Logic design?

- Digital hardware is implemented by way of logic design
- Digital circuits process and produce two discrete values: 0 and 1

- Example: 1-bit full adder (FA)
Layered design approach

- Logic design is done using logic gates
- Often we design a desired function using high-level languages and somewhat higher level than logic gates
- Two approaches in design
  - Top down
  - Bottom up

Transistor as a switch
An inverter

When $A = 1$

- "P"-type TR
- "N"-type TR

When $A = 1$:
- "OFF"
- "ON"
When $A = 0$

**Abstraction**
Logic gates

2-input AND

\[ Y = A \land B \]

2-input OR

\[ Y = A \lor B \]

2-input NAND

\[ Y = \neg (A \land B) \]

2-input NOR

\[ Y = \neg (A \lor B) \]

Describing a function

- Output_A = F(Input_0, Input_1, ..., Input_{N-1})
- Output_B = F'(Input_0, Input_1, ..., Input_{N-1})
- Output_C = F''(Input_0, Input_1, ..., Input_{N-1})
- ...

Methods
- Truth table
- Sum of products
- Product of sums
Truth table

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

Sum of products

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

- S = A’B’C_{in} + A’BC_{in}’ + AB’C_{in}’ + ABC_{in}
- C_{out} = A’BC_{in} + AB’C_{in} + ABC_{in}’ + ABC_{in}
Combinational vs. sequential logic

- **Combinational logic** = function
  - A function whose outputs are dependent only on the current inputs
  - As soon as inputs are known, outputs can be determined

- **Sequential logic** = combinational logic + memory
  - Some memory elements (i.e., "state")
  - Outputs are dependent on the current state and the current inputs
  - Next state is dependent on the current state and the current inputs
**Sequential logic**

- Inputs
- Current state
- Next state
- Outputs
- Clock

**Combinational logic**

- Any combinational logic can be implemented using sum of products or product of sums
- Input-output relationship can be defined in the truth table format
- From the truth table, each output function can be derived
- Boolean expressions can be further manipulated (e.g., to reduce cost) using various Boolean algebraic rules
Boolean algebra

- Boole, George (1815–1864): mathematician and philosopher; inventor of Boolean Algebra, the basis of all computer arithmetic

- Binary values: {0, 1}
- Two binary operations: AND (×/), OR (+)
- One unary operation: NOT (~)

Boolean algebra

- Binary operations: AND (×/), OR (+)
  - Idempotent
    - a·a = a+a = a
  - Commutative
    - a·b = b·a
    - a+b = b+a
  - Associative
    - a·(b·c) = (a·b)·c
    - a+(b+c) = (a+b)+c
  - Distributive
    - a·(b+c) = a·b + a·c
    - a+(b·c) = (a+b)·(a+c)
Boolean algebra

- De Morgan’s laws
  - \(- (a \cdot b) = \neg a + \neg b\)
  - \(- (a + b) = \neg a \cdot \neg b\)

- More...
  - \(a + (a \cdot b) = a\)
  - \(a \cdot (a + b) = a\)
  - \(\neg \neg a = a\)
  - \(a + \neg a = 1\)
  - \(a \cdot (\neg a) = 0\)

It is not true that I ate the sandwich and the soup.

\[\text{same as:}\]

I didn’t eat the sandwich or I didn’t eat the soup.

It is not true that I went to the store or the library.

\[\text{same as:}\]

I didn’t go to the store and I didn’t go to the library.

Expressive power

- With AND/OR/NOT, we can express any function in Boolean algebra
  - Sum (+) of products (⋅)

- What if we have NAND/NOR/NOT?
- What if we have NAND only?
- What if we have NOR only?
Using NAND only

A
\neg A
\neg (A \land A) = \neg A

A
B
\neg (A \land B)
A \land B
\neg (\neg (A \land B)) = A \land B

Using NOR only (your turn)

- Can you do it?
- NOR is \neg (A + B)

\begin{align*}
\text{NOT} & \quad \text{AND} & \quad \text{OR} \\
\neg (A + A) & = \neg (\neg (A + A) + \neg (B + B)) & = \neg (\neg (A + B) + \neg (A + B)) \\
\neg A \land \neg A & = \neg (\neg A \land \neg A + \neg B \land \neg B) & = (A + B) \land (A + B) \\
\neg A & = \neg (\neg A + \neg B) & = A + B \\
& = \neg (\neg A \land \neg (\neg B)) & = A \land B
\end{align*}
Using NOR only (your turn)

- Can you do it?
- NOR is \( \neg(A + B) \)
  - I.e., We need to write NOT, AND, and OR in terms of NOR

<table>
<thead>
<tr>
<th>NOT</th>
<th>AND</th>
<th>OR</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \neg A )</td>
<td>( A \land B )</td>
<td>( A + B )</td>
</tr>
<tr>
<td>( \neg A \land \neg A )</td>
<td>( \neg(\neg A) \land \neg(\neg B) )</td>
<td>( \neg(\neg (A + B) + \neg (A + B)) )</td>
</tr>
<tr>
<td>( \neg (A + A) )</td>
<td>( \neg (\neg A + \neg B) )</td>
<td>( \neg (\neg (A + A) + \neg (B + B)) )</td>
</tr>
</tbody>
</table>

Using NOR only (your turn)

\[ A \sim A \]

\[ A \land B \]

\[ A + B \]
Now, it’s really your turn….

- How about XOR?

<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>0</td>
</tr>
</tbody>
</table>

\[ C = A'B + AB' \]

Multiplexor (aka MUX)

\[ Y = S'A + SB \]
A 32-bit MUX

Use 32 1-bit muxes
Each mux selects 1 bit
S is connected to each mux

a. A 32-bit wide 2-to-1 multiplexer
b. The 32-bit wide multiplexer is actually an array of 32 1-bit multiplexors