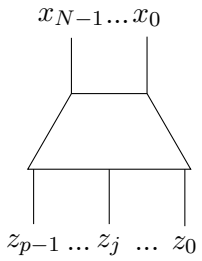
1 Lezione del 09-10-24

1.1 Decoder

Un decoder è una rete con N ingressi e p uscite con $p = 2^N$. Si indica come:



La sua legge di corrispondenza stabilisce che ogni uscita riconosce uno ed un solo stato di ingresso, in particolare l'uscita j-esima (z_j) riconosce lo stato di ingresso i cui bit sono la codifica di j in base 2, cioè:

$$(x_{n-1}, ..., x_0)_2 = j$$

Ad esempio, un decoder da 2 a 4 ha tabella di verità:

x_1	x_0	z_0	z_1	z_2	z_3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

che equivale alla codifica *one-hot* del binario in ingresso (cioè ogni numero codificato da n bit viene mandato al j-esimo di p output che corrispondono uno ad uno ai numeri rappresentabili).

Vediamo di passare da questa descrizione ad una sintesi della rete. Abbiamo che:

$$\begin{cases} z_3 = x_1 \cdot x_0 \\ z_2 = x_1 \cdot \overline{x}_0 \\ z_1 = \overline{x_1} \cdot x_0 \\ z_0 = \overline{x_1} \cdot \overline{x_0} \end{cases}$$

cioè ogni "indice" del decoder corrisponde al prodotto dei due ingressi opportunamente negati: l'ultima uscità avra tutti i bit attivi (sarebbe $2^N - 1$ considerando numeri naturali), ergo prende il prodotto di tutti gli ingressi. Di contro, la prima uscita (0) avrà tutti i bit disattivi, quindi prenderà il prodotto di tutti gli ingressi negati. Gli altri numeri vengono indirizzati prendendo il prodotto e complementando i bit che quel particolare numero si aspetterebbe come 0. Notiamo che, sebbene si abbiano 4 negazioni, nella rete fisica conviene negare gli input in entrata risparmiando 2 invertitori.

Per le figure, rimandiamo a `https://github.com/Guray00/IngegneriaInformatica/blob/master/SECONDO%20ANNO/I%20SEMESTRE/Reti%20Logiche/Diapositive%20OCR/Reti%20combinatorie%20ocr.pdf`.

Generalizziamo quindi questa struttura a decoder da N a 2^N, applicando quanto detto prima. Si avrà:

$$\begin{cases} z_0 = \overline{x_{N-1}} \cdot \overline{x_{N-2}} \cdot ... \cdot \overline{x_1} \cdot \overline{x_0} \\ z_1 = \overline{x_{N-1}} \cdot \overline{x_{N-2}} \cdot ... \cdot \overline{x_1} \cdot x_0 \\ ... \\ z_{p-2} = x_{N-1} \cdot x_{N-2} \cdot ... \cdot x_1 \cdot \overline{x_0} \\ z_{p-1} = x_{N-1} \cdot x_{N-2} \cdot ... \cdot x_1 \cdot x_0 \end{cases}$$

Vediamo quindi le codifiche in Verilog di decoder a diversi valori di N. Si definisce innanzitutto il caso banale di $N = 1$, che finora non è stato trattato. Questo servirà a definire, in maniera gerarchica (ma come vedremo imperfetta), decoder più complessi:

```
// un decoder da 1 a 2 che prende @x0 come ingresso e attiva l'uscita
// @z1_z0 in codifica one-hot
module b1to2_decoder(x0, z1_z0);
  input x0;
  output[1:0] z1_z0;

  assign z1_z0 = (x0 == 'B0) ? 'B01:
                 /*(x0 == 'B1)?*/'B10;
endmodule

// implementazione a porte logiche
module b1to2_decoder_p(x0, z1_z0);
  input x0;
  output[1:0] z1_z0;

  assign z1_z0[0] = ~x0;
  assign z1_z0[1] = x0;
endmodule
```

Possiamo quindi definire il decoder da 2 a 4 visto prima:

```
// un decoder da 2 a 4 che prende @x1_x0 come ingresso e attiva
// l'uscita @z4_z0 in codifica one-hot
module b2to4_decoder(x1_x0, z3_z0);
  input[1:0] x1_x0;
  output[3:0] z3_z0;

  assign z3_z0 = (x1_x0 == 'B00) ? 'B0001:
                 (x1_x0 == 'B01) ? 'B0010:
                 (x1_x0 == 'B10) ? 'B0100:
                 /*(x1_x0 == 'B11)?*/'B1000;
endmodule

// implementazione a porte logiche
module b2to4_decoder_p(x1_x0, z3_z0);
  input[1:0] x1_x0;
  output[3:0] z3_z0;

  assign z3_z0[0] = ~x1_x0[1] & ~x1_x0[0];
  assign z3_z0[1] = ~x1_x0[1] & x1_x0[0];
  assign z3_z0[2] = x1_x0[1] & ~x1_x0[0];
  assign z3_z0[3] = x1_x0[1] & x1_x0[0];
endmodule

// implementazione gerarchica
module b2to4_decoder_g(x1_x0, z3_z0);
  input[1:0] x1_x0;
```

```
  output[3:0] z3_z0;

  wire[1:0] x1_d;
  wire[1:0] x0_d;

  b1to2_decoder b1to2_1 (x1_x0[1], x1_d);
  b1to2_decoder b1to2_2 (x1_x0[0], x0_d);

  assign z3_z0[0] = x1_d[0] & x0_d[0];
  assign z3_z0[1] = x1_d[0] & x0_d[1];
  assign z3_z0[2] = x1_d[1] & x0_d[0];
  assign z3_z0[3] = x1_d[1] & x0_d[1];
endmodule
```

un decoder da 3 a 8:

```
// un decoder da 3 a 8 che prende @x2_x0 come ingresso e attiva
// l'uscita @z7_z0 in codifica one-hot
module b3to8_decoder(x2_x0, z7_z0);
  input[2:0] x2_x0;
  output[7:0] z7_z0;

  assign z7_z0 = (x2_x0 == 'B000) ? 'B0000_0001:
                 (x2_x0 == 'B001) ? 'B0000_0010:
                 (x2_x0 == 'B010) ? 'B0000_0100:
                 (x2_x0 == 'B011) ? 'B0000_1000:
                 (x2_x0 == 'B100) ? 'B0001_0000:
                 (x2_x0 == 'B101) ? 'B0010_0000:
                 (x2_x0 == 'B110) ? 'B0100_0000:
               /*(x2_x0 == 'B111)?*/'B1000_0000;
endmodule

// implementazione a porte logiche
module b3to8_decoder_p(x2_x0, z7_z0);
  input[2:0] x2_x0;
  output[7:0] z7_z0;

  assign z7_z0[0] = ~x2_x0[2] & ~x2_x0[1] & ~x2_x0[0];
  assign z7_z0[1] = ~x2_x0[2] & ~x2_x0[1] & x2_x0[0];
  assign z7_z0[2] = ~x2_x0[2] & x2_x0[1] & ~x2_x0[0];
  assign z7_z0[3] = ~x2_x0[2] & x2_x0[1] & x2_x0[0];
  assign z7_z0[4] = x2_x0[2] & ~x2_x0[1] & ~x2_x0[0];
  assign z7_z0[5] = x2_x0[2] & ~x2_x0[1] & x2_x0[0];
  assign z7_z0[6] = x2_x0[2] & x2_x0[1] & ~x2_x0[0];
  assign z7_z0[7] = x2_x0[2] & x2_x0[1] & x2_x0[0];
endmodule

// implementazione gerarchica
module b3to8_decoder_g(x2_x0, z7_z0);
  input[2:0] x2_x0;
  output[7:0] z7_z0;

  wire[1:0] x2_d;
  wire[3:0] x1_x0_d;

  b1to2_decoder b1to2 (x2_x0[2], x2_d);
  b2to4_decoder b2to4 (x2_x0[1:0], x1_x0_d);

  assign z7_z0[0] = x2_d[0] & x1_x0_d[0];
  assign z7_z0[1] = x2_d[0] & x1_x0_d[1];
  assign z7_z0[2] = x2_d[0] & x1_x0_d[2];
```

```
  assign z7_z0[3] = x2_d[0] & x1_x0_d[3];
  assign z7_z0[4] = x2_d[1] & x1_x0_d[0];
  assign z7_z0[5] = x2_d[1] & x1_x0_d[1];
  assign z7_z0[6] = x2_d[1] & x1_x0_d[2];
  assign z7_z0[7] = x2_d[1] & x1_x0_d[3];
endmodule
```

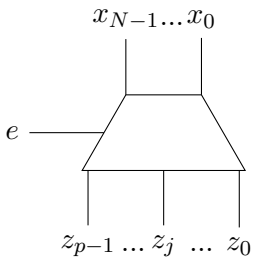
e infine, ad evidenziare quanto velocemente esplode il numero di termini (cioè esponenzialmente), un decoder da 4 a 16:

```
// un decoder da 4 a 16 che prende @x3_x0 come ingresso e attiva
// l'uscita @z15_z0 in codifica one-hot
module b4to16_decoder(x3_x0, z15_z0);
  input[3:0] x3_x0;
  output[15:0] z15_z0;

  assign z15_z0 = (x3_x0 == 'B0000) ? 'B0000_0000_0000_0001:
                  (x3_x0 == 'B0001) ? 'B0000_0000_0000_0010:
                  (x3_x0 == 'B0010) ? 'B0000_0000_0000_0100:
                  (x3_x0 == 'B0011) ? 'B0000_0000_0000_1000:
                  (x3_x0 == 'B0100) ? 'B0000_0000_0001_0000:
                  (x3_x0 == 'B0101) ? 'B0000_0000_0010_0000:
                  (x3_x0 == 'B0110) ? 'B0000_0000_0100_0000:
                  (x3_x0 == 'B0111) ? 'B0000_0000_1000_0000:
                  (x3_x0 == 'B1000) ? 'B0000_0001_0000_0000:
                  (x3_x0 == 'B1001) ? 'B0000_0010_0000_0000:
                  (x3_x0 == 'B1010) ? 'B0000_0100_0000_0000:
                  (x3_x0 == 'B1011) ? 'B0000_1000_0000_0000:
                  (x3_x0 == 'B1100) ? 'B0001_0000_0000_0000:
                  (x3_x0 == 'B1101) ? 'B0010_0000_0000_0000:
                  (x3_x0 == 'B1110) ? 'B0100_0000_0000_0000:
                /*(x3_x0 == 'B1111)?*/'B1000_0000_0000_0000;
endmodule

// implementazione a porte logiche
module b4to16_decoder_p(x3_x0, z15_z0);
  input[3:0] x3_x0;
  output[15:0] z15_z0;

  assign z15_z0[0] = ~x3_x0[3] & ~x3_x0[2] & ~x3_x0[1] & ~x3_x0[0];
  assign z15_z0[1] = ~x3_x0[3] & ~x3_x0[2] & ~x3_x0[1] & x3_x0[0];
  assign z15_z0[2] = ~x3_x0[3] & ~x3_x0[2] & x3_x0[1] & ~x3_x0[0];
  assign z15_z0[3] = ~x3_x0[3] & ~x3_x0[2] & x3_x0[1] & x3_x0[0];
  assign z15_z0[4] = ~x3_x0[3] & x3_x0[2] & ~x3_x0[1] & ~x3_x0[0];
  assign z15_z0[5] = ~x3_x0[3] & x3_x0[2] & ~x3_x0[1] & x3_x0[0];
  assign z15_z0[6] = ~x3_x0[3] & x3_x0[2] & x3_x0[1] & ~x3_x0[0];
  assign z15_z0[7] = ~x3_x0[3] & x3_x0[2] & x3_x0[1] & x3_x0[0];
  assign z15_z0[8] = x3_x0[3] & ~x3_x0[2] & ~x3_x0[1] & ~x3_x0[0];
  assign z15_z0[9] = x3_x0[3] & ~x3_x0[2] & ~x3_x0[1] & x3_x0[0];
  assign z15_z0[10] = x3_x0[3] & ~x3_x0[2] & x3_x0[1] & ~x3_x0[0];
  assign z15_z0[11] = x3_x0[3] & ~x3_x0[2] & x3_x0[1] & x3_x0[0];
  assign z15_z0[12] = x3_x0[3] & x3_x0[2] & ~x3_x0[1] & ~x3_x0[0];
  assign z15_z0[13] = x3_x0[3] & x3_x0[2] & ~x3_x0[1] & x3_x0[0];
  assign z15_z0[14] = x3_x0[3] & x3_x0[2] & x3_x0[1] & ~x3_x0[0];
  assign z15_z0[15] = x3_x0[3] & x3_x0[2] & x3_x0[1] & x3_x0[0];
endmodule

// implementazione gerarchica
module b4to16_decoder_g(x3_x0, z15_z0);
  input[3:0] x3_x0;
  output[15:0] z15_z0;
```

```
52
53    wire[3:0] x3_x2_d;
54    wire[3:0] x1_x0_d;
55
56    b2to4_decoder b2to4_1 (x3_x0[3:2], x3_x2_d);
57    b2to4_decoder b2to4_2 (x3_x0[1:0], x1_x0_d);
58
59    assign z15_z0[0] = x3_x2_d[0] & x1_x0_d[0];
60    assign z15_z0[1] = x3_x2_d[0] & x1_x0_d[1];
61    assign z15_z0[2] = x3_x2_d[0] & x1_x0_d[2];
62    assign z15_z0[3] = x3_x2_d[0] & x1_x0_d[3];
63    assign z15_z0[4] = x3_x2_d[1] & x1_x0_d[0];
64    assign z15_z0[5] = x3_x2_d[1] & x1_x0_d[1];
65    assign z15_z0[6] = x3_x2_d[1] & x1_x0_d[2];
66    assign z15_z0[7] = x3_x2_d[1] & x1_x0_d[3];
67    assign z15_z0[8] = x3_x2_d[2] & x1_x0_d[0];
68    assign z15_z0[9] = x3_x2_d[2] & x1_x0_d[1];
69    assign z15_z0[10] = x3_x2_d[2] & x1_x0_d[2];
70    assign z15_z0[11] = x3_x2_d[2] & x1_x0_d[3];
71    assign z15_z0[12] = x3_x2_d[3] & x1_x0_d[0];
72    assign z15_z0[13] = x3_x2_d[3] & x1_x0_d[1];
73    assign z15_z0[14] = x3_x2_d[3] & x1_x0_d[2];
74    assign z15_z0[15] = x3_x2_d[3] & x1_x0_d[3];
75 endmodule
```

1.1.1 Decoder con enabler

Il problema dei decoder come appena descritti è che sono poco agili nell'espansione: non si possono costruire, come avevamo visto per i gli AND o gli OR, reti di più decoder combinati, a meno di non ridursi a ritrovare quelli che sono effettivamente i mintermini della tabella di verità (come si nota dagli esempi). Introduciamo per questo motivo il decoder con **enabler**:



Questi decoder hanno $N+1$ ingressi, cioè quelli normali più l'enabler, che ha il compito di "accendere" il decoder stesso. Fisicamente, potremmo semplicemente inserire il decoder e come ingresso aggiuntivo agli AND già predisposti, per avere che:

$$z_i = \begin{cases} y_i & e = 1 \\ 0 & e = 0 \end{cases}$$

e quindi:

$$\begin{cases} z_0 = e \cdot \overline{x_{N-1}} \cdot \overline{x_{N-2}} \cdot \dots \cdot \overline{x_1} \cdot \overline{x_0} \\ z_1 = e \cdot \overline{x_{N-1}} \cdot \overline{x_{N-2}} \cdot \dots \cdot \overline{x_1} \cdot x_0 \\ \dots \\ z_{p-2} = e \cdot x_{N-1} \cdot x_{N-2} \cdot \dots \cdot x_1 \cdot \overline{x_0} \\ z_{p-1} = e \cdot x_{N-1} \cdot x_{N-2} \cdot \dots \cdot x_1 \cdot x_0 \end{cases}$$

Adesso basta accorgersi che reti di decoder con $N > 2$ possono crearsi concatenando decoder a decoder, cioè usando un decoder con i bit più significativi in entrata per generare l'enabler di N nuovi decoder, i quali ricevono i bit meno significativi in entrata.

Ad esempio, se vogliamo creare un decoder `4to16` a partire da decoder `2to4`, useremo 4 decoder, con gli stessi input (x_0 e x_1), abilitati da un quinto decoder con input x_2 e x_3.

Vediamo un'esempio pratico, dato dalle implementazioni in Verilog degli stessi decoder visti prima, ma stavolta dotati di enabler (e posti in cascata, nelle sintesi gerarchiche, attraverso tali enabler). Si inizia col decoder 1 a 2:

```
// un decoder con enabler da 1 a 2 che prende @x0 come ingresso, @e
// come enabler e attiva l'uscita @z1_z0 in codifica one-hot
module b1to2_enb_decoder(x0, e, z1_z0);
  input x0;
  input e;
  output[1:0] z1_z0;

  assign z1_z0 = ({e, x0} == 'B10) ? 'B01:
                 ({e, x0} == 'B11) ? 'B10:
                 /*   don't care  */ 'B00;
endmodule

// implementazione a porte logiche
module b1to2_enb_decoder_p(x0, e, z1_z0);
  input x0;
  input e;
  output[1:0] z1_z0;

  assign z1_z0[0] = ~x0 & e;
  assign z1_z0[1] = x0 & e;
endmodule
```

Possiamo quindi definire il decoder da 2 a 4:

```
// un decoder con enabler da 2 a 4 che prende @x1_x0 come ingresso,
// @e come enabler e attiva l'uscita @z4_z0 in codifica one-hot
module b2to4_enb_decoder(x1_x0, e, z3_z0);
  input[1:0] x1_x0;
  input e;
  output[3:0] z3_z0;

  assign z3_z0 = ({e, x1_x0} == 'B100) ? 'B0001:
                 ({e, x1_x0} == 'B101) ? 'B0010:
                 ({e, x1_x0} == 'B110) ? 'B0100:
                 ({e, x1_x0} == 'B111) ? 'B1000:
                 /*     don't care    */ 'B0000;
endmodule

// implementazione a porte logiche
module b2to4_enb_decoder_p(x1_x0, e, z3_z0);
  input[1:0] x1_x0;
  input e;
  output[3:0] z3_z0;

  assign z3_z0[0] = ~x1_x0[1] & ~x1_x0[0] & e;
  assign z3_z0[1] = ~x1_x0[1] & x1_x0[0] & e;
  assign z3_z0[2] = x1_x0[1] & ~x1_x0[0] & e;
  assign z3_z0[3] = x1_x0[1] & x1_x0[0] & e;
endmodule

```

```
// implementazione gerarchica
module b2to4_enb_decoder_g(x1_x0, e, z3_z0);
  input[1:0] x1_x0;
  input e;
  output[3:0] z3_z0;

  wire[1:0] enb;

  b1to2_enb_decoder b1to2_1 (x1_x0[0], enb[1], z3_z0[3:2]);
  b1to2_enb_decoder b1to2_2 (x1_x0[0], enb[0], z3_z0[1:0]);
  b1to2_enb_decoder b1to2_c (x1_x0[1], e, enb);
endmodule
```

un decoder da 3 a 8:

```
// un decoder con enabler da 3 a 8 che prende @x2_x0 come ingresso,
// @e come enabler e attiva l'uscita @z7_z0 in codifica one-hot
module b3to8_enb_decoder(x2_x0, e, z7_z0);
  input[2:0] x2_x0;
  input e;
  output[7:0] z7_z0;

  assign z7_z0 = ({e, x2_x0} == 'B1000) ? 'B0000_0001:
                 ({e, x2_x0} == 'B1001) ? 'B0000_0010:
                 ({e, x2_x0} == 'B1010) ? 'B0000_0100:
                 ({e, x2_x0} == 'B1011) ? 'B0000_1000:
                 ({e, x2_x0} == 'B1100) ? 'B0001_0000:
                 ({e, x2_x0} == 'B1101) ? 'B0010_0000:
                 ({e, x2_x0} == 'B1110) ? 'B0100_0000:
                 ({e, x2_x0} == 'B1111) ? 'B1000_0000:
                 /*      don't care     */ 'B0000_0000;
endmodule

// implementazione a porte logiche
module b3to8_enb_decoder_p(x2_x0, e, z7_z0);
  input[2:0] x2_x0;
  input e;
  output[7:0] z7_z0;

  assign z7_z0[0] = ~x2_x0[2] & ~x2_x0[1] & ~x2_x0[0] & e;
  assign z7_z0[1] = ~x2_x0[2] & ~x2_x0[1] & x2_x0[0] & e;
  assign z7_z0[2] = ~x2_x0[2] & x2_x0[1] & ~x2_x0[0] & e;
  assign z7_z0[3] = ~x2_x0[2] & x2_x0[1] & x2_x0[0] & e;
  assign z7_z0[4] = x2_x0[2] & ~x2_x0[1] & ~x2_x0[0] & e;
  assign z7_z0[5] = x2_x0[2] & ~x2_x0[1] & x2_x0[0] & e;
  assign z7_z0[6] = x2_x0[2] & x2_x0[1] & ~x2_x0[0] & e;
  assign z7_z0[7] = x2_x0[2] & x2_x0[1] & x2_x0[0] & e;
endmodule

// implementazione gerarchica
module b3to8_enb_decoder_g(x2_x0, e, z7_z0);
  input[2:0] x2_x0;
  input e;
  output[7:0] z7_z0;

  wire[1:0] enb;

  b2to4_decoder b2to4_1 (x2_x0[1:0], enb[1], z7_z0[7:4]);
  b2to4_decoder b2to4_2 (x2_x0[1:0], enb[0], z7_z0[3:0]);
  b1to2_decoder b1to2_c (x2_x0[2], enb);
endmodule
```

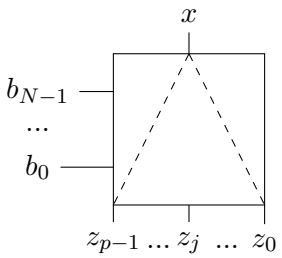
e infine, di cui notiamo la sintesi gerarchica molto più immediata rispetto al caso senza enabler, un decoder da 4 a 16:

```
// un decoder con enabler da 4 a 16 che prende @x3_x0 come ingresso,
// @e come enabler e attiva l'uscita @z15_z0 in codifica one-hot
module b4to16_enb_decoder(x3_x0, e, z15_z0);
  input[3:0] x3_x0;
  input e;
  output[15:0] z15_z0;

  assign z15_z0 = ({e, x3_x0} == 'B10000) ? 'B0000_0000_0000_0001:
                  ({e, x3_x0} == 'B10001) ? 'B0000_0000_0000_0010:
                  ({e, x3_x0} == 'B10010) ? 'B0000_0000_0000_0100:
                  ({e, x3_x0} == 'B10011) ? 'B0000_0000_0000_1000:
                  ({e, x3_x0} == 'B10100) ? 'B0000_0000_0001_0000:
                  ({e, x3_x0} == 'B10101) ? 'B0000_0000_0010_0000:
                  ({e, x3_x0} == 'B10110) ? 'B0000_0000_0100_0000:
                  ({e, x3_x0} == 'B10111) ? 'B0000_0000_1000_0000:
                  ({e, x3_x0} == 'B11000) ? 'B0000_0001_0000_0000:
                  ({e, x3_x0} == 'B11001) ? 'B0000_0010_0000_0000:
                  ({e, x3_x0} == 'B11010) ? 'B0000_0100_0000_0000:
                  ({e, x3_x0} == 'B11011) ? 'B0000_1000_0000_0000:
                  ({e, x3_x0} == 'B11100) ? 'B0001_0000_0000_0000:
                  ({e, x3_x0} == 'B11101) ? 'B0010_0000_0000_0000:
                  ({e, x3_x0} == 'B11110) ? 'B0100_0000_0000_0000:
                  ({e, x3_x0} == 'B11111) ? 'B1000_0000_0000_0000:
                  /*       don't care    */ 'B0000_0000_0000_0000;
endmodule

// implementazione a porte logiche
module b4to16_enb_decoder_p(x3_x0, e, z15_z0);
  input[3:0] x3_x0;
  input e;
  output[15:0] z15_z0;

  assign z15_z0[0] = ~x3_x0[3] & ~x3_x0[2] & ~x3_x0[1] & ~x3_x0[0]
                                                                & e;
  assign z15_z0[1] = ~x3_x0[3] & ~x3_x0[2] & ~x3_x0[1] & x3_x0[0]
                                                                & e;
  assign z15_z0[2] = ~x3_x0[3] & ~x3_x0[2] & x3_x0[1] & ~x3_x0[0]
                                                                & e;
  assign z15_z0[3] = ~x3_x0[3] & ~x3_x0[2] & x3_x0[1] & x3_x0[0] & e;
  assign z15_z0[4] = ~x3_x0[3] & x3_x0[2] & ~x3_x0[1] & ~x3_x0[0]
                                                                & e;
  assign z15_z0[5] = ~x3_x0[3] & x3_x0[2] & ~x3_x0[1] & x3_x0[0] & e;
  assign z15_z0[6] = ~x3_x0[3] & x3_x0[2] & x3_x0[1] & ~x3_x0[0] & e;
  assign z15_z0[7] = ~x3_x0[3] & x3_x0[2] & x3_x0[1] & x3_x0[0] & e;
  assign z15_z0[8] = x3_x0[3] & ~x3_x0[2] & ~x3_x0[1] & ~x3_x0[0]
                                                                & e;
  assign z15_z0[9] = x3_x0[3] & ~x3_x0[2] & ~x3_x0[1] & x3_x0[0] & e;
  assign z15_z0[10] = x3_x0[3] & ~x3_x0[2] & x3_x0[1] & ~x3_x0[0]
                                                                & e;
  assign z15_z0[11] = x3_x0[3] & ~x3_x0[2] & x3_x0[1] & x3_x0[0] & e;
  assign z15_z0[12] = x3_x0[3] & x3_x0[2] & ~x3_x0[1] & ~x3_x0[0]
                                                                & e;
  assign z15_z0[13] = x3_x0[3] & x3_x0[2] & ~x3_x0[1] & x3_x0[0] & e;
  assign z15_z0[14] = x3_x0[3] & x3_x0[2] & x3_x0[1] & ~x3_x0[0] & e;
  assign z15_z0[15] = x3_x0[3] & x3_x0[2] & x3_x0[1] & x3_x0[0] & e;
endmodule

// implementazione gerarchica
```

```
module b4to16_enb_decoder_g(x3_x0, e, z15_z0);
  input[3:0] x3_x0;
  input e;
  output[15:0] z15_z0;

  wire[3:0] enb;

  b2to4_enb_decoder b2to4_1 (x3_x0[1:0], enb[3], z15_z0[15:12]);
  b2to4_enb_decoder b2to4_2 (x3_x0[1:0], enb[2], z15_z0[11:8]);
  b2to4_enb_decoder b2to4_3 (x3_x0[1:0], enb[1], z15_z0[7:4]);
  b2to4_enb_decoder b2to4_4 (x3_x0[1:0], enb[0], z15_z0[3:0]);
  b2to4_enb_decoder b2to4_c (x3_x0[3:2], e, enb);
endmodule
```

1.2 Demultiplexer

Il demultiplexer è una rete con $N+1$ ingressi e $p = 2^N$ uscite:



Chiamiamo x la **variabile da commutare**, e le altre **variabili di comando** (b). La j-esima uscita insegue la variabile da commutare se e solo se:

$$(b_{n-1}, ..., b_0)_2 = j$$

altrimenti vale 0. Questo significa che il demultiplexer invia il suo input, x, all'output z_j tale che i controlli $b_{N-1}...b_0$ sono la codifica binaria di j.

Il multiplexer, fisicamente, è identico ad un decoder con enabler: si fa la parte di decoding con il:

$$\begin{cases} z_0 = \overline{x_{N-1}} \cdot \overline{x_{N-2}} \cdot ... \cdot \overline{x_1} \cdot \overline{x_0} \\ z_1 = \overline{x_{N-1}} \cdot \overline{x_{N-2}} \cdot ... \cdot \overline{x_1} \cdot x_0 \\ ... \\ z_{p-2} = x_{N-1} \cdot x_{N-2} \cdot ... \cdot x_1 \cdot \overline{x_0} \\ z_{p-1} = x_{N-1} \cdot x_{N-2} \cdot ... \cdot x_1 \cdot x_0 \end{cases}$$

di prima, e si moltiplica per x per ottenere il comportamento desiderato:

$$\begin{cases} z_0 = x \cdot \overline{x_{N-1}} \cdot \overline{x_{N-2}} \cdot ... \cdot \overline{x}_1 \cdot \overline{x_0} \\ z_1 = x \cdot \overline{x_{N-1}} \cdot \overline{x_{N-2}} \cdot ... \cdot \overline{x_1} \cdot x_0 \\ ... \\ z_{p-2} = x \cdot x_{N-1} \cdot x_{N-2} \cdot ... \cdot x_1 \cdot \overline{x_0} \\ z_{p-1} = x \cdot x_{N-1} \cdot x_{N-2} \cdot ... \cdot x_1 \cdot x_0 \end{cases}$$

Con $x = e$ questo è un decoder con enabler x.

Vediamo infatti l'implementazione in Verilog di un demultiplexer da 1 a 2:

```
// un demultiplexer da 1 a 2 che prende @x come ingresso, @b0 come
// comando e @z1_z0 come uscita
module b1to2_demuxer(x0, b0, z1_z0);
  input x0;
  input b0;
  output[1:0] z1_z0;

  assign z1_z0 = ({x0, b0} == 'B10) ? 'B01:
                 ({x0, b0} == 'B11) ? 'B10:
                 /*    don't care  */ 'B00;
endmodule

// implementazione a porte logiche
module b1to2_demuxer(x0, b0, z1_z0);
  input x0;
  input b0;
  output[1:0] z1_z0;

  assign z1_z0[0] = ~x0 & b0;
  assign z1_z0[1] = x0 & b0;
endmodule
```

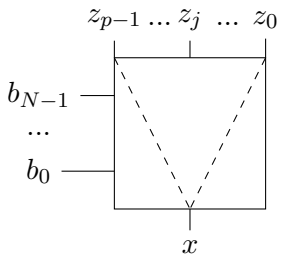
e come se ne può ricavare uno da 1 a 4:

```
// un demultiplexer da 1 a 4 che prende @x0 come ingresso, @b1_b0
// come comando e @z4_z0 come uscita
module b1to4_demuxer(x0, b3_b0, z3_z0);
  input x0;
  input[1:0] b1_b0;
  output[3:0] z3_z0;

  assign z3_z0 = ({x0, b3_b0} == 'B100) ? 'B0001:
                 ({x0, b3_b0} == 'B101) ? 'B0010:
                 ({x0, b3_b0} == 'B110) ? 'B0100:
                 ({x0, b3_b0} == 'B111) ? 'B1000:
                 /*      don't care    */ 'B0000;
endmodule

// implementazione a porte logiche
module b1to4_demuxer(x0, b1_b0, z3_z0);
  input x0;
  input[1:0] b1_b0;
  output[3:0] z3_z0;

  assign z3_z0[0] = ~b1_b0[1] & ~x1_x0[0] & x0;
  assign z3_z0[1] = ~b1_b0[1] & x1_x0[0] & x0;
  assign z3_z0[2] = b1_b0[1] & ~x1_x0[0] & x0;
  assign z3_z0[3] = b1_b0[1] & x1_x0[0] & x0;
endmodule

// implementazione gerarchica
module b1to4_demuxer(x0, b1_b0, z3_z0);
  input x0;
  input[1:0] b1_b0;
  output[3:0] z3_z0;

  wire[1:0] enb;

  b1to2_demuxer b1to2_1 (b1_b0[0], enb[1], z3_z0[3:2]);
  b1to2_demuxer b1to2_2 (b1_b0[0], enb[0], z3_z0[1:0]);
  b1to2_demuxer b1to2_c (b1_b0[1], x0, enb);
```

```
endmodule
```

1.3 Multiplexer

Il multiplexer è il duale del demultiplexer: una rete con $N + 2^N$ ingressi e 1 uscita:



Gli ingressi b_i si chiamano variabili di comando, e selezionano l'ingresso connesso all'uscita come:

$$z = x_i \Leftrightarrow (b_{N-1}, ..., b_1, b_0) = i$$

Abbiamo detto che il multiplexer è il duale del demultiplexer: se quest'ultimo prendeva un segnale x e lo inviava al j-esimo output sulla base della codifica di j ottenuta alle variabili di controllo, il multiplexer prende il j-esimo ingresso, secondo gli stessi canoni, e lo invia alla linea x di uscita.

Alla base della sintesi di un multiplexer sta un decoder: infatti, abbiamo che quest'ultimo seleziona uno solo (*one-hot*) degli output, che possiamo moltiplicare (mettiamo una AND) per l'ingresso corrispondente. Visto che solo uno degli output in uscita dagli AND è attivo in un dato momento, possiamo ricombinare il segnale finlle con un unico grande OR.

Come prima, possiamo eliminare gli AND in cascata dal decoder connettendoli agli AND già contenuti in esso.

Otteniamo quindi la descrizione algebrica (si noti che adesso abbiamo fatto sintesi $\rightarrow$ descrizione, mentre fino a questo punto avevamo fatto l'operazione inversa, descrizione $\rightarrow$ sintesi):

$$\begin{aligned} z = x_0 \cdot \overline{b_{N-1}} \cdot \overline{b_{N-2}} \cdot ... \cdot \overline{b_1} \cdot \overline{b_0} + \\ x_1 \cdot \overline{b_{N-1}} \cdot \overline{b_{N-2}} \cdot ... \cdot \overline{b_1} \cdot b_0 + \\ ... + \\ x_{p-2} \cdot b_{N-1} \cdot b_{N-2} \cdot ... \cdot b_1 \cdot \overline{b_0} + \\ x_{p-1} \cdot b_{N-1} \cdot b_{N-2} \cdot ... \cdot b_1 \cdot b_0 \end{aligned}$$

Notiamo che il multiplexer è una rete a 2 livelli di logica: il segnale passerà al massimo da una AND e una OR. Le NOT sugli ingressi non si contano, in quanto in una rete fisica le variabili di comando proverranno da registri, che forniscono già una versione negata del loro output senza bisogno di ulteriori inversori.

Vediamo quindi un'implementazione in Verilog di un multiplexer da 2 a 1:

```
// un multiplexer da 2 a 1 che prende @x1_x0 come ingresso, @b0 come
// comando e @z0 come uscita
module b2to1_muxer(x1_x0, b0, z0);
  input[1:0] x1_x0;
  input b0;
  output z0;
```

```
7
8   assign z0 = ({x1_x0, b0} == 'B000) ? 'B0:
9               ({x1_x0, b0} == 'B001) ? 'B0:
10              ({x1_x0, b0} == 'B010) ? 'B1:
11              ({x1_x0, b0} == 'B011) ? 'B0:
12              ({x1_x0, b0} == 'B100) ? 'B0:
13              ({x1_x0, b0} == 'B101) ? 'B1:
14              ({x1_x0, b0} == 'B110) ? 'B1:
15            /*({x1_x0, b0} == 'B111)?*/'B1;
16 endmodule
17
18 // implementazione a porte logiche
19 module b2to1_muxer_p(x1_x0, b0, z0);
20   input[1:0] x1_x0;
21   input b0;
22   output z0;
23
24   assign z0 = x1_x0[1] & b0 | x1_x0[0] & ~b0;
25 endmodule
26
27 // implementazione via decoder
28 module b2to1_muxer_d(x1_x0, b0, z0);
29   input[1:0] x1_x0;
30   input b0;
31   output z0;
32
33   wire[1:0] s1_s0;
34
35   b1to2_decoder b1to2(b0, s1_s0);
36
37   assign z0 = s1_s0[1] & x1_x0[1] | s1_s0[0] & x1_x0[0];
38 endmodule
```

e come se ne può ricavare uno da 4 a 1:

```
1  // un multiplexer da 4 a 1 che prende @x3_x0 come ingresso, @b1_b0
2  // come comando e @z0 come uscita
3
4  // un implementazione a tabella di verit  sarebbe troppo ingombrante
5  // da risultare utile (6 input per 64 righe di tabella...)
6
7  // implementazione a porte logiche
8  module b4to1_muxer_d(x3_x0, b1_b0, z0);
9    input[3:0] x3_x0;
10   input[1:0] b1_b0;
11   output z0;
12
13   assign z0 = (x3_x0[3] & b1_b0[1] & b1_b0[1]) |
14               (x3_x0[2] & b1_b0[1] & ~b1_b0[1]) |
15               (x3_x0[1] & ~b1_b0[1] & b1_b0[1]) |
16               (x3_x0[0] & ~b1_b0[1] & ~b1_b0[1]);
17 endmodule
18
19 // implementazione via decoder
20 module b4to1_muxer_d(x3_x0, b1_b0, z0);
21   input[3:0] x3_x0;
22   input[1:0] b1_b0;
23   output z0;
24
25   wire[3:0] s3_s0;
26
```

```
27   b2to4_decoder b2to4(b1_b0, s3_s0);
28
29   assign z0 = s3_s0[3] & x3_x0[3] | s3_s0[2] & x3_x0[2]
30             | s3_s0[1] & x3_x0[1] | s3_s0[0] & x3_x0[0];
31 endmodule
32
33 // implementazione gerarchica
34 module b4to1_muxer_g(x3_x0, b1_b0, z0);
35   input[3:0] x3_x0;
36   input[1:0] b1_b0;
37   output z0;
38
39   wire [1:0] s1_s0;
40
41   b2to1_muxer b2to1_1 (x3_x0[3:2], b1_b0[0], s1_s0[1]);
42   b2to1_muxer b2to1_2 (x3_x0[1:0], b1_b0[0], s1_s0[0]);
43   b2to1_muxer b2to1_3 (s1_s0, b1_b0[1], z0);
44 endmodule
```

1.3.1 Multiplexer come rete combinatoria universale

Dimostriamo il seguente teorema:

Teorema 1.1: Multiplexer come rete combinatoria universale

Un multiplexer con N variabili di comando è in grado di realizzare qualunque legge combinatoria ad N ingressi ed un uscita, connettendo i 2^N ingressi a generatori di costante.

Abbiamo che:

- Un multiplexer si ricava con porte AND, OR e NOT a due livelli di logica;
- Un multiplexer realizza qualsiasi rete combinatoria ad un'uscita;
- una rete a più uscite può essere scomposta in più reti con le uscite messe "in parellelo".

Allora qualsiasi rete combinatoria può essere creata combinando AND, OR e NOT su due livelli di logica.

Inoltre, si può dimostrare che per qualsiasi tabella di verità ad N ingressi, si può trovare una rete che la implementa tramite un multiplexer a $N-1$ variabili di comando, e al più porte NOT.

1.4 Modello strutturale universale per reti combinatorie

Vediamo adesso un modo per sintetizzare una rete logica ad N ingressi ed M uscite a partire da una tabella di verità. Si prende prima di tutto un decoder con N ingressi, e si creano M linee parallele alle 2^N (che è anche il numero delle righe della tabella di verità) linee di uscita del decoder. Si combinano quindi queste linee di uscita attraverso OR su ogni intersezione che corrisponde ad una certa cella della tabella di verità.

1.4.1 Riduzione dei costi

Definiamo informalmente il costo come ridotto quando si usano meno porte logiche. Troviamo quindi un modo per ridurre il costo della rete creata. Avremo che, inizialmente, tutte le uscite si presentano in una forma canonica **SP**, che sta per Somma di Prodotti, del tipo:

$$z_j = x_{n-1} \cdot ... \cdot x_0 + ... + x_{n-1} \cdot ... \cdot x_0$$

con la possibilità di complementare qualsiasi x. Questa forma equivale effettivamente a una forma normale disgiuntiva.

Possiamo quindi usare le proprietà dell'algebra di Boole per raggruppare e semplificare i termini. Vogliamo un algoritmo che ci permetta di eseguire questi passaggi in modo ordinato, e ci porti sempre alla soluzione ottimale.