UNIVERSIDADE DE SÃO PAULO

ESCOLA DE ENGENHARIA DE SÃO CARLOS

Decodificador BCD para Display de 7 Segmentos

Entrega 1

SEL 0628 – Sistemas Digitais

Prof. Dr. Maximiliam Luppe

Daniel Dias Silva Filho - 13677114

Daniel Umeda Kuhn - 13676541

Francyélio - 13676537

João Marcelo Ferreira Battaglini - 13835472

Manoel Thomaz - 13676392

Lucas Sales Duarte - 11734490

São Carlos – SP

01/07/2024

Daniel Dias Silva Filho - 13677114
Daniel Umeda Kuhn - 13676541
Francyélio - 13676537
João Marcelo Ferreira Battaglini - 13835472
Manoel Thomaz - 13676392
Lucas Sales Duarte - 11734490

1° Entrega

1° parte do trabalho de recuperação da disciplina SEL 0628
Professor: Maximiliam Luppe

São Carlos – SP

01/07/2024

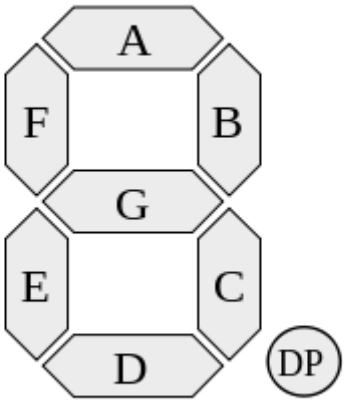
1. Introdução

Este projeto foca no desenvolvimento de um Decodificador BCD para display de 7 segmentos, essencial para converter sinais binários em representações numéricas visuais nos dispositivos eletrônicos. A implementação suporta ambos os tipos de displays, anodo comum e cátodo comum, por meio do parâmetro `common_cathode`, garantindo ampla compatibilidade com diversos tipos de hardware. Tal flexibilidade permite sua aplicação em uma vasta gama de sistemas, desde relógios digitais a complexos instrumentos de medição.

Inicialmente, o relatório detalha a construção da Tabela Verdade e a utilização dos Mapas de Karnaugh para simplificação das expressões booleanas, formando a base do circuito lógico do decodificador. Estas expressões são cruciais para definir como cada segmento do display é controlado, assegurando a correta representação dos números de 0 a 9 conforme o padrão BCD.

Em seguida, apresentamos a implementação do decodificador utilizando a linguagem de descrição de hardware Verilog, explorada através de quatro abordagens distintas: primitivas lógicas, operadores lógicos concorrentes, operador ternário e uma forma comportamental com blocos `always` e `if-else`. Cada técnica demonstra uma metodologia de design diferenciada, proporcionando uma visão abrangente sobre as possibilidades de modelagem e otimização do circuito. Por fim, a figura 1 demonstra o que cada segmento do display representa.

figura 1: representação gráfica do display de 7 segmentos



2. Desenvolvimento do projeto

2.1. Tabela verdade

Inicialmente, foi desenvolvida a tabela verdade com base em cada possível entrada e suas respectivas saídas. Como demonstrado na figura 1, cada saída representa um segmento do display de 7 segmentos. Devido à limitação de 7 segmentos, é possível representar 10 números, os dígitos de 0 a 9.

figura 2: tabela verdade

Entradas				Dígito	Saída 7 segmentos						
E1	E2	E3	E4		a	b	c	d	e	f	g
0	0	0	0	0	1	1	1	1	1	1	0
1	0	0	0	1	0	1	1	0	0	0	0
0	1	0	0	2	1	1	0	1	1	0	1
1	1	0	0	3	1	1	1	1	0	0	1
0	0	1	0	4	0	1	1	0	0	1	1
1	0	1	0	5	1	0	1	1	0	1	1
0	1	1	0	6	1	0	1	1	1	1	1
1	1	1	0	7	1	1	1	0	0	0	0
0	0	0	1	8	1	1	1	1	1	1	1
1	0	0	1	9	1	1	1	0	0	1	1

2.2. Expressões Booleanas e Mapas de Karnaugh

Com a tabela verdade em mãos, foi possível gerar as expressões booleanas correspondentes a cada segmento:

a: $E1 + E3 + \overline{E2}\,\overline{E4} + E2E4$

b: $\overline{E2} + \overline{E3}\,\overline{E4} + E3E4$

c: $\overline{E3} + E2 + E4$

d: $\overline{E2}\,\overline{E4} + \overline{E2}E3 + E2\overline{E3}E4 + E3\,\overline{E4} + E1$

e: $\overline{E2}\,\overline{E4} + E3\,\overline{E4}$

f: $\overline{E3}\,\overline{E4} + E2\,\overline{E3} + E2\,\overline{E4} + E1$

g: $\overline{E2}\,E3 + E2\,\overline{E3} + E1 + E2\,\overline{E4}$

Além disso, também foram feitos os Mapas de Karnaugh correspondentes:

a:

E1E2\E3 E4	00	01	11	10
00	1	0	1	1

00	0	1	1	1
11	X	X	X	X
10	1	1	X	X

b:

E1E2\E3 E4	00	01	11	10
00	1	1	1	1
00	1	0	1	0
11	X	X	X	X
10	1	1	X	X

c:

E1E2\E3 E4	00	01	11	10

00	1	1	1	0
00	1	1	1	1
11	X	X	X	X
10	1	1	X	X

d:

E1E2\E3 E4	00	01	11	10
00	1	0	1	1
00	0	1	0	1
11	X	X	X	X
10	1	1	X	X

e:

E1E2\E3 E4	00	01	11	10

00	1	0	0	1
00	0	0	0	1
11	X	X	X	X
10	1	0	X	X

f:

E1E2\E3 E4	00	01	11	10
00	1	0	0	0
00	1	1	0	1
11	X	X	X	X
10	1	1	X	X

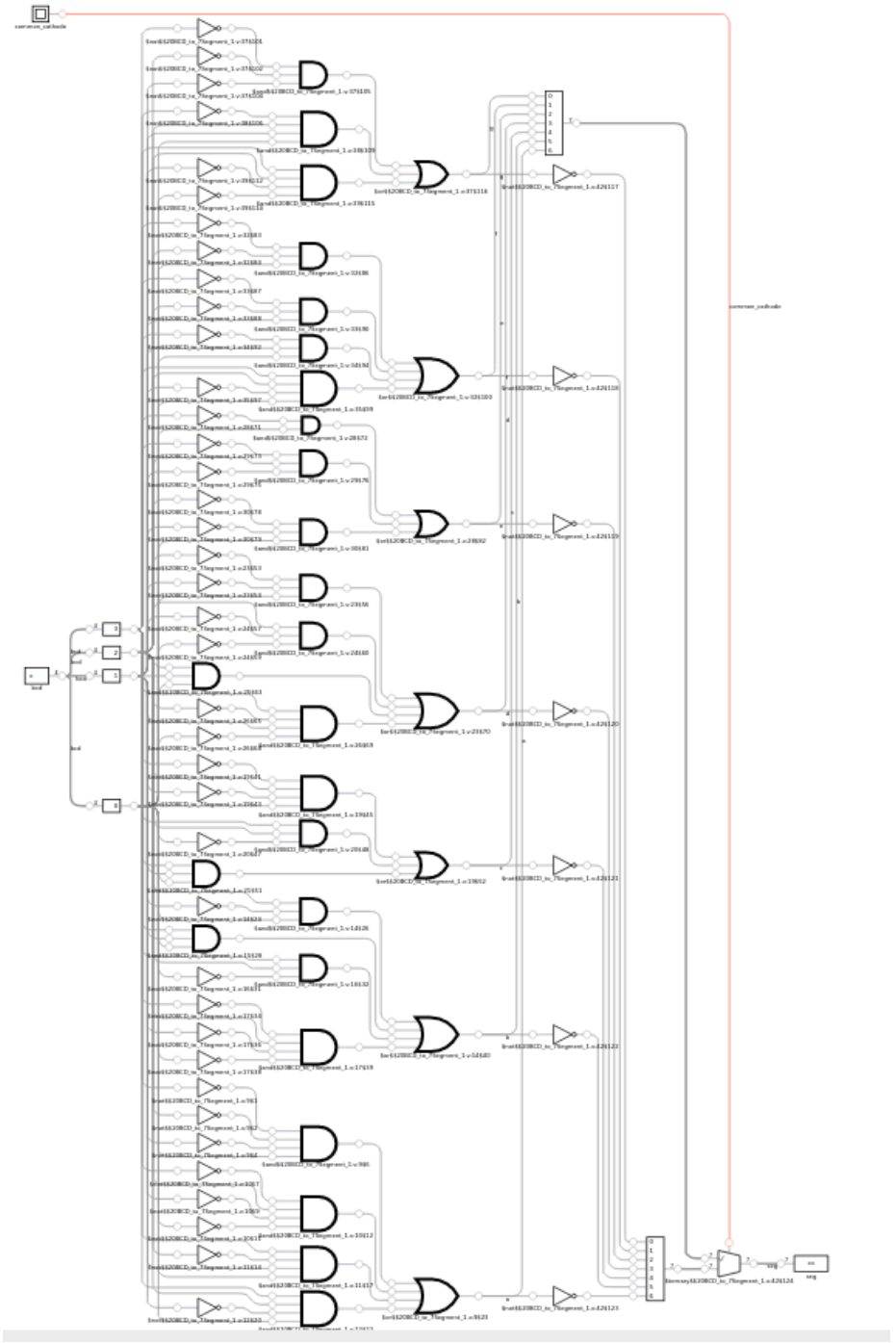
g:

E1E2\E3 E4	00	01	11	10
00	0	0	1	1
00	1	1	0	1

11	X	X	X	X
10	1	1	X	X

2.3. Circuito RTL

figura 3: Circuito RTL



2.4. Códigos

2.4.1. Primitivas de ligação

```
module BCD_to_7Segment_primitive (
    input [3:0] bcd,
    input common_cathode,
    output [6:0] seg
);
    wire a, b, c, d, e, f, g;

    // Definir as primitivas de lógica
    assign a = (~bcd[3] & ~bcd[2] & ~bcd[1] & bcd[0]) |
               (~bcd[3] & bcd[2] & ~bcd[1] & ~bcd[0]) |
               (bcd[3] & ~bcd[2] & bcd[1] & bcd[0]) |
               (bcd[3] & bcd[2] & ~bcd[1] & bcd[0]);

    assign b = (bcd[2] & ~bcd[1] & bcd[0]) |
               (bcd[3] & bcd[1] & bcd[0]) |
               (bcd[3] & bcd[2] & ~bcd[0]) |
               (~bcd[3] & bcd[2] & ~bcd[1] & ~bcd[0]);

    assign c = (~bcd[3] & bcd[2] & ~bcd[1] & bcd[0]) |
               (bcd[3] & bcd[2] & ~bcd[0]) |
               (bcd[3] & bcd[1] & bcd[0]);

    assign d = (~bcd[2] & ~bcd[1] & bcd[0]) |
               (bcd[2] & ~bcd[1] & ~bcd[0]) |
               (bcd[2] & bcd[1] & bcd[0]) |
               (bcd[3] & ~bcd[2] & bcd[1] & ~bcd[0]);

    assign e = (~bcd[3] & bcd[0]) |
               (~bcd[3] & bcd[2] & ~bcd[1]) |
               (~bcd[2] & ~bcd[1] & bcd[0]);

    assign f = (~bcd[3] & ~bcd[2] & bcd[0]) |
               (~bcd[3] & ~bcd[2] & bcd[1]) |
               (~bcd[3] & bcd[1] & bcd[0]) |
               (bcd[3] & bcd[2] & ~bcd[1] & bcd[0]);

    assign g = (~bcd[3] & ~bcd[2] & ~bcd[1]) |
               (~bcd[3] & bcd[2] & bcd[1] & bcd[0]) |
               (bcd[3] & bcd[2] & ~bcd[1] & ~bcd[0]);
```

```
    // Ajustar a polaridade das saídas dependendo de common_cathode
    assign seg = common_cathode ? {a, b, c, d, e, f, g} : {~a, ~b, ~c, ~d, ~e, ~f,
~g};
endmodule
```

2.4.2. Declarações concorrentes com operadores lógicos

```
module BCD_to_7Segment_logical (
    input [3:0] bcd,
    input common_cathode,
    output [6:0] seg
);
    wire a, b, c, d, e, f, g;

    assign a = (~bcd[3] & ~bcd[2] & ~bcd[1] & bcd[0]) |
               (~bcd[3] & bcd[2] & ~bcd[1] & ~bcd[0]) |
               (bcd[3] & ~bcd[2] & bcd[1] & bcd[0]) |
               (bcd[3] & bcd[2] & ~bcd[1] & bcd[0]);

    assign b = (bcd[2] & ~bcd[1] & bcd[0]) |
               (bcd[3] & bcd[1] & bcd[0]) |
               (bcd[3] & bcd[2] & ~bcd[0]) |
               (~bcd[3] & bcd[2] & ~bcd[1] & ~bcd[0]);

    assign c = (~bcd[3] & bcd[2] & ~bcd[1] & bcd[0]) |
               (bcd[3] & bcd[2] & ~bcd[0]) |
               (bcd[3] & bcd[1] & bcd[0]);

    assign d = (~bcd[2] & ~bcd[1] & bcd[0]) |
               (bcd[2] & ~bcd[1] & ~bcd[0]) |
               (bcd[2] & bcd[1] & bcd[0]) |
               (bcd[3] & ~bcd[2] & bcd[1] & ~bcd[0]);

    assign e = (~bcd[3] & bcd[0]) |
               (~bcd[3] & bcd[2] & ~bcd[1]) |
               (~bcd[2] & ~bcd[1] & bcd[0]);

    assign f = (~bcd[3] & ~bcd[2] & bcd[0]) |
               (~bcd[3] & ~bcd[2] & bcd[1]) |
               (~bcd[3] & bcd[1] & bcd[0]) |
```

```
                (bcd[3] & bcd[2] & ~bcd[1] & bcd[0]);

        assign g = (~bcd[3] & ~bcd[2] & ~bcd[1]) |
                (~bcd[3] & bcd[2] & bcd[1] & bcd[0]) |
                (bcd[3] & bcd[2] & ~bcd[1] & ~bcd[0]);

        assign seg = common_cathode ? {a, b, c, d, e, f, g} : {~a, ~b, ~c, ~d, ~e, ~f,
~g};
    endmodule
```

2.4.3. Declarações concorrentes com operadores ternários

```
    module BCD_to_7Segment_ternary (
        input [3:0] bcd,
        input common_cathode,
        output [6:0] seg
    );
        assign seg[0] = common_cathode ?
                ((~bcd[3] & ~bcd[2] & ~bcd[1] & bcd[0]) |
                (~bcd[3] & bcd[2] & ~bcd[1] & ~bcd[0]) |
                (bcd[3] & ~bcd[2] & bcd[1] & bcd[0]) |
                (bcd[3] & bcd[2] & ~bcd[1] & bcd[0])) :
                ~((~bcd[3] & ~bcd[2] & ~bcd[1] & bcd[0]) |
                (~bcd[3] & bcd[2] & ~bcd[1] & ~bcd[0]) |
                (bcd[3] & ~bcd[2] & bcd[1] & bcd[0]) |
                (bcd[3] & bcd[2] & ~bcd[1] & bcd[0]));

        assign seg[1] = common_cathode ?
                ((bcd[2] & ~bcd[1] & bcd[0]) |
                (bcd[3] & bcd[1] & bcd[0]) |
                (bcd[3] & bcd[2] & ~bcd[0]) |
                (~bcd[3] & bcd[2] & ~bcd[1] & ~bcd[0])) :
                ~((bcd[2] & ~bcd[1] & bcd[0]) |
                (bcd[3] & bcd[1] & bcd[0]) |
                (bcd[3] & bcd[2] & ~bcd[0]) |
                (~bcd[3] & bcd[2] & ~bcd[1] & ~bcd[0]));

        assign seg[2] = common_cathode ?
                ((~bcd[3] & bcd[2] & ~bcd[1] & bcd[0]) |
                (bcd[3] & bcd[2] & ~bcd[0]) |
```

```
                (bcd[3] & bcd[1] & bcd[0])) :
                ~((~bcd[3] & bcd[2] & ~bcd[1] & bcd[0]) |
                (bcd[3] & bcd[2] & ~bcd[0]) |
                (bcd[3] & bcd[1] & bcd[0]));

        assign seg[3] = common_cathode ?
                ((~bcd[2] & ~bcd[1] & bcd[0]) |
                (bcd[2] & ~bcd[1] & ~bcd[0]) |
                (bcd[2] & bcd[1] & bcd[0]) |
                (bcd[3] & ~bcd[2] & bcd[1] & ~bcd[0])) :
                ~((~bcd[2] & ~bcd[1] & bcd[0]) |
                (bcd[2] & ~bcd[1] & ~bcd[0]) |
                (bcd[2] & bcd[1] & bcd[0]) |
                (bcd[3] & ~bcd[2] & bcd[1] & ~bcd[0]));

        assign seg[4] = common_cathode ?
                ((~bcd[3] & bcd[0]) |
                (~bcd[3] & bcd[2] & ~bcd[1]) |
                (~bcd[2] & ~bcd[1] & bcd[0])) :
                ~((~bcd[3] & bcd[0]) |
                (~bcd[3] & bcd[2] & ~bcd[1]) |
                (~bcd[2] & ~bcd[1] & bcd[0]));

        assign seg[5] = common_cathode ?
                ((~bcd[3] & ~bcd[2] & bcd[0]) |
                (~bcd[3] & ~bcd[2] & bcd[1]) |
                (~bcd[3] & bcd[1] & bcd[0]) |
                (bcd[3] & bcd[2] & ~bcd[1] & bcd[0])) :
                ~((~bcd[3] & ~bcd[2] & bcd[0]) |
                (~bcd[3] & ~bcd[2] & bcd[1]) |
                (~bcd[3] & bcd[1] & bcd[0]) |
                (bcd[3] & bcd[2] & ~bcd[1] & bcd[0]));

        assign seg[6] = common_cathode ?
                ((~bcd[3] & ~bcd[2] & ~bcd[1]) |
                (~bcd[3] & bcd[2] & bcd[1] & bcd[0]) |
                (bcd[3] & bcd[2] & ~bcd[1] & ~bcd[0])) :
                ~((~bcd[3] & ~bcd[2] & ~bcd[1]) |
                (~bcd[3] & bcd[2] & bcd[1] & bcd[0]) |
                (bcd[3] & bcd[2] & ~bcd[1] & ~bcd[0]));
endmodule
```

2.4.4. Declaração procedural ou comportamental

```
module BCD_to_7Segment_procedural (
    input [3:0] bcd,
    input common_cathode,
    output reg [6:0] seg
);
    always @(*) begin
    case (bcd)
        4'b0000: seg = 7'b0111111; // 0
        4'b0001: seg = 7'b0000110; // 1
        4'b0010: seg = 7'b1011011; // 2
        4'b0011: seg = 7'b1001111; // 3
        4'b0100: seg = 7'b1100110; // 4
        4'b0101: seg = 7'b1101101; // 5
        4'b0110: seg = 7'b1111101; // 6
        4'b0111: seg = 7'b0000111; // 7
        4'b1000: seg = 7'b1111111; // 8
        4'b1001: seg = 7'b1100111; // 9
        default: seg = 7'b0000000; // Default to off for invalid inputs
    endcase

    if (!common_cathode) begin
        seg = ~seg;
    end
    end
endmodule
```

3. Conclusão

A implementação do decodificador foi explorada em quatro abordagens distintas, cada uma oferecendo uma perspectiva única sobre a modelagem do circuito.

Cada uma dessas técnicas tem suas próprias vantagens e desvantagens, e a escolha entre elas pode depender de fatores como a complexidade do design, a legibilidade do código e as preferências pessoais do designer.

Ao longo deste projeto, destacamos a importância de entender profundamente tanto os fundamentos teóricos quanto as práticas de implementação em hardware descritivo (HDL). Através deste estudo, os alunos puderam aplicar conceitos teóricos em práticas de design, simulação e verificação de circuitos digitais.

A implementação em Verilog demonstrou como diferentes metodologias de design podem ser aplicadas para alcançar o mesmo objetivo funcional, oferecendo uma visão abrangente sobre as possibilidades de modelagem e otimização de circuitos digitais.

Em suma, este trabalho não só cumpriu o objetivo de criar um decodificador BCD funcional para displays de 7 segmentos, mas também proporcionou um valioso aprendizado sobre a flexibilidade e as técnicas de design em Verilog, preparando os alunos para desafios mais complexos no campo da Engenharia de Computação.

4. Referências

Nelson, Victor. Nagle, H. Carroll, Bill. Irwin, David. Digital Logic Circuit Analysis and Design. Upper
Saddle River, New Jersey. Prentice Hall, 1995