Design Hierarchy

Combinatorial Circuits
A combinatorial logic circuit has:
1. A set of $m$ Boolean inputs.
2. A set of $n$ Boolean outputs
3. A function mapping inputs to outputs such that the current output depends only on the current input.

A block diagram is shown below:

- m Boolean Inputs
- n Boolean Outputs
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: 5- four bit parity trees in two levels.
   3rd Level: 3, two-bit exclusive-OR functions.
   Primitive level: 4 two-input NANDs
The design requires $5 \times 3 \times 4 = 64$ two-input NAND gates.

Reusable Functions and Design

Wherever 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.

We will start first with ANALYSIS.

---

### Analysis Procedure

**Boolean Functions from Logic Diagrams**

Given a logic diagram, the analysis process culminates with a set of Boolean equations, a truth table, or a verbal explanation of the circuit behavior.

**FIRST: Determine that the circuit is combinatorial (no feedback loops), then:**

1. Identify and label all gate outputs that are a function of the input variables. Obtain the Boolean functions for these labeled gate outputs.

2. Identify and label all gate outputs that are a function of inputs or previously labeled gates. Obtain Boolean functions for them.

3. Repeat Step 2 until all outputs are completed.

4. Back substitute until all functions are specified in terms of inputs only.
Analyze the network below

Step 1:
First label all outputs of gates near inputs.

Then write Boolean equations for them:
\[ T_1 = B' + C \]
\[ T_2 = B \cdot E' \]

Analysis (Continued)

Step 2: Identify and label all gate outputs that are a function of inputs or previously labeled gates. Obtain Boolean functions for them.

Step 3: Repeat Step 2 until all done.
\[ T_4 = T_1 \cdot T_3 \]
\[ F = A + T_4 \]
Step 4: Back substitute until all functions are specified in terms of inputs only.

\[
\begin{align*}
F &= A + T4 \\
T4 &= T1 \cdot T3 \\
T3 &= D' + T2 \\
T2 &= B \cdot E' \\
T1 &= B' + C
\end{align*}
\]

Substituting:

\[
\begin{align*}
T3 &= D' + (B \cdot E') \\
T4 &= (B' + C) \cdot (D' + (B \cdot E')) \\
F &= A + (B' + C) \cdot (D' + (B \cdot E'))
\end{align*}
\]

---

### Analyze the Code Converter

Step 1: Label gates derived from inputs and develop Boolean functions. This gives us Output F0 and F1. By inspection:

\[
\begin{align*}
F0 &= C' \\
F1 &= C + D \\
z &= D'
\end{align*}
\]

Step 2: Label the next stage of gates and develop Boolean functions. Output F2 can now be labeled. This gives us:

\[
\begin{align*}
F2 &= B \cdot F1 \\
y &= F0 \oplus D \\
x &= B \oplus F1
\end{align*}
\]
Code Converter (Cont.)

The process terminates now with all gate outputs defined. Write the equations.

Step 4: Substituting we get:

\[ z = D' \quad F_0 = C' \quad y = F_0 \oplus D = C' \oplus D = C' \cdot D' + C \cdot D \]

\[ F_1 = C + D \]

\[ F_2 = B \cdot F_1 = B \cdot (C + D) \]

\[ x = B \oplus F_1 = B \oplus (C + D) \]

\[ = B' \cdot (C + D) + B \cdot (C + D)' \]

\[ = B' \cdot (C + D) + B \cdot C' \cdot D' \]

\[ w = A \oplus F_2 = A \oplus (B \cdot (C + D)) \]

\[ = A \cdot (B \cdot (C + D))' + A' \cdot (B \cdot (C + D)) \]

\[ = A \cdot (B' + (C + D))' + A' \cdot (B \cdot C + B \cdot D) \]

\[ = A \cdot (B' + C' \cdot D') + A' \cdot B \cdot C + A' \cdot B \cdot D \]

\[ = A \cdot B' + A \cdot C' \cdot D' + A' \cdot B \cdot C + A' \cdot B \cdot D \]

Truth Tables from 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.

Example: Find the function table for the code converter.

We will use the same markings as before and define a truth table on those values.
## Code Converter Truth Table

Four inputs give 16 input vectors. Start with F0, F1 and z:

<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></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0011</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0101</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0110</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0111</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1001</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1010</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1011</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1100</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1101</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1110</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1111</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

## Truth Table Fill-In

Now we can calculate x, y, and F2:

<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>1</td>
<td>1</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>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0011</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0101</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0110</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0111</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1001</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1010</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1011</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1100</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1101</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1110</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1111</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

ECE/CS 352 Digital System Fundamentals
Finish Up Entries

Finally we can add $w$ 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>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0010</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0011</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0100</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0101</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0110</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0111</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1000</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1001</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1010</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1011</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1100</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1101</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1110</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1111</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

What Does the Circuit Do?

By inspection, we notice that 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 made an assignment to the "Don't Cares" to simplify the logic. The example here did not.

This can be seen by inspecting the two Karnaugh Maps for the function w:

Ex. 3-2 in book (used dc's)

Example here (no dc's)

Logic Design: Functional Blocks

Review

Analysis: From a design to a specification of the behavior.
  - Logic diagram to equations.
  - Logic diagram to function table.
  - "Word description" of circuit operation

Synthesis: From a specification to design implementation.
  1. Define the problem.
  2. Generate function table or equations.
  3. Minimize the Boolean function.
  4. Implement the network.
Review Combinatorial Logic

A combinatorial logic circuit has:
1. A set of $m$ Boolean inputs,
2. A set of $n$ Boolean outputs
3. 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:
1. Identify the inputs and outputs.
2. Derive truth table.
3. Obtain simplified Boolean equations.
4. Draw the logic diagram
5. Check your work to verify correctness.

We will study the design of a code converter to see these steps.
Code Converter Design

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: BCD to Excess 3

<table>
<thead>
<tr>
<th>Function table:</th>
<th>Input BCD</th>
<th>Output Excess-3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A B C D</td>
<td>w x y z</td>
</tr>
<tr>
<td>0 0 0 0</td>
<td>0 0 1 1</td>
<td></td>
</tr>
<tr>
<td>0 0 0 1</td>
<td>0 1 0 0</td>
<td></td>
</tr>
<tr>
<td>0 0 1 0</td>
<td>0 1 0 1</td>
<td></td>
</tr>
<tr>
<td>0 0 1 1</td>
<td>0 1 1 0</td>
<td></td>
</tr>
<tr>
<td>0 1 0 0</td>
<td>0 1 1 1</td>
<td></td>
</tr>
<tr>
<td>0 1 0 1</td>
<td>1 0 0 0</td>
<td></td>
</tr>
<tr>
<td>0 1 1 0</td>
<td>1 0 0 1</td>
<td></td>
</tr>
<tr>
<td>0 1 1 1</td>
<td>1 0 1 0</td>
<td></td>
</tr>
<tr>
<td>1 0 0 0</td>
<td>1 0 1 1</td>
<td></td>
</tr>
<tr>
<td>1 0 0 1</td>
<td>1 0 1 1</td>
<td></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

Minimize each equation

<table>
<thead>
<tr>
<th></th>
<th>C</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>B</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>D</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Minimized Equations:

\[ z = D' \]
\[ y = C \cdot D + C' \cdot D' \] (exnor)
\[ x = B' \cdot C + B' \cdot D + B \cdot C' \cdot D' \]
\[ w = A + B \cdot C + B \cdot D \]

BCD to Excess 3 Implementation

First we will manipulate the equations to expose some shared terms:

\[ z = D' \]
\[ y = C \cdot D + C' \cdot D' \] (This is an EXNOR)
\[ x = B' \cdot C + B' \cdot D + B \cdot C' \cdot D' \]
\[ = B' \cdot (C + D) + B \cdot C' \cdot D' \]
\[ w = A + B \cdot C + B \cdot D \]
\[ = A + B \cdot (C + D) \]

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.
Another Approach: Note that Excess-3 is defined as "BCD plus 3". Thus we can use the simple algorithm shown below to compute Excess-3 from BCD.

```
A  B  C  D
+  0  0  1  1
```

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