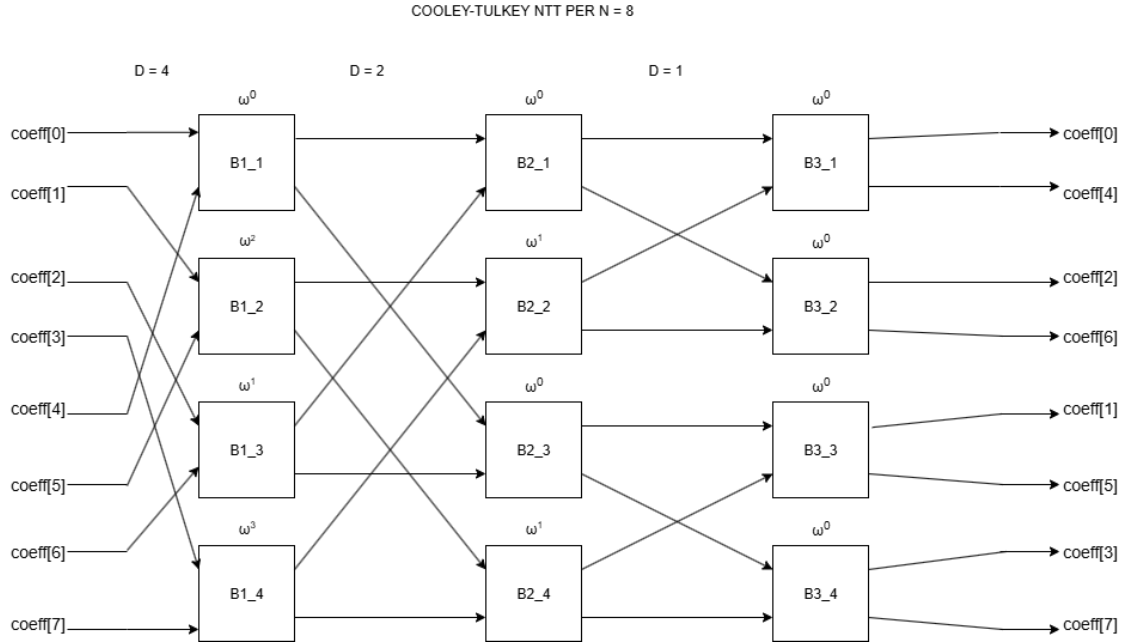


Figura 5.6: schema logico per il modulo NTT full parallel per $n = 8$. Da notare come l'output del modulo sia in ordine bit-reversed, così come l'ordine degli esponenti dei vari twiddle factor.

Il numero di Butterfly unit è direttamente proporzionale a n , crescendo in modo lineare (in $n \log n$). Poiché ogni butterfly ha una latenza di 3 cicli di clock, il modulo intero ha una latenza totale di 9 cicli, 3 per 3 livelli di esecuzione.

5.4 Modulo INTT

Il modulo per la INTT è quasi identico concettualmente al modulo per la NTT, ma l'ordine delle operazioni è invertito: si inizia da coefficienti in ordine bit-reversed e distanza 1 (considerando la posizione dei coefficienti piuttosto che il loro indice) e si termina con coefficienti in ordine normale, con una distanza di 4 per l'ultimo stadio di butterfly; sono presenti inoltre dei moltiplicatori modulari che moltiplicano ogni coefficiente per l'inverso modulare di N , che per $N = 8$ corrisponde a 2913;

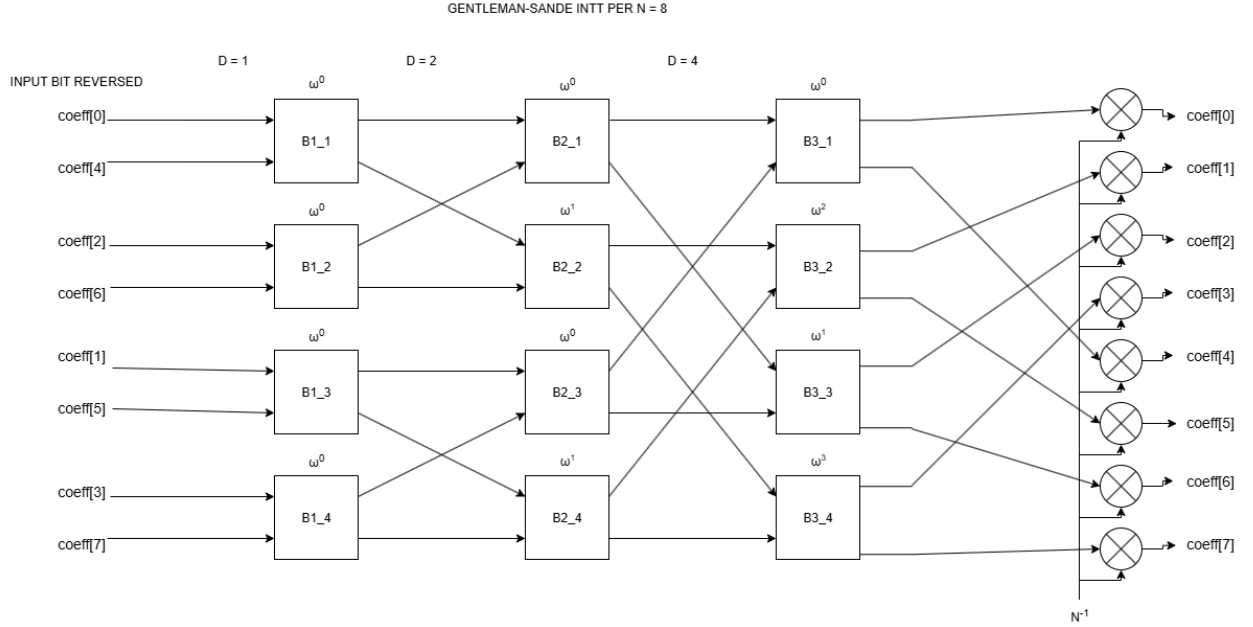


Figura 5.7: schema logico per il modulo INTT full parallel per $n = 8$. Qui l'uscita è riordinata normalmente, così da corrispondere completamente al vettore di coefficienti originario.

Il modulo per la INTT per $n = 8$ ha una latenza totale di 12 cicli di clock, 9 per i vari livelli di butterfly più 3 cicli di clock per i moltiplicatori modulari.

5.5 Modulo FULL NTT-INTT

Questo modulo testa il funzionamento dei blocchi per la NTT e la INTT collegandoli in cascata, così da ottenere alla fine dell'INTT il vettore di coefficienti inserito in input, come prima nessun hardware aggiuntivo è stato implementato.

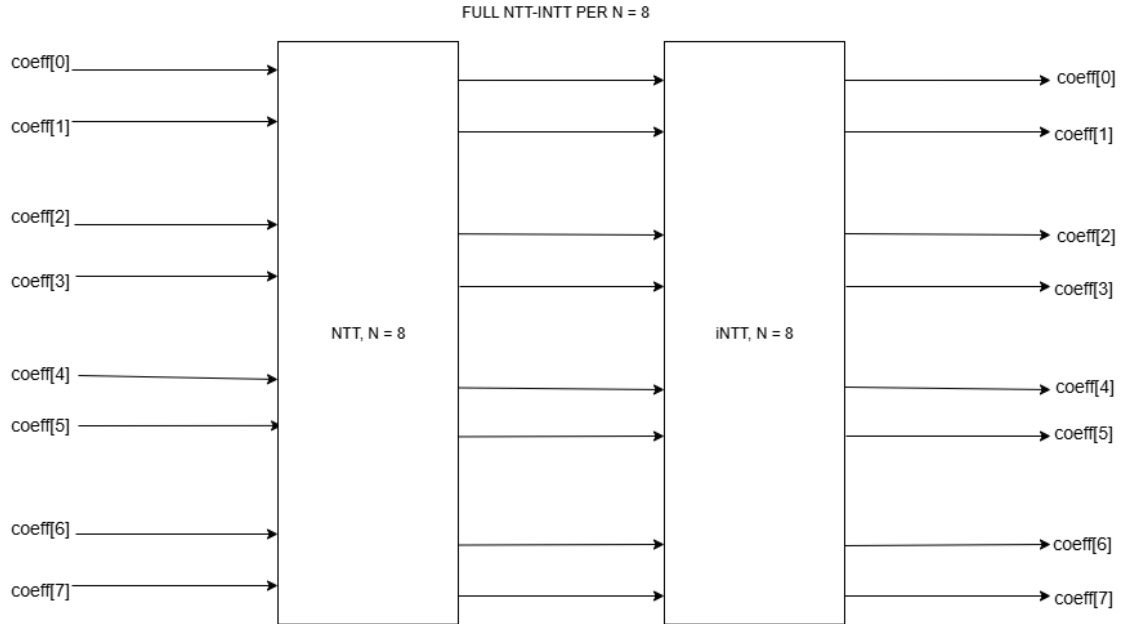


Figura 5.8: schema logico per il modulo FULL NTT-INTT per $n = 8$. Questo modulo ha la sola funzione di testare il corretto funzionamento dei moduli che eseguono la NTT e la INTT.

Il modulo intero ha una latenza totale di 21 cicli di clock tra input e output, 9 per la NTT e 12 per la INTT

5.5.1 Area utilizzata

Il modulo è stato sintetizzato per l'FPGA XC7Z020-1CLG400C; Il design completo FULL NTT-INTT utilizza un totale di:

- 3390 slice LUT;
- 3066 slice Registers;
- 18 blocchi DSP;

Ogni butterfly usa tra 200 e 400 celle logiche (il numero esatto varia a seconda delle possibili ottimizzazioni, per esempio in molte butterfly il twiddle factor corrisponde a 1, consentendo al sintetizzatore di eliminare completamente il moltiplicatore risparmiando quindi blocchi DSP), il che rende subito evidente che un modulo completamente parallelo per $n = 256$ non è fattibile, poiché avrebbe bisogno di $n/2$ butterfly per stadio per $\log_2 n$ stadi, per un totale di $\frac{n}{2} \log_2 n = 128 * 8 = 1024$ butterfly, richiedendo quindi circa 300.000 celle logiche, che sono molto di più di quelle presenti nel nostro FPGA (53200 LUT e 106.400 registri). È quindi necessario sviluppare una versione più compatta ma con una latenza inevitabilmente maggiore.

Bibliografia

- [1] Federal Information Processing Standards (FIPS). Module-lattice-based key-encapsulation mechanism standard. 2024.
- [2] Elewa Sherif and Tawfik Eslam. An efficient number theoretic transform implementation for fips-203 on fpga. *IEEE International Symposium on Circuits and Systems (ISCAS)*, 2025.