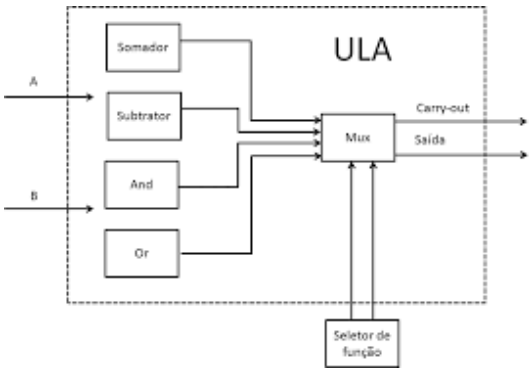
Projeto ULA

Achei uma imagem simplificada de uma ula com 2 entradas de 1 bit 4 operaçoes (AND, OR, Somador, Subtrador) e 2 saidas (Carry out e Saida). Vou usar este formato de base para iniciar o meu projeto da ULA em verilog



Vamos começar pelo MUX

Estou vendo como se faz um MUX pelo livro Elementos da Eletrônica Digital, de Capuano.

Vamos contruir um mux de 2 entradas e 1 seletor possibilitando o uso em flexivel em necessidades maiores.

Assim ficou o meu mux de 2 entradas:

```
module mux2(input i0, i1, selector, output S);
    wire i0CheckOut, i1CheckOut;

    and a0(i0CheckOut, i0, ~selector);
    and a1(i1CheckOut, i1, selector);
    or or0(S, i0CheckOut, i1CheckOut);
endmodule
```

Agora vou preparar um de 4 entradas utilizando 3 mux de 2 entradas para utilzar no nosso projeto de ula simples.
Estou usando como referencia o modelo de ula da pagina 367, capitulo 8.3.3, figura 8.16.

Assim ficou o mux de 4 entradas e 2 seletores:

```
module mux4(input i0, i1, i2, i3, selectorA, selectorB, output s);
    wire mux1Out, mux2Out;

    mux2 mux1(i0, i1, selectorB, mux1Out);
    mux2 mux2(i2, i3, selectorB, mux2Out);
```

```
    mux2 mux3(mux1Out, mux2Out, selectorA, s);
endmodule
```

Agora irei produzir as funcionalidades da ULA, começando pelo somador:

Usarei o Meio Somador apresentado na pagina 211, capitulo 5.4.1, Figura 5.31:

```
module halfAdder(input n1, n2, output result, carry);
  xor x1(result, n1, n2);
  and a1(carry, n1, n2);
endmodule
```

Agora irei fazer um Meio Subtrator, que foi apresentado na pagina 217, capitulo 5.4.4, Figura 5.44. É bastante parecido como halfAdder, exceto pela negação do n1 no and do carry:

```
module halfSubtractor(input n1, n2, output result, carry);
  xor x1(result, n1, n2);
  and a1(carry, ~n1, n2);
endmodule
```

Como o And e o Or já existem, posso agora montar a estrutura geral da ULA:

E o resultado final do metodo ula ficou assim:

```
module ula(input i0, i1, selectorA, selectorB, output result, carry);
  wire adderOut, subOut, andOut, orOut, adderCarryOut, subCarryOut;

  halfAdder ha(i0, i1, adderOut, adderCarryOut);
  halfSubtractor hs(i0, i1, subOut, subCarryOut);
  and a(andOut, i0, i1);
  or o(orOut, i0, i1);

  mux4 muxResult(adderOut, subOut, andOut, orOut, selectorA, selectorB, result);
  mux4 muxCarry(adderCarryOut, subCarryOut, 1'b0, 1'b0, selectorA, selectorB, car
endmodule
```

Agora para entender como trabalhar com multiplos bits, usaremos o mesmo circuito porém modificando-o para trabalhar com 4 bits em cada entrada.

Primeiro criei um modulo do mux para trabalhar com 4bits, para isso apenas recebi os mesmo inputs porem com 4bits e fiz um for com generator para cada bit usar a funçao anterior do mux, ficando assim:

```
module mux2FourBits(input wire [3:0] i0, input wire [3:0] i1, input selector, out
    genvar i;

     generate
       for (i = 0; i < 4; i = i + 1) begin : mux_loop
         mux2 m(.i0(i0[i]), .i1(i1[i]), .selector(selector), .s(s[i]));
        end
    endgenerate
endmodule
```

Usando o `mux2FourBits` fiz o `mux4FourBits`. Acrescentei uma melhoria na forma como recebo os bits de seleção, agora ao invés de receber dois inputs, recebo um vetor de bits. O resultado ficou assim:

```
module mux4FourBits(
  input wire [3:0] i0,
  input wire [3:0] i1,
  input wire [3:0] i2,
  input wire [3:0] i3,
  input wire [1:0] selector,
  output wire [3:0] s);
      wire [3:0] mux1Out;
      wire [3:0] mux2Out;

      mux2FourBits mux1(i0, i1, selector[0], mux1Out);
      mux2FourBits mux2(i2, i3, selector[0], mux2Out);
      mux2FourBits mux3(mux1Out, mux2Out, selector[1], s);
endmodule
```

Agora vou modificar as minhas funcionalidades para lidarem com 4 bits.
Somador:

Usarei o Ripple Carry Adder da atividade 1 como o meu somador de 4 bits.

Na atividade entreguei o RCA assim:

```
module half_adder(
 input wire a, b,
 output wire sum, carry);

 assign sum=a^b;
 assign carry=a&b;
```

```
endmodule

module full_adder(
  input wire a, b, initCarry,
  output wire sum, carry);

  logic resultHa1, carry1, carry2;

  half_adder ha1(a, b, resultHa1, carry1);
  half_adder ha2(resultHa1, initCarry, sum, carry2);

  assign carry=carry1|carry2;

endmodule


module rca(
  input wire a1, b1, a2, b2, a3, b3, a4, b4, initCarry,
  output wire carry1, carry2, carry3, carry4, s1, s2, s3, s4
);
  full_adder fa1(a1, b1, initCarry, s1, carry1);
  full_adder fa2(a2, b2, carry1, s2, carry2);
  full_adder fa3(a3, b3, carry2, s3, carry3);
  full_adder fa4(a4, b4, carry3, s4, carry4);

endmodule
```

Irei melhora-lo com os conhecimentos adquiridos de Verilog.
Mudanças:

1. Transformar as entradas em saidas em vetores de 4 bits
2. Remover carrys 1-3 do output

Resultado:

```
module rca(
  input wire [3:0] a, b,
  input initCarry,
  output wire carryOut,
  output wire [3:0] s
);
  wire carry1, carry2, carry3;
  full_adder fa1(a[0], b[0], initCarry, s[0], carry1);
  full_adder fa2(a[1], b[1], carry1, s[1], carry2);
```

```
  full_adder fa3(a[2], b[2], carry2, s[2], carry3);
  full_adder fa4(a[3], b[3], carry3, s[3], carryOut);

endmodule
```

Este somador vai servir o proposito desta ula simples, mas já encontrei um somador subtrator completo, ou seja, faz as duas operações em um unico circuito.

Subtração:

Agora irei implementar um `full subtractor` (Pagina 222, capitulo 5.4.6, Figura 5.54) utilizando o `half subtractor` desenvolvido anteriormente e utilizando o `full subtractor` criarei o **Ripple Borrow Subtractor**. Resultado:

```
module fullSubtractor(
  input n1, n2, borrowIn,
  output result, borrowOut
);
  wire s0, borrow0, borrow1;

  halfSubtractor hs1(n1, n2, s0, borrow0);
  halfSubtractor hs2(s0, borrowIn, result, borrow1);

  or o(borrowOut, borrow0, borrow1);
endmodule

module rbs(
  input wire [3:0] a, b,
  input initBarrow,
  output wire barrowOut,
  output wire [3:0] s
);
  wire borrow1, borrow2, borrow3;

  fullSubtractor fs1(a[0], b[0], initBarrow, s[0], borrow1);
  fullSubtractor fs2(a[1], b[1], borrow1, s[1], borrow2);
  fullSubtractor fs3(a[2], b[2], borrow2, s[2], borrow3);
  fullSubtractor fs4(a[3], b[3], borrow3, s[3], barrowOut);

endmodule
```

Agora vou montar a ULA de 4 bits completa:

```
module ula(
  input wire [3:0] i0, i1,
  input wire [1:0] selector,
  input carryIn,
  output wire [3:0] result,
  output carry
);
  wire [3:0] adderOut, subOut, andOut, orOut;
  wire adderCarryOut, subCarryOut;

  rca rca1(i0, i1, carryIn, adderCarryOut, adderOut);
  rbs rbs1(i0, i1, carryIn, subCarryOut, subOut);
  and a(andOut, i0, i1);
  or o(orOut, i0, i1);

  mux4FourBits muxResult(adderOut, subOut, andOut, orOut, selector, result);
  mux4 muxCarry(adderCarryOut, subCarryOut, 1'b0, 1'b0, selector, carry);
endmodule
```

Agora montarei o esboço da ula completa com 12 operaçoes de 8 bits:

Usarei 5 Flags:
Carry-Out (C): Para repassar a informação de que houve emprestimo (borrow) ou um “vai-um” (carry)
Zero (Z): Para indicar que a operação retornou 0
Sign (S): Para indicar o sinal resultante
Overflow (V): Para indicar que houve um overflow
Invalid Operation (INV_OP): Operação inválida, caso a entrada seletora não corresponda a nenhuma operação definida

// IMAGEM DO FIGMA

Começando pelos MUXs:

Enquanto monto os Mux de 8 e 16 canais tive a ideia de usar testes automaticos para verificar se o circuito está correto, aqui está o código para testar o mux de 8 entradas:

```
module mux8EightBits_tb;
  reg [7:0] i0, i1, i2, i3, i4, i5, i6, i7;
  reg [2:0] selector;
  wire [7:0] s;

  mux8EightBits uut (
```

```
    .i0(i0),
    .i1(i1),
    .i2(i2),
    .i3(i3),
    .i4(i4),
    .i5(i5),
    .i6(i6),
    .i7(i7),
    .selector(selector),
    .s(s)
  );

  initial begin
    $display("selector |   s    | Expected  | Result");
    $display("-------------------------------------");

    i0 = 8'b00000001;
    i1 = 8'b00000010;
    i2 = 8'b00000100;
    i3 = 8'b00001000;
    i4 = 8'b00010000;
    i5 = 8'b00100000;
    i6 = 8'b01000000;
    i7 = 8'b10000000;

    test_case(3'b000, 8'b00000001);
    test_case(3'b001, 8'b00000010);
    test_case(3'b010, 8'b00000100);
    test_case(3'b011, 8'b00001000);
    test_case(3'b100, 8'b00010000);
    test_case(3'b101, 8'b00100000);
    test_case(3'b110, 8'b01000000);
    test_case(3'b111, 8'b10000000);

    $finish;
  end

  task test_case(input reg [2:0] selector_val, input reg [7:0] expected);
    begin
      selector = selector_val;
      #1;
      $display("%b        | %b | %b  | %s", selector, s, expected, (s === expecte
    end
  endtask
endmodule
```

Teste para mux16:

```
module mux16EightBits_tb;
  reg [7:0] i0, i1, i2, i3, i4, i5, i6, i7;
  reg [7:0] i8, i9, i10, i11, i12, i13, i14, i15;
  reg [3:0] selector;
  wire [7:0] s;

  mux16EightBits uut (
    .i0(i0), .i1(i1), .i2(i2), .i3(i3),
    .i4(i4), .i5(i5), .i6(i6), .i7(i7),
    .i8(i8), .i9(i9), .i10(i10), .i11(i11),
    .i12(i12), .i13(i13), .i14(i14), .i15(i15),
    .selector(selector),
    .s(s)
  );

  initial begin
    $display("selector |   s    | Expected  | Result");
    $display("-------------------------------------");

    i0 = 8'b00000001; i1 = 8'b00000010; i2 = 8'b00000100; i3 = 8'b00001000;
    i4 = 8'b00010000; i5 = 8'b00100000; i6 = 8'b01000000; i7 = 8'b10000000;
    i8 = 8'b00000011; i9 = 8'b00000110; i10 = 8'b00001100; i11 = 8'b00011000;
    i12 = 8'b00110000; i13 = 8'b01100000; i14 = 8'b11000000; i15 = 8'b11111111;

    test_case(4'b0000, 8'b00000001);
    test_case(4'b0001, 8'b00000010);
    test_case(4'b0010, 8'b00000100);
    test_case(4'b0011, 8'b00001000);
    test_case(4'b0100, 8'b00010000);
    test_case(4'b0101, 8'b00100000);
    test_case(4'b0110, 8'b01000000);
    test_case(4'b0111, 8'b10000000);
    test_case(4'b1000, 8'b00000011);
    test_case(4'b1001, 8'b00000110);
    test_case(4'b1010, 8'b00001100);
    test_case(4'b1011, 8'b00011000);
    test_case(4'b1100, 8'b00110000);
    test_case(4'b1101, 8'b01100000);
    test_case(4'b1110, 8'b11000000);
    test_case(4'b1111, 8'b11111111);

    $finish;
  end
```

```
  task test_case(input reg [3:0] selector_val, input reg [7:0] expected);
    begin
      selector = selector_val;
      #1;
      $display("%b        | %b | %b  | %s", selector, s, expected, (s === expecte
    end
  endtask
endmodule
```

O arquivo mux.sv completo ficou assim:

```
module mux2(input i0, i1, selector, output S);
  wire i0CheckOut, i1CheckOut;

  and a0(i0CheckOut, i0, ~selector);
  and a1(i1CheckOut, i1, selector);
  or or0(S, i0CheckOut, i1CheckOut);
endmodule

module mux2EightBits(input wire [7:0] i0, i1 , input selector, output wire [7:0]
  genvar i;

  generate
    for (i = 0; i < 8; i = i + 1) begin : mux_loop
      mux2 m(.i0(i0[i]), .i1(i1[i]), .selector(selector), .S(s[i]));
    end
  endgenerate
endmodule

module mux4EightBits(
  input wire [7:0] i0, i1, i2, i3,
  input wire [1:0] selector,
  output wire [7:0] s
);
  wire [7:0] mux1Out;
  wire [7:0] mux2Out;

  mux2EightBits mux1(i0, i1, selector[0], mux1Out);
  mux2EightBits mux2(i2, i3, selector[0], mux2Out);
  mux2EightBits mux3(mux1Out, mux2Out, selector[1], s);
endmodule

module mux8EightBits(
```

```
  input wire [7:0] i0, i1, i2, i3, i4, i5, i6, i7,
  input wire [2:0] selector,
  output wire [7:0] s
);
  wire [7:0] mux1Out;
  wire [7:0] mux2Out;

  mux4EightBits mux1(i0, i1, i2, i3, selector[1:0], mux1Out);
  mux4EightBits mux2(i4, i5, i6, i7, selector[1:0], mux2Out);
  mux2EightBits mux3(mux1Out, mux2Out, selector[2], s);
endmodule

module mux16EightBits(
  input wire [7:0] i0, i1, i2, i3, i4, i5, i6, i7, i8, i9, i10, i11, i12, i13, i1
  input wire [3:0] selector,
  output wire [7:0] s
);
  wire [7:0] mux1Out;
  wire [7:0] mux2Out;

  mux8EightBits mux1(i0, i1, i2, i3,i4, i5, i6, i7, selector[2:0], mux1Out);
  mux8EightBits mux2(i8, i9, i10, i11, i12, i13, i14, i15, selector[2:0], mux2Out
  mux2EightBits mux3(mux1Out, mux2Out, selector[3], s);
endmodule
```

Agora vou implementar as funcionalidades. Começando pelo Somador:

O somador será igual o 4-Bit Ripple Carry Adder apresentando anteriormente, mas com mais 4 full adder em serie.

O somador ficou assim:

```
module half_adder(
    input wire a, b,
    output wire sum, carry
);
    assign sum = a ^ b;
    assign carry = a & b;
endmodule

module full_adder(
    input wire a, b, initCarry,
    output wire sum, carry
);
```

```
    wire resultHa1, carry1, carry2;

    half_adder ha1(a, b, resultHa1, carry1);
    half_adder ha2(resultHa1, initCarry, sum, carry2);

    assign carry = carry1 | carry2;
endmodule

module eightBitRCA(
    input wire [7:0] a, b,
    input initCarry,
    output wire carryOut,
    output wire [7:0] s
);
    wire [6:0] carry; // Definimos 7 bits para os sinais de carry intermediários

    // Primeira instância do full_adder (bit 0), usando initCarry diretamente
    full_adder fa0 (
        .a(a[0]),
        .b(b[0]),
        .initCarry(initCarry),
        .sum(s[0]),
        .carry(carry[0])
    );

    // Instâncias restantes usando generate
    genvar i;
    generate
        for (i = 1; i < 7; i = i + 1) begin : full_adder_loop
            full_adder fa (
                .a(a[i]),
                .b(b[i]),
                .initCarry(carry[i-1]),
                .sum(s[i]),
                .carry(carry[i])
            );
        end
    endgenerate

    // Última instância do full_adder (bit 7), conectando carryOut
    full_adder fa7 (
        .a(a[7]),
        .b(b[7]),
        .initCarry(carry[6]),
        .sum(s[7]),
```

```
        .carry(carryOut)
    );
endmodule
```

O testbench do somador ficou assim:

```
`timescale 1ns / 1ps

module eightBitRCA_tb;
    reg [7:0] a, b;
    reg initCarry;
    wire carryOut;
    wire [7:0] s;

    eightBitRCA uut (
        .a(a),
        .b(b),
        .initCarry(initCarry),
        .carryOut(carryOut),
        .s(s)
    );

    initial begin
        $display(" a        | b        | initCarry |  s        | carryOut | Expec
        $display("----------------------------------------------------------------

        // Test cases
        test_case(8'b00000001, 8'b00000001, 0, 8'b00000010, 0);   // 1 + 1 = 2
        test_case(8'b00000101, 8'b00000011, 0, 8'b00001000, 0);   // 5 + 3 = 8
        test_case(8'b11111111, 8'b00000001, 0, 8'b00000000, 1);   // 255 + 1 = 0
        test_case(8'b11111010, 8'b00000110, 0, 8'b00000000, 1);   // -6 + 6 = 0
        test_case(8'b11111010, 8'b11111100, 0, 8'b11110110, 1);   // -6 + -4 = -1
        test_case(8'b00000011, 8'b00000011, 1, 8'b00000111, 0);   // 3 + 3 + init

        $finish;
    end

    task test_case(input reg [7:0] a_val, input reg [7:0] b_val, input reg initCa
        begin
            a = a_val;
            b = b_val;
            initCarry = initCarry_val;
            #1;
            $display("%b | %b | %b         | %b | %b        | %b       | %b
```

```
                a, b, initCarry, s, carryOut, expected_s, expected_carry,
                (s === expected_s && carryOut === expected_carry) ? "PASS" : "FAI
        end
    endtask
endmodule
```

Percebi um problema em parte da logica. Eu estava trabalhando com complemento de 2 mas sem entender ele direito, agora que entendi vou abordar alguamas coisas que irão mudar na ULA:

Entradas:

- **A** e **B**: Seguirei utilizando as entradas A e B com 8 bits, porém elas deverão ser interpretadas como complemento de 2, o bit mais significativo **(bit[7])** representará o sinal do número. Então a nossa entrada maxima para as entradas de operaçoes aritiméticas serão 127 a -127.
- **B**: B será invetido e o carryIn interno receberá o valor 1 quando a operação for subtração.
- Ambas as entradas serão interpretadas como binários sem sinal quando a operação for lógica.

Flags:

Explicando melhor como as flags funcionaram:

Flags na ULA (sem `CarryOut`)

Decidi tirar a flag **Carry Out (CF)**, porque em uma ULA que trabalha só com números em complemento de dois, o `CarryOut` não tem utilidade. No complemento de dois, o **Overflow (OF)** é o que realmente indica se o resultado passou dos limites de números que cabem em 16 bits com sinal. A flag `CarryOut` geralmente só é usada em operações sem sinal, onde ele mostra se houve um “vai-um” além do limite. Como aqui tudo é com sinal, manter o `CarryOut` não faz diferença e pode até confundir.

Então, vamos às flags que ficaram:

- **Overflow (overflowFlag)**: Essa flag vai indicar se o resultado de uma operação (soma ou subtração) estourou o intervalo que a ULA consegue representar em 16 bits com sinal, que vai de -32.768 a +32.767. O `overflowFlag` fica ativado quando:
    - Ambos os números têm o mesmo sinal, mas o resultado dá um sinal oposto. Por

exemplo, somar dois números positivos que resultam em um número negativo ou somar dois negativos que resultam em um positivo.
    - Em resumo, o `overflowFlag` mostra quando o número final não cabe no espaço de 16 bits com sinal.

- **Zero (zeroFlag)**: A `zeroFlag` é bem simples: ela fica ativada se o resultado da operação for zero, ou seja, se todos os bits do resultado forem `0`. Isso vale tanto para operações aritméticas quanto para lógicas, então ela é bem útil para verificar se deu zero no final.

- **Sign (signFlag)**: A `signFlag` indica se o resultado final é positivo ou negativo. Ela olha para o bit mais significativo (agora o **bit 15** em 16 bits), que representa o sinal no complemento de dois. Se o bit 15 for `1`, o número é negativo; se for `0`, é positivo. Essa flag é importante em operações com sinal, mas não vai ser usada em operações lógicas, onde o sinal não importa.

Resumo do Motivo para Remover o `carryOut`

O `carryOut` não tem utilidade aqui porque ele não indica overflow em operações com sinal. Em vez disso, a `overflowFlag` é quem avisa quando o número estourou o intervalo de 16 bits. Então, tirar o `carryOut` deixa o design da ULA mais limpo e focado no que realmente importa para números em complemento de dois.

Agora irei reconstruir o Somador com saidas de flag, e resposta de 16 bits.

Renomearei o nome do carry de entrada para `carryFlagIn` e os carrys internos de suporte se chamarão `carryTransport` (carry de transporte), também renomearei as váriaveis `s` e `sum` para `result` para melhor clareza. E o `carryOut` agora não será mais utilizada no output e será movida para o `carryTransport` que é um vetor,

Como a entrada é menor que saida, devemos propagar o bit de sinal para os os outros 8 bits de entrada aceita, faremos assim:

```
// Extensão de sinal de A e B para 16 bits

assign aExt = { {8{a[7]}}, a }; // Propaga a[7] nos 8 bits significativos
assign bExt = { {8{b[7]}}, b }; // Propaga b[7] nos 8 bits significativos
```

Essa tabela explica os diferentes casos suportados pelo somador e qual será o resultado e as flags:

Caso	A (8 bits)	B (8 bits)	Operação	Resultado (16 bits)	overflowFlag	zeroFlag	signFlag
Soma Positiva Sem Overflow	0000 1010 (10)	0000 0011 (3)	A + B	0000 0000 0000 1101 (13)	0	0	0
Soma de Positivos (Expansão)	0111 1111 (127)	0000 0001 (1)	A + B	0000 0000 1000 0000 (128)	0	0	0
Soma de Negativos Sem Overflow	1111 0001 (-15)	1110 1100 (-20)	A + B	1111 1111 1101 1101 (-35)	0	0	1
Soma Positivo e Negativo	0000 0101 (5)	1111 1101 (-3)	A + B	0000 0000 0000 0010 (2)	0	0	0
Subtração com Resultado Zero	0000 0101 (5)	0000 0101 (5)	A - B	0000 0000 0000 0000 (0)	0	1	0
Soma Negativos com Overflow	1000 0000 (-128)	1000 0000 (-128)	A + B	1111 1111 0000 0000 (-256)	0	0	1
Soma de	0000	0000		0000 0000			

Zeros	0000 (0)	0000 (0)	A + B	0000 0000 (0)	0	1	0
Soma de -1 e 1 com Resultado Zero	1111 1111 (-1)	0000 0001 (1)	A + B	0000 0000 0000 0000 (0)	0	1	0
Soma de Dois Negativos	1111 1111 (-1)	1111 1111 (-1)	A + B	1111 1111 1111 1110 (-2)	0	0	1
Soma de Máximo Positivo	0111 1111 (127)	0111 1111 (127)	A + B	0000 0000 1111 1110 (254)	0	0	0
Soma de Máximo e Mínimo	1000 0000 (-128)	1000 0001 (-127)	A + B	1111 1111 0000 0001 (-1)	0	0	1

A primeira versão ficou assim:

```
module eightBitRCA(
    input wire [7:0] a, b,
    input carryFlagIn, operation
    output wire [15:0] result,
    output wire overflowFlag, signFlag, zeroFlag
);
    wire [15:0] aExt, bExt;
    wire [15:0] carryTransport; // Definimos 14 bits para os sinais de carry inte

      // Extensão de sinal de A e B para 16 bits
    assign aExt = { {8{a[7]}}, a };  // Propaga a[7] nos 8 bits significativos
    assign bExt = { {8{b[7]}}, b };  // Propaga b[7] nos 8 bits significativos


    // Primeira instância do full_adder (bit 0), usando carryFlagIn diretamente
    full_adder fa0 (
```

```
        .a(aExt[0]),
        .b(bExt[0]),
        .carryFlagIn(carryFlagIn),
        .result(result[0]),
        .carryFlagOut(carryTransport[0])
    );

    // Instâncias restantes usando generate
    genvar i;
    generate
      for (i = 1; i < 16; i = i + 1) begin : full_adder_loop
            full_adder fa (
                .a(aExt[i]),
                .b(bExt[i]),
                .carryFlagIn(carryTransport[i-1]),
                .result(result[i]),
                .carryFlagOut(carryTransport[i])
            );
        end
    endgenerate

    assign overflowFlag = (aExt[15] & bExt[15] & ~result[15]) | (~aExt[15] & ~bEx

    assign zeroFlag = ~(result[0] | result[1] | result[2] | result[3] |
                  result[4] | result[5] | result[6] | result[7] |
                  result[8] | result[9] | result[10] | result[11] |
                  result[12] | result[13] | result[14] | result[15]);

    assign signFlag = result[15];

endmodule
```

mas ainda tem um problema, a operação de subtração falha porque deveria ser feito uma inversão do B e passar 1 para o carryFlagIn. Para finalizar o circuito de soma, irei remover o input `carryFlagIn` , adicionar um input `operation` , esse valor poderá vir do `selector[0]` da ula já que o seletor de subtração é `0001` . Dentro do somador, no lugar do carryFlagIn irei utilizar input `operation` (0 ou 1) e no trecho que faz a extensão do valor `b` irei inverter-lo caso a operação seja 1. Ficando assim:

```
assign bExt = { {8{b[7] ^ operation}}, (b ^ {8{operation}}) }; // Inverte `b` par
```

Analisando o circuito pronto abaixo (1) notei que não é possível acontecer um overflow já que

a soma e subtração de 2 numeros no intervalo de 127 a - 127 não passa dos limites de um número com 16 bits: 32768 a -32768. Então irei remover esta flag. Resultando final (2) e testbench (3)

1.

```
module eightBitRCA(
    input wire [7:0] a, b,
    input operation,
    output wire [15:0] result,
    output wire overflowFlag, signFlag, zeroFlag
);
    wire [15:0] aExt, bExt;
    wire [15:0] carryTransport; // Definimos 16 bits para as flags de carry de tr

     // Extensão de sinal de A e B para 16 bits
    assign aExt = { {8{a[7]}}, a };  // Propaga a[7] nos 8 bits significativos
    assign bExt = { {8{b[7] ^ operation}}, (b ^ {8{operation}}) }; // Inverte `b`

    // Primeira instância do full_adder (bit 0), usando carryFlagIn diretamente
    full_adder fa0 (
        .a(aExt[0]),
        .b(bExt[0]),
        .carryFlagIn(operation),
        .result(result[0]),
        .carryFlagOut(carryTransport[0])
    );

    // Instâncias restantes usando generate
    genvar i;
    generate
      for (i = 1; i < 16; i = i + 1) begin : full_adder_loop
            full_adder fa (
                .a(aExt[i]),
                .b(bExt[i]),
                .carryFlagIn(carryTransport[i-1]),
                .result(result[i]),
                .carryFlagOut(carryTransport[i])
            );
        end
    endgenerate

    assign overflowFlag = (aExt[15] & bExt[15] & ~result[15]) | (~aExt[15] & ~bEx
```

```
    assign zeroFlag = ~(result[0] | result[1] | result[2] | result[3] |
                  result[4] | result[5] | result[6] | result[7] |
                  result[8] | result[9] | result[10] | result[11] |
                  result[12] | result[13] | result[14] | result[15]);

    assign signFlag = result[15];
endmodule
```

2.

```
module eightBitRCA(
input wire [7:0] a, b,
input operation,
output wire [15:0] result,
output wire signFlag, zeroFlag
);
    wire [15:0] aExt, bExt;
    wire [15:0] carryTransport; // Definimos 16 bits para as flags de carry de tr

 // Extensão de sinal de A e B para 16 bits
assign aExt = { {8{a[7]}}, a };  // Propaga a[7] nos 8 bits significativos
assign bExt = { {8{b[7] ^ operation}}, (b ^ {8{operation}}) }; // Inverte `b` par

// Primeira instância do full_adder (bit 0), usando carryFlagIn diretamente
full_adder fa0 (
    .a(aExt[0]),
    .b(bExt[0]),
    .carryFlagIn(operation),
    .result(result[0]),
    .carryFlagOut(carryTransport[0])
);

// Instâncias restantes usando generate
genvar i;
generate
  for (i = 1; i < 16; i = i + 1) begin : full_adder_loop
        full_adder fa (
            .a(aExt[i]),
            .b(bExt[i]),
            .carryFlagIn(carryTransport[i-1]),
            .result(result[i]),
            .carryFlagOut(carryTransport[i])
        );
    end
```

```
endgenerate

  assign zeroFlag = ~(result[0] | result[1] | result[2] | result[3] |
              result[4] | result[5] | result[6] | result[7] |
              result[8] | result[9] | result[10] | result[11] |
              result[12] | result[13] | result[14] | result[15]);

  assign signFlag = result[15];


endmodule
```

3.

```
module eightBitRCA_tb;
    reg [7:0] inputA;
    reg [7:0] inputB;
    reg operation;
    wire [15:0] result;
    wire zeroFlag;
    wire signFlag;

    eightBitRCA uut (
        .a(inputA),
        .b(inputB),
        .operation(operation),
        .result(result),
        .zeroFlag(zeroFlag),
        .signFlag(signFlag)
    );

    initial begin
        $display("Op   |   A           |   B           |  Result           | Expe
        $display("------------------------------------------------------------------

        testCase(8'b00001010, 8'b00000011, 1'b0, 16'b0000000000001101, 1'b0, 1'b0
        testCase(8'b01111111, 8'b00000001, 1'b0, 16'b0000000010000000, 1'b0, 1'b0
        testCase(8'b11110001, 8'b11101100, 1'b0, 16'b1111111111011101, 1'b0, 1'b1
        testCase(8'b00000101, 8'b11111101, 1'b0, 16'b0000000000000010, 1'b0, 1'b0
        testCase(8'b00000101, 8'b00000101, 1'b1, 16'b0000000000000000, 1'b1, 1'b0
        testCase(8'b10000000, 8'b10000000, 1'b0, 16'b1111111100000000, 1'b0, 1'b1
        testCase(8'b00000000, 8'b00000000, 1'b0, 16'b0000000000000000, 1'b1, 1'b0
        testCase(8'b11111111, 8'b00000001, 1'b0, 16'b0000000000000000, 1'b1, 1'b0
        testCase(8'b11111111, 8'b11111111, 1'b0, 16'b1111111111111110, 1'b0, 1'b1
```

```
        testCase(8'b01111111, 8'b01111111, 1'b0, 16'b0000000011111110, 1'b0, 1'b0
        testCase(8'b10000000, 8'b10000001, 1'b0, 16'b1111111100000001, 1'b0, 1'b1

        $finish;
    end

    task testCase(
        input reg [7:0] aVal,
        input reg [7:0] bVal,
        input reg op,
        input reg [15:0] expectedResult,
        input reg expectedZero,
        input reg expectedSign
    );
        begin
            inputA = aVal;
            inputB = bVal;
            operation = op;
            #1;

            $write("%s  | %b (%0d) | %b (%0d) | %b | %b |  %b  |  %b  |   %b   |
                   op ? "SUB" : "SUM", inputA, inputA, inputB, inputB, result, ex
                   zeroFlag, signFlag, expectedZero, expectedSign);

            if (result === expectedResult && zeroFlag === expectedZero && signFla
                $display("PASS");
            else
                $display("FAIL");
        end
    endtask
endmodule
```

Recebi instruções do professor o circuito deve ser feito no estilo estruturado. Farei as próximas partes do código usando este estilo e depois volto refatorando as outras partes que eu já havia construido.

Multiplicador:
O algortimo que escolhi é o Wallace Tree, ele é rápido e fácil de entender em comparações a outros que vi como o Algortimo Booth e Matrix multiplication algorithm.

Primeiro preciso gerar os produtos parciais, que é feito multiplicando todos os bits de A vezes todos os bits de B. Farei isso usando portas AND. Crie um modulo que gera os produtos parciais, resultado:

```
 module getPartialProducts(
  input wire [6:0] a, b,
  output wire [63:0] partialProduct
);
  // Variáveis para os produtos parciais
  // 8 bits x 8 bits = 64 produtos parciais
  and ppA0(partialProduct[0], a[0], b[0]);
  and ppA1(partialProduct[1], a[0], b[1]);
  and ppA2(partialProduct[2], a[0], b[2]);
  and ppA3(partialProduct[3], a[0], b[3]);
  and ppA4(partialProduct[4], a[0], b[4]);
  and ppA5(partialProduct[5], a[0], b[5]);
  and ppA6(partialProduct[6], a[0], b[6]);
  and ppA7(partialProduct[7], 1'b0, 1'b0);
  and ppA8(partialProduct[8], a[1], b[0]);
  and ppA9(partialProduct[9], a[1], b[1]);
  and ppA10(partialProduct[10], a[1], b[2]);
  and ppA11(partialProduct[11], a[1], b[3]);
  and ppA12(partialProduct[12], a[1], b[4]);
  and ppA13(partialProduct[13], a[1], b[5]);
  and ppA14(partialProduct[14], a[1], b[6]);
  and ppA15(partialProduct[15], 1'b0, 1'b0);
  and ppA16(partialProduct[16], a[2], b[0]);
  and ppA17(partialProduct[17], a[2], b[1]);
  and ppA18(partialProduct[18], a[2], b[2]);
  and ppA19(partialProduct[19], a[2], b[3]);
  and ppA20(partialProduct[20], a[2], b[4]);
  and ppA21(partialProduct[21], a[2], b[5]);
  and ppA22(partialProduct[22], a[2], b[6]);
  and ppA23(partialProduct[23], 1'b0, 1'b0);
  and ppA24(partialProduct[24], a[3], b[0]);
  and ppA25(partialProduct[25], a[3], b[1]);
  and ppA26(partialProduct[26], a[3], b[2]);
  and ppA27(partialProduct[27], a[3], b[3]);
  and ppA28(partialProduct[28], a[3], b[4]);
  and ppA29(partialProduct[29], a[3], b[5]);
  and ppA30(partialProduct[30], a[3], b[6]);
  and ppA31(partialProduct[31], 1'b0, 1'b0);
  and ppA32(partialProduct[32], a[4], b[0]);
  and ppA33(partialProduct[33], a[4], b[1]);
  and ppA34(partialProduct[34], a[4], b[2]);
  and ppA35(partialProduct[35], a[4], b[3]);
  and ppA36(partialProduct[36], a[4], b[4]);
  and ppA37(partialProduct[37], a[4], b[5]);
  and ppA38(partialProduct[38], a[4], b[6]);
```

```
  and ppA39(partialProduct[39], 1'b0, 1'b0);
  and ppA40(partialProduct[40], a[5], b[0]);
  and ppA41(partialProduct[41], a[5], b[1]);
  and ppA42(partialProduct[42], a[5], b[2]);
  and ppA43(partialProduct[43], a[5], b[3]);
  and ppA44(partialProduct[44], a[5], b[4]);
  and ppA45(partialProduct[45], a[5], b[5]);
  and ppA46(partialProduct[46], a[5], b[6]);
  and ppA47(partialProduct[47], 1'b0, 1'b0);
  and ppA48(partialProduct[48], a[6], b[0]);
  and ppA49(partialProduct[49], a[6], b[1]);
  and ppA50(partialProduct[50], a[6], b[2]);
  and ppA51(partialProduct[51], a[6], b[3]);
  and ppA52(partialProduct[52], a[6], b[4]);
  and ppA53(partialProduct[53], a[6], b[5]);
  and ppA54(partialProduct[54], a[6], b[6]);
  and ppA55(partialProduct[55], 1'b0, 1'b0);
  and ppA56(partialProduct[56], 1'b0, b[0]);
  and ppA57(partialProduct[57], 1'b0, b[1]);
  and ppA58(partialProduct[58], 1'b0, b[2]);
  and ppA59(partialProduct[59], 1'b0, b[3]);
  and ppA60(partialProduct[60], 1'b0, b[4]);
  and ppA61(partialProduct[61], 1'b0, b[5]);
  and ppA62(partialProduct[62], 1'b0, b[6]);
  and ppA63(partialProduct[63], 1'b0, 1'b0);
endmodule
```

Agora vou reescrever o `half_adder` e `full_adder` para o estilo estrutural porque ele é usado no algortimo de multiplicação (Wallace tree):

```
module half_adder(
    input wire a, b,
    output wire result, carryFlagOut
);
  xor xorResult(result, a, b);
  and andCarry(carryFlagOut, a, b);
endmodule

module full_adder(
    input wire a, b, carryFlagIn,
    output wire result, carryFlagOut
);
    wire resultHa1;
    wire [1:0] carryTransport;
```

```
    half_adder ha1(a, b, resultHa1, carryTransport[0]);
    half_adder ha2(resultHa1, carryFlagIn, result, carryTransport[1]);

    or orCarryFlagOut(carryFlagOut,  carryTransport[0] , carryTransport[1]);
endmodule
```

- **Peso 0**:
    - Produto parcial: `p00`
    - Bits usados: `partialProduct[0]`
- **Peso 1**:
    - Produtos parciais: `p10` , `p01`
    - Bits usados: `partialProduct[1]` , `partialProduct[8]`
- **Peso 2**:
    - Produtos parciais: `p20` , `p11` , `p02`
    - Bits usados: `partialProduct[2]` , `partialProduct[9]` , `partialProduct[16]`
- **Peso 3**:
    - Produtos parciais: `p30` , `p21` , `p12` , `p03`
    - Bits usados: `partialProduct[3]` , `partialProduct[10]` , `partialProduct[17]` , `partialProduct[24]`
- **Peso 4**:
    - Produtos parciais: `p40` , `p31` , `p22` , `p13` , `p04`
    - Bits usados: `partialProduct[4]` , `partialProduct[11]` , `partialProduct[18]` , `partialProduct[25]` , `partialProduct[32]`
- **Peso 5**:
    - Produtos parciais: `p50` , `p41` , `p32` , `p23` , `p14` , `p05`
    - Bits usados: `partialProduct[5]` , `partialProduct[12]` , `partialProduct[19]` , `partialProduct[26]` , `partialProduct[33]` , `partialProduct[40]`
- **Peso 6**:

    - Produtos parciais: `p60`, `p51`, `p42`, `p33`, `p24`, `p15`, `p06`
    - Bits usados: `partialProduct[6]`, `partialProduct[13]`, `partialProduct[20]`, `partialProduct[27]`, `partialProduct[34]`, `partialProduct[41]`, `partialProduct[48]`

- **Peso 7**:

    - Produtos parciais: `p70`, `p61`, `p52`, `p43`, `p34`, `p25`, `p16`, `p07`
    - Bits usados: `partialProduct[7]`, `partialProduct[14]`, `partialProduct[21]`, `partialProduct[28]`, `partialProduct[35]`, `partialProduct[42]`, `partialProduct[49]`, `partialProduct[56]`

- **Peso 8**:

    - Produtos parciais: `p71`, `p62`, `p53`, `p44`, `p35`, `p26`, `p17`
    - Bits usados: `partialProduct[15]`, `partialProduct[22]`, `partialProduct[29]`, `partialProduct[36]`, `partialProduct[43]`, `partialProduct[50]`, `partialProduct[57]`

- **Peso 9**:

    - Produtos parciais: `p72`, `p63`, `p54`, `p45`, `p36`, `p27`
    - Bits usados: `partialProduct[23]`, `partialProduct[30]`, `partialProduct[37]`, `partialProduct[44]`, `partialProduct[51]`, `partialProduct[58]`

- **Peso 10**:

    - Produtos parciais: `p73`, `p64`, `p55`, `p46`, `p37`
    - Bits usados: `partialProduct[31]`, `partialProduct[38]`, `partialProduct[45]`, `partialProduct[52]`, `partialProduct[59]`

- **Peso 11**:

    - Produtos parciais: `p74`, `p65`, `p56`, `p47`
    - Bits usados: `partialProduct[39]`, `partialProduct[46]`, `partialProduct[53]`, `partialProduct[60]`

- **Peso 12**:

    - Produtos parciais: `p75`, `p66`, `p57`
    - Bits usados: `partialProduct[47]`, `partialProduct[54]`, `partialProduct[61]`

- **Peso 13**:
    - Produtos parciais: `p76 , p67`
    - Bits usados: `partialProduct[55] , partialProduct[62]`
- **Peso 14**:
    - Produtos parciais: `p77`
    - Bits usados: `partialProduct[63]`

Estava muito dificil de entender em formato de vetor, gerava muitos erros então refatorei para utilizar 1 tire para cada um dos produtos parciais, o nome está no formato pij. Alias utilize o floxograma deste artigo para fazer o Wallace tree [A Novel In-Memory Wallace Tree Multiplier Architecture using Majority Logic]

O resultado final (finalmente, pq demorou muito), é esse:

```
module getPartialProducts(
  input wire [7:0] a, b,
  output wire p00, p01, p02, p03, p04, p05, p06, p07,
  output wire p10, p11, p12, p13, p14, p15, p16, p17,
  output wire p20, p21, p22, p23, p24, p25, p26, p27,
  output wire p30, p31, p32, p33, p34, p35, p36, p37,
  output wire p40, p41, p42, p43, p44, p45, p46, p47,
  output wire p50, p51, p52, p53, p54, p55, p56, p57,
  output wire p60, p61, p62, p63, p64, p65, p66, p67,
  output wire p70, p71, p72, p73, p74, p75, p76, p77
);

  // Produtos parciais gerados com operações AND
  and (p00, a[0], b[0]);
  and (p01, a[1], b[0]);
  and (p02, a[2], b[0]);
  and (p03, a[3], b[0]);
  and (p04, a[4], b[0]);
  and (p05, a[5], b[0]);
  and (p06, a[6], b[0]);
  assign p07 = 1'b0;

  and (p10, a[0], b[1]);
  and (p11, a[1], b[1]);
  and (p12, a[2], b[1]);
  and (p13, a[3], b[1]);
```

```
and (p14, a[4], b[1]);
and (p15, a[5], b[1]);
and (p16, a[6], b[1]);
assign p17 = 1'b0;

and (p20, a[0], b[2]);
and (p21, a[1], b[2]);
and (p22, a[2], b[2]);
and (p23, a[3], b[2]);
and (p24, a[4], b[2]);
and (p25, a[5], b[2]);
and (p26, a[6], b[2]);
assign p27 = 1'b0;

and (p30, a[0], b[3]);
and (p31, a[1], b[3]);
and (p32, a[2], b[3]);
and (p33, a[3], b[3]);
and (p34, a[4], b[3]);
and (p35, a[5], b[3]);
and (p36, a[6], b[3]);
assign p37 = 1'b0;

and (p40, a[0], b[4]);
and (p41, a[1], b[4]);
and (p42, a[2], b[4]);
and (p43, a[3], b[4]);
and (p44, a[4], b[4]);
and (p45, a[5], b[4]);
and (p46, a[6], b[4]);
assign p47 = 1'b0;

and (p50, a[0], b[5]);
and (p51, a[1], b[5]);
and (p52, a[2], b[5]);
and (p53, a[3], b[5]);
and (p54, a[4], b[5]);
and (p55, a[5], b[5]);
and (p56, a[6], b[5]);
assign p57 = 1'b0;

and (p60, a[0], b[6]);
and (p61, a[1], b[6]);
and (p62, a[2], b[6]);
and (p63, a[3], b[6]);
```

```
  and (p64, a[4], b[6]);
  and (p65, a[5], b[6]);
  and (p66, a[6], b[6]);
  assign p67 = 1'b0;

  assign p70 = 1'b0;
  assign p71 = 1'b0;
  assign p72 = 1'b0;
  assign p73 = 1'b0;
  assign p74 = 1'b0;
  assign p75 = 1'b0;
  assign p76 = 1'b0;
  assign p77 = 1'b0;

endmodule

module wallaceTreeMultiplier(
    input wire [7:0] a, b,
    output wire [15:0] result
);
  // Produtos parciais (pij)
  wire p00, p01, p02, p03, p04, p05, p06, p07;
  wire p10, p11, p12, p13, p14, p15, p16, p17;
  wire p20, p21, p22, p23, p24, p25, p26, p27;
  wire p30, p31, p32, p33, p34, p35, p36, p37;
  wire p40, p41, p42, p43, p44, p45, p46, p47;
  wire p50, p51, p52, p53, p54, p55, p56, p57;
  wire p60, p61, p62, p63, p64, p65, p66, p67;
  wire p70, p71, p72, p73, p74, p75, p76, p77;

  wire c01, c02, c03, c04, c05, c06, c07, c08;
  wire c11, c12, c13, c14, c15, c16, c17, c18;
  wire c21, c22, c23, c24, c25, c26, c27, c28;
  wire c31, c32, c33, c34, c35, c36, c37, c38;
  wire c40, c41, c42, c43, c44, c45, c46, c47;
  wire c48, c49, c50, c51, c52, c53, c54, c55;
  wire c56, c57, c58, c59, c5_10;

  wire s02, s03, s04, s05, s06, s07, s08, s11;
  wire s12, s13, s14, s15, s16, s17, s18, s22;
  wire s23, s24, s25, s26, s27, s28, s31, s32;
  wire s33, s34, s35, s36, s37, s38, s41, s42;
  wire s43, s44, s45, s46, s47, s48, s49, s51;
  wire s52, s53, s54, s55, s56, s57, s58, s59, s5_10;
```

```
// Gerar os produtos parciais
getPartialProducts gPP(
  .a(a), .b(b),
  .p00(result[0]), .p01(p01), .p02(p02), .p03(p03), .p04(p04), .p05(p05), .p06(
  .p10(p10), .p11(p11), .p12(p12), .p13(p13), .p14(p14), .p15(p15), .p16(p16),
  .p20(p20), .p21(p21), .p22(p22), .p23(p23), .p24(p24), .p25(p25), .p26(p26),
  .p30(p30), .p31(p31), .p32(p32), .p33(p33), .p34(p34), .p35(p35), .p36(p36),
  .p40(p40), .p41(p41), .p42(p42), .p43(p43), .p44(p44), .p45(p45), .p46(p46),
  .p50(p50), .p51(p51), .p52(p52), .p53(p53), .p54(p54), .p55(p55), .p56(p56),
  .p60(p60), .p61(p61), .p62(p62), .p63(p63), .p64(p64), .p65(p65), .p66(p66),
  .p70(p70), .p71(p71), .p72(p72), .p73(p73), .p74(p74), .p75(p75), .p76(p76),
);


// Etapa 1 (16 operações)

half_adder ha1(p01, p10, result[1], c01);

full_adder fa1(p02, p11, p20, s02, c02);

full_adder fa2(p03, p12, p21, s03, c03);

full_adder fa3(p04, p13, p22, s04, c04);

full_adder fa4(p05, p14, p23, s05, c05);

full_adder fa5(p06, p15, p24, s06, c06);

full_adder fa6(p07, p16, p25, s07, c07);

half_adder ha2(p17, p26, s08, c08);

half_adder ha3(p31, p40, s11, c11);

full_adder fa7(p32, p41, p50, s12, c12);

full_adder fa8(p33, p42, p51, s13, c13);

full_adder fa9(p34, p43, p52, s14, c14);

full_adder fa10(p35, p44, p53, s15, c15);

full_adder fa11(p36, p45, p54, s16, c16);

full_adder fa12(p37, p46, p55, s17, c17);
```

```
    half_adder ha4(p47, p56, s18, c18);

    // Etapa 2 (16 operações)
    half_adder ha5(s02, c01, result[2], c21);

    full_adder fa13(s03, c02, p30, s22, c22);

    full_adder fa14(s04, c03, s11, s23, c23);

    full_adder fa15(s05, c04, s12, s24, c24);

    full_adder fa16(s06, c05, s13, s25, c25);

    full_adder fa17(s07, c06, s14, s26, c26);

    full_adder fa18(s08, c07, s15, s27, c27);

    full_adder fa19(s16, c08, p27, s28, c28);

    half_adder ha6(p60, c12, s31, c31);

    full_adder fa20(p61, c13, p70, s32, c32);

    full_adder fa21(p62, c14, p71, s33, c33);

    full_adder fa22(p63, c15, p72, s34, c34);

    full_adder fa23(p64, c16, p73, s35, c35);

    full_adder fa24(p65, c17, p74, s36, c36);

    full_adder fa25(p66, c18, p75, s37, c37);

    half_adder ha7(p67, p76, s38, c38);


    // Etapa 3 (10 operações)
    half_adder ha8(s22, c21, result[3], c40);

    half_adder ha9(s23, c22, s41, c41);

    full_adder fa26(s24, c23, c11, s42, c42);

    full_adder fa27(s25, c24, s31, s43, c43);
```

```
full_adder fa28(s26, c25, s32, s44, c44);

full_adder fa29(s27, c26, s33, s45, c45);

full_adder fa30(s28, c27, s34, s46, c46);

full_adder fa31(s35, c28, s17, s47, c47);

half_adder ha10(s36, s18, s48, c48);

half_adder ha11(s37, p57, s49, c49);


// Etapa 4 (11 operações)

half_adder ha12(s41, c40, result[4], c50);

half_adder ha13(s42, c41, s51, c51);

half_adder ha14(s43, c42, s52, c52);

full_adder fa32(s44, c31, c43, s53, c53);

full_adder fa33(s45, c32, c44, s54, c54);

full_adder fa34(s46, c33, c45, s55, c55);

full_adder fa35(s47, c34, c46, s56, c56);

full_adder fa36(s48, c35, c47, s57, c57);

full_adder fa37(s49, c36, c48, s58, c58);

full_adder fa38(s38, c37, c49, s59, c59);

half_adder ha15(p77, c38, s5_10, c5_10);


wire [10:0] resultAdder;
wire [9:0] aS, bC;

assign aS = {s5_10, s59, s58, s57, s56, s55, s54, s53, s52, s51};
assign bC = {c59, c58, c57, c56, c55, c54, c53, c52, c51, c50};
```

```
  tenBitRCA rca_instance (
    .a(aS),
    .b(bC),
    .carryFlagIn(1'b0),
    .result(resultAdder)
  );

  assign result[15:5] = resultAdder;
endmodule
```

- Atribui diretamente `1'b0` aos produtos parciais que utilizam o bit 8 (index 7) do multiplicador e multiplicando já que um AND contendo zero é zero.
- Tentei utilizar o 10-bit parallel-prefix adder (Ladner Fischer) no soma final (o que também demandou muito tempo, acredito que mais de 10h), mas não deu certo, o diagrama mostrava um zero no AND 7 e isso atrapalhava o sucesso de todos as operações seguintes. Também havia o problema de no diagrama mostrar apenas portas AND e NOT, mas uma pesquisa no google mostrou que o Ladner Fischer usa portas OR também. Então desisti de usar ele e criei um RCA de 10bits para utilizar no multiplicador.
- O código acima lida apenas com multiplicação de números positivos, farei ajustes abaixo para lidar com números em complemento de 2.

Sabemos que `A * B = |-A * B| = |-A * -B| = |A * -B|`

Então iremos obter o valor absoluto do multiplicando e multiplicador que estão complemento de 2 no inicio do algoritmo de multiplicação:

Formula: `Valor absoluto = MSB ? not(N) + 1: N`
MSB = Digito mais significativo (sinal)

Ou seja, se for positivo, não faz nada, se for negativo, inverte todos os bits e adiciona 1.

O código do modulo de valor absoluto ficou assim:

```
module abs(input wire [7:0] a, output wire [7:0] s);
  wire [7:0] not1, absValue;

  not n0(not1, a);
  eightBitRCA rBRCA(
    .a(not1),
    .b(8'b00000001), // 1 em binário
    .result(absValue)
  );
```

```
  mux2EightBits mux2(a, absValue, a[7], s);
endmodule
```

Usamos o mux já apresentado neste artigo para fazer a condicional da formula.

Os próximos passos são:

- Obter o sinal do resultado:
    - Se MSB-A e MSB-B forem 0, o resultado será positivo
    - Se MSB-A e MSB-B forem diferentes, o resultado será negativo

      A formula será: `Sinal = a[7] xor b[7]`
- Converter o resultado para complemento de 2 e atribuir a uma variável `wire resultNegative`
    - Pode ser feito de forma análoga ao modulo abs, inverte N e adiciona 1;
- Por fim, utilizamos um mux passando como seletor o sinal, se for 0 retorna `result` se for 1 retorna `resultNegative`

Vamos implementar! Resultado final:

Antes do resultado final do multiplicador estou juntando todas as partes da ula em um projeto unificado para evitar me perder porque já existe muitas partes. No processo irei refatorar métodos em estilo comportamental para estrutural e vou aplicar boas práticas de programação verilog.

Reuni as boas práticas que devo utilizar:

• Inputs e outputs: use snake_case e singular para 1 bit (input wire reset, output wire valid_signal).

• Wires internos: use snake_case e plural para vetores (wire [7:0] data_bus, wire alu_enable).

• Módulos: use snake_case, evite números no nome (module alu_control_unit).

• Instâncias: use snake_case e números para múltiplas instâncias (and_gate and_inst_0).

• Prefira instanciar módulos com conexões nomeadas (AND_gate and_inst (.a(input_a),

.b(input_b), .y(output_signal))).

• Sempre conecte todas as portas de um módulo, mesmo que não utilizadas (MyModule mod_inst (.input1(signal), .output(unused_output))).

Comecei pelo mux.sv:

- padronizei
- aprendi a usar modulos que suportam menos bits para criar componentes suportam mais bits, apliquei isso ao mux-2 de 16 bits utilizando o dois mux-2 de 8 bits, assim não preciso declarar repetidamente todos os mux-2 de 1 bit novamente para os 16 bits. Esse entendimento vai ser útil para diminuir as repetições nos adders.