EL-GY 6483

NAME: YOGYA SHARMA

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 { X = 00010100 : X & 1 = 00000000 (0) / X = 00010101 : X & 1 = 00000001 (1)
2. c { X = 1110 1110 → X & (1 << 5) / = 1110 1110 & 00100000 = 00100000 : 5th bit set
3. b { X = 1110 1110 → X & ~(1 << 5) / = 1110 1110 & 1101 1111 = 1100 1110 : 5th bit unset
4. e { X = 11101110 Y = 01001100 : X^Y = 10100010 / X^Y < 0 : True (7th bit is set, result < 0)
5. d { X = 1001 0101 Y = 1110 0001 / X^Y = 0111 0100 : -(X<Y) = 11111111 / Y^((X^Y) & -(X<Y)) = 1001 0101 = X
6. g { Y = 0100 0000 (64d) X - 1 = 0011 1111 / X & (X-1) = 0000 0000 : returns 0
7. a { X = 1001 1111 X-1 = 1010 0000 / X & (X+1) = 1000 0000 : no trailing 1's

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) If we count down to 0 rather than counting up to N in a for loop, we might have the following benefits:

- Might save a register
- Might get a compare instruction with a smaller binary encoding with a decreased paired encoding, you could get a look at guidance.
- If a previous instruction sets a flag, it reduces the need of an explicit compare instruction.

With the incrementing loop i<N needs needs to be tested each time around the loop. For the decrementing version, the carry/Negative flag (set as a side effect of the subtraction, may automatically indicate i >=0. It saves the test time around the loop.

In reality, on modern pipelined processor hardware, this stuff is almost certainly trivial as there isn't a simple 1-1 mapping from instructions to clock cycles.

(b) Avoid the % operation

This is generally followed because A%B is same as A-B*(A/B).

Alternatively, % operation is equivalent to 3

operations. This is bound to to take a long time to execute. So, it should be avoided if code re-structuring is possible.
As ARM has native support of the %. operator. For other, relatively smaller processors, like AVR or MSP430, the lack of native support, makes it imperative to restructure code, where possible and avoid modulo operators.
Example: For a specific case it was evident the difference in $cycles_{total}$ was significant.

with % operation	without % operation
AVR: 29,825 cycles	AVR: 18720 cycles
ARM: 390 cycles	ARM: 384 cycles.

So, for smaller processors it is significant but for ARM it isn't.

(c) Saving memory is crucial for embedded system, in order to increase performance. So, whenever you know the variable value is going to be in range 0-255, is always more beneficial to use unsigned char over regular 32 bit int declaration.

If the program requirement for a variable is only to be 8 bits wide it is beneficial to declare it as a signed (-128 to 127) or unsigned (0 to 255) char.

The computer reads a char as an 8 bit signed integer, while we may read it is an ASCII character. Given that it saves memory, presumeably, this optimization must improve performance for all targets, hence making it a significant optimization.

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) A simple control flow is used in mission critical flight software, instead of recursive code is because it is easily verifiable. Not using recursive code guarantees there will be an acyclic function call graph. This rule is enforced likely because of the unpredictability of recursive methods. The limited availability of memory in interstellar probes also adds to the list of reasons.

(b) While reading real time data from sensors and other sources on the mission dynamic memory is not used, hence dynamic memory is disallowed after task initialization in mission critical flight software. Queues are used in most of the operations, This is also because even a small leak due to the use of dynamic memory can cause the system to crash.

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 = 0 , Z = 0 , C = 0 , V = 0
when we add r1 + r2 = 0xffffffff + 0x00000001 = 0x00000000 with carry = 1. As, the instruction add has no 's' at its end, it won't be executed conditionally so the flags won't be set.

(b) N = 0 , Z = 1 , C = 1 , V = 0
cmn instruction compares the registers as r1 + r2

r1+r2 = 0x00000000, carry = 1. The zero flag is set as addition result is 0, Z = 1. As the addition result is positive, 31 bit = 0, N = 0. As comparison result generates a carry = 1, C = 1. Because no overflow, V = 0.

(c) N = 0, Z = 1, C = 1, V = 0
r1+r2 = 0x00000000, carry = 1. As the result of addition is 0, Z = 1. As the addition result is positive, N = 0. As carry is generated, C = 1. As, the addition doesn't generate an overflow, V = 0

(d) N = 0, Z = 1, C = 1, V = 0
r1+r2 = 0x00000000, carry = 1. As the result of addition is 0, Z = 1. As the result of the addition is positive, N = 0. As this addition generates a carry, C = 1 No overflow, V = 0

(e) N = 1, Z = 0, C = 0, V = 1
r1+r2 = 0x7fffffff + 0x7fffffff = 0xfffffffe.
As the addition result is non zero, Z = 0.
As the addition result is negative, N = 1.
As, there is no carry, C = 0 when the addition numbers generate a negative number, 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:

a = 54 b = 24

iteration 1:

a != b : True, a > b : True

a = a − b = 54 − 24 = 30

a = 30 b = 24

iteration 2:

a != b : True, a > b : True

a = a − b = 30 − 24 = 6

a = 6 b = 24

iteration 3:

a != b : True, a > b : False

b = b − a = 24 − 6 = 18

a = 6 b = 18

iteration 4:

a != b : True, a > b : False

b = b − a = 18 − 6 = 12

a = 6 , b = 12

Iteration 5

a != b : True a > b : False

b = b - a = 12 - 6 = 6

a = 6 b = 6

Iteration 6

a != b : False , returns a = 6

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

r1 = 54 = 0x36 r2 = 24 = 0x18

Iteration 1:

CMP r1, r2 : r1 - r2 = 0x36 - 0x18 = 0x1E

Z = 0 N = 0 C = 0 V = 0

BEQ end : Z = 0, no branch

BLT lessthan : Z = 0 V = 0, checks N != V : False

no branch

SUB r1, r1, r2 : r1 = 0x36 - 0x18 = 0x1E (30d)

B gcd : Branch to gcd

Iteration 2:

CMP r1, r2 : r1 - r2 = 0x1E - 0x18 = 0x6

Z = 0 N = 0 C = 0 V = 0

BEQ end : Z = 0 no branch

BLT lessthan : Z = 0 V = 0 N != V : False, no branch

SUB r2, r1, r2 : r1 = r1 - r2 = 0x1E - 0x18 = 0x6 (6d)

B gcd : Branch to gcd

Iteration 3

CMP → r1 - r2 = 0x6 - 0x18

= 00000110 - 00011000 → taking 2's complement : 11101000

=

```
  00000110
+ 11101000
----------
  11101110
```

Z = 0, as this generates a -ve no., N = 1 , C = 0, V = 0

BEQ end : Z = 0, no branch

BLT lessthan : V = 0, N = 1 N != V : True, branches to less than

SUB r2, r2, r1 : r2 = r2 - r1 = 0x18 − 0x6
r2 = 0x12 (18_d)

B gcd : branch to gcd

Iteration 4:

CMP r1, r2 : r1 - r2 = 0x6 − 0x12
0000 0110
1110 1110
1110 1000
↳ taking 2's complement: 1110 1110

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

BEQ END : Z=0, no branch
BLT lessthan : N=1 V=0 N!=V : True branches to lessthan
SUB r2, r2, r1 : r2 = r2 - r1 = 0x12 - 0x6 = 0xC (12_d)
B gcd : branch to gcd

Iteration 5:

CMP r1, r2 : r1 - r2 = 0x6 − 0xC
↳ taking 2's complement: 1111 0100
0000 0110
1111 0100
1111 1010
Z=0 C=0 N=1 V=0

BEQ end : Z=0, no branch
BLT lessthan : N=1 V=0 N!=V : True, branch to less than
SUB r2, r2, r1 : r2 = r2 - r1 = 0xC - 0x6 = 0x6 (6_d)
B gcd branch to gcd

Iteration 6: CMP r1, r2 : r1 - r2 = 0x6 - 0x6 = 0x0 Z=1
BEQ end : Z=1, branch to end ; Program ends

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

r1 : 0x36 r2 : 0x18

Iteration 1

CMP r1, r2 : r1 > r2 : r1 - r2 = 0x36 - 0x18 = 0x1E
Z=0, N=0, C=0, V=0
SUBGT r1, r1, r2 : N=0, V=0; Z=0 and N=V : True
r1 = r1 - r2 = 0x36 - 0x18 = 0x1E (30_d)
SUBLT r2, r2, r1 : N=0, V=0 : N!=V : False
BNE gcd : Z!=0 : False : branch to gcd

Iteration 2

CMP r1, r2 : r1 > r2 : r1 - r2 = 0x1E - 0x18 = 0x6
Z=0, N=0, C=0, V=0
SUBGT r2, r2, r2 : N=0, V=0 ; Z=0 and N=V : True
r2 = r2 - r2 = 0x1E - 0x18 = 0x6 (6_d)
SUBLT r2, r2, r1 : N=0, V=0 ; N!=V : False
BNE gcd : Z!=0 : False : branch to gcd

Iteration 3

CMP r1, r2 : r2 > r1 : r1 - r2 = 0x6 − 0x18
↳ taking 2's complement: 1110 1000
0000 0110
1110 1000
1110 1110
Z=0, N=1, C=0, V=0

SUBGT r1, r1, r2 : checks Z=0 and N=V
here Z=0 but N=V : False
SUBLT r2, r2, r1 : N=1, V=0 : N!=V → True
r2 = r2 - r1 = 0x18 - 0x6 = 0x12 (18d)
BNE gcd : Branch to gcd

Iteration 4
CMP r1, r2 : r1 < r2 : r1 - r2 = 0x6 - 0x12 → taking 2's complement
00000110
11101110
1111 0100
Z=0, N=1, C=0, V=0

SUBGT r1, r1, r2 : checks Z=0 and N=V
here Z=0 but N=V : False
SUBLT r2, r2, r1 : N=1, V=0 : N!=V → True
r2 = r2 - r1 = 0x12 - 0x6 = 0xC (12d)
BNE gcd : Branch to gcd

Iteration 5
CMP r1, r2 : r1 < r2 : r1 - r2 = 0x6 - 0xC → taking 2's complement
00000110
1111 0100
1111 1010
Z=0, N=1, C=0, V=0

SUBGT r1, r1, r2 : checks Z=0 and N=V
here Z=0 but N=V : False
SUBLT r2, r2, r1 : N=1, V=0 : N!=V → True
r2 = r2 - r1 = 0xC - 0x6 = 0x6 (6d)
BNE gcd : branch to gcd

Iteration 6
CMP r1, r2 : r1 = r2 : Z=1
SUBGT r1, r1, r2 : Z=0 : False
SUBLT r2, r2, r1 : Z=0 : False
BNE gcd : Z=1 : doesn't branch : Programmendy

(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:

Iteration	1st ARM Routine	2nd ARM Routine
1	5+P	5+P
2	5+P	5+P
3	5+2P	5+P
4	5+2P	5+P
5	5+2P	5+P
6	2+P	4
	27+9P	29+5P

P: The number of cycles required for a pipeline refill. This ranges from 1 to 3 depending upon the alignment and width of the target instruction and whether the processor manages to speculate the address early.

CMP → 1
BEQ / BLT / B → if branch → 1+P
if not → 1
SUB → 1
SUBGT → 2
SUBLT → 2