

## Atividade do Capítulo 2

Nome: Mateus Sousa Araújo – Matrícula: 374858

Nome: José Wesley Araújo – Matrícula: 374855

### # HALF ADDER (Meio Somador)

```
1  --      HALF ADDER
2
3  --      INPUTS |   OUTPUTS   |
4  -----
5  -- a | b | carry | sum |
6  -----
7  -- 0 | 0 |  0 |  0 |
8  -- 0 | 1 |  0 |  1 |
9  -- 1 | 0 |  0 |  1 |
10 -- 1 | 1 |  1 |  0 |
11
12
13 library IEEE;
14 use IEEE.STD_LOGIC_1164.ALL;
15 use IEEE.NUMERIC_STD.ALL;
16
17 entity half_adder is
18     Port (
19         a,b      : in  std_logic; -- entrada a e b;
20         carry     : out std_logic; -- carry "vai um";
21         sum       : out std_logic -- resultado da soma;
22     );
23 end half_adder;
24
25 architecture Behavioral of half_adder is
26
27     begin
28
29         sum <= a xor b; -- soma só recebe verdadeiro se e somente se um dos dois for verdade;
30         carry <= a and b; -- carry recebe uma and entre a e b;
31
32     end Behavioral;
33
```

Para a realização do meio somador criamos duas entradas “a” e “b” e duas saídas “carry” e “sum”. A variável “sum” guarda o valor da soma, e o “carry” guarda o valor de “vai um”. De acordo com a tabela verdade do meio somador, fizemos uma “xor” entre “a” e “b” e o carry recebeu uma “and” entre “a” e “b”. Dessa forma fica correto a lógica para essa implementação.

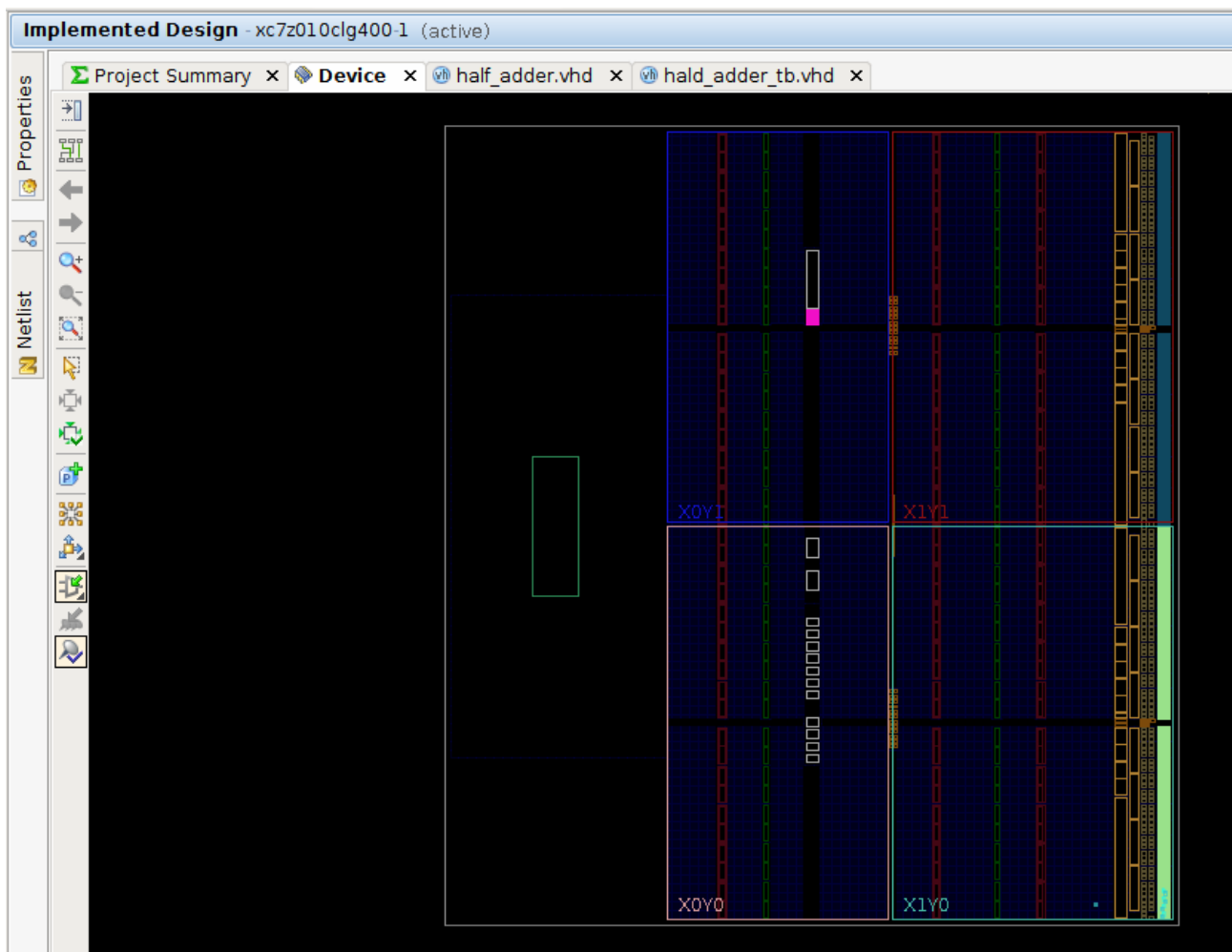
Para o Teste Bench realizamos a criação dos sinais de entrada e saída. Logo em seguida fizemos o mapeamento. Dentro do processo fizemos todos os casos em que as entradas poderiam ficar de acordo com a tabela verdade. Fizemos isso utilizando 2 sinais criados, cada qual representando as 2 entradas “a” e “b”. Ocorreu um pequeno atraso de propagação na simulação. Abaixo segue o código do teste bench e a simulação waveform. Fizemos também a implementação de Design do programa. Também fizemos as outras simulações.

```

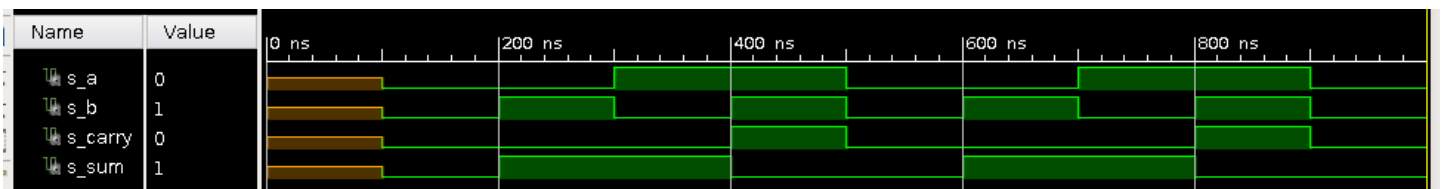
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.NUMERIC_STD.ALL;
4
5  entity hald_adder_tb is
6  -- Port ( );
7  end hald_adder_tb;
8
9  architecture Behavioral of hald_adder_tb is
10
11  component half_adder is
12  Port (
13      a,b      : in    std_logic; -- entrada a e b;
14      carry    : out   std_logic; -- carry "vai um";
15      sum      : out   std_logic  -- resultado da soma;
16  );
17  end component;
18
19  -- INPUTS
20  signal s_a      :    std_logic; -- sinal para a entrada a;
21  signal s_b      :    std_logic; -- sinal para a entrada b;
22
23  -- OUTPUTS
24  signal s_carry   :    std_logic; -- sinal para a saída de carry;
25  signal s_sum     :    std_logic; -- sinal para a saída de soma;
26
27  begin
28
29  teste: half_adder port map( a => s_a, b => s_b, carry => s_carry, sum => s_sum ); -- mapeamento;
30
31  -- Stimulus process
32  stim_proc: process
33  begin
34      -- hold reset state for 100 ns.
35      wait for 100 ns;
36
37      s_a <= '0';    s_b <= '0'; -- caso quando "a" e "b" são 00;
38
39      wait for 100 ns;
40
41      s_a <= '0';    s_b <= '1'; -- caso quando "a" e "b" são 01;
42
43      wait for 100 ns;
44
45      s_a <= '1';    s_b <= '0'; -- caso quando "a" e "b" são 10;
46
47      wait for 100 ns;
48
49      s_a <= '1';    s_b <= '1'; -- caso quando "a" e "b" são 11
50
51      end process stim_proc;
52
53  end Behavioral;
54

```

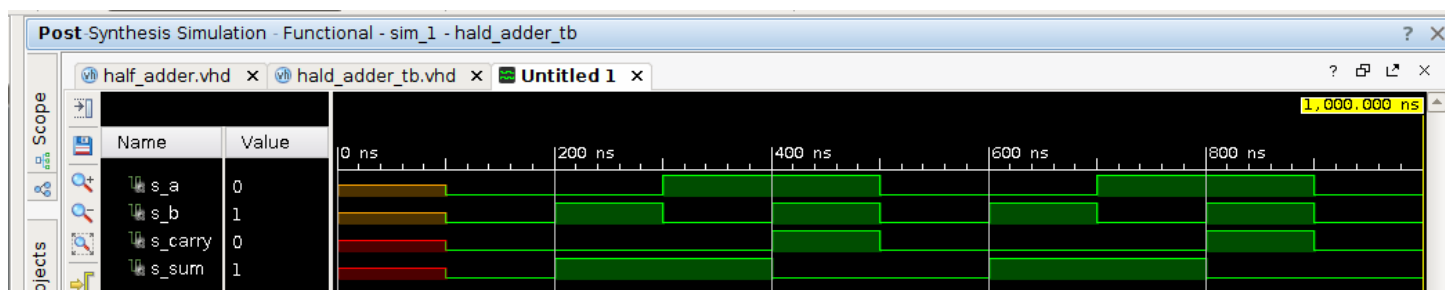
## Teste bench



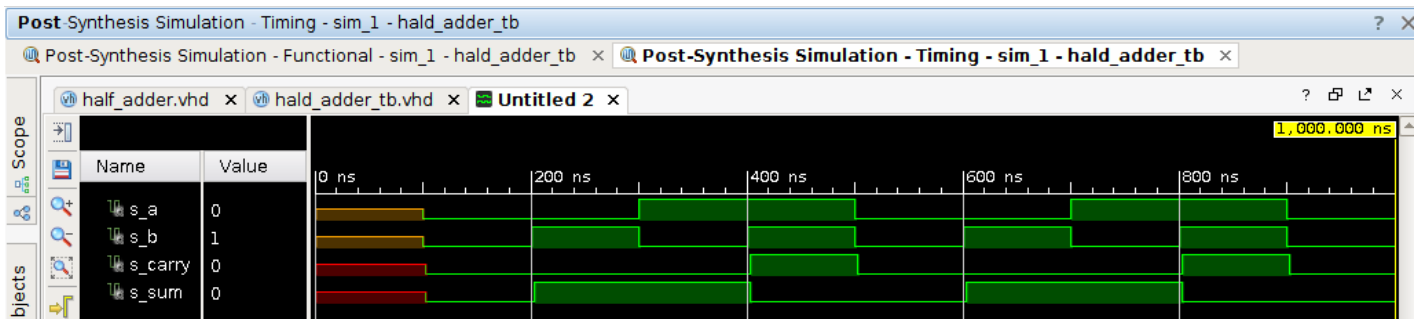
Implementation Design



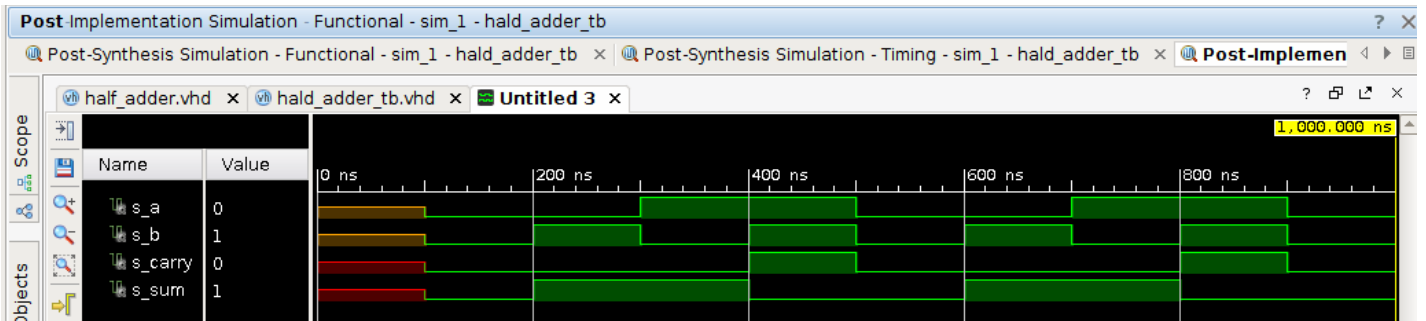
Simulação de comportamento (Behavioral)



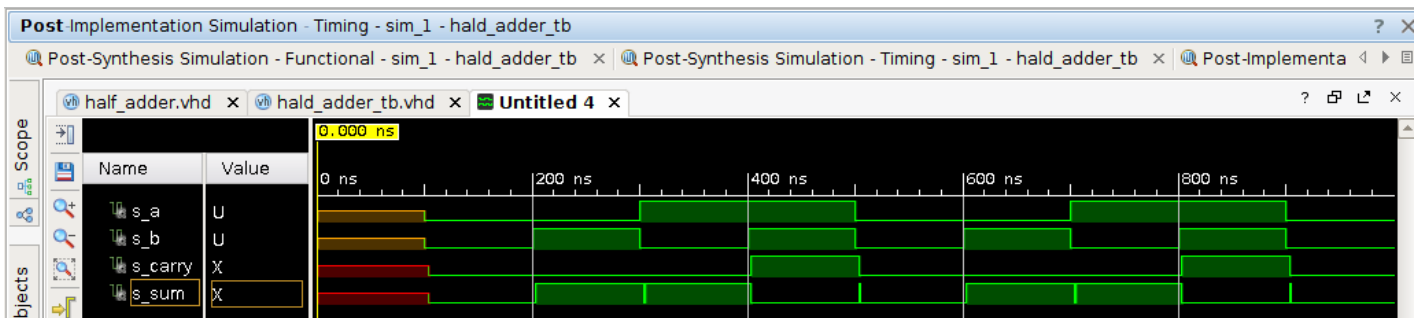
Synthesis Functional Simulation



Synthesis Timing Simulation



Implementation Functional Simulation



Implementation Timing Simulation

## # FULL ADDER (Somador Completo)

```
1  --
2  --          SOMADOR COMPLETO - FULL ADDER
3  --
4  --          a    b    cin  |    sum    |    carry    |
5  --          -----
6  --          0    0    0   |    0       |    0         |
7  --          0    0    1   |    0       |    1         |
8  --          0    1    0   |    0       |    1         |
9  --          0    1    1   |    1       |    0         |
10 --          1    0    0   |    0       |    1         |
11 --          1    0    1   |    1       |    0         |
12 --          1    1    0   |    1       |    0         |
13 --          1    1    1   |    1       |    1         |
14 --          -----
15
16 library IEEE;
17 use IEEE.STD_LOGIC_1164.ALL;
18 use IEEE.NUMERIC_STD.ALL;
19
20 entity full_adder is
21     Port (
22         a, b, cin    : in std_logic; -- entradas a, b e o "vai um - cin";
23         sum, carry    : out std_logic -- saídas da soma e do carry;
24     );
25 end full_adder;
26
27 architecture Behavioral of full_adder is
28
29     begin
30
31     sum <= a xor b xor cin; -- operação para a saída da soma;
32     carry <= (a and b) or (a and cin) or (b and cin); -- operação para a saída de carry;
33
34 end Behavioral;
35
```

Para a realização do Somador completo, foi preciso considerar agora 3 entradas de input, ao invés de apenas 2, como era no meio somador. A ideia aqui é levar em consideração uma nova entrada que é o “carry in”. Esse tipo de somador resolve o problema do meio somador por levar em consideração o “cin” do anterior. Temos 3 entradas com 2 saídas. As 2 saídas são representadas por uma de soma e outra de carry.

De acordo com a tabela verdade do somador completo, é preciso fazer algumas operações para a soma “sum” e o carry. Para o saída “sum”, usamos 2 portas do tipo xor com as entradas “a” e “b” juntamente com o carry in “cin”. Para o carry foi preciso implementar 3 and’s pra cada par de entrada, “a com b”, “a com cin” e “b com cin”. Dessa forma garantimos a saída de acordo com a tabela verdade.

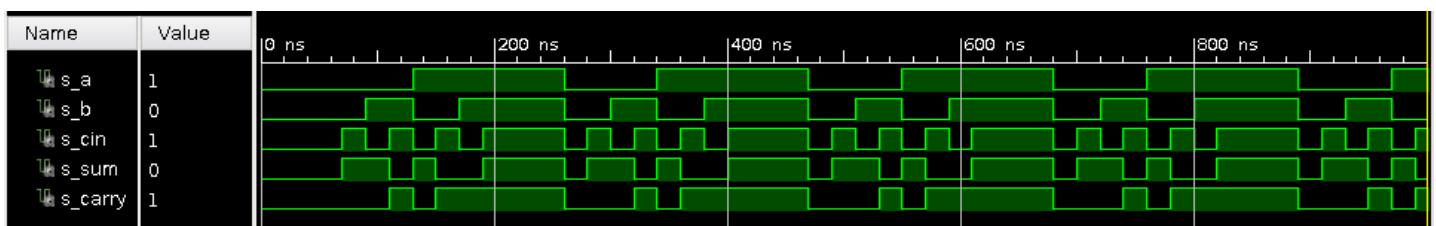
O Teste bench foi similar ao meio somador, com a única diferença de um sinal a mais de entrada referente ao carry de entrada “cin”. Logo em seguida fizemos a criação de 3 sinais de entrada e 2 de saída para fazer o mapeamento. No processo fizemos todos os casos que as 3 entradas de input podiam assumir. Abaixo se encontra o Teste bench e as simulações.

```

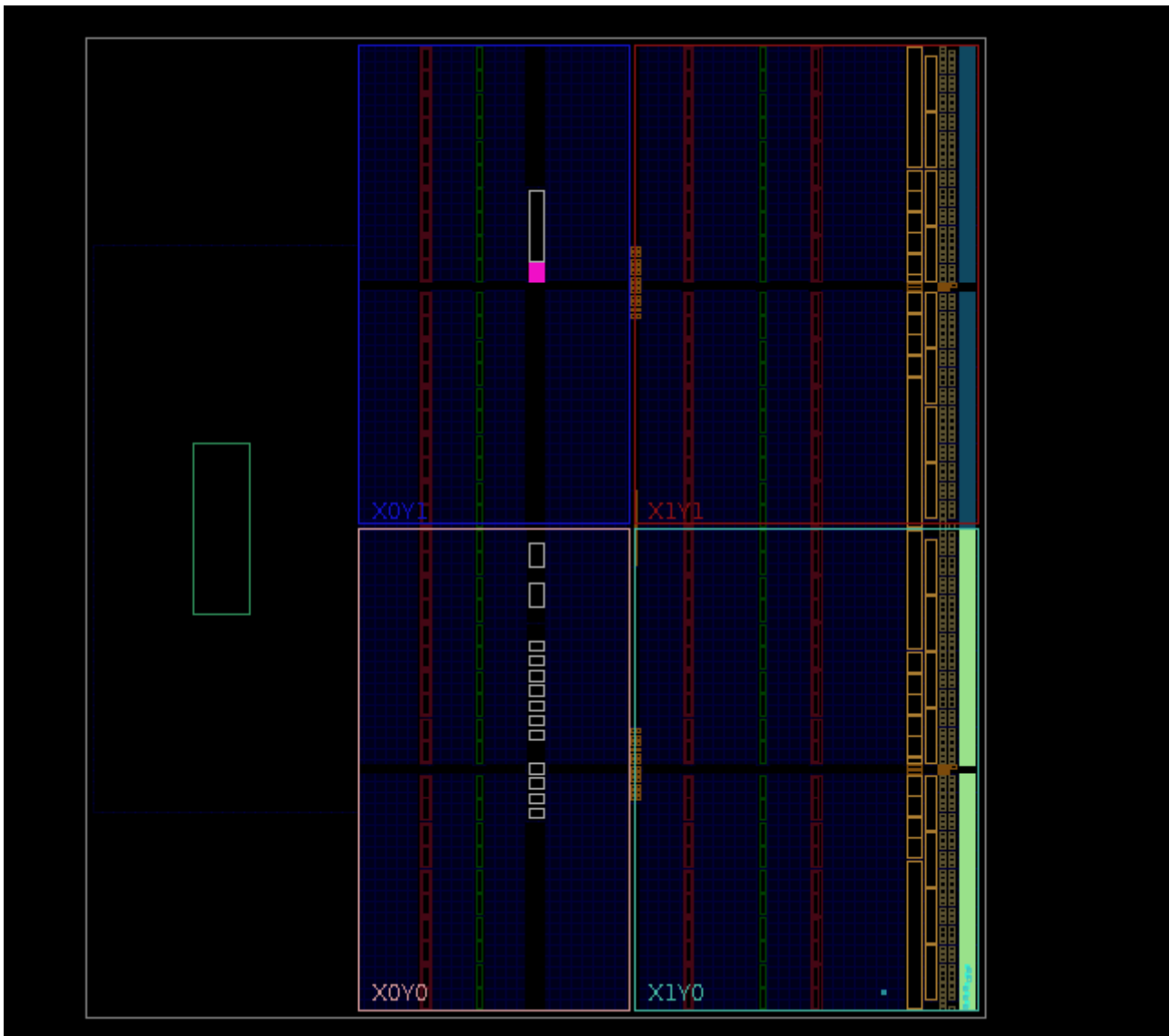
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use IEEE.NUMERIC_STD.ALL;
4
5 entity full_adder_tb is
6 -- Port ( );
7 end full_adder_tb;
8
9 Architecture Behavioral of full_adder_tb is
10     component full_adder is
11         Port(
12             a, b, cin    : in std_logic; -- entradas a, b e o "vai um - cin";
13             sum, carry    : out std_logic -- saídas da soma e do carry;
14         );
15     end component;
16
17 --Inputs
18 signal s_a      : std_logic := '0'; -- sinal da entrada a;
19 signal s_b      : std_logic := '0'; -- sinal da entrada b;
20 signal s_cin     : std_logic := '0'; -- sinal da entrada cin;
21
22 --Outputs
23 signal s_sum     : std_logic; -- sinal de saída para a soma;
24 signal s_carry   : std_logic; -- sinal de saída para o carry;
25
26 begin
27
28 teste: full_adder port map(a => s_a, b => s_b, cin => s_cin, sum => s_sum, carry => s_carry); -- mapeamento
29
30 -- Stimulus process
31 stim_proc: process
32 begin
33
34     wait for 50 ns;
35
36     s_a <= '0';    s_b <= '0';    s_cin <= '0'; -- caso "000"
37     wait for 20 ns;
38
39     s_a <= '0';    s_b <= '0';    s_cin <= '1'; -- caso "001"
40     wait for 20 ns;
41
42     s_a <= '0';    s_b <= '1';    s_cin <= '0'; -- caso "010"
43     wait for 20 ns;
44
45     s_a <= '0';    s_b <= '1';    s_cin <= '1'; -- caso "011"
46     wait for 20 ns;
47
48     s_a <= '1';    s_b <= '0';    s_cin <= '0'; -- caso "100"
49     wait for 20 ns;
50
51     s_a <= '1';    s_b <= '0';    s_cin <= '1'; -- caso "101"
52     wait for 20 ns;
53
54     s_a <= '1';    s_b <= '1';    s_cin <= '0'; -- caso "110"
55     wait for 20 ns;
56
57     s_a <= '1';    s_b <= '1';    s_cin <= '1'; -- caso "111"
58     wait for 20 ns;
59
60 end process stim_proc;
61
62 end Behavioral;
63

```

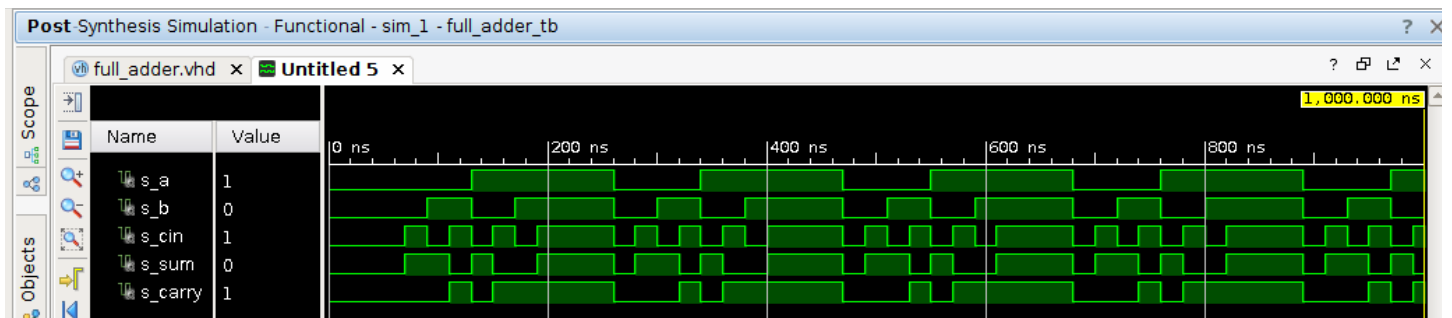
## Teste bench



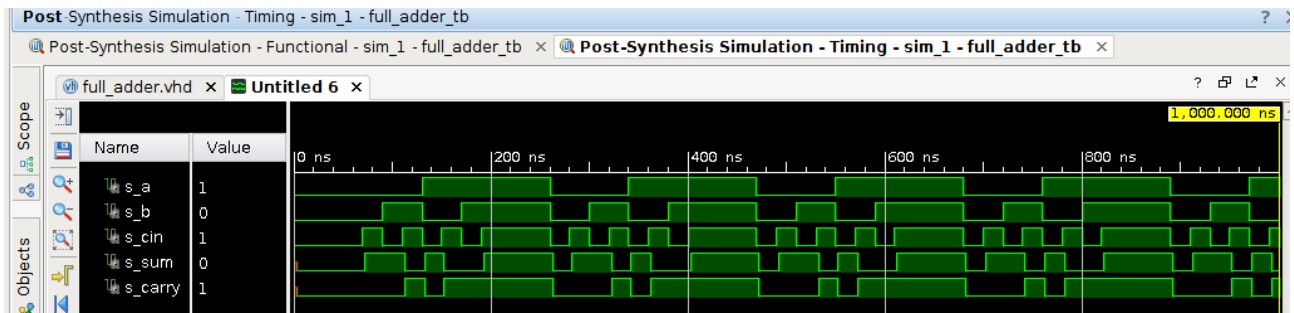
## Simulação de comportamento (Behavioral)



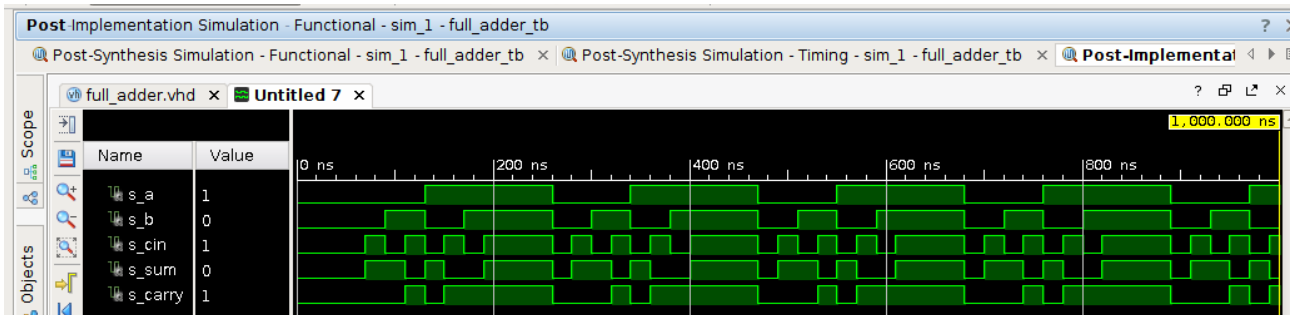
Implementation Design



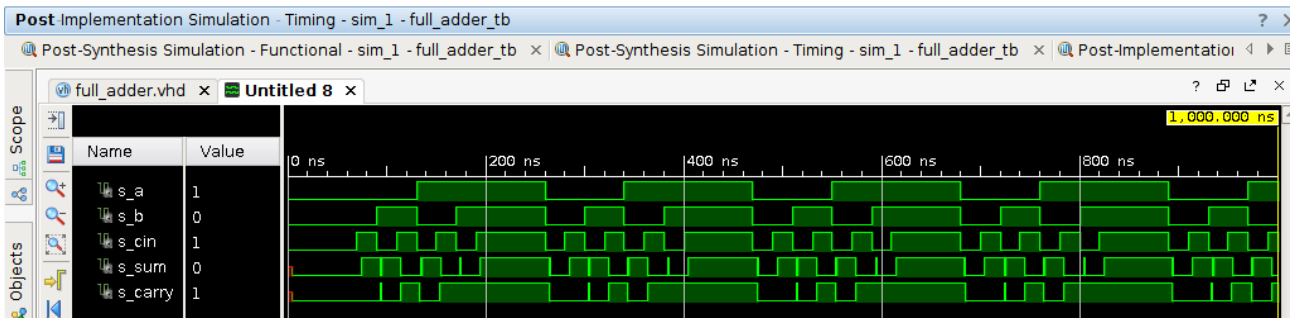
Synthesis Functional Simulation



## Synthesis Timing Simulation



## Implementation Functional Simulation



## Implementation Timing Simulation



## # 16-BIT ADDER (Somador de 16 bits)

```
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 entity bits_adder is
7     Port (
8         num1 :    in std_logic_vector(15 downto 0); -- entrada a de 16 bits;
9         num2 :    in std_logic_vector(15 downto 0); -- entrada b de 16 bits;
10        sum :      out std_logic_vector(15 downto 0); -- saída de soma com 16 bits;
11        carry :    out std_logic                    -- saída de carry out.
12    );
13 end bits_adder;
14
15 architecture Behavioral of bits_adder is
16
17     --temporary signal declarations(for intermediate carry's).
18     signal c0,c1,c2,c3, c4, c5, c6, c7, c8, c9, c10, c11, c12, c13, c14, c15 : std_logic := '0'; -- sinais de carry
19
20     begin
21
22         -- primeiro somador
23         sum(0) <= num1(0) xor num2(0); -- calculo da soma;
24         c0 <= num1(0) and num2(0); -- calculo do carry;
25
26         -- segundo somador
27         sum(1) <= num1(1) xor num2(1) xor c0;
28         c1 <= (num1(1) and num2(1)) or (num1(1) and c0) or (num2(1) and c0);
29
30         -- terceiro somador
31         sum(2) <= num1(2) xor num2(2) xor c1;
32         c2 <= (num1(2) and num2(2)) or (num1(2) and c1) or (num2(2) and c1);
33
34         -- quarto somador
35         sum(3) <= num1(3) xor num2(3) xor c2;
36         c3 <= (num1(3) and num2(3)) or (num1(3) and c2) or (num2(3) and c2);
37
38         -- quinto somador
39         sum(4) <= num1(4) xor num2(4) xor c3;
40         c4 <= (num1(4) and num2(4)) or (num1(4) and c3) or (num2(4) and c3);
41
42         -- sexto somador
43         sum(5) <= num1(5) xor num2(5) xor c4;
44         c5 <= (num1(5) and num2(5)) or (num1(5) and c4) or (num2(5) and c4);
```

Daria pra fazer o somador de 16 bits de uma forma mais fácil utilizando apenas a soma de 2 vetores de qualquer tamanho com a operação “saída <= vetor1[ ] + vetor2[ ]”. O código daria apenas uma linha, mas primeiramente tínhamos feito de forma similar com o somador completo, na qual levava em consideração o carry de entrada (“cin”). Para a criação desse modelo fizemos duas entradas “num1” e “num2” de 16 bits cada. Também fizemos a criação de uma saída que levará a soma, chamada “sum”, de 16 bits também. Criamos outra saída que nos indicará um estouro final no nosso barramento de 16 bits. O livro sugere que não seja levado em consideração o carry final, mas colocamos para ter um efeito mais interessante na soma final das duas entradas.

A ideia básica é fazer um somador bit a bit de 16 bits, levando em consideração o carry de entrada. Para isso criamos 15 sinais intermediários que vão representar os carry’s intermediários. Para isso, nos utilizamos da mesma lógica do somador completo e fizemos a operação bit a bit do barramento de 16 bits. No final colocamos carry para receber o estouro. Se nossa soma ultrapassar os 16 bits, então será representação somente os 16 primeiros em hexadecimal. A figura acima representa uma parte do código em VHDL, e a outra logo abaixo a segunda parte.

```

45
46 -- sétimo somador
47 sum(6) <= num1(6) xor num2(6) xor c5;
48 c6 <= (num1(6) and num2(6)) or (num1(6) and c5) or (num2(6) and c5);
49
50 -- oitavo somador
51 sum(7) <= num1(7) xor num2(7) xor c6;
52 c7 <= (num1(7) and num2(7)) or (num1(7) and c6) or (num2(7) and c6);
53
54 -- nono somador
55 sum(8) <= num1(8) xor num2(8) xor c7;
56 c8 <= (num1(8) and num2(8)) or (num1(8) and c7) or (num2(8) and c7);
57
58 -- décimo somador
59 sum(9) <= num1(9) xor num2(9) xor c8;
60 c9 <= (num1(9) and num2(9)) or (num1(9) and c8) or (num2(9) and c8);
61
62 -- décimo primeiro somador
63 sum(10) <= num1(10) xor num2(10) xor c9;
64 c10 <= (num1(10) and num2(10)) or (num1(10) and c9) or (num2(10) and c9);
65
66 -- décimo segundo somador
67 sum(11) <= num1(11) xor num2(11) xor c10;
68 c11 <= (num1(11) and num2(11)) or (num1(11) and c10) or (num2(11) and c10);
69
70 -- décimo terceiro somador
71 sum(12) <= num1(12) xor num2(12) xor c11;
72 c12 <= (num1(12) and num2(12)) or (num1(12) and c11) or (num2(12) and c11);
73
74 -- décimo quarto somador
75 sum(13) <= num1(13) xor num2(13) xor c12;
76 c13 <= (num1(13) and num2(13)) or (num1(13) and c12) or (num2(13) and c12);
77
78 -- décimo quinto somador
79 sum(14) <= num1(14) xor num2(14) xor c13;
80 c14 <= (num1(14) and num2(14)) or (num1(14) and c13) or (num2(14) and c13);
81
82 -- décimo sexto somador
83 sum(15) <= num1(15) xor num2(15) xor c14;
84 c15 <= (num1(15) and num2(15)) or (num1(15) and c14) or (num2(15) and c14);
85
86 carry <= c15;
87
88 end Behavioral;

```

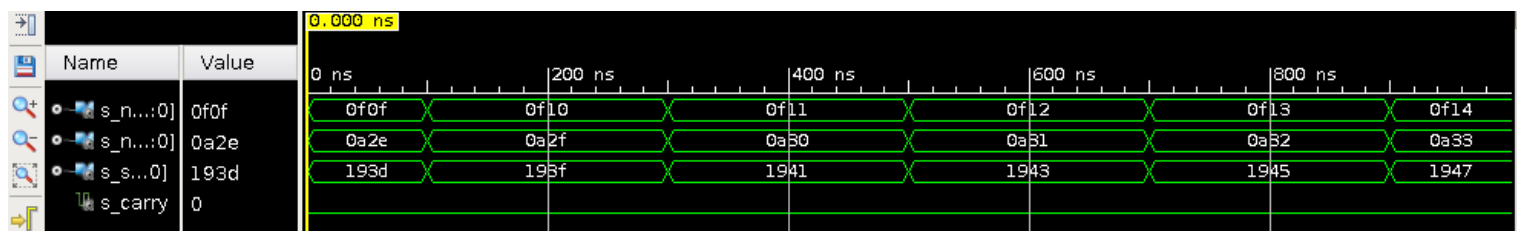
Para o Teste bench realizamos a criação de 4 sinais, cada um referente as entradas e saídas da entidade. Aqui fizemos que as entradas “num1” e “num2” recebessem valores fixos, e que dentro do processo houvesse a incrementação desses valores. Fixamos “num1” em “0F0F” e “num2” em “0A2E”. Dentro do processo esses valores serão incrementados e a soma vai sendo realizada bit a bit no barramento de 16 bits. Tivemos o cuidado para fazer um casting na incrementação de cada vetor de 16 bits para números com sinal também. Abaixo está o Teste bench e as simulações. Nesse caso na simulação waveform, não há overflow. Logo o carry ficará sempre em 0.

```

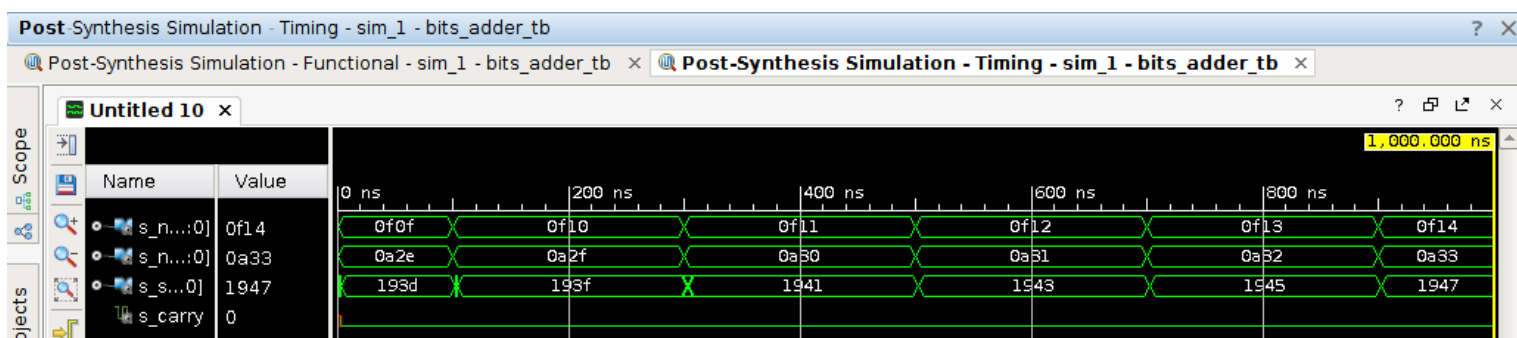
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.NUMERIC_STD.ALL;
4  use IEEE.STD_LOGIC_UNSIGNED.ALL;
5
6  entity bits_adder_tb is
7  -- Port ( );
8  end bits_adder_tb;
9
10 architecture Behavioral of bits_adder_tb is
11     component bits_adder is
12     Port(
13         num1 :    in std_logic_vector(15 downto 0);    -- entrada a de 16 bits;
14         num2 :    in std_logic_vector(15 downto 0);    -- entrada b de 16 bits;
15         sum  :    out std_logic_vector(15 downto 0);   -- saída de soma com 16 bits;
16         carry :    out std_logic                      -- saída de carry out;
17     );
18     end component;
19
20 signal s_num1 :    std_logic_vector(15 downto 0) := X"0F0F"; -- sinal para a entrada 1;
21 signal s_num2 :    std_logic_vector(15 downto 0) := X"0A2E"; -- sinal para a entrada 2;
22 signal s_sum  :    std_logic_vector(15 downto 0);   -- sinal para a saída de soma;
23 signal s_carry :    std_logic;                      -- sinal a saída de carry;
24
25 begin
26
27 UUT : bits_adder port map(num1 => s_num1, num2 => s_num2 ,sum => s_sum, carry => s_carry); -- mapeamento;
28
29 tb : process
30
31 begin
32
33 wait for 100 ns;
34 s_num1 <= std_logic_vector(signed(s_num1) + 1);
35 s_num2 <= std_logic_vector(signed(s_num2) + 1);
36 wait for 100 ns;
37
38 end process tb;
39
40 end Behavioral;

```

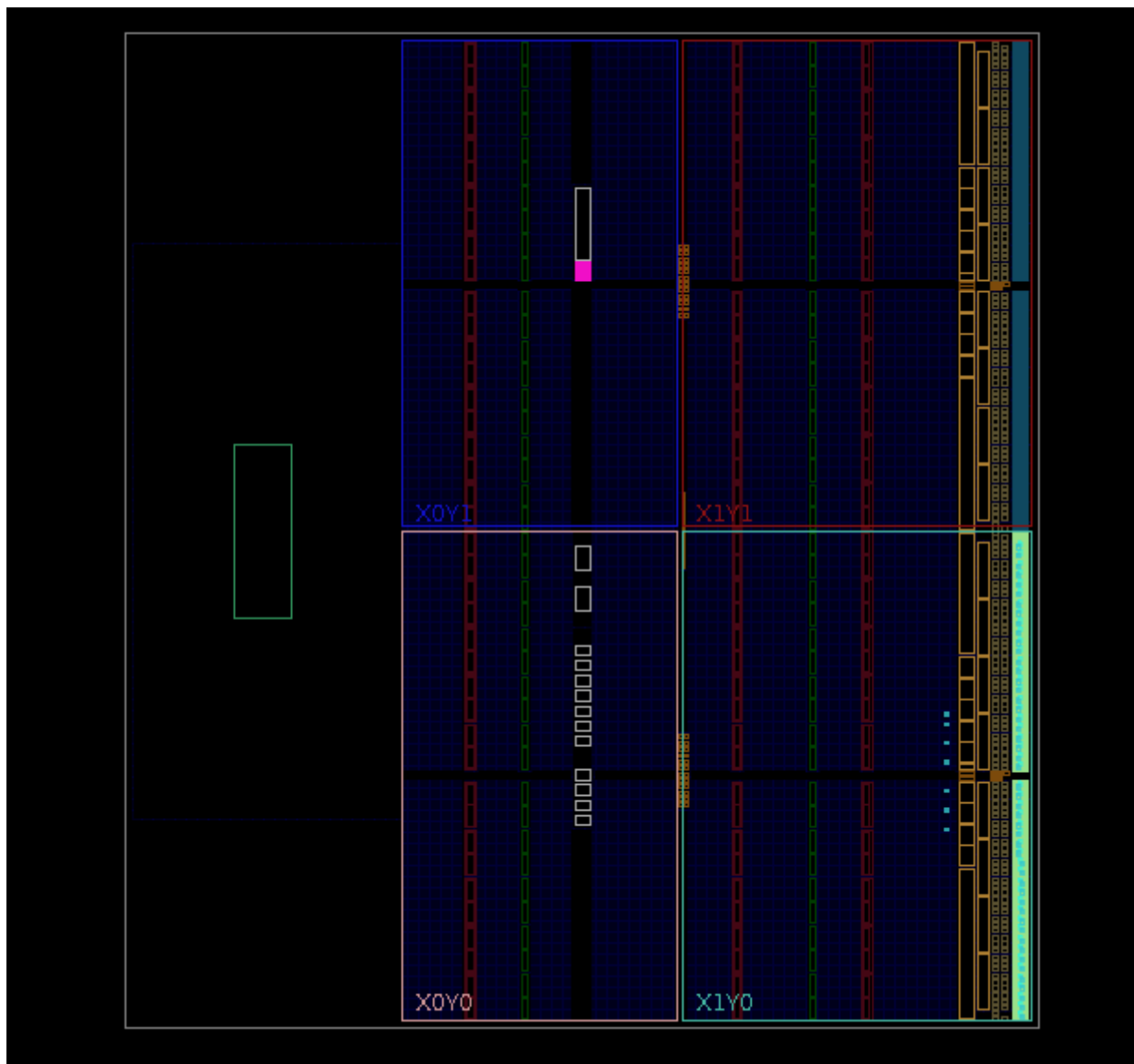
## Teste bench



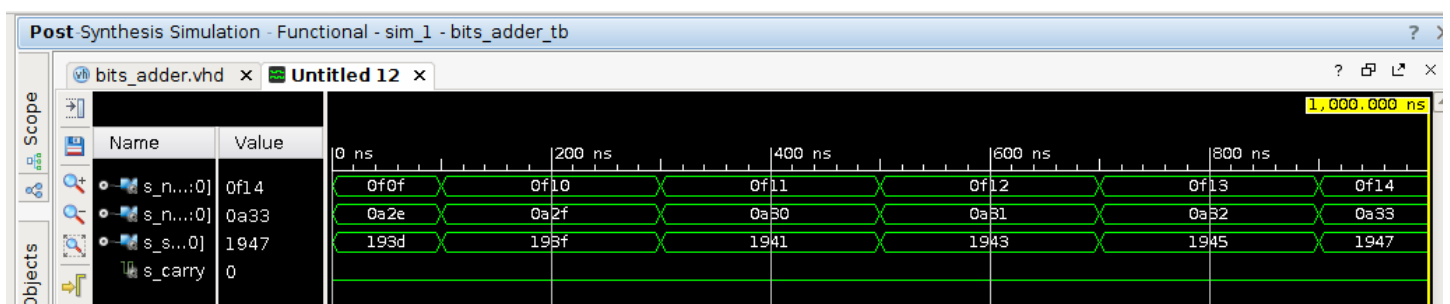
## Simulação de comportamento (Behavioral)



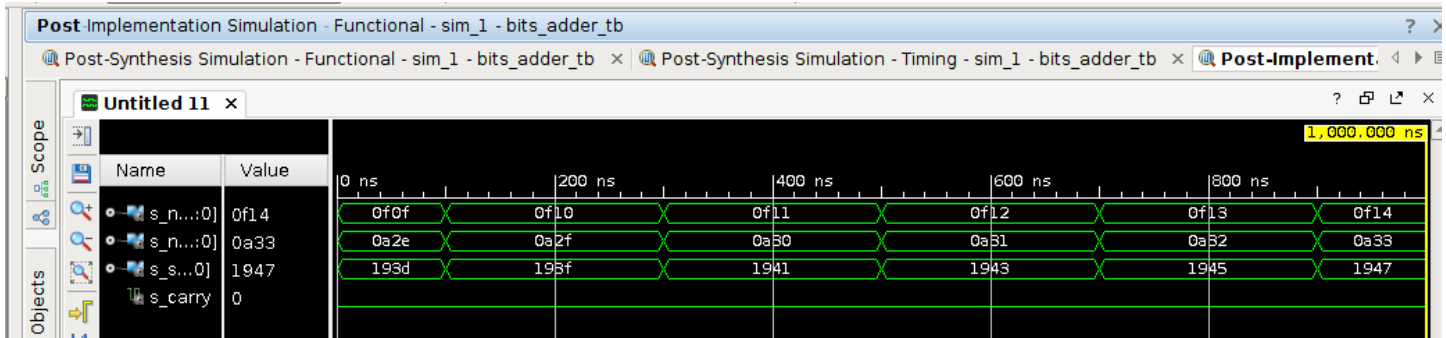
## Synthesis Timing Simulation



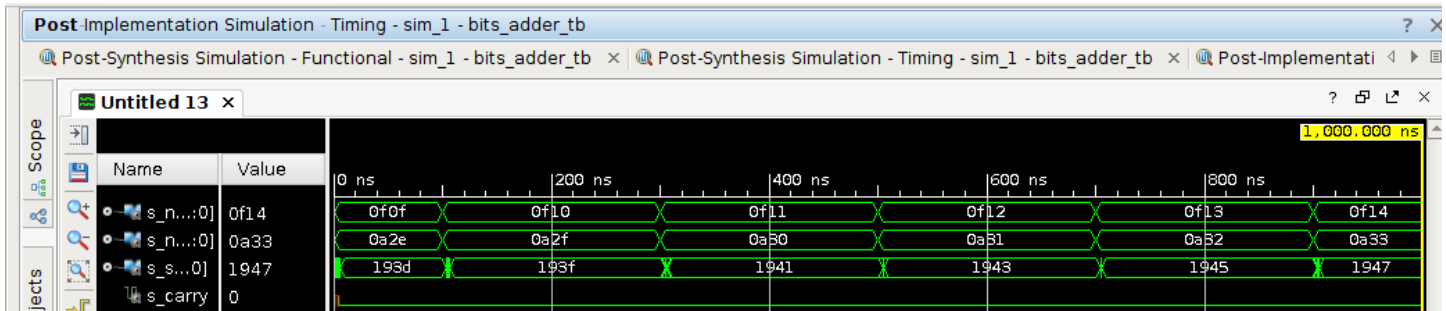
Implementation Design



Synthesis Functional Simulation



Implementation Functional Simulation



Implementation Timing Simulation

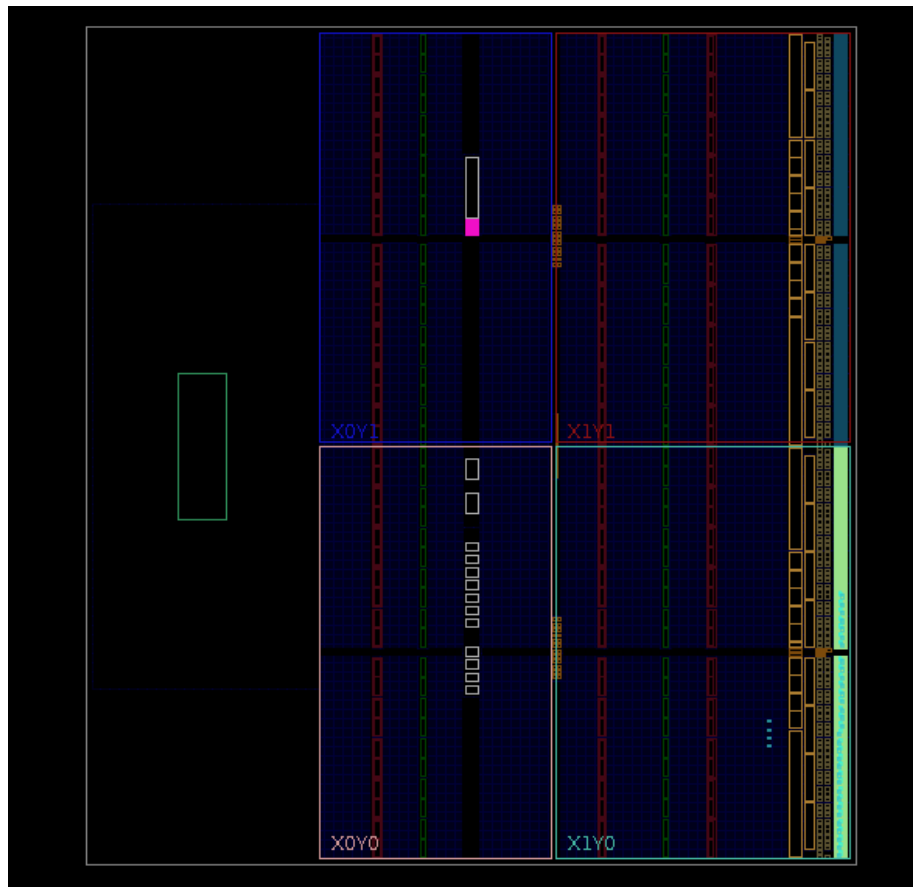
## # INC16 (Incrementador de 16 bits)

```
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 entity n_bit_incrementer is
7     Port (
8         data_A      :    in std_logic_vector(15 downto 0); -- dado de entrada A;
9         data_out     :    out std_logic_vector(15 downto 0) -- dado de saída;
10    );
11 end n_bit_incrementer;
12
13 architecture Behavioral of n_bit_incrementer is
14
15     begin
16
17     data_out <= std_logic_vector(signed(data_A) + 1); -- dado de saída recebe a entrada + 1;
18
19 end Behavioral;
20
```

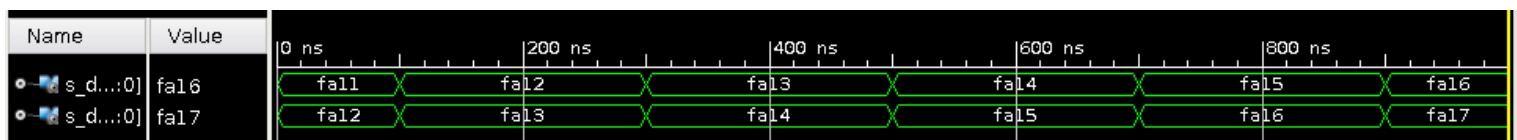
Fizemos este incrementador baseado no exemplo que o livro deu, no qual era um incrementador de 16 bits, na qual a saída receberia a entrada somada em 1. Criamos apenas 2 variáveis, uma de entrada e outra de saída. A única de entrada é o valor em hexadecimal que fixamos na simulação “data\_A” de 16 bits. A saída “data\_out” recebe a entrada somada em 1, dentro da arquitetura. A saída “data\_out” também tem 16 bits.

No teste bench, fizemos a criação de 2 sinais, um para a entrada e outro para a saída. O sinal de entrada fixamos o seu valor em “FA11” em hexadecimal. Fizemos o mapeamento em seguida e dentro do processo fizemos com que a entrada fosse somando em 1 infinitamente. Dessa forma, a saída sempre receberia a entrada acrescentada em 1. Abaixo se encontra o Teste bench e as simulações.

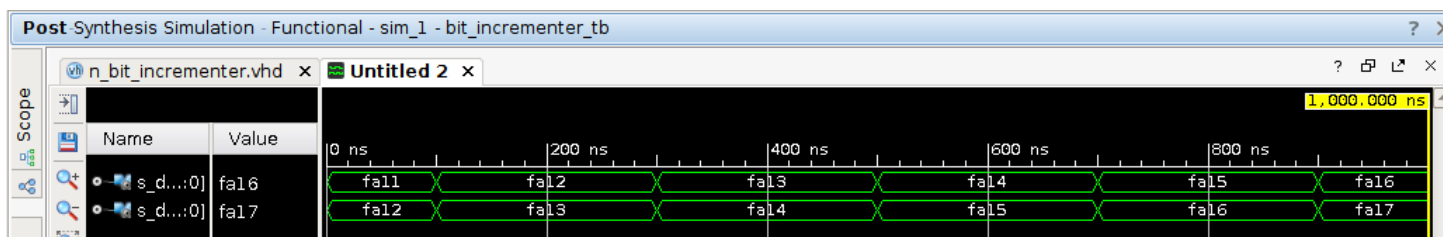
```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use IEEE.NUMERIC_STD.ALL;
4 use IEEE.STD_LOGIC_UNSIGNED.ALL;
5
6 entity bit_incrementer_tb is
7     -- Port ();
8 end bit_incrementer_tb;
9
10 architecture Behavioral of bit_incrementer_tb is
11     component n_bit_incrementer is
12         Port(
13             data_A      :    in std_logic_vector(15 downto 0); -- dado de entrada A;
14             data_out     :    out std_logic_vector(15 downto 0) -- dado de saída;
15         );
16     end component;
17
18     --INPUT
19     signal s_data_A      :    std_logic_vector(15 downto 0) := X"FA11"; -- sinal de entrada;
20
21     --OUTPUT
22     signal s_data_out     :    std_logic_vector(15 downto 0); -- sinal de saída;
23
24     begin
25
26     teste: n_bit_incrementer port map(data_A => s_data_A, data_out => s_data_out); -- mapeamento;
27
28     tb: process
29     begin
30
31     wait for 100 ns;
32
33     s_data_A <= std_logic_vector(signed(s_data_A) + 1); -- incremento dos valores no barramento de 16 bits do sinal de entrada;
34
35     wait for 100 ns;
36
37     end process tb;
38
39 end Behavioral;
40
```



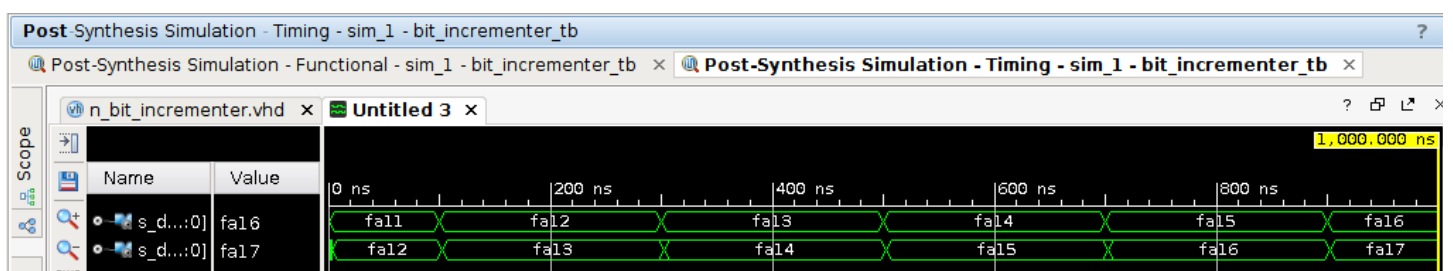
Implementation Design



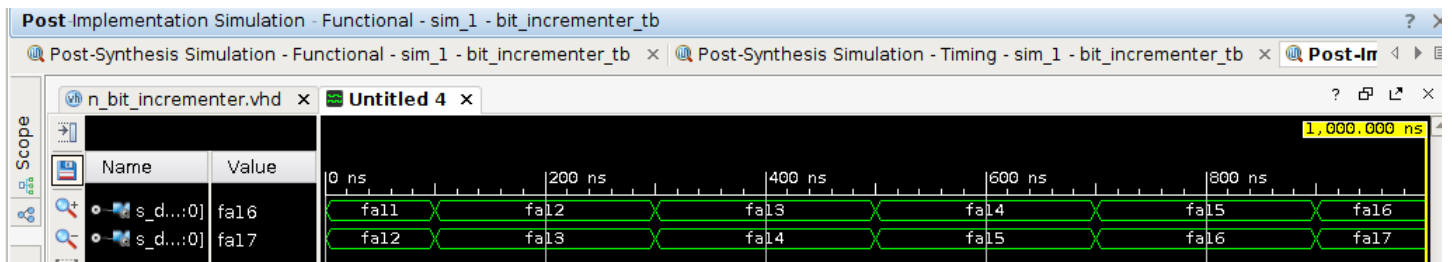
Simulação de comportamento (Behavioral)



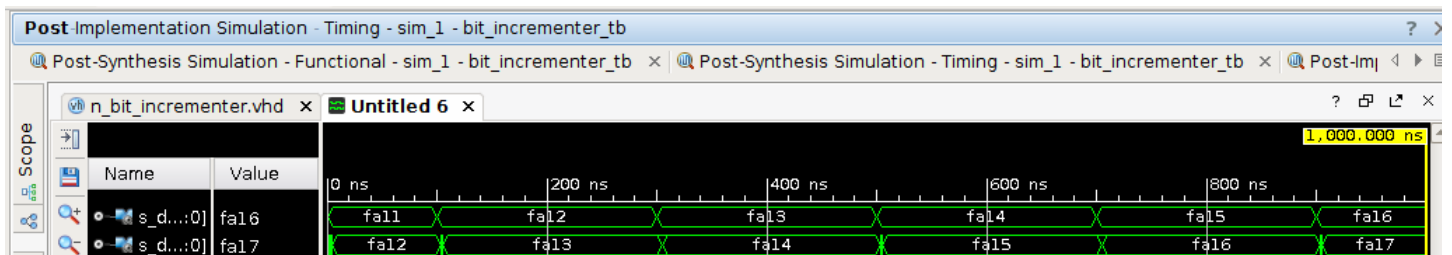
Synthesis Functional Simulation



Synthesis Timing Simulation



## Implementation Functional Simulation



## Implementation Timing Simulation

## # ULA HACK

Começamos com a implementação das entradas e saídas especificadas pelo livro na entidade. Criamos duas entradas de 16 bits “data\_X” e “data\_Y” e uma saída de output “data\_out”. Depois, na mesma entidade, fizemos a criação das opções de modificação das entradas e saídas juntamente com as flags – zx (zera entrada “data\_X”), zy (zera entrada “data\_Y”), nx (nega entrada “data\_X”), ny (nega entrada “data\_Y”), f (qual operação será realizada entre as entradas “data\_X” e “data\_Y”), no(nega a saída “data\_out”), zr(flag para “data\_out” = 0) e ng(flag para “data\_out” < 0).

```

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 entity ula is
7     Port (
8         data_X, data_Y      : in  std_logic_vector(15 downto 0); -- entradas X e Y de 16 bits cada;
9         data_out            : out std_logic_vector(15 downto 0); -- saída data_out de 16 bits;
10        zx, nx              : in  std_logic; -- zera entrada x "zx" e nega entrada x "nx";
11        zy, ny              : in  std_logic; -- zera entrada y "zy" e nega entrada y "ny";
12        no                  : in  std_logic; -- nega a saída data_out;
13        f                   : in  std_logic; -- opção de soma (f = 0) ou and (f = 1);
14        flag_zero           : out std_logic; -- flag para saída data_out = 0;
15        flag_neg            : out std_logic; -- flag para saída data_out < 0;
16    );
17 end ula;

```

Logo em seguida, fizemos a criação dos sinais intermediários que representarão as saídas e entradas dos processos criados na execução do programa. Fizemos a criação de sinais referentes a cada variavel criada na entidade seguida de um “s\_”. Todos esses sinais criados foram implementados na forma de std\_logic\_vector.



```

19 architecture Behavioral of ula is
20
21     signal s_Zero_x      :      std_logic_vector(15 downto 0);      -- sinal para zerar a entrada data_X;
22     signal s_Zero_y      :      std_logic_vector(15 downto 0);      -- sinal para zerar a entrada data_Y;
23
24     signal s_Neg_x        :      std_logic_vector(15 downto 0);      -- sinal para negar a entrada data_X;
25     signal s_Neg_y        :      std_logic_vector(15 downto 0);      -- sinal para negar a entrada data_Y;
26
27     signal s_funcao_f      :      std_logic_vector(15 downto 0);      -- sinal para receber opção de operação;
28     signal s_final         :      std_logic_vector(15 downto 0);      -- sinal para receber a saída final data_out;
29
30

```

Fizemos a criação de vários processos em vez de apenas um para aproveitar bem os recursos da linguagem VHDL. No primeiro processo com rótulo “Zero\_X” recebemos a entrada “data\_X” e a chave seletora “zx”. Dentro do processo fizemos um condicional para verificar se “zx” for igual a 1, então o sinal criado “s\_Zero\_x” receberá 0 em hexadecimal. Caso contrário, “s\_Zero\_x” recebe “data\_X”. Logo em seguida fizemos outro processo semelhante ao anterior, rotulado de “Zero\_Y” para tratar a entrada “data\_Y”.

```

31 begin
32     ----- opcao para zerar a entrada
33 Zero_X: process(data_X, zx)
34 begin
35     if zx = '1' then
36         s_Zero_x <= X"0000";
37     else
38         s_Zero_x <= data_X;
39     end if;
40 end process ;
41
42 Zero_Y: process(data_Y, zy)
43 begin
44     if zx = '1' then
45         s_Zero_y <= X"0000";
46     else
47         s_Zero_y <= data_Y;
48     end if;
49 end process;
50

```

Logo após esses dois processos, criamos mais dois processos responsáveis por tratar a saída dos dois processos anteriores. Primeiro criamos um processo chamado de “Neg\_X”, que recebe como a saída do processo anterior (s\_Zero\_x e nx). Caso, nx for igual a 1, então o sinal “s\_Neg\_x” recebe o resultado da operação anterior negada. Caso contrário, “s\_Neg\_x” recebe “s\_Zero\_x”. No processo seguinte fizemos algo semelhante, porém com as entradas “s\_Zero\_Y” e “ny”.

```

51 ----- opcao pra negar a entrada anterior
52 Neg_X: process(s_Zero_x,nx)
53   --variable X : std_logic_vector(15 downto 0);
54   begin
55     --X := s_Zero_x;
56
57     if nx = '1' then
58       s_Neg_x <= not s_Zero_x;
59       -- data_out <= X after 10ns;
60     else
61       s_Neg_x <= s_Zero_x;
62     end if;
63   end process;
64
65
66 Neg_Y: process(s_Zero_y,ny)
67   --variable X : std_logic_vector(15 downto 0);
68   begin
69     --X := s_Zero_x;
70
71     if ny = '1' then
72       s_Neg_y <= not s_Zero_y;
73       -- data_out <= X after 10ns;
74     else
75       s_Neg_y <= s_Zero_y;
76     end if;
77   end process;
78
79

```

Em seguida, fizemos a criação de um processo responsável pela escolha do tipo de operação a ser realizada (Soma ou AND) com as saídas anteriores. O processo chamado de “SOMA\_OU\_AND”, é sensível as saídas dos processos anteriores “s\_Neg\_X” e “s\_Neg\_Y” e a chave seletora “f”. Caso a chave esteja em nível lógico alto “f = 1”, então o sinal “s\_funcao\_f” recebe o sinal da soma entre as duas entradas (s\_Neg\_x e s\_Neg\_y). Caso contrário, “s\_funcao\_f” recebe “s\_Neg\_x and s\_Neg\_y”. Com 0 fazemos a operação de soma bit a bit entre os sinais de entrada e 1 fazemos a operação AND.

```

79 ----- Somar ou fazer o and bit a bit
80
81 SOMA_OU_AND: process (s_Neg_x,s_Neg_y,f)
82   begin
83
84     if f = '1' then s_funcao_f <= s_Neg_x + s_Neg_y;
85     else
86       s_funcao_f <= s_Neg_x and s_Neg_y;
87     end if;
88   end process ;
89
90

```

Após isso, criamos outro processo sensível as entradas “s\_funcao\_f” - sinal de saída do processo anterior – e “no”. O processo foi chamado de “negar\_saida”. Criamos dentro do processo uma variável do tipo signed de tamanho 16 bits (X). Logo após fizemos X ser igual ao valor de “s\_funcao\_f”. Se “no” for igual a 1, então X recebe “not X” e o sinal “s\_final” recebe o valor de X. Caso contrário, “s\_final” recebe X.

```

90 ----- negar a saída
91
92 negar_saida: process (s_funcao_f, no)
93 variable X : signed (15 downto 0);
94 begin
95     X := signed(s_funcao_f);
96     if NO = '1' then
97         X := not X;
98         s_final <= std_logic_vector( X);
99     else
100         s_final <= s_funcao_f;
101     end if;
102 end process;
103

```

Como não é possível usar a saída para realizar operações, criamos processos para a verificação das flags de status. Criamos um processo chamado “final\_flag\_zero”, que recebe o sinal “s\_final”. Se o “s\_final” for igual a “0”, então “flag\_zero” recebe 1. Caso contrário “flag\_zero” recebe “0”. Criamos um outro processo para determinar se a saída será negativa. O processo chamado de “final\_flag\_neg” recebe “s\_final”. Como para determinar se um número é negativo, olhamos apenas para o bit mais significativo (MSB), nesse caso o bit 15, e se for igual a 1, a flag é ativada, ou seja, “flag\_neg” = 1. Caso contrário, “flag\_neg” = 0.

```

104 ----- flag de saída para 0
105
106 final_flag_zero: process(s_final )
107 begin
108     if s_final = "0" then flag_zero <= '1';
109     else
110         flag_zero <= '0';
111     end if;
112 end process;
113
114 ----- flag de sinal para a saída
115
116 final_flag_neg: process(s_final )
117 begin
118     if s_final(15) = '1' then flag_neg <= '1';
119     else
120         flag_neg <= '0';
121     end if;
122 end process;
123
124

```

Finalmente, criamos um processo que após todas as operações coloca o valor da saída em “data\_out”. O nome do processo é “saida\_final”, e é sensível ao sinal “s\_final”. Toda essa estrutura é como se vários Multiplexadores funcionassem concorrentemente.

```

126 ----- saída data_out
127
128   saída_final: process (s_final)
129     variable X : signed (15 downto 0);
130     begin
131       X := signed(s_final);
132       data_out <= std_logic_vector(X);
133
134     end process;
135 end Behavioral;
136

```

No Teste bench copiamos a entidade no componente e em seguida fizemos a criação de vários sinais referentes a cada entrada e saída. Fizemos o mapeamento em seguida e logo após ficou aberto para analisar todos os 18 casos e os mais outros que não estão representados no livro. Na criação dos sinais referentes as entradas X e Y, fixamos valores para ambos. Abaixo está o teste bench da ULA.

```

1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.NUMERIC_STD.ALL;
4
5  entity ula_tb is
6    -- Port ( );
7  end ula_tb;
8
9  architecture Behavioral of ula_tb is
10    component ula is
11      Port(
12        data_X, data_Y : in std_logic_vector(15 downto 0); -- entradas X e Y de 16 bits cada;
13        data_out : out std_logic_vector(15 downto 0); -- saída data_out de 16 bits;
14        zx, nx : in std_logic; -- zera entrada x "zx" e nega entrada x "nx";
15        f : in std_logic; -- opção de soma (f = 0) ou and (f = 1);
16        no : in std_logic; -- nega a saída data_out;
17        zy, ny : in std_logic; -- zera entrada y "zy" e nega entrada y "ny";
18        flag_zero : out std_logic; -- flag para saída data_out = 0;
19        flag_neg : out std_logic; -- flag para saída data_out < 0;
20      );
21    end component;
22
23    signal s_data_X : std_logic_vector(15 downto 0) := X"FFFF"; -- atribuição de sinal de entrada data_X;
24    signal s_data_Y : std_logic_vector(15 downto 0) := X"000F"; -- atribuição de sinal de entrada data_Y;
25    signal s_data_out : std_logic_vector(15 downto 0); -- sinal para a saída data_out;
26    signal s_f : std_logic := '0'; -- sinal para a escolha de operação;
27    signal s_zx : std_logic := '0'; -- sinal para zerar entrada data_X;
28    signal s_nx : std_logic := '0'; -- sinal para negar entrada data_X;
29    signal s_zy : std_logic := '0'; -- sinal para zerar entrada data_Y;
30    signal s_ny : std_logic := '0'; -- sinal para negar entrada data_Y;
31    signal s_no : std_logic := '0'; -- sinal para negar a saída data_out;
32    signal s_flag_zero : std_logic := '0'; -- sinal de flag data_out = 0;
33    signal s_flag_neg : std_logic := '0'; -- sinal de flag data_out < 0;
34
35  begin
36
37    -- mapeamento;
38
39    teste: ula port map(flag_zero => s_flag_zero, flag_neg => s_flag_neg, no => s_no, data_X => s_data_X, data_Y => s_data_Y, data_out => s_data_out, zx => s_zx, zy => s_zy,
40                      f => s_f, nx => s_nx, ny => s_ny);
41

```

Agora em seguida faremos todos os testes propostos pelo livro. Em seguida vamos analisar se haverá redundâncias testando outras sequências com os mesmos valores de entrada para data\_X e data\_Y.

## Simulação 1

### Entradas de dados:

data\_X = "FFFF"

data\_Y = "000F"

### Entradas seletoras:

zx: 1

nx: 0

zy: 1

ny: 0

f: 1

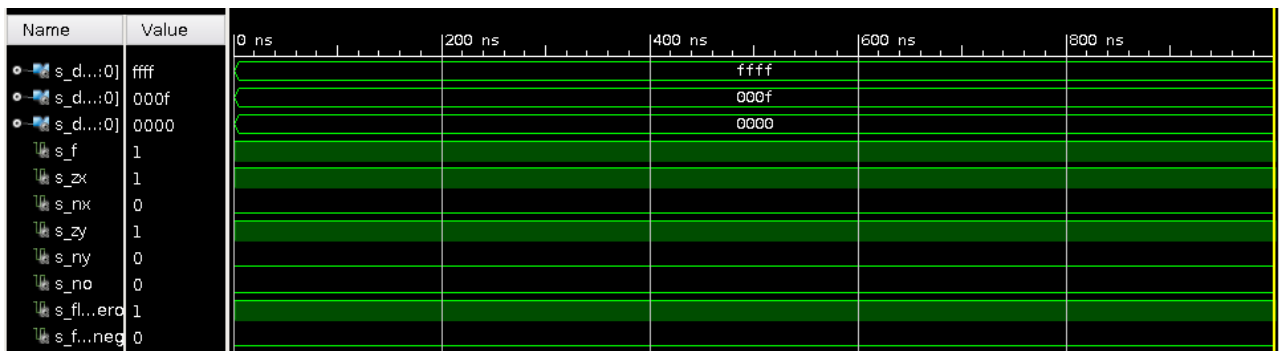
no: 0

```
--p : process
--begin
--wait    for 100 ns;
s_zx <= '1';      -- valor de zx;
--wait    for 100 ns;
s_nx <= '0';      -- valor de nx;
--wait    for 100 ns;
s_zy <= '1';      -- valor de zy;
--wait    for 100 ns;
s_ny <= '0';      -- valor de ny;
--wait    for 100 ns;
s_f <= '1';       -- valor de f;
--wait    for 100 ns;
s_no <= '0';      -- valor de no;

--end process p;
end Behavioral;
```

### Saída de dados:

data\_out: 0



## Simulação 2

### Entradas de dados:

data\_X = "FA12"

data\_Y = "0012"

### Entradas seletoras:

zx: 1

nx: 0

zy: 1

ny: 0

f: 1

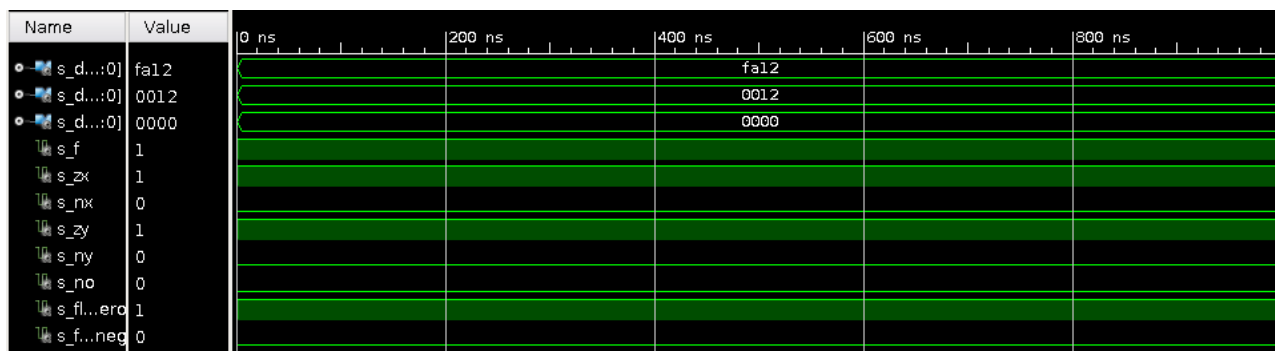
no: 0

```
--p : process
--begin
--wait    for 100 ns;
s_zx <= '1';      -- valor de zx;
--wait    for 100 ns;
s_nx <= '0';      -- valor de nx;
--wait    for 100 ns;
s_zy <= '1';      -- valor de zy;
--wait    for 100 ns;
s_ny <= '0';      -- valor de ny;
--wait    for 100 ns;
s_f <= '1';       -- valor de f;
--wait    for 100 ns;
s_no <= '0';      -- valor de no;

--end process p;
end Behavioral;
```

### Saída de dados:

data\_out = 0.



### Simulação 3

#### Entradas de dados:

data\_X = “FA12”

data\_Y = “0012”

#### Entradas seletoras:

zx: 1

nx: 1

zy: 1

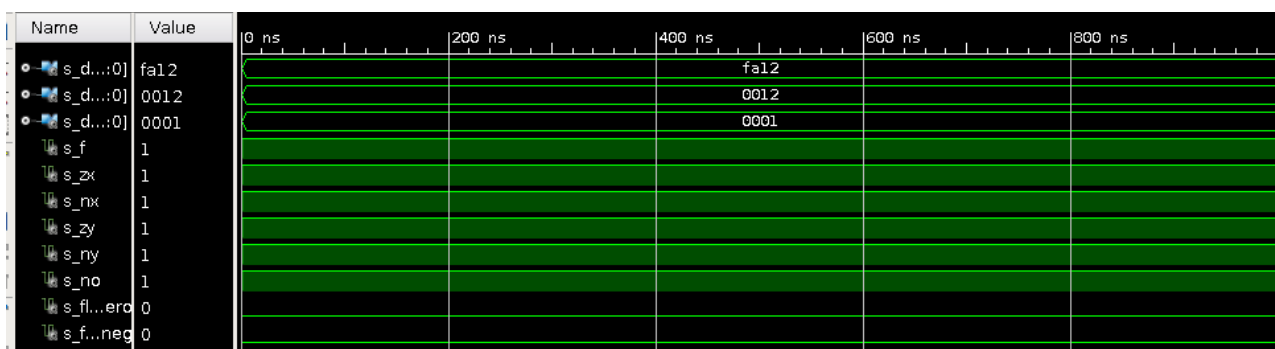
ny: 1

f: 1

no: 1

#### Saída de dados:

data\_out: 1



## Simulação 4

### Entradas de dados:

data\_X = "FA12"

data\_Y = "0012"

### Entradas seletoras:

zx: 1

nx: 1

zy: 1

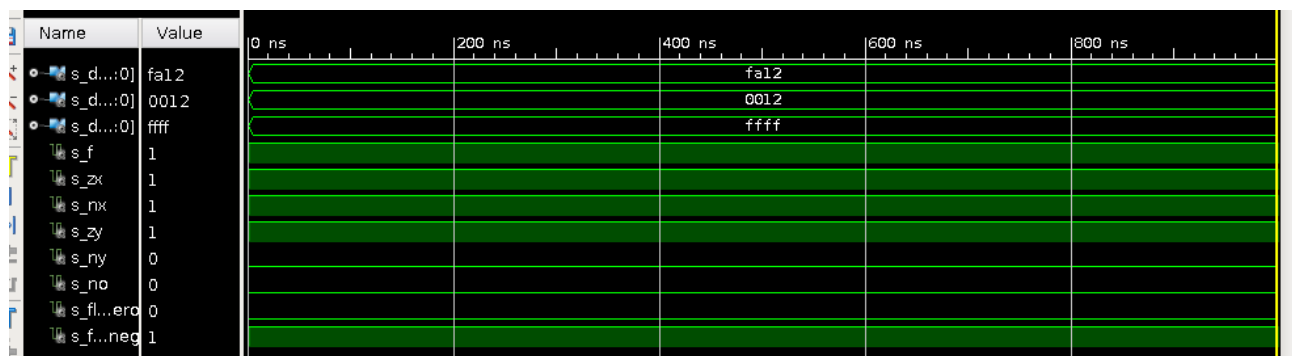
ny: 0

f: 1

no: 0

### Saída de dados:

data\_out: -1



## Simulação 5

### Entradas de dados:

data\_X = "FF00"

data\_Y = "0002"

### Entradas seletoras:

zx: 0

nx: 0

zy: 1

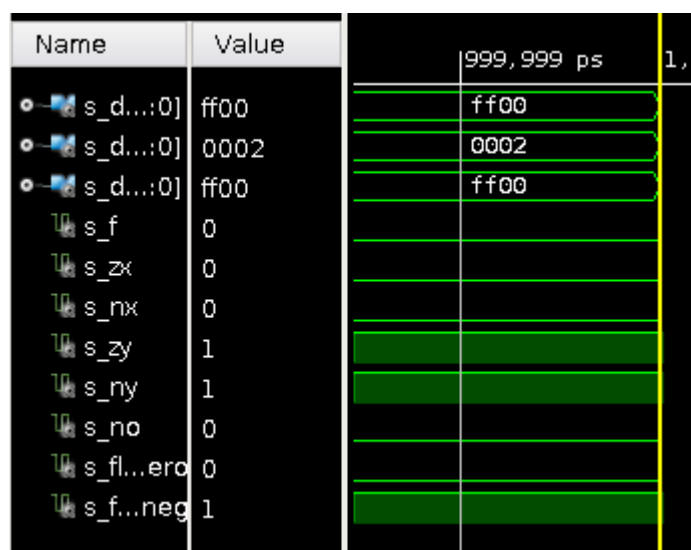
ny: 1

f: 0

no: 0

### Saída de dados:

data\_out: data\_X



## Simulação 6

### Entradas de dados:

data\_X = "FF00"

data\_Y = "0002"

### Entradas seletoras:

zx: 1

nx: 1

zy: 0

ny: 0

f: 0

no: 0

### Saída de dados:

data\_out: data\_Y

Name	Value		999,999 ps	1
s_d...:0]	ff00		ff00	
s_d...:0]	0002		0002	
s_d...:0]	0002		0002	
s_f	0			
s_zx	1			
s_nx	1			
s_zy	0			
s_ny	0			
s_no	0			
s_fl...ero	0			
s_f...neg	0			

## Simulação 7

### Entradas de dados:

data\_X = "FF00"

data\_Y = "0002"

### Entradas seletoras:

zx: 0

nx: 0

zy: 1

ny: 1

f: 0

no: 1

### Saída de dados:

data\_out: not data\_X

Name	Value		999,999 ps	1
s_d...:0]	ff00		ff00	
s_d...:0]	0002		0002	
s_d...:0]	00ff		00ff	
s_f	0			
s_zx	0			
s_nx	0			
s_zy	1			
s_ny	1			
s_no	1			
s_fl...ero	0			
s_f...neg	0			



## Simulação 8

### Entradas de dados:

data\_X = "FF00"

data\_Y = "0002"

### Entradas seletoras:

zx: 1

nx: 1

zy: 0

ny: 0

f: 0

no: 1

### Saída de dados:

data\_out: not data\_Y

Name	Value	999,999 ps	1
s_d...:0]	ff00	ff00	
s_d...:0]	0002	0002	
s_d...:0]	ffff	ffff	
s_f	0		
s_zx	1		
s_nx	1		
s_zy	0		
s_ny	0		
s_no	1		
s_fl...ero	0		
s_f...neg	1		

## Simulação 9

### Entradas de dados:

data\_X = "FF00"

data\_Y = "0002"

### Entradas seletoras:

zx: 0

nx: 0

zy: 1

ny: 1

f: 1

no: 1

### Saída de dados:

data\_out: - data\_X

Name	Value	999,999 ps	1,
s_d...:0]	ff00	ff00	
s_d...:0]	0002	0002	
s_d...:0]	0100	0100	
s_f	1		
s_zx	0		
s_nx	0		
s_zy	1		
s_ny	1		
s_no	1		
s_fl...ero	0		
s_f...neg	0		

### Simulação 10

### Entradas de datos:

```
data_X = "FF00"
```

```
data_Y = "0002"
```

### Entradas seletoras:

**ZX:** 1

nx: 1

$$zy: \quad 0$$

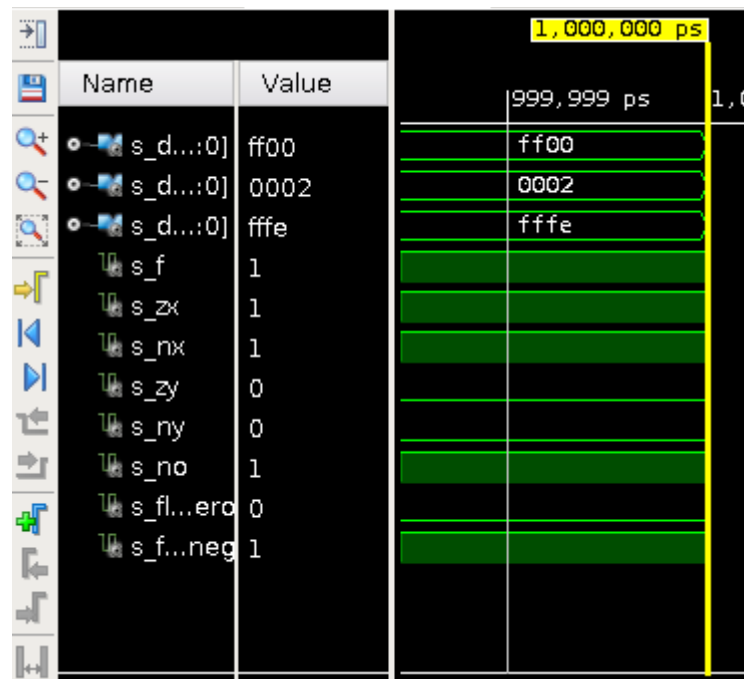
ny: 0

f: 1

no: 1

### Saída de dados:

```
data_out: - data_Y
```



### Simulação 11

### Entradas de datos:

```
data_X = "FF00"
```

```
data_Y = "0002"
```

**Entradas seletoras:**

zx: 0

nx: 1

zy: 1

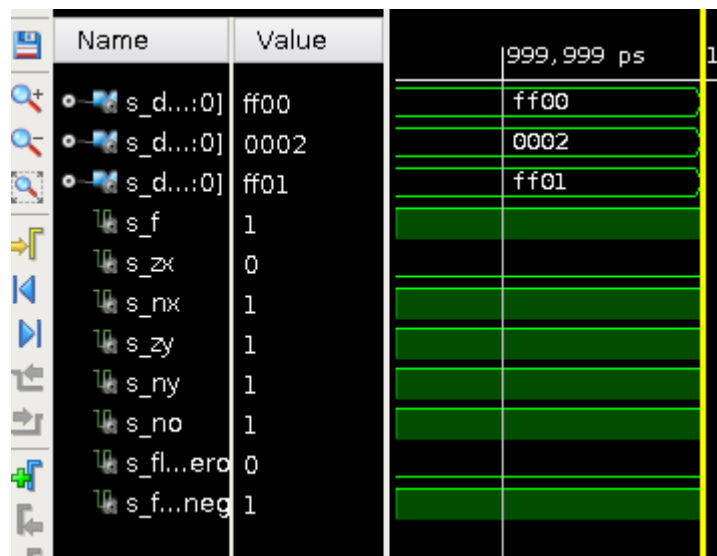
ny: 1

f: 1

no: 1

### Saída de dados:

```
data_out: data_X + 1
```



## Simulação 12

### Entradas de dados:

data\_X = "FF00"

data\_Y = "0002"

### Entradas seletoras:

zx: 1

nx: 1

zy: 0

ny: 1

f: 1

no: 1

### Saída de dados:

data\_out: data\_Y + 1

Name	Value		999,999 ps	1,
s_d...:0]	ff00		ff00	
s_d...:0]	0002		0002	
s_d...:0]	0003		0003	
s_f	1			
s_zx	1			
s_nx	1			
s_zy	0			
s_ny	1			
s_no	1			
s_fl...ero	0			
s_f...neg	0			

## Simulação 13

### Entradas de dados:

data\_X = "FF00"

data\_Y = "0002"

### Entradas seletoras:

zx: 0

nx: 0

zy: 1

ny: 1

f: 1

no: 0

### Saída de dados:

data\_out: data\_X - 1

Name	Value		1,000,000 ps	
s_d...:0]	ff00		999,999 ps	1,
s_d...:0]	0002		ff00	
s_d...:0]	feff		0002	
s_f	1		feff	
s_zx	0			
s_nx	0			
s_zy	1			
s_ny	1			
s_no	0			
s_fl...ero	0			
s_f...neg	1			

## Simulação 14

### Entradas de dados:

data\_X = "FF00"

data\_Y = "0002"

### Entradas seletoras:

zx: 1

nx: 1

zy: 0

ny: 0

f: 1

no: 0

Name	Value	999,999 ps	1
s_d...:0]	ff00	ff00	
s_d...:0]	0002	0002	
s_d...:0]	0001	0001	
s_f	1		
s_zx	1		
s_nx	1		
s_zy	0		
s_ny	0		
s_no	0		
s_fl...ero	0		
s_f...neg	0		

### Saída de dados:

data\_out: data\_Y - 1

## Simulação 15

### Entradas de dados:

data\_X = "FF00"

data\_Y = "0002"

### Entradas seletoras:

zx: 0

nx: 0

zy: 0

ny: 0

f: 1

no: 0

Name	Value	999,999 ps	1
s_d...:0]	ff00	ff00	
s_d...:0]	0002	0002	
s_d...:0]	ff02	ff02	
s_f	1		
s_zx	0		
s_nx	0		
s_zy	0		
s_ny	0		
s_no	0		
s_fl...ero	0		
s_f...neg	1		

### Saída de dados:

data\_out: data\_X + data\_Y

## Simulação 16

### Entradas de dados:

data\_X = "FF00"

data\_Y = "0002"

### Entradas seletoras:

zx: 0

nx: 1

zy: 0

ny: 0

f: 1

no: 1

### Saída de dados:

data\_out: data\_X - data\_Y

Name	Value	999,999 ps	1,0
s_d...:0]	ff00	ff00	
s_d...:0]	0002	0002	
s_d...:0]	fefe	fefe	
s_f	1		
s_zx	0		
s_nx	1		
s_zy	0		
s_ny	0		
s_no	1		
s_fl...ero	0		
s_f...neg	1		

## Simulação 17

### Entradas de dados:

data\_X = "FF00"

data\_Y = "0002"

### Entradas seletoras:

zx: 0

nx: 0

zy: 0

ny: 1

f: 1

no: 1

### Saída de dados:

data\_out: data\_Y - data\_X

Name	Value	999,999 ps	1,0
s_d...:0]	ff00	ff00	
s_d...:0]	0002	0002	
s_d...:0]	0102	0102	
s_f	1		
s_zx	0		
s_nx	0		
s_zy	0		
s_ny	1		
s_no	1		
s_fl...ero	0		
s_f...neg	0		

## Simulação 18

### Entradas de dados:

data\_X = "FF00"

data\_Y = "0002"

### Entradas seletoras:

zx: 0

nx: 0

zy: 0

ny: 0

f: 0

no: 0

### Saída de dados:

data\_out: data\_X and data\_Y

Name	Value	999,999 ps	1,0
s_d...:0]	ff00	ff00	
s_d...:0]	0002	0002	
s_d...:0]	0000	0000	
s_f	0		
s_zx	0		
s_nx	0		
s_zy	0		
s_ny	0		
s_no	0		
s_fl...ero	1		
s_f...neg	0		

## Simulação 19

### Entradas de dados:

data\_X = "FF00"

data\_Y = "0002"

### Entradas seletoras:

zx: 0

nx: 1

zy: 0

ny: 1

f: 0

no: 1

### Saída de dados:

data\_out: data\_X or data\_Y

Name	Value	999,999 ps	1,0
s_d...:0]	ff00	ff00	
s_d...:0]	0002	0002	
s_d...:0]	ff02	ff02	
s_f	0		
s_zx	0		
s_nx	1		
s_zy	0		
s_ny	1		
s_no	1		
s_fl...ero	0		
s_f...neg	1		

Fizemos os 18 casos apresentados pelo livro e todos corresponderam com o resultado final. Agora vamos fazer mais alguns casos para verificar se outras combinações podem ou não ser redundantes.

## Simulação 20

### Entradas de dados:

data\_X = "FFFF"

data\_Y = "0F00"

### Entradas seletoras:

zx: 1

nx: 0

zy: 0

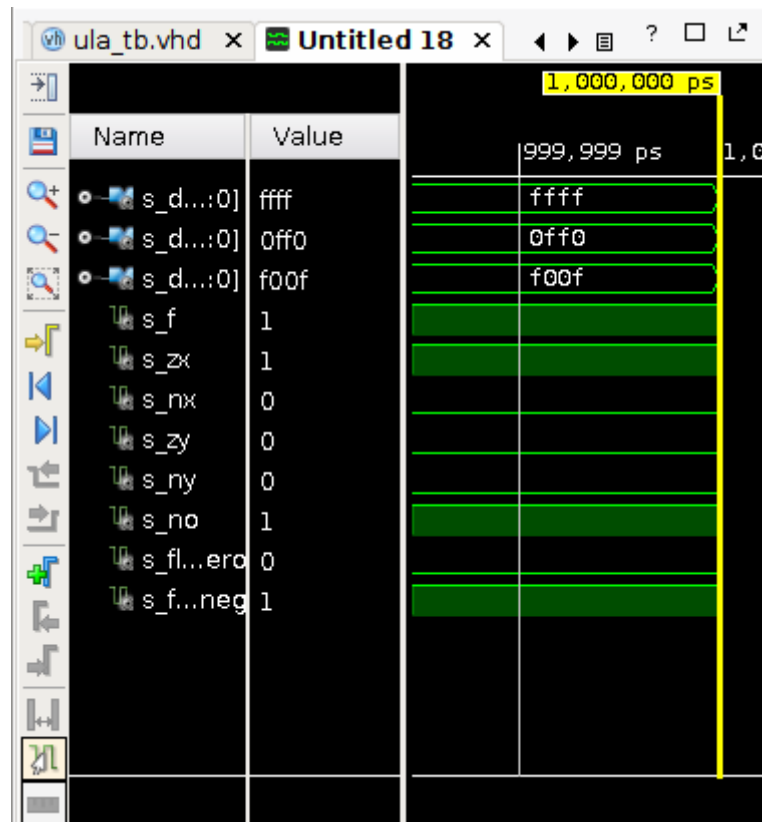
ny: 0

f: 1

no: 1

### Saída de dados:

data\_out: not data\_Y



## Simulação 21

### Entradas de dados:

data\_X = "FFFF"

data\_Y = "0F00"

### Entradas seletoras:

zx: 1

nx: 0

zy: 0

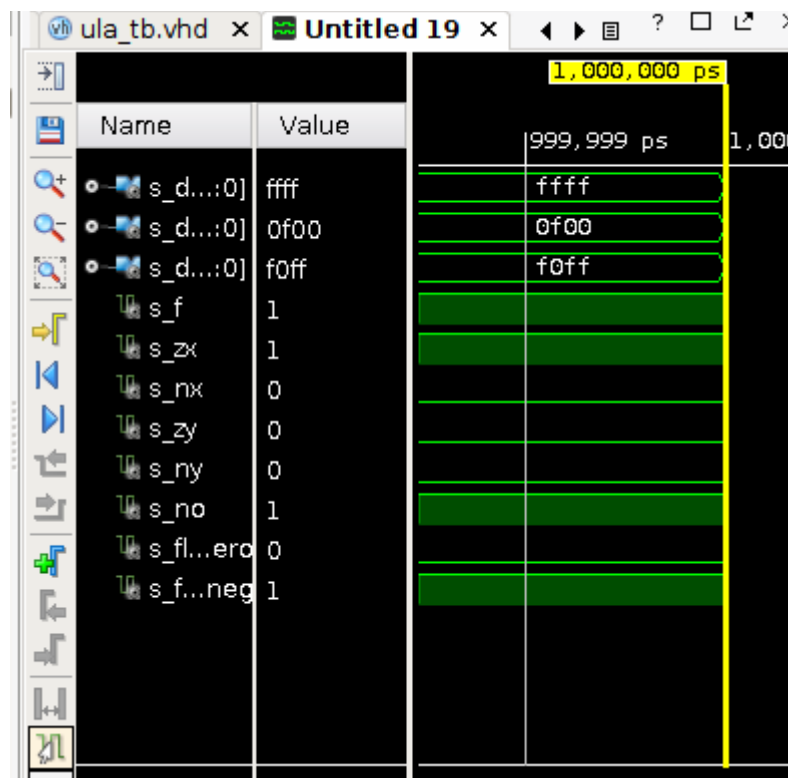
ny: 0

f: 1

no: 1

### Saída de dados:

data\_out: not data\_Y



Na simulação 20 e 21, percebemos que independentemente das entradas data\_X e data\_Y a combinação se:  $zx = 1$ ,  $nx = 0$ ,  $zy = 0$ ,  $ny = 0$ ,  $f = 1$  e  $no = 1$ , a saída de dados é igual a not data\_Y. Concluimos que esta combinação de chaves seletoras é redundante em relação as anteriores mostradas no livro.

### **Simulação 22**

Entradas de dados:

data\_X = "FF0F"

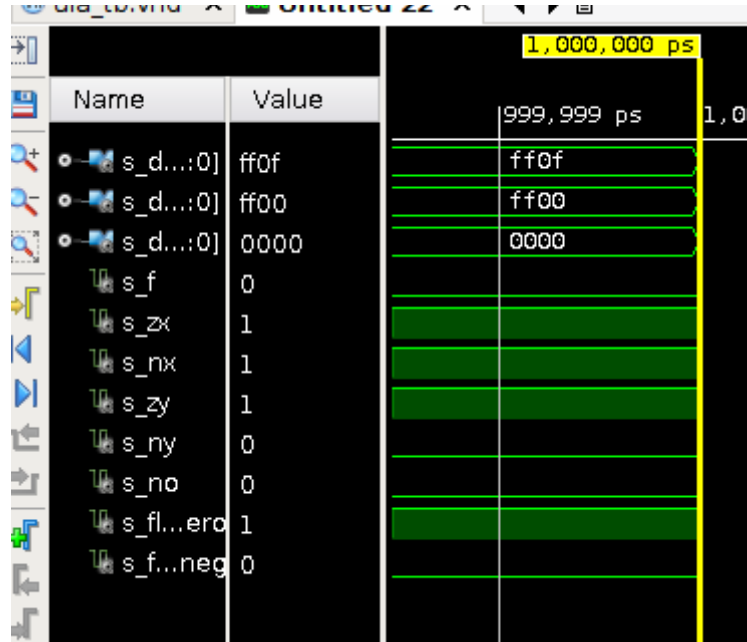
data\_Y = "FF00"

Entradas seletoras:

zx: 1  
nx: 1  
zy: 1  
ny: 0  
f: 0  
no: 0

Saída de dados:

data\_out: zero "0".



### **Simulação 23**

Entradas de dados:

data\_X = "FFFF"

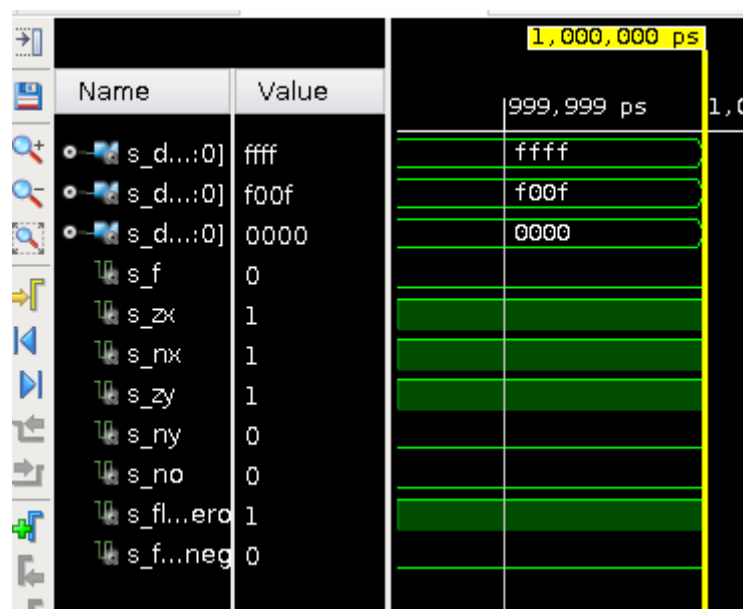
data\_Y = "F00F"

Entradas seletoras:

zx: 1  
nx: 1  
zy: 1  
ny: 0  
f: 0  
no: 0

Saída de dados:

data\_out: zero "0".



De acordo com as simulações 22 e 23, percebemos que independentemente das entradas data\_X e data\_Y a combinação se:  $zx = 1$ ,  $nx = 1$ ,  $zy = 1$ ,  $ny = 0$ ,  $f = 0$  e  $no = 0$ , a saída de dados é igual a zero "0". Concluimos que esta combinação de chaves seletoras é redundante em relação as anteriores mostradas no livro.



### Simulação 23

#### Entradas de dados:

data\_X = "FFFF"

data\_Y = "F00F"

#### Entradas seletoras:

zx: 1

nx: 1

zy: 1

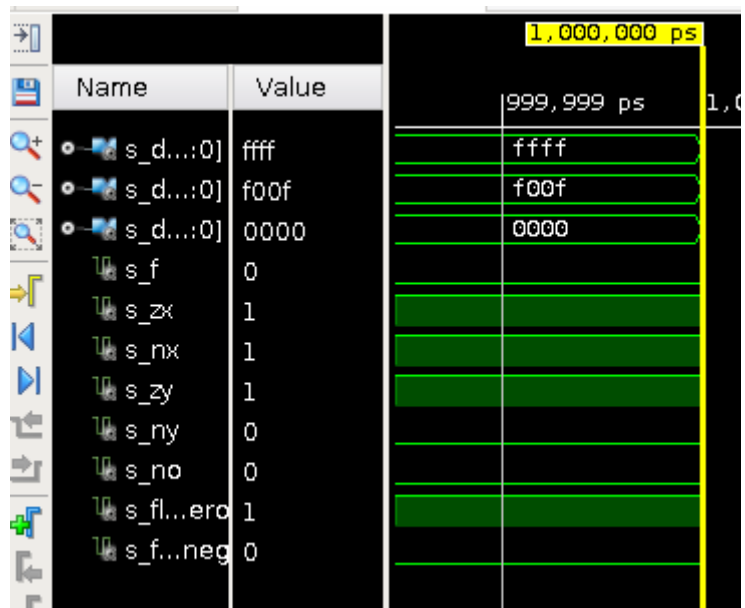
ny: 0

f: 0

no: 0

#### Saída de dados:

data\_out: zero "0".



Name	Value
s_d...:0	ffff
s_d...:0	f00f
s_d...:0	0000
s_f	0
s_zx	1
s_nx	1
s_zy	1
s_ny	0
s_no	0
s_fl...ero	1
s_f...neg	0

De acordo com as simulações 22 e 23, percebemos que independentemente das entradas data\_X e data\_Y a combinação se: zx = 1, nx = 1, zy = 1, ny = 0, f = 0 e no = 0, a saída de dados é igual a zero "0". Concluimos que esta combinação de chaves seletoras é redundante em relação as anteriores mostradas no livro.

### Simulação 24

#### Entradas de dados:

data\_X = "0FFF"

data\_Y = "0F00"

#### Entradas seletoras:

zx: 0

nx: 1

zy: 0

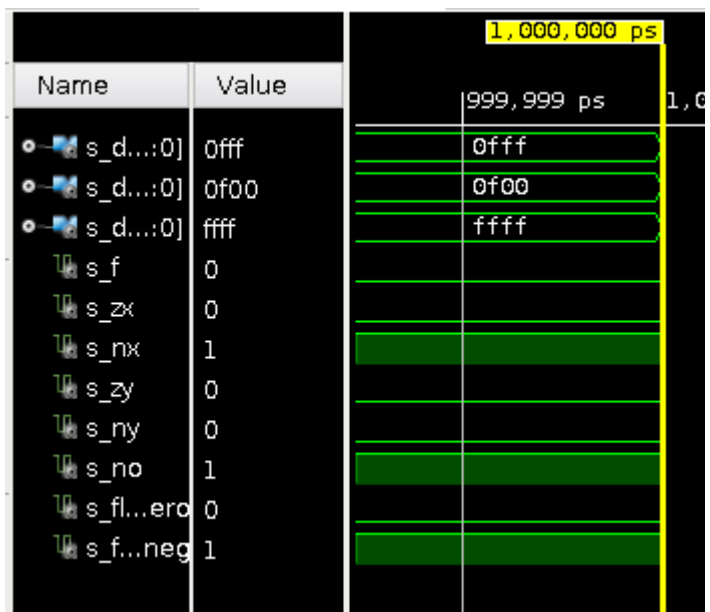
ny: 0

f: 0

no: 1

#### Saída de dados:

data\_out: "FFFF".



Name	Value
s_d...:0	0fff
s_d...:0	0f00
s_d...:0	ffff
s_f	0
s_zx	0
s_nx	1
s_zy	0
s_ny	0
s_no	1
s_fl...ero	0
s_f...neg	1

## Simulação 24

### Entradas de dados:

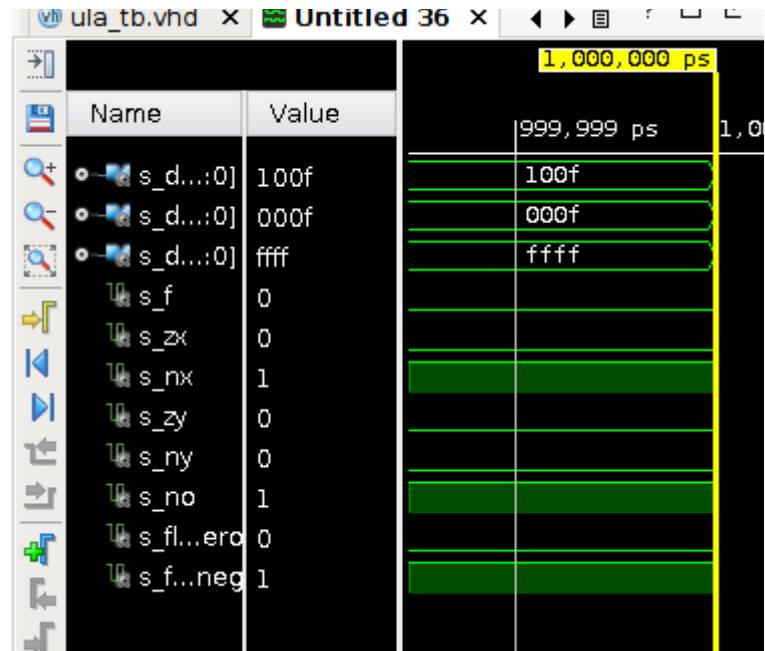
data\_X = "100F"  
data\_Y = "000F"

### Entradas seletoras:

zx: 0  
nx: 1  
zy: 0  
ny: 0  
f: 0  
no: 1

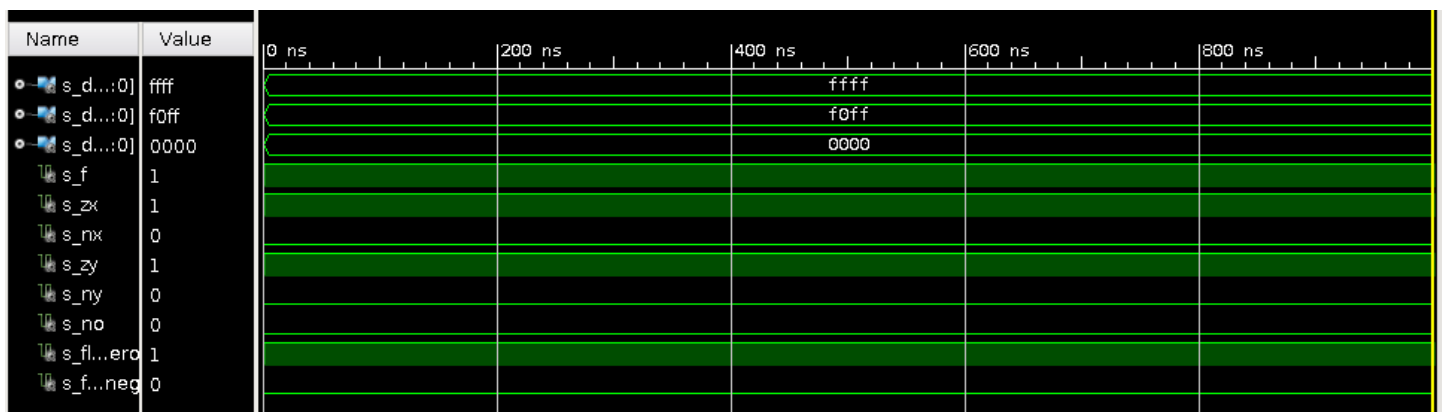
### Saída de dados:

data\_out: "FFFF";

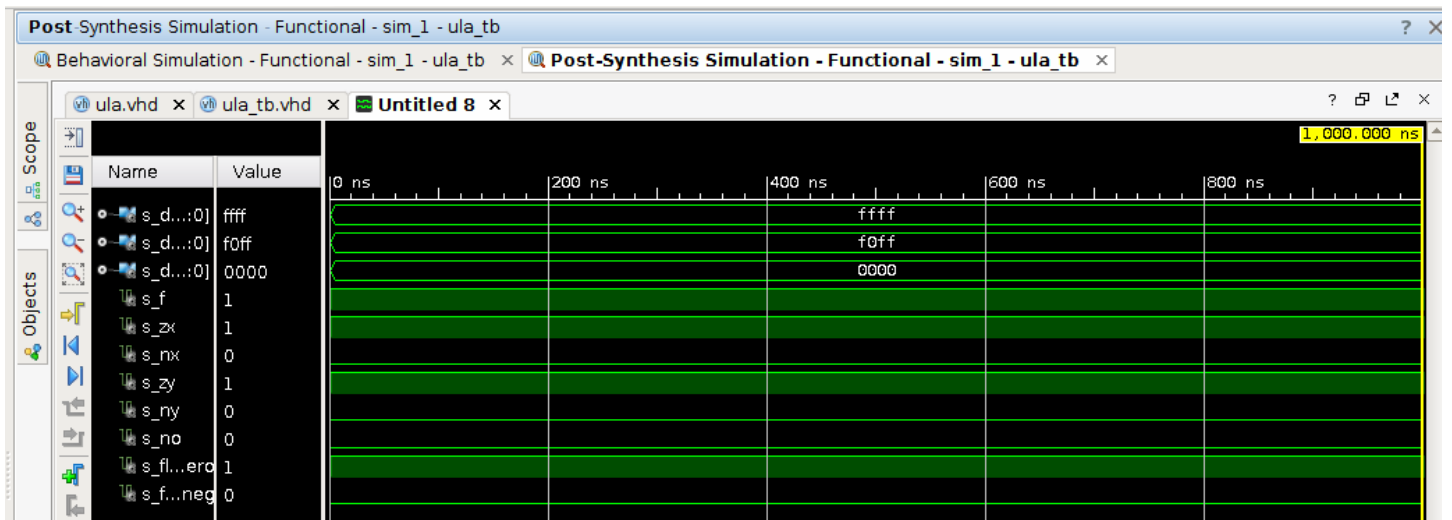


De acordo com as simulações 23 e 24, percebemos que independentemente das entradas data\_X e data\_Y a combinação se: zx = 0, nx = 1, zy = 0, ny = 0, f = 0 e no = 1, a saída de dados é igual a "FFFF". Concluimos que esta combinação de chaves seletoras é redundante em relação as anteriores mostradas no livro.

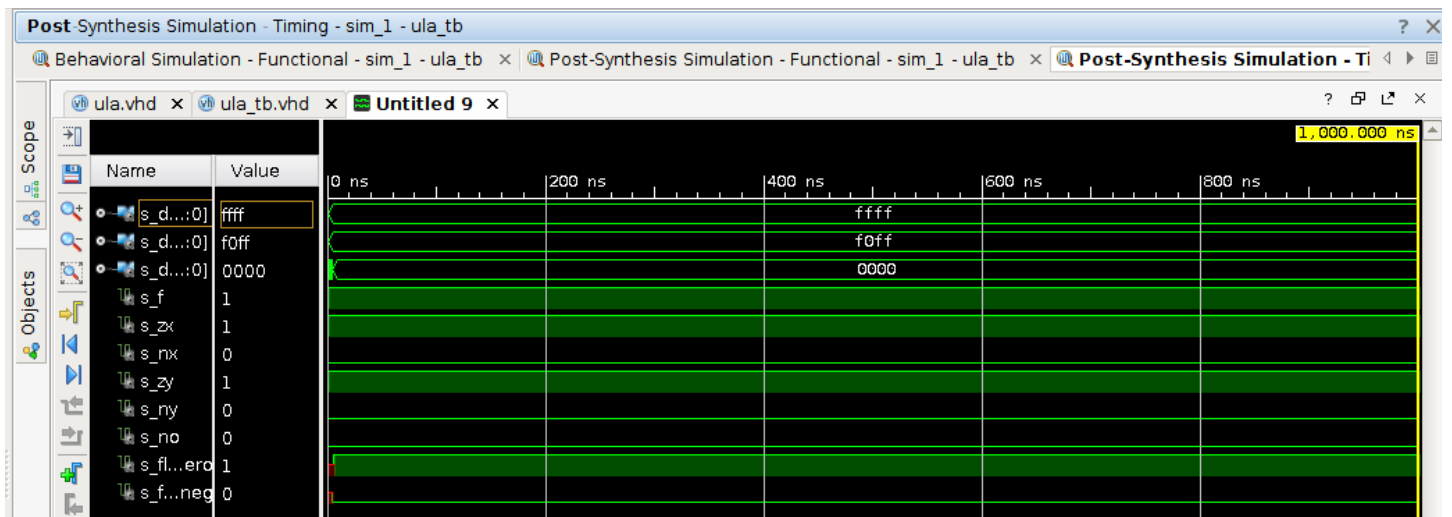
Abaixo seguem as simulações da ULA:



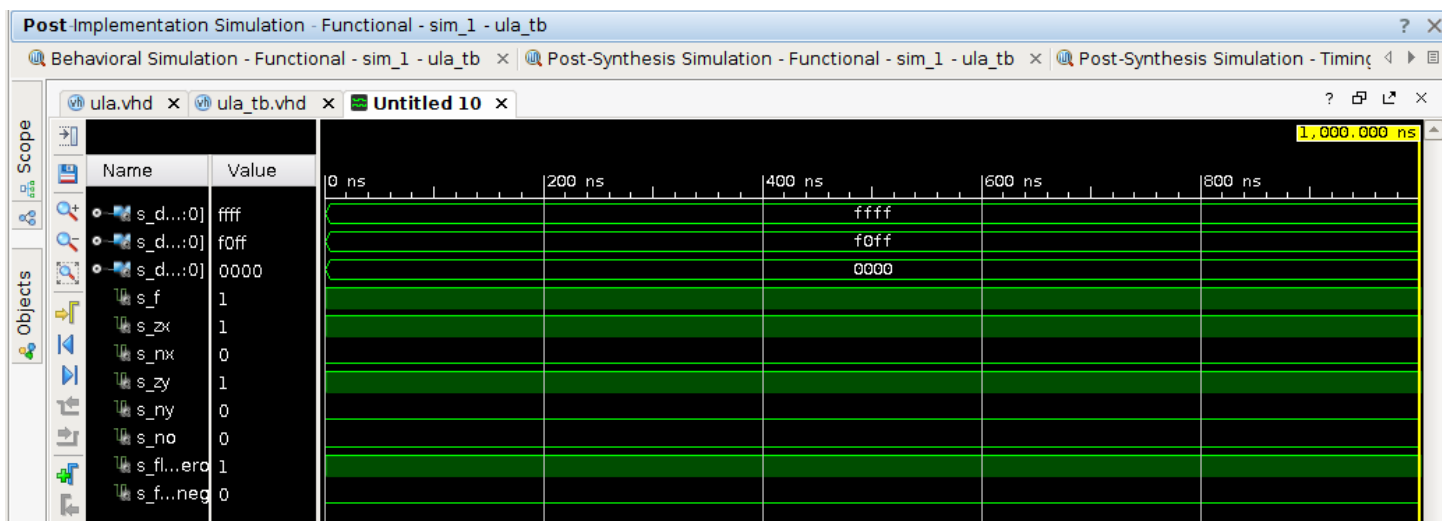
### Simulação de comportamento (Behavioral)



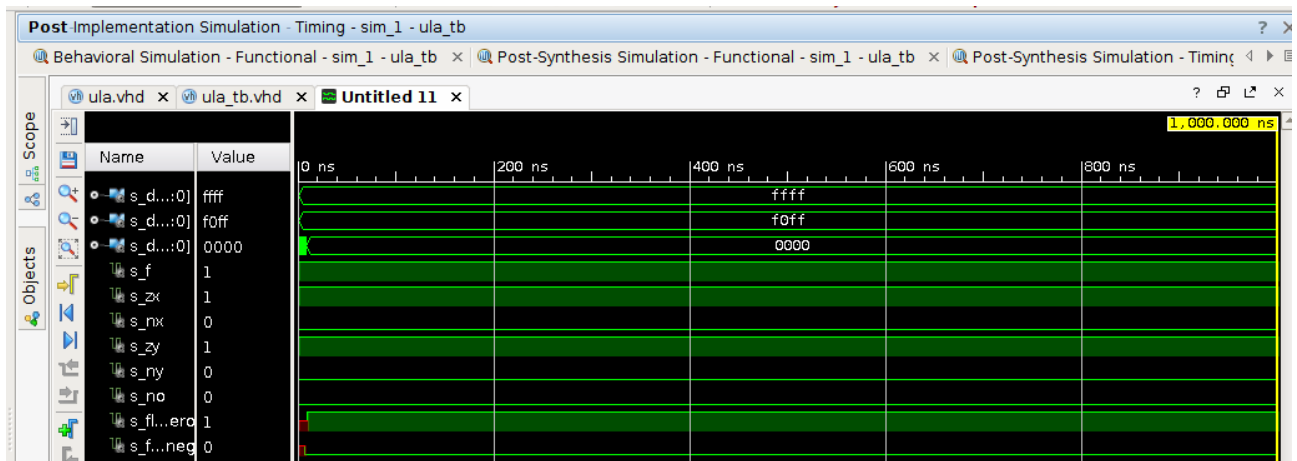
Synthesis Functional Simulation



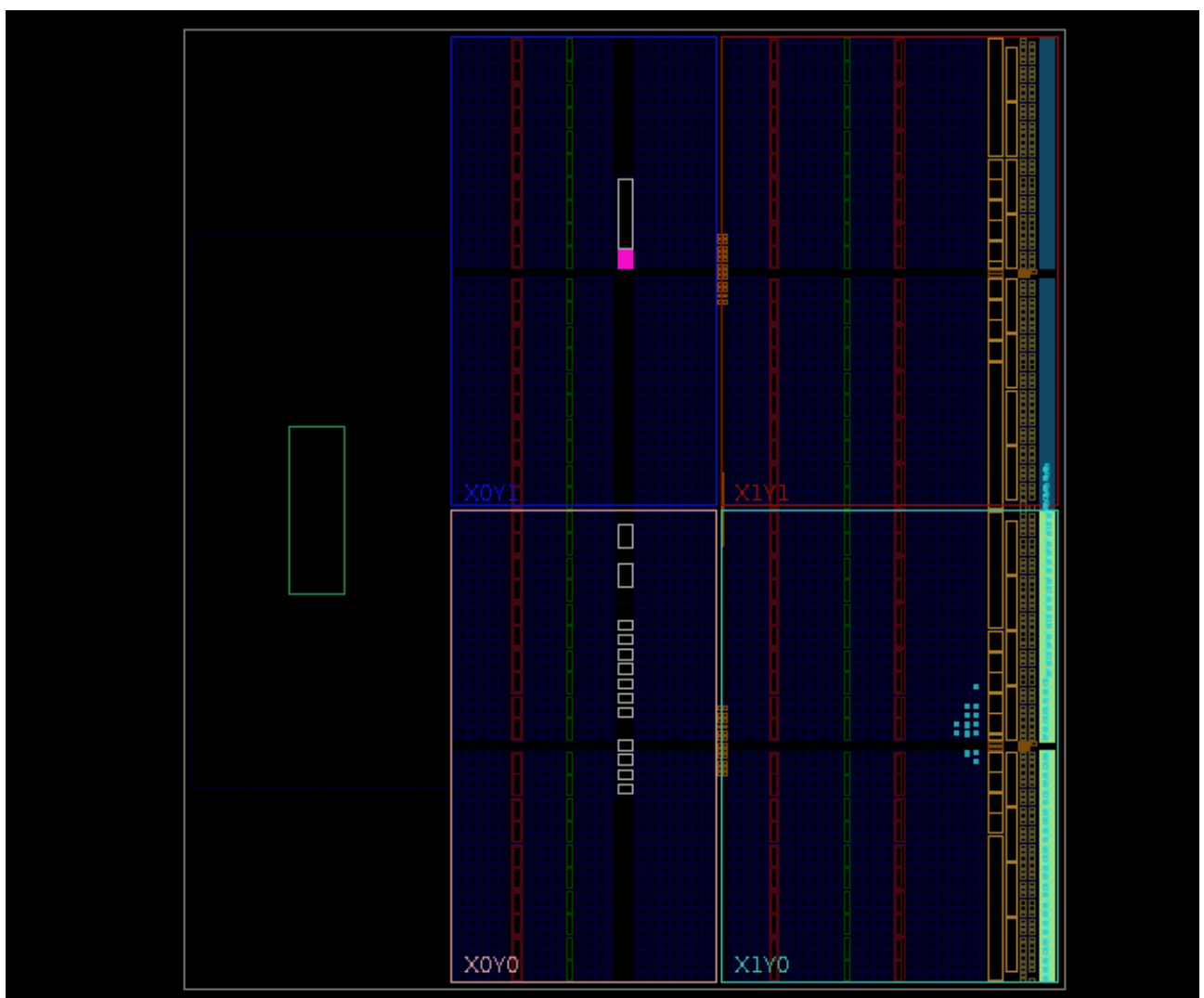
Synthesis Timing Simulation



Implementation Functional Simulation



Implementation Timing Simulation



Implemented Design