Esercitazioni PIC Midrange.

Inserto_2: i simboli degli operatori aritmetici e bitwise.

Il Macro Assembler MPASM consente l'uso di numerosi simboli che rappresentato operatori aritmetici e logici.
Il loro uso non è sempre comune negli esempi per principianti visibili nel WEB; vediamo di introdurne e chiarirne l'uso.

AVVERTENZA IMPORTANTE:

Gli operatori logici aritmetici dell' Assembler, assieme alle direttive condizionali (**if**, **while**, ecc), anche se simili a quelli del C, **NON sono codici di istruzione o generano codici di istruzione !**

Questi operatori, nell' Assembly, servono per realizzare assemblaggi condizionali o calcolare costanti.

NON agiscono sulla logica del programma, ma sulle operazioni svolte durante l'assemblaggio e non trasformano il Macro Assembler in un linguaggio C.

Operatori aritmetici.

MPASM supporta i seguenti operatori aritmetici:

Operatore	Simbolo	Funzione
somma	+	Somma aritmetica
sottrazione	-	Sottrazione aritmetica
divisione	/	Divisione
moltiplicazione	*	Moltiplicazione
modulo	%	Modulo del valore
negazione	-	Negazione del valore

Si tratta delle comuni operazioni aritmetiche, per le quali vale la precedenza di moltiplicazione/divisone su somma e sottrazione. L'uso delle parentesi consente di adeguare la sequenza delle operazioni.
Così:

```
3 + 3 * 5 = 3 + 15 = 45
```

mentre:

```
(3 + 3)*5 = 6 * 5 = 30
```

Potremo, quindi, delegare all'Assembler la gestione di molti tipi di calcoli.
Sono possibili calcoli di ogni genere, anche di formule complesse, nei limiti delle possibilità dell'Assembler.

Ad esempio, calcoli relativo al clock. Gli elementi calcolati sono resi sotto forma di label per essere utilizzati o elaborati ulteriormente durante il programma.

```
; CLOCK frequency = internal osc. - default al POR
XTAL_FREQ equ 1000000           ; INTIO1 1 MHz
CLOCK     equ XTAL_FREQ/4       ; processor clock [Hz]
TCYC      equ 1000000000/CLOCK ; cycle time [ns]
```

Oppure calcoli relativi a **baudrate** (USART), al **clock di Timer1** e a quello dell' **I2C** (MSSP).

```
; calculates baudrate when BRGH = 1, adjust for rounding errors
#define CALC_HIGH_BAUD(BaudRate)(((10*XTAL_FREQ/(16*BaudRate))+5)/10)-1
; calculates baudrate when BRGH = 0, adjust for rounding errors
#define CALC_LOW_BAUD(BaudRate)(((10*XTAL_FREQ/(64*BaudRate))+5)/10)-1

; calculates timer1 delay when prescale is 1:8, TcikTime in msec
#define CALC_TIMER1(TickTime) (0xFFFF-((TickTime*XTAL_FREQ)/32000))+1

; used for I2C calculations
#define I2CClock D'100000' ; define I2C bite rate
#define I2C_ClockValue (((XTAL_FREQ/I2CClock)/4) -1)
```

....

```
; esempio di uso
  movlw LOW(CALC_TIMER1(D'100'))
  movwf TMR1L                   ; initialize Timer1 low
  movlw HIGH(CALC_TIMER1(D'100'))
  movwf TMR1H                   ; initialize Timer1 high
```

e per il modulo PWM

```
PWM_Period = (PR2 + 1)*4*Tosc*TMR2_PreScale_value
PR2 = (PWM_Period/(4 * Tosc * TMR2 Prescale)) - 1
PWM_Duty_Cycle = (CCPR1L_CCP1CON5_4)*Tosc*TMR2_Prescale_Value
```

Oppure **settaggio di paramenti** :

```
DEST_HIGH   SET (HIGH(LABEL)&0x18)      ; save bit's 4:5 of dest adress
SOURCE_HIGH SET (HIGH($)&0x18)          ; source adress
DIFF_HIGH   SET DEST_HIGH ^ SOURCE_HIGH ; get difference (XOR)
```

Esempio di

Possiamo anche far determinare il valore per **generare un warning:**

```
  #IF ((DIFF_HIGH&0x18)==0x18)
   MESSG " WARNING ! Replace SHORT_CALL with LONG_CALL " LABEL
  #ENDIF
```

Operatori bitwise.

Operatore	Funzione	Esempio
!	NOT	`if ! (a == b)`
-	Negazione (complemento a 2)	`-1 * Length`
~	Complemento	`flags = ~flags`
<<	Left Shift	`flags = flags << 1`
>>	Right Shift	`flags = flags >> 1`
&	Bitwise AND	`flags = flags & ERROR_BIT`
^	Bitwise XOR	`flags = flags ^ ERROR_BIT`
|	Bitwise OR	`flags = flags | ERROR_BIT`
<<=	Ledt shift and set	`flags <<= 3`
>>=	Right shift and set	`flags >>= 3`
&=	AND and set	`flags &= ERROR_FLAG`
|=	OR and set	`flags |= ERROR_FLAG`
^=	XOR and set	`flags ^= ERROR_FLAG`

Si tratta di operatori logici per settare o azzerare singoli bit o gruppi di bit, posizionare bit, testarne lo stato, ecc.

Vediamo alcune particolarità.

Operatore eguaglianza.

Osserviamo la presenza di alcuni simboli che sembrano indicare la stessa operazione, ma in realtà sono previsti per due situazioni differenti.

Usiamo **=** per indicare l'equivalenza, ad esempio, tra una label ed un valore o due label:

```
TOTAL = 5 * 2 + 1
```

```
TOTAL = NewValue
```

ma dovremo usare **==** se si tratta di una comparazione:

```
#IF (Value == 0x27)
 clrf  Target
#ENDIF
```

Operatore AND &.

& esegue l'AND tra due o più valori, bit per bit. Ne abbiamo un esempio ogni volta in cui scriviamo la *configuration word.*

Se verifichiamo il file *nomeprocessore.inc* dalla directory in cui è installato MPASM, troviamo una situazione del genere (esempio per 10F320):

```
; The following is an assignment of address values for all of the
; configuration registers for the purpose of table reads
_CONFIG          EQU  H'FFF'
;----- CONFIG Options --------------------------------------------------
_IOSCFS_4MHZ         EQU  H'0FFE'    ; 4 MHz
_IOSCFS_8MHZ         EQU  H'0FFF'    ; 8 MHz
_IOFSCS_4MHZ         EQU  H'0FFE'    ; 4 MHz
_IOFSCS_8MHZ         EQU  H'0FFF'    ; 8 MHz
_MCPU_ON             EQU  H'0FFD'    ; Pull-up enabled
_MCPU_OFF            EQU  H'0FFF'    ; Pull-up disabled

_WDTE_OFF            EQU  H'0FFB'    ; WDT disabled
_WDT_OFF             EQU  H'0FFB'    ; WDT disabled
_WDTE_ON             EQU  H'0FFF'    ; WDT enabled
_WDT_ON              EQU  H'0FFF'    ; WDT enabled

_CP_ON               EQU  H'0FF7'    ; Code protection on
_CP_OFF              EQU  H'0FFF'    ; Code protection off

_MCLRE_OFF           EQU  H'0FEF'    ; GP3/MCLR pin function is digital
I/O, MCLR internally tied to VDD
_MCLRE_ON            EQU  H'0FFF'    ; GP3/MCLR pin function is MCLR
```

Quando impostiamo la configurazione nella forma classica:

```
__config  _IOSCFS_4MHZ & _MCPU_ON & _WDT_OFF & _CP_OFF & _MCLRE_OFF
```

stiamo comandando al compilatore di effettuare un AND `&` tra le varie label. Dato che le label sottintendono dei valori assoluti, è come se stessimo operando:

```
H'0FFE' and H'0FFD' and H'0FFB' and H'0FFF' and H'0FEF'
```

I numeri binari:

```
_IOSCFS_4MHZ  H'0FFE'  00 1111 1111 1110
_MCPU_ON      H'0FFD'  00 1111 1111 1101
_WDT_OFF      H'0FFB'  00 1111 1111 1011
_CP_OFF       H'0FFF'  00 1111 1111 1111
_MCLRE_OFF    H'0FEF'  00 1111 1110 1111
                       -------------------
```

il cui **AND** è

```
00 1111 1110 1000
0    F    E    8
```

Questo valore viene utilizzato dal comando **__config** come oggetto da scrivere nella locazione di configurazione.
Da qui deriva la pratica errata di configurare con un comando come:

```
__config H'0FE8'
```

cosa da evitare in quanto rende difficoltoso comprendere quale sia la configurazione necessaria al programma e altrettanto difficile identificare se in questa ci sono errori. (ved. anche l'articolo La configurazione nei PIC).

Operatore XOR ^ .

Va ricordato che l'OR esclusivo tra due valori rende 0 se i bit sono uguali.

```
  movlw  0xFF ^ b'01000101'

; this is equal to

  movlw  0xBAh
```

Una trattazione della funzione la trovate **qui**.

Operatore di shift (>> e <<).

Esegue lo shift di un bit (0 o 1) di n volte.

La scrittura:

```
movlw ((1 << GIE) | (1 << TOIE))
movwf INTCON
```

Equivale a:

```
movlw   b'10100000'
movwf   INTCON
```

Si nota come nel primo caso l' uso delle lablel al posto di numeri assoluto facilita la comprensione e la stesura del sorgente; infatti si sfrutta l' assegnazione presente nel *nomeprocessore.ini* in cui è dichiarato **GIE = 7** e **T0IE = 5**, non necessitando così di altra dichiarazione nel sorgente.

La stessa operazione sarebbe eseguibile con una somma :

```
movlw (.128 + .32)   ; move b'10100000'
movwf INTCON
```

Operatore Complemento (~).

Questo operatore effettua il complemento dell'oggetto.

La scrittura: Equivale a:

```
    movlw    ~(0x55)
```
e
```
    movlw    ~(1<<1)
```
Mentre:
```
    movlw    (1<<1)
```
equivale a:
```
    movlw    0xAA
```
equivale a:
```
    movlw    b'11111101'
```
equivale a:
```
    movlw    b'00000010'
```

Possiamo combinare vari operatori:

La scrittura:
```
 andlw  ~((1<<0) | (1<<2) | (1<<6))
```
Equivale a:
```
    andlw  b'10111010'
```
Mentre:
```
 andlw  ((1<<0) | (1<<2) | (1<<6))
```
equivale a:
```
    movlw  b'01000101'
```

La cosa è utilizzabile per il setup degli SFR
La scrittura:
```
  movlw   ~(1<<LEDOut)
  movwf   TRISB
```
Equivale a:
```
    bcf     TRISB, LEDOut
```

ed è utilizzabile per il toggle di bit:

```
  xorlw   1<<LEDOut       ; toggle pin LEDOut
```

Per più bit:
La scrittura:
```
  movlw   ~(1<<GP1|1<<GP2)
```
Equivale a:
```
    movlw   b'11111001'
```

E' possibile usare label:
La scrittura:
```
  movlw   1<<T0CS|0<<PSA|b'110'
  movwf   OPTION_REG
```
Equivale a:
```
   movlw   b'00100110'
   movwf   OPTION_REG
```

La scrittura:
```
  movlw    1<<NOT_GPPU | 1<<INTEDG | 0<<T0CS | 1<<T0SE | 0<<PSA | b'111'
  movwf   OPTION_REG
```
equivale a:
```
  movlw   b'11010111'
  movwf   OPTION_REG
```

Se si è acquistata una certa dimestichezza con queste funzioni, è possibile usarle correntemente al posto degli equivalenti classici.

Operatore incrementa/decrementa (++/--).

Il suo scopo è di incrementare (o decrementare una variabile), ad esempio in un loop per inserire una serie di istruzioni a seconda del valore di una variabile o di una costante.

```
; Loop for rotate register
some equ 4

LoopCounter = some        ; set counter
  while  LoopCounter >0
       rrncf  register
loopCounter --
  endw
```

AVVERTENZA IMPORTANTE:

E' da considerare che questo loop while riguarda non l' esecuzione del programma, ma la sua compilazione.

Fissata la variabile **some** , l' istruzione **rrncf register** viene ripetuta nel listato un equivalente numero di volte.

Operatori di confronto (>=, <=, <, >, ==, !=)

>=	Maggiore o uguale	`if entry_idx >= num_entries`
>	Maggiore	`if entry_idx > num_entries`
<	Minore	`if entry_idx < num_entries`
<=	Minore o uguale	`if entry_idx <= num_entries`
==	Uguale	`if entry_idx == num_entries`
!=	Non uguale	`if entry_idx != num_entries`

Una label può indicare una costante, ma può essere anche il risultato di una o più operazioni, dipendenti da altre variabili o condizioni.

```
; set some depending from environment
; conditions

  #if  ((UserTag >= 8) | (Debug = 1))
some = (UserTag - 4)
  #else
some = ((UserTag /2) + 1)
  #endif
```

AVVERTENZA IMPORTANTE:

Questa sequenza NON viene eseguita nel programma a seconda del valore assunto al momento da variabili durante l' esecuzione del programma.

Una sequenza condizionale come quella presentata qui sopra viene eseguita UNA sola volta al momento dell' assemblaggio, fornendo un listato differente a seconda dei valori impostati per UserTag e Debug.

Un altro esempio :

```
; check bits in a register

  while (i < 8)                ; for 8 times
    #if  ((Register & 1) != 0   ; if bit 0 is set
    .. instructions...
    #endif
; shift right for check next bit
    Register = Register >> 1
i = i + 1
  endw
```

Ecco un utile check automatico di superamento dello spazio durante le assegnazioni in RAM con lo statement CBLOK.
La label di una assegnazione "vuota" (dummy) che occupa 0 bytes viene attribuita all' ultima locazione di RAM assegnata; quindi viene verificato se il suo indirizzo supera quello del banco (in questo caso il Banco0, Access Ram). Se supera viene segnalato un errore.

```
CBLOCK 0x00   ; bank 0
    ssTXBuf
    ssRXCnt
    DlyTcyCnt   ; for DelayUS
    ...etc...

 end_AccBank0:0 ; dummy for overrun check
  ENDC

 #if end_Accbank0 > 0x7F ; check for overrun
   error "AccessRAM Bank0 space overrun"
 #endif
```

Con questa semplice aggiunta non c'è il rischio di aver assegnato memoria al di fuori dei limiti consentiti e non c'è bisogno di effettuare alcun calcolo : l' Assembler calcola per noi e ci avvisa dell' eventuale errore.

high	high byte	movlw high CTR_Table
low	low byte	movlw low CTR_Table
upper	upper byte	movlw upper CTR_Table

High/Low/Upper.

Questi operatori restituiscono il byte basso, alto e superiore di un numero (a 3 bytes) o di un indirizzo a 20/21 bit; sono di uso corrente nell'area dei PUC 18F o superiori, dove gli indirizzi o i numeri da trattare sono più ampi di 8 bit.

```
; Pointer to Table (over 64k)

  movlw low Table       ; low of address
  movwf save_low
  movlw high Table      ; high of address
  movwf save_mid
```

```
  movlw upper Table     ; upper of address
  movwf save_up
```

e anche :

```
; move byte from a multi byte label
LABEL = 0xFA32CB

  movlw low LABEL       ; low byte
  movwf save_low        ; save_low  =  CBh
  movlw high LABEL      ; high byte
  movwf save_high       ; save_high =  32h
  movlw upper Table     ; upper byte
  movwf save_up         ; now save_upper = Fah
```

Nei PIC16, che hanno limite di memoria programma inferiore agli Enhanced, la componente Upper non ha peso (indirizzamento <= 32K).

Si può utilizzare **high** e **low** per caricare numeri a 16 bit:

```
; load 2 bytes to TMR1
VALUE1 = 0xC00A

  movlw low VALUE       ; low byte
  movwf TMR1L           ; 0Ah
  movlw high VALUE      ; high byte
  movwf TMR1H           ; C0h
```