PIC
PROGRAMMAZIONE IN LINGUAGGIO MACCHINA PER PRINCIPIANTI

By Claudio Fin

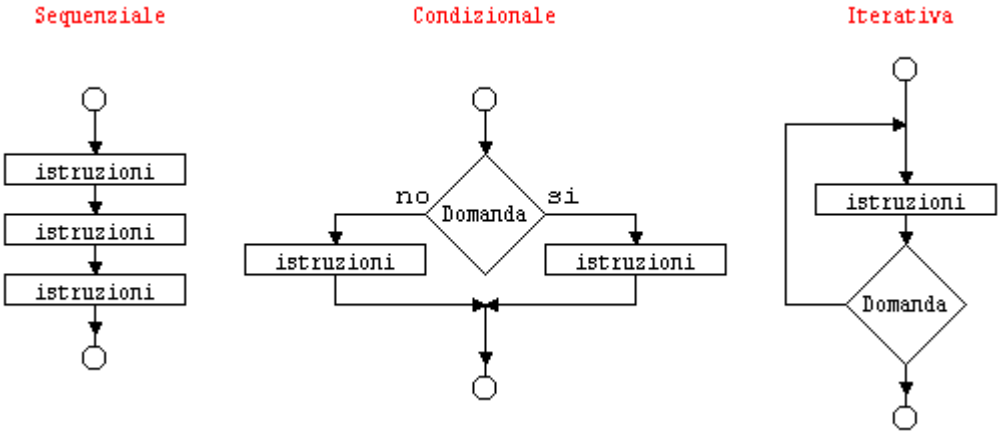
[Precedente] [Indice principale]

CONTROLLO FLUSSO, SALTI, SUBROUTINE

```
BTFSC   reg,b      Skip se bit b di (reg) = 0
BTFSS   reg,b      Skip se bit b di (reg) = 1
INCFSZ  reg,d      d = (reg) + 1  Skip se d = 0
DECFSZ  reg,d      d = (reg) - 1  Skip se d = 0
GOTO    addr       Salto all'indirizzo addr
CALL    addr       Chiamata di subroutine
RETURN             Ritorno da subroutine
RETLW   n          Ritorno da subr.con valore in W
RETFIE             Ritorno da interrupt
```

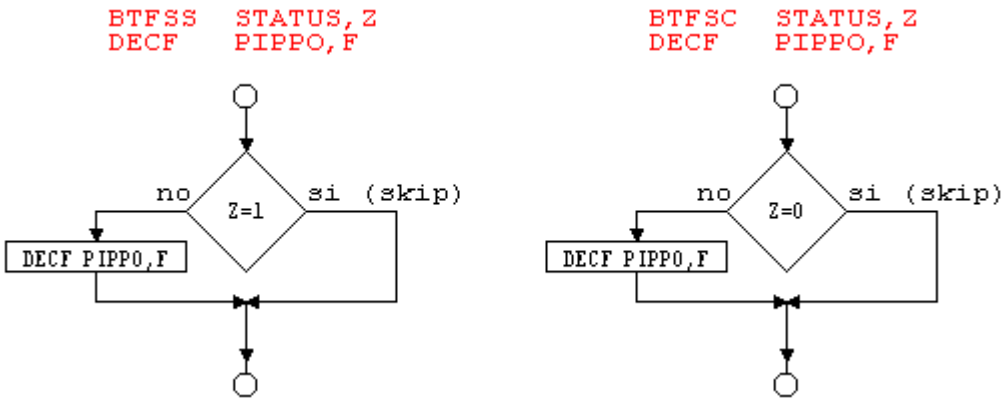
E ora eccoci ad un insieme corposo e importante di istruzioni e concetti. Fino ad adesso infatti abbiamo visto solo quali sono i modi per impostare e manipolare in modo elementare i nostri dati, scriverne il valore su 8 LED e stop. Se fosse tutto qui un micro servirebbe a ben poco, ciò che manca ancora sono delle istruzioni in grado di far «prendere delle decisioni» al programma, in modo da effettuare un'operazione o un'altra a seconda del verificarsi di una certa condizione, ed eventualmente ripetere più volte un gruppo di istruzioni, in modo da creare così un processo dinamico che può continuare a leggere gli ingressi e generare gli opportuni segnali sulle uscite.

Le istruzioni di questo gruppo si dicono di «controllo flusso» in quanto permettono effettivamente di alterare il normale flusso lineare di esecuzione, creando delle «ramificazioni» (branch, salti) a punti (indirizzi) diversi del programma. Usando i salti è possibile «strutturare» il programma a piacimento (il significato di struttura diverrà più chiaro andando avanti), a tale proposito è utile però ricordare che uno stile di programmazione corretto (che facilita il progetto, la lettura e la manutenzione) prevede l'uso di tre soli tipi fondamentali di «struttura», ciascuna con un solo punto di inizio e una sola fine:



La struttura sequenziale è quanto abbiamo già visto fino ad ora, è un insieme (o blocco) di istruzioni da eseguire una dopo l'altra. La struttura condizionale prevede la possibilità di porsi una domanda (contenuta nel rombo) a cui si può rispondere con un si o con un no, e di eseguire istruzioni (o blocchi di istruzioni) differenti a seconda dei due casi. La terza struttura permette di ripetere una o più istruzioni finchè la risposta alla domanda vale si (oppure no a seconda delle necessità). Con questa simbologia grafica è possibile rappresentare il funzionamento di un qualsiasi programma sotto forma di flowchart (diagramma di flusso). Ciascun rettangolo può essere un'istruzione elementare, oppure contenere più istruzioni o anche «sottostrutture». Uno dei rettangoli dell'istruzione condizionale può cioè contenere al suo interno altre strutture condizionali e/o iterative, che a loro volta ne possono contenere delle altre... il livello di dettaglio che si vuole rappresentare dipende dalle necessità caso per caso.

Ma vediamo allora come un programma può porsi una domanda grazie alle prime due istruzioni della tabella: BTFSS e BTFSC. Queste servono a «testare» (o controllare) il valore (stato logico 1 o 0) di un qualsiasi bit di un qualsiasi registro, e permettono di saltare (skip) l'esecuzione dell'istruzione successiva se il bit testato si trova nello stato previsto. La prima salta l'istruzione seguente se il bit testato è 1 (alto, set), la seconda se è 0 (basso, clear). Il bit da testare è specificato con il parametro **b**, e come sempre può andare da 0 per indicare il bit meno significativo, a 7 per quello più significativo.



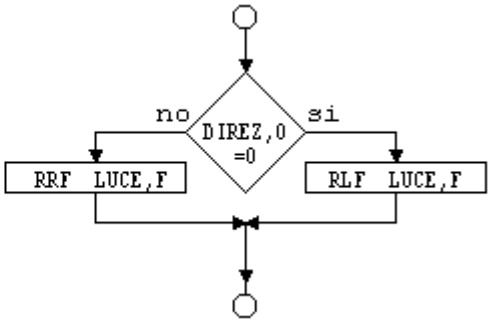
Come si vede in questi due esempi viene testato il flag Z, in fondo i flags sono due bit come tutti gli altri, e diventa evidente la loro grandissima importanza. Infatti è anche grazie al loro valore (che dipende dal risultato delle istruzioni appena eseguite) che il programma può prendere decisioni anche complesse e comportarsi in modi diversi a seconda delle circostanze.

Un semplice skip dell'istruzione successiva non fornisce però ancora la flessibilità necessaria per realizzare una completa struttura condizionale equivalente all' IF THEN ELSE (se, allora altrimenti) dei «linguaggi ad alto livello». Questa si ottiene con l'aggiunta dell'istruzione GOTO (vai a, salta), che permette di saltare ad un punto qualsiasi del programma da un qualsiasi altro punto. Quando una GOTO è usata insieme ad una istruzione di skip si crea un salto condizionato. E' naturalmente possibile mettere una GOTO in qualsiasi punto del programma, in tal caso questo salto prende il nome di salto incondizionato. E' buona norma però ridurre il più possibile l'utilizzo dei salti perché rendono complicata la lettura e la manutenzione/correzione del programma. Il loro utilizzo andrebbe limitato (se possibile) esclusivamente per la «creazione» delle strutture fondamentali

condizionale e iterativa. Per indicare il punto di arrivo di un salto (GOTO) si usano normalmente delle etichette (label), cioè dei nomi inizianti nella prima colonna del testo. E' evidente che in un programma non devono mai esserci due label uguali, in quanto ciascuna identifica un indirizzo della memoria programma ben preciso.

L'esempio seguente mostra come realizzare in assembly la struttura IF THEN ELSE completa. Nell'esempio viene testato il bit 0 del registro DIREZ, se vale 0 viene eseguita l'istruzione RLF LUCE,F (saltando la GOTO ELSE) seguita dal salto a ENDIF, altrimenti viene eseguita l'istruzione RRF LUCE,F. Le istruzioni da eseguire in un caso o nell'altro possono essere anche più di una, a differenza della skip semplice che al massimo permette di saltare l'esecuzione di una singola istruzione.

```
        BTFSC   DIREZ,0
        GOTO    ELSE
        RLF     LUCE,F
        GOTO    ENDIF
ELSE    RRF     LUCE,F
ENDIF   ........
```



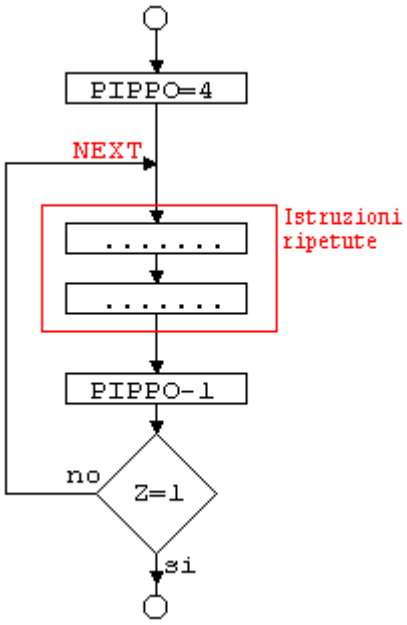
L' esempio seguente presenta la struttura iterativa, ed è molto importante comprenderlo appieno perché su questo genere di struttura/funzionamento si basa gran parte dell'elaborazione nei casi reali.

E' una generica sezione di programma in cui ad un certo punto si carica il valore 4 in un registro della RAM (chiamato come al solito PIPPO). Seguono poi delle istruzioni indicate con dei puntini. Ed infine troviamo altre 4 istruzioni. Le prime due servono a sottrarre 1 dal registro PIPPO. La terza controlla il valore del flag Z che viene settato quando il risultato dell'istruzione precedente vale 0. Se il flag Z è settato viene saltata l'istruzione seguente e il programma prosegue. Altrimenti, se PIPPO dopo la sottrazione non vale 0 e il flag Z non è settato, lo skip non viene effettuato, perciò viene eseguita la GOTO che rimanda indietro al punto indicato dall'etichetta NEXT. In pratica si crea quello che viene chiamato un loop (ciclo, anello) che ripete il blocco di istruzioni «interne» esattamente 4 volte. Il registro PIPPO funziona da contatore di ciclo. Siccome il suo valore parte da 4 e decrementa ad ogni ciclo, al quarto passaggio raggiunge lo 0 e il loop termina, il programma prosegue cioè con le istruzioni successive....

```
        MOVLW   4           ;W=4
        MOVWF   PIPPO       ;PIPPO=W

NEXT    ....
        ....

        MOVLW   1           ;W=1
        SUBWF   PIPPO,F     ;PIPPO=PIPPO-W
        BTFSS   STATUS,Z    ;SE PIPPO=0 ALLORA SKIP (FINE CICLO)
        GOTO    NEXT        ;ALTRIMENTI TORNA A NEXT
```



Abbiamo ormai quasi tutti gli elementi per poter finalmente «animare» un pò i nostri LED, prima però vediamo qualche altra istruzione. La terza e la quarta istruzione della tabella sono istruzioni «potenti», in quanto permettono in un colpo solo di incrementare o decrementare di 1 un registro, ed effettuare automaticamente uno skip dell'istruzione seguente se il risultato dell'operazione vale 0. L'esempio precedente può perciò essere riscritto più succintamente nel seguente modo:

```
        MOVLW   4           ;W=4
        MOVWF   PIPPO       ;PIPPO=W

NEXT    ....
        ....

        DECFSZ  PIPPO,F     ;DECR.PIPPO, SE=0 ALLORA SKIP (FINE CICLO)
        GOTO    NEXT        ;ALTRIMENTI TORNA A NEXT
```

Con i salti (condizionati e non) è già possibile creare qualsiasi flusso elaborativo, tuttavia se c'è bisogno di creare delle suddivisioni logiche tra sezioni di programma che eseguono compiti specifici, oppure ci sono gruppi di istruzioni ripetuti frequentemente in diversi punti del programma, è preferibile ricorrere alle subroutines (o sottoprogrammi). Ogni subroutine può essere così considerata come un «modulo» a parte, da «chiamare» quando serve il suo servizio. Le subroutines sono semplicemente porzioni di programma che vengono chiamate con l'istruzione CALL (nello stesso modo di GOTO) ma terminano con l'istruzione RETURN che fa ritornare il programma all'istruzione successiva alla CALL di partenza. Questo comportamento è possibile perché ad ogni CALL il PIC salva il contatore di programma (il program counter, che tiene conto dell'indirizzo dell'istruzione in esecuzione) in un'area speciale chiamata stack, e lo riprende quando incontra una RETURN. Il PIC 16F628 ha un'area di stack a 8 livelli, è quindi possibile chiamare fino a 8 subroutines una dentro l'altra (nidificate o nested). L'uso delle subroutines è estremamente consigliato, in quanto sono l'unica cosa in grado di tenere «in ordine» un programma complesso spezzandolo in più moduli, cioè porzioni di codice di dimensioni ridotte che eseguono ciascuna un compito ben specifico, più facile da comprendere e da mettere a punto. Un'altra buona ragione per l' utilizzo dei sottoprogrammi è che all'evenienza possono essere copiati in altri programmi per

essere riutilizzati. In gergo una raccolta utile di subroutines riutilizzabili si chiama «libreria».

Ci sono altre due forme di istruzione di ritorno da subroutine, la RETLW e la RETFIE. La prima permette il ritorno da una subroutine con un valore specifico caricato nell'accumulatore, come vedremo serve per creare delle tabelle dati nell'area programma, cosa altrimenti impossibile in quanto questa memoria non è accessibile dalle comuni istruzioni che lavorano solo sui registri dalla memoria dati. La RETFIE invece si usa nel caso in cui la subroutine da terminare sia un gestore di interrupt.

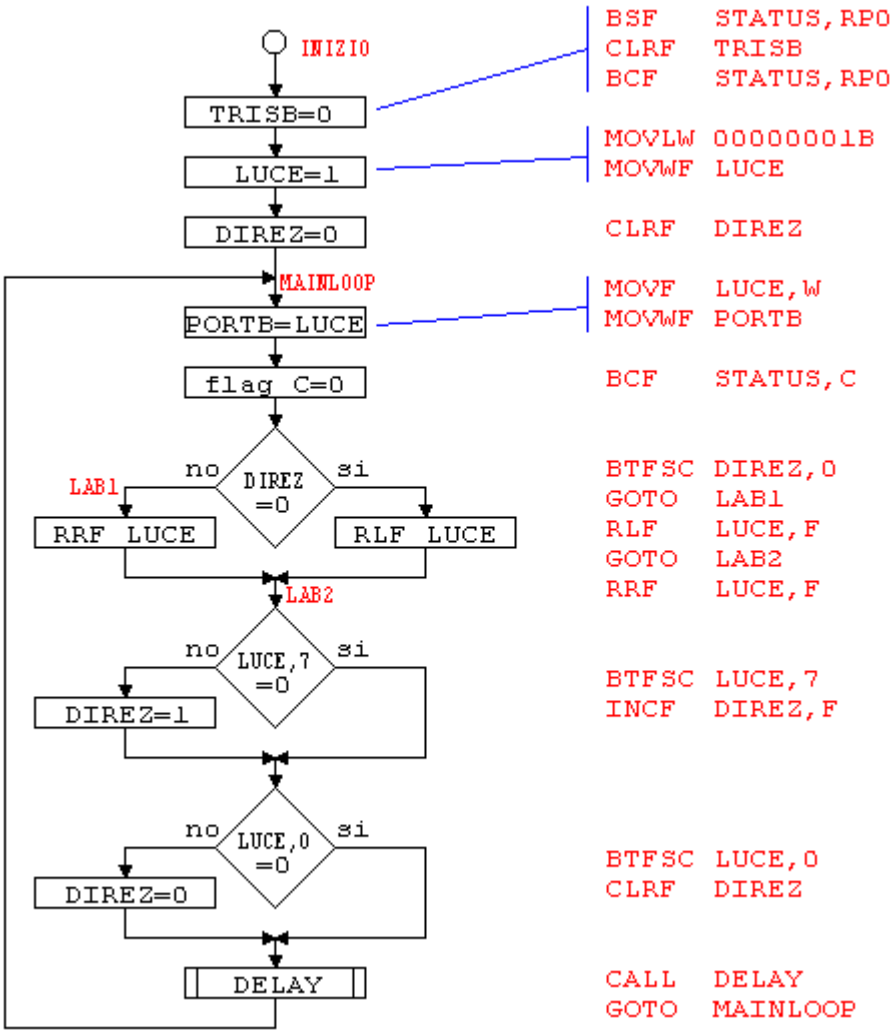
Esperimento:
Ed ora, per mettere in pratica tutto quanto, subroutines, salti condizionati e rotazioni di bit, vogliamo ottenere sui LED un «effetto supercar», cioè deve accendersi un LED alla volta scorrendo da destra verso sinistra per poi tornare indietro e così via.

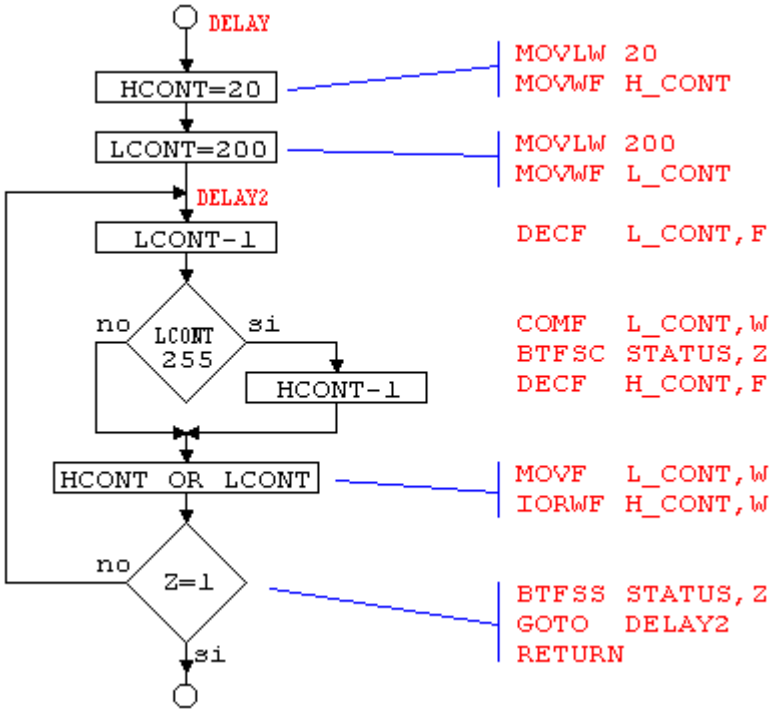
Dapprima definiamo le variabili di lavoro (registri) che ci occorrono. Un registro di nome LUCE conterrà il valore a 8 bit da scrivere sulla porta di uscita, uno chiamato DIREZ conterrà 0 o 1 a seconda che la direzione di scorrimento voluta sia verso sinistra o verso destra, ed infine due registri di nome L_CONT e H_CONT serviranno come contatore di ciclo a 16 bit per una subroutine di ritardo. Quest'ultima è necessaria perché la velocità di esecuzione sarebbe altrimenti così rapida da non poter essere notata, in pratica tutti i LED apparirebbero ugualmente illuminati a causa della persistenza dell'immagine sulla nostra retina. Visto che ogni istruzione impiega un certo tempo per essere eseguita è evidente che eseguendo decine di migliaia di volte poche istruzioni all'interno di un loop si possono ottenere ritardi di tempo considerevoli. E' necessario usare 16 bit per il conteggio in quanto usandone solo 8 il massimo ritardo ottenibile sarebbe troppo breve (meno di 2 millisecondi). Per visualizzare un effetto gradevole ci occorre invece un ritardo di diverse decine di ms. Una soluzione poteva essere quella di richiamare la sbroutine di ritardo a 8 bit della durata di 2 ms molte volte di fila, ma questo avrebbe in ogni caso comportato l'uso di un altro ciclo e di un altro registro di conteggio. In questo esempio invece si usano due byte in modo tale da considerarli come un unico registro a 16 bit, con cui realizzare cicli da 1 a 65536 ripetizioni, ed è il primo esempio di «creazione» via software di un tipo di dato che può assumere valori maggiori di quelli trattabili direttamente dal micro. Un valore a 16 bit è composto infatti da due byte detti alto (H, più significativo) e basso (L, meno significativo). Il valore contenuto in quello alto ha peso 256, una unità nel byte alto vale cioè 256 volte la stessa unità nel byte basso, questo vuol dire che il valore complessivo a 16 bit di due byte H ed L è dato da H*256 + L. Nel nostro caso impostiamo un ciclo di 5320 ripetizioni, la parte alta varrà 20 e quella bassa 200 (infatti 20*256+200=5320). Per decrementare un valore a 16 bit si decrementa prima la parte bassa, se si ha un rollover negativo (da 0 si torna a 255) allora si decrementa anche la parte alta. Quando entrambe le parti contengono zero (basta fare un OR tra di loro e verificare se si setta il flag Z) il ciclo termina. Durante l'esecuzione del programma il valore binario iniziale 00000001 di LUCE viene spostato verso sinistra di una posizione alla volta tramite l'istruzione RLF. L'azzeramento del flag C prima della rotazione fa si che da destra non entri mai un altro bit a 1. Quando il bit a 1 contenuto in LUCE raggiunge la posizione 7 (cioè l'ottava a sinistra, la piùsignificativa) il valore del registro direzione viene messo a 1, in modo tale da abilitare lo spostamento verso destra. Allo stesso modo quando il bit a 1 di LUCE ritorna nella posizione 0 il valore

della direzione cambia di nuovo riabilitando lo spostamento verso sinistra come all'inizio. Il programma è senza fine perché il GOTO finale rimanda all'infinito alla posizione MAINLOOP, in totale occupa 33 locazioni di memoria programma delle 2048 disponibili, e utilizza 4 byte di memoria RAM dei 224 disponibili.

```
;-----------------------------------------------------
; Programma effetto supercar
;-----------------------------------------------------
          PROCESSOR 16F628
          RADIX     DEC
          INCLUDE   "P16F628.INC"
          __CONFIG  11110100010000B
;-----------------------------------------------------
LUCE      EQU       32
H_CONT    EQU       33
L_CONT    EQU       34
DIREZ     EQU       35
;-----------------------------------------------------
          ORG       0
          BSF       STATUS,RP0  ;Attiva banco 1
          CLRF      TRISB       ;Rende PORTB un'uscita
          BCF       STATUS,RP0  ;Ritorna al banco 0
          MOVLW     00000001B
          MOVWF     LUCE        ;LUCE=00000001
          CLRF      DIREZ       ;DIREZ=0
MAINLOOP  MOVF      LUCE,W
          MOVWF     PORTB       ;Scrive LUCE sulla PORTB
          BCF       STATUS,C    ;Azzera flag C
          BTFSC     DIREZ,0     ;Se DIREZ=0 (sinistra) skip
          GOTO      LAB1        ;altrimenti GOTO LAB1
          RLF       LUCE,F      ;Ruota LUCE verso sinistra
          GOTO      LAB2        ;GOTO fine struttura IF
LAB1      RRF       LUCE,F      ;Ruota LUCE verso destra
LAB2      BTFSC     LUCE,7      ;Se bit 7 di LUCE <> 0 skip
          INCF      DIREZ,F     ;altrimenti direzione=destra
          BTFSC     LUCE,0      ;Se bit 0 di luce <> 0 skip
          CLRF      DIREZ       ;altrimenti direzione=sinistra
          CALL      DELAY       ;Richiama subroutine di ritardo
          GOTO      MAINLOOP    ;Nuovo ciclo del programma
;-----------------------------------------------------
DELAY     MOVLW     20          ;Carica 5320 nei 16 bit
          MOVWF     H_CONT      ;formati dai due byte
          MOVLW     200         ;H_CONT e L_CONT
          MOVWF     L_CONT
DELAY2    DECF      L_CONT,F    ;Decrementa parte bassa del contatore
          COMF      L_CONT,W    ;Inverte i bit
          BTFSC     STATUS,Z    ;Se tutti zero c'<> stato rollover
          DECF      H_CONT,F    ;allora decrementa parte alta
          MOVF      L_CONT,W    ;Carica in W la parte bassa
          IORWF     H_CONT,W    ;Mettila in OR con la parte alta
          BTFSS     STATUS,Z    ;Se tutto zero skip (fine ciclo)
          GOTO      DELAY2      ;Altrimenti ritorna a DELAY2
          RETURN
;-----------------------------------------------------
          END
```

Qui sotto sono riportati i flowchart completi della sezione principale del programma (chiamata anche main) e della subroutine delay. A fianco del disegno sono riportate le istruzioni che compongono le varie parti. Come si può vedere la chiamata alla subroutine si rappresenta con un rettangolo con aggiunte due righe laterali.





Il programma è completo e funzionante, tuttavia è possibile una piccola ottimizzazione della struttura IF THEN ELSE. Infatti nel caso cui vi sia una sola istruzione da eseguire sotto il THEN e una sola sotto l'ELSE la struttura può essere scritta in modo più compatto risparmiando una istruzione e la necessità di dover definire due etichette a cui saltare, da così:

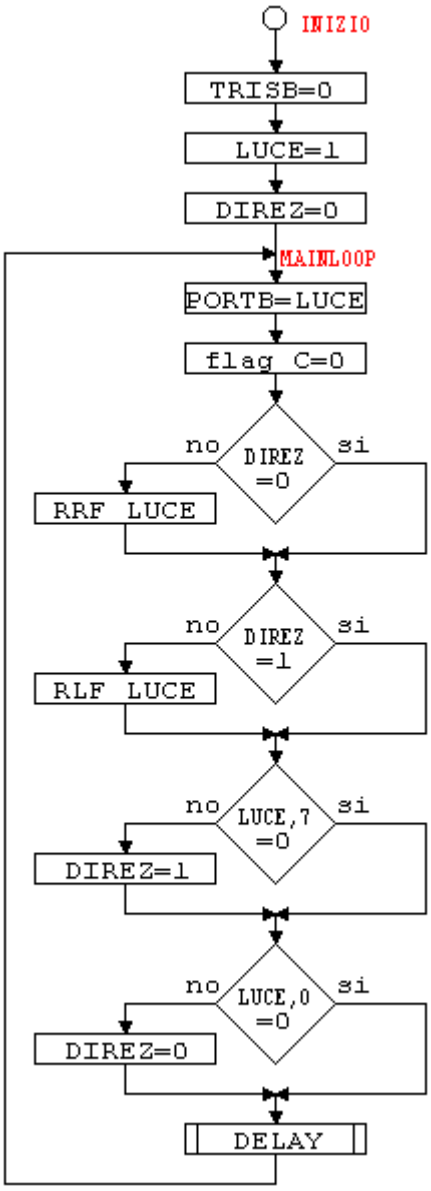
```
            BTFSC     DIREZ,0
            GOTO      LAB1
            RLF       LUCE,F
            GOTO      LAB2
LAB1        RRF       LUCE,F
LAB2        ......
```

a così:

```
            BTFSC    DIREZ,0
            RRF      LUCE,F
            BTFSS    DIREZ,0
            RLF      LUCE,F
```

Il main del nostro programma supercar diventerebbe cioè:

```
            BSF       STATUS,RP0  ;Attiva banco 1
            CLRF      TRISB       ;Rende PORTB un'uscita
            BCF       STATUS,RP0  ;Ritorna al banco 0
            MOVLW     00000001B
            MOVWF     LUCE        ;LUCE=00000001
            CLRF      DIREZ       ;DIREZ=0
MAINLOOP    MOVF      LUCE,W
            MOVWF     PORTB       ;Scrive LUCE sulla PORTB
            BCF       STATUS,C    ;Azzera flag C
            BTFSC     DIREZ,0     ;Se DIREZ=0 (sinistra) skip
            RRF       LUCE,F      ;Ruota LUCE verso destra
            BTFSS     DIREZ,0     ;Se DIREZ=1 (destra) skip
            RLF       LUCE,F      ;Ruota LUCE verso sinistra
            BTFSC     LUCE,7      ;Se bit 7 di LUCE ? 0 skip
            INCF      DIREZ,F     ;altrimenti direzione=destra
            BTFSC     LUCE,0      ;Se bit 0 di luce ? 0 skip
            CLRF      DIREZ       ;altrimenti direzione=sinistra
            CALL      DELAY       ;Richiama subroutine di ritardo
            GOTO      MAINLOOP    ;Nuovo ciclo del programma
```



Lo stesso sistema può essere usato per «spostare» o meglio «copiare» un singolo bit di un qualsiasi registro in un altro bit di un qualsiasi altro registro. Nell'esempio seguente copiamo il valore del flag C nel bit 4 del registro PIPPO:

```
BTFSC     STATUS,C
BSF       PIPPO,4
BTFSS     STATUS,C
BCF       PIPPO,4
```

La sequenza appena vista va bene anche per trasferire il valore di un singolo bit della RAM su un singolo pin di uscita di una porta. Se si lavora invece solo con registri in memoria, come in questo questo caso specifico, è possibile in realtà una ulteriore riduzione di istruzioni. Basta impostare inizialmente il bit di PIPPO ad un valore specifico, per esempio 0, e poi cambiandolo subito dopo SOLO SE il flag C è settato:

```
BCF       PIPPO,4
BTFSC     STATUS,C
BSF       PIPPO,4
```

Per concludere va ricordato che tutte le istruzioni viste in precedenza duravano sempre un ciclo macchina. Le istruzioni viste in questo capitolo invece possono

durare anche 2 cicli macchina. Per la precisione le istruzioni di salto, chiamata e ritorno (GOTO, CALL, RETURN, RETLW, RETFIE) richiedono sempre 2 cicli macchina, mentre gli skip ne richiedono 2 solo se la condizione è verificata.

ISTRUZIONI DI CONTROLLO SISTEMA

```
CLRWDT     Azzera watch dog timer
SLEEP      Standby mode
NOP        Nessuna operazione
```

Per completare la lista delle 35 istruzioni eseguibili dai PIC mancano le tre qui sopra. La prima serve per azzerare il contatore WDT, un registro hardware che incrementa automaticamente e, se abilitato, produce un reset del PIC quando arriva all'overflow (rollover). Questo serve in applicazioni critiche in cui è necessario che il PIC riparta automaticamente in caso di blocco accidentale del programma (crash) per esempio a causa di un forte disturbo elettrico. Nel caso in cui il WDT sia abilitato, nel programma si dovrà periodicamente effettuare un CLRWDT prima che il tempo scada.

La seconda manda in sleep il micro, che sospende ogni attività ed entra in modalità risparmio di energia. Per uscire dalla condizione di sleep il micro va resettato oppure deve ricevere una richiesta di interrupt nel caso in cui gli interrupt siano stati abilitati.

Della tabella rimane ancora la curiosa istruzione NOP. Come indicato nella descrizione questa istruzione non esegue nessuna funzione. E' una follia dei progettisti? Niente affatto, ogni microprocessore dispone di una istruzione NOP, che serve fondamentalmente per far passare del tempo in modo controllato senza influire su alcun altro registro. Come ogni altra istruzione anche una NOP richiede un ciclo macchina per essere eseguita, a 4 MHz un ciclo macchina dura 1 uS, pertanto l'inserzione di una NOP in un punto qualsiasi del programma determina un ritardo di 1 uS tra la fine dell'istruzione precedente e l'inizio di quella successiva. Ora, 1 uS di ritardo è poca cosa, se però una o più NOP vengono racchiuse in un ciclo che le esegue centinaia o migliaia di volte si possono ottenere ritardi precisi di qualsiasi durata.

E con questo la lista delle operazioni elementari è terminata, ora si tratta solo di vedere molti esempi su come possono essere «combinate» assieme per realizzare i compiti più disparati. Chi conosce già l'assembly di altri tipi di microprocessore noterà la mancanza di alcune cose, come un comodo stack su cui salvare i propri registri di lavoro, o l' assenza di «registri indice» che permettono l'accesso alle celle della ram puntandole da programma con un indirizzo contenuto in un altro registro. In realtà quest'ultima possibilità c'è, ma, come per altre cose in un microcontroller, la sua potenza è ridotta. E' un'operazione comunque possibile che è bene conoscere perché molto importante.

Indirizzamento RAM tramite puntatore

I PIC dispongono di un registro di nome FSR che funziona da «indirect memory data pointer». Un valore scritto in questo registro viene considerato come un indirizzo della RAM. Per scrivere o leggere a questo indirizzo così «puntato» da FSR basta scrivere o leggere nel registro INDF. Un esempio chiarisce meglio il concetto:

```
        MOVLW   32
        MOVWF   FSR     ;Punta la cella di indirizzo 32
        MOVLW   255
        MOVWF   INDF    ;Ci scrive 255


        MOVLW   32
        MOVWF   FSR     ;Punta la cella di indirizzo 32
        MOVF    INDF,W  ;Legge in W il suo contenuto
```

Siccome il valore caricato in FSR può anche essere incrementato o decrementato, è possibile gestire una certa sezione della RAM come un array, cioè una lista di byte raggiungibili con un indice progressivo senza la necessità di doverne definire un nome come si fa solitamente con gli altri registri. Un'area del genere può servire per memorizzare ad esempio dei numeri battuti su una tastiera o dei valori in arrivo da una porta seriale. Una sezione di memoria usata in questo modo viene anche chiamata «buffer». Il codice seguente azzera 10 byte a partire dall'indirizzo RAM 40 (usa il registro PIPPO come contatore di ciclo):

```
        MOVLW   40
        MOVWF   FSR      ;Punta la cella di indirizzo 40
        MOVLW   10
        MOVWF   PIPPO    ;Carica 10 in PIPPO
NEXT    CLRF    INDF     ;Azzera la cella RAM puntata da FSR
        INCF    FSR,F    ;Incrementa il puntatore
        DECFSZ  PIPPO,F  ;Se fatti 10 cicli termina
        GOTO    NEXT     ;altrimenti torna a next
```

RIEPILOGO ISTRUZIONI

Il valore di **d** può valere 0 o 1 (o, rispettivamente, W o F), **reg** indica l'indirizzo di un registro dati, **addr** indica un indirizzo di programma nelle istruzioni GOTO e CALL, **b** indica un bit all'interno di un byte (0..7), **n** è un valore costante (literal) a 8 bit (0..255).

Opcode	operando	Cyc.	Flags	Descrizione
ADDWF	reg,d	1	Z C	d= W + (reg)
ADDLW	n	1	Z C	W = W + n
ANDWF	reg,d	1	Z	d = W AND (reg)
ANDLW	n	1	Z	W = W AND n
BCF	reg,b	1		Bit b di (reg) = 0
BSF	reg,b	1		Bit b di (reg) = 1
BTFSC	reg,b	1(2)		Skip se bit b di (reg) = 0
BTFSS	reg,b	1(2)		Skip se bit b di (reg) = 1
CALL	addr	2		Chiamata di subroutine
CLRF	reg	1	Z=1	(reg) = 0
CLRW		1	Z=1	W = 0
CLRWDT		1		Azzera watch dog timer
COMF	reg,d	1	Z	d = NOT (reg)
DECF	reg,d	1	Z	d = (reg) - 1
DECFSZ	reg,d	1(2)		d = (reg) - 1 Skip se d = 0

```
GOTO     addr       2          Salto all'indirizzo addr
INCF     reg,d      1    Z     d = (reg) + 1
INCFSZ   reg,d     1(2)        d = (reg) + 1   Skip se d = 0
IORLW    n          1    Z     W = W OR n
IORWF    reg,d      1    Z     d = W OR (reg)
MOVF     reg,d      1    Z     d = (reg)
MOVLW    n          1          W = n
MOVWF    reg        1          (reg) = W
NOP                 1          Nessuna operazione
RETFIE              2          Ritorno da interrupt
RETLW    n          2          Ritorno da subr. con valore in W
RETURN              2          Ritorno da subroutine
RLF      reg,d      1      C   d = rlf (reg)
RRF      reg,d      1      C   d = rrf (reg)
SLEEP               1          Standby mode
SUBLW    n          1    Z C   W = n - W
SUBWF    reg,d      1    Z C   d = (reg) - W
SWAPF    reg,d      1          d = swap (reg)
XORLW    n          1    Z     W = W XOR n
XORWF    reg,d      1    Z     d = W XOR (reg)
```

[Segue]

Pagina creata nel gennaio 2004 - Ultimo aggiornamento 25-3-2006