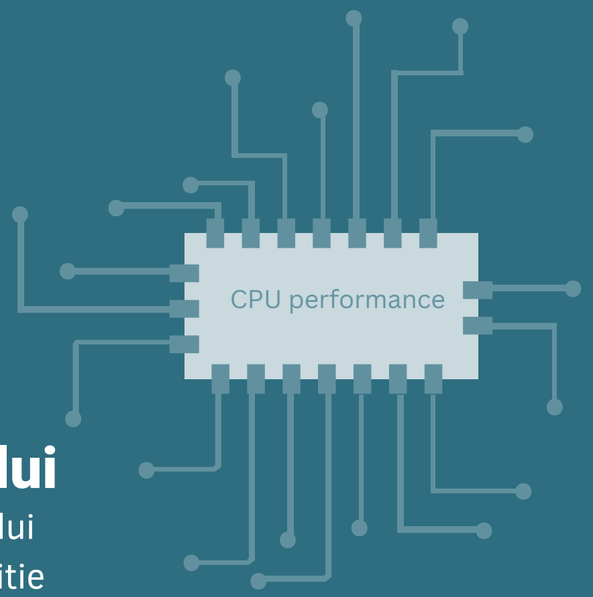
IMPLEMENTARE *ALU*

Burada Andrei Alexandru
Maroiu Crina Alina

Functionalitati implementate



001

Operatii logice

Am implementat portile de baza AND, OR, XOR, EXOR, utilizate in algoritmii utilizati in dezvoltarea functionalitatilor mai complexe.

011

Adunare

Am implementat adunarea folosind un sumator de tip Ripple Cary Adder, cu scopul de a avea performante ridicate si o complexitate redusa.

101

Inmultire

Pentru inmultire am optat sa folosim Booth Radix 4, pentru o mai buna performanta a codului.

010

Shiftarea numarului

Realizeaza shiftarea numarului primit ca argument cu o pozitie spre dreapta, respectiv stanga.

100

Scadere

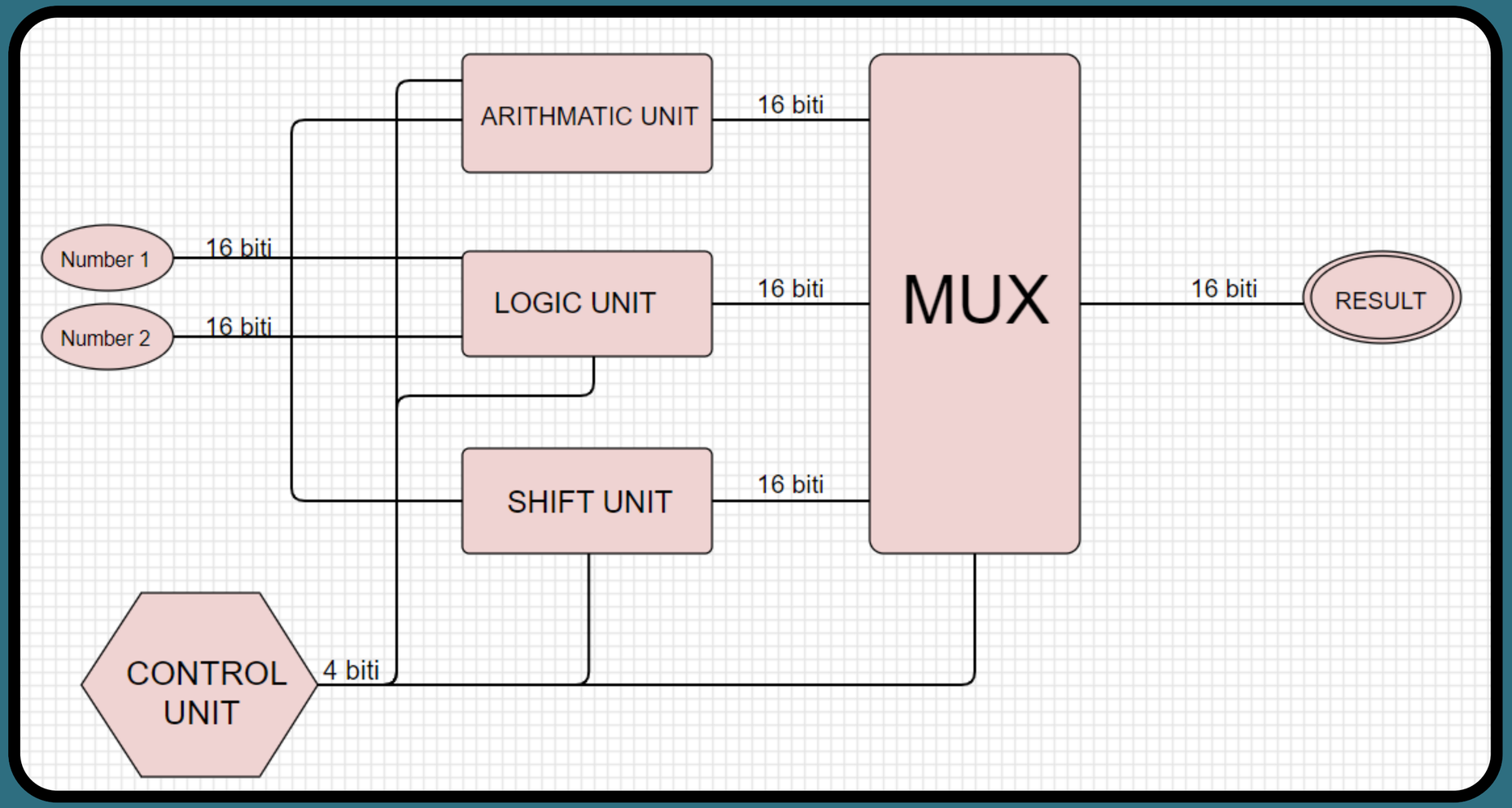
Pentru implementarea scaderii am folosit sumatorul deja implementat, unde am folosit regula: A- B = A + (-B)

110

Impartire

Am implementat impartirea folosind Non-Restoring deoarece prezinta o complexitate algoritmica redusa iar eficienta este ridicata.

Diagrama proiect ALU



1. Operatiile Logice

AND
OR
XOR
EXOR

```
module AND #(
  parameter size = 16
)(
  input [size-1:0] numar1,
  input [size-1:0] numar2,
  output [size-1:0] out

);

genvar i;

generate
 for (i = 0; i < size; i = i + 1) begin: un_text
    assign out[i] = numar1[i] & numar2[i]; //facem AND logic bit cu bit
end

endgenerate
endmodule
```

Implementare poarta AND pe 16 biti

```
module OR #(
  parameter size = 16
)(
  input [size-1:0] numar1,
  input [size-1:0] numar2,
  output [size-1:0] out
);

genvar i;
generate

 for (i = 0; i < size; i = i + 1) begin: un_test
    assign out[i] = numar1[i] | numar2[i]; //facem OR bit cu bit
end

endgenerate
endmodule
```

Implementare poarta OR pe 16 biti

```
module XOR #(
  parameter size = 16
)(
  input [size-1:0] numar1,
  input [size-1:0] numar2,
  output [size-1:0] out
);

genvar i;

generate

 for (i = 0; i < size; i = i + 1) begin: un_test
    assign out[i] = numar1[i] ^ numar2[i]; //facem XOR bit cu bit
end

endgenerate
endmodule
```

Implementare poarta XOR pe 16 biti

Implementare poarta EXOR pe 16 biti

```
module exor #(
   parameter size = 16 // numarul de biti ai cuvântului
)(
   input [size-1:0] numar,
   input select,
   output [size-1:0]output
);

genvar i;

generate
 for (i = 0; i < size; i = i + 1) begin: un_text
    assign output[i] = numar[i] ^ select; //facem EXOR
end
```

```
module exor_tb;

// Parametrii pentru testbench
parameter x = 16;

// Declaratii testbench
reg [x-1:0] numar;
reg select;
wire [x-1:0] out;

// Instantiem si modulul exor pentru testare
exor #(x) exor_inst (
   .numar(numar),
   .select(select),
   .output(out)
);

initial begin
   // initializari
   numar = 16'b1101101100000001;
   select = 1;


   $display("Num?rul de intrare: %b", numar);
   $display("Select: %b", select);
//asteptam propagarea semnalelor
   #10;


   $display("Rezultatul exorului: %b", out);
end
endmodule
```

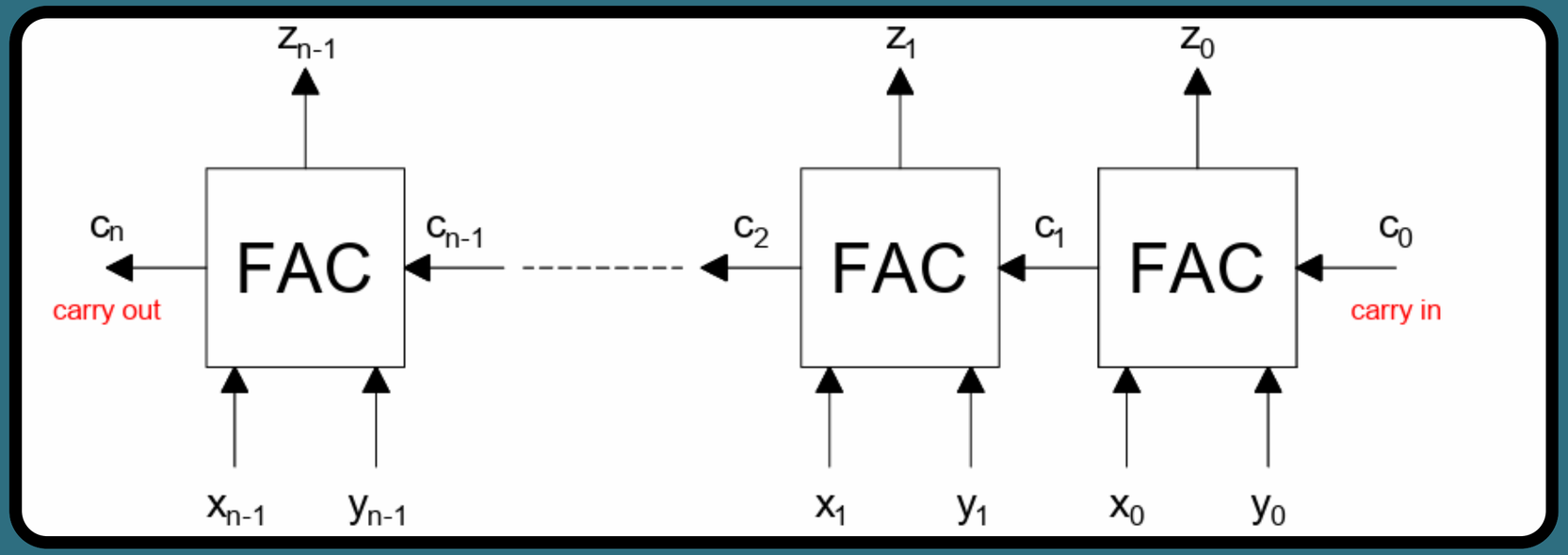
Exemplu de test_bench pentru implementarea portii EXOR

2. Adunarea(RCA)

ecuatiile iesirilor

$z_i = x_i \oplus y_i \oplus c_i$

$c_{(i+1)} = x_i \cdot y_i + x_i \cdot c_i + y_i \cdot c_i$



RCA utilizeaza celule dedicate de insumare pentru fiecare rang binar, unde propagarea carry-ului are loc catre rangurile superioare(stanga)

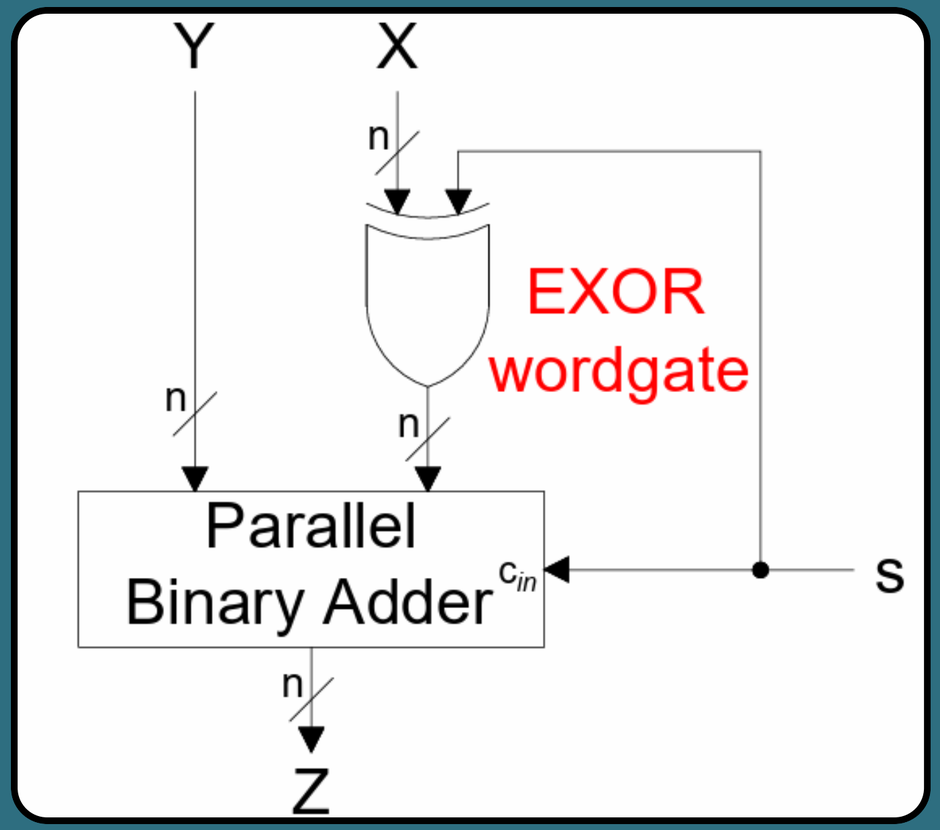
```
module parallel_adder (
    input cin,
    input [16:0]a, b,
    output [16:0]out_add
);
    wire w0, w1, w2, w3, w4, w5, w6, w7, w8;
    wire w9, w10, w11, w12, w13, w14, w15;
    wire [16:0]out;

    fac fac0(.a(a[0]), .b(b[0]), .cin(cin), .out(out[0]), .cout(w0));
    fac fac1(.a(a[1]), .b(b[1]), .cin(w0), .out(out[1]), .cout(w1));
    fac fac2(.a(a[2]), .b(b[2]), .cin(w1), .out(out[2]), .cout(w2));
    fac fac3(.a(a[3]), .b(b[3]), .cin(w2), .out(out[3]), .cout(w3));
    fac fac4(.a(a[4]), .b(b[4]), .cin(w3), .out(out[4]), .cout(w4));
    fac fac5(.a(a[5]), .b(b[5]), .cin(w4), .out(out[5]), .cout(w5));
    fac fac6(.a(a[6]), .b(b[6]), .cin(w5), .out(out[6]), .cout(w6));
    fac fac7(.a(a[7]), .b(b[7]), .cin(w6), .out(out[7]), .cout(w7));

    fac fac8(.a(a[8]), .b(b[8]), .cin(w7), .out(out[8]), .cout(w8));
    fac fac9(.a(a[9]), .b(b[9]), .cin(w8), .out(out[9]), .cout(w9));
    fac fac10(.a(a[10]), .b(b[10]), .cin(w9), .out(out[10]), .cout(w10));
    fac fac11(.a(a[11]), .b(b[11]), .cin(w10), .out(out[11]), .cout(w11));
    fac fac12(.a(a[12]), .b(b[12]), .cin(w11), .out(out[12]), .cout(w12));
    fac fac13(.a(a[13]), .b(b[13]), .cin(w12), .out(out[13]), .cout(w13));
    fac fac14(.a(a[14]), .b(b[14]), .cin(w13), .out(out[14]), .cout(w14));
    fac fac15(.a(a[15]), .b(b[15]), .cin(w14), .out(out[15]), .cout(w15));

    fac fac16(.a(a[16]), .b(b[16]), .cin(w15), .out(out[16]), .cout());

    assign out_add = out;
endmodule
// face adunare paralela, bit cu bit
```

Implementarea modul sumator RCA ce contine 17 celule FAC interconectate

Implementare unitara celula FAC

```
module fac(
    input a, b, cin,
    output out, cout
);
    assign out = (a ^ b) ^ cin;
    assign cout = (a & b) ^ ((a ^ b) & cin);
endmodule
```

3. Scaderea(RCA)



In cadrul operatiei de scadere am refactorizat din functionalitatile deja implementate in proiect, folosind adunarea prin regula:

Y - X = Y + (- X)

Singura adaugare la nivel de cod verilog este folosirea unei porti de tip EXOR ce reprezinta la randul ei poarta XOR dar cu rezultat negat printr-un inversor.

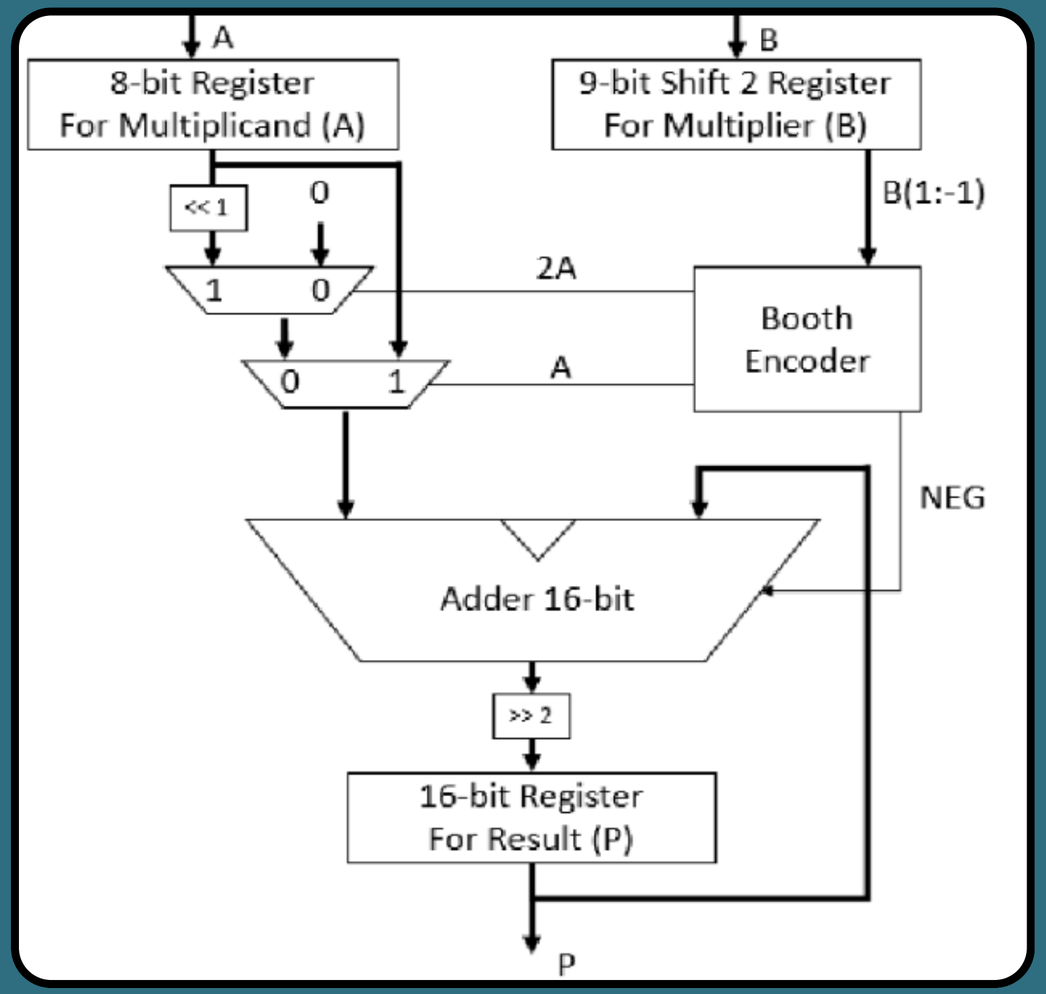
Implementare modul `scazator`

```
module subtract(
    input[15:0] a, b,
    output[15:0] result
);

wire[15:0] outputExor;

exor_w #(16) resultExor (
    .numar(b),
    .select(1),
    .exor(outputExor)
);

parallel_adder dut (
        .cin(1),
        .a(a),
        .b(outputExor),
        .out_add(result)
);
endmodule
```

Test_bench

```
module subtract_tb;
    // Inputs
    reg [15:0] a, b;

    // Outputs
    wire [15:0] subtractResult;

    // Instantiate the substract module
    subtract dut(
    .a(a),
    .b(b),
    .result(subtractResult)
    );

    // Test stimulus
    initial begin
        // Initialize inputs
        a = 16'b00000000_00001001;  //Introducem numerele pentru scadere
        b = 16'b00000000_00000010;

        // Apply inputs and display outputs
        #1500;
        $display("Input: Numarul_1 = %d, Numarul_2 = %d", a, b);
        $display("Output: DIFERENTA = %d", subtractResult);

    end
endmodule
```

4. Inmultirea(Booth R4)



Pentru operatia de inmultire am ales Booth Radix4, deoare are o performanta superioara fata de algoritmul Booth Radix2, aceasta fiind datorata modului in care se grupeaza bitii rezultatului, se formeaza 4 perechi ce au 1 bit comun, acest lucru fiind punctul sau forte.

$$1\,1\,\overline{1\,0}\,\overline{0\,0}\,\overline{1\,1}\,0$$

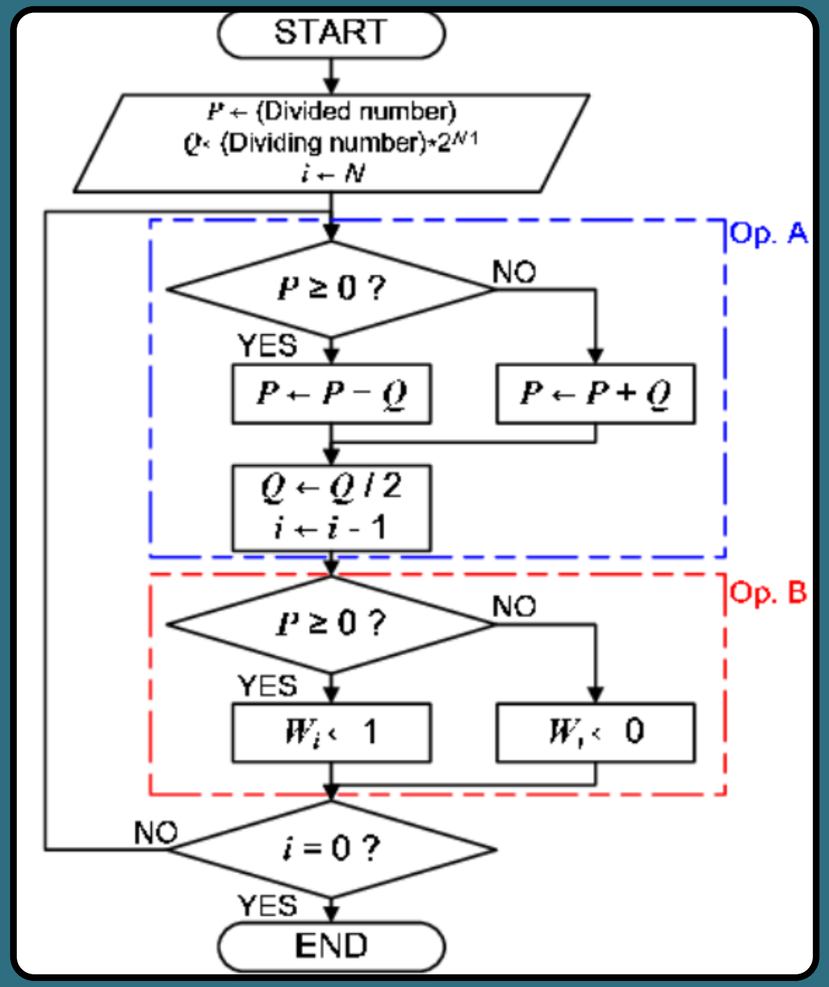
Implementare modul inmultitor Booth_R4

```
module booth_multiplier (
    input [15:0] multiplicand,
    input [15:0] multiplier,
    output reg [31:0] product
);

reg [15:0] a;
reg [32:0] s;

always @(*) begin
    a = multiplicand;
    s = {16'b0,multiplier,1'b0};

    repeat(16) begin
        if (s[1:0] == 2'b01) begin
            s = s + {a,17'b0}; //mut a cu 8 pt suma cu biti prod

        end else if (s[1:0] == 2'b10) begin
            s = s - {a,17'b0};
        end

        // Shiftam la dreapta folosind shift arithmetic pentru a pastra semnul
        s = {s[32], s} >> 1;
    end
    product = s[32:1]; //trunchiem un bit adica cel adaugat mai sus
end

endmodule
```

Test_bench Booth_R4

```
module booth_multiplier_tb;
    reg [15:0] multiplicand, multiplier;

    wire [31:0] product;

    booth_multiplier dut (
        .multiplicand(multiplicand),
        .multiplier(multiplier),
        .product(product)
    );

    initial begin //Introducere date
        multiplicand = 16'b00000000_00000010;
        multiplier   = 16'b00000000_00000011;

        #1500;

        $display("Multiplicand = %d", multiplicand);
        $display("Multiplier = %d", multiplier);
        $display("PRODUS = %d", product);
    end
endmodule
```

5. Impartirea(Non-Restoring)



- Impartirea a 2 numere intregi in format binar, se realizeaza in codul nostru prin implementarea algoritmului NonRestoring.

- Acesta este un algoritm ce prezinta multiple avantaje hardware:
  - realizarea impartirii folosind doar operatii de baza in hardware adunarea si shiftarea
  - functionalitatile sunt deja implementate in proiectul nostru, deci refolosim codul deja existent.
  - complexitate hardware redusa
  - folosire eficienta a resurselor

```
module nonRD(
    input clk, rst_b,
    input [15:0] inbus,
    output [15:0] outbus
);
    wire[15:0] reg_a, reg_m, reg_q;
    wire[16:0] adder_output;
    wire[2:0] cnt;
    wire reg_s;
    wire c0, c1, c2, c3, c4, c5, c6, c7, c8;

    control_unit inst7(.clk(clk),
                       .rst_b(rst_b),
                       .s(reg_s),
                       .is_count_7(cnt == 3'b111),
                       .c0(c0),
                       .c1(c1),
                       .c2(c2),
                       .c3(c3),
                       .c4(c4),
                       .c5(c5),
                       .c6(c6),
                       .c7(c7),
                       .c8(c8));

    reg_m inst0(.clk(clk),
                .rst_b(rst_b),
                .c2(c2),
                .inbus(inbus),
                .out(reg_m));

    reg_q inst1(.clk(clk),
                .rst_b(rst_b),
                .c1(c1),
                .c5(c5),
                .c6(c6),
                .c7(c7),
                .s(~s),
                .inbus(inbus[7:0]),
                .out(reg_q),
                .outbus(outbus));
```

PARTEA 1

```
    reg_a inst2(.clk(clk),
                .rst_b(rst_b),
                .c0(c0),
                .c3(c3),
                .c6(c6),
                .c8(c8),
                .q7(reg_q[7]),
                .inbus(inbus),
                .adder_input(adder_output[7:0]),
                .out(reg_a),
                .outbus(outbus));

    reg_s inst3(.clk(clk),
                .rst_b(rst_b),
                .adder_input(adder_output[8]),
                .a7(reg_a[7]),
                .c0(c0),
                .c3(c3),
                .c6(c6),
                .out(reg_s));

    counter inst4(.clk(clk),
                  .rst_b(rst_b),
                  .c0(c0),
                  .c6(c6),
                  .out(cnt));

    parallel_adder inst6(.cin(c4),
                         .a({reg_s, reg_a}),
                         .b({reg_m[7], reg_m} ^ {9{c4}}),
                         .out_add(adder_output));

endmodule
```

PARTEA 2

Implementare modul impartitor de tip Non_Restoring

Referinte

1. https://www.javatpoint.com/non-restoring-division-algorithm-for-unsigned-integer
2. https://cv.upt.ro/mod/folder/view.php?id=377061
3. https://www.semanticscholar.org/Modified-Radix-4-8-Bit-BoothHerdian/22aa503673d247
4. https://cv.upt.ro/pluginfile.php/611483/mod_resource/content/1/Curs-4-2022.pdf
5. https://ieeexplore.ieee.org/document/45012
6. https://www.researchgate.net/figure/Non-restoring-division-signed-binary-number-NRTS
7. https://www.sciencedirect.com/topics/computer-science/ripple-carry-adder
8. https://www.digikey.si/en/schemeit/project