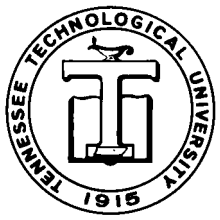
.

2004

. 2004-5



, 38505

Writing DLL in Assembler for External Calling in Maple

*by **Alec Mihailovs, Ph.D.***
Department of Mathematics
Tennessee Tech University

alec@mihailovs.com

NOTE: The article shows how to write assembler code using x86 and FPU registers. An example of calculating large factorials mod *m* is discussed in detail.

Introduction

This article started from a posting [1] in comp.soft-sys.maple newsgroup where I wrote:

In general, such calculations as huge factorials mod *p* should be programmed directly in assembly. It is fairly easy and doesn't require much assembly knowledge - just a few instructions. It can be compiled as a library (dll) and accessed from Maple through external calling.

Since quite a few people expressed an interest, I wrote the assembly code for that particular problem. I hope that it will be useful as a jump-start for writing your own DLL for the innermost loops repeating a billion times or more. For loops repeating "only" a

Facmod.asm

```
.586                                    ; for 586 processor or better
.model flat, stdcall                    ; 32-bit memory and standard
                                        ; call
.code                                   ; the beginning of the code
                                        ; section
LibMain proc h:DWORD, r:DWORD, u:DWORD  ; the dll entry point
        mov eax, 1                      ; if eax is 0, the dll won't
                                        ; start
        ret                             ; return
LibMain Endp                            ; end of the dll entry

Facmodp proc n:DWORD, p:DWORD           ; a function with dword
                                        ; parameters n and m
        push ebx                        ; save the ebx value
        mov ecx, n                      ; put n in the counter
                                        ; register ecx
        mov ebx, p                      ; put p in ebx
        mov eax, 1                      ; put 1 in eax
    L:                                  ; a loop label
        mul ecx                         ; multiply eax by ecx
        div ebx                         ; divide the product by p
        mov eax, edx                    ; move the remainder from edx
                                        ; to eax
        dec ecx                         ; decrease the counter by 1
        jnz L                           ; repeat the loop if the
                                        ; counter is not 0
        pop ebx                         ; restore the ebx value
        ret                             ; return eax
Facmodp endp                            ; end of the function
End LibMain                             ; end of the dll
```

Facmod.def

```
LIBRARY   Facmod
EXPORTS   Facmodp
```

Makeit.bat

```
@echo off

if exist Facmod.obj del Facmod.obj
if exist Facmod.dll del Facmod.dll

\masm32\bin\ml /c /coff Facmod.asm

\masm32\bin\Link /SUBSYSTEM:WINDOWS /DLL /DEF:Facmod.def Facmod.obj

pause
```

That will produce 4 files, **Facmod.dll**, **Facmod.lib**, **Facmod.exp**, and **Facmod.obj**.

Files **Facmodp.def** and **Makeit.bat** are self-explanatory. I'll add detailed comments here on **Facmod.asm**.

Facmod.asm commented

```
.586
.model flat, stdcall
```

This is the standard beginning of the asm files for MASM32. They tell the assembler that the program will be used in computers with

```
        dec ecx
```
Decrease the counter (ecx) by 1. We start from *n* there. It becomes *n* - 1 after the first loop, *n* - 2 after the second etc.

```
        jnz L
```
If counter is not 0, return to label *L*, otherwise continue below.

```
        pop ebx
```
Restore the ebx value from the stack.

```
        ret
```
Return.

```
Facmodp endp
```
The end of our function.

```
End LibMain
```
The end of the DLL.

The integer return value of the function is located in the eax register (that is a standard thing in Windows).

For the sake of simplicity, I didn't do any optimization of the calculating procedure, or adding at the beginning that if $n > p - 1$, then the result is 0, or checking whether p is 0. I'll do that in Section 3.

Section II: External Calling of Facmodp in Maple

In Maple,

```
> Facmodp:=define_external(
        'Facmodp',
        'n'::integer[4],
        'p'::integer[4],
        'RETURN'::integer[4],
        'LIB'="F:/MyProjects/Assembly/Facmod/Facmod.dll"
):
```

with changing the LIB to the location of **Facmodp.dll** creates a function calculating *n*! mod *p*. For example,

```
> Facmodp(10^9,10^9+7);
                                698611116
```

that took just a few seconds on my computer instead of the many hours that a procedure written in Maple without external calling would.

Note that in the definition of external call we used the integer[4] data descriptor for *n* and

p. It takes 4 bytes, the same as DWORD in th

> **time()-tt;**

0.070

> **facmodp(2^31,nextprime(2^31));**

1787166461

> **facmodp(17,2^32-5);**

4006859126

> calculations Tuch fast11.8.94;

```
        jae E                           ; if n>=p, exit returning 0
        mov eax, 1                      ; put 1 in eax
        test ecx, ecx                   ; it is faster than jecxz E
        jz E                            ; if n=0, exit returning 1
    L:                                  ; a loop label
        mul ecx                         ; multiply eax by ecx
        div ebx                         ; divide the product by p
        mov eax, edx                    ; move the remainder from edx
                                        ; to eax
        test eax, eax                   ; if the remainder is 0,
        jz E                            ; then exit returning 0
        dec ecx                         ; decrease the counter by 1
        jnz L                           ; repeat the loop if the
                                        ; counter is not 0
    E:                                  ; an exit label
        pop ebx                         ; restore the ebx value
        ret                             ; return eax
Facmodm endp                            ; end of the function
```

To make a dll, we can use the same **Makeit.bat** file, but **Facmod.def** should be modified by adding the new export,

Facmod.def

```
LIBRARY   Facmod
EXPORTS   Facmodp
EXPORTS   Facmodm
```

In Maple, we can define the external call to Facmodm similarly to Facmodp,

```
> Facmodm:=define_external(
        'Facmodm',
        'n'::integer[4],
        'm'::integer[4],
        'RETURN'::integer[4],
        'LIB'="F:/MyProjects/Assembly/Facmod/Facmod.dll"
):
```

Test its speed,

```
> time(Facmodp(10^8,10^8+7));
                                5.878

> time(Facmodm(10^8,10^8+7));
                                6.390
```

So, Facmodm is about 10% slower, because of the additional operations inside the loop. However, it compensates by faster calculations for composite *p*,

```
> time(Facmodm(10^8,10^8+9));
                                0.049
```

Facmodm can be extended to larger values of n and p similarly to facmodp. I included the corresponding function facmodm in the module below.

Now, when we have fast function facmodp for prime *p* and fast function facmodm for composite *p*, we can write a module, including them as well as the function facmod

working in both cases, which chooses facmodp if *p* is prime and facmodm otherwise. Since we suppose that facmodp will be used only for prime *p*, we can delete checking whether *p* is prime from its definition. . Here is the module.

Facmod module

```
> Facmod:=module()
local Facmodp, Facmodm;
export facmod, facmodp, facmodm;
description "The functions in this module calculate n! mod
p. If p is prime, facmodp(n,p) should be used. If p is
composite, facmodm(n,p) should be used. If p can be prime
or composite, facmod(n,p) should be used. The functions
```

```
if p <= n then return 0
```

the line EXPORT Fmodp to the **Facmod.def** and run **Makeit.bat**. If Maple was using external calls to functions from **Facmod.dll**, it should be shut down and then started again. The restart command is not enough for replacing the dll.

The external call in this case looks slightly different than for Facmodp. First, notice that we used QWORD = 64 bit parameters. The corresponding Maple data descriptor is the integer[8]. Second, if we want to get the returned value from the FPU, it should be declared in the external call as a floating point number, float[8] to get higher precision.

```
> Fmodp:=define_external(
        'Fmodp',
        'n'::integer[8],
        'p'::integer[8],
        'RETURN'::float[8],
        'LIB'="F:/MyProjects/Assembly/Facmod/Facmod.dll"
):
```

It does calculations more slowly than Facmodp though. For example,

```
> tt:=time():
> Fmodp(10^8,10^8+7);
```

$$0.69861116\ 10^{8}$$

```
> time()-tt;
```

$$14.171$$

However, it doesn't crash Maple for p = 0.

```
> Fmodp(3,0);
```

$$\mathrm{Float}(\mathrm{undefined})$$

It would give a wrong answer if *n* is non-positive though. That's why it is a good idea again to use it not directly, but through a Maple procedure excluding such bad input cases.

Also, it can be optimized. Agner Fog wrote a great optimization manual [3] for different Pentium processors. The optimizations steps are different for different processors, so I'll do just one optimization here - replacing the slow fprem opcode with a calculation of *N* mod *p* using formula

$$N \bmod p = N -$$

```
        fld1                            ; st=1, st(1)=p, st(2)=n
        fdiv st, st(1)                  ; st=1/p, st(1)=p, st(2)=n
        fld1                            ; st=1, st(1)=1/p, st(2)=p,
                                        ; st(3)=n
    L:                                  ; a loop label
        fmul st, st(3)                  ; st=st*st(3)
        fld st                          ; save the product, st(2)=1/p,
                                        ; st(3)=p
        fmul st, st(2)                  ; st=product/p
        frndint                         ; st=round(product/p)
        fmul st, st(3)                  ; st=round(product/p)*p
        fsubp st(1), st                 ; st=product-
                                        ; round([product/p])*p
        fld1                            ; st=1, st(4)=counter
        fsub st(4), st                  ; st=1, st(4)=counter-1
        fcomp st(4)                     ; compare 1 and st(4) and pop
                                        ; the stack
        fnstsw ax                       ; store the status word in ax
        shr ah, 1                       ; it is faster than sahf
        jc L                            ; repeat the loop if the
                                        ; counter is > 1
        fstp st(3)                      ; put the return value at the
                                        ; bottom
        fcompp                          ; clear the floating point
                                        ; stack
        ret                             ; return st
Fmodp1  endp                            ; end of the function
```

Again, before using it, we have to add the export of Fmodp1 in **Facmod.def**, rebuild the dll, shut down Maple, and start it again. Here is the external call for it in Maple,

```
> Fmodp1:=define_external(
        'Fmodp1',
        'n'::integer[8],
        'p'::integer[8],
        'RETURN'::float[8],
        'LIB'="F:/MyProjects/Assembly/Facmod/Facmod.dll"
):
```

Compare the speed for the same example,

```
> time(Fmodp1(10^8,10^8+7));
```

11.756

Faster than Fmodp and about twice as slowly as Facmodp.

Because we used the rounding instead of truncation, the answer can be negative. For example,

```
> Fmodp1(1001,10^8+7);
```

$$-0.15297205\ 10^{8}$$

```
> 1001! mod (10^8+7);
```

84702802

> **%-(10^8+7);**

-15297205

Few examples that I did, gave the correct answers for *p* up to about
> **2^49;**

562949953421312

The larger numbers can give a wrong answer. For example,
> **a:=nextprime(7*10^14);**

$a := 700000000000051$

> **Fmodp1(50000,a);**

$0.624640410570770000 \quad 10^{14}$

> **50000! mod a;**

62464041057077

Correct!

> **a:=nextprime(75*10^13);**

$a := 750000000000037$

> **Fmodp1(50000,a);**

$0.122439131674460000 \quad 10^{15}$

> **50000! mod a;**

113105262280640

Wrong!

It would be interesting to find a precise bound *M* such that all Fmodp1(*n*, *p*) give the correct answers for positive integers *n* and *p* less than *M*.

As we did in Section III for Facmodp, we can improve Fmodp1 by giving the answer 1 for non-positive *n* and the fast answer 0 for $n > p - 1$. The following assembler code for Fmodm does that. It also adds checking if the answer is 0 inside the loop, the same as in Facmodm.

Fmodm

```
Fmodm   proc n:QWORD, p:QWORD           ; a function with qword
                                        ; parameters n and p
        fninit                          ; initialize the floating
                                        ; point stack
        fld1                            ; put 1 in the FPU for
                                        ; returning 1 if n<=1
        fild qword ptr [n]              ; st=n, st(1)=1
        fcomp st(1)                     ; compare n and 1 and pop the
                                        ; stack
        fnstsw ax                       ; store the status word in ax
        and ah, 41h                     ; it is faster than sahf
        jnz E                           ; if n<=1, exit returning 1
        fstp st                         ; clear the stack
```

```
fldz                            ; put 0 in the FPU for
                                ; returning 0 if n>=p
fild qword ptr [n]              ; st=n, st(1)=0
```

In Maple, we can define the external call to Fmodm similarly to the external call to Fmodp,

```
> Fmodm:=define_external(
        'Fmodm',
        'n'::integer[8],
        'm'::integer[8],
        'RETURN'::float[8],
        'LIB'="F:/MyProjects/Assembly/Facmod/Facmod.dll"
):
```

Test its speed on the same example,

```
> time(Fmodm(10^8,10^8+7));
```

14.221

Certainly, Fmodm is faster for composite *p*,

```
> time(Fmodm(10^8,10^8+9));
```

0.090

Finally, we can write a module FactorialMod similar to the module Facmod and including its functions modified by using external calls to Fmodp1 and Fmodm for *n* and/or *p* greater than 2^32-1.

FactorialMod module

```
> FactorialMod:=module()
local Facmodp, Facmodm, Fmodp1, Fmodm;
export facmod, facmodp, facmodm;
description "The functions in this module calculate n! mod
p. If p is prime, facmodp(n,p) should be used. If p is
composite, facmodm(n,p) should be used. If p can be prime
or composite, facmod(n,p) should be used. If n and/or p are
greater than 2^49, result may be wrong.";
Digits:=max(15,Digits);
Facmodp:=define_external(
        'Facmodp',
        'n'::integer[4],
        'p'::integer[4],
        'RETURN'::integer[4],
        'LIB'="F:/MyProjects/Assembly/Facmod/Facmod.dll"
);
Facmodm:=define_external(
        'Facmodm',
        'n'::integer[4],
        'm'::integer[4],
        'RETURN'::integer[4],
        'LIB'="F:/MyProjects/Assembly/Facmod/Facmod.dll"
);
Fmodp1:=define_external(
        'Fmodp1',
```

```
        'n'::integer[8],
        'p'::integer[8],
        'RETURN'::float[8],
        'LIB'="F:/MyProjects/Assembly/Facmod/Facmod.dll"
):
Fmodm:=define_external(
        'Fmodm',
        'n'::integer[8],
        'm'::integer[8],
        'RETURN'::float[8],
        'LIB'="F:/MyProjects/Assembly/Facmod/Facmod.dll"
):
facmodp := proc (n::nonnegint, p::posint)
local r;
    if p <= n then return 0
    elif n = 0 then return 1
    elif p > 4294967295 then
        if p > 562949953421312 then WARNING("result may be
wrong") end if;
        if 1/2*p < n then
            if n = p-1 then return p-1
            else return `mod`((-1)^(p-n)/round(Fmodp1(p-n-
1,p)),p)
            end if;
        else return `mod`(round(Fmodp1(n,p)),p)
        end if
    elif 1/2*p < n then
        if n = p-1 then return p-1
        elif p < 2147483648 then r := Facmodp(p-n-1,p)
        else r := Facmodp(p-n-1,p-4294967296)
        end if;
        return `mod`((-1)^(p-n)/`if`(r <
0,4294967296+r,r),p)
    else r:=Facmodp(`if`(n<2147483648,n,n-
4294967296),`if`(p<2147483648,p,p-4294967296))
    end if;
    `if`(r < 0,4294967296+r,r)
end proc;
facmodm := proc (n::nonnegint, p::posint)
local r;
    if p <= n then return 0
    elif p > 4294967295 then
        if p > 562949953421312 then WARNING("result may be
wrong") end if;
        return `mod`(round(Fmodm(n,p)),p)
    else
    r:=Facmodm(`if`(n<2147483648,n,n-
```

```
4294967296),`if`(p<2147483648,p,p-4294967296));
    end if;
    `if`(r < 0,4294967296+r,r)
end proc;
facmod := proc (n::nonnegint, p::posint)
    if isprime(p) then facmodp(n,p)
    else facmodm(n,p)
    end if
end proc
end module:
```

Conclusion

As we saw here, the interaction between Maple and assembly language can be very fruitful. A DLL written in assembler provides speed unavailable in Maple or DLL written in high level languages. Maple, from another point of view, adds its symbolic capabilities making the calculations much faster by using division modulo *p* and isprime function in this example. The future improvements could be made by replacing the line "if p <= n then return 0" in the facmod procedure by including more sophisticated conditions guaranteed that *n*! = 0 mod *p* for composite *p*. For a specific processor, the assembler code could be further optimized. Other improvements could be achieved by using the gmp library supplied with Maple 9. It would be interesting to find a precise bound *M* such that all Fmodp1(*n*, *p*) give the correct answers for positive integers *n* and *p* less than *M*.

This article started from my posting [1]. The MASM32 author's website [2] contains useful assembler references. Dr. Agner Fog wrote a great optimization manual [3] and periodically updates it. The Intel developer's manual [4] contains a description of the assembler instructions.

I would like to thank my beautiful and wonderful wife, Bette for her support.

References

1. Alec Mihailovs, *Huge factorials*, comp.soft-sys.math.maple, 12/4/2003, http://groups.google.com/groups?threadm=oHNzb.7942%24Zu5.1208563%40news3.news.adelphia.net
2. S.L.Hutchesson, *MASM32*, http://www.movsd.com/
3. Agner Fog, *How to optimize for the Pentium family of microprocessors*, http://www.agner.org/assem/#optimize
4. *IA-32 Intel® Architecture Software Developers' Manual, v.2: Instruction Set Reference*, http://developer.intel.com/design/Pentium4/manuals/