ARDUINO PROJECT

MAKING THE NONAXX CALCULATOR…

COMING UP..

INTRODUCTION

*The NONAXX calculator is a **4-bit calculator** that can perform **6 basic logical operations** – NOR, OR, NAND, AND, XOR, XNOR – from which the name NONAXX is derived, using the first letter of each operation. The calculator **prompts the user to enter two numbers sequentially. the output is shown by the LEDs, which help in simple visualization of the binary. Just like the binary system of 0s and 1s, these LEDs represent them with OFF and ON states, respectively.***

OBJECTIVES AND THOUGHT PROCESS

I wanted to create a program that ***takes a number between 0 and 15 and outputs its binary form using a user-selected logical operation****. Due to certain limitations, I was only able to build a 4-bit version.*

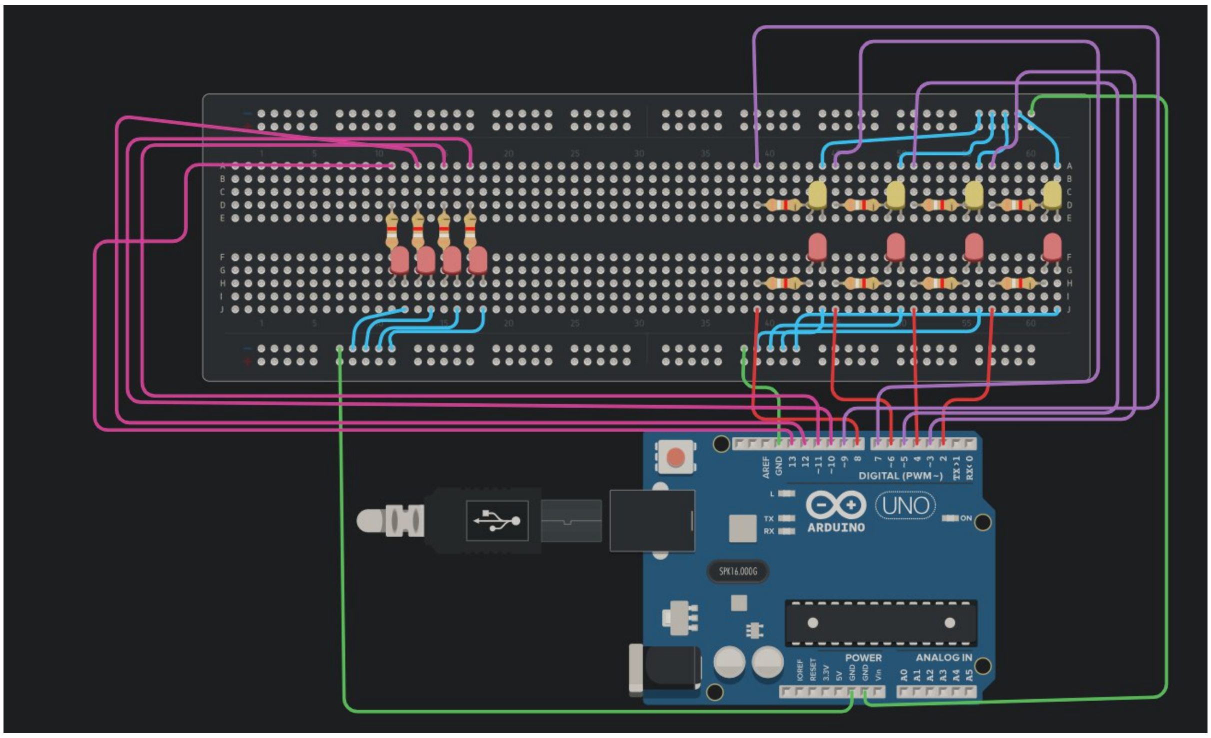
This project was more than just technical—it was about understanding how I think while performing calculations that feel "obvious." In programming, ***one of the most important skills is the ability to reflect on your thought process****. What's intuitive for us—like knowing that 1 + 1 = 2—is not inherently obvious to a computer. That's where logic gates come in.*

While my implementation might not follow a standard approach, it reflects the way I mentally process problems. If someone familiar with basic C++ reads my code, it should feel intuitive—like working through a problem on paper. That's exactly the mindset I wanted to capture using bitwise operations and functions.

COMPONENTS REQUIRED

RED LED	*8*
YELLOW LED	***4***
MALE TO MALE JUMPER WIRES	*27*
MALE TO FEMALE JUMPER WIRES	*4*
220 OHMS RESISTORS	*12*
BREADBOARD	*1*
ARDUINO UNO	*1*

CIRCUIT AND EXPLANATION



- *The red LEDs on the right display the first value entered by the user. The yellow LEDs display the second value.*
- *The red LEDs on the left display the result of the logical operation.*
- *Instead of one, three GND pins are used from Arduino because the **GND rail in this breadboard is not continuous. It is divided into 10 parts of 5 pins each.***

PRECAUTIONS:-

- ***Be sure that the wiring isn't loose, otherwise good luck with 2 hours of diagnosis and finding out that the wiring was loose. To ensure this, try wiggling a bit and give gradually increasing pressure, but not too much pressure. Start from zero and increase gradually.***
- *Don't connect the LED to the pin directly without a resistor, the LED will be damaged. Usual LEDs use 220 ohms resistor. The range goes usually from 200-1000 ohms.*
- *Follow the standard procedure of breadboards (wherever there are gaps, there is no connection. The signed rails are connected horizontally only. The other rails are connected vertically only.)*
- *The longer leg of LED must be connected to high voltage, and the shorter leg must be connected to GND. (**tip to remember: long = high, short = low/GND**)*

- *Be sure to **<u>follow the circuit from three views – top, along the long side of the breadboard, along the short side of the breadboard</u>**. This method helps a lot in detecting misconnections (like inserting the wire in the wrong pin) efficiently, cleanly and comfortably.*

<u>CODE AND EXPLANATION</u>

The code is available as a .ino file on github. One can check the code there and copy if they want to.

- *Do not get confused with the names of the parameters in the function definition with the variable assignment. They both behave differently. **<u>One name behaves as variable, the other behaves as a parameter (function definition)</u>**. If you change either of them, the code will remain intact.*
- *I named them in a way that's just for easy management of code. But in reality, It doesn't affect the code at all.*
- *Just like how we do OR in reality (assigning zero if and only if both the inputs are zero, otherwise its one), I coded the if else statement the same way.*
- *Similar methodology is used in defining remaining logic functions.*
- *Calling other functions in function definition is valid. Introducing 2 variables in function calling twice is also valid. **<u>How ever in some more complex programs, it is output is unpredictable if one uses the same variable in function calling twice</u>**. For example, in this following block,*

```
void bit_XNOR(int bin1[],int bin2[], int res[]){
  bit_XOR(bin1,bin2,res);
  bit_NOT(res,res);
}
```

- *I used **<u>bit_NOT(res,res);</u>**. Although it might seem confusing at first, but when you observe the bit_NOT definition in this code properly, it makes sense to use it.*
- *It is basically rewriting the array, which is possible and legal in this case. This **<u>saves the burden of creating a dummy array</u>** (an array that just stores temporary data and transfers to the original array again.), which is often more predictable, but I leveraged the understanding of my function definitions and skipped that step for efficiency and comfort.*
- *All in all, **<u>there is no harm in using functions like this</u>**, but be sure and confident about the function one is using, and understand that definition well.*
- *If you aren't confident, there is no harm in trying and learning anyways. That's just engineering. One fails, gets up, tries again using the lessons of failure.*

- *I've used these patterns of thought process in the entire code. They might look different blocks, but in reality, they are just the* ***<u>same idea (input arrays, bitwise iteration, logical condition, output array)</u>. <u>The only difference is in the presentation of the pattern and idea.</u>***
- *For example, I have used the same mental model in defining the bit_OR function and then for every other function. That was the hardest part, but when I successfully created it, it was way too easy for me to define other functions, be it bit_AND or bit_NOT, cause the mental model is settled in the mind. In coding, its important to set a mental model and work through it first. As you go on into coding, it evolves.*
- *For example,* ***<u>my mental models evolved when I realised that NAND is just NOT AND, which led me to an efficient solution of calling both the functions into bit_NAND definition.</u>***
- ***<u>This method also helps in debugging the code efficiently</u>*** *because I don't need to debug every block of code.* ***<u>If I know there is some problem, it will just reduce from 130 lines to just 20 lines i.e. 3 or 4 blocks</u>****, which in this case is function definitions.*
- *Hence it is important to create mental models first and then start programming. Have atleast one mental model at the start. As the code progresses, one might get more mental models to use.*
- *For example,* ***<u>the mental model I used in this code was trying to recreate something we do with pen and paper</u>****. How do we do that? We use some statements ourselves. When performing OR on paper,* ***<u>talk it out loud (or if you are shy, just murmur to yourselves) of what are you doing, it activates the metacognition (thinking about thinking), and you realise everything, that 'OK, if both the inputs in OR is 0, the output is 0, otherwise its 1.' Is your breakthrough moment.</u>***
- *The standard way to ask the user input in Arduino programming is a 3-step process. Step 1: ask (using println() to print a statement to ask), Step 2: wait (using a while loop which keeps on running until the user enters the value), Step 3: read (get the integer or float value using parseInt() or parseFloat() respectively, or get a string by readstring()).*
- *Other explainations are made as comments in the code itself, and is clear enough to create an understanding to the viewers.*

<u>SOME WAYS ON HOW I TESTED EFFICIENTLY…</u>

- *First, I only wrote until bit_OR. This way, I could test if my idea was right or not, or what should I tweak in my mental model.*
- *That's because of the* ***<u>leverage I gained due to a consistent mental model</u>*** *I used, which surely evolved, but was same at the core. I leveraged that core. If my bit_OR function goes right, then any other function definition I make in this code also would definitely work.*

- *This just reduced my testing requirement from 8 blocks to merely 2 or 3 blocks of code.*
- *It's an important skill to test efficiently and debug efficiently. Anyone can spend 2 hours in testing and debugging, but the one who can do that in 5 minutes is on the path of mastery.*
- *Similarly, to test hardware, make the complete circuit with proper color coding, similar to what I did in my circuit diagram. It helps a lot in debugging.*
- *This helps narrowing down the options effectively, and it's a common practice in circuit making.*
- *I used serial monitor to check if there were any problems. For example, I used Serial.println(digitalRead(pin[2])); to check if the problem in that specific pin is from Arduino or loose wiring or the LED.*
- *For maximum efficiency, consider using a voltmeter. If you don't have one, make one for the range of 0-5V using Arduino. Check out that project on my github profile to learn more about it.*

CHALLENGES AND LEARNINGS

- *It's the first time I made a code longer than usual, 100+ lines. So I naturally faced several challenges like simple syntax errors (yes they are a common challenge).*
- *I would go around checking the syntax through the code several times, but it is sometimes* ***as subtle as missing a bracket, and in a language like C++ where long codes overflow with brackets to a point all you see is a net of brackets and nothing else, it's a very subtle yet very common and difficult problem****.*
- *After a few failures, I realised that* ***the Arduino IDE highlights the pairs of brackets upon selecting one of the pair****. It became much easier since then.*
- *I still have the problematic habit of writing in python style, that itself creates several errors. I am still doing practicing while coding to stay conscious about it and the errors have reduced.*
- *It will be a long path to rectify it and somehow manage both the languages. But that's also a skill I would like to learn, and I would recommend others to learn as well, since Python gives speed of development. C++ gives speed of execution. Together, they make you a full-stack problem solver. This is from personal experience over using both the languages.*
- *I gained confidence in handling more complex stuff. It is definitely a good leap from a simple voltmeter to here.*
- *Interestingly, there were no surprising or "mystery" errors — most bugs were predictable once I slowed down and reviewed carefully. This showed me that clear mental models and consistent structure go a long way in preventing weird behaviour.*

CONCLUSION

It was a fun journey to make this. It almost took me a week to complete it and bring the NONAXX calculator to life. I learnt a lot of things and would like to continue learning as I continue making my projects. Through the sequence of testing, building, iterating, and then at last documenting and publishing, I am learning a lot.

For any challenges or corrections or guidance, mail me on

arduinomechtech@gmail.com