

---

Pontifícia Universidade Católica de Minas Gerais  
Instituto de Ciências Exatas e Informática  
Departamento de Engenharia de Computação

# **Relatório: Trabalho Prático 1**

## **Multiplexador de Endereçamento e ULA**

**Professores:** Antônio Hamilton Magalhães

Bruno Luiz Dias Alves de Castro  
Rafael Ramos de Andrade

Belo Horizonte  
Campus Coração Eucarístico

17 de outubro de 2024

## Conteúdo

|          |   |          |
|----------|---|----------|
| <b>1</b> | <b>Introdução</b>                       | <b>3</b> |
| 1.1      | TP1 . . . . .                           | 3        |
| 1.1.1    | Objetivos . . . . .                     | 3        |
| <b>2</b> | <b>ADDR_MUX</b>                         | <b>3</b> |
| <b>3</b> | <b>ULA (Unidade Lógica Aritimética)</b> | <b>3</b> |
| 3.1      | ULA Proposta . . . . .                  | 4        |
| 3.2      | Implementação . . . . .                 | 4        |
| 3.2.1    | XOR, OR, AND e COM . . . . .            | 4        |
| 3.2.2    | ADD, SUB, INC e DEC . . . . .           | 5        |
| 3.2.3    | PASS_A e PASS_B . . . . .               | 5        |
| 3.2.4    | SWAP . . . . .                          | 5        |
| 3.2.5    | BS e BC . . . . .                       | 5        |
| 3.2.6    | RR e RL . . . . .                       | 5        |
| 3.2.7    | z_out . . . . .                         | 6        |
| 3.2.8    | c_out . . . . .                         | 6        |
| 3.2.9    | dc_out . . . . .                        | 6        |
| <b>4</b> | <b>Conclusion</b>                       | <b>6</b> |

# 1 Introdução

Durante as aulas da disciplina de Sistemas Reconfiguráveis, fomos introduzidos à linguagem VHDL. VHDL (**V**HSIC **H**ardware **D**escription **L**anguage) é uma linguagem de descrição de hardware. Com ela, podemos montar circuitos lógicos de maneira totalmente textual, o que garante à linguagem uma grande vantagem ante às soluções visuais.

## 1.1 TP1

Como primeiro trabalho prático (TP1), são propostas as montagens de dois circuitos utilizando a linguagem VHDL para a placa de desenvolvimento Altera:

- Um ADDR\_MUX (Multiplexador de Endereçamento) (Secção 2)
- Uma ULA (Unidade Lógica Aritimética) (Secção 3)

Ambos os circuitos devem ser desenvolvidos utilizando programação concorrente, ou seja, sem trechos sequenciais no código-fonte.

### 1.1.1 Objetivos

Entre os objetivos que temos com o desenvolvimento deste trabalho prático podemos listar:

- Aprender conceitos básicos da linguagem VHDL;
- Implementar utilizando programação concorrente os dois circuitos propostos;
- Compilar os circuitos e testar os resultados na placa de desenvolvimento Altera;

## 2 ADDR\_MUX

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetur id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

## 3 ULA (Unidade Lógica Aritimética)

A ULA (Unidade Lógica Aritimética) é um dos componentes mais básicos de um processador. Como o nome já indica, a ULA é responsável por todas as operações lógicas (como OR, AND e Shift) e aritméticas (como soma e subtração) realizadas em nosso circuito.

De maneira simplificada, a ULA receberá um comando, composto por seletores de operação e bits, e operandos. Na saída temos o resultado da operação desejada.

Como uma ULA opera de maneira concorrente, todas as operações implementadas são “executadas ao mesmo tempo”. Um Multiplexador é usado para selecionar a operação correta.

### 3.1 ULA Proposta

Neste trabalho prático, a ULA proposta deve possuir as seguintes funções:

| Função | op_code | Descrição              | z_out          | c_out              | dc_out                       |
|--------|---------|------------------------|----------------|--------------------|------------------------------|
| XOR    | 0000    | XOR Lógico             | 1, se res. = 0 | -                  | -                            |
| OR     | 0001    | OR Lógico              | 1, se res. = 0 | -                  | -                            |
| AND    | 0010    | AND Lógico             | 1, se res. = 0 | -                  | -                            |
| CLR    | 0011    | Limpa                  | 1              | -                  | -                            |
| ADD    | 0100    | Soma                   | 1, se res. = 0 | 1 se <i>carry</i>  | 0 se <i>carry</i> no nibble  |
| SUB    | 0101    | Subtração              | 1, se res. = 0 | 0 se <i>borrow</i> | 0 se <i>borrow</i> no nibble |
| INC    | 0110    | Incremento             | 1, se res. = 0 | -                  | -                            |
| DEC    | 0111    | Decremento             | 1, se res. = 0 | -                  | -                            |
| PASS_A | 1000    | Passa 'A'              | 1, se res. = 0 | -                  | -                            |
| PASS_B | 1001    | Passa 'B'              | 1, se res. = 0 | -                  | -                            |
| COM    | 1010    | Complemento            | 1, se res. = 0 | -                  | -                            |
| SWAP   | 1011    | Permuta <i>nibbles</i> | 1, se res. = 0 | -                  | -                            |
| BS     | 1100    | bit_sel = 1            | a.in[bit_sel]  | -                  | -                            |
| BC     | 1101    | bit_sel = 0            | a.in[bit_sel]  | -                  | -                            |
| RR     | 1110    | Rotação p/ dir.        | -              | a.in[0]            | -                            |
| RL     | 1111    | Rotação p/ esq.        | -              | a.in[7]            | -                            |

O sinais de entrada e saída são os seguintes:

| Nome    | Tamanho | Tipo          | Descrição                          |
|---------|---------|---------------|------------------------------------|
| a_in    | 8 bits  | <i>Input</i>  | Entrada de dados A                 |
| b_in    | 8 bits  | <i>Input</i>  | Entrada de dados B                 |
| c_in    | 1 bit   | <i>Input</i>  | Entrada de <i>carry</i>            |
| op_sel  | 4 bits  | <i>Input</i>  | Seletor de operação                |
| bit_sel | 3 bits  | <i>Input</i>  | Seletor de bit                     |
| r_out   | 8 bits  | <i>Output</i> | Saída do resultado                 |
| c_out   | 1 bit   | <i>Output</i> | Saída de <i>carry/borrow</i>       |
| dc_out  | 1 bit   | <i>Output</i> | Saída de <i>digit carry/borrow</i> |
| z_out   | 1 bit   | <i>Output</i> | Saída de zero                      |

### 3.2 Implementação

Nossa implementação, feita em VHDL, consiste na utilização de métodos e bibliotecas já implementadas, bem como lógica implementada por nós. Abaixo explicamos como foram implementadas cada uma das funções propostas.

#### 3.2.1 XOR, OR, AND e COM

Esses métodos foram implementados usando as funções lógicas disponíveis nativamente na linguagem VHDL.

```

aux <=
  a XOR b WHEN "0000",      — XOR
  a OR  b WHEN "0001",      — OR
  a AND b WHEN "0010",      — AND

  [...]

  NOT a  WHEN "1010",      — COM

```

### 3.2.2 ADD, SUB, INC e DEC

As funções de ADD (Adição), SUB (Subtração), INC (Incremento) e DEC (Decremento) foram implementadas usando operações aritméticas já inclusas na linguagem. Para as funções de ADD e SUB, os operandos A e B são utilizados, e já para as operações de INC e DEC, apenas o operando A e a constante 1.

```
aux <=
  a + b  WHEN "0100",    — ADD
  a - b  WHEN "0101",    — SUB
  a + 1  WHEN "0110",    — INC
  a - 1  WHEN "0111",    — DEC
```

### 3.2.3 PASS\_A e PASS\_B

Talvez as operações mais simples, ambas não utilizam nenhuma lógica. As entradas A e B são apenas “copiadas” para a saída.

```
aux <=
  a  WHEN "1000",    — PASS_A
  b  WHEN "1001",    — PASS_B
```

### 3.2.4 SWAP

A operação de SWAP é feita invertando os dois *nibbles*

```
aux <= '0' & a(3 DOWNIO 0) & a(7 DOWNIO 4) WHEN "1011", — SWAP
```

A concatenação com zero no início será explicada na implementação do *c\_out*.

### 3.2.5 BS e BC

As operações de BS (*bit set*) e BC (*bit clear*) implementam a seguinte lógica: para “setarmos” um bit para 1, podemos fazer uma operação OR da entrada com uma *string* de zeros, mas com um único um na posição que desejamos que seja igual à 1. Isso garantirá que este bit será sempre 1, e os demais são copiados. Podemos obter essa *string* efetuando um *shift lógico* de 1 um número *bit\_sel* de casas.

A operação BC funciona de maneira semelhante, mas deve ser feita com um AND no lugar do OR, e a *string* deve ser de 1s com um único zero na posição desejada. Para isso, fazemos mesma operação que anteriormente, mas invertemos o resultado.

```
aux <=
  a OR
    STD_LOGIC_VECTOR(SHIFT_LEFT(TO_UNSIGNED(1, 8),
      TO_INTEGER(UNSIGNED(bit_sel))))
    WHEN "1100",    — BS

  a AND
    NOT STD_LOGIC_VECTOR(SHIFT_LEFT(TO_UNSIGNED(1, 8),
      TO_INTEGER(UNSIGNED(bit_sel))))
    WHEN "1101",    — BC
```

### 3.2.6 RR e RL

As operações de RR (*Rotate Right*) e RL (*Rotate Left*) foram implementadas selecionando os bits necessários, e concatenando à entrada *c\_in*.

```

aux <=
    '0' & c_in & a(7 DOWNTO 1) WHEN "1110",      — RR
    '0' & a(6 DOWNTO 0) & c_in WHEN "1111";      — RL

```

### 3.2.7 z\_out

A saída `z_out` é usada para indicar que o resultado é zero para a maioria das operações, com duas exceções: a de CLR (já que o resultado sempre é zero), e as operações BS e BC onde ele deve ser o bit da entrada A na posição `bit_sel`.

Para isso, verificamos se a operação termina com a sequência "110" (utilizada por ambas as operações) e colocamos o bit correto na saída. Caso contrário, ela é 1 caso a saída seja zero.

```

z_out <=
    a(TO_INTEGER(UNSIGNED( bit_sel)))
    WHEN op_sel(3 DOWNTO 1) = "110" — BS and BC

    ELSE '1' WHEN aux(7 DOWNTO 0) = "00000000";

```

### 3.2.8 c\_out

Para identificarmos um *carry* (ou um *borrow*), há várias rotas que podemos tomar. Poderíamos, por exemplo, implementar circuitos lógicos capazes de identificar que ele ocorreu. Mas, para simplificar a operação, podemos chegar no mesmo resultado simplesmente aumentando o número de bits dos nossos dois operandos em 1. Podemos então copiar o bit mais significativo para a saída `c_out`. Caso um *carry* ou um *borrow* ocorra, esse bit será um.

```

r_out <= aux(7 DOWNTO 0);
c_out <= aux(8);

```

Para evitarmos que o 9º bit seja copiado para a saída, usamos um buffer `aux`, e apenas os bits de 7 à 0 são copiados para a saída `c_out`.

### 3.2.9 dc\_out

Para identificarmos um *carry* ou *borrow* em um *nibble*, podemos fazer algo similar ao que fizemos `c_out`. Criamos um auxiliar com 5 bits, e verificamos o copiamos para a saída, após executada a operação necessária.

```

aux_nibble <=
    ('0' & a(3 DOWNTO 0)) +
    ('0' + b(3 DOWNTO 0)) WHEN op_sel(0) = '0'
    ELSE ('0' & a(3 DOWNTO 0)) - ('0' & b(3 DOWNTO 0));

dc_out <= aux_nibble(4);

```

## 4 Conclusion

Quisque ullamcorper placerat ipsum. Cras nibh. Morbi vel justo vitae lacus tincidunt ultrices. Lorem ipsum dolor sit amet, consectetur adipiscing elit. In hac habitasse platea dictumst. Integer tempus convallis augue. Etiam facilisis. Nunc elementum fermentum wisi. Aenean placerat. Ut imperdiet, enim sed gravida sollicitudin, felis odio placerat quam, ac pulvinar elit purus eget enim. Nunc vitae tortor. Proin tempus nibh sit amet nisl. Vivamus quis tortor vitae risus porta vehicula.