Integrador TP N° 2

Tomás Vidal
Arquitectura de Computadoras
Facultad de Ingeniería, UNLP, La Plata, Argentina.
22 de Junio, 2024.

I. AVR y MARIE

Para resolver el problema Integrador 2 se empleó nuevamente el algoritmo de ordenamiento Bubble Sort (fig 1) realizado en Marie previamente, a diferencia de MARIE la arquitectura AVR posee múltiples registros por lo que la implementación del algoritmo fue más fácil en lineas generales. Se emplearon un total de 7 registros, 1 que almacena el cero para hacer operaciones con carry (*ZERO*), uno temporal (*TEMP*), uno para el dato actual del vector **a[j]** (*currVal*), otro para el siguiente dato **a[j+1]** (*nextVal*), I y J que son los contadores del *Bubble Sort* y por último *Counter* que es el contador del para hacer mediciones del promedio temporal. No es necesario emplear tanta cantidad de registros, la cantidad de podría minimizar, pero por cuestiones de legilibilidad, *debugeo* (más fácil aislar partes de código, y funciones) y, porque el problema no lo requiere, se hace uso de estos registros.

II. El código

Consiste básicamente en 3 partes o funciones: TIME_AVG_BUBBLE_SORT, LOAD_RND_VECTOR y BUBBLE_SORT. Al principio del código se encuentran las definiciones de los registros empleados y las constantes usadas a lo largo de todo el código. Posteriormente se encuentras varios MACROS que facilitan la legilibilidad y la reusablidad de código. Luego viene una sección denominada SETUP que consiste del código que sólo se ejecuta una vez antes del *loop principal*; luego está este *loop*, que no contiene nada ya que no hay que procesar nada continuamente. Y por útlimo esta la sección donde se declaran todas las funciones (son rutinas que guardan los datos previos en el Stack Pointer para poder volver al contexto previo con la instrucción ret), allí es donde están los algoritmos.

II-A. Vector de datos aleatorios

Esta función se encarga de llenar el vector de datos (que comienza en la posición 0x0100 de la SRAM y termina en 0x01FF). Los datos se llenan con valores aleatorios, estos valores son datos que se obtienen a partir de hacer lecturas con el ADC 0 del microprocesador, como este ADC no está conectado a nada las lecturas dan valores aleatorios, debido a los campos electromagnéticos presentes en el medio (la función READ_ADC hace una lectura y la almacena en el registro TEMP). Luego de hacer una captura la misma se almacena en la posición correspondiente dictada por el puntero X (el registro X es uno de 16 bits, que consiste en la unión de 2 de 8 bits r27:r26), se emplea un registro de 16 bits ya que la memoria de datos así lo requiere.

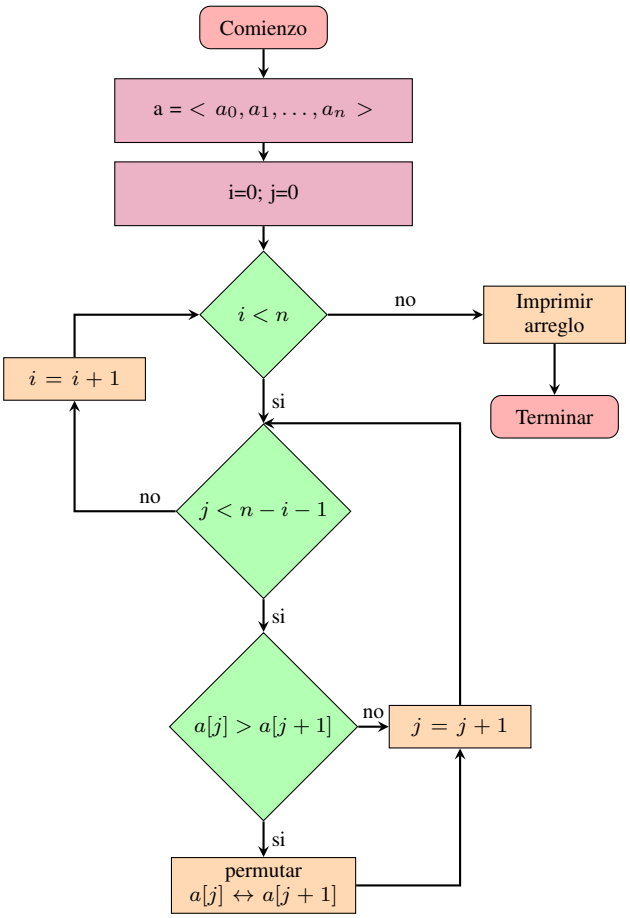

Fig. 1. Diagrama de flujo de Bubble Sort

II-B. Ordenamiento de los datos

Para ordenar los datos, como se explicó previamente se hizo uso del algoritmo Bubble Sort; como puntero del dato actual y el siguiente se empleó el registro X nuevamente, con predecrementos (-X) y postincrementos (X+). Los datos a los que se acceden se almacenan en los registros **currVal** y **nextVal**. Para hacer los saltos de memoria de programa y desarrollar la lógica del algoritmo se emplearon instrucciones que interactuan con el registro de *Status* (el cual conserva datos de operaciones entre registros) y estas instrucciones consideran que los valores con los que se trabajan son del

tipo *unsigned*, es decir sin signo; esto es importante ya que si se quiere trabajar con signo habría que utilizar otros tipos de saltos condicionales para que todo funcione correctamente.

II-C. *Promedio temporal*

La funcion TIME_AVG_BUBBLE_SORT se encarga de ejecutar y repetir LOAD_RND_VECTOR y BUBBLE_SORT en este orden 20 veces, antes de hacerlo enciende el LED 13 de la placa de desarrollo (PB5 en PORTB), y luego de finalizar el bucle apaja el LED, con esto se puede medir y promediar el tiempo de ejecución de estas funciones.

III. Mediciones y resultados

Para hacer las mediciones sobre la media que tardan conjuntamente LOAD_RND_VECTOR y BUBBLE_SORT, se realizó una simulación en el software *Proteus* (ver 2). Posteriormente se programó una placa de desarrollo con el Atmega328P (fig 3) y se midió el mismo tiempo grabando con una camara el tiempo que el LED se encontraba encendido.

	20 Ejecuciones (mín/media/máx) [s]	1 Ejecución (mín/media/máx) [ms]
Protues	1.64/1.64/1.64	82/82/82
Placa real	1.65/1.66/1.665	82.5/83/83.25

TABLA I. Medidas en la simulación de Proteus

III-A. *Conclusión de las mediciones*

Los resultados de la tabla I son congruentes con la teoría, ya que Bubble Sort es lento pero tiene consistencia en su ejecución (en cantidad de ciclos de reloj), además es muy probable que las diferencias de tiempo en la placa real sean debido a errores en medición más que en el algoritmo en sí.

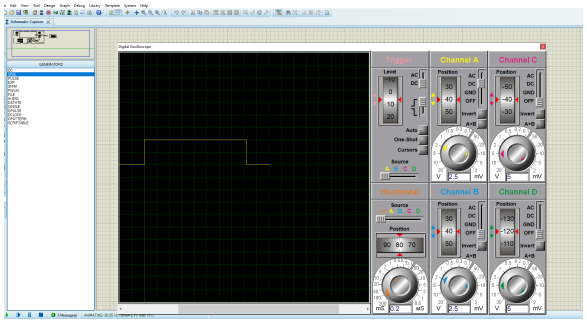

Fig. 2. Captura de la medición en Proteus

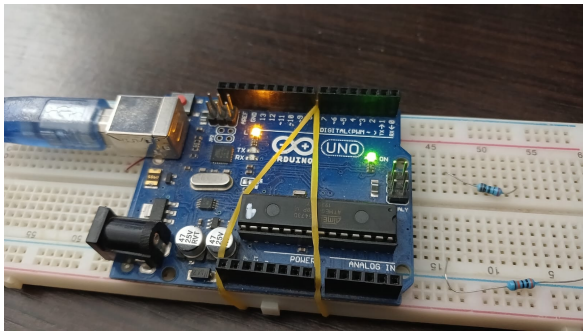

Fig. 3. Placa de desarrollo ejecutando el algoritmo

IV. Comparaciones AVR y MARIE

Para ver el rendimiento de la implementación en AVR, en comparación con la de MARIE, en el debuger de Microchip Studio se calculó cuantos ciclos de reloj le lleva al AVR concretar el ordenamiento, para el mejor y el peor de los casos de ordenamiento (y también el promedio entre ambos), aunque Bubble Sort tiene un coste aproximadamente constante, por lo que deberían ser resultados similares (cosa que ocurre como se puede apreciar en la tabla de resultados II). Una vez que se tienen las instrucciones de ambas arquitecturas se pueden calcular los ciclos de reloj sabiendo que de media MARIE ejecuta una instrucción cada 7 ciclos de reloj, por lo tanto: $CPI_{MARIE} \cong 7$, entonces se puden aproximar los ciclos de reloj haciendo $instruccionesCPI \cong ciclos$; haciendo el mismo análisis para AVR se tiene que $CPI_{AVR} \cong 1{,}5$ (los resultados de los cálculos se pueden apreciar en la tabla II).

	Cantidad de Instrucciones (mín/media/máx)	Cantidad de Ciclos de reloj (mín/media/máx)
AVR	918 / 939 / 960	1377 / 1409 / 1440
MARIE	534 / 566 / 597	3738 / 3962 / 4179

TABLA II. Instrucciones y ciclos de reloj en AVR y MARIE

	Cantidad de instrucciones	Porcentaje sobre el total
Memoria programa	160	0.5 %
Memoria datos	256	12.5 %

TABLA III. Espacio que ocupa el programa según Microchip Studio

Teniendo presente los calculos previos se puede calcular la mejora que tiene AVR sobre MARIE aplicando las siguientes ecuaciónes

$$t = \frac{N * CPI}{f_{reloj}}$$

Considerando un reloj de 16Mhz (aunque es indiferente para el Speedup)

$$t_{AVR} \cong 88\mu segundos$$

$$t_{MARIE} \cong 248\mu segundos$$

$$S(speedup) = \frac{t_{MARIE}}{t_{AVR}} \cong 2{,}8$$

Por lo que se concluye que la implementación en AVR es 2.8 veces mejor que la de MARIE.