Bitwise Operators Assignments (Basics Level):

1. **Write a program to Set, clear and toggle a particular bit using bit wise operator?**

Sure, here's a simple C program that demonstrates how to set, clear, and toggle a particular bit using bitwise operators. The program includes three functions for each operation:

1. `setBit` to set a particular bit.
2. `clearBit` to clear a particular bit.
3. `toggleBit` to toggle a particular bit.

```c
#include <stdio.h>

// Function to set a particular bit
unsigned int setBit(unsigned int num, int bit) {
    return num | (1 << bit);
}

// Function to clear a particular bit
unsigned int clearBit(unsigned int num, int bit) {
    return num & ~(1 << bit);
}

// Function to toggle a particular bit
unsigned int toggleBit(unsigned int num, int bit) {
    return num ^ (1 << bit);
}

int main() {
    unsigned int num = 0x0F; // Example number: 00001111 in binary
    int bit = 3; // Bit to operate on (0-based index)

    printf("Initial number: %u\n", num);

    // Set the bit
    num = setBit(num, bit);
    printf("After setting bit %d: %u\n", bit, num);

    // Clear the bit
    num = clearBit(num, bit);
    printf("After clearing bit %d: %u\n", bit, num);

    // Toggle the bit
    num = toggleBit(num, bit);
    printf("After toggling bit %d: %u\n", bit, num);
```

```
    return 0;
}
```

Explanation:

- **Setting a Bit:**
  - `num | (1 << bit)`:
    - `1 << bit` shifts the number 1 left by `bit` positions, creating a mask where only the `bit`th bit is set.
    - The bitwise OR operator `|` sets the corresponding bit in `num`.

- **Clearing a Bit:**
  - `num & ~(1 << bit)`:
    - `1 << bit` shifts the number 1 left by `bit` positions.
    - `~(1 << bit)` inverts all bits of the mask, setting the `bit`th bit to 0 and all others to 1.
    - The bitwise AND operator `&` clears the corresponding bit in `num`.

- **Toggling a Bit:**
  - `num ^ (1 << bit)`:
    - `1 << bit` shifts the number 1 left by `bit` positions, creating a mask where only the `bit`th bit is set.
    - The bitwise XOR operator `^` flips the corresponding bit in `num`.

Example Output:

```
Initial number: 15
After setting bit 3: 15
After clearing bit 3: 7
After toggling bit 3: 15
```

In this example, starting with `num` as `15` (binary `00001111`), the operations will affect the fourth bit (index 3, counting from 0). Setting the third bit has no effect because it is already 1, clearing it changes the number to `7` (binary `00000111`), and toggling it switches it back to `15`.

(**OR)**

To manipulate individual bits of an integer, you can use bitwise operators in many programming languages like C, C++, Java, or Python. Below, I'll provide examples in C and Python to set, clear, and toggle a specific bit in an integer.

C Program

```c
#include <stdio.h>

// Function to set a particular bit
int setBit(int num, int bit) {
   return num | (1 << bit);
}

// Function to clear a particular bit
int clearBit(int num, int bit) {
   return num & ~(1 << bit);
}

// Function to toggle a particular bit
int toggleBit(int num, int bit) {
   return num ^ (1 << bit);
}

int main() {
   int num = 29; // Example number: 29 (binary: 11101)
   int bit = 1;  // Bit position to manipulate (0-based)

   printf("Original number: %d\n", num);
   printf("Binary representation: %x\n", num);

   num = setBit(num, bit);
   printf("After setting bit %d: %d\n", bit, num);
   printf("Binary representation: %x\n", num);

   num = clearBit(num, bit);
   printf("After clearing bit %d: %d\n", bit, num);
   printf("Binary representation: %x\n", num);

   num = toggleBit(num, bit);
   printf("After toggling bit %d: %d\n", bit, num);
   printf("Binary representation: %x\n", num);

   return 0;
}
```

Python Program

```python
```

```
def set_bit(num, bit):
    return num | (1 << bit)

def clear_bit(num, bit):
    return num & ~(1 << bit)

def toggle_bit(num, bit):
    return num ^ (1 << bit)

if __name__ == "__main__":
    num = 29  # Example number: 29 (binary: 11101)
    bit = 1   # Bit position to manipulate (0-based)

    print(f"Original number: {num}")
    print(f"Binary representation: {bin(num)}")

    num = set_bit(num, bit)
    print(f"After setting bit {bit}: {num}")
    print(f"Binary representation: {bin(num)}")

    num = clear_bit(num, bit)
    print(f"After clearing bit {bit}: {num}")
    print(f"Binary representation: {bin(num)}")

    num = toggle_bit(num, bit)
    print(f"After toggling bit {bit}: {num}")
    print(f"Binary representation: {bin(num)}")
```

Explanation

- **Set a bit:**
  - Use the bitwise OR operator (`|`) to set a specific bit. `(1 << bit)` creates a mask where only the desired bit position is `1`. OR-ing this mask with the number sets the desired bit to `1`.

- **Clear a bit:**
  - Use the bitwise AND operator (`&`) with the negated bit mask to clear a specific bit. `~(1 << bit)` creates a mask where only the dSesired bit position is `0` and all other bits are `1`. AND-ing this mask with the number clears the desired bit.

- **Toggle a bit:**
  - Use the bitwise XOR operator (`^`) to toggle a specific bit. `(1 << bit)` creates a mask where only the desired bit position is `1`. XOR-ing

---

2. WAP whether a number is ODD or EVEN using bitwise ?

Sure, here's a complete C program to determine whether a number is odd or even using bitwise operations:

```c
#include <stdio.h>

// Function to check if a number is odd or even using bitwise AND
void checkOddOrEven(int num) {
    if (num & 1) {
        printf("%d is odd.\n", num);
    } else {
        printf("%d is even.\n", num);
    }
}

int main() {
    int num;

    // Input a number from the user
    printf("Enter an integer: ");
    scanf("%d", &num);

    // Check if the number is odd or even
    checkOddOrEven(num);

    return 0;
}
```

Explanation:
- **Bitwise AND Operation (`&`)**:
  - `num & 1` isolates the least significant bit (LSB) of `num`.
  - The binary representation of `1` is `00000001`.
  - If the LSB of `num` is `1`, the result of `num & 1` is `1` (indicating an odd number).
  - If the LSB of `num` is `0`, the result of `num & 1` is `0` (indicating an even number).

Example Output:
When you run the program, you will see an output like this:

```
Enter an integer: 4
4 is even.

Enter an integer: 7
7 is odd.
```

This program reads an integer from the user, checks whether it is odd or even using a bitwise operation, and then prints the result.

---

3. Write a printbinary(int , int) function consists of 2 integer variables. First one is the value of the variable and the second one is the size of the variable.

Example:
char x=5;
printbinary(x,sizeof(x));
output: 00000101

Certainly! Below is the implementation of the `printbinary` function in C that takes two integers: the value of the variable and its size in bytes. It then prints the binary representation of the variable.

```c
#include <stdio.h>

// Function to print the binary representation of a variable
void printbinary(int value, int size) {
    // Calculate the number of bits in the given size
    int numBits = size * 8;

    // Loop through each bit from most significant to least significant
    for (int i = numBits - 1; i >= 0; i--) {
        // Print '1' if the i-th bit is set, '0' otherwise
        printf("%d", (value >> i) & 1);
    }
    printf("\n"); // New line after printing the binary representation
}

int main() {
    char x = 5;
    printbinary(x, sizeof(x)); // Expected output: 00000101

    int y = 13;
    printbinary(y, sizeof(y)); // Expected output: 00000000000000000000000000001101
(for a 32-bit integer)

    return 0;
}
```

Explanation:

1. **Calculating Number of Bits**:
   - The size of the variable in bytes is multiplied by 8 to get the number of bits.
   - `int numBits = size * 8;`

2. **Loop Through Bits**:
   - The loop iterates from the most significant bit (leftmost) to the least significant bit (rightmost).
   - `for (int i = numBits - 1; i >= 0; i--) {`

3. **Print Each Bit**:
   - The expression `(value >> i) & 1` shifts the value `i` bits to the right and then applies a bitwise AND with `1` to isolate the i-th bit.
   - `printf("%d", (value >> i) & 1);`

4. **New Line After Printing**:
   - After printing all bits, a new line is printed for formatting.
   - `printf("\n");`

Example Output:
For the `char x = 5` example:
```
00000101
```
For an `int y = 13` (assuming a 32-bit integer):
```
00000000000000000000000000001101
```

This `printbinary` function will work correctly for different sizes of variables such as `char`, `short`, `int`, and `long`, depending on the size parameter provided.

---

4. WAP to count the bits set (bit value 1) in an integer? Find out and compare different possibilities?

I'd be glad to help you with that! Here's a C program that implements different methods to count the number of set bits (1s) in an integer, along with explanations and a comparison of their efficiency:

Methods:

1. **Looping and Bitwise AND:**
   - This is a straightforward approach that iteratively checks each bit and increments a counter if the bit is set (1).
   - Time complexity: O(log n), where n is the number of bits in the integer (typically 32 for most architectures).

C
```
int count_set_bits_loop(unsigned int n) {
```

```
    int count = 0;
    while (n) {
        count += n & 1;
        n >>= 1;
    }
    return count;
}
```

2. **Built-in `bin()` function (less efficient):**
    - This method converts the integer to its binary string representation using `bin()`.
    - Then, it counts the occurrences of '1' characters (excluding the '0b' prefix) but is generally less efficient than the other methods due to conversion overhead.
    - Time complexity: O(number of set bits), which can be worse than O(log n) in the worst case.

C
```
int count_set_bits_builtin(unsigned int n) {
    return bin(n).count('1') - 2;  // Subtract 2 to remove '0b'
prefix
}
```

3. **Brian Kernighan's Algorithm:**
    - This clever approach exploits the fact that subtracting 1 from a number with a set bit flips the rightmost set bit to 0 and carries a 1 to the next higher position.
    - It iteratively performs this operation until the number becomes 0, and the number of iterations is equal to the number of set bits.
    - Time complexity: O(log n), similar to the looping method.

C
```
int count_set_bits_brian(unsigned int n) {
    int count = 0;
    while (n) {
        n &= n - 1;
        count++;
    }
    return count;
}
```

Comparison:

- **Efficiency:** In general, the looping and Brian Kernighan's algorithm methods are the most efficient with a time complexity of O(log n). The built-in `bin()` function approach can be less efficient, especially for integers with a high number of set bits.
- **Readability:** The looping method is arguably the most readable, while Brian Kernighan's algorithm might be less intuitive for beginners.

Choosing the right method:

- If readability is a top priority, the looping method is a good choice.
- If you need the absolute fastest performance, consider using Brian Kernighan's algorithm or checking for architecture-specific instruction sets that might offer even faster bit counting operations (e.g., `popcnt` on x86). However, the performance difference between these methods is usually negligible for most practical applications.

I hope this comprehensive explanation helps!

---

5. WAP whether a number is a power of 2 or not?

Based on the provided search results, here are the key approaches to determine whether a number is a power of 2 or not:

1. **Using the log2 function**:
   - If `log2(n)` is an integer, then `n` is a power of 2.
   - This is a straightforward mathematical approach.

2. **Iterative division by 2**:
   - Keep dividing the number by 2 until it becomes 1.
   - If at any point the number becomes odd and is not 1, then it is not a power of 2.
   - If the number becomes 1, then it is a power of 2.

3. **Bitwise operation**:
   - If `n & (n-1)` is 0, then `n` is a power of 2.

- This approach exploits the fact that for a power of 2 number, the binary representation has only one bit set.

Here's the C code implementing these approaches:

```c
#include <math.h>
#include <stdbool.h>
#include <stdio.h>
// Using log2 function
bool isPowerOfTwoLog(int n) {
    return (ceil(log2(n)) == floor(log2(n)));
}
// Using iterative division by 2
bool isPowerOfTwoIterative(int n) {
    if (n == 0)
        return false;
    while (n != 1) {
        if (n % 2 != 0)
            return false;
        n /= 2;
    }
    return true;
}
// Using bitwise operation
bool isPowerOfTwoBitwise(int n) {
    return (n && !(n & (n - 1)));
}

int main() {
```

```
    int num = 16;
    printf("Using log2 function: %s\n", isPowerOfTwoLog(num) ? "Yes" : "No");
    printf("Using iterative division: %s\n", isPowerOfTwoIterative(num) ? "Yes" : "No");
    printf("Using bitwise operation: %s\n", isPowerOfTwoBitwise(num) ? "Yes" : "No");
    return 0;
}
```

The output of this program for `num = 16` will be:

```
Using log2 function: Yes
Using iterative division: Yes
Using bitwise operation: Yes
```

All three approaches correctly identify that 16 is a power of 2. You can test the program with different input numbers to see the results.

Citations:

[1] https://www.geeksforgeeks.org/c-program-to-find-whether-a-no-is-power-of-two/
[2] http://www.trytoprogram.com/c-examples/c-program-to-test-if-a-number-is-a-power-of-2/
[3] https://www.includehelp.com/c-programs/c-program-to-check-number-is-power-of-2-or-not.aspx
[4] https://www.ritambhara.in/check-if-number-is-a-power-of-2/
[5] https://www.tutorialspoint.com/javascript-program-to-check-whether-a-given-number-is-power-of-2

6. WAP implements subtraction functionality without using SUB('-') Operator

Here's a C program that implements subtraction functionality without using the subtraction operator ('-') by using bitwise operators:

```c
#include <stdio.h>

int subtract(int x, int y) {
    // Iterate until there is no borrow
    while (y != 0) {
        // Borrow contains common set bits of y and unset bits of x
        int borrow = (~x) & y;

        // Subtraction of bits of x and y where at least one of the bits is not set
        x = x ^ y;

        // Borrow is shifted by one so that subtracting it from x gives the required sum
        y = borrow << 1;
    }
    return x;
}

int main() {
    int x = 29, y = 13;
    printf("x - y is %d\n", subtract(x, y));
    return 0;
}
```

The `subtract` function takes two integers `x` and `y` as input and returns their difference without using the subtraction operator.

Here's how it works:

1. The function enters a loop that continues until `y` becomes zero (i.e., there is no borrow).

2. Inside the loop:

- `borrow` is calculated as the bitwise AND of the one's complement of `x` and `y`. This represents the common set bits of `y` and the unset bits of `x`.

- `x` is updated by performing a bitwise XOR of `x` and `y`. This subtracts the bits of `y` from `x` where at least one of the bits is not set.

- `y` is updated by shifting the `borrow` left by one bit. This effectively divides `borrow` by 2, which is equivalent to shifting the borrow to the next bit position.

3. After the loop, `x` will contain the final result of the subtraction, and `y` will be zero.

4. The function returns `x`, which represents the difference between the original `x` and `y`.

In the `main` function, we initialize `x` to 29 and `y` to 13, and then call the `subtract` function with these values. The result of `x - y` is printed, which should be 16.

This approach uses bitwise operations to perform subtraction without relying on the subtraction operator. It works by iteratively calculating the borrow and updating `x` and `y` until there is no borrow left.

Citations:

[1] https://www.youtube.com/watch?v=YG1bDNtO5X8

[2] https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/subtraction-operator

[3] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Subtraction

[4] https://www.geeksforgeeks.org/subtract-two-numbers-without-using-arithmetic-operators/

[5] https://www.geeksforgeeks.org/subtract-1-without-arithmetic-operators/

(OR)

Absolutely, here's a C program that implements subtraction functionality without using the subtraction operator (-):

C

```c
#include <stdio.h>

int subtract(int minuend, int subtrahend) {
  """Subtracts two numbers without using the '-' operator.

  Args:
      minuend: The number from which to subtract.
      subtrahend: The number to subtract.

  Returns:
      The difference between minuend and subtrahend.
  """

  // Handle negative subtrahend by adding its two's complement
  if (subtrahend < 0) {
    subtrahend = ~subtrahend + 1;
  }

  // Perform bitwise addition with carry propagation
  int result = 0;
  int carry = 0;
  while (minuend || subtrahend || carry) {
    int xor_bit = minuend ^ subtrahend;
    int and_bit = minuend & subtrahend;
    result = result | (xor_bit ^ carry);
    carry = (and_bit << 1);
    minuend >>= 1;
    subtrahend >>= 1;
  }

  return result;
}

int main() {
  int a = 10;
  int b = 5;
  int difference = subtract(a, b);
  printf("Difference between %d and %d is: %d\n", a, b, difference);
  return 0;
}
```

Explanation:

1. **`subtract` function:**

- Takes `minuend` (number from which to subtract) and `subtrahend` (number to subtract) as arguments.
- Handles negative `subtrahend` by converting it to its two's complement using bitwise NOT (`~`) and adding 1. Two's complement is a representation of a negative number in binary form.
- Initializes `result` to 0 and `carry` to 0 for bitwise addition with carry propagation.
- Enters a loop that continues until all bits of both numbers and the carry are processed.
  - Calculates the XOR (^) of the corresponding bits in `minuend` and `subtrahend`. This gives the actual result bit.
  - Calculates the AND (&) of the corresponding bits in `minuend` and `subtrahend`. This determines if there's a carry to the next bit.
  - Updates `result` by performing a bitwise OR (|) with the XOR result and the carry.
  - Updates `carry` by shifting the AND result left by 1 bit (representing the carry).
  - Shifts both `minuend` and `subtrahend` right by 1 bit to process the next bits.
- Returns the final `result` as the difference.

2. **`main` function:**
   - Declares variables `a` and `b` for the numbers to subtract.
   - Calls the `subtract` function to get the difference and stores it in `difference`.
   - Prints the difference.

This code effectively performs subtraction without using the `-` operator by leveraging bitwise operations and two's complement representation.

---

7. WAP implements addition functionality without using ADD('+') Operator.

Certainly! Here's a C program that implements addition functionality without using the `+` operator:

C

```c
#include <stdio.h>

int add(int a, int b) {
  """Adds two numbers without using the '+' operator.

  Args:
      a: The first number to add.
      b: The second number to add.

  Returns:
      The sum of a and b.
  """

  // Handle negative numbers using two's complement
  while (b < 0) {
    int borrow = (~a) & 1;
    a = a ^ b;
    b = (a & 1) << 1 | borrow;
  }

  // Perform bitwise addition with carry propagation
  int result = 0;
  int carry = 0;
  while (a || b) {
    int xor_bit = a ^ b;
    int and_bit = a & b;
    result = result | xor_bit;
    carry = and_bit << 1;
    a >>= 1;
    b >>= 1;
  }

  return result;
}

int main() {
  int x = 10;
  int y = 5;
  int sum = add(x, y);
  printf("Sum of %d and %d is: %d\n", x, y, sum);
  return 0;
}
```

Explanation:

1. **`add` function:**
    - Takes two integers `a` and `b` as arguments.
    - Handles negative numbers by iteratively adding `b` (converted to two's complement) to `a` until `b` becomes non-negative. Two's complement is a way to represent negative numbers in binary form.
        - The `borrow` variable is calculated using the bitwise NOT (`~`) of `a` and AND (`&`) with 1. It essentially captures the carry from the previous addition.
        - `a` is updated using the XOR (`^`) of `a` and `b`. This effectively adds the current bits without considering the carry.
        - `b` is updated by shifting the AND result of `a` and 1 to the left by 1 bit (representing the carry) and ORing (`|`) it with the `borrow`.
    - Initializes `result` to 0 and `carry` to 0 for bitwise addition with carry propagation.
    - Enters a loop that continues until all bits of both numbers are processed.
        - Calculates the XOR of the corresponding bits in `a` and `b`. This gives the actual sum bit, considering the previous carry in `carry`.
        - Calculates the AND of the corresponding bits in `a` and `b`. This determines if there's a carry to the next bit position.
        - Updates `result` by performing a bitwise OR with the XOR result.
        - Updates `carry` by shifting the AND result left by 1 bit.
        - Shifts both `a` and `b` right by 1 bit to process the next bits.
    - Returns the final `result` as the sum.
2. **`main` function:**
    - Declares variables `x` and `y` for the numbers to add.
    - Calls the `add` function to get the sum and stores it in `sum`.
    - Prints the sum.

This code demonstrates how to add two integers without using the arithmetic addition operator by employing bitwise operations and two's complement representation for handling negative numbers.

---

8. WAP implements XOR functionality without using XOR(^) operator.

Absolutely! Here's a C program that implements XOR functionality without using the XOR (^) operator:

C

```
#include <stdio.h>

int xor(int a, int b) {
  """Implements XOR functionality without using the XOR operator.

  Args:
      a: The first operand.
      b: The second operand.

  Returns:
      The XOR of a and b.
  """

  // Use bitwise AND and left shift to create a carry bit
  int carry = a & b;
  // Use bitwise AND with the complement of the other operand to get the result bit
  int result = (~a & b) | (a & ~b);
  // Shift the carry bit one position to the left for the next iteration
  return result | (carry << 1);
}

int main() {
  int x = 10;  // Binary: 1010
  int y = 5;   // Binary: 0101
  int result = xor(x, y);
  printf("XOR of %d and %d is: %d\n", x, y, result);  // Output: XOR of 10 and 5 is: 15 (Binary: 1111)
  return 0;
}
```

Explanation:

1. **`xor` function:**
    - Takes two integers `a` and `b` as arguments.
    - Initializes a variable `carry` to 0. This will store the carry bit generated during the XOR operation.
    - Calculates the bitwise AND (`&`) of `a` and `b`. This identifies the bits that are set (1) in both operands. These bits will be 0 in the result.
    - Calculates the XOR result for each bit position:

        - `~a & b`: This performs a bitwise AND of the bitwise NOT (`~`) of `a` with `b`. This effectively sets the result bit to 1 if `a` is 0 and `b` is 1.
        - `a & ~b`: This performs a bitwise AND of `a` with the bitwise NOT of `b`. This sets the result bit to 1 if `a` is 1 and `b` is 0.
        - The bitwise OR (`|`) combines these two results, ensuring the result bit is set to 1 only if exactly one of `a` or `b` is 1.
    - Shifts the `carry` left by 1 bit (`<< 1`). This effectively moves the carry to the next bit position for the next iteration.
    - Returns the bitwise OR (`|`) of the calculated result and the shifted `carry`. This combines the current result bits with any carry bits from previous iterations.
2. **`main` function:**
    - Declares variables `x` and `y` for the numbers to perform XOR on.
    - Calls the `xor` function to get the result and stores it in `result`.
    - Prints the XOR of `x` and `y`.

This code leverages bitwise operations to achieve XOR functionality without using the dedicated XOR operator. It calculates the result bit by identifying positions where only one operand has a set bit (1) and combines it with any carry bits from previous operations.

---

9. WAP to implement the sizeof operation using the bitwise operator

It's important to note that implementing a completely accurate `sizeof` equivalent using only bitwise operators in C is not possible. The `sizeof` operator is a compile-time function that determines the size (in bytes) of a data type or variable based on the compiler and architecture. Bitwise operations operate on the data itself, not on metadata about the data.

However, we can create a function that estimates the size of an integer by counting the number of leading set bits (bits with a value of 1) starting from the most significant bit (MSB). This approach works reasonably well for most architectures where integer sizes are powers of 2 (e.g., 1, 2, 4, 8 bytes).

Here's a C program that implements this estimation:

C

```c
#include <stdio.h>

int estimate_size(unsigned int n) {
  """Estimates the size of an integer using bitwise operations.

  Args:
      n: The unsigned integer whose size is to be estimated.

  Returns:
      An estimate of the size of n in bytes, assuming powers of 2 for
integer sizes.
      This might not be accurate for all architectures and data types.
  """

  // Count the number of leading set bits
  int count = 0;
  while (n) {
    n >>= 1;
    count++;
  }

  // Assuming integer sizes are powers of 2, estimate the size based on
leading set bits
  return 1 << (count - 1); // Shift 1 left by (count - 1) positions
}

int main() {
  int a = 10; // Binary: 00001010
  int size = estimate_size(a);
  printf("Estimated size of integer %d: %d bytes\n", a, size);
  return 0;
}
```

Explanation:

1. **`estimate_size` function:**
    - Takes an unsigned integer `n` as input.
    - Initializes a variable `count` to 0 to store the number of leading set bits.
    - Enters a loop that continues as long as `n` is not zero.
        - Shifts `n` right by 1 bit (`>> 1`). This effectively discards the least significant bit (LSB).
        - Increments the `count` to track the number of bits shifted.

- Assuming integer sizes are powers of 2, the number of leading set bits corresponds to the exponent needed to raise 2 to get the size in bytes.
- Returns `1 << (count - 1)`. This performs a left shift of 1 by `(count - 1)` positions. For example, if `count` is 3 (3 leading set bits), this effectively calculates 2 raised to the power of (3 - 1), which is 2^2, resulting in an estimated size of 4 bytes.

2. **`main` function:**
   - Declares an integer variable `a`.
   - Calls the `estimate_size` function with `a` to get the estimated size and stores it in `size`.
   - Prints the estimated size of the integer `a`.

Limitations:

- This approach assumes integer sizes are powers of 2. This is generally true for most architectures, but there might be exceptions.
- It only works for unsigned integers. For signed integers, the sign bit would affect the leading set bit calculation.

While this code doesn't provide a perfect equivalent to the `sizeof` operator, it demonstrates how bitwise operations can be used to estimate the size of an integer under certain assumptions.

---

10. WAP to convert Little endian integer to Big endian integer

Certainly! Here's a C program that implements the conversion of a Little Endian integer to Big Endian integer using bitwise operations and shifting:

C

```
#include <stdio.h>

unsigned int convert_to_big_endian(unsigned int value) {
  """Converts a Little Endian integer to Big Endian.

  Args:
      value: The Little Endian integer to convert.

  Returns:
      The converted Big Endian integer.
  """
```

```
  // Assuming 4 bytes (32-bit integer)
  int num_bytes = 4;
  unsigned int result = 0;

  // Loop through each byte in Little Endian order and shift them to Big
Endian positions
  for (int i = 0; i < num_bytes; i++) {
    // Extract the current byte using bitwise AND and right shift
    unsigned int byte = (value >> (8 * i)) & 0xFF;
    // Shift the extracted byte to the corresponding Big Endian position
(num_bytes - 1 - i)
    result |= (byte << (8 * (num_bytes - 1 - i)));
  }

  return result;
}

int main() {
  unsigned int little_endian_value = 0x1234ABCD;
  unsigned int big_endian_value =
convert_to_big_endian(little_endian_value);

  printf("Little endian: 0x%X\n", little_endian_value);
  printf("Big endian: 0x%X\n", big_endian_value);

  return 0;
}
```

Explanation:

1. `convert_to_big_endian` **function:**
    - Takes an unsigned integer `value` in Little Endian format as input.
    - Defines `num_bytes` as 4, assuming a 32-bit integer (you can adjust this for different integer sizes).
    - Initializes `result` to 0 to store the converted Big Endian integer.
    - Loops through each byte (0 to `num_bytes` - 1) in Little Endian order:
        - Extracts the current byte using bitwise AND (`&`) with 0xFF and a right shift (`>>`) by 8 times `i`. This isolates the byte at position `i` from the least significant end (Little Endian).
        - Shifts the extracted byte (`byte`) left by 8 times the difference between `num_bytes` - 1 and the current position `i`. This effectively

moves the byte to its corresponding position in Big Endian order (most significant end).

- Performs a bitwise OR (`|`) with the `result` to accumulate the Big Endian representation byte by byte.

2. **`main` function:**
    - Declares a Little Endian integer `little_endian_value`.
    - Calls the `convert_to_big_endian` function to get the Big Endian version and stores it in `big_endian_value`.
    - Prints both the Little Endian and Big Endian values in hexadecimal format (`0x%X`).

This code effectively demonstrates how to convert a Little Endian integer to Big Endian by manipulating individual bytes using bitwise operations and shifting them to their appropriate positions in the Big Endian format.

---

11. WAP multiply a number by 9 using bit shift.

Here's a C program that multiplies a number by 9 using bit shift:

C

```
#include <stdio.h>

int multiply_by_9(int num) {
  """Multiplies a number by 9 using bit shift.

  Args:
      num: The number to multiply by 9.

  Returns:
      The product of num and 9.
  """

  // 9 in binary is 1001. Multiplying by 9 is equivalent to:
  // - Shifting the number left by 1 bit (adding a 0 to the LSB).
  // - Adding the original number to the shifted number.

  return (num << 3) + num;  // Efficiently multiply by 9 using bit shift
and addition
}

int main() {
  int number = 10;
  int product = multiply_by_9(number);
```

```
  printf("%d x 9 = %d\n", number, product);
  return 0;
}
```

Explanation:

1. **`multiply_by_9` function:**
    - Takes an integer `num` as input.
    - Multiplies `num` by 9 using bitwise operations and addition:
        - `num << 3`: This performs a left shift of `num` by 3 bits. In binary, this is equivalent to adding a 0 to the least significant end (LSB) and effectively multiplying `num` by 2^3 (which is 8).
        - `+ num`: This adds the original `num` to the left-shifted value.

Justification:

- The binary representation of 9 is 1001.
- Shifting `num` left by 1 bit adds a 0 to the LSB, essentially multiplying by 2^1 (which is 2).
- Shifting again by another bit (total of 3 bits) is the same as multiplying by 2^2 (which is 4) and adding the previous multiplication (2).
- Adding the original `num` to the left-shifted value accounts for the remaining 1 in the binary representation of 9.

This approach efficiently multiplies a number by 9 using bitwise operations, avoiding traditional multiplication for better performance in some cases.

---

12. WAP Check if the 20th bit of a 32-bit integer is on or off?

Here's a C program that checks if the 20th bit of a 32-bit integer is on or off:

C

```
#include <stdio.h>

int check_20th_bit(int num) {
  """Checks if the 20th bit of a 32-bit integer is on or off.

  Args:
      num: The 32-bit integer to check.

  Returns:
      1 if the 20th bit is on, 0 otherwise.
```

```
  """

  // Check if the result of AND operation with 1 << 19 (left shift 1 by
19) is non-zero
  // This effectively isolates the 20th bit and checks if it's set.
  return (num & (1 << 19)) != 0;
}

int main() {
  int number = 1048576;  // Binary: 00000010 00000000 00000000 10000000
(20th bit is on)
  int is_on = check_20th_bit(number);
  printf("Is the 20th bit of %d on? %d\n", number, is_on);
  return 0;
}
```

Explanation:

1. **`check_20th_bit` function:**
    - Takes an integer `num` as input.
    - Performs a bitwise AND (`&`) operation between `num` and `1 << 19`.
        - `1 << 19`: This left-shifts the binary number 1 (00000001) by 19 positions. This creates a new number with a single 1 set at the 20th bit position (from the least significant bit).
    - The AND operation compares each bit of `num` with the corresponding bit in `1 << 19`. If the 20th bit in `num` is 1, the AND operation will result in a 1 at that position.
    - The expression `(num & (1 << 19)) != 0` checks if the result of the AND operation is not zero. If it's not zero, it means the 20th bit in `num` was 1, so the function returns 1 (true). Otherwise, it returns 0 (false).
2. **`main` function:**
    - Declares an integer `number` with a sample value.
    - Calls the `check_20th_bit` function to check the 20th bit and stores the result in `is_on`.
    - Prints whether the 20th bit is on or off based on the returned value.

This approach efficiently checks the state of the 20th bit using bitwise operations without complex loops or conditional statements.

---

13. Write a functionsetbits(x,p,n,y)that return x with then bits that begin at position p set to the rightmost n bits of y, leaving the other bits unchanged.
e.g. if x = 10101010 (170 decimal) and y = 10100111 (167 decimal) and n = 3 and p = 6 then
you need to strip off 3 bits of y (111) and put them in x at position 10xxx010 to get answer
10111010. Your answer should print out the result in binary form although input can be in
decimal form.
Your output should be like this:
x = 10101010 (binary)
y = 10100111 (binary)
setbits n = 3, p = 6 gives x = 10111010 (binary).

Here's the C code for the `setbits` function that replaces the specified bits in `x` with the rightmost `n` bits of `y`:

C

```c
#include <stdio.h>

unsigned int setbits(unsigned int x, int p, int n, unsigned int y) {
  """Sets the n bits of x starting at position p to the rightmost n bits of y.

  Args:
      x: The integer to modify.
      p: The starting position of the bits to set (0-indexed).
      n: The number of bits to set.
      y: The integer from which to get the bits to set.

  Returns:
      The modified integer with the specified bits set.
  """

  // Create a mask to isolate the bits to be cleared in x
  unsigned int mask = ~((~0) << n) << p;

  // Extract the rightmost n bits from y
  unsigned int rightmost_n_bits = y & ((~0) << (32 - n));

  // Clear the bits in x at position p using the mask
  x &= mask;

  // Shift the extracted bits to the correct position in x
  unsigned int shifted_bits = rightmost_n_bits << p;

  // Set the bits in x using bitwise OR
  return x | shifted_bits;
```

```
}

int main() {
  unsigned int x = 170;  // Binary: 10101010
  unsigned int y = 167;  // Binary: 10100111
  int n = 3;
  int p = 6;
  unsigned int result = setbits(x, p, n, y);
  printf("x = %u (binary)\n", x);
  printf("y = %u (binary)\n", y);
  printf("setbits n = %d, p = %d gives x = %u (binary)\n", n, p, result);
  return 0;
}
```

Explanation:

1. `setbits` **function:**
    - Takes `x` (integer to modify), `p` (starting position), `n` (number of bits to set), and `y` (source integer) as arguments.
    - Creates a `mask` to clear the bits in `x` at the specified position.
        - `~((~0) << n)`: This creates a number with the rightmost `n` bits as 1s and the remaining bits as 0s. Inverting it with `~` flips the pattern, resulting in a number with 0s in the rightmost `n` positions.
        - `<< p`: This left-shifts the mask by `p` positions to align it with the target bits in `x`.
    - Extracts the rightmost `n` bits from `y` using a mask with all bits set except the leftmost `32 - n` bits (`(~0) << (32 - n)`). Performing a bitwise AND (`&`) with `y` isolates the desired bits.
    - Clears the bits in `x` at position `p` using the mask and bitwise AND (`&`).
    - Shifts the extracted rightmost `n` bits from `y` to the correct position in `x` using a left shift (`<< p`).
    - Sets the cleared bits in `x` with the shifted bits from `y` using a bitwise OR (`|`).
    - Returns the modified `x`.
2. `main` **function:**
    - Declares variables for `x`, `y`, `n`, and `p` with sample values.
    - Calls the `setbits` function to get the modified `x` and stores it in `result`.

- Prints `x`, `y`, `n`, `p`, and the resulting `x` in binary format for clarity.

14. WAP Swap any two numbers using bitwise operators. How does it work?

Here's a C program that swaps two numbers using bitwise XOR (`^`) and logical AND (`&`) operators, along with an explanation of how it works:

C

```
#include <stdio.h>

void swap(int *x, int *y) {
  """Swaps the values of two integers using bitwise operators.

  Args:
      x: Pointer to the first integer.
      y: Pointer to the second integer.
  """

  *x = *x ^ *y;  // XOR x with y and store the result in x
  *y = *x ^ *y;  // XOR the new value of x with y and store the result in
y
  *x = *x ^ *y;  // XOR the new value of y with x (original x) and store
the result in x
}

int main() {
  int a = 5;
  int b = 10;
  printf("Before swap: a = %d, b = %d\n", a, b);
  swap(&a, &b);
  printf("After swap: a = %d, b = %d\n", a, b);
  return 0;
}
```

Explanation:

1. `swap` **function:**
    - Takes pointers to two integers `x` and `y` for in-place swapping.
    - Performs the swap using three XOR operations:
        - `*x = *x ^ *y`: This XORs the value of `x` with `y`. The XOR operation has the property that XORing a number with itself results in zero (0). So, the bits that are the same in both `x` and `y` become zero in the result, and the bits that differ become one. This effectively stores the "differences" between `x` and `y` in the new value of `x`.

        - `*y = *x ^ *y`: Here, the new value of `x` (containing the "differences") is XORed with the original value of `y`. This "undoes" the XOR operation for the bits that were originally the same in `x` and `y`, leaving only the bits that were originally different in `y`. These bits are now stored in the new value of `y`.
        - `*x = *x ^ *y`: Finally, the new value of `y` (containing the original "differences" of `x` and `y`) is XORed with the original value of `x`. This "undoes" the XOR for the bits that were originally different in `x` (now stored in `y`), leaving the original value of `y` in `x`.
2. **`main` function:**
    - Declares variables `a` and `b` for the numbers to swap.
    - Prints the values before swapping.
    - Calls the `swap` function with the addresses of `a` and `b` to modify the original variables.
    - Prints the values after swapping to demonstrate the successful exchange.

How it works:

The key lies in the XOR operation's properties:

- `a ^ a = 0` (XORing a number with itself results in zero).
- `a ^ b ^ a = (a ^ a) ^ b = 0 ^ b = b` (XORing a number with itself twice and then with another number results in the other number).

By exploiting these properties, the code effectively uses XORs to create temporary storage for the "differences" between `x` and `y` and then uses them to swap the values without needing a temporary variable.
