UNIVERSIDADE FEDERAL DA PARAÍBA
CENTRO DE INFORMÁTICA
CONCEPÇÃO ESTRUTURADA DE CIRCUITOS INTEGRADOS

PROJETO MIPS – ENTREGA 2

THIAGO ALVES DE ARAUJO

2016019787

1 Decodificador 5-32

Abaixo podes ver o diagrama de blocos de um Decodificador genérico. Abaixo vamos desenvolver um decodificador com 5 bits de entrada e 32 bits de saída.

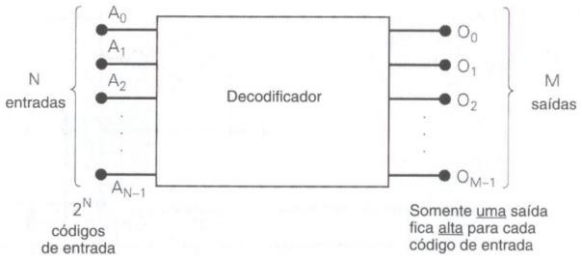

Figura 1 Decoder5_32 - Diagrama de blocos

1.1 Goldem model

Abaixo segue o trecho do código utilizado para gerar os *goldem vectors.*

```
17      //entrada de 5 bits
18      int a = 5;
19      //saida de 32 bits
20      int y = 32;
21      //Tamanho da tabela verdade.
22      int linhas = (int)pow(2,5);
23      int colunas = a + y;
24      //Matriz da tabela verdade
25      int tabelaVerdade[linhas][colunas];
26      //aux
27      int aux;
28
29      //Percorre todas as linhas
30      for(int i = 0; i < linhas; i++){
31          //Percorre as 5 primeiras colunas
32          for(int j = 0; j < 5; j++){
33              aux = (i/(int)pow(2,j))%2;
34              tabelaVerdade[i][4-j] = aux;
35          }
36
37          //Percorre as colunas restantes
38          for(int j = 5; j < colunas; j++){
39              tabelaVerdade[i][j] = 0;
40          }
41          //Y[A] = 1
42          tabelaVerdade[i][36-i] = 1;
43
44      }
```

Figura 2 Decoder5_32 - Goldem model

1.2 Goldem vector

A partir do *goldem model* criado anteriormente, geramos o seguinte vetor de testes. Da esquerda para a direita temos a entrada A de 5 bits e a saída Y de 32 bits.

```
00000_00000000000000000000000000000001
00001_00000000000000000000000000000010
00010_00000000000000000000000000000100
00011_00000000000000000000000000001000
00100_00000000000000000000000000010000
00101_00000000000000000000000000100000
00110_00000000000000000000000001000000
00111_00000000000000000000000010000000
01000_00000000000000000000000100000000
01001_00000000000000000000001000000000
01010_00000000000000000000010000000000
01011_00000000000000000000100000000000
01100_00000000000000000001000000000000
01101_00000000000000000010000000000000
01110_00000000000000000100000000000000
01111_00000000000000001000000000000000
10000_00000000000000010000000000000000
10001_00000000000000100000000000000000
10010_00000000000001000000000000000000
10011_00000000000010000000000000000000
10100_00000000000100000000000000000000
10101_00000000001000000000000000000000
10110_00000000010000000000000000000000
10111_00000000100000000000000000000000
11000_00000001000000000000000000000000
11001_00000010000000000000000000000000
11010_00000100000000000000000000000000
11011_00001000000000000000000000000000
11100_00010000000000000000000000000000
11101_00100000000000000000000000000000
11110_01000000000000000000000000000000
11111_10000000000000000000000000000000
```

Figura 3 Decoder5_32 - Goldem vectors

1.3 Testbench

Abaixo podemos ver o código em SystemVerilog responsável por testar o modulo mux2. O tempo mínimo de clock encontrado necessário para o modulo funcionar perfeitamente foi de 11000ps.

```
1   `timescale 1ns/100ps
2
3   module decoder5_32_tb;
4
5   int counter, errors, aux_error;
6   logic clk,rst;
7   logic [4:0]a;
8   logic [31:0]y, y_esperado;
9   logic [36:0]vectors[32];
10
11  decoder5_32 dut(a, y);
12
13  initial
14  begin
15     $display("Iniciando Testbench");
16     $display("|   A    |               Y                |");
17     $display("-------------------------------------------");
18     $readmemb("decoder5_32_tv.tv",vectors);
19     counter=0; errors=0;
20     rst = 1;
21     #15;
22     rst = 0;
23  end
24
25  always
26  begin
27      clk=1; #11; //O clock em 1 durar 1000ps
28     clk=0; #5; //O clock em 0 durar 1000ps
29  end
30
31  always @(posedge clk)    //Sempre (que o clock subir)
32     if(~rst)
33     begin
34        //Atribui valores do vetor nas entradas do DUT e nos valores esperados
35        a = vectors[counter][36:32];
36        y_esperado = vectors[counter][31:0];
37     end
38
```

```
39  always @(negedge clk)   //Sempre (que o clock descer)
40
41      if(~rst)
42      begin
43          aux_error = errors;
44          assert (y === y_esperado)
45      else
46      begin
47          //Mostra mensagem de erro se a saada do DUT for diferente da saada esperada
48          $display("Error: input in position %d = %b", counter, a);
49          $display("%b OPCAOO , output = %b, (%b expected)", a, y, y_esperado);
50          errors = errors + 1; //Incrementa contador de erros a cada bit errado encontrado
51      end
52
53
54      if(aux_error === errors)
55      begin
56          $display("| %b | %b | OK", a, y);
57      end
58      else
59          $display("| %b | %b | ERROR", a, y);
60          counter++;  //Incrementa contador dos vertores de teste
61
62          if(counter == $size(vectors)) //Quando os vetores de teste acabarem
63          begin
64              $display("Testes Efetuados  = %0d", counter);
65              $display("Erros Encontrados = %0d", errors);
66              #5
67              $stop;
68          end
69      end
70
71  endmodule
```

Figura 4 Decoder5_32 - Testbench

1.4 Modelo duv

Para o modulo decodificador temos como entrada A (com 5 bits) e como saída apenas o Y (com 32 bits). O comportamento deste modulo pode ser modelado com um *switch case* como é mostrado a seguir.

```
1   module decoder5_32(input logic [4:0] a, output logic [31:0] y);
2
3   always_comb
4       case(a)
5
6       5'b00000: y = 32'b00000000000000000000000000000001;
7       5'b00001: y = 32'b00000000000000000000000000000010;
8       5'b00010: y = 32'b00000000000000000000000000000100;
9       5'b00011: y = 32'b00000000000000000000000000001000;
10      5'b00100: y = 32'b00000000000000000000000000010000;
11      5'b00101: y = 32'b00000000000000000000000000100000;
12      5'b00110: y = 32'b00000000000000000000000001000000;
13      5'b00111: y = 32'b00000000000000000000000010000000;
14      5'b01000: y = 32'b00000000000000000000000100000000;
15      5'b01001: y = 32'b00000000000000000000001000000000;
16      5'b01010: y = 32'b00000000000000000000010000000000;
17      5'b01011: y = 32'b00000000000000000000100000000000;
18      5'b01100: y = 32'b00000000000000000001000000000000;
19      5'b01101: y = 32'b00000000000000000010000000000000;
20      5'b01110: y = 32'b00000000000000000100000000000000;
21      5'b01111: y = 32'b00000000000000001000000000000000;
22      5'b10000: y = 32'b00000000000000010000000000000000;
23      5'b10001: y = 32'b00000000000000100000000000000000;
24      5'b10010: y = 32'b00000000000001000000000000000000;
25      5'b10011: y = 32'b00000000000010000000000000000000;
26      5'b10100: y = 32'b00000000000100000000000000000000;
27      5'b10101: y = 32'b00000000001000000000000000000000;
28      5'b10110: y = 32'b00000000010000000000000000000000;
29      5'b10111: y = 32'b00000000100000000000000000000000;
30      5'b11000: y = 32'b00000001000000000000000000000000;
31      5'b11001: y = 32'b00000010000000000000000000000000;
32      5'b11010: y = 32'b00000100000000000000000000000000;
33      5'b11011: y = 32'b00001000000000000000000000000000;
34      5'b11100: y = 32'b00010000000000000000000000000000;
35      5'b11101: y = 32'b00100000000000000000000000000000;
36      5'b11110: y = 32'b01000000000000000000000000000000;
37      5'b11111: y = 32'b10000000000000000000000000000000;
```

```
39        default:  y = 32'bxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx;
40        endcase
41
42    endmodule
43
```

Figura 5 Decoder5_32 - Modelo duv

1.5 Simulação

Após realizar a simulação RTL podemos ver que a modelagem do decodificador 5:32 está funcionando perfeitamente.

```
# Iniciando Testbench
#  |   A    |                Y                 |
# ------------------------------------------------
# | 00000 | 00000000000000000000000000000001 | OK
# | 00001 | 00000000000000000000000000000010 | OK
# | 00010 | 00000000000000000000000000000100 | OK
# | 00011 | 00000000000000000000000000001000 | OK
# | 00100 | 00000000000000000000000000010000 | OK
# | 00101 | 00000000000000000000000000100000 | OK
# | 00110 | 00000000000000000000000001000000 | OK
# | 00111 | 00000000000000000000000010000000 | OK
# | 01000 | 00000000000000000000000100000000 | OK
# | 01001 | 00000000000000000000001000000000 | OK
# | 01010 | 00000000000000000000010000000000 | OK
# | 01011 | 00000000000000000000100000000000 | OK
# | 01100 | 00000000000000000001000000000000 | OK
# | 01101 | 00000000000000000010000000000000 | OK
# | 01110 | 00000000000000000100000000000000 | OK
# | 01111 | 00000000000000001000000000000000 | OK
# | 10000 | 00000000000000010000000000000000 | OK
# | 10001 | 00000000000000100000000000000000 | OK
# | 10010 | 00000000000001000000000000000000 | OK
# | 10011 | 00000000000010000000000000000000 | OK
# | 10100 | 00000000000100000000000000000000 | OK
# | 10101 | 00000000001000000000000000000000 | OK
# | 10110 | 00000000010000000000000000000000 | OK
# | 10111 | 00000000100000000000000000000000 | OK
# | 11000 | 00000001000000000000000000000000 | OK
# | 11001 | 00000010000000000000000000000000 | OK
# | 11010 | 00000100000000000000000000000000 | OK
# | 11011 | 00001000000000000000000000000000 | OK
# | 11100 | 00010000000000000000000000000000 | OK
# | 11101 | 00100000000000000000000000000000 | OK
# | 11110 | 01000000000000000000000000000000 | OK
# | 11111 | 10000000000000000000000000000000 | OK
# Testes Efetuados  = 32
# Erros Encontrados = 0
```

Figura 6 Decoder5_32 - Transcript da simulação RTL

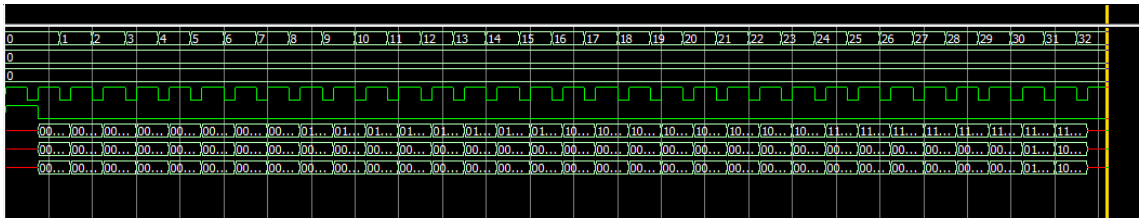

Figura 7 Decoder5_32 - Sinais de saída

Agora para a simulação em *Gate Level* vamos testar o clock para determinar a menor velocidade necessária para o modulo funcionar perfeitamente.

Iniciando o tempo de clock com 9, podemos ver no *transcript* que ocorreram diversos erros

```
27    begin
28        clk=1; #9;
29        clk=0; #5;
30    end
```

Figura 8 Decoder5_32 - Teste1 clock

```
# Error: input in position          14 = 01110
# 01110 OPCAOO , output = 00000000000000000000000000000000, (00000000000000000100000000000000 expected)
# | 01110 | 00000000000000000000000000000000 | ERROR
# | 01111 | 00000000000000000100000000000000 | OK
# Error: input in position          16 = 10000
# 10000 OPCAOO , output = 00000000000000000110000000000000, (00000000000000001000000000000000 expected)
# | 10000 | 00000000000000000110000000000000 | ERROR
# | 10001 | 00000000000000001000000000000000 | OK
# | 10010 | 00000000000000010000000000000000 | OK
# | 10011 | 00000000000010000000000000000000 | OK
# Error: input in position          20 = 10100
# 10100 OPCAOO , output = 00000000000000000000000000000000, (00000000000100000000000000000000 expected)
# | 10100 | 00000000000000000000000000000000 | ERROR
# Error: input in position          21 = 10101
# 10101 OPCAOO , output = 00000000001100000000000000000000, (00000000001000000000000000000000 expected)
# | 10101 | 00000000011000000000000000000000 | ERROR
# | 10110 | 00000000010000000000000000000000 | OK
# Error: input in position          23 = 10111
# 10111 OPCAOO , output = 00000000000000000000000000000000, (00000001000000000000000000000000 expected)
# | 10111 | 00000000000000000000000000000000 | ERROR
# Error: input in position          24 = 11000
# 11000 OPCAOO , output = 00010000100000000000000000000000, (00000001000000000000000000000000 expected)
# | 11000 | 00010000100000000000000000000000 | ERROR
# | 11001 | 00000010000000000000000000000000 | OK
# | 11010 | 00000100000000000000000000000000 | OK
# | 11011 | 00001000000000000000000000000000 | OK
# | 11100 | 00010000000000000000000000000000 | OK
# | 11101 | 00100000000000000000000000000000 | OK
# | 11110 | 01000000000000000000000000000000 | OK
# | 11111 | 10000000000000000000000000000000 | OK
# Testes Efetuados  = 32
# Erros Encontrados = 12
```

Figura 9 Decoder5_32 - Teste1 transcript

Aumentado o clock para 10, podemos ver que o número de erros diminui, porém ainda não atingimos o caso ideal.

```
27  begin
28      clk=1; #10;
29      clk=0; #5;
30  end
```

Figura 10 Decoder5_32 – Teste2 clock

```
#  |   A    |              Y               |
# -----------------------------------------
# Error: input in position          0 = 00000
# 00000 OPCAOO , output = 0000000000x00000000000000000001, (00000000000000000000000000000001 expected)
# | 00000 | 00000000000x00000000000000000001 | ERROR
# | 00001 | 00000000000000000000000000000010 | OK
# | 00010 | 00000000000000000000000000000100 | OK
# | 00011 | 00000000000000000000000000001000 | OK
# | 00100 | 00000000000000000000000000010000 | OK
# | 00101 | 00000000000000000000000000100000 | OK
# | 00110 | 00000000000000000000000001000000 | OK
# | 00111 | 00000000000000000000000010000000 | OK
# | 01000 | 00000000000000000000000100000000 | OK
# | 01001 | 00000000000000000000001000000000 | OK
# | 01010 | 00000000000000000000010000000000 | OK
# | 01011 | 00000000000000000000100000000000 | OK
# | 01100 | 00000000000000000001000000000000 | OK
# | 01101 | 00000000000000000010000000000000 | OK
# | 01110 | 00000000000000000100000000000000 | OK
# | 01111 | 00000000000000001000000000000000 | OK
# Error: input in position          16 = 10000
# 10000 OPCAOO , output = 00000000000100010000000000000000, (00000000000000001000000000000000 expected)
# | 10000 | 00000000000100010000000000000000 | ERROR
# | 10001 | 00000000000000010000000000000000 | OK
# | 10010 | 00000000000000100000000000000000 | OK
# | 10011 | 00000000000001000000000000000000 | OK
# | 10100 | 00000000000010000000000000000000 | OK
# | 10101 | 00000000001000000000000000000000 | OK
# | 10110 | 00000000010000000000000000000000 | OK
# | 10111 | 00000000100000000000000000000000 | OK
# | 11000 | 00000001000000000000000000000000 | OK
# | 11001 | 00000010000000000000000000000000 | OK
# | 11010 | 00000100000000000000000000000000 | OK
# | 11011 | 00001000000000000000000000000000 | OK
# | 11100 | 00010000000000000000000000000000 | OK
# | 11101 | 00100000000000000000000000000000 | OK
# | 11110 | 01000000000000000000000000000000 | OK
# | 11111 | 10000000000000000000000000000000 | OK
# Testes Efetuados  = 32
# Erros Encontrados = 2
```

Figura 11 Decoder5_32 – Teste2 transcript

Com o clock em 11, podemos ver que o modulo funciona perfeitamente. Logo, clock = 11 é o limite de velocidade para o Decodificador funcionar.

```
27  begin
28      clk=1; #11;
29      clk=0; #5;
30  end
31
```

Figura 12 Decoder5_32 - Teste3 clock

```
# Iniciando Testbench
#  |   A    |                Y                |
# ------------------------------------------------
#  | 00000 | 00000000000000000000000000000001 | OK
#  | 00001 | 00000000000000000000000000000010 | OK
#  | 00010 | 00000000000000000000000000000100 | OK
#  | 00011 | 00000000000000000000000000001000 | OK
#  | 00100 | 00000000000000000000000000010000 | OK
#  | 00101 | 00000000000000000000000000100000 | OK
#  | 00110 | 00000000000000000000000001000000 | OK
#  | 00111 | 00000000000000000000000010000000 | OK
#  | 01000 | 00000000000000000000000100000000 | OK
#  | 01001 | 00000000000000000000001000000000 | OK
#  | 01010 | 00000000000000000000010000000000 | OK
#  | 01011 | 00000000000000000000100000000000 | OK
#  | 01100 | 00000000000000000001000000000000 | OK
#  | 01101 | 00000000000000000010000000000000 | OK
#  | 01110 | 00000000000000000100000000000000 | OK
#  | 01111 | 00000000000000001000000000000000 | OK
#  | 10000 | 00000000000000010000000000000000 | OK
#  | 10001 | 00000000000000100000000000000000 | OK
#  | 10010 | 00000000000001000000000000000000 | OK
#  | 10011 | 00000000000010000000000000000000 | OK
#  | 10100 | 00000000000100000000000000000000 | OK
#  | 10101 | 00000000001000000000000000000000 | OK
#  | 10110 | 00000000010000000000000000000000 | OK
#  | 10111 | 00000000100000000000000000000000 | OK
#  | 11000 | 00000001000000000000000000000000 | OK
#  | 11001 | 00000010000000000000000000000000 | OK
#  | 11010 | 00000100000000000000000000000000 | OK
#  | 11011 | 00001000000000000000000000000000 | OK
#  | 11100 | 00010000000000000000000000000000 | OK
#  | 11101 | 00100000000000000000000000000000 | OK
#  | 11110 | 01000000000000000000000000000000 | OK
#  | 11111 | 10000000000000000000000000000000 | OK
# Testes Efetuados  = 32
# Erros Encontrados = 0
```

Figura 13 Decoder5_32 - Teste3 transcript