CSCI-6050

Project 2 Description

Richard Hoehn

October 1, 2024

Mystery Function 1

What it Does

This function performs a type of exponentiation, but it subtracts one (1) from the exponent first.

$$return = a^{b-1}$$

Where $b > 1$. If $b \leq 1$, the result is 1.

How I Solved It

I noticed that we first set the `%eax` register to one, meaning we have an output of at least `1`. I then saw that it jumped straight to `.L2`, where it first subtracts from `%rsi` and saves the result to `%edx`. It then checks if we are at zero. Afterward, it multiplies a by itself and stores it into `%rax` (our return value). Once the loop ends, it exits and returns `%rax`.

Mystery Function 2

What it Does

This function takes an `unsigned int` and looks at each bit. If it finds a one, it flips its location to the opposite side of the bit string. Essentially, the function **reverses** the bits of the input `num`. Each bit in `num` is mirrored to the opposite side of the 32-bit integer based on its position from the reverse side.

This is done by iterating over all `32` bits of the value. If a `1` is detected, it shifts a single `1` to that position.

How I Solved It

I noticed that we created `0x800000`, which is the same as `-2147483648` in decimal. This means we have a single bit mask that is then used to shift with `shrl %cl, %eax` and then `OR` it with the current temp value in `%edx` that is at the flipped location.

Once we have iterated over all 32 bits, we move the `%edx` to `%eax` (%rax) as the output and return it. This helped me understand that I need to shift and OR the result before returning it.

Mystery Function 3

What it Does

This function takes a `pointer` to a `long[]` array of numbers and the count of array items as `n` .

It starts by setting the **MAX** to the first number in the array. This initializes the maximum value. It then increments the count and checks if the number at `a[1]` is greater than `a[0]` . If this is the case, it sets the **MAX** to `a[1]` and increments the counter.

We continue iterating with the condition `MAX = a[n] > MAX` . Once we've completed the loop, we return the `MAX` as a `long` result.

How I Solved It

The first part was straightforward; I noticed that I was moving an `address/pointer` to `%rcx` and then incremented my counter to a fixed `1` by setting `%eax` as my counter holder.

Next, I understood that we compare the `num` entry of the C function to my counter, which was set to "1". If they were the same, we exit the function and return the value stored in `%eax` , moving it to `%rax` for return.

I then compared the memory location by multiplying the counter by "8" and adding the initial memory location of `a[0]` to a temporary holder for comparison to our **MAX** value.

It seems that the assembly code loops until we reach the end of `n` . Whatever is set as the max is then returned.

Mystery Function 4

What it Does

This function counts the number of `1` s in an `unsigned long` (64-bit) value. It simply takes in a number, right shifts it (which I used division by ”2” for), and uses a ”1” as a mask to `AND` with `0x00000001` . If there is a ”1”, we add it to our accumulator/sum counter.

Once the input number `n` is zero, we return the accumulator/sum from the function.

We covered this example in class a few weeks ago, so it was not overly difficult to solve.

How I Solved It

I noticed that we first set `%eax` to zero. We then check to see if our input `%rdi` is zero. If not, we mask it with a `1` and add it to our accumulator, similar to what we did in class. I used something like `sum += n & 1;` in C. If there is a ”1” in the last position, it will be added to the accumulator.

We then shift our value to the right by division of ”2”, like this: `n = n / 2;` to shift ”n” by one bit.

Afterward, we check again if `n == 0` . If true, we return the accumulator and exit the function.

Mystery Function 5

What it Does

This function takes two arguments of `unsigned int` size (32 bits) and `XOR` s them with each other. It then saves the result in the `%edi` variable and counts the bits in the temporary variable.

This is very similar to the last mystery function, which counted the bits from an `unsigned long` and returned the count. In this case, we simply "XOR" two numbers to get the bits that are different.

It takes in a number, right shifts it (I used division by "2"), and uses a "1" as a mask to `AND` with `1` . If there is a "1", we add it to our accumulator/sum counter.

Once the input number `n` is zero, we return the accumulator/sum from the function.

How I Solved It

I noticed that the assembly started by XORing the inputs with each other `xorl %esi, %edi` and saving the result back into `%edi` . It then takes the mask of `AND 1` , adds it to the accumulator initialized to "0" at the start of the application, and shifts to the right by "1" each iteration until `%edi` is zero. Finally, it returns the accumulated sum in `%rax` .