ENEL260 CA1: Digital Logic

Ciaran Moore

Whether we're writing software for a social media site, a mobile phone app, or an autonomous car, we're using a computer to process information, make decisions, and create interesting behaviour. But *how* does a computer do these things? Understanding the answer to that question is the purpose of this section of the course.

We'll start out by looking at *digital logic*[1], which is the essence of how a computer does its work. In a later lecture we'll look at some simple electronic circuits[2] that can be used to actually make real-world digital logic. But for now we're just going to talk about what we want a computer to do, without worrying too much about how we might actually build a computer that'll do those things.

[1] By "logic" we just mean "rules for combining or transforming information".

[2] In principle, we could build a simple computer using transistors, mechanical gates, Minecraft blocks, or pretty much anything that we can use to do simple logical operations.

Information

We can think of "information" as being an answer to a question. The question might be *"what is the weather today?"*, or *"what did my friend in Hamilton just say?"*, or *"which way should the car turn now?"*. To build a computer that can do things with information we need a way of representing information in a physical system.

Analog: One way to represent information is by using a quantity. For centuries mechanical clocks have used the amount of sand or the position of gears to represent information about time – you can answer questions about what time it is by looking at an hourglass or the hands of a clock. In the 20th century, analog computers used electrical voltages to represent quantities like speed and distance[3] and answer questions about navigation or artillery fire.

[3] The value of the voltage is *analogous* to the speed or distance, hence the name *analog* computer.

Digital: A different way to represent information is by using two different values (a switch open or closed, a voltage at 0 V or 5 V, etc.) to answer a series of yes/no questions[4]. This is how a binary (i.e., two value) *digital* computer works. It's an approach that's simple, robust, and able to represent almost any kind of information (instead of only being able to represent analog quantities).

[4] Kind of like playing twenty questions.

Simple: two-valued information can be represented as a hole (or lack of hole) in a punch card, as two different voltages, as the presence or absence of light, as the orientation of magnetic domains, or pretty much anything that has two possible values.

Robust: Small errors in an analog voltage can lead to large errors in computations, whereas small errors in digital voltages won't necessarily change the answer from "yes" to "no".

Example: To determine the time from a series of yes/no questions we might ask questions like:

- Is the hour greater than or equal to 16? Yes
- Is the hour greater than or equal to 24? No (not possible)
- Is the hour greater than or equal to 20? Yes
- Is the hour greater than or equal to 22? No
- Is the hour greater than or equal to 21? No

The hour must be equal to 20.

Logical Values

Note that it's possible to have logic systems with more than two discrete levels. But so far these haven't been found to be very useful for practical computing.

LOGIC GIVES US A SET OF RULES for "processing" information represented by answers to yes/no questions (i.e., true and false statements) to get additional information (i.e., logical conclusions).

Example:

- Statement: The power must be on for the light to be on
- Statement: The light is on
- Conclusion: The power is on

Since we're interested in digital computers, we're going to focus on binary values and binary digital logic. In a digital logic system the "positive" value goes by a lot of different names, including "high", "on", "true", and 1. The opposite, or inverse value, has names like "low", "off", "false", or 0:

- When we're writing software we typically use the names "true" and "false" for the two values.
- When we're talking about hardware we typically use the names 1 (or "high") and 0 (or "low") for the two values.

The names don't really matter. What's important is that there are two values, and that (as we'll see) we can come up with some simple rules for combining those values.

In later lectures we'll see how to encode complex information using a group of binary values. But for this lecture we're going to focus on working with information represented by just a single binary value.

We refer to a single binary value (a 0 or 1) as a *bit*. The most common grouping of binary values is the *byte*, which is made up of 8 bits (e.g., 10011010).

Logical Variables

If you've come across Boolean variables while learning programming this should all sound very familiar.

A BINARY *logical variable* REPRESENTS a single yes/no question (i.e., a single *bit* of information), and is distinguished by three attributes:

1. It can assume one of the two possible logical values;
2. The value that the variable takes on expresses a declarative statement, e.g., "the power is on", "the power is off", "*a* is TRUE";
3. The value that the variable can take on are mutually exclusive (i.e., it makes no sense for the variable to be both true and false at the same time).

If a logical state is represented by the logical variable

$$a$$

then the inverse of that state is written as

$$\overline{a}.$$

Example: When the logical variable *isPowered* has the value 1 then we interpret the statement "the power is on" as true.

When the logical variable *isPowered* has the value 0 then we interpret the statement "the power is on" as false (or "the power is off" as true).

When $isPowered = 1$ then $\overline{isPowered} = 0$.

When $isPowered = 0$ then $\overline{isPowered} = 1$.

Functions of Logic Variables

For example, the function $f(x) = 2x$ is a rule that makes the dependent variable twice the value of the independent variable.

YOU'VE PROBABLY ENCOUNTERED FUNCTIONS in maths classes. A function is just a *rule* which enables us to find the value of a second (dependent) variable from the first (independent) variable:

$$z = f(a). \tag{1}$$

In a maths class you're usually dealing with functions of integer- or real-valued variables (which have an infinite number of values), so you typically write the rule that defines a function as an *algebraic equation*. But since logical variables only have two values, we can often specify a function of a logical variable using a *"truth table"* that shows all possible values of the independent variable (e.g., a) and the corresponding values of the function.

Example: Here's the truth table for a function, $z = f(a)$:

Input (a)	Output (z)
1	0
0	1

We could also write the function as an equation. Looking at the values of z for each value of a in the truth table, it should hopefully be obvious that the equational definition of the function is:

$$z = f(a) = \overline{a}. \tag{2}$$

Another consequence of the fact that logical variables only have two values is that the number of logical functions that we can define is limited. For a single logical variable, a, there are only four possible logical functions:

Input (a)	Output (z)			
	z_1	z_2	z_3	z_4
0	0	1	0	1
1	0	0	1	1

With two logical independent variables, a and b, the set of possible functions, $z = f(a, b)$, is larger but still finite:

Inputs (a, b)	Output (z)															
	z_1	z_2	z_3	z_4	z_5	z_6	z_7	z_8	z_9	z_{10}	z_{11}	z_{12}	z_{13}	z_{14}	z_{15}	z_{16}
0,0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
0,1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1,0	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1,1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1

More generally, the number of rows in each table is the number of unique patterns for the independent logical variables, i.e. 2^N for N variables. The number of output columns (or possible functions) in each table is the number of unique patterns which can be formed over that number of rows, i.e. $2^{(\text{number of rows})}$, or 2^{2^N}. This is a number which gets very large very fast as N grows.

Note that some of these functions have special names, e.g. z_9 is the AND function, z_7 is the XOR (exclusive or) function and z_{15} is the OR function.

Fundamental Logical Functions

Aside: The exclusive-or (XOR) operator, i.e. z_7 in the table above, written $a \oplus b$, is so commonly used that it has its own name. But, from the truth table above, it's easy to see that it can be defined in terms of the three fundamental functions as $a \oplus b = a \cdot \overline{b} + \overline{a} \cdot b$.

It turns out that we only need three fundamental logical functions to represent every possible logical function:

- AND: the logical *and* (or dot) operator $a \cdot b$, often abbreviated to $a\,b$ ($a \cdot b = 1$ when *both* a and b are 1)
- OR: the logical *or* (or plus) operator $a + b$ ($a + b = 1$ when *at least* one of a or b is 1)
- NOT: the inversion operator $\overline{a}$ ($\overline{a} = 1$ when at a is 0)

Every other logical function, including functions of more than two variables, can be written in terms of these three functions (this is sometimes known as writing the function in its *canonical representation*). To see why this is so we need to understand the algebraic properties of logical variables.

As we'll see below, we can actually use just a single kind of logical function to express every other possible logical function. But AND/OR/NOT is the classical set of logical functions, and the ones that you're most likely to encounter (for example, you may have run into them when using a search engine or programming conditional statements.

Aside: Fundamental Logical Functions in C

The "C" language provides two distinct mechanisms for dealing with Boolean logic. Both mechanisms offer the sufficient operators of AND, OR, and NOT.

The difference between the two mechanisms lies in how the values contained by a variable are interpreted. Internally, a variable in C is made up of more than one bit. But when programming in C we can choose what those bits mean:

- We can interpret the variable as a single logical value, where "all bits are 0" means "false" and "any bit is non-zero" means "true".
- We can interpret the variable as a group of logical values, where each individual bit gives a piece of "true" or "false" information.

These two different interpretations give rise to the *logical* and *bitwise* operators respectively. We'll talk about the C *bitwise operators* in the next lecture. The logical operators are:

- NOT: `!varA`
- OR: `varA || varB`
- AND: `varA && varB`

These operations assume that the variables `varA` and `varB` are either "true" or "false", where "true" in the C language means non-zero and "false" means zero valued. Operations like those listed above are executed in the *arithmetic logic unit* (ALU) within a computer.

Aside: Traditionally, programmers would use the C preprocessor to define logical constants:

```
#define TRUE 1
#define FALSE 0
```

In modern C, the `stdbool` standard library provides both a Boolean datatype and values for `true` and `false`.

Useful Algebraic Properties

IN ALGEBRA WE EXPECT a number system and its operators (let's use $*$ and $\star$ to represent arbitrary operators) to be describable by a number of useful properties[5]:

[5] You may remember these properties from some of your maths classes.

- Closure: for $a, b \in \mathcal{B}$, then $a * b \in \mathcal{B}$
- Identity: $a * i = a$ (thus $a \cdot 1 = a$ and $a + 0 = a$)
- Associative: $a * (b * c) = (a * b) * c$
- Commutative: $a * b = b * a$
- Distributive: $a * (b \star c) = a * b \star a * c$

As it turns out, the logical operators AND and OR satisfy these properties. Additionally, we can define some other properties of logical operations:

See `https://en.wikipedia.org/wiki/Boolean_algebra` for a more detailed explanation of Boolean algebra.

- Annihilation: $x \cdot 0 = 0$ and $x + 1 = 1$
- Absorption: $x + (x \cdot y) = x$ and $x \cdot (x + y) = x$
- Idempotence: $x + x = x$ and $x \cdot x = x$
- Double Negation: $\overline{\overline{a}} = a$
- Or-Complement: $a + \overline{a} = 1$
- And-Complement: $a \cdot \overline{a} = 0$

The algebraic properties for true/false logic where developed by the mathematician George Boole[6] in the mid 1800s, and are hence called *Boolean algebra* or *Boolean logic*. Although Boolean algebra was originally only of interest to philosophers and mathematicians,

[6] `https://en.wikipedia.org/wiki/George_Boole`

in the 1930s the electrical engineer Claude Shannon[7] showed how Boole's ideas could be applied to the design of switching relay circuits.

[7] https://en.wikipedia.org/wiki/Claude_Shannon

Using the properties of Boolean algebra we can algebraically manipulate logical expressions. This is useful to test if two logical expressions are equivalent, or to reduce a logical expression to a simpler form. We'll test your understanding of Boolean algebra through the online quiz.

Example: Is $a\,\overline{b} + a\,b$ equivalent to a?

From the Distributive property we get $a\,\overline{b} + a\,b = a\,(\overline{b} + b)$.

Using the Or-Complement property gets us to $a\,\overline{b} + a\,b = a\,(\overline{b} + b) = a \cdot 1$.

Finally, by the Identity property we get $a\,\overline{b} + a\,b = a\,(\overline{b} + b) = a \cdot 1 = a$.

De Morgan's Theorem

De Morgan's theorem[8] is a consequence of the basic Boolean algebraic properties. It has two forms:

[8] See https://en.wikipedia.org/wiki/De_Morgan%27s_laws

$$\begin{aligned}\overline{a+b} &= \overline{a}\cdot\overline{b}\\ \overline{a\cdot b} &= \overline{a}+\overline{b}\end{aligned}$$

Both are easily verified with a truth table, and both extend to more than 2 variables.

A consequence of De Morgan's theorem is that $ab + cd = \overline{\overline{ab}\cdot\overline{cd}}$.

Sufficiency

ANY TWO-VARIABLE LOGICAL FUNCTIONS can be expressed by using a combination of AND, OR, and NOT.

Furthermore, any *multi-variable* logical function (e.g., $f(a,b,c,d)$) can be defined using the two variable functions by applying the commutative, associative, and distributive properties.

Example: The 3-variable logical function $f(a,b,c) = a\cdot b\cdot c$ is equivalent to (by associativity) $f(a,b,c) = a\cdot(b\cdot c)$. This is just a composition of two 2-variable AND functions.

Because we can build *any* logical function from AND, OR, and NOT, we say that these operations are *sufficient* to define digital logic.

The sufficiency property of AND, OR, and NOT isn't just of theoretical interest! It means that we only need to design three kinds of electronic switching circuits to create a computer.

See Wikipedia's NAND Logic article for more.

In fact, it's possible to go even further, and show that any function can be expressed in terms of only AND and NOT, or from OR and NOT. As a consequence of De Morgan's Theorem, it can even be shown that just one function, either NAND or NOR[9], is sufficient:

- NOT: $\overline{a} = a\ NAND\ a$
- AND: $a \cdot b = \overline{a\ NAND\ b} = (a\ NAND\ b)\ NAND\ (a\ NAND\ b)$
- OR: $a + b = \overline{a}\ NAND\ \overline{b} = (a\ NAND\ a)\ NAND\ (b\ NAND\ b)$ (by De Morgan's Theorem)

[9] NAND (NOT-AND) is function z_8, and is equivalent to inverting AND. NOR (NOT-OR) is function z_2, and is equivalent to inverting OR.

With just *one* kind of logic circuit to design the implementation process becomes much easier. However, in programming we tend to stay with AND, OR, and NOT (and with XOR).

Logic Gates

LOGICAL FUNCTIONS CAN BE DEFINED USING TRUTH TABLES, or, as we saw above, they can be written as algebraic equations. A third alternative is to present a function as a diagram. The logic diagram symbols for the AND, OR, NOT and XOR "gates" are shown in Fig. 1. Each gate can be implemented in a standard form (for example as a transistor circuit). Logic diagrams are an abstract way to represent the design of a physical implementation or digital electronic circuit.

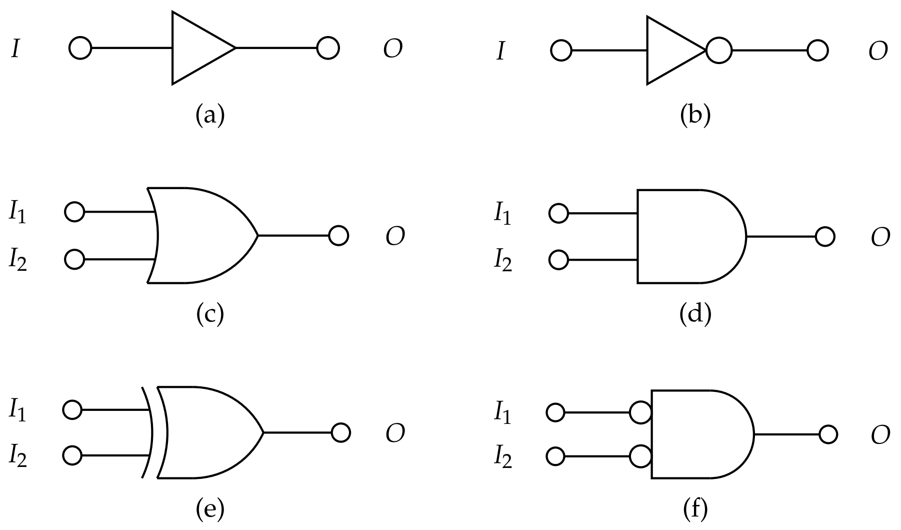

Figure 1: A range of logic gate symbols: (a) The buffer gate ($O = I$), (b) the inverter or NOT gate, (c) the OR gate, (d) the AND gate, (e) the XOR gate, and (f) an AND gate with inverted inputs. Note that an open circle for an input or an output *drawn up against the gate symbol* indicates *inversion* for that connection.

Example logic diagram symbols for the NAND and NOR gates are shown in Fig. 2.

The logic diagrams shown in Fig. 3 illustrate two important circuits: the half and full 1-bit *adders*. These circuits allow us to perform binary (base-2) arithmetic. Here we start to see the connection between digital logic and *computation*.

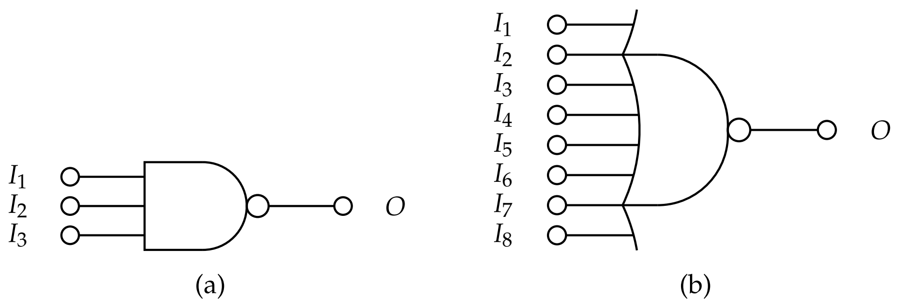

Figure 2: More logical gate symbols: (a) the NAND gate (3 inputs in this case), and (b) NOR gate (8 inputs in this case).

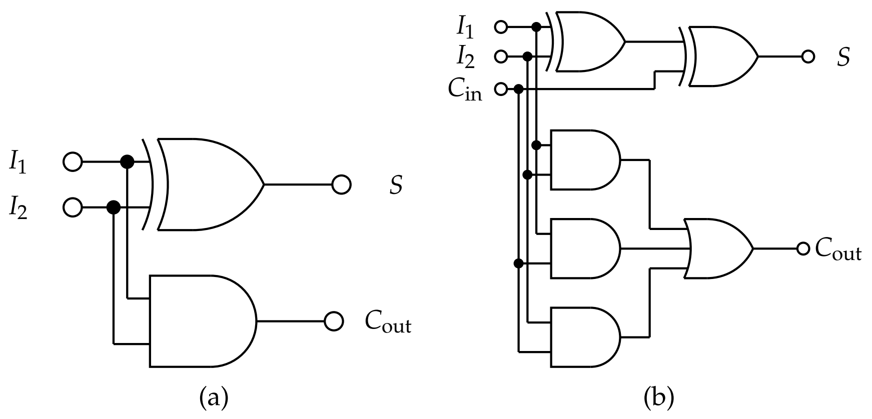

Figure 3: One-bit adder circuits: (a) half-adder, (b) full-adder.

Inputs		Outputs	
I_1	I_2	S	C_{out}
		(sum)	(carry out)
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

(a)

Inputs			Outputs	
I_1	I_2	C_{in}	S	C_{out}
		(carry in)	(sum)	(carry out)
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1

(b)

Figure 4: Truth tables for the 1-bit adder circuits: (a) half-adder, (b) full-adder.

Example: Binary (or base-2) arithmetic is similar to the base-10 arithmetic you're used to. But we only have two digits, 0 and 1. So instead of counting $0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, \ldots$, we count$0, 1, 10, 11, 100, 101, \ldots$. And addition looks like this:

- $0 + 0 = 0$
- $0 + 1 = 1$
- $1 + 1 = 0$ (and "carry" the 1 to the next column to get 10)

The "carry" inputs and outputs in the full-adder circuit allow us to connect a set of adders together to sum multi-bit binary numbers.

Combinational and Sequential Logic

Logic circuits that are designed so there are no loops in the diagram are called *combinational logic*: the present output of the circuit is a combination of the present inputs, and the complexity of the circuit is *combinatorial*. Conceptually the analysis, manipulation, and synthesis of combinatorial logic is relatively simple. The difficulties arise from the way the complexity grows in a combinatorial (factorial) way.

When the logic circuit has loops, also known as *feedback*, the circuit can retain information about what it has previously output. In other words, it has *state*[10]. This kind of circuit is called *sequential logic*: the present output of the circuit depends not just on the *present* input, but on the *sequence of previous* inputs. The analysis and synthesis of sequential logic circuits are much more difficult than analysis of combinational circuits, since (as we'll see in later lectures) we must include *time* in the analysis — i.e., the output is time dependent. Equivalently, if a logic circuit has a way of storing a value from a previous operation, the output is time dependent.

[10] As Abelson and Sussman say in *The Structure and Interpretation of Computer Programs*: "An object is said to *"have state"* if its behavior is influenced by its history."

Here's a fragment from a C program that illustrates both kinds of logic:

```
int varA;
int varB;
varA = 2057;
varB = varA * -17;
```

In the final line of the fragment the expression on the right hand side is evaluated by combinational logic (in the ALU mentioned earlier). But as each line of code is executed, the state of the variable on the left of the equals sign is changed (information is stored and can change over time), making the overall program an example of sequential logic.

Looking Deeper: Logic Gates in Python If you're interested in playing around with combinational logic without actually having to build hardware, you can model it fairly easily in Python. For example, we

You can find a Python file with additional example code on Learn.

can model a generic two-input logic gate as a function that performs a lookup in a truth table:

```
def two_input_gate(truth_table, in1, in2):
    """Return an output found from a truth table."""
    return truth_table[(in1, in2)]
```

And then build an AND gate by specifying the appropriate truth table:

```
and_table = {(0,0):0,
             (0,1):0,
             (1,0):0,
             (1,1):1}

def and_gate(in1, in2):
    return two_input_gate(and_table, in1, in2)
```

Here's how it works:

```
>>> a = 1
>>> b = 0
>>> and_gate(a, b)
0
>>> b = 1
>>> and_gate(a, b)
1
```

Using just a single gate is perhaps not that exciting. But once we have some basic logic gates, we can build and test components made up of more complex combinational logic circuits. For example, here's a half-adder:

```
def half_adder(in1, in2):
    """
    Returns a 2-tuple containing
    1. the binary sum of the inputs
    2. a 'carry' value that is non-zero if the sum overflows
    """
    return (xor_gate(in1, in2), and_gate(in1, in2))
```

And here's the half-adder in use:

```
>>> half_adder(0,1)
(1, 0)
```

Some other things you could try:

- Build a NAND gate, and then construct AND, OR, and NOT components (or more complex logic) using just the NAND alone.
- Build a NOR gate, and then construct AND, OR, and NOT components (or more complex logic) using just the NOR alone.
- Construct components for each of the two-input logical functions using only AND, OR, and NOT.
- Try building a sequential logic circuit? What problems do you run into?