ICS Lab1-DataLab 实验报告

在终端中执行 `./dlc -e bits.c` 后的截图：

```
fan@DESKTOP-DVFFB09:/mnt/c/Users/admin/Desktop/datalab-handout$ ./dlc -e bits.c
dlc:bits.c:148:tconst: 1 operators
dlc:bits.c:160:bitNand: 3 operators
dlc:bits.c:174:ezOverflow: 3 operators
dlc:bits.c:188:fastModulo: 4 operators
dlc:bits.c:201:findDifference: 5 operators
dlc:bits.c:215:absVal: 3 operators
dlc:bits.c:231:secondLowBit: 8 operators
dlc:bits.c:254:byteSwap: 18 operators
dlc:bits.c:276:byteCheck: 19 operators
dlc:bits.c:289:fractions: 5 operators
dlc:bits.c:308:biggerOrEqual: 17 operators
dlc:bits.c:328:hdOverflow: 12 operators
dlc:bits.c:351:overflowCalc: 18 operators
dlc:bits.c:366:logicalShift: 9 operators
dlc:bits.c:390:partialFill: 20 operators
dlc:bits.c:411:float_abs: 2 operators
dlc:bits.c:463:float_cmp: 30 operators
dlc:bits.c:520:float_pow2: 30 operators
dlc:bits.c:561:float_i2f: 28 operators
dlc:bits.c:601:oddParity: 16 operators
dlc:bits.c:628:bitReverse: 34 operators
dlc:bits.c:710:mod7: 55 operators
bits.c:424: Warning: function returns both with and without a value
```

在终端中执行 `./btest` 后的截图：

```
fan@DESKTOP-DVFFB09:/mnt/c/Users/admin/Desktop/datalab-handout$ ./btest
Score   Rating  Errors  Function
 1      1       0       tconst
 2      2       0       bitNand
 2      2       0       ezOverflow
 3      3       0       fastModulo
 3      3       0       findDifference
 4      4       0       absVal
 4      4       0       secondLowBit
 5      5       0       byteSwap
 5      5       0       byteCheck
 5      5       0       fractions
 6      6       0       biggerOrEqual
 6      6       0       hdOverflow
 7      7       0       overflowCalc
 8      8       0       logicalShift
 8      8       0       partialFill
 3      3       0       float_abs
 5      5       0       float_cmp
 6      6       0       float_pow2
 7      7       0       float_i2f
 2      2       0       oddParity
 2      2       0       bitReverse
 2      2       0       mod7
Total points: 96/96
```

实验思路（代码仅仅展示函数名以及关键步骤）：

P1-
```
*tconst - return a constant value 0xFFFFFFE0
int tconst(void) {
  return ~0x1F;
}
```
由于不仅可以使用超过 255 的数，只能通过取反或者取相反数来获得一个数值较大的数字；因此使用 return ~0x1F。

P2-
```
* bitNand - return ~(x&y) using only ~ and |
int bitNand(int x, int y) {
  return (~x)|(~y);
}
```
根据摩根定理，~(x&y) == (~x)|(~y)

P3-
```
* ezOverflow - determine if the addition of two signed positive numbers overflows,
 *      and return 1 if overflowing happens and 0 otherwise
int ezOverflow(int x,int y) {
  int sum = x + y;
  return (sum >> 31)&1;//由于 c 语言右移是逻辑右移，需要删除前面所有 1，才能使溢出时 return 1
}
```
由于是正整数与正整数相加，创建一个值让 x 和 y 相加即可

P4-
```
* fastModulo - return x%(2^y)
int fastModulo(int x,int y) {
  int tool = (1<<y) + ~0;
  return (x&tool);
}
```
创建一个工具，需要求余的位上是 1，其余溢出出去；让 x 与工具做与运算，进而求余

P5-
```
* findDifference - return a mask that marks the different bits of y compared to x
int findDifference(int x,int y) {
  int mask = (x&(~y))|((~x)&y);//即返回 x 和 y 的异或值
  return mask;
}
```
即数字逻辑中使用与或非门开达到异或的目的

P6-
```
* absVal - return the absolute value of x
int absVal(int x) {
  int tool = x >> 31;//考虑正负数在做什么处理的时候有区别
  return (x+tool)^tool;//负数转正数等于取反+1，等于-1 取反；负数+（～0）即负数-1
}
```
考虑正负数在做什么处理的时候有区别，负数转正数等于取反+1，等于-1 取反；负数+（～0）即负数-1。

P7-
```
* secondLowBit - return a mask that marks the position of the second least significant 1 bit.
int secondLowBit(int x) {
  int lowest = x&(~x+1);//找出最小的有效位
  int mask = (x^lowest)&(~(x^lowest)+1);//将最小的有效位删除，然后重复操作
  return mask;
}
```
一个数同上相反数即可得到最小的有效位，接下来只需将最低删去，然后重复操作。

P8-
```
* byteSwap - swaps the nth byte and the mth byte
int byteSwap(int x, int n, int m) {
  int bytemask = 0xFF;//匹配一个 byte 的掩码
  int swap_n = (x>>(n<<3))&bytemask;//找出你要交换的 byte
  x = x ^ (swap_n<<(n<<3));//删除要交换的 byte
  x = x | (swap_m<<(n<<3));//输入交换的 byte
}
```
找到想要交换的 byte 位，储存下来；然后使用异或将其删去，再使用或运算输入交换后的数值

P9-
```
* byteCheck - Returns the number of bytes that are not equal to 0
int byteCheck(int x) {
    int byteMask = 0xFF;//匹配一个 byte 的掩码
    int byte1 = (x >> 0) & byteMask;
    int byte3 = (x >> 16) & byteMask;
    int count = !!byte1 + !!byte2 + !!byte3 + !!byte4;//使用！！把不是 0 的 byte 数值强制转化为 1
}
```
将 x 右移并与 0xff 进行与运算，即可得到不同 byte 位数值；使用两次！即可将数值强制转化为 1 或 0

P10-
```
* fractions - return floor(x*7/16), for 0 <= x <= (1 << 28), x is an integer
int fractions(int x) {
  int result = x + (x<<2) + (x<<1);//先 x*7
  result >>= 4;// result/16
  return result;
}
```
乘上 7 等于*（1+2+4），右移四位即/16

P11-
```
* biggerOrEqual - if x >= y  then return 1, else return 0
int biggerOrEqual(int x, int y) {
  int Sign = (tool>>31)&(0x1); //x -y 的符号
  int bitXor = xSign ^ ySign;//x 和 y 符号相同标志位，相同为 0 不同为 1
  bitXor = (bitXor>>31)&1;//符号相同标志位格式化为 0 或 1
  return ((!bitXor)&(!Sign))|(bitXor&(y>>31));
}
```
使用 x-y 以及 x 和 y 的符号位来判断

返回 1 有两种情况：二者符号相同并且标志位为 0（二者符号相同），x-y 的符号为 0；符号相同标志为 1（二者符号不同）位，y 的符号位为 1

```
P12- * hdOverflow - determine if the addition of two signed 32-bit integers overflows,
 *      and return 1 if overflowing happens and 0 otherwise
int hdOverflow(int x, int y) {
  int overflow_condition = ((x_sign & y_sign & !sum_sign) | (!x_sign & !y_sign & sum_sign));
  return overflow_condition;
}
```

溢出条件为当 x 和 y 都为正数时，sum 为负数，或者当 x 和 y 都为负数时，sum 为正数；通过获取三者符号位即可判断

```
P13- * overflowCalc - given binary representations of three 32-bit positive numbers and add them together,
 *      return the binary representation of the part where bits are higher than 32.
int overflowCalc(int x, int y, int z) {
    int overflow = ((x & y) | ((x | y) & ~sum))>> 31;
    int overflow2 = ((sum & z) | ((sum | z) & ~sum2))>>31;
    int result = (((overflow&1) + (overflow2&1))&3);
    return result;
}
```

进行两次 P12 操作即可，第一次 x 与 y 判断第二次（x+y）和 z 判断

```
P14- * logicalShift - shift x to the right by n, using a logical shift
int logicalShift(int x, int n) {
  int mask = ((1 << 31) >> (n + ~0)); // 创建一个掩码，用于将高于 n 位的位数清零
  mask = mask + !n;//处理是 0 的情况
  return (x >> n)&(~mask); // 右移并清除高于 n 位的位数
}
```

右移后将左边的补位删除即可

```
P15- * partialFill - given l, h, fill all even digits within the [l, h] interval with 1
int partialFill(int l, int h) {
  int minus_h = ~h + 1;
  int mask = (tool2<<16)|tool2; // 创建一个掩码，其中偶数位都为 1，奇数位都为 0
  int mask2 = (tool3 >> ((31+minus_h)+(~0)));
  int maskH = mask2 + !(31+minus_h);//排除为 0 的情况
  int rangeMask = (maskL ^ maskH); // 创建一个掩码，将 [l, h] 区间内的位都设置为 1
  return (mask & rangeMask); // 结合两个掩码来填充偶数位
}
```

创建一个掩码，其中偶数位都为 1，奇数位都为 0；再创建一个掩码，将 [l, h] 区间内的位都设置为 1；将两个掩码进行与运算即可得到结果

```
P16- * float_abs  Return bit-level equivalent of expression|f|(absolute value of f) for floating point argument
f.
```

```
unsigned float_abs(unsigned uf) {
  int abs = uf & 0x7FFFFFFF;
  if  (abs > 0x7F800000){
    return uf;
  }
  return abs;
}
```

使用 if 语句对无符号整型进行判断，取反删除符号位即可

```
P17- * float_cmp - Return 1 if uf1 > uf2, and 0 otherwise.
unsigned float_cmp(unsigned uf1, unsigned uf2) {
  if((uf1 == 0x0 && uf2 == 0x80000000) || (uf1 == 0x80000000 && uf2 == 0x0)){
    return 0;//排除正负 0
  }
// 如果 uf1 和 uf2 具有不同的符号位，则比较它们的符号位来决定大小。
  if ((uf1 ^ uf2) & 0x80000000) {
    return (uf2 >> 31);
  }
}
```

首先判断是否是 NaN，然后排除掉正负 0 的情况；之后即可进行判断；（权重）符号位大于指数大于尾数，如果完全相等返回 0.

```
P18-* float_pow2 - Return bit-level equivalent of expression f*(2^n) for
unsigned float_pow2(unsigned uf, int n) {
  while(uf <= 0x7fffff){//如果是非规整数就转化为规整数，或者返回结果
      if(((uf <<1) <=0xffffff)&&(n > 0)){
        uf = (uf << 1);
        n = n-1;
      }
      if(n ==0){
      return (sign|uf);
      }
    }
    // 计算 2^n 的位级表示
    power_of_2 = (n << 23) + exp;
  if (power_of_2>=0x7f800000){
    return (0x7f800000| sign);//如果溢出，强制转化为最大或者最小
  }
    return result;
}
```

最优先考虑的应该是正负 0，直接返回；然后考虑非规格化数，使用 while 循环，将非规格化数不断左移，直到其转化为规格化数。并且考虑特殊情况，如果指数溢出，则强制转化为无穷大或者无穷小。

P19-
```
* float_i2f - Return bit-level equivalent of expression (float) x
unsigned float_i2f(int x) {
    exp=158;//exp 的初始值
  while (!(frac&0x80000000)) {//将尾数强制左移
    exp--;
    frac<<=1;
  }
//如果第八位等于 1，同时低位还有非零值则进位；如果第九位为 1 并且后八位为 0x80 则进位
  judge=(eighth&&siven)||(eighth&&!siven&&ninth);
  frac=(frac<<1)>>9;//将原尾数左移 1 位并去掉最高位，得到舍入后的尾数
  frac+=judge;
  if (frac>=0x800000) {//检查尾数是否超过规格化浮点数范围
    frac=((frac+0x800000)>>1)-0x800000;
    exp+=1;
  }
  return (sign|(exp<<23)|frac);
}
```

首先将数字转化为绝对值并且保留符号位，如果 0 单独考虑。首先将尾数最高位移动到最左边，并且用 exp 反映移动的次数；此后记录第九位、第八位、后七位的数字，如果第八位等于 1，同时低位还有非零值则进位；如果第九位为 1 并且后八位为 0x80 则进位。如果此时尾数超出规格化浮点数范围，需要进行转换。

P20-
```
* oddParity - return the odd parity bit of x, that is,
 *      when the number of 1s in the binary representation of x is even, then the return 1, otherwise return 0.
int oddParity(int x) {
  int result;
    x =~(x^(x >> 16));//因为如果偶数个数是偶数，那么奇数个数也是偶数；所以将所有数字同或
    x =~(x^(x >> 8));//即可得到偶数个数
    x =~(x^(x >> 4));
    x =~(x^(x >> 2));
    x =~(x^(x >> 1));
    result = (x & 1);
    return result;
}
```

由 int 类型可知，如果偶数个数是奇数，那么奇数的个数也是奇数；只要将偶数与偶数相抵消，奇数与奇数相抵消即可，即将 int 对折进行同或操作即可。

P21-
```
* bitReverse - Reverse bits in an 32-bit integer
int bitReverse(int x) {
  int tool1,mask,mask1,mask2,mask3,mask4;
  tool1 = 0xFF;
  mask = (tool1<<8)|tool1;;//0x0000FFFF
```

```
 mask1 = (tool1<<16)|tool1;//0x00FF00FF;
 mask2 = (mask1 <<4)^mask1;//0x0F0F0F0F;
 mask3 = (mask2 <<2)^mask2;//0x33333333;
 mask4 = (mask3<<1)^mask3;//0x55555555
 x = ((x >> 16) & mask) | (x<< 16);
 x = ((x >> 8) & mask1) | ((x & mask1) << 8);
 x = ((x >> 4) & mask2) | ((x & mask2) << 4);
 x = ((x >> 2) & mask3) | ((x & mask3) << 2);
 x = ((x >> 1) & mask4) | ((x & mask4) << 1);
 return x;
 }
```

（所有题目写的时间最长的一道）将 16 位与 16 位、8 位与 8 位、4 位与 4 位、2 位与 2 位、1 位与 1 位进行对换即可将首位倒置。而重点在五个 mask 的获取，可以通过 mask 与 mask 之间的关系来获得不同的 mask。

```
P22- * mod7 - calculate x mod 7 without using %.
int mod7(int x) {
    int tmp1, tmp2,result,result1,mask,abs_x,tool,tool2,minus_7,judge;
    int tmp11, tmp12, tmp21, tmp22, tmp31, tmp32, tmp41, tmp42, tmp51, tmp52;
    int tmp61, tmp62, tmp71, tmp72, tmp81, tmp82, tmp91, tmp92;



    mask = (x >> 31);
    abs_x = ((x^(x>>31))+(mask&1));
    tool =!(x ^ (1<<31));//如果为 0x8000000 返回 1,其余数返回 0
    // 第一次迭代
    tmp1 = abs_x >> 3;
    tmp2 = abs_x & 0x7;
    tmp1 = tmp1 + tmp2;
    // 第十次迭代
    tmp91 = tmp81 >> 3;
    tmp92 = tmp81 & 0x7;
    tmp91 = tmp91 + tmp92;
    //最后一次循环
    result  = tmp91 >> 3;
    result1 = tmp91 & 0x7;
    result = result + result1 ;//若为 0x80000000 补上 1
    tool2 =!(result ^ (7));//如果为 7 返回 1,其余数返回 0
    result = (result + (minus_7 & ((tool2<<31)>>31)));
    judge = (((mask +result )^mask)+((0x3)&((tool<<31)>>31)));
    return judge;
}
```

首先将整数转化为绝对值；仅通过位操作符可以简单的%8 的操作，通过将%8 的余数与结果进行相加，可以得到不完全的%7 操作；之后重复十次上述操作即可得到%7 的结果。当结果为 7 时，需要将其转化为 0；而当输入为 0x8000000 时因为没有对应的整数，需要单独进行考虑。

个人感受：

大部分题目还是中规中矩，能够在二十分钟内完成；flow_pow2 和 bitReverse 这两个题目做了超乎想象的久，但是复盘的时候反而觉得这两个题目没有想象中的难。发现自己做题没有能够在最初进行思考的时候考虑到所有情况，从而导致自己不断推翻之前的办法；以后应该在进行充分思考后再开始代码的编写。在写题目的过程中，体会到了用四位或者八位进行推演的好处，能够大幅度加快自己思考的速度。

在做题的过程中体会到了及时做好备份的重要性，以及 pow2 的过程中打了无数次断点查看变量的变化过程，这些都是日后很好用的小技巧。除了对题目本身的感觉，感受到了这学期好多题目之间的联系；比如使用摩根定律、以及数字逻辑中的与非门等知识来帮助解决题目。

在很多题目编写时自己的逻辑反而并不能够达到想要的效果，就去查询了计算机是怎么对这项操作进行作用的，比如说 float_I2f 和 mod7，也就是下一章的学习内容；总之还是很有趣的，看到自己的分数不断增加，非常高兴。