Alumno: Zingarelli Facundo
Nro. Alumno: 66092/1

FACULTAD DE INGENIERÍA

Algoritmo de ordenamiento en AVR

Mención sobre ejercicio 9:
En el archivo entregado se presentan dos archivos, uno correspondiente a la implementación mediante una rutina bloqueante, y el otro mediante una rutina no-bloqueante, indicado en sus nombres

Mención sobre ejercicio 10:
En el ejercicio 10 se puede observar que en el main se desactiva tanto el timer como la interrupción por el mismo, esto se debe a que de no desactivar ambos el programa no se ejecutaba correctamente, a pesar de que debería bastar con desactivar el timer. Busqué en foros el por qué de esto pero no pude encontrar una respuesta.

Introducción:

En este trabajo se implementó el algoritmo de ordenamiento utilizado en el entregable anterior: Ordenamiento por selección. Pero esta vez se implementó en el microcontrolador Atmega328p de AVR, cuya diferencia principal reside en el ISA y en la arquitectura.

Desarrollo:

Para la implementación de lo requerido en el enunciado, se implementaron dos rutinas autocontenidas: una que genera los números aleatorios y los carga en la posición del vector dada (primer elemento en 0x100), y otra que los ordena.

El algoritmo de ordenamiento está diseñado para ordenar enteros en Ca2, es decir tanto positivos como negativos. Si bien el algoritmo funciona correctamente, tuve problemas a la hora de generar los números aleatorios como se explica a continuación.

Rutina de generación de números aleatorios:

Para generar los números aleatorios, se intentó abordar el problema por distintos enfoques, primero se pensó que la forma más sencilla de lograrlo sería a través de hardware. Es decir, conectando por ejemplo un inductor toroidal que haga de

antena o receptor, y con la entrada correctamente seteada y pull-up activo, uno podría modelar el ruido medido con una densidad espectral como la del ruido blanco. Si bien la implementación es muy sencilla, me topé con el problema de no saber qué es lo que efectivamente se está guardando en los registros. Se intentó configurar el DataVisualizer para poder mostrar en pantalla los datos leídos, pero no se pudo lograr en assembly.

Dado el inconveniente mencionado, se optó por generar dichos números mediante software, pudiendo de esta manera verificar los números generados y corroborando el correcto funcionamiento de todo el programa. Para dicho fin se utilizó un algoritmo conocido como "**generador congruencial lineal**", que consiste en lo siguiente:

$$X(i+1) = (a.X(i) + b)\%m$$

Donde X(i+1) corresponde al número siguiente, y X(i) al previo.

Tanto "a", "b" como "m" son constantes, y por "%" nos referimos al resto de la división entera.

Este algoritmo permitirá generar una secuencia **pseudo-aleatoria** de números. La diferencia con un algoritmo o método aleatorio "puro", es que el pseudo-aleatorio generará una secuencia **periódica**, es decir, se repetirá cada cierta cantidad de números generados.

Investigando un poco sobre el tema, se pueden hallar algunas deducciones algebraicas sobre el rango o período de este algoritmo, de acuerdo tanto a "m", como a "a" y "b". El problema principal que tuve es que a la hora de diseñar el algoritmo que realiza la división por m, lo hice pensando en números de 8bits o 1 byte, y posteriormente descubrí que cuanto más grande pueda ser m, también lo será el rango o período de la secuencia generada. Esto se pudo apreciar claramente al implementarlo, ya que la secuencia generada se repite cada aproximadamente 29 o 30 números, que fue el rango más amplio que pude obtener para enteros de 1 byte. Me hubiera gustado poder corregir todo, y poder hacer tanto la multiplicación como la división con enteros de 2 bytes, pero no iba a llegar con el tiempo para poder terminarlo, en caso de requerirse se corregirá en una versión posterior.

Por otro lado, a este algoritmo se le provee normalmente un valor inicial (Xo), pero al corroborar empíricamente los resultados se encontró que se obtenían secuencias de período mayor si dicho valor era nulo. Por último, para que el algoritmo funcione correctamente se busca que los parámetros constantes sean primos, por eso los valores elegidos.

Respecto a la implementación:

Lo primero que se encuentra en esta rutina (Cada rutina está separada por líneas de "<" y con títulos como comentario), corresponde a las operaciones de

multiplicación y suma del valor anterior, y luego se copia ese valor a “dividendo” para la siguiente división.

Luego el código que sigue corresponde a un algoritmo de división típico basado en operaciones de desplazamiento a izquierda y algunas variables/operaciones auxiliares para corroboración del signo (División entera de 8bits con signo). Éste código de división se diseñó siguiendo los lineamientos brindados por Atmel en” AVR200: Multiply and Divide Routines”, adjuntado en bibliografía. La única modificación es la del comienzo ya explicada y al final, que se agregó un contador para evaluar si calculó los 256 elementos del vector, y operaciones con punteros para guardar el resto para la próxima iteración.

Rutina de ordenamiento:

No se realizará una introducción al algoritmo utilizado, ya que éste fue descrito en el entregable anterior, pero sí se realizarán aclaraciones sobre la implementación del mismo como del programa para el microcontrolador utilizado.

Para la implementación del algoritmo se utilizaron dos punteros, uno que recorre el vector (y), y otro que va apuntando al menor elemento de las comparaciones (x). A lo largo del diseño se pudo apreciar la diferencia en cuanto a la dificultad de implementación con respecto a MARIE, principalmente debido al repertorio considerablemente más amplio del Atmega, y sobretodo aquellas instrucciones de movimiento de datos entre registros.

El algoritmo de ordenamiento está separado en varios loops bastante similares al implementado en MARIE, el primero en el que se inicializan los punteros y contadores (rutina **autocontenida**), un loop interno que se encarga de hallar el menor de todos los elementos (**“menor”**), otro que almacena en “x” la dirección del elemento más chico (**“ymenor”**), y finalmente el loop externo que intercambia los elementos del vector situando al menor de la iteración en la posición correspondiente (**“switch”**). Se utilizaron dos contadores, uno interno y uno externo al igual que en MARIE.

El único “problema” que tuve con este algoritmo fue no poder utilizar la instrucción con post-incremento, que hubiera ahorrado unas cuantas instrucciones. Esto se debió a que la rutina tenía que ser autocontenida, de haber sido el main, podría haberse optimizado un poco más, como se hizo en MARIE.

Main o loop principal:

En esta sección del programa se configura el portB como salida, es decir, el LED, se llama a la rutina de generación primero y a la de ordenamiento después, se conmuta el LED al terminar, y se itera indefinidamente. Con ésta idea se implementó, pero en la práctica el LED parpadeaba a una velocidad que hacía imposible contar dichas conmutaciones, por ello se agregó un contador y un loop pequeño de control que genera la conmutación cada 10 secuencias de generación-ordenamiento, como se clarifica en la sección de ”perfomance”.

Perfomance:

Para la evaluación de la perfomance, se midió como se sugiere, utilizando un cronómetro y contando la cantidad de conmutaciones del led utilizado. En principio se diseñó el programa para que conmute una vez por cada secuencia de generación-ordenamiento, pero dada la velocidad era imposible contarlo. Para solucionar esto se agregó al main o programa una variable contadora (almacenada en r2), en la que por cada secuencia de generación-ordenamiento se decrementa y al llegar a 0 se realiza una conmutación. El valor utilizado de dicha variable fue 10, y se midieron:

- 109 conmutaciones en un minuto

Ahora, por cada secuencia el led **conmuta**, por lo que en realidad al contar cuándo enciende, estamos contando dos secuencias de generación-ordenamiento. Luego:

- $Nro.Secuencias = 109x10x2 = 2180$

Luego, el tiempo medio de generación-ordenamiento se obtiene:

$$t_{medio}[s] = \frac{60}{2180}[s] \cong 27{,}53ms$$

El tiempo total provisto y estimado por el compilador/simulador para el mismo programa fue de:

$$t_{simulador} = 26679{,}3\mu s \cong 26{,}8\, ms$$

Como se observa el tiempo es bastante preciso, entre el simulador y el medido hay un error de tan sólo $\sim 2{,}5\%$.

En cuanto a la ocupación de memoria, se puede observar tanto al compilar el programa, como en la visualización de los datos cargados en la FLASH. Observándose una ocupación de 172 palabras de 8 bits, o lo que es lo mismo:

- Memoria de programa: 86 palabras de 16 bits
- Memoria de datos: 5 palabras de 8 bits

Esto es notable, ya que 86 palabras de 2 bytes llevó **todo el programa**, teniendo en cuenta además que la mayor parte de ocupación del programa la tiene el algoritmo de generación de números aleatorios, esto supone una gran mejora con respecto a MARIE, donde **sólo el algoritmo de ordenamiento**, (implementado además como main) ocupaba 50 palabras de 2 bytes

Si quisiera comparar la perfomance del algoritmo implementado en ambas arquitecturas, tendría que hacer algunas salvedades importantes:

- El programa creado consta de una rutina de generación de números aleatorios que ocupa la mayor parte del tiempo de ejecución del programa. Para un vector de 7 elementos, dicha rutina lleva: ~ 811 ciclos
- En MARIE uno debía acceder a la memoria EEPROM tanto para los datos como para el programa, y sabemos que el acceso a este tipo de memoria es mucho más lento que acceder a un registro, a la memoria FLASH, o a la SRAM.
- La rutina creada de ordenamiento es ineficiente en el sentido de que hay parte de la estructura del código que se podría haber optimizado de ser dicha rutina el main o programa principal, como lo fue en MARIE.

Aclarado esto, se observó que la rutina de ordenamiento **para un vector de 7 elementos**, llevó: $\sim \mathbf{360}$ **ciclos**. Ahora bien, no podemos comparar ciclos de reloj, con instrucciones como teníamos en MARIE. Si suponemos que se implementa MARIE con la misma tecnología, es decir pudiendo obtener un f_{clock} de 16MHz como en el Atmega, y suponemos el peor de los casos, es decir, que las instrucciones de ambas arquitecturas son todas de un ciclo,

y además recordamos que : la cantidad de **instrucciones** media para MARIE era de aproximadamente **460**.

Luego, se ahorraron 100 instrucciones para la implementación, es decir:

$$mejora_r = \frac{100}{460}[\%] \cong 21{,}7\%$$

Ahora, esta asunción es extremadamente pesimista ya que en realidad muchas de las instrucciones implementadas en AVR llevan 2 ciclos de reloj. Como no tenemos forma de saber cuántos ciclos llevarían las instrucciones en MARIE ya que esta arquitectura no fue implementada, podemos hacer una suposición aproximada mirando el código en AVR. Si aproximadamente el 50% de las instrucciones llevan 2 ciclos de reloj, como se observa por inspección en el código implementado en AVR(aproximando y generalizando mucho), entonces tendremos que:

$$ciclos_{MARIE} = 460 + \frac{460}{2} = 690$$

Luego:

$$mejora_r = \frac{330}{690}[\%] \cong 47{,}8\%$$

Y esta mejora relativa se obtendría sin considerar los tiempos de acceso a memoria, que seguramente en MARIE serían el cuello de botella de la arquitectura, ni la optimización del algoritmo de ordenamiento, que en AVR se vio afectada por implementarse como rutina autocontenida de un programa mucho más extenso. Podemos concluir que hay una gran mejora incluso despreciando gran cantidad de factores que seguramente aumentarían notablemente la mejora relativa obtenida.

Conclusiones:

Más allá de los problemas de implementación que tuve con el algoritmo de generación de números aleatorios, fue muy interesante poder investigar más al respecto, conocer las alternativas mediante software y hardware, y tener una idea general para posterior estudio en caso de necesitarse o buscarse.

Por otro lado, si bien después de terminar el entregable y la práctica se pudieron explorar la mayoría de las instrucciones AVR y se pudo implementar todo, también cabe destacar la tremenda cantidad de tiempo que lleva depurar los errores de cada código en un lenguaje de tan bajo nivel. Un bit desplazado un lugar más a izquierda altera el resultado de todo el código, y en un programa extenso puede llevar muchas horas poder hallar el problema.

Bibliografía y sitios de interés:

- http://ww1.microchip.com/downloads/en/AppNotes/doc0936.pdf
- https://en.wikipedia.org/wiki/Linear_congruential_generator
- https://aaronschlegel.me/linear-congruential-generator-r.html
- http://pi.math.cornell.edu/~mec/Winter2009/Luo/Linear%20Congruential%20Generator/linear%20congruential%20gen1.html
- https://rosettacode.org/wiki/Linear_congruential_generator