Chapter 5
Logic Design with MSI Components and Programmable Logic Devices

The complexity of a chip

Scale of integration:
- SSI 1 - 10 gates
- MSI 10 - 100 gates
- LSI 100 - 1000 gates
- VLSI > 1000 gates

Specialized MSI components
- adders
- comparators
- encoders/decoders
- multiplexers/demultiplexers

Half Adder

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>sum</th>
<th>carry</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

sum = x'y + xy'
carry = xy

Full Adder

<table>
<thead>
<tr>
<th>x_i</th>
<th>y_i</th>
<th>c_i</th>
<th>c_i+1</th>
<th>s_i</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

The Karnaugh maps for a full adder
**SUM and CARRY functions**

SUM = x'y'c + x'yc' + xy'c' + xyc  
CARRY = xy + yc + cx

---

**A realization of the binary full adder**

---

**Parallel (ripple) binary adder**

Designed to add two binary numbers bit by bit

---

**Parallel binary subtracter constructed by using a parallel binary adder**

---

**Parallel binary adder/subtracter**

---

**Carry-look-ahead adder**

- Problem: the time required to do addition is proportional to the number of bits involved.
- Solution: compute the carry for each stage independently by using a carry-look-ahead network.
Carry-Look-ahead Adder

Recall that
\[ c_{i+1} = x_i y_i + x_i c_i + y_i c_i = x_i y_i + (x_i + y_i) c_i \]

Let \( g_i = x_i y_i \) be the carry-generate function, and
\( p_i = x_i + y_i \) be the carry-propagate function.

Then we can write
\[ c_{i+1} = g_i + p_i c_i \]
and
\[
\begin{align*}
    c_1 &= g_0 + p_0 c_0 \\
    c_2 &= g_1 + p_1 g_0 + p_1 p_0 c_0 \\
    c_3 &= g_2 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 c_0 \\
    &\quad \vdots
\end{align*}
\]

We see that all carry signal \( c_i \) can be computed by a two level logic circuit.
From the truth table (Table 5.4) on page 247 we obtained

\[ G_{i+1} = A_iB'_i + A_iG + B'_iG_i \]
\[ E_{i+1} = A'_iB'_iE_i + A_iB_iE_i \]
\[ L_{i+1} = A'_iB_i + B'_iL_i + A'_iL_i \]
Main function of encoder and decoder
The purpose is to reduce the number of wires required for interconnection.

An 8-to-3-line encoder
(a) Logic diagram.
(b) Truth table.
(c) Symbol

A 3-to-8-line decoder
(a) Logic diagram.
(b) Truth table.
(c) Symbol

Decoder realization of \( f_1(x_2, x_1, x_0) = \Sigma m(1,2,4,5) \) and \( f_2(x_2, x_1, x_0) = \Sigma m(1,5,7) \)
Decoder realization of Boolean functions

\[ f_1 = \Pi M(2, 7) \text{ and } f_2 = \Pi M(0, 5, 7) \]

A decoder realization of \( f_1(x_2, x_1, x_0) = \Pi M(0, 1, 3, 5) \) and \( f_2(x_2, x_1, x_0) = \Pi M(1, 3, 6, 7) \) (a) Using output or-gates. (b) Using output nor-gates.

A 3-to-8-line decoder using nand-gates

Realization of the pair of maxterm canonical expressions
\[ f_1(x_2, x_1, x_0) = \Sigma m(0, 2, 6, 7) \text{ and } f_2(x_2, x_1, x_0) = \Sigma m(3, 5, 6, 7) \] (a) Using output and-gates. (b) Using output nand-gates.

Realization of the Boolean expressions
\[ f_1(x_2, x_1, x_0) = \Pi M(0, 1, 3, 4, 7) \] with a 3-to-8-line decoder and two nand-gates.
A 4-to-16-line decoder constructed from 2-to-4-line decoder

The purpose is to reduce the number of wires required for interconnection by making the signals to time-share the link.
### MUX implementation of a Boolean function

- Any Boolean function of $n$ variables can be implemented by a multiplexer with $n$ control inputs in a straightforward manner.

### Example: $f(x, y, z) = \sum m(2, 5, 6, 7)$

<table>
<thead>
<tr>
<th>$x$</th>
<th>$y$</th>
<th>$z$</th>
<th>$f(x, y, z)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

### Implementing a function of 3 variables with a 4x1 MUX: Method 1

- Even better, any Boolean function of $n$ variables can be implemented by a multiplexer with $n-1$ control inputs as illustrated in the following.
Using a multiplexer to implement a Boolean function: Method 1

Note that the output of a 4x1 multiplexer is
\[ F(x, y, z) = x'y'I_0 + x'y'I_1 + xy'I_2 + xy'I_3 \]

Now, given a Boolean function
\[ f(x, y, z) = f(0, 0, 0)x'y'z' + f(0, 1, 0)x'yz' + f(1, 0, 0)xy'z' + f(1, 1, 0)xyz' + f(0, 0, 1)x'y'z + f(0, 1, 1)x'yz + f(1, 0, 1)xy'z + f(1, 1, 1)xyz \]

The value for input \( I_0 \) is to be determined as follows.
if \( f(0, 0, 0) = 0 \) and \( f(0, 0, 1) = 0 \) then \( f(0, 0, 0)x'y'z' + f(0, 0, 1)x'y'z = 0 \) and thus we should let \( I_0 = 0 \)

The value for \( I_1, I_2, \) and \( I_3 \) are to be determined in a similar manner.

Using a multiplexer to implement a Boolean function: Method 2

Note that the output of a 4x1 multiplexer is
\[ F(x, y, z) = I_0y'z' + I_1y'z + I_2yz' + I_3yz \]

Now, given a Boolean function
\[ f(x, y, z) = f(0, 0, 0)x'y'z' + f(0, 0, 1)x'y'z + f(0, 1, 0)x'yz' + f(0, 1, 1)x'yz + f(1, 0, 0)xy'z' + f(1, 0, 1)xy'z + f(1, 1, 0)xyz' + f(1, 1, 1)xyz \]

The value for input \( I_0 \) is to be determined as follows.
if \( f(0, 0, 0) = 0 \) and \( f(0, 0, 1) = 0 \) then \( f(0, 0, 0)x'y'z' + f(0, 0, 1)x'y'z = 0 \) and thus we should let \( I_0 = 0 \)

The value for \( I_1, I_2, \) and \( I_3 \) are to be determined in a similar manner.
Realization of a three-variable function using a 8-to-1-line multiplexer.
(a) Three-variable truth table.
(b) General realization.

Example: realization of \( f(x, y, z) = \Sigma m(0, 2, 3, 5) \)

Realizing a 3-variable Boolean function with a 4-to-1 multiplexer

Obtaining multiplexer realizations using Karnaugh maps.
(a) Cell groupings corresponding to the data line functions.
(b) Karnaugh maps for the \( I \) subfunctions.

Realization of \( f(x, y, z) = \Sigma m(0, 2, 3, 5) \) using a 4-to-1-line multiplexer

Realization of \( f(x, y, z) = \Sigma m(0, 2, 3, 5) \).
(a) Karnaugh map.
(b) \( I_0, I_1, I_2, \) and \( I_3 \) submaps.
Using Karnaugh maps to obtain multiplexer realizations under various assignments to the select inputs.

(a) Applying input variables \(y\) and \(z\) to the \(S_1\) and \(S_0\) select lines.

(b) Applying input variables \(x\) and \(y\) to the \(S_0\) and \(S_1\) select lines.

Alternative realizations of \(f(x,y,z) = \Sigma(0,2,3,5)\).

A select line assignment and corresponding data line functions for a multiplexer realization of a four-variable function.

Realizations of \(f(w,x,y,z) = \Sigma(0,1,5,6,7,9,12,15)\).

Using a four-variable Karnaugh map to obtain a Boolean function realization with a 4-to-1-line multiplexer.

Realizations of the Boolean function \(f(w,x,y,z) = \Sigma(0,1,5,6,7,9,13,14)\).
Types of PLDs

<table>
<thead>
<tr>
<th>Device</th>
<th>AND-array</th>
<th>OR-array</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROM</td>
<td>Fixed</td>
<td>Programmable</td>
</tr>
<tr>
<td>PLA</td>
<td>Programmable</td>
<td>Programmable</td>
</tr>
<tr>
<td>PAL</td>
<td>Programmable</td>
<td>Fixed</td>
</tr>
</tbody>
</table>

Programming by blowing fuses.
(a) Before programming.
(b) After programming.

Structure of a PROM
A $2^n \times m$ PROM.
(a) Logic diagram.
(b) Representation in PLD notation.

Using a PROM for logic design. (a) Truth table. (b) PROM realization.

Example of combinational logic design using a PLA. (a) Maps showing the multiple-output prime implicants. (b) Partial covering of the $f_1$ and $f_2$ maps. (c) Maps for the multiple-output minimal sum. (d) Realization using a $3 \times 4 \times 2$ PLA.

Logic diagram of an $n \times p \times m$ PLA

Example of combinational logic design using a PLA. (a) Maps showing the multiple-output prime implicants. (b) A multiple-output minimal sum covering. (c) Alternative multiple-output minimal sum covering. (d) Realization using a $3 \times 4 \times 2$ PLA.

Exclusive-or-gate with a programmable fuse.
(a) Circuit diagram. (b) Symbolic representation.
General structure of a PLA having true and complemented output capability

Karnaugh maps for the functions $f_1(x,y,z) = \Sigma m(1,2,3,7)$ and $f_2(x,y,z) = \Sigma m(0,1,2,6)$

Two realizations of $f_1(x,y,z) = \Sigma m(1,2,3,7)$ and $f_2(x,y,z) = \Sigma m(0,1,2,6)$.

A simple four-input, three-output PAL device.

An example of using a PAL device to realize two Boolean functions. (a) Karnaugh maps. (b) Realization.

A PLD programming unit
Limitations of PLAs and PALs

These chips are limited to fairly modest size, typically supporting a combined number of inputs plus outputs of not more than 32.

Complex Programmable Logic Devices (CPLDs)

A CPLD comprises multiple PAL-like blocks on a single chip with internal wiring resources to connect the circuit blocks.

It is made to implement complex circuits that cannot be done on a PAL or PLA.
**A Measure of Circuit Size**

A commonly used measure is the total number of two-input NAND gates that would be needed to build the circuit.

It is called the *number of equivalent gates*.

---

**Field-Programmable Gate Arrays (FPGAs)**

An FPGA is a PLD that supports implementation of large logic circuits.

It is different from others in that it does not contain AND or OR planes. Instead, it contains logic blocks as depicted in the next slide.

---

**Typical FPGAs**

FPGAs can be used to implement logic circuits of more than a few hundred thousand equivalent gates in size.

The most commonly used logic block is a *lookup table (LUT)* as depicted in Fig. 3.36.

---

**A two-input lookup table**

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

---

**A three-input LUT**

<table>
<thead>
<tr>
<th>x_1</th>
<th>x_2</th>
<th>x_3</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
A section of a programmed FPGA