3x3 NMOS Transistor Multiplier

By Ian Lane

Purpose

This project is being created during my Junior year at UMBC while taking CMPE 314, Principles of Electronic Circuits. In this class we have learned about the fundamentals of BJT and MOSFET transistors and how to use them under AC analysis to create AC amplifiers. My goal for this project is to take the fundamental knowledge I have learned from my class and rather than use them under AC conditions as I have already done, instead using MOSFETs in particular, use them under DC conditions such that I can use them as logical switches that can be used in VLSI.

The 3 x 3 multiplier

The 3 x 3 multiplier is unfortunately not one that is most likely ever going to be used in any VLSI design because of it only being able to handle numbers 0 through 7, however there are reasons as to why I decided to choose this over anything else.

First off, why not a simpler binary adder that can handle twice as many bits? The reason I chose to design a multiplier over an adder is because the logic required for the multiplier is more complex than that of an adder. A multiplier is essentially a multistep adder which requires more logic to handle and I not only wanted to test my knowledge and understanding of MOSFETs, but also to come up with a design, schematic, and implementation that required more problem solving.

Since I have decided I wanted to create a multiplier, why choose a 3x3 and not a 4x4 or 2x2 etc? Firstly anything smaller than a 3x3 bit multiplier has too simple of logic, even a 10x2 multiplier is essentially just a 10 bit adder and I wanted to create something more complex than that. However on the other side of things a 4x4 multiplier requires significantly more logic that a 3x3. I was originally going to create the 4x4 bit multiplier for a more significant range of numbers, however once I realized how much more logic was required I decided it was not a good idea. The main reason I decided this was not because the logic was difficult, but because of how many more transistors it would require to create. I wanted to stay within the roughly 100 transistor mark because that is all the transistors that I have. Also because I have a limited number of breadboards to be used on this project. A basic breadboard has 2 separate 62 line channels on, since every transistor has 3 terminals, it means that an absolute maximum number of transistors on one breadboard is 41, meaning I will need at least 3 breadboards just for 100 transistors. So with my constraints the 3x3 multiplier using roughly 112 transistors is the smallest multiplier I am going to create.

This multiplier is going to be made entirely out of 2n7000 MOSFETs. My original goal was to create the multiplier out of both n and p channel MOSFETs to create a stable CMOS

equivalent, however the first problem with this is that it would double the space that would need to be required from a minimum of three to six breadboards for full scale. The second major problem is that I could not find a substantial quantity of PMOS transistors, and those that I could find already cost more than the NMOS transistors for a tenth of the quantity.

Logic/Basic Concept

Table 1

				A_3	A_2	A_1
			x	B3	B_2	B_1
				A_3B_1	A_2B_2	A_1B_1
+		C_4	A_3B_2	A_2B_2	A_1B_2	0
+		A_3B_3	A_2B_3	A_1B_3	0	0
+	C_5	$C_{3.2}$	$C_{3.1}$	C_2	0	0
	D_6	D_5	D_4	D_3	D_2	D_1

This table shows the multiplication process of any two, three bit numbers. Each cell in the table containing data except for the operands “+” and “x” represent either a logical 1 or 0. A_3-A_1 represents the logical bits of the first expression with A1 being the least significant bit (LSB).
B_3-B_1 represents the logical bits of the second expression with B1 being the LSB.
D_6-D_1 represents the logical bits of the answer where D1 is the LSB.
Any bit containing a “C” represents a carry bit, the number after the C indicates which column the carry bit is coming from. For example C2 is the carry bit from adding A_2B_1 and A_1B_2 from column 2. Notice that two of the carry bits, $C_{3.1}$ and $C_{3.2}$ both come from column 3, but represent different carries which will be discussed later.
Any bits labeled with an A and a B, for example A_1B_1, represent the logical AND of bits A_1 and B_1. When multiplying in binary any number times 0 gives an answer of 0, so the only way the bit can be a 1 is if both bits are 1, and therefore you logically AND them together to get the correct answer for each bit.

Logic Expressions

Answer bits

$$D1 = A_1B_1$$
$$D2 = A_2B_1 \oplus A_1B_2$$
$$D3 = A_3B_1 \oplus A_2B_2 \oplus C_2$$
$$D4 = A_3B_2 \oplus A_2B_3 \oplus C_{3.1}$$
$$D5 = A_3B_3 \oplus C_4 \oplus C_{3.2}$$
$$D6 = C_5$$

The answer bits for this is quite easy as we simply XOR each bit in each column, which is the equivalent of adding them all together.

Carry Bits

$$C_2 = A_2B_1 \bullet A_1B_2$$

$$C_{3.1} = (A_3B_1 + A_2B_2 + A_1B_3 + C_2) \bullet \sim(A_3B_1 \bullet A_2B_2 \bullet A_1B_3 \bullet C_2) \bullet \sim(\sim A_3B_1 \bullet \sim A_2B_2 \bullet \sim A_1B_3 \bullet C_2) \bullet \sim(\sim A_3B_1 \bullet \sim A_2B_2 \bullet A_1B_3 \bullet \sim C_2) \bullet \sim(\sim A_3B_1 \bullet A_2B_2 \bullet \sim A_1B_3 \bullet \sim C_2) \bullet \sim(A_3B_1 \bullet \sim A_2B_2 \bullet \sim A_1B_3 \bullet \sim C_2)$$

$$C_{3.2} = (A_3B_1 \bullet A_2B_2 \bullet A_1B_3 \bullet C_2)$$

$$C_4 = \sim(\sim C_{3.2} + \sim(A_3B_2 + A_2B_3)) + \sim(C_{3.2} + \sim(A_3B_2 \bullet A_2B_3))$$

$$C_5 = \sim(\sim A_3B_3 + \sim(C_4 + C_{3.2})) + \sim(A_3B_3 + \sim(C_4 \bullet C_{3.2}))$$

The difference between $C_{3.1}$ and $C_{3.2}$ is that $C_{3.2}$ is a double carry whereas $C_{3.1}$ is a single carry. If all four bits in column three are a 1, then two carries need to be implemented. One way to do this would be to add both of those carries into column four, however I ran into the same problem of another possible double carry in column four, so in order to reduce the logic, and therefore reduce the number of transistors, needed then if all four bits in column four are a 1, then we add only one carry to column 5, and $C_{3.1}$ would be a 0. $C_{3.1}$ and $C_{3.2}$ can both be 0 but can never both be 1.

Truth tables

C_2

A_2B_1	A_1B_2	C_2
0	0	0
0	1	0
1	0	0
1	1	1

$C_{3.1}$

A_3B_1	A_2B_2	A_1B_3	C_2	$C_{3.1}$
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	0
1	0	0	1	1
1	0	1	0	1
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	0

$C_{3.2}$

A_3B_1	A_2B_2	A_1B_3	C_2	$C_{3.2}$
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

C_4

A_3B_2	A_2B_3	$C_{3.2}$	C_4
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

C_5

A_3B_3	$C_{3.2}$	C_4	C_5
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

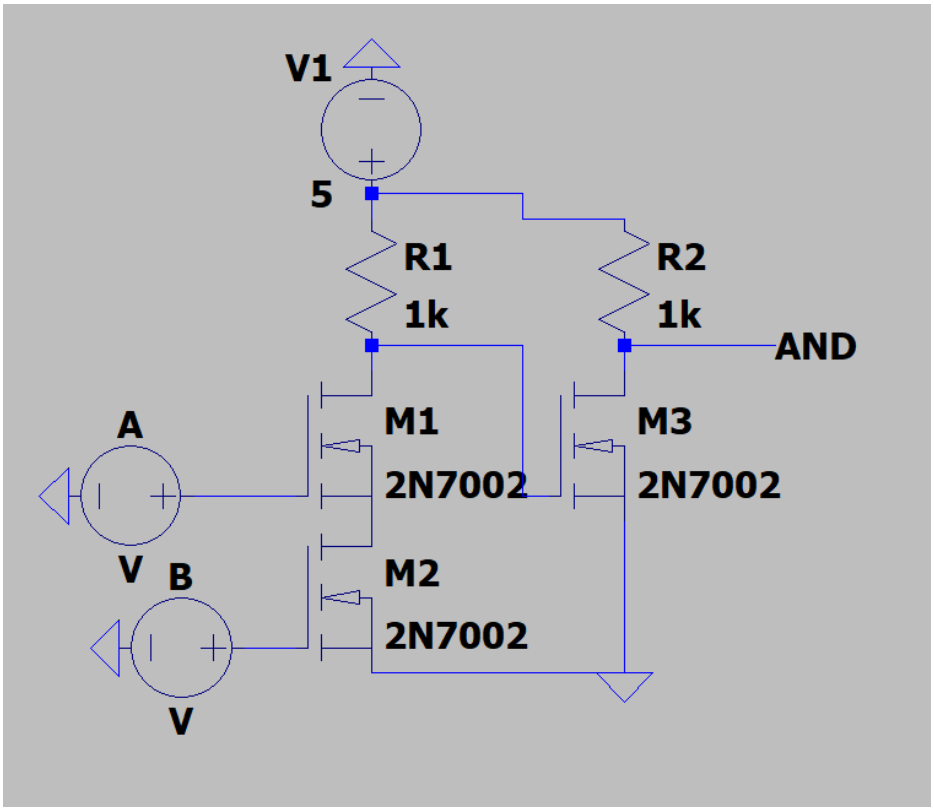
Schematic/Design

https://github.com/ianlane1/3x3MOSFETMultiplier This link will bring you to a github containing the schematic that I have created for this project.

Important notice: In the full scale design on the breadboards I am using the 2N7000 n channel MOSFET, however LTspice does not have that specific model of MOSFET so instead I am using the 2N7002 in the schematic design as a replacement, however both MOSFETs are the same.

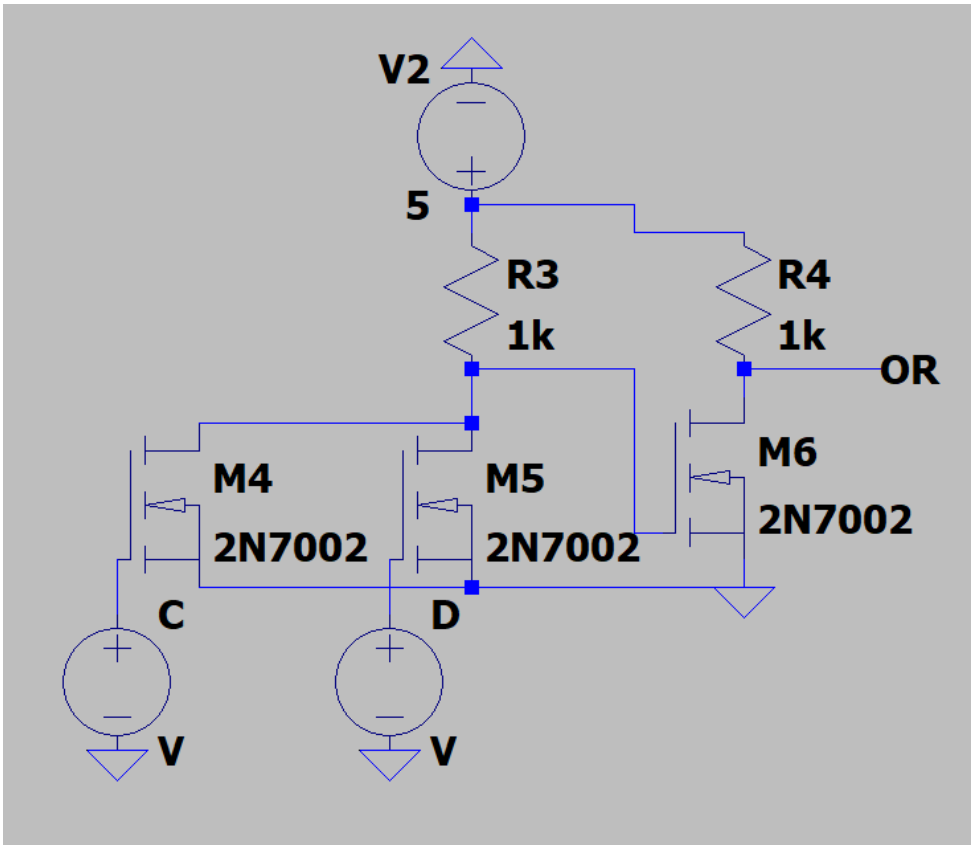
In the file called NMOS Logic.asc is a schematic going over the basic logic gates and how they are created using NMOS transistors. You can download the .asc file and switch the different voltages of each component and see how it affects the voltage of the net you are looking at.

Logical AND



In this picture you can see the schematic design of the two input AND. The first two transistors M1 and M2 are the NAND of voltage source A and B and then transistor M3 inverts the signal into an AND. In order to increase the number of inputs to this and just add however many transistors you need in series with M1 and M2 and ensure they have their own voltages at the gates.

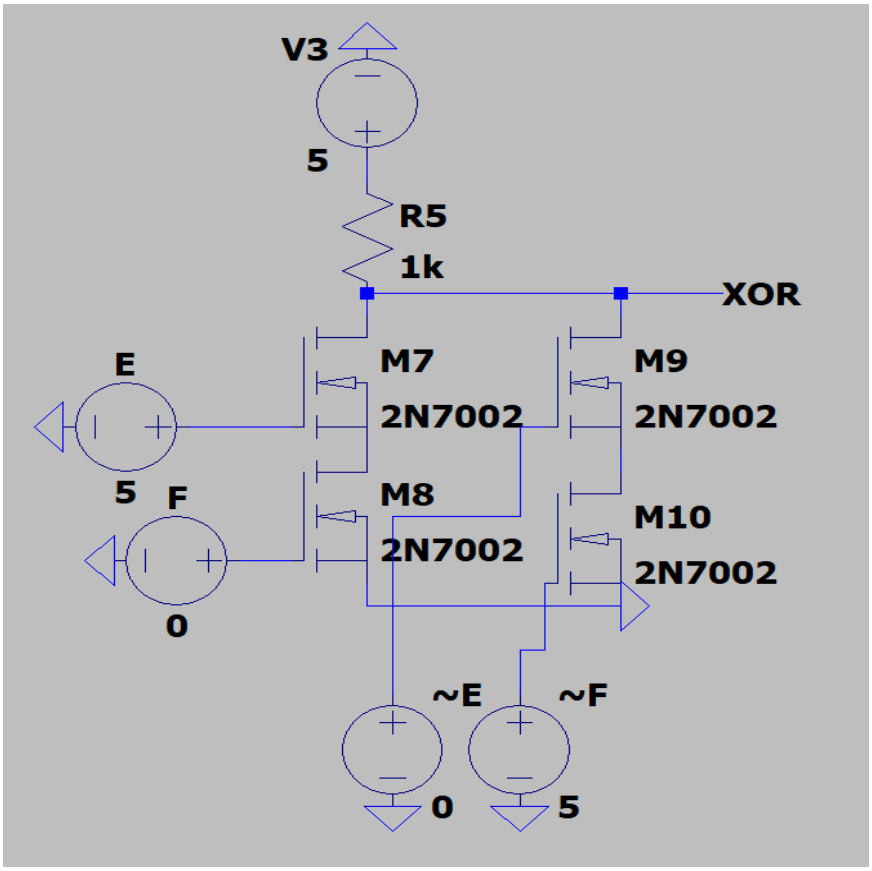
Logical OR



This picture shows the schematic of a two input logical OR gate. Transistors M4 and M5 give the output of the logical NOR whereas M6 inverts that signal to give the logical OR. In order to increase the number of inputs into this schematic you would put however many more transistors needed in parallel with M4 and M5 and give them their own gate signals.

Important notice: Whenever two signals are put into an AND the logical inverse of that signal is also given based on the creation of the AND, so when the signal A_1B_1, for example, is created I also have access to $\sim A_1B_1$ without needing any more transistors. This means that for any logic in the Logical Expressions section, it has no effect whether the bit A_nB_m or C_n is used or the logical inverse. However, DeMorgans theorem is used to reduce the logical expression created with those bits in order to reduce transistors, as logical NANDs and NORs will use one less transistor than their respective inverses.

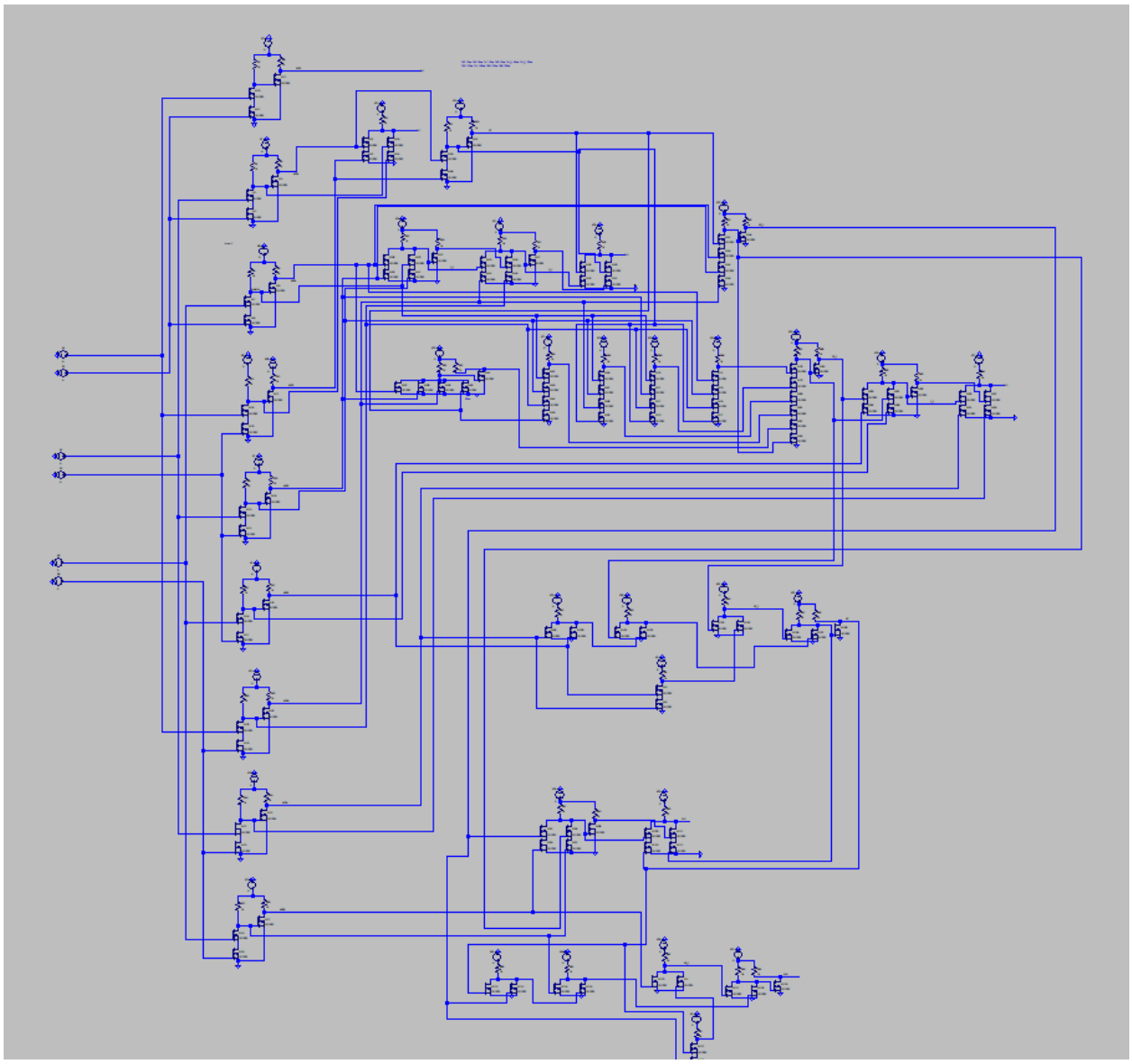
Logical XOR



This picture shows the schematic of the final logical expression used in this project, the XOR. XOR only produces a logical 1 when only one voltage source E or F is 5 volts. The inverse of voltage sources E and F are also needed to make the XOR work and these are automatically created when doing ANDs or ORs. When doing multiple XORs as is done for most of the answer bits, it is more simple and takes less transistors to simply chain multiple XORs together. In order to XOR three inputs A, B, and C, to create a three input XOR would take 16 transistors. However when chaining XORs I XOR inputs A and B, and then take that output and XOR it with C, this only takes nine transistors, four for each individual XOR and one for the inverse of the first XOR.

Note that when trying out the schematic above ~E and ~F must be the logical inverse of E and F respectively otherwise the XOR will not work. Also XNOR is not used in this project explicitly, however if it would need to be created all that would need to be changed would be switching either F and ~E, or E and ~F.

Full Scale Design



This picture shows the broad overview of the schematic. On the far left are the six voltages sources representing the 3 binary bits in A_3-A_1 and B_3-B_1. The top two voltage sources are A_1 and B_1 and then A_2 and B_2 and so on.

After the voltage sources are nine logical ANDs for each bit A_nB_m. After that I have set up blocks containing the answer and carry bits for each column. Each net labeled with a "C" is a carry bit and each net labeled with a D represents an answer bit. When conducting multiplication, any net with a 5V output represents a logical 1 and any net within the milliVolt range represents a logical 0. I would highly recommend for anyone interested to download the schematic in order to properly test and trace wire connections.

Timing Characteristics

The data sheet of the 2N7000 claims that the max time from when the transistor sees a valid input and gives out a valid output is 10ns. The table below was calculated by taking the max input time of all the inputs into a logic gate and then adding 10ns for each transistor until we get an output.

Node	Time (ns)
D_1	20
D_2	30
D_3	70
D_4	120
D_5	170
D_6	190
C_2	40
$C_{3.1}$	60
$C_{3.2}$	90
C_4	160

The reason that these increase as such is because each carry builds off of the previous carry and so it must wait until the previous carry is finished so that it can have valid data. Therefore the total time that must be waited until the full answer is received is 190ns.

Code Analysis

In order to fully test the full scale experiment I will be doing two things. The first will be manually inputting a few inputs and determining that the outputs are correct. This will be to ensure that most connections are secure and working before doing a full scale test.

In the full scale test I will be using an Arduino ATmega2560 that will send data in terms of 5 volts at the inputs A_1-A_3 and B_1-B_3 and then it will read data from nodes D_1-D_6. The Arduino will then compare the answer from the transistors with the answer it does based on its own calculations and a message will appear saying whether or not the test has passed or has failed.

The code can either be found through this link or through the github link in the Schematic/Design section
https://docs.google.com/document/d/1_CTK3dWRzgG-Gu-3iEnMvZWZGH5hBHrFmo_N65WhYi8/edit?usp=sharing

Full Scale Testing

There are going to be six different videos while working on the full scale project. The first video I go over how I use the schematic design on the different logic gates and implement them in real life on the breadboard. The next four videos will be working on the four stages of building that I will be doing in this project. The reason that I am building this in stages is to make it easier to debug. Building the whole thing at once without testing can lead to massive problems when trying to debug the system, whereas if I build the project in stages, I can catch any possible problems early, and if I encounter a problem, I know all of the previous stages are working and so I can greatly narrow down my search for the problem.

Stage 1 will be creating the 9 AND gates needed to logically AND each A bit with each B bit. This stage is important because it is fundamental to how each of the other bits act. If any wires are crossed or there is a bad connection, it could ruin the rest of the project.

Stage 2 consists of bits D_1 - D_3 and C_2

Stage 3 will only be creating the bits $C_{3.1}$ and $C_{3.2}$. This is because $C_{3.1}$ is the most logic intensive part of the whole project, there is a lot of room for error when creating this carry bit, and so rather than having multiple problems with multiple bits, I will only be doing the carry bits for the third row. $C_{3.2}$ is rather easy since it is only a four input AND gate and so because of its simplicity I will be creating it with $C_{3.1}$.

Finally stage 4 will contain bits D_4 - D_6 and C_4 - C_5. C_5 and D_6 are actually the same bit logically, so the output of C_5 is D_6.

The last video that I will be creating for this project will be the full scale testing with the Arduino. In the stage 4 video I will do some manual testing by using the DIP switch to manually input pre-determined numbers into the system, and check if the bits are correct. However, to manually check every possible combination would take time, so to thoroughly test the system the Arduino code I have above will run through every possible combination of multiplication and record any failure points within the system that I might not have caught through earlier testing.

Below I will list google drive links to all the videos once they are all done, in the meantime as I incrementally create videos they will be posted on github.