Arithmetical Logic Unit (ALU)

Horvath Maximilian Mark
Stochean Liviana
Szatmari Levente-Alex

8-bit ALU Project - CN PROJECT

Descriere generală

-> Acest proiect reprezintă implementarea unui ALU (Arithmetic Logic Unit) pe 8 biți, dezvoltat pentru proiectul din cadrul cursului de Calculatoare Numerice, dar și pentru dezvoltarea cunoștințelor. Design-ul hardware a fost realizat utilizând DIGITAL (.dig) pentru simulări și implementare schematică, iar codul Verilog aferent a fost scris și verificat în ModelSim.

-> ALU-ul suportă o varietate de operații aritmetice și logice, precum:

- Adunare, scădere, înmulțire și împărțire.
- Operații de shiftare
- Comparații simple (zero, semn, etc.)

Structură proiect

- DIGITAL (.dig): schemă completă a ALU-ului, inclusiv datapath și control unit.
- Verilog (.v): module HDL pentru simulare și testare în ModelSim.
- Testbenches: fișiere de test automate pentru validarea funcționalității.

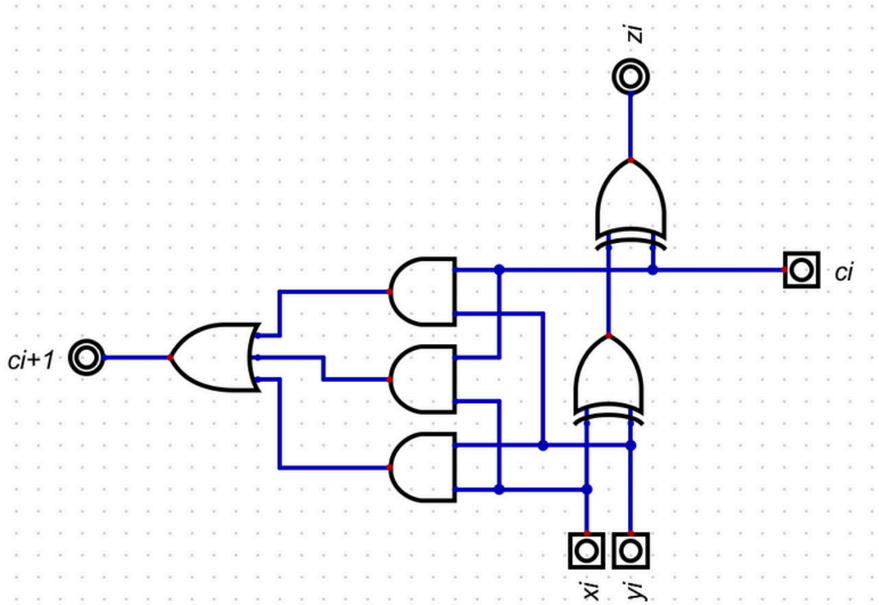
Cerințe

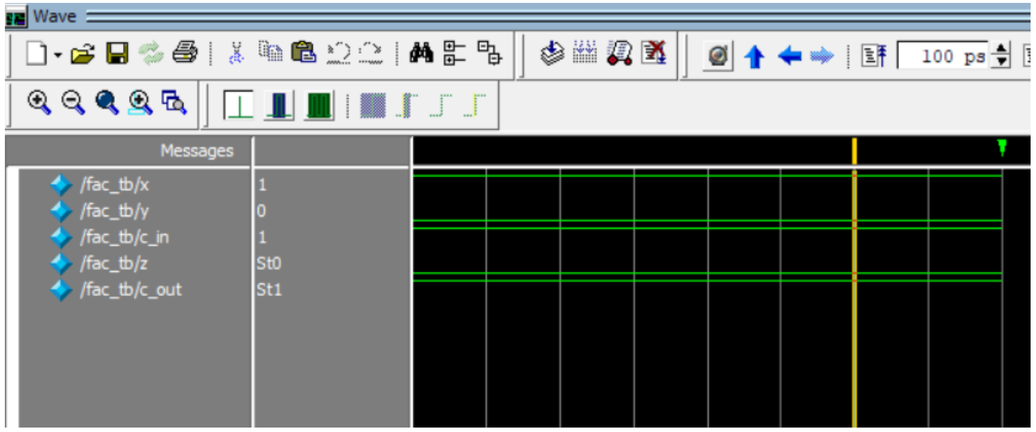
- DIGITAL (.dig) - pentru deschiderea și simularea fișierelor .dig
- ModelSim - pentru simularea modulelor Verilog

Studenți

- Horvath Maximilian-Mark
- Stochean Liviana-Maria-Cristiana
- Szatmari Levente-Alex

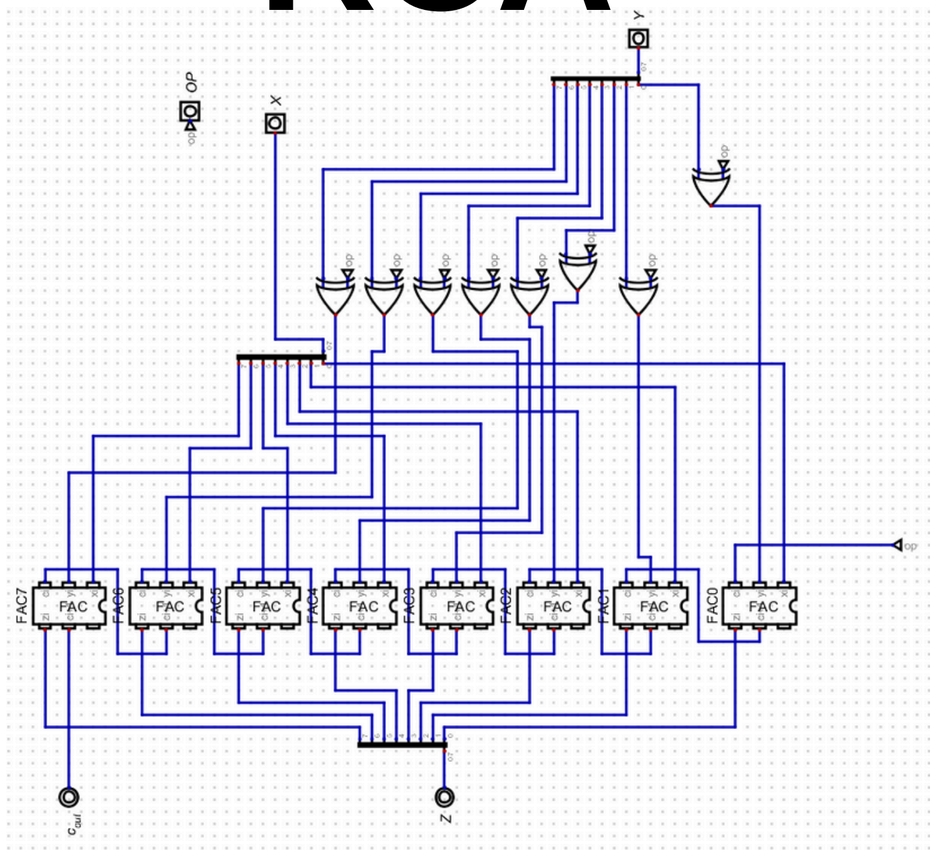
FAC

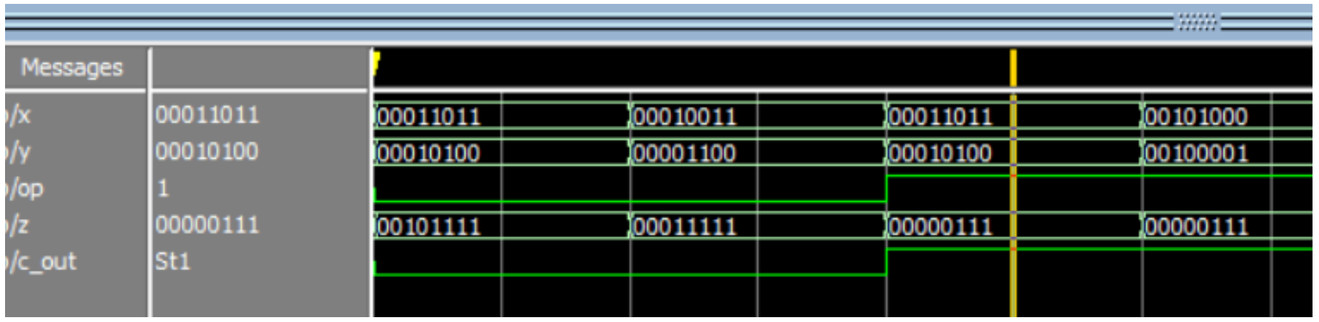





```
module fac (
    input x,
    input y,
    input c_in,
    output z,
    output c_out
);
    assign z=(x^y)^c_in;
    assign c_out=(x&y)|(x&c_in)|(y&c_in);
endmodule

module fac_tb;
  reg x;
  reg y;
  reg c_in;
  wire z;
  wire c_out;

  fac cut (
    .x(x),
    .y(y),
    .c_in(c_in),
    .z(z),
    .c_out(c_out)
  );

  initial begin
    $dumpfile("dump.vcd"); $dumpvars;
    $display("c_in\tx\t\ty\t\tz\tc_out");
    $monitor("%b\t%b\t%b\t%b\t%b",c_in,x,y,z,c_out);

    // Test

    x = 1;
    y = 0;
    c_in = 1;
    #10;

    $finish;
  end
endmodule
```

RCA

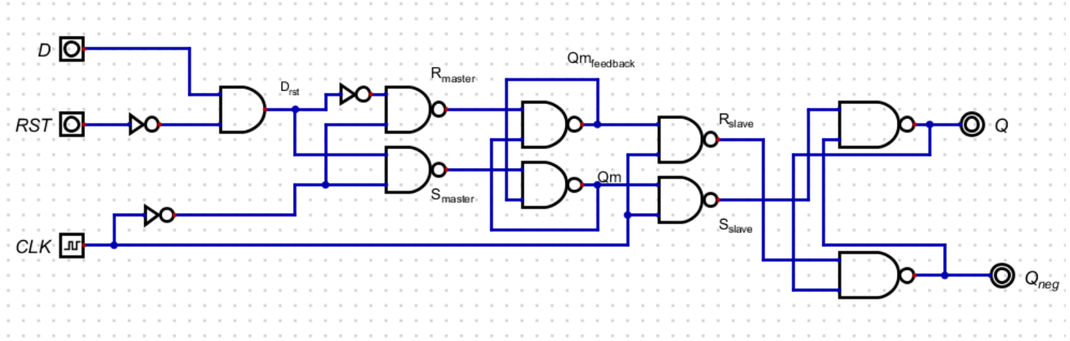


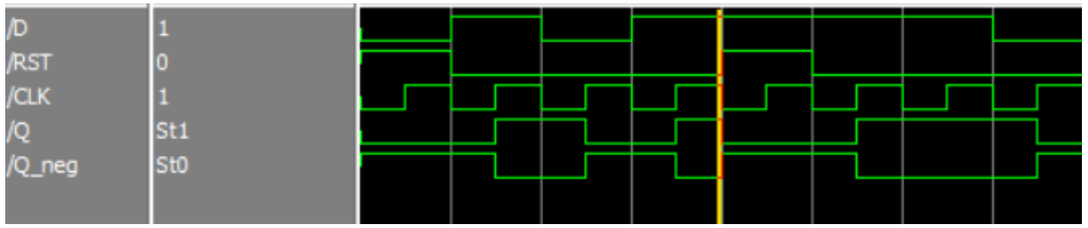



```
`include "fac.v"

module rca8b (
  input [7:0] x, y,
  input c_in, op,  //cand op=0 -> adunare, op=1 -> scadere
  output [7:0] z,
  output c_out
);

  wire [7:0] leg;
  wire [7:0] y_scadere;
  assign y_scadere=op?~y:y; //daca op=1, inverseaza bitii lui y, altfel y ramane neschimbat

  fac fac0(.x(x[0]), .y(y_scadere[0]), .c_in(op),    .z(z[0]), .c_out(leg[0]));
  fac fac1(.x(x[1]), .y(y_scadere[1]), .c_in(leg[0]),   .z(z[1]), .c_out(leg[1]));
  fac fac2(.x(x[2]), .y(y_scadere[2]), .c_in(leg[1]),   .z(z[2]), .c_out(leg[2]));
  fac fac3(.x(x[3]), .y(y_scadere[3]), .c_in(leg[2]),   .z(z[3]), .c_out(leg[3]));
  fac fac4(.x(x[4]), .y(y_scadere[4]), .c_in(leg[3]),   .z(z[4]), .c_out(leg[4]));
  fac fac5(.x(x[5]), .y(y_scadere[5]), .c_in(leg[4]),   .z(z[5]), .c_out(leg[5]));
  fac fac6(.x(x[6]), .y(y_scadere[6]), .c_in(leg[5]),   .z(z[6]), .c_out(leg[6]));
  fac fac7(.x(x[7]), .y(y_scadere[7]), .c_in(leg[6]),   .z(z[7]), .c_out(c_out));

endmodule
```

```
module rca8b_tb;
  reg [7:0] x, y;
  reg op;
  wire [7:0] z;
  wire c_out;

  rca8b cut(
    .x(x),
    .y(y),
    .op(op),
    .z(z),
    .c_out(c_out)
  );

  initial begin
    //adunare
    op=0; //facem operatia de adunare
    x=8'b00011011;  // x=27
    y=8'b00010100;  // y=20
    #10;
    $display("Adunare: %b+%b=%b, c_out=%b",x,y,z,c_out);

    //adunare
    op=0; //facem operatia de adunare
    x=8'b00010011;  //x=19
    y = 8'b00001100;  //y=12
    #10;
    $display("Adunare: %b+%b=%b, c_out=%b",x,y,z,c_out);
    //scadere
    op=1; //facem operatia de scadere
    x=8'b00011011; // x=27
    y=8'b00010100; // y=20
    #10;
    $display("Scadere: %b-%b=%b, c_out=%b",x,y,z,c_out);

    //scadere
    op=1; //facem operatia de scadere
    x=8'b00101000; // x=40
    y=8'b00100001; // y=33
    #10;
    $display("Scadere: %b-%b=%b, c_out=%b",x,y,z,c_out);

    $finish;
  end
```

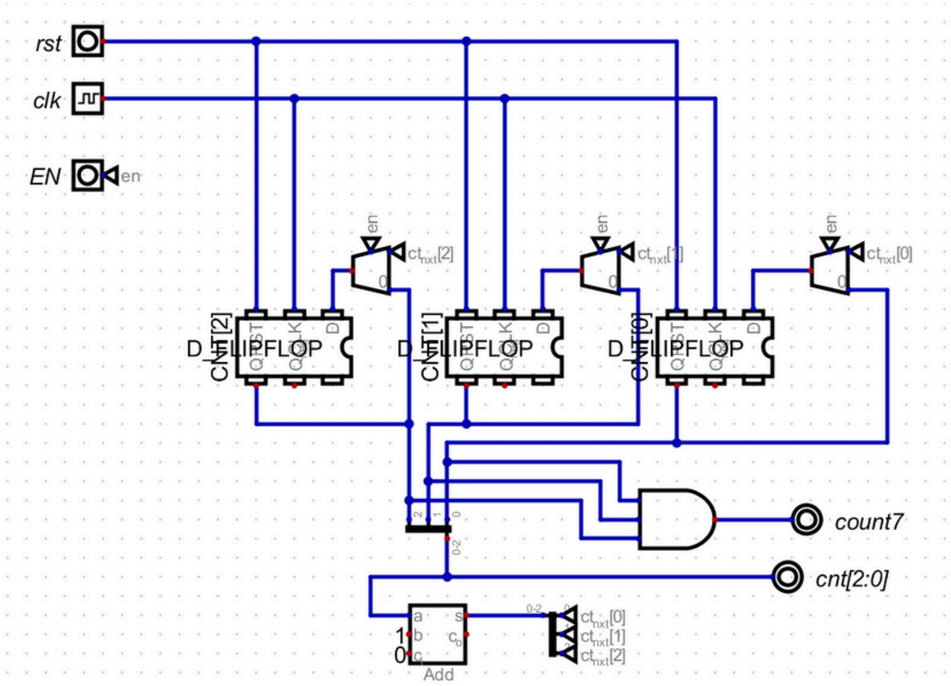
FLIP-FLOP

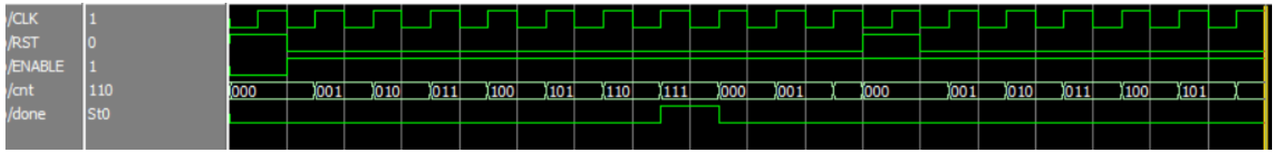





```
module d_ff (
  input wire D,
  input wire CLK,
  input wire RST,
  output reg Q,
  output wire Q_neg
);
  assign Q_neg = ~Q;

  //la front pozitiv de CLK sau RST, daca RST=1 -> Q=0, altfel, Q primeste D
  always @(posedge CLK or posedge RST)begin
    if(RST)
      Q<=1'b0;
    else
      Q<=D;
  end
endmodule

module d_ff_tb;

    reg D,RST,CLK;
    wire Q,Q_neg;

    d_ff cut (
        .D(D),
        .RST(RST),
        .CLK(CLK),
        .Q(Q),
        .Q_neg(Q_neg)
    );

    always #5 CLK = ~CLK; //genereaza un CLK care se inverseaza la fiecare 5ns

    initial begin
        $display("Time\tCLK\tRST\tD\tQ\tQ_neg");
        $monitor("%0dns\t%b\t%b\t%b\t%b\t%b",$time,CLK,RST,D,Q,Q_neg);

        CLK=0; //initializeaza la 0
        RST=1; D=0; #10; //se da reset activ timp de 10ns, D nu conteaza cand resetul e 1

        RST=0; D=1; #10; //se dezactiveaza resetul si pune D=1, se testeaza setarea D
        //se schimba D sa verifice comutarea valorii pe D
        D=0; #10;
        D=1; #10;

        RST=1; #10; //se face iar reset
        RST=0; #10; //se dezactiveaza resetul
        //se testeaza din nou D
        D=1; #10;
        D=0; #10;

        $finish;
    end
endmodule
```

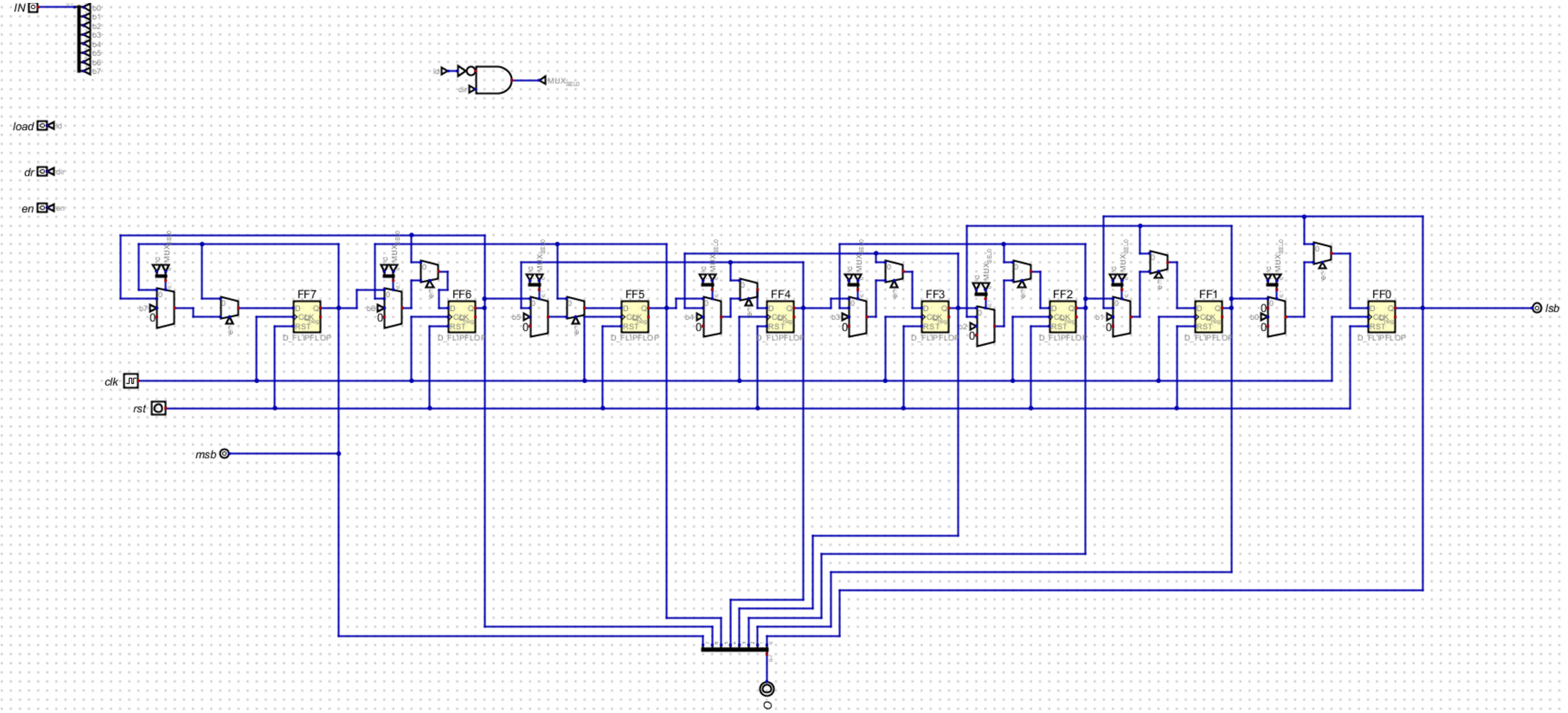
COUNTER

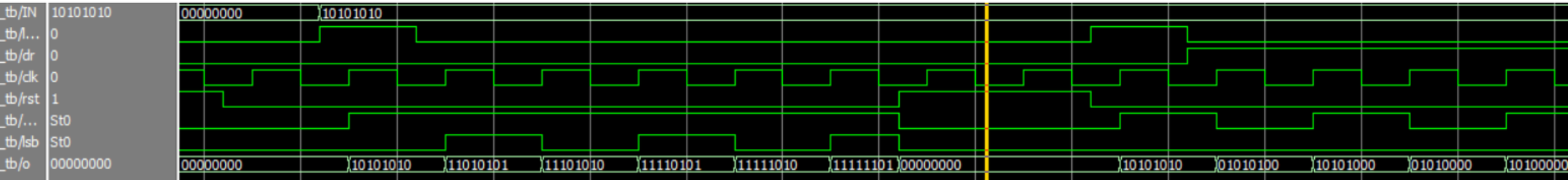





```
module d_ff (
  input wire D,
  input wire CLK,
  input wire RST,
  output reg Q,
  output wire Q_neg
);

  assign Q_neg = ~Q;

  //la front pozitiv de CLK sau RST, daca RST=1 -> Q=0, altfel, Q primeste D

  always @(posedge CLK or posedge RST)begin
    if(RST)
      Q<=1'b0; //seteaza Q la 0
    else
      Q<=D;    //Q primeste D
  end
endmodule

module counter (
    input wire CLK,
    input wire RST,
    input wire ENABLE,
    output wire [2:0] cnt,
    output wire done
);

    wire [2:0] ct_nxt;
    wire [2:0] ct;
    wire [2:0] q_neg;
    wire [2:0] add_out;
    wire [2:0] d;
    wire done_cnt;

    assign cnt=ct; //iesire contor
    assign done_cnt=(ct==3'b111); //done cand contorul ajunge la 7
    assign done=done_cnt;
    assign add_out=ct+3'b001; // incrementare cu 1
    assign ct_nxt=add_out;

    //MUX pentru fiecare bit: daca ENABLE=1, foloseste ct_nxt, altfel retine vechiul ct
    assign d[0]=(ENABLE)?ct_nxt[0]:ct[0];
    assign d[1]=(ENABLE)?ct_nxt[1]:ct[1];
    assign d[2]=(ENABLE)?ct_nxt[2]:ct[2];

    //Instantierea flip-flopurilor D
    d_ff dff0(.D(d[0]), .CLK(CLK), .RST(RST), .Q(ct[0]), .Q_neg(q_neg[0]));
    d_ff dff1(.D(d[1]), .CLK(CLK), .RST(RST), .Q(ct[1]), .Q_neg(q_neg[1]));
    d_ff dff2(.D(d[2]), .CLK(CLK), .RST(RST), .Q(ct[2]), .Q_neg(q_neg[2]));

endmodule

`timescale 1ns/1ps //seteaza unitatea de timp la 1 ns si precizia la 1 ps

module counter_tb;

    reg CLK, RST, ENABLE;
    wire [2:0] cnt;
    wire done;

    counter cut (
        .CLK(CLK),
        .RST(RST),
        .ENABLE(ENABLE),
        .cnt(cnt),
        .done(done)
    );

    initial begin
        CLK=0; //initializeaza CLK cu 0
        forever #5 CLK = ~CLK; //genereaza un CLK care se inverseaza la fiecare 5ns
    end

    initial begin
        $display("Time\tCLK\tRST\tENABLE\tCNT\tDone");
        $monitor("%0dns\t%b\t%b\t%b\t%03b\t%b", $time, CLK, RST, ENABLE, cnt, done);

        RST=1;ENABLE=0; // reset activ
        #10;
        RST=0;ENABLE=1; // pornire contor

        repeat(10) #10;

        RST=1; #10; // reset la mijloc
        RST=0; #10;
        repeat(5) #10;

        $finish;
    end
endmodule
```

BIDIRECTIONAL SHIFT REGISTER





BIDIRECTIONAL SHIFT REGISTER

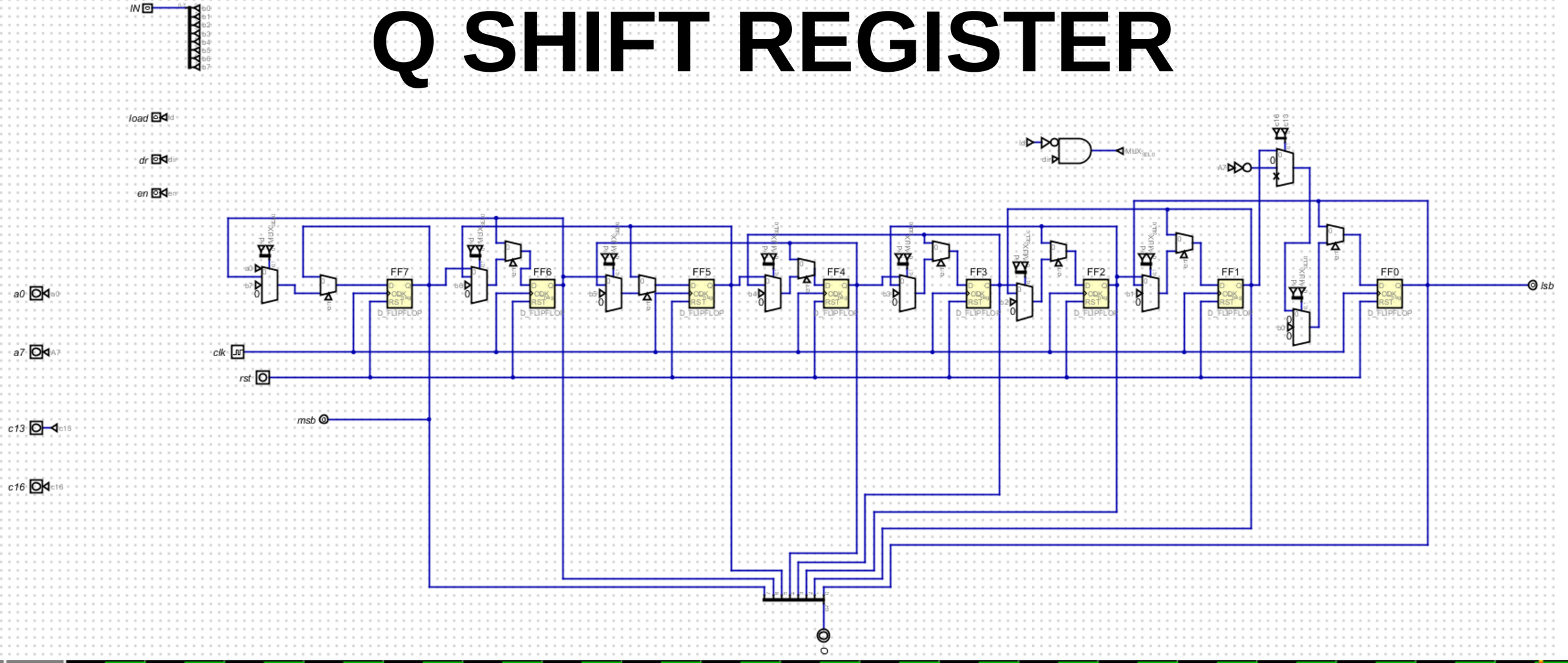
```
module Mux_4x1(
    input [1:0] sel,
    input in_0, in_1, in_2, in_3,
    output reg out
);
    //selecteaza una dintre cele 4 intrari in functie de sel
    always @(*) begin
        case(sel)
            2'b00: out = in_0;
            2'b01: out = in_1;
            2'b10: out = in_2;
            2'b11: out = 1'b0;
        endcase
    end
endmodule
```

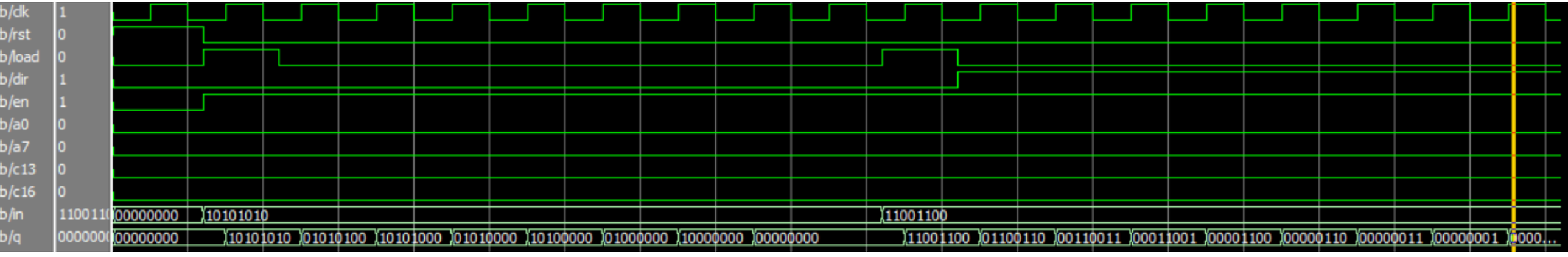
```
module D_FLIPFLOP (
    input D, CLK, RST,
    output reg Q,
    output wire Q_neg
);
    assign Q_neg = ~Q;

  //la front pozitiv de CLK sau RST, daca RST=1 -> Q=0, altfel, Q primeste D
    always @(posedge CLK or posedge RST) begin
        if(RST)
            Q<=0;
        else
            Q<=D;
    end
endmodule
```

```
module shift_reg_bidir (
    input [7:0] IN,
    input load, dr, clk, rst,
    output msb, lsb,
    output [7:0] o
);
    wire [7:0] d, q;
    wire [1:0] sel={load,~load&dr};

    assign lsb=q[0]; //salveaza cel mai putin semnificativ bit din q
    assign msb=q[7]; //salveaza cel mai semnificativ bit din q
    assign o=q;

    //fiecare mux alege intre shift stanga,dreapta,load
    genvar i;
    generate
        for (i=0;i<8;i=i+1)begin : muxes //in fiecare pozitie i, se creeaza un mux care poate sa
            wire in_left=(i==7)?msb:q[i+1]; //shiftare stanga, din dreapta vecinului q[i+1]
            wire in_right=(i==0)?lsb:q[i-1]; //shiftare dreapta, din stânga vecinului q[i-1]
            wire load_val=IN[i]; //valoare din IN[i]

            Mux_4x1 cut ( .sel(sel), .in_0(in_left), .in_1(in_right), .in_2(load_val), .in_3(1'b0), .out(d[i]) );
            D_FLIPFLOP cutt ( .D(d[i]), .CLK(clk), .RST(rst), .Q(q[i]) );
        end
    endgenerate
endmodule
```

```
module shift_reg_bidir_tb;
    reg [7:0] IN;
    reg load, dr, clk, rst;
    wire msb, lsb;
    wire [7:0] o;

    shift_reg_bidir cut (.IN(IN), .load(load), .dr(dr), .clk(clk), .rst(rst),.msb(msb), .lsb(lsb), .o(o));

    initial begin
        clk=0; //initializeaza CLK cu 0
        forever #5 clk = ~clk; //genereaza un CLK care se inverseaza la fiecare 5ns
    end

    initial begin
        $display("Time\tRST\tLOAD\tDR\tIN\t\tOUT\t\tMSB\tLSB");
        $monitor("%t\t%b\t%b\t%b\t%b\t%b\t%b\t%b",$time,rst,load,dr,IN,o,msb,lsb);

        //se da rst=1, nu se face load sau shift, datele de intrare sunt 0, apoi 12ns si rst=0
        rst=1;load=0;dr=0;IN=8'b00000000;
        #12 rst=0;

        // incarcam valoarea
        #10 IN=8'b10101010;load=1;
        #10 load=0;
        // shiftam dreapta
        #10 dr=0; #40;
        rst=1; #10;

        // shiftam stanga
        #10 rst=0; load=1;
        #10 load=0; dr=1; #40;

        $finish;
    end
endmodule
```

Q SHIFT REGISTER



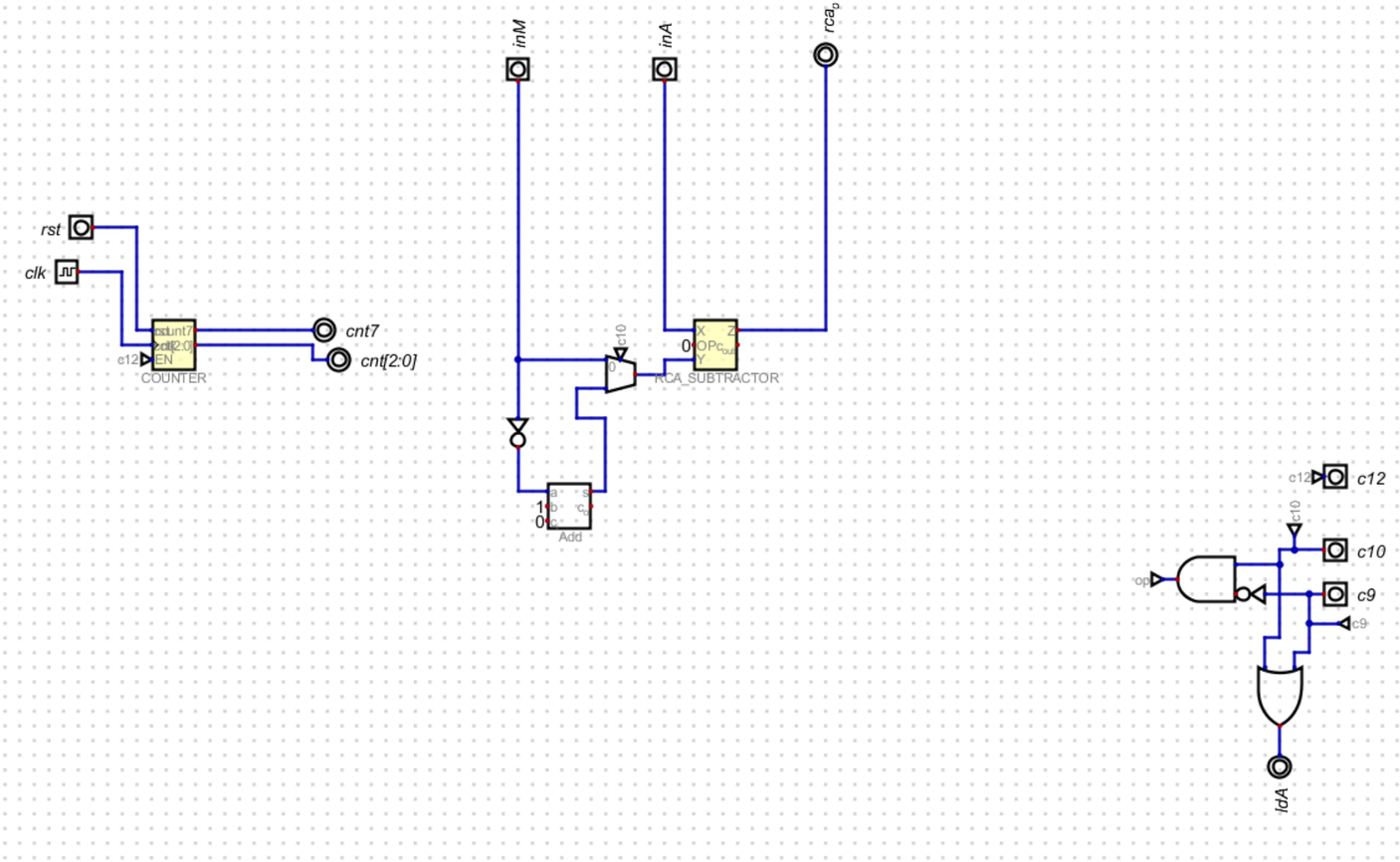


Q SHIFT REGISTER

```
module d_ff(
    input wire D,
    input wire CLK,
    input wire RST,
    output reg Q,
    output wire Q_neg
);
    assign Q_neg = ~Q;

    //la front pozitiv de CLK sau RST, daca RST=1 -> Q=0, altfel, Q primeste D
    always @(posedge CLK or posedge RST) begin
        if(RST)
            Q<=1'b0;
        else
            Q<=D;
        end
endmodule


module Q_shift_reg(
    input wire clk,
    input wire rst,
    input wire load,
    input wire dir,
    input wire en,
    input wire a0,
    input wire a7,
    input wire c13,
    input wire c16,
    input wire [7:0] in,
    output wire [7:0] q
);

    wire [7:0] d_in; // semnalele de intrare catre flip-flop uri

    //fiecare intrare pentru fiecare flip-flop in functie de load/shift/en
    genvar i;
    generate
        for(i=0; i<8;i=i+1)begin:gen_dff
            wire left_in=(i==7)?a7:q[i+1]; //pentru shiftare la stanga, daca e ultimul bit, foloseste a7 ca margine
            wire right_in=(i==0)?a0:q[i-1]; //pentru shiftare la dreapta, daca e primul bit, foloseste a0 ca margine
            wire mux_shift=dir?left_in:right_in; //alege intre left sau right in functie de dir
            assign d_in[i]=(en)?(load?in[i]:mux_shift):q[i]; //decide ce intrare merge la flip-flop..daca en=1-> daca load=1: in[i], altfel mux_shift
                                                             //daca en=0-> pastreaza vechiul q[i]

            d_ff cutt ( .D(d_in[i]), .CLK(clk), .RST(rst), .Q(q[i]), .Q_neg() );
        end
    endgenerate
endmodule
```

```
module Q_shift_reg_tb;
    reg clk, rst, load, dir, en;
    reg a0, a7, c13, c16;
    reg [7:0] in;
    wire [7:0] q;

    Q_shift_reg cut ( .clk(clk), .rst(rst), .load(load), .dir(dir), .en(en), .a0(a0), .a7(a7), .c13(c13), .c16(c16), .in(in), .q(q) );

    initial begin
        clk=0; //initializeaza CLK cu 0
        forever #5 clk = ~clk; //genereaza un CLK care se inverseaza la fiecare 5ns
    end

    initial begin
        $display("Time\tCLK\tRST\tLOAD\tDIR\tEN\tA0\tA7\tC13\tC16\tIN\t\tQ");
        $monitor("%0dns\t%b\t%b\t%b\t%b\t%b\t%b\t%b\t%b\t%b\t%08b\t%08b",$time,clk,rst,load,dir,en,a0,a7,c13,c16,in,q);

        //se da rst=1, nu se face load sau shift, datele de intrare sunt 0, a0,a7,c13,c16 sunt 0, apoi 12ns si rst=0
        rst=1;load=0;dir=0;en=0;a0=0;a7=0;c13=0;c16=0;in=8'b0;
        #12 rst=0;

        //initial
        en=1;load=1; in=8'b10101010;
        #10 load=0;

        //shiftare dreapta
        dir=0;a0=0;a7=0;
        repeat(8) #10;

        //load
        load=1;in=8'b11001100;
        #10 load=0;

        //shiftare stanga
        dir=1;a0=0;a7=0;
        repeat(8) #10;

        $finish;
    end
endmodule
```

BOOTH RADIX 2



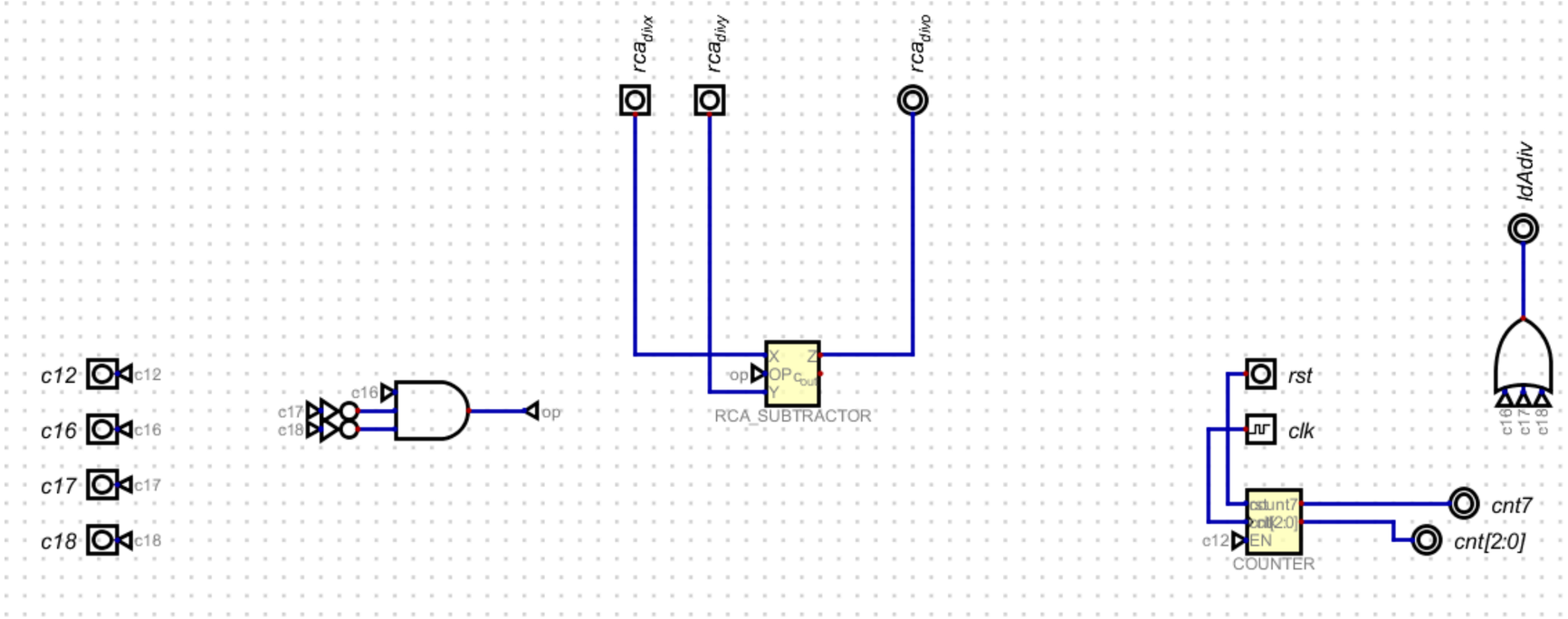
BOOTH RADIX 2

```
`timescale 1ns/1ps

`include "D_FLIPFLOP.v"

`include "Mux_2x1.v"

`include "COUNTER.v"

`include "RCA_SUBTRACTOR.v"

// Multiplicator Booth Radix-2: iterativ A,Q,Q0 cu 8 cicluri
module booth_radix_2 (
  input  wire        clk,
  input  wire        rst,
  input  wire        start,     // load initial A=0, Q=multB, Q0=0
  input  wire        enable,    // step signal (from Control Unit)
  input  wire [7:0]  multA,
  input  wire [7:0]  multB,
  output wire        done,      // pulses high when count==7, then stays high
  output wire [15:0] product,
  output wire        Q0,        // LSB of Q
  output wire        Q0_1       // âminus-1â bit
);

  // 1) Count 8 cycles, then latch done
  wire [2:0] count;
// Contor de 3 biti: numara ciclurile si genereaza done dupa 8 iteratii
  COUNTER cnt (
    .CLK    (clk),
    .RST    (rst),
    .ENABLE (enable),   // counter itself stops at 7 internally
    .count  (count),
    .done   (done)
  );
  // we only iterate while enable is high AND not yet done

  wire iter = enable && !done;

  // 2) Partial-product regs
  wire [7:0] A_hold, Q_hold;
  wire       Q0_hold;
  wire [7:0] A_next, Q_next;
  wire       Q0_next;
  wire [7:0] A_shift, Q_shift;
  wire       Q0_shift;

  // decide add/sub based on {Q[0],Q0}
  wire do_add =  Q_hold[0] & ~Q0_hold;  // 01
  wire do_sub = ~Q_hold[0] &  Q0_hold;  // 10

  // sum and difference
  wire [7:0] sum_add, sum_sub;
// Sumator/rezistor ripple-carry: realizeaza add sau sub in functie de op
  RCA_SUBTRACTOR add_i (
    .x     (A_hold),
    .y     (multA),
    .c_in  (1'b0),
    .op    (1'b0),
    .z     (sum_add),
    .c_out()
  );
// Sumator/rezistor ripple-carry: realizeaza add sau sub in functie de op
  RCA_SUBTRACTOR sub_i (
    .x     (A_hold),
    .y     (multA),
    .c_in  (1'b1),
    .op    (1'b1),
    .z     (sum_sub),
    .c_out()
  );
```

```
// Sumator/rezistor ripple-carry: realizeaza add sau sub in functie de op
  RCA_SUBTRACTOR sub_i (
    .x     (A_hold),
    .y     (multA),
    .c_in  (1'b1),
    .op    (1'b1),
    .z     (sum_sub),
    .c_out()
  );

  // 3) Select A_next = hold / add / sub
  genvar i;
  generate
    for (i = 0; i < 8; i = i + 1) begin : sel_addsub
      wire stage1;
// Multiplexor 2-la-1 pe un bit: selecteaza intre cele doua intrari
      Mux_2x1 m0(.sel(do_add), .in_0(A_hold[i]), .in_1(sum_add[i]), .out(stage1));
// Multiplexor 2-la-1 pe un bit: selecteaza intre cele doua intrari
      Mux_2x1 m1(.sel(do_sub), .in_0(stage1),    .in_1(sum_sub[i]), .out(A_next[i]));
    end
  endgenerate

  // 4) Arithmetic right-shift {A_next,Q_hold,Q0_hold}
  assign A_shift  = { A_next[7],    A_next[7:1] };
  assign Q_shift  = { A_next[0],    Q_hold[7:1] };
  assign Q0_shift =           Q_hold[0]   ;

  // 5a) iterate-gate: when !iter hold old, when iter shift
  wire [7:0] A_iter, Q_iter;
  wire       Q0_iter;
  generate
    for (i = 0; i < 8; i = i + 1) begin : gen_iter
// Multiplexor 2-la-1 pe un bit: selecteaza intre cele doua intrari
      Mux_2x1 mA(.sel(iter), .in_0(A_hold[i]), .in_1(A_shift[i]), .out(A_iter[i]));
// Multiplexor 2-la-1 pe un bit: selecteaza intre cele doua intrari
      Mux_2x1 mQ(.sel(iter), .in_0(Q_hold[i]), .in_1(Q_shift[i]), .out(Q_iter[i]));
    end
  endgenerate
// Multiplexor 2-la-1 pe un bit: selecteaza intre cele doua intrari
  Mux_2x1 mQ0(.sel(iter), .in_0(Q0_hold), .in_1(Q0_shift), .out(Q0_iter));

  // 5b) start-gate: when start load initial, else take iterate result
  wire [7:0] A_load, Q_load;
  wire       Q0_load;
  generate
    for (i = 0; i < 8; i = i + 1) begin : gen_start
      // A_load = start ? 0        : A_iter
// Multiplexor 2-la-1 pe un bit: selecteaza intre cele doua intrari
      Mux_2x1 msA(.sel(start), .in_0(1'b0), .in_1(A_iter[i]), .out(A_load[i]));
      // Q_load = start ? multB[i] : Q_iter
// Multiplexor 2-la-1 pe un bit: selecteaza intre cele doua intrari
      Mux_2x1 msQ(.sel(start), .in_0(multB[i]), .in_1(Q_iter[i]), .out(Q_load[i]));
    end
  endgenerate
// Multiplexor 2-la-1 pe un bit: selecteaza intre cele doua intrari
  Mux_2x1 msQ0(.sel(start), .in_0(1'b0), .in_1(Q0_iter), .out(Q0_load));

  // 6) Clock the registers
  generate
    for (i = 0; i < 8; i = i + 1) begin : regs
// Flip-flop D: stocheaza un bit la flanc pozitiv, cu reset asincron
      D_FLIPFLOP dA(.D(A_load[i]), .CLK(clk), .RST(rst), .Q(A_hold[i]), .Q_neg());
// Flip-flop D: stocheaza un bit la flanc pozitiv, cu reset asincron
      D_FLIPFLOP dQ(.D(Q_load[i]), .CLK(clk), .RST(rst), .Q(Q_hold[i]), .Q_neg());
    end
```

```
`timescale 1ns/1ps
`include "../src/D_FLIPFLOP.v"
`include "../src/Mux_2x1.v"
`include "../src/COUNTER.v"
`include "../src/RCA_SUBTRACTOR.v"
`include "../src/booth_radix_2.v"

module booth_radix_2_tb;
  // Inputs
  reg         clk;
  reg         rst;
  reg         start;
  reg         enable;
  reg  [7:0] multA;   // now matches DUT port
  reg  [7:0] multB;   // now matches DUT port

  // Outputs
  wire        done;
  wire [15:0] product;

  // Instantiate the DUT
// Multiplicator Booth Radix-2: iterativ A,Q,Q0 cu 8 cicluri
  booth_radix_2 uut (
    .clk     (clk),
    .rst     (rst),
    .start   (start),
    .enable  (enable),
    .multA   (multA),
    .multB   (multB),
    .done    (done),
    .product (product)
  );

  // Clock: 10 ns period
  initial clk = 0;
  always #5 clk = ~clk;

  initial begin
    $dumpfile("booth_radix_2_tb.vcd");
    $dumpvars(0, booth_radix_2_tb);

    // Test Case 1: 15 * 3 = 45
    rst    = 1; start = 0; enable = 0; multA = 0; multB = 0;
    #12;
    rst    = 0;                                  // release reset
    #8;
    start  = 1; multA = 8'd15; multB = 8'd3;
    #10;
    start  = 0;    // latch inA, inB
    enable = 1;    // begin 8-cycle shift/add sequence
    repeat (8) @(posedge clk);
    enable = 0;
    #10;
    $display("TC1: %0d * %0d = %0d (done=%b)", multA, multB, product, done);
    if (product !== 16'd45 || !done) $error("TC1 FAILED");

    // Test Case 2: 7 * 8 = 56
    #10;
    rst    = 1; start = 0; enable = 0;
    #12;
    rst    = 0;
    #8;
    start  = 1; multA = 8'd7; multB = 8'd8;
    #10;
    start  = 0;
    enable = 1;
    repeat (8) @(posedge clk);
    enable = 0;
    #10;
    $display("TC2: %0d * %0d = %0d (done=%b)", multA, multB, product, done);
    if (product !== 16'd56 || !done) $error("TC2 FAILED");

    // Test Case 3: 255 * 255 = 65025
    #10;
    rst    = 1; start = 0; enable = 0;
    #12;
    rst    = 0;
    #8;
    start  = 1; multA = 8'hFF; multB = 8'hFF;
    #10;
    start  = 0;
    enable = 1;
    repeat (8) @(posedge clk);
    enable = 0;
    #10;
    $display("TC3: %0d * %0d = %0d (done=%b)", multA, multB, product, done);
    if (product !== 16'd65025 || !done) $error("TC3 FAILED");

    $finish;
  end
endmodule
```

RESTORING DIVISION

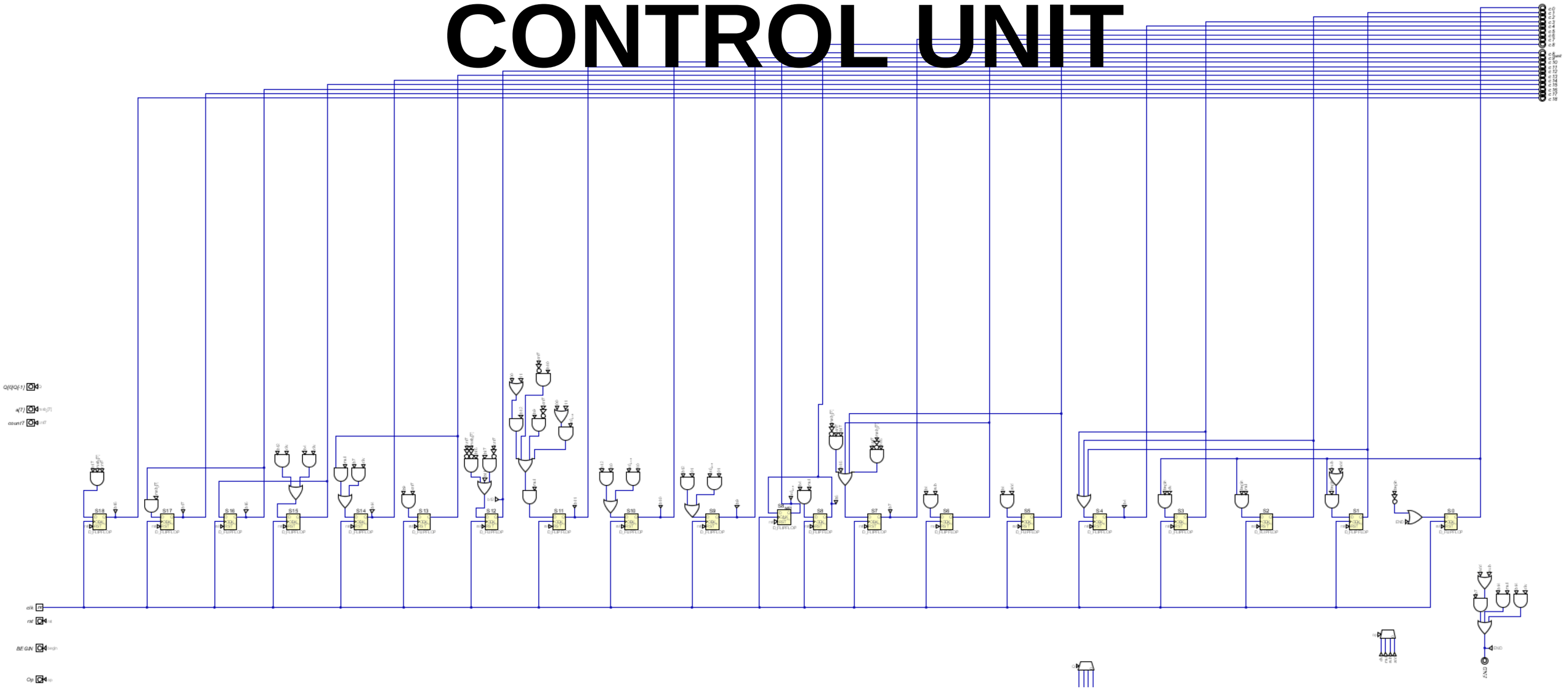


RESTORING DIVISION

```
`timescale 1ns/1ps
`include "D_FLIPFLOP.v"
`include "Mux_2x1.v"
`include "COUNTER.v"
`include "RCA_SUBTRACTOR.v"

module restoring_div (
  input  wire        clk,
  input  wire        rst,
  input  wire        start,      // load Q = dividend, R = 0, Qm1 = 0
  input  wire        enable,     // step signal (from Control Unit)
  input  wire [7:0]  dividend,
  input  wire [7:0]  divisor,
  output wire [7:0]  quotient,
  output wire [7:0]  remainder,
  output wire        done
);

  // 1) Counter
  wire [2:0] count;
// Contor de 3 biti: numara ciclurile si genereaza done dupa 8 iteratii
  COUNTER cnt (
    .CLK    (clk),
    .RST    (rst),
    .ENABLE (enable),
    .count  (count),
    .done   (done)
  );
  // only iterate while enable=1 & not done
  wire iter = enable && !done;

  // 2) State regs
  wire [7:0] R_hold, Q_hold;
  wire       Qm1_hold;
  wire [7:0] R_next, Q_next;
  wire       Qm1_next;
  wire [7:0] R_shift, Q_shift;
  wire       Qm1_shift;

  // 3) Shiftâleft
  assign R_shift = {R_hold[6:0], Q_hold[7]};
  assign Q_shift = {Q_hold[6:0], Qm1_hold};
  assign Qm1_shift = Q_hold[0];

  // 4) Subtract divisor
  wire [7:0] R_sub;
// Sumator/rezistor ripple-carry: realizeaza add sau sub in functie de op
  RCA_SUBTRACTOR sub_i (
    .x    (R_shift),
    .y    (divisor),
    .c_in (1'b1),
    .op   (1'b1),
    .z    (R_sub),
    .c_out()
  );

  // 5) Decide restore or keep
  wire go = ~R_sub[7];
  genvar i;
  generate
    for (i = 0; i < 8; i = i + 1) begin : rem_next
// Multiplexor 2-la-1 pe un bit: selecteaza intre cele doua intrari
      Mux_2x1 mR(.sel(go), .in_0(R_shift[i]), .in_1(R_sub[i]), .out(R_next[i]
    end
  endgenerate
  assign Q_next = {Q_shift[7:1], go};

  // 6a) iterateâgate: hold vs update
  wire [7:0] R_iter, Q_iter;
  wire       Qm1_iter;
  generate
    for (i=0; i<8; i=i+1) begin : iter_mux
// Multiplexor 2-la-1 pe un bit: selecteaza intre cele doua intrari
      Mux_2x1 mRi(.sel(iter), .in_0(R_hold[i]), .in_1(R_next[i]), .out(R_iter[i]));
// Multiplexor 2-la-1 pe un bit: selecteaza intre cele doua intrari
      Mux_2x1 mQi(.sel(iter), .in_0(Q_hold[i]), .in_1(Q_next[i]), .out(Q_iter[i]));
    end
  endgenerate
// Multiplexor 2-la-1 pe un bit: selecteaza intre cele doua intrari
  Mux_2x1 mQm1i(.sel(iter), .in_0(Qm1_hold), .in_1(Qm1_shift), .out(Qm1_iter));

  // 6b) startâgate: load initial vs iterate
  wire [7:0] R_load, Q_load;
  wire       Qm1_load;
  generate
    for (i=0; i<8; i=i+1) begin : start_mux
// Multiplexor 2-la-1 pe un bit: selecteaza intre cele doua intrari
      Mux_2x1 mRld(.sel(start), .in_0(1'b0),       .in_1(R_iter[i]), .out(R_load[i]));
// Multiplexor 2-la-1 pe un bit: selecteaza intre cele doua intrari
      Mux_2x1 mQld(.sel(start), .in_0(dividend[i]),.in_1(Q_iter[i]), .out(Q_load[i]));
    end
  endgenerate
// Multiplexor 2-la-1 pe un bit: selecteaza intre cele doua intrari
  Mux_2x1 mQm1ld(.sel(start), .in_0(1'b0), .in_1(Qm1_iter), .out(Qm1_load));

  // 7) Clocked regs
  generate
    for (i = 0; i < 8; i = i + 1) begin : regs
// Flip-flop D: stocheaza un bit la flanc pozitiv, cu reset asincron
      D_FLIPFLOP dR(.D(R_load[i]), .CLK(clk), .RST(rst), .Q(R_hold[i]), .Q_neg());
// Flip-flop D: stocheaza un bit la flanc pozitiv, cu reset asincron
      D_FLIPFLOP dQ(.D(Q_load[i]), .CLK(clk), .RST(rst), .Q(Q_hold[i]), .Q_neg());
    end
  endgenerate
 // Flip-flop D: stocheaza un bit la flanc pozitiv, cu reset asincron
   D_FLIPFLOP dQ1(.D(Qm1_load), .CLK(clk), .RST(rst), .Q(Qm1_hold), .Q_neg());

   // 8) Outputs
   assign remainder = R_hold;
   assign quotient  = Q_hold;
endmodule
```

```
`timescale 1ns/1ps
`include "../src/D_FLIPFLOP.v"
`include "../src/Mux_2x1.v"
`include "../src/COUNTER.v"
`include "../src/RCA_SUBTRACTOR.v"
`include "../src/restoring_div.v"

module restoring_div_tb;
  // Clock and reset
  reg         clk;
  reg         rst;
  // Control
  reg         start;
  reg         enable;
  // Inputs
  reg  [7:0]  dividend;
  reg  [7:0]  divisor;
  // Outputs
  wire [7:0]  quotient;
  wire [7:0]  remainder;
  wire        done;

  // Instantiate DUT
// Divizor restaurator: calculeaza catul si restul in 8 cicluri
  restoring_div uut (
    .clk      (clk),
    .rst      (rst),
    .start    (start),
    .enable   (enable),
    .dividend (dividend),
    .divisor  (divisor),
    .quotient (quotient),
    .remainder(remainder),
    .done     (done)
  );

  // 10 ns clock
  initial clk = 0;
  always #5 clk = ~clk;

  // Dump waves
  initial begin
    $dumpfile("restoring_div_tb.vcd");
    $dumpvars(0, restoring_div_tb);
  end

  // Test sequence
  initial begin
    // ------------------------------------------------------------------
    // 1) Reset
    // ------------------------------------------------------------------
    rst     = 1;
    start   = 0;
    enable  = 0;
    dividend= 0;
    divisor = 0;
    #12;
    rst     = 0;
    #8;

    // ------------------------------------------------------------------
    // Test Case 1: 100 Ã· 3 = 33 rem 1
    // ------------------------------------------------------------------
    dividend = 8'd100;
    divisor  = 8'd3;
    start    = 1;
    enable   = 0;
    // Load dividend & clear remainder
    @(posedge clk);
    start    = 0;
    enable   = 1;
    // Perform 8 iterations
    repeat (8) @(posedge clk);
    enable   = 0;
    @(posedge clk);
    $display("TC1: %0d Ã· %0d = %0d rem %0d (done=%b)",
             dividend, divisor, quotient, remainder, done);
    if (quotient !== 8'd33 || remainder !== 8'd1 || !done)
      $error("TC1 FAILED: got %0d rem %0d done=%b", quotient, remainder, done);

    // ------------------------------------------------------------------
    // Test Case 2:  7 Ã· 8 = 0 rem 7
    // ------------------------------------------------------------------
    #10;
    dividend = 8'd7;
    divisor  = 8'd8;
    start    = 1;
    enable   = 0;
    @(posedge clk);
    start    = 0;
    enable   = 1;
    repeat (8) @(posedge clk);
    enable   = 0;
    @(posedge clk);
    $display("TC2: %0d Ã· %0d = %0d rem %0d (done=%b)",
             dividend, divisor, quotient, remainder, done);
    if (quotient !== 8'd0 || remainder !== 8'd7 || !done)
      $error("TC2 FAILED: got %0d rem %0d done=%b", quotient, remainder, done);

    // ------------------------------------------------------------------
    // Test Case 3: 255 Ã· 5 = 51 rem 0
    // ------------------------------------------------------------------
    #10;
    dividend = 8'd255;
    divisor  = 8'd5;
    start    = 1;
    enable   = 0;
    @(posedge clk);
    start    = 0;
    enable   = 1;
    repeat (8) @(posedge clk);
    enable   = 0;
    @(posedge clk);
    $display("TC3: %0d Ã· %0d = %0d rem %0d (done=%b)",
             dividend, divisor, quotient, remainder, done);
    if (quotient !== 8'd51 || remainder !== 8'd0 || !done)
      $error("TC3 FAILED: got %0d rem %0d done=%b", quotient, remainder, done);

    // ------------------------------------------------------------------
    // Test Case 4:  50 Ã· 25 = 2 rem 0
    // ------------------------------------------------------------------
    #10;
    dividend = 8'd50;
    divisor  = 8'd25;
    start    = 1;
    enable   = 0;
    @(posedge clk);
    start    = 0;
    enable   = 1;
    repeat (8) @(posedge clk);
    enable   = 0;
    @(posedge clk);
    $display("TC4: %0d Ã· %0d = %0d rem %0d (done=%b)",
             dividend, divisor, quotient, remainder, done);
    if (quotient !== 8'd2 || remainder !== 8'd0 || !done)
      $error("TC4 FAILED: got %0d rem %0d done=%b", quotient, remainder, done);

    // Finish
    #10;
    $display("All tests completed.");
    $finish;
  end
endmodule
```

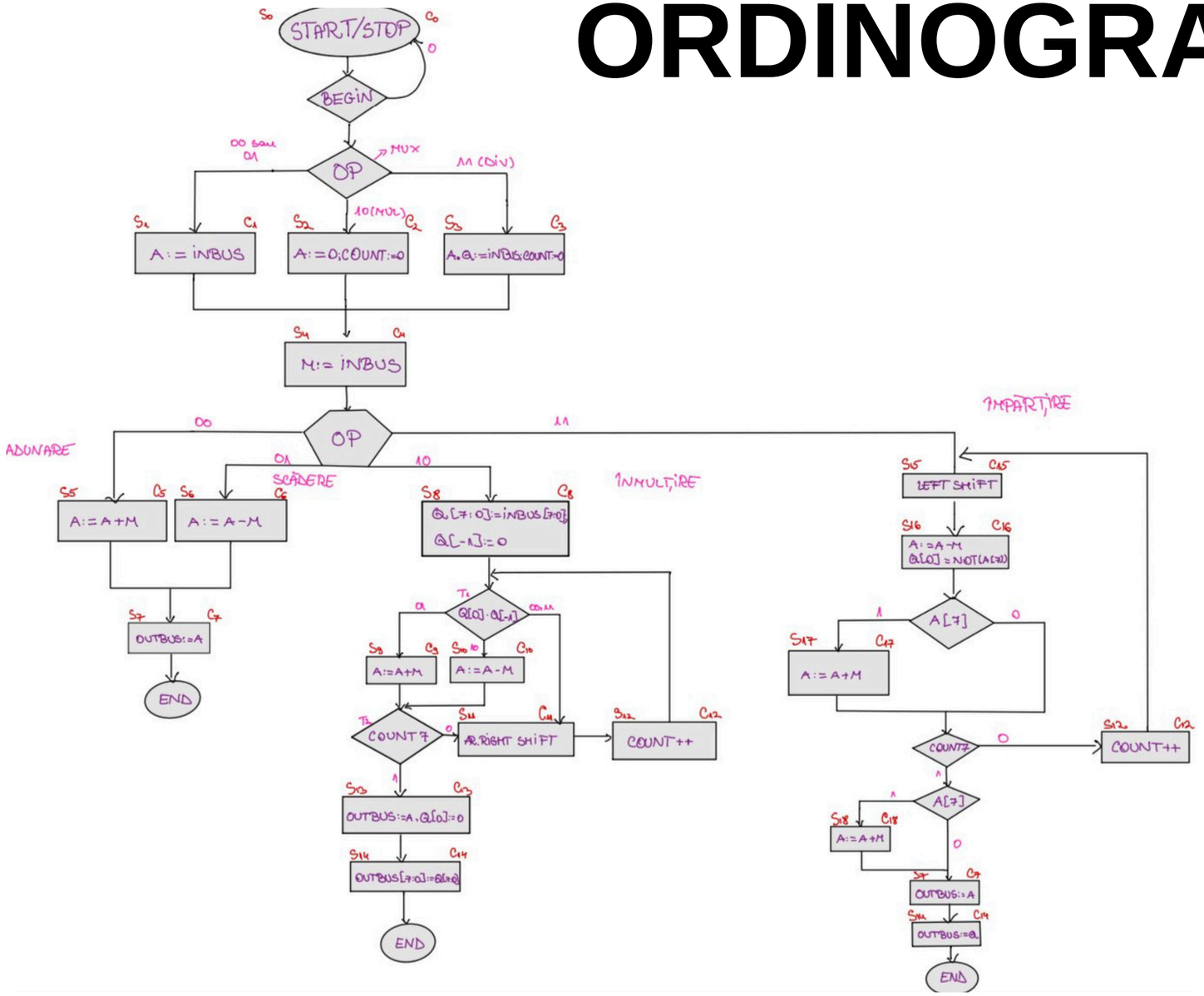
CONTROL UNIT



CONTROL UNIT

```
module Decoder2 (
    output out_0,
    output out_1,
    output out_2,
    output out_3,
    input [1:0] sel
);
    assign out_0 = (sel == 2'h0)? 1'b1 : 1'b0;
    assign out_1 = (sel == 2'h1)? 1'b1 : 1'b0;
    assign out_2 = (sel == 2'h2)? 1'b1 : 1'b0;
    assign out_3 = (sel == 2'h3)? 1'b1 : 1'b0;
endmodule

// Unitate de control ALU: genereaza semnalele de control c0..c18
module Control_Unit_ALU (
  input clk,
  input rst,
  input \BEGIN ,
  input [1:0] Op,
  input count7,
  input \a[7] ,
  input [1:0] \Q[0]Q[-1] ,
  output \END ,
  output c0,
  output c1,
  output c2,
  output c3,
  output c4,
  output c5,
  output c6,
  output c7,
  output c8,
  output c9,
  output c10,
  output c11,
  output c12,
  output c13,
  output c14,
  output c15,
  output c16,
  output c17,
  output c18,
  output c8_wait
);
  wire s0;
  wire c0_temp;
  wire s1;
  wire c1_temp;
  wire s2;
  wire c2_temp;
  wire s3;
  wire c3_temp;
  wire s4;
  wire c5_temp;
  wire s5;
  wire c6_temp;
  wire s6;
  wire c7_temp;
  wire s7;
  wire c8_temp;
  wire s8;
  wire c9_temp;
  wire s9;
  wire c4_temp;
  wire s10;
  wire c10_temp;
  wire s11;
  wire c11_temp;
  wire s12;
  wire c13_temp;
  wire s13;
  wire c14_temp;
  wire s14;
  wire c15_temp;
  wire c16_temp;
  wire s15;
  wire c17_temp;
  wire s16;
  wire c12_temp;
  wire s17;
  wire c18_temp;
  wire END_temp;
  wire add;
  wire sub;
  wire mul;
  wire div;
  wire \00 ;
  wire \01 ;
  wire \10 ;
  wire \11 ;
  wire c8_wait_temp;
  Decoder2 Decoder2_i0 (
    .sel( Op ),
    .out_0( add ),
    .out_1( sub ),
    .out_2( mul ),
    .out_3( div )
  );
  Decoder2 Decoder2_i1 (
    .sel( \Q[0]Q[-1]  ),
    .out_0( \00  ),
    .out_1( \01  ),
    .out_2( \10  ),
    .out_3( \11  )
  );
  // S0
// Flip-flop D: stocheaza un bit la flanc pozitiv, cu reset asincron
  D_FLIPFLOP D_FLIPFLOP_i2 (
    .D( s0 ),
    .CLK( clk ),
    .RST( rst ),
    .Q( c0_temp ),
    .Q_neg ()
  );
  // S1
// Flip-flop D: stocheaza un bit la flanc pozitiv, cu reset asincron
  D_FLIPFLOP D_FLIPFLOP_i3 (
    .D( s1 ),
    .CLK( clk ),
    .RST( rst ),
    .Q( c1_temp ),
    .Q_neg ()
  );
  // S2
// Flip-flop D: stocheaza un bit la flanc pozitiv, cu reset asincron
  D_FLIPFLOP D_FLIPFLOP_i4 (
    .D( s2 ),
    .CLK( clk ),
    .RST( rst ),
    .Q( c2_temp ),
    .Q_neg ()
  );
  // S3
// Flip-flop D: stocheaza un bit la flanc pozitiv, cu reset asincron
  D_FLIPFLOP D_FLIPFLOP_i5 (
    .D( s3 ),
    .CLK( clk ),
    .RST( rst ),
    .Q( c3_temp ),
    .Q_neg ()
  );
  // S4
// Flip-flop D: stocheaza un bit la flanc pozitiv, cu reset asincron
  D_FLIPFLOP D_FLIPFLOP_i11 (
    .D( s9 ),
    .CLK( clk ),
    .RST( rst ),
    .Q( c4_temp ),
    .Q_neg ()
  );
  // S5
// Flip-flop D: stocheaza un bit la flanc pozitiv, cu reset asincron
  D_FLIPFLOP D_FLIPFLOP_i6 (
    .D( s4 ),
    .CLK( clk ),
    .RST( rst ),
    .Q( c5_temp ),
    .Q_neg ()
  );
  // S6
// Flip-flop D: stocheaza un bit la flanc pozitiv, cu reset asincron
  D_FLIPFLOP D_FLIPFLOP_i7 (
    .D( s5 ),
    .CLK( clk ),
    .RST( rst ),
    .Q( c6_temp ),
    .Q_neg ()
  );
  // S7
// Flip-flop D: stocheaza un bit la flanc pozitiv, cu reset asincron
  D_FLIPFLOP D_FLIPFLOP_i8 (
    .D( s6 ),
    .CLK( clk ),
    .RST( rst ),
    .Q( c7_temp ),
    .Q_neg ()
  );
  // S8
// Flip-flop D: stocheaza un bit la flanc pozitiv, cu reset asincron
  D_FLIPFLOP D_FLIPFLOP_i9 (
    .D( s7 ),
    .CLK( clk ),
    .RST( rst ),
    .Q( c8_temp ),
    .Q_neg ()
  );
  // S9
// Flip-flop D: stocheaza un bit la flanc pozitiv, cu reset asincron
  D_FLIPFLOP D_FLIPFLOP_i10 (
    .D( s8 ),
    .CLK( clk ),
    .RST( rst ),
    .Q( c9_temp ),
    .Q_neg ()
  );
  // S10
// Flip-flop D: stocheaza un bit la flanc pozitiv, cu reset asincron
  D_FLIPFLOP D_FLIPFLOP_i12 (
    .D( s10 ),
    .CLK( clk ),
    .RST( rst ),
    .Q( c10_temp ),
    .Q_neg ()
  );
  // S11
// Flip-flop D: stocheaza un bit la flanc pozitiv, cu reset asincron
  D_FLIPFLOP D_FLIPFLOP_i13 (
    .D( s11 ),
    .CLK( clk ),
    .RST( rst ),
    .Q( c11_temp ),
    .Q_neg ()
  );
  // S13
// Flip-flop D: stocheaza un bit la flanc pozitiv, cu reset asincron
  D_FLIPFLOP D_FLIPFLOP_i14 (
    .D( s12 ),
    .CLK( clk ),
    .RST( rst ),
    .Q( c13_temp ),
    .Q_neg ()
  );
  // S14
// Flip-flop D: stocheaza un bit la flanc pozitiv, cu reset asincron
  D_FLIPFLOP D_FLIPFLOP_i15 (
    .D( s13 ),
    .CLK( clk ),
    .RST( rst ),
    .Q( c14_temp ),
    .Q_neg ()
  );
  // S15
// Flip-flop D: stocheaza un bit la flanc pozitiv, cu reset asincron
  D_FLIPFLOP D_FLIPFLOP_i16 (
    .D( s14 ),
    .CLK( clk ),
    .RST( rst ),
    .Q( c15_temp ),
    .Q_neg ()
  );
  // S16
// Flip-flop D: stocheaza un bit la flanc pozitiv, cu reset asincron
  D_FLIPFLOP D_FLIPFLOP_i17 (
    .D( c15_temp ),
    .CLK( clk ),
    .RST( rst ),
    .Q( c16_temp ),
    .Q_neg ()
  );
  // S17
// Flip-flop D: stocheaza un bit la flanc pozitiv, cu reset asincron
  D_FLIPFLOP D_FLIPFLOP_i18 (
    .D( s15 ),
    .CLK( clk ),
    .RST( rst ),
    .Q( c17_temp ),
    .Q_neg ()
  );
  // S12
// Flip-flop D: stocheaza un bit la flanc pozitiv, cu reset asincron
  D_FLIPFLOP D_FLIPFLOP_i19 (
    .D( s16 ),
    .CLK( clk ),
    .RST( rst ),
    .Q( c12_temp ),
    .Q_neg ()
  );
  // S18
// Flip-flop D: stocheaza un bit la flanc pozitiv, cu reset asincron
  D_FLIPFLOP D_FLIPFLOP_i20 (
    .D( s17 ),
    .CLK( clk ),
    .RST( rst ),
    .Q( c18_temp ),
    .Q_neg ()
  );
  assign s0 = (~ \BEGIN  | END_temp);
  assign END_temp = ((div & c14_temp) | (mul & c14_temp) | ((sub | add) & c7_temp));
  assign s1 = ((add | sub) & \BEGIN  & c0_temp);
  assign s2 = (mul & \BEGIN  & c0_temp);
  assign s3 = (div & \BEGIN  & c0_temp);
  assign s4 = (add & c4_temp);
  assign s5 = (sub & c4_temp);
  assign s6 = ((c16_temp & ~ \a[7]  & count7) | c5_temp | c6_temp | c18_temp | (c17_temp & count7 & ~ \a[7] ));
  assign s7 = (mul & c4_temp);
  assign s16 = ((~ count7 & c17_temp) | c11_temp | (c16_temp & ~ \a[7]  & ~ count7));
  assign s12 = (count7 & c9_temp);
  assign s13 = ((div & c7_temp) | (mul & c13_temp));
  assign s15 = (\a[7]  & c16_temp);
  assign s17 = (count7 & \a[7]  & c17_temp);
  assign s14 = ((div & c4_temp) | (div & c12_temp));
  assign s11 = (mul & ((c8_wait_temp & (\11  | \00 )) | (~ count7 & c9_temp) | (c10_temp & ~ count7) | (c12_temp & (\11  | \00 ))));
  // S8_wait
// Flip-flop D: stocheaza un bit la flanc pozitiv, cu reset asincron
  D_FLIPFLOP D_FLIPFLOP_i21 (
    .D( c8_temp ),
    .CLK( clk ),
    .RST( rst ),
    .Q( c8_wait_temp ),
    .Q_neg ()
  );
  assign s8 = ((\01  & c8_wait_temp) | (\01  & c12_temp));
  assign s10 = ((\10  & c8_wait_temp) | (\10  & c12_temp));
  assign s9 = (c1_temp | c2_temp | c3_temp);
  assign \END  = END_temp;
  assign c0 = c0_temp;
  assign c1 = c1_temp;
  assign c2 = c2_temp;
  assign c3 = c3_temp;
  assign c4 = c4_temp;
  assign c5 = c5_temp;
  assign c6 = c6_temp;
  assign c7 = c7_temp;
  assign c8 = c8_temp;
  assign c9 = c9_temp;
  assign c10 = c10_temp;
  assign c11 = c11_temp;
  assign c12 = c12_temp;
  assign c13 = c13_temp;
  assign c14 = c14_temp;
  assign c15 = c15_temp;
  assign c16 = c16_temp;
  assign c17 = c17_temp;
  assign c18 = c18_temp;
  assign c8_wait = c8_wait_temp;
endmodule
```

ORDINOGRAMA



S_0: $D_0 = BEGIN \mid END$, $c_0 = B_0$

S_1: $D_1 = B_0 \cdot BEGIN \cdot (ADD \mid SUB)$, $c_1 = B_1$

S_2: $D_2 = B_0 \cdot BEGIN \cdot MUL$, $c_2 = B_2$

S_3: $D_3 = B_0 \cdot BEGIN \cdot DIV$, $c_3 = B_3$

S_4: $D_4 = B_1 \mid B_2 \mid B_3$, $c_4 = B_4$

S_5: $D_5 = B_4 \cdot ADD$, $c_5 = B_5$

S_6: $D_6 = B_4 \cdot SUB$, $c_6 = B_6$

S_7: $D_7 = (\overline{A[7]} \cdot COUNT7 \cdot B_{17}) \mid B_{18} \mid B_6 \mid B_5 \mid (\overline{A[7]} \cdot COUNT7 \cdot B_{16})$, $c_7 = B_7$

S_8: $D_8 = B_4 \cdot MUL$, $c_8 = B_8$

S_9: $D_9 = (B_{12} \cdot Q[0] \cdot Q[-1] == 01) \mid (B_8 \cdot Q[0] [-1] == 01)$, $c_9 = B_9$

S_{10}: $D_{10} = (B_{12} \cdot Q[0] \cdot Q[-1] == 10) \mid (B_8 \cdot Q[0] \cdot Q[-1] == 10)$, $c_{10} = B_{10}$

S_{11}: $D_{11} = [[Q[0] \cdot Q[-1] == 00 \mid 11) \cdot B_{12}] \mid (B_{10} \cdot \overline{COUNT7}) \mid [(Q[0] \cdot Q[-1] == 00 \mid 11) \cdot B_8]) \cdot MUL$, $c_{11} = B_{11}$

S_{12}: $D_{12} = (\overline{COUNT7} \cdot \overline{A[7]} \cdot B_{16}) \mid B_{11}) (B_{17} \cdot \overline{COUNT7})$, $c_{12} = B_{12}$

S_{13}: $D_{13} = B_9 \cdot COUNT7$, $c_{13} = B_{13}$

S_{14}: $D_{14} = (B_{13} \cdot MUL) \mid (B_7 \cdot DIV)$, $c_{14} = B_{14}$

S_{15}: $D_{15} = (B_{12} \cdot DIV) \mid (B_4 \cdot DIV)$, $c_{15} = B_{15}$

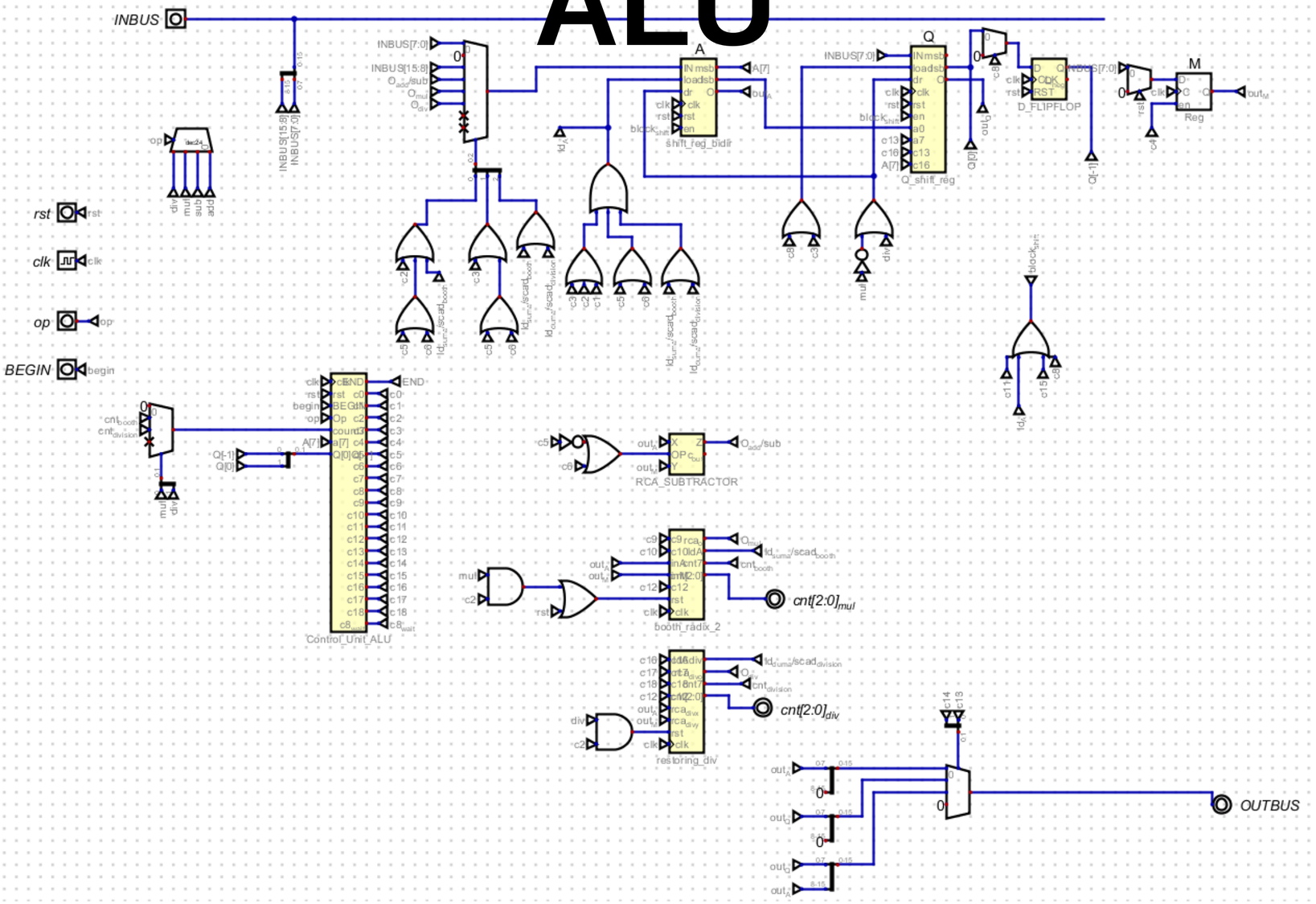
S_{16}: $D_{16} = B_{15}$, $c_{16} = B_{16}$

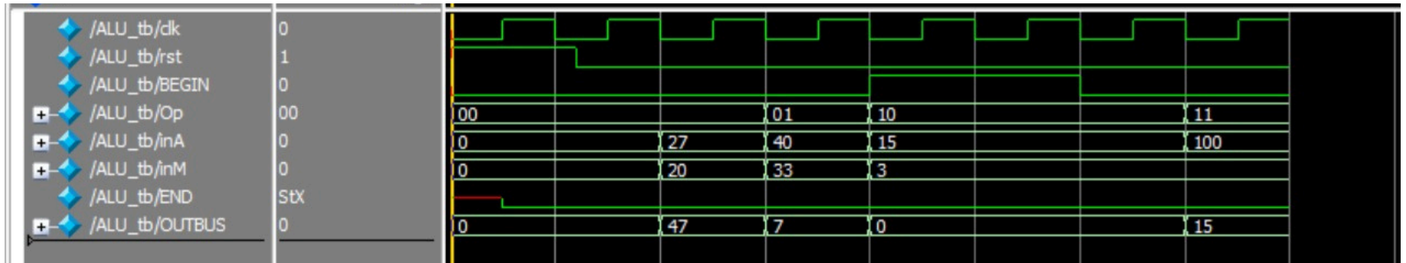
S_{17}: $D_{17} = B_{16} \cdot A[7]$, $c_{17} = B_{17}$

S_{18}: $D_{18} = [A[7]$, $c_{18} = B_{18}$

END: $(ADD \mid SUB) \cdot B_7) \mid (B_{14} \cdot MUL) \mid (B_{14} \cdot DIV)$

ALU


INBUS
rst
clk
op
BEGIN
A
Q
M
shift_reg_bidir
Q_shift_reg
D_FLIPFLOP
Reg
RCA_SUBTRACTOR
booth_radix_2
restoring_div
Control_Unit_ALU
cnt[2:0]mul
cnt[2:0]div
OUTBUS


/ALU_tb/clk
/ALU_tb/rst
/ALU_tb/BEGIN
/ALU_tb/Op
/ALU_tb/inA
/ALU_tb/inM
/ALU_tb/END
/ALU_tb/OUTBUS
0
1
0
00
0
0
StX
0
00
01
10
11
0
27
40
15
100
0
20
33
3
0
47
7
0
15

ALU

```
`timescale 1ns/1ps
`include "../src/Control_Unit_ALU.v"
`include "../src/D_FLIPFLOP.v"
`include "../src/Mux_2x1.v"
`include "../src/COUNTER.v"
`include "../src/RCA_SUBTRACTOR.v"
`include "../src/booth_radix_2.v"
`include "../src/restoring_div.v"

module ALU (
  input  wire        clk,
  input  wire        rst,
  input  wire        BEGIN,    // pulse to start mul/div
  input  wire [1:0]  Op,       // 00=add,01=sub,10=mul,11=div
  input  wire [7:0]  inA,
  input  wire [7:0]  inM,
  output wire        END,
  output wire [15:0] OUTBUS
);

  // Control signals
  wire c0, c1, c2, c3, c4, c5, c6, c7,
       c8, c9, c10, c11, c12, c13, c14, c15,
       c16, c17, c18, c8_wait;
  wire count7_mul, count7_div, Q0, Qm1;
  wire a7;

  // select correct counter done for CU
  wire count7_sig;
  assign count7_sig = (Op == 2'b10) ? count7_mul : count7_div;

// Unitate de control ALU: genereaza semnalele de control c0..c18
  Control_Unit_ALU cu (
    .clk        (clk),
    .rst        (rst),
    .\BEGIN     (BEGIN),
    .Op         (Op),
    .count7     (count7_sig),
    .\a[7]      (a7),
    .\Q[0]Q[-1] ({Q0, Qm1}),
    .\END       (END),
    .c0(c0), .c1(c1), .c2(c2), .c3(c3),
    .c4(c4), .c5(c5), .c6(c6), .c7(c7),
    .c8(c8), .c9(c9), .c10(c10), .c11(c11),
    .c12(c12), .c13(c13), .c14(c14), .c15(c15),
    .c16(c16), .c17(c17), .c18(c18), .c8_wait(c8_wait)
  );

  // A/M registers
  wire [7:0] A_reg, M_reg;
  assign a7 = A_reg[7];
  genvar i;
  generate
    for (i = 0; i < 8; i = i + 1) begin : AM
// Flip-flop D: stocheaza un bit la flanc pozitiv, cu reset asincron
      D_FLIPFLOP dA (
        .D   (c0 ? inA[i] : A_reg[i]),
        .CLK (clk),
        .RST (rst),
        .Q   (A_reg[i]),
        .Q_neg()
      );
```

```
// Flip-flop D: stocheaza un bit la flanc pozitiv, cu reset asincron
      D_FLIPFLOP dM (
        .D   (c1 ? inM[i] : M_reg[i]),
        .CLK (clk),
        .RST (rst),
        .Q   (M_reg[i]),
        .Q_neg()
      );
    end
  endgenerate

  // Combinational ADD/SUB
  wire [7:0] sumA, sumS;
// Sumator/rezistor ripple-carry: realizeaza add sau sub in functie de op
  RCA_SUBTRACTOR add_u (
    .x     (inA),
    .y     (inM),
    .c_in (1'b0),
    .op    (1'b0),
    .z     (sumA),
    .c_out()
  );
// Sumator/rezistor ripple-carry: realizeaza add sau sub in functie de op
  RCA_SUBTRACTOR sub_u (
    .x     (inA),
    .y     (inM),
    .c_in (1'b1),
    .op    (1'b1),
    .z     (sumS),
    .c_out()
  );


  // Booth multiplier
  wire [15:0] mulP;
// Multiplicator Booth Radix-2: iterativ A,Q,Qm1 cu 8 cicluri
  booth_radix_2 bm (
    .clk      (clk),
    .rst      (rst),
    .start    (c2),
    .enable   (c7),
    .multA    (A_reg),
    .multB    (M_reg),
    .done     (count7_mul),
    .product  (mulP),
    .Q0       (Q0),
    .Qm1      (Qm1)
  );

  // Restoring divider
  wire [7:0] divQ;
// Divizor restaurator: calculeaza catul si restul in 8 cicluri
  restoring_div rd (
    .clk      (clk),
    .rst      (rst),
    .start    (c8),
    .enable   (c12),
    .dividend (A_reg),
    .divisor  (M_reg),
    .quotient (divQ),
    .remainder(),
    .done     (count7_div)
  );
```

```
  // Final result MUX by Op
  wire [15:0] extA = {8'b0, sumA},
              extS = {8'b0, sumS},
              extM = mulP,
              extD = {8'b0, divQ};

  generate
    for (i = 0; i < 16; i = i + 1) begin : OUT
      wire useA = (Op == 2'b00),
           useS = (Op == 2'b01),
           useM = (Op == 2'b10),
           useD = (Op == 2'b11);
      wire [15:0] selbus = useA ? extA :
                           useS ? extS :
                           useM ? extM :
                           useD ? extD :
                                  16'b0;
      assign OUTBUS[i] = selbus[i];
    end
  endgenerate

endmodule
```

```
`timescale 1ns/1ps
`include "../src/Control_Unit_ALU.v"
`include "../src/D_FLIPFLOP.v"
`include "../src/Mux_2x1.v"
`include "../src/COUNTER.v"
`include "../src/RCA_SUBTRACTOR.v"
`include "../src/booth_radix_2.v"
`include "../src/restoring_div.v"
`include "../src/ALU.v"

module ALU_tb;
  // Inputs
  reg        clk;
  reg        rst;
  reg        BEGIN;
  reg [1:0]  Op;
  reg [7:0]  inA;
  reg [7:0]  inM;
  // Outputs
  wire       END;
  wire [15:0] OUTBUS;

  // Instantiate the Unit Under Test
  ALU uut (
    .clk     (clk),
    .rst     (rst),
    .BEGIN   (BEGIN),
    .Op      (Op),
    .inA     (inA),
    .inM     (inM),
    .END     (END),
    .OUTBUS  (OUTBUS)
  );

  // Clock generator: 10 ns period
  initial clk = 0;
  always #5 clk = ~clk;

  initial begin
    // Dump waves for GTKWave, etc.
    $dumpfile("ALU_tb.vcd");
    $dumpvars(0, ALU_tb);

    //=== Reset ===
    rst    = 1; BEGIN = 0; Op = 2'b00; inA = 8'd0; inM = 8'd0;
    #12;
    rst = 0;
    #8;

    //=== Test 1: ADD 27 + 20 = 47 ===
    inA   = 8'd27; inM = 8'd20; Op = 2'b00; BEGIN = 0;
    #10;
    if (OUTBUS !== 16'd47)
      $error("ADD failed: got %0d, expected 47", OUTBUS);
    else
      $display("ADD passed: 27+20 = %0d", OUTBUS);

    //=== Test 2: SUB 40 - 33 = 7 ===
    inA   = 8'd40; inM = 8'd33; Op = 2'b01; BEGIN = 0;
    #10;

    if (OUTBUS !== 16'd7)
      $error("SUB failed: got %0d, expected 7", OUTBUS);
    else
      $display("SUB passed: 40-33 = %0d", OUTBUS);

    //=== Test 3: MUL 15 * 3 = 45 ===
    inA    = 8'd15; inM = 8'd3; Op = 2'b10; BEGIN = 1;
    #20;
    wait (END);
    if (OUTBUS !== 16'd45)
      $error("MUL failed: got %0d, expected 45", OUTBUS);
    else
      $display("MUL passed: 15*3 = %0d", OUTBUS);
    BEGIN = 0;
    #10;

    //=== Test 4: DIV 100 / 3 = 33 ===
    inA    = 8'd100; inM = 8'd3; Op = 2'b11; BEGIN = 1;
    wait (END);
    if (OUTBUS[7:0] !== 8'd33)
      $error("DIV failed: got %0d, expected 33", OUTBUS[7:0]);
    else
      $display("DIV passed: 100/3 = %0d", OUTBUS[7:0]);
    BEGIN = 0;
    #10;

    $display("All ALU tests completed.");
    $finish;
  end
endmodule
```