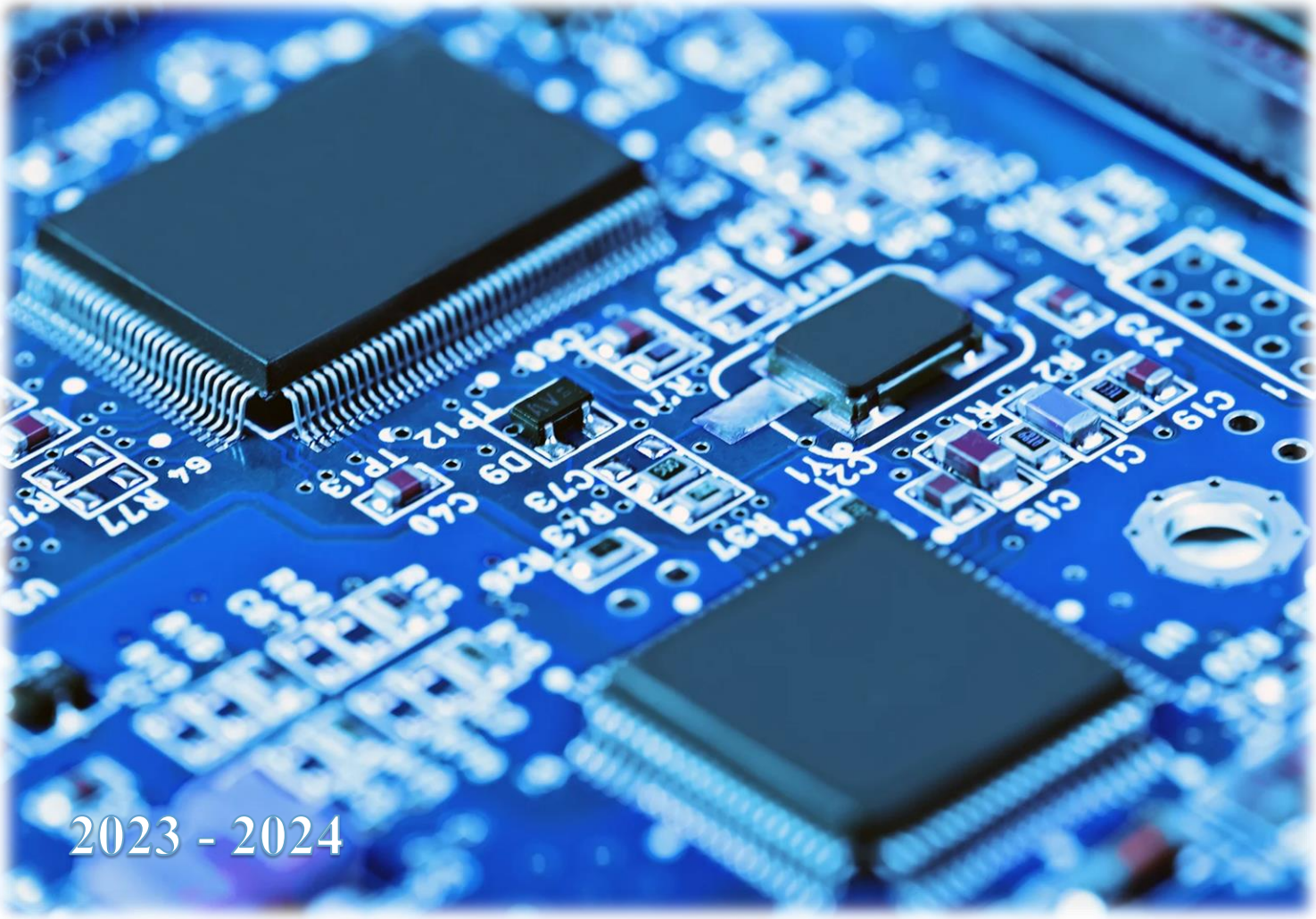
PROIECTAREA UNEI UNITĂȚI ARITMETICO-LOGICE (UAL)

Documentație



UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Marcu Ariana-Mălina

Grupa 30232/2

Cuprins:

1.1 Context

Scopul acestei unități aritmetico-logice (în acest proiect fiind pe 32 de biți) este de a proiecta, implementa și testa o serie de operații complexe pentru manipularea numerelor. Aceste operații depășesc funcționalitatea operațiilor de bază și adaugă un nivel avansat de performanță și flexibilitate. Aceste operații includ:

- *Adunare și Scădere în Complement față de 2*: ALU este capabilă de adunare și scădere folosind reprezentarea în complement față de 2, asigurând precizie și corectitudine pentru operațiile aritmetice.
- *Incrementare și Decrementare*: poate efectua operații de incrementare și decrementare pentru optimizarea calculelor în cadrul proceselor matematice complexe.
- *Operații Logice (ȘI, SAU, NU):* susține operațiile logice, oferind putere de procesare pentru manipularea datelor în baza logicii.

- *Negare*: Funcționalitatea de negare permite transformarea datelor în opusul lor, oferind astfel versatilitate în operațiile logice.

- *Rotație Stânga și Dreapta*: poate efectua rotații în stânga și dreapta asupra datelor, permițând manipularea și deplasarea biților într-un mod eficient.

- *Utilizarea Acumulatorului*: ALU folosește un acumulator pentru a stoca un operand de intrare și pentru a furniza rezultatul final al operațiilor. Acest acumulator servește ca registru temporar esențial în procesarea datelor.

- *Operații de Înmulțire și Împărțire Suplimentare*: În plus față de operațiile de bază, ALU dispune de un circuit suplimentar pentru efectuarea operațiilor de înmulțire și împărțire, oferind o funcționalitate avansată și complexă.

1.2 Specificații

Dispozitivul va fi simulat în mediul de dezvoltare furnizat de Vivado (și apoi va putea fi programat și pe o placă Basys3). ALU poate fi folosită într-o varietate de aplicații, de la calculatoare simple pentru operații de bază, la integrarea în procesoare sau microprocesoare pentru a accelera și optimiza operațiile complexe de calcul. Acest dispozitiv poate funcționa independent sau ca parte a unui sistem mai mare, primind comenzi pentru efectuarea diferitelor operații complexe menționate mai sus.

1.3 Obiective

Crearea unui ALU este un aspect fundamental în proiectarea și dezvoltarea unui procesor sau unitate centrală de prelucrare (CPU). Scopul principal al unui ALU este să efectueze operațiile aritmetice și logice necesare pentru procesarea datelor într-un calculator. Ca și obiectiv general avem proiectarea și implementarea eficientă a unei unități care să calculeze multe operații aritmetice și logice de toate tipurile. Obiectivele specifice ar fi implementarea corectă a acestora, verificarea funcționalității prin testare amănunțită, optimizarea proiectului și elaborarea unei documentații ușor de înțeles.

2. Studiu bibliografic

Unitatea aritmetică-logică, acronim UAL (eng. Arithmetic-Logic Unit – ALU), este o componentă de bază a unui procesor, fiind acea parte din procesor care execută operațiile aritmetice și logice (după cum sugerează și numele), făcând parte din gama circuitelor combinaționale.

După cum este prezentat în carte și din research-ul pe care am reușit să îl fac, în mod similar cu abordarea pentru sumator, voi porni mai întâi cu o UAL care să funcționeze pe operanzi de 1 bit. Unitatea UAL pe 1 bit se va proiecta inițial pentru operațiile de adunare, ȘI și SAU, după care, exploatând avantajele complementului față de 2, se va extinde pentru scădere.

Pentru cele 3 operații, circuitele necesare sunt: sumator complet pe 1 bit, poarta ȘI și poarta SAU. Problema este de a selecta ce se trimite ca rezultat la ieșirea UAL. Așadar, se va folosi un multiplexor, unde semnalul de selecție alege operația al cărei rezultat este dorit la ieșirea UAL. Semnalul de control *Op* va selecta ce rezultat să apară pe ieșirea UAL: 0 pentru adunare, 1 pentru ȘI, 2 pentru SAU, 3 rezervat pentru adăugarea unei noi eventuale operații.

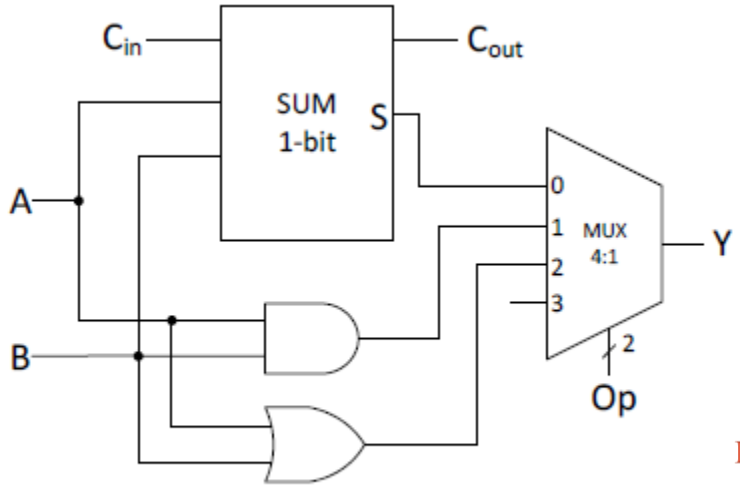


Figura 1

Diagrama pentru UAL 1 bit, cu operații de adunare, ȘI logic și SAU Logic. Ieșirea este Y, iar semnalul de selecție pentru UAL este *Op* (Operație)

Deci, pentru a obține o UAL pe *n* biți ar trebui înlănțuite *n* UAL pe 1 bit, notate ca UAL_{n-1}, …., UAL_1, UAL_0, prin legarea corespunzătoare a semnalelor de transport.

În cazul nostru, vom avea nevoie de un multiplexor cu mai multe intrări deoarece avem mai multe operații de efectuat (mai exact toate cele menționate în context). Voi mai folosi două unități de deplasare, una stânga și cealaltă dreapta, iar cu ajutorul acumulatorului vom putea procesa temporar datele. În ceea ce privește înmulțirea, cea binara este asemănătoare celei zecimale. Ea se realizează prin însumarea produselor parțiale dintre operandul A și fiecare digit din operandul B. Rezultatul înmulțirii are un numar dublu de biți (față de operanzi). Deoarece fiecare digit din B reprezintă o putere a lui 2, înmulțirea este asigurată prin deplasarea la stanga a lui A.

Se observă faptul că pentru a înmulți 2 numere pe N biți este nevoie de N-1 deplasări și (în cel mai defavorabil caz când toți biții din B sunt 1) de N adunari.

Împărțirea a două numere reprezentate în binar se face asemănător cu situația în care ele sunt reprezentate în zecimal. Se încearcă să se scadă împărțitorul din deîmpărțit (începând cu poziția cea mai semnificativă).

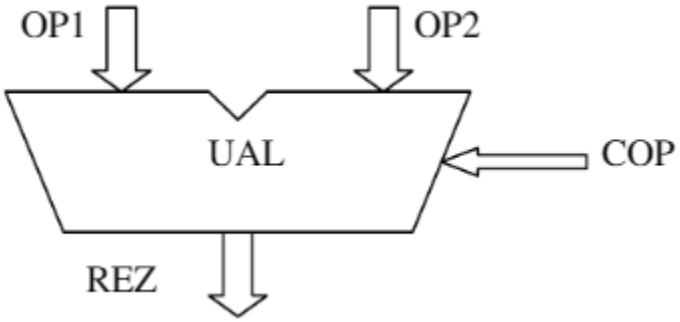


Figura 2

Reprezentarea simbolică a unei unități logice aritmetice folosită în diagramele bloc

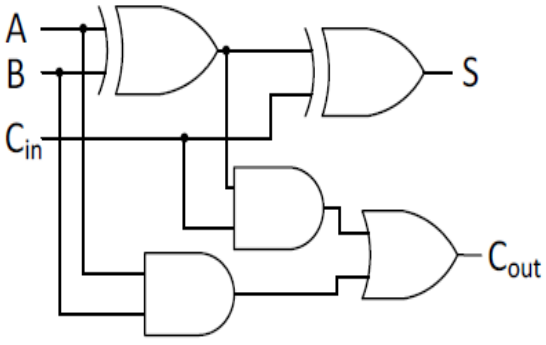
i. OP1, OP2 – cei doi operanzi reprezentați fiecare pe 32 de biți
ii. REZ – rezultatul operației dintre ei
iii. COP – codul de selecție al operației (codul operației), reprezentat pe m biți, deci se pot codifica în total 2^m operații diferite.

3. Analiza

3.1 Adunare și scădere în complement față de 2

Sumatorul este un circuit care adună numere binare. Se începe cu construcția unui sumator pe 1 bit. Pentru acesta, intrările sunt cei doi biți care se adună, A și B, iar ieșirile sunt bitul de sumă S și bitul de transport de ieșire C_{out} (eng. carry out). Acest tip de sumator se numește semi-sumator.

Semi-sumatorul rezolvă corect adunarea unor biți singulari. În cazul adunării numerelor pe mai mulți biți, acesta poate fi utilizat doar pentru perechea de biți de pe poziția cea mai puțin semnificativă. Ținând cont de aritmetica adunării pe mai mulți biți, pentru restul pozițiilor de biți trebuie adunat și transportul de la perechea anterioară de biți. Acest transport poartă numele de transport de intrare C_{in} (eng. carry in), iar un sumator care permite și acest tip de transport se numește *sumator complet*:



A	B	CIN	S	COUT
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

$$S = (A \oplus B) \oplus C_{in} \qquad C_{out} = A \cdot B + (A \oplus B) \cdot C_{in}$$

Figura 3

Pentru a construi un sumator pe N biți (în cazul nostru N=32) se pot folosi N sumatoare pe 1 bit. Dacă numerele de intrare pe *n* biți sunt $A_{n-1}…A_1A_0$ și $B_{n-1}…B_1B_0$, iar suma $S_{n-1}…S_1S_0$, atunci diagrama sumatorului pe *n* biți arată ca mai jos. Pe transportul de intrare al sumatorului pentru bitul 0 trebuie pusă valoarea 0 pentru a funcționa corect (pe bitul cel mai puțin semnificativ nu există transport de intrare). Mai jos e reprezentat sumatorul cu propagarea succesivă a transportului:

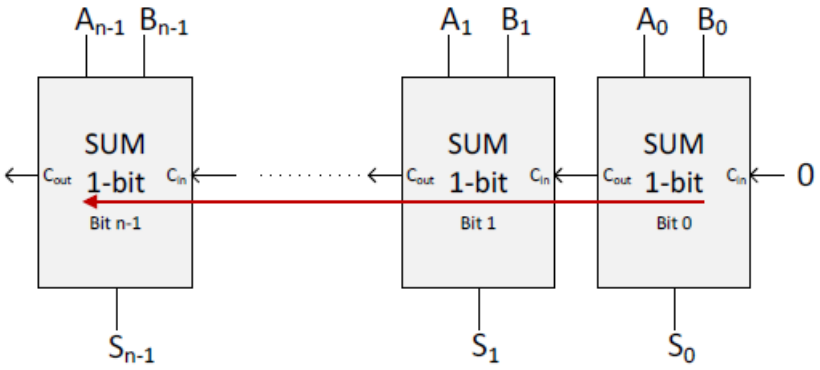


Figura 4

Pentru operația de scădere, am ales, în același fel ca la sumator, să folosesc un scăzător complet pe 1 bit si apoi să cascadez 32 din acestea. Un full subtractor este un circuit combinațional care efectuează scăderea a doi biți, luând în considerare împrumutul bitului anterior adiacent. Acest circuit are trei intrări și două ieșiri. Cele trei intrări sunt A, B și Bin (borrow In). Cele două ieșiri, D și Bout(borrow Out), reprezintă diferența și împrumutul de ieșire. Deși scăderea este de obicei realizată prin avantajele reprezentării în complement față de 2 (o scădere A – B este rescrisă ca A + (–B) și se ține cont de proprietatea că negativul unui număr întreg se obține negând
reprezentarea sa în complement față de 2 și adunând cu 1), este important să se elaboreze Tabelul de Adevăr și realizarea logică a acesteia, ce vor fi detaliate în paginile ce urmează.

3.2 Incrementare/Decrementare

În ceea ce privește incrementarea, ea este de fapt o adunare cu 1, iar decrementarea o scadere cu 1. Așadar, vom folosi sumatorul deja analizat și vom seta cei 32 de biți pe care îi reprezintă opearatorul B pe zero, iar pe cel mai puțin semnificativ pe 1. Astfel, nu voi realiza componente separate în cod pentru incrementare și decrementare.

3.3 ȘI/SAU/NU logic

Pentru operațiile logice pe biți, voi realiza un MUX 4:1, cu care în funcție de o anumită selecție, să se poată alege operația dorită. Adăugăm astfel o poartă logică ȘI (Sel = 00), una SAU (Sel = 01) și una NU (Sel = 10).

3.4 Rotație stânga și dreapta

Rotația la stânga și la dreapta a unui număr pe 32 de biți în hardware se poate realiza folosind operații logice de bază, cum ar fi shiftarea și rotirea cu o poziție. În exemplul de mai jos, n reprezintă cu câți biți trebuie să fie shiftat numărul.

număr_rotit_stânga = (număr << n) | (număr >> (32-n))

număr_rotit_dreapta = (număr >> n) | (număr << (32-n))

3.5 Circuitul pentru înmulțire și împărțire cu acumulatorul pentru operandul de intrare și rezultat

Pentru înmulțire, am ales să implementez algoritmul Shift-and-Add. Metoda aceasta este una dintre cele mai simple și de bază metode pentru adunarea a două numere. În principiu, ideea constă în adunarea multiplicandului (X) la el însuși de Y ori (multiplicator). Algoritmul se bazează pe luarea fiecărei cifre a multiplicandului pe rând și înmulțirea acesteia cu o singură cifră a multiplicatorului. Fiecare produs intermediar este plasat în pozițiile corespunzătoare, la stânga rezultatelor anterioare. În final, toate produsele intermediare sunt adunate împreună pentru a obține rezultatul. Diagrama bloc și organigrama ar fi cele de mai jos:

Figura 5 – organigrama pentru Shift and Add Multiplication

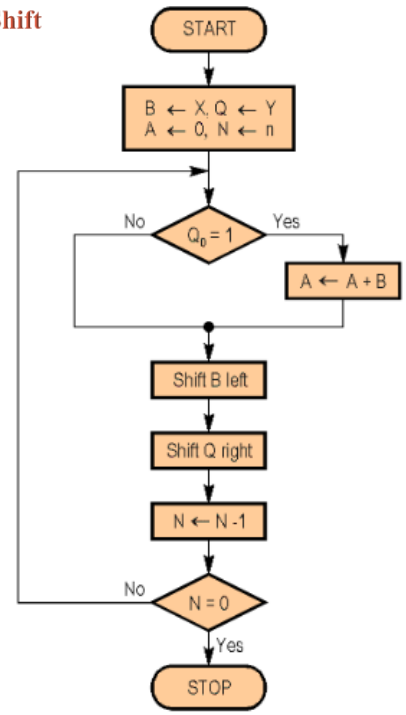
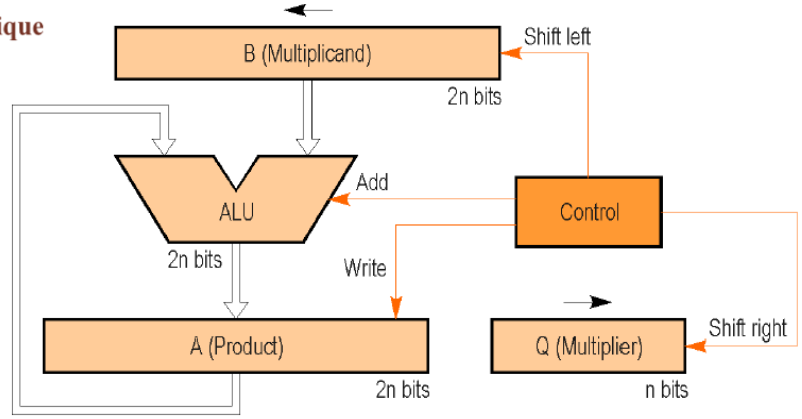


Figura 6 - Block design of the shift-and-add multiplication technique



Împărțirea s-ar putea face prin mai multe metode, dintre care eu am ales-o pe cea prin metoda refacerii restului parțial (Restoring Division):

Figura 7 – *flowchart Restoring Division*

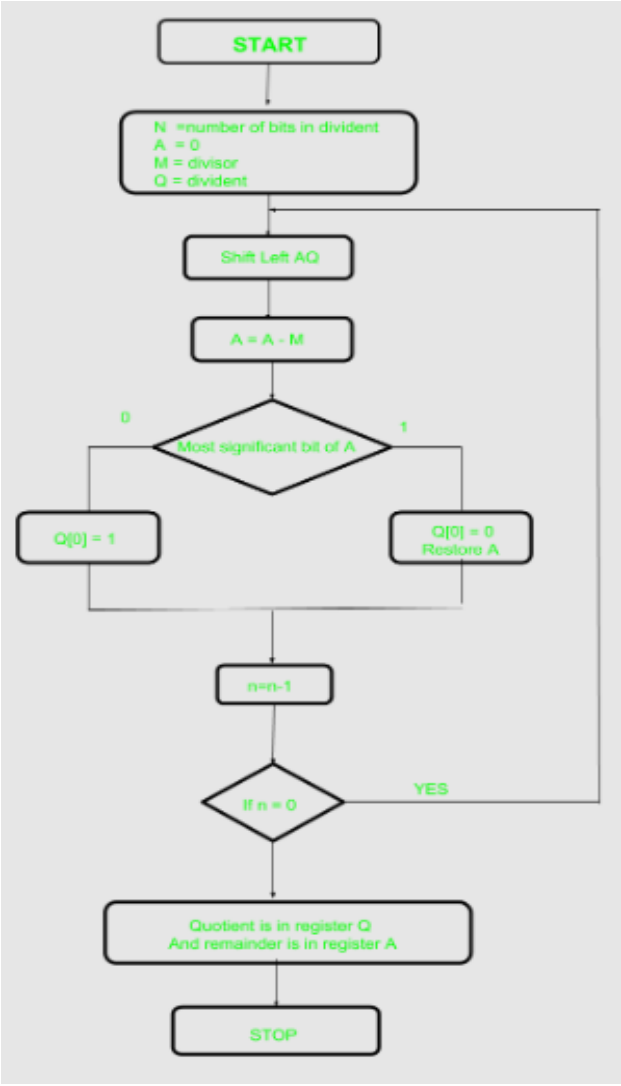
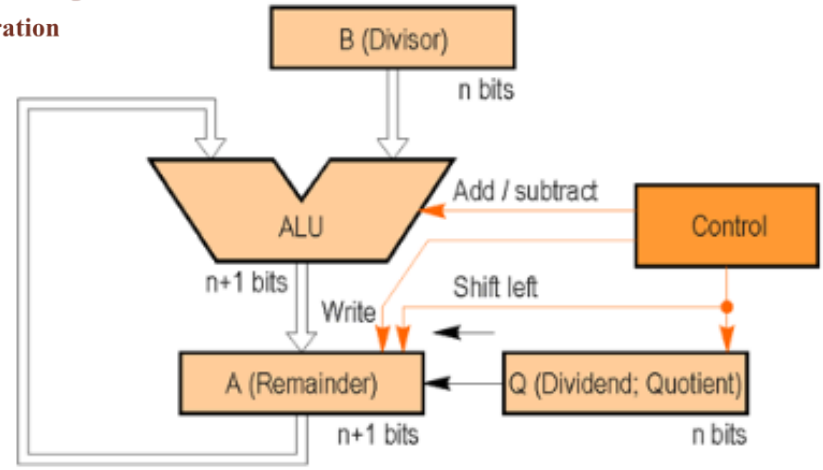


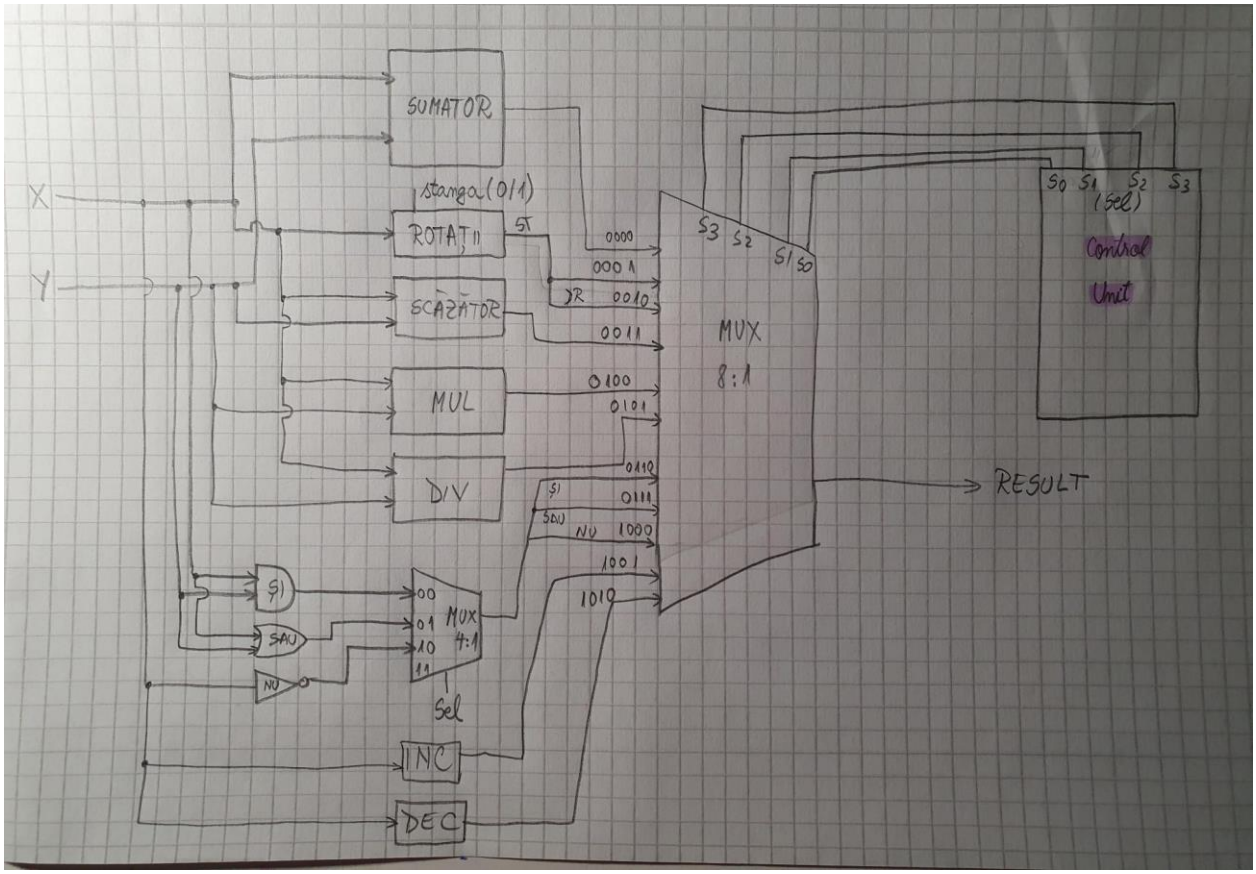
Figura 8 - Basic block design of the division operation

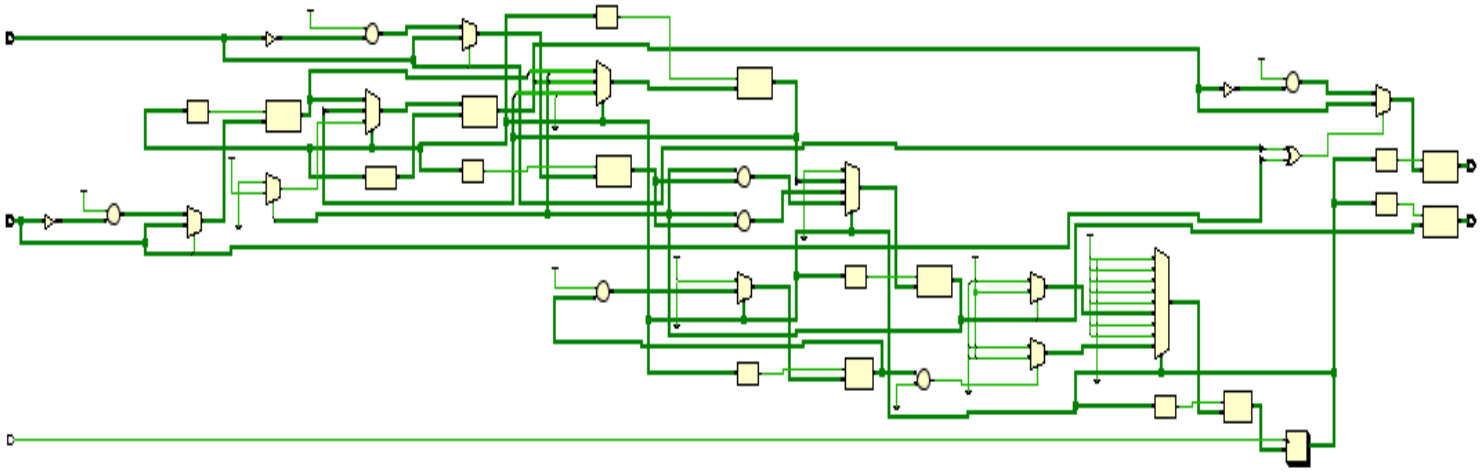


4. Proiectare

În capitolul de proiectare, se va realiza concatenarea tuturor unităților prezentate până acum, o imagine de ansamblu regăsindu-se mai jos. În cea de a doua poză este RTL schematic realizat in VHDL.

Figura 9 – diagrama bloc a întregului proiect și RTL schematic





5. Implementare

În realizarea implementării, am început cu sumatorul (*project_1*), pentru care m-am folosit de Ripple Carry Adder -ul studiat la laborator. Prima oară, am implementat un sumator complet pe un bit, pe care ulterior l-am folosit concatenând 32 din acestea. Ripple Carry Adder-ele sunt implementate prin mai multe sumatoare complete în serie, unde ieșirea de transport a fiecăruia este conectată la intrarea celui următor. Acest tip este un sumator în paralel și este folosit pentru adunarea unui număr de n biți. Are avantajul simplității și costului redus, dar cu costul unei viteze reduse. Figura de mai jos ilustrează diagrama bloc a unui sumator cu transport în cascada pentru adunarea numerelor de 4 biți.

Figura 10 – implementare sumator și tabel de adevăr

```
entity FullAdder is
    port(
        A, B: in STD_LOGIC;
        CarryIn: in STD_LOGIC;
        CarryOut: out STD_LOGIC;
        Si: out STD_LOGIC
        );
end entity;
architecture Behavioral of FullAdder is
begin
Si <= A xor B xor CarryIn;
CarryOut <= (A and B) or (CarryIn and (A xor B));
end Behavioral;
```

Table 1. Full adder truth table

X_i	Y_i	C_i	S_i	C_{i+1}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

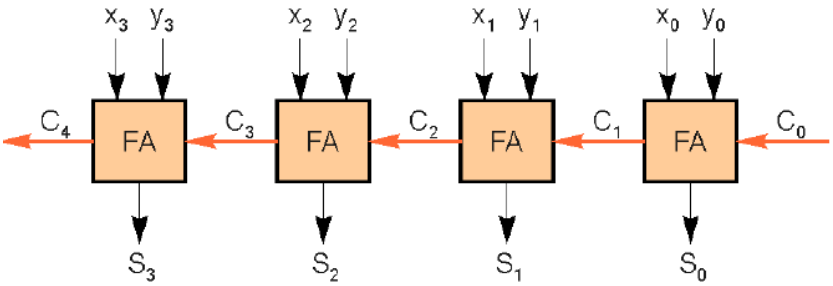
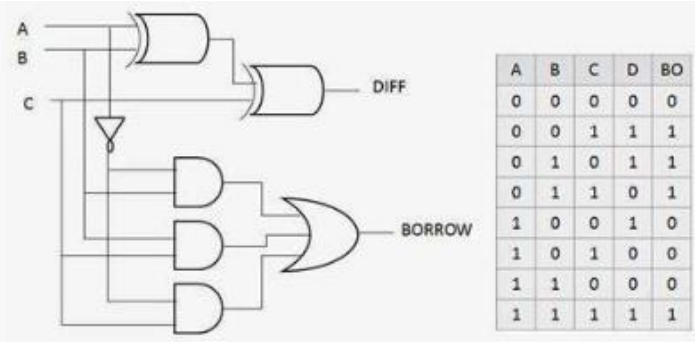


Figure 2. Ripple carry adder using 4 full adders, for 4-bit numbers

Am continuat implementarea codului VHDL cu scăzătorul(*subtractor_32biti*) complet pe 1 bit, și în același fel ca la adder, am cascadat 32 pentru a putea realiza scăderea dintre 2 numere pe 32 de biți, folosindu-mă de semnalele de borrow in si borrow out, precup si tabelul de adevăr:

Figura 11 – tabelul de adevăr al scăzătorului complet pe un bit + implementări în VHDL

```
entity FullSubtractor is
    Port(
        A, B: in STD_LOGIC;
        BorrowIn: in STD_LOGIC;
        BorrowOut: out STD_LOGIC;
        Diff: out STD_LOGIC
    );
end FullSubtractor;
architecture Behavioral of FullSubtractor is
begin
    Diff <= (A XOR B) XOR BorrowIn;
    BorrowOut <= ((not A) AND (B OR BorrowIn)) OR (B AND BorrowIn);
end Behavioral;
```



A	B	C	D	BO
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

```
sub0: FullSubtractor port map(X(0), Y(0), BI, C(0), Dif(0));
subtractor_i: for i in 1 to 30 generate
FullSubtractor_i: FullSubtractor port map (X(i), Y(i), C(i-1), C(i), Dif(i));
end generate;
sub31: FullSubtractor port map(X(31), Y(31), C(30), BO, Dif(31));
```

Pentru a putea implementa înmulțitorul(*Multiplier*) pe 32 de biți, m-am folosit de algoritmul Shift and Add, a cărui organigramă se regăsește în figura 5, și după care m-am ghidat în implementarea state machine-ului meu, împreună cu figura 6. În scrierea codului, am avut grijă să tratez și cazurile pentru numerele negative, și să adaug un adder pe 64 de biți (*Add64bit*) deoarece aveam nevoie la final, având în vedere că rezultatul unui produs a două numere pe 32 de biți va putea avea (aici) maxim 64 de biți.

```
process(clk)
begin
    if rising_edge(clk) then
        if start = '1' then
            if multiplicand(31) = '1' then
                MUL1 <= (not multiplicand) + x"00000001";
            else
                MUL1 <= multiplicand;
            end if;
            if multiplier(31) = '1' then
                MUL2 <= (not multiplier) + x"00000001";
            else
                MUL2 <= multiplier;
            end if;
            B(63 downto 32) <= x"00000000";
            B(31 downto 0) <= MUL1;
            A <= x"0000000000000000";
            Q <= MUL2;
        else
            if shiftL = '1' then
                B <=  B(62 downto 0) & "0"; -- shiftare la stanga
            end if;
            if shiftR = '1' then
                Q <= '0' & Q(31 downto 1); -- shiftare la dreapta
            end if;
            if scriere = '1' then
                A <= aux;
            end if;
        end if;
    end if;
```

Figura 12 – implementarea codului VHDL pentru Multiplier

```
ShiftAndAdd: Add64bit port map(A, B, '0', res, aux);

process(multiplicand, multiplier, A)
begin
    if (multiplicand(31) xor multiplier(31)) = '1' then
        rez <= (not A) + x"00000001";
    else
        rez <= A;
    end if;
end process;
```

În continuare, folosindu-mă de figurile 7 si 8, am realizat și implementarea împărțitorului(*Division*) pe 32 de biți, cu algoritmul Restoring Division, regăsit în link-ul 8 și 9 din bibliografie. Mai întâi, registrele sunt inițializate cu valorile corespunzătoare (Q = deîmpărțitul, M = împărțitorul, A = 0, n = numărul de biți din deîmpărțit). Apoi, conținutul registrelor A și Q este deplasat la stânga ca și cum ar fi o unitate unică. Pasul 3, conținutul registrului M este scăzut din A, iar rezultatul este stocat în A. Pasul 4, cel mai semnificativ bit al lui A este verificat; dacă este 0, cel mai puțin semnificativ bit al lui Q este setat la 1; în caz contrar, dacă este 1, cel mai puțin semnificativ bit al lui Q este setat la 0, iar valoarea registrului A este restaurată, adică valoarea lui A înainte de scăderea cu M. Valoarea contorului n este decrementată, iar dacă valoarea lui n devine zero, ieșim din buclă; în caz contrar, repetăm de la pasul 2. În final, registrul Q conține câtul, iar A conține restul.

Totodată, am aplicat si aici aceleași reguli ca la multiplier, având grijă să tratez cazurile unor împărțiri subunitare, supraunitare, cu rest sau fără rest, cu numere negative și pozitive:

Figura 13 – cod VHDL – implementarea algoritmului de împărțire

```
process(state, M, Q, sM, sQ)
    begin
        case state is
            when S0 =>
                if M(31) = '1' then
                    sM <= (not M) + x"00000001";
                else
                    sM <= M;
                end if;
                if Q(31) = '1' then
                    sQ <= (not Q) + x"00000001";
                else
                    sQ <= Q;
                end if;
                A1 <= X"00000000";
                AQ <= A1 & sQ;
                Q1 <= sQ;
                n <= "100000";--numarul de biti ai deimpartitului = 32
                next_state <= shiftLeftAQ;
            when shiftLeftAQ =>
                AQ <= A1 & Q1;
                next_state <= nextShift;
            when nextShift =>
                AQ <= AQ(62 downto 0) & '0';
                next_state <= AQSplit;
            when AQSplit =>
                A1 <= AQ(63 downto 32);
                Q1 <= AQ(31 downto 0);
                next_state <= A_minus_M;
            when A_minus_M =>
                A1  <= STD_LOGIC_VECTOR(unsigned(A1)-unsigned(sM));
                next_state <= condition1;
            when condition1 =>
                if A1(31) = '1' then
                    Q1(0) <= '0';
                    next_state <= restoreA;
                else
                    Q1(0) <= '1';
                    next_state <= n_minus_1;
                end if;
            when restoreA =>
                A1 <= STD_LOGIC_VECTOR(unsigned(A1)+unsigned(sM));
                next_state <= n_minus_1;
            when n_minus_1 =>
                n <= STD_LOGIC_VECTOR(unsigned(n)-"000001");
                next_state <= condition2;
            when condition2 =>
                if n = "000000" then
                    next_state <= finish_state;
                else
                    next_state <= shiftLeftAQ;
                end if;
            when finish_state =>
                A <= A1;
                if (M(31) xor Q(31)) = '1' then
                    Q_out <= (not Q1) + x"00000001";
                else
                    Q_out <= Q1;
                end if;
```

După ce am finalizat operațiile complexe, am încorporat și rotațiile(*Rotations*) stânga și dreapta, folosindu-mă de multiplexoare 2:1 pe un bit. Selecția o reprezintă semnalul numit *stanga*, care, setat pe 0 va shifta numărul X la dreapta cu o poziție, iar dacă e setat pe 1, la stânga. O explicație mai în detaliu a felului în care am procedat, se regăsește în figura 16.

Figura 14 – cod rotații

```
mux1: MUX1bit port map(stanga, nr(0), nr(30), rez(31));
bucla: for i in 31 downto 2 generate
MUX: MUX1bit port map(stanga, nr(i), nr(i-2), rez(i-1));
end generate;
mux32: MUX1bit port map(stanga, nr(1), nr(31), rez(0));
```

În cele din urmă, am implementat și operațiile logice ȘI, SAU și NU(*OperatiiLogice*), după cum urmează, iar pentru incrementare și decrementare, m-am folosit de sumator, respectiv scăzător, pentru a nu mai adăuga componente în plus.

Figura 15 – cod VHDL AND/OR/NOT

```
process(A, B, Sel)
 begin
    case Sel is
        when "00" => Res <= A and B;
        when "01" => Res <= A or B;
        when "10" => Res <= not A;
        when others => Res <= x"FFFFFFFF";
    end case;
end process;
```

Finalul l-a constituit concatenarea tuturor componentelor realizate până acum, astfel încât să întruchipeze o unitate aritmetico-logică. Am realizat o unitate de control(*ControlUnit*), care să îmi reprezinte selecția între operații, însă pentru înmulțire și împărțire ar fi trebuit să folosesc un registru acumulator pentru intrare și ieșire. Totuși, am ajuns la rezultate bune și în acest fel. Totodată, trebuie menționat și faptul că am realizat câte un TestBench pentru fiecare dintre operații, pentru a putea fi testată funcționalitatea.

```
if rising_edge(CLK) then
    if ENABLE = '1' then
        case SEL is
            when "0000" =>
                RESULT(31 downto 0) <= suma;
                RESULT(63 downto 32) <= x"00000000";
            when "0001" =>
                RESULT(31 downto 0) <= rotit_dreapta;
                RESULT(63 downto 32) <= x"00000000";
            when "0010" =>
                RESULT(31 downto 0) <= rotit_stanga;
                RESULT(63 downto 32) <= x"00000000";
            when "0011" =>
                RESULT(31 downto 0) <= dif;
                RESULT(63 downto 32) <= x"00000000";
            when "0100" =>
                RESULT <= produs;
            when "0101" =>
                RESULT(31 downto 0) <= cat;
                RESULT(31 downto 0) <= rest;
            when "0110" =>
                RESULT(31 downto 0) <= rezAND;
                RESULT(63 downto 32) <= x"00000000";
            when "0111" =>
                RESULT(31 downto 0) <= rezOR;
                RESULT(63 downto 32) <= x"00000000";
            when"1000" =>
                RESULT(31 downto 0) <= rezNOT;
                RESULT(63 downto 32) <= x"00000000";
            when"1001" =>
                RESULT(31 downto 0) <= inc;
                RESULT(63 downto 32) <= x"00000000";
            when"1010" =>
                RESULT(31 downto 0) <= dec;
                RESULT(63 downto 32) <= x"00000000";
            when others => RESULT <= (others => '0');
    end case;
```

Figura 16 – implementarea tuturor componentelor și selecția lor

```
SUMATOR: project_1 port map(X, Y, '0', carryOut, suma);
ROTATIE_DREAPTA: Rotations port map(X, rotit_dreapta, '0');
ROTATIE_STANGA: Rotations port map(X, rotit_stanga, '1');
SCAZATOR: subtractor_32biti port map(X, Y, '0', borrowOut, dif);
INMULTITOR: Multiplier port map(CLK, st, X, Y, produs);
IMPARTITOR: Division port map(CLK, Y, rest, X, cat);
SI_LOGIC: OperatiiLogice port map("00", X, Y, rezAND);
SAU_LOGIC: OperatiiLogice port map("01", X, Y, rezOR);
NU_LOGIC: OperatiiLogice port map("10", X, Y, rezNOT);
INCREMENTARE: project_1 port map(X, x"00000001", '0', carryOut, inc);
DECREMENTARE: subtractor_32biti port map(X, x"00000001", '0', borrowOut, dec);
```

6. Testare

✓ Am început testarea programului în VHDL, în primul rând cu sumatorul și mai apoi multiplicatorul, extinse pe 32 de biți.

Pentru sumator, am testat pe mai multe numere, atât pozitive cât și negative, simularea realizându-se cu succes în fiecare caz. Primul exemplu, adună numere simple, pozitive, iar în următoarele două avem calculele: *(-5) + 1515 = 1510* și *(-12345) + (-7896) = -20241*

$1510_{10} = 0101\_1110\_0110_2$ (5e6 din simulare)
$-20241_{10} = 1111\_1111\_1111\_1111\_1011\_0000\_1110\_1111_2$ (ffffb0ef în simulare)

Figura 17 – testare adder

✓ Pentru multiplicator, am testat cu câteva numere negative și pozitive, mari și mici, iar algoritmul folosit funcționează, atât timp cât numerele și rezultatul nu depășesc 32 de biți:

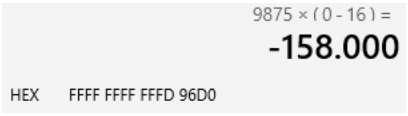
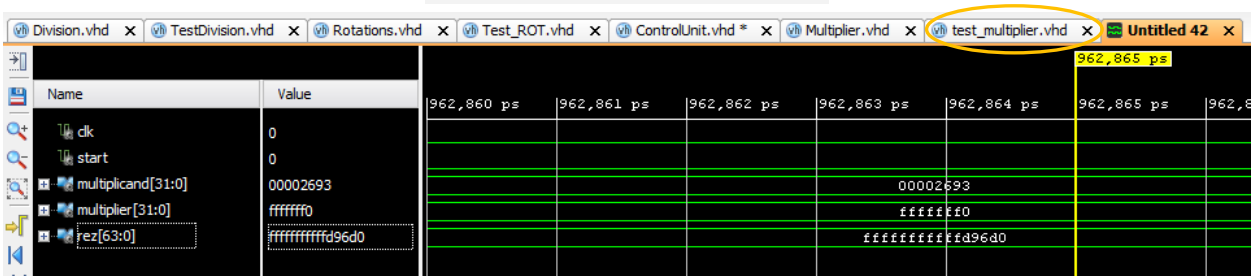
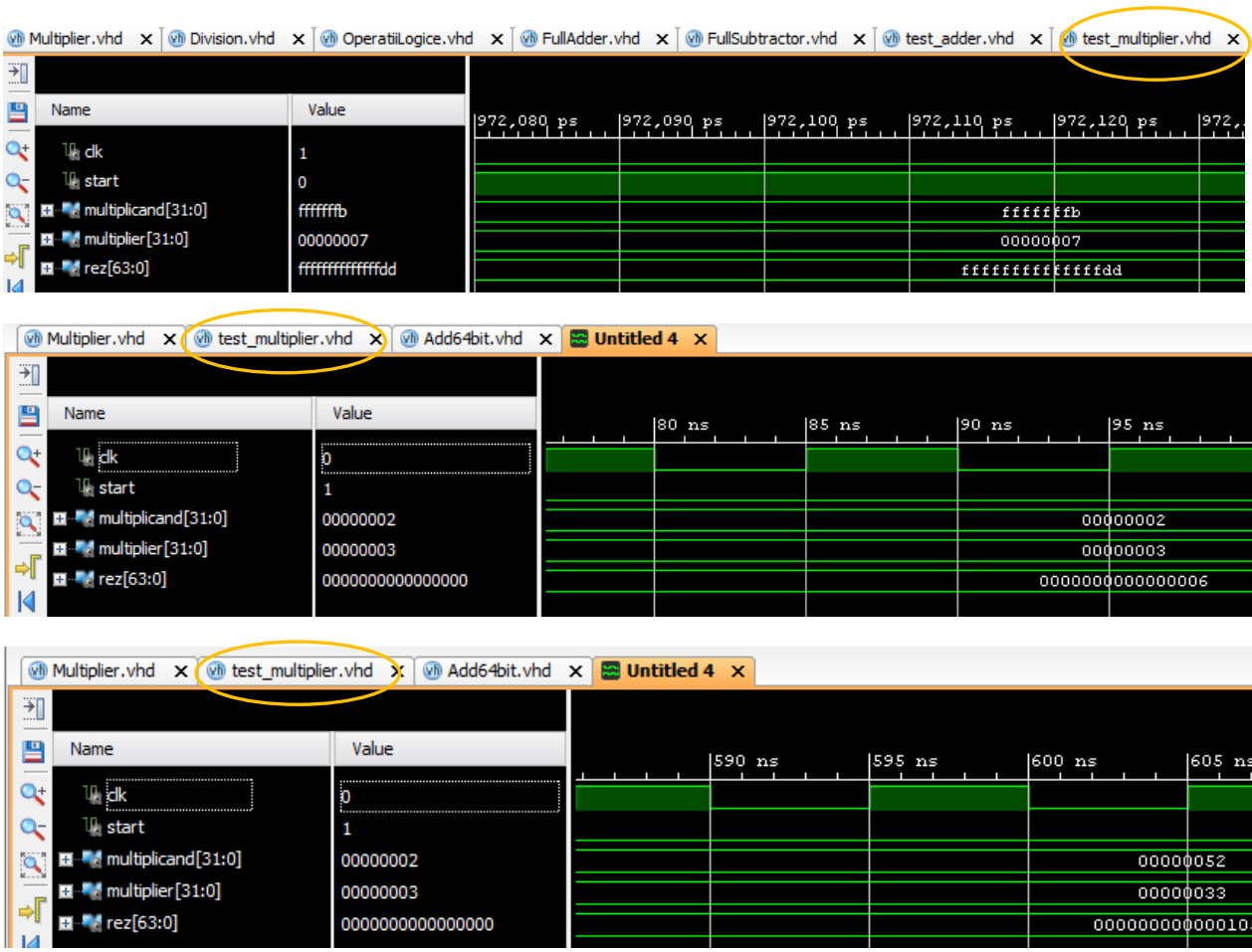


Figura 18 – testare multiplier

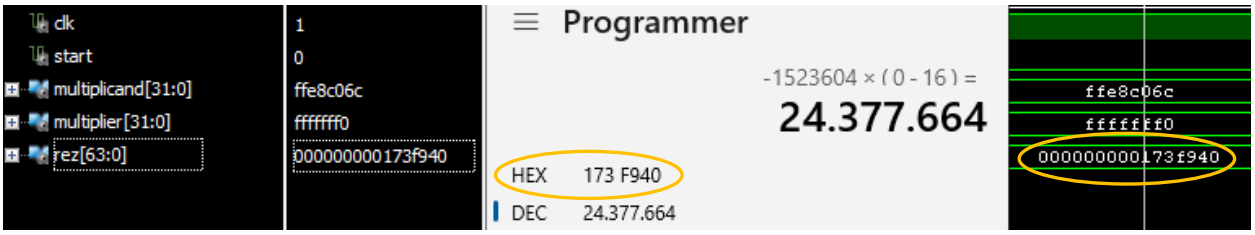


(-5)*7 = -35



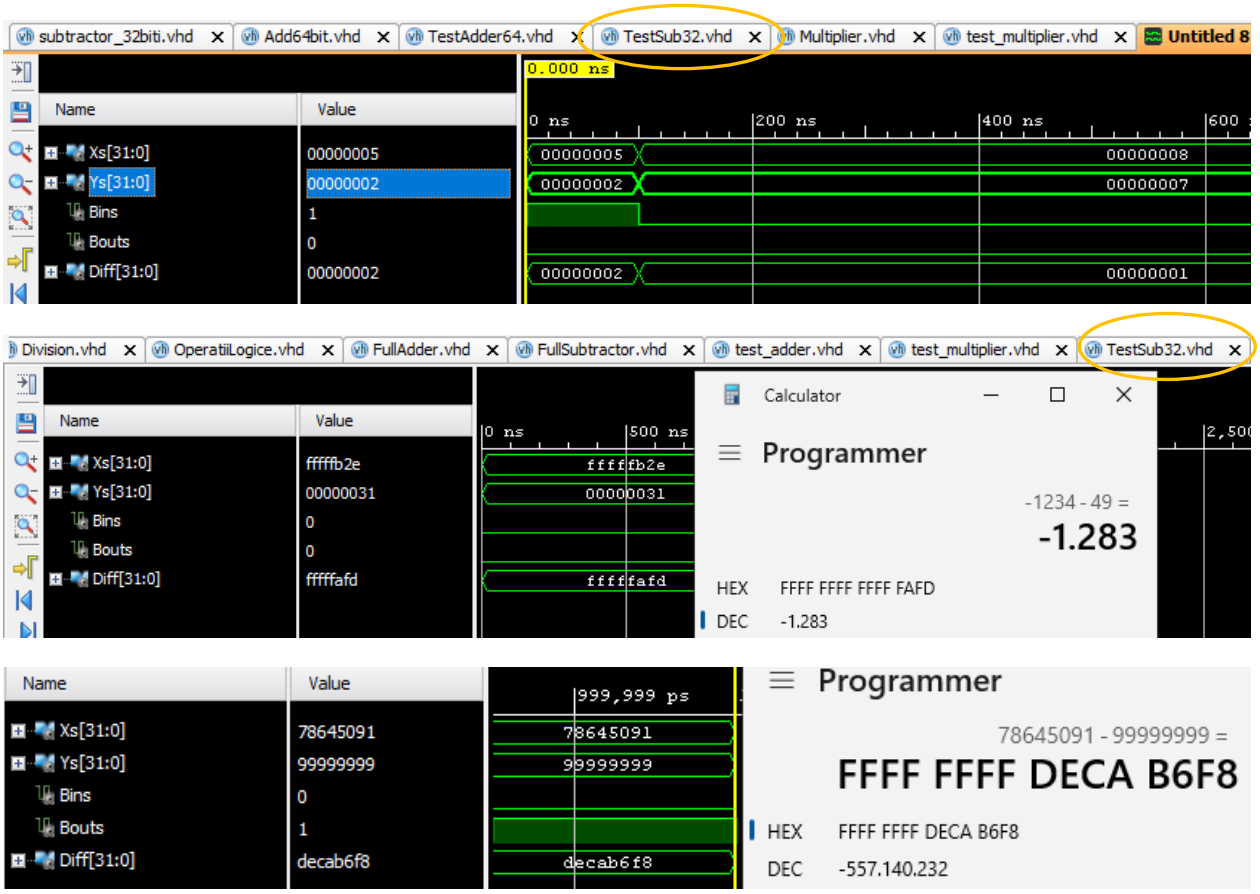
Se poate observa că, pentru multiplicator am folosit un adder extins să funcționeze pentru adunarea a două numere pe 64 de biți, deoarece trebuia realizată adunarea între A și B din figura 5, iar mai apoi, folosind algoritmul, am obținut rezultatele corecte: numărul x00000052 este de fapt egal cu 82 în decimal, iar x00000033 = 51. Înmulțite, rezultatul va fi 4182 care în binar este 0001_0000_0101_0110 (exact numărul 1056 din poza de mai sus – figura 11). Mai jos, am înmulțit și două numere negative, rezultatul fiind pozitiv. Am verificat toate calculele folosind calculatorul:

Figura 19 – testare multiplier



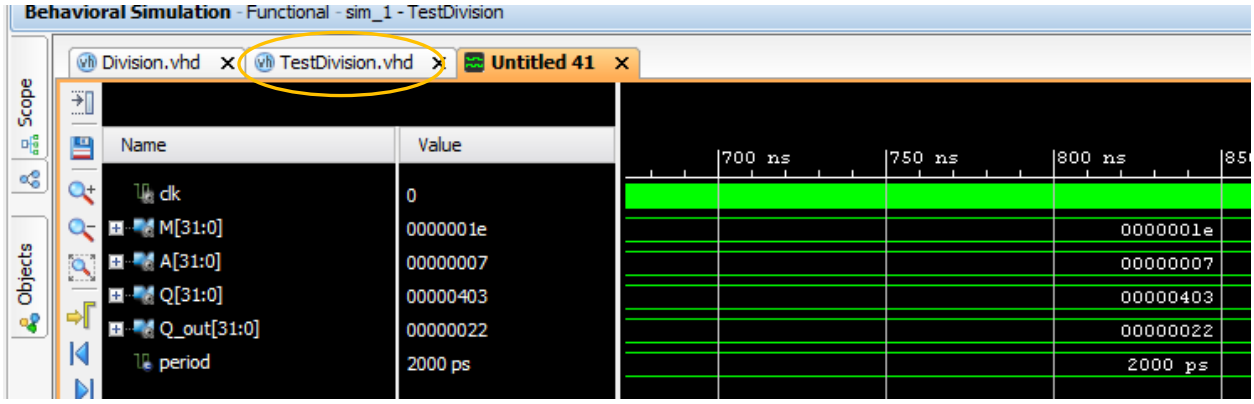
✓ Pentru scăzătorul pe 32 de biți, abordarea este aceeași ca la adder, iar funcționalitatea e dovedită prin testare pe toate tipurile de numere:

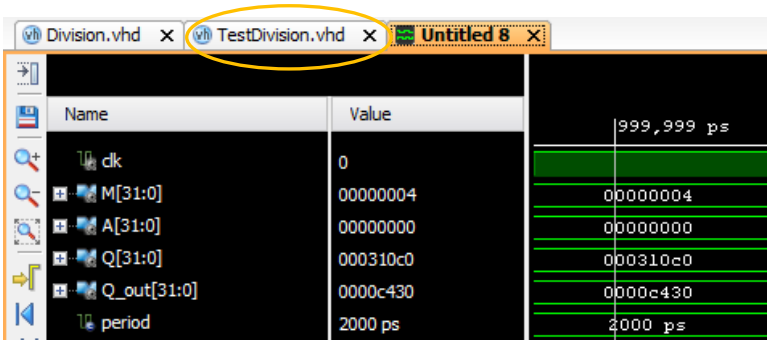
Figura 20 - testare scăzător



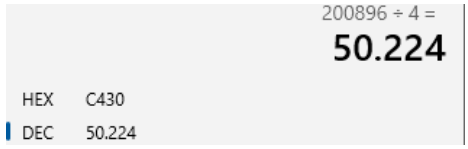
✓ În continuare, am realizat testarea împărțitorului pe 32 de biți. Se poate observa calculul 1027:30 = 34 rest 7. Q = 0100_0000_0011(4_0_3), care se împarte la M = 1_1110(1_e), rezultând câtul Q_out = 10_0010(2_2) și restul A = 0111(7). Mai jos este atașată simularea:

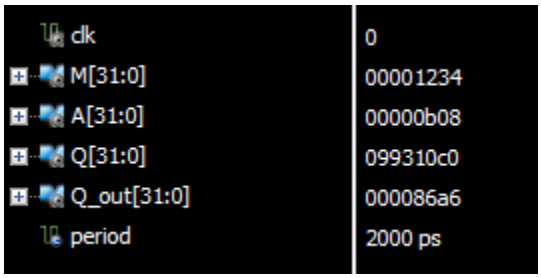
Figura 21 – testare divider





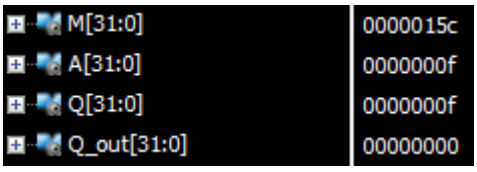
Împărțire cu restul zero:





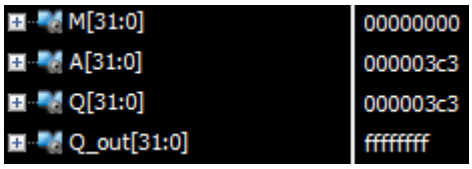
$160.633.024_{10} : 4660_{10} = 34470_{10}$ rest 2824_{10}

Am testat și pentru fracții subunitare:

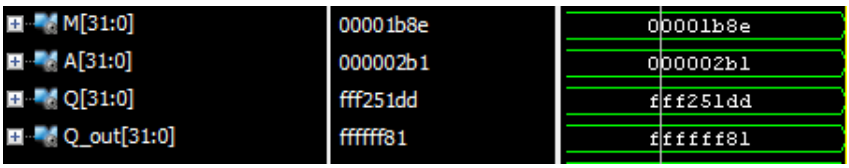


15 : 348 = 0 rest 15

Împărțire la zero(imposibilă):



Funcționează și pe numerele negative, am tratat toate cazurile:



$(-896.547)_{10} : 7054_{10} =$

$(-127)_{10}$ rest 689_{10}

✓ După ce am finalizat operațiile de bază, am început să testez rotațiile stânga și dreapta, folosindu-mă de multiplexoare 2:1 pe un bit. O explicație mai în detaliu a felului în care am procedat, se regăsește în figura 16. Am testat și pe numere negative, iar în figura de mai jos, cu numărul în hexa x"800005eb" = 2.147.485.163(în zecimal) rotit la dreapta și cu $(-987.521)_{10}$ rotit la stânga, avem un rezultat corect la simularea cu TestBench.

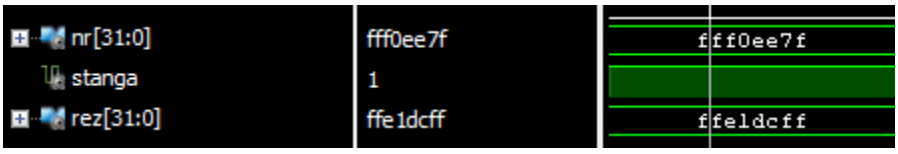


Figura 22 – testare rotații pe biți

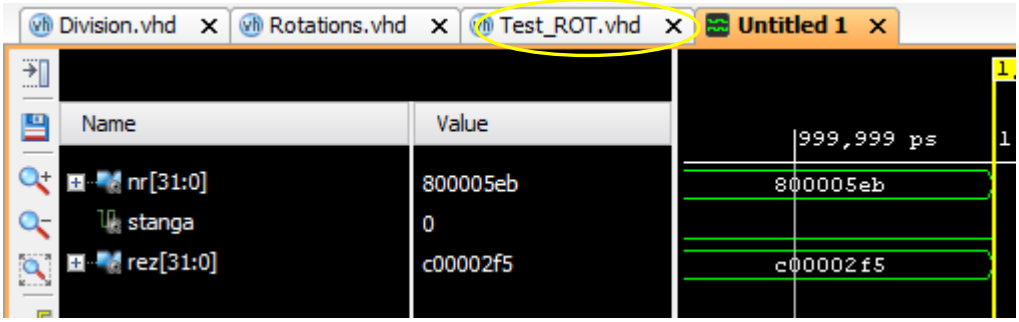
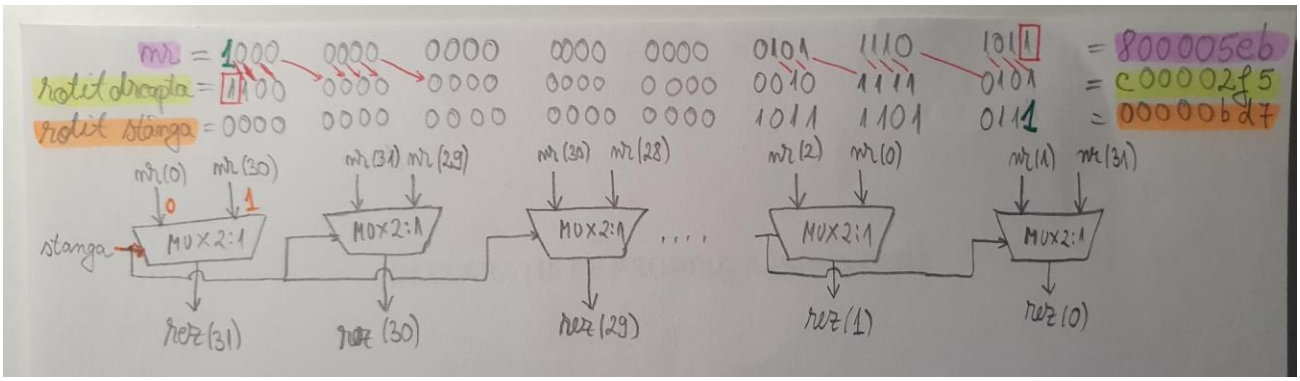
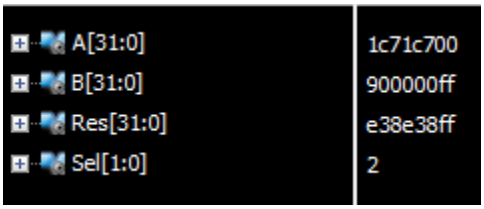
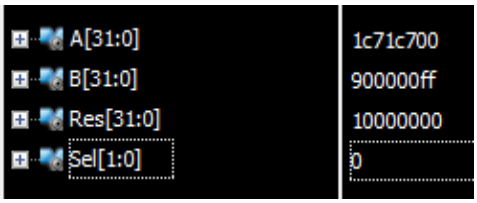


Figura 23 – explicație funcționare rotații



✓ În final, am realizat și operațiile logice ȘI, SAU, NU (and, or și not), în funcție de selecția Sel, pe doi biți(MUX 2:1 cu selecție pe 2 biți), iar mai jos am exemplificat numărul hexazecimal x"1c71c700" și felul în care trece prin toate fazele după testare.





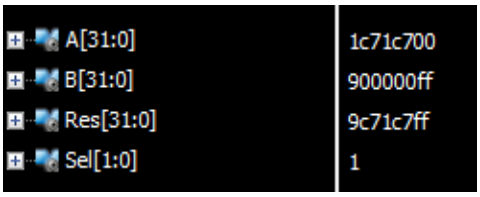


Figura 24 – testare operații logice

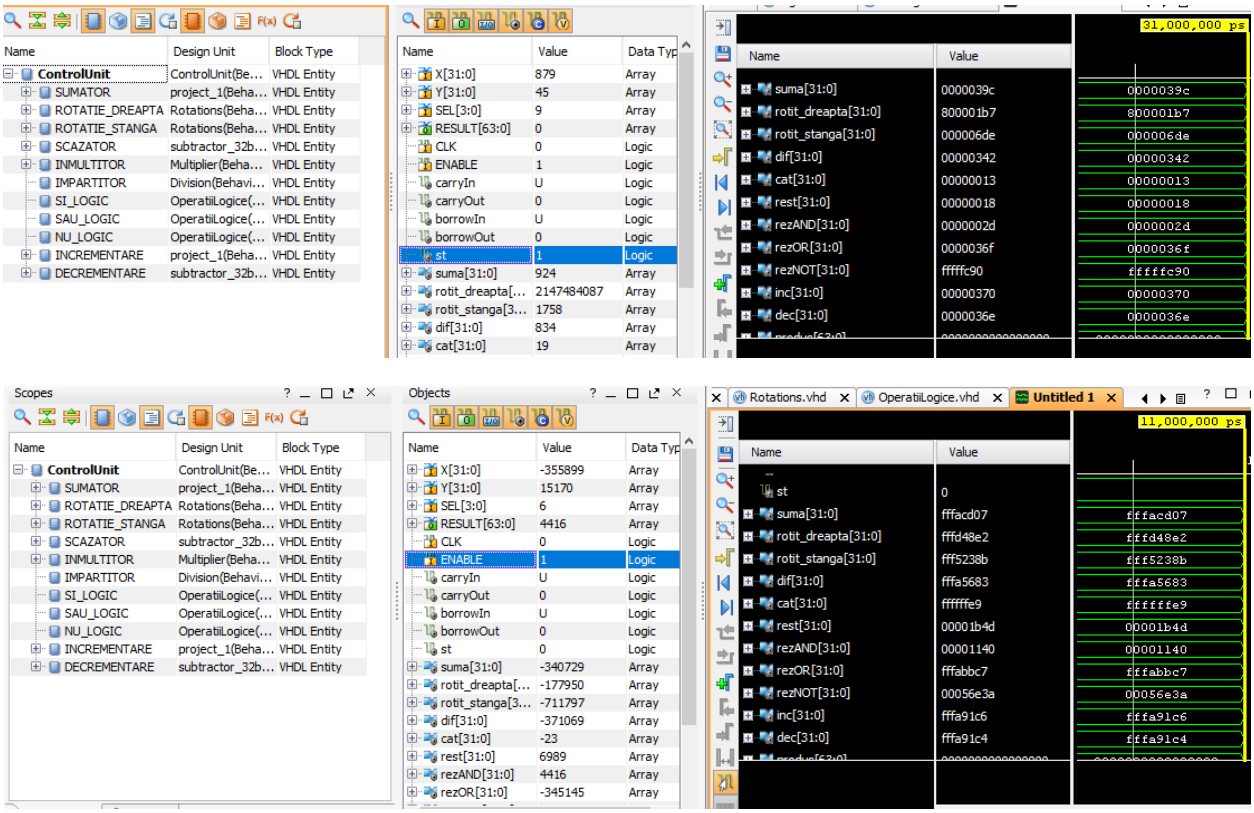
A = 0001_1100_0111_0001_1100_0111_0000_0000

B = 1001_0000_0000_0000_0000_0000_1111_1111

Res = 1110_0011_1000_1110_0011_1000_1111_1111 (not)
Res = 1000_0000_0000_0000_0000_0000_0000_0000 (and)
Res = 1001_1100_0111_0001_1100_0111_1111_1111(or)

Testarea componentei finale s-a realizat cu succes, după cum se poate observa în cele două simulări de mai jos, în care mă folosesc de numerele 879 și 45, respectiv -355899 și 15170.

Figura 25 – testare finală



7. Concluzii

În concluzie, proiectul de implementare a Unității Aritmetice și Logice (ALU) în limbajul VHDL a fost un succes, atingând obiectivele stabilite inițial. ALU-ul dezvoltat și documentat prezintă funcționalități robuste, inclusiv adunare, scădere, înmulțire, împărțire, rotații pe biți, incrementări sau decrementări și operații logice pe toate tipurile de numere întregi, toate în conformitate cu specificațiile proiectului. Testarea riguroasă a confirmat corectitudinea funcționalităților implementate.

Problemele întâmpinate pe parcurs au fost rezolvate cu succes, oferindu-mi posibilitatea de a învăța mult mai în profunzime materia. Cred că proiectul realizat de mine ar putea avea o multitudine de dezvoltări ulterioare, principala fiind posibilitatea implementării unui cod mai bun în ceea ce privește unitatea de control, registrele și automatele cu stări finite.

8.Bibliografie

În realizarea acestui proiect, m-am ajutat de cursurile predate anul acesta la Structura Sistemelor de Calcul, în special cursul 3, referitor la unitatea aritmetică logică, plus laboratoarele de exerciții în VHDL. De asemenea, site-urile în care am găsit inspirație ar fi:

01. *https://www.wikiwand.com/ro/Unitate_aritmetic%C4%83-logic%C4%83*

02. *De la bit la procesor – Florin Oniga (cartea folosită la arhitectura calculatoarelor) extension://bfdogplmndidlpjfhoijckpakkdjkkil/pdf/viewer.html?file=https%3A%2F%2Fandrei.clubcisco.ro%2F2cn1%2Flaboratoare%2FLaboratorul%25204.pdf*

03. *Algoritmul de multiplicare al lui Booth - frwiki.wiki*

04. *UAL - 4 Unitatea aritmetica - logica UAL executa toate operatiile aritmetice si logice din - Studocu*

05. *extension://bfdogplmndidlpjfhoijckpakkdjkkil/pdf/viewer.html?file=https%3A%2F%2Finst.eecs.berkeley.edu%2F~cs150%2Fsp13%2Fagenda%2Flec%2Flec21-mult-shift.pdf*

06. *https://www.youtube.com/watch?v=U62iP8RkZIk*

07. *Restoring Division Algorithm for Unsigned Integer - YouTube*

08. *Restoring Division Algorithm For Unsigned Integer - GeeksforGeeks*

09. *extension://bfdogplmndidlpjfhoijckpakkdjkkil/pdf/viewer.html?file=https%3A%2F%2Fusers.utcluj.ro%2F~baruch%2Fbook_ssce%2FSSCE-Basic-Division.pdf*

10. *Full Subtractor in Digital Logic - GeeksforGeeks*