

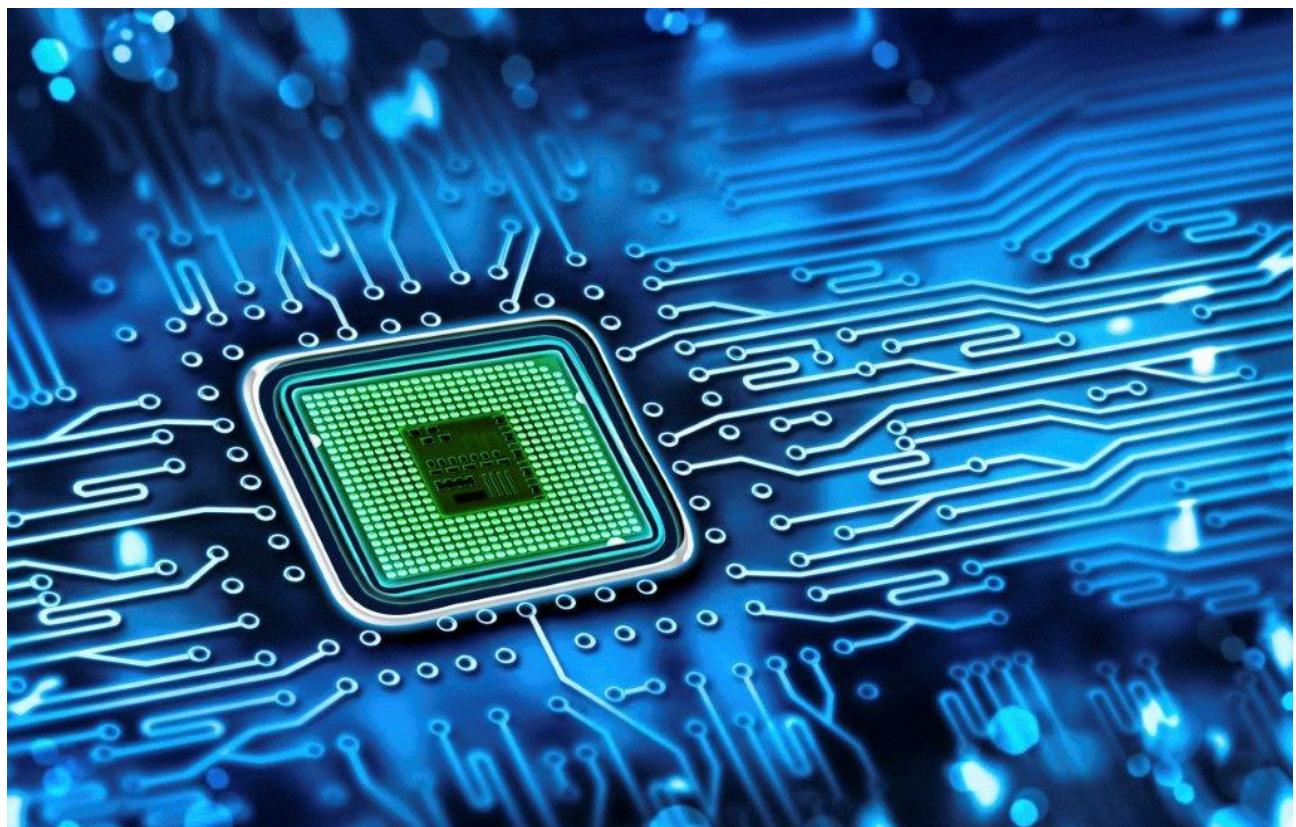
RELAZIONE DI PROGETTO: RAMMUL A 16 BIT (MSB × LSB)

In questo progetto si è realizzato un **RamMUL a 16 bit**, ovvero un circuito che impiega una memoria RAM single port per memorizzare 1024 dati a 16 bit ciascuno e un moltiplicatore che svolge la sua operazione tra **MSB** [*douta(15 downto 8)*] e **LSB** [*douta(7 downto 0)*].

Si è partiti implementando inizialmente i blocchi che lo costituiscono, dunque, facendo uso di una gerarchia.

La strategia adoperata è stata quella di sviluppare i blocchi più semplici per arrivare progressivamente a quelli più complessi.

È stato approfondito, di volta in volta, lo studio di ciascun componente: si è iniziato con i **Registri a *n* bit** proseguendo poi con un **CarrySave a *n* bit** (in grado di effettuare somme in modo più efficiente e in tempi migliori rispetto ad altri sommatori).



DEFINIZIONI

RETI LOGICHE

In elettronica digitale, una **rete logica sequenziale** è un tipo di circuito logico, la cui uscita dipende non solo dal valore dei suoi segnali in ingresso, ma anche dalla sequenza degli ingressi passati, la cosiddetta "storia degli ingressi".

Quanto detto la contraddistingue da una **rete logica combinatoria**, la cui uscita è funzione solo degli ingressi presenti.

Per tale motivo, le reti sequenziali sono dotate di "stati" (**la memoria**), a differenza delle reti combinatorie.

In teoria, tutti i circuiti usati sui dispositivi sono un mix di reti combinatorie e sequenziali.

I circuiti digitali **logico-sequenziali** vengono suddivisi in **sincroni** e **asincroni**.

Nei **circuiti sequenziali sincroni**, lo stato del dispositivo cambia solo in tempi discreti in risposta a un segnale di clock.

Nei **circuiti sequenziali asincroni**, invece, lo stato del dispositivo può variare in qualunque istante in risposta alla variazione degli ingressi.

Il termine **clock**, in elettronica, indica un segnale periodico, generalmente un'onda quadra, utilizzato per sincronizzare il funzionamento dei **dispositivi elettronici digitali**; scansiona, dunque, gli eventi nel tempo.

In elettronica digitale, il **latch** (letteralmente "serratura", "chiavistello") è un circuito elettronico bistabile, caratterizzato quindi da almeno due stati stabili, in grado di memorizzare un bit di informazione nei sistemi a logica sequenziale *asincrona*.

Il clock per i *latch* funge da porta tra l'ingresso D e l'uscita Q:

- nel latch negativo, la porta si apre quando il clock è pari a 0
- nel latch positivo, quando il clock è pari a 1

Nei *flip-flop* l'uscita Q è determinata dai fronti del clock:

- fronte di salita = *rising edge*
- fronte di discesa = *falling edge*

Il **flip-flop** è un circuito sequenziale molto semplice, utilizzato, per esempio, come dispositivo di memoria elementare.

Ne esistono di vari tipi: flip-flop SR, flip-flop JK ecc.

In elettronica e in informatica, la **RAM** (acronimo dell'inglese Random Access Memory ovvero memoria ad accesso casuale, in contrapposizione con la memoria ad accesso sequenziale) è un tipo di memoria volatile, caratterizzata dal permettere l'accesso diretto a qualunque indirizzo di memoria con lo stesso tempo di accesso.

Viste le sue caratteristiche, la memoria RAM è un componente fondamentale per qualsiasi computer.

Ciò, per due motivi principali:

- lavora a stretto contatto con la CPU.
- permette di memorizzare velocemente informazioni e dati.

La sua struttura interna è formata da un insieme di **flip-flop**, opportunamente connessi, mediante un sistema di indirizzamento e trasferimento di parole (gruppi di bit).

I flip-flop della memoria possono dividersi in 2^k gruppi di h flip-flop, ciascuno dei quali memorizza una parola di h bit.

Ogni gruppo è sostanzialmente un registro, detto cella di memoria, cui è logicamente associato un indirizzo di k bit nell'intervallo $[0 : (2^k - 1)]$.

Il sistema di indirizzamento e trasferimento comprende un registro di k bit detto **MAR** (per Memory Address Register), un registro di h bit detto **MBR** (per Memory Buffer Register), collegati con l'esterno, ed inoltre, un decodificatore a k ingressi e 2^k uscite.

Le operazioni di base della memoria sono la **lettura** e la **scrittura** di una parola.

Esse si riferiscono alla cella di memoria il cui indirizzo è specificato in MAR, detta cella selezionata, e consistono nelle azioni seguenti:

1. lettura: il contenuto della cella selezionata è trasferito nel registro MBR.
2. scrittura: il contenuto del registro MBR è trasferito nella cella selezionata.

L'accesso alla cella selezionata è ottenuto mediante un decoder, che attiva la corrispondente linea di selezione cella.

Il trasferimento di una parola da una cella verso MBR (lettura), e da MBR verso una cella (scrittura), avviene in parallelo.

Il termine **pipeline** indica una tecnica per l'implementazione del *parallelismo* a livello di istruzione all'interno di un singolo processore.

Il **pipelining** cerca di mantenere ogni *core* del processore occupato con alcune istruzioni da svolgere, dividendo le istruzioni in arrivo in una serie di passaggi sequenziali eseguiti da diverse unità del processore con diverse parti di istruzioni elaborate in parallelo.

Il lavoro svolto da un processore con *pipelining*, per eseguire un'istruzione, è diviso in passi (stadi della pipeline), che richiedono una frazione del tempo necessario all'esecuzione dell'intera istruzione.

Gli stadi sono connessi in maniera seriale per formare la pipeline. Le istruzioni:

1. entrano da un'estremità della pipeline;
2. vengono elaborate dai vari stadi secondo l'ordine previsto;
3. escono dall'altra estremità della pipeline.

Il tempo necessario per fare avanzare un'istruzione di uno stadio lungo la pipeline corrisponde ad un ciclo di clock di pipeline, per questo motivo, poiché gli stadi della pipeline sono collegati in sequenza, devono operare in modo sincrono.

LINGUAGGIO VHDL

TIPO di input/output

Il tipo STANDARD LOGIC amplia la possibilità di rappresentazione e, per poterlo utilizzare, bisogna importare ad inizio pagina la libreria in cui è presente:

```
library IEEE; --rende visibile la libreria IEEE
```

```
use IEEE.STD_LOGIC_1164.ALL; --rende visibili i contenuti del package STD_LOGIC_1164
```

Per questo motivo quelli che sono stati chiamati bit per comodità, in realtà vengono definiti come STD_LOGIC nel codice scritto.

Il package *IEEE.STD_LOGIC_UNSIGNED.ALL* permette il calcolo *unsigned* di operazioni ad alto livello, quali ad esempio “**+**” (utilizzato nel file **Usa_RAM**) e “*****” (utilizzato nel file **Test_MyRAM**, per testare la correttezza del circuito).

COMPONENT

Il **VHDL** permette una modellazione gerarchica, ovvero assemblare un modulo attraverso sotto-moduli; ciò avviene con l'utilizzo della parola chiave **COMPONENT** che serve per richiamare i codici già scritti in altri file di testo.

PROCESS

Il **PROCESS** aiuta a gestire la sequenzialità delle istruzioni, ovvero la loro dipendenza dal tempo.

SCHEMATIC

Eseguendo la **sintesi**, si è ottenuta una mappa (**SCHEMATIC**) che rappresenta le vere porte logiche, messe a disposizione dalla piattaforma scelta ad inizio progetto: [ZedBoard Zynq Evaluation and Development Kit](#). Quanto descritto rappresenta l'hardware.

Sono state ricavate tabelle di verità sotto forma di *LUT* (che usano *FPGA*).

Dopo aver eseguito la sintesi, si è proceduto con l'implementazione per poter fissare i collegamenti.

GENERIC

Generic supporta le informazioni statiche nei blocchi in modo simile alle costanti, ma a differenza di esse, i valori di **generic** possono essere forniti esternamente.

Analogamente alle porte, anch'essi possono essere dichiarati nell'*entity* e nei *component*, definiti prima delle porte.

I valori di **generic** dichiarati nelle *entity* possono essere letti nell'*entity* stessa o nell'*architecture* associata.

In particolare, è possibile utilizzare un **generic** per:

- specificare la dimensione delle porte
- il numero di sottocomponenti all'interno di un blocco
- le caratteristiche temporali di un blocco
- le caratteristiche fisiche di un progetto

- la larghezza dei vettori all'interno di un'architecture
- numero di iterazioni di loop, ecc.

For Generate

Lo statement del **for generate** è generalmente usato per istanziare “array” di componenti. Il parametro generate può essere usato per indicizzare *signal* di tipo array associati con porte di vari componenti. La struttura generale è:

```
etichetta: for parametro in un range generate
    statement concorrenti
end generate etichetta;
```

Lo statement **for generate** è particolarmente potente quando usato con dei **generic**.

Viceversa, il **for loop** definisce un parametro di loop che riceve valori da un range definito. Per esempio, un intervallo *0 to 3*. La struttura generale è:

```
for parametro in un range loop
    statement sequenziali
end loop;
```

Block Memory

Si realizza un banco di memoria impiegando le primitive disponibili nel chip **FPGA** di cui stiamo facendo uso.

Selezionando la categoria “Memories & Storage Elements”, visualizziamo altre sottocategorie che fanno riferimento a delle particolari strutture di memoria.

Project Manager > IP Catalog > Memories & Storage Elements

> **ECC**

ECC è un modulo legato alle memorie poiché svolge una funzione di *encoding* e *decoding*.

> **RAMs & ROMs**

> *Distributed Memory Generator*

Mette a disposizione dei banchi di memoria realizzati mediante le **LUT** (*Look Up Table*).

Dunque, oltre che per la realizzazione di funzioni logiche, le LUT possono essere utilizzate anche per realizzare delle memorie distribuite.

> **RAMs & ROMs & BRAM**

Blocchetti di memoria customizzati dall’utente sull’impiego di **BRAM** (ovvero *Block Ram*).

> *Block Memory Generator*

All’interno dei chip **FPGA** sono disponibili dei piccoli banchi di memoria che hanno una capacità di memorizzazione di 18 kbit o 36 kbit che vengono messi insieme per realizzare un banco di memoria più grande a seconda dei settaggi che si specificano.

Nel nostro **RamMUL** utilizziamo la primitiva BRAM.

Viene resa disponibile una finestra nella quale si possono settare tutte le caratteristiche che ha il blocco di memoria interno al chip.

Memory type → Single Port RAM.

Interface Type → Native.

Nativa = più semplice possibile:

- nel caso della **scrittura**, basata sulla specifica di un indirizzo e la proposizione di un dato.
- nel caso della **lettura**, basata sulla specifica di un indirizzo.

AXI4 = interfaccia basata su un protocollo più complicato, utilizzato e supportato per la realizzazione di sistemi EMBEDDED.

Write Enable:

- **Byte Write Enable**

Caratteristica utilizzata quando l’utente vuole abilitare la scrittura del dato byte a byte: se la parola di memoria è superiore a 8 bit, con questo flag, si determina la scrittura un byte per volta.

Algorithm → Minimum Area

Minimum Area e Low Power:

con queste opzioni, le due tipologie di **Block Ram** disponibili all'interno del chip (18 kbit e 36 kbit) vengono utilizzate in maniera mista per minimizzare l'**occupazione dell'area** o la **dissipazione di potenza**.

Questo viene eseguito automaticamente dal generatore della memoria.

Fixed Primitives:

Quando si vuole utilizzare solo un tipo di primitiva (o solo 18 kbit o solo 36 kbit).

Memory size

WRITE WIDTH = 16 bit

READ WIDTH = 16 bit

Write Depth = 1024 = quante parole di 16 bit si vogliono memorizzare all'interno della memoria.

Read Depth = 1024

Operating mode = caratteristica che determina il tipo di comportamento che deve avere la memoria se acceduta; ovvero, in caso di conflitto tra operazione di *write* e *read*, quale dev'essere eseguita prima.

"No Change" = in questo modo si evita che in uscita alla memoria vi siano variazioni indesiderate e si limita il numero di commutazioni avendo un effetto benefico in termini di dissipazione di potenza.

Enable Port Type = si può stabilire se la memoria dev'essere sempre abilitata o se bisogna avere un PIN apposito.

È preferibile usare un PIN poiché nel *Testbench* o in un'opportuna macchina di controllo, sia possibile stabilire se la porta dev'essere abilitata oppure no.

Port A Optional Output Registers

Di default il generatore del banco di memoria va a piazzare all'uscita del banco un registro, che introduce un ciclo di latenza rispetto alla funzionalità di lettura, quando andiamo ad accedere ai dati in lettura.

Questa caratteristica può essere utile quando in uscita dalla memoria si vuole inserire un livello di pipeline.

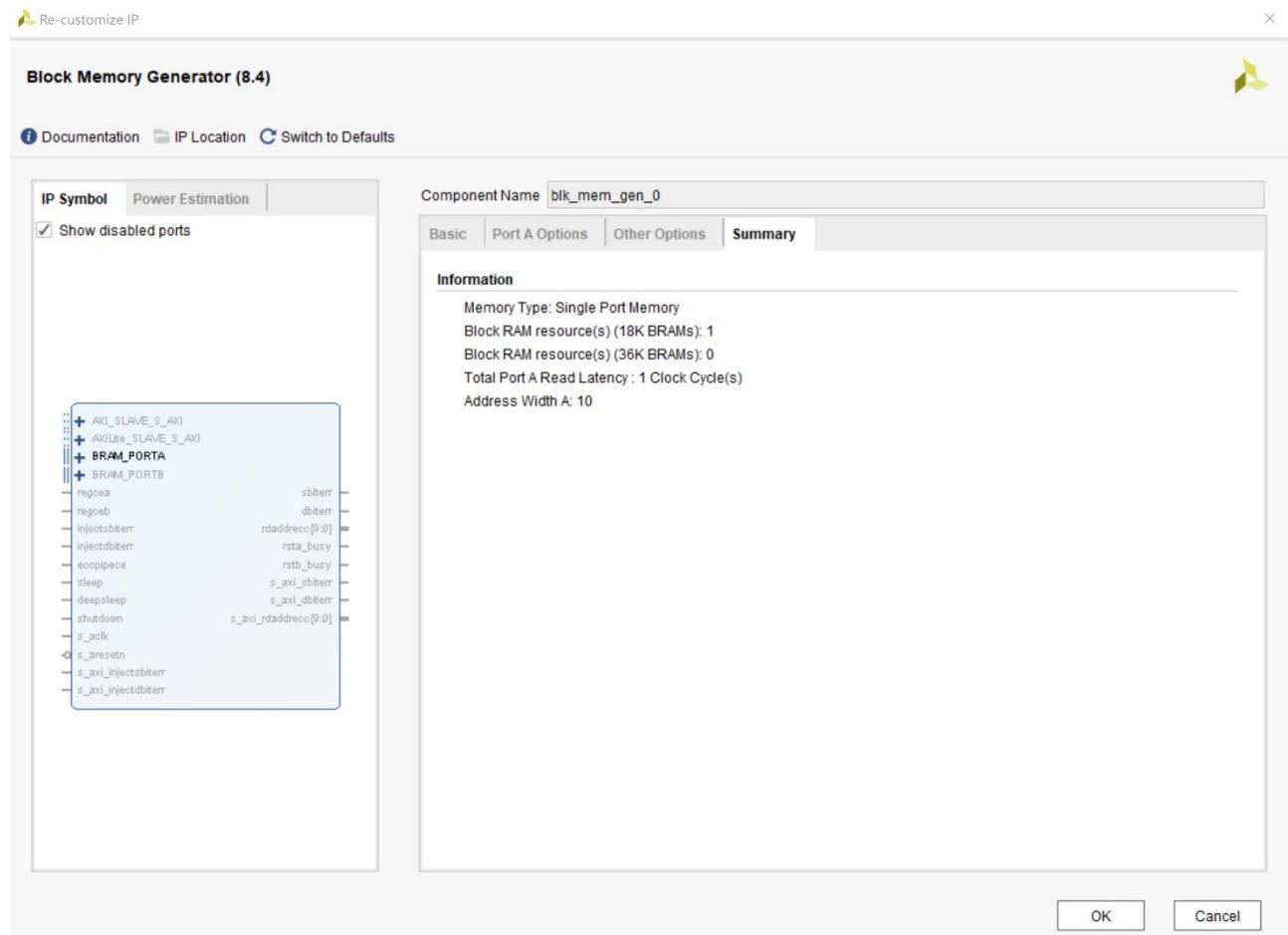
Scriviamo i dati nella memoria tramite il **Testbench**.

Il nostro file di simulazione è suddiviso in due operazioni:

- scrittura
- lettura

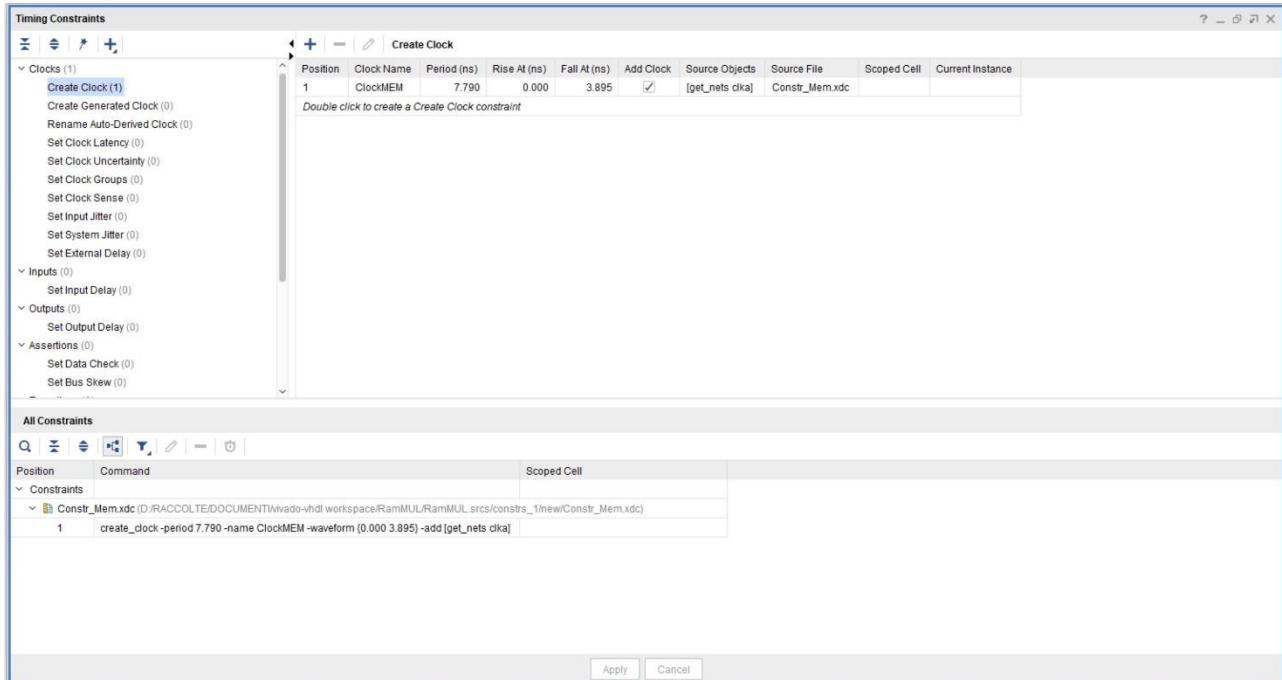
Total Port A Read Latency = porta che viene utilizzata sia per scrittura che per lettura e nel caso della lettura, il dato verrà reso disponibile in uscita alla memoria con un ciclo di latenza.

Poiché abbiamo fissato come profondità della memoria 1024, l'indirizzo con cui poter accedere sia in scrittura che in lettura i nostri dati sarà composto da 10 bit.



clock

TIMING CONSTRAINTS



Constraint Contr_Mem

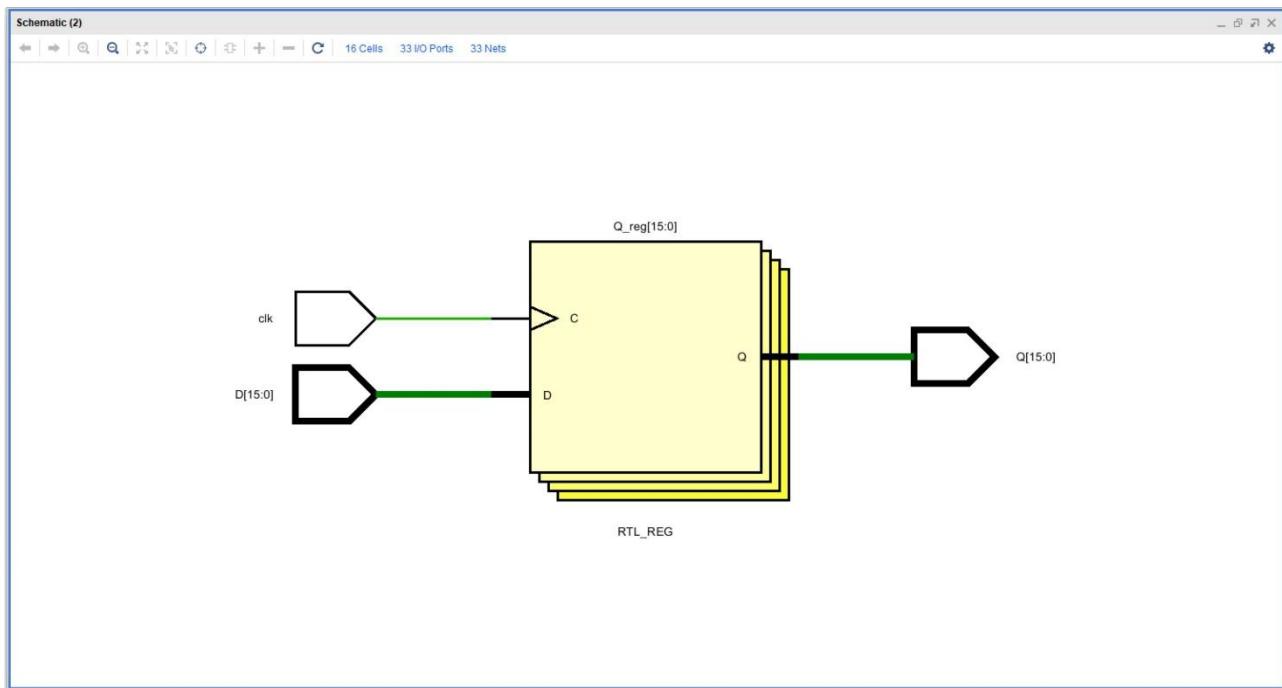
The screenshot shows the Vivado Constraints Editor window titled "Constr_Mem.xdc". The path is D:/RACCOLTE/DOCUMENTI/vivado-vhdl workspace/RamMUL/RamMUL.srsrcs/constrs_1/new/Constr_Mem.xdc. The toolbar includes search, filter, and other file operations.

```
create_clock -period 7.790 -name ClockMEM -waveform {0.000 3.895} -add [get_nets clka]
```

The code editor shows the single line of XDC code defining the clock constraint.

Registro

Schematic



Definizione

In elettronica digitale, i **registri hardware** sono un tipo di circuito tipicamente composto da **flip-flop**, spesso con molte caratteristiche simili alla memoria, come:

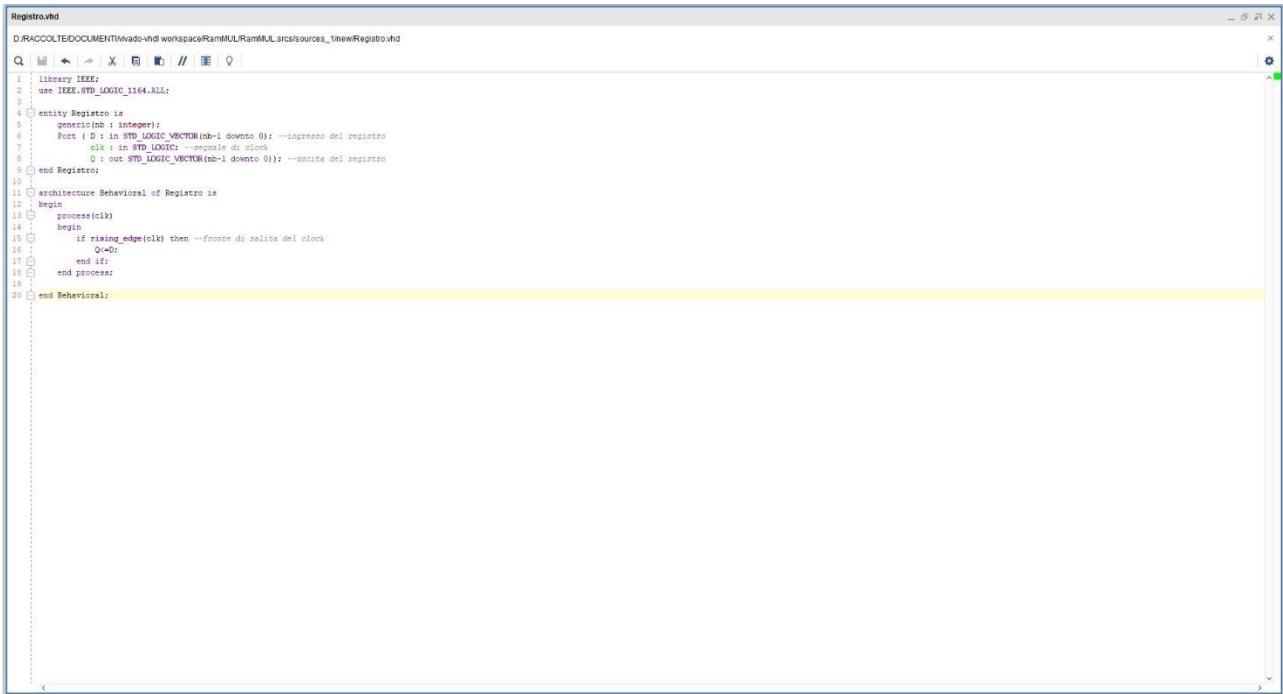
- L'abilità di leggere (*read*) o scrivere (*write*) multipli *bit* simultaneamente.
- Usare un indirizzo per selezionare un particolare registro in maniera simile a un indirizzo di memoria.

Nella pratica, i registri hardware sono usati come interfaccia tra il lato software e le periferiche. I software scrivono sui registri per inviare informazioni al dispositivo, e viceversa.

Alcuni dispositivi hardware inoltre includono registri utili per il loro uso interno, ma non visibili al software.

Nel progetto realizzato, il **Registro** è stato implementato seguendone la definizione: in particolare è stato reso sensibile al fronte di salita del segnale di clock (*rising edge*), in modo da memorizzare il valore in ingresso **D** (uno **STD_LOGIC_VECTOR** generico in *n* bit) e renderlo disponibile come uscita **Q** (sempre **STD_LOGIC_VECTOR** generico in *n* bit).

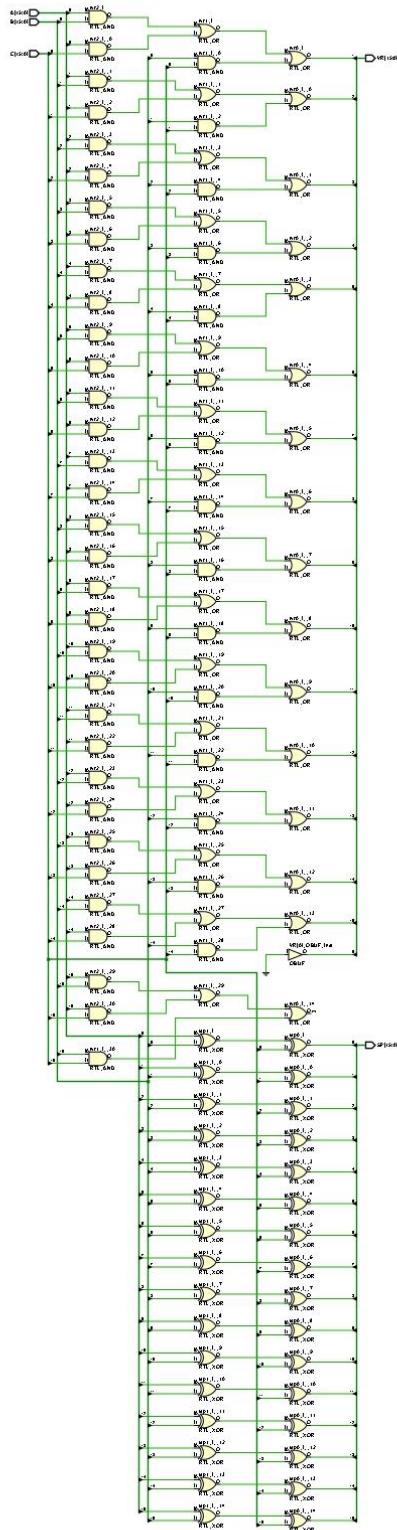
Codice



```
Registro.vhd
D:\ACCOLTEDOCUMENTI\adob\vhdl\workspace\RamMUL\RamMUL_src\source_1newRegistro.vhd

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity Registro is
    generic(width : integer);
    port ( D : in STD_LOGIC_VECTOR (width-1 downto 0); --ingresso del registro
           clk : in STD_LOGIC; --segnale di clock
           Q : out STD_LOGIC_VECTOR (width-1 downto 0)); --uscita del registro
end Registro;
architecture Behavioral of Registro is
begin
process(clk)
begin
    if rising_edge(clk) then --fronte di salita del clock
        if D=>0 then
            Q=>0;
        end if;
    end process;
end Behavioral;
```

CarrySave Schematic



Definizione

Un **Carry-Save Adder** è un tipo di sommatore utilizzato per gestire efficientemente la somma di tre o più numeri binari (*operands*).

Si differenzia dagli altri *adders* in quanto esso non restituisce il risultato esatto, ma due numeri o più a seconda delle implementazioni e il risultato esatto della somma originale deve essere ottenuto combinando insieme, di solito attraverso una somma, queste uscite.

Un **Carry-Save Adder** è tipicamente usato in circuiti più complessi come il moltiplicatore binario: il moltiplicatore “*carta e penna*” durante il calcolo impiega una somma di più di due numeri binari (somma i risultati parziali).

Un **adder** più grande implementato usando questa tecnica sarà solitamente molto più veloce di uno che usa l'addizione convenzionale.

Esempio (in decimale):

Consideriamo la somma:

$$\begin{array}{r} 12345678 + \\ \underline{87654322} = \\ 100000000 \end{array}$$

Usando l'aritmetica di base, calcoliamo da destra a sinistra, "8 + 2 = 0 e porto 1", "7 + 2 + 1 = 0 e porto 1", "6 + 3 + 1 = 0 e porto 1" e così via fino alla fine della somma.

Sebbene conosciamo l'ultima cifra del risultato ogni volta, non possiamo conoscere la prima cifra finché non abbiamo esaminato ogni cifra nel calcolo, passando il riporto da ciascuna cifra a quella alla sua sinistra.

Quindi la somma di due numeri a *n cifre* deve richiedere un tempo proporzionale a *n*, anche se la macchina che stiamo usando fosse in grado di eseguire molti calcoli simultaneamente.

In elettronica, e in particolare nella progettazione dei circuiti sommatori, si utilizzano ovviamente i bit (cifre binarie e non decimali!), ciò significa che anche se abbiamo a disposizione *n* sommatori di un bit a nostra disposizione, dobbiamo comunque attendere un tempo proporzionale ad *n* per consentire a un possibile riporto di propagarsi da un'estremità del numero all'altra.

Fino a quando ciò non sarà avvenuto:

- Non conosciamo il risultato della somma.
- Non sappiamo se il risultato dell'addizione sia maggiore o minore di un dato numero (ad esempio, non sappiamo se sia positivo o negativo nei casi in complemento a 2).

La soluzione al problema sopra riportato, si basa appunto sull'idea di sommare *in cascata* (in parallelo) ogni coppia (o tripla, o *n-pla* in generale) di cifre dei possibili operandi e poi combinare opportunamente i risultati.

In particolare, per rispettare la definizione di **Carry-Save Adder** sono stati usati:

- Tre ingressi (*operandi*) a n bit (**A**, **B** e **C**).
 - Un’uscita somma parziale (**SP**), sempre a n bit, essa contiene le somme “*pure*” ovvero non influenzate dal riporto.
 - Un’uscita vettore di riporto (**VR**), come sopra a n bit, contenente i riporti delle somme.

Oltre all'implementazione generale, sono state quindi utilizzate delle *signal*:

- Due segnali **tsp** e **tvr**, creati come **STD_LOGIC_VECTOR** generici a n bit, usati per gestire le somme e i riporti (*la t nella loro definizione sta per temporaneo*).

Infine, con l'utilizzo del costrutto ***for generate*** disponibile in VHDL, al variare dell'indice *i* verranno generati singoli **Full-Adder** in grado di eseguire le somme in parallelo.

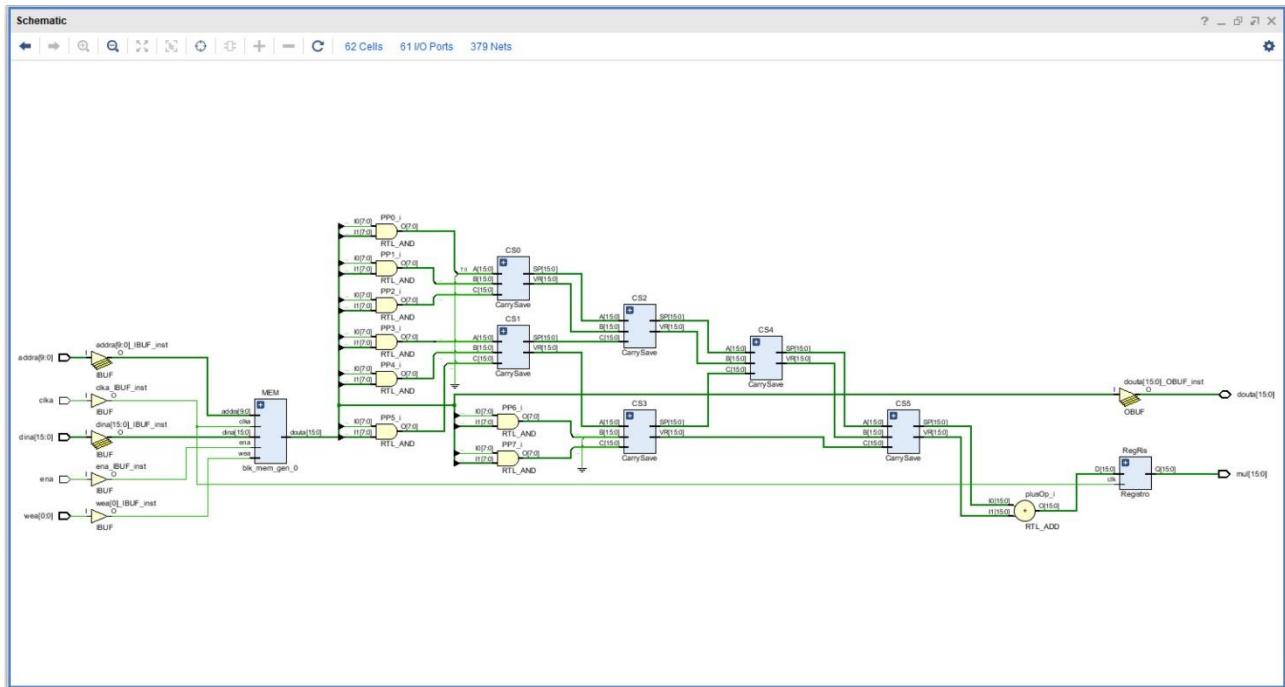
Codice

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity CarrySave is
5     generic (nb : integer);
6     Port ( A : in STD_LOGIC_VECTOR(nb-1 downto 0); --ingresso A
7            B : in STD_LOGIC_VECTOR(nb-1 downto 0); --ingresso B
8            C : in STD_LOGIC_VECTOR(nb-1 downto 0); --ingresso C
9            SF : out STD_LOGIC_VECTOR(nb-1 downto 0); --somma SP tra A, B e C
10           VR : out STD_LOGIC_VECTOR(nb-1 downto 0)); --vettore di riporto
11 end CarrySave;
12
13 architecture Behavioral of CarrySave is
14
15 signal tpr, tvr : STD_LOGIC_VECTOR(nb-1 downto 0);
16
17 begin
18
19     --STEP 1
20
21     GEN_FA:
22         for i in 0 to (nb-1) generate --for generate
23             --PURE ADDRESS
24             begin
25                 tpr(i) <= A(i) XOR B(i) XOR C(i);
26                 tvr(i) <= (A(i) AND B(i)) OR (A(i) AND C(i)) OR (B(i) AND C(i));
27             end generate GEN_FA;
28
29     --STEP 2
30     SF<- tpr;
31     VR<- (tvr(nb-2 downto 0))*'0';
32
33 end Behavioral;
```

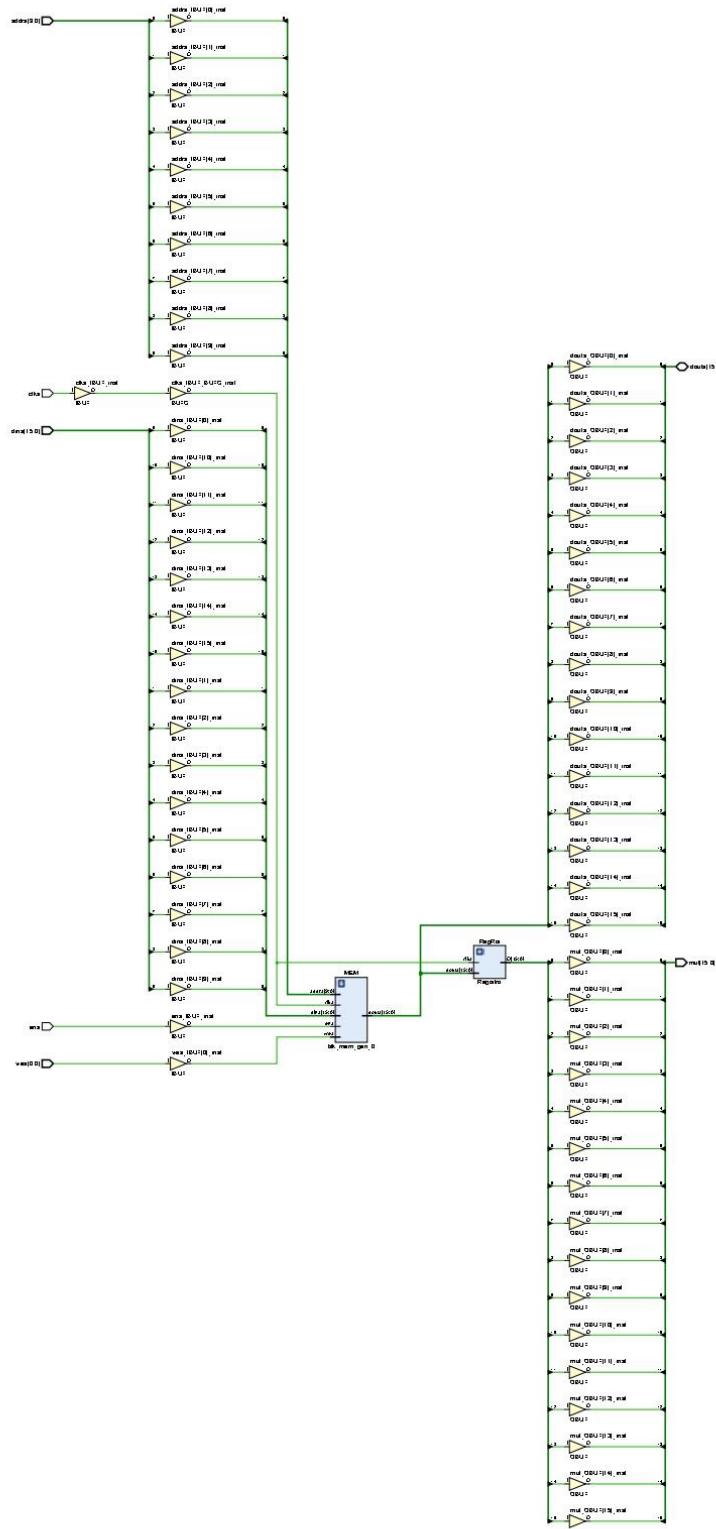
Usa_RAM

Schematic

RTL ANALYSIS > Schematic



IMPLEMENTATION > Schematic



Definizione

Il circuito discusso in questo paragrafo, che abbiamo scelto di chiamare **Usa_RAM**, può essere visto come composizione di circuiti più semplici già trattati in precedenza: un'unità **RAM**, un **AdderTree** composto da sei **Carry-Save** e un **registro** per memorizzare il risultato finale in *pipeline*.

Considerato che il circuito dovrà svolgere l'operazione algebrica $MSB \times LSB$, i cui valori sono presi da **douta** (dal bit in posizione 15 alla posizione 8 per l'MSB e dalla posizione 7 alla posizione 0 per l'LSB), con gli operandi a 8 bit, e valutando che un prodotto di due operandi può essere svolto come somme di *prodotti parziali*, l'implementazione scelta consiste di:

- Un ingresso **clka**, uno **STD_LOGIC**, segnale di clock in ingresso.
- Un ingresso **ena**, anch'esso **STD_LOGIC**, segnale di abilitazione per controllare che il banco di memoria sia accessibile o meno.
- Un ingresso **wea**, questa volta **STD_LOGIC_VECTOR** che, caso particolare, è un vettore di un solo elemento; segnale di Write Enable attivo alto (se il suo valore è 1, la memoria viene scritta, se 0, viceversa, viene letta).
- Un ingresso **addr**, uno **STD_LOGIC_VECTOR** che rappresenta l'indirizzo con cui si accede alle celle di memoria.
- Un ingresso **dina**, anch'esso **STD_LOGIC_VECTOR**, indica la porta utilizzata per fissare il dato da scrivere.
- Un ingresso/uscita (INOUT) **douta**, **STD_LOGIC_VECTOR**, porta utilizzata per leggere il dato (come scritto sopra, è da qui che si ricavano gli operandi per la moltiplicazione).
- Un'uscita risultato **mul**, anche questa volta **STD_LOGIC_VECTOR**, che contiene il risultato finale dell'operazione $MSB \times LSB$.

Come nel caso del **Carry-Save** generico a n bit, sono state usate delle *signal*:

- Serie di segnali **v0, v1, v2, v3, v4, v5, v6, v7**, creati come **STD_LOGIC_VECTOR** ad 8 bit, usati per raccogliere e replicare l'*i-esimo* di LSB (0, 1 ecc. fino a 7).
- Serie di segnali **PP0, PP1, PP2, PP3, PP4, PP5, PP6, PP7**, anch'essi **STD_LOGIC_VECTOR** ad 8 bit, creati per memorizzare il *prodotto logico* (l'AND) tra l'*i-esimo* segnale **v** e l'**MSB**.
- Serie di segnali **R0, R1, R2, R3, R4, R5, R6, R7**, **STD_LOGIC_VECTOR** a 16 bit, che contengono i prodotti parziali opportunatamente shiftati con l'operatore & (estensione).

ADDER TREE

Per svolgere la somma di tutti i risultati parziali ricavati (**R0, R1, R2, R3, R4, R5, R6, R7**) si è fatto ricorso ad un **Adder Tree**, strutturato in cinque livelli e ad ulteriori *signal* di supporto:

Primo livello

Il primo livello dell'AdderTree è costituito da due **Carry-Save** che ricevono, come visto nel paragrafo dedicato, tre operandi ciascuno.

Gli operandi sono i risultati parziali ricavati precedentemente (**R0, R1, R2** per il primo sommatore, **R3, R4, R5** per il secondo).

In uscita da ciascun **Carry-Save** vi saranno due vettori, rispettivamente, uno di somme parziali (**sp0, sp1**) e uno di riporti (**vr0, vr1**).

Secondo livello

Il secondo livello dell'AdderTree è costituito, anch'esso, da due **Carry-Save**.

Gli operandi sono le uscite dei **Carry-Save** del livello precedente: il primo **Carry-Save** riceve **sp0, vr0, sp1** e il secondo **vr1, R6, R7**. In uscita da ciascun **Carry-Save** vi saranno due vettori, rispettivamente, uno di somme parziali (**sp2, sp3**) e uno di riporti (**vr2, vr3**).

Terzo livello

Il terzo livello dell'AdderTree è costituito stavolta da un solo **Carry-Save**.

Gli operandi sono **sp2, vr2** e **sp3**. In uscita dal **Carry-Save** vi saranno due vettori: **sp4** e **vr4**.

Quarto livello

Il quarto livello dell'AdderTree è anch'esso costituito da un solo **Carry-Save**.

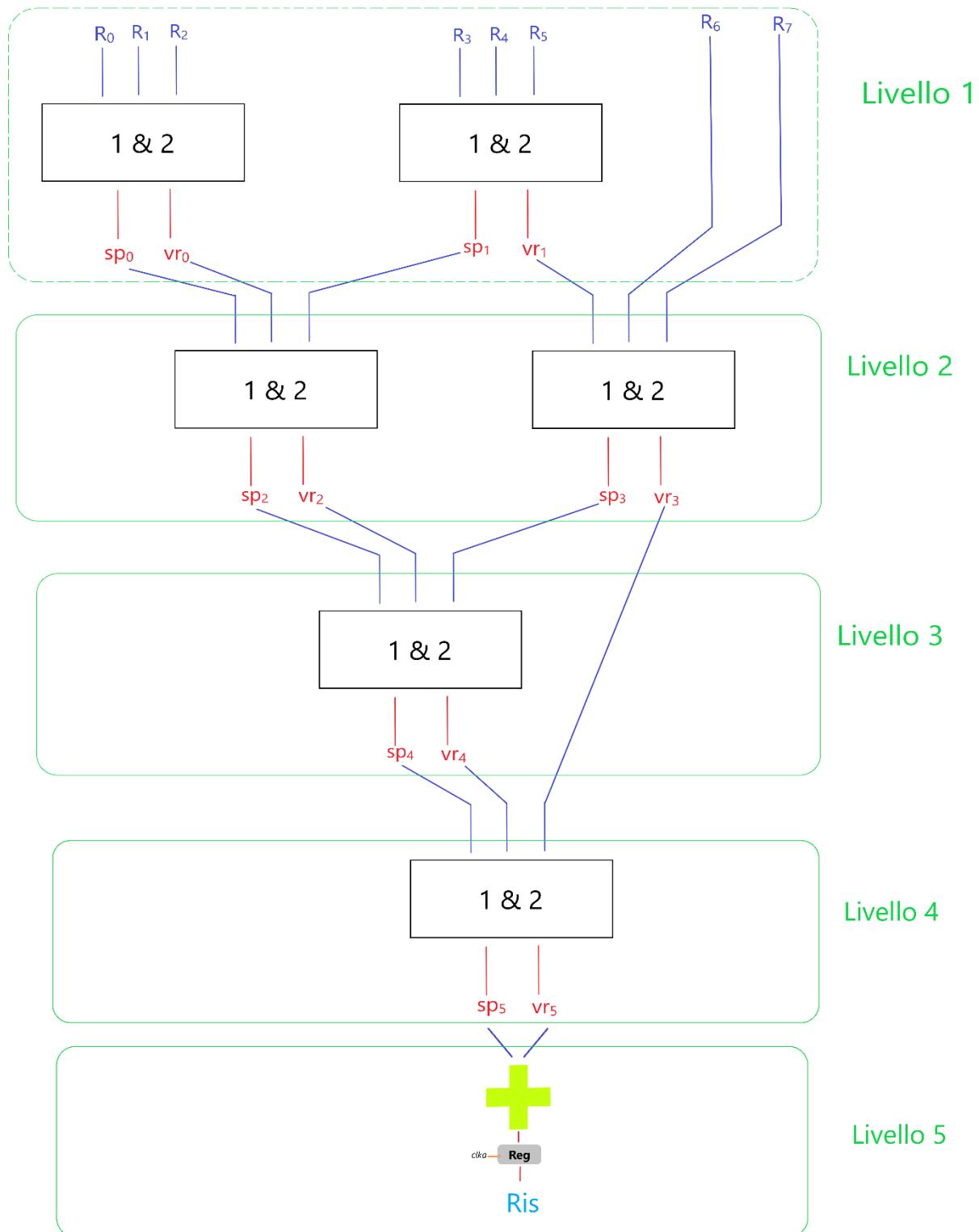
Gli operandi sono **sp4, vr4** e **vr3**. In uscita dal **Carry-Save** vi saranno due vettori: **sp5** e **vr5**.

Quinto livello

Si importa la libreria *UNSIGNED* per poter eseguire l'operazione di somma impiegando l'operatore ad alto livello predefinito.

Il risultato di questa operazione, che avviene tra **sp5** e **vr5**, tramite un registro sensibile al fronte di salita del clock viene inviato in uscita come risultato finale (**mul**).

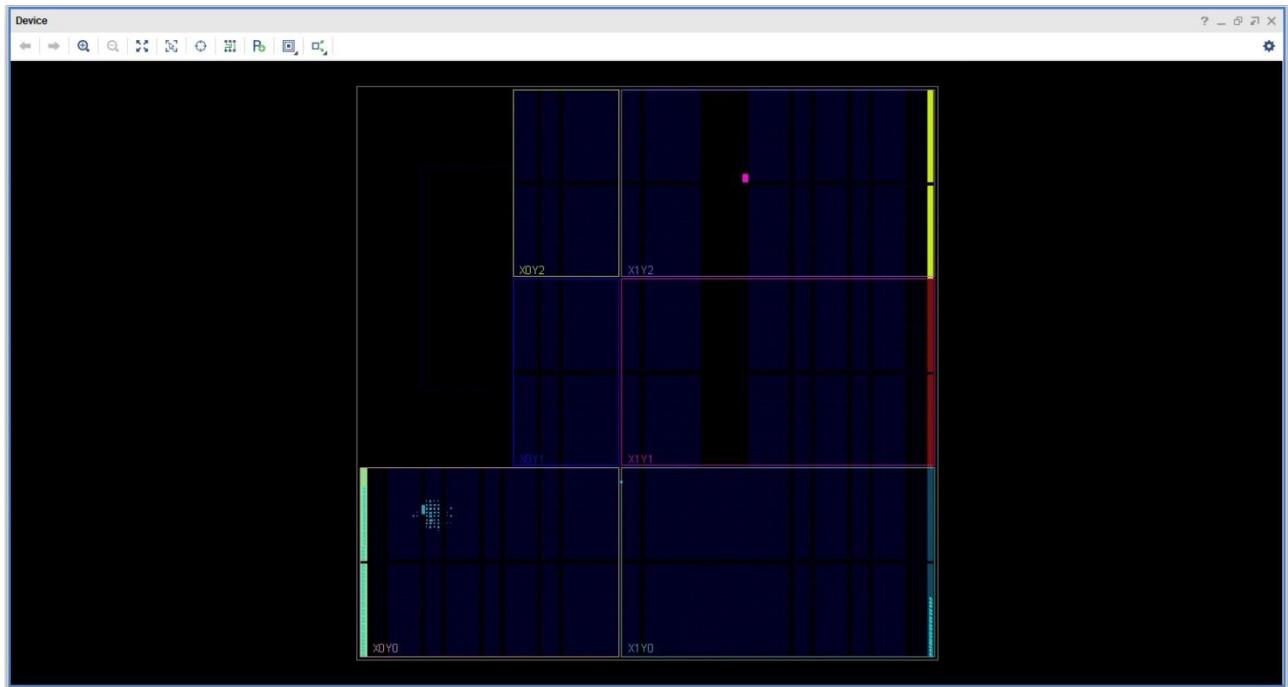
Schema dell'Adder Tree



Codice

Device (*Implemented Design*)

DEVICE



Frequenza di funzionamento (*Report Timing Summary*)

IMPL_ROUTE_REPORT_TIMING_SUMMARY

Risorse occupate (*Report Utilization*)

IMPLEMENTATION_1_PLACE_REPORT_UTILIZATION

```

impl_1_place_report_utilization_0 - impl_1
D:\RACCOLTE\DOCUMENTI\Wedo-vhdl\workspace\RamfUL\RamMUL\runs\impl_1\Usa_RAM_utilization_placed.rpt
Q | < | > | X | B | // | █ | ? | Read-only | X |
28 1. Slice Logic
29 -----
30
31 +-----+
32 | Site Type | Used | Fixed | Available | Util% |
33 +-----+
34 | Slice Logic | 132 | 0 | 53200 | 0.25 |
35 | LUT as Logic | 132 | 0 | 53200 | 0.25 |
36 | LUT as Memory | 0 | 0 | 17400 | 0.00 |
37 | Slice Registers | 16 | 0 | 106400 | 0.02 |
38 | Register as Flip Flop | 16 | 0 | 106400 | 0.02 |
39 | Register as Latch | 0 | 0 | 106400 | 0.00 |
40 | FF Muxes | 0 | 0 | 24600 | 0.00 |
41 | FB Muxes | 0 | 0 | 13200 | 0.00 |
42 +-----+
43
44 1.1 Summary of Registers by Type
45 -----
46
47 +-----+
48 | Total | Clock Enable | Synchronous | Asynchronous |
49 +-----+
50 | | | | |
51 | 0 | - | - | - |
52 | 0 | - | - | Set |
53 | 0 | - | - | Reset |
54 | 0 | - | Set | - |
55 | 0 | - | Reset | - |
56 | 0 | Yes | - | - |
57 | 0 | Yes | - | Set |
58 | 0 | Yes | - | Reset |
59 | 0 | Yes | Set | - |
60 | 16 | Yes | Reset | - |
61 +-----+
62
63
64 2. Slice Logic Distribution
65 -----
66
67 +-----+
68 | Site Type | Used | Fixed | Available | Util% |
69 +-----+
70 | Slice | 48 | 0 | 13200 | 0.32 |
71 | SLICEL | 32 | 0 | 0 |
72 | SLICEN | 20 | 0 | 0 |
73 | LUT as Logic | 132 | 0 | 53200 | 0.25 |
74 | using OS output only | 2 | 0 | 0 |
75 | using OS output only | 116 | 0 | 0 |
76 | using OS output only | 14 | 0 | 0 |
77 | LUT as Memory | 0 | 0 | 17400 | 0.00 |
78 | LUT as Distributed RAM | 0 | 0 | 0 |
79 | LUT as Shift Register | 0 | 0 | 0 |
80 | LUT FF or FF Pairs | 16 | 0 | 53200 | 0.03 |
81 | Only one LUT-FF pairs | 0 | 0 | 0 |
82 | LUT-FF pairs with one unused LUT output | 14 | 0 | 0 |
83 | LUT-FF pairs with one unused Flip Flop | 14 | 0 | 0 |
84 | Unique Control Sets | 1 | 0 | 0 | 0 |
85 +-----+
86 * Note: Review the Control Sets Report for more information regarding control sets.
87
88
89 3. Memory
90 -----
91
92 +-----+
93 | Site Type | Used | Fixed | Available | Util% |
94 +-----+
95 | Block RAM Tile | 0.5 | 0 | 140 | 0.36 |
96 | Block FIFO* | 0 | 0 | 140 | 0.00 |
97 | RAM16 | 1 | 0 | 200 | 0.36 |
98 | RAMB18E1 only | 1 | 0 | 0 | 0 |
99 +-----+
100 * Note: Each Block RAM Tile only has one FIFO logic available and therefore can accommodate only one FIFO36E1 or one FIFO18E1. However, if a FIFO18E1 occupies a Block RAM Tile, that tile can still accommodate a RAMB18E1
101
102
103 0. Primitives
104 -----
105
106 +-----+
107 | Ref Name | Used | Functional Category |
108 +-----+
109 | LUT2 | 57 | LUT |
110 | LUT6 | 50 | LUT |
111 | LUT12 | 32 | ID |
112 | IDFY | 20 | ID |
113 | LUT4 | 18 | LUT |
114 | FDRE | 16 | Flip & Latch |
115 | LUT3 | 11 | LUT |
116 | DFFE | 10 | LUT |
117 | ROM16 | 1 | Cache/Logic |
118 | RAMB18E1 | 1 | Block Memory |
119 | BUFG | 1 | Clock |
120 +-----+
121 <

```

Utilization

Hierarchy

Name	Slice LUTs (53200)	Slice Registers (106400)	Slice (13300)	LUT as Logic (53200)	LUT Flip Flop Pairs (53200)	Block RAM Tile (140)	Bonded IOB (200)	BUFGCTRL (32)
Usa_RAM	132	16	42	132	16	0.5	61	1
> MEM (blk_mem_gen_0)	0	0	0	0	0	0.5	0	0
> RegRis (Registro)	132	16	42	132	16	0	0	0

Memory

DSP

IO and GT Specific

Clocking

Primitives

Black Boxes

utilization_1

Utilization

Summary

Resource	Utilization	Available	Utilization %
LUT	132	53200	0.25
FF	16	106400	0.02
BRAM	0.50	140	0.36
IO	61	200	30.50

Hierarchy

Summary

Utilization

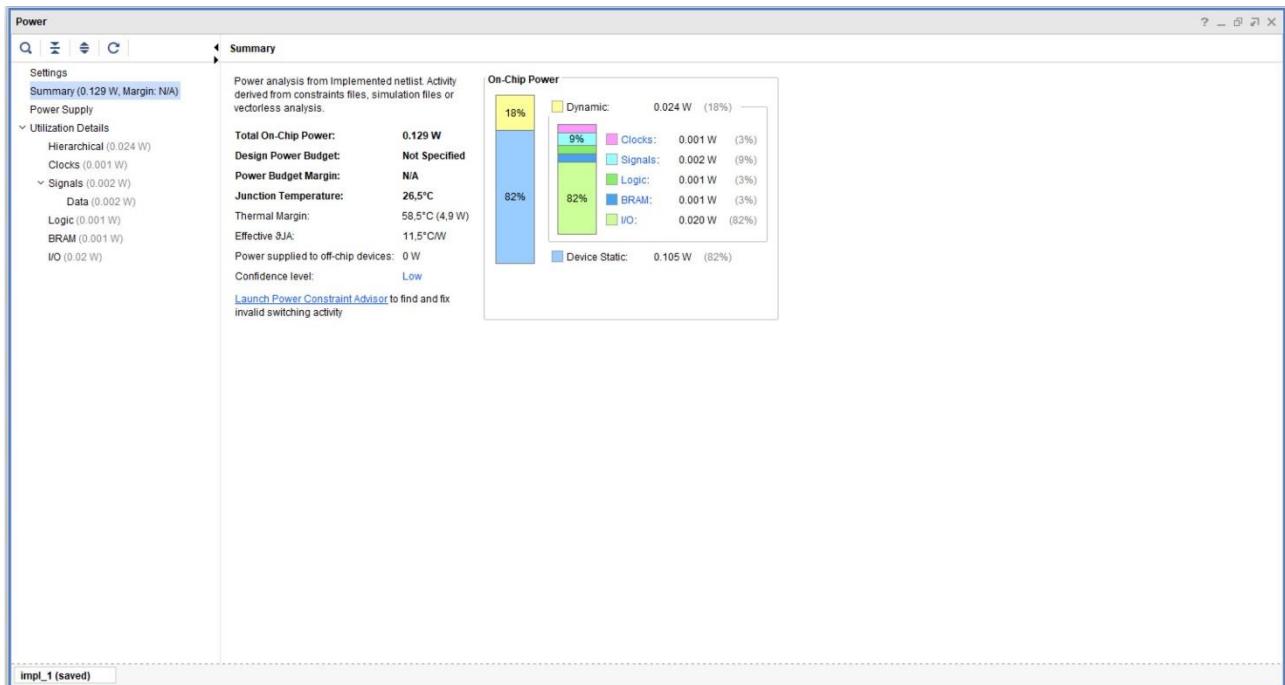
Primitives

Black Boxes

utilization_1

Dissipazione di potenza

(Implementation > Report Power)



Test_MyRAM

Per eseguire un TEST bisogna scrivere un altro file VHDL che prende il nome di **TESTBENCH**, ovvero un file dedicato alle simulazioni che non è né sintetizzabile né implementabile (nel nostro caso è chiamato **Test_MyRAM**).

Il **TESTBENCH** gestisce la variazione degli ingressi nel tempo.

La keyword **wait for** si utilizza per attendere un determinato tempo prima di proseguire con le istruzioni.

La keyword **others** si utilizza per assegnazioni indipendenti dalla grandezza del vettore.

Da **Odouta**, è possibile creare un **virtual bus** per:

- **MSB** = Odouta(15 downto 8).
- **LSB** = Odouta(7 downto 0).

Iwea è stato generato come un vettore, sebbene composto da un solo bit (come **wea** in **Usa_RAM**); quando assegniamo il bit di controllo, **Vivado** si aspetta di ricevere un'assegnazione vettoriale (con **others**): anziché assegnare il bit '**1**', si assegna **others=>'1'**.

La *signal* **TestMUL** è usata per ragioni di debug. In questo modo in simulazione si può già visualizzare se il risultato prodotto dal circuito è corretto o meno. Per farlo, si utilizza l'operatore ad alto livello “*****” che moltiplica **MSB × LSB**.

Poiché la profondità è 1024, il numero di bit con cui andiamo a rappresentare gli indirizzi è 10, ovvero $\log_2 1024$.

Per questo motivo, l'indice *i* si converte in un STD_LOGIC_VECTOR composto da 10 bit.

Per semplicità si utilizza lo stesso *for* anche per l'assegnazione del dato **Idina**.

In questo modo, ad ogni colpo di clock viene generata una coppia di indirizzi e di conseguenza, dopo un'opportuna latenza (*pari a 1*), viene generato il risultato finale della moltiplicazione: il throughput è 1.

Codice

```
Test_MyRAM.vhd

D:\RACCOLTE\DOCUMENTI\niavdo vhdl\workspace\RamMUL\RamMUL.scs\sim_1\new\test_M\RAM.vhd

library IEEE;
use IEEE.STD.LOGIC_1164.ALL;
use IEEE.STD.LOGIC_UNSIGNED.ALL;
use IEEE.STD.TEXT.TEXTIO.ALL;

entity Test_MyRAM is
    -- Port ( );
    end Test_MyRAM;

architecture Behavioral of Test_MyRAM is
begin
component Usa_RAM is
    port( clike : IN STD.LOGIC; -- segnale di clock in ingresso
          ena : IN STD.LOGIC; -- segnale di abilitazione per controller che il banco di memoria sia accessibile o meno
          we : IN STD.LOGIC; -- segnale di write enable attivo alto (singolo bit)
          wdata : STD.LOGIC_VECTOR(15 DOWNTO 0); --segnale di write enable attivo alto (singolo bit)
          raddr : STD.LOGIC_VECTOR(5 DOWNTO 0); --indirizzo di ingresso
          rdina : IN STD.LOGIC_VECTOR(15 DOWNTO 0); --dato da leggere
          dout : OUT STD.LOGIC_VECTOR(15 DOWNTO 0); --dato da leggere
          mdata : OUT STD.LOGIC_VECTOR(15 DOWNTO 0)); --risultato finale
end component;

signal Iclock : STD.LOGIC := '0'; --inizializzato a 0 poiché il clock verrà utilizzato sui fronti di salita
signal Tena : STD.LOGIC;
signal Twaa : STD.LOGIC_VECTOR(15 DOWNTO 0); --singole bit
signal Taddr : STD.LOGIC_VECTOR(5 DOWNTO 0);
signal Tdina : STD.LOGIC_VECTOR(15 DOWNTO 0);
signal Odouta : STD.LOGIC_VECTOR(15 DOWNTO 0);
signal Cmnl : STD.LOGIC_VECTOR(15 DOWNTO 0);
signal TestMUL : STD.LOGIC_VECTOR(15 DOWNTO 0):=(others=>'0'); --per debug: moltiplica ad alto livello

constant Tolk : time:=1ns;
constant mdepth: integer:=1024;
constant naddr: integer:=10;
constant ndata: integer:=16;
begin
    C0: Usa_RAM port map(Iclock, Tena, Twaa, Taddr, Tdina, Odouta, Cmnl);

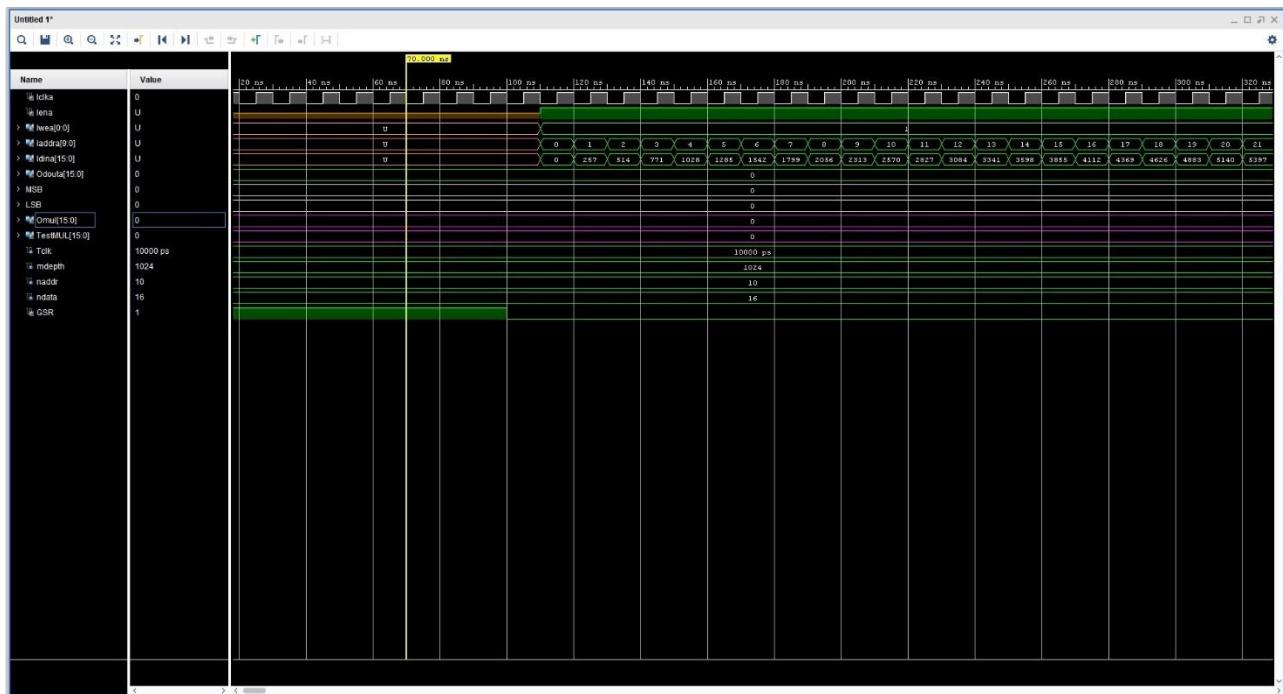
process
begin
    begin --tempistica per il segnale di clock
        wait for Tolk/2;
        Iclock<not Iclock;
    end process;
end;

AcMUL: process
begin
    --write data
    wait for Iclock+100ns;
    wait until falling_edge(Iclock);

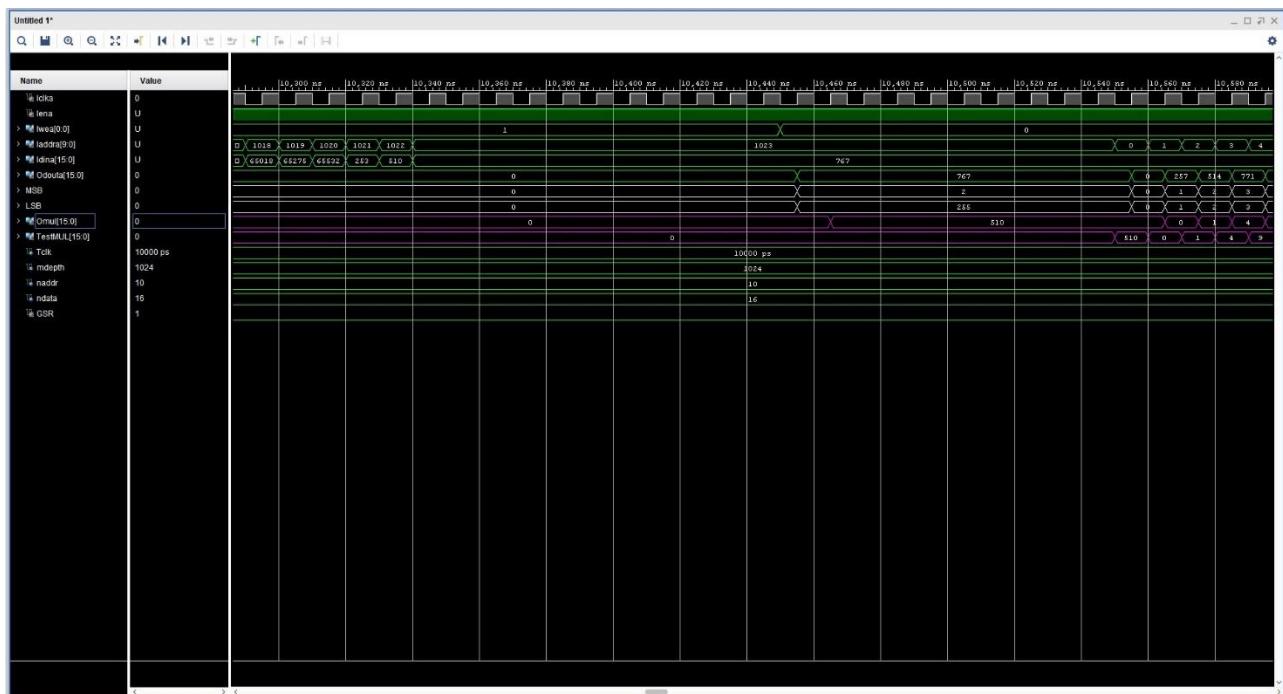
    Idata<='1';
    Deac<(others='1');
    for i in 0 to mdepth-1 loop
        Iaddr<=conv_std_logic_vector(i,naddr);
        Idina<=conv_std_logic_vector(1*256*i,ndata);
        wait for Tolk;
    end loop;
    -----
    wait for 10*Iclock; --arbitrary waiting
    --read data
    Deac<(others='0');
    wait for 10*Tclk;
    for i in 0 to mdepth-1 loop
        Iaddr<=conv_std_logic_vector(i,naddr);
        TestMUL<=Odouta(15 downto 8)*Odouta(7 downto 0); --per debug
        wait for Tolk;
    end loop;
end process;
end Behavioral;
```

SCREENSHOTS DEI TEST

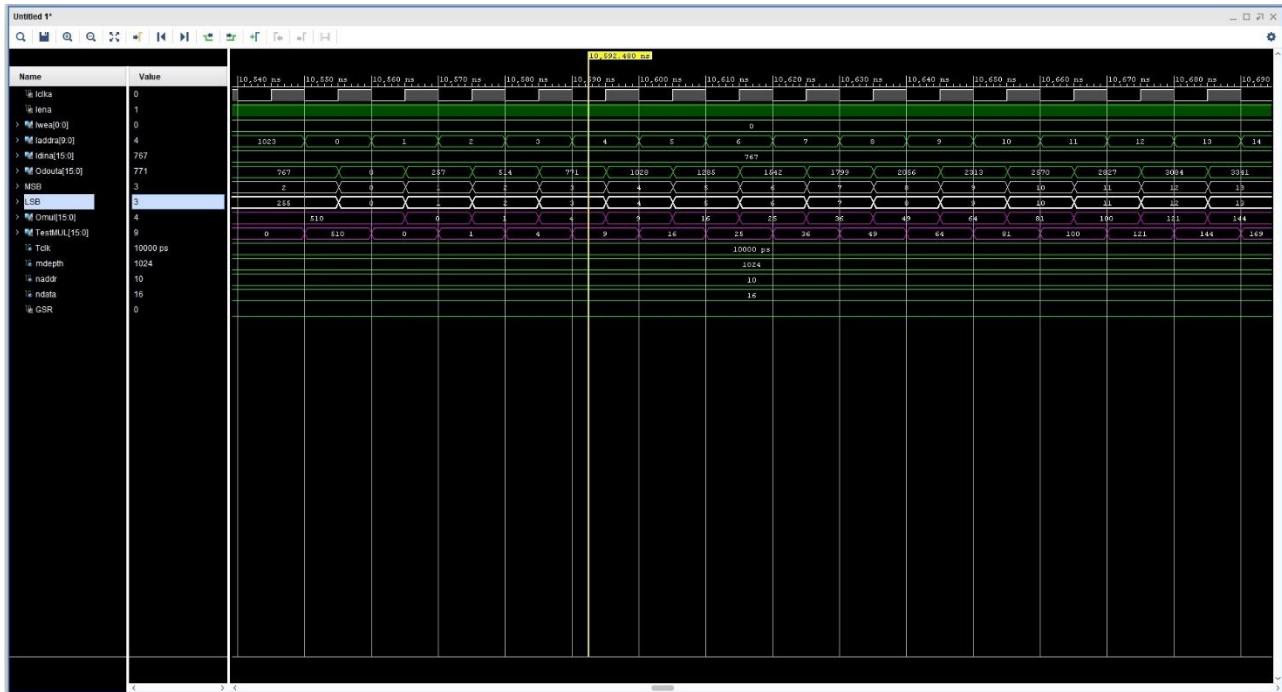
Behavioral SIMULATION Segnale GSR (Global RESET)



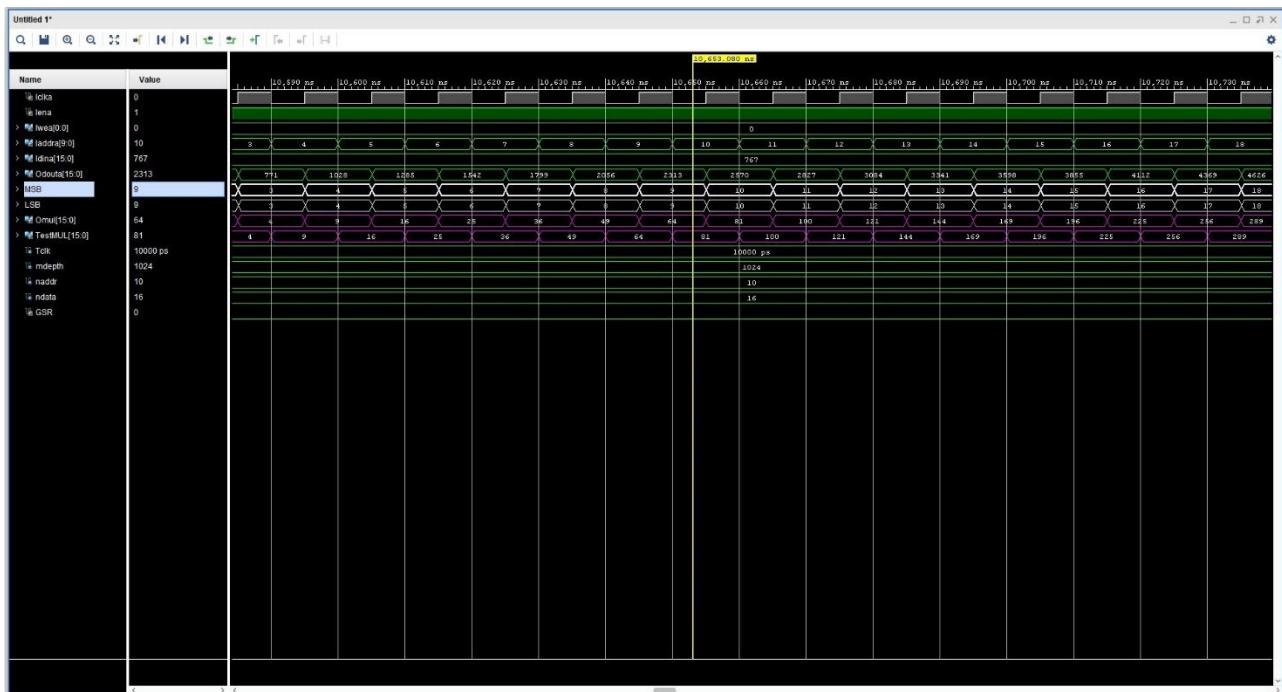
Passaggio da WRITE a READ all'indirizzo 1024



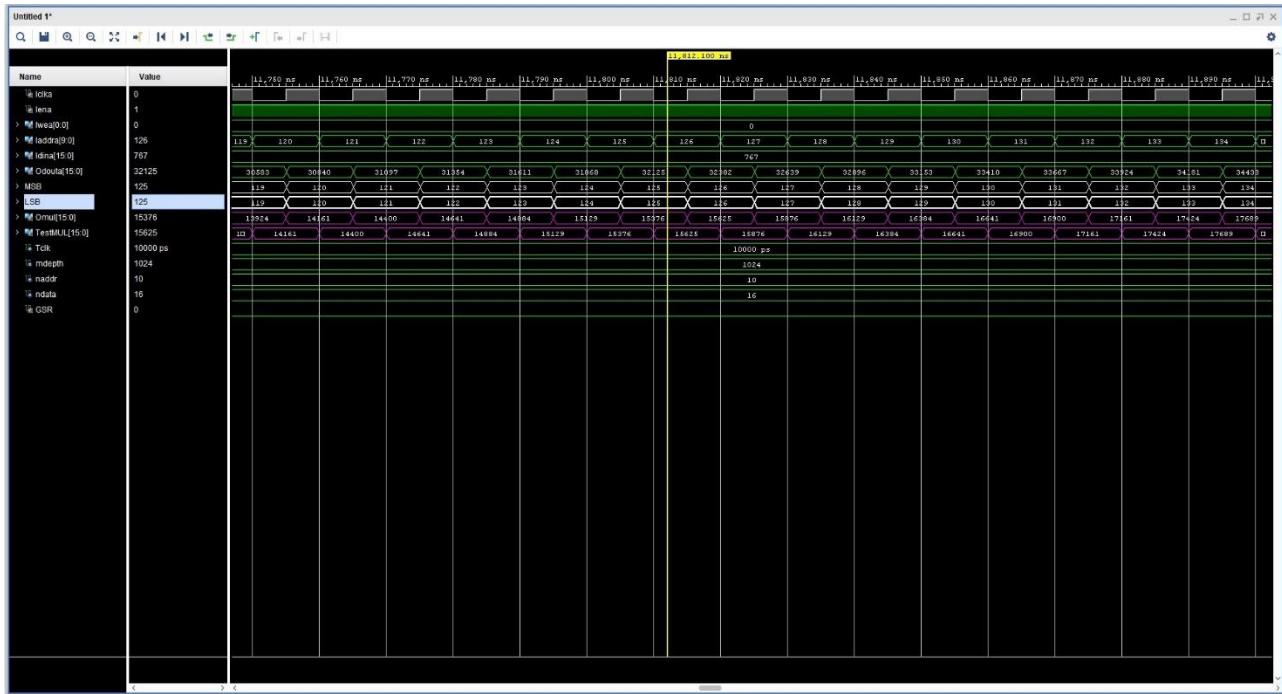
3x3



9x9

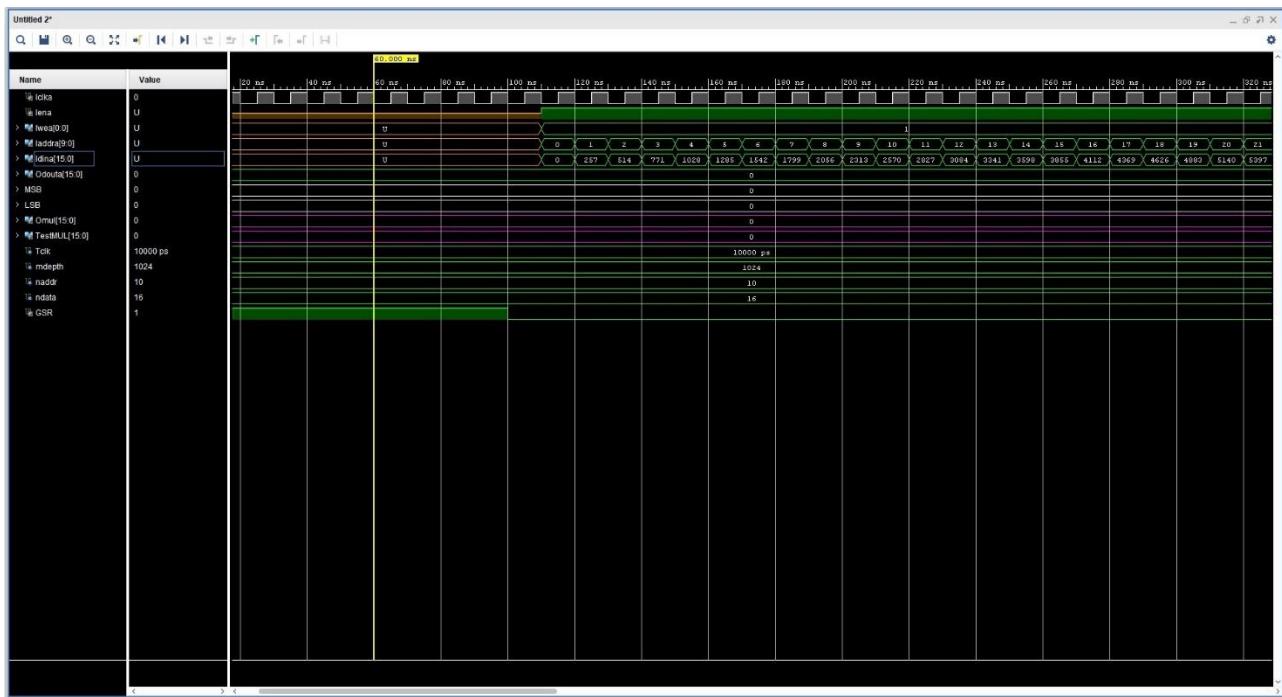


125x125

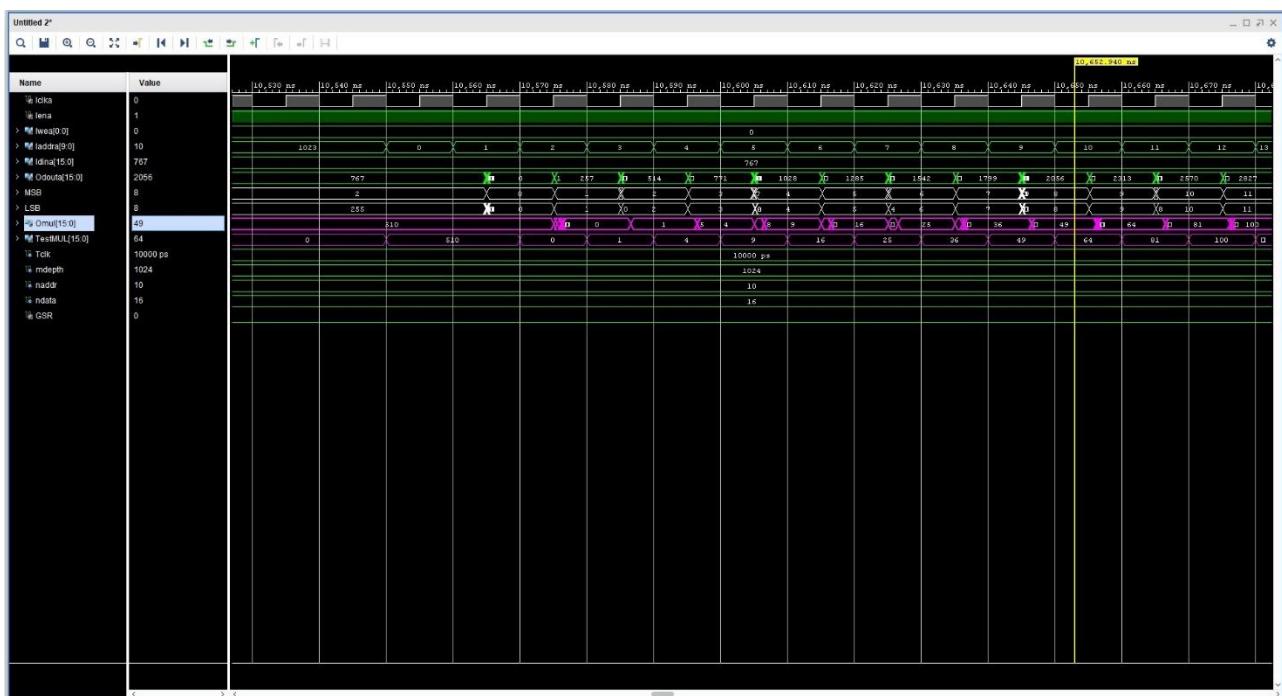


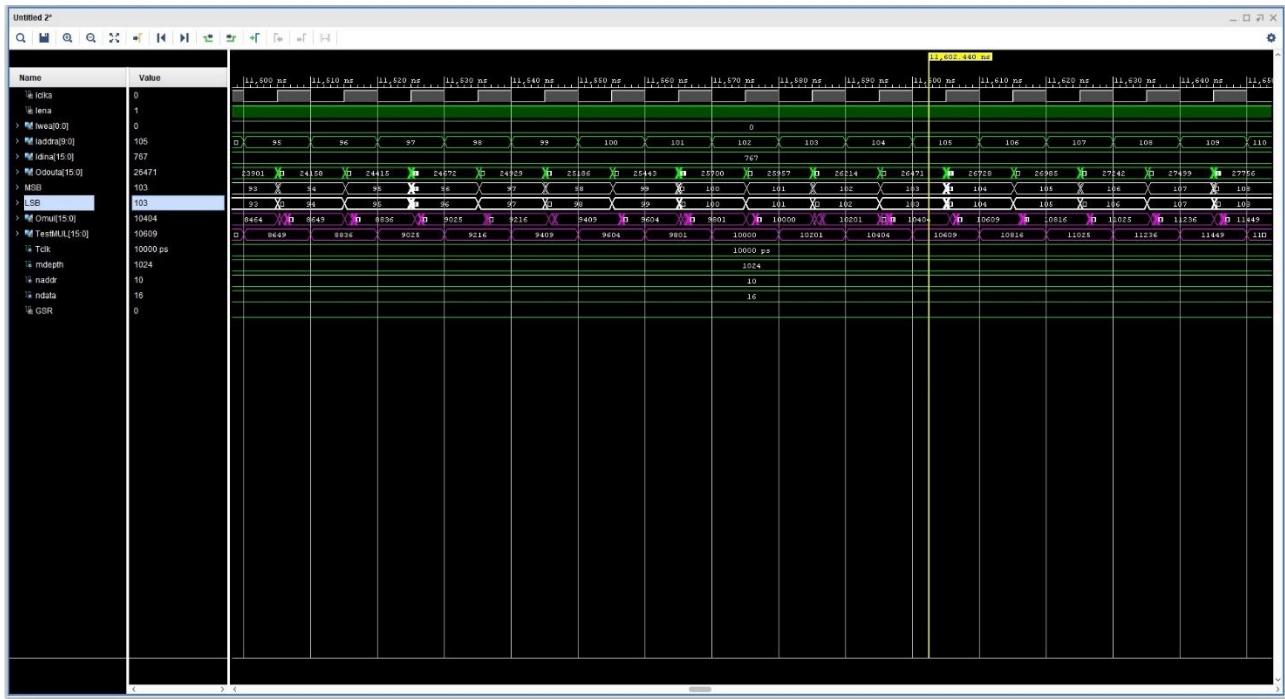
Post-Implementation Timing SIMULATION

Segnale GSR (Global RESET)



8x8



103x103

Appendice

In aggiunta a quanto detto fino ad ora, il progetto si può realizzare anche facendo uso di più registri, in modo tale da rendere tutta l'operazione di moltiplicazione in pipeline, dunque sensibile, in questo caso, al fronte di salita del clock.

Ciò è stato realizzato grazie ai component “**AdderTree**” e “**Registro**”.

AdderTree

In questo codice, si utilizzano registri sia per gli operandi iniziali, che per i risultati parziali che devono andare in ingresso ai CarrySave: in questo modo il circuito svolge la sua intera operazione in pipeline, sensibile ai fronti di salita del clock (clka).

Questo modulo sarà importato come *COMPONENT* nel file **Usa_RAM**.

Codice Adder Tree

Codice Usa_RAM

```
Usa_RAM.vhd
D:\RACCOLTE\DOCUMENTI\Invado-whd\workspace\RamMUL_appendice\RamMUL_appendice.srcc\sources_1\newUsa_RAM.vhd

1 library IEEE;
2 use IEEE.STD.LOGIC_1164.ALL;
3 use IEEE.STD.UNSIGNED.ALL;
4
5 entity Usa_RAM is
6   Port ( clk : IN STD_LOGIC; --segnale di clock in ingresso
7         ena : IN STD_LOGIC; --segnale di abilitazione per controllare che il banco di memoria sia accessibile o meno
8         wea : IN STD_LOGIC_VECTOR(0 DONT0 0); --singolo bit
9         wea : IN STD_LOGIC_VECTOR(0 DONT0 0);
10        addra : IN STD_LOGIC_VECTOR(15 DONT0 0);
11        addra : IN STD_LOGIC_VECTOR(15 DONT0 0);
12        dina : IN STD_LOGIC_VECTOR(15 DONT0 0);
13        dina : INSTD STD_LOGIC_VECTOR(15 DONT0 0);
14        douta : OUT STD_LOGIC_VECTOR(15 DONT0 0);
15 end Usa_RAM;
16
17 architecture Behavioral of Usa_RAM is
18
19 component blk_mem_gen_0 is
20   Port ( clk : IN STD_LOGIC;
21         ena : IN STD_LOGIC;
22         wea : IN STD_LOGIC_VECTOR(0 DONT0 0);
23         addra : IN STD_LOGIC_VECTOR(15 DONT0 0);
24         dina : IN STD_LOGIC_VECTOR(15 DONT0 0);
25         douta : OUT STD_LOGIC_VECTOR(15 DONT0 0));
26 end component;
27
28 component Register is
29   generic(nbitinteger);
30   Port ( D : in STD_LOGIC_VECTOR(nbit-1 downto 0); --ingresso del registro
31         clk : in STD_LOGIC; --segnale di clock
32         Q : out STD_LOGIC_VECTOR(nbit-1 downto 0)); --uscita del registro
33 end component;
34
35 component AdderTree is
36   Port ( A : in STD_LOGIC_VECTOR(15 downto 0);
37         B : in STD_LOGIC_VECTOR(15 downto 0);
38         C : in STD_LOGIC_VECTOR(15 downto 0);
39         D : in STD_LOGIC_VECTOR(15 downto 0);
40         E : in STD_LOGIC_VECTOR(15 downto 0);
41         F : in STD_LOGIC_VECTOR(15 downto 0);
42         G : in STD_LOGIC_VECTOR(15 downto 0);
43         H : in STD_LOGIC_VECTOR(15 downto 0);
44         clk : in STD_LOGIC;
45         Prod : out STD_LOGIC_VECTOR(15 downto 0));
46 end component;
47
48 signal w0,w1,w2,w3,w4,w5,w6,w7 : STD_LOGIC_VECTOR(7 DONT0 0);
49 signal F0,F1,F2,F3,F4,F5,F6,F7 : STD_LOGIC_VECTOR(7 DONT0 0);
50 signal R0,R1,R2,R3,R4,R5,R6,R7 : STD_LOGIC_VECTOR(15 DONT0 0);
51 signal R0,R1,R2,R3,R4,R5,R6,R7 : STD LOGIC_VECTOR(15 DONT0 0);
52 signal PP : STD LOGIC_VECTOR(15 DONT0 0);--(others=>"0");
53
54 --ARRY_SAVING
55 signal sp0,vr0,sp1,vr1 : STD_LOGIC_VECTOR(15 DONT0 0);--CS del primo livello
56 signal Reg0,Pr0,Reg1,Rv1: STD LOGIC_VECTOR(15 DONT0 0);--Registri del primo livello
57 signal sp2,vr2,sp3,vr3 : STD LOGIC_VECTOR(15 DONT0 0);--CS del secondo livello
58 signal Reg2,Pr2,Reg3,Rv3: STD LOGIC_VECTOR(15 DONT0 0);--Registri del secondo livello
59 signal sp4,vr4,sp5,vr5 : STD LOGIC_VECTOR(15 DONT0 0);--CS del terzo livello
60 signal Reg4,Pr4,Reg5,Rv5: STD LOGIC_VECTOR(15 DONT0 0);--Registri del terzo livello
61 signal sp6,vr6,sp7,vr7 : STD LOGIC_VECTOR(15 DONT0 0);--CS del quarto livello
62 signal Reg6,Pr6,Reg7,Rv7: STD LOGIC_VECTOR(15 DONT0 0);--Registri del quarto livello
63 signal Cm1: STD_LOGIC_VECTOR(15 DONT0 0);--RES FINALE
64
65 begin
66
67   MEM: blk_mem_gen_0 port map(clk, ena, wea, addra, dina, douta);
68
69   v0<= (others => douta(0));
70   PP0 <= v0 AND douta(15 DONT0 8);
71   R0<= ("00000000" &#xF0);
72
73
74   v1<= (others => douta(1));
75   PP1 <= v1 AND douta(15 DONT0 8);
76   R1<= ("00000000" &#xF1 "00");
77
78
79   v2<= (others => douta(2));
80   PP2 <= v2 AND douta(15 DONT0 8);
81   R2<= ("00000000" &#xF2 "00");
82
83   v3<= (others => douta(3));
84   PP3 <= v3 AND douta(15 DONT0 8);
85   R3<= ("00000000" &#xF3 "00");
86
87   v4<= (others => douta(4));
88   PP4 <= v4 AND douta(15 DONT0 8);
89   R4<= ("00000000" &#xF4 "0000");
90
91   v5<= (others => douta(5));
92   PP5 <= v5 AND douta(15 DONT0 8);
93   R5<= ("00000000" &#xF5 "0000");
94
95   v6<= (others => douta(6));
96   PP6 <= v6 AND douta(15 DONT0 8);
97   R6<= ("00000000" &#xF6 "000000");
98
99   v7<= (others => douta(7));
100  PP7 <= v7 AND douta(15 DONT0 8);
101  R7<= ("00000000" &#xF7 "000000");
102
103  --ADDER TREE
104  ALERO: AdderTree
105    port map(R0,R1,R2,R3,R4,R5,R6,R7,clk,Cm1);
106
107  --Registro del risultato finale
108  RegRes: Registro
109  generic map(14)
110  port map(Cm1,clk,R0);
111
112 end Behavioral;
113
```

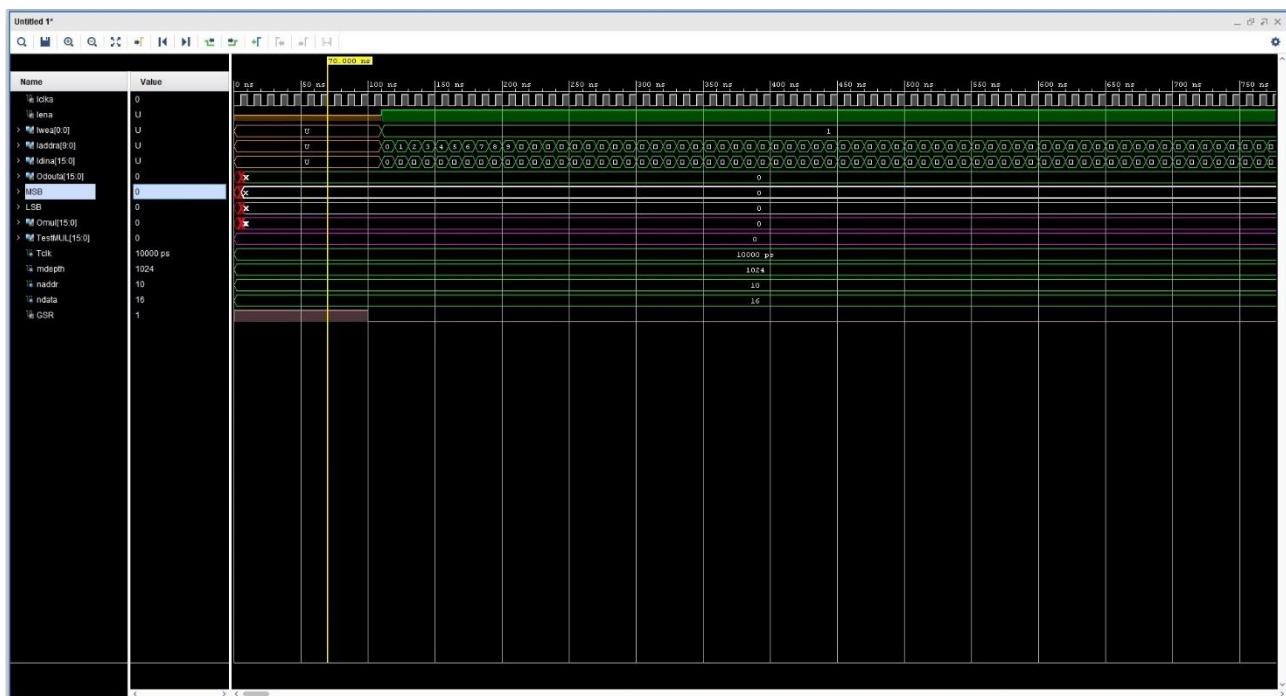
TestBench

A causa dell'uso di un elevato numero di registri, la **latenza** aumenta notevolmente.

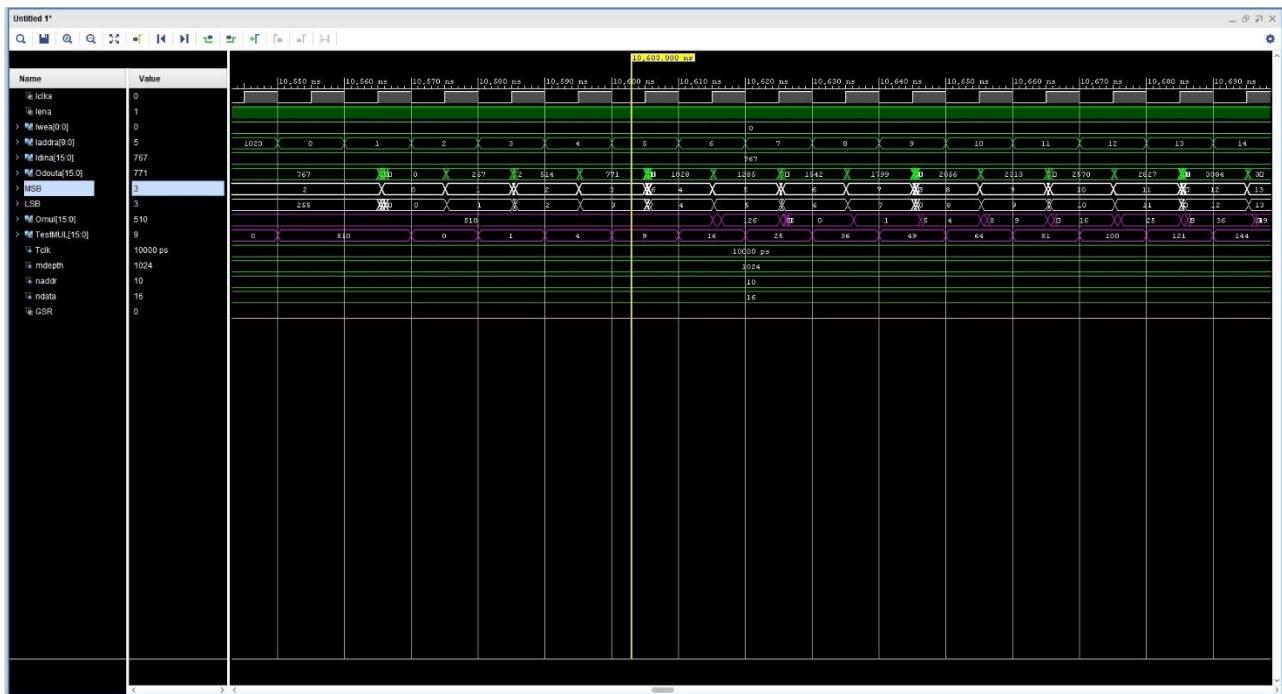
Infatti, il risultato corretto della moltiplicazione risulta in uscita dopo sei cicli di clock (nel progetto realizzato, invece, la latenza è pari ad un ciclo di clock).

Post-Implementation Timing SIMULATION

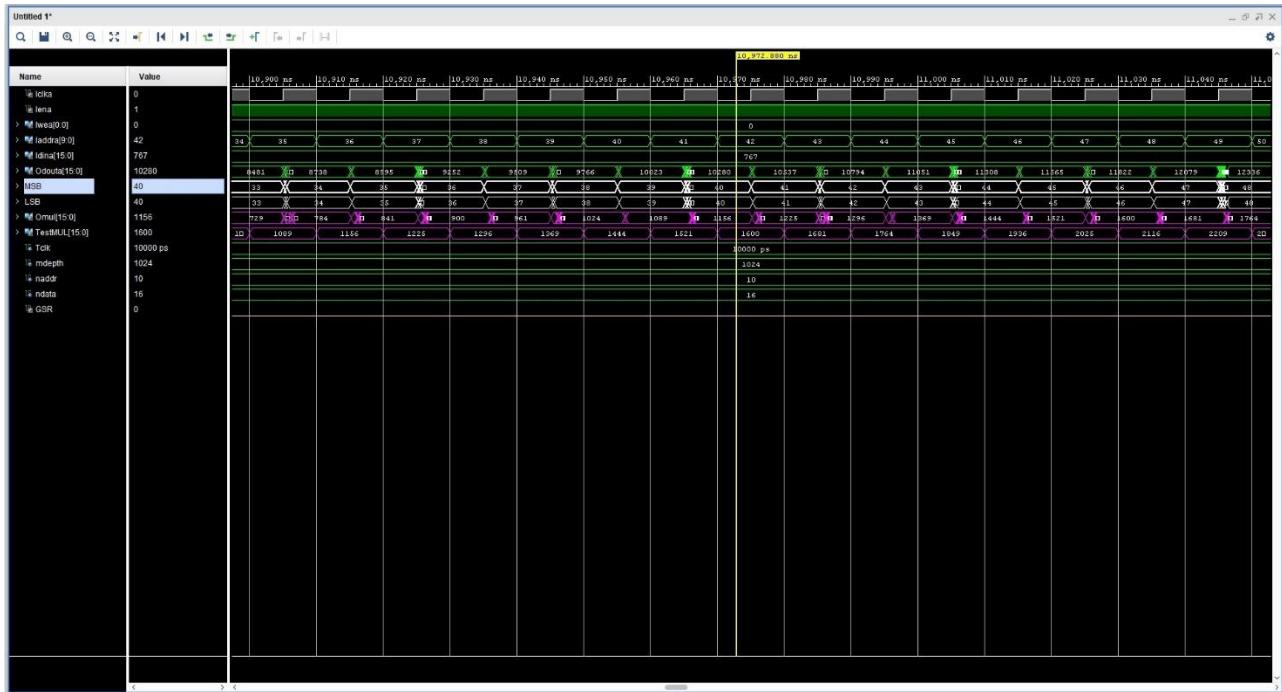
Segnale GSR (Global RESET)



3x3



40x40



Progetto realizzato da:
 Michele Purrone
 Antonino Vaccarella