

Projeto de ULA

Cris – DRE: XXXXXXXXXX

Gus – DRE: XXXXXXXXXX

Leo – DRE: XXXXXXXXXX

¹Universidade Federal do Rio de Janeiro (UFRJ)

leonardongc@poli.ufrj.br

1. Enunciado

Fazer ULA de 2 operandos(**A** e **B**) de 4 bits com e 8 operações:

- $A + B$
- $A - B$
- $-A$
- $A + 1$
- Até 2 Operações de Comparação Bit a Bit
- O Restante das Operações A Escolha do Grupo Utilizando A e B

Com os Flags de **Zero, Negativo, Carry & Overflow**

2. Escolhas

2.1. Lista de Operações Atualizadas

Escolhemos utilizar XOR e ??? pois utilizam submódulos já existentes para outras operações e as operações de deslocamento de bits a direita e a esquerda.

- Soma
- Subtração
- Inversão
- Incremento de 1
- XOR
- Antecipação dos Carries
- Right Bitshift
- Left Bitshift

2.2. Projeto do Somador

Separamos a soma em duas etapas, uma que faz a antecipação dos carries e outra que faz a soma bit a bit com os carries.

2.2.1. Antecipador

Calculado Carry a Carry:

$$Cin_0 = 0$$

$$Cin_1 = A_0.B_0$$

$$\begin{aligned}
Cin_2 &= A_1.B_1 + Cin_1(A_1 + B_1) = A_1.B_1 + A_0.B_0(A_2 + B_2) \\
Cin_3 &= A_2.B_2 + Cin_2(A_2 + B_2) = A_2.B_2 + A_1.B_1 + A_0.B_0(A_2 + B_2)(A_2 + B_2) \\
Cin_4^* &= A_3.B_3 + Cin_3(A_3 + B_3) = A_3.B_3 + A_2.B_2 + A_1.B_1 + A_0.B_0(A_2 + B_2)(A_2 + B_2)(A_3 + B_3)
\end{aligned}$$

2.2.2. Somador

Com os carries antecipados é possível calcular a soma bit a bit:

$$F_x = A_x \oplus B_x \oplus Cin_x$$

2.3. Projeto do Inversor

O inversor de complemento de 2 tem a seguinte tabela:

A_3	A_2	A_1	A_0	F_3	F_2	F_1	F_0
0	0	0	0	0	0	0	0
0	0	0	1	1	1	1	1
0	0	1	0	1	1	1	0
0	0	1	1	1	1	0	1
0	1	0	0	1	1	0	0
0	1	0	1	1	0	1	1
0	1	1	0	1	0	1	0
0	1	1	1	1	0	0	1
1	0	0	0	0	0	0	0
1	0	0	1	0	1	1	1
1	0	1	0	0	1	1	0
1	0	1	1	0	1	0	1
1	1	0	0	0	1	0	0
1	1	0	1	0	0	1	1
1	1	1	0	0	0	1	0
1	1	1	1	0	0	0	1

Repare que para $A = 1000_{bin} = -8_{dec}$ pela regra ($F = A + 1$) deveria resultar em 1000, mas como não existe representação válida para o número 8 o resultado é 0000 e o flag de overflow é acionado.

$$F_3 = \overline{A_3}.A_2 + \overline{A_3}.A_1 + \overline{A_3}.A_0$$

$$F_2 = \overline{A_2}.A_1 + \overline{A_2}.A_0 + A_2.\overline{A_1}.\overline{A_0}$$

$$F_1 = \overline{A_1}.A_0 + A_1.\overline{A_0}$$

$$F_0 = A_0$$

2.4. Projeto do Subtrator

Com os módulos de soma e inversão prontos é possível utilizar seus submódulos para construir uma operação de subtração.

2.5. Projeto do Incremento de 1

Utilizamos o módulo de soma substituindo o operando B pelo vetor $B_3, B_2, B_1, B_0 = 0$ e a entrada $Cin_0 = 1$

2.6. Projeto do Comparador XOR

Desconsiderando o submódulo antecipador na soma o submódulo somador vale $A \oplus B$

$$F_x = A_x \oplus B_x$$

2.7. Projeto da Antecipação de Carry como Operação

É possível considerar o antecipador de Carry como uma operação por si só que já foi descrita no item 2.2.1.

2.8. Projeto do Left Bit Shifter

Montando a Tabela Verdade Para operações de Shift a esquerda é possível verificar que é certo que não haverá resultado não nulo para $B > 3$, portanto podemos considerar apenas os 2 bits menos significativos do operando B : $B_1 \& B_0$:

B_1	B_0	F_3	F_2	F_1	F_0
0	0	A_3	A_2	A_1	A_0
0	1	A_2	A_1	A_0	0
1	0	A_1	A_0	0	0
1	1	A_0	0	0	0

$$F_3 = \overline{B_1} \cdot \overline{B_0} \cdot A_3 + \overline{B_1} \cdot B_0 \cdot A_2 + B_1 \cdot \overline{B_0} \cdot A_1 + B_1 \cdot B_0 \cdot A_0$$

$$F_2 = \overline{B_1} \cdot \overline{B_0} \cdot A_2 + \overline{B_1} \cdot B_0 \cdot A_1 + B_1 \cdot \overline{B_0} \cdot A_0$$

$$F_1 = \overline{B_1} \cdot \overline{B_0} \cdot A_1 + \overline{B_1} \cdot B_0 \cdot A_0$$

$$F_0 = \overline{B_1} \cdot \overline{B_0} \cdot A_0$$

2.9. Projeto do Right Bit Shifter

O Shift a direita por sua vez forma uma tabela similar com as mesmas entradas:

B_1	B_0	F_3	F_2	F_1	F_0
0	0	A_3	A_2	A_1	A_0
0	1	0	A_3	A_2	A_1
1	0	0	0	A_3	A_2
1	1	0	0	0	A_3

$$F_3 = \overline{B_1} \cdot \overline{B_0} \cdot A_3$$

$$F_2 = \overline{B_1} \cdot \overline{B_0} \cdot A_2 + \overline{B_1} \cdot B_0 \cdot A_3$$

$$F_1 = \overline{B_1} \cdot \overline{B_0} \cdot A_1 + \overline{B_1} \cdot B_0 \cdot A_2 + B_1 \cdot \overline{B_0} \cdot A_3$$

$$F_0 = \overline{B_1} \cdot \overline{B_0} \cdot A_0 + \overline{B_1} \cdot B_0 \cdot A_1 + B_1 \cdot \overline{B_0} \cdot A_2 + B_1 \cdot B_0 \cdot A_3$$

Como as operações de translado de bits ocorrem num mesmo domínio, é possível combiná-las acoplando suas tabelas e diferenciando as por um único bit de seleção de operação S formando um único submódulo:

S	B_1	B_0	F_3	F_2	F_1	F_0
0	0	0	A_3	A_2	A_1	A_0
0	0	1	0	A_3	A_2	A_1
0	1	0	0	0	A_3	A_2
0	1	1	0	0	0	A_3
1	0	0	A_3	A_2	A_1	A_0
1	0	1	A_2	A_1	A_0	0
1	1	0	A_1	A_0	0	0
1	1	1	A_0	0	0	0

Nota: as operações de bitshift estão apenas interessadas no vetor de bits, sendo a representação de sinal ignorada para isso.

3. Execução

3.1. Código dos Submódulos

3.1.1. Módulo Antecipador

```
entity antecipador is
    Port ( A,B : in  STD_LOGIC_VECTOR (3 downto 0);
          Y : out
          STD_LOGIC_VECTOR (3 downto 0));
end antecipador;

    architecture Behavioral of antecipador is
        signal S1,S2,S3: std_logic;

    begin
        Y (0) <= '0';
        S1 <= A(0) and B(0);
        Y(1) <= S1;
        S2 <= (((A(1) and B(1)) or S1) and (A(1) or B(1)));
        Y (2) <= S2;
        S3 <= ( ( A (2) and B (2)) or S2) and (A (2) or B(2)));
        Y(3) <= S3;

    end Behavioral;
```

3.1.2. Módulo Somador

```
entity somador is
    Port ( A,B : in  STD_LOGIC_VECTOR (3 downto 0);
          Y : out
          STD_LOGIC_VECTOR (3 downto 0));
```

```

end somador;

architecture Behavioral of somador is

component antecipador
    port(A,B : in    std_logic_vector(3 downto 0);
          Y : out   std_logic_vector(3 downto 0));
end component;

signal Ya: std_logic_vector(3 downto 0);

begin
    ax: antecipador port map(A,B,Ya);

    Y(0) <= A(0) xor B(0) xor Ya(0);
    Y(1) <= A(1) xor B(1) xor Ya(1);
    Y(2) <= A(2) xor B(2) xor Ya(2);
    Y(3) <= A(3) xor B(3) xor Ya(3);

end Behavioral;

```

3.1.3. Módulo Inversor

```

entity inversor is port
(
    A    : in    std_logic_vector(3 downto 0);
    Y    : out   std_logic_vector(3 downto 0)
);

end inversor;

architecture Behavioral of inversor is

component somador
    port(A,B : in    std_logic_vector(3 downto 0);
          Y : out   std_logic_vector(3 downto 0));
end component;

signal sum: std_logic_vector(3 downto 0);

begin
    ax: somador port map((not A),"0001",sum);

    Y <= sum;

```

```
end Behavioral;
```

3.1.4. Módulo Subtrator

```
entity subtrator is
    Port ( A,B : in  STD_LOGIC_VECTOR (3 downto 0);
          Y : out
STD_LOGIC_VECTOR (3 downto 0));
end subtrator;

architecture Behavioral of subtrator is

    component somador
        port(A,B : in  std_logic_vector(3 downto 0);
              Y : out  std_logic_vector(3 downto 0));
    end component;

    component inversor
        port(A : in  std_logic_vector(3 downto 0);
              Y : out  std_logic_vector(3 downto 0));
    end component;

    signal Ya, Ys: std_logic_vector(3 downto 0);

begin
    ax: inversor port map(B,Ya);
    as: somador port map(A,Ya,Ys);
    Y <= Ys;

end Behavioral;
```

3.1.5. Módulo Incremento

```
entity incremento is
    Port ( A: in  STD_LOGIC_VECTOR (3 downto 0);
          Y : out
STD_LOGIC_VECTOR (3 downto 0));
end incremento;

architecture Behavioral of incremento is

    component antecipador
        port(A,B : in  std_logic_vector(3 downto 0);
```

```

                                Y : out  std_logic_vector(3 downto 0));
end component;

signal Ya: std_logic_vector(3 downto 0);

begin
    ax: antecipador port map(A,"0001",Ya);

    Y(0) <= A(0) xor '1' xor Ya(0);
    Y(1) <= A(1) xor '0' xor Ya(1);
    Y(2) <= A(2) xor '0' xor Ya(2);
    Y(3) <= A(3) xor '0' xor Ya(3);

end Behavioral;

```

3.1.6. Módulo Right Shifter

```

entity rbs is
port(A,B : in  std_logic_vector(3 downto 0);
      Y : out  std_logic_vector(3 downto 0));
end rbs;

architecture Behavioral of rbs is

begin
Y(3) <= ((not B(1)) and (not B(0)) and A(3));
Y(2) <= (((not B(1)) and (not B(0)) and A(2))
or (((not B(1)) and B(0)) and A(3)));
Y(1) <= (((not B(1)) and (not B(0)) and A(1))
or (((not B(1)) and B(0)) and A(2))
or ((B(1) and (not B(0))) and A(3)));
Y(0) <= (((not B(1)) and (not B(0)) and A(0))
or (((not B(1)) and B(0)) and A(1)) or ((B(1)
and (not B(0))) and A(2))
or ((B(1) and B(0)) and A(3)));

end Behavioral;

```

3.1.7. Módulo Left Shifter

```

entity lbs is
port(A,B : in  std_logic_vector(3 downto 0);

```

```

                                Y : out  std_logic_vector(3 downto 0));
end lbs;

architecture Behavioral of lbs is

begin
Y(0) <= ((not B(1)) and (not B(0)) and A(0));
Y(1) <= (((not B(1)) and (not B(0)) and A(1))
        or (((not B(1)) and B(0)) and A(0)));
Y(2) <= (((not B(1)) and (not B(0)) and A(2))
        or (((not B(1)) and B(0)) and A(1))
        or ((B(1) and (not B(0))) and A(0)));
Y(3) <= (((not B(1)) and (not B(0)) and A(3))
        or (((not B(1)) and B(0)) and A(2))
        or ((B(1) and (not B(0))) and A(1))
        or ((B(1) and B(0)) and A(0)));

end Behavioral;

```

3.1.8. Controlador da ULA

```

entity ula is port
(
A,B:  in   std_logic_vector(3 downto 0);
Y      : out   std_logic_vector(3 downto 0);
F : out std_logic_vector (3 downto 0);
sel    :  in   std_logic_vector(2 downto 0)
);

end ula;

architecture Behavioral of ula is

component antecipador
port(A,B : in   std_logic_vector(3 downto 0);
Y : out  std_logic_vector(3 downto 0));
end component;

component somador
port(A,B : in   std_logic_vector(3 downto 0);
Y : out  std_logic_vector(3 downto 0));
end component;

component subtrator

```



```

port(A,B : in    std_logic_vector(3 downto 0);
Y : out  std_logic_vector(3 downto 0));
end component;

```

```

component incremento
port(A : in    std_logic_vector(3 downto 0);
Y : out  std_logic_vector(3 downto 0));
end component;

```

```

component inversor
port(A : in    std_logic_vector(3 downto 0);
Y : out  std_logic_vector(3 downto 0));
end component;

```

```

component lbs
port(A,B : in    std_logic_vector(3 downto 0);
Y : out  std_logic_vector(3 downto 0));
end component;

```

```

component rbs
port(A,B : in    std_logic_vector(3 downto 0);
Y : out  std_logic_vector(3 downto 0));
end component;

```

```

signal AC: std_logic_vector(3 downto 0);
signal sum,inverse ,sub ,incr ,rs ,ls: std_logic_vector(3 downto 0);
signal seletor: std_logic_vector(2 downto 0):= "001";
signal seg: std_logic_vector(3 downto 0):= "0100";
signal Y0,Y1,Y2,Y3: std_logic;

```

```

begin
ax: antecipador port map(A,B,AC);
aa: inversor port map(A,inverse);
ab: subtrator port map(A,B,sub);
ad: incremento port map(A,incr);
s: somador port map(A,seg ,sum);
af: lbs port map(A,B, ls);
ag: rbs port map(A,B, rs);

```

```

process(A,B, seletor)
begin
case seletor is

```

```

when "000" => Y <= sum;
Y3 <= sum(3);
F(0) <= not((sum(0) or sum(1)) or (sum(2) or sum(3)));

```

```

F(1) <= Y3;
F(2) <= ((A(3) xnor B(3)) and AC(3));
F(3) <= ((A(3) xnor B(3)) and AC(3));

when "001" => Y <= inverse;-- inversor
F(0) <= ((A(0) and A(1)) and (A(2) and A(3)));
F(1) <= (A(3) and (A(1) or (A(2) or A(0))));
F(2) <= A(3) and not((A(1) or (A(2) or A(0))));
F(3) <= '0';

when "010" => Y <= sub; -- sub
F(0) <= (((A(0) xnor B(0)) and (A(1)
        xnor B(1))) and ((A(2) xnor B(2)) and (A(3) xnor B(3))));
F(1) <= sub(3);
F(2) <= ((A(3) xnor B(3)) and AC(3));
F(3) <= ((A(3) xnor B(3)) and AC(3));

when "011" => Y <= incr;-- incremento
F(0) <= ((A(0) and A(1)) and (A(2) and A(3)));
F(1) <= A(3) and not(((A(0) and A(1)) and (A(2) and A(3))));
F(2) <= not A(3) and (A(1) and (A(2) and A(3)));
F(3) <=((A(0) and A(1)) and (A(2) and A(3)));

when "100" => Y <= A xor B;-- xor
F(0) <= (((A(0) xnor B(0)) and (A(1) xnor B(1)))
        and ((A(2) xnor B(2)) and (A(3) xnor B(3))));
F(1) <= '0';
F(2) <= '0';
F(3) <= '0';

when "101" => Y <= AC;-- AC
F(0) <= not ((AC(0) and AC(1)) and (AC(2) and AC(3)));
F(1) <= '0';
F(2) <= AC(3);
F(3) <= AC(3);

when "110" => Y <= ls;-- lsb
F(0) <= '0';
F(1) <= '0';
F(2) <= '0';
F(3) <= '0';

when "111" => Y <= rs;-- rsb
F(0) <= '0';
F(1) <= '0';
F(2) <= '0';

```

```
F(3) <= '0';
```

```
when others => Y <= "0000" ;
```

```
end case;
```

```
end process;
```

```
end Behavioral;
```

3.2. Implementação dos Flags

3.2.1. Flag de Zero

O flag de zero será acionado para cada operação da forma:

- **Soma**
Se um operando for inverso do outro(em complemento de 2) ou os números forem zero.
- **Inversão**
Se o vetor de entrada valer 0.
- **Subtração**
Se o os operandos forem iguais.
- **Incremento de 1**
Se o operando for o vetor 1111
- **Comparador XOR**
Se os operandos forem iguais.
- **Antecipador de Carry**
O flag de zero é acionado quando nenhum carry é acionado: $C_0, C_1, C_2, C_3 = 0$
- **RBS**
Se o operando que determina o deslocamento (B) for maior que o índice do 1 mais a esquerda do operando (A) aciona o flag de 0.
- **LBS**
Se o vetor de entrada A valer 0.

3.2.2. Flag de Negativo

Da mesma forma que o flag de 0, flag de negativo será acionado:

- **Soma**
Se os operandos forem negativos ou se um operando negativo for maior que o positivo.
- **Inversão**
Se o vetor de entrada for diferente de 0 e tiver o bit de sinal 0.
- **Subtração**
A depender do bit de sinal dos operandos A e B e dos seus módulos: Se $A_3 = B_3 = 0$ e $|B| > |A|$, ou $A_3 = B_3 = 1$ e $|A| > |B|$, ou $A_3 = 1 \& B_3 = 0$
- **Incremento de 1**
Se o operando tiver bit de sinal igual a 1 e não for o vetor 1111.

- **Comparador XOR**

Essa operação não produz negativos

- **Antecipador de Carry**

O Antecipador não gera números negativos.

- **RBS e LBS**

As operações de bitshift desconsideram a representação de sinal, portanto não aciona o flag de negativo.

3.2.3. Flag de Carry

O flag de zero serve para guardar informação de *vai um* para uma operação futura:

- **Soma**

Tendo os bits de sinal iguais o flag de carry será o sinal Cin_3 do submódulo antecipador.

- **Inversão**

Inversão apenas ativa carry para o número -8 (vetor 1000) com a saída 0000 zero.

- **Subtração**

Da mesma forma que a soma, com o mesmo sinal.

- **Incremento de 1**

o único vetor que vai um é o número 7, pois é o número que aciona Cin_3 na soma.

- **Comparador XOR**

Um comparador não gera sinal de vai um.

- **Antecipador de Carry**

O Flag de Carry do Antecipador é a saída $C_4 = A_3.B_3 + A_2.B_2 + A_1.B_1 + A_0.B_0(A_2 + B_2)(A_2 + B_2)(A_3 + B_3)$

- **RBS**

Translados de bits à direita não geram carries.

- **LBS**

Como o bit de sinal é ignorado para essa operação não faz sentido considerar o vai um.

3.2.4. Flag de Overflow

Por sua vez o flag de overflow será acionado quando os 4 bits não forem o suficiente para representar o resultado:

- **Soma**

O flag de Overflow é o flag de carry da soma, pois é acionado seguindo os mesmos critérios.

- **Inversão**

Apenas o vetor 1000 gera overflow, pois não há representação em 4 bits com sinal para o número 8.

- **Subtração**

Apenas se $A_3 = B_3$ e se o carry do terceiro bit da operação for gerado $Cin_3 = 1$

- **Incremento de 1**

O mesmo critério da soma.

- **Comparador XOR**

Comparadores bit a bit não geram overflow.

- **Antecipador de Carry**

O mesmo critério do Flag de Carry é aplicado aqui.

- **RBS & LBS**

Nenhum BitShifter gera overflow, pois não há preocupação com o espaço de representação.