Report of Assignment 2

20131218 Park Kee Hun

I wrote description of how my implementation work as annotation at side of each function, so I just wrote more explanation if annotation is lack or incorrect.

1. bitXor function

```
int bitXor(int x, int y) {
        //TODO
        int Xor_1, Xor_0, result;

        Xor_1 = ~(x&y);                 // exclude case of x=1, y=1.
        Xor_0 = ~(~x&~y);               // exclude case of x=0, y=0.
        result = (Xor_1&Xor_0);         // return 1 if only x!=y.
        return result;
}
```

2. tmax function

```
int tmax(void) {
        //TODO
        int overf, result;

        overf = 1<<31;          // -2^31 is stored at overf due to overflow.
        result = ~overf;        // 100...0(31 zero) changed into 111...1(31 one)
        return result;
}
```

In this function, overf = −2^31 which is 100...000, and result = 2^31−1 which is 0111...111

3. isNotEqual function

```
int isNotEqual(int x, int y) {
        //TODO
        int same, result;

        same = x^y;             // if x=y, return 0 by XOR.
        result = !!same;        // for output 1 when x is not equal to y, use !! operation.
        return result;
}
```

4. replaceByte function

```
int replaceByte(int x, int n, int c) {
        //TODO
        int shift_byte, replace_space, x_r_s, insert_value, result;

        shift_byte = (n<<3); // n is byte, which is 8bits, so we should n<<3, not just n.
        replace_space = (0xFF << shift_byte); // 0xFF represents 8bits will be replaced.
        x_r_s = x & ~replace_space;     // replaced space of x is filled with 00000000.
        insert_value = (c<<shift_byte); // 8bits will be moved to same position with 00000000 of x.

        result = x_r_s | insert_value;  // just merge both value with OR operation.
                                        // In result, 8bits will be inserted into 00000000 of x.
        return result;
}
```

5. fitsBits function

```
int fitsBits(int x, int n) {
        //TODO
        int check_sign, check_sign2, bits_need, bits_n, result;

        check_sign = (x>>31)&1;                 // if x>=0, 0; if x<0, 1;
        check_sign2 = ~((check_sign)+(~0));     // if x>=0, ~(~0)=0; if x<0, ~0; which is form for XOR operation.
        bits_need = x^check_sign2;              // bits_need means that required bits to express the x.
        bits_n = n+(~0);        // n+(~0) = n-1, because in frong of result there is ! operation, so we use n-1, not n.

        result = !(bits_need >> bits_n);        // compare length of bits_need with bits_n.
        return result;
}
```

6. rotateLeft function

```
int rotateLeft(int x, int n) {
        //TODO
        int bits_n, bits_live, LtoR1, del_sign, LtoR2, RtoL, result;

        bits_n = ~n+1;          // because ~n = -(n+1), ~n+1 = -n.
        bits_live = 32+bits_n;  // This bits will remain after shifting.
        LtoR1 = (x>>bits_live); // Applying shift to x, if x>=0, then 0000..00 n bits of x from left.
        del_sign = ~((1<<31)>>(31+bits_n));     // del_sign is for case of x<0
                                                // del_sign will be ~(1000..00 n+1 bits of 1<<31 from left.) = 0111..11011..
        LtoR2 = LtoR1 & del_sign;       // LtoR2 is consider the case both of x>=0 and x<0.
        RtoL = x<<n;    // RtoL is shifted x up to n from right to left, then n bits of x from right 000..00.
                        // In RtoL, we don't have to consider the case of x<0, because shifting is from right to left.
        result = LtoR2|RtoL; // Then just merge LtoR2 and RtoL by OR operation, they fill each other at zero section.
        return result;
}
```

7. isPower2 function

```
int isPower2(int x) {
        //TODO
        int check_sign, check_sign2, count_one, check_ispower2, result;

        check_sign = x>>31;             // if x>=0, 0; if x<0, -1;
        check_sign2 = (~0)^check_sign;  // if x>=0, ~0; if x<0, 0;
        count_one = x + check_sign2;    // if x<0, x; if x>=0, x+(~0)=x-1.
        check_ispower2 = x & count_one; // When x>=0, if x has only one of 1, x&(x+(~0)) shoule be zero.
                                        //            if x has more one of 1, x&(x+(~0)) is not zero.
        result = !(check_ispower2 + !x);        // !x means that if x=0, 1; else 0.
                                                // That is, it is only for case of x=0.
        return result;
}
```

8. rempwr2 function

```
int rempwr2(int x, int n) {
        //TODO
        int powOf2, restOf, restOf_x, isThereRest, neg, neg_c, result;

        powOf2 = 1<<n;          // which is 2^n, ex)000..010000
        restOf = (powOf2)+(~0); // which makes all bits in right side of 2^n into 1, that is rest(%)
        restOf_x = x&restOf; // which means that all bits in right side of most significant 1 bit are 1 in x.
                                // Above code is not consider the case of x<0.
        isThereRest = !!restOf_x;       // if there is rest, 1; if there is no rest, 0;
        neg = ~(isThereRest << n) + 1;  // make all bits in left side of 2^n into 1, containing 2^n
                                        // which is 111...11000
        neg_c = (x>>31);                // neg_c let us use neg when x<0.
                                        // If x>=0, neg_c must be zero, so (neg&neg_c) also must be zero.
        result = restOf_x + (neg&neg_c);        // By using + operation, we can get output both case of x>=0 and x<0.
        return result;
}
```

9. conditional function

```
int conditional(int x, int y, int z) {
        //TODO
        int check_xzero, output_z, output_y, result;

        check_xzero = (!x+(~0));

        output_z = (~check_xzero)&z;
        output_y = (check_xzero)&y;

        result = output_z|output_y;

        return result;
}
```

In this function, check_xzero is that if(x=0) 0, else -1.
output_z outputs z if x is zero.
output_y outputs y if x is not zero.
output z and y have opposite condition, so we should use OR operation in result.

10. bitParity function

```
int bitParity(int x) {
        //TODO
        int result;

        x = x^(x>>16);  // Compare left side 16bits with right side 16 bits.
        x = x^(x>>8);   // Then we don't think about left side 16bits of x.
        x = x^(x>>4);   // We just compare each other with dividing x by 2 section.
        x = x^(x>>2);   // When comparing, 1 bits in same position disappear,
        x = x^(x>>1);   // which can't affect whether the number of 0's in origin x is odd or not.
                        // because they always disappear with a pair.

        result = x&1; // So, we just check the last 1 bit. If it is odd, odd number of 0's is also odd.
        return result;
}
```

11. greatestBitPos function

```
int greatestBitPos(int x) {
        //TODO
        int case_normal, case_negative, result;

        x |= x>>16;     // This 5 lines makes all bits in right side of most significant 1 bit of x into 1.
        x |= x>>8;      // Because it keeps dividing 2 section upto 1-1bits,
        x |= x>>4;      // and it uses OR operation.
        x |= x>>2;      //
        x |= x>>1;      // That is, 000001(most significant bit)111..111.

        case_normal = x&(~x>>1);        // By shifting ~x to right in 1 bit, we get only one of 1 that is most significant bit.
                                        // which considers only the case of x>=0.
        case_negative = x&(~!x<<31);    // This is considering case of x<0.
        result = case_normal^case_negative; // For get only most significant bit, we should use XOR operation.
        return result;
}
```

12. logicalNeg function

```
int logicalNeg(int x) {
        //TODO
        int left, right, result, result_modify;

        left = x>>31;   // if x>=0, 0; if x<0, -1;
        right = (~x+1)>>31; // because ~x = -(x+1), ~x+1 = -x.
                            // if x<=0, 0; if x>0, -1;
        result = left|right;    // just merge the range of left and right.
                                // if x=0, 0; if x>0 or x<0, -1;
        result_modify = result + 1;     // Adjust output into if x=0, 1; else 0;
        return result_modify;
}
```

13. bitAnd Function

```
int bitAnd(int x, int y) {
        //TODO
        int result, result_modify;

        result = ~x|~y;
        result_modify = ~result;        // for output And operation, we should ~result.
        return result_modify;
}
```

14. logical_OR Function

```
int logical_OR(int x, int y) {
        //TODO
        int oper_or, result;

        oper_or = x|y;          // bit OR operation.
        result = !!oper_or;     // convert bit operation to logical OR operation.
        return result;
}
```

15. concatenate Function

```
int concatenate(int x, int y) {
        //TODO
        int shift_x, result;

        shift_x = x<<8;         // shift x to right by 8bts
        result = shift_x|y;     // merge shift_x and y with OR operation
        return result;
}
```

16. isMult4 function

```
int isMult4(int x) {
        //TODO
        int restOf, restOfx, isThereRest, result;

        restOf = (1<<2)+(~0);           // for make 000..00100
        restOfx = x&restOf;             // restOfn means n%4
        isThereRest = restOfx&3;        // if there is a rest, rest; else, 0;
        result = !isThereRest;          // if there is a rest, 0; else, 1;
        return result;
}
```