Universidade Federal do Rio de Janeiro
Departamento de Engenharia Eletrônica
Escola Politécnica

Relatório da Atividade Prática – Somador Vetorial

João Paulo Garcia de Assis Rangel – 124130234
Danilo Santiago Baptista – 124057567
Gabriel dos Santos Percu Saldanha – 124007041
Henrique Gebara Machado Brenes – 124007368

Rio de Janeiro, julho de 2025

Sumário

1 Introdução

O presente documento tem como objetivo abordar o desenvolvimento de um somador vetorial com a funcionalidade de executar instruções vetoriais para soma e subtração. Nesse contexto, o somador recebe entradas de 32 bits e consegue efetuar a operação em múltiplas seções independentes, com tamanhos variando entre 4 e 32, seguindo potências de dois. Com isso, torna-se possível realizar cálculos com vetores que possuem até 8 elementos distintos de 4 bits cada, bastando informar o tamanho do vetor.

Por fim, para os interessados, os circuitos utilizados estão disponíveis no seguinte repositório do GitHub: https://github.com/carol-santiago/Somador-Vetorial-EEL580.**********

2 Explicação do Código

O componente descrito corresponde a um somador e subtrator vetorial operando em paralelo com largura de n bits, onde n pertence ao conjunto 4, 8, 16, 32. Sua implementação foi realizada em VHDL com o propósito de executar, de maneira eficiente, operações de adição ou subtração entre dois vetores de comprimento n.

2.1 Entradas e Saídas

```
1  LIBRARY ieee;
2  USE ieee.std_logic_1164.all;
3
4  ENTITY SomadorVetorial IS
5    PORT (
6      -- Inputs
7      A_i          : IN  std_logic_vector(31 DOWNTO 0);
8      B_i          : IN  std_logic_vector(31 DOWNTO 0);
9      vecSize_i    : IN  std_logic_vector(1 DOWNTO 0);
10     mode_i       : IN  std_logic;
11
12     -- Outputs
13     S_o          : OUT std_logic_vector(31 DOWNTO 0)
14   );
15 END SomadorVetorial;
```

Nesta parte, é descrito o módulo VHDL denominado **SomadorVetorial**, o qual possui quatro sinais de entrada e uma saída. As entradas são as seguintes:

- **A_i**: *Sinal de entrada do vetor A, com largura de 32 bits.*
- **B_i**: *Sinal de entrada do vetor B, também com 32 bits.*
- **vecSize_i**: *Entrada responsável por indicar o tamanho do vetor (utilizando 2 bits).*
- **mode_i**: *Entrada que especifica o tipo de operação. Valor 0 representa soma e valor 1 indica subtração.*

A saída é **S_o**, *correspondente ao resultado da operação aritmética (adição ou subtração).*

2.2 Declaração de Sinais

```
ARCHITECTURE TypeArchitecture OF SomadorVetorial IS
  signal s_C     : std_logic_vector(31 downto 0);
  signal s_P     : std_logic_vector(31 downto 0);
  signal Carry   : std_logic_vector(31 downto 0);
  signal s_Soma  : std_logic_vector(31 downto 0);
  signal B_op    : std_logic_vector(31 downto 0);
```

Nesta etapa, especificamos os sinais indispensáveis para a implementação do circuito. São eles:

- **s_C**: *Sinal responsável por contribuir na geração do carry.*
- **s_P**: *Sinal que colabora com a propagação do carry.*
- **Carry**: Sinal que indica o resultado da propagação do carry.
- **s_soma**: *Sinal que indica o valor obtido da soma ou subtração entre os vetores A e B.*
- **B_dP**: *Sinal correspondente ao complemento de dois do vetor B, utilizado no processo de subtração.*

A operação de complemento de dois é aplicada ao sinal **B_op**, sendo que o vetor **B_i** é invertido (utilizando o operador `not`) quando **mode_i** for igual a `'1'` (indicando subtração), e permanece inalterado quando **mode_i** for igual a `'0'` (indicando soma).

2.3 Complemento de 2 na Subtração e Geração de s_C e s_P

Nesta etapa, realizamos a operação de complemento de dois durante a subtração e geramos os sinais auxiliares **s_C** (carry gerado) e **s_P** (carry propagado), essenciais para o funcionamento das operações de adição ou subtração.

2.3.1 Complemento de 2

O complemento de dois é uma técnica utilizada para representar números negativos em sistemas binários. Durante a subtração, é necessário inverter bit a bit o vetor **B_i** antes de somá-lo ao vetor **A_i**.

Utiliza-se o sinal **B_op** para armazenar o complemento de dois de **B_i**. A expressão aplicada para calcular esse complemento é:

$$B_{\text{op}}[n] = \begin{cases} B_i[n] & \text{se } \texttt{mode\_i} =' 0' \quad \text{(modo de adição)} \\ \sim B_i[n] & \text{se } \texttt{mode\_i} =' 1' \quad \text{(modo de subtração)} \end{cases}$$

Onde n é a posição do bit(0 a 31). Ela foi implementada da seguinte maneira no código:

```
B_op <= B_i when (mode_i = '0') else
    not(B_i) when (mode_i = '1');
```

2.3.2 Geração de s_C e s_P

Os sinais auxiliares **s_C** e **s_P** são utilizados para o cálculo antecipado do carry em cada bit, durante as operações de adição ou subtração. Esses sinais são obtidos por meio das operações lógicas **AND** e **XOR**, realizadas bit a bit entre os vetores **A_i** e **B_op**.

O laço `for` percorre todos os bits dos vetores **A_i** e **B_op**, executando as instruções a seguir:

$$s\_C[n] = A_i[n] \wedge B_{op}[n] \quad \text{(carry gerado)}$$

$$s\_P[n] = A_i[n] \oplus B_{op}[n] \quad \text{(carry propagado)}$$

Onde **n** representa a posição do bit (de 0 a 31).

Os sinais **s_C** e **s_P** armazenam, respectivamente, os valores correspondentes ao carry gerado e ao carry propagado.

```
GERADORCARRY: for nn in 0 to 31 generate
s_C(nn)     <= A_i(nn) and B_op(nn);
s_P(nn)     <= A_i(nn) xor B_op(nn);
end generate GERADORCARRY;
```

2.3.3 Geração de Carry Antecipado

Com os valores dos sinais **s_C** e **s_P**, torna-se possível determinar o carry bit a bit para cada posição de saída (`Carry(i)` até `Carry(31)`). Para `Carry(0)`, o cálculo é direto e depende exclusivamente do modo de operação: `'0'` indica adição, enquanto `'1'` representa subtração.

```
Carry(0)  <= '0' when mode_i = '0' else '1';
```

A partir disso, os próximos valores de carry são obtidos por:

```
Carry(1) <= s_C(0) or (s_P(0) and Carry(0));
Carry(2) <= s_C(1) or (s_P(1) and Carry(1));
Carry(3) <= s_C(2) or (s_P(2) and Carry(2));
```

Na sequência, o cálculo do carry passa a depender dos valores dos sinais **vecSize_i** e **mode_i**. Por exemplo, para `Carry(4)`, o cálculo segue a lógica:

```
Carry(4) <= '0' when (vecSize_i = "00" and mode_i = '0') else
            '1' when (vecSize_i = "00" and mode_i = '1') else
            s_C(3) or (s_P(3) and Carry(3));
```

Esse mesmo raciocínio é repetido para cada novo bloco de 4 bits (como 8, 12, 16, etc.), em função das variáveis **vecSize_i** e **mode_i**, assegurando a correta propagação do carry entre os blocos para vetores de maior dimensão:

```
Carry(8) <= '0' when (vecSize_i = "00" and mode_i = '0') else
            '1' when (vecSize_i = "00" and mode_i = '1') else
            '0' when (vecSize_i = "01" and mode_i = '0') else
            '1' when (vecSize_i = "01" and mode_i = '1') else
            s_C(7) or (s_P(7) and Carry(7));

Carry(9)  <= s_C(8) or (s_P(8) and Carry(8));
Carry(10) <= s_C(9) or (s_P(9) and Carry(9));
Carry(11) <= s_C(10) or (s_P(10) and Carry(10));

```

```
...

Carry(16) <= '0' when (vecSize_i = "00" and mode_i = '0') else
             '1' when (vecSize_i = "00" and mode_i = '1') else
             '0' when (vecSize_i = "01" and mode_i = '0') else
             '1' when (vecSize_i = "01" and mode_i = '1') else
             '0' when (vecSize_i = "10" and mode_i = '0') else
             '1' when (vecSize_i = "10" and mode_i = '1') else
             s_C(15) or (s_P(15) and Carry(15));

Carry(17) <= s_C(16) or (s_P(16) and Carry(16));
Carry(18) <= s_C(17) or (s_P(17) and Carry(17));
Carry(19) <= s_C(18) or (s_P(18) and Carry(18));
...
```

Após o cálculo dos carries, realiza-se a soma bit a bit para determinar o resultado final da operação.

```
GERADOR_SOMA : for mm in 0 to 31 generate
  s_Soma(mm) <= A_i(mm) xor B_op(mm) xor Carry(mm);
end generate GERADOR_SOMA;
```

O valor final é armazenado no sinal **S_o**, o qual contém o resultado da adição ou subtração paralela de n bits.

3 Resultados

Sinais de controle:

vecSize_i	Operação
00	8 somas de 4 bits
01	4 somas de 8 bits
10	2 somas de 16 bits
11	1 soma de 32 bits

mode_i	Operação
0	soma
1	substração

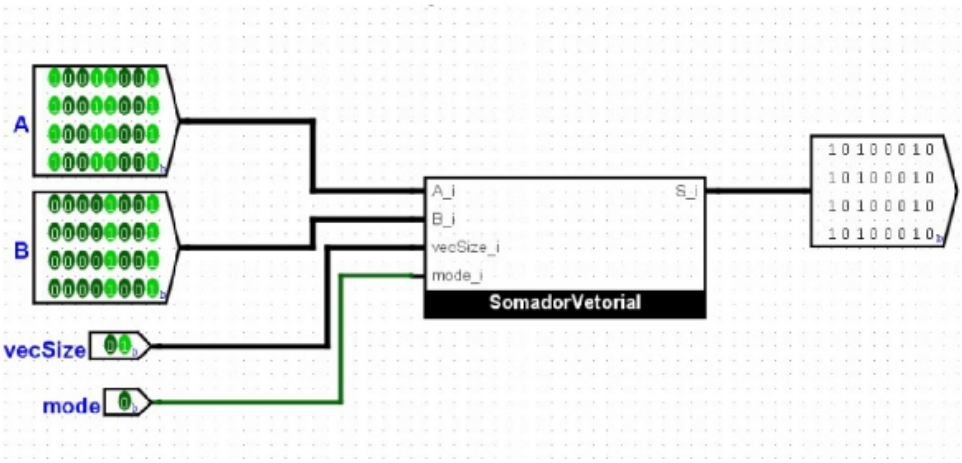

Figura 1: Teste 1

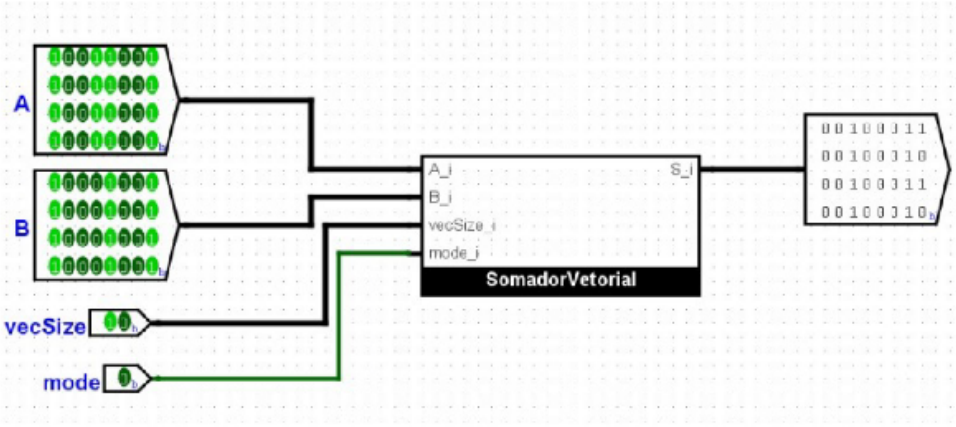

Figura 2: Teste 2

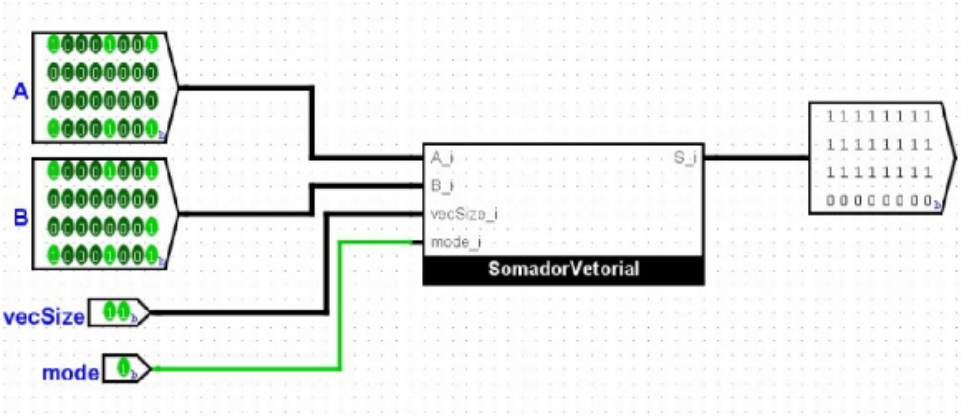

Figura 3: Teste 3

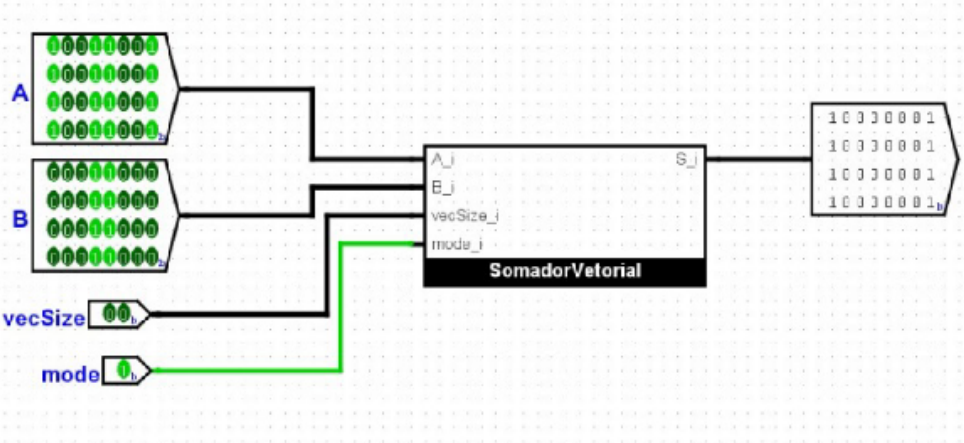

Figura 4: Teste 4