UNIVERSIDADE FEDERAL DA PARAÍBA - UFPB
CENTRO DE INFORMÁTICA

Relatório Terceira Avaliação

Disciplina: Circuitos Lógicos II

Professor:

Jose Antonio Gomes de Lima

Equipe :

Pedro Ricardo Cavalcante Silva Filho 20200126968

João Pessoa, PB
2023

SUMÁRIO

1. Flip-flop tipo D

Esta seção oferece uma explanação abrangente do circuito flip-flop tipo D, um componente crucial em sistemas digitais para armazenamento de informações. Projetado para operar como um elemento de memória de um bit, o circuito baseia-se em uma configuração que permite a transferência de dados síncrona ou assíncrona entre diferentes partes de um circuito digital. A principal função deste circuito é manter e estabilizar o valor de saída, permitindo assim a sincronização precisa dos dados em operações críticas.

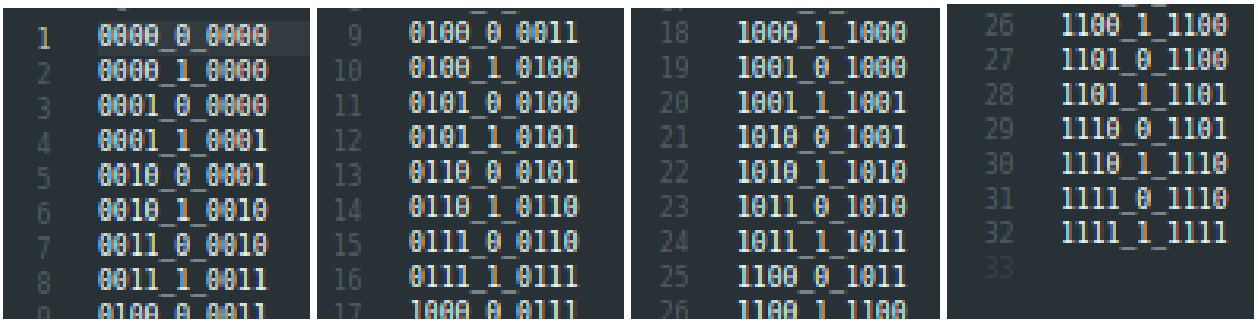
O propósito central desta seção é destacar a notável capacidade e eficiência do circuito flip-flop tipo D em reter e transmitir informações de maneira consistente e confiável. A confiabilidade e a estabilidade operacional são garantidas em uma ampla gama de configurações, enfatizando a importância e a aplicabilidade desse componente em uma variedade de contextos digitais.

1.1. Modelo de referência

O modelo de referência é um arquivo .tv que contém vetores de teste a serem utilizados durante o testbench. Esses vetores correspondem, essencialmente, a linhas de uma tabela verdade. Para gerar este arquivo, foi desenvolvido um breve programa em C, demonstrado na imagem a seguir:

```c
#include <stdio.h>

int main() {
    FILE *fp;
    fp = fopen("flipflop.tv", "w"); // abre o arquivo para escrita
    if(fp == NULL) {
        printf("Erro ao abrir o arquivo.\n");
        return 1;
    }

    int a, b, c, d, clock;
    int anterior_a, anterior_b, anterior_c, anterior_d;

    anterior_a = 0;
    anterior_b = 0;
    anterior_c = 0;
    anterior_d = 0;

    for(a=0; a<=1; a++) {
        for(b=0; b<=1; b++) {
            for(c=0; c<=1; c++) {
                for(d=0; d<=1; d++) {
                    clock = 0; // gera o Clock para os Flip-Flops (sens\x9avel \x8 borda de subida)
                    fprintf(fp, "%d%d%d%d_%d_%d%d%d%d\n", a, b, c, d, clock, anterior_a, anterior_b, anterior_c, anterior_d);  // escreve a combina\x8ao pela primeira vez
                    clock = 1; // gera o clock para os Flip-Flops (sens\x9avel \x8 borda de subida)
                    fprintf(fp, "%d%d%d%d_%d_%d%d%d%d\n", a, b, c, d, clock, a, b, c, d); // escreve a combina\x8\x8o pela segunda vez
                    anterior_a = a; // armazena a combina\x8\x8o atual
                    anterior_b = b;
                    anterior_c = c;
                    anterior_d = d;

                }
            }
        }
    }

    fclose(fp); // fecha o arquivo
    printf("Arquivo gerado com sucesso.\n");
    return 0;
}
```

Este código gera um arquivo chamado "flipflop.tv" que contém 32 linhas de vetores de teste em binário para o modelo de referência do circuito flip-flop tipo D.



Cada linha no arquivo .tv representa um vetor de teste para o circuito flip-flop tipo D. Os quatro primeiros dígitos indicam o valor de D em binário de 4 bits, o do meio representa o sinal de clock, indica o momento em que a operação do flip-flop deve ocorrer, enquanto os quatro dígitos seguintes representam o valor da saída Q esperada, em binário de 4 bits.

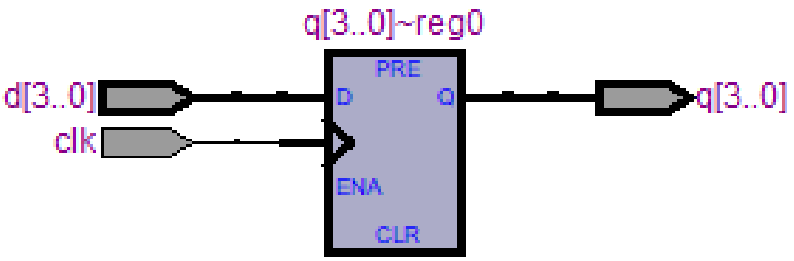
1.2. Descrição do Hardware

A descrição de hardware em SystemVerilog (flipflop.sv) do módulo "Flip-flop Tipo D" é de significativa relevância. Projetado para operar como um componente fundamental em sistemas digitais, o módulo é responsável por armazenar e transmitir dados de 4 bits de entrada de forma síncrona. O circuito implementado atua com eficiência e confiabilidade, assegurando a transferência precisa de informações entre diferentes partes de um sistema digital.

A funcionalidade integral e o comportamento operacional do módulo podem ser compreendidos plenamente por meio da descrição detalhada no diagrama esquemático fornecido abaixo. O flip-flop tipo D foi cuidadosamente otimizado para garantir uma resposta consistente e precisa, demonstrando sua eficácia em uma variedade de contextos digitais que requerem armazenamento e transferência de dados de 4 bits.

```
1   module flipflop
2   (input logic [3:0] d,
3    input logic clk,
4    output logic [3:0] q);
5
6    always_ff @ (posedge clk)
7
8    q <= d;
9
10  endmodule
```

Após a compilação deste código no Quartus II podemos ver a visualização RTL deste módulo:



1.3. Testbench

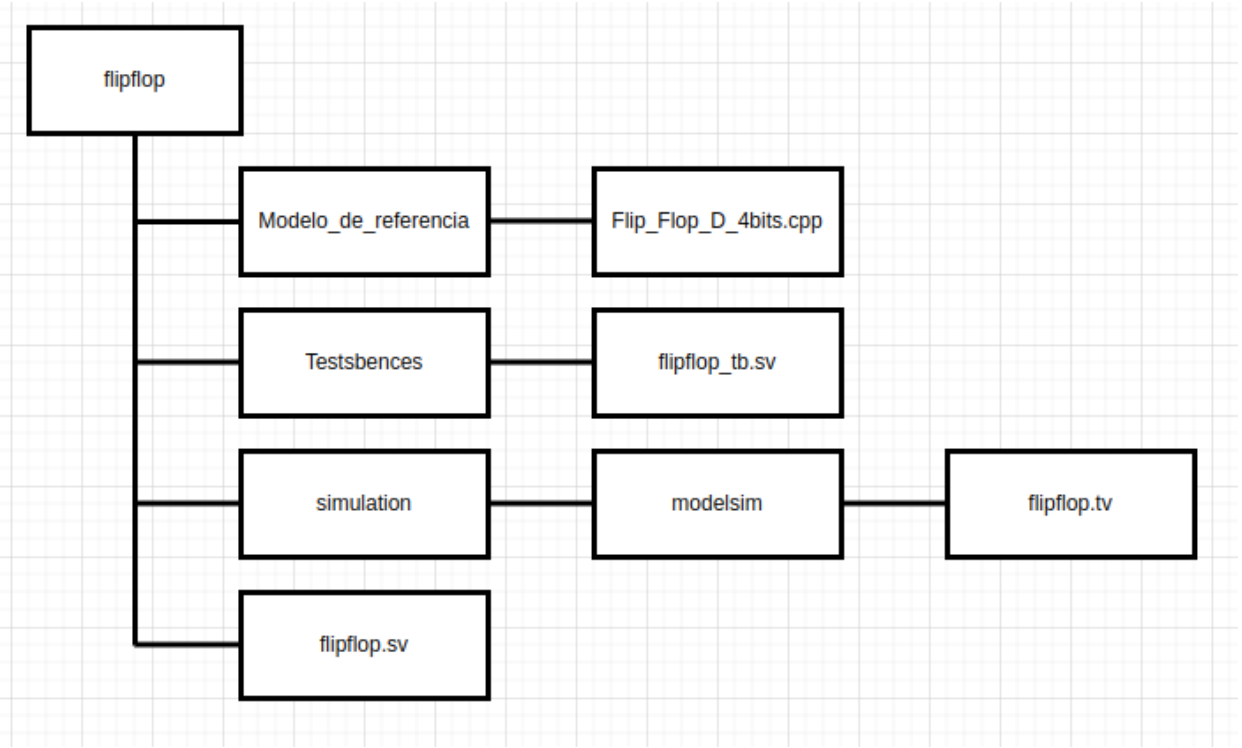
Com a finalização da descrição do hardware e a elaboração do Modelo de Referência, podemos validar a precisão da nossa descrição por meio da criação de um 'testbench'. Este 'testbench' é um programa escrito em SystemVerilog que opera comparando os resultados do nosso módulo com os resultados do Modelo de Referência. Para isso, ele lê o arquivo de teste (.tv), linha por linha, utilizando os bits de entrada como parâmetros de entrada do módulo, e comparando os bits de saída com a saída do módulo.

A seguir, apresentamos o código deste 'testbench'

```
47                counter++;
48              end
49
50
51            else begin
52              $display ("  linha %2d | %b | %b | %b |", counter+1, d, clk, q);
53              if(q !== q_expected) begin
54               for (int i = 0; i < 4; i++)
55                 if(q[i] !== q_expected[i]) begin
56                   $display("Erro: (Q_expected[%0d] = %b)", i, q_expected[i]);
57                   errors++;
58                 end
59          end
60          counter++;
61
62          if (counter === 32) //Quando os vetores de teste acabarem
63           begin
64            $display("Testes Efetuados = %0d", counter);
65            $display("Erros Encontrados = %0d", errors);
66            #10
67            $stop;
68           end
69           end
70
71        end
72
73    endmodule
```

1.4. Hierarquia dos arquivos

Com o Modelo de Referência, a descrição do hardware e o testbench prontos, é crucial armazená-los de maneira específica para que o Quartus II possa acessá-los. Devemos organizar os arquivos conforme ilustrado na imagem a seguir:

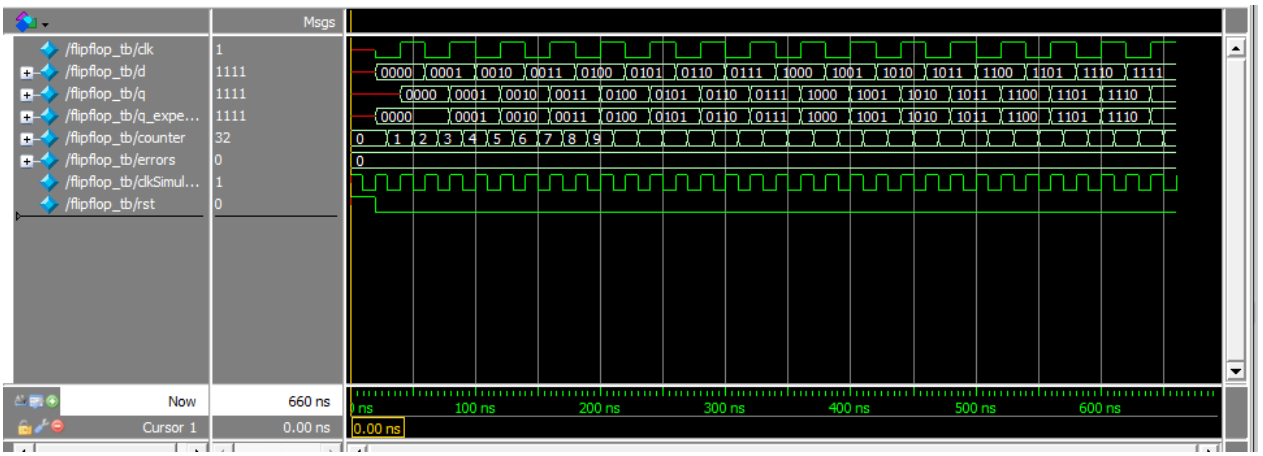


1.5. Simulação RTL LEVEL

Ao realizar a simulação de nível RTL (Register-Transfer Level), estamos concentrando nossa análise exclusivamente na lógica do módulo. Essa simulação pode ser iniciada acessando o menu Tools, e em seguida selecionando Run EDA Simulation Tools e, posteriormente, EDA RTL Simulation.

Ao executar essa simulação, é possível confirmar, conforme demonstrado nas figuras a seguir, que não foram identificados quaisquer erros. Dessa forma, podemos afirmar que a lógica do nosso módulo está correta.

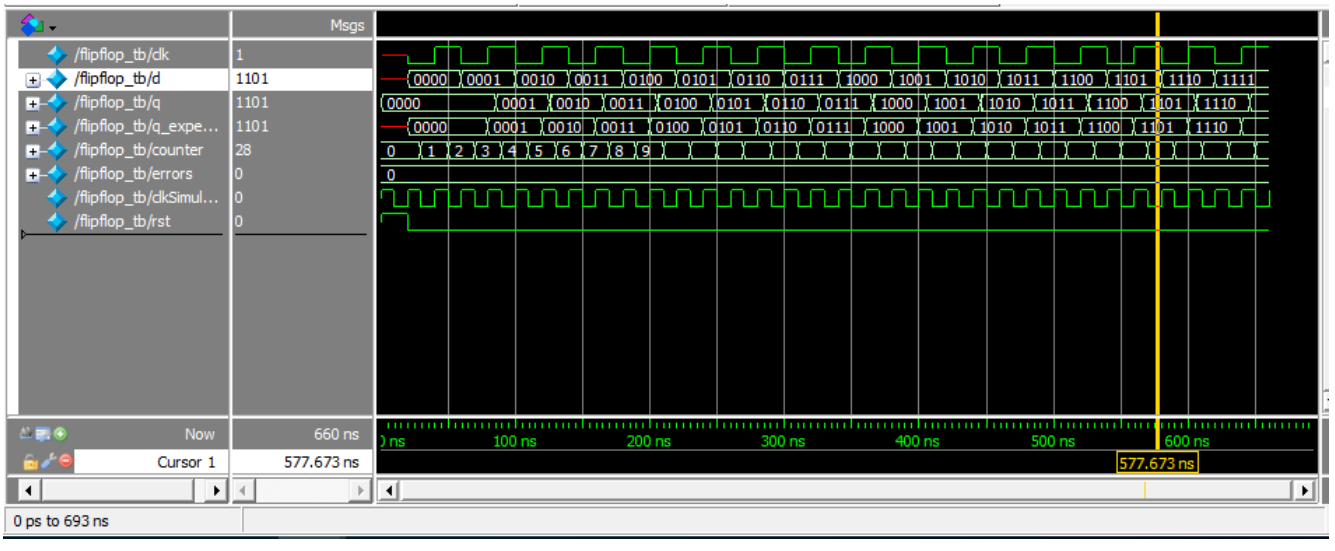
```
#              Iniciando Testbench
#                | D | CLK | Q |
#                ---------------
#    linha  1 | 0000 | 0 | 0000 |
#    linha  2 | 0000 | 1 | 0000 |
#    linha  3 | 0001 | 0 | 0000 |
#    linha  4 | 0001 | 1 | 0001 |
#    linha  5 | 0010 | 0 | 0001 |
#    linha  6 | 0010 | 1 | 0010 |
#    linha  7 | 0011 | 0 | 0010 |
#    linha  8 | 0011 | 1 | 0011 |
#    linha  9 | 0100 | 0 | 0011 |
#    linha 10 | 0100 | 1 | 0100 |
#    linha 11 | 0101 | 0 | 0100 |
#    linha 12 | 0101 | 1 | 0101 |
#    linha 13 | 0110 | 0 | 0101 |
#    linha 14 | 0110 | 1 | 0110 |
#    linha 15 | 0111 | 0 | 0110 |
#    linha 16 | 0111 | 1 | 0111 |
#    linha 17 | 1000 | 0 | 0111 |
#    linha 18 | 1000 | 1 | 1000 |
#    linha 19 | 1001 | 0 | 1000 |
#    linha 20 | 1001 | 1 | 1001 |
#    linha 21 | 1010 | 0 | 1001 |
#    linha 22 | 1010 | 1 | 1010 |
#    linha 23 | 1011 | 0 | 1010 |
#    linha 24 | 1011 | 1 | 1011 |
#    linha 25 | 1100 | 0 | 1011 |
#    linha 26 | 1100 | 1 | 1100 |
#    linha 27 | 1101 | 0 | 1100 |
#    linha 28 | 1101 | 1 | 1101 |
#    linha 29 | 1110 | 0 | 1101 |
#    linha 30 | 1110 | 1 | 1110 |
#    linha 31 | 1111 | 0 | 1110 |
#    linha 32 | 1111 | 1 | 1111 |
# Testes Efetuados = 32
# Erros Encontrados = 0
```



1.6. Simulação Gate Level

Essa simulação considera os tempos de atraso e de propagação de cada porta e sinal individualmente. Ao executar essa simulação através do menu Tools -> Run Simulation Tools -> Gate Level Simulation, é possível observar, conforme mostrado nas imagens a seguir, a identificação de nenhum erro, indicando que o tempo de atraso inicialmente previsto está coerente, não precisando ajustar o período do clock em nível alto no nosso testbench.

```
#            Iniciando Testbench
#              | D | CLK | Q |
#               ---------------
#   linha  1 | 0000 | 0 | 0000 |
#   linha  2 | 0000 | 1 | 0000 |
#   linha  3 | 0001 | 0 | 0000 |
#   linha  4 | 0001 | 1 | 0001 |
#   linha  5 | 0010 | 0 | 0001 |
#   linha  6 | 0010 | 1 | 0010 |
#   linha  7 | 0011 | 0 | 0010 |
#   linha  8 | 0011 | 1 | 0011 |
#   linha  9 | 0100 | 0 | 0011 |
#   linha 10 | 0100 | 1 | 0100 |
#   linha 11 | 0101 | 0 | 0100 |
#   linha 12 | 0101 | 1 | 0101 |
#   linha 13 | 0110 | 0 | 0101 |
#   linha 14 | 0110 | 1 | 0110 |
#   linha 15 | 0111 | 0 | 0110 |
#   linha 16 | 0111 | 1 | 0111 |
#   linha 17 | 1000 | 0 | 0111 |
#   linha 18 | 1000 | 1 | 1000 |
#   linha 19 | 1001 | 0 | 1000 |
#   linha 20 | 1001 | 1 | 1001 |
#   linha 21 | 1010 | 0 | 1001 |
#   linha 22 | 1010 | 1 | 1010 |
#   linha 23 | 1011 | 0 | 1010 |
#   linha 24 | 1011 | 1 | 1011 |
#   linha 25 | 1100 | 0 | 1011 |
#   linha 26 | 1100 | 1 | 1100 |
#   linha 27 | 1101 | 0 | 1100 |
#   linha 28 | 1101 | 1 | 1101 |
#   linha 29 | 1110 | 0 | 1101 |
#   linha 30 | 1110 | 1 | 1110 |
#   linha 31 | 1111 | 0 | 1110 |
#   linha 32 | 1111 | 1 | 1111 |
# Testes Efetuados = 32
# Erros Encontrados = 0
# ** Note: $stop    : Z:/Circuit
```



2. Registrador

Esta seção oferece uma análise abrangente do circuito registrador, uma peça fundamental em sistemas digitais para armazenamento e registro de dados. Projetado para funcionar como um mecanismo de memória capaz de armazenar múltiplos bits de informações, o circuito baseia-se em uma configuração que facilita a retenção e a transmissão precisa de dados entre diferentes partes de um sistema digital. A principal função desse circuito é capturar e manter os valores de entrada, garantindo a integridade dos dados durante operações críticas e sequenciais.

O objetivo central desta seção é enfatizar a impressionante capacidade e eficiência do circuito registrador em armazenar e reter informações de forma estável e confiável. A confiabilidade e a estabilidade operacional são asseguradas em uma ampla gama de configurações, ressaltando a importância e a relevância desse componente em diversos contextos digitais e aplicações práticas.

2.1. Modelo de referência

O modelo de referência é um arquivo .tv que contém vetores de teste a serem utilizados durante o testbench. Esses vetores correspondem, essencialmente, a linhas de uma tabela verdade. Para gerar este arquivo, foi desenvolvido um breve programa em C, demonstrado na imagem a seguir:

```c
#include <stdio.h>

int main() {
    FILE *fp;
    fp = fopen("registrador.tv", "w"); // abre o arquivo para escrita
    if(fp == NULL) {
        printf("Erro ao abrir o arquivo.\n");
        return 1;
    }

    int a, b, c, d, e, f, g, h, clock;
    int anterior_a, anterior_b, anterior_c, anterior_d, anterior_e, anterior_f, anterior_g, anterior_h;

    anterior_a = 0;
    anterior_b = 0;
    anterior_c = 0;
    anterior_d = 0;
    anterior_e = 0;
    anterior_f = 0;
    anterior_g = 0;
    anterior_h = 0;

    for(a=0; a<=1; a++) {
        for(b=0; b<=1; b++) {
            for(c=0; c<=1; c++) {
                for(d=0; d<=1; d++) {
                    for(e=0; e<=1; e++ ){
                        for(f=0; f<=1;f++){
                            for(g=0; g<=1;g++){
                                for(h=0; h<=1; h++){
                                    clock = 0; // gera o Clock para os Flip-Flops (sens$vel @ borda de subida)
                                    fprintf(fp, "%d%d%d%d%d%d%d%d_%d_%d%d%d%d%d%d%d%d\n", a, b, c, d, e, f, g, h, clock, anterior_a, anterior_b, anterior_c, anterior_d, anterior_e, anterior_f, anterior_g, anterior_h);  // escreve a combina@@o pela primeira vez
                                    clock = 1; // gera o clock para os Flip-Flops (sens$vel @ borda de subida)
                                    fprintf(fp, "%d%d%d%d%d%d%d%d_%d_%d%d%d%d%d%d%d%d\n", a, b, c, d, e, f, g, h, clock, a, b, c, d, e, f, g, h); // escreve a combina@@o pela segunda vez
                                    anterior_a = a; // armazena a combina@@o atual
                                    anterior_b = b;
                                    anterior_c = c;
                                    anterior_d = d;
                                    anterior_e = e;
                                    anterior_f = f;
                                    anterior_g = g;
                                    anterior_h = h;
                                }
                            }
                        }
                    }
                }
            }
        }
    }

    fclose(fp); // fecha o arquivo
    printf("Arquivo gerado com sucesso.\n");
    return 0;
}
```

Este código gera um arquivo chamado "registrador.tv" que contém 512 linhas de vetores de teste em binário para o modelo de referência do circuito registrador.

```
1 00000000_0_00000000
2 00000000_1_00000000
3 00000001_0_00000000
4 00000001_1_00000001
5 00000010_0_00000001
6 00000010_1_00000010
7 00000011_0_00000010
8 00000011_1_00000011
9 00000100_0_00000011
```

```
243 01111001_0_01111000
244 01111001_1_01111001
245 01111010_0_01111001
246 01111010_1_01111010
247 01111011_0_01111010
248 01111011_1_01111011
249 01111100_0_01111011
250 01111100_1_01111100
```

```
504 11111011_1_11111011
505 11111100_0_11111011
506 11111100_1_11111100
507 11111101_0_11111100
508 11111101_1_11111101
509 11111110_0_11111101
510 11111110_1_11111110
511 11111111_0_11111110
512 11111111_1_11111111
```

Os primeiros 8 dígitos representam o valor de entrada "d" em binário de 8 bits,o dígito no meio indica o sinal do clock, indicando o momento em que a operação do registrador é acionada, enquanto os últimos 8 dígitos representam o valor esperado de saída "q" em binário de 8 bits.

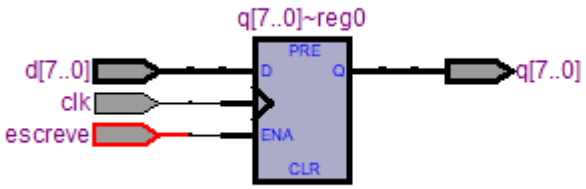
2.2. Descrição do Hardware

A descrição em SystemVerilog (registrador.sv) do módulo "Registrador de 8 bits" desempenha um papel crucial na arquitetura de sistemas digitais. Projetado para armazenar e transmitir dados de 8 bits, o módulo atua como um componente essencial para a manutenção e registro confiáveis de informações vitais. O circuito implementado opera de forma eficiente, garantindo a integridade e a precisão na captura e retenção de dados durante operações críticas.

Detalhes completos sobre as funcionalidades e o comportamento operacional deste módulo podem ser encontrados no diagrama esquemático fornecido abaixo. O registrador de 8 bits foi otimizado para garantir uma resposta consistente e confiável, destacando sua eficácia em uma ampla gama de aplicações que demandam armazenamento e transmissão precisos de dados de 8 bits.

```
1  module registrador
2  (output logic [7:0] q,
3   input logic [7:0] d,
4   input clk, escreve);
5
6   always_ff @(posedge clk) // borda de subida
7   if (escreve ==1)
8   q <= d;
9  endmodule
```

Após a compilação deste código no Quartus II podemos ver a visualização RTL deste módulo:



2.3. Testbench

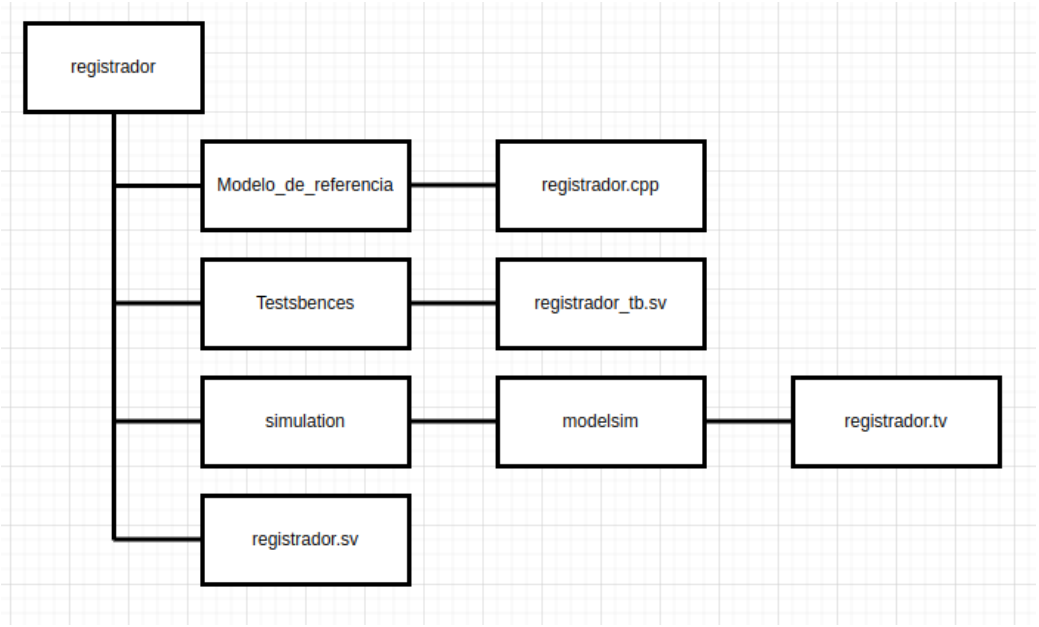
Com a finalização da descrição do hardware e a elaboração do Modelo de Referência, podemos validar a precisão da nossa descrição por meio da criação de um 'testbench'. Este 'testbench' é um programa escrito em SystemVerilog que opera comparando os resultados do nosso módulo com os resultados do Modelo de Referência. Para isso, ele lê o arquivo de teste (.tv), linha por linha, utilizando os bits de entrada como parâmetros de entrada do módulo, e comparando os bits de saída com a saída do módulo.

A seguir, apresentamos o código deste 'testbench'

```
`timescale 1ns / 100ps

module registrador_tb;


  logic clk;
  logic [7:0]d, q, q_expected;
  int counter, errors;
  logic [16:0] vectors [512];
  logic clkSimulation, rst;
  logic  escreve,flag;

  registrador dut(

  .d(d), .clk(clk), .q(q), .escreve(escreve)

  );


  initial // No inicio de toda execu
    begin
      $display ("             Iniciando Testbench");
      $display ("           |   D     | CLK |   Q     |");
      $display ("            -------------------------");
      $readmemb("registrador.tv", vectors); //Carrega os vetores descritos em flop.tv
      counter = 0; errors = 0;  // Inicializa contadores
      rst = 1; #20; rst = 0; // Reset em 1 por 20 ns
    end        // #10


   always // Sempre
    begin
     clkSimulation = 1; #10; // clock em 1 dura 12 ns
     clkSimulation = 0; #10; // clock em 0 dura 7 ns
    end

   always @ (negedge clkSimulation) begin
       if (flag == 0)begin
        escreve = 1;
        flag = 1;
       end
       else begin
        escreve = 0;
        flag = 0;
       end
      end


  always @ (posedge clkSimulation) // Sempre que o clock subir vetores lidos do arquiv
    if(~rst)
     begin
       {d, clk, q_expected} = vectors[counter];
     end

    always @ (negedge clkSimulation) // Sempre que o clock descer
      if(~rst)
       begin
        if (q_expected[0]=== 1'b0) begin
           $display ("  linha %2d | %b | %b | %b |", counter+1, d, clk, q_expected);
           counter++;
          end


        else begin
         $display ("  linha %2d | %b | %b | %b |", counter+1, d, clk, q);
         if(q !== q_expected) begin
          for (int i = 0; i < 8; i++)
            if(q[i] !== q_expected[i]) begin
              $display("Erro: (Q_expected[%0d] = %b)", i, q_expected[i]);
              errors++;
            end
       end
       counter++;

       if (counter === 512) //Quando os vetores de teste acabarem
        begin
         $display("Testes Efetuados = %0d", counter);
         $display("Erros Encontrados = %0d", errors);
          #10
          $stop;
         end



        end

    end

endmodule
```

2.4. Hierarquia dos arquivos

Com o Modelo de Referência, a descrição do hardware e o testbench prontos, é crucial armazená-los de maneira específica para que o Quartus II possa acessá-los. Devemos organizar os arquivos conforme ilustrado na imagem a seguir:

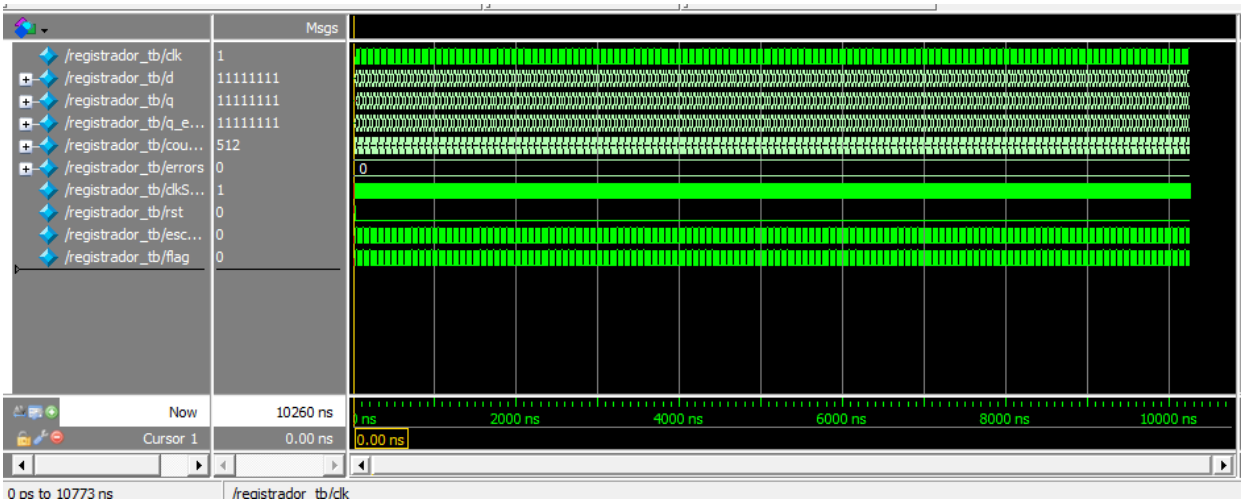


2.5. Simulação RTL LEVEL

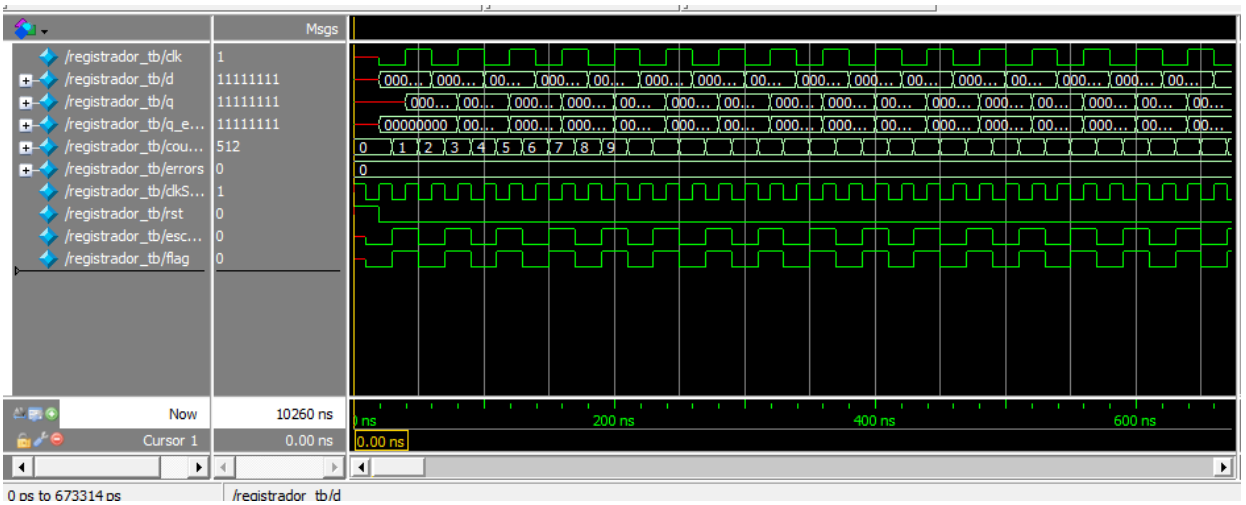
Ao realizar a simulação de nível RTL (Register-Transfer Level), estamos concentrando nossa análise exclusivamente na lógica do módulo. Essa simulação pode ser iniciada acessando o menu Tools, e em seguida selecionando Run EDA Simulation Tools e, posteriormente, EDA RTL Simulation.

Ao executar essa simulação, é possível confirmar, conforme demonstrado nas figuras a seguir, que não foram identificados quaisquer erros. Dessa forma, podemos afirmar que a lógica do nosso módulo está correta.

```
#              Iniciando Testbench
#            |   D      | CLK |   Q      |
#             -------------------------
#   linha  1 | 00000000 | 0 | 00000000 |
#   linha  2 | 00000000 | 1 | 00000000 |
#   linha  3 | 00000001 | 0 | 00000000 |
#   linha  4 | 00000001 | 1 | 00000001 |
#   linha  5 | 00000010 | 0 | 00000001 |
#   linha  6 | 00000010 | 1 | 00000010 |
#   linha  7 | 00000011 | 0 | 00000010 |
#   linha  8 | 00000011 | 1 | 00000011 |
#   linha  9 | 00000100 | 0 | 00000011 |
#   linha 10 | 00000100 | 1 | 00000100 |
#   linha 11 | 00000101 | 0 | 00000100 |
#   linha 12 | 00000101 | 1 | 00000101 |
#   linha 13 | 00000110 | 0 | 00000101 |
#   linha 14 | 00000110 | 1 | 00000110 |
#   linha 15 | 00000111 | 0 | 00000110 |
#   linha 16 | 00000111 | 1 | 00000111 |
#   linha 17 | 00001000 | 0 | 00000111 |
#   linha 495 | 11110111 | 0 | 11110110
#   linha 496 | 11110111 | 1 | 11110111
#   linha 497 | 11111000 | 0 | 11110111
#   linha 498 | 11111000 | 1 | 11111000
#   linha 499 | 11111001 | 0 | 11111000
#   linha 500 | 11111001 | 1 | 11111001
#   linha 501 | 11111010 | 0 | 11111001
#   linha 502 | 11111010 | 1 | 11111010
#   linha 503 | 11111011 | 0 | 11111010
#   linha 504 | 11111011 | 1 | 11111011
#   linha 505 | 11111100 | 0 | 11111011
#   linha 506 | 11111100 | 1 | 11111100
#   linha 507 | 11111101 | 0 | 11111100
#   linha 508 | 11111101 | 1 | 11111101
#   linha 509 | 11111110 | 0 | 11111101
#   linha 510 | 11111110 | 1 | 11111110
#   linha 511 | 11111111 | 0 | 11111110
#   linha 512 | 11111111 | 1 | 11111111
# Testes Efetuados = 512
# Erros Encontrados = 0
```



Devido à ampla gama de vetores presentes em nosso modelo de referência, ao visualizarmos os gráficos de ondas gerados pela simulação em nível RTL, não observamos imediatamente uma correspondência direta com a representação apresentada na figura. No entanto, é importante notar que a visualização pode ser aprimorada consideravelmente ao aplicarmos o zoom no gráfico, permitindo uma análise mais detalhada e precisa.



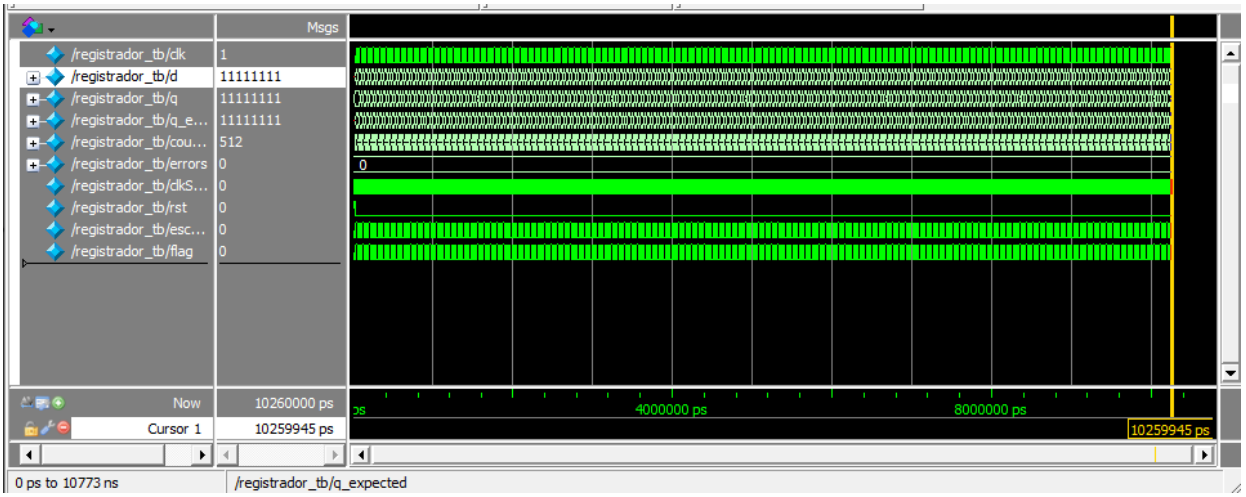
2.6. Simulação Gate Level

Essa simulação considera os tempos de atraso e de propagação de cada porta e sinal individualmente. Ao executar essa simulação através do menu Tools -> Run Simulation Tools -> Gate Level Simulation, é possível observar, conforme mostrado nas imagens a seguir, a identificação de nenhum erro, indicando que o tempo de atraso inicialmente previsto está coerente, não precisando ajustar o período do clock em nível alto no nosso testbench.

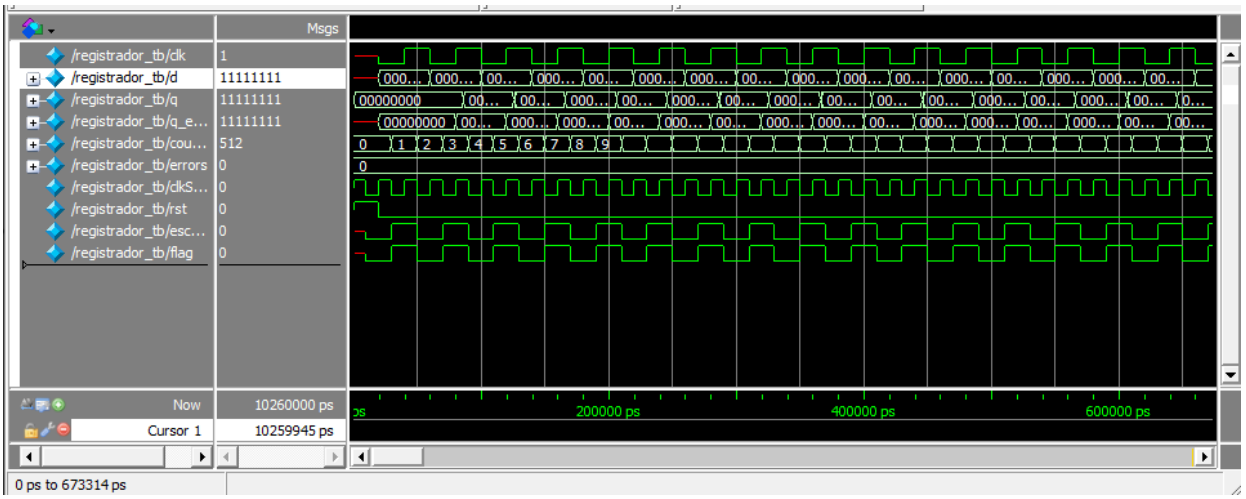
```
#              Iniciando Testbench
#            |   D      | CLK |   Q      |
#             -------------------------
#    linha  1 | 00000000 | 0 | 00000000 |
#    linha  2 | 00000000 | 1 | 00000000 |
#    linha  3 | 00000001 | 0 | 00000000 |
#    linha  4 | 00000001 | 1 | 00000001 |
#    linha  5 | 00000010 | 0 | 00000001 |
#    linha  6 | 00000010 | 1 | 00000010 |
#    linha  7 | 00000011 | 0 | 00000010 |
#    linha  8 | 00000011 | 1 | 00000011 |
#    linha  9 | 00000100 | 0 | 00000011 |
#    linha 10 | 00000100 | 1 | 00000100 |
#    linha 11 | 00000101 | 0 | 00000100 |
#    linha 12 | 00000101 | 1 | 00000101 |
#    linha 13 | 00000110 | 0 | 00000101 |
#    linha 14 | 00000110 | 1 | 00000110 |
#    linha 15 | 00000111 | 0 | 00000110 |
#    linha 16 | 00000111 | 1 | 00000111 |
#    linha 17 | 00001000 | 0 | 00000111 |
#    linha 18 | 00001000 | 1 | 00001000 |
#    linha 19 | 00001001 | 0 | 00001000 |
#    linha 20 | 00001001 | 1 | 00001001 |
#    linha 21 | 00001010 | 0 | 00001001 |
#    linha 22 | 00001010 | 1 | 00001010 |
```

```
#    linha 64 | 00011111 | 1 | 00011111 |
#    linha 65 | 00100000 | 0 | 00011111 |
#    linha 66 | 00100000 | 1 | 00100000 |
#    linha 67 | 00100001 | 0 | 00100000 |
#    linha 68 | 00100001 | 1 | 00100001 |
#    linha 69 | 00100010 | 0 | 00100001 |
#    linha 70 | 00100010 | 1 | 00100010 |
#    linha 71 | 00100011 | 0 | 00100010 |
#    linha 72 | 00100011 | 1 | 00100011 |
#    linha 73 | 00100100 | 0 | 00100011 |
#    linha 74 | 00100100 | 1 | 00100100 |
#    linha 75 | 00100101 | 0 | 00100100 |
#    linha 76 | 00100101 | 1 | 00100101 |
#    linha 77 | 00100110 | 0 | 00100101 |
#    linha 78 | 00100110 | 1 | 00100110 |
#    linha 79 | 00100111 | 0 | 00100110 |
#    linha 80 | 00100111 | 1 | 00100111 |
#    linha 81 | 00101000 | 0 | 00100111 |
#    linha 82 | 00101000 | 1 | 00101000 |
#    linha 83 | 00101001 | 0 | 00101000 |
#    linha 84 | 00101001 | 1 | 00101001 |
#    linha 85 | 00101010 | 0 | 00101001 |
#    linha 86 | 00101010 | 1 | 00101010 |
#    linha 87 | 00101011 | 0 | 00101010 |
#    linha 88 | 00101011 | 1 | 00101011 |
#    linha 89 | 00101100 | 0 | 00101011 |
#    linha 90 | 00101100 | 1 | 00101100 |
#    linha 91 | 00101101 | 0 | 00101100 |
```

```
#    linha 488 | 11110011 | 1 | 11110011 |
#    linha 489 | 11110100 | 0 | 11110011 |
#    linha 490 | 11110100 | 1 | 11110100 |
#    linha 491 | 11110101 | 0 | 11110100 |
#    linha 492 | 11110101 | 1 | 11110101 |
#    linha 493 | 11110110 | 0 | 11110101 |
#    linha 494 | 11110110 | 1 | 11110110 |
#    linha 495 | 11110111 | 0 | 11110110 |
#    linha 496 | 11110111 | 1 | 11110111 |
#    linha 497 | 11111000 | 0 | 11110111 |
#    linha 498 | 11111000 | 1 | 11111000 |
#    linha 499 | 11111001 | 0 | 11111000 |
#    linha 500 | 11111001 | 1 | 11111001 |
#    linha 501 | 11111010 | 0 | 11111001 |
#    linha 502 | 11111010 | 1 | 11111010 |
#    linha 503 | 11111011 | 0 | 11111010 |
#    linha 504 | 11111011 | 1 | 11111011 |
#    linha 505 | 11111100 | 0 | 11111011 |
#    linha 506 | 11111100 | 1 | 11111100 |
#    linha 507 | 11111101 | 0 | 11111100 |
#    linha 508 | 11111101 | 1 | 11111101 |
#    linha 509 | 11111110 | 0 | 11111101 |
#    linha 510 | 11111110 | 1 | 11111110 |
#    linha 511 | 11111111 | 0 | 11111110 |
#    linha 512 | 11111111 | 1 | 11111111 |
# Testes Efetuados = 512
# Erros Encontrados = 0
# ** Note: $stop    : Z:/Circuitoslogicos
```



Devido à ampla gama de vetores presentes em nosso modelo de referência, ao visualizarmos os gráficos de ondas gerados pela simulação em nível GATE, não observamos imediatamente uma correspondência direta com a representação apresentada na figura. No entanto, é importante notar que a visualização pode ser aprimorada consideravelmente ao aplicarmos o zoom no gráfico, permitindo uma análise mais detalhada e precisa.



3. Contador

Esta seção fornece uma visão detalhada do circuito contador, um elemento crucial em sistemas digitais para contagem e rastreamento de eventos. Projetado para operar como um componente fundamental na geração de sequências de contagem, o circuito baseia-se em uma configuração que permite o monitoramento preciso e contínuo de eventos em um circuito digital. Sua principal função é gerar e manter sequências de contagem, possibilitando a contabilização precisa de ocorrências em operações críticas.

O propósito central desta seção é ressaltar a notável capacidade e eficiência do circuito contador em rastrear e registrar eventos de maneira consistente e confiável. A confiabilidade e a estabilidade operacional são mantidas em uma ampla gama de configurações, enfatizando a importância e a versatilidade desse componente em diversos contextos digitais e aplicações práticas.

3.1. Modelo de referência

O modelo de referência é um arquivo .tv que contém vetores de teste a serem utilizados durante o testbench. Esses vetores correspondem, essencialmente, a linhas de uma tabela verdade. Para gerar este arquivo, foi desenvolvido um breve programa em C, demonstrado na imagem a seguir:

```c
#include <stdio.h>

int main() {
    FILE *fp;
    fp = fopen("contador_8bits.tv", "w"); // abre o arquivo para escrita
    if(fp == NULL) {
        printf("Erro ao abrir o arquivo.\n");
        return 1;
    }

    int a, b, c, d, e, f, g, h, clock;

    for(a=0; a<=1; a++) {
        for(b=0; b<=1; b++) {
            for(c=0; c<=1; c++) {
                for(d=0; d<=1; d++) {
                    for(e=0; e<=1; e++ ){
                        for(f=0; f<=1;f++){
                            for(g=0; g<=1;g++){
                                for(h=0; h<=1; h++){
                                    clock = 1; // gera o Clock para os Flip-Flops (sens\0vel \0 borda de subida)
                                    fprintf(fp, "%d_%d%d%d%d%d%d%d%d\n",clock, a, b, c, d, e, f, g, h);  // escreve a combina\0ao pela primeira vez
                                    clock = 0; // gera o clock para os Flip-Flops (sens\0vel \0 borda de subida)
                                    fprintf(fp, "%d_%d%d%d%d%d%d%d%d\n",clock, a, b, c, d, e, f, g, h); // escreve a combina\0\0o pela segunda vez

                                }
                            }
                        }
                    }
                }
            }
        }
    }

    fclose(fp); // fecha o arquivo
    printf("Arquivo gerado com sucesso.\n");
    return 0;
}
```

Este código gera um arquivo chamado "contador_8bits.tv" que contém 512 linhas de vetores de teste em binário para o modelo de referência do circuito contador.

1	1_00000000	263	1_10000011	501	1_11111010
2	0_00000000	264	0_10000011	502	0_11111010
3	1_00000001	265	1_10000100	503	1_11111011
4	0_00000001	266	0_10000100	504	0_11111011
5	1_00000010	267	1_10000101	505	1_11111100
6	0_00000010	268	0_10000101	506	0_11111100
7	1_00000011	269	1_10000110	507	1_11111101
8	0_00000011	270	0_10000110	508	0_11111101
9	1_00000100	271	1_10000111	509	1_11111110
10	0_00000100	272	0_10000111	510	0_11111110
11	1_00000101	273	1_10001000	511	1_11111111
12	0_00000101	274	0_10001000	512	0_11111111

O primeiro dígito indica se o contador está sendo redefinido (0 para não redefinido, 1 para redefinido)e os próximos 8 dígitos, representando o valor do contador em binário de 8 bits, mostram o valor esperado do contador após o ciclo do clock correspondente.

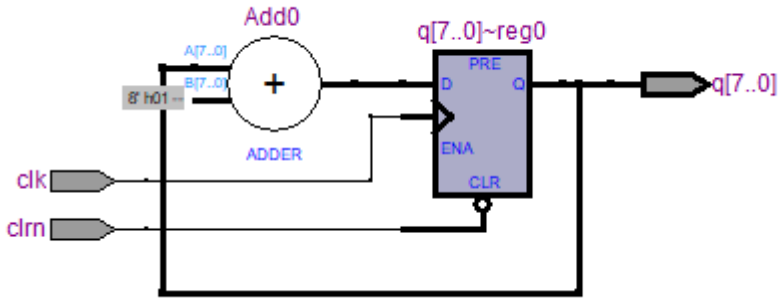
3.2. Descrição do Hardware

A descrição em SystemVerilog (contador_8bits.sv) do módulo "Contador de 8 bits" desempenha um papel crucial na arquitetura de sistemas digitais. Projetado para realizar a contagem precisa de eventos, o módulo é responsável por gerar sequências de contagem de até 8 bits, fornecendo um mecanismo confiável para monitorar e registrar eventos em operações críticas. O circuito implementado opera com eficiência, garantindo a contagem precisa e confiável de eventos em uma variedade de cenários digitais.

Detalhes completos sobre as funcionalidades e o comportamento operacional desse módulo podem ser encontrados no diagrama esquemático fornecido abaixo. O contador de 8 bits foi otimizado para fornecer uma resposta consistente e confiável, demonstrando sua eficácia em uma ampla gama de aplicações que demandam a contagem precisa de eventos de até 8 bits.

```
1   module contador_8bits
2
3    (output logic [7:0] q,
4     input logic clk, clrn);
5
6   always_ff@(posedge clk, negedge clrn)
7     if (~clrn)
8       q <= 0;
9       else q <= q + 1;
10  endmodule
```

Após a compilação deste código no Quartus II podemos ver a visualização RTL deste módulo:



3.3. Testbench

Com a finalização da descrição do hardware e a elaboração do Modelo de Referência, podemos validar a precisão da nossa descrição por meio da criação de um 'testbench'. Este 'testbench' é um programa escrito em SystemVerilog que opera comparando os resultados do nosso módulo com os resultados do Modelo de Referência. Para isso, ele lê o arquivo de teste (.tv), linha por linha, utilizando os bits de entrada como parâmetros de entrada do módulo, e comparando os bits de saída com a saída do módulo.

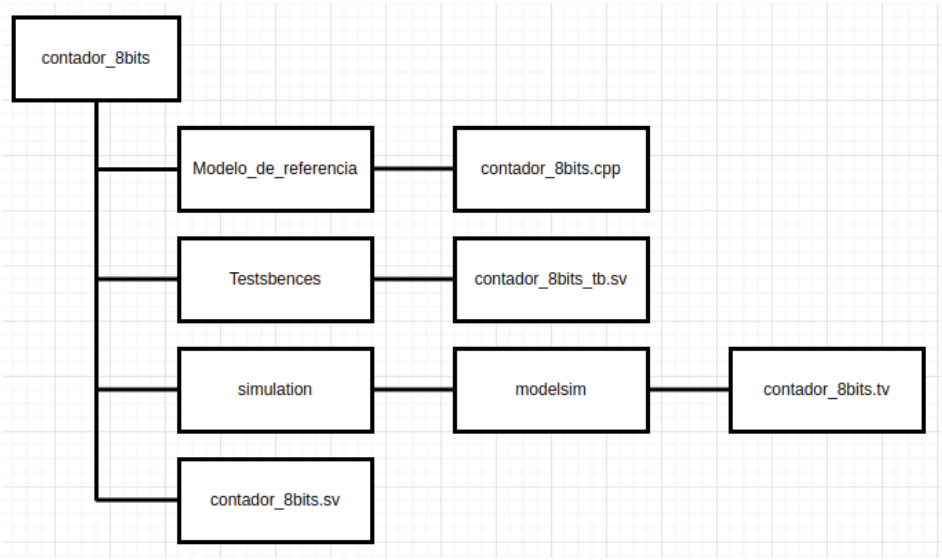
A seguir, apresentamos o código deste 'testbench'

```
1   `timescale 1ns / 100ps
2
3   module contador_8bits_tb;
4
5
6     logic clk;
7     logic [7:0] q, q_expected;
8     int counter, errors;
9     logic [8:0] vectors [512];
10    logic clkSimulation, rst, clrn;
11
12
13    contador_8bits dut(
14
15    .clk(clk), .q(q), .clrn(clrn)
16
17    );
18
19
20    initial // No inicio de toda execu
21       begin
22         $display ("          Iniciando Testbench");
23         $display ("           | CLK |   Q    |");
24         $display ("            --------------");
25         $readmemb("contador_8bits.tv", vectors);
26         counter = 0; errors = 0;  // Inicializa contadores
27         rst = 1; #20; rst = 0; // Reset em 1 por 20 ns
28         clrn = 0; #5; clrn = 1;
29       end        // #10
30
31
32     always // Sempre
33       begin
34        clkSimulation = 1; #10; // clock em 1 dura 12 ns
35        clkSimulation = 0; #10; // clock em 0 dura 7 ns
36       end
37
```

```
    always @ (posedge clkSimulation) // Sempre que o clock subir vetores lidos do arquivo
      if(~rst)
       begin
         {clk, q_expected} = vectors[counter];
       end

      always @ (negedge clkSimulation) // Sempre que o clock descer
        if(~rst)
         begin
          if (q_expected[0]=== 1'b0) begin
             $display ("  linha %2d | %b | %b |", counter+1, clk, q_expected);
             counter++;
            end
          else begin
           $display ("  linha %2d | %b | %b |", counter+1, clk, q);
           if(q !== q_expected) begin
            for (int i = 0; i < 8; i++)
              if(q[i] !== q_expected[i]) begin
                $display("Erro: (Q_expected[%0d] = %b)", i, q_expected[i]);
                errors++;
              end
          end
          counter++;
          if (counter === 512) //Quando os vetores de teste acabarem
           begin
            $display("Testes Efetuados = %0d", counter);
            $display("Erros Encontrados = %0d", errors);
            #10
            $stop;
           end
          end
      end
  endmodule
```

3.4. Hierarquia dos arquivos

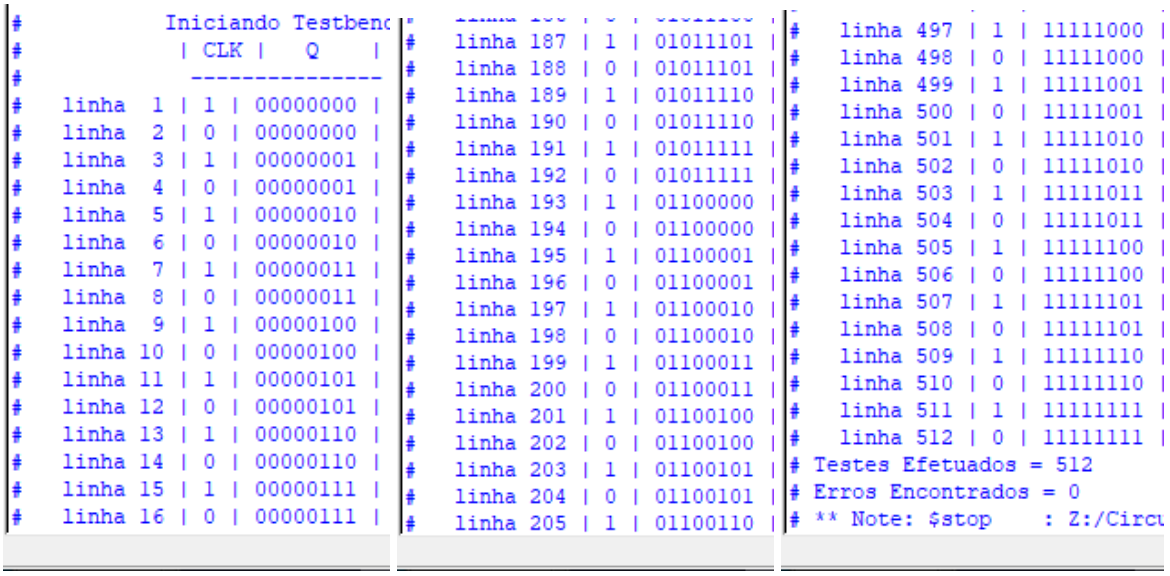
Com o Modelo de Referência, a descrição do hardware e o testbench prontos, é crucial armazená-los de maneira específica para que o Quartus II possa acessá-los. Devemos organizar os arquivos conforme ilustrado na imagem a seguir:



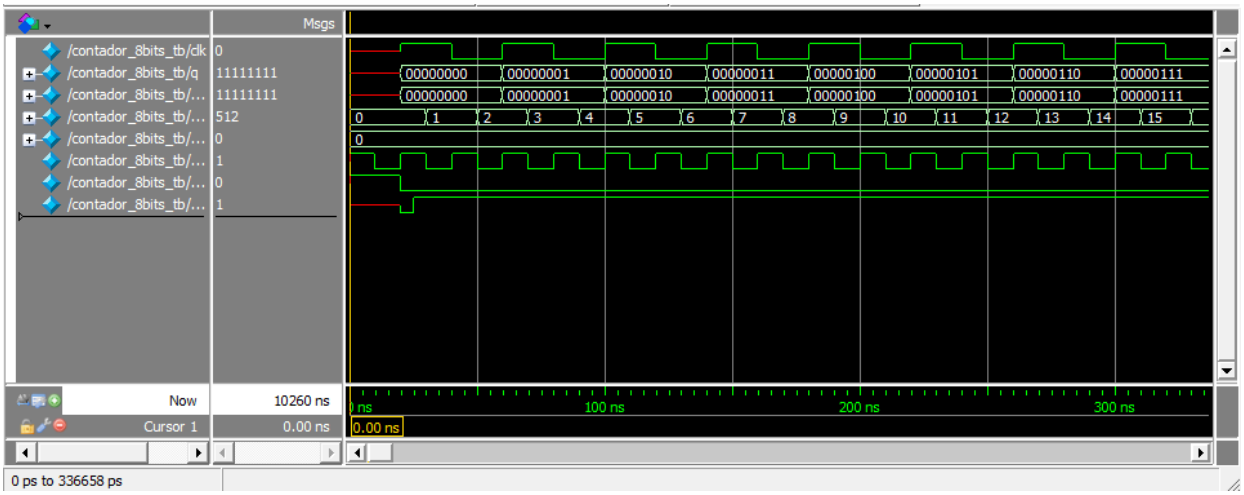
3.5. Simulação RTL LEVEL

Ao realizar a simulação de nível RTL (Register-Transfer Level), estamos concentrando nossa análise exclusivamente na lógica do módulo. Essa simulação pode ser iniciada acessando o menu Tools, e em seguida selecionando Run EDA Simulation Tools e, posteriormente, EDA RTL Simulation.

Ao executar essa simulação, é possível confirmar, conforme demonstrado nas figuras a seguir, que não foram identificados quaisquer erros. Dessa forma, podemos afirmar que a lógica do nosso módulo está correta.



Devido à ampla gama de vetores presentes em nosso modelo de referência, ao visualizarmos os gráficos de ondas gerados pela simulação em nível RTL, não observamos imediatamente uma correspondência direta com a representação apresentada na figura. No entanto, é importante notar que a visualização pode ser aprimorada consideravelmente ao aplicarmos o zoom no gráfico, permitindo uma análise mais detalhada e precisa.

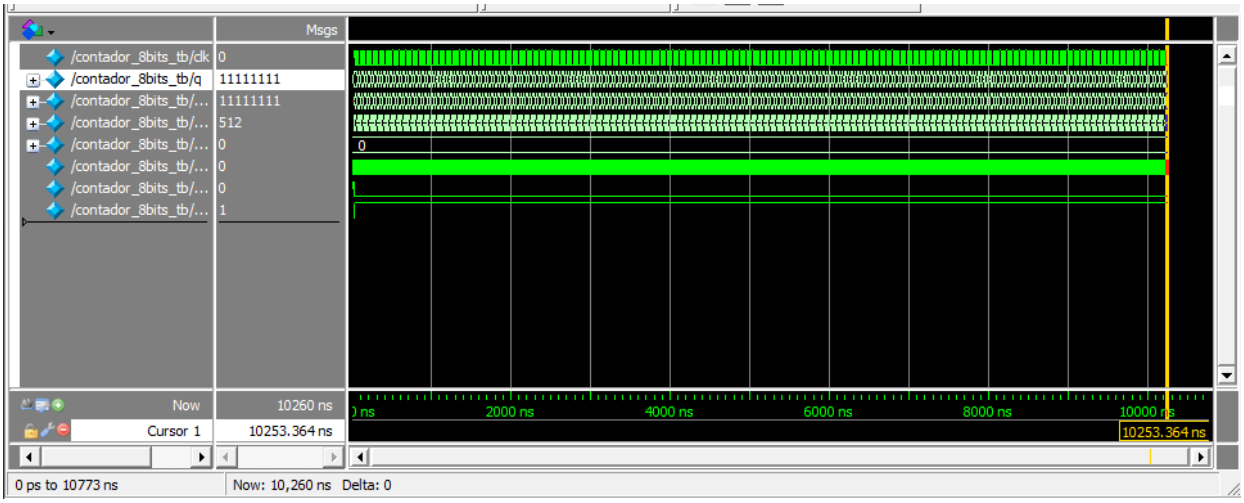


3.6. Simulação Gate Level

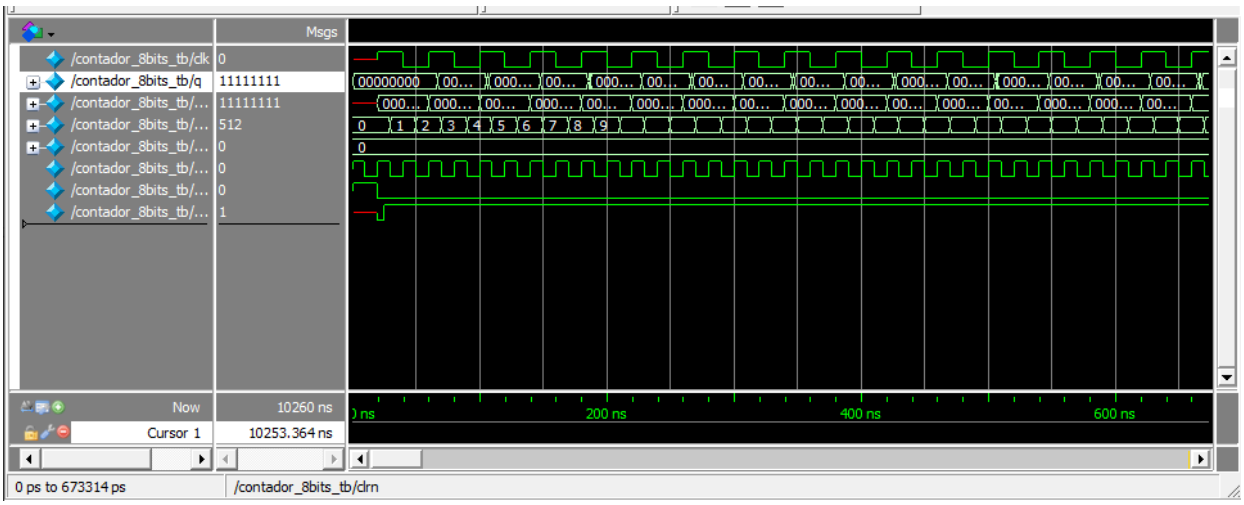
Essa simulação considera os tempos de atraso e de propagação de cada porta e sinal individualmente.

Ao executar essa simulação através do menu Tools -> Run Simulation Tools -> Gate Level Simulation, é possível observar, conforme mostrado nas imagens a seguir, a identificação de nenhum erro, indicando que o tempo de atraso inicialmente previsto está coerente, não precisando ajustar o período do clock em nível alto no nosso testbench.

```
#            Iniciando Testbench
#             | CLK |    Q     |
#              ---------------
#    linha   1 | 1 | 00000000 |
#    linha   2 | 0 | 00000000 |
#    linha   3 | 1 | 00000001 |
#    linha   4 | 0 | 00000001 |
#    linha   5 | 1 | 00000010 |
#    linha   6 | 0 | 00000010 |
#    linha   7 | 1 | 00000011 |
#    linha   8 | 0 | 00000011 |
#    linha   9 | 1 | 00000100 |
#    linha  10 | 0 | 00000100 |
#    linha  11 | 1 | 00000101 |
#    linha  12 | 0 | 00000101 |
#    linha  13 | 1 | 00000110 |
#    linha  14 | 0 | 00000110 |
#    linha  15 | 1 | 00000111 |
#    linha  16 | 0 | 00000111 |
#    linha  17 | 1 | 00001000 |
#    linha  18 | 0 | 00001000 |
#    linha  19 | 1 | 00001001 |
#    linha  20 | 0 | 00001001 |
#    linha  21 | 1 | 00001010 |
#    linha 335 | 1 | 10100111 |
#    linha 336 | 0 | 10100111 |
#    linha 337 | 1 | 10101000 |
#    linha 338 | 0 | 10101000 |
#    linha 339 | 1 | 10101001 |
#    linha 340 | 0 | 10101001 |
#    linha 341 | 1 | 10101010 |
#    linha 342 | 0 | 10101010 |
#    linha 343 | 1 | 10101011 |
#    linha 344 | 0 | 10101011 |
#    linha 345 | 1 | 10101100 |
#    linha 346 | 0 | 10101100 |
#    linha 347 | 1 | 10101101 |
#    linha 348 | 0 | 10101101 |
#    linha 349 | 1 | 10101110 |
#    linha 350 | 0 | 10101110 |
#    linha 351 | 1 | 10101111 |
#    linha 352 | 0 | 10101111 |
#    linha 353 | 1 | 10110000 |
#    linha 354 | 0 | 10110000 |
#    linha 355 | 1 | 10110001 |
#    linha 356 | 0 | 10110001 |
#    linha 357 | 1 | 10110010 |
#    linha 358 | 0 | 10110010 |
#    linha 492 | 0 | 11110101 |
#    linha 493 | 1 | 11110110 |
#    linha 494 | 0 | 11110110 |
#    linha 495 | 1 | 11110111 |
#    linha 496 | 0 | 11110111 |
#    linha 497 | 1 | 11111000 |
#    linha 498 | 0 | 11111000 |
#    linha 499 | 1 | 11111001 |
#    linha 500 | 0 | 11111001 |
#    linha 501 | 1 | 11111010 |
#    linha 502 | 0 | 11111010 |
#    linha 503 | 1 | 11111011 |
#    linha 504 | 0 | 11111011 |
#    linha 505 | 1 | 11111100 |
#    linha 506 | 0 | 11111100 |
#    linha 507 | 1 | 11111101 |
#    linha 508 | 0 | 11111101 |
#    linha 509 | 1 | 11111110 |
#    linha 510 | 0 | 11111110 |
#    linha 511 | 1 | 11111111 |
#    linha 512 | 0 | 11111111 |
# Testes Efetuados = 512
# Erros Encontrados = 0
# ** Note: $stop    : Z:/Circui
```



Devido à ampla gama de vetores presentes em nosso modelo de referência, ao visualizarmos os gráficos de ondas gerados pela simulação em nível GATE, não observamos imediatamente uma correspondência direta com a representação apresentada na figura. No entanto, é importante notar que a visualização pode ser aprimorada consideravelmente ao aplicarmos o zoom no gráfico, permitindo uma análise mais detalhada e precisa.



4. Máquina de estados

Esta seção oferece uma análise abrangente da Máquina de Estados do circuito "Contador Sequencial 0-1-2-3-10-13". Esse componente desempenha um papel essencial em sistemas digitais para controlar e gerenciar a sequência de operações com base em diferentes estados. Projetado para operar como um elemento central na definição de diferentes etapas do processo digital, a Máquina de Estados baseia-se em uma configuração que permite a transição de forma controlada entre estados específicos, proporcionando um controle preciso e sincronizado das operações críticas.

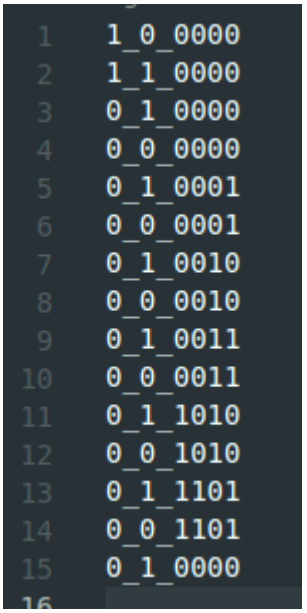
O propósito central desta seção é destacar a notável capacidade e eficiência da Máquina de Estados do circuito "Contador Sequencial 0-1-2-3-10-13" em coordenar e controlar as transições entre estados de maneira consistente e confiável. A confiabilidade e a estabilidade operacional são mantidas em uma ampla gama de configurações, enfatizando a importância e a aplicabilidade desse componente em diversos contextos digitais e cenários práticos.

4.1. Modelo de referência

O modelo de referência é um arquivo .tv que contém vetores de teste a serem utilizados durante o testbench. Esses vetores correspondem, essencialmente, a linhas de uma tabela verdade. Para gerar este arquivo, foi desenvolvido um breve programa em C, demonstrado na imagem a seguir:

```c
#include <stdio.h>

int main() {
    // Nome do arquivo
    char nomeArquivo[] = "contador_sequencia_0_1_2_3_10_13.tv";

    // Abra o arquivo para escrita
    FILE *arquivo = fopen(nomeArquivo, "w");

    // Verifique se o arquivo foi aberto com sucesso
    if (arquivo == NULL) {
        printf("Erro ao abrir o arquivo.\n");
        return 1;
    }

    // Sequência de dados desejada
    char sequencia[][10] = {
        "1_0_0000",             /* Ordem dos dados: RESET_CLK_Yesperado */
        "1_1_0000",
        "0_1_0000",
        "0_0_0000",
        "0_1_0001",
        "0_0_0001",
        "0_1_0010",
        "0_0_0010",
        "0_1_0011",
        "0_0_0011",
        "0_1_1010",
        "0_0_1010",
        "0_1_1101",
        "0_0_1101",
        "0_1_0000"
    };

    // Escreva a sequência de dados no arquivo
    for (int i = 0; i < 15; i++) {
        fprintf(arquivo, "%s\n", sequencia[i]);
    }

    // Feche o arquivo
    fclose(arquivo);

    printf("Sequência de dados escrita com sucesso no arquivo %s.\n", nomeArquivo);

    return 0;
}
```

Este código gera um arquivo chamado "contador_sequencia_0_1_2_3_10_13.tv" que contém 15 linhas de vetores de teste em binário para o modelo de referência do circuito contador.



Cada linha no arquivo .tv representa um vetor de teste para o circuito. O primeiro dígito indica o valor de resete (1 para resete ativo, 0 para resete inativo) , O segundo dígito indica o sinal do clock (1 para clock ativo, 0 para clock inativo), enquanto os quatro dígitos seguintes representam o valor esperado da saída em binário de 4 bits, correspondendo ao estado do circuito "Contador Sequencial 0-1-2-3-10-13" após a operação do clock e considerando o estado do resete.

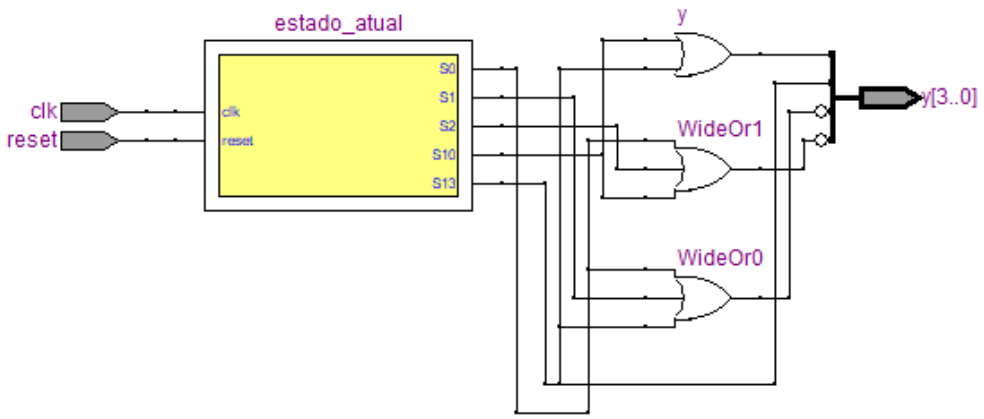
4.2. Descrição do Hardware

A descrição em SystemVerilog (contador_sequencia_0_1_2_3_10_13.sv) do módulo "Contador Sequencial 0-1-2-3-10-13" desempenha um papel crucial na arquitetura de sistemas digitais. Projetado para realizar uma sequência ordenada de contagem de valores, o módulo é responsável por gerar uma série específica de saídas com base no estado atual e no sinal de clock. O circuito implementado opera de maneira precisa e confiável, garantindo uma sequência consistente e ordenada de valores em uma variedade de cenários digitais.

Detalhes completos sobre as funcionalidades e o comportamento operacional desse módulo podem ser encontrados no diagrama esquemático fornecido abaixo. O "Contador Sequencial 0-1-2-3-10-13" foi otimizado para fornecer uma resposta consistente e confiável, demonstrando sua eficácia em uma ampla gama de aplicações que requerem uma sequência específica de contagem e saída de valores.

```
1  //
2  module contador_sequencia_0_1_2_3_10_13
3
4   (output logic [3:0] y,
5    input logic clk, reset);
6
7    logic [2:0] estado_atual;
8
9   parameter S0 = 0, S1 = 1, S2 = 2, S3 = 3, S10 = 4, S13 = 5;
10
11   always_comb  begin // parte combinacional
12    case (estado_atual)
13     S0: y = 0;
14     S1: y = 1;
15     S2: y = 2;
16     S3: y = 3;
17    S10: y = 10;
18    S13: y = 13;
19    endcase
20 end
21
22  always_ff @ (posedge clk, posedge reset) // parte sequencial
23   if (reset)
24    estado_atual <= S0;
25  else
26   case (estado_atual)
27    S0: estado_atual <= S1;
28    S1: estado_atual <= S2;
29    S2: estado_atual <= S3;
30    S3: estado_atual <= S10;
31   S10: estado_atual <= S13;
32   S13: estado_atual <= S0;
33  endcase
34 endmodule
```

Após a compilação deste código no Quartus II podemos ver a visualização RTL deste módulo:



4.3. Testbench

Com a finalização da descrição do hardware e a elaboração do Modelo de Referência, podemos validar a precisão da nossa descrição por meio da criação de um 'testbench'. Este 'testbench' é um programa escrito em SystemVerilog que opera comparando os resultados do nosso módulo com os resultados do Modelo de Referência. Para isso, ele lê o arquivo de teste (.tv), linha por linha, utilizando os bits de entrada como parâmetros de entrada do módulo, e comparando os bits de saída com a saída do módulo.

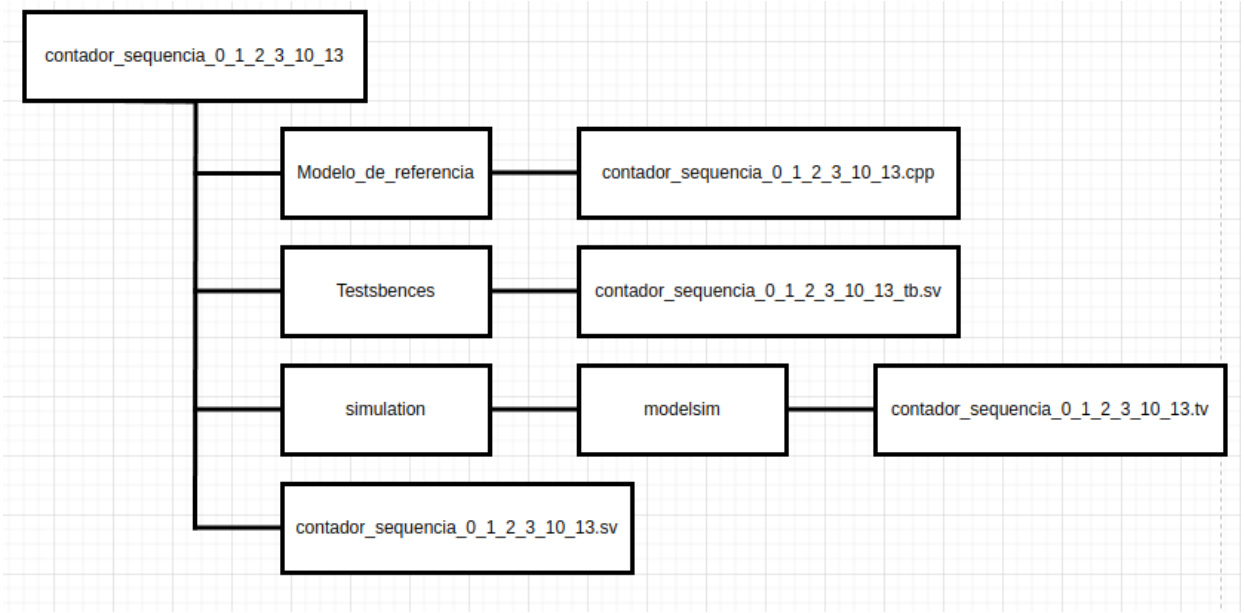
A seguir, apresentamos o código deste 'testbench'

```
module contador_sequencia_0_1_2_3_10_13_tb;


  logic clk;
  logic [3:0] y, y_expected;
  int counter, errors;
  logic [7:0] vectors [15];
  logic clkSimulation, rst, reset;


  contador_sequencia_0_1_2_3_10_13 dut(

  .clk(clk), .y(y), .reset(reset)

  );


  initial // No inicio de toda execu
    begin
      $display (" Iniciando Testbench");
      $display ("          RESET|CLK|  Y  |");
      $display ("      ----------------------");
      $readmemb("contador_sequencia_0_1_2_3_10_13.tv", vectors); //Carrega os vetores
      counter = 0; errors = 0;  // Inicializa contadores
      rst = 1; #20; rst = 0; // Reset em 1 por 20 ns
     end        // #10


    always // Sempre
      begin
       clkSimulation = 1; #10; // clock em 1 dura 12 ns
       clkSimulation = 0; #10; // clock em 0 dura 7 ns
      end
```

```
    always @ (posedge clkSimulation) // Sempre que o clock subir vetores lidos do arquivo
      if(~rst)
       begin
         {reset, clk, y_expected} = vectors[counter];
       end

      always @ (negedge clkSimulation) // Sempre que o clock descer
        if(~rst)
         begin

             $display ("  linha %2d | %b | %b | %b ", counter+1, reset, clk, y_expected);
             if( y!== y_expected)
              begin
                $display("Erro: (y_expected = %b)", y_expected);
                errors++;
            end
        counter++;

        if (counter === 15) //Quando os vetores de teste acabarem
         begin
          $display("Testes Efetuados = %0d", counter);
          $display("Erros Encontrados = %0d", errors);
          #10
          $stop;
         end



       end

    //  end

  endmodule
```

4.4. Hierarquia dos arquivos

Com o Modelo de Referência, a descrição do hardware e o testbench prontos, é crucial armazená-los de maneira específica para que o Quartus II possa acessá-los. Devemos organizar os arquivos conforme ilustrado na imagem a seguir:

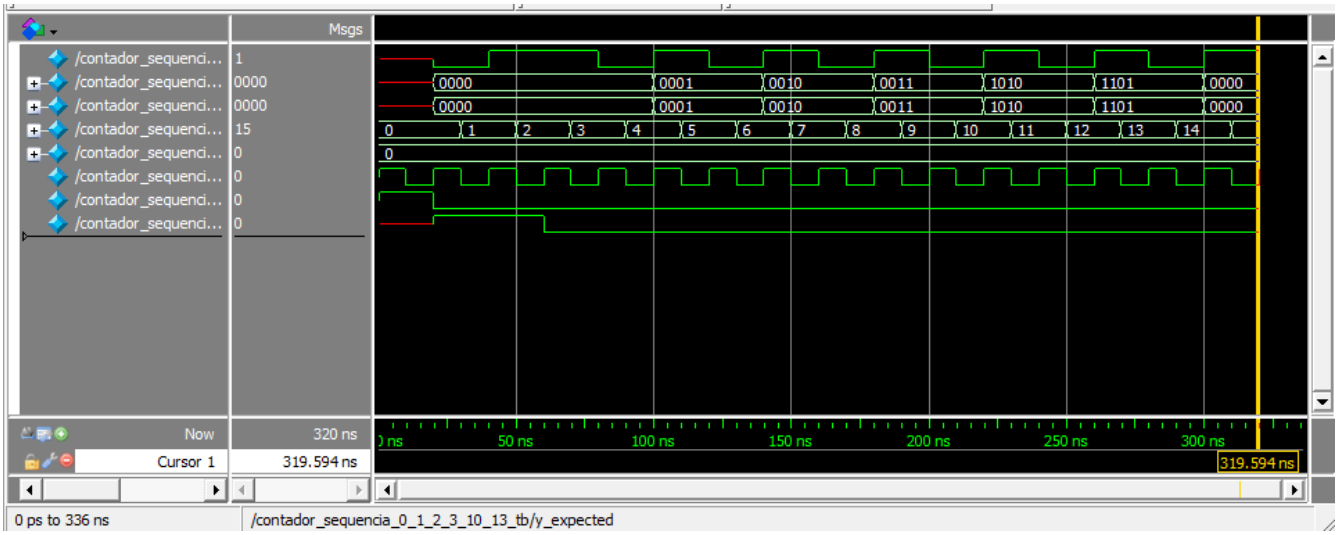


4.5. Simulação RTL LEVEL

Ao realizar a simulação de nível RTL (Register-Transfer Level), estamos concentrando nossa análise exclusivamente na lógica do módulo. Essa simulação pode ser iniciada acessando o menu Tools, e em seguida selecionando Run EDA Simulation Tools e, posteriormente, EDA RTL Simulation.

Ao executar essa simulação, é possível confirmar, conforme demonstrado nas figuras a seguir, que não foram identificados quaisquer erros. Dessa forma, podemos afirmar que a lógica do nosso módulo está correta.

```
#  Iniciando Testbench
#           RESET|CLK|  Y  |
#       --------------------
#    linha  1 | 1 | 0 | 0000
#    linha  2 | 1 | 1 | 0000
#    linha  3 | 0 | 1 | 0000
#    linha  4 | 0 | 0 | 0000
#    linha  5 | 0 | 1 | 0001
#    linha  6 | 0 | 0 | 0001
#    linha  7 | 0 | 1 | 0010
#    linha  8 | 0 | 0 | 0010
#    linha  9 | 0 | 1 | 0011
#    linha 10 | 0 | 0 | 0011
#    linha 11 | 0 | 1 | 1010
#    linha 12 | 0 | 0 | 1010
#    linha 13 | 0 | 1 | 1101
#    linha 14 | 0 | 0 | 1101
#    linha 15 | 0 | 1 | 0000
# Testes Efetuados = 15
# Erros Encontrados = 0
```



4.6. Simulação Gate Level

Essa simulação considera os tempos de atraso e de propagação de cada porta e sinal individualmente.

Ao executar essa simulação através do menu Tools -> Run Simulation Tools -> Gate Level Simulation, é possível observar, conforme mostrado nas imagens a seguir, a identificação de nenhum erro, indicando que o tempo de atraso inicialmente previsto está coerente, não precisando ajustar o período do clock em nível alto no nosso testbench.

```
#  Iniciando Testbench
#           RESET|CLK|  Y   |
#       ----------------------
#    linha  1 | 1 | 0 | 0000
#    linha  2 | 1 | 1 | 0000
#    linha  3 | 0 | 1 | 0000
#    linha  4 | 0 | 0 | 0000
#    linha  5 | 0 | 1 | 0001
#    linha  6 | 0 | 0 | 0001
#    linha  7 | 0 | 1 | 0010
#    linha  8 | 0 | 0 | 0010
#    linha  9 | 0 | 1 | 0011
#    linha 10 | 0 | 0 | 0011
#    linha 11 | 0 | 1 | 1010
#    linha 12 | 0 | 0 | 1010
#    linha 13 | 0 | 1 | 1101
#    linha 14 | 0 | 0 | 1101
#    linha 15 | 0 | 1 | 0000
# Testes Efetuados = 15
# Erros Encontrados = 0
```