

Proiectarea unei unități aritmetico-logice

Adela Iosif
Grupa 30233



UNIVERSITATEA TEHNICĂ
DIN CLUJ-NAPOCA



Cuprins

1. Introducere	2
2. Studiu bibliografic	3
3. Design și analiză	4
4. Implementare	10
5. Testare	11
6. Concluzii	16

1. Introducere

1.1. Context

Unitatea aritmetico-logică (UAL) este o parte importantă a unității centrale de procesare (CPU), a unității de procesare grafică (GPU) sau a altor dispozitive electronice (chiar și a celor mai simple microprocesoare). De aceea, UAL trebuie să fie capabilă să calculeze o mulțime de operații, de la cele mai simple (adunare, scădere, și logic, sau logic, etc.), până la altele mai complexe (operații în virgulă mobilă, operații trigonometrice, încărcare/descărcare date, etc.). Mai mult, odată cu complexitatea unității aritmetico-logice, crește spațiul ocupat de aceasta în procesor, prețul, dar și puterea disipată.

1.2. Motivație

Unitatea proiectată de mine va avea o funcționalitate extinsă, efectuând următoarele operații: adunare, scădere în C2, incrementare, decrementare, și logic, sau logic, nu logic, rotație stânga și rotație dreapta, dar voi avea și un circuit suplimentar pentru înmulțire și împărțire. Astfel, UAL poate acoperi o gamă largă de necesități, de la calcule matematice simple, la manipularea datelor logice. În plus, având în vedere că se ocupă cu aceste operații de bază, poate oferi un nivel ridicat de eficiență și performanță, accelerând execuția programelor. Tocmai de aceea, pentru aplicații care necesită prelucrare în timp real sau prelucrarea datelor la viteze ridicate, aceste operații simple sunt esențiale și eficiente. Totodată, UAL proiectată aici poate fi considerată optimizată din punctul de vedere al costurilor și al consumului de energie, deoarece nu are funcționalități complexe.

1.3. Obiective

Obiectivele acestui proiect sunt de a implementa într-un mod cât mai eficient operațiile de bază cerute: adunarea, scăderea în C2, incrementarea, decrementarea, și logic, sau logic, nu logic, rotația stânga și rotația dreapta, înmulțirea și împărțirea. De asemenea, încerc o proiectare cât mai flexibilă, pentru a putea fi ușor de extins sau actualizat în viitor. După finalizarea

implementării, unitatea aritmetico-logică ar trebui să fie capabilă să execute operațiile menționate, pe baza datelor introduse de utilizator. Testarea va fi făcută prin simularea în Vivado (eventual și pe plăcuța Basys 3 – pentru înmulțire și adunare).

2. Studiu bibliografic

Printre resursele bibliografice utile găsite pentru acest proiect se numără:

- Lucia Văcariu, Octavian Creț, *Probleme de proiectare logică – a sistemelor numerice-*, U.T.PRESS, Cluj-Napoca, 2013, pp. 32, 90-95
- Baruch Zoltan Francisc, *Aplicații de proiectare digitală cu circuite FPGA*, Editura Mega, Cluj-Napoca, 2020, pp. 129-141
- Department of Electronics and Telecommunication K.C. College of Engineering & Management Studies & Research, Kopri, Thane, India, *Implementation of ALU on FPGA*

Scăderea se cere a fi implementată în complement față de 2. Aceasta este o metodă de reprezentare binară atât a numerelor pozitive, cât și a celor negative. Operațiile aritmetice sunt ușor de realizat cu sumatoare clasice, dar necesită un bit de stocare suplimentar chiar și în cazul în care se reprezintă numai numere pozitive. Câteva exemple de reprezentare a numerelor în complement față de 2:

Număr binar	Întregi fără semn	Complementul față de 2
000	0	0
001	1	1
010	2	2
011	3	3
100	4	-4
101	5	-3
110	6	-2
111	7	-1

Intervalul de reprezentare (number range) pentru gama de 2^n numere reprezentabile pe n biți este $[-2^{(n-1)} - 2^{(n-1)} - 1]$ – numere negative: de la -2^{n-1} până la -1 , 0 și numere pozitive: de la 1 la $2^{n-1} - 1$.

Pentru numerele întregi pozitive: pe cei $n-1$ biți se trece reprezentarea în baza doi a valorii absolute a numărului, bitul cel mai din stânga fiind bitul de semn – mereu 0 pentru numere pozitive.

Pentru numerele negative: reprezentarea unui astfel de număr în C2 este valoarea $2^n - V$, V reprezentând valoarea absolută a numărului reprezentat.

La adunare, există o regulă de overflow: dacă se adună două numere cu același semn (ambele sunt fie pozitive, fie negative), depășirea are loc doar dacă rezultatul este de semn opus.

0101 = 5
+0110 = 4
1001 = overflow
- Se întâmplă pentru că 9 este în afara reprezentării pe 4 biți în C2
 $[-2^{4-1} = -8, 2^{4-1} - 1 = 7]$

1001 = -7
+1010 = -6
10011 = overflow
- Se întâmplă pentru că -13 este în afara reprezentării pe 4 biți în C2
 $[-2^{4-1} = -8, 2^{4-1} - 1 = 7]$

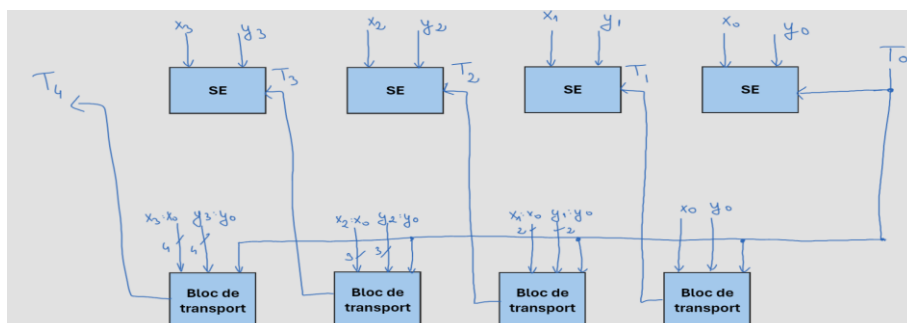
La scădere: pentru a scădea numărul b din a : $(a-b) = a+(-b) = a + (\text{C2 al lui } b)$.

Ex: $2-3=-1$
 $2=0010$, $3=0011$, C2 lui 3=1101
 $2+(-3)=0010$ (2 în binar)
 $\begin{array}{r} -1101 \text{ (-3)} \\ 0010 \\ \hline 1111 \text{ (-1)} \end{array}$

3. Design și analiză

- **Adunare**

Pentru adunarea celor două numere pe 4 biți am ales să folosesc sumatorul cu anticiparea transportului (carry lookahead adder), deoarece acesta crește viteza operației de adunare prin reducerea timpului necesar pentru generarea semnalelor de transport. Astfel, intrarea de transport necesară pentru un etaj este generată în mod direct, utilizând semnale de la toate etajele precedente, în loc de a se aștepta propagarea lentă a transportului de la un etaj la altul.



O reprezentare inițială a acestui sumator surprinde blocurile de transport ca generatoare de intrări de transport pentru sumatoarele elementare. Expresia booleană pentru un bloc de transport este: $T_{i+1} = x_i y_i + (x_i + y_i) T_i$

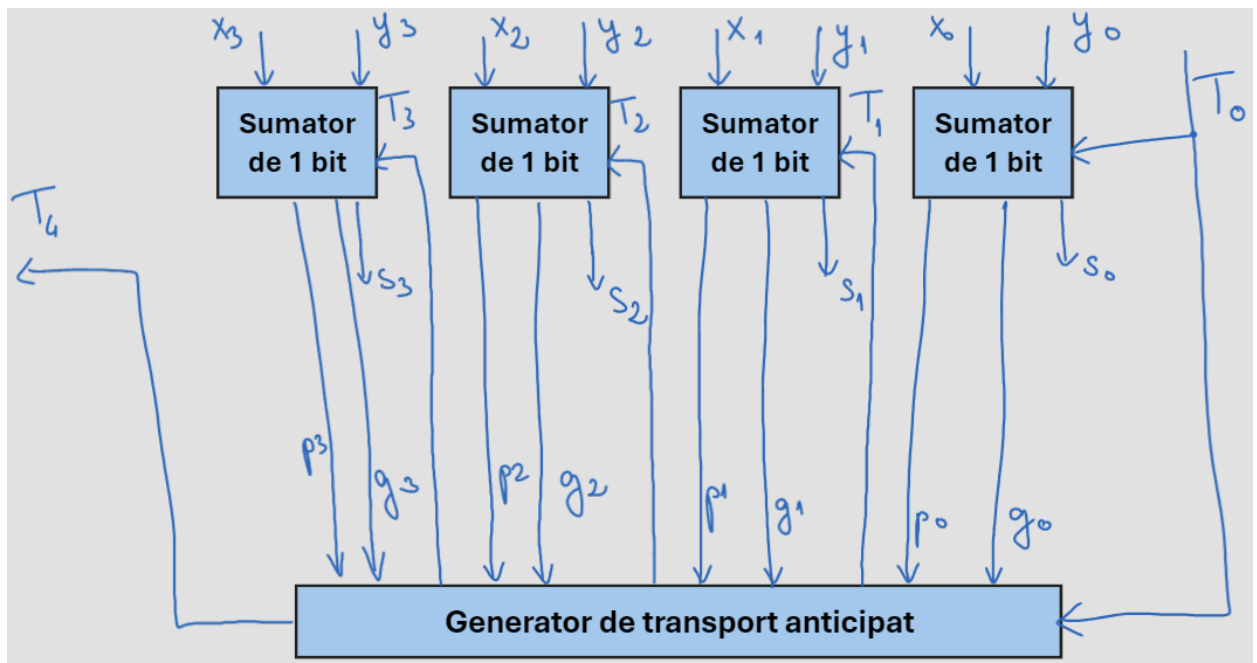
Pentru simplificarea expresiilor semnalelor de transport se introduc două funcții: g – pentru **g**enerarea transportului de un bloc de transport, p – pentru **p**ropagarea intrării de transport la ieșirea de transport a blocului.

$$g_i = x_i y_i \quad p_i = x_i + y_i$$

De aceea, ecuația booleană pentru semnalele de transport devine:

$$T_{i+1} = g_i + p_i T_i$$

Fiecare sumator de un bit produce semnalele de propagare a transportului (p) și de generare a transportului (g) în locul ieșirilor de transport. Generatorul de transport anticipat convertește cele patru seturi de semnale p, g în intrările de transport necesare pentru cele patru sumatoare. Fiecare sumator de un bit produce și bitul corespunzător al sumei, în același mod ca și un sumator elementar.

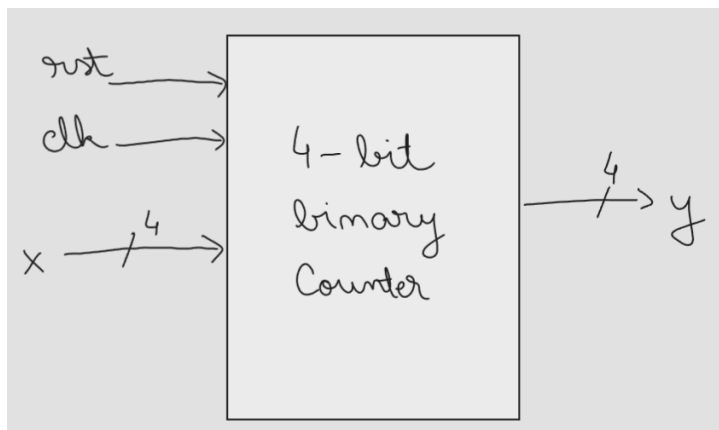


- **Scădere**

Algoritmul folosit la scădere utilizează sumatorul deja construit, dar y se înlocuiește cu complementul său față de 2.

- **Incrementare**

Pentru incrementare folosesc un numărător simplu pe 4 biți, cu semnal de ceas și reset.



Astfel, atunci când semnalul de reset este activ (1), ieșirea se va reseta la valoarea de inițială, altfel se incrementează la fiecare pas (când reset este 0) pe frontul crescător.

Rst	Clk	y(3:0)
1	—	x(3:0)
0	↑	x(3:0) + 1

• Decrementare

La fel ca și la incrementare, decrementarea folosește un numărător cu semnal de ceas și reset, dar numărătorul acesta va fi reversibil.

Rst	Clk	y(3:0)
1	—	x(3:0)
0	↑	x(3:0) - 1

• Și logic

Această operație are următoarea interpretare:

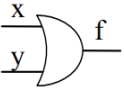
- Dacă cel puțin una din intrări se află în 0 logic, atunci ieșirea va fi 0 logic
- Dacă ambele intrări sunt în 1 logic, atunci ieșirea va fi 1 logic

Simbolul porții ȘI:	Tabelul de adevăr	Ecuția booleană:															
	<table> <tr> <th>x</th><th>y</th><th>f</th></tr> <tr> <td>0</td><td>0</td><td>0</td></tr> <tr> <td>0</td><td>1</td><td>0</td></tr> <tr> <td>1</td><td>0</td><td>0</td></tr> <tr> <td>1</td><td>1</td><td>1</td></tr> </table>	x	y	f	0	0	0	0	1	0	1	0	0	1	1	1	$f = x \cdot y$
x	y	f															
0	0	0															
0	1	0															
1	0	0															
1	1	1															

- **Sau logic**

Această operație are următoarea interpretare:

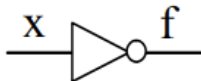
- Dacă cel puțin una din intrări este adevărată, ieșirea este adevărată (1 logic)
- Dacă ambele intrări sunt false, ieșirea este falsă (0 logic)

Simbolul porții SAU:	Tabelul de adevăr	Ecuția booleană:															
	<table> <tr> <th>y</th><th>x</th><th>f</th></tr> <tr> <td>0</td><td>0</td><td>0</td></tr> <tr> <td>0</td><td>1</td><td>1</td></tr> <tr> <td>1</td><td>0</td><td>1</td></tr> <tr> <td>1</td><td>1</td><td>1</td></tr> </table>	y	x	f	0	0	0	0	1	1	1	0	1	1	1	1	$f = x + y$
y	x	f															
0	0	0															
0	1	1															
1	0	1															
1	1	1															

- **Nu logic**

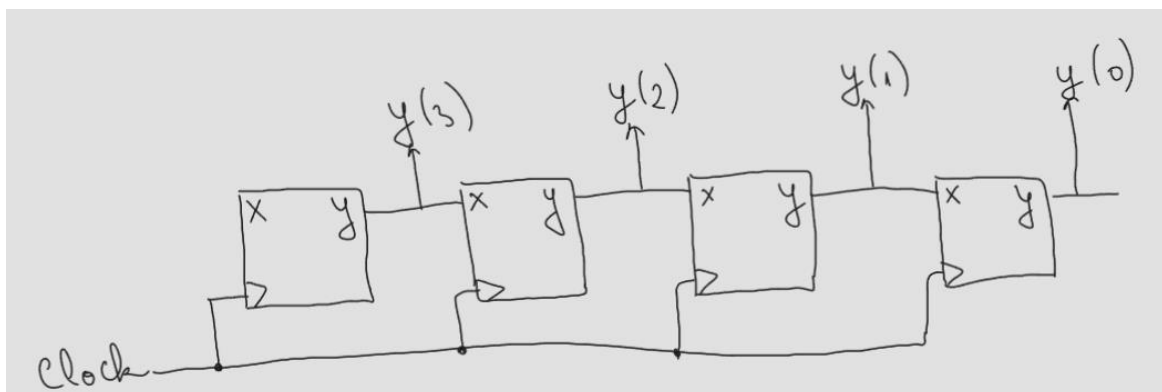
Această operație are următoarea interpretare:

- Dacă intrarea este falsă (0 logic), atunci ieșirea este adevărată (1 logic)
- Dacă intrarea este adevărată, atunci ieșirea este falsă

Simbolul porții NU:	Tabelul de adevăr	Ecuția booleană:						
	<table><tr><th>x</th><th>f</th></tr><tr><td>0</td><td>1</td></tr><tr><td>1</td><td>0</td></tr></table>	x	f	0	1	1	0	$f = \overline{x}$
x	f							
0	1							
1	0							

- **Rotație stânga și dreapta**

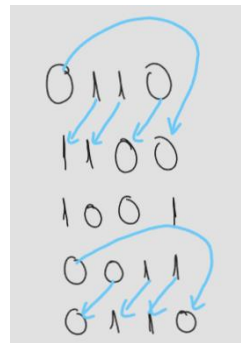
Pentru rotațiile stânga și dreapta folosesc registre de deplasare (câte unul separat pentru fiecare direcție).



Rotirea dreapta:

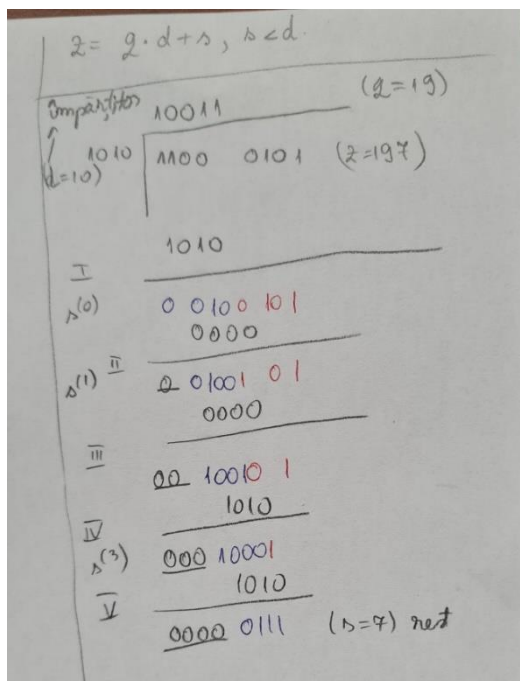


Rotirea stânga:

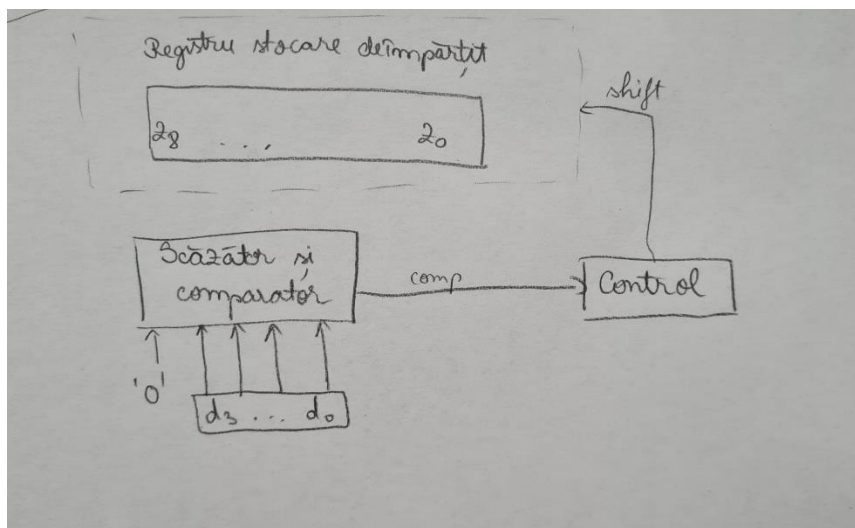


• Împărțire

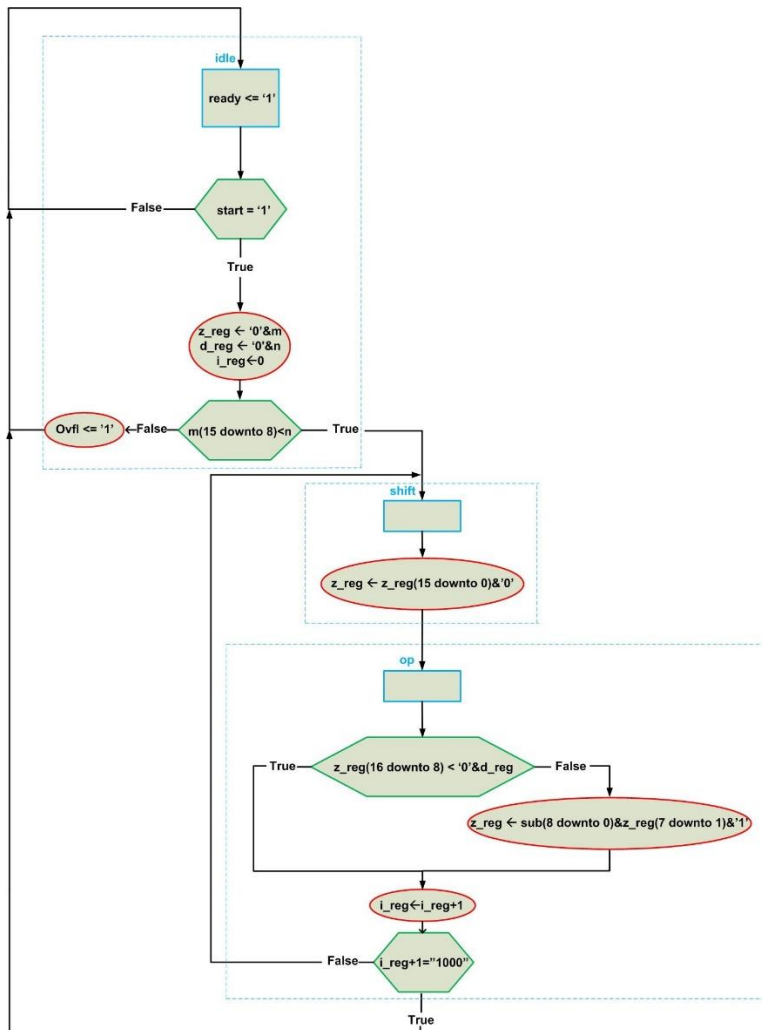
Algoritm:



Schema bloc simplificată:

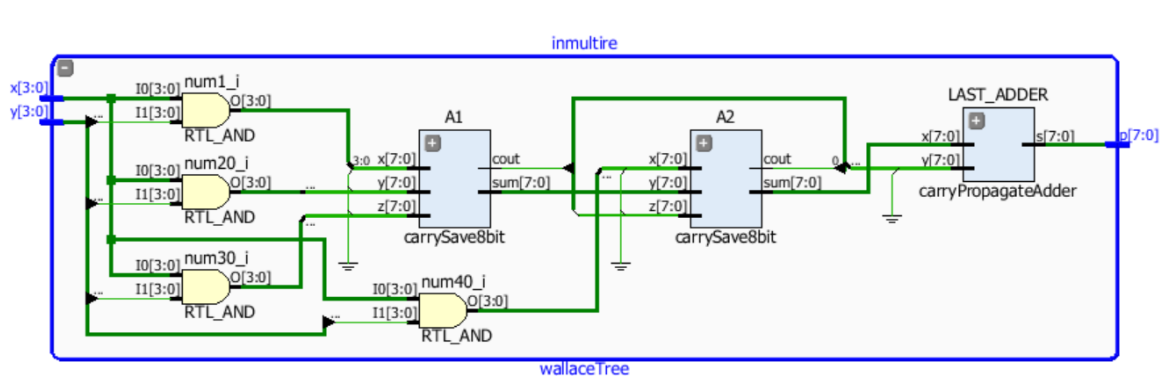


Împărțirea unui nr. pe 16 biți la unul pe 8:

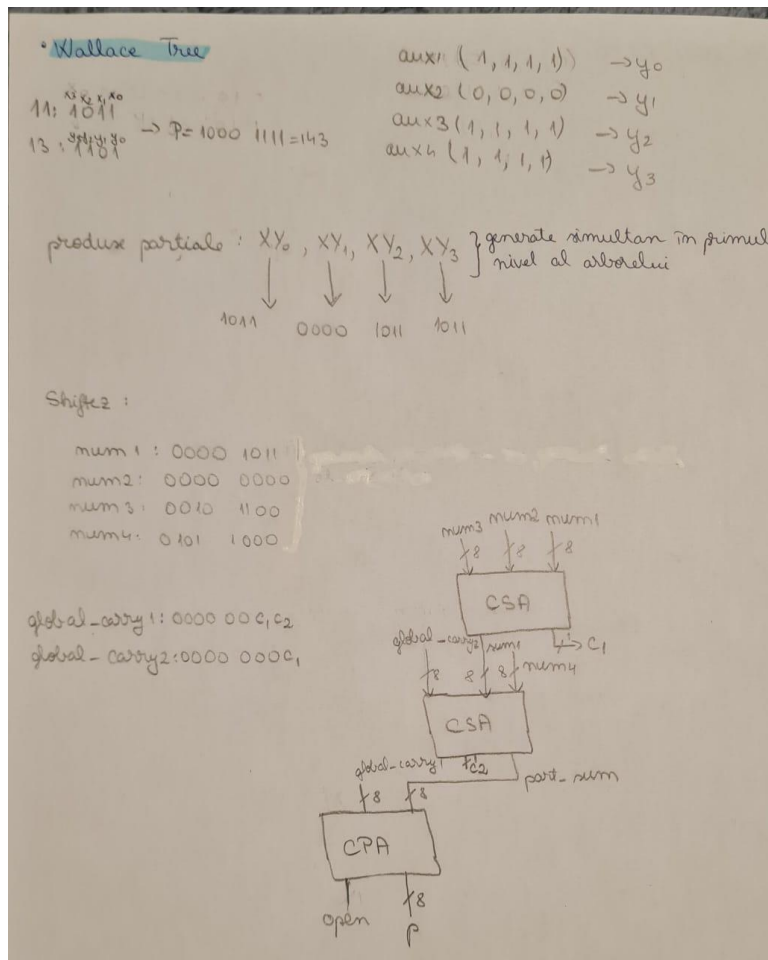


• Înmulțire

Prin utilizarea unui arbore în înmulțirea a două numere de n biți se poate reduce timpul de execuție la $O(\log n)$, față de timpul obișnuit în care se efectuează această operație (prin adunarea a n produse parțiale): $O(n)$.



Nu folosesc semnal de ceas pentru realizarea acestui algoritm de multiplicare, dar după ce trec la următorul bit din înmulțitor, fac o shiftare a biților manuală. Câteva explicații suplimentare se găsesc mai jos:



4. Implementare

Entitatea principală are la bază un multiplexor care asigură selecția operației dorite de utilizator. De aceea, structura ei se bazează pe cele două numere pe 4 biți care se introduc în testbench și pe alte 9 semnale de selecție care arată ce operație se dorește a fi efectuată (și, sau, nu, adunare, scădere, rotație stânga, rotație dreapta, incrementare, decrementare). Așadar, când semnalul de selecție corespunzător unei operații este activ, pe ieșire se pot observa rezultatele corespunzătoare operației alese pe ciclul de ceas respectiv.

```

process(x, y, sel_si, sel_sau, sel_not, sel_increm, sel_decrem,
sel_adunare, sel_scadere, sel_rstanga, sel_rdreapta, clk)
begin
    iesireSi <= x and y;
    iesireSau <= x or y;
    iesireNot <= not x;
    if rising_edge(clk) then --si logic
        if sel_si = '1' then
            iesire(3 downto 0) <= iesireSi;
        elsif sel_sau = '1' then --sau logic
            iesire(3 downto 0) <= iesireSau;
        elsif sel_not = '1' then --nu logic
            iesire(3 downto 0) <= iesireNot;
        elsif sel_increm = '1' then --incrementare
            iesire(3 downto 0) <= iesireIncrem;
        elsif sel_decrem = '1' then --decremenetare
            iesire(3 downto 0) <= iesireDecrem;
        elsif sel_adunare = '1' then --adunare
            iesire(3 downto 0) <= sSumator;
            coutS <= coutSumator;
        elsif sel_scadere = '1' then --scadere in c2
            iesire(3 downto 0) <= outScadere;
            coutS <= coutScadere;
        elsif sel_rstanga = '1' then --rotatie stanga
            iesire(3 downto 0) <= iesireShiftL;
        elsif sel_rdreapta = '1' then --rotatie dreapta
            iesire(3 downto 0) <= iesireShiftR;
        else
            iesire(3 downto 0) <= (others => 'X');
        end if;
    end if;
end process;

```

Mai mult, pentru înmulțire și împărțire se folosesc circuite separate de unitatea creată.

5. Testare

Având în vedere că am atât o unitate aritmetico-logică care efectuează 9 operații, dar și 2 circuite suplimentare pentru înmulțire și împărțire, am testarea este făcută cu ajutorul testbench-ului. Astfel, primul testbench este pentru UAL: mă folosesc de semnalele de selecție corespunzătoare fiecărei operații pentru a le verifica funcționalitatea pe rând.

```

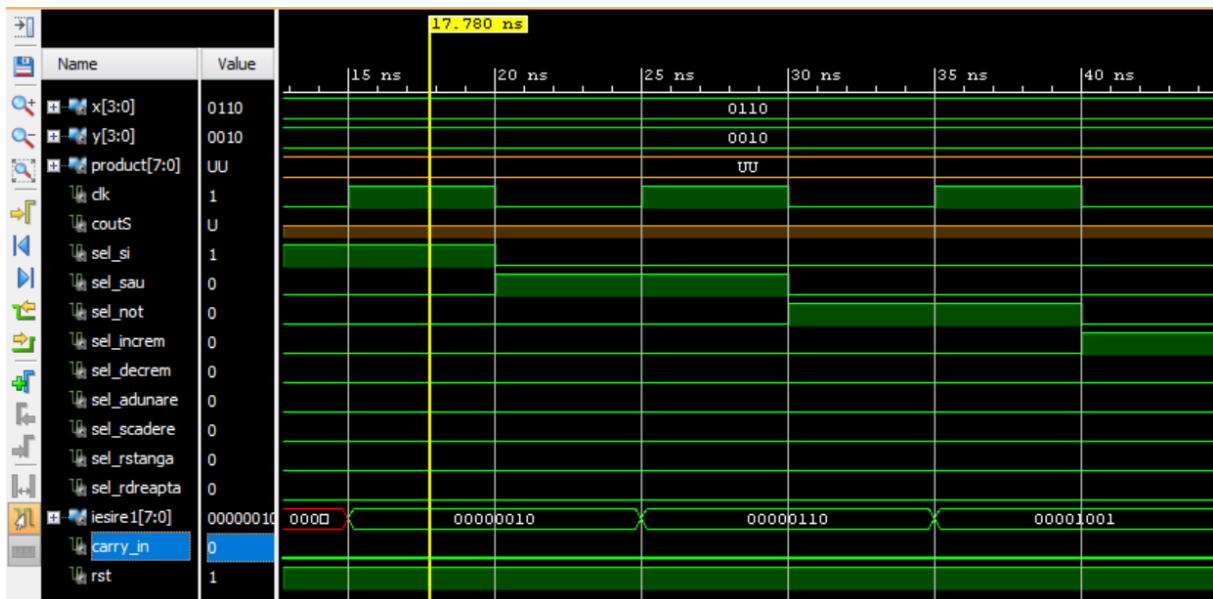
x <= "0110"; --6
y <= "0010"; --2
clk <= not clk after 5ns;
rst <= '0' after 45 ns, '1' after 70 ns, '0' after 75 ns;

sel_si <= '1' after 10ns, '0' after 20 ns;
sel_sau <= '1' after 20 ns, '0' after 30 ns;
sel_not <= '1' after 30 ns, '0' after 40 ns;
sel_increm <= '1' after 40 ns, '0' after 50 ns, '1' after 55 ns, '0' after 70 ns;
sel_decrem <= '1' after 70 ns, '0' after 80 ns;
sel_adunare <= '1' after 100 ns, '0' after 120 ns, '1' after 125 ns, '0' after 140 ns;
sel_scadere <= '1' after 140ns, '0' after 150 ns;
sel_rstanga <= '1' after 150ns, '0' after 180 ns;
sel_rdreapta <= '1' after 180ns, '0' after 190 ns;

test: muxSelectie port map(x => x, y => y, clk => clk, rst => rst, carry_in => carry_in,
sel_si => sel_si, sel_sau => sel_sau, sel_not => sel_not, sel_adunare => sel_adunare,
sel_scadere => sel_scadere, sel_increm => sel_increm, sel_decrem => sel_decrem,
sel_rstanga => sel_rstanga, sel_rdreapta => sel_rdreapta, coutS => coutS, iesire => iesire1);

```

- Testarea operațiilor: și, sau, nu



Putem observa că atunci când semnalul corespunzător operației este activ (dacă sel_si, sel_sau, sel_not sunt 1) și suntem pe frontul crescător, operația se execută corect:

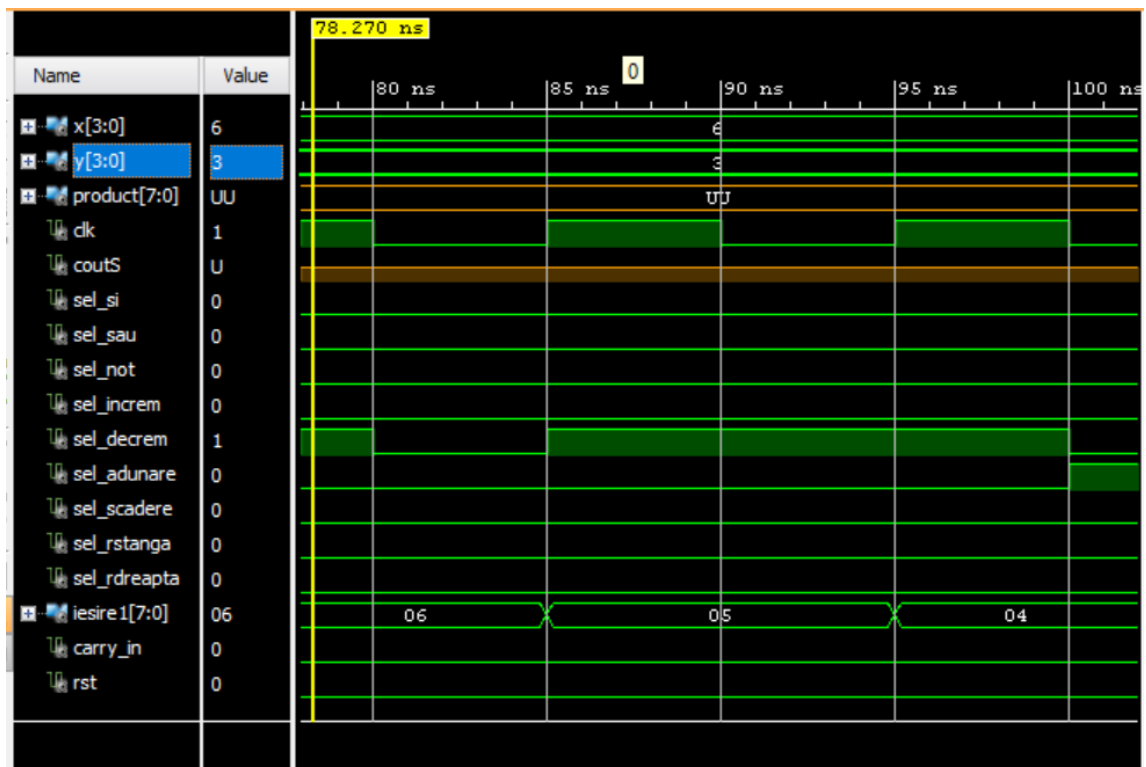
- pentru și: x ȘI y
- pentru sau: x SAU y
- pentru nu: NOT x

- Testarea operațiilor de incrementare și decrementare

Incrementare: dacă semnalul sel_increm este 1 și suntem pe front crescător, se face incrementarea lui x.

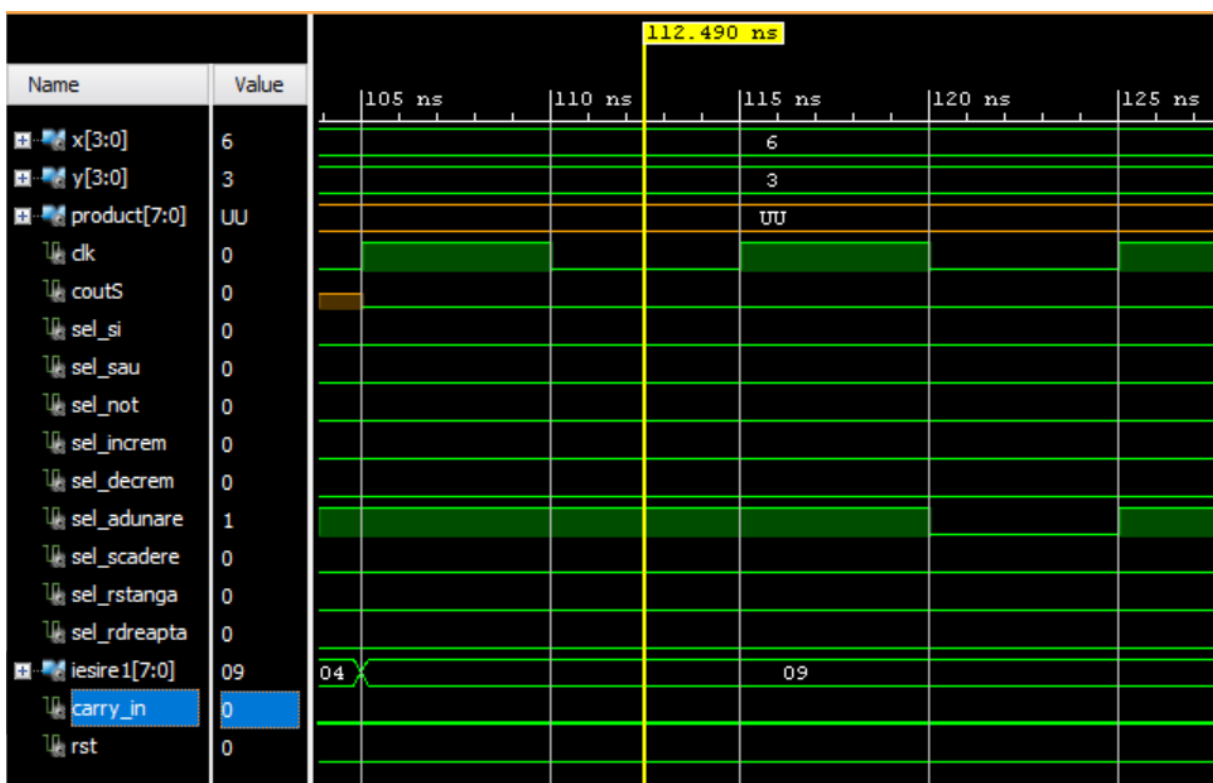


Decrementare: dacă semnalul sel_decrem este 1 și suntem pe front crescător, se face decrementarea lui x.



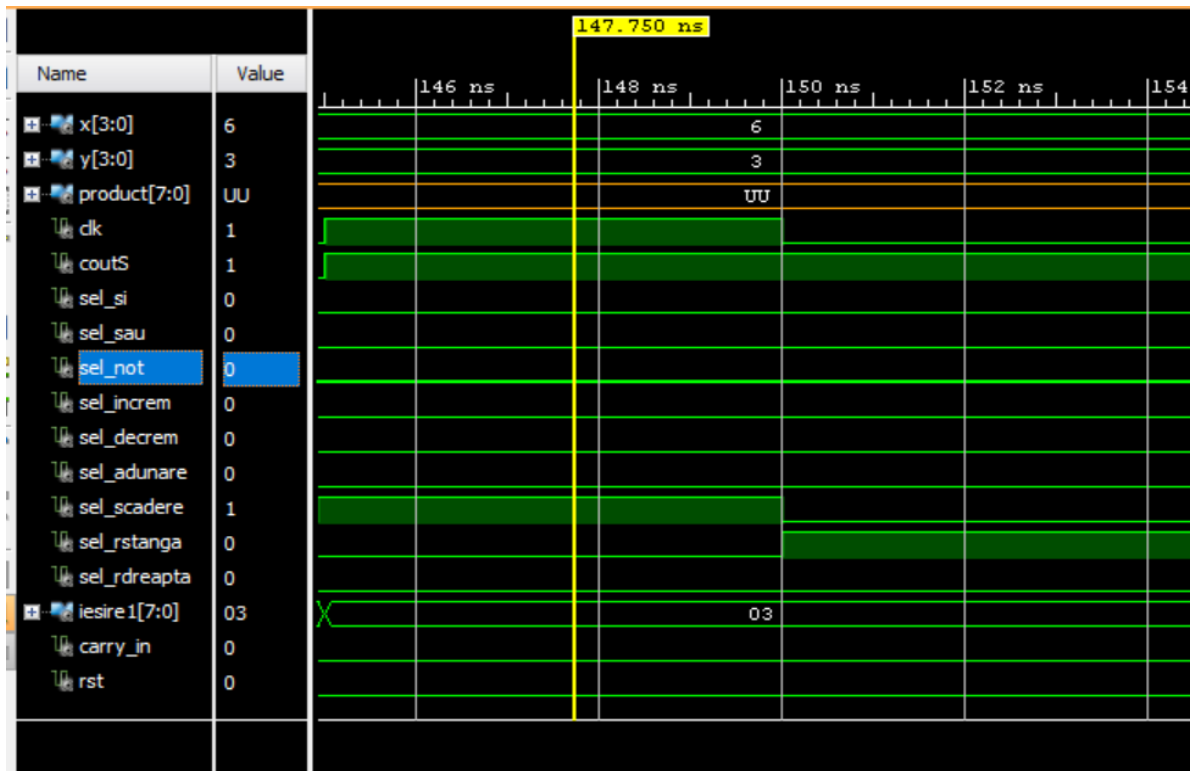
- Testarea operațiilor de adunare și scădere

Adunare: este activ semnalul sel_adunare și frontul este crescător, deci se adună x și y



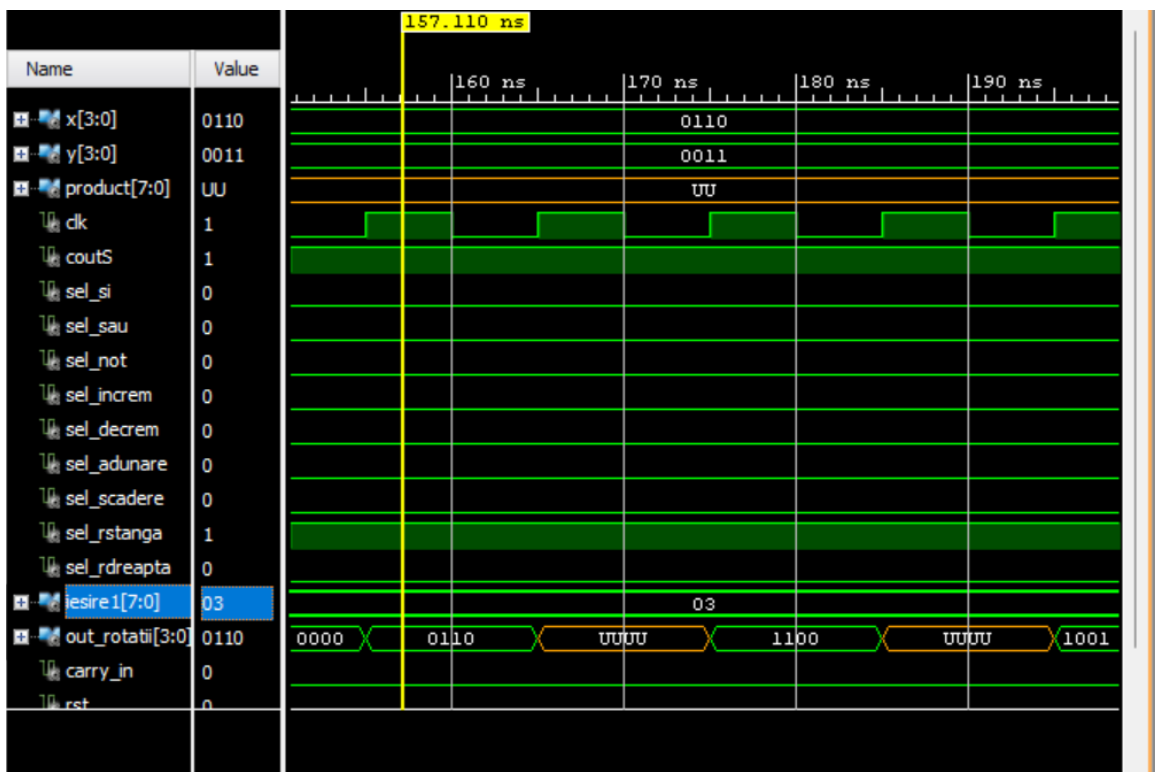
Scădere: este activ semnalul sel_scadere și frontul este crescător, deci se va face operația

x-y

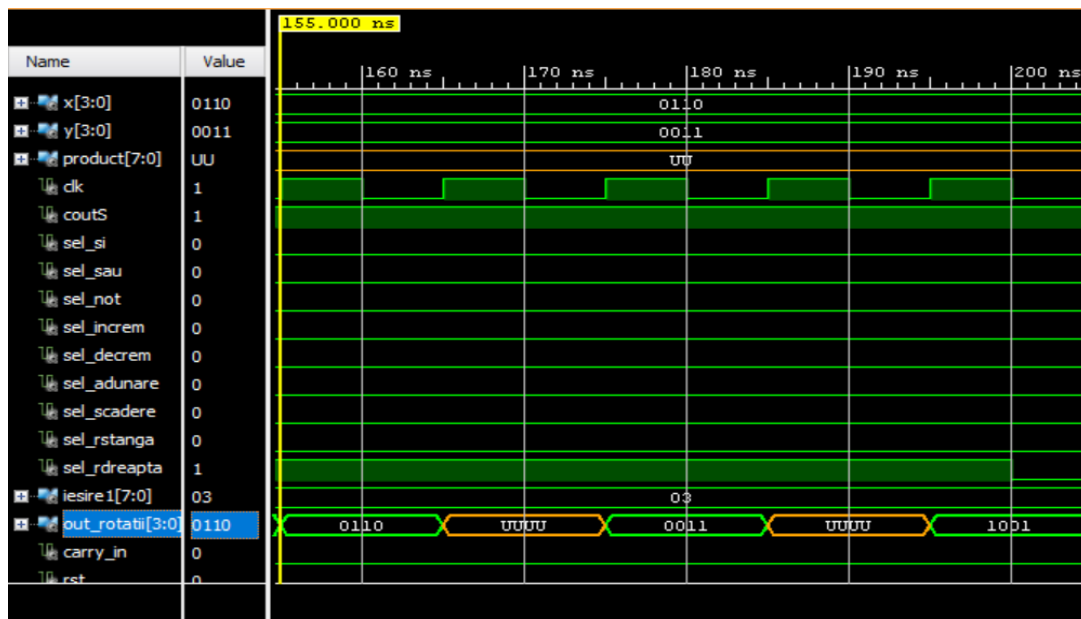


- Testarea operațiilor de rotație

Rotație stânga: semnalul sel_rstanga este pe 1 și frontul este crescător, deci se va face o rotație cu o poziție la stânga a biților din x, rezultatul observându-se pe ieșirea out_rotatii

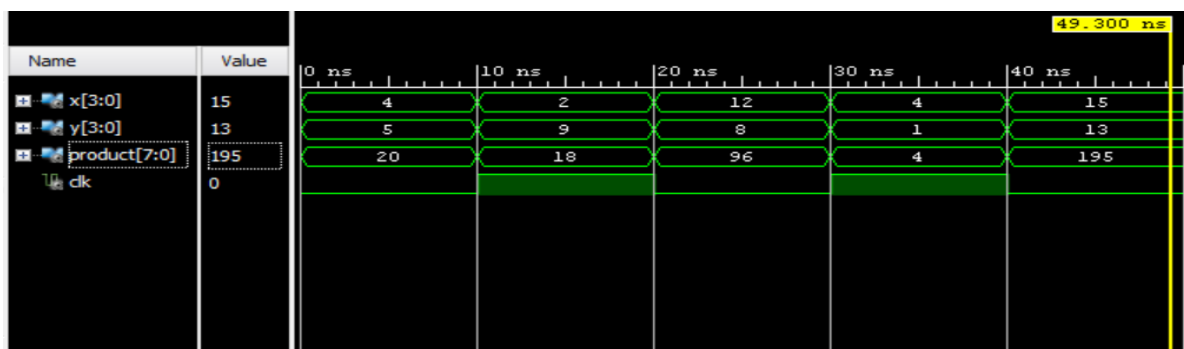


Rotație dreapta: semnalul sel_rdreapta este pe 1 și frontul este crescător, deci se va face o rotație cu o poziție la dreapta a biților din x, rezultatul observându-se pe ieșirea out_rotatii



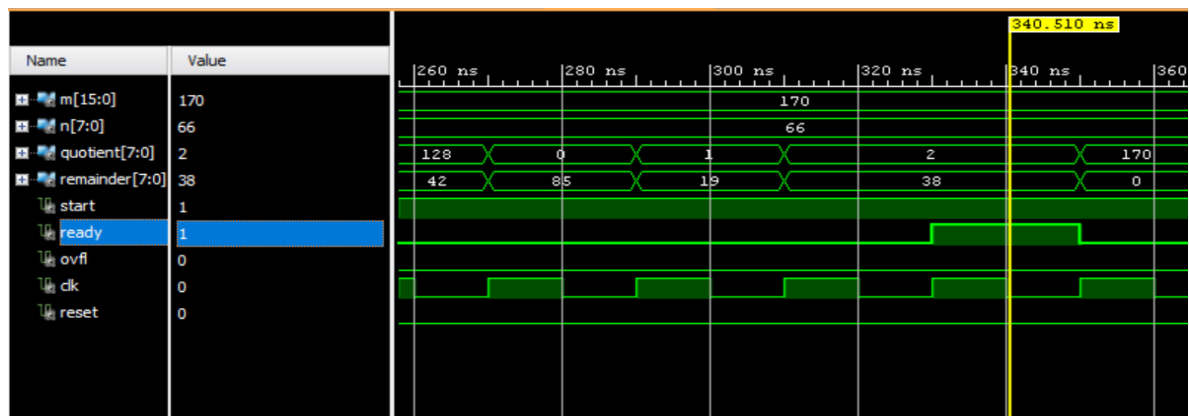
- Testarea înmulțirii

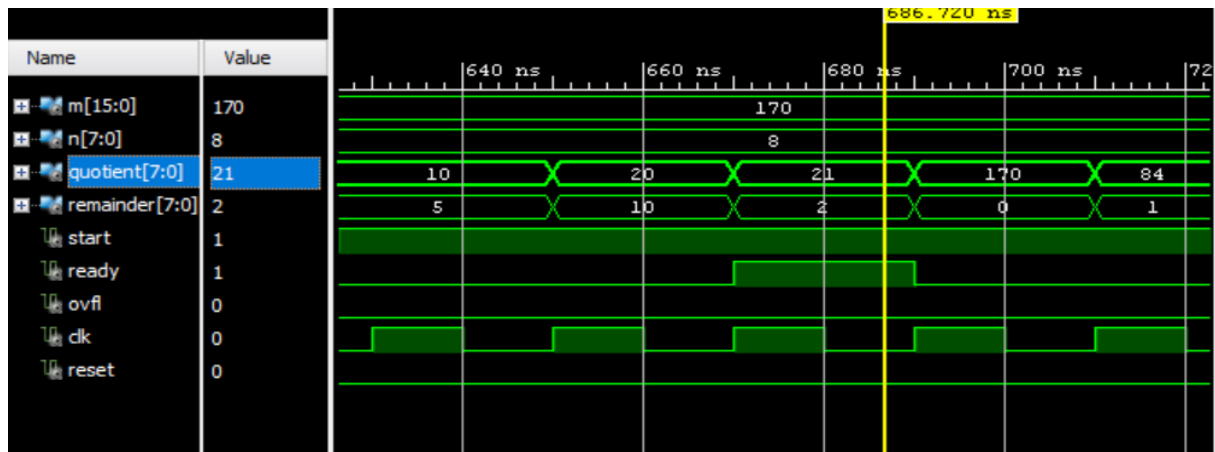
Am făcut un testbench separat pentru unitatea de înmulțire, iar aceasta funcționează conform implementării, pentru înmulțirea oricăror întregi fără semn pe 4 biți.



- Testarea împărțirii

Când flag-ul de ready este pe 1, înseamnă că am obținut rezultatele împărțirii (cât și rest).





6. Concluzii

Obiectivul proiectului a fost design-ul, implementarea și testarea unei unități aritmetico-logice, cu două circuite suplimentare pentru înmulțire și împărțire. Operațiunile au fost realizate pe numere întregi fără semn. Datorită acestui proiect am reușit să cunosc și mai în detaliu limbajul de proiectare hardware, VHDL, dar am și învățat să organizez un proiect când trebuie să efectueze mai multe operații, separate chiar în unități diferite.