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

\[ \text{sum} = x'y + xy' \]

\[ \text{carry} = xy \]

<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>
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>0</td>
<td>1</td>
<td>0</td>
<td>1</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>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>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</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/subtractor
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 \textit{carry-generate} function, and \( p_i = x_i + y_i \) be the \textit{carry-propagate} function.

Then we can write \( c_{i+1} = g_i + p_i c_i \) and
\[
\begin{align*}
  c_1 &= g_0 + p_0 c_0 \\
  c_2 &= g_1 + p_1 g_0 + p_1 p_0 c_0 \\
  c_3 &= g_2 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 c_0 \\
  \vdots \\
  \text{We see that all carry signal } c_i \text{ can be computed by a two level logic circuit.}
\end{align*}
\]
A carry lookahead adder. (a) General organization. (b) Sigma block
A 4-bit carry lookahead adder
Cascade connection of 4-bit carry lookahead adders
A 4-bit carry look-ahead generator
A 16-bit high-speed adder
Organization of a single-decade decimal adder

Two input decimal digits

$A_3 \ A_2 \ A_1 \ A_0 \ B_3 \ B_2 \ B_1 \ B_0$

Single decade decimal adder

Sum decimal digit

$Z_3 \ Z_2 \ Z_1 \ Z_0$

$C_{out}$ (Carry to next decade)

$C_{in}$ (Carry from previous decade)
Organization of a single-decade BCD adder
A single-decade BCD adder
Organization of a 1-bit comparator

(A_i A_{i-1} \cdots A_1 A_0 > B_i B_{i-1} \cdots B_1 B_0)

(A_i A_{i-1} \cdots A_1 A_0 = B_i B_{i-1} \cdots B_1 B_0)

(A_i A_{i-1} \cdots A_1 A_0 < B_i B_{i-1} \cdots B_1 B_0)

1-bit comparator

G_{i+1}

E_{i+1}

L_{i+1}

G_i

E_i

L_i

(A_{i-1} \cdots A_1 A_0 > B_{i-1} \cdots B_1 B_0)

(A_{i-1} \cdots A_1 A_0 = B_{i-1} \cdots B_1 B_0)

(A_{i-1} \cdots A_1 A_0 < B_{i-1} \cdots B_1 B_0)
From the truth table (Table 5.4) on page 247 we obtained

\[ G_{i+1} = A_iB'_i + A_iG_i + B'_iG_i \]

\[ E_{i+1} = A'_iB'_iE_i + A_iB_iE_i \]

\[ L_{i+1} = A'_iB_i + B_iL_i + A'_iL_i \]
Comparing two binary numbers $A$ and $B$. (a) 1-bit comparator network. (b) Cascade connection of 1-bit comparators.
Main function of encoder and decoder

The purpose is to reduce the number of wires required for interconnection.
A $2^n$-to-$n$-line encoder symbol
An 8-to-3-line encoder
Symbol for an $n$-to-$2^n$-line decoder
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

\[ f_1 = \Pi M(2, 7) \] 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

\[
\begin{align*}
    z_0 &= \overline{x_3}x_2x_1x_0 = x_2 + x_1 + x_0 = M_0 \\
    z_1 &= \overline{x_3}x_1x_0 = x_2 + x_1 + \overline{x_0} = M_1 \\
    z_2 &= \overline{x_2}x_1x_0 = x_2 + \overline{x_1} + x_0 = M_2 \\
    z_3 &= \overline{x_2}x_1x_0 = x_2 + \overline{x_1} + \overline{x_0} = M_3 \\
    z_4 &= \overline{x_2}x_1x_0 = \overline{x_2} + x_1 + x_0 = M_4 \\
    z_5 &= \overline{x_2}x_1x_0 = \overline{x_2} + x_1 + \overline{x_0} = M_5 \\
    z_6 &= \overline{x_2}x_1x_0 = \overline{x_2} + \overline{x_1} + x_0 = M_6 \\
    z_7 &= \overline{x_2}x_1x_0 = \overline{x_2} + \overline{x_1} + \overline{x_0} = M_7 
\end{align*}
\]
Realization of the pair of maxterm canonical expressions \( f_1(x_2, x_1, x_0) = \Pi M(0, 3, 5) \) and \( f_2(x_2, x_1, x_0) = \Pi M(2, 3, 4) \) with a 3-to-8-line decoder and two and-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 decoder realization of $f_1(x_2,x_1,x_0) = \Sigma m(0,2,6,7)$ 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.
And-gate 2-to-4-line decoder with an enable input. (a) Logic diagram. (b) Compressed truth table. (c) Symbol.
Nand-gate 2-to-4-line decoder with an enable input

![Nand-gate 2-to-4-line decoder with an enable input](image)

\[
\begin{align*}
z_0 &= \overline{x_1x_0E} \\
   &= x_1 + x_0 + \overline{E} \\
\end{align*}
\]

\[
\begin{align*}
z_1 &= \overline{x_1x_0E} \\
   &= x_1 + \overline{x_0} + \overline{E} \\
\end{align*}
\]

\[
\begin{align*}
z_2 &= \overline{x_1x_0E} \\
   &= \overline{x_1} + x_0 + \overline{E} \\
\end{align*}
\]

\[
\begin{align*}
z_3 &= \overline{x_1x_0E} \\
   &= \overline{x_1} + \overline{x_0} + \overline{E} \\
\end{align*}
\]

<table>
<thead>
<tr>
<th>Inputs $E \ x_1 \ x_0$</th>
<th>Outputs $z_0 \ z_1 \ z_2 \ z_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0</td>
<td>0 1 1 1</td>
</tr>
<tr>
<td>0 0 1</td>
<td>1 0 1 1</td>
</tr>
<tr>
<td>0 1 0</td>
<td>1 1 0 1</td>
</tr>
<tr>
<td>0 1 1</td>
<td>1 1 1 0</td>
</tr>
<tr>
<td>1 × ×</td>
<td>1 1 1 1</td>
</tr>
</tbody>
</table>

(b)
A 4-to-16-line decoder constructed from 2-to-4-line decoder
Main function of multiplexer and demultiplexer

The purpose is to reduce the number of wires required for interconnection by making the signals to time-share the link.
A multiplexer/demultiplexer arrangement for information transmission.
A 4-to-1-line multiplexer
Demultiplexer

Select lines
\[ \begin{align*}
  x_0 & \quad 0 \\
  x_1 & \quad 1
\end{align*} \]

Data input line
\[ E \]

2-to-4 DEC

Outputs
\[ \begin{align*}
  z_0 &= \bar{x}_1 \bar{x}_0 E \\
  z_1 &= \bar{x}_1 x_0 E \\
  z_2 &= x_1 \bar{x}_0 E \\
  z_3 &= x_1 x_0 E
\end{align*} \]
A $2^n$-to-1-line multiplexer symbol

Data input lines

Enable line

Select input lines

Output line
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: \( f(x, y, z) = \Sigma m(2, 5, 6, 7) \)

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>z</th>
<th>f(x, y, z)</th>
<th>=</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>f(0, 0, 0)</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>f(0, 0, 1)</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>f(0, 1, 0)</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>f(0, 1, 1)</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>f(1, 0, 0)</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>f(1, 0, 1)</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>f(1, 1, 0)</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>f(1, 1, 1)</td>
<td>1</td>
</tr>
</tbody>
</table>
MUX implementation of a Boolean function

- Even better, any Boolean function of $n$ variables can be implemented by a multiplexer with $n-1$ control inputs as illustrated in the following.
Implementing a function of 3 variables with a 4x1 MUX: Method 1
Using a multiplexer to implement a Boolean function: Method 1

Note that the output of a 4x1 multiplexer is

\[ F(x, y, z) = x'y'I_0 + x'yI_1 + xy'I_2 + xyI_3 \]

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.

<table>
<thead>
<tr>
<th>$f(0, 0, 0)$</th>
<th>$f(0, 0, 1)$</th>
<th>$f(0, 0, 0)x'y'z' + f(0, 0, 1)x'y'z = 0$</th>
<th>and thus we should let $I_0 =$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>$0 = x'y'0$</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>$x'y'z$</td>
<td>$z$</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>$x'y'z'$</td>
<td>$z'$</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>$x'y' = x'y'1$</td>
<td>1</td>
</tr>
</tbody>
</table>

The value for $I_1$, $I_2$, and $I_3$ are to be determined in a similar manner.
Implementing a function of 3 variables with a 4x1 MUX: Method 2
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

\[ f(x, y, z) = f(0, 0, 0)x'y'z' + f(0, 0, 1)x'y'z + f(0, 1, 0)x'yz' + f(0, 1, 1)x'y z + f(1, 0, 0)xy'z' + f(1, 0, 1)xy'z + f(1, 1, 0)xyz' + f(1, 1, 1)xyz \]
The value for input $I_0$ is to be determined as follows.

<table>
<thead>
<tr>
<th>if $f(0, 0, 0) =$</th>
<th>and $f(0, 0, 1) =$</th>
<th>then $f(0, 0, 0)x'y'z' + f(1, 0, 0)xy'z' =$</th>
<th>and thus we should let $I_0 =$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0</td>
<td>0 0 1 1</td>
<td>$0=0y'z'$</td>
<td>0 0 1 0</td>
</tr>
<tr>
<td>0 1 0 1</td>
<td>1 1 1 1</td>
<td>$xy'z'$</td>
<td>0 0 1 1</td>
</tr>
<tr>
<td>1 0 0 0</td>
<td>0 1 0 1</td>
<td>$x'y'z'$</td>
<td>0 0 1 1</td>
</tr>
<tr>
<td>1 1 0 1</td>
<td>1 0 0 0</td>
<td>$y'z'=1y'z'$</td>
<td>0 0 1 1</td>
</tr>
</tbody>
</table>

The value for $I_1$, $I_2$, and $I_3$ are to be determined in a similar manner.
A multiplexer tree to form a 16-to-1-line multiplexer
Realization of a three-variable function using a 8-to-1-line multiplexer.

(a) Three-variable truth table.

(b) General realization.

\[
\begin{array}{ccc|c}
  x & y & z & f \\
  0 & 0 & 0 & f_0 \\
  0 & 0 & 1 & f_1 \\
  0 & 1 & 0 & f_2 \\
  0 & 1 & 1 & f_3 \\
  1 & 0 & 0 & f_4 \\
  1 & 0 & 1 & f_5 \\
  1 & 1 & 0 & f_6 \\
  1 & 1 & 1 & f_7 \\
\end{array}
\]
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

\[ f_0 \cdot \bar{z} + f_1 \cdot z \]
\[ f_2 \cdot \bar{z} + f_3 \cdot z \]
\[ f_4 \cdot \bar{z} + f_5 \cdot z \]
\[ f_6 \cdot \bar{z} + f_7 \cdot z \]

4-to-1 MUX

\[ I_0 \]
\[ I_1 \]
\[ I_2 \]
\[ I_3 \]

1

\[ E \]

\[ S_1 \]
\[ S_0 \]

\[ x \]
\[ y \]

\[ f \]
Realization of $f(x,y,z) = \Sigma m(0,2,3,5)$ using a 4-to-1-line multiplexer.
Obtaining multiplexer realizations using Karnaugh maps. 
(a) Cell groupings corresponding to the data line functions. 
(b) Karnaugh maps for the $I_i$ subfunctions.
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 \( f(x, y, z) = \Sigma m(0, 2, 3, 5) \).
A select line assignment and corresponding data line functions for a multiplexer realization of a four-variable function.
Realizations of $f(w, x, y, z) = \Sigma m(0, 1, 5, 6, 7, 9, 12, 15)$. 
Using a four-variable Karnaugh map to obtain a Boolean function realization with a 4-to-1-line multiplexer.
Realizations of the Boolean function

\[ f(w, x, y, z) = \Sigma m(0, 1, 5, 6, 7, 9, 13, 14). \]
General structure of Programmable Logic Devices (PLDs)

$n$ buffer/inverters

$n$ input lines

$p$ product-term lines

And array

$m$ output lines

Or array
Buffer/inverter. (a) Symbol. (b) Logic equivalent
## 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.
PLD notation

(a) \[ a \quad b \quad c \]

(b) \[ a \quad b \quad c \]

(c) \[ a \quad b \quad c \]

(d) \[ a \quad b \quad c \]

(e) \[ a \quad b \quad c \]

(f) \[ a \quad b \quad c \]

(g) \[ a \quad b \quad c \]

(h) \[ a \quad b \quad c \]
Structure of a PROM

- \( n \) input (address) lines
- \( n \)-to-\( 2^n \) DEC
- (and-array with buffer/inverters)
- \( 2^n - 1 \) lines

- \( 2^n \) word lines

- Programmable or-array (memory array)

- \( m \) output (bit) lines
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.
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) 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.
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 $f_2(x,y,z) = \Sigma m(0,1,2,6)$

- $f_1 = \bar{x}z + \bar{x}y + yz$
- $f_2 = \bar{x}y + y\bar{z}$

- $\bar{f}_1 = x\bar{y} + x\bar{z} + y\bar{z}$
- $\bar{f}_2 = x\bar{y} + yz$
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
A PLCC package with socket
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.
Structure of a CPLD
A section of a CPLD

PAL-like block (details not shown)
CPLD packaging and programming

(a) CPLD in a Quad Flat Pack (QFP) package

(b) JTAG programming
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.
Structure of an FPGA

- Logic block
- Interconnection switches

I/O block
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
A three-input LUT
A section of a programmed FPGA