Digital Logic Design

- Basics
- Combinational Circuits
- Sequential Circuits

Pu-Jen Cheng

Adapted from the slides prepared by S. Dandamudi for the book, Fundamentals of Computer Organization and Design.
Introduction to Digital Logic Basics

- Hardware consists of a few simple building blocks
  - These are called logic gates
    - AND, OR, NOT, ...
    - NAND, NOR, XOR, ...

- Logic gates are built using transistors
  - NOT gate can be implemented by a single transistor
  - AND gate requires 3 transistors

- Transistors are the fundamental devices
  - Pentium consists of 3 million transistors
  - Compaq Alpha consists of 9 million transistors
  - Now we can build chips with more than 100 million transistors
Basic Concepts

- Simple gates
  - AND
  - OR
  - NOT

- Functionality can be expressed by a truth table
  - A truth table lists output for each possible input combination

- Precedence
  - NOT > AND > OR
  - \( F = A \overline{B} + \overline{A} B \)
  - \( = (A (\overline{B})) + ((\overline{A}) B) \)
Basic Concepts (cont.)

- Additional useful gates
  - NAND
  - NOR
  - XOR

- **NAND = AND + NOT**
- **NOR = OR + NOT**
- XOR implements exclusive-OR function
- **NAND and NOR gates require only 2 transistors**
  - AND and OR need 3 transistors!

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</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>

**NAND gate**

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**NOR gate**

<table>
<thead>
<tr>
<th>A</th>
<th>B</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>

**XOR gate**

Logic symbol

Truth table
Basic Concepts (cont.)

- **Number of functions**
  - With $N$ logical variables, we can define $2^N$ functions.
  - Some of them are useful:
    - AND, NAND, NOR, XOR, …
  - Some are not useful:
    - Output is always 1
    - Output is always 0
  - “Number of functions” definition is useful in proving completeness property.
Basic Concepts (cont.)

Complete sets

- A set of gates is complete
  - If we can implement any logical function using only the type of gates in the set
    - You can use as many gates as you want

- Some example complete sets
  - \{AND, OR, NOT\} ➔ Not a minimal complete set
  - \{AND, NOT\}
  - \{OR, NOT\}
  - \{NAND\}
  - \{NOR\}

- Minimal complete set
  - A complete set with no redundant elements.
Basic Concepts (cont.)

- Proving NAND gate is universal
- NAND gate is called *universal gate*
Basic Concepts (cont.)

- Proving NOR gate is universal
- NOR gate is called *universal gate*

![OR gate](image1.png)
![NOT gate](image2.png)
![AND gate](image3.png)
Integration levels

- **SSI (small scale integration)**
  - Introduced in late 1960s
  - 1-10 gates (previous examples)

- **MSI (medium scale integration)**
  - Introduced in late 1960s
  - 10-100 gates

- **LSI (large scale integration)**
  - Introduced in early 1970s
  - 100-10,000 gates

- **VLSI (very large scale integration)**
  - Introduced in late 1970s
  - More than 10,000 gates
Logic Functions

- Logical functions can be expressed in several ways:
  - Truth table
  - Logical expressions
  - Graphical form

Example:
- Majority function
  - Output is 1 whenever majority of inputs is 1
  - We use 3-input majority function
Logic Functions (cont.)

3-input majority function

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</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>1</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>

Logical expression form

\[ F = A \overline{B} + \overline{B} C + A C \]
Logical Equivalence

- All three circuits implement $F = A \cdot B$ function

- (a)
- (b)
- (c)
Logical Equivalence (cont.)

- Proving logical equivalence of two circuits
  - Derive the logical expression for the output of each circuit
  - Show that these two expressions are equivalent
    - Two ways:
      - You can use the truth table method
        - For every combination of inputs, if both expressions yield the same output, they are equivalent
        - Good for logical expressions with small number of variables
      - You can also use algebraic manipulation
        - Need Boolean identities
Derivation of logical expression from a circuit
  - Trace from the input to output
  - Write down intermediate logical expressions along the path
Logical Equivalence (cont.)

- Proving logical equivalence: Truth table method

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>F1 = A B</th>
<th>F3 = (A + B) (A + \overline{B}) (\overline{A} + B)</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>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Name</td>
<td>AND version</td>
<td>OR version</td>
<td></td>
</tr>
<tr>
<td>-----------------</td>
<td>---------------------</td>
<td>-------------------</td>
<td></td>
</tr>
<tr>
<td>Identity</td>
<td>$x \cdot 1 = x$</td>
<td>$x + 0 = x$</td>
<td></td>
</tr>
<tr>
<td>Complement</td>
<td>$x \cdot \overline{x} = 0$</td>
<td>$x + \overline{x} = 1$</td>
<td></td>
</tr>
<tr>
<td>Commutative</td>
<td>$x \cdot y = y \cdot x$</td>
<td>$x + y = y + x$</td>
<td></td>
</tr>
<tr>
<td>Distribution</td>
<td>$x \cdot (y + z) = xy + xz$</td>
<td>$x + (y \cdot z) =$ $(x+y) \cdot (x+z)$</td>
<td></td>
</tr>
<tr>
<td>Idempotent</td>
<td>$x \cdot x = x$</td>
<td>$x + x = x$</td>
<td></td>
</tr>
<tr>
<td>Null</td>
<td>$x \cdot 0 = 0$</td>
<td>$x + 1 = 1$</td>
<td></td>
</tr>
</tbody>
</table>
### Boolean Algebra (cont.)

<table>
<thead>
<tr>
<th>Name</th>
<th>AND version</th>
<th>OR version</th>
</tr>
</thead>
<tbody>
<tr>
<td>Involution</td>
<td>$\overline{x} = x$</td>
<td>---</td>
</tr>
<tr>
<td>Absorption</td>
<td>$x \cdot (x + y) = x$</td>
<td>$x + (x \cdot y) = x$</td>
</tr>
<tr>
<td>Associative</td>
<td>$x \cdot (y \cdot z) = (x \cdot y) \cdot z$</td>
<td>$x + (y + z) = (x + y) + z$</td>
</tr>
<tr>
<td>de Morgan</td>
<td>$\overline{x \cdot y} = \overline{x} + \overline{y}$</td>
<td>$\overline{x + y} = \overline{x} \cdot \overline{y}$</td>
</tr>
</tbody>
</table>
Boolean Algebra (cont.)

- Proving logical equivalence: Boolean algebra method
  - To prove that two logical functions $F_1$ and $F_2$ are equivalent
    - Start with one function and apply Boolean laws to derive the other function
    - Needs intuition as to which laws should be applied and when
      - Practice helps
    - Sometimes it may be convenient to reduce both functions to the same expression
  - Example: $F_1 = A \cdot B$ and $F_3$ are equivalent
    - $A \cdot B = (A + B) (A + \overline{B}) (\overline{A} + B)$
A simple logic design process involves

- Problem specification
- Truth table derivation
- Derivation of logical expression
- Simplification of logical expression
- Implementation
Deriving Logical Expressions

- Derivation of logical expressions from truth tables
  - sum-of-products (SOP) form
  - product-of-sums (POS) form

- SOP form
  - Write an AND term for each input combination that produces a 1 output
    - Write the variable if its value is 1; complement otherwise
  - OR the AND terms to get the final expression

- POS form
  - Dual of the SOP form
Deriving Logical Expressions (cont.)

- 3-input majority function

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</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>1</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>

- SOP logical expression

- Four product terms
  - Because there are 4 rows with a 1 output

\[
F = \overline{A} \overline{B} C + \overline{A} B \overline{C} + A \overline{B} \overline{C} + A B C
\]
3-input majority function

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</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>1</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>

POS logical expression

Four sum terms
- Because there are 4 rows with a 0 output

\[ F = (A + B + C) (A + \overline{B} + C) \]
\[ (A + \overline{B} + C) (\overline{A} + B + C) \]
Logical Expression Simplification

- Two basic methods
  - Algebraic manipulation
    - Use Boolean laws to simplify the expression
      - Difficult to use
      - Don’t know if you have the simplified form
  - Karnaugh map (K-map) method
    - Graphical method
    - Easy to use
      - Can be used to simplify logical expressions with a few variables
Algebraic Manipulation

- **Majority function example**

  \[
  \overline{A} \overline{B} \overline{C} + A \overline{B} \overline{C} + A B \overline{C} + A B C = \\
  A \overline{B} \overline{C} + A \overline{B} C + A B \overline{C} + A B C + A B C + A B C
  \]

- We can now simplify this expression as

  \[
  B C + A C + A B
  \]

- A difficult method to use for complex expressions
Karnaugh Map Method

Note the order

(a) Two-variable K-map  (b) Three-variable K-map  (c) Four-variable K-map
Karnaugh Map Method (cont.)

Simplification examples

(a) Majority function

(b) Even-parity function
Karnaugh Map Method (cont.)

First and last columns/rows are adjacent
Karnaugh Map Method (cont.)

Minimal expression depends on groupings

(a)

(b)
Karnaugh Map Method (cont.)

No redundant groupings

(a) Nonminimal simplification

(b) Minimal simplification
Example

- Seven-segment display
- Need to select the right LEDs to display a digit
## Karnaugh Map Method (cont.)

### Truth table for segment d

<table>
<thead>
<tr>
<th>No</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>Seg.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</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>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>No</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>Seg.</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>?</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>?</td>
</tr>
<tr>
<td>12</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>?</td>
</tr>
<tr>
<td>13</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>?</td>
</tr>
<tr>
<td>14</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>?</td>
</tr>
<tr>
<td>15</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>?</td>
</tr>
</tbody>
</table>

The table above represents the truth table for segment d, showing the input values for A, B, C, D and the corresponding segments for each combination.
Karnaugh Map Method (cont.)

Don’t cares simplify the expression a lot

(a) Simplification with no don’t cares

(b) Simplification with don’t cares
Implementation Using NAND Gates

- Using NAND gates
  - Get an equivalent expression
    \[ A \cdot B + C \cdot D = \overline{A \cdot B + C \cdot D} \]
  - Using de Morgan’s law
    \[ \overline{A \cdot B + C \cdot D} = \overline{A} \cdot \overline{B} \cdot \overline{C} \cdot \overline{D} \]
  - Can be generalized
    - Majority function
      \[ A \cdot B + B \cdot C + A \cdot C = A \cdot B \cdot B \cdot C \cdot A \cdot C \]

Implementation Using NAND Gates (cont.)

- Majority function

A B C

\[ \text{Diagram of Majority function using NAND gates.} \]
Introduction to Combinational Circuits

- Combinational circuits
  - Output depends only on the current inputs
- Combinational circuits provide a higher level of abstraction
  - Help in reducing design complexity
  - Reduce chip count
- We look at some useful combinational circuits
Multiplexers

- Multiplexer
  - $2^n$ data inputs
  - $n$ selection inputs
  - a single output

- Selection input determines the input that should be connected to the output

4-data input MUX

<table>
<thead>
<tr>
<th>S1</th>
<th>S0</th>
<th>O</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>I0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>I1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>I2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>I3</td>
</tr>
</tbody>
</table>
4-data input MUX implementation
Multiplexers (cont.)

MUX implementations

Majority function

Even-parity function
Multiplexers (cont.)

Example chip: 8-to-1 MUX

(a) Connection diagram
(b) Logic symbol
## Efficient implementation: Majority function

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>F₁</th>
<th>A</th>
<th>B</th>
<th>F₁</th>
</tr>
</thead>
<tbody>
<tr>
<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>0</td>
<td>1</td>
<td>C</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>C</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

The diagram illustrates the multiplexer with inputs A, B, and C, and output F₁.
Demultiplexers (DeMUX)

- Demultiplexer
  - a single input
  - $n$ selection inputs
  - $2^n$ outputs
Decoders

- Decoder selects one-out-of-N inputs

<table>
<thead>
<tr>
<th>I₁</th>
<th>I₀</th>
<th>O₁</th>
<th>O₂</th>
<th>O₃</th>
<th>O₀</th>
</tr>
</thead>
<tbody>
<tr>
<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>0</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>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Decoders (cont.)

Logic function implementation

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C&lt;sub&gt;in&lt;/sub&gt;</th>
<th>Sum</th>
<th>C&lt;sub&gt;out&lt;/sub&gt;</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>

![Logic diagram](image-url)
Comparator

- Used to implement comparison operators ($=, >, <, \geq, \leq$)
Comparator (cont.)

4-bit magnitude comparator chip

A=B: $O_x = I_x \ (x = A < B, A = B, & \ A > B)$

(a) Connection diagram
(b) Logic symbol
Comparator (cont.)

Serial construction of an 8-bit comparator
1-bit Comparator

CMP

x>y
x=y
x<y

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>x&gt;y</th>
<th>x=y</th>
<th>x&lt;y</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
8-bit comparator

- $x_n > y_n$ (Input) → $x > y$ (Output)
- $x_n = y_n$ (Input) → $x = y$ (Output)
- $x_n < y_n$ (Input) → $x < y$ (Output)

Diagram:

- Input: $x$ and $y$
- Outputs: $x > y$, $x = y$, $x < y$
Adders

- **Half-adder**
  - Adds two bits
    - Produces a *sum* and *carry*
  - Problem: Cannot use it to build larger inputs

- **Full-adder**
  - Adds three 1-bit values
    - Like half-adder, produces a *sum* and *carry*
  - Allows building N-bit adders
    - Simple technique
      - Connect *Cout* of one adder to *Cin* of the next
    - These are called *ripple-carry adders*
Adders (cont.)

(a) Half-adder truth table and implementation

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Sum</th>
<th>C_{out}</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>

(b) Full-adder truth table and implementation
A 16-bit ripple-carry adder
Adders (cont.)

- Ripple-carry adders can be slow
  - Delay proportional to number of bits

- Carry lookahead adders
  - Eliminate the delay of ripple-carry adders
  - Carry-ins are generated independently
    - $C_0 = A_0 B_0$
    - $C_1 = A_0 B_0 A_1 + A_0 B_0 B_1 + A_1 B_1$
    - . . .
  - Requires complex circuits
  - Usually, a combination carry lookahead and ripple-carry techniques are used
Programmable Logic Arrays

- PLAs
  - Implement sum-of-product expressions
    - No need to simplify the logical expressions
  - Take $N$ inputs and produce $M$ outputs
    - Each input represents a logical variable
    - Each output represents a logical function output
  - Internally uses
    - An AND array
      - Each AND gate receives $2N$ inputs
    - An OR array
Programmable Logic Arrays (cont.)

A blank PLA with 2 inputs and 2 outputs

I₀  ┌───┐
    │    │
    │    │
    │    │
I₁  └───┘

AND array

P₀  P₁  P₂  P₃

OR array

F₀

F₁
Programmable Logic Arrays (cont.)

Implementation examples

\[ F_0 = \overline{A}B + \overline{A}B + \overline{A}\overline{B} \]

\[ F_1 = \overline{A}B + \overline{A}B + AB \]
Simplified notation

\[ F_0 = A \overline{B} + \overline{A} \overline{B} + \overline{A} \overline{B} \]

\[ F_1 = A \overline{B} + \overline{A} \overline{B} + A \overline{B} \]
1-bit Arithmetic and Logic Unit

Preliminary ALU design

<table>
<thead>
<tr>
<th>$F_1$</th>
<th>$F_0$</th>
<th>$F$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>A and B</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>A or B</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>A + B</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>A - B</td>
</tr>
</tbody>
</table>

2’s complement
Required 1 is added via $C_{in}$
1-bit Arithmetic and Logic Unit (cont.)

Final design
16-bit ALU
Arithmetic and Logic Unit (cont’d)

4-bit ALU

(a) Connection diagram

(b) Active low operands

(c) Active high operands
Introduction to Sequential Circuits

- Output depends on current as well as past inputs
  - Depends on the history
  - Have “memory” property

- Sequential circuit consists of
  - Combinational circuit
  - Feedback circuit
  - Past input is encoded into a set of state variables
    - Uses feedback (to feed the state variables)
      - Simple feedback
      - Uses flip flops
Introduction (cont.)

Main components of a sequential circuit

Sequential circuit

Input

Combinational circuit

Feedback circuit

Feedback

Output
Clock Signal

Clock cycle

Rising edge

Falling edge

Time

(a) Symmetric

(b) Smaller ON period

(c) Smaller OFF period
Clock Signal (cont.)

- Clock serves two distinct purposes
  - Synchronization point
    - Start of a cycle
    - End of a cycle
    - Intermediate point at which the clock signal changes levels
  - Timing information
    - Clock period, ON, and OFF periods
- Propagation delay
  - Time required for the output to react to changes in the inputs
(a) Circuit diagram

(b) Timing diagram

Clock Signal (cont.)
SR Latches

- Can remember a bit
- Level-sensitive (not edge-sensitive)

A NOR gate implementation of SR latch

(a) Circuit diagram
(b) Logic symbol
(c) Truth table
SR Latches (cont.)

- SR latch outputs follow inputs
- In clocked SR latch, outputs respond at specific instances
  - Uses a clock signal

(a) Circuit diagram
(b) Logic symbol
D Latches

- D Latch
  - Avoids the SR = 11 state

(a) Circuit diagram
(b) Logic symbol
(c) Truth table

<table>
<thead>
<tr>
<th>D</th>
<th>Q_n+1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Positive Edge-Triggered D Flip-Flops

- Edge-sensitive devices
  - Changes occur either at positive or negative edges

(a) Circuit diagram
(b) Logic symbol
Notation for Latches & Flip-Flops

- Not strictly followed in the literature

Latches

(a) Low level
(b) High level

Flip-flops

(c) Positive edge
(d) Negative edge
Example of Shift Register Using D Flip-Flops

74164 shift Register chip

(a) Connection diagram

(b) Logic diagram
Memory Design Using D Flip-Flops

Require separate data in and out lines
JK Flip-Flops

JK flip-flop  
(master-slave)

<table>
<thead>
<tr>
<th>J</th>
<th>K</th>
<th>$Q_{n+1}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>$Q_n$</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>$Q_n$</td>
</tr>
</tbody>
</table>
Examples of D & JK Flip-Flops

Two example chips

D latches

JK flip-flops

(a) 7477

(b) 7476
Example of Shift Register Using JK Flip-Flops

- Shift Registers
  - Can shift data left or right with each clock pulse

A 4-bit shift register using JK flip-flops
Example of Counter Using JK Flip-Flops

- Counters
  - Easy to build using JK flip-flops
    - Use the JK = 11 to toggle
  - Binary counters
    - Simple design
      - B bits can count from 0 to $2^B - 1$
    - Ripple counter
      - Increased delay as in ripple-carry adders
      - Delay proportional to the number of bits
  - Synchronous counters
    - Output changes more or less simultaneously
    - Additional cost/complexity
Modulo-8 Binary Ripple Counter Using JK Flip-Flops

(a) Circuit diagram

(b) Timing diagram
Synchronous Modulo-8 Counter

- Designed using the following simple rule
  - Change output if the preceding count bits are 1
    - Q1 changes whenever Q0 = 1
    - Q2 changes whenever Q1Q0 = 11
Example Counters

(a) Connection diagram

(b) Logic symbol

(c) State diagram of 74161

(d) State diagram of 74160
Sequential Circuit Design

- Sequential circuit consists of
  - A combinational circuit that produces output
  - A feedback circuit
    - We use JK flip-flops for the feedback circuit
- Simple counter examples using JK flip-flops
  - Provides alternative counter designs
  - We know the output
    - Need to know the input combination that produces this output
    - Use an excitation table
      - Built from the truth table
## Sequential Circuit Design (cont.)

(a) JK flip-flop truth table

<table>
<thead>
<tr>
<th>J</th>
<th>K</th>
<th>Qn</th>
<th>Qn+1</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>1</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>1</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>0</td>
</tr>
</tbody>
</table>

(b) Excitation table for JK flip-flops

<table>
<thead>
<tr>
<th>Qn</th>
<th>Qn+1</th>
<th>J</th>
<th>K</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>d</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>d</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>d</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>d</td>
<td>0</td>
</tr>
</tbody>
</table>
Build a design table that consists of
- Current state output
- Next state output
- JK inputs for each flip-flop

Binary counter example
- 3-bit binary counter
- 3 JK flip-flops are needed
- Current state and next state outputs are 3 bits each
- 3 pairs of JK inputs
### Design table for the binary counter example

<table>
<thead>
<tr>
<th>Present state</th>
<th>Next state</th>
<th>JK flip-flop inputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>A  B  C</td>
<td>A  B  C</td>
<td>J_A  K_A</td>
</tr>
<tr>
<td>0  0  0</td>
<td>0  0  1</td>
<td>0   d</td>
</tr>
<tr>
<td>0  0  1</td>
<td>0  1  0</td>
<td>0   d</td>
</tr>
<tr>
<td>0  1  0</td>
<td>0  1  1</td>
<td>0   d</td>
</tr>
<tr>
<td>0  1  1</td>
<td>1  0  0</td>
<td>1   d</td>
</tr>
<tr>
<td>1  0  0</td>
<td>1  0  1</td>
<td>d   0</td>
</tr>
<tr>
<td>1  0  1</td>
<td>1  1  0</td>
<td>d   0</td>
</tr>
<tr>
<td>1  1  0</td>
<td>1  1  1</td>
<td>d   0</td>
</tr>
<tr>
<td>1  1  1</td>
<td>0  0  0</td>
<td>d   1</td>
</tr>
<tr>
<td>1  1  1</td>
<td>0  0  0</td>
<td>d   1</td>
</tr>
</tbody>
</table>
Use K-maps to simplify expressions for JK inputs.
Final circuit for the binary counter example

- Compare this design with the synchronous counter design
A more general counter design
- Does not step in sequence

0 → 3 → 5 → 7 → 6 → 0

Same design process

One significant change
- Missing states
  - 1, 2, and 4
  - Use don’t cares for these states
## Design Table for the General Counter Example

<table>
<thead>
<tr>
<th>Present state</th>
<th>Next state</th>
<th>JK flip-flop inputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>A  B  C</td>
<td>A  B  C</td>
<td>J_A  K_A</td>
</tr>
<tr>
<td>0  0  0</td>
<td>0  1  1</td>
<td>0   d</td>
</tr>
<tr>
<td>0  0  1</td>
<td>—   —   —</td>
<td>d   d</td>
</tr>
<tr>
<td>0  1  0</td>
<td>—   —   —</td>
<td>d   d</td>
</tr>
<tr>
<td>0  1  1</td>
<td>1  0  1</td>
<td>1   d</td>
</tr>
<tr>
<td>1  0  0</td>
<td>—   —   —</td>
<td>d   d</td>
</tr>
<tr>
<td>1  0  1</td>
<td>1  1  1</td>
<td>d   0</td>
</tr>
<tr>
<td>1  1  0</td>
<td>0  0  0</td>
<td>d   1</td>
</tr>
<tr>
<td>1  1  1</td>
<td>1  1  0</td>
<td>d   0</td>
</tr>
</tbody>
</table>
Sequential Circuit Design (cont.)

K-maps to simplify JK input expressions

$$J_A = B$$

$$J_B = 1$$

$$J_C = \overline{A}$$

$$K_A = \overline{C}$$

$$K_R = \overline{A} + \overline{C}$$

$$K_C = A \cdot B$$
Sequential Circuit Design (cont.)

Final circuit for the general counter example
General Design Process

- FSM can be used to express the behavior of a sequential circuit
  - Counters are a special case
    - State transitions are indicated by arrows with labels X/Y
      - X: inputs that cause system state change
      - Y: output generated while moving to the next state
  - Look at two examples
    - Even-parity checker
    - Pattern recognition
Even-parity checker
- FSM needs to remember one of two facts
  - Number of 1’s is odd or even
  - Need only two states
    - 0 input does not change the state
    - 1 input changes state
- Simple example
  - Complete the design as an exercise
Pattern recognition example

- Outputs 1 whenever the input bit sequence has exactly two 0s in the last three input bits
- FSM requires three special states during the initial phase
  - S0 – S2
- After that we need four states
  - S3: last two bits are 11
  - S4: last two bits are 01
  - S5: last two bits are 10
  - S6: last two bits are 00
General Design Process (cont.)

State diagram for the pattern recognition example
Steps in the design process

1. Derive FSM
2. State assignment
   * Assign flip-flop states to the FSM states
     * Necessary to get an efficient design
3. Design table derivation
   * Derive a design table corresponding to the assignment in the last step
4. Logical expression derivation
   * Use K-maps as in our previous examples
5. Implementation
General Design Process (cont.)

- State assignment
  - Three heuristics
    - Assign adjacent states for:
      - states that have the same next state
      - states that are the next states of the same state
      - States that have the same output for a given input
  - For our example
    - Heuristic 1 groupings: (S1, S3, S5)\(^2\) (S2, S4, S6)\(^2\)
    - Heuristic 2 groupings: (S1, S2) (S3, S4)\(^3\) (S5, S6)\(^3\)
    - Heuristic 1 groupings: (S4, S5)
General Design Process (cont.)

State table for the pattern recognition example

<table>
<thead>
<tr>
<th>Present state</th>
<th>Next state</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$X = 0$</td>
<td>$X = 1$</td>
</tr>
<tr>
<td>$S_0$</td>
<td>$S_2$</td>
<td>$S_1$</td>
</tr>
<tr>
<td>$S_1$</td>
<td>$S_4$</td>
<td>$S_3$</td>
</tr>
<tr>
<td>$S_2$</td>
<td>$S_6$</td>
<td>$S_5$</td>
</tr>
<tr>
<td>$S_3$</td>
<td>$S_4$</td>
<td>$S_3$</td>
</tr>
<tr>
<td>$S_4$</td>
<td>$S_6$</td>
<td>$S_5$</td>
</tr>
<tr>
<td>$S_5$</td>
<td>$S_4$</td>
<td>$S_3$</td>
</tr>
<tr>
<td>$S_6$</td>
<td>$S_6$</td>
<td>$S_5$</td>
</tr>
</tbody>
</table>
General Design Process (cont.)

K-map for state assignment

<table>
<thead>
<tr>
<th>BC</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>S0</td>
</tr>
<tr>
<td>01</td>
<td>S3</td>
</tr>
<tr>
<td>11</td>
<td>S5</td>
</tr>
<tr>
<td>10</td>
<td>S1</td>
</tr>
<tr>
<td>0</td>
<td>S4</td>
</tr>
<tr>
<td>1</td>
<td>S6</td>
</tr>
<tr>
<td>2</td>
<td>S2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>State assignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>State</td>
</tr>
<tr>
<td>--------</td>
</tr>
<tr>
<td>S0</td>
</tr>
<tr>
<td>S1</td>
</tr>
<tr>
<td>S2</td>
</tr>
<tr>
<td>S3</td>
</tr>
<tr>
<td>S4</td>
</tr>
<tr>
<td>S5</td>
</tr>
<tr>
<td>S6</td>
</tr>
</tbody>
</table>
### General Design Process (cont.)

#### Design Table

<table>
<thead>
<tr>
<th>Present state</th>
<th>Present state</th>
<th>Next state</th>
<th>Present state</th>
<th>JK flip-flop inputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>X</td>
<td>A</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</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>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</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>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>1</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>0</td>
<td>1</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>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>
General Design Process (cont.)

K-maps for JK inputs

K-map for the output

\[ Y = \overline{A} B C \overline{X} + A B C X + A \overline{B} \overline{X} \]
General Design Process (cont.)

Final implementation