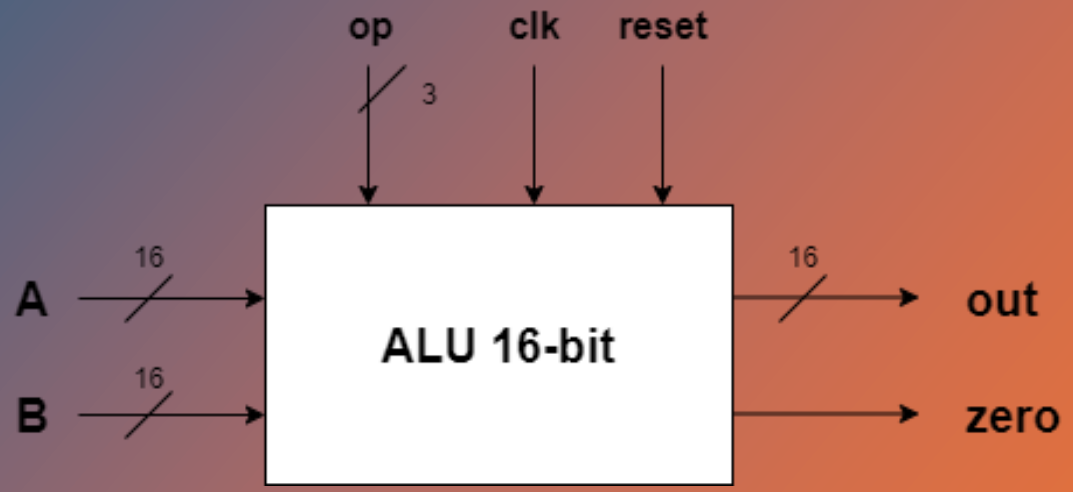
Modul ALU 16-biti în Verilog

Blaga Cristian- Marian

Bîzoi Fabian- Mario

Daichendt Ioana- Patricia



ALU 16-biti

- Acest proiect implementează un modul Arithmetic Logic Unit (ALU) în Verilog. ALU poate efectua diverse operații aritmetice și logice în funcție de codul operațional dat.

ALU 16-biti

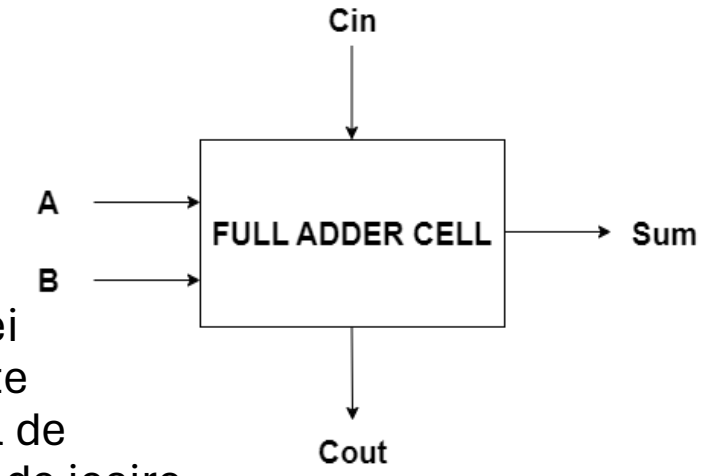
Am efectuat următoarele operații:

- - Adunare
- - Scădere
- - Înmulțire
- - Împărțire
- - Shiftare la stânga
- - Shiftare la dreapta

- - XOR
- - XNOR
- - NOT
- - AND
- - OR
- - Înmulțire Booth Radix 4
- - Împărțire Non-Restoring

- Fiecare operație este implementată ca un modul separat în cadrul proiectului.

FullAdder



- Modulul FullAdder: Acesta primește trei intrări și furnizează două ieșiri. Sum este calculat ca XOR între A, B și transportul de intrare Cin. Cout reprezintă transportul de ieșire și este calculat ca un OR.
- Modulul ADD: Acesta primește două intrări pe 16 biți și furnizează o ieșire pe 17 biți. Si utilizeaza un vector pentru a gestiona carry-ul. Se apelează modulul full_add pentru a efectua adunarea pe un singur bit. Se stabilește ultimul bit al sumei (sum[16]) ca fiind carry-ul ultimului bit adunat.
- Modulul full_add: Acesta primește trei intrări și furnizează două ieșiri. Sum reprezintă suma binară a celor trei intrări, calculată prin XOR.

```
// Define a Full Adder module
module FullAdder (
    input signed  A,
    input signed B,
    input Cin,
    output signed Sum,
    output Cout
);

    assign Sum = A ^ B ^ Cin;
    assign Cout = (A & B) | (A & Cin) | (B & Cin);

endmodule // FullAdder

module ADD(A, B, sum);
  input signed [15:0] A,B;
  output signed [16:0] sum;
  wire [16:0] C;

  genvar i;
  // sets initial carryin to 0
  assign C[0] = 0;

  generate
    for (i = 0; i < 16; i = i+1)
    begin:addbit
      full_add stage(A[i],B[i],C[i],sum[i],C[i+1]);
    end
  endgenerate

  assign sum[16] = C[16];  //carry overflow, if the last carry is 1

endmodule // ADD


module full_add(input A, input B, input C0, output sum, output C1);
  assign sum = A^B^C0;
  assign C1 = (A&B)|(A&C0)|(B&C0);
endmodule // full_add
```

```
module SUB(A, B, difference);
    input signed [15:0] A, B;
    output signed [15:0] difference;

    assign difference = A + (~ B) +1;
endmodule // SUB


module MUL(A, B, product);
  input signed [15:0] A, B;
  output signed [31:0] product;

  assign product = A * B;
endmodule // MUL
```

- **SUB**:
  - Efectuează scăderea a două numere pe 16 biți, adunând primului operand, negatul celui de al doilea.
- **MUL**:
  - Efectuează înmulțirea a două numere pe 16 biți.

Modulul partialproduct

- Preia un număr întreg semnat de 16 biți input1 și o valoare segment de 3 biți segment ca intrări.
- Calculează un produs parțial bazat pe valoarea segment și îl atribuie ieșirii output1 pe 32 biți.
- Cinci valori diferite de segment:
  - 3'b000: Setează output1 la zero (practic multiplicând cu 0).
  - 3'b001 și 3'b010: multiplicare cu 1
  - 3'b011: Deplasează input1 la stânga cu un bit (multiplicare cu 2).
  - 3'b100 : Neaga input1, aduna 1 si shifteaza la stanga cu un bit(multiplicare cu –2)
  - 3'b101: si 3'b110: Neaga output1 si shifteaza la stanga cu un bit(multiplicare cu –1)
  - 3'b111: Setează output1 la zero (practic multiplicând cu 0).

Block	Partial product
000	0
001	1 * multiplicand
010	1 * multiplicand
011	2 * multiplicand
100	–2 * multiplicand
101	–1 * multiplicand
110	–1 * multiplicand
111	0

```
module partialproduct(input1,segment,output1);
  input [15:0] input1;
  input [2:0] segment;
  output reg [31:0] output1;

      always @(*) begin
    case (segment)
      3'b000:output1=$signed(1'b0);
      3'b001:output1=$signed(input1);
      3'b010:output1=$signed(input1);
      3'b011:
        begin
     output1=$signed(input1);
     output1=$signed(input1)<<<1;
        end
      3'b100:begin
    output1=$signed(input1);
    output1=$signed(~output1+1'b1);
    output1=$signed(output1)<<<1;
      end
      3'b101:begin
    output1=$signed(input1);
    output1=$signed(~output1+1'b1);
      end
      3'b110:begin
    output1=$signed(input1);
    output1=$signed(~output1+1'b1);
      end
      3'b111:output1=$signed(16'b0);
    endcase
  end

endmodule
```

Booth Radix 4

```
module BOOTH_4(A,B,product);
  input [15:0] A;
  input [15:0] B;
  output [31:0] product;
  wire [31:0] temp [14:0];

  partialproduct p0(A,{B[1:0],1'b0},temp[0]);
  partialproduct p1(A,B[3:1],temp[1]);
  partialproduct p2(A,B[5:3],temp[2]);
  partialproduct p3(A,B[7:5],temp[3]);



  assign product = $signed(temp[0])+$signed(temp[1]<<<2)+$signed(temp[2]<<<4)+$signed(temp[3]<<<6);


endmodule
```

- Declară un array temp pentru a stoca produsele parțiale (unul pentru fiecare pereche de biți din B). Fiecare element are 32 de biți.
- Creează patru module partialproduct (p0 până la p3) pentru a calcula produsele parțiale:
  - p0 multiplică A cu cei mai puțin semnificativi doi biți din B (deplasat cu un bit spre dreapta).
  - p1 multiplică A cu următorii doi biți din B (biții 3 și 2).
  - p2 multiplică A cu următorii doi biți (biții 5 și 4), și așa mai departe.
- Asigură produsul final la ieșirea product:
  - temp[0] este adăugat fără deplasare (multiplicare cu bitul cel mai puțin semnificativ al lui B).
  - temp[1] este deplasat la stânga cu doi biți (multiplicare cu al doilea bit cel mai puțin semnificativ al lui B, deplasat cu o poziție) si asa mai departe.

Non-Restoring

- **Registre interne:**
- c: Registru acumulator pentru a păstra valori intermediare în timpul procesului de împărțire
- M: Registru temporar pentru a păstra o copie a divisorului B
- newc: Registru temporar pentru a păstra valoarea actualizată a lui c
- Q: Registru al rezultatului parțial
- n: Numarul de biti
- pp: Registru temporar pentru a păstra produsele partiale
- p: Registru flag pentru a indica starea de inițializare

```
module non_rest_div(A, B, quot, clk);
  input [15:0] A, B;
  output reg [15:0] quot;
  reg [15:0] reminder;

  input clk, reset;

  reg [15:0] c = 0, M, newc;
  reg [15:0] Q;
  reg [15:0] n = 16;
  reg [31:0] pp;

  reg p=0;
```

Non-Restoring

- Verifică dacă semnalul de reset este activ (reset == 1).
- Dacă da, inițializează variabilele necesare pentru o nouă operație de împărțire.
- În caz contrar, procesul efectuează pașii necesari pentru împărțire, precum calculul produselor parțiale și actualizarea variabilelor de stare (c, Q, n).
- Operația continuă până când toți cei 16 biți sunt împărțiți (n == 0), moment în care rezultatul împărțirii și restul sunt finalizate și stocate în ieșiri.

```
reg p=0;

always @(posedge clk)
  begin
if(!reset) begin
   if( ! p) begin
     Q = A;
     M = B;
     p = 1;
  end
else begin
   if(n != 0) begin
      if(c[15] == 0) begin
   pp= {c, Q} << 1;
   newc = pp[31:15] +(~ M + 1);
   Q = pp[31:0];

      end
     else begin
   pp= {c, Q} <<1;
   newc= pp[31:15] + M;
   Q = pp[31:0];
      end
```

```
      c = newc;
      if(c[15] == 0) begin
   Q[0] = 1;
   end
      else begin
   Q[0] = 0;

      end
      n = n-1;
   end // if (n != 0)
end // else: !if( ! p)



if( n == 0) begin
   if( c[15]) begin
      c = c + M;
   end
   quot = Q;
   reminder = c;
end
end // if (!reset)

   end // always @ (posedge clk)

endmodule // Rest_div
```

- **DIV:**
  - Efectueaza impartirea a doua numere pe 16 biti
- **SHIFT_LEFT**:
  - Efectueaza shiftarea numarului A, la stanga, cu B biti
- **SHIFT_RIGHT**:
  - Efectueaza shiftarea numarului A, la dreapta, cu B biti
- **XOR**:
  - Efectueaza operatia logica XOR intre operanzii A si B
- **XNOR**:
  - Efectueaza operatia logica XNOR intre operanzii A si B

```
module DIV(A, B, quotient);
  input signed [15:0] A, B;
  output signed [15:0] quotient;

  assign quotient = A / B;
endmodule // DIV

module SHIFT_LEFT(A, B, result);
  input signed [15:0] A;
  input [15:0] B;
  output signed [15:0] result;

  assign result = A << B;
endmodule


module SHIFT_RIGHT(A, B, result);
  input signed [15:0] A;
  input [15:0] B;
  output signed [15:0] result;

  assign result = A >> B;
endmodule // SHIFT_RIGHT


module XOR(A, B, result);
  input signed [15:0] A, B;
  output signed [15:0] result;

  assign result = A ^ B;
endmodule // XOR


module XNOR(A, B, result);
  input signed [15:0] A, B;
  output signed [15:0] result;
  assign result = ~(A ^ B);
endmodule // XNOR
```

```
module NOT(A, result);
  input signed [15:0] A;
  output signed [15:0] result;

  assign result = ~A;
endmodule // NOT


module AND (A, B, result);
    input signed [15:0] A;
    input signed [15:0] B;
    output signed [15:0]   result;

    assign result= A & B;


endmodule


module OR(A, B, result);
  input signed [15:0] A, B;
  output signed [15:0] result;

  assign result = A | B;
endmodule
```

- **AND:**
  - Efectueaza operatia 'si' intre operanzii A si B
- **NOT**:
  - Efectueaza operatia de negare a numarului
- **OR**:
  - Efectueaza operatia 'sau' intre operanzii A si B

ALU

- Se declara variabilele
- Se declara variabile pentru rezultatele input-urilor
- Se instantiaza fiecare modul

```
module ALU16 (
  input signed [15:0] A,
  input signed [15:0] B,
  input [3:0] op,
  output reg signed [15:0] out,
  output reg zero,
  input clk,
  input reset
);

    wire [16:0] sum_add;  //signal for the inputs result
    wire [15:0] difference;
    wire [31:0] product;
    wire [15:0] quotient;
    wire [15:0] shift_left;
    wire [15:0] shift_right;
    wire [15:0] xor_result;
    wire [15:0] xnor_result;
    wire [15:0] not_result;
    wire [15:0] and_result;
    wire [15:0] or_result;
    wire [31:0] product2;
    wire [15:0] quotient2;

    ADD add_inst(.A(A), .B(B), .sum(sum_add));
    SUB sub_inst(.A(A), .B(B), .difference(difference));
    MUL mul_inst(.A(A), .B(B), .product(product));
    BOOTH_4 booth_inst(.A(A), .B(B), .product(product2));
    DIV div_inst(.A(A), .B(B), .quotient(quotient));
    non_rest_div non_rest_div_inst(.A(A), .B(B), .quot(quotient2),.clk(clk));
    SHIFT_LEFT shl_inst(.A(A), .B(B), .result(shift_left));
    SHIFT_RIGHT shr_inst(.A(A), .B(B), .result(shift_right));
    XOR xor_inst(.A(A), .B(B), .result(xor_result));
    XNOR xnor_inst(.A(A), .B(B), .result(xnor_result));
    NOT not_inst(.A(A), .result(not_result));
    AND and_inst(.A(A), .B(B), .result(and_result));
    OR or_inst(.A(A), .B(B), .result(or_result));
```

ALU-control unit

- Procesul always @(*): Acest proces este activat de orice modificare a intrărilor.
- Se verifica coul operatiei
- In functie de ce operatie este, se da valoarea corespunzatoare output-ului si se setează semnalul zero în funcție de faptul dacă rezultatul este zero sau nu

```
// Assign outputs based on operation code
always @(*) begin
  case (op)
    0: begin // (Addition)
      out <= sum_add[16:0];
      zero <= (out == 0); //Assign the variable 'zero' the value 1 if the result is 0, or 0 otherwise
    end
    1: begin // (Subtraction)
      out <= difference[15:0];
      zero <= (out == 0);
    end
    2: begin // (Multiplication)
      out <= product;
      zero <= (out == 0);
    end
    3: begin // (Division)
      out <= quotient[15:0];
      zero <= (out == 0);
    end
    4: begin // (Shift Left)
      out <= shift_left[15:0];
      zero <= (out == 0);
    end
    5: begin // (Shift Right)
      out <= shift_right[15:0];
      zero <= (out == 0);
    end
    6: begin // (XOR)
      out <= xor_result[15:0];
      zero <= (out == 0);
    end
    7: begin // (XNOR)
      out <= xnor_result[15:0];
      zero <= (out == 0);
    end
    8: begin // (NOT)
      out <= not_result[15:0];
      zero <= (out == 0);
    end
    9: begin // (AND)
 out<= and_result[15:0];
 zero <= (out ==0);
    end
```

```
      10: begin // (OR)
    out <= or_result[15:0];
    zero <= (out == 0);
      end
      11: begin //booth
    out<=product2[15:0];
          zero <= (out == 0);
      end
      12: begin //non_restoring division
    out<=quotient2[15:0];
          zero <= (out == 0);
      end

      default: begin
          out <= 0;
          zero <= 1;
      end
    endcase
  end

endmodule
```

Rezultate

```
Test 1: Addition (A =   -529, B =     10)
  Expected output:   -519
  Actual output:   fdf9 (decimal:   -519)
  Zero:             0
Test 2: Subtraction (A =   1245, B =    433)
  Expected output:    812
  Actual output:   032c (decimal:    812)
  Zero:             0
Test 3: Multiplication (A =     30, B =    -41)
  Expected output:  -1230
  Actual output:   fb32 (decimal:  -1230)
  Zero:             0
Test 4: Division (A =  16900, B =     20)
  Expected output:    845
  Actual output:   034d (decimal:    845)
  Zero:             0
Test 5: Shift Left (A =     85, B =      4)
  Expected output:   1360
  Actual output:   0550 (decimal:   1360)
  Zero:             0
Test 6: Shift Right( A =    167, B =      3)
 Expected output:     20
 Actual output:   0014 (decimal:     20)
 Zero:            0
Test 7: XOR( A =      8, B =     12)
 Expected output:      4
 Actual output:   0004 (decimal:      4)
 Zero:            0
Test 8: XNOR( A =   6553, B =    133)
 Expected output:  -6429
 Actual output:   e6e3 (decimal:  -6429)
 Zero:            0
```

```
Test 6: Shift Right( A =    167, B =      3)
 Expected output:     20
 Actual output:   0014 (decimal:     20)
 Zero:            0
Test 7: XOR( A =      8, B =     12)
 Expected output:      4
 Actual output:   0004 (decimal:      4)
 Zero:            0
Test 8: XNOR( A =   6553, B =    133)
 Expected output:  -6429
 Actual output:   e6e3 (decimal:  -6429)
 Zero:            0
Test 9: NOT( A=    123)
 Expected output:   -124
 Actual output:   ff84 (decimal:   -124)
 Zero:            0
Test 10: AND( A=      3, B=      2)
 Expected output:      2
 Actual autput:   0002 (decimal:      2)
 Zero:            0
Test 11: OR( A=      3, B=      2)
 Expected output:      3
 Actual autput:   0003 (decimal:      3)
 Zero:            0
Test 12: BOOTH( A=     30, B=     40)
 Expected output:   1200
 Actual autput:   04b0 (decimal:   1200)
 Zero:            0
Test 13:NON RESTORING( A=     80, B=     40)
 Expected output:      2
 Actual autput:   xxxx (decimal:      x)
 Zero:            x
```