

## 1º Relatório: Projeto ULA de 8 operações

**Anna Luisa de Sá dos Santos**(DRE: 121050671),  
**Larissa Rocha dos Santos**(DRE: 121072380 ),  
**Reginaldo da Silva Cardoso Santos**(DRE: 121039544).

June 9, 2024





# Sumário

<b>Introdução</b>	<b>iii</b>
0.1 Apresentação . . . . .	iii
0.2 Objetivos: . . . . .	iii
0.3 Requisitos: . . . . .	iii
<b>Desenvolvimento</b>	<b>v</b>
0.4 Funcionamento : . . . . .	v
0.5 A eletrônica : . . . . .	v
0.6 O código e simulações : . . . . .	viii
<b>Conclusão</b>	<b>xxxi</b>
0.7 Observações: . . . . .	xxxi
<b>Referências Bibliográficas</b>	<b>xxxiii</b>
0.8 Como usar o VHDL: . . . . .	xxxiii
0.9 Planejamento do Circuito: . . . . .	xxxiii



# Introdução

## 0.1 Apresentação

A Unidade Lógica e Aritmética (ULA) é um componente fundamental em diversos sistemas digitais, desempenhando um papel crucial na execução de operações lógicas e aritméticas essenciais para o funcionamento de processadores e outros dispositivos. Este trabalho tem como objetivo desenvolver uma ULA de 4 bits capaz de realizar oito operações distintas, atendendo a requisitos específicos de um ambiente de hardware baseado no Kit Xilinx Spartan3.

## 0.2 Objetivos:

Como já citado, o trabalho tem como objetivo desenvolver uma Unidade Lógica e Aritmética (ULA) de 4 bits e com 8 operações. Sob este viés, o grupo optou por realizar a ULA com as operações lógicas 'AND', 'NAND', 'OR', 'NOR', 'XOR' e 'XNOR', oferecendo uma gama completa de possibilidades para manipulação de bits. No âmbito aritmético, a ULA incluirá um somador e um subtrator com complemento de 2, que é uma técnica amplamente utilizada em sistemas digitais para simplificar a representação e manipulação de números negativos.

## 0.3 Requisitos:

Os requisitos do projeto especificam que a operação da ULA será selecionada por meio de chaves disponíveis no Kit Xilinx Spartan3, garantindo a interatividade e controle direto sobre as funções executadas. Além disso, os dados de entrada serão gerados por um módulo auxiliar denominado “bancada de testes”, o que facilita a verificação e validação das operações da ULA. Os resultados das operações serão exibidos em binário através de um conjunto

de 8 LEDs no Kit Xilinx Spartan3, proporcionando uma visualização clara e direta do processamento realizado.

As saídas da ULA incluirão, além do resultado da operação, quatro flags: Zero, Negativo, Carry out e Overflow. Estes indicadores são fundamentais para a detecção das condições especiais durante as operações, como a ocorrência de um resultado zero, um resultado negativo, um estouro no bit de transporte e um estouro aritmético, respectivamente. Esses elementos são essenciais para a correta operação e controle de sistemas digitais complexos.

Desta forma, os requisitos do trabalho estão de acordo com o exposto abaixo:

- Operação da ULA selecionada por chaves do Kit Xilinx Spartan3;
- Operações obrigatórias: soma e subtração em complemento a 2;
- Dados de entrada gerados por um módulo “bancada de testes” auxiliar;
- Exibição dos dados em binário no conjunto de 8 LEDs do Kit Xilinx Spartan 3;
- Saídas: resultado e 4 flags: Zero, Negativo, Carry out e Overflow.

# Desenvolvimento

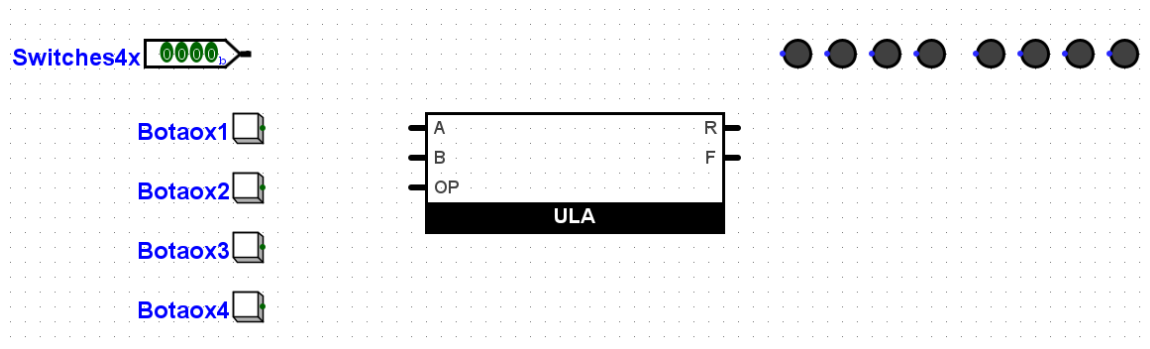
## 0.4 Funcionamento :

As entradas da ULA são geradas por um módulo “bancada de testes”, parte integrante do projeto. As duas entradas são mostradas sequencialmente nos LEDs de saída. Os LEDs mais à esquerda indicam um código auxiliar para as entradas e também indicam as “flags”, em intervalos regulares. Entrada A, Entrada B, Operador e resultado são exibidos nos 4 LEDs mais à direita do kit.

## 0.5 A eletrônica :

Para realizar o projeto, primeiramente o mesmo foi ”rascunhado” utilizando o software ”LogiSim Evolution”, que permitiu a visualização em alto nível do circuito dada as limitações de Entradas e Saídas da FPGA que eram as seguintes :

- 8x LEDS
- 4x SWITCHES
- 4x BOTÕES



Analizando

o módulo e os componentes disponíveis, chegamos nas seguintes especificações:

Entradas :

- $A = (A_3, A_2, A_1, A_0) | A_i \in (0, 1)$
- $B = (B_3, B_2, B_1, B_0) | B_i \in (0, 1)$
- $Op = (Op_2, Op_1, Op_0) | Op_i \in (0, 1)$
- $Clock \in (0, 1)$
- Clock do Registrador A  $\in (0, 1)$
- Clock do Registrador B  $\in (0, 1)$
- Clock do Registrador OP  $\in (0, 1)$
- Clock do seletor de Enable  $\in (0, 1)$

Saídas :

- $Resultado = (R_7, R_6, R_5, R_4, R_3, R_2, R_1, R_0) | R_i \in (0, 1)$

Funções :

- $Op = 000, R = A + B$
- $Op = 001, R = A - B$
- $Op = 010, R = A \text{ AND } B$
- $Op = 011, R = A \text{ NAND } B$
- $Op = 100, R = A \text{ OR } B$
- $Op = 101, R = A \text{ NOR } B$
- $Op = 110, R = A \text{ XOR } B$
- $Op = 111, R = A \text{ XNOR } B$

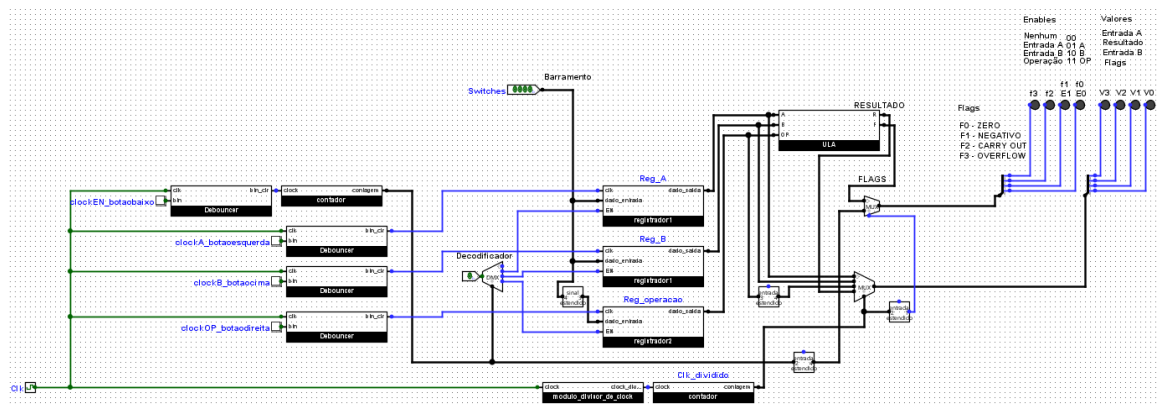
Modulos necessários :

- Precisamos de um barramento para os switches definirem cada uma das entradas de cada vez.

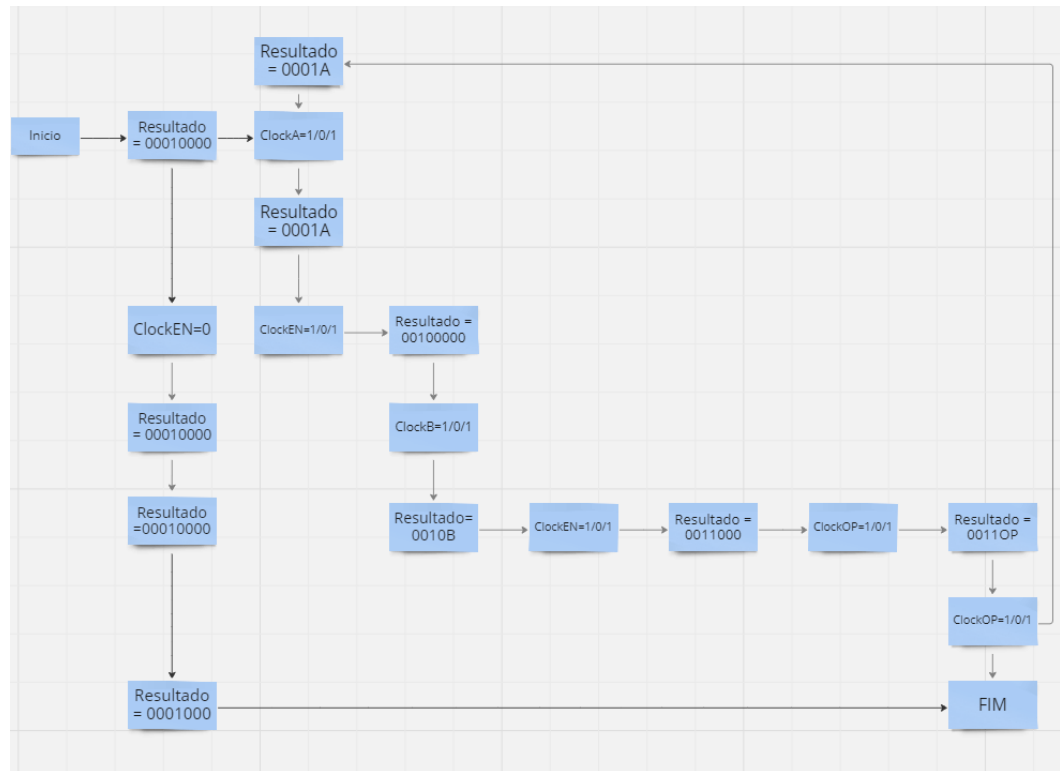


- Muxes para distribuir à cada LED de maneira organizada o Resultado, Operação, Entrada A e Entrada B.
- Um decodificador para selecionar qual das entradas recebe a entrada única.
- Registradores pois precisamos que nossas entradas sejam salvas antes de seu transbordo.
- Contadores já que apesar da nossa ULA ser combinacional queremos mostrar os valores nos LEDS sincronamente.
- E por fim, para evitar problemas físicos, e de temporização, um circuito de Debounce.
- E um circuito capaz de dividir o clock.

Organizando o circuito pelo LogiSim, teremos a seguinte configuração :



Com o nosso design de alto nível pronto, criamos um diagrama de fluxo de dados da seguinte forma :



Portanto, além do clock da FPGA, utilizaremos botões como o clock dos registradores da nossa bancada de testes.

## 0.6 O código e simulações :

Nesta seção visualizaremos os módulos da bancada de teste.

### Módulo divisor de Clock :

Para efetuar a divisão do clock de 50mhz fizemos a seguinte operação:

Sabemos que  $50\text{mhz}(50 \times 10^6 \text{s}^{-1}) = 20\text{ns}$ ,

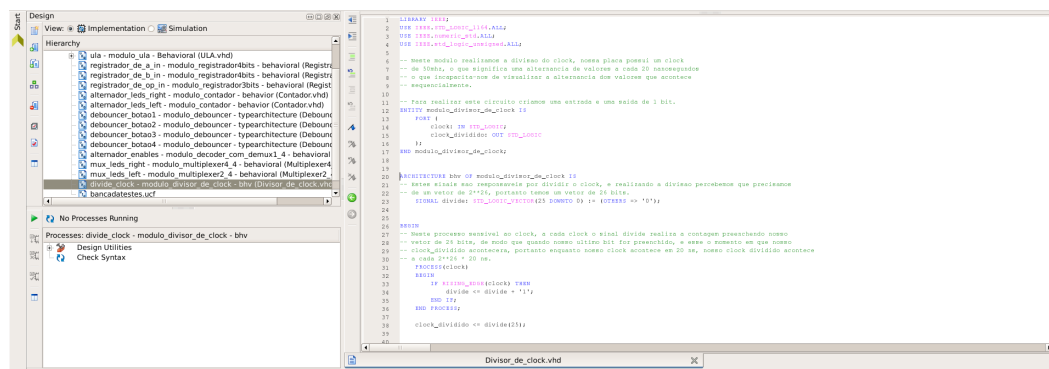
portanto precisamos dividir o nosso clock de modo que seja possível visualizar as saídas em no mínimo milissegundos ou segundos, então utilizamos flip-flops divisores que incrementase 1 a cada clock, e utilizamos 26 entradas,

portanto:

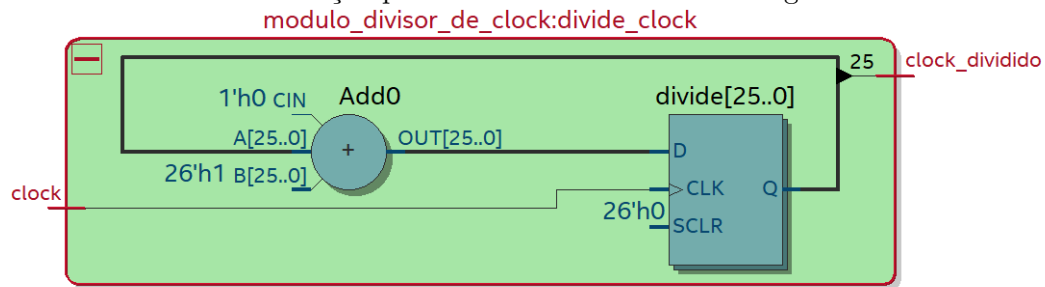
$2^{26} = 67.108.864$ ,  $20\text{ns} = 2^{-8}\text{s}$ , portanto temos um clock de aproximadamente 1,4 segundo, logo que :

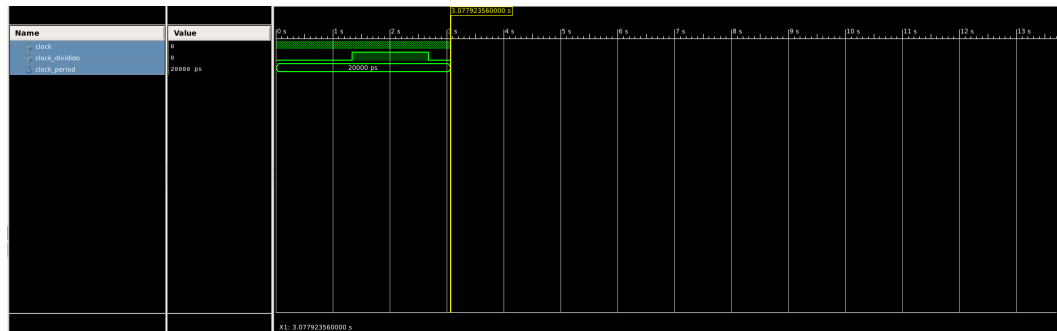
Se contamos 1 em cada  $2^{-8}$  segundo, para determinar quanto tempo levaríamos, faremos a seguinte operação :  $67.108.864 \times 0,00000002\text{s} = 1,34\text{s}$

O código VHDL fica da seguinte forma :



O esquemático RTL e sua simulação podem ser visualizados nas imagens a seguir :





### Módulo debouncer:

O circuito de debouncer age toda vez que apertamos ou soltamos o botão após pressionar, pois possuímos um problema de instabilidade físico nestes instantes de tempo, portanto, para resolvê-lo, utilizaremos dois sinais, um que realiza uma contagem (count) e outro no qual recebe o estado atual do botão (temp), se temp for diferente do valor recebido por temp nosso contador é resetado e temp recebe o novo valor do botão. Se o valor atual do botão e temp forem iguais o contador começa a decrementar até 0, se permanecerem iguais nosso sinal - agora estável - é recebido pelo circuito. A precisão de estabilidade é proporcional ao tamanho de count, optamos por deixar em 1.250.000, pois corresponde à 25ms, foi o mais adequado para realizar as simulações, entretanto ao realizar a apresentação foi alterado para 5.000.000 o que corresponde à 100ms, para termos uma precisão melhor.

$$5.000.000 \times 0,00000002 = 0.100s$$

O código VHDL fica da seguinte forma :

```

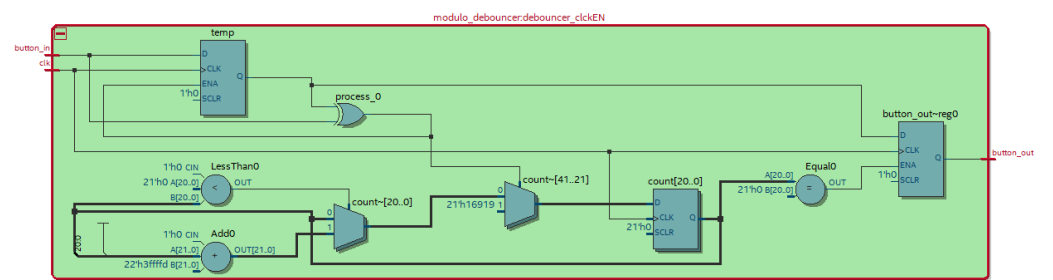
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use IEEE.NUMERIC_STD.ALL;
4
5 entity modulo_debouncer is
6     Port (
7         clk : in STD_LOGIC;
8         button_in : in STD_LOGIC;
9         button_out : out STD_LOGIC
10    );
11 end modulo_debouncer;
12
13 architecture Behavioral of modulo_debouncer is
14     signal count : integer range 0 to 1250000 := 1250000;
15     signal temp : STD_LOGIC := '0';
16
17 begin
18     process (clk)
19     begin
20         if rising_edge(clk) then
21             if button_in /= temp then
22                 count <= 1250000;
23                 temp <= button_in;
24             elsif count > 0 then
25                 count <= count - 1;
26             end if;
27         end if;
28     end process;
29
30     button_out <= temp;
31 end architecture Behavioral;
  
```

0.6. O CÓDIGO E SIMULAÇÕES :

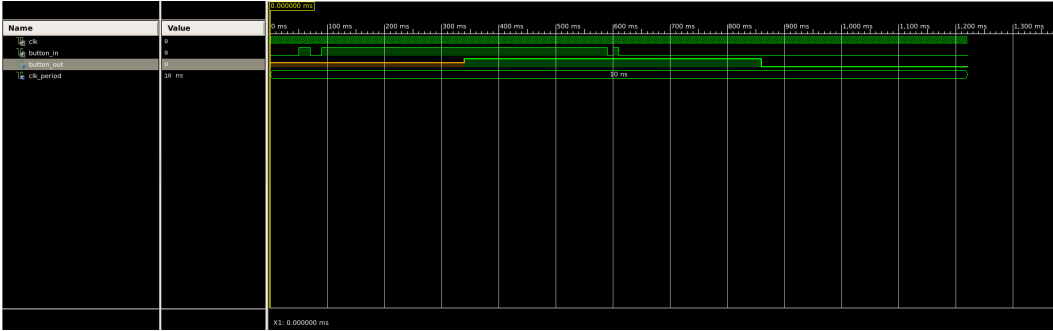
xi

O esquemático RTL e sua simulação podem ser visualizados nas imagens a seguir :

Ex: Debouncer do clockA :

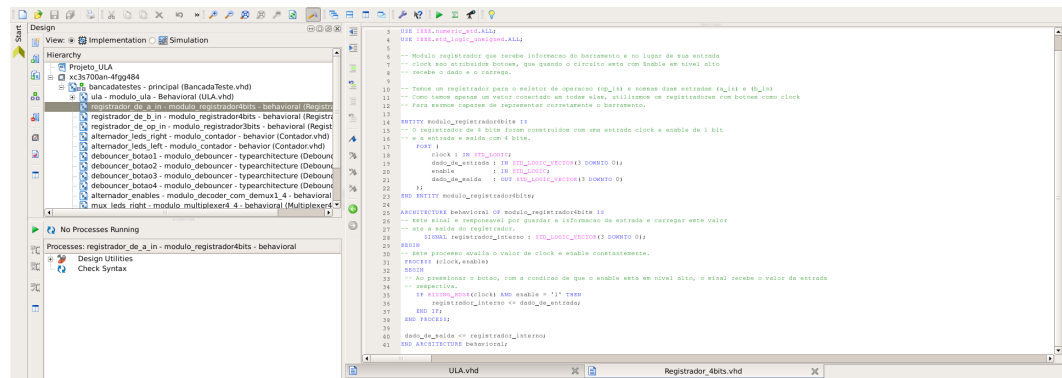


Obs: Para essa simulação foi utilizado  $count = 12.500.000,250ms$



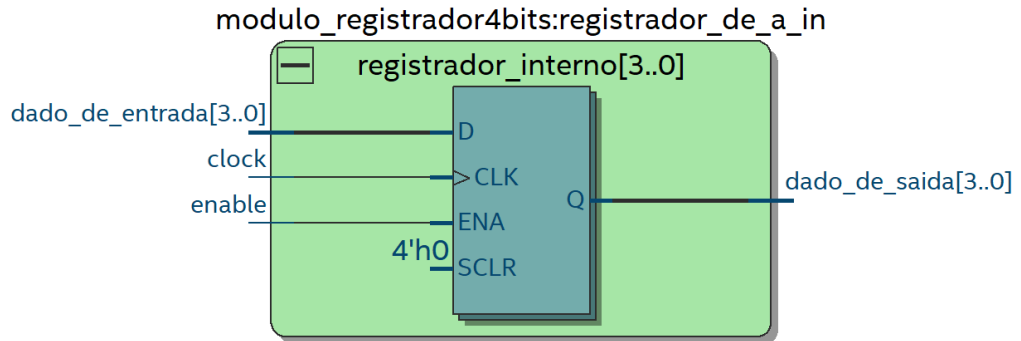
### Módulo registrador :

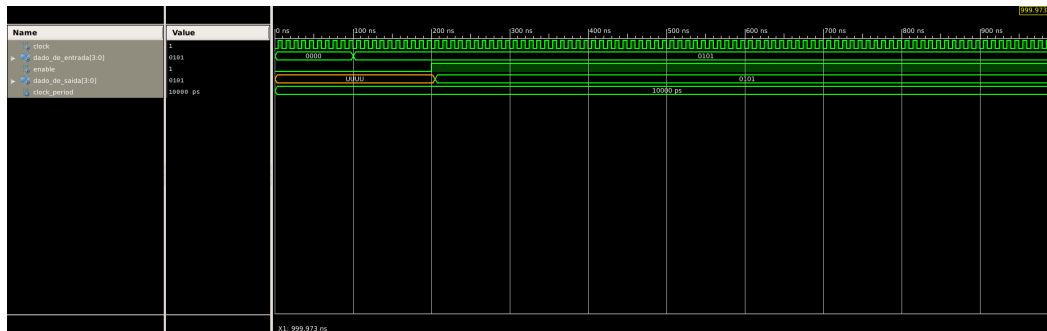
O módulo registrador de 4 e 3 bits possuem uma entrada ENABLE que foi necessária para que pudessemos controlar o barramento, dessa forma somos capazes de utilizar uma única entrada para 3 registradores distintos, dois deles sendo este registrador de 4 bits e apenas um de 3 bits. Que são basicamente 4/3 flip-flops concatenados que registram a entrada a cada clock. O código VHDL fica da seguinte forma :



O esquemático RTL e sua simulação podem ser visualizados nas imagens a seguir :

Ex: Registrador de 4 bits :



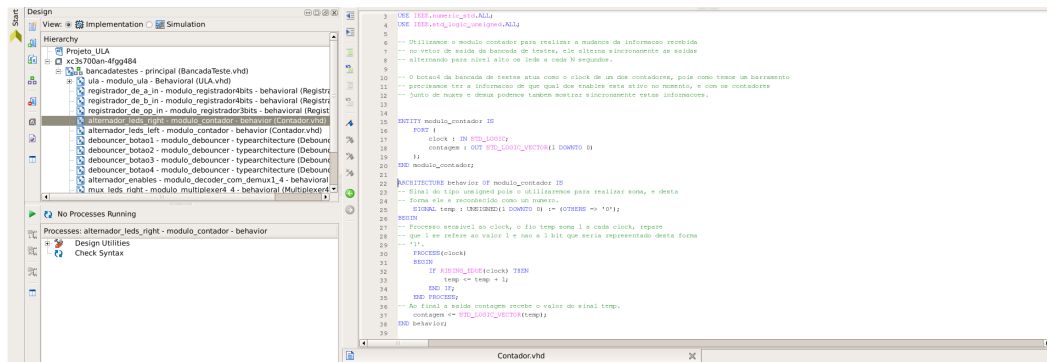


### Módulo Contador :

Foi projetado um contador simples, pois precisávamos alternar nossa saída junto ao clock de maneira controlada, portanto temos apenas um flip-flop que realimenta sua entrada com o valor anterior, obtendo então :

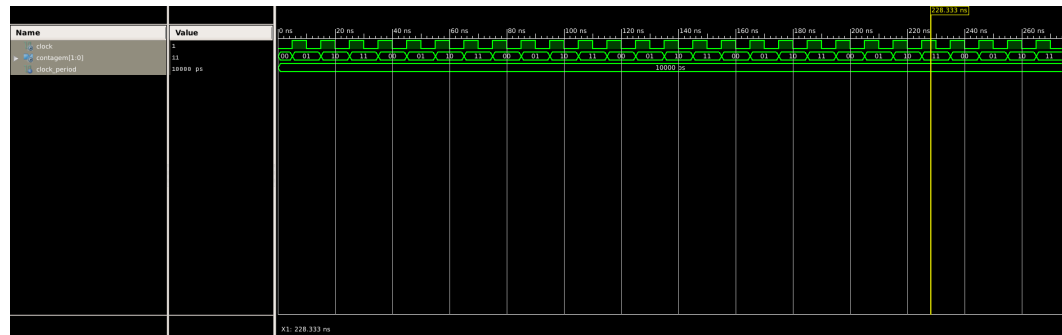
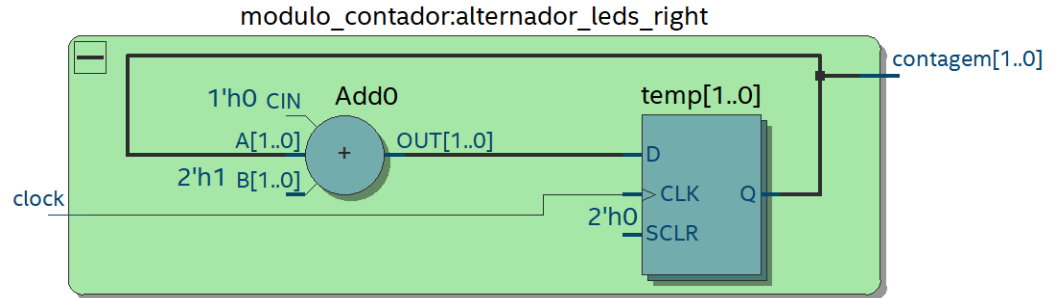
$temp = temp + 1$  obtendo assim um contador que varia de 1 até 4, pois temos apenas 2 flip-flops concatenados.

O código VHDL fica da seguinte forma :



O esquemático RTL e sua simulação podem ser visualizados nas imagens a seguir :

Ex: Contador que altera LEDS da direita :



### Módulo decodificador :

Como decodificador utilizamos uma demux com sua entrada de 1 bit alimentada em nível alto, ela é responsável por habilitar cada um dos Enables dos registradores a cada clock ( que nesse caso seria o nosso botão seletor de Enables fazendo o papel de clock), desse modo temos que :  
O código VHDL fica da seguinte forma :

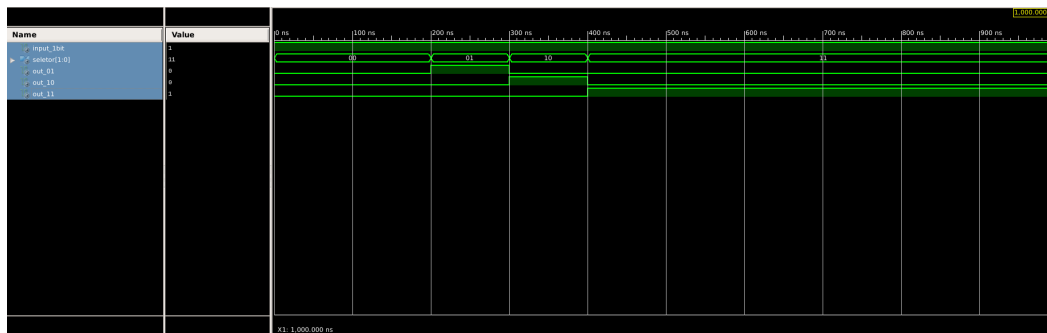
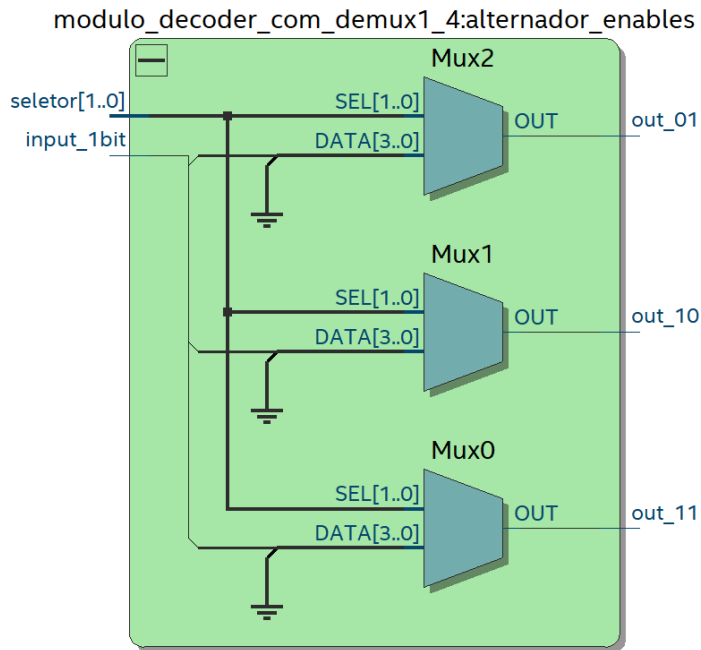
```

1  LIBRARY IEEE;
2  USE IEEE.STD_LOGIC_1164.ALL;
3  USE IEEE.STD_LOGIC_ARITH.ALL;
4  USE IEEE.STD_LOGIC_MISC.ALL;
5
6  -- O módulo decodificador 3bits com demux tem a intenção de alternar entre
7  -- os enables de cada um dos registradores que atuam como um buffer, sendo assim
8  -- como separe de registrar individualmente para os registradores alternadamente.
9  ENTITY modulo_decoder_com_demux_4 IS
10     -- Temos portanto um input de 3 bits que inicialmente recebe '1' configurando assim
11     -- nossa demux como um decodificador.
12     PORT (
13         input_3bits : IN STD_LOGIC_3;
14         output_0 : OUT STD_LOGIC_1;
15         output_1 : OUT STD_LOGIC_1;
16         output_2 : OUT STD_LOGIC_1;
17         output_3 : OUT STD_LOGIC_1;
18     );
19 END ENTITY modulo_decoder_com_demux_4;
20
21 ARCHITECTURE behavioral OF modulo_decoder_com_demux_4 IS
22     SIGNAL
23     -- Temos um processo assimétrico a estado de 3 bits e ao output.
24     PROCESS(input_3bits)
25     BEGIN
26         -- Nesta etapa utilizamos o valor 00 pois precisamos apenas alternar entre 3 enables
27         -- E seria mais adequado emitir o valor inicial 00 de que o valor final 11.
28         ONE <= input_3bits(2);
29         -- Temos as condições que constituem um decodificador/demux onde para cada valor
30         -- selecionado, habilitamos cada um dos enables
31         WHEN input_3bits(2) <= '00' THEN output_0 <= '1'; output_1 <= '0';
32         WHEN input_3bits(2) <= '01' THEN output_1 <= '1'; output_0 <= '0';
33         WHEN input_3bits(2) <= '10' THEN output_2 <= '1'; output_0 <= '0';
34         WHEN input_3bits(2) <= '11' THEN output_3 <= '1'; output_0 <= '0';
35     END PROCESS;
36 END ARCHITECTURE;

```

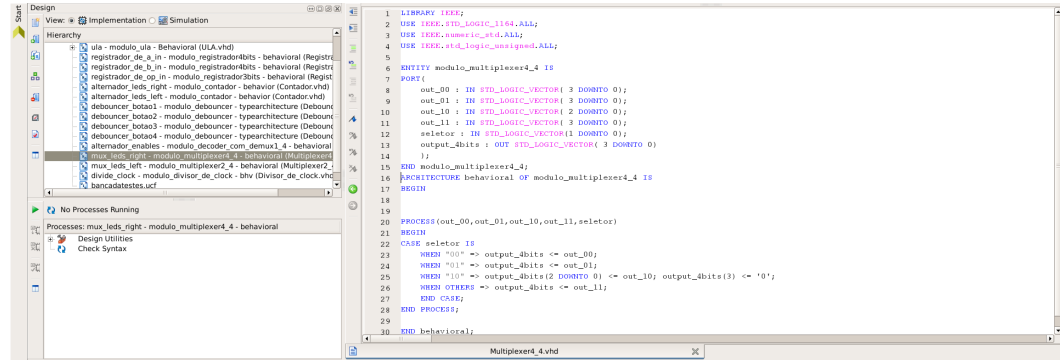


Ex: Decoder feito com Demux :



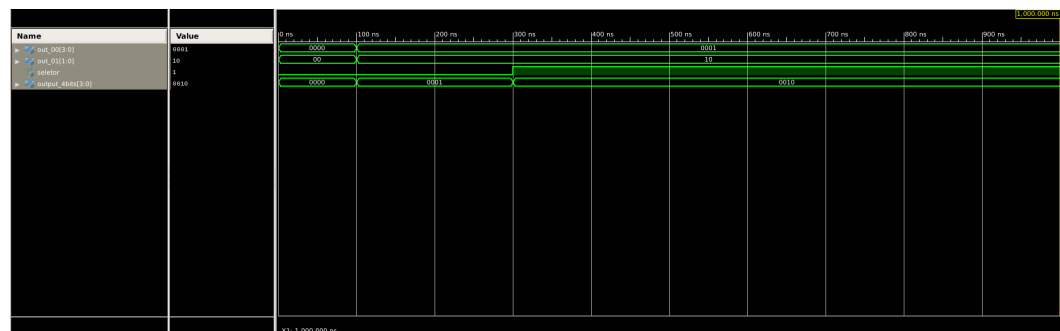
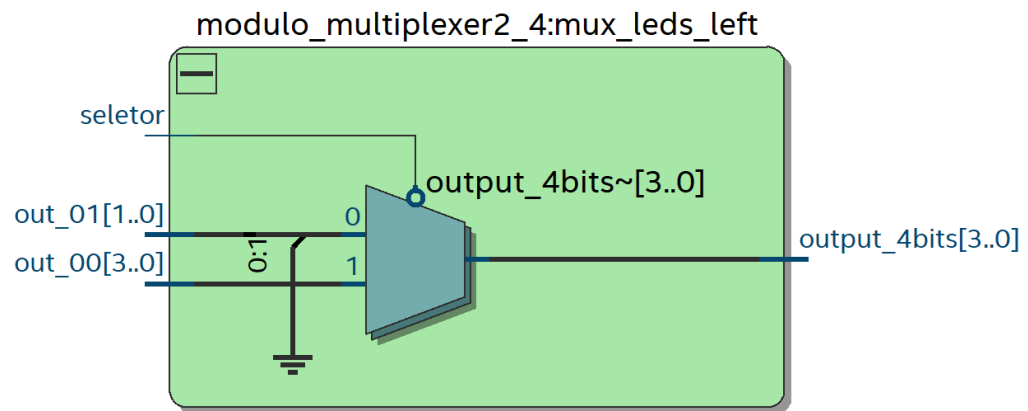
### Módulo Mux :

Utilizamos dois módulos distintos de Mux, a diferença entre elas são somente a quantidade de entradas, sendo uma mux de quatro entradas de 4 bits e a outra com duas entradas de 4 bits, este módulo foi essencial para que pudéssemos distribuir entre os Leds do resultado onde cada flag, entrada, resultado e operação apareceria sequencialmente. O código VHDL da mux de quatro entradas de 4 bits fica da seguinte forma :



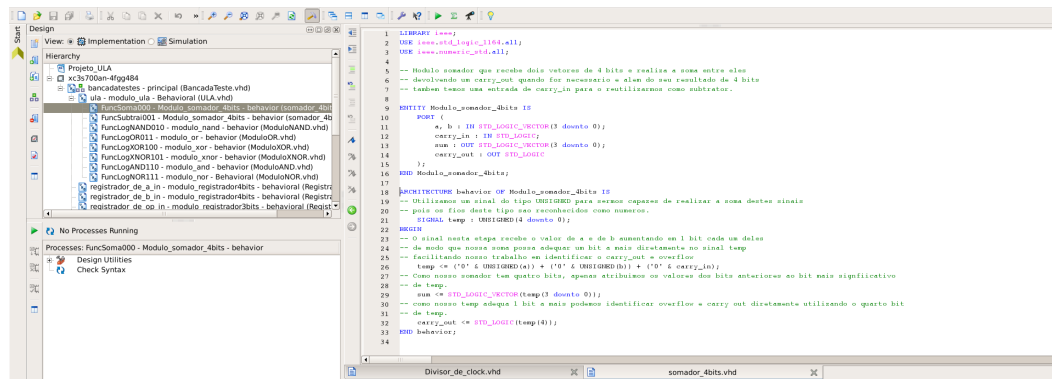
O esquemático RTL e sua simulação podem ser visualizados nas imagens a seguir :

Ex: Mux de duas entradas de 4 bits(obs: uma das entradas foram aproveitadas apenas 2 bits, pois se refere ao Enable selecionado no momento, que vem do contador controlado pelo botão seletor de enables :



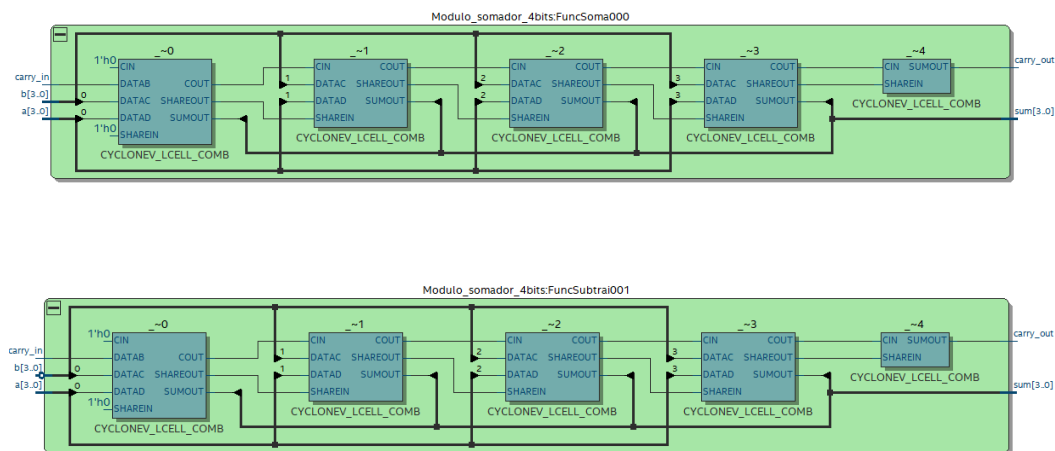
### Módulos de operações aritméticas :

Para as operações aritméticas temos as seguintes : Soma e subtração, para realizar cada um deles projetamos um somador de 4 bits e realizamos o complemento de 2 reaproveitando o mesmo módulo somador habilitando sua entrada *carryin* e negando o sinal da entrada B, sendo assim obtemos além do somador, também um subtrator. O código VHDL fica da seguinte forma :



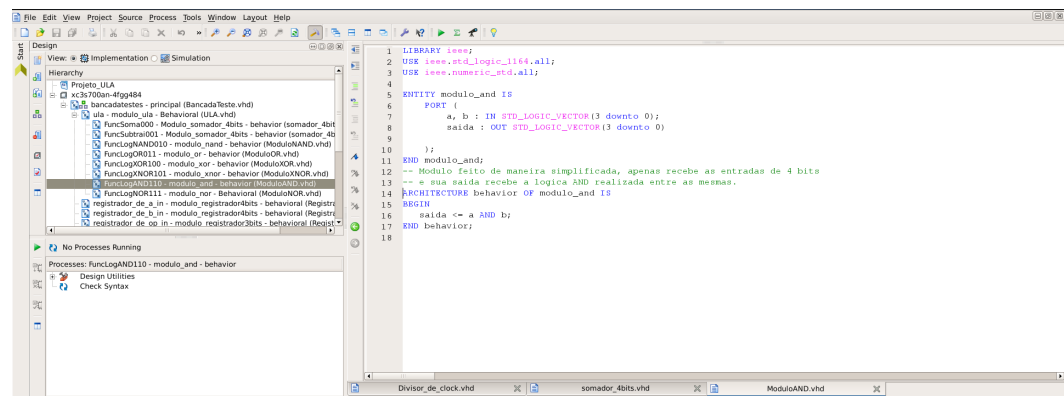
O esquemático RTL e sua simulação podem ser visualizados nas imagens a seguir :

Ex: Mux de duas entradas de 4 bits (obs: uma das entradas foram aproveitadas apenas 2 bits, pois se refere ao Enable selecionado no momento, que vem do contador controlado pelo botão 4 :



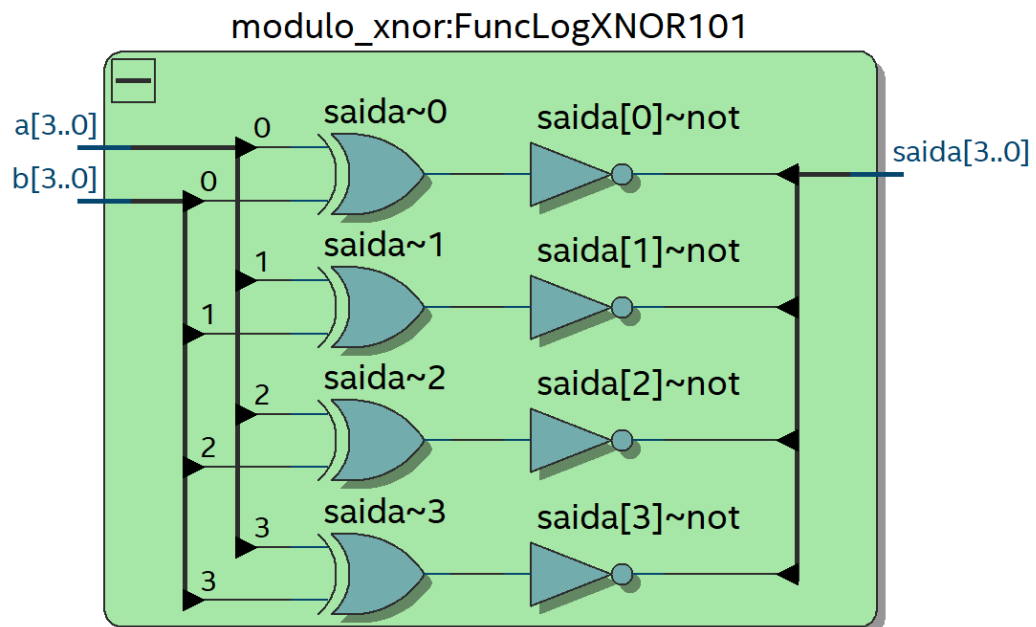
### Módulo de operações lógicas :

Projetando as operações lógicas utilizamos diretamente a praticidade de se utilizar a linguagem VHDL, portanto conseguimos diretamente nossa operação apenas indicando se AND, NAND, OR, NOR, XOR, XNOR para a linguagem, logo temos : Por exemplo o código VHDL da porta lógica AND :



O esquemático RTL e sua simulação podem ser visualizados nas imagens a seguir :

Ex: Operação lógica XNOR :

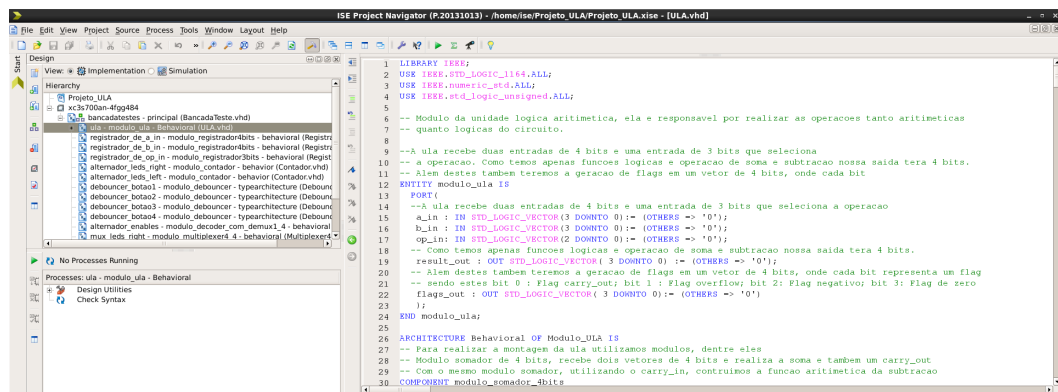


## Módulo ULA :

O projeto do módulo ULA consistiu em :

- Projetar módulos de funções lógicas e aritméticas.
- Utilizar o código base da ULA somente como bancada, ou seja, no seu código consta apenas seus componentes, o mapeamento, os sinais(fios) e a configuração em que cada operação será realizada.

O código VHDL da ULA:

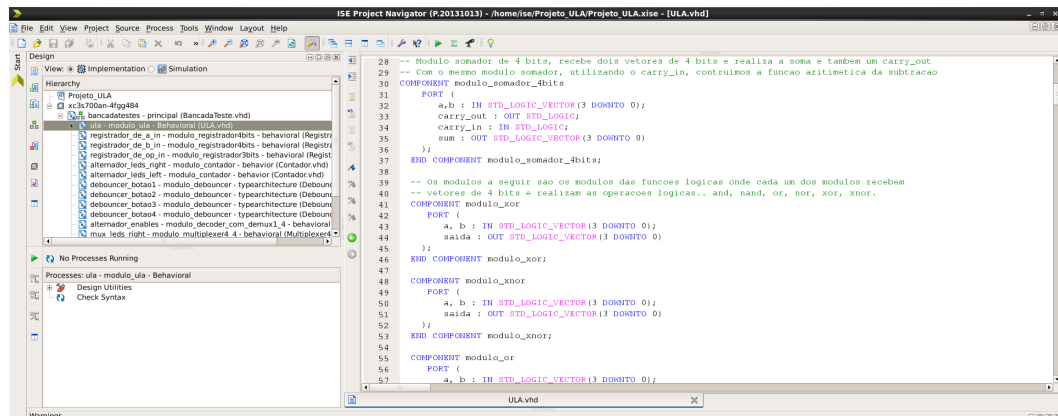


```

1 LIBRARY IEEE;
2 USE IEEE.STD_LOGIC_1164.ALL;
3 USE IEEE.numeric_std.ALL;
4 USE IEEE.std_logic_unsigned.ALL;
5
6 -- Módulo da unidade logica aritmetica, ela e responsavel por realizar as operacoes tanto aritmeticas
7 -- quanto logicas do circuito.
8
9 --A ula recebe duas entradas de 4 bits e uma entrada de 3 bits que seleciona
10 -- a operacao. Como temos apenas funcoes logicas e operacao de soma e subtracao nossa saida tera 4 bits.
11 -- Alen destes tambem teremos a geracao de flags em um vetor de 4 bits, onde cada bit
12 ENTITY modulo_ula IS
13 PORT (
14     --A ula recebe duas entradas de 4 bits e uma entrada de 3 bits que seleciona a operacao
15     a_in : IN STD_LOGIC_VECTOR(3 DOWNTO 0) := (OTHERS => '0');
16     b_in : IN STD_LOGIC_VECTOR(3 DOWNTO 0) := (OTHERS => '0');
17     op_in : IN STD_LOGIC_VECTOR(2 DOWNTO 0) := (OTHERS => '0');
18     -- Como temos apenas funcoes logicas e operacao de soma e subtracao nossa saida tera 4 bits.
19     result_out : OUT STD_LOGIC_VECTOR(3 DOWNTO 0) := (OTHERS => '0');
20     -- Alen destes tambem teremos a geracao de flags em um vetor de 4 bits, onde cada bit representa um flag
21     -- sendo estes bit 0 : Flag carry_out; bit 1 : Flag overflow; bit 2: Flag negativo; bit 3: Flag de zero
22     flags_out : OUT STD_LOGIC_VECTOR(3 DOWNTO 0) := (OTHERS => '0');
23 );
24 END modulo_ula;
25
26 ARCHITECTURE Behavioral OF Modulo_ULA IS
27     -- Para realizar a montagem da ula utilizamos modulos, dentre eles
28     -- Modulo somador de 4 bits, recebe dois vetores de 4 bits e realiza a soma e tambem um carry_out
29     -- Com o mesmo modulo somador, utilizando o carry_in, contruimos a funcao aritmetica da subtracao
30     COMPONENT modulo_somador_4bits
31     PORT (
32         a,b : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
33         carry_out : OUT STD_LOGIC;
34         carry_in : IN STD_LOGIC;
35         sum : OUT STD_LOGIC_VECTOR(3 DOWNTO 0);
36     );
37 END COMPONENT modulo_somador_4bits;
38
39 -- Os modulos a seguir sao os modulos das funcoes logicas onde cada um dos modulos recebem
40 -- vetores de 4 bits e realizam as operacoes logicas.. and, nor, xor, nor, xor, xnor.
41 COMPONENT modulo_xor
42 PORT (
43     a, b : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
44     saida : OUT STD_LOGIC_VECTOR(3 DOWNTO 0);
45 );
46 END COMPONENT modulo_xor;
47
48 COMPONENT modulo_xnor
49 PORT (
50     a, b : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
51     saida : OUT STD_LOGIC_VECTOR(3 DOWNTO 0);
52 );
53 END COMPONENT modulo_xnor;
54
55 COMPONENT modulo_or
56 PORT (
57     a, b : IN STD_LOGIC_VECTOR(3 DOWNTO 0);

```

Declaração de componentes da ULA:



```

28 -- Módulo somador de 4 bits, recebe dois vetores de 4 bits e realiza a soma e tambem um carry_out
29 -- Com o mesmo modulo somador, utilizando o carry_in, contruimos a funcao aritmetica da subtracao
30 COMPONENT modulo_somador_4bits
31 PORT (
32     a,b : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
33     carry_out : OUT STD_LOGIC;
34     carry_in : IN STD_LOGIC;
35     sum : OUT STD_LOGIC_VECTOR(3 DOWNTO 0);
36 );
37 END COMPONENT modulo_somador_4bits;
38
39 -- Os modulos a seguir sao os modulos das funcoes logicas onde cada um dos modulos recebem
40 -- vetores de 4 bits e realizam as operacoes logicas.. and, nor, xor, nor, xor, xnor.
41 COMPONENT modulo_xor
42 PORT (
43     a, b : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
44     saida : OUT STD_LOGIC_VECTOR(3 DOWNTO 0);
45 );
46 END COMPONENT modulo_xor;
47
48 COMPONENT modulo_xnor
49 PORT (
50     a, b : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
51     saida : OUT STD_LOGIC_VECTOR(3 DOWNTO 0);
52 );
53 END COMPONENT modulo_xnor;
54
55 COMPONENT modulo_or
56 PORT (
57     a, b : IN STD_LOGIC_VECTOR(3 DOWNTO 0);

```

```

-- Fios que carregam os resultados das somas e da subtração, os fios possuem um bit a mais(5 bits) pois,
-- então assim podemos identificar de maneira mais simplificada a ocorrência de carry-out e overflow.
SIGNAL soma, subtrai: STD_LOGIC_VECTOR (4 DOWNTO 0); (OTHERS <= '0');

-- Fios que carregam as saídas de cada um dos módulos das funções lógicas.
SIGNAL func_xor,func_nor,func_and,func_nand,func_or,func_not: STD_LOGIC_VECTOR (3 DOWNTO 0); (OTHERS <= '0');

-- Este sinal foi essencial para sermos capazes de fazer o subtrator, pois este sinal carrega o valor
-- de B invertido, então, Bx := se o valor de B = "0000" este sinal carrega o valor "1111";
SIGNAL bnegado: STD_LOGIC_VECTOR (3 DOWNTO 0); (OTHERS <= '0');

-- Fios que carregam os valores entre os módulos -----
begin
    -- Comportamento da ULA, esta seção mostra o mapeamento do circuito, onde cada entrada se conecta
    -- bnegado(3 DOWNTO 0) <= NOT(B_in); -- Negar a entrada B de modo que nosso somador se torne um somador realizando o comp
    -- lemento de dois complementos quando necessário. Assim como antes tínhamos quatro vetores que eram
    -- Função00000 : modulos_somador_bits PORT MAP (a => a_in, b => b_in,sum => soma(3 DOWNTO 0),carry_out => soma(4), c
    -- ); -- Modulo subtrator, utilizando o somador de 4 bits, o valor de b negado e o carry_in recebendo nilai lógico alto, co
    -- mo Subtrai001 : modulos_somador_bits PORT MAP (a => a_in, b => bnegado(3 DOWNTO 0), sum => subtrai(3 DOWNTO 0), ca
    -- );

    -- Aqui podemos ver os módulos das funções lógicas com cada uma das suas entradas atribuídas no circuito.
    FuncLogica001 : module AND_F (a => a_in, b => b_in, saida => func_and);
    FuncLogica011 : module OR_F (a => a_in, b => b_in, saida => func_or);
    FuncLogica010 : module XOR_F (a => a_in, b => b_in, saida => func_xor);
    FuncLogica010 : module NOR_F (a => a_in, b => b_in, saida => func_nor);
    FuncLogica010 : module NAND_F (a => a_in, b => b_in, saida => func_nand);
    FuncLogica010 : module NOT_F (a => a_in, b => b_in, saida => func_not);

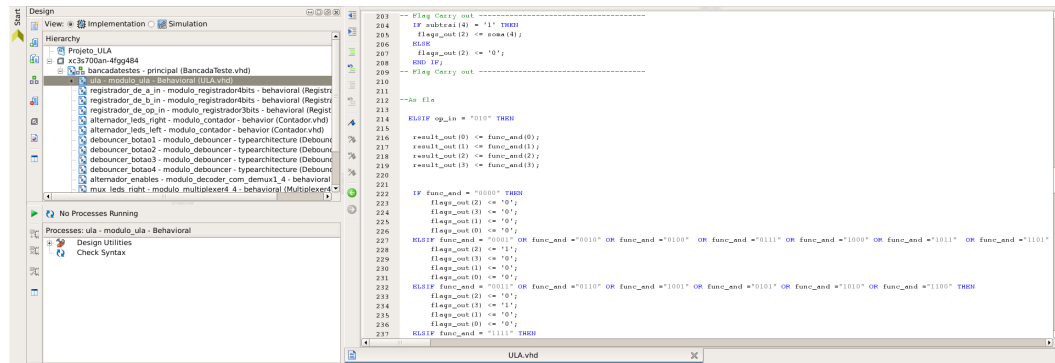
    -- Este processo permite que cada uma das entradas e sinais listados sejam sensíveis a mudança, re-executando o proces
    PROCES (op_in,soma,subtrai,a_in,b_in,func_and,func_nand,func_or,func_xor,func_nor)
end

```

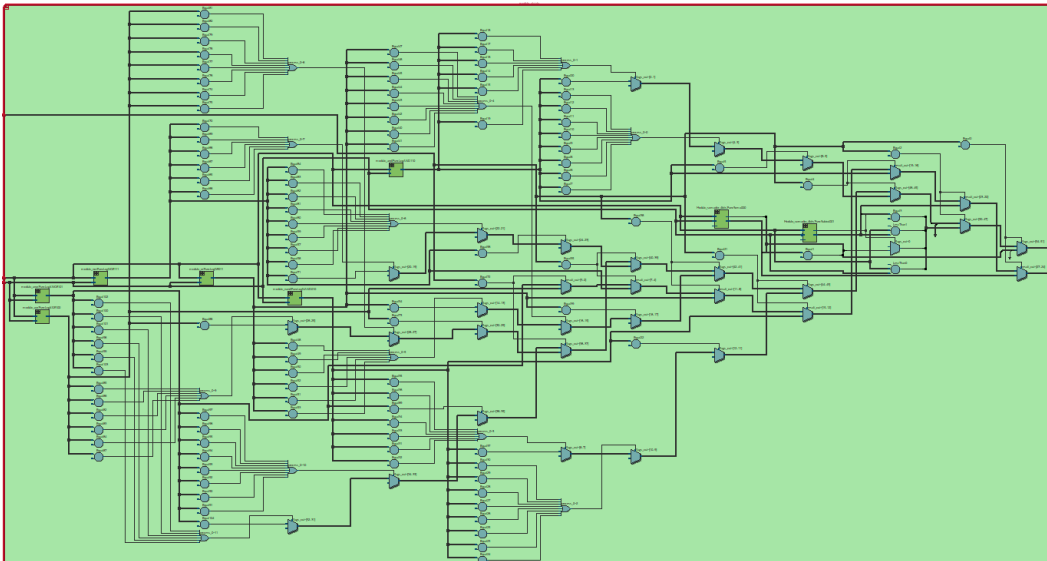
- ```

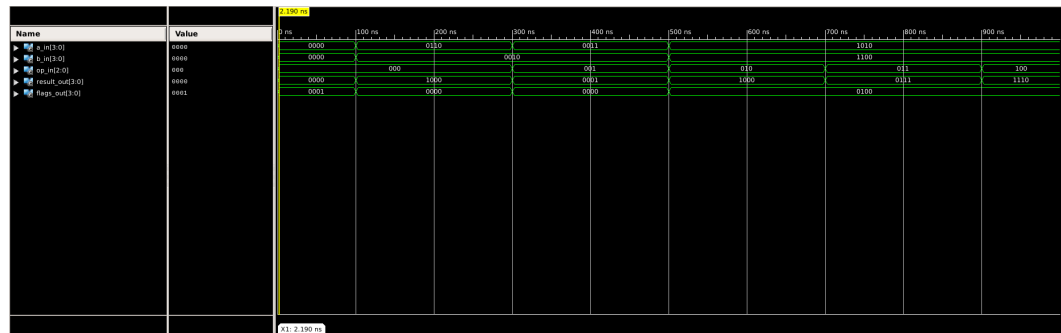
114
115 -- Esta processo permite que cada uma das entradas a serem listadas sejam sensíveis a mudança, re-avaliando o processo a cada mudança
116 PROCESS (op_in, soma, subtra1, a_in, b_in, func_and, func_orand, func_xor, func_nor, func_nand)
117 BEGIN
118 -- Nesta etapa do processo definimos qual operação lógica a utilizará e se realizada dependendo distamente
119 -- do valor de op_in, que por ter 8 operações, o seu valor de 0 a 7, sendo o seu primeiro valor 000 e último 111.
120 -- o que o sistema o que queremos.
121
122 --Portanto atribuímos o valor 000 a soma.
123 IF op_in = "0000" THEN
124     result_out(0) <= soma(0);
125     result_out(1) <= soma(1);
126     result_out(2) <= soma(2);
127     result_out(3) <= '0';
128
129 -- Nesta etapa definimos as instruções para que o circuito identifique cada uma das flags.
130
131 -- Flag Zero -----
132 -- Se nosso resultado for "0000" nosso flag zero sinaliza.
133 IF soma = "000000" THEN
134     flag_zout(0) <= '1';
135 ELSE
136     flag_zout(0) <= '0';
137 END IF;
138
139 -- Flag Zero -----
140
141 -- Flag Overflow -----
142 -- Esta flag sinaliza quando nossa entrada não consegue ser representada corretamente no nosso valor de saída.
143 -- Para a soma de 4 bits a soma acontece de forma frequente, o que se ocorre o carry-out, então podemos
144 -- utilizar nosso quinto bit para identificar igualmente o doar.
145
146 IF soma(4) = '1' THEN -- Adição
147     flag_ovout(0) <= '1';
148 ELSE
149     flag_ovout(0) <= '0';
150 END IF;
151
152 -- Flag Overflow -----
153
154 -- Flag Carry -----
155 -- Esta flag sinaliza quando o resultado da soma ultrapassar o valor máximo representável por 4 bits, ou seja, 15.
156 -- Para isso, vamos utilizar o bit 4 do resultado da soma para identificar o carry-out.
157
158 IF soma(4) = '1' THEN
159     flag_cout(0) <= '1';
160 ELSE
161     flag_cout(0) <= '0';
162 END IF;
163
164 -- Flag Carry -----
165
166 -- Flag Sign -----
167 -- Esta flag sinaliza quando o resultado da soma for negativo, ou seja, quando o bit de sinal (bit 3) for 1.
168 -- Para isso, vamos utilizar o bit 3 do resultado da soma para identificar o sinal.
169
170 IF soma(3) = '1' THEN
171     flag_sout(0) <= '1';
172 ELSE
173     flag_sout(0) <= '0';
174 END IF;
175
176 -- Flag Sign -----
177
178 -- Flag Paridade -----
179 -- Esta flag sinaliza quando o resultado da soma for par ou ímpar.
180 -- Para isso, vamos utilizar o bit 0 do resultado da soma para identificar a paridade.
181
182 IF soma(0) = '1' THEN
183     flag_pout(0) <= '1';
184 ELSE
185     flag_pout(0) <= '0';
186 END IF;
187
188 -- Flag Paridade -----
189
190 -- Flag Zero -----
191 -- Se nosso resultado for "0000" nosso flag zero sinaliza.
192 IF soma = "000000" THEN
193     flag_zout(0) <= '1';
194 ELSE
195     flag_zout(0) <= '0';
196 END IF;
197
198 -- Flag Zero -----
199
200 -- Flag Overflow -----
201 -- Esta flag sinaliza quando nossa entrada não consegue ser representada corretamente no nosso valor de saída.
202 -- Para a soma de 4 bits a soma acontece de forma frequente, o que se ocorre o carry-out, então podemos
203 -- utilizar nosso quinto bit para identificar igualmente o doar.
204
205 IF soma(4) = '1' THEN -- Adição
206     flag_ovout(0) <= '1';
207 ELSE
208     flag_ovout(0) <= '0';
209 END IF;
210
211 -- Flag Overflow -----
212
213 -- Flag Carry -----
214 -- Esta flag sinaliza quando o resultado da soma ultrapassar o valor máximo representável por 4 bits, ou seja, 15.
215 -- Para isso, vamos utilizar o bit 4 do resultado da soma para identificar o carry-out.
216
217 IF soma(4) = '1' THEN
218     flag_cout(0) <= '1';
219 ELSE
220     flag_cout(0) <= '0';
221 END IF;
222
223 -- Flag Carry -----
224
225 -- Flag Sign -----
226 -- Esta flag sinaliza quando o resultado da soma for negativo, ou seja, quando o bit de sinal (bit 3) for 1.
227 -- Para isso, vamos utilizar o bit 3 do resultado da soma para identificar o sinal.
228
229 IF soma(3) = '1' THEN
230     flag_sout(0) <= '1';
231 ELSE
232     flag_sout(0) <= '0';
233 END IF;
234
235 -- Flag Sign -----
236
237 -- Flag Paridade -----
238 -- Esta flag sinaliza quando o resultado da soma for par ou ímpar.
239 -- Para isso, vamos utilizar o bit 0 do resultado da soma para identificar a paridade.
240
241 IF soma(0) = '1' THEN
242     flag_pout(0) <= '1';
243 ELSE
244     flag_pout(0) <= '0';
245 END IF;
246
247 -- Flag Paridade -----
248
249 -- Flag Zero -----
250 -- Se nosso resultado for "0000" nosso flag zero sinaliza.
251 IF soma = "000000" THEN
252     flag_zout(0) <= '1';
253 ELSE
254     flag_zout(0) <= '0';
255 END IF;
256
257 -- Flag Zero -----
258
259 -- Flag Overflow -----
260 -- Esta flag sinaliza quando nossa entrada não consegue ser representada corretamente no nosso valor de saída.
261 -- Para a soma de 4 bits a soma acontece de forma frequente, o que se ocorre o carry-out, então podemos
262 -- utilizar nosso quinto bit para identificar igualmente o doar.
263
264 IF soma(4) = '1' THEN -- Adição
265     flag_ovout(0) <= '1';
266 ELSE
267     flag_ovout(0) <= '0';
268 END IF;
269
270 -- Flag Overflow -----
271
272 -- Flag Carry -----
273 -- Esta flag sinaliza quando o resultado da soma ultrapassar o valor máximo representável por 4 bits, ou seja, 15.
274 -- Para isso, vamos utilizar o bit 4 do resultado da soma para identificar o carry-out.
275
276 IF soma(4) = '1' THEN
277     flag_cout(0) <= '1';
278 ELSE
279     flag_cout(0) <= '0';
280 END IF;
281
282 -- Flag Carry -----
283
284 -- Flag Sign -----
285 -- Esta flag sinaliza quando o resultado da soma for negativo, ou seja, quando o bit de sinal (bit 3) for 1.
286 -- Para isso, vamos utilizar o bit 3 do resultado da soma para identificar o sinal.
287
288 IF soma(3) = '1' THEN
289     flag_sout(0) <= '1';
290 ELSE
291     flag_sout(0) <= '0';
292 END IF;
293
294 -- Flag Sign -----
295
296 -- Flag Paridade -----
297 -- Esta flag sinaliza quando o resultado da soma for par ou ímpar.
298 -- Para isso, vamos utilizar o bit 0 do resultado da soma para identificar a paridade.
299
300 IF soma(0) = '1' THEN
301     flag_pout(0) <= '1';
302 ELSE
303     flag_pout(0) <= '0';
304 END IF;
305
306 -- Flag Paridade -----
307
308 -- Flag Zero -----
309 -- Se nosso resultado for "0000" nosso flag zero sinaliza.
310 IF soma = "000000" THEN
311     flag_zout(0) <= '1';
312 ELSE
313     flag_zout(0) <= '0';
314 END IF;
315
316 -- Flag Zero -----
317
318 -- Flag Overflow -----
319 -- Esta flag sinaliza quando nossa entrada não consegue ser representada corretamente no nosso valor de saída.
320 -- Para a soma de 4 bits a soma acontece de forma frequente, o que se ocorre o carry-out, então podemos
321 -- utilizar nosso quinto bit para identificar igualmente o doar.
322
323 IF soma(4) = '1' THEN -- Adição
324     flag_ovout(0) <= '1';
325 ELSE
326     flag_ovout(0) <= '0';
327 END IF;
328
329 -- Flag Overflow -----
330
331 -- Flag Carry -----
332 -- Esta flag sinaliza quando o resultado da soma ultrapassar o valor máximo representável por 4 bits, ou seja, 15.
333 -- Para isso, vamos utilizar o bit 4 do resultado da soma para identificar o carry-out.
334
335 IF soma(4) = '1' THEN
336     flag_cout(0) <= '1';
337 ELSE
338     flag_cout(0) <= '0';
339 END IF;
340
341 -- Flag Carry -----
342
343 -- Flag Sign -----
344 -- Esta flag sinaliza quando o resultado da soma for negativo, ou seja, quando o bit de sinal (bit 3) for 1.
345 -- Para isso, vamos utilizar o bit 3 do resultado da soma para identificar o sinal.
346
347 IF soma(3) = '1' THEN
348     flag_sout(0) <= '1';
349 ELSE
350     flag_sout(0) <= '0';
351 END IF;
352
353 -- Flag Sign -----
354
355 -- Flag Paridade -----
356 -- Esta flag sinaliza quando o resultado da soma for par ou ímpar.
357 -- Para isso, vamos utilizar o bit 0 do resultado da soma para identificar a paridade.
358
359 IF soma(0) = '1' THEN
360     flag_pout(0) <= '1';
361 ELSE
362     flag_pout(0) <= '0';
363 END IF;
364
365 -- Flag Paridade -----
366
367 -- Flag Zero -----
368 -- Se nosso resultado for "0000" nosso flag zero sinaliza.
369 IF soma = "000000" THEN
370     flag_zout(0) <= '1';
371 ELSE
372     flag_zout(0) <= '0';
373 END IF;
374
375 -- Flag Zero -----
376
377 -- Flag Overflow -----
378 -- Esta flag sinaliza quando nossa entrada não consegue ser representada corretamente no nosso valor de saída.
379 -- Para a soma de 4 bits a soma acontece de forma frequente, o que se ocorre o carry-out, então podemos
380 -- utilizar nosso quinto bit para identificar igualmente o doar.
381
382 IF soma(4) = '1' THEN -- Adição
383     flag_ovout(0) <= '1';
384 ELSE
385     flag_ovout(0) <= '0';
386 END IF;
387
388 -- Flag Overflow -----
389
390 -- Flag Carry -----
391 -- Esta flag sinaliza quando o resultado da soma ultrapassar o valor máximo representável por 4 bits, ou seja, 15.
392 -- Para isso, vamos utilizar o bit 4 do resultado da soma para identificar o carry-out.
393
394 IF soma(4) = '1' THEN
395     flag_cout(0) <= '1';
396 ELSE
397     flag_cout(0) <= '0';
398 END IF;
399
400 -- Flag Carry -----
401
402 -- Flag Sign -----
403 -- Esta flag sinaliza quando o resultado da soma for negativo, ou seja, quando o bit de sinal (bit 3) for 1.
404 -- Para isso, vamos utilizar o bit 3 do resultado da soma para identificar o sinal.
405
406 IF soma(3) = '1' THEN
407     flag_sout(0) <= '1';
408 ELSE
409     flag_sout(0) <= '0';
410 END IF;
411
412 -- Flag Sign -----
413
414 -- Flag Paridade -----
415 -- Esta flag sinaliza quando o resultado da soma for par ou ímpar.
416 -- Para isso, vamos utilizar o bit 0 do resultado da soma para identificar a paridade.
417
418 IF soma(0) = '1' THEN
419     flag_pout(0) <= '1';
420 ELSE
421     flag_pout(0) <= '0';
422 END IF;
423
424 -- Flag Paridade -----
425
426 -- Flag Zero -----
427 -- Se nosso resultado for "0000" nosso flag zero sinaliza.
428 IF soma = "000000" THEN
429     flag_zout(0) <= '1';
430 ELSE
431     flag_zout(0) <= '0';
432 END IF;
433
434 -- Flag Zero -----
435
436 -- Flag Overflow -----
437 -- Esta flag sinaliza quando nossa entrada não consegue ser representada corretamente no nosso valor de saída.
438 -- Para a soma de 4 bits a soma acontece de forma frequente, o que se ocorre o carry-out, então podemos
439 -- utilizar nosso quinto bit para identificar igualmente o doar.
440
441 IF soma(4) = '1' THEN -- Adição
442     flag_ovout(0) <= '1';
443 ELSE
444     flag_ovout(0) <= '0';
```

- As flags das operações lógicas são bit/flag : (2 e 3 em '0')NENHUMA PORTAS LÓGICAS ATIVA ; (2 em '1')PELO MENOS UMA PORTAS LÓGICAS ATIVA ; (3 em '1')NUMERO IGUAL DE PORTAS LÓGICAS ATIVAS : (2 e 3 em '1')TODAS ATIVAS.



O esquemático RTL e sua simulação podem ser visualizados nas imagens a seguir :

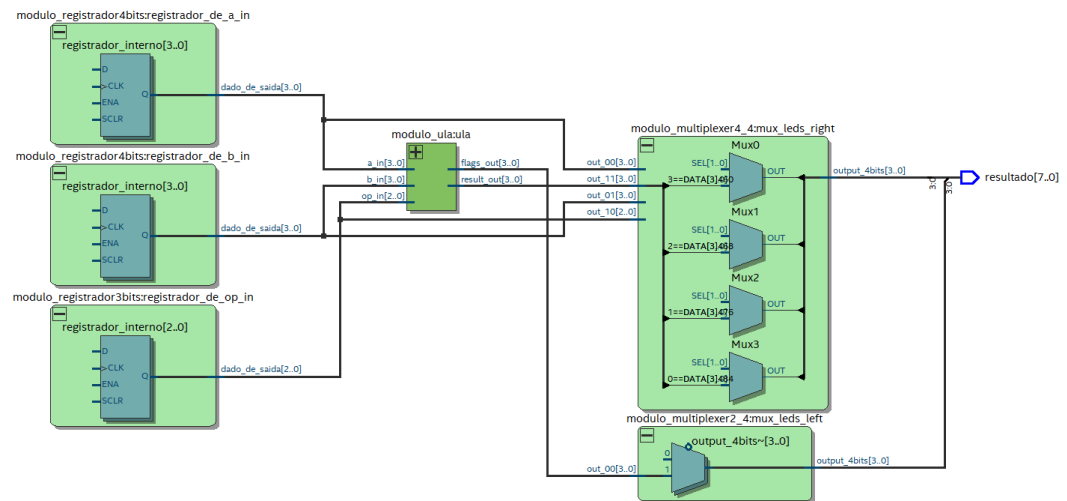




Obs:

As simulações dos módulos somador, subtrator e funções lógicas constam dentro da simulação da ULA, portanto decidimos simplificar apenas mostrando o da ULA.

Ao fim teremos esta configuração entre a bancada de testes e a ULA, a seguir existe a relação entre fonte dos dados e saída dos mesmos.



### Bancada de testes :

A bancada de testes foi feita exatamente como a ULA, declaração de componentes, e após feita configuração dos sinais e mapeamento apenas tivemos o trabalho de delegar quais bits da saída Resultado de 8 bits seria alimentado. A configuração final ficou da seguinte forma :

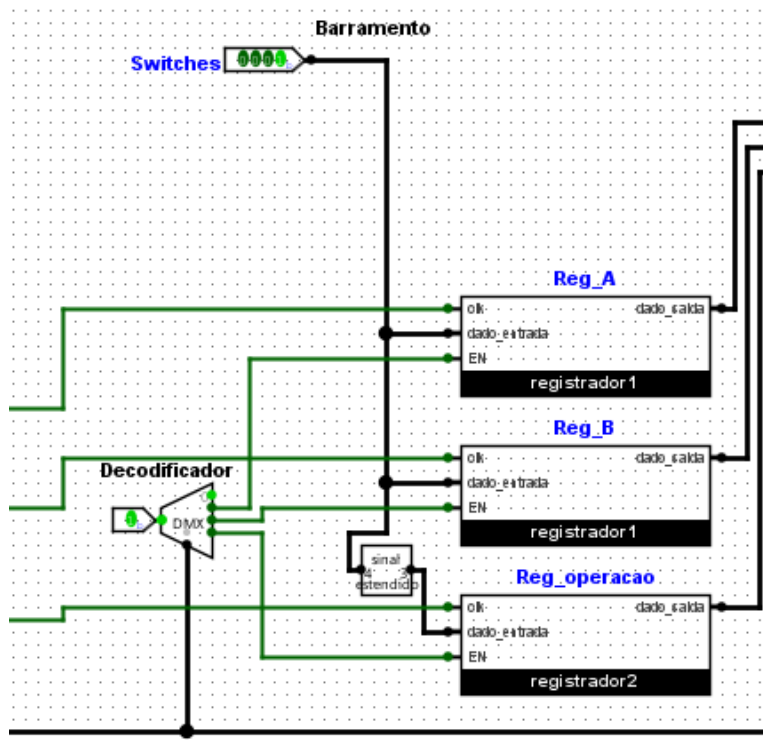
- 4 Switches da FPGA é representado por uma entrada vetorial de 4



bits e indica o valor das nossas entradas em geral.

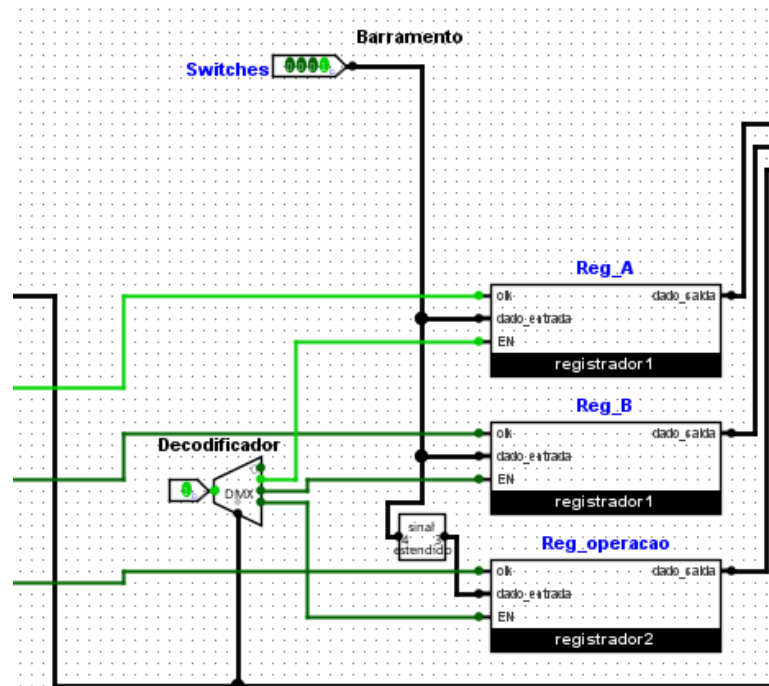
- Botões da FPGA agem como clock em nosso circuito, de modo que ao pressionar o botão correspondente do registrador de A ou do registrador de B, registrador da Operação, indicam respectivamente os clocks do Registrador A, Registrador B e Registrador de Operação, se o Enable correspondente estiver ativo, a entrada é registrada, caso contrário, se Enable estiver em nível baixo, nenhum o registrador recebe o dado, mesmo que pressione o botão clock do registrador correspondente. E o nosso botão seletor de Enables é responsável por alternar a alimentação dos enables dos registradores de A, B e Operação, configurando assim nosso Barramento de informação.

No barramento temos 4 registradores e só podemos enviar informação para 1 enable de cada vez, logo, como mostra a imagem, nenhum dos registradores está com seu respectivo enable em nível alto:



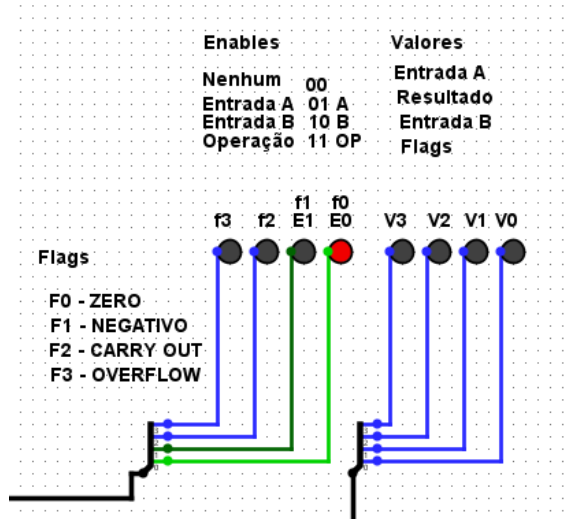
Entretanto, ao pressionar o botão seletor de Enable, nosso decodifi-

cador seleciona a próxima saída através do contador, sendo assim, com nosso decodificador com (01) em seu seletor, encontramos o Enable do registrador de A em nível alto como na imagem a seguir, o que nos torna capaz de selecionar uma entrada qualquer nos Switches e enviar a informação para o registrador de A através do botão correspondente ao clock do registrador de A:

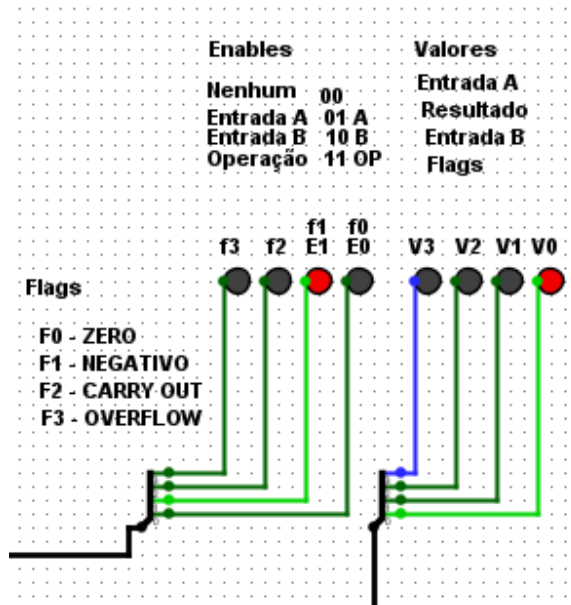


- Os 8 LEDs da FPGA é representado por uma entrada vetorial de 8 bits e indica o valor das nossas saídas e flags do resultado. Os 4 leds mais à **esquerda** indicam : Flags e o enable atual habilitado, sendo que os flags utilizam todos os 4 leds e os que indicam os enables apenas 2 leds ( contando da direita para a esquerda). Os 4 leds mais à **direita** indicam : Entrada A, Entrada B, Operação e Resultado, respeitam esta sequência, com a observação de que a operação utiliza apenas 3/4 leds.

Como vimos no exemplo anterior, alternamos de **nenhum enable habilitado**, para o **enable do registrador de A habilitado** portanto nosso circuito terá a seguinte configuração:

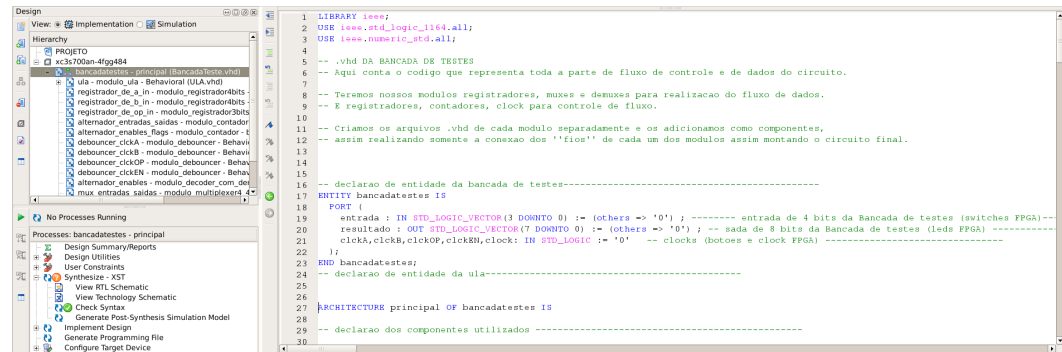


Após 1 clock podemos visualizar as FLAGS. Neste exemplo podemos ver a flag correspondente do resultado negativo : 0001 – 0010, portanto teremos :



- Clock é utilizado tanto como indicador para o debounce, como também para a alternância junto aos contadores e muxes, considerando o clock da placa e o dividido, permitindo que alterne sequencialmente.

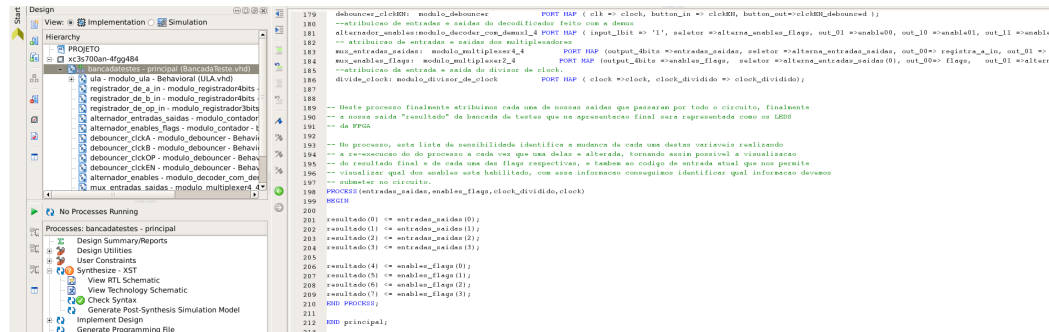
O código VHDL da Bancada de testes:



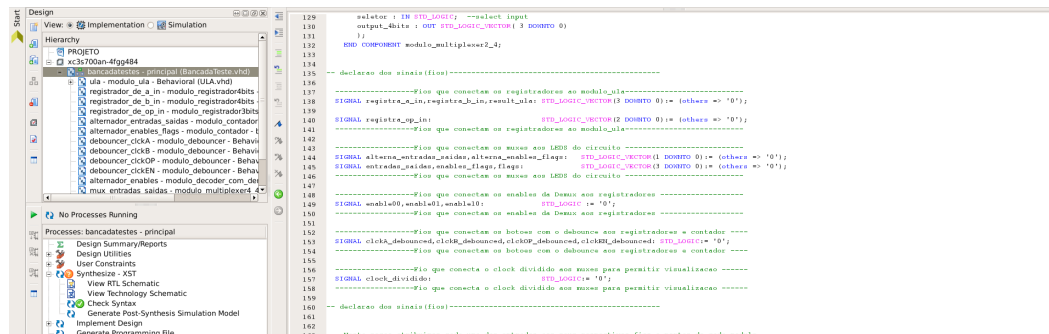
## O código do mapeamento da Bancada de testes:



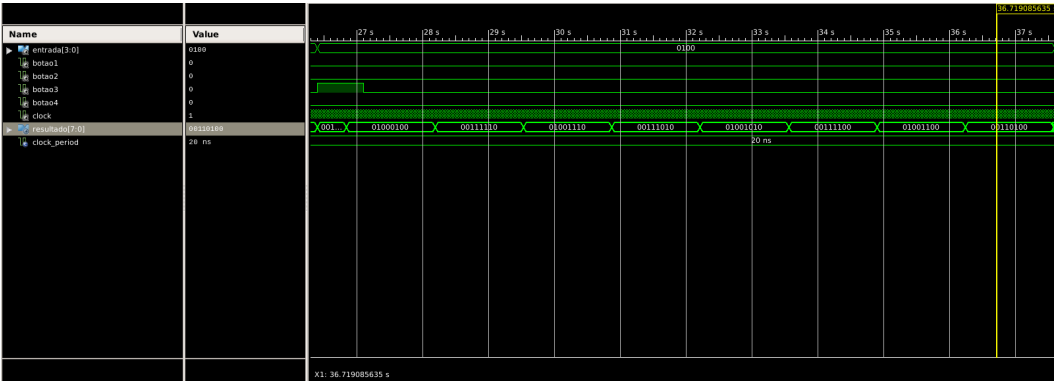
## O código VHDL da arquitetura da bancada de testes:



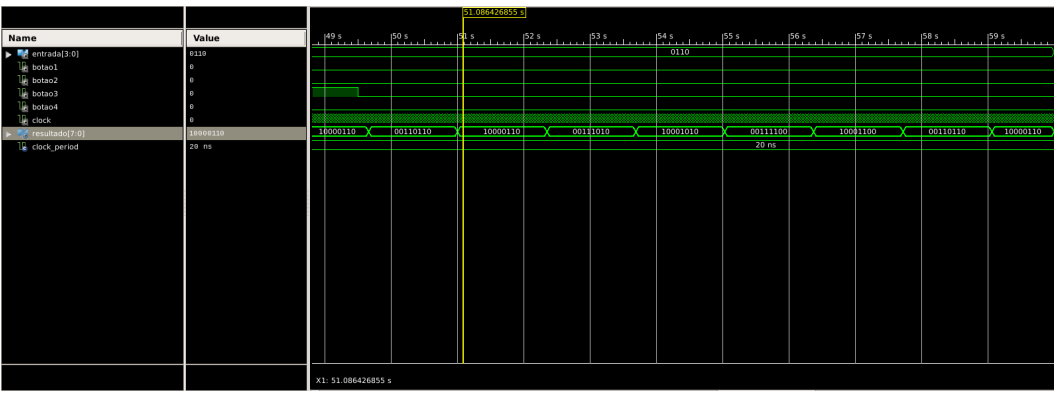
## O código dos sinais(fios) da bancada de teste:







Porta XOR:



Obs : Como nosso clock dividido possui um tempo de 1,34s precisamos de 4 ciclos para a alternagem total dos leds mais à direita (Entrada A, Entrada B, Entrada Operação, Resultado) enquanto à esquerda em apenas 2 ciclos conseguimos identificar o Enable atual ativo e as Flags.

xxx

*DESENVOLVIMENTO*



# Conclusão

## 0.7 Observações:

- No início não estava muito claro que deveríamos utilizar a ULA como módulo e não como a bancada de testes, no andamento do trabalho, vimos que ao usarmos a ULA como módulo, além de organizar o projeto, existe uma permissibilidade de reutilizar o módulo ULA, e isto era exatamente o que buscávamos.
- Organizar o projeto em módulos separados possibilitou a fácil manutenção e visualização em nível alto.
- Fez-se claro inteligível como o PORT MAP, SIGNAL, COMPONENTS, etc funcionam na linguagem VHDL, o que facilitou a compreensão do projeto ao visualizarmos ele como um mapeamento de fios que conectam as entradas de cada módulo do circuito entre si.



# Referências Bibliográficas

## 0.8 Como usar o VHDL:

- Vhdlguru : <https://vhdlguru.blogspot.com/p/example-codes.html>;
- Introdução à linguagem VHDL: <https://www.gta.ufrj.br/ensino/EEL480/Introducao-VHDL.pdf>;
- UFSJ : <https://ufsj.edu.br/portal2-repositorio/File/nepomuceno/fpga-vhdl/vhdl-altera.pdf>;

## 0.9 Planejamento do Circuito:

- Debounce: <https://www.youtube.com/watch?v=8ISfNm9zv18>;
- Divisor de Clock: <https://www.youtube.com/watch?v=XyIkr8OkDYU&list=WL&index=1>;
- Quartus Prime : <https://www.intel.com.br/content/www/br/pt/products/details/fpga/development-tools/quartus-prime.html>;
- Logisim : <https://github.com/logisim-evolution/logisim-evolution>;