Microprocesadores y microcontroladores 3

Fichas 3 y 4

MINISTERIO de EDUCACIÓN CIENCIA y TECNOLOGÍA PRESIDENCIA de la NACIÓN

inet
Instituto Nacional de Educación Tecnológica

serie/desarrollo de contenidos
colección/fluídica y controladores lógicos programables

Autoridades

Presidente de la Nación
Néstor C. Kirchner

Ministro de Educación, Ciencia y Tecnología
Daniel Filmus

Directora Ejecutiva del Instituto Nacional de Educación Tecnológica
María Rosa Almandoz

Director Nacional del Centro Nacional de Educación Tecnológica
Juan Manuel Kirschenbaum

Especialista en contenidos

- Marcelo Estévez

Todos los derechos reservados. Ley 11.723
Ministerio de Educación, Ciencia y Tecnología
Instituto Nacional de Educación Tecnológica
Saavedra 789. C1229ACE
Ciudad Autónoma de Buenos Aires
República Argentina

serie/desarrollo de contenidos

Colecciones

- Autotrónica
- Comunicación de señales y datos
- Diseño gráfico industrial
- Electrónica y sistemas de control
- Fluídica y controladores lógicos programables
    1. Tecnología neumática
    2. Controladores lógicos programables –PLC–
    3. Microprocesadores y microcontroladores
- Gestión de la calidad
- Gestión de las organizaciones
- Informática
- Invernadero computarizado
- Laboratorio interactivo de idiomas
- Procesos de producción integrada
- Proyecto tecnológico
- Unidades de cultura tecnológica

Índice

El Centro Nacional de Educación Tecnológica 7

¿De qué se ocupa *Microprocesadores y microcontroladores*?
- El problema tecnológico 12
- Las primeras decisiones 13

Ficha 1. Introducción a los sistemas basados en microprocesador
- Referencia histórica 23
- Sistema mínimo microprocesador 29
- Arquitecturas básicas de microprocesadores y microcontroladores. Harvard versus Von Neumann 39

Ficha 2. Introducción a los microcontroladores
- Controlador y microcontrolador 48
- Microprocesador y microcontrolador 49
- Aplicaciones de los microcontroladores 50
- ¿Qué microcontrolador emplear? 51
- Almacenamiento y ejecución del programa 55
- Recursos comunes a todos los microcontroladores 56
- Recursos especiales 62
- Herramientas para el desarrollo de aplicaciones 66
- Ejemplos de microcontroladores y aplicaciones 67

Ficha 3. Microcontroladores más utilizados
- Motorota 68HC908 (68HC908KX8) 79
- Intel 8051 (ATMEL AT89S8252) 83
- Microchip PIC 16F84 89

Ficha 4. Programación de microcontroladores
- Registros del microcontrolador 109
- Lenguaje assembler 112
- Estructura de un programa en assembler 122
- Desarrollo de un programa en assembler 128
- Archivo de código objeto 134

Ficha 5. Set de instrucciones
- Modos de direccionamiento 145
- Clasificación de las instrucciones 161
  1. Instrucciones de movimiento de datos 161
  2. Instrucciones aritméticas 164
  3. Instrucciones lógicas 168
  4. Instrucciones de manipulación de bits 169
  5. Instrucciones de manipulación de datos 169
  6. Instrucciones de control del programa 170
  7. Instrucciones de operaciones BCD 173
  8. Instrucciones especiales 173

Ficha 6. Procesando excepciones
- Reset e interrupciones 177
- Vinculación con el mundo exterior 182
- Volviendo a nuestro problema 184

Anexos
- Sistemas de numeración 199
- Representación de la información 201
- Set de instrucciones de la familia 68HC08 208
- Set de instrucciones PIC 16xxx 217
- Bibliografía 218

FICHA 3

Microcontroladores más utilizados

Para resolver aplicaciones sencillas se precisan pocos recursos; en cambio, las aplicaciones grandes requieren recursos numerosos y potentes. Siguiendo esta filosofía, las empresas que desarrollan microcontroladores construyen diversos modelos de microcontroladores orientados a cubrir, de forma óptima, las necesidades de cada proyecto. Así, hay disponibles microcontroladores sencillos y baratos para atender las aplicaciones simples, y otros complejos y más costosos para las de mucha envergadura.

En este capítulo vamos a plantearle una reseña de microcontroladores de prestaciones semejantes, de las empresas líderes en desarrollo de microcontroladores. Para esto, tomamos como ejemplo tres microcontroladores, típicos de cada familia, que permiten ver su potencial y compararlos significativamente:

MOTOROLA	MC68HC08	http://www.motorola.com; http:// www.mcu.motsps.com
intel	Intel 8051	http://www.intel.com http://www.atmel.com
MICROCHIP	PIC16F84	http://www.microchip.com

Motorola 68HC908 (68HC908KX8)

Para caracterizar este microcontrolador, abarcaremos:

- Arquitectura de ejecución.
- Descripción.
- Características.
- Mapa de memoria.
- Diagrama en bloques.
- Portfolio de la familia HC908.

El MC68HC908KX8 es un miembro de la familia de microcontroladores de 8 bit MCU M68HC08 de bajo costo y alta performance. Esta familia está basada en *customer-specified integrated circuit* –CSIC– *design strategy*.

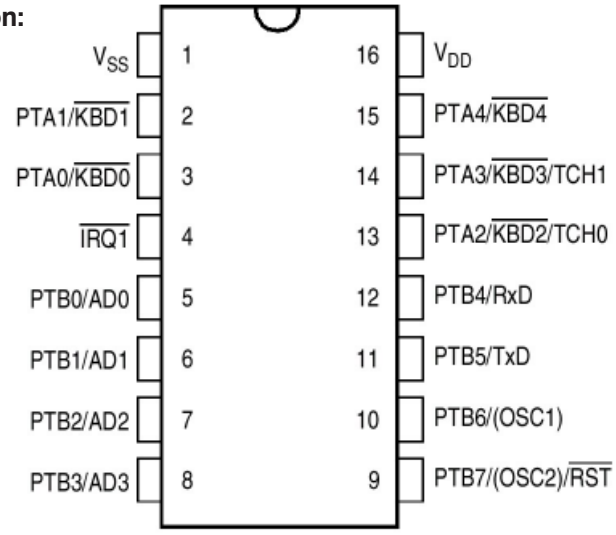
Todos los MCU de la familia usan la avanzada unidad central de procesos CPU08 y están disponibles con una variedad de módulos, tamaños de memoria y tipos de encapsulados.

- **Arquitectura de ejecución:**

El CPU 08 pertenece a la arquitectura del tipo Von Neumann clásica, característica de la familia 68xx de Motorola y ampliamente utilizada en el mundo.

En este tipo de arquitectura, existe un solo bus de datos, tanto para memoria de programas como para memoria de datos, lo que da origen a un mapa lineal de acceso a memoria; por consiguiente, no existen instrucciones especiales y diferentes para trabajar con datos o con código de programa. De esta forma, todas las instrucciones son aplicables en cualquier parte del mapa de memoria, sin importar si se trabaja con datos o código. Por esto, no es raro encontrar aplicaciones cuyos programas corren desde RAM como si estuvieran en Flash –lo que no podría hacer una arquitectura Harvard clásica–.

- **Descripción:**



- **Características:**

  - Código objeto compatible con M6805, M146805 y familia M68HC05.
  - Frecuencia interna de bus máxima:
    - 8 MHz at 5.0 V
    - 4 MHz at 3.0 V
  - Oscilador interno, no requiere componentes externos:
    - Frecuencias de bus selectables y programables por software.
    - Capacidad de ajuste por software (Oscilador interno ajustable al 2% de error).
    - Clock monitor.
    - Posibilidad de optar por fuente de clock externo o cristal externo o resonador externo Oscilador a Xtal hasta 32Mhz.
  - 8 Kbytes de memoria on-chip, in-circuit programmable FLASH.
  - Seguridad de programa en flash.
  - 192 bytes de RAM on-chip random-access memory –RAM–.
  - Un timer multifunción de 16-bit, 2-canales módulo timer interface –TIM– para Icap, Ocomp, PWM.
  - 4-canales, 8-bit, conversor analógico a digital –ADC–.
  - Módulo de comunicación serial asincrónica –SCI–.
  - 5-bit keyboard interrupt –KBI– Líneas de puerto para utilizarlas como teclado.
  - 13 líneas de entrada/salida de puertos de propósito general, con 15-mA source/ 15-mA sink y con pullups programable, cuatro entradas analógicas, dos para comunicación serial asincrónica.
  - Reset por baja tensión (LVI) programable p/ 3V y 5V.
  - Módulo de temporización timebase module –TBM– con:
    - Clock prescaler con 8 selecciones de usuario de interrupciones periódicas.
    - Fuente de clock activo en modo stop para...
  - Interrupción externa con pullup (IRQ1).
  - Sistema de protección:
    - Watchdog (COP) reset.
    - Detección de baja tensión con reset.
    - Detección de código ilegal con reset.
    - Detección de dirección ilegal con reset.
  - Internal power-up, circuito de reset sin pin exterior.

- **Mapa de memoria:**

El mapa de memoria, al igual que en el resto de la familia, es del tipo “lineal” sin saltos de página y de acceso continuo. O sea, el usuario puede disponer de la memoria sin

direccionamientos previos especiales. De esta forma, pueden ejecutarse programas desde RAM o desde Flash sin limitaciones de algún tipo.

Se observa que se mantiene la disposición típica de ubicar los registros de los puertos I/O y otros registros de uso general dentro de los primeros 256 bytes del mapa de memoria. También aquí se encuentra la zona de memoria RAM que, en algunos modelos, supera los 256 Bytes del mapa de memoria. (Aquellos usuarios de las familias HC05 y HC11 encontrarán esta disposición similar a la que usan habitualmente).

A continuación se observa una zona de espacio de memoria no-asignado (variable, según el modelo de HC908); seguido a éste se halla implementada la memoria de programa Flash.

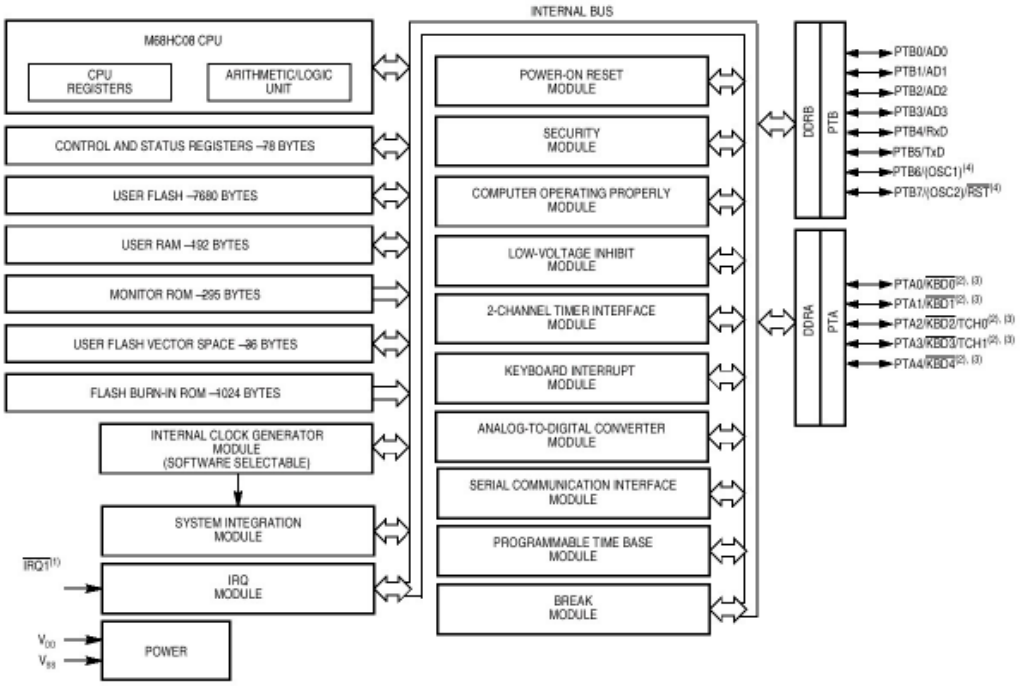
En la zona "final" del mapa de memoria, también se encuentran –como es costumbre en Motorola– todos los vectores de interrupciones, incluyendo el vector de reset y, además, los registros de distintos periféricos (SCI / SPI / TIMER / USB)

Memory Map

$0000 ↓ $003F	I/O REGISTERS (64 BYTES)
$0040 ↓ $00FF	RAM (192 BYTES)
$0100 ↓ $0FFF	UNIMPLEMENTED (3839 BYTES)
$1000 ↓ $13FF	FLASH BURN-IN ROM (1024 BYTES)
$1400 ↓ $DFFF	UNIMPLEMENTED (52,224 BYTES)
$E000 ↓ $FDFF	USER FLASH MEMORY (7680 BYTES)

$FE00	RESERVED
$FE01	SIM RESET STATUS REGISTER (SRSR)
$FE02	RESERVED
$FE03	RESERVED
$FE04	INTERRUPT STATUS REGISTER 1 (INT1)
$FE05	INTERRUPT STATUS REGISTER 2 (INT2)
$FE06	INTERRUPT STATUS REGISTER 3 (INT3)
$FE07	RESERVED
$FE08	FLASH CONTROL REGISTER (FLCR)
$FE09	BREAK ADDRESS REGISTER HIGH (BRKH)
$FE0A	BREAK ADDRESS REGISTER LOW (BRKL)
$FE0B	BREAK STATUS AND CONTROL REGISTER (BRKSCR)
$FE0C	LVI STATUS REGISTER (LVISR)
$FE0D ↓ $FE1F	UNIMPLEMENTED (18 BYTES)
$FE20 ↓ $FF46	MONITOR ROM (295 BYTES)
$FF47 ↓ $FF7D	UNIMPLEMENTED (57 BYTES)
$FF7E	FLASH BLOCK PROTECT REGISTER (FLBPR)
$FF7F ↓ $FFDB	UNIMPLEMENTED (90 BYTES)
$FFDC ↓ $FFFF	FLASH VECTORS (36 BYTES)

- **Diagrama en bloques:**



Notes:
1. Pin contains integrated pullup resistor
2. High-current source/sink pin
3. Pin contains software selectable pullup resistor if general function I/O pin is configured as input.
4. Pins are used for external clock source or crystal/ceramic resonator option.

- **Portfolio de la familia HC908:**

El portfolio de la familia HC908 de Motorola es amplio y cubre las principales necesidades en cuanto a costos y prestaciones.

Los derivativos de uso general son:

- MC68HC908GP32
- MC68HC908GT16
- MC68HC908GT8
- MC68HC908AB32
- MC68HC908GR8
- MC68HC908GR4
- MC68HC908JL3
- MC68HC908JL8
- MC68HC908JK3
- MC68HC908JK8
- MC68HC908JK1

Los derivativos de pequeño tamaño:

- MC68HC908KX2
- MC68HC908KX8
- MC68HC908RK2
- MC68HC908RF2
- MC68HC908QT1
- MC68HC908QT2
- MC68HC908QT4

- MC68HC908QY1
- MC68HC908QY2
- MC68HC908QY4

Los derivativos especiales para control industrial (PWM / ADC de 10 bits, módulo analógico, etc.) son:

- MC68HC908MR32
- MC68HC908MR16
- MC68HC908SR12

Los derivativos con USB –*Universal Serial Bus*– son:

- MC68HC908KH12
- MC68HC908JB8

Los derivativos con CAN –*Controller Area Network*– son:

- MC68HC908AZ60A
- MC68HC908AZ32
- MC68HC908GZ8

Intel 8051 (ATMEL AT89S8252)

Para realizar la comparación, utilizaremos un derivativo del 8051 de Intel que posee memoria flash y características semejantes a las presentadas en las otras CPU. Nos referimos a la CPU de ATMEL AT89S8252.

Serie 89 Microcontrolador Flash

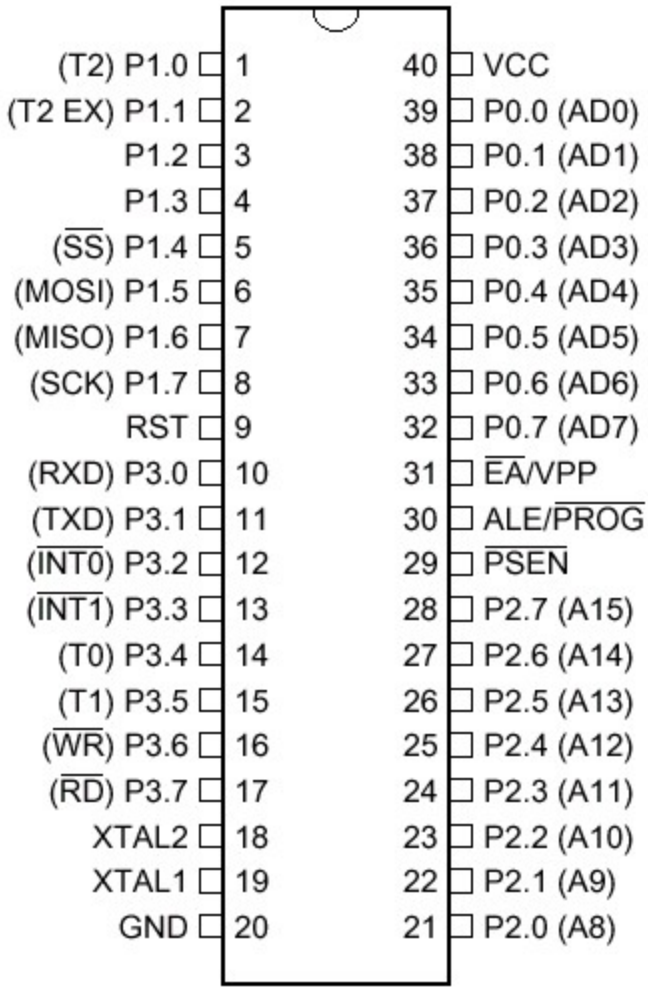
El AT89LS8252 es un mC de baja potencia, rango de voltaje medio, alta performance CMOS 8-bit microcomputador con 8K bytes de memoria Flash programable y borrable, y 2K bytes de EEPROM.

Este dispositivo es compatible con el estándar 80C51 set de instrucciones y pinout. El dispositivo puede ser reprogramado en el propio sistema mediante la interfaz SPI serial.

Para conocerlo más detalladamente, consideraremos:

- Descripción.
- Características.
- Diagrama en bloques.
- Set de instrucciones.
- Mapa de memoria.

- **Descripción:**



- **Características:**

  - Compatible con los productos MCS-51 ™.
  - Memoria de programa de 64 Kbytes.
  - Memoria de datos de 64 Kbytes.
  - 8K Bytes de memoria flash In-System Reprogramable Downloadable Flash Memory.
  - 2K Bytes EEPROM.
  - SPI Serial Interface para programación y descarga de programas.
  - Duración: 1,000 ciclos de escritura / borrado.
  - Rango de operación 2.7V a 6V.
  - Frecuencia de operación: 0 Hz to 12 MHz (algunas versiones a 16 Mhz).
  - 256 x 8 bit de RAM interna.
  - 32 líneas programables de I/O.
  - 3 timers / contadores de 16-bit .
  - 9 fuentes de interrupción.
  - Programable UART. Comunicación serial asincrónica.
  - SPI Serial Interface. Comunicación serial sincrónica.
  - Modos de bajo consumo.
  - Recuperación por interrupción de modo de bajo consumo.
  - Programable Watchdog Timer.
  - Doble puntero de datos.

- **Diagrama en bloques:**

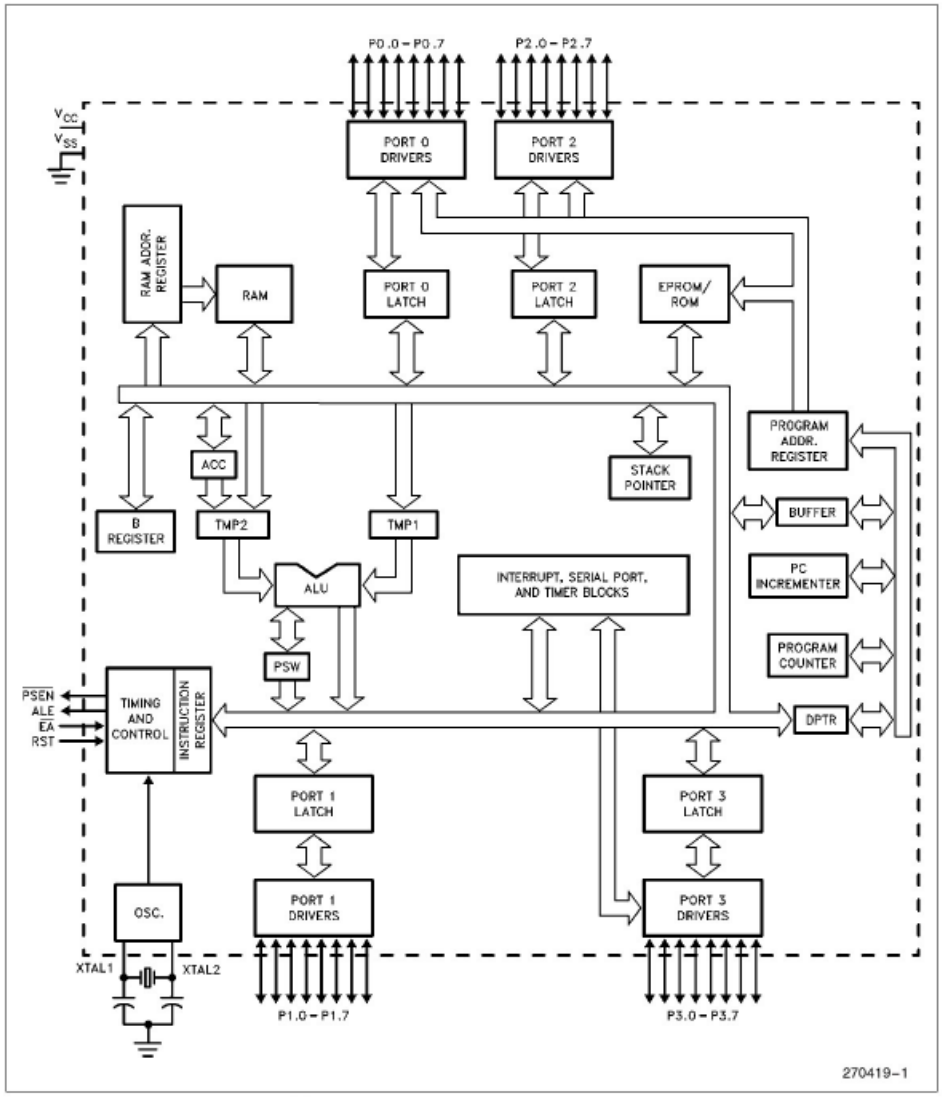


Figure 1. MCS® 51 Microcontroller Architectural Block Diagram

- **Set de instrucciones:**

MNEMÓNICO	DESCRIPCIÓN	BYTE	CICLOS DE CLOCK
	ARITHMETIC OPERATIONS		
ADD A,Rn	Add register to Accumulator	1	12
ADD A,direct	Add direct byte to Accumulator	2	12
ADD A,@Ri	Add indirect RAM to Accumulator	1	12
ADD A,#data	Add immediate data to Accumulator	2	12
ADDC A,Rn	Add register to Accumulator with carry	1	12
ADDC A,direct	Add direct byte to Accumulator with carry	2	12
ADDC A,@Ri	Add indirect RAM to Accumulator with carry	1	12
ADDC A,#data	Add immediate data to ACC with carry	2	12
SUBB A,Rn	Subtract Register from ACC with borrow	1	12
SUBB A,direct	Subtract direct byte from ACC with borrow	2	12
SUBB A,@Ri	Subtract indirect RAM from ACC with borrow	1	12
SUBB A,#data	Subtract immediate data from ACC with borrow	2	12
INC A	Increment Accumulator	1	12
INC Rn	Increment register	1	12
INC direct	Increment direct byte	2	12
INC @Ri	Increment indirect RAM	1	12
DEC A	Decrement Accumulator	1	12

MNEMÓNICO	DESCRIPCIÓN	BYTE	CICLOS DE CLOCK
DEC Rn	Decrement Register	1	12
DEC direct	Decrement direct byte	2	12
DEC @Ri	Decrement indirect RAM	1	12
INC DPTR	Increment Data Pointer	1	24
MUL AB	Multiply A and B	1	48
DIV AB	Divide A by B	1	48
DA A	Decimal Adjust Accumulator	1	12
	LOGICAL OPERATIONS		
ANL A,Rn	AND Register to Accumulator	1	12
ANL A,direct	AND direct byte to Accumulator	2	12
ANL A,@Ri	AND indirect RAM to Accumulator	1	12
ANL A,#data	AND immediate data to Accumulator	2	12
ANL direct,A	AND Accumulator to direct byte	2	12
ANL direct,#data	AND immediate data to direct byte	3	24
ORL A,Rn	OR register to Accumulator	1	12
ORL A,direct	OR direct byte to Accumulator	2	12
ORL A,@Ri	OR indirect RAM to Accumulator	1	12
ORL A,#data	OR immediate data to Accumulator	2	12
ORL direct,A	OR Accumulator to direct byte	2	12
ORL direct,#data	OR immediate data to direct byte	3	24
XRL A,Rn	Exclusive-OR register to Accumulator	1	12
XRL A,direct	Exclusive-OR direct byte to Accumulator	2	12
XRL A,@Ri	Exclusive-OR indirect RAM to Accumulator	1	12
XRL A,#data	Exclusive-OR immediate data to Accumulator	2	12
XRL direct,A	Exclusive-OR Accumulator to direct byte	2	12
XRL direct,#data	Exclusive-OR immediate data to direct byte	3	24
CLR A	Clear Accumulator	1	12
CPL A	Complement Accumulator	1	12
RL A	Rotate Accumulator left	1	12
RLC A	Rotate Accumulator left through the carry	1	12
RR A	Rotate Accumulator right	1	12
RRC A	Rotate Accumulator right through the carry	1	12
SWAP A	Swap nibbles within the Accumulator	1	12
	DATA TRANSFER		
MOV A,Rn	Move register to Accumulator	1	12
MOV A,direct	Move direct byte to Accumulator	2	12
MOV A,@Ri	Move indirect RAM to Accumulator	1	12
MOV A,#data	Move immediate data to Accumulator	2	12
MOV Rn,A	Move Accumulator to register	1	12
MOV Rn,direct	Move direct byte to register	2	24
MOV Rn,#data	Move immediate data to register	2	12
MOV direct,A	Move Accumulator to direct byte	2	12
MOV direct,Rn	Move register to direct byte	2	24
MOV direct,direct	Move direct byte to direct	3	24
MOV direct,@Ri	Move indirect RAM to direct byte	2	24
MOV direct,#data	Move immediate data to direct byte	3	24
MOV @Ri,A	Move Accumulator to indirect RAM	1	12
MOV @Ri,direct	Move direct byte to indirect RAM	2	24
MOV @Ri,#data	Move immediate data to indirect RAM	2	12
MOV DPTR,#data16	Load Data Pointer with a 16-bit constant	3	24
MOVC A,@A+DPTR	Move Code byte relative to DPTR to A CC	1	24
MOVC A,@A+PC	Move Code byte relative to PC to A CC	1	24
MOVX A,@Ri	Move external RAM (8-bit addr) to A CC	1	24

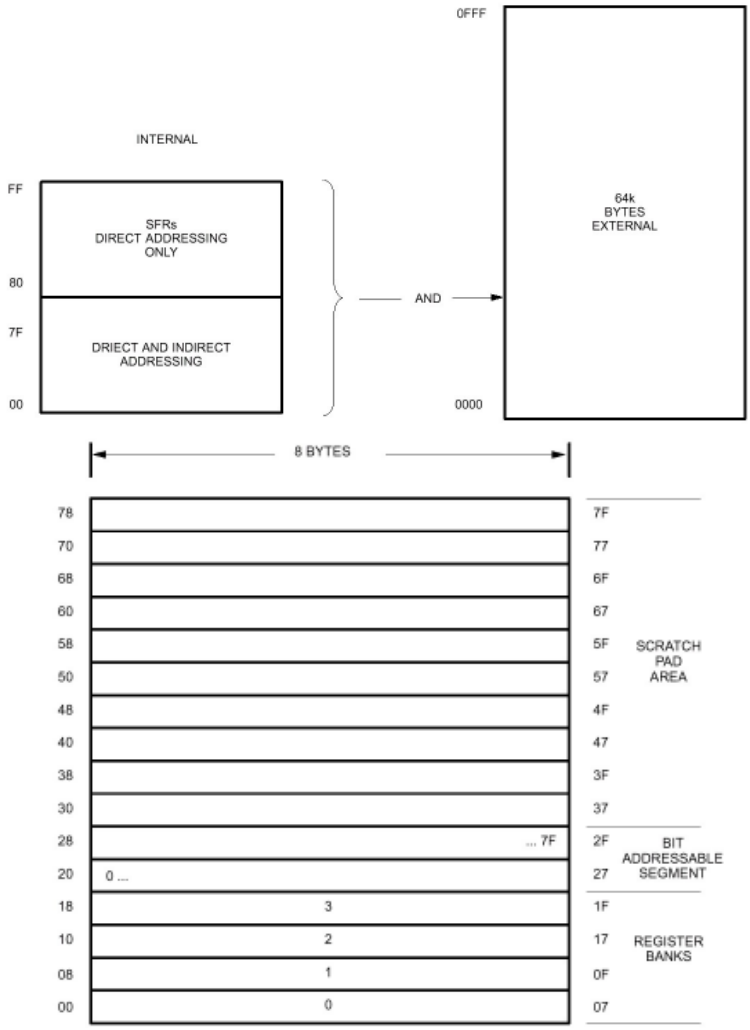
MNEMÓNICO	DESCRIPCIÓN	BYTE	CICLOS DE CLOCK
MOVX A,@DPTR	Move external RAM (16-bit addr) to A CC	1	24
MOVX A,@Ri,A	Move A CC to external RAM (8-bit addr)	1	24
MOVX @DPTR,A	Move A CC to external RAM (16-bit addr)	1	24
PUSH direct	Push direct byte onto stack	2	24
POP direct	Pop direct byte from stack	2	24
XCH A,Rn	Exchange register with Accumulator	1	12
XCH A,direct	Exchange direct byte with Accumulator	2	12
XCH A,@Ri	Exchange indirect RAM with Accumulator	1	12
XCHD A,@Ri	Exchange low-order digit indirect RAM with A CC	1	12
	BOOLEAN VARIABLE MANIPULATION		
CLR C	Clear carry	1	12
CLR bit	Clear direct bit	2	12
SETB C	Set carry	1	12
SETB bit	Set direct bit	2	12
CPL C	Complement carry	1	12
CPL bit	Complement direct bit	2	12
ANL C,bit	AND direct bit to carry	2	24
ANL C,/bit	AND complement of direct bit to carry	2	24
ORL C,bit	OR direct bit to carry	2	24
ORL C,/bit	OR complement of direct bit to carry	2	24
MOV C,bit	Move direct bit to carry	2	12
MOV bit,C	Move carry to direct bit	2	24
JC rel	Jump if carry is set	2	24
JNC rel	Jump if carry not set	2	24
JB rel	Jump if direct bit is set	3	24
JNB rel	Jump if direct bit is not set	3	24
JBC bit,rel	Jump if direct bit is set and clear bit	3	24
	PROGRAM BRANCHING		
ACALL addr11	Absolute subroutine call	2	24
LCALL addr16	Long subroutine call	3	24
RET	Return from subroutine	1	24
RETI	Return from interrupt	1	24
AJMP addr11	Absolute jump	2	24
LJMP addr16	Long jump	3	24
SJMP rel	Short jump (relative addr)	2	24
JMP @A+DPTR	Jump indirect relative to the DPTR	1	24
JZ rel	Jump if Accumulator is zero	2	24
JNZ rel	Jump if Accumulator is not zero	2	24
CJNE A,direct,rel	Compare direct byte to A CC and jump if not equal	3	24
CJNE A,#data,rel	Compare immediate to A CC and jump if not equal	3	24
CJNE RN,#data,rel	Compare immediate to register and jump if not equal	3	24
CJNE @Ri,#data,rel	Compare immediate to indirect and jump if not equal	3	24
DJNZ Rn,rel	Decrement register and jump if not zero	2	24
DJNZ direct,rel	Decrement direct byte and jump if not zero	3	24
NOP	No operation	1	12

Referencias:

Rn: Register R7-R0 of the currently selected Register Bank.
direct: 8-bit internal data location's address. This could be an Internal Data RAM location (0-127) or a SFR [i.e., I/O port, control register, status register, etc. (128-255)].

@Ri: 8-bit internal data RAM location (0-255) addressed indirectly through register R1 or R0.
#data: 8-bit constant included in the instruction.
#data 16: 16-bit constant included in the instruction.
addr 16: 16-bit destination address. Used by LCALL and LJMP. A branch can be anywhere within the 64k-byte Program Memory address space.
addr 11: 11-bit destination address. Used by ACALL and AJMP. The branch will be within the same 2k-byte page of program memory as the first byte of the following instruction.
rel: Signed (two's complement) 8-bit offset byte. Used by SJMP and all conditional jumps. Range is –128 to +127 bytes relative to first byte of the following instruction.
bit: Direct Addressed bit in Internal Data RAM or Special Function Register.

- **Mapa de memoria:**



PSW: PROGRAM STATUS WORD BIT ADDRESSABLE

CY	AC	F0	RS1	RS0	OV	—	P

CY	PSW.7	Carry Flag
AC	PSW.6	Auxiliary Cary Flag
F0	PSW.5	Flag 0 available to the user for general purpose
RS1	PSW.4	Register Bank selector bit 1 (SEE NOTE 1)
RS0	PSW.3	Register Bank selector bit 0 (SEE NOTE 1)
CV	PSW.2	Overflow Flag

—	PSW.1	Usable as a general purpose Flag
P	PSW.0	Parity Flag. Set/Cleaned by hardware each instruction cycle to indicate an odd/even number of '1' bus in the accumulator

Note:
The value presented by RS0 and RS1 selects the corresponding register bank.

RS1	RS0	REGISTER BANK	ADDRESS
0	0	0	00H-07H
0	1	1	08H-0Fh
1	0	2	10H-17H
1	1	3	18H-1FH

Microchip PIC16F84

El procesador responde a la arquitectura RISC, que se identifica porque el juego de instrucciones se reduce a 35, la mayoría de las cuales se ejecuta en un solo ciclo de reloj, excepto las instrucciones de salto que necesitan dos ciclos.

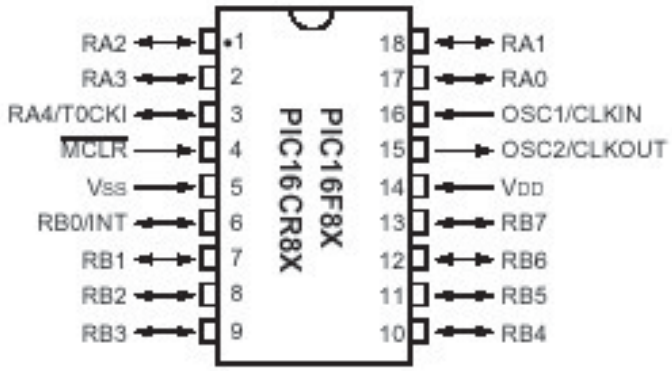
Vamos a detenernos en:

- Características.
- Descripción.
- Gamas de PIC.
- Repertorio de instrucciones.
- Diagrama en bloques.
- Memoria de programa.
- Memoria de datos.
- Mapa de memoria.
- Puertas de entrada y salida.
- Conexionado típico.

- **Características:**

  - Parte de la memoria de datos es de tipo EEPROM (64 registros de 8 bits).
  - Memoria de programa (1024 registros de 14 bits) de tipo flash, de iguales prestaciones que la EEPROM, pero con mejor rendimiento.
  - 2 temporizadores: TMR0 y *watchdo*g. El TMR0 puede actuar como temporizador o como contador.
  - 4 posibles fuentes de interrupción que pueden habilitarse o deshabilitarse por software.
  - Reinicialización del sistema o *RESET* por cinco causas distintas.
  - Estado de funcionamiento en bajo consumo o *Slee*p, con un consumo de 40 mA.
  - Frecuencia de trabajo máxima puede ser de 10 MHz.

- **Descripción:**



- **Arquitectura.** La arquitectura del procesador sigue el modelo Harvard. En esta arquitectura, la CPU se conecta de forma independiente, y con buses distintos con la memoria de instrucciones y con la de datos.
- **Segmentación.** Se aplica la técnica de segmentación *pipe-line* en la ejecución de las instrucciones. La segmentación permite al procesador realizar, al mismo tiempo, la ejecución de una instrucción y la búsqueda del código de la siguiente. De esta forma, se puede ejecutar cada instrucción en un ciclo (un ciclo de instrucción equivale a cuatro ciclos de reloj).
- **Formato de las instrucciones.** El formato de todas las instrucciones es de la misma longitud. Todas las instrucciones de los microcontroladores de la gama baja tienen una longitud de 12 bits. Las de la gama media tienen 14 bits y más las de la gama alta. Esta característica es muy ventajosa en la optimización de la memoria de instrucciones, y facilita enormemente la construcción de ensambladores y compiladores.
- **Juego de instrucciones.** Se trata de un procesador RISC –Computador de juego de instrucciones reducido–. Los modelos de la gama baja disponen de un repertorio de 33 instrucciones, 35 los de la gama media y casi 60 los de la alta. Todas las instrucciones son ortogonales. Cualquier instrucción puede manejar cualquier elemento de la arquitectura como fuente o como destino.

La arquitectura está basada en un "banco de registros"; esto significa que todos los objetos del sistema (puertas de E/S, temporizadores, posiciones de memoria, etc.) están implementados físicamente como registros.

Existe diversidad de modelos de microcontroladores, con prestaciones y recursos diferentes. La gran variedad de modelos de microcontroladores PIC permite que el usuario pueda seleccionar el más conveniente para su proyecto.

Las herramientas de soporte son potentes y económicas. La empresa Microchip –y otras que utilizan los PIC– ponen a disposición de los usuarios numerosas herramientas para desarrollar hardware y software. Son muy abundantes los programadores, los simuladores software, los emuladores en tiempo real, ensambladores, compiladores C, intérpretes y compiladores BASIC, etc.

- **Gamas de PIC:**

Una de las labores más importantes del ingeniero de diseño es la elección del microcontrolador que mejor satisfaga las necesidades del proyecto con el mínimo presupuesto.

Microchip dispone de varias familias de microcontroladores de 8 bits para adaptarse a las necesidades de la mayoría de los clientes potenciales:

- La gama enana: PIC12C(F)XXX de 8 patitas
- La gama baja o básica: PIC16C5X con instrucciones de 12 bits
- La gama media: PIC16CXXX con instrucciones de 14 bits
- La gama alta: PIC17CXXX con instrucciones de 16 bits
- La gama "mejorada": PIC18CXXX, (palabra de programa de 16 bits)

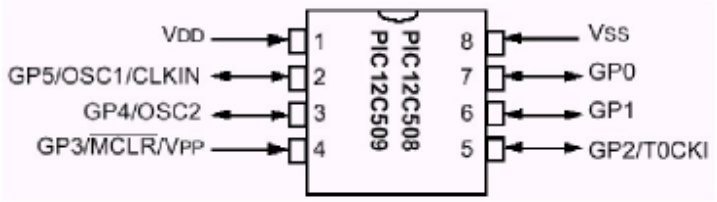
Gama enana. PIC12C(F)XXX de 8 patitas:

Se trata de un grupo de PIC que ha acaparado la atención del mercado. Su principal característica es su reducido tamaño, al disponer todos sus componentes de 8 patitas.

Se alimenta con un voltaje de corriente continua comprendido entre 2,5 V y 5,5 V, y consume menos de 2 mA cuando trabaja a 5 V y 4 MHz.

El formato de sus instrucciones puede ser de 12 o de 14 bits y su repertorio es de 33 o 35 instrucciones, respectivamente.

El diagrama de conexionado de uno de estos PIC es:



Aunque los PIC enanos sólo tienen 8 patitas, pueden destinar hasta 6 como líneas de E/S para los periféricos, porque disponen de un oscilador interno R-C.

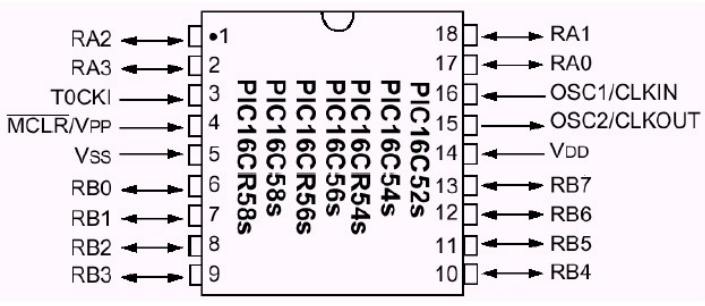
Son muy apreciados en las aplicaciones de control de personal, en sistemas de seguridad y en dispositivos de bajo consumo que gestionan receptores y transmisores de señales. Su pequeño tamaño los hace ideales en muchos proyectos donde esta cualidad es fundamental.

Gama baja o básica. PIC16C5X con instrucciones de 12 bits:

Se trata de una serie de PIC de recursos limitados, pero con una de la mejores relaciones coste/prestaciones.

Sus versiones están encapsuladas con 18 y 28 patitas, y pueden alimentarse a partir de una tensión de 2,5 V, lo que las hace ideales en las aplicaciones que funcionan con pilas, teniendo en cuenta su bajo consumo (menos de 2 mA a 5 V y 4 MHz).

Tiene un repertorio de 33 instrucciones, cuyo formato consta de 12 bits. No admite ningún tipo de interrupción y la pila sólo dispone de dos niveles. Su diagrama de conexionado:



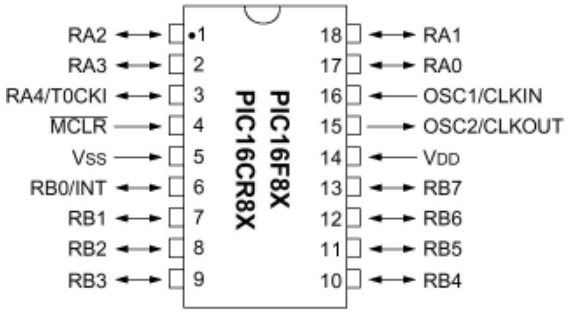
1. Sistema POR –*Power On Reset*–
2. Perro guardián –Watchdog o WDT–
3. Código de protección
4. Líneas de E/S de alta corriente
5. Modo de reposo (Bajo consumo o *sleep*)

Conviene considerar dos restricciones importantes:

- La pila o *stack* sólo dispone de dos niveles, lo que supone no poder encadenar más de dos subrutinas.
- Los microcontroladores de la gama baja no admiten interrupciones.

Gama media. PIC16CXXX con instrucciones de 14 bits:

Es la gama más variada y completa de los PIC. Abarca modelos con encapsulado desde 18 patitas hasta 68, cubriendo varias opciones que integran abundantes periféricos. Dentro de esta gama se halla el PIC16X84 y sus variantes. Su diagrama de conexionado:



En esta gama, sus componentes añaden nuevas prestaciones a las que poseían los de la gama baja, haciéndolos más adecuados en las aplicaciones complejas. Admiten interrupciones, poseen comparadores de magnitudes analógicas, convertidores A/D, puertos serie y diversos temporizadores. El repertorio de instrucciones es de 35, de 14 bits cada una y compatible con el de la gama baja. Sus distintos modelos contienen todos los recursos que se precisan en las aplicaciones de los microcontroladores de 8 bits. También dispone de interrupciones y una pila de 8 niveles que permite el anidamiento de subrutinas.

Gama alta. PIC17CXXX con instrucciones de 16 bits:

Se alcanzan las 58 instrucciones de 16 bits en el repertorio. Sus modelos disponen de un sistema de gestión de interrupciones vectorizadas muy potente. También incluyen variados controladores de periféricos, puertas de comunicación serie y paralelo con elementos externos, un multiplicador hardware de gran velocidad y mayores capacidades de memoria –que alcanza los 8 k palabras en la memoria de instrucciones y 454 bytes en la memoria de datos–.

Quizás la característica más destacable de los componentes de esta gama es su arquitectura abierta, que consiste en la posibilidad de ampliación del microcontrolador con elementos externos. Para este fin, las patitas sacan al exterior las líneas de los buses de datos, direcciones y control, a las que se conectan memorias o controladores de periféricos. Esta facultad obliga a estos componentes a tener un elevado número de patitas, comprendido entre 40 y 44. Esta filosofía de construcción del sistema es la que se empleaba en los microprocesadores y no suele ser una práctica habitual cuando se trata de microcontroladores, que sólo se utilizan en aplicaciones muy especiales con grandes requerimientos.

Gama "mejorada". PIC18CXXX, (palabra de programa de 16 bits):

El conjunto de instrucciones se halla mejorado.

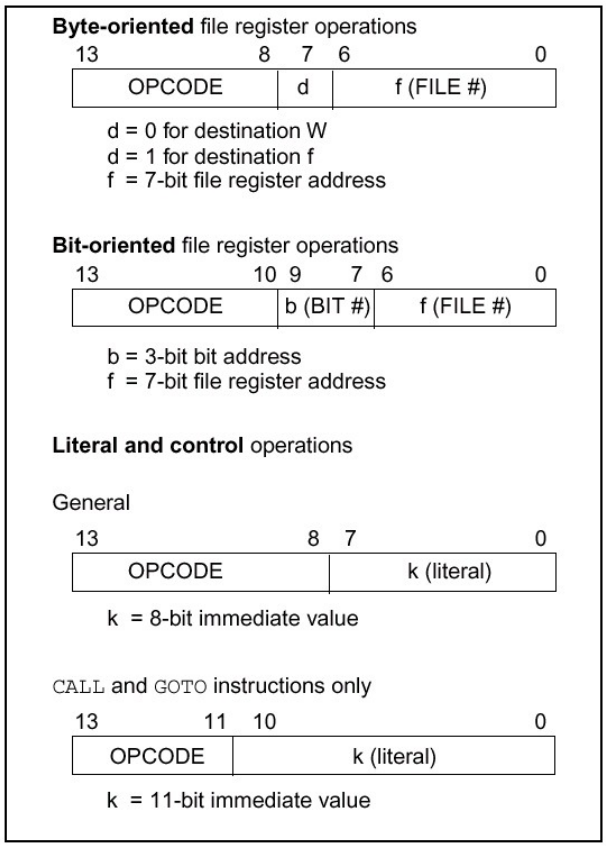
Incluye detección de bajo voltaje programable (PLVD).

PIC18CXXX	8-Bit, arquitectura mejorada.
PIC18F0XX	8-Pin, 8-Bit arquitectura mejorada FLASH con EEPROM, PLVD, BOR y PWM.
PIC18FXX2	Protección de código, 256 EEPROM de datos, Detección de bajo voltaje programable (LVD), Phase-locked Loop (PLL), modo SLEEP, multiplicador 8x8 , PSP, In-Circuit Debugging
PIC18FX32	28/40-Pin FLASH alto rendimiento, modo de bajo consumo, EEPROM dede 256 datos, PLVD y A/D 10-bit.
PIC18FXX31	28/40-Pin FLASH con EEPROM, PLVD, PBOR, A/D 10-bit y módulo PWM.
PIC18FXX5	28/40-Pin FLASH con USB y A/D 10-bit.
PIC18FXX30	18/20-pin FLASH con EEPROM, PBOR, A/D 10-bit y módulo PWM 3-fase.

- **Repertorio de instrucciones:**

ABREVIATURA	DESCRIPCIÓN
PC	Contador de programa que direcciona la memoria de instrucciones. Tiene un tamaño de 11 bits en la gama baja, de los cuales los 8 de menos peso configuran el registro PCL que ocupa el registro 0x02 del área de datos.
TOS	Cima de la pila, con 2 niveles en la gama baja y 8 en la media.
WDT	Perro guardián –*Watchdog*–.
W	Registro W, similar al acumulador.
F	Suele ser un campo de 5 bits (fffff) que contiene la dirección del banco de registros que ocupa el banco 0 del área de datos.Direcciona uno de esos registros.
D	Bit del código OP de la instrucción que selecciona el destino. Si d=0, el destino es W; si d=1, el destino es f.
Dest	Destino (registro w o f).
TO	Bite *Time Out* del registro de estado.
PD	Bit *Power Down* del registro de estado.
b	Suele ser un campo de 3 bits (bbb) que determinan la posición de un bit dentro de un registro de 8 bit.
k	Se trata, normalmente, de un campo de 8 bits (kkkkkkkk) que representa un dato inmediato. También puede constar de 9 bits en las instrucciones de salto que cargan al PC.
x	Valor indeterminado (puede ser un 0 o un 1). Para mantener la compatibilidad con las herramientas software de Microchip conviene hacer x=0.
label	Nombre de la etiqueta.
[ ]	Opciones.
( )	Contenido.
⇒	Se asigna a.
< >	Campo de bits de un registro.
∈	Pertenece al conjunto.
Z	Señalizador de cero en W. Pertenece al registro de estado.
C	Señalizador de acarreo en el octavo bit del W. Pertenece al registro de estado.
DC	Señalizador de acarreo en el 4 bit del W. Pertenece al registro de estado.
Itálicas	Términos definidos por el usuario.

El formato general de las instrucciones es:



Veamos el repertorio de instrucciones de la gama media:

ADDLW Suma un literal

Sintaxis: [label] ADDLW k
Operandos: $0 \leq k \leq 255$
Operación: (W) + (k) $\Rightarrow$ (W)
Flags afectados: C, DC, Z
Código OP: 11 111x kkkk kkkk
Descripción: Suma el contenido del registro W y k, guardando el resultado en W.
Ejemplo: ADDLW 0xC2
Antes: W = 0x17
Después: W = 0xD9

ANDLW W AND literal

Sintaxis: [label] ANDLW k
Operandos: $0 \leq k \leq 255$
Operación: (W) AND (k) $\Rightarrow$ (W)
Flags afectados: Z
Código OP: 11 1001 kkkk kkkk
Descripción: Realiza la operación lógica AND entre el contenido del registro W y k, guardando el resultado en W.
Ejemplo: ANDLW 0xC2
Antes: W = 0x17
Después: W = 0x02

ADDWF W + F

Sintaxis: [label] ADDWF f,d
Operandos: d $\in$ [0,1], $0 \leq f \leq 127$
Operación: (W) + (f) $\Rightarrow$ (dest)
Flags afectados: C, DC, Z
Código OP: 00 0111 dfff ffff
Descripción: Suma el contenido del registro W y el registro f. Si d es 0, el resultado se almacena en W; si d es 1, se almacena en f.
Ejemplo: ADDWF REG,0
Antes: W = 0x17., REG = 0xC2
Después: W = 0xD9, REG = 0xC2

BTFSS Test de bit y salto

Sintaxis: [label] BTFSS f,b
Operandos: , $0 \leq b \leq 7$, $0 \leq f \leq 127$
Operación: Salto si (f<b>) = 1
Flags afectados: Ninguno
Código OP: 01 11bb bfff ffff
Descripción: Si el bit b del registro f es 1, se salta una instrucción y se continúa con la ejecución. En caso de salto, ocupará dos ciclos de reloj.
Ejemplo: BTFSS REG,6
GOTO SI_ES_0
NO_ES_0 Instrucción
SI_ES_0 Instrucción

ANDWF W AND F

Sintaxis: [label] ANDWF f,d
Operandos: d $\in$ [0,1], $0 \leq f \leq 127$
Operación: (W) AND (f) $\Rightarrow$ (dest)
Flags afectados: Z
Código OP: 00 0101 dfff ffff
Descripción: Realiza la operación lógica AND entre los registros W y f.Si d es 0, el resultado se almacena en W, si d es 1 se almacena en f.
Ejemplo: ANDWF REG,1
Antes: W = 0x17., REG = 0xC2
Después: W = 0x17, REG = 0x02

BCF Borra un bit

Sintaxis: [label] BCF f,b
Operandos: $0 \leq f \leq 127$, $0 \leq b \leq 7$
Operación: 0 $\Rightarrow$ (f<b>)
Flags afectados: Ninguno
Código OP: 01 00bb bfff ffff
Descripción: Borra el bit b del registro f.
Ejemplo: BCF REG,7
Antes: REG = 0xC7
Después: REG = 0x47

BTFSC Test de bit y salto

Sintaxis: [label] BTFSC f,b
Operandos: $0 \leq b \leq 7$, $0 \leq f \leq 127$
Operación: Salto si (f<b>) = 0
Flags afectados: Ninguno
Código OP: 01 10bb bfff ffff
Descripción: Si el bit b del registro f es 0, se salta una instrucción y se continúa con la ejecución. En caso de salto, ocupará dos ciclos de reloj.
Ejemplo: BTFSC REG,6
GOTO NO_ES_0
SI_ES_0 Instrucción
NO_ES_0 Instrucción

BSF Activa un bit

Sintaxis: [label] BSF f,b
Operandos: $0 \leq f \leq 127$, $0 \leq b \leq 7$
Operación: $1 \Rightarrow$ (f<b>)
Flags afectados: Ninguno
Código OP: 01 01bb bfff ffff
Descripción: Activa el bit b del registro f
Ejemplo: BSF REG,7
Antes: REG = 0x0A
Después: REG = 0x8A

CALL Salto a subrutina

Sintaxis: [label] CALL k
Operandos: $0 \leq k \leq 2047$
Operación: PC $\Rightarrow$ Pila; k $\Rightarrow$PC
Flags afectados: Ninguno
Código OP: 10 0kkk kkkk kkkk
Descripción: Salto a una subrutina. La parte baja de k se carga en PCL, y la alta en PCLATCH. Ocupa 2 ciclos de reloj.
Ejemplo: ORIGEN CALL DESTINO
Antes: PC = ORIGEN
Después: PC = DESTINO.

CLRF Borra un registro

Sintaxis: [label] CLRF f
Operandos: $0 \leq f \leq 127$
Operación: 0x00 $\Rightarrow$ (f), $1 \Rightarrow$ Z
Flags afectados: Z
Código OP: 00 0001 1fff ffff
Descripción: El registro f se carga con 0x00. El flag Z se activa.
Ejemplo: CLRF REG
Antes: REG = 0x5A
Después: REG = 0x00, Z = 1

CLRW Borra el registro W

Sintaxis: [label] CLRW
Operandos: Ninguno
Operación: 0x00 ⇒ W, 1 ⇒ Z
Flags afectados: Z
Código OP: 00 0001 0xxx xxxx
Descripción: El registro de trabajo W se carga con 0x00. El flag Z se activa.
Ejemplo: CLRW
Antes: W = 0x5A
Después: W = 0x00, Z = 1

CLRWDT Borra el WDT

Sintaxis: [label] CLRWDT
Operandos: Ninguno
Operación: 0x00 ⇒ WDT, 1 ⇒/TO, 1 ⇒/PD
Flags afectados: /TO, /PD
Código OP: 00 0000 0110 0100
Descripción: Esta instrucción borra tanto el WDT como su preescaler. Los bits /TO y /PD del registro de estado se ponen a 1.
Ejemplo: CLRWDT
Después: Contador WDT = 0,
Preescales WDT = 0,
/TO = 1, /PD = 1

INCFSZ Incremento y salto

Sintaxis: [label] INCFSZ f,d
Operandos: d ∈ [0,1], 0 ≤ f ≤ 127
Operación: $_{(f) + 1 \rightarrow (destination)}$, ; Salto si Result = 0
Flags afectados: Ninguno
Código OP: 00 1111 dfff ffff
Descripción: Incrementa el contenido del registro f. Si d es 0, el resultado se almacena en W; si d es 1, se almacena en f. Si la resta es 0, salta la siguiente instrucción, en cuyo caso costaría 2 ciclos.
Ejemplo: INCFSC REG,0
GOTO NO_ES_0
SI_ES_0 Instrucción
NO_ES_0 Salta instrucción anterior

COMF Complemento de f

Sintaxis: [label] COMF f,d
Operandos: d ∈ [0,1], 0 ≤ f ≤ 127
Operación $_{(\text{complemento de f})} \rightarrow$ (destination)
Flags afectados: Z
Código OP: 00 1001 dfff ffff
Descripción: El registro f es complementado. El flag Z se activa si el resultado es 0. Si d es 0, el resultado se almacena en W; si d es 1, se almacena en f.
Ejemplo: COMF REG,0
Antes: REG = 0x13
Después: REG = 0x13, W = 0XEC

INCF Incremento de f

Sintaxis: [label] INCF f,d
Operandos: d ∈ [0,1], 0 ≤ f ≤ 127
Operación: (f) + 1 ⇒ (dest)
Flags afectados: Z
Código OP: 00 1010 dfff ffff
Descripción: Incrementa en 1 el contenido de f. Si d es 0, el resultado se almacena en W; si d es 1, se almacena en f.
Ejemplo: INCF CONT,1
Antes: CONT = 0xFF, Z = 0
Después: CONT = 0x00, Z = 1

GOTO Salto incondicional

Sintaxis: [label] GOTO k
Operandos: 0 ≤ k ≤ 2047
Operación: k ⇒ PC <10:0>
Flags afectados: Ninguno
Código OP: 10 1kkk kkkk kkkk
Descripción: Se trata de un salto incondicional. La parte baja de k se carga en PCL, y la alta en PCLATCH. Ocupa 2 ciclos de reloj.
Ejemplo: ORIGEN GOTO DESTINO
Antes: PC = ORIGEN
Después: PC = DESTINO

DECFSZ Decremento y salto

Sintaxis: [label] DECFSZ f,d
Operandos: d ∈ [0,1], 0 ≤ f ≤ 127
Operación: $_{(f) - 1 \rightarrow (destination)}$; Salto si Result = 0
Flags afectados: Ninguno
Código OP: 00 1011 dfff ffff
Descripción: Decrementa el contenido del registro f. Si d es 0, el resultado se almacena en W; si d es 1, se almacena en f. Si la resta es 0 salta la siguiente instrucción, en cuyo caso costaría 2 ciclos.
Ejemplo: DECFSC REG,0
GOTO NO_ES_0
SI_ES_0 Instrucción
NO_ES_0 Salta instrucción anterior

DECF Decremento de f

Sintaxis: [label] DECF f,d
Operandos: d ∈ [0,1], 0 ≤ f ≤ 127
Operación: (f) – 1 ⇒ (dest)
Flags afectados: Z
Código OP: 00 0011 dfff ffff
espuésión: Decrementa en 1 el contenido de f. Si d es 0, el resultado se almacena en W; si d es 1, se almacena en f.
Ejemplo: DECF CONT,1
Antes: CONT = 0x01, Z = 0
Después: CONT = 0x00, Z = 1.

IORWF W AND F

Sintaxis: [label] IORWF f,d
Operandos: d ∈ [0,1], 0 ≤ f ≤ 127
Operación: (W) OR (f) ⇒ (dest)
Flags afectados: Z
Código OP: 00 0100 dfff ffff
Después: Realiza la operación lógica OR entre los registros W y f. Si d es 0, el resultado se almacena en W; si d es 1, se almacena en f.
Ejemplo: IORWF REG,0
Antes: W = 0x91, REG = 0x13
Después: W = 0x93, REG = 0x13

MOVLW Cargar literal en W

Sintaxis: [label] MOVLW f
Operandos: 0 ≤ f ≤ 255
Operación: (k) ⇒ (W)
Flags afectados: Ninguno
Código OP: 11 00xx kkkk kkkk
Descripción: El literal k pasa al registro W.
Ejemplo: MOVLW 0x5A
Después: REG = 0x4F, W = 0x5A

IORLW W OR literal

Sintaxis :[label] IORLW k
Operandos: 0 ≤ k ≤ 255
Operación : (W) OR (k) ⇒ (W)
Flags afectados: Z
Código OP: 11 1000 kkkk kkkk
Descripción: Se realiza la operación lógica OR entre el contenido delregistro W y k, guardando el resultado en W.
Ejemplo: IORLW 0x35
Antes: W = 0x9A
Después: W = 0xBF

RETFIE Retorno de interrupción

Sintaxis: [label] RETFIE
Operandos: Ninguno
Operación: 1 ⇒ GIE; TOS ⇒ PC
Flags afectados: Ninguno
Código OP: 00 0000 0000 1001
Descripción: El PC se carga con el contenido de la cima de la pila(TOS): dirección de retorno. Consume 2 ciclos. Las interrupciones vuelven a ser habilitadas.(GIE)
Ejemplo: RETFIE
Después: PC = dirección de retornoGIE = 1

RETLW Retorno, carga W

Sintaxis: [label] RETLW k
Operandos: $0 \leq k \leq 255$
Operación: (k) $\Rightarrow$ (W); TOS $\Rightarrow$ PC
Flags afectados: Ninguno
Código OP: 11 01xx kkkk kkkk
Descripción: El registro W se carga con la constante k. El PC se carga con el contenido de la cima de la pila (TOS) dirección de retorno. Consume 2 ciclos.
Ejemplo: RETLW 0x37
Después: PC = dirección de retornoW = 0x37

MOVWF Mover a f

Sintaxis: [label] MOVWF f
Operandos: $0 \leq f \leq 127$
Operación: W $\Rightarrow$ (f)
Flags afectados: Ninguno
Código OP: 00 0000 1fff ffff
Descripción: El contenido del registro W pasa el registro f.
Ejemplo: MOVWF REG,0
Antes: REG = 0xFF, W = 0x4F
Después: REG = 0x4F, W = 0x4F

MOVF Mover a f

Sintaxis: [label] MOVF f,d
Operandos: $d \in [0,1]$, $0 \leq f \leq 127$
Operación: (f) $\Rightarrow$ (dest)
Flags afectados: Z
Código OP: 00 1000 dfff ffff
Descripción: El contenido del registro f se mueve al destino d. Si des 0, el resultado se almacena en W; si d es 1, se almacena en f. Permite verificar el registro, puesto que afecta a Z.
Ejemplo: MOVF REG,0
Después: W = REG

NOP No operar

Sintaxis: [label] NOP
Operandos: Ninguno
Operación: No operar
Flags afectados: Ninguno
Código OP: 00 0000 0xx0 0000
Descripción: No realiza operación alguna. En realidad consume un ciclo de instrucción sin hacer nada.
Ejemplo: nop
Después: todo igual

RETURN Retorno de rutina

Después: [label] RETURN
Operandos: Ninguno
Después: TOS ⇒PC
Flags afectados: Ninguno
Código OP: 00 0000 0000 1000
Descripción: El PC se carga con el contenido de la cima de la pila
(TOS): dirección de retorno. Consume 2 ciclos.
Ejemplo: RETURN
Después: PC = dirección de retorno.

SLEEP Modo bajo consumo

Sintaxis: [label] SLEEP
Operandos: Ninguno
Operación: 0x00 ⇒WDT; 1 ⇒ / TO, 0 ⇒ WDT Preescaler, 0 ⇒/ PD
Flags afectados: / PD, / TO
Código OP: 00 0000 0110 0011
Descripción: El bit de energía se pone a 0 y a 1 el de descanso. El WDT y su preescaler se borran. El micro para el oscilador, yendo al modo “durmiente” .
Ejemplo: SLEEP
Preescales WDT = 0,
/TO = 1, /PD = 1

RLF Rota f a la izquierda

Sintaxis: [label] RLF f,d
Operandos: d ∈ [0,1], 0 ≤ f ≤ 127
Operación: Rotación a la izquierda
Flags afectados: C
Código OP: 00 1101 dfff ffff
Descripción: El contenido de f se rota a la izquierda. El bit de mayor peso de f pasa al carry y el carry se coloca en el de menor peso. Si d es 0, el resultado se almacena en W; si d es 1, se almacena en f.
Ejemplo: RLF REG,0
Antes: REG = 1110 0110, C = 0
Después: REG = 1110 0110,W = 1100 1100, C = 1

RRF Rota f a la derecha

Sintaxis: [label] RRF f,d
Operandos: d ∈ [0,1], 0 ≤ f ≤ 127
Operación: Rotación a la derecha
Flags afectados: C
Código OP: 00 1100 dfff ffff
Descripción: El contenido de f se rota a la derecha. El bit de menos peso de f pasa al carry y el carry se coloca en el de mayor peso. Si d es 0, el resultado se almacena en W; si d es 1, se almacena en f.
Ejemplo: RRF REG,0
Antes: REG = 1110 0110, C = 1
Después: REG = 1110 0110,W = 1111 0011, C = 0

SWAPF Intercambio de f

Sintaxis: [label] SWAPF f,d
Operandos: d ∈ [0,1], 0 ≤ f ≤ 127
Operación: (f<3:0>) → (destination<7:4>), (f<7:4>) → (destination<3:0>)
Flags afectados: Ninguno
Código OP: 00 1110 dfff ffff
Descripción: Los 4 bits de más peso y los 4 de menos son intercambiados.Si d es 0, el resultado se almacena en W; si d es 1, se almacena en f.
Ejemplo: SWAPF REG,0
Antes: REG = 0xA5
Después: REG = 0xA5, W = 0x5A

SUBWF Resta f – W

Sintaxis: [label] SUBWF f,d
Operandos: d ∈ [0,1], 0 ≤ f ≤ 127
Operación: (f) – (W) ⇒ (dest)
Flags afectados: C, DC, Z
Código OP: 00 0010 dfff ffff
Descripción: Mediante el método del complemento a dos, el contenido deW es restado al de f. Si d es 0, el resultado se almacena en W; si d es 1,se almacena en f.
Ejemplos: SUBWF REG,1
Antes: REG = 0x03, W = 0x02, C = ?
Después: REG=0x01, W = 0x02, C=1, z=0
Antes: REG = 0x02, W = 0x02, C = ?
Después: REG=0x00, W =0x02, C= 1, z=1
Antes: REG= 0x01, W= 0x02, C= ?
Después: REG=0xFF, W=0x02, C= 0, z=0 (Resultado negativo)

SUBLW Resta Literal – W

Sintaxis: [label] SUBLW k
Operandos: 0 ≤ k ≤ 255
Operación: (k) - (W) ⇒ (W)
Flags afectados: Z, C, DC
Código OP: 11 110x kkkk kkkk
Descripción: Mediante el método del complemento a dos el contenido deW es restado al literal. El resultado se almacena en W.
Ejemplos: SUBLW 0x02
Antes:W=1,C=?.
Después: W=1, C=1
Antes:W=2,C=?.
Después: W=0, C=1
Antes:W=3,C=?.
Después:W=FF,C=0 (El resultado es negativo)

XORWF W XOR F

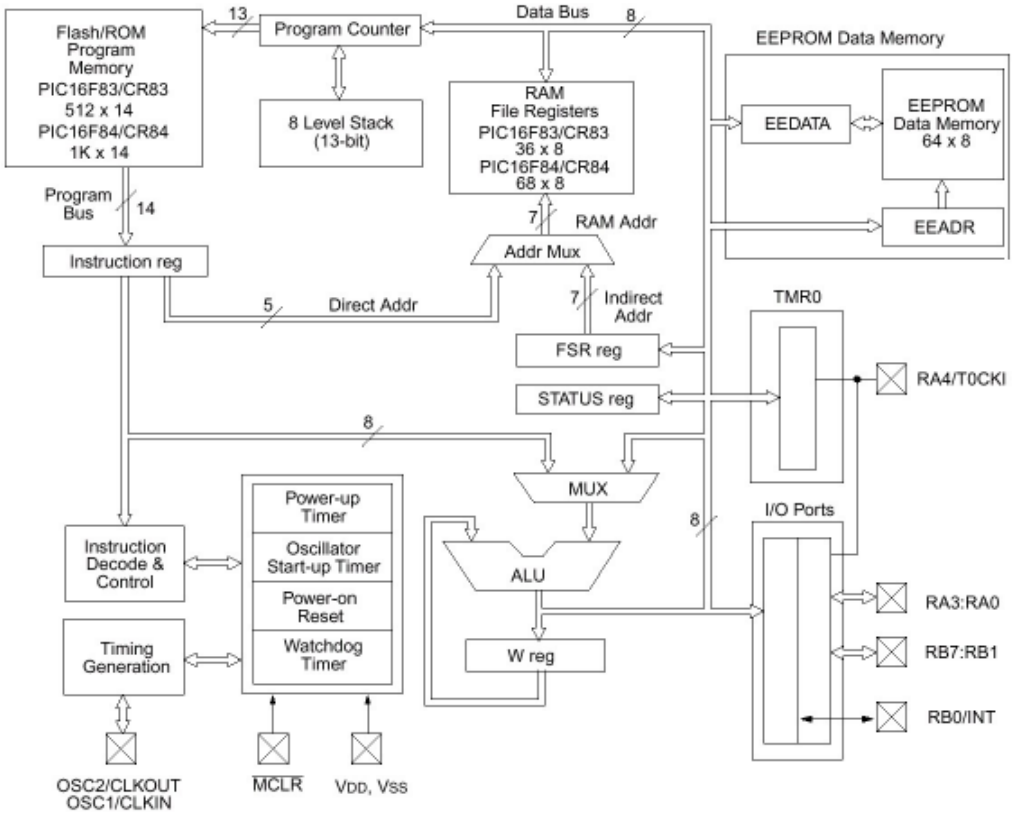
Sintaxis: [label] XORWF f,d
Operandos: d ∈ [0,1], 0 ≤ f ≤ 127
Operación: (W) XOR (f) ⇒ (dest)
Flags afectados: Z
Código OP: 00 0110 dfff ffff
Descripción: Realiza la operación lógica XOR entre los registros W y f.Si d es 0, el resultado se almacena en W, si d es 1 se almacena en f.
Ejemplo: XORWF REG,0
Antes: W = 0xB5, REG = 0xAF
Después: W = 0xB5, REG = 0x1A

XORLW W XOR literal

Sintaxis: [label] XORLW k
Operandos: 0 ≤ k ≤ 255
Operación: (W) XOR (k) ⇒ (W)
Flags afectados: Z
Código OP: 11 1010 kkkk kkkk
Descripción: Se realiza la operación lógica XOR entre el contenido delregistro W y k, guardando el resultado en W.
Ejemplo: XORLW 0xAF. Antes: W=0xB5. Después: W=0x1A

La gama media tiene un total de 35 instrucciones, cada una de las cuales ocupa 14 bits.

- **Diagrama en bloques:**



- **Memoria de programa:**

La memoria de programa es del tipo flash. La memoria flash es una memoria no volátil, de bajo consumo que se puede escribir y borrar eléctricamente. Es programable en el circuito como la EEPROM, pero tiene mayor densidad y es más rápida.

El PIC16F84 posee una memoria de programa de 1K palabras; es decir, permite hasta 1024 instrucciones de 14 bits cada una (de la 0000h a la 03FFh).

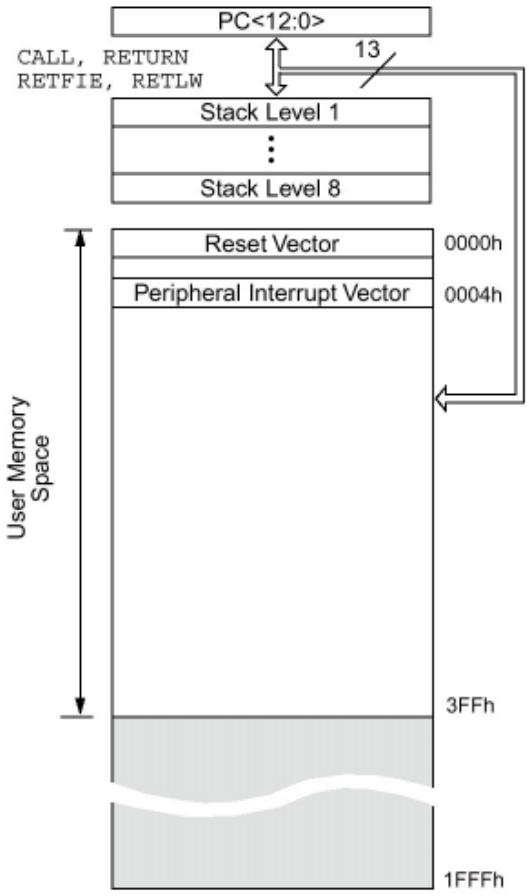
En la posición **0000h** se encuentra el **vector de reset**. Cuando se produce un reset, el programa salta a dicha posición (0000h).

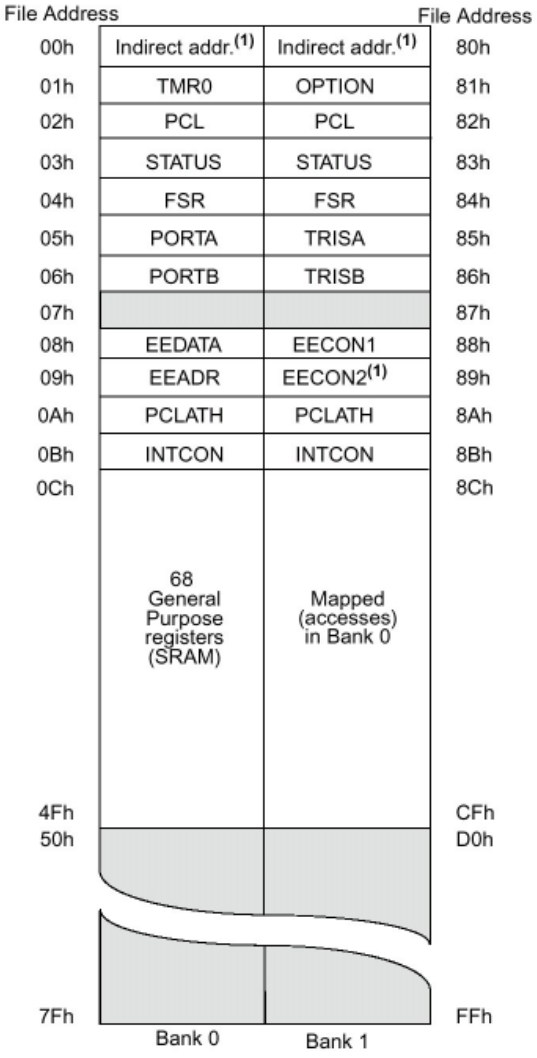
- **Memoria de datos:**

Se encuentra en dos zonas bien diferenciadas:

- Memoria tipo RAM (SRAM): Se divide en dos bancos o páginas de 128 registros de 8 bits cada uno, aunque sólo los 80 primeros de la página '0' (del 00h al 4Fh) y los 12 primeros de la página '1' (80h al 8Bh) se utilicen en el PIC16F84. Los primeros 12 registros de cada página son específicos (SFR) y los restantes (68 de la página '0') son de propósito general (GPR).
- La memoria de datos del tipo EEPROM está compuesta por 64 registros de 8 bits cada uno. Este tipo de memoria es capaz de guardar la información más de 40 años. No podemos acceder directamente a estos registros; para ello hay que utilizar registros específicos (SFR).

- **Mapa de memoria:**





- **Puertas de entrada y salida:**

El microcontrolador PIC16F84 posee 13 patillas, cada una de las cuales se puede configurar como entrada o como salida. Éstas se encuentran distribuidas en dos puertos: el puerto A con cinco patillas –desde RA0 a RA4– y el puerto B con ocho patillas –desde RB0 a RB7–.

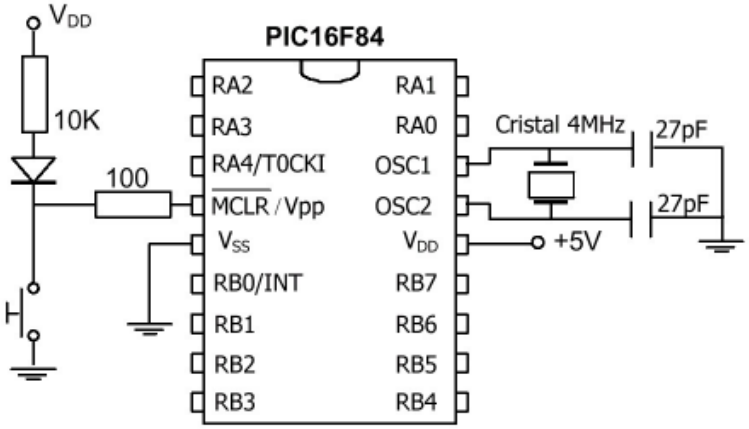
- RA0- RA4. Son los terminales de entrada/salida del puerto A. Pueden suministrar una corriente por cada pin de 20mA.; pero, la suma de las cinco líneas del puerto A no puede exceder de 50 mA. La corriente absorbida por cada pin puede ser de 25 mA; pero, la suma de las cinco líneas no puede exceder de 80 mA. El pin RA4 tiene una doble función que se puede seleccionar por programa y es la de ser la entrada del contador/temporizador TMR0, es decir T0CK1.

- RB0-RB7. Son los terminales de entrada/salida del puerto B. Pueden suministrar una corriente por cada pin de 20 mA; pero, la suma de las ocho líneas no puede superar los 100 mA. La corriente absorbida por cada pin puede ser de 25 mA; pero, la suma de todas no puede exceder de 150 mA. El pin RB0 tiene una doble función seleccionable por programa, que es la de ser entrada de interrupción externa (INT). Los pines del RB4 al RB7 tienen una doble función

que se puede seleccionar por programa, que es la de ser entrada de interrupción interna por cambio de estado.

- **Conexionado típico:**

En los circuitos donde se utiliza el PIC16F84 es habitual emplear tensión de alimentación de +5V y, como circuito de reloj externo, un cristal a una frecuencia de 4MHz. Con esta configuración, el conexionado fijo para cualquier aplicación es:



Las patillas que no están conectadas son las dedicadas a transferir la información con los periféricos que utilice la aplicación.

FICHA 4

Programación de microcontroladores

Registros del microcontrolador

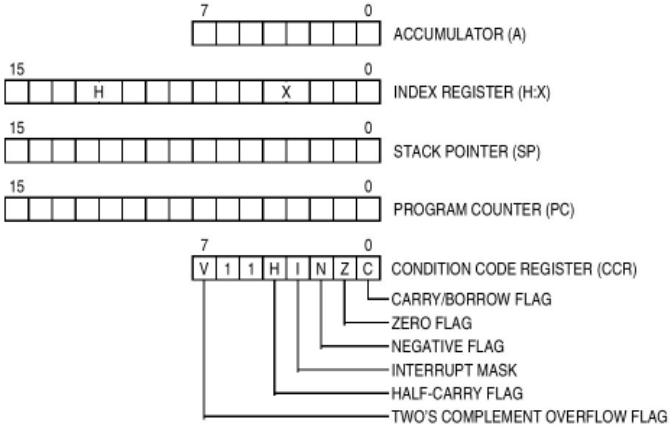
Decodificador de instrucciones

Cada código de operación es decodificado para determinar cuántos operandos se necesitan y qué secuencia de etapas se requiere para completar una instrucción. Al finalizar una instrucción, el próximo código de operación es leído y decodificado.

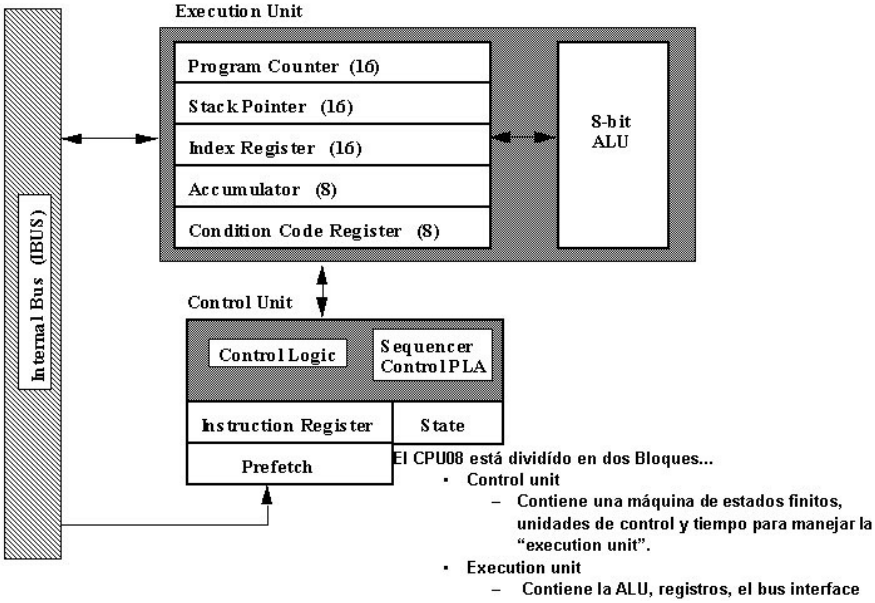
Un elemento central de la sección de control de la CPU es el **decodificador de instrucciones.**

La CPU contiene cinco registros que se alojan dentro del microprocesador (no son parte del mapa de memoria):

- registro acumulador –A–
- registro índice –H:X–
- registro contador de programa –PC–
- registro de código de condición –CCR–
- registro puntero a la pila –*stack pointer*; SP–.



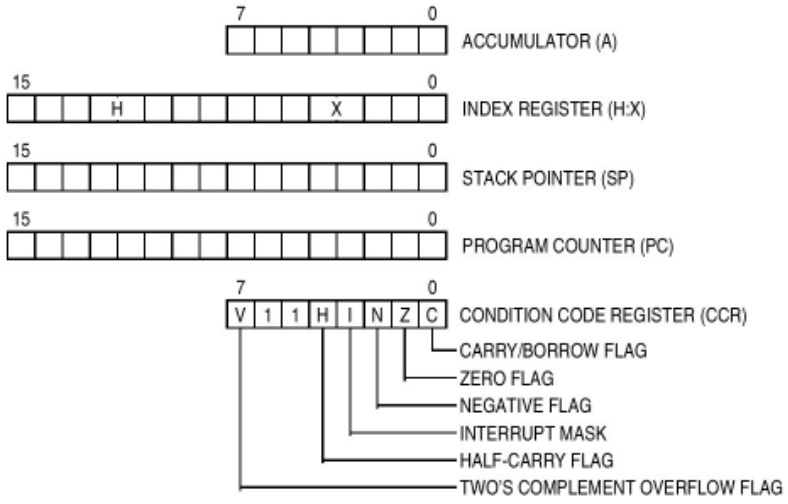
CPU08 Arquitectura de ejecución



El conjunto de registros de la CPU es denominado, a veces, el **modelo de programación** (Un programador experimentado puede aventurarnos la suerte de una computadora a partir de su modelo de programación).

Diferentes CPU poseen diferentes conjuntos de registros internos. Las diferencias son, principalmente, el número y el tamaño de estos registros. Los registros de la CPU que encontramos en el MC68HC08 constituyen un relativamente sencillo conjunto de registros de la CPU, que resulta ser representativo de todos los tipos de registros de la CPU y puede usarse para explicar todo lo necesario sobre los conceptos fundamentales.

Aquí brindamos una breve descripción de los registros del MC68HC08, como una introducción a la arquitectura de la CPU en general[1]:



Registros de la CPU MC68HC08

- El registro A, es un registro de almacenamiento temporario de 8 bits. Se lo llama **acumulador** ya que se usa a menudo para alojar uno de los operandos o el resultado de una operación aritmética.

  El acumulador es un registro de propósitos generales. Él es directamente accesible a la CPU para operaciones no aritméticas. El acumulador es usado durante la ejecución de un programa donde el contenido de alguna posición de memoria es cargado en el acumulador.

  La instrucción *almacenar*, también causa que el contenido del acumulador sea almacenado en alguna posición de memoria preestablecida.

- El **registro índice –H:X–** es de 16 bits de longitud. El principal propósito de un registro índice es apuntar a un área en la memoria desde donde la CPU cargará (leerá) o almacenará (escribirá) información; además, puede servir como un simple registro de almacenamiento temporario.

  Un registro índice suele llamarse **registro puntero.** Está formado por una parte "baja" (el byte de menor peso) denominado "X" y una parte alta (el byte de mayor peso) denominado "H". Estos registros se encuentran concatenados para formar un único registro H:X. Esto permite direccionamientos indexados de hasta 64 Kbytes de espacio de memoria.

  Para conservar la compatibilidad con la familia HC05, en el registro índice puede utilizarse sólo la parte baja ("X"), en los distintos modos de direccionamiento, de igual forma que en ésta. Sólo se debe tener en cuenta que cuando en una instrucción con direccionamiento indexado se menciona el registro "X", en realidad se está haciendo mención al registro concatenado H:X de 16 bits de largo, por lo que deberá ponerse a cero (forzar el valor $00) la parte superior del registro índice, o sea "H", para guardar total compatibilidad con la familia HC705. De esta forma, cuando se utilice el registro índice, su contenido será $00xx, donde "xx" contendrá el valor del registro "X" propiamente dicho.

---

[1] Veremos información más detallada respecto a los registros del 68hC08 en el repertorio de instrucciones.

- El **registro contador de programa –PC–** es usado por la CPU para no perder de vista la dirección de la próxima instrucción a ejecutar. Al resetear la CPU (encenderla), el PC es cargado con el contenido de un par de posiciones de memoria específico, denominado **vector de reset.** El número de bits del PC coincide exactamente con el número de líneas del bus de direcciones. Esto determina el total espacio de memoria potencialmente disponible que puede ser accedido por la CPU.

  El contador de programa (PC), es de 16 bits de longitud; puede moverse entre $0000 y $FFFF. De esta forma, el PC puede moverse, teóricamente (muchos MCU de la familia HC908, poseen memorias de programas inferiores a los 64Kbytes), por los 64 Kbytes de espacio de memoria.

  Durante el reset, el contador de programa (PC) se carga con la dirección contenida en el **vector de reset** que, para el MC68HC908, se encuentra en la posición $FFFE y $FFFF.

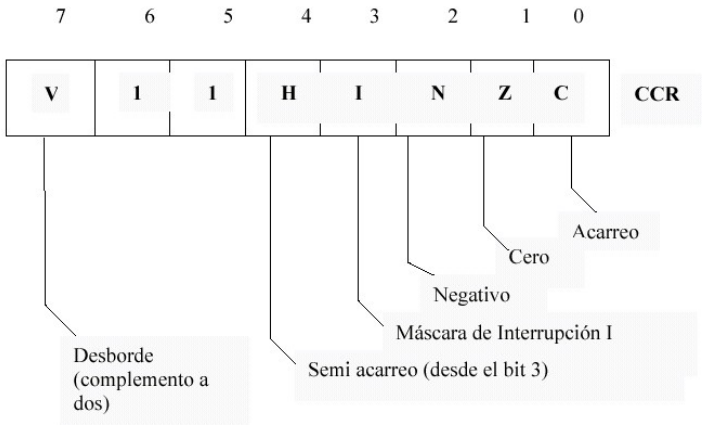
  La dirección contenida en el vector es la dirección de la primera instrucción a ser ejecutada después de salir del estado de RESET.

- El **registro de código de condición –CCR–** es de 8 bits; almacena indicadores de estado que reflejan al resultado de alguna operación previa de la CPU. Las **instrucciones de bifurcación** usan a los bits de estado para tomar simples decisiones respecto a su estado.

  El registro de código de condición contiene una máscara de interrupción y seis indicadores de estado que reflejan el resultado de operaciones aritméticas y de otro tipo de la CPU.

  Las seis banderas son:

  - desborde –V; Overflow–,
  - semi acarreo –H–,
  - máscara de interrupción –I–,
  - negativo –N–,
  - cero –Z– y
  - acarreo/préstamo –C–.



- El **puntero a la pila –*stack pointer*; SP–** es usado como puntero a la próxima posición disponible de una pila –*stack*– del tipo último en ingresar / primero en salir –LIFO; *Last In, First Output* –.

  El stack puede tomarse como una pila de cartas, en la que cada carta almacena un solo byte de información. En el momento que se desee, la CPU puede poner una carta arriba de la pila o retirar una de arriba de la pila. No podemos retirar una carta del interior de la pila antes de haber retirado todas las cartas que tiene encima. La CPU sigue el comportamiento del stack con el SP. El SP apunta a la posición de memoria libre, la que se considera es la dirección de la próxima carta disponible. Cuando la CPU agrega una porción de dato en la pila, el dato es escrito en la posición apuntada por el SP y el SP, luego, se decrementa para apuntar a la posición de memoria previa. Cuando la CPU retira una porción de dato de la pila, el SP se incrementa para apuntar a la posición previamente usada y de ella se retira el dato. Al encender la CPU o bien luego de ejecutar la instrucción *reset stack pointer* –RSP–, el SP apunta a una específica posición de memoria en RAM.

  El puntero de pila es un registro de 16 bits que contiene la dirección del próximo lugar en la pila. Durante un reset, el puntero de pila es preseteado a $00FF. La instrucción *reset stack pointer* –RSP–, setea al byte menos significativo a $FF y no afecta al byte más significativo. Esto se hace para mantener la compatibilidad con el modo de funcionamiento del puntero de pila de la familia HC05.

  El puntero de pila es decrementado cuando un dato es almacenado –*push*– dentro de la pila e incrementado cuando un dato es recuperado –*pull*– desde la pila. La localización de la pila es arbitraria y puede ser «reubicada» en cualquier parte de la memoria RAM. Moviendo el puntero fuera de la página 0 ($0000 a $00FF), libera el espacio del direccionamiento directo. Para una operación correcta, el puntero de pila debe apuntar solamente posiciones de RAM aunque, por su longitud, pueda "barrer" todo el espacio de memoria del MCU.

  Gracias a esta característica, en los modos de direccionamiento con el SP con 8 bits de offset y 16 bits de offset, el puntero de pila puede funcionar como un segundo registro índice de 16 bits o bien para acceder a datos en la pila. El uso del SP como un segundo registro índice es muy utilizado en los compiladores de lenguaje de alto nivel, como los compiladores "C" y otros.

Lenguaje assembler

El procesador o CPU está diseñado para ejecutar instrucciones y puede, en un pequeño lapso, "reconocer" y ejecutar miles de ellas.

Instrucción
Cada instrucción es una secuencia de unos y ceros residente en la memoria, que la CPU puede interpretar a través de sus circuitos.

Cuando se escribe un programa en cualquier lenguaje, el producto final que el procesador puede entender es ese mismo programa traducido a lenguaje de máquina.

Se llega al programa en lenguaje de máquina después de una serie de "traducciones" que va sufriendo el programa fuente.

De este modo, cada instrucción de un programa se transforma, finalmente, en una o más instrucciones en lenguaje de máquina.

En particular, cada instrucción de un programa escrito en un lenguaje de alto nivel

(C++, PASCAL, BASIC, etc.), equivale a varias instrucciones en lenguaje de máquina. Distinto es lo que ocurre con el lenguaje assembler.

Lenguaje assembler
Es un lenguaje de bajo nivel que representa, en realidad, el código mnemotécnico del lenguaje de máquina; es decir que cada instrucción de assembler equivale a una instrucción en código de máquina.

Para convertir un programa en lenguaje assembler a código de máquina, se debe convertir cada instrucción del programa en assembler a su equivalente en código de máquina. Ésta es una de las tareas efectuada por el compilador.

El formato de una instrucción en código de máquina incluye la suficiente información para identificar qué operación es hecha, qué operando u operandos son usados, si la operación que se efectúa es sobre byte o sobre palabra, si la operación incluye a operandos que están ubicados en registros o en un registro y en una posición de memoria, y, si uno de los operandos es una posición de memoria, cómo es la generación de su dirección. Toda esta información está codificada en los bits de cada instrucción en código de máquina.

Las instrucciones en código de máquina del microcontrolador varían en el número de bytes que necesitan para ser codificadas.

Microinstrucción
Es la unidad más pequeña que puede ser ejecutada y se corresponde directamente a los circuitos electrónicos de la CPU.

Algunas instrucciones pueden ser codificadas en un solo byte, como es el caso de operaciones simples con un registro; otras, en cambio, requieren dos o más bytes. El máximo número de bytes que puede ocupar una instrucción es de 6.

Cada instrucción de máquina es, en realidad, un conjunto de microinstrucciones.

Un programa es, así, un conjunto ordenado de instrucciones que determina el procesado de los datos y la obtención de los resultados. Cada procesador entiende un conjunto de instrucciones, a las cuales se llama “instrucciones máquina”; en este microcontrolador, cada una de ellas consta de 8 bits con los cuales se indica el código de la operación.

Como sería muy complejo escribir las instrucciones, a cada una se la referencia con un **mnemónico**, que es un conjunto de letras que expresa, de forma resumida, la operación que hace la instrucción. Al lenguaje que utiliza estos mnemónicos se lo llama ensamblador –*assembler*–.

Código de operación –opcode–
Instruye a la CPU en la ejecución de una muy específica secuencia de etapas que debe seguirse para cumplir con la operación propuesta.

Las computadoras utilizan otros códigos para dar instrucciones a la CPU. Este código se denomina código de operación

Las computadoras de distintos fabricantes usan diferentes repertorios de códigos de operación, previstos en la lógica cableada de la CPU. El repertorio –*set*– de instrucciones para una CPU es el conjunto de instrucciones que ésta es capaz de realizar. Los códigos de operación son una representación del set de instrucciones y los mnemónicos son otra[2].

Veamos...

Un opcode tal como $4C es interpretado por la CPU, pero no es fácilmente manejable por una persona. Para resolver este problema, se usa un sistema de mnemónicos

---

[2] Aún cuando difieren de una a otra, todas las computadoras digitales binarias realizan el mismo tipo de tareas básicas de modo similar. La CPU en la MCU MC68HC08 puede entender alrededor de 119 instrucciones básicas. Algunas de éstas presentan mínimas variaciones, cada una de las cuales requiere su propio código de operación. El set de instrucciones del MC68HC08 incluye más de 200 opcodes distintos. CPU08 posee una “prebúsqueda” de código, por lo que la velocidad final se ve multiplicada por cinco con respecto a las aplicaciones de CPU anteriores.

de instrucción equivalentes: El opcode $4C corresponde al mnemónico INCA, que se lee “Incrementar el acumulador”.

Aunque contamos con la información impresa que muestra la relación entre el mnemónico de cada instrucción y el opcode que la representa, ésta es rara vez utilizada por el programador puesto que el proceso de traducción es realizado automáticamente por un programa de computadora específico denominado ensamblador. Este programa es el que convierte los mnemónicos de las instrucciones de un programa en una lista de códigos de máquina (códigos de operación e información adicional), para que puedan ser utilizados por la CPU.

Un ingeniero desarrolla un grupo de instrucciones para una computadora en la forma de mnemónicos y, luego, utiliza un ensamblador para trasladar estas instrucciones a los opcodes que la CPU pueda entender.

Las computadoras aceptan dos niveles lógicos (0 y 1), con los que trabaja en el sistema de numeración binario. Las personas, por tener diez dedos en las manos, trabajamos con el sistema de numeración decimal.

Los números hexadecimales utilizan dieciséis símbolos: del 0 al 9 y de la A a la F. Cada dígito hexadecimal puede representarse mediante cuatro dígitos binarios. Existe una equivalencia entre decimal, binario y hexadecimal. Para poder distinguir un valor hexadecimal de otro decimal, colocaremos un símbolo $ antecediendo al hexadecimal o una “H” después.

El ASCII –*American Standard Code for Information Interchange*–[3] es un código ampliamente aceptado y nos permite representar tanto información alfanumérica como valores binarios.

Cada instrucción o variante de una instrucción posee un único código de operación (valor binario) que la CPU reconoce como un pedido para realizar una instrucción específica. Las CPU de diferentes fabricantes poseen repertorios de códigos de operación distintos.

Programar en lenguaje ensamblador es programar con las mismas instrucciones que, directamente, puede ejecutar el procesador. Hay lenguajes de alto nivel en los que una instrucción equivale a muchas instrucciones-máquina. Son más fáciles de utilizar, pero la utilización de los lenguajes más cercanos a la máquina (ensamblador) representan un considerable ahorro de código en la confección de los programas, lo que es muy importante dada la limitada capacidad de la memoria de instrucciones.

Entre los lenguajes de alto nivel que se pueden emplear con estos microcontroladores se encuentran el lenguaje C y Basic.

Los compiladores –*assembler compiler*– son programas que se encargan de traducir el programa escrito en lenguaje ensamblador (u otro lenguaje) a código máquina, único lenguaje que el microcontrolador es capaz de entender. Tras la compilación, se graba este código binario en la memoria de programa del microcontrolador o se realiza una simulación con algún software de depuración –*debugger*–.

---

[3] El código ASCII es una correlación ampliamente aceptada entre caracteres alfanuméricos y valores binarios específicos. En este código, el número $41 corresponde a una letra “A” mayúscula, el $20 al carácter espacio, etc. Este estándar hace posible las comunicaciones entre equipos hechos por diversos fabricantes, puesto que todas las máquinas utilizan el mismo código.

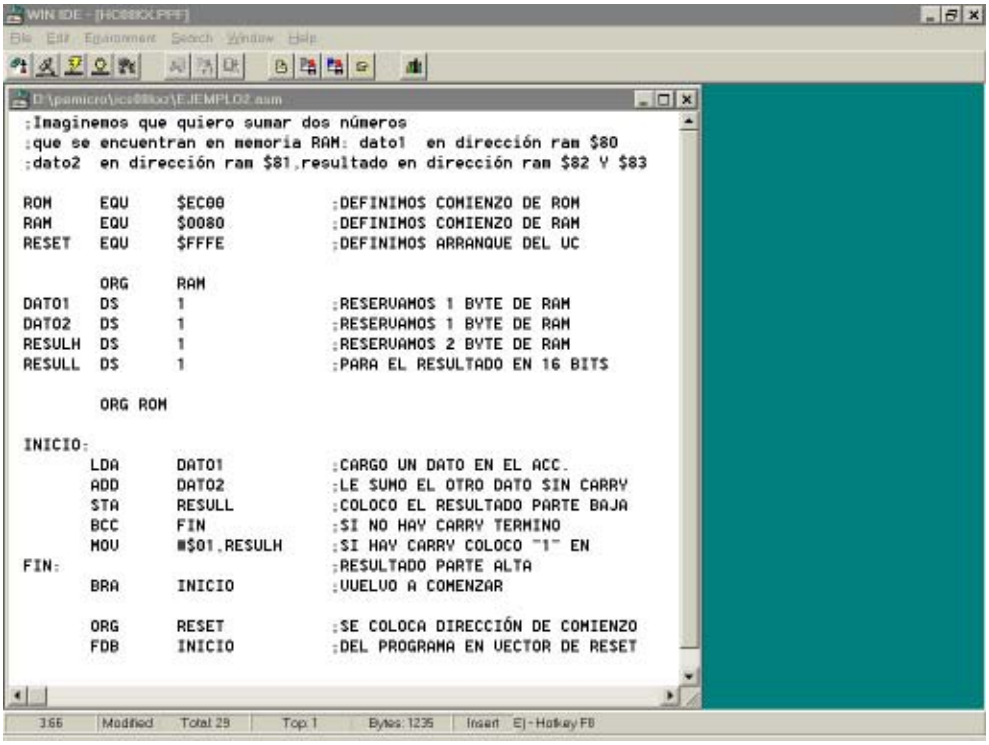
Veamos cómo es todo el proceso con un ejemplo:

Imaginemos que quiero sumar dos números que se encuentran en memoria RAM:

dato1 en dirección ram $80
dato2 en dirección ram $81

resultado en dirección ram $82 y $83

Los pasos son: Edición con el *WinIDE®*[4] –entorno de trabajo que integra edición, ensamblado y linkeado–, es decir codificado nemotécnico en archivo de texto, traducción a lenguaje de máquina y enlazado de librerías generando un archivo de extensión LST –listado, ensamblado y codificado en formato texto– y el código ejecutable para el microcontrolador en un archivo de extensión S19, quedando listo para depurarlo con un simulador o grabar la EEPROM del microcontrolador para su utilización.



Edición con Winide*, editor de texto, compilador y linkeador*

[4] *El entorno desarrollado por P&E es denominado WinIDE –Windows Integrated Development Enviroment– y es, básicamente, un poderoso conjunto de herramientas reunidas bajo una plataforma Windows 3.x, Windows 95 / 98 / Me.*
El editor de texto que incorporan los ICS tienen todas las facilidades que le otorga el entorno Windows: copiado, pegado y otras herramientas similares al conocido Wordpad de Windows, lo que hace realmente agradable la tarea, a la hora de escribir programas de gran extensión. En cuanto al compilador CASM08, es una excelente herramienta que permite detectar errores de sintaxis, duplicación de etiquetas, saltos fuera de rango y etiquetas sin correspondencia, cuando se compila el programa fuente.
Las herramientas se encuentran en:

- *http://www.pemicro.com/*
- *www.electrocomponentes.com.ar*
- *www.mcu.motsps.com*
- *http://www.motorola.com/mcu*
- *http://www.motorola.com/semiconductors/*
- *http://www.pemicro.com/*

Programa compilado y linkeado (ejemplo2.LST):

```
EJEMPLO2.asm    Assembled with      CASM08Z
                1  ;Imaginemos que quiero sumar dos números
                2  ;que se encuentran en memoria RAM: dato1  en dirección ram $80
                3  ;dato2  en dirección ram $81,resultado en dirección ram $82 Y $83
                4
0000            5  ROM      EQU    $EC00        ;DEFINIMOS COMIENZO DE ROM
0000            6  RAM      EQU    $0080        ;DEFINIMOS COMIENZO DE RAM
0000            7  RESET    EQU    $FFFE        ;DEFINIMOS ARRANQUE DEL UC
                8
0080            9           ORG    RAM
0080            10 DATO1    DS     1            ;RESERVAMOS 1 BYTE DE RAM
0081            11 DATO2    DS     1            ;RESERVAMOS 1 BYTE DE RAM
0082            12 RESULH   DS     1            ;RESERVAMOS 2 BYTE DE RAM
0083            13 RESULL   DS     1            ;PARA EL RESULTADO EN 16 BITS
                14
EC00            15          ORG ROM
                16
                17 INICIO:
EC00 [03] B680  18          LDA  DATO1          ;CARGO UN DATO EN EL ACC.
EC02 [03] BB81  19          ADD  DATO2          ;LE SUMO EL OTRO DATO SIN CARRY
EC04 [03] B783  20          STA  RESULL         ;COLOCO EL RESULTADO PARTE BAJA
EC06 [03] 2403  21          BCC  FIN            ;SI NO HAY CARRY TERMINO
EC08 [04] 6E0182 22         MOV  #$01,RESULH    ;SI HAY CARRY COLOCO "1" EN
                23 FIN:                         ;RESULTADO PARTE ALTA
EC0B [03] 20F3  24          BRA  INICIO         ;VUELVO A COMENZAR
                25
FFFE            26          ORG  RESET          ;SE COLOCA DIRECCIÓN DE COMIENZO
FFFE    EC00    27          FDB  INICIO         ;DEL PROGRAMA EN VECTOR DE RESET
                28
                295
```

Tabla de símbolos:

DATO1	0080
DATO2	0081
FIN	EC0B
INICIO	EC00
RAM	0080
RESET	FFFE
RESULH	0082
RESULL	0083

Código máquina generado:

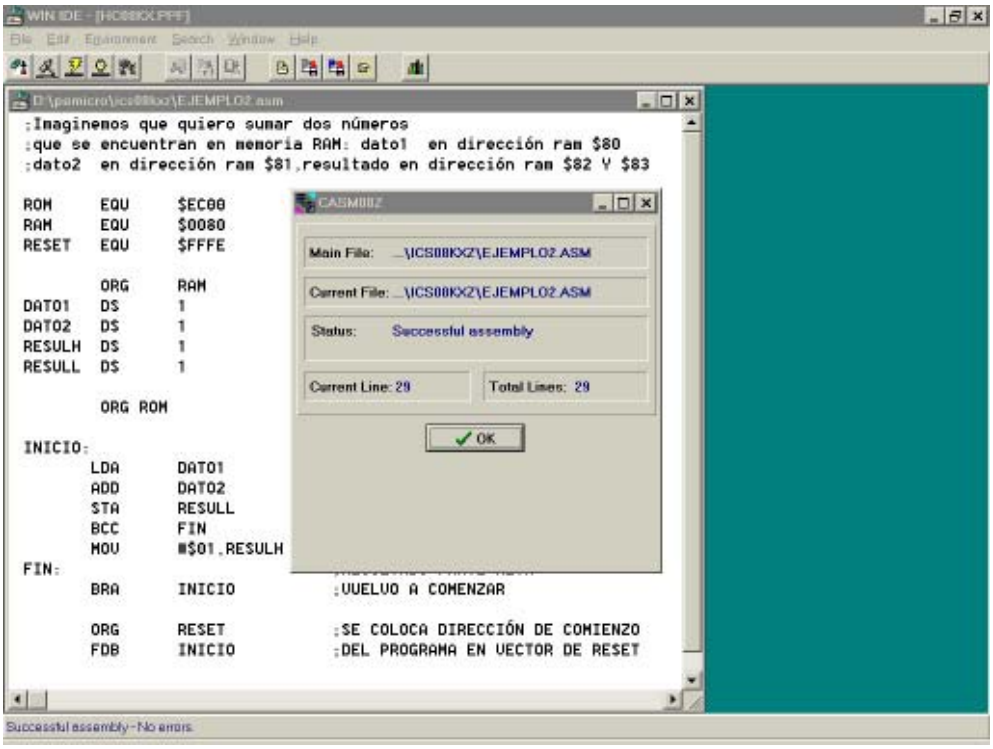
S110EC00**B680BB81B78324036E018220F3**2C
S105FFFE**EC00**11
S9030000FC

---

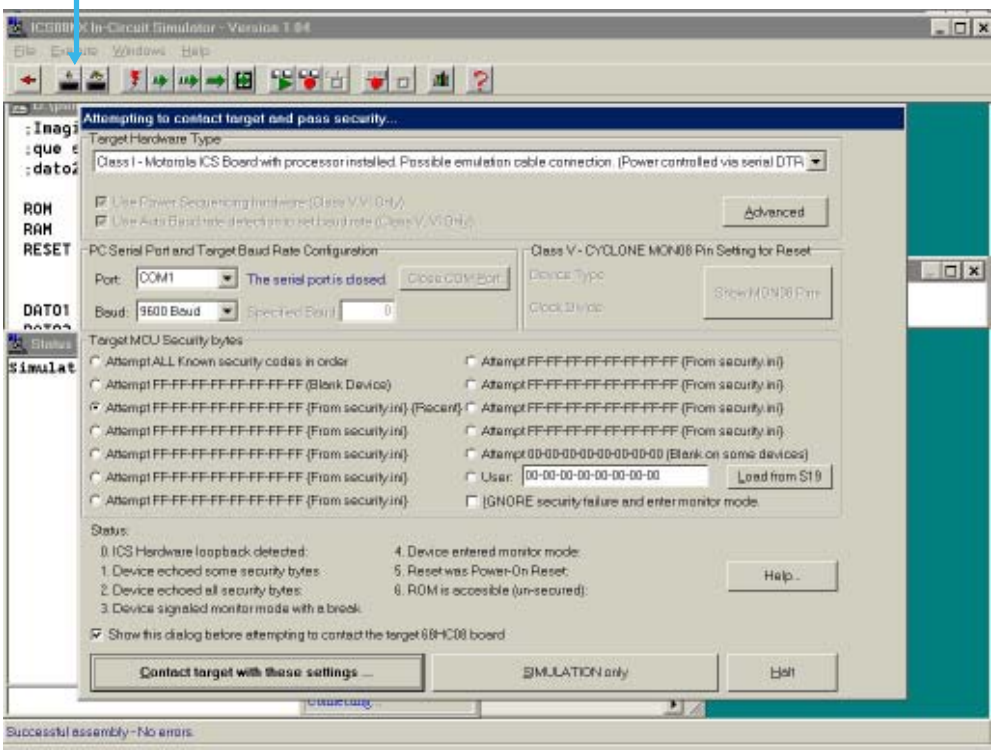
[5] *A este programa faltaría agregarle una línea, al menos, para que no se active el COP –dispositivo que resetea al microcontrolador si no es refrescado con una escritura de un registro especial–. Para esto definimos el registro y colocamos la línea siguiente*
- sta copctl
en un lugar del programa, que se ejecute cíclicamente:
COPCTL EQU $FFFF
Sta   copctl
También se puede inhibir dicho funcionamiento colocando la siguiente instrucción y correspondiente definición de registros:
CONFIG1 EQU $001F   ;Registros de configuración
CONFIG2 EQU $001E
MOV #$01,CONFIG1   ;configuración

En negrita figuran las instrucciones para el microcontrolador en código máquina: código de operación. Por ejemplo, **B6 80** que significa cargar en el registro acumulador dato de dirección $80 (LDA DATO1).

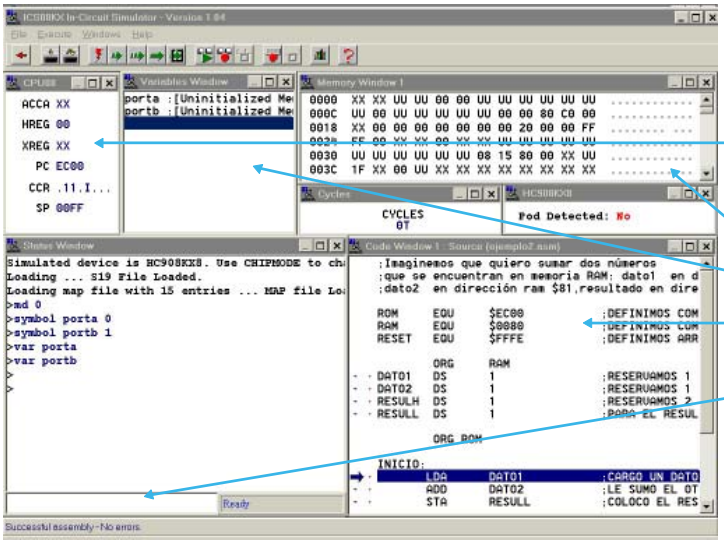
Para compilar, pulsamos el primer ícono y obtenemos el siguiente mensaje:

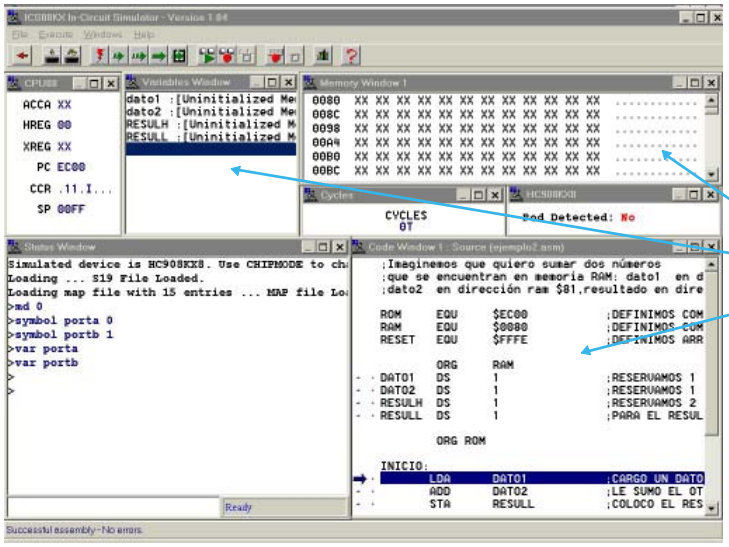


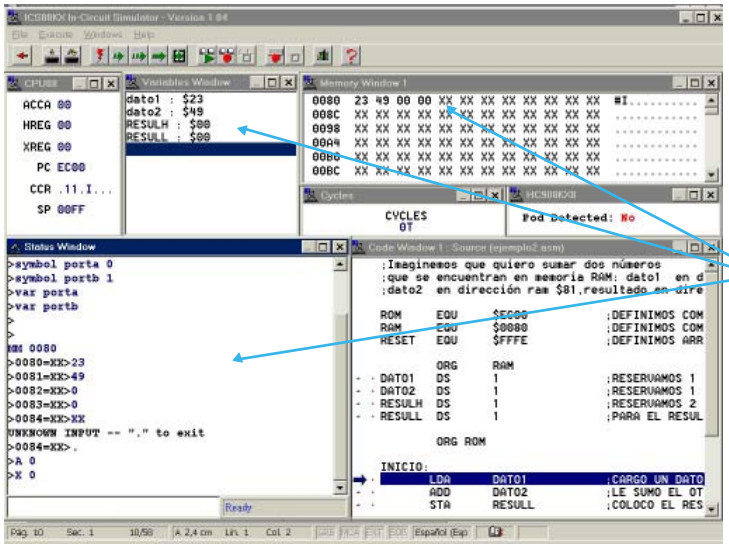
Para simulación solamente o depuración en circuito, pulsamos el siguiente icono:

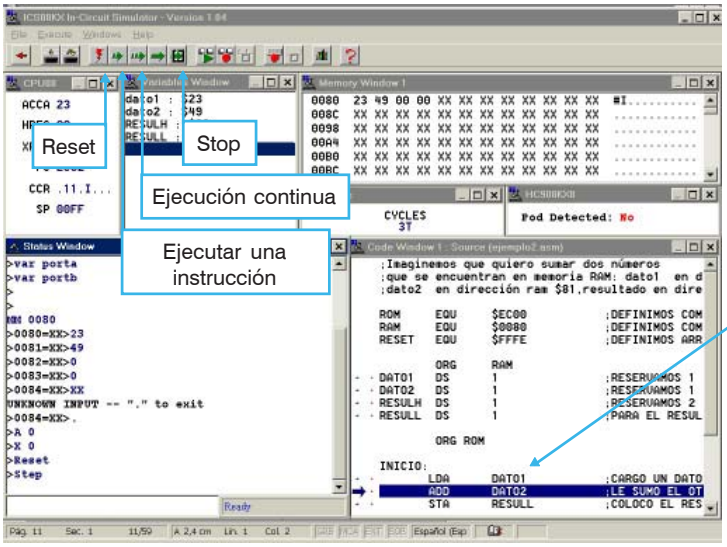


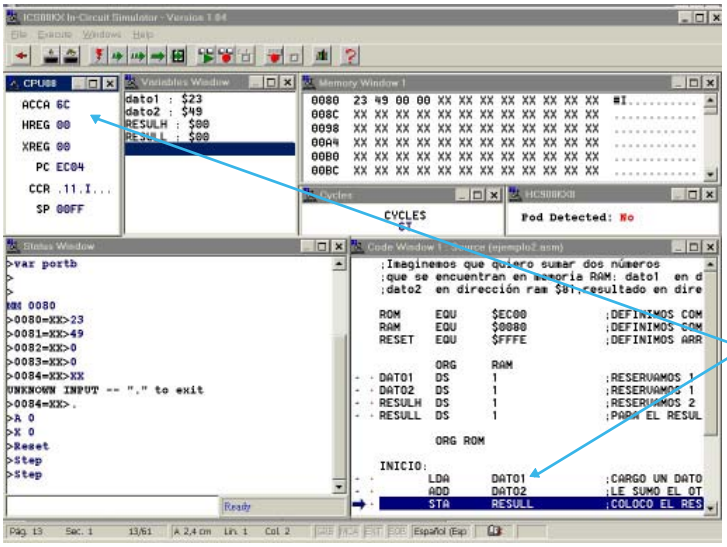
Y elegimos simulación, solamente.

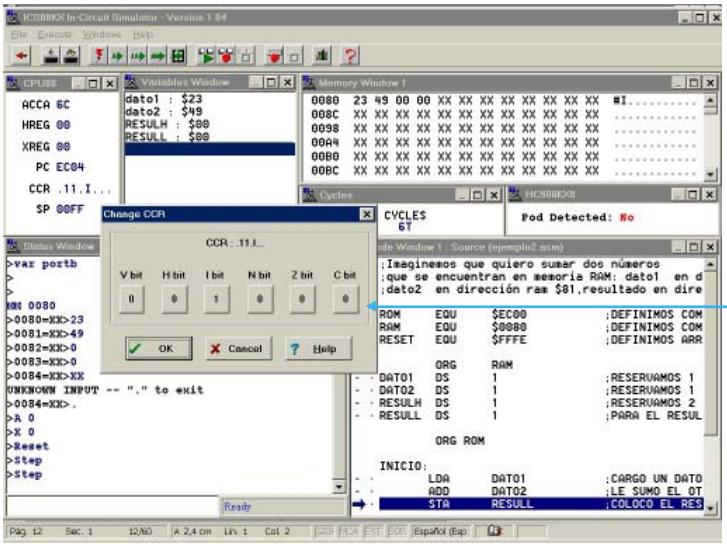

Ventana de registros de la CPU
Mapa de memoria
Variables del usuario
Seguimiento del programa
Ingreso de comandos del operador

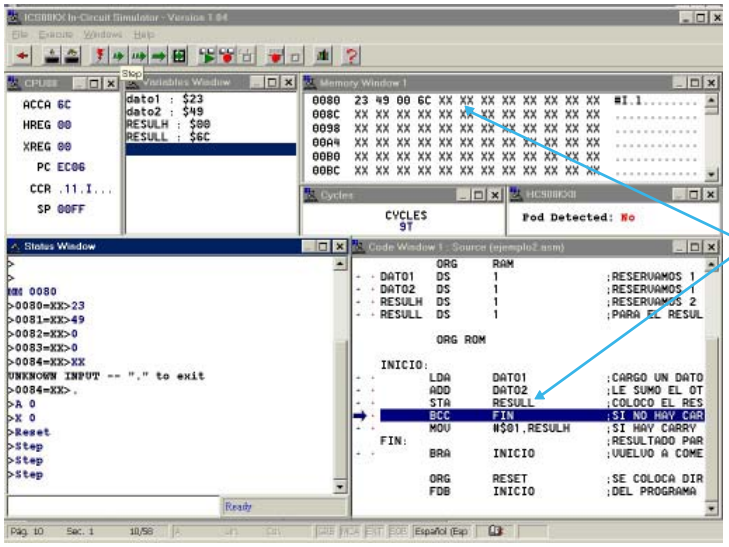

Mapa de memoria
Variables del usuario
Seguimiento del programa

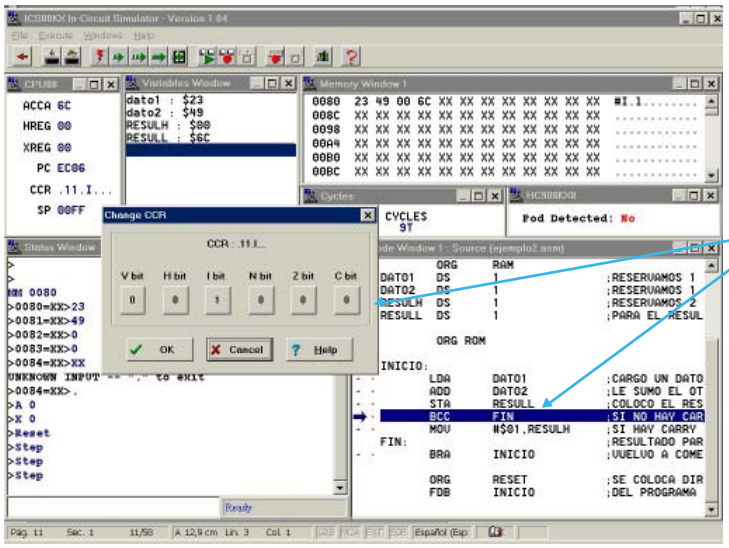

Ingreso de valores iniciales

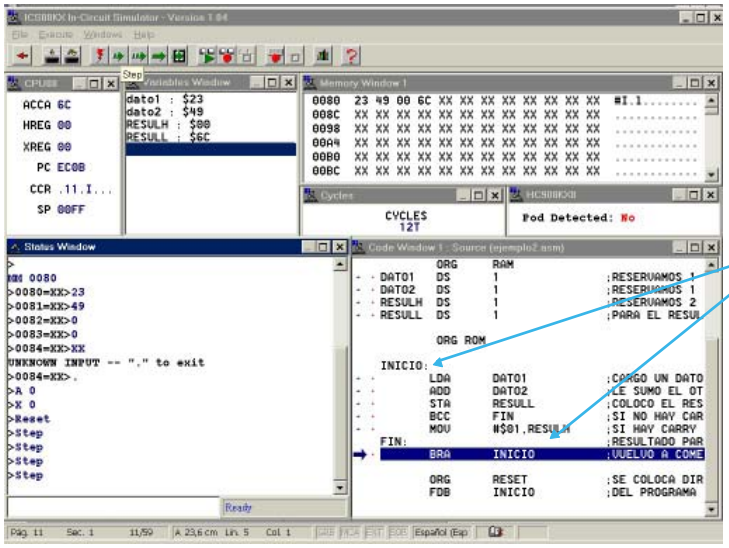

Reset
Stop
Ejecución continua
Ejecutar una instrucción
Ejecución de la primera instrucción

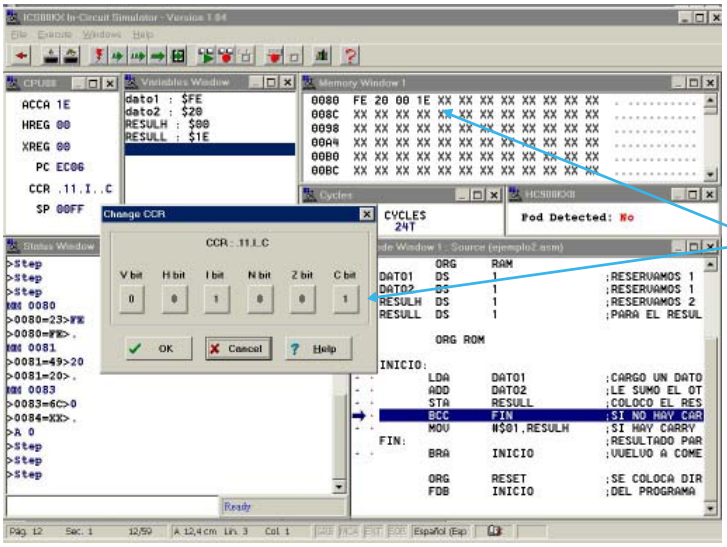

Próxima instrucción

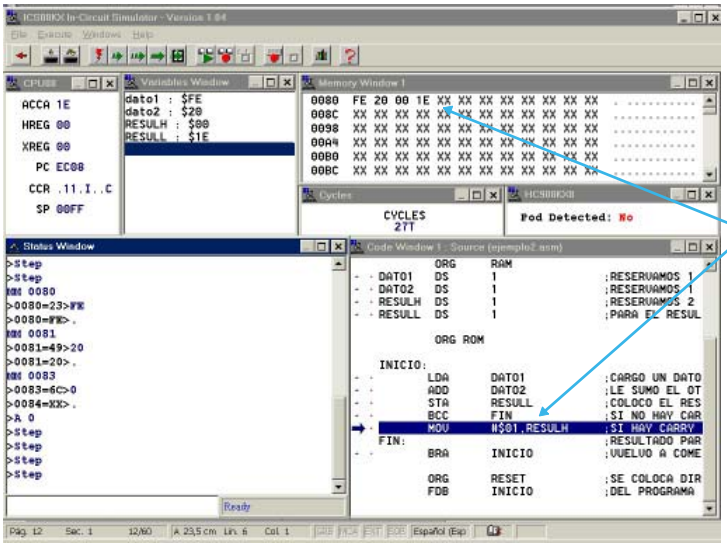

Suma sin acarreo

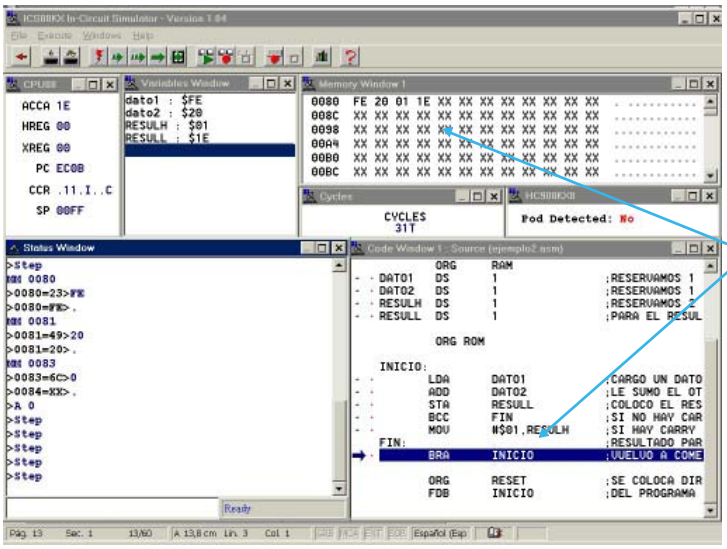

Resultado de la suma


No hay acarreo entonces ir al fin


Ir al inicio


Change CCR
CCR : .11.I..C
V bit H bit I bit N bit Z bit C bit
OK
Cancel
Help
CYCLES 24T
Pod Detected: No
dato1 : $FE
dato2 : $20
RESULH : $00
RESULL : $1E
ACCA 1E
HREG 00
XREG 00
PC EC06
CCR .11.I..C
SP 00FF
Otros valores donde la suma produjo acarreo


CYCLES 27T
Pod Detected: No
dato1 : $FE
dato2 : $20
RESULH : $00
RESULL : $1E
ACCA 1E
HREG 00
XREG 00
PC EC08
CCR .11.I..C
SP 00FF
No salto y sumo acarreo


CYCLES 31T
Pod Detected: No
dato1 : $FE
dato2 : $20
RESULH : $01
RESULL : $1E
ACCA 1E
HREG 00
XREG 00
PC EC08
CCR .11.I..C
SP 00FF
Vemos los resultados con acarreo y vuelvo al inicio a sumar

Estructura de un programa en assembler

Veremos, a continuación, los elementos que forman parte de un programa assembler, de modo de poder comenzar a escribir nuestros propios programas.

Un **programa fuente** es una secuencia de líneas de **sentencias fuente**. Cada línea de programa contiene sólo una sentencia fuente.

Un programa fuente contiene dos tipos de sentencias:

- instrucciones y
- directivas o pseudo-operaciones –*pseudo-ops*–.

Las **instrucciones** son representaciones simbólicas de las instrucciones de máquina; le indican al microprocesador qué operación debe ejecutar.

En cambio, las **pseudo-ops** son mandatos al compilador que le indican qué debe hacer con las instrucciones y datos.

La diferencia entre instrucción y pseudo-op es un concepto fundamental para interpretar correctamente cómo es la estructura de un programa assembler: mientras que las instrucciones se aplican a tiempo de ejecución, las pseudo-ops se aplican a tiempo de compilación.

Las instrucciones en assembler deben ser escritas cumpliendo ciertas reglas de sintaxis. Esto le permite al compilador poder interpretar cada uno de los elementos que forman parte de una línea de programa.

El formato general de una instrucción es el siguiente:

[Rótulo:] Código de Operación [Operando] [;Comentario]

De los cuatro campos indicados, sólo el de código de operación es obligatorio.

Los campos de rótulo y de comentario son opcionales. El campo de operando debe ir solamente en aquellas instrucciones que lo requieren (de lo contrario, debe ser omitido).

Todos los campos de una misma instrucción deben estar siempre en la misma línea de programa y separados entre sí por, al menos, un blanco.

Un ejemplo de instrucción usando los 4 campos, es el siguiente:

```
start:  MOV  #$08,PB    ;Inicializar puerto B
```

En este ejemplo:

- *start:* es un rótulo,
- *MOV* es el código de operación,
- *#$08,PB* es el campo de operando y
- *; Inicializar puerto B* es el comentario.

- **Rótulo:**

El rótulo o etiqueta es la asignación de un nombre a una instrucción. Esto permite que otras instrucciones hagan referencia a la instrucción rotulada a través de ese nombre.

El rótulo se utiliza, fundamentalmente, para definir variables de datos que luego el programa va a utilizar para operar y para establecer puntos del programa donde se habrá de bifurcar por alguna condición.

Resulta aconsejable:

- Hacer los nombres lo más cortos posible, siempre que ello sea razonable. Por ejemplo, es preferible el rótulo KPH en vez de KILOMETROS_POR_HORA.
- Definir nombres sencillos para evitar errores de escritura. Estos errores ocurren, generalmente, cuando se definen rótulos que incluyen varias veces la misma letra, como XXX; también, cuando se utilizan caracteres que se pueden confundir: la letra O y el dígito 0, la letra I y el dígito 1, o la letra S y el dígito 5.
- Definir rótulos que no se confundan con otros. Por ejemplo, puede producirse confusión entre un rótulo XXXX y otro XXYX.

- **Código de operación:**

Cada una de las operaciones básicas que puede ejecutar el microcontrolador se identifica mediante un código simbólico de entre 2 y 6 letras, denominado código de operación.

El código de operación debe seguir al campo de rótulo y estar separado de éste por, al menos, un blanco. Si no existe rótulo –es decir, cuando el código de operación es el primer campo de una sentencia–, se debe dejar, como mínimo, un blanco a la izquierda.

Por ejemplo, los códigos simbólicos para las operaciones de suma, resta, multiplicación y división son, respectivamente, ADD, SUB, MUL y DIV.

El compilador utiliza una tabla interna para convertir cada código simbólico a su equivalente absoluto.

- **Operando:**

El campo del operando le indica al microcontrolador dónde se encuentran los datos a ser procesados y cómo se accede a ellos.

Una instrucción puede tener ninguno, uno o dos operandos, dependiendo del tipo de instrucción que sea. Cuando una operación incluye dos operandos, éstos deben estar separados entre sí por una coma.

En una instrucción con dos operandos, el primero es el operando fuente y el segundo es el operando destino[6]. Por ejemplo, en la instrucción MOV PA,PB, el contenido del puertoA (operando fuente) se copia en el puertoB (operando destino).

Los operandos pueden ser de los siguientes tipos:

---

[6] *En Intel esta regla se invierte.*

reg8 : registro de 8 bits
reg16: registro de 16 bits
reg : registro de 8 o 16 bits
mem8 : byte (8 bits)
mem16: palabra (16 bits)
mem : byte o palabra (8 o 16 bits)
val8 : valor inmediato que puede almacenarse en 8 bits
val16: valor inmediato que puede almacenarse en 16 bits
val : valor inmediato que puede almacenarse en 8 o 16 bits

La manera en que se accede a los operandos es a través de los **modos de direccionamiento.** Los modos de direccionamiento son medios para permitir especificar la ubicación de los operandos; cada operando incluye, siempre, un modo de direccionamiento.

En unas páginas más describimos cada uno de los modos de direccionamiento provistos en el macro-assembler.

- **Comentario:**

El campo de comentario es utilizado para describir la operación efectuada por una instrucción. Siempre debe ir precedido por un punto y coma (;).

En general, es muy útil el uso del comentario, especialmente si se describe qué hace la instrucción en el contexto del programa. Por ejemplo:

MOV #$00,PB ;Inicializa en cero el puertoB

es mucho más claro de entender, en el contexto de un programa, que:

MOV #$00,PB ;Coloca 0 en PB

En un programa fuente, se pueden incluir líneas solamente de comentarios. Esto permite, por ejemplo, describir un bloque completo de programa o bien servir de separador entre dos bloques.

El compilador ignora los comentarios cuando genera el módulo objeto; solamente en el listado del programa fuente, generado por el compilador, aparecen los comentarios.

El uso de comentarios en un programa permite tener una importante documentación acerca de él, que es muy útil en caso de, por ejemplo, tener que modificarlo.

A diferencia de las instrucciones, las pseudo-ops –que se utilizan para fijar segmentos y procedimientos, definir símbolos, reservar memoria para almacenamiento temporario, etc.– no generan código objeto.

La sintaxis usada para escribir sentencias con pseudo-ops es, esencialmente, la misma que la usada para escribir instrucciones.

El formato general es el siguiente:

[Rótulo:] Código de Pseudo-op [Operando] [;Comentario]

El significado de los campos es el mismo que para las instrucciones. La única diferencia de importancia es que, para una pseudo-op, el campo de operando puede incluir más de 2 operados.

Por ejemplo, la sentencia:

DB 0FFH,0FFH,0FFH,0FFH,0FFH

le indica al ensamblador que cargue, en las cinco posiciones siguientes de memoria, el valor hexadecimal FF.

DS n or RMB n	Define reserva de almacenamiento (Storage), n = número o etiqueta, reserva n bytes.
DB m or FCB m	Define almacenamiento de un valor de un Byte (storage), m = label, número o string. Strings genera código ASCII para múltiple bytes. Números y etiquetas define sólo un byte por parámetro. Múltiples parámetros son seguidos y separados por comas.
DW n or FDB n	Define almacenamiento de un valor de dos bytes (Word storage), n=label, number o string. Los dos bytes están dados por los bytes siguientes. Múltiples parámetros son seguidos y separados por comas
label: EQU n	Setea el origen del valor del numero o label n. Todas las líneas ensambladas luego de esta sentencia son ubicadas en memoria seguido a partir de n.

Ejemplo:

```
VAR   DS 1              ;ubica 1 byte de memora para variable VAR
      DB 1,2,3,4,5      ;Setea un array de constantes en memoria
      DW vector         ;coloca una constante tipo word 'vector' en memoria

LBL1: EQU $100          ;Setea etiqueta LBL1 en el valor $100
      ORG $800          ;ensamblar código desde dirección $800
```

En un programa assembler se pueden escribir distintos tipos de **constantes**. La forma de indicar constantes es la siguiente:

Binarias: Es una secuencia de unos y ceros, seguida por la letra Q. Por ejemplo, 10111010Q, o precedida por el símbolo **%** en algunos casos seguida por la letra B
Decimales: Es una secuencia de los dígitos entre 0 y 9, con o sin la letra D o T. Por ejemplo, 129D o, bien 129, 129T o precedida por el símbolo **!**.
Hexadecimales: Es una secuencia de los dígitos entre 0 y 9 y de las letras A hasta F, seguida por la letra H o precedida por el símbolo $. El primer carácter siempre debe ser un dígito entre 0 y 9, para permitir la diferenciación de números hexadecimales con nombres de variables y símbolos. Por ejemplo, 0E23H (el 0 como primer dígito indica que el número es E23H).
De caracteres: Es una secuencia de letras, números o símbolos encerrada entre apóstrofes o comillas. El uso de ambos, apóstrofes y comillas, brinda la facilidad de poder definir como constante un mensaje encerrado entre comillas. Por ejemplo, “Dato incorrecto” o “Escriba FINAL para terminar el proceso”.

Las constantes numéricas pueden ser, también, números negativos. Si el número es un valor decimal, debe simplemente colocarse el signo menos delante de la constan-

te; en cambio, si es un valor binario, octal o hexadecimal, la constante debe ser escrita en notación complemento a 2.

Por ejemplo:

11100000B y 0E0H son las formas binaria y hexadecimal del decimal -32.
10010111Q = %10010111 = 97H = $97

db 'este es un string'

Base	Prefix	Suffix
02	%	Q
10	!	T
16	$	H

En todos los ejemplos vistos hasta aquí, los operandos utilizados fueron registros, constantes, variables y rótulos. Pero, también, es posible utilizar expresiones como operadores.

- **Operadores:**

Los operadores son modificadores que se usan en el campo de operando de las sentencias assembler (instrucciones y pseudo-ops); en una misma sentencia se pueden utilizar varios operadores y combinaciones de ellos.

Por ejemplo, en la instrucción:

MOV ADR,ram+2

el operando fuente es una expresión aritmética.

Es posible reconocer cinco diferentes tipos de operadores:

1. aritméticos,
2. lógicos,
3. relacionales,
4. de retorno de valores y
5. de atributos[7].

Existe, además, un conjunto de operadores para macros.

Veamos cada uno de estos tipos.

1. Operadores aritméticos:

Combinan operandos numéricos y producen un resultado numérico. Los operadores aritméticos más comunes son los de suma (+), resta (-), multiplicación (*) y división (/). Por ejemplo:

MOV ADR,ram+2

[7] *No vamos a ocuparnos de estos dos últimos. Nos referimos a los operadores relacionales que son utilizados comúnmente en ensambladores de microprocesadores de las PC y no en los ensambladores de los microcontroladores.*

*	multiplicación
/	división
\	especial división
+	adición
-	sustracción
<	rotar izquierda
>	rotar derecha
%	resto luego de división
&	operador and
|	operador or
^	operador xor

2. Operadores lógicos:

Los operadores lógicos son usados para manipular valores binarios. Sin embargo, los operadores lógicos son más útiles para manejar bits individuales que grupos de bits.

- El operador **AND** es útil para filtrar, enmascarar o extraer ciertos bits. Su funcionamiento consiste en establecer el bit de resultado en 1, por cada posición en que los respectivos bits de los operandos sean 1; en cualquier otra combinación de bits, AND fija el resultado en 0.

  Por ejemplo:
  00110100B AND 11010111B produce el resultado 00010100B.

- El operador **OR** establece el bit de resultado en 1, por cada posición de los operandos en la cual uno o ambos contienen un 1; en las posiciones en que ambos operandos tienen 0, el bit de resultado es 0.

  En el ejemplo anterior:
  00110100B OR 11010111B
  el resultado será 11110111B.

- El operador **XOR** –OR exclusivo– establece el bit de resultado en 1 por cada posición de los operandos en la que ambos contengan bits distintos (un operando con 1 y el otro con 0); en las posiciones en que ambos operandos tienen el mismo bit (1 o 0), el bit de resultado es 0.

  En el mismo ejemplo:
  00110100B XOR 11010111B
  el resultado será 11100011.

La siguiente es una tabla con los resultados de los operadores AND, OR y XOR para las distintas combinaciones de bits:

OP 1	OP 2	AND	OR	XOR
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

- El último operador lógico, **NOT**, trabaja sólo con un operando y su efecto es el de invertir cada uno de los bits del operando.

  Por ejemplo:
  NOT 01101001B
  produce el resultado 10010110B.

El microcontrolador tiene, también, instrucciones lógicas AND, OR, XOR y NOT; pero, ellas no deben ser confundidas con los operadores lógicos.

Mientras que las instrucciones lógicas actúan en tiempo de ejecución, los operadores lógicos actúan a tiempo de ensamblado o compilación.

3. Operadores relacionales:

Los operadores relacionales comparan dos valores numéricos o dos direcciones de memoria del mismo segmento, y producen un resultado numérico.

Este resultado es:

- 0 si la comparación es falsa o
- FF si la comparación es verdadera.

En general, los operadores relacionales aparecen integrando expresiones en combinación con otros operadores. Su uso es bastante limitado, dado que sólo producen dos posibles resultados.

Los operadores relacionales son:

- EQ (igual),
- NE (no igual),
- LT (menor que),
- GT (mayor que),
- LE (menor o igual) y
- GE (mayor o igual).

No todos los ensambladores admiten este tipo de operadores.

Desarrollo de un programa en assembler

El desarrollo de un programa en assembler requiere de una serie de pasos, desde que se escribe hasta que está en condiciones de ser ejecutado. Esos pasos son, en general, los mismos que deben efectuarse para cualquier programa en cualquier lenguaje:

- diseñarlo,
- escribirlo,
- compilarlo,
- linkeditarlo,
- depurarlo y
- ejecutarlo.

La **escritura** de un programa fuente se hace a través de un programa editor, que permite escribir las líneas de sentencias fuente. Para ello, se utiliza cualquier editor de líneas o procesador de la palabra disponible para los PC, tal como *Wordpad, Microsoft Word, EDLIN*, etc. En particular, veremos aquí el *Winide*®, que es el editor incluido con el sistema contenedor o entorno de trabajo del microcontrolador que utilizaremos.

El procedimiento de **compilación** (generación del programa objeto) se efectúa utilizando el compilador del assembler, llamado CASM08, o bien el compilador del macroassembler,

El procedimiento de **linkedición** o vinculación (generación del programa ejecutable) se efectúa con el utilitario LINK, que, en este entorno, está integrado al ensamblador.

La **depuración** de un programa es un procedimiento que consiste en seguirlo durante su ejecución para detectar y corregir todos los errores que se produzcan. En particular, usaremos el utilitario ICD08 –*in circuit debugger*– para ejecutar un programa compilado y linkeditado, de modo de observar los cambios que se producen en las distintas posiciones de memoria y para, eventualmente, efectuar las correcciones que sean necesarias.

Un programa, ya compilado, linkeditado y depurado, se puede **ejecutar** a través de un simulador o grabarlo en el propio chip. Para programas assembler, se utiliza la extensión .ASM.

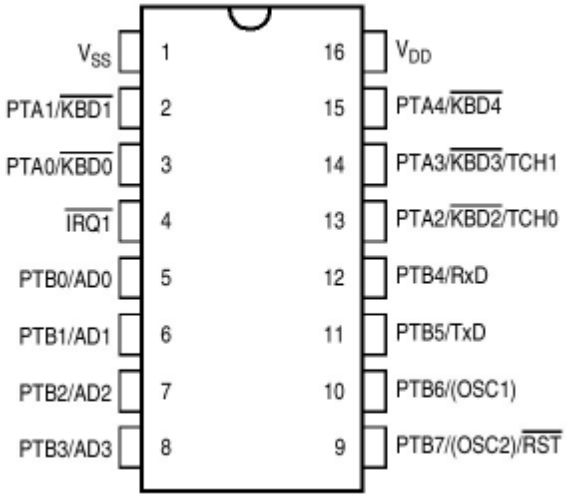
- **Un programa ejemplo:**

En un microcontrolador hay, usualmente, sólo un programa, el que atiende una aplicación específica de control. La CPU MC68HC08 reconoce alrededor de 119 instrucciones diferentes; no obstante, ellas son representativas del conjunto de instrucciones de un sistema de computadora.

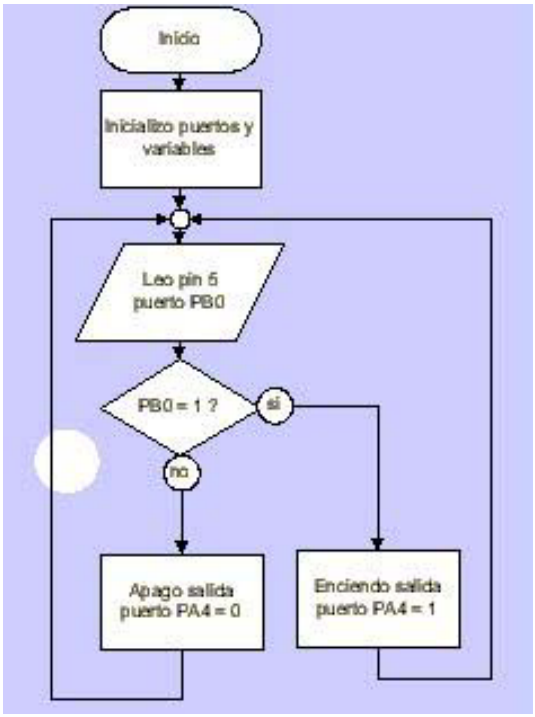
Este modesto sistema de computadora resulta ser un buen modelo para aprender los fundamentos de la operación de una computadora, puesto que es posible conocer con exactitud qué sucede en cada pequeña etapa en que la CPU ejecuta un programa.

Imaginemos que queremos que lo que sucede en la entrada 0 del puerto B (pin5) sea copiado en la salida 4 del puerto A (pin15). Como ejemplo, pensemos que tenemos un interruptor en dicha entrada y que acciona una lámpara al presionarlo.

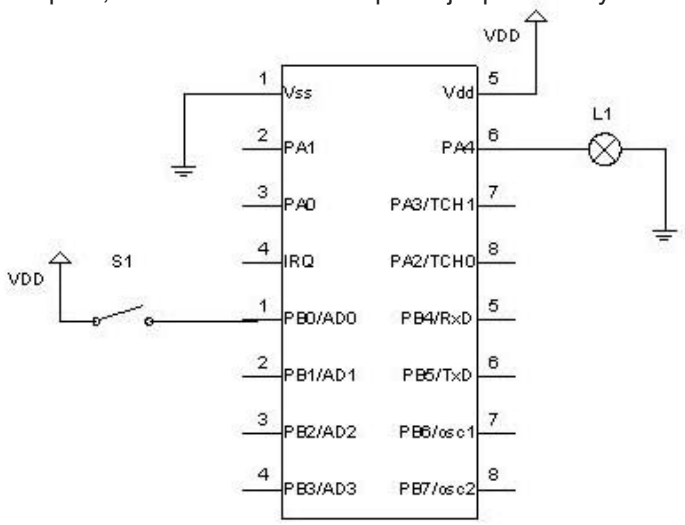
Nuestro microcontrolador tiene el siguiente pinout:



Veamos el diagrama de flujo de la tarea a realizar:



Veamos el circuito de una forma esquemática (faltan transistores de salida para poder comandar la lámpara, así como resistencias para fijar potencial y limitar corriente):



Así se vería el programa:

- Programa ejemplo 68HC908KX8
- Copia el estado de pin5 (PB0) y lo coloca en pin15 (PA4)

```
PA          EQU   $0000       ;Puerto A
PB          EQU   $0001       ;Puerto B
DDRA        EQU   $0004       ;Registro de dirección de puerto A
DDRB        EQU   $0005       ;Registro de dirección de puerto B

CONFIG1     EQU   $001F       ;Registros de configuración
CONFIG2     EQU   $001E

RAM         EQU   $0040       ;comienzo de la ram
ROM         EQU   $EC00       ;comienzo de la rom
RESET       EQU   $FFFE       ;vector de reset

            ORG   RAM
mem1        DS    4           ;reserva de memoria (no se utiliza; está a modo de ejemplo)
```

```
            ORG   ROM
inicio
            MOV   #$01,CONFIG1;configuración
            CLR   PA
            CLR   PB
            MOV   #$00,DDRB    ;puerto B en entrada
            MOV   #$FF,DDRA    ;puerto A en salida

tarea
            BRSET 0,PB,enciendo  ;leo estado de PB0 pin5 si está
apago                            ;en uno enciendo PA4 pin15 sino apago
            BCLR  4,PA
            BRA   tarea
enciendo
            BSET  4,PA
            BRA   tarea

            ORG   RESET
            DW    inicio         ; Reset Vector

            END
```

La **directiva ORG** seguida de una posición de memoria, indica al ensamblador dónde debe situar el siguiente fragmento de código. También es recomendable incluirla en todo programa, como mínimo, antes de la primera instrucción (Vamos a describir los casos de direcciones especiales un poco más adelante).

Antes de comenzar a escribir instrucciones máquina, debemos definir la dirección de la memoria de programa a partir de la cual se desea comenzar a cargar el programa. Para ello se emplea la directiva ORG. En los μC de la familia de Motorola HC08, el origen del programa se define de acuerdo a cuánta ROM tiene; en nuestro caso, se pone en la dirección $EC00 porque es donde comienza a ejecutarse el programa después de hacer un reset.

Definimos el origen de la siguiente manera:

```
ORG         $EC00          ;Inicio de programa
```

La **directiva END** es imprescindible en algunos casos e indica al ensamblador el final del programa.

El "**;**" es empleado a modo de comando REM; es decir, se sobreentiende que lo que le sigue es un comentario.

El ensamblador exige una cierta tabulación mínima de sus distintos elementos. De este modo, la definición de variables podrá escribirse en la primera columna de cualquier línea, mientras que las directivas e instrucciones deberán ir en la segunda columna, como mínimo. Las tabulaciones características son las empleadas por nosotros, ya que, aunque no son imprescindibles, clarifican la lectura del programa.

Las posiciones de la memoria de datos se utilizan para guardar operandos y resultados, además de almacenar registros especiales. Para que al programador le sea más sencillo confeccionar el programa, en lugar de hacer referencia a las posiciones de la memoria donde se encuentran los datos que va a emplear, a cada una de estas posiciones les asocia un nombre.

La **directiva EQU** relaciona un nombre con la dirección que se asigna; así, el programador trabaja con nombres y el compilador traduce automáticamente éstos a las direcciones correspondientes. Por ejemplo, el registro que contiene la información del puerto A se encuentra en la dirección $0000, el puerto B en $0001, etc. Si queremos emplear nombres de variables para estas direcciones de memoria, escribimos:

```
PortA     equ   $0000 ;La etiqueta "PortA" está asociada a la dirección $0000
PuertoB   equ   $0001 ;La etiqueta "PuertoB" está asociada a la dirección $0001
```

La directiva EQU se usa para asociar un valor binario con una "etiqueta" o "label". El valor puede ser tanto un valor de 8 bits de longitud como una dirección de 16 bits. Esta directiva no genera un código objeto.

Durante el proceso de ensamblado, el ensamblador debe mantener una lista cruzada de referencia donde almacena el equivalente binario de cada etiqueta. Cuando una etiqueta aparece en el programa fuente, el ensamblador mira en esta tabla cruzada de referencia para encontrar el equivalente binario. Cada directiva EQU genera una entrada en la tabla cruzada de referencia.

Un ensamblador lee el programa fuente dos veces:

- En la primera pasada, el ensamblador cuenta bytes del código objeto e, internamente, construye la tabla cruzada de referencia.
- En la segunda pasada, el ensamblador genera un archivo listado y/o el archivo objeto S-record.

Este arreglo de dos pasadas permite al programador generar etiquetas de referencia que son definidas más tarde en el programa.

Las directivas EQU deberían aparecer cerca del comienzo de un programa, antes de que las etiquetas allí definidas sean usadas por otros pasos del programa. Si el ensamblador encuentra una etiqueta antes de que ésta sea definida, no tiene elección, pero asume el peor caso de un valor de 16 bits de dirección, lo que causa que se use el modo de direccionamiento extendido en lugar de un modo más eficiente como el de direccionamiento directo. En otros casos, el modo de direccionamiento indexado con 16 bits de offset, podría ser usado por el ensamblador, en lugar de un direccionamiento indexado de 8 bits o un indexado sin offset mucho más eficiente.

Los argumentos para la **directiva formulario de constantes de un byte –FCB o DB–** son etiquetas o números separados por comas que pueden ser convertidos en un solo byte de datos. Cada byte especificado en una directiva FCB, genera un byte de código de máquina en el archivo de código objeto. Las directivas FCB se usan para definir constantes en un programa como, por ejemplo, una tabla de constantes, etc.

Los argumentos para la **directiva formulario de constantes de doble byte –FDB o DW–** son etiquetas o números separados por comas que pueden ser convertidos en valores de 16 bits de datos. Cada argumento especificado en una directiva FDB, genera dos bytes de código de máquina en el archivo de código objeto.

La **directiva reserva de memoria de un byte –RMB o DS–** se usa para reservar un lugar (asignar un espacio) en RAM para las variables de un programa. La directiva RMB no genera código objeto alguno; pero sí una entrada en la tabla cruzada interna del ensamblador. Esta forma es preferible a la del uso de EQU en la asignación de espacio en RAM, porque es muy común, en el desarrollo de un programa, borrar o agregar variables sobre la marcha. Si se usaran directivas EQU, se debe tener en cuenta cambiar varias sentencias después de remover una sola variable. Con la directiva RMB, el ensamblador asigna direcciones tanto como sean necesarias.

Algunos ensambladores, tal como el *P&E Microcomputers Systems®*, asumen que cualquier valor que no sea específicamente marcado será interpretado como un número hexadecimal.

La idea es simplificar el ingreso de información numérica por la eliminación de la necesidad del uso del símbolo $ antes de cada valor. Si se quiere que el ensamblador asuma que los valores no marcados sean valores decimales, es necesario usar la **directiva $BASE.**

$BASE 10T – Cambia la base numérica por default a decimal.

El uso de las mayúsculas y minúsculas en este código obedece a una serie de reglas o normas de estilo, comunes entre los programadores en ensamblador que, aunque no son obligatorias, facilitan la lectura del código fuente.

Un resumen de las reglas empleadas es el siguiente:

- Directivas del compilador en mayúsculas.
- Nombres de variables en minúsculas.
- Mnemónicos (instrucciones) en mayúsculas.
- Programa bien tabulado.

Para poder comenzar de un lugar conocido, se debe hacer el reset de la computadora. El reset obliga a los sistemas periféricos incluidos en el chip y a la lógica de I/O, a ir a condiciones conocidas y carga al contador de programa con una dirección de inicio conocida. El usuario especifica la posición de memoria de inicio deseada, colocando los bytes de mayor y menor peso de esta dirección en las posiciones de memoria del vector de reset ($FFFE y $FFFF en el MC68HC08).

El Reset Vector (vector de Reset) debe especificarse siempre y es una buena práctica especificar también los vectores de interrupciones aún si éstos no se usaran. Un error muy común en los programadores noveles (recién iniciados) es omitir los vectores de Reset e interrupciones, originando de esta forma que el ensamblador no los incluya en la memoria de programa del MCU. Este error genera que el MCU, al ser alimentado y salir de la etapa de *Power On Reset* –inicialización interna–, busque el vector de Reset con un contenido frecuente de $0000 (memoria de programa virgen del MCU tipo OTP ROM) o $FFFF (memoria de programa virgen del MCU tipo HC908 FLASH) que **no son posiciones de memoria de programa válidas** y dan como resultado que el MCU quede en un loop errático sin posibilidad de salir de él.

Compilado, genera el listado siguiente “nombre.**LST**” y un archivo de extensión “nombre.**S19**” que es el verdadero programa ejecutable que es enviado al microcontrolador cuando es programado:

```
                        1  * Programa ejemplo 68HC908KX8
                        2  * Copia estado de pin5 (PB0) y lo coloca en pin15  (PA4)
                        3
0000                    4  PA       EQU   $0000    ;Puerto A
0000                    5  PB       EQU   $0001    ;Puerto B
0000                    6  DDRA     EQU   $0004    ;Registro de dirección de puerto A
0000                    7  DDRB     EQU   $0005    ;Registro de dirección de puerto B
                        8
0000                    9  CONFIG1  EQU   $001F    ;Registros de configuración
0000                   10  CONFIG2  EQU   $001E
                       11
0000                   12  RAM      EQU   $0040
0000                   13  ROM      EQU   $EC00
0000                   14  RESET    EQU   $FFFE
                       15
                       16
0040                   17           ORG   RAM
0040                   18  mem1     DS    4                ;reserva de memoria
                       19
```

```
EC00                20              ORG    ROM
                    21  inicio
EC00 [04] 6E011F    22              MOV    #$01,CONFIG1        ;configuración
EC03 [03] 3F00      23              CLR    PA
EC05 [03] 3F01      24              CLR    PB
EC07 [04] 6E0005    25              MOV    #$00,DDRB           ;puerto B en entrada
EC0A [04] 6EFF04    26              MOV    #$FF,DDRA           ;puerto A en salida
                    27
                    28  tarea
EC0D [05] 000104    29              BRSET  0,PB,enciendo       ;leo estado dePB0 pin5 si está
                    30  apago                                  ;en uno enciendo PA4 pin15 sino apago
EC10 [04] 1900      31              BCLR   4,PA
EC12 [03] 20F9      32              BRA    tarea
                    33  enciendo
EC14 [04] 1800      34              BSET   4,PA
EC16 [03] 20F5      35              BRA    tarea
                    36
FFFE                37              ORG    RESET
FFFE    EC00        38              DW     inicio          ; Reset Vector
                    39              END
```

El código ejecutable –que es el que se graba en la memoria flash del microcontrolador, que viene dentro del archivo objeto S– record–, es el siguiente:

```
S113EC006E011F3F003F016E00056EFF040001040A
S10BEC10190020F9180020F599
S105FFFEEC0011
S9030000FC
```

Archivo de código objeto

Nosotros aprendimos que la computadora espera que el programa sea una serie de valores de 8 bits en memoria. Nada tan alejado de ello en su aspecto, ya que nuestro programa fue escrito para personas. La versión que la computadora necesita cargar en su memoria es llamada **archivo de código objeto –*Object Code File*–.**

Para los microcontroladores Motorola, la forma más común de archivo de código objeto es el archivo S-Record[8].

El ensamblador puede generar en forma directa un archivo de listado y/o un archivo de código objeto.

Un archivo S-record es un archivo de texto que puede ser leído por un editor de texto o un procesador de palabras.

Usted no debería tratar de editar este tipo de archivo, porque la estructura y el contenido de este archivo son críticos para el buen funcionamiento.

Cada línea de un archivo S-record es un registro. Cada registro comienza con una S mayúscula seguida por un número de código desde 0 hasta 9. Los únicos números de código que son importantes para nosotros son S0, S1 y S9.

- S0 es un registro encabezado opcional –*header record*– que puede contener el nombre del archivo para beneficio del personal humano que necesita mantener dicho archivo.

[8] *Intelhex para Intel, "Hex".*

- S1 es un registro que contiene los datos principales del programa.
- S9 se usa para marcar el fin de un archivo S-record. Para el trabajo que estamos haciendo con microcontroladores de 8 bits, la información en un registro S9 no es importante; pero, un registro S9 siempre es requerido en el fin de nuestros archivos S-record.

Veamos cómo interpretarlo.

- Todos los números en un archivo S-record están en hexadecimal.
- Nosotros usaremos tipos de campos S0, S1 y S9 para nuestros archivos S-record (1° campo).
- El largo del campo es el número de pares de dígitos hexadecimales en el registro, excluyendo los campos de tipo y longitud.
- El campo de dirección es una dirección de 16 bits donde el primer byte de dato será almacenado en memoria.
- Cada par de dígitos hexadecimales en el campo de datos de código de máquina representa un valor de dato de 8 bits a ser almacenado en sucesivas posiciones de memoria.
- El campo de *Checksum* es un valor de 8 bits que representa el complemento a uno de la suma de todos los bytes en el archivo S-record a excepción de los campos de tipo y checksum respectivamente. Este Checksum se usa durante la carga de un archivo S-record para verificar que los datos están completos y correctos en cada registro.

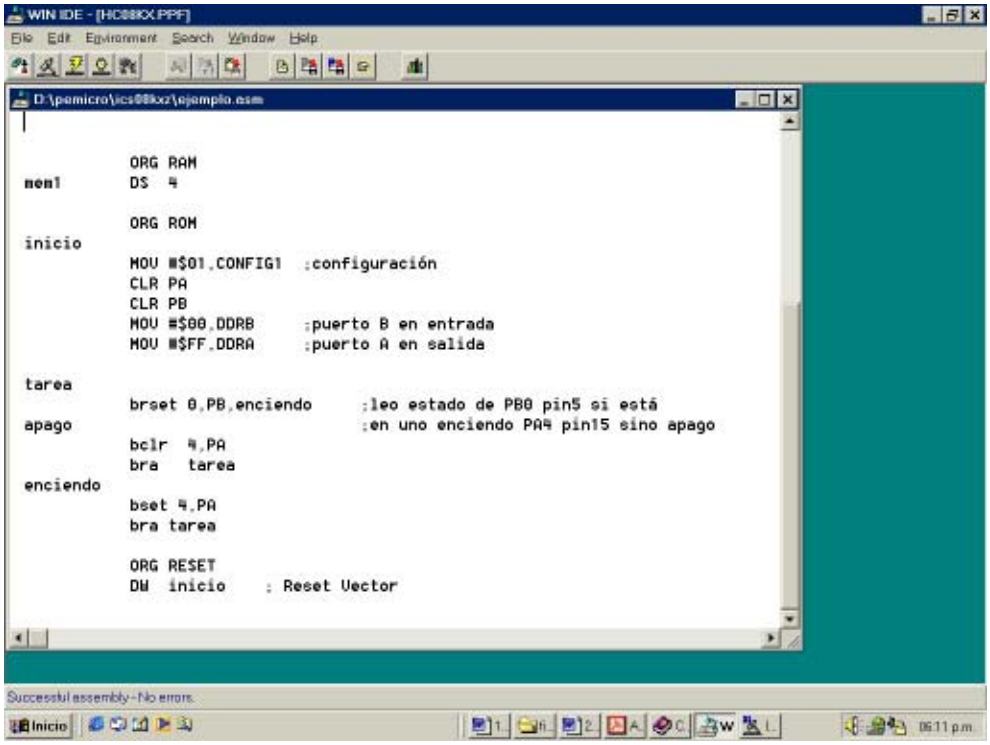
S1 indica código a ensamblar
S9 indica fin de archivo

```
S113EC006E011F3F003F016E00056EFF040001040A
S10BEC10190020F9180020F599
S105FFFEEC0011
S9030000 FC
```

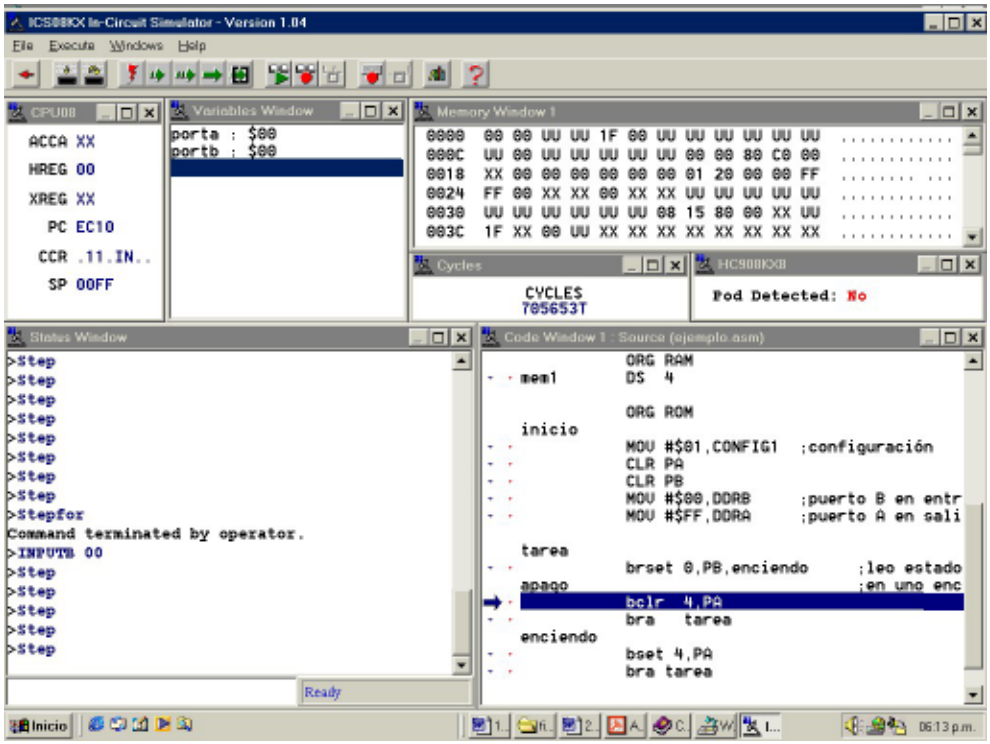
Por supuesto que éste está expresado en hexadecimal para ser enviado a la memoria con un programador o para ser simulado por un software depurador .

- Dirección de comienzo
- Datos en hexadecimal (código máquina del microcontrolador)
- Checksum
- Cantidad de Bytes por línea en hexadecimal

Observemos cómo se ve en el entorno de trabajo este programa. Primeramente, el editor, ensamblador y linkeador Winide para el MC68HC908KX8:



El entorno de simulación donde se depura el programa dado por ICS08KX –simulador en circuito–:



En las próximas unidades de trabajo veremos cómo resolvemos nuestro problema con todas estas herramientas. Ya tenemos las pautas para editar nuestro programa, que resolverá las situaciones planteadas; ahora, nos resta conocer las instrucciones posibles, y sus modos de utilización respecto al tratamiento de los datos y variables.