EL-GY 6483

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: The following table gives the solutions:

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

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:

Donald Knuth is often quoted as saying,

> We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil.

Micro-optimizations in general are often not useful or actively harmful. Specifically,

- Micro-optimizations don't help if your code is not logically correct, and they can often introduce logical errors since they typically involve writing code that is less intuitive and natural.
- Embedded systems typically involve a specific, focused application with specific deadline and size constraints. If your program already meets those constraints, it fully satisfies the system requirements and micro-optimizations are pointless.
- Micro-optimizations may potentially be useful in an area of your code that is
  - highly inefficient, and
  - where your system spends a lot of time.

  A reasonable approach is therefore to design first, code from the design and then profile/benchmark the result to see which parts should be optimized.

(a) Count down to zero in `for()` loops

The reason this is often more efficient is because "compare to zero" is a "free" side effect of many instructions. For example, in the ARM instruction set, we saw instructions like ADDS or SUBS that perform an arithmetic operation (and store the result in a register) and also compare the result to zero (and set the Z flag to the outcome of the comparison) in the same operation. We don't have the same advantage in comparing to some arbitrary N.

This example comes from the ARM Information Center. The incrementing version,

```c
int fact1(int n)
{
    int i, fact = 1;
    for (i = 1; i <= n; i++)
        fact *= i;
    return (fact);
}
```

compiles to

```
fact1 PROC
    MOV      r2, r0
    MOV      r0, #1
    CMP      r2, #1
    MOV      r1, r0
    BXLT     lr
|L1.20|
    MUL      r0, r1, r0
    ADD      r1, r1, #1
    CMP      r1, r2
    BLE      |L1.20|
    BX       lr
    ENDP
```

while the decrementing version,

```
int fact2(int n)
{
    unsigned int i, fact = 1;
    for (i = n; i != 0; i--)
        fact *= i;
    return (fact);
}
```

compiles to

```
fact2 PROC
    MOVS     r1, r0
    MOV      r0, #1
    BXEQ     lr
|L1.12|
    MUL      r0, r1, r0
    SUBS     r1, r1, #1
    BNE      |L1.12|
    BX       lr
    ENDP
```

Depending on the situation, however, it may or may not be necessary or helpful for the programmer to attempt this kind of micro-optimization. For example, if the value of `n` for the incrementing version is known at compile time, and `n` is only a loop index variable that is not used anywhere else, then the compiler may perform this optimization itself (possibly together with other, more sophisticated, loop optimization techniques).

Note that counting down to zero is a very natural and common procedure in a certain area of embedded systems: timers.

(b) Avoid the % operation

The modulo operation is typically implemented using a divide instruction, which is an expensive operation on many architectures. On platforms with a DIV instruction, this instruction typically takes several times more cycles than other basic instructions, but this can still be tolerable. On platforms without a DIV instruction, however, division is implemented in software and can take several orders of magnitude longer than other basic instructions.

For example, consider the following benchmarks reported for a modulo-heavy test case. The ARM Cortex has native integer division (which can be used to compute the modulus) in 2-12 cycles, but the AVR and MSP430 do not, and the same operation may take several hundred cycles:

```
AVR:  29,825 cycles
MSP430: 27,019 cycles
ARM Cortex: 390 cycles
```

(Benchmark source: Nigel Jones, “Efficient C Tip #13 - use the modulus (%) operator with caution,” Embedded Gurus)

If you are always calculating a number mod some power of 2, you can implement it much more efficiently using bitwise AND. That is,

```
x % 2^n
```

is the same as

```
x & (2^n - 1)
```

If your value of `n` is hard-coded or the compiler is otherwise sure that it will always be a power of 2, the compiler may do this optimization for you.

Note that the modulo operation is important in embedded systems because many cryptography schemes use it heavily. An embedded system that will need crypto should generally use a microcontroller with native mod support!

(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)

Usually the stated goal of this advice is to save on memory space. If you are programming an 8-bit microcontroller, using 8-bit variables whenever possible *will* often be more efficient and save memory.

However, if you are using a 32-bit microcontroller, manipulating an 8-bit variable can be less efficient than a 32-bit one. The microcontroller may need to pad these variables before doing arithmetic on them, and depending on the address alignment, it may also need to waste space when storing them 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:

From the JPL document:

(a) Recursion:

> The presence of statically verifiable loop bounds and the absence of recursion prevent runaway code, and help to secure predictable performance for all tasks. The absence of recursion also simplifies the task of deriving reliable bounds on stack use *[note: avoiding stack overflow]*. The two rules combined secure a strictly acyclic function call graph and control-flow structure, which in turn enhances the capabilities for static checking tools to catch a broad range of coding defects.

(b) Dynamic memory allocation:

> This rule is common for safety and mission critical software and appears in most coding guidelines. The reason is simple: memory allocators and garbage collectors often have unpredictable behavior that can significantly impact performance. A notable class of coding errors stems from mishandling memory allocation and free routines: forgetting to free memory or continuing to use memory after it was freed, attempting to allocate more memory than physically available, overstepping boundaries on allocated memory, using stray pointers into dynamically allocated memory, etc. Forcing all applications to live within a fixed, pre-allocated, area of memory can eliminate many of these problems and make it simpler to verify safe memory use.

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) The `add` operation without the `s` suffix does not update CPSR flags.

(b) `cmn` works exactly like `add`, just without storing the results. The result of the add operation would be 0x100000000, but the top bit is lost because it does not fit into the 32-bit destination register and so the real result is 0x00000000. In this case, the flags will be set as follows: N = 0, Z = 1, C = 1, V = 0

(c) The result of the add operation would be 0x100000000, but the top bit is lost because it does not fit into the 32-bit destination register and so the real result is 0x00000000. In this case, the flags will be set as follows: N = 0, Z = 1, C = 1, V = 0

Here are the signed/unsiged interpretations of this operation:

```
      Signed:         -1 adds          1 =          0
    Unsigned: 4294967295 adds          1 =          0
 Hexadecimal: 0xffffffff adds 0x00000001 = 0x00000000
```

Note that the result is wrong according to the *unsigned* interpretation and correct according to the *signed* interpretation. The machine does not care which interpretaion the program is using; values are stored and flags are set the same way, regardless of the programmer's intentions.

(d) The `add` operation without the `s` suffix does not update CPSR flags.

(e) This triggers a signed overflow. The flags will be set as follows: N = 1, Z = 0, C = 0, V = 1

Here are the signed/unsiged interpretations of this operation:

```
      Signed: 2147483647 adds 2147483647 =         -2
    Unsigned: 2147483647 adds 2147483647 = 4294967294
 Hexadecimal: 0x7fffffff adds 0x7fffffff = 0xfffffffe
```

Note that the result is wrong according to the *signed* interpretation and correct according to the *unsigned* interpretation. The machine does not care which interpretaion the program is using; values are stored and flags are set the same way, regardless of the programmer's intentions.

As a rule, you can determine whether the V flag will be set by checking the 31st bit of the operands and the result. If:

- The 31st bit of both operands are the same (either both 0, indicating that both operands are positive numbers, or both 1, indicating that both operands are negative numbers), AND
- The 31st bit of the result is the opposite of the operands,

then the V flag will be set.

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 (we looked at this version in class):

```
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: First iteration:

```
    while (a != b) {      // 54 != 24
        if (a > b)        // True
            a = a - b;    // a = 54-24 = 30
        else              // False
            b = b - a;
      }                   // return to top of while loop
```

Second iteration:

```
    while (a != b) {      // 30 != 24
        if (a > b)        // True
            a = a - b;    // a = 30-24 = 6
        else              // False
            b = b - a;
      }                   // return to top of while loop
```

Third iteration:

```
    while (a != b) {      // 6 != 24
        if (a > b)        // False
            a = a - b;
        else              // True
            b = b - a;    // b = 24-6 = 18
      }                   // return to top of while loop
```

Fourth iteration:

```
    while (a != b) {      // 6 != 18
        if (a > b)        // False
            a = a - b;
        else              // True
            b = b - a;    // b = 18-6 = 12
      }                   // return to top of while loop
```

Fifth iteration:

```
    while (a != b) {      // 6 != 12
        if (a > b)        // False
            a = a - b;
        else              // True
            b = b - a;    // b = 12-6 = 6
      }                   // return to top of while loop
```

Sixth iteration:

```
    while (a != b) {      // 6 == 6, drop out of while loop
      ...

      }
     return a;            // Finished, return 6
```

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

Solution:
Note: unless otherwise specified, all flags are set to zero.
First iteration:

```
gcd
        CMP        r1, r2;       ; do 54 - 24, C flag set to 1
        BEQ        end           ; not executed
        BLT        lessthan      ; not executed
        SUB        r1, r1, r2    ; do 54 - 24, store 30 in r1
        B          gcd           ; go back to gcd
```

Second iteration:

```
gcd
        CMP        r1, r2;       ; do 30 - 24, C flag set to 1
        BEQ        end           ; not executed
        BLT        lessthan      ; not executed
        SUB        r1, r1, r2    ; do 30 - 24, store 6 in r1
        B          gcd           ; go back to gcd
```

Third iteration:

```
gcd
        CMP        r1, r2;       ; do 6 - 24, N flag set to 1
        BEQ        end           ; not executed
        BLT        lessthan      ; is executed, go to lessthan
        SUB        r1, r1, r2
        B          gcd
lessthan
        SUB        r2, r2, r1    ; do 24 - 6, store 18 in r2
        B          gcd           ; go back to gcd
```

Fourth iteration:

```
gcd
        CMP        r1, r2;       ; do 6 - 18, N flag set to 1
        BEQ        end           ; not executed
        BLT        lessthan     ; is executed, go to lessthan
        SUB        r1, r1, r2
        B          gcd
lessthan
        SUB        r2, r2, r1   ; do 18 - 6, store 12 in r2
        B          gcd          ; go back to gcd
```

Fifth iteration:

```
gcd
        CMP        r1, r2;      ; do 6 - 12, N flag set to 1
        BEQ        end          ; not executed
        BLT        lessthan     ; is executed, go to lessthan
        SUB        r1, r1, r2
        B          gcd
lessthan
        SUB        r2, r2, r1   ; do 12 - 6, store 6 in r2
        B          gcd          ; go back to gcd
```

Sixth iteration:

```
gcd
        CMP        r1, r2;      ; do 6 - 6, Z, C flags set to 1
        BEQ        end          ; is executed, go to end
        BLT        lessthan
        SUB        r1, r1, r2
        B          gcd
end
        ...                  ; finished; gcd was 6
```

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

Solution:
Note: unless otherwise specified, all flags are set to zero.
First iteration:

```
gcd
        CMP     r1, r2;      ; do 54 - 24, C flag set to 1
        SUBGT   r1, r1, r2   ; this is executed, store 30 in r1
        SUBLT   r2, r2, r1   ; not executed
        BNE     gcd          ; go back to gcd
```

Second iteration:

```
gcd
        CMP     r1, r2;      ; do 30 - 24, C flag set to 1
        SUBGT   r1, r1, r2   ; this is executed, store 6 in r1
        SUBLT   r2, r2, r1   ; not executed
        BNE     gcd          ; go back to gcd
```

Third iteration:

```
gcd
        CMP     r1, r2;      ; do 6 - 24, N flag set to 1
        SUBGT   r1, r1, r2   ; not executed
        SUBLT   r2, r2, r1   ; this is executed, store 18 in r2
        BNE     gcd          ; go back to gcd
```

Fourth iteration:

```
gcd
        CMP     r1, r2;      ; do 6 - 18, N flag set to 1
        SUBGT   r1, r1, r2   ; not executed
        SUBLT   r2, r2, r1   ; this is executed, store 12 in r2
        BNE     gcd          ; go back to gcd
```

Fifth iteration:

```
gcd
        CMP     r1, r2;      ; do 6 - 12, N flag set to 1
        SUBGT   r1, r1, r2   ; not executed
        SUBLT   r2, r2, r1   ; this is executed, store 6 in r2
        BNE     gcd          ; go back to gcd
```

Sixth iteration:

```
gcd
        CMP     r1, r2;      ; do 6 - 6, Z, C flags set to 1
        SUBGT   r1, r1, r2   ; not executed
        SUBLT   r2, r2, r1   ; not executed
        BNE     gcd          ; not executed, end with gcd being 6
```

(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:
From the ARM Technical Reference Manual, we find that:

- CMP, SUB take 1 cycle (as do SUBLT, SUBGT)
- BEQ, BLT, BNE take 1 cycle if not executed and 1 + P cycles if executed
- B takes 1 + P cycles

P is the number of cycles required for a pipeline refill. This ranges from 1 to 3.
The first routine takes

- First, second iterations: 5 + P cycles each
- Third, fourth, fifth iterations: 5 + 2P cycles each
- Sixth iteration: 2 + P cycles

for a total of 27 + 9P cycles.
The second routine takes

- First through fifth iterations: 4 + P cycles each
- Sixth iteration: 4 cycles

for a total of 24 + 5P cycles (3 + 4P cycles fewer than the previous routine).