UNIVERSIDADE FEDERAL DA BAHIA
ESCOLA POLITÉCNICA
DEPARTAMENTO DE ENGENHARIA ELÉTRICA E DE COMPUTAÇÃO
ENGC40 – ELETRÔNICA DIGITAL

JOÃO PEDRO DA SILVA CERQUEIRA

MIGUEL FELICIANO MOTA ALVES

RODRIGO FREITAS SÁ BARRETTO

CORREÇÃO DE ERROS NA COMUNICAÇÃO SERIAL

Salvador, BA, Brasil

2023

JOÃO PEDRO DA SILVA CERQUEIRA
MIGUEL FELICIANO MOTA ALVES
RODRIGO FREITAS SÁ BARRETTO

CORREÇÃO DE ERROS NA COMUNICAÇÃO SERIAL

Projeto apresentado à matéria de Eletrônica Digital, como requisito de obtenção de nota.
Docente: Tiago Trindade Ribeiro.

SALVADOR
2023

SUMÁRIO

1 INTRODUÇÃO

O envio de mensagens verbais ou não no dia a dia por humanos é uma parte indispensável desde os primórdios da civilização. Com o aumento da complexidade dos problemas tecnológicos, notou-se a necessidade da comunicação entre máquinas também e, posteriormente, surgiu-se a necessidade de criar um protocolo para que isso fosse possível, fazendo com que as informações passadas por um dispositivo fosse compreendida pelo outro, criando-se, assim, as comunicações seriais.

A partir disso, com o aumento na complexidade e da distância nas redes de comunicação, seja por latências, interferências eletromagnéticas, falhas de sincronização, defeitos de componentes, as chances de ocorrência de erros nos bits dessas mensagens se elevaram. Desta forma, foi necessário criar métodos para a detecção e correção de erros, permitindo, assim, que os sistemas funcionem com confiabilidade e eficiência.

2 OBJETIVO

Esse trabalho tem como fim a construção de um projeto que receba uma mensagem como entrada e, mesmo que ocorra os erros supracitados, envie a mensagem original utilizando a comunicação serial.

3 REVISÃO BIBLIOGRÁFICA

Para o entendimento do funcionamento do código Hamming, o artigo da Vera Pless juntamente com o artigo da USP foram essenciais. Nele, o passo a passo para a codificação e decodificação da mensagem são muito claros. Entretanto, a implementação dele em uma linguagem de descrição de hardware não está descrita nesses trabalhos. Portanto - para ter um aprofundamento nessa área - foi utilizado o livro do Peter Minns et al., o qual tem uma abordagem mais didática na construção do código em si.

4 SOLUÇÃO E AVALIAÇÃO PROPOSTA

4.1 SOLUÇÃO IMPLEMENTADA

Para a solução dos problemas supracitados, as seguintes soluções foram implementadas.

- Código Hamming:

Esse método baseia-se em introduzir bits de paridade (baseados no método de paridade par ou ímpar) em meio aos bits de informação e realizar operações para identificar e, caso exista algum bit errado, corrigi-lo. Esse método foi implementado por meio de operações xor e utilizando o Verilog, o qual possibilitou a criação do código de forma mais rápida e eficiente, uma vez que se torna extremamente complicado se utilizar diretamente de portas or, xor, and... para criar todas as funcionalidades desejadas.

- Interface de comunicação serial:

Essa funcionalidade foi implementada criando-se 2 (dois) canais para receber a mensagem, sendo o usuário responsável por escolhê-lo. Dito isso, a solução escolhida para correção de erros foi considerada ideal para o projeto, uma vez que ela funciona bem com o tipo de comunicação escolhida.

- Criação de erro:

Essa funcionalidade foi pensada no intuito de mostrar o funcionamento pleno do código hamming, visto que sua função se dá na correção de um bit que foi, de alguma forma, alterado durante o percurso da comunicação. Tendo isso em vista, a equipe pensou e conseguiu aplicar uma variação de um bit do código hamming de posição aleatória, que simulasse, então, um possível erro de comunicação. Para isso, utilizamos de um somador de clock que, ao alterar o seu valor de maneira extremamente rápida a partir de cada subida de clock e, então, ser impossível de ser acompanhado por um humano, tem, como resultado, uma posição imprevisível, ou, em outras palavras, aleatória. Assim, passamos a acompanhar, também, toda a correção dos bits errôneos, tornando o projeto mais visual e didático.

4.1.1 Implementação em Código HDL

Abaixo estará descrito o passo a passo da implementação do código em Verilog:

Criação do módulo, registradores e wires e declaração de inputs e outputs:

```
module hamming (clk, endereco , mudanca, mensagem, out0, out1, out2, out3, out4, out5, out6, out7, encode, enviar, reset);
   input clk;
   input endereco;
   input enviar;
   input [7:0] mensagem;
   input mudanca;
   input encode;
   input reset;
   reg [3:0] aleatorio;
   reg[11:0] hamming;
   reg[7:0] final;
   reg [3:0] detectar;
   reg [3:0] deteccao;
   reg [3:0] concat;
   reg[7:0] slave1;
   reg[7:0] slave2;
   reg[11:0] semifinal;
   output reg [6:0] out0;
   output reg [6:0] out1;
   output reg [6:0] out2;
   output reg [6:0] out3;
   output reg [6:0] out4;
   output reg [6:0] out5;
   output reg [6:0] out6;
   output reg [6:0] out7;
```

Bloco always para implementação do código Hamming e geração do erro. Declaração do valor inicial da variável "aleatório" seguida do implementação da aleatoriedade baseada nos ciclos de clock:

```
28      always @(posedge encode or posedge reset) begin
29          if(reset) begin
30              hamming[2] = 1'bx;
31              hamming[4] = 1'bx;
32              hamming[5] = 1'bx;
33              hamming[6] = 1'bx;
34              hamming[8] = 1'bx;
35              hamming[9] = 1'bx;
36              hamming[10] = 1'bx;
37              hamming[11] = 1'bx;
38              hamming[0] = 1'bx;
39              hamming[1] = 1'bx;
40              hamming[3] = 1'bx;
41              hamming[7] = 1'bx;
42          end else begin
43              hamming[2] = mensagem[0];
44              hamming[4] = mensagem[1];
45              hamming[5] = mensagem[2];
46              hamming[6] = mensagem[3];
47              hamming[8] = mensagem[4];
48              hamming[9] = mensagem[5];
49              hamming[10] = mensagem[6];
50              hamming[11] = mensagem[7];
51              hamming[0] = mensagem[0] ^ mensagem[1] ^ mensagem[3] ^ mensagem[4] ^ mensagem[6];
52              hamming[1] = mensagem[0] ^ mensagem[2] ^ mensagem[3] ^ mensagem[5] ^ mensagem[6];
53              hamming[3] = mensagem[1] ^ mensagem[2] ^ mensagem[3] ^ mensagem[7];
54              hamming[7] = mensagem[4] ^ mensagem[5] ^ mensagem[6] ^ mensagem[7];
55              if(mudanca) begin
56                 hamming[aleatorio] <= !hamming[aleatorio];
57              end
58          end
59      end
60  initial begin
61      aleatorio = 4'b0000;
62  end
63
64  always @(posedge clk) begin
65      aleatorio <= aleatorio + 1'b1;
66      if (aleatorio == 4'b1011)begin
67          aleatorio <= 4'b0000;
68      end
69  end
```

Criação da task responsável pelo funcionamento do display de 7 segmentos:

```
task bcd7seg_bl;
output reg [6:0] out0;
output reg [6:0] out1;
output reg [6:0] out2;
output reg [6:0] out3;
output reg [6:0] out4;
output reg [6:0] out5;
output reg [6:0] out6;
output reg [6:0] out7;
input [7:0] final;
begin
   case(final[0])
        1'b0: out0 <= 7'b1000000;
        1'b1: out0 <= 7'b1111001;
        default: out0 <= 7'b0000110;  // 0
   endcase
   case(final[1])
        1'b0: out1 <= 7'b1000000;
        1'b1: out1 <= 7'b1111001;
        default: out1 <= 7'b0000110;  // 0
   endcase
   case(final[2])
        1'b0: out2 <= 7'b1000000;
        1'b1: out2 <= 7'b1111001;
        default: out2 <= 7'b0000110;  // 0
   endcase
   case(final[3])
        1'b0: out3 <= 7'b1000000;
        1'b1: out3 <= 7'b1111001;
        default: out3 <= 7'b0000110;  // 0
   endcase
   case(final[4])
        1'b0: out4 <= 7'b1000000;
        1'b1: out4 <= 7'b1111001;
        default: out4 <= 7'b0000110;  // 0
   endcase
   case(final[5])
        1'b0: out5 <= 7'b1000000;
        1'b1: out5 <= 7'b1111001;
        default: out5 <= 7'b0000110;  // 0
   endcase
   case(final[6])
        1'b0: out6 <= 7'b1000000;
        1'b1: out6 <= 7'b1111001;
        default: out6 <= 7'b0000110;  // 0
   endcase
   case(final[7])
        1'b0: out7 <= 7'b1000000;
        1'b1: out7 <= 7'b1111001;
        default: out7 <= 7'b0000110;  // 0
   endcase
end
endtask
```

Blocos responsáveis por realizar as operações que irão detectar o erro e avaliar se o erro existe ou não. Implementação da funcionalidade de correção:

```
123
124   always @(posedge clk) begin
125       deteccao[0] <= hamming[0] ^ hamming[2] ^ hamming[4] ^ hamming[6] ^ hamming[8] ^ hamming[10];
126       deteccao[1] <= hamming[1] ^ hamming[2] ^ hamming[5] ^ hamming[6] ^ hamming[9] ^ hamming[10];
127       deteccao[2] <= hamming[3] ^ hamming[4] ^ hamming[5] ^ hamming[6] ^ hamming[11];
128       deteccao[3] <= hamming[7] ^ hamming[8] ^ hamming[9] ^ hamming[10] ^ hamming[11];
129       if(reset)begin
130         deteccao[0] <= 1'bx;
131         deteccao[1] <= 1'bx;
132         deteccao[2] <= 1'bx;
133         deteccao[3] <= 1'bx;
134       end
135   end
136
137   always @(posedge clk) begin
138       concat <= {deteccao[3], deteccao[2], deteccao[1], deteccao[0]};
139       if (concat != 4'b0000) begin
140           case(concat)
141            4'b0001: detectar <= 4'd1;
142            4'b0010: detectar <= 4'd2;
143            4'b0011: detectar <= 4'd3;
144            4'b0100: detectar <= 4'd4;
145            4'b0101: detectar <= 4'd5;
146            4'b0110: detectar <= 4'd6;
147            4'b0111: detectar <= 4'd7;
148            4'b1000: detectar <= 4'd8;
149            4'b1001: detectar <= 4'd9;
150            4'b1010: detectar <= 4'd10;
151            4'b1011: detectar <= 4'd11;
152            4'b1100: detectar <= 4'd12;
153            4'b1101: detectar <= 4'd13;
154            4'b1110: detectar <= 4'd14;
155            4'b1111: detectar <= 4'd15;
156            default: detectar <= 4'd0;
157         endcase
158       end
159   end
160   always @(*) begin // correção do código
161      semifinal[11:0] = hamming[11:0];
162      semifinal[detectar-1] = !semifinal[detectar-1];
163      final[0] = semifinal[2];
164      final[1] = semifinal[4];
165      final[2] = semifinal[5];
166      final[3] = semifinal[6];
167      final[4] = semifinal[8];
168      final[5] = semifinal[9];
169      final[6] = semifinal[10];
170      final[7] = semifinal[11];
171      if (reset) begin
172         final[7:0] = 8'bxxxxxxxx;
173         semifinal[11:0] = 12'bxxxxxxxxxxxx;
174      end
175   end
```

Criação dos endereços da interface de comunicação e do mecanismo de decisão:

```
176   always @(posedge clk) begin
177      if (reset)begin
178          out0 <= 7'bxxxxxxx;
179          out1 <= 7'bxxxxxxx;
180          out2 <= 7'bxxxxxxx;
181          out3 <= 7'bxxxxxxx;
182          out4 <= 7'bxxxxxxx;
183          out5 <= 7'bxxxxxxx;
184          out6 <= 7'bxxxxxxx;
185          out7 <= 7'bxxxxxxx;
186          slave1 <= 8'bxxxxxxxx;
187          slave2 <= 8'bxxxxxxxx;
188      end
189      if (endereco == 1'b0) begin
190          if (enviar == 1'b1) begin
191             slave1[7:0] <= final[7:0];
192          end
193          bcd7seg_bl(out0, out1, out2, out3, out4, out5, out6, out7, slave1);
194      end
195      if (endereco == 1'b1) begin
196          if (enviar == 1'b1) begin
197             slave2[7:0] <= final[7:0];
198          end
199          bcd7seg_bl(out0, out1, out2, out3, out4, out5, out6, out7, slave2);
200      end
201   end
202
203 endmodule
```

4.2 AVALIAÇÃO DA SOLUÇÃO

As soluções se apresentaram serem eficientes para a realização do projeto, tanto a interface de comunicação serial quanto o código Hamming mostraram-se funcionais dentro das funcionalidades propostas. Porém, é importante destacar que para outros projetos, como onde há maior uma maior quantidade de bits sendo transmitida, por exemplo, as soluções implementadas podem não ser ideais, tendo em vista as limitações práticas delas. Todavia, para o proposto, tanto nos testes manuais utilizando o Modelsim, quanto no testbench, todos os resultados apresentaram-se consistentes, evidenciando que, posteriormente, todos os resultados foram conferidos utilizando um código criado em Python.

Ademais, é plausível ressaltar que uma das maiores dificuldades, do projeto, além do código Hamming e da interface de comunicação serial, foi implementar uma funcionalidade que gerasse o erro na mensagem. Contudo, essa dificuldade foi sanada e mostrou-se funcional.

Para avaliar o trabalho final, a equipe dedicou boa parte do tempo de estudos, pesquisa e desenvolvimento para a aplicação de um testbench que conseguisse não só testar a funcionalidade do projeto, mas como todas as suas possibilidades de entradas. Nisso, usando-se todos os 11 inputs necessários para o funcionamento, conseguimos testar, com êxito, as 2048 combinações de entradas, com suas saídas tendo uma porcentagem de 100% de acerto. Para a comparação, utilizamos diferentes códigos em linguagem de programação para chegar a essa conclusão. Vamos, agora, detalhar o passo a passo da produção do testbench.

Primeiramente, criamos um código em python que conseguisse produzir todas as possibilidades de inputs do nosso projeto. O código em questão é a figura:

```python
x = 0
variaveis = input('Ponha quantas variáveis tem: ')
variaveis = int(variaveis)
acrescentar = ''
with open('inputsCONTADOR.txt', 'w', encoding ='utf-8') as arquivo:
    for i in range(2**variaveis):
        x = x + 1
        def converte(numDecimal, base):
            if numDecimal != 0:
                numConvertido = ""
                while numDecimal > 0:
                    resto = numDecimal % base
                    numConvertido = str(resto) + numConvertido
                    numDecimal = numDecimal // base
            else:
                numConvertido = "0"
            return numConvertido
        k = converte(x, 2)
        if (len(k) < variaveis):
            for i in range(variaveis - len(k)):
                acrescentar = acrescentar + '0'
            k = acrescentar + k
            acrescentar = ''
        arquivo.write(f"{k}\n")
```

Após conseguir todos esses inputs, criamos, novamente, um código em linguagem de programação que simulasse o funcionamento do projeto e, a partir disso, nos informasse todos os outputs que deveríamos ter, caso o código em verilog estivesse completamente correto. Com isso, temos o código na figura:

```python
import random

arquivo = open('inputs.txt', 'r')
linhas = arquivo.readlines()
arquivo.close()

with open('outputs.txt', 'w', encoding='utf-8') as arquivo_saida:
    for linha in linhas:
        slave1 = 'xxxxxxxx'
        slave2 = 'xxxxxxxx'
        hamming = 'xxxxxxxxxxxx'
        encode = linha[0]
        endereco = linha[1]
        enviar = linha[2]
        d = linha[3:11]
        if (encode == '1'):
            data = list(d)
            data.reverse()
            c, ch, j, r, h = 0, 0, 0, 0, []

            while (len(d) + r + 1) > (pow(2, r)):
                r = r + 1

            for i in range(0, (r + len(data))):
                p = 2 ** c

                if p == (i + 1):
                    h.append(0)
                    c = c + 1

                else:
                    h.append(int(data[j]))
                    j = j + 1

            for parity in range(0, (len(h))):
                ph = 2 ** ch
                if ph == (parity + 1):
                    startIndex = ph - 1
                    i = startIndex
                    toXor = []

                    while i < len(h):
                        block = h[i: i + ph]
                        toXor.extend(block)
                        i += 2 * ph

                    for z in range(1, len(toXor)):
                        h[startIndex] = h[startIndex] ^ toXor[z]
                    ch += 1
            h.reverse()
            hamming ="".join(map(str, h))

        if endereco == '0' and enviar == '1' and encode == '1':
            slave1 = d
        if endereco == '1' and enviar == '1' and encode == '1':
            slave2 = d
        arquivo_saida.write(f'{slave1}{slave2}{hamming}{d}\n')
```

Após conseguirmos essas informações, criamos o código do testbench, para conseguir os outputs que estávamos obtendo diretamente da linguagem de descrição de hardware. Encontra-se na figura:

```
module hamming_tb();
  integer i;
  reg [10:0] data [0:2047];
  reg clk_tb, endereco_tb, mudanca_tb, encode_tb, enviar_tb, reset_tb;
  reg [7:0] mensagem_tb;
  wire [7:0] resultado_tb;
  wire [6:0] out0_tb, out1_tb, out2_tb, out3_tb, out4_tb, out5_tb, out6_tb, out7_tb;


  hamming ham(.clk(clk_tb), .endereco(endereco_tb) , .mudanca(mudanca_tb), .mensagem(mensagem_tb), .resultado(resultado_tb), .out0(out0_tb),
.out1(out1_tb), .out2(out2_tb), .out3(out3_tb), .out4(out4_tb), .out5(out5_tb), .out6(out6_tb), .out7(out7_tb), .encode(encode_tb),
.enviar(enviar_tb), .reset(reset_tb));

  always begin
    #1 clk_tb = ~clk_tb;
  end

  initial begin
    clk_tb = 1'b0;
    $readmemb("inputs.txt", data);
    for (i = 0; i < 2048; i = i + 1) begin
      encode_tb = 1'b0;
      endereco_tb = 1'b0;
      enviar_tb = 1'b0;
      mensagem_tb[0] = 1'b0;
      mensagem_tb[1] = 1'b0;
      mensagem_tb[2] = 1'b0;
      mensagem_tb[3] = 1'b0;
      mensagem_tb[4] = 1'b0;
      mensagem_tb[5] = 1'b0;
      mensagem_tb[6] = 1'b0;
      mensagem_tb[7] = 1'b0;
      reset_tb = 1'b1;
      #2;
      reset_tb = 1'b0;
      {encode_tb, endereco_tb, enviar_tb, mensagem_tb[7], mensagem_tb[6], mensagem_tb[5], mensagem_tb[4], mensagem_tb[3], mensagem_tb[2],
mensagem_tb[1], mensagem_tb[0]} = {data[i][10], data[i][9], data[i][8], data[i][7], data[i][6], data[i][5], data[i][4], data[i][3], data[i]
[2], data[i][1], data[i][0]};
      //$display("Resultado[%0d]: encode:%b endereco:%b enviar:%b mensagem: %b", i, encode_tb, endereco_tb, enviar_tb, mensagem_tb);
      #2;
      //$display("Resultado[%0d]: %b%b%b%b", i, ham.slave1, ham.slave2, ham.hamming, mensagem_tb);
      $display("%b%b%b%b", ham.slave1, ham.slave2, ham.hamming, mensagem_tb);
    end
    $finish;
  end
endmodule
```

Finalmente, após obter todos os outputs do testbench, fizemos um código em python que comparasse, individualmente, cada output desejado com o que foi, de fato, adquirido. O código está na figura:

comparar.py > ...

```python
outputs_tb = open("outputstestbenchCONTADOR.txt", "r")
linhas1 = outputs_tb.readlines()
outputs_tb.close()
outputs = open("outputsCONTADOR.txt", "r")
linhas2 = outputs.readlines()
outputs.close()
corretos = 0
for i in range(256):
    outputs_c = linhas1[i]
    outputs_tb_c = linhas2[i]
    if outputs_c == outputs_tb_c:
        corretos = corretos + 1
    else:
        print(f"errado: {outputs_c} e {outputs_tb_c} em {i+1}")
print(f"corretos: {corretos} de {i+1}")

```

5 RESULTADOS

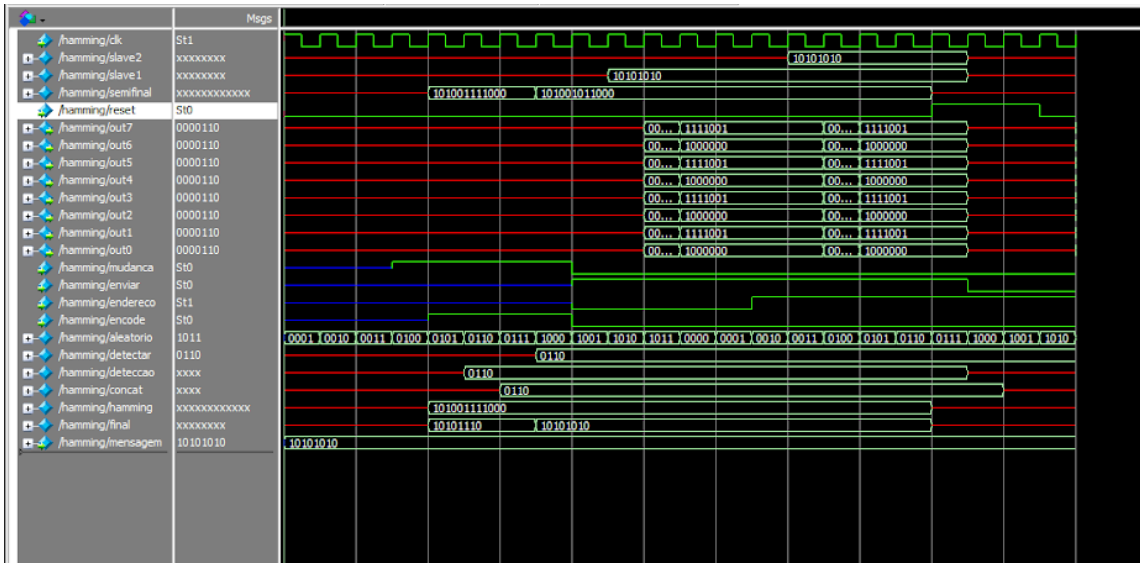
Após todo esse processo para a construção do testbench, e rodando o código de comparação, obtivemos o seguinte resultado, na figura:

```
comparar.py > ...
1   outputs_tb = open("outputs.txt", "r")
2   linhas1 = outputs_tb.readlines()
3   outputs_tb.close()
4   outputs = open("outputtestbench.txt", "r")
5   linhas2 = outputs.readlines()
6   outputs.close()
7   corretos = 0
8   for i in range(2048):
9       outputs_c = linhas1[i]
10      outputs_tb_c = linhas2[i]
11      if outputs_c == outputs_tb_c:
12          corretos = corretos + 1
13      else:
14          print(f"errado: {outputs_c} e {outputs_tb_c} em {i+1}")
15  print(f"corretos: {corretos} de {i+1}")
16

PROBLEMAS   SAÍDA   CONSOLE DE DEPURAÇÃO   TERMINAL

PS C:\Users\joao0\codigos> & C:/Users/joao0/AppData/Local/Programs/Python/Pyth
corretos: 2048 de 2048
PS C:\Users\joao0\codigos>
```

Além disso, utilizamos o waveform para ter mais um tipo de segurança, após a compilação final do projeto no Quartus, com a ferramenta ModelSim:



Com isso, obtivemos um feedback de que o projeto estava de acordo com o esperado.

6 CRONOGRAMA E RECURSOS

Evento	Data
Entrega da Proposta Parcial	09/05/2023
Construção do Código em Verilog	15/06/2023
Término do Testbench em Verilog	09/07/2023

As datas dos códigos apresentam-se somente como forma de uma média, uma vez que - quando era detectado erro a partir do testbench - havia modificações no módulo principal.

Para a escrita do código e compilação foram utilizados os softwares EDA Playground e Quartus, tendo em vista a aplicação do projeto na placa DE2-115.

7 REFERÊNCIAS BIBLIOGRÁFICAS

PLESS, Vera. **Introduction to the theory of error-correcting codes**. New York: Wiley, 1982, ISBN 0471086843.

UNIVERSIDADE DE SÃO PAULO. **Código de Hamming**. São Paulo, 2004.

MINNS, Peter et al. **FSM-based Digital Design using Verilog HDL**. Inglaterra: Wiley, 2008.