MINISTERUL EDUCAȚIEI ȘI CERCETĂRII ȘTIINȚIFICE

UNIVERSITATEA TEHNICĂ
DIN CLUJ-NAPOCA

FACULTATEA DE AUTOMATICĂ ŞI CALCULATOARE
CATEDRA CALCULATOARE

Circuite de înmulțire pipeline pentru numere întregi

Hulea Andrei-Florin, grupa 30235

An academic :2020 – 2021

Cuprins:

1. Rezumat

Alaturi de adunare si scadere, inmultirea este una din cele mai importante functii artimetice. In multe programe stiintifice aceasta apare la fel de des, daca nu chiar mai frecvent decat adunarea si scaderea.

Scopul acestui proiect este de a prezenta functionarea si implementarea circuitele de inmultire pipeline in limbajul VHDL, cat si algoritmii pe care acestea sunt bazate si avantajele aplicarii unui sistem pipeline asupra circuitelor de inmultire.

2. Introducere

In acest proiect vor fi prezentate diverse metode de inmultire, fiecare cu propriile avantaje si dezavantaje, cat si algoritmii si circuitele necesare pentru implementarea acestora. Caracteristica comuna a acestor circuite este ca toate se folosesc de sumatoare pentru a ajunge la rezultatul final.

Metodele posibile pentru inmultire care vor fi prezentate sunt: inmultirea prin deplasare si adunare, tehnica Booth, inmultirea intr-o baza superioara, inmultirea matriceala si arborele Wallace. Dintre acestea se va prezenta implementarea cu tehnica pipeline a inmultirii maticeale si a arborelui Wallace.

Inmultirea prin deplasare si adunare presupune adunarea deinmultitului cu el insusi pana se ajunge la rezultatul dorit, numarul adunarilor fiind dat de inmultitor. Astfel, se vor selecta cifrele inmultitorului pe rand, una cate una, deinmultitul fiind inmultit cu fiecare dintre acestea, rezultatul fiind suma acestora fiecare shiftate cu cate o pozitie.

Tehnica Booth este folosita pentru numere cu semn, testand cate doi biti ai inmultitorului la fiecare pas, astfle eliminand necesitatea conversiei operanzilor la numere pozitive. De asemenea permite operatii cu numere cu semn si poate reduce numarul operatiilor necesare.

Arborele Wallace porneste de la ideea ca pentru inmultirea a doua numere de cate n biti trebuie adunate n produse partiale. Avantajul arborelui lui Wallace este ca inmultirea se executa in O(log n) fata de O(n), deoarece acesta utilizeaza un arbore ce poate fi implementat utilizand sumatoare elementare. In inmultitoarele din ziua de azi cel mai frecvent este utilizata o combinatie dintre tehnica Booth si arborele Wallace. De asemenea tehnica pipeline este foarte avantajoasa pentru circuite de inmultire combinationala ce folosesc inmultirea matriciala si arborele lui Wallace, aceasta permitand cresterea vitezei.

Pipeline-urile sunt utilizate pe scara larga pentru a imbunatati performanta circuitelor digitale deoarece acestea ofera o modalitate simpla de implementare a palalelismului pentru operatii secventiale. Cu cat se introduc mai multe nivele pipeline, fiecare etapa devine mai scurta si in mod ideal prezinta o intarziere din ce in ce mai mica. Deci, circuitul rezultat va prezenta o latenta mai mare dar si performanta mai mare atunci cand pipeline-ul este pe deplin utilizat.

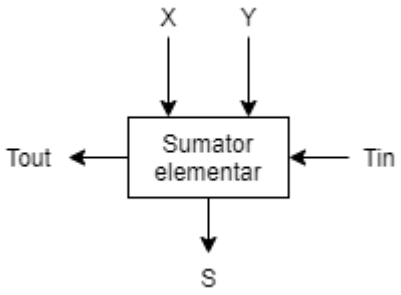
3. Fundamentare teoretica

❖ Circuite

Circuitele pe care le vom folosi la implementarea circuitelor de inmultire sunt urmtoarele:

- **Sumator elementar**

Un sumator elementar are trei intrari de cate un bit, mai exact bitii X si Y care trebuie adunati si Tin (Ti), transportul de la bitul din pozitia mai putin semnificativa. Acesta are doua iesiri: bitul de suma S si bitul de transport la pozitia mai semnificativa Tout(Ti+1). Expresiile iesirilor sunt urmatoarele: $S = X \oplus Y \oplus Tin$, $Tout = XY+(X+Y)Tin$. Majoritatea sumatoarelor mai complexe sunt bazate tot pe acest principiu.



x_i	yi	Ti		T_{i+1}	S_i
0	0	0		0	0
0	0	1		0	1
0	1	0		0	1
0	1	1		1	0
1	0	0		1	0
1	1	0		1	0
1	1	1		1	1

Fig 1. Schema bloc sumator elementar si tabel de valori

- **Sumator cu propagarea succesiva a transportului**

Algoritmul de adunare succesiva a transportului este urmatorul:

$$\begin{array}{r} x_3\ x_2\ x_1\ x_0 + \\ \underline{y_3\ y_2\ y_1\ y_0} \\ S_4\ S_3\ S_2\ S_1\ S_0 \end{array}$$

Acesta poate fi implementat prin conectarea mai multor sumatoare elementare in serie, astfel incat fiecarui bit sa ii fie asignat cate un sumator elementar. De asemenea este un sumator paralel si transportul trebuie sa se propage succesiv prin toate sumatoarele inainte de a se cunoaste rezultatul final. Deci, se poate observa ca avantajele acestuia sunt simplitatea si costul redus, dezavantajul fiind viteza redusa. Schema unui sumator cu propagarea succesiva a transportului pe 4 biti este urmatoarea:

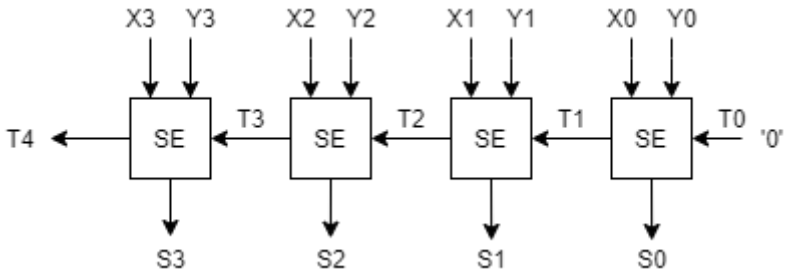


Fig 2. Schema bloc sumator cu propagarea transportului

- **Sumator cu salvarea transportului**

Acest sumator este folosit atunci cand trebuie adunate mai mult de doua numere deoarece reduce timpul de propagare al semnalelor de transport, fiind format din o colectie de sumatoare elementare ce functioneaza independent unul fata de celalalt (semnalele de transport nu sunt transmise de la un sumator la celalalt). Astfel, acesta are ca intrari trei numere de n biti si ca iesire un cuvant suma de n biti si un cuvant de transport de n biti.

De asemenea, deoarece transportul nu este propagat intre sumatoare, pentru a obtine un rezultat final, transportul si suma vor trebui adunate cu un sumator obisnuit cu propagarea transportului.

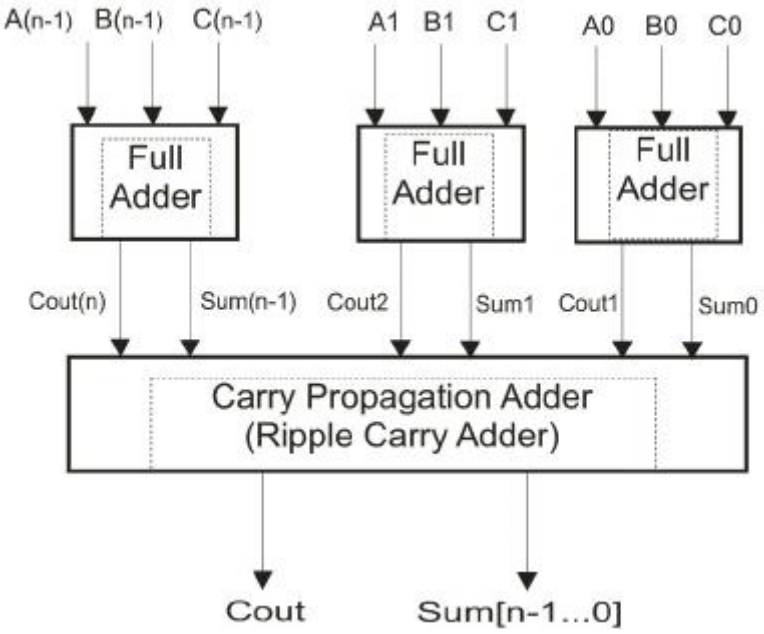


Fig 3. Schema bloc sumator cu salvarea transportului

- **Registru**

Registrul va fi folosit pentru stocarea elementelor din etajele pipeline. Datorita intrarii de ceas, in registru se va scrie doar pe un singur etaj la fiecare ciclu. Intrarea de date D reprezinta cuvantul format din toti bitii pe care vrem sa ii stocam intr-un etaj. Semnalul R prentru reset si CE pentru clock enable au logica pozitiva.

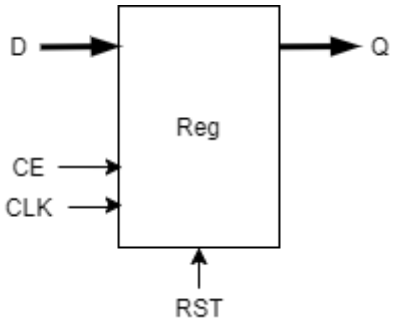
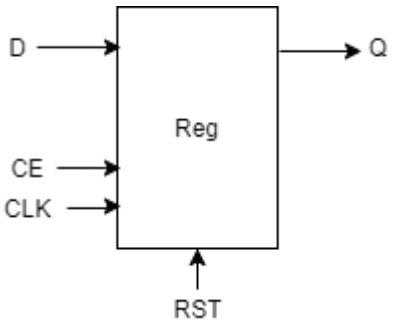


Fig 4,5. Schema bloc registru si bistabil

- **Bistabil**

Un registru este format din mai multe bistabile. Un bistabil reprezinta o celula de memorie de 1 bit ce functioneaza in mod sincron.



❖ **<u>Metode de inmultire</u>**

- **Tehnica Booth**

Pentru cresterea vitezei inmultirii, Booth a descoperit o tehnica prin care se reduce numarul etapelor de adunare si se elimina conversia operanzilor la forma pozitiva. Astfel, daca se poate efectua atat adunare, cat si scadere, exista mai multe posibilitati de a calcula un produs. De exemplu, un sir de cifre de 0 din inmultitor nu are nevoie de adunare, ci numai de deplasare sau un sir de cifre de 1 poate fi tratat ca un numar cu valoarea L-R(L= ponderea cifrei 0 dinaintea cifrei 1 celei mai din stanga, R= ponderea cifrei 1 celei mai din dreapta). Deci tehnica Booth are rolul de a elimina necesitatea conversiei operanzilor la forma pozitiva si poate reduce numarul operatiilor necesare pentru realizarea operatiei, motiv pentru care este folosita impreuna cu arborele Wallace in majoritatea circuitelor de inmultire folosite in ziua de azi.

La inmultirea prin tehnica Booth se considera cate doi biti adiacenti ai inmultitorului pentru a determina operatia care trebuie efectuata, conform tabelului urmator(y_i reprezinta bitul curent si y_{i-1} reprezinta bitultestat in pasul precedent):

y_i	y_{i-1}	$y_{i-1} - y_i$	**Operatii**
0	0	0	Deplasare la dreapta
0	1	1	Adunare deinmultit, deplasare la dreapta
1	0	-1	Scadere deinmultit, deplasare la dreapta
1	1	0	Deplasare la dreapta

- **Inmultirea matriceala**

O alta metoda de crestere a vitezei este reprezentata de utilizarea circuitelor combinationale de inmultire deoarece nu au nevoie de semnal ce ceas, care comparativ cu circuitele secventiale de inmultire care sunt dependente de semnalul de ceas pentru realizarea calculelor, contin o logica suplimentara pentru calculul produsului intr-un singur pas. Aceste circuite sunt formate din matrici de elemente combinationale simple, fiecare din acestea implementand o operatie de adunare si deplasare pentru un bit sau un numar redus de biti.

Produsul P al doua numere X,Y cu bitii $x_{n-1}... x_1 x_0$, respectiv $y_{n-1}... y_1 y_0$ va fi de forma $P = (\sum_{i=0}^{n-1} 2^i * x_i) * (\sum_{j=0}^{n-1} 2^j * y_j) = \sum_{i=0}^{n-1} 2^i * (\sum_{j=0}^{n-1} x_i * y_j * 2^j)$.

Prentru fiecare termen produs de 1 bit $x_i * y_j$ se vor folosi porti SI. Astfel o matrice de n x n porti SI poate calcula simultan toti termenii $x_i * y_j$, dupa care acestia vor fi adunati cu o matrice de n(n-1) sumatoare elementare.

Ex: Inmultirea a doua numere de cate 4 biti,
$X = x_3x_2x_1x_0$, $Y = y_3y_2y_1y_0$

				x_3	x_2	x_1	x_0 *
				y_3	y_2	y_1	y_0
0	0	0	0	x_3*y_0	x_2*y_0	x_1*y_0	x_0*y_0
0	0	0	x_3*y_1	x_2*y_1	x_1*y_1	x_0*y_1	0
0	0	x_3*y_2	x_2*y_2	x_1*y_2	x_0*y_2	0	0
0	x_3*y_3	x_2*y_3	x_1*y_3	x_0*y_3	0	0	0
P7	P6	P5	P4	P3	P2	P1	P0

$P0 = x_0*y_0$
$P1 = x_1*y_0 + x_0*y_1$
$P2 = x_2*y_0 + x_1*y_1 + x_0*y_2$
$P3 = x_3*y_0 + x_2*y_1 + x_1*y_2 + x_0*y_3$
$P4 = x_3*y_1 + x_2*y_2 + x_1*y_3$
$P5 = x_3*y_2 + x_2*y_3$
$P6 = x_3*y_3$

Circuitele de inmultire matriceala pot fi modificate in mod simplu pentru a utiliza tehnica pipeline, prin adaugarea de registre.

Un circuit de inmultire matriceala care foloseste tehnica pipeline cu n etaje poate suprapune calculul a n produse, deci acesta va putea genera un nou rezultat la fiecare ciclu de ceas. Totodata, tehnica pipeline vine si cu un dezavantaj, acesta fiind viteza redusa de propagare a transportului din fiecare etaj si un numar ce n^2 celule pentru calculul produsului partial.

- **Arborele Wallace**

Cel mai simplu mod de a inmulti doua numere de cate n biti este adunarea a n produse partiale. Circuitele de inmultire prezentate pana acum executa inmultirea intr-un timp O(n), timp care poate fi redus la O(log n) prin utilizarea unui arbore, acesta avand rolul de a reduce numarul produselor partiale de la n la n/2. Produsele urmeaza sa fie adunate apoi trei cate trei cu ajutorul sumatoarelor cu salvarea transportului, obtinundu-se suma finala dupa doar $\log_2 n$ etape si trecerea ultimei sume si transport printr-un sumator cu propagarea transportului.

A fost demonstrat de catre Wallace ca produsele partiale pot fi adunate intr-un mod rapid si economic utilizand mai multe nivele de sumatoare cu salvarea transportului(SST), fiecare sumator adunand produsele partiale grupate cate 3 si fiind organizate pe nivele intr-o structura numita arbore Wallace. Procesul continua pana raman numai doua numere de adunat, pentru care vom folosi un sumator cu propagarea transportului. De asemenea, fiecare nivel reduce numarul termenilor care trebuie adunati cu un factor de 1,5.

Metoda arborelui Wallace poate fi combinata cu tehnica Booth pentru cresterea vitezei. Tehnica Booth va fi utilizata pentru generarea produselor partiale si arborele Wallace va fi folosit pentru adunarea acestor produse partiale. De asemenea, se pot adauga si registre la fiecare etaj pentru a folosi tehnica pipeline si a creste productivitatea, structura de arbore fiind perfecta pentru aceeasta.

4. Proiectare si implementare

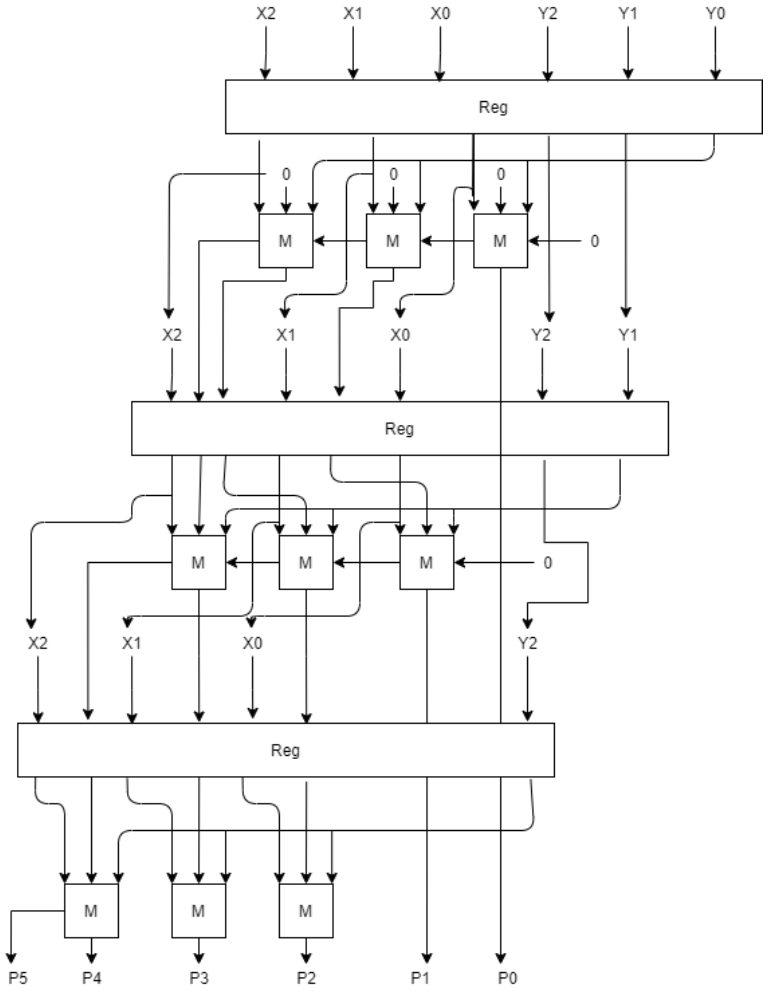
Operatiile aritmetice pot fi implementate printr-un sistem pipeline. Avantajele acestei tehnici sunt paralelismul si cresterea productivitatii. Fiecare etapa e executata de un etaj al sistemului pipeline.

Mai jos sunt prezentate inmultirea matriceala si arborele Wallace, ambele fiind implementate folosind tehnica pipeline. Aceste doua metode sunt cele mai potrivite pentru a folosi tehnica pipeline deoarece structura lor pe nivele permite o usoara introducere de registre, care au rolul de a creste productivitatea.

I. Inmultirea matriceala cu tehnica pipeline

Pentru realizarea acestei metode am implementat componentele necesare descrise in schema bloc de mai jos: celula M de 1 bit de inmultire si adunare si registrul. Vom avea nevoie de n^2 (n pentru fiecare etaj) celule M si de n registre (unul pentru fiecare etaj).

Implementarea circuitului de inmultire matriceala pipeline si a registrului au fost realizate in mod generic, pentru a putea fi folosit pentru numere cu dimensiunea variabila. Astfel pentru verificarea rezultatelor inmultitorului, numarul de biti al inmultirotului trebuie schimbat doar in test bench, sau direct in componenta pentru a putea fi testat pe placa FPGA.



Celulele M sunt sumatoare elementare de 1 bit.

Registrii au fost implementati cu parametru generic. Acestia salveaza valoarea ce o au pe intrare doar daca semnalul EN (Enable) este activ pe ’1’, semnal pe care il primesc de la unitatea principala a inmultitorului.

In entitatea principala a inmultitorului pipeline, acesta va fi implementat cu parametru generic. Astfel vom defini un semnal AUX care va avea rolul de semnal ce intra in registre. In acelasi fel definim si semnalul REGISTRU ce reprezinta iesirea din registre. Cu ajutorul parametrului generate vom genera numarul necesar de registre, ce poate fi usor dedus observand schema bloc de mai sus, asadar tot timpul vom avea nevoie de n registre, unde n reprezinta numarul de biti al operanzilor de pe intrarea inmultitorului. De asemenea pentru a putea respecta regula si a tine bitul curent din Y in cea mai din dreapta parte a registrului, acesta trebuie inversat. Astfel in generate-ul de la celulele M, IN_REGISTRU(i)(2*n-1) va reprezenta tot timpul acel bit din Y.

Pentru stocarea in registre am considerat urmatoarea ordine: Bitii lui X, urmati debitii ramasi de la Y, urmati de bitul de transport, bitii de la sumatoare si bitii care raman sa fie transportati in continuare, considerand ca bitii rezervati pentru Y se micesc cu o pozitie la fiecare registru, dupa cum se observa si din schema bloc.

Ultimul pas este generarea sumatoarelor de 1 bit si a produselor finale care respecta regulile enuntate anterior.

II. Arborele Wallace cu tehnica pipeline

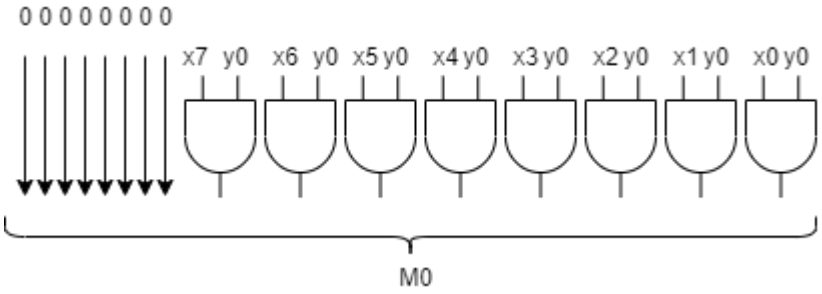
Pentru realizarea acestei metode am realizat componentele neceseare care apar si in schema bloc de mai jos: sumatoare cu salvarea transportului si un sumator cu propagarea transportului(ambele formate din sumatoare elementare) si registre pentru salvarea rezultatelor partiale de pe etaje la fiecare ciclu de ceas.

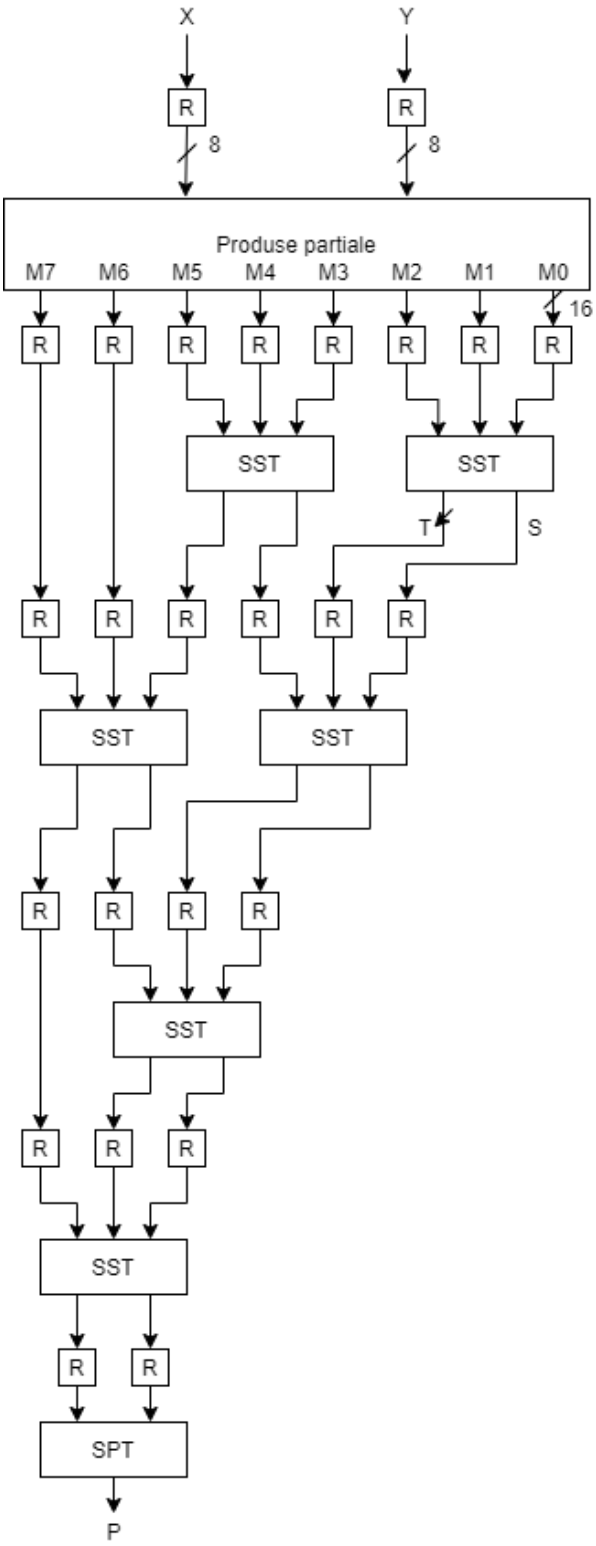
Spre diferenta de inmultirea matriceala pipeline, aceasta metoda mai greu de implementat in mod generic deoarece numarul de sumatoare cu salvarea transportului si numarul registrelor variaza la schimbarea numarului de biti, acestea nefiind egale.

Dupa definirea entitatii si a semnalelor necesare primul pas este stocarea in operanzilor X si Y in cate un registru. Registrul a fost implementat cu parametru generic, astfel cu acceasi componenta putem stoca vectori de biti de lungimi diferite. Dupa stocarea acestora, cu valorile din registrii respectivi vom crea produsele partiale. Primul produs partial M0 este format din n biti de 0 dupa care sunt concatenate rezultatele unui AND aplicat pe bitii din X cu bitul 0 din Y. Aceasta regula se respecta si pentru restul produselor partiale, doar ca indexul bitului din Y este incrementat si rezultatul este shiftat la stanga cu o pozitie.

Urmatorul pas este generarea sumatoarelor si salvarea transportului si a registrelor la pozitiile potrivite.

In figura de mai jos este reprezentat un inmultitor Wallace pe 8 biti urmat de o celula de produs partial. Produsele partiale sunt shiftate la stanga cu cate un bit unul fata de celalalt.

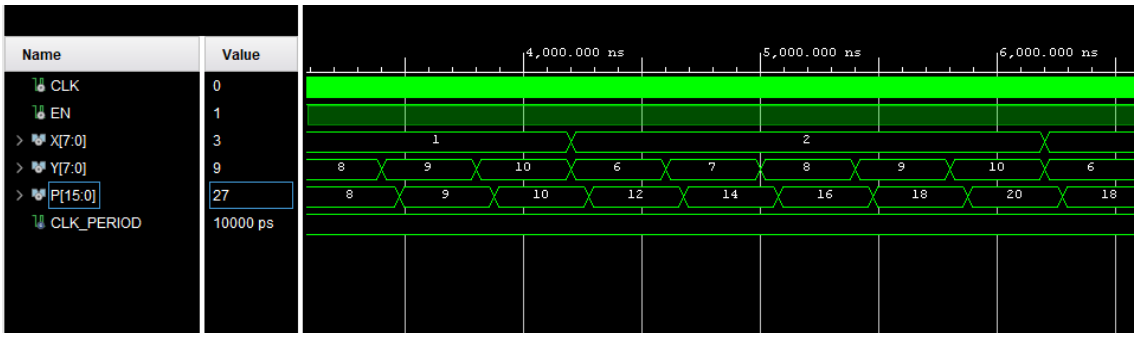


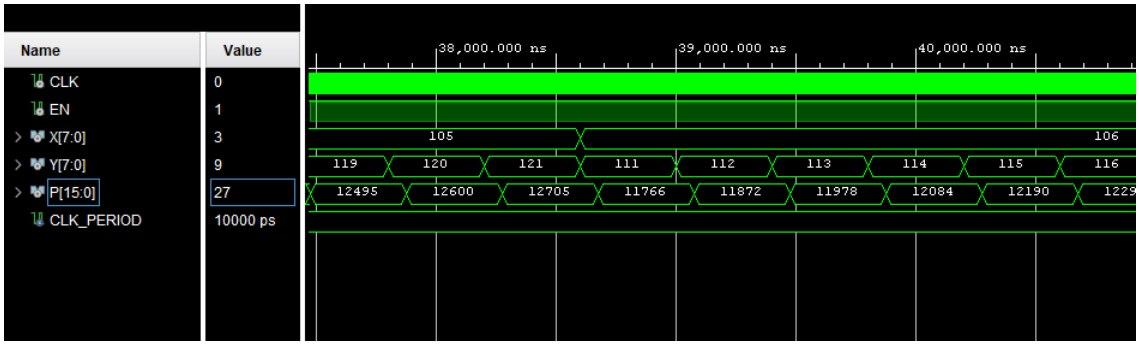


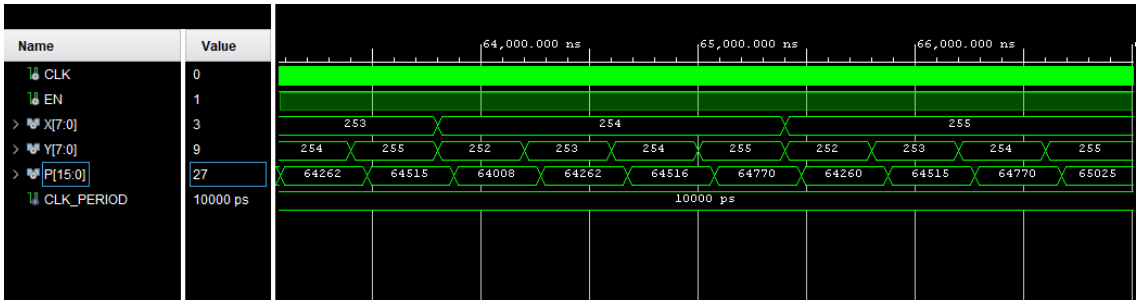
5. Rezultate experimentale

Pentru verificarea corectitudinii rezultatelor celor 2 inmultitoare am creat cate un banc de test pentru fiecare dintre ele, unde am testat valori asemanatoare pentru ambele. La inceput am luat cateva valori mici incepand de la 0, dupa care cateva valori medii in jurul lui 100 si in final valori in jurul lui 255 care sa atinga maximul de 8 biti.

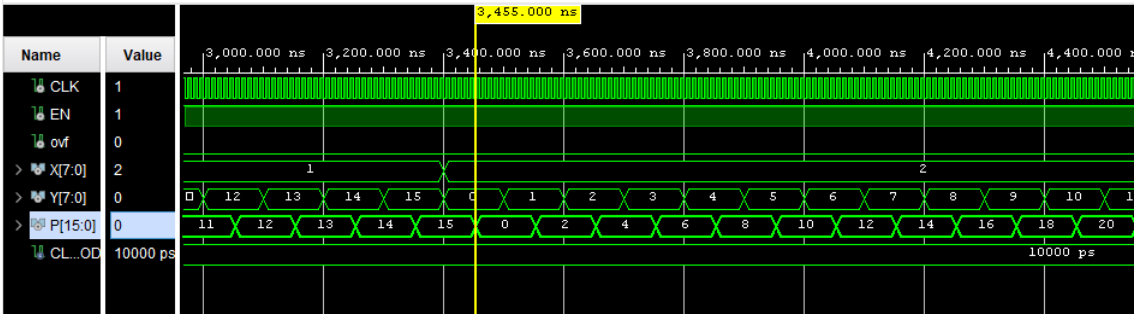
Rezultatele obtinute la simularea inmultirii matriciale pipeline:

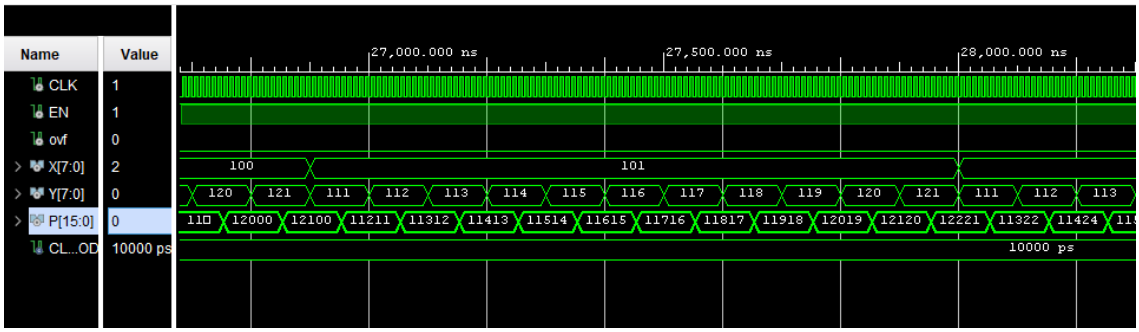


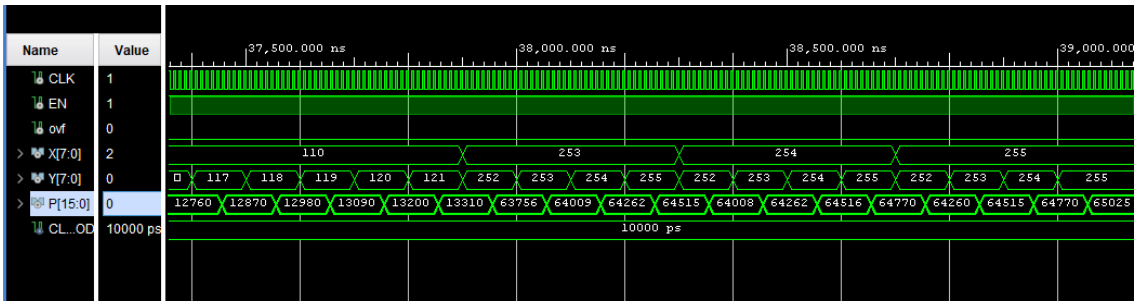




Rezultate obtinute la simularea imnultirii cu arborele Wallace pipeline:







6. Concluzii

In urma acestui proiect am reusit sa inteleg mult mai amanuntit modul de functionare a circuitelor de adunare, inmultire si de inmultite pipeline, cat si ce importanta are cresterea productivitatii si vitezei calculelor mai ales in ziua de azi cand viteza si performanta sunt mandatorii.

7. Bibliografie

- Z. F. Baruch, Structura sistemelor de calcul – Înmulţire
- https://computationstructures.org/notes/tradeoffs/notes.html
- http://www.csitsun.pub.ro/courses/cn1labs/Laborator4rom.pdf?fbclid=IwAR2yEBxM6cL8S6k2OQF7THT7xS7WJCEeiuMj0UBpKbcAOqMoa5N8g01mPmU
- https://en.wikipedia.org/wiki/Binary_multiplier
- Design of Very Deep Pipelined Multipliers for FPGAs - Alex Panato, Sandro Silva, Flávio Wagner, Marcelo Johann, Ricardo Reis, Sergio Bampi Universidade Federal do Rio Grande do Sul - Instituto de Informática

Anexa A
-Cod componente

- Registru

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

entity Reg is
generic (n : INTEGER);
  Port (

  CLK : in std_logic;
  EN  : in std_logic;
  D   : in std_logic_vector(n-1 downto 0);
  Q : out std_logic_vector(n-1 downto 0)

  );
end Reg;

architecture Behavioral of Reg is
begin

    process(CLK)
    begin
        if rising_edge(CLK) then
             if EN = '1' then
                Q <= D;
            end if;
        end if;
    end process;

end Behavioral;
```

- M

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


entity M is
  Port (

  X  : in std_logic;
  Y  : in std_logic;
  A  : in std_logic;
  Tin: in std_logic;

  S   : out std_logic;
  Tout: out std_logic
  );
```

```
end M;

architecture Behavioral of M is

begin

    S    <= A xor (X and Y) xor Tin;

    Tout <= (A and (X and Y)) or ((A or (X and Y)) and Tin);

end Behavioral;
```

- Adder

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

entity Adder is
 Port (
      A : in STD_LOGIC;
      B : in STD_LOGIC;
      Tin : in STD_LOGIC;
      S : out STD_LOGIC;
      Tout : out STD_LOGIC);
end Adder;

architecture Behavioral of Adder is

begin

 S <= A XOR B XOR Tin ;
 Tout <= (A AND B) OR (A AND Tin) OR (B AND Tin) ;

end Behavioral;
```

- SST

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

entity SST is
Port (
    A : in STD_LOGIC_VECTOR (15 downto 0);
    B : in STD_LOGIC_VECTOR (15 downto 0);
    C : in STD_LOGIC_VECTOR (15 downto 0);
    S : OUT STD_LOGIC_VECTOR (15 downto 0);
    T : OUT STD_LOGIC_VECTOR (15 downto 0));
end SST;
```

```
•
•  architecture Behavioral of SST is
•
•  component Adder is
•  Port (
•       A : in STD_LOGIC;
•       B : in STD_LOGIC;
•       Tin : in STD_LOGIC;
•       S : out STD_LOGIC;
•       Tout : out STD_LOGIC);
•  end component Adder;
•
•  signal X,Y,T_aux: STD_LOGIC_VECTOR(15 downto 0);
•  signal C1,C2,C3,c4,c5,c6,c7: STD_LOGIC;
•
•
•  begin
•
•  ad1:  Adder port map( A(0)  ,B(0)  ,C(0)  ,S(0)  ,T_aux(0));
•  ad2:  Adder PORT MAP( A(1)  ,B(1)  ,C(1)  ,S(1)  ,T_aux(1));
•  ad3:  Adder PORT MAP( A(2)  ,B(2)  ,C(2)  ,S(2)  ,T_aux(2));
•  ad4:  Adder PORT MAP( A(3)  ,B(3)  ,C(3)  ,S(3)  ,T_aux(3));
•  ad5:  Adder PORT MAP( A(4)  ,B(4)  ,C(4)  ,S(4)  ,T_aux(4));
•  ad6:  Adder PORT MAP( A(5)  ,B(5)  ,C(5)  ,S(5)  ,T_aux(5));
•  ad7:  Adder PORT MAP( A(6)  ,B(6)  ,C(6)  ,S(6)  ,T_aux(6));
•  ad8:  Adder PORT MAP( A(7)  ,B(7)  ,C(7)  ,S(7)  ,T_aux(7));
•  ad9:  Adder PORT MAP( A(8)  ,B(8)  ,C(8)  ,S(8)  ,T_aux(8));
•  ad10: Adder PORT MAP( A(9)  ,B(9)  ,C(9)  ,S(9)  ,T_aux(9));
•  ad11: Adder PORT MAP( A(10) ,B(10) ,C(10) ,S(10) ,T_aux(10));
•  ad12: Adder PORT MAP( A(11) ,B(11) ,C(11) ,S(11) ,T_aux(11));
•  ad13: Adder PORT MAP( A(12) ,B(12) ,C(12) ,S(12) ,T_aux(12));
•  ad14: Adder PORT MAP( A(13) ,B(13) ,C(13) ,S(13) ,T_aux(13));
•  ad15: Adder PORT MAP( A(14) ,B(14) ,C(14) ,S(14) ,T_aux(14));
•  ad16: Adder PORT MAP( A(15) ,B(15) ,C(15) ,S(15) ,T_aux(15));
•
•  T <= T_aux(14 downto 0) & '0'; --shift
•
•  end Behavioral;
```

- SPT

```
•  library IEEE;
•  use IEEE.STD_LOGIC_1164.ALL;
•
•  entity SPT is
•  generic (n : INTEGER);
•   Port(
•      X  : in std_logic_vector(n-1 downto 0);
•      Y  : in std_logic_vector(n-1 downto 0);
•      Tin: in std_logic;
•
```

```
      S    : out std_logic_vector(n-1 downto 0);
      Tout: out std_logic
      );
  end SPT;

  architecture Behavioral of SPT is
  begin

      process(X, Y, Tin)
      variable AUX : std_logic_vector(n downto 0);
      begin

          AUX(0) := Tin;

          for i in 0 to n-1 loop
              S(i) <= X(i) xor Y(i) xor AUX(i);
              AUX(i+1) := (X(i) and Y(i)) or (X(i) and AUX(i)) or
   (Y(i) and AUX(i));
          end loop;

          Tout <= AUX(n);

      end process;

  end Behavioral;
```

- Pipeline Matricial

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


entity Matrix is
generic(n:natural := 8);
 Port (

    CLK : in std_logic;
    EN  : in std_logic;
    X   : in std_logic_vector(n-1 downto 0);
    Y   : in std_logic_vector(n-1 downto 0);

    P   : out std_logic_vector(2*n-1 downto 0)
  );
end Matrix;

architecture Behavioral of Matrix is

    component M is
      Port (

      X  : in std_logic;
```

```
      Y  : in std_logic;
      A  : in std_logic;
      Tin: in std_logic;

      S   : out std_logic;
      Tout: out std_logic
      );
    end component;


    component Reg is
      generic ( n : natural);
      Port (

      CLK : in std_logic;
      EN  : in std_logic;

      D  : in std_logic_vector(3*n - 1 downto 0);
      Q : out std_logic_vector(3*n - 1 downto 0)

      );
    end component Reg;

    signal iy: std_logic_vector(n-1 downto 0) := (others => '0');

    type registre is array(0 to n-1) of std_logic_vector(3*n-1 downto
0);
    signal OUT_REGISTRU : registre := (others => (others =>'0'));
    signal IN_REGISTRU : registre := (others => (others =>'0'));

    type sumatoare is array(0 to n-1) of std_logic_vector(n-1 downto
0);
    signal SUMATOR_NIVEL : sumatoare := (others => (others =>'0'));

    type transport_sumatoare is array(0 to n-1) of std_logic_vector(n
downto 0);
    signal TRANSPORT_NIVEL : transport_sumatoare := (others => (others
=>'0'));

begin

generare_registru: for i in 0 to n-1 generate
    Regg: Reg generic map (n) port
map(CLK,EN,IN_REGISTRU(i),OUT_REGISTRU(i));
end generate generare_registru;

swap: for i in 0 to n-1 generate
    iy(i) <= Y(n-1-i);
end generate swap;

IN_REGISTRU(0)(3*n-1 downto n) <= X & iy;
IN_REGISTRU(1) <= OUT_REGISTRU(0)(3*n-1 downto 2*n) &
OUT_REGISTRU(0)(2*n-2 downto n) & TRANSPORT_NIVEL(0)(n) &
SUMATOR_NIVEL(0);
intermediate_val: for i in 2 to n-1 generate
    IN_REGISTRU(i) <= OUT_REGISTRU(i-1)(3*n-1 downto 2*n) &
OUT_REGISTRU(i-1)(2*n-2 downto n-1+i) &TRANSPORT_NIVEL(i-1)(n) &
SUMATOR_NIVEL(i-1) & OUT_REGISTRU(i-1)(i-2 downto 0);
end generate intermediate_val;

P(2*n-2) <= SUMATOR_NIVEL(n-1)(n-1);
```

```
P(2*n-1) <= TRANSPORT_NIVEL(n-1)(n);

f : for i in 0 to n-2 generate
    P(n+i-1) <= SUMATOR_NIVEL(n-1)(i);
    P(i) <= OUT_REGISTRU(n-1)(i);
end generate f;

ii: for i in 0 to n-1 generate
    jj: for j in 0 to n-1 generate
        sums: M port map( OUT_REGISTRU(i)(2*n+j),OUT_REGISTRU(i)(2*n-
1),OUT_REGISTRU(i)(i+j),TRANSPORT_NIVEL(i)(j),SUMATOR_NIVEL(i)(j),TRAN
SPORT_NIVEL(i)(j+1));
    end generate jj;
end generate ii;


end Behavioral;
```

- Pipeline Wallace

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

entity Wallace is
  Port (
        Clk: in std_logic;
        EN : in std_logic;
        X: in std_logic_vector(7 downto 0);
        Y: in std_logic_vector(7 downto 0);

        P: out std_logic_vector(15 downto 0)
   );
end Wallace;

architecture Behavioral of Wallace is

component Reg is
generic (n : INTEGER);
  Port (

  CLK : in std_logic;
  EN  : in std_logic;
  D   : in std_logic_vector(n-1 downto 0);
  Q : out std_logic_vector(n-1 downto 0)

  );
end component Reg;

component SST is
Port (
    A : in STD_LOGIC_VECTOR (15 downto 0);
```

```
      B : in STD_LOGIC_VECTOR (15 downto 0);
      C : in STD_LOGIC_VECTOR (15 downto 0);
      S : OUT STD_LOGIC_VECTOR (15 downto 0);
      T : OUT STD_LOGIC_VECTOR (15 downto 0));
  end component SST;

  component SPT is
  generic (n : INTEGER);
   Port(
      X  : in std_logic_vector(n-1 downto 0);
      Y  : in std_logic_vector(n-1 downto 0);
      Tin: in std_logic;

      S   : out std_logic_vector(n-1 downto 0);
      Tout: out std_logic
      );
  end component SPT;

  type reg_data is array(0 to 4) of std_logic_vector(127 downto
  0);
  signal REG_IN : reg_data := (others => (others => '0'));
  signal REG_OUT: reg_data := (others => (others => '0'));

  type sum_and_transp is array(0 to 5) of std_logic_vector(15
  downto 0);
  signal SUMA : sum_and_transp := (others => (others => '0'));
  signal TRANSP : sum_and_transp := (others => (others => '0'));

  signal X_aux, Y_aux : std_logic_vector(7 downto 0) := (others =>
  '0');
  signal M0,M1,M2,M3,M4,M5,M6,M7 : std_logic_vector(15 downto 0)
  := (others => '0');
  signal PP_reg : std_logic_vector(127 downto 0) := (others =>
  '0');
  signal Tout : std_logic := '0';

  begin
  stocare_deinmultit: Reg generic map (8) port map
  (CLK,EN,X,X_aux);
  stocare_inmultitor: Reg generic map (8) port map
  (CLK,EN,Y,Y_aux);

  M0 <= b"0000_0000"& (X_aux(7) and Y_aux(0)) & (X_aux(6) and
  Y_aux(0)) & (X_aux(5) and Y_aux(0)) & (X_aux(4) and
  Y_aux(0))&(X_aux(3) and Y_aux(0)) & (X_aux(2) and Y_aux(0)) &
  (X_aux(1) and Y_aux(0)) & (X_aux(0) and Y_aux(0));
  M1 <= b"0000_000"&  (X_aux(7) and Y_aux(1)) & (X_aux(6) and
  Y_aux(1)) & (X_aux(5) and Y_aux(1)) & (X_aux(4) and
  Y_aux(1))&(X_aux(3) and Y_aux(1)) & (X_aux(2) and Y_aux(1)) &
  (X_aux(1) and Y_aux(1)) & (X_aux(0) and Y_aux(1)) & '0';
  M2 <= b"0000_00"&   (X_aux(7) and Y_aux(2)) & (X_aux(6) and
  Y_aux(2)) & (X_aux(5) and Y_aux(2)) & (X_aux(4) and
  Y_aux(2))&(X_aux(3) and Y_aux(2)) & (X_aux(2) and Y_aux(2)) &
  (X_aux(1) and Y_aux(2)) & (X_aux(0) and Y_aux(2)) & "00";
```

```
M3 <= b"0000_0"&    (X_aux(7) and Y_aux(3)) & (X_aux(6) and
Y_aux(3)) & (X_aux(5) and Y_aux(3)) & (X_aux(4) and
Y_aux(3))&(X_aux(3) and Y_aux(3)) & (X_aux(2) and Y_aux(3)) &
(X_aux(1) and Y_aux(3)) & (X_aux(0) and Y_aux(3)) & "000";
M4 <= b"0000"&      (X_aux(7) and Y_aux(4)) & (X_aux(6) and
Y_aux(4)) & (X_aux(5) and Y_aux(4)) & (X_aux(4) and
Y_aux(4))&(X_aux(3) and Y_aux(4)) & (X_aux(2) and Y_aux(4)) &
(X_aux(1) and Y_aux(4)) & (X_aux(0) and Y_aux(4)) & "0000";
M5 <= b"000"&       (X_aux(7) and Y_aux(5)) & (X_aux(6) and
Y_aux(5)) & (X_aux(5) and Y_aux(5)) & (X_aux(4) and
Y_aux(5))&(X_aux(3) and Y_aux(5)) & (X_aux(2) and Y_aux(5)) &
(X_aux(1) and Y_aux(5)) & (X_aux(0) and Y_aux(5)) & b"0000_0";
M6 <= b"00"&        (X_aux(7) and Y_aux(6)) & (X_aux(6) and
Y_aux(6)) & (X_aux(5) and Y_aux(6)) & (X_aux(4) and
Y_aux(6))&(X_aux(3) and Y_aux(6)) & (X_aux(2) and Y_aux(6)) &
(X_aux(1) and Y_aux(6)) & (X_aux(0) and Y_aux(6)) & b"0000_00";
M7 <= '0' &         (X_aux(7) and Y_aux(7)) & (X_aux(6) and
Y_aux(7)) & (X_aux(5) and Y_aux(7)) & (X_aux(4) and
Y_aux(7))&(X_aux(3) and Y_aux(7)) & (X_aux(2) and Y_aux(7)) &
(X_aux(1) and Y_aux(7)) & (X_aux(0) and Y_aux(7)) & b"0000_000";

PP_reg <= M7 & M6 & M5 & M4 & M3 & M2 & M1 & M0;

REG_IN(0) <= PP_reg;
Reg_0: Reg generic map(128) port map(CLK, EN, REG_IN(0),
REG_OUT(0) );

SST_0: SST  port map(
        A => REG_OUT(0)(47 downto 32),
        B => REG_OUT(0)(31 downto 16),
        C => REG_OUT(0)(15 downto 0),
        S => SUMA(0),
        T => TRANSP(0)
);

SST_1: SST port map(

        A => REG_OUT(0)(95 downto 80),
        B => REG_OUT(0)(79 downto 64),
        C => REG_OUT(0)(63 downto 48),
        S => SUMA(1),
        T => TRANSP(1)
);

REG_IN(1)(95 downto 80) <= REG_OUT(0)(127 downto 112);
REG_IN(1)(79 downto 64) <= REG_OUT(0)(111 downto 96);
REG_IN(1)(63 downto 48) <= TRANSP(1);
REG_IN(1)(47 downto 32) <= SUMA(1);
REG_IN(1)(31 downto 16) <= TRANSP(0);
REG_IN(1)(15 downto 0) <= SUMA(0);
Reg_1: Reg generic map(128) port map(CLK, EN, REG_IN(1),
REG_OUT(1));

SST_2: SST  port map(
        A => REG_OUT(1)(47 downto 32),
```

```
          B => REG_OUT(1)(31 downto 16),
          C => REG_OUT(1)(15 downto 0),
          S => SUMA(2),
          T => TRANSP(2)
  );

  SST_3: SST port map(

          A => REG_OUT(1)(95 downto 80),
          B => REG_OUT(1)(79 downto 64),
          C => REG_OUT(1)(63 downto 48),
          S => SUMA(3),
          T => TRANSP(3)
  );

  REG_IN(2)(63 downto 48) <= TRANSP(3);
  REG_IN(2)(47 downto 32) <= SUMA(3);
  REG_IN(2)(31 downto 16) <= TRANSP(2);
  REG_IN(2)(15 downto 0) <= SUMA(2);
  Reg_2:Reg generic map(128) port map(CLK, EN, REG_IN(2),
  REG_OUT(2));

  SST_4: entity work.SST port map(
          A => REG_OUT(2)(47 downto 32),
          B => REG_OUT(2)(31 downto 16),
          C => REG_OUT(2)(15 downto 0),
          S => SUMA(4),
          T => TRANSP(4)
  );

  REG_IN(3)(47 downto 32)  <= REG_OUT(2)(63 downto 48);
  REG_IN(3)(31 downto 16) <= TRANSP(4);
  REG_IN(3)(15 downto 0) <= SUMA(4);
  Reg_3: Reg generic map(128) port map(CLK, EN, REG_IN(3),
  REG_OUT(3));

  SST_5: SST port map(
          A => REG_OUT(3)(47 downto 32),
          B => REG_OUT(3)(31 downto 16),
          C => REG_OUT(3)(15 downto 0),
          S => SUMA(5),
          T => TRANSP(5)
  );

  REG_IN(4)(31 downto 16) <= TRANSP(5);
  REG_IN(4)(15 downto 0) <= SUMA(5);
  Reg_4: Reg generic map(128) port map(CLK, EN, REG_IN(4),
  REG_OUT(4));

  SPT_0 : SPT generic map (16) port map(
              X => REG_OUT(4)(31 downto 16),
              Y => REG_OUT(4)(15 downto 0),
              Tin => '0',
```

```
                S => P,
                Tout => Tout
        );
    end Behavioral;

```