Digital Logic
8 \text{ of } 10V

44 \uparrow 400
36 - 14 = 22

1 - 0 = 1
1 - 1 = 0
0 - 1 = 
- 0 
- 1 

0 0 0 0 1 1 1 0
36 - 14 = 22

\[ \begin{array}{c}
00100000 \\
-000001110 \\
\hline
00010110
\end{array} \]

36 + -14

\[ \begin{array}{c}
11110001 \\
+1 \\
\hline
111110010 \\
\hline
00100100 \\
+111110010 \\
\hline
000101110
\end{array} \]
56 - 21 = 35

\[
\begin{array}{cccc}
0 & 0 & 0 & 1 \\
- & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\
\hline
1 & 0 & 0 & 0 & 1 & 1
\end{array}
\]
Digital Values for Analog Signals

- Implement logical functions: AND, OR, NOT
- Digital symbols:
  - We assign a range of analog voltages to each digital (logic) symbol
  - Assignment of voltage ranges depends on electrical properties of transistors being used

![Image showing digital and analog voltage ranges]
Truth Table

- The most basic representation of a logic function
- It is a perfect induction proof - Lists the output for all possible input combinations
- How many rows of the truth table needed?

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>A B ...</td>
<td>X Y ...</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>2^#inputs</td>
<td></td>
</tr>
</tbody>
</table>
Truth Table: Inverter

- Inverted signals are denoted with an overbar \( \overline{A} \).
- Or with a prime symbol \( A' \).
- Or with a bubble in a circuit diagram.

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Truth Table: AND Gate

- The result of an AND operation is 1 if and only if all inputs are 1
- Depict AND by the multiplication symbol
  - $A \cdot B$
- Or by lumping the signals together
  - $AB$

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>A B</td>
<td>Y = A \cdot B</td>
</tr>
<tr>
<td>0 0</td>
<td>0</td>
</tr>
<tr>
<td>0 1</td>
<td>0</td>
</tr>
<tr>
<td>1 0</td>
<td>0</td>
</tr>
<tr>
<td>1 1</td>
<td>1</td>
</tr>
</tbody>
</table>
Truth Table: NAND Gate

- The result of an NAND operation is 0 if and only if all inputs are 1
- Depicted by adding a Bar to the +
- Or adding a dot to the gate

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Y = A \cdot B</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>
Truth Table: OR Gate

- The result of an OR operation is 1 if and only if any inputs are 1
- Depict OR by the addition symbol
  - \( A + B \)

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>A  B</td>
<td>Y = A + B</td>
</tr>
<tr>
<td>0  0</td>
<td>0</td>
</tr>
<tr>
<td>0  1</td>
<td>1</td>
</tr>
<tr>
<td>1  0</td>
<td>1</td>
</tr>
<tr>
<td>1  1</td>
<td>1</td>
</tr>
</tbody>
</table>

[Diagram of an OR gate]
Truth Table: NOR Gate

- The result of an OR operation is 1 if and only if all inputs are 0
- Depict NOR by the addition symbol with bar
- Or add a dot to the gate

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>A  B</td>
<td>Y = A + B</td>
</tr>
<tr>
<td>0  0</td>
<td>1</td>
</tr>
<tr>
<td>0  1</td>
<td>0</td>
</tr>
<tr>
<td>1  0</td>
<td>0</td>
</tr>
<tr>
<td>1  1</td>
<td>0</td>
</tr>
</tbody>
</table>
Truth Table: XOR Gate

- The result of an XOR operation is 1 if and only if its inputs differ
- Depict OR by the addition symbol
  - $\overline{A + B}$

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>A  B</td>
<td></td>
</tr>
<tr>
<td>0  0</td>
<td>0</td>
</tr>
<tr>
<td>0  1</td>
<td>1</td>
</tr>
<tr>
<td>1  0</td>
<td>1</td>
</tr>
<tr>
<td>1  1</td>
<td>0</td>
</tr>
</tbody>
</table>
Truth table to Gates

So giving some arbitrary truth table, how do you go about creating a transistor-based circuit for it?
1. Read each row of the truth table independently.
2. For each row that is 1, draw an AND gate that is 1 if and only if the inputs match that line of the truth table.
   1. This will require many inverters
3. Once all AND gates have been created, OR their outputs together.
4. This solution will work but is not always optimal...
Simple example

- XOR Gate – one or the other, but not both

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Y</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>
## Synthesis of an Arbitrary gate

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

Diagram of gate implementation:
Binary Adding Review

\[\begin{align*}
0 + 0 &= 0 \\
1 + 0 &= 1 \\
1 + 1 &= 2, \text{ which is } 10 \text{ in binary, sum is } 0, \text{ carry is } 1. \\
1 + 1 + 1 &= 3, \text{ sum is } 1, \text{ carry is } 1.
\end{align*}\]

Can we write that as a truth table?
A + B + Cin = Cout, S

**Full Adder Truth Table**

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Cin</th>
<th>Cout</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>-</td>
<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>
<td>0</td>
</tr>
<tr>
<td>-</td>
<td>1</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>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>-</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Diagram of full adder circuit.
# Full Adder Truth Table

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C(_{in})</th>
<th>C(_{out})</th>
<th>S</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>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</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>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<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>
<td>0</td>
</tr>
</tbody>
</table>
Boolean Algebra

- $0 \cdot 0 = 0$
- $1 + 1 = 1$
- $1 \cdot 1 = 1$
- $0 + 0 = 0$
- $0 \cdot 1 = 1 \cdot 0 = 0$
- $1 + 0 = 0 + 1 = 1$
- if $x = 0$ then $x' = 1$
- if $x = 1$ then $x' = 0$
Single-Variable Boolean Algebra

- \( x \cdot 0 = \)
- \( x + 1 = \)
- \( x \cdot 1 = \)
- \( x + 0 = \)
- \( x \cdot x = \)
- \( x + x = \)
- \( x \cdot x' = \)
- \( x + x' = \)
- \( (x')' = \)
Properties of Boolean Algebra

• Commutative
  \[- x \cdot y =
  \Rightarrow x + y =\]

• Associative
  \[- x \cdot (y \cdot z) =
  \Rightarrow x + (y + z) =\]

• Distributive
  \[- x \cdot (y + z) =
  \Rightarrow x + y \cdot z =\]
De Morgan’s Laws

• “Break the line, change the sign”

• Two laws:
  - \( A' + B' = (AB)' \)
    • This is the NAND gate
  - \( A' B' = (A+B)' \)
    • This is the NOR gate
De Morgan’s Laws

\[(A + B)' = A'B' \quad \text{conversely} \quad (AB)' = A' + B'\]

“Break the line, change the sign”

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>A+B</td>
<td>A+B</td>
<td>(\overline{A})</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>-----</td>
<td>-----</td>
<td>-------------</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</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>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>
</tr>
</tbody>
</table>
De Morgan’s Laws

\[(A + B)' = A'B' \quad \text{conversely} \quad (AB)' = A' + B'\]

“Break the line, change the sign”

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>AB</th>
<th>AB</th>
<th>A</th>
<th>B</th>
<th>A + B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<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>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Digital Logic Structures
Basic Logic Gates

- **NOT**: $\overline{A}$
- **OR**: $A + B$
- **NOR**: $\overline{A + B}$
- **AND**: $AB$
- **NAND**: $\overline{AB}$
- **XOR**: $A \oplus B$
More Than Two Inputs?

- AND and OR gates can take any number of inputs
  - AND gives 1 if all inputs are 1
  - OR gives 1 if any input is 1
- NAND??  NOR??
  - Not associative!
One-Bit Full Adder

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>C\textsubscript{in}</th>
<th>C\textsubscript{out}</th>
<th>S</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>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>
</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>1</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>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>
Four-Bit Full Adder

Ripple-carry adder:

Look ahead adder:

3 0 1
\rightarrow 2
Multiply

$2 \cdot 3 = 2 + 2 + 2$
Divide

$16 / 3 = 16 + \cdot 3 = 13 + \cdot 3 = 10 + \cdot 3 \ldots$
More Logic Structure

- As we start to build more complex structures we need ways to control parts of them
  - To select signals
  - To activate certain outputs
Signal Selection

A + ? + adder
Two-Way Multiplexer
Two-Way Multiplexer

2-way multiplexer: the output is equal to one of the two inputs, based on a selector.

<table>
<thead>
<tr>
<th>S</th>
<th>A</th>
<th>B</th>
<th>Y</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>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>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Four-Way Multiplexer

- $n$-bit selector and $2^n$ inputs, one output
  - output equals one of the inputs, depending on selector
- “Four-to-one mux”
Two-to-Four Decoder

- \( n \) inputs, \( 2^n \) outputs
  - exactly one output is 1 for each possible input pattern
- Generates a walking-ones pattern
- Uses:
  - Convert memory or register address to a control line
  - Convert an opcode to one of \( n \) control lines
  - We will get to this in the LC-3 material
Building functions from logic gates

• Combinational Logic Circuit
  – Output depends only on the current inputs
  – Stateless (memoryless)

• Sequential Logic Circuit
  – Output depends on the sequence of inputs (past and present)
  – Stores information (state) from past inputs

Memory
Combinational vs. Sequential

Two types of “combination” locks

**Combinational**
Success depends only on the values, not the order in which they are set.

**Sequential**
Success depends on the sequence of values (e.g., R-13, L-22, R-3).
Combinational vs. Sequential

• Combinational circuit
  – Always gives the same output for a given set of inputs
  – Example: Adder always generates sum and carry, regardless of previous inputs

• Sequential circuit
  – Remembers previous input
  – Output depends on state and input
Synchronization of Sequential Circuits

- These are real devices and require time to compute.
- If we want proper results we need a way to ensure consistent timing.
- One way of doing this is a clock:
  - Repeating signal at certain frequency
  - When you buy a computer this is the number in gigahertz
- All our actions take place in relation to this clock.
D-Flip-Flop (the one for Lab)

- Basic Memory Device
- Stores the value of D when conditions are met and outputs it on Q
- Otherwise Q holds the last value of D
- D-flip-flop is edge-triggered (changes only on the edge of the clock)
- This can be both edges or a single type (up or down)
D-Flip Flop: Timing Diagram
core dump
D-Flip-Flop with Write Enable

- Same idea as a Flip-Flop but adds another input
- Instead of changing on clock edges. You can only change on a clock edge when WE is high
D-Flip Flop: Timing Diagram (up)
Register

- A register stores a multi-bit value
- Common WE which latches the n-bit value
Memory

Now that we know how to store bits, we can build a memory – a logical $k \times m$ array of stored bits.

Address Space:
number of locations
(usually a power of 2)

Addressability:
number of bits per location
(e.g., byte-addressable)

$k = 2^n$ locations

$m$ bits
XP 32-bit

3GB
Memory

D

WE

CLK

n

Q

n
Memory

D + D + A + A + R/W

+2

clk

Q

CMPE-012/L

Maxwell James Dunne
2^2 x 3 Memory

- Address
- Word select
- Word WE
- Input bits
- Write enable
- Address decoder
- Output bits
State Machine
The basic type of sequential circuit
– Combines combinational logic with storage
– “Remembers” state, and changes output (and state) based on inputs and current state
Representing Multi-bit Values

- Number bits from right (0) to left (n-1)
  - just a convention -- could be left to right, but must be consistent
- Use brackets to denote range:
  D[l:r] denotes bit l to bit r, from left to right

\[
A = \overline{15} 010100110101010101
\]

\[
\]

May also see \(A<14:9>\), especially in hardware block diagrams.