Atividade do Capítulo 3 e Moodle

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

FLIP FLOP D

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


entity FF_D is
  Port (
            D,clk    :       in std_logic; -- entrada de dados e clock;
            Q        :       out std_logic -- saída de Dados Q;
        );
end FF_D;

architecture Behavioral of FF_D is

begin

process(clk)
begin
    if(clk' event and clk = '1') then -- Flip Flop sensível a borda de subida do clock;
        Q <= D;                       -- Saída recebe dado quando há um pulso de clock;
    end if;
end process;

end Behavioral;
```

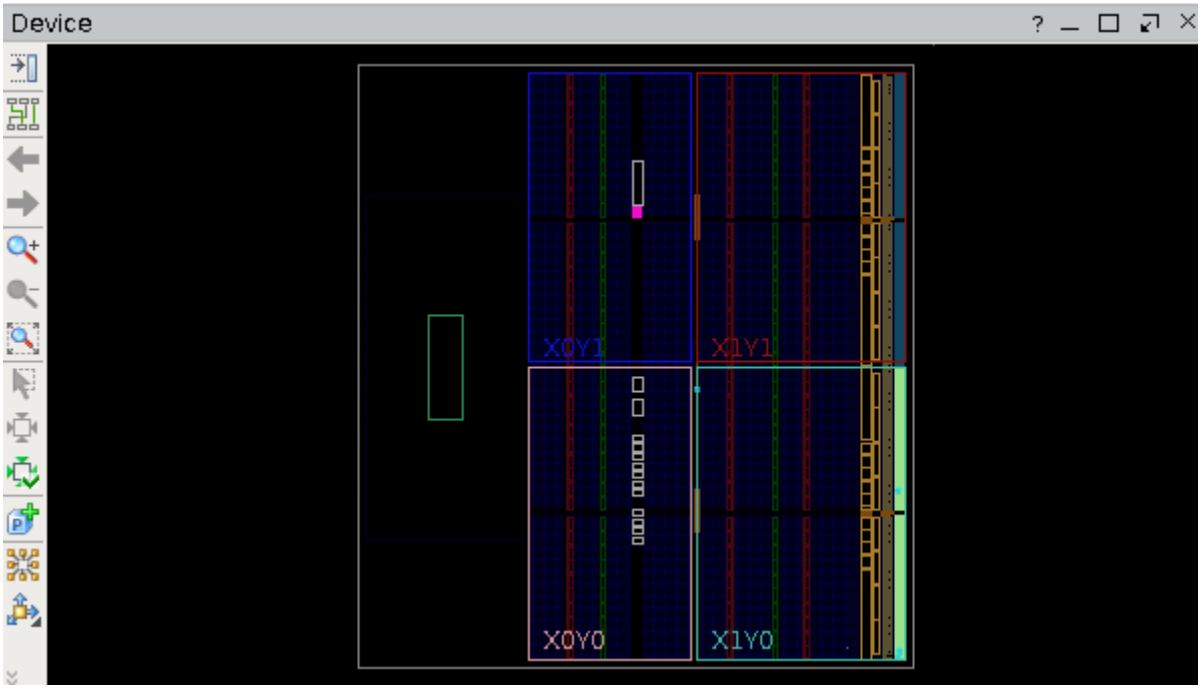
O Flip Flop D consiste de uma única entrada, uma saída e um clock. A saída recebe a entrada quando houver um pulso de clock, seja ele de subida ou de descida. Na implementação desse Flip Flop fizemos a criação de 2 entradas, uma de dados “D” e outra para o clock “clk”. Fizemos também uma saída de dados “Q”. As entradas e as saídas são compostas de apenas 1 bit. Dentro da arquitetura fizemos com que o Flip Flop fosse sensível a borda de subida do clock, ou seja, quando ouver um pulso no sentido da subida do clock a saída recebe o dado.

No teste bench fizemos a criação de sinais referentes a cada entrada e saída da entidade. Fixamos o clock em 0 e a entrada de dados em 1. Dentro do processo fizemos com que a entrada e o clock ficassem alternando entre 0 e 1. O teste bench e as simulações se encontram abaixo:

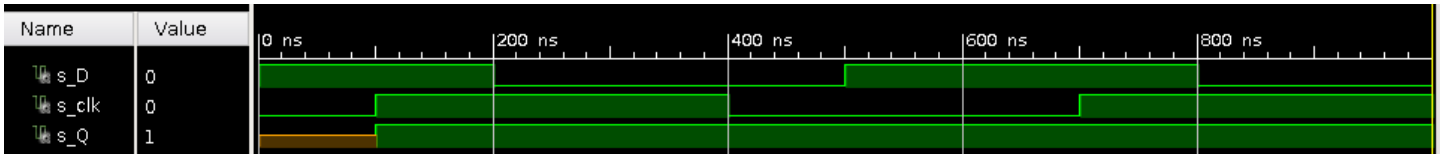
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;


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

architecture Behavioral of FFD_tb is

component FF_D is

Port(
            D,clk: in std_logic;     -- entrada de dados e clock;
            Q:     out std_logic     -- saída de Dados Q;
     );

end component;
```

```
signal s_D      :   std_logic := '1'; -- sinal para a entrada de dados;
signal s_clk    :   std_logic := '0'; -- sinal para o clock;
signal s_Q      :   std_logic;        -- sinal para a saída de dados;

begin

uut: FF_D port map(D => s_D, clk => s_clk, Q => s_Q); -- mapeamento

teste: process

begin

wait for  100 ns;

s_clk <= not s_clk; -- variação no clock;

wait for  100 ns;

s_D <= not s_D; -- variação na entrada de dados;

wait for  100 ns;

end process teste;

end Behavioral;
```

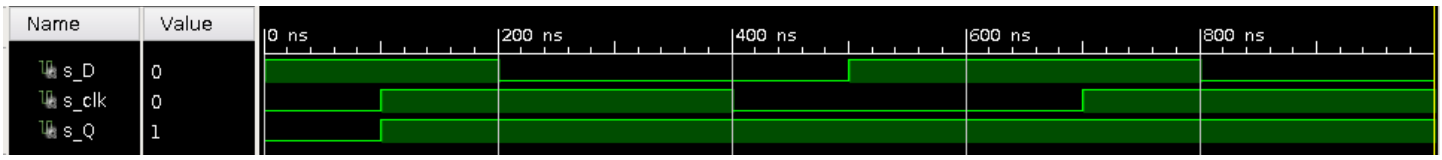
Teste bench



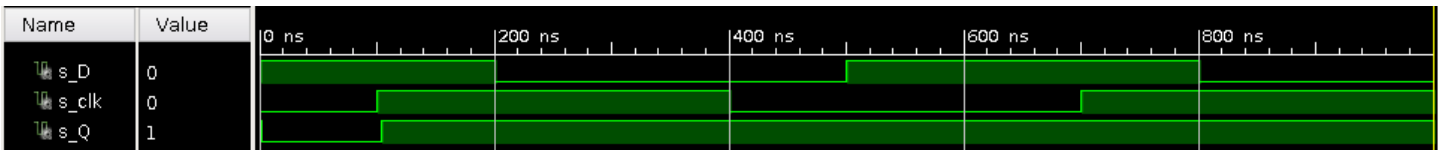
Implementation design



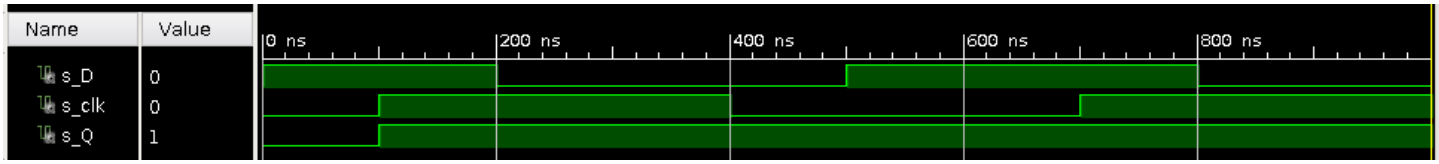
Simulação de comportamento



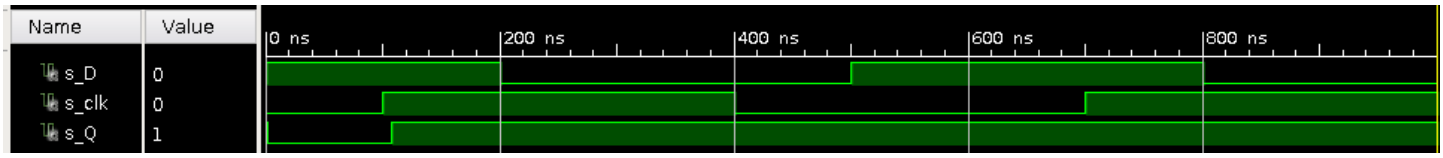
Synthesis functional simulation



Synthesis timing simulation



Implementation functional simulation



Implementation timing simulation

REGISTRADOR DE 1 BIT

Registradores são unidades de memória que podem ser implementados utilizando flip flops. O mais simples de todos os registradores são os de 1 bit. O projeto é bem parecido com o flip flop do tipo D, porém aqui surge uma nova entrada chamada “load”, que só permite que o dado de entrada saia para a saída caso “load” esteja em nível lógico alto. Registradores guardam o estado anterior, e só atualizam o dado de entrada após o clock estar ativado e o load estiver em 1. Na entidade criamos uma entrada D, o clock “clk” e o load. Criamos uma saída Q.

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

entity reg_1 is
  Port (
            load    :   in std_logic; -- entrada de load;
            inp0    :   in std_logic; -- entrada de dados;
            clk     :   in std_logic; -- clock;
            Q       :   out std_logic -- saída de dados;
        );
end reg_1;
```

Na arquitetura fizemos o clock ser sensível a borda de subida. Se o load estiver carregado em 1, então o dado de entrada vai para a saída. Caso contrário, a saída permanece a anterior, e dessa forma é guardado dados ao decorrer do tempo.

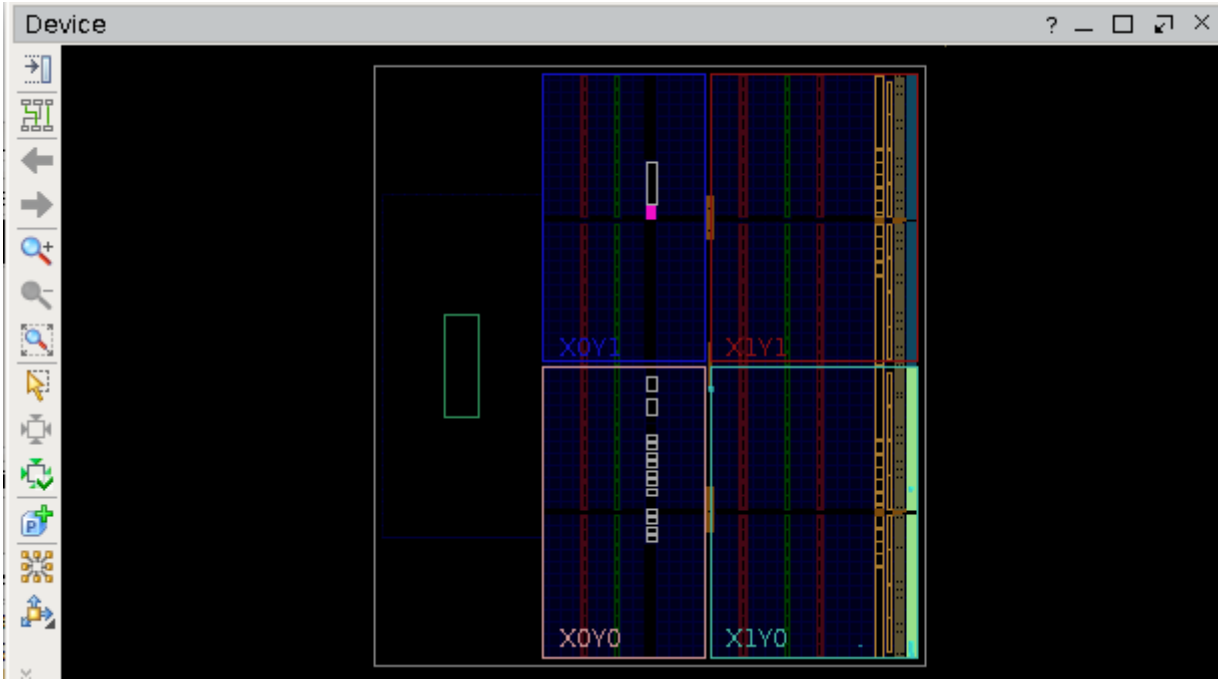
```
architecture Behavioral of reg_1 is

begin

process(clk)

begin

if(clk' event and clk = '1') then -- Se o clock tiver nível de subida...
    if load = '1' then            -- Se o load estiver ativo a entrada vai pra saída;
        Q <= inp0;
    end if;
end if;

end process;


end Behavioral;
```

No teste bench fizemos a criação de sinais referentes a cada variável da entidade Fixamos valores para o load, clock e entrada. Fizemos o mapeamento e no processo fizemos com que os sinais de load, clock e entrada alternassem entre 0 e 1. O teste bench e as simulações se encontram abaixo:

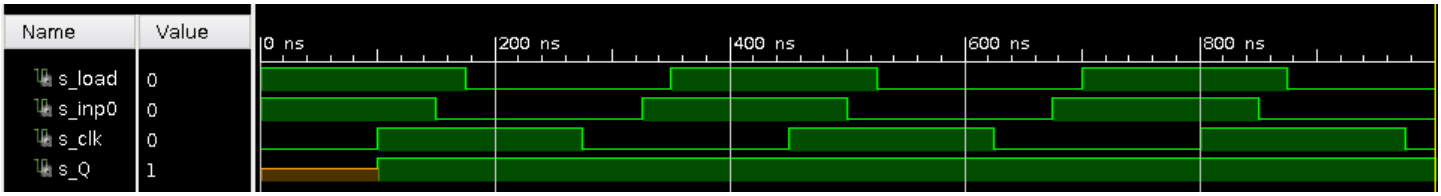
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity reg_1_tb is
--  Port ( );
end reg_1_tb;

architecture Behavioral of reg_1_tb is

component reg_1 is

Port (
            load    :   in std_logic; -- entrada de load;
            inp0    :   in std_logic; -- entrada de dados;
            clk     :   in std_logic; -- clock;
            Q       :   out std_logic -- saída de dados;
    );

end component;

signal s_load       :   std_logic := '1'; -- sinal para o load;
signal s_inp0       :   std_logic := '1'; -- sinal para a entrada de dados;
signal s_clk        :   std_logic := '0'; -- sinal para o clock;
signal s_Q          :   std_logic;        -- sinal para a saída;

begin

uut : reg_1 port map(load => s_load, inp0 => s_inp0, clk => s_clk, Q => s_Q); -- mapeamento
```

```
teste: process
begin

wait for 100 ns;

s_clk <= not s_clk; -- variação do sinal de clock;

wait for 50 ns;

s_inp0 <= not s_inp0; -- variação do sinal da entrada de dados;

wait for 25 ns;

s_load <= not s_load; -- variação do sinal de load;

end process teste;

end Behavioral;
```

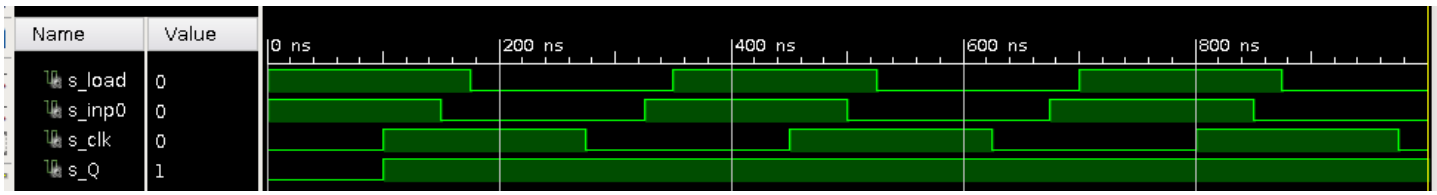
Teste bench



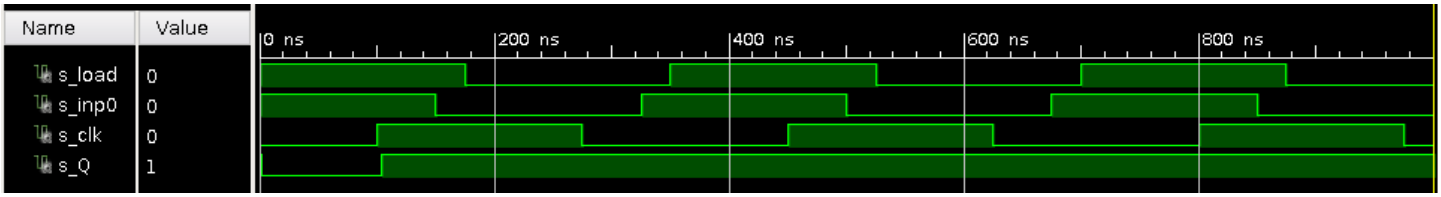
Implementation design



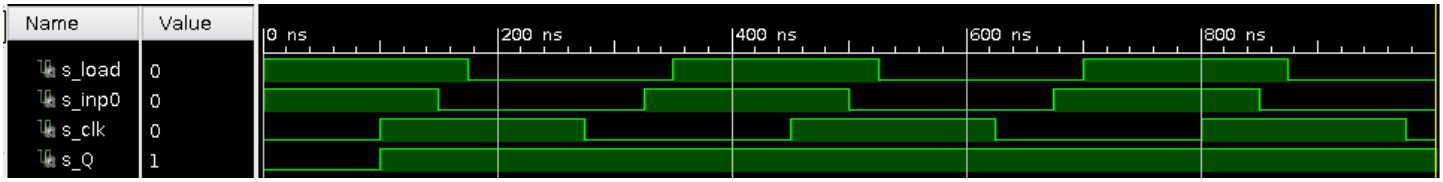
Simulação de comportamento



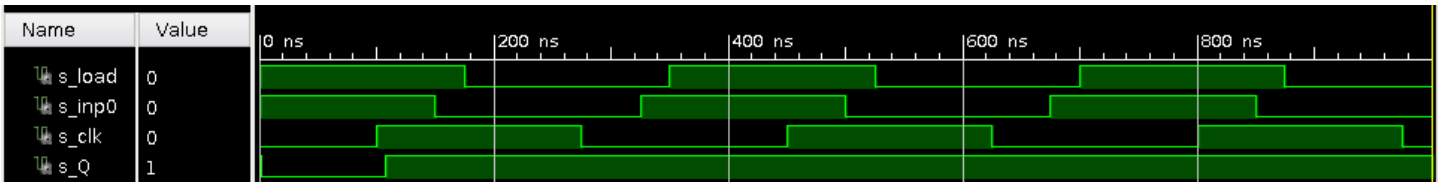
Synthesis functional simulation



Synthesis timing simulation



Implementation functional simulation



Implementation timing simulation

REGISTRADOR DE 16 BITS

A implementação do registrador de 16 bits é semelhante ao registrador de 1 bit. Aqui temos uma entrada de 16 bits e uma saída de 16 bits. Temos um clock e um load. Quando o load está em 1 e o clock está em nível de subida o barramento de dados de 16 bits vai pra saída. Na entidade fizemos uma entrada de dados de 16 bits, uma saída também de 16 bits, um clock e um load.

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

entity reg_16 is
  Port (
            data_in     :   in std_logic_vector(15 downto 0);  -- entrada de 16 bits;
            data_out    :   out std_logic_vector(15 downto 0); -- saída de 16 bits;
            clk         :   in std_logic;                      -- clock;
            load        :   in std_logic                       -- load;
      );
end reg_16;
```

Na arquitetura fizemos a condição do load em nível lógico alto. Se load estiver em 1 a entrada vai pra saída. Caso não seja, a saída permanece a mesma anterior.

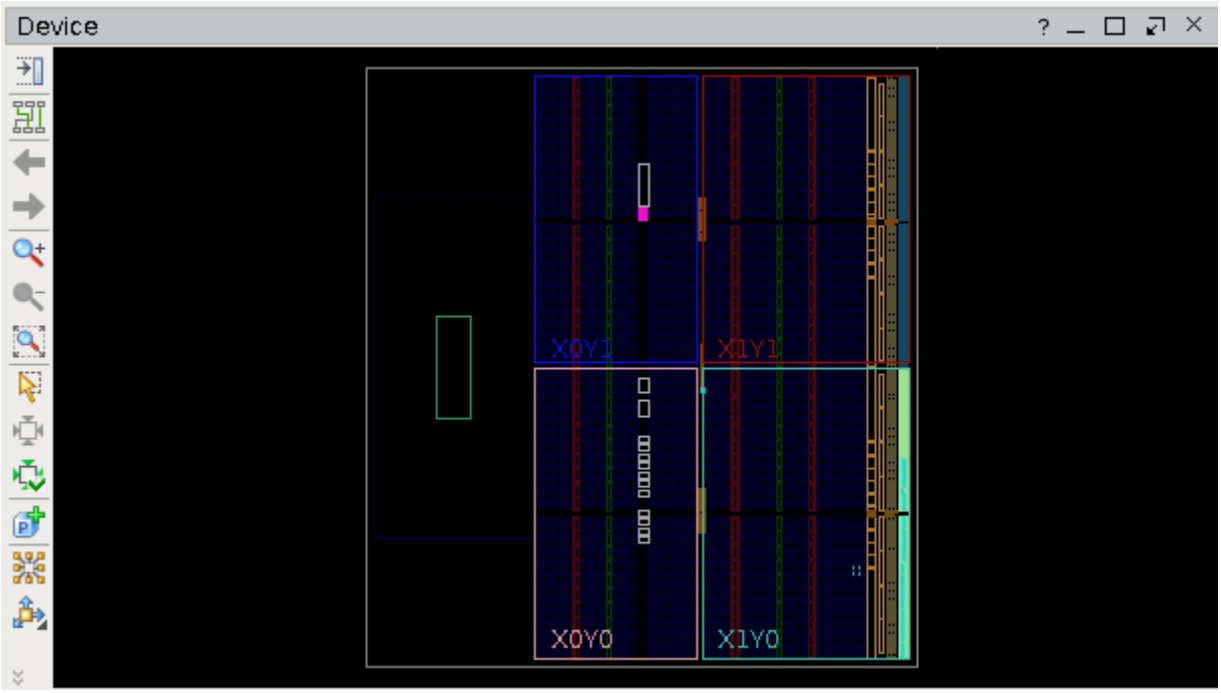
```
architecture Behavioral of reg_16 is

begin

process(clk)

begin

    if(clk' event and clk = '1') then -- clock sensível a nivel de subida;
        if load = '1' then            -- barramento vai pra saída quando load está em 1;
            data_out <= data_in;
        end if;
    end if;

end process;

end Behavioral;
```

No teste bench fizemos a criação de sinais referentes a cada variável na entidade. Inicializamos a entrada com X“FFFF”. Iniciamos o load em nível lógico alto e o clock em nível lógico baixo. Fizemos o mapeamento em seguida referente a cada variável. Dentro do processo fizemos com que a entrada de dados incrementasse em uma unidade. Fizemos também uma variação do clock e do load utilizando uma not em cada sinal referente. O teste bench e as simulações seguem abaixo:

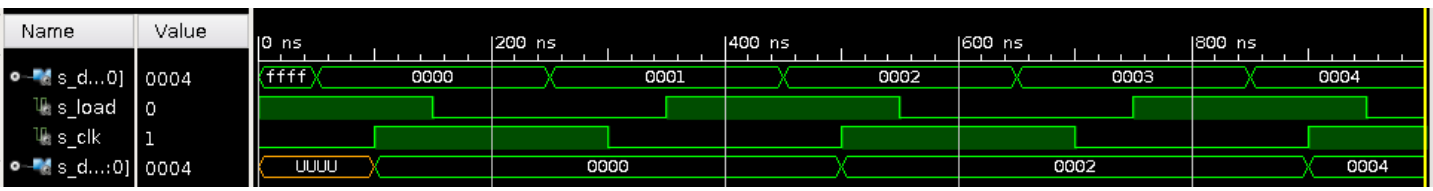
```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.NUMERIC_STD.ALL;
4
5  entity reg_16_tb is
6  --  Port ( );
7  end reg_16_tb;
8
9  architecture Behavioral of reg_16_tb is
10
11 component reg_16 is
12
13 Port(
14         data_in     :   in std_logic_vector(15 downto 0);  -- entrada de 16 bits;
15         data_out    :   out std_logic_vector(15 downto 0); -- saída de 16 bits;
16         clk         :   in std_logic;                      -- clock;
17         load        :   in std_logic                       -- load;
18     );
19
20 end component;
21
22 signal s_data_in    :   std_logic_vector(15 downto 0) := X"FFFF"; -- sinal para entrada fixado em X"FFFF";
23 signal s_load       :   std_logic := '1';                         -- sinal para o load;
24 signal s_clk        :   std_logic := '0';                         -- sinal para o clock;
25 signal s_data_out   :   std_logic_vector(15 downto 0);           -- sinal para a saída;
26
```

```
27  begin
28
29  uut: reg_16 port map(data_in => s_data_in, data_out => s_data_out, clk => s_clk, load => s_load);
30
31  teste: process
32
33  begin
34
35      s_data_in <= std_logic_vector(unsigned(s_data_in) + 1); -- incrementa a entrada;
36
37      wait for 100 ns;
38
39      s_clk <= not s_clk;      -- variação do clock;
40
41      wait for 100 ns;
42
43      s_load <= not s_load;    -- variação do load;
44
45      wait for 100 ns;
46
47  end process teste;
48
49  end Behavioral;
50
```

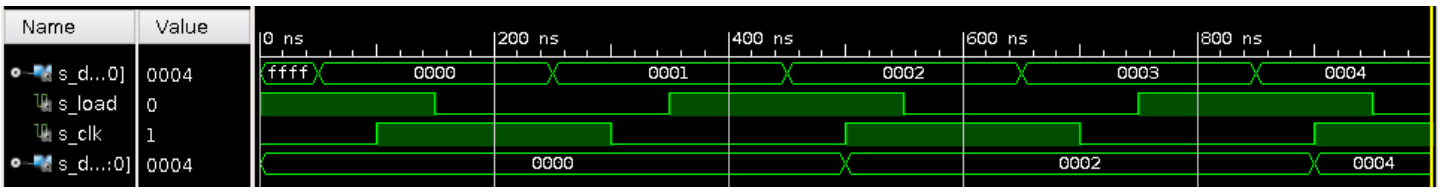
Teste bench



Implementation design

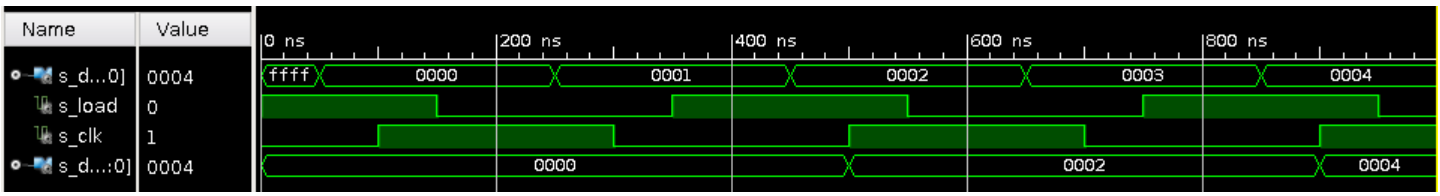


Simulação de comportamento

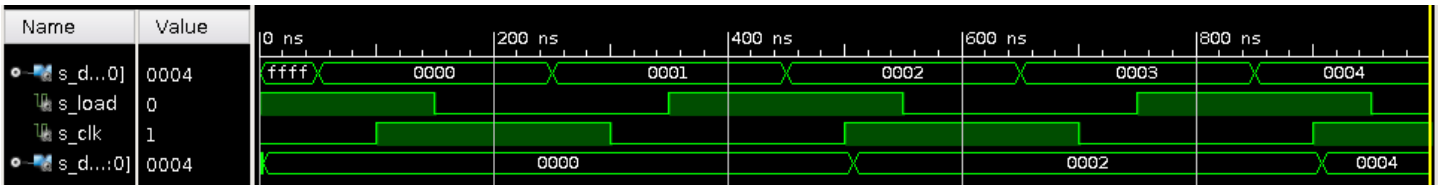


Synthesis functional simulation

Synthesis timing simulation



Implementation functional simulation



Implementation timing simulation

8-REGISTER MEMORY (RAM8)

Para a criação da RAM8 fizemos a criação de 5 variáveis na entidade. Uma entrada de 16 bits e uma saída de 16 bits. Um clock, um load e um address. Diferente das memórias anteriores, temos agora uma entrada de endereço que será salvo na saída. Esse endereço possui 3 posições, e é do tipo std_logic_vector. Fizemos também a criação de um array de 8 posições, e que em cada uma dessas posições tenha-se uma capacidade de armazenamento de 16 bits. Essa é a ideia da construção deste tipo de memória. 8 registradores de 16 bits cada. Dentro do processo que é sensível ao clock, fizemos uma variável (index) que fará a identificação do endereço selecionado para colocar na saída. Se o clock for de nível de subida, então faremos a conversão do endereço para inteiro e isso será armazenada na nossa variável de índice (index). Logo após a saída recebe o conteúdo da posição de memória indicada pelo índice. Fizemos a inicialização das posições de memória com valores fixados em todas as 8 posições como segue abaixo:

```
entity RAM8 is
    Port ( input : in STD_LOGIC_VECTOR (15 downto 0);  --- entrada  da memoria
           address : in STD_LOGIC_VECTOR (2 downto 0); ---  endereço a ser salvo na saida
           load : in STD_LOGIC;
                        ---- sinal de carga na Ram
           clk : in STD_LOGIC;                          -- clock da Memáorai
           output : out STD_LOGIC_VECTOR (15 downto 0));  ---- saida
end RAM8;

architecture Behavioral of RAM8 is


    type RAM is array (7 downto 0) of std_logic_vector(15 downto 0); -- array com 8 posicoes
    signal MEM8: RAM := (X"7777",X"6666",X"5555",X"4444" ,X"3333",X"2222",X"1111",X"0000") ;

     ----------------------------------------------------------------------------------------


    ------------------------------------------------------------------------------------

begin
    FDD: process(clk)
    variable index: integer;                          --- indice que indica o endereço selecionado para colocar na saida
    begin

        if rising_edge(clk) then
```

```
begin

    FDD: process(clk)
    variable index: integer;                        -- indice que indica o endereço selecionado para colocar na saida;
    begin
        if rising_edge(clk) then
            index := to_integer(unsigned(address));    -- passa o endereço para inteiro e armazenada no indice
            output <= MEM8(index);                     -- saida recebe o conteudo da posicao da  memória indicada pelo indice
        end if;
    end process;
```

Depois foi criado um outro processo que será sensível ao load. Fizemos uma variável que indicará a posição ser salvo na memória (index). Se o load for igual a 1 então o índice receberá em qual posição da memória salvar a entrada. Logo o valor é carregado para a saída.

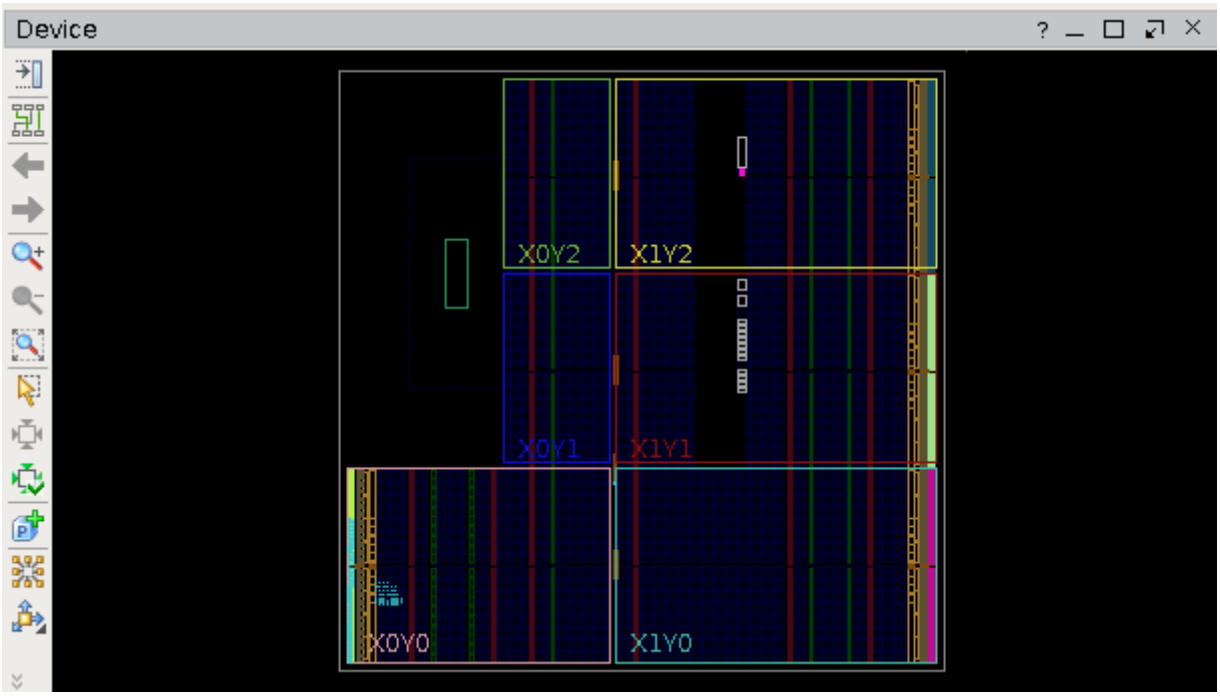
```
    MUX: process(load)                                  -- processo sensivel ao load
    variable index: integer;                            -- variavel que indica a posicao a salvar na memoria
    begin
        if load = '1' then                              -- caso seja 1
            index := to_integer(unsigned(address));     -- indice que indica em qual posicao da memoria salvar a entrada
            MEM8(index) <= input;                       -- carrega o valor da entrada na saída
        end if;
    end process;

end Behavioral;
```

No teste bench fizemos a criação de sinais correspondentes a cada variável da entidade. O valor da entrada de dados foi fixado em “F11F” hexadecimal. O sinal de endereço foi fixado manualmente. Aqui fixamos em “111”. Fizemos um sinal de clock e de load. Logo em seguida fizemos o mapeamento referente a cada sinal e no primeiro processo fizemos a alternância entre o load e o clock utilizando uma porta inversora. Escolhemos outra placa para fazer simulação. Na zybo acontecia muito erro ou nem fazia as simulações direito. Na simulação de comportamento 1 temos que o primeiro endereço fixado “7777” irá para a saída caso houver um pulso de clock e o load estiver ativo. Caso isso não ocorra então a saída permanece com o estado anterior. Temos um carregamento dos endereços antes inicializados na medida que o clock e o load estiverem ativos. Já na simulação de comportamento 2 temos o valor da entrada em “F11F” e o endereço 7. Cada pulso de clock em que o load estiver ativo em 1, temos o carregamento do valor na memória. Quando o load está em 0 é pego o valor no endereço 7 e carregado na memória.

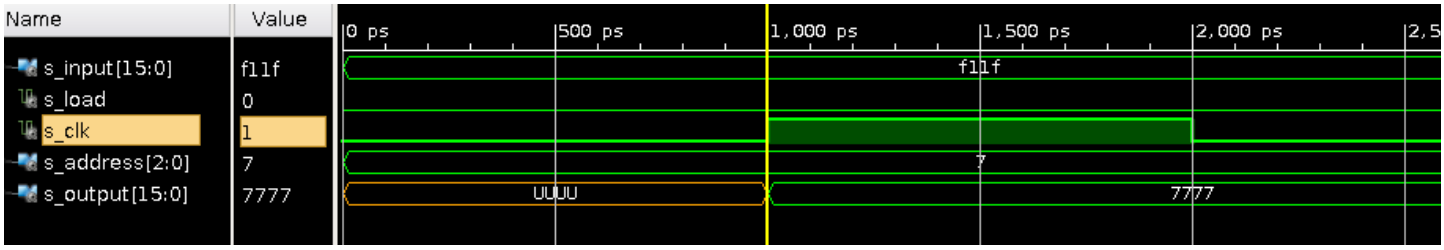
Quando há um pulso de clock e o load estiver em 0 então o valor é carregado na saída. Isso mostra na simulação de comportamento 2. Escolhemos a Zynch. O teste bench e as simulações se encontram abaixo:

```
entity tb_RAM8 is
--  Port ( );
end tb_RAM8;

architecture Behavioral of tb_RAM8 is
 ------------------------------------------------------------- component ------------
    component RAM8 is
        Port ( input : in STD_LOGIC_VECTOR (15 downto 0);
           address : in STD_LOGIC_VECTOR (2 downto 0);
           load : in STD_LOGIC;
           clk : in STD_LOGIC;
           output : out STD_LOGIC_VECTOR (15 downto 0));
     end component;
------------------------------------ sinais  mapeados- ----
     signal s_input  : std_logic_vector(15 downto 0) := x"F11F";
     signal s_load : std_logic :=  '0';
     signal s_clk : std_logic   :=  '0';
     signal s_address: std_logic_vector(2 downto 0)  :=   "111";
     signal s_output : std_logic_vector(15 downto 0);         --- saida da memoria
begin

    UTT: RAM8 port map (s_input, s_address, s_load, s_clk, s_output);

     process
     begin
            wait for 1ns;
            s_clk<= not s_clk;  --------pulsos de clock
            WAIT FOR 3 NS;
            s_LOAD<= not S_LOAD;
     end process;

end Behavioral;
```

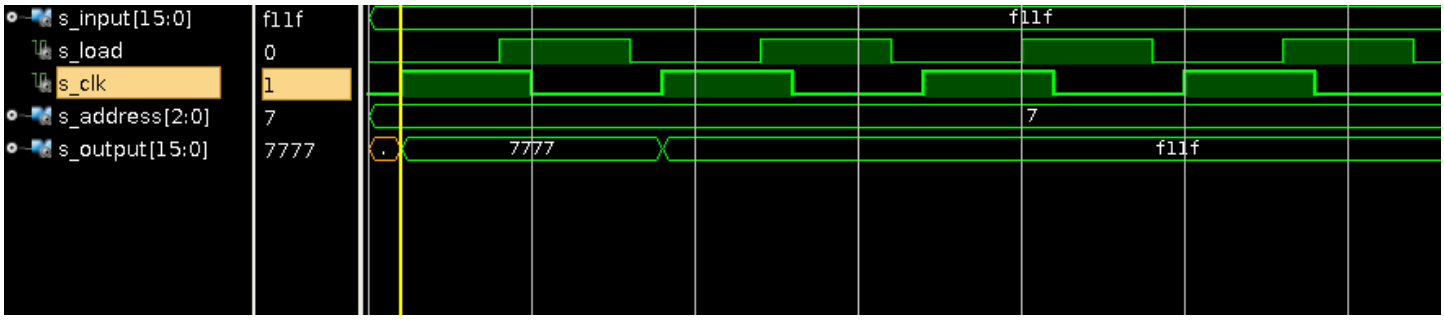
Teste bench



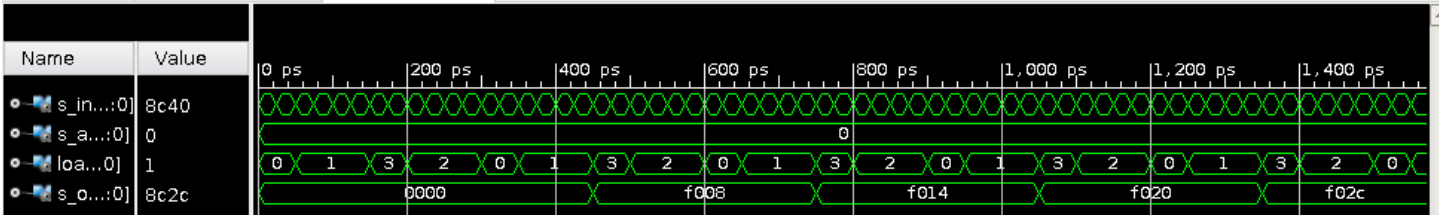
Implementation design



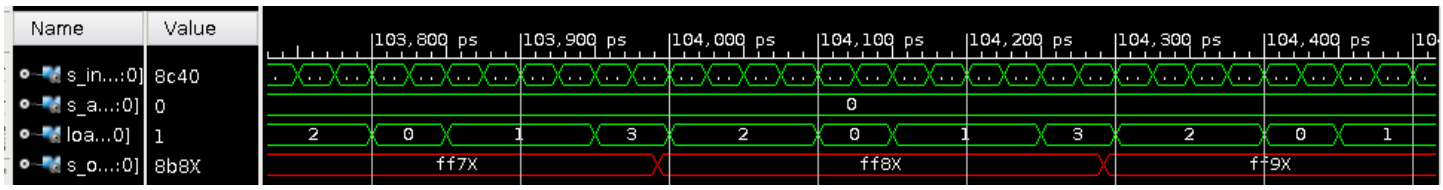
Simulação de comportamento 1 (Leitura)



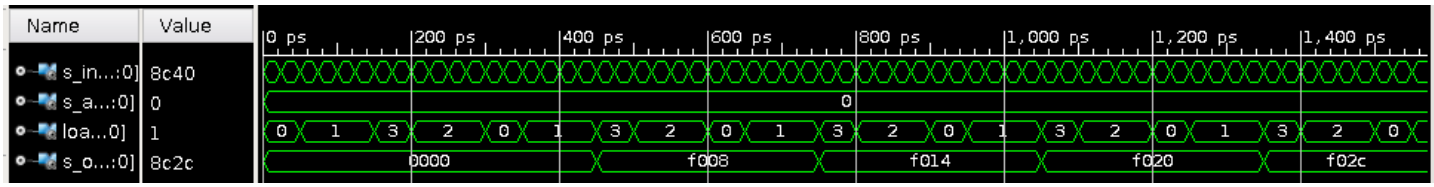
Simulação de comportamento 2 (Escrita)



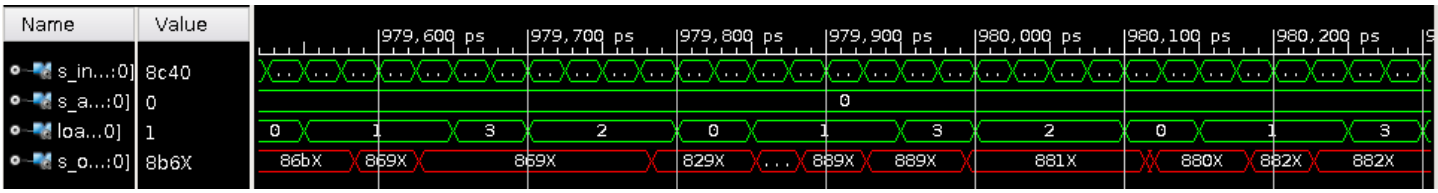
Synthesis functional simulation



Synthesis timing simulation



Implementation functional simulation



Implementation timing simulation

N-REGISTER MEMORY

A construção dessa memória é bem parecida com a anterior, com a diferença de que agora teremos um conjunto de 8 RAM's de 8. Na entidade fizemos a construção das mesmas variáveis da memória anterior. Uma entrada de 16 bits e uma saída de 16 bits. Um load, clock e uma variável de endereço. Logo em seguida fizemos a criação de um "banco de memória", ou seja, a construção de 8 memórias RAM's de tamanho 8. Depois criamos uma variável que faz o papel de um ponteiro para a escolha do endereço de memória da RAM.

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

entity N_RAM is
    Port (  input   : in std_logic_vector(15 downto 0);   -- entrada de 16 bits;
            clk     : in std_logic;                       -- clock;
            load    : in std_logic;                       -- load;
            address : in std_logic_vector(5 downto 0);    -- endereço;
            output  : out std_logic_vector(15 downto 0)); -- saída de dados de 16 bits;
end N_RAM;

architecture Behavioral of N_RAM is
------------------------------CONTRUCAO DO BANCO DE MEMORIA-------------------------

        type RAM   is array ( 7 downto 0, 7 downto 0) of std_logic_vector( 15 downto 0);  -- CONJUNTO DE 8 RAM8
        SIGNAL PONTEIRO_D_RAM : ram;                                                      -- "ponteiro" para escolha de endereço;
```

Em seguida dentro da arquitetura fizemos um processo para a escolha do endereço das memórias que foram criadas. Fizemos a criação de duas variáveis, uma que escolhe a posição da memória RAM (index_RAM) e outra que escolhe uma das memórias (index_D_RAM). Todas essas variáveis do tipo inteiro. Depois fizemos o clock ser sensível a borda de subida. Se isso for verdade, então a variável "index_D_RAM" recebe os três primeiros dígitos da escolha de endereço da RAM8. Depois a variável "index_RAM" recebe os 3 últimos dígitos de endereço da RAM8. Esses endereços são convertidos para tipo inteiro para facilitar o recebimento dos valores dessas variáveis. Em seguida será colocado na saída o endereço especificado utilizando a variável que faz o papel do ponteiro (PONTEIRO_D_RAM). A variável que faz o papel de ponteiro recebe as duas primeiras variáveis que fizeram o papel de recebimento dos dígitos anteriormente.

```
begin
    ESCOLHA_DE_RAM: PROCESS(clk)
              --- escolhe o endereço do banco de memória
        variable index_RAM : integer;         --- escolhe a posicao da RAM8
        variable index_D_RAM : integer;       --- escolhe uma das RAM8
    begin
        if rising_edge(clk) then
            index_D_RAM := TO_INTEGER(unsigned ( address(5 downto 3)));  --- xxxyyyy (corresponde aos tres primeiros digitos
                                                                         ---          de escolha de ram8)
            index_RAM := TO_INTEGER  (unsigned ( address(2 downto 0)));  --- xxxyyyy (corresponde aos tres ultimos digitos
                                                                         ---          de palavra na ram8)
            OUTPUT <=  PONTEIRO_D_RAM(index_D_RAM, INDEX_RAM);       --- coloca na saida o conteudo do endereço especificado
        end if;
    end process;
```

Depois fizemos o carregamento da memória dentro de outro processo. Fizemos as mesmas duas variáveis (index_RAM e index_D_RAM) para auxílio. Se o load estiver ativo em 1, então "index_D_RAM" recebe os 3 primeiros dígitos da escolha de RAM. Em seguida fizemos a variável "index_RAM" receber os outros 3 últimos dígitos do endereço. Depois fizemos o carregamento da entrada nesses respectivos endereços usando as variáveis auxiliares antes criadas.

```
-------------------------------------------------------------------LOAD DA MEMORIA -------------------------------
    CARREGAMENTO: Process (Load)       --- escolhe o endereço do banco de memória
        variable index_RAM : integer;         --- escolhe a posicao da RAM8
        variable index_D_RAM : integer;       --- escolhe uma das RAM8
    begin
        if LOAD = '1' then
            index_D_RAM := TO_INTEGER(unsigned ( address(5 downto 3))); -- xxxyyyy (corresponde aos tres primeiros
                                                                        --          digitos de escolha de ram8)
            index_RAM := TO_INTEGER  (unsigned ( address(2 downto 0))); -- xxxyyyy (corresponde aos tres ultimos digitos
                                                                        --           de palavra na ram8)
            PONTEIRO_D_RAM(index_D_RAM, INDEX_RAM)<= INPUT;       -- coloca na MEMORIA o conteudo O VALOR DA ENTRADA
        end if;
    end process;
end Behavioral;
```

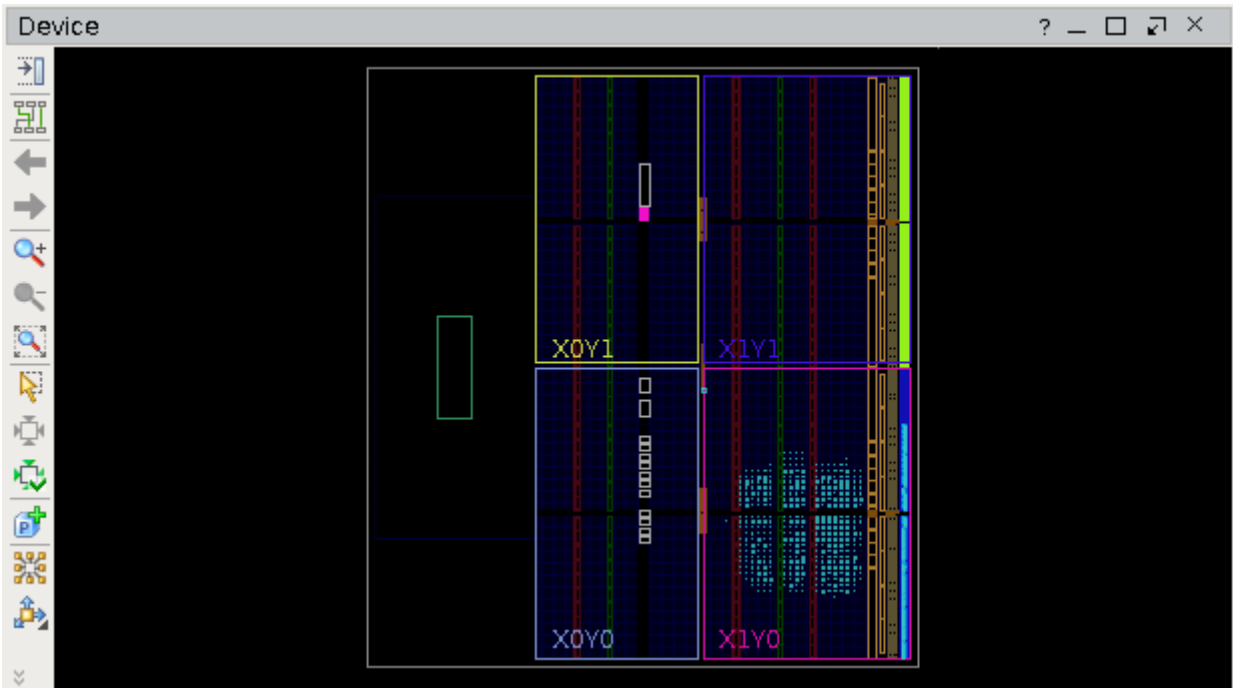
No teste bench fizemos a criação dos sinais respectivos as variáveis criadas na entidade. Fizemos o teste para dois testes bench. Para o primeiro fixamos a entrada de dados juntamente com o clock, o load e o endereço. O código abaixo representa o processo de leitura da memória. O endereço foi fixado em “111111” e a entrada de dados em “0FF0”. O clock e o load em 0. O clock alterna o sinal depois de 1 ns. A simulação 1 representa a leitura referente a este teste bench. A ideia é ler o endereço fixado da memória e jogar na saída.

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

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

architecture Behavioral of N_RAM_TB is
       component N_ram is
        Port (  input   : in std_logic_vector( 15 downto 0);
            clk     : in std_logic;
            load    : in std_logic;
            address : in std_logic_vector(5 downto 0);
            output  : out std_logic_vector(15 downto 0));
        end component;
 ------------------------------------------------------------------------

         --LENDO  O ENDEREÇO : 111111 DA MEMORIA E JOGANDO  NA SAIDA

 ------------------------------------------------------------------------------------------
       signal s_input  :  std_logic_Vector(15 downto 0) := x"0FF0";
       signal s_output : std_logic_Vector(15 downto 0);
       signal s_address : std_logic_vector( 5 downto 0):= "111111";
       signal s_clk : std_logic  := '0';
       signal s_load : std_logic := '0';

      begin

      UTT: N_RAM port map(s_input, s_clk,s_load, s_address,s_output);

               s_clk <= not s_clk after 1ns;

        ----carrega a posicao de memoria 0000010 na saida       --- pulso mantem o
```

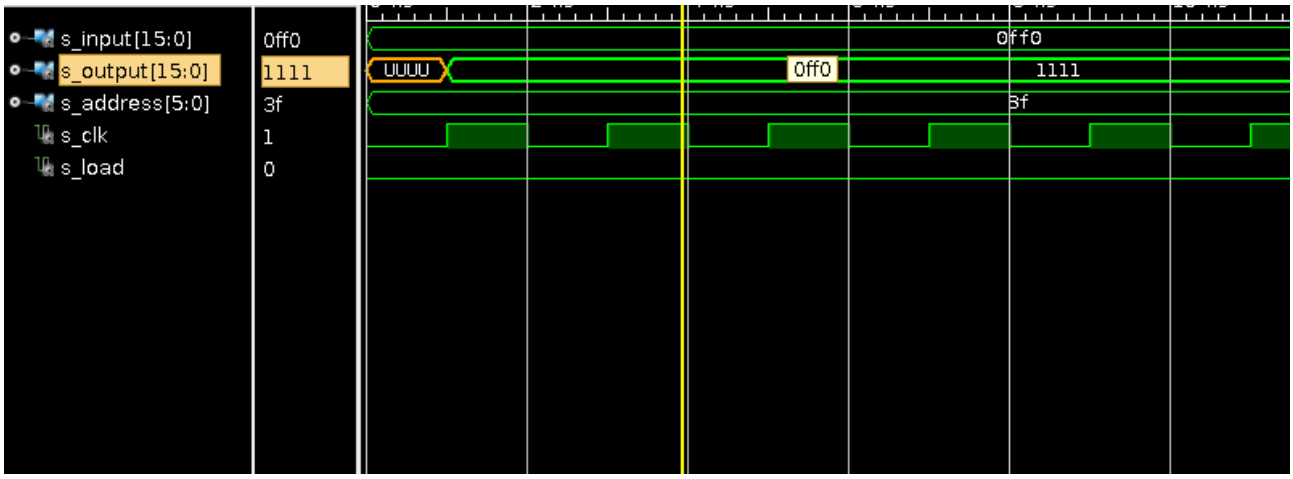
Teste bench

No segundo teste bench fizemos o processo de escrita. Os mesmos sinais referentes a cada variável da entidade foi criada semelhante ao teste bench passado. A ideia agora é escrever no endereço fixado de memória e jogar na saída. Temos a alternância do clock depois de 1 ns. O sinal de load vale 1 e logo depois de 2 ns ele recebe 0. Temos o carregamento da posição de memória na saída. A simulação 2 representa a simulação de escrita.

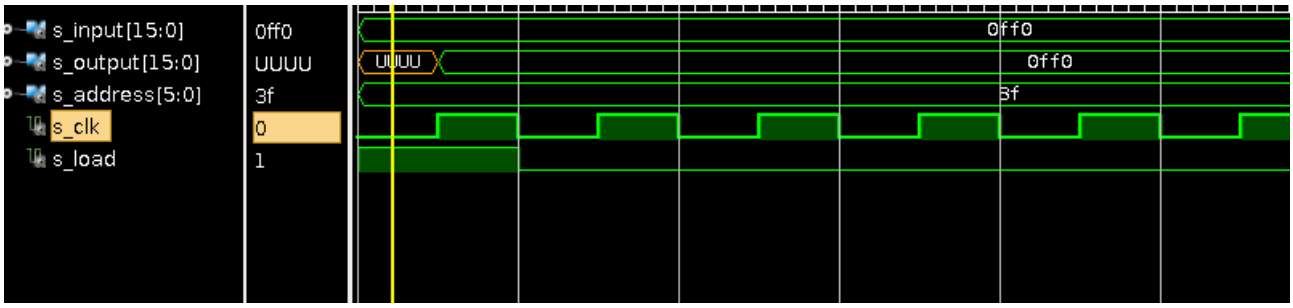
```
entity N_RAM_TB is
--  Port ( );
end N_RAM_TB;

architecture Behavioral of N_RAM_TB is
       component N_ram is
        Port (  input   : in std_logic_vector( 15 downto 0);
            clk     : in std_logic;
            load    : in std_logic;
            address : in std_logic_vector(5 downto 0);
            output  : out std_logic_vector(15 downto 0));
        end component;
 ------------------------------------------------------------------------

         --ESCREVER   NO ENDEREÇO : llllll DA MEMORIA E JOGANDO  NA SAIDA

-------------------------------------------------------------------------------------------
       signal s_input  :  std_logic_Vector(15 downto 0) := x"0FF0";
       signal s_output : std_logic_Vector(15 downto 0);
       signal s_address : std_logic_vector( 5 downto 0):= "111111";
       signal s_clk : std_logic  := '0';
       signal s_load : std_logic := '0';

     begin

     UTT: N_RAM port map(s_input, s_clk,s_load, s_address,s_output);
               s_load <= '1', '0' after 2ns;
              s_clk <= not s_clk after 1ns;

        ----carrega a posicao de memoria 0000010 na saida       --- pulso mantem o que esta


end Behavioral;
```



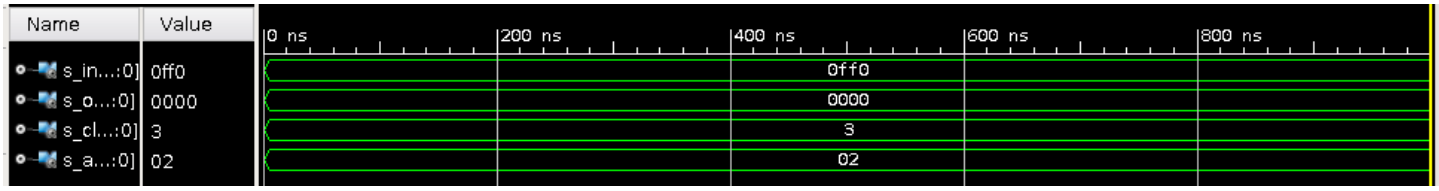
Implementation design



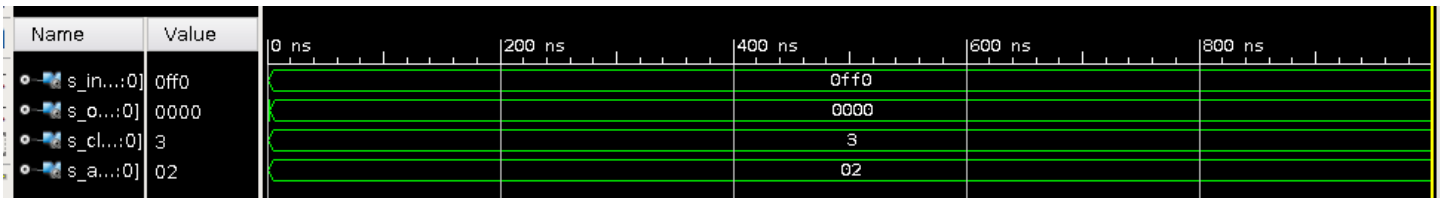
Simulação de comportamento 1 (Leitura de RAM)



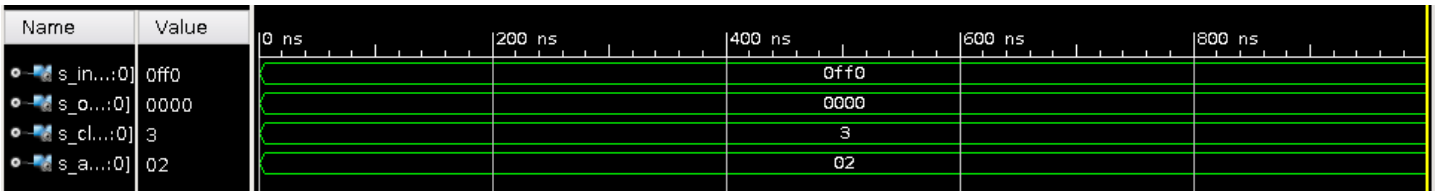
Simulação de comportamento 2 (Escrita na RAM)



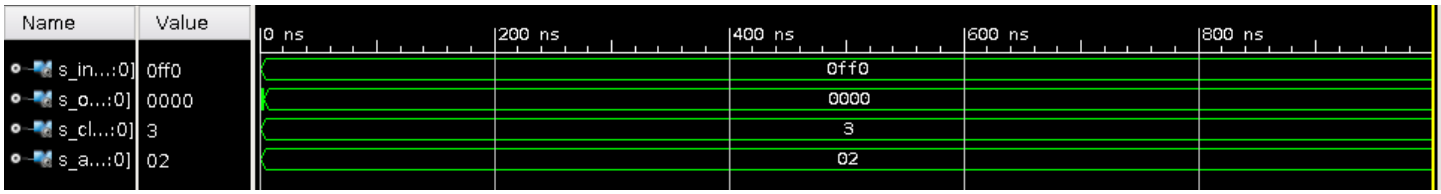
Synthesis functional simulation



Synthesis timing simulation



Implementation functional simulation



Implementation timing simulation

DIVISOR DE FREQUÊNCIA

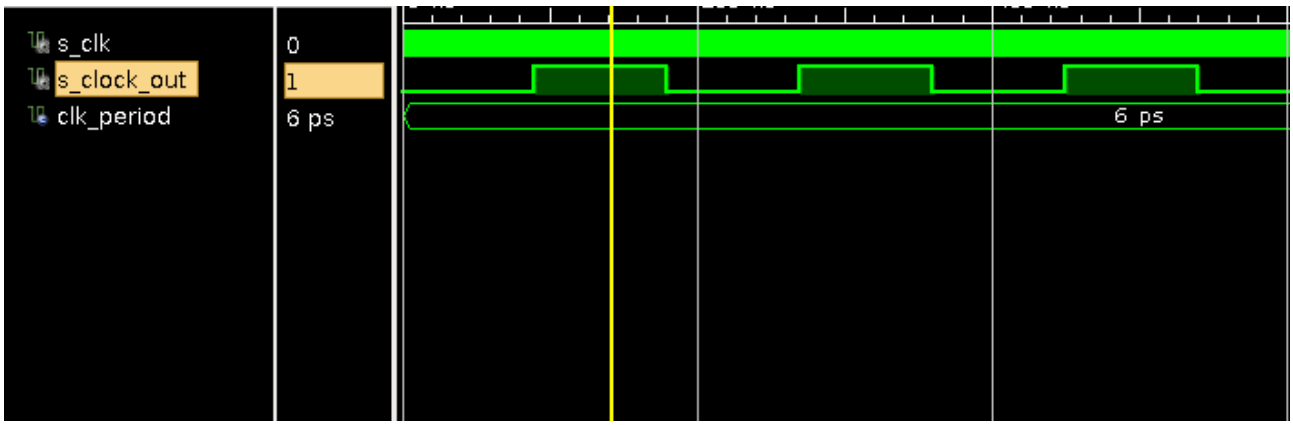
Para criamos o divisor de frequênica, declaramos um clock de entrada – clk- e um clock de saída – clock_out – que serão trabalhados na arquitetura. Logo após isso, iniciamos nossa arquitetura criando dois sinais : o primeiro é um contador – count – e o segundo é uma sinal std_logic. Após estas declarações, criamos um process que será sensível às mudanças de clock – clk . Com isso, cada vez que houver um evento de clock e ele for clk = '1', então, o contador – count- irá incrementar em 1 unidade de tempo até que se deseje. Caso o contador chegue a uma quantidade determinada, ele recebe 1 e o sinal tmp alterna. Como a saída recebe tmp, logo a saída do clock terá uma frequência menor. No exemplos, posteriores não foi dada uma frequência de saída em 1MHz em razão da demora da simulação, porém, essa questão tão pouco importa, visto que, aumentando o valor limite para o contador a frequência de saída será menor ou maior, regulando assim como desejar, no caso, para atingir uma frequência 1MHz na placa zybo teríamos que colocar o valor 75 000 000 como limite para contador. Na figura abaixo vemos o código para este divisor de frequência:

```
entity div_clock is
  Port (
            clk         :     in std_logic;
            clock_out          :     out std_logic
        );
end div_clock;

architecture Behavioral of div_clock is

signal count: integer := 1;
signal tmp  : std_logic := '0';

begin

process(clk)

    begin

    if(clk'event and clk='1') then
        count <= count+1;

    if (count = 7500) then
        tmp <= not tmp;
        count <= 1;

    end if;
    end if;
        clock_out <= tmp;
end process;

end Behavioral;
```

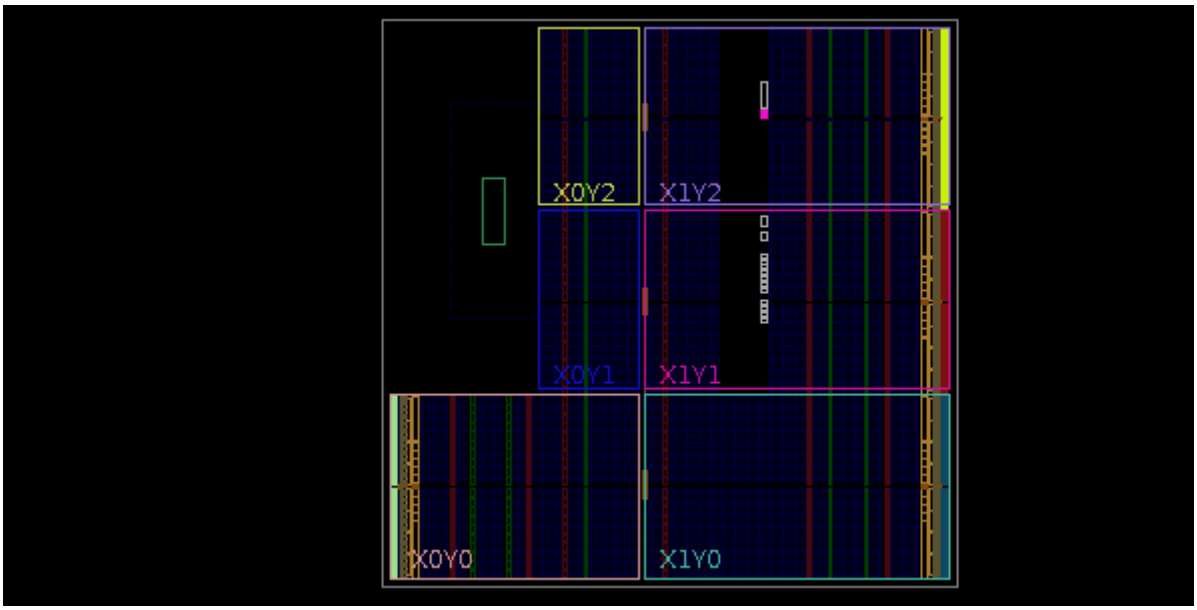
DIVISOR DE FREQUÊNCIA (Teste Bench)

Para a criação dos testes, procedemos com a criação dos componentes que representarão a entity anterior. Criamos um sinal – s_clk- e outro chamado de s_clock_out. Eles mapearão ckl e clock_out do divisor de frequência, respectivamente. Criamos uma contante chamada de clk_periodo, e representará o periodo de clock de entrada. Ou seja, uma placa que tenha um clock de 150 Mhz, devemos ter clk_periodo = 6, 66 ns. Desta forma, conseguiremos dividir e transformar 150 Mhz em outra frequência, no caso, 1Mhz. Na simulação foi usado um período de 6ps a título de simulação e facilitação de visualização dos resultados. Caso quiséssemos, realmente, uma frequência de 1Mhz, alteraríamos o valor do clk_período para 666ps. Após fazermos o mapeamento das entradas e saídas, iniciamos um procss que alternará os pulsos de clock – s_clk – após clk_período. Assim, após sucessivas subidas de clock de entrada, teremos algumas subidas no clock de saída. Na figura abaixo vemos o código deste e a respectiva simulação:

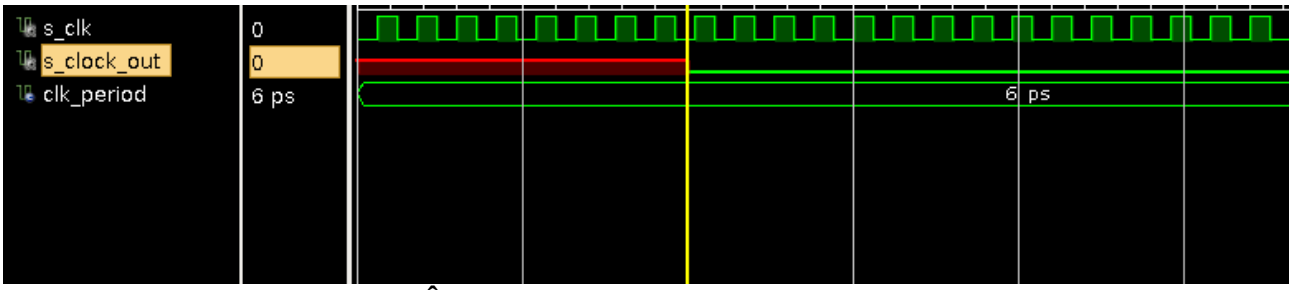
```
architecture Behavioral_tb of Tb_div_clock is
-- Component Declaration for the Unit Under Test (UUT)
    component div_clock is
    port(
         clk : in std_logic;
          clock_out : out std_logic
        );
end component;

--INPUTS
signal s_clk : std_logic := '0';

--OUTPUTS
signal s_clock_out : std_logic := '0';

-- Clock period definitions
constant clk_period : time := 6ps;

     begin

         teste:  div_clock port map(
                                    clk => s_clk, -- entrada 1 atribui seu valor ao sinal de entrada 1;
                                    clock_out => s_clock_out -- saída 1 atribui seu valor ao sinal de saída 1;
                                   );

         p: process --label
         begin
          wait for clk_period;
         s_clk  <=  not s_clk ;

         end process p;
```



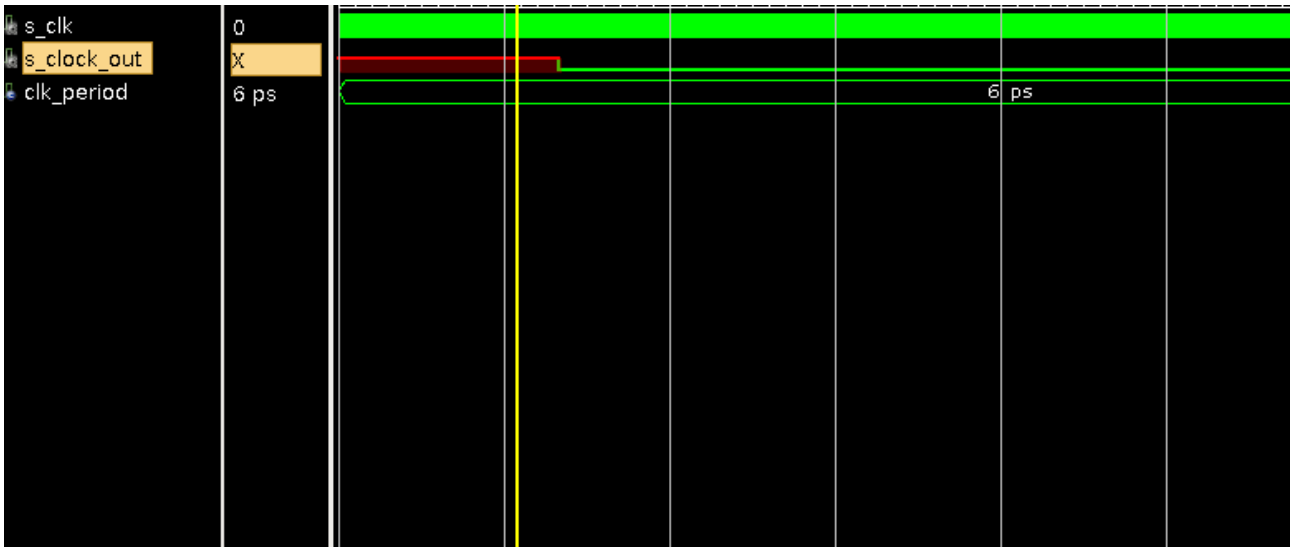
DIVISOR DE FREQUÊCIA : FUNCIONAL SIMULATION



IMPLEMENTATION DESIGN



DIVISOR DE FREQUÊCIA : IMPLEMENTATION TIMING SIMULATION



DIVISOR DE FREQUÊCIA : IMPLEMENTATION FUNCIONAL SIMULATION

#Contador BCD

Para a realização do contador BCD utilizamos a mesma ideia do divisor de frequência. Na entidade fizemos a criação de um clear, um clock, um enable e uma saída de tamanho 3. Na arquitetura criamos um sinal auxiliar “s_conta”. Fizemos a criação dos mesmos sinais do divisor de frequência logo em seguida.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
entity CONTADOR is
    Port ( clr, clk     : in std_logic;                     -- clear e clock
           Enable       : in std_logic;                     -- enable
           Q            : out std_logic_vector(3 downto 0)); -- saida Q
end CONTADOR;

architecture Behavioral of CONTADOR is
    Signal s_conta : signed ( 3 downto 0) := X"0";

---------------------------- codigo do divisor de clock------------
    signal count   :    integer := 1;          --- sinal que conta o numero de pulsos já dados
    signal clk_out :    std_logic := '0';      --- sinal que responde para o processo contador  um sinal de clock
    signal tmp     :    std_logic := '0';      --- sinal que recebe o pulso de clock
```

Em seguida fizemos a criação de um processo com a mesma ideia do divisor de frequência, com a única diferença de ter mudado o valor da frequência para facilitar a visualização da simulação.

```
 begin

process(clk)
    begin

    if(clk'event and clk='1') then
        count <= count+1;
        if (count = 75) then                --- alteramos a frequência para facilitar a visualização na simulacao
        tmp <= not tmp;                     --- a frequência desejada pode ser conseguida aumentado o valor no if
        count <= 1;
    end if;
    end if;
    clk_out <= tmp;
end process;
```

Depois fizemos a criação de outro processo sensível ao clock. Se houver a ocorrência de clock e o clear estiver ativo em 1 a saída final é zerada. Se o enable estiver em 1, então o sinal auxiliar “s_conta” recebe 9 e depois volta pra zero, fazendo “s_conta” igual a 0. Se essas condições anteriores não forem satisfeitas, então o contador continua incrementando. No final o contador é transferido para a saída.

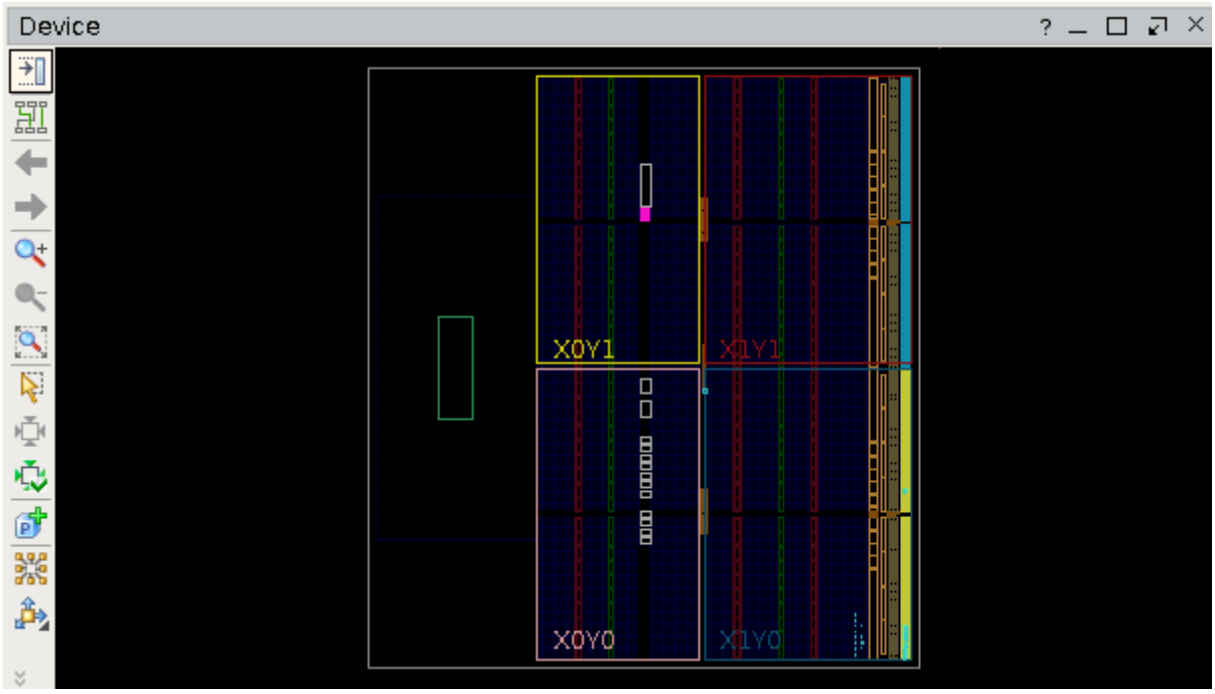
```
---------------Codigo afetado pelo divisor de clock---------------------------
    Contando: process(clk_out)                      ---- em razao do processo anterior, a frequencia de clock foi diminuida
     begin
       if clk_out'event and clk_out = '1' then                  ------  evento de clock ocorrer
           if clr = '1' then  s_conta <= X"0";                 ------- caso clear esteja em 1, tudo zera
           elsif Enable = '1' then                               ----- se não , caso enbale esteja em 1
               if s_conta = "1001" then                        --- se ja for nove, ele volta pra zero
                  s_conta <= X"0";
               else s_conta <= s_conta + 1;                         --- se nao for nove, incrementa
                                                                    -- ao final transfere pra SAIDA Q
               end if;
          end if;
      end if;
    end process;
    Q <= std_logic_vector(signed(s_conta));
end Behavioral;
```

No teste bench criamos sinais referentes a cada variável criada na entidade. Fixamos as entradas de clock, load e enable. Em seguida fizemos o mapeamento. Fizemos o clock variar e o enable zerar depois de 666 ps. O teste bench e as simulações seguem abaixo:

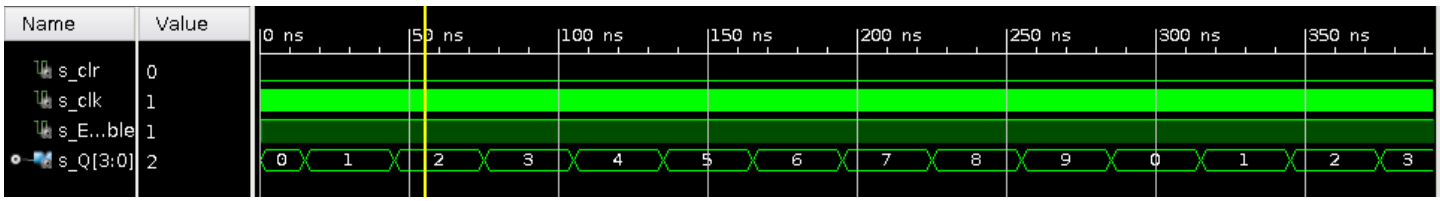
```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

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

architecture Behavioral_tb of contador_tb is
  component CONTADOR is
        Port ( clr, clk : in std_logic;                          -- entrada de clear e clock
           Enable       : in std_logic;                          -- entrada de enbabel
           Q            : out std_logic_vector(3 downto 0));     -- saida de dados
    end component;

   signal s_clr     : std_logic := '0';              -- sinal que representa o clear
   signal s_clk     : std_logic := '0';              -- sinal que representa o clear
   signal s_Enable  : std_logic := '1';              -- sinal que representa o enable
   signal s_Q       : std_logic_vector(3 downto 0);  -- sinal que representa a saida de dados
```

```vhdl
begin
    UTT: CONTADOR Port map( clr => s_clr, enable => s_enable, clk => s_clk, Q => s_Q);
        P: process
            begin
            wait for 100 ps;
            s_clk <= not s_clk;
            --s_clk <= not s_clk after 666 ps;    --- codigo que faz o pulso de clock
            --s_enable <= '1' after 666 ps;

        end process;

end Behavioral_tb;
```

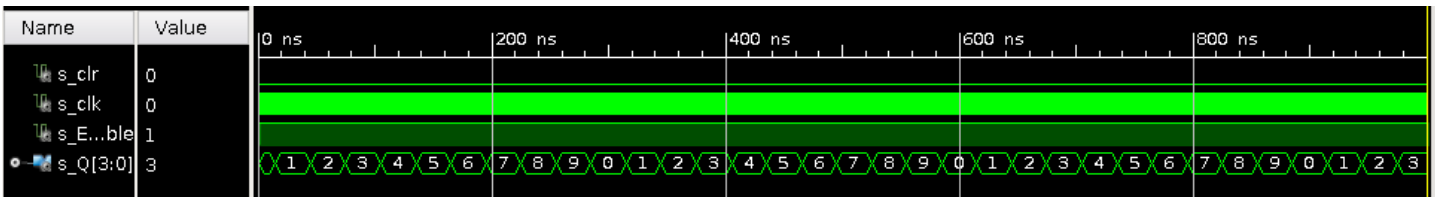
Teste bench



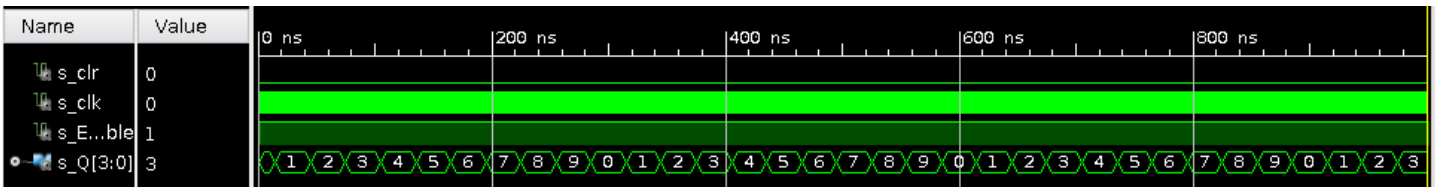
Implementation design



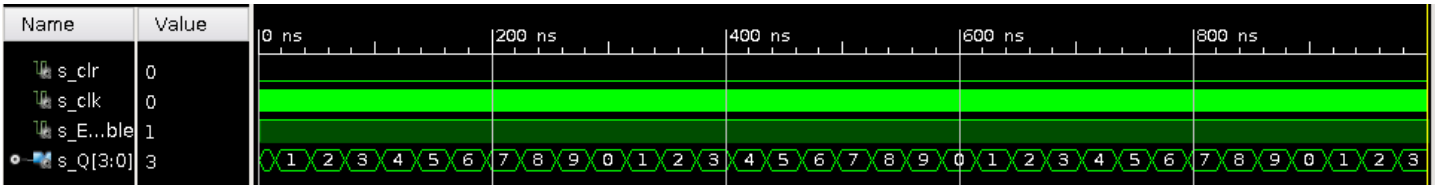
Simulação de comportamento



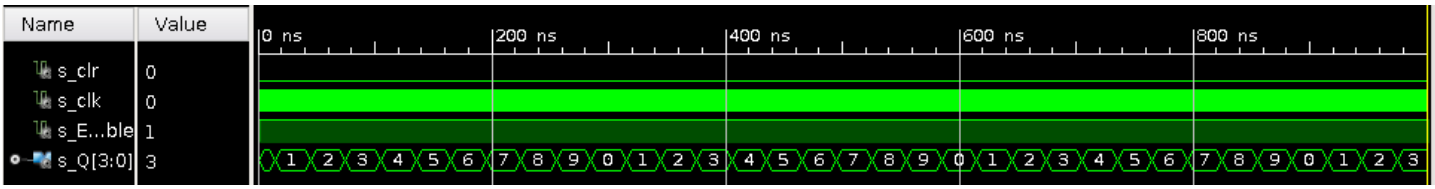
Synthesis functional simulation



Synthesis timing simulation



Implementation functional simulation



Implementation timing simulation

DENTE DE SERRA

Para fazer o dente de serra fizemos a criação de um clock, um reset e um vetor de saída de 8 posições na entidade. Fizemos um sinal auxiliar para servir de contador.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity DENTE_SERRA is

     Port ( clk : in std_logic;                                  -- clock;
            rst : in std_logic;                                  -- reset;
            out_wave : out std_logic_Vector(7 downto 0) );   -- saída;
end DENTE_SERRA;

architecture Behavioral of DENTE_SERRA is
    signal cont : integer := 0;                                  -- contador;
```

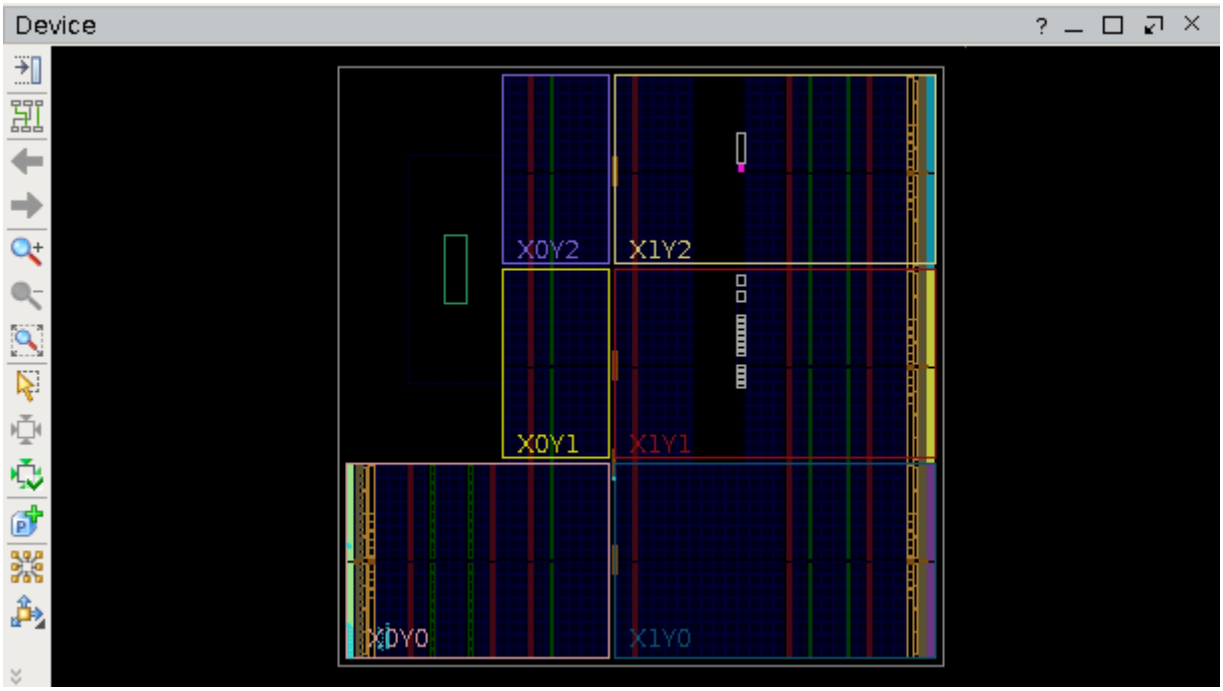
Na arquitetura fizemos um processo sensível ao clock e ao reset. Se o reset estiver em 1 tudo é zerado na saída. Se houver borda de subida e o contador estiver em 255 na contagem então o contador é zerado novamente para começar a contagem. Obrigatoriamente em algum momento entraremos nessa condição, pois logo a seguir existe um eles if caso o contador não esteja em 255. No final a saída recebe o contador e 8.

```
begin
    process(clk, rst)

    begin
        if rst = '1' then -- se reset for 1 então:
            cont <= 0;
        elsif clk'event and clk = '1' then -- se houver borda de subida;
            if ( cont = 255) then
                cont <= 0;
            else
              cont <= cont + 1;            -- incrementa contador;
           end if;
    end if;
    end process;
    out_wave <= conv_std_logic_vector(cont,8);  -- saída recebe contador e 8;
end Behavioral;
```

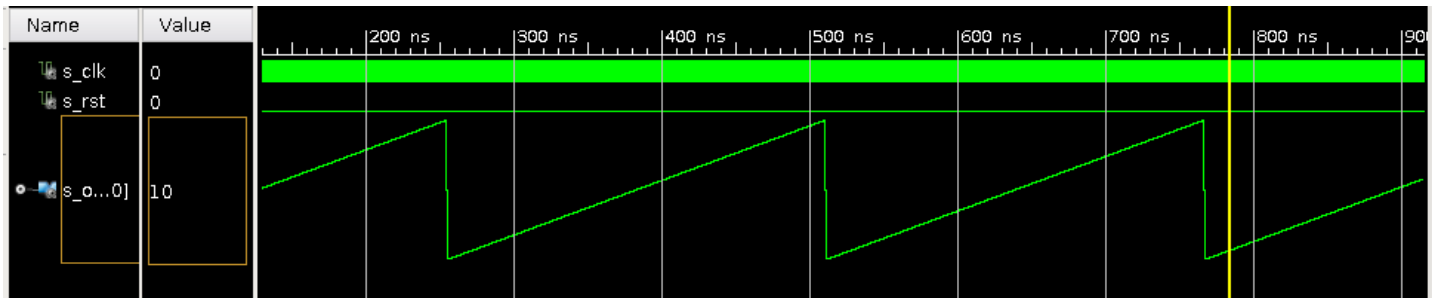
No teste bench realizamos a criação de sinais correspondentes a cada uma das variáveis criadas na entidade. Fizemos o mapeamento em seguida e ficamos alternando o clock dentro do processo para melhor visualização de simulação. Em algumas simulações aconteceu ruído. Talvez esses ruídos sejam devido ao curto tempo que foi fixado no teste bench. O mesmo acontece para o contador triangular. O teste bench e as simulações seguem abaixo:

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

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

architecture Behaviora_tb of dente_serra_tb is
  component dente_serra is
    Port ( clk : in std_logic; -- clock;
           rst : in std_logic;  -- reset;
           out_wave : out std_logic_Vector( 7 downto 0) ); -- saída
    end component;

   signal s_clk : std_logic := '0'; -- sinal para o clock;
   signal s_rst : std_logic := '0'; -- sinal para o reset;
   signal s_out_wave : std_logic_vector( 7 downto 0) ; -- sinal para a saída;

begin
    UTT: dente_serra port map (s_clk, s_rst,s_out_wave); -- mapeamento;
    process
    begin
        wait for 500 ps;
        s_clk <= not s_clk; -- varia o clock;
    end process;

end Behaviora_tb;
```

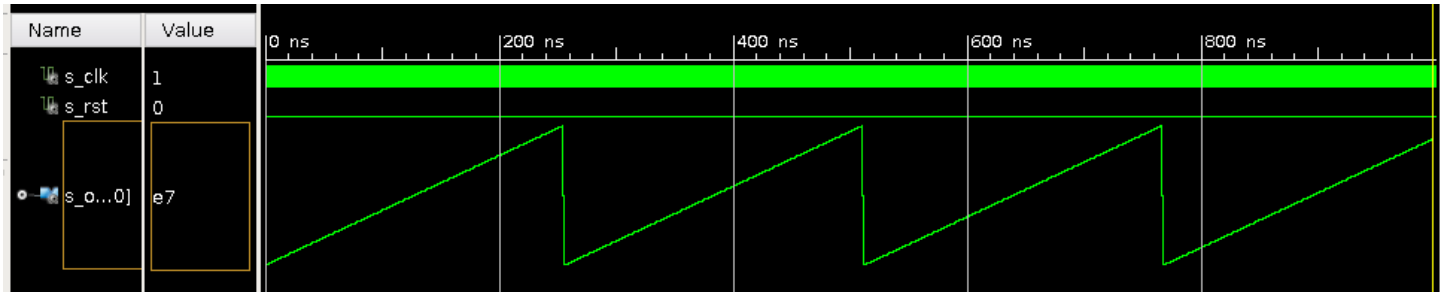
Teste bench



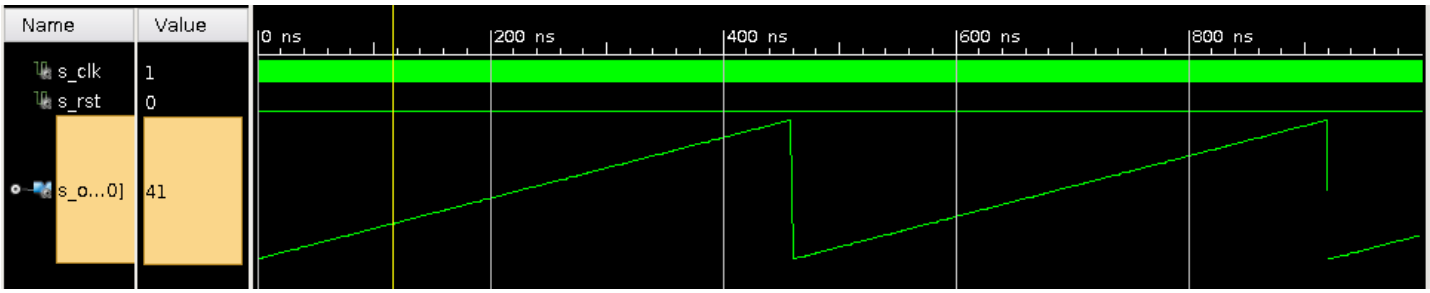
Implementation design



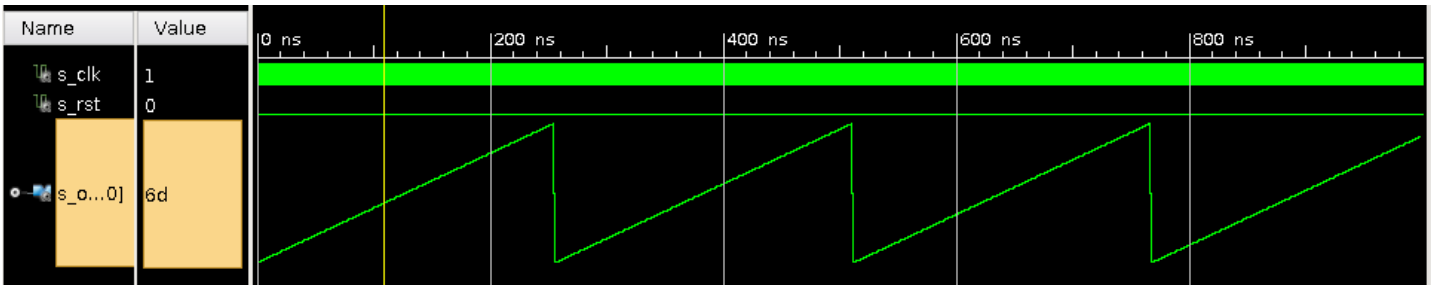
Simulação de comportamento



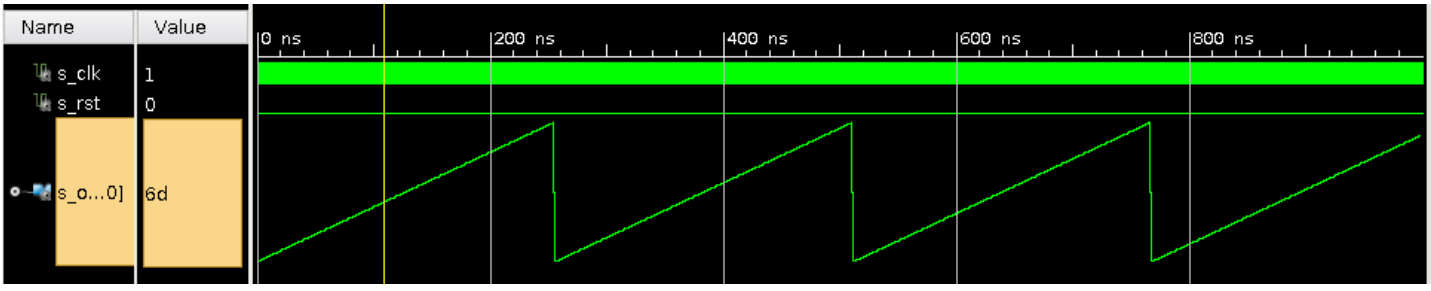
Synthesis functional simulation



Synthesis timing simulation (tempo maior)



Implementation functional simulation



Implementation timing simulation

CONTADOR TRIANGULAR

Para fazer o contador triangular fizemos a criação do clock, reset e de uma saída. Fizemos uma variável auxiliar para nos orientar na direção do sinal.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity triangular is
    Port ( clk :   in STD_LOGIC;     -- clock;
           rst:   in std_logic;     -- reset;
          saida : out STD_LOGIC_VECTOR (7 downto 0));  -- saida;
end triangular;

architecture Behavioral of triangular is
signal direcao : std_logic:= '0';       -- variavel de direção
```

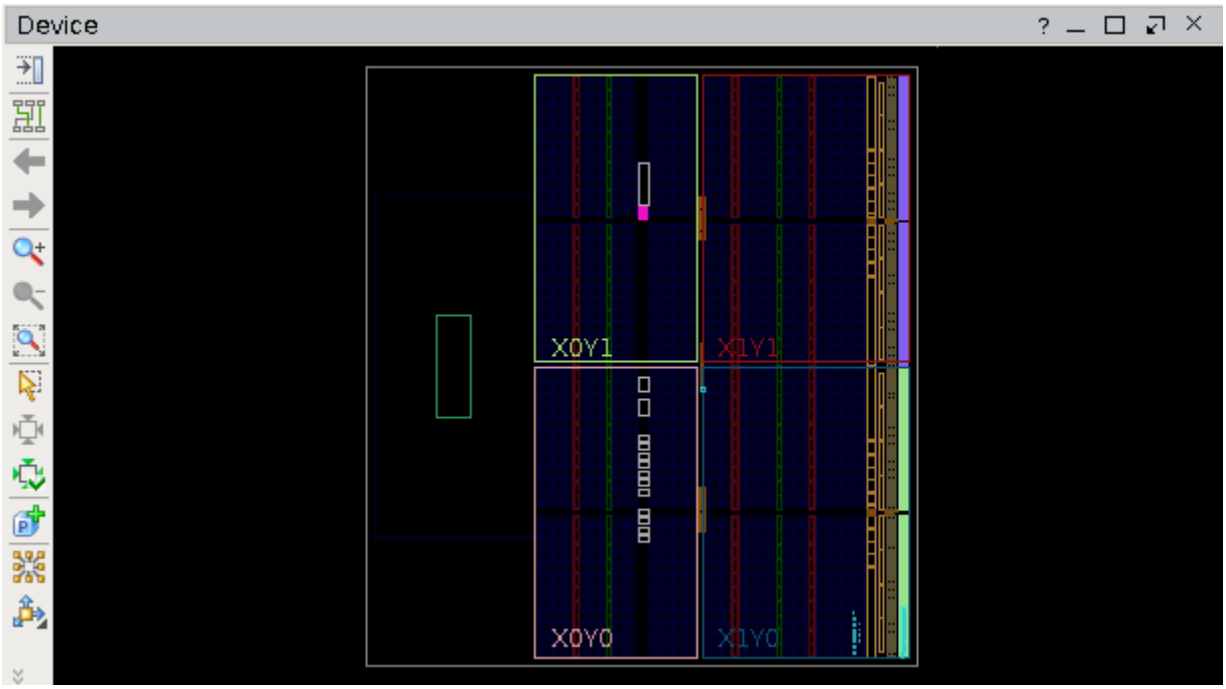
Dentro da arquitetura nós fizemos várias condições para que no final houvesse uma contagem e que a forma de onde final analógica seguisse um triângulo de contagem. Se o reset for igual a 1, então o contador é zerado e a variável assume o valor de 1. Se a direção tiver o valor de 0 e o contador estiver numa contagem de 255 então a direção assume o valor de 1. Se essas condições não forem satisfeitas então o contador é incrementado logo em seguida. Se o contador estiver em 0 então o sinal de direção assume 0. Caso contrário, o contador é decrementado. No final a saída recebe o valor do contador e 8, semelhante ao dente de serra.

```
begin

    TRIANGULO: process( rst,clk)
                variable cont : integer:= 0;
                begin
                if rst = '1' then
                    cont := 0;
                    direcao<= '1';
                else
                if direcao = '0' then
                    if cont = 255 then
                    direcao <= '1';
                    else
                        cont := cont+ 1;
                    end if;
               else
                    if ( cont = 0) then
                        direcao<= '0';
                    else
                        cont := cont -1;
                    end if;
                end if;
              end if;
                saida <= conv_std_logic_vector(cont,8);
          end process;
end Behavioral;
```

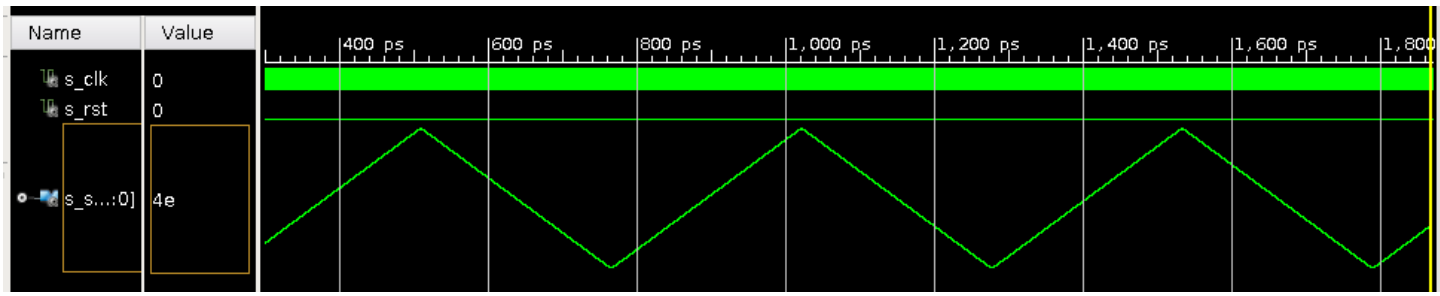
No teste bench fizemos a criação de sinais referentes a cada variável criada na entidade. Fixamos o clock e o reset em nível lógico 0 e depois fizemos o mapeamento. Fizemos o reset alternar logo em seguida juntamente com o clock. O teste bench e as simulações seguem abaixo:

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

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

architecture Behavioral of triangular_tb is
  component triangular is
        Port ( clk :   in STD_LOGIC;     -- clock;
               rst:   in std_logic;      -- reset;
              saida : out STD_LOGIC_VECTOR (7 downto 0)); -- saida;
              end component;

        signal s_clk : std_logic := '0'; -- sinal referente ao sinal de clock;
        signal s_rst : std_logic := '0'; -- sinal referente ao reset;
        signal s_saida : std_logic_vector(7 downto 0) ; -- sinal referente a saída;


begin
       UTT: triangular port map (s_clk, s_rst, s_saida); -- mapeamento
       s_rst <= '1', '0' after 255 ps;                   -- reset recebe 1, depois 0, depois de 255 ps;
       process
       begin
            wait for 1ps;
            s_clk <= not s_clk;                          -- clock alterna;
       end process;

end Behavioral;
```

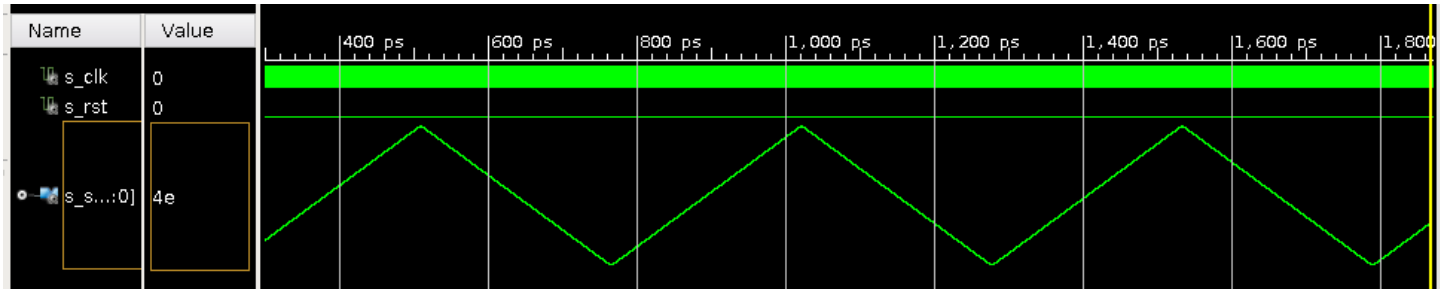
Teste bench



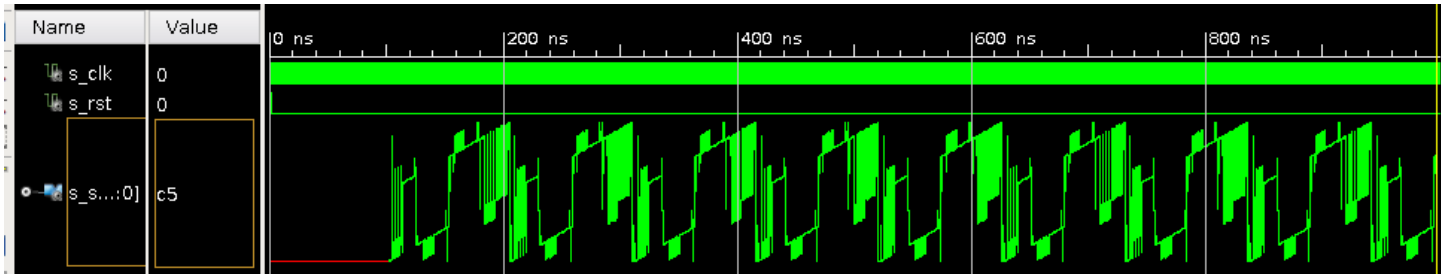
Implementation design



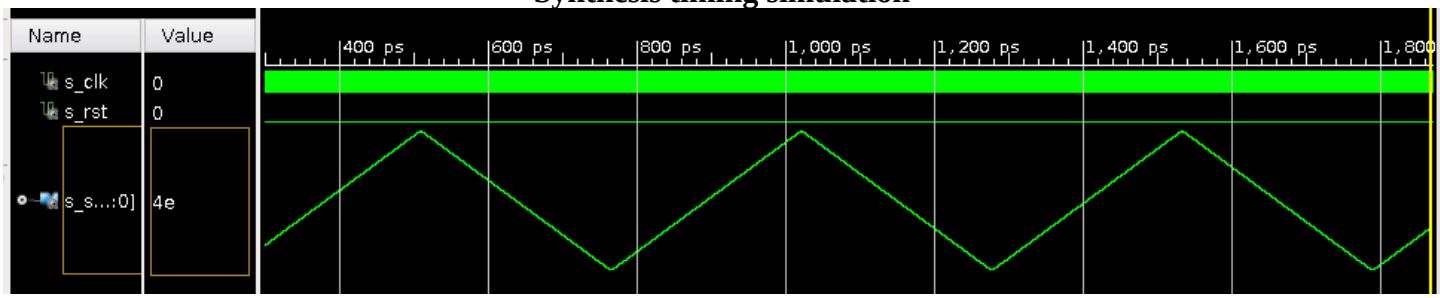
Simulação de comportamento



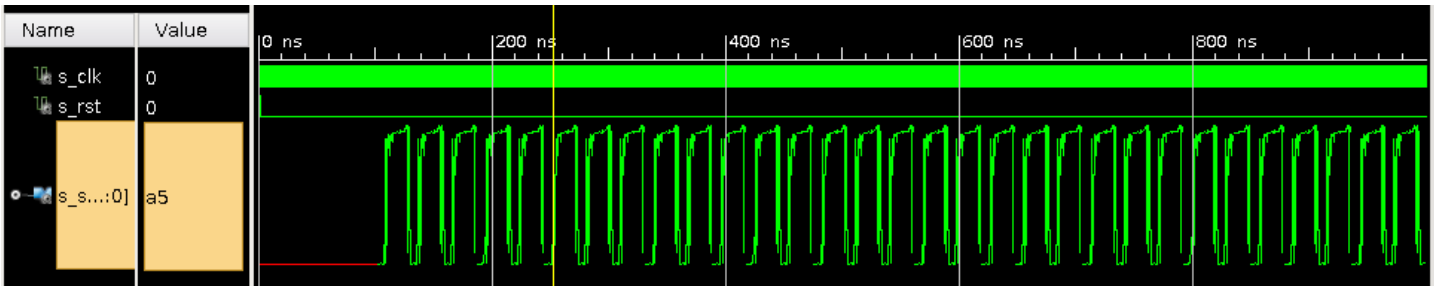
Synthesis functional simulation



Synthesis timing simulation



Implementation functional simulation



Implementation timing simulation

 # CONTADOR DE PROGRAMA

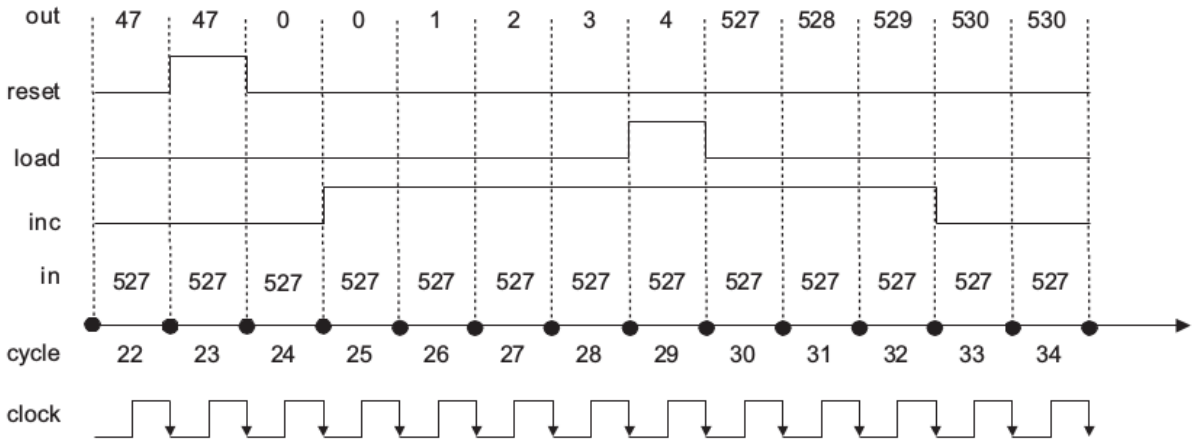
Para a realização desse projeto seguimos o padrão apresentado na página 51 do livro “The Elements of computer system”. Criamos na entidade uma entrada e uma saída de 16 bits cada. Um clock, um load, um reset e um incrementador. Criamos dois sinais para auxílio, que é um sinal para o registrador e outro para a descida. O sinal do registrador possui 16 bits.

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

entity COUNTER_PROGRAM is
     Port ( input : in std_logic_vector (15 downto 0);      -- entrada de dados de 16 bits;
            inc   : in std_logic ;                          -- incrementador;
            load  : in std_logic ;                          -- load;
            clk   : in std_logic ;                          -- clock;
            rst :  in std_logic ;                           -- reset;
            output : out std_logic_vector( 15 downto 0));   -- saída de dados de 16 bits;
end COUNTER_PROGRAM;

architecture Behavioral of COUNTER_PROGRAM is
    signal registrador  : std_logic_vector (15 downto 0):=x"0000"; -- sinal do registrador;
    signal descida       :std_logic := '0';                        -- sinal de descida;
```

Fizemos dois processos na arquitetura. O primeiro é sensível ao sinal de descida do clock. Fizemos com sinal de descida para deixar igual a que o livro apresenta. Se o reset estiver ativado em 1 então o registrador carrega zero. Se o load estiver em 1 a entrada é carregada no registrador. Se o valor do incrementador for 1 e o load estiver em 1 também então o valor do registrador é incrementado. No segundo processo fizemos com que cada vez que o clock fosse de descida o processo anterior decidisse o que fazer. Neste processo atual fizemos uma variação na variável auxiliar descida e colocamos o valor do registrador na saída de dados de 16 bits. Abaixo segue o modelo que o livro apresentou e o restante do programa em VHDL.

```
begin
   Carrega_Memoria: process(descida)
     begin
        if rst = '1' then            ---- se rst for 1, carrega o valor zero no registardor
            registrador <= x"0000";
        else
            if load = '1' then           --- se caso for zero, e load for 1, a entrada é carregado no registrador
               registrador <= input;

            elsif inc = '1' then          --- se load for zero,  e inc for 1 .entao o valor do registrador é incrementado;
                  registrador <= std_logic_vector(unsigned(registrador) + 1);
            end if;             --- caso o inc seja o, valor do registrador continua igual

        end if;
     end process Carrega_Memoria  ;

    Qualidade_Sincronia: process (clk)
    begin
        if clk'event and clk = '0' then
            descida <= not descida;           --- cada vez  que o clock descer o process carrega_memoria decide o que fazer
            output <= registrador;
         end if;
    end process Qualidade_Sincronia;


end Behavioral;
```

Restante do programa em VHDL



Exemplo proposto pelo livro

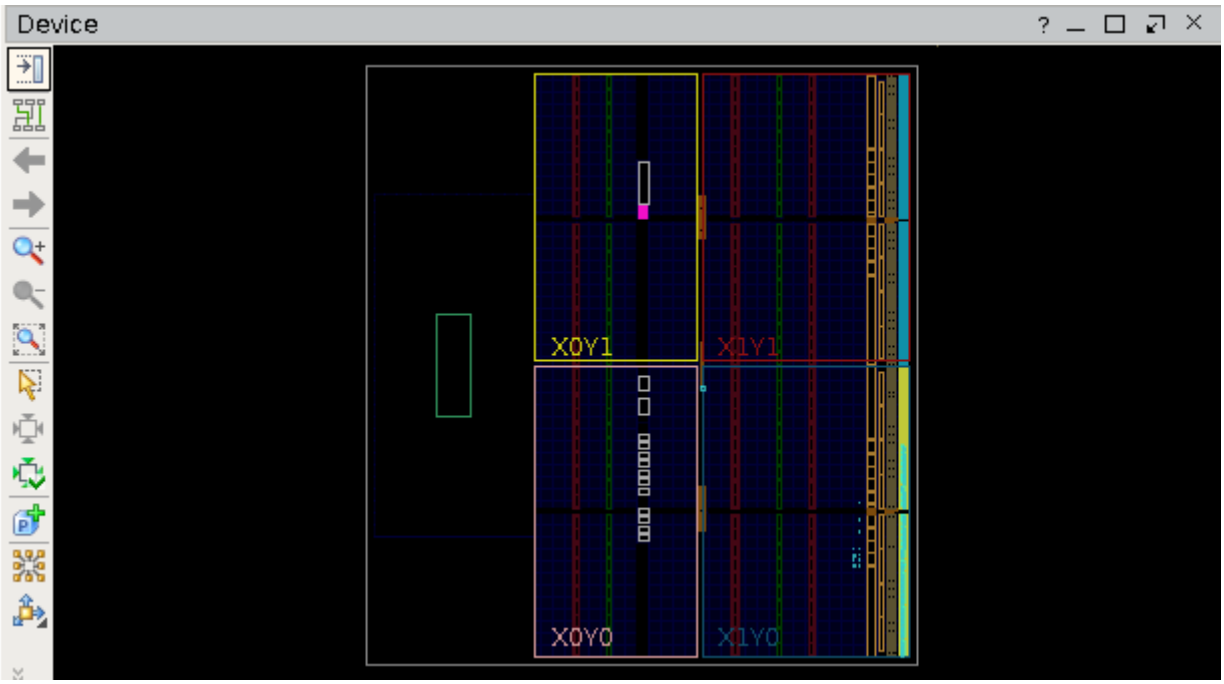
No teste bench fizemos a criação dos sinais referentes a cada variável criada na entidade. Fixamos a entrada de dados com o valor em hexa de “0527”. Fizemos o mapeamento logo em seguida. No processo ficamos alternando os sinais de clock, reset e inc em intervalos de tempo curtos para melhor visualização na simulação. O teste bench se encontra abaixo junto com as simulações:

```
    signal s_output: std_logic_vector(15 downto 0);         -- sinal de saída;
    signal s_reset : std_logic := '0';                      -- sinal para reset;
    signal s_load  : std_logic := '0';                      -- sinal para load;
    signal s_inc   : std_logic := '0';                      -- sinal para o incrementador;
    signal s_input : std_logic_vector(15 downto 0) := x"0527";  -- sinal para input;
    signal s_clk   : std_logic := '0';                       -- sinal de clock

begin

    UTT: COUNTER_PROGRAM port map(s_input, s_inc, s_load, s_clk, s_reset, s_output);
```

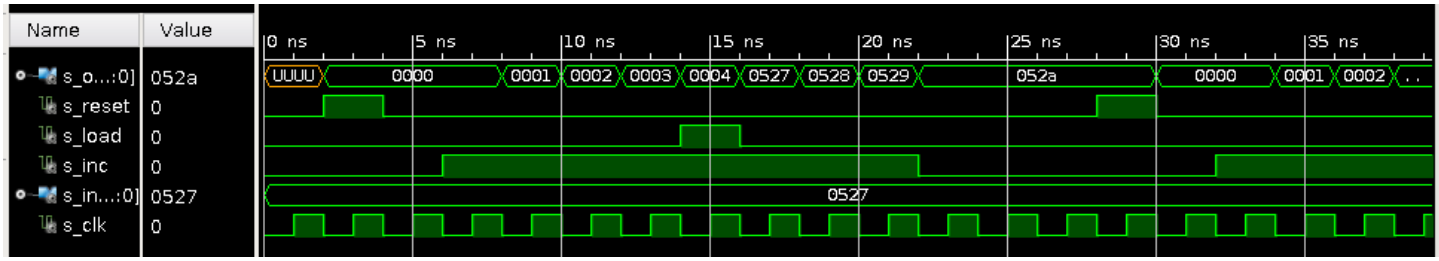
```
        wait for 1ns;
        s_clk <= '1';
        wait for 1ns;
        s_clk <= '0';
        s_inc <= '1';

        wait for 1ns;
        s_clk <= '1';
        wait for 1ns;
        s_clk <= '0';

        wait for 1ns;
        s_clk <= '1';
        wait for 1ns;
        s_clk <= '0';

        wait for 1ns;
        s_clk <= '1';
        wait for 1ns;
        s_clk <= '0';

        wait for 1ns;
        s_clk <= '1';
        wait for 1ns;
        s_clk <= '0';
        s_load <= '1';

        wait for 1ns;
        s_clk <= '1';
        wait for 1ns;
        s_clk <= '0';
        s_load <= '0';
```

```
        wait for 1ns;
        s_clk <= '1';
        wait for 1ns;
        s_clk <= '0';

        wait for 1ns;
        s_clk <= '1';
        wait for 1ns;
        s_clk <= '0';

        wait for 1ns;
        s_clk <= '1';
        wait for 1ns;
        s_clk <= '0';
        s_inc <= '0';

        wait for 1ns;
        s_clk <= '1';
        wait for 1ns;
        s_clk <= '0';

        wait for 1ns;
        s_clk <= '1';
        wait for 1ns;
        s_clk <= '0';
    end process;

end Behavioral;
```

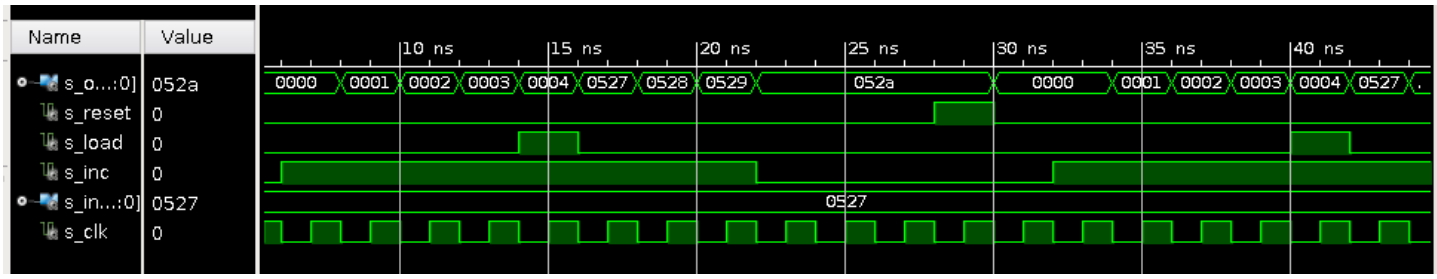
Teste bench



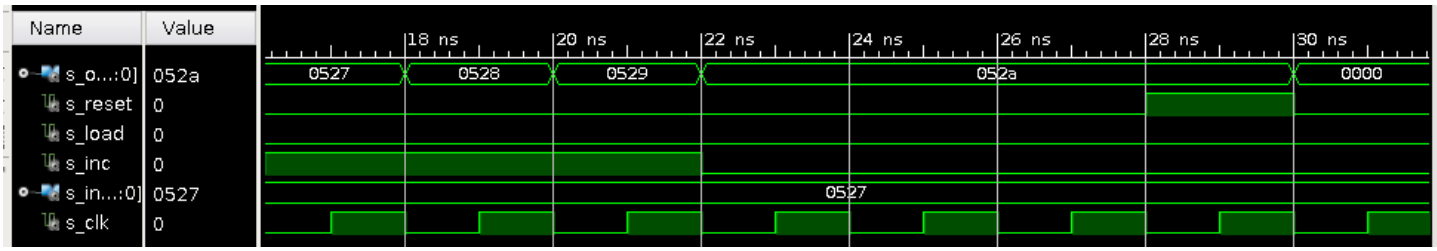
Implementation Design



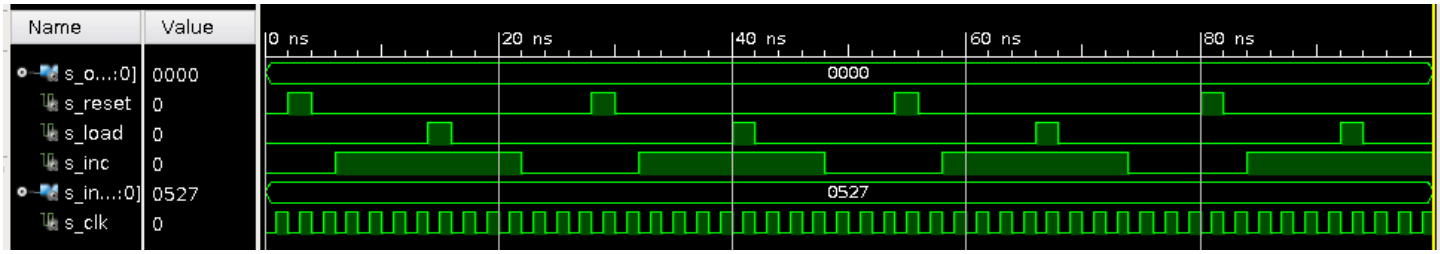
Simulação de comportamento



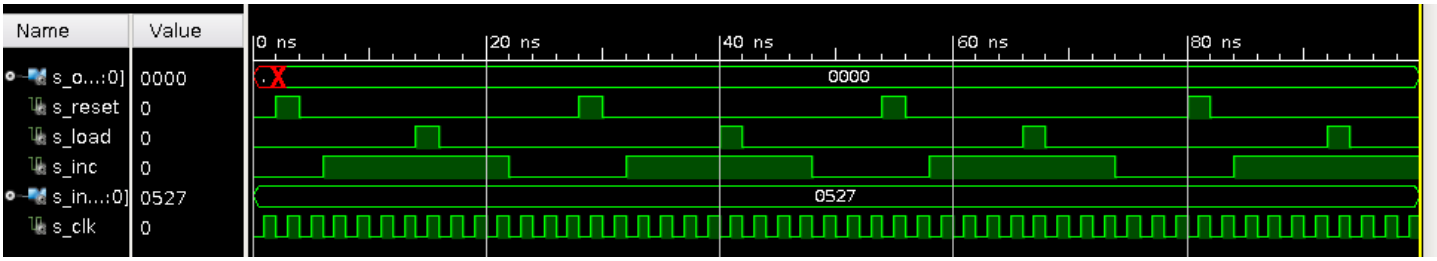
Synthesis functional simulation



Synthesis timing simulation



Implementation functional simulation



Implementation timing simulation

 # CONTADOR PWM

Foi criado 3 variáveis na entidade. Uma variável de clock, uma de saída e uma variável auxiliar. Foi criado uma variável do tipo contador na arquitetura, que na qual terá a função de fazer a contagem máxima para fazer uma comparação com outra variável contador no processo. O primeiro contador vai servir de referência na passagem de sinal alto. Foi criado um sinal “duty” de 8 bits na qual armazenará um valor de 30 em hexadecimal. Antes de entrar no processo duty recebe o valor da variável auxiliar da entidade. Dentro do processo um outro contador foi feito. Se houver subida de clock então o contador é incrementado. Se o nosso segundo contador for maior ou igual ao nosso contador de contagem máxima então a saída vai pra 0 e recomeça a contagem. No final se o contador for menos ou igual ao sinal de “duty” então a saída recebe 1. Caso contrário a saída recebe 0. Na simulação temos uma porcentagem de ativamento do sinal em cada barramento de bits. Por exemplo, nos primeiros 200 ns na qual nossa variável auxiliar contém um conjunto de bits em 1 temos o sinal de saída ativo apenas em uma determinada porcentagem nessa faixa. Na outra parte em que o a variável auxiliar está em um conjunto de 8 bits igual a 3 o sinal está numa porcentagem maior de ativamento, e assim em diante até a contagem ser recomeçada. O sinal vai assumindo níveis altos de duração diferentes em intervalos de tempos iguais de acordo com nossos contadores criados. O código em VHDL se encontra abaixo:

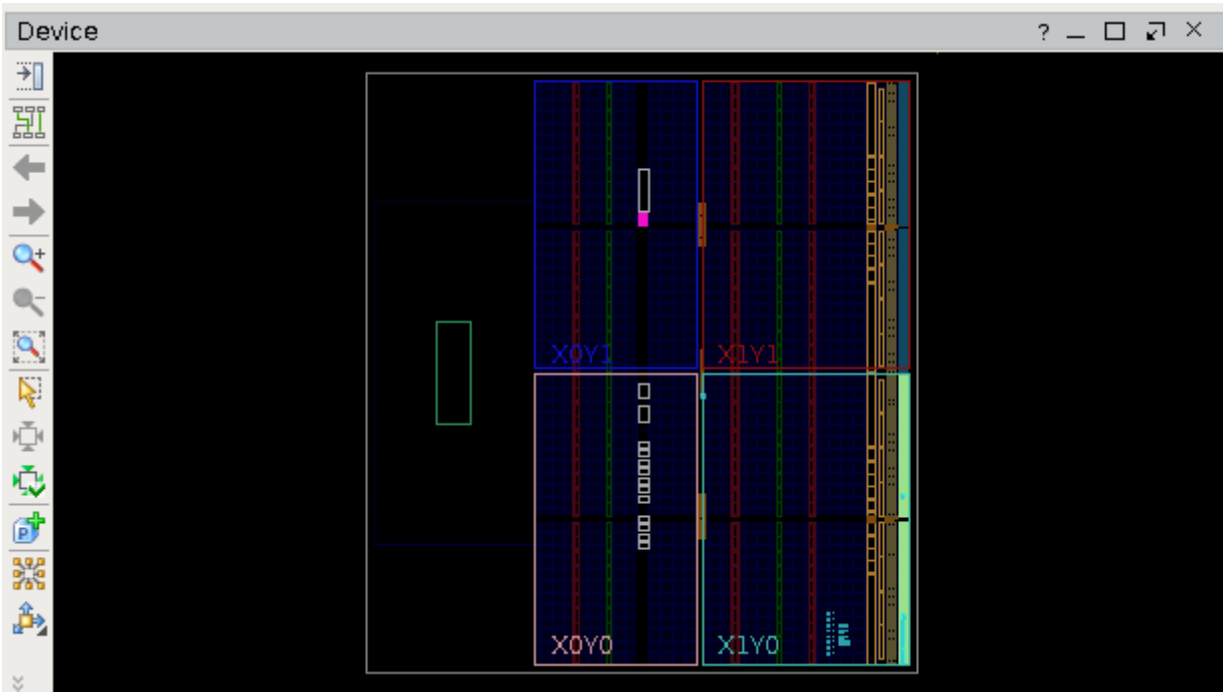
```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity pwm is
Port (
            clk : in std_logic;                     -- entrada de clock;
            aux : in unsigned(7 downto 0);          -- nível de referência auxiliar;
            out_pwm : out std_logic                 -- saída de dados;
      );
end pwm;

architecture Behavioral of PWM is

constant cont_max : integer := 10;  -- variável constante inteira que vai armazenar a contagem máxima de referência;

signal duty :  unsigned(7 downto 0) := X"30";     -- sinal de armazenamento de 8 bits;

begin

duty <= aux;     -- duty recebe o valor da variável auxiliar;

teste: process(clk,duty)

variable cont : integer := 0;  -- variável contador;

    begin
        if clk'event and clk = '1' then -- na subida de clock faça...
            cont := cont + 1;           -- incrementar o contador;
                if cont >= cont_max then cont := 0; -- Se o contador for igual ao contador máximo então recomeça a contagem;
            end if;
        end if;

    if cont <= duty then    -- se  contador for menor que duty então...
        out_pwm <= '1';     -- saída recebe 1;
    else                    -- Caso contrário...
        out_pwm <= '0';     -- saída recebe 0;
    end if;

end process teste;

end Behavioral;
```

Código em VHDL

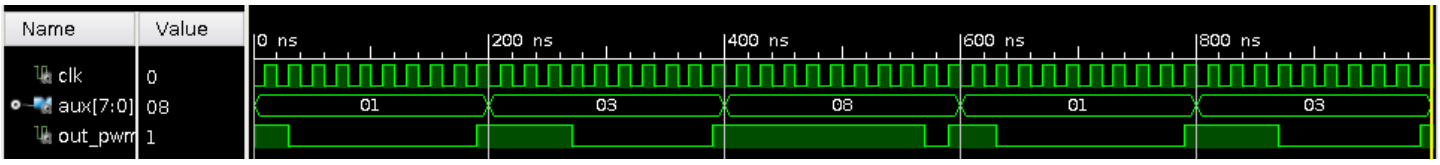
No teste bench foi criado sinais referentes a cada variável criada na arquitetura. Fizemos em seguida uma variação no clock depois de 10 ns. Depois foi definido uma espera de 200 ns para a variável auxiliar ficar alternando entre 3 valores fixados. O teste bench e as simulações se encontram abaixo.

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

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

architecture Behavioral of pwm_tb is
component pwm is
    Port (
            clk           : in std_logic;                 -- entrada de clock;
            aux           : in unsigned(7 downto 0);      -- nível de referência auxiliar;
            out_pwm       : out std_logic                 -- saída de dados;
          );
end component;

    signal clk          : std_logic := '0';                   -- sinal para o clock;
    signal aux          : unsigned(7 downto 0) := X"01";      -- sinal para variável auxiliar;
    signal out_pwm      : std_logic;                          -- sinal para saída de dados;

begin

UUT: pwm port map(clk,aux,out_pwm);     -- mapeamento

    clk <= not clk after 10 ns;  -- clock é barrado depois de 10 ns;

teste: process
    begin
      wait for 200ns;
      aux <= X"03";             -- variável auxiliar num barramento de 8 bits com valor 3;
      wait for 200ns;
      aux <= X"08";             -- variável auxiliar num barramento de 8 bits com valor 8;
      wait for 200ns;
      aux <= X"01";             -- variável auxiliar num barramento de 8 bits com valor 1;
end process teste;

end Behavioral;
```

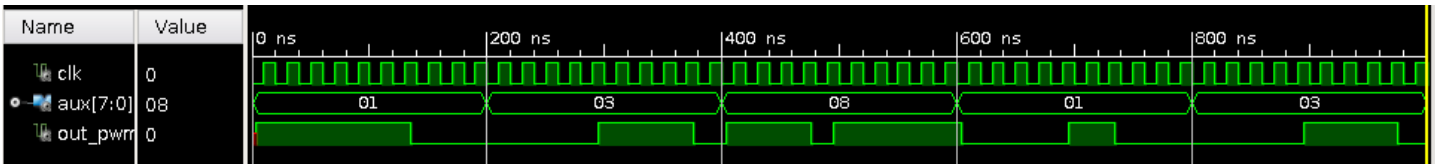
Teste bench



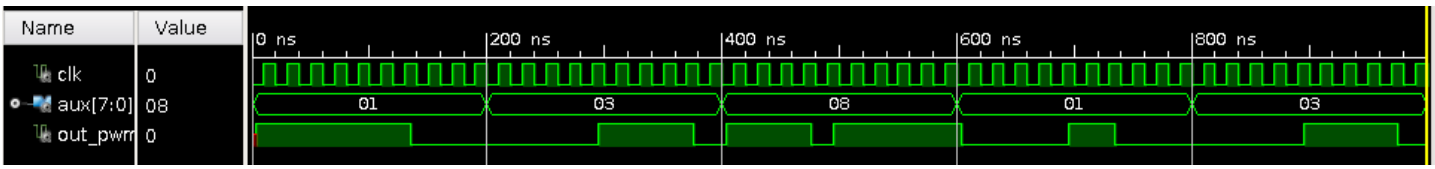
Implementation design



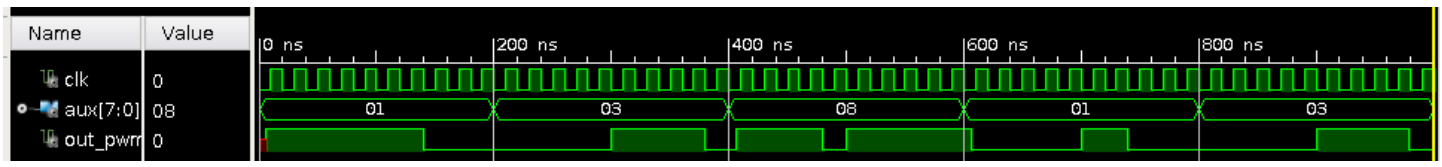
Simulação de comportamento



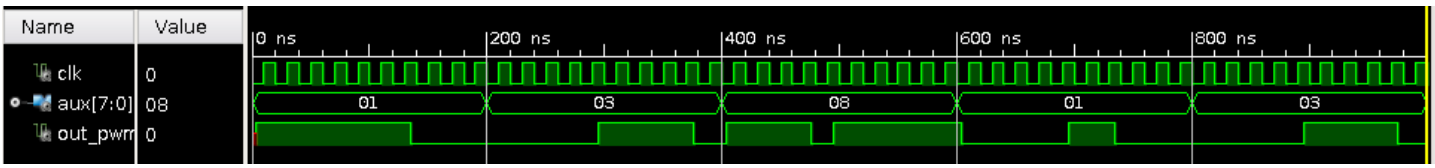
Synthesis functional simulation



Synthesis timing simulation



Implementation timing simulation



Implementation functional simulation