UNNOBA

Manual Microcontrolador PIC16F84A

Guía de estudio

Cátedra Arquitectura;Lucas Benjamin Cicerchia;Pablo Addante

2014

Microcontroladores PIC

Los microcontroladores son circuitos integrados programables, pequeñas computadoras que caben en un chip y pueden ser programadas a la medida de la necesidad específica, para aplicaciones muy diversas.

Dentro de los más populares encontramos a la familia PIC, de Microchip Technology Inc. Se destacan por:

- su bajo costo,
- su amplia disponibilidad,
- su enorme base de usuarios,
- la extensiva cantidad de ejemplos,
- su facilidad de programación y reprogramación, y
- la existencia de herramientas de desarrollo gratuitas o de costo reducido.

Si bien muchos de estos microcontroladores pueden ser programados en lenguajes de alto nivel, como C, los mejores resultados se obtienen al programar en Assembler, ya que se logra un control más directo del dispositivo. Hay varias familias de PIC y cada una de ellas tiene particularidades de diseño que introducen pequeñas variantes en la cantidad de memoria, de puertos y el conjunto de instrucciones disponibles. Para el presente texto se usará el PIC16F84A.

DESCRIPCIÓN DE PINES

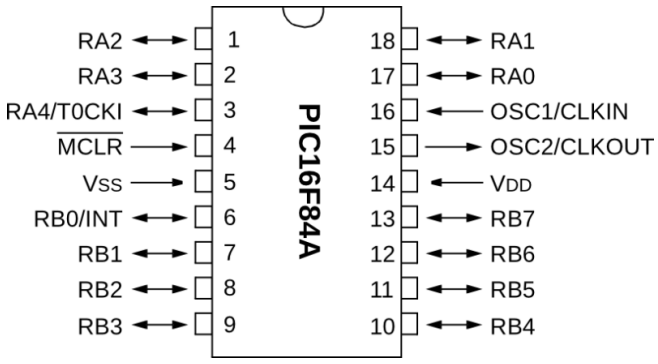


Ilustración 1 - PIC16F84A

Nombre Pin	Pin N°	Tipo	Descripción
OSC1/CLKIN	16	E	Entrada del cristal oscilador/entrada de fuente de reloj externa.
OSC2/CLKOUT	15	S	Salida del cristal oscilador/salida de frecuencia de ciclo de instrucción (1/4 de la frecuencia de oscilador).
$\overline{\text{MCLR}}$ /V_{PP}	4	E/A	Entrada Master Clear (Reset)/entrada de tensión de programación. RESET activo con nivel bajo.
			PORTA es un puerto E/S bidireccional.
RA0	17	E/S	
RA1	18	E/S	
RA2	1	E/S	
RA3	2	E/S	
RA4/T0CKI	3	E/S	También puede ser seleccionado como entrada de reloj para el temporizador/contador TMR0.
			PORTB es un puerto E/S bidireccional.
RB0/INT	6	E/S	RB0/INT también puede ser seleccionado como pin de interrupción externa.
RB1	7	E/S	
RB2	8	E/S	
RB3	9	E/S	
RB4	10	E/S	Pin de interrupción ante un cambio.
RB5	11	E/S	Pin de interrupción ante un cambio.
RB6	12	E/S	Pin de interrupción ante un cambio. Reloj de programación serial.
RB7	13	E/S	Pin de interrupción ante un cambio. Datos de programación serial.
V_{SS}	5	A	Referencia de tierra para los pines lógicos y de E/S.
V_{DD}	14	A	Alimentación positiva para los pines lógicos y de E/S.

Leyenda: E = Entrada S = Salida E/S = Entrada/Salida A = Alimentación

ARQUITECTURA

Para lograr un alto rendimiento y un aprovechamiento eficiente de los limitados recursos, hay varias particularidades de diseño de los PIC que los diferencian de las computadoras tradicionales.

ARQUITECTURA HARVARD

En la arquitectura de von Neumann tradicional, tanto el programa como los datos se buscan en la misma memoria usando el mismo bus. De esta forma, no es posible acceder a datos y programa (instrucciones a ejecutar) simultáneamente.

En la arquitectura Harvard, la memoria de programa y la memoria de datos están separadas físicamente en memorias distintas que son accedidas a través de diferentes buses. Esto permite leer o escribir la memoria de datos mientras se accede a la memoria de programa, ya que ambas utilizan buses separados. El PIC saca provecho de esta arquitectura para ejecutar una instrucción y, al mismo tiempo, buscar la próxima instrucción a ejecutar.

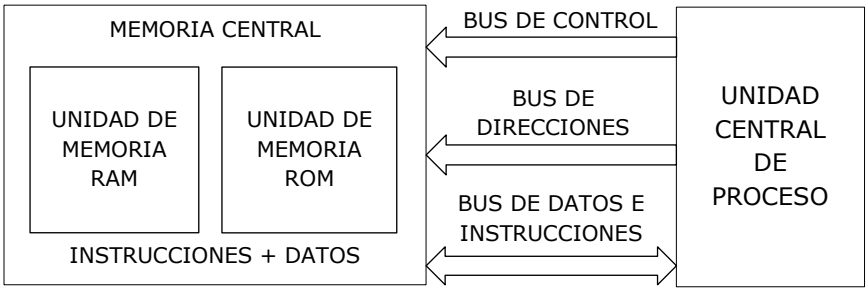


Ilustración 2 – Arquitectura de von Neumann

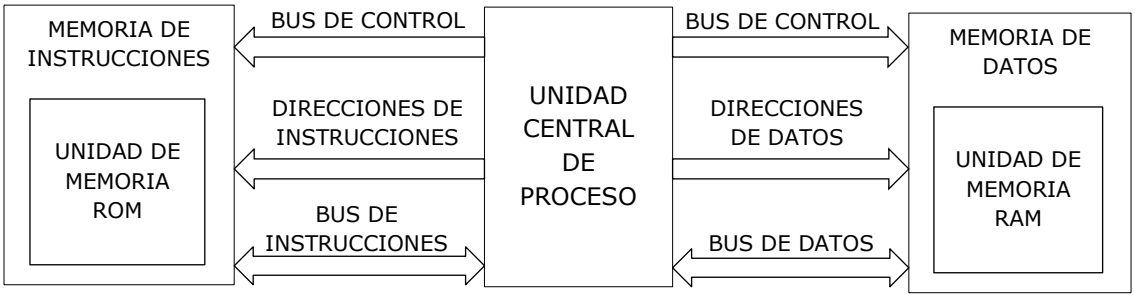


Ilustración 3 – Arquitectura Harvard

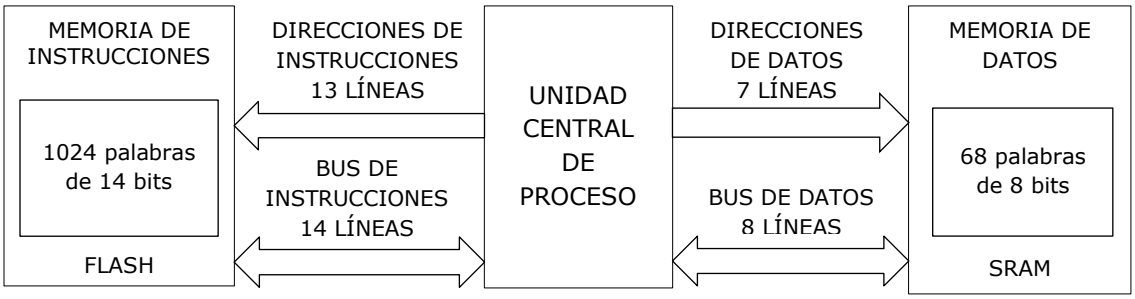


Ilustración 4 – Arquitectura del PIC16F84A

ARQUITECTURA RISC

El PIC se basa en un microprocesador de arquitectura RISC (*Reduced Instruction Set Computer*, Computadora de conjunto de instrucciones reducido). "Conjunto de instrucciones reducido" no significa que haya operaciones faltantes o eliminadas, sino que solo existe el mínimo de instrucciones necesarias, sin ninguna duplicación. El término "reducido" alude a la cantidad de trabajo realizado por cada instrucción. Al ser más simples y con función más acotada, requieren un menor número de ciclos para ejecutarse.

Algunas otras características de esta arquitectura están presentes en el PIC:

- **Instrucciones simétricas:** cualquier operación puede ser utilizada en cualquier registro con cualquier modo de direccionamiento. Esto simplifica la programación, reduce su curva de aprendizaje, y la hace más eficiente.
- **Instrucciones de palabra larga:** tener buses separados para la memoria de datos y para la de programa permite que las palabras de instrucción tengan un ancho mayor que las palabras de datos, que son de 8 bits, logrando así un uso más eficiente de la memoria de programa.
- **Instrucciones de una sola palabra:** los códigos de operación (*opcodes*) de 14 bits permiten que todas las instrucciones sean de una sola palabra, con lo cual se requiere un único ciclo y un único acceso al bus de memoria de programa para buscar cada instrucción.

- **Instrucciones de un solo ciclo:** la instrucción contiene toda la información requerida y se ejecuta en un único ciclo. Podría consumir un ciclo adicional si el resultado de la instrucción modifica el Contador de Programa.
- **Segmentación de instrucciones (*Instruction Pipelining*):** la búsqueda de una instrucción ocurre en un ciclo de máquina, mientras que su ejecución ocurre en el ciclo siguiente. Sin embargo, a través de la segmentación de dos etapas se superpone la búsqueda de una instrucción con la ejecución de la instrucción previa. De esta manera, en cada ciclo de máquina se ejecuta una instrucción y se busca la siguiente a ser ejecutada.

FLUJO DE INSTRUCCIONES/SEGMENTACIÓN

La entrada de reloj (de OSC1) es dividida por cuatro internamente para generar cuatro pulsos de reloj no superpuestos, llamados Q1, Q2, Q3 y Q4. Internamente, el contador de programa (PC, *Program Counter*) se incrementa en cada Q1, y la instrucción es buscada en la memoria de programa y almacenada en el Registro de Instrucción (IR, *Instruction Register*) en Q4. La instrucción es decodificada y ejecutada durante los siguientes Q1 a Q4.

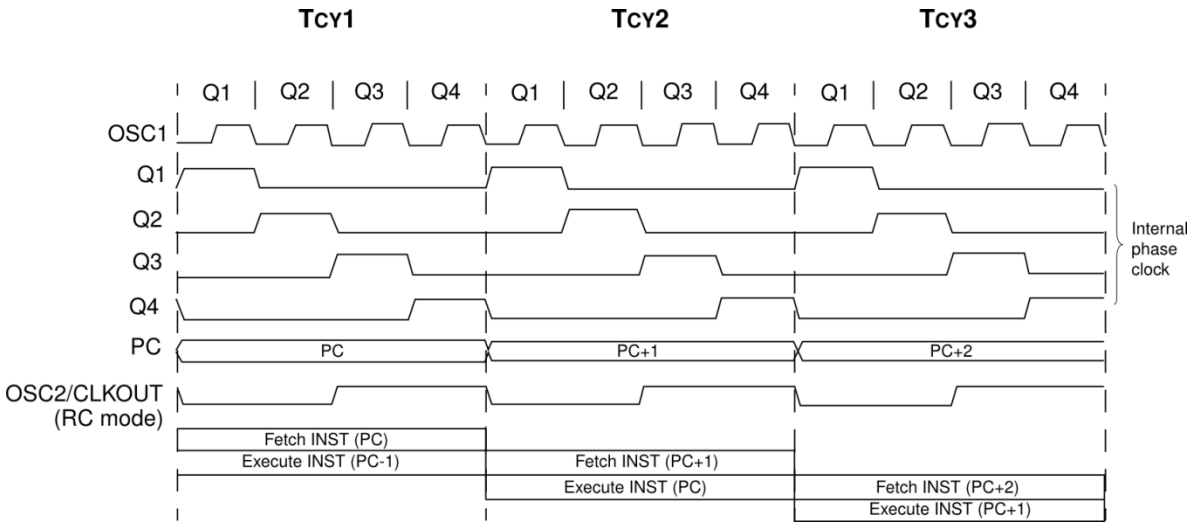


Ilustración 5 – Ciclos de reloj

Un ciclo de instrucción consiste, entonces, de cuatro ciclos (Q1, Q2, Q3 y Q4). La búsqueda toma un ciclo de instrucción, mientras que la decodificación y la ejecución toman otro. Sin embargo, debido a la segmentación, cada instrucción se ejecuta efectivamente en un ciclo. Si una instrucción provoca que el contador de programa cambie (por ejemplo, en el caso de GOTO), se requiere un ciclo adicional para completar la ejecución. La búsqueda de una instrucción comienza con el contador de programa incrementándose en Q1.

En el ciclo de ejecución, la instrucción previamente buscada está almacenada en el Registro de Instrucción (IR) en el ciclo Q1. Está instrucción es luego decodificada y ejecutada durante los ciclos Q2, Q3 y Q4. La memoria de datos es leída durante el ciclo Q2 (lectura de operando) y escrita durante Q4 (escritura en destino).

El siguiente ejemplo muestra la operación de la segmentación de dos etapas para una secuencia de instrucciones.

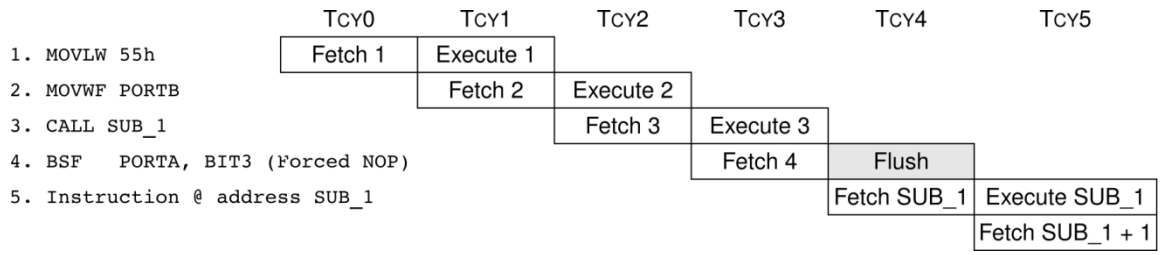


Ilustración 6 – Ejemplo de segmentación de dos etapas

En TCY0 se busca la primera instrucción n la memoria de programa. Durante TCY1 se ejecuta la primera instrucción mientras se busca la segunda. Durante TCY2 se ejecuta la segunda instrucción mientras se busca la tercera. Durante TCY3 se busca la cuarta instrucción mientras se ejecuta la tercera (CALL SUB_1). Cuando la tercera instrucción completa su ejecución, la CPU coloca la dirección de la instrucción cuatro en la Pila y luego cambia el Contador de Programa (PC) a la dirección de SUB_1. Esto significa que la instrucción buscada durante TCY3 necesita ser descartada y el proceso de segmentación debe reiniciarse buscando la nueva instrucción (SUB_1). Durante TCY4, se descarta la instrucción cuatro (ejecutando una instrucción NOP) y se busca la

instrucción en la dirección SUB_1. Finalmente, durante TCY5 se ejecuta la instrucción cinco y se busca la instrucción en la dirección SUB_1 + 1.

DIAGRAMA DE BLOQUES DEL MICROCONTROLADOR

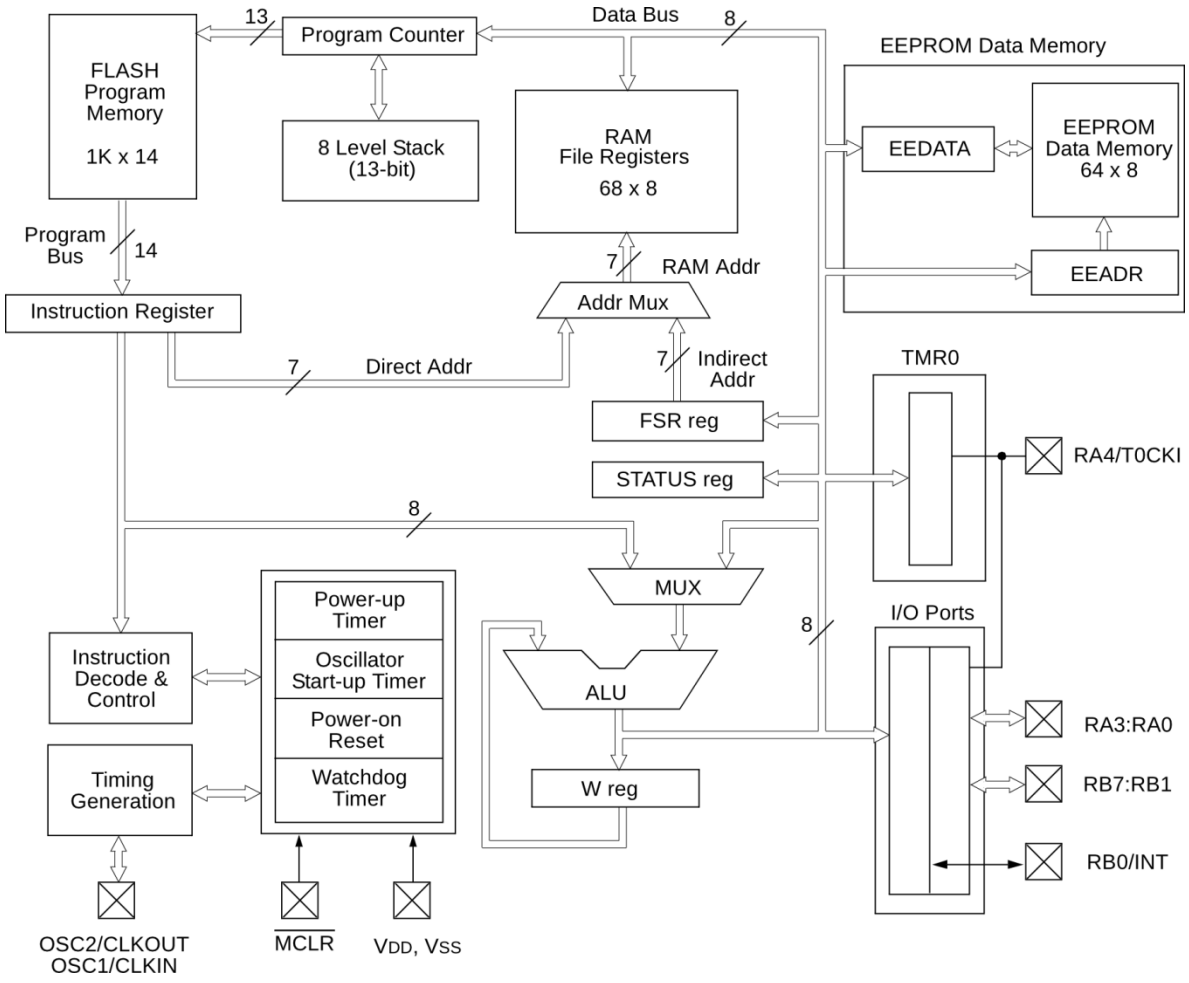


Ilustración 7 – Diagrama de bloques del microcontrolador

UNIDAD CENTRAL DE PROCESAMIENTO (CPU)

La CPU puede pensarse como el "cerebro" del dispositivo. Es responsable de buscar la instrucción correcta a ejecutar, de decodificar dicha instrucción y, finalmente, de ejecutarla. En ocasiones, la CPU trabaja conjuntamente con la ALU para completar la ejecución de una instrucción (en las operaciones aritméticas y lógicas).

La CPU controla el bus de direcciones de la memoria de programa, el bus de direcciones de la memoria de datos y los accesos a la pila.

RELOJ DE INSTRUCCIONES

Cada ciclo de instrucción (TCY) está compuesto por cuatro ciclos Q (Q1-Q4). El tiempo de cada ciclo Q es el mismo que el tiempo de ciclo del oscilador del dispositivo (TOSC). Los ciclos Q proveen la temporización para la Decodificación, Lectura, Procesamiento de Datos, Escritura, etc., de cada ciclo de instrucción.

Los cuatro ciclos Q que componen un ciclo de instrucción (TCY) pueden generalizarse de la siguiente manera:

Q1: Ciclo de Decodificación de Instrucción o No operation (NOP) forzado.

Q2: Ciclo de Lectura de Datos de Instrucción o No operation (NOP).

Q1: Procesamiento de Datos.

Q1: Ciclo de Escritura de Datos de Instrucción o No operation (NOP).

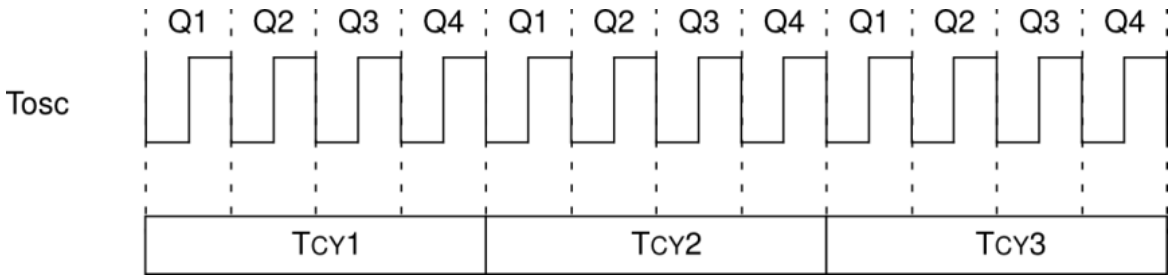


Ilustración 8 – Ciclos de instrucción

UNIDAD ARITMÉTICO LÓGICA (ALU)

El PIC contiene una ALU de 8 bits y un registro de trabajo (registro W) de 8 bits. La ALU es una unidad aritmética y lógica de propósito general. Ejecuta funciones aritméticas y booleanas entre los datos en el registro de trabajo y cualquier otro registro. En las instrucciones de dos operandos, uno de los operandos es el registro W, mientras que el otro es cualquier registro o una constante inmediata. En las instrucciones de un solo operando, el operando es o el registro W o cualquier registro. El registro W solo es usado en operaciones de la ALU y no es direccionable.

Dependiendo de la instrucción ejecutada, la ALU puede afectar los valores de los bits *Carry* (C), *Digit Carry* (DC) y *Zero* (Z) en el registro STATUS.

REGISTRO STATUS

El registro STATUS contiene el estado aritmético de la ALU, el estado de RESET y los bits de selección de banco para la memoria de datos. Dado que la selección de los bancos de la memoria de datos es controlada por este registro, se requiere que esté presente en cada banco. Además, este registro está en la misma posición relativa (desplazamiento u *offset*) en cada banco.

El registro STATUS puede ser el destino para cualquier instrucción, como cualquier otro registro. Si es el destino para una instrucción que afecta los bits Z, DC o C, la escritura de esos tres bits se deshabilita, ya que su valor será modificado por la lógica del dispositivo. Más aún, los bits $\overline{\text{TO}}$ y $\overline{\text{PD}}$ no pueden ser escritos. En conclusión, el resultado de una instrucción con el registro STATUS como destino puede ser distinto al esperado.

Se recomienda usar solo las instrucciones BCF, BSF, SWAPF y MOVWF para alterar el registro STATUS, dado que las mismas no afectan a los bits Z, C o DC.

Registro STATUS (Dirección 03h, 83h)

U **(0)**	U **(0)**	R/W **(0)**	R **(1)**	R **(1)**	R/W **(x)**	R/W **(x)**	R/W **(x)**
IRP	RP1	RP0	$\overline{TO}$	$\overline{PD}$	Z	DC	C
bit 7							bit 0

Leyenda: R = Lectura W = Escritura U = No implementado, leído como 0
(0) = valor inicial 0 **(1)** = valor inicial 1 **(x)** = valor inicial desconocido

bit 7 **No implementado**: Mantener en '0'

bit 6 **No implementado**: Mantener en '0'

bit 5 **RP0**: bit de selección de banco de registros (para direccionamiento directo)

1 = Banco 1 (80h – FFh)

0 = Banco 0 (00h – 7Fh)

bit 4 $\overline{\textbf{TO}}$: bit de *time-out*

1 = Después del encendido (*power-up*), o de las instrucciones CLRWDT o SLEEP

0 = Ocurrió un time-out del WDT (*WatchDog Timer*)

bit 3 $\overline{\textbf{PD}}$: bit de apagado (*power-down*)

1 = Después del encendido (*power-up*) o por la instrucción CLRWDT

0 = Por la ejecución de la instrucción SLEEP

bit 2 **Z**: bit de cero (*zero*)

1 = El resultado de una operación aritmética o lógica es cero

0 = El resultado de una operación aritmética o lógica no es cero

bit 1 **DC**: bit de *Digit carry/* $\overline{borrow}$ (instrucciones ADDWF, ADDLW, SUBLW, SUBWF)

1 = Hubo acarreo desde el 4° bit menos significativo del resultado

0 = No hubo acarreo desde el 4° bit menos significativo del resultado

(para $\overline{digit\ borrow}$, la polaridad está invertida)

bit 0 **C**: bit de *Carry/* $\overline{borrow}$ (instrucciones ADDWF, ADDLW, SUBLW, SUBWF)

1 = Hubo acarreo desde el bit más significativo del resultado

0 = No hubo acarreo desde el bit más significativo del resultado

(para $\overline{borrow}$, la polaridad está invertida)

MEMORIA

Hay dos bloques de memoria: la memoria de programa y la memoria de datos. Cada bloque tiene su propio bus, por lo que el acceso a cada uno puede ocurrir en el mismo ciclo de reloj.

MEMORIA DE PROGRAMA

El contador de programa de 13 bits es capaz de direccionar un espacio de memoria de 8K x 14. Para el PIC16F84A solo están implementados físicamente los primeros 1K x 14 (0000h-03FFh). El acceso a una dirección por encima de lo físicamente implementado causará un solapamiento o vuelta circular. Por ejemplo, para las ubicaciones 20h, 420h, 820h, C20h, 1020h, 1420h, 1820h y 1C20h, la instrucción será la misma.

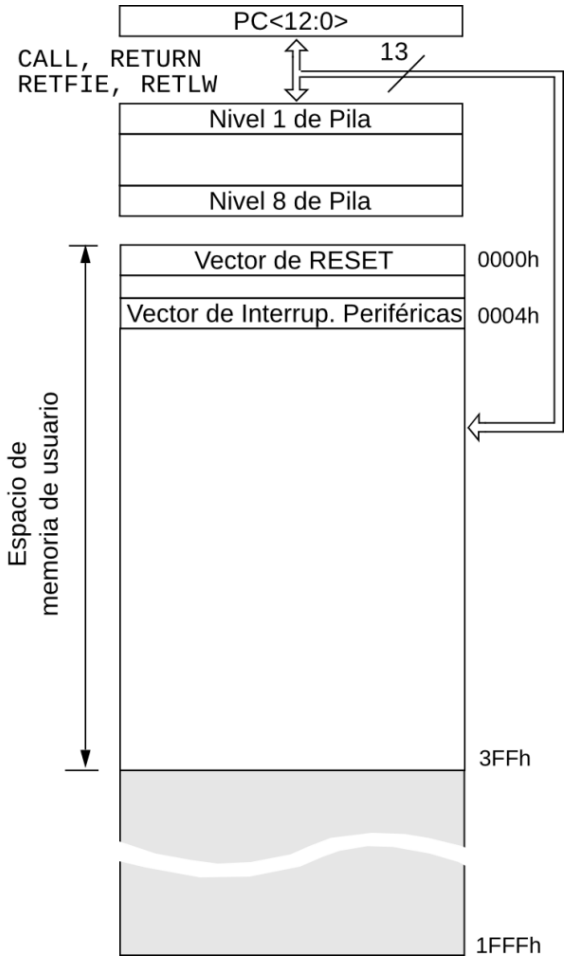


Ilustración 9 – Memoria de programa

PILA

En ocasiones, la ejecución normal del programa debe detenerse, ya sea para ejecutar una subrutina o para atender una interrupción externa. Una vez terminado ese salto, es necesario retomar la ejecución del programa principal desde el punto donde se había detenido. La pila permite guardar la dirección de retorno del salto.

El PIC16F84A tiene una pila por hardware con 8 niveles de profundidad y 13 bits de ancho. Esto brinda la posibilidad de hasta 8 llamadas a subrutinas o interrupciones combinadas. El espacio de pila no es parte ni del espacio de programa ni del de datos y el puntero de pila no puede ser leído ni escrito.

El Contador de Programa (PC) es agregado (PUSH) a la pila cada vez que se ejecuta una instrucción CALL o que una interrupción provoca un salto. Al ejecutar una instrucción RETURN, RETLW o RETFIE, se quita (POP) el elemento en el tope de la pila (el que ha sido agregado más recientemente). Después de agregar 8 elementos a la pila, el noveno PUSH sobrescribe el valor que fue almacenado en el primer PUSH. El décimo PUSH sobrescribe el segundo, y así sucesivamente.

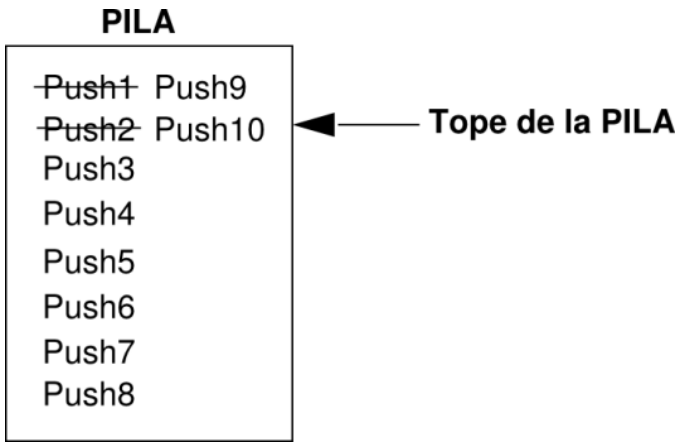


Ilustración 10 – Pila

Es importante notar que no hay instrucciones o mnemónicos llamados PUSH o POP, son acciones que ocurren internamente a partir de la ejecución de las instrucciones CALL, RETURN, RETLW y RETFIE, o a partir de interrupciones.

MEMORIA DE DATOS

La memoria de datos está dividida en dos bancos: el Banco 0 y el Banco 1. Para cambiar de banco se utiliza el bit RP0 del registro STATUS (STATUS<5>). RP0 en '0' selecciona el Banco 0, mientras que RP0 en '1' selecciona el Banco 1. Cada banco se extiende hasta 128 bytes. Solo los primeros 80 están implementados en el PIC16F84A. Las primeras 12 ubicaciones de cada banco están reservadas para los Registros de Función Especial (SFR, *Special Function Registers*). Los 68 restantes conforman el área de Registros de Propósito General (GPR, *General Purpose Registers*).

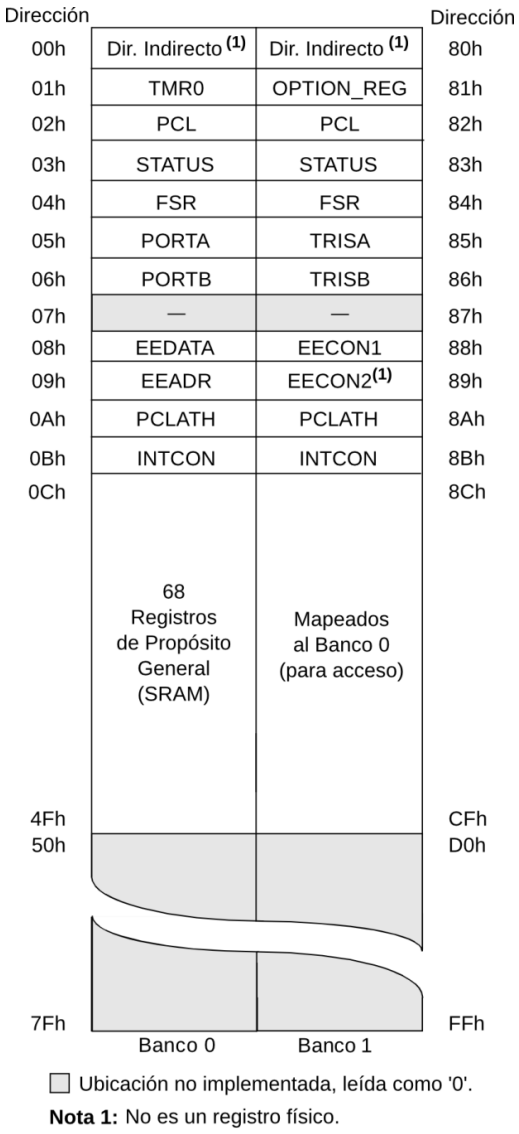
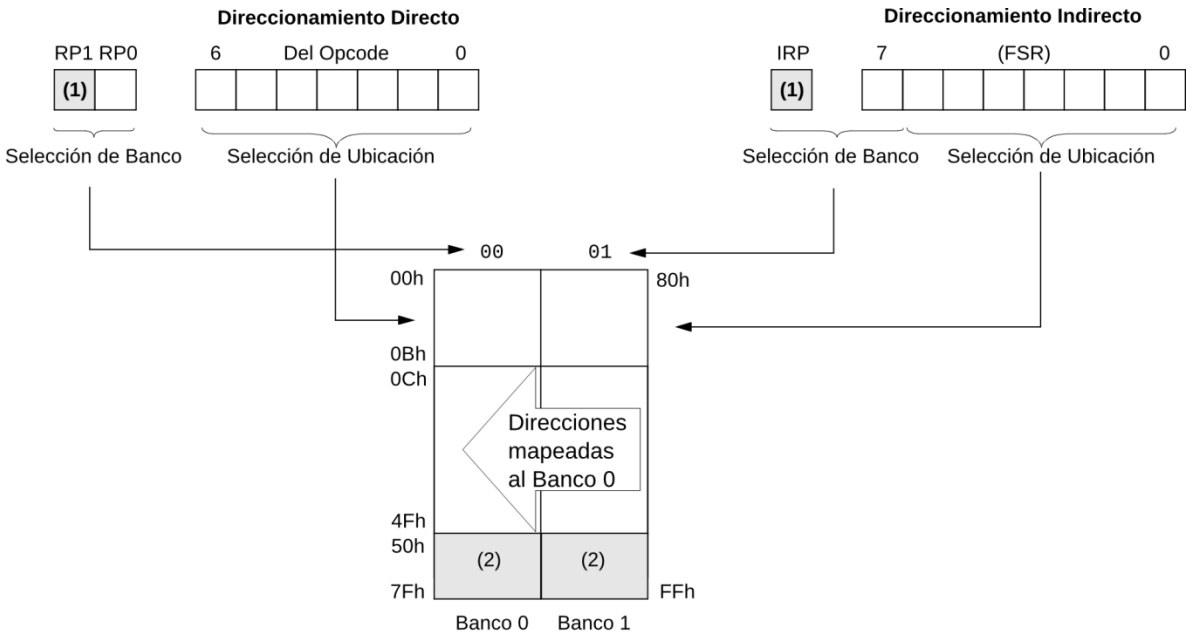


Ilustración 11 – Memoria de datos

DIRECCIONAMIENTO INDIRECTO

El direccionamiento indirecto es un modo de direccionar la memoria de datos en el cual la dirección en la instrucción no es fija. Se usa el registro FSR (*File Select Register*, Registro de Selección de Archivo) como puntero a la ubicación a ser leída o escrita. Dado que el puntero está en la RAM, el contenido puede ser modificado por el programa. Puede ser útil para manejar tablas de datos.



Nota 1: No utilizado.
2: No implementada.

Ilustración 12 - Direccionamiento Indirecto

El direccionamiento indirecto es posible mediante el uso del registro INDF. Cualquier instrucción que use este registro en realidad accede al registro apuntado por el registro FSR. Veamos un ejemplo:

- El Registro 05 contiene el valor 10h.
- El Registro 06 contiene el valor 0Ah.
- Se carga el valor 05 en el registro FSR.
- Una lectura del registro INDF retornará el valor 10h.
- El valor del registro FSR se incrementa en uno (FSR = 06).
- Una lectura del registro INDF ahora retornará el valor 0Ah.

Leer el registro INDF indirectamente (con FSR = '0') leerá 00h. Escribir al registro indirectamente resultará en un *no-operation* (aunque los bits de STATUS podrían ser afectados).

A continuación se muestra un programa sencillo para limpiar las ubicaciones 20h-2Fh de la memoria RAM utilizando direccionamiento indirecto.

```
            movlw       0x20        ;inicializar puntero
            movwf       FSR         ;a RAM
NEXT        clrf        INDF        ;limpiar el registro INDF (todos '0')
            incf        FSR         ;incrementar el puntero
            btfss       FSR,4       ;todo listo?
            goto        NEXT        ;NO, limpiar el siguiente
CONTINUE :                          ;SI, continuar
```

PUERTOS DE ENTRADA/SALIDA (E/S)

Los pines de E/S de propósito general pueden ser considerados como los más simples de los periféricos. Le permiten al microcontrolador monitorear y controlar otros dispositivos.

PUERTO A (PORTA)

PORTA es un puerto bidireccional de 5 bits. TRISA es el registro correspondiente a la configuración del puerto, para definir la dirección de los datos (entrada o salida). Un '1' en un bit de TRISA hará que el pin correspondiente del PORTA actúe como entrada. Un '0' en un bit de TRISA hará que el pin correspondiente del PORTA actúe como salida. Al momento de encendido del PIC, los pines del puerto están configurados como entradas y su valor es leído como '0'.

Registros asociados con el Puerto A

Dir.	Nombre	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Valor al encendido	Valor al RESET
05h	PORTA	—	—	—	RA4	RA3	RA2	RA1	RA0	---x xxxx	---u uuuu
85h	TRISA	—	—	—	TRISA4	TRISA3	TRISA2	TRISA1	TRISA0	---1 1111	---1 1111

Leyenda: x = desconocido, u = inalterado, - = no implementado, leído como '0'.

Ejemplo de inicialización del Puerto A

```
BCF   STATUS, RP0 ;

CLRF  PORTA       ; Inicializar PORTA limpiando los latches

                  ; de salida de datos

BSF   STATUS, RP0 ; Seleccionar Banco 1

MOVLW 0x0F        ; Valor usado para inicializar la dirección

                  ; de los datos: 00001111

MOVWF TRISA       ; Configurar RA<3:0> como entradas y RA4 como

                  ; salida. TRISA<7:5> siempre se leen como '0'
```

PUERTO B (PORTB)

PORTB es un puerto bidireccional de 8 bits. TRISB es el registro correspondiente a la configuración del puerto, para definir la dirección de los datos (entrada o salida). Un '1' en un bit de TRISB hará que el pin correspondiente del PORTB actúe como entrada. Un '0' en un bit de TRISB hará que el pin correspondiente del PORTB actúe como salida.

Registros asociados con el Puerto B

Dir.	Nombre	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Valor al encender	Valor al RESET
06h	PORTB	RB7	RB6	RB5	RB4	RB3	RB2	RB1	RB0/INT	xxxx xxxx	uuuu uuuu
86h	TRISB	TRISB7	TRISB6	TRISB5	TRISB4	TRISB3	TRISB2	TRISB1	TRISB0	1111 1111	1111 1111

Leyenda: x = desconocido, u = inalterado.

Ejemplo de inicialización del Puerto B

```
BCF   STATUS, RP0 ;

CLRF  PORTB       ; Inicializar PORTB limpiando los latches

                  ; de salida de datos

BSF   STATUS, RP0 ; Seleccionar Banco 1

MOVLW 0xCF        ; Valor usado para inicializar la dirección

                  ; de los datos: 11001111

MOVWF TRISB       ; Configurar RB<3:0> como entradas, RB<5:4>

                  ; como salidas y RB<7:6> como entradas
```

INSTRUCCIONES

Cada instrucción de un PIC16F84A es una palabra de 14 bits, dividida en un OPCODE que especifica la operación de la instrucción y uno o más operandos que especifican la operación de la instrucción.

Para las instrucciones **orientadas a byte**, 'f' representa un designador de registro (*file register*) y 'd' representa un designador de destino. El designador de registro especifica qué registro es utilizado por la instrucción. El designador de destino especifica dónde se ubicará el resultado de la operación. Si 'd' es 0, el resultado se ubicará en el registro W. Si 'd' es 1, el resultado se ubicará en el registro especificado en la instrucción.

Para las instrucciones **orientadas a bit**, 'b' representa un designador de bit que selecciona el número de bit afectado por la operación, mientras que 'f' representa la dirección del registro en el cual se encuentra el bit.

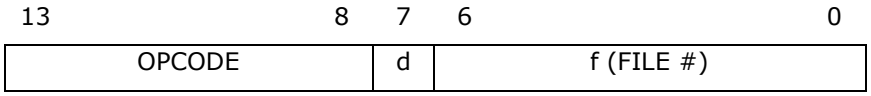
Para las operaciones **de literal y control**, 'k' representa una constante o valor literal de 8 u 11 bits.

Todas las instrucciones se ejecutan en un único ciclo de instrucción, a menos que un chequeo condicional sea verdadero o el contador de programa cambie como resultado de una instrucción.

En este caso, la ejecución toma dos ciclos de instrucciones, con el segundo ciclo ejecutado como NOP. Un ciclo de instrucción consiste de cuatro períodos de oscilador. Entonces, para una frecuencia de 4 Mhz, el tiempo de ejecución normal es de 1 µs. Si un chequeo condicional es verdadero o el contador de programa cambia como resultado de una instrucción, el tiempo de ejecución de la instrucción es de 2 µs.

Formatos generales de instrucción

Operaciones de registro orientadas a byte

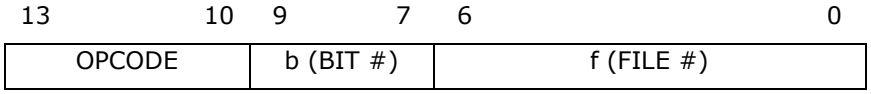


d = 0 para destino W

d = 1 para destino f

f = dirección de registro (7 bits)

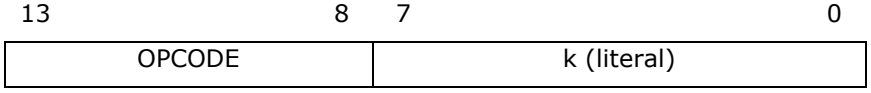
Operaciones de registro orientadas a bit



b = dirección de bit (3 bits)

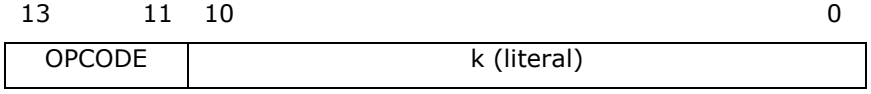
f = dirección de registro (7 bits)

Operaciones de literal y control General



k = valor inmediato (8 bits)

Solo instrucciones CALL y GOTO



k = valor inmediato (11 bits)

Mnemónico, Operandos		Descripción	Ciclos	Opcode de 14 Bits		STATUS Afectado
				MSb	LSb	
OPERACIONES DE REGISTRO ORIENTADAS A BYTE						
ADDWF	f, d	Add W and f	1	00 0111 dfff ffff		C, DC, Z
ANDWF	f, d	AND W with f	1	00 0101 dfff ffff		Z
CLRF	f	Clear f	1	00 0001 1fff ffff		Z
CLRW	-	Clear W	1	00 0001 0xxx xxxx		Z
COMF	f, d	Complement f	1	00 1001 dfff ffff		Z
DECF	f, d	Decrement f	1	00 0011 dfff ffff		Z
DECFSZ	f, d	Decrement f, Skip if 0	1(2)*	00 1011 dfff ffff		
INCF	f, d	Increment f	1	00 1010 dfff ffff		Z
INCFSZ	f, d	Increment f, Skip if 0	1(2)*	00 1111 dfff ffff		
IORWF	f, d	Inclusive OR W with f	1	00 0100 dfff ffff		Z
MOVF	f, d	Move f	1	00 1000 dfff ffff		Z
MOVWF	f	Move W to f	1	00 0000 1fff ffff		
NOP	-	No Operation	1	00 0000 0xx0 0000		
RLF	f, d	Rotate Left f through Carry	1	00 1101 dfff ffff		C
RRF	f, d	Rotate Right f through Carry	1	00 1100 dfff ffff		C
SUBWF	f, d	Subtract W from f	1	00 0010 dfff ffff		C, DC, Z
SWAPF	f, d	Swap nibbles (1/2 bytes) in f	1	00 1110 dfff ffff		
XORWF	f, d	Exclusive OR W with f	1	00 0110 dfff ffff		Z
OPERACIONES DE REGISTRO ORIENTADAS A BIT						
BCF	f, b	Bit Clear f	1	01 00bb bfff ffff		
BSF	f, b	Bit Set f	1	01 01bb bfff ffff		
BTFSC	f, b	Bit Test f, Skip if Clear	1(2)*	01 10bb bfff ffff		
BTFSS	f, b	Bit Test f, Skip if Set	1(2)*	01 11bb bfff ffff		
OPERACIONES DE LITERAL Y CONTROL						
ADDLW	k	Add literal and W	1	11 111x kkkk kkkk		C, DC, Z
ANDLW	k	AND literal with W	1	11 1001 kkkk kkkk		Z
CALL	k	Call subroutine	2	10 0kkk kkkk kkkk		
CLRWDT	-	Clear Watchdog Timer	1	00 0000 0110 0100		TO, PD
GOTO	k	Go to address	2	10 1kkk kkkk kkkk		
IORLW	k	Inclusive OR literal with W	1	11 1000 kkkk kkkk		Z
MOVLW	k	Move literal to W	1	11 00xx kkkk kkkk		
RETFIE	-	Return from interrupt	2	00 0000 0000 1001		
RETLW	k	Return with literal in W	2	11 01xx kkkk kkkk		
RETURN	-	Return from Subroutine	2	00 0000 0000 1000		
SLEEP	-	Go into standby mode	1	00 0000 0110 0011		TO, PD
SUBLW	k	Subtract W from literal	1	11 110x kkkk kkkk		C, DC, Z
XORLW	k	Exclusive OR literal with W	1	11 1010 kkkk kkkk		Z

* 2 ciclos si se modifica el Contador de Programa o si un condicional es verdadero. El 2° ciclo ejecuta NOP.

PROGRAMACIÓN

El lenguaje de programación usado para desarrollar el código fuente de la aplicación es el *assembly*. Puede ser escrito usando cualquier editor de texto ASCII.

CÓDIGO FUENTE

Cada línea de código fuente puede contener hasta cuatro tipos de información.

- Etiquetas
- Mnemónicos, Directivas y Macros
- Operandos
- Comentarios

El orden y la posición son importantes. Se recomienda que las etiquetas comiencen en la columna uno y los mnemónicos en la columna dos o posterior. Los operandos van seguidos a los mnemónicos. Y a continuación los comentarios, que pueden comenzar en cualquier columna. El ancho máximo de columna es de 255 caracteres.

Las etiquetas y los mnemónicos deben estar separados por dos puntos (:) o espacio en blanco. Los mnemónicos y los operandos deben estar separados por espacio en blanco. En el caso de haber múltiples operandos, los mismos deben estar separados por comas. El espacio en blanco es uno o más espacios o tabulaciones. Los comentarios deben comenzar con punto y coma (;).

```
                Mnemónicos
                Directivas
Etiquetas       Macros      Operandos       Comentarios
   |               |            |               |
                list        p=16f84a
                #include    p16f84a.inc
Dest            equ         0x0B            ;Definir una constante
                org         0x00            ;Reiniciar vector
                goto        Inicio
                org         0x20            ;Comenzar programa
Inicio
                movlw 0x0A
                movwf Dest
                bcf         Dest, 3         ;Esta línea usa 2 operandos
                goto        Inicio
                end
```

ETIQUETAS

Una etiqueta se usa para representar una línea o grupo de código, o un valor constante. Las instrucciones de salto necesitan indefectiblemente de etiquetas. Las etiquetas deben comenzar en la columna 1. Deben estar seguidas por dos puntos (:), espacio, tabulación, o fin de línea. Pueden tener hasta 32 caracteres de longitud y no pueden comenzar con un número.

MNEMÓNICOS, DIRECTIVAS Y MACROS

Los mnemónicos le indican al ensamblador qué instrucciones de máquina ensamblar. Por ejemplo, suma (`add`), saltos (`goto`) o movimientos (`movwf`). Al contrario de las etiquetas, que son creadas por el programador, los mnemónicos son provistos por el lenguaje assembly. Los mnemónicos no son *case sensitive*, pueden escribirse en minúsculas o mayúsculas indistintamente.

Las directivas son comandos del ensamblador que aparecen en el código fuente pero normalmente no se traducen de forma directa en *opcodes*. Se usan para controlar el ensamblador: su entrada, salida y la asignación de datos. Las directivas tampoco son *case sensitive*.

Las macros son conjuntos de instrucciones y directivas definidos por el usuario que permiten aislar y nombrar fragmentos de código que se repite para reutilizarlo más fácilmente en el programa.

OPERANDOS

Los operandos le proveen a la instrucción información sobre los datos que deben ser usados y la ubicación de almacenamiento para el resultado.

Los operandos deben estar separados de los mnemónicos por uno o más espacios, o por tabulaciones. En caso de haber dos operandos, deben estar separados entre ellos por comas.

COMENTARIOS

Los comentarios son texto que explica la operación de una o varias líneas de código.

El ensamblador MPASM trata como comentario a todo lo que siga a un punto y coma (;). Todos los caracteres que siguen a un punto y coma son ignorados hasta el final del final de la línea. Los strings o cadenas de texto pueden contener punto y coma sin ser confundidos con comentarios.

ARCHIVO INCLUDE (.INC)

Un archivo *include*, o cabecera (*header*) es un archivo de código assembly válido que usualmente contiene asignaciones de registros y bits específicos a un dispositivo. Este archivo puede ser "incluido" en el código de manera que puede ser reutilizado por muchos programas.

Por ejemplo, para agregar el archivo de cabecera estándar para el PIC16F84A al código assembly propio, se debe usar:

```
#include    p16f84a.inc
```

CONSTANTES NUMÉRICAS Y BASE

El ensamblador MPASM soporta las siguientes bases para las constantes: hexadecimal, decimal, octal, binaria y ASCII, La base por defecto, asignada a las constantes sin descriptor de base explícito, es la hexadecimal.

Base	**Sintaxis**	**Ejemplo**
Binaria	`B'dígitos_binarios'`	`B'00111001'`
Octal	`O'dígitos_octales'`	`O'777'`
Decimal	`D'dígitos'` `.dígitos`	`D'100'` `.100`
Hexadecimal	`H'dígitos_hexa'` `0xdígitos_hexa`	`H'9f'` `0x9f`
ASCII	`A'caracter'` `'character'`	`A'C'` `'C'`

DIRECTIVAS

Las directivas son comandos del ensamblador que aparecen en el código fuente pero que usualmente no se traducen directamente en *opcodes*. Se usan para controlar el ensamblador: su entrada, su salida y la asignación de datos.

org – ESTABLECER EL ORIGEN DEL PROGRAMA

Sintaxis

```
[label] org expr
```

Descripción

Establece el origen del programa para el código subsiguiente a la dirección definida en `expr`. Si no se especifica `org` en un programa, la generación de código comenzará a partir de la dirección 0.

end – FINALIZAR EL BLOQUE DEL PROGRAMA

Sintaxis

```
end
```

Descripción

Indica el fin del programa. Esta directiva debe estar siempre presente una única vez y al final del código fuente.

Ejemplo

```
#include p18f452.inc

:

; código ejecutable

:

;

end ; fin de instrucciones
```

equ – DEFINIR UNA CONSTANTE

Sintaxis

```
label equ expr
```

Descripción

El valor de `expr` es asignado a la etiqueta `label`. Comúnmente se usa para asignarle un nombre de variable a una dirección de memoria RAM.

Ejemplo

```
cuatro equ 4 ; se asigna el valor numérico 4 a la etiqueta cuatro
```

cblock – DEFINIR UN BLOQUE DE CONSTANTES

Sintaxis

```
cblock [ expr ]

label[:increment][,label[:increment]]

endc
```

Descripción

Define una lista de símbolos secuenciales. El propósito de esta directiva es asignar desplazamientos de direcciones a muchas etiquetas. La lista de nombres termina cuando se encuentra una directiva `endc`.

`expr` indica el valor de comienzo para el primer nombre en el bloque. Si no hay ninguna expresión, el primer nombre recibirá el valor siguiente al nombre final en el `cblock` previo. Si el primer `cblock` en el archivo fuente no tiene `expr`, los valores asignados comienzan en 0.

Si se especifica `increment`, a la próxima etiqueta se le asigna el valor de la etiqueta previa más el incremento. Pueden especificarse múltiples nombres en una línea, separados por comas.

`cblock` es útil para definir constantes en un programa.

Ejemplo simple

```
cblock 0x20             ; a nombre_1 se le asigna 20.
  nombre_1, nombre_2    ; a nombre_2, 21 y así sucesivamente
  nombre_3, nombre_4    ; a nombre_4 se le asigna 23
  WordVar: 0, WordHigh, WordLow    ; WordVar=24 WordHigh=24 WordLow=25
  perimetro:2           ; perimetro=26
  largo                 ; largo=28
  ancho                 ; ancho=29
endc
```