CISC 131
Introduction to Programming and Problem Solving
Spring 2020
Bit Manipulation

Due: **Friday, May 1, 2020, at start of class**
Points: **none**

Start of Today's Lecture

Shifting is the movement of digits inside a value. The digits can either be shifted to the right or shifted to the left. Shifting to the left multiplies the value. Shifting to the right divides the value. Here is an example using integer decimal values. Each row uses the value from the preceding row.

	decimal
original value	265
shift to right one digit	26
shift to left one digit	260
shifted to the right two digits	2
shifted to the left two digits	200
shifted to the right 3 digits	0
shifted to the left 1 digit	0
original value	130
shift to right one digit	13
shift to left one diit	130
shifted to the right two digits	1

When you shift a decimal value, shifting to the left multiplies it by ten and shifting it to the right divides it by ten. Ten is the amount because each digit in a decimal number represents a power of ten. Notice that the result of shifting an integer to the right will always result in an integer value. In the table above, you can see that when 265 was shifted to the right (divided by 10) the result was 26 and not 26.5.

The central processing unit (CPU) of the computer only does simple operations such as adding, comparing, and shifting. When talking about the CPU, shifting is the movement of *bits* inside a value. Recall that a *bit* is an acronym for **bi**nary digi**t** and that each bit can have a value of zero or a value of one – the only two values possible in base two.

If a variable is assigned an integer value and then shifted one bit to the left, the resulting value will be twice as large as the original value. If it is shifted one bit to the right, the resulting value will be half the original value because the value is stored in base two. The value is halved or doubled because each bit represents a power of two.

Inside the machine, eight bits are aggregated to form a *byte*. Languages typically use either four bytes (thirty-two bits) or eight bytes (sixty-four bits) to store a value. When doing shifting, the JavaScript language uses thirty-two bits and the result is always an integer value.

If a variable is assigned an integer value and then shifted one bit to the left, the resulting value will be twice as large as the original value since the value is stored in base two. If it is shifted one bit to the right, the resulting value will be half the original value – again, because the value is stored in base two. If the values are integers, the result of shifting will always be an integer. For example, 63 divided by two is 31.5 but if the *binary* version of 63 is shifted one bit to the right, the result will be 31. The fractional part is lost. Here are a couple of examples.

	binary	**decimal**
original value	00011010	26
shift to right one bit	00001101	13
shift to left one bit	00011010	26
original value	00011111	31
shift to right one bit	00001111	15
shift to left one bit	00011110	30

The value is halved or doubled because each bit in a binary number represents a power of two. This is analogous to a base ten number where each digit represents a power of ten. Shifting is done using thirty-two bits in Javascript and the result of shifting is always an integer.

The Javascript operator for left shifting is **<<** and for right shifting is **>>**. For example:

```
tmp = tmp << 1;
```

will shift the value in *tmp* one bit to the left and store the result back into *tmp*. Recall that the only way to change the value of a variable, is to have the variable name appear to the left of the assignment operator.

Shifting is done using thirty-two bits in Javascript and the result of shifting is always an integer. The Javascript operator for left shifting is **<<** and for right shifting is **>>**. For example:

```
tmp = tmp << 1;
```

will shift the value in *tmp* one bit to the left and store the result back into *tmp*. Recall that the only way to change the value of a variable, is to have the variable name appear to the left of the assignment operator.

Step 1
Write the HTML and CSS that will display the following in the browser:

```
<< 0 >>
```

You can use three *span* elements for this but make sure to put them on the same line in the HTML source file. Assign an *id* of *leftShift* to the span containing **<<**, and *id* of *value* to the *span* containing zero, and an *id* of *rightShift* to the *span* containing **>>**.

Step 2
Write the statements in the *window.onload* function that will store the element references to each of the three span *elements*. After these, write the statements that will ask the user to enter an integer and then store the entered number in the *span* element that has the *id* of *value*. Test to see if everything is working. The number entered by the user should appear between the **<<** and **>>**.

Step 3
Write a function with this function header:

```
function leftShift ( element )
```

The function is passed a reference to an HTML element. The function does not return anything. The function gets the *innerHTML* of *element*, explicitly changes it to a number[1], left shifts the number one bit, and then stores the result back into the *innerHTML* of the element.

In the *window.onload* function, associate this function with the *onclick* behavior of the *leftShift* element. This will have to be done using a *function closure* since you need to pass the element reference of the *value* element to the *leftShift* function. Function closures were introduced in Lecture 8, April 1, 2020.

Test it out. Clicking on the **<<** should multiply the *value* by two each time. Enter one as the value and click the **<<** thirty-one times. The large positive value that was produced by thirty shifts now becomes a large negative value. Since Javascript represents integers using two's complement, the high order bit of the integer is zero if the integer is non-negative and one if it is negative. If a variable is initialized to one and you continually shifted it to the left one bit, the one value would continue to more to the left until it reached the high order bit and then the value would become negative.

Step 4

Now write a similar function and establish the appropriate *onclick* behavior that will cause the *value* to be right shifted each time the **>>** is clicked. This will cause the *value* to be divided by two each time the function is called. Test everything out with positive, negative, and non-integer numbers. Why is it that when you right shift negative integers the *value* never goes to zero but when you right shift positive integers, it does? Just something to think about.

Step 5 – Something Different

Create a new HTML and Javascript program. Write these two functions:

```
function isOdd ( number )
function isEven ( number )
```

Both will be passed an integer. The *isOdd* function returns Boolean *true* only if the parameter is an odd value. The *isEven* function returns Boolean *true* only if the parameter is an even value. In all other cases, the functions return Boolean *false*. When writing functions like these, it is good programming practice to implement one of them by having it call the other.

Now write a program that does the following:

1. Gets two integers greater than zero from the user. One is stored in *a* and the other in *b*. Remember that input from the user is always a *String* so you need to explicitly convert that *String* into a number after getting it from the user.
2. The variables *a* and *b* are passed to a function named *mystery* that does the following:
   i. If *a* is even, it initializes the variable *result* to zero otherwise it initializes the variable *result* to the value of *b*.
   ii. Enters a loop that will continue as long as *a* is greater than one. Each time through the loop, the following is done:
      a. The value of *a* is changed by shifting it one bit to the right.
      b. The value of *b* is changed by shifting it one bit to the left.
      c. If either *a* or *b* is an odd value, *result* is changed by having *b* added to it.
   iii. Returns the value of *result*.

[1] use the Number function to do this

3. Uses a *window.alert* to display the values entered by the user and the value returned from the call to the *mystery* function.

Run the program a few times using small integer values greater than zero. What does this mystery function do? Why does it do this? What determines how many times the loop will execute? Is there a way you can minimize the number of times the loop will execute?