UNIVERSIDADE FEDERAL DA PARAÍBA - UFPB
CENTRO DE INFORMÁTICA

Relatório Segunda Avaliação

Disciplina: Circuitos Lógicos II

Professor:

Jose Antonio Gomes de Lima

Equipe :

Pedro Ricardo Cavalcante Silva Filho 20200126968

João Pessoa, PB
2023

SUMÁRIO

1. **Configuração do Projeto……3**
2. **Comparador de números de 4 bits……...6**
3. **Contador de bits 1……12**

1. Configuração do projeto:

Durante este projeto, utilizaremos o software Quartus II, uma ferramenta de design digital da Intel, para escrever a descrição, e o ModelSim, um simulador de hardware da Mentor Graphics, para a simulação. Antes de tudo, é necessário configurar o software para permitir simulações em níveis RTL (Register-Transfer Level) e Gate Level.

As simulações em nível RTL se referem a uma representação de alto nível do circuito digital, onde os registros e transferências de dados são simulados. Essa simulação é essencial para verificar o comportamento do circuito em um nível mais abstrato, antes de passar para a fase de síntese.

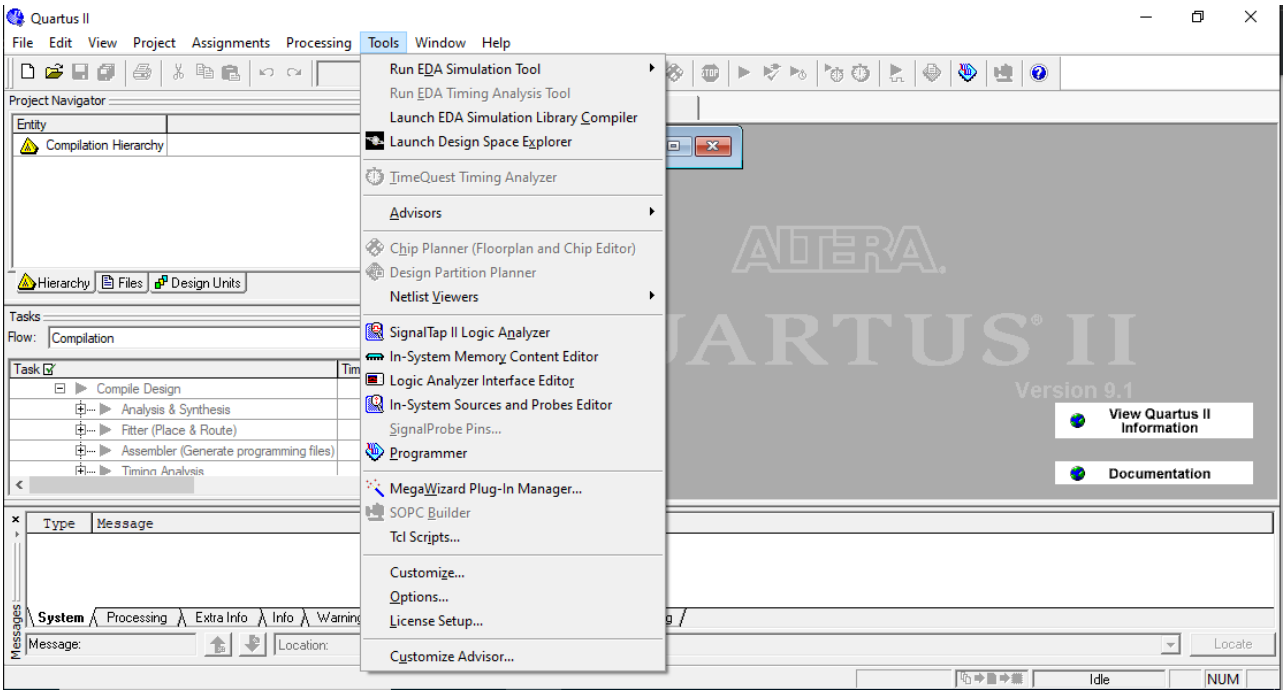
Já as simulações em nível Gate Level são realizadas em um nível mais baixo de abstração, em que os dispositivos lógicos programáveis (PLDs) e a interconexão entre eles são levados em consideração. Esta simulação é crucial para verificar se a implementação física do circuito corresponde à lógica descrita em nível RTL, permitindo uma validação mais próxima do comportamento real do circuito.

Para exemplificar e configurar essas simulações, utilizaremos como estudo de caso os módulos de um comparador de números de 4 bits e um contador de bits 0, ambos representando circuitos combinacionais. É imprescindível ter acesso à descrição em hardware do módulo, ao arquivo testbench e ao Modelo de Referência.

1.1. Passo a passo

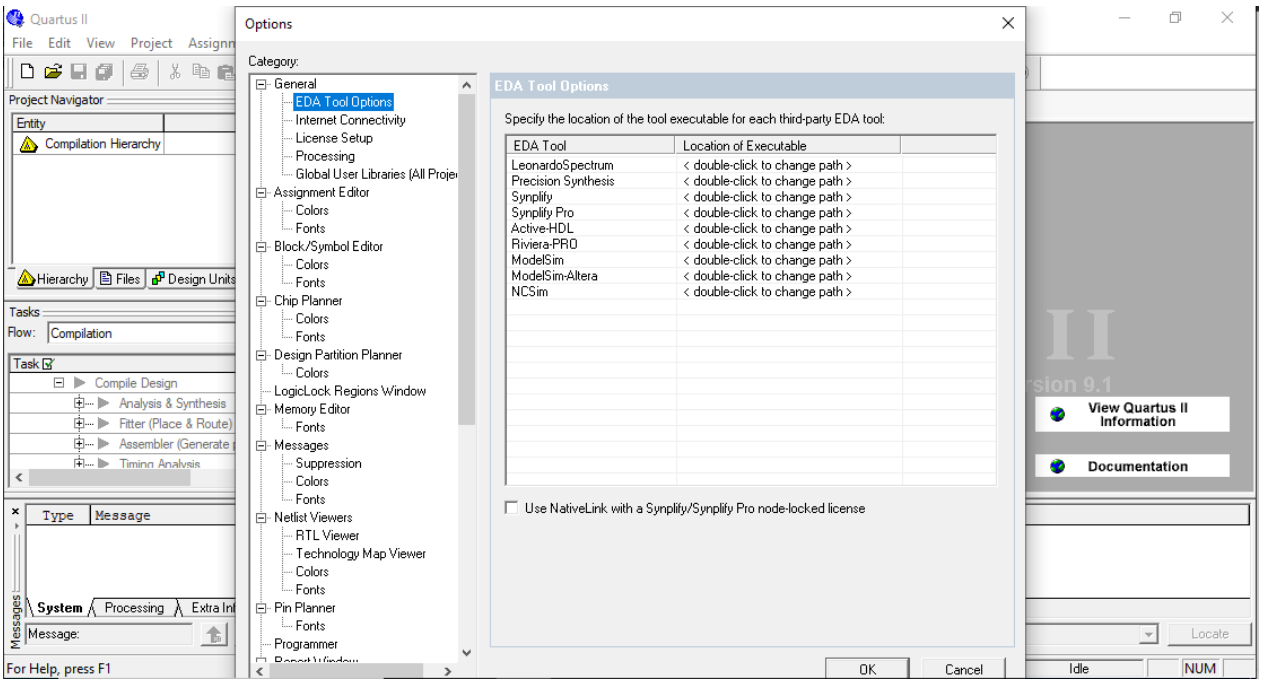
1.1.1. Etapa 1:

Abra o Quartus II e vá até Tools/options:



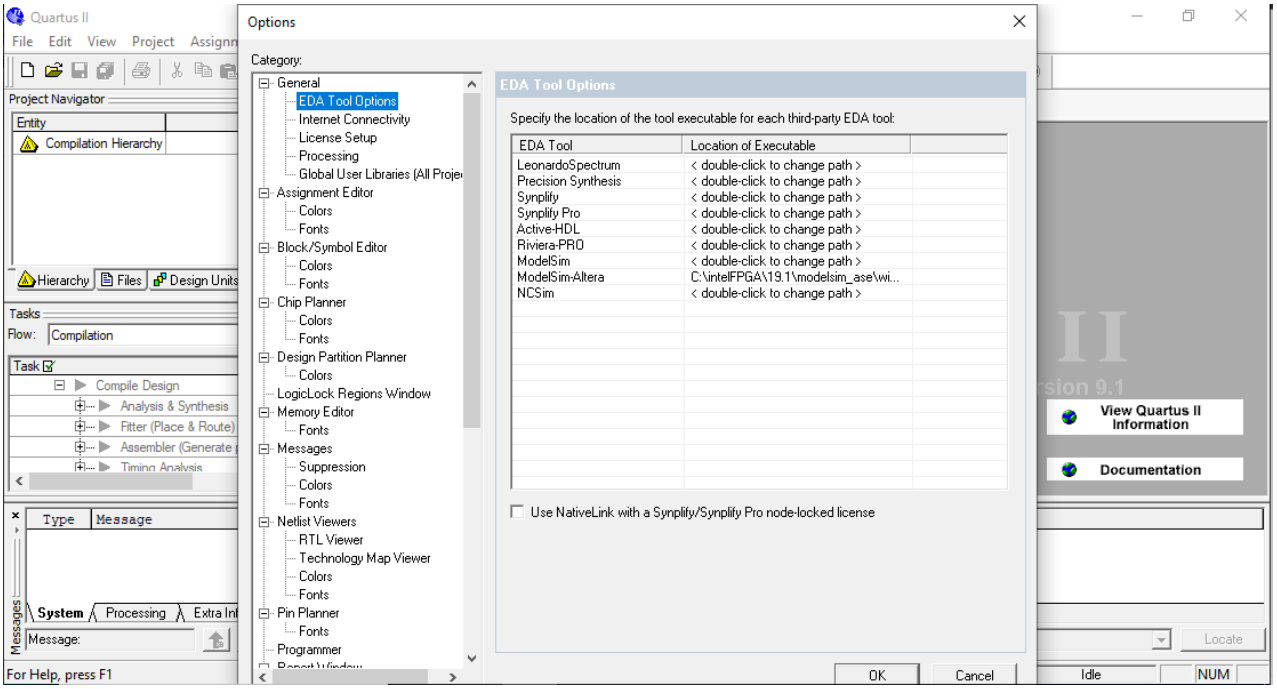
1.1.2. Etapa 2:

Em Options escolha EDA Tool Options



1.1.3. Etapa 3:

Clique na linha ModelSim-Altera e preencha com o caminho do ModelSim, no presente exemplo seria: C:\intelFPGA\19.1\modelsim_ase\win32aloem\

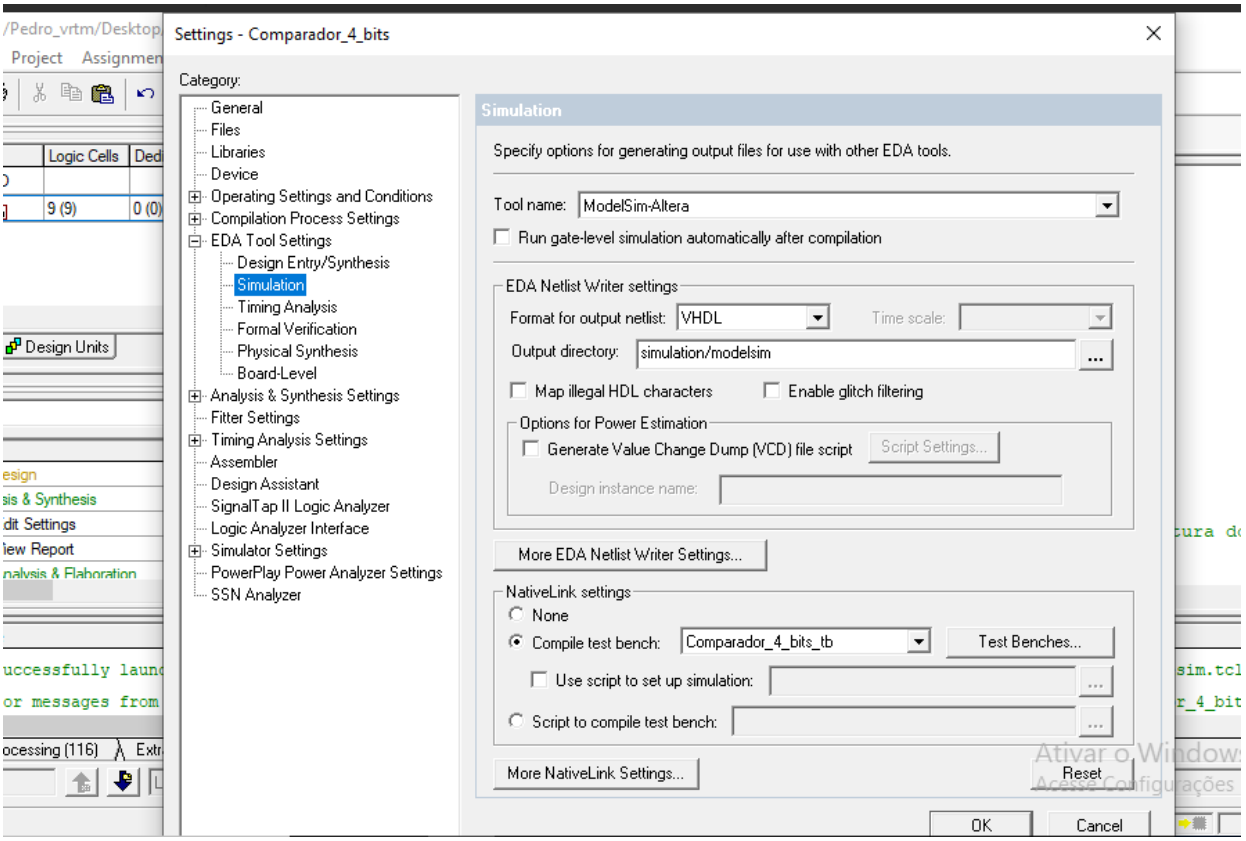


Pronto, os ambientes estão integrados, precisamos agora fazer a descrição do circuito e o testbench para iniciar a simulação.

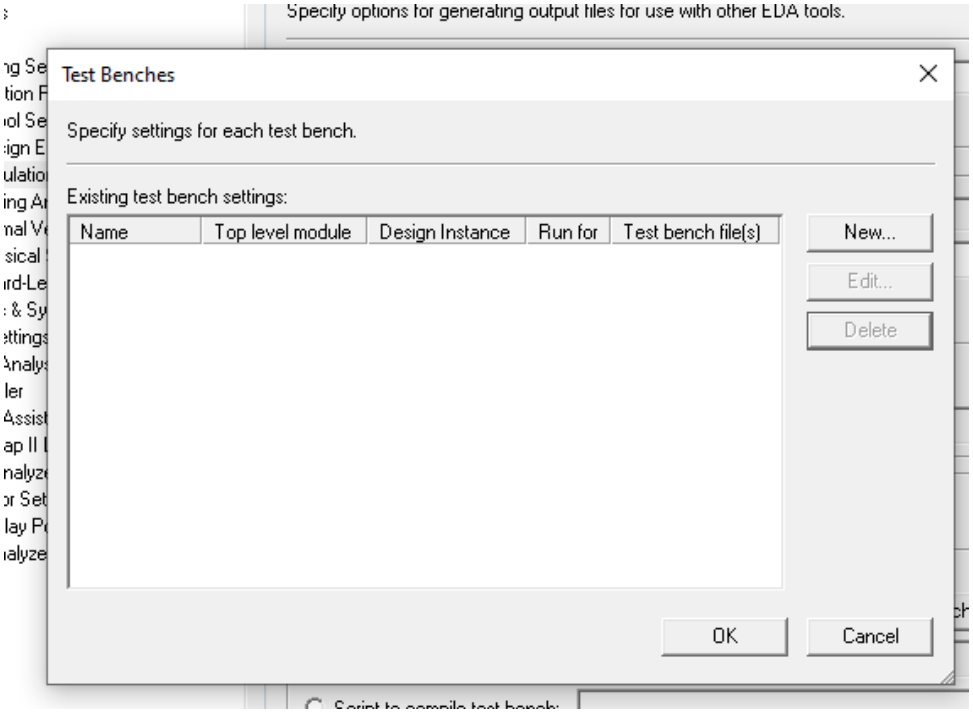
1.1.4. Configurando Testbench:

Para mostrar como se configura o Testbench, vamos usar como exemplo a configuração para o circuito comparador de números de 4 bits.

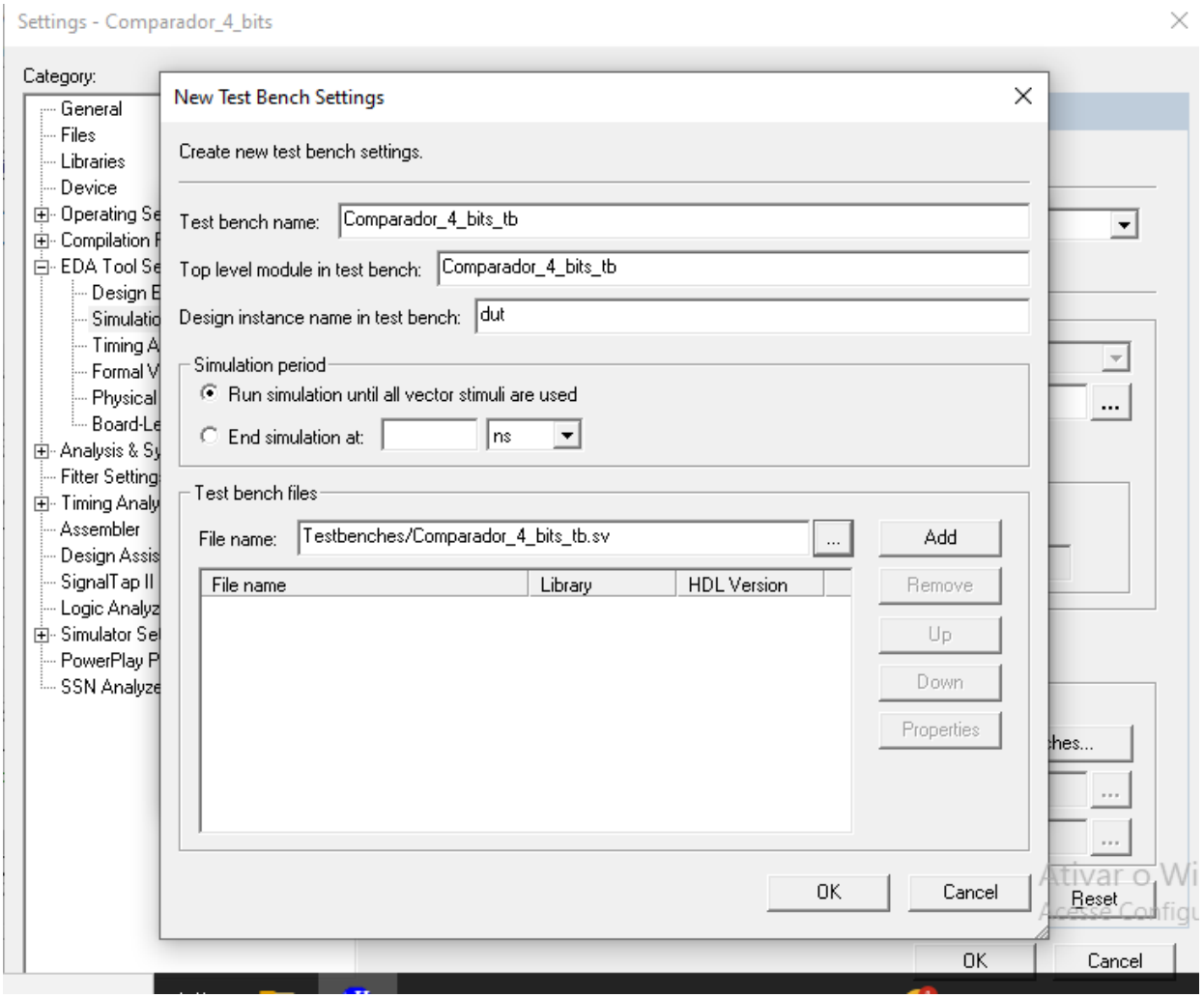
É preciso ir no menu Assignments -> EDA Tool Settings . No campo simulation temos a opção de escolher a ferramenta ModelSim-Altera, como mostrado na figura a seguir:



Na mesma janela do passo anterior devemos marcar a opção compile test bench e selecionar o botão Test Benches. Isso abrirá uma janela.



Nesta janela devemos apertar o botão new que abrirá outra janela, nesta nova janela devemos configurar os campos *Test bench name* e *Top level module in test bench* que devem ser configurados com o nome do arquivo de testbench, o campo *Design instance name in test bench* deve ser configurado com a palavra *dut*. Além disso, devemos configurar o caminho para o arquivo de testbench e apertar o botão *add*.



Após essas configurações todas as ferramentas necessárias durante esse projeto estarão disponíveis.

2. Comparador de números de 4 bits:

Esta seção oferece uma visão detalhada de um circuito combinacional desenvolvido para comparar dois números binários de 4 bits. Projetado para operar sem a necessidade de memória, o circuito baseia-se em uma lógica puramente combinacional. A principal função deste circuito é determinar se os dois números de 4 bits de entrada são iguais, se um é maior que o outro ou se um é menor que o outro.

O propósito central é demonstrar a capacidade precisa e eficiente do circuito comparador de 4 bits em determinar as relações de magnitude entre dois números binários de 4 bits distintos. A confiabilidade e a precisão operacional são garantidas em uma ampla gama de cenários de entrada, o que enfatiza a utilidade e a confiabilidade desse circuito em diversas aplicações.

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>
#include <stdlib.h>
#include <time.h>

char* decimal_to_binary(int n) {
    int c, d, count;
    char *pointer;

    count = 0;
    pointer = (char*)malloc(4+1);

    if (pointer == NULL)
        exit(EXIT_FAILURE);

    for (c = 3; c >= 0; c--) {
        d = n >> c;

        if (d & 1)
            *(pointer+count) = 1 + '0';
        else
            *(pointer+count) = 0 + '0';

        count++;
    }
    *(pointer+count) = '\0';

    return pointer;
}

int main() {
    FILE *file;
    file = fopen("comparador_4bits.tv", "w");

    if (file == NULL) {
        printf("Erro ao criar o arquivo.");
        exit(1);
    }

    srand(time(0));

    for (int i = 0; i < 10; i++) {
        int A = rand() % 16; // Gera um número aleatório entre 0 e 15 (equivalente a 0000 e 1111 em binário de 4 bits)
        int B = rand() % 16;
        int igual = (A == B);
        int maior = (A > B);
        int menor = (A < B);

        char *binary_A = decimal_to_binary(A);
        char *binary_B = decimal_to_binary(B);

        fprintf(file, "%s_%s_%d_%d_%d\n", binary_A, binary_B, igual, maior, menor);

        free(binary_A);
        free(binary_B);
    }

    fclose(file);
    printf("Arquivo .tv gerado com sucesso.");
    return 0;
}
```

Este código gera um arquivo chamado "comparador_4bits.tv" que contém 10 linhas de vetores de teste em binário para o modelo de referência do circuito comparador de 4 bits.

```
1011_1110_0_0_1
1101_0101_0_1_0
0001_0111_0_0_1
1010_1101_0_0_1
1001_0101_0_1_0
0001_1001_0_0_1
0111_0101_0_1_0
1010_1011_0_0_1
1110_0110_0_1_0
0111_1001_0_0_1
```

Cada linha no arquivo .tv representa um vetor de teste para o circuito comparador de 4 bits. Os quatro primeiros dígitos indicam o valor de A em binário de 4 bits, enquanto os quatro dígitos seguintes representam o valor de B em binário de 4 bits.

Os três valores binários seguintes em cada linha indicam se A é igual a B, se A é maior que B e se A é menor que B, respectivamente. Por exemplo, se os valores forem "1_0_0" para um vetor de teste específico, isso indica que A não é igual a B, A é maior que B e A não é menor que B.

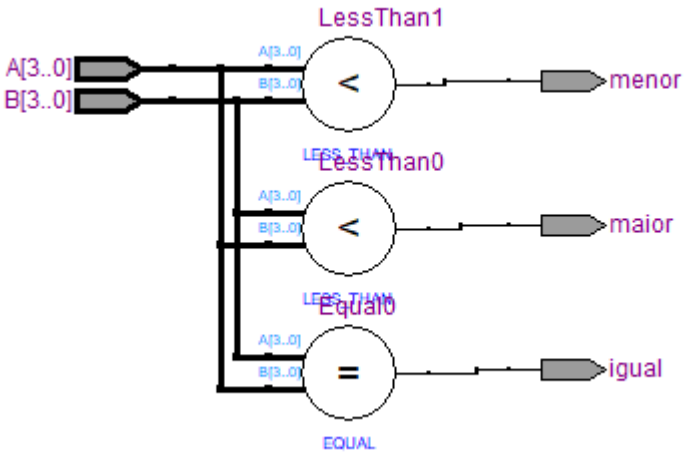
2.2. Descrição do Hardware

A descrição de hardware em SystemVerilog (Comparador_4_bits.sv) do módulo "Comparador de 4 bits" é de importância vital. Projetado para receber duas entradas de 4 bits, o módulo é responsável por determinar se os valores de entrada são iguais, se um é maior que o outro ou se um é menor que o outro. O circuito implementado opera de forma eficiente e confiável, facilitando a comparação precisa de números binários de 4 bits.

A descrição completa das funcionalidades e do comportamento desse módulo pode ser encontrada no diagrama esquemático fornecido abaixo. O módulo foi otimizado para garantir uma resposta precisa e consistente, demonstrando sua eficácia em diversas aplicações que exigem comparação de valores binários de 4 bits.

```
module Comparador_4_bits (
    input logic [3:0] A,
    input logic [3:0] B,
    output logic igual,
    output logic maior,
    output logic menor
);
    assign igual = (A == B);
    assign maior = (A > B);
    assign menor = (A < B);
endmodule
```

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



2.3. Testbench

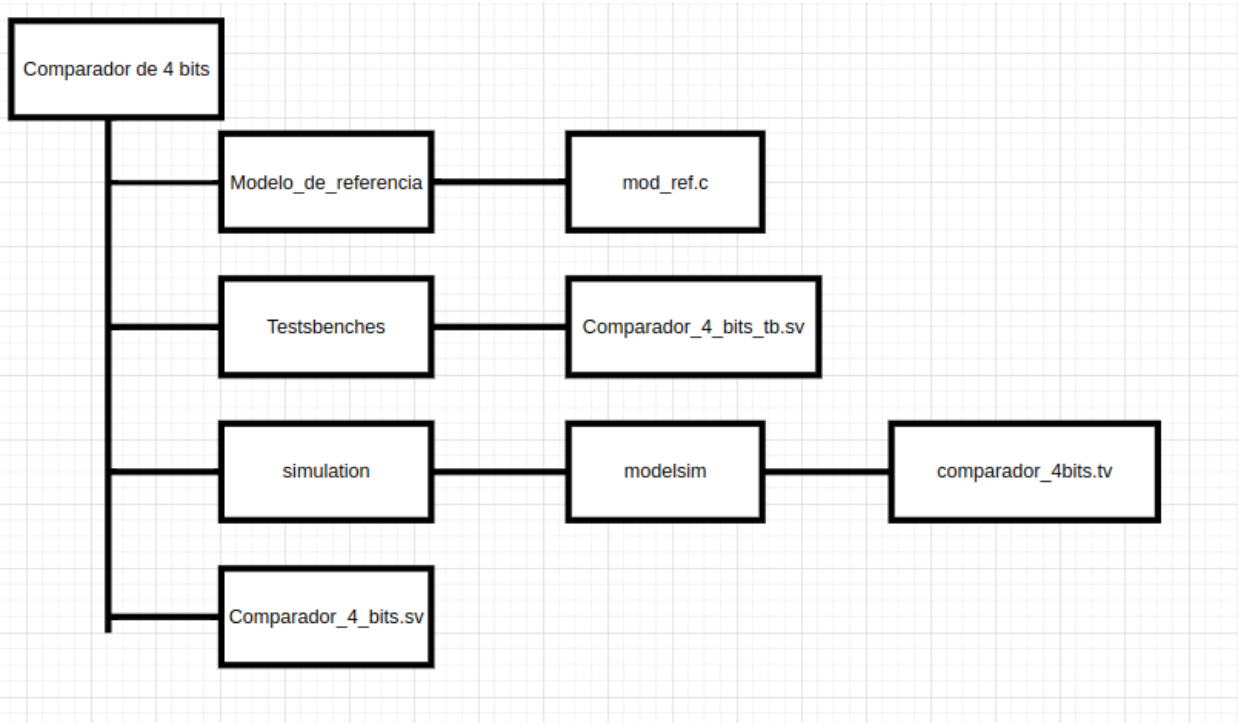
Com a finalização da descrição do hardware e a elaboração do Modelo de Referência, podemos validar a precisão da nossa descrição por meio da criação de um 'testbench'. Este 'testbench' é um programa escrito em SystemVerilog que opera comparando os resultados do nosso módulo com os resultados do Modelo 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 // Ajusta a escala de tempo para ns
module Comparador_4_bits_tb;

    logic [3:0] A;
    logic [3:0] B;
    logic igual;
    logic maior;
    logic menor;
    logic maior_esperado;
    logic menor_esperado;
    logic igual_esperado;
    logic [10:0] vectors [10];
    int testes;
    int erros;
    logic clk, reset;

    Comparador_4_bits dut
    (
        .A(A),
        .B(B),
        .igual(igual),
        .maior(maior),
        .menor(menor)
    );


    initial
        begin
            $display("Inicializando testbench");
            $readmemb("comparador_4bits.tv", vectors); // Leitura do arquivo de teste
            testes = 0; erros = 0;
            reset =1; #20; reset =0; // Reseta o contador
        end

    always
        begin
            clk =1; #10;
            clk =0; #10;
        end

    always @(posedge clk)
        if(~reset)
        begin
            {A, B, igual_esperado, maior_esperado, menor_esperado} = vectors[testes];
        end

    always @(negedge clk)
        if(~reset)
        begin
            assert (menor === menor_esperado && maior === maior_esperado && igual === igual_esperado)
                $display("|%b | %b | = | %b | %b | %b | OK", A, B, igual, maior, menor);
            else
            begin
                $display("|%b | %b | = | %b | %b | %b | ERRO, saida esperada: | %b | %b | %b |", A, B, igual, maior, menor,igual_esperado, maior_esperado, menor_esperado);
                erros = erros + 1;
            end

            testes++;

            if(vectors[testes] === 11'bx)
            begin
                $display("Testes finalizados");
                $display("Total de testes: %0d", testes);
                $display("Total de erros: %0d", erros);
                #10
                $stop;
            end
        end
endmodule
```

2.4. Hierarquia dos arquivos

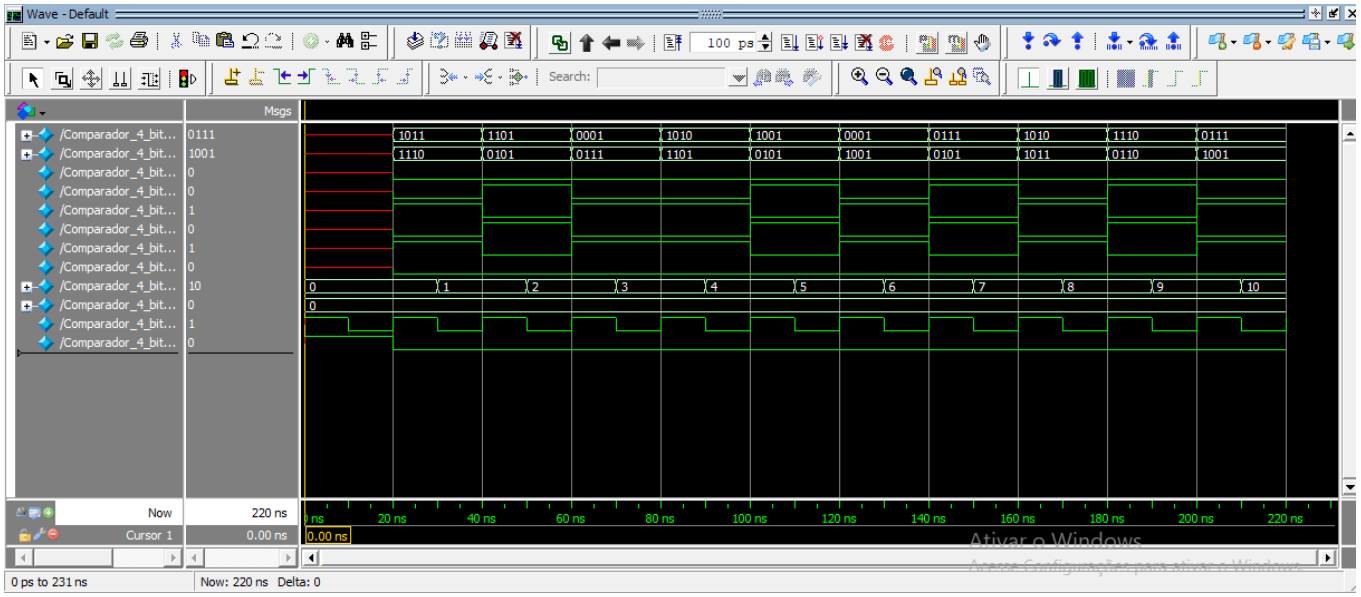
Com o Modelo de Referência, a descrição do hardware e o testbench prontos, é crucial armazená-los de maneira específica para que o Quartus II possa acessá-los. Devemos organizar os arquivos conforme ilustrado na imagem a seguir:



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.

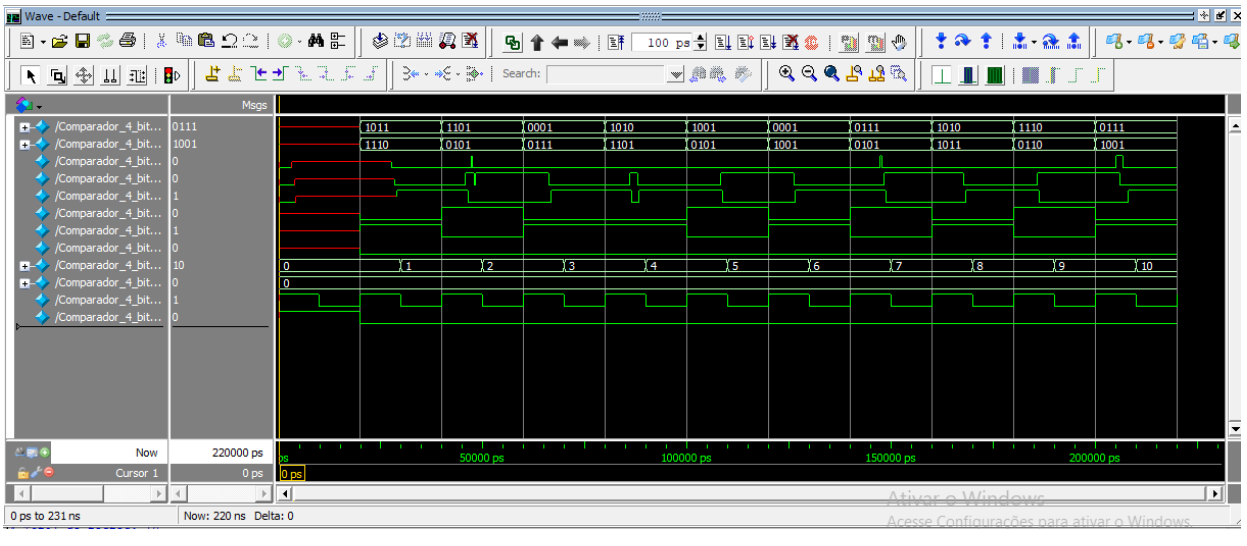
```
# Inicializando testbench
# |1011 | 1110 | = | 0 | 0 | 1 | OK
# |1101 | 0101 | = | 0 | 1 | 0 | OK
# |0001 | 0111 | = | 0 | 0 | 1 | OK
# |1010 | 1101 | = | 0 | 0 | 1 | OK
# |1001 | 0101 | = | 0 | 1 | 0 | OK
# |0001 | 1001 | = | 0 | 0 | 1 | OK
# |0111 | 0101 | = | 0 | 1 | 0 | OK
# |1010 | 1011 | = | 0 | 0 | 1 | OK
# |1110 | 0110 | = | 0 | 1 | 0 | OK
# |0111 | 1001 | = | 0 | 0 | 1 | OK
# Testes finalizados
# Total de testes: 10
# Total de erros: 0
# ** Note: $stop    : C:/Users/Pedro
```



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.

```
# Inicializando testbench
# |1011 | 1110 | = | 0 | 0 | 1 | OK
# |1101 | 0101 | = | 0 | 1 | 0 | OK
# |0001 | 0111 | = | 0 | 0 | 1 | OK
# |1010 | 1101 | = | 0 | 0 | 1 | OK
# |1001 | 0101 | = | 0 | 1 | 0 | OK
# |0001 | 1001 | = | 0 | 0 | 1 | OK
# |0111 | 0101 | = | 0 | 1 | 0 | OK
# |1010 | 1011 | = | 0 | 0 | 1 | OK
# |1110 | 0110 | = | 0 | 1 | 0 | OK
# |0111 | 1001 | = | 0 | 0 | 1 | OK
# Testes finalizados
# Total de testes: 10
# Total de erros: 0
```



3. Contador de bits 1

Esta seção apresenta a descrição de um circuito combinacional projetado para contar o número de 1s em um vetor de 8 bits. O circuito opera com base em uma lógica puramente combinacional, sem a presença de elementos de memória. A funcionalidade principal é determinar a contagem de bits de valor '1' presentes no vetor de entrada de 8 bits.

O objetivo principal é demonstrar a eficácia e precisão do circuito combinacional ao contar o número de 1s em um vetor de 8 bits, garantindo sua confiabilidade e funcionamento preciso em diferentes cenários de entrada.

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>
#include <stdlib.h>

void decToBinary(int n, FILE *fp) {
    // Array para armazenar o binário
    int binaryNum[4] = {0, 0, 0, 0};

    // Contador para o array binário
    int i = 3;
    while (n > 0 && i >= 0) {
        // Armazenando o resto da divisão por 2 no array binário
        binaryNum[i] = n % 2;
        n = n / 2;
        i--;
    }

    // Imprimindo o array binário
    for (int j = 0; j < 4; j++)
        fprintf(fp, "%d", binaryNum[j]);
}

int main() {
    FILE *fp;
    fp = fopen("modelo_de_referencia.tv", "w");
    if (fp == NULL) {
        printf("Erro ao abrir o arquivo.");
        return 1;
    }

    // Criando 8 vetores de entrada aleatórios de 8 bits
    int entradas[8][8];
    for (int i = 0; i < 8; i++) {
        for (int j = 0; j < 8; j++) {
            entradas[i][j] = rand() % 2; // Preenche com 0 ou 1
        }
    }

    // Calculando o número de 1s em cada vetor de entrada e escrevendo no arquivo de saída
    for (int i = 0; i < 8; i++) {
        int count = 0;
        for (int j = 0; j < 8; j++) {
            if (entradas[i][j] == 1) {
                count++;
            }
            fprintf(fp, "%c", entradas[i][j] + '0'); // Escreve cada bit da entrada no arquivo como um caractere '0' ou '1'
        }
        fprintf(fp, "_");
        decToBinary(count, fp); // Escreve a quantidade de 1s em binário com 4 bits no arquivo
        fprintf(fp, "\n"); // Nova linha no arquivo
    }

    fclose(fp);
    return 0;
}
```

Neste arquivo .tv os bits antes do _ servem como as entradas e o após servem como a saída.

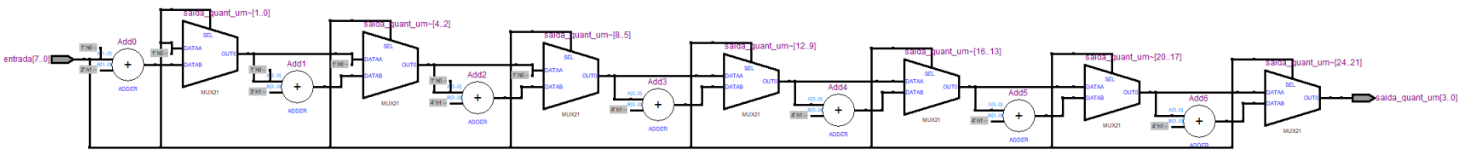
```
10111100_0101
11010110_0101
00001011_0011
00011110_0100
00111010_0100
11110100_0101
10101001_0100
00011101_0100
```

3.2. Descrição do Hardware

Agora, é importante redigir a descrição de hardware (Contador_bits_1.sv) do módulo "Contador de Bits 1" em System Verilog. Este módulo é projetado para receber uma entrada de 8 bits e retornar a quantidade de bits 1 presentes nessa entrada. A descrição detalhada pode ser visualizada na figura abaixo:

```
1  module Contador_bits_1
2  (input logic [7:0] entrada,
3  output logic [3:0] saida_quant_um);
4
5  always @ * begin //O always é usado para definir um bloco de código que será
   executado sempre que houver uma mudança nos sinais de entrada do módulo.
6  saida_quant_um = 0; //Inicializa a saída com 0
7
8  for (int i = 0; i < 8; i++) begin //Loop para percorrer os 8 bits da entrada
9      if (entrada[i] == 1) begin //Se o bit for 1, incrementa a saída
10         saida_quant_um = saida_quant_um + 1; //Incrementa a saída
11     end
12 end
13
14 end
15
16 endmodule
```

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



3.3. Testbenche

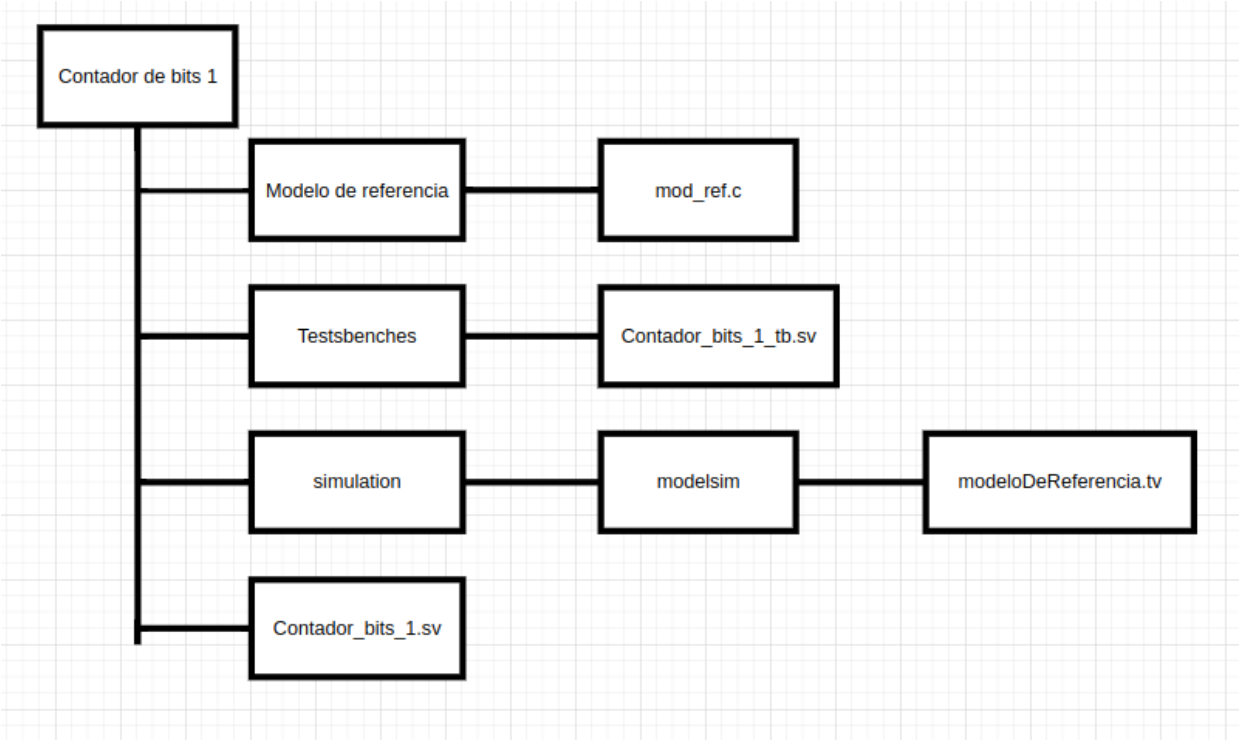
Com a finalização da descrição do hardware e a elaboração do Modelo de Referência, podemos validar a precisão da nossa descrição por meio da criação de um 'testbench'. Este 'testbench' é um programa escrito em SystemVerilog que opera comparando os resultados do nosso módulo com os resultados do Modelo 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 // Ajusta a escala de tempo para ns
module Contador_bits_1_tb;

    logic [7:0] entrada;
    logic [3:0] saida_esperada,saida_quant_um;
    logic [11:0] vectors [8];
    int testes;
    int erros;
    logic clk, reset;

    Contador_bits_1 dut(.entrada(entrada), .saida_quant_um(saida_quant_um)); // Instancia o testbench


    initial
        begin
            $display("Inicializando testbench");
            $readmemb("modeloDeReferencia.tv", vectors); // Leitura do arquivo de teste
            testes = 0; erros = 0;
            reset =1; #20; reset =0; // Reseta o contador
        end

    always
        begin
            clk =1; #10;
            clk =0; #10;
        end

    always @(posedge clk)
        if(~reset)
        begin
            {entrada, saida_esperada} = vectors[testes]; // Leitura dos vetores de teste
        end

    always @(negedge clk)
        if(~reset)
        begin
            assert (saida_quant_um === saida_esperada)
                $display("|%b | %b | OK", entrada, saida_quant_um);
            else
            begin
                $display("|%b | %b | ERRO, saida esperada: %b", entrada, saida_quant_um, saida_esperada);
                erros = erros + 1;
            end

            testes++;

            if(vectors[testes] === 12'bx)
            begin
                $display("Testes finalizados");
                $display("Total de testes: %0d", testes);
                $display("Total de erros: %0d", erros);
                #10
                $stop;
            end
        end
endmodule
```

3.4. Hierarquia dos arquivos

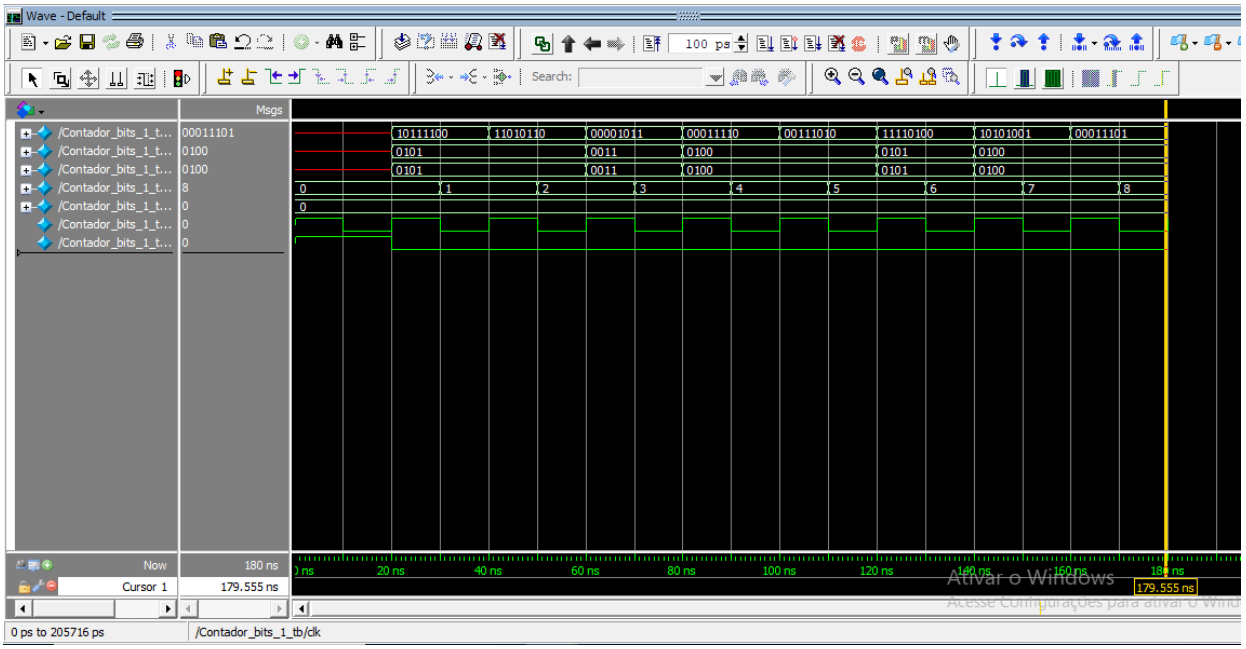
Com o Modelo de Referência, a descrição do hardware e o testbench prontos, é crucial armazená-los de maneira específica para que o Quartus II possa acessá-los. Devemos organizar os arquivos conforme ilustrado na imagem a seguir:



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.

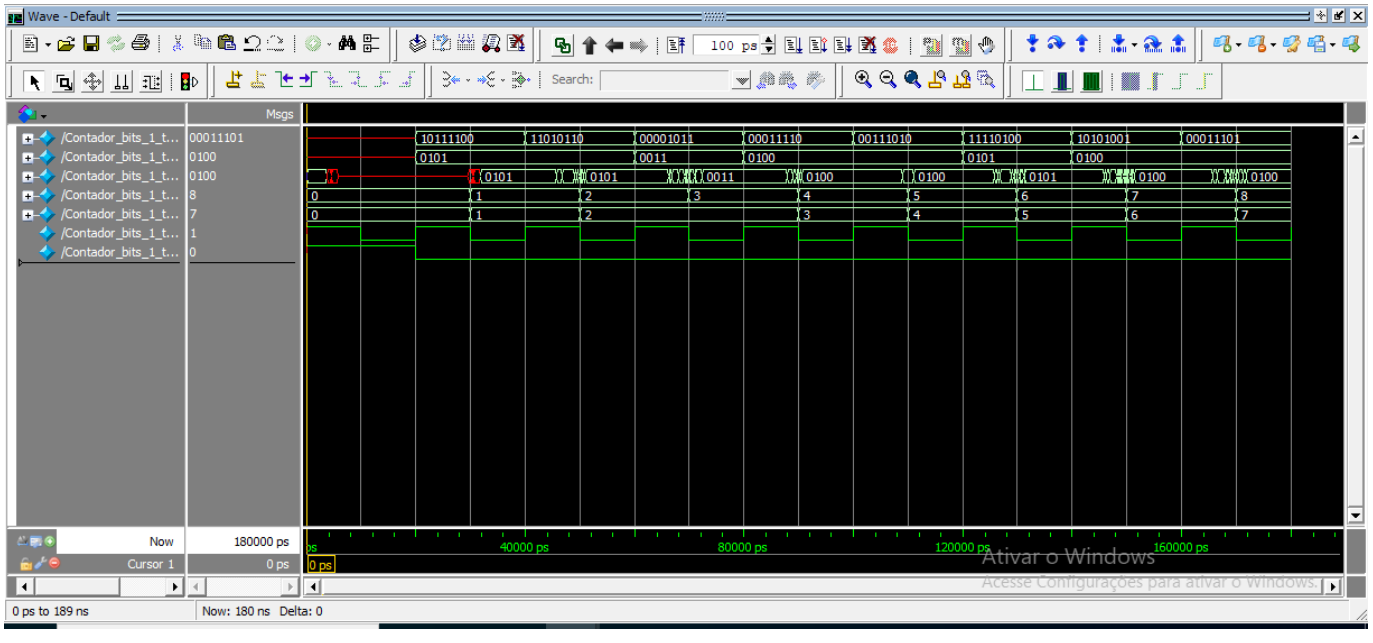
```
# Inicializando testbench
# |10111100 | 0101 | OK
# |11010110 | 0101 | OK
# |00001011 | 0011 | OK
# |00011110 | 0100 | OK
# |00111010 | 0100 | OK
# |11110100 | 0101 | OK
# |10101001 | 0100 | OK
# |00011101 | 0100 | OK
# Testes finalizados
# Total de testes: 8
# Total de erros: 0
```



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 vários erros que não foram detectados durante a simulação RTL.

```
# Inicializando testbench
# |10111100 | xx0x | ERRO, saida esperada: 0101
# |11010110 | 0100 | ERRO, saida esperada: 0101
# |00001011 | 0011 | OK
# |00011110 | 0000 | ERRO, saida esperada: 0100
# |00111010 | 0101 | ERRO, saida esperada: 0100
# |11110100 | 1100 | ERRO, saida esperada: 0101
# |10101001 | 0110 | ERRO, saida esperada: 0100
# |00011101 | 0001 | ERRO, saida esperada: 0100
# Testes finalizados
# Total de testes: 8
# Total de erros: 7
```



Esses erros podem ser atribuídos a um tempo de atraso maior do que o inicialmente previsto. Uma maneira de abordar esse problema é ajustar o período do clock em nível alto no nosso testbench.

```
always
    begin
        clk =1; #13;
        clk =0; #10;
    end
```

Após essas mudanças podemos ver que a simulação termina sem apontar nenhum erro:

```
# Inicializando testbench
# |10111100 | 0101 | OK
# |11010110 | 0101 | OK
# |00001011 | 0011 | OK
# |00011110 | 0100 | OK
# |00111010 | 0100 | OK
# |11110100 | 0101 | OK
# |10101001 | 0100 | OK
# |00011101 | 0100 | OK
# Testes finalizados
# Total de testes: 8
# Total de erros: 0
```