Chapter 5

Representing Stuff with Bits: Coding

A big part of electrical engineering is thinking about the electrical transfer of information. This information is often in the form of numbers, but what is the best way to go about sending this information? Computers are digital machines that are have transistors that can perform logical operations and store information, but transistors only store two possible values, True or False. If this is the case, how do we send the value "3"? Or "1,000,000"? Or even "3.1415?" How do we even communicate images, music, or text?

There needs to be a set standards, or **codes**, of which we can communicate between computers. A quick search on Google on the definition of codes gives us:

- a system of words, letters, figures, or other symbols substituted for other words, letters, etc., especially for the purposes of secrecy.
- a system of signals, such as sounds, light flashes, or flags, used to send messages.
- a series of letters, numbers, or symbols assigned to something for the purposes of classification or identification.

In computers, the series of symbols that we use to send messages are going to be in the form of **binary bits**.

A single bit can either be 0 or 1, but doesn't really communicate much, so what needs to be done is that a collection of bits will be necessary to send more complicated information.

5.1 Unary Code

One implementation of sending only two possible values is to have a bit for each value you wish to present. For example, if you a code with 4 bits, each one of

those bits could represent. This is called **unary coding** or **one-hot coding** because one bit is on at a time.

Number	Unary Code
0	0001
1	0010
2	0100
3	1000

Table 5.1: Unary Code for a 4-bit System

The advantage of this system is that it's easy to use the code directly to control individual hardware. If there were four LEDs that could needed to turned on sequentially such that only one LED is on, one-hot coding could be used to directly control the LEDs. This will actually be done for the LED cude lab.

The most obvious downside is that the solution is not scalable as you go to higher numbers. The number of bits you have is the maximum number that you represent, so if you need to represent a very large number, you'll need a very large number of transistors.

5.2 Binary Numbers

Binary coding is a more efficient way of storing integer value and is very similar to how humans represent numbers.

A majority of the people in the world use a **decimal** or **base-10** numbering system. This means that we represent each digit in the number can have potentially 10 different values, which are 0-9.

To get the actual final quantity of a number, for example 145, we start on the right most digit of the number which is 5, multiply by 10^0 (which is just 1). Then we move left to the next digit 4, multiply by 10^1, then add it to 5, which we get 45. Then we move left again to the digit 1, multiply by 10^2, then add it again to our accumulating sum, which we finally get 45.

10^2	10^1	10^0
1	4	5

Table 5.2: Decimal 145

$$145 = 10^2 * 1 + 10^1 * 4 + 10^0 * 5$$

We can technically continue to do this as we continue to add more and more digits, and the more digits we have, the larger the number we can represent.

Now that we see how a base-10 system works, we can do the same thing with bits in a base-2 system. Instead of multiplying each digit value by 10 to the power of the digit index, we multiply each bit value by 2 to the power of the bit index.

For example, let's take the 8-bit value 00101010. To convert this value into a base-10 number that makes sense to humans, we can do the same process in base-10 to determine the actual value of this binary value. Remember, we start at the right most bit. Then as we shift left, we multiply that value by 2 to the power of the number of bits we've shifted from the right most bit, then accumulate that sum.

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
0	0	1	0	1	0	1	0

Table 5.3: Binary 00101010

$$42 = 2^7 * 0 + 2^6 * 0 + 2^5 * 1 + 2^4 * 0 + 2^3 * 1 + 2^2 * 0 + 2^1 * 1 + 2^0 * 0$$

This is great because in unary coding, 8-bits can only represent 8 numbers, but with binary, the largest number that can be represented is 11111111, which is $2^7 * 1 + 2^6 * 1 + 2^5 * 1 + 2^4 * 1 + 2^3 * 1 + 2^2 * 1 + 2^1 * 1 + 2^0 * 1$, or 255.

Now, let us try converting from a base-10 value into binary. What if we wanted to represent the number 23 in 8-bit binary?

The process is to with the largest bit index with the right most bit index being 0. Divide the number by 2 to the power of that index value, which will be 2^7. The number we get as a result is the bit value for that index (should be either 0 or 1) and then the remainder gets passed down to the next step. In this case, the value is 0 and the remainder is 23.

Now the index value is 6, meaning

Operation	Output	Remainder
$23/2^7$	0	23
$23/2^6$	0	23
$23/2^5$	0	23
$23/2^4$	1	7
$7/2^3$	0	7
$7/2^2$	1	3
$3/2^1$	1	1
$1/2^0$	1	0

Table 5.4: Converter Decimal 23 to 00010111

Problem 5.1 : Representing numbers in binary

1. Find the 4-bit binary representation of 15.
2. Find the 8-bit binary representation of 15.
3. Find the 8-bit representation of 101.

5.3 Arithmetic

Addition and subtraction in binary are very similar to their decimal counterparts. You add by carrying 1s and subtract by borrowing 2s (instead of borrowing 10s).

For example, let's examination how addition works when adding 11 and 3, which are 1011 and 0011 in binary respectively:

1	0	1	1
0	0	1	1

Table 5.5: Addition step 1

		1	
1	0	1	1
0	0	1	1
			0

Table 5.6: Addition step 2

	1	1	
1	0	1	1
0	0	1	1
		1	0

Table 5.7: Addition step 3

We get the final result 1110, which is converted to 14 in decimal. This checks out because 11+3=14.

And, for subtraction, we will subtract 9 and 3:

In the end, we get the binary value 0110, which is 6 in binary. This is correct because 9-3=6.

	1	1	
1	0	1	1
0	0	1	1
	1	1	0

Table 5.8: Addition step 4

	1	1	
1	0	1	1
0	0	1	1
1	1	1	0

Table 5.9: Addition step 5

1	0	0	1
0	0	1	1

Table 5.10: Subtraction step 1

1	0	0	1
0	0	1	1
			0

Table 5.11: Subtraction step 2

	2		
~~1~~ 0	0	0	1
0	0	1	1
			0

Table 5.12: Subtraction step 3

	~~2~~ 1	2	
~~1~~ 0	0	0	1
0	0	1	1
			0

Table 5.13: Subtraction step 4

	~~2~~ 1	2	
~~1~~ 0	0	0	1
0	0	1	1
		1	0

Table 5.14: Subtraction step 5

	~~2~~ 1	2	
~~1~~ 0	0	0	1
0	0	1	1
	1	1	0

Table 5.15: Subtraction step 6

	~~2~~ 1	2	
~~1~~ 0	0	0	1
0	0	1	1
0	1	1	0

Table 5.16: Subtraction step 7

Problem 5.2 : Binary arithmetic

Compute the following binary arithmetic problems. You can check your answer by converting your binary solution into decimals.

1. 0010 + 0001 = ?
2. 00001111 + 01100101 = ?

5.3.1 Integer Overflow

It turns out that computers have a limited number of places to store numbers. The number of bits (places) that a computer has to store a number depends on the computer. Common numbers of bits computers use include 8, 16, 32, and 64. For example, Arduino Nanos use 16 bits to store integers.

There is a maximum value that can be stored in an integer. For example, the largest 4 digit decimal number is 9999. Analogously, the largest 4 bit binary number is $1111 = 2^0 + 2^1 + 2^2 + 2^3 = 1 + 2 + 4 + 8 = 15$.

It turns out that the maximum value that can be stored in an n bit binary number is equal to $2^n - 1$ (this can be shown by using the formula for a sum of a geometric sequence). Thus, the largest **unsigned** integer that the Arduino can store is $2^{16} - 1 = 65535$.

This limited number of places to store numbers can cause problems when we want to get numbers larger than the maximum or smaller than the minimum.

For example, when we add 1 to the maximum integer, we get **integer overflow**.

1	1	1	
1	1	1	1
0	0	0	1
0	0	0	0

Table 5.17: Integer overflow. $15 + 1 = 0$?

This problem stems from the fact that the leftmost value place has a carried 1 that can't be added.

Similarly, when we subtract 1 from the minimum integer, we get **integer underflow**.

~~2~~ 1	~~2~~ 1	~~2~~ 1	2
0	0	0	0
0	0	0	1
1	1	1	1

Table 5.18: Integer underflow. $0 - 1 = 15$?

This problem stems from the fact that the leftmost value place has to borrow a 2 that doesn't actually exist.

Problem 5.3 : More binary sums

1. $1111 + 0101 = ?$
2. $11110001 + 00001111 = ?$

5.4 Negative Numbers

So far we've developed a system to represent non-negative integers, but what if negatives numbers are needed?

One solution is to dedicate the most significant bit to the sign value and set that value to 1. For example, the number 3 in 8-bit can be represented as 00000011. If we wanted to represent -3, we flip the sign bit, giving us 10000011. Notice that while this solution is easy to flip the signs.

The downside however is that whenever we need to perform arithmetic on these values, we would need to look at the sign bits to see if we need to add or to subtract and that would slow down the operation. So we don't do that, we use two complement numbers instead.

5.4.1 Two's complement

As discussed in the earlier section, overflow is observed when the final number goes beyond the upper of lower bounds of the number.

Since numbers wrap around due to integer overflow and underflow, we can just designate that certain numbers are negative, and use modulo (or for 4 bit numbers, subtracting 16) to determine negative numbers!

This concept of using this circular wrap-around for negative numbers is known as **two's complement**.

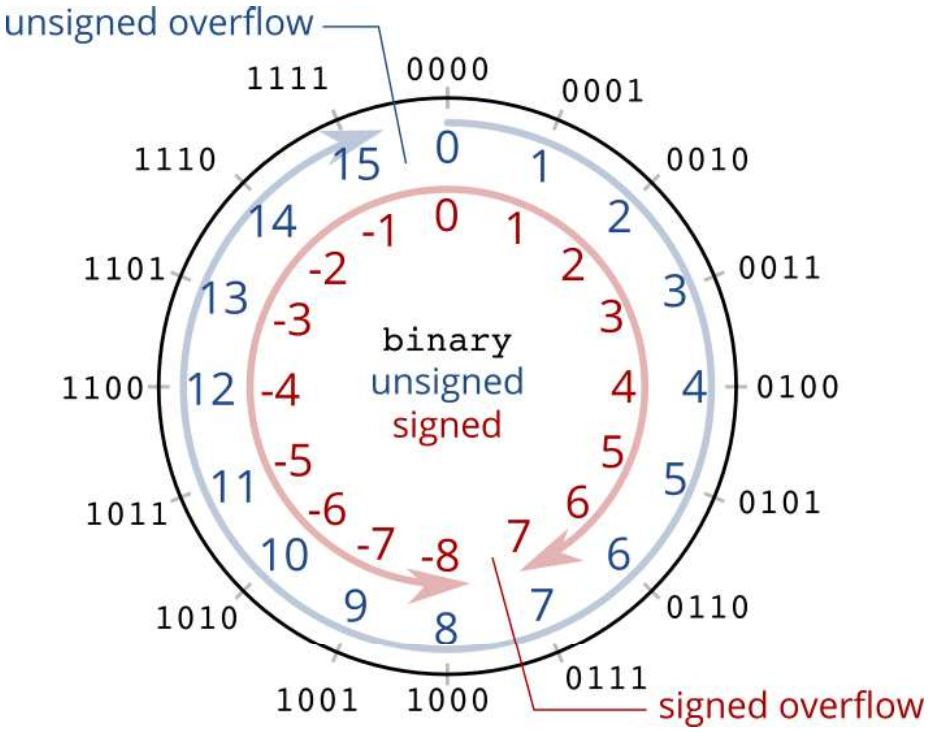

Figure 5.1: Two's complement circle

Since we don't want the bits interpreted to be both -2 and 14 at the same time, we're going to define two conventions for interpreting bits: unsigned and signed.

Unsigned convention is exactly what you've seen in the previous section (binary without any negative numbers). Overflow and underflow occur at the maximum integer and 0.

Signed convention is a little bit different. We want a way to represent negative numbers, so all numbers whose most significant bit (the leftmost bit) is 1 will be negative. This also means that overflow and underflow occur at a different location (see the two's complement circle).

Binary Code	Unsigned Value	Signed Value
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	-8
1001	9	-7
1010	10	-6
1011	11	-5
1100	12	-4
1101	13	-3
1110	14	-2
1111	15	-1

Table 5.19: Binary Code for a 4-bit System, both Signed and Unsigned

For example, consider the interpretation of binary 1011.

Unsigned: We get $2^0 + 2^1 + 2^3 = \mathbf{11}$.

Signed: We have unsigned 11, but the most significant bit is 1. Thus, we subtract $2^{\text{number of bits}} = 2^4 = 16$, so we get $11 - 16 = \mathbf{-5}$.

And binary 0111:

Unsigned: We get $2^0 + 2^1 + 2^2 = \mathbf{5}$.

Signed: We have unsigned 5, and the most significant bit is 0. Thus, we have **5**.

Also, changing the sign (multiplying by negative 1) is easy: just invert all the bits and add 1.

1. Decimal 5 = Binary 0101
2. Invert bits: 1010
3. Add 1: 1011
4. Decimal -5 = Binary 1011
5. Invert bits: 0100
6. Add 1: 0101

7. Decimal 5 = Binary 0101

It turns out that, under two's complement, the most positive number is a 0 followed by all 1s, while the most negative number is a 1 followed by all 0s.

Problem 5.4 :

1. Represent the following numbers in 4-bit two's complement form, then in 8-bit two's complement form.

   - 5
   - -5
   - 8
   - -8

2. Represent the following numbers in 8-bit two's complement form.

   - 15
   - -15

5.4.2 Two's Complement Arithmetic

The great advantage with Two's Complement is that arithmetic is much easier. No matter the sign, the number are simply added together. If two numbers need to be subtracted, we simply perform the operation to turn the number we wish to subtract away into a negative number, and then add the numbers together.

Using a 4-bit system, let's first compute 2 - 6 in binary. We should get -4 at the end, but let's go through the process in binary.

We establish first that:

- 2 is 0010 (2^1)
- 6 is 0110 ($2^2 + 2^1 = 6$).

Next we need to convert 6 into -6, and we do this by:

- inverting the bits: 1001
- add one : 1010

Finally, we can perform the addition of 0010 and 1010, which we should get 1100.

What is 1100 in decimal? Let's converter this back into a positive number and see

- invert bits: 0011

- add one: 0100

0100 is 4 (2^2), and this confirms our answer and that two's complements works.

5.4.3 Overflow in Two's complement

Notice that arithmetic overflow can still exist in a Two's complement system. The largest positive number is 0 followed by only 1's, so for an 8-bit system, it would be 01111111. If you add 1 to that value, you will get 1000000, which is a negative number in two's complement. This is overflowing past the upper bound.

Furthermore, the largest negative number is 1 followed by only 0's, so for an 8-bit system, it would be 10000000. If we add negative one to this value (00000001 is 1, inverting gives us 11111110, adding one gives us 11111111), will give us 01111111 where we lose an extra 1 due to overflow. This an example of underflowing past the lower bound.

Be very cautious when performing arithmetic because it's possible to go past the upper and lower bounds. The maximum number of positive values expressable by a fixed number of bits in a two's complement system compared to an unsigned integer system has halved because those values are not being used to represent negative numbers.

5.5 Real numbers

To be done in the future. This section will talk about floating point number representation.

5.6 Representing Other Types of Data

Now that representing integers is possible with bits, we can take this idea further and use binary numbers to represent other values.

5.6.1 Characters

There are two common ways to encode characters. The first is ASCII (American Standard Code for Information Interchange). They use 8-bit numbers and assigned letters, numbers, and other commands like "enter", "backspace", etc. Since the numbers are 8 bits, you represent 2^8=255 values. Check out the link http://www.asciitable.com/ to see what

The other is Unicode and has two encoding standards. While UTF-8 uses 8-bit and contains the similar encoding structure to ASCII, UTF-16 uses 16 bits and can contain 65536 values. This encoding scheme include English characters and other characters form Arabic, Chinese, Spainish, and many other languages.

The link of converter binary numbers to unicode value is here http://unicode-table.com/en/.

It is imperative that both machines that are communicating in characters need to both agree on a standard and not mix them up.

5.6.2 RGB Colors

All colors can be represented in the combination of red, blue, and green (RGB) light of various intensities. While the eye has a very large dynamic range, our screens/projectors are more limited, and generally 8 bits for each color are enough. Cameras can record from 8 to 14 bits of intensity resolution per color, depending on the quality (and size) of the light sensor. In std JPEG images, these are then compressed into 8bit values. Thus most colors consists of 3 8 bit values.

5.7 Error Correcting Codes

When sending digital information through a communication channel, all packets of data will be transferred and received successfully in a perfect world, but this is not always the case in reality. Various reasons such as noise, outside interference, or simple system glitches causes information to be lost at some probability. For example, if I send N packets of data through a stream, I may lose K packets. The links where we lose data are called **erasure channels**.

Because of this limitation, how much data can we send in N packets if we lose at most K packets? For every piece of data, we need to send it at least K+1 times. That way, the worst-case scenario would be if the K packets of data that is lost are the same copies of data, leaving one copy left. Therefore, if we can send 12 packets of data at once and we can potentially lose 2 pieces of data, we need to replicate all data at least 3 times, meaning we can only send $12/3 = 4$ packets of data at once that is copied three times.

This seems very limiting, since we duplicate all the data, which means that most of the data being recieved will be copies of data that has already been received. Perhaps there is a better way of sending data?

We can easily determine the upper bound of data that can be transmitted through this link is (N-K), or the number of packets that are successfully sent through the communication channel. However, in order for the reciever to get this number of bits, each packet of data that gets through must have unique information, which is hard to understand how we can do this, if we don't know which packets are going to be lost.

This seems like an impossible problem, until you realize that there no reason you can't send multiple pieces of data in the same packet. So rather than sending one packet of information each time, you can send multiple packets by adding them together.[1] Now we can make sure each data is replicated in

[1] If you are paying close attention you will realize that adding packets together might use more bits than before, since it could generate bigger numbers. When this is done in practice,

multiple packets, while at the same time ensuring that each packet is unique (up to N-K).

Let's do an example of this:

Say that we need to send the numbers X1, X2, and X3 through a communication channel that sends 5 packets but can lose any two packets. If we send the following 5 packets, it's possible to recover X1,X2 and X3 no matter what 2 packets we lose:

- X1
- X2
- X3
- X1+X2+X3
- X1+2*X2+3*X3

For example, if we only received the values, X1, X2, and X1+X2+X3, we can recover X3 by subtracting X1 and X2 from the final value.

What if we only received X1+2*X2+3*X3, X1+X2+X3, and X3? We can simply perform (X1+2*X2+3*X3) - (X1+X2+X3) - 2*(X3) to get X2. Then using X3 and X2, we can get X1 from the second value.

Bottom line is that if we as long as we have 3 linearly independent equations with 3 unknowns, we can easily determine the three values.

This shows that if we can send N packets and we lose at most K packets, we can acquire N-K=M packets. Another way of thinking about how to solve this problem is by using linear algebra to create our new values that are a function of the other pieces of data. If X is our input data and Y is the data that we send, we can perform Y = AX where A is a NxM matrix.

$$Y = AX$$

$$\begin{bmatrix} x1 \\ x2 \\ x3 \\ x1+x2+x3 \\ x1+2*x2+3*x3 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 1 \\ 1 & 2 & 3 \end{bmatrix} \begin{bmatrix} x1 \\ x2 \\ x3 \end{bmatrix}$$

Finally, to convert the original values back when we lose data, we take the rows from A of the data that we recovered, invert the new matrix, then multiply it by Y.

$$X = A^{*-1}Y^{*}$$

Where * refers to the data you've received. If you lose the data x2 and x3, the procedure you perform will be:

---

the addition is done modulo a large prime number, so the numbers don't really get larger, but to really explain this I would need to explain properties of modulo arithmetic which is really outside the scope of this class.

$$\begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 1 \\ 1 & 2 & 3 \end{bmatrix}^{-1} \begin{bmatrix} x1 \\ x1 + x2 + x3 \\ x1 + 2 * x2 + 3 * x3 \end{bmatrix} = \begin{bmatrix} x1 \\ x2 \\ x3 \end{bmatrix}$$

5.8 Time Multiplexed Codes

Imagine you have an LED screen that has 1920 x 1080 resolution, meaning that is the number of pixels in the X and Y dimension. Each pixel has an RGB value, so has 3 values for each pixel, and each value will require 8 bits How do we go about controlling each pixel in the screen?

The first approach would be to wire each individual pixel and color to a control circuitry, but is not a very scalable solution. For the 1920 x 1080 monitor, the number of pins needed to power the screen will be 1920 * 1080 * 3 * 8 or around 48 million bits. This doesn't seem very scalable. Moreover, we will need a 48 million transistors to control every pixel in the screen, meaning we'll need to use a 48 million bit long binary number.

How do we go about solving this issue? The issue is that if we send the data all at once, the amount of transistors and wiring needed to do this will enormous. How about instead of sending everything all at once, what if we send data sequentially in time?

This concept is called **Time Multiplexing**, or **Serial Communication**. Many common communication protocols take advantage of this concept, such as Ethernet, USB, I2C, SPI, JTAG, and many more. The number of wires needed to transmit data is relatively small because the data sent through those communication channels is sent as packets over time.

Case Study: LED array

A 4 by 4 by 4 LED cube is constructed for the third lab. If we controlled each LED individually we would need to use 4*4*4=64 wires for a pin to LED connection. Our Arduino doesn't have that many pins, so how do we use time multiplexing to reduce our pin count?

The first point to make is that an LED can only turn on when there is current flowing through the LED. This is only possible when the positive end of a voltage source is connected to the LED from the Anode end and the negative end is connected to the Cathode end.

The second point is that we don't need to have all of the LEDs on at once. Suppose we can only turn on only one LED on at once, rapidly alternating the LEDs we turn will trick our eyes into thinking that all of those LEDs are on. This concept is called optical persistence and is the trick to using time multiplexing to turn on LEDs serially. In the cube we will use this idea, but will turn on 8 LEDs at a time, so we only need to have 8 time slots.

Note that optical persistence only works if you are turning on and off the LEDs fast enough. If not, then your eyes will be able to see the LEDs flashes.

Let's examine a row of four LEDs and connect all of the anodes together to a positive voltage as shown in Figure 5.2. If we wanted to turn on the first LED

from the left, we need to allow current to flow through the LED by connecting its cathode low (of course through a current limiting resistor that is not shown) and leave the other cathodes connected high. Similarly, if we wanted to turn on just the second LED, set cathode of the second LED low and set the other cathodes high.

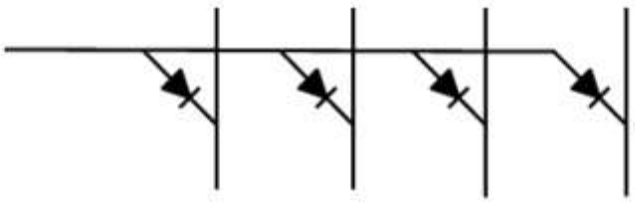

Figure 5.2: One row of LEDs

Finally, if we wanted to turn on the 1st and 3rd LED, we can set both the first and third cathode low (again using a current limiting resistor on each cathode). Right now, we can control 4 LEDs with 4 pins, but what happens when we start connecting multiple rows together? Let's do this such that all LEDs in the same row have their anodes connected and all LEDS in the same column have their cathodes connected. See Figure 5.3 for the schematic.

Notice with this configuration we can control 16 LEDs with 8 pins. Four pins control the rows while the other four control the columns. In this design it is critical that only one row is driven to Vdd at a time, while any column can be driven low to turn on the desired light in that row.[2] If you notice in Figure 5.3, there are four LEDs that are in the array. How would we go about lighting up those LEDs by using Time Multiplexing?

The driver will sequentially turn on T0 for a short time, and then rotate through T1-T3 for the same time each before returning to T0 and repeating the cycle.

Let's look at what happens when T3 goes high and everything else is low. Clearly the LEDs connected to T0, T1, and T2 will be off, since their anodes will not be able to support any diode current. For the top row, there will a path from a high voltage to a low voltage through the four LEDs on the top row of the array. In other words, current can flow from T3 to N0, N1, N2, and N3, which causes all four of those LEDs to turn on.

However, what if we only want to have the second LED light up, like what is show in the figure? In order to prevent current flow from T3 to N0, N2, and

[2]Note that we just chose to drive the anode sequentially, since in our design they are driven by external pMOS devices and can drive more current. It is possible to drive the cathode column low one at a time, and then drive any of the rows high to light LEDs in that column, but then we would need to have external nMOS transistors driving the columns to support the high column current.

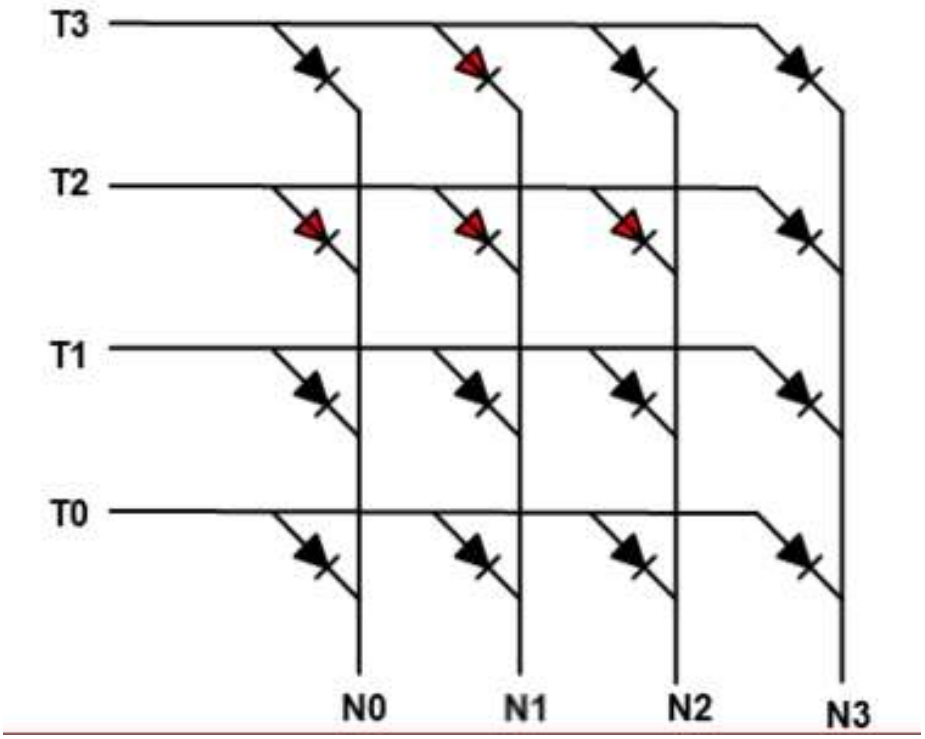

Figure 5.3: Four rows of Time Multiplexed LEDs

N3, we will need to set those pins high. That way, both ends of those LEDs will be high, meaning there is no voltage drop across those LEDs, and it will no be on.

Furthermore, if a row is low and a column is high, those LEDs won't turn on either because these LEDs can only function with current going in one direction. If the cathode has a higher voltage than the anode, no current will flow because the LED is a diode, and the LED will be off.

As a summary, in order to turn on the LED on the first row and second column, we set T3, N0, N2, and N3 high and we set T0, T1, T2, and N1 low.

Overall, we turn on an LEDs by having the anode must be switched high and the cathode must be switched low. This is done by setting your target row or anodes high and your target column of cathodes low.

During the next period T0 is driven high, but since we don't want any of those lights on, all the columns N0-N3 are driven high. Then T0 is driven low and then T1 is driven high, and again all the columns must be high to keep these lights off. During the next period T2 is driven high, so N0-N2 are driven low, and only N3 is driven high to light all but the right LED in that row.

Case Study: Keyboard

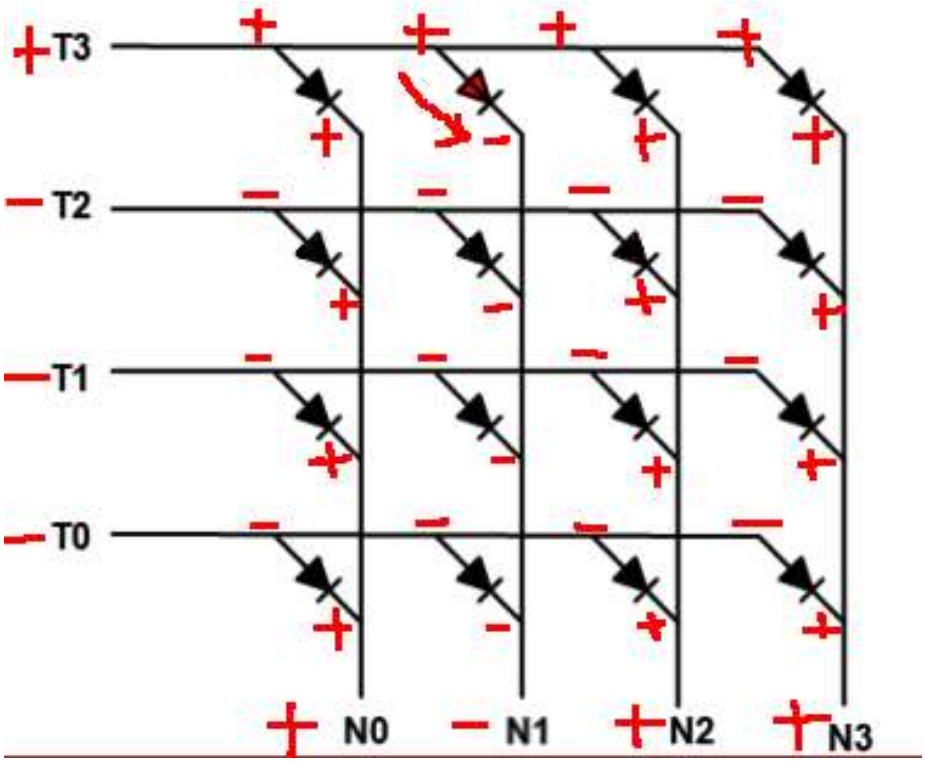

Figure 5.4: Turning on LED on row T3 and Column N1. Polarity is noted.

Now that we can figure how to output multiple things by multiplexing, what if we engineered something that can input multiple things by multiplexing? An example of something that takes a lot of inputs is a keyboard, where is has a lot of buttons that need to be read.

Similarly enough, we can perform time multiplexing to determine whether a key is pressed down or not without having to wire every single key to a wire.

Just like before, let's lay down a row of buttons.

Let T3 be the input pin to a microcontroller, and let's make it be a pulled-down pin, meaning it's default input value is low. In order to trigger a reading of the button, the button needs to short the input pin high.

If we initially set N0-3 low, no matter what button we press, the switches will always short T3 to ground.

But what happens when we set N0 and only N0 high? As long as N0 gets pressed, T3 will get pulled high, but if N0 isn't pressed, T3 stays low. If we register a high on T3 when we apply high on N0, we know the switch connecting N0 and T3 got pressed. Also, notice that if we pressed the other switches, nothing happens because T3 will still get connected low.

Next, we quickly set N1 and only N1 high. If the switch connecting T3 and

Figure 5.5: Turning on LED on row T3 and Column N1. Polarity is noted.

N1 gets pressed, then T3 gets pulled high. However, pushing the others switches don't do anything to T3.

What this shows is that if we rapidly select pins N0-3 to go high individually, we can read which button gets pressed. If we read them rapidly enough, we can detect which of the four buttons are pressed.

As a result, we can do exactly the same thing we did for the LED array which is to connect all the columns together to the same pins to get an array of switches that can be sensed.

5.9 Solutions to practice problems

Solution 5.1:
1.)The most significant bit of a 4-bit binary representation has a value of $2^3 = 8$. From then, the next bits have values of $2^2 = 4, 2^1 = 2, 2^0 = 1$ respectively.

$$15 = 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0$$

Therefore, the binary representation of 15 in 4-bit binary is: **1111**.
2.)The 8-bit binary representation of 15 would be **00001111**.
3.)The 8-bit binary representation of 101 is **01100101**.

$$101 = 0 \cdot 2^7 + 1 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$$

Solution 5.2:
1.) 0010 + 0001 = 0011; In decimal: 2 + 1 = 3
2.) 00001111 + 01100101 = 01110100; In decimal: 15 + 101 = 116

Solution 5.3:
1.) 1111 + 0101 = 10100; In decimal: 15 + 5 = 20
2.) 11110001 + 00001111 = 100000000; In decimal: 241 + 15 = 256

Note that in both cases overflow occurs, and we had the extend the number of bits we are using. In a real computer this often isn't possible (for example, your Arduinos have a limited number of bits you can use for numbers), and we would simply overflow (wrap around). In that case, question would wrap to 5, and question two would wrap to 1.

Solution 5.4:

1. The 4-bit two's complement form and 8-bit two's complement form of the numbers are as follows:

   - 4-bit: 0101 8-bit: 00000101
   - 4-bit: 1011 8-bit: 11111011
   - 4-bit: You can't! 4-bits is not enough to represent 8 in two's complement - the highest we can go is 7. 8-bit: 00000111

- 4-bit: 1000 8-bit: 11111001

2. The 8-bit two's complement form of the numbers are as follows:

   - 00001111
   - 11110001

Observe the difference between the positive of a number and the negative of it. Once you've figured out how two's complement works, the rule where you can find the negative version of a number by flipping all the bits and adding 1 will always work (as long as you are still within the allowable range of the number of bits you have!)