Proiect realizat la laboratorul de CN de către
Braniște Dragoș - 2 TI 1.1
Ciumacenco Victor - 2 TI 1.1

8 bit ALU

Unitate aritmetică și logică pe 8 biți - documentație

Link-uri utile:
https://github.com/branisted/8-bit-ALU

1. Unitatea aritmetică

1.1 Modulul de adunare (și scădere)

Acest modul implementează un CLA (Carry Lookahead Adder) pentru a obţine timpi de calcul constanţi, independenți de lungimea cuvântului, prin calculul anticipat al semnalelor de transport. Generatorii de carry (`G = A & B`) și propagatorii (`P = A ^ B`) sunt calculați simultan, iar vectorul de carry-uri intermediare se obține în același ciclu de ceas, reducând latența comparativ cu un adder în cascadă. Rezultatul extins pe 16 biți cu semn este obținut prin semn-extensie a semifabricatelor de sumă, facilitând detectarea corectă a overflow-ului. În modul de scădere, operandul B este complementat la doi, iar treapta inițială de `C[0] = 1` asigură adăugarea corectă a unui unit bit de împrumut. Acest design oferă un bun echilibru între complexitate logica și performanță în aplicaţii ce necesită calcule rapide pe 8 biți cu semn.

```
`timescale 1ns/1ps

module alu_add (
    input clk,
    input reset,
    input start,
    input signed [7:0] a, b,
    output reg signed [15:0] sum,
    output reg done
);

    localparam IDLE = 3'b000,
               INIT = 3'b001,
               CALC = 3'b010,
               DONE = 3'b011;

    reg [2:0] state, next_state;

    reg [7:0] A, B;
    reg [7:0] G, P, C;
    reg [8:0] S;
    reg carry_out;

    always @(posedge clk) begin
        if (reset) begin
            state <= IDLE;
            done <= 0;
            sum <= 0;
        end else begin
            state <= next_state;
        end
    end

    always @(*) begin
        case (state)
            IDLE: next_state = start ? INIT : IDLE;
            INIT: next_state = CALC;
            CALC: next_state = DONE;
            DONE: next_state = IDLE;
            default: next_state = IDLE;
```

```
            C <= 0;
            S <= 0;
            carry_out <= 0;
            sum <= 0;
            done <= 0;
        end else begin
            case (state)
                IDLE: begin
                    done <= 0;
                end

                INIT: begin
                    A <= a;
                    B <= b;
                    done <= 0;
                end

                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

                DONE: begin
```

```
        endcase
    end

    always @(posedge clk) begin
        if (reset) begin
            A <= 0;
            B <= 0;
            G <= 0;
            P <= 0;
```

```
                done <= 0;
            end

            default: begin
                done <= 0;
            end
        endcase
    end
end
endmodule
```

```
`timescale 1ns/1ps

module alu_sub (
    input clk,
    input reset,
    input start,
    input signed [7:0] a, b,
    output reg signed [15:0] diff,
    output reg done
);

    localparam IDLE = 3'b000,
               INIT = 3'b001,
               CALC = 3'b010,
               DONE = 3'b011;

    reg [2:0] state, next_state;

    reg [7:0] A, B;
    wire [7:0] B_comp;
    reg [7:0] G, P, C;
    reg [8:0] S;
    reg carry_out;

    always @(posedge clk) begin
        if (reset) begin
            state <= IDLE;
            done <= 0;
            diff <= 0;
        end else begin
            state <= next_state;
        end
    end

    always @(*) begin
        case (state)
            IDLE: next_state = start ? INIT : IDLE;
            INIT: next_state = CALC;
            CALC: next_state = DONE;
            DONE: next_state = IDLE;
            default: next_state = IDLE;
        endcase
    end

    always @(posedge clk) begin
        if (reset) begin
            A <= 0;
            B <= 0;
            G <= 0;
            P <= 0;
```

```
            C <= 0;
            S <= 0;
            carry_out <= 0;
            diff <= 0;
            done <= 0;
        end else begin
            case (state)
                IDLE: begin
                    done <= 0;
                end

                INIT: begin
                    A <= a;
                    B <= ~b;
                    done <= 0;
                end

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

                    C[0] = 1;
                    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])};

                    diff <= {{8{S[7]}}, S[7:0]};
                    done <= 1;
                end

                DONE: begin
                    done <= 0;
                end

                default: begin
                    done <= 0;
                end
            endcase
        end
    end

endmodule
```

1.2 Modulul de înmulțire

Înmulțirea este realizată prin algoritmul Booth modificat (Radix-2), care combină operaţiile de adunare și shiftare pentru a procesa doi biți ai multiplicatorului la fiecare iterație. În funcţie de perechea curentă $Q[0],\ Q_{-1}$, se alege adunarea, scăderea sau ignorarea multiplicandului, urmând o shiftare aritmetică la dreapta a registrului concatenat $\{A,Q,Q_{-1}\}$. Contorul de 3 biți urmărește numărul de iteraţii, iar la final se obține produsul sincronizat cu semnalul `done`. Implementarea asigură suport pentru numere cu semn şi gestionează eficient cazurile de overflow prin rețea de shiftare cu păstrarea bitului de semn. Astfel, modulul oferă o soluție compactă, potrivită pentru calculatoare încorporate cu resurse hardware limitate.

```
`timescale 1ns/1ps

module alu_mul (
    input clk,
    input reset,
    input start,
    input signed [7:0] a,
    input signed [7:0] b,
    output reg signed [15:0] product,
    output reg done
);

    reg signed [7:0] M;
    reg signed [7:0] A;
    reg [7:0] Q;
    reg Q_m1;
    reg [2:0] count;
    reg [2:0] state, next_state;

    reg signed [7:0] temp_A;
    reg signed [7:0] shifted_A;
    reg [7:0] shifted_Q;
    reg shifted_Q_m1;

    localparam IDLE = 3'b000,
               INIT = 3'b001,
               CALC = 3'b010,
               DONE = 3'b011;

    always @(posedge clk) begin
        if (reset) begin
            state <= IDLE;
            done <= 0;
            product <= 0;
        end else begin
            state <= next_state;
        end
    end

    always @(*) begin
        case (state)
            IDLE: next_state = start ? INIT : IDLE;
            INIT: next_state = CALC;
            CALC: next_state = (count == 3'd7) ? DONE :
CALC;
            DONE: next_state = IDLE;
            default: next_state = IDLE;
        endcase
    end
```

```
            shifted_Q_m1 <= 0;
        end else begin
            case (state)
                IDLE: begin
                    done <= 0;
                end

                INIT: begin
                    M <= a;
                    A <= 0;
                    Q <= b;
                    Q_m1 <= 0;
                    count <= 0;
                    done <= 0;
                end

                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
```

```
    always @(posedge clk) begin
        if (reset) begin
            M <= 0;
            A <= 0;
            Q <= 0;
            Q_m1 <= 0;
            count <= 0;
            done <= 0;
            product <= 0;
            temp_A <= 0;
            shifted_A <= 0;
            shifted_Q <= 0;
```

```
                DONE: begin
                    done <= 0;
                end

                default: begin
                    done <= 0;
                    product <= product;
                end
            endcase
        end
    end
endmodule
```

1.3 Modulul de împărțire

Împărțirea este implementată cu algoritmul Non-Restoring Division, care alternează între scădere și adunare bazate pe semnul restului intermediar. În faza de inițializare, semnele operandului și divizorului sunt memorate pentru corectarea semnului rezultatului, iar ambii operanzi sunt convertiți la valori pozitive dacă este necesar. Fiecare pas de calcul face shiftare la stânga a registrelor `{A,Q}`, urmată de ajustarea cu registrul `M`, setând bitul de cotă în funcție de semnul noului rest. După 8 cicluri, se aplică corecția semnului și semnalul `done` marchează finalizarea operației. Modulul gestionează explicit cazul divizorului zero, returnând zero și semnalând terminarea instantanee în acest scenariu.

```
`timescale 1ns/1ps

module alu_div (
    input clk,
    input reset,
    input start,
    input signed [7:0] a,
    input signed [7:0] b,
    output reg signed [15:0] quotient,
    output reg done
);

    localparam IDLE = 3'b000,
               INIT = 3'b001,
               CALC = 3'b010,
               DONE = 3'b011;

    reg [2:0] state, next_state;
    reg [3:0] count;
    reg a_sign, b_sign;
    reg [15:0] A;
    reg [7:0] Q;
    reg [7:0] M;
    // Semnale intermediare
    reg [15:0] next_A;
    reg [7:0] next_Q;

    always @(posedge clk) begin
        if (reset) begin
            state <= IDLE;
            done <= 1'b0;
            quotient <= 16'sd0;
        end else begin
            state <= next_state;
        end
    end

    always @(*) begin
        case (state)
            IDLE: next_state = start ? INIT : IDLE;
            INIT: next_state = (b == 0) ? DONE : CALC;
            CALC: next_state = (count == 4'd8) ? DONE : CALC;
            DONE: next_state = IDLE;
            default: next_state = IDLE;
        endcase
    end
    always @(posedge clk) begin
        if (reset) begin
            count <= 4'd0;
            A <= 16'd0;
            Q <= 8'd0;
            M <= 8'd0;
            a_sign <= 1'b0;
            b_sign <= 1'b0;
            done <= 1'b0;
            quotient <= 16'sd0;
        end else begi
```

```
            case (state)
                IDLE: begin
                    done <= 1'b0;
                end

                INIT: begin
                    a_sign <= a[7];
                    b_sign <= b[7];
                    Q <= a[7] ? -a : a;
                    M <= b[7] ? -b : b;
                    A <= 16'd0;
                    count <= 4'd0;
                    done <= 1'b0;
                end

                CALC: begin
                    // 1. Se shifteaza la stanga
                    {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 produslui
                    A <= next_A;
                    Q <= next_Q;
                    count <= count + 4'd1;
                end

                DONE: begin
                    // Cazul in care impartitorul e 0
                    if (b == 0) begin
                        quotient <= 16'sd0;
                    end else begin
                        // Se corecteaza semnul
                        if (a_sign ^ b_sign)
                            quotient <= -{8'd0, Q};
                        else
                            quotient <= {8'd0, Q};
                    end
                    done <= 1'b1;
                end

                default: begin
                    done <= 1'b0;
                end
            endcase
        end
    end
endmodule
```

2. Unitatea logică

2.1 Modulul AND

Acest modul realizează operația bit-wise AND între cei doi operanzi pe 8 biți, iar rezultatul este stocat pe 16 biți pentru consistență cu celelalte operații ALU. Mașina de stări determină exact ciclul de ceas în care datele sunt încărcate și momentul în care operația este executată, astfel încât semnalul done reflectă finalizarea strictă a procesării. Latența este de un singur ciclu de ceas după starea INIT, oferind un timp de răspuns predictibil.

```
`timescale 1ns/1ps
// ALU AND
module alu_and (
    input clk,
    input reset,
    input start,
    input [7:0] a, b,
    output reg [15:0] res,
    output reg done
);

    localparam IDLE = 3'b000,
               INIT = 3'b001,
               CALC = 3'b010,
               DONE = 3'b011;

    reg [2:0] state, next_state;

    // Operanzi interni
    reg [7:0] A, B;

    always @(posedge clk) begin
        if (reset) begin
            state <= IDLE;
            done <= 0;
            res <= 0;
        end else begin
            state <= next_state;
        end
    end

    always @(*) begin
        case (state)
            IDLE: next_state = start ? INIT
: IDLE;
            INIT: next_state = CALC;
            CALC: next_state = DONE;
            DONE: next_state = IDLE;
            default: next_state = IDLE;
        endcase
    end

    always @(posedge clk) begin
        if (reset) begin
            A <= 0;
            B <= 0;
            res <= 0;
            done <= 0;
        end else begin
            case (state)
                IDLE: begin
                    done <= 0;
                end

                INIT: begin
                    A <= a;
                    B <= b;
                    done <= 0;
                end

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

                DONE: begin
                    done <= 0;
                end

                default: begin
                    done <= 0;
                end
            endcase
        end
    end
endmodule
```

2.2 Modulul OR

Similar modulului AND, operatorul OR este implementat printr-o operație combinațională `A | B` care se activează în starea CALC. FSM-ul intern asigură două cicluri de propagare (INIT și CALC) înainte de semnalizarea `done`, garantând separarea clară a fazelor de încărcare și calcul. Rezultatul este extins pe 16 biți (păstrându-se semnul), ceea ce permite compatibilitate directă cu circuitul de multiplexare la nivel de top-unit.

```
`timescale 1ns/1ps
// ALU OR
module alu_or (
    input clk,
    input reset,
    input start,
    input [7:0] a, b,
    output reg [15:0] res,
    output reg done
);

    localparam IDLE = 3'b000,
               INIT = 3'b001,
               CALC = 3'b010,
               DONE = 3'b011;

    reg [2:0] state, next_state;

    reg [7:0] A, B;

    always @(posedge clk) begin
        if (reset) begin
            state <= IDLE;
            done <= 0;
            res <= 0;
        end else begin
            state <= next_state;
        end
    end

    always @(*) begin
        case (state)
            IDLE: next_state = start ?
INIT : IDLE;
            INIT: next_state = CALC;
            CALC: next_state = DONE;
            DONE: next_state = IDLE;
            default: next_state = IDLE;
        endcase
    end

    always @(posedge clk) begin
        if (reset) begin
            A <= 0;
            B <= 0;
            res <= 0;
            done <= 0;
        end else begin
            case (state)
                IDLE: begin
                    done <= 0;
                end

                INIT: begin
                    A <= a;
                    B <= b;
                    done <= 0;
                end

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

                DONE: begin
                    done <= 0;
                end

                default: begin
                    done <= 0;
                end
            endcase
        end
    end
endmodule
```

2.3 Modulul XOR

Modulul XOR efectuează operația `A ^ B` într-un singur ciclu combinatorial după pregătirea operandilor, utilizând aceeași structură de FSM cu stările IDLE, INIT, CALC și DONE. Rezultatul bitwise este pe 16 biți pentru uniformitate cu celelalte operații, iar semnalul `done` este generat imediat după atribuirea rezultatului. Datorită naturii simple a XOR-ului, latența calculului este minimă, dar semnalizarea secvențială asigură sincronizarea corectă în rețeaua ALU.

```
`timescale 1ns/1ps
// ALU XOR
module alu_xor (
    input clk,
    input reset,
    input start,
    input [7:0] a, b,
    output reg [15:0] res,
    output reg done
);

    localparam IDLE = 3'b000,
               INIT = 3'b001,
               CALC = 3'b010,
               DONE = 3'b011;

    reg [2:0] state, next_state;

    reg [7:0] A, B;

    always @(posedge clk) begin
        if (reset) begin
            state <= IDLE;
            done <= 0;
            res <= 0;
        end else begin
            state <= next_state;
        end
    end

    always @(*) begin
        case (state)
            IDLE: next_state = start ?
INIT : IDLE;
            INIT: next_state = CALC;
            CALC: next_state = DONE;
            DONE: next_state = IDLE;
            default: next_state = IDLE;
        endcase
    end

    always @(posedge clk) begin
        if (reset) begin
            A <= 0;
            B <= 0;
            res <= 0;
            done <= 0;
        end else begin
            case (state)
                IDLE: begin
                    done <= 0;
                end

                INIT: begin
                    A <= a;
                    B <= b;
                    done <= 0;
                end

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

                DONE: begin
                    done <= 0;
                end

                default: begin
                    done <= 0;
                end
            endcase
        end
    end
endmodule
```

3. Elemente de control

3.1 Unitatea de Control

Unitatea de control este un FSM cu patru stări (IDLE, LOAD, EXECUTE, STORE) care orchestrează toate operațiile ALU: încărcarea registrelor de intrare, pornirea operației selectate, așteptarea semnalului `op_done` și stocarea rezultatului. Opcode-ul este capturat în IDLE doar la semnalul `start`, apoi este folosit în EXECUTE pentru a activa numai semnalul de start al modulului corespunzător. Multiplexarea semnalelor de done printr-un vector intern permite detectarea unică a finalizării oricărei operații și tranziția sigură către STORE. Această structură centralizată favorizează adăugarea rapidă de noi operații.

```
`timescale 1ns/1ps

module alu_control (
    input  wire        clk,
    input  wire        reset,
    input  wire        start,
    input  wire [2:0]  opcode,
    input  wire        op_done,

    output reg         done,
    output reg         load_a,
    output reg         load_b,
    output reg         load_out,

    output reg         start_add,
    output reg         start_sub,
    output reg         start_mul,
    output reg         start_div,
    output reg         start_and,
    output reg         start_or,
    output reg         start_xor,

    output reg [2:0]   sel_op
);

    localparam IDLE    = 2'b00,
               LOAD    = 2'b01,
               EXECUTE = 2'b10,
               STORE   = 2'b11;

    reg [1:0] current_state, next_state;
    reg [2:0] opcode_latched;

    always @(posedge clk or posedge reset) begin
        if (reset) begin
            current_state  <= IDLE;
            done           <= 1'b0;
            opcode_latched <= 3'b000;
        end else begin
            current_state  <= next_state;
            done           <= (current_state ==
```

```
            default: next_state = IDLE;
        endcase
    end

    always @(*) begin
        load_a    = 1'b0;
        load_b    = 1'b0;
        load_out  = 1'b0;

        start_add = 1'b0;
        start_sub = 1'b0;
        start_mul = 1'b0;
        start_div = 1'b0;
        start_and = 1'b0;
        start_or  = 1'b0;
        start_xor = 1'b0;

        sel_op    = opcode_latched;

        case (current_state)
            IDLE: begin

            end

            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;
```

```
STORE);
            if (current_state == IDLE && start)
                opcode_latched <= opcode;
        end
    end

    always @(*) begin
        case (current_state)
            IDLE:    next_state = start ? LOAD
: IDLE;
            LOAD:    next_state = EXECUTE;
            EXECUTE: next_state = op_done ?
STORE   : EXECUTE;
            STORE:   next_state = IDLE;
```

```
                    default:;
                endcase
            end

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

endmodule
```

3.2 ALU Top Unit

Top Unit-ul interconectează controlerul, registrele de intrare `reg_a` și `reg_b`, toate modulele aritmetice și logice, plus multiplexoarele de selecție a rezultatului și semnalului `done`. Input-urile de tip `clk`, `reset` și `start` propagă sincronizarea și semnalele de inițiere în toată structura, în timp ce un mux8to1 pe 16 biți selectează rezultatul final pe baza codului de operație. Designul modular și parametrizat permite extinderea la operații pe 16 sau 32 biți și integrarea directă într-un pipeline CPU. Semnalul global `done` este generat din starea STORE a control unit-ului, garantând coerența semnalelor de terminare a operației.

```
`timescale 1ns/1ps

module alu_top (
    input clk,
    input reset,
    input start,
    input [2:0] op,
    input signed [7:0] in_a,
    input signed [7:0] in_b,
    output signed [15:0] result,
    output done
);

    wire load_a, load_b, load_out;
    wire start_mul, start_div, start_add,
start_sub, start_and, start_or, start_xor;
    wire add_done, sub_done, and_done,
or_done, xor_done, mul_done, div_done;
    wire [2:0] sel_op;

    wire [7:0] a_reg_out, b_reg_out;
    wire [15:0] alu_result_add,
alu_result_sub, alu_result_and,
alu_result_or, alu_result_xor;
    wire [15:0] alu_result_mul,
alu_result_div;

    // Selectarea semnalului done
    wire [7:0] done_signals;
    assign done_signals[0] = add_done;
    assign done_signals[1] = sub_done;
    assign done_signals[2] = mul_done;
    assign done_signals[3] = div_done;
    assign done_signals[4] = and_done;
    assign done_signals[5] = or_done;
    assign done_signals[6] = xor_done;
    assign done_signals[7] = 1'b0;

    mux8to1 op_done_mux (
        .in(done_signals),
        .load_a(load_a),
        .load_b(load_b),
        .load_out(load_out),
        .start_add(start_add),
        .start_sub(start_sub),
        .start_mul(start_mul),
        .start_div(start_div),
        .start_and(start_and),
        .start_or(start_or),
        .start_xor(start_xor),
        .sel_op(sel_op)
    );

    // Registri de input
    regn #(8) reg_a (.clk(clk),
.en(load_a), .d(in_a), .q(a_reg_out));
    regn #(8) reg_b (.clk(clk),
.en(load_b), .d(in_b), .q(b_reg_out));

    // Instantierea modulelor aritmetice
    alu_add add_unit (.clk(clk),
.reset(reset), .a(a_reg_out),
.b(b_reg_out), .start(start_add),
.sum(alu_result_add), .done(add_done));

    alu_sub sub_unit (.clk(clk),
.reset(reset), .a(a_reg_out),
.b(b_reg_out), .start(start_sub),
.diff(alu_result_sub), .done(sub_done));

    alu_and and_unit (.clk(clk),
.reset(reset), .a(a_reg_out),
.b(b_reg_out), .start(start_and),
.res(alu_result_and), .done(and_done));

    alu_or  or_unit  (.clk(clk),
.reset(reset), .a(a_reg_out),
.b(b_reg_out), .start(start_or),
.res(alu_result_or), .done(or_done));
```

```
        .sel(op),
        .y(op_done)
    );

    // Instantierea control unit-ului
    alu_control ctrl (
        .clk(clk),
        .reset(reset),
        .start(start),
        .opcode(op),
        .op_done(op_done),
        .done(done),
```

```
    alu_xor xor_unit (.clk(clk),
.reset(reset), .a(a_reg_out),
.b(b_reg_out), .start(start_xor),
.res(alu_result_xor), .done(xor_done));

    alu_mul mul_unit (.clk(clk),
.reset(reset), .start(start_mul),
.a(a_reg_out), .b(b_reg_out),
.product(alu_result_mul), .done(mul_done));

    alu_div div_unit (.clk(clk),
.reset(reset), .start(start_div),
.a(a_reg_out), .b(b_reg_out),
.quotient(alu_result_div),
.done(div_done));

    // Mux 8 to 1 16 bit
    wire [15:0] result_mux;

    mux8to1_16bit result_mux_inst (
        .in0(alu_result_add),
        .in1(alu_result_sub),
        .in2(alu_result_mul),
        .in3(alu_result_div),
        .in4(alu_result_and),
        .in5(alu_result_or),
        .in6(alu_result_xor),
        .in7(16'h0000),
        .sel(sel_op),
        .y(result_mux)
    );

    // Registru de iesire
    regn #(16) reg_out (.clk(clk),
.en(load_out), .d(result_mux), .q(result));

endmodule
```

4. Testing

4.1 Testbench-ul

Testbench-ul `tb_alu_top` verifică funcționarea completă a ALU Top Unit, incluzând cazuri de overflow, semne mixte, zero, diviziune la zero și operații pe biți. Utilizează un clock de 10 ns per perioadă și un task `test_operation` care aplică perechi de operanzi, așteaptă semnalul `done` și afișează rezultatele cu `$display`. Fișierele VCD generate permit vizualizarea formelor de undă în GTKWave pentru depanare detaliată.

```
`timescale 1ns / 1ps

module tb_alu_top();

    reg clk = 0;
    reg rst = 1;
    reg start = 0;
    reg [2:0] alu_op = 3'b000;
    reg [7:0] operand_a = 8'h00;
    reg [7:0] operand_b = 8'h00;

    wire done;
    wire [15:0] result;

    alu_top DUT (
        .clk(clk),
        .reset(rst),
        .start(start),
        .op(alu_op),
        .in_a(operand_a),
        .in_b(operand_b),
        .done(done),
        .result(result)
    );

    always #5 clk = ~clk;

    task display_state(input [127:0]
opname);
        $display("T=%8t | %-14s | A=%5d,
B=%5d => Result=%7d | Done=%b",
            $time, opname,
$signed(operand_a), $signed(operand_b),
$signed(result), done);
    endtask


    initial begin
        $dumpfile("tb_alu_top.vcd");
        test_operation(3'b010, -10, 10,
"MUL - +");
        test_operation(3'b010, -10, -10,
"MUL - -");
        test_operation(3'b010, 1, 127, "MUL
1 Max");
        test_operation(3'b010, -1, 127,
"MUL -1 Max");
        test_operation(3'b010, 1, -128,
"MUL 1 Min");
        test_operation(3'b010, -1, -128,
"MUL -1 Min");

        // Impartire
        test_operation(3'b011, 127, 1, "DIV
Max");
        test_operation(3'b011, 0, 1, "DIV
ZeroNum");
        test_operation(3'b011, 100, -10,
"DIV + -");
        test_operation(3'b011, -100, 10,
"DIV - +");
        test_operation(3'b011, -100, -10,
"DIV - -");
        test_operation(3'b011, 10, 0, "DIV
Div0");
        test_operation(3'b011, 0, 0, "DIV
0/0");

        // AND, OR, XOR cu diverse
combinatii de biti
        test_operation(3'b100, 8'b11110000,
8'b10101010, "AND 1");
        test_operation(3'b100, 8'b00001111,
8'b11110000, "AND 2");
        test_operation(3'b100, 8'b11111111,
8'b00000000, "AND 3");
        test_operation(3'b100, 8'b10101010,
8'b01010101, "AND 4");
```

```
        $dumpvars(0, tb_alu_top);
    end

    initial begin
        $display("==== STARTING ALU
TESTBENCH ====");

        #20 rst = 0;
        #20;

        // Adunare
        test_operation(3'b000, 127, 1, "ADD
Overflow+");
        test_operation(3'b000, -128, -1,
"ADD Overflow-");
        test_operation(3'b000, 0, 0, "ADD
Zero");
        test_operation(3'b000, -50, -50,
"ADD - -");

        // Scadere
        test_operation(3'b001, 127, 127,
"SUB Max");
        test_operation(3'b001, 0, 0, "SUB
Zero");
        test_operation(3'b001, 50, -25,
"SUB + -");
        test_operation(3'b001, -50, 25,
"SUB - +");
        test_operation(3'b001, -50, -50,
"SUB - -");

        // Inmultire
        test_operation(3'b010, 127, 2, "MUL
Overflow+");
        test_operation(3'b010, 0, 0, "MUL
Zero");
        test_operation(3'b010, 10, -10,
"MUL + -");
```

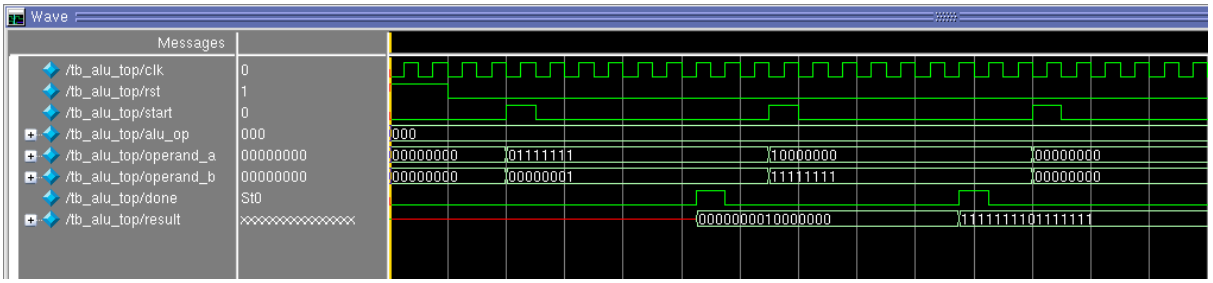
```
        test_operation(3'b101, 8'b11110000,
8'b10101010, "OR 1");
        test_operation(3'b101, 8'b00001111,
8'b11110000, "OR 2");
        test_operation(3'b101, 8'b11111111,
8'b00000000, "OR 3");
        test_operation(3'b101, 8'b10101010,
8'b01010101, "OR 4");

        test_operation(3'b110, 8'b11110000,
8'b10101010, "XOR 1");
        test_operation(3'b110, 8'b00001111,
8'b11110000, "XOR 2");
        test_operation(3'b110, 8'b11111111,
8'b00000000, "XOR 3");
        test_operation(3'b110, 8'b10101010,
8'b01010101, "XOR 4");

        $display("==== ALU TESTBENCH
COMPLETE ====");
        $stop;
    end

    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

endmodule
```

5. Vizualizarea funcționalității

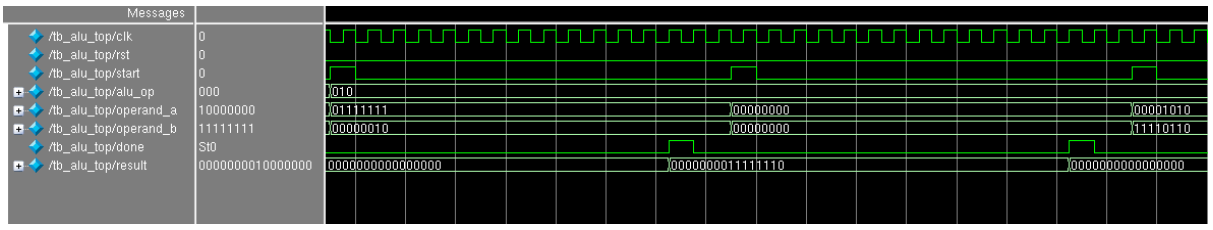
Funcționalitatea completă a ALU-ului a fost verificată printr-un testbench structural (`tb_alu_top.v`) care simulează secvențial toate operațiile suportate, inclusiv adunare, scădere, multiplicare, împărțire, AND, OR și XOR. Fiecare operație este testată prin apeluri repetitive ale unui task dedicat, care aplică perechi de operanzi și monitorizează semnalul `done` pentru a asigura sincronizarea corectă a execuției.

Pentru vizualizarea comportamentului intern, s-au generat fișiere de tip VCD (`.vcd`) care au fost analizate cu ajutorul instrumentului GTKWave. În cadrul acestei vizualizări au fost urmărite tranzițiile semnalului de ceas `clk`, sincronizarea dintre semnalele `start` și `done`, evoluția registrelor de intrare `a`, `b`, precum și a registrului `opcode` și a semnalului de ieșire `res`. De asemenea, au fost observate cu atenție stările mașinii de stări finite (`alu_control`), inclusiv tranziția corectă între stările `IDLE`, `LOAD`, `EXECUTE` și `STORE`. Vizualizarea a confirmat activarea corectă a fiecărui modul operațional individual (cum ar fi `alu_add`, `alu_mul` sau `alu_div`) în funcție de codul de operație setat.

Exemplu operații adunare:



Exemplu înmulțiri:



```
# Top level modules:
#       tb_alu_top
# vsim tb_alu_top
# Loading work.tb_alu_top
# Loading work.alu_top
# Loading work.mux8to1
# Loading work.mux2to1
# Loading work.alu_control
# Loading work.regn
# Loading work.alu_add
# Loading work.alu_sub
# Loading work.alu_and
# Loading work.alu_or
# Loading work.alu_xor
# Loading work.alu_mul
# Loading work.alu_div
# Loading work.mux8to1_16bit
# ==== 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
# T= 3230000 | AND 1          | A=  -16, B=  -86 => Result=    160 | Done=1
# T= 3320000 | AND 2          | A=   15, B=  -16 => Result=      0 | Done=1
# T= 3410000 | AND 3          | A=   -1, B=    0 => Result=      0 | Done=1
# T= 3500000 | AND 4          | A=  -86, B=   85 => Result=      0 | Done=1
# T= 3590000 | OR 1           | A=  -16, B=  -86 => Result=    250 | Done=1
# T= 3680000 | OR 2           | A=   15, B=  -16 => Result=    255 | Done=1
# T= 3770000 | OR 3           | A=   -1, B=    0 => Result=    255 | Done=1
# T= 3860000 | OR 4           | A=  -86, B=   85 => Result=    255 | Done=1
# T= 3950000 | XOR 1          | A=  -16, B=  -86 => Result=     90 | Done=1
# T= 4040000 | XOR 2          | A=   15, B=  -16 => Result=    255 | Done=1
# T= 4130000 | XOR 3          | A=   -1, B=    0 => Result=    255 | Done=1
# T= 4220000 | XOR 4          | A=  -86, B=   85 => Result=    255 | Done=1
# ==== ALU TESTBENCH COMPLETE ====
# Break in Module tb_alu_top at ../testbenches/tb_alu_top.v line 95
```

6. Observații

Deși proiectul atinge în mare parte obiectivele propuse și funcționalitatea generală este corectă, există o serie de aspecte care ar putea fi îmbunătățite din punct de vedere al stilului de implementare și al rigurozității arhitecturale. Unul dintre punctele cele mai evidente este utilizarea frecventă a codului comportamental, în special în implementarea modulelor de tip FSM și în unitățile aritmetice complexe, cum ar fi multiplicatorul Booth sau divizorul non-restoring. Deși aceste blocuri respectă logica secvențială, folosirea expresiilor `always @(posedge clk)`, a instrucțiunilor `if`, `case`, precum și a variabilelor temporare intermediare în loc de instanțieri structurale stricte (muxuri, registre, comparatoare) contravine cerințelor unui design complet structural. Această abordare reduce modularitatea și dificultatea de sintetizare pe un FPGA real, unde resursele logice trebuie să fie clar definite.

Un alt aspect discutabil este lipsa unei separări clare între logica de control și cea de date în unele module, ceea ce îngreunează depanarea și reutilizarea componentelor în contexte diferite. În plus, nu sunt implementate flaguri standard precum `zero`, `negative`, `carry` sau `overflow`, care ar fi esențiale într-un ALU complet funcțional, utilizabil într-un procesor real. De asemenea, structura de testare, deși funcțională, este construită tot în stil comportamental, folosind task-uri și semnale de control verificate cu așteptări explicite (`wait(done)`) în locul unui cadru mai formal de validare automatizată.

În concluzie, deși ALU e funcțional, stilul de implementare combină prea multă logică comportamentală cu cea structurală. Pentru o versiune îmbunătățită și scalabilă, am putea refactoriza complet codul în stil strict structural, să implementăm unele flaguri de stare și să separăm clar task-urile între unitatea de control și datapath.