整数运算

- 简介
  - 此次作业是通过用一种编码通过无符号整型实现计算机中数据的加减乘除取模的运算。
  - 我选择用补码来进行计算，因为对于补码来说，他的加法操作更加容易实现。对于一正一负的情况更容易处理。而且计算机内部本身使用的就是补码储存。
  - 程序的基本使用方式是输入两个二进制数（这里为了方便，统一将二进制数转化成了十进制储存，例如本身需要处理的数字是1111，这里输入的就是15，从而方便了后面的位运算）
  - 首先通过基本的位运算实现了加减乘除模，然后结合加减乘除设计了字符串和数字之间的互相转化。

- 算法推导
  - 加法：
    - $x > 0$，$y > 0$ 则 $x + y > 0$

      由于参加运算的数都为正数，故运算结果也一定为正数。又由于正数的补码与真值有相同的表示形式，所以根据补码定义可得：

      $[x]补 + [y]补 = x + y = [x + y]补 \pmod 2$
    - $x > 0$，$y < 0$，则 $x + y > 0$ 或 $x + y < 0$

      由于参加运算的两个数一个为正、一个为负，则相加结果有正、负两种可能。根据补码定义，有

      $[x]补 = x$，$[y]补 = 2 + y$

      所以

      $[x]补 + [y]补 = x + 2 + y = 2 + (x + y)$

      当 $x + y > 0$ 时，$2 + (x + y) > 2$，进位 2 必丢失，又因为 $x + y > 0$，所以

      $[x]补 + [y]补 = 2 + (x + y) = [x + y]补 \pmod 2$

      当 $x + y < 0$时，$2 + (x + y) < 2$，又因为 $x + y < 0$，所以

      $[x]补 + [y]补 = 2 + (x + y) = [x + y]补 \pmod 2$
    - $x < 0$，$y > 0$，则 $x + y > 0$ 或 $x + y < 0$

      这种情况和第 2 种情况一样，把 x 和 y 的位置对调即可得证。
    - $x < 0$，$y < 0$，则 $x + y < 0$

      由于参加运算的数都为负数，故运算结果也一定为负数。根据补码定义可得：

      $[x]补 = 2 + x$，　　$[y]补 = 2 + y$

      所以

      $[x]补 + [y]补 = 2 + x + 2 + y = 2 + (2 + x + y)$

      由于 $x + y < 0$，其绝对值又小于1，那么$(2 + x + y)$一定是小于 2 而大于 1 的数，所以上式等号右边的 2 必然丢掉，又因为 $x + y < 0$，所以

      $[x]补 + [y]补 = 2 + (x + y) = [x + y]补 \pmod 2$

    综上，两个数补码的和等于和的补码。
  - 减法

    只要证明$[-y]补 = -[y]补$，上式即得证。

[x+y]补 = [x]补 + [y]补 (mod 2)

[y]补 = [x + y]补 - [x]补

[x-y]补 = [x + (- y)]补 = [x]补 + [- y]补

[- y]补 = [x - y]补 - [x]补

[- y]补 + [y]补

= [x + y]补 + [x - y]补 - [x]补 - [x]补

= [x + y + x - y]补 - [x]补 - [x]补

= [x + x]补 - [x]补 - [x]补

= 0

故 [- y]补 = - [y]补(mod 2)

- 乘法

  类似于原码和整数的乘法，即很多次加法和的累加，易证明

- 除法

  乘法的逆运算，易证明

- 程序
  - 加法

    此实验最重要的部分是设计过 32bit 的全加器，主要是把 carry 依次加到 sum 上，代码如下：

    ```
    word madd( word operand1, word operand2)
    {
        word sum = operand1;
        word carry = operand2;
        word temp;
        // simulate the full-adder.
        while (carry)
        {
            temp = sum;
            sum = temp ^ carry; // sum without carry
            carry = (temp & carry) << 1; // get the carry of each bit.
        }
        return sum;
    }
    ```

    基本算法为操作数x与y先进行与位运算，得出每一位上的进位，之后操作数x与y再进行异或操作，得出没有加进位的和。最后将得到的和当作新的x，所得的进位左移一位作为新的y，重复操作，直到进位为0，此时x即为一开始x与y的和。

  - 减法

    而减法可以通过补码的规则，求其相反数进行计算。由于我是使用的补码，故只需对操作数y进行取反加一，得到其相反数，再让x与-y相加，即得出x与y的差。

```
word msub(word operand1, word operand2)
{
    // the sub of 2s' complement is the add of op1 and the opposite of
op2
    word sum;
    word opposite = madd(~operand2, 1);
    sum = madd(operand1, opposite);
}
```

- 乘法

  乘法是比较难的一个，我目前的实现方法是模拟竖式的过程进行计算，先对x与y取绝对值。从y的最右一位开始，乘以x（因为y的每位只有0和1，相当于当此位为1时加一个x，此位为0时不变），然后是y的第二位，乘以x左移一位（相当于乘2），接着是y的第三位，乘以x左移两位，一直循环直到y的各个位与移位后的x做过乘积，最后将各项乘积相加，所得和即为x与y的乘积。然后再判断x与y的符号，确定乘积的正负。但是此方法只能计算较小的数字，当数字较大（二进制大于32位）时很可能会产生溢出。

```
word mmul( word operand1, word operand2)
{
    // judge if the operands are positive,
    // if not, get the abs.
    word x = (operand1 >> 31) ? ~msub(operand1, 1) : operand1;
    word y = (operand2 >> 31) ? ~msub(operand2, 1) : operand2;
    word sum = 0;
    // simulate the vertical form.
    while(y){
        if(y & 0x1)
            sum = madd(sum, x);
        y >>= 1;
        x <<= 1;
    }
    // if exists one negative, get the opposite number.
    if( (operand1 >> 31) ^ (operand2 >> 31) )
        sum = madd(~sum, 1);
    return sum;
}
```

- 除法和求模

  除法和求模，也是先对 x 与 y 取绝对值，然后模拟列竖式的过程，依次减掉（如果 x 够减的 y^(2^31),y^(2^30),...y^8,y^4,y^2,y^1。减掉相应数量的 y 就在结果加上相应的数量。直到x小于y。 取模是通过先求出商，得出商与除数的乘积，用被除数与乘积的差值表示模。

```
word mdiv( word operand1, word operand2)
{
    // judge if the operands are positive,
    // if not, get the abs.
    word x = (operand1 >> 31) ? madd(~operand1, 1) : operand1;
    word y = (operand2 >> 31) ? madd(~operand2, 1) : operand2;
    word quo = 0;
    // simulate the vertical form.
    // sub 2^31, 2^30 ... 2^1
    for( int i = 31; i >= 0; i--)
    {
        if( ( x >> i ) >= y ) // can be divided, noted as 1.
```

```
        {
            quo += (1 << i);
            x -= (y << i);
        }
    }
    // if exists one negative, get the opposite number.
    if ((operand1 >> 31) ^ (operand2 >> 31))
        quo = madd(~quo, 1);
    return quo;
}

word mmod(word operand1, word operand2)
{
    // judge if the operands are positive,
    // if not, get the abs.
    word x = (operand1 >> 31) ? madd(~operand1, 1) : operand1;
    word y = (operand2 >> 31) ? madd(~operand2, 1) : operand2;
    word quo, mod;
    word product;
    quo = mdiv(operand1, operand2);
    product = mmul(quo, operand2);
    mod = msub(operand1, product);
    if (mod >> 31)
        mod = madd(mod, y);
    return mod;
}
```

- 字符串和数的互相转化

  字符串转换为十进制数字，是通过每位先减去'0'，再乘以相应权重（由于已用位运算表示出乘法，故在此直接使用乘法），最后将结果相加。
  十进制数字转换为字符串，是取出该十进制的每一位数字，通过取余和除10完成（除法和取余操作已经由位运算实现）。

- 结果

输出格式采用了%d和%u两种，能够方便的显示出差别。

```
Please input the decimal number of the two operands:
Please divide the two number using a space
For example: 3621 -47382
3621 -47382
The decimal number of 3621 is 3621, unsigned decimal is 3621
The decimal number of -47382 is -47382, unsigned decimal is 4294919914
Now calculate the arithmetic result of the two numbers.
Sum        of 3621 and -47382 is -43761 or 4294923535
Difference of 3621 and -47382 is 51003 or 51003
Product    of 3621 and -47382 is -171570222 or 4123397074
Quotient   of 3621 and -47382 is 0 or 0
Modulo     of 3621 and -47382 is 3621 or 3621
Please input the test operand for mtoa:
1234
The mtoa of 1234 is: 1234
```

```
Please input the decimal number of the two operands:
Please divide the two number using a space
For example: 3621 -47382
123456 -123456
The decimal number of 123456 is 123456, unsigned decimal is 123456
The decimal number of -123456 is -123456, unsigned decimal is 4294843840
Now calculate the arithmetic result of the two numbers.
Sum        of 123456 and -123456 is 0 or 0
Difference of 123456 and -123456 is 246912 or 246912
Product    of 123456 and -123456 is 1938485248 or 1938485248
Quotient   of 123456 and -123456 is -1 or 4294967295
Modulo     of 123456 and -123456 is 0 or 0
Please input the test operand for mtoa:
123456789
The mtoa of 123456789 is: 123456789
```

- 分析
  - 优缺点
    - 原码特点是简明，可以很清楚的表示一个数字。但是在计算上不方便
    - 补码的优点是能使符号位与有效值部分一起参与运算，从而简化了运算规则，同时它也使减法运算转换为加法运算，进一步简化计算机中运算器的电路，这使得在大部分计算机系统中，数据都使用补码表示。但是对于直接看出一个数字的值不是很方便。
    - 移码是在补码的基础上把首位取反得到的，这样使得移码非常适合于阶码的运算，所以移码常用于表示阶码。但是对于直接看出一个数字的值不是很方便。
  - 补码的位扩展

    比较简单，当该数是负数时,高位补1;当该数是非负数时,高位补0。
  - 补码的溢出

    补码运算的溢出判别方式为双高位判别法，利用CS表示符号位是否进位，利用CP表示最高数值位是否进位。如果CS ^ CP的结果为真，则代表发生了溢出（运算结果大于0则为负溢出，小于0则为正溢出），否则运算结果正确。也就是数字逻辑课程里全加器设计时判断的方法。
  - 大小比较

    先比较符号位，即最高位，1为负数0为正数（负数肯定小于正数），然后由高位向低位进行字典序依次比较，如果这个数是正数即符号位为0，则字符串比较（字典序）结果大的数，其值大；如果为负数，则字符串比较（字典序）结果小的数，其值反而大。