Dispositivos Lógicos Progamáveis II

Consumo de área em bin2bcd e contagem binária

Arthur Cadore Matuella Barcella e Gabriel Luiz Espindola Pedro

02 de Abril de 2023

Sumário

1. Introdução

Neste relatório, está explicita duas implementações de soma de números representados em binário. Os números a serem somados serão inseridos no circuito como dois vetores de bits (8bits cada), desta forma, o somador poderá somar números de 0 a 99.

A primeira implementação será realizada diretamente em BCD, ou seja, a soma dos vetores de bits é realizada em BCD e em seguida convertida para SSD (representação para display de 7-segmentos) enquanto que na segunda implementação, a soma é realizada em binário primeiro, e em seguida convertida para BCD e posteriormente para SSD.

O objetivo deste relatório é comparar o consumo de área e o tempo de propagação entre as duas implementações, e determinar qual é a mais eficiente e adequada para cenários de aplicação.

2. Implementação com somador BCD

A primeira implementação foi realizada diretamente com a soma em BCD e a conversão para SSD. Para isso, utilizamos um somador BCD de 4 dígitos, que soma dois números BCD de 4 dígitos e retorna o resultado em BCD.

Como a soma é de números com 2 algarismos, é necessário 8 bits (4 mais 4) para a representação em BCD. Portanto, a entrada do somador é de dois vetores de 8bits.

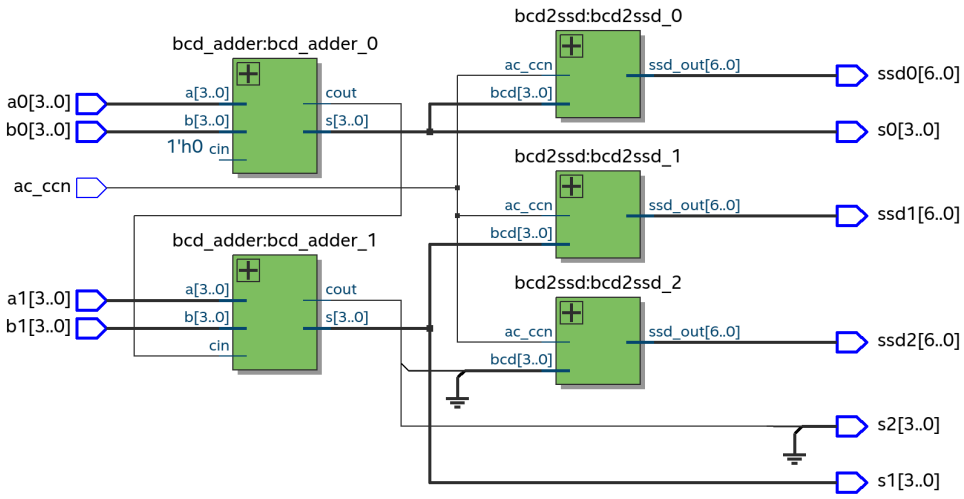

Figura 1: RTL - Circuito da Primeira Parte
Figura elaborada pelo autor

Como podemos ver na imagem apresentada acima, o RTL mostra a implementação do somador BCD de 4 dígitos. O somador recebe dois vetores de 4 bits cada (um para dezena e outro para a unidade), e retorna um vetor de 4 bits, para o somador da unidade, o carry out é enviado para o somador da dezena.

Em seguida, são utilizados três conversores BCD para SSD, que convertem um número BCD de 4 bits para um display de 7 segmentos, para a representação do número somado.

Para verificar seu funcionamento, na primeira etapa, realizamos um testebench para verificar se a implementação está correta. Abaixo está a simulação do testbench:

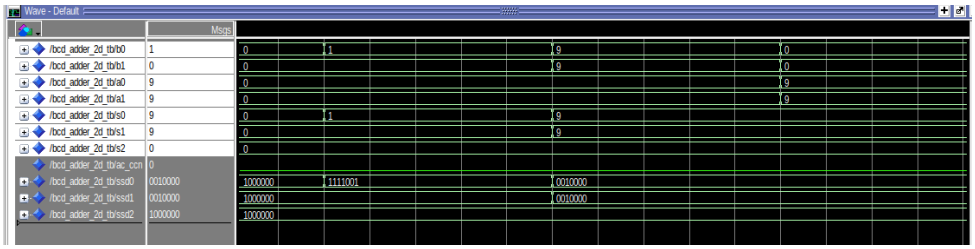

Figura 2: Simulação RTL - Primeira parte
Figura elaborada pelo autor

3. Implementação com somador binário e conversor BCD

Para realizar a segunda etapa da atividade, implementamos quatro códigos VHDL para a contagem ocorrer em binário (de maneira mais simples), e em seguida realizar sua conversão para BCD (oque é mais custoso em relação a primeira parte da atividade). Para isso, utilizamos os códigos bin2bcd, binAdder e bcd2ssd, além de um código que declara os componentes utilizados, chamado project1.

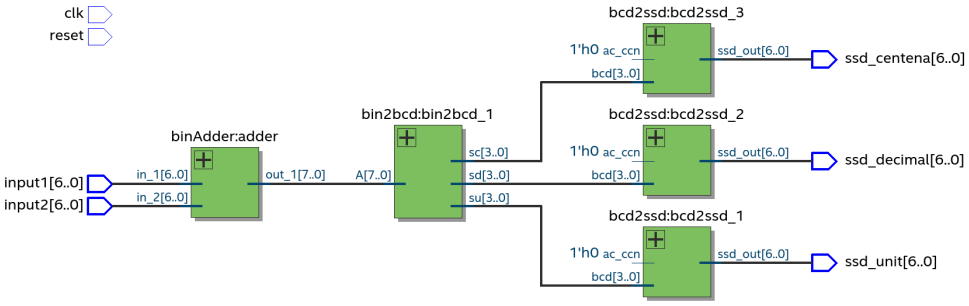

Figura 3: RTL - Circuito da Segunda Parte
Figura elaborada pelo autor

Como podemos notar, nesta segunda implementação, os dois vetores de bits são diretamente somados e em seguida convertidos para BCD e posteriormente para SSD, para isso, os seguintes códigos (representados por componentes) foram utilizados:

O código bin2bcd é responsável por converter um número binário de 8 bits para BCD, dividindo o número em centenas, dezenas e unidades.

O código binAdder é responsável por somar dois números binários de 8 bits, e o código bcd2ssd é responsável por converter um número BCD para um display de 7 segmentos.

Por fim, o código “project1” declara os componentes utilizados e realiza a conexão entre eles.

Para verificarmos seu funcionamento, realizamos um testbench para verificar se a implementação está correta. Abaixo está a simulação do testbench:

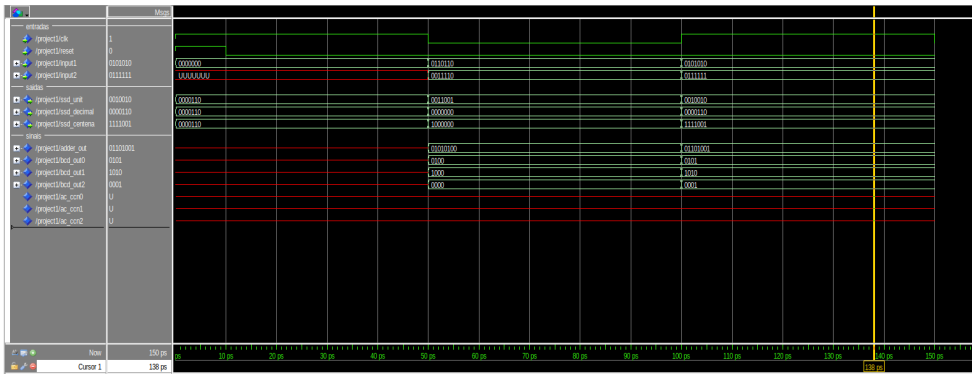

Figura 4: Simulação RTL - Segunda parte
Figura elaborada pelo autor

Podemos notar que a implementação 2 é mais complexa que a implementação 1, pois a implementação 2 realiza a soma em binário e em seguida converte para BCD e SSD, enquanto que a implementação 1 realiza a soma diretamente em BCD e converte para SSD.

4. Conclusão

A partir dos resultados obtidos, podemos concluir que a implementação 1 é mais rápida devido ao tempo de propagação amostrado em cada um dos casos. Também podemos concluir que a implementação Y é mais eficiente em termos de área, pois o consumo de área foi menor em relação à implementação X.

Implementacao	Área (LE)	Tempo de propagação (ns)
Parte 1	48	3.823
Parte 2	83	13.699

5. Códigos VHDL utilizados - Parte 2:

Para a segunda parte, utilizei outro arquivo bin2bcd.vhd, ao invés do arquivo provido pelo professor em aula, devido a dificuldades de coloca-lo em operação.

Como a implementação abaixo não necessita de registradores para operar, o calculo de área e tempo de propagação foi feito sem necessidade de alterações, garantindo confiabilidade no resultado obtido.

5.1. bin2bcd

```
1  library ieee;
```

```
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity bin2bcd is
    port (
        A      : in  std_logic_vector (7 downto 0);
        sd, su, sc : out std_logic_vector (3 downto 0)
    );
end entity;

architecture ifsc_v1 of bin2bcd is
    signal A_uns          : unsigned (7 downto 0);
    signal sd_uns, su_uns, sc_uns : unsigned (7 downto 0);

begin
    A_uns  <= unsigned(A);
   sc_uns <= A_uns/100;
    sd_uns <= A_uns/10;
    su_uns <= A_uns rem 10;
    sc     <= std_logic_vector(resize(sc_uns, 4));
    sd     <= std_logic_vector(resize(sd_uns, 4));
    su     <= std_logic_vector(resize(su_uns, 4));
end architecture;
```

O código binAdder é reponsavel por somar dois números binários de 7 (128 represetações possiveis, e portanto atendendo a especificação) bits e retornar o resultado em binário com 8 bits.

5.2. binAdder

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity binAdder is
    generic(
    Nlength: integer := 7
    );
    port(

   in_1: in std_logic_vector(Nlength - 1 downto 0);
   in_2: in std_logic_vector(Nlength - 1 downto 0);
   out_1: out std_logic_vector(Nlength downto 0)

    );
end binAdder;

architecture v1 of binAdder is

   signal bin1, bin2: unsigned(Nlength downto 0);
   signal out_bin: unsigned(Nlength downto 0);

begin

    bin1 <= resize(unsigned(in_1),Nlength + 1);
```

```
    bin2 <= resize(unsigned(in_2),Nlength + 1);

    out_bin <= bin1 + bin2;
    out_1 <= std_logic_vector(out_bin);

end v1;
```

5.3. bcd2ssd:

O código bcd2ssd é responsável por converter um número BCD de 4 bits para um display de 7 segmentos.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity bcd2ssd is
    port (
        bcd     : in std_logic_vector(3 downto 0);
        ssd_out : out std_logic_vector(6 downto 0);
        ac_ccn  : in std_logic
    );
end entity bcd2ssd;

architecture bcd2ssd_v1 of bcd2ssd is

    signal ssd : std_logic_vector(6 downto 0);
    signal bcd_int : integer range 0 to 9;

begin

    ssd_out <= ssd when ac_ccn = '1' else
        not ssd;
    bcd_int <= to_integer(unsigned(bcd));

    with bcd_int select ssd <=
        "0111111" when 0,
        "0000110" when 1,
        "1011011" when 2,
        "1001111" when 3,
        "1100110" when 4,
        "1101101" when 5,
        "1111101" when 6,
        "0000111" when 7,
        "1111111" when 8,
        "1101111" when 9,
        -- Character "E" when others:
        "1111001" when others;

end architecture bcd2ssd_v1;
```

5.4. Project-1 (declaração de componentes):

O código project1 declara os componentes utilizados e realiza a conexão entre eles.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity project1 is
port(
  clk        : in    std_logic;
  reset      : in    std_logic;

  input1     : in std_logic_vector(6 downto 0);
  input2     : in std_logic_vector(6 downto 0);

  ssd_unit    : out std_logic_vector(6 downto 0);
  ssd_decimal : out std_logic_vector(6 downto 0);
  ssd_centena : out std_logic_vector(6 downto 0)
);
end entity;

architecture ifsc of project1 is

  component div_clk is
    generic (
      div : natural := 50
    );
    port (
      clk_in  : in    std_logic;
      rst     : in    std_logic;
      clk_out : out   std_logic
    );
  end component div_clk;

  component bin2bcd is
    port (
        A      : in  std_logic_vector (7 downto 0);
        sd, su, sc : out std_logic_vector (3 downto 0)
    );
  end component bin2bcd ;

  component bcd2ssd is
    port (
      bcd     : in    std_logic_vector(3 downto 0);
      ssd_out : out   std_logic_vector(6 downto 0);
      ac_ccn  : in    std_logic
    );
  end component bcd2ssd;

  component binAdder is
    generic (
    Nlength : natural := 7
    );
    port(
   in_1: in std_logic_vector(Nlength - 1 downto 0);
   in_2: in std_logic_vector(Nlength - 1 downto 0);
   out_1: out std_logic_vector(Nlength downto 0)
    );
  end component binAdder;

```

```
  signal adder_out : std_logic_vector(7 downto 0);
  signal bcd_out0, bcd_out1, bcd_out2 : std_logic_vector(3 downto 0);
  signal ac_ccn0, ac_ccn1, ac_ccn2 : std_logic;

begin

 adder : component binAdder
   generic map(
   Nlength => 7
   )
    port map (
      in_1  => input1,
      in_2  => input2,
      out_1     => adder_out
    );

 bin2bcd_1 : component bin2bcd
    port map (
      A => adder_out,
    su => bcd_out0,
    sd => bcd_out1,
    sc => bcd_out2
    );

  bcd2ssd_1 : component bcd2ssd
  port map (
    bcd     => bcd_out0,
    ssd_out => ssd_unit,
   ac_ccn => ac_ccn0
  );

  bcd2ssd_2 : component bcd2ssd
  port map (
    bcd     => bcd_out1,
    ssd_out => ssd_decimal,
   ac_ccn => ac_ccn1
  );

  bcd2ssd_3 : component bcd2ssd
  port map (
    bcd     => bcd_out2,
    ssd_out => ssd_centena,
   ac_ccn => ac_ccn2
  );


end architecture;
```