

Introdução à SystemVerilog e modelagem de Circuitos Combinacionais

Fundamentos de Sistemas Digitais

Este conjunto de slides contempla 3 aulas:

- 1 – Intro a SV** – 3 a 21 (sala de aula, final da aula demonstração de simulação)
- 2 – Combinacionais em SV** – 22 a 42 (sala de aula, exercícios ao final da aula)
- 2 – Simulação de Combinacionais em SV** – 43 a 57 (em laboratório)

HDLs versus Linguagens de Programação

(HDL: Hardware Description Language)

Possuem construções sintaticamente semelhantes:

- Tipos de dados, variáveis, atribuições, instruções if, laços, etc.

Mas com mentalidade e modelo semântico muito diferentes:

- Tudo é executado em **paralelo**, a menos que seja especificado o contrário
- Cada instrução modela um bloco de hardware
- O hardware é inerentemente paralelo

Programas de software são compostos principalmente por **sub-rotinas**:

- Sub-rotinas chamam umas às outras
- Ao entrar em uma sub-rotina, a execução da função chamadora é pausada

Descrições de hardware são compostas principalmente por **módulos**:

- Uma hierarquia de módulos conectados entre si
- Os módulos estão ativos simultaneamente

HDLs são linguagens de programação?

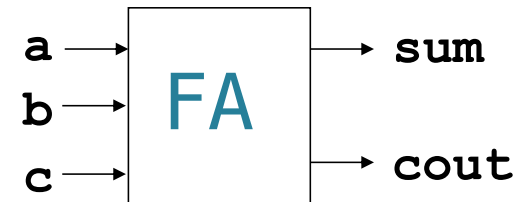
- **Resposta educada** 😊
 - **Não!** É uma linguagem de descrição de hardware!
- **Código é executado em um simulador**
 - Não se enxerga o “compilador” de HDL, não há um “código executável” visível
- **Testbench**
 - Uma descrição em SystemVerilog ou outra linguagem para o teste de um circuito
 - Especificação comportamental do ambiente externo ao projeto (estímulos externos)
 - Interage com o projeto

Referência recomendada

<https://compas.cs.stonybrook.edu/~nhonarmand/courses/sp15/cse502/slides/03-systemverilog.pdf>

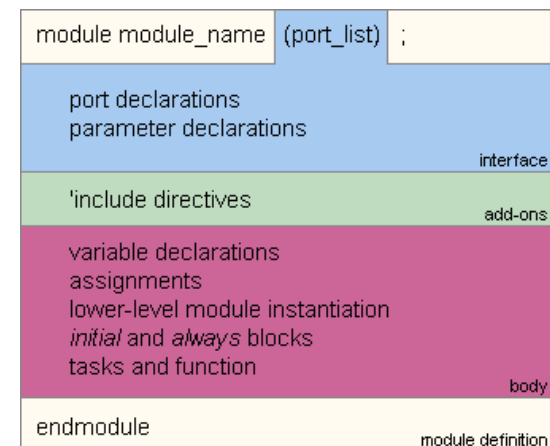
Começando por um exemplo em SystemVerilog: Full Adder

```
module FA (  
    input logic a, b, c,  
    output logic sum, cout  
);  
  
    assign sum = a ^ b ^ c;  
    assign cout = (a & b) | (a & c) | (b & c);  
  
endmodule
```



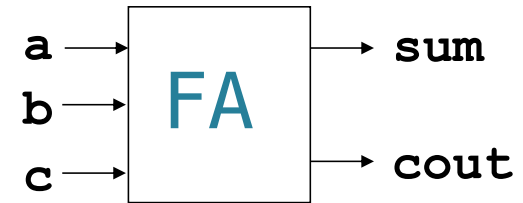
→ module:

- interface externa: especifica a interface entre o módulo e o ambiente
- os sinais de entrada e saída são aqueles que realizam a comunicação do módulo com o ambiente externo
- não contém definição do comportamento ou da estrutura interna



Full Adder (2/4)

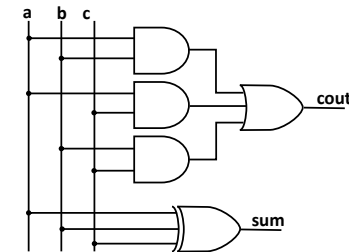
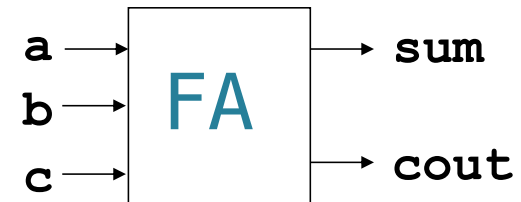
```
module FA (  
    input logic a, b, c,  
    output logic sum, cout  
);  
  
    assign sum = a ^ b ^ c;  
    assign cout = (a & b) | (a & c) | (b & c);  
  
endmodule
```



→ **logic**: tipo utilizado para níveis lógicos: 0, 1, X (desconhecido), Z (alta impedância)

Full Adder (3/4)

```
module FA (  
    input logic a, b, c,  
    output logic sum, cout  
);  
  
    assign sum = a ^ b ^ c;  
    assign cout = (a & b) | (a & c) | (b & c);  
  
endmodule
```



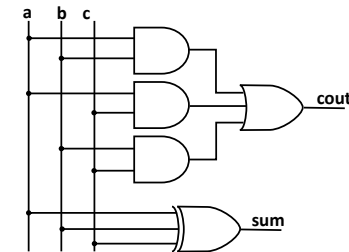
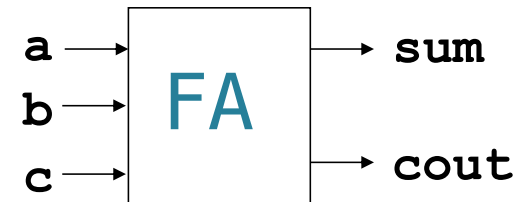
→ **assign:**

- o comando **assign** é usado para **atribuições**. Ele é uma maneira de ligar sinais **combinacionais**, ou seja, que não dependem de *clock* ou de memória.
- toda vez que algum sinal do lado direito da equação muda, o valor do lado esquerdo é atualizado, ou seja, qualquer mudança nas entradas se propaga para a saída.

→ Operadores lógicos bit a bit: ^ (xor), & (and), | (or), ~ (not)

Full Adder (4/4)

```
module FA (  
    input logic a, b, c,  
    output logic sum, cout  
);  
  
    always_comb begin  
        sum = a ^ b ^ c;  
        cout = (a & b) | (a & c) | (b & c);  
    end  
endmodule
```



→ **always_comb** :

- forma recomendada em SystemVerilog para descrever lógica **combinacional**
- usaremos sempre que possível *always_comb*, mas também usaremos para atribuições mais simples *assign*

SystemVerilog suporta portas lógicas

→ descrição estrutural:

```
module FA (  
    input logic a, b, c,  
    output logic sum, cout  
);  
  
    logic t1, t2, t3, t4, t5;  
  
    xor (t1, a, b); // t1 = a ^ b  
    xor (sum, t1, c); // sum = (a ^ b) ^ c  
  
    and (t2, a, b); // t2 = a & b  
    and (t3, a, c); // t3 = a & c  
    and (t4, b, c); // t4 = b & c  
    or (t5, t2, t3); // t5 = (a & b) | (a & c)  
    or (cout, t5, t4); // cout = (a & b) | (a & c) | (b & c)  
  
endmodule
```

Somador de 4 bits (ou, criando hierarquia de projeto)

```
module Adder4 (  
    input logic [3:0] A, B,  
    input logic Cin,  
    output logic [3:0] Sum,  
    output logic Cout  
);
```

Declaração do módulo

Vetores: **logic [n:0]**

```
    logic c1, c2, c3; // sinais intermediários de carry
```

```
// Instancia os FAs conectando carry-out para o próximo carry-in
```

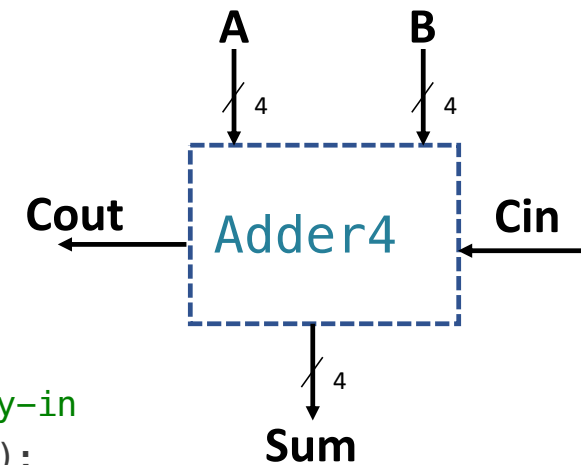
```
FA fa0 (.a(A[0]), .b(B[0]), .c(Cin), .sum(Sum[0]), .cout(c1));
```

```
FA fa1 (.a(A[1]), .b(B[1]), .c(c1), .sum(Sum[1]), .cout(c2));
```

```
FA fa2 (.a(A[2]), .b(B[2]), .c(c2), .sum(Sum[2]), .cout(c3));
```

```
FA fa3 (.a(A[3]), .b(B[3]), .c(c3), .sum(Sum[3]), .cout(Cout));
```

```
endmodule
```



→ **Instanciação:** **<módulo> <nome>**

→ Conexão de sinais às portas de um determinado módulo: **.porta(sinal)**

Somador de 4 bits

```
module Adder4 (
  input logic [3:0] A, B,
  input logic Cin,
  output logic [3:0] Sum,
  output logic Cout
);
```

```
  logic c1, c2, c3; // sinais intermediários de carry
```

```
  // Instancia os FAs conectando carry-out para o próximo carry-in
```

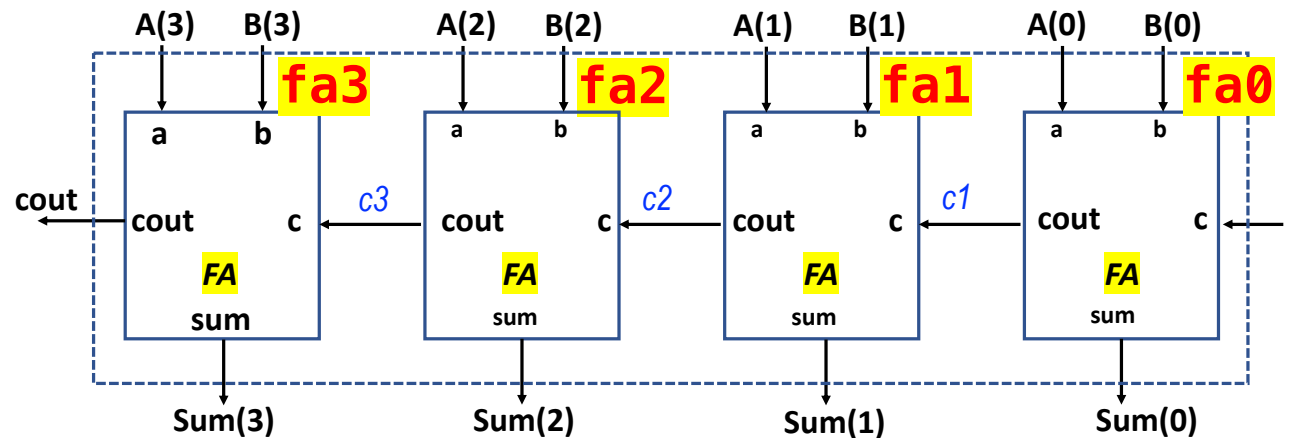
```
  FA fa0 (.a(A[0]), .b(B[0]), .c(Cin), .sum(Sum[0]), .cout(c1));
```

```
  FA fa1 (.a(A[1]), .b(B[1]), .c(c1), .sum(Sum[1]), .cout(c2));
```

```
  FA fa2 (.a(A[2]), .b(B[2]), .c(c2), .sum(Sum[2]), .cout(c3));
```

```
  FA fa3 (.a(A[3]), .b(B[3]), .c(c3), .sum(Sum[3]), .cout(Cout));
```

```
endmodule
```

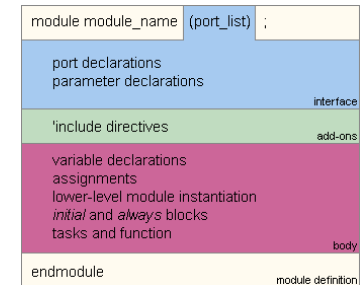


Modo de descrição ineficiente!

Somador de 4 bits – geração com laço

```
module Adder4 #(parameter N = 4)
    ( input logic [N-1:0] A, B,
      input logic Cin,
      output logic [N-1:0] Sum,
      output logic Cout
    );
```

Declaração dos sinais de interface precedida de um parâmetro



```
logic [N:0] carry; // vetor de N+1 bits para carregar Cin até Cout
```

```
assign carry[0] = Cin;
```

```
// Geração de N instâncias de FA
```

```
genvar i;
generate
```

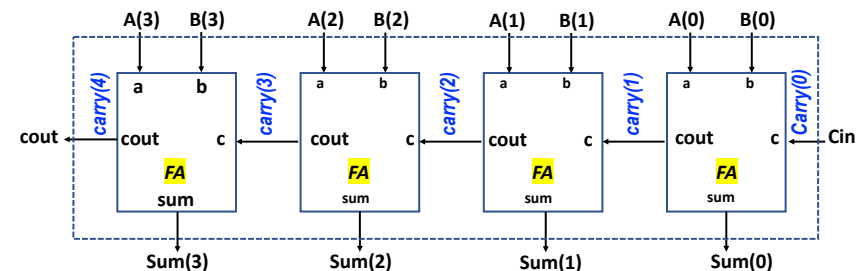
```
    for (i = 0; i < N; i++) begin
        FA fa_inst (.a(A[i]), .b(B[i]), .c(carry[i]), .sum(Sum[i]), .cout(carry[i+1]));
```

```
    end
```

```
endgenerate
```

```
assign Cout = carry[N];
```

```
endmodule
```



- **genvar** : variável apenas para o laço de geração
- **generate** : replicação de estruturas de hardware

Pausa para conceitos de SystemVerilog - vetores (1/2)

// Vetor de 8 bits

logic [7:0] W1;

// W1[7] é o bit mais significativo (MSB)

// Seleção de bit

logic b = W1[3];

// Seleciona o bit de índice 3

// Seleção de faixa de bits (range select)

logic [1:0] faixa = W1[3:2];

// Pega os bits 3 e 2

// Concatenação de vetores

logic [7:0] x, y, z;

logic [23:0] vec = {x, y, z};

// Concatena x, y, z formando um vetor de 24 bits

// Atribuição por fatia (fatiamento com offset) - [start +: width]

logic carry;

logic [1:0] sum;

{carry, sum} = vec[0 +: 3];

// Pega os 3 bits menos significativos (bit 0,1,2) de vec

// Trocar os bytes alto e baixo de um vetor de 16 bits

logic [15:0] w1, w2;

assign w2 = {w1[7:0], w1[15:8]};

// Byte baixo vai para o alto e vice-versa

Pausa para conceitos de SystemVerilog - vetores (2/2)

```
assign vec = {16{x}}; // 16 copies of x
```

```
logic [7:0] out;  
wire [3:0] A;  
assign out = {{4{0}}, A[3:0]};  
assign out = {{4{A[3]}}, A[3:0]};
```

→ Inicializar vetor, ou preencher com valores iguais

```
assign vec = {16{1'b0}}; // zera o vec
```

→ Extensão de zeros (para valor unsigned)

```
assign out = {{4{0}}, A[3:0]};
```

→ Extensão de sinal (para valor signed, complemento de dois)

```
assign out = {{4{A[3]}}, A[3:0]};
```

Pausa para conceitos de SystemVerilog - operadores

→ Operadores lógicos semelhantes aos de C ou Java

```
// Operações em bits individuais (escalares)
logic a, b, c;
assign c = a & b;      // E lógico
assign c = a | b;      // OU lógico
assign c = ~a;         // NÃO lógico
assign c = a ^ b;      // XOR (ou exclusivo)
```

```
// Operações entre vetores (bit a bit)
logic [7:0] vec1, vec2, vec3;
assign vec1 = vec2 & vec3; // AND bit a bit
assign vec1 = vec2 | vec3; // OR bit a bit
assign vec1 = ~vec2;       // Inversão bit a bit
assign vec1 = vec2 ^ vec3;  // XOR bit a bit
```

→ **Redução** de vetor para um único bit

```
logic [7:0] vec;
logic r_and, r_or, r_xor;

assign r_and = &vec;      // AND de todos os bits do vetor vec
assign r_or = |vec;       // OR de todos os bits do vetor vec
assign r_xor = ^vec;      // XOR de todos os bits do vetor vec
```

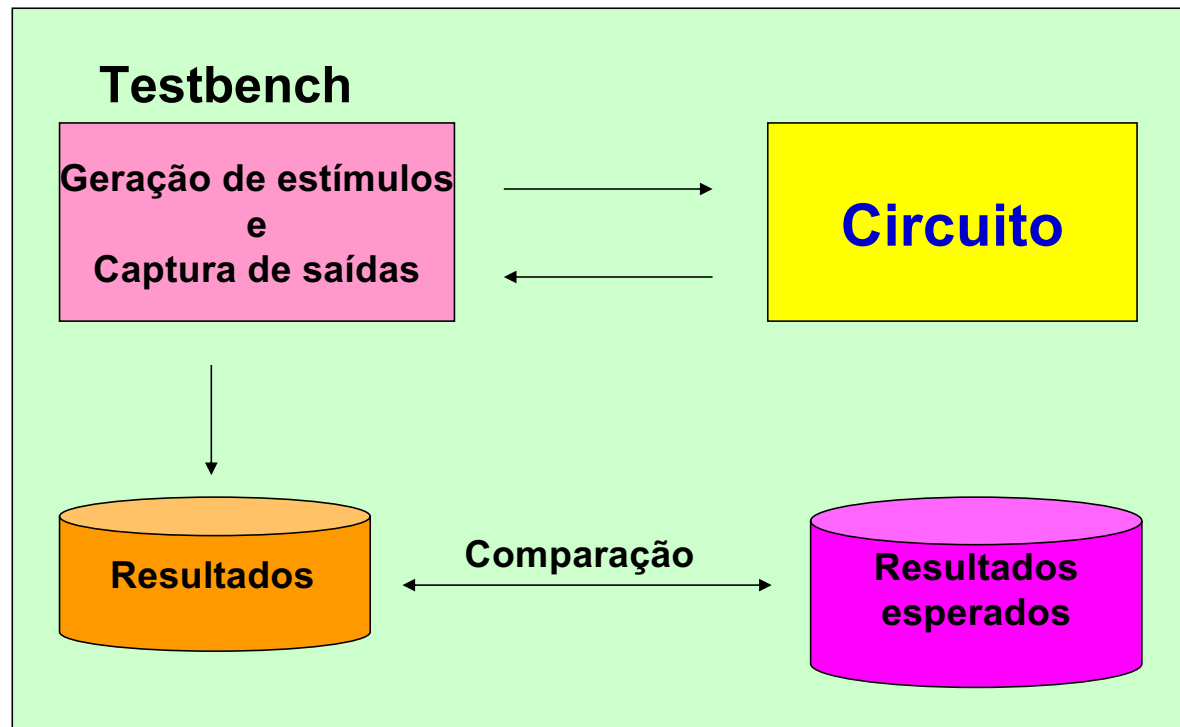
→ Comparações entre vetores (ou bits)

```
logic [3:0] A, B;
logic igual, diferente;
assign igual = (A == B); // Retorna 1 se A e B forem iguais
assign diferente = (A != B); // Retorna 1 se A e B forem diferentes
```

→ Operações aninhadas podem ser feitas livremente

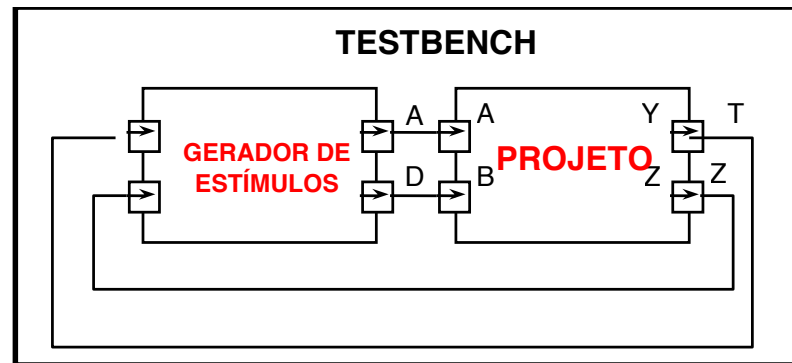
```
logic y;
assign y = (a & ~b) | c; // E entre a e o inverso de b, depois OU com c
```

Ambiente de Verificação




Simulação


- Uma forma simples de testar o projeto: *testbenches*
 - Na sua forma mais simples, um *testbench* consiste em um ou mais “processos geradores de estímulos” e uma **instância** do projeto que se quer testar
 - O *testbench* é construído como um módulo que **não** contém portas de entrada/saída. Ou seja, trata-se de um sistema fechado, ou autônomo






Exemplo de testbench para o FA

```
module tb;  Testbench não tem sinais externos

    logic a, b, c;
    logic sum, cout;


    // Instancia o módulo FA
    FA dut ( .a(a), .b(b), .c(c), .sum(sum), .cout(cout) );  DUT: design under test

    initial begin  executar uma sequência de instruções para controlar a simulação

        // Varredura completa de combinações binárias para {a,b,c}
        for (int i = 0; i < 8; i++) begin
            {a, b, c} = i[2:0];
            #10;  Tempo entre os estímulos
        end
        $finish;  Final da simulação # token representa atraso

    end

endmodule
```

 Geração exaustiva de estímulos (todas as combinações de a, b, c)

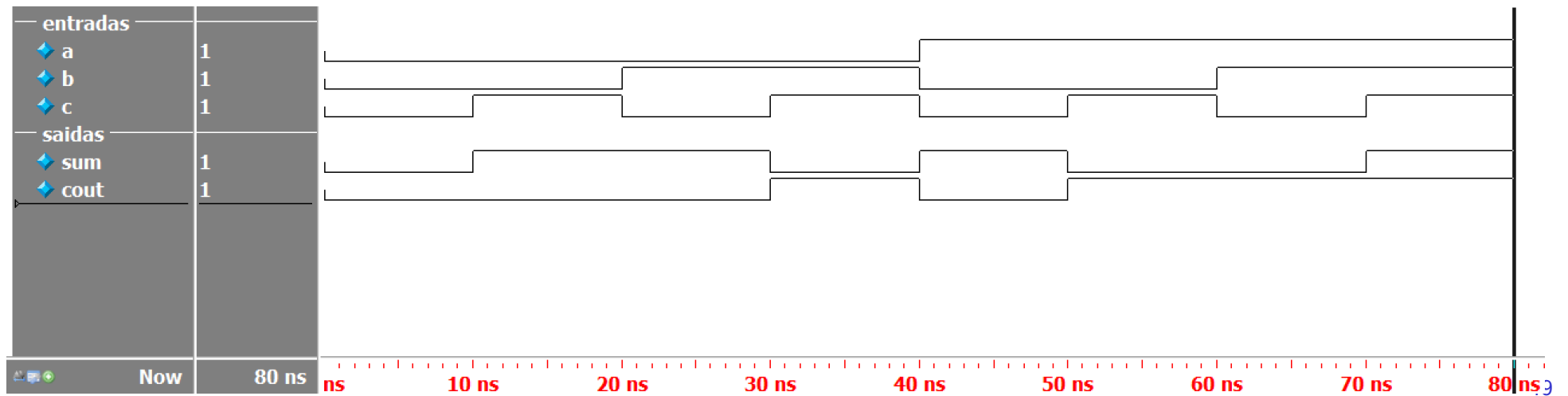
`{a, b, c} = i[2:0];` : para cada valor de `i`, de 0 a 7, concatena-se os sinais a, b, c

Simulação do FA

```
for (int i = 0; i < 8; i++) begin
    {a, b, c} = i[2:0];
    #10;
end

$finish;
```

Visualização da simulação, após execução do testbench:



Exemplo de testbench para o somador de 4 bits

```
module tb_Adder4;
    parameter N = 4;
    logic [N-1:0] A, B;
    logic Cin;
    logic [N-1:0] Sum;
    logic Cout;

    Adder4 #(N(N)) dut ( .A(A), .B(B), .Cin(Cin), .Sum(Sum), .Cout(Cout) );

    // Vetores de entrada constantes (exemplos de 4 bits)
    const logic [N-1:0] vec_A [0:4] = '{4'd3, 4'd7, 4'd2, 4'd8, 4'd15};
    const logic [N-1:0] vec_B [0:4] = '{4'd1, 4'd4, 4'd5, 4'd6, 4'd0};
    const logic vec_Cin[0:4] = '{1'b0, 1'b1, 1'b0, 1'b1, 1'b0};

    initial begin
        for (int i = 0; i < 5; i++) begin
            A = vec_A[i];
            B = vec_B[i];
            Cin = vec_Cin[i];
            #10;
        end
        $finish;
    end
endmodule
```

Testbench não tem sinais externos

Parâmetro N e sinais internos

DUT: design under test

Declaração das entradas na forma de vetores (matriz 2D)

<size>'<base><value>

Tempo entre os estímulos

Final da simulação

Simulação do somador de 4 bits

```
const logic [N-1:0] vec_A [0:4] = '{4'd3, 4'd7, 4'd2, 4'd8, 4'd15};
const logic [N-1:0] vec_B [0:4] = '{4'd1, 4'd4, 4'd5, 4'd6, 4'd0};
const logic vec_Cin[0:4] = '{1'b0, 1'b1, 1'b0, 1'b1, 1'b0};
```

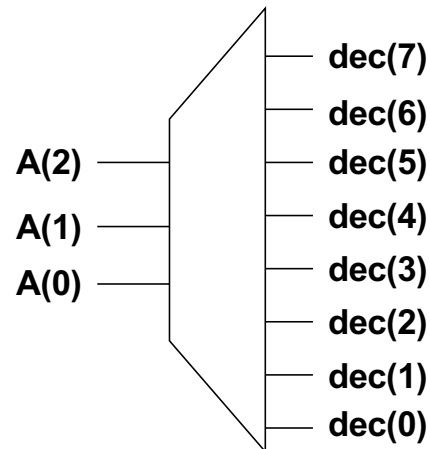
```
for (int i = 0; i < 5; i++) begin
    A = vec_A[i];
    B = vec_B[i];
    Cin = vec_Cin[i];
    #10;
end
```

◆ N	4	4
entradas		
◆ A	15	3 7 2 8 15
◆ B	0	1 4 5 6 0
◆ Cin	0	
saidas		
◆ Sum	15	4 12 7 15
◆ Cout	0	

Circuitos Combinacionais em SystemVerilog

(DE)CODIFICADOR

Decodificador de 3 entradas (**A(2:0)**) e 2^3 (8) saídas (**dec(7:0)**)



Versão com deslocamento à esquerda

```
always_comb begin
    dec = 8'b00000001 << A;
end
```

```
module decoder3to8 (
    input logic [2:0] A,
    output logic [7:0] dec
);

always_comb begin
    case (A)
        3'b000: dec = 8'b00000001;
        3'b001: dec = 8'b00000010;
        3'b010: dec = 8'b00000100;
        3'b011: dec = 8'b00001000;
        3'b100: dec = 8'b00010000;
        3'b101: dec = 8'b00100000;
        3'b110: dec = 8'b01000000;
        3'b111: dec = 8'b10000000;
        default: dec = 8'b00000000;
    endcase
end

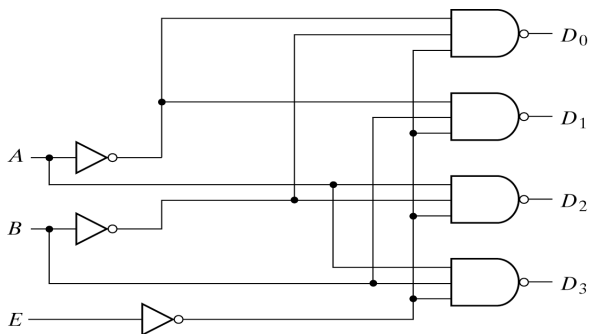
endmodule
```

"casting"
<size>'<base><value>

Recomenda-se ter sempre uma condição default

DECODIFICADOR – com sinal de habilitação

Conforme indicado pela tabela de verdade, apenas uma saída pode ser igual a 0 a qualquer momento, todas as outras saídas são iguais a 1.

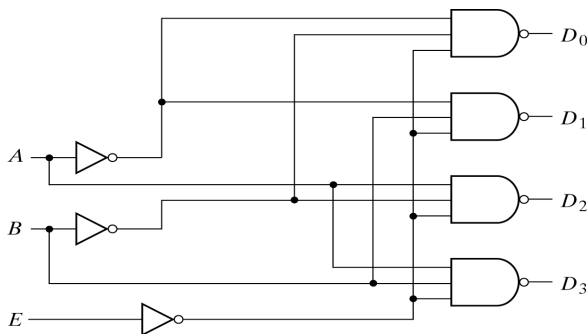


E	A	B	D ₀	D ₁	D ₂	D ₃
1	X	X	1	1	1	1
0	0	0	0	1	1	1
0	0	1	1	0	1	1
0	1	0	1	1	0	1
0	1	1	1	1	1	0

```
module decoder2to4_en (  
    input logic A, B, E,  
    output logic [3:0] D  
);  
  
    logic nA, nB, nE;  
  
    // Inversores  
    not (nA, A);           // nA = ~A  
    not (nB, B);           // nB = ~B  
    not (nE, E);           // nE = ~E (enable ativo em 0)  
  
    // Saídas ativas em 0, diretamente com NANDs  
    nand (D[0], nE, nA, nB); // D[0] = ~(~E & ~A & ~B)  
    nand (D[1], nE, nA, B);  // D[1] = ~(~E & ~A & B)  
    nand (D[2], nE, A, nB);  // D[2] = ~(~E & A & ~B)  
    nand (D[3], nE, A, B);   // D[3] = ~(~E & A & B)  
  
endmodule
```


DECODIFICADOR – com sinal de habilitação

Conforme indicado pela tabela de verdade, apenas uma saída pode ser igual a 0 a qualquer momento, todas as outras saídas são iguais a 1.



<i>E</i>	<i>A</i>	<i>B</i>	<i>D</i> ₀	<i>D</i> ₁	<i>D</i> ₂	<i>D</i> ₃
1	<i>X</i>	<i>X</i>	1	1	1	1
0	0	0	0	1	1	1
0	0	1	1	0	1	1
0	1	0	1	1	0	1
0	1	1	1	1	1	0

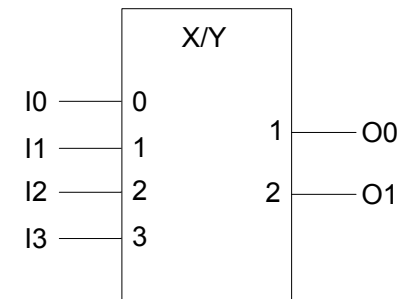
```
module decoder2to4_en (  
    input logic A, B, E,  
    output logic [3:0] D // Saída decodificada ativa em 0  
);  
  
always_comb begin  
    case ({E, A, B}) // concatenação das 3 entradas  
        3'b000: D = 4'b1110;           // E=0, A=0, B=0  
        3'b001: D = 4'b1101;           // E=0, A=0, B=1  
        3'b010: D = 4'b1011;           // E=0, A=1, B=0  
        3'b011: D = 4'b0111;           // E=0, A=1, B=1  
        default: D = 4'b1111;          // E=1 (desabilitado)  
    endcase  
end  
  
endmodule
```

CODIFICADOR

- ✓ O **codificador** binário é um circuito combinacional que indica qual das entradas possíveis está ativa (neste caso, “1”).

$2^n \rightarrow n$

I (3)	I (2)	I (1)	I (0)	O
0	0	0	1	00
0	0	1	0	01
0	1	0	0	10
1	0	0	0	11



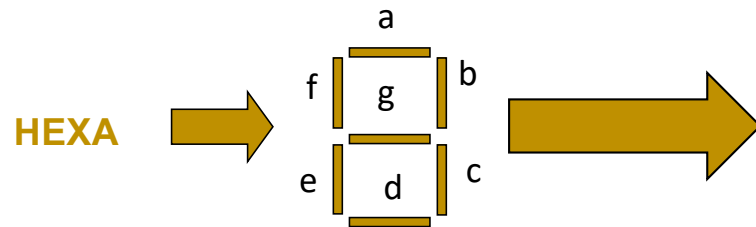
```
module encoder4to2 (
    input logic [3:0] I,
    output logic [1:0] O
);
    always_comb begin
        case (I)
            4'b0001: O = 2'b00;
            4'b0010: O = 2'b01;
            4'b0100: O = 2'b10;
            default: O = 2'b11;
        endcase
    end
endmodule
```

```
module encoder4to2 (
    input logic [3:0] I,
    output logic [1:0] O
);
    always_comb begin
        O = (I == 4'b0001) ? 2'b00 :
            (I == 4'b0010) ? 2'b01 :
            (I == 4'b0100) ? 2'b10 :
            2'b11; // default
    end
endmodule
```

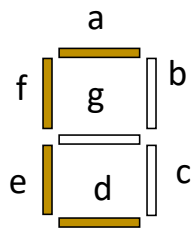
operador condicional ternário,
forma compacta de escrever uma
escolha entre dois valores

Display 7 segmentos

- ✓ Simplificando o *display de 7 segmentos* através da descrição em HDL:



Exemplo: Entrada vetorEnt = "1100" (C) e saída:



Onde, "abcdefg" é a ordem dos bits em saída.

```
module display7seg (
    input logic [3:0] vetorEnt, // Entrada binária (0 a 15)
    output logic [6:0] saida // Saída para display (a até g)
);

always_comb begin
    case (vetorEnt)
        4'b0000: saida = 7'b1111110; // 0
        4'b0001: saida = 7'b0110000; // 1
        4'b0010: saida = 7'b1101101; // 2
        4'b0011: saida = 7'b1111001; // 3
        4'b0100: saida = 7'b0110011; // 4
        4'b0101: saida = 7'b1011011; // 5
        4'b0110: saida = 7'b1011111; // 6
        4'b0111: saida = 7'b1110000; // 7
        4'b1000: saida = 7'b1111111; // 8
        4'b1001: saida = 7'b1110011; // 9
        4'b1010: saida = 7'b1110111; // A
        4'b1011: saida = 7'b0011111; // B
        4'b1100: saida = 7'b1001110; // C
        4'b1101: saida = 7'b0111101; // D
        4'b1110: saida = 7'b1001111; // E
        4'b1111: saida = 7'b1000111; // F
        default: saida = 7'b0000000; // Desliga tudo
    endcase
end

endmodule
```

COMPARADOR

Dois números hexadecimal como entradas, 3 saídas de um bit ($>$, $=$, $<$)

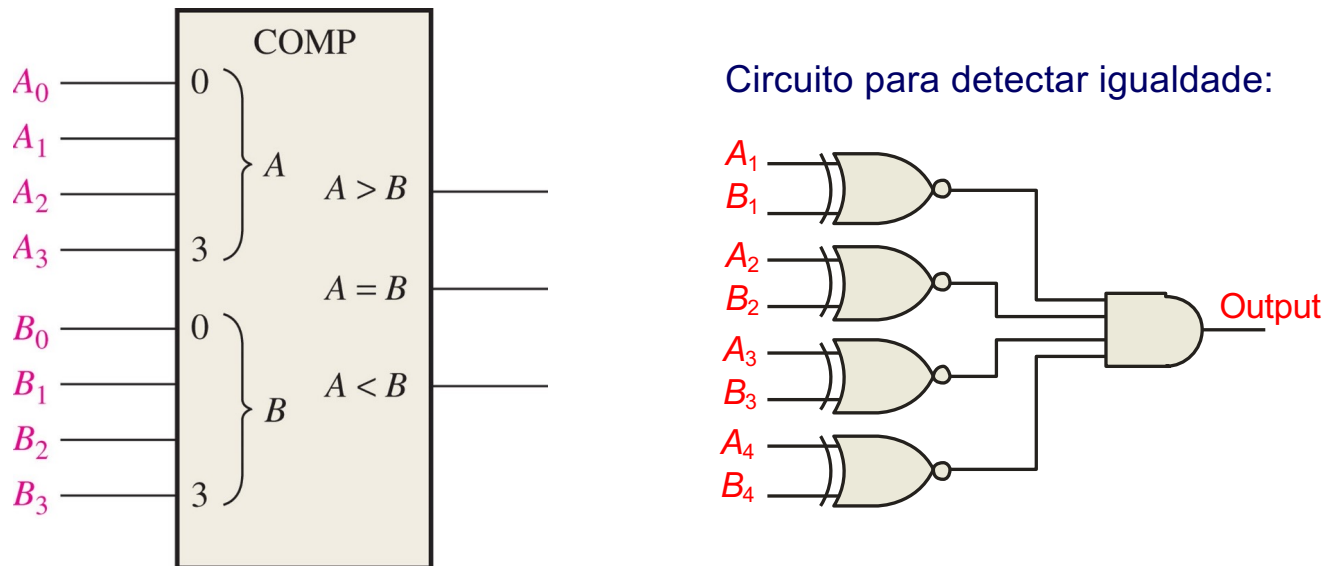
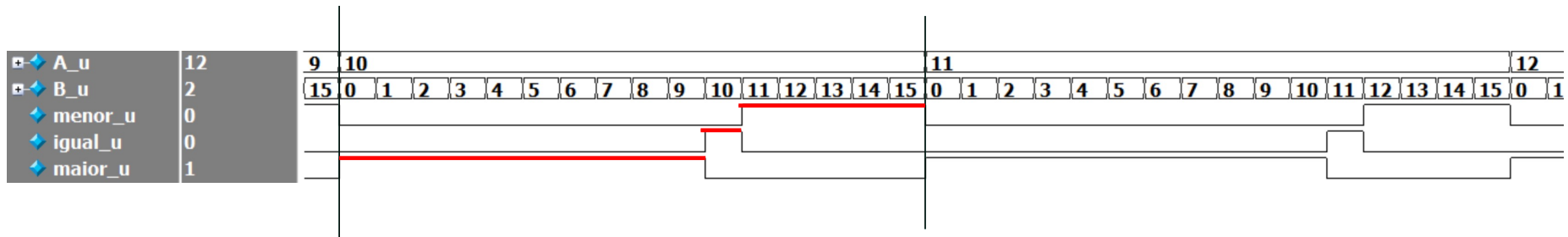


FIGURE 6-21 Logic symbol for a 4-bit comparator with inequality indication.

COMPARADOR – unsigned

```
module comparador_unsigned (  
    input logic [3:0] A, B,  
    output logic igual,  
    output logic maior,  
    output logic menor  
);  
  
always_comb begin  
    igual = (A == B) ? 1'b1 : 1'b0;  
    maior = (A > B) ? 1'b1 : 1'b0;  
    menor = (A < B) ? 1'b1 : 1'b0;  
end  
  
endmodule
```

A e B são interpretados como **valores binários sem sinal** (de 0 a 15)

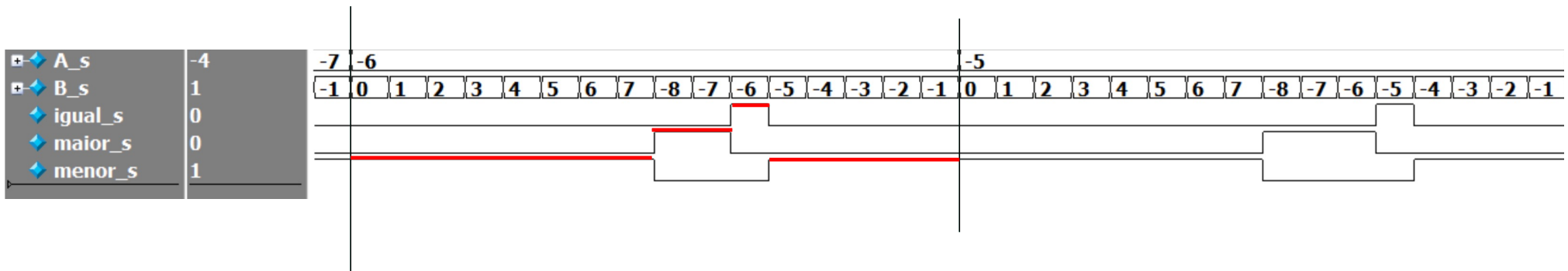


COMPARADOR – signed

```
module comparador_unsigned (  
    input logic signed [3:0] A, B,  
    output logic igual,  
    output logic maior,  
    output logic menor  
);
```

A e B são interpretados como **valores binários inteiros**
(de -8 a 7)

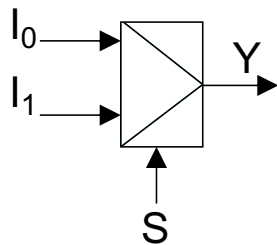
```
always_comb begin  
    igual = (A == B) ? 1'b1 : 1'b0;  
    maior = (A > B) ? 1'b1 : 1'b0;  
    menor = (A < B) ? 1'b1 : 1'b0;  
end  
  
endmodule
```



(DE)MULTIPLEXADOR

- É um circuito que permite selecionar uma dentre várias entradas em função de uma variável de controle

2-to-1 MUX

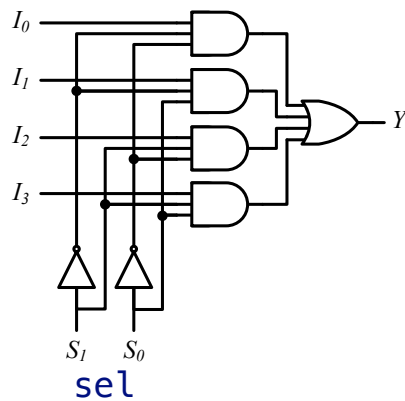


```
module mux2to1_8bit (  
    input logic [7:0] I0, I1,    // entradas de 8 bits  
    input logic S,              // endereço  
    output logic [7:0] Y        // saída de 8 bits  
);  
  
    always_comb begin  
        Y = (S == 0) ? I0 : I1;  
    end  
  
endmodule
```

operador condicional ternário, forma compacta de escrever uma escolha entre dois valores

Multiplexadores (4x1)

4-to-1 MUX



Usando case:

```
always_comb begin
  case (sel)
    2'b00: Y = I0;
    2'b01: Y = I1;
    2'b10: Y = I2;
    2'b11: Y = I3;
    default: Y = 8'h00;
  endcase
end
```

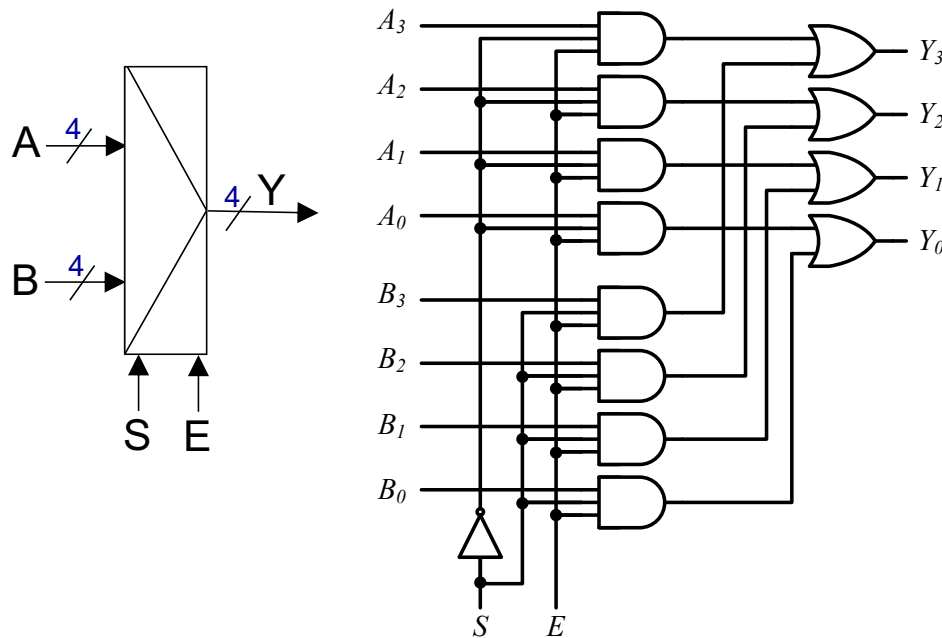
Operador ternário aninhado:

```
always_comb begin
  Y = (sel == 2'b00) ? I0 :
      (sel == 2'b01) ? I1 :
      (sel == 2'b10) ? I2 : I3;
end
```

Seletor como índice de um vetor:

```
always_comb begin
  Y = I[sel];
end
```


Multiplexadores (2x1 - 4 bits com enable)



```

module mux2x1_4bit_en (
    input logic [3:0] I0, I1,
    input logic sel,    // Seletor
    input logic en,     // Enable (ativo em 1)
    output logic [3:0] Y

);

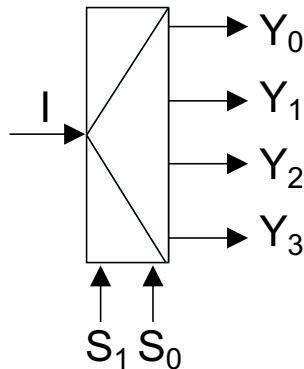
always_comb begin
    Y = (en == 0) ? 4'b0000 :
        (sel == 1) ? I0 : I1; //seleciona I0 ou I1
end

endmodule
    
```

Demultiplexador

- É um circuito que opera de forma inversa ao multiplexador. Ou seja, recebe uma entrada e distribui esta em uma de várias saídas conforme um sinal de seleção
- Exemplo de um multiplexador 1x4:

2 bits de seleção (**S(1:0)**) e 2^2 bits de saídas (**Y(3:0)**)



```
module demux1to4_8bit (  
    input logic [7:0] I,  
    input logic [1:0] S, // Seletor de 2 bits  
    output logic [7:0] Y0, Y1, Y2, Y3 // Saídas de 8 bits  
);
```

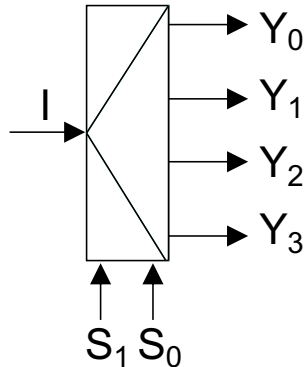
```
always_comb begin  
    Y0 = (S == 2'b00) ? I : 8'b0;  
    Y1 = (S == 2'b01) ? I : 8'b0;  
    Y2 = (S == 2'b10) ? I : 8'b0;  
    Y3 = (S == 2'b11) ? I : 8'b0;  
end
```

```
endmodule
```

Demultiplexador – versão 2

- É um circuito que opera de forma inversa ao multiplexador. Ou seja, recebe uma entrada e distribui esta em uma de várias saídas conforme um sinal de seleção
- Exemplo de um multiplexador 1x4:

2 bits de seleção (**S(1:0)**) e 2^2 bits de saídas (**Y(3:0)**)



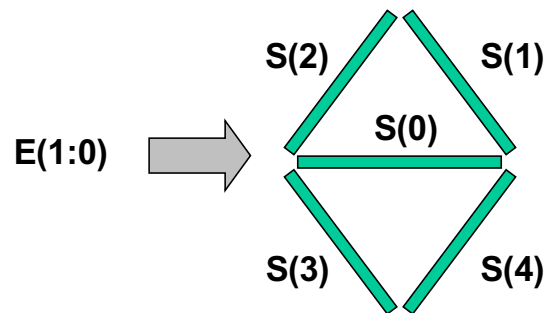
Y sendo vetor de vetor

```
module demux1to4_8_b (  
    input logic [7:0] I,           // Entrada de dados  
    input logic [1:0] S,           // Seletor de 2 bits  
    output logic [7:0] Y [0:3]     // Saídas de 8 bits  
);  
  
    always_comb begin  
        Y = '{default: 8'b0}; // inicializa saídas com 0  
        Y[S] = I;  
    end  
  
endmodule
```

Exercício I

Modele em SystemVerilog o decodificador do display de elevador ilustrado abaixo:

Entrada um vetor de 2 bits que recebe a seguinte codificação



parado	00	-
subindo	01	Λ
descendo	10	∨
defeito	11	todos segmentos acesos

```
always_comb begin
  S = (E == 2'b00) ? 5'b00001 :
      (E == 2'b01) ? 5'b00110 :
      (E == 2'b10) ? 5'b11000 :
                  5'b11111;
end
```

```
always_comb begin
  case (E)
    2'b00: S = 5'b00001;
    2'b01: S = 5'b00110;
    2'b10: S = 5'b11000;
    default: S = 5'b11111;
  endcase
end
```

Exercício 2

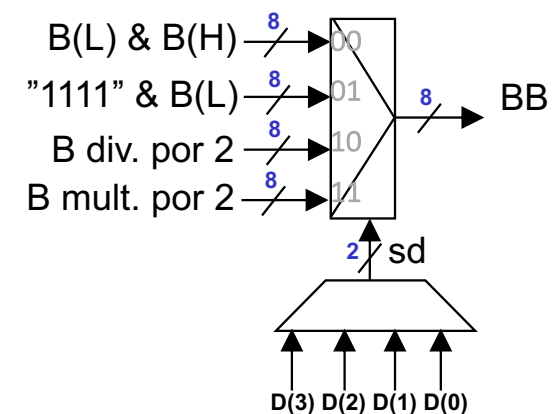
O circuito combinacional ao lado, é composto por um **multiplexador** e um **codificador com prioridade** ($4 \rightarrow 2$).

As entradas do multiplexador são obtidas a partir do sinal **B** de 8 bits.

As entradas do **mux** correspondem:

- ✓ 1ª entrada: inverte os nibbles alto e baixo de B
- ✓ 2ª entrada: fixa o nibble alto em "1111" concatenando-o com a parte baixa de B
- ✓ 3ª entrada: desloca B à direita por 1 bit (B div. por 2)
- ✓ 4ª entrada: desloca B à esquerda por 1 bit (B mult. por 2)

A prioridade do codificador é do bit mais alto ao mais baixo do sinal **D**



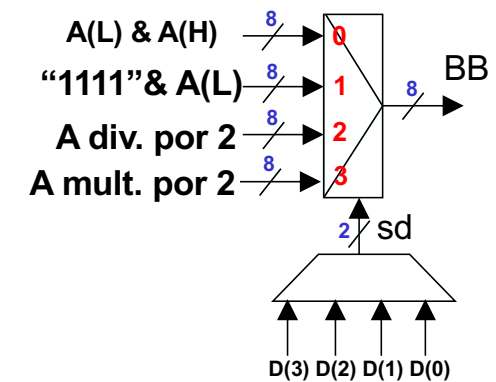
```

module shift_decision (
    input logic [3:0] d,
    input logic [7:0] A, B,
    output logic [7:0] BB
);
    logic [1:0] sd,

    // Decodificação de `d` em `sd`
    always_comb begin
        case (1'b1)
            d[3]: sd = 2'b11;
            d[2]: sd = 2'b10;
            d[1]: sd = 2'b01;
            default: sd = 2'b00;
        endcase
    end

    // Seleção da saída `BB` com base em `sd`
    always_comb begin
        case (sd)
            2'b00: BB = {A[3:0], A[7:4]};
            2'b01: BB = {4'b1, A[3:0]};
            2'b10: BB = {1'b0, A[7:1]};
            default: BB = {A[6:0], 1'b0};
        endcase
    end
endmodule

```



Exercício 3

Implemente um módulo que receba um vetor de 8 bits e produza 1 se o número de bits 1 for ímpar.

Dica: use operador de redução XOR (^).

```
module parity (  
    input logic [7:0] in,  
    output logic odd  
);  
    assign odd = ^in;  
endmodule
```

Exercício 4

Implemente um MUX parametrizável de 8 bits com N entradas ($N = 2^K$).

A seleção é feita com K bits

```
module mux_param #(parameter N = 4, parameter K = 2)
    ( input logic [7:0] in [N],
      input logic [K-1:0] sel,
      output logic [7:0] out
    );

    assign out = in[sel];

endmodule
```


Exercício 5

Implemente um codificador de prioridade com 4 entradas, onde a prioridade é do bit **menos** significativo para o mais **significativo**.

```
module encoder_lowprio (  
    input logic [3:0] I,  
    output logic [1:0] O  
);  
  
    always_comb begin  
        if (I[0])           O = 2'b00;  
        else if (I[1])      O = 2'b01;  
        else if (I[2])      O = 2'b10;  
        else                 O = 2'b11;  
    end  
  
endmodule
```

Exercício 6

Implemente um comparador de 4 bits que funcione com ou sem sinal, baseado em um parâmetro booleano 'SIGNED'.

```
module comparator #(parameter SIGNED = 0) (  
    input logic [3:0] A, B,  
    output logic eq, gt, lt  
);  
  
    generate  
        if (SIGNED) begin  
            logic signed [3:0] sA = A, sB = B;  
            assign eq = (sA == sB);  
            assign gt = (sA > sB);  
            assign lt = (sA < sB);  
        end else begin  
            assign eq = (A == B);  
            assign gt = (A > B);  
            assign lt = (A < B);  
        end  
    endgenerate  
endmodule
```

Simulador MODELSIM

MATERIAL DE APOIO E SIMULADOR

1. Login no LINUX
2. Baixar do Moodle o **comb_sv.zip** com os exercícios de SystemVerilog

Após o *unzip*, o resultado é o conjunto de arquivos ao lado

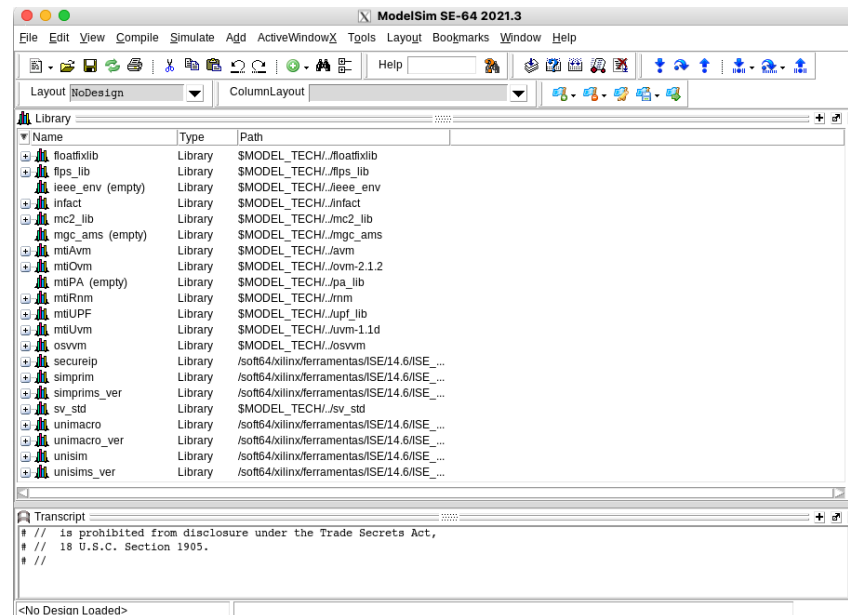
```

comb_sv
├── 01_fa
│   ├── FA.sv
│   ├── sim.do
│   ├── tb_FA.sv
│   └── wave.do
├── 02_soma4
│   ├── FA.sv
│   ├── sim.do
│   ├── sum4.sv
│   ├── tb_sum4.sv
│   └── wave.do
├── 03_somasub
│   ├── FA.sv
│   ├── sim.do
│   ├── sum_sub.sv
│   ├── tb_sum_sub.sv
│   └── wave.do
└── 04_ula
    ├── FA.sv
    ├── sim.do
    ├── sum_sub.sv
    ├── tb_ula.sv
    ├── ula.sv
    └── wave.do

```

Nos laboratórios da PUC:

- Abrir um terminal no Linux
- Digitar: `source /soft64/source_model`
- Iniciar o simulador: `vsim &`



Operação do simulador

Observar que iremos trabalhar através de *scripts* para controlar o processo de simulação

- pode-se trabalhar com a interface gráfica, mas é mais produtivo utilizar *scripts*

Dois scripts que estarão presentes no material de apoio:

- **sim.do** – controla o processo de simulação
- **wave.do** – lista inicial de sinais – pode ser alterado em função dos sinais a serem visualizados, base numérica dos sinais, etc....

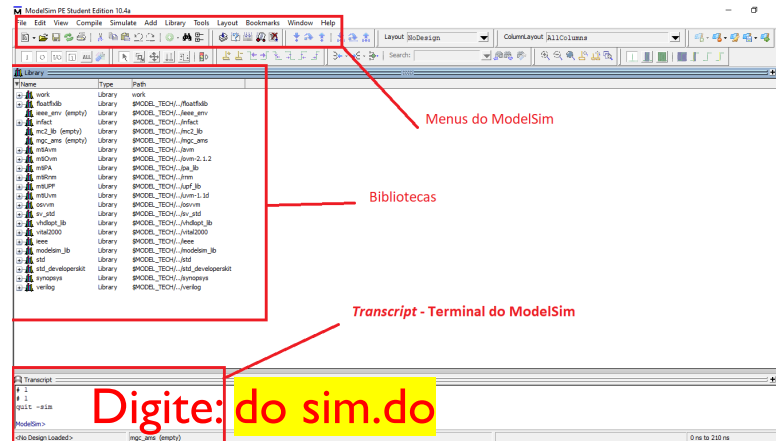
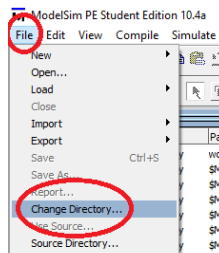
Exemplo de *script* **sim.do** (sintaxe Tcl):

<pre>if {[file isdirectory work]} { vdel -all -lib work } vlib work vmap work work</pre>	}	Verifica se existe um diretório chamado work. Se existir o diretório ele é apagado (usando <i>vdel</i>). Isso garante um ambiente “limpo” antes da nova simulação.
<pre>----- vlog -work work FA.sv vlog -work work sum_sub.sv vlog -work work tb_sum_sub.sv</pre>	}	Cada linha “compila” um arquivo SystemVerilog (.sv) na biblioteca work.
<pre>----- vsim -voptargs=+acc -t ns work.tb</pre>	}	Inicia a simulação do módulo de testbench (tb) na biblioteca work -voptargs=+acc : acesso a todas os sinais (para visualização de ondas) -t ns : unidade de tempo da simulação para nanosegundo
<pre>----- do wave.do</pre>	}	Chama o script <i>wave.do</i> para exibição de sinais selecionados
<pre>----- run 100 ns</pre>	}	Executa a simulação por 100 ns

```
comb_sv  
├── 01_fa  
│   ├── FA.sv  
│   ├── sim.do  
│   ├── tb_FA.sv  
│   └── wave.do  
├── 02_soma4  
│   ├── FA.sv  
│   ├── sim.do  
│   ├── sum4.sv  
│   ├── tb_sum4.sv  
│   └── wave.do  
├── 03_somasub  
│   ├── FA.sv  
│   ├── sim.do  
│   ├── sum_sub.sv  
│   ├── tb_sum_sub.sv  
│   └── wave.do  
└── 04_ula  
    ├── FA.sv  
    ├── sim.do  
    ├── sum_sub.sv  
    ├── tb_ula.sv  
    ├── ula.sv  
    └── wave.do
```

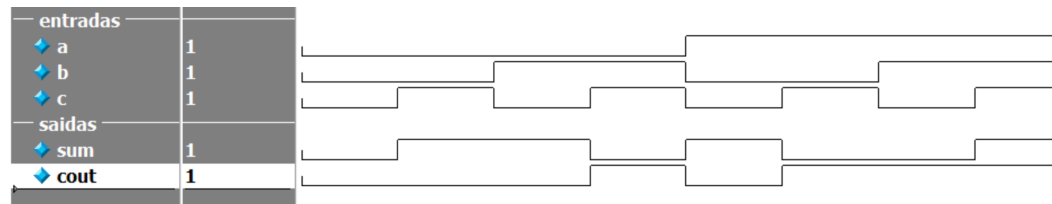
(1) Simulação do HA - meio-somador

Ir para a pasta **01_fa**
Pode-se mudar de pasta via
Interface gráfica:



comb_sv
├─ **01_fa**
│ ├── FA.sv
│ ├── sim.do
│ ├── tb_FA.sv
│ └── wave.do

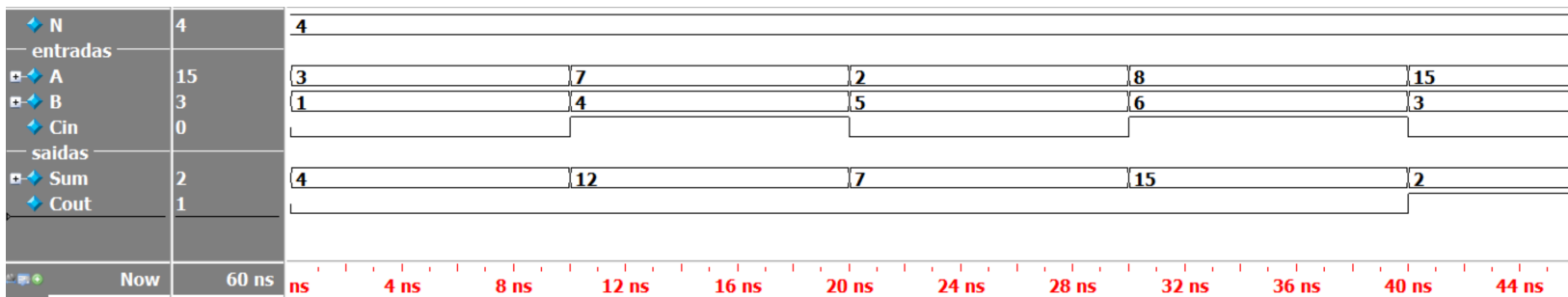
→ Na **aba** Wave deve aparecer a seguinte forma de onda (para o zoom digite **f** sobre as formas de onda):



→ Digite: **quit -sim** para encerrar a simulação e depois **cd ../02_soma4** (sempre no terminal do modelsim)

(2) Simulação do somador de 4 bits

→ Ir para a pasta **02_soma4**, e repetir os procedimentos anteriores e analisar o resultado da simulação (*do sim.do*)



comb_sv

- 02_soma4
 - FA.sv
 - sim.do
 - sum4.sv
 - tb_sum4.sv
 - wave.do

(3) Simulação do somador/subtrator de 4 bits

Ir para a pasta 03_somasub

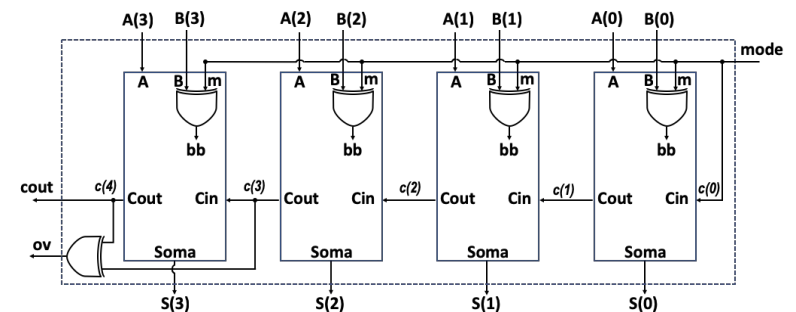
O `sum_sub.sv` está igual ao `soma.sv`

Alterar

- (1) Na descrição do *module*: alterar a entrada **Cin** por **mode**, e acrescentar saída **ov**
- (2) Declarar um vetor **bb** para receber a saída da operação xor entre **B** e **mode**: `logic [N-1:0] bb;`
- (3) Atribuir a `carry[0]` ao **mode** e não o **Cin**: `assign carry[0] = mode;`
- (4) No laço de geração fazer a operação ou exclusivo e inserir **bb** na instanciação

```
genvar i;
generate
  for (i = 0; i < N; i++) begin
    assign bb[i] = B[i] ^ mode; // inverte o B caso seja subtração
    FA fa_inst (.a (A[i]), .b (bb[i]), .c (carry[i]), .sum (Sum[i]), .cout (carry[i+1]));
  end
endgenerate
```

- (5) Fazer ao final a atribuição do **ov**: `assign ov = carry[N] ^ carry[N-1];`



```
comb_sv
├─ 01_fa
├─ .do
├─ 02_soma4
├─ 03_somasub
│   ├── FA.sv
│   ├── sim.do
│   ├── sum_sub.sv
│   ├── tb_sum_sub.sv
│   └─ wave.do
└─ 04_ula
```


(3) Resultado do somador subtrator

Resultado da simulação

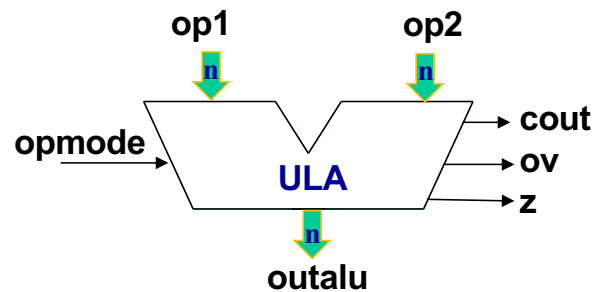
◆ N	4	4
entradas		
◆ A	f	a b 8 5 e f
◆ B	f	6 2 c e 5 f
mode	1	
saidas		
◆ Sum	0	0 d 4 7 9 e 0
◆ Cout	1	
◆ ov	0	

Cinco primeiros testes, como visto na aula de combinacionais:

A	A	B	8	5	E
B	6	2	C	E	5
mode					
S	0	D	4	7	9
C					
V					

ULA

Inicia-se a ULA por um *package*, o qual irá declarar as operações a serem realizadas



```
package p_ula;
```

```
// Declaração do tipo enumerado para operações da ULA
typedef enum logic [3:0] { //até 16 operações
uAND,      // AND lógico
uOR,       // OR lógico
uXOR,      // XOR lógico
uADD,      // Soma
uSUB,      // Subtração
uINC,      // Incremento
uNEG,      // Negação
uSLL,      // Shift lógico à esquerda
uSRL,      // Shift lógico à direita
} op_alu;
```

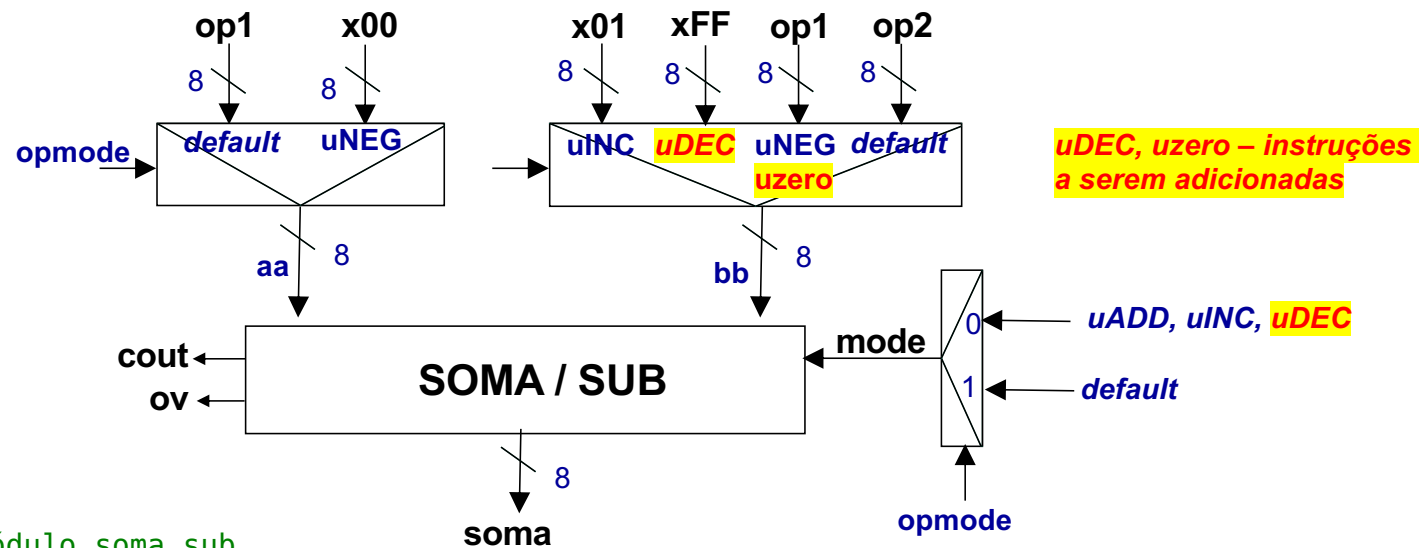
```
endpackage
```

Módulo Aritmético (4 instruções: soma/sub/inc/neg)

```
assign mode = (opmode == uADD || opmode == uINC) ? 1'b0 : 1'b1;
```

```
assign aa = (opmode == uNEG) ? 8'h00 : op1;
```

```
assign bb = (opmode == uINC) ? 8'h01 :  
            (opmode == uNEG) ? op1 :  
            op2;
```



```
// Instancia o módulo soma_sub
```

```
sum_sub #(.N(8)) sb ( .A(aa), .B(bb), .mode(mode), .Sum(soma), .Cout(cout), .ov(ov) );
```

ULA - Descrição completa

```
module ula (
    input logic [7:0] op1, op2,
    input p_ula::op_alu opmode,
    output logic z, cout, ov,
    output logic [7:0] outalu
);
```

```
import p_ula::*;
```

:: → **resolução de escopo (scope resolution)** permite acessar elementos definidos em pacotes ou classes

importa todos os identificadores públicos (tipos, funções, constantes, etc.) do pacote **p_ula**.

```
logic [7:0] aa, bb, soma;
logic mode;
```

```
assign z = (outalu == 8'h00) ? 1'b1 : 1'b0; // Detecta zero na saída
```

```
assign mode = (opmode == uADD || opmode == uINC) ? 1'b0 : 1'b1;
```

```
assign aa = (opmode == uNEG) ? 8'h00 : op1;
```

```
assign bb = (opmode == uINC) ? 8'h01 :
            (opmode == uNEG) ? op1 :
            op2;
```

```
sum_sub #(.N(8)) sb ( .A(aa), .B(bb), .mode(mode), .Sum(soma), .Cout(cout), .ov(ov) );
```

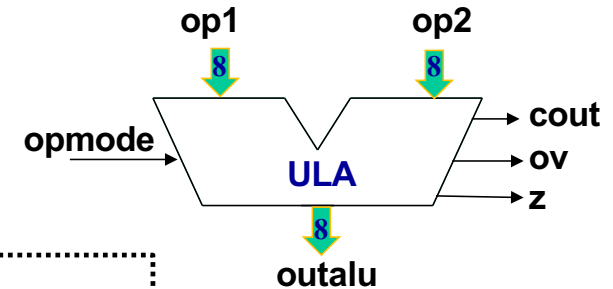
```
// Saída principal da ALU
```

```
always_comb begin
```

```
    outalu = (opmode == uAND) ? (op1 & op2) :
              (opmode == uOR) ? (op1 | op2) :
              (opmode == uXOR) ? (op1 ^ op2) :
              (opmode == uSLL) ? {op1[6:0], 1'b0} :
              (opmode == uSRL) ? {1'b0, op1[7:1]} :
              soma; // default: resultado da unidade soma/sub
```

```
end
```

```
endmodule
```



← **Parte Aritmética**

← **Mux entre parte lógica e aritmética**

Simulando a ULA

		AND						OR						XOR					
		uAND						uOR						uXOR					
◆ opmode	uSLL	aa	ff	34	00	56	05	aa	ff	34	00	56	05	aa	ff	34	00	56	05
+◆ op1	34	55	ff	00	22	01	0a	55	ff	00	22	01	0a	55	ff	00	22	01	0a
+◆ op2	00	00	ff	00				ff		34	22	57	0f	ff	00	34	22	57	0f
+◆ outalu	68																		
◆ z	0																		
◆ cout	1																		
◆ ov	0																		

AND: AA.55= 0, **FF.FF=FF**, 34.00=0, 00.22=0, 56.1=0, 5.A=0

OR: AA+55= FF, FF+FF=FF, 34+00=34, 00+22=22, **56+1=57**, 5+A=F

XOR: **AA⊕55= FF**, FF⊕FF=00, 34⊕00=34, 00⊕22=22, 56⊕1=57, 5⊕A=F

```
const test_array_t padrao_de_teste = '{
  '{8'hAA, 8'h55},
  '{8'hFF, 8'hFF},
  '{8'h34, 8'h00},
  '{8'h00, 8'h22},
  '{8'h56, 8'h01},
  '{8'h05, 8'h0A}
};
```

Simulando a ULA

◆ opmode		9	uXOR																									
◆ entradas			uADD																									
◆ op1		-1	5	-86	-1	52	0	86	5	-86	-1	52	0	86	5	-86	-1	52	0	86	5	-86	-1	52	0	86	5	
◆ op2		-1	1	10	85	-1	0	34	1	10	85	-1	0	34	1	10	85	-1	0	34	1	10	85	-1	0	34	1	10
◆ saidas			uSUB																									
◆ outalu		0	15	-1	-2	52	34	87	15	85	0	52	-34	85	-5	-85	0	53	1	87	6	86	1	-52	0	-86	-5	
◆ z		1																										
◆ cout		1																										
◆ ov		0																										

Valores em decimal (complemento de dois)

ADD: $-86+85=-1$, $-1+-1=-2$, ...

SUB: $-86-85=85$ (ov=1), $-1-(-1)=0$, ...

INC: soma 1 ao op1 - $-86+1=-85$, $-1+1=0$, ...

NEG: $-(-86)=86$, $-(-1)=1$, ...

Simulando a ULA

opmode	9	uNEG	uSLL					uSRL							9
entradas															
op1	170	5	170	255	52	0	86	5	170	255	52	0	86	5	170
op2	85	1	10	85	255	0	34	1	10	85	255	0	34	1	10
saidas															
outalu	85	251	84	254	104	0	172	10	85	127	26	0	43	2	85
z	0														
cout	1														
ov	1														

Valores em inteiros

SSL: $52 * 2 = 104$,

SSR: $170 / 2 = 85$, $52 / 2 = 25$

(3) Simulação da ULA

```
module tb;

import p_ula::*;

logic [7:0] op1, op2, outalu;
logic z, cout, ov;
op_alu opmode;

// Estrutura de teste
typedef struct packed {
    logic [7:0] a;
    logic [7:0] b;
} test_record_t;

//Inicialização dos vetores de teste
typedef test_record_t test_array_t[];

const test_array_t padrao_de_teste = '{
    '{8'hAA, 8'h55},
    '{8'hFF, 8'hFF},
    '{8'h34, 8'h00},
    '{8'h00, 8'h22},
    '{8'h56, 8'h01},
    '{8'h05, 8'h0A}
};

// Instanciação da ULA
ula dut ( .op1(op1), .op2(op2), .outalu(outalu), .z(z), .cout(cout),
    .ov(ov), .opmode(opmode) );

// Processo de teste
initial begin
    // varredura de operações até a última
    for (int op = 0; op <= 10; op++) begin
        opmode = op_alu'(op);

        foreach (padrao_de_teste[i]) begin
            op1 = padrao_de_teste[i].a;
            op2 = padrao_de_teste[i].b;
            #10;
        end
    end

    $finish;
end

endmodule
```


(3) Simulação da ULA

Exercício:

- **modificar** o código para incluir **zero** e **dec**

		Msgs															
◆ opmode		uDEC	uSRL					uZero					uDEC				
— entradas																	
▣ ◆ op1	05		00	56	05	aa	ff	34	00	56	05	aa	ff	34	00	56	05
▣ ◆ op2	0a		22	01	0a	55	ff	00	22	01	0a	55	ff	00	22	01	0a
— saidas																	
▣ ◆ outalu	04		00	2b	02	.00						a9	fe	33	ff	55	04
◆ z		0															
◆ cout		1															
◆ ov		0															