Organización de Computadoras - TP 1

Martín Cohen, Padron: 100812 martincohen98@gmail.com

1er Cuatrimestre 2020

Resumen

En este trabajo practico se programaron dos versiones del juego de la Vida, una de ellas escrita en C y otra con una función en Assembly, con el objetivo de familiarizarse con la forma de escribir en el Assembly de MIPS, y las herramientas y lógica de escribir en Assembly.

1 Introducción

El trabajo practico tiene como objetivo lograr familiarizarse con la programación en el lenguaje Assembly de MIPS32. Esto incluye el uso y sintaxis de sus instrucciones y conceptos como el la Call Convention y su ABI.

Para esto es necesario hacer una implementación de el Juego de la Vida de Conway. Se pide hacer una implementación totalmente portable en C, y otra con una de las funciones claves del proyecto implementada en Assembly de MIPS32. Este juego modela una progresión en las que una matriz de celdas con organismos vivos se van ocupando y desocupando dependiendo de su entorno. A esto se le agrega la dificultad de que si se llega al borde de la matriz se considera adyacente el bloque del borde opuesto que está a la misma altura.

2 Diseño e Implementación

2.1 Análisis de la Linea de Comandos

En la linea de comandos los parametros deben ser enviados de la siguiente manera:

```
./conway i M N inputfile
```

Siendo i la cantidad de iteraciones, M y N el tamaño de la matriz MxN a utilizarse, e inputfile el archivo donde se detallan las cordenadas de los puntos iniciales del programa.

Los flags que pueden utilizarse son:

```
-h, --help Imprime este mensaje.
-V, --version Da la versión del programa.
-o Prefijo de los archivos de salida.
```

2.2 Desarrollo del Código Fuente

La implementación portable y la implementación de MIPS32 comparten una gran parte de su código. De hecho, todo menos la función vecinos, su declaración y el mensaje del flag -v son exactamente iguales. Debido a esto hay **dos** archivos conway.c, uno para la version C y uno para la version MIPS32, y un archivo vecinos.S.

Cualquier operación exitosa es notificada al usuario a través del standard output. Esto tiene el propósito de que en caso de que haya un error y se notifique por el standard error se pueda confirmar que no hay otros problemas con los otros parámetros.

Asumo que los nombres de los archivos dados por el flag -o no van a ser extremadamente altos, ya que uso un buffer de tamaño fijo (100 caracteres) para el nombre.

Ambas implementaciones tienen una visualización por consola al cargar la matriz del archivo y luego de cada iteración. Esto sirve para que no sea necesario abrir los archivos para poder ver el estado del juego.

En la implementación de la función vecinos podría haber optado en usar un loop doble que recorriera los casilleros rodeando al cual se la llama la función. Sin embargo, tomando en cuenta la especificación de los bordes, una solución más simple es inicialmente fijar las filas y columnas adyacentes al casillero y recorrerlos de manera estructurada en vez de iterativa. Esto resulta en código más largo pero menos complejo.

En la versión de Assembly de esa misma función, opte por usar una variable en memoria menos. Como en la versión de C solo la usaba para acumular la cantidad de vecinos utilicé un registro temporal en vez de alocar memoria. Esto mejora el rendimiento ya que se necesitan menos accesos a memoria sin aumentar la complejidad del código.

2.2.1 ABI

El ABI del lenguaje Assembly de MIPS32 está dividido en tres secciones. Estas son:

- El SRA (Saved Register Area) se usa para guardar registros cuyo valor debe ser devuelto al original al retornar de la función.
- El LTA (Local and Temporary Area) se utiliza para guardar variables locales que se utilizan en la función.
- El ABA (Argument Building Area) se utiliza para guardar los argumentos de la función. Si hay un argumento, es necesario dejar espacio suficiente como para que haya cuatro argumentos. Se agrega más espacio de ser necesario.

Estas tres secciones deben tener un tamaño múltiplo de 8 bytes. De lo contrario se debe agregar padding para compensar.

Dicho esto, el stack frame de la función vecinos en Assembly Esta formado de la siguiente mantera:

	padding ancho alto j i matriz
	gp fp
	columnaSiguiente columnaAnterior filaSiguiente columnaSiguiente

El rectángulo superior es el ABA a donde van los argumentos de la función "vecinos". Los rectángulos que le siguen son el SRA y LTA respectivamente.

Notar como en el ABA se dejan 4 bytes de padding como se mencionó anteriormente.

3 Proceso de Compilación

El proceso de compilación es dentro de todo simple. Para la versión en C, lo único que se debe hacer es llamar a gcc con el archivo conway.c de la version C de la siguiente manera:

```
gcc conway.c -o conway
```

Con esto ya se tiene el ejecutable listo para correr el programa.

La versión de MIPS32 es un poco mas compleja en principio, pero esencialmente es similar. Al tener una función programada en Assembly especifico de MIPS32 no se puede simplemente compilar en un linux corriendo en una computadora con arquitectura diferente. Para esto se necesita correr una maquina virtual corriendo una distribucion de linux de MIPS32 y compilarlo en la misma.

Una vez que ya tenemos esto la compilación vuelve a ser trivial. Se realiza de la siguiente manera, utilizando la versión MIPS32 de conway.c.

```
gcc conway.c vecinos.S -o conway
```

Es importante notar que para ser debuggeado el programa debe ser compilado de una manera distinta. Se le deben agregar los flags siguientes a ambas versiones.

```
gcc conway.c vecinos.S -g -O0 -o conway
```

Estos flags mantienen la información de debuggeo al compilar y me aseguran que no se va a optimizar el código de ninguna manera respectivamente.

4 Casos de Prueba

En esta sección explico puntos importantes de las ejecuciones pedidas en el enunciado y ejecuciones importantes.

Se puede notar que la condición de borde se cumple en las siguientes fotos que son dos pasos consecutivos de la ejecución pedida en el enunciado de pento en ambas versiones del programa.

4.1 Manejo de Errores

En esta sección doy ejemplos de ejecuciones en las que el programa maneja varios tipos de errores. Los errores siempre se imprimen por stderr.

Si al programa se le presentan una cantidad incorrecta de parámetros este muestra un mensaje acorde al problema.

Si el archivo que se especifica no existe, el programa lo hace saber cuando falla la apertura del mismo.

Si cualquiera de los puntos se encuentra fuera del rango de la matriz se produce un error. Esto se puede reproducir fácilmente con la siguiente ejecución:

```
./conway 1 1 1 sapo
```

Como queda una matriz de tamaño 1x1 todos los puntos quedan fuera de rango.

Si el formato del archivo es incorrecto se produce un error y se indica el problema. Para eso se puede usar el archivo errorFormato adjuntado con el código.

5 Conclusiones

El trabajo practico tuvo más efecto de lo esperado. No solo sirvió para familiarizarse correctamente con el Assembly de MIPS, entendiendo la call convention y el ABI, sino que también tuvo una función de acercarse más a la programación de bajo nivel y la fuerte dependencia de esta al hardware. Me llevo a averiguar sobre compiladores tomando como ejemplo a gcc, el compilador utilizado en este trabajo.

Se puede notar que el lenguaje C tiene casi una traducción directa a Assembly, con la diferencia que el lenguaje C se puede escribir de la misma manera sin importar la arquitectura de la computadora en la que va a ser ejecutado. Una de las razones por las que se querría utilizar assembly en lugar de C es para mejorar la velocidad de ejecución del programa. Esto se debe a que en Assembly se tiene completo control sobre el código, ya que se traduce directamente a código de maquina. Esto suele ahorrar varias escrituras y lecturas de memoria acelerando fuertemente la ejecucion del programa.

El trabajo me llevo a descubrir que gcc en una computadora con una arquitectura no solo traduce el código C de manera distinta que en una computadora con otra arquitectura (cosa que es trivial si se sabe el el código Assembly es

completamente distinto), sino que no **puede** compilar el código para otra arquitectura. Llegue a esta conclusión luego de intentar encontrar una forma de compilar la versión de MIPS sin utilizar la maquina virtual para acelerar el proceso de debugeo, y fallando en el intenta. Esto significa que ambas versiones de gcc (las de x86 y MIPS32) tienen código para la traducción del código que la otra no tiene.

Una característica del ABI de MIPS que me pareció extraña es que las funciones deben guardar sus propios argumentos en memoria cuando se prepara el stack frame. No encuentro el propósito ya que los argumentos solo son útiles para la función misma y otras funciones no los necesitan. La única razón que se me ocurre es a la hora de hacer un backtrace durante un debuggeo o un crash-dump se pueda ver los parámetros con los que se llamo la función problemática.

6 Codigo

El codigo se encuentra en el siguiente link de github:
https://github.com/MartinCohen98/Organizacion-de-Computadoras-TP1

Para la versión MIPS32, las únicas diferencias son la declaración de vecinos que se le agrega la keyword extern:

```
extern unsigned int vecinos(unsigned char* matriz,
unsigned int i, unsigned int j, unsigned int alto,
unsigned int ancho);
```

Tambien se cambia el mensaje de versión para MIPS32 de la siguiente manera:

```
printf("Conway-1.0.1-MIPS32\n");
```

Para la implementación de vecinos en Assembly, no pude hacer funcionar macros que acortaban código, así que resulto en una función larga.