Homework #1

Tanner J. Evans

CS341

August 27, 2020

2.59 ....

Write a C expression that will yield a word consisting of the least significant byte of x and the remaining bytes of y. For operands x=0x89ABCDEF and y=0x76543210, this would give 0x765432EF.

```
int result = (x & 0xFF) + (y & ~0xFF);
```

This expression uses a hex bit-mask with a bitwise AND to extract specific, complementary bits from x and y which we can then add together. It uses a bitwise NOT to obtain the bits of y in order to account for different int size possibilities.

2.61 ....

Write C expressions that evaluate to 1 when the following conditions are `true` and to 0 when they are false. Assume `x` is of type `int`.

(a) Any bit of `x` equals 1.

```
int nonZero = !!x;
```

This is my favorite method for normalizing non-zero numbers. We can leverage the function of the ! (NOT) operator, which converts any nonzero value to zero, and any zero value to one. Hence, a double application yields a 1 if the original was non-zero or non-null. This avoids an if statement and unecessary conditional control flow.

(b) Any bit of `x` equals 0.

```
int anyZero = !!~x;
```

Here we can apply the bitwise NOT first to reverse the 1s and 0s, and then test for the existence of any 1. (Please forgive LaTeX's nearly insurmountable and absurd difficulties with the tilde.)

(c) Any bit in the least significant byte of `x` equals 1.

```
int lsbNonZero = !!(x & 0xFF);
```

Here we must simply isolate the final byte for comparison with a bitwise AND bit-mask.

(d) Any bit in the most significant byte of `x` equals 0.

```
int msbAnyZero = !!((~x) & ((~0)<<((sizeof(int)-1)*8)));
```

Here we must get a little more creative. In order for our expression to evaluate properly for a wide variety of int sizes, we must find a way to scale our mask to that size. To do that, I applied a bitwise NOT to 0, which will yield an integer of all ones. Then we must left-shift this number (since right-shifting a negative number is problematic). We can call the sizeof() method to obtain the size of int, subtract one, and multiply by eight, to obtain the distance we must left shift to mask for only one remaining byte in the most significant position.

2.68 ....

Write code for a function with the following prototype:

```
/*
 * Mask with least signficant n bits set to 1
 * Examples:  n = 6 -> 0x3F, n = 17 -> 0x1FFFF
 * Assume 1 <= n <= w
 */
int lower_one_mask(int n);
```

Your function should follow the bit-level integer coding rules (page 164). Be careful of the case $n = w$.

```
int lower_one_mask(int n)
{
  if (n == (sizeof(int)*8)) return ~0;
  else return ~(~0 << n);
}
```

We have only the special case of n = w to account for, so we return the proper value if that is the case. In all others, we can left-shift our inverted 0 n times and then invert the bits to get a mask for the lower n bits.

2.81 ....

Write C expressions to generate the bit patterns that follow, where a^k represents k repetitions of symbol a. Assume a w-bit data type. Your code may contain references to parameters j and k, representing the values of j and k, but not a parameter representing w.

(a) $1^{w-k}0^k$

```
int upperOneMask = ~0 << k;
```

This amounts to an upper one bit mask. We can simply invert a zero bitwise to all ones and left-shift k times.

(b) $0^{w-k-j}1^k0^j$

```
int middleMask = (~0 << j) & ~(~0 << (j+k));
```

This amounts to an middle one bit mask. First, we can invert a zero bitwise to all ones and left-shift j times. We can then construct a bitmask that selects everything below j+k bits and bitwise AND this bit mask with the obtained number.

2.82 ....

We are running programs where values of type int are 32 bits. They are represented in two's complement, and they are right shifted arithmetically. Values of type unsigned are also 32 bits. We generate arbitrary values x and y, and convert them to unsigned values as follows:

```
/* Create some arbitrary values */
int x = random();
int y = random();
/* Convert to unsigned */
unsigned ux = (unsigned) x;
unsigned uy = (unsigned) y;
```

For each of the following C expressions, you are to indicate whether or not the expression always yields 1. If it always yields 1, describe the underlying mathematical principles. Otherwise, give an example of arguments that make it yield 0.

(a) $(x < y) == (-x > -y)$

This expression always yields 1. The $<$ and $>$ symbols are comparators, and will take the operands as input and return a 1 if true, and a 0 if false. Similarly, $==$ returns a 1 if the operands are equal and a 0 if not. Given this, if x is less than y, then x lies between y and infinity of the opposite sign of y. We know that negating a number mirrors it over 0, so in negating both, we move x and y such that x now lies between y and infinity of the opposite sign again. Since y's sign has changed, this means that -x is larger than -y. The argument holds for if x is not less than y.

(b) $((x + y) << 4) + y - x == 17 * y + 15 * x$
Left-shifting is tantamount to multiplying a number by 2 for each bit moved. We know that we can distrubute multiplication, so we can distrubute 2^4 to x and y, yielding 16*x and 16*y + y - x. We have like terms, so we can add and subtract to obtain 15*x + 17*y, which is the same as the right side of the expression. This expression therefore always evaluates to true.

(c) $\sim x + \sim y + 1 == \sim (x + y)$
This expression does not always evaluate to true. If either x or y approach the maximum that their data type can hold, inverting them individually will yield individual small numbers, mainly zeros. Adding them and then adding one will still obtain a small number. However, adding them together first will overflow the container. A simple example is x = 1110 and y = 1011. $\sim 1110 = 0001$, $\sim 1011 = 0100$, $0001 + 0100 = 0101$, $0101 + 1 = 0110$. Given these same values: $1110 + 1011 = 1101$, $\sim 1101 = 0010 \neq 0110$.

(d) $(ux - uy) == -(unsigned)(y - x)$
This expression does not always evaluate to 1. Take the example of $x = -4$, $y = 4$. In that case, the left hand becomes: $4 - 4 = 0$, while the right hand becomes $-(unsigned)(4 - (-4)) = -(unsigned)8 = -8$, which is clearly not equal to 0.

(e) $((x >> 2) << 2) <= x$
We must look no further than x = 1 to show that this does not always evaluate to 1. $x = 0001$, $0001 >> 2 = 0000$, $0000 << 2 = 0000 \neq x$.