Proiect 8 bit – ALU cu 4 operații

Autori:

Vintan Iulia – Grupa 3.2 C

Simion Vlad – Grupa 2.2 C

Profesor coordonator:

Bozdog Alexandru

An universitar 2024–2025

Cuprins

Prezentare generala a proiectului si obiective

Proiectul are ca scop dezvoltarea unei unități aritmetico-logice (ALU) capabile să efectueze patru operații fundamentale: adunare, scădere, înmulțire și împărțire. Proiectul este implementat în Verilog HDL și simulat utilizând un testbench dedicat. Scopul este dezvoltarea unui design modular hardware, utilizarea semnalelor de control si verificarea funcționării corecte a unitatii aritmetico-logice.

Arhitectura hardware este construita in jurul unei ordinograme care descrie pașii necesari pentru realizarea fiecărei operații (adunare, scădere, înmulțire, împărțire). În cadrul acesteia sunt evidențiate atât succesiunea operațiilor, cât și semnalele de control asociate, necesare pentru activarea diferitelor funcționalități precum: adunare, deplasare de biți (shiftare), încărcarea valorilor de pe magistrale în registre, și altele. *(fig 1.1)*

Structura ordinogramei asigură o execuție secvențială corectă a operațiilor și oferă o viziune clară asupra modului în care resursele hardware sunt coordonate prin semnale de control.

Unitatea aritmetico-logica implementeaza urmatoarele operatii:

- Adunarea
- Scaderea
- Inmultirea – pentru care ne-am folosit de algoritmul **Booth Radix 2**
- Impartirea – pentru care ne-am folosit de algoritmul **Restoring Division**

În ceea ce privește Unitatea de Control am ales să implementăm un sequence counter pentru a avansa prin stările de control. Această abordare este mai eficientă din punct de vedere al resurselor hardware, cum ar fi flip-flopurile. Această soluție permite astfel un design mai compact și mai eficient din punct de vedere al consumului de energie și al utilizării resurselor, aspecte esențiale pentru performanța globală a sistemului.

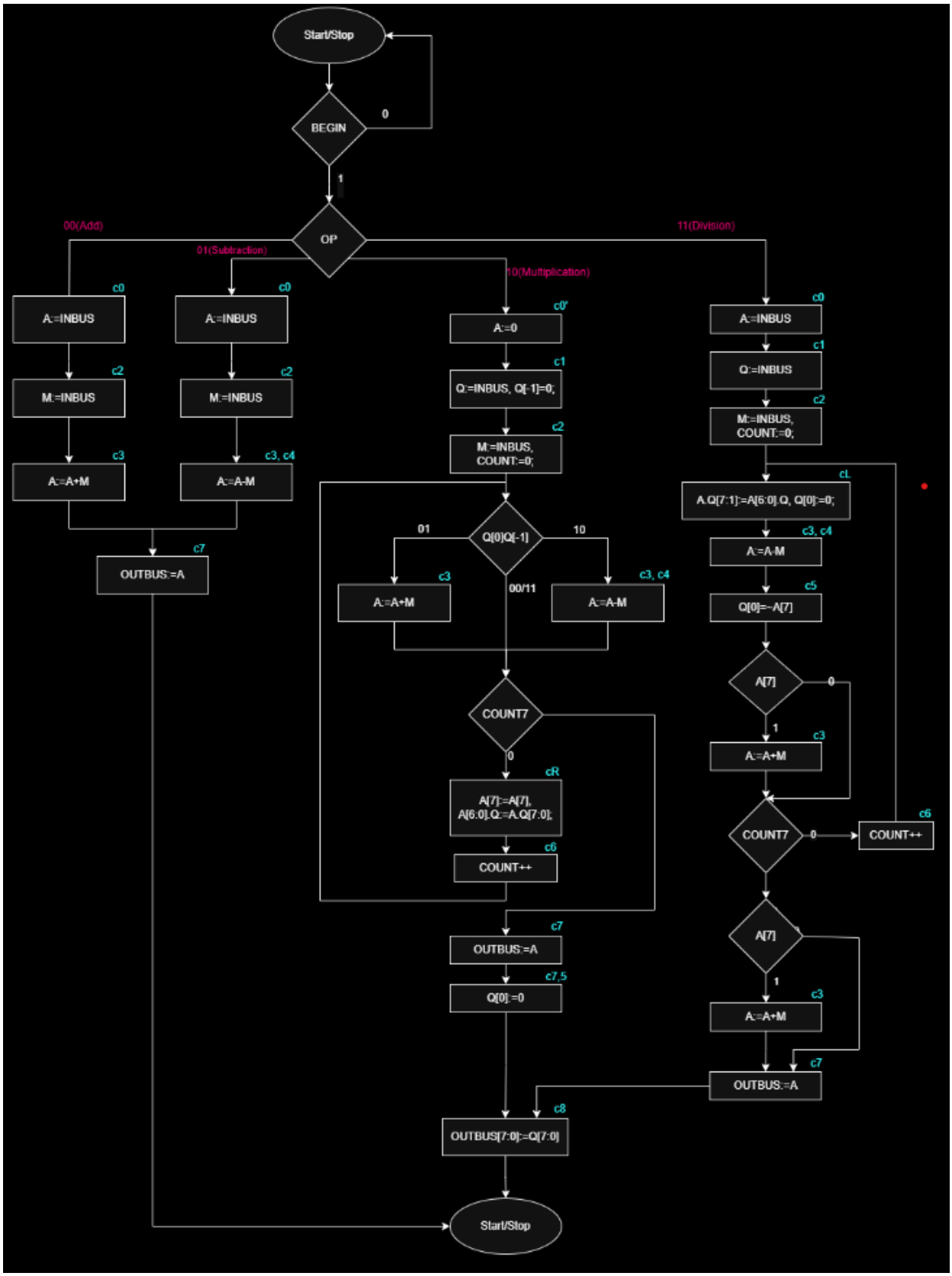

Start/Stop
BEGIN
0
1
OP
00(Add)
01(Subtraction)
10(Multiplication)
11(Division)
c0
A:=INBUS
c2
M:=INBUS
c3
A:=A+M
c0
A:=INBUS
c2
M:=INBUS
c3, c4
A:=A-M
c7
OUTBUS:=A
c0'
A:=0
c1
Q:=INBUS, Q[-1]=0;
c2
M:=INBUS, COUNT:=0;
Q[0]Q[-1]
01
10
00/11
c3
A:=A+M
c3, c4
A:=A-M
COUNT7
0
cR
A[7]:=A[7], A[6:0],Q:=A.Q[7:0];
c6
COUNT++
c7
OUTBUS:=A
c7,5
Q[0]:=0
c8
OUTBUS[7:0]:=Q[7:0]
Start/Stop
c0
A:=INBUS
c1
Q:=INBUS
c2
M:=INBUS, COUNT:=0;
cL
A.Q[7:1]:=A[6:0].Q, Q[0]:=0;
c3, c4
A:=A-M
c5
Q[0]:=~A[7]
A[7]
0
1
c3
A:=A+M
COUNT7
0
c6
COUNT++
A[7]
1
c3
A:=A+M
c7
OUTBUS:=A

fig 1.1

Componente Hardware

Shifting Registers

Pentru implementarea corectă a operațiilor de înmulțire și împărțire în cadrul ALU-ului, am folosit registre cu functionalitate de shiftare *bidirecționala* pentru stocarea valorilor lui A și Q. Aceste registre au fost proiectate pentru a permite atât deplasarea valorilor la stânga pentru impartire, cât și la dreapta pentru inmultire, însă operația de shiftare se realizează doar în cadrul algoritmilor specifici mentionati. Astfel, în timpul înmulțirii sau împărțirii, registrele A și Q sunt deplasate corespunzător pentru a executa pașii algoritmici necesari, în timp ce în restul operațiilor, valorile stocate sunt menținute constante. În schimb, pentru M, am folosit un registru simplu care nu dispune de facilitatea de shiftare, deoarece valoarea acestuia rămâne constantă pe durata întregului proces de împărțire, fără a necesita modificări.

Functionalitati Operanzi:
Adunare

- Termeni suma: **A, M** (8 biti fiecare)
- Rezultat: **A** (8 biti)

Scadere

- Descazut: **A** (8 biti fiecare)
- Scazator: **M** (8 biti)
- Rezultat: **A** (8 biti)

Inmultire

- Factori: **Q, M** (8 biti fiecare)
- Rezultat: **A.Q** (16 biti)

Impartire

- Deimpartit: **A.Q** (16 biti)
- Impartitor: **M** (8 biti)
- Cat: **Q** (8 biti)
- Rest: **A** (8 biti)

Design

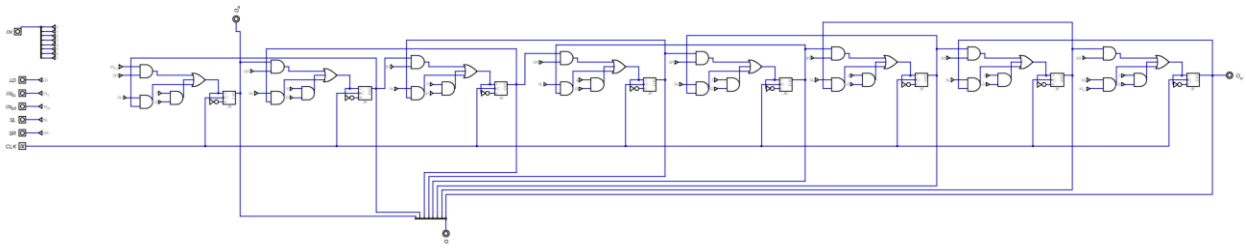


fig 2.1

Pentru implementarea shifting registerului folosit pentru stocarea fiecărui bit al operandului, am utilizat JK flip-flops asincrone, fiecare având rolul de a stoca un bit individual al operandului. Aceste flip-flopuri sunt controlate de trei semnale principale: LD (load), SL (shift left) și SR (shift right). Semnalul LD permite încărcarea valorii în registru fără a efectua operația de shift, în timp ce semnalele SL și SR controlează direcția de deplasare a valorii, respectiv la stânga și la dreapta.

Registrul este alimentat cu următoarele inputuri:

- IN: valoarea de 8 biți care este încărcată în registru atunci când semnalul LD este activ.
- IN_SL: bitul care intră în registru atunci când se face shift left (deplasare la stânga).
- IN_SR: bitul care intră în registru atunci când se face shift right (deplasare la dreapta).

Ieșirile sunt:

- O: ieșirea principală a registrului pe 8 biți.
- O_SR: bitul ieșit din registru la efectuarea unui shift right.
- O_SL: bitul ieșit din registru la efectuarea unui shift left.

În figura *fig. 2.1* este ilustrat designul complet al shifting registerului pe 8 biți, iar în figura *fig. 2.2* sunt evidențiate semnalele și inputurile, cu un zoom pe 3 biți pentru a clarifica conexiunile și modul de operare.

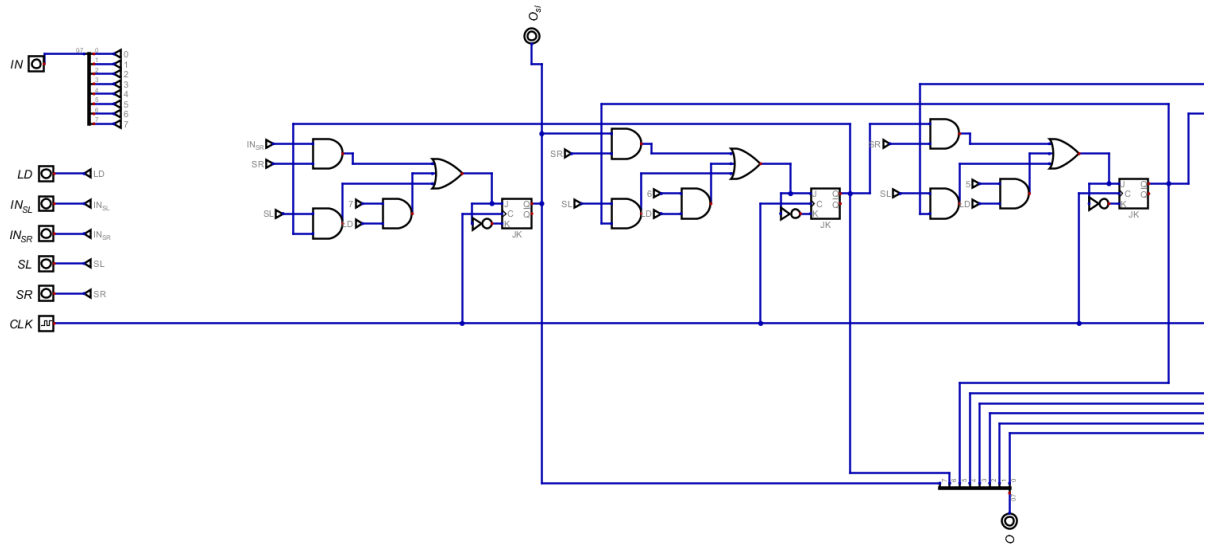


fig 2.2

Implementare in Verilog

Modulul în Verilog pentru shifting register este descris în detaliu mai jos.

```
module shf2 (
  input CLK,
  input SR,
  input SL,
  input IN_SR,
  input IN_SL,
  input [7:0] IN,
  input LD,
  output O_sr,
  output O_sl,
  output [7:0] O
);
  wire [7:0] q;
  wire [7:0] j;
  wire [7:0] k;
  wire [7:0] left_in;
  wire [7:0] right_in;

  // intrari shift left
  assign left_in[0] = IN_SL;
  assign left_in[1] = q[0];
  assign left_in[2] = q[1];
  assign left_in[3] = q[2];
  assign left_in[4] = q[3];
  assign left_in[5] = q[4];
  assign left_in[6] = q[5];
  assign left_in[7] = q[6];

  // intrari shift right
  assign right_in[0] = q[1];
  assign right_in[1] = q[2];
  assign right_in[2] = q[3];
  assign right_in[3] = q[4];
  assign right_in[4] = q[5];
  assign right_in[5] = q[6];
  assign right_in[6] = q[7];
  assign right_in[7] = IN_SR;
```

```
  genvar i;
  generate
    for (i = 0; i < 8; i = i + 1) begin : v_shft
      wire sr_part, sl_part, ld_part;

      assign sr_part = SR & right_in[i];
      assign sl_part = SL & left_in[i];
      assign ld_part = LD & IN[i];
      assign j[i] = sr_part | sl_part | ld_part;
      assign k[i] = ~j[i];

      JK_FF #(.Default(0)) FF (
        .J(j[i]),
        .K(k[i]),
        .C(CLK),
        .Q(q[i])
      );
    end
  endgenerate

  assign O = q;
  assign O_sr = q[0];
  assign O_sl = q[7];
endmodule
```

Parallel Adder

Sumatorul paralel joacă un rol esențial în implementarea tuturor celor 4 operații realizate de ALU.

Adunare: realizeaza operatia, rezultatul fiind stocat in registrul A

Scădere: la fel ca la adunare, insa operandul M este negat folosind un EXOR wordgate

Înmulțire: utilizat pentru adunarea și scăderea intermediară a valorilor in functie de Q[0] si Q[-1], conform pașilor algoritmului Booth Radix 2

Împărțire: folosit pentru a efectua scăderile succesive între valoarea A și M. Dacă rezultatul scăderii este negativ (indicând faptul că valoarea A este mai mică decât M), se aplică operația de restore asupra lui A (restituirea valorii anterioare a lui A)

De fiecare dată când se efectuează o operație de adunare, scădere sau restore, rezultatul este stocat în registrul A, actualizând valoarea acestuia pentru a fi folosit în următorul pas al calculului.

Sumatorul utilizat este un RCA (Ripple Carry Adder, *fig 2.4*) compus din 8 FAC-uri (Full Adder Cell, *fig 2.3*).

Design

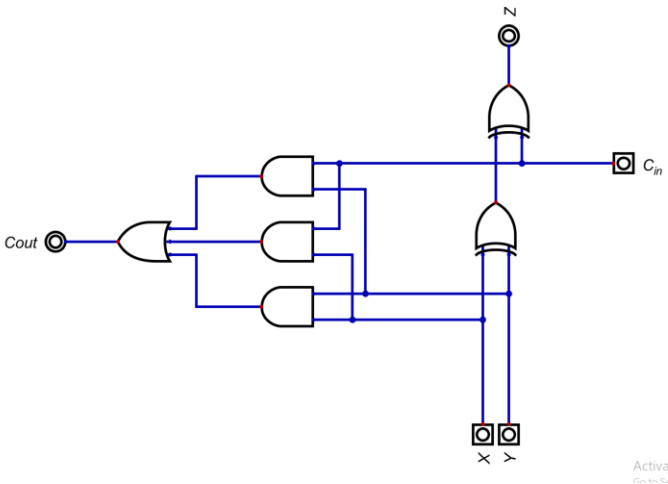


fig 2.3

Pentru implementarea Ripple Carry Adder-ului am inseriat 8 FAC-uri conectand la fiecare c_{in} c_{out}-ul de la FAC-ul precedent dupa cum se poate vedea in design (*fig2.3*) precum si in codul Verilog.

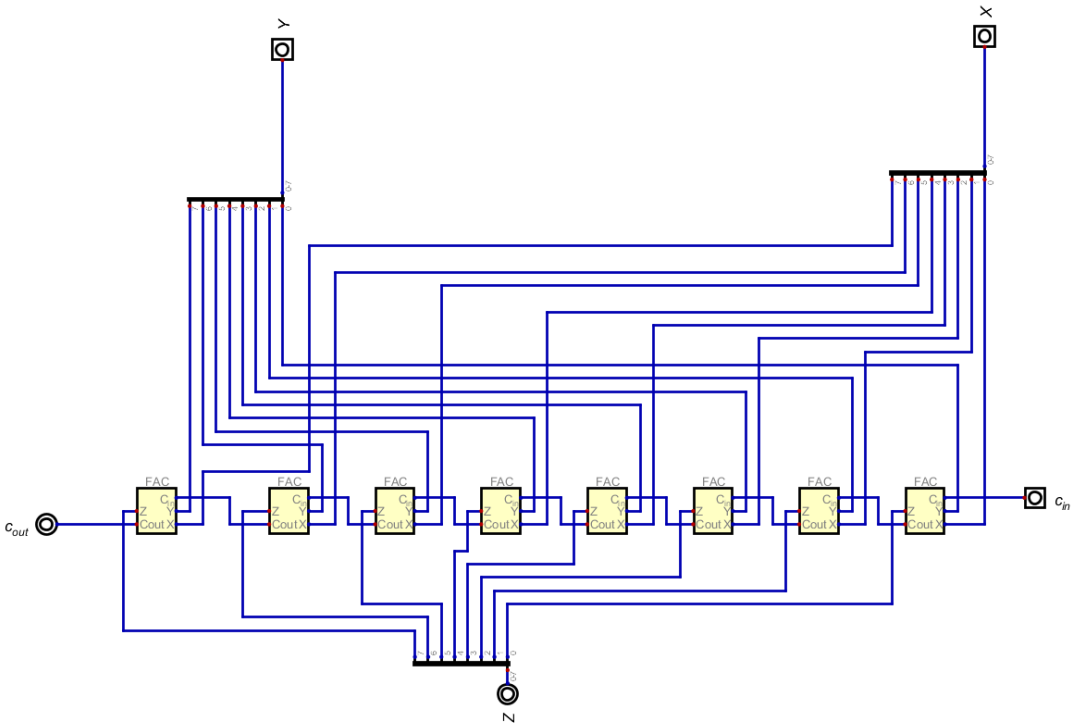


Fig 2.4

Implementare in Verilog

Modulele în Verilog pentru FAC si RCA sunt descrise în detaliu mai jos.

```
202 module FAC (
203   input X,
204   input Y,
205   input C_in,
206   output Cout,
207   output Z
208 );
209   assign Z = ((X ^ Y) ^ C_in);
210   assign Cout = ((X & Y) | (X & C_in) | (Y & C_in));
211 endmodule
```

Pentru reprezentarea in cod a modulului RCA am folosit un vector de instante ale modulului FAC.

```
module RCA8 (
  input c_in,
  input [7:0] Y,
  input [7:0] X,
  output c_out,
  output [7:0] Z
);
  wire [6:0] carry;

  genvar i;
  generate
    for (i = 0; i < 8; i = i + 1) begin : v
      if (i == 0) begin
        FAC FAC_i (
          .X(X[i]),
          .Y(Y[i]),
          .C_in(c_in),
          .Cout(carry[i]),
          .Z(Z[i])
        );
      end else if (i == 7) begin
        FAC FAC_i (
          .X(X[i]),
          .Y(Y[i]),
          .C_in(carry[i-1]),
          .Cout(c_out),
          .Z(Z[i])
        );
      end else begin
        FAC FAC_i (
          .X(X[i]),
          .Y(Y[i]),
          .C_in(carry[i-1]),
          .Cout(carry[i]),
          .Z(Z[i])
        );
      end
    end
  endgenerate
endmodule
```

Control Unit

Unitatea de Control este utilizată pentru a genera semnalele necesare controlului operațiilor ALU, în funcție de operația selectată. Controlul stărilor se realizează secvențial, în fiecare ciclu de clock, contorul avansează la următoarea stare, activând semnalele corespunzătoare pentru operația dorită.

- 00 – adunare
- 01 – scadere
- 10 – inmultire
- 11 – impartire

In cadrul Unitatii de contro avem 13 semnale de control ca iesiri, fiecare activand o alta functionalitate. Ordinea si logica utilizarii acestora se poate urmari mai clar pe ordinograma (fig 1.1).

- c0 – incarca de pe inbus valoarea lui A
- c0’ – incarca in A valoarea 0 (inmultire)
- c1 – incarca de pe inbus valoarea lui Q si il seteaza pe Q[-1] pe 0 (inmultire)
- c2 – incarca de pe inbus valoarea lui M si initializeaza counterul cu 0 (inmultire si impartire)
- c3 - suma
- c4 – neaga valoarea lui M, folosit in acelasi timp cu c3 pentru scadere
- c5 – Q[0]=~A[7] (impartire)
- c6 – incrementeaza count-ul
- cL – shifteaza registrii A si Q la stanga
- cR - shifteaza registrii A si Q la dreapta
- c7 – incarca A pe outbus
- c7_5 – Q[0]=0 (inmultire)
- c8 – incarca Q pe outbus

Design

Sequence counter-ul este proiectat să gestioneze 6 faze distincte (Φ0, Φ1, Φ2, etc.), fiind construit dintr-un Counter pe 3 biti conectat unui Decodor 3-to-5. Design-ul acestuia este prezentat in figura de mai jos (*fig 2.5*).

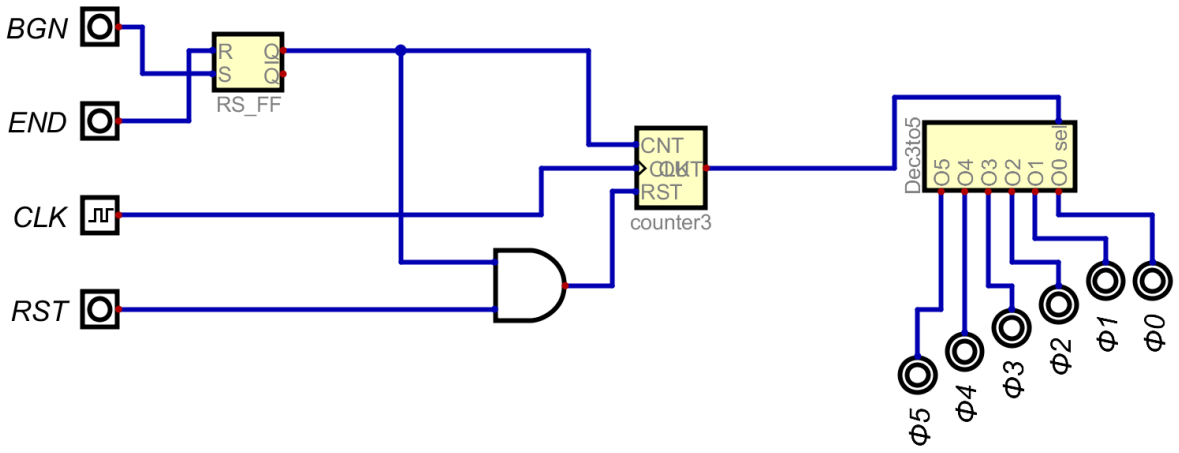


fig 2.5

În cadrul unității de control, am implementat un sistem cu 3 cicluri, fiecare fiind gestionat de câte 3 flip-flopuri RS. Fiecare ciclu controlează diferite etape ale execuției ALU:

- *Cycle0*: Acesta este primul ciclu și este responsabil pentru inițializarea registrelor. În această fază, sunt activitate semnalele necesare pentru încărcarea valorilor inițiale în registrele ALU și pentru setarea stării de start a unității de control. Acesta activeaza semnalele: **c0, c0’, c1, c2**

- *Cycle1-8*: Aceste cicluri controlează semnalele care se repetă pe parcursul operațiilor de înmulțire și împărțire. Semnale activate: **c3, c4, c5, c6, cL, cR**
- *Cycle9*: Acesta este ultimul ciclu, care se ocupă de transmiterea rezultatelor finale pe outbus. Semnale activate: **c7, c7_5, c8, c3** (pentru corectia finala in cazul impartirii)

Tranzitia intre cicluri se realizeaza in functie de ultimul ciclu activ si de faza in care ne aflam (in cazul tranzitiei intre Cycle1-8 si Cycle9 la inmultire si impartire este necesara si activarea semnalului CNT7).

Pe langa cele 3 flip-flopuri RS necesare pentru fiecare ciclu, am mai folosit un flip-flop RS pentru mentinerea semnalului c8 destul de mult timp cat sa nu fie oprit de activarea semnalului END.

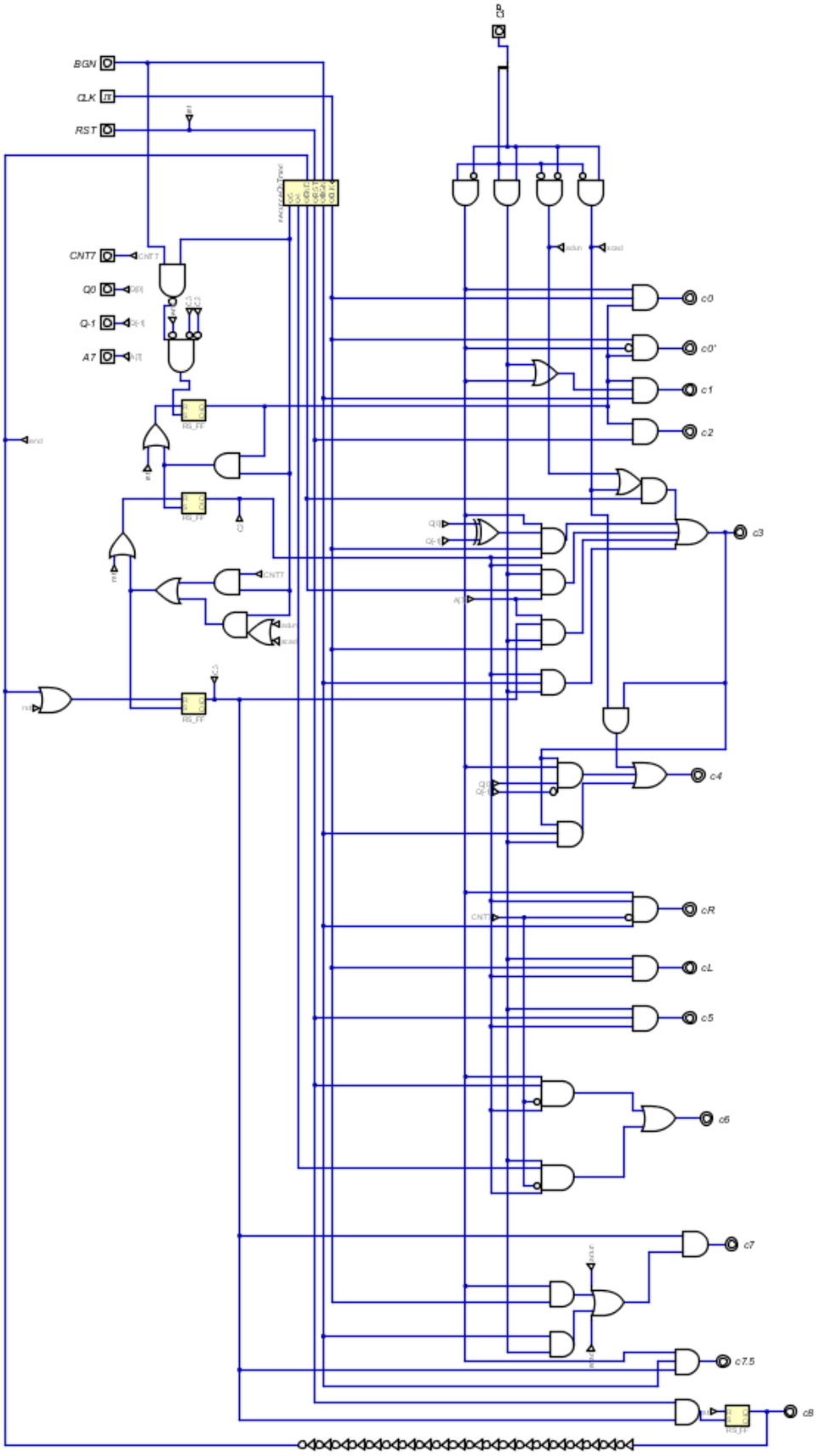
Intregul design al control unit-ului poate fi urmarit in figura *fig2.6.*

Implementare in Verilog

```
wire END ;
wire fi0;
wire fi1;
wire fi2;
wire fi3;
wire fi4;
wire fi5;
wire out_Cycle0;
wire rset_Cycle0;
wire set_Cycle0;
wire rset_Cycle_1_8;
wire set_Cycle_1_8;
wire out_Cycle_1_8;
wire rset_Cycle9;
wire set_Cycle9;
wire out_Cycle9;
wire inm;
wire imp;
wire adun;
wire scad;

wire c3_temp;
wire set_END;
wire c8_temp;

 assign inm = (~ OP[0] & OP[1]); //10
 assign imp = (OP[0] & OP[1]); //11
 assign adun = (~ OP[0] & ~ OP[1]); //00
 assign scad = (OP[0] & ~ OP[1]); //01
sequnceCNTmod sequnceCNTmod_i0 (
  .CLK( CLK ),
  .BGN( BGN ),
  .RST( RST ),
  .END ( END  ),
  .fi0 ( fi0 ),
  .fi1 ( fi1 ),
  .fi2 ( fi2 ),
  .fi3 ( fi3 ),
  .fi4 ( fi4 ),
  .fi5 ( fi5 )
);
```

```
  assign c0 = (inm & fi0 & out_Cycle0);
  assign c0_prim  = (fi0 & ~ inm & out_Cycle0);
  assign c1 = (out_Cycle0 & (imp | inm) & fi1);
  assign c3_temp = (((adun | scad) & fi3) | (inm & (Q0 ^ Q_1 ) & fi0 & out_Cycle_1_8) | (out_Cycle_1_8 & imp & fi3 &
A7) | (A7 & out_Cycle9 & imp & fi0) | (out_Cycle_1_8 & fi1 & imp));
  assign c4 = ((c3_temp & scad) | (c3_temp & inm & Q0 & ~ Q_1 ) | (c3_temp & fi1 & imp));
  assign cR = (inm & out_Cycle_1_8 & ~ CNT7 & fi1);
  assign cL = (imp & fi0 & out_Cycle_1_8);
  assign c5 = (imp & fi2 & out_Cycle_1_8);
  assign c7 = (out_Cycle9 & (adun | (inm & fi0) | (fi1 & imp) | scad));
  assign c6 = ((inm & fi2 & ~ CNT7 & out_Cycle_1_8) | (imp & fi4 & ~ CNT7 & out_Cycle_1_8));
  assign c7_5  = (inm & fi1 & out_Cycle9);
  assign set_Cycle9 = (((scad | adun) & fi5) | (fi5 & CNT7));
  assign rset_Cycle0 = (RST | set_Cycle_1_8);
  assign rset_Cycle_1_8 = (RST | set_Cycle9);
  assign rset_Cycle9 = (END  | RST);
  assign set_Cycle0 = (~ out_Cycle_1_8 & ~ out_Cycle9 & ~ END  & ~ (fi5 & BGN));

 RS_FF RS_FF_i3 ( //ff Cycle 0
   .R( rset_Cycle0 ),
   .S( set_Cycle0 ),
   .Q( out_Cycle0 )
 );

  assign c2 = (out_Cycle0 & fi2);
  assign set_Cycle_1_8 = (fi5 & out_Cycle0);

  RS_FF RS_FF_i4 ( //ff Cycle 1-8
   .R( rset_Cycle_1_8 ),
   .S( set_Cycle_1_8 ),
   .Q( out_Cycle_1_8 )
 );

  assign END = c8_temp;
  RS_FF RS_FF_i2 ( //ff Cycle 9
   .R( rset_Cycle9 ),
   .S( set_Cycle9 ),
    .Q( out_Cycle9 )
 );

   RS_FF RS_FF_i1 ( //ff care mentine semnalul c8 cat sa se activeze END
   .R( RST ),
   .S( set_END ),
   .Q( c8_temp )
 );

 assign set_END = (fi2 & out_Cycle9);
 assign c3 = c3_temp;
 assign c8 = c8_temp;
endmodule
```

fig 2.6



Design-ul complet ALU

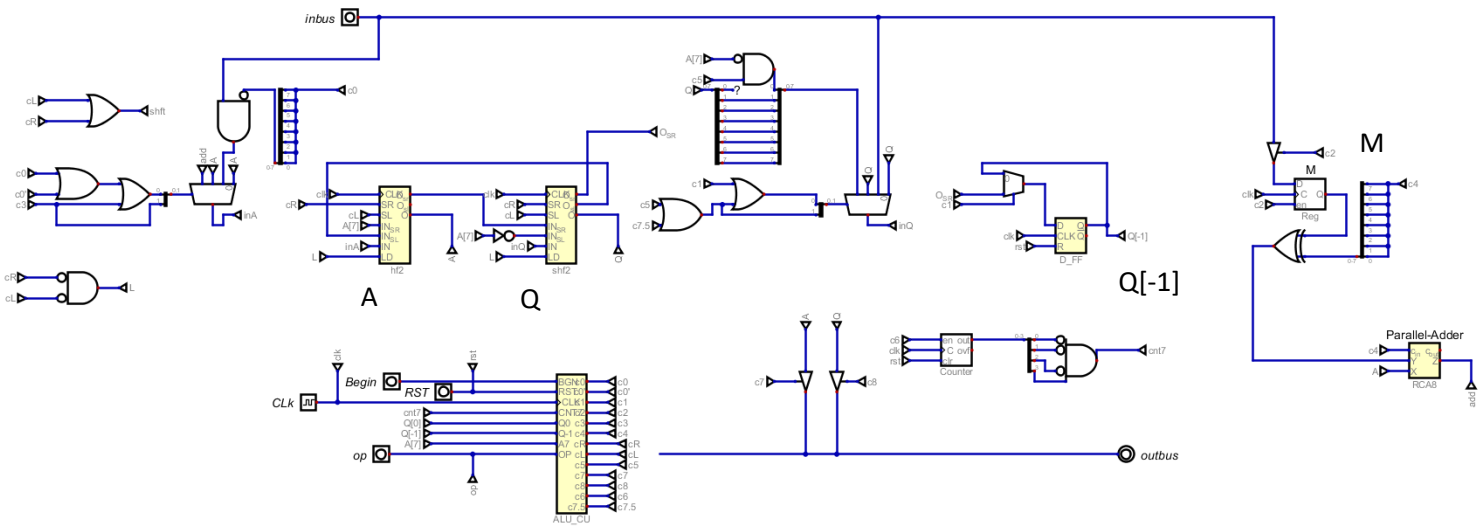


fig2.7

În cadrul designului complet al ALU-ului, modulele individuale — registrele de shiftare, sumatorul și unitatea de control — sunt integrate într-o arhitectură coerentă, coordonată de semnalele de control.

Simularea magistralelor

Pentru a simula magistralele de intrare (inbus) am folosit Buffere Tri-state care incarca registrii cu valorile date la inbus la momentul activarii semnalelor de control corespunzatoare in cazul lui M. Pentru a pune valori pe outbus am folosit aceeasi logica in cazul registrilor A si Q. In cazul in care bufferul nu primeste un semnal de activare acesta da la output *impedanta ridicata*.

```
173 module DriverBus#(parameter Bits = 2)(
174     input [(Bits-1):0] in,
175     input sel,
176     output [(Bits-1):0] out
177 );
178     assign out = (sel == 1'b1)? in : {Bits{1'bz}};
179 endmodule
```

O alta metoda de care ne-am folosit pentru a simula magistrala de intrare este utilizarea unui Multiplexor pentru incarcarea registrilor A si Q. Această soluție a permis selectarea dinamică a valorii încărcate în registre, în funcție de semnalele de control primite. De exemplu, în registrul A se putea încărca fie valoarea zero, fie valoarea citită de pe magistrala de intrare, fie rezultatul operației aritmetice realizate de sumator.

Implementare in Verilog

- Declararea wire-rurilor

```
505 module ALU_2 (
506   input [7:0] inbus,
507   input [1:0] op,
508   input CLk,
509   input Begin ,
510   input RST,
511   output [7:0] outbus
512 );
513   wire cnt7;
514   wire Q_1 ;
515   wire c0;
516   wire c0_prim ;
517   wire c1;
518   wire c2;
519   wire c3;
520   wire c4;
521   wire cR;
522   wire cL;
523   wire c5;
524   wire c7;
525   wire c8;
526   wire c6;
527   wire c7_5 ;
528   wire [7:0] inbus_A; //input de pe inbus pt A, cand c0 activ e 0, cand nu ia de pe inbus
529   wire [7:0] A; //A din registru
530   wire [7:0] inA; //iesire mux A
531   wire [7:0] Q;
532   wire [7:0] Q_new;
533   wire [7:0] inQ;
534   wire [7:0] M; //ce vine de pe inbus in driver
535   wire [7:0] M_reg; //ce vine din driver in registrul M
536   wire [1:0] sel_muxA; //conditie mux A
537   wire [1:0] sel_muxQ; //conditie mux Q
538   wire [7:0] addordif_term; //termenul pt adunare sau scadere
539   wire [7:0] sum; //rezultatul dat de parallel adder
540   wire [7:0] XOR_mask;
541   wire bit_sh_left;
542   wire L;
543   wire bit_sh_right;
544   wire o_sr; //iasa din Q
545   wire [3:0] O_cnt; //output counter
546   wire out_muxQ_1;
547   wire [7:0] en_bus_A;
548
```

- Instantiere Control Unit

```
//control unit

ALU_CU ALU_CU_i0 (
  .BGN( Begin  ),
  .RST( RST ),
  .CLK( CLk ),
  .CNT7( cnt7 ),
  .Q0( Q[0] ),
  .Q_1 ( Q_1  ),
  .A7( A[7]  ),
  .OP( op ),
  .c0( c0 ),
  .c0_prim ( c0_prim  ),
  .c1( c1 ),
  .c2( c2 ),
  .c3( c3 ),
  .c4( c4 ),
  .cR( cR ),
  .cL( cL ),
  .c5( c5 ),
  .c7( c7 ),
  .c8( c8 ),
  .c6( c6 ),
  .c7_5 ( c7_5  )
);

assign L = (~ cR & ~ cL);
assign shft = (cL | cR);

```

- Instantiere registrii

```
//A

assign inbus_A = ( ~en_bus_A & inbus);
assign en_bus_A={c0, c0, c0, c0, c0, c0, c0, c0};

assign sel_muxA[0] = ((c0 | c0_prim ) | c3);
assign sel_muxA[1] = c3;

 Mux_4to1 #(
  .Bits(8)
)
Mux_4to1_A (
  .sel( sel_muxA ),
  .in_0( A ),
  .in_1( inbus_A ),
  .in_2( A ),
  .in_3( sum ),
  .out( inA )
);

shf2 Reg_A (
  .CLK( CLk ),
  .SR( cR ),
  .SL( cL ),
  .IN_SR( A[7]  ),
  .IN_SL( bit_sh_left ),
  .IN( inA ),
  .LD( L ),
  .O_sr( bit_sh_right ),
  .O( A )
);
```

```
//Q

assign sel_muxQ[0] = (c1 | c5 | c7_5 );
assign sel_muxQ[1] = (c5 | c7_5 );

Mux_4to1 #(
  .Bits(8)
)
Mux_4to1_Q (
  .sel( sel_muxQ ),
  .in_0( Q ),
  .in_1( inbus ),
  .in_2( Q ),
  .in_3( Q_new ),
  .out( inQ )
);


shf2 Reg_Q (
  .CLK( CLk ),
  .SR( cR ),
  .SL( cL ),
  .IN_SR( bit_sh_right ),
  .IN_SL( ~A[7] ),
  .IN( inQ ),
  .LD( L ),
  .O_sr( o_sr ),
  .O_sl( bit_sh_left ),
  .O( Q )
);

 assign Q_new={ Q[7], Q[6], Q[5], Q[4],  Q[3], Q[2], Q[1], ~A[7] & c5 };
```

```
//Q[-1]

  Mux_2to1 Mux_Q_1 (
  .sel( shft | c1 ),
  .in_0( Q_1  ),
  .in_1(o_sr ),
  .out( out_muxQ_1 )
);

D_FF D_FF_Q_1 (
  .D( out_muxQ_1 ),
  .CLK( CLk ),
  .R( RST ),
  .Q( Q_1  )
);
```

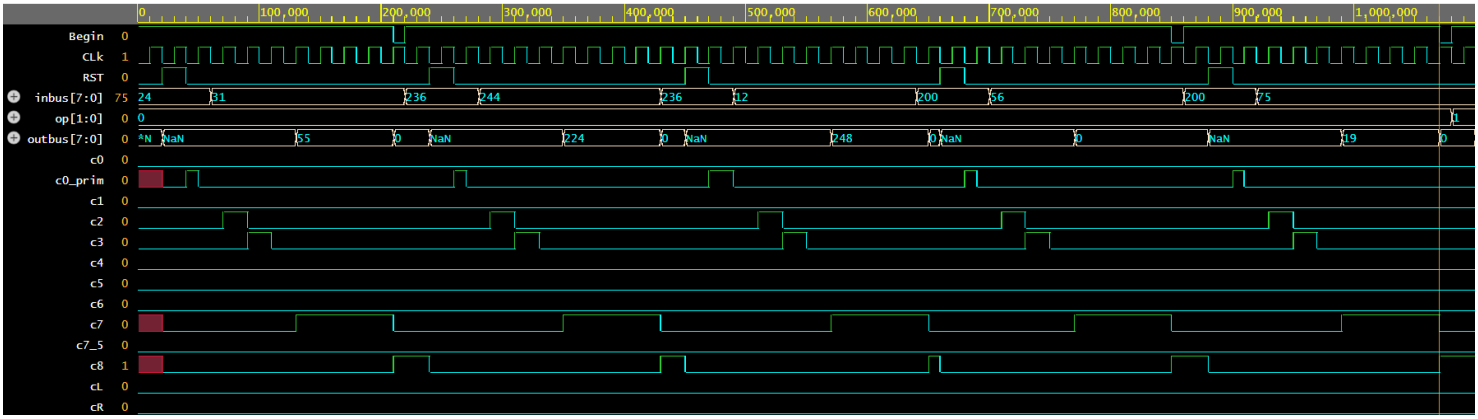
```
659    //M
660
661   DriverBus #(
662     .Bits(8)
663   )
664   DriverBus_M_in (
665     .in( inbus ),
666     .sel( c2 ),
667     .out( M )
668   );
669
670     Register #(
671     .Bits(8)
672   )
673   Register_M (
674     .D( M ),
675     .C( CLk ),
676     .en( c2 ),
677     .Q( M_reg )
678   );
```

Testare si forme de unda

Cu ajutorul platformei Eda Playground, am simulat prin intermendiul unor testbench-uri relevante cele 4 operatii pentru a testa functionalitatea, corectitudinea algoritmilor si pentru a verifica cazurile limita.

Valorile obtinute se urmaresc cu ajutorul waveform-urilor generate de platforma.

Adunare



In cazul adunarii s-a realizat un testbench pentru a testa functionalitatea acesteia luandu-se in considerare cazurile posibile

- Doua numere pozitive reprezentate pe 8 biti: 24 + 31 = 55
- Doua numere negative reprezentate pe 8 biti:

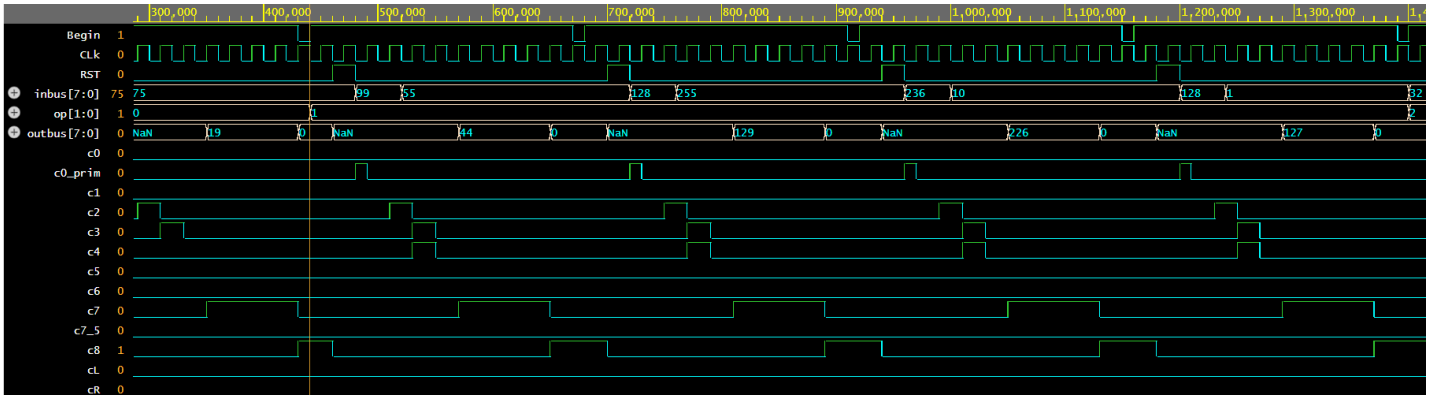
-20(`8b'11101100`) + (-12)(`8b'11110100`) = -32(`8b'11100000`)

- Un numar pozitiv si unul negativ:

-20(8b'11101100) + 12 = -8(8b'11111000)

- Cazul de overflow: valorile stocandu-se in registrii pe 8 biti, daca rezultatul adunarii unor numere pozitive este mai mare decat 2^8-1 (255) aceasta nu va fi reprezentata corespunzator
  200 + 56 = 0 ;
  200 + 75 = 19 ;
  Valorile mai mari decat limita admisa vor vi reprezentate prin *suma obtinuta – 256 (ex: 275-256=19)*

Scadere



In cazul scaderii s-a realizat un testbench prin care se testeaza functionalitatea avand in vedere cazurile posibile:

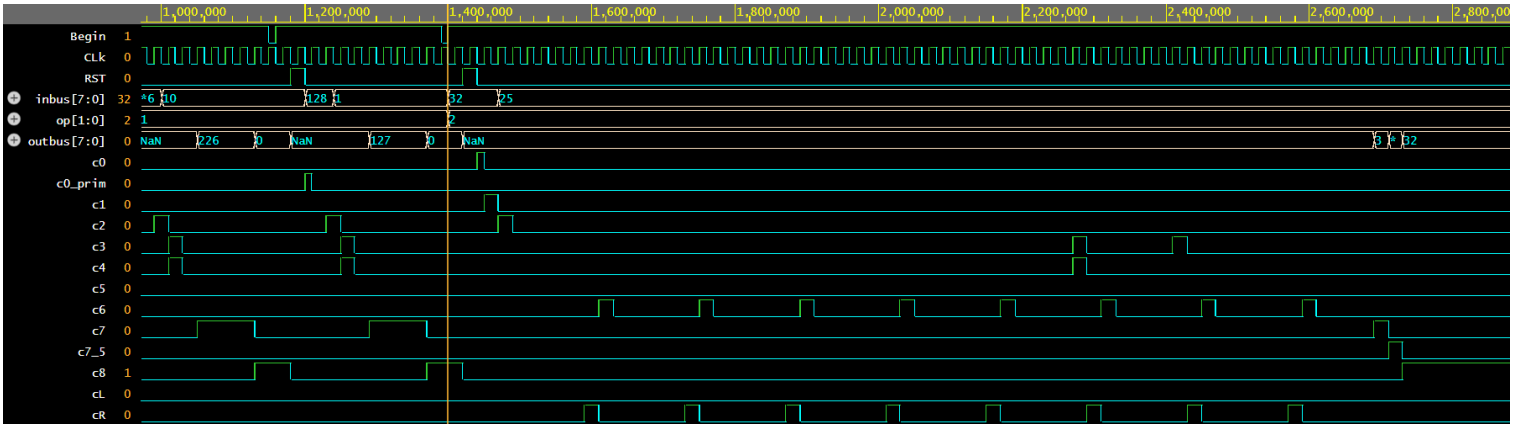
- Doua numere pozitive:
  99 – 55 = 44
- Doua numere negative:
  -128(8b'10000000) – (-1)(8b'11111111) = -127(8b'10000001)
- Un numar negativ si unul pozitiv:
  -20(8b'11101100) – 10 = -30(8b'11100010)
- Cazul de overflow/underflow: valorile se stocheaza intr-un registru de 8 biti, iar cel mai mic numar reprezentat pe 8 biti este -128 (8b'10000000). Daca se incearca sa se reprezinte orice valoare mai mica decat aceasta, atunci vom avea underflow. Putem intampina acest caz atunci cand incercam sa scadem dintr-un numar negativ un numar pozitiv, iar suma lor in modul e mai mare decat 128. Overflow-ul se manifesta la fel ca la adunare in momentul in care dintr-un numar pozitiv se scade un numar negativ rezultand o valoare. Daca depasim 255 atunci avem overflow.

  -128(8b'10000000) – 1 = - 1 – acest caz poate fi considerat unul de underflow, rezultatul corect fiind -129, insa 128 si -128 au aceeasi reprezentare pe 8 biti in C2 deci calculul poate fi interpretat si ca 128-1=127.

Inmultire

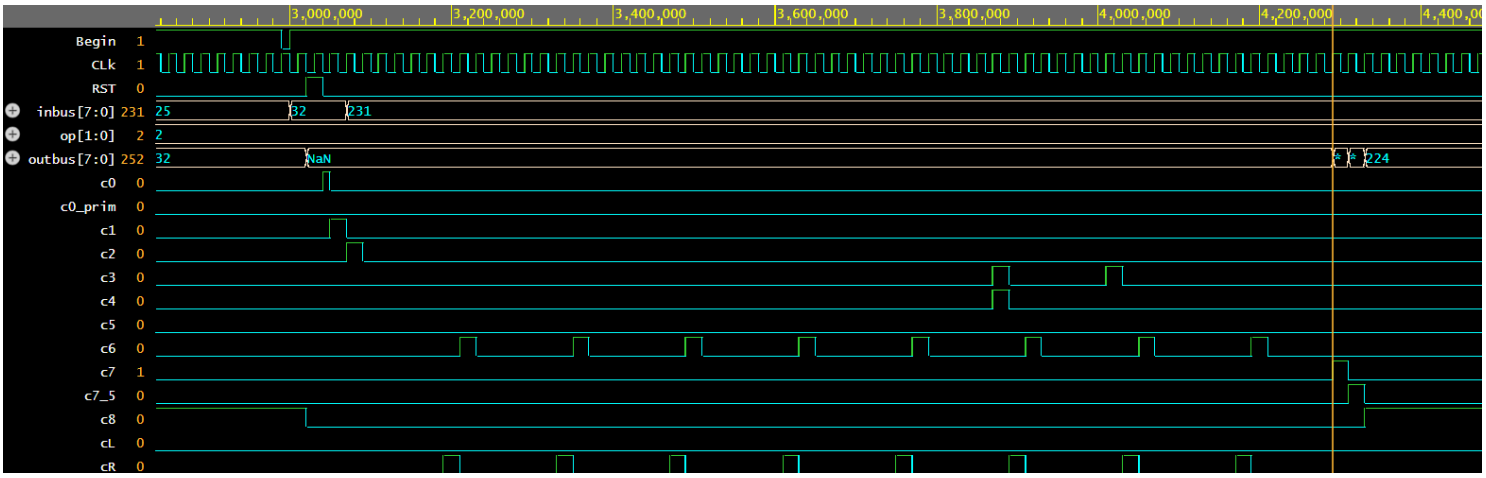
In cazul inmultirii s-a realizat un testbench prin care se testeaza functionalitatea algoritmului Booth Radix 2 avand in vedere cazurile posibile:

- Doua numere pozitive:



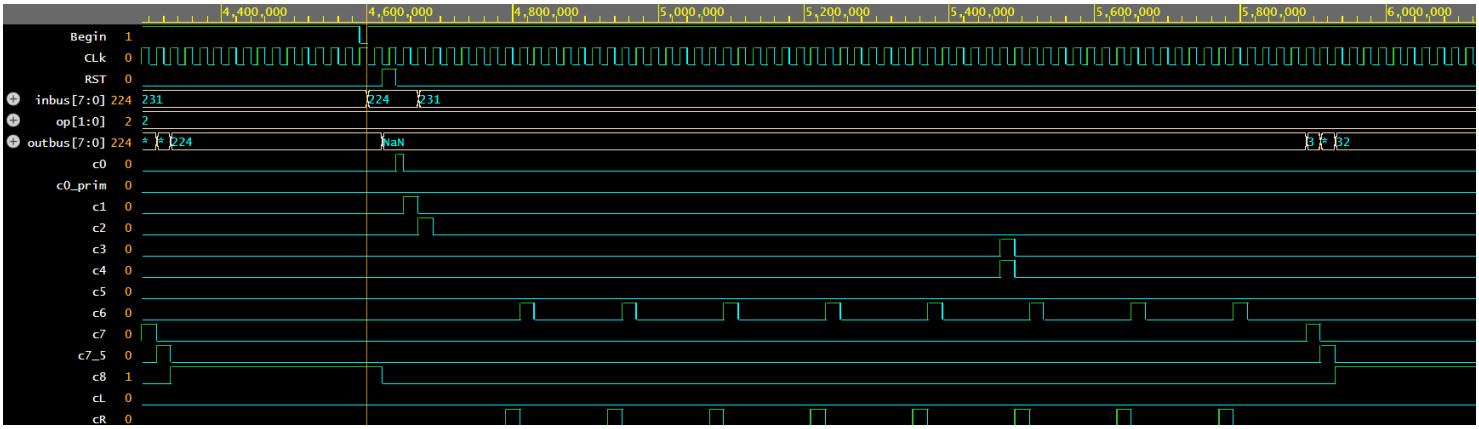
25 x 32 = 16b'00000011 00100000 (800)

- Un numar pozitiv si unul negativ:



32 x (-25) = 16b'11111100 11100000 (-800)

- Doua numere negative:

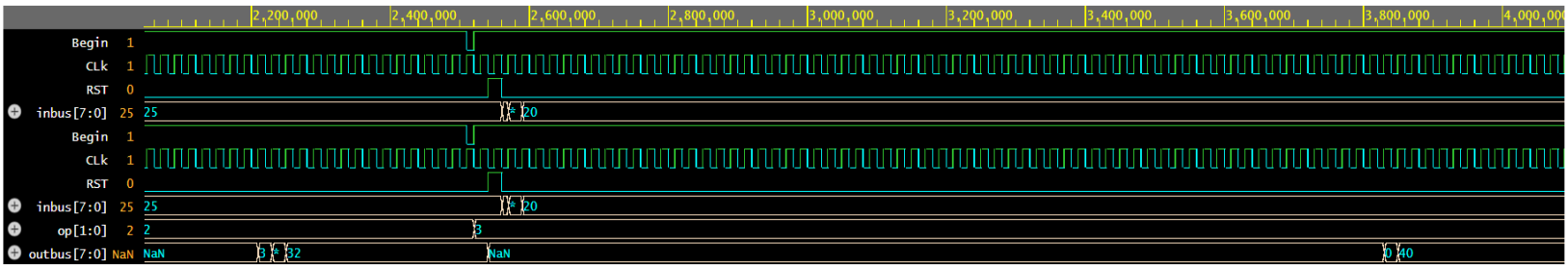


-25 x (-32) = 16b'00000011 00100000 (800)

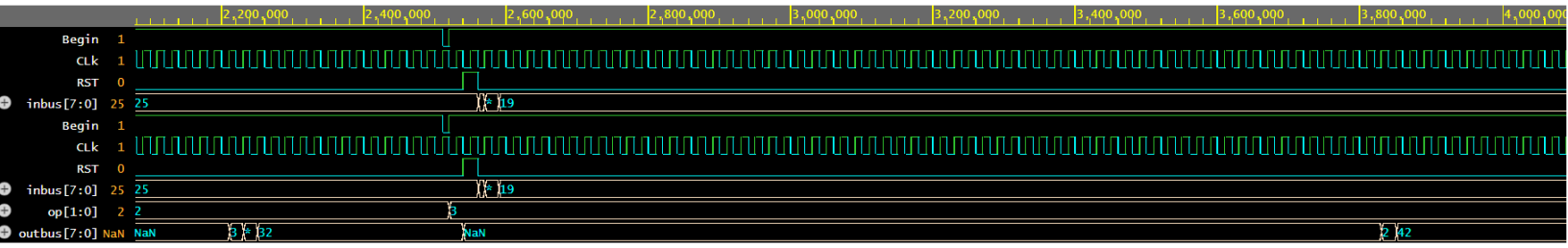
- Cazul de overflow: Avand in vedere ca produsul se stocheaza pe 16 biti(registrii A si Q concatenati), valoarea maxima posibila este 2^{16}-1(65535), insa valoarea maxima efectiva este 255*255(65025), asadar in cazul inmultirii nu exista posibilitate de overflow.

Impartire

Avand in vedere ca algoritmul folosit pentru impartire este Restoring Division, toti operanzii sunt considerati pozitivi. Nu exista overflow in cazul impartirii.



800(16b'00000011 00100000) / 20(8b'00010100) =40(8b'00101000), rest 0



800(16b'00000011 00100000) / 19(8b'00010011) = 42(8b'00101010) , rest 2

In cazul in care se efectueaza impartirea la 0, algoritmul are un comportament nedeterminat.

Concluzie

Proiectul a urmărit proiectarea și implementarea unei unități aritmetico-logice (ALU) capabile să execute operațiile de adunare, scădere, înmulțire și împărțire, folosind descriere hardware în limbajul Verilog. Arhitectura propusă a fost structurată modular, incluzând componente precum registre, sumator, unitate de control și multiplexoare, toate integrate într-un design coerent, coordonat prin semnale de control. Fiecare operație a fost testată printr-un testbench dedicat, acoperind cazuri particulare precum operanzi de semn opus, condiții de overflow și underflow, precum și comportamente limită în reprezentarea pe 8 biți în complement față de doi. Rezultatele simulării validează funcționalitatea corectă a ALU-ului și demonstrează eficiența arhitecturii propuse în ceea ce privește performanța logicii de control.