

FEC - Reed Solomon - Aplicado em Verilog

Autores: **Ausilon V. Souza , Erick Moreira, Lucas S. Ross , Luiz R. L. Rodriges, Nicolas F. Amaral, Rafael A. Reis, and Tarciso G. B. de Bell**

Tópicos:

- Introdução
- Implementação - Descrição dos Módulos
- Conclusão

INTRODUÇÃO

O código Reed-Solomon (RS) é uma das técnicas mais utilizadas de correção de erros (FEC – Forward Error Correction) em sistemas digitais. Ele foi desenvolvido em 1960 por Irving Reed e Gustave Solomon, e desde então se consolidou como referência em confiabilidade para transmissão e armazenamento de dados.

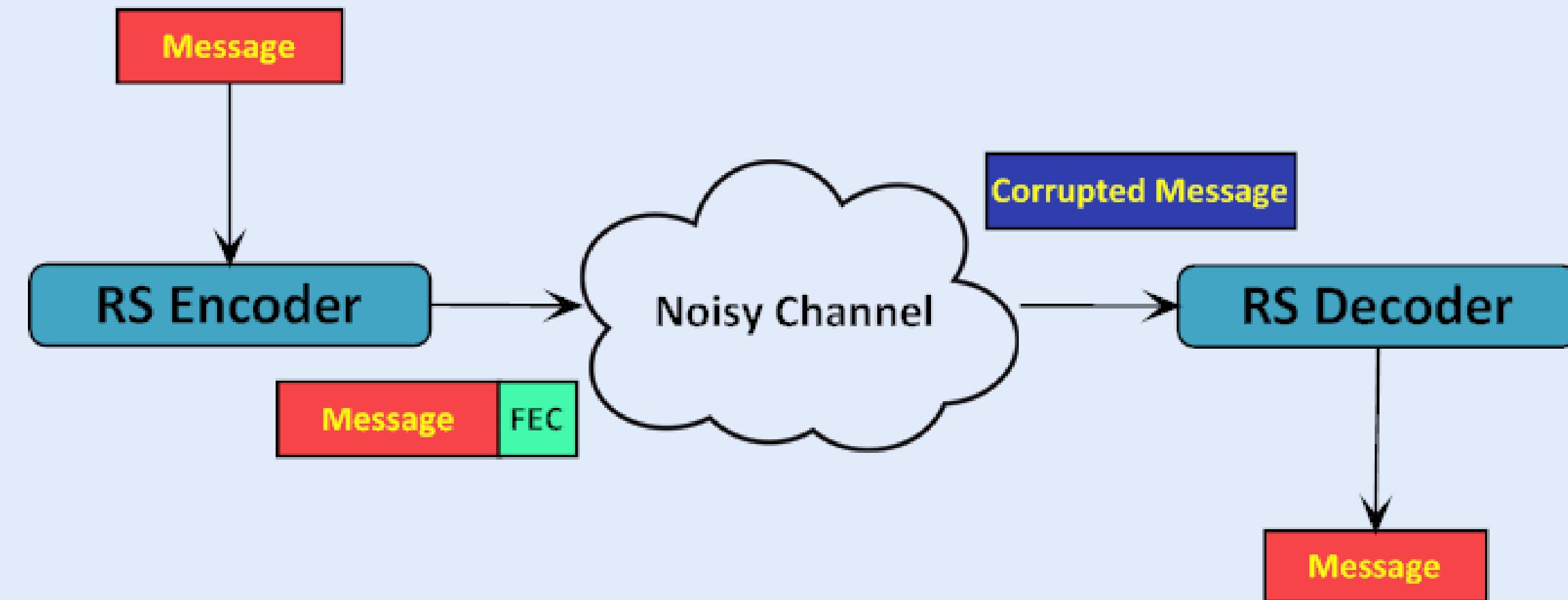
INTRODUÇÃO

O funcionamento do RS é baseado em operações matemáticas sobre campos finitos, conhecidos como campos de Galois (GF). Essas operações permitem que o código RS detecte e corrija múltiplos erros dentro de um bloco de dados, o que o torna ideal para canais ruidosos e ambientes propensos a falhas.

INTRODUÇÃO

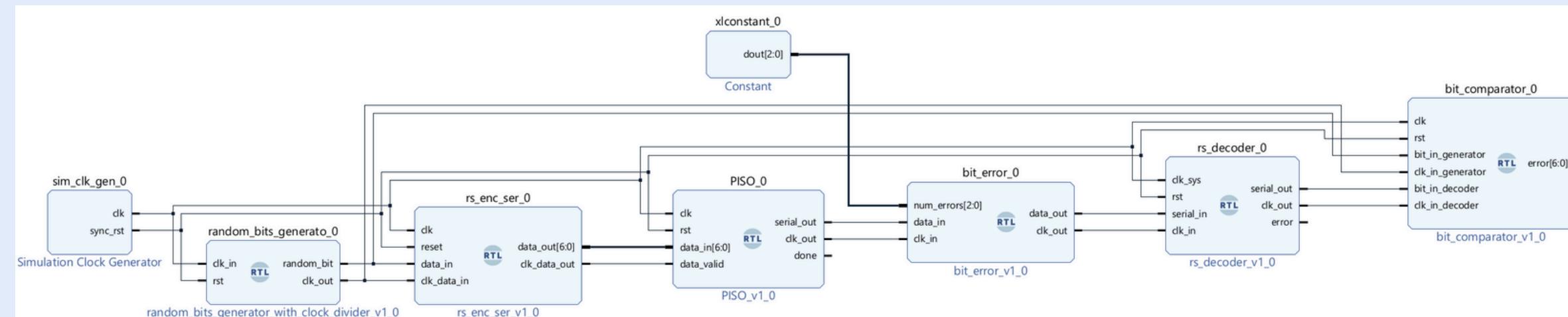
Atualmente, o código Reed-Solomon continua sendo aprimorado para atender as exigências de sistemas modernos como redes 5G, satélites de comunicação e aplicações em IoT e 6G. Sua flexibilidade e eficácia o tornam um elemento-chave na confiabilidade das comunicações digitais de alta performance.

INTRODUÇÃO



Módulos :

- Divisor de Clock;
- Gerador de Bit Aleatório;
- Codificador de Reed Solomon;
- Conversor Serial/Paralelo;
- Canal;
- Conversor Paralelo/Serial,
- Decodificador de Reed Solomon
- Comparador
- Testbench Geral.

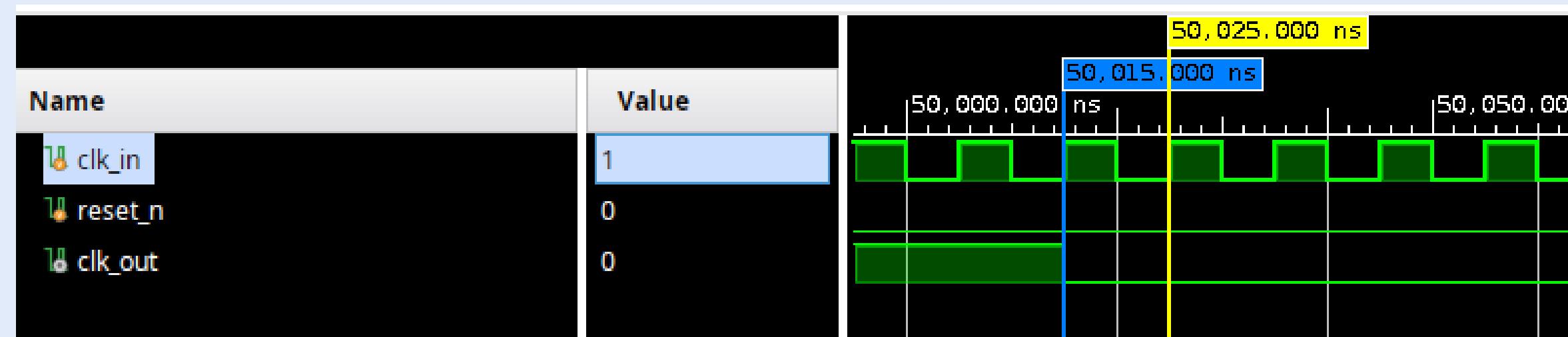


DIVISOR DE CLOCK

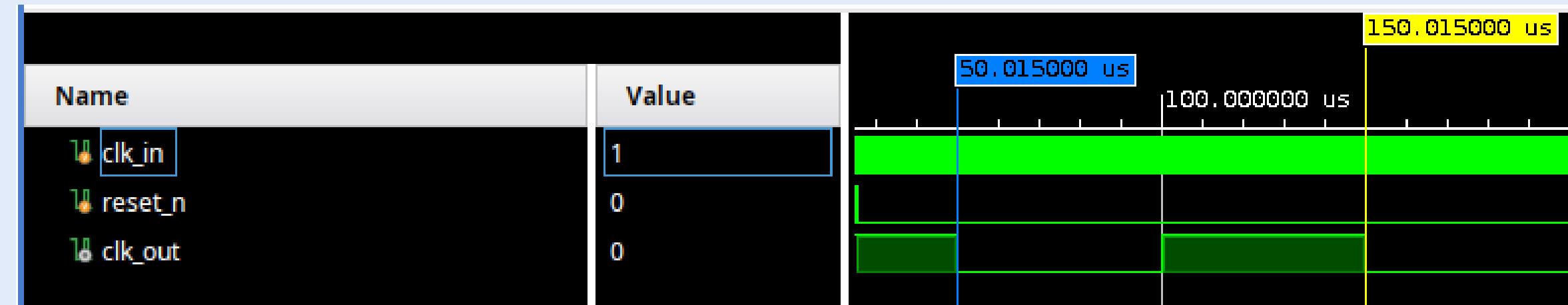
- Bloco digital utilizado para reduzir a frequência de um sinal de clock de entrada (mais rápido), gerando um novo sinal de clock de saída (mais lento)
- A divisão permite que vários blocos funcionem sincronizados com o mesmo clock mestre, mas com velocidades diferentes

DIVISOR DE CLOCK

- 100 Mhz



- 10 Khz



GERADOR DE BIT ALEATÓRIO

- Um algoritmo FEC (Reed Solomon) tem por objetivo a transmissão de dados correta, por meio da capacidade de detecção e recuperação de dados.
- Assim para implementar o algoritmo, é preciso de uma fonte geradora de dados aleatórios.

GERADOR DE BIT ALEATÓRIO

- 1^a rodada de implementação
 - Buffer 32 bits como LFSR
 - Foi usado o método \$random disponível no verilog
- 2^a rodada de implementação
 - Buffer 32 bits como LFSR
 - SEED inicial
 - Embaralhamento com shift e xor

GERADOR DE BIT ALEATÓRIO

- 3^a rodada de implementação
 - Buffer 32 bits como LFSR
 - Foi usado o método “middle squared”

1. Eleva o número ao quadrado
2. Extrai os dígitos centrais do resultado
3. Usa esses dígitos como a nova semente

Iteração	Seed	Quadrado	Zero-padded	Meio (4 dígitos)
1	6759	45683781	0045683781	6837
2	6837	46744209	0046744209	7442
3	7442	553827364	0553827364	8273
4	8273	684418529	0684418529	4185
5	4185	175210225	0175210225	2102

CODIFICADOR DE REED SOLOMON

- O codificador recebe uma sequencia de bits e a codifica para transmissão, adicionando bits de correção (ECC).
- O código escolhido foi o RS(7, 3), significando que possui 7 símbolos no total ($N=7$) sendo que 3 são realmente dados ($K=3$). Cada símbolo é composto por único bit para simplificar a implementação.

CODIFICADOR DE REED SOLOMON

- O processo de codificação segue a seguinte fórmula:

$$h(x) = \frac{m(x)^{N-K}}{g(x)} + m(x)$$

$h(x)$ • Mensagem codificada

$m(x)$ • Dados da mensagem

$g(x)$ • Polinômio gerador, foi escolhido usar $g(x) = x^4 + x + 1$

CODIFICADOR DE REED SOLOMON

- Por exemplo, para uma mensagem 0b101, o procedimento seria:

$$h(x) = \frac{(x^2 + 1)^4}{x^4 + x + 1} + (x^2 + 1)$$

$$h(x) = x^6 + x^5 + x^4 + x^3 + x^2 + 1$$

- E convertendo para bits: 0b1111101

CODIFICADOR DE REED SOLOMON

- Quanto ao código:

A parte mais complexa é a divisão polinomial, que necessita de um algoritmo iterativo.

Para aumentar a capacidade do sistema, todo o módulo funciona sobre uma pipeline

```
module poly_div_pip(  
    input wire          clk,  
    input wire          reset,  
    input wire [6:0]    dividend,  
    input wire [6:0]    divisor,  
    output reg [3:0]   remainder  
);  
    /* ... */  
endmodule
```

CODIFICADOR DE REED SOLOMON

A pipeline possui 3 estágios, pois

este é o tamanho do dado de
entrada, logo a latência do sistema é
de 3 ciclos.

Como a divisão é feita no GF(2), cada
etapa é simplesmente um XOR.

```
always @(posedge clk, posedge reset) begin
    /* ... */

    // step 1
    divisor_buf[0] <= divisor;
    if (dividend[6])
        dividend_buf[0] <= dividend ^ (divisor << 2);
    else
        dividend_buf[0] <= dividend;

    // step 2
    divisor_buf[1] <= divisor_buf[0];
    if (dividend_buf[0][5])
        dividend_buf[1] <= dividend_buf[0] ^ (divisor_buf[0] << 1);
    else
        dividend_buf[1] <= dividend_buf[0];

    // step 3
    if (dividend_buf[1][4])
        remainder <= dividend_buf[1] ^ divisor_buf[1];
    else
        remainder <= dividend_buf[1];
end
```

CODIFICADOR DE REED SOLOMON

Para facilitar a integração com o

resto do sistema o encoder recebe
em sua entrada os bits individuais.

Embora a saída do módulo seja
paralela, esta já é convertida para
serial imediatamente.

```
module rs_enc#(
    parameter N = 7,
    parameter K = 3
)(
    input wire          clk,
    input wire          reset,
    input wire [K-1:0]  data_in,
    output reg [N-1:0] data_out
);
    /* ... */
endmodule
```

CODIFICADOR DE REED SOLOMON

Vários buffers são utilizados para se determinar os pontos de transição dos clocks e armazenar os dados para execução.

```
reg [K-1:0] msg_in;
reg [1:0] msg_in_count;

reg [1:0] clk_data_in_buf;
wire clk_data_in_posedge;
assign clk_data_in_posedge = clk_data_in_buf[0] & ~clk_data_in_buf[1];

reg rs_enc_clk;
assign clk_data_out = rs_enc_clk;

rs_enc#(N, K) rs_enc(rs_enc_clk, reset, msg_in, data_out);
```

CODIFICADOR DE REED SOLOMON

Na entrada serial, cada bit é acumulado antes de o módulo interno (paralelo-paralelo) do encoder ser ativado.

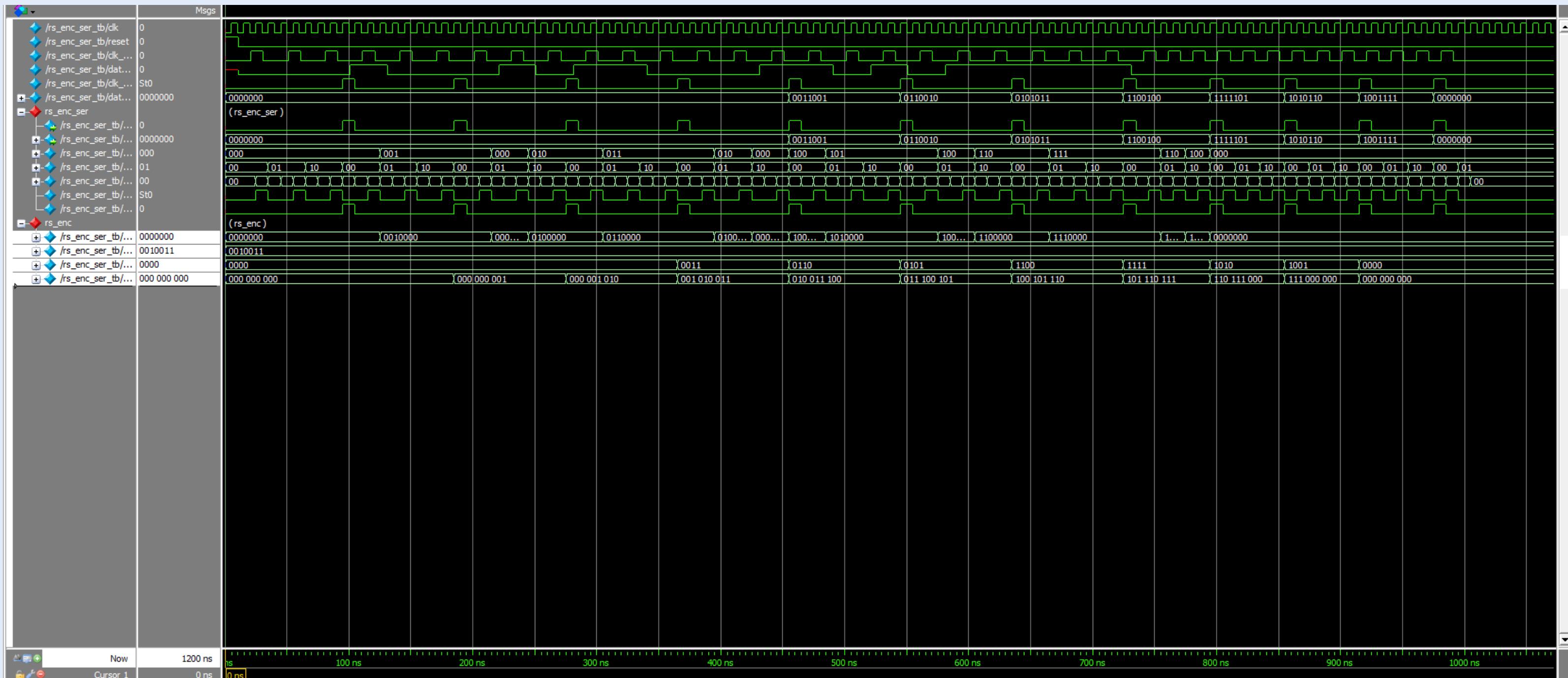
```
always @(posedge clk, posedge reset) begin
    /* ... */

    clk_data_in_buf[0] <= clk_data_in;
    clk_data_in_buf[1] <= clk_data_in_buf[0];

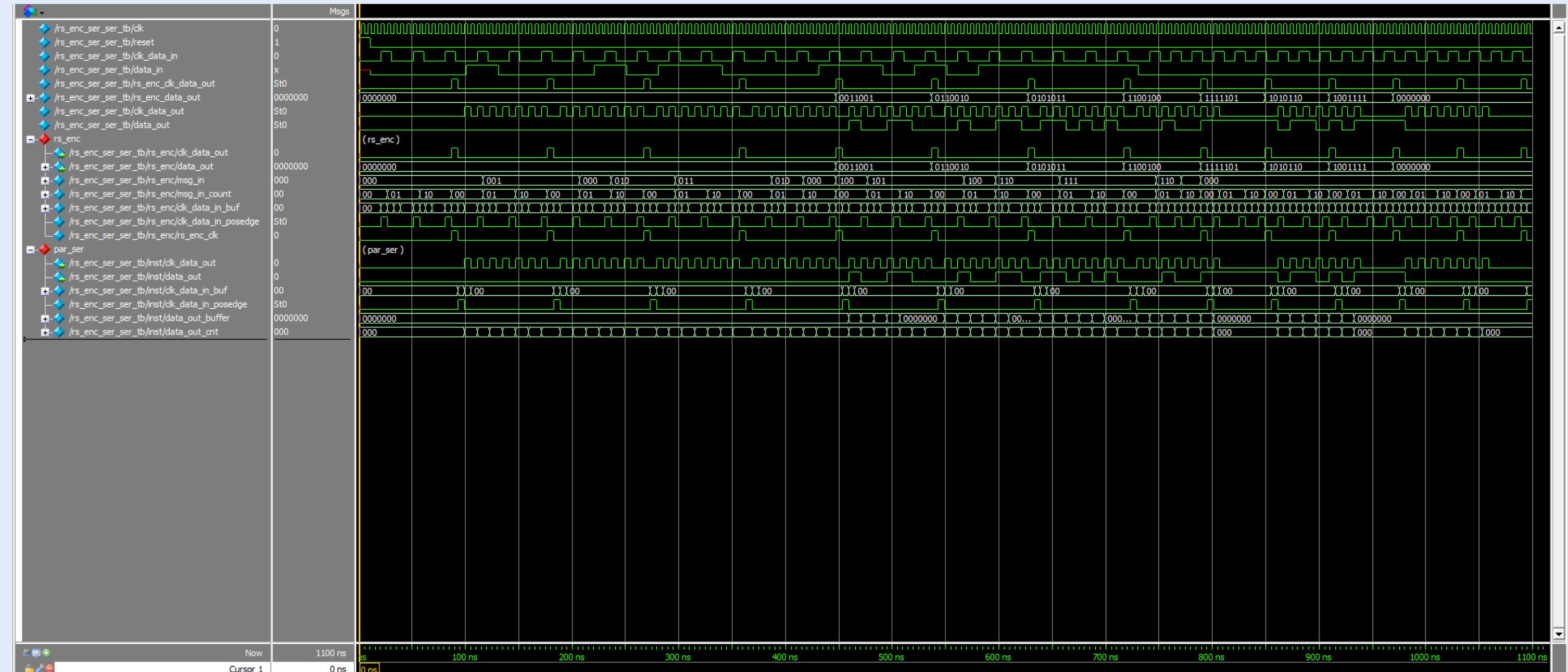
    if (clk_data_in_posedge) begin
        case (msg_in_count)
            2'b00: begin
                msg_in_count <= msg_in_count + 1;
                msg_in[0] <= data_in;
            end
            2'b01: begin
                msg_in_count <= msg_in_count + 1;
                msg_in[1] <= data_in;
            end
            2'b10: begin
                msg_in_count <= 0;
                msg_in[2] <= data_in;

                rs_enc_clk <= 1'b1;
            end
        endcase
    end
```

CODIFICADOR DE REED-SOLOMON



CODIFICADOR DE REED SOLOMON



CANAL

O módulo de canal é fundamental em simulações e testes de sistemas de transmissão e correção de erros. Para este projeto o módulo deveria atender aos requisitos;

- Entrada de dados serial
- Ser controlável
- Possuir Alguma aleatoriedade
- Operar com fluxo continuo

CANAL

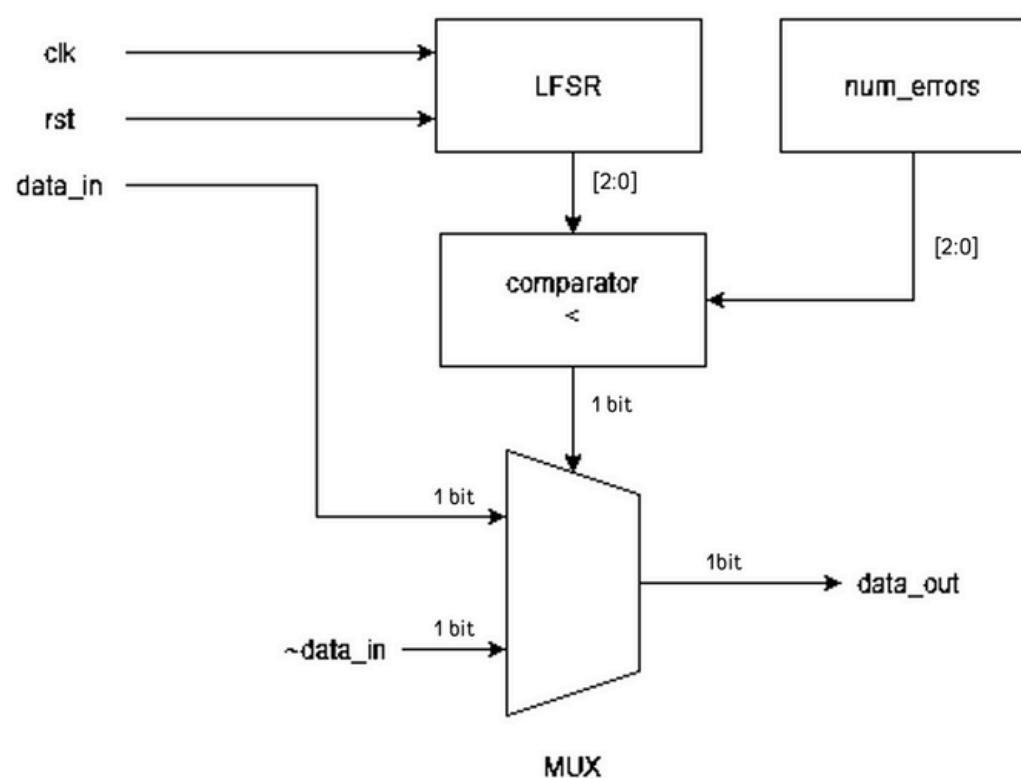
Para atender os requisitos foi proposto e projetado um gerador de dados pseudo aleatorios do tipo Linear Feedback Shift Register (LFSR) .

O LFSR tem 3 bits (lfsr[2:0]) e polinômio gerador $P(x)=x^3+x^2+1$. O feedback é calculado por uma operação XOR entre o bit mais significativo (lfsr[2]) e o segundo bit (lfsr[1]).

A cada pulso de clock, os bits são deslocados para a esquerda (lfsr[1:0] desloca para lfsr[2:1]), e o bit de feedback é inserido em lfsr[0]

CANAL

Diagrama funcional



Exemplo (com $\text{num_errors} = 3$):

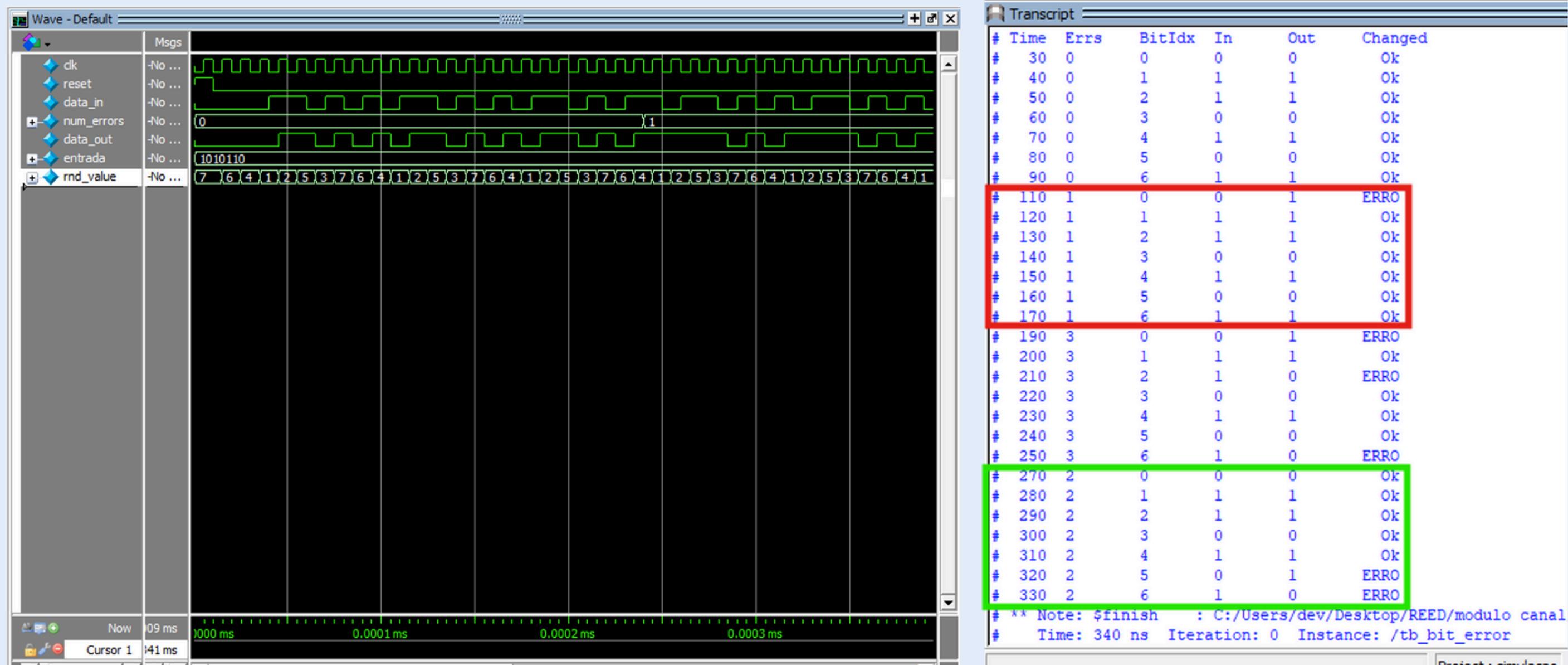
Suponha uma sequência de entrada:
 $\text{data_in}: 0110101\dots$

Para cada bit dessa sequência, o LFSR vai gerar algo como:
 $\text{rnd_value}: \begin{matrix} 2 \\ 5 \\ 3 \\ 7 \\ 6 \\ 4 \\ 1 \end{matrix}$
 $\text{data_out}: \dots 0 \ 0 \ 0 \ 0 \ 1 \ 1 \ 1$

Os valores ≤ 3 causam erro: 3, 2, 1.
Então a inversão vai ocorrer nos bits 1, 5, e 7 .

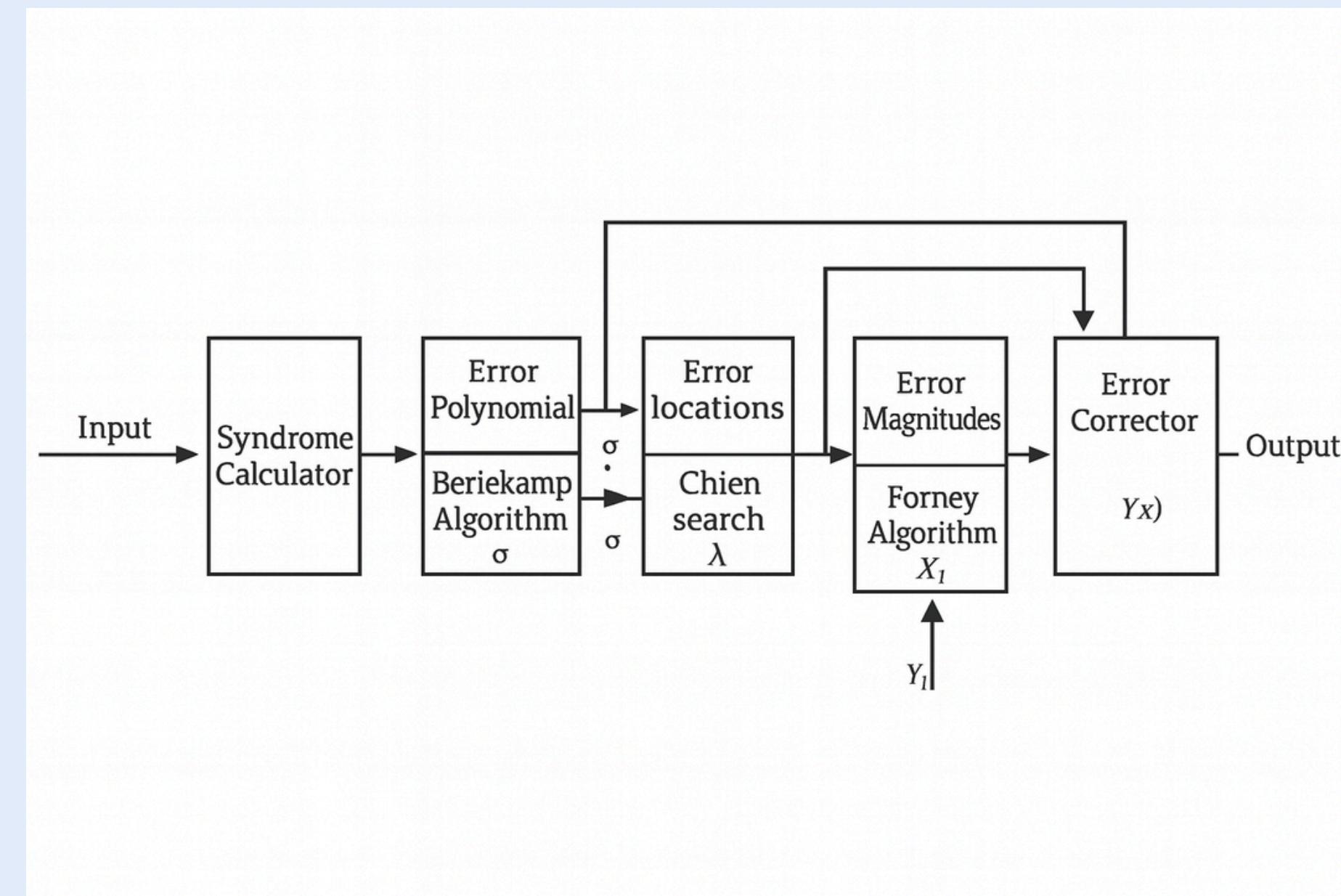
CANAL

Resultados



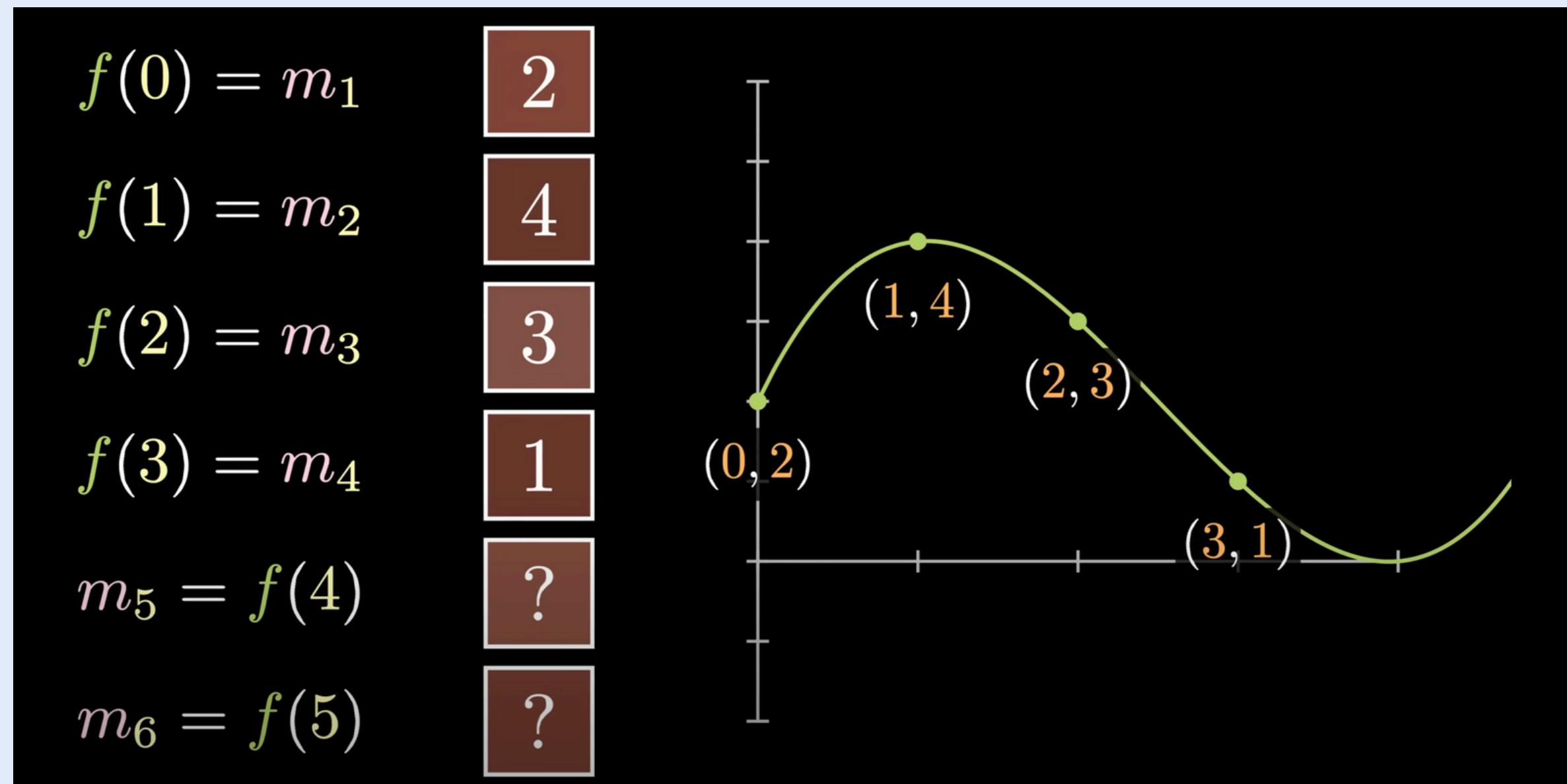
DECODIFICADOR DE REED SOLOMON

- Funcionamento



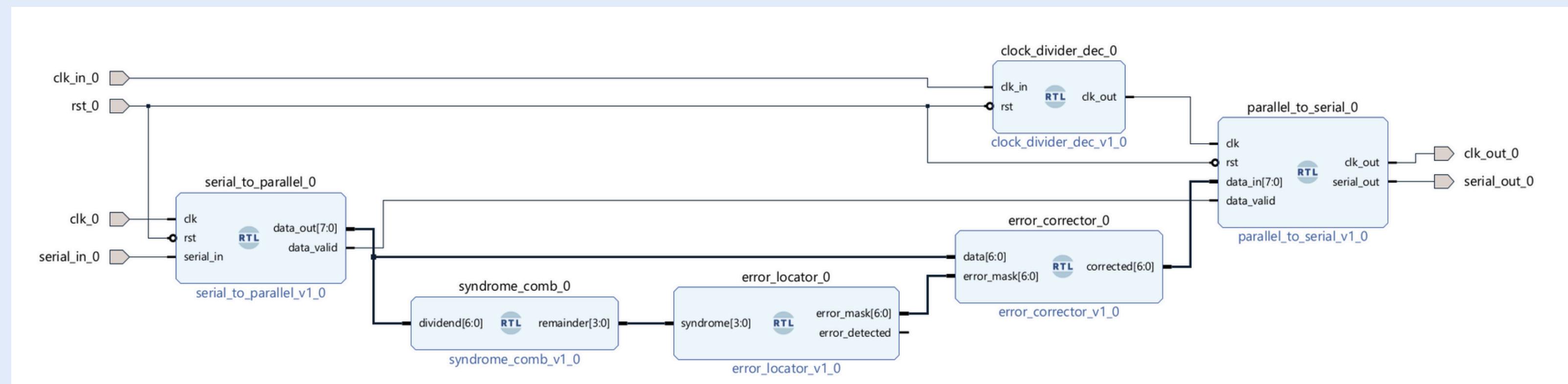
DECODIFICADOR DE REED SOLOMON

- Funcionamento



DECODIFICADOR DE REED SOLOMON

- Estratégia de Implementação



DECODIFICADOR DE REED SOLOMON

- Código

```

1 module syndrome_cyclic_7_3 (
2     input      clk,
3     input      rst,
4     input [6:0] r,          // palavra recebida: 7 bits (MSB=r[6])
5     output reg [3:0] syndrome // resto da divisão (grau 3 ou menos)
6 );
7 // g(x) = x^4 + x + 1 → 5 bits: 1 0 0 1 1
8 parameter [4:0] G = 5'b10011;
9
10 reg [7:0] dividend;
11 integer i;
12
13 always @(posedge clk or posedge rst) begin
14     if (rst) begin
15         syndrome <= 4'b0000;
16     end else begin
17         // Alinhar dividend com os bits recebidos
18         dividend = {r, 1'b0}; // espaço para divisão (grau 7 com 1 zero extra)
19         for (i = 7; i >= 4; i = i - 1) begin
20             if (dividend[i] == 1'b1)
21                 dividend[i -: 5] = dividend[i -: 5] ^ G;
22             end
23             // Resto está nos 4 bits inferiores
24             syndrome <= dividend[3:0];
25         end
26     end
27 endmodule

```

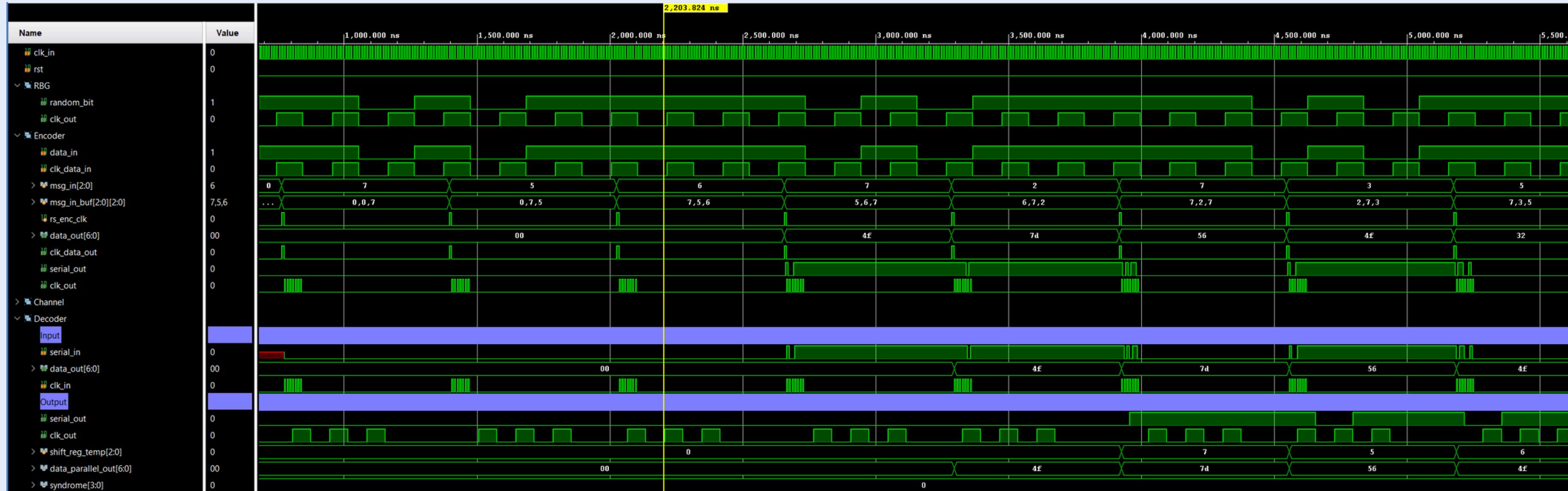
```

30 // module error_locator (
31     input      [3:0] syndrome,
32     output reg [6:0] error_mask,
33     output reg      error_detected
34 );
35
36 always @(*) begin
37     case (syndrome)
38         4'b0000: begin
39             error_mask    = 7'b0000000;
40             error_detected = 0;
41         end
42         4'b0001: begin error_mask = 7'b0000001; error_detected = 1; end // erro em bit 0
43         4'b0010: begin error_mask = 7'b0000010; error_detected = 1; end // bit 1
44         4'b0011: begin error_mask = 7'b0000100; error_detected = 1; end // bit 2
45         4'b0100: begin error_mask = 7'b0001000; error_detected = 1; end // bit 3
46         4'b0101: begin error_mask = 7'b0010000; error_detected = 1; end // bit 4
47         4'b0110: begin error_mask = 7'b0100000; error_detected = 1; end // bit 5
48         4'b0111: begin error_mask = 7'b1000000; error_detected = 1; end // bit 6
49
50         // Síndromes de erro duplo ou inválidas
51     default: begin
52         error_mask    = 7'b0000000; // não tenta corrigir
53         error_detected = 1;           // mas sinaliza erro detectado
54     end
55 endcase
56 end
57 endmodule

```

DECODIFICADOR DE REED SOLOMON

- Simulação



COMPARADOR

- Tem por objetivo validar a implementação do módulo decodificador, através da comparação entre os dados do gerador de bits aleatórios e a saída do decodificador
- Não é uma implementação comum num sistema de transmissão e recepção de dados, mas no presente caso é usado para concatenar o funcionamento do trabalho

COMPARADOR

- Estabelecer a amostragem de cada dado sob uma base de tempo
- Sincronizar informações para comparação
- Entender os limites nas operações de comparação
- Representar o erro medido numa base comum

COMPARADOR

Primeira implementação

```
module bit_comparator #(  
    parameter DELAY  
)  
(  
    input reset,  
    clk_in_decoder,  
    bit_in_generator,  
    bit_in_decoder,  
    output [6:0]error  
,  
);
```

```
bit_in_generator_delay = bit_in_generator_delay << 1;  
bit_in_generator_delay [0] = bit_in_generator;  
  
if(count_delay < DELAY)  
begin  
    count_delay <= count_delay + 1;  
end
```

COMPARADOR

```
else
begin
    {bit_error_out, vet_error} = vet_error << 1;

    if((bit_in_generator_delay[DELAY]) ^~ (bit_in_decoder) == 1'b1)
begin
    vet_error[0] = 1'b0;

    if(bit_error_out)
begin
    count_error = count_error - 1;
end
end

else
begin
    vet_error[0] = 1'b1;

    if(!bit_error_out)
begin
    count_error = count_error + 1;
end
end
```

COMPARADOR

```
if(count_bit < 100)
begin
    count_bit = count_bit + 1;
    mult_div = (PORCENTAGE * count_error)/count_bit;
    percentage_error = mult_div;
end

else
begin
    mult_div = 0;
    percentage_error = count_error;
end

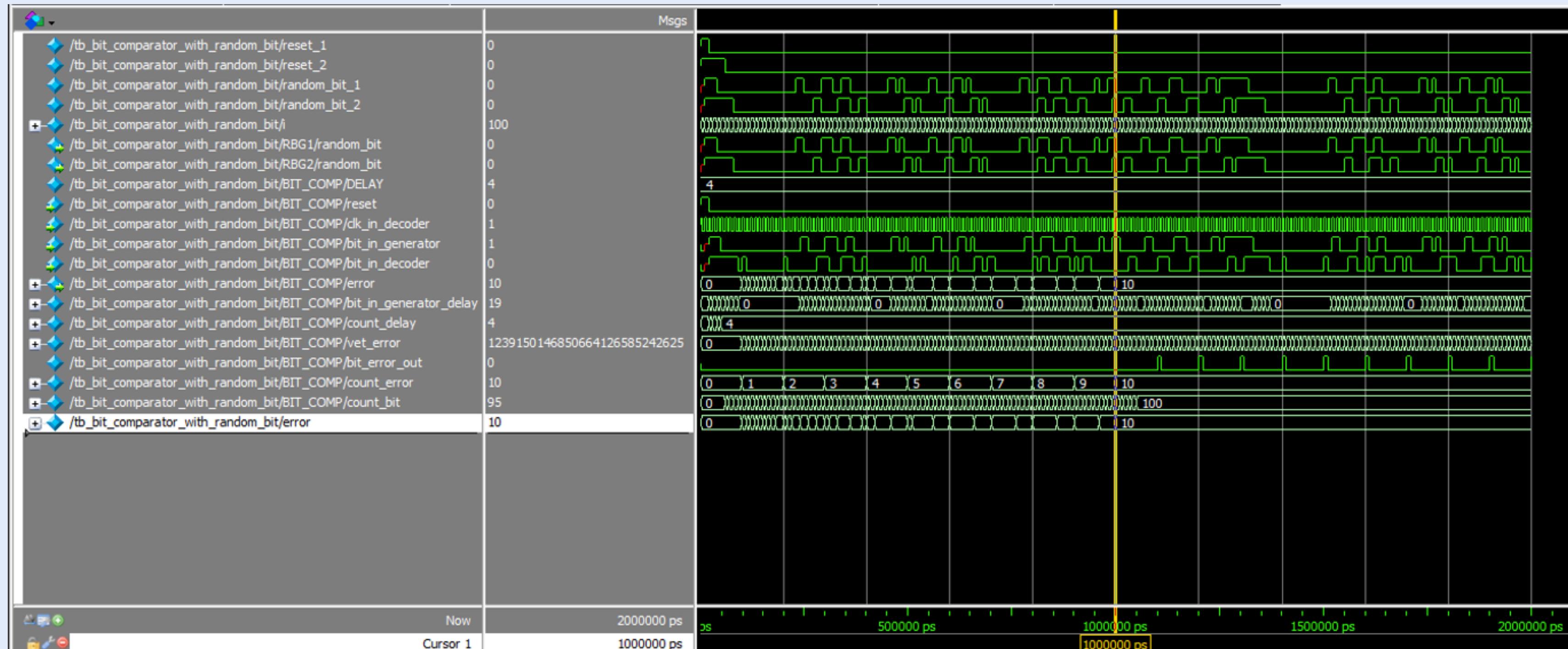
end
end

assign error = percentage_error;
```

COMPARADOR

```
for(i=0;i<1000;i=i+1)
begin
    #10;
    bit_in_generator = random_bit_1;
    if(i%10>8)
        bit_in_decoder = ~random_bit_2;
    else
        bit_in_decoder = random_bit_2;
end
end
```

COMPARADOR



COMPARADOR

Implementação final

```
module bit_comparator (
    input rst,
    clk,
    clk_in_generator,
    clk_in_decoder,
    bit_in_generator,
    bit_in_decoder,
    output [6:0]error
);
```

```
last_clk_generator = new_clk_generator;
new_clk_generator = clk_in_generator;
last_clk_decoder = new_clk_decoder;
new_clk_decoder = clk_in_decoder;

rising_edge_generator = ~last_clk_generator && new_clk_generator;
rising_edge_decoder = ~last_clk_decoder && new_clk_decoder;
```

COMPARADOR

```
always @(posedge rising_edge_generator or posedge rising_edge_decoder)
begin

    last_fifo_is_full = fifo_is_full;

    if(rising_edge_generator)
    begin
        if(elements_fifo == 127)
            fifo_is_full = 1;
        else
            begin
                bit_in_generator_fifo[w_ptr] = bit_in_generator;
                w_ptr = w_ptr + 1;
                fifo_is_full = 0;
                elements_fifo = elements_fifo + 1;
            end
    end
end
```

COMPARADOR

```
always @(posedge rising_edge_generator or posedge rising_edge_decoder)
begin

    last_fifo_is_full = fifo_is_full;

    if(rising_edge_generator)
    begin
        if(elements_fifo == 127)
            fifo_is_full = 1;
        else
            begin
                bit_in_generator_fifo[w_ptr] = bit_in_generator;
                w_ptr = w_ptr + 1;
                fifo_is_full = 0;
                elements_fifo = elements_fifo + 1;
            end
    end
end
```

COMPARADOR

```
if ((rising_edge_decoder) && (count_dequeue < 127))
begin

    if(fifo_is_full)
    begin
        count_dequeue = count_dequeue + 1;
    end

    bit_in_generator_fifo_out = bit_in_generator_fifo[r_ptr];
    r_ptr = r_ptr + 1;

    if(~fifo_is_full)
    begin
        elements_fifo = elements_fifo - 1;
    end
```

COMPARADOR

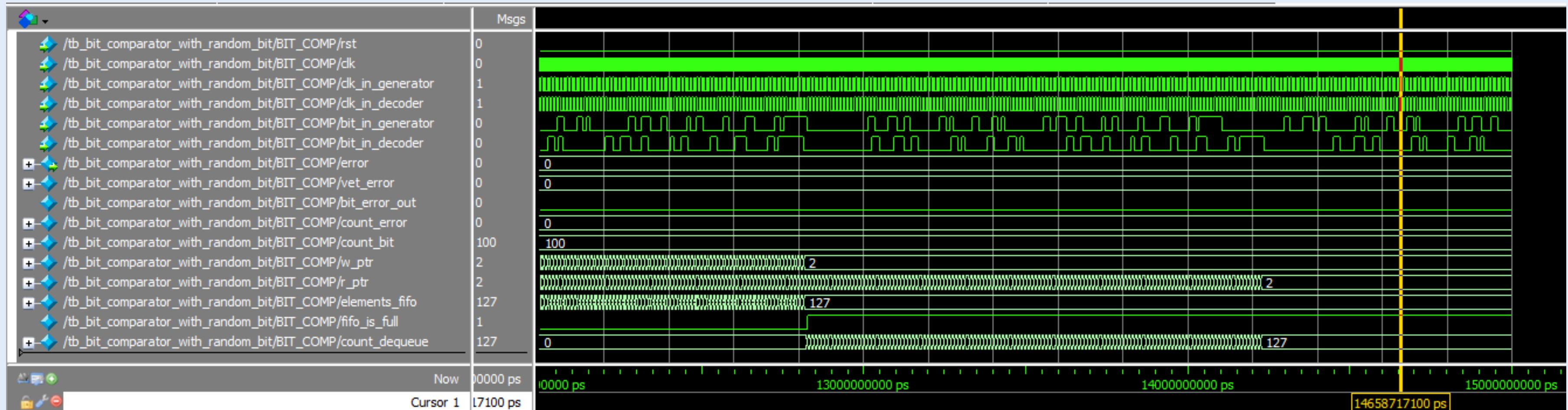
```
localparam SEED = 128'hA5A5A5A5A5A5A5A5A5A5A5A5A5A5A5A5;

reg rst;
reg clk;
wire clk_in_generator;
wire clk_in_decoder;
wire bit_in_generator, bit_in_decoder;
wire [6:0] error;

clock_divider_non_synth #(INPUT_FREQ(10000000), OUTPUT_FREQ(100000)) CLK_GENERATOR (.clk_in(clk), .rst(rst), .clk_out(clk_in_generator));
clock_divider_non_synth #(INPUT_FREQ(10000000), OUTPUT_FREQ(90000)) CLK_DECODER (.clk_in(clk), .rst(rst), .clk_out(clk_in_decoder));
random_bits_generator #(SEED(SEED)) RBG1(.clk_in(clk_in_generator), .rst(rst), .random_bit(bit_in_generator));
random_bits_generator #(SEED(SEED)) RBG2(.clk_in(clk_in_decoder), .rst(rst), .random_bit(bit_in_decoder));

bit_comparator BIT_COMP (.rst(rst), .clk(clk), .clk_in_generator(clk_in_generator), .clk_in_decoder(clk_in_decoder),
| | | | | | | | | | .bit_in_generator(bit_in_generator), .bit_in_decoder(bit_in_decoder), .error(error));
```

COMPARADOR



TESTBENCH GERAL

No início do projeto, foi proposto um testbench simplificado com os seguintes objetivos:

- Validar o funcionamento do SIPO, PISO, Canal e Gerador de pulsos até o codificador e decodificador Reed-Solomon ficarem prontos

TESTBENCH GERAL

```

// Clock de 100 MHz
clock_generator #(FREQUENCY, 0) CLOCK100M(
    .clk(clk_100MHz)
);

//Divisor de clock para 50MHz
clock_divider #(.N(2)) CLOCK50M(
    .clk(clk_100MHz),
    .rst(rst),
    .clk_out(clk_50MHz)
);

//Modulo de Geração de Dados Aleatórios
signal_generation #(.N(8)) SIGNAL (
    .clk(clk_50MHz),
    .rst(rst),
    .data_out(signal)
);

/*
//Modulo Reed-Solomon Encoder
reed_solomon_encoder #(.N(8)) RS_ENCODER (
    .clk(clk_50MHz),
    .rst(rst),
    .data_in(),
    .clk_out(),
    .data_out()
);
*/

//Modulo PISO
PISO #(.N(8)) PISO_UPGRADE (
    .clk(clk_50MHz),
    .rst(rst),
    .load(load_piso),
    .shift_en(shift_en_piso),
    .data_in(signal),
    .data_out(data_out_piso)
);

/*
//Modulo Simulação de Canal com perda controlada
controlled_loss_channel #(.N(8)) LOSS_CHANNEL (
    .clk(clk_50MHz),
    .rst(rst),
    .data_in(data_out_piso),
    .data_out(data_out_channel)
);

//Modulo SIPO
SIPO #(.N(8)) SIPO_UPGRADE (
    .clk(clk_50MHz),
    .rst(rst),
    .shift_en(shift_en_sipo),
    .data_in(data_out_channel),
    .data_out(data_out_sipo)
);

/*
//Modulo Reed-Solomon Decoder
reed_solomon_decoder #(.N(8)) RS_DECODER (
    .clk(clk_50MHz),
    .rst(rst),
    .data_in(),
    .clk_out(),
    .data_out()
);

*/

//Modulo Comparador de Sequencias de Atraso
delayed_sequence_comparator #(.N(8), .DELAY_CYCLES(9)) COMPARATOR (
    .clk(clk_50MHz),
    .rst(rst),
    .data_in(signal),
    .data_received(data_out_sipo),
    .data_match(data_match)
);

```

TESTBENCH GERAL

- Realizar a recepção e saída dos dados e comparar com a sequência original.
- Clocks sincronizados entre os módulos usando o mesmo divisor de clock.
- A comparação de dados era feita de forma simples, diretamente entre signal e data_out_sipo.

TESTBENCH GERAL

```

// Geração do clock de 100 MHz
clock_generator #(.FREQ_HZ(INPUT_FREQ), .START_POLARITY(0)) clockGenerator (
    .clk(clk)           // Saída do gerador de clock
);

// Instância do módulo Gerador de sinal aleatório
random_bits_generator_with_clock_divider #(.USE_PATTERN(8'b11110000), .DIVISOR_FREQ(DIVISOR_FREQ)) uut (
    .clk_in(clk),        // Sinal de clock
    .reset(rst),         // Sinal de reset
    .random_bit(bit_random_out), // bit aleatório
    .clk_out(clk_random_bit_out) // Sinal de clock de saída
);

// Instância do modulo RS Encoder
RS_encoder rs_enc (
    .clk(clk),           // Sinal de clock
    .rst(rst),           // Sinal de reset
    .clk_in_serial(clk), // Clock de dados de entrada
    .serial_in(bit_random_out), // Entrada bit a bit do sinal
    .codeword(data_rs_enc_out), // palavra de 7 bits
    .data_valid(clk_rs_enc_out) // vai para 1 quando codeword estiver pronto
);

// Instância do modulo PISO p/ mandar para o canal
PISO #(.N(N)) p2s_inst1 (
    .clk(clk),           // Sinal de clock
    .rst(rst),           // Sinal de reset
    .data_in(data_rs_enc_out), // Entrada de 7 bits
    .data_valid(clk_rs_enc_out), // Sinalização p/ entrada
    .clk_out(clk_serial_encoder_out), // Sinal de clock de saída
    .serial_out(serial_encoder_out), // Saída do bit
    .done(done_serial_encoder_out) // vai para 1 quando PISO estiver pronto
);

// Instância do módulo Gerador de bit com erro no canal
bit_error_channel(
    .clk(clk),           // Sinal de clock
    .num_errors(num_errors), // Quantidade de bits com erro
    .data_in(serial_encoder_out), // Dados de entrada (um bit por vez)
    .data_out(data_channel_out) // Dados de saída (com ou sem erro injetado)
);

// Instância do modulo SISO para mandar dados paralelos ao RS Decoder
SISO #(.N(N)) s2p_inst (
    .clk(clk),           // Sinal de clock
    .rst(rst),           // Sinal de reset
    .enable(done_serial_encoder_out), // Sinalização p/ entrada
    .serial_in(data_channel_out), // Entrada bit a bit
    .data_out(data_sipo_out), // Saída do dado 7 bits
    .data_valid(data_sipo_valid_out) // Sinalização de saída
);

// Instância do modulo RS Descoder
RS_decoder rs_dec (
    .clk(clk),           // Sinal de clock
    .rst(rst),           // Sinal de reset
    .received(data_sipo_out), // Entrada de 7 bits
    .corrected(corrected), // Sinal corrigido
    .message(message), // 3 bits de mensagem
    .syndrome(syndrome), // Syndrome 000 quando não tiver erro, caso contrario tem erro
    .error_corrected(error_corrected), // Erro corrigido
    .clk_out(clk_decoder_out) // Sinal de clock de saída
);

// Instância do modulo PISO para comparar os bits com o comparador no final
PISO #(.N(K)) p2s_inst2 (
    .clk(clk),           // Sinal de clock
    .rst(rst),           // Sinal de reset
    .data_in(message), // Entrada dos 7 bits
    .data_valid(clk_rs_enc_out), // Sinalização p/ entrada
    .clk_out(clk_piso_out), // Sinal de clock de saída
    .serial_out(serial_decoder_out), // Saída do bit
    .done(done_serial_decoder_out) // vai para 1 quando PISO estiver pronto
);

// Instância do comparador de bits
bit_comparator BIT_COMP(
    .rst(rst),           // Sinal de reset
    .clk(clk),           // Sinal de clock
    .clk_in_generator(clk_random_bit_out), // Clock do sinal de entrada
    .clk_in_decoder(clk_decoder_out), // Clock do RS decoder
    .bit_in_generator(bit_random_out), // Bit do sinal de entrada
    .bit_in_decoder(serial_decoder_out), // Bit do RS decoder
    .error(error_comparator_out) // Saída do erro
);

```

TESTBENCH GERAL

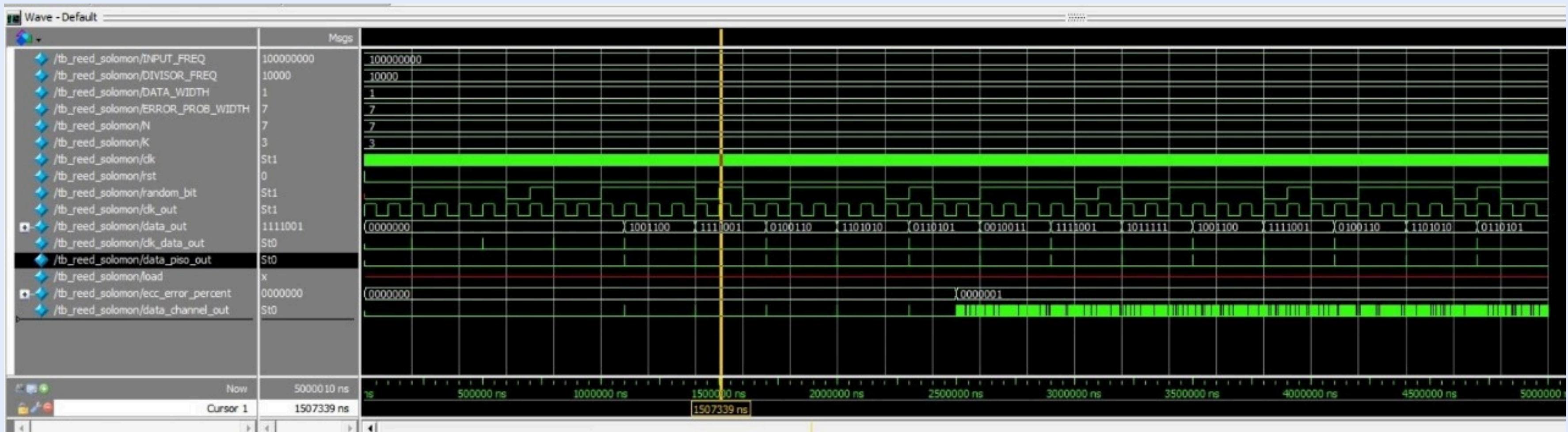
Na versão final, entramos com o codificador e decodificador de fato, porém o clock de sincronismo entre os módulos foi um desafio para sincronizar cada modulo.

TESTBENCH GERAL

Para garantir a comunicação sem perda de bits, foi preciso alinhar também o SIPO e PISO com os demais clocks vindos de cada módulo e adição de sinais de controle (data_valid, done, clk_out).

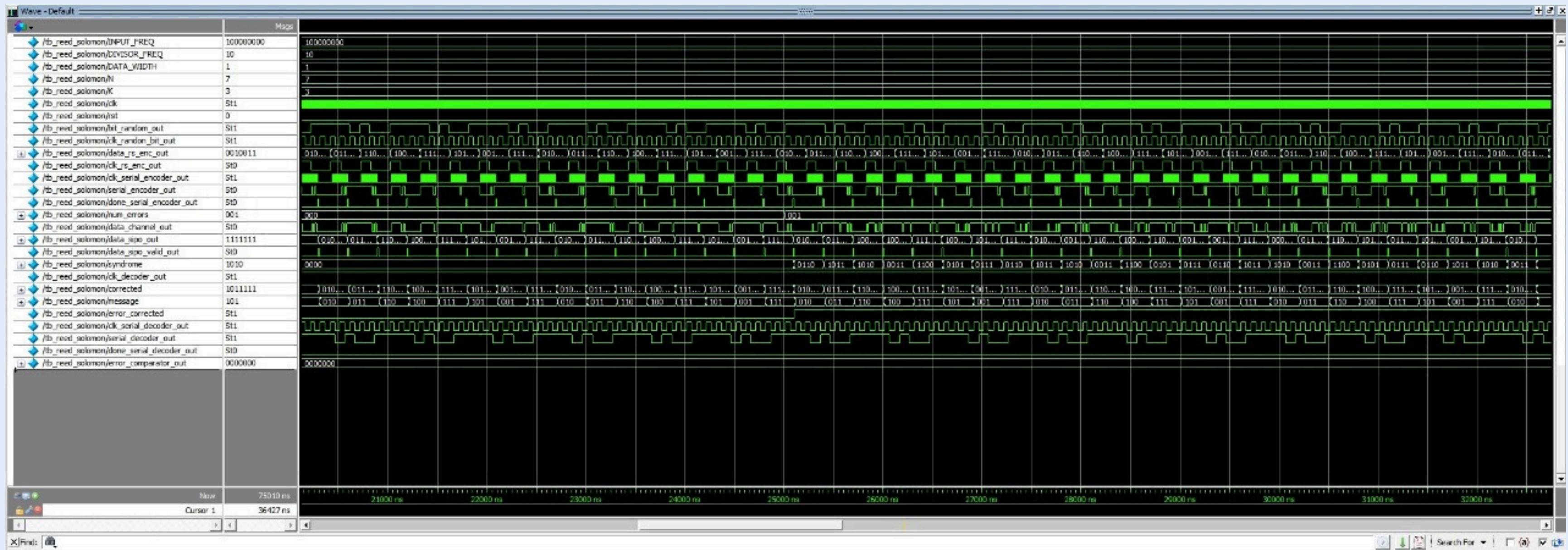
Todos os módulos agora operam com clock vindos de outros anteriores, afim de obter o sincronismo entre eles.

TESTBENCH GERAL



Testbench antes de aplicar sincronismo distintos nos módulos

TESTBENCH GERAL MODELSIM



Testbench final, após a aplicação de sincronismo de clocks

TESTBENCH GERAL

Por que o sincronismo foi essencial?

- Em um sistema onde dados circulam de forma sequencial, qualquer desalinhamento de clock pode fazer com que um módulo leia ou processe dados inválidos ou incompletos.

TESTBENCH GERAL

- Módulos como PISO/SIPO e o canal com erro operam com sinais seriais. Caso não compartilhassem o mesmo domínio de clock, seria necessário implementar sincronizadores extras para evitar falhas intermitentes.

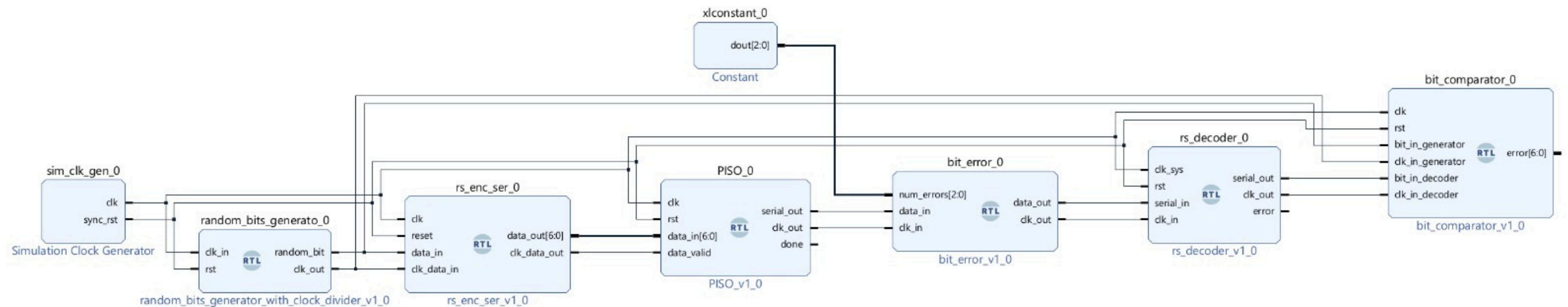
TESTBENCH GERAL

- Esses sinais de controle sincronizados entre os módulos garantiram que o sistema operasse como um pipeline coerente, onde cada etapa só era ativada quando os dados estavam prontos e válidos.

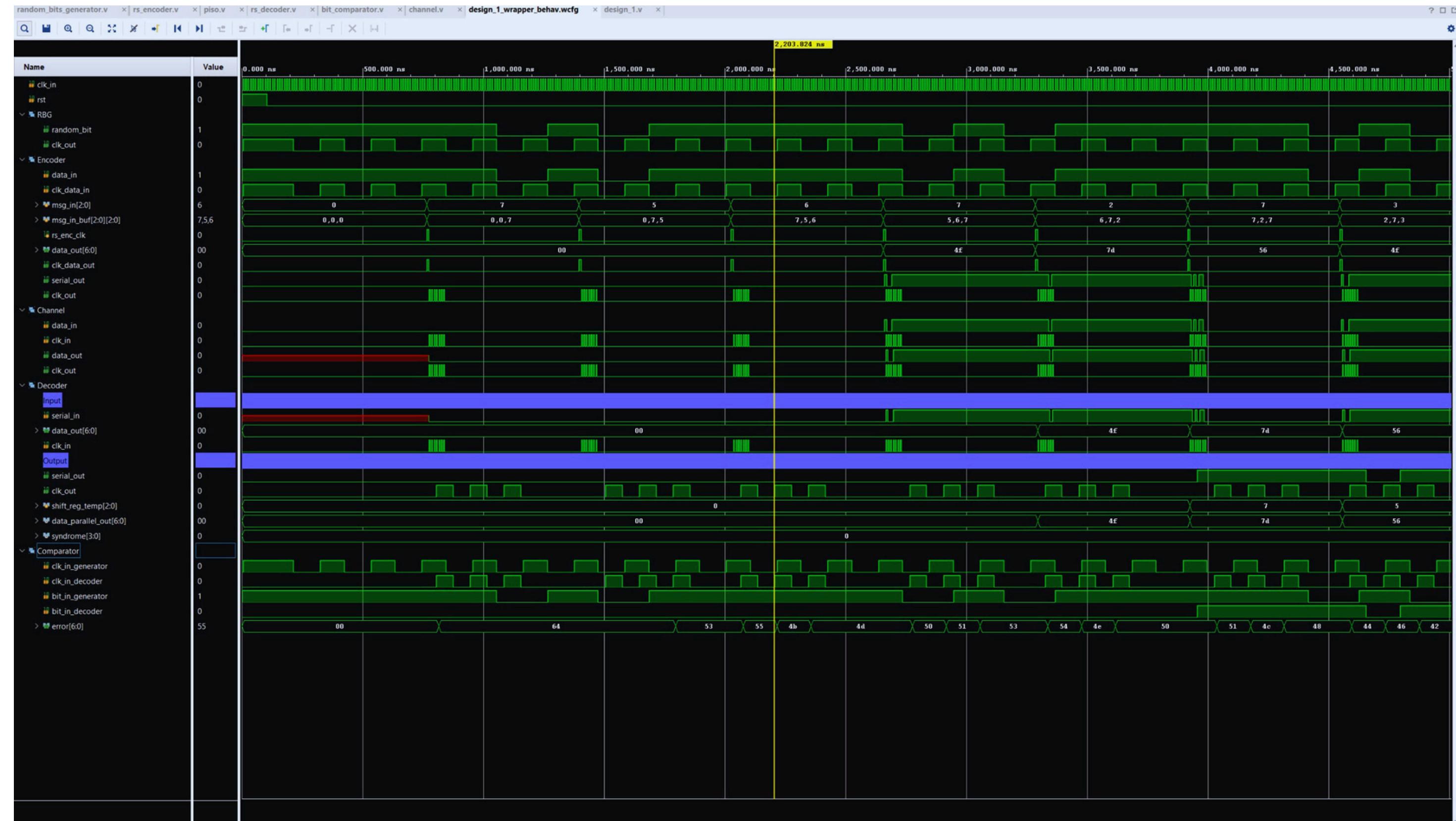
TESTBENCH GERAL

O sincronismo entre os módulos não foi apenas uma boa prática de projeto – ele foi imprescindível para o sucesso da simulação, validação e análise do sistema completo de codificação e decodificação Reed-Solomon com canal de erro.

TESTBENCH GERAL VIVADO



TESTBENCH GERAL VIVADO



TESTBENCH GERAL

Eficiência na prática:

Durante a simulação com injeção controlada de erros, o código de Reed

Solomon demonstrou alta eficácia nos seguintes pontos:

TESTBENCH GERAL

1. Correção de erro único

- Quando um único bit era corrompido durante a transmissão, o síndrome calculado indicava corretamente a posição do erro;
- O bit corrompido foi invertido e o dado original foi recuperado com sucesso;

2. Detecção de 2 erros

- Quando dois bits eram alterados simultaneamente, o sistema detectava que havia erro, mas não conseguia corrigi-lo;
- Isso foi refletido por um síndrome não compatível com erro de um único bit;

Embora o dado não tenha sido corrigido, o erro foi sinalizado corretamente, impedindo o uso de dados errados silenciosamente.

TESTBENCH GERAL

- 3. Zero erro = transmissão perfeita
- Quando não havia erro, os dados passavam pelo sistema sem modificação, confirmando o funcionamento correto do pipeline.

CONCLUSÃO

A implementação do código Reed-Solomon em Verilog foi bem-sucedida, com todos os módulos essenciais (codificador, decodificador, conversores, etc.) funcionando corretamente.

CONCLUSAO

O sistema foi testado via simulação no ModelSim, onde foi possível observar a correção de erros de bits . O decodificador demonstrou a capacidade de recuperar dados originais mesmo em condições de erro, confirmando a eficácia do código Reed-Solomon na recuperação de informações.