Atividade do Capítulo 1

Nome: Mateus Sousa Araújo – **Matrícula:** 374858
Nome: José Wesley Araújo – **Matrícula:** 374855

Portas básicas

Basicamente fizemos todas as portas básicas com um mesmo princípio e mesma ideia. Os códigos em VHDL e o Teste Bench para todas as portas são muito parecidos, com exceção da porta NOT, que possui algumas diferenças.

• OR GATE

A porta OR comum possui 2 entradas e uma saída. Declaramos na entidade duas entradas representadas por “a” e “b”. Para a saída colocamos “s” para a representação de output. Na arquitetura fizemos com que “s” recebesse a operação OR com a entrada “a” e “b”.

Para a simulação realizamos a criação da parte componente, e também realizamos a criação de 2 sinais. O sinal “i_ab” é do tipo vetor que armazena duas posições, cada posição armazena um número do tipo unsigned. A ideia desse sinal é armazenar cada posição das entradas “a” e “b”, fazendo com que possamos percorrer toda a tabela verdade da porta OR. Um sinal “i_s” que vai mapear o saída “s” da porta OR. Em seguida fizemos o mapeamento de cada variável a cada sinal respectivo. Em seguida, na parte de processo, fizemos todos os casos da tabela verdade da porta OR, somando o vetor em cada posição mais um. Essa será a ideia para todas as portas básicas. Em seguida fizemos a simulação.

or_gate.vhd
/home/matt/project_1/project_1.srcs/sources_1/new/or_g

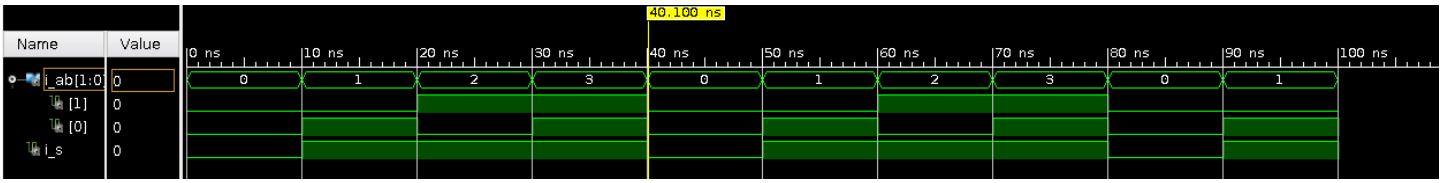
```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity or_gate is
    Port(
        a   :   in std_logic;   --entrada a
        b   :   in std_logic;   --entrada b
        s   :   out std_logic  --saída s
         );
end or_gate;

architecture Behavioral of or_gate is

begin

    s <= a or b;

end Behavioral;
```

Código em VHDL da porta OR

or_tb.vhd
/home/matt/project_1/or_tb.vhd

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;


entity or_tb is

end or_tb;

architecture Behavioral_tb of or_tb is
    component or_gate is
        port(
            a   : in    std_logic;
            b   : in    std_logic;
            s   : out   std_logic
        );
    end component;

    signal i_ab : unsigned (1 downto 0) := "00";
    signal i_s  : std_logic;

begin

    teste:  or_gate port map(
                                a => i_ab(1),
                                b => i_ab(0),
                                s => i_s
                            );
p: process --label

    begin
        wait for 10 ns;
            i_ab <= i_ab + 1;
    end process;

end Behavioral_tb;
```

Teste Bench da porta OR



Simulação da porta OR

• AND GATE

Para a porta AND fizemos a mesma ideia da porta OR. Duas entradas e uma saída declaradas na entidade. Logo depois a operação entre as duas entradas. No teste de simulação também declaramos sinais, os mesmos para a porta OR. Um vetor de 2 posições responsável por "varrer" a tabela verdade da AND e colocar esse resultado na simulação. Os códigos e o Teste Bench seguem abaixo:

and_gate.vhd *

/home/matt/project_2/and_gate.vhd

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
entity and_gate is
  Port (
        a   :   in std_logic;
        b   :   in std_logic;
        s   :   out std_logic
         );
end and_gate;

architecture Behavioral of and_gate is

begin

s <= a and b;

end Behavioral;

```

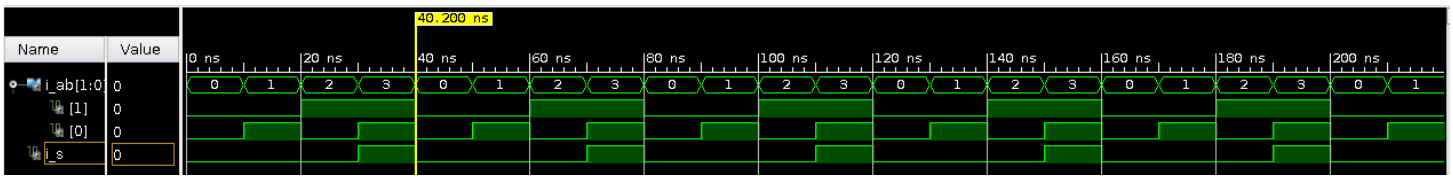
Código em VHDL da porta AND

and_tb.vhd

/home/matt/project_2/and_tb.vhd

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity and_tb is

end and_tb;

architecture Behavioral_tb2 of and_tb is
    component and_gate is
        port(
            a   : in    std_logic;
            b   : in    std_logic;
            s   : out   std_logic
        );
    end component;

    signal i_ab : unsigned (1 downto 0) := "00";
    signal i_s  : std_logic;

begin

    teste:  and_gate port map(
                                a => i_ab(1),
                                b => i_ab(0),
                                s => i_s
                            );
p: process --label

    begin
        wait for 10 ns;
            i_ab <= i_ab + 1;
    end process;

end Behavioral_tb2;
```

Teste Bench da porta AND



Simulação da porta AND

● **NOR GATE**

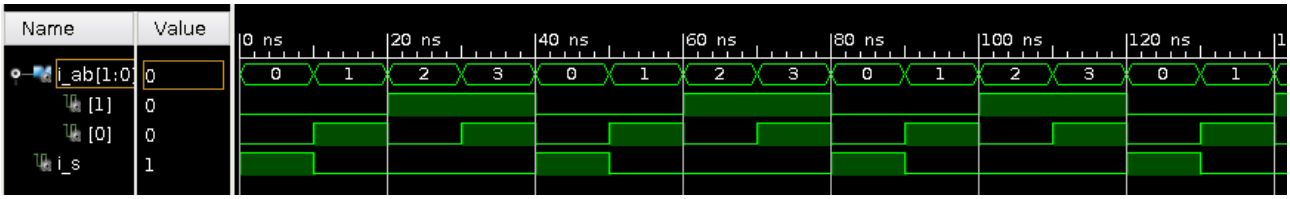
Com a mesma ideia das anteriores, 2 entradas e 1 saída. Teste Bench realizado da mesma forma.

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity nor_gate is
  Port (
        a   : in std_logic;
        b   : in std_logic;
        s   : out std_logic
        );
end nor_gate;

architecture Behavioral of nor_gate is

begin

s <= a nor b;

end Behavioral;
```

Código em VHDL da porta NOR

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity nor_tb is
--  Port ( );
end nor_tb;

architecture Behavioral_tb of nor_tb is

component nor_gate is
        port(
            a   : in    std_logic;
            b   : in    std_logic;
            s   : out   std_logic
        );
    end component;

    signal i_ab : unsigned (1 downto 0) := "00";
    signal i_s  : std_logic;

begin

    teste:  nor_gate port map(
                                a => i_ab(1),
                                b => i_ab(0),
                                s => i_s
                            );
p: process --label

    begin
        wait for 10 ns;
            i_ab <= i_ab + 1;
    end process;

end Behavioral_tb;
```

Teste Bench da porta NOR



Simulação da porta NOR

● XOR GATE

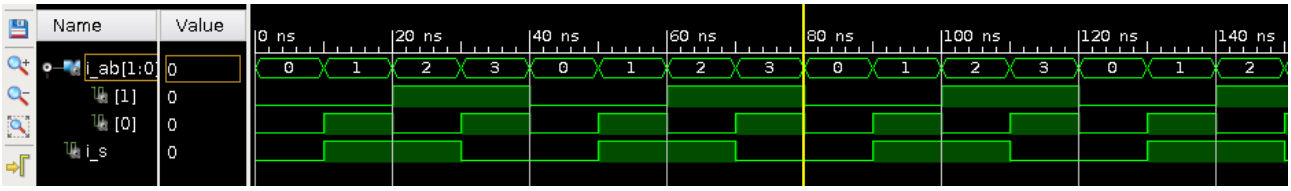
Porta composta por 2 entradas e 1 saída. Da mesma ideia das anteriores temos a criação de dois sinais, fazendo o mapeamento em seguida para a simulação.

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
entity xor_gate is
  Port (
        a   :   in std_logic;
        b   :   in std_logic;
        s   :   out std_logic
        );
end xor_gate;

architecture Behavioral of xor_gate is

begin

s <= a xor b;

end Behavioral;

```

Código em VHDL da porta XOR

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;


entity xor_tb is

end xor_tb;

architecture Behavioral_tb of xor_tb is
    component xor_gate is
        port(
            a   : in    std_logic;
            b   : in    std_logic;
            s   : out   std_logic
        );
    end component;

    signal i_ab : unsigned (1 downto 0) := "00";
    signal i_s  : std_logic;

begin

    teste:  xor_gate port map(
                                a => i_ab(1),
                                b => i_ab(0),
                                s => i_s
                            );
p: process --label

    begin
        wait for 10 ns;
            i_ab <= i_ab + 1;
    end process;

end Behavioral_tb;
```

Teste Bench da porta XOR



Simulação da porta XOR

● **NAND GATE**

Aqui temos a mesma ideia. Duas entradas “a” e “b”, com uma saída “s”. O teste de simulação foi realizado da mesma forma das demais.

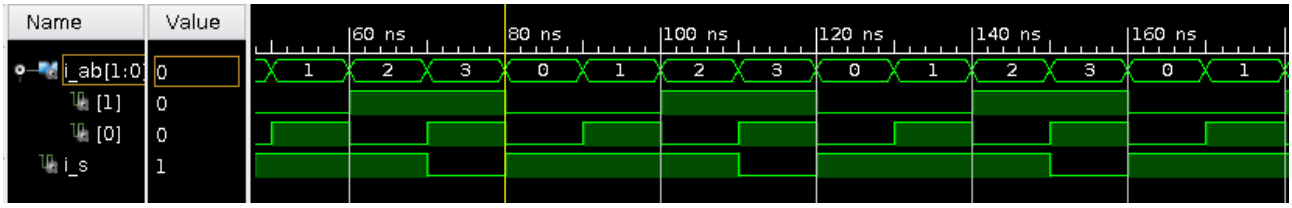
/home/matt/project_5/nand_gate.vhd

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.NUMERIC_STD.ALL;
4  entity nand_gate is
5    Port (
6          a    :    in std_logic;
7          b    :    in std_logic;
8          s    :    out std_logic
9           );
10 end nand_gate;
11
12 architecture Behavioral of nand_gate is
13
14 begin
15
16 s <= a nand b;
17
18
19 end Behavioral;
20
```

Código em VHDL da porta NAND

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.NUMERIC_STD.ALL;
4
5
6  entity nand_tb is
7
8  end nand_tb;
9
10 architecture Behavioral_tb of nand_tb is
11     component nand_gate is
12         port(
13             a   : in    std_logic;
14             b   : in    std_logic;
15             s   : out   std_logic
16         );
17     end component;
18
19     signal i_ab : unsigned (1 downto 0) := "00";
20     signal i_s  : std_logic;
21
22 begin
23
24     teste:  nand_gate port map(
25                                 a => i_ab(1),
26                                 b => i_ab(0),
27                                 s => i_s
28                             );
29 p: process --label
30
31     begin
32         wait for 10 ns;
33             i_ab <= i_ab + 1;
34     end process;
35
36 end Behavioral_tb;
```

Teste Bench da porta NAND



Simulação da porta NAND

● **XNOR GATE**

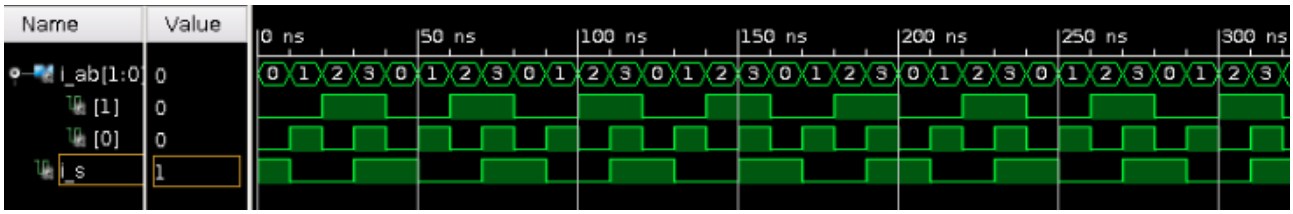
Da mesma forma das anteriores, temos 2 entradas e 1 saída. O teste de simulação foi feito semelhante aos demais.

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity xnor_gate is
  Port (
        a   :   in std_logic;
        b   :   in std_logic;
        s   :   out std_logic
        );
end xnor_gate;

architecture Behavioral of xnor_gate is

begin

s <= a xnor b;

end Behavioral;
```

Código em VHDL da porta XNOR

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity xnor_tb is
--  Port ( );
end xnor_tb;

architecture Behavioral_tb of xnor_tb is

component xnor_gate is
        port(
            a   : in    std_logic;
            b   : in    std_logic;
            s   : out   std_logic
        );
    end component;

    signal i_ab : unsigned (1 downto 0) := "00";
    signal i_s  : std_logic;

begin

    teste:  xnor_gate port map(
                                a => i_ab(1),
                                b => i_ab(0),
                                s => i_s
                            );
p: process --label

    begin
        wait for 10 ns;
            i_ab <= i_ab + 1;
    end process;

end Behavioral_tb;
```

Teste Bench da porta XNOR



Simulação da porta XNOR

● **NOT GATE**

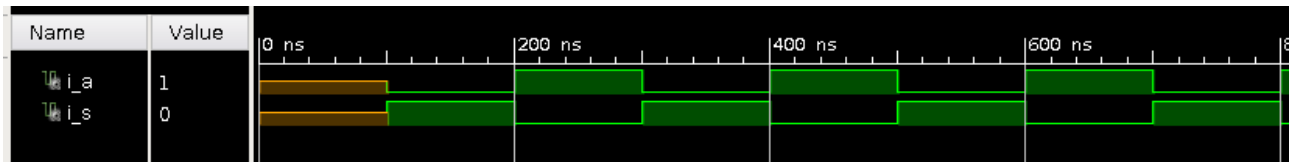
Aqui fizemos apenas uma entrada “a” e uma saída “s”. Precisamos fazer um sinal do tipo std_logic para forçar a entrada a um número. O sinal “i_a” dessa vez não é mais do tipo vetor, pois teremos apenas 2 casos possíveis, nível lógico alto – 1 – e nível lógico baixo – 0. No processo fizemos que “i_a” recebesse dois níveis diferentes, dessa forma invertendo o sinal na saída. Percebemos um pequeno atraso de sinal na simulação nas entradas.

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity not_gate is
  Port (
        a   :   in std_logic; -- entrada a;
        s   :   out std_logic -- saida s;
         );
end not_gate;

architecture Behavioral of not_gate is

begin

s <= not a; -- saida recebe "a" negado;

end Behavioral;
```

Código em VHDL da porta NOT

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity not_tb is
end not_tb;

architecture Behavioral_tb of not_tb is
    component not_gate is
        port(
            a   : in    std_logic; -- entrada a;
            s   : out   std_logic  -- entrada s;
        );
    end component;

    signal i_a  : std_logic; -- sinal do tipo std_logic para forçar a entrada;
    signal i_s  : std_logic; -- sinal de saida;

begin

    teste:  not_gate port map( -- mapeamento de pinos
                               a => i_a,
                               s => i_s
                             );
p: process --label

    begin
        wait for 100 ns;
            i_a <= '0'; -- i_a recebe 0 (nivel lógico baixo);
        wait for 100 ns;
            i_a <= '1'; -- i_a recebe 1 (nivel lógico alto);
    end process;

end Behavioral_tb;
```

Teste Bench da porta NOT



Simulação da porta NOT

Portas básicas array

● OR16 GATE

Nesta porta, utilizamos duas entradas (**a** e **b**) de 16 cada e uma saída (**s**) com 16 bits. O código é bem semelhante ao da porta OR de duas entradas, porém, neste caso, temos dois vetores std_logic_vector de 16 posições, representados por **a** e **b**, além do vetor de saída **s** com 16 posições.

Abaixo vemos a figura que demonstra o código da porta lógica em VHDL.

```
1
2   library IEEE;
3   use IEEE.STD_LOGIC_1164.ALL;
4   use IEEE.NUMERIC_STD.ALL;
5   entity OR16 is
6      Port (
7            a : in std_logic_vector(15 downto 0);
8            b : in std_logic_vector(15 downto 0);
9            s : out std_logic_vector(15 downto 0)
10           );
11  end OR16;
12
13  architecture Behavioral of OR16 is
14
15  begin
16   s <= a or b;
17
18  end Behavioral;
19
```

***Figura 1**- código OR16 bits.*

OR16- TESTE BENCH

Para fazer o teste bench, criamos uma nova entidade chamada OR16_tb. Em seguida, declaramos os componentes de entrada **a** e **b**, e o de saída **s**; ambos std_logic_vector. Após isso, criamos os sinais que foram parte da simulação. Para facilitar a visualização da simulação, deixamos a entrada com valor i_**a** = **“F000”** em hexadecimal e i_**b**= **“0000”,** também em hexadecimal. Logo em seguida, mapeamos as portas da entidade OR16, como vemos na **figura2**, abaixo. Após termos feito tudo isso, iniciamos um process. No início utilizamos o comando **wait** para retardar a operação a seguir em 100 ns. A nível de simulação, variamos apenas o sinal i_b. Para facilitar, fazemos um cast na incrementação do sinal **i_b** em função do seu tipo **std_logic_vector** . Desta forma, uma operação OR bit a bit será feito entre as entradas. A seguir temos um exemplo de como seria esta operação:

a = **1111 0000 0000 0000** e
b = **0000 0000 0000 0001,** logo a saída **s** será:

1111 0000 0000 0000
0000 0000 0000 0001
s = 1111 0000 0000 0001

Portanto, a saída do exemplo dado será **s = F001** em hexadecimal.
Abaixo, vemos o código e os resultados da simulação, nas figuras 2 e figura 3, respectivamete.

```
1
2   library IEEE;
3   use IEEE.STD_LOGIC_1164.ALL;
4   use IEEE.NUMERIC_STD.ALL;
5   entity OR16_tb is
6   --  Port ( );
7   end OR16_tb;
8
9   architecture Behavioral_tb of OR16_tb is
10      component OR16 is
11          port (
12              a : in std_logic_vector (15 downto 0);
13              b : in std_logic_vector (15 downto 0);
14              s : out std_logic_vector(15 downto 0)
15          );
16       end component;
17       -- iniciando os valores do sinal i_a com F000 em hexal, para facilitar a visualização no simulação
18      signal i_a : std_logic_vector (15 downto 0) := X"F000" ;
19      signal i_b : std_logic_vector (15 downto 0) := X"0000" ;
20      signal i_s : std_logic_vector (15 downto 0);
21  begin
22      TESTE: OR16  port map ( a => i_a , b => i_b , s => i_s);
23      P: process
24              begin
25                  wait for 100ns;
26                  -- cria um cast para sempre converter o valor do sinal so std i_b + 1 para i_b
27                  i_b <= std_logic_vector (unsigned(i_b) +1);  -- desta forma o i_b é incrementado
28      end process;
29  end Behavioral_tb;
```

Figura 2- código or16_tb

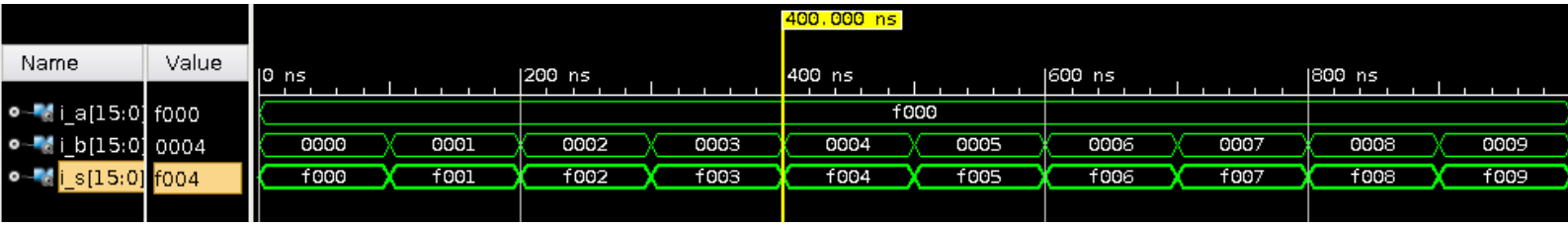


Figura 3 – simulação or16

•AND16 GATE

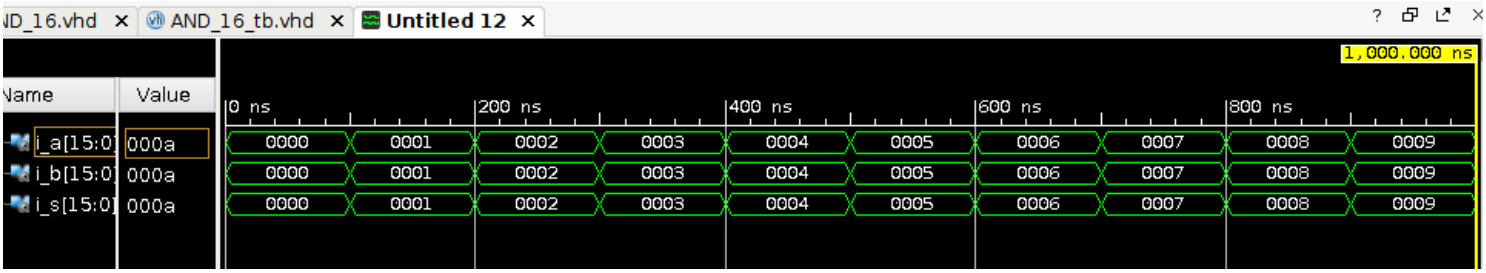
O código da porta ANDGATE é semelhante ao da porta OR. Inicialmente declaramos entradas **a** e **b** com 16 bits cada, e a saída **s,** também com 16 bits. Após isso, fazemos uma operação AND bit a bit entre as duas entradas e armazenamos em S. Figura abaixo ilustra isso.

```vhdl


library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
use IEEE.NUMERIC_STD.ALL;

entity AND_16 is
 PORT(
       a : in std_logic_vector(15 downto 0);
       b : in std_logic_vector(15 downto 0);
       s : out std_logic_vector(15 downto 0)
       );
end AND_16;

architecture Behavioral of AND_16 is

begin

    s <= a and b;

end Behavioral;

```

AND16 GATE – TESTE BENCH

No teste bench da AND16 criamos uma entidade – AND16_tb – e, após isso, os componentes da AND_16 : **a,b e s ,** ambos de 16 bits. Logo em seguida declaramos os sinais que serão utilizados. O sinal **i_a** e o sinal **i_b** serão inicializados com **0000** em hexadecimal para facilitar a visualização da simulação. Em seguida, as portas são mapeadas em AND16 e iniciasse o processo . Como no caso anterior, é feito um cast nos sinais para facilitar a incrementação. Como **i_a** e **i_b** serão iguais, pois sempre serão incrementados em quantidades iguais, logo a saída **i_s** deve ser igual a **i_a e i_b.** Logo abaixo, comprovamos esta afirmação com a simulação e o código do teste.

```vhdl


library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
entity AND_16_tb is
--  Port ( );
end AND_16_tb;
architecture Behavioral_tb of AND_16_tb is
   component AND_16 is

       Port (
            a: in std_logic_vector(15 downto 0);         -- criando vetor entrada em a para os barramentos a, b e s
            b: in std_logic_vector(15 downto 0);
            s: out std_logic_vector(15 downto 0));
      end component;
   signal i_a : std_logic_vector (15 downto 0) := X"0000"; -- i_a iniciado com 0000 EM HEXA pra testar valores de AND com b
   signal i_b : std_logic_vector (15 downto 0) := X"0000";-- i_b iniciado com 0000 EM HEXA
   signal i_s : std_logic_vector (15 downto 0)         ; -- saída
begin
    TESTE: AND_16 port map (   a  => i_a ( 15 downto 0) ,  b  => i_b ( 15 downto 0), s  => i_s );
    -- iniciando o processo para incrementar o vetor e notas as diferentes saídas de s partindo de a or b.
    P : process
        begin
                wait for 100 ns;
                -- NO CASO, A E B SEMPRE TÊM VALORES IGUAIS, PORTANTO, A SAIDA É IGUAL A ELES.
                i_b <= std_logic_vector( unsigned (i_b) + 1);  -- CAST USADO PRA IGUALAR O TIPO DE SAÍDA
                 i_a <= std_logic_vector( unsigned (i_a) + 1); -- I_A  é do tipo std_logic e esta sendo incrementado
        end process p;                                          -- só é possível pelo cast
end Behavioral_tb;

```



•NOT16 GATE

O código desta porta é semelhante ao da porta **not** de uma entrada, porém, esta porta age sobre um conjunto de 16 bits. Inicialmente, no código, declaramos uma entrada **a** de 16 bits e uma saída **s**, também de 16 bits. Neste caso, dado um vetor de entrada, a saída será a negação bit a bit desta entrada. Abaixo vemos o código da entidade NOT16:

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.NUMERIC_STD.ALL;
4
5  entity NOT16 is
6      Port (
7          a : in std_logic_vector(15 downto 0);
8          s : out std_logic_vector(15 downto 0)
9      );
10 end NOT16;
11
12 architecture Behavioral of NOT16 is
13
14 begin
15         -- Declaração da porta que será usada
16     s <= not a;
17
18 end Behavioral;
19
```

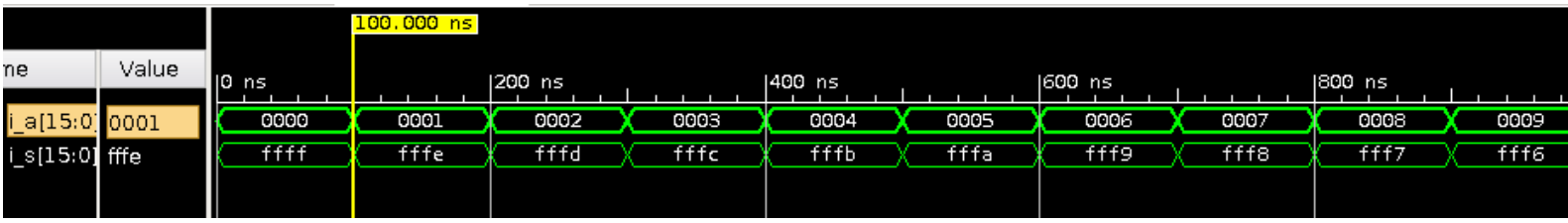
NOT16 teste bench

Para fazermos o teste bench da porta NOT16, criamos uma entidade chamada NOT16_tb e declaramos o componente de entrada **a** e de saída **s,** ambos do tipo **std_logic_vector** e tamanho 16 bits. Em seguida declaramos sinais **i_a** e **i_s** . Neste caso, inicializamos o **i_a** com zero para facilitar a visualização da simulação. Após isso iniciamos um processo e damos um comando **wait** para abortar o processo durante 100 ns. Em razão do sinal **i_a** ser do tipo **std_logic_vector,** temos de fazer um cast de tipos para facilitar a incrementação.

Portanto, caso **a = 0000 0000 0000 0000, s = 1111 1111 1111 1111.**

O código e a simulação são vistos abaixo:

```vhdl

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity NOT16_tb is
--  Port ( );
end NOT16_tb;

architecture Behavioral_tb of NOT16_tb is
    component NOT16 is
       Port(
            -- portas que serao usadas
            a : in std_logic_vector(15 downto 0);
            s : out std_logic_vector(15 downto 0)
       );
      end component;
      signal i_a :std_logic_vector(15 downto 0) := X"0000"; -- inicializa o vetor com zero
      signal i_s :std_logic_vector (15 downto 0);
begin
     TESTE: NOT16 port map(  -- mapeando de NOT16 para NOT16_TB
                           a => i_a,
                           s => i_s
                          );
     P : process
         begin
              wait for 100 ns; -- Espera 100ns, muda i_A pra unsigned e adiciona um depois transforma em std_logic
             i_a <= std_logic_vector ( unsigned( i_a) + 1); -- Desta forma o vetor ficará com todas as possibilidades de a
        end process;
end Behavioral_tb;

```



- **OR8_1 GATE**

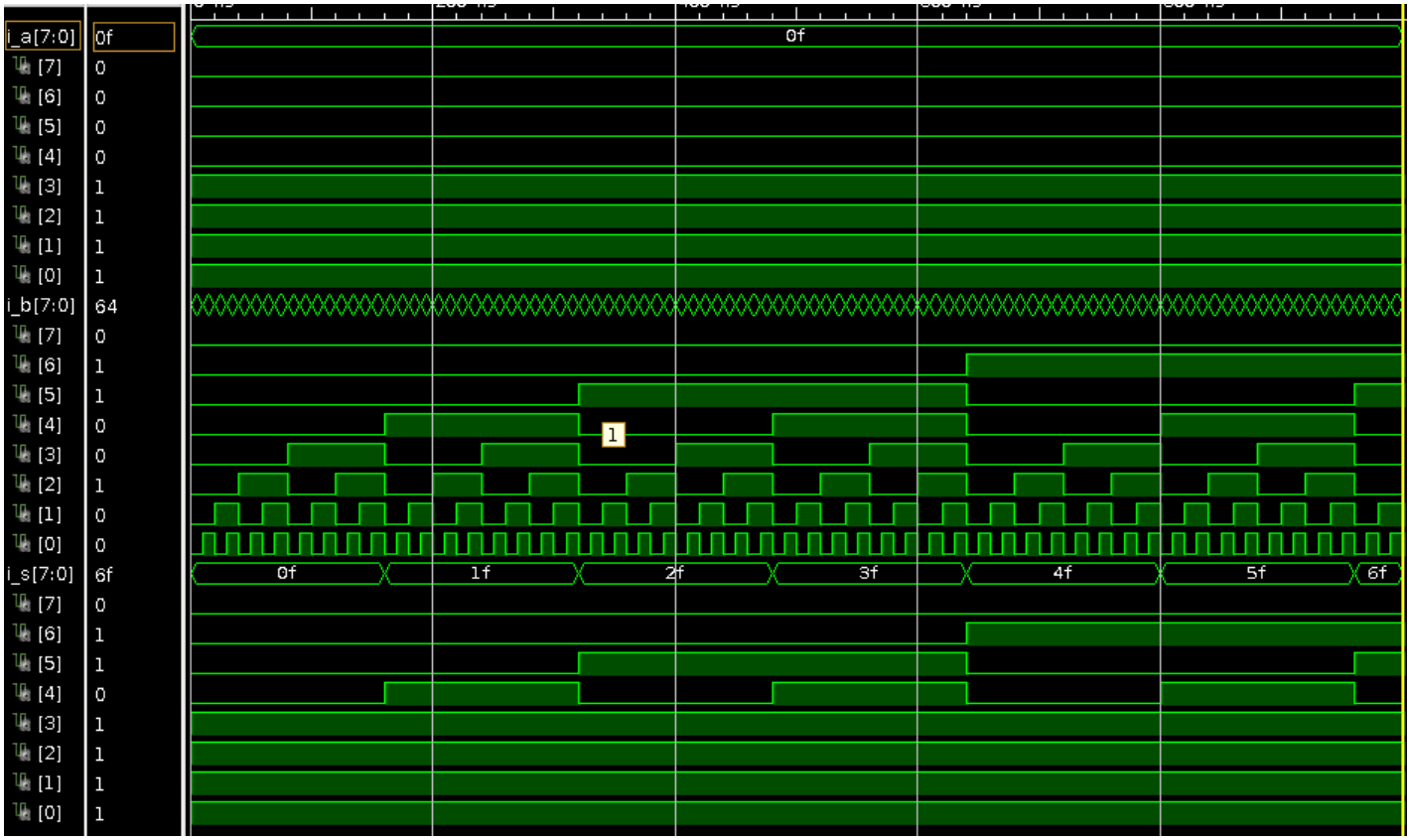
O desenvolvimento do projeto desta porta é igual ao da porta OR16, exceto no número de bits das entradas. Sendo assim, o processo é o mesmo visto na OR16. São declaradas duas entradas **a e b ,** e uma saída **s.** Todas as entradas são **std_logic_vector** de 8 bits cada. Após declarar a arquitetura, fazemos um OR bit a bit das duas entradas; a saída recebe o resultado. A figura abaixo, mostra o código da Entidade OR8.

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
use IEEE.NUMERIC_STD.ALL;

entity OR_8 is
    -- DECLARA VETORES DE ENTRADA A E B
   Port ( a : in std_logic_vector(7 downto 0);
          b : in std_logic_vector(7 downto 0);
          s : out std_logic_vector(7 downto 0)
          );

end OR_8;

architecture Behavioral of OR_8 is

begin
    -- CODIGO DA OR_8_1 PARA AS SAIDAS DE S E ENTRADAS DE A E B
    s(0) <= a(0) or b(0);
    s(1) <= a(1) or b(1);
    s(2) <= a(2) or b(2);
    s(3) <= a(3) or b(3);
    s(4) <= a(4) or b(4);
    s(5) <= a(5) or b(5);
    s(6) <= a(6) or b(6);
    s(7) <= a(7) or b(7);

end Behavioral;
```

OR8_1 – TESTE BENCH

No teste bench da porta OR8 criamos a entidade OR8_tb. Declamaramos as entradas **a** e **b,** ambas de 8 bits, e uma saída **s** também de 8 bits. Logo após isso, declaramos sinais que serão usadas na implementação: **i_a , i_b e i_s.** Inicializamos sinal **i_a** com o valor **i_a = 0F** e o sinal **i_b = 00.** Depois disso, mapeamos as portas da OR8_1. Como o **i_a** foi inicializado como **0F e**m hexadecimal , logo a saída sempre terá o segundo byte igual a F, e o primeiro dependerá de i_b, em razão do OR bit a bit da operação. Depois de mapear, criamos um processo e cada 20 ns criamos um incrementamos b em uma unidade. O resultado da simulação é mostrado abaixo.

```
7     --  Port ( );
8     end OR_8_tb;
9     architecture Behavioral_tb of OR_8_tb is
10       component OR_8 is
11           Port (
12                a: in std_logic_vector(7 downto 0);         -- criando vetor entrada em a para os barramentos a, b e s
13                b: in std_logic_vector(7 downto 0);
14                s: out std_logic_vector(7 downto 0));
15          end component;
16       signal i_a : unsigned (7 downto 0) := X"0F"; -- i_a iniciado com 0000 1111 pra testar valores de or com b
17       signal i_b : unsigned (7 downto 0) := X"00";-- i_b iniciado com 0000 0000 pra testar valores
18       signal i_s : unsigned (7 downto 0); -- saída
19    begin
20        TESTE: OR_8 port map ( a(0) => i_a(0),      -- mapeando os valores de a em i_a e i_b em b
21                               a(1) => i_a(1),
22                               a(2) => i_a(2),
23                               a(3) => i_a(3),
24                               a(4) => i_a(4),
25                               a(5) => i_a(5),
26                               a(6) => i_a(6),
27                               a(7) => i_a(7),
28                               b(0) => i_b(0),
29                               b(1) => i_b(1),
30                               b(2) => i_b(2),
31                               b(3) => i_b(3),
32                               b(4) => i_b(4),
33                               b(5) => i_b(5),
34                               b(6) => i_b(6),
35                               b(7) => i_b(7),
36                               s(0) => i_s(0),
37                               s(1) => i_s(1),
38                               s(2) => i_s(2),
39                               s(3) => i_s(3),
40                               s(4) => i_s(4),
41                               s(5) => i_s(5),
42                               s(6) => i_s(6),
43                               s(7) => i_s(7)
44                               );
45        -- iniciando o processo para incrementar o vetor e notas as diferentes saídas de s partindo de a or b.
46        P : process
47            begin
48                    wait for 10 ns;
49                    i_b <= i_b + 1;
50            end process;
51    end Behavioral_tb;
```


i_a[7:0]
0f
[7]
[6]
[5]
[4]
[3]
[2]
[1]
[0]
i_b[7:0]
64
[7]
[6]
[5]
[4]
[3]
[2]
[1]
[0]
i_s[7:0]
6f
[7]
[6]
[5]
[4]
[3]
[2]
[1]
[0]
0f
1f
2f
3f
4f
5f
6f
1

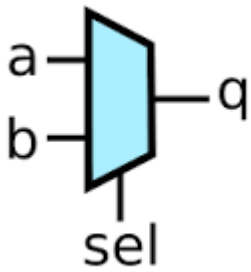
Multiplexadores e Demultiplexadores

Para os multiplexadores e demultiplexadores utilizamos algumas maneiras diferentes de implementação de cada um.

● MUX2_1

Para a construção do MUX2_1, declaramos 2 entradas (“in1” e “in2”), uma chave de seleção (“sel”) e uma saída (“output”) na entidade. Temos nesse caso um MUX com 2 entradas, uma chave de seleção e um canal de saída. Na arquitetura implementamos uma lógica que dá certo para a seleção das entradas nas saídas, no qual é:

output <= (in1 and not sel) OR (in2 and sel);



Se fizermos todos os testes possíveis, percebemos que essa operação booleana acima faz com que a saída esteja de acordo com a seleção de MUX. Por exemplo, se tivermos a entrada 1 igual a 0 e a entrada 2 igual a 1 e a chave de seleção igual 1, significa que escolheremos a segunda entrada para a nossa saída. Se fizermos o teste em pseudocódigo:

in1 = 0;
in2 = 1;
sel = 1;

Dessa forma a saída será 1.

output <= (0 ^ 0) ou (1 ^ 1) = 0 ou 1 = 1;

Com esse exemplo provamos nossa expressão. Os outros casos também se verificam. De acordo com a figura do MUX2_1 acima, ficariam:

a = 0;
b = 1;
sel = 1;
q = 1;

No teste bench fizemos a criação de 3 sinais e atribuímos valores a eles. No nosso exemplo as entradas estão em 1 e 0, in1 e in2, respectivamente. E a seleção em 0. Nesse caso ele vai escolher a entrada 2 para a saída. No processo ficamos alternando o sinal de seleção para melhorar a visualização da simulação.

Abaixo estão os códigos para a implementação desse MUX com a simulação:

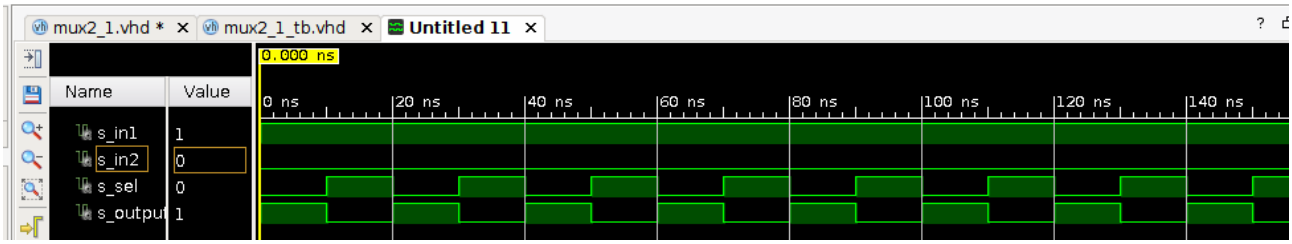
/home/matt/mux_2_1/mux2_1.vhd

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity mux2_1 is
  Port (
        in1, in2     :      in std_logic; -- 2 entradas para o mux;
        sel          :      in std_logic; -- 1 seleção para o mux;
        output       :      out std_logic -- uma saída para o mux;
        );
end mux2_1;

architecture Behavioral of mux2_1 is

begin

output <= (in1 and not sel) OR (in2 and sel); -- operação para qual entrada irá para a saída;

end Behavioral;

```

Código do MUX2_1 em VHDL

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity mux2_1_tb is
end mux2_1_tb;

architecture Behavioral_tb of mux2_1_tb is
    component mux2_1 is
        port(
           in1, in2     :      in std_logic; -- 2 entradas para o mux;
           sel          :      in std_logic; -- 1 seleção para o mux;
           output       :      out std_logic -- uma saída para o mux;
        );
    end component;


    --INPUTS
    signal s_in1         : std_logic := '1'; -- fixamos a 1ª entrada do mux em 1;
    signal s_in2         : std_logic := '0'; -- fixamos a 2ª entrada do mux em 0;
    signal s_sel         : std_logic := '0'; -- fixamos a chave de seleção em 0;

    --OUTPUTS
    signal s_output    : std_logic;        -- sinal para a saída;

begin

    teste:  mux2_1 port map(
                              in1 => s_in1, -- entrada 1 atribui seu valor ao sinal de entrada 1;
                              in2 => s_in2, -- entrada 2 de seleção atribui seu valor ao sinal de seleção 2;
                              sel => s_sel, -- saída 1 atribui seu valor ao sinal de saída 1;
                              output => s_output   -- saída 2 atribui seu valor ao sinal de saída 2;
                             );
p: process --label

    begin
        wait for 10 ns;
            s_sel <= not s_sel; -- sinal de seleção fica alternando entre nível lógico alto e baixo;
    end process p;

end Behavioral_tb;
```

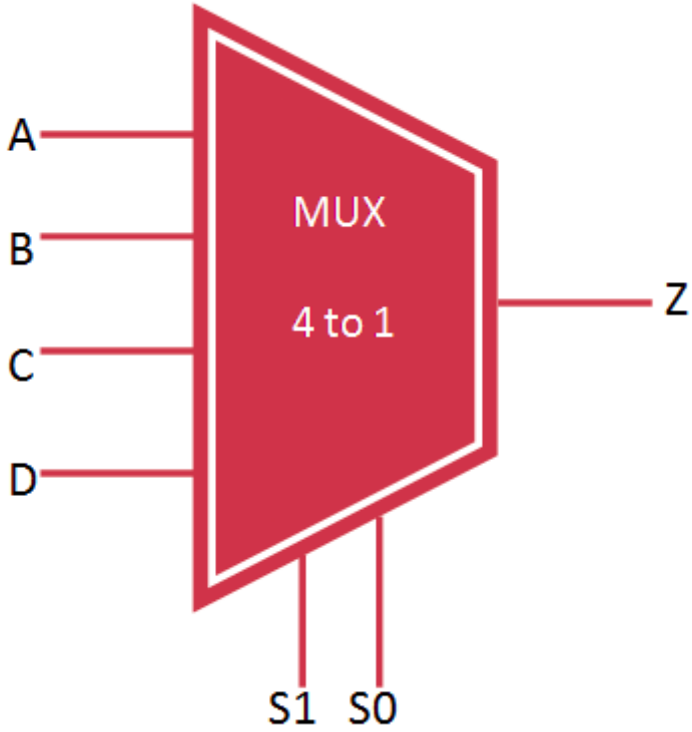
Teste Bench do MUX2_1 em VHDL



Simulação do MUX2_1

● MUX4_1

Aqui temos um MUX com 4 entradas e uma saída. Um MUX desse tipo pode ser representado por:



Representamos nossas entradas, seleções e saídas na entidade por:

Entradas : “a”, “b”, “c” e “d”;
Seleção : “s0” e “s1”;
Saída : “output”.

No código VHDL decidimos fazer um sinal de controle, que receberá as chaves de seleção “s0” e “s1”. Fizemos isso para um controle maior usando o comando “with” “select”, dessa forma passando por cada caso da chave de seleção estar em “00”, “01”, “10” e “11”. Se o controle, por exemplo, estiver em “01”, a entrada “b” irá para o canal de saída e assim por diante.

Na simulação criamos 4 sinais de entrada e um sinal de saída. Cada sinal será mapeado de acordo com o projeto em código em VHDL, porém, fixamos os valores dos sinais de entrada, como por exemplo, fixamos o sinal a (s_A) para 0, o s_B para 1, s_C para 0 e o s_D para 1. Criamos um sinal de controle do tipo array e que guarda valores do tipo unsigned.

No processo fizemos a incrementação da seleção sempre em +1, para que pudéssemos percorrer todos os casos possíveis de seleção. Abaixo estão os códigos em VHDL, a simulação e o teste bench.

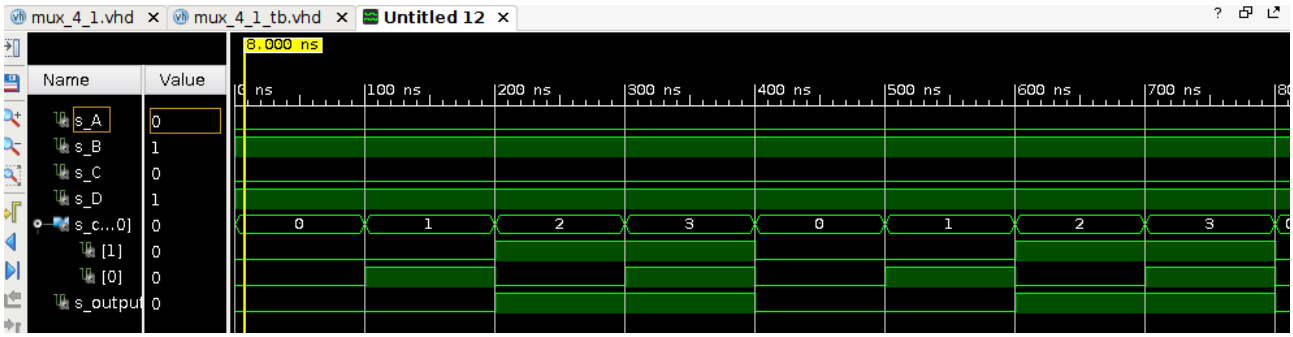
```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity mux_4_1 is
  Port (
        a,b,c,d   :   in std_logic; -- entradas a,b,c e d;
        s0, s1    :   in std_logic; -- chaves de seleção s0 e s1;
        output    :   out std_logic );  -- saída;
end mux_4_1;

architecture Behavioral of mux_4_1 is

signal controle: unsigned(1 downto 0); -- sinal que vai guardar as informações das chaves de seleção;

begin

    controle <= s0 & s1;    -- controle recebe chaves de seleção (s0 e s1);

    with controle select
    output <= a when "00", -- saída recebe "a" quando controle estiver em "00";
              b when "01", -- saída recebe "b" quando controle estiver em "01";
              c when "10", -- saída recebe "c" quando controle estiver em "10";
              d when others; -- saída recebe "d" quando ocorrer o contrário;
end Behavioral;
```

Código do MUX4_1 em VHDL

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity mux_4_1_tb is
--  Port ( );
end mux_4_1_tb;

architecture Behavioral_tb of mux_4_1_tb is
    component mux_4_1 is
        port(
                a,b,c,d         :   in std_logic; -- entradas a,b,c e d;
                s0, s1          :   in std_logic; -- chaves de seleção s0 e s1;
                output          :   out std_logic -- saída;
            );
    end component;

--Inputs
signal s_A  : std_logic := '0'; -- sinal lógico de entrada A;
signal s_B  : std_logic := '1'; -- sinal lógico de entrada B;
signal s_C  : std_logic := '0'; -- sinal lógico de entrada C;
signal s_D  : std_logic := '1'; -- sinal lógico de entrada D;
signal s_controle : unsigned (1 downto 0) := "00"; -- sinal de controle que irá guardar o valor de seleção;

--Outputs
signal s_output : std_logic; -- sinal lógico de saida "output";

begin

-- Instantiate the Unit Under Test (UUT)
   uut: mux_4_1 port map(
          a => s_A, -- a atribui seu valor ao sinal A;
          b => s_B, -- b atribui seu valor ao sinal B;
          c => s_C, -- c atribui seu valor ao sinal C;
          d => s_D, -- d atribui seu valor ao sinal D;
          output => s_output, -- "output" atribui seu valor ao sinal output;
          s0 => s_controle(0), -- bit menos significativo do sinal receberá o valor de s0 na 1ª posição;
          s1 => s_controle(1)  -- bit mais significativo do sinal receberá o valor de s1 na 2ª posição;

        );

-- Stimulus process
   stim_proc: process
```

```vhdl
begin
      -- hold reset state for 100 ns.
    wait for 100 ns;

    s_controle <= s_controle + 1; -- controle incrementará, passando por todos os casos de seleção;

    end process stim_proc;

end Behavioral_tb;
```

Teste Bench do MUX4_1 em VHDL



Simulação do MUX4_1

● **DEMUX1_2**

Nesse caso temos um DEMUX com uma entrada e 2 saídas. Para fazer esse projeto precisamos declarar 4 variáveis na entidade. Criamos 2 variáveis de entrada, uma entrada do DEMUX (“demu_in”) e a seleção (“sel”). Criamos 2 variáveis de saída, “out1” para a saída 1 e “out2” para a saída 2. Fizemos uma operação básica na arquitetura para que o sinal de entrada saísse no canal de saída desejado. A operação levou mais ou menos a mesma ideia do MUX2_1, vamos pegar como exemplo:

demu_in = 1;
sel = 0;
out1 = 1;
out2 = 0;

Escolhemos o 1º canal de seleção de saída para a entrada do DEMUX sair, logo “out1” ficará em 1 e “out2” ficará em zero. Para os demais casos também ocorrerá o mesmo.

Aplicamos a operação lógica básica:

out1 <= demu_in and (not sel); -- saída do out1
out2 <= demu_in and sel; -- saída do out2

out1 = 1 ^ 1 = 1;
out2 = 1 ^ 0 = 0;

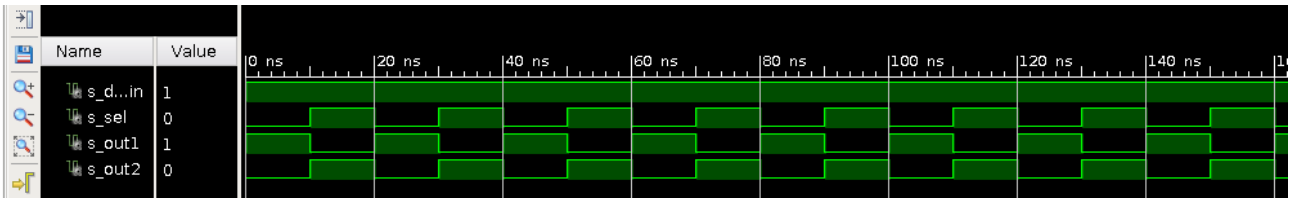
No teste de simulação criamos 4 sinais, cada um relativo as entradas e saídas do DEMUX. Para o sinal de entrada e seleção fixamos alguns valores, como por exemplo, fixamos o sinal de entrada do DEMUX (s_demu_in) em 1 e a seleção (s_sel) em 0. No processo, colocamos para a chave de seleção ficar alternando para passar por todos os valores possíveis de seleção. Abaixo se encontra o código, o teste bench e a simulação.

```
 1  library IEEE;
 2  use IEEE.STD_LOGIC_1164.ALL;
 3  use IEEE.NUMERIC_STD.ALL;
 4
 5  entity demux_1_2 is
 6    Port (
 7          demu_in :   in std_logic;    -- entrada do demux;
 8          sel     :   in std_logic;    -- chave de seleção;
 9          out1    :   out std_logic;   -- output 1;
10          out2    :   out std_logic    -- output 2;
11          );
12  end demux_1_2;
13
14  architecture Behavioral of demux_1_2 is
15
16  begin
17
18  out1 <= demu_in and (not sel); -- saída do out1
19  out2 <= demu_in and sel;       -- saída do out2
20
21  end Behavioral;
22
```

Código do DEMUX1_2 em VHDL

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity demux_1_2_tb is
end demux_1_2_tb;

architecture Behavioral_tb of demux_1_2_tb is
    component demux_1_2 is
        port(
           demu_in :   in std_logic;   -- entrada do demux;
           sel     :   in std_logic;   -- chave de seleção;
           out1    :   out std_logic;  -- output 1;
           out2    :   out std_logic   -- output 2;
        );
    end component;

    signal s_demu_in : std_logic := '1'; -- fixamos a entrada do demux em 1;
    signal s_sel     : std_logic := '0'; -- fixamos a chave de seleção em 0;
    signal s_out1    : std_logic;        -- sinal para a saída 1;
    signal s_out2    : std_logic;        -- sinal para a saída 2;
begin

    teste:  demux_1_2 port map(
                               demu_in => s_demu_in, -- entrada atribui seu valor ao sinal de entrada;
                               sel => s_sel,         -- entrada de seleção atribui seu valor ao sinal de seleção;
                               out1 => s_out1,       -- saída 1 atribui seu valor ao sinal de saída 1;
                               out2 => s_out2        -- saída 2 atribui seu valor ao sinal de saída 2;
                              );
p: process --label

    begin
        wait for 10 ns;
            s_sel <= not s_sel; -- sinal de seleção fica alternando entre nível lógico alto e baixo;
    end process p;

end Behavioral_tb;
```

Teste Bench do DEMUX1_2 em VHDL

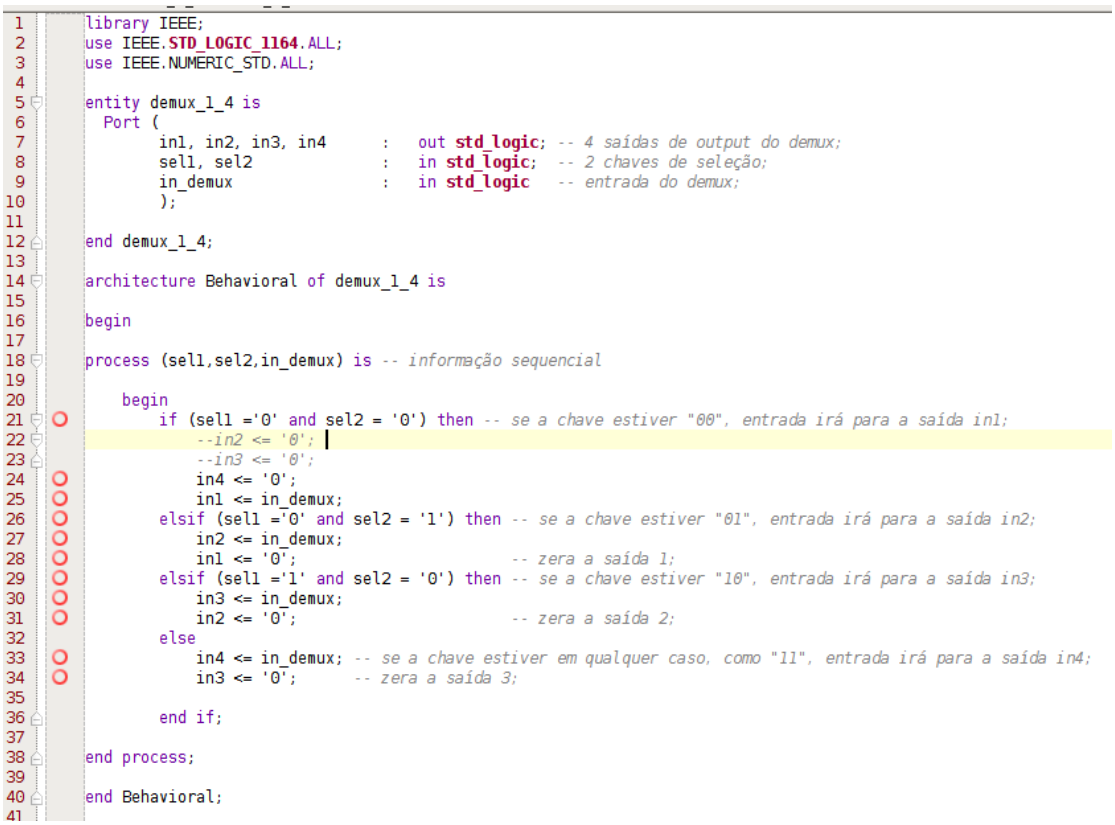


Simulação do DEMUX1_2

● DEMUX1_4

Nesse DEMUX fizemos algumas modificações ao longo do código. Primeiramente esse demux tem 1 entrada e 4 saídas. Tivemos uma entrada (“in_demux”), duas chaves de seleção, (“sel1” e “sel2”) e 4 saídas (“in1”, “in2”, “in3” e “in4”). Tudo isso foi declarado na entidade . Na arquitetura fizemos com que cada caso fosse analisado em código sequencial dentro de um processo. Fizemos todos os casos de seleção e zerando cada saída anterior. Inicializamos a 4ª saída com 0, e não inicializamos as saídas 2 e 3 em zero, e no final ocorreu um pequeno atraso nas saídas 2 e 3. Ajeitamos esse atraso inicializando as 3 últimas entradas em 0. Colocaremos abaixo a simulação com atraso e outro print com a simulação sem atraso.

No teste de simulação criamos os sinais respectivos para mapeamento. Inicializamos o sinal de entrada em 1, e a chave de seleção em “00”. Em seguida fomos passando por cada caso na chave de seleção. Os sinais de saída que estão defasados foi um detalhe que percebemos logo após a simulação e decidimos relatar aqui. Esse atraso foi resolvido com a inicialização em 0 dos 3 últimos sinais de saída. Abaixo seguem os códigos e a simulação com a defasagem. Em seguida os códigos e a simulação sem defasagem.



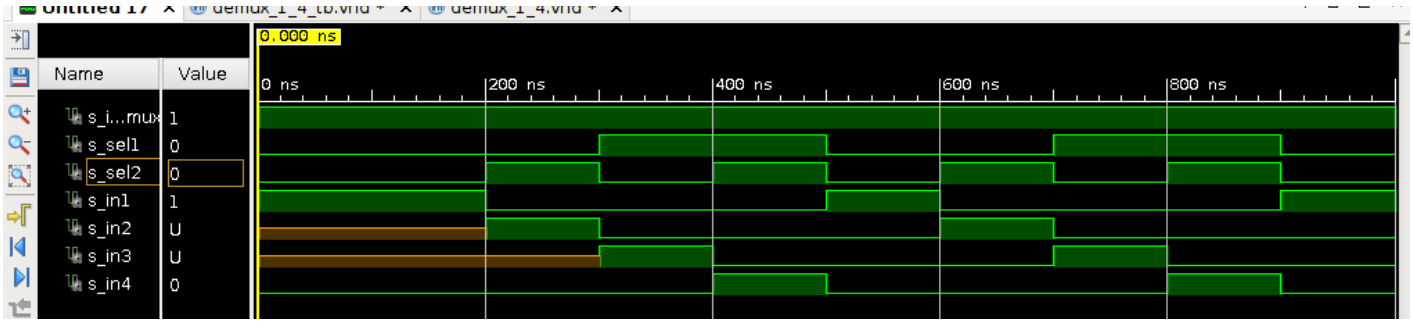
```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity demux_1_4 is
  Port (
        in1, in2, in3, in4      :   out std_logic; -- 4 saídas de output do demux;
        sel1, sel2              :   in std_logic;  -- 2 chaves de seleção;
        in_demux                :   in std_logic   -- entrada do demux;
        );

end demux_1_4;

architecture Behavioral of demux_1_4 is

begin

process (sel1,sel2,in_demux) is -- informação sequencial

    begin
        if (sel1 ='0' and sel2 = '0') then -- se a chave estiver "00", entrada irá para a saída in1;
            --in2 <= '0';
            --in3 <= '0';
            in4 <= '0';
            in1 <= in_demux;
        elsif (sel1 ='0' and sel2 = '1') then -- se a chave estiver "01", entrada irá para a saída in2;
            in2 <= in_demux;
            in1 <= '0';                       -- zera a saída 1;
        elsif (sel1 ='1' and sel2 = '0') then -- se a chave estiver "10", entrada irá para a saída in3;
            in3 <= in_demux;
            in2 <= '0';                       -- zera a saída 2;
        else
            in4 <= in_demux; -- se a chave estiver em qualquer caso, como "11", entrada irá para a saída in4;
            in3 <= '0';      -- zera a saída 3;

        end if;

end process;

end Behavioral;
```

Código do DEMUX1_4 em VHDL (Com atraso de propagação nas saídas)

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity demux_1_4_tb is
end demux_1_4_tb;

architecture Behavioral_tb of demux_1_4_tb is
    component demux_1_4 is
        port(
            in1, in2, in3, in4      :   out std_logic; -- 4 saídas de output do demux;
            sel1, sel2              :   in std_logic;  -- 2 chaves de seleção;
            in_demux                :   in std_logic   -- entrada do demux;
        );
    end component;

    --Inputs
    signal s_in_demux : std_logic := '1'; -- sinal de atribuição para a entrada do demux;
    signal s_sel1     : std_logic := '0'; -- sinal de atribuição para a 1ª chave de seleção;
    signal s_sel2     : std_logic := '0'; -- sinal de atribuição para a 2ª chave de seleção;

    --Outputs
    signal s_in1 : std_logic; -- sinal de atribuição para a saída 1;
    signal s_in2 : std_logic; -- sinal de atribuição para a saída 2;
    signal s_in3 : std_logic; -- sinal de atribuição para a saída 3;
    signal s_in4 : std_logic; -- sinal de atribuição para a saída 4;

begin

-- Instantiate the Unit Under Test (UUT)
    uut: demux_1_4 port map (
                                in_demux => s_in_demux, -- mapeia sinal de entrada com sinal de entrada do demux;
                                in1 => s_in1,           -- mapeia sinal de saída 1 com a saída um do demux;
                                in2 => s_in2,           -- mapeia sinal de saída 2 com a saída dois do demux;
                                in3 => s_in3,           -- mapeia sinal de saída 3 com a saída três do demux;
                                in4 => s_in4,           -- mapeia sinal de saída 4 com a saída quatro do demux;
                                sel1 => s_sel1,         -- mapeia sinal de seleção 1 com a seleção 1 do demux;
                                sel2 => s_sel2          -- mapeia sinal de seleção 2 com a seleção 2 do demux;
                            );
```

```vhdl
    -- Stimulus process
    stim_proc: process
        begin
        -- hold reset state for 100 ns.
        wait for 100 ns;

        s_sel1 <= '0'; s_sel2 <= '0'; -- sinal de seleção em "00", respectivamente;

        wait for 100 ns;

        s_sel1 <= '0'; s_sel2 <= '1'; -- sinal de seleção em "01", respectivamente;

        wait for 100 ns;

        s_sel1 <= '1'; s_sel2 <= '0'; -- sinal de seleção em "10", respectivamente;

        wait for 100 ns;

        s_sel1 <= '1'; s_sel2 <= '1'; -- sinal de seleção em "11", respectivamente;


     end process stim_proc;

end Behavioral_tb;
```

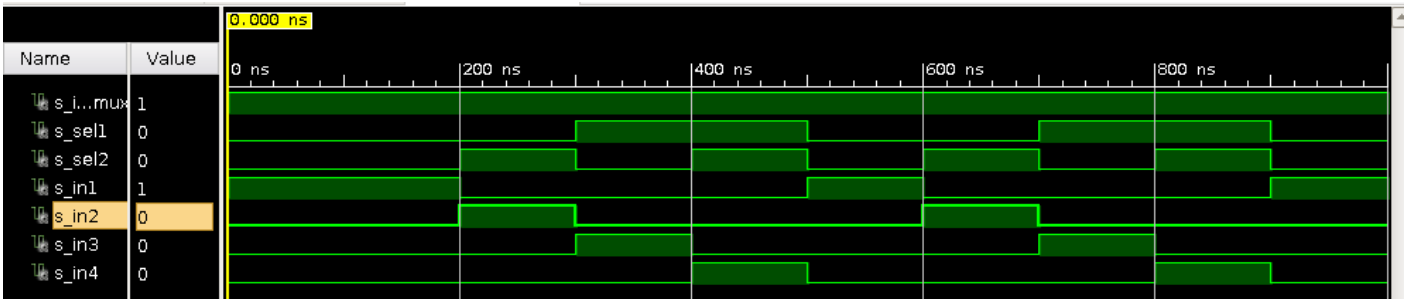
Teste Bench do DEMUX4_1 em VHDL



Simulação do DEMUX1_2 (Com atraso de sinal)

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity demux_1_4 is
  Port (
        in1, in2, in3, in4      :   out std_logic; -- 4 saídas de output do demux;
        sel1, sel2              :   in std_logic;  -- 2 chaves de seleção;
        in_demux                :   in std_logic   -- entrada do demux;
        );

end demux_1_4;

architecture Behavioral of demux_1_4 is

begin

process (sel1,sel2,in_demux) is -- informação sequencial

    begin
        if (sel1 ='0' and sel2 = '0') then -- se a chave estiver "00", entrada irá para a saída in1;
            in2 <= '0'; -- saída 2 inicializada com 0;
            in3 <= '0'; -- saída 3 incializada com 0;
            in4 <= '0'; -- saída 4 inicializada com 0;
            in1 <= in_demux;
        elsif (sel1 ='0' and sel2 = '1') then -- se a chave estiver "01", entrada irá para a saída in2;
            in2 <= in_demux;
            in1 <= '0';                       -- zera a saída 1;
        elsif (sel1 ='1' and sel2 = '0') then -- se a chave estiver "10", entrada irá para a saída in3;
            in3 <= in_demux;
            in2 <= '0';                       -- zera a saída 2;
        else
            in4 <= in_demux; -- se a chave estiver em qualquer caso, como "11", entrada irá para a saída in4;
            in3 <= '0';      -- zera a saída 3;

        end if;

end process;

end Behavioral;
```

Código do DEMUX1_4 em VHDL (Sem atraso de propagação)



Simulação do DEMUX1_2 (Sem atraso de sinal)

#MUX e DEMUX array

● MUX8_16

Nesse multiplexador, temos 8 entradas de 16 bits cada. Como um MUX pode ser representado de acordo com 2^n entradas onde n é o número de bits de seleção, aqui teremos 8 entradas com 3 bits de seleção. Primeiramente criamos 16 entradas do tipo std_logic_vector. Cada entrada é um vetor de tamanho 16. Em seguida criamos uma chave de seleção do tipo std_logic_vector, que armazenará 3 bits. Por último fizemos a criação de uma saída do tipo std_logic_vector que armazenará, assim como nas entradas, 16 bits.

Na parte da arquitetura fizemos a saída receber todos os casos possíveis para cada opção de seleção, indo de “000” ate “111”. No teste de simulação fizemos a parte da componente com todas as entradas e saídas, logo em seguida fizemos a criação dos sinais de entrada, cada um do tipo std_logic_vector que armazena 16 posições. Fixamos valores em hexadecimal para cada um destes sinais para facilitar a visualização na simulação. Também fizemos o mesmo para os sinais de saída e de seleção. O sinal de seleção armazenará 3 bits, onde inicializamos no 1º caso “000”. Logo em seguida fizemos o mapeamento de todas as entradas e saídas.

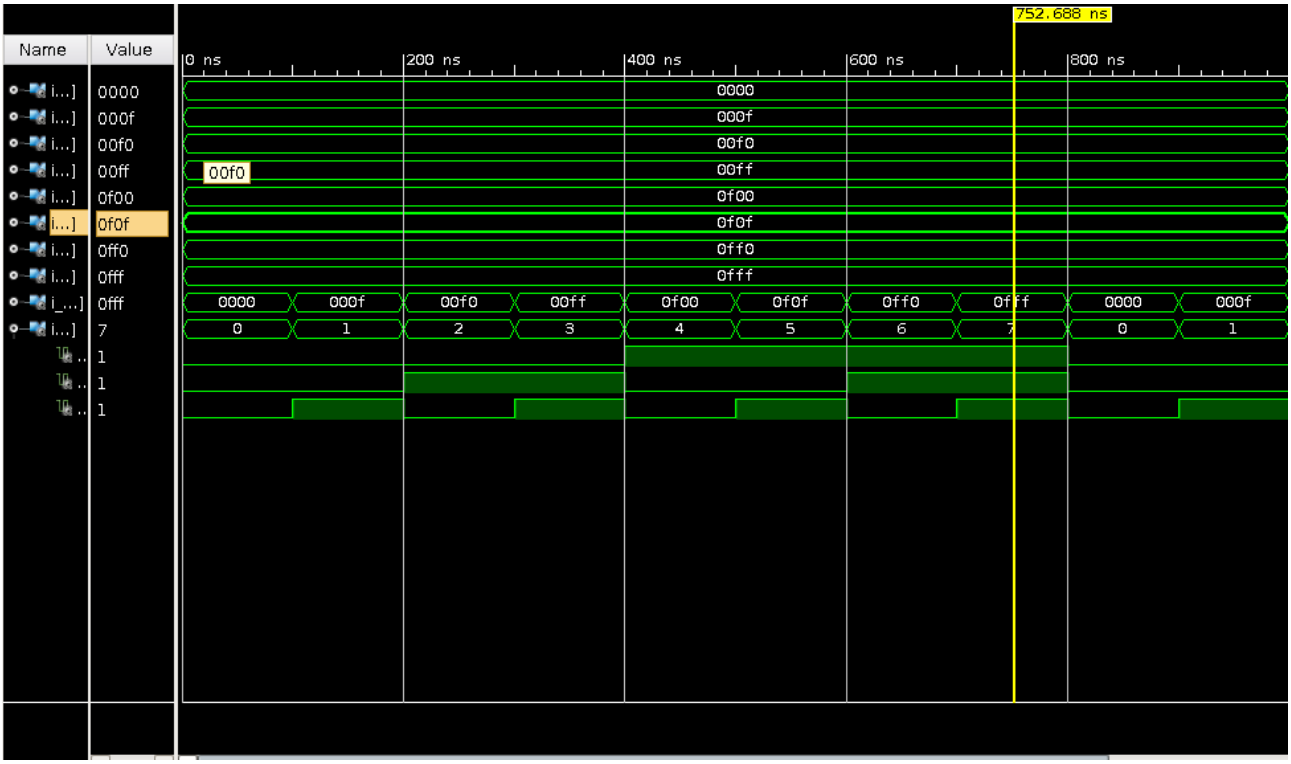
Na parte de processo fizemos com que a chave de seleção fosse incrementando em cada posição em cada bit. Para isso fizemos uma operação semelhante ao que temos em linguagem C, que é o cast. Fizemos a incrementação da chave de seleção dentro de um unsigned, gerando com isso que a chave de seleção faça uma varredura em todos os casos. Na medida que a chave de seleção avança, as entradas de 8 bits serão deslocadas para a saída, de acordo com nossa simulação que está seguida com o código em VHDL e do teste bench.

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.NUMERIC_STD.ALL;
4  entity MUX8_16 is
5        Port ( e0 : in std_logic_vector(15 downto 0);
6               e1 : in std_logic_vector(15 downto 0);
7               e2 : in std_logic_vector(15 downto 0);
8               e3 : in std_logic_vector(15 downto 0);
9               e4 : in std_logic_vector(15 downto 0);
10              e5 : in std_logic_vector(15 downto 0);
11              e6 : in std_logic_vector(15 downto 0);
12              e7 : in std_logic_vector(15 downto 0);
13              key: in std_logic_vector (2 downto 0);
14              s  : out std_logic_vector(15 downto 0)
15             );
16 end MUX8_16;
17
18 architecture Behavioral of MUX8_16 is
19
20 begin
21              with key select
22              s <=  e0 when  "000",
23                    e1 when  "001",
24                    e2 when  "010",
25                    e3 when  "011",
26                    e4 when  "100",
27                    e5 when  "101",
28                    e6 when  "110",
29                    e7 when others;
30 end Behavioral;
31
```

Código em VHDL do MUX8_16

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
entity MUX8_16_tb is
--  Port ( );
end MUX8_16_tb;

architecture Behavioral_tb of MUX8_16_tb is
    component MUX8_16 is
        port(
             e0, e1, e2, e3, e4 , e5, e6, e7 : in std_logic_vector(15 downto 0);
             key                             : in std_logic_vector (2 downto 0);
             s                               : out std_logic_vector(15 downto 0)
        );
        end component;
      signal  i_e0 :  std_logic_vector(15 downto 0) := X"0000";
      signal  i_e1 :  std_logic_vector(15 downto 0) := X"000F";
      signal  i_e2 :  std_logic_vector(15 downto 0) := X"00F0";
      signal  i_e3 :  std_logic_vector(15 downto 0) := X"00FF";
      signal  i_e4 :  std_logic_vector(15 downto 0) := X"0F00";
      signal  i_e5 :  std_logic_vector(15 downto 0) := X"0F0F";
      signal  i_e6 :  std_logic_vector(15 downto 0) := X"0FF0";
      signal  i_e7 :  std_logic_vector(15 downto 0) := X"0FFF";
      signal  i_s  :  std_logic_vector(15 downto 0) := X"FFFF";
      signal  i_key : std_logic_vector(2 downto 0)  :=   "000";
begin
    TESTE : MUX8_16 port map( e0 => i_e0,
                              e1 => i_e1,
                              e2 => i_e2,
                              e3 => i_e3,
                              e4 => i_e4,
                              e5 => i_e5,
                              e6 => i_e6,
                              e7 => i_e7,
                              s => i_s,
                              key => i_key
                              );
    P : process
        begin
            wait for 100 ns;
            i_key <= std_logic_vector(unsigned(i_key) + 1);
        end process P;
end Behavioral_tb;
```

Teste Bench do MUX8_16



Simulação do MUX8_16

● MUX16

Nesse MUX temos 2 entradas com 16 de tamanho cada uma. Na entidade declaramos 3 entradas “a” e “b”, juntamente com a seleção “key”, e uma saída. As entradas e saídas são do tipo std_logic_vector, cada uma guardando 16 posições cada. A saída é do tipo std_logic.

Na arquitetura decidimos realizar um processo caso a chave de seleção assuma algum valor. Para cada caso a saída recebe a entrada. Isso foi feito utilizando if-else.

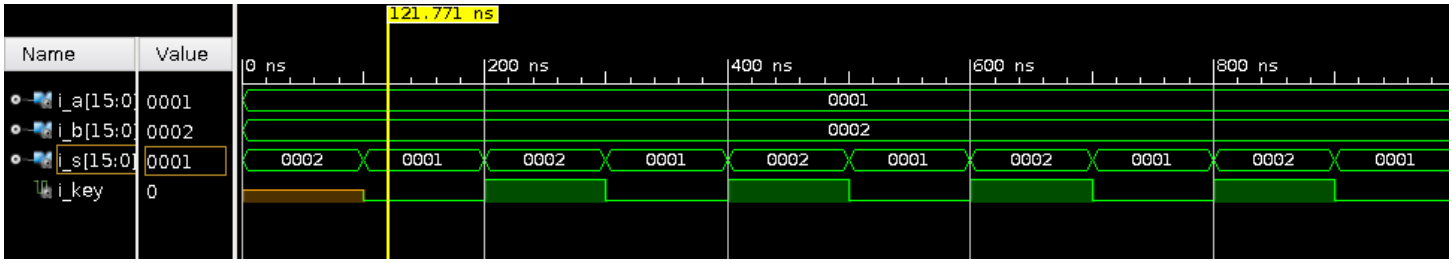
Na parte de simulação declaramos o componente com todas as entradas e saídas. Em seguida declaramos sinais para mapeamento. Fixamos valores para cada entrada em hexadecimal, assim facilitando a visualização da simulação. Depois fizemos o mapeamento e dentro do processo fizemos com que a chave de seleção variasse entre 0 e 1, para que todas as possibilidades de seleção fossem satisfeitas. Para cada valor de seleção, uma entrada é escolhida para o canal de saída. Abaixo está o código em VHDL, o teste bench e a simulação.

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
entity MUX16_tb is
--  Port ( );
end MUX16_tb;

architecture Behavioral_tb of MUX16_tb is
    component MUX16 is port(
         a : in  std_logic_vector(15 downto 0);
         b : in  std_logic_vector(15 downto 0);
         s : out std_logic_vector(15 downto 0);
         key : in std_logic
        );
    end component;

      signal i_a :  std_logic_vector(15 downto 0) := X"0001";
      signal i_b :  std_logic_vector(15 downto 0) := X"0002";
      signal i_s :  std_logic_vector(15 downto 0);
      signal i_key :  std_logic;

begin
    TESTE: MUX16 port map ( a => i_a,
     b=> i_b , s => i_s, key => i_key);
    P: process
       begin
        wait for 100 ns;
        i_key <= '0';
        wait for 100 ns;
         i_key <= '1';
    end process p;
end Behavioral_tb;
```

Teste Bench do MUX16

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity MUX16 is
   Port (
          a : in std_logic_vector(15 downto 0);
          b : in std_logic_vector(15 downto 0);
          s : out std_logic_vector(15 downto 0);
          key: in std_logic
          );
end MUX16;

architecture Behavioral of MUX16 is

begin
    MUX16: process(key)
            begin
              if    key = '0' then s <= a;
              else                 s <= b;
              end if;
           end process MUX16;
end Behavioral;
```

Código em VHDL do MUX16



Simulação do MUX16

● **DEMUX4_16**

Temos um DEMUX com 4 saídas, cada uma composta por 16 bits. Na entidade fizemos a criação do sinal de entrada de 16 bits (“data”), das saídas (“s0”, “s1”, “s2” e “s3) e da chave de seleção (“key”). Na parte da arquitetura fizemos um processo para varrer todas as possibilidades de seleção, e fazer com que em cada caso qual saída receberia a entrada. O nome da entidade é MUX4_16, porém foi apenas um engano na declaração correta de entidade, que seria DEMUX4_16. O código executa normalmente. Lembrando que a entrada do DEMUX e saídas são do tipo std_logic_vector, que guardam 16 posições. Temos um std_logic_vector para a chave de seleção de tamanho 2.

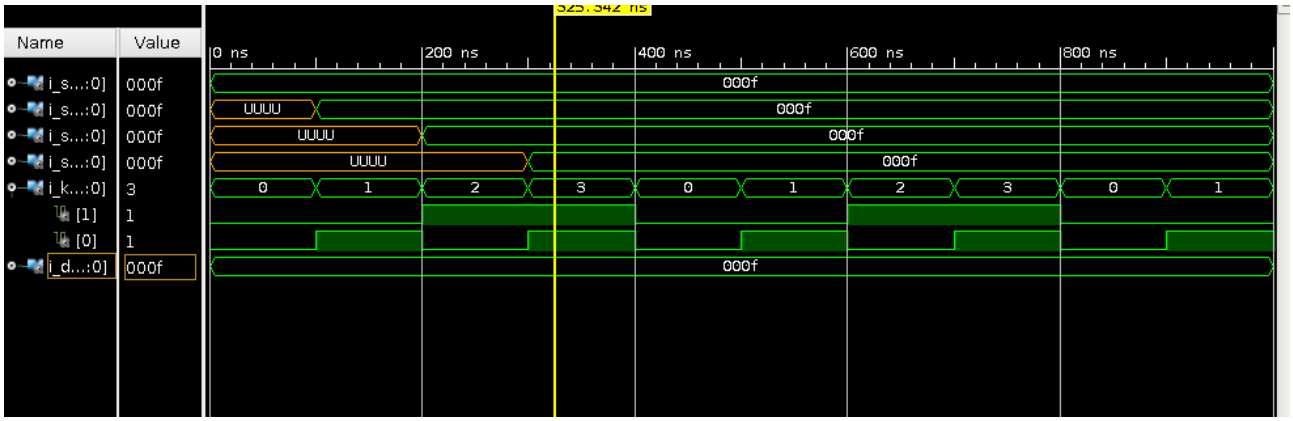
Na parte de simulação, criamos o componente com todas as entradas e saídas já declaradas no arquivo .vhd. Em seguida criamos sinais para o mapeamento, e também fixamos valores em hexadecimal para a entrada do DEMUX. A seleção também está fixada em “00”. Logo fizemos o mapeamento para cada entrada e saída. No processo fizemos algo semelhante ao “cast” em C. Incrementamos cada posição da seleção para podermos varrer todas as possibilidades possíveis, dessa forma para que na simulação os resultados estivessem esperados. O código em VHDL, o teste bench e a simulação seguem abaixo:

```
1
2   library IEEE;
3   use IEEE.STD_LOGIC_1164.ALL;
4   use IEEE.NUMERIC_STD.ALL;
5
6   entity MUX4_16 is    ---- o NOME DA ENTIDADE ESTA ERRADO, MAS NÃO HA PROBLEMA NA EXECUCAO
7    Port (  data : in std_logic_vector   (15 downto 0);
8            s0   : out std_logic_vector (15 downto 0);
9            s1   : out std_logic_vector (15 downto 0);
10           s2   : out std_logic_vector (15 downto 0);
11           s3   : out std_logic_vector (15 downto 0);
12           key  : in std_logic_vector (1 downto 0) );
13  end MUX4_16;
14
15  architecture Behavioral of MUX4_16 is
16
17  begin
18      MUX4_16: process(key)
19               begin
20               if    key = "00" then s0 <= data;
21               elsif key = "01" then s1 <= data;
22               elsif key = "10" then s2 <= data;
23               else                  s3 <= data;
24               end if;
25               end process;
26  end Behavioral;
27
```

Código em VHDL do DEMUX 4_16

```vhdl
use IEEE.NUMERIC_STD.ALL;

entity MUX4_16_tb is
--  Port ( );
end MUX4_16_tb;

architecture Behavioral_tb of MUX4_16_tb is
      component MUX4_16 is    -- NOME DA ENTIDADE ESTA ERRADO
        port(
                data  : in  std_logic_vector (15 downto 0);
                 s0   : out std_logic_vector (15 downto 0);
                 s1   : out std_logic_vector (15 downto 0);
                 s2   : out std_logic_vector (15 downto 0);
                 s3   : out std_logic_vector (15 downto 0);
                 key  : in  std_logic_vector (1 downto 0) );
        end component;
        signal i_s0  :  std_logic_vector(15 downto 0);
        signal i_s1  :  std_logic_vector(15 downto 0);
        signal i_s2  :  std_logic_vector(15 downto 0);
        signal i_s3  :  std_logic_vector(15 downto 0);
        signal i_key :  std_logic_vector(1 downto 0 ) := "00";
        signal i_data:  std_logic_vector(15 downto 0) := X"000F";
begin
    TESTE: MUX4_16 port map( data => i_data, --NOME DA ENTIDADE ESTA ERRADO---SERIA DEMUX4_16
                             s0 => i_s0,
                             s1 => i_s1,
                             s2 => i_s2,
                             s3 => i_s3,
                             key => i_key);
          P: process
             begin
              wait for 100ns;
              i_key <= std_logic_vector(unsigned(i_key)+ 1);
            end process P;
end Behavioral_tb;
```

Teste bench do DEMUX4_16



Simulação do DEMUX4_16