What is the difference between arithmetic shifts and logical shifts??

Generally based on the data type, the number of bits will be allocated for that variable in memory. So, if you declare a variable as char it will take 8 bits, int 32bits, short 16 bits etc.... If the data type is signed means one bit will be allocated for sign bit.

Examples:
short a = 8;
In memory it will be stored as: 0000000000001000

short a = -8;
In memory: 1111111111111000
How it will store negative numbers is it will take ones complement of 8 and it will add 1 to it, i.e, two's complement of number

Coming to shifts based on the shifts , it will shift the bits left or right.

Logical Right Shift (>>):
It will shift the bits right by introducing zero in the shifted place.
Example: 8>>1 =4
In memory: 0---->0000000000000100
So your answer for logical shift is correct it will divide the number by 2^(shiftlen)

Logical Left Shift or Arithmetic Left Shift(<<)

Both will shifts the bits left by introducing zero in the shifted place. Only difference is arithmetic left shift causes overflow:

Example: 8<<1 = 16
In memory: 0000000000010000 <----0
So your answer for logical right is correct it will multiply the number by 2^(shiftlen)

Arithmetic Right Shift(>>):

Here comes the true problem, what happens if we logically right shift a number which is negative, so it will place zero in the shifted bit position, but negative numbers must contain 1 in their Most Significant Bit(MSB). So here comes to the rescue the Arithmetic Shift Right. What it will do is it will shift the bits right but it will replaces with whatever bit present in MSB

For example :
-8>>1;
((1111111111111000) >>1) . So here MSB(last bit) is 1 so it will shift right and replaces with 1

1---->1111111111111100

8>>1;
((0000000000001000) >>1) . So here MSB(last bit) is 0 so it will shift right and replaces with 1

0---->0000000000001000