Trabajo Práctico 1: Conjunto de Instrucciones MIPS

José Ignacio Sbruzzi, *Padrón Nro. 97.452*
jose_sbruzzi@hotmail.com
Leandro Huemul Desuque, *Padrón Nro. 95.836*
desuqueleandro@gmail.com

2do. Cuatrimestre de 2016
66.20 Organización de Computadoras
Facultad de Ingeniería, Universidad de Buenos Aires

Resumen

Se desarrolló un programa en C que simula el Juego de la Vida de Conway, habiéndose implementado una parte específica en Assembly MIPS. El objetivo del presente trabajo fue además familiarizarse con el entorno GXEmul y con LaTeX.

1. Introducción

Inicialmente, se implementó en C el Juego de la Vida de Conway. El Juego de la Vida es un autómata celular descripto por John Conway en 1970 **??**. Inicialmente fue publicado como un juego matemático recreacional. El juego es "de cero jugadores.[en] el sentido de que no es necesario propiamente jugar, sino que lo único que hace el jugador es decidir la configuración inicial y observar cómo evoluciona el sistema. Se rige por reglas sencillas: las células nacen, sobreviven o mueren dependiendo de la cantidad de vecinos vivos que tienen, tal como se describe en el enunciado de este trabajo práctico. Así, es un sistema simple, y una buena elección si el objetivo final no es construir un sistema elaborado sino familiarizarse con el software necesario para ello.

Posteriormente, se reprogramó una parte del programa en Assembly MIPS, para esto se utilizó el emulador GXEmul. GXEmul emula arquitecturas de computadoras y permite correr sistemas operativos sobre ellos **??**. De esta manera, se compiló y se corrió el programa en Assembly MIPS pese a no disponer de un equipo MIPS.

2. Desarrollo

2.1. Documentación del código C

La documentacion de las funciones se detalla por orden de aparición en el código fuente.

2.1.1. `help`

`help` despliega la ayuda para el usuario final.

2.1.2. `version`

`version` informa la version del código fuente.

2.1.3. `getPosicion`

`getPosicion` calcula y devuelve la posición relativa de una posicion (i,j) en base al tamaño de la matriz.

2.1.4. `cargarMatriz`

`cargarMatriz` es una función que, en base a la matriz inicial y una posicion específica (que se corresponde a la leída por el archivo de entrada) asigna dicha posición como ENCENDIDA (o viva) dentro de la matriz.

2.1.5. `infoValida`

`infoValida` se encarga de validar que una coordenada no sea mayor al tamaño de la matriz.

2.1.6. `procesarArchivo`

`procesarArchivo` se encarga de leer el archivo de entrada y llamar a las respectivas funciones para dejar la matriz en un estado válido.

2.1.7. `inicializarMatriz`

`inicializarMatriz` se encarga de asignar la memoria de la matriz y dejar todas las posiciones como APAGADAS (o muertas).

2.1.8. `siguienteMatriz`

`siguienteMatriz` genera, a partir de una matriz (es decir, un estado), la matriz que le sigue según las reglas del juego.

2.1.9. `grabarEstado`

`grabarEstado` genera la imagen de salida en formato plain PBM.

2.1.10. `liberarRecursos`

`liberarRecursos` es una función que se encarga de liberar la memoria asignada para contener la matriz.

2.1.11. `avanzarEstados`

`avanzarEstados` es una función encargada de llamar a grabarEstado y liberar los recursos a medida que las imagenes de salida son generadas.

2.1.12. `vecinos`

`vecinos` es una función que, a partir de la matriz y de una celda, determina cuántos vecinos vivos (es decir, casilleros ocupados) tiene una celda. De esta manera, la función `siguienteMatriz` puede definir dónde suceden nacimientos, muertes y supervivencias.

2.2. Documentación del código Assembly MIPS

La función específica que se desarrolló en Assembly MIPS es la llamada "vecinos", o bien "vecinos_s"(nombre utilizado en el código fuente para hacer referencia a la versión MIPS).

Esta función implementa la búsqueda de vecinos ENCENDIDOS para una determinada posición (i,j). Se utiliza un contador para llevar cuenta de la cantidad de vecinos vivos encontrados dentro de los 8 posibles (que representaría el caso límite de todos los vecinos de una determinada posición vivos).

Para esto, la función recibe por parámetro la dirección de memoria de inicio de la matriz, la posicion (i,j) a procesar y el tamaño final de la matriz (ancho y alto).

A diferencia de la versión de C, la cual calcula la posición en memoria en base a la posición relativa (i,j) utilizando la función "getPosicion", en la implementación MIPS se optó por desarrollar dicha funcionalidad dentro de la misma funcion "vecinos".

2.2.1. `Especificaciones`

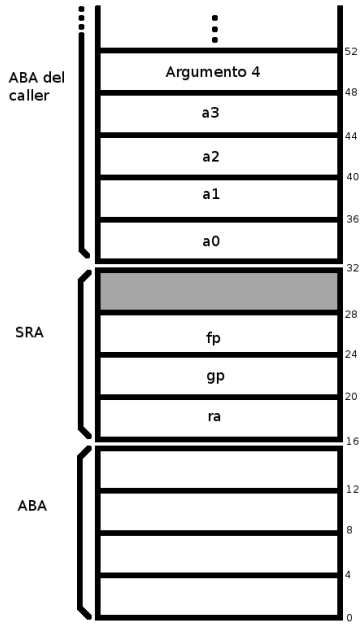

Figura 1: Stack de la función vecinos

Se reservaron 12 bytes para el área de SRA, 4 bytes para salvar el registro *ra* (no era necesario ya que en ningun momento los saltos para realizar comparaciones modifican el ra, de cualquier forma y como buena práctica, se guardó su valor), 4 bytes para salvar el registro *gp* y 4 bytes para salvar el registro *fp*.

Al tratarse de una matriz cuadrada (MxM), el cálculo de la posición relativa se hizo utilizando solo el ancho. Se pasó el alto para cumplir con la firma pedida por el trabajo práctico y para contemplar posibles mejoras u optimizaciones a futuro (por ejemplo, que se permitan matrices de MxN). Se reservaron 16 bytes en concepto de ABA, respetando ABI. **??**.

2.3. Dificultades

La construcción de una imagen en formato PBM presentó algunas dificultades ya que se usó PBM en vez de plain PBM, y en la descripción del formato PBM no había ejemplos de archivos PBM (sí los había de plain PBM). Un punto relativamente problemático del formato PBM es que cada pixel está asociado a un bit. Para evitar esta dificultad, se decidió que cada celda correspondería a un cuadrado de 8x8 pixeles, simplificando el fragmento del programa que guarda las imágenes.

3. Compilación

En lugar de utilizar un *makefile* se optó por programar un script (*hacer.sh*) para dicha tarea, que cumple el mismo objetivo: ejecutar las instrucciones adecuadas de compilacion.

Se compilan de manera individual cada uno de los archivos fuente de extension *c* y *S* a traves del ejecutable *gcc*.

Los argumentos utilizados para la compilación son los siguientes:

-c Compila el código fuente pero no corre el linker. Genera el código objeto.

-o Especifica el archivo de salida (ya sea un archivo objeto, ejecutable, ensamblado).

-Wall Activa los mensajes de warning.

-I Agrega el directorio especificado a la lista de directorios buscados para los archivos header

4. Resultados

4.1. Medidas de tiempo

Se analizarán los tiempos de ejecución tanto para la versión completa en C (sin optimizaciones del compilador), como para la versión MIPS. Se tomarán distintos casos de prueba para visualiar el rendimiento de cada una.

Todas las pruebas se realizaron utilizando los archivos propuestos por la cátedra sobre un entorno Ubuntu (para la versión en C) y sobre el emulador GXemul (para la version MIPS).

Se obtuvieron los siguientes resultados (el tiempo informado es el tiempo usr + real).

En todas las pruebas, las matrices se establecieron con las dimensiones del caso de ejemplo: 20x20.

Archivo	Pasos	Tiempo de ejecución(C)	Tiempo de ejecución (MIPS)
Glider	10	0,022s	2,248s
Glider	100	0,124s	14,769s
Glider	1000	1,620s	157,485s
Archivo	Pasos	Tiempo de ejecución(C)	Tiempo de ejecución (MIPS)
Pento	10	0,013s	1,43s
Pento	100	0,089s	14,983s
Pento	1000	1,629s	163,15s
Archivo	Pasos	Tiempo de ejecución(C)	Tiempo de ejecución (MIPS)
Sapo	10	0,052s	1,092s
Sapo	100	0,253s	18,252s
Sapo	1000	1,904s	172,32s

Cuadro 1: Tiempos medidos

4.2. Corridas de prueba

A continuación se detalla el resultado de las corridas de prueba de glider, pento y sapo para 10 operaciones en una matriz de 20 por 20 tal como fue pedido.

4.2.1. Glider

Glider es una configuración del autómata que se mueve por la pantalla sin destruirse: atravieza un ciclo de cuatro estados, al cabo de los cuales regresa a la configuración inicial, pero desplazada. En la figura **??** se muestra la configuración inicial del glider junto con 10 iteraciones. El glider se mueve, en cada ciclo, una celda hacia abajo y una celda a la derecha, como puede notarse en la figura **??**.

4.2.2. Pento

Pento es un patrón que se estabiliza luego de cierta cantidad de iteraciones. El resultado final varía según el tamaño de la matriz. Para una matriz de 20 por 20, se muestra la configuración inicial, las primeras 10 iteraciones y el resultado final estable, alcanzado en la iteración 60, en la figura **??**. En la imagen **??** puede verse la configuración final de Pento para una matriz de otro tamaño.

4.2.3. Sapo

El patrón Sapo, al igual que Glider, atravieza un ciclo, pero de 2 estados (no de 4), y no se desplaza. En la figura **??** se muestran el estado inicial y cuatro iteraciones de Sapo. No se muestran más iteraciones porque los estados se repiten constantemente.

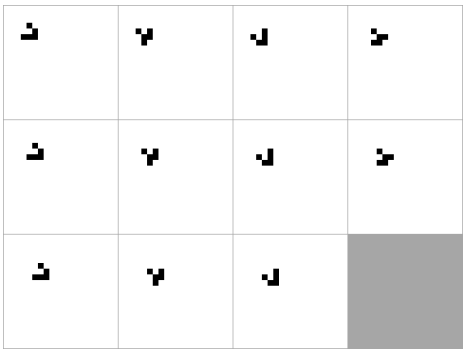

Figura 2: Desde la esquina izquierda superior: 1:Estado inicial descripto en el archivo glider. 2 a 11: Iteraciones 1 a 10

Figura 3: El movimiento del glider: se superponen la iteración 4 y 8

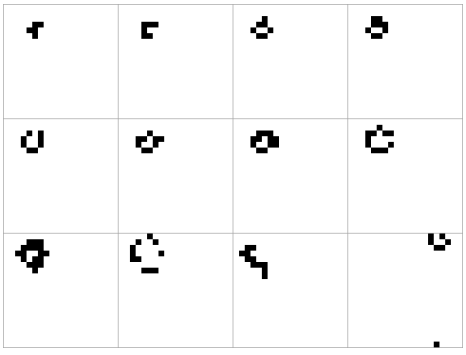

Figura 4: Desde la esquina superior izquierda: 1: estado inicial de Pento, 2 a 11: primeras 10 iteraciones, 12: estado final luego de 60 iteraciones. El tamaño de la matriz es 20x20

Figura 5: La configuración final de Pento en una matriz de 30 por 30.

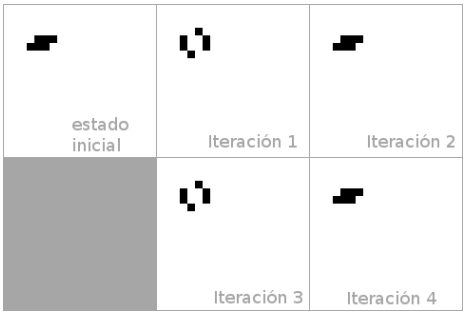

Figura 6: La configuración final de Pento en una matriz de 30 por 30.

5. Conclusiones

Analizando los resultados obtenidos se puede observar que la implementación completa en C es varios órdenes de mágnitud más eficiente que la versión implementada con una porción en MIPS, aun cuando no se activaron las optimizaciones del compilador. Esta diferencia de tiempo y performance, se hace más evidente a medida que incrementamos la cantidad de pasos, por ende, de cálculos e imágenes de salida.

Aun cuando la función a implementar era sencilla, llevar su codificación a MIPS conlleva un trabajo extra tanto de aprendizaje como de pruebas, dado que no es simple encontrar errores en caso de que no se llegue al resultado esperado.

Por último, no debemos descartar que la porción del programa total desarrollada en MIPS es pequeña en relación a la desarrollada en C y que se está utilizando un emulador para correr las instrucciones. Esto último, sumado a que se está trabajando con matrices y archivos en todo momento, son factores que no se tuvieron en cuenta en los tiempos finales calculados y sin embargo son de importancia crítica si se quisiera realizar un estudio exhaustivo de rendimiento.

Se puede concluir que, pese a no ser más eficiente la versión en MIPS, es una alternativa interesante cuando se quiere desarrollar software para un sistema específico. Por otra parte, al programar en MIPS se pueden conocer los tiempos precisos de ejecución de nuestros programas, sin depender de las modificaciones posteriores realizadas de manera automática por el compilador, esto puede resultarnos de suma importancia en aplicaciones en los que el tiempo de ejecución es un factor clave.

Referencias

[1] Gardner, Martin. “Mathematical Games - The fantastic combinations of John Conway’s new solitaire game ’life‘ ” Scientific America, 223. pp. 120-123. ISBN 0-89454-001-7. Archivado del original en: https://ddi.cs.uni-potsdam.de/HyFISCH/Produzieren/lis_projekt/proj_gamelife/ConwayScientificAmerican.htm

Consultado en septiembre 2016.

[2] Sitio web de GXemul http://gxemul.sourceforge.net/

Consultado en septiembre 2016.