Lecture 22
Chapters 3
Logic Circuits Part 1
LC-3
Data Path
Revisited

How are the components
Seen here implemented?
Computing Layers

Problems

Algorithms

Language

Instruction Set Architecture

Microarchitecture

Circuits

Devices
Transistor: Building Block of Computers

Logically, each transistor acts as a switch
Combined to implement logic functions (gates)
  • AND, OR, NOT

Combined to build higher-level structures
  • Adder, multiplexer, decoder, register, memory …
  • Adder, multiplier …

Combined to build simple processor
  • LC-3
Simple Switch Circuit

Switch **open**:  
- Open circuit, no current  
- Light is **off**  
- \( V_{out} \) is **+2.9V**

Switch **closed**:  
- Short circuit across switch, current flows  
- Light is **on**  
- \( V_{out} \) is **0V**

Switch-based circuits can easily represent two states: on/off, open/closed, voltage/no voltage.
n-type MOS Transistor

MOS = Metal Oxide Semiconductor

- two types: n-type and p-type

**n-type**

- when Gate has **positive** voltage, short circuit between #1 and #2 (switch **closed**)
- when Gate has **zero** voltage, open circuit between #1 and #2 (switch **open**)

Terminal #2 must be connected to GND (0V).
p-type MOS Transistor

**p-type** is *complementary* to n-type

- when Gate has **positive** voltage, open circuit between #1 and #2 (switch **open**)
- when Gate has **zero** voltage, short circuit between #1 and #2 (switch **closed**)

Terminal #1 must be connected to +2.9V.
Logic Gates

Use switch behavior of MOS transistors to implement logical functions: AND, OR, NOT.

Digital symbols:
- recall that we assign a range of analog voltages to each digital (logic) symbol

- assignment of voltage ranges depends on electrical properties of transistors being used
  - typical values for "1": +5V, +3.3V, +2.9V
  - from now on we'll use +2.9V
CMOS Circuit

Complementary MOS uses both n-type and p-type MOS transistors

- p-type
  - Attached to + voltage (2.9v)
  - Pulls output voltage UP when input is zero
- n-type
  - Attached to GND (0v)
  - Pulls output voltage DOWN when input is one

For all inputs, output is either connected to GND or to +, but not both!

No direct connection between + and GND, except switching. Low power consumption.
**Inverter (NOT Gate)**

**Truth table**

<table>
<thead>
<tr>
<th>In</th>
<th>Out</th>
<th>In</th>
<th>Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 V</td>
<td>2.9 V</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2.9 V</td>
<td>0 V</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**Symbol**

- **P-type**
  - In = 0 → Out = 1
  - In = 1 → Out = 0

- **N-type**
Logical Operation: OR and NOR

Truth tables

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

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>NOR</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>

Inputs: 2 or more

Logic symbols

Output = A + B

Boolean algebra notation

Output = A + B
AND and NAND

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

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

Inputs: 2 or more

Output = A.B

Output = A.B
NOR Gate (OR-NOT)

Note: Serial structure on top, parallel on bottom.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</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>
OR Gate

Truth table

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

Add inverter to NOR.
Basic Logic Gates

**NOT**

**OR**

**NOR**

**AND**

**NAND**
Boolean Algebra

\[ x \cdot 0 = 0 \]
\[ x \cdot 1 = x \]
\[ x \cdot \overline{x} = 0 \]
\[ x + 0 = x \]
\[ x + 1 = \]
\[ x + x = \]

\[ \overline{x} \cdot 0 = 0 \]
\[ \overline{x} \cdot 1 = \overline{x} \]
\[ \overline{x} \cdot \overline{\overline{x}} = 0 \]
\[ \overline{x} + 0 = \overline{x} \]
\[ \overline{x} + 1 = \]
\[ \overline{x} + \overline{x} = \]
Boolean Algebra Laws (2)

Commutative  \( A+B = B+A \)  \( A \cdot B = B \cdot A \)

Associative
- \( A+(B+C)=(A+B)+C = A+B+C \)
- \( A.(B.C)=(A.B).C = ABC \)

Distributive
- \( A.(B+C)=A.B+A.C \)
- \( A+(B.C)=(A+B).(A+C) \)
Some Useful Identities for simplification

**AB+AB = A**

Proof: \( AB+AB = A(B+B) \)  
= \( A \)

**A+AB = A**

Proof: \( A+AB = A(1+B) \)  
= \( A \)
DeMorgan's Law

Converting AND to OR (with some help from NOT)
Consider the following gate:

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<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>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>

Same as A OR B!

To convert AND to OR (or vice versa), invert inputs and output.
More than 2 Inputs?

AND/OR can take any number of inputs.

- AND = 1 if all inputs are 1.
- OR = 1 if any input is 1.
- Similar for NAND/NOR.

Can implement with multiple two-input gates, or with single CMOS circuit.
Propagation Delay

• Each gate has a propagation delay, typically fraction of a nanosecond ($10^{-9}$ sec).
• Delays add depending on the chain of gates the signals have to go through.
• Clock frequency is determined by the delay of the longest combinational path between storage elements. Measured in GHz ($10^9$ cycles per sec).
Summary

MOS transistors are used as switches to implement logic functions.

- n-type: connect to GND, turn on (1) to pull down to 0
- p-type: connect to +2.9V, turn on (0) to pull up to 1

Basic gates: NOT, NOR, NAND

- Boolean Algebra: Logic functions are usually expressed with AND, OR, and NOT

DeMorgan's Law

- Convert AND to OR (and vice versa) by inverting inputs and output
Building Functions from Logic Gates

**Combinational Logic Circuit**
- output depends only on the current inputs
- stateless

**Sequential Logic Circuit**
- output depends on the sequence of inputs (past and present)
- stores information (state) from past inputs

We'll first look at some useful combinational circuits, then show how to use sequential circuits to store information.
Combinatorial Logic

Cascading set of logic gates

Digital circuit

Truth table
Logisim Simulator

- Logic simulator: allows interactive design and layout of circuits with AND, OR, and NOT gates
- Simulator web page (linked on class web page)
  [http://www.cburch.com/logisim](http://www.cburch.com/logisim)
- Overview, tutorial, downloads, etc.
- Windows or Linux operating systems
- Logisim demonstration
Functional Blocks

- Decoder
- Multiplexer
- Full Adder
- Any general function
Decoder

$n$ inputs, $2^n$ outputs

- exactly one output is 1 for each possible input pattern

2-bit decoder
Multiplexer (MUX)

An $n$-bit selector and $2^n$ inputs, one output
- output equals one of the inputs, depending on selector

4-to-1 MUX

A, if $S=00$
B, if $S=01$
C, if $S=10$
D, if $S=11$
Full Adder

Add two bits and carry-in, produce one-bit sum and carry-out.
Four-bit Adder (*ripple carry*)

2 levels of delay per stage
Logical Completeness

Can implement **ANY** truth table with combo of AND, OR, NOT gates.

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

1. AND combinations that yield a "1" in the truth table.

2. OR the results of the AND gates.
Truth Table (to circuit)

How do we design a circuit for this?

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>X</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<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>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>0</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>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>
Programmable Logic Array

Front end is decoder for inputs
Back end defines the outputs
Any truth table can be built
Not necessarily minimal circuit!

Requires (at least) ten gates.
Circuit Minimization using Boolean Algebra

Boolean logic lets us reduce the circuit

- \( X = A'B'C' + A'BC' + ABC' + ABC \)
  \[ = A'C' + AB \]

- \( Y = A'B'C + A'BC + AB'C + ABC \)
  \[ = A'C + AC = C \]

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>X</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<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>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>0</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>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>

Only three gates!

Try with Logisim!
Karnaugh maps to minimize literals

Based on set-theory

• Visual representation of algebraic functions
• Allow algorithmic minimizing of boolean functions in sum-of-products form
• “adjacent” terms can be combined.
  • Adjacent: differ in one variable, complemented in one, not complemented in the other.

Example:

\[ \text{ABC} + \text{ABC}' = \text{AB}(\text{C} + \text{C}') = \text{AB} \]

Thus ABC and ABC’ are two pieces of AB.

Combining Minterms

• For n-variables, there are \(2^n\) minterms, corresponding to each row of truth table.
• Some of them can be combined into groups of 2, (or 4 or 8 ..) to simplify the function.
Karnaugh maps

Visual representation of algebraic functions to make it easy to spot “adjacent” minterms

- Columns arranged so that adjacent terms are visually adjacent.
- Identify groups of 2, 4, 8 etc. terms that can be combined.
- All 1’s must be covered.
- A 1 can be used more than once, if needed.
- Sometimes the solution is not unique
- Next: maps for X(A,B,C) and Y(A,B,C)

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>X</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<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>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>0</td>
<td>1</td>
<td>1</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>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</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>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Karnaugh Maps: Visualization of algebra

The Karnaugh Map is a graphical representation used in digital logic design to simplify Boolean expressions. It's particularly useful for minimizing two-level logic circuits.

The map is divided into squares, each representing a combination of input variables (A, B, C, etc.). The values 0 and 1 are placed in the squares according to the truth table of the Boolean function.

The example shown here illustrates the use of a Karnaugh Map to visualize a function with three variables (A, B, C) and two output variables (X, Y). The map is divided into four groups of four squares each, with each group representing a different variable combination.

The shaded areas in the map highlight the groups that can be combined to simplify the Boolean expression. This simplification process involves grouping together the squares with the same output values and removing redundant terms from the expression.

The process of simplification can significantly reduce the complexity of the circuit, leading to a more efficient design.
Karnaugh Maps: Visualization of algebra

\[ A'B'C'+A'BC' = A'C' \]
\[ ABC+ABC' = AB \]
\[ A'B'C+A'BC+AB'C+ABC = A'C+AC = C \]

Thus minimized function is

\[ X = A'C'+AB \quad Y = C \]
## 4-variable Kmaps / Design

### K-Map

<table>
<thead>
<tr>
<th></th>
<th>00</th>
<th>01</th>
<th>11</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>1</td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td></td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>00</th>
<th>01</th>
<th>11</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

### Truth Tables

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

\[ F(A,B,C,D) = ABC' + A' C' D' + A' BC + ACD + \]

### Try them with Logisim
4-variable Kmaps / Design

\[ F(A, B, C, D) = B'D' + A'BC'D \]

\[ F(A, B, C, D) = ABC' + A'C'D + A'BC + ACD + ? \]

Try them with Logisim