A decoder is a logic circuit that converts coded inputs into coded outputs.

Each input code word produces a different output code word (there is a one-to-one mapping between inputs and outputs).
Decoder Example

- **BCD to seven-segment decoder**
  - has 4-bit BCD as input code and the “seven-segment code” as its output code

![Decoder Diagram]

Binary Decoder

- Accepts a \( n \)-bit binary input code and generates a 1-out-of-\( 2^n \) output code
- Used to activate exactly one of \( 2^n \) outputs based on \( n \)-bit input value
- Examples: 2-to-4, 3-to-8, 4-to-16, etc.
- Note: BCD to seven-segment decoder is NOT a binary decoder
  - Because multiple outputs active simultaneously
- Binary decoders are simple and general; can be used to build general decoders (shown later)
Gate Level Implementation of Decoders

- 1:2 decoders

- 2:4 decoders

How It Works

- 2:4 decoder:
  - input combination: “00”
  - output: O₀
  - input combination: “10”
  - output: O₂
Binary 2-to-4 Decoder

- Note that the outputs of the decoder correspond to the *minterms*: \( Y_i = m_i \)
  - e.g., \( Y_0 = I_1' \cdot I_0' \)
  - \( Y_1 = I_1' \cdot I_0 \) etc.

MSI 2-to-4 Decoder

- Input buffering (less load on input circuit)
- NAND gates (faster operation)

Note: "x" (don't care) notation.
Complete 74x139 Decoder

Two 2-to-4 decoders in a single packaging

(compare to 74x138 3-to-8 decoder, described next)

74x138: 3-to-8 decoder

- Commercially available MSI 3-to-8 decoder
  - Note that its outputs are active low
    » because TTL and CMOS 
    inverting 
    gates are faster 
    than non-inverting gates

- Logic equations for internal output signals include “enable” signals.
- Example:
  \[ Y_5 = G_1 \cdot G_{2A} \cdot G_{2B} \cdot C \cdot B' \cdot A \]
Truth Table for a 3-to-8 Decoder

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>G1 G2A_L G2B_L C B A</td>
<td>Y7_L Y6_L Y5_L Y4_L Y3_L Y2_L Y1_L Y0_L</td>
</tr>
<tr>
<td>0 x x x x x</td>
<td>1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>x 1 x x x x</td>
<td>1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>x x 1 x x x</td>
<td>1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>1 0 0 0 0 0</td>
<td>1 1 1 1 1 1 1 0</td>
</tr>
<tr>
<td>1 0 0 0 0 1</td>
<td>1 1 1 1 1 1 0 1</td>
</tr>
<tr>
<td>1 0 0 0 1 0</td>
<td>1 1 1 1 1 0 1 1</td>
</tr>
<tr>
<td>1 0 0 0 1 1</td>
<td>1 1 1 1 0 1 1 1</td>
</tr>
<tr>
<td>1 0 0 1 0 0</td>
<td>1 1 1 0 1 1 1 1</td>
</tr>
<tr>
<td>1 0 0 1 0 1</td>
<td>1 1 0 1 1 1 1 1</td>
</tr>
<tr>
<td>1 0 0 1 1 0</td>
<td>1 0 1 1 1 1 1 1</td>
</tr>
<tr>
<td>1 0 0 1 1 1</td>
<td>0 1 1 1 1 1 1 1</td>
</tr>
</tbody>
</table>

Because of the inversion bubbles, we have the following relations between internal and external signals:

- G2A = G2A_L'
- Y5 = Y5_L'
- etc.

3-to-8 Decoder Logic Diagram
Decoder Cascading

- Decoders can be cascaded hierarchically to decode larger code words

Example:
Design a 4-to-16 decoder using 74x128s (3-to-8 decoders)

More Cascading

5-to-32 decoder
### Decoder Applications

- **Microprocessor memory systems**
  - Selecting different banks of memory
- **Microprocessor input/output systems**
  - Selecting different devices
- **Microprocessor instruction decoding**
  - Enabling different functional units
- **Memory chips**
  - Enabling different rows of memory depending on address
- **Lots of other applications**

### Decoders as General-Purpose Logic

- \( n \)-to-\( 2^n \) decoders can implement any function of \( n \) variables
  - with the variables used as control inputs
  - the appropriate minterms summed to form the function

![3-to-8 decoder diagram](image-url)

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>EN</th>
<th>Y0</th>
<th>Y1</th>
<th>Y2</th>
<th>Y3</th>
<th>Y4</th>
<th>Y5</th>
<th>Y6</th>
<th>Y7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>A·B·C’</td>
<td>A·B’·C</td>
<td>A’·B·C’</td>
<td>A’·B·C</td>
<td>A’·B’·C’</td>
<td>A·B·C’</td>
<td>A·B’·C’</td>
<td>A·B·C’</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>A·B·C’</td>
<td>A·B’·C</td>
<td>A’·B·C’</td>
<td>A’·B·C</td>
<td>A’·B’·C’</td>
<td>A·B·C’</td>
<td>A·B’·C’</td>
<td>A·B·C’</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>A·B·C’</td>
<td>A·B’·C</td>
<td>A’·B·C’</td>
<td>A’·B·C</td>
<td>A’·B’·C’</td>
<td>A·B·C’</td>
<td>A·B’·C’</td>
<td>A·B·C’</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>A·B·C’</td>
<td>A·B’·C</td>
<td>A’·B·C’</td>
<td>A’·B·C</td>
<td>A’·B’·C’</td>
<td>A·B·C’</td>
<td>A·B’·C’</td>
<td>A·B·C’</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>A·B·C’</td>
<td>A·B’·C</td>
<td>A’·B·C’</td>
<td>A’·B·C</td>
<td>A’·B’·C’</td>
<td>A·B·C’</td>
<td>A·B’·C’</td>
<td>A·B·C’</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>A·B·C’</td>
<td>A·B’·C</td>
<td>A’·B·C’</td>
<td>A’·B·C</td>
<td>A’·B’·C’</td>
<td>A·B·C’</td>
<td>A·B’·C’</td>
<td>A·B·C’</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>A·B·C’</td>
<td>A·B’·C</td>
<td>A’·B·C’</td>
<td>A’·B·C</td>
<td>A’·B’·C’</td>
<td>A·B·C’</td>
<td>A·B’·C’</td>
<td>A·B·C’</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>A·B·C’</td>
<td>A·B’·C</td>
<td>A’·B·C’</td>
<td>A’·B·C</td>
<td>A’·B’·C’</td>
<td>A·B·C’</td>
<td>A·B’·C’</td>
<td>A·B·C’</td>
</tr>
</tbody>
</table>

Decoder generates appropriate minterm based on control signals (it "decodes" control signals)
Decoders as General-Purpose Logic

- $F_1 = A'\cdot B\cdot C\cdot D + A\cdot B'\cdot C\cdot D + A\cdot B\cdot C\cdot D$
- $F_2 = A\cdot B\cdot C\cdot D + A\cdot B\cdot C$
- $F_3 = A' + B' + C' + D'$

Customized Decoder Circuit

<table>
<thead>
<tr>
<th>CS_L</th>
<th>RD_L</th>
<th>A2</th>
<th>A1</th>
<th>A0</th>
<th>Output(s) to Assert</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>none</td>
</tr>
<tr>
<td>x</td>
<td>1</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>none</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>BILL_L, MARY_L</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>MARY_L, KATE_L</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>JOAN_L</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>PAUL_L</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>ANNA_L</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>FRED_L</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>DAVE_L</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>KATE_L</td>
</tr>
</tbody>
</table>

Truth table

Circuit diagram
Decoder-Based Circuits

Designing a circuit for the logic function
\[ F = \sum_{X,Y,Z} (0,2,3,5): \]
(a) Karnaugh map;
(b) NAND-based minimal sum-of-products;
(c) decoder-based canonical sum.

Multiple Decoding w/ a Single Decoder

Multiple decoding with a single decoder using decoder-based circuits.