Discente: Samuel Oliveira Silva Bianch
Lista 1

Todos os códigos estão também disponíveis no meu repositório do GitHub: https://github.com/samuelbianch/sistemas_embarcados

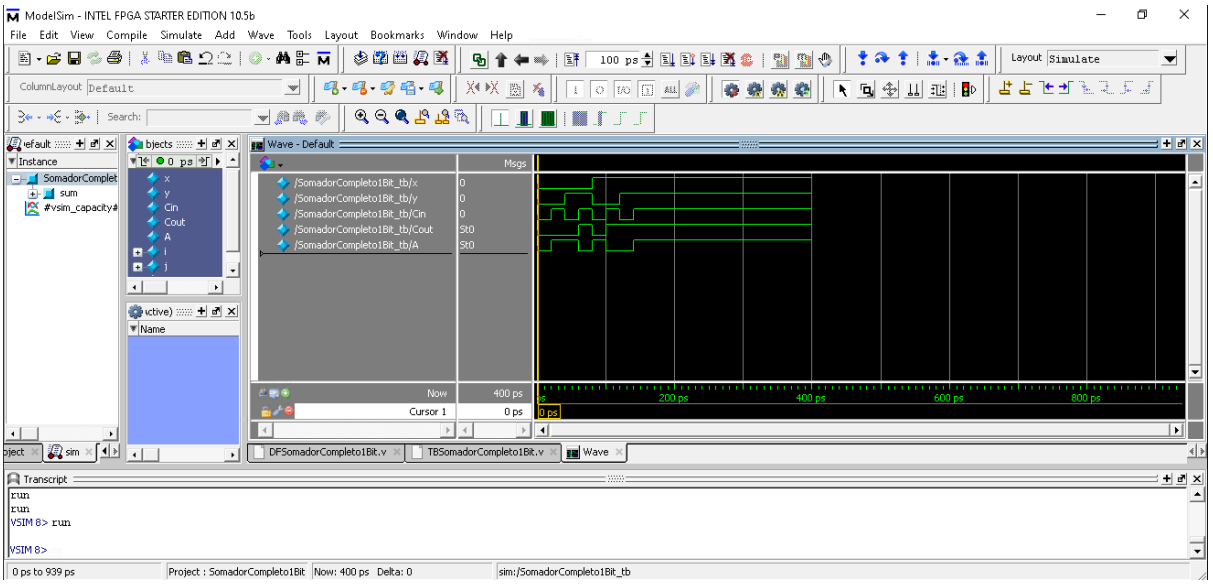
1 - Implemente o somador completo de 1-bit
Segue implementações

```
// Data Flow Modeling
module SomadorCompleto1Bit(x, y, Cin , Cout, A);
        input x, y, Cin;
        output Cout, A;
        assign A = (x ^ y) ^ Cin;
        assign Cout = ((x ^ y) & Cin) | (x & y);
endmodule

// Behavioral Modeling
module SomadorCompleto1Bit(x, y, Cin , Cout, A);
input x, y, Cin;
        output reg Cout, A;
        always @(x, y, Cin) begin
        A = (x ^ y) ^ Cin;
        Cout = ((x ^ y) & Cin) | (x & y);
        end
endmodule

// Structural Modeling
module SomadorCompleto1Bit(x, y, Cin , Cout, A);
        input x, y, Cin;
        output Cout, A;
        wire p, r, s;
        xor(p, x, y);
        xor(A, p, Cin);
        and(r, p, Cin);
        and(s, x, y);
        or(Cout, r, s);
endmodule
```

Desta forma, verifica-se as semelhanças entre a linguagem Behavioral Modeling e Data Flow Modeling, diferenciando só na sua capacidade de sincronia das implementações de portas. Porém, a linguagem estrutural é muito fácil de ser abstraída para o código da linguagem Verilog.

Para este exercício foi implementado um *test bench* que varia todas as combinações possíveis do circuito, com 3 *for*.

```
// Test Bench
module SomadorCompleto1Bit_tb;
    reg x, y, Cin;
    wire Cout, A;
    integer i, j, k;
    // Instanciando o módulo a ser testado
    SomadorCompleto1Bit sum (
        .x(x),
        .y(y),
        .Cin(Cin),
        .Cout(Cout),
        .A(A)
    );

    // Testando todas as combinações possíveis de entrada
    initial begin
        for (i = 0; i <= 1; i = i+1) begin
            for (j = 0; j <= 1; j = j+1) begin
                for (k = 0; k <= 1; k = k+1) begin
                    x = i;
                    y = j;
                    Cin = k;
                    #5; // Aguarda alguns ciclos para as saídas se estabilizarem
                end
            end
        end
    end
endmodule
```



Verifica-se então na simulação as saídas Cout e A, de acordo com as mudanças de estados de X, Y e Cin.

2 - Com base no somador de 1-bit do exercício anterior, implemente um somador completo de 8 bits.

```
// Structural Modeling
module SomadorCompleto1Bit(x, y, Cin , Cout, A);
input [7:0]x;
input [7:0]y;
input Cin;
output Cout;
output [7:0]A;
wire [7:0]p;
wire [7:0]r;
wire [7:0]s;
genvar i, j, k;

for(i = 0; i<8; i=i+1) begin
  for(j = 0; j<8; j=j+1) begin
    for(k = 0; k<8; k=k+1) begin
      xor(p[i], x[j], y[k]);
    end
  end
end

for(i = 0; i<8; i=i+1) begin
  for(j = 0; j<8; j=j+1) begin
    xor(A[i], p[i], Cin);
  end
end

for(i = 0; i<8; i=i+1) begin
  for(j = 0; j<8; j=j+1) begin
    and(r[i], p[i], Cin);
  end
end

for(i = 0; i<8; i=i+1) begin
  for(j = 0; j<8; j=j+1) begin
    for(k = 0; k<8; k=k+1) begin
      and(s[i], x[j], y[y]);
    end
  end
end

for(i = 0; i<8; i=i+1) begin
  for(j = 0; j<8; j=j+1) begin
    or(Cout, r[i], s[j]);
  end
```

```
end


endmodule

// Behavior Modeling
module SomadorCompleto8Bits(
    input [7:0] x,
    input [7:0] y,
    input Cin,
    output reg [7:0] A,
    output reg Cout
);

reg [7:0] sum;
reg [7:0] carry;

always @(x, y, Cin) begin
  sum = x + y + Cin;
  Cout = carry[7];
  A = sum;
end

endmodule

// Data Flow
module SomadorCompleto8BitsDF(
    input [7:0] x,
    input [7:0] y,
    input Cin,
    output [7:0] A,
    output Cout
);

wire [7:0] sum;
wire [7:0] carry;

assign sum = x + y + Cin;
assign Cout = carry[7];
assign A = sum;

endmodule
```

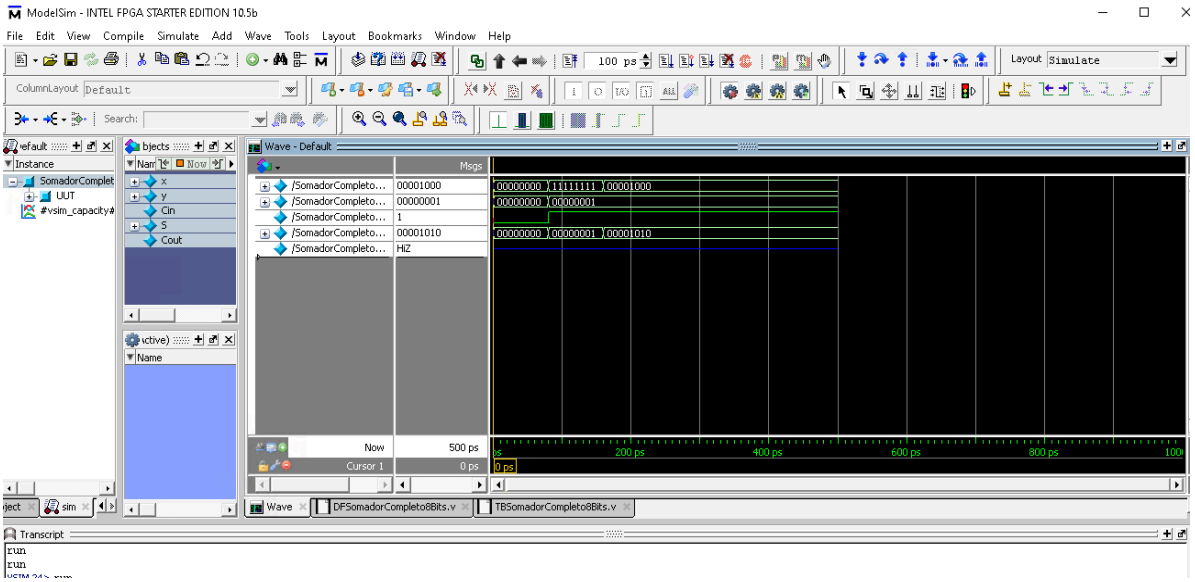
Evidencia-se a simplicidade no modelo Data Flow.

Segue meu exemplo para teste:

```
module SomadorCompleto8Bits_tb;
```

```
    reg [7:0] x;
    reg [7:0] y;
    reg Cin;
    wire [7:0] S;
    wire Cout;

    // Instanciando o módulo a ser testado
    SomadorCompleto8BitsDF UUT (
        .x(x),
        .y(y),
        .Cin(Cin),
        .A(S),
        .Cout(Cout)
    );
    initial begin
        // Inicialização das entradas
        x = 8'b00000000;
        y = 8'b00000000;
        Cin = 0;

        // Espera alguns ciclos antes de começar
        #80;

        // A soma de 255 + 1 + 1, que deve resultar em 1 com Cout 1
        x = 8'b11111111;
        y = 8'b00000001;
        Cin = 1;

        #80;
       // A soma de 8 + 1 + 1, que deve resultar em 10 com Cout 10
        x = 8'b00001000;
        y = 8'b00000001;
        Cin = 1;
    end
endmodule
```

Aparentemente o carry não está funcionando, estou debugando o projeto para conferir o que pode ser.



3 - Implemente os Flip Flops tipo: SR, JK, T e D. Os FFs devem ser implementados no nível de portas lógicas.

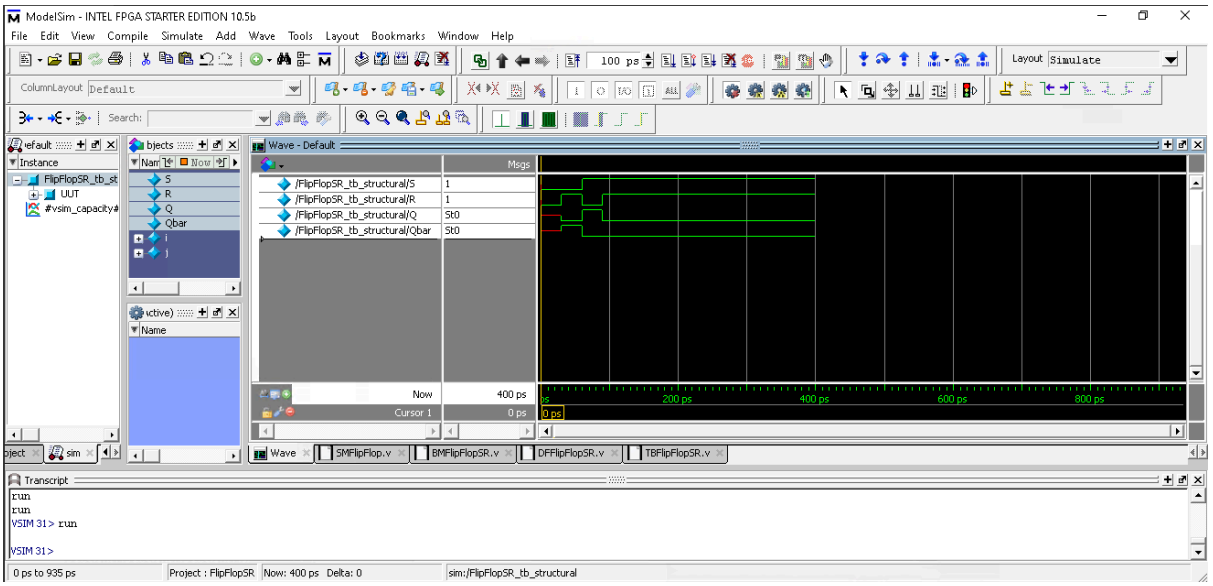
```
// Structure Model
module SMFlipFlopSR(
    input S, R,
    reg Q, Qbar
);

    nor n1(Qbar, S, Q);
    nor n2(Q, R, Qbar);

endmodule

// Bahavior Modeling
module BMFlipFlopSR(
    input S, R,
    output reg Q, Qbar
);

    always @(S, R) begin
        Qbar = ~(S | Q);
        Q = ~(R | Qbar);
    end
endmodule

// Data Flow
module DFFlipFlopSR(
    input S, R,
    output Q, Qbar
);
```

```
    assign Qbar = ~(S | Q);
    assign Q = ~(R | Qbar);
endmodule
```

Uma novidade que encontrei, foi que quando usamos o (*) dentro do always no Bahavior Modeling, o compilador sempre retorna um *warning* recomendando o desenvolvedor sempre usar as variáveis no parâmetro do loop.

Para testar nosso circuito, pode-se usar este exemplo:

```
// Test Bench
module FlipFlopSR_tb_structural;
    reg S, R;
    wire Q, Qbar;
    integer i, j;
    DFFlipFlopSR UUT (
        .S(S),
        .R(R),
        .Q(Q),
        .Qbar(Qbar)
    );
    // Testando o flip-flop SR
    initial begin
        for(i = 0; i<=1; i=i+1) begin
            for(j = 0; j<=1; j=j+1) begin
                S=i;
                R=j;
                #30;
            end
        end
    end
endmodule
```



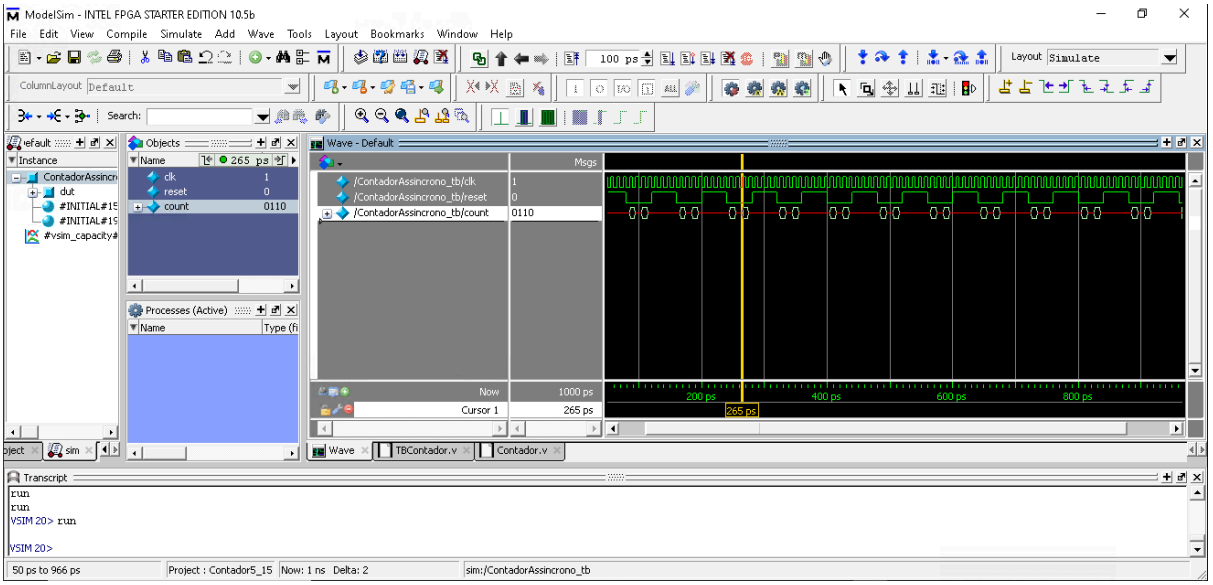
4 - **Usando o Flip Flop adequado, Implemente um contador assíncrono de 4 bits que conte de 5 até 15. O contador deve possuir um botão de reset para reiniciar a contagem.**

```
// Exemplo de Contador Assíncrono utilizando o Flip Flop do Tipo D de 4 Bits
module ContadorAssincrono(
    input clk, // Sinal de clock
    input reset, // Botão de reset
    output reg [3:0] count // Saída do contador de 4 bits
);

    // Flip-flops D para cada bit do contador
    reg [3:0] next_count;

    always @(posedge clk or posedge reset) begin
        if (reset) begin
            // Reiniciar a contagem se o botão de reset estiver pressionado
            next_count <= 4'b0101; // Iniciar a contagem em 5 (0101)
        end
            if ( c == 4'b1111) begin
                    c <= 4'b0101;
            end else begin
            // Incrementar o contador
            next_count <= count + 4'b0001;
        end
    end

            assign count = next_count;

    // Atribuir o próximo estado ao estado atual do contador
    always @(posedge clk) begin
        if (!reset) begin
            count <= next_count;
        end
    end

endmodule
```

Acredito que este tenha sido o mais difícil de ser implementado, por ser da criatividade de implementação do desenvolvedor, porém quando entendemos melhor o problema, ele se torna relativamente simples.

```
// Test Bench
module ContadorAssincrono_tb;
    // Sinais
    reg clk;       // Sinal de clock
    reg reset;     // Botão de reset
    wire [3:0] count; // Saída do contador de 4 bits
```

```
    // Instância do módulo a ser testado
    ContadorAssincrono dut (
        .clk(clk),
        .reset(reset),
        .count(count)
    );

    // Teste
    initial begin
            clk = 0;
            forever #5 clk = ~clk;
    end
    initial begin
            reset = 0;
            forever #40 reset = ~reset;
    end

endmodule
```



5 - **Desenvolva o circuito de uma Unidade Lógica e Aritmética (ULA) simples, mostrada na figura abaixo. Os resultados do testbench devem incluir variações no maior número possível de variáveis de entrada.**

```
// Structure Model
module StructureModelUlda (
  input A, B, Ainv, Binv, Cinv,
  output reg [2:0] op,
  output Cout,
  reg S
);

wire cabo1, cabo2, cabo3;
```

```
// Fazendo o Mux 2 -> 1 para o A
and(cabo1, A, ~Ainv);
and(cabo2, ~A, Ainv);
or(cabo3, cabo1, cabo2);

wire b1, b2, b3;
// Fazendo o Mux 2 -> 1 para o A
and(b1, B, ~Binv);
and(b2, ~B, Binv);
or(b3, b1, b2);

wire p1, p2, p3, p4;
and(p1, cabo3, b3);
or(p2, cabo3, b3);
xor(p3, cabo3, b3);
nor(p4, cabo3, b3);

wire sum;
// Half Adder
xor(sum, cabo3, b3);
and(Cout, cabo3, b3);

// Mux 5 to 1
always @(op) begin
if (op[0] == 0 & op[1] == 0 & op[2] == 0) S = p1;
if (op[0] == 0 & op[1] == 0 & op[2] == 1) S = p2;
if (op[0] == 0 & op[1] == 1 & op[2] == 0) S = p3;
if (op[0] == 0 & op[1] == 1 & op[2] == 1) S = p4;
if (op[0] == 1 & op[1] == 0 & op[2] == 0) S = sum;
end

endmodule

// Behavior Model
module BehaviorModelUlda (
  input A, B, Ainv, Binv, Cinv,
  input [2:0] op,
  output reg S,
  output reg Cout
);

reg cabo1, cabo2, cabo3;
reg b1, b2, b3;
reg p1, p2, p3, p4;
reg sum;

always @(*) begin
```

```
    // Fazendo o Mux 2 -> 1 para o A
    if (Ainv)
        cabo1 = A;
    else
        cabo1 = ~A;
    if (Ainv)
        cabo2 = ~A;
    else
        cabo2 = A;
    cabo3 = cabo1 | cabo2;

    // Fazendo o Mux 2 -> 1 para o B
    if (Binv)
        b1 = B;
    else
        b1 = ~B;
    if (Binv)
        b2 = ~B;
    else
        b2 = B;
    b3 = b1 | b2;

    // Portas lógicas para p1, p2, p3, p4
    p1 = cabo3 & b3;
    p2 = cabo3 | b3;
    p3 = cabo3 ^ b3;
    p4 = ~(cabo3 | b3);

    // Half Adder
    sum = cabo3 ^ b3;
    Cout = cabo3 & b3;

    // Mux 5 to 1
    case (op)
        3'b000: S = p1;
        3'b001: S = p2;
        3'b010: S = p3;
        3'b011: S = p4;
        3'b100: S = sum;
        default: S = 1'b0;
    endcase
end

endmodule

// Data Flow
module DataFlowModelUlda (
  input A, B, Ainv, Binv, Cinv,
```

```
  input [2:0] op,
  output S,
  output Cout
);

// Fazendo o Mux 2 -> 1 para o A
wire cabo1 = A & ~Ainv;
wire cabo2 = ~A & Ainv;
wire cabo3 = cabo1 | cabo2;

// Fazendo o Mux 2 -> 1 para o B
wire b1 = B & ~Binv;
wire b2 = ~B & Binv;
wire b3 = b1 | b2;

// Portas lógicas para p1, p2, p3, p4
wire p1 = cabo3 & b3;
wire p2 = cabo3 | b3;
wire p3 = cabo3 ^ b3;
wire p4 = ~(cabo3 | b3);

// Half Adder
wire sum = cabo3 ^ b3;
assign Cout = cabo3 & b3;

// Mux 5 to 1
assign S = (op == 3'b000) ? p1 :
          (op == 3'b001) ? p2 :
          (op == 3'b010) ? p3 :
          (op == 3'b011) ? p4 :
          (op == 3'b100) ? sum : 1'b0;

endmodule

// Test Bench
module DataFlowModelUlda_tb;

    // Parâmetros
    parameter PERIOD = 10;

    // Sinais de Entrada
    reg A, B, Ainv, Binv, Cinv;
    reg [2:0] op;

    // Sinais de Saída
    wire S, Cout;

    // Instância do Módulo a ser testado
```

```
    DataFlowModelUlda dut (
        .A(A),
        .B(B),
        .Ainv(Ainv),
        .Binv(Binv),
        .Cinv(Cinv),
        .op(op),
        .S(S),
        .Cout(Cout)
    );

    // Estímulo
    initial begin
        A = 0; B = 0; Ainv = 0; Binv = 0; Cinv = 0; op = 0;

        // Teste 1
        #20 A = 1; B = 0; Ainv = 1; Binv = 1; Cinv = 0; op = 0;
        #20 A = 0; B = 1; Ainv = 1; Binv = 1; Cinv = 0; op = 1;
        #20 A = 1; B = 1; Ainv = 0; Binv = 0; Cinv = 1; op = 2;
        #20 A = 0; B = 0; Ainv = 0; Binv = 0; Cinv = 1; op = 3;
        #20 A = 1; B = 1; Ainv = 1; Binv = 0; Cinv = 1; op = 4;
        #20;
    end

endmodule
```

Para facilitar o entendimento, construí também no meu Logisim este circuito.