



**UNIVERSITATEA
TEHNICĂ**
DIN CLUJ-NAPOCA

Catedra de calculatoare



Circuite de înmulțire pipeline pentru numere întregi

Student: Moldovan Liana

Grupa: 30238

Îndrumator: dr.ing Cristi Mocan

Data: 10.10.2021

CUPRINS

1	Rezumat	3
2	Introducere	4
2.1	Circuite de inmultire pipeline	4
3	Fundamentare teoretică.....	5
3.1	Înmulțirea prin deplasare si adunare.....	5
3.2	Înmulțirea într-o bază superioară	5
3.3	Înmulțirea matriceală:.....	6
3.4	Arborele Wallace.....	7
3.5	Algoritmul lui Booth	9
3.7	Solutii posibile.....	10
4	Proiectare si implementare.....	12
4.1	Schema bloc a circuitului	13
4.3	Componentele circuitului	14
4.3.1	Sumator elementar.....	14
4.3.2	Sumatorul cu propagarea succesivă a transportului(SPT).....	14
4.3.3	Bistabil D.....	15
4.3.4	Sumator cu salvarea tranportului.....	15
4.3.5	Modul pentru generarea produselor partiale.....	16
4.3.6	Registru de memorare	17
4.3.7	Modulul principal	17
5	Rezultate experimentale.....	18
5.1	Simularea modulelor	18
5.1.1	Sumatorul elementar.....	18
5.1.2	Sumator cu salvarea succesiva a transportului:	19
5.1.3	Bistabil D:.....	19
5.1.4	Sumator cu propagarea succesiva a transportului:	20
5.1.5	Modul pentru generarea produselor partiale:	20
5.1.6	Modulul principal:	21
5	Concluzii.....	22
6	Bibliografie	23
8.	Anexe(cod).....	24

1 Rezumat

Multiplicarea este unul dintre cele mai semnificative procese în procesarea semnalului digital. Performanța vitezei de multiplicare influențează performanța generală a sistemelor informatice digitale, precum și în procesarea semnalului digital. Mulți algoritmi și arhitecturi de înaltă implementare au fost sugerați pentru a îmbunătăți și accelera operațiunile de multiplicare.

Obiectivul proiectului consta în realizarea înmulțirii pipeline a două numere întregi pe 32 biți. Limbajul de programare utilizat este VHDL iar ca tehnologie utilizată pentru exemplificarea funcționării am folosit placa Basys3.

Realizarea proiectului a condus la obținerea rezultatului unor înmulțiri, la o diferență de timp egală cu un tact. Deoarece în arhitectura proiectului avem 10 etaje, iar fiecare etaj păstrează cu ajutorul registrelor un rezultat intermediar, putem avea simultan 10 perechi de numere care se pot înmulți.

Ca și concluzii, înmulțirea pipeline este o tehnică care oferă rezultate rapide pentru volume mai mari de date. În pipeline, întârzierea apariției rezultatului după ultimul etaj, este egală cu întârzierea maximă a unui etaj, pe când întârzierea în circuitele combinatoriale este egală cu întârzierea semnalelor pe întreg circuitul.

2 Introducere

Circuitul de înmulțire este unul dintre blocurile hardware cele mai folosite în procesoare de semnale digitale și microprocesoare. Aceste circuite de înmulțire sunt o opțiune mai bună pentru procesări de date unde este importantă viteza și se pune accent pe aceasta. Există mai multe arhitecturi și algoritmi posibili pentru înmulțirea numerelor întregi, dintre acestea unele dintre soluțiile posibile ale acestui proiect ar fi: Înmulțirea prin tehnica pipeline, tehnica de adunare cu salvarea transportului, metoda arborelui Wallace, tehnica Booth modificată folosind Pipeline. Modul în care funcționează aceste înmulțitoare este următorul: manipulează două intrări de date pentru a genera mai multe rezultate de înmulțiri parțiale, până se ajunge la rezultatul final.

2.1 Circuite de înmulțire pipeline

Prin utilizarea procedurii pipeline, procesele executate sunt paralelizate, existând etaje, astfel circuitul se împarte în blocuri mai mici iar salvarea rezultatelor se face folosind registre.

Registrele au un rol foarte important, deoarece salvează rezultatele intermediare.

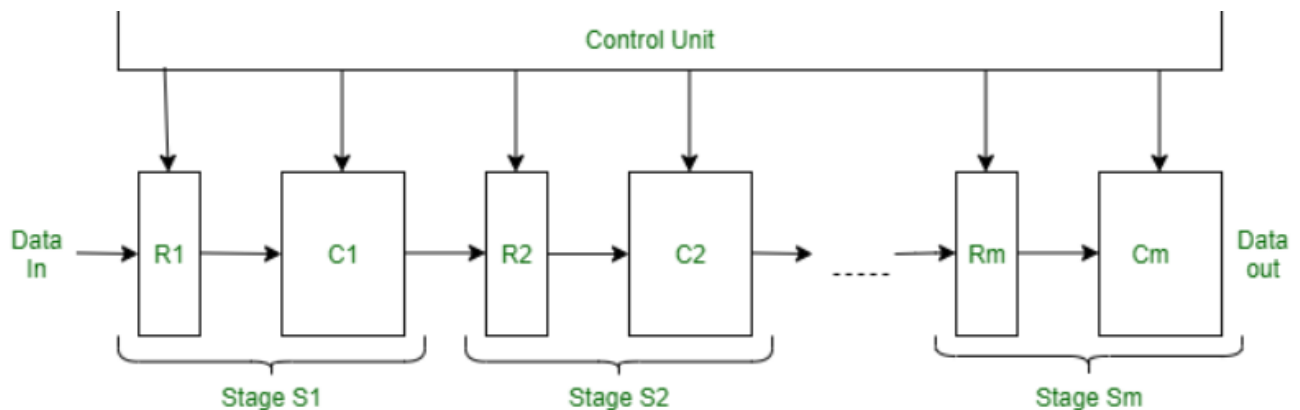


Figure - Structure of a Pipeline Processor

3 Fundamentare teoretică

Se analizează diferitele metode de înmulțire:

3.1 Înmulțirea prin deplasare și adunare

Adună de înmulțitul X cu el însuși de Y ori.

Algoritmul:

- Se selectează cifrele înmulțitorului una câte una de la dreapta la stânga.
- Se înmulțește de înmulțitul cu cifra selectată a înmulțitorului.
- Se plasează produsul intermediar la stânga rezultatelor precedente.

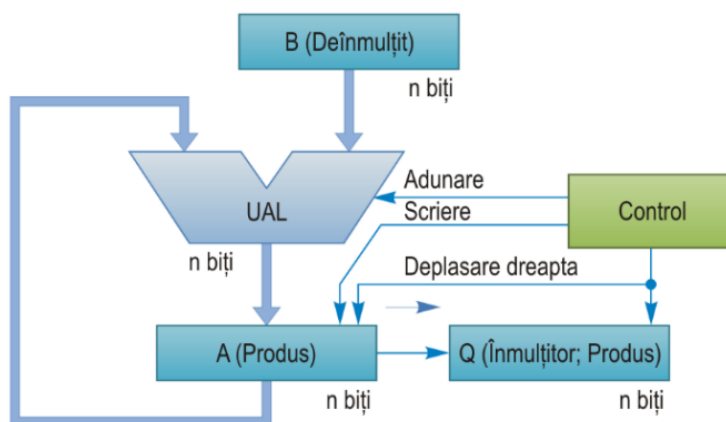


Fig 3.1.1 Schema bloc a unui circuit de deplasare și adunare

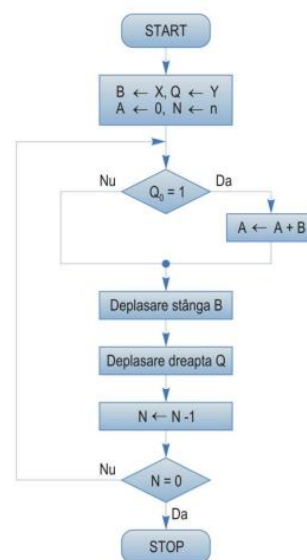


Fig 1.2 Organigrama algoritmului de înmulțire prin deplasare și adunare

3.2 Înmulțirea într-o bază superioară

Se examinează mai mulți biți ai înmulțitorului Y în fiecare pas → crește viteza operației.

Înmulțire în baza 4: sunt examinați 2 biți:

- 00: nu se execută adunare.
- 01: se adună X la produsul parțial.
- 10: se adună $2X$ la produsul parțial.
- 11: se adună $X + 2X$ la produsul parțial.

Y_{2i+1}	Y_{2i}	Y_{2i-1}	Operații
0	0	0	Deplasare la dreapta cu două poziții
0	0	1	Adunare X, deplasare la dreapta cu două poziții
0	1	0	Adunare X, deplasare la dreapta cu două poziții
0	1	1	Adunare 2X, deplasare la dreapta cu două poziții
1	0	0	Adunare -2X, deplasare la dreapta cu două poziții
1	0	1	Adunare -X, deplasare la dreapta cu două poziții
1	1	0	Adunare -X, deplasare la dreapta cu două poziții
1	1	1	Deplasare la dreapta cu două poziții

Fig 3.2.1 Operațiile pentru înmulțirea într-o baza superioară

3.3 Înmulțirea matriceală:

Pentru creșterea vitezei, se pot utiliza circuite combinaționale de înmulțire.

Logică suplimentară care permite calculul produsului într-un pas

Matrice de elemente combinaționale simple → adunare, deplasare Considerăm înmulțirea a două numere binare întregi fără semn:

$$\begin{aligned} X &= x_{n-1} \dots x_1 x_0 \\ Y &= y_{n-1} \dots y_1 y_0 \end{aligned} \quad (1)$$

Produsul P se poate scrie:

$$P = X * Y = \left(\sum_{i=0}^{n-1} 2^i * x_i \right) * \left(\sum_{j=0}^{n-1} 2^j * y_j \right)$$

Fig3.3.1 Ecuația produsului P

Fiecare termen produs de 1 bit $x_i \cdot y_j$ se poate calcula cu ajutorul unei porți ȘI. O matrice de $n \times n$ porți ȘI poate calcula toți termenii $x_i \cdot y_j$ simultan. Termenii sunt însumați cu o matrice de $n(n-1)$ sumatoare elementare. Circuitul rezultat este similar cu un sumator bidimensional cu transport succesiv. Deplasările implicate de factorii 2^i și 2^j sunt implementate prin deplasarea spațială a sumatoarelor pe direcția x și y.

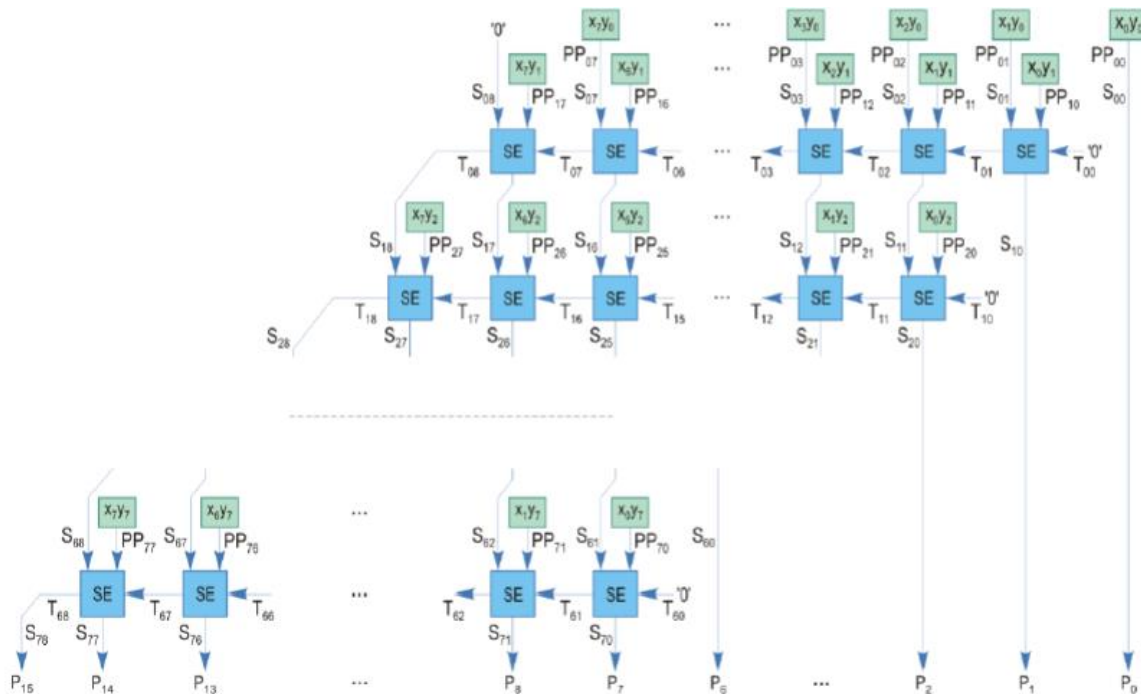


Fig 3.3.2 Înmulțirea matriceală pentru numere de 8 biți fara semn.

3.4 Arborele Wallace

Înmulțirea a două numere de câte n biți necesită adunarea a n produse parțiale. Circuitele de înmulțire anterioare execută înmulțirea într-un timp $O(n)$.

Timpul poate fi redus la $O(\log n)$ prin utilizarea unui arbore.

Arborele cel mai simplu: combină perechi de produse parțiale \rightarrow numărul produselor parțiale ar fi redus de la n la $n/2$.

C. S. Wallace a arătat că produsele parțiale pot fi adunate mai rapid utilizând nivele multiple de SST (Sumatoare cu salvarea transportului) \rightarrow arbore Wallace

- În fiecare nivel al arborelui, numerele sunt grupate câte trei.
- Se utilizează câte un SS pentru adunarea numerelor din fiecare grup
- Procesul continuă până când rămân numai două numere de adunat.

- Pentru adunarea lor se utilizează un SPT. Fiecare nivel reduce numărul termenilor care trebuie adunați cu un factor de 1,5 $\rightarrow O(\log_{1,5} n)$.

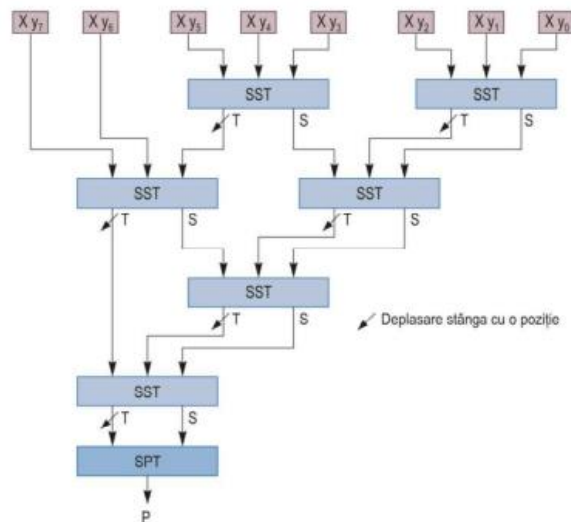


Fig 3.4.1 Circuit de înmulțire pentru numere de opt biți utilizând un arbore Wallace

3.5 Algoritmul lui Booth

Tehnica Booth:

- Elimină necesitatea conversiei operanzilor la forma pozitivă.
- Poate reduce numărul operațiilor necesare.

Ideea principală: dacă se poate efectua atât adunare, cât și scădere, există mai multe posibilități de a calcula un produs.

- Un șir de cifre de 0 din înmulțitor necesită numai deplasare.
- Un șir de cifre de 1 poate fi tratat ca un număr cu valoarea $L - R$.
 - L – ponderea cifrei 0 dinaintea cifrei 1 celei mai din stânga.
 - R – ponderea cifrei 1 celei mai din dreapta.
- La înmulțirea prin tehnica Booth se consideră câte doi biți adiacenți ai înmulțitorului pentru a determina operația care trebuie efectuată.

Y_i	Y_{i-1}	Operații
0	0	Deplasare la dreapta
0	1	Adunare de înmulțit, deplasare la dreapta
1	0	Scădere de înmulțit, deplasare la dreapta
1	1	Deplasare la dreapta

Fig 3.5.1 Operații efectuate la înmulțirea prin metoda Booth

3.6 Circuite de înmulțire pipeline

Tehnica pipeline: Utilizează paralelismul

- Se suprapun etapele de execuție ale unei operații aritmetice.
- Fiecare etapă este executată de un etaj al sistemului pipeline.
- Etaj: registru + circuit de prelucrare.

Un circuit de înmulțire matriceală pipeline cu n etaje poate suprapune calculul a n produse \rightarrow înmulțirea vectorilor întregi.

- Poate genera un nou rezultat în fiecare ciclu de ceas.
- Dezavantaj: viteza redusă a logicii de propagare a transportului din fiecare etaj.
- Numărul celulelor M necesare: n^2 .
- Capacitatea tuturor registrelor buffer: $\sim 3n^2 \rightarrow$ costisitor din punct de vedere hardware.

Tehnica de adunare cu salvarea transportului \rightarrow avantajoasă pentru implementarea pipeline.

- Se pot aduna m numere printr-o rețea de sumatoare cu salvarea transportului \rightarrow rezultatul este sub forma (S, T).
- S și T trebuie adunate printr-un sumator convențional cu propagarea transportului.

3.7 Soluții posibile

O tehnică plauzibilă care se poate utiliza este **tehnica de înmulțire matriceală pipeline** care folosește o celulă M de înmulțire(funcția si) și adunare specifică.

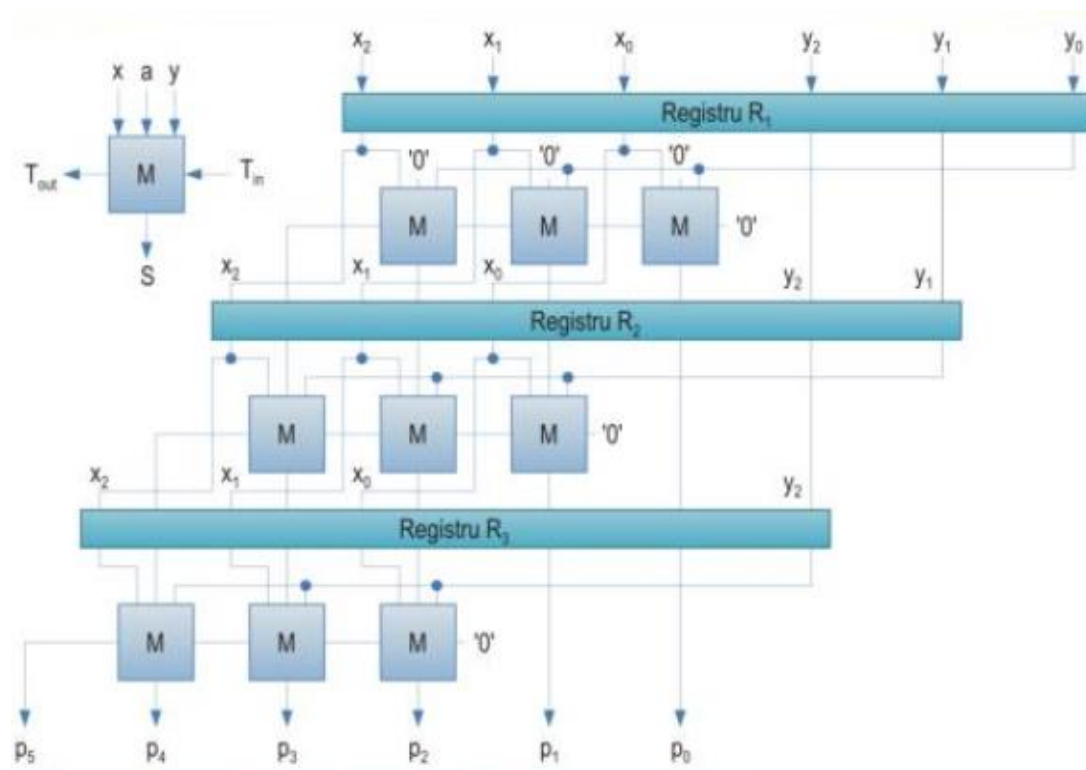


Fig 3.7.1 Înmulțire matriceală pipeline

O altă tehnică derivă din prima, modificarea care se efectuează este pentru creșterea vitezei, și anume faptul că se folosește un sumator cu salvarea transportului, fapt care elimină propagarea transportului între elementele din același etaj, până la ultimul etaj, unde se găsește transportul final. Această tehnică poartă numele de **adunare cu salvarea transportului**.

O alta tehnica este tehnica **Wallace pipeline**, prin aceasta se reduce timpul de inmultire $O(n)$ la timpul $O(\log n)$, la fel cu folosirea sumatoarelor cu salvare transport(SST). Dar aici produsele parțiale se grupează pentru creșterea vitezei circuitului.

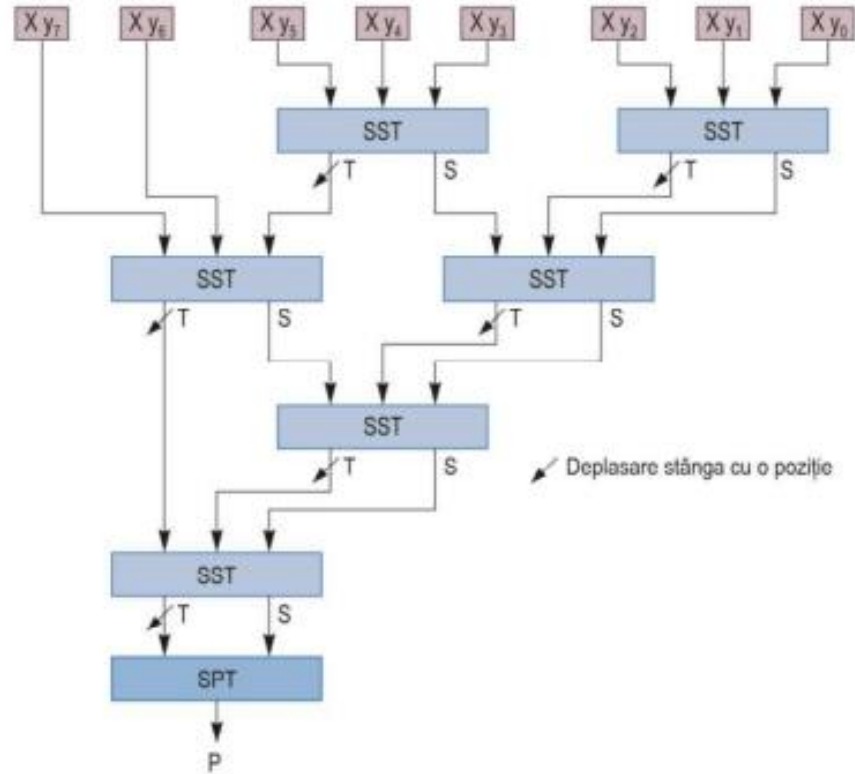


Fig 2.7.2 Circuit de înmulțire pentru numere de 8 biți utilizând un arbore Wallace

4 Proiectare si implementare

Operațiile aritmetice pot fi implementate printr-un sistem pipeline.

Inmultirea pipeline se realizeaza pe mai multe etaje, fiecare etaj fiind compus din registre de memorare formate din bistabile care realizeaza memorarea a informatiilor de la etajul anterior si un circuit combinational de calculare a informatiilor necesare urmatorului etaj. In acest fel informatia poate sa circule prin etajele circuitului cumva in paralel. Astfel se pot efectua in circuitul realizat mai multe calcule ale produselor in paralel, din acest punct de vedere, pentru calcularea mai multor produse circuitul este unul eficient deoarece intarzierea dintre 2 rezultate succesive va fi data de intarzierea maxima dintre etaje.

Un dezavantaj al introducerii tehnicii pipeline este ca circuitul devine secvential, fiind necesar un semnal de tact la nivelul fiecarui etaj, pentru a se realiza circulatia informatiei.

Arborele Wallace prezinta un principiu de simplificare a numarului de SST-uri necesare pentru adunarea produselor partiale. C.S. Wallace a aratat ca produsele partiale pot fi adunate mai rapid folosind mai multe nivele de SST-uri. Astfel in fiecare nivel al arborelui, numerele sunt grupate cate 3 si se utilizeaza un SST(Sumator cu salvarea transportului) pentru fiecare grup. Acest procedeu se repeta pentru fiecare nivel pana cand raman doar 2 numere de adunat. Pentru adunarea acestor 2 numere ramase se utilizeaza un SPT(Sumator cu propagarea transportului). Se observa faptul ca fiecare nivel al arborelui reduce numarul termenilor care trebuie adunati cu un factor de 1.5 $\Rightarrow O(\log_{1.5}n)$. Intre nivelele arborelui Wallace se adauga registre de memorare a sumei si transportului pentru a aplica tehnica pipeline.

Ca element inovator pentru un circuit de tip pipeline cu generator de produse partiale, am implementat tot in tehnica pipeline si transmiterea semnului operanzilor, prin etaje adiacente etajelor principale, cu scopul identificarii semnului corect a rezultatului dupa ce operanzii circula prin etajele pipeline. Acest lucru duce la eliminarea necesitatii de a crea 2 moduri pentru circuitul de inmultire: acela de lucru cu operanzi de acelasi semn sau de semne opuse.

4.1 Schema bloc a circuitului

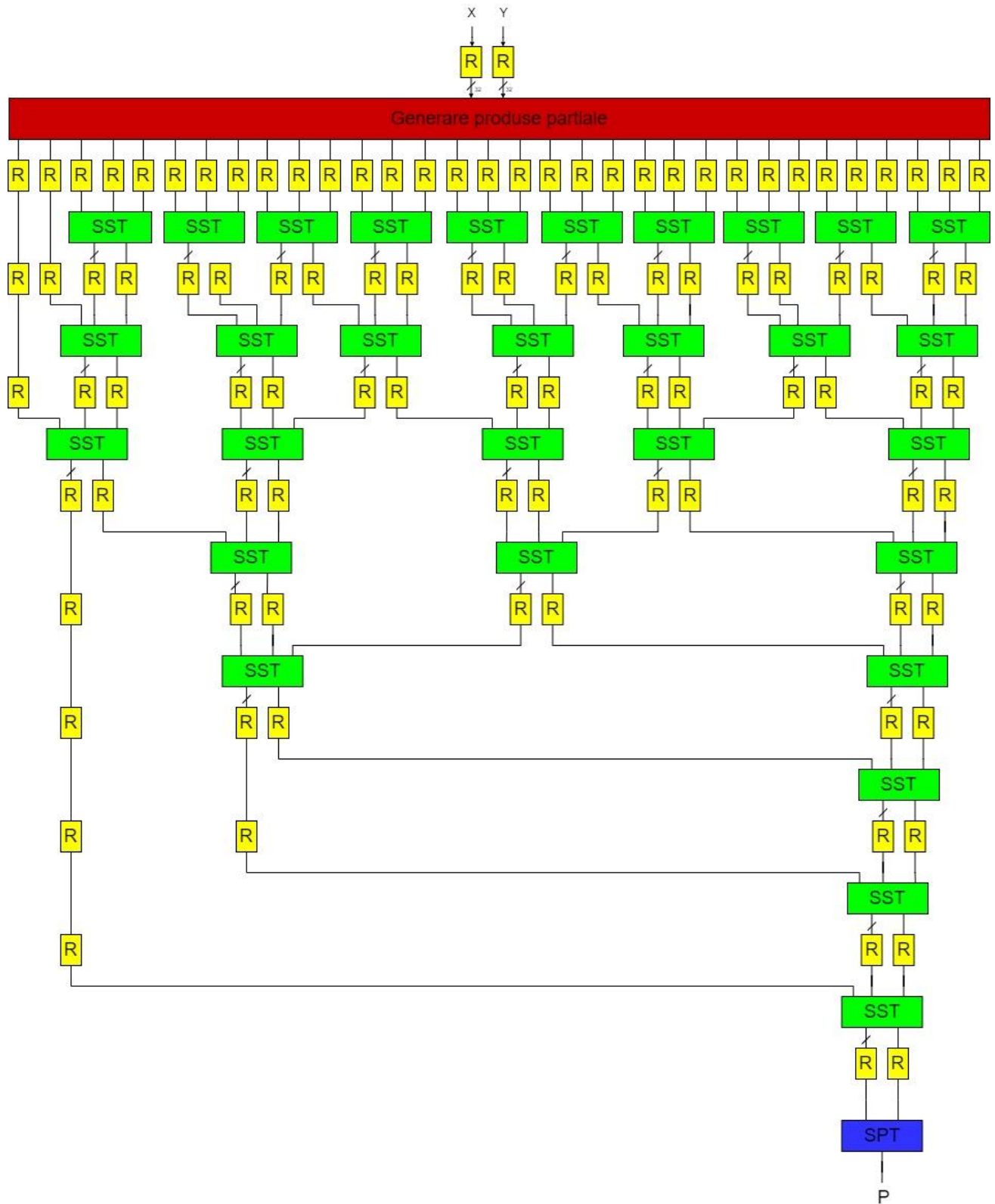


Fig 3.1.1 Arhitectura generala a proiectului

4.3 Componentele circuitului

4.3.1 Sumator elementar

Sumatorul elementar este o componenta principala in acest proiect, fiind necesar in implementarea metodelor.

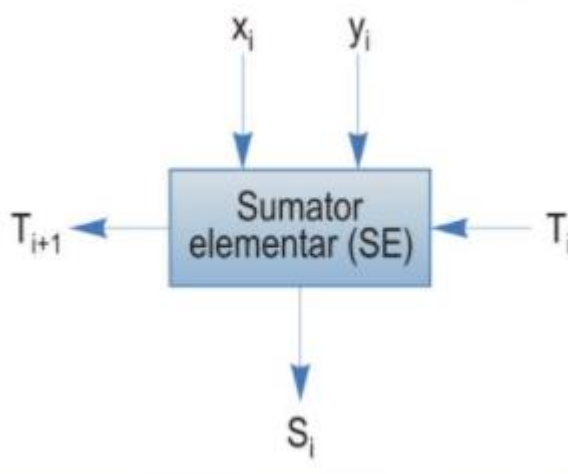


Fig 4.3.1 Schema sumatorului elementar

Un sumator elementar aduna doua intrari de cate un bit. Doua din intrari sunt bitii care trebuie adunați, notati cu x_i, y_i iar cealalta intrare este transportul de la bitul din pozitia mai puțin semnificativă, notat cu T_i . Ieșirile sunt bitul suma S_i si transportul către bitul din poziția mai semnificativă, T_{i+1} .

In arhitectura mare intra in componenta SST-urilor care ajuta la metoda arborelui Wallace folosita. SST este un sumator cu salvarea transportului, al carui rol este acela de a reduce timpul de propagare al semnalelor de transport.

4.3.2 Sumatorul cu propagarea succesivă a transportului(SPT)

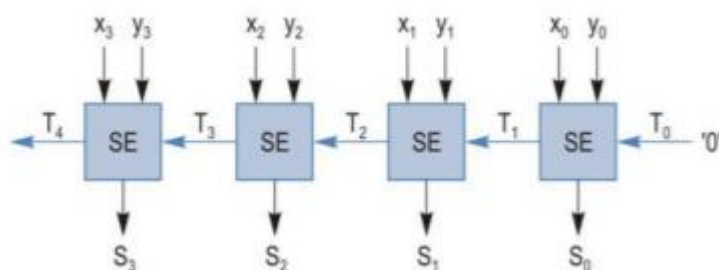


Fig 4.3.2 Schema sumatorului cu propagarea transportului pentru numere pe 4 biti

Unul din algoritmi de bază pentru adunare este algoritmul de adunare cu propagarea succesivă a transportului. Acest algoritm poate fi implementat în mod simplu cu sumatoare elementare. Principul este similar cu adunarea obișnuită. Fie $x_3x_2x_1x_0$ și $y_3y_2y_1y_0$ două numere binare de câte patru biți. Pentru adunarea acestor numere, se adună x_0 și y_0 pentru a determina cifra cea mai puțin semnificativă a sumei. Dacă rezultă un transport, acesta se adună cu x_1 și y_1 pentru a determina următoarea cifră mai semnificativă a sumei. Acest proces continuă până când se adună x_3 și y_3 . Dacă rezultă un transport final, acesta va deveni cifra cea mai semnificativă a sumei.

SPT va fi folosit în proiectarea arborelui Wallace.

4.3.3 Bistabil D

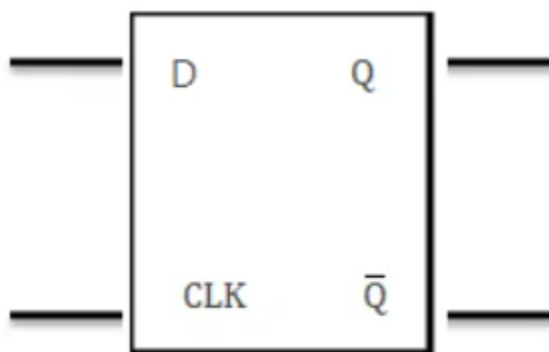


Fig 4.3.3 Schema unui bistabil D

Bistabilul D implementează un element de memorare. Durata în care $CLK=1$ și semnalul de ieșire urmărește semnalul de intrare se numește track, iar durata în care $CLK=0$ și ieșirea Q menține starea se numește hold.

4.3.4 Sumator cu salvarea transportului

Tehnica de adunare cu salvarea transportului este folosită pentru a reduce în principal timpul de propagare al semnalului de transport. Un astfel de sumator de 8 biți, spre exemplu, este o simplă insiruire a 8 sumatoare elementare pe intrările cărora se pun biții numerelor de adunat. Fiecare sumator elementar, generează 2 ieșiri: un bit de sumă și un bit de transport spre poziția mai semnificativă. Din aceste două, rezultă 2 siruri de biți, ca rezultate ale sumatorului cu salvarea transportului: un sir de sumă **S** de n biți și un sir de transport **T** de n biți.

Pentru obținerea rezultatului final, suma și transportul rezultate trebuie adunate folosind un SPT (sumator cu propagarea transportului).

Un astfel de sumator cu salvarea transportului este folosit pentru înmulțire la adunarea produselor parțiale, iar în arhitectura finală, intră în structura arborelui Wallace, pentru generarea acestor produse parțiale pe diferite nivele.

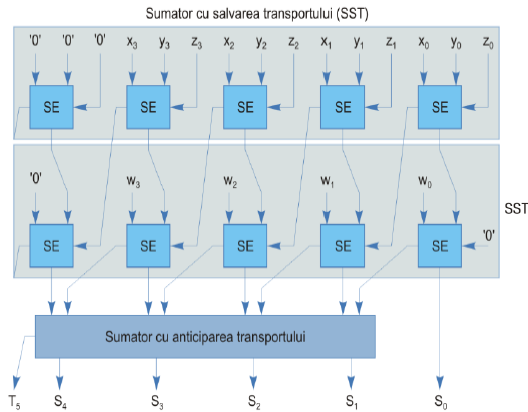


Fig 4.3.4 Sumator cu salvarea transportului (SST)

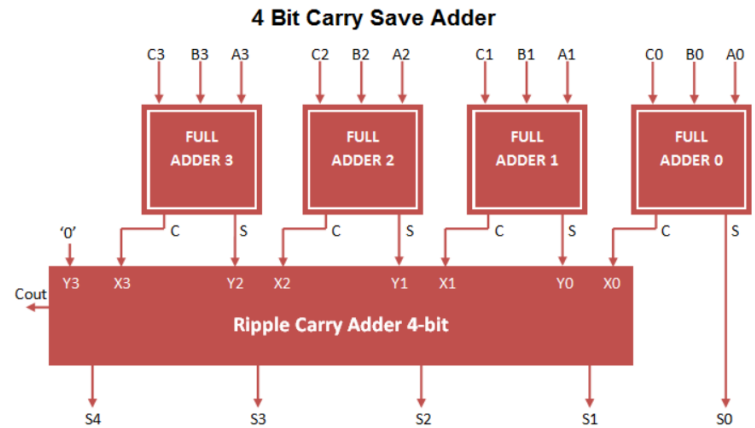


Fig 4.3.5 Schema sumator cu salvarea transportului pe 4 biti

4.3.5 Modul pentru generarea produselor parțiale

Pentru rezolvarea problemei generării produselor parțiale am folosit niste porti logice SI, pe langa un principiu de deplasare stanga la fiecare produs partial.

Dimensiunea produselor parțiale a 2 numere de n biti reprezinta numere pe $2n$ biti, deoarece trebuie efectuate deplasari. Xy_0 reprezinta produsul partial rezultat din tratarea primului bit din inmultitor, daca y_0 este egal cu valoare logica '1' atunci portile SI vor trece spre rezultatul generatorului exact bitii numarului X (32 in cazul nostru) pe pozitiile cele mai putin semnificative din cele 64 ale produsului partial. Generarea produsului partial urmator Xy_1 va avea acelasi principiu, dar portile SI vor fi plasate cu o pozitie mai la stanga (pentru a efectua o deplasare la stanga). Principiul se repeta si pentru celelalte produse parțiale.

De exemplu pentru 2 numere de 4 biti produsul partial Xy_0 va fi generat in urmatoarea mod:

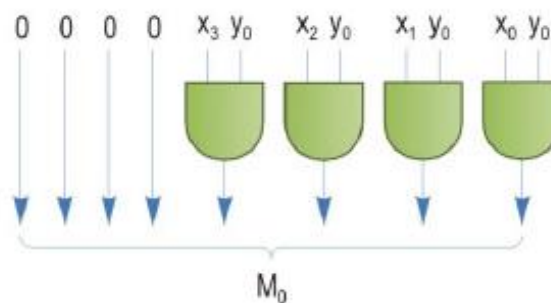


Fig 4.3.6 Generarea produsului partial M_0 pentru 2 numere pe 4 biti

4.3.6 Registru de memorare

Registru de memorare este necesar pentru pastrarea rezultatelor intermediare din fiecare etaj si constituie componenta principala intr-un circuit pipeline.

4.3.7 Modulul principal

Inmultirea matriceala cu SST-uri realizate pe numere de 32 de biti fara semn, necesita existenta a 30 de SST-uri interconectate, primul facand insumarea primelor 3 produse parțiale Xy_0 , Xy_1 si Xy_2 . Apoi in etajele de mai jos, se insumeaza urmatorul produs parțial si suma temporara de deasupra. Tot pentru a avea schema completa sunt necesare 32 de generatoare de produse.

Produsele parțiale rezultate vor fi pe 64 biti, la fel ca dimensiunea SST-urilor necesare, din considerente de deplasari necesare pentru un rezultat corect.

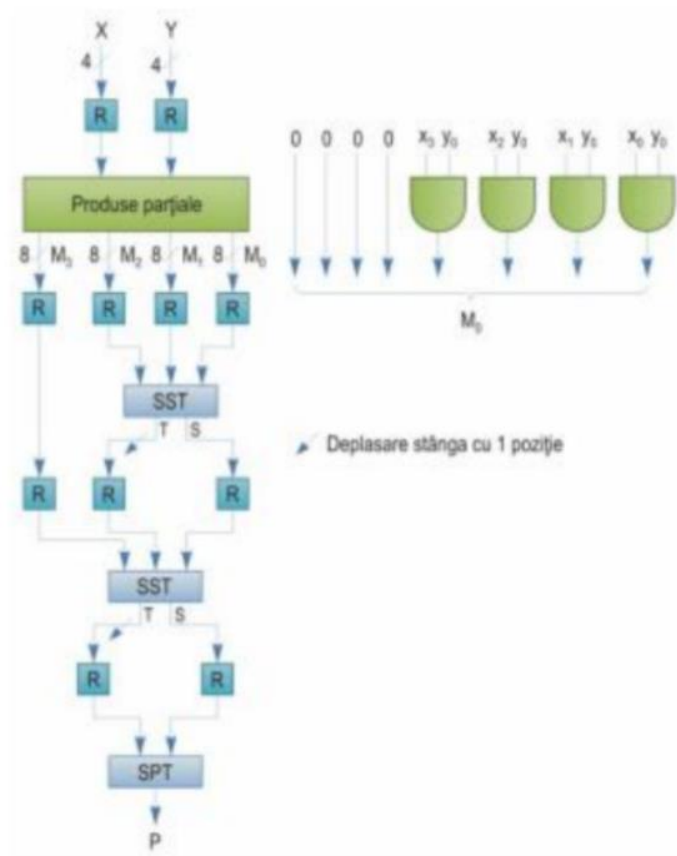


Fig 4.3.7 Schema bloc pentru un circuit de inmultire pipeline folosind metoda arborelui Wallace pe 4 biti.

Pe 32 de biti schema va fi mult extinsa, mergand pe acelasi pattern, pana la 30 de SST-uri. La final este necesar un SPT tot pe 64 de biti care sa realizeze insumarea lui S si T generate de ultimul SST.

5 Rezultate experimentale

5.1 Simularea modulelor

5.1.1 Sumatorul elementar

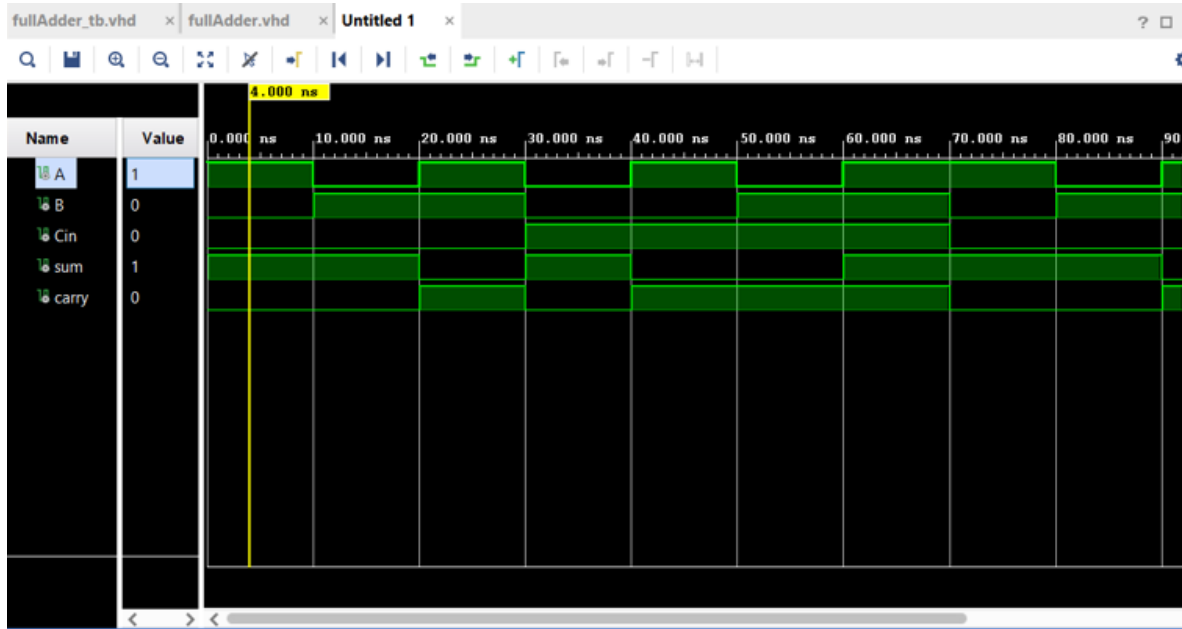


Fig 4.1.1

Cazuri:

x_i	y_i	T_i	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	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

5.1.2 Sumator cu salvarea succesiva a transportului:

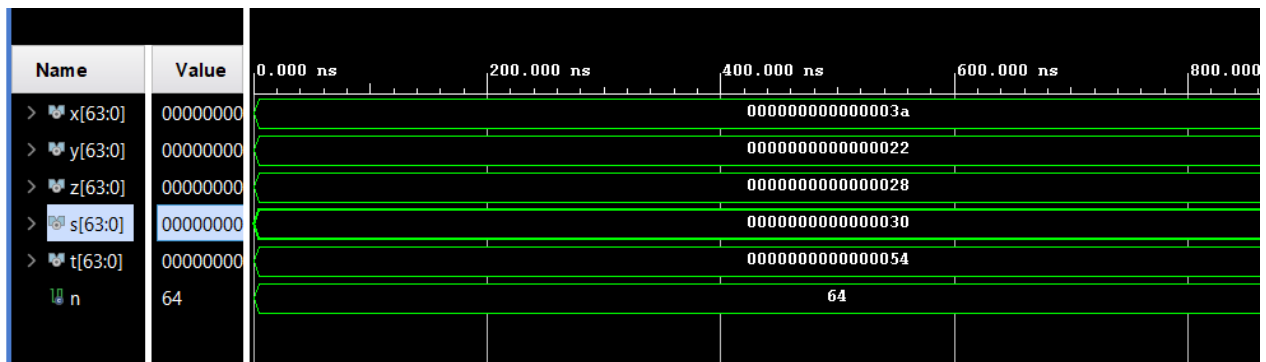


Fig 5.1.2

Am adunat numerele 3a(111010), folosind SST.

22(100010)

28(101000)

Rezultand: S=30(110000)

T=54(1010100)

5.1.3 Bistabil D:

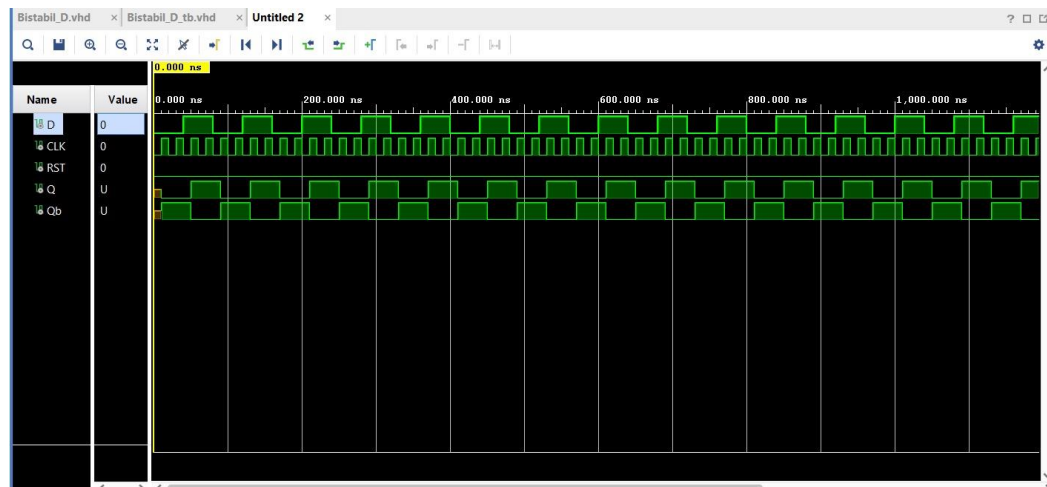





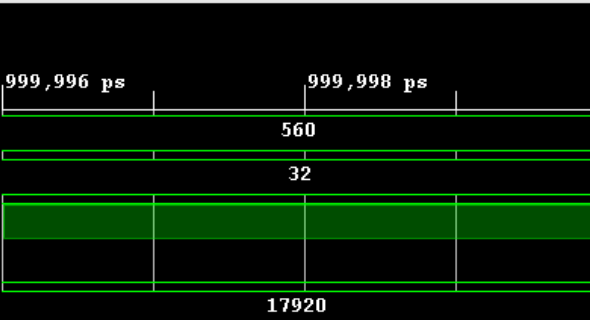


Fig 5.1.3

5.1.6 Modulul principal:

Name	Value
>  X[31:0]	560
>  Y[31:0]	32
 Clk	0
 Rst	0
>  P[63:0]	17920



Timing diagram showing signals X, Y, Clk, Rst, and P over time. The diagram has four columns representing time intervals. X is high (red) from 999,996 ps to 999,998 ps. Y is high (red) from 999,996 ps to 999,998 ps. Clk is high (red) from 999,996 ps to 999,998 ps. Rst is high (red) from 999,996 ps to 999,998 ps. P is high (red) from 999,996 ps to 999,998 ps.

5 Concluzii

Problema unui circuit de înmulțire pipeline a fost rezolvată prin împartirea unui circuit combinational în etaje, un etaj fiind constituit dintr-un circuit combinational și registre. Pentru rezolvarea tehnicii pipeline am folosit un generator de produse parțiale, astfel se vor obține în prima fază un număr de 32 de produse parțiale care sunt adunate mai apoi fiind trecute prin registre și sumatoare cu salvarea transportului. Principiul folosit pentru însumarea produselor parțiale este Wallace. Generatorul de produse parțiale prezintă o inovație prin faptul că generarea produselor parțiale se face pe baza unui generic transmis din modulul principal, iar sumatoarele (SST și SPT) sunt implementate cu sumatoare elementare pentru creșterea vitezei.

Avantajele proiectului constau în accentul pe viteză, pipeline aduce o ridicare a vitezei de lucru semnificative în cazul volumelor mari de numere de înmulțit. Întârzierea maximă devine egală cu întârzierea maximă a unui etaj, evitându-se întârzierea pe întreg circuitul. Pentru date voluminoase (microprocesoare, semnale digitale) care folosesc mult tehnica înmulțirii, varianta pipeline este una extrem de avantajoasă. Printre dezavantajele proiectului, o tehnică pipeline nu este folosită pentru una, două înmulțiri, deoarece numerele trebuie inevitabil să treacă prin toate etajele circuitului. Astfel pentru a înmulți o singură pereche de numere trebuie să se treacă prin 10 semnale de tact, sau, în varianta implementării circuitului pe placută, prin 10 apăsări de buton.

Ca aplicații, proiectul poate folosi la microprocesoare, tehnica înmulțirii fiind probabil cea mai folosită în cadrul acestora, se pot de altfel procesa semnele analogice sau digitale care necesită înmulțiri aferente, pentru calculatoare electronice sau diferiți senzori care necesită înmulțiri înainte de procesarea efectivă.

6 Bibliografie

- [1] Monika Maneet al, International Journal of Computer Science and Mobile Computing, Vol.4 Issue.3, March-2015, pg. 536-541 Performance Pipeline Signed 64*64 bit Multiplier using Radix-32 Modified Booths Algorithm and Wallace Structure.
- [2] Z. F. Baruch, Structura sistemelor de calcul – Înmulțire

8. Anexe(cod)

8.1 Sumator elementar

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

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx leaf cells in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity SE is
    Port (X:in std_logic;
          Y:in std_logic;
          Tin:in std_logic;
          Tout:out std_logic;
          S:out std_logic );
end SE;

architecture Behavioral of SE is

begin

    S<=X xor Y xor Tin;
    Tout<= (X and Y) or ((X or Y) and Tin);

end Behavioral;
```


8.2 Bistabil D

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

entity Bistabil_D is
Port ( D, CLK, RST : in  STD_LOGIC;
      Q, Qb : out  STD_LOGIC);
end Bistabil_D;

architecture Behavioral of Bistabil_D is

begin
process (D, CLK, RST)
begin

if (RST = '1') then
Q <= '0';
elsif (rising_edge(CLK)) then
Q <= D;
Qb <= not D;

end if;
end process;
end Behavioral;
```

8.3 Sumator cu salvarea transportului

```

entity SST is
    generic (n: INTEGER);
    Port (X:in std_logic_vector(n-1 downto 0);
          Y:in std_logic_vector(n-1 downto 0);
          Z:in std_logic_vector(n-1 downto 0);
          S:out std_logic_vector(n-1 downto 0);
          T:out std_logic_vector(n-1 downto 0));
end SST;

architecture Behavioral of SST is

    component SE is
        Port (X:in std_logic;
              Y:in std_logic;
              Tin:in std_logic;
              Tout:out std_logic;
              S:out std_logic );
    end component;

    signal C:std_logic_vector(n downto 0);

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

    begin

        C(0) <= '0';

        g0: for i in 0 to n-1 generate
            SE_i: SE port map (X=>X(i), Y=>Y(i), Tin=>Z(i), Tout=>C(i+1), S=>S(i));
            tmp(i) <= C(i+1);
        end generate;

        g2: for i in 0 to n-2 generate
            T(i+1) <= tmp(i);
        end generate;
        T(0) <= '0';
    end Behavioral;

```

8.4 Registru de memorare

```
entity register_n is
    generic(n:INTEGER);
    Port (D:in std_logic_vector(n-1 downto 0);
          Clk:in std_logic;
          Rst:in std_logic;
          Q:out std_logic_vector(n-1 downto 0) );
end register_n;

architecture Behavioral of register_n is

begin

process(Clk,Rst)
begin
    if Rst='1' then Q<=(others=>'0');
    elsif Clk'event and Clk='1' then
        Q<=D;
    end if;

end process;
end Behavioral;
```

8.5 Complementul fata de 2

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;

entity twoscompliment is
    generic(n:integer);
    Port (X:in std_logic_vector(n-1 downto 0);
          Y:out std_logic_vector(n-1 downto 0) );
end twoscompliment;

architecture Behavioral of twoscompliment is

signal temp:std_logic_vector(n-1 downto 0);

begin

    temp <= not X;
    Y    <= std_logic_vector(unsigned(temp + 1));

end Behavioral;
```

8.6 Sumator cu propagarea succesivă a transportului

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx leaf cells in this code.
--library UNISIM;
--use UNISIM.VComponents.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);
          Cin:in std_logic;
          S:out std_logic_vector(n-1 downto 0);
          Cout: out std_logic );
end SPT;

architecture Behavioral of SPT is

    signal C:std_logic_vector(1 to n-1);

    component SE is
        Port (X:in std_logic;
              Y:in std_logic;
              Tin:in std_logic;
              Tout:out std_logic;
              S:out std_logic );
    end component;

    begin

        SE_0: SE port map(X(0),Y(0),Cin,C(1),S(0));

        G_1: for i in 1 to n-2 generate
            SE_i: SE port map(X(i), Y(i),C(i),C(i+1),S(i));
        end generate;

        SE_n: SE port map(X(n-1),Y(n-1),C(n-1),Cout,S(n-1));
    end Behavioral;

```

8.7 Generarea de produse parțiale

```

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

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx leaf cells in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity Product_basic_generator is
    generic(n:integer:=32; nr_shiftari:in integer);
    Port (X:in std_logic_vector(n-1 downto 0);
          Y:in std_logic;
          P:out std_logic_vector(2*n-1 downto 0) );
end Product_basic_generator;

architecture Behavioral of Product_basic_generator is

    signal tmp:std_logic_vector(2*n -1 downto 0):=x"0000000000000000";
    signal shift:std_logic_vector(nr_shiftari-1 downto 0);

begin

    shift<=conv_std_logic_vector(0,nr_shiftari);
    tmp(n+nr_shiftari-1 downto 0) <=X & shift when nr_shiftari >0 else
        X ;

    P<=tmp when Y='1' else
        x"0000000000000000" when Y='0';

end Behavioral;

```

```

entity product_generator is
    generic(n:integer:=32);
    Port (X:in std_logic_vector(n-1 downto 0);
          Y:in std_logic_vector(n-1 downto 0);
          P:out my_array(n-1 downto 0) );
end product_generator;

architecture Behavioral of product_generator is

    component Product_basic_generator is
        generic(n:integer:=32; nr_shiftari:in integer);
        Port (X:in std_logic_vector(n-1 downto 0);
              Y:in std_logic;
              P:out std_logic_vector(2*n-1 downto 0) );
    end component;

    type array32_64 is array(0 to n-1) of std_logic_vector(2*n-1 downto 0);
    signal PP:array32_64:=(others=>x"0000000000000000");

begin

    g0: for i in 0 to n-1 generate
        pbgi: Product_basic_generator generic map(n,i) port map(X=>X, Y=>Y(i), P=>PP(i));
    end generate;

    g1: for i in 0 to n-1 generate
        et_i: P(i)<=PP(i);
    end generate;

end Behavioral;

```

8.9 Modulul principal

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

package pkg is
    constant n: integer:=32;
    type my_array is array (natural range <>) of std_logic_vector(2*n-1 downto 0);
end package;

package body pkg is
end package body;

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
library work;
use work.pkg.all;

```

```

entity multiplier is
    Port (X:in std_logic_vector(31 downto 0 );
          Y:in std_logic_vector(31 downto 0);
          Clk:in std_logic;
          Rst:in std_logic;
          P: out std_logic_vector(63 downto 0));
end multiplier;

architecture Behavioral of multiplier is

    component SST is
        generic(n:INTEGER);
        Port (X:in std_logic_vector(n-1 downto 0);
              Y:in std_logic_vector(n-1 downto 0);
              Z:in std_logic_vector(n-1 downto 0);
              S:out std_logic_vector(n-1 downto 0);
              T:out std_logic_vector(n-1 downto 0));
    end component;

    component product_generator is
        generic(n:integer:=32);
        Port (X:in std_logic_vector(n-1 downto 0);
              Y:in std_logic_vector(n-1 downto 0);
              P:out my_array(n-1 downto 0) );
    end component;

    component register_n is
        generic(n:INTEGER);
        Port (D:in std_logic_vector(n-1 downto 0);
              Clk:in std_logic;
              Rst:in std_logic;
              Q:out std_logic_vector(n-1 downto 0) );
    end component;

    component SPT is
        generic(n:INTEGER:=32);
        Port (X:in std_logic_vector(n-1 downto 0);
              Y:in std_logic_vector(n-1 downto 0);
              Cin:in std_logic;
              S:out std_logic_vector(n-1 downto 0);
              Cout: out std_logic );
    end component;

    component twoscompliment is
        generic(n:integer);
        Port (X:in std_logic_vector(n-1 downto 0);
              Y:out std_logic_vector(n-1 downto 0) );
    end component;

    component mpg is
        Port ( en:out STD_LOGIC;
              input:in STD_LOGIC;
              clock: in STD_LOGIC);
    end component;

    constant n:integer:=32;
    constant n_sign:integer:=2;
    signal R11,R12:std_logic_vector(n-1 downto 0);

    signal M: my_array(n-1 downto 0) ;

    type R2_type is array(31 downto 0) of std_logic_vector(2*n-1 downto 0);
    signal R2 : R2_type;

    type R3_type is array(21 downto 0) of std_logic_vector(2*n-1 downto 0);
    signal R3: R3_type;
    signal ST1:R3_type;

```

```

type R4_type is array(14 downto 0) of std_logic_vector(2*n-1 downto 0);
signal R4: R4_type;
signal ST2:R4_type;

type R5_type is array(9 downto 0) of std_logic_vector(2*n-1 downto 0);
signal R5:R5_type;
signal ST3:R5_type;

type R6_type is array(6 downto 0) of std_logic_vector(2*n-1 downto 0);
signal R6:R6_type;
signal ST4:R6_type;

type R7_type is array(4 downto 0) of std_logic_vector(2*n-1 downto 0);
signal R7:R7_type;
signal ST5:R7_type;

type R8_type is array(3 downto 0) of std_logic_vector(2*n-1 downto 0);
signal R8:R8_type;
signal ST6:R8_type;

type R9_type is array(2 downto 0) of std_logic_vector(2*n-1 downto 0);
signal R9:R9_type;
signal ST7:R9_type;

type R10_type is array(1 downto 0) of std_logic_vector(2*n-1 downto 0);
signal R10:R10_type;
signal ST8:R10_type;

signal SPT_Tout:std_logic;

signal X_in:std_logic_vector(31 downto 0);
signal Y_in:std_logic_vector(31 downto 0);
signal X_complemented:std_logic_vector(31 downto 0);
signal Y_complemented:std_logic_vector(31 downto 0);
signal complemented_x:std_logic:='0';
signal complemented_y:std_logic:='0';
signal complemented:std_logic:='0';
signal P_out:std_logic_vector(63 downto 0);
signal P_complemented:std_logic_vector(63 downto 0);

begin

twoc1:twoscompliment generic map(32) port map(X,X_complemented);
twoc2:twoscompliment generic map(32) port map(Y,Y_complemented);

```



```

complemented_x <= '1' when X(31)='1' else '0';

complemented_y <= '1' when Y(31)='1' else '0';

with complemented_x select X_in<= X when '0', X_complemented when others;

with complemented_y select Y_in<=Y when '0', Y_complemented when others;

i11:register_n generic map(n) port map(X_in,Clk,Rst,R11);
i12:register_n generic map(n) port map(Y_in,Clk,Rst,R12);

i1: product_generator generic map(n) port map(X=>R11,Y=>R12,P=>M);

i2: for i in 0 to n-1 generate
    et_i:register_n generic map(2*n) port map(M(i),Clk,Rst,R2(i));
end generate;

i21: for i in 0 to 9 generate
    et_i:SST generic map(2*n) port map(X=>R2(3*i),Y=>R2(3*i+1),Z=>R2(3*i+2),S=>ST1(2*i),T=>ST1(2*i+1));
end generate;

i3: for i in 0 to 19 generate
    et_i: register_n generic map(2*n) port map(ST1(i),Clk,Rst,R3(i));
end generate;

i301: register_n generic map(2*n) port map(R2(30),Clk,Rst,R3(20));
i302: register_n generic map(2*n) port map(R2(31),Clk,Rst,R3(21));

i31: for i in 0 to 6 generate
    et_i: SST generic map(2*n) port map(X=>R3(3*i),Y=>R3(3*i+1),Z=>R3(3*i+2),S=>ST2(2*i),T=>ST2(2*i+1));
end generate;

i4: for i in 0 to 13 generate
    et_i:register_n generic map(2*n) port map(ST2(i),Clk,Rst,R4(i));
end generate;

i401:register_n generic map(2*n) port map(R3(21),Clk,Rst,R4(14));

i41: for i in 0 to 4 generate
    et_i:SST generic map(2*n) port map(X=>R4(3*i),Y=>R4(3*i+1),Z=>R4(3*i+2),S=>ST3(2*i),T=>ST3(2*i+1));
end generate;

i5: for i in 0 to 9 generate
    et_i:register_n generic map(2*n) port map(ST3(i),Clk,Rst,R5(i));
end generate;

i51: for i in 0 to 2 generate
    et_i:SST generic map(2*n) port map(X=>R5(3*i),Y=>R5(3*i+1),Z=>R5(3*i+2),S=>ST4(2*i),T=>ST4(2*i+1));
end generate;

```

```

i6: for i in 0 to 5 generate
    et_i:register_n generic map(2*n) port map(ST4(i),Clk,Rst,R6(i));
end generate;

i601:register_n generic map(2*n) port map(R5(9),Clk,Rst,R6(6));

inst61: for i in 0 to 1 generate
    et_i:SST generic map(2*n) port map(X=>R6(3*i),Y=>R6(3*i+1),Z=>R6(3*i+2),S=>ST5(2*i),T=>ST5(2*i+1));
end generate;

i7: for i in 0 to 3 generate
    et_i:register_n generic map(2*n) port map(ST5(i),Clk,Rst,R7(i));
end generate;

i701:register_n generic map(2*n) port map(R6(6),Clk,Rst,R7(4));

i71:SST generic map(2*n) port map(X=>R7(0),Y=>R7(1),Z=>R7(2),S=>ST6(0),T=>ST6(1));

i801:register_n generic map(2*n) port map(ST6(0),Clk,Rst,R8(0));
i802:register_n generic map(2*n) port map(ST6(1),Clk,Rst,R8(1));
i803:register_n generic map(2*n) port map(R7(3),Clk,Rst,R8(2));
i804:register_n generic map(2*n) port map(R7(4),Clk,Rst,R8(3));

i81:SST generic map(2*n) port map(X=>R8(0),Y=>R8(1),Z=>R8(2),S=>ST7(0),T=>ST7(1));

i901:register_n generic map(2*n) port map(ST7(0),Clk,Rst,R9(0));
i902:register_n generic map(2*n) port map(ST7(1),Clk,Rst,R9(1));
i903:register_n generic map(2*n) port map(R8(3),Clk,Rst,R9(2));

i91:SST generic map(2*n) port map(X=>R9(0),Y=>R9(1),Z=>R9(2),S=>ST8(0),T=>ST8(1));

i101:register_n generic map(2*n) port map(ST8(0),Clk,Rst,R10(0));
i102:register_n generic map(2*n) port map(ST8(1),Clk,Rst,R10(1));

i10:SPT generic map(2*n) port map(X=>R10(0),Y=>R10(1),Cin=>'0',S=>P_out,Cout=>SPT_Tout);

twoc3:twoscomplement generic map(64) port map(P_out,P_complemented);

end Behavioral;

```

