如何实现在32位平台实现64位的整数运算?

在如mips32、x86这样的32位处理器上并没有64位的寄存器，也没有对64位整数进行运算的指令，但是一些类似于java的编程语言却有64位整数的数据类型并且可以对64位整数进行计算，遇到这种情况我们只能对64位整数的运算进行模拟。

这篇文章中主要考虑mips32和x86两个平台，说明如何在这两个平台上如何实现常见的64位整数运算。

模拟的方式通常是利用两个32位寄存器分别存放64位整数的高位和低位，通过对gcc生成的代码进行观察我们很容易就可以看出编译器是怎么做的:

```
gcc -S -m32 <源码文件>
```

不过需要记得安装上 `gcc-multilib` 以支持交叉编译，同时使用 `stdint.c` 中的 `int64_t` 而不是 `long` ，因为在32位模式下 `long` 并不是64位的，而 `int64_t` 可以保证这一点。

下面的内容分别说明了如何使用两个32位寄存器对64位整数运算进行模拟，使用了c++代码，同时也说了不同体系结构上的汇编实现。

我把这些运算使用c++实现了一遍，用两个32位模拟了64位，可以查看文件int64Ops.cpp看完整代码。

下面是表示64位整数的结构体。

```
typdef uint32_t uint_32;
struct uint_64 {
    uint_32 hi, lo;
    // hi 为高32位。
    // lo 为低32位。

    // 从64位整数创建。
    static uint_64 from(uint64_t d) {
        return {(uint_32) (d >> 32), (uint_32) d};
    }

    // 转为64位整数。
    static uint64_t to(uint_64 d) {
        return ((uint64_t) d.hi << 32) + d.lo;
    }
};
```

位运算

位运算实现起来最为容易，对高低两个部分分别进行32位的位运算就行了，下面用或作为一个例子:

```
// uint_64模拟的64位或。
uint_64 orl(uint_64 a, uint_64 b) {
```

```
    return {a.hi | b.hi, a.lo | b.lo};
}
```

加减

处理加减法要稍微复杂一些，不过思路是类似的。以加法为例，高低两个部分分别进行运算，先两个低位部分相加，这可能会产生进位，如果产生了进位那么在对两个高位部分相加时需要加上这个进位：

```
// uint_64模拟的64位加法。
uint_64 addl(uint_64 a, uint_64 b) {
    uint_32 lo = a.lo + b.lo;
    uint_32 carry_flag = 0;
    // 检查是否发生了进位，通常可以通过非跳转的指令序列实现。
    if (lo < a.lo) carry_flag = 1;
    return {a.hi + b.hi + carry_flag, lo};
}
```

减法是类似的，不过减法不是进位而是借位，借位就是在对高位进行减的时候同时减去借位：

```
// uint_64模拟的64位减法。
uint_64 subl(uint_64 a, uint_64 b) {
    uint_32 lo = a.lo - b.lo;
    uint_32 borrow_flag = 0;
    // 检查是否发生了借位，通常可以通过非跳转的指令序列实现。
    if (lo > a.lo) borrow_flag = 1;
    return {a.hi - b.hi - borrow_flag, lo};
}
```

两个需要注意的地方，第一个是如何检测进位或者借位，另外一个是mips下的 `addu` / `add` 指令。

检测进位和借位，在x86下并不需要检测进位和借位，因为x86有标志寄存器而且有 `adc` （add with carry）、 `sbb` （subtract with borrow），可以在做高位部分的计算的时候直接使用这两个指令。

在mips32下则可以用 `sltu` 和 `sgtu` 分别检查是否产生了进位或者借位，下面是mips32上模拟的加法[1]：

```
addu  $t1, $t3, $t5    # add least significant word
sltu  $t0, $t1, $t5    # set carry-in bit
addu  $t0, $t0, $t2    # add in first most significant word
addu  $t0, $t0, $t4    # add in second most significant word
```

这么做的原因是如果 `a+b=c` 相加后的 `c` 小于 `a` 或 `b` 任意一个，那么肯定是发生了进位，借位的检测也是如此。

mips add/addu，注意一下要mips的加不要使用 `add` 而是应该使用 `addu` ， `add` 是带有溢出检测的，而 `addu` 不带，更具体原因在[4]中给出了，简单来说就是编译器不需要硬件支持的溢出检测，而是通过其他方式实现。

位移

位移要更加麻烦一些，不过思路也是类似的，对高低位分开来处理，给出c++模拟的逻辑左移的例子，vc++中的实现大致就是这样[2]：

```
// uint_64模拟的64位逻辑左移。
uint_64 shll(uint_64 a, uint_32 b) {
    if (b == 0) return a;
    if (b > 63) return {0, 0};
    // 大于31小于64时，低位全为0。
    if (b > 31) return {a.lo << (b - 32), 0};
    // b in (0,31]则需要一个临时变量t，将a.lo的高b位移动为低b位。
    uint_32 t = (a.lo >> (32 - b));
    return {a.hi << b | t, a.lo << b};
}
```

应该非常直观，分4种情况处理位移，如果为0，就什么都不用处理，如果，位移长度超过63，那么高低位都变为0，如果位移长度∈ $(31, 63]$，那么丢弃高位，如果位移长度∈ $(0, 31]$，那么将低位和高位同时移动，然后将低位部分多出来的部分补充到高位的末尾，当然条件可以简化到2个，不过为了方便说明还是使用了4中情况。

下面是两个情况的版本：

```
// uint_64模拟的64位逻辑左移。
uint_64 shll(uint_64 a, uint_32 b) {
    // 当b>63时，b-32>32，高低部分全部为0。
    if (b > 31) return {a.lo << (b - 32), 0};
    // 当b=0，时，t=0。
    uint_32 t = (a.lo >> (32 - b));
    return {a.hi << b | t, a.lo << b};
}
```

x86，gcc：

```
; #include<stdint.h>
; int main() {
;       int64_t w = 19;
;       int64_t a = 100000;
;       int64_t c = a << w;
;       return 0;
; }

 804915d:       c7 44 24 18 13 00 00    mov    DWORD PTR [esp+0x18],0x13
 8049164:       00
 8049165:       c7 44 24 1c 00 00 00    mov    DWORD PTR [esp+0x1c],0x0
 804916c:       00
 804916d:       c7 44 24 10 a0 86 01    mov    DWORD PTR [esp+0x10],0x186a0
 8049174:       00
 8049175:       c7 44 24 14 00 00 00    mov    DWORD PTR [esp+0x14],0x0
 804917c:       00
```

```
804917d:       8b 4c 24 18             mov    ecx,DWORD PTR [esp+0x18]
8049181:       8b 44 24 10             mov    eax,DWORD PTR [esp+0x10]
8049185:       8b 54 24 14             mov    edx,DWORD PTR [esp+0x14]
8049189:       0f a5 c2                shld   edx,eax,cl
804918c:       d3 e0                   shl    eax,cl
804918e:       f6 c1 20                test   cl,0x20
8049191:       74 04                   je     8049197 <main+0x45>
8049193:       89 c2                   mov    edx,eax
8049195:       31 c0                   xor    eax,eax
8049197:       89 c3                   mov    ebx,eax
8049199:       89 d6                   mov    esi,edx
804919b:       89 5c 24 08             mov    DWORD PTR [esp+0x8],ebx
804919f:       89 74 24 0c             mov    DWORD PTR [esp+0xc],esi
```

gcc在x86上实现的就是两个情况的版本，不过x86有 `shld` （SHLD — Double Precision Shift Left），可以实现64位的位移，不过这个位移只是将高32位部分移动CL位，然后将低32位的高CL位放到高32位部分的尾部进行填充，其中CL是用来计数的寄存器。同时，只有CL的低5位有效，也就是最多移动31位，并不能实现移动32位以上的情况，shl指令的CL也是一样的，只能最高移动31位。

为了处理这样的问题，所以有一个判断 `test cl,0x20` 查看是否大于 `32` ，如果不是那么执行 `mov edx,eax` 和 `xor eax,eax` ，会将低32位移动到高32位，然后清空为0。不过这样的做法只能保证移动长度$\in [0, 63]$是正确的，我不知道这是编译器的要求还是ANSI C的，不过只需要加上一个是否大于63的判断就可以解决这个问题了。

乘

乘法是类似的，下面给出一个答案[3]：

```
; x, y: 64-bit integer
; x_h/x_l: higher/lower 32 bits of x
; y_h/y_l: higher/lower 32 bits of y

; x*y  = ((x_h*2^32 + x_l)*(y_h*2^32 + y_l)) mod 2^64
;      = (x_h*y_h*2^64 + x_l*y_l + x_h*y_l*2^32 + x_l*y_h*2^32) mod 2^64
;      = x_l*y_l + (x_h*y_l + x_l*y_h)*2^32

; Now from the equation you can see that only 3(not 4) multiplication needed.
```

由于有一个 `mod 2^64` ，所以 `x_h*y_h*2^64` 变为0，实际上也就是左移了64位消失了，同时 `(x_h*y_l + x_l*y_h)*2^32` 有一个左移32位，丢弃这一项的高32位。

对应的c++代码为：

```
// uint_64模拟的64位乘法。
uint_64 mull(uint_64 a, uint_64 b) {
    // mips和x86都会将结果保存在两个32位寄存器，下面模拟了这一点。
    auto alxbl = uint_64::from(((int64_t) a.lo * (int64_t) b.lo));
```

```
    // 下面两项只保留低32位，同时将低32位相加。
    auto ahxbl = a.hi * b.lo;
    auto alxbh = a.lo * b.hi;
    auto sum = ahxbl + alxbh;
    return {alxbl.hi + sum, alxbl.lo};
}
```

x86，gcc， `10000000*1000000` ：

```
        movl    $10000000, 24(%esp)
        movl    $0, 28(%esp)
        movl    $1000000, 16(%esp)
        movl    $0, 20(%esp)
        movl    28(%esp), %eax
        imull   16(%esp), %eax
        movl    %eax, %edx
        movl    20(%esp), %eax
        imull   24(%esp), %eax
        leal    (%edx,%eax), %ecx
        movl    16(%esp), %eax
        mull    24(%esp)
        addl    %edx, %ecx
        movl    %ecx, %edx
        movl    %eax, 8(%esp)
```

mips下也是类似的，不过mips有两个特殊的寄存器 `Hi` 和 `Lo` ，分别用来存放两个32位寄存器相乘产生的64位值的高低部分。

除

除和模是最为复杂的所以需要耐心，这里的主要内容是根据gcc的 `__divdi3` 实现来的， `glibc/stdlib/longlong.h` 等几个文件包含了一些和 `__divdi3` 有关的宏和函数，我把它们都整理在同一个文件夹中了，不过我只整理了i386和mips两种体系结构有关的代码。

先从最大的外层来说，先不看里面的细节，就看看分了几种情况进行讨论就好了。在看代码之前先来看看几个类型的宏和结构体，这里和源码有出入，不过差不多：

```
#define W_TYPE_SIZE    32

#define USItype uint32_t
#define UDItype uint64_t
#define  SItype  int32_t
#define  DItype  int64_t

#define UWtype       USItype
```

```c
#define UDWtype      UDItype
#define  DWtype       DItype
#define   Wtype       SItype


struct DWstruct {
    Wtype low, high;
};

typedef union {
    struct DWstruct s;
    DWtype ll;
} DWunion;
```

三个宏函数，这里先只说其作用，不说如何实现的：

```
umul_ppmm:  pp = m * m，两个单字相乘结果为一个双字。
sub_ddmmss: dd= mm - ss，一个字母代表一个字，也就是说这是双字的减法。
udiv_qrnnd: q = nn / d，r = nn %d，双字÷单字。
```

`__divdi3` ：

```c
DWtype __divdi3(DWtype u, DWtype v) {
    Wtype c = 0;
    DWtype w;
    // 先全部变为正数。
    // 记录翻转次数，如果翻转两次仍旧为正。
    if (u < 0) {
        c = ~c;
        u = -u;
    }
    if (v < 0) {
        c = ~c;
        v = -v;
    }
    w = __udivmoddi4(u, v, NULL);
    if (c)
        w = -w;
    return w;
}
```

`__udivmoddi4` ：

```c
static UDWtype
__udivmoddi4(UDWtype n, UDWtype d, UDWtype *rp) {
    // 下面是一些变量的初始化。
    DWunion ww;
    DWunion nn, dd;
    DWunion rr;
    UWtype d0, d1, n0, n1, n2;
    UWtype q0, q1;
    UWtype b, bm;

    nn.ll = n;
    dd.ll = d;

    d0 = dd.s.low;
    d1 = dd.s.high;
    n0 = nn.s.low;
    n1 = nn.s.high;

    // 当除数 (divisor) 的高32位d1为0。
    if (d1 == 0) {
        if (d0 > n1) {
            /* 0q = nn / 0D */
            // 计算d0开头有多少0，d0高16位非0才能保证udiv_qrnnd的正确性。
            bm = count_leading_zeros(d0);

            if (bm != 0) {
                /* Normalize, i.e. make the most significant bit of the
              denominator set.  */
                // Normalize可以保证udiv_qrnnd不会发生溢出，可以用最极端的0xFFFFFFFF÷0x1看看是否

                d0 = d0 << bm;

                // n1:n0 << bm
                // 由于 d0 > n1，所以前面的0肯定比n1少或者一样多，所以不会造成n1:n0有效位丢失。
                n1 = (n1 << bm) | (n0 >> (W_TYPE_SIZE - bm));
                n0 = n0 << bm;
            }
            // q0 = n1:n0 / d0
            udiv_qrnnd (q0, n0, n1, n0, d0); // 这里一定不会发生除法溢出，因为最高位是1，后面也是
            // 此时商高位一定为0。我先没有办法证明这一点，但是试了好几次发现确实如此。
            q1 = 0;

        } else {
            /* qq = NN / 0d */

            if (d0 == 0) // d1:d0=0，保持语义，发生一个异常，上面的d0 > n1能保证d0!=0。
```

```
            d0 = 1 / d0;    /* Divide intentionally by zero.  */

        bm = count_leading_zeros(d0);

        // bm == 0 是一个特殊情况。
        if (bm == 0) {
            /* From (n1 >= d0) /\ (the most significant bit of d0 is set),
           conclude (the most significant bit of n1 is set) /\ (the
           leading quotient digit q1 = 1).
           This special case is necessary, not an optimization.
           (Shifts counts of W_TYPE_SIZE are undefined.)  */
            // 当bm=0，b=32，会导致未定义的行为，所以下面的情况不适用。
            // d0在高位为1且n1>=d0的情况下，两者相除商为1，同时产生余数，这个余数是小于d0的，余
            n1 -= d0; // 余数
            q1 = 1;
        } else {
            /* Normalize.  */

            b = W_TYPE_SIZE - bm;

            d0 = d0 << bm;
            // 用n2放可能被溢出的有效位。
            n2 = n1 >> b;
            n1 = (n1 << bm) | (n0 >> b);
            n0 = n0 << bm;
            // 就是一般的竖式计算的方式，先和高位计算出高位，然后余数n1乘位宽和n0组成新的被除数，
            // 可以用十进制数试试看，原理差不多。
            udiv_qrnnd (q1, n1, n2, n1, d0);
        }

        /* n1 != d0...  */

        // 上次计算的余数作为高位。
        udiv_qrnnd (q0, n0, n1, n0, d0);

    }
} else {
    // 当d1!=0，思路还是类似的。
    if (d1 > n1) {
        // 此时 DD > nn，所以商为0。
        /* 00 = nn / DD */

        q0 = 0;
        q1 = 0;

        /* Remainder in n1n0.  */
```

```
} else {
    /* 0q = NN / dd */

    bm = count_leading_zeros( d1);
    if (bm == 0) {
        /* From (n1 >= d1) /\ (the most significant bit of d1 is set),
           conclude (the most significant bit of n1 is set) /\ (the
           quotient digit q0 = 0 or 1).
           This special case is necessary, not an optimization.  */

        /* The condition on the next line takes advantage of that
           n1 >= d1 (true due to program flow).  */
        // n1:n0 > d1:d0
        // 由于这里保证n1 >= d1，所以还需要n0 >= d0就可以保证大于号成立。
        if (n1 > d1 || n0 >= d0) {
            q0 = 1; // 最高位是0的情况下最大商为1，0xF÷0x8=1。
            sub_ddmmss (n1, n0, n1, n0, d1, d0);
        } else
            q0 = 0;

        q1 = 0;

    } else {
        UWtype m1, m0;
        /* Normalize.  */

        b = W_TYPE_SIZE - bm;

        d1 = (d1 << bm) | (d0 >> b);
        d0 = d0 << bm;
        n2 = n1 >> b;
        n1 = (n1 << bm) | (n0 >> b);
        n0 = n0 << bm;

        udiv_qrnnd (q0, n1, n2, n1, d1);
        // 各自忽略了最低的32位，这种方式实际上导致了向上取整[6]。

        umul_ppmm (m1, m0, q0, d0); // m1:m0 = q0xd0
        // n1 = (n2:n1) % d1
        // 即m1:m0 > n1:n0，q0xd0 > n1:n0
        // (n1:n0)是(n2:n1:n0)÷(d1:0)的余数，第二个n1是udiv_qrnnd之前的n1。
        // 商q0 乘上 被忽略的d0 比上面那个余数大，那么就应该减1。
        // 对十进制的529÷79的竖式计算和529÷70的竖式计算进行对比就知道发生什么了。
        // 将52÷7的余数3组合上9就可以还原为529÷70的余数39。然后商7乘上9=63，这个就是不忽略
        if (m1 > n1 || (m1 == n1 && m0 > n0)) {
            q0--;
```

```
                sub_ddmmss (m1, m0, m1, m0, d1, d0);
            }
            q1 = 0;
        }
    }
}
ww.s.low = q0;
ww.s.high = q1;
return ww.ll;
}
```

umul_ppmm、sub_ddmmss、udiv_qrnnd，这3个宏函数在不同体系结构下有不同的定义，我们这里只关注mips下和x86下的，因为两个体系结构下的实现非常不同，也算是有代表性了：

```
// x86的64位整数相减，结果放在sh:sl。
// 花括号里面的东西是为了支持dialects[6]。
#define sub_ddmmss(sh, sl, ah, al, bh, bl) \
  __asm__ ("sub{l} {%5,%1|%1,%5}\n\tsbb{l} {%3,%0|%0,%3}"        \
       : "=r" ((USItype) (sh)),                    \
         "=&r" ((USItype) (sl))                    \
       : "0" ((USItype) (ah)),                    \
         "g" ((USItype) (bh)),                    \
         "1" ((USItype) (al)),                    \
         "g" ((USItype) (bl)))

// mips中逻辑和c++模拟代码类似。

// mips的32位整数相乘，结果放在w1:w0。
#define umul_ppmm(w1, w0, u, v)                                        \
  do {                                                                 \
    UDItype __x = (UDItype) (USItype) (u) * (USItype) (v);             \
    (w1) = (USItype) (__x >> 32);                                      \
    (w0) = (USItype) (__x);                                            \
  } while (0)

// x86上只使用了一条mul指令实现。

// 通用的64位整数÷32位整数，余数放在r，商放在q。
#define __udiv_qrnnd_c(q, r, n1, n0, d) \
  do {                                 \
    UWtype __d1, __d0, __q1, __q0;                 \
    UWtype __r1, __r0, __m;                   \
    __d1 = __ll_highpart (d);                    \
    __d0 = __ll_lowpart (d);                   \
                                         \
```

```
    __r1 = (n1) % __d1;                             \
    __q1 = (n1) / __d1;                             \
    __m = (UWtype) __q1 * __d0;                         \
    __r1 = __r1 * __ll_B | __ll_highpart (n0);                \
    if (__r1 < __m)                             \
      {                                     \
    __q1--, __r1 += (d);                            \
    if (__r1 >= (d))  \
      if (__r1 < __m)                           \
        __q1--, __r1 += (d);                        \
      }                                     \
    __r1 -= __m;                                \
                                        \
    __r0 = __r1 % __d1;                             \
    __q0 = __r1 / __d1;                             \
    __m = (UWtype) __q0 * __d0;                         \
    __r0 = __r0 * __ll_B | __ll_lowpart (n0);               \
    if (__r0 < __m)                             \
      {                                     \
    __q0--, __r0 += (d);                            \
    if (__r0 >= (d))                            \
      if (__r0 < __m)                           \
        __q0--, __r0 += (d);                        \
      }                                     \
    __r0 -= __m;                                \
                                        \
    (q) = (UWtype) __q1 * __ll_B | __q0;                    \
    (r) = __r0;                                 \
  } while (0)

// mips并没有定义__udiv_qrnnd而是使用了通用的__udiv_qrnnd_c，而x86上的__udiv_qrnnd实现非常简单，因
#define __udiv_qrnnd __udiv_qrnnd_c
```

重点在 `__udiv_qrnnd_c` 是如何实现的，代码看着很长，其实很大一部分逻辑都是一样的，这里的逻辑其实和 `__udivmoddi4` 的最后一种情况有点类似。这个算法是*knuth*提出的，叫做**algorithm D**[5]，链接给出了一个很不错的讲解，自己找个数字试试看应该能知道大致做了什么。网上能够找到的**algorithm D**算法都是循环实现的[7]，而gcc中只减了两次，难道是能够保证估算的误差在2之内么？暂时不深究了吧。

模

gcc中使用了 `__umoddi3` 来实现模运算，而其中又使用了 `__udivmoddi4` 来实现：

```
// glibc/sysdeps/wordsize-32/divdi3.c
UDWtype
__umoddi3 (UDWtype u, UDWtype v)
```

```
{
  UDWtype w;

  __udivmoddi4 (u, v, &w);
  return w;
}
```

在上面展示 `__udivmoddi4` 时，我删除了和取余有关的部分，不过这部分很简单加回来看就好了

参考

[1] Adding two 64 bit numbers in Assembly

[2] 64-bit types and arithmetic on 32-bit CPUs

[3] Assembly: 64 bit multiplication with 32-bit registers

[4] Specific situations to use addi vs addiu in MIPS

[5] Labor of Division (Episode IV): Algorithm D

[6] Extended Asm (Using the GNU Compiler Collection (GCC))

[7] Donald Knuth’s "Algorithm D", its Implementation in "Hacker’s Delight", and elsewhere