Departamento de Ciência da Computação

Prof. Bruno de Abreu Silva

GCC260 – Laboratório de Circuitos Digitais

Circuitos lógicos MSI

1. Objetivos

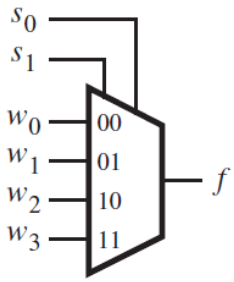
- Aprender sobre estruturas de circuitos combinacionais muito comuns, como multiplexadores, codificadores e decodificadores e Unidade Lógica e Aritmética;
- Aprender estruturas básicas em Verilog para circuitos combinacionais comuns;
- Simular os circuitos para melhor entendimento sobre seu funcionamento.

2. Introdução

Até o momento, a maioria dos circuitos trabalhados nas práticas eram compostos de operações bit-a-bit ou operações lógicas básicas (*and*, *or* e *not*) para descrever projetos em nível de portas lógicas (*gate-level design*). Nesta prática, serão trabalhadas descrições de circuitos compostas por componentes de tamanho intermediário (MSI – *Medium Scale Integration*), como somadores, comparadores e multiplexadores. Diferente do *gate-level design*, estes componentes são os blocos básicos de construção usados na metodologia de projeto conhecida como transferência de registradores (*Register Transfer*). Por isso, são também conhecidos como componentes de *RT-level design* (Projeto em nível de transferência de registradores).

2.1. Multiplexadores

Um multiplexador define qual de suas entradas será direcionada para a saída em função do valor enviado para a entrada de seleção. A Figura 1 ilustra o símbolo gráfico de um multiplexador 4x1 e sua tabela-verdade.



(a) Graphical symbol

s_1	s_0	f
0	0	w_0
0	1	w_1
1	0	w_2
1	1	w_3

(b) Truth table

Figura 1: Multiplexador 4x1.

Uma possível implementação de um multiplexador 4x1 em Verilog é usando o comando *case*, como no exemplo a seguir:

```
// mux4to1.v
module mux4to1
#(
        parameter N = 4
)
(
        input wire [N-1:0] A, B, C, D,
        input wire [1:0] S,
        output reg [N-1:0] Z
);
  always@*
  begin
          case(S)
                  0: Z = A;
                  1: Z = B;
                  2: Z = C;
                  3: Z = D;
          endcase
  end
 endmodule
```

Para simular o circuito do multiplexador, é necessário construir um *testbench* capaz de realizar alguns testes. O código abaixo é um exemplo de *testbench* para o mux4to1.

```
// mux4to1_tb.v
`timescale 1ns/10ps

module mux4to1_tb;
  // signal declaration
  reg [3:0] test_A;
  reg [3:0] test_B;
  reg [3:0] test_C;
  reg [3:0] test_D;
  reg [1:0] test_S;
  wire [3:0] test_Z;

  // instantiate the circuit under test
  mux4to1 #(.N(4)) uut (.A(test_A), .B(test_B), .C(test_C), .D(test_D), .S(test_S), .Z(test_Z));

  // test vector generator
  initial
  begin

    test_A = 4'b1000;
    test_B = 4'b0100;
    test_C = 4'b0010;
    test_D = 4'b0001;

    test_S = 2'b00;
    #200;
    test_S = 2'b01;
```

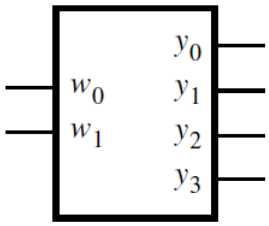
```
    #200;
    test_S = 2'b10;
    #200;
    test_S = 2'b11;
    #200;

    // stop simulation
    $stop;
  end
endmodule
```

2.2. Decodificadores

O decodificador binário recebe como entrada um número binário de tamanho *N* e decodifica em suas 2^N saídas o binário recebido na entrada. A Figura 2 ilustra um decodificador de duas entradas e quatro saídas (decodificador 2x4) com sua tabela-verdade e símbolo gráfico. Observe que, para cada possível número binário da entrada, somente uma saída fica ativada.

w_1	w_0	y_0	y_1	y_2	y_3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

(a) Truth table



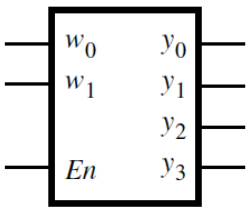
(b) Graphical symbol

Figura 2: Decodificador de duas entradas e quatro saídas.

Os diferentes circuitos podem ser implementados contendo uma entrada chamada *Enable*. Essa entrada é responsável por ativar o funcionamento do circuito. Caso seja verdadeira, o circuito funciona de acordo com sua tabela-verdade, caso contrário, o circuito gera uma saída “nula”. Veja o mesmo circuito do decodificador 2x4, porém contendo uma entrada do tipo *Enable* na Figura 3.

En	w_1	w_0	y_0	y_1	y_2	y_3
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1
0	x	x	0	0	0	0

(a) Truth table



(b) Graphical symbol

Figura 3: Decodificador 2x4 com entrada Enable.

Para implementar o funcionamento da entrada *enable* em Verilog, basta usar o comando *if-else*. O código a seguir mostra uma possibilidade para implementar o decodificador 2x4 com *enable*.

```
// dec2to4.v

module dec2to4(
  input wire enable,
        input wire [1:0] I,
        output reg [3:0] Z
);


always@*
        if(enable)
                case(I)
                        0: Z = 4'b0001;
                        1: Z = 4'b0010;
                        2: Z = 4'b0100;
                        3: Z = 4'b1000;
                endcase
        else
                Z = 4'b0000;


endmodule
```

Um *testbench* simples para o decodificador 2x4 deve testar todas as possibilidades de entrada binária com o *enable* ativado e, posteriormente, testar com o *enable* desativado. O código a seguir mostra um exemplo.

```
// dec2to4_tb.v

`timescale 1ns/10ps


module dec2to4_tb;
  // signal declaration
  reg test_enable;
  reg [1:0] test_I;
  wire [3:0] test_Z;

  // instantiate the circuit under test
  dec2to4 uut (.enable(test_enable), .I(test_I), .Z(test_Z));

  // test vector generator
  initial
  begin

    test_enable = 1'b1;
    test_I = 2'b00;
    #200;
    test_I = 2'b01;
    #200;
    test_I = 2'b10;
    #200;
    test_I = 2'b11;
    #200;

    test_enable = 1'b0;
    test_I = 2'b00;
```

```
    #200;
    test_I = 2'b01;
    #200;
    test_I = 2'b10;
    #200;
    test_I = 2'b11;
    #200;

    // stop simulation
    $stop;
  end
endmodule
```

2.3. Demultiplexadores

O circuito demultiplexador é o contrário do multiplexador. Ou seja, ele possui uma única entrada de dados e, em função de uma entrada de seleção de tamanho N, seleciona para qual das 2^N saídas o dado de entrada será enviado. O código a seguir ilustra uma possível implementação de um demux1to4.

```
// demux1to4.v
module demux1to4
#(
        parameter N = 4
)
(
        input wire [N-1:0] I,
        input wire [1:0] sel,
        output reg [N-1:0] A, B, C, D
);

always@*
begin
```

```
        A = 0; B = 0; C = 0; D = 0;
        case(sel)
                0: A = I;
                1: B = I;
                2: C = I;
                3: D = I;
        endcase
end
endmodule
```

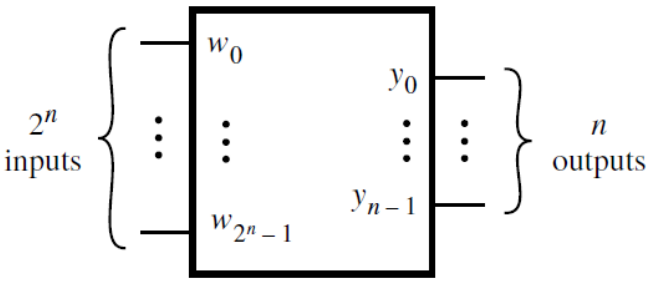
O *testbench* para o demultiplexador deve testar se a entrada é direcionada corretamente para cada saída dependendo do valor da entrada de seleção. A seguir, veja um exemplo de um *testbench* para o demux1to4.

```
// demux1to4_tb.v
`timescale 1ns/10ps

module demux1to4_tb;
        reg [3:0] test_I;
        reg [1:0] test_sel;
        wire [3:0] test_A, test_B, test_C, test_D;

        demux1to4 #(.N(4)) uut (.I(test_I), .sel(test_sel), .A(test_A), .B(test_B),
.C(test_C), .D(test_D));

        initial
        begin
                test_I = 4'b1111;
                test_sel = 2'b00;
                #200;
                test_sel = 2'b01;
                #200;
                test_sel = 2'b10;
```

```
        #200;
        test_sel = 2'b11;
        #200;

        $stop;
    end

endmodule
```

2.4. Codificador binário

A Figura 4 apresenta um codificador binário e sua tabela-verdade. Esse circuito possui diversas entradas em que somente uma delas pode estar ativa em determinado momento. A saída é a codificação da entrada em binário.



w_3	w_2	w_1	w_0	y_1	y_0
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1

Figura 4: Codificador binário e sua tabela-verdade.

Existem situações onde é colocada uma prioridade sobre as entradas do codificador, de modo que se mais de uma entrada está ativa em determinado momento, somente aquela com maior prioridade é codificada para binário na saída do codificador. A Figura 5 ilustra a tabela-verdade para o codificador de prioridade. Veja que, quando uma entrada de maior prioridade está ativa, os valores das outras entradas não importam (*don't care*). A saída *z* indica se as saídas y_1 e y_0 são válidas.

w_3	w_2	w_1	w_0	y_1	y_0	z
0	0	0	0	d	d	0
0	0	0	1	0	0	1
0	0	1	x	0	1	1
0	1	x	x	1	0	1
1	x	x	x	1	1	1

Figura 5: Tabela-verdade para o codificador de prioridade.

Em Verilog, é possível implementar o comportamento desejado para um codificador de prioridade, usando a estrutura *casex*. Essa estrutura é semelhante ao *case*, porém, permite o uso de *don't cares*, como mostra o código abaixo.

```
// priority.v
module priority(
        input wire [3:0] W,
        output reg [1:0] Y,
        output reg Z
);


always@*
begin
        Z = 1;
        casex(W)
                4'b1xxx: Y = 3;
                4'b01xx: Y = 2;
                4'b001x: Y = 1;
                4'b0001: Y = 0;
                default: begin
                                Z = 0;
                                Y = 2'bxx;
                        end
        endcase
end
endmodule
```

Pode-se fazer o *testbench* gerar todos os valores possíveis para as entradas do codificador. O exemplo abaixo mostra como fazer isso usando um *for* e uma variável inteira *i*.

```
// priority_tb.v
`timescale 1ns/10ps

module priority_tb;
	reg [3:0] test_W;
	wire [1:0] test_Y;
	wire test_Z;

	priority uut (.W(test_W), .Y(test_Y), .Z(test_Z));

	integer i = 0;
	initial
	begin
		for(i = 0; i < 16; i = i + 1) begin
			test_W = i;
			#200;
		end

		$stop;
	end

endmodule
```

2.5. Comparador

O circuito comparador de números inteiros sem sinal em Verilog pode ser feito utilizando os operadores relacionais, como por exemplo ==, < e >. Sendo assim, um código possível para um comparador capaz de verificar se dois números são iguais (*equal*), se A é maior que B (*greater than*) ou se A é menor que B (*less than*) é:

```
// compare.v
module compare
#(
	parameter N = 4
)
(
	input wire [N-1:0] A, B,
	output reg AeqB, AgtB, AltB
);

always@*
begin
	AeqB = 0;
	AgtB = 0;
	AltB = 0;
	if(A == B)
		AeqB = 1;
	else if(A > B)
		AgtB = 1;
	else
		AltB = 1;
end

endmodule
```

Um possível *testbench* seria capaz de verificar se cada uma das saídas está sendo ativada corretamente. Portanto, um código bem básico para esse teste pode ser conforme a seguir.

```
// compare_tb.v
`timescale 1ns/10ps

module compare_tb;
	reg [3:0] test_A, test_B;
	wire test_AeqB, test_AgtB, test_AltB;

	compare #(.N(4)) uut (.A(test_A), .B(test_B), .AeqB(test_AeqB), .AgtB(test_AgtB), .AltB(test_AltB));

	initial
	begin
		test_A = 12;
		test_B = 12;
		#200;

		test_A = 8;
		test_B = 12;
		#200;

		test_A = 10;
		test_B = 5;
		#200;

		$stop;
	end

endmodule
```

2.6. Unidade Lógica e Aritmética (ALU – *Arithmetic Logic Unit*)

Existem circuitos prontos que implementam diversas operações aritméticas e lógicas. Tais circuitos são muito úteis na construção de dispositivos mais complexos. Veja na Figura 6 um exemplo de como funciona a ALU 74381.

Table 4.1 The functionality of the 74381 ALU.

Operation	Inputs $s_2\ s_1\ s_0$	Outputs F
Clear	0 0 0	0 0 0 0
B−A	0 0 1	$B - A$
A−B	0 1 0	$A - B$
ADD	0 1 1	$A + B$
XOR	1 0 0	A XOR B
OR	1 0 1	A OR B
AND	1 1 0	A AND B
Preset	1 1 1	1 1 1 1

Figura 6: Funcionalidade da ALU 74381.

A ALU possui duas entradas numéricas A e B, com 4 bits cada uma e possui uma entrada de seleção de 3 bits (s2, s1 e s0) ou também chamado de *opcode*. Dependendo do valor dessa entrada de seleção, uma das 8 possíveis operações é realizada. Em Verilog, é possível implementar facilmente este circuito usando a estrutura *case* e os operadores existentes na linguagem. O código a seguir ilustra uma possível implementação.

```
// alu.v

module alu(

        input wire [3:0] A, B,

        input wire [2:0] opcode,

        output reg [3:0] S

);


always@*

        case(opcode)
```

```
            0: S = 4'b0000;
            1: S = B - A;
            2: S = A - B;
            3: S = A + B;
            4: S = A ^ B;
            5: S = A | B;
            6: S = A & B;
            7: S = 4'b1111;
        endcase

endmodule
```

O *testbench*, por sua vez, deve verificar pelo menos um caso para cada operação possível.

```
// alu_tb.v
`timescale 1ns/10ps

module alu_tb;
        reg [3:0] test_A, test_B;
        reg [2:0] test_opcode;
        wire [3:0] test_S;

        alu uut (.A(test_A), .B(test_B), .opcode(test_opcode), .S(test_S));

        integer i = 0;

        initial
        begin
                test_A = 4'b1010;
                test_B = 4'b0011;
```

```
        for(i = 0; i < 8; i = i + 1)
        begin
            test_opcode = i;
            #200;
        end

        $stop;
    end

endmodule
```

3. Passo-a-passo

Como o objetivo desta prática é conhecer a implementação em Verilog de diversos componentes MSI, você deverá implementar todos os arquivos presentes neste documento e realizar a sua simulação usando o ModelSim. Caso haja dúvida sobre o uso do ModelSim, consulte o PDF da Prática 4.

Baixe os seguintes arquivos neste link https://github.com/brabreus/GCC260-UFLA/tree/main/Pratica6 :

- mux4to1.v;
- mux4to1_tb.v;
- dec2to4.v;
- dec2to4_tb.v;
- demux1to4.v;
- demux1to4_tb.v;
- priority.v;
- priority_tb.v;
- compare.v;
- compare_tb.v;
- alu.v;
- alu_tb.v.

Em seguida, crie o projeto no ModelSim e adicione todos os arquivos. Por fim, deverá ser feita a simulação uma de cada vez para cada um dos arquivos de *testbench*. Analise os resultados da simulação se estão condizentes com os códigos e com o funcionamento teórico de cada um dos componentes. Apresente ao Professor cada simulação e envie a pasta compactada para o Campus Virtual.