Representing and Manipulating Numbers – Exercises

Alexandre David

1 Data Representation

1 Code into binary and hexadecimal the following decimal numbers: -127, 131, 11, -16, 14. Choose the minimal amount of bits for the coding (in power of 2). Decode into decimal the following integers on 8 bits interpreted as signed and unsigned numbers: 0x7a, 0xb1, 0x16.

2 Write a program to test the "bus error" on Sun presented during the lecture. Check that the program works on Intel. Reminder: to generate this error you should try to access data with a wrong alignment with respect to its size.

3 Write a program to test endianness on Sun and Intel architectures. You can write particular values with an "int" type and then read with "char" type to see which bytes were written in which order.

4 Write a program to experiment with signed and unsigned shifts. Exhibit the particular behaviour of the shift on signed integers.

5 Assume we are running code on a 32-bit machine using 2-complement arithmetic for signed values. Right shifts are performed arithmetically for signed values and logically for unsigned values. The variables are declared and initialized as follows:

```
int x = foo(); /* some value */
int y = bar(); /* some value */
unsigned ux = x;
unsigned uy = y;
```

For each of the following C expressions, either (1) argue that it is true (evaluates to 1) for all values of `x` and `y` or (2) give values of `x` and `y` for which it is false (evaluates to 0):

A. `(x >=0) || ((2*x) < 0)`

B. `(x & 7) != 7 || ((x << 30) < 0)`

C. `x*x >= 0`

D. `x < 0 || -x <= 0`

E. `x > 0 || -x >= 0`

F. `x*y == ux*uy`

G. `( x)*y + uy*ux == -y`

2 Arithmetic Optimization

1 The following code:

```
int arith(int x, int y)
{
  return x*M + y/N;
}
```

is compiled for a particular choice for M and N to the following (translation of machine code back to C to make it readable):

```
int arith(int x, int y)
{
  int t = x;
  x <<= 4;
  x -= t;
  if (y < 0) y += 3;
  y >>= 2;
  return x+y;
}
```

What are the values of M and N?

2 Implement a hack to take advantage of the free lower bits of pointers. What is the main issue with this hack? Hint: 32/64-bit architecture. How to solve it?

3 What does the following function do?

```
unsigned int foo(unsigned int x)
{
    x -= (x & 0xaaaaaaaa) >> 1;
    x = ((x >> 2) & 0x33333333) + (x & 0x33333333);
    x = ((x >> 4) + x) & 0x0f0f0f0f;
    x = ((x >> 8) + x);
    x = ((x >> 16) + x) & 0xff;
    return x;
}
```

4 Assume that an integer on 32 bits has the following value: 0x00RRGGBB where RR, GG, and BB are values for respectively red, green, and blue. This integer codes the color of a pixel. We want to make it lighter (double the values of red, green, and blue) and darker (half the value of red, green, and blue). As this is a pixel, the values saturate, which means any value (for RR, GG, and BB) that goes beyond 255 saturates at 255. Implement *without if statements* these two operations by using logical and arithmetic operations.

5 Similarly to (4), implement the addition of two pixels.

3 Floats

Assume variables `x`, `f`, and `d` are of type `int`, `float`, and `double`, respectively. Their values are arbitrary, except that neither `f` nor `d` equals $+\text{inf}$, $-\text{inf}$ or *NaN*. For each of the following C expressions, either argue that it wil always be true (i.e., evaluate to 1) or give a value for the variables such that it is not true (i.e., evaluates to 0).

A. `x == (int)(float) x`

B. `x == (int)(double) x`

C. `f == (float)(double) f`

D. `d == (float) d`

E. `f == -(-f)`

F. `2/3 == 2/3.0`

G. `(d >= 0.0) || ((d*2) < 0.0)`

H. `(d+f)-d == f`