1. Using some test cases, match these bit operations to their associated function:

1. `x & 1`	a) Return x without trailing 1s (e.g. 11011111 becomes 11000000)
2. `x & (1 « n)`	b) Unset the n_{th} bit
3. `x & ~(1 « n)`	c) Return true if n_{th} bit is set
4. `(x ^ y) < 0`	d) Return the minimum of x and y
5. `y ^ ((x ^ y) & -(x < y))`	e) Return true if x and y have opposite signs
6. `x & (x - 1)`	f) Return true if x is odd, false if x is even
7. `x & (x + 1)`	g) Return 0 if x is a power of 2 for $x > 0$

Solution:

1. **f**
2. **c**
3. **b**
4. **e**
5. **d**
6. **g**
7. **a**

2. The following C "optimizations" are said to improve the performance of embedded systems. In reality, some of them are useless or even counterproductive on certain architectures. For each of the "optimizations" given,

   - Find out why it optimizes performance on some architectures
   - Find out if there are any targets on which it does not improve performance, or decreases performance
   - On the architectures on which it improves performance, how great is the improvement? (e.g., one instruction overall, one instruction per iteration of a loop, etc.) Is the improvement significant or trivial?

   Here are the "optimizations":

   (a) Count down to zero, not up to N, in `for()` loops

   (b) Avoid the % operation

   (c) Use an 8-bit `unsigned char` whenever you have a value that you know won't go beyond 0-255 (e.g., some loop index variables)

Solution:

(a) Count down to zero loops make use of SUBS along with Z flag whereas if it is up to N loops then it uses ADD and CMP instruction pair along with an extra register to store N. Hence using count down saves an instruction as well as a register.

(b) The % operator is a combination of division, multiplication and subtraction and hence it uses more number of cycles. It is better to avoid the % operator. It also takes more cycles than other basic instructions and hence decreases the performance.

(c) 8 bit unsigned char uses lesser memory compared to other data types. Doesn't decrease performance. When using 32 bit MCU, manipulating a 8-bit variable will be less efficient than a 32 bit. Hence it may lead to a waste of space when storing in memory.

3. Refer to the JPL Institutional Coding Standard for the C Programming Language (http://lars-lab.jpl.nasa.gov/JPL_Coding_Standard_ext.pdf). This standard describes their rules for mission critical flight software written in the C programming language. (The NASA Jet Propulsion Laboratory was responsible for the Mars Curiosity rover.)
   (a) Why is recursion not permitted in mission critical flight software?
   (b) Why is dynamic memory allocation disallowed after task initialization in mission critical flight software?

Solution:

(a) Recursion is not permitted in mission-critical flight software due to its potential to lead to unbounded stack usage, which can cause stack overflow and, consequently, system failure. Since mission-critical systems must operate with high reliability and predictability, the risk of running out of stack space due to recursive calls is unacceptable. Recursion makes it difficult to analyze and prove the software's maximum stack usage, which is crucial for ensuring that the system can handle its worst-case execution scenario without failing.

(b) Dynamic memory allocation is disallowed after task initialization in mission-critical flight software to prevent runtime errors related to memory fragmentation, memory leaks, and allocation failures. In a real-time or high-reliability system, such as those used in space exploration, these issues can lead to unpredictable behavior, system instability, or crashes. By allocating all necessary memory upfront and avoiding further dynamic allocation, the system's behavior becomes more predictable, and the risk of running out of memory or encountering fragmentation-related issues is minimized. This approach ensures that the software maintains its reliability and performance throughout its operation.

4. Fill in the blanks with the word “signed” or “unsigned”:

(a) In _____arithmetic, if the overflow flag (V in CPSR) is set on an operation, the result is wrong.

(b) In _____arithmetic, the overflow flag (V in CPSR) does not indicate anything meaningful about the result of the operation.

(c) In _____arithmetic, if the carry flag (C in CPSR) is set on an operation, the result is wrong.

(d) In _____arithmetic, the carry flag (C in CPSR) does not indicate anything meaningful about the result of the operation.

Solution:

(a) signed

(b) unsigned

(c) unsigned

(d) signed

5. Describe the status of the N, Z, C, and V flags of the CPSR after each of the following:

(a)
```
ldr     r1, =0xffffffff
ldr     r2, =0x00000001
add     r0, r1, r2
```

(b)
```
ldr     r1, =0xffffffff
ldr     r2, =0x00000001
cmn     r1, r2
```

(c)
```
ldr     r1, =0xffffffff
ldr     r2, =0x00000001
adds    r0, r1, r2
```

(d)
```
ldr     r1, =0xffffffff
ldr     r2, =0x00000001
addeq   r0, r1, r2
```

(e)
```
ldr     r1, =0x7fffffff
ldr     r2, =0x7fffffff
adds    r0, r1, r2
```

Solution:

(a) None the flags are updated since there is no update status

(b) N=0, Z=1, C=1, V=0

(c) N=0, Z=1, C=1, V=0

(d) None the flags are set

(e) N=1, V=0

6. The following C code implements the Euclid algorithm for calculating the greatest common divisor:

```
int gcd(int a, int b)
{
    while (a != b)
      {
        if (a > b)
            a = a - b;
        else
            b = b - a;
      }
    return a;
}
```

Here is an equivalent ARM assembly routine that only uses conditional execution on the branch instructions:

```
gcd
        CMP     r1, r2
        BEQ     end
        BLT     lessthan
        SUB     r1, r1, r2
        B       gcd
lessthan
        SUB     r2, r2, r1
```

```
        B       gcd
end
        ---
```

And here is an equivalent ARM assembly routine that uses full conditional execution :

```
gcd
        CMP     r1, r2
        SUBGT   r1, r1, r2
        SUBLT   r2, r2, r1
        BNE     gcd
```

Assume a is 54 and is loaded into r1, b is 24 and is loaded into r2.

(a) Run through the C algorithm until its completion to find the greatest common divisor.

Solution:

Starting values: a = 54, b = 24.

Iteration 1: a > b, so a = 54 - 24 = 30. New values: a = 30, b = 24.

Iteration 2: a > b, so a = 30 - 24 = 6. New values: a = 6, b = 24.

Iteration 3: b > a, so b = 24 - 6 = 18. New values: a = 6, b = 18.

Iteration 4: b > a, so b = 18 - 6 = 12. New values: a = 6, b = 12.

Iteration 5: b > a, so b = 12 - 6 = 6. New values: a = 6, b = 6.

Iteration ends because a = b.

GCD is 6.

(b) Run through the ARM assembly version without full conditional execution.

Solution:

The assembly routine without full conditional execution directly mirrors the logic of the C code, using conditional branch instructions to determine the flow of execution.

CMP compares r1 and r2.

BEQ ends the loop if r1 equals r2.

BLT branches if r1 is less than r2, leading to the subtraction in the lessthan label.

SUB (inside the loop) subtracts r2 from r1 or vice versa based on the comparison, replicating the if-else structure of the C code.

This process iterates, reducing r1 and r2 until they are equal, at which point the GCD is found. The exact number of iterations and cycles depends on the specific values of r1 and r2 and the efficiency of the conditional execution.

(c) Run through the ARM assembly version with full conditional execution.

Solution:

The version with full conditional execution uses conditional instructions (SUBGT and SUBLT) to perform the subtraction based on the result of the comparison without branching to different labels. This approach can potentially reduce the number of branches and make the code more efficient on certain architectures.

CMP compares r1 and r2.

SUBGT subtracts r2 from r1 if r1 is greater than r2.

(d) Refer to the ARM Cortex-M4 Technical Reference Manual (available online) to find out the timing of each instruction. How many cycles does the first ARM routine take? How many cycles does the second ARM routine take?

Solution:

Without access to the ARM Cortex-M4 Technical Reference Manual in this environment, I can provide a general approach to determining the cycle count:

Each instruction in the ARM architecture typically consumes a specific number of cycles.

CMP, SUB, and B instructions have their own cycle counts, which can be looked up in the manual.

Conditional execution may save cycles by reducing the number of branches, which can be costly in terms of cycles due to pipeline flushing or stalling.

For a precise cycle count, one would need to reference the technical manual for the specific cycle costs of each instruction and add them up for each iteration of the loop, considering the optimizations made by using conditional execution.