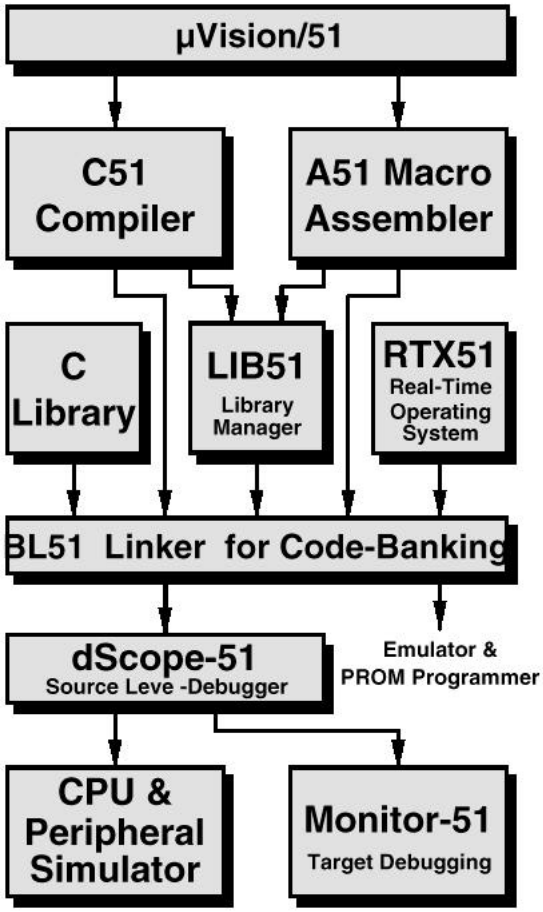
Universidad
de Huelva

Escuela Politécnica Superior
Universidad de Huelva

TERCER CURSO. ELECTRÓNICA DIGITAL

TÉCNICAS BÁSICAS DE PROGRAMACIÓN DEL 8051



Manuel Sánchez Raya
Versión 1.0
31 de Marzo de 2000

ÍNDICE

BIBLIOGRAFÍA

C and the 8051, Mc Graw-Hill

Apuntes de la asignatura Procesadores de Propósito General

Introducción a los Microcontroladores, Jose Adolfo González Vázquez Mc Graw-Hill

Apuntes de prácticas: Manual del compilador C51 de KEIL

Practicas con Microcontroladores de 8 bits. Aplicaciones Industriales. Javier Martínez Pérez, Mariano Barrón Ruiz. Mc Graw-Hill, 1993.

The Microcontroler Idea Book. MC Graw-Hill

TÉCNICAS BÁSICAS DE PROGRAMACIÓN DEL 8051

1.- Instrucciones máquina del 8051.

Comenzaremos dando una visión general de la programación en ensamblador del microcontrolador 8051. Incluso si no vamos a programar en lenguaje ensamblador, es necesario introducir las instrucciones que encontraremos si la búsqueda de errores en el código nos fuerza a observar el listado que genera el compilador.

Las tablas que siguen muestran todas las instrucciones para referencia posterior. Las siguientes definiciones son necesarias para entender las instrucciones:

- ***A***: (el acumulador o ACC) es el registro más relacionado con la ALU.
- ***#dato***: es un valor numérico precedido por el signo #, no una dirección de memoria. Es un dato numérico.
- ***#dato 16***: es una constante de 16 bits incluida en la instrucción.
- ***directo***: es la dirección de memoria donde se deben encontrar los datos utilizados por la instrucción, **NO** los datos si no una dirección interna entre 0 y 7FH (o 80H a FFH para registros de función especial).
- ***Rn***: se refiere al contenido de un registro.
- ***@Ri***: precedido por el signo @ indica que el registro es un puntero y se refiere al valor en memoria a donde apunta el registro. Solo R0 y R1 pueden usarse de esta forma.
- ***DPTR***: El puntero de datos, usado para direccionar datos fuera del chip y código con la instrucción MOVX o MOVC.
- ***PC***: el contador de programa, almacena la dirección de donde se obtendrá el siguiente byte de código.

1.1.- Movimiento de datos.

Las instrucciones más simples son las que mueven números entre los registros y varios tipos de memoria. Hay varias formas de hacerlo, denominadas ***modos de direccionamiento***. La instrucción MOV tiene varias formas dependiendo de donde venga el dato y a donde vaya.

MOV no destruye el dato de la fuente si no que lo copia al destino. Por ejemplo, no podemos mover datos de un registro a otro, pero una vez visto esto, para un banco de registros determinado cada registro también tiene una dirección directa, a la que podemos acceder. Las instrucciones de movimiento de bit están agrupadas con la boleanas.

Para referirnos a los registros de función especial se emplea el nombre, por lo que podemos escribir ***MOV TCON, #013H***. En ensamblador podemos sustituir números por nombres. en lugar de usar direcciones directas podemos emplear etiquetas mediante EQU: ***RELAY EQU 045H*** y más adelante en el programa usar: ***MOV RELAY, #0FFH***.

Acumulador/Registro:

```
MOV A,R7      ; copia el contenido de R7 al acumulador
MOV R1, A     ; copia el contenido del acumulador a R7
```

Acumulador/Directo:

```
MOV A, 22H    ; copia el contenido de la dir. 22H al acumulador
MOV 03H, A    ; acumulador => 03H (R3 en banco de registros 0)
MOV A, 1BH    ; 1BH (R3 en banco de registros 3) => acumulador
```

Acumulador/Datos:

```
MOV A, #22    ; el número 22 (16H) pasa al acumulador
MOV A, #7EH   ; el número 7EH => acumulador
```

Registro/Datos:

```
MOV R1, #5FH  ; el número 5FH => R1
```

Acumulador/Indirecto : Esto se puede hacer solo con R0 o R1 como punteros.

```
MOV R1, #4BH  ; el número 4BH => R1
MOV A, @R1    ; contenido de donde apunte R1 (4BH) =>
acumulador
MOV @R0, A    ; acumulador => donde R0 apunte
```

Registro/Directo

```
MOV R3, 7FH   ; contenido de la dirección 7FH => R3
MOV 6EH, R2   ; R2 => dirección 6EH
```

Directo/Directo

```
MOV 1FH, 7EH  ; contenido de dirección 7EH => dirección 1FH
```

Directo/Indirecto

```
MOV R3, @R1   ; contenido de dirección apuntada por R1 => R3
MOV @R0, R6   ; R6 => dirección apuntada por R0
```

Directo/Datos

```
MOV R7, #01   ; el número 01 => R7
```

Indirecto/Datos

```
MOV   @R0, #7FH        ; el número 7FH => donde apunte R0
```

Puntero de datos/Datos : Esta es una instrucción de carga de 2 bytes que toma un número de 16 bits y lo pone en el puntero (DPTR) que se usa para acceder la memoria fuera del chip (mediante MOVX)

```
MOV DPTR, #0FFC0H      ; el número C0H => DPL; el número FFH => DPH
```

MOVC : Permite el acceso al espacio de código (normalmente EPROM) que es por definición solo lectura. La instrucción MOVC es bastante útil para extraer datos de tablas basadas en ROM. Es la única instrucción que usa el modo de direccionamiento indexado.

```
MOVC A,@A+DPTR ; contenido de dirección ROM apuntada por la suma de DPTR y
               ACC => ACC (ojo: destruye el valor del acumulador)
MOVC A, @A+PC  ; contenido de dirección de código ACC-bytes desde el código
               apuntado por el valor actual del contador de programa => ACC
```

MOVX : Esta es la principal instrucción usada en programas largos porque con las instrucciones de direccionamiento directo a RAM externa no pueden trabajar con más de 256 bytes de almacenamiento.

```
MOV DPTR, #021C5H ; 21H => DPH, C5H => DPL
MOVX A, @DPTR     ; contenido RAM externa en dirección apuntada por DPTR
                  => ACC
MOVX @DPRT, A  ; ACC => dirección apuntada por DPTR
MOVX A, @R0    ; contenido de la dirección externa de 8 bits => ACC
MOVX @R1, A    ; ACC => dirección externa de 8 bits
```

XCH : A diferencia de la instrucción MOV que copia de un lugar a otro, XCH intercambia los dos bytes. Esto es bastante útil con las instrucciones que deben realizarse a través del acumulador.

Acumulador/Registro

```
XCH A,R4       ; se intercambian el contenido de R4 y ACC
```

Acumulador/Directo

```
XCH A,1EH      ; se intercambian contenido de dirección 1EH y ACC
```

Acumulador/Indirecto

```
MOV R1, #05BH  ; coloca 5BH en R1
XCH A, @R1     ; intercambia contenido de dirección apuntada por
               R1 y ACC
```

1.2.- La Pila.

Hay otra forma de almacenar datos, la pila. Hasta ahora hemos puesto un byte de información en una dirección especifica. Con una pila, los valores se almacenan en direcciones sucesivas direccionadas por el puntero de pila. Cada vez que introducimos un valor en la pila, el puntero de pila se decrementa en uno. Constituye un buffer primero en entrar, ultimo en salir.

Con la familia 8051, la pila se localiza en la RAM interna. Dos instrucciones introducen bytes en la pila. Primero, la instrucción PUSH introduce un byte cada vez que se usa. Segundo, cada instrucción CALL y cada interrupción hardware introduce el valor actual del contador de programa (2 bytes) en la pila.

PUSH : El contenido de cualquier de las direcciones directas puede introducirse incluyendo los SFR. El puntero de pila apunta al valor de la cima, no el espacio vacio sobre la pila. Es posible guardar ACC, B, PSW y los registros de control hardware. No es posible introducir R0 a R7 en la pila por nombre porque se espera que cambiemos el banco de registros modificando los dos bits del PSW.

```
MOV SP, #09CH  ; pone el puntero de pila apuntando a la dirección 9CH
PUSH B         ; incrementa el puntero de pila hasta 9DH y pone el
               contenido del registro B (dirección directa F0H) en la pila
               en la dirección de memoria interna 9DH
```

POP : Esta el la instrucción inversa de PUSH. Recordemos que restaurar tras multiples PUSH debe hacerse en orden inverso. Push/pop no en orden pueden servir para intercambiar valores fácilmente.

```
POP PSW   ; cima de la pila => PSW (dirección directa D0H) y SP
          decrementado en una unidad
```

1.3.- Instrucciones de Salto.

Las instrucciones de salto encauzan la secuencia de ejecución del programa. Hay tres métodos de direccionamiento para las instrucciones de salto y llamada a subprograma. El salto corto (SJMP) cubre un rango de direcciones de 128 bytes atrás a 127 bytes delante de la dirección de la próxima instrucción. Los saltos absolutos, AJMP y ACALL proporcionan los 11 bits menos significativos de la dirección de 16 bits necesaria y mantienen los 5 bits de la siguiente instrucción del programa. Esto fuerza al destino a estar en el mismo bloque de 2K que la instrucción CALL. Finalmente, tenemos los saltos largos, LCALL o LJMP que incluyen la dirección absoluta y completa de 16 bits del destino.

1.3.1.- Salto incondicional

Lo veremos usando etiquetas en lugar de números puesto que esta es la forma de escribir código ensamblador legible, pero podemos poner una dirección de código numérica especifica usando # (como #215EH). La instrucción provoca que el contador de programa (registro PC) cambie de apuntar a la próxima instrucción tras la instrucción de salto a apuntar a la nueva dirección. La siguiente instrucción a ejecutar es, por tanto, la de la nueva dirección.

```
AJMP SUB        ; en el mismo bloque de 2K
LJMP POINTA     ; en cualquier sitio del código
SJMP LOOP       ; relativo: +127 a –128
JMP @A+DPTR     ; esta instrucción puede usarse para hacer un salto a
                múltiples direcciones, pero no se usa mucho. Incluso con una
                tabla de saltos el valor de ACC debe multiplicarse por dos o
                tres para encontrar la instrucción de salto.
```

1.3.2.- Salto condicional.

Esta instrucción realiza la prueba y hace un salto corto o en caso contrario sigue con la siguiente instrucción.

```
JZ POINTX       ; salta si ACC es todo cero
JNZ POINTY      ; salta si cualquier bit de ACC es diferente de cero
JC POINTZ       ; salta si el indicador de acarreo es 1 (puesto)
JNC POINTZ
JB P3.5,POINTA ; salta si el bit del puerto es 1 (aquí en puerto 3)
JNB 06EH, POINTB  ; salta si bit en zona de RAM direccionable a bit es 0
JBC 22.3, POINTB  ; esto también pone a cero el bit (22.3= bit 19 = 13H)
```

1.3.3.- Comparaciones.

Comparar es para la ALU una cuestión de restar (sumar el inverso) y comprobar el acarreo. Hay dispositivos de comparación que devuelven tres resultados sobre la magnitud relativa de dos números binarios (igual, mayor que o menor que), pero la ALU del 8051 evita esto usando el acarreo de la sustracción.

CJNE : CJNE es comparar y saltar (corto) si no igual. No existen todas las combinaciones. Podemos comparar solo un registro con el acumulador usando la dirección directa del registro, que depende del banco de registros usado. No podemos, por ejemplo referirnos a R0 o R7, pero si usamos el banco de registros 0, nos referiremos como 00 o 07 (o 08 y 0FH si usamos el banco de registros 1).

```
CJNE A, 3EH, POINTZ    ; compara contenido de dirección 3EH
CJNE A, #10, POINTW    ; compara ACC con número 10
CJNE R5, #34, LOOP
CJNE @R1, #5, GOINGON  ; compara número 5 con contenido RAM interna
                       donde apunta registro R1
```

El flag de carry se ve alterado por esta instrucción. CJNE puede ser la primera parte de una prueba mayor que/igual que/menor que. Si el segundo número de la comparación es mayor, no habrá carry. Si el segundo número es igual o menor que el primero, habrá carry en la salida. Por lo que es posible comprobar y saltar a tres direcciones de la siguiente forma:

```
      CJNE A, X, NEXT_TEST
      JMP IGUAL
NEXT_TEST:  JNC X_MENOR
      JMP X_MAYOR
IGUAL:      ......
      JMP FIN
X_MENOR:    .......
      JMP FIN
X_MAYOR:    .......
      JMP FIN
FIN:  ......
```

Con esta combinación, podemos saltar a uno de tres lugares diferentes dependiendo del valor relativo del acumulador frente la variable X.

DJNZ : Esta es una instrucción muy útil para bucles iterativos donde el número de veces que se realiza se fija fuera del bucle y vamos decrementando hasta llegar a cero. Por ejemplo, podemos tener un bucle para hacer avanzar un motor paso a paso diez veces, usando una llamada a subrutina:

```
      MOV 03FH, #10     ; fijamos la cuenta a 10
LOOP1:
      CALL STEP
      DJNZ 3FH, LOOP1   ; decrementamos el contenido dirección 3FH (donde
                        tenemos el número 10, luego la llamada anterior
                        se ejecutará diez veces.
```

1.3.4.- Llamadas.

Una llamada hace que el programa cambie la ejecución a una subrutina y luego vuelva cuando esta finalice. Si podemos saltar a un trozo de código desde varios sitios, no hay forma de saber que instrucción de salto origino que se ejecutara este trozo y no habría forma de volver atrás tras ejecutarlo. CALLs pueden provenir de múltiples lugares. El contador de programa se guarda en la pila al hacer la llamada, y la instrucción RET recupera el contador de programa de la pila. El programa continua con la instrucción justo siguiente tras la instrucción CALL que produjo el salto a la subrutina. Aunque hay diferentes tipos de llamada, es el ensamblador el que escoge la apropiada en cada momento.

ACALL : ACALL llama de forma incondicional a una rutina localizada en la dirección indicada. El contador de programa, ya puesto a la dirección de la siguiente instrucción se introduce en la pila (byte bajo primero), y se ajusta el puntero de pila. Para esta instrucción la dirección de la subrutina se calcula combinando los cinco bits de más peso del PC incrementado con los bits 7 a 5 del código de operación de la instrucción y con el segundo byte de la instrucción. La dirección de la subrutina debe comenzar en el mismo bloque de 2K del espacio de código que la siguiente instrucción al ACALL. Ningún indicador se ve afectado.

```
ACALL STEPRUTINA    ; si la siguiente línea de código comienza en 201DH,
                    STEPRUTINA comienza en 2500H, y si el SP está
                    inicialmente en 95H, esta instrucción deja el puntero
                    de pila con 97H, 1DH en la dirección 96H, 20H en 97H,
                    y deja el contador de programa con 2500H
```

Las direcciones se colocan de forma simbólica como:

```
      ACALL STEPRUTINA
      ..........
STEPRUTINA:   ..........
      ..........
      RET
```

LCALL : La llamada larga puede estar en cualquier posición del espacio de código. Las funciones de la pila y contador de programa son las misma que en el caso anterior, con la diferencia que la dirección completa de 16 bits se encuentra en el segundo y tercer bytes de la instrucción.

```
LCALL DISPLAY ; si la siguiente línea de código comienza en 23F1H,
              DISPLAY comienza en 24AFH, y el SP está en 38H, esta
              instrucción deja el puntero de pila con 3AH, deja F1 en la
              dirección 39H, deja 23H en 3AH, y deja el contador de
              programa con 24AFH.
```

RET : La instrucción de retorno pone los dos valores de la cima de la pila en el contador de programa. Permite que continúe el flujo del programa principal después que ha terminado la subrutina.

```
RET  ; si la situación es la del ACALL anterior, esto devolvería el SP a
     95H, y devuelve el contador de programa a 201DH
```

RETI : La instrucción RETI funciona como RET con la característica adicional que pone a cero de forma automática el hardware que permite atender más interrupciones del mismo nivel de prioridad. O sea que se usa para retornar de una interrupción.

NOP : Esta instrucción no hace nada, pero es muy utilizada por el tiempo que tarda en ejecutarse para producir retardos de tiempo.

1.4.- Instrucciones Lógicas.

ANL: Realiza un AND lógico produce un 1 solo en las posiciones de bit donde ambos bits valgan 1. Esta instrucción deja el resultado en el acumulador.

```
ANL A,R6       ; ACC(1101 1000) con R6(1000 1111) da en ACC(1000 1000)
ANL A, 25H     ; AND ACC con el contenido de la dirección 25H
ANL A, @R1     ; AND ACC con contenido de dirección externa apuntada por
R1
ANL A, #03H    ; AND ACC con el número 3
```

```
ANL 25H, A      ; lo mismo que ANL A, 25H
```

ORL: Hacer un OR lógico pone 1 en lugares donde cualquier bit valga 1.

```
ORL   A, R6     ; ACC(1101 1000) con R6(1000 1111) da en ACC(1101 1111)
ORL   A, 25H
ORL   A, @R1
ORL   A, #03H
ORL   25H, A
```

XRL: realiza el OR eXclusivo dando un 1 en posiciones si uno y solo uno de los bits es 1, si ambos son 1 pone 0.

```
XRL   A, R6     ; ACC(1101 1000) con R6(1000 1111) da en ACC(0101 0111)
XRL   A, 25H
XRL   A, @R1
XRL   A, #03H
XRL   25H, A
```

CPL : realiza el complemento lógico que cambia ceros por unos y unos por ceros.

```
CPL   A         ; con ACC(1101 1000) deja en ACC(0010 0111)
```

CLR : esto pone todos los bits a cero.

```
CLR   A         ; pone a cero los 8 bits del acumulador
```

1.4.1.- Rotaciones.

Además de sumar e invertir hay más formas de mover datos a izquierda y derecha, denominadas desplazamiento o rotaciones. Un valor de 0001 desplazado a la izquierda es 0010. De la misma forma, 1010 desplazado a derecha es 0101. En matemática binaria, un desplazamiento es multiplicar o dividir por dos.

RR, RL, RRC, RLC : Con estas instrucciones, el acumulador se desplaza un lugar a la izquierda (hacia MSB) o hacia la derecha (hacia LSB). si el acarreo está incluido el bit final va al acarreo y el acarreo va a entrar por el otro lado de la rotación.

```
RL A  ; desplazamiento de 8 bits hacia el MSB: 1011 1100 pasa a 0111 1001
RR A  ; desplazamiento a derecha, 1011 1100 pasa a 0101 1110
RRC A ; desplazamiento de 9 bits hacia LSB, el carry se copia al MSB: 1 1011
        1100 pasa a 0 1101 1110
RLC A ; desplazamiento de 9 bits a izquierdas, el carry va al LSB: 1 1011
        1100 pasa a 1 0111 1001
```

1.5.- Instrucciones de manipulación de bits.

Indicadores y sus usos: Los indicadores o banderas (flags) son variables de un solo bit que podemos manipular directamente o que cambian de forma indirecta debido a la ejecución de instrucciones. En el 8051, hay 128 bits posibles que podemos usar como variables de 1 bit así como bits dentro de bytes de los SFR.

Los flags direccionables de forma directa son útiles para marcar la ocurrencia de un evento. Tomemos por ejemplo un programa de semáforos; si alguien pulsa un botón de petición, podemos activar(set) (darle un valor de 1) un flag, y luego desactivar(clear) (dar un valor de 0) el flag después en el momento correcto del ciclo cuando el programa encienda la luz verde. La próxima vez en el ciclo de las luces, si nadie ha pulsado el botón desde el último ciclo, la luz no necesita encenderse de nuevo.

Los flags afectados individualmente resultan particularmente útiles para las operaciones matemáticas con varios bytes. Podemos sumar los dos bytes menos significativos sin el acarreo (ADD) y luego sumar los dos bytes siguientes con el acarreo (DAC) que se genero en la primera suma, hasta el número de bytes que queramos. Un ejemplo de suma de dos bytes debería ser:

```
; dos números en R6, R7 y 31H, 32H; resultado en R6, R7
MOV A, R6
ADD A, 31H     ; sin acarreo
MOV R6, A
MOV A, R7
ADDC A, 32H    ; con el acarreo generado en la primera suma
MOV R7, A
```

Las instrucciones SETB C y CLR C afectan de forma directa al flag de carry, mientras que las MUL y DIV las ponen a cero. De los flags afectados indirectamente, los más significativos son los tres bits C, OV y AC del PSW.

CLR (bit) : Esto pone a cero el bit seleccionado. Funciona con uno de los 128 bits del área direccionable 20H a 2FH y también con los SFR.

```
CLR C ; pone el bit de acarreo a 0
CLR ACC.7      ; pone el MSB del acumulador a 0
CLR P1.5       ; pone el tercer bit desde arriba del puerto 1 a
0
```

SETB : Pone el bit indicado a 1.

```
SETB C         ; pone el bit de acarreo a 1
SETB 20.3      ; pone el bit de memoria a 1 (dirección de byte
20H, dirección bit 03H)
SETB ACC.7     ; pone el msb del acumulador a 1
```

CPL (bit) : Complementa el bit (cambia 0 por 1 o 1 por 0).

```
CPL C ; invierte el bit de acarreo
CPL P3.5      ; invierte el tercero desde msb del puerto 3
```

ANL (bit) :

```
ANL C,ACC.5    ; AND acarreo y bit 5 del acumulador, resultado en carry
ANL C, /ACC.5 ; AND carry y complemento de bit 5 del acum.. resultado en
                carry, el acumulador no cambia
```

ORL (bit) :

```
ORL C, ACC.5   ; OR acarreo y bit 5 del acumulador, resultado en carry
ORL C, /ACC.5 ; OR carry y complemento de bit 5 del acum. resultado en
                carry, el acumulador no cambia
```

MOV (bit): Esta puede agruparse con las otras instrucciones MOV, pero la vemos aquí porque está orientada a bit.

```
MOV C, ACC.2  ; contenido del tercer bit del acumulador (un 1 ó 0) se
                copia en el bit de carry
MOV 20.3, C   ; el contenido del acarreo se copia en la dirección de RAM
                direccionable a bit 3 (20.3 es el cuarto bit del byte 20H
                que es el primer byte de la zona de RAM direccionable a
                nivel de bit)
```

1.6.- Instrucciones matemáticas.

La familia básica 8051 dispone de una capacidad matemática muy limitada. Se pueden encadenar instrucciones sobre byte simple en trozos de código para realizar tratamientos matemáticos más elaborados. Excepto el incremento y decremento, todas las operaciones aritméticas sobrescriben el contenido del acumulador con el nuevo resultado.

1.6.1.- Suma.

ADD : Esta instrucción no suma el bit de acarreo con el bit menos significativo, si no que produce un acarreo de salida. Es la primera instrucción de una operación con varios bytes, pero podemos poner a cero el bit de carry y usar ADDC a través de una secuencia de varios bytes.

```
ADD A, R5     ; suma contenido de R5 a ACC; resultado en ACC; overflow en
                carry
ADD A, 22     ; suma contenido de RAM interna 22(16H) al ACC: resultado en
                ACC, overflow en carry
ADD A, @R0    ; suma contenido de RAM interna apuntada por el contenido de
                R0 a ACC; resultado en ACC; overflow en carry
ADD A, #22    ; suma número 22(16H) al ACC; resultado en ACC; overflow en
                carry
```

ADDC : Esta instrucción incluye el bit de carry en la suma.

```
ADDC A, R5      ; suma contenido de R5 con acarreo a ACC; resultado en ACC;
                  overflow en carry
ADDC A, 22      ; suma contenido de RAM interna 22(16H) con acarreo al ACC:
                  resultado en ACC, overflow en carry
ADDC A, @R0     ; suma contenido de RAM interna apuntada por el contenido de
                  R0 con acarreo a ACC; resultado en ACC; overflow en carry
ADDC A, #22     ; suma número 22(16H) con acarreo al ACC; resultado en ACC;
                  overflow en carry
```

1.6.2.- Resta.

SUBB : El indicador de borrow es el inverso del acarreo obtenido de un sumador normal. No existe instrucción de resta sin el borrow, por lo que tendremos que ponerlo a cero, antes de comenzar la resta. El borrow indica que algo demasiado grande se restó y obtendremos una respuesta negativa en lugar de lo que parece un número positivo muy grande. Para la resta de múltiples bytes, comenzamos desde el byte de menor peso hasta el de mayor peso, borrows generados durante la operación son aceptables. La respuesta es positiva (y correcta) si no hay borrow cuando los bytes más significativos hayan sido restados. Recordar que estamos realizando matemática sin signo. Para usar números negativos tenemos que hacer algunos ajustes en el software. Operaciones con enteros con signo de 16 bits también se pueden realizar de forma directa, pero se sugiere emplear C.

```
SUBB A, R5      ; resta contenido de R5 con borrow a ACC; resultado en ACC;
                  overflow en carry
SUBB A, 22      ; resta contenido de RAM interna 22(16H) con borrow al ACC:
                  resultado en ACC, overflow en carry
SUBB A, @R0     ; resta contenido de RAM interna apuntada por el contenido
                  de R0 con borrow a ACC; resultado en ACC; overflow en carry
SUBB A, #22     ; resta número 22(16H) con borrow al ACC; resultado en ACC;
                  overflow en carry
```

1.6.3.- Otras operaciones.

INC y DEC : Estas son instrucciones simétricas excepto para el puntero de datos. Decrementar 0 o incrementar FFH genera FFH o 0 respectivamente.

```
INC A           DEC A
INC R2          DEC R5
INC 45H         DEC 3EH
INC @R0         DEC @R1
INC DPTR        ;no existe equivalente para decremento
```

MUL : Solo hay una multiplicación por hardware, que produce un resultado de 16 bits en el acumulador (byte bajo) y el registro B (byte alto). Si el producto excede 8 bits, el indicador de overflow se pone a uno. El indicador de acarreo siempre se pone a cero.

```
MUL AB          ; ACC se multiplica por el contenido del registro B;
                  resultado en ACC (byte bajo) y B (byte alto); por ejemplo
                  B=160, ACC=80, resultado es 12,800; B=32H, ACC=00H, el
                  acarreo se pone a cero, OV=1 indica que el resultado es
                  mayor de 8 bits (B almacena parte del resultado)
```

DIV : B divide el acumulador, con el resultado en ACC y el resto (no la fracción) en B. Los indicadores de acarreo y overflow siempre se ponen a cero. Ninguna operación permite extensiones multibyte ni matemática con signo, ni mucho menos operaciones en punto flotante.

```
DIV AB          ; ACC se divide por el contenido del registro B; ACC
                  almacena la parte entera del cociente mientras que B el
                  resto entero; por ejemplo 251 en ACC dividido por 18 da 13
                  en ACC, un resto de 17 en B y los flags de acarreo y
                  overflow se ponen a cero.
```

1.6.3.- Instrucciones Decimales.

XCHD : Esta instrucción intercambia el nibble de orden bajo del acumulador con el valor direccionado de forma indirecta. El nibble alto permanece en el sitio original.

```
XCHD A,@R0      ; por ejemplo ACC contiene 3AH, R0 apunta a una dirección
                  que contiene F0H, resulta que ACC contiene 30H, R0 apunta a
                  la dirección que contiene FAH.
```

SWAP : Esta instrucción invierte los nibbles superior e inferior del acumulador.

```
SWAP A          ; por ejemplo ACC comienza con 34H; termina con 43H.
```

DAA : Esta instrucción ajusta en decimal el acumulador. Si estamos sumando dígitos BCD empaquetados (un byte conteniendo dos números BCD de 4 bits) y el indicador AC esta a uno o los valores del cualquier nibble excede 9, esta instrucción suma 00H, 06H, 60H o 66H como sea necesario para devolver a los dígitos a formato BCD.

```
ADCC A,R3       ; si ACC contiene 56H (BCD empaquetado para 56) y R3
                  contiene 67H (BCD empaquetado para 67); el resultado es BEH
                  con acarreo e indicador AC también a 1.
DAA  ; el resultado es 24H en ACC con acarreo (indicando BCD
                  empaquetado de 24 con overflow o 124, el resultado
                  decimal de 56+67)
```

Esto no realiza una conversión mágica hexadecimal a decimal y no se puede usar para resta decimal. Es preferible realizar matemática en binario y convertir a forma decimal solo cuando se precise entrada de datos por teclado o salida de datos a pantalla.

2.- Lenguaje C y Ensamblador.

C es el lenguaje usado originalmente para escribir el sistema operativo UNIX y por mucho tiempo estrechamente relacionado con UNIX. Es un ***lenguaje estructurado***, y puede generar código fuente compacto. Llaves { } en lugar de palabras marcan la estructura y el lenguaje emplea bastantes símbolos poco usados. Podemos controlar varias funciones a nivel máquina sin emplear lenguaje ensamblador. Podemos escribir C condensado, sin embargo, el siguiente programador que tenga que estudiar el programa empleará bastante tiempo intentando entender como funciona. C es mucho más fácil de escribir que lenguaje ensamblador, porque el software de desarrollo se encarga de los detalles.

El lenguaje ensamblador para el 8051, descrito con detalle en el capitulo anterior, es como cualquier otro lenguaje ensamblador. Aunque es un fastidio aprender otro lenguaje ensamblador, el proceso no es difícil si ya se ha visto uno.

2.1.- Variables.

Tipo de datos	Tamaño	Rango
Bit	1 bit	0 o 1
unsigned char	1 byte	0 a 255
unsigned int	2 bytes	0 a 65535
(signed) char	1 byte	-128 a +127
(signed) (int)	2 bytes	-32768 a +32767
(signed) long	4 bytes	-2147483647 a +2147483646
unsigned long	4 bytes	0 a 4294967295
float	4 bytes	6 dígitos decimales
double	8 bytes	10 dígitos decimales

Debido a que la computación comienza con números, necesitamos entender como se almacenan y se representan. La elección del tipo de la variable o tipo de datos es más critico en el 8051 que con otros ordenadores. Las instrucciones máquina soportan de forma directa solo los dos primeros tipos de la tabla. Aunque una operación de lenguaje de alto nivel puede parecer muy simple en el programa, son necesarias una serie de instrucciones máquina para realizar la manipulación de las variables más complejas. Usar variables de punto flotante en particular añade tiempo de calculo al programa y aumenta en gran medida su tamaño.

Un tipo de variable es ***bit***. Un bit puede ser 1 (“verdadero”) o 0 (“falso”). Las variables bit que usan las instrucciones del 8051 se colocan en la RAM interna. Aunque las instrucciones 8051 lo soportan de forma directa, en C el uso de estos bits es una extensión al lenguaje estándar. El lenguaje C no está orientado a bit. No podemos emplear notación binaria, debemos emplear notación hexadecimal. La mayoría de compiladores 8051 añaden alguna forma de definir y usar el direccionamiento a bit del 8051, pero técnicamente estas extensiones hacen que el lenguaje no sea plenamente portable. Un programa realizado con estas extensiones del lenguaje no funcionaria en otro procesador que no dispusiese de este tipo de direccionamiento a bit.

Las variables de tipo ***char*** en C son valores de 8 bits, que encajan idealmente con el 8051 puesto que este puede manejar solo 8 bits a la vez. Tiene valores de 0 a 255 (sin signo) a no ser que sean con signo. El bit más significativo será el signo. Un 1 representa negativo , por lo que

las representaciones con signo y sin signo son las mismas para 0 a 127. El ordenador representa números negativos mediante notación en complemento a dos, que hace –1 11111111 y -2 11111110.

las variables ***int*** (enteros) en C son valores de 16 bits. A diferencia de otras familias de computadores se almacena el byte más significativo en la dirección menor. Los valores con signo también tienen en el msb el bit de signo y emplean la notación en complemento a dos. De forma similar a las variables int tenemos las variables ***long*** de 4 bytes (32 bits).

Representaciones exponenciales más complicadas en C son ***float*** y ***double***.

2.2.- #define y EQU.

Varios programadores emplean abreviaturas para evitar escribir demasiado. Esto se puede hacer fácilmente con expresiones #define en la cabecera del listado. Podemos usar en C por ejemplo la abreviatura ***uchar*** por ***unsigned char*** y ***uint*** por ***unsigned int***.

En ensamblador podemos hacer lo mismo con una línea ***EQU***. Podemos sustituir una larga expresión por una simple palabra.

2.3.- Espacios de memoria.

Al menos tres espacios de memoria diferentes pueden tener la misma dirección. Primero, tenemos el espacio de código, ***code***, para el código del programa y otra información que no cambie (constantes). Esto se introduciría en EPROM. No existen instrucciones para escribir en el espacio de código porque el programa no puede modificarse a sí mismo. El espacio de código también es el lugar lógico para guardar los mensajes empleados para la interacción con el usuario.

El segundo espacio de memoria es la RAM interna ***data***, (el lugar para las variables de datos). Tiene un tamaño entre 64 y 256 bytes dependiendo del procesador, y siempre forma parte del microcontrolador. Esto no es mucha memoria, pero hay gran cantidad de formas de accederla. La memoria de datos interna es un buen lugar para mantener variables temporales para cálculos, así como para mantener variables que se usan con frecuencia.

Finalmente, hay memoria externa de datos ***xdata*** que no radica en el propio chip. Se emplea normalmente de 2 a 64Kbytes en un solo chip externo. Las instrucciones máquina para acceder esta memoria deben trasladar el valor a memoria interna antes de poder usarlo, un proceso que implica varias instrucciones máquina en sí mismo, y luego devolver el resultado a memoria externa. La memoria externa es el lugar para almacenar variables usadas con poca frecuencia y para recoger datos que esperan ser procesados o enviados a otro ordenador.

```
#define PORTA XBYTE[0x4000];
bit flag1;    /* ejemplo de asignación */
code char table1[ ]={1, 2, 3, “AYUDA”, 0xff};
data unsigned int temp1;
```

Otros tipos de memoria poco usados que están disponibles en lenguaje ensamblador son idata y pdata. El primero es la memoria interna direccionable de forma indirecta (por encima de 127). El segundo es xdata con una dirección de un byte donde P2 se reserva para E/S.

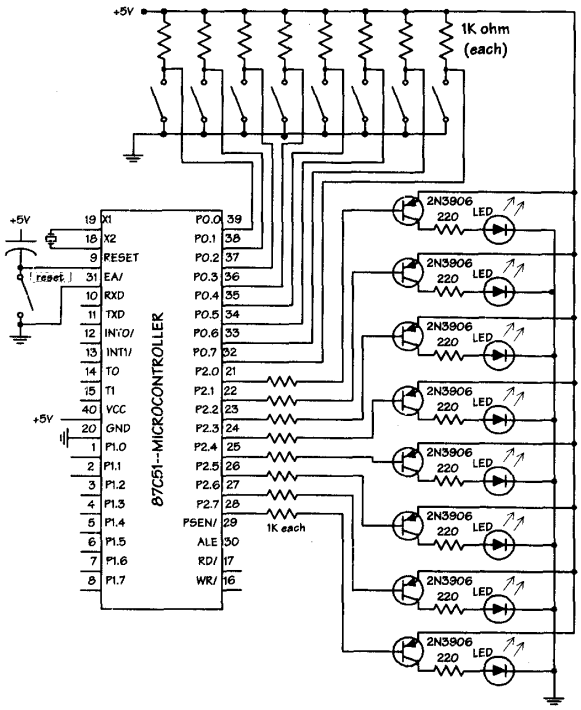
La disposición de espacio de memoria es diferente para ensamblador, como se muestra a continuación. El ejemplo muestra segmentos reubicables. Este proceso tendrá lugar una vez ensamblado el código fuente en la fase de linkado.

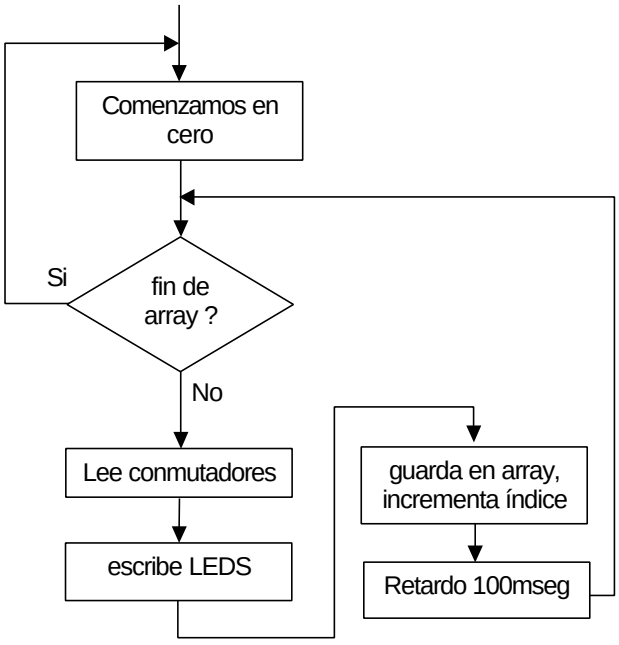
```
BITS SEGMENT BIT
ROM SEGMENT CODE
IRAM SEGMENT DATA

RSEG BITS
      FLAG1:    DBIT 1
RSEG ROM
  TABLA1:  DB  1,2,3,'AYUDA',0FFH
RSEG IRAM
 TEMP1:  DS 2
XSEG AT 6000H
  PORTA:  DS 1
END
```

2.4.- Ejemplo: conmutadores y leds.

Comenzamos con un ejemplo algo complicado que puede funcionar en el hardware mostrado. Este programa se ejecuta en un 8751 (con EPROM interna). Lee un número de ocho conmutadores conectados a P0, almacena la lectura en un array de 10 bytes, muestra la lectura más reciente sobre 8 leds conectados al puerto P2, y espera una décima de segundo para repetir el proceso.





```
#include <reg51.h>
void msec (unsigned int);
void main(void) {
    unsigned char array[10];
    unsigned char I;
    while (1) {
      for (i=0; i<=9;i++) {
               array[i]=P2=P0;
              msec(100);
      }
   }
}
```

En ensamblador resulta:

```
CODIGO SEGMENT CODE
DATOS SEGMENT DATA

RSEG DATOS
      ARRAY DS 10;
RSEG CODIGO
INICO:         MOV R0, #ARRAY   ; pone puntero de array
OTRO: MOV ACC, P0
      MOV P2, ACC
      MOV @R0, ACC      ; almacenamos un byte
      MOV R2, #0        ; byte alto
      MOV R1, #100      ; byte bajo
      CALL MSEC
      INC R0   ; apuntamos al siguiente byte
      CJNE R0, #ARRAY+10, OTRO  ; fin?
      JMP INICO
END
```

2.5.- Operadores Lógicos.

Las aplicaciones de control a menudo emplean operaciones lógicas de bit en lugar de aritméticas. Con los puertos de entrada y salida, resulta deseable leer o cambiar un bit de un byte sin afectar a los demás bits. Puede ser que este bit apague o encienda un motor mientras que los otros bits del puerto activan indicadores de aviso o comienzan la conversión de un convertidor A/D. Algunos puertos se direccionan a bit (esos conectados directamente al chip, por ejemplo), pero la mayoría de puertos añadidos responden solo como bytes enteros. Aquí es donde entran en juego los operadores lógicos de bit. La tabla siguiente los lista para los dos lenguajes.

Operación lógica	Instrucción ensamblador	C
NOT	CPL A	?
AND	ANL A, #	&
OR	ORL A, #	|
OR EXCLUSIVA	XRL A, #	^

En los siguientes ejemplos PORTA es un puerto externo direccionable a byte del que necesitamos que el tercer bit desde abajo (bit 2, porque comenzamos con 0) se ponga a uno y que el bit 6 se ponga a cero sin afectar a los demás bits.

```
extern xdata unsigned char PORTA;

void main (void) {
      PORTA = (PORTA & 0xbf) | 0x04;
}
```

En ensamblador:

```
PORTA EQU 6000H
      CSEG     AT 2000H
      MOV      DPTR, #PORTA
      MOVX     A, @DPTR          ; Obtiene la lectura actual
      ANL      A, #10111111      ; pone bit 6 bajo
      ORL      A, #00000100      ; pone bit 2 alto
      MOBS     @DPTR, A          ; emite nuevos valores
END
```

2.6.- Rotaciones y Desplazamientos.

Además de las operaciones ya mencionadas, dos operadores relacionados a bit reorganizan un byte. Primero tenemos la rotación. Si pensamos que un byte es una colección de bits ordenados desde la izquierda (msb, bit más significativo) a la derecha (lsb, bit menos significativo), podemos observar que una rotación a la derecha mueve todos los bits a la derecha un lugar. Es decir, cada bit se mueve a un lugar con menos pero. El resultado es una división por dos, de la misma forma que mover un punto decimal a la izquierda para un sistema base 10 es una división por 10. Una rotación a la izquierda produce una multiplicación por dos de la misma forma.

¿ Que le ocurre al último bit de la fila ? En lenguaje ensamblador tenemos dos tipos de rotaciones. La primera, usando RR o RL, es una rotación simple de 8 bits donde el bit más a la izquierda vuelve a la posición más a la derecha o viceversa. Por otro lado, RRC o RLC, incluye el bit de acarreo. Si el acarreo es cero, un cero rota dentro del byte.

C soporta solo el desplazamiento, a=x>>3; a la izquierda y a=x<<3; a la derecha. Un desplazamiento es una rotación que siempre rellena con ceros la entrada y descarta cualquier bit que salga por el otro lado. Un desplazamiento de 8 bits en una variable de tipo char siempre da cero.

2.7.- Asignaciones.

Solo C dispone de una representación abreviada para modificar una variable y reasignar el resultado de nuevo a la variable original. Normalmente escribiremos algo así:

```
PORTA = PORTA & 0xf7;
```

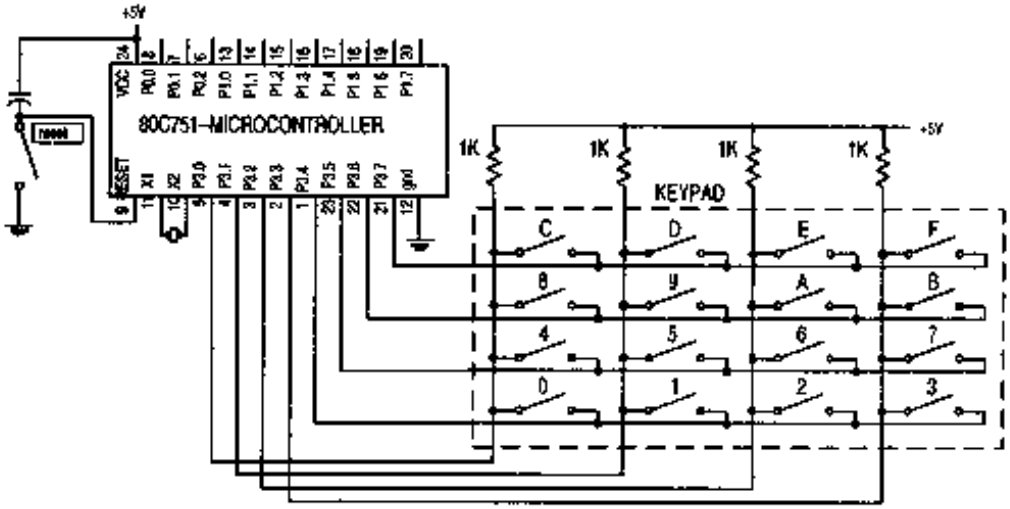
para poner a 1 el cuarto bit (bit 3 cuando contamos desde cero) a nivel bajo. En C, podemos hacer esto:

```
PORTA &= 0xf7;
```

usando el operador de asignación. Los operadores de asignación son válidos para +,-,*,/,%,<<,>>,&,^, y |.

2.8.- Cambios de un Bit.

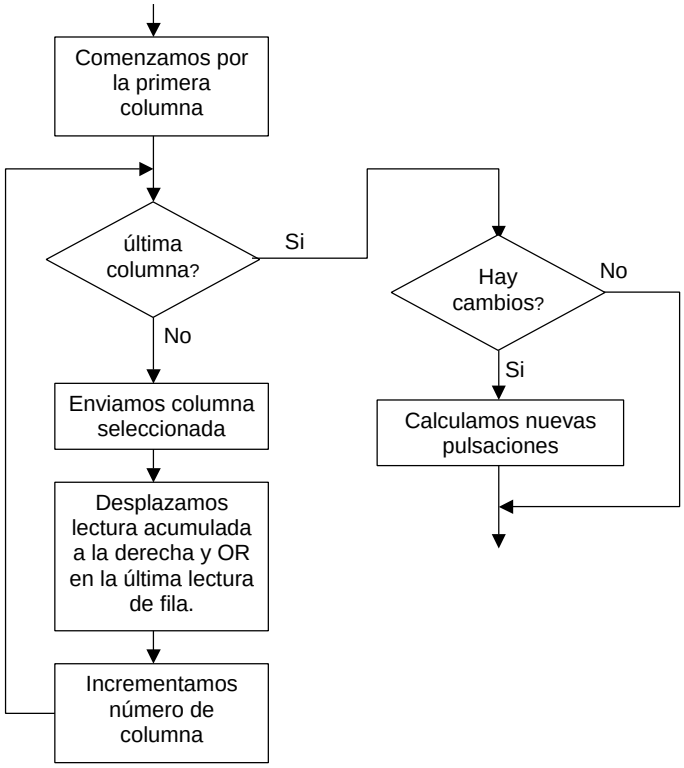
Los operadores lógicos resultan útiles para identificar cambios en una variable. Si tenemos un teclado como entrada, podemos escanear la matriz columna por columna. Cada vez ponemos una conexión de columna a nivel bajo, leyendo las filas. Las teclas pulsadas pasarán a nivel bajo, mientras que las demás teclas en la columna seleccionada quedarán a nivel alto.



Si repetimos este barrido de columnas cada 40 mseg, podremos reconocer las entradas del usuario y evitar también por el retardo de 40 mseg los problemas con los rebotes de los pulsadores. El resto del programa solo necesita detectar cambios en la entradas porque se hayan pulsado nuevos botones o se hayan soltado otros.

Lógica para detectar cambios en las teclas	
lectura previa (ant)	0 0 1 0 1 1 0 1
lectura más reciente (nuevo)	0 1 0 0 1 1 0 0
ant XOR nuevo	0 1 1 0 0 0 0 1
teclas pulsadas (ant ^ nuevo & nuevo)	0 1 0 0 0 0 0 0
teclas soltadas (ant ^ nuevo & ant)	0 0 1 0 0 0 0 1

Mediante las operaciones de bit podemos encontrar de forma eficiente las teclas que han cambiado. La primera parte del software de lectura de teclas es el barrido de una matriz 4 x 4. El nibble alto (4 bits) del puerto controlan las líneas de columna, y el nibble bajo del puerto lee las filas. El programa combina los resultados leídos (4 bits para cada columna seleccionada) en un número de 2 bytes. El último paso involucra la comparación de la nueva lectura con la anterior para identificar nuevas pulsaciones.



El programa C es corto aunque no demasiado comprensible.

```
#include <reg51.h>

unsigned int ant, nuevo, pulsado, soltado, temp;
unsigned char clmn_pat;

void main(void) {
      for (clmn_pat=0x10; clmn_pat != 0;clmn_pat<<1) {
               P3=?clmn_pat;
               nuevo = (nuevo<<4) | (P3 &0x0f);
               }
      if ((temp=nuevo ^ ant)>0) {
               pulsado = temp & nuevo;
               soltado = temp & viejo;
      }
}
```

En ensamblador resulta:

```
DSEG
      VIEJO:            DS 2
      NUEVO:            DS 2
      PULSADO:          DS 2
      SOLTADO:          DS 2

CSEG
BARRIDO:       MOV  R0, #00010000B       ; columna 1
```

```
BUCLE:          MOV  A, R0
                CPL  A
                MOVX P3, A          ; seleccionamos columna
                MOVX A, P3          ; obtenemos lectura fila
                MOV  R2, #NUEVO     ; almacenamos
                LCALL DESPLAZA
                MOV  A, R0
                RL   A              ; columna siguiente desplaza uno
                MOV  R0, A
                JNZ  BUCLE          ; bucle de barrido
                MOV  A, NUEVO       ; prueba de cambios
                XRL  A, VIEJO
                JNZ  CAMBIO
                MOV  R0, A
                MOV  A, NUEVO+1
                XRL  A, VIEJO+1
                JZ   HECHO          ; no hay cambios
CAMBIO:         MOV  R1, A
                MOV  A, NUEVO       ; obtiene pulsaciones
                ANL  A, R0
                MOV  PULSADO, A
                MOV  A, NUEVO+1
                MOV  A, VIEJO       ; obtiene botones soltados
                ANL  A, R0
                MOV  SOLTADO, A
                MOV  A, VIEJO+1
                ANL  A, R1
                MOV  SOLTADO+1, A
HECHO:          MOV  VIEJO, NUEVO   ; actualiza viejo
                MOV  VIEJO+1, NUEVO+1
                LJMP BARRIDO        ; vuelta a comenzar

DESPLAZA:       MOV  R1, #4         ; pasa los 4 bits bajos->16bits
S2:             CLR  C
                RLC  @R2+1
                RLC  @R2
                DJNZ R1, S2
                ANL  ACC,#0FH
                ORL  @R2, ACC
                RET
END
```

2.9.- Operadores Aritméticos.

Suma, resta, multiplicación y división son las operaciones matemáticas que soporta directamente el hardware de la mayoría de microcontroladores. La familia del 8051 soporta las cuatro operaciones básicas pero solo para bytes sin signo. Para poder manejar variables más grandes hay que manejar grupos de instrucciones de lenguaje ensamblador. Para variables unsigned int, los compiladores de C realizarán por nosotros las llamadas adecuadas a las librerías del compilador que realizan las operaciones matemáticas necesarias. Los compiladores completos que cumplan la norma ANSI C deben permitir también operar con variables de punto flotante de simple y doble precisión y enteros largos con signo.

Hay que emplear la precisión necesaria en cada tipo de operación a no ser que los resultados intermedios de las operaciones corran el riesgo de producir overflow. También hay que tener en cuenta que ciertos cálculos complejos que generan curvas no lineales o funciones trigonométricas es conveniente almacenarlas en forma de tabla. Un acceso a tabla es un proceso muy rápido que permite implementar controladores industriales en tiempo real.

C realiza de forma automática una ***conversión de tipos*** (expansión de bytes a word/entero, etc) cuando sea necesario. Por ejemplo, si sumamos un byte a un entero, el resultado será un entero. C tiene también una operación para forzar una variable a ser de otro tipo.

```
unsigned int a,b;
unsigned char c;
a=b+c;   /* conversión automática de tipos */
```

```
unsigned int a;
unsigned char b,c;
a=b+(unsigned int)c;  /* casting */
```

Existen instrucciones máquina que involucran el bit de acarreo, haciendo trivial el procesado de cualquier número de bytes, uno a la vez, mediante la propagación del acarreo entre diferentes bytes.

El siguiente ejemplo muestra la suma o resta de dos números sin signo de 16 bits. Obviamente, esto resulta trivial en C.

```
unsigned int x,y;
x=x-y;
x=x+y;
```

Hay que tener en cuenta que la división que proporciona el 8051 entre dos números de 8 bits, que proporciona como resultado 16 bits no devuelve un número entero y una fracción, si no que devuelve el ***resto*** en lugar de una fracción.

```
A16:  MOV  A, R3        ; suma de dos bytes
      ADD  A, R5
      MOV  R3, A
      MOV  A, R2
```

```
      ADDC A, R4
      MOV  A, R2
```

```
S16:  MOV  A, R3        ; resta de dos bytes
      CLR  C
      SUBB  A, R5
      MOV  R3, A
      MOV  A, R2
      SUBB A, R4
      MOV  A, R2
```

Los siguientes ejemplos muestran una multiplicación 8x8 y una división 8x8.

```
/* multiplicación de un byte */
unsigned char x,y;
unsigned int z;
z = (unsigned int) x * y;

/* división de un byte */
unsigned char x,y,a,b;
a = x / y; /* cociente */
b = x % y; /* resto */
```

En ensamblador:

```
; multiplicación de un byte, genera una word
VAR SEGMENT DATA
RSEG VAR
      TERM1:   DS 1
      TERM2:   DS 1
MULT SEGMENT CODE
RSEG MULT
      MOV  R0, #TERM1
      MOV  R1, #TERM2
      MOV  A, @R0
      MOV  B, @R1
      MUL  AB
      MOV  R7, A
      MOV  R6, B
END
; división de un byte, genera una word
VAR SEGMENT DATA
RSEG VAR
      TERM1:   DS 1
      TERM2:   DS 1
DIV SEGMENT CODE
RSEG DIV
      MOV  R0, #TERM1
      MOV  R1, #TERM2
      MOV  A, @R0
```

```
      MOV  B, @R1
      DIV  AB
      MOV  R7, A
      MOV  R6, #0
END
```

Como ejemplo de mecanismos para manejar números más grandes, la siguiente función realiza una suma de 4 bytes. Los mecanismos son los mismos que antes, pero un contador decide cuando el cuarto byte se ha sumado. Los valores se almacenan en las variables stka y stkb, con el msb de cada uno en la dirección más baja y el LSByte en STK+3.

```
/* suma de cuatro bytes */
unsigned long stka, stkb;
stka += stkb;
```

```
; SUMA DE 4 BYTES
SUMA4 SEGMENT CODE
ALMACEN SEGMENT DATA
RSEG ALMACEN
  STKA:       DS 4
  STKB:       DS 4
  STKC:       DS 4
RSEG SUMA4
  DADD:       MOV  R0, #STKA+3  ;LSB de A
              MOV  R1, #STKB+3  ;LSB de B
              MOV  R2, #4       ; 4 bytes a procesar
              CLR  C            ; no hay carry en la 1ª suma
  DAD1:       MOV  A, @R0
              ADDC A, @R1       ; A+B
              MOV  @R0, A       ; guardamos en A
              DEC  R0
              DEC  R1
              DJNZ R2, DAD1     ; 4 veces
              MOV  R0, #STKC
              MOV  R1, #STKB
              CALL QMOV         ; transfiere C a B
              RET
  QMOV:       MOV  R2, #4
  QMO1:       MOV  A, @R0
              MOV  @R1, A
              INC  R0
              INC  R1
              DJNZ R2, QMO1
              RET
END
```

2.10.- Operadores Lógicos.

Una característica de C que produce confusión es la distinción entre los operadores lógicos a nivel de bit y los operadores lógicos. Como hemos dicho, los operadores lógicos a bit modifican los valores de los bits individuales de una variable. Los operadores lógicos producen una respuesta del tipo ***verdadero*** o ***falso*** según las relaciones entre variables o expresiones. Las pruebas de los bucles y los saltos los emplean profusamente. El problema es que los operadores lógicos a nivel de bit y los operados lógicos tienen el mismo nombre. También tenemos un operador de asignación igual y la prueba lógica de igualdad. El igual lógico es == (un igual doble) y el OR lógico y AND lógico son ***&&*** y ||.

Si un valor de una variable en una prueba lógica es mayor que cero, es ***verdadera***. Una abreviatura resultante de esto es la omisión de ***mayor que cero*** en una prueba por un bit: *if (PORTA|0x40)* es igual que *if ((PORTA|0x40)>0).*

2.11.- Precedencia de operadores.

¿Como interpretan los compiladores líneas con varias operaciones? Por ejemplo, *A+B*C puede significar*, leyendo de izquierda a derecha, sumar A a B y luego multiplicar C por el resultado. Realmente significa, multiplicar B por C *antes de* sumar A; como es normal en álgebra, la multiplicación tiene mayor ***precedencia*** que la suma.

Empleando paréntesis, A+ (B*C), evitamos el problema y eliminamos cualquier duda porque hemos escogido evaluar la expresión encerrada por paréntesis antes de usar el resultado exterior. En ensamblador, solo tenemos una operación por línea y un flujo arriba abajo, por lo que no hay problemas de precedencia. En C, las reglas de precedencia son las del álgebra, pero la mezcla de expresiones matemáticas y lógicas puede resultar engañosa.

Operador C (más alta precedencia antes)	orden de evaluación
( ) [ ] -> .	izquierda a derecha
! ? ++ * - (tipo) & sizeof	derecha a izquierda
* / %	izquierda a derecha
+ -	izquierda a derecha
<< >>	izquierda a derecha
< <= > >=	izquierda a derecha
== !=	izquierda a derecha
&	izquierda a derecha
^	izquierda a derecha
|	izquierda a derecha
&&	izquierda a derecha
||	izquierda a derecha
?:	derecha a izquierda
= += -= %= |= &=	derecha a izquierda
,	izquierda a derecha

Aquí tenemos todas las reglas de precedencia en forma de tabla.

3.- Saltos.

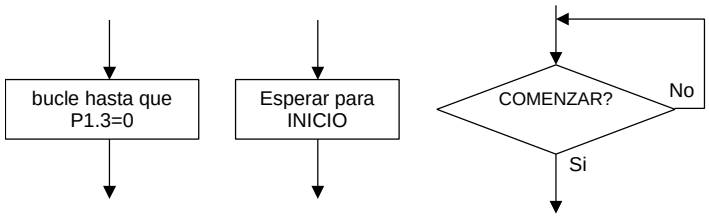
Habiendo visto que hacer con los números, el siguiente paso es controlar cuando hacerlo. Los microcontroladores demuestran su utilidad cuando toman la información del exterior y la emplean para tomar decisiones sobre la tarea que tengan que realizar:

- Dependiendo de la condición de un pulsador, activan una válvula de control o no.
- Si esta operación se ha realizado 22 veces, avanza y realiza la siguiente operación.
- Se mantiene comprobando la señal que informa que el chip de síntesis vocal puede tomar el código de la siguiente palabra.

Todos estos ejemplos son ***decisiones*** que realiza un microcontrolador de manera rutinaria. Basado en el test de decisión, el flujo del programa realizará un ***bucle*** (vuelta hacia atrás) o ***salto*** (avanzar en una de las posibles direcciones).

3.1.- Diagramas de Flujo.

Los diagramas de flujo no son dibujos detallados del programa, si no en su lugar una visión rápida del método seguido en su solución. Como cualquier código de un programa que no puede ser más largo que una página, un diagrama debe estar también contenido en una sola página. El principal requisito de un diagrama de flujo es que sea fácilmente entendible por cualquier persona. Cuando se hagan tan complejos que no se puedan colocar en una página es el momento de simplificarlos (y el programa que representa el diagrama) haciendo subrutinas.

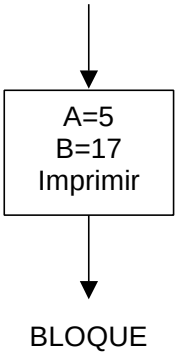


Habrá que expandir los detalles de estas subrutinas en diagramas separados en otras páginas. De esta forma, el diagrama de flujo principal siempre dará una visión general de las piezas del programa. Si necesitamos más detalle sobre el método de solución, nos dirigiremos a la subrutina apropiada.

Todos los diagramas deberían usar nombres funcionales y generalmente no referirse a nombres de variables especificas.

3.2.- Lenguaje Estructurado.

C es un lenguaje estructurado, hay reglas rígidas que prohíben romper el flujo del programa de forma arbitraria. Un lenguaje estructurado nunca permite saltar dentro de otra función sin guardar y restaurar la pila y cualquier otro registro pertinente. Empleando programación estructurada, aparte del caso especial de las interrupciones, no podemos corromper la pila con ningún conjunto de instrucciones aceptables.



El elemento básico de un lenguaje estructurado es el bloque. Es un trozo de programa donde el flujo entra solo por un lugar y lo abandona solo por otro lugar. No se puede entrar en la mitad del bloque o ejecutarlo parcialmente. El ejemplo siguiente muestra una estructura de bloque simple en C. Los tres programas ejemplo son variaciones del mismo bloque.

```
{  /* estructura básica de bloque */
a=5;
b=17;
print;
}

{ /* estilos alternativos de bloque */
A=5; B=17; print();
}

{A=5; B=17; print();}
```

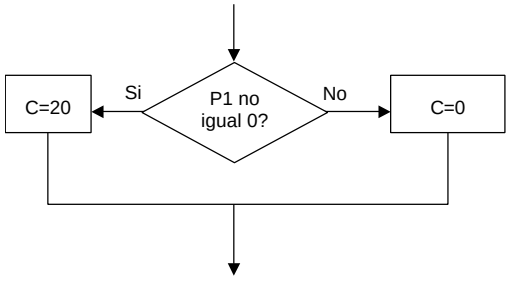
Las diferencias son relativas al hecho que C no tiene en cuenta los saltos de línea. Las sentencias individuales de C no tienen que estar en líneas separadas. Podemos hacer de esta forma los listados del programa más cortos o más largos. Esto puede servir para agrupar varias expresiones cortas cuando tengan relación entre ellas.

3.3.- Construcciones.

Las paginas que siguen discuten las construcciones de bucles y saltos. Los ejemplos en lenguaje ensamblador muestran una serie de instrucciones para realizar la misma función que en el lenguaje de alto nivel.

3.3.1.- Bifurcación.

Una decisión básica o bifurcación en el flujo del programa lo representa el bloque ***if/else***. El else es opcional, y las expresiones pueden ser bloques ellos mismos. En C, la estructura if/else es sencilla.



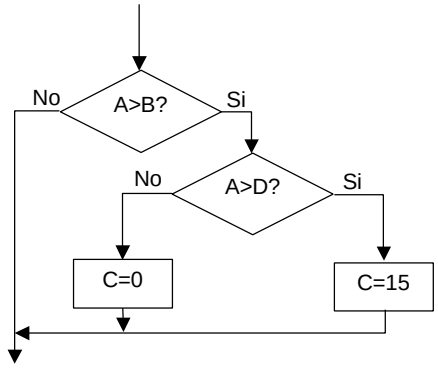

```
/* decisión if/else */
if (P1!=0) c=20;
else c=0;
```

En ensamblador, usamos las instrucciones *JZ*, *JNZ*, *JB*, *JNB*, *JC*, *JNC*. Todas las bifurcaciones condicionales involucran saltos, no hay llamadas condicionales disponibles con el 8051. Hay algunas instrucciones ensamblador de bifurcación no usuales. Con el bit de acarreo, hay que tener cuidado de no colocar ninguna instrucción que lo modifique entre la modificación del acarreo y la instrucción que realiza su test para que no cambie su valor.

```
; decisión if/else
      MOV  R0, #C
      MOV  A, P1
      JZ   X
Y:    MOV  @R0, #20H
      SJMP Z
X:    MOV  @R0, #0
Z:
```

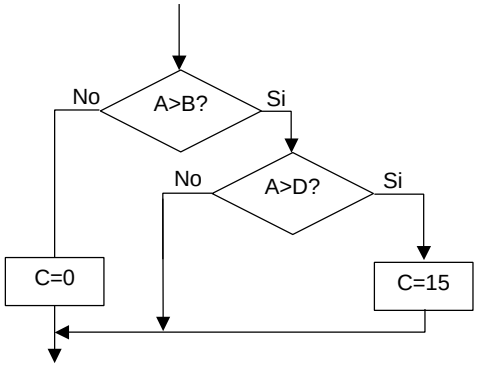
C permite realizar asignaciones dentro de la expresión de test, esto se denomina asignación implícita. Aquí es donde ++i opuesto a i++ puede resultar significativo, puesto que esto determina si la variable se incrementa antes o después del test.

Versiones 1 y 2



Los bloques else pueden anidarse, como se muestra a continuación. El anidamiento de los if enlaza el else con el if más reciente a no ser que la estructura de bloques lo defina de otra forma. El primer y segundo trozos de código en el siguiente ejemplo son idénticos, aunque la indentación sugiera que el segundo trozo de código ponga C=0 si A no es mayor que B. Si esto es lo que queremos, entonces, en el tercer trozo de código colocamos el bloque necesario {} para que el else se aplique al primer if.

Versión 3



```
/* bloques if/else anidados versiones 1, 2 y 3 */
if (a>b) {
      if (a>d) c=15;
      else c=0;
}
/* ver2 */
if (a>b)
      if (a>d) c=15;
      else c=0;
/* ver3  */
if (a>b) {
      if (a>d) c=15;
}
else c=0;
```

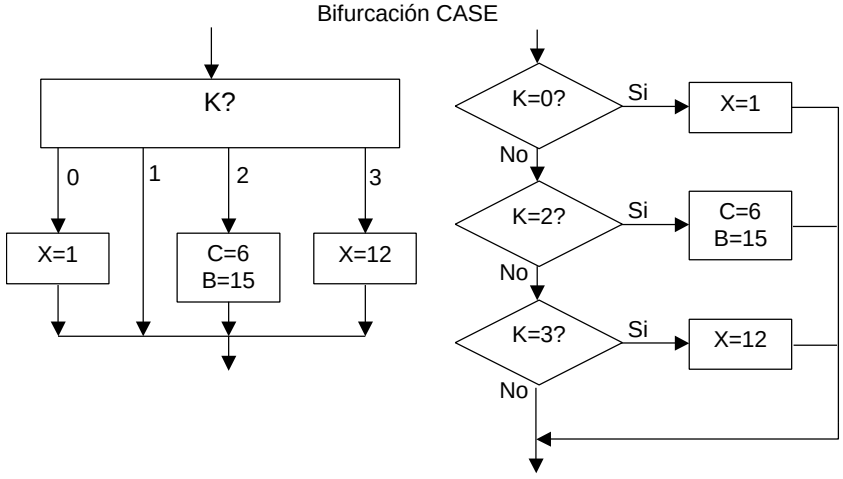
3.3.2.- Operador Condicional.

Una expresión única en C es el operador condicional. Es una abreviatura para una decisión if/else donde las dos opciones asignan simplemente un valor diferente a una variable. Es un test donde la condición cierta asigna el primer valor y la condición falsa asigna el segundo valor.

```
/* operador condicional */
c = (a>d) ? 15 : 0;
```

3.3.3.- Switch

Además de las simples construcciones de bifurcación en C tenemos la construcción switch. Permite cambiar el flujo del programa en muchas formas basándose en el valor de una variable o expresión. Una cadena de construcciones if/else pueden realizar la misma función, pero un switch puede hacer un salto multivía más entendible. La forma más simple es cuando la bifurcación depende de un entero, pero puede depender también del valor de una expresión. No tenemos que listar todos los casos posibles, tenemos un caso por defecto, o cualquier caso no existente caerá en este grupo.



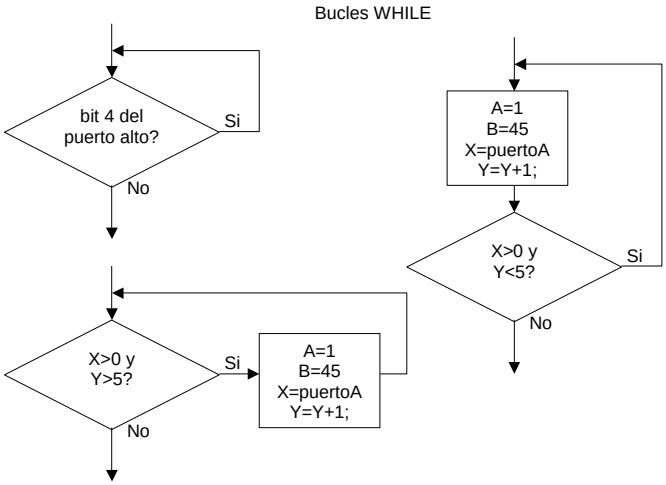
```
/* bifurcación case básica */

switch (k) {
      case 0:
               x=1;
               break;
      case 2:
               c=6;
               b=15;
               break;
      case 3:  x=12;
               break;
      default: break;
}
```

3.5.- Bucles.

3.5.1.- Bucle While.

Una construcción de bucle es el bloque ***while***. El flujo del programa continua con el bucle hasta que el test deje de ser cierto. Una forma emplea el test primero, entrando en el bloque solo si pasa el test. Si no, el flujo se salta el bloque y continua con la primera sentencia tras el bloque. Una segunda forma, el bucle do...while(), realiza el test al final del bloque para decidir si volver atrás y hacerlo de nuevo o continuar. De esta forma, el bloque siempre se ejecuta al menos una vez.




```
/* bucle while simple (vacio) */

while ((P1 & 0x10)==0);

/* bucle while normal */
while (x>0 && y++==5) {
      a=1;
      b=45;
      x=P1;
}
```

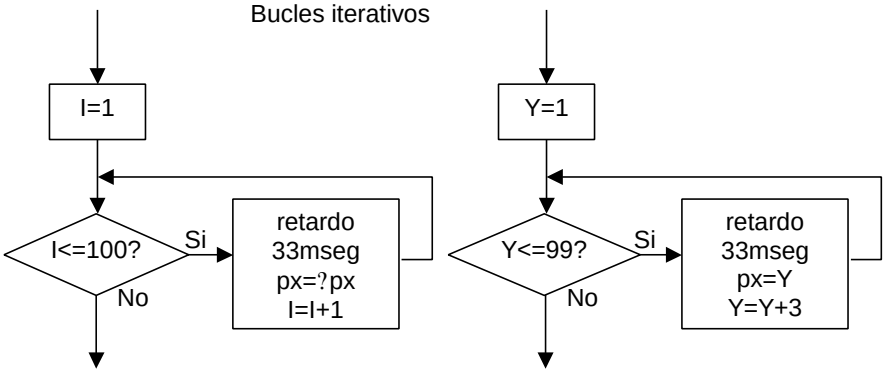
```
/* bucle do while */
do {
      a=1;
      b=45;
      x=P1;
}
while (x>0 && y++==5);
```

Existe una instrucción en ensamblador para realizar bucles. La instrucción CJNE compara los primeros dos operandos y salta solo si no son iguales. Se puede realizar fácilmente un bucle para leer un puerto hasta que aparezca un valor particular.

```
; bucle while
      MOV   DPTR, #PORTA
X:    MOVX  A, @DPTR
      ANL   A, 00010000B
      CJNE  A, 00010000B, X
```

3.5.2.- Bucle Iterativo.

Una segunda estructura muy usada en C es el bucle ***iterativo***. Emplea constantes, variables e incluso expresiones complejas para controlar el número de veces que se ejecuta el bucle. La instrucción tiene tres partes. La primera es la ***expresión inicial***. Esto normalmente asigna un valor numérico, pero puede ejecutar cualquier expresión cuando entramos por primera vez en el bucle. Luego tenemos la ***prueba para finalizar*** el bucle. Esto normalmente verifica que se cumple una determinada condición de un índice, pero podemos tener cualquier sentencia condicional cuyo fallo terminará el bucle. Finalmente, tenemos el ***incremento de índice***. Normalmente es positivo. Cada vez que se ejecuta el bucle incrementamos (o decrementamos) la variable índice. Esta parte puede tener cualquier operación o expresión que se realiza tras fallar la condición de salida antes de volver a ejecutar el bloque. Podemos hacer cosas casi incomprensibles con esta construcción.



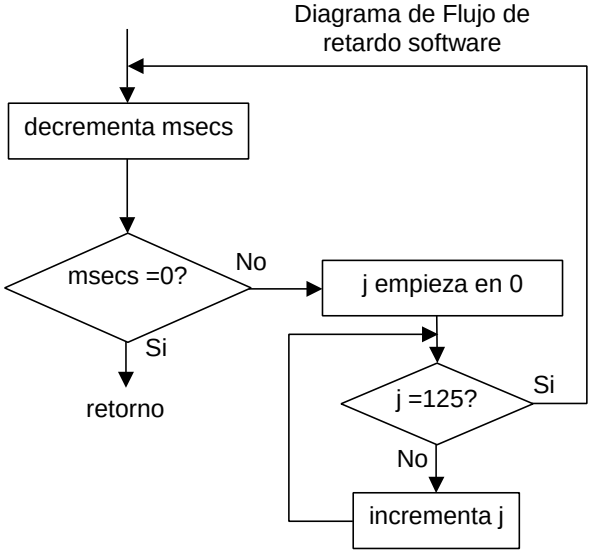

```
/* estructuras de bucles iterativos */
for (i=1;i<=100;i++) {
      retardo(33);
      px=?px;
}
/* **** */
for (y=0;y<=99;y=y+3) {
      retardo(33);
```

```
      px=y;
}
/* **** */
for (da=inicio;estado==ocupado;leds=?leds) {
      retardo(33);
}
```

El ensamblador no tiene la utilidad de C en una sola línea, pero una instrucción, DJNZ es bastante útil para pequeños bucles iterativos. Podemos usar uno de los registros o una dirección especifica de la RAM interna como índice del contador. Por su naturaleza, decrementa en uno cada vez, pero para otros decrementos, colocamos instrucciones adicionales antes del test del bucle.

```
; estructura de bucle iterativo
      MOV   R0, #100
X:    MOV   R1, #33
      LCALL RETARDO
      MOV   A, P1
      CPL   A
      MOVX  P1, A
      DJNZ  R0, X
```

3.6.- Retardo de tiempo.



Es común producir un retardo de tiempo mediante bucles anidados de forma que la ejecución de instrucciones consuma una cantidad de tiempo conocida. El siguiente ejemplo muestra estos retardos de tiempo. El tiempo en el bucle depende del número de ciclos de reloj y la frecuencia de reloj. La función de retardo aquí consume alrededor de un mseg para cada unidad que se le pasa. Pasando un valor de 50 tenemos un retardo de aproximadamente 50 x 100 = 5000 ?seg = 5 mseg.

```
/* retardo mediante bucle software */
void msec(unsigned int x) {
      unsigned char j;
      while (x-- > 0) {
               for (j=0;j<125;j++){;}
}
}
```

En ensamblador :

```
; retardo mediante bucle software
PUBLIC MSEC
MSECM SEGMENT CODE
RSEG MSECM
      MSEG:    JZ   X            ; sale si ACC=0
               MOV  R0, #250     ;250 x 4 = 1000
      Z:       NOP
               NOP
               DJNZ R0, Z        ;4 USEG por bucle
               DJNZ ACC, MSEC
      X:       RET
END
```

4.- Arrays y Punteros.

Hasta ahora la programación ha mantenido cada variable en un lugar especifico, bien escogiéndolo nosotros o dejando al compilador que lo haga por nosotros. Si vamos a tener un trozo de código operando sobre diferentes variables en momentos diferentes, necesitamos tener la posibilidad de apuntar ahora a una variable ahora y luego a otra distinta. Una de las ventajas de C es la posibilidad de hacer referencia a variables mediante punteros. La variable está basada en el puntero, es una *variable apuntada*. Nos vamos a limitar a aprender lo necesario ahora y dejaremos el resto para futura referencia.

4.1.- Arrays.

Un array es una colección de variables referenciadas por un nombre común. Hay una estrecha relación entre arrays y punteros, pero podemos usar arrays en C sin entender los punteros. Los arrays involucran un almacenamiento en un *bloque de memoria conectado*. Para un array de bytes (char), los bytes ocupan direcciones sucesivas de memoria.

```
/* acceso a un elemento array */
unsigned int ary[20];
unsigned int x;
ary[9] = x;
```

En ensamblador, se accede a un array almacenando una *dirección de comienzo* para el grupo de bytes y luego añadiendo a esta el valor de un *índice*. Supongamos que queremos el cuarto elemento de un array o vector de bytes: conseguimos la dirección de comienzo y luego añadimos 3 a esta. Esta es la razón por la que en C se empiezan a contar los elementos desde cero, para sumar directamente el índice a la dirección de comienzo.

En lenguaje ensamblador, existen diferentes instrucciones de acceso para arrays que empleen memoria interna, en estas instrucciones se usan los registros *R0* y *R1* como punteros. Debido al espacio limitado, es más común mantener los arrays en la memoria RAM o ROM externa. El ejemplo siguiente muestra una rutina que devuelve el elemento *R2* de un array de

RAM externa cuya dirección de comienzo se carga en DPTR. Esta rutina suma el valor de *R2* al puntero para colocarlo en el lugar correcto del array.

```
; ACCESO A UN ELEMENTO DE UN VECTOR
FETCH:          MOV   A, R2       ; obtiene el índice
                ADD   DPL, A      ; Suma DPTR + índice
                CLR   A
                ADDC  A, DPH      ; Propaga el acarreo
                MOV   DPH, A
                MOVX  A, @DPTR    ; obtiene valor elemento en A
                RET
```

Los arrays o vectores son mucho más fáciles de visualizar en lenguajes de alto nivel. El siguiente ejemplo crea un array, *ary* que tiene 20 miembros de 2 *bytes* (*unsigned int*) y luego copia *x* en el lugar número 10. El compilador asigna realmente *x* a las posiciones novena y décima por encima del comienzo del array. El elemento 0 en los bytes 0 y 1, el primer elemento en los bytes 2 y 3, etc, el noveno elemento en los bytes 18 y 19.

En C, podemos tener arrays bidimensionales. El siguiente ejemplo muestra un vector bidimensional de variables en punto flotante. El programa obtiene un valor y los coloca en otra variable de punto flotante. En este caso (*[5] [0]*) el compilador apunta al byte número 20 desde el comienzo del array (*C8*) porque la primera de las "filas" de diez elementos pueden tener hasta 20 bytes. El segundo ejemplo muestra la inicialización de un array para una serie de cadenas de mensaje.

```
/*arrays bidimensionales */
flota xdata ary2d [10][10];
float xdata x;
x = ary2d[5][0];
uchar code msg [][17]=
{{"Esto es una prueba.",\n"},
  {"mensaje 1",\n}
  {"mensaje 2",\n}};
```

4.2.- Tablas.

El empleo de arrays esta justificado para tablas como se muestra a continuación. En varias aplicaciones industriales resulta más eficiente usar tablas en lugar de cálculos matemáticos porque se puede ejecutar una lectura de tabla más rápido y normalmente involucra menos código que un algoritmo matemático. Calculamos previamente las tablas y las incluimos en memoria ROM.

```
/* Tabla para conversión de temperaturas */
#define uchar unsigned char
uchar code CFtbl[]={32,34,36,37,39,41};
uchar F, C;
uchar CtoF(uchar degc) {
      return CFTbl[degc];
}
```

```
main() {
C=5;
F = CtoF(c);
}
```

```
; conversión de temperatura mediante tabla
; funciona con una entrada C de 0 a 5 grados
CTOF: MOV      DPTR, #TEMPTBL    ; apunta a la tabla
      MOVC     A, @A+DPTR        ; obtenemos el valor de la tabla
      RET
      TEMPTBL: DB       32, 34, 36, 37, 39, 41
```

4.3.- Estructuras (struc).

Una ***estructura*** es un grupo de variables relacionadas referenciadas por un solo nombre. En lenguaje ensamblador, manejamos estructuras solo de forma indirecta como variables individuales o mediante indices. En C, sin embargo, las estructuras pueden resultar muy útiles. Puede ayudar el pensar en una variable de 2 bytes como una estructura formada por 2bytes. El primer byte es la parte alta del número binario, y el segundo byte es la parte baja. Cuando hacemos un vector o array de valores de 2 bytes, cada uno consiste en un par de byte alto-bajo. Conjuntamente forman el número entero.

En una escala mayor, una estructura puede representar la información acerca de por ejemplo, los reles de un sistema de control. Una parte pueden ser los 32 bits que representen 32 reles. Un 1 para un bit puede significar que un rele está “activo”. Otra parte de la estructura puede ser la cantidad de tiempo para mantener el estado, y una tercera parte puede representar cuando un estado particular es el último de la secuencia. El ejemplo muestra esta estructura.

```
struct {
      unsigned long s;          /* estado del secuenciador */
      unsigned int t;           /* tiempo de secuencia */
      unsigned char fin;        /* estado de finalización */
} estado;
estado.t=321;  /* empleo de la estructura */
```

Si tenemos varias estructuras con el mismo formato podemos definir el formato de forma separada en una *plantilla de estructura*. El nombre para el formato es la *etiqueta de estructura*, mostrada en el ejemplo anterior. Podemos usar la etiqueta para declarar nuevas estructuras.

```
#define uchar unsigned char
#define uint unsigned
struct stateform {
      unsigned long s;
      uint t;
      uchar fin;
};
struct stateform estado;
estado.t=321; /* empleo de la estructura */
```

4.3.1.- Nuevos tipos de datos: typedef.

Esta sentencia tiene usos parecidos a la sentencia #define donde el nombre sustituye a la declaración de variable. El ejemplo siguiente muestra un ***typedef*** para pares de coordenadas, definiendo un vector de pares y accediendo a él.

```
/* definición de un nuevo tipo */
#define uchar unsigned char
uchar w;
typedef struct {uchar x,y;} coordenada;
coordenada mover[20];
void main(void) {
     w=move[3].x;
}
```

La principal ventaja de un *typedef* es que la definición de variables se realiza en una sola línea. Si decidimos después que debemos tener todos los datos de coordenadas en variables *unsigned int*, solo tenemos que cambiar la información en la línea del *typedef*. También ayuda a clarificar el objetivo de la estructura.

4.3.2.- Vectores de estructuras.

En algunos casos, tendría más sentido emplear un vector o *array de estructuras*. Si tomamos el ejemplo anterior y construimos un array es bastante directo, como se muestra a continuación.

```
/* array de estructuras */
#define uchar unsigned char
#define uint unsigned
struct stateform {unsigned long s; uint t; uchar fin;};
struct stateform estado[20];
estado[11].s |= 0x04000000; /* activamos rele en estado 11 */
```

4.3.3.- Vectores dentro de estructuras.

Finalmente, en C es posible tener *arrays dentro de estructuras*. En el anterior ejemplo podría ser más eficiente en el código final evitar la referencia a los tipos grandes de variables que, dependiendo del compilador, podrían necesitar gran cantidad de código. Modificamos ligeramente el ejemplo anterior.

```
/* array de estructuras con arrays */
#define uchar unsigned char
#define uint unsigned
struct stateform {
     uchar s[4];uint t; uchar fin;};
struct stateform estado[20];
estado[11].s[0] |= 0x04; /* activamos rele en estado 11 */
```

El lenguaje C también puede admitir *estructuras anidadas*. Es difícil aportar un ejemplo en control por ordenador, pero supongamos que podemos tomar el ejemplo anterior y añadir un trozo separado de información relacionada con pulsos de un sensor, como se muestra en el siguiente ejemplo.

```
/* estructuras anidadas */
#define uchar unsigned char
#define uint unsigned
struct sf { uchar s[4]; uint t; uchar fin;};
struct sensortipo{          struct sf estado;
                            uchar cuentasensor;};
struct sensortipo sensores[20];
sensores[11].estado.s[0] |= 0x04;
```

```
/* estructuras anidadas (sin etiquetas) */
#define uchar unsigned char
#define uint unsigned
struct {struct{ uchar s[4]; uint t; uchar fin;} estado;
                    uchar cuentasensor;} sensores[20];
sensores[11].estado.s[0] |= 0x04;
```

Este ejemplo no es nada simple y probablemente será innecesario para aplicaciones de control. Se emplea para procesamiento de datos, donde una estructura puede contener el nombre y la dirección de un empleado y otras estructuras pueden emplear esta plantilla con datos adicionales para almacenar otra información. Si queremos liar al lector ocasional del código habrá que emplear la versión sin etiquetas. Además, también podemos colocarlo con el mínimo número de líneas posible.

Al compilador no le importa, porque ignora los saltos de línea, pero a cualquiera que tenga que entenderlo le costara bastante más trabajo. Por favor, utilizar saltos de línea y tabulaciones lo más posible y colocar las definiciones de elementos internos a las estructuras en una misma columna.

4.4.- Escoger espacios de memoria para variables.

Con la familia de microcontroladores 8051, al menos hay tres tipos diferentes de memoria. Hemos hablado de variables por su nombre, ignorando los detalles de donde se almacenaban. En ensamblador nosotros hacemos la elección de forma deliberada, solo hay unas pocas instrucciones (MOVX y MOVC) que admiten memoria externa.

C prefiere dejar la asignación de variables al compilador, a no ser que nosotros especifiquemos otra cosa. En C, podemos dejar la decisión de donde colocar las variables al ***modelo de memoria***, que por defecto emplea el modelo ***small*** (todas las variables en memoria interna). Si se emplea *small*, el compilador almacena todas las variables a no ser que se especifique lo contrario (con *xdata*, *code*, etc..) en la memoria interna (128 bytes en 8051). Para pequeños programas con pocas variables, tenemos bastante con la memoria interna.

Los modelos de memoria más grandes almacenan las variables en memoria externa. Podemos escoger el espacio de memoria para cada variable de forma individual en su definición empleando extensiones del C como *xdata*, *code* o *data*.

4.5.- Punteros.

Un ***puntero*** es una variable que guarda la dirección de otra variable. La variable a la que un puntero apunta es una ***variable apuntada*** o basada. En ensamblador, podemos usar un byte en el registro *R0* o *R1* para apuntar al espacio de memoria interna, o un valor de 2 bytes puesto en el registro *DPTR* para apuntar a RAM externa o a espacio de código (EPROM). Los puntero en C son más complicados. Aquí tenemos un ejemplo en C para hacer que una variable puntero apunte a una variable y luego usar la variable *apuntada*.

```
#define uchar unsigned char
uchar count;
uchar *x;
uchar xdata *y;
uchar data *z;
uchar code *w;
uchar data *xdata zz;
x = &count;    /* indirección de count, x apunta a count */
*x = 0xfe;     /* acceso a apuntado por puntero x -> count */
```

El siguiente ejemplo carga una variable apuntada desde un elemento de un array de estructuras. El código ensamblador resultante es correcto.

```
/* puntero a xdata que reside en data */
#define uchar unsigned char
uchar xdata *data y;
typedef struct {uchar x,y;} coord;
coord mover[20];
void main (void) {
      y=0x6000;
      *y = mover[3].x;
}

; FUNCTION main (BEGIN)
      MOV      y,#060H
      MOV      y+01H, #00H       ; y=0x6000;
      MOV      DPL, y+01H
      MOV      DPH, y            ; DPTR=y
      MOV      A, mover+06H      ; A=mover[3*2];
      MOVX     @DPTR, A          ; (0x6000) <- A
      RET
; FUNCTION main (END)
```

4.5.1.- Punteros universales.

Algunos compiladores disponen de punteros que incluyen un tercer byte que guarda un código que identifica que tipo de espacio de memoria se emplea. Para los punteros especificos, el compilador usa punteros de 2 bytes. El puntero de 3 bytes permite que funciones de librería admitan punteros a cualquier tipo de espacio de memoria y por tanto puedan funcionar con variables localizadas en cualquier espacio de memoria. En otro caso, el programador debe conocer que espacio de memoria y usar la función adecuada con el espacio de memoria correspondiente sin equivocarse. En todas estas situaciones hay que llegar a un compromiso entre eficiencia del programa y flexibilidad.

4.5.2.- Punteros a arrays.

Las cosas comienzan a complicarse cuando consideramos punteros a arrays. C puede hacerlo sin problemas. Un array apuntado es bastante directo

```
/* array basados */
#define uint unsigned
uint xdata a[];
a[22] = 0xff;
/* de otra forma */
#define uint unsigned
uint xdata ar[];
uint xdata *a;
a=&ar;
*(a+22) = 0xff;
```

4.5.3.- Arrays de punteros a arrays.

Las relaciones en C entre arrays y punteros pueden producir confusión. El nombre de un array es un puntero. Podemos tener un puntero a un array y podemos tener un array de punteros. Podemos incluso tener un puntero a un array de punteros (esto es ***indirección múltiple***).

En el siguiente ejemplo, se emplea un array de punteros para apuntar a un grupo de trozos de mensaje que se agrupan para mandarlos a la consola. El ejemplo mantiene todos los trozos del mensaje, así como el array de punteros en *espacio de código*.

Durante el funcionamiento de cualquier programa unas cadenas de texto se deben extraer de la zona de código mientras que otras de variables (en RAM), para esto o bien usamos un puntero genérico para apuntar a cualquier zona de memoria o hacemos un array de estructuras donde un elemento designa el tipo de memoria y un segundo elemento es el puntero. En el ejemplo, con los punteros a los trozos de mensaje puestos en un array, podemos pasar el puntero al array de punteros a la función de visualización, que trabajará con la cadena entera.

Aunque esto parece un poco complicado, si tenemos que ahorrar espacio de memoria para los mensajes, no nos queda otra alternativa, tenemos que identificar frases repetidas y reutilizar el código con este tipo de técnica de sistema de punteros.

```
/* indirección múltiple */
#define uchar unsigned char
uchar code m1[]={“esto es una prueba”};
uchar code m2[]={“respuesta incorrecta”};
uchar code m3[]={“respuesta correcta”};
uchar code m3[]={0};
uchar code *code fallo[]={&m1[0],&m2[0],0};
uchar code *code paso[]={&m1[0],&m2[0],0};

void pantalla (uchar code **mensaje) {
      uchar code *m;
      for (;*mensaje!=0;mensaje++) {
               for (m=*mensaje;*m!=0;m++) {
                        P1=*m;
```

```
                }
        }
}
main () {
        pantalla(&paso[0]);
}
```

Cada array que se referencia por el array de punteros no necesita tener el mismo tamaño. Alguna gente denomina esto ***array disperso***. Para este ejemplo, el compilador asume que mantenemos los índices dentro de los márgenes por nuestros propios medios. Para arrays predefinidos, C tiene varias funciones de librería como *sizeof* y *strlen* que nos dan el tamaño de los datos.

4.5.4.- Punteros a estructuras.

Solo es un pequeño paso ir de arrays basados a estructuras basadas. Una aplicación real para emplear estructuras basadas es el formato de un mensaje en un sistema multitarea. Una tarea se comunica con otra "mandando" un mensaje. Podemos manejar tan solo el *puntero* al mensaje del sistema operativo, que pasará el mensaje (el puntero) a la tarea que lo recibe. Ambas tareas deben estar de acuerdo en la estructura del mensaje.

```
/* estructuras basadas en punteros */
#define uint unsigned
#define uchar unsigned char
struct msg1 {uint lnk; uchar len, flg, nod, sdt, cmd, stuff;};
struct msg1 *msg;
void mandamensaje(struct msg1 *m); /*prototipo de función */
main() {
uchar datos;
msg->len=8;
msg->flg=0;
msg->nod=0;
msg->sdt=0x12;
msg->cmd=0;
msg->stuff=datos;
mandamensaje(msg);
}
```

4.6.- Uniones.

Normalmente se incluye con las estructuras otro tipo de datos denominado ***union***. Una union es, como el nombre implica, una combinación de diferentes tipos de datos, aplicándose diferentes nombres y tipos para el *mismo* espacio de memoria. Supongamos que queremos almacenar un temporizador de 16 bits que queremos leer como 2 bytes. Aunque podemos usar un cast a un entero y un desplazamiento de 8 bits para obtener los 8 bits altos en su lugar, también es posible definir una union formada por una estructura de 2 bytes y un entero. Cuando queremos rellenar el byte alto, nos referimos a este espacio como 2 bytes, pero cuando queramos usar el resultado, nos referimos al espacio como un entero.

```
/* union de entero y bytes */
#define uint unsigned
#define uchar unsigned char
union split (  uint palabra;
               struct{uchar alto; uchar bajo;} bytes;
};
union split nuevacuenta;
nuevacuenta.bytes.alto=TH1;
nuevacuenta.bytes.bajo=TL1;
viejacuenta=nuevacuenta.palabra;
```