

# ICS-LAB1 实验报告

计算机科学与技术系

祖东珏

171180541

gwynsmilezjdj@gmail.com

218.10.12

## 一、实现 multimod

通过定义结构 `Big_int { uint64_t num[2]; }` 并实现有关 `Big_int` 的相关运算：`leftmove(<<)`, `rightmove(>>)`, `big_equal(==)`, `big_above(>)`, `big_add(+)`, `big_sub(-)`, `big_mul(*)`, `big_mod(%)`。在完成相关运算后，只要通过 `big_mul(*)` 计算两数的乘积，在用 `big_mod(%)` 计算模即可。

为了保证基准实现的正确性，采用随机数进行测试。由于 C 语言中不支持 128 位整数运算，因此使用支持 128 位整数的 python 语言生成  $[0, 2^{63}-1]$  随机数 a, b, m，并进行运算后输出至 data.txt，

```
1 import random
2 i=1;
3 fo = open("data", "w")
4 while( i <= 1000000):
5     i = i + 1
6     a=random.randint(0,2**63-1)
7     b=random.randint(0,2**63-1)
8     m=random.randint(0,2**63-1)
9     fo.write(str(a))
10    fo.write(" ")
11    fo.write(str(b))
12    fo.write(" ")
13    fo.write(str(m))
14    fo.write(" ")
15    fo.write(str((a*b)%m))
16    fo.write("\n")
17 fo.close()
18
```

```
769977 4567943344476104977 7141763060732500498 4936173378180325753 65279099431231776
769978 2555546457247899984 8486820293045060607 4741007117427644788 2414984139253158248
769979 136502957655499057 979174601508126528 1847290525389599288 1529551884045764968
769980 4751796204960603813 2560340525188976079 9196487788971804997 8821095982027506857
769981 8872620055788577357 3921794426795367762 8150082318430093395 1972233878129512209
769982 3517088182055006860 2173865810170577246 7036795221109844523 309506577765707144
769983 4823404564349353780 5174969891944227244 447059635841690657 107849880509692291
769984 6083083461720407760 4192916715777648536 1533060383381571944 1498733943179565536
769985 8308097024061642682 2281845077517962021 7725418899374299798 7209831915460487074
769986 7947380466682821739 2826557096792743675 2187813393088687841 940845065825513314
769987 3388173261525965 3913033530568873639 2094956601399942282 1055802469629290309
769988 281279934680767809 2619068519098149562 4311532878363009625 522294179185542783
769989 7452382514639756537 2757238818512165126 168527524663985153 140141943636415195
769990 4697125892366449803 6851578032316298365 5367742031350987465 721873439889398180
769991 1678640251872991257 8348800696979670341 3588637947331485305 1922197083710887247
769992 5995912134817666454 6010195927480733559 3096174722736513262 1722373603894364912
769993 8163623038486170819 4465982042213974003 759774015349260519 469969719170335760
769994 1160741617148449913 2436899735575367517 5839052646678041263 2020196734519769804
769995 1212924922945289889 7910493518931862596 797550843816753448 732945740663231340
769996 3120656813291654617 5785781812096539680 8320337307180175506 1599569476472382320
769997 3804092220434709669 4070762959395203863 2371991902707822641 158826672784473891
769998 141936778373961804 2494594323197675165 1324736434115044539 1301044692741365885
769999 1248627245584619440 5328102277107773188 5887097055438002194 2221845603500173536
770000 073598553860476684 4463646210240987899 1413557568297545289 1196543911628913796
770001 4710167018375368640 6207434427996005411 8065381153036023761 1680529551681668208
770002 1955232786545952409 7741937548792704661 317635177883988147 276596870191256767
770003 2469565128504864160 2623649445108141820 2802216202082831379 2708233439122662316
770004 2092147856869352478 3933400196684769946 5870500861008042457 3855353733823207907
770005 5143258022507861620 5514255011126798339 6928782987754935808 6201009531295593308
770006 6881624137929879650 6378535269574258668 8053692657123534618 721147568628430098
770007 7227348487329669656 2124731718978015525 2847298937146609995 1457739078486233970
770008 6759470175041877255 6764495796455373160 5917820403809179863 3244119446721193722
770009 5380229380459239883 7830952331909394912 4278209016845588934 2772116435565060924
770010 3014837109603239891 1050462225665951322 3542367662174456673 2957818054899184058
770011 1667991446123368591 1896725710066358658 8074913058421067404 1988034054308034198
770012 6237644513260672258 3457533269168239529 5035403998382432642 1788119345422560212
770013 6547926838091238715 4773506500101962635 640779698472191128 403601200719038281
770014 7869968967822134967 5496454255945598235 2768620271541127606 836370818415167271
770015 2051210612075898126 18511330823192211 4256397852095560749 2400367065116083104
770016 914168229729770265 2165799010887510534 3669036963700952703 2330410152482961294
770017 3391951431378518949 9188132322659031972 3213557938917490144 102245646674090932
770018 2105619643956634792 3000434120533871173 2727316235357494797 360032036731794271
770019 3055621639340438052 4429861002782363518 6890661820242870888 5398389364290304824
770020 1378437544407080240 6359930714833813007 4796203145369947641 1042980536474652259
770021 7321085802066093115 4193763467479387091 7866036814143717840 2364390300871636625
770022 6281661590307691026 569016523226427320 8508768906603736042 8471841340065118856
770023 4790426059038210340 2139436450593066222 8317595465998848388 760268988939125868
~/lab1/data[1] unix utf-8 Ln 770000, Col 1/100000
```

再将 a, b, m 和运算结果读入，与 multimod 得出的结果对比。若相同则输出“success!”，若不相同则 1 输出错误信息“Fail!”。若测试中出现一次“Fail”则测试立刻停止。由于 a, b, m 取值范围都较大，因此要用较多的测试用例，这里选取了 1,000,000 个测试用例，最后全部输出正确。

实现中用到的运算只有加法、乘法可能造成溢出，而又因为乘法由加法和移位实现，运用的是无符号数运算，因此不会产生有符号数溢出。

## 二、性能优化

将 B 按二进制展开可得： $b = b[62] \cdot 2^{62} + \dots + b[0] \cdot 2^0$ ，因此， $a \cdot b$  可以写成：

$$a \cdot b = a \cdot b[62] \cdot 2^{62} + a \cdot b[61] \cdot 2^{61} + \dots + a \cdot b[0] \cdot 2^0$$

$$\begin{aligned} &= (b[62] == 1 ? a \cdot 2^{62} : 0) + \\ &\quad (b[61] == 1 ? a \cdot 2^{61} : 0) + \dots \\ &\quad (b[0] == 1 ? a \cdot 2^0 : 0) \end{aligned}$$

用同余的性质，只需要知道  $a, 2a, 4a, \dots, 2^{62} \cdot a$  模 m 的值即可。又  $2a = a + a, 4a = 2a + 2a, \dots, 2^{62} \cdot a = 2^{61} \cdot a + 2^{61} \cdot a$ ,

因此可以递推求值。即，在计算  $a \cdot b \bmod m$  前，先计算  $a, 2a, 4a, \dots, 2^{62} \cdot a \bmod m$  的值并运用一个数组来存储，具体计算过程为：第一步计算  $a \bmod m$ ，存储在 `base` 中，第二步计算 `base = (base + base) % m`，即可知  $2a \bmod m$  的值，后面的结果都依次递推。

为了验证正确性，运用相同的方法，使用 python 生成随机数和结果测试。结果如下：

```
4 uint64_t src[63];
5
6 int64_t multimod(int64_t a, int64_t b, int64_t m) {
7     uint64_t base = a % m;
8     src[0]=base;
9     for(int i=1; i<=62; i++) {
10         base = (base<<1) % m;
11         src[i]=base;
12     }
13     uint64_t res = 0;
14     int c = 0;
15     while(b!=0) {
16         res = (b&0x1)==1? (res+src[c])%m : res;
17         b = b>>1;
18         c = c+1;
19     }
20 }
21
22 int main() {
23     uint64_t a,b,m,real_re,re,mul;
24     int i=0;
25     FILE *fp=fopen("data","r");
26
27 int main() {
28     uint64_t a,b,m,real_re,re,mul;
29     int i=0;
30     FILE *fp=fopen("data","r");
31     while(!feof(fp)) {
32         i++;
33         fscanf(fp,"%llu %llu %llu\n",&a,&b,&m,&real_re);
34         re = multimod(a,b,m);
35         if(re != real_re) {
36             printf("%d:FAIL! real=%llu get=%llu\n",i,real_re,re);
37             break;
38         }
39         else
40             printf("%d:Success!\n",i);
41     }
42     return 0;
43 }
```

```
99972:Success!
99973:Success!
99974:Success!
99975:Success!
99976:Success!
99977:Success!
99978:Success!
99979:Success!
99980:Success!
99981:Success!
99982:Success!
99983:Success!
99984:Success!
99985:Success!
99986:Success!
99987:Success!
99988:Success!
99989:Success!
99990:Success!
99991:Success!
99992:Success!
99993:Success!
99994:Success!
99995:Success!
99996:Success!
99997:Success!
99998:Success!
99999:Success!
100000:Success!
gwynsmile@8adcd289d027:~/lab1$
```

性能评估：两种实现分别使用 `-O0 -O1 -O2` 三种优化选项编译后，计算 500,000 组随机输入，统计计算时间/时钟周期用以评价程序性能。

方法 1：运用 Timer 计时

实现 1，p1.c:

```
int main() {
    uint64_t a,b,m,re;
    FILE *fp=fopen("data","r");
    Big_int x,y;
    clock_t start,end;
    double total;
    start = clock();
    while(!feof(fp)) {
        fscanf(fp,"%llu %llu %llu\n",&a,&b,&m);
        x.num[0]=a;
        x.num[1]=0;
        y.num[0]=b;
        y.num[1]=0;
        re = multimod(a,b,m);
    }
    end = clock();
    total =(double)(end-start)/CLOCKS_PER_SEC;
    printf("use time: %lf\n",total);
    printf("%llu\n",re);
    return 0;
}
```

```
gwynsmile@8adcd289d027:~/lab1$ gcc -O0 -o p1 p1.c
gwynsmile@8adcd289d027:~/lab1$ ./p1
use time: 2.062726
1851188733189480754
gwynsmile@8adcd289d027:~/lab1$ gcc -O1 -o p1 p1.c
gwynsmile@8adcd289d027:~/lab1$ ./p1
use time: 0.974796
1851188733189480754
gwynsmile@8adcd289d027:~/lab1$ gcc -O2 -o p1 p1.c
gwynsmile@8adcd289d027:~/lab1$ ./p1
use time: 0.708620
1851188733189480754
```

为了保证在 `-O0 -O1 -O2` 的编译中不会将 `multimod` 指令删除，使用 `objdump` 查看在 `-O2` 优化选项下编译出的汇编代码，发现在 `main` 函数的每一次循环中都会执行 `callq c20<multimod>`，因此可以确定优化不影响测试目的，并且不同环境下输出的。因此，实现 1 在 `-O0 -O1 -O2` 的优化下执行时间分别为 2.062726 s, 0.974796 s, 0.708620 s。


```
710: 4d 89 f8          mov     %r15,%r8
713: 4c 89 f1          mov     %r14,%rcx
716: 4c 89 ea          mov     %r13,%rdx
719: 4c 89 e6          mov     %r12,%rsi
71c: 48 89 df          mov     %rbx,%rdi
71f: 31 c0            xor     %eax,%eax
721: e8 3a ff ff ff    callq  660 <_isoc99_fscanf@plt>
726: 48 8b 54 24 28    mov     0x28(%rsp),%rdx
72b: 48 8b 74 24 20    mov     0x20(%rsp),%rsi
730: 48 8b 7c 24 18    mov     0x18(%rsp),%rdi
735: e8 e6 04 00 00    callq  c20 <multimod>
73a: 48 89 c5          mov     %rax,%rbp
73d: 48 89 df          mov     %rbx,%rdi
740: e8 4b ff ff ff    callq  690 <feof@plt>
745: 85 c0            test    %eax,%eax
747: 74 c7            je      710 <main+0x50>
```



实现 2, p2.c:

```
int main() {
    uint64_t a,b,m,re;
    FILE *fp=fopen("data","r");
    clock_t start,end;
    double total;
    start = clock();
    while(!feof(fp)) {
        fscanf(fp,"%llu %llu %llu\n",&a,&b,&m);
        re = multimod(a,b,m);
    }
    end = clock();
    total = (double)(end-start)/CLOCKS_PER_SEC;
    printf("use time: %lf\n",total);
    printf("%llu\n",re);
    return 0;
}
```

```
gwynsmile@8adcd289d027:~/lab1$ gcc -O0 -o p2 p2.c
gwynsmile@8adcd289d027:~/lab1$ ./p2
use time: 0.922059
1851188733189480754
gwynsmile@8adcd289d027:~/lab1$ gcc -O1 -o p2 p2.c
gwynsmile@8adcd289d027:~/lab1$ ./p2
use time: 0.600014
3849930299508495731
gwynsmile@8adcd289d027:~/lab1$ gcc -O2 -o p2 p2.c
gwynsmile@8adcd289d027:~/lab1$ ./p2
use time: 0.585228
0
```



三种优化下最后输出的 re 值都是计算最后一组测试用例的到的, 应当相同。而从上面结果可以发现, 在-O0 的优化下结果是正确的, 而在-O1 和-O2 的优化下结果错误。对比 C 代码和-O1 下的汇编代码:

```
int64_t multimod(int64_t a,int64_t b,int64_t m) {
    uint64_t base = a % m;
    src[0]=base;
    for(int i=1;i<=62;i++) {
        base = (base<<1) % m;
        src[i]=base;
    }
    uint64_t res = 0;
    int c = 0;
    while(b!=0) {
        res = (b&0x1)==1? (res+src[c])%m : res;
        b = b>>1;
        c = c+1;
    }
}
```

```
00000000000007f0 <multimod>:
7f0: 48 89 f8      mov     %rdi,%rax
7f3: 49 89 d0      mov     %rdx,%r8
7f6: 48 99        cqto
7f8: 49 f7 f8      idiv    %r8
7fb: 48 89 d0      mov     %rdx,%rax
7fe: 48 89 15 7b 08 20 00 mov     %rdx,0x20087b(%rip)
805: 48 8d 0d 7c 08 20 00 lea     0x20087c(%rip),%rcx
80c: 48 8d 3d 65 0a 20 00 lea     0x200a65(%rip),%rdi
813: 48 01 c0      add     %rax,%rax
816: ba 00 00 00 00 mov     $0x0,%edx
81b: 49 f7 f0      div     %r8
81e: 48 89 d0      mov     %rdx,%rax
821: 48 89 11      mov     %rdx,(%rcx)
824: 48 83 c1 08   add     $0x8,%rcx
828: 48 39 f9      cmp     %rdi,%rcx
82b: 75 e6        jne     813 <multimod+0x23>
82d: 48 85 f6      test    %rsi,%rsi
830: 74 05        je      837 <multimod+0x47>
832: 48 d1 fe      sar     %rsi
835: 75 fb        jne     832 <multimod+0x42>
837: f3 c3      repz retq
```

经过对比可知, 优化并没有影响求  $a, 2a \cdots 2^{62}a$  模  $m$  的值并存储在数组中这一过程(对应汇编指令 813-82b)但是影响了计算结果的语句, 似乎将计算语句仅仅保留了 `while(b!=0) { b=b>>1; }`, 从而使得结果错误。

经过检查, 由于 multimod 缺少了 return, 导致编译器自动删除了未产生副作用的语句。而在-O0 时没有 return 语句也能生成正常结果是由于 res 变量正好使用 %rax 寄存器, 碰巧产生正常结果。加上 return 语句后, 再次在三种优化选项下编译运行:

这时三种情况下的结果都正确, 运行时间分别为: 0.970960s, 0.839891s, 0.871181s。可


以发现优化对运行时间影响不大。与实现 1 对比可以发现, 在-O0 优化的情况下, 实现 2 效率明显高与实现 1, 而随着编译优化程度的增加, 实现 2 与实现 1 效率差距在变小。在-O2 的选项下, 实现 1 的效率高于实现 2。

方法 2: 尝试使用 rdtsc 指令获取指令执行周期数

本想使用 rdtsc 指令, 但是由于 CPU 核心的主频不是恒定的了, time stamp counter 的值不代表时间了; 同时, 又由于 CPU 有多个核心, 这些核心之间的 time stamp counter 不一定是同步的, 所以当进程在核心之间迁移后, rdtsc 的结果就未必有意义。又由于 CPU 执行指令可能乱序, 这可能使测试代码完全失去意义。因此可以使用 rdtsc 指令, rdtsc 可以保证指令顺序执行。

先使用简单的代码测试 rdtsc 的可行性:

```
gwynsmile@8adcd289d027:~/lab1$ gcc -O0 -o p2 p2.c
gwynsmile@8adcd289d027:~/lab1$ ./p2
use time: 0.970960
1851188733189480754
gwynsmile@8adcd289d027:~/lab1$ gcc -O1 -o p2 p2.c
gwynsmile@8adcd289d027:~/lab1$ ./p2
use time: 0.839891
1851188733189480754
gwynsmile@8adcd289d027:~/lab1$ gcc -O2 -o p2 p2.c
gwynsmile@8adcd289d027:~/lab1$ ./p2
use time: 0.871181
1851188733189480754
```



```
typedef unsigned long long cycles_t;

inline cycles_t currentcycles()
{
    cycles_t result;
    __asm__ __volatile__ ("rdtscp" : "=A" (result));
    return result;
}

cycles_t t1, t2;

int main(void)
{
    double result = 100;
    t1 = currentcycles();
    for(int i=0;i<=100000;i++) {
        result*=0.9999999;
    }
    t2 = currentcycles();
    printf("result is %lf\n", result);
    printf("you use cycles : %llu\n", t2-t1);
    return 0;
}
```

```
you use cycles : 321736
gwynsmile@8adcd289d027:~/lab1$ ./test
result is 99.004973
you use cycles : 551165
gwynsmile@8adcd289d027:~/lab1$ ./test
result is 99.004973
you use cycles : 417063
gwynsmile@8adcd289d027:~/lab1$ ./test
result is 99.004973
you use cycles : 312816
gwynsmile@8adcd289d027:~/lab1$ ./test
result is 99.004973
you use cycles : 596732
gwynsmile@8adcd289d027:~/lab1$ ./test
result is 99.004973
you use cycles : 331148
gwynsmile@8adcd289d027:~/lab1$ ./test
result is 99.004973
you use cycles : 441088
```

从测试结果发现，每次返回的周期数差距较大。这与 CPU 的复杂结构和执行过程有关，目前没有找到好的方法将误差缩减到可接受的范围，因此**无法使用** rdtscp 指令来度量程序执行周期数。

### 三、解析神秘代码

利用同样的 500,000 组测试数据对不同编译优化选项下的 multimod\_fast 进行测试，结果如下：

可以发现，在不同的优化选项下，multimod\_fast 的效率都是之前实现的 3-4 倍，效率很高。这是由于神秘代码仅仅使用了 C 语言中提供的运算符，并且只有一条语句，没有循环，访问内存等操作，大大提高了效率。

但由于测试数据中整数较大，结果有错误，说明此代码对 a,b,m 的范围有要求。分析如下 ( $0 < a, b < 2^{63}-1$ )：

```
gwynsmile@8adcd289d027:~/lab1$ gcc -g -O0 -o myth myth.c && ./myth
use time: 0.296349
7466834643511527350
gwynsmile@8adcd289d027:~/lab1$ gcc -g -O1 -o myth myth.c && ./myth
use time: 0.288684
7466834643511527350
gwynsmile@8adcd289d027:~/lab1$ gcc -g -O2 -o myth myth.c && ./myth
use time: 0.319379
7466834643511527350
```

```
int64_t multimod_fast(int64_t a, int64_t b, int64_t m) {

    int64_t t = a * b - (int64_t)((double)a * b / m + 1e-8) * m;

    return t < 0 ? t + m : t;

}
```

根据学过的知识可知，IEEE 754 标准中，64 位的双精度浮点类型由 1 位符号位，11 位指数位，52 位尾数位。由于尾数带一个隐藏位，因此连续表示的整数中最大的一个是 100……000 (53 个 0)，即  $2^{53}$ 。当整数大于  $2^{53}$  时，double 表示会损失精度，数值越大，损失的精度越多。可以得出，当用 double 表示整数  $s \in (2^n, 2^{(n+1)})$  且  $n \geq 53$  时，任意两个 s 之间的差最小是  $2^{(n-52)}$ ，令 double 类型可表示的整数叫“可表示整数”。因此，若有一个整数在两个“可表示整数”之间，则下规至最近的“可表示整数”。

首先从最简单的情况，当  $a*b \leq 2^{53}$ ，此时  $a*b$  不会触发整数溢出，用 double 来存储也不会损失精度。在将  $a*b/m$  的结果转换回 int64\_t 时会进行小数舍入，只保留整数部分。假设舍入时丢失的值为 lost， $lost \in (0,1)$ 。根据模运算性质可知：

$$a * b \equiv t \pmod{m} \Leftrightarrow \exists k \in \mathbb{Z}, k > 0, a * b = k * m + t$$

而当 k 取得**最大值**时，有  $t \in [0, m)$ ，即  $a*b \bmod m = t$ 。由此可知，代码中  $(\text{int64\_t})((\text{double})a*b/m + 1e-8)$  求的就是 k 的最大值，因此 t 就是  $a*b \bmod m$  的结果。这是这个算法的核心原理。剩下只需要证明对其余的特定范围内 a,b,m 成立。

首先证明，只取低 64 位进行运算不会导致结果错误，只要证明  $a*b$  的高 64 位和  $(\text{int64\_t})((\text{double})a * b / m + 1e-8) * m$  的高 64 位相同且  $a*b > (\text{int64\_t})((\text{double})a * b / m + 1e-8) * m$ ，两数相减便会将高 64 位清零，不会对最终结果 t 产生影响。首先，由于从 int64\_t 转为 double 时可能下规，double 转为 int64\_t 可能截断小数，所以易证  $(\text{int64\_t})((\text{double})a*b/m + 1e-8) > a*b$ ；另外，根据上述算法原理可知， $t \in [0, m)$  且  $m < 2^{63}$ ，所以  $(\text{int64\_t})((\text{double})a * b / m + 1e-8) * m$  的值与  $a*b$  的值相差不超过  $2^{63}$ ，即高 64 位相同。综上，得证算法中只取低 64 位运算不会产生错误。

该算法中误差产生于从 int64\_t 转换到 double 时：当  $a, b \leq 2^{53}$  且  $a*b > 2^{53}$  时，在将  $a*b$  转换到 double 类

型时就会下规，可能产生的最大误差为  $2^{(n-52)}$ 。当  $a > 2^{53}$  或  $b > 2^{53}$  时，将  $a/b$  转换为 double 类型是就会产生下规。

首先假设  $a, b < 2^{53}$ ，这样误差只产生于  $a*b$  转换为 double 时。假设由于下规， $a*b$  的结果减少了  $p \in (0, 2^{(n-52)})$ 。为了得到正确的结果，要使  $(a*b-p)/m+1e-8$  和  $(a*b)/m+1e-8$  的整数部分相同。由于  $2^{(-12)} < e^{(-8)} < 2^{(-11)}$ ，因此只要保证  $2^{(n-52)}/m < 2^{(-12)}$  即可。假设  $a < 2^A, b < 2^B, m < 2^M$ ，则  $A+B < M+40$ ；因为  $M < 63$ ，所以  $A+B < 103$ 。然而，在实测中  $A+B \leq 97$  时才能得到正确答案，这其中的差距原因尚不明了。

当  $a, b > 2^{53}$  时，误差明显增大不符合要求。

所以这个在整数运算中看似奇怪的  $1e-8$  实际上是为了在一定程度上防止下规对结果的影响。但这个  $1e-8$  也会有一定副作用。

如果  $(double)(a*b/m)$  的结果小数部分很大，接近 1，在加上  $1e-8$  后可能会导致整数部分加 1，而实际结果应当是在转为 `int64_t` 时将小数部分舍去。这样的结果就是可能导致多减去了一个  $m$ ，所以在返回结果时要判断如果返回值小于 0，说明产生了这种情况，这样就在返回值上加一个  $m$ ，以得到正确答案。