CENNI SUL LINGUAGGIO PIC-ASM

Il linguaggio PIC-ASM che utilizzeremo nei nostri programmi è un assembler dedicato ai microcontrollori PIC.

Nella famiglia PIC16 esso si compone di 35 istruzioni in grado di operare su byte, su bit, su literal oppure sul funzionamento del MCU.

La sintassi è semplice, ed è ordinata nel modo seguente:

```
comando
comando operando
comando operando,destinazione
comando operando,bit
```

Al fine di gestire in modo coerente e comprensibile il codice, si utilizzano alcune convenzioni, in parte anche richieste dallo stesso programma assemblatore.

Ad esempio, nella prima colonna di ogni riga NON si scrivono comandi, ma solo etichette del codice (necessarie per eseguire i salti).

Per comodità, nei programmi presentati si lasciano libere da codice le prime otto colonne.

E' altresì evidente che i commenti non possono mancare. Certamente si può essere meno prolissi, ma è consigliabile esagerare piuttosto che omettere qualcosa di importante.

Oltre alle istruzioni PIC-ASM, è possibile utilizzare delle “macro”, cioè sequenze di istruzioni che possono essere poi scritte come un unico comando. E' da tenere presente, però, che in fase di compilazione tutte le macro vengono sostituite con il relativo codice esteso (questo può creare problemi se si utilizza l'indirizzamento indiretto del codice).

Nei programmi didattici si è evitato di utilizzare costrutti potenzialmente complessi, come ad esempio l'indirizzamento indiretto del codice, e si è cercato di limitare al massimo quello delle variabili.
Come esempio, si riporta una breve porzione di codice didattico e una possibile equivalenza (leggermente estremizzata) di quanto si può trovare anche su application notes del produttore:

DIDATTICO

```
PROGRAM:
;1 ciclo=1,25us
;program: 1 ciclo + loop2 + 5 o 6 cicli = 195846 / 195847 cicli ~ 245
ms
      MOVLF 0xFF,b_CONT1            ;carica contatore 1 (65535)
;loop2: (loop1 + 3 cicli) * 255 = 195840 cicli
LOOP2 MOVLF 0xFF,b_CONT2            ;carica contatore 2
;loop1: 3 cicli * 255 = 765
LOOP1 DECFSZ b_CONT2                ;decrementa contatore 2
      GOTO LOOP1                    ; cicla finchè contatore 2 > 0
      DECFSZ b_CONT1                ;decrementa contatore 1
      GOTO LOOP2                    ; cicla finchè contatore 1 > 0
      BTFSS c_LED_PORT,c_LED_PIN    ;controlla stato led
      GOTO LED_CLR                  ; clear
LED_SET BCF c_LED_PORT,c_LED_PIN    ; set-->clear
      GOTO PROGRAM                  ;torna a inizio ciclo
LED_CLR BSF c_LED_PORT,c_LED_PIN    ; clear-->set
      GOTO PROGRAM                  ;torna a inizio ciclo
      END
```

PRATICO

```
PROGRAM:
      MOVLF 0xFF,INDF
      INCF FSR,F
      MOVLF 0xFF,INDF
      DECFSZ INDF
      GOTO $-1
      DECF FSR,F
      DECFSZ INDF
      GOTO $-7
      BTFSS c_LED_PORT,c_LED_PIN
      GOTO $+3
      BCF c_LED_PORT,c_LED_PIN
      GOTO $-13
      BSF c_LED_PORT,c_LED_PIN
      GOTO $-15
      END
```

In questo caso si può anche notare che, pur essendo scritte come unica istruzione, le macro vanno conteggiate a seconda del numero di istruzioni che le compongono.

Dovrebbe risultare subito evidente cosa fa il codice di sinistra, mentre il codice di destra è di difficile lettura.

GESTIONE DELLE VARIABILI

L'allocazione delle variabili si effettua scrivendo il seguente codice:

```
nome1 EQU H'n'
nome2 EQU H'n+1'
```

dove “nome” è il nome della variabile, e “n” è l'indirizzo di memoria in esadecimale.

Tipicamente si utilizzano dati di dimensioni 8bit (BYTE), 16bit (WORD), 32bit (LONG).
Quando si opera su WORD, si considera f=MSB ed f+1=LSB
Quando si opera su LONG, si considera f=MSB ed f+3=LSB

L'istruzione di allocazione opera sui byte, quindi se si vogliono allocare dati di dimensioni maggiori occorre tenerne conto nell'indirizzamento:

```
nome1 EQU H'21'   (word)
nome2 EQU H'23'   (byte)
nome3 EQU H'24'   (long)
nome4 EQU H'28'   (byte)
```

Al fine di identificare correttamente i tipi di dati, si utilizza una nomenclatura così composta:

`a_B_C`

- “`a`” (sempre in minuscolo) può essere:
  `b` indica un dato variabile di tipo BYTE
  `w` indica un dato variabile di tipo WORD
  `l` indica un dato variabile di tipo LONG
  `c` indica un LITERAL
  `f` indica un FLAG

  Per le variabili multiuso e/o componibili generalmente si omette il prefisso.
  Ad esempio, `R1` è una variabile generica WORD, composta da `R1H` ed `R1L` (BYTE). Risulterebbe fuorviante dichiarare `w_R1` se poi si utilizzano i singoli bytes, come d'altra parte dichiarare `b_R1H` e `b_R1L` se poi si utilizzano come MSB ed LSB di una word.

- “`B`” (sempre in maiuscolo) indica il campo di utilizzo del dato, ad esempio “`SW`” indicherà la gestione della tastiera, “`TMR`” la gestione dei timer, ecc...

- “`C`” (sempre in maiuscolo) è il nome del dato. Si possono utilizzare nomi di fantasia, ma è preferibile avere un riferimento all'utilizzo della variabile

Quindi, ad esempio, `b_SW_CTRL` è una variabile di tipo BYTE, utilizzata nella gestione tastiera, e presumibilmente contiene i flag di controllo dei pulsanti.

Le variabili “globali” si possono indicare semplicemente nella forma `a_C`.

Un pin di I/O si definisce invece come una tripletta di costanti `c_B_C_D`, in cui “`D`” indicherà rispettivamente `PORT`, `TRIS` e `PIN` ,mentre per “`B`” e “`C`” vale quanto detto in precedenza. In questo modo, per modificare l'assegnazione dei singoli I/O non occorre andarli a cercare all'interno del codice, ma è sufficiente aprire il file delle costanti e modificare i valori una sola volta.
Esempio:

```
#define c_SW_STST_TRIS    TRISA          ;pin del led
#define c_SW_STST_PORT    PORTA
#define c_SW_STST_PIN     2
```

Tra l'altro, in questo modo si può anche ovviare a dubbi relativi al BANK di appartenenza dei registri. Cioè, se non so in che banco si trova, per ipotesi, il TRIS del pin, scriverò semplicemente `BANKSET c_SW_STST_TRIS` e non avrò più dubbi.

Sebbene questo sistema possa apparire poco utile nei programmi didattici, esso diventa molto utile quando si gestiscono decine di dati differenti in programmi più complessi. Vale sempre la regola che, una volta imparato un metodo, è preferibile utilizzarlo sempre.

ISTRUZIONI DELLA FAMIGLIA PIC16

TABLE 15-2: PIC16F627A/628A/648A INSTRUCTION SET

Mnemonic, Operands		Description	Cycles	14-Bit Opcode MSb			LSb	Status Affected	Notes
BYTE-ORIENTED FILE REGISTER OPERATIONS									
ADDWF	f, d	Add W and f	1	00	0111	dfff	ffff	C,DC,Z	1, 2
ANDWF	f, d	AND W with f	1	00	0101	dfff	ffff	Z	1, 2
CLRF	f	Clear f	1	00	0001	lfff	ffff	Z	2
CLRW	—	Clear W	1	00	0001	0xxx	xxxx	Z	
COMF	f, d	Complement f	1	00	1001	dfff	ffff	Z	1, 2
DECF	f, d	Decrement f	1	00	0011	dfff	ffff	Z	1, 2
DECFSZ	f, d	Decrement f, Skip if 0	1(2)	00	1011	dfff	ffff		1, 2, 3
INCF	f, d	Increment f	1	00	1010	dfff	ffff	Z	1, 2
INCFSZ	f, d	Increment f, Skip if 0	1(2)	00	1111	dfff	ffff		1, 2, 3
IORWF	f, d	Inclusive OR W with f	1	00	0100	dfff	ffff	Z	1, 2
MOVF	f, d	Move f	1	00	1000	dfff	ffff	Z	1, 2
MOVWF	f	Move W to f	1	00	0000	lfff	ffff		
NOP	—	No Operation	1	00	0000	0xx0	0000		
RLF	f, d	Rotate Left f through Carry	1	00	1101	dfff	ffff	C	1, 2
RRF	f, d	Rotate Right f through Carry	1	00	1100	dfff	ffff	C	1, 2
SUBWF	f, d	Subtract W from f	1	00	0010	dfff	ffff	C,DC,Z	1, 2
SWAPF	f, d	Swap nibbles in f	1	00	1110	dfff	ffff		1, 2
XORWF	f, d	Exclusive OR W with f	1	00	0110	dfff	ffff	Z	1, 2
BIT-ORIENTED FILE REGISTER OPERATIONS									
BCF	f, b	Bit Clear f	1	01	00bb	bfff	ffff		1, 2
BSF	f, b	Bit Set f	1	01	01bb	bfff	ffff		1, 2
BTFSC	f, b	Bit Test f, Skip if Clear	1(2)	01	10bb	bfff	ffff		3
BTFSS	f, b	Bit Test f, Skip if Set	1(2)	01	11bb	bfff	ffff		3
LITERAL AND CONTROL OPERATIONS									
ADDLW	k	Add literal and W	1	11	111x	kkkk	kkkk	C,DC,Z	
ANDLW	k	AND literal with W	1	11	1001	kkkk	kkkk	Z	
CALL	k	Call subroutine	2	10	0kkk	kkkk	kkkk		
CLRWDT	—	Clear Watchdog Timer	1	00	0000	0110	0100	$\overline{TO}$,$\overline{PD}$	
GOTO	k	Go to address	2	10	1kkk	kkkk	kkkk		
IORLW	k	Inclusive OR literal with W	1	11	1000	kkkk	kkkk	Z	
MOVLW	k	Move literal to W	1	11	00xx	kkkk	kkkk		
RETFIE	—	Return from interrupt	2	00	0000	0000	1001		
RETLW	k	Return with literal in W	2	11	01xx	kkkk	kkkk		
RETURN	—	Return from Subroutine	2	00	0000	0000	1000		
SLEEP	—	Go into Standby mode	1	00	0000	0110	0011	$\overline{TO}$,$\overline{PD}$	
SUBLW	k	Subtract W from literal	1	11	110x	kkkk	kkkk	C,DC,Z	
XORLW	k	Exclusive OR literal with W	1	11	1010	kkkk	kkkk	Z	

Note 1: When an I/O register is modified as a function of itself (e.g., `MOVF PORTB, 1`), the value used will be that value present on the pins themselves. For example, if the data latch is '1' for a pin configured as input and is driven low by an external device, the data will be written back with a '0'.

2: If this instruction is executed on the TMR0 register (and, where applicable, d = 1), the prescaler will be cleared if assigned to the Timer0 Module.

3: If Program Counter (PC) is modified or a conditional test is true, the instruction requires two cycles. The second cycle is executed as a NOP.

Field	Description
f	Register file address (0x00 to 0x7F)
W	Working register (accumulator)
b	Bit address within an 8-bit file register
k	Literal field, constant data or label
x	Don't care location (= 0 or 1) The assembler will generate code with x = 0. It is the recommended form of use for compatibility with all Microchip software tools.
d	Destination select; d = 0: store result in W, d = 1: store result in file register f. Default is d = 1
$\overline{TO}$	Time-out bit
$\overline{PD}$	Power-down bit

MACRO AGGIUNTIVE UTILIZZATE

MACRO	***DESCRIZIONE***
`CLI`	Disabilita Interrupts
`SEI`	Abilita Interrupts
`PUSH`	Salvataggio registri W,STATUS,FSR,PCLATH
`POP`	Ripristino registri W,STATUS,FSR,PCLATH
`BANKSET k`	Selezione RP (Indirizzamento diretto)
`BANKISET k`	Selezione IRP (Indirizzamento Indiretto)
`BANK_x`	Imposta indirizzo banco di memoria per allocazione variabili
`BYTE label`	Da usare assieme a BANK_x, alloca automaticamente un byte. Equivale a `label EQU H'n'` con variabile successiva in n+1
`WORD label`	Da usare assieme a BANK_x, alloca automaticamente una word (16 bit). Equivale a `label EQU H'n'` con variabile successiva in n+2
`LONG label`	Da usare assieme a BANK_x, alloca automaticamente un long (32 bit). Equivale a `label EQU H'n'` con variabile successiva in n+4
`ANDFF f1,f2`	AND tra due registri, risultato in f1
`MOVFF f1,f2`	Copia f1 in f2
`MOVLF k,f`	Scrive il valore di k in f (byte)
`MOVL2W k,f`	Scrive il valore di k in f (word)
`DECW f`	Decrementa word
`INCW f`	Incrementa word
`ADDWB f1,f2`	Somma un byte f2 ad una word f1
`ADDW f1,f2`	Somma tra due word, risultato in f1
`ADDWL f,k`	Somma un literal ad una word
`SUBW f1,f2`	Sottrazione tra due word, risultato in f1
`SUBWL f,k`	Sottrae un literal ad una word
`CMPB f1,f2`	Confronta due byte (modifica i flag C,DC,Z)
`CMPBL f,k`	Confronta byte e literal(modifica i flag C,DC,Z)
`CMPW f1,f2`	Confronta due byte (modifica i flag C,DC,Z)
`B k *(macro di sistema)`	Salto incondizionato a k
`BC k *(macro di sistema)`	Salta a k se il flag C è settato
`BDC k *(macro di sistema)`	Salta a k se il flag DC è settato
`BNC k *(macro di sistema)`	Salta a k se il flag C non è settato
`BNDC k *(macro di sistema)`	Salta a k se il flag DC non è settato
`BNZ k *(macro di sistema)`	Salta a k se il flag Z non è settato
`BZ k *(macro di sistema)`	Salta a k se il flag Z è settato
`BEQ k`	Dopo un confronto (CMPB/BL/W), salta a k se uguali
`BNE k`	Dopo un confronto (CMPB/BL/W), salta a k se diversi
`BG k`	Dopo un confronto (CMPB/BL/W), salta a k se operando 1 maggiore
`BGE k`	Dopo un confronto (CMPB/BL/W), salta a k se operando 1 maggiore o uguale
`BL k`	Dopo un confronto (CMPB/BL/W), salta a k se operando 1 minore
`BLE k`	Dopo un confronto (CMPB/BL/W), salta a k se operando 1 minore o uguale