Universidade de Aveiro

Arquiteturas de Alto Desempenho

trabalho nr.2

Guilherme Duarte (107766), Guilherme Andrade (107696)

Departamento de Eletrónica, Telecomunicações e Informática

December 27, 2024

List of Figures

Introdução

Este projeto integra a disciplina de Arquiteturas de Alto Desempenho (AAD) do ano letivo de 2024/2025 e tem como objetivo principal o desenvolvimento e simulação de um circuito lógico sequencial utilizando a linguagem VHDL. O circuito implementa um acumulador indexado, cujo comportamento é baseado numa função em C fornecida. A tarefa inclui a descrição de diferentes arquiteturas do circuito, bem como a criação de bancadas de teste (testbenches) para validar o seu funcionamento.

O acumulador deverá ser concebido para receber e processar valores de entrada em cada ciclo de relógio, realizando operações de leitura, incremento e escrita. Adicionalmente, o projeto explora variantes do acumulador, incluindo implementações com diferentes ciclos de relógio e a introdução de componentes adicionais, como um barrel shifter para operações de deslocamento.

As principais etapas deste trabalho incluem:

1. **Implementação de ciclo único**, onde as operações de leitura, soma e escrita ocorrem no mesmo ciclo de relógio.
2. **Implementação pipeline**, distribuindo as operações por dois ciclos de relógio, assegurando o tratamento de situações específicas, como o uso de endereços consecutivos.
3. **Extensões opcionais**, tais como o uso de deslocamento no incremento e a integração de componentes eficientes, como o barrel shifter.

Este trabalho não se limita à implementação funcional do circuito, mas também inclui a optimização do seu desempenho e a determinação do menor período de relógio funcional. O desenvolvimento requer o domínio das entidades VHDL fornecidas e a adaptação de módulos pré-definidos, como o *adder_n* e o *dual_port_ram*.

Capítulo 1

Tarefa 1

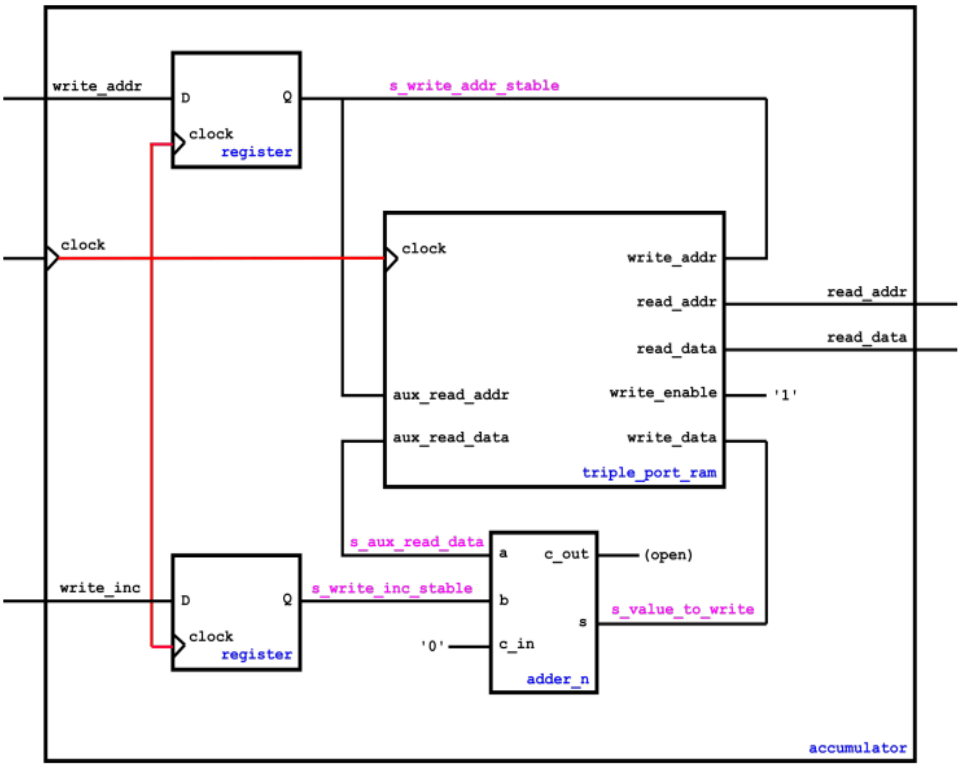


Figura 1: Circuito lógico utilizado

1 Single-cycle implementation

1.1 Implementação do Register

Figura 2:Register Block

```vhdl
library IEEE;
use IEEE.std_logic_1164.all;

entity vector_register is
    generic
    (
        DATA_BITS : integer range 1 to 32 := 4
    );
    port
    (
        clock : in std_logic;
        d     : in std_logic_vector(DATA_BITS-1 downto 0);
        q     : out std_logic_vector(DATA_BITS-1 downto 0) := (others => '0');
        en    : in std_logic
    );
end vector_register;

architecture behavioral of vector_register is
begin
    process(clock) is
    begin
        if rising_edge(clock) and en = '1' then
            q <= transport d after 10 ps;
        end if;
    end process;
end behavioral;
```

Figura 3: Register Block Code

1.1.1 Explicação do Código do Register

A entidade vector_register define a interface externa do componente. Um dos parâmetros genéricos, chamado DATA_BITS, permite configurar o tamanho do vetor de dados de entrada (d) e saída (q), permitindo que o registo armazene palavras de dados com um número variável de bits, entre 1 e 32. Por predefinição, este parâmetro está configurado para 4 bits. Isto torna o componente flexível e reutilizável em diferentes contextos onde seja necessário armazenar vetores de diferentes tamanhos.

O componente possui quatro portas de entrada e saída. O sinal clock é o sinal de relógio que sincroniza o funcionamento do registo. A entrada d é o vetor de dados que se pretende armazenar no registo, enquanto a saída q é o vetor onde o valor armazenado é disponibilizado após a atualização. Inicialmente, a saída q é configurada para ter todos os seus bits a zero, como indicado pela atribuição (others => '0'). Por fim, o sinal en é um sinal de habilitação, que determina se o valor de entrada deve ou não ser armazenado no registo durante o próximo ciclo de relógio.

Na arquitetura, chamada behavioral, é descrito o funcionamento interno do registo. Este comportamento é implementado através de um processo sensível ao sinal de relógio (clock). Dentro deste processo, é verificada a ocorrência de uma borda ascendente do relógio, ou seja, o momento em que o sinal de relógio muda do nível lógico 0 para o nível lógico 1. Este tipo de deteção é feito utilizando a função rising_edge(clock).

Quando uma borda ascendente do relógio é detetada, o registo verifica o estado do sinal de habilitação (en). Se este sinal estiver ativo, ou seja, com valor lógico '1', o registo atualiza a saída q com o valor presente na entrada d. Esta atribuição de valor é feita utilizando um atraso de transporte (transport), que simula um atraso de 10 picosegundos na propagação do sinal. Este atraso é utilizado principalmente em simulações para modelar comportamentos físicos realistas, mas não afeta o comportamento lógico do registo no hardware.

Se o sinal de habilitação não estiver ativo (en = '0'), o registo mantém o valor armazenado anteriormente, sem alterações na saída q. Desta forma, o componente garante que os valores só são atualizados quando desejado, proporcionando controlo adicional sobre o armazenamento de dados.

1.1.2 Explicação dos resultados esperados na testbench

```
stim_proc: process
begin
  -- Test case 1: Enable is '0', no data should be loaded
  tb_d <= "0001";
  tb_en <= '0';
  wait for 40 ns;

  -- Test case 2: Enable is '1', data should be loaded
  tb_en <= '1';
  tb_d <= "0010";
  wait for 40 ns;

  -- Test case 3: Change data while enabled
  tb_d <= "0100";
  wait for 40 ns;

  -- Test case 4: Disable enable, data should not change
  tb_en <= '0';
  tb_d <= "1000";
  wait for 40 ns;

  -- End simulation
  wait;
end process;
```

Figura 4: Código da Testbench

Clock	Enable(en)	Input(d,decimal)	Output(q,decimal)
0	0	1	0
1	0	1	0
2	1	2	2
3	1	4	4
4	0	8	4

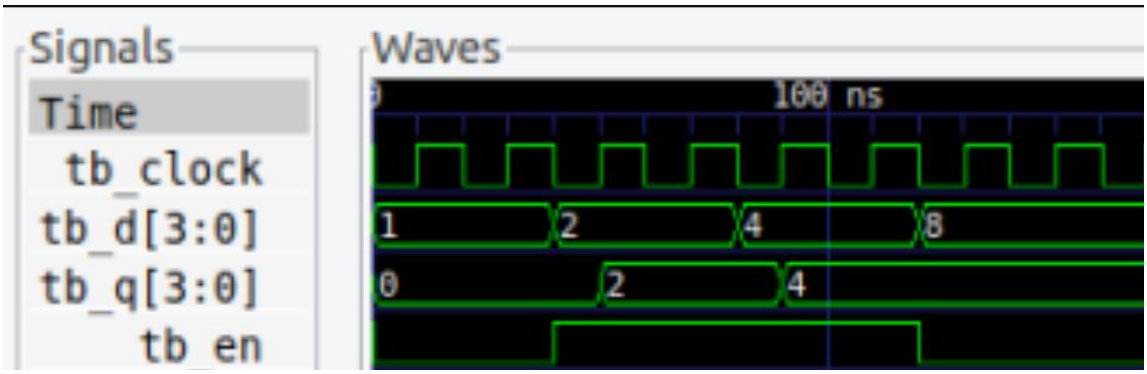


Figura 5: Resultados da testbench

1.2 Implementação do Adder_n

1.2.1 Explicação do xor_gate_3 e testbench

1. O componente recebe três entradas lógicas (input0, input1, input2).
2. Realiza a operação XOR sobre estas três entradas, seguindo as regras da lógica XOR.
3. Atribui o resultado à saída output0, com um atraso de transporte de 20 ps (útil para simulações).

```vhdl
library IEEE;
use IEEE.std_logic_1164.all;

entity xor_gate_3 is
  port(input0  : in  std_logic;
       input1  : in  std_logic;
       input2  : in  std_logic;
       output0 : out std_logic);
end xor_gate_3;

architecture behavioral of xor_gate_3 is
begin
  output0 <= transport input0 xor input1 xor input2 after 20 ps;
end behavioral;
```

Figura 6: xor_gate_3 Código

Input0	Input1	Input2	Output0
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

```vhdl
stim_proc: process
begin
   -- Test case 1: All inputs are '0'
   tb_input0 <= '0';
   tb_input1 <= '0';
   tb_input2 <= '0';
   wait for 50 ps;

   -- Test case 2: One input is '1'
   tb_input0 <= '1';
   tb_input1 <= '0';
   tb_input2 <= '0';
   wait for 50 ps;

   -- Test case 3: Two inputs are '1'
   tb_input0 <= '1';
   tb_input1 <= '1';
   tb_input2 <= '0';
   wait for 50 ps;

   -- Test case 4: All inputs are '1'
   tb_input0 <= '1';
   tb_input1 <= '1';
   tb_input2 <= '1';
   wait for 50 ps;

   -- Test case 5: Alternating inputs
   tb_input0 <= '0';
   tb_input1 <= '1';
   tb_input2 <= '1';
   wait for 50 ps;

   -- Test case 6: Another alternating pattern
   tb_input0 <= '1';
   tb_input1 <= '0';
   tb_input2 <= '1';
   wait for 50 ps;
```

Figura 7: Código da Testbench do xor_gate_3

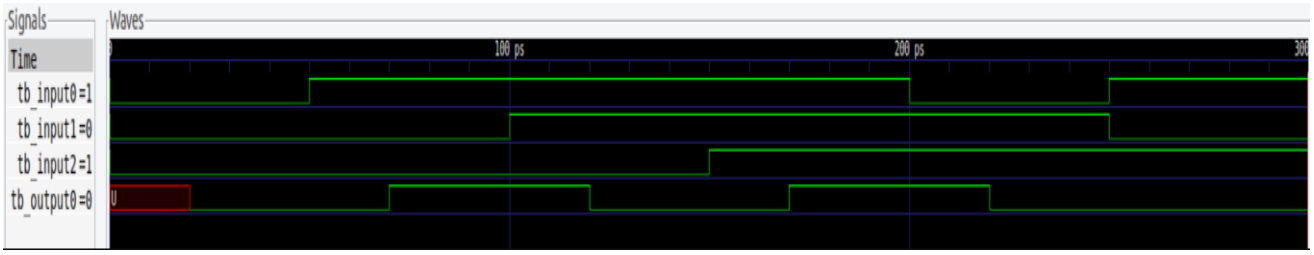


Figura 8: Resultado da Simulação da Testbench do xor_gate_3

1.2.2 Explicação do and_gate_2 e testbench

```
library IEEE;
use IEEE.std_logic_1164.all;

entity and_gate_2 is
  port(input0  : in  std_logic;
       input1  : in  std_logic;
       output0 : out std_logic);
end and_gate_2;

architecture behavioral of and_gate_2 is
begin
  output0 <= transport input0 and input1 after 10 ps;
end behavioral;
```

Figura 9: and_gate_2 Código

1. Este componente recebe dois sinais de entrada (input0 e input1).
2. Realiza a operação lógica AND entre as duas entradas.
3. Atribui o resultado da operação à saída output0, com um atraso simulado de 10 ps.

```
stim_proc: process
begin
  -- Test case 1: Both inputs are '0'
  tb_input0 <= '0';
  tb_input1 <= '0';
  wait for 50 ps;

  -- Test case 2: First input is '1', second is '0'
  tb_input0 <= '1';
  tb_input1 <= '0';
  wait for 50 ps;

  -- Test case 3: First input is '0', second is '1'
  tb_input0 <= '0';
  tb_input1 <= '1';
  wait for 50 ps;

  -- Test case 4: Both inputs are '1'
  tb_input0 <= '1';
  tb_input1 <= '1';
  wait for 50 ps;

  -- End of simulation
  wait;
end process;
```

Figura 10: Código da Testbench do and_gate_2

Input0	Input1	Output0
0	0	0
0	1	0
1	0	0
1	1	1

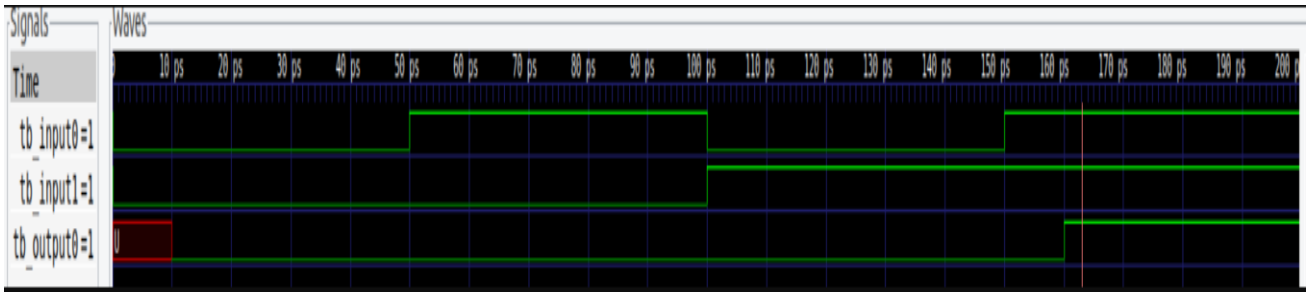


Figura 11: Resultado da Simulação da Testbench do and_gate_2

1.1.3 Explicação do or_gate_3 e testbench

```
library IEEE;
use IEEE.std_logic_1164.all;

entity or_gate_3 is
  port(input0  : in  std_logic;
       input1  : in  std_logic;
       input2  : in  std_logic;
       output0 : out std_logic);
end or_gate_3;

architecture behavioral of or_gate_3 is
begin
  output0 <= transport input0 or input1 or input2 after 15 ps;
end behavioral;
```

Figura 12: or_gate_3 Código

1.O componente recebe três sinais de entrada (input0, input1, input2).
2.Realiza a operação lógica OR entre as três entradas.
3.Atribui o resultado à saída output0 com um atraso de propagação de 15 ps.

```
stim_proc: process
begin
   -- Test case 1: All inputs are '0'
   tb_input0 <= '0';
   tb_input1 <= '0';
   tb_input2 <= '0';
   wait for 50 ps;

   -- Test case 2: One input is '1'
   tb_input0 <= '1';
   tb_input1 <= '0';
   tb_input2 <= '0';
   wait for 50 ps;

   -- Test case 3: Two inputs are '1'
   tb_input0 <= '0';
   tb_input1 <= '1';
   tb_input2 <= '1';
   wait for 50 ps;

   -- Test case 4: All inputs are '1'
   tb_input0 <= '1';
   tb_input1 <= '1';
   tb_input2 <= '1';
   wait for 50 ps;

   -- Test case 5: Alternating inputs
   tb_input0 <= '1';
   tb_input1 <= '0';
   tb_input2 <= '1';
   wait for 50 ps;

   -- Test case 6: Another alternating pattern
   tb_input0 <= '0';
   tb_input1 <= '1';
   tb_input2 <= '0';
   wait for 50 ps;

   -- End simulation
   wait;
end process;
```

Figura 13: Código da Testbench do or_gate_3

Input0	Input1	Input2	Output0
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

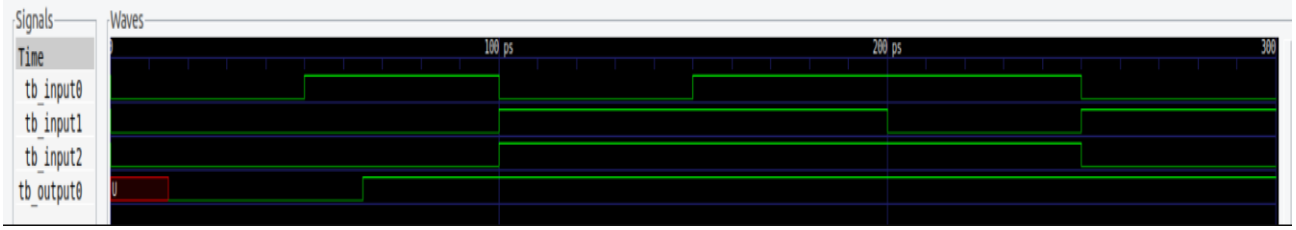


Figura 14: Resultado da Simulação da Testbench do or_gate_3

1.1.4 Explicação do full_adder e testbench

```vhdl
library IEEE;
use IEEE.std_logic_1164.all;

entity full_adder is
   port(a     : in  std_logic;
        b     : in  std_logic;
        c_in  : in  std_logic;
        s     : out std_logic;
        c_out : out std_logic);
end full_adder;

architecture behavioral of full_adder is
begin
  s <= a xor b xor c_in;
  c_out <= (a and b) or (a and c_in) or (b and c_in);
end behavioral;

architecture structural of full_adder is
  signal s_and_ab : std_logic;
  signal s_and_ac : std_logic;
  signal s_and_bc : std_logic;
begin
  -- s <= a xor b xor c_in;
  my_xor :    entity work.xor_gate_3(behavioral)
              port map(input0  => a,
                       input1  => b,
                       input2  => c_in,
                       output0 => s);
  -- c_out <= (a and b) or (a and c_in) or (b and c_in);
  my_and_ab : entity work.and_gate_2(behavioral)
              port map(input0  => a,
                       input1  => b,
                       output0 => s_and_ab);
  my_and_ac : entity work.and_gate_2(behavioral)
              port map(input0  => a,
                       input1  => c_in,
                       output0 => s_and_ac);
  my_and_bc : entity work.and_gate_2(behavioral)
              port map(input0  => b,
                       input1  => c_in,
                       output0 => s_and_bc);
  my_or     : entity work.or_gate_3(behavioral)
              port map(input0  => s_and_ab,
                       input1  => s_and_ac,
                       input2  => s_and_bc,
                       output0 => c_out);
```

Figura 15: full_adder Código

1. O componente calcula o bit de soma (s) utilizando uma porta XOR de três entradas (xor_gate_3).
2. O componente calcula os três termos AND necessários (a AND b, a AND c_in, e b AND c_in) usando três portas and_gate_2.
3. Os resultados das três operações AND são combinados numa porta OR de três entradas (or_gate_3) para gerar o bit de transporte (c_out).

```
-- Stimulus process
stim_proc: process
begin
   -- Test case 1: 0 + 0 + 0
   tb_a <= '0'; tb_b <= '0'; tb_c_in <= '0';
   wait for 50 ps;

   -- Test case 2: 0 + 0 + 1
   tb_a <= '0'; tb_b <= '0'; tb_c_in <= '1';
   wait for 50 ps;

   -- Test case 3: 0 + 1 + 0
   tb_a <= '0'; tb_b <= '1'; tb_c_in <= '0';
   wait for 50 ps;

   -- Test case 4: 0 + 1 + 1
   tb_a <= '0'; tb_b <= '1'; tb_c_in <= '1';
   wait for 50 ps;

   -- Test case 5: 1 + 0 + 0
   tb_a <= '1'; tb_b <= '0'; tb_c_in <= '0';
   wait for 50 ps;

   -- Test case 6: 1 + 0 + 1
   tb_a <= '1'; tb_b <= '0'; tb_c_in <= '1';
   wait for 50 ps;

   -- Test case 7: 1 + 1 + 0
   tb_a <= '1'; tb_b <= '1'; tb_c_in <= '0';
   wait for 50 ps;

   -- Test case 8: 1 + 1 + 1
   tb_a <= '1'; tb_b <= '1'; tb_c_in <= '1';
   wait for 50 ps;

   -- End simulation
   wait;
end process;
```

Figura 16: Código da Testbench do full_adder

A	B	C_in	S(Soma)	C_out(carry)
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

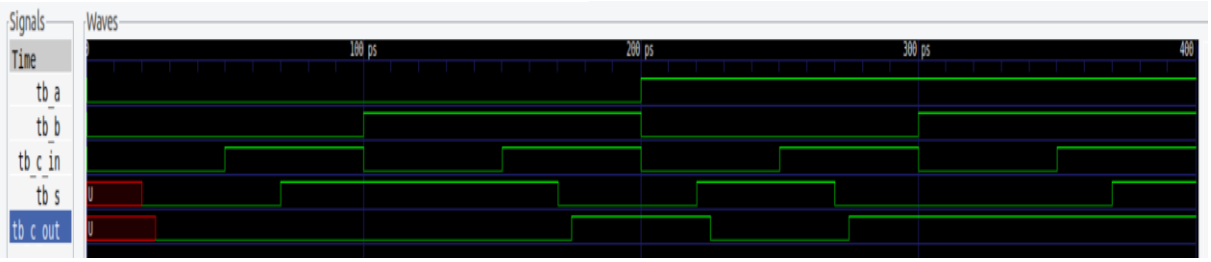
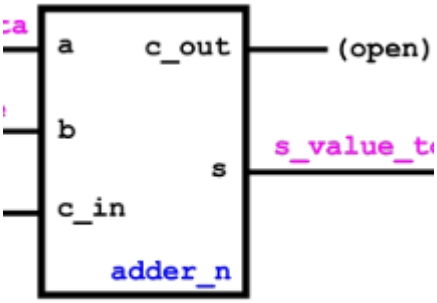


Figura 17: Resultado da Simulação da Testbench do full_adder

1.1.5 Explicação do adder_n e testbench




```
library IEEE;
use IEEE.std_logic_1164.all;
--use IEEE.numeric_std.all;

entity adder_n is
  generic(N     : positive := 8);
     port(a     : in  std_logic_vector(N-1 downto 0);
          b     : in  std_logic_vector(N-1 downto 0);
          c_in  : in  std_logic;
          s     : out std_logic_vector(N-1 downto 0);
          c_out : out std_logic);
end adder_n;

architecture structural of adder_n is
  signal s_carry_in  : std_logic_vector(N-1 downto 0);
  signal s_carry_out : std_logic_vector(N-1 downto 0);
begin
  s_carry_in(0) <= c_in;
  c_out <= s_carry_out(N-1);
  gen_adder : for i in 0 to (N-1) generate
       adder:   entity work.full_adder(structural)
                     port map(a     => a(i),
                              b     => b(i),
                              c_in  => s_carry_in(i),
                              s     => s(i),
                              c_out => s_carry_out(i));
  end generate;
  gen_carry : for i in 1 to (N-1) generate
    s_carry_in(i) <= s_carry_out(i-1);
  end generate;
end structural;
```

Figura 18: adder_n Código

1. O somador menos significativo (bit 0) calcula a soma de a(0), b(0) e c_in, gerando o bit de soma s(0) e o transporte s_carry_out(0).
2. O transporte gerado (s_carry_out(0)) é propagado para o próximo somador.
3. Este processo continua para os bits seguintes, com cada somador processando um bit correspondente e propagando o transporte.
4. O somador mais significativo gera o transporte final, atribuído à saída c_out.

```
stim_proc: process
begin
  -- Test case 1: Add two zeros
  tb_a <= "00000000";
  tb_b <= "00000000";
  tb_c_in <= '0';
  wait for 100 ps;

  -- Test case 2: Add 1 + 1 with no carry in
  tb_a <= "00000001";
  tb_b <= "00000001";
  tb_c_in <= '0';
  wait for 100 ps;

  -- Test case 3: Add 1 + 1 with carry in
  tb_a <= "00000001";
  tb_b <= "00000001";
  tb_c_in <= '1';
  wait for 100 ps;

  -- Test case 4: Add 255 + 1 (overflow case)
  tb_a <= "11111111";
  tb_b <= "00000001";
  tb_c_in <= '0';
  wait for 100 ps;

  -- Test case 5: Add 128 + 128 (carry out case)
  tb_a <= "10000000";
  tb_b <= "10000000";
  tb_c_in <= '0';
  wait for 100 ps;

  -- Test case 6: Add 85 + 85 (binary pattern)
  tb_a <= "01010101";
  tb_b <= "01010101";
  tb_c_in <= '0';
  wait for 100 ps;

  -- End simulation
  wait;
```

Figura 19: Código da Testbench do adder_n

a(decimal)	b(decimal)	c_in	s(decimal)	c_out
0	0	0	0	0
1	1	0	2	0
1	1	1	3	0
255	1	0	0	1
128	128	0	0	1
85	85	0	170	0

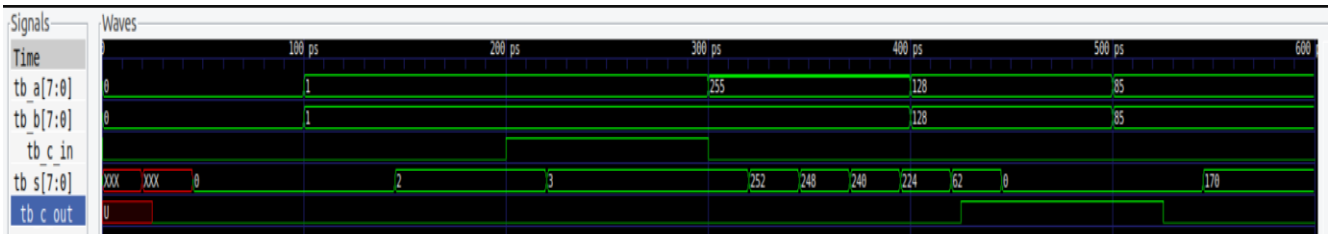


Figura 20: Resultado da Simulação da Testbench do adder_n

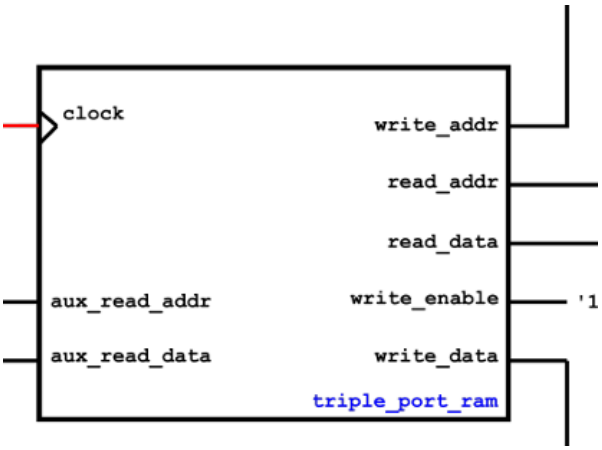
1.3 Implementação do triple_port_ram e testbench

```vhdl
library IEEE;
use     IEEE.std_logic_1164.all;
use     IEEE.numeric_std.all;

entity triple_port_ram is
  generic
  (
    ADDR_BITS : integer range 2 to 16;
    DATA_BITS : integer range 1 to 32
  );
  port
  (
    clock         : in std_logic;
    -- write port
    write_addr : in  std_logic_vector(ADDR_BITS-1 downto 0);
    write_data : in  std_logic_vector(DATA_BITS-1 downto 0);
    -- read port
    read_addr  : in  std_logic_vector(ADDR_BITS-1 downto 0);
    read_data  : out std_logic_vector(DATA_BITS-1 downto 0) := (others => '0');
    -- aux read port
    aux_read_addr : in  std_logic_vector(ADDR_BITS-1 downto 0);
    aux_read_data : out std_logic_vector(DATA_BITS-1 downto 0) := (others => '0')
  );
end triple_port_ram;

architecture behavioral of triple_port_ram is
  type ram_t is array(0 to 2**ADDR_BITS-1) of std_logic_vector(DATA_BITS-1 downto 0);
  signal ram : ram_t := (others => (others => '0'));
begin
  -- synchronous n
  process(clock) is
  begin
    if rising_edge(clock) then
      ram(to_integer(unsigned(write_addr))) <= write_data;
    end if;
  end process;
  process(clock) is
  begin
    if rising_edge(clock) then
      read_data <= transport ram(to_integer(unsigned(read_addr))) after 200 ps;
    end if;
  end process;
  -- aux read port (asynchronous, read old data, warning, no process)
  aux_read_data <= transport ram(to_integer(unsigned(aux_read_addr))) after 200 ps;
end behavioral;
```

Figura 21: triple_port_ram Código



Funcionamento

1. **Escrita síncrona**:
   - Os dados são escritos na memória na borda ascendente do relógio.
   - A escrita ocorre no endereço especificado por write_addr.
2. **Leitura síncrona**:
   - Os dados são lidos da memória na borda ascendente do relógio.
   - O dado no endereço especificado por read_addr é atribuído a read_data.
3. **Leitura auxiliar assíncrona**:
   - Os dados são lidos continuamente (independentemente do relógio) no endereço especificado por aux_read_addr e atribuídos a aux_read_data.

Características

a) **Três portas independentes**:
   i. Uma para escrita e duas para leitura, permitindo operações simultâneas de leitura e escrita em endereços diferentes.

b) **Leitura síncrona e assíncrona:**
   i. A leitura principal (`read_data`) é sincronizada com o relógio, enquanto a leitura auxiliar (`aux_read_data`) é assíncrona.

c) **Atraso de transporte:**
   i. Simula o tempo físico necessário para acessar os dados da memória (200 ps).

Clock cycle	Write_Addr(dec)	Write_Data(dec)	Read_Addr(dec)	Read_Data	Aux_Read_Addr(dec)	Aux_Read_Data(dec)
1	0	170	-	-	-	-
2	-	-	0	170	-	-
3	1	204	-	-	-	-
4	-	-	-	-	1	204
5	2	240	-	-	-	-
6	-	-	-	-	2	240
7	15	15	-	-	-	-
8	-	-	15	15	-	-

```
stim_proc: process
begin
  -- Test case 1: Write data to address 0 and read back
  tb_write_addr <= "0000";
  tb_write_data <= "10101010";
  wait for 20 ns;

  tb_read_addr <= "0000";
  wait for 20 ns;

  -- Test case 2: Write data to address 1 and read from aux port
  tb_write_addr <= "0001";
  tb_write_data <= "11001100";
  wait for 20 ns;

  tb_aux_read_addr <= "0001";
  wait for 20 ns;

  -- Test case 3: Write data to address 2 and read asynchronously
  tb_write_addr <= "0010";
  tb_write_data <= "11110000";
  wait for 20 ns;

  tb_aux_read_addr <= "0010";
  wait for 20 ns;

  -- Test case 4: Write data to address 15 and read back
  tb_write_addr <= "1111";
  tb_write_data <= "00001111";
  wait for 20 ns;

  tb_read_addr <= "1111";
  wait for 20 ns;

  -- End simulation
  wait;
end process;
```

Figura 22: Código da Testbench do triple_port_ram

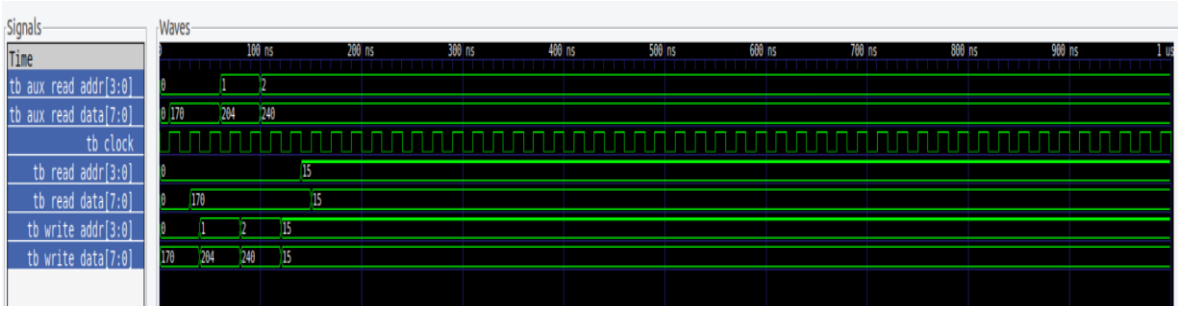


Figura 23: Resultado da Simulação da Testbench do triple_port_ram

1.4 Implementação do accumulator

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity accumulator is
  generic (
    ADDR_BITS : integer range 2 to 8 := 4;
    DATA_BITS : integer range 4 to 32 := 8;
    DATA_BITS_LOG2 : integer range 1 to 4 := 3
  );
  port (
    clock      : in  std_logic;
    write_addr : in  std_logic_vector(ADDR_BITS-1 downto 0);
    write_inc  : in  std_logic_vector(DATA_BITS-1 downto 0);
    read_addr  : in  std_logic_vector(ADDR_BITS-1 downto 0);
    read_data  : out std_logic_vector(DATA_BITS-1 downto 0)
  );
end accumulator;

architecture structural of accumulator is
  signal s_write_addr_stable : std_logic_vector(ADDR_BITS-1 downto 0);
  signal s_write_inc_stable  : std_logic_vector(2**DATA_BITS_LOG2-1 downto 0);
  signal s_value_to_write    : std_logic_vector(2**DATA_BITS_LOG2-1 downto 0);
  signal s_aux_read_data     : std_logic_vector(2**DATA_BITS_LOG2-1 downto 0);
begin
  addr_reg : entity work.vector_register(behavioral)
    generic map (
      DATA_BITS => ADDR_BITS
    )
    port map (
      clock => clock,
      d     => write_addr,
      q     => s_write_addr_stable,
      en    => '1'  -- literal correto para std_logic
    );

  inc_reg : entity work.vector_register(behavioral)
    generic map (
      DATA_BITS => 2**DATA_BITS_LOG2
    )
    port map (
      clock => clock,
      d     => write_inc,
      q     => s_write_inc_stable,
      en    => '1'  -- literal correto para std_logic
    );

  adder : entity work.adder_n(structural)
    generic map (
      N => 2**DATA_BITS_LOG2
    )
    port map (
      a    => s_aux_read_data,
      b    => s_write_inc_stable, -- Alterar para usar o shifter, se necessário
      c_in => '0',
      s    => s_value_to_write,
      c_out => open
    );

  memory : entity work.triple_port_ram(behavioral)
    generic map (
      ADDR_BITS => ADDR_BITS,
      DATA_BITS => 2**DATA_BITS_LOG2
    )
    port map (
      clock          => clock,
      write_addr     => s_write_addr_stable,
      write_data     => s_value_to_write,
      read_addr      => read_addr,
      read_data      => read_data,
      aux_read_addr  => s_write_addr_stable,
      aux_read_data  => s_aux_read_data
    );

end structural;
```

Figura 24: Accumulator Código

Arquitetura `structural`:

Sinais internos:

- `s_write_addr_stable`: Registrador que armazena o endereço de escrita.
- `s_write_inc_stable`: Registrador que armazena o incremento de escrita.
- `s_value_to_write`: Armazena o valor a ser escrito na memória.
- `s_aux_read_data`: Armazena os dados lidos da memória auxiliarmente.

Componentes principais

I. **Registrador de endereço (`addr_reg`)**:
   a. Componente `vector_register` usado para armazenar o endereço de escrita.
   b. Sincronizado com o sinal de relógio.

   addr_reg : entity work.vector_register(behavioral)

II. **Registrador de incremento (`inc_reg`)**:
   a. Outro componente `vector_register`, usado para armazenar o valor de incremento.
   b. Também sincronizado com o relógio.

   inc_reg : entity work.vector_register(behavioral)

III. **Somador (`adder`)**:
   a. Componente `adder_n`, que realiza a soma entre o valor lido da memória (`s_aux_read_data`) e o incremento armazenado (`s_write_inc_stable`).
   b. O resultado da soma é armazenado em `s_value_to_write`.

   adder: entity work.adder_n(structural)

IV. **Memória (memory):**
   a. Componente `triple_port_ram`, usado como uma memória RAM com três portas:
      i. Porta de escrita para armazenar dados em endereços.
      ii. Porta de leitura principal para acessar dados.
      iii. Porta de leitura auxiliar para acessar dados adicionais.

   memory: entity work.triple_port_ram(behavioral)

Fluxo de dados

- O endereço de escrita (`write_addr`) é armazenado no registrador `s_write_addr_stable`.
- O incremento de escrita (`write_inc`) é armazenado no registrador `s_write_inc_stable`.
- A memória é lida no endereço especificado pelo endereço auxiliar (`aux_read_addr`), e os dados lidos são armazenados em `s_aux_read_data`.
- O somador combina o valor lido da memória com o incremento armazenado, gerando o valor a ser escrito (`s_value_to_write`).
- A memória é atualizada no endereço especificado (`write_addr`) com o valor calculado (`s_value_to_write`).
- O dado no endereço especificado pela porta de leitura principal (`read_addr`) é disponibilizado na saída `read_data`.

A testbench utilizada para testes foi a fornecida pelo professor:

Clock	Endereço(s_write_addr)	Incremento(s_write_inc)	Valor acumulado(dec)
T=0	3	7	7
T=1	15	3	3
T=2	3	1	8(7+1)
T=3	3	7	15(8+7)
T=4	5	64	64
T=5	4	1	1
T=6	4	16	17(1+16)
T=7	3	1	16(15+1)
T=8	10	100	100
T=9	2	17	17

Endereço	Valores Acumulados
a[2]	17
a[3]	7->8->15->16
a[4]	1->17
a[5]	64
a[10]	100
a[15]	3

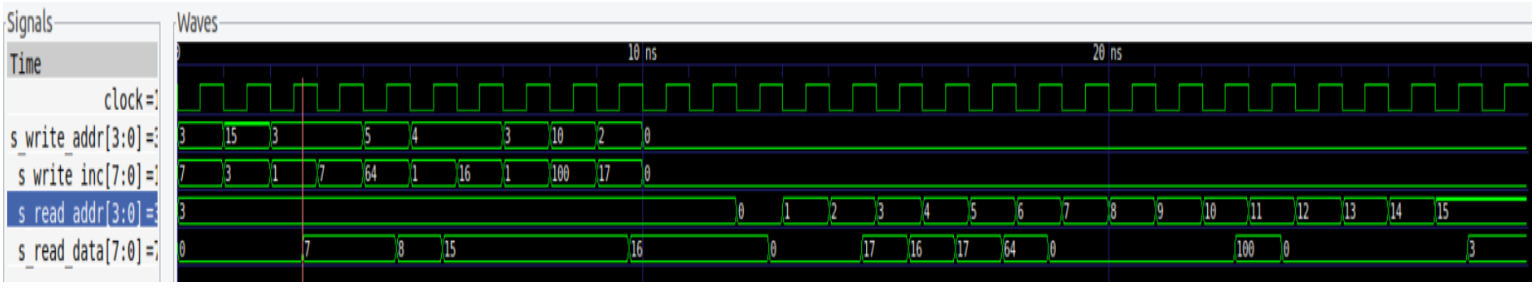


Figura 25: Resultado da Simulação da Testbench do accumulator

Capítulo 2

Tarefa 2

2.1 Pipelined implementation

```
architecture structural of accumulator is

  signal s_write_addr_stable : std_logic_vector(ADDR_BITS-1 downto 0);
  signal s_write_inc_stable  : std_logic_vector(DATA_BITS-1 downto 0);
  signal s_mux_read_data     : std_logic_vector(DATA_BITS-1 downto 0);
  signal s_staged_read_data  : std_logic_vector(DATA_BITS-1 downto 0) := (others => '0');
  signal s_staged_write_addr : std_logic_vector(ADDR_BITS-1 downto 0) := (others => '0');
  signal s_staged_write_inc  : std_logic_vector(DATA_BITS-1 downto 0) := (others => '0');
  signal s_staged_result     : std_logic_vector(DATA_BITS-1 downto 0) := (others => '0');
  signal s_staged_write_addr : std_logic_vector(ADDR_BITS-1 downto 0) := (others => '0');
  signal adder_a             : std_logic_vector(DATA_BITS-1 downto 0);
  signal adder_b             : std_logic_vector(DATA_BITS-1 downto 0);
  signal s_value_to_write    : std_logic_vector(DATA_BITS-1 downto 0);

begin

  addr_reg : entity work.vector_register(behavioral)
    generic map (
      DATA_BITS => ADDR_BITS
    )
    port map (
      clock => clock,
      d     => write_addr,
      q     => s_write_addr_stable,
      en    => '1'
    );

  inc_reg : entity work.vector_register(behavioral)
    generic map (
      DATA_BITS => DATA_BITS
    )
    port map (
      clock => clock,
      d     => write_inc,
      q     => s_write_inc_stable,
      en    => '1'
    );

  process(clock)
    variable v_hazard : std_logic;
  begin
    if rising_edge(clock) then
      s_staged_read_data  <= s_mux_read_data;
      s_staged_write_addr <= s_write_addr_stable;
      s_staged_write_inc  <= s_write_inc_stable;

      if (s_staged_write_addr = s_write_addr_stable) then
        v_hazard := '1';
      else
        v_hazard := '0';
      end if;

      if v_hazard = '1' then
        adder_a <= s_staged_result;
      else
        adder_a <= s_staged_read_data;
      end if;
      adder_b <= s_staged_write_inc;

      s_staged_write_addr <= s_staged_write_addr;
    end if;
  end process;

  adder : entity work.adder_n(structural)
    generic map (
      N => 2**DATA_BITS_LOG2
    )
    port map (
      a     => adder_a,
      b     => adder_b,
      c_in  => '0',
      s     => s_value_to_write,
      c_out => open
    );

  process (clock)
  begin
    if rising_edge(clock) then
      s_staged_result <= s_value_to_write;
    end if;
  end process;

  memory : entity work.triple_port_ram(behavioral)
    generic map (
      ADDR_BITS => ADDR_BITS,
      DATA_BITS => DATA_BITS
    )
    port map (
      clock      => clock,
      write_addr => s_staged_write_addr,
      write_data => s_staged_result,
      read_addr  => read_addr,
      read_data  => read_data,
      aux_read_addr => s_write_addr_stable,
      aux_read_data => s_mux_read_data
    );

end structural;
```

Figura 26: Accumulator Código pipelined

- Estabilização dos Dados

  write_addr e write_inc são registrados para garantir que seus valores estejam estáveis durante o ciclo de clock.
- Estágio 1 do Pipeline

  Os dados lidos da memória (s_aux_read_data) são registados como s_stage1_read_data.

  O endereço de escrita e o incremento também são registados (s_stage1_write_addr e s_stage1_write_inc).
- Detecção de Hazard

  Verifica se o endereço de escrita atual coincide com o endereço usado anteriormente (s_stage2_write_addr).

  Em caso positivo, usa o resultado da operação anterior (s_stage2_result) como entrada para o somador; caso contrário, usa os dados lidos.
- Cálculo da Soma

  O somador calcula o valor acumulado com base em suas entradas (adder_a e adder_b).
- Escrita na Memória

O resultado da soma (s_value_to_write) é armazenado em s_stage2_result e escrito na memória no endereço correspondente (s_stage2_write_addr).
- Leitura da Memória

Um endereço de leitura externo (read_addr) é usado para acessar e retornar o valor da memória através de read_data.

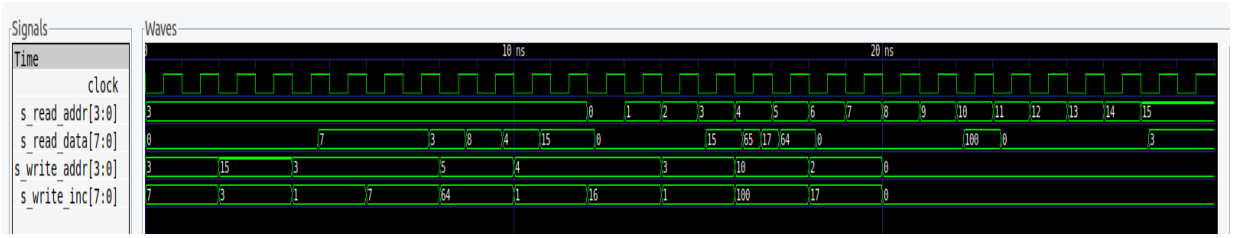


Figura 27: Resultado da Simulação da Testbench do accumulator pipelined

Capítulo 3

Tarefa 3

3.1 Modified accumulator

3.1.1 Implementação do Barrel_shifter

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

entity barrel_shifter is
    generic (
        DATA_BITS_LOG2 : integer := 4  -- log2(DATA_WIDTH)
    );
    port (
        data_in  : in  std_logic_vector((2**DATA_BITS_LOG2)-1 downto 0);
        shift    : in  std_logic_vector(DATA_BITS_LOG2-1 downto 0);
        data_out : out std_logic_vector((2**DATA_BITS_LOG2)-1 downto 0)
    );
end barrel_shifter;

architecture behavioral of barrel_shifter is
begin
    process(data_in, shift)
        variable temp_data_var : unsigned((2**DATA_BITS_LOG2)-1 downto 0);
    begin
        temp_data_var := unsigned(data_in);
        temp_data_var := temp_data_var sll to_integer(unsigned(shift));
        data_out <= std_logic_vector(temp_data_var);
    end process;
end behavioral;
```

Figura 28: Barrel_shifter Código

Funcionamento

1. O circuito recebe um vetor de entrada (`data_in`) e um vetor de controle (`shift`) que especifica o número de posições para deslocar.
2. Realiza o deslocamento lógico à esquerda dos bits do vetor de entrada.
3. O resultado do deslocamento é disponibilizado na saída `data_out`.

```
-- Processo de estímulo
stim_proc: process
begin
    -- Test Case 1: Shift por 0 bits
    data_in <= "0000000000010010";  -- Valor: 18 (16 bits
    shift   <= "0000";              -- Shift por 0 bits
    wait for 10 ns;

    -- Test Case 2: Shift por 1 bit (2^0 = 1)
    shift   <= "0001";              -- Shift por 1 bit
    wait for 10 ns;

    -- Test Case 3: Shift por 3 bits (2^0 + 2^1 = 3)
    shift   <= "0011";              -- Shift por 3 bits
    wait for 10 ns;

    -- Test Case 4: Shift por 4 bits (2^2 = 4)
    shift   <= "0100";              -- Shift por 4 bits
    wait for 10 ns;

    -- Test Case 5: Shift por 5 bits (2^2 + 2^0 = 5)
    shift   <= "0101";              -- Shift por 5 bits
    wait for 10 ns;

    -- Test Case 6: Shift por 7 bits (2^2 + 2^1 + 2^0 = 7
    shift   <= "0111";              -- Shift por 7 bits
    wait for 10 ns;

    -- Test Case 7: Shift por 15 bits (2^3 + 2^2 + 2^1 +
    shift   <= "1111";              -- Shift por 15 bits
    wait for 10 ns;

    -- Test Case 8: Shift por 8 bits (2^3 = 8)
    shift   <= "1000";              -- Shift por 8 bits
    wait for 10 ns;

    -- Test Case 9: Shift por 10 bits (2^3 + 2^1 = 10)
    shift   <= "1010";              -- Shift por 10 bits
    wait for 10 ns;

    -- Test Case 10: Shift por valor máximo
    data_in <= "0000000011111111";  -- Valor: 255
    shift   <= "1111";              -- Shift por 15 bits
    wait for 10 ns;

    -- Finaliza a simulação
    wait;
end process;
```

Test case	Data_in(bin)	Shift(dec)	Data_out(bin)
1	0000000000010010	0	0000000000010010
2	0000000000010010	1	0000000000100100
3	0000000000010010	3	0000000010100000
4	0000000000010010	4	0000001001000000
5	0000000000010010	5	0000010010000000
6	0000000000010010	7	0100100000000000
7	0000000000010010	15	1000000000000000
8	0000000000010010	8	1001000000000000
9	0000000000010010	10	0100100000000000
10	0000000011111111	15	1000000000000000

Figura 29: Código da Testbench do Barrel_shifter

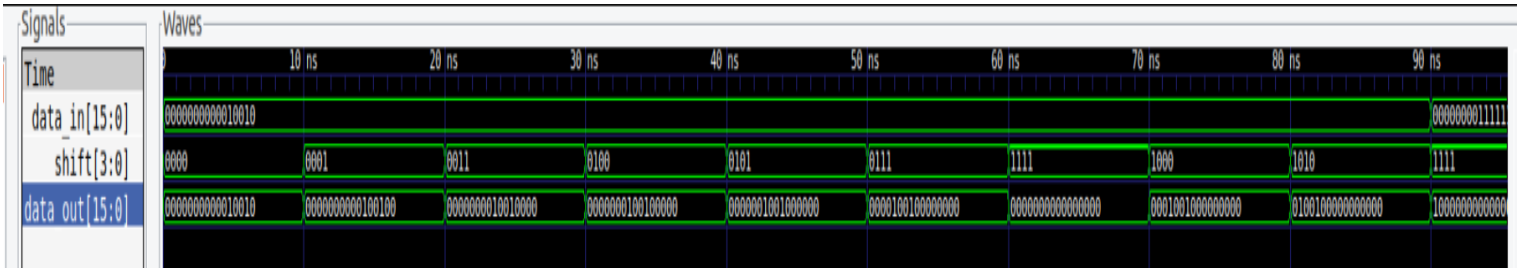


Figura 30: Resultado da Simulação da Testbench do barrel_shifter

3.1.2 Implementação do Accumulator with Barrel_shifter

```
architecture structural of accumulator is
  signal s_write_addr_stable : std_logic_vector(ADDR_BITS-1 downto 0) := (others => '0');
  signal s_write_inc_stable  : std_logic_vector(2**DATA_BITS_LOG2-1 downto 0) := (others => '0');
  signal s_write_shift_stable: std_logic_vector(DATA_BITS_LOG2-1 downto 0) := (others => '0');
  signal s_shifted_write_inc : std_logic_vector(2**DATA_BITS_LOG2-1 downto 0) := (others => '0');
  signal s_value_to_write    : std_logic_vector(2**DATA_BITS_LOG2-1 downto 0) := (others => '0');
  signal s_aux_read_data     : std_logic_vector(2**DATA_BITS_LOG2-1 downto 0) := (others => '0');
begin
  -- Instanciação do componente vector_register para write_addr
  addr_reg : entity work.vector_register(behavioral)
    generic map (
      DATA_BITS => ADDR_BITS
    )
    port map (
      clock => clock,
      d     => write_addr,
      q     => s_write_addr_stable,
      en    => '1'
    );

  -- Instanciação do componente vector_register para write_inc
  inc_reg : entity work.vector_register(behavioral)
    generic map (
      DATA_BITS => 2**DATA_BITS_LOG2
    )
    port map (
      clock => clock,
      d     => write_inc,
      q     => s_write_inc_stable,
      en    => '1'
    );

  -- Instanciação do componente vector_register para write_shift
  shift_reg : entity work.vector_register(behavioral)
    generic map (
      DATA_BITS => DATA_BITS_LOG2
    )
    port map (
      clock => clock,
      d     => write_shift,
      q     => s_write_shift_stable,
      en    => '1'
    );

  -- Instanciação do componente barrel_shifter
  shifter : entity work.barrel_shifter(behavioral)
    generic map (
      DATA_BITS_LOG2 => DATA_BITS_LOG2
    )
    port map (
      data_in  => s_write_inc_stable,
      shift    => s_write_shift_stable,
      data_out => s_shifted_write_inc
    );

  -- Instanciação do componente adder_n
  adder : entity work.adder_n(structural)
    generic map (
      N => 2**DATA_BITS_LOG2
    )
    port map (
      a     => s_aux_read_data,
      b     => s_shifted_write_inc,
      c_in  => '0',
      s     => s_value_to_write,
      c_out => open
    );

  -- Instanciação do componente triple_port_ram
  memory : entity work.triple_port_ram(behavioral)
    generic map (
      ADDR_BITS => ADDR_BITS,
      DATA_BITS => 2**DATA_BITS_LOG2
    )
    port map (
      clock          => clock,
      write_addr     => s_write_addr_stable,
      write_data     => s_value_to_write,
      read_addr      => read_addr,
      read_data      => read_data,
      aux_read_addr  => s_write_addr_stable,
      aux_read_data  => s_aux_read_data
    );
end structural;
```

Figura 31: Accumulator with Barrel_shifter Código

Mudanças principais

1. **Adição de um registrador para write_shift**:
    a. Um registrador adicional foi introduzido para armazenar o valor do deslocamento (`write_shift`). Este valor será usado pelo barrel shifter para realizar deslocamentos lógicos nos dados de incremento antes da soma.
    b. Este registrador é instanciado como outro componente vector_register.
2. **Introdução do `barrel_shifter`**:
    a. Um **barrel shifter** foi adicionado para deslocar os dados de incremento (write_inc) com base no valor armazenado em write_shift. O resultado do deslocamento é armazenado no sinal s_shifted_write_inc.
3. **Modificação no somador (`adder`)**:
    a. O somador agora recebe os dados deslocados (`s_shifted_write_inc`) como entrada em vez do incremento bruto (`write_inc`).
    b. Isto permite realizar acumulações baseadas em valores deslocados dinamicamente.
4. **Sinais internos adicionais**:
    a. `s_write_shift_stable`: Registrador que armazena o valor do deslocamento estável.
    b. `s_shifted_write_inc`: Sinal que armazena o resultado do deslocamento realizado pelo barrel shifter.

A testbench utilizada para testes foi a fornecida pelo professor:

cycle	Write_addr	Write_inc	Write_shift	Shift_inc	Aux_read_data	Value_to_write
0	3	7	0	7	0	7
1	15	3	4	48	0	48
2	3	1	1	2	7	9
3	3	7	0	7	9	16
4	5	64	1	128	0	128
5	4	1	6	64	0	64
6	4	16	6	1024(over)	64	64
7	3	1	4	16	16	32
8	10	100	2	400(over)	0	144
9	2	17	1	34	0	34

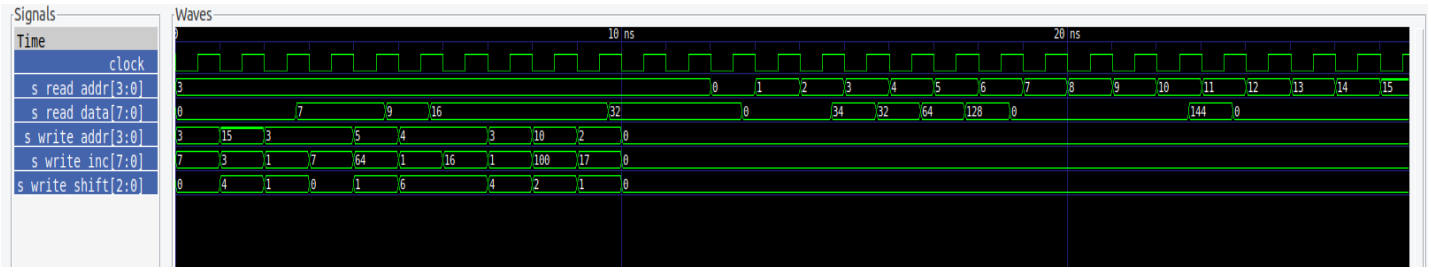


Figura 32: Resultado da Simulação da Testbench do accumulator with barrel_shifter