§9 Multiplexers, Decoders, and Programmable Logic Devices

Primary Structure: Determined by the number, type, and order of amino acids, and forms a polypeptide.

Secondary Structure: Polypeptide can form a helix-like sheet or an alpha helix; hydrogen bonds form between carbonyl and amino acids.

Tertiary Structure: Omega chains fold into regions, determining the globin chains, and helix (or sheet) is folded in specific ways, forming globin-like proteins characteristic of globular proteins.

Quaternary Structure: Helical and globular proteins may join to form a single protein consisting of several subunits. For example, the oxygen-carrying protein hemoglobin possesses two alpha chains and two beta chains.

HEMOGLOBIN
- Globin chains
- Pair of oxygen molecules
- Normal blood: The oxygen-receptor hemoglobin molecules give blood its red color.

OXYHEMOGLOBIN
- Hemoglobin molecules with iron atoms
- Pair of oxygen molecules
- In iron-deficiency anemia, the number of red blood cells is low; these cells are pale due to the lack of hemoglobin, and they carry less oxygen than normal red blood cells.

www.miracleofthebloodandheart.com
Outline

- Introduction
- Multiplexers
- Three-state buffers
- Decoders and encoders
- Read-only memories
- Programmable logic devices
- Decomposition of switching functions

Introduction

- Building blocks of logic design
  - Gates
  - More complex integrated circuits
    - Multiplexers, decoders, encoders, 3-state buffers, ROMs, PLAs, PAL, CPLDs, FPGAs, etc.
- Integrated circuits
  - Small-scale integration (SSI)
    - Roughly 1~10 gates
    - NAND, NOR, AND, OR, inverters, flip-flops
  - Medium-scale integration (MSI)
    - Roughly 10~100 gates
    - Adders, multiplexers, decoders, registers, counters
  - Large-scale integration (LSI)
    - Roughly 100~1000 gates
    - Memories, microprocessors
  - Very-large-scale integration (VLSI)
    - Roughly 1000+ gates
Multiplexers

- Multiplexer (data selector, MUX)
  - Select one of the data inputs and connect it to the output terminal

2-to-1 MUX

I₀ → 0
I₁ → 1

A

Z = A'I₀ + AI₁

I₀ when A = 0
I₁ when A = 1

Z = A'I₀ + AI₁

4-to-1 MUX

I₀ → Z
I₁ → Z
I₂ → Z
I₃ → Z

A

B

Z = A'B'I₀ + A'BI₁ + AB'I₂ + ABI₃

8-to-1 MUX

I₀ → Z
I₁ → Z
I₂ → Z
I₃ → Z
I₄ → Z
I₅ → Z
I₆ → Z
I₇ → Z

A

B

C

Z = A'B'C'I₀ + A'B'C'I₁ + A'BC'I₂ + A'BC'I₃ + AB'C'I₄ + AB'C'I₅ + ABC'I₆ + ABC'I₇

Z = A'B'I₀ + A'BI₁ + AB'I₂ + ABI₃

Z = A'I₀ + AI₁

Z = A'B'C'I₀ + A'B'C'I₁ + A'BC'I₂ + A'BC'I₃ + AB'C'I₄ + AB'C'I₅ + ABC'I₆ + ABC'I₇
Multiplexers

- 4-to-1 MUX realization using 2-to-1 MUXes

\[ Z = A'B'I_0 + A'BI_1 + AB'I_2 + AB'I_3 \]

We will learn later how to realize any Boolean function with MUXes (Shannon expansion)

---

Multiplexers

- 4-to-1 MUX realization using two-level AND-OR logic

\[ Z = A'B'I_0 + A'BI_1 + AB'I_2 + AB'I_3 \]
Multiplexers

- General $2^n$-to-1 MUX

$$Z = \sum_{k=0}^{2^n-1} m_k I_k$$

- Multi-bit data selection

$$Z = \begin{cases} X \text{ when } A = 0 \\ Y \text{ when } A = 1 \end{cases}$$

$X$, $Y$ are multi-bit words.
Buffers

- Logic gates have limited driving capability
  - Gate output can only drive a limited number of other gate inputs without degrading circuit performance
  - **Buffers** can be inserted to increase the driving capability of a gate output

\[ Y = X \]

Buffer

\[ X \rightarrow \text{Buffer} \rightarrow Y \]

Inverter pair

Multi-stage buffer insertion for high fan-out gates
- Amortize capacitive loads
Three-State Buffers

Direct connection of two or more gate outputs together is dangerous

Use of **three-state (tri-state) buffers** permits the connection

\[ C = \begin{cases} A, & \text{when } B = 1 \\ \text{Hi-Z}, & \text{when } B = 0 \end{cases} \]

Three-State Buffers

Four kinds of three-state buffers

<table>
<thead>
<tr>
<th>B</th>
<th>A</th>
<th>C</th>
<th>B</th>
<th>A</th>
<th>C</th>
<th>B</th>
<th>A</th>
<th>C</th>
<th>B</th>
<th>A</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Z</td>
<td>0</td>
<td>0</td>
<td>Z</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Z</td>
<td>0</td>
<td>1</td>
<td>Z</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>Z</td>
<td>1</td>
<td>0</td>
<td>Z</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>Z</td>
<td>1</td>
<td>1</td>
<td>Z</td>
</tr>
</tbody>
</table>
Three-State Buffers

- Data selection using three-state buffers

\[ A \rightarrow C \equiv B \rightarrow D \]

Are the set of three-state buffers functionally complete?

---

Three-State Buffers

- Circuit with two three-state buffers

\begin{align*}
\text{F} & \quad \text{S2} \\
X & \quad X \times X \times X \\
0 & \quad X \times 0 \times 0 \\
1 & \quad X \times 1 \times 1 \\
Z & \quad X \times 0 \times Z \\
\end{align*}
Three-State Buffers

Applications

- Data selection using tri-state buffers (alternative to MUXes)

Example

- Data communication over a shared channel (bus)

Only one of \{EnA, EnB, EnC, EnD\} is active, i.e., they are mutually exclusive

---

Three-State Buffers

Applications

- Integrated circuit with bi-directional input/output pin
  - **Bi-directional I/O pin** can be used as an input pin and as an output pin, but not both at the same time

---
Decoders and Encoders

Decoders

- 3-to-8 line decoder
  - Exactly one output line will be 1 for each combination of input values
  - Truth table:
    | a | b | c | Y0 | Y1 | Y2 | Y3 | Y4 | Y5 | Y6 | Y7 |
    |---|---|---|----|----|----|----|----|----|----|----|
    | 0 | 0 | 0 | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |
    | 0 | 0 | 1 | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  |
    | 0 | 1 | 0 | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  |
    | 0 | 1 | 1 | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  |
    | 1 | 0 | 0 | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  |
    | 1 | 0 | 1 | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  |
    | 1 | 1 | 0 | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 0  |
    | 1 | 1 | 1 | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  |

- 4-to-10 decoder with inverted outputs (indicated by circles)
  - Exactly one output line will be 0 for each combination of input values
  - Logic diagram:
    - Inputs: A, B, C, D
    - Outputs: 0 to 9
    - Truth table:
      | BCD input | Decimal Output |
      | A B C D   | 0 1 2 3 4 5 6 7 8 9 |
      | 0 0 0 0   | 0 1 1 1 1 1 1 1 1 1 1 1 1 |
      | 0 0 0 1   | 1 0 1 1 1 1 1 1 1 1 1 1 1 1 |
      | 0 0 1 0   | 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 |
      | 0 0 1 1   | 1 1 1 0 1 1 1 1 1 1 1 1 1 1 |
      | 0 1 0 0   | 1 1 1 1 0 1 1 1 1 1 1 1 |
      | 0 1 0 1   | 1 1 1 1 1 0 1 1 1 1 1 1 |
      | 0 1 1 0   | 1 1 1 1 1 1 0 1 1 1 1 |
      | 0 1 1 1   | 1 1 1 1 1 1 1 0 1 1 1 |
      | 1 0 0 0   | 1 1 1 1 1 1 1 1 0 1 1 |
      | 1 0 0 1   | 1 1 1 1 1 1 1 1 1 0 1 1 |
      | 1 0 1 0   | 1 1 1 1 1 1 1 1 1 1 0 1 |
      | 1 0 1 1   | 1 1 1 1 1 1 1 1 1 1 1 0 1 |
      | 1 1 0 0   | 1 1 1 1 1 1 1 1 1 1 1 1 0 1 |
      | 1 1 0 1   | 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 |
      | 1 1 1 0   | 1 1 1 1 1 1 1 1 1 1 1 1 1 1 |
      | 1 1 1 1   | 1 1 1 1 1 1 1 1 1 1 1 1 1 1 |

19

Decoders and Encoders

Decoders

- 3-to-8 line decoder
  - Exactly one output line will be 1 for each combination of input values

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>Y0 = a'b'c'</th>
<th>Y1 = a'b'c</th>
<th>Y2 = a'bc'</th>
<th>Y3 = a'bc</th>
<th>Y4 = ab'c'</th>
<th>Y5 = ab'c</th>
<th>Y6 = abc'</th>
<th>Y7 = abc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<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>
<td>0</td>
<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>0</td>
<td>1</td>
<td>0</td>
<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>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<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>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

19

Decoders and Encoders

Decoders

- 4-to-10 decoder with inverted outputs (indicated by circles)
  - Exactly one output line will be 0 for each combination of input values
  - Logic diagram:
    - Inputs: A, B, C, D
    - Outputs: 0 to 9
    - Truth table:
      | BCD input | Decimal Output |
      | A B C D   | 0 1 2 3 4 5 6 7 8 9 |
      | 0 0 0 0   | 0 1 1 1 1 1 1 1 1 1 |
      | 0 0 0 1   | 1 0 1 1 1 1 1 1 1 1 |
      | 0 0 1 0   | 1 1 0 1 1 1 1 1 1 1 |
      | 0 0 1 1   | 1 1 1 0 1 1 1 1 1 1 |
      | 0 1 0 0   | 1 1 1 1 0 1 1 1 1 1 |
      | 0 1 0 1   | 1 1 1 1 1 0 1 1 1 1 |
      | 0 1 1 0   | 1 1 1 1 1 1 0 1 1 1 |
      | 0 1 1 1   | 1 1 1 1 1 1 1 0 1 1 |
      | 1 0 0 0   | 1 1 1 1 1 1 1 1 0 1 |
      | 1 0 0 1   | 1 1 1 1 1 1 1 1 1 0 1 |
      | 1 0 1 0   | 1 1 1 1 1 1 1 1 1 1 0 1 |
      | 1 0 1 1   | 1 1 1 1 1 1 1 1 1 1 1 0 1 |
      | 1 1 0 0   | 1 1 1 1 1 1 1 1 1 1 1 1 0 1 |
      | 1 1 0 1   | 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 |
      | 1 1 1 0   | 1 1 1 1 1 1 1 1 1 1 1 1 1 1 |
      | 1 1 1 1   | 1 1 1 1 1 1 1 1 1 1 1 1 1 1 |

20
Decoders and Encoders

Decoders

- n-to-2^n line decoder generates all 2^n minterms (or maxterms) of the n input variables
  - The outputs are defined by the equations
    \[ y_i = m_i, \quad \text{for } i = 0, \ldots, 2^n-1 \] (noninverted outputs)
    \[ y_i = m'_i = M_i, \quad \text{for } i = 0, \ldots, 2^n-1 \] (inverted outputs)
  - Can be used for function construction
    - Example
      \[ f_1(a,b,c,d) = m_1 + m_2 + m_4 = (m_1'm_2'm_4')' \]
      \[ f_2(a,b,c,d) = m_4 + m_7 + m_9 = (m_4'm_7'm_9')' \]

Encoders

- An encoder performs the inverse function of a decoder
- 8-to-3 priority encoder
  - If input \( y_i \) is 1 with \( y_j = 0 \) for all \( j > i \), then the outputs \((a,b,c)\) represent a binary number equal to \( i \)
  - Output \( d \) is 1 if any input is 1, otherwise, \( d \) is 0

Note that \( m'_i \) with \( i \geq 10 \) is not available in the 4-to-10 line decoder.

Don't care distinguished by \( d \).
Read-Only Memories

A read-only memory (ROM) stores an array of binary data, which can be read out whenever desired, but cannot be changed under normal operating conditions.

- An output pattern stored in the ROM is called a word.
- An input combination serves as an address which can select one of the words stored in the ROM.

**Block diagram**

```
ROM
8 Words
x 4 Bits
```

**Truth table for ROM**

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>F₀</th>
<th>F₁</th>
<th>F₂</th>
<th>F₃</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
```

- Typical data stored in a ROM.
  (here, \(2^3\) words of 4 bits each)

Read-Only Memories

A ROM which has \(n\) input lines and \(m\) output lines contains an array of \(2^n\) words, and each word is \(m\) bits long.

- The input lines serve as an address to select one of the \(2^n\) words.
- A \((2^n \times m)\) ROM can realize \(m\) functions of \(n\) variables.

**Truth table for ROM**

<table>
<thead>
<tr>
<th>(m) output variables</th>
<th>(n) input variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>100...110</td>
<td>00...00</td>
</tr>
<tr>
<td>010...111</td>
<td>00...01</td>
</tr>
<tr>
<td>101...101</td>
<td>00...10</td>
</tr>
<tr>
<td>110...010</td>
<td>00...11</td>
</tr>
<tr>
<td>...</td>
<td>:</td>
</tr>
<tr>
<td>001...011</td>
<td>11...00</td>
</tr>
<tr>
<td>110...110</td>
<td>11...01</td>
</tr>
<tr>
<td>011...000</td>
<td>11...10</td>
</tr>
<tr>
<td>111...101</td>
<td>11...11</td>
</tr>
</tbody>
</table>

- Typical data array stored in ROM.
  (here, \(2^n\) words of \(m\) bits each)
A ROM consists of a decoder and a memory array.

A ROM is constructed of a decoder and a memory array.

### Decoder

- **n input lines**
- **2^n lines**

### Memory Array

- **2^n Words x m Bits**
- **m output lines**

#### Example

The contents of a ROM are usually specified by a truth table.

<table>
<thead>
<tr>
<th>(m_i)</th>
<th>(F_0)</th>
<th>(F_1)</th>
<th>(F_2)</th>
<th>(F_3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1 0 1 0</td>
<td>1 0 1 0</td>
<td>0 1 1 1</td>
<td>0 1 0 1</td>
</tr>
<tr>
<td>1</td>
<td>1 0 1 0</td>
<td>0 1 1 1</td>
<td>1 1 0 0</td>
<td>1 1 0 0</td>
</tr>
<tr>
<td>2</td>
<td>0 1 1 1</td>
<td>1 1 0 0</td>
<td>0 0 0 1</td>
<td>0 0 0 1</td>
</tr>
<tr>
<td>3</td>
<td>0 1 0 1</td>
<td>0 0 0 1</td>
<td>1 1 1 1</td>
<td>1 1 1 1</td>
</tr>
<tr>
<td>4</td>
<td>1 1 0 0</td>
<td>1 1 1 1</td>
<td>0 1 0 1</td>
<td>0 1 0 1</td>
</tr>
<tr>
<td>5</td>
<td>0 0 0 1</td>
<td>0 1 0 1</td>
<td>1 1 0 0</td>
<td>1 1 0 0</td>
</tr>
<tr>
<td>6</td>
<td>1 1 1 1</td>
<td>1 1 1 1</td>
<td>0 1 0 1</td>
<td>0 1 0 1</td>
</tr>
<tr>
<td>7</td>
<td>0 1 0 1</td>
<td>0 1 0 1</td>
<td>1 1 1 1</td>
<td>1 1 1 1</td>
</tr>
</tbody>
</table>

\[ F_0 = \sum m(0,1,4,6) = A'B'+AC' \]
\[ F_1 = \sum m(2,3,4,6,7) = B+AC' \]
\[ F_2 = \sum m(0,1,2,6) = A'B'+BC' \]
\[ F_3 = \sum m(2,3,5,6,7) = AC+B \]
Read-Only Memories

Basic Structure

- Multiple-output combinational circuits can be realized using ROMs

Example

<table>
<thead>
<tr>
<th>Input WXYZ</th>
<th>Hex Digit</th>
<th>ASCII Code for Hex Digit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0</td>
<td>0 1 1 0 0 0 0</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>0 1 1 0 0 0 1</td>
</tr>
<tr>
<td>0010</td>
<td>2</td>
<td>0 1 1 0 0 1 0</td>
</tr>
<tr>
<td>0011</td>
<td>3</td>
<td>0 1 1 0 0 1 1</td>
</tr>
<tr>
<td>0100</td>
<td>4</td>
<td>0 1 1 0 1 0 0</td>
</tr>
<tr>
<td>0101</td>
<td>5</td>
<td>0 1 1 0 1 0 1</td>
</tr>
<tr>
<td>0110</td>
<td>6</td>
<td>0 1 1 0 1 1 0</td>
</tr>
<tr>
<td>0111</td>
<td>7</td>
<td>0 1 1 1 0 1 1</td>
</tr>
<tr>
<td>1000</td>
<td>8</td>
<td>0 1 1 1 1 0 0</td>
</tr>
<tr>
<td>1001</td>
<td>9</td>
<td>0 1 1 1 1 0 1</td>
</tr>
<tr>
<td>1010</td>
<td>A</td>
<td>1 0 0 0 0 0 0</td>
</tr>
<tr>
<td>1011</td>
<td>B</td>
<td>1 0 0 0 0 0 1</td>
</tr>
<tr>
<td>1100</td>
<td>C</td>
<td>1 0 0 0 0 1 1</td>
</tr>
<tr>
<td>1101</td>
<td>D</td>
<td>1 0 0 0 1 0 0</td>
</tr>
<tr>
<td>1110</td>
<td>E</td>
<td>1 0 0 0 1 0 1</td>
</tr>
<tr>
<td>1111</td>
<td>F</td>
<td>1 0 0 0 1 1 0</td>
</tr>
</tbody>
</table>

Common types of ROMs

- Mask-programmable ROMs
  - At the time of manufacturing, data array is permanently stored in a mask-programmable ROM

- Programmable ROMs (PROMs)
  - Can be programmed once

- Electrically erasable programmable ROM (EEPROMs); flash memories
  - Can be erased and reprogrammed a limited number of times
Programmable Logic Devices

A **programmable logic device (PLD)** is a general name for a digital IC capable of being programmed to provide a variety of different logic functions.

- For a digital system designed using a PLD, changes in the design can easily be made by changing the programming of the PLD without changing the wiring.
- Common types of PLD:
  - Programmable logic arrays (PLAs)
  - Programmable array logic (PAL)
    - A special case of PLAs

Programmable Logic Devices
Programmable Logic Arrays

A **programmable logic array (PLA)** performs the same basic function as a ROM:

- A PLA with n inputs and m outputs can realize m functions of n variables.
- A PLA implements an SOP expression, while a ROM implements a truth table.
- The decoder of a ROM is replaced with an AND array realizing selected product terms of the input variables; the OR array ORs together the product terms.
Programmable Logic Devices
Programmable Logic Arrays

Example

Programmable Logic Devices
Programmable Logic Arrays

Example (Eq. 7-23b)

\[ f_1 = a'bd + abd + ab'c' + b'c \]
\[ f_2 = c + a'bd \]
\[ f_3 = bc + ab'c' + abd \]

Programmable Logic Devices
Programmable Logic Arrays

Example (Eq. 7-23b)

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>A B C</td>
<td>F0 F1 F2 F3</td>
</tr>
<tr>
<td>0 0 1 0</td>
<td>1 0 1 0</td>
</tr>
<tr>
<td>1 1 0 0</td>
<td>1 1 0 0</td>
</tr>
<tr>
<td>0 1 0 1</td>
<td>0 1 0 1</td>
</tr>
<tr>
<td>0 0 1 0</td>
<td>0 0 1 0</td>
</tr>
<tr>
<td>0 0 0 1</td>
<td></td>
</tr>
</tbody>
</table>

Programmable Logic Devices
Programmable Logic Arrays

Example (Eq. 7-23b)

\[ F_0 = A'B' + AC' \]
\[ F_1 = AC' + B \]
\[ F_2 = A'B' + BC' \]
\[ F_3 = B + AC \]
Programmable Logic Devices

Programmable Logic Arrays

- A PLA table for a PLA differs from a truth table for a ROM
  - In a truth table, exactly one row will be selected by each combination of input values
  - In a PLA table, zero, one, or more rows may be selected by each combination of input values (since each row represents a general product term)
    - To determine the value of $f_i$ for a given input combination, the values in the selected rows must be ORed together

- Common types of PLAs
  - Mask-programmable PLAs
  - Field-programmable PLAs (FPLAs)

Programmable Logic Devices

Programmable Array Logic

- The **programmable array logic** (PAL) is a special case of PLA in which the AND array is programmable and the OR array is fixed
  - PAL is less expensive and easier to program

![PAL device](image)

![Standard PAL representation](image)
Programmable Logic Devices
Programmable Array Logic

Example

I₁ → F₁

I₂ → F₈

I₁I₂' + I₁'I₂

Fixed OR gate with fixed input size

Programmable product terms

Unprogrammed PAL segment

Programmed PAL segment

Programmable Logic Devices
Programmable Array Logic

☐ Notation

I₁

I₂

I₁I₂' + I₁'I₂

Non-Inverted Output

Inverted Output

A
B
C

ABC

A
B
C

ABC

A
B
C

ABC
Example

PAL implementation of full adder

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>C_in</th>
<th>Sum</th>
<th>C_out</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>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</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>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

\[
\text{Sum} = X'Y'C_{in} + X'YC_{in}' + XY'C_{in}' + XYC_{in}
\]

\[
\text{Cout} = XC_{in} + YC_{in} + XY
\]
**Other Programmable Logic Devices**

- **Complex programmable logic devices (CPLDs)**
  - §9.7 (skipped)

- **Field programmable gate arrays (FPGAs)**
  - §9.8 (skipped)

**Function Generators**

- **Implementation of a 4-input Lookup Table (LUT)**

<table>
<thead>
<tr>
<th>a b c d</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0</td>
<td>(x_0)</td>
</tr>
<tr>
<td>0 0 0 1</td>
<td>(x_1)</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>1 1 1 1</td>
<td>(x_{15})</td>
</tr>
</tbody>
</table>

- \(x_i \in \{0,1\}\)

- **Diagram:**
  - Bits stored in the LUT (enable/disable minterms)
Function Generators

- Another implementation of a Lookup Table (LUT)

<table>
<thead>
<tr>
<th>a b c d</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0</td>
<td>x_0</td>
</tr>
<tr>
<td>0 0 0 1</td>
<td>x_1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>1 1 1 1</td>
<td>x_{15}</td>
</tr>
</tbody>
</table>

a_i \in \{0,1\}

Decomposition of Switching Functions

- To implement a Boolean function with more than 4 inputs using 4-variable function generators, it has to be decomposed into subfunctions, each with \( \leq 4 \) inputs

  - Shannon's expansion theorem provides a decomposition method
Shannon’s expansion theorem

By expanding a function $f(a,b,c,d)$ about variable $a$, the following equality holds:

$$f(a,b,c,d) = a'f(0,b,c,d) + af(1,b,c,d) = a'f_0 + af_1$$

Example

$$f(a,b,c,d) = c'd'+a'b'c+bcd+ac'$$

$$= a'(c'd'+b'c+bcd)+a(c'd'+bcd+c')$$

$$=a'(c'd'+b'c+cd)+a(c'+bd)$$

fewer inputs

Shannon’s expansion using K-map

\[
\begin{array}{cccc}
00 & 01 & 11 & 10 \\
00 & 1 & 1 & 1 & 1 \\
01 & 0 & 0 & 1 & 1 \\
11 & 1 & 1 & 1 & 0 \\
10 & 1 & 0 & 0 & 0 \\
\end{array}
\]

\[
\begin{array}{cccc}
00 & 01 & 11 & 10 \\
00 & 1 & 1 & 1 & 1 \\
01 & 0 & 0 & 1 & 1 \\
11 & 1 & 1 & 1 & 0 \\
10 & 1 & 0 & 0 & 0 \\
\end{array}
\]

\[a=0, f_0=c'd'+b'c+cd\]
\[a=1, f_1=c'+bd\]
Decomposition of Switching Functions

Shannon’s Expansion Theorem

- Shannon’s expansion theorem
  - By expanding a function \( f(a,b,c,d) \) about variable \( a \), the following equality holds:
    \[
    f(x_1, \ldots, x_{i-1}, x_i, x_{i+1}, \ldots, x_n) = x_i' \cdot f(x_1, \ldots, x_{i-1}, 0, x_{i+1}, \ldots, x_n) + x_i \cdot f(x_1, \ldots, x_{i-1}, 1, x_{i+1}, \ldots, x_n)
    \]
  - \( f_0 + f_1 \)
  - fewer inputs

Decomposition of Switching Functions

Multiplexers and Shannon Expansion

- Any 5-variable function can be realized using two 4-variable function generators and a 2-to-1 MUX
  - Apply Shannon’s expansion once:
    \[
    f(a,b,c,d,e) = a' \cdot f(0,b,c,d,e) + a \cdot f(1,b,c,d,e) = a'f_0 + af_1
    \]
Decomposition of Switching Functions
Multiplexers and Shannon Expansion

- Any 6-variable function can be realized using four 4-variable function generators and three 2-to-1 MUXes
  - Apply Shannon's expansion twice:
    \[ G(a,b,c,d,e,f) = a' \cdot G(0,b,c,d,e,f) + a \cdot G(1,b,c,d,e,f) = a'G_0 + aG_1 \]
    \[ G_0 = b' \cdot G(0,0,c,d,e,f) + b \cdot G(0,1,c,d,e,f) = b'G_{00} + bG_{01} \]
    \[ G_1 = b' \cdot G(1,0,c,d,e,f) + b \cdot G(1,1,c,d,e,f) = b'G_{10} + bG_{11} \]
    \[ G(a,b,c,d,e,f) = a'b'G_{00} + a'bG_{01} + ab'G_{10} + abG_{11} \]

Exercises

- Realize \( F = A'B' + AC \) using a 4-to-1 MUX
- Realize \( F = A'B'C' + B'CD + A'BC + BCD' + AC'D' \) using an 8-to-1 MUX
- Realize \( F = A'B'C' + B'CD + A'BC + BCD' + AC'D' \) using a 4-to-1 MUX and 2-input function generators