Nate Flasher

Integer Representation Homework

2.72)

a) The maxbytes variable is always going to be larger than the val variable and therefore the if conditional will always result in the value of 1 (true). Therefore, the conditional test will always succeed. Also the return value of the sizeof is size\_t which is an unsigned type which could become problematic.

b) To make the test work properly the code should be rewritten as follows:

if (maxbytes => sizeof(val))

memcpy(buf, (void \*) &val, sizeof(val));

2.77)

a) x << 4 + x

b) -(x << 3) + x

c) (x << 6) – (x\*4)

d) (-1) \* ((x << 7) – (x << 4))

2.81)

a) ~((1 << k)-1)

b) ~(-1 << k) << j

2.82)

a) If x =2 and y =-4, then the left side of the expression will yield a 0 (false) because 2 is not less than -4. And the right side of the expression will yield a 1 (true) because -2 is not greater than 4.

b) This expression will always result in true. This is because x << 4 is the same as multiplying x by 16, and y << 4 is also the same as multiplying y by 16. So the expression can be simplified to x\*16 + x + y\*16 – y which will always be equivalent to 17\*y+15\*x.

c) This expression will always result to be true because both sides of the expression are essentially doing the exact same thing. The left side of the expression is negating x and y individually and then adding 1. The left side is another way of writing the expression where you first add x and y and then negate that. There is no difference between the 2 expressions besides the way they are written out.

d) This expression will always result in true. The left side of the expression is always taking an unsigned number and subtracting another unsigned number. The right side of the expression is actually doing the exact same thing. First it is subtracting two signed numbers and then casting that to an unsigned number. At this point it still is not equivalent to the other side of the expression until the value is negated.

e) This expression will always result in true. When shifting bit representations of any x to the right, then to the left (dividing by 2m then multiplying by 2), x will never be more than what it was to begin with because we are always shifting to the right first and then to the left. This means that the 1’s in the bit representation of x may go down or to the right or even get gotten rid of, but when they are shifted back to the left there will never be any 1’s in a position higher than what they were in before, however, x could be less than what it was because the 1’s could have been gotten rid of when shifting to the right.