Universidade Federal do Rio de Janeiro

Politécnica UFRJ

Atividade Prática AVX

Alunos	Bruno Arosa Ferreira Jorge - 118068529
	Marco Antonio Machado Gomes - 119181369
	Wilson Albuquerque Ramos - 118092402
Disciplina	Arquitetura de Computadores
Professor	Diego Leonel Cadette Dutra

Rio de Janeiro, Julho de 2025

Conteúdo

1 Introdução

Com o avanço das tecnologias de hardware, especialmente nas arquiteturas modernas de processadores como as da Intel e AMD, tornou-se possível alcançar níveis mais altos de desempenho por meio do uso de instruções vetoriais. Essas instruções, conhecidas como Extensões Vetoriais Avançadas (AVX – **Advanced Vector Extensions**), permitem que diversas operações sejam realizadas em paralelo sobre conjuntos de dados (vetores), otimizando o uso dos recursos internos da CPU e reduzindo significativamente o tempo de execução de tarefas intensivas em processamento.

Inspirado por esse conceito, o objetivo deste trabalho é projetar, utilizando a ferramenta *Digital*, um circuito combinacional capaz de realizar operações de soma e subtração vetorial sobre operandos inteiros de 32 bits. O circuito deverá operar com vetores de 4, 8, 16 e 32 bits, definidos por um sinal de controle, buscando alta performance, ou seja, baixa latência na execução das operações.

2 Sobre o Projeto

2.1 Especificações Técnicas

O circuito desenvolvido deve ser capaz de operar sobre entradas de tamanhos variáveis. Para isso, os operandos de 32 bits (A_i e B_i) são divididos em sub-blocos de diferentes tamanhos: 4, 8, 16 ou 32 bits. A operação é realizada de forma vetorial, ou seja, em paralelo sobre cada sub-bloco, conforme o valor do sinal $vecSize_i$. A escolha entre soma ou subtração é controlada pelo sinal $mode_i$.

- **Operandos (32 bits cada):**
  - $A_i \rightarrow$ Primeiro operando
  - $B_i \rightarrow$ Segundo operando
- **Tamanho dos elementos do vetor (2 bits):**
  - $vecSize_i$:
    - 00 $\rightarrow$ Vetores de 4 bits (8 operações de 4 bits)
    - 01 $\rightarrow$ Vetores de 8 bits (4 operações de 8 bits)
    - 10 $\rightarrow$ Vetores de 16 bits (2 operações de 16 bits)
    - 11 $\rightarrow$ Vetor de 32 bits (1 operação de 32 bits)
- **Modo da operação (1 bit):**
  - $mode_i$:
    - 0 $\rightarrow$ Soma
    - 1 $\rightarrow$ Subtração
- **Saída (32 bits):**
  - $S_o \rightarrow$ Resultado da operação vetorial: $A_i \pm B_i$

2.2 Funcionamento do Circuito

2.2.1 Estrutura Modular por Blocos de 4 Bits

O circuito é composto por 8 módulos `ALU4bits`, cada um responsável por processar um segmento de 4 bits. A depender do valor de `vecSize_i`, esses módulos podem operar de forma independente ou interconectada via carry, permitindo a realização de operações vetoriais de diferentes tamanhos.

Cada módulo de 4 bits é uma unidade aritmética-lógica (ALU) capaz de realizar soma ou subtração. A interligação entre os módulos é controlada pelo vetor de sinal

`carry_in_enable`, que determina, bit a bit, se o carry-out do módulo anterior será propagado como carry-in para o próximo módulo. Isso permite configurar dinamicamente o tamanho das operações vetoriais (4, 8, 16 ou 32 bits), conforme o valor de `vecSize_i`.

2.2.2 Controle da Cadeia de Carry (`carry_in_enable`)

Esse sinal é um vetor de 8 bits, em que cada bit está associado a um dos oito módulos de 4 bits. Ele indica se o carry-in de um módulo será o carry-out do módulo anterior (bit = '1') ou se será gerado localmente (bit = '0'), o que significa que o bloco operará de forma independente.

- **4 bits:** Todos os bits de `carry_in_enable` são '0'. Os 8 blocos operam de forma totalmente independente.
- **8 bits:** Bits alternados (1, 3, 5, 7) são '1', conectando pares de blocos para formar 4 unidades de 8 bits.
- **16 bits:** Blocos agrupados em dois conjuntos de 4 módulos, com os bits correspondentes ativados para formar 2 unidades de 16 bits.
- **32 bits:** Todos os bits de `carry_in_enable` são '1', formando uma cadeia completa de carry entre os 8 módulos, resultando em uma única operação de 32 bits.

2.2.3 Operação de Cada ALU

Cada ALU4bits recebe dois vetores de 4 bits, um seletor (mode_i) e dois sinais adicionais:

- carry_in_enable: indica se o bloco está conectado à cadeia de carry;
- carry: valor real de carry-in recebido.

Dentro da ALU:

- O operando B_i é invertido (bit a bit) se a operação for subtração.
- O valor de Cin é:
  - 0: se a soma for sem carry;
  - 1: se a subtração for sem carry (para realizar o complemento de dois);
- O resultado é produzido por um somador de 4 bits com carry-out, que pode alimentar o próximo bloco se necessário.

3 Código

O somador vetorial foi desenvolvido utilizando a linguagem VHDL. O código está dividido em diferentes submódulos, apresentados nas subseções a seguir. Todo o código também está disponível no repositório do GitHub: `https://github.com/MarcoAGomes/ArqComp/tree/main/AVX`.

3.1 Somador/Subtrator Vetorial AVX

```
entity SomadorVetorialAVX is
  port (
    A_i: in std_logic_vector(31 downto 0);
    B_i: in std_logic_vector(31 downto 0);
    mode_i: in std_logic;
    vecSize_i: in std_logic_vector(1 downto 0);
    S_o: out std_logic_vector(31 downto 0));
end SomadorVetorialAVX;

architecture structural of SomadorVetorialAVX is
  component ALU4bits is
    port (
      X: in std_logic_vector(3 downto 0);
      Y: in std_logic_vector(3 downto 0);
      seletor: in std_logic;
      Cin: in std_logic;
      carry_in_enable: in std_logic;
      resultado: out std_logic_vector (3 downto 0);
      Cout: out std_logic);
  end component;

  signal carry_in_enable: std_logic_vector (7 downto 0);
  signal carry: std_logic_vector (8 downto 0);

begin
  carry_in_enable(0) <= '0';
  carry(0) <= '0';

  carry_in_enable(1) <= vecSize_i(1) OR vecSize_i(0);
  carry_in_enable(3) <= carry_in_enable(1);
  carry_in_enable(5) <= carry_in_enable(3);
  carry_in_enable(7) <= carry_in_enable(5);

  carry_in_enable(2) <= vecSize_i(1);
  carry_in_enable(6) <= carry_in_enable(2);

  carry_in_enable(4) <= vecSize_i(1) AND vecSize_i(0);
```

```
38
39    alu_gen: for i in 0 to 7 generate
40      alu_inst: ALU4bits port map(
41        A_i(i*4+3 downto i*4),
42        B_i(i*4+3 downto i*4),
43        mode_i,
44        carry(i),
45        carry_in_enable(i),
46        S_o(i*4+3 downto i*4),
47        carry(i+1));
48    end generate alu_gen;
49  end structural;
```

3.2 4 bits ALU para soma/subtração com carry chain controlado

```
1  entity ALU4bits is
2    port (
3      X: in std_logic_vector (3 downto 0);
4      Y: in std_logic_vector (3 downto 0);
5      seletor: in std_logic;
6      Cin: in std_logic;
7      carry_in_enable: in std_logic;
8      resultado: out std_logic_vector (3 downto 0);
9      Cout: out std_logic);
10 end ALU4bits;
11
12 architecture structural of ALU4bits is
13   component Somador4Bit is
14     port (
15       X: in std_logic_vector (3 downto 0);
16       Y: in std_logic_vector (3 downto 0);
17       Cin: in std_logic;
18       S: out std_logic_vector (3 downto 0);
19       Cout: out std_logic);
20   end component;
21   component MUX21_4Bits is
22     port (
23       X: in std_logic_vector (3 downto 0);
24       Y: in std_logic_vector (3 downto 0);
25       seletor:  in std_logic;
26       saida:   out std_logic_vector (3 downto 0));
27   end component;
28   signal operando_2: std_logic_vector (3 downto 0);
29   signal cin_somador: std_logic;
30
31 begin
```

```
32   MUX_21: MUX21_4Bits port map(Y, NOT Y, seletor, operando_2);
33
34   cin_somador <= (NOT (carry_in_enable) AND seletor) OR (carry_in_enable
         AND Cin);
35
36   ALU_4BIT: Somador4Bit port map (X, operando_2, cin_somador, resultado,
         Cout);
37 end structural;
```

3.3 Somador de 4 bits com carry in/out

```
1  entity Somador4Bit is
2    port (
3      X:  in std_logic_vector (3 downto 0);
4      Y: in std_logic_vector (3 downto 0);
5      Cin: in std_logic;
6      S: out std_logic_vector (3 downto 0);
7      Cout: out std_logic);
8  end Somador4Bit;
9
10 architecture structural of Somador4Bit is
11   component FullAdder1bit is
12     port (
13     A: in std_logic;
14     B: in std_logic;
15     Cin: in std_logic;
16     S: out std_logic;
17     Cout: out std_logic);
18   end component;
19
20   signal c1: std_logic;
21   signal c2: std_logic;
22   signal c3: std_logic;
23 begin
24   fa1: FullAdder1bit port map (X(0), Y(0), Cin, S(0), c1);
25   fa2: FullAdder1bit port map (X(1), Y(1), c1, S(1), c2);
26   fa3: FullAdder1bit port map (X(2), Y(2), c2, S(2), c3);
27   fa4: FullAdder1bit port map (X(3), Y(3), c3, S(3), Cout);
28 end structural;
```

3.4 Multiplexador 2:1 de 4 bits

```
1  entity MUX21_4Bits is
2    port (
3      X: in std_logic_vector (3 downto 0);
```

```
    Y: in std_logic_vector (3 downto 0);
    seletor: in std_logic;
    saida:  out std_logic_vector (3 downto 0));
end MUX21_4Bits;

architecture behavioral of MUX21_4Bits is
begin
  saida <= (X AND NOT (seletor & seletor & seletor & seletor)) OR
           (Y AND (seletor & seletor & seletor & seletor));
end behavioral;
```

3.5 Somador completo de 1 bit

```
entity FullAdder1bit is
  port (
    A: in std_logic;
    B: in std_logic;
    Cin: in std_logic;
    S: out std_logic;
    Cout: out std_logic);
end FullAdder1bit;

architecture dataflow of FullAdder1bit is
begin
  S <= A XOR B XOR Cin;
  Cout <= (A AND B) OR (B AND Cin) OR (A AND Cin);
end dataflow;
```

4 Resultados

Nesta seção, são apresentados os principais resultados obtidos a partir das simulações realizadas no software *Digital*. As simulações foram conduzidas para diferentes larguras de vetor (4, 8, 16 e 32 bits), avaliando o funcionamento do somador/subtrator vetorial tanto para operações de soma quanto de subtração.

4.1 Operações com Vetores de 4 bits

4.1.1 Soma

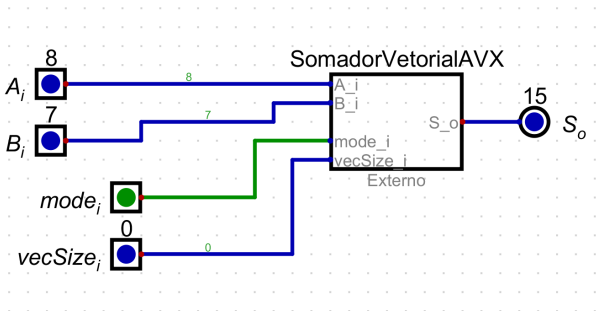

Figura 1: Somador de 4 bits. Operandos 1000_2 e 111_s. Resultado = 1111_2

4.1.2 Subtração

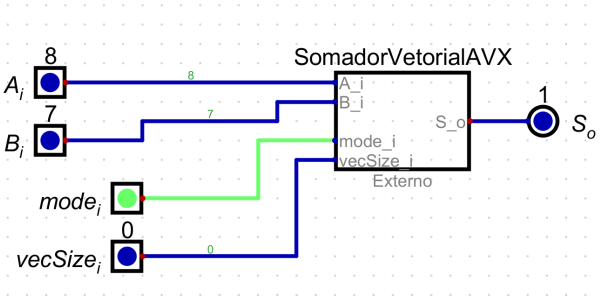

Figura 2: Subtrator de 4 bits. Operandos 1000_2 e 111_s. Resultado = 01_2

4.2 Operações com Vetores de 8 bits

4.2.1 Soma

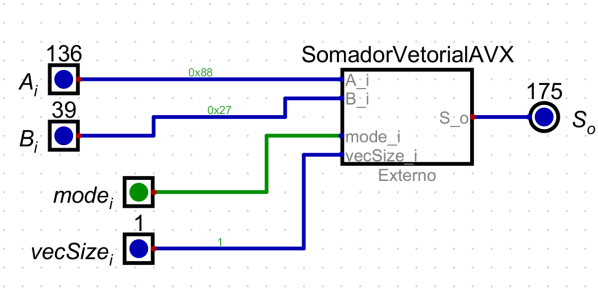

Figura 3: Somador de 8 bits. Operandos 10001000_2 e 100111_s. Resultado = 10101111_2

4.2.2 Subtração

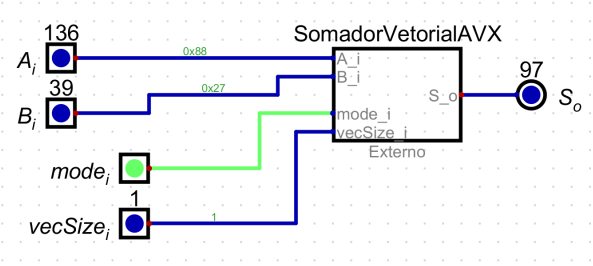

Figura 4: Subtrator de 8 bits. Operandos 10001000_2 e 100111_s. Resultado = 1100001_2

4.3 Operações com Vetores de 16 bits

4.3.1 Soma

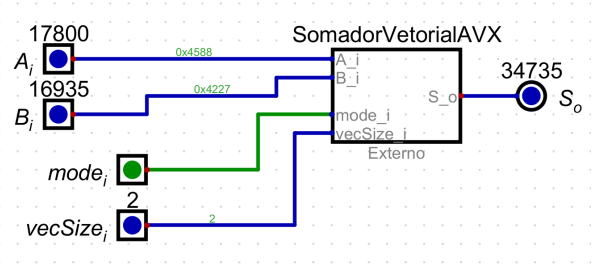

Figura 5: Somador de 16 bits. Operandos 1000101100001000_2 e 1000010001001111_s. Resultado = 1000011110101111_2

4.3.2 Subtração

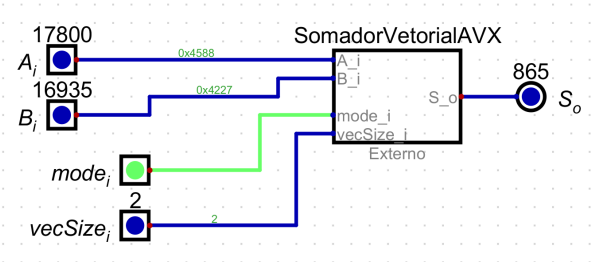

Figura 6: Subtrator de 16 bits. Operandos 100010110001000_2 e 100001000100111_s. Resultado = 1101100001_2

4.4 Operações com Vetores de 32 bits

4.4.1 Soma

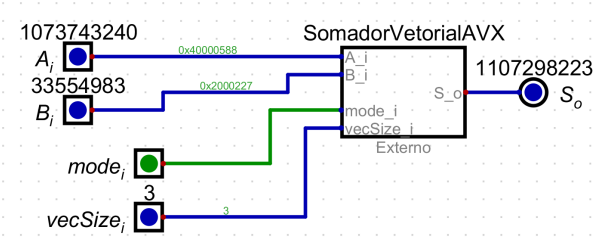

Figura 7: Somador de 32 bits. Operandos $10000000000000000000010110001000_2$ e $10000000000000001000100111_s$. Resultado = $10000100000000000000011110101111_2$

4.4.2 Subtração

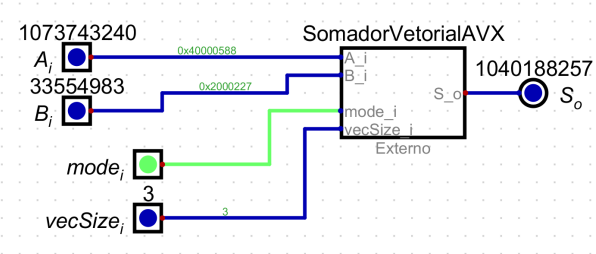

Figura 8: Subtrator de 32 bits. $10000000000000000000010110001000_2$ e $10000000000000001000100111_s$. Resultado = $11111000000000000000011011000001_2$

5 Referências

[1] Patterson, D. A., Hennessy, J. L.. – "Computer Organization and Design RISC-V Edition: The Hardware Software Interface"