Functional Block: Decoders

A Decoder converts $n$ binary bits to a maximum of $2^n$ unique output lines.

An $m$-to-$n$ line decoder, where $m \leq 2^n$, can be used to:

- Generate $2^n$ (or fewer) minterms,
- Select one-of $2^n$ items

Decoders are sometimes known as demultiplexers when enabled with a separate data-in line.
2-to-4 Line Decoder

This device takes:

\( n = 2 \) input lines

and decodes minterms for:

\( m = 2^n = 4 \) output lines.

---

2-to-4 Line Demultiplexer

This device takes:

\( n = 2 \) input lines

and decodes minterms for:

\( m = 2^2 = 4 \) output lines

where each output is:

ANDed with an input, X.

If X is viewed as an Enable, all outputs are 0 for X = 0 and one output is 1 for X = 1.

If X is viewed as Data, then this data is sent to one or the outputs.
Example: 74F138 Demultiplexer

<table>
<thead>
<tr>
<th>Enables E1 E2 E3</th>
<th>Inputs x y z</th>
<th>Outputs D0 D1 D2 D3 D4 D5 D6 D7</th>
</tr>
</thead>
<tbody>
<tr>
<td>x x 0</td>
<td>x x x</td>
<td>1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>x 1 x</td>
<td>x x x</td>
<td>1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>1 x x</td>
<td>x x x</td>
<td>1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>0 0 1</td>
<td>0 0 0</td>
<td>0 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>0 0 1</td>
<td>0 0 1</td>
<td>1 0 1 1 1 1 1 1</td>
</tr>
<tr>
<td>0 0 1</td>
<td>0 1 0</td>
<td>1 1 0 1 1 1 1 1</td>
</tr>
<tr>
<td>0 0 1</td>
<td>0 1 1</td>
<td>1 1 1 0 1 1 1 1</td>
</tr>
<tr>
<td>0 0 1</td>
<td>1 0 0</td>
<td>1 1 1 1 0 1 1 1</td>
</tr>
<tr>
<td>0 0 1</td>
<td>1 0 1</td>
<td>1 1 1 1 1 0 1 1</td>
</tr>
<tr>
<td>0 0 1</td>
<td>1 1 0</td>
<td>1 1 1 1 1 1 0 1</td>
</tr>
<tr>
<td>0 0 1</td>
<td>1 1 1</td>
<td>1 1 1 1 1 1 1 0</td>
</tr>
</tbody>
</table>

Note: This "Truth Table" uses the x (or - ) to mean "this could be either 0 or 1". Thus, it "compacts" some of $2^6 = 64$ lines.

Implementing Logic with Decoders

Decoders provide minterms directly. Simply "OR" the appropriate minterm outputs to make any logic function desired.

Active low decoders behave as the first NAND gate in a NAND-NAND, Sum of Products implementation.

Active high decoders behave as first stage AND gates in a AND-OR Sum of Products implementation.

Two or more active high decoders driven from different bits of a binary code can be used to form minterms by "ANDING" their outputs. Similarly, active low decoders can be used to form minterms by "ORING" their outputs.
Example 1: $F(A,B) = \Sigma m(0,3)$

For this we use a 2-to-4 line decoder and sum minterms 0 and 3 with an OR gate:

Example 1: $F(X,Y,Z) = \Sigma m(0,3,5,6)$
Implementing Larger Minterms

Minterm m15 is formed by "ANDING" the D3 outputs of each decoder.

Similarly m0 is formed by "ANDING" the D0 outputs of each decoder.

What minterm is formed by "ANDING" D1 (upper) and D2 (lower) outputs?

This works best with widely scattered, sparse minterms.

---

Functional Block: Encoders

- Encoders perform the "inverse" operation of decoders, taking a code in one format and encoding it into another format.

- Many encoders consist of just OR gates. For example an 8-to-3 binary encoder consists of three 4-input OR gates, OR2, OR1 and OR0. Input Ii, i = 0,…,7 is connected to an input on ORj if the binary representation of i has a 1 in position j.

- A priority encoder is used to generate a code for the "most significant" bit set in a string of bits. This can be used to find the first one in a word, or to select external events in priority order. An example of a MSI priority encoder is the 74F148, 8 line to 3 line priority encoder. It can be cascaded to encode higher numbers of bits.
Encoder Example

- Encode 4 lines 0, 1, 2, 3 into the corresponding binary codes.

Review: Decoders and Encoders

A **Decoder** converts $n$ binary bits to a maximum of $2^n$ unique output lines.

Decoders are sometimes know as demultiplexers when enabled with a separate data-in line.

Decoders implement minterms directly.

Use a decoder and an OR gate to form Sum-of-Minterms directly.

Encoders perform the "inverse" operation of decoders, taking a code in one format and encoding it into another format.
Multiplexers

A Multiplexer (MUX) is another common functional block.

A Multiplexer uses \( n \) binary select bits to choose from a maximum of \( 2^n \) unique input lines.

Like a decoder, it decodes minterms internally.

Unlike a decoder, it has only one output line.

The decoded minterms are used to select data from one of up to \( 2^n \) unique data input lines.

The output of the multiplexer is the data input whose index is specified by the \( n \) bit code.

Example: A 4-to-1 multiplexer

The 4-to-1 line
Multiplexer uses the same minterm decoder core.

It is like a demultiplexer with individual data input lines (instead of just one) and an output OR gate.
Multiplexer Versus Decoder

Note how similar the two are internally.

Functions with Multiplexers

It is possible to implement any Boolean function of \( n \) variables with a \( 2^n \) input multiplexer.

Simply tie each input to the "1" or "0" line as desired.

It is also possible to implement any \( n+1 \) variable function with a \( 2^n \) multiplexer. Simply use the \( (n+1)st \) variable in true or complement form depending upon what the truth table requires.

A Boolean function of more than \( n \) variables can be partitioned into several easily implemented sub-functions defined on a subset of the variables.

The multiplexer will then select among these sub-functions.
Example: Gray to Binary Code

The Gray code has adjacent elements separated by only one bit change.

We wish to convert a 3-bit Gray code to a binary code.

The function table on the right documents the required conversion.

⇒

The Gray to Binary Code Converter requires us to implement three separate, three-input Boolean functions.

Gray to Binary (Continued)

First step:

Let's get the function table into a logical order by re-ordering the input Gray code values in binary sequence:

⇒

By inspection:

\[ x = F(A,B,C) = \Sigma m(1, 3, 5, 7) \]
\[ y = G(A,B,C) = \Sigma m(1, 2, 5, 6) \]
\[ z = H(A,B,C) = \Sigma m(1, 2, 4, 7) \]
Gray to Binary (Continued)

We know that \(2^n\) to 1 Multiplexers can be used to implement arbitrary functions of \(n\) bits. We simply connect the inputs to '0'.

Use two eight-input multiplexers to implement functions for \(y\) and \(z\):

\[
\Rightarrow
\]

In this case, the MUX elements are acting like a "Read Only Memory" (ROM).
Other MUX Implementations

We can also use two 4-to-1 MUX blocks and implement \( y \) and \( z \).

Suppose we factor out \( A \) and use \( B \) and \( C \) as the select inputs to the multiplexers:

\[
\begin{array}{c|cccc}
A & 00 & 01 & 11 & B10 \\
\hline
Y & 0 & 1 & 0 & 1 \\
\end{array}
\]

\[
\begin{array}{c|cccc}
A & 00 & 01 & 11 & B10 \\
\hline
Z & 0 & 1 & 0 & 1 \\
\end{array}
\]

As before, \( x = C \).

MUX Implementations (Cont.)

Factoring out variable \( A \) leads to the following implementation with two, 4-to-1 Multiplexers:
MUX: (Cont.) Factoring Out C

We could have factored out other variables. As in the book, we will factor out C and apply AB to the select inputs:

<table>
<thead>
<tr>
<th>Gray</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>A B C</td>
<td>x</td>
</tr>
<tr>
<td>0 0 0</td>
<td>D0 = C</td>
</tr>
<tr>
<td>0 0 1</td>
<td>1</td>
</tr>
<tr>
<td>0 1 0</td>
<td>D1 = C</td>
</tr>
<tr>
<td>0 1 1</td>
<td>1</td>
</tr>
<tr>
<td>1 0 0</td>
<td>D2 = C</td>
</tr>
<tr>
<td>1 0 1</td>
<td>1</td>
</tr>
<tr>
<td>1 1 0</td>
<td>D3 = C</td>
</tr>
<tr>
<td>1 1 1</td>
<td>1</td>
</tr>
</tbody>
</table>

This is slightly larger than selecting A to factor out.

MUX: (Cont.) Factoring out B

<table>
<thead>
<tr>
<th>Gray</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>A B C</td>
<td>x</td>
</tr>
<tr>
<td>0 0 0</td>
<td>D0 = B</td>
</tr>
<tr>
<td>0 0 1</td>
<td>1</td>
</tr>
<tr>
<td>0 1 0</td>
<td>D1 = B</td>
</tr>
<tr>
<td>0 1 1</td>
<td>1</td>
</tr>
<tr>
<td>1 0 0</td>
<td>D2 = B</td>
</tr>
<tr>
<td>1 0 1</td>
<td>1</td>
</tr>
<tr>
<td>1 1 0</td>
<td>D3 = B</td>
</tr>
<tr>
<td>1 1 1</td>
<td>1</td>
</tr>
</tbody>
</table>

Note: We re-arranged the table (fixing A and C and varying B from 0 to 1 in each cell) to simplify this procedure. It still looks like factoring A was better.
MUX: (Cont.) Factoring out A

<table>
<thead>
<tr>
<th>Gray</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>A B C</td>
<td>x</td>
</tr>
<tr>
<td>0 0 0</td>
<td>0</td>
</tr>
<tr>
<td>1 0 0</td>
<td>0</td>
</tr>
<tr>
<td>0 0 1</td>
<td>1</td>
</tr>
<tr>
<td>1 0 1</td>
<td>1</td>
</tr>
<tr>
<td>0 1 0</td>
<td>0</td>
</tr>
<tr>
<td>1 1 0</td>
<td>0</td>
</tr>
<tr>
<td>0 1 1</td>
<td>1</td>
</tr>
<tr>
<td>1 1 1</td>
<td>1</td>
</tr>
</tbody>
</table>

Note: We re-arranged the table (fixing B and C and varying A from 0 to 1 in each cell) to simplify this procedure. Factoring A is best! Note also that x = C holds.

Summary

- Know the functions performed by the following functional blocks:
  - Decoders
  - Demultiplexers
  - Encoders
  - Multiplexers

- Know how to implement Boolean functions using:
  - Multiplexers
  - Decoders