Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ingeniería Eléctrica

IEE2713 — Sistemas Digitales – 2019-2

Pauta Tarea 5

1. Necesitamos 11 bits para obtener el complemento de dos para un número de 10 bits, ya que hay que agregar el bit de signo. Para obtenerlo invertimos todos los bits y sumamos uno al número final.

   a) Convertimos de 10 a 11 bits:

   $$01\ 1101\ 1110 \rightarrow 001\ 1101\ 1110$$

   Invertimos cada bit y sumamos uno:

   $$110\ 0010\ 0001 + 01 = 110\ 0010\ 0010$$

   b) Mismo procedimiento:

   $$10\ 1110\ 0111 \rightarrow 010\ 1110\ 0111$$
   $$101\ 0001\ 1000 + 01 = 101\ 0001\ 1001$$

   c) Repetimos:

   $$11\ 1111\ 1110 \rightarrow 011\ 1111\ 1110$$
   $$100\ 0000\ 0001 + 01 = 100\ 0000\ 0010$$

2. En primer lugar, un número positivo en complemento de 2 es el mismo número. Para los números negativos primero los escribimos con 12 bits sin considerar el signo y después los transformamos a complemento de 2 de la misma forma que en el punto anterior.

   a) $73 \rightarrow 100\ 1001 \rightarrow 0000\ 0100\ 1001$

   b) $1906 \rightarrow 111\ 0111\ 0010 \rightarrow 0111\ 0111\ 0010$

   c) $-95 \rightarrow 0000\ 0101\ 1111 \rightarrow 1111\ 1010\ 0001$

   d) $-1630 \rightarrow 0110\ 0101\ 1110 \rightarrow 1001\ 1010\ 0010$

3. El decodificador genera una salida diferente por cada miniterm requerido por la función. Estas salidas se combinan en la compuerta OR. El resultado final es el siguiente:

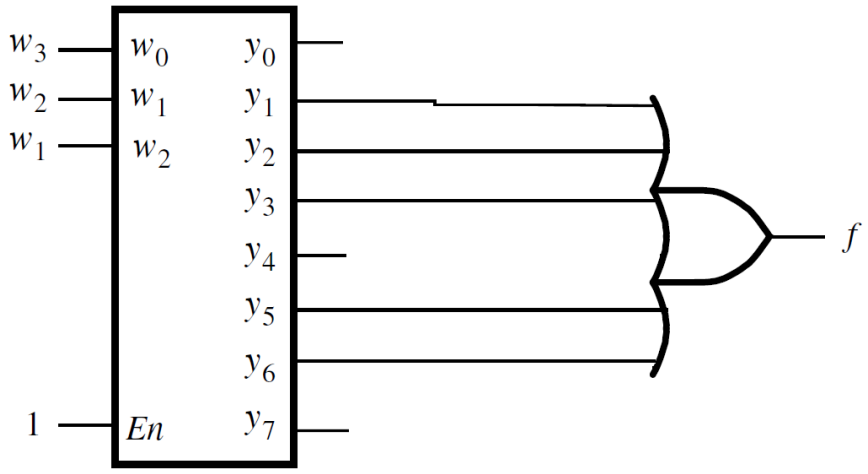


Figura 1: Decoder con compuerta OR para problema 3.

4. Primero, decidimos arbitrariamente usar w_1 como selector del multiplexor. Luego, realizamos el siguiente procedimiento, donde creamos un circuito para cada entrada del multiplexor en base al valor de w_1. Esto se puede ver en las siguientes imágenes:

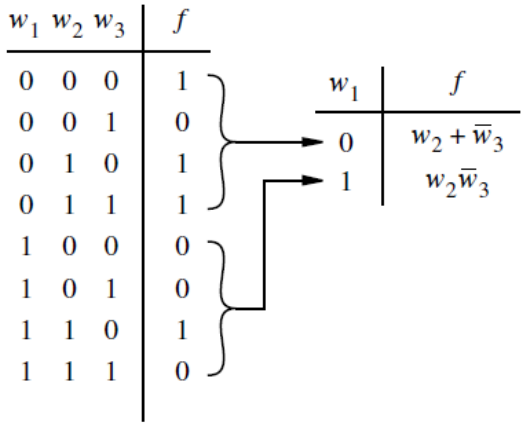


Figura 2: Proceso para implementación con multiplexor.

5. Vamos a elegir nuevamente a w_1 para realizar la expansión de Shannon. El procedimiento es como sigue:

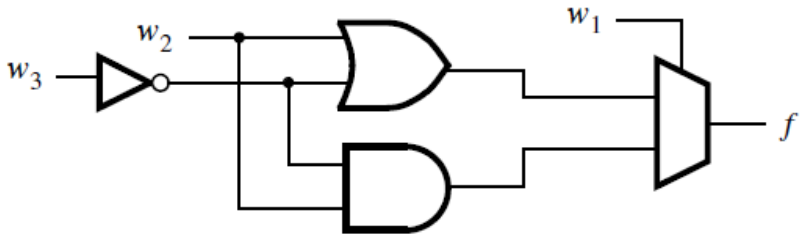


Figura 3: Circuito final para problema 4.

$$f = \bar{w_1}\bar{w_3} + w_2\bar{w_3} + \bar{w_1}w_2$$
$$f = w_1 f_{w_1} + \bar{w_1} f_{\bar{w_1}}$$

Ahora tenemos a w_1 separados de tal forma que funciona como un multiplexor, teniendo un circuito f_{w_1} para cuando $w_1 = 1$ y otro para el contrario.

Para determinar $f_{\bar{w_1}}$, nos fijamos en los minitérminos que contienen a $\bar{w_1}$, luego, como los juntamos en función de éste, similar a simplificar la función pero de manera conveniente. Para f_{w_1} usamos el término que no contiene a w_1 ya que no influye en la decisión. Si hubieran términos con w_1 debemos hacer el mismo procedimiento anterior.

Finalmente obtenemos:

$$f = w_1(w_2 + \bar{w_3}) + \bar{w_1}(w_2\bar{w_3})$$

El circuito es idéntico al del problema 4 pero obtenido de manera diferente.

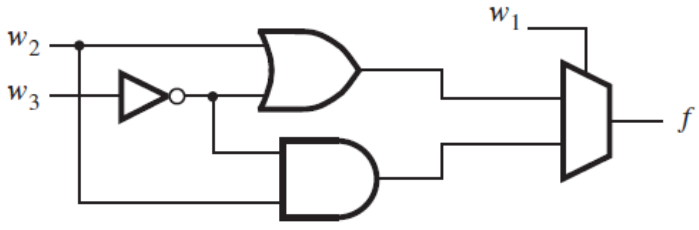


Figura 4: Circuito final para problema 5.

6. Módulo PGen:

Listing 1: PGen.v

```
//------------------------------------------------------
// Modulo    : PGen
// Archivo   : PGen.v
// Function  : Generar bits de paridad
//------------------------------------------------------

module PGen(input  m1, m2, m3, m4,
            output p1, p2, p3);

  // P = p1p2p3
  assign p1 = m1 ^ m2 ^ m4;
  assign p2 = m1 ^ m3 ^ m4;
  assign p3 = m2 ^ m3 ^ m4;

endmodule
```

7. Módulo CGen:

Listing 2: CGen.v

```
//------------------------------------------------------
// Modulo    : CGen
// Archivo   : CGen.v
// Function  : Generar bits de chequeo
//------------------------------------------------------

module CGen(input  p1, p2, m1, p3, m2, m3, m4,
            output c1, c2, c3);

  // T = p1p2m1p3m2m3m4
  assign c1 = p1 ^ m1 ^ m2 ^ m4;
  assign c2 = p2 ^ m1 ^ m3 ^ m4;
  assign c3 = p3 ^ m2 ^ m3 ^ m4;

endmodule
```

8. Módulo Corr:

Listing 3: Corr.v

```
//----------------------------------------------------
// Modulo   : Corr
// Archivo  : Corr.v
// Function : Correccion de error
//----------------------------------------------------

module Corr(input  px1, px2, mx1, px3, mx2, mx3, mx4,
            input  c1, c2, c3,
            output pc1, pc2, mc1, pc3, mc2, mc3, mc4);

  wire [6:0] Terr;
  reg  [6:0] Tc; // Tipo reg por estar dentro de un always
  wire [2:0] C;

  // Concatenacion de bits:
  assign Terr = {px1, px2, mx1, px3, mx2, mx3, mx4};
  assign C = {c3, c2, c1};

  // Correccion del bit erroneo:
  always @(*) begin
    case(C)
      3'b000: Tc = Terr;
      3'b001: Tc = {Terr[6:1], ~Terr[0]};
      3'b010: Tc = {Terr[6:2], ~Terr[1], Terr[0]};
      3'b011: Tc = {Terr[6:3], ~Terr[2], Terr[1:0]};
      3'b100: Tc = {Terr[6:4], ~Terr[3], Terr[2:0]};
      3'b101: Tc = {Terr[6:5], ~Terr[4], Terr[3:0]};
      3'b101: Tc = {Terr[6], ~Terr[5], Terr[4:0]};
      3'b110: Tc = {~Terr[6], Terr[5:0]};
      default: Tc = Terr;
    endcase
  end

  // Asignacion final
  assign {pc1, pc2, mc1, pc3, mc2, mc3, mc4} = Tc;

endmodule
```

9. Módulo EmuCom:

Listing 4: EmuCom.v

```
//------------------------------------------------------
// Modulo    : EmuCom
// Archivo   : EmuCom.v
// Function  : Emulador de comunicacion con error
//------------------------------------------------------

module EmuCom(input  p1, p2, m1, p3, m2, m3, m4,
              input  x1, x2, x3, x4, x5, x6, x7,
              output px1, px2, mx1, px3, mx2, mx3, mx4);

  // Si x = 1, entonces se recibe el bit respectivo equivocado
  assign px1 = (x1) ? ~p1 : p1;
  assign px2 = (x2) ? ~p2 : p2;
  assign mx1 = (x3) ? ~m1 : m1;
  assign px3 = (x4) ? ~p3 : p3;
  assign mx2 = (x5) ? ~m2 : m2;
  assign mx3 = (x6) ? ~m3 : m3;
  assign mx4 = (x7) ? ~m4 : m4;

endmodule
```

10. Módulo final:

Listing 5: Basys.v

```
//------------------------------------------------------
// Modulo    : Basys
// Archivo   : Basys.v
// Function  : Codigo Hamming
//------------------------------------------------------

module Basys(input  wire [11:0] sw,
             output wire [10:4] led,
             output wire [3:0]  an,
             output reg  [6:0]  seg);

  wire p1, p2, p3; // P
  wire px1, px2, mx1, px3, mx2, mx3, mx4; // Terr
  wire c1, c2, c3; // C
  wire pc1, pc2, mc1, pc3, mc2, mc3, mc4; // Tc
  wire [3:0] M; // Mensaje que va al 7-seg

  PGen pgen1(sw[0], sw[1], sw[2], sw[3], // M
```

```
19              p1, p2, p3); // P
20
21  EmuCom emucom1(p1, p2, sw[0], p3, sw[1], sw[2], sw[3], // T
22                 sw[4], sw[5], sw[6], sw[7], sw[8], sw[9], sw[
                      10], // X
23                 px1, px2, mx1, px3, mx2, mx3, mx4); // Terr
24
25  CGen cgen1(px1, px2, mx1, px3, mx2, mx3, mx4, // Terr
26             c1, c2, c3); // C
27
28  Corr corr1(px1, px2, mx1, px3, mx2, mx3, mx4, // Terr
29             c1, c2, c3, // C
30             pc1, pc2, mc1, pc3, mc2, mc3, mc4); // Tc
31
32  // Prender LED segun la posicion del bit erroneo
33  // Usamos un decoder simple para la asignacion
34  assign led[10:4] = (1 << {c3, c2, c1});
35
36  // Seleccionamos que mensaje pasa al 7-seg con un multiplexor
37  // Si sw[11] = 1, entonces pasa el mensaje corregido
38  // Si sw[11] = 0, entonces pasa el mensaje original
39  assign M = (sw[11]) ? {mc1, mc2, mc3, mc4} : {mx1, mx2, mx3,
       mx4};
40
41  // Display 7-seg
42  always @(*) begin
43    case(M)
44   // Segmentos: 7'bABCDEFG
45      15: seg <= 7'b0111000;
46      14: seg <= 7'b0110000;
47      13: seg <= 7'b1000010;
48      12: seg <= 7'b0110001;
49      11: seg <= 7'b1100000;
50      10: seg <= 7'b0001000;
51      9:  seg <= 7'b0000100;
52      8:  seg <= 7'b0000000;
53      7:  seg <= 7'b0001111;
54      6:  seg <= 7'b0100000;
55      5:  seg <= 7'b0100100;
56      4:  seg <= 7'b1001100;
57      3:  seg <= 7'b0000110;
58      2:  seg <= 7'b0010010;
59      1:  seg <= 7'b1001111;
60      0:  seg <= 7'b0000001;
61    endcase
```

```
62    end
63
64  endmodule
```