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

PROJETO MIPS – ENTREGA 4

THIAGO ALVES DE ARAUJO

2016019787

Sumário

1 Tabela de figuras

2 Shift Left

Agora vamos desenvolver um deslocador de 2 bits. Abaixo podemos ver uma ilustração deste modulo.

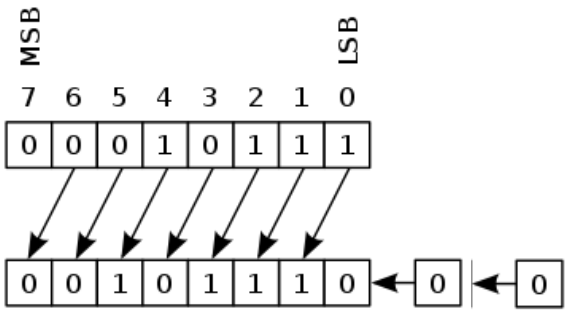


Figura 1 Shift Left - Diagrama

2.1 Goldem model

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

```
int input[32];

for(int i = 0; i < 30; i++){
    aux = input[i+2];
    input[i] = input[i+2];
    input[i+2] = aux;
}

input[30] = 0;
input[31] = 0;
```

Figura 2 Shift Left- Goldem Model

2.2 Goldem vector

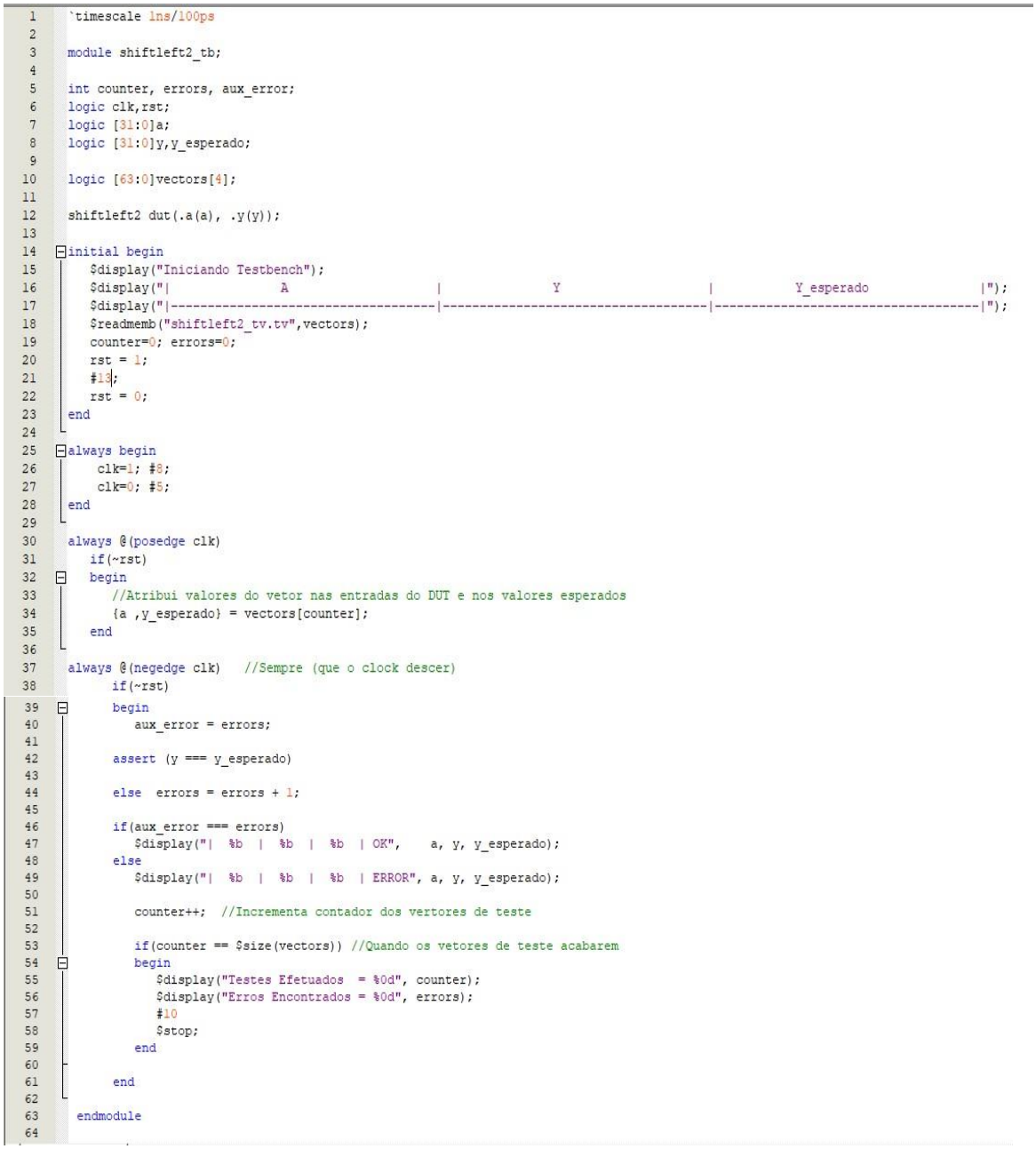
A partir do *goldem model* criado anteriormente, geramos o seguinte vetor de testes. Vamos utilizar quatro vetores de teste onde os 32 bits da esquerda são os dados de entrada e os 32 bits da direita são os dados de saída.

```
00000010010010000001001001000_
00001001001000000100100100000
00110011011010100011001001010000_
11001101101010001100100101000000
01110000110010010000001001000000_
11000011001001000000100100000000
11110010010010000001001001111_
11001001001000000100100100111100
```

Figura 3 Shift Left - Goldem vectors

2.3 Testbench

Abaixo podemos ver o código em SystemVerilog responsável por testar o deslocador.



```
`timescale 1ns/100ps

module shiftleft2_tb;

int counter, errors, aux_error;
logic clk,rst;
logic [31:0]a;
logic [31:0]y,y_esperado;

logic [63:0]vectors[4];

shiftleft2 dut(.a(a), .y(y));

initial begin
   $display("Iniciando Testbench");
   $display("|              A                  |              Y                  |          Y_esperado             |");
   $display("|---------------------------------|---------------------------------|---------------------------------|");
   $readmemb("shiftleft2_tv.tv",vectors);
   counter=0; errors=0;
   rst = 1;
   #13;
   rst = 0;
end

always begin
    clk=1; #8;
    clk=0; #5;
end

always @(posedge clk)
   if(~rst)
   begin
      //Atribui valores do vetor nas entradas do DUT e nos valores esperados
      {a ,y_esperado} = vectors[counter];
   end

always @(negedge clk)   //Sempre (que o clock descer)
      if(~rst)
      begin
         aux_error = errors;

      assert (y === y_esperado)

      else  errors = errors + 1;

      if(aux_error === errors)
         $display("|  %b  |  %b  |  %b  | OK",    a, y, y_esperado);
      else
         $display("|  %b  |  %b  |  %b  | ERROR", a, y, y_esperado);

         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);
            #10
            $stop;
         end

      end

 endmodule
```

Figura 4 Shift Left - Testbench

2.4 Modelo duv

Para o modulo Shift Left temos como entrada um barramento de 32 bits e como saída os mesmos 32 bits deslocados dois bits a esquerda. Para isso, basta pegarmos os 30 bits mais significativos e concatenarmos eles com 2 bits zero.

```
1  module shiftleft2 (input logic [31:0] a, output logic [31:0] y);
2
3  assign y = {a[29:0], 2'b00};
4
5  endmodule
```

Figura 5 Shift Left - Modelo duv

2.5 Simulação RTL

Com o modulo pronto, podemos iniciar a simulação RTL. Como podemos ver na imagem abaixo, o modulo está se comportando como o esperado.

```
# Iniciando Testbench
# |                A                |                Y                |           Y_esperado            |
# |---------------------------------|---------------------------------|---------------------------------|
# |  00000010010010000000100100100  |  00001001001000000010010010000  |  00001001001000000010010010000  | OK
# |  00110011011010100011001001010  |  11001101101010001100100101000  |  11001101101010001100100101000  | OK
# |  01110000110010010000001001000  |  11000011001001000000100100000  |  11000011001001000000100100000  | OK
# |  11110010010010000001001001111  |  11001001001000000100100100111  |  11001001001000000100100100111  | OK
# Testes Efetuados  = 4
# Erros Encontrados = 0
```

Figura 6 Shift Left - Transcript da simulação RTL

Também podemos ver o *RTL view* gerado a partir do modulo testado.

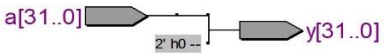


Figura 7 Shift Left - RTL view

2.6 Simulação *Gate Level*

Agora iniciamos a simulação Gate Level. Iniciamos o clock com um valor consideravelmente baixo e vamos subindo até que nenhum erro seja encontrado. Para este modulo, foram feitos dois testes. É possível observar que à medida que o clock aumenta, o numero de erros vai diminuindo.

```
# Iniciando Testbench
# |                A                |                Y                |           Y_esperado            |
# |---------------------------------|---------------------------------|---------------------------------|
# |  00000010010010000000100100100  |  0000x00100100000010010010x100000  |  00001001001000000010010010000  | ERROR
# |  00110011011010100011001001010  |  11001101101010001100100100000000  |  11001101101010001100100101000  | ERROR
# |  01110000110010010000001001000  |  11001011001001000000100101000000  |  11000011001001000000100100000  | ERROR
# |  11110010010010000001001001111  |  11000001001000000100100100111100  |  11001001001000000100100100111  | ERROR
# Testes Efetuados  = 4
# Erros Encontrados = 4
```

Figura 8 Shift Left - clk #7

```
# Iniciando Testbench
# |                A                |                Y                |           Y_esperado            |
# |---------------------------------|---------------------------------|---------------------------------|
# |  00000010010010000000100100100  |  00001001001000000010010010000  |  00001001001000000010010010000  | OK
# |  00110011011010100011001001010  |  11001101101010001100100101000  |  11001101101010001100100101000  | OK
# |  01110000110010010000001001000  |  11000011001001000000100100000  |  11000011001001000000100100000  | OK
# |  11110010010010000001001001111  |  11001001001000000100100100111  |  11001001001000000100100100111  | OK
# Testes Efetuados  = 4
# Erros Encontrados = 0
```

Figura 9 Shift Left - clk #8

Por fim, podemos ver os sinais de entrada e saída do modulo.

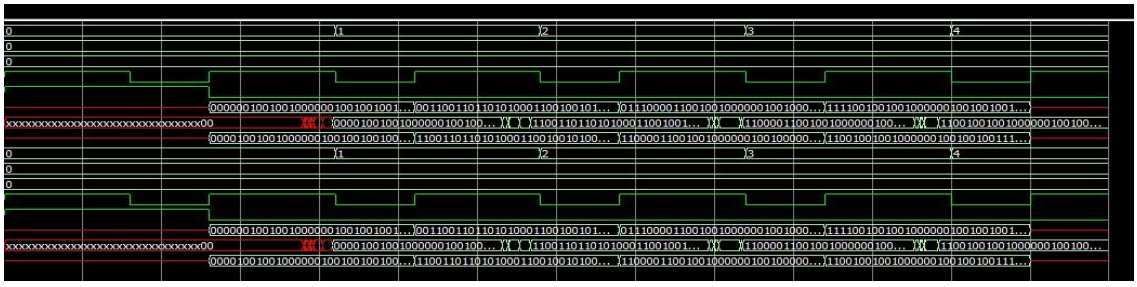


Figura 10 Shift Left - Sinais de entrada/saída

3 Sign Extend

Agora vamos desenvolver um extensor de sinal. Abaixo podemos ver uma ilustração deste modulo. Como podemos ver, o bit mais significativo é copiado 16 vezes e concatenado com o sinal original.

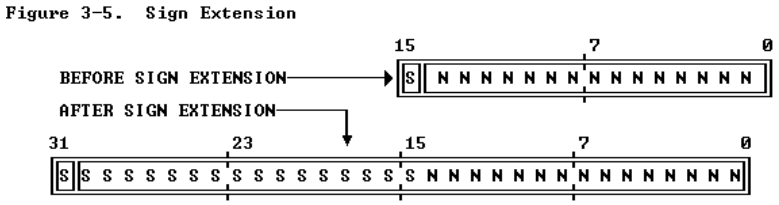


Figura 11 Sign extend - Diagrama

3.1 Goldem model

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

```
int input[16];
int output[32];

for(int i = 0; i < 32; i++){

    if(i < 16) output[i] = input[0];
    else       output[i] = input[i-16];
}
```

Figura 12 Sign extend- Goldem Model

3.2 Goldem vector

A partir do *goldem model* criado anteriormente, geramos o seguinte vetor de testes. Vamos utilizar dezesseis vetores de teste onde os 16 bits da esquerda são os dados de entrada e os 32 bits da direita são os dados de saída.

```
0000000000000001_00000000000000000000000000000001
0000000000000010_00000000000000000000000000000010
0000000000000100_00000000000000000000000000000100
0000000000001000_00000000000000000000000000001000
0000000000010000_00000000000000000000000000010000
0000000000100000_00000000000000000000000000100000
0000000001000000_00000000000000000000000001000000
0000000010000000_00000000000000000000000010000000
0000000100000000_00000000000000000000000100000000
0000001000000000_00000000000000000000001000000000
0000010000000000_00000000000000000000010000000000
0000100000000000_00000000000000000000100000000000
0001000000000000_00000000000000000001000000000000
0010000000000000_00000000000000000010000000000000
0100000000000000_00000000000000000100000000000000
1000000000000000_11111111111111111000000000000000
```

Figura 13 Sign extend - Goldem vectors

3.3 Testbench

Abaixo podemos ver o código em *SystemVerilog* responsável por testar modulo.

```
`timescale 1ns/100ps

module signextend_tb;

int counter, errors, aux_error;
logic clk,rst;
logic [15:0]a;
logic [31:0]y,y_esperado;

logic [47:0]vectors[16];

signextend dut(.a(a), .y(y));

initial begin
   $display("Iniciando Testbench");
   $display("|         A          |              Y                    |          Y_esperado               |");
   $display("|--------------------|-----------------------------------|-----------------------------------|");
   $readmemb("signextend_tv.tv",vectors);
   counter=0; errors=0;
   rst = 1;
   #12;
   rst = 0;
end

always begin
    clk=1; #7;
    clk=0; #5;
end

always @(posedge clk)
   if(~rst)
   begin
      //Atribui valores do vetor nas entradas do DUT e nos valores esperados
      {a ,y_esperado} = vectors[counter];
   end

always @(negedge clk)   //Sempre (que o clock descer)
      if(~rst)
      begin
         aux_error = errors;

      assert (y === y_esperado)

      else  errors = errors + 1;

      if(aux_error === errors)
         $display("|  %b  |  %b  |  %b  | OK",    a, y, y_esperado);
      else
         $display("|  %b  |  %b  |  %b  | ERROR", a, y, y_esperado);

         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);
            #10
            $stop;
         end

      end

endmodule
```

Figura 14 Sign extend - Testbench

3.4 Modelo duv

Para o modulo extensor de sinal, temos como entrada um barramento de 16 bits e como saída os mesmo sinal de entrada estendido para 32 como descrito anteriormente.

```
module signextend(input logic [15:0] a, output logic [31:0] y);

assign y = {{16{a[15]}}, a};

endmodule
```

Figura 15 Sign extend - Modelo duv

3.5 Simulação RTL

Com o modulo pronto, podemos iniciar a simulação RTL. Como podemos ver na imagem abaixo, o modulo está se comportando como o esperado.

```
# Iniciando Testbench
# |         A          |                Y                 |           Y_esperado             |
# |--------------------|----------------------------------|----------------------------------|
# |  0000000000000001  |  00000000000000000000000000000001  |  00000000000000000000000000000001  | OK
# |  0000000000000010  |  00000000000000000000000000000010  |  00000000000000000000000000000010  | OK
# |  0000000000000100  |  00000000000000000000000000000100  |  00000000000000000000000000000100  | OK
# |  0000000000001000  |  00000000000000000000000000001000  |  00000000000000000000000000001000  | OK
# |  0000000000010000  |  00000000000000000000000000010000  |  00000000000000000000000000010000  | OK
# |  0000000000100000  |  00000000000000000000000000100000  |  00000000000000000000000000100000  | OK
# |  0000000001000000  |  00000000000000000000000001000000  |  00000000000000000000000001000000  | OK
# |  0000000010000000  |  00000000000000000000000010000000  |  00000000000000000000000010000000  | OK
# |  0000000100000000  |  00000000000000000000000100000000  |  00000000000000000000000100000000  | OK
# |  0000001000000000  |  00000000000000000000001000000000  |  00000000000000000000001000000000  | OK
# |  0000010000000000  |  00000000000000000000010000000000  |  00000000000000000000010000000000  | OK
# |  0000100000000000  |  00000000000000000000100000000000  |  00000000000000000000100000000000  | OK
# |  0001000000000000  |  00000000000000000001000000000000  |  00000000000000000001000000000000  | OK
# |  0010000000000000  |  00000000000000000010000000000000  |  00000000000000000010000000000000  | OK
# |  0100000000000000  |  00000000000000000100000000000000  |  00000000000000000100000000000000  | OK
# |  1000000000000000  |  11111111111111111000000000000000  |  11111111111111111000000000000000  | OK
# Testes Efetuados  = 16
# Erros Encontrados = 0
```

Figura 16 Sign extend - Transcript da simulação RTL

Também podemos ver o *RTL view* gerado a partir do modulo testado.

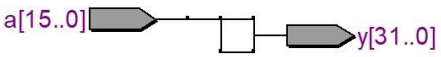


Figura 17 Sign extend - RTL view

3.6 Simulação *Gate Level*

Agora iniciamos a simulação Gate Level. Iniciamos o clock com um valor consideravelmente baixo e vamos subindo até que nenhum erro seja encontrado. Para este modulo, foram feitos três testes. É possível observar que à medida que o clock aumenta, o número de erros vai diminuindo.

```
# Iniciando Testbench
# |         A          |                Y                 |           Y_esperado             |
# |--------------------|----------------------------------|----------------------------------|
# |  0000000000000001  |  0xxxxx00xx00000xxx00000000000001  |  00000000000000000000000000000001  | ERROR
# |  0000000000000010  |  00000000000000000000000000000010  |  00000000000000000000000000000010  | OK
# |  0000000000000100  |  00000000000000000000000000000100  |  00000000000000000000000000000100  | OK
# |  0000000000001000  |  00000000000000000000000000001000  |  00000000000000000000000000001000  | OK
# |  0000000000010000  |  00000000000000000000000000010000  |  00000000000000000000000000010000  | OK
# |  0000000000100000  |  00000000000000000000000000100000  |  00000000000000000000000000100000  | OK
# |  0000000001000000  |  00000000000000000000000001000000  |  00000000000000000000000001000000  | OK
# |  0000000010000000  |  00000000000000000000000010000000  |  00000000000000000000000010000000  | OK
# |  0000000100000000  |  00000000000000000000000100000000  |  00000000000000000000000100000000  | OK
# |  0000001000000000  |  00000000000000000000001000000000  |  00000000000000000000001000000000  | OK
# |  0000010000000000  |  00000000000000000000010000000000  |  00000000000000000000010000000000  | OK
# |  0000100000000000  |  00000000000000000000100000000000  |  00000000000000000000100000000000  | OK
# |  0001000000000000  |  00000000000000000001000000000000  |  00000000000000000001000000000000  | OK
# |  0010000000000000  |  00000000000000000010000000000000  |  00000000000000000010000000000000  | OK
# |  0100000000000000  |  00000000000000000000000000000000  |  00000000000000000100000000000000  | ERROR
# |  1000000000000000  |  11110111111111111100000000000000  |  11111111111111111000000000000000  | ERROR
# Testes Efetuados  = 16
# Erros Encontrados = 3
```

Figura 18 Sign extend - clk #7

```
# Iniciando Testbench
# |         A          |                Y                 |          Y_esperado                |
# |--------------------|----------------------------------|------------------------------------|
# |  0000000000000001  |  0000x0000000000000x00000000000001  |  00000000000000000000000000000001  | ERROR
# |  0000000000000010  |  00000000000000000000000000000010  |  00000000000000000000000000000010  | OK
# |  0000000000000100  |  00000000000000000000000000000100  |  00000000000000000000000000000100  | OK
# |  0000000000001000  |  00000000000000000000000000001000  |  00000000000000000000000000001000  | OK
# |  0000000000010000  |  00000000000000000000000000010000  |  00000000000000000000000000010000  | OK
# |  0000000000100000  |  00000000000000000000000000100000  |  00000000000000000000000000100000  | OK
# |  0000000001000000  |  00000000000000000000000001000000  |  00000000000000000000000001000000  | OK
# |  0000000010000000  |  00000000000000000000000010000000  |  00000000000000000000000010000000  | OK
# |  0000000100000000  |  00000000000000000000000100000000  |  00000000000000000000000100000000  | OK
# |  0000001000000000  |  00000000000000000000001000000000  |  00000000000000000000001000000000  | OK
# |  0000010000000000  |  00000000000000000000010000000000  |  00000000000000000000010000000000  | OK
# |  0000100000000000  |  00000000000000000000100000000000  |  00000000000000000000100000000000  | OK
# |  0001000000000000  |  00000000000000000001000000000000  |  00000000000000000001000000000000  | OK
# |  0010000000000000  |  00000000000000000010000000000000  |  00000000000000000010000000000000  | OK
# |  0100000000000000  |  00000000000000000100000000000000  |  00000000000000000100000000000000  | OK
# |  1000000000000000  |  11111111111111111000000000000000  |  11111111111111111000000000000000  | ERROR
# Testes Efetuados  = 16
# Erros Encontrados = 2
```

Figura 19 Sign extend - clk #8

```
# Iniciando Testbench
# |         A          |                Y                 |          Y_esperado                |
# |--------------------|----------------------------------|------------------------------------|
# |  0000000000000001  |  00000000000000000000000000000001  |  00000000000000000000000000000001  | OK
# |  0000000000000010  |  00000000000000000000000000000010  |  00000000000000000000000000000010  | OK
# |  0000000000000100  |  00000000000000000000000000000100  |  00000000000000000000000000000100  | OK
# |  0000000000001000  |  00000000000000000000000000001000  |  00000000000000000000000000001000  | OK
# |  0000000000010000  |  00000000000000000000000000010000  |  00000000000000000000000000010000  | OK
# |  0000000000100000  |  00000000000000000000000000100000  |  00000000000000000000000000100000  | OK
# |  0000000001000000  |  00000000000000000000000001000000  |  00000000000000000000000001000000  | OK
# |  0000000010000000  |  00000000000000000000000010000000  |  00000000000000000000000010000000  | OK
# |  0000000100000000  |  00000000000000000000000100000000  |  00000000000000000000000100000000  | OK
# |  0000001000000000  |  00000000000000000000001000000000  |  00000000000000000000001000000000  | OK
# |  0000010000000000  |  00000000000000000000010000000000  |  00000000000000000000010000000000  | OK
# |  0000100000000000  |  00000000000000000000100000000000  |  00000000000000000000100000000000  | OK
# |  0001000000000000  |  00000000000000000001000000000000  |  00000000000000000001000000000000  | OK
# |  0010000000000000  |  00000000000000000010000000000000  |  00000000000000000010000000000000  | OK
# |  0100000000000000  |  00000000000000000100000000000000  |  00000000000000000100000000000000  | OK
# |  1000000000000000  |  11111111111111111000000000000000  |  11111111111111111000000000000000  | OK
# Testes Efetuados  = 16
# Erros Encontrados = 0
```

Figura 20 Sign extend - clk #9

Por fim, podemos ver os sinais de entrada e saída do modulo.

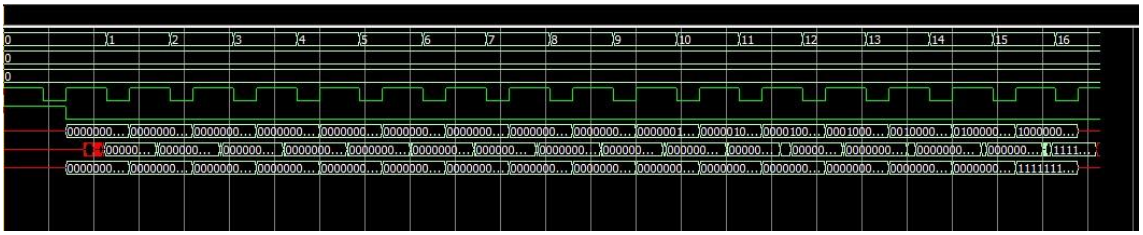


Figura 21 Sign extend - Sinais de entrada/saída

4 Shift PC

Agora vamos desenvolver um extensor de sinal. O funcionamento deste modulo é bem simples e consiste apenas em concatenar dois sinais de entrada com dois bits 0.

4.1 Goldem model

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

```
int pc[4];
int a[26];
int out[32];

for(int i = 0; i < 30; i++){
    if(i < 4) out[i] = pc[i];
    else out[i] = a[i-4];
}

out[30] = 0;
out[31] = 0;
```

Figura 22 Shift PC- Goldem Model

4.2 Goldem vector

A partir do *goldem model* criado anteriormente, geramos o seguinte vetor de testes. Vamos utilizar quatro vetores de teste onde da esquerda para a direta temos 4 bits do pc, 26 bits de entrada e 32 bits de saída.

```
0000_11111111111111111111111111_
00001111111111111111111111111100
1111_11111111111111111111111111_
11111111111111111111111111111100
0000_00000000000000000000000000_
00000000000000000000000000000000
1111_00000000000000000000000000_
11110000000000000000000000000000
```

Figura 23 Shift PC - Goldem vectors

4.3 Testbench

Abaixo podemos ver o código em *SystemVerilog* responsável por testar modulo.

```
1   `timescale 1ns/100ps
2
3   module shiftpc_tb;
4
5   int counter, errors, aux_error;
6   logic clk,rst;
7   logic [25:0]a;
8   logic [3:0]pc;
9   logic [31:0]y,y_esperado;
10
11  logic [61:0]vectors[4];
12
13  shiftpc dut(.pc(pc), .a(a), .y(y));
14
15  initial begin
16     $display("Iniciando Testbench");
17     $display("|             A              |   PC   |              Y                   |          Y_esperado              |");
18     $display("|----------------------------|--------|----------------------------------|----------------------------------|");
19     $readmemb("shiftpc_tv.tv",vectors);
20     counter=0; errors=0;
21     rst = 1;
22     #14;
23     rst = 0;
24  end
25
26  always begin
27      clk=1; #9;
28      clk=0; #5;
29  end
30
31  always @(posedge clk)
32     if(~rst)
33     begin
34        {pc, a ,y_esperado} = vectors[counter];
35     end
36
37  always @(negedge clk)   //Sempre (que o clock descer)
38        if(~rst)
```

```
39      begin
40         aux_error = errors;
41
42      assert (y === y_esperado)
43
44      else  errors = errors + 1;
45
46      if(aux_error === errors)
47         $display("|  %b  |  %b  |  %b  |  %b  | OK",    a, pc, y, y_esperado);
48      else
49         $display("|  %b  |  %b  |  %b  |  %b  | ERROR", a, pc, y, y_esperado);
50
51         counter++;  //Incrementa contador dos vertores de teste
52
53         if(counter == $size(vectors)) //Quando os vetores de teste acabarem
54         begin
55            $display("Testes Efetuados  = %0d", counter);
56            $display("Erros Encontrados = %0d", errors);
57            #10
58            $stop;
59         end
60
61      end
62
63   endmodule
64
```

Figura 24 Shift PC - Testbench

4.4 Modelo duv

Para o modulo extensor de sinal, temos como entrada 4 bits para o contador de programa e 26 bits para a entrada A. A saída é um barramento de 32 bits formado pela concatenação de pc, a e dois bits 0.

```
1   module shiftpc( input logic [3:0] pc, input logic [25:0] a, output logic [31:0] s);
2
3   assign s = {pc,a,2'b00};
4
5   endmodule
6
```

Figura 25 Shift PC - Modelo duv

4.5 Simulação RTL

Com o modulo pronto, podemos iniciar a simulação RTL. Como podemos ver na imagem abaixo, o modulo está se comportando como o esperado.

```
# Iniciando Testbench
# |                A                |   PC   |                Y                 |           Y_esperado             |
# |---------------------------------|--------|----------------------------------|----------------------------------|
# |  11111111111111111111111111  |  0000  |  00001111111111111111111111111100  |  00001111111111111111111111111100  | OK
# |  11111111111111111111111111  |  1111  |  11111111111111111111111111111100  |  11111111111111111111111111111100  | OK
# |  00000000000000000000000000  |  0000  |  00000000000000000000000000000000  |  00000000000000000000000000000000  | OK
# |  00000000000000000000000000  |  1111  |  11110000000000000000000000000000  |  11110000000000000000000000000000  | OK
# Testes Efetuados  = 4
# Erros Encontrados = 0
```

Figura 26 Shift PC - Transcript da simulação RTL

Também podemos ver o *RTL view* gerado a partir do modulo testado.

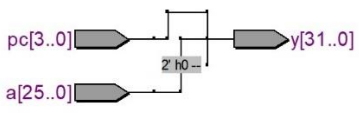


Figura 27 Shift PC - RTL view

4.6 Simulação *Gate Level*

Agora iniciamos a simulação Gate Level. Iniciamos o clock com um valor consideravelmente baixo e vamos subindo até que nenhum erro seja encontrado. Para este modulo, foram feitos três testes. É possível observar que à medida que o clock aumenta, o número de erros vai diminuindo.

```
# Iniciando Testbench
# |                A                |   PC   |                Y                 |           Y_esperado                 |
# |---------------------------------|--------|----------------------------------|--------------------------------------|
# |  11111111111111111111111111111  |  0000  |  x0x011111x111111x111111111111100  |  00001111111111111111111111111100  | ERROR
# |  11111111111111111111111111111  |  1111  |  01011111111111111111111111111100  |  11111111111111111111111111111100  | ERROR
# |  00000000000000000000000000000  |  0000  |  10100000010000001000000000000000  |  00000000000000000000000000000000  | ERROR
# |  00000000000000000000000000000  |  1111  |  01010000000000000000000000000000  |  11110000000000000000000000000000  | ERROR
# Testes Efetuados  = 4
# Erros Encontrados = 4
```

Figura 28 Shift PC - clk #7

```
# Iniciando Testbench
# |                A                |   PC   |                Y                 |           Y_esperado                 |
# |---------------------------------|--------|----------------------------------|--------------------------------------|
# |  11111111111111111111111111111  |  0000  |  x0001111111111111111111111111100  |  00001111111111111111111111111100  | ERROR
# |  11111111111111111111111111111  |  1111  |  11111111111111111111111111111100  |  11111111111111111111111111111100  | OK
# |  00000000000000000000000000000  |  0000  |  10000000000000000000000000000000  |  00000000000000000000000000000000  | ERROR
# |  00000000000000000000000000000  |  1111  |  11110000000000000000000000000000  |  11110000000000000000000000000000  | OK
# Testes Efetuados  = 4
# Erros Encontrados = 2
```

Figura 29 Shift PC - clk #8

```
# Iniciando Testbench
# |                A                |   PC   |                Y                 |           Y_esperado                 |
# |---------------------------------|--------|----------------------------------|--------------------------------------|
# |  11111111111111111111111111111  |  0000  |  00001111111111111111111111111100  |  00001111111111111111111111111100  | OK
# |  11111111111111111111111111111  |  1111  |  11111111111111111111111111111100  |  11111111111111111111111111111100  | OK
# |  00000000000000000000000000000  |  0000  |  00000000000000000000000000000000  |  00000000000000000000000000000000  | OK
# |  00000000000000000000000000000  |  1111  |  11110000000000000000000000000000  |  11110000000000000000000000000000  | OK
# Testes Efetuados  = 4
# Erros Encontrados = 0
```

Figura 30 Shift PC - clk #9

Por fim, podemos ver os sinais de entrada e saída do modulo.

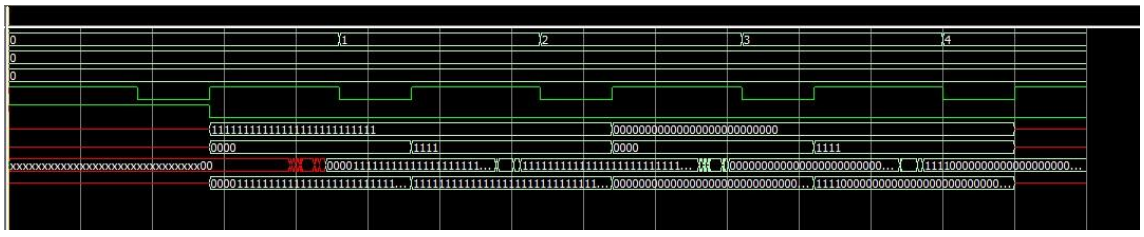


Figura 31 Shift PC - Sinais de entrada/saída