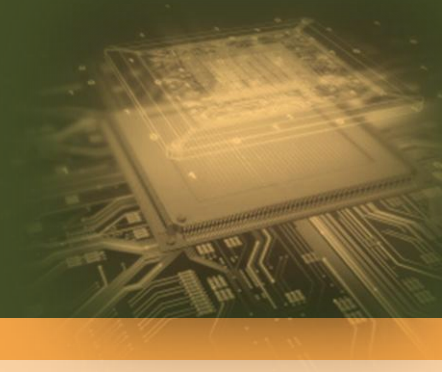
Sistemas Digitais



- Linguagem Verilog

Monitoria SD 2011.2
Daniel Alexandro/Reniê Delgado/Vanessa Ogg

Editado por
(DARA)

Introdução

- **Verilog** é uma linguagem, como VHDL, largamente usada para descrever sistemas digitais e utilizada universalmente;
- **Histórico:** Inicialmente, Verilog era uma linguagem proprietária e desenvolvida pela empresa Gateway. Verilog foi desenvolvida nos anos 1980 e foi inicialmente usada para modelar dispositivos ASIC. Em 1990, Verilog caiu no domínio público e agora está sendo padronizada como IEEE 1364.

Tipos de Variáveis

- Tipos de dados físicos:
    1. Input;
    2. Output;
    3. Output reg;
    4. Inout
    5. Reg; //São um conjunto de FF's que guardam valores
    6. Wire. //São fios que ligam dois pontos do circuito

- Tipos de dados abstratos:
    1. Real;
    2. Integer;
    3. Time.

    OBS.: Variáveis não sintetizadas no Hardware

Exemplos

- input entrada;
- output saída;
- output saida_constante;
- reg [0:7] A, B;

  //[7:0] bits na configuração little endian (menos significativo à direita)

  //[0:7] bits na configuração big endian (mais significativo à direita)
- wire [0:3] Dataout;
- integer count; //Inteiro simples de 32 bits com sinal
- Integer K [1:64]; //Array de 64 inteiros
- time start, stop; //Duas variáveis time de 64 bits

Formato dos Números

- **Formato dos Números:**
  <tamanho><base><número>

Bases: b – Binário
d – Decimal
o – Octal
h – Hexadecimal

Exemplos

- 549 // número decimal
- 12'h 8FF // número hexadecimal
- 9'o765 // número octal
- 4'b11 // número binário de 4-bits: 0011
- 3'b10x // número binário de 3-bits com último bit desconhecido
- 5'd3 // número decimal de 5-bits
- -4'b11 // complementos de dois de 4-bits de 0011 ou 1011

Estrutura de um Código em Verilog

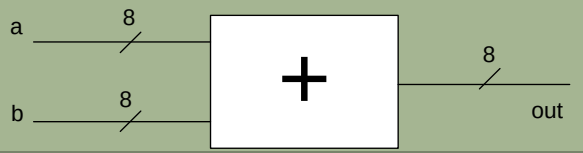
```
module <nome_módulo> (<lista_portas>);

        <declarações>
        <elementos_módulo>

endmodule
```

Exemplo de Programa

```
module add (a, b, out);          // cabeçalho
        input [7:0] a, b;        // entradas
        output [7:0] out;        // saídas
        assign out = a + b;      // operação
endmodule
```



Modelo Comportamental de uma porta NAND

```
module NAND(in1, in2, out);
        input in1, in2;
        output out;
        assign out = ~(in1 & in2);  //atribuição contínua
endmodule
```

Modelo Estrutural de uma porta AND a partir de duas NAND's

```
module AND(in1, in2, out);
        input in1, in2;
        output out;
        wire w1;
        NAND NAND1 (in1, in2, w1);  //Instância de NAND
        NAND NAND2 (w1, w1, out);  //Instância de NAND
endmodule
```

Operadores

- Aritméticos

  Binários:

+	->	Adição
-	->	Subtração
*	->	Multiplicação
/	->	Divisão
%	->	Módulo

  Unário:

-	->	Simétrico (muda o sinal do operador)

  OBS: Divisão por 0 resulta em um valor desconhecido

Operadores

- Relacionais

>	->	Maior
>=	->	Maior ou Igual
<	->	Menor
<=	->	Menor ou Igual
==	->	Igual
!o	->	Diferente

- Lógicos

!	->	Negação Lógica
&&	->	Conjunção Lógica
||	->	Disjunção Lógica
?	->	Condicional

Operadores

- Entre Bits

~	->	Negação
&	->	Conjunção
|	->	Disjunção Inclusiva
^	->	Disjunção Exclusiva
~&	->	NAND
~|	->	NOR
~^ ou ^~	->	Equivalência

- Unário de Redução

&	->	AND
|	->	OR
^	->	XOR
~&	->	NAND
~|	->	NOR
~^	->	XNOR

Operadores

- Outros

{a , b}	->	**Conatenação:** Concatena os bits de 2 ou mais expressões separadas por vírgulas (que neste caso são as variáveis a e b).
<<	->	**Shift à esquerda:** Deslocamento a esquerda de bits (Espaços de bits vazios enchem-se com 0's).
>>	->	**Shift à direita:** Deslocamento a direita de bits (Espaços de bits vazios enchem-se com 0's).

Operadores de Atribuição

- Existem dois tipos de operadores de atribuição. São eles: **“=” e “<=”;**
- O operador de atribuição “=” é similar ao operador em C, onde cada atribuição é executada sequencialmente (a próxima atribuição só acontece quando a anterior é realizada). Este operador é usado para lógica combinacional;
- O operador de atribuição “<=” executa paralelamente (todas as atribuições “<=” são executadas ao mesmo tempo). É usado para flip-flop’s, latches e registradores.

Precedência dos Operadores

Operadores Unários: ! & ~& | ~| ^ ~^ + - (precedência mais alta)

* / %

\+ -

<< >>

< <= > >+

== != === ~==

& ~& ^ ~^

| ~|

&&

||

?:

Parte Lógica

- **Comandos:** assign, assign + function e always;
- Elementos combinatórios podem ser modelados usando comandos assign e always;
- Elementos sequênciais só podem ser modelados com comandos always.

Assign

- **Assign** é usado para modelar somente lógica combinatória;

```
assign add_out = a + b;          // soma
assign or_out = a | b | c;       // or de 3 entradas
```

- **Assign** é executado em paralelo;

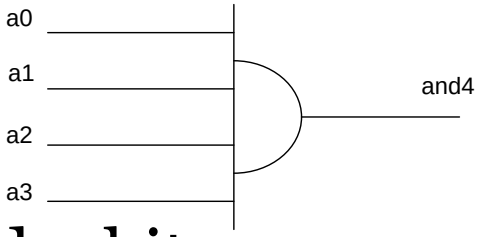
Exemplo

- **Alta Impedância:**

```
assign out = (enable) ? data : 1'bz;
```

  - ✓ Este é um exemplo de um buffer tri-state que utiliza o operador condicional, o valor de alta impedância e o comando assign . Quando "*enable*" é 1, "*out*" recebe o valor de *"data"*. Senão, ocorre alta impedância na saída.

Exemplo



- **Redução:** Resume uma quantidade de bits em uma única operação.

```
assign and4  = &a;
```

Equivale a:

```
assign and4 = a[3] & a[2] & a[1] & a[0];
```

Possível também para: & (AND), | (OR), ^ (XOR)

Function

- Usado quando um mesmo código é repetido várias vezes. Exemplo:

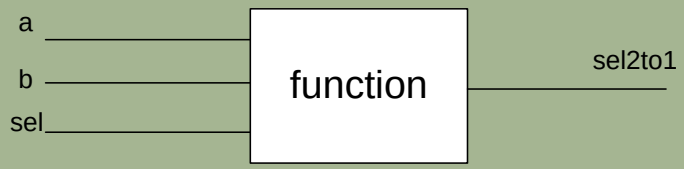
```
function parity;
        input [31:0] data;
        integer i;
                begin
                        parity = 0;
                        for (i = 0; i < 32; i = i + 1)
                        begin
                                parity = parity ^ data[i];
                        end
                end
endfunction
```

Assign + Function

```
assign out = sel2to1(a,b,sel);

function [3:0] sel2to1;
        input [3:0] a,b;
        input sel;

        if (sel)
                sel2to1 = a;
        else
                sel2to1 = b;
endfunction
```




```
case (sel)
        1: sel2to1 = a;
        0: sel2to1 = b;
endcase
```

Blocos

- If – else;
- Case;
- While;
- For;
- Repeat;
- Initial;
- Always.

If - else

```
if (A == 4)
        begin
                B = 2;
        end
else
        begin
                B = 4;
        end
```

Case

```
case (<expressão>)

        <value1>: <instruções>
        <value2>: <instruções>
        default: <instruções>

endcase
```

While

```
while(i < 10)
        begin
                count = count + 1;
                i = i + 1;
        end
```

For

```
for(i = 0; i < 10; i = i + 1)
        begin
                count = count + 1;
        end
```

OBS.: Em Verilog, os operadores ++ e -- não existem, devendo por isso utililizar [i = i + (ou -) 1].

Repeat

- O código é executado a quantidade de vezes que está dentro do parênteses que fica ao lado do **repeat**.

```
repeat (5)
        begin
                count = count + i;
                i = i + 1;
        end
```

Initial

- Inicializa as variáveis.

```
initial
        begin           //Valores de teste
                a = 0;
                b = 0;
        end
```

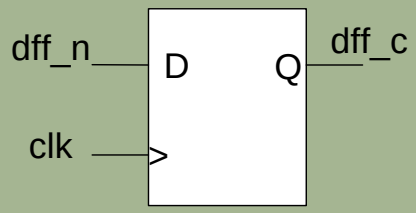
Always

- O comando **Always** do Verilog é equivalente ao **Process** do VHDL;
- Todos os comandos **Always** são executados em paralelo, enquanto que, internamente a um comando **Always**, os comandos são executados em sequência;
- **Always** recebe como parâmetros uma lista de sensibilidade, ou seja, uma lista que diz quando o bloco de código é executado;
- Quando um dos parâmetros é antecipado por **posedge**, isso significa que o **Always** vai ser executado quando aquela determinada variável (parâmetro) realizar uma transição de subida (0 -> 1);
- Quando um dos parâmetros é antecipado por **negedge**, isso significa que o **Always** vai ser executado quando aquela determinada variável (parâmetro) realizar uma transição de descida (1 -> 0);
- Caso o parâmetro seja **(*)**, isso significa que o **Always** vai ser executado quando qualquer variável (porta) de entrada realizar qualquer transição.

Exemplos

```
always @ (a or b or sel)
begin
        y = 0;
        if (sel ==0)
        begin
                y = a;
        end
        else
        begin
                y = b;
        end
end
```

lista de sensibilidade – diz quando o bloco de código é executado.




```
always @ (posedge clk)
begin
        dff_c <= dff_n;
end
```

Exemplos

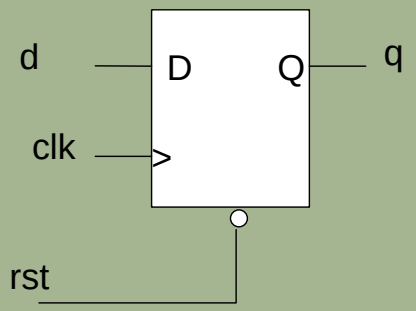
```
always @ (posedge clk or posedge rst)
begin
        if (rst)
        begin
                count <= 0;
        end
        else
        begin
                while (enable)
                begin
                        count <= count + 1;
                end
        end
end
```

Exemplos

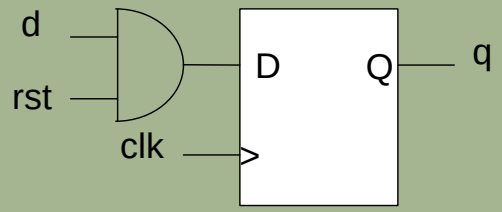
```
module toplevel (input clock , input reset );
        reg flop1 ;
        reg flop2 ;
        always @ ( posedge reset or posedge clock )
        begin
                if ( reset )
                begin
                        flop1 <= 0 ;
                        flop2 <= 1 ;
                end
                else
                begin
                        flop1 <= flop2 ;
                        flop2 <= flop1 ;
                end
        end
endmodule
```

Exemplos

```
always @ (posedge clk or negedge rst)
begin
        if (rst ==1'b0)
                q <= 1'b0;
        else
                q <= d;
end
```



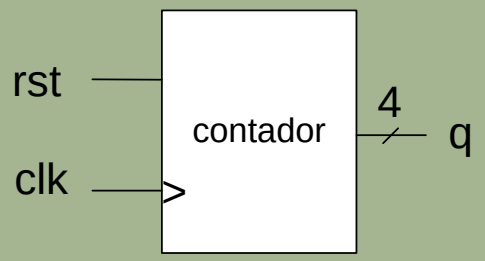

```
always @(posedge clk)
begin
        if (rst ==1'b0)
                q <= 1'b0;
        else
                q <= d;
end
```



Exemplo de um contador

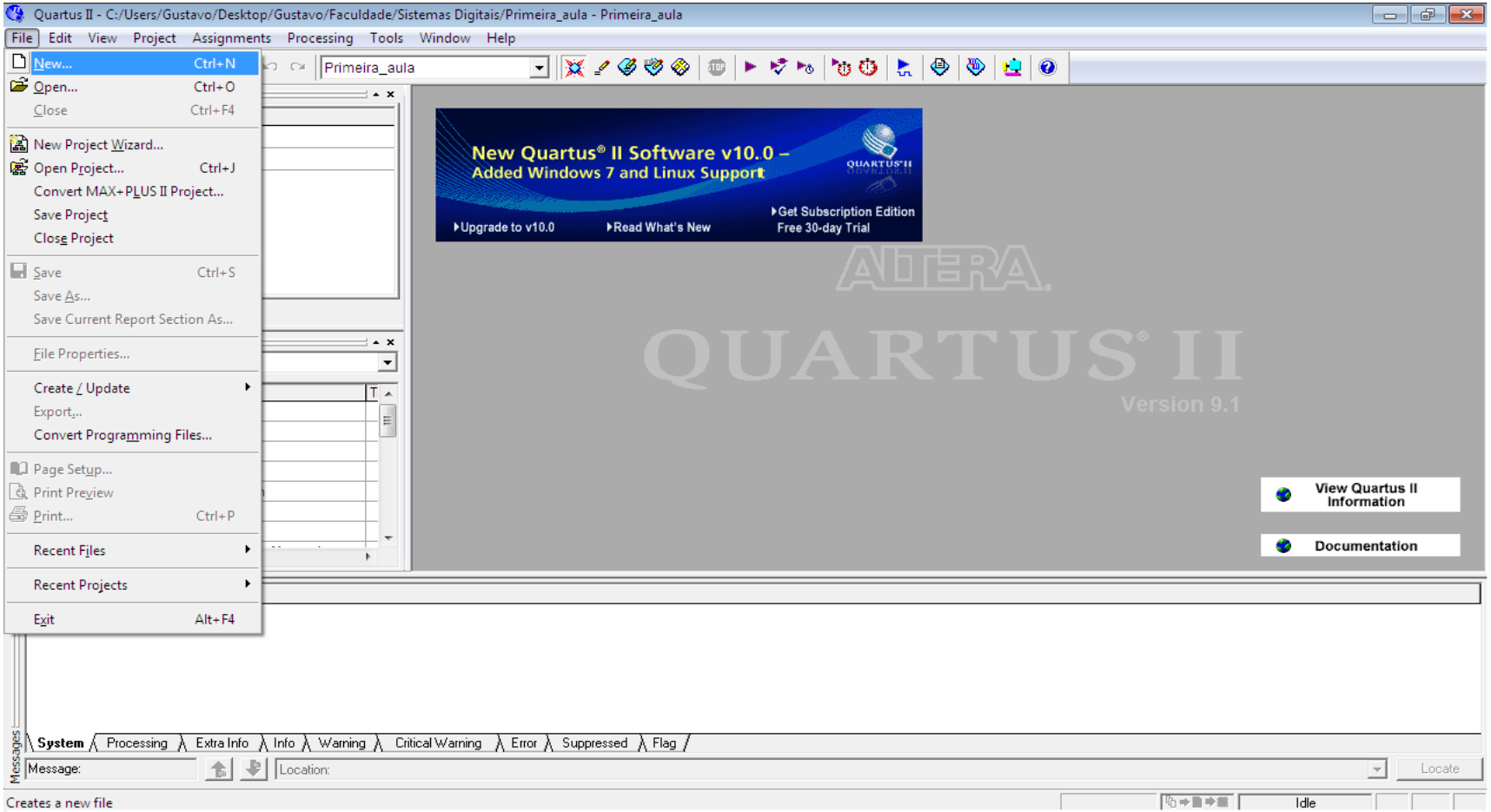
```
module count4 (clk, rst, q);
        input clk, rst;
        output [3:0] q;
        reg [3:0] count;

        always @(posedge clk or negedge rst)
        begin
                if (rst == 1'b0)
                        count <= 4'b0000;
                else if (count == 4'b1110)
                        count <= 4'b0000;
                else
                        count <= count + 1;
        end
                assign q = count;
endmodule
```



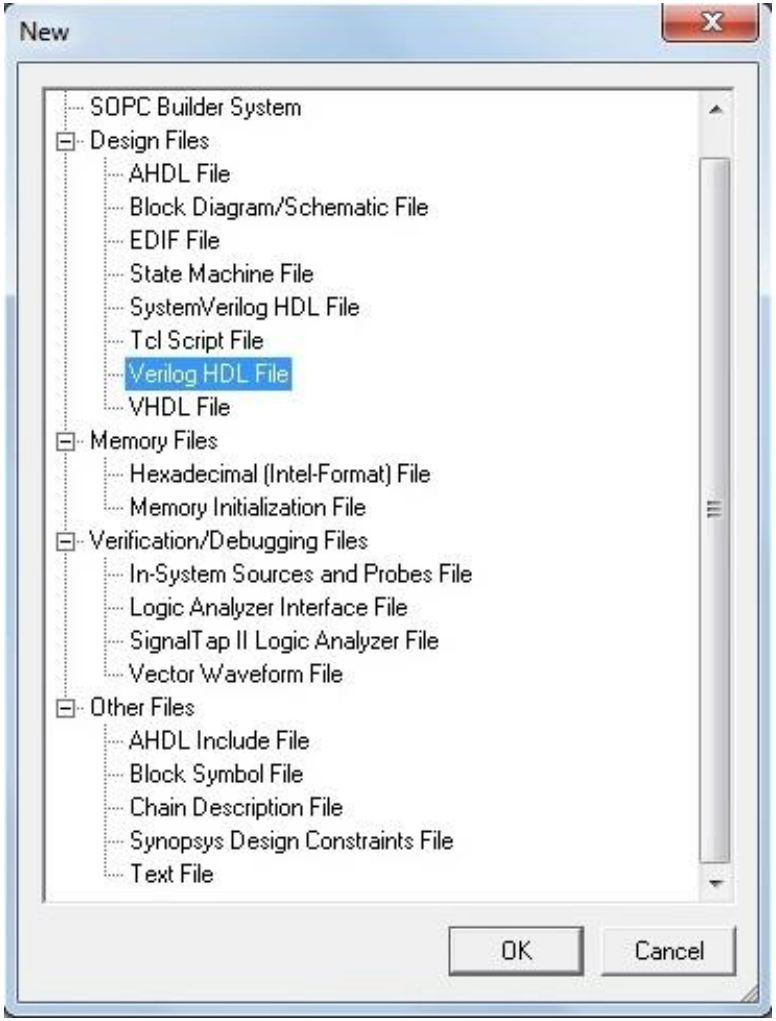
Criando um Arquivo em Verilog no Quartus II...

- Para criar um Arquivo em Verilog, você deve seguir os mesmos passos iniciais... (Inicializar o Quartus e criar um novo projeto);
- Vá em [FILE -> NEW]



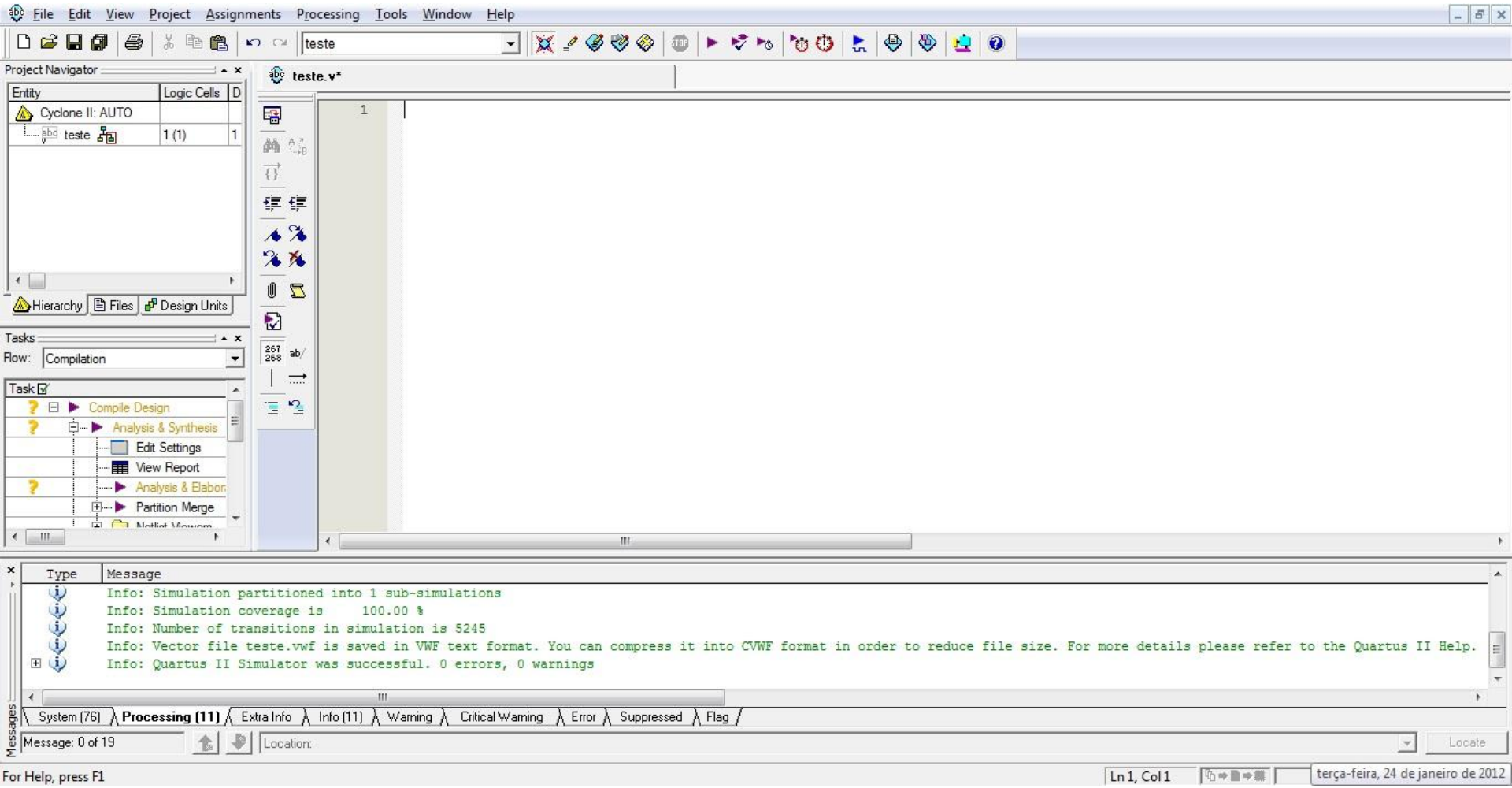
Criando um Arquivo em Verilog no Quartus II...

- Agora vá em [FILE -> NEW -> VERILOG HDL FILE]



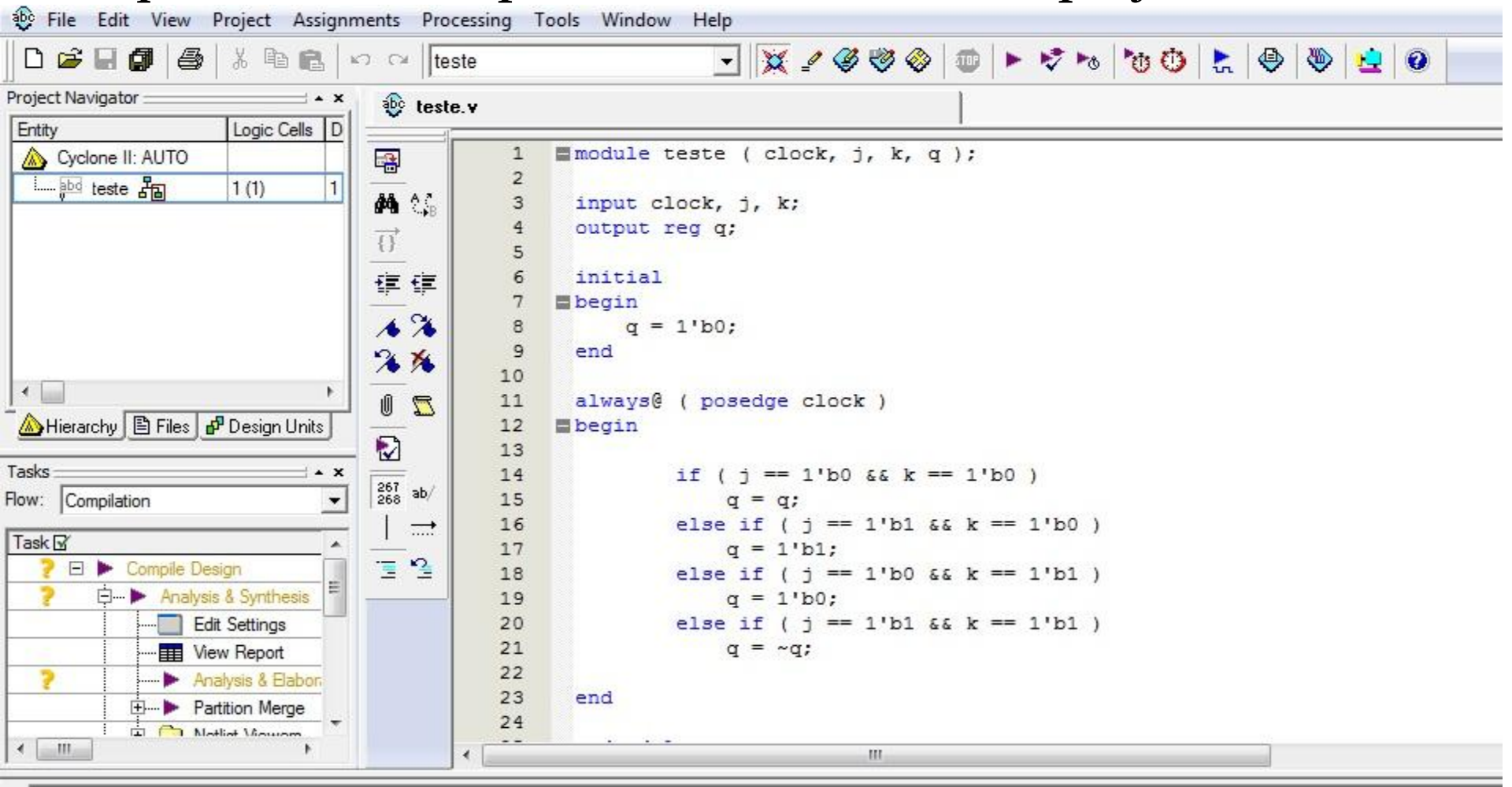
Criando um Arquivo em Verilog no Quartus II...

- Você verá o ambiente abaixo! Nele você pode programar em Verilog a vontade!



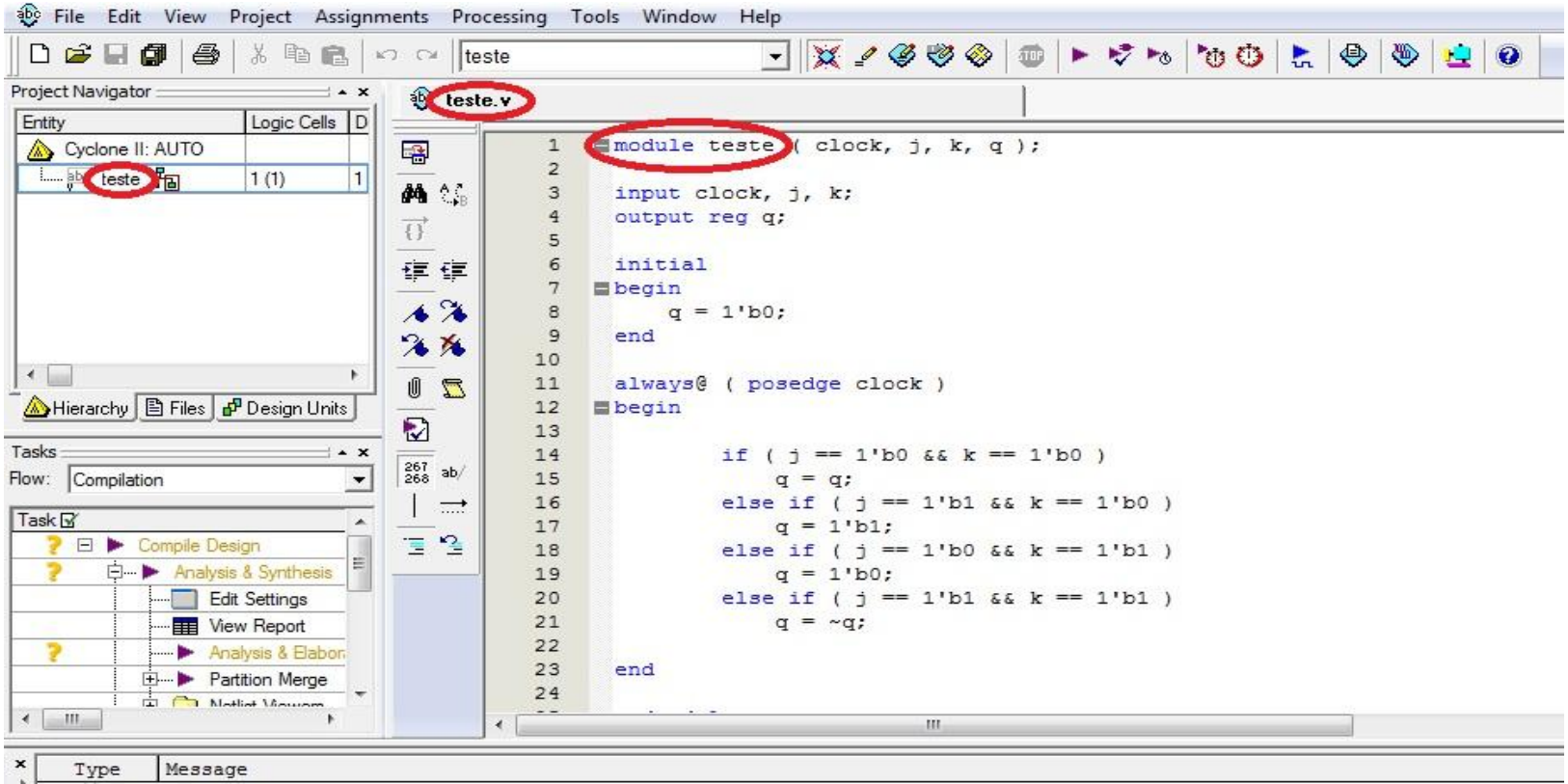
Criando um Arquivo em Verilog no Quartus II...

- Abaixo temos um exemplo de um Flip-Flop tipo JK em Verilog. Depois de terminar de construir o programa, salve o arquivo na mesma pasta onde se encontra o projeto!



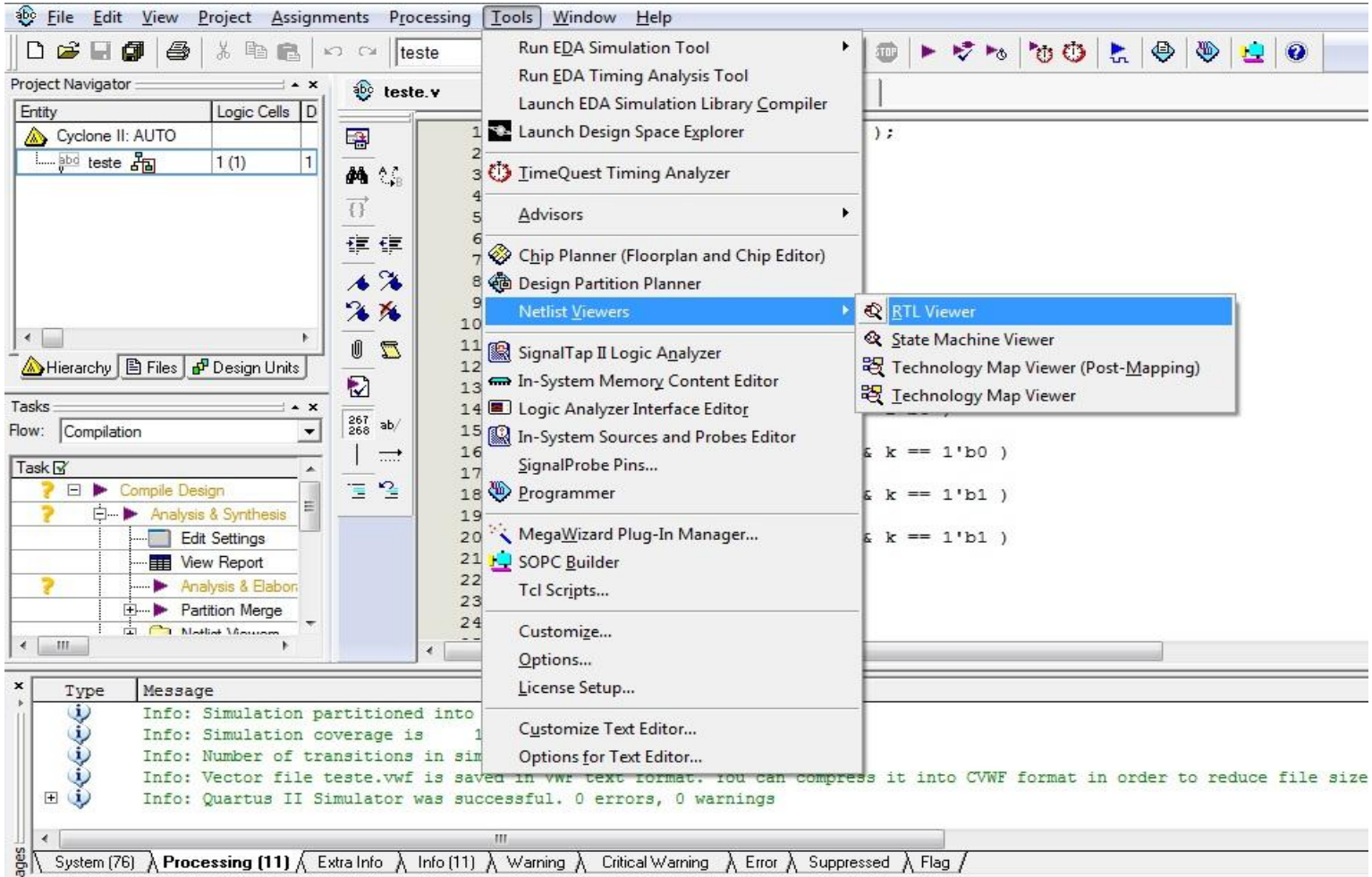
Criando um Arquivo em Verilog no Quartus II...

- Agora, compile e simule o programa da mesma forma como você aprendeu com circuitos em bloco diagrama;
- ATENÇÃO! O Nome do Módulo referente ao Arquivo Verilog que você criou no Quartus devem ter nomes equivalentes para que a compilação funcione! (No exemplo abaixo o nome é "teste").



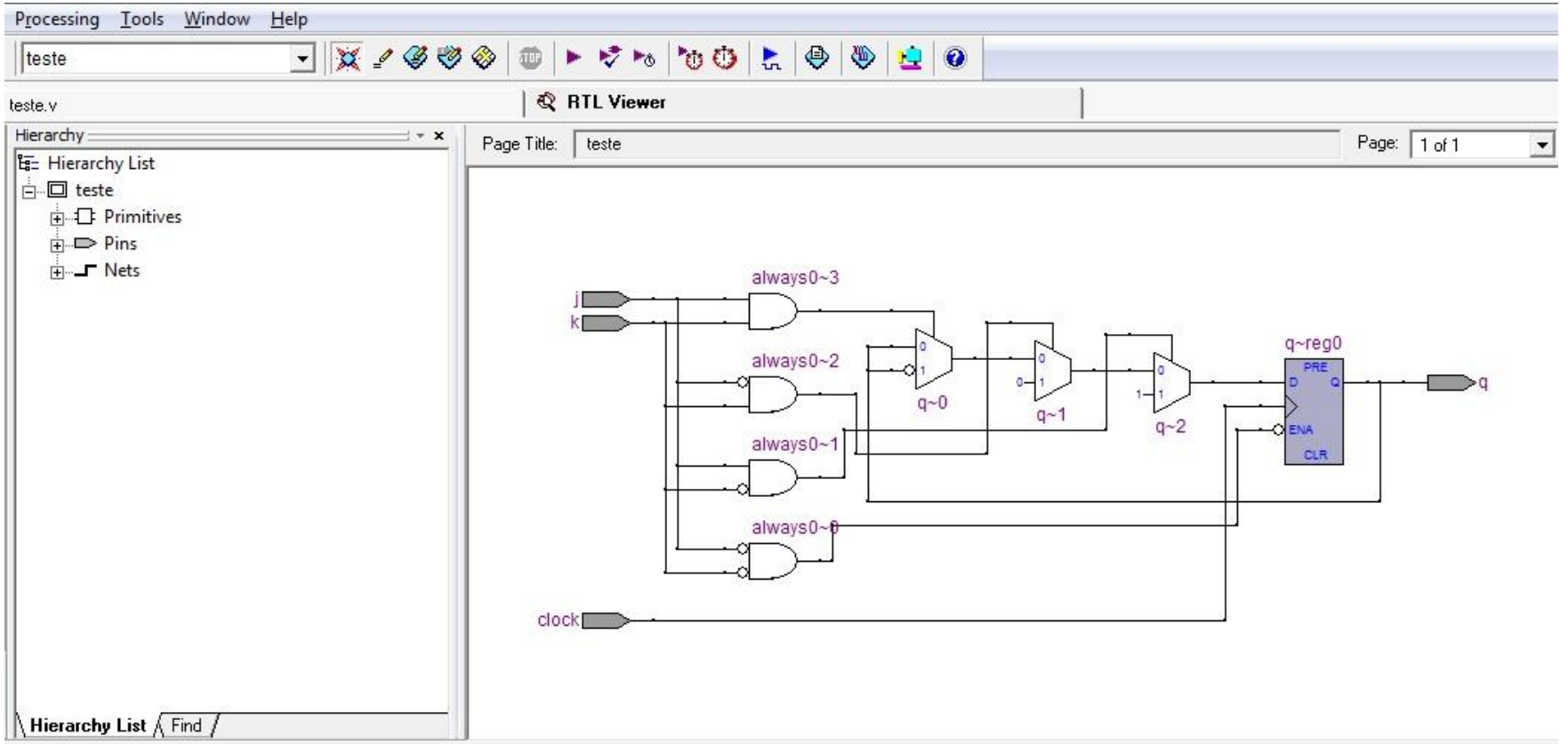
RTL Viewer

- Você também pode ver como ficaria o seu programa em circuito lógico! Basta ir em [TOOLS -> NETLIST VIEWERS -> RTL VIEWER]



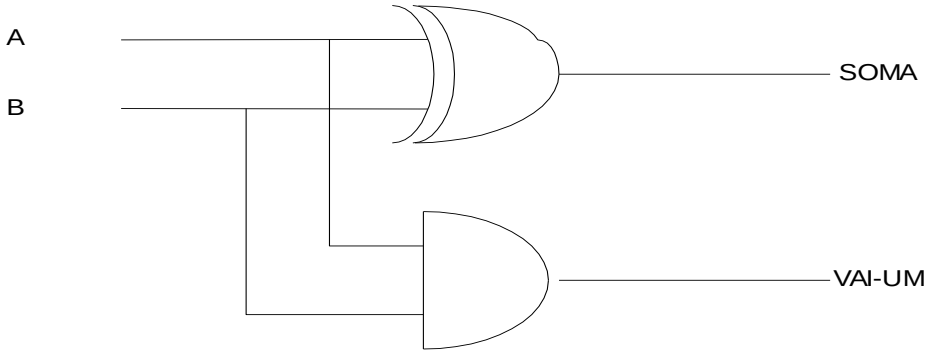
RTL Viewer

- Veja como ficou o nosso Flip-Flop JK! Uma combinação de portas lógicas, MUX's e registrador!

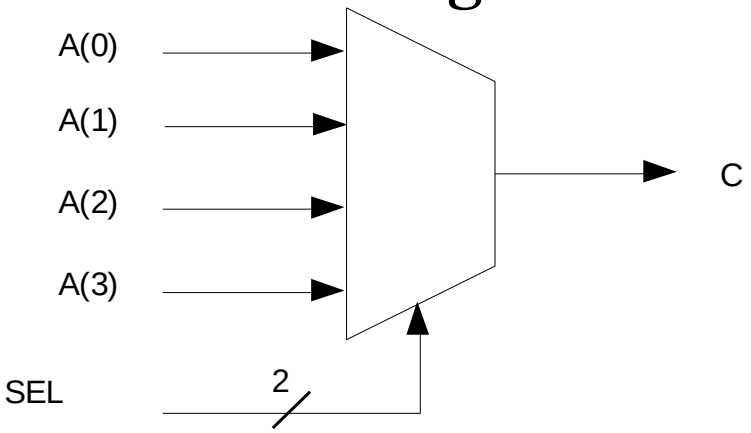


Exercícios

1. Desenvolva um programa em Verilog para o circuito meio somador mostrado na figura abaixo:



2. Desenvolva um programa em Verilog para o circuito multiplexador mostrado na figura abaixo:



Exercícios

3. Implemente uma caixinha em Verilog que faça uma operação de rotação com um operando de entrada (4 bits). A rotação poderá ser para a direita (0) ou para a esquerda (1). A quantidade de vezes que os bits serão rotacionados será colocada na entrada.

   Exemplo: Entradas: Operando – 0110
   Rotação p/ Direita – 0
   Nº de Rotações – 2
   Saída: 1001

4. Implemente uma ALU em Verilog que execute as operações de soma (0) e subtração (1), tendo operandos com 8 bits.