gohan

ASSEMBLY : THE CODEFATHER

26 August 2011

Questa non è una storia di mafia, nemmeno un tutorial di programmazione ma bensì un **articolo orientato all'Assembly**, il quale io definisco come il padre di tutti i linguaggi di programmazione ad alto livello moderni. Un paio di commenti in seguito al **mio precedente articolo** mi hanno dato uno stimolo a scriverlo, sperando che possa risultare interessante.

Il mio interesse per l'elettronica è nato dal mio studio autodidatta sulla programmazione a basso livello con l'architettura Intel x86. Nonostante i pochi programmi scritti nella fase di apprendimento, mi dava la sensazione di interagire direttamente con l'hardware, così un giorno mi venne in mente di creare un semplice programma in Assembly che potesse aprire il lettore CD/DVD del mio portatile. Quel programma purtroppo non è mai arrivato, ma è stato il trampolino di lancio per cominciare a studiare l'elettronica da un giorno all'altro. La curiosità di conoscere come funziona un microprocessore ed in seguito un hardware attraverso istruzioni mnemoniche è stato e lo è tutt'ora con i microcontrollori, un grande stimolo per continuare i miei studi. Ecco come ben presto ho avuto modo di conoscere un pò più da vicino resistenze e condensatori, transistori, operazionali, porte logiche ed altro ancora.

Ho così deciso di scrivere il mio secondo articolo parlando di questo linguaggio di programmazione a basso livello (dato che forse è merito suo se sono quì), mettendo alla luce alcuni dettagli importanti che lo contraddistinguono dai linguaggi ad alto livello, che io considero importanti ed utili per capire il funzionamento del architettura con cui si lavora.

Introduzione

Quando si è interessati a conoscere più a fondo un dispositivo (come quelli menzionati in precedenza), e capirne il funzionamento attraverso il proprio computer, la maniera migliore è (oltre a consultarne il datasheet) comunicare con questo attraverso il linguaggio base progettato dal fabbricante, ovvero programmandolo in Assembly, che a sua volta usa istruzioni mnemoniche ed operandi (1, 2 o in alcuni casi nessuno) che se non sono valori numerici sono altrettanto i propri registri del dispositivo (o locazioni di memoria). Saper programmare in questo linguaggio vuol dire saper intercambiare i dati da un registro all'altro in maniera corretta al

fine di garantire il corretto funzionamento del programma. Per esempio in questo frammento di codice Assembly per PIC

```
movlw 0xff
movwf TRISB
```

si evidenzia come il valore **0xff** viene prima caricato nel registro **W** attraverso l'istruzione **movlw** e successivamente salvato nel registro di configurazione **TRISB** relativo al **PORTB**. Diversamente, con un'istruzione C come potrebbe essere quella a seguire

```
TRISB = 0xff;
```

questo dettaglio viene fin da subito nascosto agli occhi del programmatore. Altri dettagli potrebbero essere il cambio di banco attraverso il registro STATUS, oppure l'inizializzazione dello stack con i microcontrollori AVR. Sono queste cose che mi spingono a pensare che è bene imparare a programmare (o almeno praticare un pò) in linguaggio a basso livello per l'architettura con cui si vuole lavorare (il quale è quello che in definitiva viene proposto principalmente dal fabbricante) al fine di capirne il più possibile il funzionamento. Per esempio a pagina 43 del datasheet di un **PIC16F877x** vediamo come l'unico esempio di codice di configurazione del PORTA, viene appunto rappresentato in Assembly. Se infine il fabbricante propone delle librerie differenti (i.e. C), esempi alternativi vengono altrettanto proposti.

Dalle istruzioni al codice binario

La definizione "basso livello" riferita ad un linguaggio di programmazione sta a significare che quest'ultimo ha delle caratteristiche che lo relazionano particolarmente all'architettura hardware per la quale è stato progettato e si vuole comunicare. Vediamo il perchè di questa definizione.

Si ricorda che l'unico linguaggio che un microprocessore ed un microcontrollore possono interpretare è il **codice macchina** e non l'Assembly. L'Assembler è l'intermediario fra questi due, ovvero il compilatore. Molte volte questo termine viene confuso con il nome del linguaggio. Per esporre un esempio chiaro, partiamo da un semplice frammento di output ottenuto con **objdump** (il disassemblatore GNU) dopo la compilazione di un semplice programma in C scritto per l'architettura x86.

```
:~$ objdump -D file.out

 80483b5:    89 e5                    mov    %esp,%ebp
 80483b7:    83 ec 10                 sub    $0x10,%esp
 80483ba:    c7 45 fc 0a 00 00 00     movl   $0xa,-0x4(%ebp)
```

```
 80483c1:    b8 00 00 00 00           mov    $0x0,%eax
 80483c6:    c9                       leave
 80483c7:    c3                       ret
 80483c8:    90                       nop
```

La colonna di sinistra visualizza locazioni di memoria riguardo alle istruzioni del programma. Nella colonna centrale possiamo invece distinguere il codice macchina mentre nella colonna di destra vengono riportate le relative istruzioni secondo la sintassi **AT&T**.

Si ricorda che questa si differenzia dalla sintassi Intel. Per poter visualizzare il codice disassemblato con quest'ultima bisogna lanciare il programma con l'opzione **-M intel**.

```
:~$ objdump -D -M intel file.out

 80483b5:    89 e5                    mov    ebp, esp
 80483b7:    83 ec 10                 sub    esp, 0x10
 80483ba:    c7 45 fc 0a 00 00 00     mov    DWORD PTR [ebp-0x4], 0xa
 80483c1:    b8 00 00 00 00           mov    eax, 0x0
 80483c6:    c9                       leave
 80483c7:    c3                       ret
 80483c8:    90                       nop
```

Le differenze sono visibili fin da subito. Chi è comunque interessato ad approfondire queste caratteristiche può leggere attentamente **questo** articolo della IBM, decisamente più esaustivo a riguardo.

Tuttavia quello che volevo mettere in evidenza non è la sintassi (per questo mi sono limitato a postare il link), ma bensì il codice macchina relativo a ciascuna linea di istruzioni, dato che non è nient'altro che la traduzione vera e propria di ciò che riesce ad interpretare il dispositivo. Chiariamo il tutto con un semplice e chiaro esempio. L'istruzione

```
mov    ebp, esp
```

che muove il contenuto del registro ESP al registro EBP viene interpretata dalla CPU come **0x89e5**, ovvero **10001001 11100101**.

Per rendere il tutto ancora più esplicito, ritorniamo al codice C di **Blink2** scritto per un microcontrollore **ATMEGA328P**.

```
/*
  Arduino LED blinking on board pin nº 13
  Author: Gohan
```

```
*/

/* Set XTAL frequency */
#define F_CPU 16000000UL

/* Header includes for I/O & delay functions */
#include <avr/io.h>
#include <util/delay.h>

int main (void)
{
    /* set PORTB as Output    */
    DDRB = 0xff;

    /* Infinite loop */
    while(1)
    {
        PORTB |= _BV(PB5);
        _delay_ms(1000);
        PORTB &= ~_BV(PB5);
        _delay_ms(1000);
    }

    return 1;
}
/* end of LED program */
```

e sfruttiamo le tools di **avr-binutils** per esporre il concetto. Compiliamo il programma digitando

avr-gcc -c -O -mmcu=atmega32 blink2.c -o blink2

successivamente creiamo il file .hex con

avr-objcopy -O ihex blink2 blink2.hex

di cui se ne rappresenta il contenuto.

```
:100000008FEF87BBE8E3F0E060E177E240E951E0A1
:1000100080818062808 39B01CA01019701F4215095
:10002000304001F480818F7D80839B01CA0101975C
:0A00300001F42150304001F400C03B
:00000001FF
```

Analizzando il file oggetto compilato dal sorgente C con

avr-objdump -D blink2

l'ouptut risulta il seguente

```
blink:     file format elf32-avr

Disassembly of section .text:

00000000 <main>:
   0:    8f ef           ldi    r24, 0xFF    ; 255
   2:    87 bb           out    0x17, r24    ; 23
   4:    e8 e3           ldi    r30, 0x38    ; 56
   6:    f0 e0           ldi    r31, 0x00    ; 0
   8:    60 e1           ldi    r22, 0x10    ; 16
   a:    77 e2           ldi    r23, 0x27    ; 39
   c:    40 e9           ldi    r20, 0x90    ; 144
   e:    51 e0           ldi    r21, 0x01    ; 1
  10:    80 81           ld     r24, Z
  12:    80 62           ori    r24, 0x20    ; 32
  14:    80 83           st     Z, r24
  16:    9b 01           movw   r18, r22
  18:    ca 01           movw   r24, r20
  1a:    01 97           sbiw   r24, 0x01    ; 1
  1c:    01 f4           brne   .+0          ; 0x1e <main+0x1e>
  1e:    21 50           subi   r18, 0x01    ; 1
  20:    30 40           sbci   r19, 0x00    ; 0
  22:    01 f4           brne   .+0          ; 0x24 <main+0x24>
  24:    80 81           ld     r24, Z
  26:    8f 7d           andi   r24, 0xDF    ; 223
  28:    80 83           st     Z, r24
  2a:    9b 01           movw   r18, r22
  2c:    ca 01           movw   r24, r20
  2e:    01 97           sbiw   r24, 0x01    ; 1
  30:    01 f4           brne   .+0          ; 0x32 <main+0x32>
  32:    21 50           subi   r18, 0x01    ; 1
  34:    30 40           sbci   r19, 0x00    ; 0
  36:    01 f4           brne   .+0          ; 0x38 <main+0x38>
  38:    00 c0           rjmp   .+0          ; 0x3a <__CCP__+0x6>
```

nel quale possiamo osservare le istruzioni in Assembly AVR ed il relativo codice macchina. Il file .hex non è esattamente uguale al codice macchina ottenuto in output con il disassemblatore, ma possiamo ugualmente notare come il codice relativo alle

prime otto istruzioni è tutto presente nella prima riga del file blink2.hex, esattamente dal primo 8 fino all'ultimo 0. Continuando l'analisi si trovano le istruzioni successive nel contenuto del file.

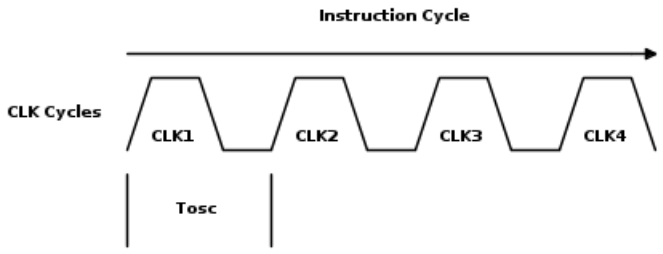
Voglio precisare che gli esempi a riga di comando riportati sono solo a scopo illustrativo, in quanto potrebbero non essere efficienti al momento di programmare il microcontrollore.

È bene quindi sottolineare che ogni istruzione mnemonica (i.e. mov, ret, int, movlw, bsf, ldi, out etc.) in funzione degli operandi e del loro contenuto assume un determinato valore. Questo è facilmente visibile sempre facendo riferimento all' output del codice x86 disassemblato: possiamo benissimo notare che l'istruzione **leave** ha un valore di riconoscimento **0xc9**, la **ret** equivale a **0xc3** mentre la **nop** a **0x90**. Wikipedia è altrettanto abbastanza chiara con un esempio analogo nella sezione dell'articolo a **questo** link.

Quello che cambia da una architettura all'altra sono appunto le istruzioni (anche se in molti casi alcune sono uguali) ed i registri (gli operandi) quindi la decodificazione del programma secondo l'hardware. Questi sono i dettagli che rendono il linguaggio Assembly dipendente dall'architettura, ed è da quì che viene appunto il termine "basso livello". Ovviamente chi prestabilisce tutte le codificazioni del linguaggio secondo le caratteristiche hardware, è il fabbricante.

Tempi di esecuzione di una istruzione

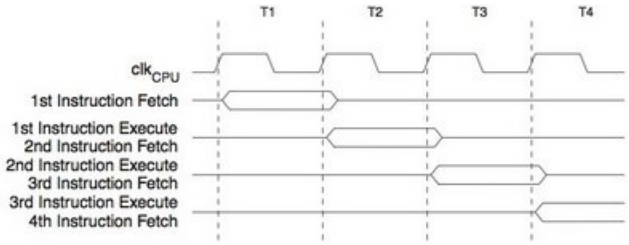
Ogni istruzione, a seconda dell'architettura, necessita di un tempo minimo per essere eseguita, ed un altro dettaglio a cui è bene tener conto è il tempo che il dispositivo impiega ad eseguirla. Nei PIC per esempio, una istruzione può impiegare da 4 a 8 cicli di clock per essere eseguita, ovvero 1 o 2 cicli di istruzione. Un diagramma temporale d'esempio per un ciclo di istruzione è il seguente



Normalmente sono le istruzioni di salto che impiegano più tempo nell'essere eseguite, come ad esempio la **call**, la **goto**, **return** etc. Per informazioni più dettagliate, è bene consultare il datasheet del dispositivo a programmare. Possiamo quindi dire che, se il nostro PIC sta lavorando ad una frequenza di 4MHz, una

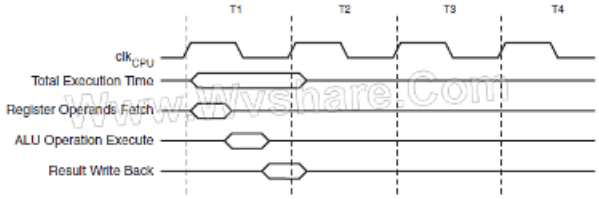
istruzione impiegarà quindi $IExTime = \frac{N \cdot Tosc}{Fosc} = \frac{4}{4MHz} = 1\mu s$ che equivale ad un ciclo di istruzione, ovvero il tempo necessario per eseguire una istruzione da 1 ciclo.

Nei microcontrollori AVR però sembra che qualcosa cambia, e di molto. Se per esempio andiamo a pagina 14 del datasheet del **ATMEGA328p** nella sezione **Instruction Execution Timing** osserviamo il seguente schema.



Atmega 16.pdf (page 13 of 358).jpg

Dal digramma temporale vediamo che ogni singola istruzione viene eseguita in un tempo equivalente ad 1 ciclo di clock. Più in dettaglio



M128A-P15-2.png

Questo vuol dire che un AVR è 4 volte più veloce di un PIC, dettaglio molto importante visto che la frequenza di oscillazione è un valore determinante del consumo di potenza del dispositivo.

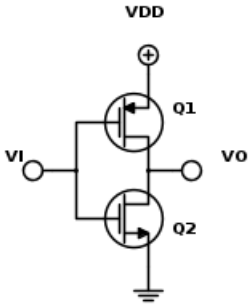
Nel software, i tempi di esecuzione sono altrettanto molto importanti quando abbiamo a che fare con le temporizzazioni. Se per esempio vogliamo ottenere una temporizzazione di 20us con un PIC, potremmo anche mettere 20 nop una dopo l'altra, per fare un banale esempio. Generalmente, è consigliabile usare i timer interni del dispositivo in quanto offrono una precisione molto più accurata ed una implementazione molto più versatile, basta pensare ad una temporizzazione di antirimbalzo (20ms o poco più sono sufficienti) per l'uso di interruttori o di tempi più lungi come ad esempio 1s.

Facendo riferimento ai datasheet riguardo ai tempi di esecuzione, possiamo farci una idea di come calcolare il tempo di istruzione di un programma in base al codice scritto.

Ancora più in basso

Durante la mia esperienza con la programmazione, mi sono sempre chiesto come venissero implementati registri e memorie, così visto che siamo su un portale di elettricità ed elettronica, mi è sembrata una buona idea scrivere un piccolo cenno a riguardo, sperando che il tutto risulti interessante e stimolante per quelle persone che si affacciano al mondo dell'elettronica e non.

Al giorno d'oggi la grande maggioranza (se non tutti) di circuiti integrati viene fabbricata in tecnologia **CMOS**. Le sue qualità (bassa dissipazione di potenza, alta impedenza di ingresso, alta immunità al rumore, dimensioni ridotte etc.) fanno di questa una tecnologia ideale per la fabbricazione di circuiti digitali. Ricordo che il primo giorno a lezione di elettronica digitale una delle prime domande che feci al professore era come venissero costruite le porte logiche. Il solo semplice funzionamento di una NOT o di una NAND era già una curiosità molto grande. Fu così che qualche mese dopo conobbi lo schema di una NOT CMOS.



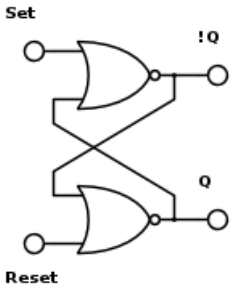
Tuttavia una analisi basilare sul funzionamento del circuito è qualcosa di più complesso che una semplice funzione logica come $Y = \overline{A}$. Conoscendo le regole di base per la polarizzazione dei MOSFET, e considerando che nel circuito $V_{Thn} = V_{Thp} = |V_{Th}|$, nella situazione in cui $V_I \approx V_{DD}$, sul PMOS (Q1) avremo una $V_{GSp} \approx 0V$, non sufficienti ad accendere il transistore essendo che $V_{GSp} > V_{Th}$, il quale sarà interdetto, visto che non si compie la condizione per accenderlo, ossia $V_{GS} \leq V_{Th}$. Sarà invece l'NMOS (Q2) ad essere acceso con una $V_{GSn} > V_{Th}$, ma con una impedenza molto elevata sul Drain imposta da Q1 (r_{DSp}) che si comporta come un circuito aperto. Quindi, in questa situazione, il circuito è analogo ad uno switch aperto (Q1) tra V_{DD} ed una resistenza di pull-down (r_{DSn}), con il terminale di uscita collegato al morsetto superiore di questa, con $V_O \approx 0V$ quindi stato logico **0**.

Diversamente, quando $V_I \approx 0V$, Q1 avrà una $V_{GSp} < V_{Th}$, quindi sarà finalmente acceso. Questa volta però è Q2 a trovarsi nella zona di interdizione, avendo altrettanto una $V_{GSn} < V_{Th}$. Non si compie quindi la condizione necessaria per accenderlo, ossia $V_{GSn} \geq V_{Thn}$, attuando così come un circuito aperto ed imponendo

quindi una impedenza molto elevata (r_{DSn}) sul Drain di Q1. In questa situazione il circuito è equivalente ad uno switch aperto (Q2) collegato fra il potenziale di riferimento ed una resistenza di pull-up (r_{DSp}) con il terminale d'uscita collegato al morsetto inferiore di essa. È facile quindi immaginare che in questo caso $V_O \approx V_{DD}$ quindi stato logico **1**.

L'analisi precedente è decisamente analogica, visto che i segnali da analizzare non sono cifre binarie a parte la rappresentazione digitale delle tensioni di entrata e di uscita. È però in circuiti come questi che l'elettronica analogica da una stretta di mano all'elettronica digitale, permettendoci con quest'ultima di progettare circuiti che senza rendercene conto usiamo di giorno in giorno. Per esempio un circuito aritmetico non è formato nient'altro che da porte logiche, come per esempio il **Full Adder**.

Registri e memorie a loro volta, sono circuiti formati da Flip Flop, elemento fondamentale della logica sequenziale (quindi di rilevante importanza) composto da due o più porte logice, che permette di memorizzare temporaneamente lo stato logico grazie ad una rete di retroazione. Prendiamo come semplice esempio un Flip Flop SR con porte logiche NOR

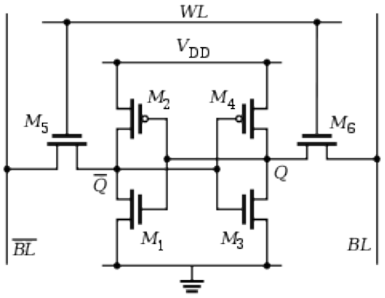


Se analizziamo un momento con attenzione la tabella della verità

S	R	Output
0	0	Q (No change)
1	0	S
0	1	R
1	1	Undefinied

possiamo notare che, ipotizzando uno stato logico iniziale di Q a 0 dovuto ad un precedente Reset, se successivamente viene dado un semplice impulso positivo all'entrata di Set, Q rimarrà a stato logico 1 fino alla prossima condizione di Reset, che a priori non si sà quando può avvenire. Questo è quindi un semplice ed esplicito esempio di memorizzazione.

A seguire si mostra una illustrazione di una cella di memoria SRAM a 6 transistori in tecnologia CMOS.



SRAM_Cell_(6_Transistors).svg.png

Tuttavia questa sezione voleva solo illustrare una piccolissima parte di quei circuiti che, nel complesso, formano microcontrollori, microprocessori ed altri integrati, e visto che il mio percorso di studi (autodidatta e non) mi ha portato fino a quì, ripeto, mi è sembrata una buona idea dedicare qualche riga a riguardo sperando che sia risultato interessante. Alla fine dell'articolo ho provveduto a riportare qualche link utile.

Esempio con PIC & AVR

Tornando all'Assembly, voglio quindi riportarne due esempi con due programmi rispettivamente scritti per i microcontrollori PIC16F877A e un ATMEGA328P:

Il primo programma è scritto per il PIC appena menzionato, con il quale ho avuto modo di praticare durante gli studi. Ha qualche mese ed era un esercizio che richiedeva di visualizzare su un display a 7 segmenti a catodo comune il numero di bit del PORTC a stato logico alto, a sua volta collegati ad uno switch ed una resistenza di pull-down per ciascun pin.

```
;
; This program displays on a 7 segment common cathode
; display connected to PORTD, the number of bits
; of the PORTC set to logic state 1. Oscillator frequency: 4MHz
;
; Author: Gohan
;

CBLOCK   0x20
   I
ENDC
```

```
LIST p=16f877a
INCLUDE p16F877A.INC

__CONFIG _CP_OFF&_DEBUG_OFF&_WRT_OFF&_CPD_OFF&_LVP_OFF&_BODEN_ON&_PWRTE_ON
&_WDT_OFF&_HS_OSC

   ORG 0x00             ;
                        ; step to bank 1
   bsf   STATUS, RP0    ; set RP0
   bcf   STATUS, RP1    ; clear RP1

   movlw 0xff           ; set PORTC as input
   movwf TRISC

   movlw 0x00           ; set PORTD as output
   movwf TRISD
                        ; step to bank 0
   bcf   STATUS, RP0    ; clear RP0
   bcf   STATUS, RP1    ; clear RP1


main:
   clrf  I              ; clear I to make sure it starts from 0

   btfsc PORTC, 0       ; if bit0 is not set, check bit1
   incf  I, 1           ; if it's set, increse I by one

   btfsc PORTC, 1       ; if bit1 is not set, check bit2
   incf  I, 1           ; if it's set, increse I by one

   btfsc PORTC, 2       ; if bit2 is not set, check bit3
   incf  I, 1           ; if it's set, increse I by one

   btfsc PORTC, 3       ; if bit3 is not set, check bit4
   incf  I, 1           ; if it's set, increse I by one

   btfsc PORTC, 4       ; if bit4 is not set, check bit5
   incf  I, 1           ; if it's set, increse I by one

   btfsc PORTC, 5       ; if bit5 is not set, check bit6
   incf  I, 1           ; if it's set, increse I by one

   btfsc PORTC, 6       ; if bit6 is not set, check bit7
```

```
   incf  I, 1           ; if it's set, increse I by one

   btfsc PORTC, 7       ; if bit7 is not set leave function
   incf  I, 1           ; if it's set, increse I by one

   movf  I, W           ; store the value of I in W
   call  display        ; decode W to 7 segment binary equal
   movwf PORTD          ; output it on PORTD

   goto  main           ; return to main
                        ;

display
   addwf PCL, F         ;
   retlw 0x3f           ; display 0
   retlw 0x06           ; display 1
   retlw 0x5b           ; display 2
   retlw 0x4f           ; display 3
   retlw 0x66           ; display 4
   retlw 0x6d           ; display 5
   retlw 0x7d           ; display 6
   retlw 0x07           ; display 7
   retlw 0x7f           ; display 8

   end                  ; end of program
```

Con MPLAB è altrettanto possibile osservare il codice macchina relativo ad ogni istruzione accedendo al menu **View >> Disassembly Listing**.

Dopo le richieste di codice Assembly per Blink2, ho così provveduto a procurarmi il codice equivalente. Non avendo ancora assimilato l'esperienza necessaria per scrivere tale programma, mi sono servito ancora una volta di avr-gcc, in questo caso per generare il sorgente .asm partendo da quello C, con il comando

avr-gcc -O -mmcu=atmega32 --save-temps -S blink2.c

Tuttavia, prima di effettuare il tutto, è stata eseguita una modifica di ottimizzazione a Blink2.

```
#define F_CPU 16000000UL
#include <avr/io.h>
#include <util/delay.h>

__attribute__((noinline)) void delay_one_second(void) {
```

```
  _delay_ms(1000);
}

int main (void)
{
   DDRB = 0xff;

   while(1)
   {
      PORTB |= _BV(PB5);
      delay_one_second();
      PORTB &= ~_BV(PB5);
      delay_one_second();
   }

   return 1;
}
```

ed il risultante codice Assembly generato dal compilatore è il seguente:

```
__SREG__ = 0x3f
__SP_H__ = 0x3e
__SP_L__ = 0x3d
__CCP__  = 0x34
__tmp_reg__ = 0
__zero_reg__ = 1
    .global __do_copy_data
    .global __do_clear_bss
    .text
.global    delay_one_second
    .type    delay_one_second, @function
delay_one_second:
    ldi r18,lo8(10000)
    ldi r19,hi8(10000)
    ldi r20,lo8(400)
    ldi r21,hi8(400)
.L2:
    movw r24,r20
 sbiw r24,1
    brne 1b
    subi r18,lo8(-(-1))
    sbci r19,hi8(-(-1))
    brne .L2
```

```
    ret
    .size    delay_one_second, .-delay_one_second
.global    main
    .type    main, @function
main:
    push r28
    push r29
    ldi r24,lo8(-1)
    out 55-32,r24
    ldi r28,lo8(56)
    ldi r29,hi8(56)
.L6:
    ld r24,Y
    ori r24,lo8(32)
    st Y,r24
    call delay_one_second
    ld r24,Y
    andi r24,lo8(-33)
    st Y,r24
    call delay_one_second
    rjmp .L6
    .size    main, .-main
```

Conclusioni e ringraziamenti

Con quest'articolo volevo semplicemente esporre quei dettagli che rendono l'Assembly dipendente dall'architettura hardware. Spero quindi di essere riuscito nell'intento e che il tutto possa essere di altrettanto interesse per i lettori e partecipanti del forum, nuovi e non.

Ringrazio **IsidoroKZ** per avermi corretto riguardo ad un dettaglio sull'analisi del funzionamento della porta logica CMOS NOT, mentre per l'ottimizzazione del codice di Blink2.c, si ringrazia gli utenti di **AVRFreaks**. Nel caso vengano riscontrati errori, vi prego di comunicarmelo e provvederò alle dovute modifiche. Grazie mille.

Links Utili

- **The Art of Assembly Language**
- **PIC Assembly Tutorial**
- **AVRBeginners**
- **AVRBeginners/new**
- **AVR Assembly Tutorial**

- **AVR Instruction Set Manual**
- **Fundamentals of Digital Electronics**
- **Digital Electronics Openbook**
- **Circuits, Devices & Systems**
- **CMOS Logic Gates Review**
- **Inside CMOS Logic Gates**
- **CMOS Memories**
- **Flip Flop Applications**
- **Flash Memory Technology**