

Curso del microcontrolador PIC12F509

Este modelo de microcontrolador es el mas sencillo de todos los modelos y por consiguiente la explicación resultará más sencilla que la de algún otro modelo mas complejo.

El PIC12F509 es un dispositivo programable. Su fabricante define previamente un conjunto de instrucciones que el dispositivo sabe ejecutar. El usuario no inventa nuevas instrucciones; construye programas utilizando esas instrucciones.

El PIC12F509 pertenece a la arquitectura PIC baseline de 12 bits. Su conjunto de instrucciones es de **33 instrucciones**.

INTRODUCCIÓN

En este curso vamos a estudiar el funcionamiento del microcontrolador **PIC12F509** desde sus conceptos más básicos hasta llegar a la ejecución de programas completos.

Pero no queremos aprender únicamente a escribir programas. Nuestro objetivo principal será comprender:

¿Qué ocurre dentro del PIC cuando ejecuta una instrucción?

Para conseguirlo, estudiaremos primero la arquitectura del microcontrolador y posteriormente construiremos un **simulador didáctico del PIC12F509**.

¿Qué es un microcontrolador?

Un **microcontrolador** es un circuito integrado que contiene los elementos necesarios para ejecutar un programa y controlar un sistema electrónico.

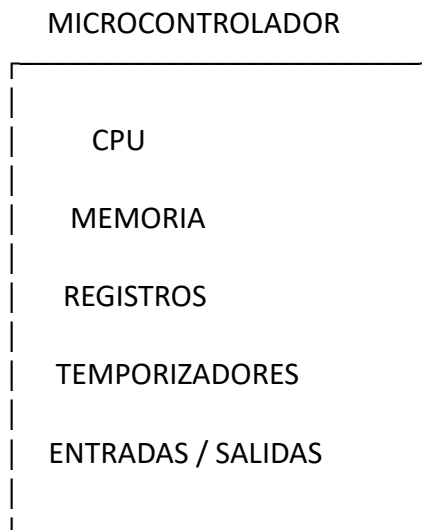
Podemos imaginarlo como una pequeña computadora especializada.

Una computadora convencional contiene, entre muchos otros elementos:

- procesador;
- memoria;
- entradas;
- salidas;
- temporizadores;
- dispositivos de comunicación.

Un microcontrolador integra muchos de estos elementos dentro de un solo circuito integrado.

Podemos representarlo de manera simplificada:



Por eso un microcontrolador puede recibir información, procesarla de acuerdo con un programa y producir una respuesta.

¿Qué significa PIC?

PIC es el nombre utilizado por Microchip para una familia de microcontroladores.

El dispositivo que estudiaremos es: **PIC12F509**

Cada parte de este nombre identifica una familia y características del dispositivo.

No es necesario memorizar ahora el significado exacto de cada carácter. Lo importante es reconocer que el **PIC12F509 es un microcontrolador de 8 bits de la familia PIC.**

Microchip lo clasifica dentro de los PIC de arquitectura **baseline**, que utilizan instrucciones de 12 bits.

El PIC12F509 es un dispositivo programable

Esto significa que podemos introducir en su memoria un programa que determina las operaciones que debe realizar.

Por ejemplo, podemos crear un programa que diga, conceptualmente:

1. Cargar un valor.
2. Guardarlo en una posición de memoria.
3. Realizar una operación.
4. Encender una salida.
5. Esperar.
6. Repetir.

El PIC ejecutará estas operaciones siguiendo las instrucciones que forman el programa.

¿Puede el usuario inventar instrucciones nuevas?

No. El fabricante del microcontrolador diseñó la CPU para reconocer un conjunto determinado de instrucciones. Esas instrucciones forman lo que se denomina:

CONJUNTO DE INSTRUCCIONES

En inglés: **Instruction Set**

El PIC12F509 dispone de un conjunto de **33 instrucciones**.

Por lo tanto, podemos decir:

El PIC12F509 ya viene diseñado para reconocer 33 tipos de instrucciones.

Nosotros construimos nuestros programas utilizando esas instrucciones.

Por ejemplo:

MOVLW 05h

MOVLW no es una instrucción que nosotros hayamos inventado.

Es una instrucción que la CPU del PIC12F509 ya está diseñada para reconocer.

¿Qué es una instrucción?

Una **instrucción** es una orden que la CPU puede interpretar y ejecutar.

Por ejemplo:

MOVLW 05h

significa:

Colocar el valor 05h en el registro W.

Otra instrucción:

CLRW

significa:

Borrar el registro W.

Otra:

GOTO 020h

significa:

Continuar la ejecución del programa en la dirección 020h.

Por lo tanto, una instrucción puede modificar registros, realizar operaciones, cambiar el flujo del programa o controlar determinadas funciones del microcontrolador.

¿Qué es un programa?

Un **programa** es una secuencia ordenada de instrucciones.

Por ejemplo:

MOVLW 05h

MOVWF 08h

CLRW

GOTO 000h

Aquí tenemos varias instrucciones colocadas una después de otra.

La CPU no ejecuta todas simultáneamente.

Las va ejecutando siguiendo la secuencia determinada por el programa y por el contador de programa.

Podemos imaginarlo así:

INSTRUCCIÓN 1
↓
INSTRUCCIÓN 2
↓
INSTRUCCIÓN 3
↓
INSTRUCCIÓN 4
↓
INSTRUCCIÓN 5
↓
...

¿Qué es la memoria de programa?

El programa tiene que estar almacenado en algún lugar.

El PIC12F509 dispone de memoria Flash para almacenar el programa.

Su memoria de programa es de:

1K × 12 bits

Esto significa que dispone de hasta 1024 posiciones de programa, con una palabra de instrucción de 12 bits.

Podemos representarla de manera simplificada:

DIRECCIÓN	INSTRUCCIÓN
000	MOVLW 05h
001	MOVWF 08h
002	CLRW
003	GOTO 000h
004	...
005	...
...	
3FF	...

Más adelante veremos exactamente cómo se codifica cada instrucción en esos 12 bits.

¿Qué significa que una instrucción tenga 12 bits?

El PIC12F509 pertenece a una arquitectura que utiliza instrucciones de **12 bits**.

Por ejemplo, una instrucción no se almacena internamente como el texto:

MOVLW 05h

Ese texto es una representación que nosotros podemos leer.

Internamente existe un código binario.

Por ejemplo, conceptualmente:

MOVLW 05h

Equivale a los siguientes 12 bits:

1100 0000 0101

La CPU recibe esos bits, los interpreta y determina qué operación debe realizar.

Por eso existe una diferencia entre:

Lenguaje que entendemos nosotros

MOVLW 05h

y

Código binario que entiende la CPU

1100xxxxxxx

El ensamblador se encarga de convertir la representación que escribimos nosotros en el código que necesita el microcontrolador.

¿Qué es lenguaje ensamblador?

El lenguaje ensamblador es una forma de escribir instrucciones utilizando palabras o abreviaturas que representan las operaciones de la CPU.

Por ejemplo:

MOVLW

MOVWF

ADDWF

GOTO

CALL

Estas palabras se llaman **mnemónicos**.

Un mnemónico es simplemente una palabra corta que representa una instrucción.

Por ejemplo:

MOVLW

representa:

Move Literal to W

es decir:

Mover un valor literal al registro W.

¿Qué es un ensamblador?

El **ensamblador** es un programa que traduce el código escrito en lenguaje ensamblador a los códigos binarios que el microcontrolador puede ejecutar.

El proceso puede representarse así:

PROGRAMA DEL USUARIO

MOVLW 05h
MOVWF 08h
GOTO 000h



ENSAMBLADOR



CÓDIGOS BINARIOS

12 bits
12 bits
12 bits

Por lo tanto:

Programador → ensamblador → código máquina → PIC

¿Qué es código máquina?

El **código máquina** es la información binaria que la CPU puede interpretar directamente.

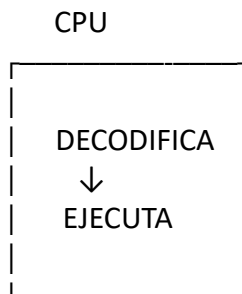
¿Qué es la CPU?

CPU significa: **Central Processing Unit**

En español: **Unidad Central de Procesamiento.**

Es la parte del microcontrolador encargada de interpretar y ejecutar las instrucciones.

De manera simplificada:



La CPU es el elemento que estudiaremos con mayor profundidad.

¿Qué es la ALU?

ALU significa: **Arithmetic Logic Unit**

En español: **Unidad Aritmético-Lógica.**

Es la parte de la CPU que realiza determinadas operaciones aritméticas y lógicas.

Por ejemplo:

SUMA

$$\begin{array}{r} 03 \\ + 02 \\ \hline 05 \end{array}$$

Y operaciones lógicas como:

AND
OR
XOR

En el PIC12F509, la ALU trabaja con datos de 8 bits.

Por eso veremos frecuentemente valores como:

00000000

00000001

00000101

11111111

¿Qué es un registro?

Un **registro** es una pequeña zona de almacenamiento interna del microcontrolador.

Podemos imaginarlo como una pequeña caja capaz de guardar información.

Por ejemplo:

REGISTRO

0 0 0 0 1 0 1

Cada registro puede tener una función determinada.

Algunos contienen datos.

Otros controlan funciones del microcontrolador.

Otros contienen información sobre el estado de la CPU.

¿Qué es el registro W?

Uno de los registros más importantes del PIC12F509 es:

W

W es el registro de trabajo de la CPU.

Muchas operaciones utilizan este registro.

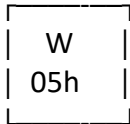
Por ejemplo:

MOVLW 05h

produce:

W = 05h

Podemos imaginar:



Durante el curso aprenderemos que W aparece constantemente en las instrucciones del PIC12F509.

¿Qué es STATUS?

STATUS es un registro que contiene información sobre determinadas condiciones del procesador y sobre los resultados de algunas operaciones.

Entre sus bits encontramos:

Z

DC

C

y otros bits relacionados con el estado del microcontrolador.

Por ejemplo, el bit:

Z — Zero

indica si el resultado de determinadas operaciones fue cero.
Más adelante veremos cada bit cuidadosamente.

¿Qué es el PC?

PC significa: **Program Counter**

En español: **Contador de Programa.**

Su función fundamental es indicar la posición de memoria de programa que corresponde a la siguiente instrucción que se ejecutará.

Podemos imaginar:

MEMORIA

000 MOVLW 05h

001 MOVWF 08h

002 CLRW

003 GOTO 000h

Si:

PC = 000

la CPU obtiene la instrucción de la posición **000**.

Después el flujo continúa.

El PC será uno de los elementos principales de nuestro simulador.

¿Qué es RAM?

RAM significa: **Random Access Memory**

En español: **Memoria de Acceso Aleatorio.**

En términos sencillos:

Es una memoria que el microcontrolador utiliza para almacenar datos durante la ejecución del programa.

El PIC12F509 dispone de **41 bytes de SRAM.**

Podemos imaginarla:

MEMORIA RAM

00 — dato

01 — dato

02 — dato

03 — dato

04 — dato

...

Algunas posiciones están asociadas a registros de función especial.

¿Qué son los registros de función especial?

Son registros que no solamente sirven para guardar números.

Controlan o representan funciones específicas del microcontrolador.

Por ejemplo, existen registros relacionados con:

- GPIO;
- TMR0;
- OPTION;
- TRIS;
- STATUS.
-

Cuando nuestro programa escribe un valor en uno de estos registros, puede provocar una acción en el hardware.

Por ejemplo, modificar un registro relacionado con las entradas/salidas puede cambiar el estado de los pines del PIC.

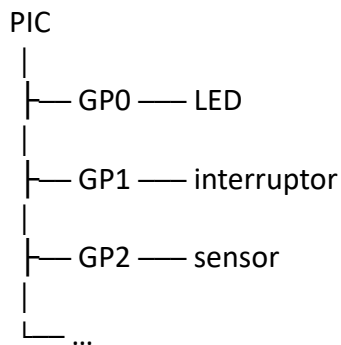
¿Qué es GPIO?

GPIO significa: **General Purpose Input/Output**

En español: **Entrada/Salida de Propósito General.**

Son las líneas que permiten que el microcontrolador se comuniqué con el mundo exterior.

Por ejemplo:



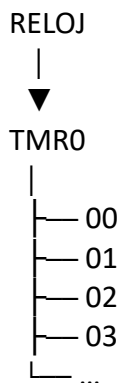
Una línea GPIO puede utilizarse como entrada o salida, dependiendo de su configuración y de las funciones disponibles en el dispositivo.

¿Qué es TMR0?

TMR0 significa: **Timer 0**

Es un temporizador/contador de 8 bits.

Su función será estudiada posteriormente, pero podemos entenderlo inicialmente como un registro que puede cambiar automáticamente siguiendo determinados eventos del reloj.



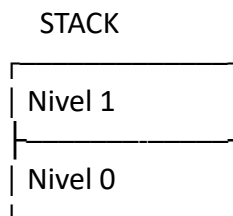
Más adelante estudiaremos cómo se utiliza para generar temporizaciones.

¿Qué es la pila o STACK?

La **pila**, o **stack**, es una pequeña memoria utilizada por el procesador para guardar direcciones relacionadas principalmente con las llamadas a subrutinas.

El PIC12F509 dispone de una pila de hardware de **2 niveles**.

Conceptualmente:



La estudiaremos cuando lleguemos a:

CALL
RETLW

¿Qué es un bit?

Un **bit** es la unidad más pequeña de información digital.
Solamente puede tener dos estados:

0 (cero)

o

1 (uno)

Por ejemplo:

10110100

contiene 8 bits.

Los bits son fundamentales porque todo lo que ocurre dentro del microcontrolador termina representándose mediante ceros y unos.

¿Qué es un byte?

Un **byte** está formado por 8 bits.

1 byte = 8 bits

Por ejemplo:

10110100
es un byte.

Como 8 bits pueden representar 256 combinaciones diferentes:

00000000 = 00h
00000001 = 01h
00000010 = 02h
...
11111111 = Ffh

Por eso un registro de 8 bits puede representar valores desde:
0 hasta **255**

¿Qué es hexadecimal?

Durante el curso utilizaremos constantemente números hexadecimales.

El sistema hexadecimal utiliza 16 símbolos:
0 1 2 3 4 5 6 7 8 9 A B C D E F

Por ejemplo:

10 decimal = 0Ah
15 decimal = 0Fh
16 decimal = 10h
255 decimal = FFh

La letra h indica que estamos expresando el número en hexadecimal.

El hexadecimal es especialmente cómodo para trabajar con microcontroladores porque cada dígito hexadecimal representa exactamente cuatro bits.

¿Qué es el reloj?

La CPU necesita una referencia temporal para coordinar sus operaciones.
Esta referencia se obtiene mediante el **reloj**.

Podemos imaginarlo como una señal que marca el ritmo:
El reloj permite sincronizar la ejecución de las instrucciones.

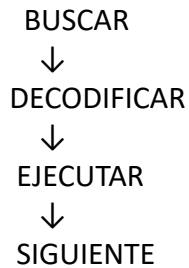
Más adelante estudiaremos el oscilador y cómo se relaciona con el tiempo de ejecución de las instrucciones.

¿Qué es un ciclo de instrucción?

Una instrucción no aparece mágicamente ejecutada.

La CPU realiza una secuencia interna para obtenerla, interpretarla y ejecutarla.

De manera simplificada podemos representarlo como:



Estos conceptos reciben nombres como:

Fetch

Decode

Execute

Los estudiaremos con mucho más detalle en el módulo de arquitectura.

Las 33 instrucciones del PIC12F509

Ahora sí podemos presentar la lista completa de instrucciones.

Microchip las organiza en tres grandes grupos.

A. Operaciones orientadas a registros

Instrucción	Significado sencillo
ADDWF f,d	Suma W con un registro
ANDWF f,d	AND lógico entre W y un registro
CLRF f	Borra un registro
CLRW	Borra W
COMF f,d	Complementa un registro
DECF f,d	Decrementa un registro
DECFSZ f,d	Decrementa y salta si llega a cero
INCF f,d	Incrementa un registro
INCFSZ f,d	Incrementa y salta si llega a cero
IORWF f,d	OR lógico entre W y un registro
MOVF f,d	Mueve el contenido de un registro

MOVWF f	Mueve W a un registro
NOP	No realiza ninguna operación
RLF f,d	Rotación a la izquierda mediante Carry
RRF f,d	Rotación a la derecha mediante Carry
SUBWF f,d	Resta W de un registro
SWAPF f,d	Intercambia los grupos alto y bajo de 4 bits
XORWF f,d	XOR lógico entre W y un registro

B. Operaciones orientadas a bits

Instrucción	Significado sencillo
BCF f,b	Pone un bit en 0
BSF f,b	Pone un bit en 1
BTFSC f,b	Comprueba un bit y salta si está en 0
BTFSS f,b	Comprueba un bit y salta si está en 1

C. Literales y control

Instrucción	Significado sencillo
ANDLW k	AND entre W y un valor literal
CALL k	Llama a una subrutina
CLRWDT	Borra el Watchdog Timer
GOTO k	Salta a otra posición del programa
IORLW k	OR entre W y un valor literal
MOVLW k	Carga un valor literal en W
OPTION	Carga el registro OPTION
RETLW k	Regresa de una subrutina colocando un literal en W
SLEEP	Coloca al dispositivo en modo de bajo consumo
TRIS f	Carga el registro de dirección de E/S
XORLW k	XOR entre W y un valor literal

Estas tres categorías suman las **33 instrucciones** del PIC12F509.

¿Qué significan las letras f, d, b y k?

Al principio la lista puede parecer complicada.

Por ejemplo:

ADDWF f,d

No debemos aprender todavía la instrucción de memoria.
Primero debemos entender la nomenclatura.

f

f representa un **registro o posición de memoria de datos**.

d

d indica dónde se coloca el resultado.

En las instrucciones que utilizan **d**, posteriormente estudiaremos las opciones que permiten determinar si el resultado termina en **W** o en el **registro de origen**.

b

b representa un **número de bit**.

Por ejemplo:

BSF f,3

significa conceptualmente:

Poner a 1 el bit número 3 del registro indicado.

k

k representa un **valor literal**.

Por ejemplo:

MOVLW 25h

Aquí:

k = 25h

Es decir, 25 hexadecimal es el valor que queremos cargar.

¿Qué es un valor literal?

Un literal es un valor escrito directamente dentro de la instrucción.

Por ejemplo:

MOVLW 25h

El valor: 25h es un literal.

La diferencia sería:

MOVLW 25h

El valor está dentro de la instrucción.

Mientras que:

MOVF 10h,W

obtiene información de una posición de memoria.

Esta diferencia será muy importante cuando estudiemos el direccionamiento.

Una idea fundamental para todo el curso

Debemos distinguir perfectamente tres cosas:

El hardware

Es el circuito físico del PIC12F509.

Las instrucciones

Son las operaciones que la CPU está diseñada para reconocer.

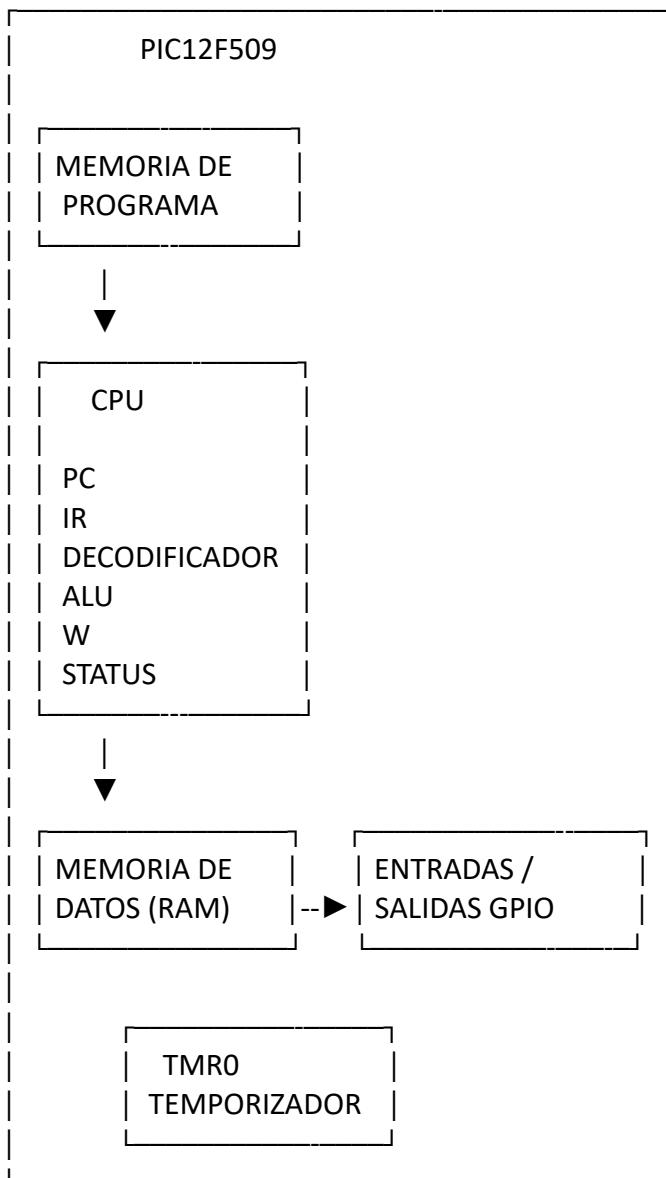
El programa

Es la secuencia de instrucciones que nosotros elegimos para resolver un problema.

MÓDULO 2 — ARQUITECTURA INTERNA DEL PIC12F509

El mapa interno del PIC12F509

Antes de estudiar cada bloque por separado, vamos a observar el conjunto.
La arquitectura puede entenderse mediante cuatro grandes zonas:



No debemos interpretar este dibujo como si todos los bloques fueran independientes.

Están comunicados entre sí.

Y precisamente eso es lo que vamos a estudiar.

Dos tipos de memoria

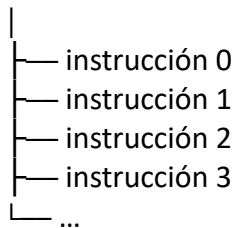
Una de las primeras características importantes del PIC12F509 es que utiliza una arquitectura en la que la **memoria de programa y la memoria de datos están separadas**.

Tenemos:

Memoria de programa

Contiene las instrucciones.

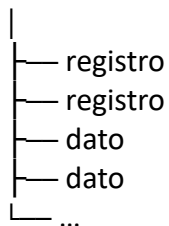
MEMORIA DE PROGRAMA



Memoria de datos

Contiene datos y registros utilizados durante la ejecución.

MEMORIA DE DATOS



Esto es muy importante porque:

El lugar donde está el programa no es el mismo lugar donde se almacenan los datos que utiliza el programa.

Memoria de programa

El PIC12F509 dispone de:

1K × 12 bits de memoria Flash de programa.

Es decir:

1024 posiciones

×

12 bits por posición

Podemos representarlo así:

Dirección	Contenido
000	_____ 12 bits
001	_____ 12 bits
002	_____ 12 bits
003	_____ 12 bits
004	_____ 12 bits
...	
3FF	_____ 12 bits

Por ahora no nos preocuparemos por conocer los bits individuales.

Lo importante es comprender:
cada posición contiene una instrucción codificada en 12 bits.

¿Quién selecciona la posición?

Aquí aparece uno de los registros más importantes:

PC — Program Counter

El contador de programa indica dónde se encuentra la instrucción que corresponde al flujo de ejecución.

Imaginemos:

PC = 002

La CPU buscará la instrucción almacenada en:
MEMORIA[002]

Por ejemplo:

```
000  MOVLW 05h
001  MOVWF 08h
002  CLRW
003  GOTO 000h
```

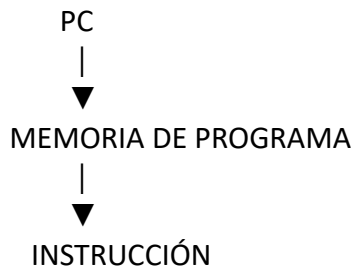
Si:

PC = 002

la instrucción obtenida será:
CLRW

Primera relación importante

Ya podemos establecer nuestra primera cadena:



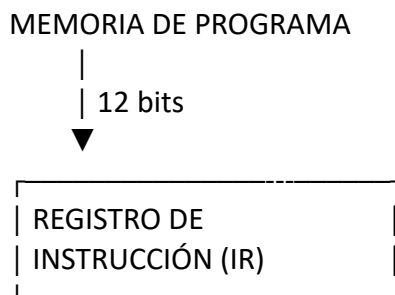
Registro de instrucción

Una vez obtenida la instrucción, necesitamos que la CPU la tenga disponible para interpretarla. Aquí interviene el:

Instruction Register — IR

Registro de instrucción.

Podemos imaginar:



Por ejemplo:

PC = 000

la memoria proporciona una instrucción.
Esa instrucción pasa al IR.

Entonces:

IR = instrucción actual

¿Qué hacemos ahora con la instrucción?

Aquí aparece otro bloque:

Decodificador de instrucciones

Su trabajo es determinar **qué instrucción recibió la CPU**.

Por ejemplo, el programa puede contener:

MOVLW 05h

El decodificador reconoce que se trata de:

MOVLW

y no de:

ADDWF

ni:

GOTO

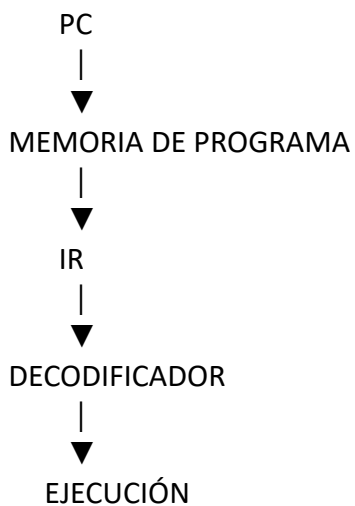
ni:

CLRW

Esto es posible porque cada instrucción tiene una codificación binaria específica.

La idea completa

Tenemos ahora:

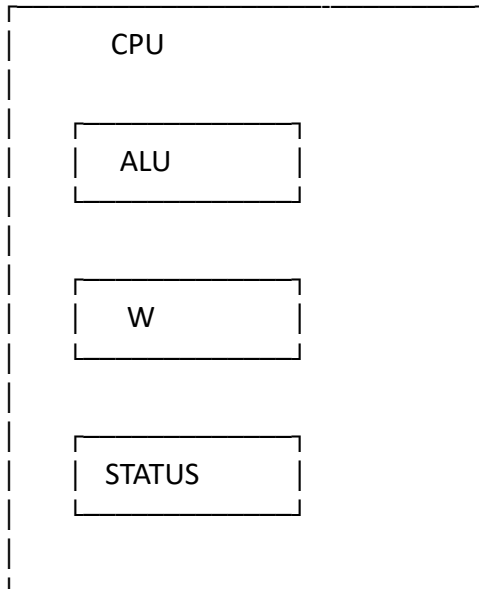


Este es el comienzo del ciclo de funcionamiento del PIC.

¿Quién ejecuta la operación?

La ejecución ocurre dentro de la CPU.

Dentro de ella encontramos elementos como:



Ahora ya tenemos suficientes elementos para comenzar a seguir una instrucción real.

Primera instrucción: MOVLW

Utilizaremos:

MOVLW 05h

Recordemos su significado:

Cargar el literal 05h en W.

Supongamos que antes de ejecutar tenemos:

PC = 000

W = 00h

Entonces sucede:

Paso 1

El PC indica:

000

Paso 2

Se obtiene de memoria la instrucción:
MOVLW 05h

Paso 3

La instrucción llega al IR.

Paso 4

El decodificador identifica:
MOVLW

Paso 5

La CPU ejecuta la operación.
05h → W

Paso 6

W cambia:
00h → 05h

Paso 7

El PC continúa con la siguiente posición.
Conceptualmente:
000 → 001

Esto nos lleva al siguiente concepto

Ya conocemos:

MEMORIA

↓

PC

↓

IR

↓

DECODIFICADOR

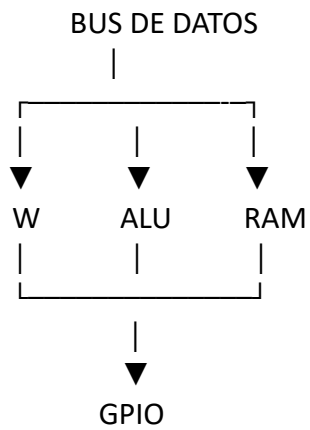
↓

CPU

Ahora falta comprender **cómo se mueven los datos entre estos elementos**.
Para eso estudiaremos el:

BUS DE DATOS

Y aquí veremos algo muy importante:



El bus es el "camino" por el que se trasladan los datos.

MÓDULO 2 — ARQUITECTURA INTERNA DEL PIC12F509

El bus de datos de 8 bits

Ya sabemos que dentro del PIC12F509 existen diferentes bloques:

CPU

ALU

registro W

registros de datos

STATUS

RAM

GPIO

TMR0

Pero ahora aparece una pregunta fundamental:

¿Cómo se comunican estos elementos entre sí?

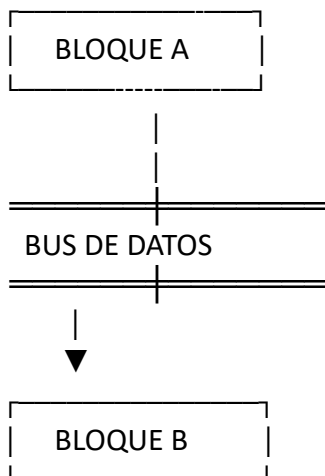
}La respuesta es:

mediante buses internos.

¿Qué es un bus?

Un bus es un conjunto de líneas que permite transportar información entre diferentes partes de un sistema electrónico.

Podemos imaginarlo como una carretera.



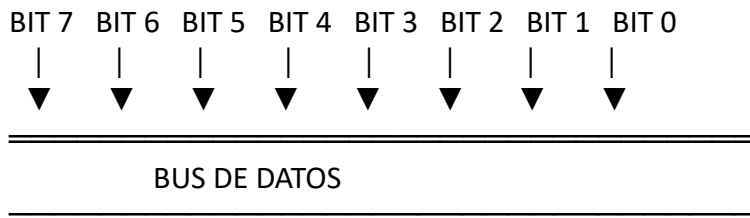
En una carretera circulan automóviles.

En un bus electrónico circulan **bits**.

Un bus de 8 bits

El PIC12F509 es un microcontrolador de 8 bits y posee una ALU y un registro W de 8 bits. Su arquitectura de datos utiliza palabras de 8 bits.

Por tanto, podemos imaginar el bus de datos como:



Son ocho líneas.

Cada una puede transportar un 0 o un 1.

Por ejemplo:

10100101

representa un dato de 8 bits.

En hexadecimal:

1010 0101

A 5

= A5h

¿Qué puede viajar por este bus?

Por el bus de datos pueden desplazarse valores entre los diferentes elementos que intervienen en una operación.

Por ejemplo:

W



ALU



Registro

o:

Registro



ALU



W

Dependiendo de la instrucción que se esté ejecutando.

Un ejemplo muy sencillo

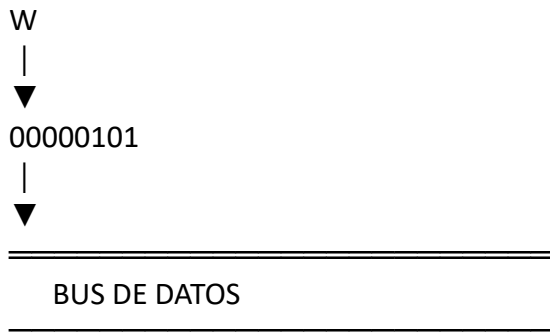
Supongamos:

W = 05h

En binario:

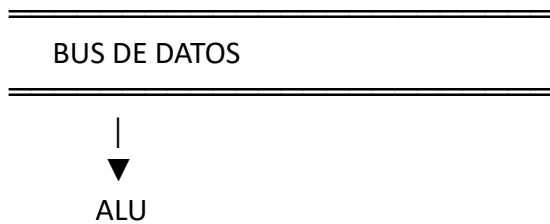
W = 00000101

Podemos imaginar que ese valor aparece sobre el bus:



Después otro bloque puede recibir ese valor.

Por ejemplo:



La ALU recibe los bits y puede utilizarlos en una operación.

El bus de datos NO es el bus de programa

Esta diferencia es fundamental.

El PIC12F509 utiliza una arquitectura **Harvard**.

Eso significa que la memoria de programa y la memoria de datos utilizan caminos separados.

Tenemos entonces:

PIC12F509

BUS DE PROGRAMA

12 bits



MEMORIA FLASH
DE PROGRAMA

BUS DE DATOS

8 bits



MEMORIA DE
DATOS / RAM

No debemos imaginar un único bus haciendo todo.
Hay **dos caminos diferentes**.

¿Por qué el bus de programa tiene 12 bits?

Esta es una característica muy interesante del PIC12F509.
Sus instrucciones tienen una palabra de **12 bits**.

Por eso el camino que lleva la instrucción desde la memoria de programa hacia la CPU tiene 12 bits. Microchip indica que el acceso a la memoria de programa es mediante un bus de 12 bits.

Podemos representarlo:

MEMORIA FLASH



12 bits



INSTRUCCIÓN
12 bits



DECODIFICADOR

En cambio, los datos son de 8 bits

Los datos que maneja la CPU son de 8 bits.

Por ejemplo:

05h

equivale a:

00000101

Por tanto:

MEMORIA DE DATOS

|

| 8 bits



BUS DE DATOS

|



CPU

Esto explica una característica que al principio puede resultar extraña:

El PIC12F509 utiliza instrucciones de 12 bits, pero trabaja con datos de 8 bits.

No es una contradicción.

Son dos cosas diferentes:

INSTRUCCIÓN → 12 bits

DATO → 8 bits

Hagamos una comparación

Imaginemos que tenemos:

MOVLW 05h

La instrucción completa ocupa una palabra de programa de 12 bits.

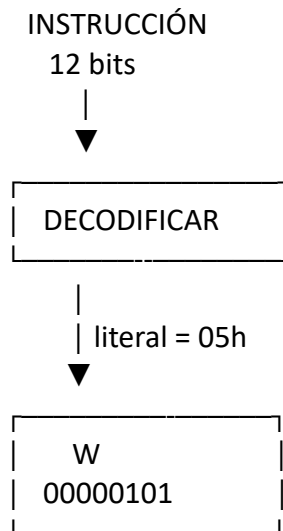
Pero el valor que queremos cargar en W es:

05h

y ese valor es de 8 bits:

00000101

Por lo tanto podemos imaginar:



Esto será muy importante cuando estudiemos cómo están codificadas las instrucciones.

El camino de una instrucción

Podemos seguir ahora una instrucción completa.

Supongamos:
MOVLW 05h

Paso 1 — PC

El contador de programa contiene la dirección de la instrucción.
PC = 000

Paso 2 — Memoria de programa

La dirección 000 selecciona una posición de la memoria Flash.
000 → MOVLW 05h

Paso 3 — Bus de programa

La instrucción de 12 bits viaja hacia la CPU.

MEMORIA

|
| 12 bits
▼

BUS DE PROGRAMA

Paso 4 — Registro de instrucción

La instrucción queda disponible para ser interpretada.
IR = MOVLW 05h

Paso 5 — Decodificador

El decodificador identifica:

MOVLW

y determina qué operación debe realizarse.

Paso 6 — Literal

Se obtiene:

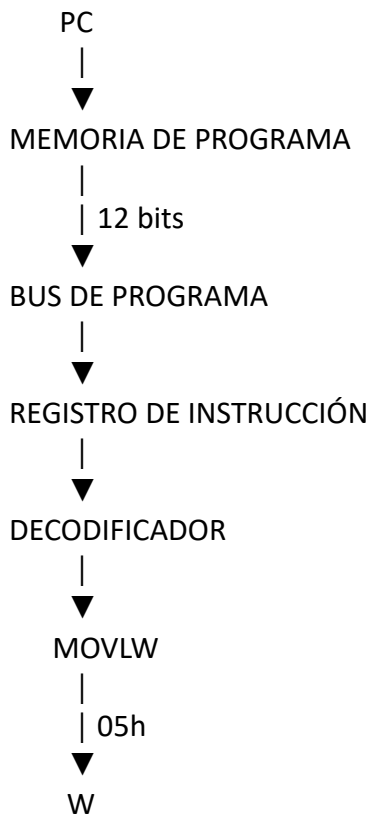
05h

Paso 7 — W

El valor se carga en W:

W = 05h

El recorrido completo puede representarse así:



Ahora veamos una operación con la ALU

Supongamos que tenemos:

W = 03h

y queremos realizar una suma con otro dato:

02h

La operación será:

03h
+ 02h

05h

En binario:

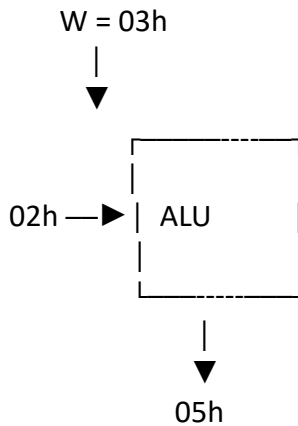
00000011

+

00000010

00000101

La ALU recibe los operandos y realiza la suma.



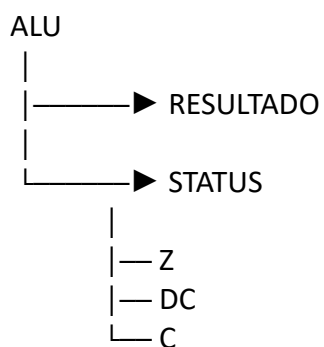
La documentación de Microchip indica que la ALU de estos dispositivos realiza operaciones aritméticas y booleanas, incluyendo suma, resta, desplazamientos y operaciones lógicas.

La ALU también puede modificar STATUS

Aquí aparece una relación muy importante.

La ALU no solamente produce un resultado.

Dependiendo de la instrucción, puede modificar determinados bits del registro STATUS:



Por ejemplo, si una operación produce:

00000000

puede establecerse:

Z = 1

El bit Z significa que el resultado fue cero.

Los bits C y DC también pueden verse afectados por determinadas operaciones.

Esto será fundamental cuando lleguemos a las instrucciones:

ADDWF
SUBWF
INCF
DECF
y otras.

El bus y la RAM

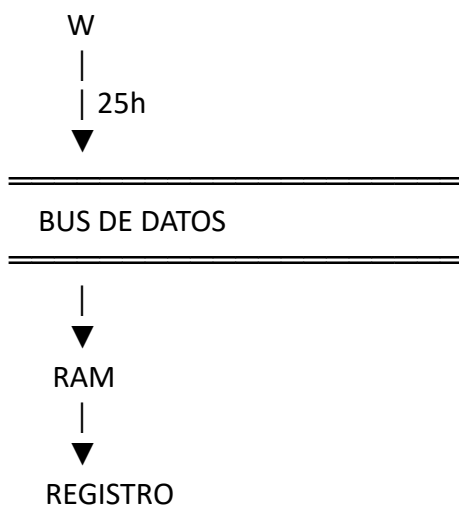
Ahora podemos imaginar una transferencia entre W y un registro de memoria.

Por ejemplo, conceptualmente:

W = 25h

y queremos almacenar ese valor en una posición de RAM.

El camino puede representarse:



Después:

RAM = 25h

Esto es precisamente el tipo de operación que veremos con:

MOVWF

¿Y qué ocurre con GPIO?

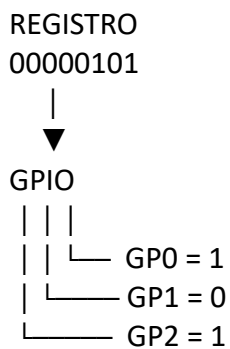
Los puertos de entrada/salida están relacionados con registros internos.

Por eso podemos tener conceptualmente:



Si el programa coloca determinados bits en el registro correspondiente, esos bits pueden reflejarse en las salidas GPIO cuando el puerto está configurado como salida.

Por ejemplo, conceptualmente:



Más adelante estudiaremos exactamente cómo funciona GPIO y qué registros intervienen.

El bus como una carretera

Una analogía que utilizaremos durante todo el curso:



El bus sería la carretera.

Los bloques serían ciudades.

Los datos serían los vehículos.

Pero existe una diferencia fundamental:

No todos los bloques pueden enviar información al mismo tiempo por el mismo camino.

La CPU controla cuándo un elemento entrega información y cuándo otro la recibe.

Esto será especialmente importante cuando estudiemos el funcionamiento interno de una instrucción.

El PIC12F509 utiliza un **pipeline de dos etapas** que permite superponer la búsqueda de una instrucción con la ejecución de otra. Microchip indica que, debido a esta arquitectura, las instrucciones normales se ejecutan en un ciclo, mientras que las instrucciones de salto constituyen una excepción.

La idea puede visualizarse así:

CICLO 1

FETCH

buscar instrucción A

CICLO 2

FETCH

buscar instrucción B

EXECUTE

ejecutar instrucción A

CICLO 3

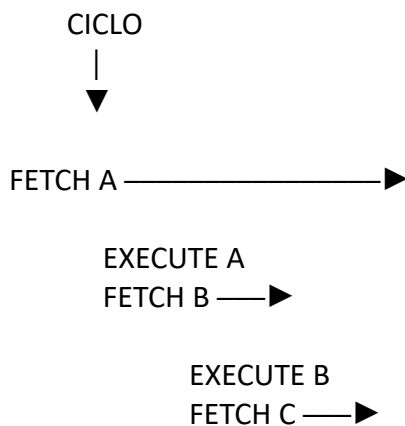
FETCH

buscar instrucción C

EXECUTE

ejecutar instrucción B

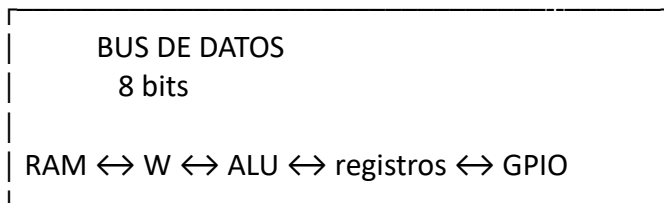
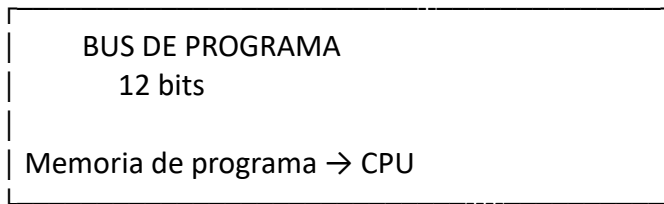
Es decir:



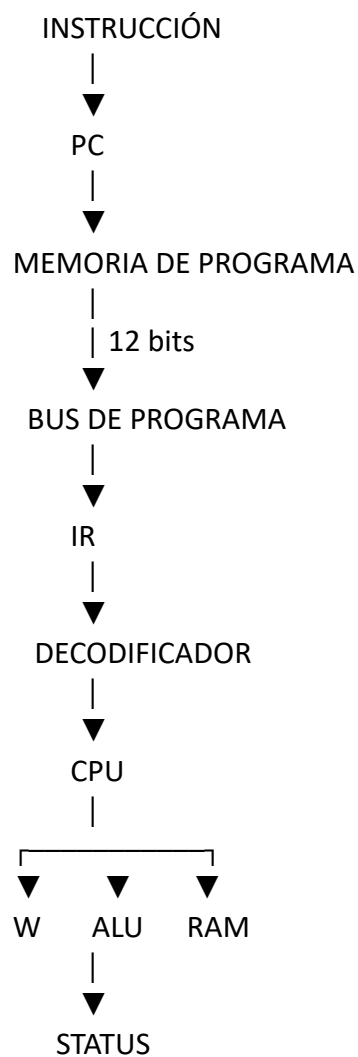
Y posteriormente, si queremos profundizar, podremos incorporar el pipeline.

Resumen de esta sección

Ya podemos establecer una diferencia fundamental:

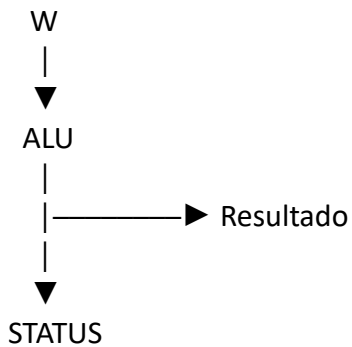


Y podemos resumir el recorrido de una instrucción:



W + ALU + STATUS

Hasta ahora sabemos que existen tres elementos que trabajan muy estrechamente:



Ahora vamos a entender **qué ocurre realmente cuando la ALU realiza una operación.**

Recordemos qué es W

El registro **W** es un registro de trabajo de 8 bits.

Puede contener cualquier valor entre:

0 y 255 decimal

o:

00h y FFh

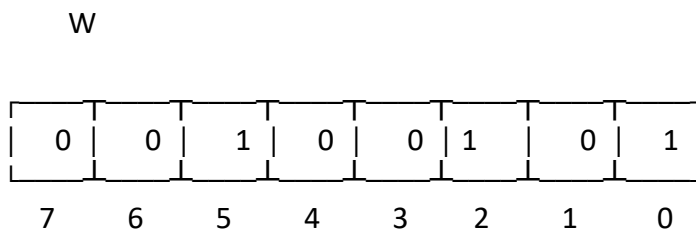
Por ejemplo:

W = 25h

En binario:

W = 0010 0101

Podemos visualizarlo:



Cada casilla es un **bit**.

¿Por qué W tiene 8 bits?

Porque el PIC12F509 es una arquitectura de datos de 8 bits.

Eso significa que la CPU trabaja fundamentalmente con grupos de:
8 bits

Por ejemplo:

00000000

00000001

00000010

...

11111111

El número máximo que podemos representar con 8 bits es:

$11111111_2 = 255 \text{ decimal} = \text{FFh}$

¿Qué ocurre si sumamos dos números?

Supongamos:

W = 03h

y queremos sumar:

02h

La operación matemática es:

03
+ 02
——
05

En binario:

00000011
+ 00000010
——
00000101

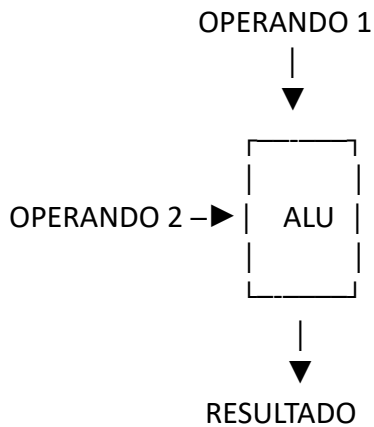
El resultado cabe perfectamente dentro de 8 bits.

Por tanto:

resultado = 05h

La ALU

La encargada de realizar esta operación es la **ALU**.
Podemos visualizarla:



La ALU recibe los datos, realiza la operación correspondiente y produce un resultado.

Pero hay algo más:

también puede producir información sobre cómo fue realizada la operación.

Esa información se refleja en determinados bits de STATUS.

El bit Carry — C

Comencemos con uno de los conceptos más importantes:

Carry

En español: **acarreo**.

Veamos una suma sencilla:

```
  00001111
+ 00000001
-----
  00010000
```

Aquí hubo un acarreo entre posiciones.

Pero para entender mejor el Carry del procesador debemos observar una situación en la que el resultado necesita **más de 8 bits**.

Una suma que supera 8 bits

Supongamos:

$255 + 1$

En hexadecimal:

$\text{FFh} + 01\text{h}$

Matemáticamente:

$\text{FFh} + 01\text{h} = 100\text{h}$

Pero:

100h

necesita **9 bits**:

$1\ 0000\ 0000$

El registro W solamente puede guardar 8 bits.

Por tanto:

$1\ 0000\ 0000$

↑

Acarreo

Los 8 bits inferiores son:

00000000

y el bit que sobra es el:

Carry = 1

Podemos representarlo:

C
|
▼
 $1\ 00000000$
└────────┘
8 bits

Por lo tanto:

Resultado = 00h

$C = 1$

¿Qué significa entonces C?

Podemos decir de forma sencilla:

El bit C indica que hubo un acarreo fuera del bit más significativo en una operación aritmética que lo modifica.

No debemos pensar que C es simplemente "el resultado es mayor que 255" en todos los casos, porque su comportamiento depende de la instrucción y, en las restas, su interpretación requiere especial cuidado.

Eso lo veremos cuando estudiemos SUBWF.

El bit Zero — Z

Ahora tenemos otro indicador.

Z = Zero

Indica que el resultado de determinadas operaciones fue cero.

Supongamos:

$$03h - 03h = 00h$$

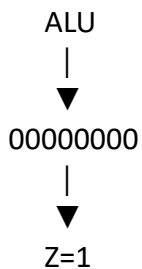
Resultado:

00000000

Entonces:

$$Z = 1$$

Visualmente:



Si el resultado NO es cero

Supongamos:

$$03h + 02h = 05h$$

Como:

$$05h \neq 00h$$

tenemos:

$$Z = 0$$

Por tanto:

$$\text{Resultado} = 05h$$

$$Z = 0$$

Una observación muy importante

El bit Z **no contiene el resultado**.

}

El resultado está en un registro o destino determinado.

Z solamente informa:

"El resultado de esta operación fue cero."

Por ejemplo:

Resultado = 00h

Z = 1

pero también:

Resultado = 05h

Z = 0

El bit Digit Carry — DC

Existe otro indicador:

DC — Digit Carry

Este bit está relacionado con el acarreo entre los **4 bits inferiores y los 4 bits superiores** de una operación.

Recordemos que un byte puede separarse en dos grupos:

0011 1010

alto bajo

En hexadecimal:

3^A

El dígito hexadecimal de la derecha corresponde a los cuatro bits inferiores.

Ejemplo de DC

Consideremos:

0Fh + 01h

En binario:

0000 1111

0000 0001

0001 0000

Observemos los cuatro bits inferiores:

1111

+

0001

0000

}

Se produce un acarreo hacia los cuatro bits superiores.

Por eso:

DC = 1

Y el resultado es:

10h

Los tres indicadores juntos

Ya podemos tener una visión más completa.

Supongamos:

0Fh

+ 01h

10h

Tenemos:

Resultado = 10h

Z = 0

DC = 1

C = 0

¿Por qué?

Z

El resultado no es cero:

10h $\neq$ 00h

Por tanto:

Z = 0

DC

Hubo acarreo de los 4 bits inferiores:

1111 + 0001

Por tanto:

DC = 1

C

No hubo acarreo fuera del bit 7.

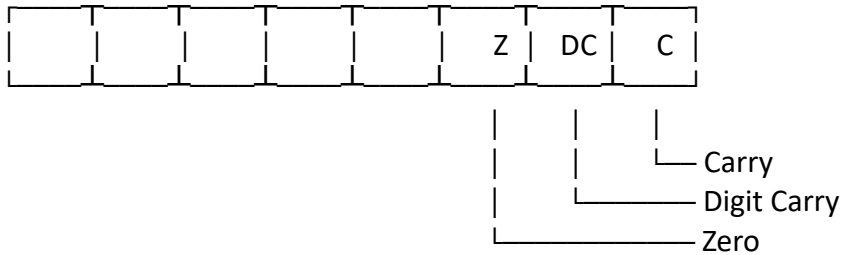
Por tanto:

$C = 0$

STATUS

Ahora podemos visualizar STATUS:

STATUS



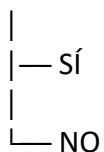
La posición exacta y el significado de todos los bits de STATUS los estudiaremos cuando analicemos el registro completo según la documentación del PIC12F509.

¿Por qué son útiles estos bits?

Porque permiten que el programa **tome decisiones**.

Por ejemplo:

¿El resultado fue cero?



Esto permite construir programas como:

restar dos números

↓

¿resultado = 0?

SÍ ↓ ↓ NO
hacer A hacer B

Esto será fundamental cuando estudiemos las instrucciones de prueba y salto:

BTFSC
BTFSS
DECFSZ
INCFSSZ

W, registro y ALU

Hasta ahora hemos hablado de W y de la ALU.

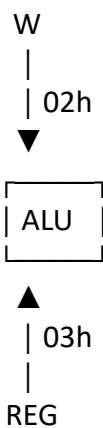
Pero muchas instrucciones trabajan también con un registro de memoria.

Por ejemplo, imaginemos:

W = 02h

REG = 03h

La ALU puede recibir:



La operación:

02h + 03h

produce:

05h

¿Dónde termina el resultado?

Aquí aparece una característica muy importante de las instrucciones del PIC. Algunas instrucciones permiten determinar si el resultado se almacena en:

W

o en:

el registro de origen

Esto se representa mediante la letra:

d

que vimos en la introducción.

Por ejemplo:

ADDWF f,d

No necesitamos memorizar todavía la sintaxis.

Lo importante es comprender que la CPU puede tener que decidir:

¿Dónde guardo el resultado?

Ejemplo conceptual

Supongamos:

$W = 02h$

$REG = 03h$

La ALU realiza:

$02h + 03h = 05h$

Podríamos tener conceptualmente:

Resultado $\rightarrow W$

o:

Resultado $\rightarrow REG$

Dependiendo de la instrucción y del bit de destino.

Algo todavía más interesante: la suma bit por bit

Para comprender realmente una ALU no basta con decir:

$03 + 02 = 05$

Vamos a abrirla y observar cómo puede realizarse una suma binaria.

Tomemos:

$03h = 00000011$

$02h = 00000010$

Comenzamos por el bit 0:

$1 + 0 = 1$

Bit 1:

$1 + 1 = 0$

pero aparece un acarreo:

Carry = 1

Bit 2:

$0 + 0 + \text{Carry}$

$= 1$

Los demás bits:

$0 + 0 = 0$

Resultado:

00000101

es decir:

05h

Podemos representarlo:

```
  0 0 0 0 0 1 1
+ 0 0 0 0 0 1 0
-----
  0 0 0 0 1 0 1
```

El ciclo de ejecución de una instrucción

Hasta ahora hemos visto que existen:

- memoria de programa;
- PC;
- registro de instrucción;
- decodificador;
- CPU;
- W;
- ALU;
- RAM;
- STATUS.

Ahora vamos a unir todos esos elementos.

La pregunta es:

¿Qué sucede dentro del PIC desde que encuentra una instrucción en la memoria hasta que termina de ejecutarla?

De forma simplificada, podemos dividir el proceso en tres etapas:

```
FETCH
↓
DECODE
↓
EXECUTE
```

En español:

```
BUSCAR
↓
DECODIFICAR
↓
EJECUTAR
```

FETCH — Buscar la instrucción

La primera pregunta que tiene que responder la CPU es:

¿Dónde está la siguiente instrucción?

La respuesta está en el **PC**.

Supongamos:

PC = 000

La CPU utiliza ese valor para acceder a la memoria de programa.

Tenemos:

MEMORIA DE PROGRAMA

Dirección	Instrucción
-----------	-------------

000	MOVLW 05h
001	MOVWF 08h
002	CLRW
003	GOTO 000h

Como:

PC = 000

se obtiene:

MOVLW 05h

La instrucción está codificada

Hay algo que debemos recordar.

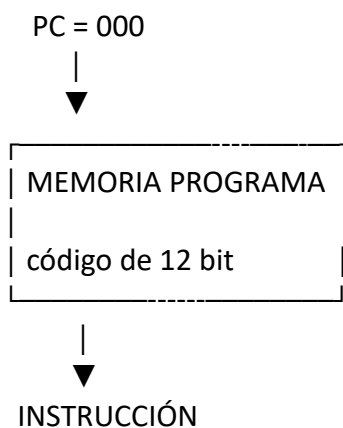
En la memoria no se almacena realmente el texto:

MOVLW 05h

La memoria contiene una palabra de instrucción de **12 bits**.

El ensamblador transforma el mnemónico y su operando en la codificación binaria correspondiente.

Por tanto, conceptualmente:



Para nosotros podemos mostrar:

MOVLW 05h

porque es mucho más fácil de entender.

DECODE — Decodificar

Una vez obtenida la instrucción, la CPU necesita determinar:

¿Qué operación representa este código?

Supongamos que obtiene:

MOVLW 05h

El decodificador identifica:

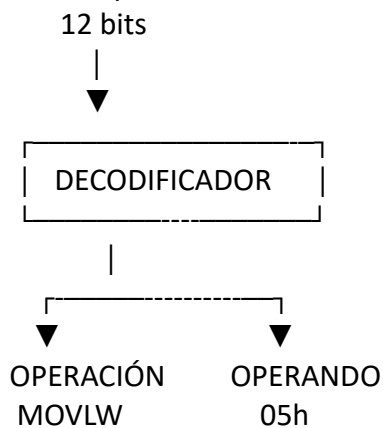
OPERACIÓN = MOVLW

y además identifica:

LITERAL = 05h

Por tanto, la CPU sabe que debe:
colocar 05h en W.

Podemos representarlo:



EXECUTE — Ejecutar

Ahora viene la tercera etapa.

La CPU realiza la operación.

En este caso:

05h → W

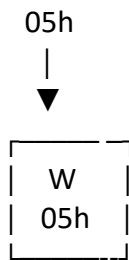
Si antes:

W = 00h

después:

W = 05h

Podemos representarlo:



¿Qué pasó con STATUS?

En este caso, MOVLW carga un literal en W y no realiza una operación aritmética que produzca cambios en los bits de estado aritmético que estamos estudiando.

Por tanto, podemos mostrar:

STATUS
sin cambio

¿Qué ocurre con el PC?

Después de ejecutar una instrucción normal, el flujo continúa hacia la siguiente posición de programa.

Por ejemplo:

ANTES
PC = 000

DESPUÉS
PC = 001

Podemos representarlo:

000 —————▶ 001

Por lo tanto, después de ejecutar:

MOVLW 05h
podemos tener:
PC = 001
W = 05h

Primera ejecución completa

Vamos a reunir todo.

Programa:

MOVLW 05h

MOVWF 08h

CLRWF

GOTO 000h

Estado inicial:

PC = 000

W = 00h

Ejecutamos:

MOVLW 05h

FETCH

PC = 000

Se obtiene la instrucción.

DECODE

MOVLW

literal = 05h

EXECUTE

$W \leftarrow 05h$

Resultado

PC = 001

W = 05h

Segunda instrucción: MOVWF

Ahora ejecutaremos:

MOVWF 08h

Recordemos que después de la instrucción anterior:

W = 05h

La instrucción:

MOVWF 08h

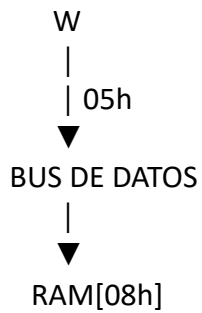
significa, conceptualmente:

Copiar el contenido de W a la posición de datos indicada.

Por tanto:
W = 05h
se copia a:
RAM[08h] = 05h

El recorrido de MOVWF

Podemos imaginar:



Antes:
W = 05h
RAM[08h] = 00h

Después:
W = 05h
RAM[08h] = 05h

Por lo tanto:
CAMBIÓ:

RAM[08h]
00h → 05h
Mientras:
W
05h → 05h
no cambia.

Tercera instrucción: CLRW

Ahora tenemos:
W = 05h

Ejecutamos:
CLRW

Esta instrucción borra W.
Entonces:
05h → 00h

El resultado es:

W = 00h

Y además debemos tener en cuenta el efecto correspondiente sobre el bit **Z** de STATUS según la definición de la instrucción.

En este caso:

resultado = 00h

por lo que:

Z = 1

Así podemos observar algo muy importante:

Una instrucción puede modificar **más de un registro o bit**.

Estado después de CLRW

Podemos mostrar:

ANTES

W = 05h

Z = 0

Ejecutamos:

CLRW

Después:

W = 00h

Z = 1

Por lo tanto:

CAMBIOS

W:

05h → 00h

STATUS.Z:

0 → 1

Ahora tenemos tres tipos de cambios

Con las primeras instrucciones ya encontramos tres situaciones:

MOVLW

W cambia

MOVWF

RAM cambia

CLRW

W cambia

STATUS cambia

Una tabla muy importante

Podemos empezar una tabla que posteriormente se convertirá en parte del diseño del simulador:

Instrucción	Lee	Modifica	STATUS
MOVLW	literal	W	según especificación
MOVWF	W	registro f	no modifica los flags aritméticos
CLRW	—	W	modifica Z
ADDWF	W + f	W o f	puede modificar Z, DC, C
SUBWF	W + f	W o f	puede modificar Z, DC, C
INCF	f	W o f	puede modificar Z
DECF	f	W o f	puede modificar Z

Esta tabla irá creciendo conforme estudiemos las 33 instrucciones.

El cuarto paso: GOTO

Ahora vamos a utilizar:

GOTO 000h

Aquí aparece algo nuevo.

Hasta ahora el PC avanzaba normalmente:

000 → 001 → 002 → 003

Pero GOTO cambia el flujo.

Supongamos:

PC = 003

y ejecutamos:

GOTO 000h

Después:

PC = 000

Tenemos:

003

|

| GOTO 000

▼

000

Por tanto:

GOTO modifica el flujo normal del programa.

Nuestro primer bucle

Ahora podemos escribir:

inicio:

MOVLW 05h

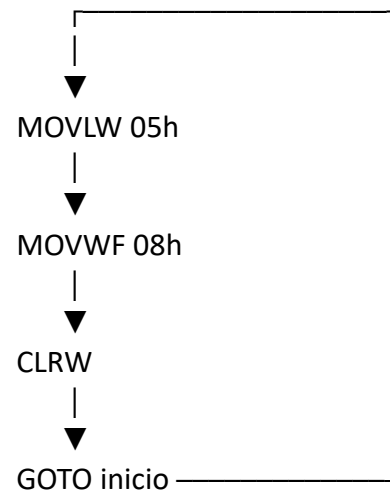
MOVWF 08h

CLRW

GOTO inicio

El programa vuelve continuamente al principio.

Visualmente:



Esto es un **bucle**.

Y acabamos de construir uno de los conceptos fundamentales de la programación de microcontroladores.

El PIC no "entiende" el bucle

Esto es importante.

El PIC no sabe que nosotros pensamos:

"Quiero repetir estas instrucciones."

Para el PIC solamente existe:

instrucción

instrucción

instrucción

Instrucción

Y cuando llega a:

GOTO 000h

el PC recibe una nueva dirección.

Por eso vuelve a ejecutar:

000

El bucle aparece como consecuencia de **modificar el PC**.

Esta es una idea central

Podemos decir:

Un programa no es solamente una lista de operaciones; también es una secuencia controlada por el PC.

Las instrucciones pueden:

continuar

o:

saltar

o:

llamar una subrutina

o:

saltar dependiendo de una condición

Esto será fundamental cuando estudiemos:

GOTO

CALL

BTFSC

BTFSS

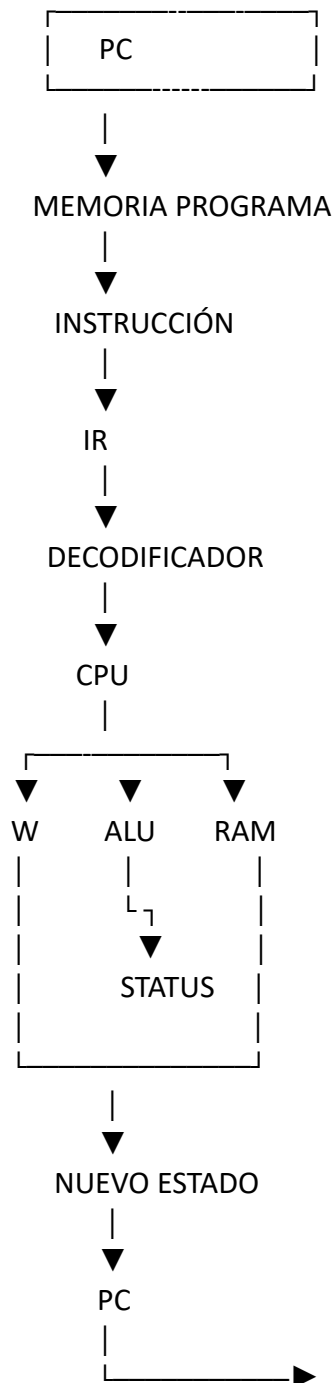
DECFSZ

INCFSSZ

RETLW

El ciclo completo

Ya podemos representar el funcionamiento general:



Organización de la memoria de datos

Hasta ahora hemos visto que el PIC12F509 tiene dos espacios de memoria diferentes:

MEMORIA DE PROGRAMA

|
└─ contiene instrucciones

MEMORIA DE DATOS

|
└─ contiene registros y datos

Ahora nos concentraremos en la **memoria de datos**.

La pregunta que queremos responder es:

¿Dónde guarda el PIC los datos que utiliza durante la ejecución de un programa?

La memoria de datos no es solamente una RAM

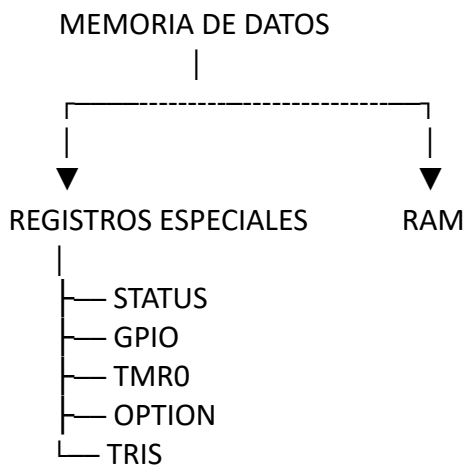
Cuando escuchamos la palabra RAM normalmente pensamos:

dirección → dato

Pero en un microcontrolador ocurre algo más interesante.

Algunas posiciones de la memoria de datos están asociadas con **registros que controlan el hardware**.

Por eso podemos pensar en la memoria de datos como:



Esto significa que una posición de memoria puede representar algo mucho más importante que un simple número.

Una dirección de memoria

Imaginemos que tenemos:

08h

Esta es una **dirección**.

La dirección nos dice:

"Quiero acceder a esta posición de la memoria de datos."

Por ejemplo:

DIRECCIÓN	CONTENIDO
-----------	-----------

00h	XX
01h	XX
02h	XX
03h	XX
04h	XX
05h	XX
06h	XX
07h	XX
08h	25h
09h	XX

...

Si queremos guardar:

25h

en la posición:

08h

conceptualmente tendremos:

RAM[08h] = 25h

Dirección y contenido son cosas diferentes

Esto es muy importante.

No debemos confundir:

08h

con:

25h

El primero es:

dirección

El segundo es:

dato

Podemos imaginar una casa:

CALLE → dirección
CASA → contenido

En memoria:
08h → dirección
25h → contenido

Por tanto:
RAM[08h] = 25h
significa:
En la dirección 08h está almacenado el valor 25h.

¿Cómo accedemos a una posición?

Aquí aparece la letra:
f
que ya encontramos al estudiar las instrucciones.

Por ejemplo:
MOVWF f

La **f** representa el registro o posición de memoria de datos que será utilizada por la instrucción.

Si conceptualmente hacemos:
MOVWF 08h

estamos indicando:
Guardar el contenido de W en la dirección 08h.

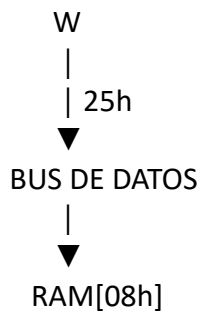
Si:
W = 25h
entonces:
RAM[08h] = 25h

Visualicemos MOVWF

Tenemos:
W = 25h

y ejecutamos:
MOVWF 08h

El camino de los datos será:



Antes:

W = 25h

RAM[08h] = 00h

Después:

W = 25h

RAM[08h] = 25h

¿Y cómo leemos un registro?

Ahora hagamos lo contrario.

Supongamos:

RAM[08h] = 25h

y queremos llevar ese valor a W.

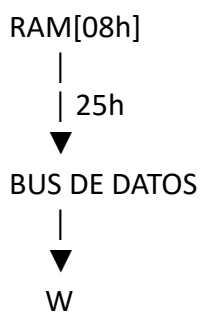
La instrucción:

MOVF 08h,W

indica conceptualmente:

Leer el contenido de la dirección 08h y colocar el resultado en W.

Entonces:



Resultado:

W = 25h

Ya tenemos dos instrucciones fundamentales

Podemos compararlas:

MOVWF

MOVWF 08h

Dirección:

W → RAM

MOVF

MOVF 08h,W

Dirección:

RAM → W

Visualmente:

MOVWF
W —————▶ RAM

y:
MOVF
RAM —————▶ W

¿Qué es un registro de función especial?

Ahora podemos introducir el concepto de:

SFR

Special Function Register

En español:

Registro de Función Especial.

Un **SFR** es un registro que está asociado con una función específica del microcontrolador.

Por ejemplo, en el PIC12F509 encontramos registros asociados con:

- STATUS;
- GPIO;
- TMR0;
- OPTION;
- TRIS.

No debemos pensar en ellos como simples lugares donde guardar números.

Por ejemplo:

REGISTRO GPIO

está relacionado con los pines físicos del PIC.

Por eso cambiar su contenido puede afectar las salidas del microcontrolador.

Una idea muy importante

Imaginemos:

RAM[08h] = 25h

Aquí simplemente estamos almacenando un dato.

Pero imaginemos ahora un registro que controla un periférico:

REGISTRO ESPECIAL = 25h

El valor puede provocar un cambio en el hardware.

Por eso:

Leer o escribir un registro de función especial puede significar interactuar directamente con el hardware del PIC.

STATUS como registro

Ya conocemos **STATUS**.

Ahora podemos verlo desde otro punto de vista.

STATUS es también una posición dentro del espacio de registros de datos.

Es decir, la CPU puede acceder a él mediante las instrucciones correspondientes.

Conceptualmente:

CPU



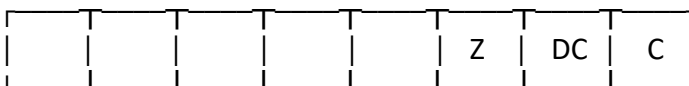
BUS DE DATOS



STATUS

Y dentro de **STATUS** tenemos diferentes bits.

STATUS

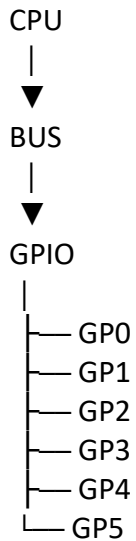


GPIO como registro

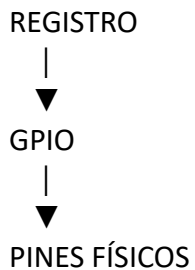
También podemos pensar en GPIO desde esta perspectiva.

Existe un registro asociado al puerto.

Conceptualmente:



De esta manera tenemos:

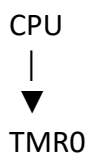


Esta relación será fundamental cuando construyamos el simulador visual.

TMRO

Lo mismo ocurre con el temporizador.

Tenemos:



TMR0 es un registro de 8 bits que participa en la función de temporización/contador.

Su valor puede cambiar debido al funcionamiento del temporizador.

Por eso el simulador tendrá que distinguir entre:

valor escrito por el programa

y:

valor modificado por el hardware

Esto será especialmente importante cuando estudiemos TMR0.

¿Qué significa f?

Ahora podemos entender mejor la notación:

ADDWF f,d

Aquí:

f = registro o posición de datos

Por ejemplo:

ADDWF 08h,W

significa que uno de los operandos procede de:

RAM[08h]

y el otro es:

W

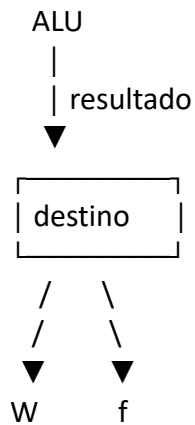
La ALU puede entonces realizar:

$W + \text{RAM}[08h]$

¿Qué significa d?

d indica el destino del resultado en las instrucciones que lo utilizan.

Podemos visualizar:



Por tanto, el resultado puede dirigirse hacia **W** o hacia el registro indicado por **f**, según la instrucción y el valor del bit de destino.

Ejemplo con ADDWF

Supongamos:

W = 03h

RAM[08h] = 05h

Ejecutamos:

ADDWF 08h,**W**

La ALU recibe:

	03h
+	
	05h

	08h

Resultado:

W = 08h

Y como el destino es **W**:

W = 08h

La RAM no cambia:

RAM[08h] = 05h

¿Qué pasaría con ADDWF 08h,f?

Ahora el destino sería el registro f.

Conceptualmente:

W = 03h

RAM[08h] = 05h

La ALU realiza:

$03h + 05h = 08h$

pero ahora:

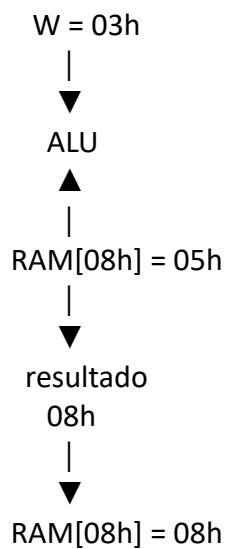
RAM[08h] = 08h

mientras:

W = 03h

permanece igual.

Tenemos:



Esto demuestra por qué el destino es tan importante.

Estado del PIC

Introducimos aquí un concepto que utilizaremos durante todo el desarrollo:

Estado del microcontrolador

El estado del PIC es, en esencia, el conjunto de valores que tienen sus registros y elementos internos en un momento determinado.

Por ejemplo:

ESTADO DEL PIC

PC = 003h
W = 05h
STATUS = 18h

RAM[08h] = 05h
GPIO = 00h
TMR0 = 12h

Después de ejecutar una instrucción podemos tener:

ESTADO NUEVO

PC = 004h
W = 08h
STATUS = 18h

RAM[08h] = 05h
GPIO = 00h
TMR0 = 12h

Una observación sobre la arquitectura

Hay que tener cuidado con una simplificación que suele hacerse al explicar microcontroladores:

"Toda la RAM es memoria libre para guardar datos."

En realidad, en el PIC12F509 hay posiciones del espacio de datos que corresponden a **registros de función especial** y otras posiciones destinadas a datos generales.

FORMATO DE LAS INSTRUCCIONES DEL PIC12F509

Hasta ahora hemos visto que el PIC12F509 obtiene una instrucción de su memoria de programa y después la CPU la decodifica.

Ahora vamos a abrir esa instrucción y mirar **qué contiene cada uno de sus bits**.

leer los 12 bits → identificar la instrucción → extraer sus campos → ejecutarla.

El PIC12F509 utiliza instrucciones de **12 bits**. Microchip divide las instrucciones en tres grandes grupos:

Operaciones orientadas a registros.

Operaciones orientadas a bits.

Operaciones literales y de control.

Una instrucción es una palabra de 12 bits

Supongamos que tenemos una instrucción almacenada en la memoria de programa:

1011 0010 0101

Son:

1011 0010 0101

AAAA AAAA AAAA

12 bits

Podemos numerar los bits:

Bit: 11 10 9 8 7 6 5 4 3 2 1 0

11	10	9	8	7	6	5	4	3	2	1	0
_____				_____				_____			
parte				parte				parte			

Pero hay algo muy importante:

los 12 bits no tienen siempre el mismo significado.

Dependiendo de la instrucción, algunos bits pueden representar:

el código de operación (**opcode**),

un registro f,

el destino d,

un número de bit b,

un valor constante k.

Por eso existen diferentes formatos.

¿Qué es el opcode?

La palabra **opcode** viene de:

Operation Code

es decir:

código de operación.

Es la parte de la instrucción que le dice al decodificador qué operación debe realizar la CPU.

Por ejemplo, el PIC12F509 tiene instrucciones como:

MOVLW
MOVWF
ADDWF
ANDWF
GOTO
BCF
BSF

El hardware no recibe literalmente las letras:

MOVLW

La CPU recibe una combinación de bits.

El decodificador reconoce esa combinación y determina:

"Esta combinación corresponde a MOVLW"

Primer ejemplo: MOVLW

La instrucción:

MOVLW k

significa:

$W \leftarrow k$

donde k es un valor literal de 8 bits.

Por ejemplo:

MOVLW 25h

significa:

$W \leftarrow 25h$

La codificación de MOVLW utiliza:

1100 kkkk kkkk

Por lo tanto, la estructura es:

1100	k
opcode	8 bits

Los primeros 4 bits identifican la operación:
1100

y los siguientes 8 bits contienen el valor.

Ejemplo completo de MOVLW 25h

Primero convertimos:

25h = 0010 0101

Entonces:

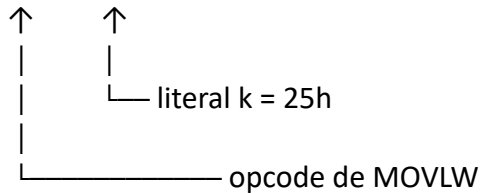
MOVLW 25h

queda codificado como:

1100 0010 0101

Ahora podemos separar los campos:

1100 | 0010 0101



Cuando el decodificador recibe:

1100 0010 0101

determina:

Operación = MOVLW

k = 25h

y la CPU ejecuta:

$W \leftarrow 25h$

Instrucciones orientadas a registros

Ahora veamos un formato diferente.

Por ejemplo:

ADDWF f,d

Esta instrucción realiza:

W + f

y coloca el resultado en:

W si d = 0

el registro f si d = 1

La codificación es:

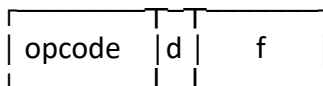
0001 11df ffff

Aquí aparecen tres campos:

0001 11 | d | ffff

opcode

Más claramente:



Donde:

d = destino

f = registro utilizado

Microchip especifica que en estas instrucciones f identifica el registro y d selecciona el destino. Si d=0, el resultado va a W; si d=1, vuelve al registro f.

¿Qué significa f?

Cuando encontramos:

f

en una instrucción, no significa que exista un registro llamado "F".

Significa:

file register

es decir, el registro o posición de memoria de datos sobre la que trabajará la instrucción.

Por ejemplo:

ADDWF 08h,0

podemos leerlo como:

W + registro 08h → W

Mientras que:

ADDWF 08h,1

significa:

$W + \text{registro } 08h \rightarrow \text{registro } 08h$

¿Qué significa d?

La letra:

D

significa:

Destination

es decir:

destino del resultado.

Tenemos solamente dos posibilidades:

$d = 0$

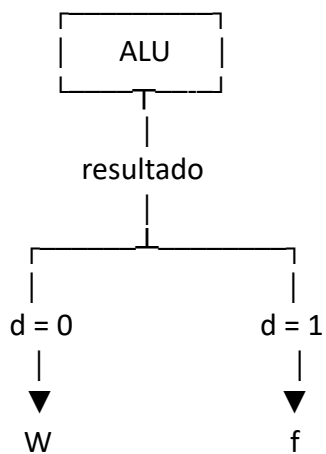
Resultado $\rightarrow W$

o:

$d = 1$

Resultado $\rightarrow$ registro f

Podemos visualizarlo:



Esto será muy importante cuando construyamos el simulador.

Ejemplo con ADDWF

Supongamos:

$W = 05h$

$\text{RAM}[08h] = 03h$

Ejecutamos:

ADDWF 08h,0

La ALU realiza:
 $05h + 03h = 08h$

Como:
 $d = 0$
el resultado va a W:
 $W = 08h$

El registro 08h permanece:
 $RAM[08h] = 03h$

En cambio:
ADDWF 08h,1
produce:
 $05h + 03h = 08h$

pero ahora:
 $d = 1$

por lo que:

$RAM[08h] = 08h$
y:
 $W = 05h$

Formato de las instrucciones de bits

Existe otro grupo:

BCF
BSF
BTFSC
BTFSS

Estas instrucciones trabajan sobre **un bit específico de un registro**.

Su formato general es:
0100 bbb ffff

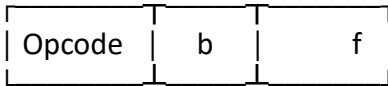
para BCF,
0101 bbb ffff

para BSF,
0110 bbb ffff

para BTFSC,
0111 bbb fffff

y:

para BTFSS.
Aquí aparecen:



donde:
b = número del bit

f = registro

¿Qué puede representar b?

b selecciona uno de los ocho bits de un registro.

Por ejemplo:

b = 0

selecciona:

bit 0

y:

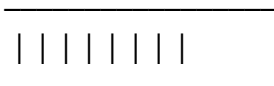
b = 7

selecciona:

bit 7

Visualmente:

7 6 5 4 3 2 1 0



↑
bit 0

Ejemplo: BCF

Supongamos:

RAM[08h] = 1011 0111

Ejecutamos:

BCF 08h,4

Significa:

poner en cero el bit 4 del registro 08h.

Antes:

bit: 7 6 5 4 3 2 1 0

dato: 1 0 1 1 0 1 1 1

↑

bit 4

Después:

bit: 7 6 5 4 3 2 1 0

dato: 1 0 1 0 0 1 1 1

↑

0

Por tanto:

1011 0111

↓

1010 0111

Ejemplo: BSF

Ahora:

BSF 08h,4

significa:

poner en uno el bit 4.

Si tenemos:

RAM[08h] = 1010 0111

obtenemos:

1011 0111

El resto de los bits permanece igual.

Instrucciones de salto condicional

Hay dos instrucciones especialmente interesantes:

BTFSC

BTFSS

Porque pueden modificar el flujo del programa.

Por ejemplo:

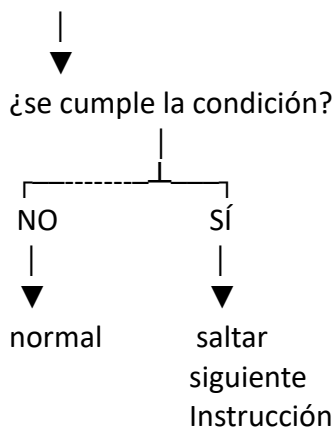
BTFSC 08h,0

significa:

comprobar el bit 0 de 08h y saltar la siguiente instrucción si ese bit está en cero.

Esto introduce un concepto muy importante:

INSTRUCCIÓN



Cuando el salto se produce, la instrucción tiene una duración de dos ciclos en lugar de uno.

Formato de las instrucciones literales

Otro grupo utiliza un valor constante:

K

Por ejemplo:

MOVLW k

ANDLW k

IORLW k

XORLW k

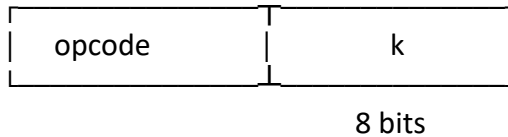
RETLW k

Aquí **k** normalmente representa un valor literal de 8 bits.

Por ejemplo:
MOVLW 55h

contiene:
k = 55h

Por eso podemos pensar:



Formato de GOTO

GOTO es diferente porque no necesita un dato para la ALU.
Su objetivo es modificar el flujo del programa.

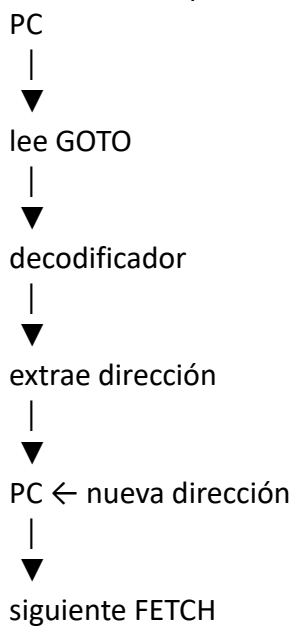
La codificación es:
101 kkkkk kkkk

Es decir, contiene un campo de dirección dentro de la instrucción.

Cuando se ejecuta:
GOTO dirección

el PC cambia para continuar la ejecución en la nueva posición.

Por eso conceptualmente tenemos:



Microchip indica además que GOTO es una excepción importante respecto de las instrucciones que escriben el PC, por la forma en que se maneja el bit superior del contador de programa. Este detalle lo estudiaremos cuando analicemos el PC completo.

La idea fundamental

Ahora podemos entender algo muy importante.

Cuando escribimos:

MOVLW 25h

el ensamblador no guarda en la memoria las letras:

M O V L W

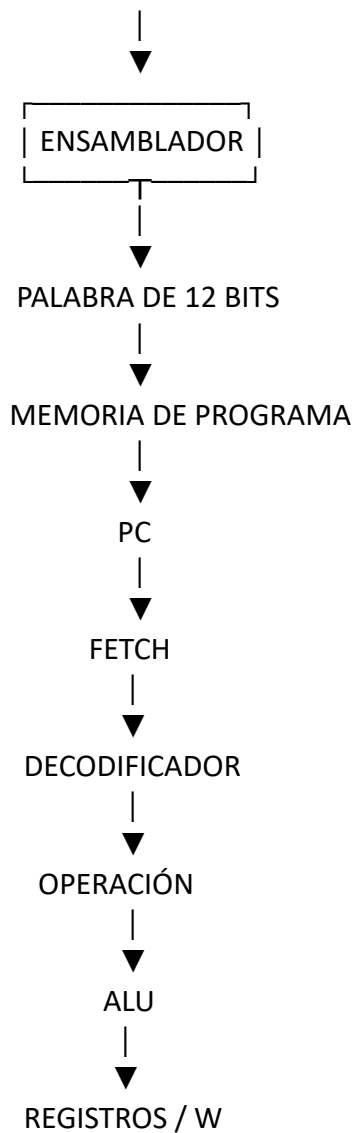
Convierte la instrucción a una palabra binaria de 12 bits:

1100 0010 0101

La memoria de programa contiene esos bits.

Entonces el proceso completo es:

PROGRAMA ESCRITO POR EL USUARIO



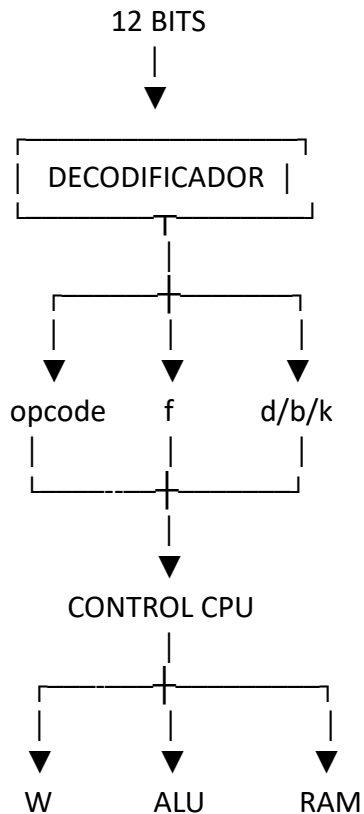
2.17.19 Tabla que usaremos para construir el simulador

Podemos comenzar a organizar las instrucciones de esta manera:

Tipo	Instrucción	Campos principales
Registro	ADDWF	opcode + d + f
Registro	ANDWF	opcode + d + f
Registro	CLRF	opcode + f
Registro	CLRW	opcode
Registro	MOVF	opcode + d + f
Registro	MOVWF	opcode + f
Registro	INCF	opcode + d + f
Registro	DECF	opcode + d + f
Registro	RLF	opcode + d + f
Registro	RRF	opcode + d + f
Bit	BCF	opcode + b + f
Bit	BSF	opcode + b + f
Bit	BTFSC	opcode + b + f
Bit	BTFSS	opcode + b + f
Literal	MOVLW	opcode + k
Literal	ANDLW	opcode + k
Literal	IORLW	opcode + k
Literal	XORLW	opcode + k
Control	GOTO	opcode + dirección
Control	CALL	opcode + dirección

La CPU como un intérprete de bits

Podemos imaginar al decodificador como un pequeño intérprete electrónico:



Esto nos permite entender finalmente qué significa realmente "**decodificar una instrucción**".

No es algo misterioso.

Es:

mirar determinados bits de la palabra de 12 bits y determinar qué operación representan y qué datos necesita esa operación.

Preparación para el siguiente paso

Ya tenemos las piezas necesarias para dar el siguiente paso:

MEMORIA



INSTRUCCIÓN 12 bits



DECODIFICADOR



opcode

f

d

b

k



EJECUTOR



W

ALU

RAM

STATUS

GPIO

PC

DECODIFICACIÓN REAL DE UNA INSTRUCCIÓN, BIT POR BIT

Ahora vamos a hacer algo fundamental.
Hasta este momento hemos dicho:

"La CPU recibe una palabra de 12 bits y la decodifica."

Ahora veremos **cómo podemos abrir esos 12 bits y descubrir qué instrucción contienen.**

Los 12 bits

Recordemos:

Bit: 11 10 9 8 7 6 5 4 3 2 1 0

12 BITS

Una instrucción puede ser, por ejemplo:
1100 0010 0101

El procesador no ve:
MOVLW 25h

Ve:
110000100101

Nosotros debemos aprender a interpretar esos bits.

Ejemplo 1 — MOVLW 25h

La documentación oficial indica:
MOVLW k
}
tiene el formato:
1100 kkkk kkkk

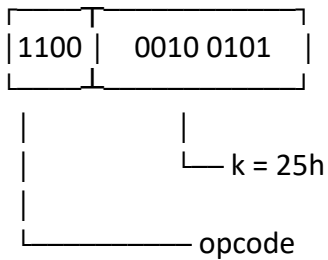
y carga el literal de 8 bits k en W.

Tenemos:
MOVLW 25h

Primero:
25h = 0010 0101

Por tanto:
1100 0010 0101

Ahora separamos:



El decodificador encuentra:
1100

y sabe:
OPERACIÓN = MOVLW

Después extrae:
k = 00100101

que es:
k = 25h

Finalmente:
 $W \leftarrow 25h$

Ejemplo 2 — MOVWF

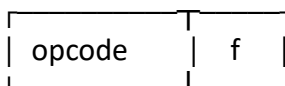
La instrucción:
MOVWF f

tiene el formato:
0000 001f ffff

según la tabla oficial.

Aquí:
f = 5 bits

Por tanto:



Supongamos:
MOVWF 08h

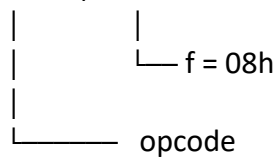
Convertimos:
08h = 01000
porque f solamente necesita 5 bits.

La instrucción queda:

0000 0011 1000

La separación conceptual es:

0000 001 | 01000



La CPU interpreta:

MOVWF

f = 08h

y ejecuta:

RAM[08h] $\leftarrow$ W

Una observación muy importante sobre f

Aquí aparece una diferencia que debemos tener muy clara.

Cuando escribimos:

MOVWF 08h

08h es una **dirección de registro**.

Pero cuando se almacena dentro de la instrucción, no se guardan los ocho bits de 08h. Se utilizan solamente los bits necesarios para el campo f.

En este caso:

08h = 01000

y:

f = 01000

Por eso el formato de la instrucción es:

0000 001f ffff

y no:

0000 kkkkkkkk

Ejemplo 3 — ADDWF

Ahora vamos a una instrucción más interesante:

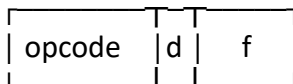
ADDWF f,d

Microchip especifica:

0001 11df ffff

y la instrucción suma W y f; el resultado se coloca en W cuando d=0 o en f cuando d=1.

Tenemos tres partes:



ADDWF 08h,0

Primero:

f = 08h = 01000

y:

d = 0

Entonces:

0001 11 0 01000

Agrupamos los bits:

0001 1101 0000

La CPU puede interpretarlo como:

opcode = ADDWF

d = 0

f = 08h

Ahora supongamos:

W = 05h

RAM[08h] = 03h

La ALU realiza:

05h

+

03h

08h

Como:

$d = 0$

el resultado va a W:

$W = 08h$

y:

$RAM[08h] = 03h$

permanece igual.

ADDWF 08h,1

Ahora solamente cambia:

$d = 1$

La CPU sigue haciendo:

$W + RAM[08h]$

pero ahora el resultado se guarda en:

$RAM[08h]$

Por ejemplo:

$W = 05h$

$RAM[08h] = 03h$

produce:

$05h + 03h = 08h$

y:

$RAM[08h] = 08h$

mientras:

$W = 05h$

permanece sin modificación.

Esto demuestra por qué el bit **d** es tan importante.

El bit d es solamente un bit

Para el estudiante puede parecer que:

ADDWF 08h,0

y:

ADDWF 08h,1

son instrucciones completamente diferentes.

Pero internamente son prácticamente la misma instrucción.

La diferencia está en **un solo bit**:

ADDWF 08h,0

↑
d=0

o:

ADDWF 08h,1

↑
d=1

Por eso podemos decir:

El bit **d** le dice al hardware dónde colocar el resultado.

Ejemplo 4 — BCF

Ahora pasemos a las instrucciones orientadas a bits.

La instrucción:

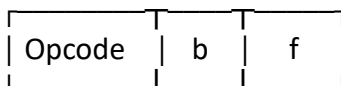
BCF f,b

tiene el formato:

0100 bbbf ffff

según Microchip.

Aquí tenemos:



Donde:

b = número de bit

f = registro

BCF 08h,4

Supongamos:

BCF 08h,4

Tenemos:

f = 08h = 01000

y:

b = 4 = 100

Entonces:
0100 100 01000

que agrupado queda:
0100 1000 1000

El decodificador obtiene:
opcode = BCF
b = 4
f = 08h

La CPU entonces realiza:
bit 4 de RAM[08h] $\leftarrow$ 0

Visualización

Supongamos:
RAM[08h] = 1011 0111

Tenemos:
bit: 7 6 5 4 3 2 1 0

dato: 1 0 1 1 0 1 1 1
 ↑
 bit 4

Ejecutamos:
BCF 08h,4

Resultado:
1010 0111

Solamente cambia un bit:
ANTES DESPUÉS

1011 0111	1010 0111
↑	↑
1	0

Ejemplo 5 — BSF

BSF tiene:
0101 bbbf ffff

Por ejemplo:
BSF 08h,4

significa:

bit 4 de RAM[08h] $\leftarrow$ 1

Si inicialmente:

1010 0111

después:

1011 0111

Nuevamente:

la CPU no necesita modificar todo el registro; modifica solamente el bit indicado.

Ejemplo 6 — BTFSC

Ahora tenemos una instrucción especialmente importante:

BTFSC f,b

Su formato es:

0110 bbbf ffff

Su significado es:

Bit Test f, Skip if Clear

Es decir:

mira el bit **b**;

si está en cero, salta la siguiente instrucción;

si está en uno, continúa normalmente.

Ejemplo práctico

Tenemos:

GPIO = 0000 0001

y ejecutamos:

BTFSC GPIO,0

El bit 0 vale:

1

Por tanto:

¿bit 0 = 0?

Respuesta:

NO

Entonces:

NO SALTAR

La siguiente instrucción se ejecuta normalmente.

Ahora supongamos:

GPIO = 0000 0000

Ejecutamos:
BTFSC GPIO,0

El bit 0 vale:
0

Entonces:
¿bit 0 = 0?

Respuesta:
Sí

Por lo tanto:
SALTAR LA SIGUIENTE INSTRUCCIÓN

Ejemplo 7 — GOTO

Ahora veremos una instrucción que no trabaja con W ni con un registro de datos:
GOTO k

Su formato es:
1011 kkkk kkkk

Aquí:
1011
identifica:
GOTO

y los ocho bits restantes contienen el campo k.

Por ejemplo:
GOTO 020h

El objetivo es modificar el flujo de ejecución.

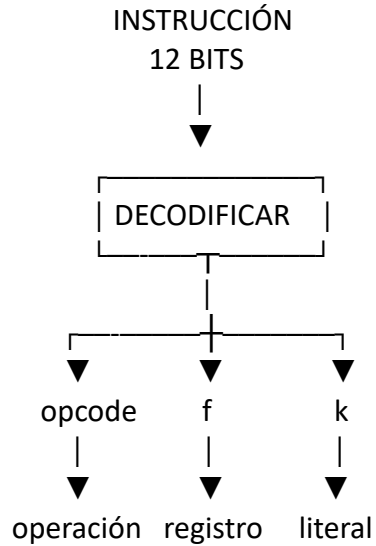
Conceptualmente:
PC = 010
tenemos:
GOTO 020h

y después:
PC = 020

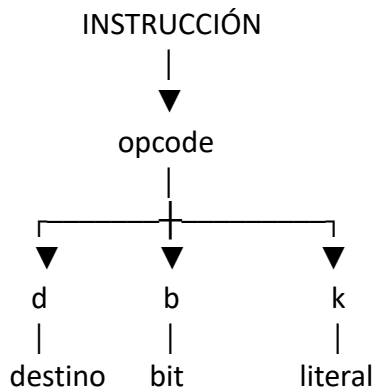
Hay un detalle arquitectónico adicional sobre el bit superior del PC y GOTO que veremos posteriormente cuando estudiemos en profundidad el Program Counter. Microchip señala expresamente esta excepción en la documentación.

El decodificador ya comienza a tomar forma

Podemos representar nuestro futuro decodificador así:



Y para instrucciones especiales:



Instrucción	Formato
ADDWF	0001 11df ffff
ANDWF	0001 01df ffff
CLRF	0000 011f ffff
CLRW	0000 0100 0000
COMF	0010 01df ffff
DECF	0000 11df ffff
DECFSZ	0010 11df ffff
INCF	0010 10df ffff
INCFSZ	0011 11df ffff
IORWF	0001 00df ffff
MOVF	0010 00df ffff
MOVWF	0000 001f ffff
NOP	0000 0000 0000
RLF	0011 01df ffff
RRF	0011 00df ffff
SUBWF	0000 10df ffff
SWAPF	0011 10df ffff
XORWF	0001 10df ffff
BCF	0100 bbbf ffff
BSF	0101 bbbf ffff
BTFSC	0110 bbbf ffff
BTFSS	0111 bbbf ffff
ANDLW	1110 kkkk kkkk
CALL	1001 kkkk kkkk
GOTO	1011 kkkk kkkk
IORLW	1101 kkkk kkkk
MOVLW	1100 kkkk kkkk
RETLW	1000 kkkk kkkk
XORLW	1111 kkkk kkkk

Los formatos anteriores corresponden a la tabla de instrucciones del PIC12F508/509/16F505 de Microchip.

FETCH, DECODE Y EXECUTE EN EL PIC12F509

Del modelo sencillo al funcionamiento real

Hasta ahora hemos utilizado este modelo:

```
FETCH
↓
DECODE
↓
EXECUTE
```

Este modelo es excelente para aprender.

Pero ahora vamos a acercarnos al funcionamiento real del PIC12F509.

El PIC12F509 utiliza una **canalización de instrucciones**.

Esto significa que mientras una instrucción se está ejecutando, el PIC puede estar obteniendo de memoria la siguiente instrucción.

Microchip describe esta arquitectura como una canalización entre **fetch** y **execute**. Podemos imaginarla así:

TIEMPO →

Ciclo 1:	FETCH 1	
Ciclo 2:	FETCH 2	EXECUTE 1
Ciclo 3:	FETCH 3	EXECUTE 2
Ciclo 4:	FETCH 4	EXECUTE 3

Por eso, después de que la canalización está funcionando, normalmente se completa una instrucción por cada ciclo de instrucción.

¿Por qué esto puede resultar confuso?

Porque podríamos preguntar:

"Si una instrucción necesita FETCH y después EXECUTE, ¿por qué tarda solamente un ciclo?"

La respuesta está precisamente en la canalización.
Las operaciones se superponen.

Mientras:
MOVWF

está siendo ejecutada, el PIC puede estar obteniendo de memoria la instrucción siguiente.

Por ejemplo:
Programa:

```
000 MOVLW 05h
001 MOVWF 08h
002 INCF 08h,1
003 GOTO 002h
```

La ejecución puede visualizarse así:

	CICLO			
	1	2	3	4
FETCH	MOVLW	MOVWF	INCF	GOTO
EXECUTE		MOVLW	MOVWF	INCF

La idea fundamental es:

FETCH de una instrucción y EXECUTE de otra pueden ocurrir en el mismo ciclo.

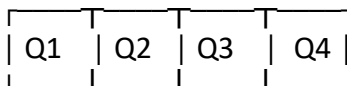
Esto está documentado por Microchip para el PIC12F508/509/16F505.

El ciclo de instrucción tiene cuatro fases

Un ciclo de instrucción está formado por:

Q1
Q2
Q3
Q4

Es decir:



1 ciclo

Microchip indica que la entrada de reloj se divide internamente para generar estas cuatro fases.

Por tanto:

1 ciclo de instrucción = 4 períodos del oscilador

Por ejemplo, si hipotéticamente:

$F_{osc} = 4 \text{ MHz}$

el período del oscilador es:

$T_{osc} = 1 / 4\,000\,000$

por lo tanto:

$T_{osc} = 0.25 \mu s$

Y:

$4 \times 0.25 \mu s = 1 \mu s$

Así:

1 ciclo de instrucción = $1 \mu s$

para un reloj de 4 MHz.

La documentación de Microchip confirma esta relación.

Q1

En Q1 ocurre algo muy importante.

El Program Counter avanza.

Microchip indica que durante Q1 el PC se incrementa y comienza la búsqueda de la instrucción.

Podemos representarlo:

Q1

PC

|



PC + 1

y simultáneamente comienza el proceso de obtener la instrucción correspondiente.

Q2

Durante Q2 comienza la parte de ejecución de la instrucción que ya había sido obtenida.

En instrucciones que necesitan leer memoria de datos, Microchip especifica que la lectura del operando ocurre durante Q2.

Por ejemplo:

ADDWF 08h,0

necesita conocer:

W

y:

RAM[08h]

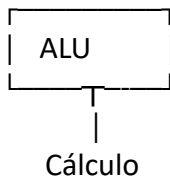
Durante esta fase se realiza la lectura necesaria del registro.

Q3

Durante Q3 continúa el procesamiento interno.

Podemos simplificarlo didácticamente como:

Q3



En una operación aritmética, por ejemplo:

$W + f$

la ALU realiza la operación.

Q4

Durante Q4 se realiza la escritura del resultado cuando la instrucción necesita escribir en memoria de datos.

Microchip especifica que la escritura del destino ocurre durante Q4.

Por ejemplo:

ADDWF 08h,1

si produce:

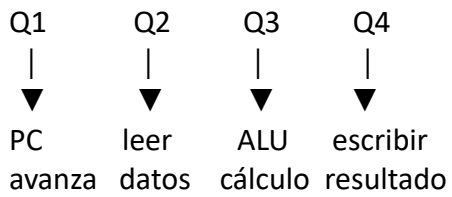
08h

el resultado se escribe en:

RAM[08h]

durante la fase correspondiente.

Podemos resumir:



El Instruction Register

Recordemos el registro de instrucción:

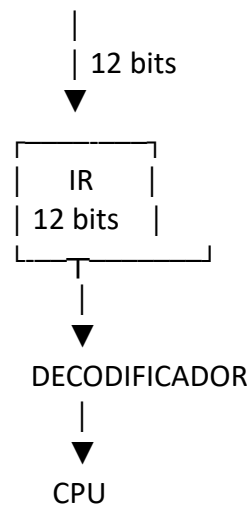
IR

Su función es mantener la instrucción que está siendo ejecutada.

Microchip indica que durante el ciclo de ejecución la instrucción obtenida se carga en el Instruction Register (**IR**) y posteriormente es decodificada y ejecutada.

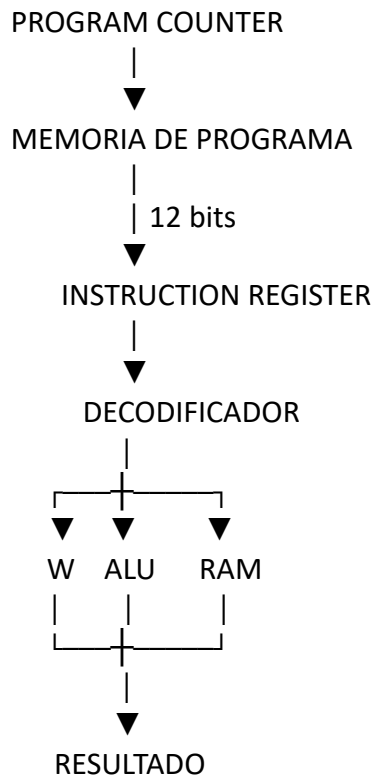
Podemos visualizarlo:

MEMORIA DE PROGRAMA



El recorrido completo

Ahora podemos juntar todo.



Y paralelamente:

PC

└─▶ siguiente instrucción

Esto es la canalización.

Ejemplo completo

Supongamos este programa:

```
000  MOVLW 05h
001  MOVWF 08h
002  INCF 08h,1
003  GOTO 002h
```

Inicialmente:

PC = 000

W = 00h

RAM[08h] = 00h

CICLO 1

El PIC obtiene:
MOVLW 05h

La instrucción queda preparada para ejecución.

Después:
 $W \leftarrow 05h$
Estado:
PC = 001
W = 05h
RAM[08h] = 00h

CICLO 2

Mientras se ejecuta:
MOVWF 08h

el PIC ya ha obtenido la instrucción correspondiente al siguiente punto del programa.

La ejecución produce:
 $RAM[08h] \leftarrow W$

Como:
W = 05h

tenemos:
RAM[08h] = 05h

Estado:
PC = 002
W = 05h
RAM[08h] = 05h

CICLO 3

Ahora:
INCF 08h,1

significa:
 $RAM[08h] + 1$

Tenemos:
 $05h + 01h = 06h$

Como:
d = 1

el resultado vuelve a:
RAM[08h]

Estado:
PC = 003
W = 05h
RAM[08h] = 06h

CICLO 4 — GOTO

Ahora encontramos:
GOTO 002h

Aquí sucede algo diferente.
Normalmente el PC avanzaría:
003 → 004

pero GOTO dice:
PC ← 002
Por tanto:
PC = 002

Y aquí aparece un detalle fundamental:
Cuando una instrucción modifica el PC, la instrucción que ya estaba siendo obtenida para la secuencia normal debe descartarse.

Microchip denomina esto un **flush de la canalización**. Por eso las instrucciones que cambian el PC necesitan dos ciclos de instrucción.

¿Qué significa "flush"?

Imaginemos:

000 MOVLW
001 MOVWF
002 INCF
003 GOTO 002
004 BSF

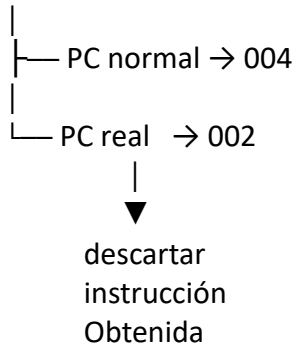
Cuando el PIC está ejecutando:
GOTO 002

puede haber comenzado a obtener:
BSF
porque la secuencia normal indicaría:
003 → 004

Pero GOTO dice:
003 → 002

Por lo tanto:
BSF
ya no debe ejecutarse.

Hay que descartarla.
GOTO



Eso es el **flush**.

Por eso GOTO tarda dos ciclos

Una instrucción normal:
MOVLW

puede completar su ejecución en:
1 ciclo

Pero:
GOTO

necesita:
2 ciclos
porque debe cambiar el flujo de ejecución y eliminar la instrucción que estaba en la canalización.

Microchip especifica esta excepción directamente.

MAPA REAL DE LA MEMORIA DE DATOS DEL PIC12F509

Ya no hablaremos de una RAM genérica

Hasta ahora hemos utilizado ejemplos como:

RAM[08h] = 25h

Esto nos ayudó a entender las instrucciones.

Pero ahora debemos hacer una distinción fundamental:

En el PIC12F509, las posiciones de memoria de datos no son todas RAM para guardar variables.

Algunas posiciones están conectadas directamente al hardware del microcontrolador.

Por ejemplo:

01h → TMRO

02h → PCL

03h → STATUS

04h → FSR

05h → OSCCAL

06h → GPIO

Y la posición:

00h → INDF

no corresponde a un registro físico normal.

Microchip clasifica estos registros como **Special Function Registers (SFR)**.

El mapa básico

Para el PIC12F509, la parte inicial del mapa es:

Dirección	Registro
00h	INDF
01h	TMRO
02h	PCL
03h	STATUS
04h	FSR
05h	OSCCAL
06h	GPIO

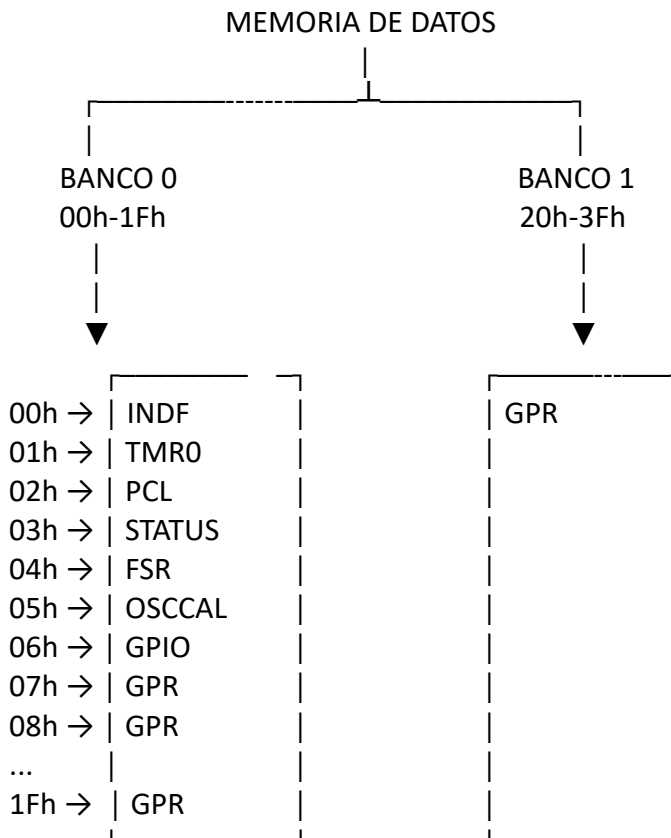
Después aparecen registros de propósito general.

Pero aquí aparece una característica muy importante:

El PIC12F509 tiene más de 32 registros de archivo, por lo que utiliza **bancos de memoria**.
El banco se selecciona mediante el bit correspondiente del FSR.

Mapa simplificado del PIC12F509

Podemos visualizarlo así:



En el PIC12F509, el espacio de datos direccionable llega hasta 3Fh, mediante dos bancos de 32 posiciones.

¿Qué es un SFR?

SFR significa:

Special Function Register

o:

Registro de Función Especial.

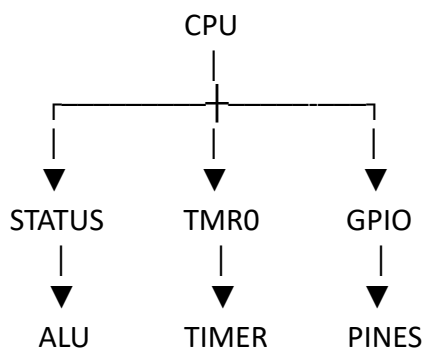
Es un registro que está conectado con alguna parte específica del microcontrolador.

Por ejemplo:
STATUS
está relacionado con:
ALU

Mientras:
TMR0
está relacionado con:
Timer0

Y:
GPIO
está relacionado con:
terminales físicas del PIC

Podemos visualizarlo:



Por eso escribir en un SFR puede provocar un efecto físico.

Registro 00h — INDF

La dirección:
00h
corresponde a:
INDF

Pero aquí debemos tener muchísimo cuidado.
INDF no es un registro físico.

Microchip especifica que INDF utiliza el contenido de FSR para determinar qué posición de memoria se está direccionando.

Por eso:
INDF
puede entenderse como una especie de:
"puerta de acceso indirecto a otra posición de memoria".

Ejemplo de INDF

Supongamos:

FSR = 08h

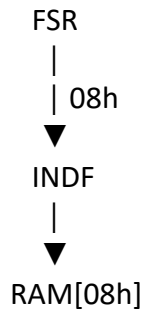
Entonces:

INDF

se refiere indirectamente a:

RAM[08h]

Podemos imaginar:



Por ejemplo:

MOVLW 08h

MOVWF FSR

Después:

CLRF INDF

no significa:

CLRF registro físico INDF

Significa:

CLRF RAM[08h]

porque FSR contiene:

08h

FSR es un puntero

Esta es una palabra que conviene aprender: **puntero**

Un puntero contiene una dirección.

Por ejemplo:

FSR = 08h

significa:

"la posición que me interesa está en 08h".

Si cambiamos:

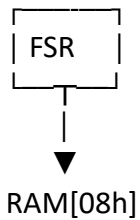
FSR = 09h

ahora INDF apunta a:

09h

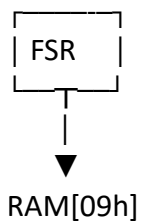
Visualmente:

FSR = 08h



y posteriormente:

FSR = 09h



¿Por qué necesitamos esto?

Porque permite trabajar con una serie de posiciones de RAM sin tener que escribir una instrucción diferente para cada dirección.

Por ejemplo:

FSR = 08h

trabajamos con:

INDF → 08h

Después:

INCF FSR,F

y ahora:

FSR = 09h

por lo que:

INDF → 09h

Después:

FSR = 0Ah

y:

INDF → 0Ah

Este mecanismo será muy importante cuando estudiemos **direccionamiento indirecto**.

Registro 01h — TMR0

La dirección:

01h

corresponde a:

TMR0

Es un registro de 8 bits asociado al Timer0.

01h

|



TMR0

|



TIMER 0

Pero TMR0 no es simplemente una variable.

Puede cambiar como consecuencia del funcionamiento del temporizador.

Microchip describe TMR0 como un contador/temporizador de 8 bits.

Esto significa que el simulador tendrá que distinguir entre:

REGISTRO MODIFICADO POR SOFTWARE

y:

REGISTRO QUE TAMBIÉN PUEDE CAMBIAR POR HARDWARE

Ejemplo con TMR0

Supongamos:

MOVLW 00h

y después:

MOVWF TMR0

Estamos cargando:

TMR0 = 00h

Pero después el Timer0 puede comenzar a incrementarse según su configuración.

Por eso:

TMR0 = 00h

no necesariamente permanecerá así.

Registro 02h — PCL

La dirección:

02h

corresponde a:

PCL

Es la parte baja del Program Counter.

Aquí tenemos que detenernos porque aparece una idea muy importante:

El PC completo es más grande que los 8 bits almacenados directamente en PCL.

El PIC12F509 tiene un PC de **11 bits**, aunque la memoria físicamente implementada del dispositivo es de 1K × 12.

Por eso podemos imaginar:

PC

bit 10 bit 8 | bit 7 bit 0
parte alta | PCL

El PCL proporciona los bits bajos.

PCL y el salto

Esto explica por qué instrucciones como:

GOTO

CALL

necesitan un tratamiento especial.

No basta con pensar:

PC = dirección

Hay que considerar cómo se construye el PC.

Por ejemplo, Microchip especifica para GOTO que el campo de dirección carga los bits bajos del PC y que los bits superiores proceden de los bits de página de STATUS.

Esto lo estudiaremos con detalle cuando lleguemos al PC.

Registro 03h — STATUS

Este es uno de los registros más importantes de todo el PIC12F509.

La dirección es:

03h

y su estructura es:

bit: 7 6 5 4 3 2 1 0

GPWUF - PA0 TO PD Z DC C

Microchip define estos bits específicamente para el PIC12F509.

Bit Z

El bit:

Z

es:

bit 2

Indica si el resultado de determinadas operaciones es cero.

Por ejemplo:

CLRW

produce:

W = 00h

y:

Z = 1

Microchip especifica que Z=1 cuando el resultado de una operación aritmética o lógica es cero.

Bit C

El bit:

C

es:

bit 0

y se utiliza como Carry/Borrow en determinadas instrucciones.

Por ejemplo, en:

ADDWF

SUBWF

RLF

RRF

su significado depende de la operación.

En una suma:

FFh + 01h = 100h

el resultado de 8 bits es:

00h

y:

C = 1

Microchip define explícitamente el significado de C para ADDWF, SUBWF, RLF y RRF.

Bit DC

El bit:

DC

está en:

bit 1

Se utiliza para indicar Carry/Borrow en el nibble bajo para determinadas operaciones aritméticas.

Por ejemplo:

0Fh + 01h

produce:

10h

porque:

0000 1111

+

0000 0001

0001 0000

Se produjo un acarreo desde el bit 3 hacia el bit 4.

Por tanto:

DC = 1

Microchip especifica este comportamiento para ADDWF y SUBWF.

Bit PA0

El bit:

PA0

es:

bit 5

En el PIC12F509 se utiliza para seleccionar la página de memoria de programa.

Microchip define:

PA0 = 0

para:

000h – 1FFh

y:

PA0 = 1

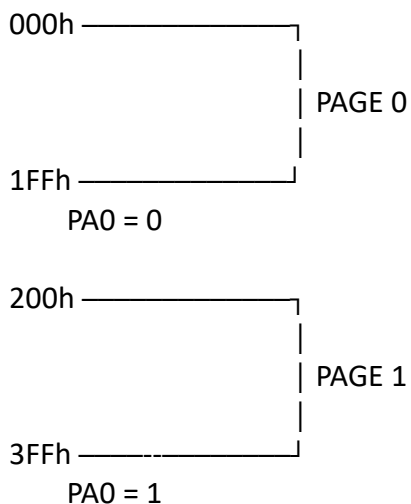
para:

200h – 3FFh

Es decir, las dos páginas de 512 palabras del programa.

Podemos visualizar:

PROGRAM MEMORY



Esto será esencial cuando estudiemos GOTO y CALL.

Bit TO

El bit:

TO

está en:

bit 4

Está relacionado con el estado del Watchdog Timer y determinadas operaciones de reset/sleep.

Por ejemplo, Microchip indica que TO=1 después de Power-up, CLRWDT o SLEEP, mientras que un time-out del WDT lo lleva a 0.

No necesitamos memorizar todavía todas estas condiciones.

Lo importante ahora es reconocer que:

STATUS

no contiene solamente información de la ALU.

También contiene información relacionada con:

RESET

WATCHDOG

POWER-DOWN

PROGRAM PAGE

Bit PD

El bit:

PD

está en:

bit 3

Está relacionado con el estado de Power-down/SLEEP.

Microchip especifica que:

PD = 1

después de Power-up o CLRWDT.

Y:

PD = 0

cuando se ejecuta SLEEP.

Por tanto, STATUS es realmente un registro con información de varias partes del microcontrolador.

Registro 04h — FSR

La dirección:

04h

corresponde a:

FSR

File Select Register.

Su función principal es seleccionar una dirección para el direccionamiento indirecto.

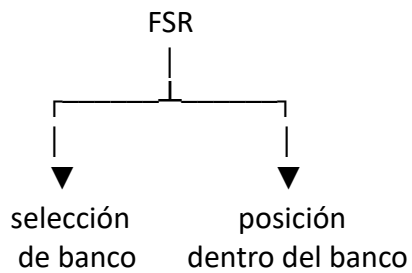
En el PIC12F509, FSR<5> selecciona el banco:

FSR<5> = 0 → Bank 0

FSR<5> = 1 → Bank 1

y los bits inferiores seleccionan la posición dentro del banco.

Esto significa que FSR tiene un papel doble:



Ejemplo de FSR en PIC12F509

Supongamos:

FSR = 00001 01000

Los bits relevantes pueden interpretarse como:

bit 5 = 0

por tanto:

Bank 0

y:

bits 4..0 = 01000

por tanto:

08h

Así:

INDF → 08h

Ahora:

FSR = 00101 01000

con:

bit 5 = 1

tenemos:

Bank 1

y la posición correspondiente.

Esto es lo que permite acceder a la segunda mitad de la memoria de datos.

Registro 05h — OSCCAL

La dirección:

05h

corresponde a:

OSCCAL

Este registro contiene la información de calibración del oscilador interno.

En el mapa de registros aparece con bits de calibración CAL6...CAL0.

Este registro será importante cuando estudiemos:

OSCILADOR

y especialmente el valor de calibración almacenado en la última posición de memoria de programa.

Recordemos que para el PIC12F509 la dirección:

03FFh

contiene el valor de calibración del oscilador y **no debe sobrescribirse**.

Registro 06h — GPIO

La dirección:

06h

corresponde a:

GPIO

Este registro conecta el mundo interno del PIC con sus terminales de entrada/salida.

El registro tiene:

GP5 GP4 GP3 GP2 GP1 GP0

con los bits superiores sin implementar.

Visualmente:

GPIO

bit: 7 6 5 4 3 2 1 0

- - 5 4 3 2 1 0

Por tanto, cuando nuestro programa ejecuta algo que modifica GPIO, estamos modificando potencialmente el estado de los pines físicos.

GPIO no es simplemente RAM

Si tenemos:

MOVLW 01h

MOVWF GPIO

no debemos mostrar únicamente:

RAM[06h] = 01h

Debemos mostrar algo como:

GPIO = 000001

y además:

GP0 = 1

GP1 = 0

GP2 = 0

GP3 = 0

GP4 = 0

GP5 = 0

Y finalmente representar:

PIC12F509

GP0 —————▶ LED
GP1 —————▶ entrada
GP2 —————▶ salida
...

TRISGPIO

Aquí aparece otro concepto interesante.

El registro:

TRISGPIO

no aparece como una dirección normal dentro del mapa de memoria.

Microchip lo muestra como un registro de control especial.

La instrucción:

TRIS 6

transfiere:

W → TRISGPIO

El datasheet especifica que el valor de TRIS determina si los pines funcionan como entradas o salidas.

Conceptualmente:

TRISGPIO

bit = 1 → entrada / alta impedancia

bit = 0 → salida

Por ejemplo:

W = 11111111

TRIS 6

configura los pines correspondientes como entradas.

Mientras:

W = 00000000

TRIS 6

los configura como salidas.

OPTION

Algo parecido ocurre con:

OPTION

No es un registro de memoria de datos normal.

Es un registro de control especial y se carga mediante:

OPTION

La operación es:

W → OPTION

Microchip especifica que OPTION controla funciones relacionadas con Timer0 y el prescaler del WDT/Timer0.

Sus bits son:

7 6 5 4 3 2 1 0

GPWU GPPU TOCS TOSE PSA PS2 PS1 PS0

Esto lo estudiaremos posteriormente junto con Timer0.

2.20.27 El mapa completo que debemos recordar

Podemos resumir el comienzo de la memoria de datos así:

Dirección	Registro	Función
00h	INDF	direccionamiento indirecto
01h	TMR0	Timer/contador
02h	PCL	parte baja del PC
03h	STATUS	estado de CPU
04h	FSR	puntero de memoria
05h	OSCCAL	calibración oscilador
06h	GPIO	entradas/salidas
07h...	GPR	propósito general
10h-1Fh	GPR	propósito general

El segundo banco contiene más registros de propósito general. Las direcciones de banco 1 se seleccionan mediante FSR<5>.

GPR: registros de propósito general

GPR significa:

General Purpose Register

Son posiciones destinadas principalmente a que el programa almacene información.

Por ejemplo:

08h

09h

0Ah

0Bh

...

pueden utilizarse para:

contador

variable

dato temporal

resultado

etc.

Por ejemplo:

MOVLW 25h

MOVWF 08h

produce:

RAM[08h] = 25h

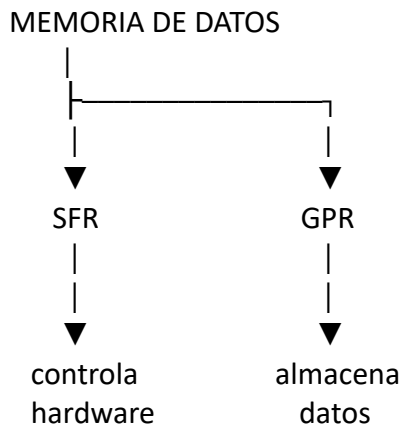
En este caso:

08h

funciona como un registro de propósito general.

SFR y GPR juntos

Ahora podemos hacer una distinción fundamental:



Por ejemplo:

STATUS → estado de CPU

TMR0 → Timer0

GPIO → pines

FSR → direccionamiento

mientras:

08h → variable

09h → variable

0Ah → variable

Ejemplo: MOVWF STATUS

Supongamos:

MOVLW 20h

MOVWF STATUS

No debemos pensar simplemente:

RAM[03h] = 20h

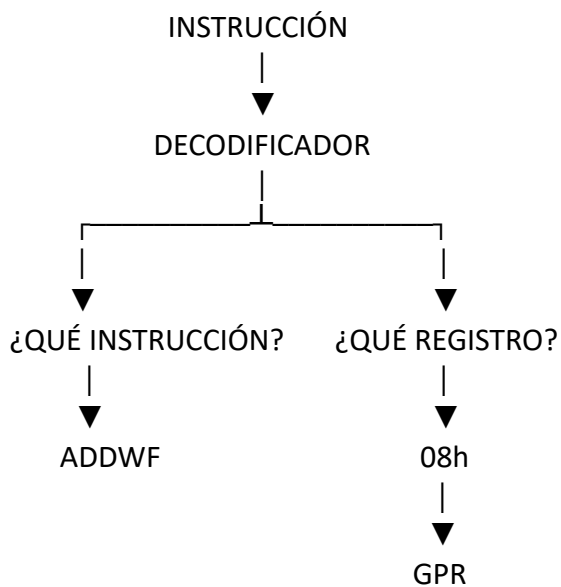
porque STATUS tiene bits con funciones específicas.

Microchip además advierte que cuando STATUS es destino de instrucciones que afectan Z, DC o C, esas escrituras tienen un comportamiento especial; TO y PD tampoco son bits normalmente escribibles de la misma manera.

Por eso el simulador deberá tener reglas especiales para STATUS.

Dos decodificaciones diferentes

Nuestro simulador tendrá conceptualmente:



Después:
ADDWF 08h,0
se convierte internamente en:

Operación = ADDWF
Registro = GPR[08h]
Destino = W
Entonces la ALU puede trabajar.

El concepto más importante de este apartado

Ya podemos formular una idea fundamental:

La memoria de datos del PIC12F509 es una interfaz entre la CPU y el hardware interno.

No es solamente almacenamiento.

Por ejemplo:

CPU



STATUS —► estado de la CPU



TMR0 —► temporizador



GPIO —► pines



FSR —► direccionamiento



OSCCAL —► oscilador

Mientras:

GPR



└─► dat

FSR, INDF Y EL DIRECCIONAMIENTO INDIRECTO

¿Por qué necesitamos el direccionamiento indirecto?

Ya conocemos el direccionamiento directo.

Por ejemplo:

MOVWF 08h

significa:

W → registro 08h

La dirección está escrita directamente dentro de la instrucción.

Eso es:

direccionamiento directo.

Pero imaginemos que queremos trabajar sucesivamente con:

08h

09h

0Ah

0Bh

0Ch

...

Tendríamos que cambiar la dirección de la instrucción cada vez.

El PIC12F509 dispone de otra posibilidad:

colocar la dirección en FSR y utilizar INDF para acceder indirectamente a ella.

Microchip define precisamente FSR como un puntero y INDF como una referencia indirecta a la posición indicada por ese puntero.

La idea más sencilla

Podemos imaginar:

FSR = dirección

y:

INDF = contenido de esa dirección

Por ejemplo:

FSR = 08h

entonces:

INDF → 08h

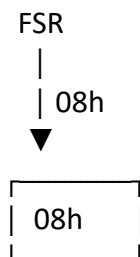
Si cambiamos:

FSR = 09h

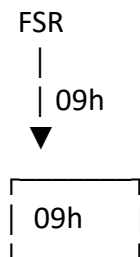
ahora:

INDF → 09h

Visualmente:



Después:



El programa no tuvo que cambiar la instrucción que utiliza INDF.

INDF no es una posición física

Este concepto es fundamental.

Si tenemos:

FSR = 08h

y ejecutamos:

MOVF INDF,W

no significa:

$W \leftarrow \text{INDF físico}$

porque:

INDF no es un registro físico.

Significa:

$W \leftarrow$ contenido de la posición indicada por FSR

Por tanto:

FSR = 08h

hace que:

MOVF INDF,W

equivalga conceptualmente a:

MOVF 08h,W

Microchip describe explícitamente este mecanismo.

Primer ejemplo

Supongamos:

RAM[08h] = 25h

y:

FSR = 08h

Ahora:

MOVF INDF,W

El recorrido es:

FSR

|

| contiene 08h



INDF

|

| direcciona 08h



RAM[08h]

|

| contiene 25h



W

Resultado:

W = 25h

Comparación con direccionamiento directo

Tenemos dos programas que pueden producir el mismo resultado.

DIRECTO

MOVF 08h,W

La dirección está dentro de la instrucción:

MOVF → 08h

INDIRECTO

MOVLW 08h

MOVWF FSR

MOVF INDF,W

Aquí:

MOVLW 08h

carga la dirección:

FSR = 08h

y después:

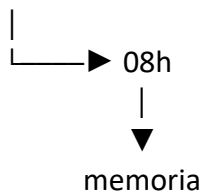
MOVF INDF,W

usa esa dirección.

Podemos representarlo:

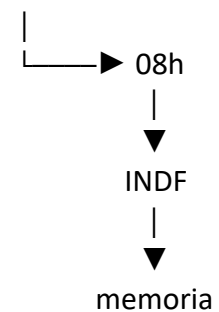
DIRECTO

instrucción



INDIRECTO

FSR



FSR como puntero

La palabra **puntero** significa simplemente:
un registro que contiene una dirección.

Supongamos:

FSR = 10h

Entonces podemos decir:

"FSR apunta a 10h."

Si hacemos:

INCF FSR,F

tenemos:

$10h + 1 = 11h$

Ahora:

FSR = 11h

El puntero avanzó.

Visualmente:

ANTES

FSR

|



10h

DESPUÉS

FSR

|



11h

Esto permite recorrer una zona de memoria.

El ejemplo oficial de Microchip

Microchip proporciona precisamente un ejemplo de este tipo.

La idea es limpiar las posiciones:

10h

11h

12h

...

1Fh

mediante direccionamiento indirecto. El programa utiliza:

```
MOVLW 10h
```

```
MOVWF FSR
```

```
NEXT:
```

```
    CLRF INDF
```

```
    INCF FSR,F
```

```
    BTFSC FSR,4
```

```
    GOTO NEXT
```

Vamos a entenderlo paso a paso.

Paso 1 — Cargar el puntero

Tenemos:

```
MOVLW 10h
```

Por tanto:

W = 10h

Después:

```
MOVWF FSR
```

por lo que:

FSR = 10h

Ahora:

FSR

|



10h

Paso 2 — Limpiar INDF

Tenemos:

CLRF INDF

Como:

FSR = 10h

la operación realmente afecta a:
RAM[10h]

Por tanto:

RAM[10h] = 00h

No se está borrando un registro físico llamado INDF.

Se está borrando:

la posición apuntada por FSR

Paso 3 — Avanzar el puntero

Ahora:

INCF FSR,F

significa:

FSR = FSR + 1

Si:

FSR = 10h

entonces:

10h + 01h = 11h

Resultado:

FSR = 11h

Ahora INDF apunta a:

11h

Paso 4 — Repetir

Volvemos:

CLRF INDF

Ahora:

RAM[11h] = 00h

Después:

INCF FSR,F

tenemos:

FSR = 12h

Y nuevamente:

CLRF INDF

produce:

RAM[12h] = 00h

El proceso continúa:

10h → 11h → 12h → 13h → ...

hasta llegar a:

1Fh

El recorrido completo

Podemos visualizarlo así:

FSR

10h —▶ RAM[10h]

|

▼

11h —▶ RAM[11h]

|

▼

12h —▶ RAM[12h]

|

▼

13h —▶ RAM[13h]

|

▼

...

|

▼

1Fh —▶ RAM[1Fh]

La instrucción:

CLRF INDF

es exactamente la misma durante todo el recorrido.

Lo único que cambia es:

FSR

Esta es la gran ventaja

Sin direccionamiento indirecto tendríamos que escribir algo parecido a:

```
CLRF 10h  
CLRF 11h  
CLRF 12h  
CLRF 13h  
CLRF 14h  
...  
CLRF 1Fh
```

Eso sería muy poco práctico.

Con FSR:

```
MOVLW 10h  
MOVWF FSR
```

```
CLRF INDF  
INCF FSR,F
```

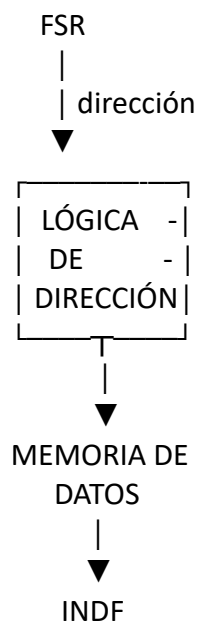
```
CLRF INDF  
INCF FSR,F
```

...

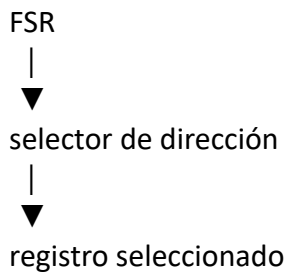
La dirección se genera mediante el puntero.

¿Qué ocurre dentro del PIC?

Podemos imaginar el hardware de esta manera:



Pero conceptualmente INDF no es una celda de memoria.
Es más correcto imaginar:



Los 5 bits inferiores de FSR

En el PIC12F509, FSR<4:0> selecciona posiciones de:
00h
hasta:
1Fh

Es decir:
00000 → 00h
00001 → 01h
00010 → 02h
...
11111 → 1Fh

Microchip especifica que esos cinco bits seleccionan la ubicación dentro del banco.
Esto encaja perfectamente con:
 $2^5 = 32$

Por tanto:
5 bits → 32 posiciones

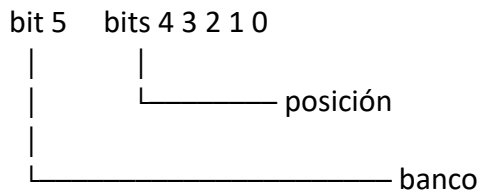
Pero el PIC12F509 tiene dos bancos

Aquí aparece la segunda parte del mecanismo.

El PIC12F509 utiliza:
FSR<5>
para seleccionar entre:
Bank 0
Bank 1

Microchip especifica expresamente que el PIC12F509 usa FSR<5> para seleccionar el banco.
Por tanto podemos pensar en FSR así:

FSR

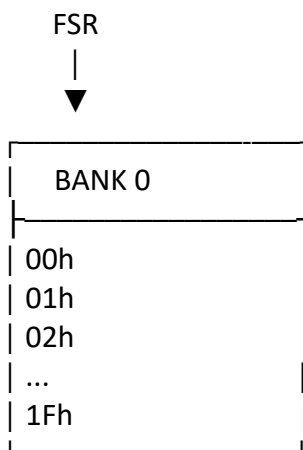


Banco 0

Si:
FSR<5> = 0
tenemos:
BANK 0

y los cinco bits inferiores seleccionan:
00h – 1Fh

Visualmente:
FSR<5> = 0



Banco 1

Si:
FSR<5> = 1
se selecciona:
BANK 1

La posición correspondiente se encuentra en la segunda región.

Por ejemplo, conceptualmente:

FSR = 28h

corresponde a:

Bank 1

posición 08h

porque:

$28h = 20h + 08h$

Entonces podemos pensar:

FSR

|
├── bit 5 = 1 → Bank 1
|
└── bits 4..0 = 08h

Una característica especial de las direcciones bajas

Las direcciones:

00h – 0Fh

tienen un comportamiento especial respecto de los bancos.

Microchip muestra en el mapa de memoria que determinadas direcciones de la zona inferior se mapean nuevamente hacia Bank 0.

Esto significa que no debemos programar suponiendo simplemente:

dirección = FSR

para todos los casos.

Necesitaremos una función que traduzca:

valor de FSR

a:

registro físico realmente seleccionado

EL REGISTRO STATUS BIT POR BIT

¿Qué es STATUS?

El registro:

STATUS

se encuentra en la dirección:

03h

y contiene información sobre:

el estado de la ALU;

el resultado de determinadas operaciones;

el estado del reset;

la selección de página de memoria de programa.

Microchip lo define precisamente como el registro que contiene el **estado aritmético de la ALU, el estado de Reset y el bit de selección de página**.

Por eso STATUS es uno de los registros que nuestro simulador deberá mostrar permanentemente.

Los ocho bits de STATUS

El registro completo es:

7	6	5	4	3	2	1	0
GPWUF	—	PA0	TO	PD	Z	DC	C

Es decir:

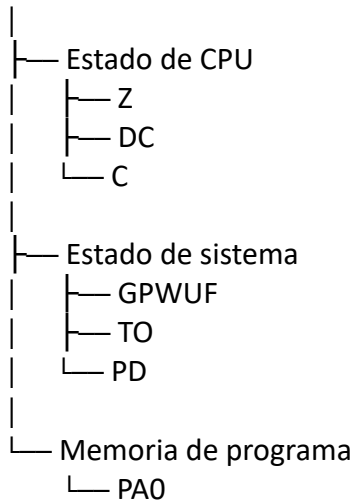
Bit	Nombre	Función
7	GPWUF	Indica wake-up por cambio en GPIO
6	—	Reservado/no usar
5	PA0	Selección de página de programa
4	TO	Time-out del Watchdog
3	PD	Estado de Power-down/SLEEP
2	Z	Resultado cero
1	DC	Carry/Borrow del nibble bajo
0	C	Carry/Borrow

Esta distribución está especificada en el datasheet del PIC12F509.

Primera clasificación

Podemos dividir los bits en tres grupos:

STATUS



Esto facilita mucho su comprensión.

Bit 0 — C

Comencemos por el bit más sencillo de visualizar:

C = Carry

Está en:

bit 0

Para determinadas operaciones aritméticas indica si se produjo un acarreo fuera del bit más significativo.

Por ejemplo:

```
  1111 1111
+ 0000 0001
-----
 1 0000 0000
```

Tenemos un resultado matemático de:

100h

pero el registro solamente puede conservar 8 bits:

0000 0000

El bit que salió por la izquierda es:

1

Por tanto:

$C = 1$

Microchip especifica este comportamiento para ADDWF.

Ejemplo con ADDWF

S

upongamos:

$W = FFh$

$f = 01h$

Ejecutamos:

ADDWF f,W

La ALU realiza:

$FFh + 01h = 100h$

Como W es de 8 bits:

$W = 00h$

y:

$C = 1$

Además:

$Z = 1$

porque el resultado almacenado es cero.

Por tanto:

ANTES

$W = FFh$

$f = 01h$

ALU

$FFh + 01h = 100h$

DESPUÉS

$W = 00h$

$C = 1$

$Z = 1$

Bit 1 — DC

El bit:

DC

está en:

bit 1

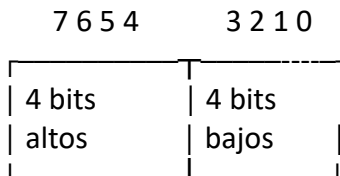
y significa:

Digit Carry / Digit Borrow

Para ADDWF, indica si hubo acarreo desde el nibble bajo.

Recordemos:

8 bits = 2 nibbles



Por ejemplo:

0Fh + 01h

en binario:

0000 1111

0000 0001

0001 0000

El nibble inferior:

1111

se desborda al sumar:

1111 + 0001 = 1 0000

Por tanto:

DC = 1

Microchip define DC de esta manera para ADDWF.

Ejemplo completo

Tenemos:

W = 0Fh

f = 01h

Ejecutamos:

ADDWF f,W

La ALU:

0Fh + 01h = 10h

Resultado:

W = 10h

Como:

10h ≠ 00h

tenemos:

Z = 0

Pero hubo acarreo del bit 3 al bit 4:

DC = 1

Y no hubo acarreo fuera del bit 7:

C = 0

Por tanto:

Z = 0

DC = 1

C = 0

Los tres bits forman:

STATUS

...	0	0	1	0
	Z	DC	C	

Bit 2 — Z

Ahora llegamos al bit:

Z

que está en:

bit 2

Significa:

Zero

Indica que el resultado de una operación aritmética o lógica que afecta a Z es cero.

Por ejemplo:

05h - 05h = 00h

Entonces:

Z = 1

Mientras:

05h + 02h = 07h

produce:

Z = 0

Una confusión muy común

No debemos pensar:

"Z indica que W vale cero."

Eso no es necesariamente cierto.

La definición correcta es:

Z indica que el resultado de la operación que afecta a Z fue cero.

Por ejemplo:

ADDWF 08h,1

puede dejar:

W = 25h

y:

RAM[08h] = 00h

En ese caso, si la operación produjo cero:

Z = 1

aunque:

W ≠ 0

Esto será importante para entender instrucciones que guardan el resultado en f.

Ejemplo

Tenemos:

W = FFh

RAM[08h] = 01h

Ejecutamos:

ADDWF 08h,1

La ALU calcula:

FFh + 01h = 100h

El resultado de 8 bits es:

00h

Como:

d = 1

el resultado se guarda en:

RAM[08h]

Por tanto:

$W = FFh$

$RAM[08h] = 00h$

y:

$Z = 1$

$C = 1$

Observemos:

$W = FFh$

no es cero.

Sin embargo:

$Z = 1$

porque **el resultado de la operación** fue cero.

Bit 3 — PD

Ahora dejamos momentáneamente la ALU.

El bit:

PD

está en:

bit 3

y significa:

Power-Down

Está relacionado con la instrucción:

SLEEP

y con el estado de bajo consumo del PIC.

Microchip especifica:

$PD = 1$

después de Power-up o CLRWDT.

Y:

$PD = 0$

después de ejecutar SLEEP.

Ejemplo con SLEEP

Supongamos:

PD = 1

Ejecutamos:

SLEEP

Entonces:

PD = 0

Podemos imaginar:

ANTES

STATUS

xxxx 1 xxx

↑

PD

SLEEP

DESPUÉS

STATUS

xxxx 0 xxx

↑

PD

Bit 4 — TO

El bit:

TO

está en:

bit 4

y significa:

Time-Out

Está relacionado con el Watchdog Timer.

Microchip especifica:

TO = 1

después de:

Power-up;

CLRWDT;

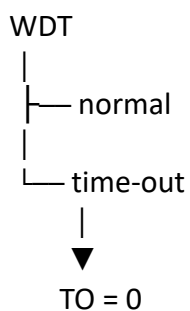
SLEEP.

Y:

TO = 0

cuando ocurre un time-out del Watchdog Timer.

Podemos visualizar:



TO y PD juntos

Estos dos bits son especialmente útiles porque ayudan a determinar qué tipo de evento ocurrió.

Tenemos:

TO

PD

y diferentes combinaciones pueden proporcionar información sobre el estado anterior del PIC.

Por ejemplo:

TO = 1

PD = 1

es compatible con un estado posterior a Power-up.

Mientras que:

TO = 1

PD = 0

puede corresponder al estado después de SLEEP.

Bit 5 — PA0

Ahora aparece un bit que conecta STATUS con la memoria de programa:

PA0

Está en:

bit 5

En el PIC12F509 selecciona la página de programa. Microchip especifica:

PA0 = 0 → página 0

PA0 = 1 → página 1

con:

Página 0: 000h – 1FFh

Página 1: 200h – 3FFh

Visualmente:

MEMORIA DE PROGRAMA

000h _____
 PÁGINA 0
 PA0 = 0
1FFh _____

200h _____
 PÁGINA 1
 PA0 = 1
3FFh _____

¿Por qué necesitamos PA0?

Porque una instrucción como:

GOTO

no contiene suficientes bits para representar por sí sola toda la dirección de programa del PIC12F509.

El hardware utiliza información adicional de página.

Por eso:

GOTO

y:

PA0

están relacionados.

Este será un punto fundamental cuando estudiemos detalladamente:

PC

PCL

PA0

GOTO

CALL

Bit 6 — reservado

El bit:

bit 6

aparece como:

—

No debemos utilizarlo como un bit normal.

Microchip lo identifica como reservado/no implementado para este registro.

Bit 7 — GPWUF

El bit:

GPWUF

está en:

bit 7

e indica una condición relacionada con el **wake-up por cambio en GPIO**.

Microchip especifica:

GPWUF = 1

cuando el reset ocurrió por wake-up desde SLEEP debido a un cambio en un pin GPIO.

En otros resets:

GPWUF = 0

Este bit pertenece más al estudio de:

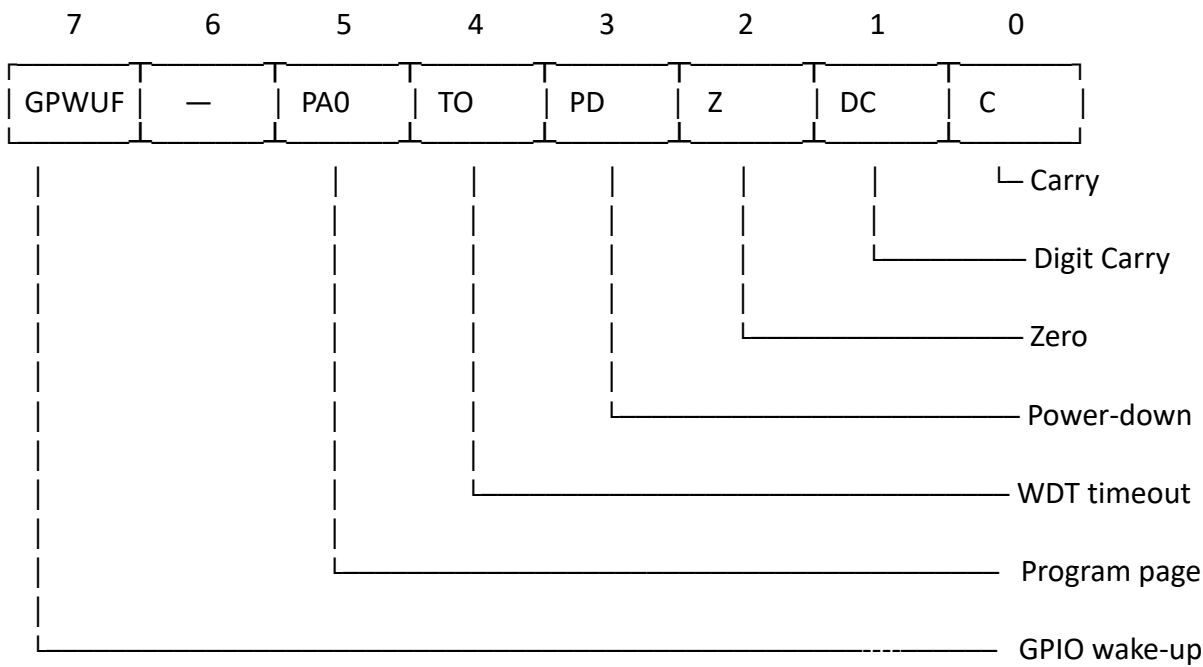
GPIO + SLEEP + RESET

que al de la ALU.

Lo estudiaremos nuevamente cuando lleguemos a esos módulos.

STATUS completo

Ahora podemos verlo completo:
STATUS — 03h



EL PROGRAM COUNTER (PC), PCL Y PA0

Ya sabemos que el PIC12F509 ejecuta las instrucciones una después de otra. Ahora debemos entender **quién indica cuál es la siguiente instrucción**.

Ese elemento es el:

PC — Program Counter (Contador de Programa)

¿Qué hace el PC?

El **PC** contiene la dirección de la instrucción que se está ejecutando o de la siguiente instrucción que debe obtenerse de la memoria de programa.

Recordemos que el PIC12F509 tiene:
1024 posiciones de memoria de programa.
Direcciones desde 000h hasta 3FFh.
Cada posición contiene una instrucción de 12 bits.

Por lo tanto:

MEMORIA DE PROGRAMA

000h —► Instrucción 1
001h —► Instrucción 2
002h —► Instrucción 3
003h —► Instrucción 4
...
3FEh —► Instrucción
3FFh —► Calibración del oscilador

El PC es el que permite recorrer esta memoria.

El PC normalmente aumenta de uno en uno

Supongamos que tenemos:

000h MOVLW 05h
001h MOVWF 08h
002h INCF 08h,F
003h GOTO 002h

Cuando el programa comienza:

PC = 000h

El PIC obtiene:
MEMORIA[000h] = MOVLW 05h

Después de ejecutar la instrucción:
PC = 001h

Ahora obtiene:
MEMORIA[001h] = MOVWF 08h

Después:
PC = 002h

Y así sucesivamente.

La secuencia normal es:

000h

↓

001h

↓

002h

↓

003h

↓

004h

↓

005h

↓

...

Microchip especifica que el PC se incrementa en cada ciclo de instrucción, excepto cuando una instrucción modifica el PC.

Pero entonces aparece GOTO

La instrucción:
GOTO 020h

significa:
"No continúes con la siguiente instrucción. Ve a la dirección 020h."

Por ejemplo:

000h MOVLW 05h

001h MOVWF 08h

002h GOTO 020h

003h MOVLW 99h

004h MOVWF 09h

...

020h CLRF 08h

Cuando se ejecuta:
GOTO 020h

el PC ya no continúa:
003h
sino que pasa a:
020h

Tenemos entonces:

```
000 → 001 → 002 |  
                   | GOTO 020h  
                   ↓  
                   020  
                   ↓  
                   021  
                   ↓  
                   022
```

Esto es lo que permite construir:
saltos,
bucles,
decisiones,
subrutinas,
programas que se repiten.

El PC no es solamente un número de 10 bits

Aquí aparece una particularidad importante del PIC12F509.

Aunque existen 1024 posiciones:
000h ————— 3FFh

el PC utilizado internamente tiene **11 bits**.

¿Por qué 11 bits?

Porque:

$$2^{11} = 2048$$

Sin embargo, el PIC12F509 solamente tiene 1024 palabras de programa disponibles:
 $2^{10} = 1024$

Por eso existe una organización particular de las páginas del programa.

Las dos páginas del PIC12F509

El espacio de programa se divide en dos páginas:

PÁGINA 0

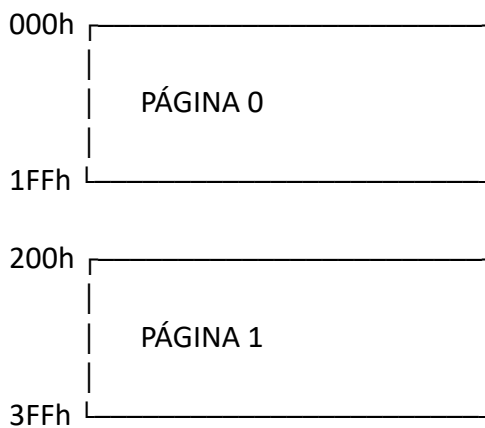
000h ————— 1FFh
512 palabras

PÁGINA 1

200h ————— 3FFh
512 palabras

Visualmente:

MEMORIA DE PROGRAMA



La selección de la página utiliza el bit:
STATUS<5> = PA0

En el PIC12F509:

PA0 = 0 → página 0

PA0 = 1 → página 1

Cada página tiene 512 palabras.

¿Qué es PCL?

Existe un registro llamado:
PCL
que se encuentra en:
02h

PCL representa:
Program Counter Low

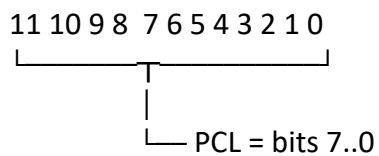
Es decir:
parte baja del Program Counter.

PCL contiene los bits bajos:
PC<7:0>

Por ejemplo, si:
PC = 0125h

entonces:
PCL = 25h

Podemos visualizarlo así:
PC

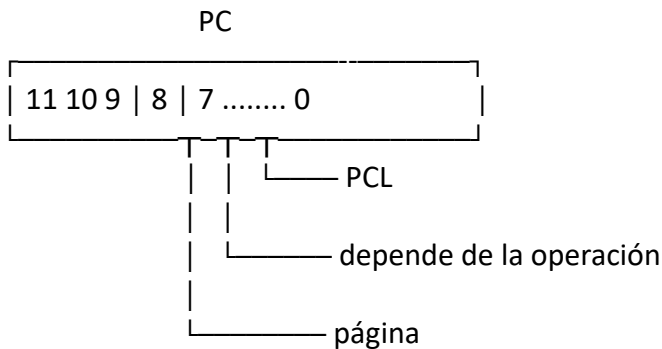


El byte superior del PC no es directamente accesible como un registro normal.

¿Dónde está el resto del PC?

Aquí está una de las características interesantes del PIC12F509.

El PC puede representarse conceptualmente como:



Para un GOTO, los bits correspondientes al destino se combinan con el bit de página PA0.

Microchip especifica que para GOTO, los bits 8:0 del PC proceden de la palabra de instrucción y que STATUS<5> proporciona el bit de página para el PC.

Ejemplo con GOTO

Supongamos:
GOTO 025h

Si estamos en:
PA0 = 0
estamos en página 0:
000h ——— 1FFh

Por lo tanto el salto será dentro de la página correspondiente.

Ahora supongamos:
PA0 = 1

Entonces estamos en:
200h ——— 3FFh

El mismo concepto de salto utiliza ahora la página 1.

Por eso debemos pensar en:

DESTINO = página + dirección
No basta con mirar solamente los bits bajos.

¿Cómo cambiamos de página?

El programa puede modificar:

STATUS<5>

es decir:

PA0

mediante instrucciones que permitan modificar ese bit.

Por ejemplo, conceptualmente:

BSF STATUS,5

significa:

STATUS<5> = 1

y por tanto:

PA0 = 1

Se selecciona la página 1.

Para regresar:

BCF STATUS,5

produce:

PA0 = 0

y se selecciona la página 0.

Esto será particularmente importante cuando el programa tenga más de 512 instrucciones.

CALL y el PC

Otra instrucción que modifica el PC es:

CALL dirección

Pero CALL tiene una función adicional.

Supongamos:

000h MOVLW 10h

001h CALL 050h

002h MOVWF 08h

...

050h ...

051h ...

Cuando se ejecuta:

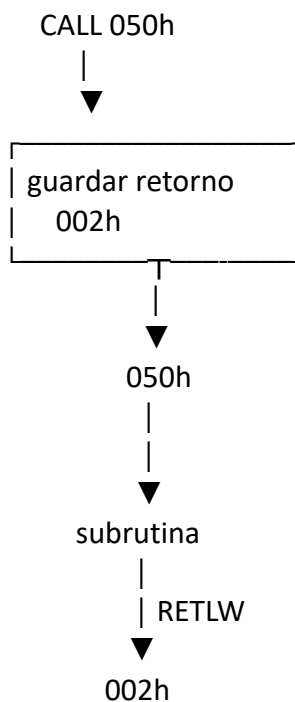
CALL 050h

el PIC necesita recordar:

"Cuando termine la subrutina, debo regresar a la instrucción 002h."

Por eso CALL guarda una dirección de retorno en la pila hardware.

Conceptualmente:



El PIC12F509 posee una pila hardware de **2 niveles y 12 bits**. CALL almacena la dirección de retorno y RETLW recupera la dirección guardada.

Una característica muy importante de CALL

En el PIC12F509 hay una diferencia entre:

GOTO

y:

CALL

Para GOTO, los bits de destino permiten utilizar la información de página.

Pero en CALL, y también en instrucciones que modifican PCL, el bit:

PC<8>

se fuerza a cero.

Esto significa que las llamadas a subrutinas y los saltos calculados mediante modificación de PCL están limitados a las primeras 256 posiciones de cada página de 512 palabras.

¿Qué ocurre al hacer MOVWF PCL?

Recordemos:

MOVWF f

copia:

W → f

Pero PCL es un registro especial.

Por ejemplo:

MOVLW 20h

MOVWF PCL

conceptualmente significa:

W = 20h



MOVWF PCL



PCL = 20h



PC cambia



el programa continúa
desde otra dirección

Por eso PCL permite realizar **saltos calculados**.

Microchip identifica MOVWF PC, ADDWF PC y BSF PC,5 entre las instrucciones que modifican el PC.

¿Qué sucede después de un RESET?

Esta parte es especialmente interesante.

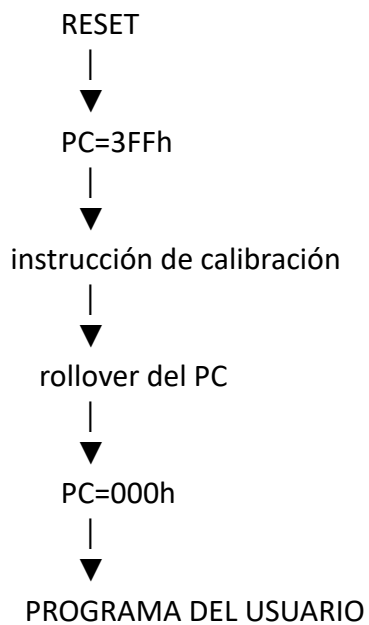
En el PIC12F509, después de un Reset, el PC se coloca en la última posición de la memoria de programa, donde se encuentra la instrucción de calibración del oscilador.

Es decir:
PC = 3FFh

Después de ejecutarse la instrucción MOVLW de calibración, el PC continúa y finalmente hace rollover hacia:
000h

donde comienza el programa del usuario.

Además, después del Reset se selecciona inicialmente la página 0.
Podemos representarlo:



2.24 — CONJUNTO DE INSTRUCCIONES DEL PIC12F509

El PIC12F509 dispone de **33 instrucciones**.

Ya conocemos qué son una instrucción, f, d, b y k, por lo que ahora estudiaremos directamente qué hace cada instrucción.

Para evitar repeticiones, utilizaremos siempre la misma ficha:

INSTRUCCIÓN

Sintaxis

Operación

STATUS afectado

Ejemplo

Resultado

Grupo 1 — Transferencia de datos

Estas instrucciones permiten mover información entre:

una constante y W,

W y un registro,

un registro y W.

Son las primeras instrucciones que necesitaremos para escribir prácticamente cualquier programa.

MOVLW

Sintaxis

MOVLW k

k es un valor de 8 bits.

Operación

$W \leftarrow k$

Ejemplo:

MOVLW 25h

Resultado:

W = 25h

Ejemplo decimal:

MOVLW 10

significa:

W = 0Ah

si el ensamblador interpreta 10 como decimal.

STATUS - MOVLW no modifica Z, DC ni C.

MOVWF

Sintaxis

MOVWF f

Operación

$f \leftarrow W$

Ejemplo:

MOVLW 25h

MOVWF 08h

Después:

$W = 25h$

$RAM[08h] = 25h$

W conserva su valor.

STATUS

No modifica los indicadores Z, DC y C.

MOVF

Sintaxis

MOVF f,d

Esta instrucción lee el registro f.

El destino depende de d.

d = 0

$W \leftarrow f$

Ejemplo:

MOVF 08h,W

Si:

$RAM[08h] = 37h$

entonces:

$W = 37h$

d = 1

$f \leftarrow f$

El contenido permanece en el mismo registro.

Aunque parezca extraño, esta forma es útil porque la instrucción puede utilizarse para comprobar si el contenido es cero.

STATUS

MOVF afecta el bit:

Z

pero no C ni DC.

Por ejemplo:

RAM[08h] = 00h

ejecutando:

MOVF 08h,F

produce:

Z = 1

CLRW

Sintaxis

CLRW

Operación

$W \leftarrow 00h$

Además:

Z = 1

Ejemplo:

MOVLW 55h

tenemos:

W = 55h

Después:

CLRW

queda:

W = 00h

Z = 1

Primer pequeño programa

Ahora podemos utilizar las cuatro instrucciones juntas.

MOVLW 25h

MOVWF 08h

MOVLW 40h

MOVWF 09h

Seguimos el programa:

PC	Instrucción	W antes	Acción
000h	MOVLW 25h	00h	W = 25h
001h	MOVWF 08h	25h	RAM[08h] = 25h
002h	MOVLW 40h	25h	W = 40h
003h	MOVWF 09h	40h	RAM[09h] = 40h

Estado final:

W = 40h

RAM[08h] = 25h

RAM[09h] = 40h

Aquí ya tenemos un programa real que almacena dos valores.

Grupo 2 — Operaciones aritméticas

Ahora estudiaremos las instrucciones que realizan operaciones numéricas.

ADDWF

SUBWF

INCF

DECF

INCFSZ

DECFSZ

ADDWF

Sintaxis

ADDWF f,d

Operación

$W + f$

El resultado se coloca según d.

$d = 0 \rightarrow W$

$d = 1 \rightarrow f$

Ejemplo:

W = 05h

RAM[08h] = 03h

Ejecutamos:

ADDWF 08h,W

Resultado:

$05h + 03h = 08h$

Por tanto:

$W = 08h$

$RAM[08h] = 03h$

Con:

`ADDWF 08h,F`

queda:

$W = 05h$

$RAM[08h] = 08h$

STATUS

Afecta:

Z

DC

C

SUBWF

Sintaxis

`SUBWF f,d`

Aquí debemos prestar mucha atención al orden.

La operación es:

$f - W$

No:

$W - f$

Ejemplo:

$RAM[08h] = 10h$

$W = 03h$

Entonces:

`SUBWF 08h,W`

produce:

$10h - 03h = 0Dh$

Resultado:

$W = 0Dh$

Si usamos:

`SUBWF 08h,F`

entonces:

$RAM[08h] = 0Dh$

STATUS

Afecta:

Z

DC

C

Pero en resta hay una particularidad importante:

$C = 1 \rightarrow$ no hubo borrow

$C = 0 \rightarrow$ hubo borrow

Lo mismo aplica al significado de DC para la resta.

INCF

Sintaxis

INCF f,d

Incrementa en uno:

$f + 1$

Ejemplo:

RAM[08h] = 05h

Ejecutamos:

INCF 08h,F

Resultado:

RAM[08h] = 06h

Si:

RAM[08h] = FFh

entonces:

$\text{FFh} + 1 = 00\text{h}$

El resultado se limita a 8 bits:

RAM[08h] = 00h

y:

Z = 1

Importante

INCF **no debe tratarse como ADDWF.**

DECF

Sintaxis

DECF f,d

Decrementa:

$f - 1$

Ejemplo:

RAM[08h] = 05h

después:

DECF 08h,F

queda:

RAM[08h] = 04h

Caso especial:

RAM[08h] = 00h

entonces:

$00h - 1 = FFh$

Resultado:

RAM[08h] = FFh

INCF y DECF como contadores

Estas instrucciones son especialmente útiles para construir contadores.

Por ejemplo:

MOVLW 00h

MOVWF 08h

BUCLE:

INCF 08h,F

GOTO BUCLE

La memoria irá cambiando:

00h

01h

02h

03h

...

FFh

00h

01h

...

Este comportamiento será muy sencillo de visualizar en nuestro simulador.

INCFSZ

Sintaxis

INCFSZ f,d

Significa:

Increment File, Skip if Zero

Es decir:

Incrementa f.

Si el resultado es cero, salta la siguiente instrucción.

Ejemplo:

INCFSZ 08h,F

GOTO BUCLE

Si inicialmente:

RAM[08h] = FEh

después de INCFSZ:

RAM[08h] = Ffh

No es cero.

Por tanto:

GOTO BUCLE

sí se ejecuta.

Pero si inicialmente:

RAM[08h] = FFh

tenemos:

$\text{FFh} + 1 = \text{00h}$

Entonces:

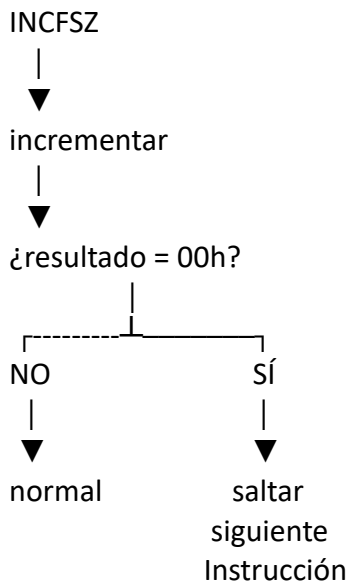
RAM[08h] = 00h

y la siguiente instrucción:

GOTO BUCLE

se salta.

Visualmente:



Esta instrucción será muy importante para nuestros futuros programas de prueba.

DECFSZ

Es equivalente conceptualmente, pero decrementando.

Sintaxis

DECFSZ f,d

Proceso:

$f \leftarrow f - 1$

y si el resultado es:

00h

se salta la siguiente instrucción.

Ejemplo:

MOVLW 03h

MOVWF 08h

BUCLE:

DECFSZ 08h,F

GOTO BUCLE

La secuencia es:

03h

↓

02h

↓

01h

↓

00h

Cuando llega a cero:

GOTO BUCLE

se salta.

Algo nuevo: una instrucción puede alterar el flujo sin ser GOTO

Esto es importante para comprender el procesador.

Hasta ahora vimos:

GOTO → cambia PC

CALL → cambia PC

Pero ahora aparece otra posibilidad:

INCFSZ

DECFSZ

BTFSC

BTFSS

Estas instrucciones pueden hacer que la **siguiente instrucción no se ejecute**.

Por ejemplo:

DECFSZ 08h,F

GOTO BUCLE

Si el resultado es cero:

DECFSZ

|

|— ejecuta

|

|— salta la siguiente instrucción

|

X

GOTO

Grupo 3 — Operaciones lógicas

Las siguientes instrucciones trabajan bit por bit:

ANDWF

IORWF

XORWF

COMF

SWAPF

ANDWF

Sintaxis

ANDWF f,d

Operación:

W AND f

Ejemplo:

W = A5h

f = 0Fh

En binario:

1010 0101

0000 1111

0000 0101

Resultado:

05h

IORWF

Sintaxis

IORWF f,d

Operación:

W OR f

Ejemplo:

1010 0000

0000 1111

1010 1111

Resultado:

AFh

XORWF

Sintaxis

XORWF f,d

Operación:

W XOR f

Ejemplo:

1010 0101

0000 1111

1010 1010

Resultado:

AAh

COMF

Sintaxis

COMF f,d

Realiza el complemento bit a bit:

0 → 1

1 → 0

Ejemplo:

f = 1010 0101

Después:

0101 1010

Por tanto:

A5h → 5Ah

SWAPF

Esta instrucción intercambia los dos nibbles.

Un byte tiene:

alto bajo

1111 0000

Por ejemplo:

A5h

en binario:

1010 0101

Después de SWAPF:

0101 1010

es decir:

5Ah

Por tanto:

AB

conceptualmente se transforma en:

BA

cuando hablamos de los dos nibbles hexadecimales.

Grupo 4 — Rotaciones

Tenemos:

RLF

RRF

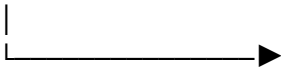
Estas instrucciones son diferentes de un simple desplazamiento porque utilizan el bit C de STATUS.

RLF

RLF f,d

Rotación hacia la izquierda:

f7 f6 f5 f4 f3 f2 f1 f0



El bit que sale por la izquierda pasa a:

C

y el antiguo C entra por la derecha.

Ejemplo conceptual:

f = 10110010

C = 1

Después:

01100101

y:

C = 1

RRF

Es el proceso inverso:

RRF f,d

El bit que sale por la derecha pasa a C.

El antiguo C entra por la izquierda.

Por eso estas instrucciones deben modelarse cuidadosamente en el simulador.

Grupo 5 — Operaciones individuales sobre bits

Aquí tenemos:

BCF

BSF

BTFSC

BTFSS

Estas instrucciones son particularmente importantes para trabajar con GPIO y STATUS.

BCF

Bit Clear f

BCF f,b

Pone el bit indicado en cero.

Ejemplo:

BCF 06h,2

significa:

$\text{GPIO}<2> = 0$

si 06h corresponde al registro GPIO en ese contexto.

BSF

Bit Set f

BSF f,b

Pone el bit indicado en uno.

Ejemplo:

BSF 06h,2

produce:

$\text{GPIO}<2> = 1$

BTFSC

Bit Test f, Skip if Clear

Comprueba un bit.

Si está:

0

se salta la siguiente instrucción.

Ejemplo:

BTFSC GPIO,0

GOTO ENCENDIDO

Si:

$\text{GPIO}<0> = 0$

entonces:

GOTO ENCENDIDO

se salta.

Si:

$\text{GPIO}<0> = 1$

entonces sí se ejecuta.

BTFSS

Es lo contrario a la anterior:

Bit Test f, Skip if Set

Si el bit es:

1

se salta la siguiente instrucción.

Una pareja fundamental

Podemos recordar:

BTFSC

↓

Skip if Clear

↓

salta si = 0

y:

BTFSS

↓

Skip if Set

↓

salta si = 1

Esta diferencia será esencial cuando hagamos programas de entrada/salida.

Grupo 6 — Operaciones con literales

Estas instrucciones trabajan directamente con W y una constante:

ANDLW

IORLW

XORLW

La estructura es:

W + literal

o, según la operación:

W AND k

W OR k

W XOR k

Ejemplo:

MOVLW A5h

ANDLW 0Fh

Resultado:

W = 05h

Grupo 7 — Control del programa

Aquí encontramos:

GOTO

CALL

RETLW

Ya hemos estudiado su función general, por lo que solamente fijaremos su comportamiento.

GOTO

PC → dirección de destino

CALL

STACK ← dirección de retorno

PC → subrutina

RETLW

W ← literal

PC ← dirección almacenada en STACK

Un ejemplo:

CALL TABLA

...

TABLA:

RETLW 25h

Al regresar:

W = 25h

Esto hace que RETLW sea especialmente útil para implementar **tablas de datos**.

Grupo 8 — Control del microcontrolador

Finalmente tenemos:

OPTION

TRIS

CLRWDT

SLEEP

NOP

Estas no son simplemente operaciones matemáticas.

Controlan aspectos especiales del dispositivo.

OPTION

Configura funciones relacionadas con:

Timer0,
prescaler,
fuente del Timer0,
flanco de T0CKI,
pull-ups de GPIO,
wake-up por cambio de GPIO.

TRIS

Configura la dirección de GPIO:

1 → entrada

0 → salida

CLRWDT

Limpia el Watchdog Timer.

SLEEP

Coloca al PIC en modo de bajo consumo.

NOP

No realiza una operación visible sobre los datos.

Su función principal es consumir un ciclo de instrucción.

Tabla maestra inicial

Ya podemos reunir las 33 instrucciones:

Grupo	Instrucciones
Transferencia	MOVLW, MOVWF, MOVF, CLRW
Aritmética	ADDWF, SUBWF, INCF, DECF, INCFSZ, DECFSZ
Lógica	ANDWF, IORWF, XORWF, COMF, SWAPF
Rotación	RLF, RRF
Bits	BCF, BSF, BTFSC, BTFSS
Literales	ANDLW, IORLW, XORLW
Programa	GOTO, CALL, RETLW
Control	OPTION, TRIS, CLRWDT, SLEEP, NOP

Total:

$$4 + 6 + 5 + 2 + 4 + 3 + 3 + 5$$

$$= 32$$

Aquí aparece algo que debemos corregir antes de continuar: **la tabla anterior suma 32, no 33.**

La instrucción que falta es:

DECFSZ

ya está incluida en las seis aritméticas,

Fichas completas de instrucciones

Cada instrucción tendrá una ficha como:

ADDWF f,d
Operación: W + f
Destino: W o f
STATUS: Z, DC, C
Ciclos: 1
Palabra: 12 bits
EJEMPLO
W = 05h
f = 03h
05h + 03h = 08h
Resultado = 08h