CMSC 311 Answers-Draft

Quiz 1
Spring, 2009
Dr. Michelle Carter McElvany Hugue

Problem 1: Reason It Out (15pts)

So, grab your calculator program and plug this stuff in and convert. I'm not taking time to give you detailed explanations of stuff that you all learned in 250, or even some math class in a galaxy far, far away. Oh, and please reread the directions for this problem. I stated in class repeatedly, and in the instructions, that powers of two or powers of 16 were acceptable answers. Some of you can't do anything without a calculator, and I can't allow calculators because if you are smart enough to graduate with a CS degree from UMCP, you are smart enough to figure out how to program information into all but the most basic calculator–and I'm not supplying the class with such things.

So, I provide you with an alternative–just show me your answer in an intermediate format–still technically base 10, but in a way that also makes it clear how well you understand the concepts we are actually testing, *not* whether or not you can do arithmetic by hand.

This policy will be in effect throughout the course; so, don't waste time doing work that will earn you no points–and is guaranteed to lose you points because you need the *time* that you are wasting to answer other questions on the exams.

1. The 20-bit unsigned binary number **0xF0194** is

$$4 \ + \ 9 * 16 \ + \ 1 * 16^2 \ + \ 15 * 16^4$$

2. The 20-bit sign magnitude binary number **0xF0194** is

$$-(4 \ + \ 9 * 16 \ + \ 1 * 16^2 \ + \ 7 * 16^4)$$

3. The 20-bit 2's complement number **0xF0194**

$$-(1 + (11 \ + \ 6 * 16 \ + \ 14 * 16^2 \ + \ 15 * 16^3)) = -(12 \ + \ 6 * 16 \ + \ 14 * 16^2 \ \ 15 * 16^3)$$

4. The 20-bit sign magnitude number `(1 << 19) | (15)` .

$$(0x80000 \mid 0x0000F) \ = \ -15_{10}$$

5. The 20-bit 2's complement number `( ~(1 << 10) >> 6 )`.

$$(\tilde{(}0x00400) \ >> \ 6) \ = \ (0xFFBFF \ >> \ 6) \ = \ (0xFFFEF) \ = \ -(1 + 0x00010) \ = \ -(0x00011) \ = \ -17_{10}$$

Problem 2: Some Utter: " Numbers!!!" (12pts)

1. A computer with 15G (15 gigabytes) of byte-addressable memory needs a minumum of 33 bits to label each byte with a unique address.

   False: A machine with 8 gigbytes of byte addressable memory needs 33 bits, since that's $2^3 * 2^{30} \ = \ 2^{33}$ bytes. A machine with more than 8G and at most 16G of byte addressable memory will require a minimum of 34 bits since $2^4 * 2^{30} \ = \ 2^{34}$. That is, you still need one more bit, once you have more than 8G, but not more than 16G of byte addressable memory.

2. The 16 bit two's complement and 16 bit sign magnitude representations for integers do not span the same range of integers.

   True: The most negative SM number, SM_{min} is $-(2^{15}-1)$. The most negative 2's complement number, T_{min} is $-(2^{15})$.

3. The C expression `((a < 0) ? 1 : 0)` is equivalent to `!(a >> 31)`, where `a` is declared to be an `int` on a machine with 32-bit long integers.

   False: The C expression is equivalent to returning the sign bit of `a`, since it returns 1 if `a` is negative, and 0 otherwise. Hopefully, you've learned by now that `(a >> 31)` gives you a word containing the sign bit of `a` in all 32 bits. Thus, `!(a >> 31)` is 0 if `a` is negative, and 1 if `a` is positive or zero.

   Oh, and a counter example is always acceptable, and preferred when the problem contains the word "Prove". So, evaluating the ternary expression for 0 returns 0, and the second expression returns 1, which differes from 0.

4. The C expression `( p + ( -(q < p) & (q - p) ) )` returns the minimum of `q` and `p`.

   True: In order to see this, we will make a few intermediate variables.

```
int pbig    = -(q < p);  \* all ones if   (q < p) is true , else 0 *\
int diff_qp = (q-p);      \* sign = 1 if p < q, else sign is zero *\
int bias = diff_qp & pbig; \* bias = 0  if p < q,  else  (q-p)*\
return (p+bias)
```

   Yeah, that was a lot of help. Not. So, here is how this creature works on a 32 bit machine.

Case	pbig	bias	result
q < p	0xFFFF FFFF	(q-p)	q
q = p	0	0	p
q>p	0	0	p

   Note that it is possible that the quantity `(q - p)` might overflow and trigger an *exception* while evaluating the original impression that would interrupt the machine. This does not mean that the computation is in error. In fact, the computation workes even when `(q-p)` overflows. So, to avoid the issue of exceptions, casting `p` and `q` as unsigned will prevent the "two's complement overflow" from interrupting otherwise correct computations.

Problem 3: Basically Annoying–Just Answer (12pts)

1. Write a C code fragment similar to those in Lab 1, satisfying the following specification.

```
/*
 * isNotEqual - return 0 if x == y, and 1 otherwise
 *   Examples: isNotEqual(5,6) = 1, isNotEqual(5,5) = 0
 *   Legal ops: ! ~ & ^ | + << >>
 *   Max ops: 6
 *   Rating: 2
 */
int isNotEqual(int x, int y) {
  return !!(x^y);
}
```

   Why does that work? Because (`x != y`) if and only if `(x^y)` is a not string of zeros. But, `1 == !!(x^y)`. unless `(x^y)` is a string of zeros. Any equivalent expression should have received full credit.

2. What function does the following code fragment implement? Justify your answer for full credit.

```
int whoami(int x) }

    int s = x >> 31;
    return ( (x ^ s) - s);

}
```

Answer: As discussed previously, `s` will consist of 32 bits of the sign bit of `x`, either all 1's when `a` is negative or zero otherwise. If `x` is not negative, we return `x` since `s` is zero.

What happens when `x` is negative? Well, when `x` is negative, `s` represents the 2's complement version of -1. The expression `(x ^ -1) - (-1)` is equivalent to `~x + 1`, which is the 2's complement of `x` or `-x`. Thus, we return `x` if `x>0` and `-x` otherwise, which is the definition of the absolute value function.

3. Briefly explain the `result` of the C code fragment below. That is, what is the output value of the function? Be as specific as possible. Hint: Try working with a word size smaller than 32 bits first.

```
int mystery(int x)
{
  return (x-0x01010101) & ~x & 0x80808080;
}
```

Answer: This function returns a non-zero result if there is a byte in `x` that is all zeros, and returns zero otherwise. This assumes that the byte is aligned as for byte-addressable memory, as opposed to eight consecutive zero bits beginning at an arbitrary bit position. It can be used to accelerate zero-terminated string functions for byte-aligned character strings.

How does this work? First `(x-0x01010101) ^ x` indicates whether there was a change of sign in bit 7 of each byte of the word after subtraction. Anding with `~x` masks out the `80-1=0x7f` and `00-1=xff` case. Anding with 80 gives us each bit we're after.

But with De Morgan, bit twiddling can be reduced somewhat.

Assume v = x-0x01010101, then:

```
(v ^ x) & ~x =>
 (v & ~x | ~v & x) & ~x =>
   (v & ~x & ~x | ~v & x & ~x) =>
     (v & ~x | 0) =>
        (v & ~x)
```

If this returns a non-zero result, then there's a correctly aligned zero byte in the word.