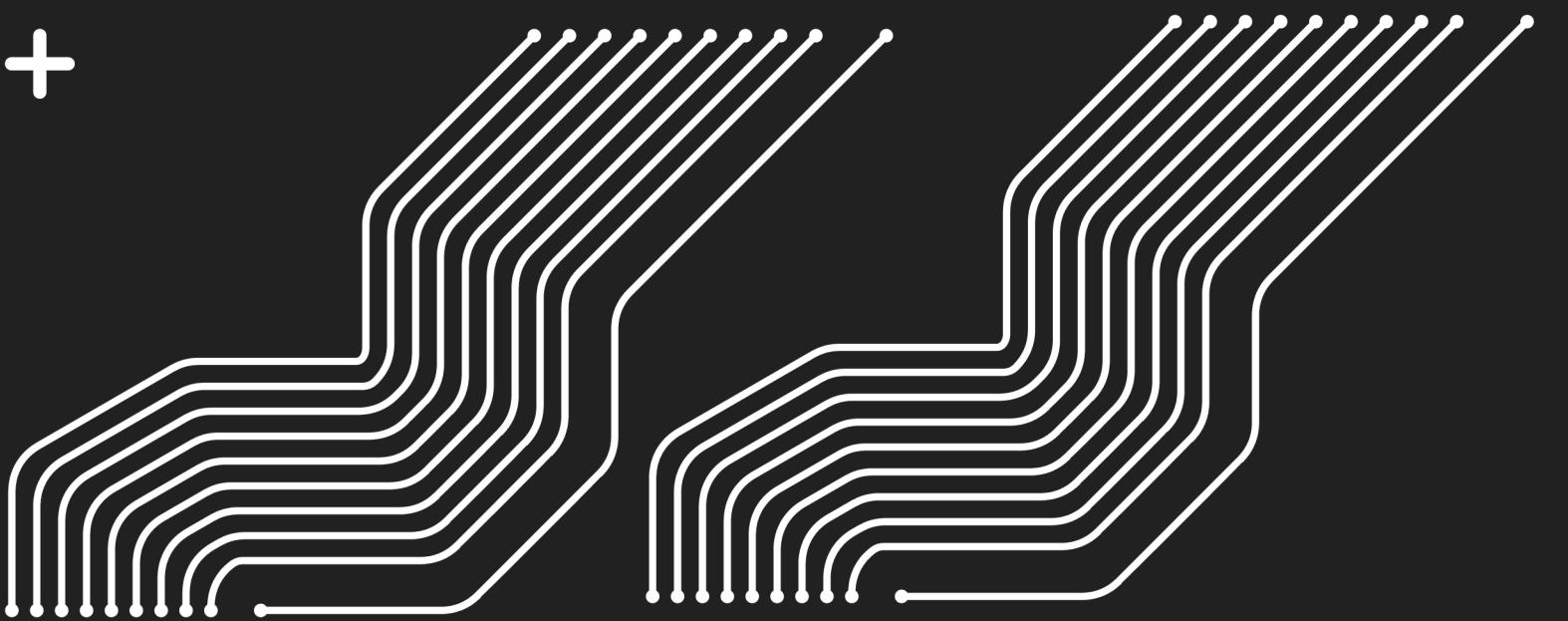


Project realizat de:
Branîte Dragos
Ciumacenco Victor

8-BIT ALU SIMULATION

OBIECTIVELE

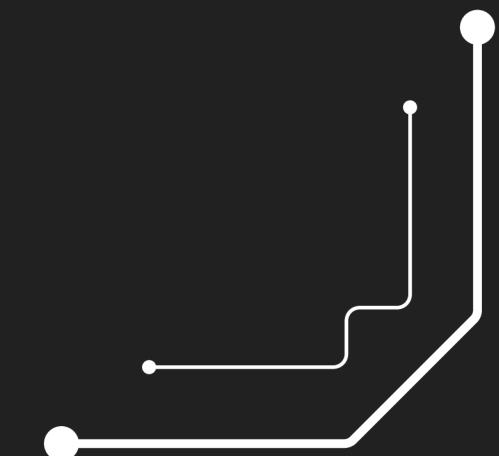


Proiectarea unui 8-Bit ALU utilizând un Limbaj de Descriere Hardware

Implementarea operațiilor:

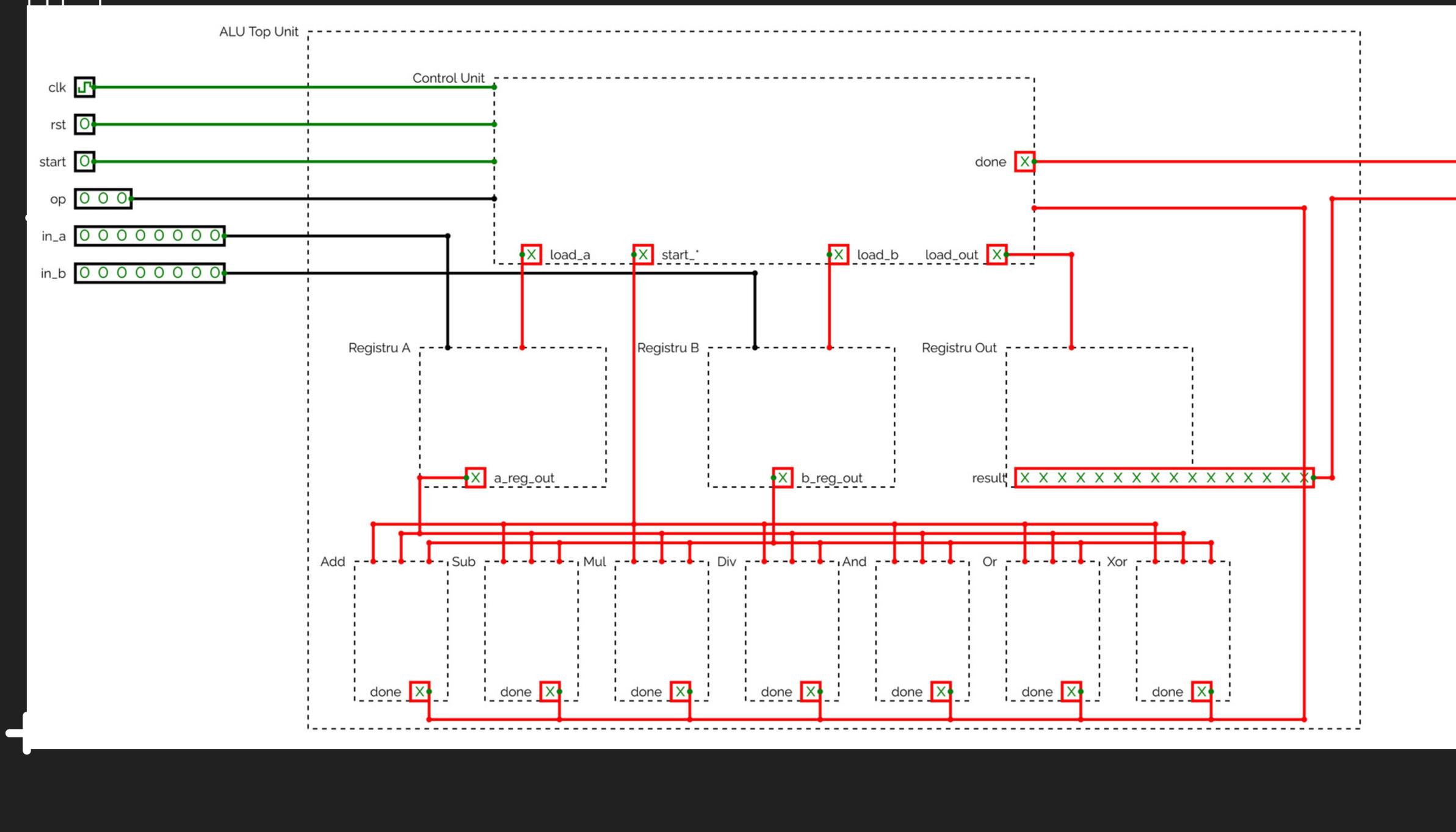
- Adunare
- Scădere
- Înmulțire
- Împărțire

Proiectarea a unui Control Unit capabil să emita semnale de selecție a fiecării operații aritmetice

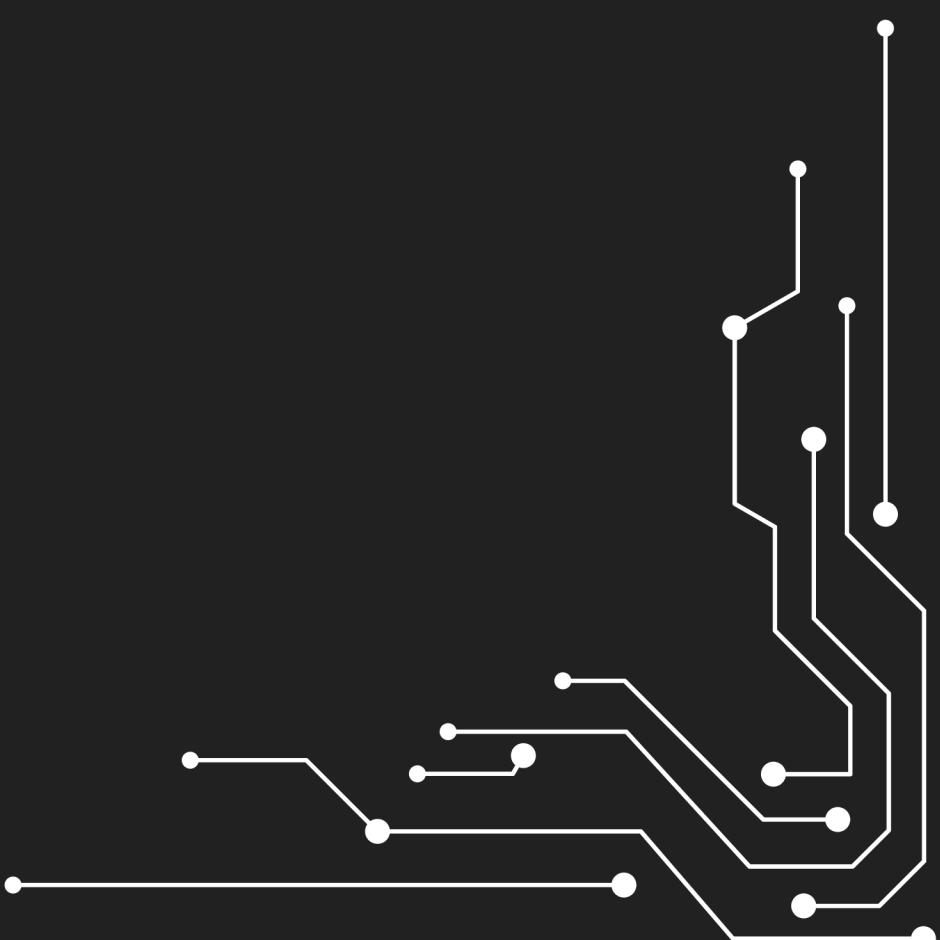


SCHEMA

+



Schema generală simplificată a ALU.



ADUNARE (ȘI SCĂDERE): CARRY LOOK-AHEAD

Un sumator Carry Lookahead (CLA) este mult mai rapid decât un sumator obișnuit Ripple Carry Adder (RCA), deoarece calculează transportul în avans și nu așteaptă propagarea acestuia de la un bit la altul. Acest lucru îl face ideal pentru circuite digitale unde viteza este esențială.



```
CALC: begin
    G = A & B;
    P = A ^ B;

    // Carry lookahead - formula din cursul de AC
    C[0] = 0;
    C[1] = G[0] | (P[0] & C[0]);
    C[2] = G[1] | (P[1] & C[1]);
    C[3] = G[2] | (P[2] & C[2]);
    C[4] = G[3] | (P[3] & C[3]);
    C[5] = G[4] | (P[4] & C[4]);
    C[6] = G[5] | (P[5] & C[5]);
    C[7] = G[6] | (P[6] & C[6]);
    carry_out = G[7] | (P[7] & C[7]);

    S = {carry_out, (P ^ C[7:0])};

    sum <= {{8{S[8]}}}, S;
    done <= 1;
end
```

OPERAȚII LOGICE:

AND, OR SI XOR

Am utilizat operatori pe biți pentru a efectua operațiile date.



```
// AND
CALC: begin
    res <= A & B;
    done <= 1;
end

// OR
CALC: begin
    res <= A | B;
    done <= 1;
end

// XOR
CALC: begin
    res <= A ^ B;
    done <= 1;
end
```

ÎNMULTIRE:

BOOTH MODIFIED (RADIX 2)

Algoritmul Booth modificat Radix-2 optimizează înmulțirea numerelor binare cu semn prin analizarea perechilor de biți ai multiplicatorului, reducând astfel numărul de operații necesare. Am utilizat acest algoritm spre deosebire de varianta cu Radix 4 sau 8 datorită faptului că această variantă e mult mai simplu de implementat, mai ales în contextul proiectului.

```
CALC: begin
    // 1: Operatie bazata pe bitii curenti
    case ({Q[0], Q_m1})
        // Adaugam multiplicandul
        2'b01: temp_A = A + M;
        // Scadem multiplicandul
        2'b10: temp_A = A - M;
        // Nici o operatie
        default: temp_A = A;
    endcase

    // 2. Shiftare aritmetica la dreapta a {A, Q, Q_m1}, cu pastrarea bitului de semn
    shifted_A = {temp_A[7], temp_A[7:1]};
    shifted_Q = {temp_A[0], Q[7:1]};
    shifted_Q_m1 = Q[0];

    // Registri updatati
    A <= shifted_A;
    Q <= shifted_Q;
    Q_m1 <= shifted_Q_m1;
    count <= count + 1;

    // 3. Setarea produsului dupa 8 iteratii
    if (count == 3'd7) begin
        product <= {shifted_A, shifted_Q};
        done <= 1;
    end
end
```

ÎMPĂRTIIRE:

NON-RESTORING DIVISION

Algoritmul Non-Restoring Division realizează împărțirea binară fără a restabili restul după fiecare scădere, folosind alternanța între adunare și scădere. Am ales acest algoritm deoarece e mai eficient ca restoring division, însă mai simplu de implementat decât SRT 2 sau 4.

```
CALC: begin
    // 1. Se shiftă la stanga {A[14:0], Q, 1'b0}
    {next_A, next_Q} = {A[14:0], Q, 1'b0};

    // 2. Operatii bazate pe primul bit
    if (next_A[15] == 1'b0) begin
        // Se scade M
        next_A = next_A - {8'd0, M};
        next_Q[0] = 1'b1;
    end else begin
        // Se aduna M
        next_A = next_A + {8'd0, M};
        next_Q[0] = 1'b0;
    end

    // 3. Setarea produsului
    A <= next_A;
    Q <= next_Q;
    count <= count + 4'd1;
end
```

CONTROL UNIT:

Control unit-ul gestionează pașii principali ai unei operații prin intermediul unui FSM: încărcarea datelor de intrare, pornirea operației dorite (cum ar fi adunare, scădere, înmulțire, etc.), așteptarea finalizării operației și apoi salvarea rezultatului.

În funcție de codul operației (opcode), control unit-ul emite semnale specifice pentru a porni modulul aritmetic sau logic corespunzător (de exemplu, start_add pentru adunare sau start_and pentru operația AND). Practic, acest modul se ocupă să trimită comenziile potrivite la momentul potrivit, astfel încât fiecare operație să fie executată corect și la timp.



```
LOAD: begin
    // Semnalul de load
    load_a = 1'b1;
    load_b = 1'b1;
end

EXECUTE: begin
    // 2. Inceperea operatiei pe datele incarcate in registrii de input
    case (opcode_latched)
        3'b000: start_add = 1'b1;
        3'b001: start_sub = 1'b1;
        3'b010: start_mul = 1'b1;
        3'b011: start_div = 1'b1;
        3'b100: start_and = 1'b1;
        3'b101: start_or = 1'b1;
        3'b110: start_xor = 1'b1;
        default:;
    endcase
end

STORE: begin
    load_out = 1'b1;
end
```

TESTBENCH:

Testbench-ul simulează funcționarea ALU-ului, generând semnale de clk, reset și diverse operații (adunare, scădere, înmulțire, împărțire, AND, OR, XOR) pe care le aplică pe rând. Pentru fiecare operație, testbench-ul setează valorile de intrare, pornește operația și așteaptă semnalul de finalizare (done), după care afișează rezultatul obținut.

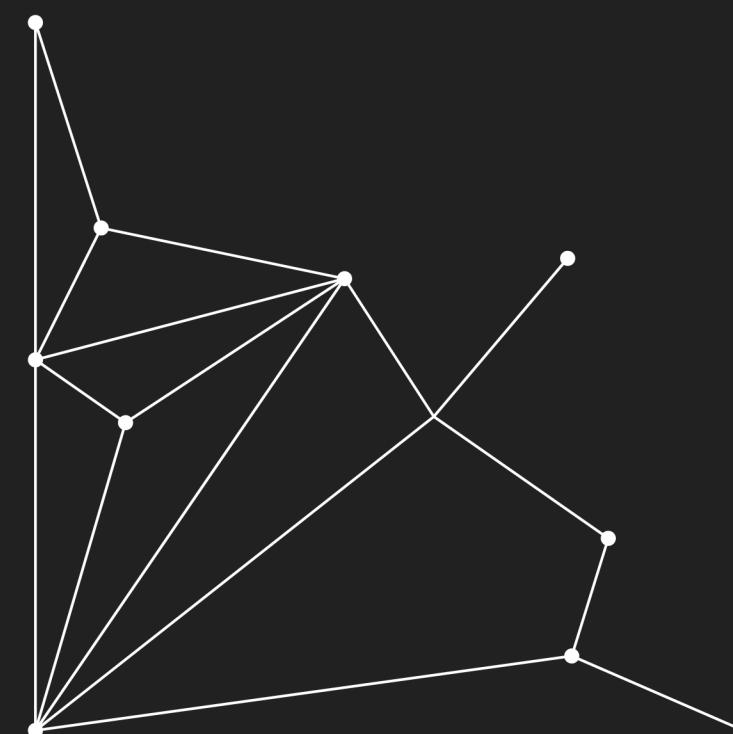
Acest cod verifică dacă ALU-ul răspunde corect la diverse combinații de date și operații.

```
task test_operation(input [2:0] op,
                    input signed [7:0] a,
                    input signed [7:0] b,
                    input [127:0] opname
);
begin
    // Sincronizare cu clock-ul
    @(negedge clk);
    alu_op = op;
    operand_a = a;
    operand_b = b;
    start = 1;
    // Scoate start dupa un ciclu de clock
    @(negedge clk);
    start = 0;
    // Asteapta semnalul de done
    @(posedge done);
    // Asteapta un ciclu adaugator de clock
    @(negedge clk);
    display_state(opname);
    // Cativa cicli de clock intre fiecare operatie
    #20;
end
endtask
```

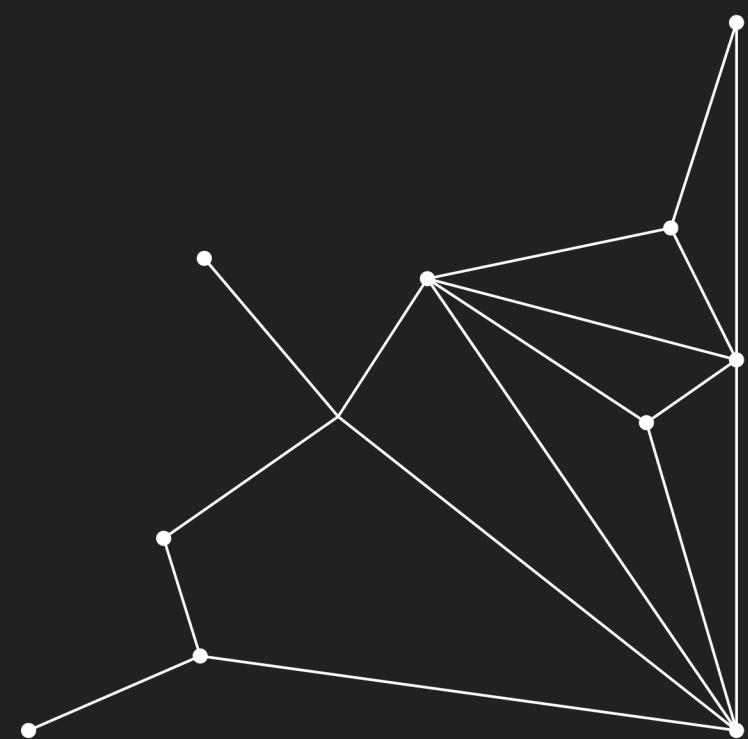
REZULTATE TESTBENCH-ULUI

```
● ● ●  
# ===== STARTING ALU TESTBENCH =====  
# T= 110000 | ADD Overflow+ | A= 127, B= 1 => Result= 128 | Done=1  
# T= 200000 | ADD Overflow- | A= -128, B= -1 => Result= -129 | Done=1  
# T= 290000 | ADD Zero | A= 0, B= 0 => Result= 0 | Done=1  
# T= 380000 | ADD - - | A= -50, B= -50 => Result= -100 | Done=1  
# T= 470000 | SUB Max | A= 127, B= 127 => Result= 0 | Done=1  
# T= 560000 | SUB Zero | A= 0, B= 0 => Result= 0 | Done=1  
# T= 650000 | SUB + - | A= 50, B= -25 => Result= 75 | Done=1  
# T= 740000 | SUB - + | A= -50, B= 25 => Result= -75 | Done=1  
# T= 830000 | SUB - - | A= -50, B= -50 => Result= 0 | Done=1  
# T= 990000 | MUL Overflow+ | A= 127, B= 2 => Result= 254 | Done=1  
# T= 1150000 | MUL Zero | A= 0, B= 0 => Result= 0 | Done=1  
# T= 1310000 | MUL + - | A= 10, B= -10 => Result= -100 | Done=1  
# T= 1470000 | MUL - + | A= -10, B= 10 => Result= -100 | Done=1  
# T= 1630000 | MUL - - | A= -10, B= -10 => Result= 100 | Done=1  
# T= 1790000 | MUL 1 Max | A= 1, B= 127 => Result= 127 | Done=1  
# T= 1950000 | MUL -1 Max | A= -1, B= 127 => Result= -127 | Done=1  
# T= 2110000 | MUL 1 Min | A= 1, B= -128 => Result= -128 | Done=1  
# T= 2270000 | MUL -1 Min | A= -1, B= -128 => Result= 128 | Done=1  
# T= 2450000 | DIV Max | A= 127, B= 1 => Result= 127 | Done=1  
# T= 2570000 | DIV ZeroNum | A= 0, B= 1 => Result= 127 | Done=1  
# T= 2690000 | DIV + - | A= 100, B= -10 => Result= 0 | Done=1  
# T= 2810000 | DIV - + | A= -100, B= 10 => Result= -10 | Done=1  
# T= 2930000 | DIV - - | A= -100, B= -10 => Result= -10 | Done=1  
# T= 3050000 | DIV Div0 | A= 10, B= 0 => Result= 0 | Done=1  
# T= 3140000 | DIV 0/0 | A= 0, B= 0 => Result= 0 | Done=1  
# ===== ALU TESTBENCH COMPLETE =====
```

MULTUMIM PENTRU ATENȚIE!



PROIECT REALIZAT DE:
BRANIȘTE DRAGOȘ
CIUMACENCO VICTOR



+

+