Prithvi Kannan

405110096

bitParity

To do this function, I started by working through test cases in my head to understand what was going on at the low level. I wrote out the examples provided in their bit form and tried to understand what this function was doing. I realized that the XOR operator preserved the existing parity, so I used this property to half the input until there was only 1 bit remaining, which I then could easily compare with something that I know had an odd number of 0’s, which for simplicity, I chose 1. To test this I ran the dlc tester.

rotateRight

To do this function, I knew that I had to understand what was happening at a bit level, so I ran through the provided test examples multiple times, and tried to develop my own to make sure I understood what it meant to rotate right in this context. One of the challenges I faced on this one was the incorrect example, because it meant that my original understanding of rotateRight was wrong, which meant that I spent a lot of time struggling with this problem. Once the correct example was sent out, it made much more sense, and I was able to figure out how this function was working. In order to implement this, I knew that I had calculate the amount to move, which required me to somehow represent 32-n. Since “-“ was not allowed, I had to use 2s complement arithmetic to rewrite 32-n with only “+” and “~”. I tested this function using the dlc tester.

byteSwap

To start out, I tried to understand what was going on at the bit level. I began by analyzing the provided example, and how the 1st byte was being swapped with the 3rd byte. After understanding this example, I checked my understanding by trying to make some more of my own. In order to do this function, I realized that I needed to produce some masks. I would use these masks to cover up the parts that I did/did not want to be changed. I used the value of 0xFF and shifted by m << 3 or n << 3 depending on if this was my m or n mask. I used the << 3 to perform the equivalent of multiplication by 8, which is how I could convert from bits to bytes. After this, I applied the masks by using the and operator with the input. I combined the two masks into one by using the or and the ~ operator. Finally, I used the mask to actually modify x. I tested my code using the dlc tester.

fitsShort

This was one of the first few functions I did, since it had a level 1 rating. This meant that my understanding of how to handle integers was very little at this time, since I had no experience yet. I realized that if something were to not be able to fit a short, that would mean that its value was too big to fit 16 bits. Originally, I thought that this meant that there were 0’s in the first 16 bits of the function and tried to implement it that way. I struggled with this solution a lot in the beginning, but I later realized that negative numbers, which have 1’s in the MSB can still fit a short. I proved this to myself with the example of -1, which is all 1’s and would still fit a short. My final solution took advantage of sign extension and involved left shifting by 16 and then right shifting by 16 to see if anything changed. I tested this code using the dlc tester, and through code tracing.

bitAnd

I realized based on the provide constraints that I had to cleverly form an and using only ~ and or. From my past experience in high school class called digital electronics, I recalled some of DeMorgan’s laws, and figured that those would be very useful for this problem since I was attempting to basically use or to perform the and operation. I looked up DeMorgan’s laws, and read that !(x&y) can be rewritten as !x | !y. However, DeMorgan’s laws were calculating !(x&y), not x&y, so I had to apply another ! to do !!(x&y), which was x&y. Since all of these operations were being done at the bit level, I knew that I had to use ~ in the place of !. To test this code, I ran the dlc tester.

subOK

The description for this function mentioned overflow, so the first thing I did was remember what it meant for an integer to overflow. Overflow meant that the number has exceeded Tmin or Tmax and has therefore wrapped around. At the bit level representation, this meant that the MSB has changed from 1 to 0 and vice versa. I realized that the situations that overflow could happen were when x and y have different signs or when x-y would have a different sign than x. Otherwise, x-y would be okay, so I focused on the cases where overflow could occur. I calculated x-y and stored that value in a variable a. To calculate x-y, I had to use 2s complement arithmetic to create the “-“ since only “+” was allowed. I rewrote x-y as x + ~y + 1. Then I calculated x XOR y to see if x and y would have different signs. Finally, I tested using the dlc tester.

isGreater

For the isGreater function, I started by looking at the examples of comparing 4 and 5. I wrote out 4 and 5’s representations in bits, and began to understand what I had to do at the bit level in order to implement this function. I realized that I had to isolate the MSBs of both x and y to compare those. However, before doing this step, I actually had to calculate the difference between x and y, which would allow me to have a something to represent whether x was greater than y. I right shifted this number all the way to isolate the MSB. I also realized that anytime x had a 0 as MSB (positive) and y had a 1 as MSB (negative), x was always going to greater than y. I also realized that if this situation did not occur, I had to check if the MSB of x wasn’t negative when the MSB of y was positive and return the difference. I checked my results using the dlc tester.

fitsBits

To handle fitsBits, I started by recognizing that this was just a generalization of a function I had previously written, fitsShort. Naively, I thought that I could just re-use the code from fitsShort, and replace 16 with the input n. However, I tested this on the dlc tester and realized the flaw in my logic. After writing out examples by hand, I realized that I could not just replace 16 with n. Instead, I had to compute 32-n, and shift by that amount instead of shifting by n. Creating the expression 32-n proved to be a challenge and required 2s complement arithmetic to represent 32-n as 33 + ~n instead. Lastly, I tested this with the dlc tester again, and this time got the correct results.

Negate

Negate was one of the more simple function in my opinion. I ended up doing this one early on, which really helped me understand 2s complement representations, and used this idea many times later on for other puzzles. Originally, I was thinking of just changing the MSB to be negative or positive, but I realized that that was a completely wrong way of thinking about the problem. Instead, I decided to analyze what the ~ operator did, and finally realized that x + ~x = -1. Then I used this statement to rearrange -x as ~x +1, which gave me my desired output. I tested this through code tracing and the dlc tester.

isTMax

To handle isTMax, I immediately though of checking for overflow. This intuition came from the fact that I knew that integers had maximum values, and that when we overflow an integer, it will wrap around. Now I had to determine how to check for overflow at the bitwise level. After a lot of experimentation, I reached my solution. I started by adding 1 to the input, and then I began to check for overflow. I added the new number to x, and then applied the not operator on the newx variable. I flipped the bits of x using the ~ operator, and compared the values of x and newx with an or operator. I ran this on the dlc tester, and got incorrect results. I realized after tracing the code that I forgot to apply the not on the final output I got since I wanted to return 1 if x was the maximum, whereas my original code was returning 0 if x was the maximum. This small error took a while to figure out, but I ended up realizing this by tracing the code with tmax as the input. I added the final not, and got success with the dlc tester.

Overall Difficulties:

One of the major difficulties in this assignment was handling bit-level representations of integers. This is something that I had not been previously exposed to in classes like CS31 or CS32, where I was used to just using integers without considering how they were interpreted by the computer. Many of the issues that I had to deal with were related to how left and right shifts, and how to understand how I could utilize left and right shifts to compare different parts of the integers. For example, many of the puzzles required a comparison of the most significant bits of a number. To solve this, I realized that this was simply a right shift by 31. Lastly, I re-watched the lecture on integers and read some online resources about bitwise representations of integers to make sure my understanding was correct at the fundamental level.