Conversor Binário
Complemento 2 - Excesso-K

1.0 - PROBLEMA:

Criar um circuito combinacional capaz de converter uma palavra de 5 bits em Complemento de 2 para Excesso-K, e que possua um MUX e um DEMUX para poder escolher a base binária da saída. Para números em Complemento de 2 para o qual não existem equivalentes em Excesso-K, deverá ser ativada uma saída de ERRO. O circuito combinacional deverá ser implementado em VHDL.

Componentes:

- Conversor;
- Mux 2x6;
- Demux 2x6;
- Saída erro;
- No VHDL, haverão diversos sinais para conectar os diversos elementos do circuito.

2.0 - CIRCUITO COMBINACIONAL:

2.1 - CONVERSOR (Complemento de 2 → Excesso-K)

1) Comportamento:

Entradas: Palavra de 6 bits, {c5, c4, c3, c2, c1, c0}, escrita em Complemento de 2. O bit mais significativo (c5) equivale ao fio de erro, e sempre entra como '0'.

Saídas: Palavra de 6 bits, {s5, s4, s3, s2, s1, s0}, convertida para Excesso-K. Caso a conversão for inválida, o bit mais significativo (s5) será 1.

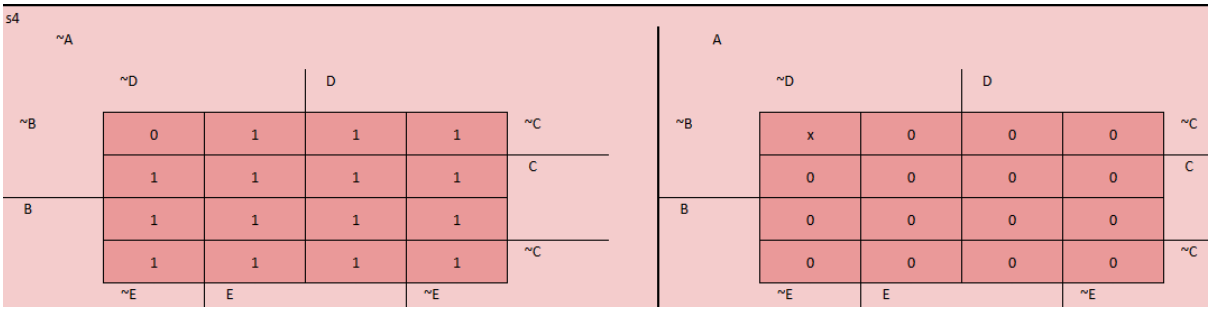
2) Tabela-Verdade:

Caso	c5 erro	c4	c3	c2	c1	c0	VALOR	ERRO	s4	s3	s2	s1	s0
1	0	0	0	0	0	0	0	0	0	1	1	1	1
2	0	0	0	0	0	1	1	0	1	0	0	0	0
3	0	0	0	0	1	0	2	0	1	0	0	0	1
4	0	0	0	0	1	1	3	0	1	0	0	1	0
5	0	0	0	1	0	0	4	0	1	0	0	1	1
6	0	0	0	1	0	1	5	0	1	0	1	0	0
7	0	0	0	1	1	0	6	0	1	0	1	0	1
8	0	0	0	1	1	1	7	0	1	0	1	1	0
9	0	0	1	0	0	0	8	0	1	0	1	1	1
10	0	0	1	0	0	1	9	0	1	1	0	0	0
11	0	0	1	0	1	0	10	0	1	1	0	0	1
12	0	0	1	0	1	1	11	0	1	1	0	1	0
13	0	0	1	1	0	0	12	0	1	1	0	1	1
14	0	0	1	1	0	1	13	0	1	1	1	0	0
15	0	0	1	1	1	0	14	0	1	1	1	0	1
16	0	0	1	1	1	1	15	0	1	1	1	1	0

Caso	c5 erro	c4	c3	c2	c1	c0	VALOR	ERRO	s4	s3	s2	s1	s0
17	0	1	0	0	0	0	ERRO-16	1	x	x	x	x	x
18	0	1	0	0	0	1	-15	0	0	0	0	0	0
19	0	1	0	0	1	0	-14	0	0	0	0	0	1
20	0	1	0	0	1	1	-13	0	0	0	0	1	0
21	0	1	0	1	0	0	-12	0	0	0	0	1	1
22	0	1	0	1	0	1	-11	0	0	0	1	0	0
23	0	1	0	1	1	0	-10	0	0	0	1	0	1
24	0	1	0	1	1	1	-9	0	0	0	1	1	0
25	0	1	1	0	0	0	-8	0	0	0	1	1	1
26	0	1	1	0	0	1	-7	0	0	1	0	0	0
27	0	1	1	0	1	0	-6	0	0	1	0	0	1
28	0	1	1	0	1	1	-5	0	0	1	0	1	0
29	0	1	1	1	0	0	-4	0	0	1	0	1	1
30	0	1	1	1	0	1	-3	0	0	1	1	0	0
31	0	1	1	1	1	0	-2	0	0	1	1	0	1
32	0	1	1	1	1	1	-1	0	0	1	1	1	0

Diagramas de Veitch Karnaugh + Circuitos Lógicos:

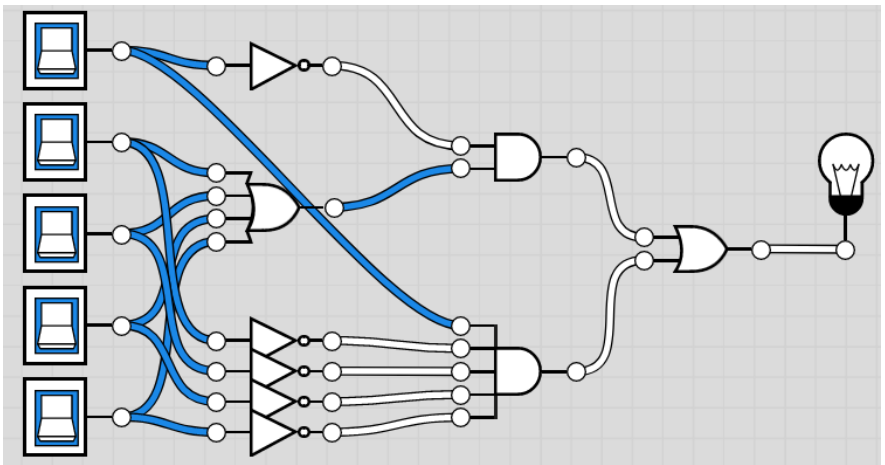
s4:



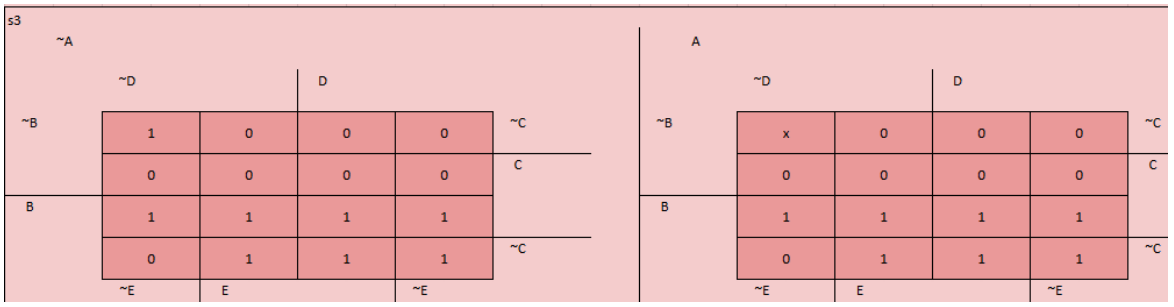
s4 <= (~c4 * c0) + (~c4 * c1) + (~c4* c2) + (~c4 * c3) + (c4 * ~c3 * ~c2 * ~c1 * ~c0)

ou

s4 <= (~c4 * (c0 + c1 + c2 + c3)) + (c4 * ~c3 * ~c2 * ~c1 * ~c0)



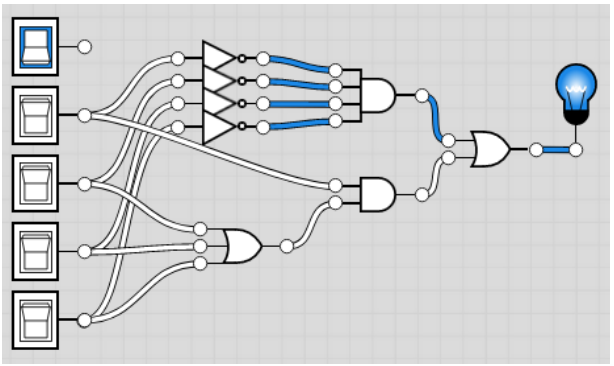
s3:



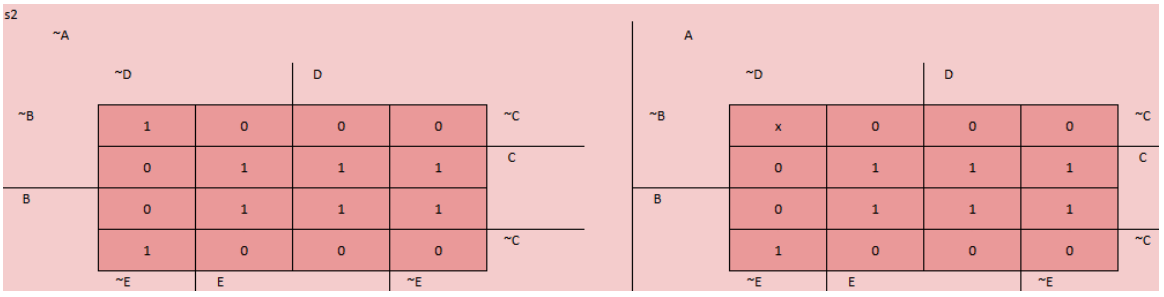
s3 <= (c3 * c2) + (c3 * c1) + (c3 * c0) + (~c3 * ~c2 * ~c1 * ~c0)

ou

s3 <= [c3 * (c2 + c1 + c0)] + (~c3 * ~c2 * ~c1 * ~c0)



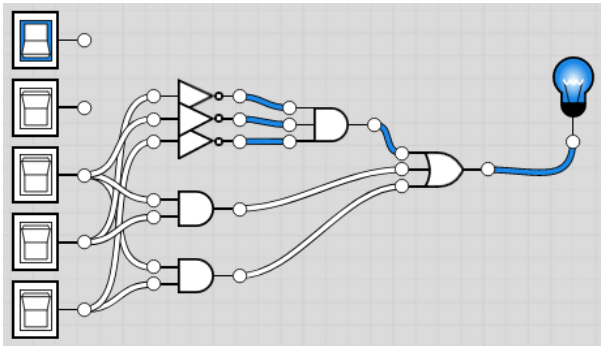
s2:



s2 <= (c2 * c0) + (c2 * c1) + (~c0 * ~c1 * ~c2)

ou

s2 <= [c2 * (c0 + c1)] + (~c0 * ~c1 * ~c2)



s1:

s1

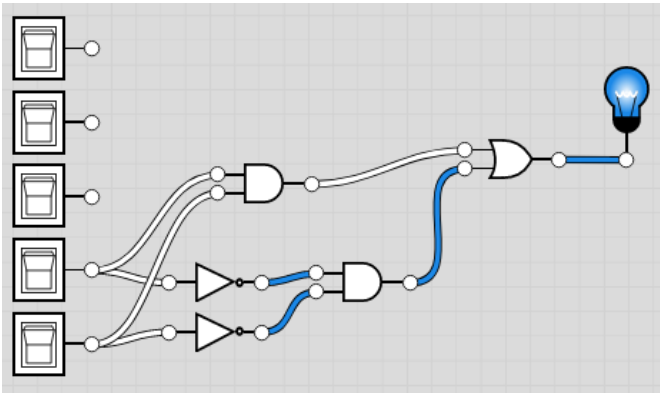
	~A ~D	~A D	A ~D	A D
~B ~C	1	0	1	0
~B C	1	0	1	0
B ~C	1	0	1	0
B ~C	1	0	1	0
	~E	E	~E	

	A ~D	A D		
~B ~C	x	0	1	0
~B C	1	0	1	0
B ~C	1	0	1	0
B ~C	1	0	1	0
	~E	E	~E	

s1 <= (~c1 * ~c0) + (c1 * c0)

ou

s1 <= c1 XNOR c0



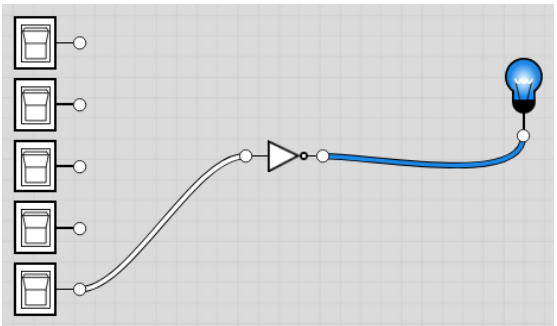
s0:

s0

	~A ~D	~A D	A ~D	A D
~B ~C	1	0	0	1
~B C	1	0	0	1
B ~C	1	0	0	1
B ~C	1	0	0	1
	~E	E	~E	

	A ~D	A D		
~B ~C	x	0	0	1
~B C	1	0	0	1
B ~C	1	0	0	1
B ~C	1	0	0	1
	~E	E	~E	

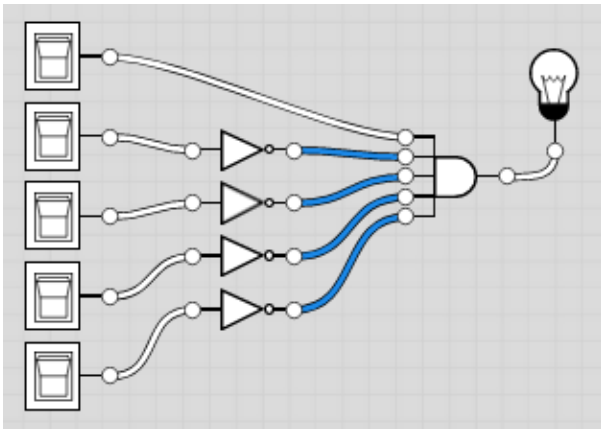
s0 <= ~c0



s5 (ERRO):

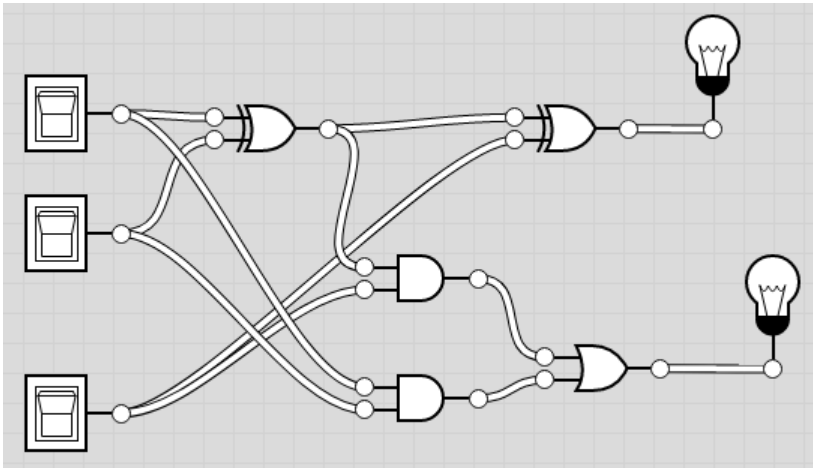
Para definir o circuito da saída erro, não foi necessária a criação de um DVK, pois ela só é ativada em um único caso específico. Sendo assim, juntou-se c4 com as negadas de c3, c2, c1, e c0 sob uma porta and:

s5 <= c4 * ~c3 * ~c2 * ~c1 * ~c0



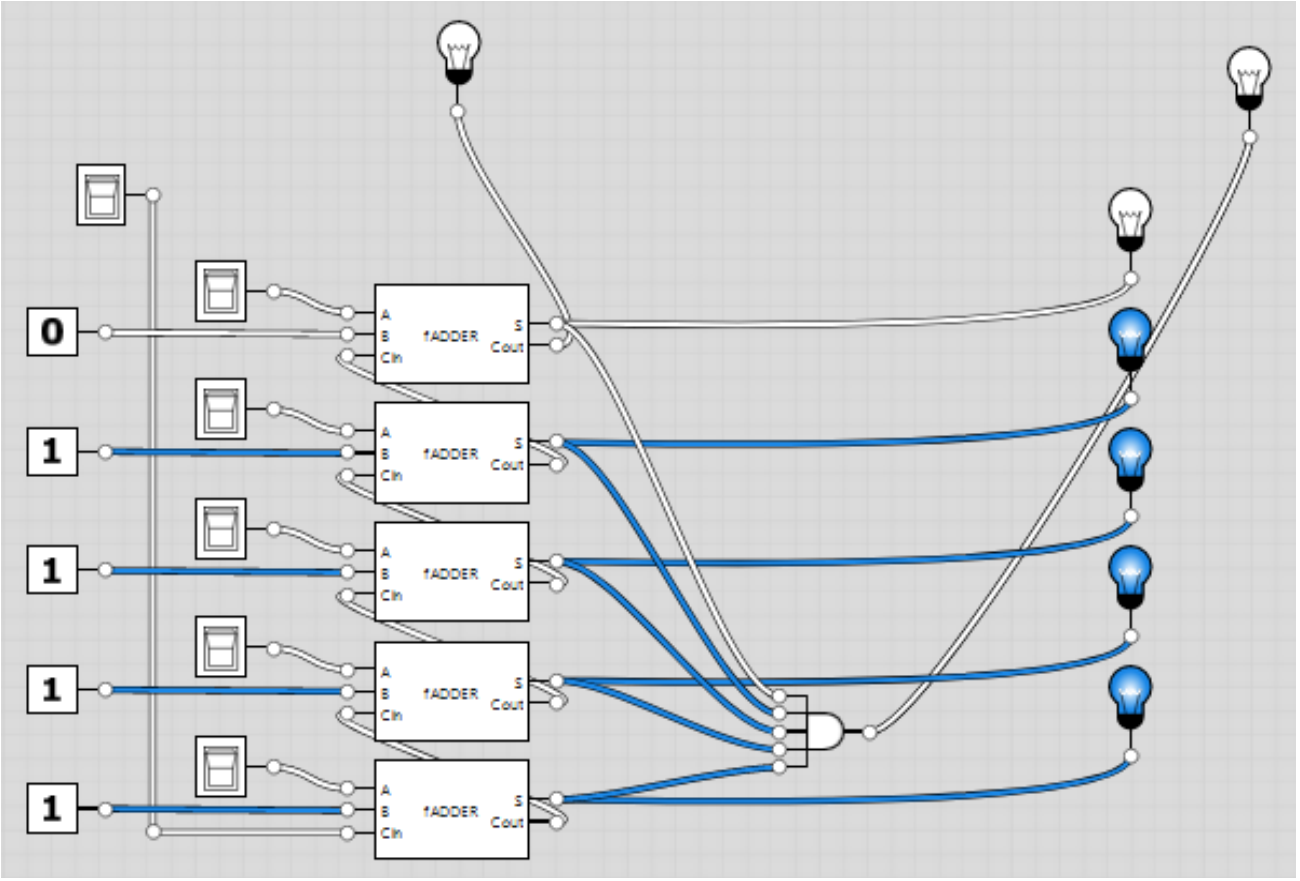
2.2 - Método Alternativo: Conversão Por Soma (Complemento de 2 → Excesso-K):

Ao se observar a tabela verdade, percebe-se que o vetor de saída é igual à soma do vetor de entrada com “0 1 1 1 1”. Sendo assim, um método alternativo de conversão se dá pela soma do vetor de entrada com uma constante 0 1 1 1 1. Para tal, utilizam-se 5 blocos full adder:



(Na imagem acima, vemos o circuito interno de um bloco full adder [fADDER])

A saída erro é ativada quando o resultado da soma é ‘1 1 1 1 1’ . Ela é representada pelo único bloco ‘*and*’ na imagem, que ativa a lâmpada isolada à direita:



2.3 - Multiplexador (MUX 2x6):

Descrição:

Entradas:

Vetor ‘C2’ (6 fios);

Vetor ‘EK’ (6 fios);

Limite do $\log_2 = 2 \rightarrow$ 1 fio seletor;

Saída:

Vetor ‘Z’ de 6 fios, idêntico ao escolhido pelo seletor;

Funcionamento:

‘Z’ <= ‘C2’ quando seletor = ‘0’;

‘Z’ <= ‘EK’ quando seletor = ‘1’;

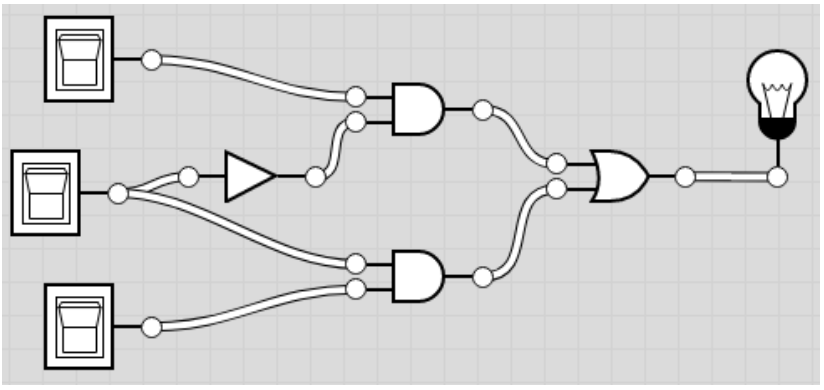
Tabela Verdade Compacta:

T.V.C.	Select0	Z
Sit 0	0	Vetor C2
Sit 2	1	Vetor EK

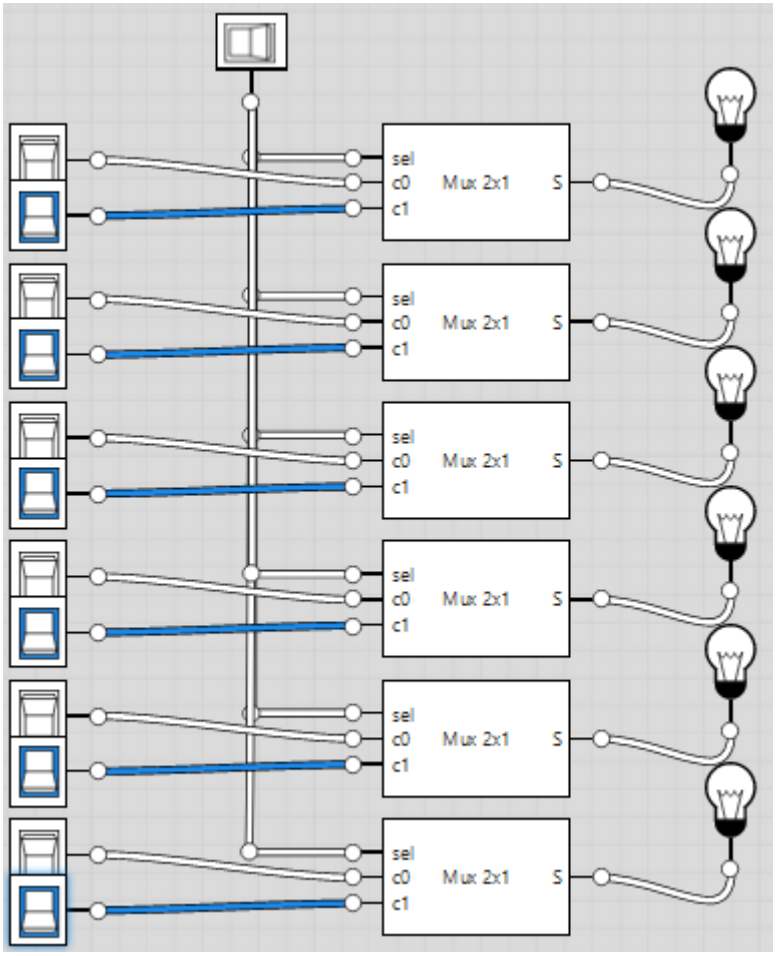
Expressão booleana:

(~Select0 * Vetor ’C2’) + (Select0 * Vetor ‘EK’)

Circuito:



Sendo assim, ao combinarmos 6 desses circuitos, teremos um MUX 2x6:



2.4 - Demultiplexador (DEMUX 2x6):

Descrição:

Entradas:

Vetor ‘Z’ de 6 fios;

Limite do $\log_2 = 2 \rightarrow$ 1 fio seletor;

Saída:

Vetor ‘C2’ (6 fios);

Vetor ‘EK’ (6 fios);

Limite do $\log_2 = 2 \rightarrow$ 1 fio seletor;

Funcionamento:

‘C2’ <= ‘Z’ quando seletor = ‘0’;

‘EK’ <= ‘Z’ quando seletor = ‘1’;

Tabela Verdade Simplificada:

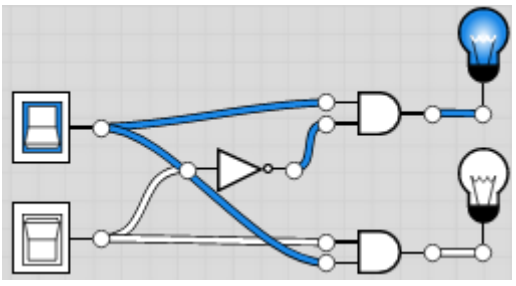
T.V.C.	Select0	Z
Sit 0	0	Vetor C2 <= Z
Sit 2	1	Vetor EK <= Z

Expressão booleana:

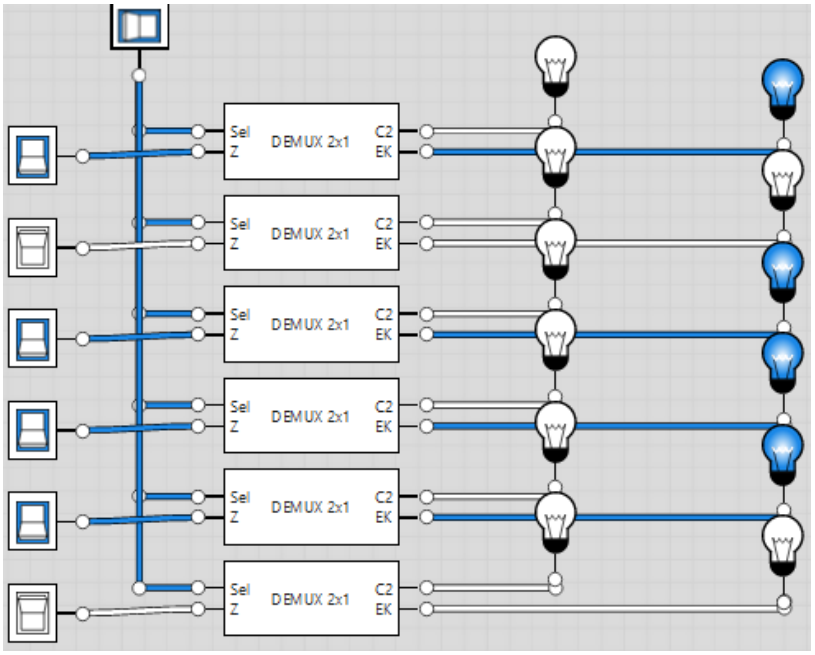
‘C2’ <= (~Sel * Z)

‘EK‘ <= (Sel * Z)

Circuito:



Juntando 6 desses, teremos um DEMUX 2x6:



3.0 - VHDL:

O código em VHDL foi feito pelo sistema operacional Windows 10, escrito utilizando o software Visual Studio Code, com as extensões “TerosHDL” e “VHDL”, e usando a biblioteca “*ieee*”. Os códigos em foram convertidos num arquivo .ghw com os mesmos comandos que se usariam no terminal do Linux.

Mux2x6 e Demux2x6:

O multiplexador foi feito de acordo com o apresentado nas aulas práticas sobre, com a exceção de que as entradas foram substituídas por variáveis do tipo ‘*std_logic_vector(5 downto 0)*’, pertencentes à biblioteca “*ieee*”. A seguir, uma print do código do multiplexador:

```
1   library ieee;
2   use ieee.std_logic_1164.all;
3
4   entity muxf2x6 is
5     port (
6       C2_v  : in std_logic_vector (5 downto 0);
7       EK_v  : in std_logic_vector (5 downto 0);
8       selm  : in bit;
9       MUX_s : out std_logic_vector (5 downto 0)
10    );
11  end entity;
12
13  architecture muxagem of muxf2x6 is
14  begin
15
16    MUX_s <= C2_v when selm = '0' else EK_v;
17
18  end architecture;
```

O multiplexador foi feito de maneira um pouco diferente. A entrada dele possui 6 bits, ao passo que a sua saída possui apenas 5. Isso se deve ao fato de que o bit c5 (bit mais significativo) é removido do vetor para entrar na saída erro, que está embutida no código do demux. Além disso, a saída do demux que não for selecionada vai receber um valor fixo de “00000”, em vez de não receber sinal algum. Isso foi feito por que assim fica mais bonito no GTKwave.

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

entity demuxf2x6 is
  port (
    C2_v2  : out std_logic_vector (4 downto 0);
    EK_v2  : out std_logic_vector (4 downto 0);
    MUX_e  : in  std_logic_vector (5 downto 0);
    seld   : in  bit;
    ERRO   : out std_logic

  );
end entity;

architecture demuxagem of demuxf2x6 is
begin

  C2_v2 <= MUX_e(4 downto 0) when seld = '0' else "00000";
  EK_v2 <= MUX_e(4 downto 0) when seld = '1' else "00000";
  ERRO  <= '1' when MUX_e(5) = '1' else '0';

end architecture;
```

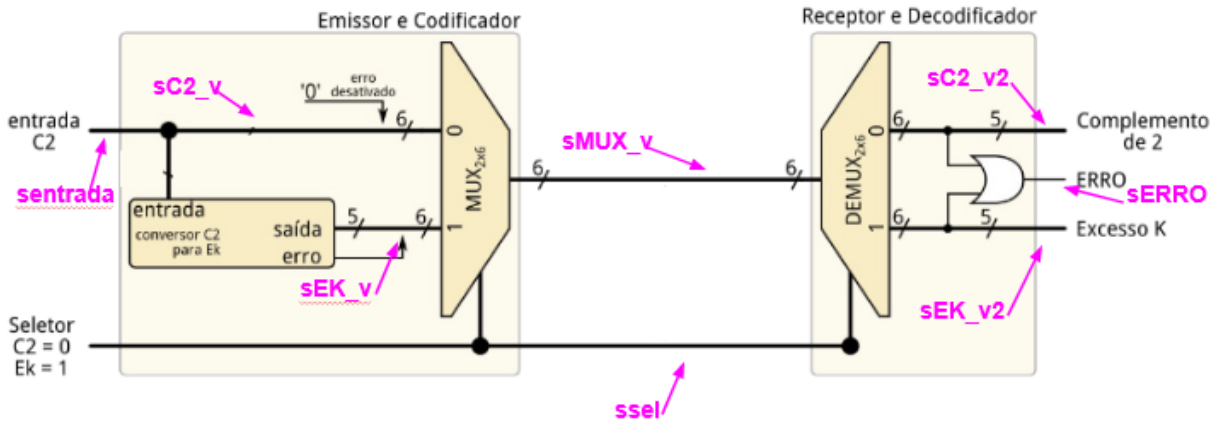
Conversor:

O conversor foi feito aplicando-se as expressões booleanas criadas anteriormente.

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

entity conversorf is
    port(
        entrada : in  std_logic_vector(5 downto 0);
        C2_v0   : out std_logic_vector(5 downto 0);
        EK_v0   : out std_logic_vector(5 downto 0)
    );
end entity;

architecture convertidor of conversorf is
begin
    C2_v0(5 downto 0) <= entrada(5 downto 0);

    EK_v0(5) <= (entrada(4) and not(entrada(3)) and not(entrada(2)) and not(entrada(1)) and not(entrada(0)));
    EK_v0(4) <= (not(entrada(4)) and (entrada(0) or entrada(1) or entrada(2) or entrada(3))) or (entrada(4) and not(entrada(3)) and not(entrada
    EK_v0(3) <= (entrada(3) and (entrada(2) or entrada(1) or entrada(0))) or (not(entrada(3)) and not(entrada(2)) and not(entrada(1)) and not(e
    EK_v0(2) <= (entrada(2) and (entrada(0) or entrada(1))) or (not(entrada(0)) and not(entrada(1)) and not(entrada(2)));
    EK_v0(1) <= entrada(1) xnor entrada(0);
    EK_v0(0) <= not(entrada(0));

end architecture;
```

Testbench:

Por fim, o *testbench* foi feito utilizando 8 sinais. Os nomes dos sinais e seus fios físicos equivalentes estão representados na figura abaixo:



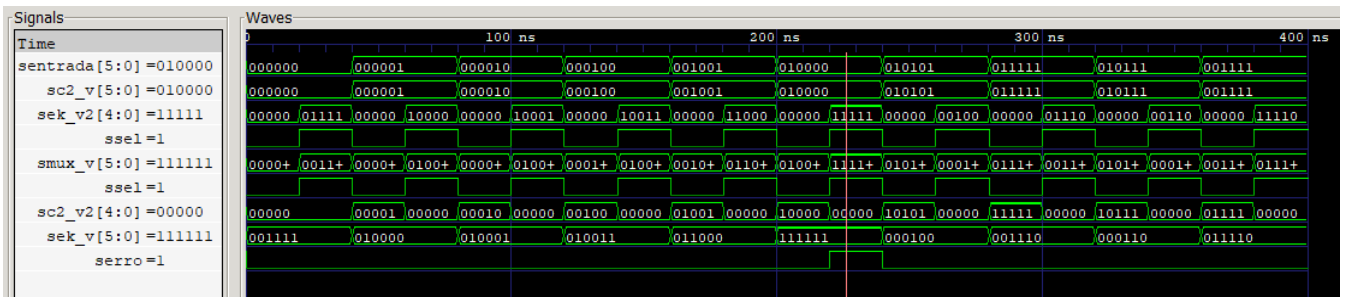
Os sinais feitos e o portmap dos componentes estão representado abaixo:

```
37      signal sentrada: std_logic_vector (5 downto 0);
38      signal sC2_v   : std_logic_vector (5 downto 0);
39      signal sEK_v   : std_logic_vector (5 downto 0);
40      signal sC2_v2  : std_logic_vector (4 downto 0);
41      signal sEK_v2  : std_logic_vector (4 downto 0);
42      signal sMUX_v  : std_logic_vector (5 downto 0);
43      signal sERRO   : std_logic;
44      signal ssel    : bit;
45
46  begin
47      u_conversor : conversorf port map (sentrada, sC2_v, sEK_v);
48      u_mux2x6    : muxf2x6    port map (sC2_v,  sEK_v,  ssel, sMUX_v);
49      u_demux2x6  : demuxf2x6  port map (sC2_v2, sEK_v2, sMUX_v, ssel, sERRO);
```

Por fim, o testbench é finalizado com um processo que define diversos valores randômicos para a entrada e para o fio seletor. Estes servem para testar o circuito no GTKwave.

3.0 - GTKwave:

Quando o *testbench* é executado no GTKwave, o diagrama de ondas resultante é o seguinte:



Após a análise dos resultados, percebe-se que o conversor funciona conforme o esperado, e que a saída de erro recebe um valor de ‘1’ apenas quando o sinal de entrada equivale a “100000”.