Rodger Lee

UID: 204914719

CS33: Professor Reinman

Homework 1

**2.71** ◆  
You just started working for a company that is implementing a set of procedures to operate on a data structure where 4 signed bytes are packed into a 32-bit unsigned. Bytes within the word are numbered from 0 (least significant) to 3 (most significant). You have been assigned the task of implementing a function for a machine using two’s-complement arithmetic and arithmetic right shifts with the following prototype:

/\* Declaration of data type where 4 bytes are packed

into an unsigned \*/

typedef unsigned packed\_t;

/\* Extract byte from word. Return as signed integer \*/

int xbyte(packed\_t word, int bytenum);

That is, the function will extract the designated byte and sign extend it to be a 32-bit int.

Your predecessor (who was fired for incompetence) wrote the following code:

/\* Failed attempt at xbyte \*/  
int xbyte(packed\_t word, int bytenum) {

return (word >> (bytenum << 3)) & 0xFF;

}

1. What is wrong with this code?

The function can’t return a negative value because the bitwise & with 0xFF makes the most significant bit 0, so the function never returns a negative value regardless of the input.

1. Give a correct implementation of the function that uses only left and right shifts, along with one subtraction.

I want to left shift and right shift while keeping the 1 or 0 in the most significant bit. First declare int size = sizeof(unsigned) which will give us a number that is 1 + the most significant number , in this case 1 + (3).

int xbyte(packed\_t word, int bytenum) {

int size = sizeof(unsigned); // = 4 , in our case

int shift\_left\_val = (size – 1 – bytenum) << 3; //

int shift\_right\_val = (size – 1) << 3;

return (int) word << shift\_left\_val >> shift\_right\_val;

}

**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.

1. (x<y) == (-x>-y)
   * Does not always yield 1
   * false when x = Tmin.
2. ((x+y)<<4) + y-x == 17\*y+15\*x
   * Always yields 1
   * ((x+y)<<4) + y-x => x<<4 -x + y<<4 + y
   * => x\*16 -x + y\*16 + y
   * => x\*15 + y\*17
3. ~x+~y+1==~(x+y)
   * Always yields 1
   * ~x+~y+1 => ~x+ 1 + ~y+1 -1
   * => -x + -y -1
   * => - (x+y) – 1
   * => ~(x+y) + 1 – 1 => ~(x+y)
4. (ux-uy) == -(unsigned)(y-x)
   * Always yields 1
   * -(ux-uy) ==(unsigned)(y-x)
   * =>
   * (ux-uy) == (unsigned)(x-y) -> yields positive result.
5. ((x>>2)<<2)<=x
   * Always yields 1,
   * This can be proved with an example
     1. 1010 >> 2 = 1110
     2. 1110 << 2 = 1000 < 1010
   * We must pay attention to the case where the two least significant bits are both unequal to 0. Like the example I gave, 1010, the two least significant bits contains at least 1 nonzero bit. This example will prove ((x>>2) << 2) = x to be false, because the immediate left shift following the right shift will not recover the nonzero bit that was originally there in the two least significant bits.