# Lab 4: Double-word multiplication

## Objectives

- to become more familiar with assembly language for the Intel architecture

- to see how to work with numbers larger than a machine handles

## Overview

In class, we looked at a program for computing factorials. Unfortunately, because it deals with 32-bit numbers, it can compute factorials only up to $13!$ — beyond that, the factorials exceed the maximum 32-bit two's-complement integer. In this lab, we'll extend the program to work with 64-bit numbers. This will extend its capability to compute as much as $20!$.

The program uses the mull instruction, which performs unsigned multiplication on the 32-bit numbers found in eax and the location named in the argument, placing the upper 32 bits of the product in edx and the lower 32 bits in eax. (This is just like the imull instruction of Section 5.2.1, except that mull is intended for unsigned numbers.)

Unfortunately, you will want to multiply a 64-bit number $A$ by a 32-bit number $B$, whereas mull can multiply two numbers of only 32 bits. To compute the product of $A$ and $B$, the following fact is useful: Suppose we split $A$ into two pieces: $A_u$ represents the upper 32 bits of $A$ and $A_\ell$ represents the lower 32 bits. We also use $(A_u B)_\ell$ to represent the lower 32 bits of the product of $A_u$ and $B$. Using this notation we can view the multiplication as the following.

$$
\begin{array}{r r}
A_u & A_\ell \\
\times & B \\
\hline
(A_u B)_\ell + (A_\ell B)_u & (A_\ell B)_\ell
\end{array}
$$

That is, the upper 32 bits of the product of $A$ and $B$ is $(A_u B)_\ell + (A_\ell B)_u$, while the lower 32 bits of the product is $(A_\ell B)_\ell$.[1]

For an example, suppose each register holds 8 bits (not 32), and $A$ were $10\ 11111111_{(2)}$ and $B$ were $10_{(2)}$. Then we could perform the multiplication as follows.

$$
\begin{array}{r r l}
10 & 11111111 & (A_u\ A_\ell) \\
\times & 10 & (B) \\
\hline
1 & 11111110 & (A_\ell \times B) \\
0\quad 100 & & (A_u \times B) \\
\hline
101 & 11111110 & (A \times B)
\end{array}
$$

## Setup

Run "`getcs 210 4`" to get the file `fact.s`, which contains the fact subroutine that you will modify for this lab. You cannot assemble this on one of the Suns (which have SPARC processors). But you can assemble it using a Linux computer named `chomas`, which you can reach from the terminal window with the *ssh* command.[2]

```
sunX% xhost +chomas
sunX% ssh chomas
```

---

[1] Actually, $(A_u B)_\ell + (A_\ell B)_u$ represents the *middle* 32 bits, while $(A_u B)_u$ would be the top 32 bits. But we'll ignore these top 32 bits, since they lie beyond the 64-bit product we're trying to compute.

[2] If `chomas` is very busy, you might choose to use `monte` instead.

Once you're logged into a Linux computer, you can *cd* to the proper directory and begin the editor using the following commands, where sun$X$ is the name of the computer at which you are seated.[3]

```
chomas% setenv DISPLAY sunX:0.0
chomas% nedit fact.s &
```

You can assemble `fact.s` using *gcc* and execute the resulting executable placed in `a.out`.

```
chomas% gcc fact.s test.s
chomas% ./a.out
ebx 401051ec   esi 4000ae60   edi bffff8a4
Number? 4
4! = 24
edx 00000000   eax 00000018
ebx 401051ec   esi 4000ae60   edi bffff8a4
```

## Programming details

You should modify fact so that it returns the lower 32 bits of the factorial in eax and the upper 32 bits in edx. (The parameter, n, will still be a 32-bit quantity.) Note that main has already been modified to handle such a 64-bit return value from fact.

If your modified subroutine modifies any callee-save registers, it must restore them before returning. The output of `a.out` includes printouts of the values of the callee-save registers before and after calling fact; you can examine this data to verify that your code preserves these registers' values. **Note:** Your modifications may not use statically located variables — any additional memory use should be through the stack.

Included in `test.s` are two subroutines that you may find useful in debugging your program: printregs displays the current values of all registers, and printstack displays the current values in the stack, beginning with the stack's top. Neither of these subroutines take any arguments, and both preserve all register values. This conveniently allows you to place a call to either subroutine wherever you want in the program. (Actually, they destroy the eflags register, so don't try to call them between a cmpl and a jge instruction, for example.)

Your lab report need only contain an introduction, your modified fact subroutine, and a conclusion. Run "`handincs 210 4`" to submit your code electronically.

---

[3]I don't recommend editing the file directly on sun$X$. Files stored on the server are cached on individual computers. If you edit the file on sun$X$ while assembling and editing it chomas, you will end up in situations where you have saved the file in sun$X$'s cache, but chomas' cache still contains the old version, and so you assemble the version before it was saved. This causes many headaches.