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

PROJETO MIPS – ENTREGA 3

THIAGO ALVES DE ARAUJO

2016019787

1 Banco de Registradores

Para o banco de registradores, utilizaremos 32 flopenr com o clock, reset e entrada de dados sincronizados. Cada flop tem seu próprio enable e sua própria saída Q.

1.1 Goldem model

Abaixo temos um trecho do código que gera os goldem vectors para o banco de registradores.

```
32      //Cria uma tabela com todas as possibilidades
33      for(int i = 0; i < tamanho; i++){
34          //clk
35          aux = i%2;
36          tabelaVerdade[i][0] = aux;
37          //reset
38          aux = (i/8)%2;
39          tabelaVerdade[i][1] = 1 - aux;
40          //en
41          for(int j = 2; j < 34; j++){
42              aux = (i/2)%2;
43              tabelaVerdade[i][j] = aux;
44          }
45          //d
46          aux = (i/2)%2;
47          tabelaVerdade[i][34] = aux;
48
49           for(int j = 2; j < 34; j++){
50              //Se en = 1, q = d
51              if(tabelaVerdade[i][j] == 1){
52                  tabelaVerdade[i][j+35] = 1;
53              }
54          }
```

1.2 Goldem vector

Abaixo podemos ver os goldem vectors gerados. Da esqueda para direita temos clock, reset, enable, dado e saída.

```
0_1_00000000000000000000000000000000_0_
00000000000000000000000000000000
1_1_00000000000000000000000000000000_0_
00000000000000000000000000000000
0_1_11111111111111111111111111111111_0_
00000000000000000000000000000000
1_1_11111111111111111111111111111111_0_
00000000000000000000000000000000
0_1_00000000000000000000000000000000_1_
00000000000000000000000000000000
1_1_00000000000000000000000000000000_1_
00000000000000000000000000000000
0_1_11111111111111111111111111111111_1_
00000000000000000000000000000000
1_1_11111111111111111111111111111111_1_
00000000000000000000000000000000
0_0_00000000000000000000000000000000_0_
00000000000000000000000000000000
1_0_00000000000000000000000000000000_0_
00000000000000000000000000000000
0_0_11111111111111111111111111111111_0_
00000000000000000000000000000000
1_0_11111111111111111111111111111111_0_
00000000000000000000000000000000
0_0_00000000000000000000000000000000_1_
00000000000000000000000000000000
1_0_00000000000000000000000000000000_1_
00000000000000000000000000000000
0_0_11111111111111111111111111111111_1_
00000000000000000000000000000000
1_0_11111111111111111111111111111111_1_
11111111111111111111111111111111
```

1.3 Testbench

Abaixo podemos ver o código do testbench utilizado para simulação do modulo.

```
1  `timescale 1ns/100ps
2
3  module bank_tb;
4
5  int counter, errors, aux_error;
6  logic clk,rst;
7  logic clk2,rst2;
8  logic d;
9  logic [0:31][0:0]en;
10 logic [0:31][0:0]q, q_esperado;
11 logic [0:66]vectors[16];
12
13 bank dut(.clk(clk2), .reset(rst2), .en(en), .d(d), .q(q));
14
15 initial begin
16    $readmemb("bank_tv.tv",vectors);
17    $display("Iniciando Testbench");
18    $display("|  CLK  |  RST  |              EN              |       SAIDA ESPERADA      |           SAIDA           |");
19    $display("|-------|-------|------------------------------|---------------------------|---------------------------|");
20    $display("|  CLK  |  RST  |  EN0  |  EN1  | ... |  EN31  |  QE0 | QE1   | ... | QE31 |  Q0  |  Q1  | ... |  Q31  |");
21    counter=0; errors=0;
22    rst = 1;
23    #14;
24    rst = 0;
25 end
26
27 always
28 begin
29     clk=1; #9;
30     clk=0; #5;
31 end
32
33 always @(posedge clk)
34    if(~rst)
35    begin
36       clk2 = vectors[counter][0];
37       rst2 = vectors[counter][1];
38
39       for(int i = 2; i < 34; i=i+1)begin
40          en[i-2][0:0] = vectors[counter][i];
41       end
42
43       d = vectors[counter][34];
44
45       for(int i = 35; i < 67; i=i+1)begin
46          q_esperado[i-35][0:0] = vectors[counter][i];
47       end
48
49    end
50
51 always @(negedge clk)   //Sempre (que o clock descer)
52       if(~rst)
53       begin
54          aux_error = errors;
55          assert (q === q_esperado)
56       else
57       begin
58          errors = errors + 1; //Incrementa contador de erros a cada bit errado encontrado
59          end
60
61       if(aux_error === errors)
62          $display("|  %b    |  %b    |  %b    |  %b    | ... |  %b    |  %b   |  %b   | ... |  %b   |  %b   |  %b   |
63       else
64          $display("|  %b    |  %b    |  %b    |  %b    | ... |  %b    |  %b   |  %b   | ... |  %b   |  %b   |  %b   |
65
66          counter++;  //Incrementa contador dos vertores de teste
67
68          if(counter == $size(vectors)) //Quando os vetores de teste acabarem
69          begin
70             $display("Testes Efetuados  = %0d", counter);
71             $display("Erros Encontrados = %0d", errors);
72             #15
73             $stop;
74          end
75
76       end
```

1.4 Modelo DUV

Abaixo podemos ver como os flopenr foram instanciados dentro do banco de registradores. Tanto a entrada En como a saída Q são 32 barramentos de 1bit.

```
module bank(input logic clk, input logic reset, input logic [0:31][0:0]en, input logic d, output logic [0:31][0:0]q);

   flopenr r0 (clk, reset, en[0][0:0] , d, q[0][0:0] );
   flopenr r1 (clk, reset, en[1][0:0] , d, q[1][0:0] );
   flopenr r2 (clk, reset, en[2][0:0] , d, q[2][0:0] );
   flopenr r3 (clk, reset, en[3][0:0] , d, q[3][0:0] );
   flopenr r4 (clk, reset, en[4][0:0] , d, q[4][0:0] );
   flopenr r5 (clk, reset, en[5][0:0] , d, q[5][0:0] );
   flopenr r6 (clk, reset, en[6][0:0] , d, q[6][0:0] );
   flopenr r7 (clk, reset, en[7][0:0] , d, q[7][0:0] );
   flopenr r8 (clk, reset, en[8][0:0] , d, q[8][0:0] );
   flopenr r9 (clk, reset, en[9][0:0] , d, q[9][0:0] );
   flopenr r10(clk, reset, en[10][0:0], d, q[10][0:0]);
   flopenr r11(clk, reset, en[11][0:0], d, q[11][0:0]);
   flopenr r12(clk, reset, en[12][0:0], d, q[12][0:0]);
   flopenr r13(clk, reset, en[13][0:0], d, q[13][0:0]);
   flopenr r14(clk, reset, en[14][0:0], d, q[14][0:0]);
   flopenr r15(clk, reset, en[15][0:0], d, q[15][0:0]);
   flopenr r16(clk, reset, en[16][0:0], d, q[16][0:0]);
   flopenr r17(clk, reset, en[17][0:0], d, q[17][0:0]);
   flopenr r18(clk, reset, en[18][0:0], d, q[18][0:0]);
   flopenr r19(clk, reset, en[19][0:0], d, q[19][0:0]);
   flopenr r20(clk, reset, en[20][0:0], d, q[20][0:0]);
   flopenr r21(clk, reset, en[21][0:0], d, q[21][0:0]);
   flopenr r22(clk, reset, en[22][0:0], d, q[22][0:0]);
   flopenr r23(clk, reset, en[23][0:0], d, q[23][0:0]);
   flopenr r24(clk, reset, en[24][0:0], d, q[24][0:0]);
   flopenr r25(clk, reset, en[25][0:0], d, q[25][0:0]);
   flopenr r26(clk, reset, en[26][0:0], d, q[26][0:0]);
   flopenr r27(clk, reset, en[27][0:0], d, q[27][0:0]);
   flopenr r28(clk, reset, en[28][0:0], d, q[28][0:0]);
   flopenr r29(clk, reset, en[29][0:0], d, q[29][0:0]);
   flopenr r30(clk, reset, en[30][0:0], d, q[30][0:0]);
   flopenr r31(clk, reset, en[31][0:0], d, q[31][0:0]);

endmodule
```

1.5 Simulação RTL

Abaixo podemos ver o resultado do transcript da simulação RTL. Como podemos perceber, o modulo está se comportando perfeitamente como o esperado.

```
# Iniciando Testbench
# |  CLK  |  RST  |              EN                |       SAIDA ESPERADA       |           SAIDA           |
# |-------|-------|--------------------------------|----------------------------|---------------------------|
# |  CLK  |  RST  |  EN0  |  EN1  | ... |  EN31  |  QE0  |  QE1  | ... | QE31 |  Q0  |  Q1  | ... |  Q31  |
# |  0    |  1    |  0    |  0    | ... |  0     |  0    |  0    | ... |  0   |  0   |  0   | ... |  0    | OK
# |  1    |  1    |  0    |  0    | ... |  0     |  0    |  0    | ... |  0   |  0   |  0   | ... |  0    | OK
# |  0    |  1    |  1    |  1    | ... |  1     |  0    |  0    | ... |  0   |  0   |  0   | ... |  0    | OK
# |  1    |  1    |  1    |  1    | ... |  1     |  0    |  0    | ... |  0   |  0   |  0   | ... |  0    | OK
# |  0    |  1    |  0    |  0    | ... |  0     |  0    |  0    | ... |  0   |  0   |  0   | ... |  0    | OK
# |  1    |  1    |  0    |  0    | ... |  0     |  0    |  0    | ... |  0   |  0   |  0   | ... |  0    | OK
# |  0    |  1    |  1    |  1    | ... |  1     |  0    |  0    | ... |  0   |  0   |  0   | ... |  0    | OK
# |  1    |  1    |  1    |  1    | ... |  1     |  0    |  0    | ... |  0   |  0   |  0   | ... |  0    | OK
# |  0    |  0    |  0    |  0    | ... |  0     |  0    |  0    | ... |  0   |  0   |  0   | ... |  0    | OK
# |  1    |  0    |  0    |  0    | ... |  0     |  0    |  0    | ... |  0   |  0   |  0   | ... |  0    | OK
# |  0    |  0    |  1    |  1    | ... |  1     |  0    |  0    | ... |  0   |  0   |  0   | ... |  0    | OK
# |  1    |  0    |  1    |  1    | ... |  1     |  0    |  0    | ... |  0   |  0   |  0   | ... |  0    | OK
# |  0    |  0    |  0    |  0    | ... |  0     |  0    |  0    | ... |  0   |  0   |  0   | ... |  0    | OK
# |  1    |  0    |  0    |  0    | ... |  0     |  0    |  0    | ... |  0   |  0   |  0   | ... |  0    | OK
# |  0    |  0    |  1    |  1    | ... |  1     |  0    |  0    | ... |  0   |  0   |  0   | ... |  0    | OK
# |  1    |  0    |  1    |  1    | ... |  1     |  1    |  1    | ... |  1   |  1   |  1   | ... |  1    | OK
# Testes Efetuados  = 16
# Erros Encontrados = 0
```

1.6 Simulação Gate Level

Para a simulação em Geete Level, iniciamos o clock em 6 e vamos aumentado o valor até que nenhum erro seja encontrado. Com o clock passando 6000ps em alta e 5000ps em baixa, foi encontrado 1 erro.

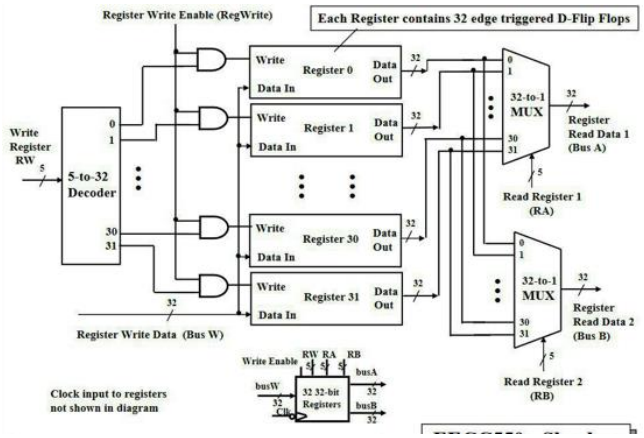
```
# Iniciando Testbench
#  |  CLK  |  RST  |                EN                |        SAIDA ESPERADA        |             SAIDA            |
#  |-------|-------|----------------------------------|------------------------------|------------------------------|
#  |  CLK  |  RST  |  EN0  |  EN1  | ... |  EN31  |  QE0  |  QE1  | ... |  QE31  |  Q0  |  Q1  | ... |  Q31  |
#  |  0    |  1    |  0    |  0    | ... |  0     |  0    |  0    | ... |  0     |  0   |  0   | ... |  0    | OK
#  |  1    |  1    |  0    |  0    | ... |  0     |  0    |  0    | ... |  0     |  0   |  0   | ... |  0    | OK
#  |  0    |  1    |  1    |  1    | ... |  1     |  0    |  0    | ... |  0     |  0   |  0   | ... |  0    | OK
#  |  1    |  1    |  1    |  1    | ... |  1     |  0    |  0    | ... |  0     |  0   |  0   | ... |  0    | OK
#  |  0    |  1    |  0    |  0    | ... |  0     |  0    |  0    | ... |  0     |  0   |  0   | ... |  0    | OK
#  |  1    |  1    |  0    |  0    | ... |  0     |  0    |  0    | ... |  0     |  0   |  0   | ... |  0    | OK
#  |  0    |  1    |  1    |  1    | ... |  1     |  0    |  0    | ... |  0     |  0   |  0   | ... |  0    | OK
#  |  1    |  1    |  1    |  1    | ... |  1     |  0    |  0    | ... |  0     |  0   |  0   | ... |  0    | OK
#  |  0    |  0    |  0    |  0    | ... |  0     |  0    |  0    | ... |  0     |  0   |  0   | ... |  0    | OK
#  |  1    |  0    |  0    |  0    | ... |  0     |  0    |  0    | ... |  0     |  0   |  0   | ... |  0    | OK
#  |  0    |  0    |  1    |  1    | ... |  1     |  0    |  0    | ... |  0     |  0   |  0   | ... |  0    | OK
#  |  1    |  0    |  1    |  1    | ... |  1     |  0    |  0    | ... |  0     |  0   |  0   | ... |  0    | OK
#  |  0    |  0    |  0    |  0    | ... |  0     |  0    |  0    | ... |  0     |  0   |  0   | ... |  0    | OK
#  |  1    |  0    |  0    |  0    | ... |  0     |  0    |  0    | ... |  0     |  0   |  0   | ... |  0    | OK
#  |  0    |  0    |  1    |  1    | ... |  1     |  0    |  0    | ... |  0     |  0   |  0   | ... |  0    | OK
#  |  1    |  0    |  1    |  1    | ... |  1     |  1    |  1    | ... |  1     |  0   |  0   | ... |  0    | ERROR
# Testes Efetuados  = 16
# Erros Encontrados = 1
```

Após alguns testes, percebemos que o tempo mínimo para o modulo funcionar perfeitamente é com o clock passando 9000ps em alta.

```
# Iniciando Testbench
#  |  CLK  |  RST  |                EN                |        SAIDA ESPERADA        |             SAIDA            |
#  |-------|-------|----------------------------------|------------------------------|------------------------------|
#  |  CLK  |  RST  |  EN0  |  EN1  | ... |  EN31  |  QE0  |  QE1  | ... |  QE31  |  Q0  |  Q1  | ... |  Q31  |
#  |  0    |  1    |  0    |  0    | ... |  0     |  0    |  0    | ... |  0     |  0   |  0   | ... |  0    | OK
#  |  1    |  1    |  0    |  0    | ... |  0     |  0    |  0    | ... |  0     |  0   |  0   | ... |  0    | OK
#  |  0    |  1    |  1    |  1    | ... |  1     |  0    |  0    | ... |  0     |  0   |  0   | ... |  0    | OK
#  |  1    |  1    |  1    |  1    | ... |  1     |  0    |  0    | ... |  0     |  0   |  0   | ... |  0    | OK
#  |  0    |  1    |  0    |  0    | ... |  0     |  0    |  0    | ... |  0     |  0   |  0   | ... |  0    | OK
#  |  1    |  1    |  0    |  0    | ... |  0     |  0    |  0    | ... |  0     |  0   |  0   | ... |  0    | OK
#  |  0    |  1    |  1    |  1    | ... |  1     |  0    |  0    | ... |  0     |  0   |  0   | ... |  0    | OK
#  |  1    |  1    |  1    |  1    | ... |  1     |  0    |  0    | ... |  0     |  0   |  0   | ... |  0    | OK
#  |  0    |  0    |  0    |  0    | ... |  0     |  0    |  0    | ... |  0     |  0   |  0   | ... |  0    | OK
#  |  1    |  0    |  0    |  0    | ... |  0     |  0    |  0    | ... |  0     |  0   |  0   | ... |  0    | OK
#  |  0    |  0    |  1    |  1    | ... |  1     |  0    |  0    | ... |  0     |  0   |  0   | ... |  0    | OK
#  |  1    |  0    |  1    |  1    | ... |  1     |  0    |  0    | ... |  0     |  0   |  0   | ... |  0    | OK
#  |  0    |  0    |  0    |  0    | ... |  0     |  0    |  0    | ... |  0     |  0   |  0   | ... |  0    | OK
#  |  1    |  0    |  0    |  0    | ... |  0     |  0    |  0    | ... |  0     |  0   |  0   | ... |  0    | OK
#  |  0    |  0    |  1    |  1    | ... |  1     |  0    |  0    | ... |  0     |  0   |  0   | ... |  0    | OK
#  |  1    |  0    |  1    |  1    | ... |  1     |  1    |  1    | ... |  1     |  1   |  1   | ... |  1    | OK
# Testes Efetuados  = 16
# Erros Encontrados = 0
```

2 Register File

Para este modulo, temos o seguinte diagrama de blocos:



Como podemos ver, são utilizados 1 decodificador 5:32, 32 portas and de 2 bits, 1 banco de registradores de 32bits e dois multiplexadores de 32bits. As conexões se dão da seguinte forma: O dado RW entra no barramento de entrada do decodificador. Cada uma das 32 saídas do decodificador é conectada a entrada de uma and que também recebe um sinal WE. Cada saída das 32 and é conectada a uma entrada de En do banco de registradores. Por fim, a saída do banco de registradores entra em dois multiplexadores de 32 bits que recebem os dados RA e RB como sinais de controle e saem com uma saída de 1bit chamada de busA e busB para cada mux respectivamente.

2.1 Goldem model

Abaixo temos um trecho do código que gera os goldem vectors para o banco de registradores.

```
28      //Cria uma tabela com todas as possibilidades
29      for(int i = 0; i < tamanho; i++){
30          aux = i%2;
31          tabelaVerdade[i][0] = aux;
32          aux = (i/4)%2;
33          tabelaVerdade[i][1] = aux;
34          aux = (i/2)%2;
35          tabelaVerdade[i][2] = aux;
36
37      }
38
39      //Determina a saída de acordo com as entradas
40      for(int i = 0; i < tamanho; i++){
41          //Se reset = 1, y = 0
42          if(tabelaVerdade[i][1] == 1){
43              tabelaVerdade[i][4] = 0;
44          }
45           //Se en = 1, y = d;
46          if(tabelaVerdade[i][2] == 1){
47              tabelaVerdade[i][4] = tabelaVerdade[i][3];
48          }
49      }
```

2.2 Goldem vector

Abaixo podemos ver os goldem vectors gerados. Da esqueda para direita temos clock, reset, enable, dado e saída.

```
00000_1_0_0_00000_00000_0_0
00001_1_1_0_00001_00001_0_0
00010_1_0_0_00010_00010_0_0
00011_1_1_0_00011_00011_0_0
00100_1_0_1_00100_00100_0_0
00101_1_1_1_00101_00101_0_0
00110_1_0_1_00110_00110_0_0
00111_1_1_1_00111_00111_0_0
01000_1_0_0_01000_01000_0_0
01001_1_1_0_01001_01001_1_0
01010_1_0_0_01010_01010_1_0
01011_1_1_0_01011_01011_0_0
01100_1_0_1_01100_01100_0_0
01101_1_1_1_01101_01101_0_0
01110_1_0_1_01110_01110_1_0
01111_1_1_1_01111_01111_0_0
10000_1_0_0_10000_10000_0_1
10001_1_1_0_10001_10001_0_1
10010_1_0_0_10010_10010_0_0
10011_1_1_0_10011_10011_0_0
10100_1_0_1_10100_10100_0_0
10101_1_1_1_10101_10101_0_0
10110_1_0_1_10110_10110_0_0
10111_1_1_1_10111_10111_0_0
11000_1_0_0_11000_11000_0_0
11001_1_1_0_11001_11001_0_0
11010_1_0_0_11010_11010_1_0
11011_1_1_0_11011_11011_0_0
11100_1_0_1_11100_11100_0_0
11101_1_1_1_11101_11101_0_0
11110_1_0_1_11110_11110_0_0
```

2.3 Testbench

Abaixo podemos ver o código do testbench utilizado para simulação do modulo.

```
`timescale 1ns/100ps

module register_tb;

int counter, errors, aux_error;
logic clk, rst;
logic [0:4]RW;
logic en, clk2, busW;
logic [0:4]RA;
logic [0:4]RB;
logic busA, busA_esperado;
logic busB, busB_esperado;

logic [0:19]vectors[32];

register dut(.RW(RW), .en(en), .clk(clk2), .busW(busW), .RA(RA), .RB(RB), .busA(busA), .busB(busB));

initial begin
   $display("Iniciando Testbench");
   $display("|   RW    | CLK | EN  | busW|   RA    |   RB    |busA |  bAes   |busB |  bBes   |");
   $display("|---------|-----|-----|-----|---------|---------|-----|---------|-----|---------|");
   $readmemb("register_tv.tv",vectors);
   counter=0; errors=0;
   rst = 1;
   #30;
   rst = 0;
end

always begin
   clk=1; #25;
   clk=0; #5;
end

always @(posedge clk)
   if(~rst) begin

      for(int i = 0; i < 5; i=i+1)begin
         RW[i] = vectors[counter][i];
      end
         en = vectors[counter][5];
         clk2 = vectors[counter][6];
         busW = vectors[counter][7];


      for(int i = 8; i < 13; i=i+1)begin
         RA[i-8] = vectors[counter][i];
      end

      for(int i = 13; i < 18; i=i+1)begin
         RB[i-13] = vectors[counter][i];
      end

      busA_esperado = vectors[counter][18];
      busB_esperado = vectors[counter][19];

   end
always @(negedge clk)

   if(~rst) begin
      aux_error = errors;
      assert(busA === busA_esperado)
   else
      begin
      errors = errors + 1; //Incrementa contador de erros a cada bit errado encontrado
      end

   if(aux_error === errors)
      $display("|  %b  |  %b  |  %b  |  %b  |  %b  |  %b  |  %b  |    %b    |  %b  |    %b    | OK", RW
   else
      $display("|  %b  |  %b  |  %b  |  %b  |  %b  |  %b  |  %b  |    %b    |  %b  |    %b    | ERRADO"
      counter++;  //Incrementa contador dos vertores de teste

   if(counter == $size(vectors)) //Quando os vetores de teste acabarem
      begin
      $display("Testes Efetuados  = %0d", counter);
      $display("Erros Encontrados = %0d", errors);
      #15
      $stop;
      end

end

endmodule
```

2.4 Modelo DUV

Abaixo podemos ver como todos os módulos foram instanciados. Assim como descrito anteriormente, cada saída de um determinado modulo entra no outro formando o caminho dos dados.

```
module register(input logic [0:4]RW, input logic en, input logic clk, input logic busW, input logic [0:4]RA, input logic [0:4]RB,
                output logic busA, output logic busB);

   logic [0:31][0:0]out_dec;
   logic [0:31][0:0]out_and;
   logic [0:31][0:0]out_bank;

   //Entra com um array de 5bits no decodificador e salva a saida em um array de 32bits
   decoder5_32 d(RW, out_dec);

   //Faz um And (bit a bit) da saida do decodificador com o en e salva em um  array de 32bits
   genvar i;

   generate
   for(i = 0; i < 32; i=i+1)begin : registradores
      and an2(out_and[i], en, out_dec[i][0:0]);
   end
   endgenerate

   //Entra com um array de 32bits  no banco e salva a saida em um array de 32bits
   bank b(clk, 0, out_and, busW, out_bank);

   //Entra com um array de 5bits e um array de 32bits (saida do banco)
   mux32 m1(RA, out_bank, busA);
   mux32 m2(RB, out_bank, busB);

endmodule
```

2.5 Simulação RTL e Gate Level

Abaixo podemos ver o transcript da simulação gate level do modulo *register file.* Depois de algumas simulações, descobrimos que o tempo mínimo para o perfeito funcionamento do modulo é de 30000ps

```
# Iniciando Testbench
#  |    RW     | CLK | EN  | busW|    RA     |    RB     |busA |   bAes    |busB |   bBes    |
#  |-----------|-----|-----|-----|-----------|-----------|-----|-----------|-----|-----------|
#  |   00000   |  0  |  1  |  0  |   00000   |   00000   |  0  |     0     |  0  |     0     | OK
#  |   00001   |  1  |  1  |  0  |   00001   |   00001   |  0  |     0     |  0  |     0     | OK
#  |   00010   |  0  |  1  |  0  |   00010   |   00010   |  0  |     0     |  0  |     0     | OK
#  |   00011   |  1  |  1  |  0  |   00011   |   00011   |  0  |     0     |  0  |     0     | OK
#  |   00100   |  0  |  1  |  1  |   00100   |   00100   |  0  |     0     |  0  |     0     | OK
#  |   00101   |  1  |  1  |  1  |   00101   |   00101   |  0  |     0     |  0  |     0     | OK
#  |   00110   |  0  |  1  |  1  |   00110   |   00110   |  0  |     0     |  0  |     0     | OK
#  |   00111   |  1  |  1  |  1  |   00111   |   00111   |  0  |     0     |  0  |     0     | OK
#  |   01000   |  0  |  1  |  0  |   01000   |   01000   |  0  |     0     |  0  |     0     | OK
#  |   01001   |  1  |  1  |  0  |   01001   |   01001   |  1  |     1     |  0  |     0     | OK
#  |   01010   |  0  |  1  |  0  |   01010   |   01010   |  1  |     1     |  0  |     0     | OK
#  |   01011   |  1  |  1  |  0  |   01011   |   01011   |  0  |     0     |  0  |     0     | OK
#  |   01100   |  0  |  1  |  1  |   01100   |   01100   |  0  |     0     |  0  |     0     | OK
#  |   01101   |  1  |  1  |  1  |   01101   |   01101   |  0  |     0     |  0  |     0     | OK
#  |   01110   |  0  |  1  |  1  |   01110   |   01110   |  1  |     1     |  0  |     0     | OK
#  |   01111   |  1  |  1  |  1  |   01111   |   01111   |  0  |     0     |  0  |     0     | OK
#  |   10000   |  0  |  1  |  0  |   10000   |   10000   |  0  |     0     |  1  |     1     | OK
#  |   10001   |  1  |  1  |  0  |   10001   |   10001   |  0  |     0     |  1  |     1     | OK
#  |   10010   |  0  |  1  |  0  |   10010   |   10010   |  0  |     0     |  0  |     0     | OK
#  |   10011   |  1  |  1  |  0  |   10011   |   10011   |  0  |     0     |  0  |     0     | OK
#  |   10100   |  0  |  1  |  1  |   10100   |   10100   |  0  |     0     |  0  |     0     | OK
#  |   10101   |  1  |  1  |  1  |   10101   |   10101   |  0  |     0     |  0  |     0     | OK
#  |   10110   |  0  |  1  |  1  |   10110   |   10110   |  0  |     0     |  0  |     0     | OK
#  |   10111   |  1  |  1  |  1  |   10111   |   10111   |  0  |     0     |  0  |     0     | OK
#  |   11000   |  0  |  1  |  0  |   11000   |   11000   |  0  |     0     |  0  |     0     | OK
#  |   11001   |  1  |  1  |  0  |   11001   |   11001   |  0  |     0     |  0  |     0     | OK
#  |   11010   |  0  |  1  |  0  |   11010   |   11010   |  1  |     1     |  0  |     0     | OK
#  |   11011   |  1  |  1  |  0  |   11011   |   11011   |  0  |     0     |  0  |     0     | OK
#  |   11100   |  0  |  1  |  1  |   11100   |   11100   |  0  |     0     |  0  |     0     | OK
#  |   11101   |  1  |  1  |  1  |   11101   |   11101   |  0  |     0     |  0  |     0     | OK
#  |   11110   |  0  |  1  |  1  |   11110   |   11110   |  0  |     0     |  0  |     0     | OK
#  |   xxxxx   |  x  |  x  |  x  |   xxxxx   |   xxxxx   |  x  |     x     |  x  |     x     | OK
# Testes Efetuados  = 32
```