Combinational Logic Design

Originals by: Charles R. Kime and Tom Kaminsksi
Modified for course use by: Kewal K. Saluja and Yu Hen Hu
© 2001 Prentice Hall, Inc

Design Hierarchy
- Combinatorial Circuits
- A combinatorial logic circuit has:
  - A set of n Boolean inputs,
  - A set of m Boolean outputs, and
  - n switching functions mapping the 2^n input combinations to a
    output such that the current output depends only on the
    current inputs.
- A block diagram:

Hierarchical Design
- The function mapping inputs to outputs may be very complex
  - To control complexity, we decompose the function into smaller
    pieces called blocks
  - The blocks are subdivided into finer blocks
  - The "leaves" in the hierarchy are called primitive blocks
- Example: 16 input parity tree
  - Top Level: 16 inputs, one output
  - 2nd Level: Five 4-bit parity trees in two levels
  - 3rd Level: Three 2-bit exclusive-OR functions
  - Primitive level: Four 2-input NANDs
  - The design requires 5 X 3 X 4 = 60 two-input NAND gates

Reusable Functions and Design
- Whenever possible, we try to decompose a complex
design into common, reusable function blocks
- These blocks are tested and well documented
- Computer-aided design (CAD) tools might include
  them in libraries
- Computer-aided manufacturing (CAM) tools might
  know how to manufacture and test them
- Other tools:
  - Schematic Capture
  - Logic Simulators
  - Timing Verifiers
  - Hardware Description Languages (HDL)

Top-Down verses Bottom-Up
- A Top-Down design proceeds from an abstract, high
  level specification to a more and more detailed design
  by decomposition and successive refinement
- A Bottom-Up design starts with detailed primitive
  elements and combines them into larger and larger and
  more complex functions
- Designs usually proceed from both directions
  simultaneously
  - Top-Down design answers: What are we building?
  - Bottom-Up design answers: How do we build it?
  - Top-Down controls complexity while Bottom-Up
    "sweats" the details

Analysis Procedure
- Switching Functions from Logic Diagrams
  - Given a logic diagram, the analysis process provides a set
    of Boolean equations, a truth table, or a verbal
    explanation of circuit behavior.
  - Procedure:
    1. Determine that the circuit is combinational (no feedback loops),
    then:
    2. Identify and label all gate outputs that are a function of the input
       variables. Obtain the Boolean functions for these labeled gate
       outputs.
    3. Identify and label all gate outputs that are a function of inputs or
       previously labeled gates. Obtain Boolean functions for them.
    4. Repeat Step 2 until all outputs are completed.
    5. Back substitute until all functions are specified in terms of inputs
       only.
**Analysis Example**

- **Step 2:** Label all outputs of gates near inputs.
- **Write Boolean equations for them:**
  - \( T_1 = \overline{B} + C \)
  - \( T_2 = B \overline{E} \)

**Analysis (Continued)**

- **Step 3:** Identify and label all gate outputs that are a function of inputs or previously labeled gates. Obtain Boolean functions for them.
  - \( T_3 = D + T_2 \)
  - **Step 4:** Repeat Step 3 until all done
    - \( T_4 = T_1 \cdot T_3 \)
    - \( F = A + T_4 \)

**Analysis Example: Code Converter**

- **Step 2:** Label gates derived from inputs and develop Boolean functions.
- **Step 3:** Label the next stage of gates and develop Boolean functions.

**Code Converter Analysis (Cont.)**

- The process terminates with all gate outputs defined. Proceeding with Step 4, substituting.

**Truth Tables from Logic Diagrams**

1. Determine the number of input variables, \( n \). There will be \( 2^n \) input vectors from zero to \( (2^n) - 1 \). Enter them in the table.
2. Label the outputs of selected gates with symbols and enter a column for each one in the table.
3. Obtain the truth table for the outputs of those gates that are a function of only input variables.
4. Proceed to fill in the outputs of all gates that are derived from inputs and previously calculated terms.
Truth Tables from Logic Diagrams

**Procedure:**
- Determine the number of input variables, n. There will be $2^n$ input vectors from zero to $2^n - 1$. Enter them in the table.
- Label the outputs of selected gates with symbols and enter a column for each one in the table.
- Obtain the truth table for the outputs of those gates that are a function of only input variables.
- Proceed to fill in the outputs of all gates that are derived from inputs and previously calculated terms.

**Example:** Find the function table for the code converter.

<table>
<thead>
<tr>
<th>ABCD</th>
<th>F0</th>
<th>F1</th>
<th>F2</th>
<th>w</th>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0011</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0101</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0110</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0111</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1001</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1010</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1011</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1100</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1101</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1110</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1111</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Complete Entries

**Finally we can fill in w to complete the table:**

<table>
<thead>
<tr>
<th>ABCD</th>
<th>F0</th>
<th>F1</th>
<th>F2</th>
<th>w</th>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0011</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0101</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0110</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0111</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1001</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1010</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1011</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1100</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1101</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1110</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1111</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

What Does the Circuit Do?

- By inspection, the output variable vector $(w,x,y,z)$ is just the input variable vector $(A,B,C,D)$ plus three.
- The function(s) $F(A,B,C,D) = (w,x,y,z)$ are:
  - "ADD THREE TO THE INPUT VECTOR"
  - Function $F_1$ has the meaning:
    - "ADD ONE TO THE UPPER TWO BITS"
  - Similarly, function $F_2$ has the meaning:
    - "ADD ONE TO THE UPPER BIT"
- Generally, it is not this obvious to figure out what the functions mean!

Final Note (and warning)

- The use of "Don't Cares" in the original specification can cloud the analysis.
  - Note that the functions for the "w" bit differ from the implementation in Ex. 3-2 of the book.
  - The book used "Don't Cares" to simplify the logic.
- This example here did not.

[Diagram of K-maps for w]
Logic Design: Functional Blocks

- Analysis: From a design to a specification of the behavior
  - Logic diagram to equations
  - Function table
  - "Word description" of circuit operation
- Design and Synthesis: From a specification to design implementation
  - Define the problem
  - Generate function table or equations
  - Minimize the Boolean function
  - Implement the circuit

Combinatorial Logic Implementation

- A combinatorial logic circuit has:
  - A set of \( n \) Boolean inputs,
  - A set of \( m \) Boolean outputs, and
  - A function mapping inputs to outputs.
- We think of the function as \( n \) separate Boolean functions of \( m \) inputs.
- Procedure:
  - Treat each output as a separate function
  - Minimize the equations for each function
  - Implement each function independently
  - Sometimes an implementation can share product or sum logic terms to arrive at a lower literal cost solution.

Design Procedure

- First, start with the specification of the circuit to be designed.
  - Note: this can sometimes require a lot of work to complete the specification process, especially if it is poorly specified initially.
- Second, follow these steps: We will study the design of a code converter to see these steps.
  - Identify the inputs and outputs
  - Derive truth table
  - Obtain simplified Boolean equations
  - Draw the logic diagram
  - Check your work to verify correctness.

Code Converter Design Example

- A code converter transforms one internal representation of data to another
- We will start with a table of the desired conversion and minimize the resulting multiple output Boolean function.
- Sometimes terms can be shared to minimize the implementation cost.
- The Problem:
  - Design a BCD to Excess-3 code converter
  - Specification:
    - BCD code -- 4-bit patterns "0000" to "1001" for digits 0 to 9 base 10
    - Excess-3 -- BCD code plus binary "0011" for digits 0 to 9 base 10
- Example (Cont.): BCD to Excess 3
  - Map functions and find minimum cost SOP equations for each

Example: BCD to Excess 3

<table>
<thead>
<tr>
<th>Input BCD</th>
<th>Output Excess-3</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABCD</td>
<td>WXYZ</td>
</tr>
<tr>
<td>0000</td>
<td>0011</td>
</tr>
<tr>
<td>0001</td>
<td>0100</td>
</tr>
<tr>
<td>0011</td>
<td>0110</td>
</tr>
<tr>
<td>0100</td>
<td>0111</td>
</tr>
<tr>
<td>0101</td>
<td>1000</td>
</tr>
<tr>
<td>0110</td>
<td>1001</td>
</tr>
<tr>
<td>0111</td>
<td>1010</td>
</tr>
<tr>
<td>1000</td>
<td>1011</td>
</tr>
<tr>
<td>1001</td>
<td>1011</td>
</tr>
</tbody>
</table>

Note: All BCD codes greater than "9" can be assigned "Don't Cares" in the K-Map. Such BCD codes are never possible.

Example (Cont.): BCD to Excess 3

- Map functions and find minimum cost SOP equations for each
Example (Cont.): BCD to Excess 3

- Next, we will manipulate the equations to expose some shared terms:
  - The term (C + D) can be used more than once to simplify the implementation
  - See Fig. 3-10 in Mano and Kime for the implementation

An Alternative: BCD to Excess 3

- Another Approach: Excess-3 is defined as BCD plus 3.
- Adding 3 to BCD to Excess-3:

\[
\begin{array}{ccc}
A & B & C & D \\
\hline
0 & 0 & 1 & 1 \\
\end{array}
\]

Here HA is a Half-Adder and FA is a Full-Adder (We will discuss these later in the chapter).

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:
- \( m=2 \) input lines
- and decodes minterms for:
  \( m=2^2 = 4 \) output lines.

2-to-4 Line Demultiplexer

This device takes:
- \( m=2 \) input lines
- and decodes minterms for:
  \( m=2^2 = 4 \) output lines
- where each output is:
  ANDised 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

74F138 truth table:

<table>
<thead>
<tr>
<th>Enables</th>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>D3 D2 D1 D0</td>
<td>0 0 1 1</td>
<td>0 1 1 1</td>
</tr>
<tr>
<td>0 1 0 0</td>
<td>0 1 1 1</td>
<td></td>
</tr>
<tr>
<td>0 0 1 0</td>
<td>0 0 0 0</td>
<td></td>
</tr>
<tr>
<td>0 1 0 1</td>
<td>0 0 1 0</td>
<td></td>
</tr>
<tr>
<td>0 0 1 1</td>
<td>0 0 0 0</td>
<td></td>
</tr>
<tr>
<td>0 1 1 0</td>
<td>0 0 0 0</td>
<td></td>
</tr>
<tr>
<td>0 1 1 1</td>
<td>0 0 0 0</td>
<td></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 \( 2^5 = 64 \) 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: \( F(A,B) = \sum m(0,3) \)

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

Example 1: \( F(X,Y,Z) = \sum m(0,3,5,6) \)

Implementing Larger Minterms

Minterm \( m_{15} \) is formed by "ANDING" the \( D_3 \) outputs of each decoder.

Similarly \( m_0 \) is formed by "ANDING" the \( D_0 \) outputs of each decoder.

What minterm is formed by "ANDING" \( D_1 \) (upper) and \( D_2 \) (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 \( I_i \), \( i = 0, \ldots, 7 \) is connected to an input on OR\( j \) 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 known 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.

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.

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: 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 reordering the input Gray code values in binary sequence:

\begin{tabular}{|c|c|c|c|c|c|}
\hline
Gray & Binary & x & y & z \\
\hline
A & B & C & x & y & z \\
\hline
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 1 & 1 & 1 & 1 \\
1 & 0 & 0 & 1 & 1 & 1 \\
1 & 1 & 1 & 0 & 0 & 1 \\
\hline
By inspection:
\end{tabular}

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

Note:
\( x(A,B,C) = C \), is an easy function to implement. (No logic gates needed!)
Function \( y(A,B,C) = B' \cdot C + B' \cdot C \) is a bit harder to implement.
Function \( z(A,B,C) \) looks familiar. What is it?

The K-Maps

\begin{tabular}{|c|c|c|c|c|c|c|c|c|}
\hline
A & B & C & Y & Z \\
\hline
A & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\
A & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\
A & 1 & 0 & 0 & 1 & 1 & 1 & 0 \\
A & 1 & 1 & 1 & 1 & 0 & 0 & 1 \\
\hline
Note:
\[ x(A,B,C) = C \], is an easy function to implement. (No logic gates needed!)
Function \( y(A,B,C) = B' \cdot C + B' \cdot C \) is a bit harder to implement.
Function \( z(A,B,C) \) looks familiar. What is it?

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:

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

MUX Implementations (Cont.)

Factoring out variable \( A \) leads to the following implementation with two, 4-to-1 MUXes:

\[ \begin{align*}
A & \quad \text{4-to-1} \\
A & \quad \text{4-to-1} \\
\end{align*} \]

As before, \( x = C \).
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 y z</td>
</tr>
<tr>
<td>0 0 0</td>
<td>0 y = B' 0 z = B</td>
</tr>
<tr>
<td>0 1 0</td>
<td>1 y = B' 1 z = B'</td>
</tr>
<tr>
<td>0 1 1</td>
<td>0 y = B' 1 z = B'</td>
</tr>
<tr>
<td>1 0 0</td>
<td>0 y = B 1 z = B'</td>
</tr>
<tr>
<td>1 0 1</td>
<td>1 y = B 0 z = B'</td>
</tr>
<tr>
<td>1 1 0</td>
<td>0 y = B 1 z = B</td>
</tr>
<tr>
<td>1 1 1</td>
<td>0 y = B 0 z = B</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 y z</td>
</tr>
<tr>
<td>0 0 0</td>
<td>0 D0 = 0 0 D0 = 0 0 D0 = A</td>
</tr>
<tr>
<td>1 0 0</td>
<td>1 D1 = 1 1 D1 = 1 1 D1 = A'</td>
</tr>
<tr>
<td>0 1 0</td>
<td>0 D2 = 0 1 D2 = 1 1 D2 = A'</td>
</tr>
<tr>
<td>1 1 0</td>
<td>0 D3 = 0 0 D3 = A</td>
</tr>
<tr>
<td>1 1 1</td>
<td>0 D3 = 1 1 D3 = 0 0 D3 = A</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

- Analysis
  - Forward – backward trace through the circuit to obtain output equations or truth table
  - Vice versa will also find the equations and truth table
- Know the functions performed by the following functional blocks:
  - Decoders, Demultiplexers, Encoders, Multiplexers
- Know how to implement Boolean functions using:
  - Multiplexers
  - Decoders