Laboratorio de Principios de Mecatrónica: Práctica 2

Jorge Alejandro Ramírez Gallardo
Ingeniería Mecatrónica e Ingeniería Industrial
ITAM
Ciudad de México, México
rmrez.alejandro.g@gmail.com

Ricardo Edward Meadowcroft
Ingeniería en Computación y Licenciatura en Matemáticas Aplicadas
ITAM
Ciudad de México, México
rmeadowc@itam.mx

Abstract— Se analizaron los propósitos del lenguaje ensamblador y sus características con respecto a la arquitectura de computadoras. Además, se realizó una comparación entre el uso de este tipo de lenguaje y lenguajes de alto nivel. Finalmente, se dio una introducción a la configuración y uso de temporizadores e interrupciones.

INTRODUCCIÓN

Para esta práctica se buscó, en primer lugar, realizar la implementación de código en lenguaje ensamblador para la simulación de rutinas, para posteriormente hacer una comparación entre el uso de Arduino y lenguaje ensamblador, con sus respectivas aplicaciones. También, es importante el entendimiento de las interrupciones y los temporizadores y sus respectivos usos.

I. CONSTRUCCIÓN

A. Material

El material que se utilizó para el presente trabajo contiene los siguientes elementos.

- Microcontrolador ATMEGA 2560, que contiene los siguientes elementos de utilidad para la práctica:
  - 2 temporizadores/contadores de 8 bits con preescalador y comparador.
  - 4 temporizadores/contadores de 16 bits con preescalador, comparador y captura.
  - Contador de tiempo real con oscilador dedicado.
  - 4 canales PWM de 8 bits.
  - 12 canales PWM con resoluci´on programmable de 2 a 16 bits.
  - 16 canales de convesi´on anal´ogico-digital (ADC) de 10 bits. 4 puertos seriales USART programables.
  - Interfaz serial SPI Maestro/Esclavo.
  - Temporizador tipo Watchdog programmable con oscilador dedicado.
  - Interrupciones y Wake-up activados por cambio en pin.
- Computadora con software Arduino.

B. Desarrollo

1) Tipo de Instrucciones del microcontrolador ATMEGA

- INSTRUCCIONES LÓGICO-ARITMÉTICAS: Generalmente aceptan como argumentos dos ubicaciones de registro Rd y Rr (en ese orden), y aplican una operación lógica (como OR, AND, EOR (or exclusivo) o aritmética (como suma (ADD), resta (SUB) o multiplicación (MUL)) a ambas, y almacenan el valor resultante en Rd. Algunas instrucciones solo aceptan un argumento, como la negación (NEG), complemento a 2 (COM), o shifts y rotaciones (LSR, ASR, ROR).

  Algunos comandos tienen variantes que permiten usar un carry (ADC, SBC), o, en lugar de aceptar una dirección de registro Rr, en su lugar acepta un valor constante proporcionado en el código mismo (SUBI, ANDI, ORI, etc.). Un comando importante adicional es CP, que compara dos valores Rd y Rr, dejando sus valores intactos, y marca en 1 la bandera cero para uso en flujo de control.
- INSTRUCCIONES DE BIT: Manejan los valores 1/0 individuales de bits localizados dentro de un byte. Con SBI (con argumentos ‘io’ y ‘b’), particularmente, se puede asignar un bit ‘b’ específico de un registro I/O ‘io’ para ser de salida. Tal asignación se puede quitar análogamente con CBI. Con instrucciones de bit también se pueden manipular directamente las banderas localizadas en el registro de estado (SREG), usando BSET con la posición del bit que se desea cambiar para activarlo, y BCLR para desactivarlo.
- INSTRUCCIONES DE TRANSFERENCIA: Los microcontroladores AVR cuentan con una variedad de instrucciones que mueven información entre las distintas partes de memoria. Con la instrucción MOV (con argumentos Rd y Rr en ese orden) se

puede copiar la información de un registro Rr y colocarla en Rd; para copiar información desde un espacio de memoria que puede estar fuera de los registros se usa LDS junto con el registro a donde se guarda y la dirección de origen, mientras que la operación inversa de guardar un valor en registro a otro espacio de memoria se realiza con STS.

Con LDI se puede cargar un valor constante definido en código. IN (con Rd, io argumentos) copia la información de un espacio de i/o "io" (definido la dirección en términos relativos desde el inicio del rango de i/o) hacia un registro "Rd", mientras que OUT (con io y Rd en ese órden) copia la información del registro a la dirección i/o. Adicionalmente, operaciones PUSH y POP permiten repsectivamente agregar y sacar información de un stack.

- INSTRUCCIONES DE SALTO: Permiten al programa continuar ejecución en un número de línea que es distinto al que sigue y continuar en orden desde este punto, siendo entonces muy útil para formar estructuras de control. JMP y RJMP ambos saltan a una dirección definida en el código, pero JMP usa direcciones absolutas en el código, resultando un espacio más amplio que puede saltar, mientras que RJUMP es relativo a la posición actual, dando distancias más cortas, pero con menor tamaño de instrucción. IJMP salta a una dirección apuntada por el registro Z, que puede cambiar durante la ejecución del programa.
- OPERACIONES DE SALTO CONDICIONAL: Son cruciales para el funcionamiento de programas más complejos ya que permiten la toma de decisiones basado en una variedad de criterios. Muchos evalúan si una bandera de SREG se encuentra encendida o apagada, por ejemplo, BREQ salta si una comparación anterior entre registros dió una igualdad, basada en la bandera Z, mientras que BRNE salta si no hubo igualdad; similarmente BRSH salta si un valor es mayor o igual a otro mientras que BRLO salta si es menor. SBRS realiza un skip, es decir no llevar a cabo la instrucción inmediatamente subsecuente, si, para el registro Rr y posición de bit b, tal bit en el registro se encuentra encendido, mientras que SBRC hace skip si está apagado.
- OPERACIONES DE LLAMADA: Las operaciones CALL, RCALL, etc. funcionan de manera similar a las JMP análogas, al saltar a otra línea de código y continuar desde ahí; sin embargo, no continúan indefinidamente desde ahí, sino que cuando se llega a una instrucción RET/RETI regresa a la posición posterior a donde se llamó el CALL. Esto sirve para implementar subrutinas en el código.

II. RESULTADOS

A continuación, se presentan los resultados obtenidos durante la realización de la práctica.

A. Ensamblador

El objetivo del presente documento es comparar la funcionalidad del lenguaje ensamblador con respecto a las instrucciones de Arduino. En primera instancia, se generó una rutina de tiempo de 1 segundo de duración a través de las localidades de memoria del microcontrolador.

```
"tiempo: \n\t"
"LDI r22, 80 \n\t"
"LOOP_3: \n\t"
"LDI r21, 255 \n\t"
"LOOP_2: \n\t"
"LDI r20, 255 \n\t"
"LOOP_1: \n\t"
"DEC r20 \n\t"
"BRNE LOOP_1 \n\t"
"DEC r21 \n\t"
"BRNE LOOP_2 \n\t"
"DEC r22 \n\t"
"BRNE LOOP_3 \n\t"
"ret \n\t"
```

Posteriormente, utilizamos las instrucciones de Arduino para hacer parpadear 4 diodos emisores de luz (LED) a diferentes frecuencias: 0.25, 0.5, 1 y 2, revisando con ello la manera con la cual el programa permite hacer secuencias de tiempo (consultar en GitHub).

Para notar más claramente la diferencia, se implementó el mismo código con lenguaje ensamblador, combinando el primer programa con instrucciones adicionales que modificaron el valor de los LEDs a través del puerto de configuración de B (DDRB), como se enuncia a continuación.

```
void setup ()
{
DDRB = DDRB | B11111111; // Data Direction
Register B: Inputs 0-6, Output 7
}

"inicio: \n\t"
".EQU PORTB, 0x05 \n\t"
"LDI R16, 0B01000010 \n\t"
"LDI R17, 0B00100010 \n\t"
"LDI R18, 0B00010100 \n\t"
"LDI R19, 0B00011000 \n\t"
"OUT PORTB, R16 \n\t"
"call tiempo \n\t"
```

```
"call tiempo \n\t"
"call tiempo \n\t"
"call tiempo \n\t"
"call tiempo \n\t"
"OUT PORTB, R17 \n\t"
"call tiempo \n\t"
"OUT PORTB, R18 \n\t"
"call tiempo \n\t"
"call tiempo \n\t"
"call tiempo \n\t"
"call tiempo \n\t"
"call tiempo \n\t"
"OUT PORTB, R19 \n\t"
"call tiempo \n\t"
"jmp main \n\t"
```

Resaltamos que el número de instrucciones utilizadas para implementar el código es mayor en lenguaje ensamblador que en Arduino a nivel usuario, sin embargo, debido a que Arduino utiliza un compilador basado en C que lleve las instrucciones al lenguaje de bajo nivel (ensamblador), lleva más tiempo de ejecución y ocupa más recursos de memoria del microcontrolador, lo que lo hace más ineficiente su implementación en el AVR.

Para el último punto de esta sección se realizó la simulación de un semáforo mediante el uso de LEDs rojos, amarillos y verdes, en la cual el LED rojo permanece prendido durante 6 segundos, el amarillo durante 1 y el verde durante 5.

Se utilizó el lenguaje ensamblador para programarlo de manera intuitiva y sencilla, con una sola rutina repetitiva. Esto se realizó haciendo uso del Puerto B, en el cual se indicaron las secuencias de encendido y apagado respectivas a los 6 LEDs, para posteriormente realizar ciclos en dependiendo de los tiempos que debían ser aplicados. La imagen correspondiente al circuito construido se anexa a continuación.

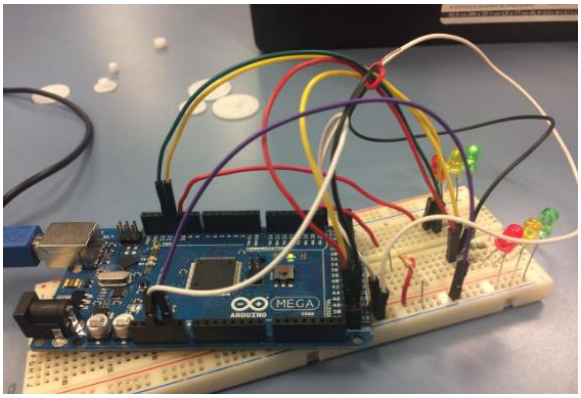

Imagen 1. Implementación de la sección A.

B. Interrupciones

En esta sección de la práctica se buscó realizar una introducción al concepto de interrupciones y el trabajo que se puede llevar a cabo con ellas. Primeramente, se realizó un programa en Arduino para hacer que un LED parpadeara a una frecuencia de 1 Hz, para seguidamente y sin interferir, implementar una rutina de contador, el cual es incrementado con respecto a un botón y la señal que se reciba de este.

Para el siguiente punto solamente se intercambió el botón por un interruptor óptico, con el cual primeramente se tuvieron problemas ya que este no funcionaba debido a que estaba roto, pero una vez que se intercambió, funcionó sin problemas.

El último punto que se tenía que realizar, se configuró una interrupción externa, la cual ejecutaba el código hecho para el punto del push-button. La implementación de imagen se muestra a continuación.

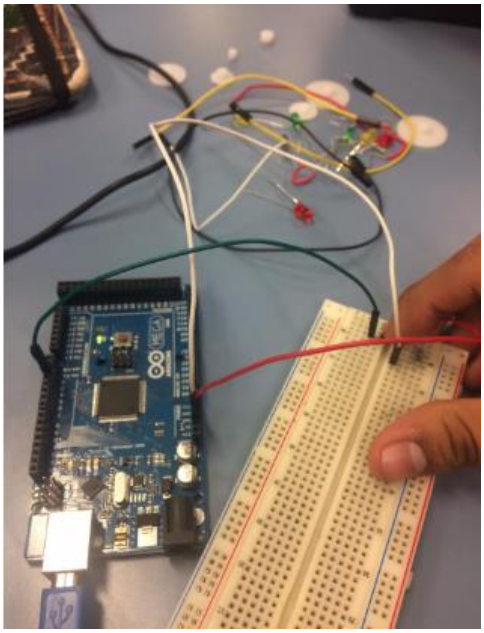

Imagen 2. Implementación de la sección B.

C. Temporizador

En la última sección se implementaron las interrupciones a través del temporizador interno del AVR. Como ejercicio de práctica previa, se habilitó y configuró un temporizador de 1 segundo de duración y, posteriormente, para la implementación del código, se deshabilitaron las interrupciones, se configuraron los registros de control del temporizador y se establecieron los preescaladores y el valor inicial del contador con la fórmula de la práctica propuesta. El código correspondiente se anexa a continuación.

```
//DETENER INTERRUPCIONES
  noInterrupts();
  //CONTROL REGISTER A EN 0
  TCCR1A=0;
  //CTC TIMER
  TCCR1B=0;
  TCCR1B |=(1<<WGM12);
  //PREESCALADOR 1024
  TCCR1B |=(1<<CS12) | (1<<CS10);
```

```
//INICIALIZAR
TCNT1=0;
//FÓRMULA
OCR1A=34286;//7812
//COMPARADOR
TIMSK1 |=(1<<OCIE1A);
//ACTIVAR INTERRUPCIONES
interrupts();
```

Por último, se realizó la ejecución del semáforo con la rutina descrita anteriormente en combinación con el repositorio de GitHub, mostrando con ello otra forma de realizar la tarea del semáforo dentro del microprocesador ATMEGA 2560. La imagen del circuito implementado se muestra a continuación.

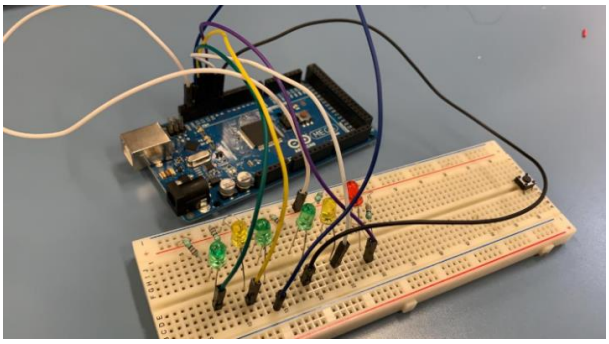

Imagen 3. Implementación de la sección C.

III. CONCLUSIONES

En esta práctica se pudo apreciar en más detalle las distintas instrucciones que lleva a cabo el AVR como piezas fundamentales que se usan construir los programas que se desean en el microcontrolador. De aquí, una comprehensión del código de ensamblado permite un entendimiento mejor de la relación entre el código y el hardware al empezar a escribir los programas a un nivel más alto de abstracción: el lenguaje tipo C de Arduino.

En adición, en relación con el manejo de entradas y salidas de AVR, se introdujo unos de los elementos integrados de i/o más importantes para los proyectos de mecatrónica: los timers y las interrupciones. Al permitir estos ejercer control sobre la manera en que cambia el comportamiento del programa conforme avanza el tiempo y la manera en que se modifican o se mantienen patrones de acción cuando el programa acepta entrada de información externa, el manejo correcto de tales herramientas se vuelve un elemento esencial de los proyectos futuros de Arduino.

REFERENCIAS

[1] EcuRed. Disponible en: https://www.ecured.cu/Pwd.