The Janus-faced Boolean Function
One the side a truth table
The other an expression
Like snowflakes
Every truth table is unique
Expressions? Dime a dozen
Fueled by Boolean algebra
Many an expression
Gets you to your truth table destination

Frequently asked questions from the previous class survey

- Single precision vs double precision floating points
  - When would you use one over the other?
  - Operations such as add, subtract, multiply, divide on them
Topics covered in this lecture

- Hexadecimal numbers
- Boolean Algebra
  - Not, Or, and And
  - Xor
  - Nand
- De Morgan’s Laws
- Synthesizing Boolean functions
Hexadecimal

- Hexadecimal is **base 16**!
- Given what we’ve already seen so far, you probably know what that means
- Hexadecimal, or just **hex** for short, is a place-value system where
  - Each place represents a power of 16
  - and each place can be one of 16 symbols

Hexadecimal number representation

- As in all place-value systems, the rightmost place will still be the ones place
- The next place to the left will be the sixteens place, then the 256s (16 \( \times \) 16) place, then the 4,096s (16 \( \times \) 16 \( \times \) 16) place, and so on
- Simple enough!
But what about the other requirement that each place can be one of 16 symbols? [1/2]

- We usually have ten symbols to use to represent numbers, 0 through 9
- We need to add **six more** symbols to represent the other values

But what about the other requirement that each place can be one of 16 symbols? [2/2]

- We could pick some random symbols like & @ #, but these symbols have no obvious order
- Instead, the standard is to use A, B, C, D, E, and F
  - Either uppercase or lowercase is fine!
- In this scheme, A represents ten, B represents eleven, and so on, up to F, which represents fifteen
Symbols in the hexadecimal or hex number system

<table>
<thead>
<tr>
<th>Hexadecimal</th>
<th>Decimal</th>
<th>Binary (4-bit)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0000</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0001</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0010</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>0011</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>0100</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>0101</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>0110</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>0111</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>1000</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>1001</td>
</tr>
<tr>
<td>A</td>
<td>10</td>
<td>1010</td>
</tr>
<tr>
<td>B</td>
<td>11</td>
<td>1011</td>
</tr>
<tr>
<td>C</td>
<td>12</td>
<td>1100</td>
</tr>
<tr>
<td>D</td>
<td>13</td>
<td>1101</td>
</tr>
<tr>
<td>E</td>
<td>14</td>
<td>1110</td>
</tr>
<tr>
<td>F</td>
<td>15</td>
<td>1111</td>
</tr>
</tbody>
</table>

Consider the number 0x1A5 in hexadecimal

- What’s the value of this number in decimal?
- The rightmost place is worth 5
- The next place has a weight of 16, and there’s an A there, which is 10 in decimal, so the middle place is worth $16 \times 10 = 160$
- The leftmost place has a weight of 256, and there’s a 1 in that place, so that place is worth 256
- The total value then is $256 + 160 + 5 = 421$ in decimal
Another view of the number $0x1A5$ in hexadecimal

$$1 \times 256 + 10 \times 16 + 1 \times 5$$
$$= 256 + 160 + 5$$
$$= 421 \text{ in decimal}$$

Conversion from decimal to hex

- The hex number is constructed from **right to left**
- Step 1: Divide the decimal number by 16 and note down the remainder
- Step 2: Divide the obtained *quotient* by 16, and note remainder again
- Step 3: Repeat the above steps until you get **0** as the quotient
  - **Stopping criteria**
Example: Decimal number 421

- \( 421 \div 16 = 26 \) (Remainder 5)
- \( 26 \div 16 = 1 \) (Remainder 10 or A)
- \( 1 \div 16 = 0 \) (Remainder 1)

Hex representation: \( 0x1A5 \)
- \( = 1 \times 256 + 10 \times 16 + 1 \times 5 \)
- \( = 256 + 160 + 5 \)
- \( = 421 \) in decimal

Top to Bottom is Right to Left \{LSB to MSB\}

Some conversions across number systems

<table>
<thead>
<tr>
<th></th>
<th>Example 1</th>
<th>Example 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary</td>
<td>1111 0000 0000 1111</td>
<td>1000 1000 1000 0001</td>
</tr>
<tr>
<td>Hexadecimal</td>
<td>F00F</td>
<td>8881</td>
</tr>
<tr>
<td>Decimal</td>
<td>61,455</td>
<td>34,945</td>
</tr>
</tbody>
</table>
Logic Operations

- One use of bits is to represent the answers to yes/no questions such as “Is it cold?” or “Do you like my hat?”
  - We use the terms true for yes and false for no

- Questions like “Where’s the party?” don’t have a yes/no answer and can’t be represented by a single bit

- We often combine several yes/no clauses into a single sentence
  - We might say, “Wear a coat if it is cold or if it is raining” or “Go skiing if it is snowing and it’s not a school day”
Logic Operations

- Another way of saying those things might be
- “Wear coat is true if cold is true or raining is true” and “Skiing is true if snowing is true and school day is not true”
- These are logic operations that each produce a new bit based on the contents of other bits

Boolean Functions

- A boolean function is a function that operates on binary inputs and returns binary outputs
- Since computer hardware is based on representing and manipulating binary values ...
  - Boolean functions play a central role in the specification, analysis, and optimization of hardware architectures
Every Boolean function can be defined using two alternative representations

- First, we can define the function using a **truth table**
  - For each one of the \(2^n\) possible tuples of input variable values, the table lists the value of \(f(v_1, v_2, \ldots, v_n)\)
  - Can be thought of as a data-driven definition

- In addition to this data-driven definition, we can also define Boolean functions using **Boolean expressions**
  - For example: \((x \text{ Or } y) \text{ And Not } (z)\)

---

Boolean Algebra

- Algebra is a set of rules for operating on numbers

- **Boolean algebra** manipulates **two-state binary values** that are typically labeled true/false, 1/0, yes/no, on/off, and so forth

- We will use 1 and 0
Boolean Algebra

- **Boolean algebra**, invented in the 1800s by English mathematician George Boole, is a set of rules that we use to operate on bits.

- As with regular algebra: the associative, commutative, and distributive rules also apply:
  - $x \cdot y = y \cdot x$ Commutative
  - $(x \cdot y) \cdot z = x \cdot (y \cdot z)$ Associative
  - $x \cdot (y + z) = x \cdot y + x \cdot z$ Distributive

Boolean Operations

- There are three basic Boolean operations:
  - **Not**, **And**, and **Or**
  - Composite operations: **Xor** (short for “exclusive-or”), **Nand**, and **Nor**
NOT: This operation means “the opposite”

- For example, if a bit is false, NOT that bit would be true
- If a bit is true, NOT that bit would be false

<table>
<thead>
<tr>
<th>$x$</th>
<th>$\text{Not } x$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

AND: This operation involves 2 or more bits

- In a 2-bit operation, the result is true only if both the first AND second bit are true
- When more than 2 bits are involved, the result is true only if all bits are true

<table>
<thead>
<tr>
<th>$x$</th>
<th>$y$</th>
<th>$x \text{ And } y$</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>
**OR: This operation also involves 2 or more bits**

- In a 2-bit operation, the result is true if the first OR second bit is true; otherwise, the result is false.
- With more than 2 bits, the result is true if **any** bit is true.

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(x)</td>
<td>(y)</td>
<td>(x \text{ Or } y)</td>
</tr>
<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>

**Xor: Or Exclusive OR**

- The result of an exclusive-or operation is true if the first and second bits have **different values**
  - It’s either but not both.
- Because “exclusive-or” is a mouthful, we often use the abbreviation Xor (pronounced “ex-or”)

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(x)</td>
<td>(y)</td>
<td>(x \text{ Xor } y)</td>
</tr>
<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>
Nand: This operation involves 2 or more bits

- The name of the Nand operator is shorthand for Not-And, coming from the observation that Nand \((x, y)\) is equivalent to Not (And \((x, y)\))
- Pipes the output of the And gate through a Not gate

<table>
<thead>
<tr>
<th>(x)</th>
<th>(y)</th>
<th>(x) And (y)</th>
<th>(x) Nand (y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<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>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

What makes And, Or, and Not more interesting, or privileged, than any other subset of Boolean operators?

- The short answer is that indeed there is nothing special about And, Or, and Not
- A deeper answer is that various subsets of logical operators can be used for expressing any Boolean function, and \{And, Or, Not\} is one such subset
What makes And, Or, and Not more interesting, or privileged, than any other subset of Boolean operators?

- If you find this claim impressive, consider this: any one of these three basic operators can be expressed using yet another operator—Nand
  - The name of the Nand operator is shorthand for Not-And, coming from the observation that Nand \((x, y)\) is equivalent to Not (And \((x, y)\))

- Now, that’s impressive!
- It follows that any boolean function can be realized using Nand gates only
Representing a Boolean function using truth tables and Boolean expressions

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>z</th>
<th>( f(x, y, z) = (x \text{ Or } y) \text{ And } \text{Not}(z) )</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>1</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>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Truth Tables and Boolean Expressions [1/2]

- Given a Boolean function of \( n \) variables represented by a Boolean expression, we can always construct from it the function's truth table.
- We simply compute the function for every set of values (row) in the table.
  - This construction is laborious, and obvious.
Truth Tables and Boolean Expressions  [2/2]

- At the same time, the dual construction is not obvious at all:
- Given a truth table representation of a Boolean function, can we always **synthesize** from it a Boolean expression for the underlying function?
  - The answer to this intriguing question is Yes!

When it comes to building computers

- The truth table representation, the Boolean expression, and the ability to construct one from the other are all highly relevant
Suppose that we are called to build some hardware for sequencing DNA data

- Our domain expert biologist wants to describe the sequencing logic using a truth table
- Our job is to realize this logic in hardware
- With the truth table data as a point of departure, we can synthesize from it a Boolean expression that represents the underlying function
  - After simplifying the expression using Boolean algebra, we can proceed to implement it using logic gates

Truth table vs Boolean Expression

- A truth table is often a convenient means for describing some states of nature
- Whereas a Boolean expression is a convenient formalism for realizing this description in silicon
- The ability to move from one representation to the other is one of the most important practices of hardware design
Although the truth table representation of a Boolean function is unique

- Every Boolean function can be represented by many different yet equivalent Boolean expressions
  - And some will be shorter and easier to work with
- For example, the expression:
  - $(\neg (x \land y) \land (\neg x \lor y) \land (\neg y \lor y))$
  - Is equivalent to the expression $\neg x$
- The ability to simplify a Boolean expression is the first step toward hardware optimization

**Logic Gates**
Gates

- A gate is a physical device that implements a simple Boolean function.
- Most digital computers today use electricity to realize gates and represent binary data.
  - Today, gates are typically implemented as transistors etched in silicon, packaged as chips.

Lots of “can do” implementations of gates also exist alongside practical ones

- Any alternative technology permitting switching and conducting capabilities can be employed.
- Over the years, many hardware implementations of Boolean functions were created:
  - Including magnetic, optical, biological, hydraulic, pneumatic, quantum-based, and even domino-based mechanisms.
  - Many of these implementations are whimsical “can do” feats.
Implication of switching technologies and Boolean algebra [1/2]

- The availability of alternative switching technologies, on the one hand, and the observation that Boolean algebra can be used to abstract the behavior of logic gates, on the other, is extremely important.

- Implies that computer scientists don't have to worry about physical artifacts like electricity, circuits, switches, relays, and power sources.

Implication of switching technologies and Boolean algebra [2/2]

- Allows computer scientists to be content with the abstract notions of Boolean algebra and gate logic.

- Trusting blissfully that someone else—physicists and electrical engineers—will figure out how to actually realize them in hardware.
Primitive Gates as black boxes

- Primitive gates can be viewed as black box devices that implement elementary logical operations.

![And, Or, Not Gates](image)

Composite gates

- Since all logic gates have the same input and output data types (0's and 1's), they can be combined, creating composite gates of arbitrary complexity.

- For example, suppose we are asked to implement the three-way Boolean function And \((a, b, c)\), which returns 1 when every one of its inputs is 1, and 0 otherwise.

- Using Boolean algebra, we can begin by observing that \(a \cdot b \cdot c = (a \cdot b) \cdot c\).
Next, we can use this result to construct the composite gate

Let us consider another logic design example: Xor

- By definition, Xor \((a, b)\) is 1 exactly when either \(a\) is 1 and \(b\) is 0 or \(a\) is 0 and \(b\) is 1
- Said otherwise, Xor \((a,b) = \text{Or} (\text{And} (a, \text{Not}(b)), \text{And} (\text{Not} (a), b))\)
Note that the **interface** of any given gate is **unique**: there is only one way to specify it

- This is normally done using a truth table, a Boolean expression, or a verbal specification
- This interface, however, can be **realized in many different ways**
  - Some will be more elegant and efficient than others
- For example, the Xor implementation we saw in the previous slide is one possibility
  - There are more efficient ways to realize Xor, using less logic gates and less inter-gate connections

---

**Functionality vs Efficiency**

- From a **functional** standpoint, the fundamental requirement of logic design is that the gate implementation will **realize its stated interface**
  - One way or another
- From an **efficiency** standpoint, the general rule is to try to use **as few gates as possible**, since fewer gates imply less cost, less energy, and faster computation
Art of Logic Design: Abstraction to Implementation

- Given a gate abstraction (also referred to as specification, or interface) ...
- Find an efficient way to implement it using other gates that were already implemented

The contents of this slide-set are based on the following references
