FACULTAD DE CIENCIAS
FÍSICAS Y MATEMÁTICAS
UNIVERSIDAD DE CHILE

Introducción al lenguaje ensamblador

EL4002 - Sistemas Digitales

Semestre Primavera - 2016

Profesor:
Ricardo Finger
Ayudantes: Francisco Casado, Ricardo Ramos

Generalidades

Esta guía pretende dar una introducción de los conceptos básicos relacionados a los lenguajes ensambladores, en particular, para el del micro-controlador **PIC18F4550**, de la empresa Microchip. Es indispensable acompañar esta guía con la información contenida en la hoja de datos del micro-controlador, disponible en ✐

Además es imprescindible tener una noción del funcionamiento del hardware que se utiliza, porque lo que la primera sección se avoca a explicarlo brevemente.

1 Arquitectura PIC18

El término **arquitectura** se usa para referirse a cómo está estructurada la lógica interna de un sistema digital, en cuanto a cómo es el *flujo de información*. En este sentido, los micro-controladores están orientados a realizar operaciones lógicas y aritméticas de manera secuencial sobre información almacenada en **registros**.

La familia de micro-controladores PIC18 (y varios otros de la empresa Microchip) tienen una arquitectura como lo muestra la figura 1.

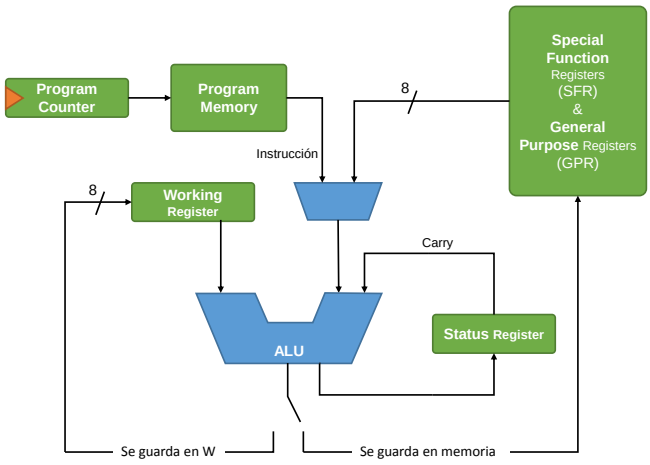


Figure 1: Arquitectura simplificada del *PIC18*.

Como se puede observar, el corazón del microprocesador es la **ALU** (*Aritmetic-Logic Unit*), la cual puede operar siempre con el **working register** (8 bits) y eventualmente con información proveniente de la memoria de programa, registros de funciones especiales ó registros de propósito general. Nótese además que se dispone del registro **Status**, el cual es usado por algunas operaciones específicas, como por ejemplo, las que involucran *carries*.

A grandes rasgos, lo que hace el micro-controlador es ejecutar **instrucciones**, almacenadas en la memoria de programa, de manera secuencial y síncrona (toma como referencia temporal un reloj), operando la información de la memoria de datos (*GPR*) y del **working register**.

Como ya debería haber notado, el PIC18 es un ejemplo de arquitecturas de tipo *Harvard*, puesto que tiene separadas ambas memorias. A continuación se describen brevemente cada una de ellas y otros bloques relevantes.

[1] Ver página 57 de la hoja de datos.

1.1 Memoria de Programa

Es un bloque de memoria[1] donde se almacenan palabras de 16 bits. Cada una de estas palabras corresponde a una **instrucción**. Es decir, cuando se programa un micro-controlador, todas las instrucciones que realizará se guardan en este bloque. La cantidad de instrucciones que se pueden escribir en esta memoria (profundidad) varía según la familia y el modelo.

Existen varios tipos de instrucciones[2]y se clasifican según los registros que se operan:

- Operaciones orientadas a Bytes: corresponden a operaciones aritméticas entre el **working register** y/o registros de la memoria de datos. El resultado puede gardarse en el **working register** ó en la memoria de datos,
- Operaciones orientadas a bits: operan sobre los mismos registro anteriormente mencionados, sin embargo modifican a lo más 1 bit de ellos.
- Operaciones de control: Cambian el valor del **Program Counter** para cambiar la instrucción que se ejecutará en el siguiente ciclo.
- Operaciones de literales: Se carga un valor en el **working register** a partir de una **LUT** (*Look-Up Table*) interna.

La pregunta entonces es **¿Cómo ejecutar cada instrucción de manera secuencial?**

Program Counter

Es uno de lo los registros más importantes, ya que el valor que contiene es la dirección de la memoria de programa de la instrucción que se ejecuta en cada momento. Es decir, es el encargado de que el programa *avance*. Las instrucciones se ejecutan por **ciclos**, cada vez que se termina de ejecutar una instrucción, se incrementa (o actualiza) el valor del **program counter** para pasar a la instrucción siguiente.

1.2 Memoria de Datos

Corresponde al segundo bloque de memoria del micro-controlador, donde se almacena toda la información que se operará y la que ya se ha operado. En este sentido, pueden ser dos números que serán sumados, o el resultado de una multiplicación, etc. Las palabras en esta memoria tienen largo 8 bits.

Ahora bien, es cierto que esta memoria se puede leer o escribir a voluntad, sin embargo, existen registros reservados que sirven para configurar todas las funciones del micro-controlador y sus periféricos (*ADC, GPIO,* puertos seriales, etc.), los que se conocen como **SFR** (*Special Function Register*). Estos no se pueden modificar tan libremente.

1.3 Working Register

Este registro es por el que pasará la mayor cantidad de información durante un programa. De la figura 1 se puede ver que siempre será uno de los operandos de la **ALU** y es por esto mismo que para operar dos palabras que se encuentran en la memoria de datos es necesario primero cargar uno de ellos al **working register** y en el siguiente **ciclo de instrucción** hacer la operación. Más detalles en la sección

1.4 Resumiendo todo

Lo que realiza un micro-controlador es lo siguiente:

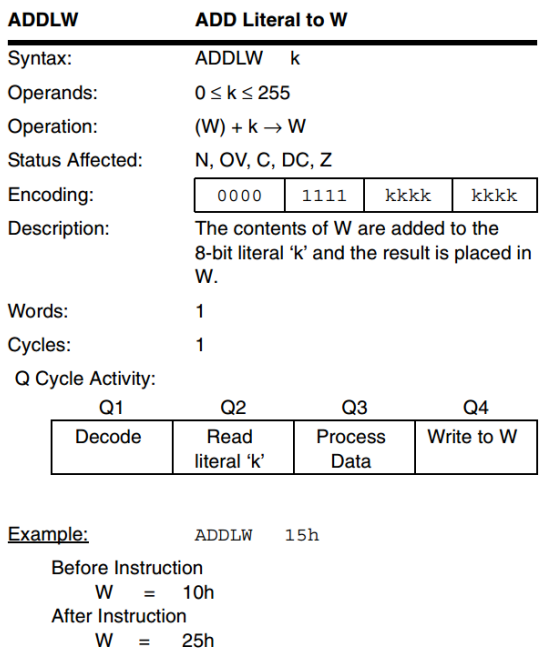
1) Actualizar el **Program Counter** y cargar la nueva instrucción,
2) Operar los registros a la entrada de la **ALU** de acuerdo a la instrucción en curso,
3) Realizar alguna de las siguientes operaciones:
   - Operaciones orientadas a bits y bytes: Guardar el resultado en el registro correspondiente,
   - Operación de **branching** o saltos: Cambiar el valor del **program counter** para ir a otra parte del programa.

2 Programación y Compilación: Assembler

[2]Más información sobre la **instrucciones** del PIC18 se encuentra en la página 307 de la hoja de datos.

La programación del **PIC18F4550** puede hacerse en lenguajes de alto nivel (por ej. C) o en el lenguaje de bajo nivel **Assembler**. Este lenguaje tiene la propiedad de estar directamente relacionado con el hardware que se está utilizando, ya que todos los comandos corresponden exáctamente a lo que se realiza a niver de compuertas lógicas, a diferencia de los lenguajes de alto nivel.

¿Qué comandos tiene Assembler? La respuesta a esta pregunta es tan variada como arquitecturas de micro-controladores existe. Es decir, cada arquitectura tiene su propio lenguaje ensamblador. Más aún, las **instrucciones** antes mencionadas corresponden al conjunto de comandos que se utilizan para programar un microcontrolados. Como ya se dijo, para el caso del PIC18F4550, todos las **instrucciones** (comandos) se encuentran en la página 307 de la hoja de datos. Y posteriormente se desglosa cada una con todas sus particularidades, por ejemplo, para la instrucción `addlw` se tiene la siguiente descripción:



ADDLW	**ADD Literal to W**			
Syntax:	ADDLW k			
Operands:	$0 \le k \le 255$			
Operation:	$(W) + k \rightarrow W$			
Status Affected:	N, OV, C, DC, Z			
Encoding:	0000	1111	kkkk	kkkk
Description:	The contents of W are added to the 8-bit literal 'k' and the result is placed in W.			
Words:	1			
Cycles:	1			

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read literal 'k'	Process Data	Write to W

Example: ADDLW 15h

Before Instruction
W = 10h
After Instruction
W = 25h

Figure 2: Descripción de `addlw`.

Pero momento...¿Cómo se *cargan* estas instrucciones en la memoria de programa? La respuesta es simple: el lenguaje ensamblador es una forma más humanamente cómoda de escribir los unos y ceros que se escriben en la memoria de programa. Si se presta atención a la figura 2, existe un item denominado *Encodig* que muestra una palabra de 16 bits. Esta secuencia de ceros y unos es lo que realmente se escribe en al memoria de datos. Para hacer la traducción de **assembler** a ceros y unos existen software denominados **compiladores** (¿Le suena esa palabra?).

Ahora bien, existen otras sintaxis que también admite el lenguajes, además de las instrucciones, que facilitan la programación.

2.1 Asignación de variables

Para hacer uso de la memoria de datos es necesario escribir la dirección en la que se almacena un cierto dato. Un ejemplo de esto es el siguiente código:

```
movlw 0x0A
movwf 0x000
movlw 0x03
movwf 0x001
```

Este código carga el literal `0x0A` ($= 10_{10} = 1010_2$) en el working register (`movlw`), luego guarda el working register en la dirección `0x000` de la memoria de programa (`movwf`). Nuevamente carga otro literal en el working register (`0x03`$= 3_{10} = 11_2$) y lo guarda en la dirección `0x001`. No hay mayor problema con entenderlo, pero cuando los códigos poseen más líneas y más variables se hace necesario tener un nombre más amigable para estas direcciones. Es por esto que se puede usar una **asignación de variables** que usa la siguienre sintaxis:

```
A equ 0x000
B equ 0x001
```

Así, las direcciones `0x000` y `0x001` tendrán los nombres `A` y `B`, respectivamente, con lo que el código se puede escribir como

```
A equ 0x000
B equ 0x001

movlw 0x0A
movwf A
movlw 0x03
movwf B
```

2.2 Etiquetas (labels)

Así como se puede asignar un nombre a una dirección de la memoria de datos, también se puede asignar nombre a direcciones de la memoria de programa. En estricto rigor, **no** son líneas de instrucciones, si no sólo información para que el compilador haga su trabajo. Es decir, no se traducirán como ceros y unos a la memoria de programa.

Esto es útil, por ejemplo, para cuando se guiere hacer **saltos** desde una rutina a otra. Ejemplo de esto es el código siguiente:

```
    A equ 0x000
    B equ 0x001

    org 0x0000
    goto start

configurar
    movlw 0x0A
    movwf A
    movlw 0x03
    movwf B
    return

operar
    movf A, 0
    andwf B, 0
    return

start
    call configurar
    call operar

loop
    goto loop
```

La línea `org 0x0000` se usa para inciar el **program counter** en esa dirección (la primera de la memoria). El compilador traducirá todas las instrucciones escritas hacia abajo, sin considerar los espacios entre líneas.

Entonces, la primera instrucción corresponde a un **salto** (`goto start`) para cambiar el valor del **program counter** hasta la dirección en la que está la linea `call configurar`. Esta instrucción salta hasta la instrucción `movlw 0x0A`. De aquí en adelante el código se ejecuta línea a línea hasta llegar a `return`, donde se salta hasta la línea `call operar`. Ocurre otro salto más a la línea `movf A, 0`, se ejecuta la operación `andwf B, 0` y se vuelve a la línea siguiente a `call operar`, que en este caso sería `goto loop`. Como se puede notar, este programa que por siempre saltando a `goto loop`. Estas líneas se suelen agregar para que el micro controlador quede encendido, pues la mayoría, al terminar de correr la memoria de programa, entra en modo *ahorro de energía*.

Dicho todo esto, el proceso de compilación es el que se ilustra en la figura 3.

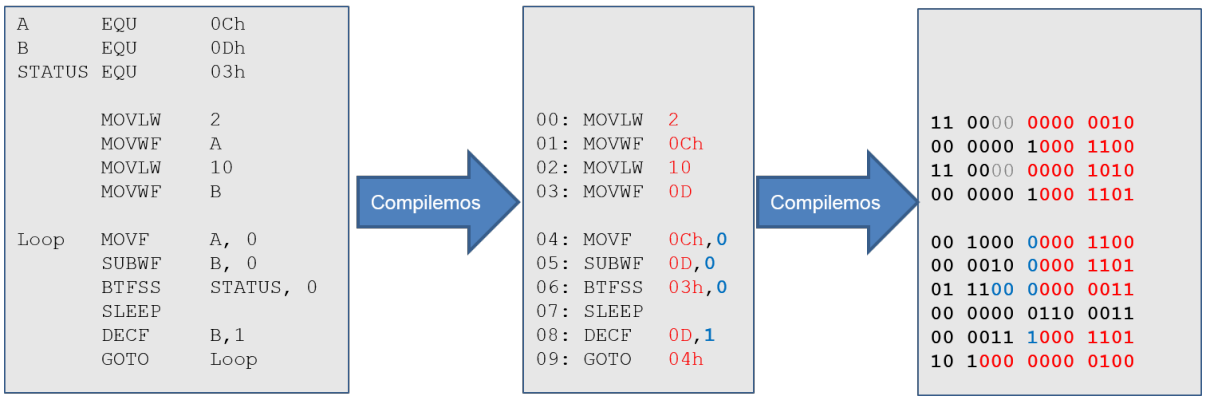


Figure 3: Proceso de compilación de un programa.

3 Frecuencia del reloj y Ciclos de instrucción

Contra la intuición, la frecuencia a la que se ejecutan las instrucciones no siempre corresponde a la frecuencia del reloj (`clk`) que usa el microcontrolador. Para el caso del PIC18, cada instrucción se ejecuta en un tiempo equivalente a 4 periodos de de reloj. Esto porque así puede implementar un *flujo de información* que funcionar de manera más eficiente[3]

Para ser más exactos, una instrucción se puede ejecutar en 1, 2 y hasta 3 **ciclos de instrucción**, los que equivalen a los ya mencionados 4 periodos de reloj. Por ejemplo, para la instrucción `decfsz` (ver figura 4), la cual "*decrementa* el dato contenido en `f` y salta una línea de programa si el resultado (dato en `f`-1) es cero", utiliza 1 ciclo para realizar la resta ó ocupa al menos 2 si es que el resultado de la resta es cero, ya que la siguiente instrucción no será realizada. El siguiente código muestra un ejemplo:

```
delay
    decfsz A
    goto delay
```

Este trozo de código descuenta 1 a al valor almacenado en `A` y luego se itera sobre si mismo (`goto delay`. Pero cuando este valor llega a cero, en vez de volver a decrementar el valor de `A`, sigue con la siguiente línea de código (si es que existe).

DECFSZ — **Decrement f, Skip if 0**

Syntax: DECFSZ f {,d {,a}}

Operands: $0 \le f \le 255$, $d \in [0,1]$, $a \in [0,1]$

Operation: (f) – 1 → dest, skip if result = 0

Status Affected: None

Encoding:

0010	11da	ffff	ffff

Description: The contents of register 'f' are decremented. If 'd' is '0', the result is placed in W. If 'd' is '1', the result is placed back in register 'f' (default). If the result is '0', the next instruction, which is already fetched, is discarded and a NOP is executed instead, making it a two-cycle instruction. If 'a' is '0', the Access Bank is selected. If 'a' is '1', the BSR is used to select the GPR bank (default). If 'a' is '0' and the extended instruction set is enabled, this instruction operates in Indexed Literal Offset Addressing mode whenever f ≤ 95 (5Fh). See **Section 26.2.3 "Byte-Oriented and Bit-Oriented Instructions in Indexed Literal Offset Mode"** for details.

Words: 1

Cycles: 1(2)
Note: 3 cycles if skip and followed by a 2-word instruction.

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	Write to destination

If skip:

Q1	Q2	Q3	Q4
No operation	No operation	No operation	No operation

If skip and followed by 2-word instruction:

Q1	Q2	Q3	Q4
No operation	No operation	No operation	No operation
No operation	No operation	No operation	No operation

Figure 4: Instrucción `decfsz`.

[3]Si quiere saber más, tome el curso *EL4102-Arquitectura de Computadores* o lea sobre procesadores *single-cycle* y *pipelining*.