EL-GY 6483

NetID: AA2537
NYUID: N15996814

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)

① The reason why it optimizes in some architectures is that: Comparing to zero is much more efficient in many situations than comparing to some other number N.

②

⇒ Decrease performance example:

```
Void function2 (int number){
unsigned int i;
unsigned int sum = 0;
for ( i=number ; i != 0 ; i--){
    sum += i
}
Printf("the sum is:", sum)
```

③ This optimization improves the performance when N is known when the program is compiling, and this number remains unchange and never used elsewhere. In other situations, like not comparing to a specific number to terminate the loop, the performance will not be improved.

b)

① The reason why it optimizes in some architectures is that: % operation will use division, which is extremly inefficient and expasive on most of architectures.

② This optimization works in % operation and improve the performance significantly by using basic instructions. Once it implements, it improves the performance.

③ Basic instruction division will be much faster than % operation

C)

① The reason why it optimizes in some architectures is that: It saves memory space. This improvement is significant on 8-bit microcontroller.

② This optimization will decrease the performance on 32-bit microcontrollers. More operations maybe needed if it's implemented on 32-bit microcontrollers.

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) this answere refers to JPL_Coding_Standard_ext.pdf page 10.

> 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. 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) this answere refers to JPL_Coding_Standard_ext.pdf page 10.

> Specifically, this rule disallows the use of malloc(), sbrk(), alloca(), and similar routines, after task initialization.
>
> 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) N, Z, C, V-flags of the CPSR have no update, since the program use add operation instead of adds operation.

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

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

d) N, Z, C, V flags of the CPSR have no update, since the program use add operation instead of adds operation.

e)
N=1, Z=0, C=0, V=1

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:

①Iteration 1=

```
int gcd(int a, int b)
{
    while (a != b)          54 != 24. True
      {
        if (a > b)          54 > 24 True
            a = a - b;      a=30
        else                False, Skip.
            b = b - a;      skip
      }                     next iteration
    return a;
}
```

②Iteration2

```
int gcd(int a, int b)
{
    while (a != b)          30 != 24  True
      {
        if (a > b)          30 > 24  True
            a = a - b;      a = 6
        else                False  skip
            b = b - a;      skip
      }                     next iteration
    return a;
}
```

③Iteration3.

```
int gcd(int a, int b)
{
    while (a != b)          6!=24 True
      {
        if (a > b)          False skip
            a = a - b;      skip
        else                True
            b = b - a;      b = 18
      }                     next iteration
    return a;
}
```

```
int gcd(int a, int b)
{
    while (a != b)          6!=18
      {
        if (a > b)          False skip
            a = a - b;      skip
        else                True
            b = b - a;      b= 12
      }                     next iteration
    return a;
}
```

```
int gcd(int a, int b)
{
    while (a != b)        6!= 12 True
     {
       if (a > b)         False skip
          a = a - b;      skip
       else               True
          b = b - a;      b= 6
     }                    next iteration
    return a;
}
```

```
int gcd(int a, int b)
{
    while (a != b)        6=6 False skip
     {
       if (a > b)         skip
          a = a - b;      skip
       else               skip
          b = b - a;      skip
     }
    return a;             return 6.
}
```

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

① Iteration.

```
gcd
        CMP       r1, r2        do 54-24, C=1
        BEQ       end           skip
        BLT       lessthan      skip
        SUB       r1, r1, r2    do 54-24, store 30 into r1
        B         gcd           go back to gcd
lessthan
        SUB       r2, r2, r1
        B         gcd
end

gcd
        CMP       r1, r2        do 30-24, C=1
        BEQ       end           skip
        BLT       lessthan      skip
        SUB       r1, r1, r2    do 30-24, store 6 into r1
        B         gcd           go back to gcd
lessthan
        SUB       r2, r2, r1
        B         gcd
end

gcd
        CMP       r1, r2        do 6-24, N=1
        BEQ       end           skip
        BLT       lessthan      go to less than
        SUB       r1, r1, r2
        B         gcd
lessthan
        SUB       r2, r2, r1    do 24-6, store 18 into r2
        B         gcd
end

gcd
        CMP       r1, r2        do 6-18, N=1
        BEQ       end           skip
        BLT       lessthan      go to less than
        SUB       r1, r1, r2
        B         gcd
lessthan
        SUB       r2, r2, r1    do 18-6, store 12 into r2
        B         gcd           go back to gcd
end
```

```
gcd
        CMP         r1, r2      do 6-12 , N = 1
        BEQ         end         skip
        BLT         lessthan    go to less than
        SUB         r1, r1, r2
        B           gcd
lessthan
        SUB         r2, r2, r1  do 12-6, store 6 into r2
        B           gcd         go back to gcd
 end
```

```
gcd
        CMP         r1, r2      do 6-6 , C = 1
        BEQ         end         go to end
        BLT         lessthan
        SUB         r1, r1, r2
        B           gcd
lessthan
        SUB         r2, r2, r1
        B           gcd
 end                            Finish. gcd=6
```

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

```
gcd
        CMP         r1, r2      do 30-24 C=1
        SUBGT       r1, r1, r2  store 30 into r1
        SUBLT       r2, r2, r1  skip
        BNE         gcd         back to gcd

gcd
        CMP         r1, r2      do 30-24 C=1
        SUBGT       r1, r1, r2  store 6 into r1
        SUBLT       r2, r2, r1  skip
        BNE         gcd         back to gcd

 gcd
        CMP         r1, r2      do 6-24, N=1
        SUBGT       r1, r1, r2  skip
        SUBLT       r2, r2, r1  store 18 in r2
        BNE         gcd         back to gcd
```

```
gcd
        CMP         r1, r2      do 6-18, N=1
        SUBGT       r1, r1, r2  skip
        SUBLT       r2, r2, r1  store 12 in r2
        BNE         gcd         back-to gcd

gcd
        CMP         r1, r2      do 6-12, N=1
        SUBGT       r1, r1, r2  skip
        SUBLT       r2, r2, r1  store 6 in r2
        BNE         gcd         back-to gcd

gcd
        CMP         r1, r2      do 6-6, C=1
        SUBGT       r1, r1, r2  skip
        SUBLT       r2, r2, r1  skip
        BNE         gcd         skip, gcd=6, end
```

(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:

Q1: According to Reference Manual:
① SUB and CMP = 1 cycle
② B: 1+P cycles. (Required by pipline refill, range from 1 to 3)
③ BEQ, BNE, BLT = 0 or 1+P (if executed).

Q2: First routine:
Iteration 1: 3+P, Iteration 2: 3+P, Iteration 3: 3+2P, Iteration 4: 3+2P,
Iteration 5: 3+2P, Iteration 6: 2+P
Sum = 27+9P

Q3. Second routine: Iteration 1 to 5 are all 4+P, sixth is 4

Sum = (4+P)×5 + 4 = 20+5P+4 = 24+5P