Università della Calabria

Progetto Elettronica Digitale

Moltiplicatore "carta e penna" con l'utilizzo di un carry save

Francesca Vaccaro 239641
Luca Timpano 240236
Greta Tigano 239619

Instructor: Stefania Perri
Corso di Studio in Ingegneria Informatica

2024/2025

Contents

1 Introduzione

Il progetto si concentra sulla realizzazione di un moltiplicatore binario basato sul metodo carta e penna, sviluppato utilizzando il linguaggio di descrizione hardware VHDL.

L'obiettivo principale è stato quello di implementare un circuito in grado di eseguire moltiplicazioni tra due operandi binari a 16 bit unsigned. La progettazione ha sfruttato la tecnica del carry save per ottenere un'implementazione efficiente e ottimizzata, in particolare per il calcolo e la somma dei prodotti parziali.

Questa relazione illustra tutti i passi della progettazione, partendo dalla descrizione delle tecniche utilizzate e del codice VHDL, fino all'analisi delle simulazioni. In particolare, l'ultima parte della relazione approfondisce un aspetto cruciale del progetto, ovvero l'analisi delle prestazioni del circuito. Tra queste, vengono esaminati l'occupazione delle risorse hardware, con particolare attenzione al numero di LUTs (Look-Up Tables) e Flip-Flops utilizzati, e la massima frequenza operativa, al fine di valutare l'efficienza temporale del circuito.

1.1 Cenni teorici

1.1.1 Moltiplicatore "carta e penna"

La logica utilizzata nel moltiplicatore carta e penna è basata su quella binaria. Procediamo prendendo ogni bit del moltiplicando (ad esempio A) e viene svolto il prodotto con il moltiplicatore (ad esempio B), ottenendo una serie di prodotti parziali. Possiamo infatti descrivere la moltiplicazione mediante tale formula:

$$P(A,B) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} (b_i \cdot a_j) \cdot 2^{i+j}$$

Una volta generati questi prodotti parziali, ognuno di essi viene shiftato verso sinistra considerando il bit del moltiplicatore, se è in posizione 0, manteniamo il prodotto parziale non shiftato altrimenti effettuiamo lo shift sinistro.
La parte finale riguarda la somma tra tutti i prodotti parziale mediante al quale otteniamo il risultato finale.

1.1.2 Carry Save

Un sommatore carry-save è un tipo di sommatore digitale progettato per eseguire efficientemente la somma di tre o più numeri binari. A differenza degli altri sommatori, un sommatore carry-save restituisce due (o più) risultati separati, e la somma finale può essere ottenuta combinando questi output. Questa tecnica è frequentemente utilizzata nei moltiplicatori binari, in quanto la moltiplicazione comporta l'aggiunta di più numeri binari dopo il calcolo dei prodotti parziali.
A differenza degli altri sommatori in cui la somma e i carry di riporto vengono calcolati e propagati bit per bit, questa tipologia di sommatore, tratta la somma

parziale e il carry come due differenti vettori, infatti li calcola separatamente in ogni bit. Procede poi con la somma di tutte le somme parziale e solo dopo sommiamo i carry ottenendo il risulato finale.
Dunque, considerati tre qualsiasi numeri A, B, C, la somma di ogni bit produce una somma parziale, ottenuta grazie all'utilizzo dell'operatore **XOR**. Mentre, i carry, cioè i riporti, vengono calcolati separatamente rispetto alla somma, e usiamo l'operatore **AND**.

2 Progettazione

2.1 Schema di progettazione

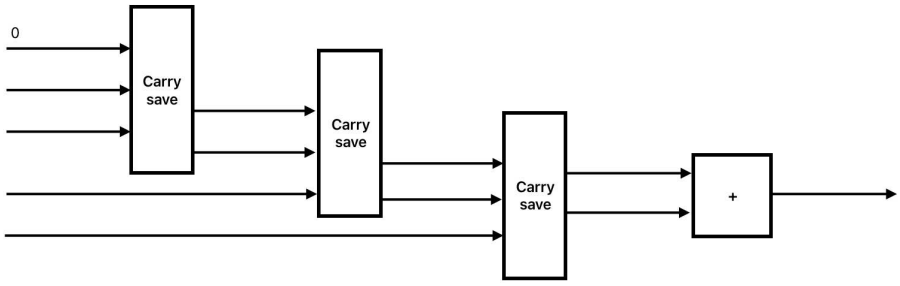

Figure 1: Esempio di architettura del moltiplicatore a 4 bit

2.2 Constraint Clock

Il constraint di clock in VHDL è uno strumento fondamentale per garantire il corretto funzionamento del design digitale. Esso definisce i requisiti temporali del clock, come il periodo e la frequenza operativa, e permette ai tool di sintesi e implementazione di ottimizzare il circuito per rispettare i vincoli temporali specificati. Grazie al constraint di clock, è possibile identificare e risolvere problemi di timing, come violazioni dei tempi di setup e hold, assicurando che il circuito possa funzionare correttamente alla frequenza desiderata. Inoltre, l'uso di vincoli temporali consente di migliorare le prestazioni globali del design, riducendo i tempi di propagazione e garantendo una maggiore affidabilità del sistema.

2.3 Implementazione VHDL

```
    library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Registro is
generic(m : integer := 16);
    Port (
```

```
    input : in std_logic_vector(m - 1 downto 0);
    clk : in std_logic;
    output : out std_logic_vector(m - 1 downto 0)
);
end Registro;
architecture BehaviouralRegistro of Registro is
begin
    process(clk) begin
        if rising_edge(clk) then
            output <= input;
        end if;
    end process;
end BehaviouralRegistro;

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity AdderUnit is
    generic(y : integer := 32);
    port(
    A, B : in std_logic_vector(y - 1 downto 0);
    output : out std_logic_vector(y - 1 downto 0)
    );
end AdderUnit;
architecture BehaviouralAdderUnit of AdderUnit is

signal carry : std_logic_vector(y downto 0);

begin
carry(0) <= '0';
myFor: for i in 0 to y - 1 generate
    output(i) <= A(i) xor B(i) xor carry(i);
    carry(i + 1) <= (A(i) and B(i)) or (A(i) and carry(i)) or
    (B(i) and carry(i));
end generate myFor;
end BehaviouralAdderUnit;

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity carrySave is
    generic(z : integer := 32);
    port(
    A,B,C : in std_logic_vector(z - 1 downto 0);
    carry, output : out std_logic_vector(z - 1 downto 0)
    );
```

```
end carrySave;

architecture BehaviouralCarrySave of carrySave is

signal carryIntermedio : std_logic_vector(z - 1 downto 0);

begin
    output <= ((A xor B) xor C);
    carryIntermedio <= (((A and B) or (A and C)) or (B and C));
    carry <= carryIntermedio(z - 2 downto 0) & '0';
end BehaviouralCarrySave;


library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.numeric_std.all;


-- Entità del moltiplicatore
entity Moltiplicatore is
generic(n : integer := 16);
Port (
    A,B : in std_logic_vector(n - 1 downto 0);
    output : out std_logic_vector((n * 2) - 1 downto 0);
    clk : in std_logic
);
end Moltiplicatore;

-- Architettura del moltiplicatore
architecture Behavioral of Moltiplicatore is

signal regA, regB : std_logic_vector(n - 1 downto 0);
signal regOut : std_logic_vector((n * 2) - 1 downto 0);
signal shiftVector : std_logic_vector(n downto 0);

--dichiaro la matrice
type partial_product_matrix is array (15 downto 0)
of std_logic_vector(31 downto 0);
type matrix_somme is array (31 downto 0)
of std_logic_vector(31 downto 0);


signal partial_products: partial_product_matrix;
signal shifted_partial_products: partial_product_matrix;
signal somme_parziali_carry : matrix_somme;
```

```
component Registro is
    generic(m : integer := 16);
    Port (
    input : in std_logic_vector(m - 1 downto 0);
    clk : in std_logic;
    output : out std_logic_vector(m - 1 downto 0)
);
end component;

component AdderUnit is
    generic(y : integer := 32);
    port(
    A, B : in std_logic_vector(y - 1 downto 0);
    output : out std_logic_vector(y - 1 downto 0)
    );
end component;

component carrySave is
    generic(z : integer := 32);
    port(
    A,B,C : in std_logic_vector(z - 1 downto 0);
    carry, output : out std_logic_vector(z - 1 downto 0)
    );
end component;

begin

    registroA: Registro generic map(16) port map(A, clk, regA);
    registroB: Registro generic map(16) port map (B, clk, regB);
    registroOut: Registro generic map(32) port map(regOut, clk, output);

    shiftVector <= (others => '0');

    myForProd: for i in 0 to n - 1 generate
        partial_products(i) <= (15 downto 0 => '0' ) &
        (regA and (15 downto 0 => regB(i)));
    end generate;

    -- Genera gli shift per ogni riga della matrice
    genShift: for i in 0 to 15 generate
        shifted_partial_products(i) <= partial_products(i)(31-i downto 0)
        & (i downto 1 => '0');
    end generate;

    --logica principale
    carrySave1_0: carrySave generic map (32)
```

```
    port map(shifted_partial_products(0), shifted_partial_products(1),
    (others => '0'), somme_parziali_carry(0), somme_parziali_carry(1));
    belFor: for i in 1 to n - 2 generate
        carrySaveMap: carrySave generic map(32)
        port map(somme_parziali_carry(2 * i - 2),
        somme_parziali_carry(2 * i - 1),
        shifted_partial_products(i + 1), somme_parziali_carry(2 * i),
        somme_parziali_carry(2 * i + 1));
    end generate;

    final_adder : AdderUnit generic map(32)
    port map(somme_parziali_carry(28),
    somme_parziali_carry(29), regOut);

end Behavioral;
```

Il codice riportato sopra implementa un moltiplicatore a 16 bit, la cui struttura è costituita da diversi componenti che andremo ad analizzare nel dettaglio.
Il primo componente riportato è il **Registro**. Il suo compito, descritto nell'architettura, in particolare nel *process*, è quello di memorizzare i dati in uscita su un fornte di salita del clock, attraverso l'utilizzo della funzione *rising edge(clk)*.

Il secondo componente è l'**Adder Unit**. Si tratta di un'unità di somma che somma due vettori di bit, A e B, generando un risultato di larghezza pari alla dimensione di A (o B), che di default è 32 bit. Nell'architecture abbiamo dichiarato un vettore, *carry* usato per gestire il riporto durante la somma e attraverso un ciclo andiamo a eseguire la somma bit per bit dei vettori in input. Tutto ciò utilizzanto gli operatori logici *AND* e *XOR*.

In seguito troviamo il modulo **Carry Save**, che implementa la somma tra tre vettori di bit. I vettori binari in ingresso sono A, B e C, ognuno di 32 bit, mentre le porte in unscita sono il carry, che presenta il risultato dei carry parziali generati dalla somma, ma con un bit aggiuntivo a zero per allineare correttamente la larghezza del vettore di output e infine l'output che è la somma senza riporto.
Nell'architecture calcoliamo la somma dei tre vettori senza considerare il riporto e successivamente utilizziamo un segnale, precedentemente dichiarato, che è il carry intermedio, in cui la combinazione tra and e or fa in modo di identificare i bit che causano un riporto, cioè quando almeno due dei tre bit sono pari a 1. Dopo aver calcolato questo carry intermedio, possiamo riportarlo nel carry finale con l'aggiunta di un bit (0) come bit più significativo.

Infine abbiamo la descrizione del componente più importante, cioè il **Moltiplicatore**. Nell'entità dichiariamo A e B, come vettori di input a 16 bit, cioè i numeri che devono essere moltiplicati. Poi riportiamo anche un segnale di clock per sincronizzare le operazione e l'output, che rappresenta il risultato finale del prodotto che

dunque avrà una lunghezza pari a due volte la dimensione dei vettori in input (2n). Nell'architettura descriviamo la sua logica e il funzionamento effettivo che sta alla base del moltiplicatore, partiamo dalla dichiarazione dei segnali.

- **regA e regB**: Sono i registri che salvano i numeri di ingresso A e B.
- **regOut**: Registro per memorizzare il risultato del prodotto finale.
- **shiftVector**:Segnale usato per l'operazione di shift nel calcolo dei prodotti parziali.
- **partial products**: Rappresenta una matrice di prodotti parziali, infatti, contiene i prodotti tra i singoli bit di A e B, e avrà una larghezza di 32 bit per ciascun elemento.
- **shifted partial products**: Questa matrice possiede i prodotti parziali shiftati in base alla loro posizione.
- **somme parziali carry**: Matrice di carry e somme parziali che viene usata nei passaggi successivi per sommare i prodotti parziali.

In seguito dichiariamo i components utlizzati per implementare l'operazione di moltimplicazione. Dunque riportiamo il **Registro**, l'**Adder Unit** e il **Carry Save** precedentemente discussi.
Procediamo con la descrizione delle operazioni. I numeri A e B, attraverso il segnale di clock, vengono caricati nei registri regA e regB, garantendo di sincronizzare l'inizio del calcolo.

Segue il calcolo della matrice dei prodotti parziali, dove ogni elemento di è il prodotto parziale tra il bit i di regB e l'intero valore di regA, eseguito utilizzando l'operatore AND. Inoltre, i risultati vengono estesi a 32 bit, con la parte superiore (i bit più significativi) inizializzata a zero.

Successivamente, ogni prodotto parziale nella matrice partial products viene shiftato verso sinistra in base alla posizione del bit di B che ha generato il prodotto parziale.

Poi procediamo con la somma dei prodotti parziali spostati utilizzando il carry save.Nel primo livello, i primi due prodotti spostati vengono sommati con uno zero iniziale (per il carry) usando un componente carrySave, nei successivi livelli, i risultati delle somme parziali vengono continuamente sommati a gruppi di tre utilizzando il sommatore carry-save.

Solo dopo aver eseguito questi passaggi possiamo sommare i due risultati con l'Adder Unit. Il risultato sarà quindi il prodotto tra A e B, che viene memorizzato nel resistro regOut e poi resituito in output.

2.3.1 Schematic

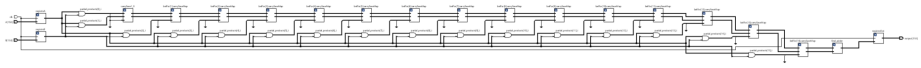

Figure 2: Schematic

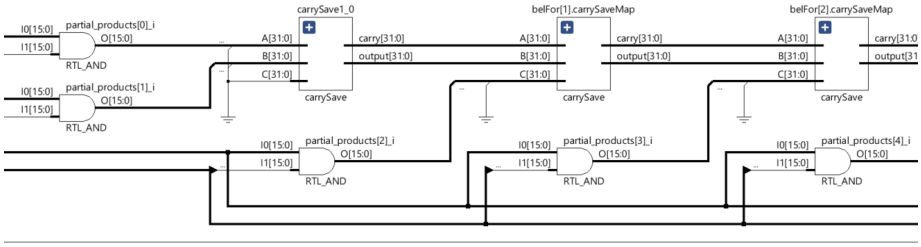

Figure 3: Zoom Schematic

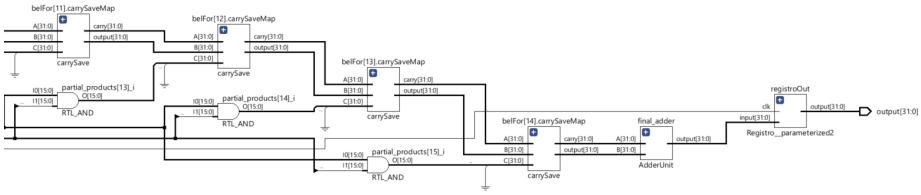

Figure 4: Zoom Schematic

3 Test Bench

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
```

```
use IEEE.NUMERIC_STD.ALL;
use work.all;

entity moltiplicatore_tb is
end moltiplicatore_tb;
architecture Behavioral of moltiplicatore_tb is
        component Moltiplicatore is
        port(
        A, B: in std_logic_vector(15 downto 0);
        clk: in std_logic;
        output: out std_logic_vector(31 downto 0)
        );
        end component;
        constant n : integer := 16;
        signal term1, term2: std_logic_vector(15 downto 0);
        signal result: std_logic_vector(31 downto 0);
        signal clk: std_logic;
begin
        A: Moltiplicatore port map(
        A => term1,
        B => term2,
        output => result,
        clk => clk
        );
process
begin
        myFor1: for i in 0 to 2**n-1 loop
                myFor2: for j in 0 to 2**n-1 loop
                        wait until (rising_edge(clk));
                        term1 <= std_logic_vector(to_signed(i, n));
                        term2 <= std_logic_vector(to_signed(j, n));
                end loop;
        end loop;
        wait;
end process;
end Behavioral;
```

Con il test bech andiamo a simulare il moltiplicatore e a verificarne il funzionamento. Dopo aver riportato l'entity del test bench che come ben sappiamo è vuota, procediamo con la definizione dell'architettura. Ovviamente riportiamo il componente che vogliamo testare, dunque il moltiplicatore, successivamente dichiariamo le variabili:

- **constant n**: Tale costante, con valore 16, definisce la dimensione dei vettori term1 e term2.
- **signal term1, term2**: Vettori usati come ingressi del moltiplicatore.

- **signal result**: Vettore che contiene il risultato finale.

- **signal clk**: Segnale che serve per garantire la sincronizzazione.

Procediamo, come prima cosa a collegare i segnali appena dichiarati alle porte del componente.

Successivamente implementiamo il processo di simulazione.
Con un primo ciclo esterno iteriamo su tutti i valori possibili di i(da 0 a $2^n - 1$) e simuliamo il primo numero. Poi inseriamo un secondo ciclo, che itera su tutti i possibili valori di j (da 0 a $2^n - 1$) e simuliamo il secondo numero. Dopo di che il processo si sospende fino al fronte di salita del segnale di clock. Questo simula un comportamento sincrono, dove le operazioni vengono effettuate a ogni ciclo di clock. Dopo l'istruzione di attesa, viene assegnato il valore di i a term1 e il segnale di j a term2. Con un'ultimo *wait* la simulazione attende indefinitamente dopo l'esecuzione dei cicli, fermando il processo.

4 Simulazione

Il progetto si propone di sviluppare una componente aritmetica capace di eseguire operazioni di moltiplicazione. La presente sezione è dedicata alle ultime fasi del processo di progettazione e verifica di un circuito. Verranno trattati i seguenti aspetti:

- **Simulazione Comportamentale**: durante questa fase si effettua una verifica funzionale per analizzare il comportamento logico del circuito.

- **Analisi Temporale Post-Sintesi**: include lo studio dei ritardi e delle prestazioni temporali del circuito sintetizzato.

- **Sintesi e Simulazione sul Dispositivo Implementato**: in questa fase il codice viene elaborato per produrre una netlist che comprende elementi come **Look-Up Table (LUT)**, Flip-Flop e altri componenti logici.

- **Frequenza Massima di Funzionamento e Analisi del Consumo Energetico**: si analizzano le prestazioni in termini di frequenza massima raggiungibile dal circuito e si valutano i consumi energetici per stimare l'efficienza complessiva del design.

4.1 Behavioral Simulation

Il primo passo affrontato è stato la simulazione comportamentale (*Behavioral Simulation*), mirata a verificare il corretto funzionamento del circuito prima della fase di sintesi e della mappatura sull'hardware.
Nella sezione precedente è stato descritto il *testbench* dedicato, in cui è stato specificato come i segnali di ingresso A e B influenzano l'uscita.
Durante questa simulazione, si è osservato che il risultato corretto viene calcolato dopo due colpi di clock. Ad esempio, considerando il prodotto tra 1 e 34838, il circuito fornisce correttamente il risultato 34838 rispettando la latenza prevista di

due cicli di clock. Questo comportamento conferma il funzionamento atteso del codice VHDL sviluppato.

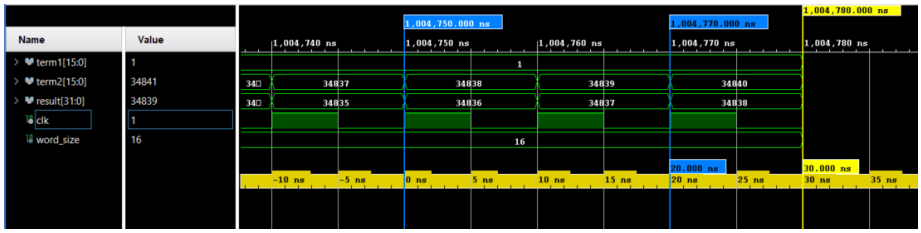

Figure 5: Behavioural Simulation

4.2 Synthesis e Implementation Device Simulation

L'implementazione rappresenta la fase in cui il circuito progettato viene sintetizzato e mappato sul dispositivo FPGA utilizzato, che nel nostro caso è un `XC7Z020CLG400-3`. In questa fase, il codice VHDL è stato trasformato in una netlist composta da elementi logici come *Look-Up Tables (LUT)*, Flip-Flop, e altri blocchi, ottimizzati per l'hardware target. La simulazione di implementazione ha mostrato che il design è stato correttamente allocato nelle risorse dell'FPGA, rispettando i vincoli temporali e di area definiti in fase di progettazione.
È importante sottolineare che, a seconda del *clock constraint* utilizzato, possono variare l'implementazione del circuito e l'uso degli elementi logici. Un vincolo temporale più stringente può richiedere una maggiore ottimizzazione, portando a un utilizzo più intensivo delle risorse disponibili, come LUT e Flip-Flop, per garantire il rispetto delle tempistiche. Al contrario, vincoli temporali meno restrittivi possono comportare una progettazione più rilassata, con un minore consumo di risorse logiche. Questo aspetto evidenzia l'importanza di una corretta definizione dei vincoli di progetto in funzione delle esigenze applicative.

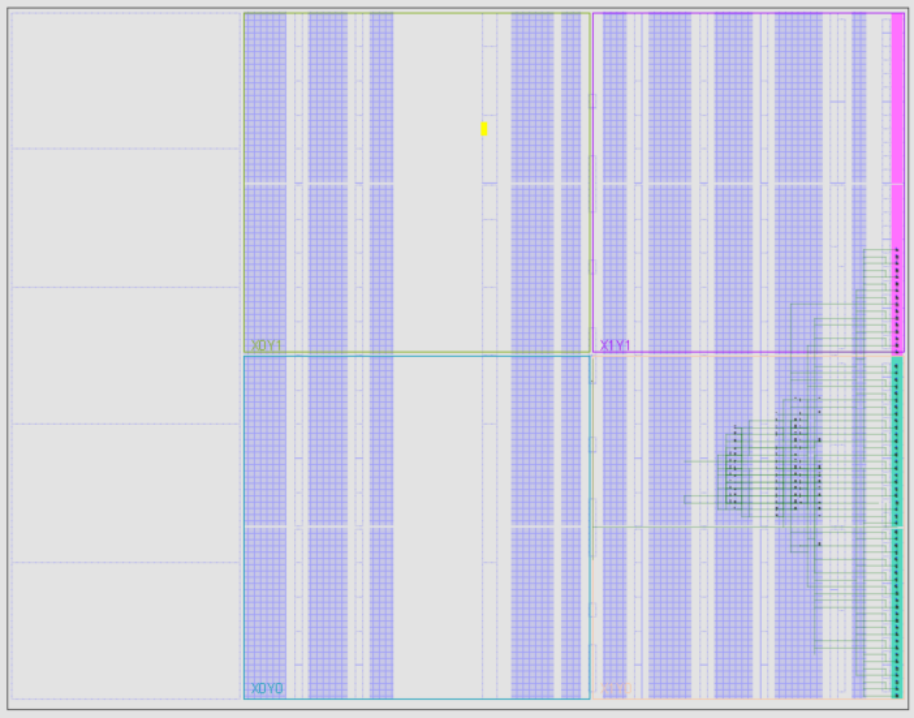

Figure 6:

Aprendo i report di sintesi, possiamo analizzare le caratteristiche delle componenti utilizzate per la realizzazione del moltiplicatore, confrontando i risultati per i due vincoli di clock, 12 ns e 25 ns.

Table 1: Risorse utilizzate per i clock a 12 ns e 25 ns

Ref Name	**Used (12 ns)**	**Used (25 ns)**	**Available**	**Util% (12 ns / 25 ns)**
LUT6	234	234	17600	1.33% / 1.33%
LUT5	49	49	17600	0.28% / 0.28%
LUT4	150	150	17600	0.85% / 0.85%
LUT2	92	92	17600	0.52% / 0.52%

Come si può osservare, l'utilizzo delle risorse logiche rimane invariato tra i due vincoli di clock, indicando che la variazione nella frequenza non influisce sulla quantità di risorse occupate. Le LUT (Look-Up Tables) rappresentano la componente principale utilizzata, con un utilizzo massimo dell'1.33%, mostrando che il design sfrutta in modo efficiente le risorse disponibili sul dispositivo FPGA.

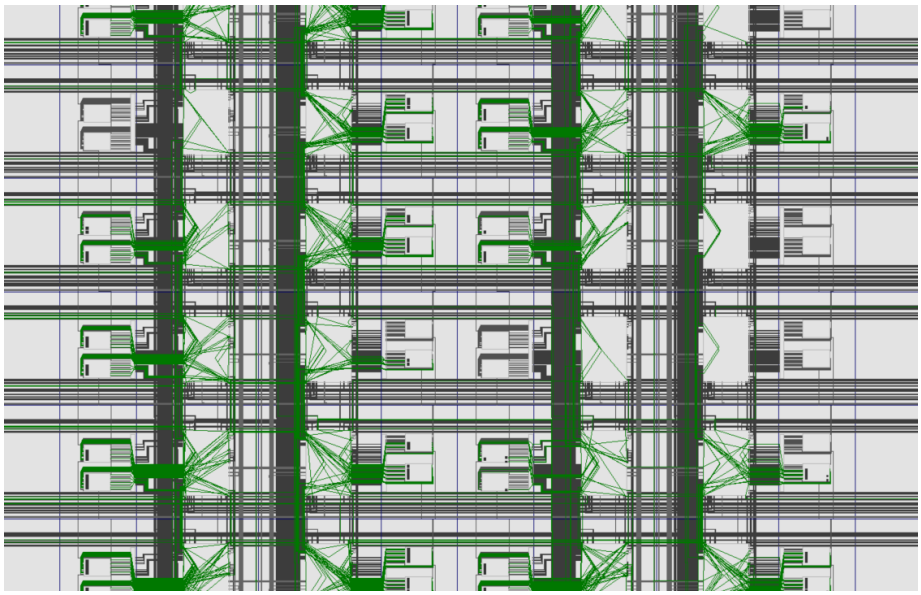

Figure 7: Zoom sul circuito

4.3 Post Synthesis Timing Analysis

La simulazione post-temporale rappresenta una fase essenziale per verificare il corretto funzionamento del moltiplicatore implementato sull'FPGA. Questa simulazione tiene conto dei ritardi fisici associati ai componenti del circuito e al design del moltiplicatore carta e penna basato sull'algoritmo *carry-save*. A differenza della simulazione comportamentale (*Behavioral Simulation*), che assume un comportamento ideale senza ritardi, nella simulazione post-temporale vengono considerati i tempi di propagazione reali introdotti dalle risorse logiche e dalle interconnessioni. Come mostrato nella figura 8, il risultato del prodotto non è immediatamente disponibile in uscita, a causa dei **tempi di propagazione** dovuti alle *Look-Up Tables (LUT)* e al metodo *carry-save*, che genera una propagazione parziale dei riporti tra i vari stadi di calcolo. Ogni LUT introduce un ritardo legato al tempo necessario per calcolare i parziali e combinare i riporti. Inoltre, i segnali di ingresso devono attraversare la rete di interconnessione dell'FPGA per raggiungere le LUT assegnate a ciascuna fase del moltiplicatore e propagarsi fino all'uscita.
Un fenomeno analogo si osserva per gli ingressi A e B, che possono subire sfasamenti temporali a causa di differenze nei tempi di propagazione lungo i percorsi fisici interni al dispositivo. Tali sfasamenti possono influenzare i calcoli intermedi, in particolare la generazione dei riporti e la loro accumulazione nei successivi stadi del moltiplicatore.
Questi ritardi e sfasamenti possono influire sulle prestazioni complessive del circuito. Per ottimizzare il comportamento del moltiplicatore, è possibile adottare strategie come la riduzione delle lunghezze delle interconnessioni o l'ottimizzazione del posizionamento delle risorse logiche per minimizzare i tempi di propagazione e

garantire una migliore sincronizzazione tra gli stadi del moltiplicatore *carry-save*.

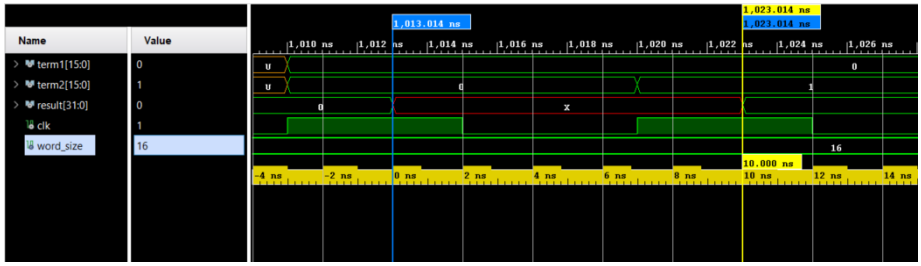

Figure 8: Post Timing Simulation

4.4 Massima frequenza di funzionamento e power analysis

L'analisi della potenza, della temperatura e della frequenza di funzionamento è stata eseguita per due configurazioni di clock differenti: una con un periodo di 12 ns e l'altra con un periodo di 25 ns. I risultati mostrano una differenza significativa nei consumi di potenza dinamica, influenzati direttamente dalla frequenza di clock, e nelle frequenze operative massime.
Per il clock con periodo di 12 ns, la frequenza di funzionamento massima è pari a 83,3333 MHz. In questa configurazione, il consumo totale di potenza *on-chip* è pari a 0,126 W, di cui 27% è dovuto alla potenza dinamica (0,034 W) e il restante 73% alla potenza statica (0,092 W). La componente dinamica è suddivisa tra i clock (4%), i segnali (16%), la logica (14%) e l'I/O (66%). La temperatura di giunzione del dispositivo è risultata essere 26,5°C, con un margine termico di 73,5°C, indicando che il dispositivo opera in condizioni di sicurezza termica.
Nel caso del clock con periodo di 25 ns, la frequenza di funzionamento massima si riduce a 40 MHz. In questa configurazione, il consumo totale di potenza *on-chip* si abbassa a 0,108 W, con una componente dinamica che rappresenta solo il 15% (0,016 W) e una componente statica invariata al 73% (0,092 W). La suddivisione della potenza dinamica è simile, con i clock che consumano meno di 1%, i segnali 14%, la logica 17%, e l'I/O 66%. Anche in questo caso, la temperatura di giunzione è risultata pari a 26,2°C, con un margine termico di 73,8°C.
Questi risultati evidenziano che il periodo di clock influisce significativamente non solo sul consumo di potenza dinamica ma anche sulla frequenza massima di funzionamento. Il clock più rapido (12 ns, corrispondente a 83,3333 MHz) comporta un consumo energetico maggiore, principalmente a causa del maggior numero di commutazioni logiche per unità di tempo. D'altro canto, il clock più lento (25 ns, corrispondente a 40 MHz) consente di ridurre significativamente il consumo dinamico, mantenendo comunque la temperatura del dispositivo ampiamente sotto i limiti di sicurezza.

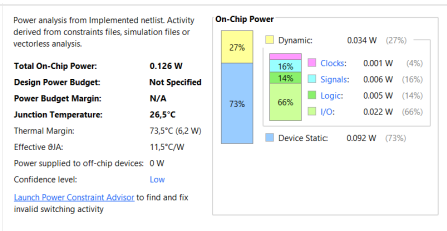

Figure 9: Power Analysis con clock a 12 ns

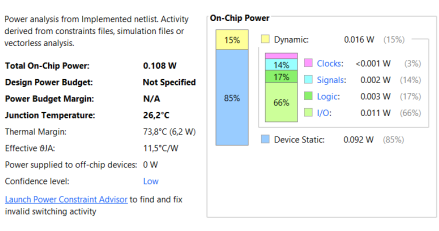

Figure 10: Power Analysis con clock a 25 ns

4.4.1 Timing Analysis

L'analisi delle tempistiche è stata eseguita per confrontare le prestazioni del moltiplicatore con due diverse configurazioni di clock: 12 ns e 25 ns. I risultati mostrano come la variazione del periodo di clock influisca sui parametri di timing.

Table 2: Timing Analysis per clock a 12 ns e 25 ns

Parametro	**Clock 12 ns**	**Clock 25 ns**	**Differenza**
Worst Negative Slack (WNS)	0,872 ns	11,160 ns	+10,288 ns
Worst Hold Slack (WHS)	0,156 ns	0,262 ns	+0,106 ns
Worst Pulse Width Slack (WPWS)	5,500 ns	12,000 ns	+6,500 ns
Total Failing Endpoints	0	0	-

Per il clock con periodo di 12 ns, il Worst Negative Slack (WNS) è pari a 0,872 ns, il che indica che i tempi di setup sono rispettati con un margine accettabile. Tuttavia, con il clock più lento (25 ns), il WNS aumenta notevolmente a 11,160 ns, mostrando che il design ha un margine temporale più ampio grazie alla riduzione della frequenza operativa.
Il Worst Hold Slack (WHS) passa da 0,156 ns con il clock a 12 ns a 0,262 ns con il clock a 25 ns, migliorando lievemente il margine dei tempi di hold.
Infine, il Worst Pulse Width Slack (WPWS), che rappresenta la larghezza minima dell'impulso necessario per garantire il corretto funzionamento del design, aumenta da 5,500 ns con il clock a 12 ns a 12,000 ns con il clock a 25 ns, proporzionalmente alla variazione del periodo di clock.
Entrambe le configurazioni rispettano i vincoli temporali specificati, ma il clock a 25 ns offre margini più ampi, rendendolo più robusto in termini di timing, sebbene a scapito della velocità operativa.
Durante l'analisi temporale, i seguenti parametri sono stati considerati fondamentali per valutare il comportamento del circuito:

- **Worst Negative Slack (WNS)**:
  - **Definizione**: Indica quanto il circuito si avvicina o supera il limite di tempo richiesto per preparare i segnali (*setup time*) prima del prossimo impulso di clock. Se il valore è negativo, significa che il circuito potrebbe non funzionare correttamente alla frequenza specificata.

- **Worst Hold Slack (WHS)**:
  - **Definizione**: Misura quanto i segnali arrivano in anticipo rispetto al tempo minimo richiesto per essere stabili (*hold time*). Un valore negativo indica che i dati potrebbero cambiare troppo presto, causando errori.
- **Worst Pulse Width Slack (WPWS)**:
  - **Definizione**: Rappresenta il margine di tempo rispetto alla durata minima dell'impulso del clock. Un impulso troppo breve potrebbe impedire il corretto funzionamento del circuito.

Questi parametri permettono di verificare che il design rispetti tutti i vincoli temporali e funzioni correttamente alla frequenza desiderata. Il rispetto di valori positivi per WNS, WHS e WPWS è fondamentale per evitare violazioni di timing e garantire la robustezza del circuito.