Introducción a la programación en Lenguaje Assembler.

✧ Que son las instrucciones:

El set de instrucciones de un microprocesador es el set de entradas binarias que producen acciones definidas durante un ciclo de instrucción. Un set de instrucciones es para el microprocesador lo mismo que una tabla de verdad es para una compuerta lógica, un registro de desplazamiento o un circuito sumador.

Por supuesto, las acciones que realiza un microprocesador con cada instrucción, son más complejas que las que realizan los dispositivos y compuertas antes mencionados.

✧ Instrucciones Binarias:

Una instrucción es un patrón de dígitos binarios el cual debe estar a disposición del microprocesador en el tiempo y forma que éste lo requiera.

Por ejemplo, cuando un microcontrolador PIC16C5X recibe el patrón binario de 12 bits '000010000000' en el momento adecuado, significa:

" CLEAR (borrar o poner a cero) EL REGISTRO W"

✧ Programa:

Un programa es una serie de instrucciones que causan que la computadora realice una tarea en particular.

✧ Programa en assembler :

¿Cómo introducimos un programa en assembler en una computadora?

Tenemos que traducirlo a hexadecimal o a binario. Se puede hacer instrucción a instrucción "a mano" o, como en nuestro caso, a través de un programa en una PC llamado **CROSS-ASSEMBLER.** Éste sería un ejemplo de programación en assembler:

```
            org   0
startmovlw 0Eh
            movwf REG1
            movlw 100
            addwf REG1,1
            end
```

✧ Desventajas del Assembly:

El lenguaje assembler no resuelve todos los problemas de programación. Uno de ellos es la tremenda diferencia entre el set de instrucciones del microprocesador y las tareas que el microprocesador debe realizar. Las instrucciones del microprocesador tienden a hacer cosas como: sumar contenidos de dos registros, desplazar el contenido de un acumulador un bit, o colocar un nuevo valor en el contador de programa (PC).

Por otro lado, el usuario generalmente quiere que el computador haga cosas como: chequear si un valor analógico leído se exedió de un cierto umbral, buscar y reaccionar ante un comando particular de una consola o teletipo, o activar un relé en el momento apropiado. El programador de lenguaje assembler debe traducir estas tareas a secuencias de simples instrucciones de microprocesador.

Esta tradución suele ser dificultosa, consume tiempo de trabajo.

Otro inconveniente es la *no portabilidad*. Cada microprocesador posee su propio set de instrucciones en el cual está reflejado su arquitectura interna.

Un programa en assembler escrito para 6809, no correrá en un 6502, Z80, 8080, o cualquier microprocesador de 16 o 32 bits. Ni siquiera dentro de la misma familia de microcontroladores de *Microchip Technology* (PICs) existen las mismas instrucciones. Cada modelo tiene un set de instrucciones propio que difiere en algunas instrucciones con los demás.

Para solucionar este inconveniente se utiliza un lenguaje de programación de alto o medio nivel, como puede ser el **lenguaje C**.

Convenciones en la escritura del código fuente

Para hacer la tarea del programador más grata, se usan algunas convenciones. Cada uno puede adoptar las que más le agraden y ayuden para ser más productivo. En general, las convenciones son cualquier acción que facilita la revisión y comprensión de un programa, especialmente el que uno mismo ha escrito cuando tiene que revisarlo algunos meses después. Comentamos algunas convenciones que usaremos:

Los ficheros de código fuente llevarán la extensión *.ASM
Los ficheros de listado llevarán la extensión *.LST
Los ficheros de código objeto llevarán la extensión *.OBJ
Los ficheros de errores llevarán la extensión *.ERR
Los ficheros ejecutables en formato *Intel Hex* llevarán la extensión *.HEX
Los nemónicos escritos en mayúscula hacen que el código escrito sea más visible.
Comentarios explicando cada línea de código.
Un párrafo de comentarios explica las rutinas o conjunto de instrucciones ya que los campos de comentarios suelen ser pequeños.
El espacio entre caracteres se escribe " _ " . *RB0_ES_1* es más fácil de leer que *RB0ES1*.

Recuerde que las convenciones son cualquier cosa que le haga más fácil la lectura y comprensión de su código, como por ejemplo:

1- Una cabecera estandarizada.
2- Colocar las rutinas en el mismo sitio, todas contiguas.
3- Dibujar diagramas de flujo o escribir seudocódigo.

Características del Assembler

✧ Campos del lenguaje assembler:

Las instrucciones están divididas en un número de campos, como se muestra debajo.

ETIQUETAS	OPERACIÓN	OPERANDO	COMENTARIOS
caso	movf	5,w	; lee puerto A
	retlw	4	; retorna de subrutina
trio	sleep		; bajo consumo

El campo del código de operación es el único que nunca puede estar vacío; éste siempre contiene una instrucción o una directiva del assembler.

El campo del operando o dirección puede contener una dirección o un dato, o puede estar en blanco.

El campo del comentario o de etiquetas es opcional. El programador asignará una etiqueta a una línea de instrucción o agregará un comentario según su conveniencia: normalmente, para hacer más fácil el uso y la lectura; por ejemplo si va a retomar el trabajo dentro de tres semanas.

Delimitadores:

Los campos van separados sólo con espacios y/o tabulaciones. No agregar nunca otros caracteres (comas, puntos, etc).
No utilice espacios extra, particularmente después de comas que separan operandos. (Ej: **movlw 5,w**)
No use caracteres delimitadores (espacios y tabulaciones) en nombres o etiquetas.

Etiquetas:

La etiqueta es el primer campo en una línea en lenguaje assembler y puede no existir.
Si una etiqueta está presente, el assembler la define como el equivalente a la dirección del 1° byte correspondiente a esa instrucción.
Esta etiqueta puede volver a usarse en otro lugar pero como ***operando*** de una instrucción.

El assembler reemplazará ésta etiqueta por el valor de cuando fue creada.
Se usan frecuentemente en las instrucciones de salto.

- No puede existir más de una etiqueta en el 1° campo de instrucción.

*- No pueden usarse como nombres de etiquetas a palabras ya reservadas por el assembler (**ORG**, **EQU**, etc.) o nombres de instrucciones (**movlw**, **call**, **nop**, etc.)*

Ejemplo:

```
START           movlw       DATO
                :
                :
                goto        START

DATO            EQU         05h
```

La instrucción **goto** **`START`** causa que la dirección de la instrucción con la etiqueta ***`START`*** ***`(movlw)`*** se cargue en el PC. Por lo tanto ésta instrucción será luego ejecutada.

*- No se permite el uso de números o caracteres no alfabéticos como 1° letra de la etiqueta. Como regla práctica: usar siempre **letras**, y en **mayúscula**.*

Mnemónicos (códigos de operación):

La tarea principal del assembler es la traducción de los códigos de operación en mnemónico en sus equivalentes binarios.
El assembler realiza ésta tarea usando una tabla como si lo haríamos " a mano" .
El assembler debe hacer algo más que traducir los códigos de operación. También debe determinar cuantos operandos requiere la instrucción y de que tipo. Esto es un poco complejo; algunas instrucciones (como **clrw**) no tienen operandos, otras (como sumas o saltos) tienen una, mientras que otras (manipulación de bits o *skips*) requieren dos.

Directivas:

Algunas instrucciones del lenguaje assembler no se traducen directamente a instrucciones del lenguaje máquina. Éstas instrucciones son directivas para el assembler; éstas asignan al programa ciertas áreas de memoria, definen símbolos, designan áreas de RAM para almacenamiento de datos temporales, colocan tablas o datos constantes en memoria y permiten referencias a otros programas.

Las directivas se utilizan como comandos escritos en el código fuente para realizar un control directo o ahorrar tiempo a la hora de ensamblar. El resultado de incorporar directivas se puede ver en el fichero *.LST, después de ensamblar el programa.

Para usar éstas directivas o pseudo-operandos, el programador las coloca en el ***campo del código de operación***, y, si lo requiere la directiva, una dirección o dato en el campo de dirección.

Las directivas más comunes son:

```
EQU  (Equate)
ORG  (Origin)
DEFB (Define Byte)
DEFW (Define Word)
END  (fin del código fuente)
```

♦ **EQU** (Equate - Equivalente)**:**

La directiva **EQU** permite al programador igualar *nombres* a datos o direcciones. Esta pseudo-operación se nota **EQU**. Los nombres utilizados se refieren generalmente a

direcciones de dispositivos, datos numéricos, direcciones de comienzo, direcciones fijas, posiciones de bits, etc.

```
PORT_A      EQU   5
START       EQU   0
CARRY       EQU   3
TIEMPO      EQU   5
```

Tambien se puede definir una equivalencia con el nombre de otra equivalencia ya definida.

```
PORT_B      EQU   PORT_A+1
PORT_C      EQU   PORT_A+2
FIN         EQU   START+100
FIN2        EQU   START+200
```

El valor del operando debe estar ya definido anteriormente, sino el compilador entregará un error.

♦ **ORG** (Origin - Origen)**:**

La directiva origen (se nota **ORG**) permite al programador especificar la posición de memoria donde programas, subrutinas o datos residirán. Los programas y los datos pueden estar alojados en diferentes áreas de memoria dependiendo de la configuración de memoria. Rutinas de comienzo, subrutinas de interrupción y otros programas deben comenzar en locaciones de memoria fijados por la estructura de microprocesador.

La directiva **ORG** hace al compilador colocar el código que le sigue en una nueva dirección de memoria (la salida del compilador no solo coloca los códigos de operación sino también las direcciones de cada instrucción y datos del programa).

Usualmente se la utiliza para: reset, programas de servicios de interrupción, almacenamiento en RAM, stack, programa principal, subrutinas.

Por Ej:

```
                ORG  00h
                goto inicializa
                org   04h   ; vector de interrupcion
                goto  interr
                ORG   05h
inicializa      movlw 08h   ; aquí comienza el programa
                :
                :
```

♦ **DEFB** (Define Byte), **DEFW** (Define Word)**:**

Esta directiva le permite al programador ingresar datos fijos en la memoria de programa. Estos datos pueden ser:

Tablas de conversión
Mensajes
Nombres
Umbrales
Comandos
Factores de conversión
Identificación de teclas
Direcciones de subrutinas

Esta directiva trata a los datos como parte permanente del programa.
El formato es muy simple:

```
MENSAJE    DEFB      'M','I','C','R','O',0
DELAY      DEFB      10
DATOS      DEFW      $A100
           DEFW      $A400
           DEFW      $0000
POTENCIA   DEFB      1,4,9,16,25,36,49,64,81
```

IMPORTANTE: *En nuestro caso esta directiva nose utiliza porque los microcontroladores PIC tienen separada la memoria de datos de la de programa, por lo tanto no pueden coexistir datos e instrucciones en la misma memoria.*

Operandos y direcciones:

Los ensambladores permiten elegir con libertad el tipo de elemento a colocar en el campo de operando o dirección.

Números decimales:

La mayoría de los ensambladores asumen todos los números como decimales a no ser que se marquen de otra manera.

Por ejemplo: `movlw 100`

Significa: "mover el número literal 100 (en decimal) al registro de trabajo W".

Otros sistemas de numeración:

Los ensambladores también aceptan números Hexadecimales, octales o binarios.
Esta es la forma de representarlos:

```
0A00h       ⎫ Hexadecimal
$0A00       ⎭

%01001011   ⎫ Binario
01011010b   ⎭

@123        ⎫ Octal (Algunos ensambladores aceptan
123Q        ⎭        también O o C.)
```

Si se utiliza la forma 0A000H para representar números hexadecimales, hay que tener en cuenta que el número debe comenzar siempre con un dígito entre 0 y 9 (no acepta comenzar con las letras A a F); es por eso que el número A000h se lo escribe 0A000h.

Nombres:

Los nombres pueden aparecer en el campo de operando; éstos son tratados como el dato que representan. (Ver directiva EQU).

Códigos de caracteres:

Algunos ensambladores permiten el uso de caracteres en ASCII.
Por ejemplo:

```
CHAR   EQU    't'
       movlw  'R'
```

Expresiones lógicas y aritméticas:

Los ensambladores permiten combinaciones de datos con operandos especiales, aritméticos o lógicos.
Éstos operandos se llaman *expresiones*.

Por ejemplo:

```
REG1   EQU    05h
VALOR  EQU    20h
       movlw  VALOR+2
       addwf  REG1,1
       addwf  REG1+1,1
```

Ensamblado condicional:

Algunos ensambladores les permiten incluir o excluir partes de programa, dependiendo de condiciones que existan en el tiempo de compilación.

La forma típica es:

```
IF  CONDICION
:
:
:
ENDIF
```

Si la **CONDICION** es verdadera en el tiempo de compilación, las instrucciones que están entre **IF** y **ENDIF** se incluirán en el programa.

Los usos típicos son:

Para incluir o excluir variables extras.
Para incluir cógigo de diagnóstico en condiciones de testeo (DEBUG).
Para permitir datos de distintos tamaños.

Desafortunadamente, el ensamblado condicional, tiende a complicar la lectura del programa, por lo tanto, traten de utilizarlo sólo si es necesario.

Inclusión de Código:

Algunos ensambladores permiten incluir código fuente (partes de programas) desde otros archivos.

Por ejemplo:

```
INCLUDE   DISPLAY.ASM
```

Le dice al compilador que incluya el código que se encuentra en el archivo *DISPLAY.ASM* como si fuese parte del propio programa.

Esto se utiliza para reutilizar códigos realizados con anterioridad. En el ejemplo del siguiente punto lo vemos más claro.

Macros:

A veces ocurren secuencias de instrucciones particulares en los programas que son repetitivas. Estas secuencias de instrucciones se pueden eliminar utilizando *MACROS.*

Las macros permiten asignarle un nombre a una secuencia de instrucciones. Luego se utiliza el nombre de la macro en el programa como si se usase la secuencia de instrucciones anterior.

Las macros no son lo mismo que las subrutinas. El código de las subrutinas aparece una sola vez en un programa y la ejecución del programa salta a la subrutina. En cambio, el ensamblador reemplaza cada ocurrencia del nombre de la macro con la secuencia especificada de instrucciones. Por consiguiente la ejecución del programa no salta a la macro como una subrutina.

Veamos un ejemplo utilizando los conceptos vistos hasta ahora:

Archivo "MULX10.ASM" :

```
MULX10 MACRO                  ; comienzo de la macro
            movf   tiempo,w ; guarda el tiempo en W
            rlf    tiempo   ; multiplica por 2
            rlf    tiempo   ; multiplica por 2
            rlf    tiempo   ; multiplica por 2
            addwf  tiempo   ; le suma una vez más
            addwf  tiempo   ; le suma una vez más
            ENDM             ; fin de la macro
```

Archivo "EJEMPLO1.ASM":

```
INCLUDE   MULX8.ASM

tiempo    EQU    0Ch
resultado EQU    0Dh

          movlw     20
          movwh  tiempo
          MULX10
          movwf  resultado

          end
```

Si ensamblamos el "*EJEMPLO1.ASM*" notaremos que el listado final queda de la siguiente forma:

```
tiempo    EQU    0Ch
resultado EQU    0Dh

          movlw     20
          movwh  tiempo
          movf   tiempo,w ; guarda el tiempo en W
          rlf    tiempo   ; multiplica por 2
          rlf    tiempo   ; multiplica por 2
          rlf    tiempo   ; multiplica por 2
          addwf  tiempo   ; le suma una vez más
          addwf  tiempo   ; le suma una vez más
          movwf  resultado

          end
```

Apéndice A

Puerto Paralelo de una PC

	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0	
OUTPUT	Pin 9	Pin 8	Pin 7	Pin 6	Pin 5	Pin 4	Pin 3	Pin 2	**278h, 378h, 3BCh**
OUTPUT	-	-	-	IRQ enable	Pin 17	Pin 16	Pin 14	Pin 1	**27Ah, 37Ah, 3BEh**
				IRQ enable	**select in**	**init**	**Auto FD**	**STROBE**	
INPUT	Pin 11	Pin 10	Pin 12	Pin 13	Pin 15	-	-	-	**279h, 379h, 3BDh**
	BUSY	**ACK**	**PE**	**SLCT**	**ERROR**				

DATA LATCH (278h, 378h o 3BCh):

Escribiendo en ésta dirección produce que los datos sean almacenados en el buffer de datos.

Leyendo ésta dirección se retorna el contenido del buffer.

Los drivers de salida de éste port manejan 2,6 mA a 2,4 Vdc y 24 mA a 0,5 Vdc. Resistores de 39 Ω están en serie con los drivers de salida.

PRINTER CONTROLS (27Ah, 37Ah o 3BEh):

Las señales de control del port paralelo son controladas a través de ésta dirección.

Éstas señales son manejadas por dispositivos colector abierto (pull up a +5Vdc a través de resistores de 4k7). Éstos dispositivos pueden manejar 16 mA a 0,4 Vdc.

Descripción de los bits:

BIT 7-5: Reservados

BIT 4: **IRQ Enable** - Cuando se setea (1), éste bit permite que una interrupción ocurra cuando ACK cambia de activa a inactiva.

BIT 3: **Slct In** - Cuando se setea (1), éste bit selecciona el dispositivo.

BIT 2: **Init** - Cuando se borra (0), éste bit resetea el dispositivo (50 μs de pulso mínimo).

BIT 1: **Auto FD** - Cuando se setea (1), éste bit causa que el dispositivo avance una línea después que una línea sea impresa.

BIT 0: **STROBE** - Pulso activo, mínimo 0,5 μs; coloca el dato en el dispositivo. Los datos válidos deben estar presentes por un tiempo mínimo de 0,5 μs antes y después del pulso de **STROBE**.

PRINTER STATUS (279h, 379h o 3BDh):

El estado del port paralelo se guarda en ésta dirección para ser leídos por el μP.

Descripción de los bits:

BIT 7: **BUSY** - Éste bit indica el estado de la señal de 'ocupado' del dispositivo. Cuando ésta señal está activa, este bit es un 0 y el dispositivo no puede aceptar datos. Está activo cuando entran datos, mientras el dispositivo está fuera de línea o mientras está en un estado de 'error'.

BIT 6: **ACK** - Éste bit representa el estado corriente de la señal 'acknowledge' del dispositivo. Un 0 significa que el dispositivo ha recibido el caracter y está listo para aceptar otro. Normalmente ésta señal está activa por aproximadamente 5 μs antes que **BUSY** se vuelva activa.

BIT 5: **PE** - Cuando se va a 1, indica que la impresora detectó el final del papel.

BIT 4: **Slct** - Cuando se va a 1, indica que el dispositivo se seleccionó.

BIT 3: **ERROR** - Cuando se va a 0, indica que el dispositivo encontró una condición de error.

BIT 2-0: Reservado.

Ejemplo:

Este programa muestra la forma de utilizar el puerto paralelo usando los bits de datos como salidas.

El programa realiza una secuencia de 8 LEDs conectados en modo *source* en los pines 2 al 9 y el 19 (CNG o masa) para los cátodos comunes.

```
/* leds.c */
/* Secuencia 8 LEDs conectados al puerto paralelo */

#include <stdio.h>
#include <conio.h>
#include <dos.h>

void main()
{
      /* colocar el valor del puerto que corresponda: 0x278, 0x378, 0x3BC */
      int puerto = 0x278;
      int led    = 1;
```

```
        /* apago todos los LEDs */
        outportb( puerto, 0 );

        /* si se pulsa cualquier tecla termina */
        while( kbhit() == 0 )
        {
                /* enciendo el LED */
                outportb( puerto, led );

                /* siguiente LED (1, 2, 4, 8, 16, 32, 64, 128) */
                led = led * 2 ;

                /* si excede 128 tiene que volver a 1 */
                if( led > 128 )
                        led = 1;
        }
}

/* FIN */
```

Tabla de caracteres ASCII

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL							BEL		HT	LF		FF	CR	SO	SI
1							ESC									
2	ESP	!	“	#	$	%	&	‘	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	

Los códigos ASCII menores a 32_{10} (00_{16} a $1F_{16}$) son los llamados caracteres de control. No se pueden representar gráficamente, se utilizan como comandos en los dispositivos series y paralelos (terminales, impresoras, etc.) efectuando operaciones como: avance de papel, retorno de carro, fin de transmisión, fin del archivo, etc.