**Data Lab**

**Source Code:**

(The analysis of the solution to each problem is included in the code comments)

//1

/\*

 \* bitXor - x^y using only ~ and &

 \*   Example: bitXor(4, 5) = 1

 \*   Legal ops: ~ &

 \*   Max ops: 14

 \*   Rating: 1

 \*/

int bitXor(int x, int y) {

  return ~(x & y) & ~(~x & ~y); // DeMorgan's law

}

/\*

 \* tmin - return minimum two's complement integer

 \*   Legal ops: ! ~ & ^ | + << >>

 \*   Max ops: 4

 \*   Rating: 1

 \*/

int tmin(void) {

  return 1 << 31; // set the sign bit to 1

}

//2

/\*

 \* isTmax - returns 1 if x is the maximum, two's complement number,

 \*     and 0 otherwise

 \*   Legal ops: ! ~ & ^ | +

 \*   Max ops: 10

 \*   Rating: 1

 \*/

int isTmax(int x) {

  // If x equals TMAX, ~(x + 1) equals 0x0FFFFFFF(TMAX), thus (~(x + 1)) ^ x equals 0

  // If x not equals TMAX, we will get the oppsite result

  // Nevertheless, if x equals -1, ~(x + 1) ^ x also equals 0, so -1 is a special case that needs extra judgement

  return !((~(x + 1)) ^ x) & !!(x + 1);

}

/\*

 \* allOddBits - return 1 if all odd-numbered bits in word set to 1

 \*   where bits are numbered from 0 (least significant) to 31 (most significant)

 \*   Examples allOddBits(0xFFFFFFFD) = 0, allOddBits(0xAAAAAAAA) = 1

 \*   Legal ops: ! ~ & ^ | + << >>

 \*   Max ops: 12

 \*   Rating: 2

 \*/

int allOddBits(int x) {

  // If all odd bits of x is 1, thus x & 0xAAAAAAAA == 0xAAAAAAAA

  // Owing to the limitations to declare a constant, we need to use a special tip to generate 0xAAAAAAAA

  int a;

  int b;

  int c;

  a = 0xAA << 8; // get 0xAA00

  b = a + 0xAA; // get 0xAAAA

  c = (b << 16) + b; // get 0xAAAAAAAA

  return !((x & c) ^ c);

}

/\*

 \* negate - return -x

 \*   Example: negate(1) = -1.

 \*   Legal ops: ! ~ & ^ | + << >>

 \*   Max ops: 5

 \*   Rating: 2

 \*/

int negate(int x) {

  return (~x + 1);

}

//3

/\*

 \* isAsciiDigit - return 1 if 0x30 <= x <= 0x39 (ASCII codes for characters '0' to '9')

 \*   Example: isAsciiDigit(0x35) = 1.

 \*            isAsciiDigit(0x3a) = 0.

 \*            isAsciiDigit(0x05) = 0.

 \*   Legal ops: ! ~ & ^ | + << >>

 \*   Max ops: 15

 \*   Rating: 3

 \*/

int isAsciiDigit(int x) {

  // '0' - 0011 0000B

  // '1' - 0011 0001B

  // ...

  // '9' - 0011 1001B

  // We can see from above that if x is ASCII digit, it should start with 0011B

  // Furthermore, the maximum of the last 4 bits is 1001B if x is ASCII digit

  // Hence, we should judge whether the value of last 4 bits is greater than 1001

  int sign1 = !(x >> 4 ^ 0x3); // here to judge whether x is a number starting with 0011B

  int least\_4\_bits = x & 0xF; // get the last 4 bits of x for further judgement

  int judge = ~0xA + 1 + least\_4\_bits; // ~0xA + 1 == 0xFFFFFFF6(1111...1111 0110), if least\_4\_bits > 0110B, sign2 == 0x0000000X(the last bit is unknown)

  int sign2 = !!(judge & 0x10); // if least\_4\_bits > 0110B, then the fifth bit from bottom of the judge won't equal 1

  return sign1 & sign2;

}

/\*

 \* conditional - same as x ? y : z

 \*   Example: conditional(2,4,5) = 4

 \*   Legal ops: ! ~ & ^ | + << >>

 \*   Max ops: 16

 \*   Rating: 3

 \*/

int conditional(int x, int y, int z) {

  // To implement x ? y : z using only legal ops, we can consider a form like this: (y + y1) + (z + z1)

  // If x == 1, y1 should be 0 and z1 should be -z so that we can get y

  // else if x == 0, y1 should be -y and z1 should be 0 so that we can get z

  int condition = !!x; // get 1 if x != 0, else get 0

  int sign = ~condition + 1; // if condition equals 1,then sign equals 0xFFFFFFFF, else if condition equals 0, then sign equals 0x00000000

  int first\_result = ~(y & ~sign) + 1; // get 0 or -y according the condition

  int second\_result = ~(z & sign) + 1; // get 0 or -z according the condition

  return y + z + first\_result + second\_result;

}

/\*

 \* isLessOrEqual - if x <= y  then return 1, else return 0

 \*   Example: isLessOrEqual(4,5) = 1.

 \*   Legal ops: ! ~ & ^ | + << >>

 \*   Max ops: 24

 \*   Rating: 3

 \*/

int isLessOrEqual(int x, int y) {

  // To implement x <= y, we consider using x - y <= 0

  // Here are all the cases of x - y:

  // case1: x > 0, y > 0 (x - y <= 0 ?)

  // case2: x > 0, y < 0 (having risk of overflowing) -> x > y

  // case3: x < 0, y > 0 (having risk of overflowing) -> x < y

  // case4: x < 0, y < 0 (x - y <= 0 ?)

  // Conclusion:

  // If it is case3, then it must be x < y;

  // else if it isn't case 2 and x - y < 0, then it must be x < y

  // else if x = 0 and y = 0, then it must be x = y

  // else it must be x > y

  int x\_minus\_y = x + (~y + 1); // get x - y

  int signbit\_x\_minus\_y = x\_minus\_y >> 31 & 0x1; // get the sign bit of x - y

  int signbit\_x = x >> 31 & 0x1; // get the sign bit of x

  int signbit\_y = y >> 31 & 0x1; // get the sign bit of y

  int is\_case2 = (!signbit\_x) & signbit\_y;

  int is\_case3 = signbit\_x & (!signbit\_y);

  return is\_case3 | ((!is\_case2) & signbit\_x\_minus\_y) | (!(x ^ y));

}

//4

/\*

 \* logicalNeg - implement the ! operator, using all of

 \*              the legal operators except !

 \*   Examples: logicalNeg(3) = 0, logicalNeg(0) = 1

 \*   Legal ops: ~ & ^ | + << >>

 \*   Max ops: 12

 \*   Rating: 4

 \*/

int logicalNeg(int x) {

  // Any x that geater than 0 subjects to (~x + 1) >> 31 == -1

  // x < 0 is a special case that needs extra judgement:

  // Considering x < 0 makes (~x + 1) >> 31 == 0, we can use x | (~x + 1)) >> 31

  return ((x | (~x + 1)) >> 31) + 1;

}

/\* howManyBits - return the minimum number of bits required to represent x in

 \*             two's complement

 \*  Examples: howManyBits(12) = 5

 \*            howManyBits(298) = 10

 \*            howManyBits(-5) = 4

 \*            howManyBits(0)  = 1

 \*            howManyBits(-1) = 1

 \*            howManyBits(0x80000000) = 32

 \*  Legal ops: ! ~ & ^ | + << >>

 \*  Max ops: 90

 \*  Rating: 4

 \*/

int howManyBits(int x) {

  // minimum number of bits = sign bit + the most significant bit

  int sign, b16, b8, b4, b2, b1;

  sign = x >> 31;

  x = (~sign & x) | (sign & ~x); // if x < 0 then x = ~x (Test the positive and negative numbers together)

  b16 = !!(x >> 16) << 4;

  x >>= b16;

  b8 = !!(x >> 8) << 3;

  x >>= b8;

  b4 = !!(x >> 4) << 2;

  x >>= b4;

  b2 = !!(x >> 2) << 1;

  x >>= b2;

  b1 = !!(x >> 1);

  x >>= b1;

  return b16 + b8 + b4 + b2 + b1 + x + 1;

}

//float

// 31 30        23 22                      0

// s     exp              frac

/\*

 \* floatScale2 - Return bit-level equivalent of expression 2\*f for

 \*   floating point argument f.

 \*   Both the argument and result are passed as unsigned int's, but

 \*   they are to be interpreted as the bit-level representation of

 \*   single-precision floating point values.

 \*   When argument is NaN, return argument

 \*   Legal ops: Any integer/unsigned operations incl. ||, &&. also if, while

 \*   Max ops: 30

 \*   Rating: 4

 \*/

unsigned floatScale2(unsigned uf) {

  unsigned signbit = (uf >> 31) & 0x1;

  unsigned exp\_mask = 0x7F800000; // 0111 1111 1000 -> 0 11111111

  unsigned exp = (uf & exp\_mask) >> 23;

  unsigned frac\_mask = 0x007FFFFF;

  unsigned frac = (uf & frac\_mask);

  unsigned res = 0;

  if (exp == 0xFF) { // NaN

    return uf;

  }

  if (exp == 0x00) { // Denorm

    frac <<= 1;

    res = (signbit << 31) | (exp << 23) | frac;

  }

  else {

    exp += 1;

    res = (signbit << 31) | (exp << 23) | frac;

  }

  return res;

}

/\*

 \* floatFloat2Int - Return bit-level equivalent of expression (int) f

 \*   for floating point argument f.

 \*   Argument is passed as unsigned int, but

 \*   it is to be interpreted as the bit-level representation of a

 \*   single-precision floating point value.

 \*   Anything out of range (including NaN and infinity) should return

 \*   0x80000000u.

 \*   Legal ops: Any integer/unsigned operations incl. ||, &&. also if, while

 \*   Max ops: 30

 \*   Rating: 4

 \*/

int floatFloat2Int(unsigned uf) {

  unsigned S = (uf >> 31) & 0x1;

  unsigned exp\_mask = 0x7F800000;

  unsigned exp = (uf & exp\_mask) >> 23;

  unsigned frac\_mask = 0x007FFFFF;

  unsigned frac = (uf & frac\_mask);

  unsigned result = 0;

  int E = exp - 127;

  if (E >= 31) { // out of bound

    return 0x80000000u;

  }

  if (E < 0) { // rounding

    return 0;

  }

  result = frac | (1 << 23); // add the implicit zero

  if (E < 23) { // rounding (The mantissa of a single-precision floating point number is 23 bits, then some bits should be dicarded if E < 23 when converting to integer)

    result >>= (23 - E);

  }

  else {

    result <<= (E - 23);

  }

  if (S) {

    return -result;

  }

  return result;

}

/\*

 \* floatPower2 - Return bit-level equivalent of the expression 2.0^x

 \*   (2.0 raised to the power x) for any 32-bit integer x.

 \*

 \*   The unsigned value that is returned should have the identical bit

 \*   representation as the single-precision floating-point number 2.0^x.

 \*   If the result is too small to be represented as a denorm, return

 \*   0. If too large, return +INF.

 \*

 \*   Legal ops: Any integer/unsigned operations incl. ||, &&. Also if, while

 \*   Max ops: 30

 \*   Rating: 4

 \*/

unsigned floatPower2(int x) {

    // max normal value = 2 ^ 127 \* (2 - 2 ^ -23)

    // min denorm value = 2 ^ (-23 - 126) = 2 ^ -149

    if (x > 127) {

      return 0xFF << 23;

    }

    if (x < -149) {

      return 0;

    }

    if (x >= -126) { // norm

      int exp = x + 127;  // bias = 127;

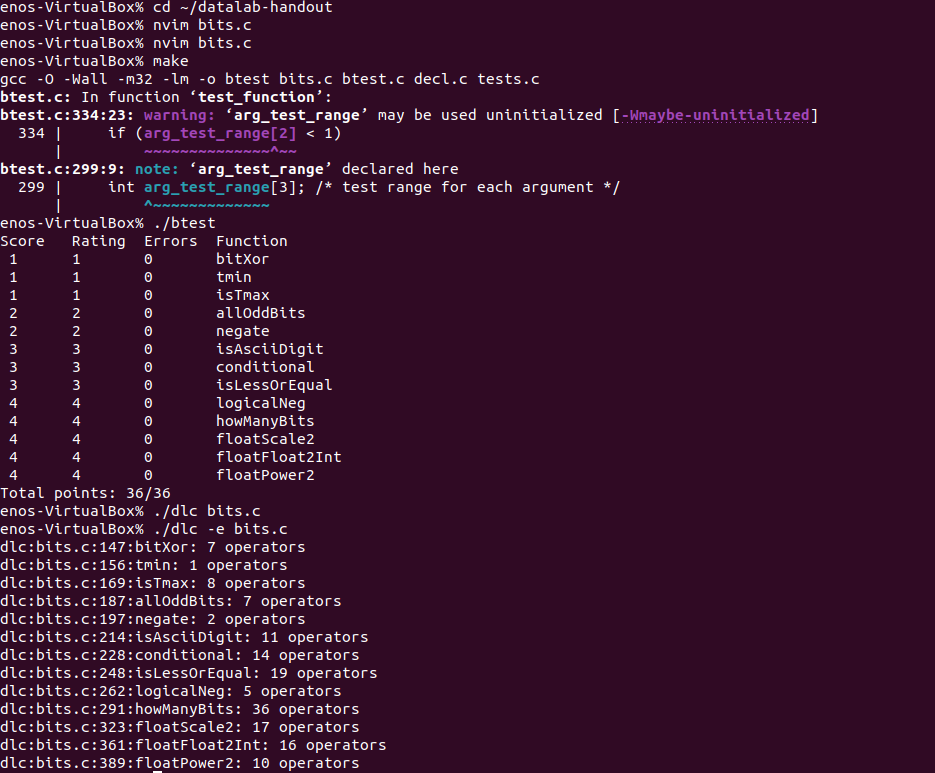
      return (exp << 23);

    }

    return 1 << (148 + x); // denorm

}

**Result:**

****