Projetos de Circuitos Digitais em VHDL e FPGA

Cap. 4 - Projetos de Circuitos Combinacionais Aritméticos com VHDL

Prof. Erivelton Geraldo Nepomuceno

Engenharia Elétrica
UFSJ - Universidade Federal de São João del-Rei

13 de fevereiro de 2019

Objetivo

Objetivo Principal

Descrever os fundamentos para a criação de circuitos combinacionais aritméticos multibit.

Objetivos Específicos

- Projeto de Somadores. Ex.: Carry-Ripple, Carry-Lookahead.
- Projeto de Somadores/Subtratores sem e com sinal.
- Projeto de Multiplicadores. Ex.: Multiplicação via técnica pipeline.
- Projeto de Divisores.

Somador Carry-Ripple

- Contêm um conjunto de FAs (Full-Adders).
- Nesse circuito, o carry tem que se propagar serialmente por todos os estágios.
- Arquitetura lenta. Por outro lado, esse é geralmente o somador mais econômico em termos de área de silício e consumo de energia.

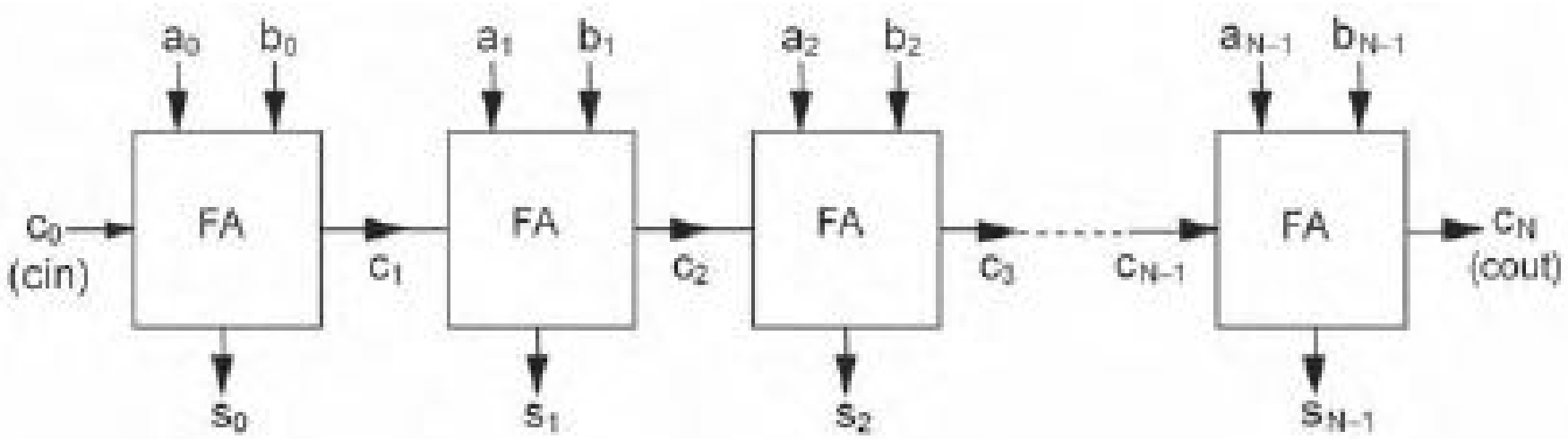


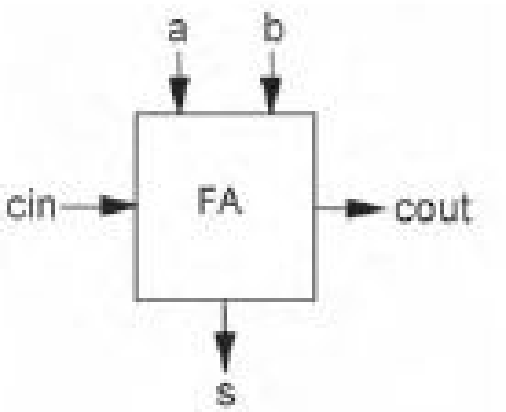
Figura 1: Somador Carry-Ripple.

Somador Carry-Ripple

- Ao observar a tabela verdade, é possível obter as seguintes expressões.

$$s = a \oplus b \oplus cin \tag{1}$$

$$cout = a \cdot b + a \cdot cin + b \cdot cin \tag{2}$$



cin	a	b	s	cout
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

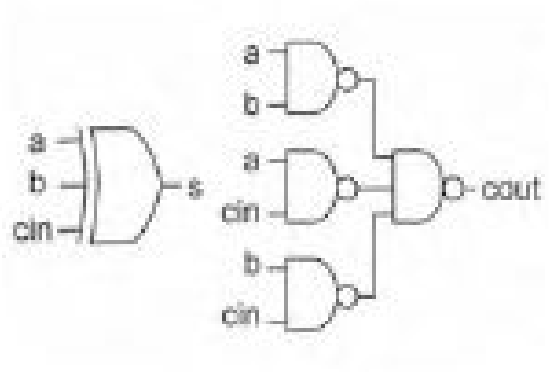


Figura 2: Somador Full-Adder.

Somador Carry-Ripple - Código em VHDL

```
1  ----------------------------------------------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ----------------------------------------------------------
5  ENTITY carry_ripple_adder IS
6      GENERIC (N : INTEGER := 8); --number of bits
7      PORT (a, b: IN STD_LOGIC_VECTOR(N-1 DOWNTO 0);
8            cin: IN STD_LOGIC;
9            s: OUT STD_LOGIC_VECTOR(N-1 DOWNTO 0);
10           cout: OUT STD_LOGIC);
11 END carry_ripple_adder;
12 ----------------------------------------------------------
13 ARCHITECTURE structure OF carry_ripple_adder IS
14 BEGIN
15     PROCESS(a, b, cin)
16         VARIABLE carry : STD_LOGIC_VECTOR (N DOWNTO 0);
17     BEGIN
18         carry(0) := cin;
19         FOR i IN 0 TO N-1 LOOP
20             s(i) <= a(i) XOR b(i) XOR carry(i);
21             carry(i+1) := (a(i) AND b(i)) OR (a(i) AND
22                carry(i)) OR (b(i) AND carry(i));
23         END LOOP;
24         cout <= carry(N);
25     END PROCESS;
26 END structure;
27 ----------------------------------------------------------
```

Somador Carry-Lookahead

- Somador com performance superior ao somador carry-ripple.
- Utiliza um número maior de componentes no FPGA, comparado ao somador anterior.

Somador Carry-Lookahead

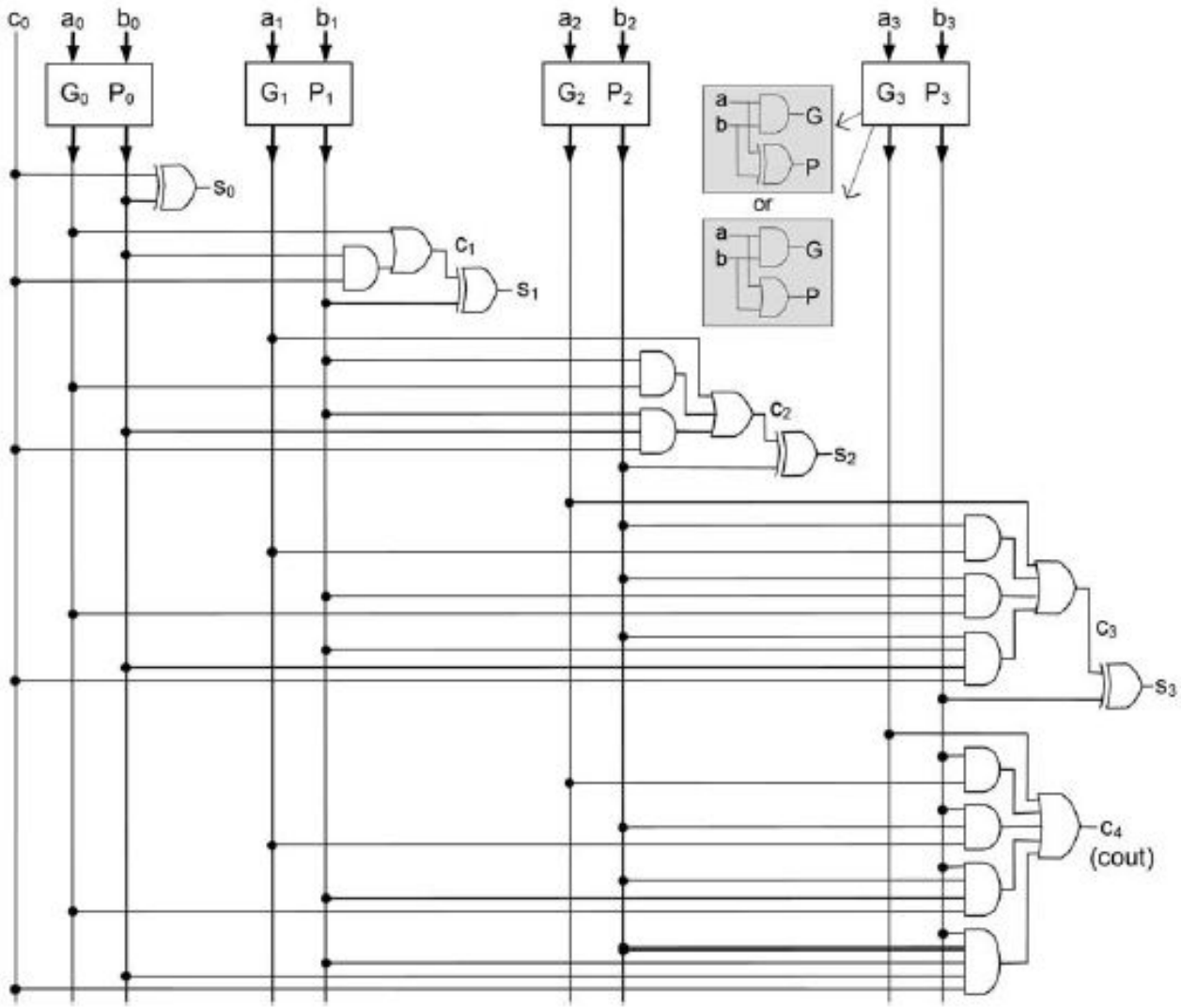


Figura 4: Somador Carry-Lookahead.

Somador Carry-Lookahead

- As equações que descrevem o circuito são:

$$G_i = a_i b_i \tag{3}$$

$$P_i = a_i \oplus b_i \tag{4}$$

$$c_1 = G_0 + P_0 c_0 \tag{5}$$

$$c_2 = G_1 + P_1 G_0 + P_1 P_0 c_0 \tag{6}$$

$$c_3 = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 c_0 \tag{7}$$

$$c_4 = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 c_0 \tag{8}$$

Somador Carry-Lookahead - 4 bits

```
1  ------- 4-bit carry-lookahead adder: ----------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ---------------------------------------------------
5  ENTITY carry_lookahead_adder IS
6      PORT (a, b: IN STD_LOGIC_VECTOR(3 DOWNTO 0);
7            cin: IN STD_LOGIC;
8            sum: OUT STD_LOGIC_VECTOR(3 DOWNTO 0);
9            cout: OUT STD_LOGIC);
10 END carry_lookahead_adder;
11 ---------------------------------------------------
12 ARCHITECTURE structure OF carry_lookahead_adder IS
13     SIGNAL G, P, c: STD_LOGIC_VECTOR(3 DOWNTO 0);
```

Figura 5: Somador Carry-Ripple.

Somador Carry-Lookahead - 4 bits

```
14 BEGIN
15      ------- Computation of G and P:
16      G <= a AND b;
17      P <= a XOR b;
18      ------- Computation of carry:
19      c(0) <= cin;
20      c(1) <= G(0)  OR
21              (P(0) AND cin);
22      c(2) <= G(1)  OR
23              (P(1) AND G(0)) OR
24              (P(1) AND P(0) AND cin);
25      c(3) <= G(2)  OR
26              (P(2) AND G(1)) OR
27              (P(2) AND P(1) AND G(0)) OR
28              (P(2) AND P(1) AND P(0) AND cin);
29      cout <= G(3)  OR
30              (P(3) AND G(2)) OR
31              (P(3) AND P(2) AND G(1)) OR
32              (P(3) AND P(2) AND P(1) AND G(0)) OR
33              (P(3) AND P(2) AND P(1) AND P(0) AND cin);
34      ------- Computation of sum:
35      sum <= P XOR c;
36 END structure;
37 ------------------------------------------------------
```

Figura 6: Somador Carry-Ripple.

Somador Carry-Lookahead - 32 bits

```
1  ------- Main code: ----------------------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ------------------------------------------------------
5  ENTITY big_adder IS
6      PORT (a, b: IN STD_LOGIC_VECTOR(31 DOWNTO 0);
7            cin: IN STD_LOGIC;
8            sum: OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
9            cout: OUT STD_LOGIC);
10 END big_adder;
11 -------------------------------------------------------
12 ARCHITECTURE big_adder OF big_adder IS
13     SIGNAL carry: STD_LOGIC_VECTOR(8 DOWNTO 0);
14     -----------------------------------------------
```

Figura 7: Somador Carry-Ripple de 32 bits (Parte 1).

Somador Carry-Lookahead - 32 bits

```
15      COMPONENT carry_lookahead_adder IS
16          PORT (a, b: IN STD_LOGIC_VECTOR(3 DOWNTO 0);
17                cin: IN STD_LOGIC;
18                sum: OUT STD_LOGIC_VECTOR(3 DOWNTO 0);
19                cout: OUT STD_LOGIC);
20      END COMPONENT;
21      ------------------------------------------------
22  BEGIN
23      carry(0) <= cin;
24      gen_adder: FOR i IN 1 TO 8 GENERATE
25          adder: carry_lookahead_adder PORT MAP (
26              a(4*i-1 DOWNTO 4*i-4),
27              b(4*i-1 DOWNTO 4*i-4),
28              carry(i-1),
29              sum(4*i-1 DOWNTO 4*i-4),
30              carry(i));
31      END GENERATE;
32      cout <= carry(8);
33  END big_adder;
34  ------------------------------------------------
```

Figura 8: Somador Carry-Ripple de 32 bits (Parte 2).

Somadores/Subtratores Sem e Com Sinal

- Quando usar a linguagem VHDL na construção de somadores/subtratores, deve-se preocupar apenas com os tipos de dados.

- Os operadores de adição (+) e subtração (-) já estão definidos nas bibliotecas padronizadas de VHDL.

Somador/Subtrator Com Entradas e Saídas do Tipo Inteiro

```
1  ------- Adder/subtracter with INTEGER: -------------
2  ENTITY adder_subtracter IS
3      GENERIC (N: INTEGER := 8); --number of input bits
4      PORT (a, b: IN INTEGER RANGE 0 TO 2**N-1;
5            sum, sub: OUT INTEGER RANGE 0 TO 2**N-1);
6  END adder_subtracter;
7  -------------------------------------------------
8  ARCHITECTURE adder_subtracter OF adder_subtracter IS
9  BEGIN
10     sum <= a + b;
11     sub <= a - b;
12 END adder_subtracter;
13 -------------------------------------------------
```

Figura 9: Código de um somador e subtrator.

Somador/Subtrator Com Entradas e Saídas do Tipo *STD_LOGIC_VECTOR*

- Deve deixar claro a natureza das entradas e saídas (SIGNED ou UNSIGNED).

```
1  ----- Adder/subtracter with STD_LOGIC_VECTOR: ----------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  USE ieee.std_logic_signed.all;
5  --USE ieee.std_logic_unsigned.all;
6  ---------------------------------------------------------
7  ENTITY adder_subtracter IS
8      GENERIC (N: INTEGER := 8); --number of input bits
9      PORT (a, b: IN STD_LOGIC_VECTOR(N-1 DOWNTO 0);
10           sum, sub: OUT STD_LOGIC_VECTOR(N-1 DOWNTO 0));
11 END adder_subtracter;
12 ---------------------------------------------------------
13 ARCHITECTURE adder_subtracter OF adder_subtracter IS
14 BEGIN
15     sum <= a + b;
16     sub <= a - b;
17 END adder_subtracter;
18 ---------------------------------------------------------
```

Figura 10: Código de um somador e subtrator.

Multiplicadores/Divisores Sem e Com Sinal

- As FPGAs já possuem multiplicadores e divisores embarcados.

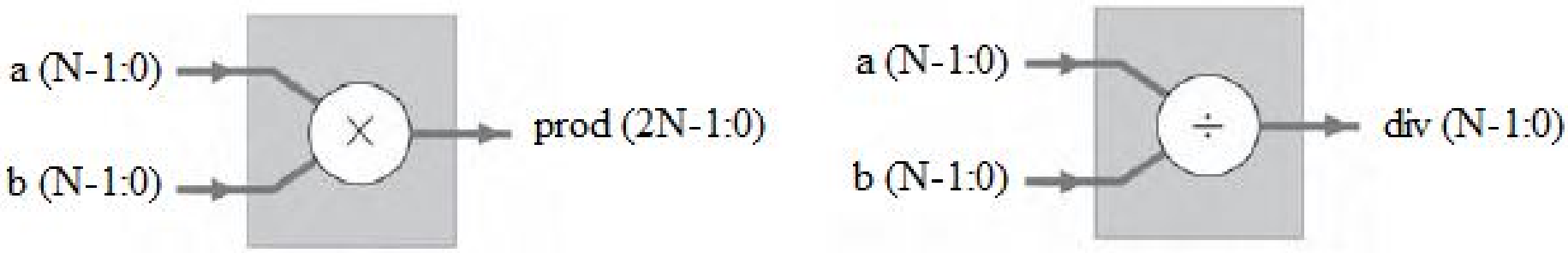


Figura 11: Representações das operações de multiplicação e divisão.

Multiplicador/Divisor Sem Sinal Com I/Os do Tipo Inteiro

```
1  --- Unsigned mult/div with INTEGER: --------------------
2  ENTITY multiplier_divider IS
3      GENERIC (N: INTEGER := 8); --number of bits
4      PORT (a, b: IN INTEGER RANGE 0 TO 2**N-1;
5            prod: OUT INTEGER RANGE 0 TO 4**N-1;
6            div: OUT INTEGER RANGE 0 TO 2**N-1;
7  END multiplier_divider;
8  ------------------------------------------------------
9  ARCHITECTURE multiplier_divider OF multiplier_divider IS
10 BEGIN
11     prod <= a * b;
12     div <= a / b;
13 END multiplier_divider;
14 ------------------------------------------------------
```

Figura 12: Código de um multiplicador/divisor sem sinal.

Multiplicador/Divisor Com Sinal Com I/Os do Tipo Inteiro

```
1  ------ Signed mult/div with INTEGER: ------------------
2  LIBRARY ieee;
3  USE ieee.numeric_std.all;
4  ---------------------------------------------------------
5  ENTITY multiplier_divider IS
6      GENERIC (N: INTEGER := 8); --number of bits
7      PORT (a, b: IN INTEGER RANGE 0 TO 2**N-1;
8            prod: OUT INTEGER RANGE 0 TO 4**N-1;
9            div: OUT INTEGER RANGE 0 TO 2**N-1);
10 END multiplier_divider;
11 ---------------------------------------------------------
12 ARCHITECTURE multiplier_divider OF multiplier_divider IS
13     SIGNAL a_sig, b_sig: SIGNED(N-1 DOWNTO 0);
14 BEGIN
15     a_sig <= TO_SIGNED(a, N);
16     b_sig <= TO_SIGNED(b, N);
17     prod <= TO_INTEGER(a_sig * b_sig);
18     div <= TO_INTEGER(a_sig / b_sig);
19 END multiplier_divider;
20 ---------------------------------------------------------
```

Figura 13: Código de um multiplicador/divisor com sinal.

Multiplicador/Divisor Com Sinal Com I/Os do Tipo *STD_LOGIC_VECTOR*

- Deve deixar claro a natureza das entradas e saídas (SIGNED ou UNSIGNED).

```
1  --- Signed mult/div with STD_LOGIC_VECTOR: -------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  USE ieee.numeric_std.all;
5  ------------------------------------------------------
6  ENTITY multiplier_divider IS
7      GENERIC (N: INTEGER := 8); --number of bits
8      PORT (a, b: IN STD_LOGIC_VECTOR(N-1 DOWNTO 0);
9            prod: OUT STD_LOGIC_VECTOR(2*N-1 DOWNTO 0);
10           div: OUT STD_LOGIC_VECTOR(N-1 DOWNTO 0));
11 END multiplier_divider;
12 ------------------------------------------------------
13 ARCHITECTURE multiplier_divider OF multiplier_divider IS
14     SIGNAL a_sig, b_sig: SIGNED(N-1 DOWNTO 0);
15 BEGIN
16     a_sig <= SIGNED(a);
17     b_sig <= SIGNED(b);
18     prod <= STD_LOGIC_VECTOR(a_sig * b_sig);
19     div <= STD_LOGIC_VECTOR(a_sig / b_sig);
20 END multiplier_divider;
21 ------------------------------------------------------
```

Figura 14: Código de multiplicação/divisão proposto.

Multiplicação via técnica pipeline

- Técnica que consome mais componentes da FPGA comparado a multiplicação convencional.

- Frequência de operação é superior quando comparado com outras técnicas.

Multiplicação via técnica pipeline

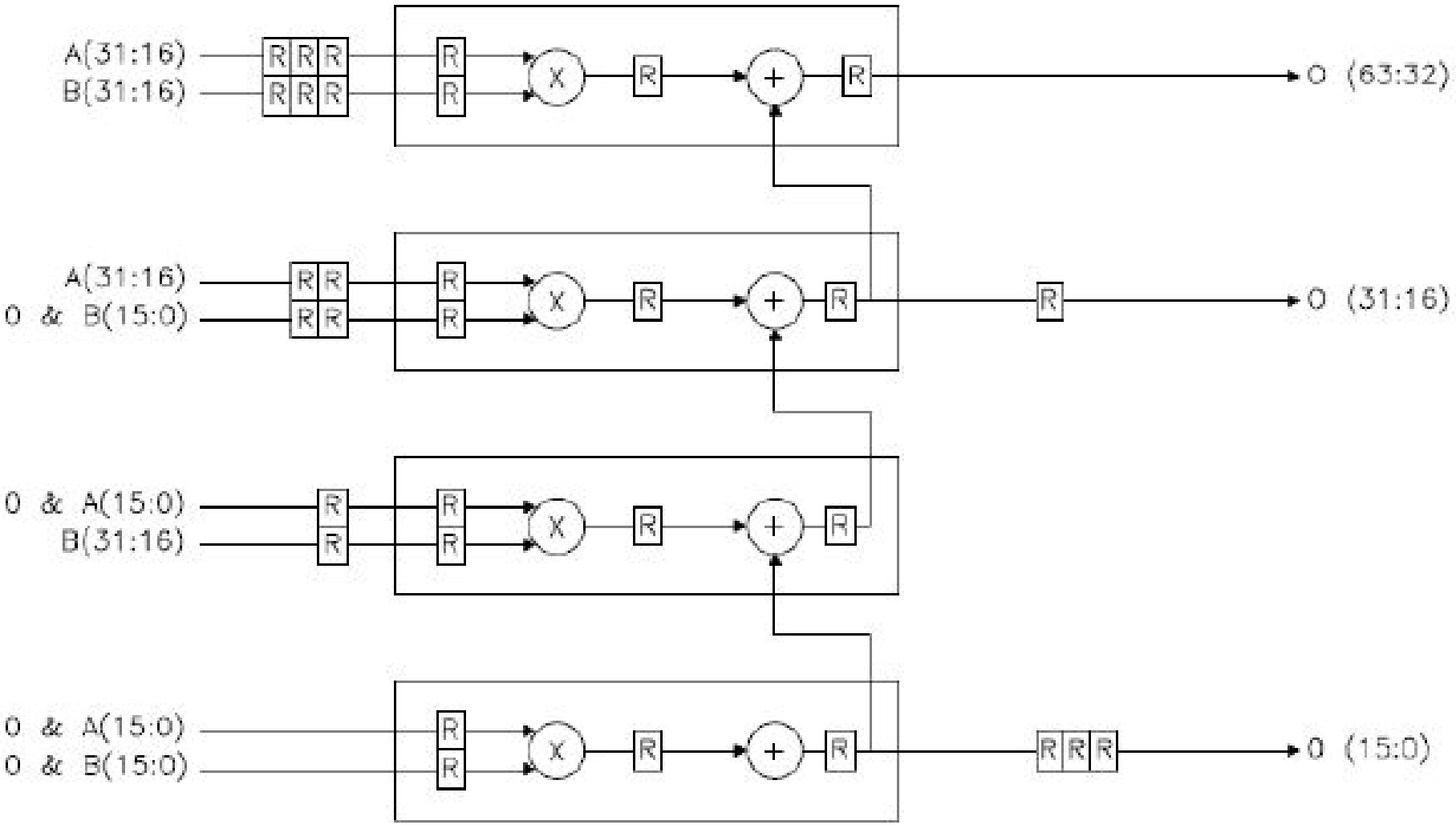


Figura 15: Multiplicação de entradas com 32 bits.

Multiplicação via técnica pipeline - Código

```
 2    library ieee ;
 3    use ieee.std_logic_1164.all;
 4    use ieee.numeric_std.all;
 5
 6  entity mult32pipe is
 7  port (
 8      i_clk       : in  std_logic;
 9      i_rstb      : in  std_logic;
10      i_ma        : in  std_logic_vector(31 downto 0);
11      i_mb        : in  std_logic_vector(31 downto 0);
12      o_m         : out std_logic_vector(63 downto 0));
13  end mult32pipe;
14
```

Figura 16: Código de multiplicação via pipeline com entradas de 32 bits - Parte 1.

Multiplicação via técnica pipeline - Código

```
15  architecture hardware of mult32pipe is
16    type p_operand_hi is array(0 to 3) of signed(15 downto 0);
17    type p_operand_lo is array(0 to 3) of signed(15 downto 0);
18    signal p_ma_hi          : p_operand_hi;
19    signal p_ma_lo          : p_operand_lo;
20    signal p_mb_hi          : p_operand_hi;
21    signal p_mb_lo          : p_operand_lo;
22
23    signal r_p1             : signed(33 downto 0);
24    signal r_p2             : signed(32 downto 0);
25    signal r_p3             : signed(32 downto 0);
26    signal r_p4             : signed(31 downto 0);
27
28    signal r_m1             : signed(33 downto 0);
29    signal r_m2             : signed(32 downto 0);
30    signal r_m3             : signed(32 downto 0);
31    signal r_m4             : signed(31 downto 0);
32
33    signal p_m1             : p_operand_lo;
34    signal p_m3             : signed(15 downto 0);
35
```

Figura 17: Código de multiplicação via pipeline com entradas de 32 bits - Parte 2.

Multiplicação via técnica pipeline - Código

```
36  begin
37
38    o_m(63 downto 32)  <= std_logic_vector(r_m4(31 downto 0));
39    o_m(31 downto 16)  <= std_logic_vector(p_m3);
40    o_m(15 downto  0)  <= std_logic_vector(p_m1(2));
41
42  p_mult : process(i_clk,i_rstb)
43    begin
44      if(i_rstb='0') then
45        p_ma_hi          <= (others=>(others=>'0'));
46        p_ma_lo          <= (others=>(others=>'0'));
47        p_mb_hi          <= (others=>(others=>'0'));
48        p_mb_lo          <= (others=>(others=>'0'));
49        p_m1             <= (others=>(others=>'0'));
50        p_m3             <= (others=>'0');
51
52        r_m1             <= (others=>'0');
53        r_m2             <= (others=>'0');
54        r_m3             <= (others=>'0');
55        r_m4             <= (others=>'0');
56        p_m1             <= (others=>(others=>'0'));
57        p_m3             <= (others=>'0');
```

Figura 18: Código de multiplicação via pipeline com entradas de 32 bits - Parte 3.

Multiplicação via técnica pipeline - Código

```
58    elsif(rising_edge(i_clk)) then
59      p_ma_hi         <= signed(i_ma(31 downto 16))&p_ma_hi(0 to p_ma_hi'length-2);
60      p_ma_lo         <= signed(i_ma(15 downto  0))&p_ma_lo(0 to p_ma_lo'length-2);
61      p_mb_hi         <= signed(i_mb(31 downto 16))&p_mb_hi(0 to p_mb_hi'length-2);
62      p_mb_lo         <= signed(i_mb(15 downto  0))&p_mb_lo(0 to p_mb_lo'length-2);
63      p_m1            <= r_m1(15 downto 0)&p_m1(0 to p_m1'length-2);
64      p_m3            <= r_m3(15 downto 0);
65
66      r_p1            <= signed('0'&p_ma_lo(0)) * signed('0'&p_mb_lo(0));
67      r_p2            <= signed('0'&p_ma_lo(1)) * p_mb_hi(1);
68      r_p3            <= p_ma_hi(2) * signed('0'&p_mb_lo(2));
69      r_p4            <= p_ma_hi(3) * p_mb_hi(3);
70
71      r_m1            <= r_p1;
72      r_m2            <= r_p2 + r_m1(31 downto 16);
73      r_m3            <= r_p3 + r_m2;
74      r_m4            <= r_p4 + r_m3(32 downto 16);
75    end if;
76  end process p_mult;
77
78  end hardware;
```

Figura 19: Código de multiplicação via pipeline com entradas de 32 bits - Parte 4.

Exercício

- Projetar uma unidade lógica aritmética (ULA). A partir de uma instrução, dada pelo OPCODE, a ULA deve realizar uma das seguintes operações: adição, subtração, multiplicação e divisão.

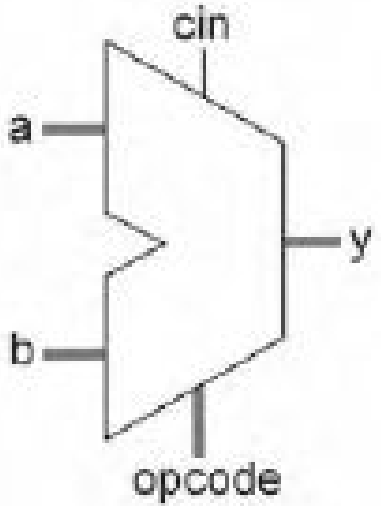


Figura 20: Unidade Lógica Aritmética.

Referências

- PEDRONI, Volnei A. **Digital electronics and design with VHDL**. Morgan Kaufmann, 2008.

- PEDRONI, Volnei A. **Eletrônica digital moderna e VHDL**. Rio de Janeiro, RJ: Elsevier, 2010.

- Surf VHDL. How to Implement a Pipeline Multiplier in VHDL. 2016. Disponível em: <https://surf-vhdl.com/how-to-implement-pipeline-multiplier-vhdl/>. Acesso em: 02/10/2018.