lab1datalab/datalab/bits.c

```
/*
 * CS:APP Data Lab
 *
 * 王俊崴 22302016002
 *
 * bits.c - Source file with your solutions to the Lab.
 *          This is the file you will hand in to your instructor.
 *
 * WARNING: Do not include the <stdio.h> header; it confuses the dlc
 * compiler. You can still use printf for debugging without including
 * <stdio.h>, although you might get a compiler warning. In general,
 * it's not good practice to ignore compiler warnings, but in this
 * case it's OK.
 */

#if 0
/*
 * Instructions to Students:
 *
 * STEP 1: Read the following instructions carefully.
 */

You will provide your solution to the Data Lab by
editing the collection of functions in this source file.

INTEGER CODING RULES:

  Replace the "return" statement in each function with one
  or more lines of C code that implements the function. Your code
  must conform to the following style:

  int Funct(arg1, arg2, ...) {
      /* brief description of how your implementation works */
      int var1 = Expr1;
      ...
      int varM = ExprM;

      varJ = ExprJ;
      ...
      varN = ExprN;
      return ExprR;
  }

  Each "Expr" is an expression using ONLY the following:
  1. Integer constants 0 through 255 (0xFF), inclusive. You are
      not allowed to use big constants such as 0xffffffff.
  2. Function arguments and local variables (no global variables).
  3. Unary integer operations ! ~
  4. Binary integer operations & ^ | + << >>

  Some of the problems restrict the set of allowed operators even further.
  Each "Expr" may consist of multiple operators. You are not restricted to
```

```
  one operator per line.

  You are expressly forbidden to:
  1. Use any control constructs such as if, do, while, for, switch, etc.
  2. Define or use any macros.
  3. Define any additional functions in this file.
  4. Call any functions.
  5. Use any other operations, such as &&, ||, -, or ?:
  6. Use any form of casting.
  7. Use any data type other than int.  This implies that you
     cannot use arrays, structs, or unions.


  You may assume that your machine:
  1. Uses 2s complement, 32-bit representations of integers.
  2. Performs right shifts arithmetically.
  3. Has unpredictable behavior when shifting if the shift amount
     is less than 0 or greater than 31.


EXAMPLES OF ACCEPTABLE CODING STYLE:
  /*
   * pow2plus1 - returns 2^x + 1, where 0 <= x <= 31
   */
  int pow2plus1(int x) {
     /* exploit ability of shifts to compute powers of 2 */
     return (1 << x) + 1;
  }

  /*
   * pow2plus4 - returns 2^x + 4, where 0 <= x <= 31
   */
  int pow2plus4(int x) {
     /* exploit ability of shifts to compute powers of 2 */
     int result = (1 << x);
     result += 4;
     return result;
  }

FLOATING POINT CODING RULES

For the problems that require you to implement floating-point operations,
the coding rules are less strict.  You are allowed to use looping and
conditional control.  You are allowed to use both ints and unsigneds.
You can use arbitrary integer and unsigned constants. You can use any arithmetic,
logical, or comparison operations on int or unsigned data.

You are expressly forbidden to:
  1. Define or use any macros.
  2. Define any additional functions in this file.
  3. Call any functions.
  4. Use any form of casting.
  5. Use any data type other than int or unsigned.  This means that you
     cannot use arrays, structs, or unions.
  6. Use any floating point data types, operations, or constants.
```

```
NOTES:
  1. Use the dlc (data lab checker) compiler (described in the handout) to
     check the legality of your solutions.
  2. Each function has a maximum number of operations (integer, logical,
     or comparison) that you are allowed to use for your implementation
     of the function.  The max operator count is checked by dlc.
     Note that assignment ('=') is not counted; you may use as many of
     these as you want without penalty.
  3. Use the btest test harness to check your functions for correctness.
  4. Use the BDD checker to formally verify your functions
  5. The maximum number of ops for each function is given in the
     header comment for each function. If there are any inconsistencies
     between the maximum ops in the writeup and in this file, consider
     this file the authoritative source.

/*
 * STEP 2: Modify the following functions according the coding rules.
 *
 *   IMPORTANT. TO AVOID GRADING SURPRISES:
 *   1. Use the dlc compiler to check that your solutions conform
 *      to the coding rules.
 *   2. Use the BDD checker to formally verify that your solutions produce
 *      the correct answers.
 */


#endif
//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) {
  // compute the bitwise complement of x & y, equivalent to the bitwise XOR of x and y
(all bits flipped)
  int a = ~(x & y);

  // compute the bitwise complement of x & y, which is the bitwise OR of x and y (all
bits flipped)
  int b = ~(~x & ~y);

  // the AND of these two returns the XOR of x and y
  return a & b;
}
/*
 * tmin - return minimum two's complement integer
 *   Legal ops: ! ~ & ^ | + << >>
 *   Max ops: 4
 *   Rating: 1
 */
int tmin(void) {
  // -1 is 1111...1111, all the bits after the MSB are flipped,
  // so the largest negative number would be 1000...0000
```

```
    // hence, we shift 1 to the left by 31 bits

    return (0x01 << 31);
  }
  //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 is the the maximum int, then adding 1 to it would make it the smallest int
    // 0111 .... 1111 + 1 => 1000 .... 0000

    int x1 = x + 1;
    // XOR x1 with x would give all 1s because their bits must be all different
    int XOR = x^x1;

    // flip all the bits in XOR, if x is the max then it would be all 0s, and the logical !
  would turn it into true
    /// we also need to make sure x1 is not -1
    return !(~XOR | (!x1));

    // 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) {
    // return !((x^0xAA)&((x >> 8) ^ 0xAA)&((x >> 16) ^ 0xAA)&((x >> 24) ^ 0xAA));
    // printf("%d\n", (x^0xAA));
    // return (!(x^0xAA))|(!((x >> 8)^0xAA))|(!((x >> 16)^0xAA))|(!((x >> 24)^0xAA));
    // return ~((x & 0xAA) ^ 0xAA) | ~(((x >> 8) & 0xAA) ^ 0xAA) | ~(((x >> 16) & 0xAA) ^
  0xAA) | ~(((x >> 24) & 0xAA) ^ 0xAA);

    // rightmost bit is 0th position
    // 0xAA => 0000 ... 1010 1010
    // the hex value 0xAA has the pattern of all odd-numbered bits set to 1
    // so all we have to do is to bitwise AND the value with x
    // and then shift x to the right by the size of 0xAA, which is 8 bits

    int p1 = (x & 0xAA);
    int p2 = ((x >> 8) & 0xAA);
    int p3 = ((x >> 16) & 0xAA);
    int p4 = ((x >> 24) & 0xAA);

    // p1 to p4 should all be equal to 0xAA, or 1010 1010, if x has all number bits

    // bitwise AND between p1 to p4 would still yield 0xAA if they are all equal
    // use bitwise NOT with 0xAA to compare (similar to ==)
```

```c
  // because this funciton needs use to return true when it is all odd bits, we simply do a logical NOT on the result
  return !((p1 & p2 & p3 & p4) ^ 0xAA);
}
/*
 * negate - return -x
 *   Example: negate(1) = -1.
 *   Legal ops: ! ~ & ^ | + << >>
 *   Max ops: 5
 *   Rating: 2
 */
int negate(int x) {
  // very simple as long as we know that ~x = -x - 1
  // so rearranging the equation we get -x = ~x + 1

  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) {
  int lower_bound = 0x30;
  int upper_bound = 0x3a;

  int diff1 = x + (~lower_bound + 1);  // x - lower_bound
  int diff2 = upper_bound + (~x);  // upper_bound - x

  int sign1 = diff1 >> 31 & 1;  // 1 if diff1 < 0, 0 otherwise
  int sign2 = diff2 >> 31 & 1;  // 1 if diff2 < 0, 0 otherwise

  return !(sign1 | sign2);  // return 1 if neither sign1 nor sign2 is 1, 0 otherwise
}
/*
 * 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) {
  // create a mask that depends on the value of x
  // if x is not 0, create a all 1 mask
  // else if x is 0, create a all 0 mask

  // !x => 1 if x = 0
  // if x = 0, !x - 1 = 0
  // if x != 0, !x - 1 = -1 (all 1 binary)
  // ~0x00 produces an all 1 binary array (aka -1 in base10)
  int mask = ((!x) + ~0x00);
```

```
  return ((mask) & y) | ((~mask) & z);
}
/*
 * 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) {
  // There are two conditions in which x <= y holds true:
  // 1. When the SIGN is the same and x - y < 0 or y - x > 0
  // 2. When the SIGN differs, we just compare the signs

  // first retrieve the signs of x and y
  int sign_x = x >> 31;
  int sign_y = y >> 31;

  // using XOR to compare yields true when they differ
  // and false otherwise, add a logical NOT to flip the bit
  int same_sign = (!(sign_x ^ sign_y));

  // given than ~x = -x - 1
  // x - y < 0 => (x + ~y + 1)
  //      condition 1                         condition 2
  return (same_sign & ((x + (~y)) >> 31)) | ((!same_sign) & sign_x);
}
//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) {
  // ~x = -x - 1
  // -x = ~x + 1
  int negativeX = ~x + 1;

  // bitwise OR x with -x produces 1111 ... 1111 if x != 0
  // 0000 ... 0000 if x == 0

  // then shift the bitwise OR result by 31 to extract the sign bit
  // 0 if x != 0, -1 if x == 0
  // adding one to those would give 1 if x != 0, and 0 if x == 0 which
  // gives up the effect of logical neg without using !

  return ((x | negativeX) >> 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) {
  // keep in mind that this asks for the minimum number of bits required
  // to represent a number in binary, not the number of bits set (equal to 1)

  // the concept behind this is simple, we first get the sign from x
  // because the MSB is always the sign

  // if the number is negative then we add one to the final result
  // otherwise we can disregard the MSB because it would be 0

  // 1) get the sign from the number
  // 2) flip all the bits if the number is negative (turn it into a
  // positive number
  // 3) starting from the middle of the binary array, check if the left side is equal
  // to 0 using !!(x >> 16)
  // 3.1) if the left side is equal to zero, that means there's no data there, so we
don't need
  // to shift x (when x = 0, !!(x >> 16) is 0, therefore 0 << 4 is still 0)
  // 3.2) if the left side is not zero, then that means all the bits to the right
contains USEFUL
  // information, hence we set pos16 to 16 and then shift x by 16 bits to the left
  // Rinse and repeat with 8 bit, 4 bit, 2 bit, and one bit shifts

  // This is like doing a binary serach for the position of the last set bit after the
MSB

  int sign = x >> 31;
  // printf("%d\n", sign);
  int pos16, pos8, pos4, pos2, pos1, pos0;
  x = x ^ sign; // flip each bit of x if x < 0

  pos16 = !!(x >> 16) << 4; // pos16 = 16 if (x >> 16) != 0
  x >>= pos16;

  pos8 = !!(x >> 8) << 3; // pos8 = 8 if (x >> 8) != 0
  x >>= pos8;

  pos4 = !!(x >> 4) << 2; // pos4 = 4 if (x >> 4) != 0
  x >>= pos4;

  pos2 = !!(x >> 2) << 1; // pos2 = 2 if (x >> 2) != 0
  x >>= pos2;

  pos1 = !!(x >> 1); // pos1 = 1 if (x >> 1) != 0
  x >>= pos1;

  pos0 = x;

  // because we disregarded the MSB earlier, add a 1 because the MSB is always required
to
```

```
    // differentiate between positive and negative numbers
    // we always add a one because the MSB is required regardless of the sign
    return pos16 + pos8 + pos4 + pos2 + pos1 + pos0 + 1;

    // int sign = x >> 31;
    // int not_x = ~x;
    // int mask = sign | not_x;
    // int num_bits = 0;

    // int i = 0;
    // i += 2;

    // num_bits = (mask >> 16) & 16;
    // num_bits |= ((mask >> (num_bits + 8)) & 8);
    // num_bits |= ((mask >> (num_bits + 4)) & 4);
    // num_bits |= ((mask >> (num_bits + 2)) & 2);
    // num_bits |= ((mask >> (num_bits + 1)) & 1);

    // return num_bits + 1;
  }
  //float
  /*
   * 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) {
    /*
    1 10101010 00000000000000000000
    1 10101010 10100000000000000000

    11111111111111111111111111111111
    */

    // The point of this function is bascially to "scale" the input, unsigned integer uf,
  by a factor of 2
    // The return value is the bit-level representation of the input scaled by a factor of
  2

    // Scaling a float by a factor of 2 is very straightforward
    // If the input is a normalized float, we simply add one to the exponent
    // When the input is denormalized, we can scale it by shifting the mantissa one bit to
  the left (effectivly multiplying it by 2)

    // structure of single precision floating point
    // Sign | Exponent | Mantissa
    // 1    | 8        | 23
    // Bias = 127

    // extract the sign and exp from uf
    unsigned sign = (uf >> 31);
```

```
  unsigned exp = (~(0x1 << 31) & uf) >> 23;

  // 0xF = 1111
  // since the mantissa is 23 bits long, we need 5 0xF and a 0x7 (0111)
  // Mantissa mask => 0x7FFFFF;
  unsigned frac = (uf & 0x7FFFFF);

  // if uf is NaN or inf, then return uf
  if(exp == 0xFF) return uf;

  if(exp == 0) {
    frac <<= 1;
  } else {
    exp += 1;
  }

  return (sign << 31) | (exp << 23) | frac;

  // return 2;
}
/*
 * 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) {

  // extract the sign and exp from uf
  unsigned sign = (uf >> 31);
  unsigned exp = (~(0x1 << 31) & uf) >> 23;

  // 0xF = 1111
  // since the mantissa is 23 bits long, we need 5 0xF and a 0x7 (0111)
  // Mantissa mask => 0x7FFFFF;
  unsigned frac = (uf & 0x7FFFFF);

  int E = exp - 127;

  // if the exponent is less that one, we are working with a denomalized
  // float (fractional value), this automatically rounds to 0 so we
  // just return 0 because we are converting to int (losing information)
  if (E < 0)  return 0;

  // inf & NaN base case
  if (exp == 0xFF || E >= 31)  return 0x80000000u;
  else {
    // When converting a 32-bit float to an integer using bitwise operations, we need to
extract the integer value of the float by multiplying the mantissa by 2 raised to the
power of the exponent. However, the mantissa only contains 23 bits, so if the exponent is
```

```
larger than 23, we need to shift the mantissa left by the difference between the exponent
and 23 to get the full integer value.

    // In the IEEE 754 binary representation of the float, the leading bit of the
mantissa field is always assumed to be 1. Therefore, we can add this bit back to the
mantissa by OR-ing it with the value 1 shifted to the left by 23 bits, which sets the
24th bit of the mantissa to 1.
    frac = frac | (1 << 23);
    if (E <= 23)  frac >>= (23 - E);
    else  frac <<= (E - 23);
  }

  // return the correct value according to the sign bit
  // because the mantissa does not
  if (sign) return -frac;
  else  return frac;
}
/*
 * 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) {
  // (1) first check if the result is too large to be represented as a normalized float
  // (2) then check if the reuslt is too small to be represented as a denormal float
  // (3) if the result it neither too small, nor too large, then we calculate and return
the appropriate bit-level equivalent o fthe expression 2.0^x

  // (1)
  // largest normalized < 2 * 2 ^ 127 = 2 ^ 128
  if (x > 127)  return 0xFF << 23;

  // (2)
  // smallest denormalized = 2 ^ (1 - 127) + 2 ^ (-23) = 2 ^ (-149)
  else if (x < -149)  return 0;

  // (3)
  // check whether the number should be represented in as a normalized float or
denormalized float

  // acutal exponent e = E - 127 (for single precision floats)
  // E = e + 127

  // normalized float is simple, we just convert x to E and shift it by 23 bits into the
correct position

  // dormalized float is a bit different, we use the mantissa to represent the value
  // 0.111 gives 1/2 1/4 1/8 ...
  // therefore the position of 1 in the mantissa behind the decimal has the effect of
2^(-x)
```

```
    // hence we first convert x to E, E = 1 - 127, therefore E = -126
    // to represent 1 at the proper position, we add 126 to offest E, then add 23 again to
  shift it to the left of mantissa

    // smallest normalized = 2 ^ (-126)
    else if (x >= -126)  return (x + 127) << 23;
    // denormalized representation for negative powers
    else return 1 << (x + 126 + 23);
  }
```