Brandeis University
Fall 2014

Physics 29a
Professor Bensinger

Notes for Lab 8
Basics

1. Digital versus Analog

An analog signal is one that can have continuous values. For example, a voltage can be any number in principle and can vary continuously. A digital signal has just two discrete values. The signal may be a voltage, a current, or something else (such as magnetization of a hard disk or reflectivity of an area of a CD).

Combinations of digital signals can be used to represent just about anything, including logic levels, numbers, sound (as on a CD), pictures, movies, etc. The power of digital electronics is the ease of storing, transmitting, and manipulating digital signals.

2. Logic Levels

There are different conventions for logic levels. We will primarily work with TTL (which stands for transistor-transistor logic, which we will explain a bit more later), a very common convention where digital signals are 0V for low and 5V for high.

It is not necessary or even possible for the signals to be exactly 0V or exactly 5V. For TTL, a signal is considered low if the signal is below 0.8V and is considered high if the signal is above 2.4V (see figure 1). Signals between levels are indeterminant, and if input to a logic device, the output is uncertain.

3. Transistor Switch and a TTL inverter

Logic gates are implemented as simple transistor circuits. Figure 2 shows a simple one transistor switch. If the input A is low (OV), the base-emitter junction voltage difference is below 0.7V, meaning that the transistor is off and no current (I) flows through it. Since there is no current in the resistor, the output Y is 5V (that is, the voltage across the resistor is zero).

When the input is high (near 5V), the base-emitter junction is forward biased, and the transistor is saturated (that is, conducting). In this case, the collector of the transistor is a few tenths of a volt above the emitter, that is, the output is around 0.4V, which is a logic low.

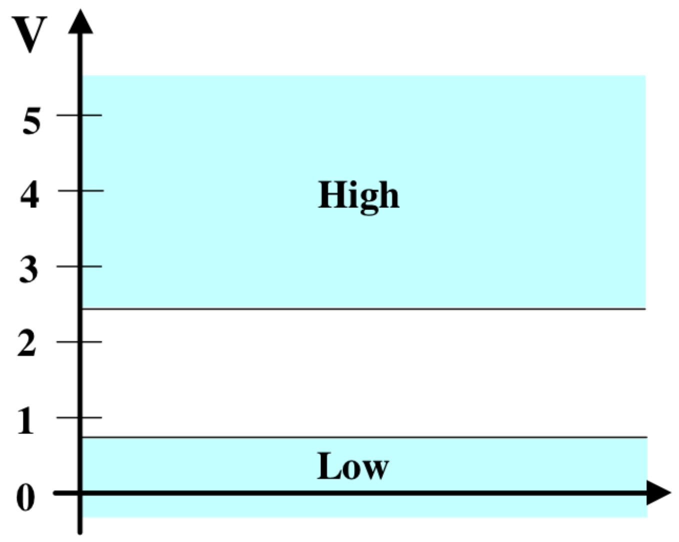


FIGURE 1. TTL logic level ranges.

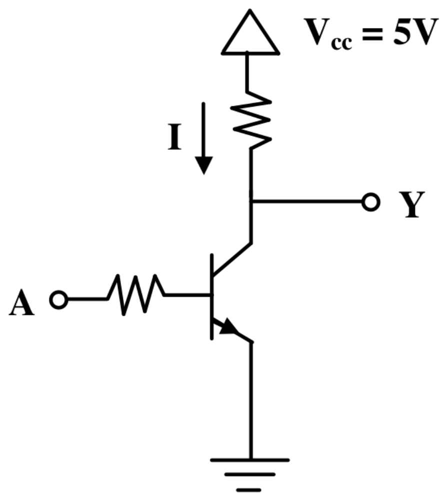


FIGURE 2. Transistor switch inverter circuit. A is the input and Y is the output.

Thus, this transistor switch acts as an inverter (also known as a NOT gate). The output is high if the input is low and is low if the input is high.

This simple circuit has two primary disadvantages. First, and more important, is the fact that since the transistor goes from fully off to fully on (or vice versa), transitions are relatively slow. Secondly, the output impedance is not very low, limiting how many other gates or other circuitry can be driven.

These problems are overcome in actual TTL circuits by adding a few more transistors. An actual TTL implementation of an inverter is shown in figure 3. I won't go into the details, but it works basically as the transistor switch circuit.

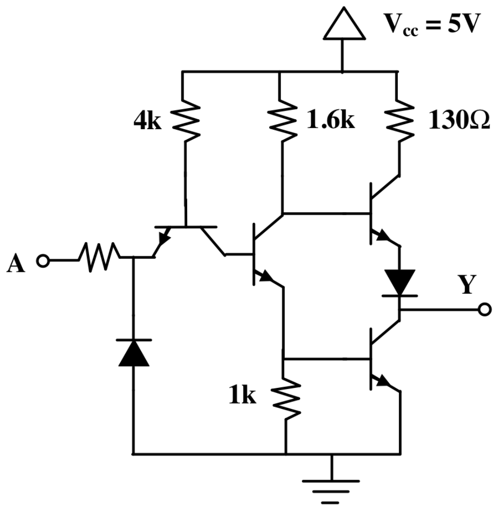


FIGURE 3. TTL inverter circuit. A is the input and Y is the output.

A major advantage to simple transistor circuits like these is that they and similar ones for other gates can be easily implemented in integrated circuits, take up little space, and can be repeated many times. Thus, it is possible to construct chips with a very large number of gates (millions) that can do very complex digital manipulations.

4. Combinatorial Logic and Logic Gates

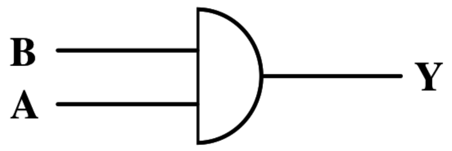
Combinatorial logic circuits are ones where the output depends only on the current state of the inputs, not on the history of the inputs. The basic units of combinatorial logic are logic gates, which are simple circuits that have a few inputs (typically two, although it can be from one up to a few) and one output. For each type of gate, there is a simple rule for how the output depends on the inputs.

4.1. **AND Gate.**

Figure 4a shows the symbol for an AND gate with inputs A and B and output Y. We think of the signals as 0 or 1 (alternately, False or True). The rule for an AND gate "the output is 1 if A and B are both 1; otherwise the output is 0." We summarize this rule in a truth table having one column for each input and output and having one row giving the output for each possible combination of values of the inputs. Figure 4b shows the truth table for an AND gate. We will write a Boolean algebra expression for an AND gate as Y = A·B or Y = AB. More will be said about Boolean algebra below.

Note that AND gates (like other gates) may have more than two inputs. In this case, the rule generalizes to "the output is 1 if and only if all the inputs are high."

(a)



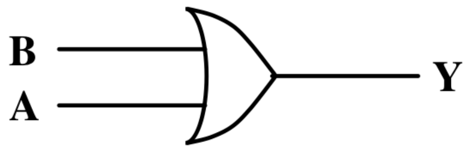
(b)

A	B	Y
0	0	0
1	0	0
0	1	0
1	1	1

FIGURE 4. (a) Symbol for an AND gate. (b) Truth table for an AND gate.

4.2. **OR Gate.** An OR gate and its truth table are shown in figure 5. The rule for an OR gate is "output is 1 if either input is 1; otherwise the output is 0." The Boolean expression is Y = A + B.

(a)



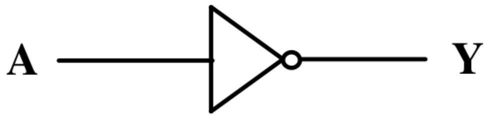
(b)

A	B	Y
0	0	0
1	0	1
0	1	1
1	1	1

FIGURE 5. (a) Symbol for an OR gate. (b) Truth table for an OR gate.

4.3. **NOT Gate.** The NOT gate (also known as an inverter) and its truth table are shown in figure 6. This is a simple one input gate whose rule is "the output is the opposite logic level from the input." The Boolean expression for this is Y = $\overline{\mathbf{A}}$.

(a)

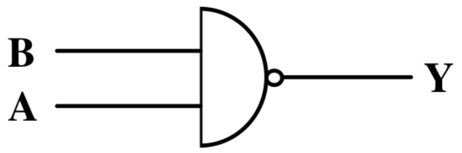


(b)

A	Y
0	1
1	0

FIGURE 6. (a) Symbol for a NOT gate. (b) Truth table for a NOT gate.

4.4. **NAND Gate.** The NAND gate (figure 7) is in essence an AND gate followed by a NOT gate (NAND stands for Not AND). The Boolean expression is Y = $\overline{\mathbf{AB}}$. Note that any one or two input gate can be constructed from NAND gates.

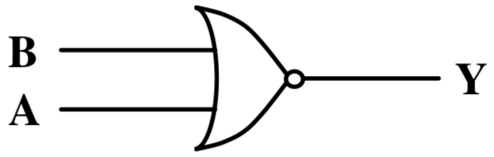
(a) 

(b)

A	B	Y
0	0	1
1	0	1
0	1	1
1	1	0

FIGURE 7. (a) Symbol for an NAND gate. (b) Truth table for an NAND gate.

4.5. **NOR Gate.** The NOR gate (figure 8) is in essence an OR gate followed by a NOT gate (NOR stands for Not OR). The Boolean expression is Y = $\overline{\mathbf{A + B}}$. Note that any one or two input gate can be constructed from NOR gates.

(a) 

(b)

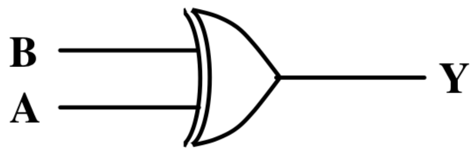
A	B	Y
0	0	1
1	0	0
0	1	0
1	1	0

FIGURE 8. (a) Symbol for a NOR gate. (b) Truth table for a NOR gate.

4.6. **XOR Gate.** The Exclusive OR gate (XOR) is shown in figure 9 and has the rule "the output is 1 if A or B is 1 but not both; otherwise the output is 0." The Boolean expression for an XOR gate is Y = A $\oplus$ B.

4.7. **XNOR Gate.** An exclusive NOR gate is a combination of an XOR gate and a NOT gate (figure 10).

4.8. **Other 2 Input Gates.** Since there are four rows in a two-input gate truth table and the output in each row can have two values, there are 16 (=2^4) possible truth tables. The AND, OR, NAND, NOR, XOR, and XNOR gates represent six of these. Two more where the output is always 0 (Y = 0) or always 1 (Y = 1) are trivial, and we don't need gates to do this [if you need a signal that is always 0 (1), tie it to ground (+5V)]. There are four more

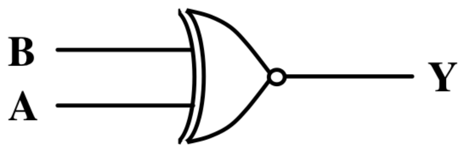


(a)

(b)

A	B	Y
0	0	0
1	0	1
0	1	1
1	1	0

FIGURE 9. (a) Symbol for an XOR gate. (b) Truth table for an XOR gate.



(a)

(b)

A	B	Y
0	0	1
1	0	0
0	1	0
1	1	1

FIGURE 10. (a) Symbol for an XNOR gate. (b) Truth table for an XNOR gate.

combinations that represent either one of the inputs or its negation, that is, Y = A, Y = B, Y = $\overline{\mathbf{A}}$, or Y = $\overline{\mathbf{B}}$. This leaves four non-trivial combinations that aren't represented by standard gates. These correspond to Y = A$\overline{\mathbf{B}}$, Y = $\overline{\mathbf{A}}$B, Y = A+$\overline{\mathbf{B}}$, and Y = $\overline{\mathbf{A}}$+B. There are not standard gates for these combinations. However, circuits that do these functions can be constructed from the standard gates.

4.9. **Logic Circuits.** Logic gates can be combined to form logic circuits that do any logical function that one desires. For example, figure 11 shows a logic circuit and its truth table. There are three inputs (A, B, and C) and the output expression is Y = $\overline{\mathbf{AB} + \overline{\mathbf{C}}}$.

4.10. **TTL Chips.** Logic gates are implemented in many electronic forms. We will use a very standard one known as TTL chips. Each chip is a DIP (dual inline package) and has one or more gates on it.

On each chip is a number of the form 74XXNN. XX is zero to three letters that tell you something about the transistor circuits that implement the gates. Possibilities are standard (no letters), high speed (H), low power (L), Schottky (S), low power Schottky (LS), advanced Schottky (AS), or advanced low power Schottky (ALS). In this course, we will not be pushing the limits of either speed or power consumption, so these letters don't make much difference to us.

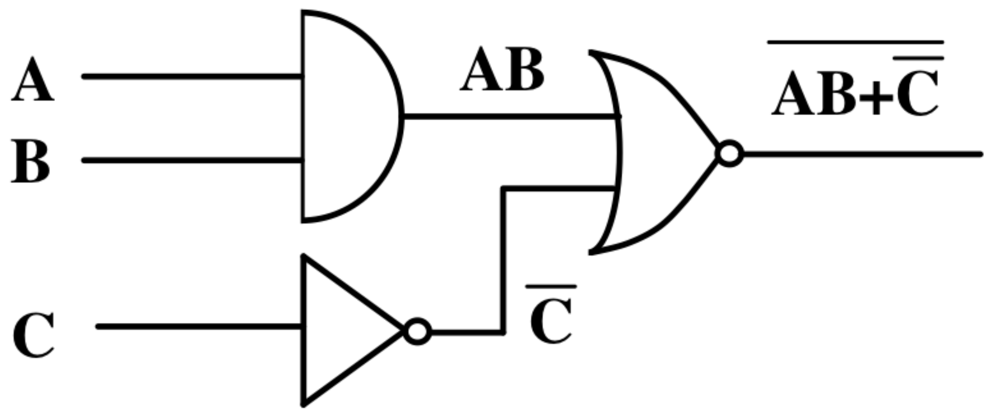


FIGURE 11. An example logic circuit.

The NN on the chips is two or three digits that describe the function of the chip. These can be looked up in a TTL Data Book, which gives the function, pin outs, internal circuits, and other specs. There is also a link on LATTE giving information on many of the common chips.

For example, a 74LS00 is a low power Schottky quad NAND gate, meaning that the chip contains four 2-input NAND gates. Figure 12 shows the pin out for a 74XX08, which is a quad AND gate. The chip has 14 pins arranged in two rows of seven. On TTL chips, you will find a notch a one end. If you are looking at the top and the notch is on the left (as in the figure), then pin 1 is in the lower left. Often, there is a small dot near pin 1. Pin 7 (GND) is connected to ground, and pin 14 (VCC) is connected to +5V.

5. Boolean Algebra

Boolean Algebra is an arithmetic where the only values are 0 and 1. The addition and multiplication tables are simple.

$$\begin{array}{llll} 0+0=0 & 0+1=1 & 1+0=1 & 1+1=1 \\ 0\cdot 0=0 & 0\cdot 1=0 & 1\cdot 0=0 & 1\cdot 1=1 \end{array}$$

Note that addition behaves like an OR gate and multiplication behaves like an AND gate. Both addition and multiplication are commutative and associative. Furthermore, they obey the distributive principle.

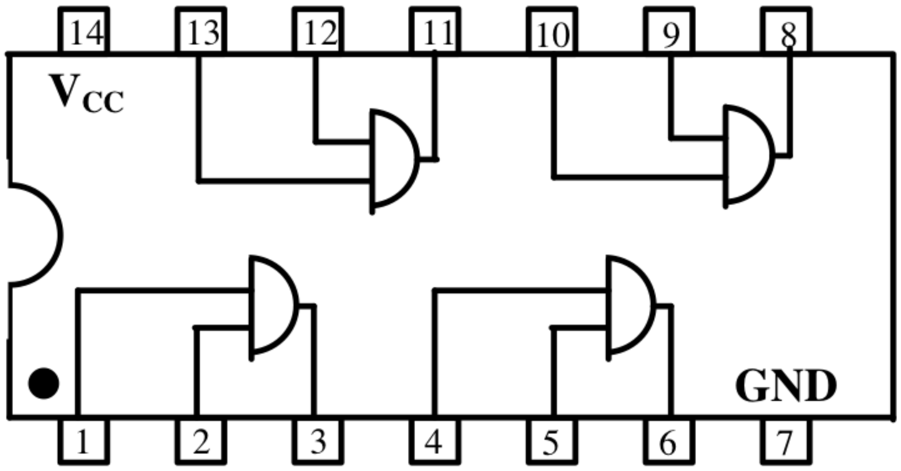


Figure 12. Pin out for an 74XX08 quad AND gate.

There is another operation in Boolean algebra known as negation, in which a value of 1 is changed to 0 and vice versa. If A is a Boolean variable, the negation of A (or the not of A) is symbolized by $\overline{\mathrm{A}}$.

Below are some useful identities in Boolean algebra. Some seem natural, but some are a little strange (like A + A = A) because variables have only two values.

$$\begin{array}{ll} \mathrm{A}\cdot 0 = 0 & \mathrm{A} + 0 = \mathrm{A} \\ \mathrm{A}\cdot 1 = \mathrm{A} & \mathrm{A} + 1 = 1 \\ \mathrm{A}\cdot \mathrm{A} = \mathrm{A} & \mathrm{A} + \mathrm{A} = \mathrm{A} \\ \mathrm{A}\cdot \overline{\mathrm{A}} = 0 & \mathrm{A} + \overline{\mathrm{A}} = 1 \\ \overline{\overline{\mathrm{A}}} = \mathrm{A} & \end{array}$$

5.1. **DeMorgan and Other Special Identities.** Let A and B be Boolean numbers. Some special (and useful) identities are

$$\begin{array}{ll} \mathrm{A} + \mathrm{AB} = \mathrm{A} & \mathrm{A} + \overline{\mathrm{A}}\mathrm{B} = \mathrm{A} + \mathrm{B} \\ \overline{\mathrm{A} + \mathrm{B}} = \overline{\mathrm{A}}\,\overline{\mathrm{B}} & \overline{\mathrm{AB}} = \overline{\mathrm{A}} + \overline{\mathrm{B}} \end{array}$$

The bottom two are known as DeMorgan's theorem. All four of these can be derived by constructing the truth tables.

DeMogran's theorem gives an equivalence between AND and OR gates, as shown in figure 13. Note the small circle on the inputs to gates means negating (inverting) the signal before it goes into the gate.

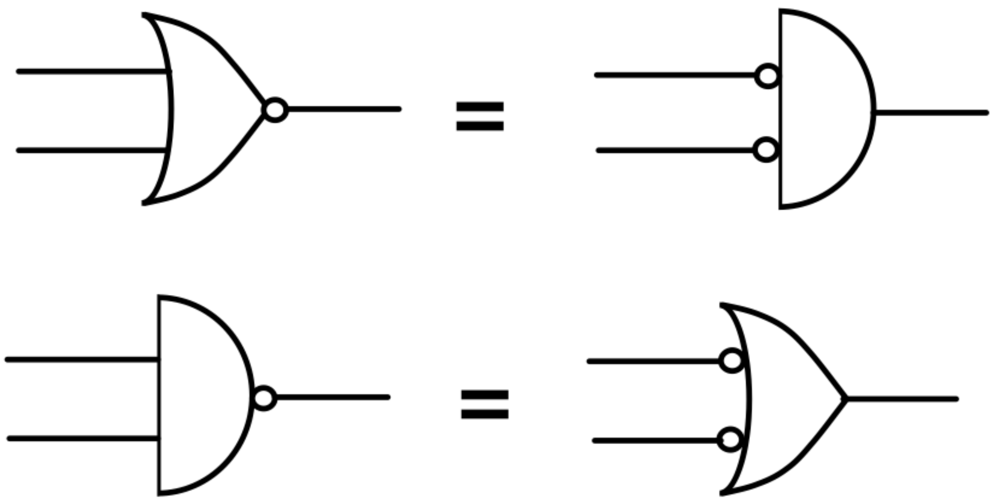


FIGURE 13. DeMorgan's theorem shown for gates.

5.2. **Example.** Boolean algebra can be very useful for simplifying logic expressions (and hence simplifying circuits). An expression may be derived from what the designer knows he wants or from a truth table. Applying Boolean identities to simplify the expression can reduce the number of gates needed in the circuit. Note that there is no exact prescription for doing this - experience must be your guide. Also note that XOR's are not natural to handle in Boolean algebra.

As an example, suppose, we have the following truth table.

A	B	C	Y
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

The first step is to write an expression for each row that has an output of 1. Then these expressions are or'ed. Finally, the expression is simplified. Applying this to this truth table gives

$$\begin{aligned}
Y &= \overline{A}\,\overline{B}\,\overline{C} + \overline{A}\,\overline{B}C + \overline{A}B\overline{C} + \overline{A}BC + A\overline{B}\,\overline{C} + A\overline{B}C \\
&= \overline{A}\,\overline{B}(\overline{C}+C) + \overline{A}B(\overline{C}+C) + A\overline{B}(\overline{C}+C) \\
&= \overline{A}\,\overline{B} + \overline{A}B + A\overline{B} \\
&= \overline{A}(\overline{B}+B) + A\overline{B} \\
&= \overline{A} + A\overline{B} \\
&= \overline{A} + \overline{B} \\
&= \overline{AB}.
\end{aligned}$$

Thus, this truth table can be implemented in one NAND gate.

This example was chosen to simplify greatly to illustrate the point. You might be able to deduce the answer straight from the truth table. However, in more realistic examples, particularly ones with more inputs, getting the simplest circuit by inspecting the truth table is usually quite difficult.

6. Digital Representations

6.1. **Single Digital Signal.** A single digital signal can be either high or low. These two states can be thought of as any two things the designer wishes. Common examples are on or off, true or false, high or low, and 1 or 0. However, they can be anything, such as, elephant or lion.

The behavior of digital circuits is well defined by the low and high states. What this means to the designer or user depends on the interpretation assigned.

6.2. **Binary.** A set of digital signals can easily represent binary (base two) numbers. Each signal (bit) has a value of 0 or 1 and represents one digit of the binary number. The least significant bit (LSB) represents 2^0, the next bit represents 2^1, the next 2^2, etc. until the most significant bit, which represents 2^{N-1}, where N is the number of bits.

For example, a set of digital signals might represent the binary number 1100101001_2 (the subscript indicates base 2). This number has a decimal value given by

$$1100101001_2 = 2^9 + 2^8 + 2^5 + 2^3 + 2^0 = 512 + 256 + 32 + 8 + 1 = 809_{10}.$$

Note that one becomes very facile with powers of two with working with digital number representations.

It is also easy to convert a number in decimal (base 10) to binary. You first subtract the largest power of two that is equal or smaller than the number. You then continue this process with the remainder until the final remainder is zero. The binary digit corresponding to each power of 2 subtracted is then 1. For example,

$$\begin{aligned}186_{10} &= 128+58\\ &= 128+32+26\\ &= 128+32+16+10\\ &= 128+32+16+8+2\\ &= 10111010_2.\end{aligned}$$

Each binary digit is known as a bit. A set of 8 bits is known as a byte.

6.3. **Hexadecimal.** Binary numbers can have many digits. It is often easier to represent them in hexadecimal (base 16, usually just called hex). Since there are 16 basic digits in hex (0 to 15_{10}), we need six additional symbols to represent the digits corresponding to 10_{10} to 15_{10}. These symbols could be anything we invent, but the convention is to use the letters A to F (or lower case).

Since 16 is a power of 2, it is very easy to convert from binary to hex and vice versa. To convert from binary to hex, just group the binary digits in groups of 4 starting with the LSB. Each group represents one hex digit. For example,

$$185_{10} = 10111001_2 = (1011)(1001) = (11_{10})(9_{10}) = \mathrm{B}9_{16}.$$

Convert from hex to binary is just as easy - just expand each hex digit to its 4 binary bits (don't drop any leading zeroes). For example,

$$\mathrm{C}4_{hex} = (1010)(0100) = 10100100_2.$$

Note that 2 hex digits are exactly a byte.

6.4. **Octal.** Occasionally, people dealing with binary number like to use octal (base 8). This is just like hex, except that 3 bits correspond to one octal digit. Since 8 is less than 10, there is no need for additional symbols (and the digits 8 and 9 don't appear in octal). Octal has the disadvantages that it is not as compact as hex and one byte doesn't contain an integer number of octal digits.

6.5. **BCD.** While hex and octal are convenient when dealing with digital representations of numbers, most people do not naturally think in these terms. We are all more comfortable in base 10. Thus, it is sometimes more easy to represent numbers in BCD (binary coded decimal), where each decimal digit is represented by 4 bits. For example,

$$185_{10} = (0001)(1000)(0101) = 000110000101_{\mathrm{BCD}}.$$

Since 4 bits can represent any number from 0 to 15_{10}, there are some combinations of bits that have no meaning in BCD (such as 1100). BCD is convenient to us 10-fingered species, but is wasteful of bits and is not as easy to implement in computational circuits.

First hex digit → Second hex digit	0	1	2	3	4	5	6	7
0			blank	0		P	‘	p
1			!	1	A	Q	a	q
2			"	2	B	R	b	r
3			#	3	C	S	c	s
4			$	4	D	T	d	t
5			%	5	E	U	e	u
6			&	6	F	V	f	v
7	BELL		’	7	G	W	g	w
8	BACKSPACE		(	8	H	X	h	x
9	TAB		)	9	I	Y	i	y
A	LINEFEED		*	:	J	Z	j	z
B			+	;	K	[	k	{
C			,	<	L	\	l	|
D	RETURN		-	=	M	]	m	}
E			.	>	N	ˆ	n	˜
F			/		O	_	o	DEL

Table 1. A partial table of Ascii codes.

6.6. **Others.** There are other numerical representations, such as excess 3, Gray codes, and 2’s complement. I will not go into any of these here, but if in your future digital life you need them, you can look them up in any standard reference.

6.7. **Alphanumeric.** Sometimes, we wish to represent text as digital signals. We do this by assigning a number to each character. Since there are 26 lower case letters (a, b, c, ...), 26 upper case letters (A, B, C, ...), 10 numerical digits (0, 1, ..., 9), and several punctuation marks (blank, ., !, #, $, &, *, (,), etc.), there are at least 75 characters we would wish to represent. This means we need at least 7 bits (since 7 bits covers the values from 0 to 2^7-1 = 127).

We could assign this numbers to characters in anyway we like, and there have been many such systems. However, today there is one that is much more common that others known as ASCII. It uses 8 bits and uses the extra numbers to represent other special characters. The standard Ascii codes for the usual characters are given in table 1. Note that each character requires 8 bits, that is, 1 byte or two hex digits.

7. Negative Numbers: Sign-Magnitude

One of the more straight-forward ways to represent negative numbers is called sign-magnitude. In this representation, one bit (usually the most significant bit) gives the sign of the integer, and the other bits give the magnitude. For example, if we are using 8-bit integers, the numbers 14_{10} and -14_{10} are represented by

$$\begin{aligned} 14 &\rightarrow 00001110 \\ -14 &\rightarrow 10001110. \end{aligned}$$

Although sign-magnitude representation is conceptionally easy, it is difficult to do digital calcuations in this representation. Note, for example, that if you add the representations for 14 and -14, you don't get anything like zero, which is what you would like.

8. 1's Complement

In 1's complement representation, you find the negative of a number by taking the bit-by-bit complement, that is, each 0 in the number is changed to a 1 and each 1 to a 0. In this representation,

$$\begin{aligned} 14 &\rightarrow 00001110 \\ -14 &\rightarrow 11110001. \end{aligned}$$

In this representation, the most significant bit also indicates the sign (0 for positive, 1 for negative). If you sum the representations of a number and its negative, you get all one's (for example, 11111111 for an 8-bit number). This is not zero (as we would like), but it is close, since we just need to add 1 and ignore the carry. This is the basis for the next representation.

9. 2's Complement

In 2's complement, you first take the 1's complement of a number and then add 1 (ignoring any carry out bit). For example,

$$\begin{aligned} 14 &\rightarrow 00001110 \\ -14 &\rightarrow 11110001 + 1 = 11110010. \end{aligned}$$

Here again, the most significant bit will be 0 for positive numbers and 1 for negative numbers. Although it isn't obvious, the 2's complement has the property (likes 1's complement) that if you take the negative of a number twice, you get back to the original number:

$$-(-14) \rightarrow 00001101 + 1 = 00001110 = 14.$$

The major advantage to 2's complement is that digital calculation is much easier. To subtract a number, you take the 2's complement and then do a normal binary addition (ignoring any carry bit). For example,

$$14 - 14 \rightarrow 14 + (-14) \quad \rightarrow \quad \begin{array}{r} 00001110 \\ +11110010 \\ \hline 100000000 \end{array} \rightarrow 0$$

$$53 - 10 \rightarrow 53 + (-10) \quad \rightarrow \quad \begin{array}{r} 00110101 \\ +11110110 \\ \hline 100101011 \end{array} \rightarrow 43$$

For 8-bit numbers, there are 7 bits available to represent positive numbers, that is, 0 to 2^7-1 = 127 can be represented. Negative integers range from -1 (11111111) to -127 (10000001). This leaves one bit combination that isn't assigned, namely, 10000000. This combination is usually assigned to be -128, although you should note that if you 2's complement this representation, you get -128 (not 128). For N bits, the range of integers that can be represented is -2^{N-1} to 2^{N-1}-1.

Since there is a finite range of integers that can be for a given number of bits, it is possible to start with two integers that are within the range and end up with an integer that is outside the range (for example, 64 + 64 = 128). This is called an overflow or underflow. Most computers use at least 16 bits for integers and usually use 32 bits. In many computer languages, it is possible to specify that integers have more bits (such as 64).

2's complement representation also works for multiplication and divisiion. For example, consider 5 times 11. In 2's complement, this multiplication looks like

$$\begin{array}{r} 1011 \\ 101 \\ \hline 1011 \\ 1011 \\ \hline 110111 \end{array} \rightarrow 55.$$

The multiplication of 5 times -11 ($\rightarrow$ 11110101) looks like

$$\begin{array}{r} 11110101 \\ 101 \\ \hline 11110101 \\ 11110101 \\ \hline 10011001001 \end{array} \rightarrow -55.$$