Entrega final práctica 1
Organización de Computadores

Sebastián Arcila Valenzuela (*sarcilav@eafit.edu.co*),
Ruben Dario Bueno Angel (*rbuenoan@eafit.edu.co*) y Sergio Botero Uribe (*sbotero2@eafit.edu.co*).

***Resumen*—Entrega final de la práctica 2 de Organización de computadores, el informe contiene los detalles de la construcción del procesador así como las especificaciones de la solución planteada, las dificultades encontradas y los comentarios por cada uno de los integrantes del equipo sobre la práctica.**

I. Introducción

"In C++ and Java I experience a certain amount of angst when you ask how to do this and they say, "Well, you do it like this or you could do it like that."There are obviously too many features if you can do something that many ways-and they are more or less equivalent. I think there are smaller concepts that fit better in Inferno."[**?**]

II. Detalles de diseño de construcción

Podemos decir que un algoritmo trivial para atacar las series de taylor seria el siguiente, tomemos como ejemplo de aquí en adelante la función seno:

```
float sen(int x, int n)
//n numero de iteraciones
{
 float xx = to_radianes(x);
 float ans = 0;
 for(int i = 0; i<n;++i)
 {
  ans += pow(-1,i)*pow(xx,2i+1)/(2i+1)!;
 }
 return ans;
}
```

Pero como podemos notar hay varias acotaciones que hacer a este código, primero no es necesario tener una función pow, ni una función factorial, puesto que cuando salgo de la iteración $m, (m > 0)$ y paso a la iteración $m+1$, tenemos ya calculado hasta xx^{2m+1} y el valor que necesitamos es xx^{2m+3}, entonces bastaría con solo multiplicar por xx^2 a xx^{2m+1} para obtener el valor en $m+1$, casi de igual manera para el factorial cuando entramos en la iteración $m+1$ ya tenemos previamente el valor de $(2m+1)!$, y si nos fijamos el valor que necesitamos es $(2m+3)!$ que es igual a $(2m+1)!(2m+2)(2m+3)$, y para la situación del signo simplemente es en cada iteración hacer $signo = \neg signo$, y voila! no necesitamos funciones externas que desperdicien cálculos y tiempo; obteniendo algo similar a esto:

```
float seno(int x, int n)
{
 float xx=to_radianes(x);
 float ans=0;
 float factorial=1;
 float acum_x2n=xx;
 int sign = -1;
 for(int i = 1; i<n;++i)
 {
   sign *= -1;
   ans += sign*acum_x2n/factorial;
   acum_x2n*=xx*xx;
   factorial*=(2*i)*(2*i+1);
 }
 return ans;
}
```

También vale la pena resaltar que los tipos de datos de factorial y de acum_x2n es de tipo flotante, por que por obvias razones en números enteros da overflow para factorial y ademas los radianes (xx) que vamos a usar tienen precisión flotante.

Si lo notamos bien de la misma manera podemos deducir el siguiente algoritmo para coseno:

```
float coseno(int x,int n)
{
 float xx=to_radianes(x);
 float ans=0;
 float factorial=1;
 float acum_x2n=1;
 int sign = -1;
 for(int i = 1; i<n;++i)
 {
  sign *= -1;
  ans += sign*acum_x2n/factorial;
  acum_x2n*=xx*xx;
  factorial*=(2*i-1)*(2*i);
 }
 return ans;
}
```

Ahora bien la operación de pasar de grados a radianes es lo más simple de este mundo, es simplemente multiplicar los grados por el factor de conversión que es $\frac{\pi}{180} = 0{,}017453293$. Y estás serian todas la anotaciones respecto al diseño del algoritmo.

III. Especificaciones del módulo

La práctica fue desarrollado para GNU/Linux y para Mac, la practica en GNU/Linux fue desarrollado en ubuntu 9.04 y en Mac OS X.5

A. GNU/Linux

Usamos nasm(NASM version 2.05.01 compiled on Nov 5 2008) para ensamblar nuestro código y generar el código objeto que se linkearia como una Dynamically linked shared object libraries (.so) con gcc (gcc versión 4.3.3 (Ubuntu 4.3.3-5ubuntu4)). Para ensamblar:

```
nasm -f elf -o trigo.o trigo.s
```

Para generar el .so:

```
gcc -shared trigo.o -o libtrigo.so
```

Después de esto libtrigo.so debe ser llevado a /usr/lib y ya simplemente para compilar algo que haga uso de nuestra librería con gcc, solo necesitamos colocar -ltrigo en los flags de compilación y agregar trigo.h al directo de los include de la distribución en particular, en nuestro caso /usr/include, y agregar la cabecera $\#include < trigo.h >$ en el código en particular.

Las dependencias necesarias son: libglade2-dev libgtk2.0-bin.

Dependencias para construir desde los fuentes: libglade2-dev libgtk2.0-bin gcc make nasm

Para más detalles mirar la aplicación piloto que se puede encontrar en `http://code.assembla.com/trigo_assembla/subversion/nodes/branches/linux/app`, mirar el Makefile y trigonometria.c

B. Mac OS X.5

Para usar el código de las funciones trigonométricas sobre Mac OS X Leopard, se debe primero generar un archivo objeto que tiene por formato Mach-O, un equivalente al ELF en Linux, en el cual aparecen todas librerías y programas. El objeto Mach-O puede tener código que puede ser 'linkeado' dinámica y estáticamente.

Este objeto con los símbolos (en este caso las funciones) se crea con el ensamblador de NASM de la siguiente forma:

```
$ nasm -f macho -o nasm.o nasm.s
```

Después del cual obtenemos el objeto Mach-O que podremos usar en nuestro proyecto. El trabajo en OS X para probar la librería se dio en dos casos diferentes, el primero para probarla con una aplicación de línea de comando en Objective-C y la segunda segunda en una aplicación con interfaz gráfica usando Objective-C y Cocoa en donde para los dos cosos desde Objective-C las funciones de la librería se invocan de la misma forma que en C/C++, usando:

```
extern float seno();
extern float coseno();
extern float tangente();
```

en los encabezados. Para el caso de la aplicación en línea de comando la forma más efectiva para obtener el ejecutable es compilando el proyecto junto con el archivo objeto generado por NASM de la siguiente forma.

```
$ gcc Proyecto.m nasm.o -o ejecutable \
      -framework Foundation
```

Para luego simplemente ejecutarlo. Cuando se trabaja en Xcode, basta con importar el archivo objeto al proyecto y trabajar de la misma forma que se haría con la aplicación de línea de comando. Existe la posibilidad de compilar y ejecutar el proyecto de Xcode mediante línea de comandos de la siguiente forma:

```
$ xcodebuild clean build install -activetarget \
      -activeconfiguration
```

pero de todas formas se necesita importar el objeto generado por NASM desde Xcode.

NOTA: El archivo 'trigo.s' en Linux y 'nasm_dll.s' en OS X tienen una pequeña diferencia, ya que el ensamblador NASM de Linux permite que para las operaciones de flotantes se deje implícito el campo de parámetros para indicar que se está trabajando con las primeras direcciones de la pila, cosa que se debe dejar explícita en OS X, esa es la mayor diferencia en cuanto al código en ensamblador que se debe tener en cuenta para obtener la portabilidad de esta librería entre los dos sistemas operativos.

IV. Dificultades encontradas durante la creación del procesador

La principal dificultad que encontré para realizar la práctica, fue que en un principio cuando empezamos a trabajar con FASM, no tenía forma de ensamblar el código que escribiamos en mi equipo, por lo que los avances se daban muy lentamente al tener que probar el código siempre en otra parte. Después de la decisión de cambiar FASM, los inconvenientes estaban en pequeñas diferencias en sintaxis que variaban entre Linux y Mac, las cuales siempre que se quería ensamblar había que acomodar.

-Sergio Botero Uribe

El mayor dificultad fue escoger el ensamblador correcto. Al principio intentamos mucho con fasm pero en vano, ponía mucho problema en el uso de las cabeceras y en la generación de código objeto sin mencionar que los errores que encontraba y devolvía eran casi incomprensibles. Luego cuando tomamos la decisión de ensayar Gnu AS, funciona a las mil maravillas pero en sintaxis AT&T, que a mi parecer es horrible, y aun asi usando el flag .intel_syntax, no funcionaba nuestro programa escrito en asembler. Pero en realidad todo empezó a funcionar con Netwide Assembler la sintaxis es intel por defecto y aparte de eso para ensamblar permite usar flags para distinguir el tipo de objeto que se debe hacer(-elf -macho).

-Sebastián Arcila Valenzuela

V. Opiniones sobre la práctica

Este tipo de programación en la que se le debe decir cada cosa al procesador, me parece que debió aparecer antes en la carrera, es mejor en cierta forma irse acomodando a los nuevos conceptos y a las nuevas formas de programación después de aprender lo que está más abajo. La práctica no solo fue en lenguaje ensamblador, para poder hacer los aportes y completarla tuve que investigar cosas que a la

final resultaron no tan claras como creía. Un aspecto en el que quiero hacer mucho énfasis y que me pareció una idea que ojalá se repita para las próximas prácticas es la posibilidad y flexibilidad que se da para poder elegir la plataforma para hacer el trabajo, los incentivos son muy válidos pero más que eso es comprender que muchos estudiantes pueden sacar mucho más que el conocimiento y las habilidades que se espera obtengan del trabajo si tienen la posibilidad de trabajarla en donde les interesa hacer desarrollos, que para este caso particular, es desarrollo en Linux y por los lados en Mac. Puedo decir que esta práctica me ha incentivado a leer y aprender más cosas sobre la forma en que se trabaja más cerca del procesador en Mac.

-Sergio Botero Uribe

Este es el tipo de prácticas con las que uno aprende, pues nos toco que investigar y leer mucho por cuenta propia, por ejemplo las refencias para desarrolladores de intel, como generar librería en Unix, como linkear, como generar código objeto, como pegarle desde C a una librería hecha por nosotros. Además al programar en asembler me recuerda que lo bello es lo simple. En general puedo decir que me sentí programando como un hombre de verdad.

También quiero anotar que lo que mas me gusto es que no nos sesgaran a trabajar bajo una única plataforma, sino que nos insinuaron a probar en cosas diferentes mediante estímulos, aunque valederos, no enteramente necesarios para los que nos gusta aprender y hacer las cosas desde el "lado de la luz" (*nix).

-Sebastián Arcila Valenzuela