# PROGRAMMAZIONE ASSEMBLER

Esempio di semplice lampeggiatore a LED

072805 – Sistemi Elettronici
Dicembre 2006

Ultimo aggiornamento: 11 dicembre 2006

# OBBIETTIVI

1 – Discutere delle metodologie di progetto e dei sistemi di sviluppo hw/sw di applicazioni basate sui microcontrollori

2 – Realizzare un semplice circuito che accende e spegne un diodo LED a intervalli regolari di tempo basato su microcontrollori Microchip PIC

3 – Acquisire una maggiore familiarità con il linguaggio assembler per microcontrollori

4 – Approfondire la conoscenza dell'architettura interna del micro necessaria per comprendere la struttura del firmware assembler

5 – Mettere in evidenza i punti di forza e gli svantaggi della programmazione tramite linguaggio macchina.

# IL MICROCONTROLLORE PIC16F84A

## HIGH PERFORMANCE RISC CPU FEATURES

- Only 35 single word instructions to learn
- Operating speed: DC - 20 MHz clock input
- 1024 words of program memory
- 68 bytes of Data RAM
- 64 bytes of Data EEPROM
- 14-bit wide instruction words
- 8-bit wide data bytes
- 15 Special Function Hardware registers
- Eight-level deep hardware stack
- Direct, indirect and relative addressing modes
- Four interrupt sources:
  - External RB0/INT pin
  - TMR0 timer overflow
  - PORTB<7:4> interrupt-on-change

## PERIPHERAL FEATURES

- 13 I/O pins with individual direction control
- High current sink/source for direct LED drive
  - 25 mA sink max. per pin
  - 25 mA source max. per pin
- TMR0: 8-bit timer/counter with 8-bit programmable prescaler

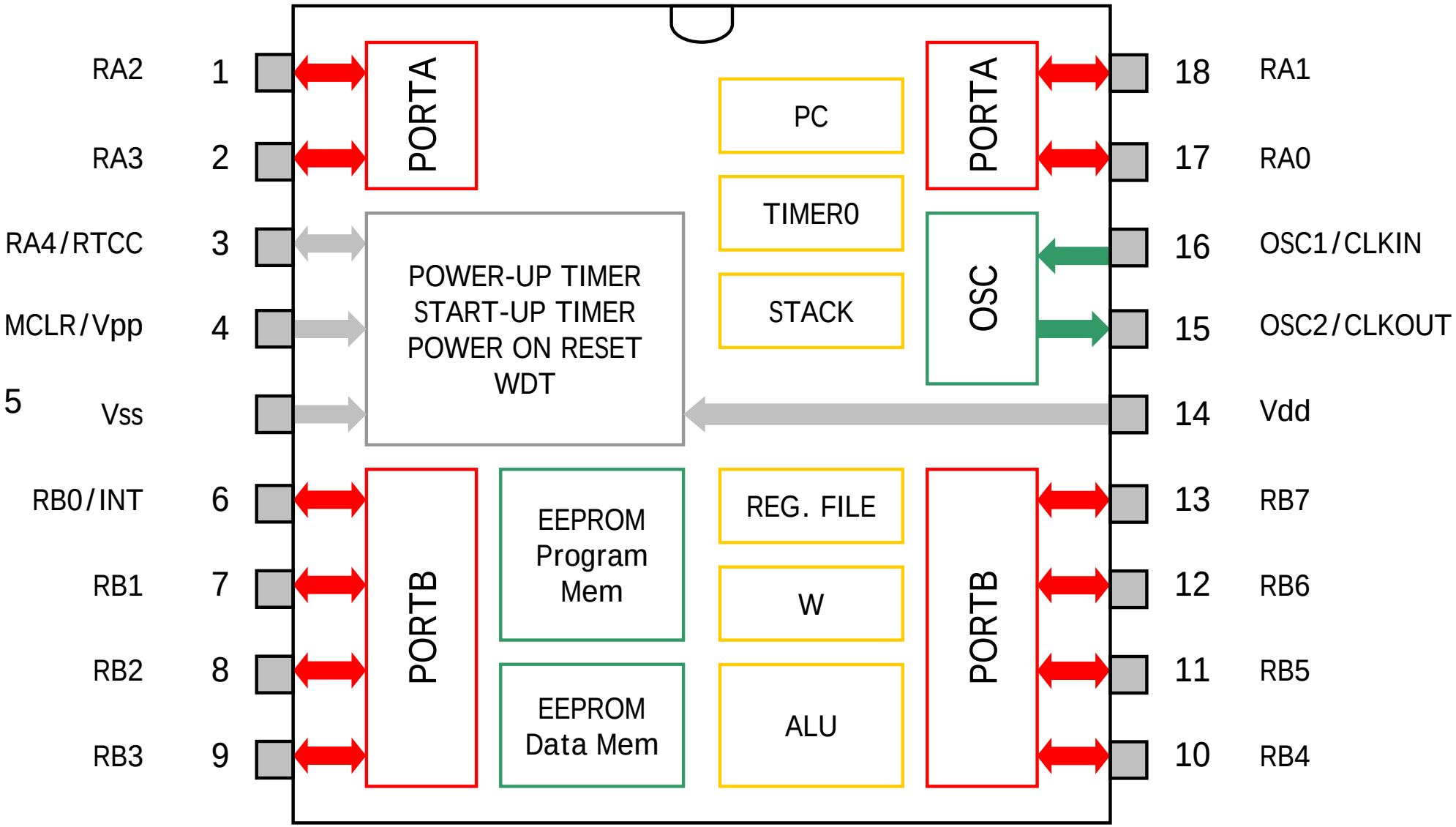
## CMOS ENHANCED FLASH/EEPROM TECHNOLOGY

- Low power, high speed technology
- Fully static design
- Wide operating voltage range:
  - Commercial: 2.0V to 5.5V
  - Industrial: 2.0V to 5.5V
- Low power consumption:
  - < 2 mA typical @ 5V, 4 MHz
  - 15 mA typical @ 2V, 32 kHz
  - < 0.5 mA typical standby current @ 2V

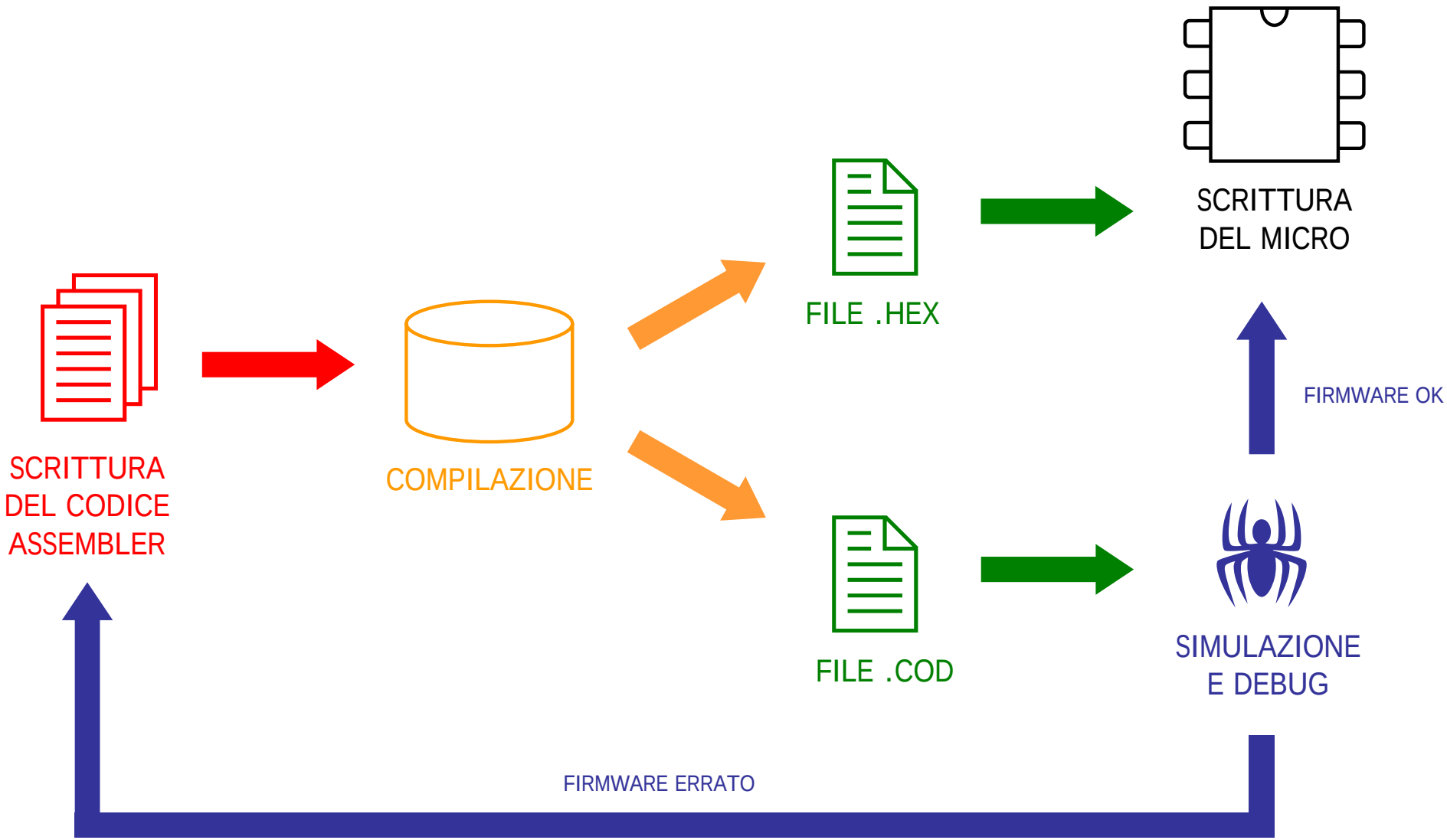
## SPECIAL MCU FEATURES

- In-Circuit Serial Programming™ (ICSP™)
- Power-on Reset, Power-up Timer, Oscillator Start-up Timer
- Watchdog Timer with On-Chip RC oscillator
- Code protection
- Power saving SLEEP mode
- Selectable oscillator options

# PIN E PERIFERICHE INTERNE



# DESIGN FLOW


SCRITTURA DEL CODICE ASSEMBLER
COMPILAZIONE
FILE .HEX
FILE .COD
SCRITTURA DEL MICRO
FIRMWARE OK
SIMULAZIONE E DEBUG
FIRMWARE ERRATO

Blinking Led - MPLAB IDE v7.42

File Edit View Project Debugger Programmer Tools Configure Window Help

Checksum 0xd9dc

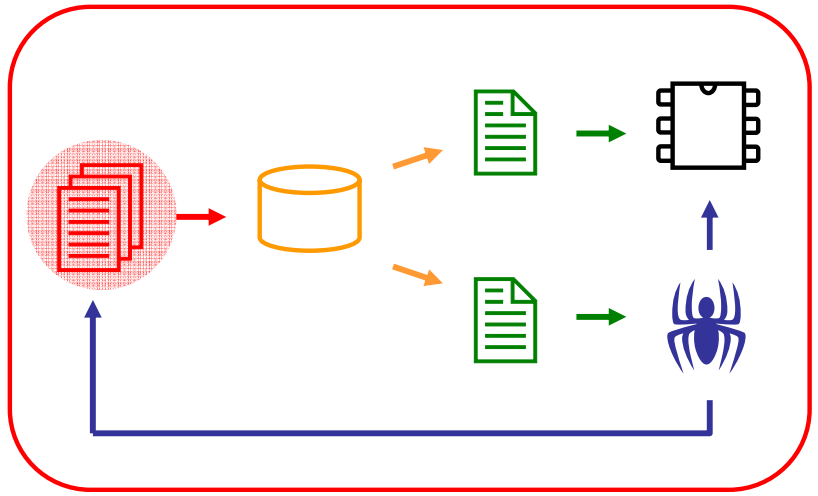
Blinking L...

- Blinking Led.mcp
  - Source Files
    - blinking_led1.as
  - Header Files
    - P16F84.INC
  - Object Files
  - Library Files
  - Linker Scripts
  - Other Files

Files | Symbols

C:\Sorgenti_ASM\blinking_led1\blinking_led1.asm

```
    PROCESSOR   16F84A
    RADIX       DEC

    INCLUDE     "P16F84A.INC"

LED     EQU 0
;SWITCH EQU 1

    ORG     0CH
Count   RES 2

    ORG     00H
    bsf     STATUS,RP0

    movlw   00011111B
    movwf   TRISA

    movlw   11111110B
    movwf   TRISB

    bcf     STATUS,RP0

    bsf     PORTB,LED

;----------------- MAIN --------------

;   btfsc   PORTB,SWITCH
;   goto    $-1
```



PIC16F84A | W:0 | z dc c | bank 0 | Ln 1, Col 1 | INS | WR

Blinking Led - MPLAB IDE v7.42

File Edit View Project Debugger Programmer Tools Configure Window Help

Checksum 0xd9dc

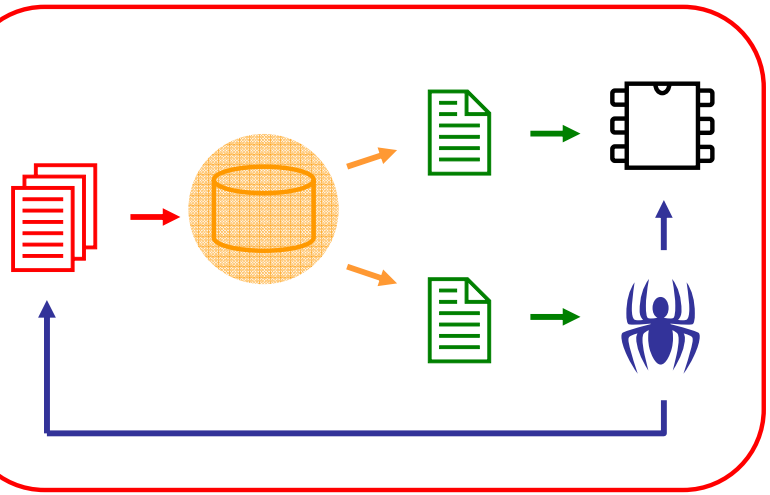
Blinking L...

- Blinking Led.mcp
  - Source Files
    - blinking_led1.as
  - Header Files
    - P16F84.INC
  - Object Files
  - Library Files
  - Linker Scripts
  - Other Files

Files | Symbols

C:\Sorgenti_ASM\blinking_led1\blinking_led1.asm

```
        PROCESSOR   16F84A
        RADIX       DEC

        INCLUDE     "P16F84A.INC"

LED     EQU 0
;SWITCH EQU 1

        ORG     0CH
Count   RES 2

        ORG     00H
        bsf     STATUS,RP0
```



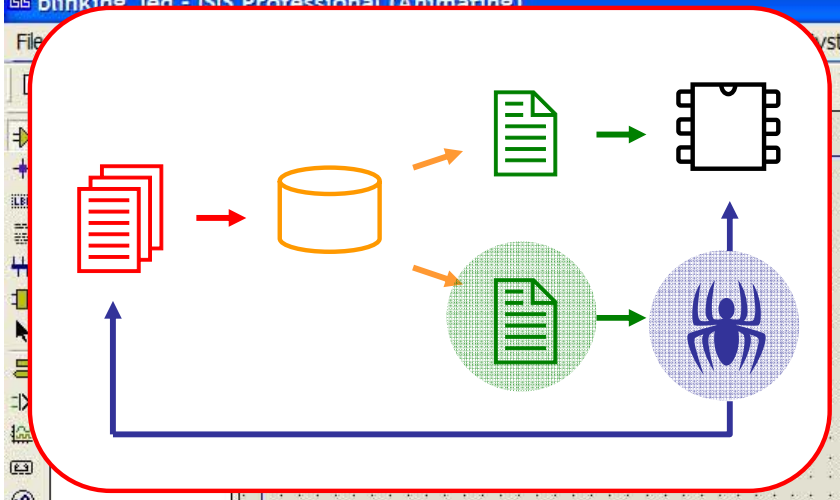
Output

Build | Version Control | Find in Files

```
Clean: Deleted file "C:\Sorgenti_ASM\blinking_led1\Blinking Led.mcs".
Clean: Done.
Executing: "C:\Programmi\Microchip\MPASM Suite\MPASMWIN.exe" /q /p16
Message[302] C:\SORGENTI_ASM\BLINKING_LED1\BLINKING_LED1.AS
Message[302] C:\SORGENTI_ASM\BLINKING_LED1\BLINKING_LED1.AS
Loaded C:\Sorgenti_ASM\blinking_led1\blinking_led1.COD.
BUILD SUCCEEDED: Tue Dec 05 18:17:06 2006
```

```
;       goto    $-1
```

PIC16F84A | W:0 | z dc c | bank 0 | WR



blinking_led - ISIS Professional (Animating)

File ... ystem Help

PIC CPU Stac...

```
1: 0x0008
2: 0x2781 <--
3: 0x046B
4: 0x394B
5: 0x15FB
6: 0x19E2
7: 0x1CFB
8: 0x3F54
```

PIC CPU Registers - U1

```
PC: $0010                W.MSGS: 0
INSTR.: Flush (NOP)
W: 254 $FE %11111110      SP: 1
STATUS: 28 $1C %00011100  RPX: 0  Z:1 DC:0 C:0
FSR: 28 $1C %00011100     PCLATH: 0  $00 %00000000
OPTION: 255 $FF %11111111 INTCON: 0  $00 %00000000
PORT A: 1  $01 %00000001  TRIS A: 31 $1F %00011111
PORT B: 0  $00 %00000000  TRIS B: 254 $FE %11111110
```

PIC CPU Program Memory - U1

```
0000 | 83 16 1F 30 | 85 00 FE 30 | 86 00 83 12 | ...0...0....
000C | 06 14 0E 20 | 06 18 0C 28 | 06 14 07 28 | ... ...(...(
0018 | 06 10 07 28 | 8C 01 8D 01 | 8C 0B 10 28 | ...(.......(
0024 | 8D 0B 10 28 | 08 00 00 00 | 00 00 00 00 | ...(........
0030 | 00 00 00 00 | 00 00 00 00 | 00 00 00 00 | ............
003C | 00 00 00 00 | 00 00 00 00 | 00 00 00 00 | ............
0048 | 00 00 00 00 | 00 00 00 00 | 00 00 00 00 | ............
0054 | 00 00 00 00 | 00 00 00 00 | 00 00 00 00 | ............
0060 | 00 00 00 00 | 00 00 00 00 | 00 00 00 00 | ............
006C | 00 00 00 00 | 00 00 00 00 | 00 00 00 00 | ............
0078 | 00 00 00 00 | 00 00 00 00 | 00 00 00 00 | ............
0084 | 00 00 00 00 | 00 00 00 00 | 00 00 00 00 | ............
0090 | 00 00 00 00 | 00 00 00 00 | 00 00 00 00 | ............
009C | 00 00 00 00 | 00 00 00 00 | 00 00 00 00 | ............
```

C1 22pF C2 22pF

U1

OSC1/CLKIN, OSC2/CLKOUT, MCLR, RA0, RA1, RA2, RA3, RA4/T0CKI, RB0/INT, RB1, RB2, RB3, RB4, RB5, RB6, RB7

PIC16F84A

R1(1) 500

D1 LED-RED <TEXT>

PIC CPU Source Code - U1

C:\SORGENTI_ASM\BLINKING_LED1\BLINKING_LED1.ASM

```
---- ;---------------- DELAY --------------
----
---- Delay
000E          clrf    Count
000E CLRF 0x000C
000F          clrf    Count+1
000F CLRF 0x000D
---- DelayLoop
0010          decfsz  Count,1
0010 DECFSZ 0x000C,F
0011          goto    DelayLoop
0011 GOTO PCLATH+0x0010
0012          decfsz  Count+1,1
0012 DECFSZ 0x000D,F
0013          goto    DelayLoop
0013 GOTO PCLATH+0x0010
0014          return
0014 RETURN
----
----          END
```

PIC CPU Variables - U1

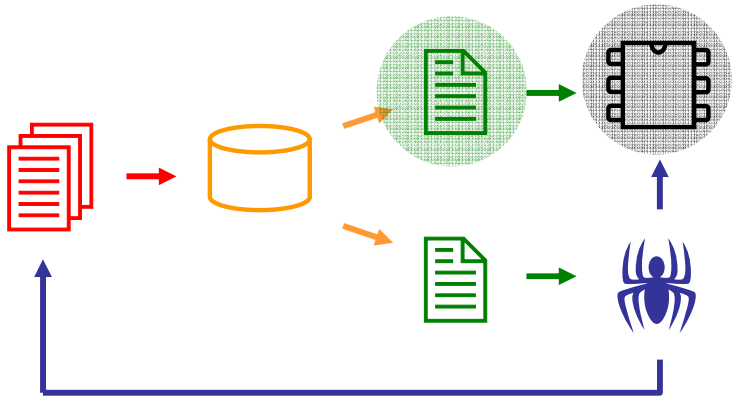
| Name | Address |
|---|---|
| C | 0x0000 |
| DC | 0x0001 |
| EEADR | 0x0009 |
| EEDATA | 0x0008 |
| EEIE | 0x0006 |
| EEIF | 0x0004 |
| F | 0x0001 |
| FSR | 0x0004 |
| GIE | 0x0007 |
| INDF | 0x0000 |
| INTCON | 0x000B |
| INTE | 0x0004 |
| INTEDG | 0x0006 |
| INTF | 0x0001 |

Watch Window

| Name | Address | Value |
|---|---|---|
| PORTB | 0x0006 | 0b00000000 |

PAUSED: 00:00:05.00

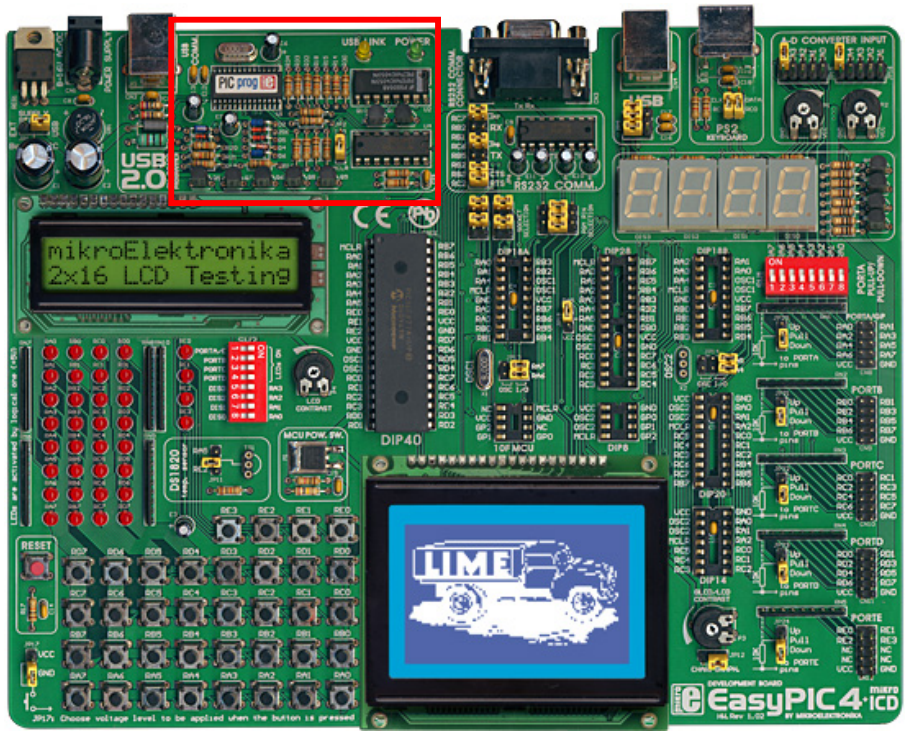
-300 +100



Ogni microcontrollore dispone di specifici pin per la programmazione, in genere è possibile programmare il micro anche quando già montato sul circuito finale

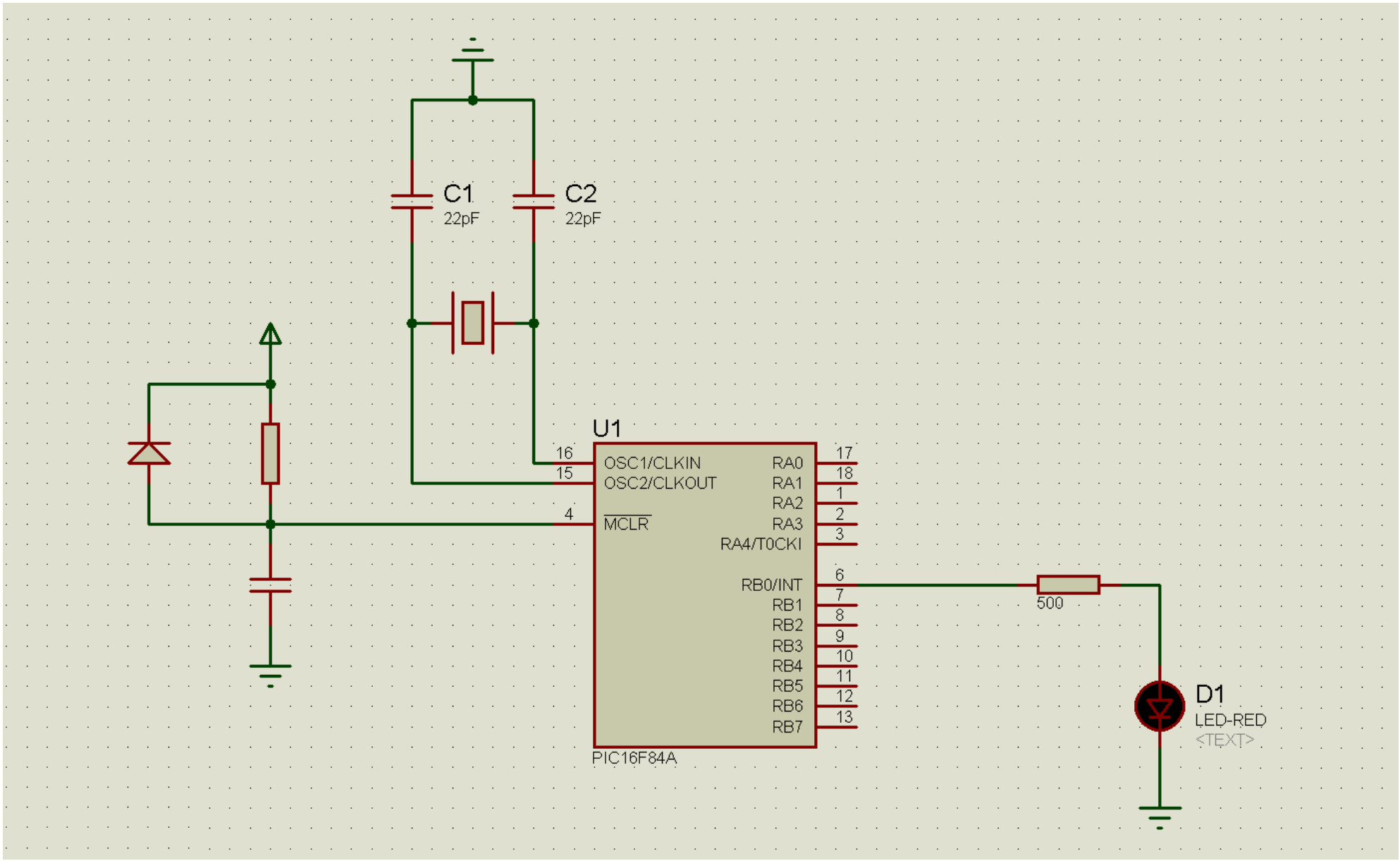
(In-Circuit-Programming)

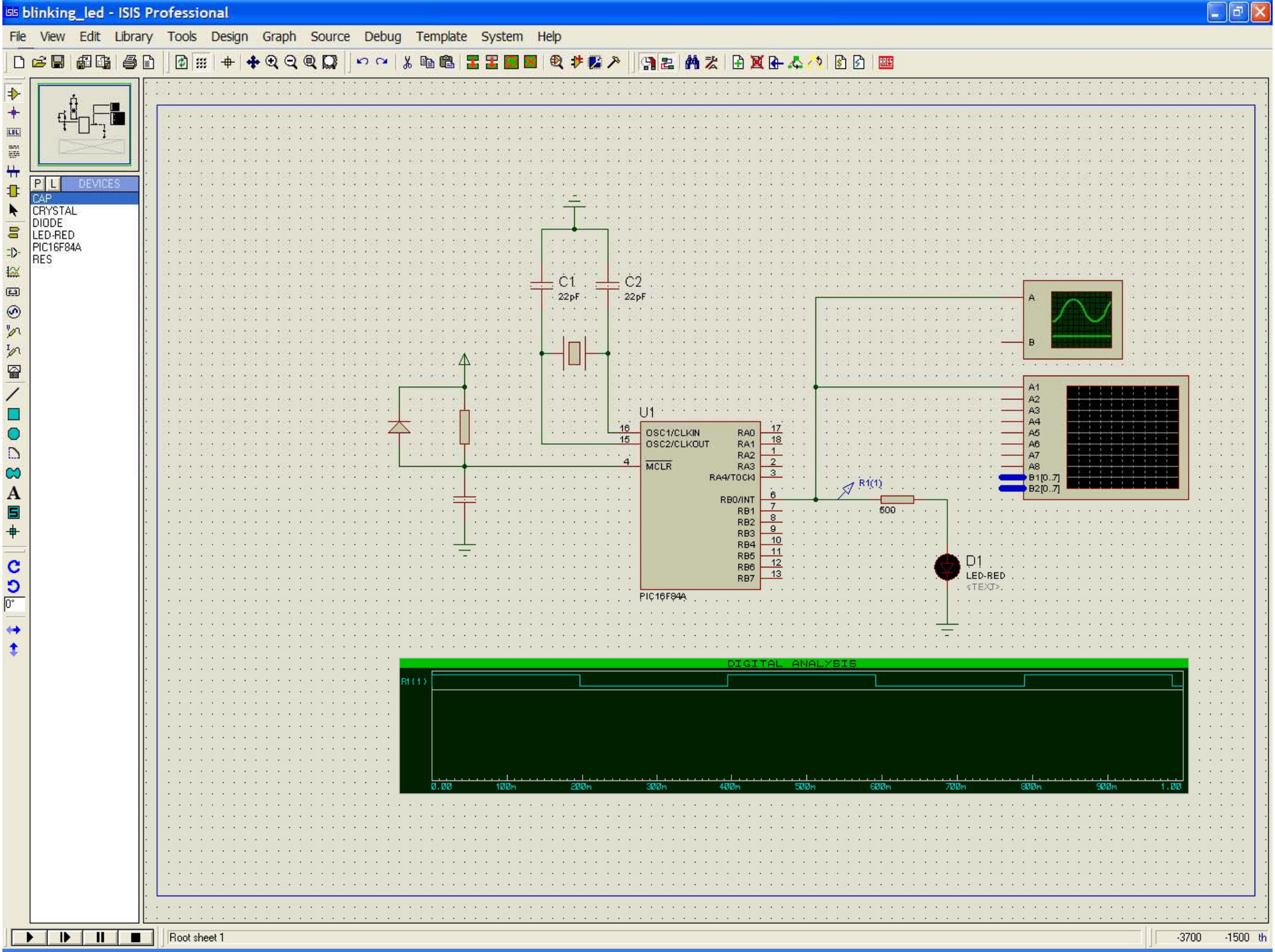

PROGRAMMATORI STAND-ALONE

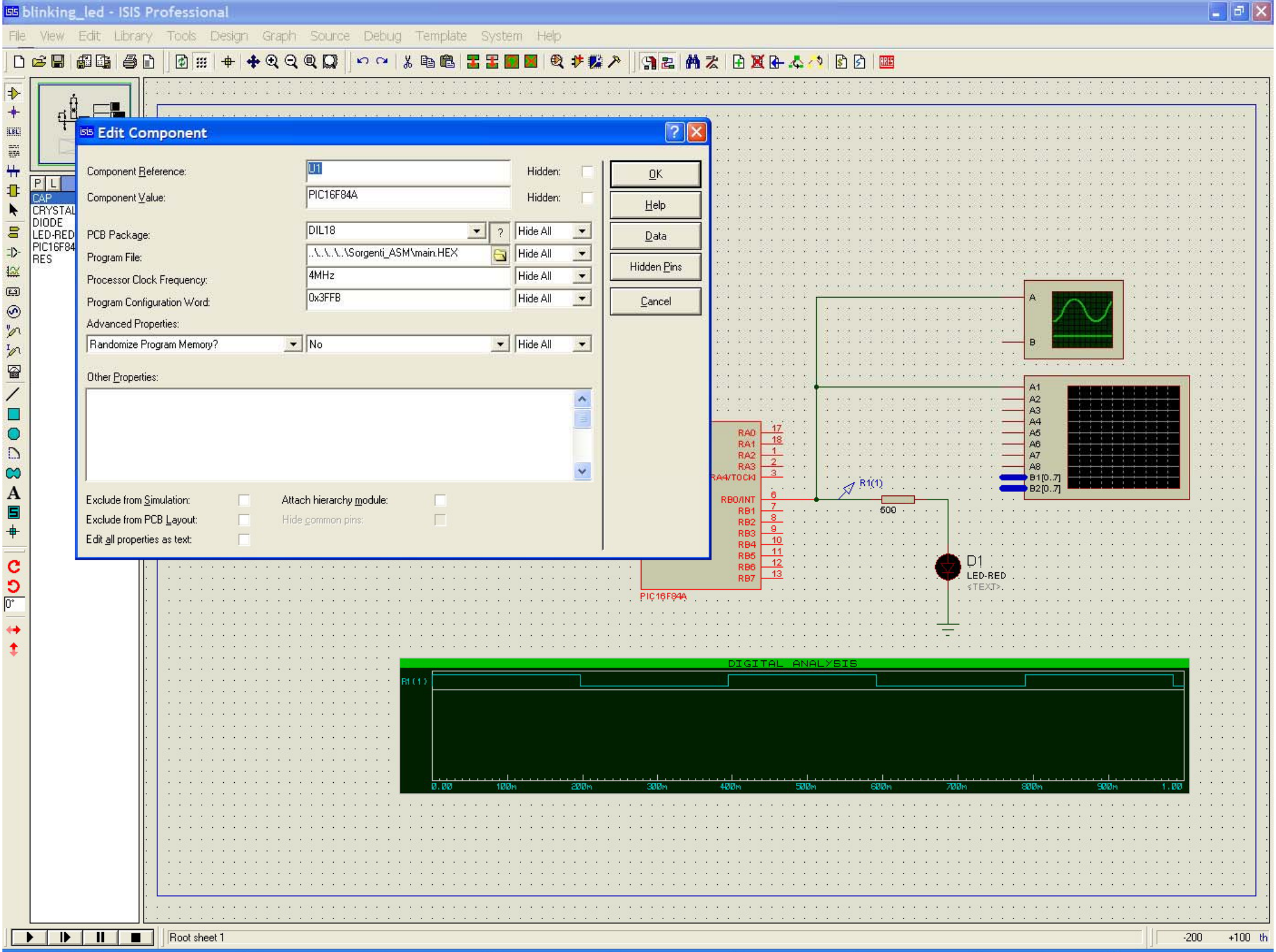



DEVELOPMENT-BOARD

# LO SCHEMA ELETTRICO




blinking_led - ISIS Professional
File View Edit Library Tools Design Graph Source Debug Template System Help
DEVICES
CAP
CRYSTAL
DIODE
LED-RED
PIC16F84A
RES
C1
22pF
C2
22pF
U1
OSC1/CLKIN
OSC2/CLKOUT
MCLR
RA0
RA1
RA2
RA3
RA4/T0CKI
RB0/INT
RB1
RB2
RB3
RB4
RB5
RB6
RB7
PIC16F84A
R1(1)
500
D1
LED-RED
A
B
A1
A2
A3
A4
A5
A6
A7
A8
B1[0..7]
B2[0..7]
DIGITAL ANALYSIS
Root sheet 1


blinking_led - ISIS Professional
File View Edit Library Tools Design Graph Source Debug Template System Help
Edit Component
Component Reference:
U1
Hidden:
Component Value:
PIC16F84A
Hidden:
PCB Package:
DIL18
Hide All
Program File:
..\..\..\..\Sorgenti_ASM\main.HEX
Hide All
Processor Clock Frequency:
4MHz
Hide All
Program Configuration Word:
0x3FFB
Hide All
Advanced Properties:
Randomize Program Memory?
No
Hide All
Other Properties:
Exclude from Simulation:
Attach hierarchy module:
Exclude from PCB Layout:
Hide common pins:
Edit all properties as text:
OK
Help
Data
Hidden Pins
Cancel
CAP
CRYSTAL
DIODE
LED-RED
PIC16F84
RES
RA0
RA1
RA2
RA3
RA4/T0CKI
RB0/INT
RB1
RB2
RB3
RB4
RB5
RB6
RB7
PIC16F84A
R1(1)
500
D1
LED-RED
<TEXT>
A
B
A1
A2
A3
A4
A5
A6
A7
A8
B1[0..7]
B2[0..7]
DIGITAL ANALYSIS
R1(1)
0.00
100m
200m
300m
400m
500m
600m
700m
800m
900m
1.00
Root sheet 1
-200
+100

# DIRETTIVE INIZIALI

```
        PROCESSOR       16F84A
        RADIX           DEC

        INCLUDE         "P16F84A.INC"

;---------- COSTANTI SIMBOLICHE -------------

LED     EQU             0

;---------- DEFINIZIONE VARIABILI -------------

        ORG             0CH
Count           RES 2
```

E' una direttiva per il compilatore che definisce il microcontrollore per cui si sta scrivendo il codice

Indica al compilatore che tutti i numeri sono in notazione decimale, salvo se diversamente specificato

Include nel sorgente del programma il file P16F84A.INC che contiene tutte costanti simboliche utilizzate nella programmazione

# DIRETTIVE INIZIALI

```
        PROCESSOR       16F84A
        RADIX           DEC

        INCLUDE         "P16F84A.INC"

;---------- COSTANTI SIMBOLICHE -------------

LED     EQU             0

;---------- DEFINIZIONE VARIABILI -------------

        ORG             0CH
Count           RES 2
```

E' una direttiva per il compilatore che definisce il microcontrollore per cui si sta scrivendo il codice

Indica al compilatore che tutti i numeri sono in notazione decimale, salvo se diversamente specificato

Include nel sorgente del programma il file P16F84A.INC che contiene tutte costanti simboliche utilizzate nella programmazione

# DIRETTIVE INIZIALI

```
        PROCESSOR       16F84A
        RADIX           DEC

        INCLUDE         "P16F84A.INC"

;----------- COSTANTI SIMBOLICHE -------------

LED     EQU             0

;---------- DEFINIZIONE VARIABILI -------------

        ORG             0CH
Count           RES 2
```

E' una direttiva per il compilatore che definisce il microcontrollore per cui si sta scrivendo il codice

Indica al compilatore che tutti i numeri sono in notazione decimale, salvo se diversamente specificato

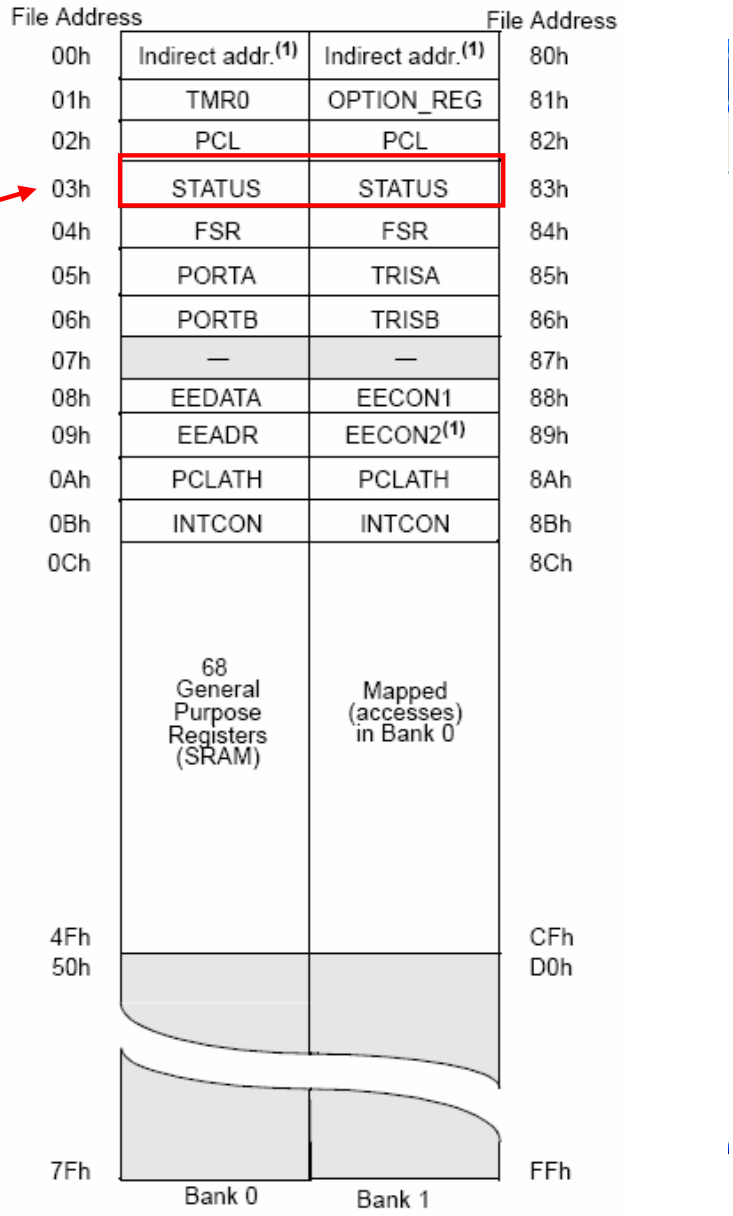
Include nel sorgente del programma il file P16F84A.INC che contiene tutte costanti simboliche utilizzate nella programmazione

# IL FILE INCLUDE “P16F84A.INC”

P16F84A.INC - Blocco note

File Modifica Formato Visualizza ?

```
TMR0                            EQU     H'0001'
PCL                             EQU     H'0002'
STATUS                          EQU     H'0003'
FSR                             EQU     H'0004'
PORTA                           EQU     H'0005'
PORTB                           EQU     H'0006'
EEDATA                          EQU     H'0008'
EEADR                           EQU     H'0009'
PCLATH                          EQU     H'000A'
INTCON                          EQU     H'000B'

OPTION_REG                      EQU     H'0081'
TRISA                           EQU     H'0085'
TRISB                           EQU     H'0086'
EECON1                          EQU     H'0088'
EECON2                          EQU     H'0089'

;----- STATUS Bits --------------------------------

IRP                             EQU     H'0007'
RP1                             EQU     H'0006'
RP0                             EQU     H'0005'
NOT_TO                          EQU     H'0004'
```



# DIRETTIVE INIZIALI

```
        PROCESSOR       16F84A
        RADIX           DEC

        INCLUDE         "P16F84A.INC"

;----------- COSTANTI SIMBOLICHE -------------

LED     EQU             0

;---------- DEFINIZIONE VARIABILI -------------

        ORG             0CH
Count           RES 2
```

Definisce una costante simbolica per una maggiore comodità di programmazione. Da questo punto in poi all'etichetta LED è associato "0"

E' una direttiva che serve per decidere in quale parte della memoria dati verranno allocate le variabili dichiarate di seguito

Riserviamo due byte di file register alla variabile Count nell'area dati

# DIRETTIVE INIZIALI

```
        PROCESSOR       16F84A
        RADIX           DEC

        INCLUDE         "P16F84A.INC"

;---------- COSTANTI SIMBOLICHE -------------

LED     EQU             0

;---------- DEFINIZIONE VARIABILI -------------

        ORG             0CH
Count           RES 2
```

Definisce una costante simbolica per una maggiore comodità di programmazione. Da questo punto in poi all'etichetta LED è associato "0"

E' una direttiva che serve per decidere in quale parte della memoria dati verranno allocate le variabili dichiarate di seguito

Riserviamo due byte di file register alla variabile Count nell'area dati

# DIRETTIVE INIZIALI

```
        PROCESSOR       16F84A
        RADIX           DEC

        INCLUDE         "P16F84A.INC"

;----------- COSTANTI SIMBOLICHE -------------

LED     EQU             0

;---------- DEFINIZIONE VARIABILI -------------

        ORG             0CH
Count           RES 2
```

Definisce una costante simbolica per una maggiore comodità di programmazione. Da questo punto in poi all'etichetta LED è associato "0"

E' una direttiva che serve per decidere in quale parte della memoria dati verranno allocate le variabili dichiarate di seguito

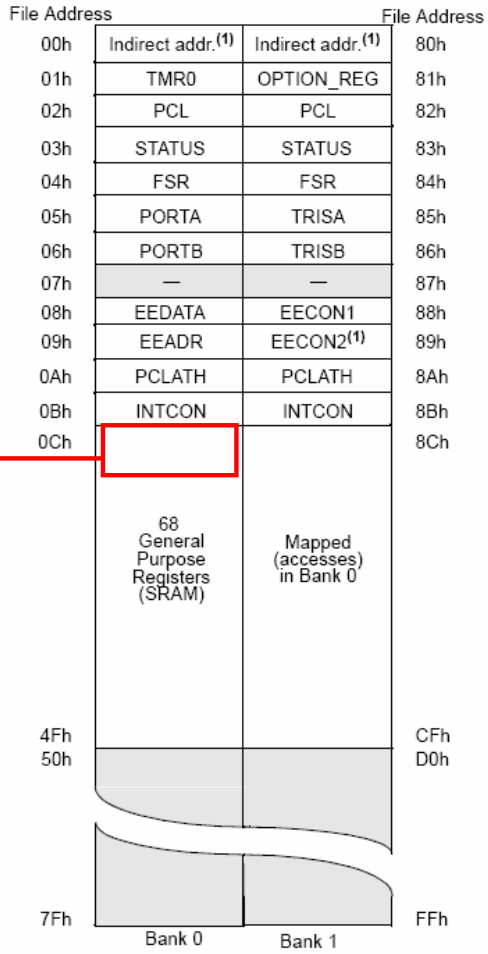
Riserviamo due byte di file register alla variabile Count nell'area dati

# COSA ABBIAMO FATTO FINO AD ORA?

Sostanzialmente quasi nulla!!! Il programma vero e proprio non è ancora cominciato, infatti non è apparsa ancora alcuna istruzione, ma abbiamo solo scritto direttive.

Le direttive sono servite a:

- Dare al compilatore alcune istruzioni per compilare il firmware
- Riservare dello spazio nella memoria per la variabile Count
- Definire una costante simbolica: LED == 0



# IMPOSTAZIONE DEI REGISTRI

```
;-------- IMPOSTAZIONE DEI REGISTRI ------

        ORG     00H
        bsf     STATUS,RP0

        movlw   00011111B
        movwf   TRISA

        movlw   11111110B
        movwf   TRISB

        bcf     STATUS,RP0

        bsf     PORTB,LED
```

Direttiva che stabilisce in quale punto della memoria programmi inserire le successive istruzioni.

Mette a "1" il bit RP0 del registro STATUS che stabilisce a quale banco del file register ci si riferisce. Stiamo infatti per agire sui registri TRISA e TRISB che stanno nel banco 1

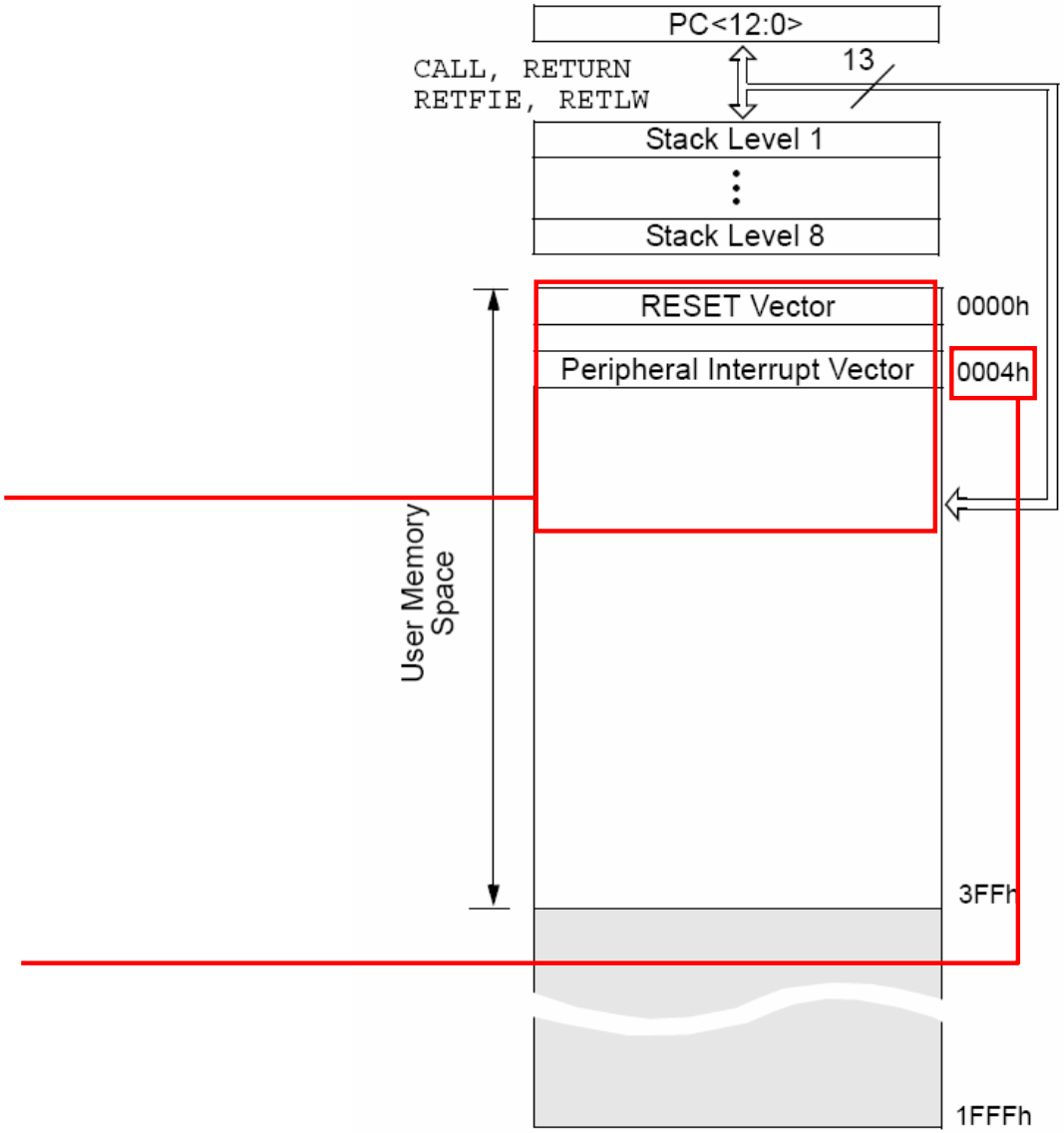
Copia nel registro W il byte 00011111 (tutti i pin della porta A sono degli input)

Copia nel registro TRISA il contenuto di W precedentemente copiato

# IMPOSTAZIONE DEI REGISTRI

Il programma che stiamo creando verrà allocato in questa porzione della memoria

Possiamo far partire il programma dalla posizione 0000h senza ulteriori accorgimenti perché in questo caso non abbiamo intenzione di utilizzare gli interrupt. L'interrupt vector si trova infatti all'indirizzo 0004h.



# IMPOSTAZIONE DEI REGISTRI

```
;-------- IMPOSTAZIONE DEI REGISTRI ------

        ORG     00H
        bsf     STATUS,RP0

        movlw   00011111B
        movwf   TRISA

        movlw   11111110B
        movwf   TRISB

        bcf     STATUS,RP0

        bsf     PORTB,LED
```

Direttiva che stabilisce in quale punto della memoria programmi inserire le successive istruzioni.

Mette a “1” il bit RP0 del registro STATUS che stabilisce a quale banco del file register ci si riferisce. Stiamo infatti per agire sui registri TRISA e TRISB che stanno nel banco 1

| | | | |
|---|---|---|---|
| 03h | STATUS | STATUS | 83h |
| 04h | FSR | FSR | 84h |
| 05h | PORTA | TRISA | 85h |
| 06h | PORTB | TRISB | 86h |
| 07h | — | — | 87h |
| 08h | EEDATA | EECON1 | 88h |
| 09h | EEADR | EECON2(1) | 89h |
| 0Ah | PCLATH | PCLATH | 8Ah |
| 0Bh | INTCON | INTCON | 8Bh |
| 0Ch | | | 8Ch |

# IMPOSTAZIONE DEI REGISTRI

```
;-------- IMPOSTAZIONE DEI REGISTRI ------

        ORG     00H
        bsf     STATUS,RP0

        movlw   00011111B
        movwf   TRISA

        movlw   11111110B
        movwf   TRISB

        bcf     STATUS,RP0

        bsf     PORTB,LED
```

Direttiva che stabilisce in quale punto della memoria programmi inserire le successive istruzioni.

Mette a "1" il bit RP0 del registro STATUS che stabilisce a quale banco del file register ci si riferisce. Stiamo infatti per agire sui registri TRISA e TRISB che stanno nel banco 1

Copia nel registro W il byte 00011111 (tutti i pin della porta A sono degli input)

Copia nel registro TRISA il contenuto di W precedentemente copiato

# IMPOSTAZIONE DEI REGISTRI

```
;-------- IMPOSTAZIONE DEI REGISTRI ------

        ORG     00H
        bsf     STATUS,RP0

        movlw   00011111B
        movwf   TRISA

        movlw   11111110B
        movwf   TRISB

        bcf     STATUS,RP0

        bsf     PORTB,LED
```

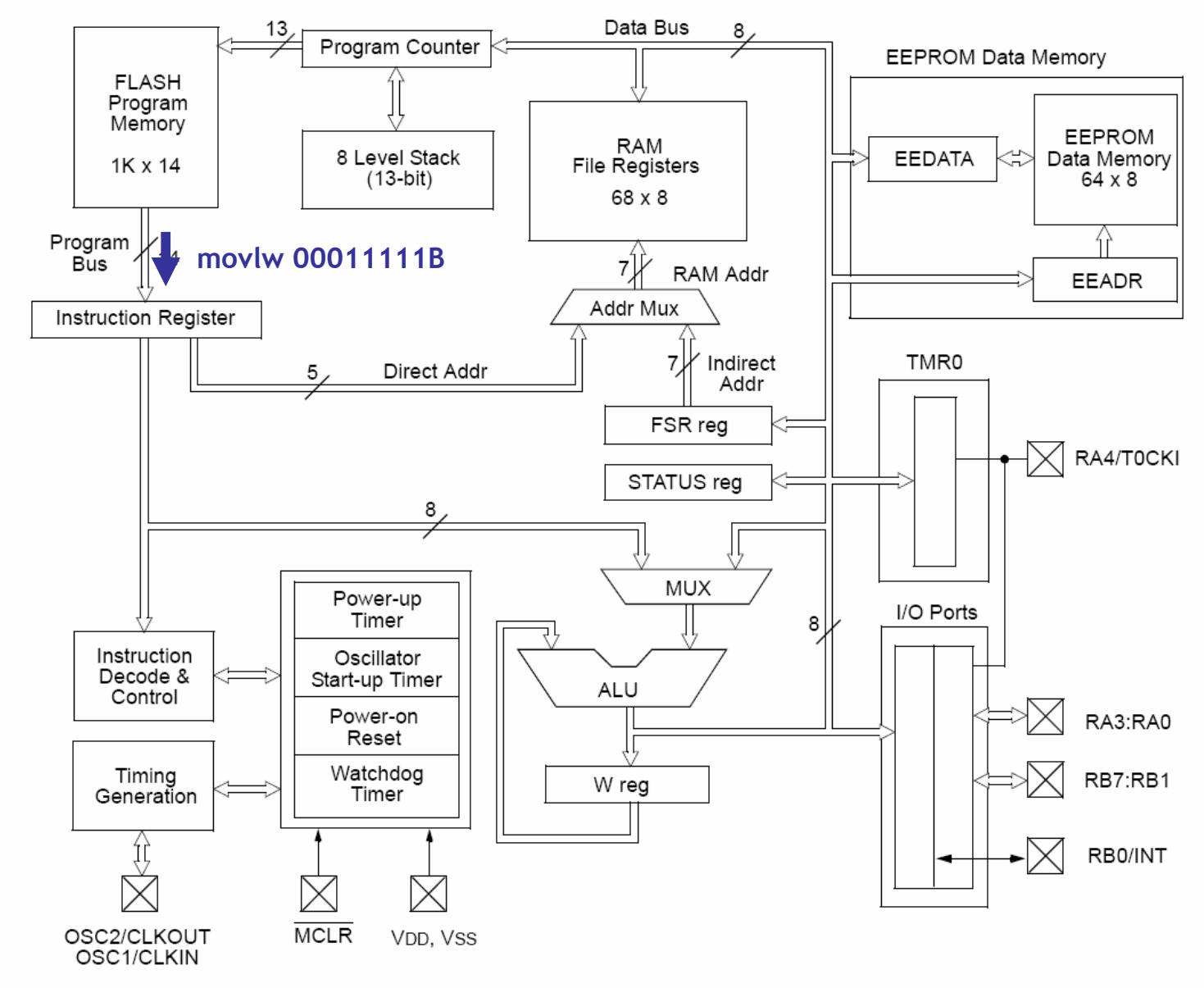
Direttiva che stabilisce in quale punto della memoria programmi inserire le successive istruzioni.

Mette a "1" il bit RP0 del registro STATUS che stabilisce a quale banco del file register ci si riferisce. Stiamo infatti per agire sui registri TRISA e TRISB che stanno nel banco 1

Copia nel registro W il byte 00011111 (tutti i pin della porta A sono degli input)

Copia nel registro TRISA il contenuto di W precedentemente copiato

# PERCHE' PER COPIARE UN DATO IN UN REGISTRO DOBBIAMO PASSARE DA W?

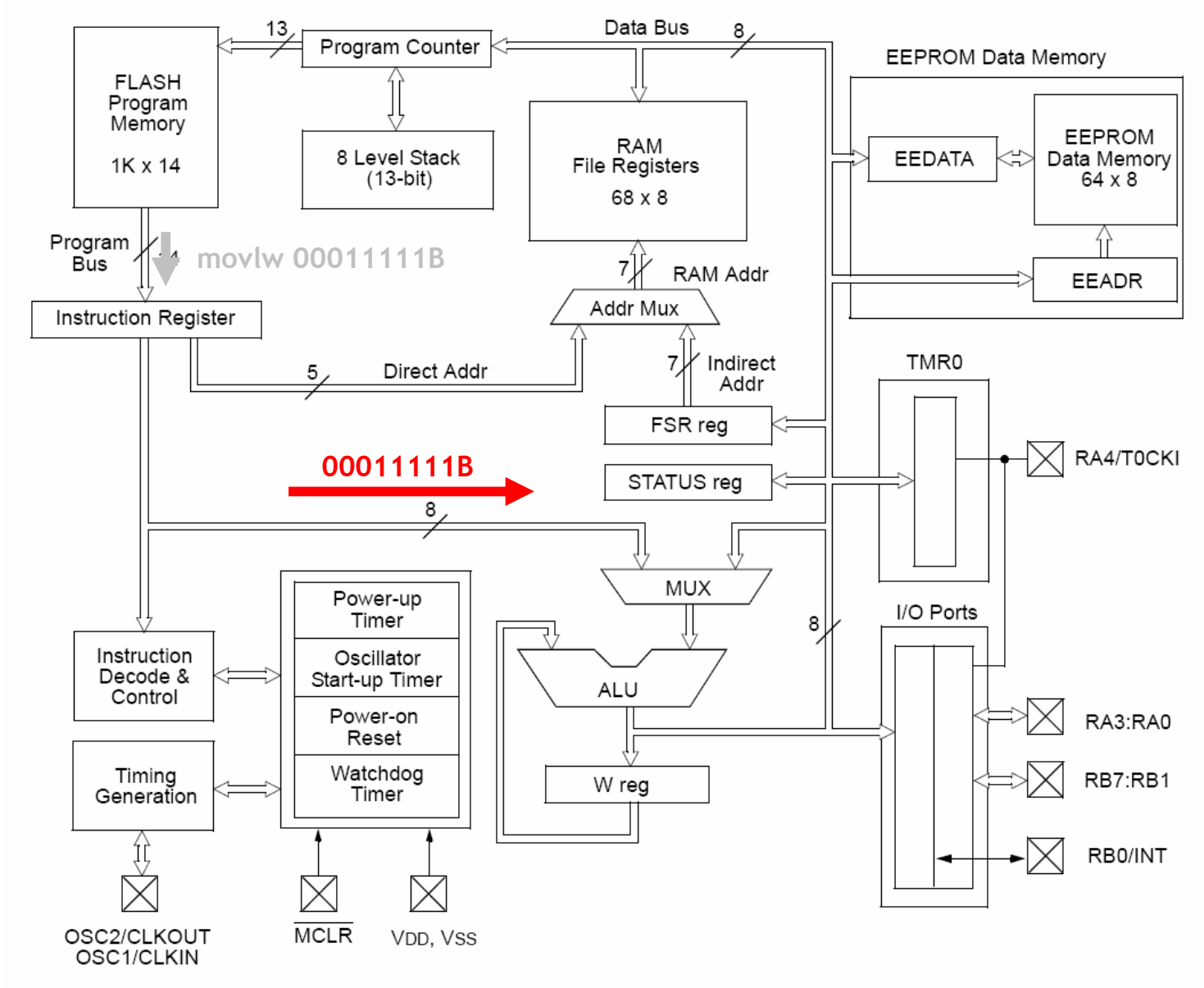



```
...
movlw   00011111B
movwf   TRISA
...
```

Il dato da memorizzare viene inviato verso l'ALU ed il registro W

Il dato viene memorizzato nel registro W

# PERCHE' PER COPIARE UN DATO IN UN REGISTRO DOBBIAMO PASSARE DA W?




...

movlw 00011111B

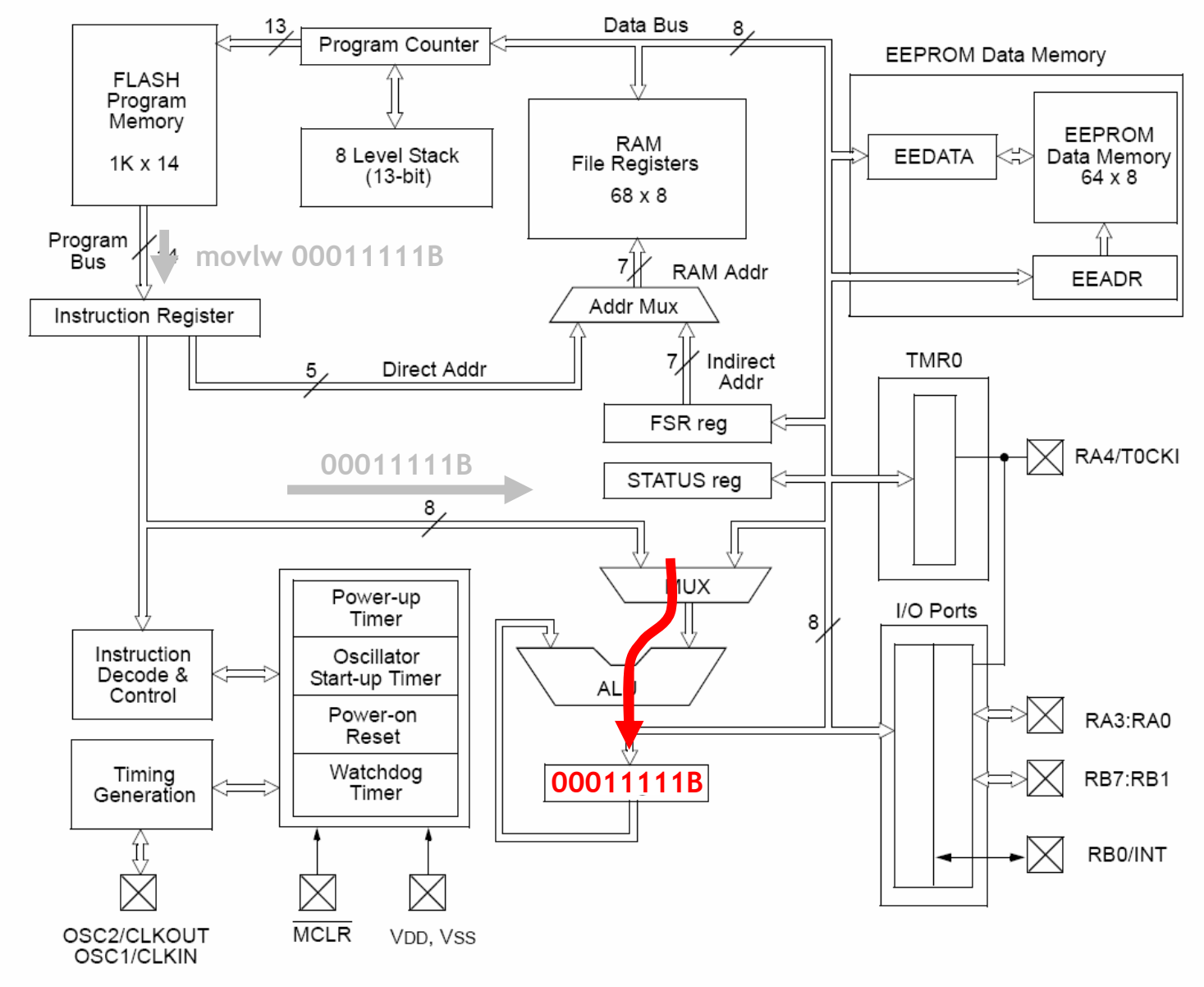
movwf TRISA

...

Il dato da memorizzare viene inviato verso l'ALU ed il registro W

Il dato viene memorizzato nel registro W

# PERCHE' PER COPIARE UN DATO IN UN REGISTRO DOBBIAMO PASSARE DA W?

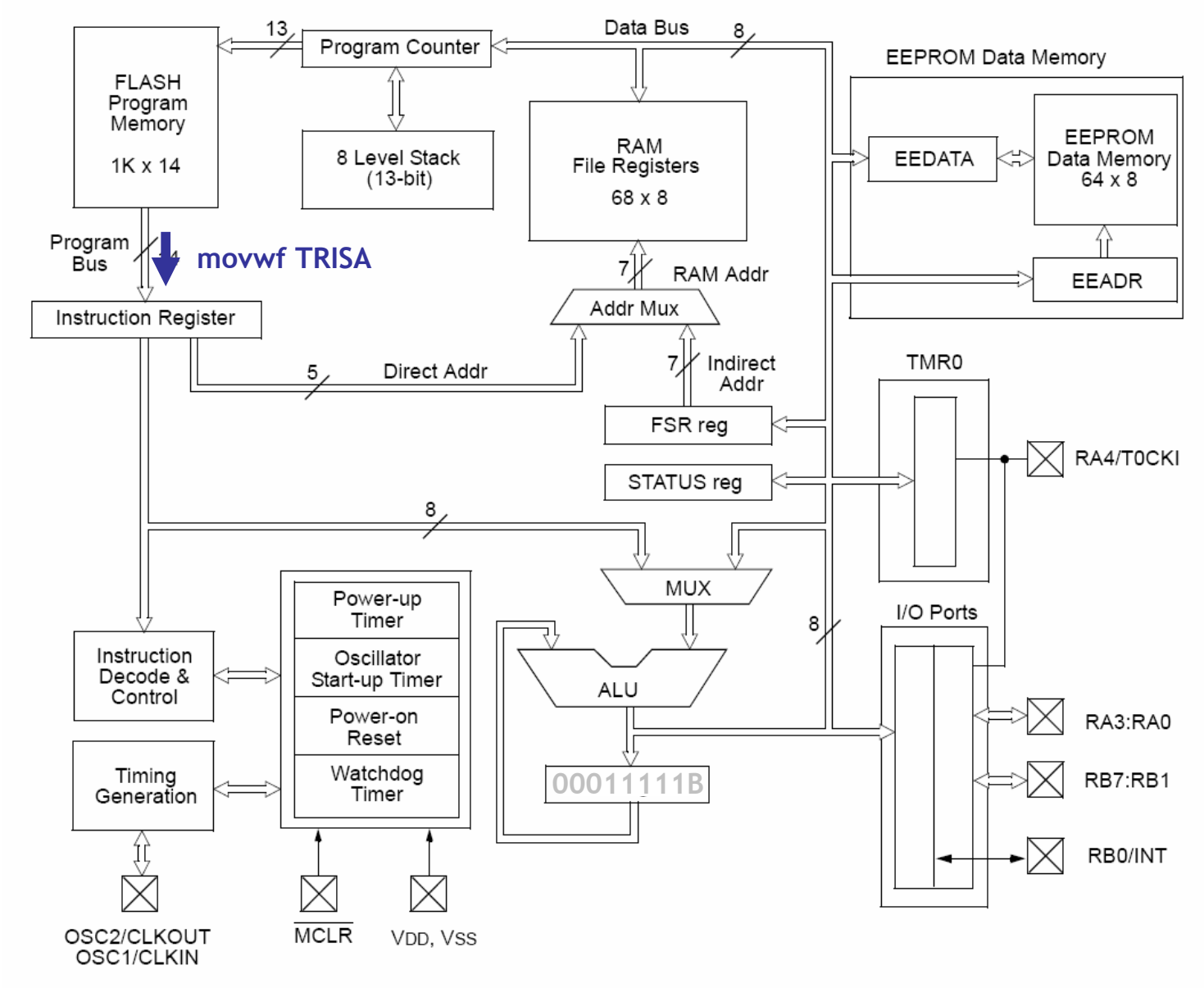



```
...
movlw   00011111B
movwf   TRISA
...
```

Il dato da memorizzare viene inviato verso l'ALU ed il registro W

Il dato viene memorizzato nel registro W

# PERCHE' PER COPIARE UN DATO IN UN REGISTRO DOBBIAMO PASSARE DA W?

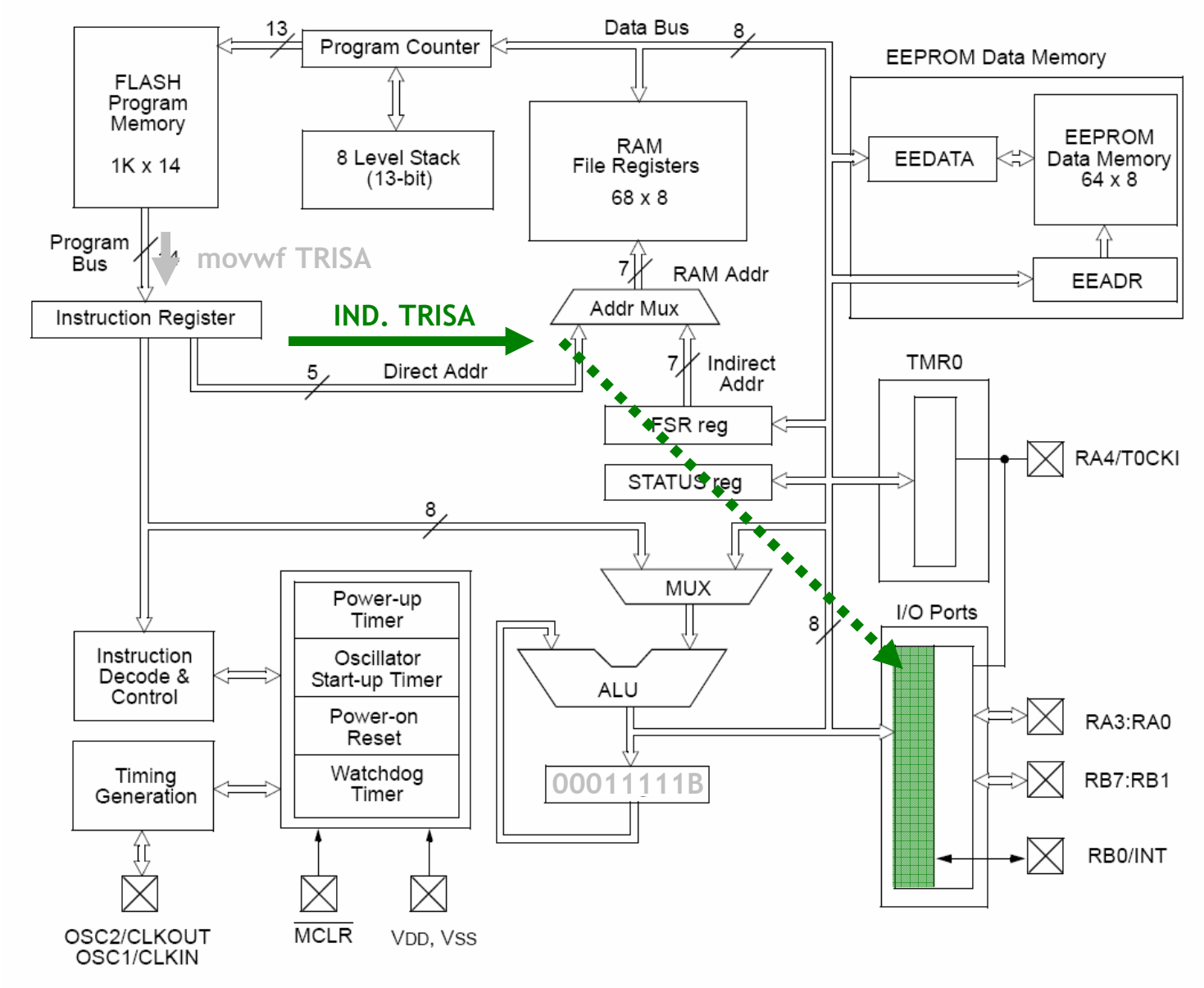



```
...
movlw   00011111B
movwf   TRISA
...
```

Viene indirizzato il registro (TRISA) nel quale verrà salvato il dato

Il dato precedentemente memorizzato in W ripassa attraverso l'ALU e raggiunge quindi il bus per essere memorizzato nel registro TRISA

# PERCHE' PER COPIARE UN DATO IN UN REGISTRO DOBBIAMO PASSARE DA W?

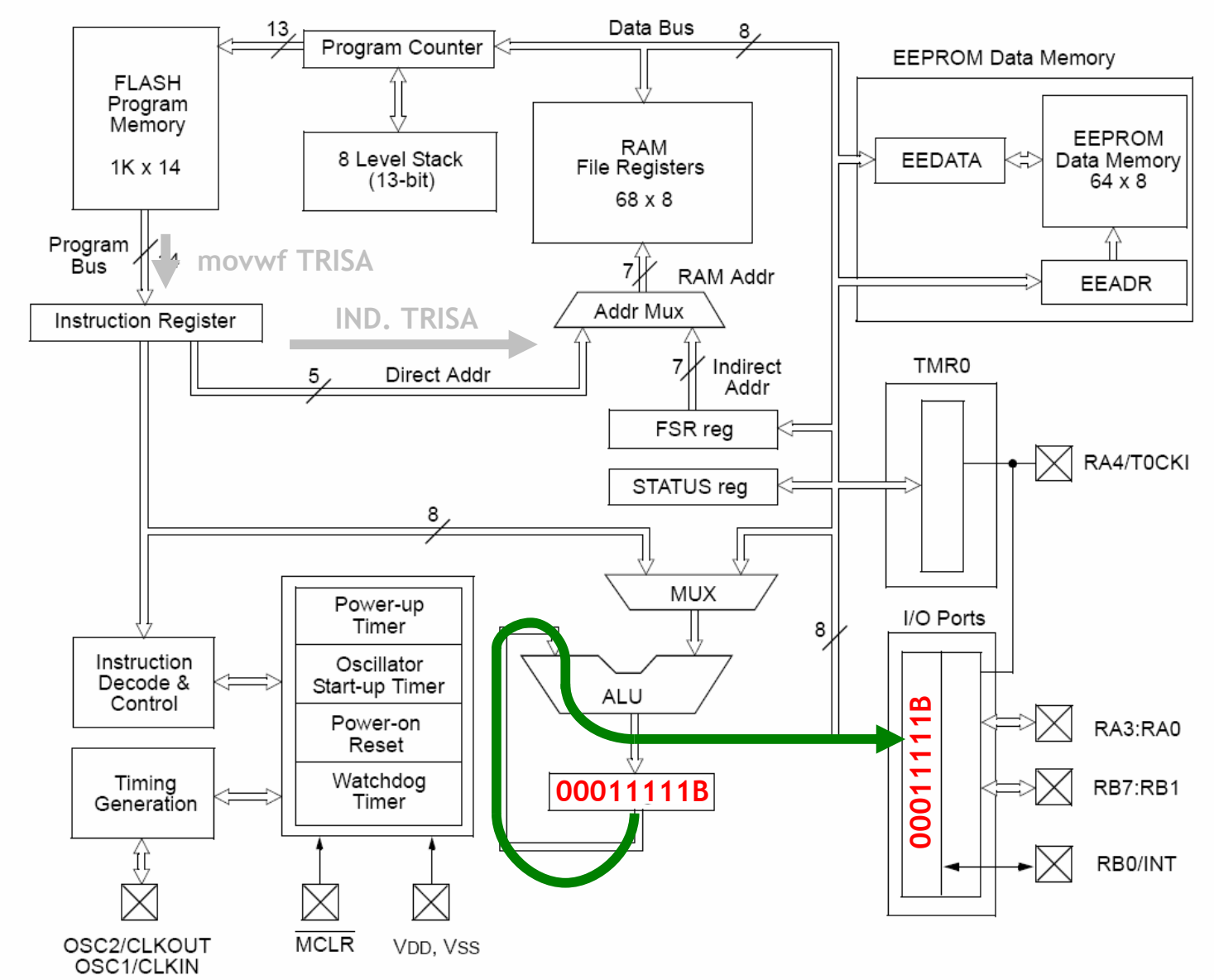



```
...
movlw   00011111B
movwf   TRISA
...
```

Viene indirizzato il registro (TRISA) nel quale verrà salvato il dato

Il dato precedentemente memorizzato in W ripassa attraverso l'ALU e raggiunge quindi il bus per essere memorizzato nel registro TRISA

# PERCHE' PER COPIARE UN DATO IN UN REGISTRO DOBBIAMO PASSARE DA W?

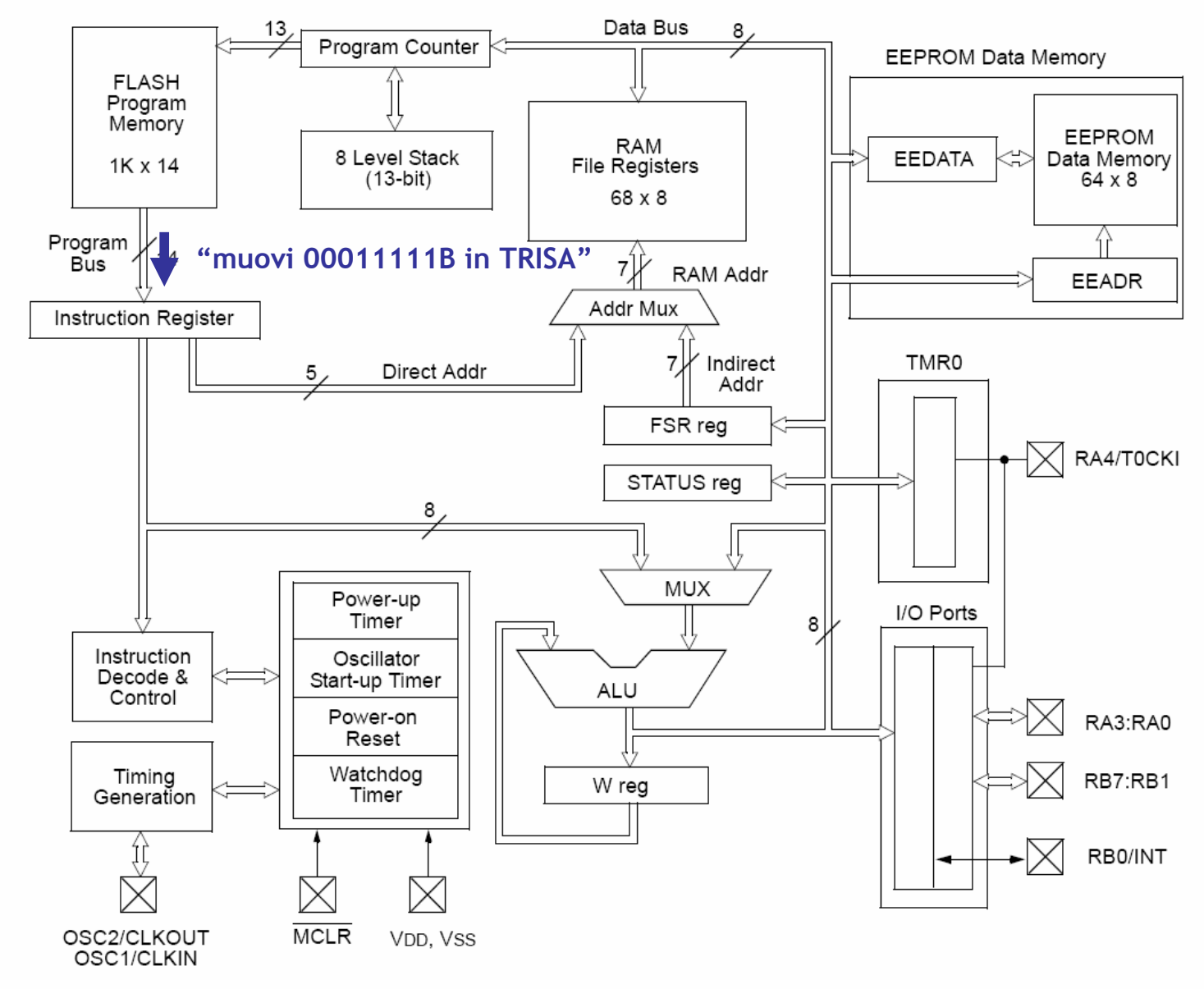



```
...
movlw   00011111B
movwf   TRISA
...
```

Viene indirizzato il registro (TRISA) nel quale verrà salvato il dato

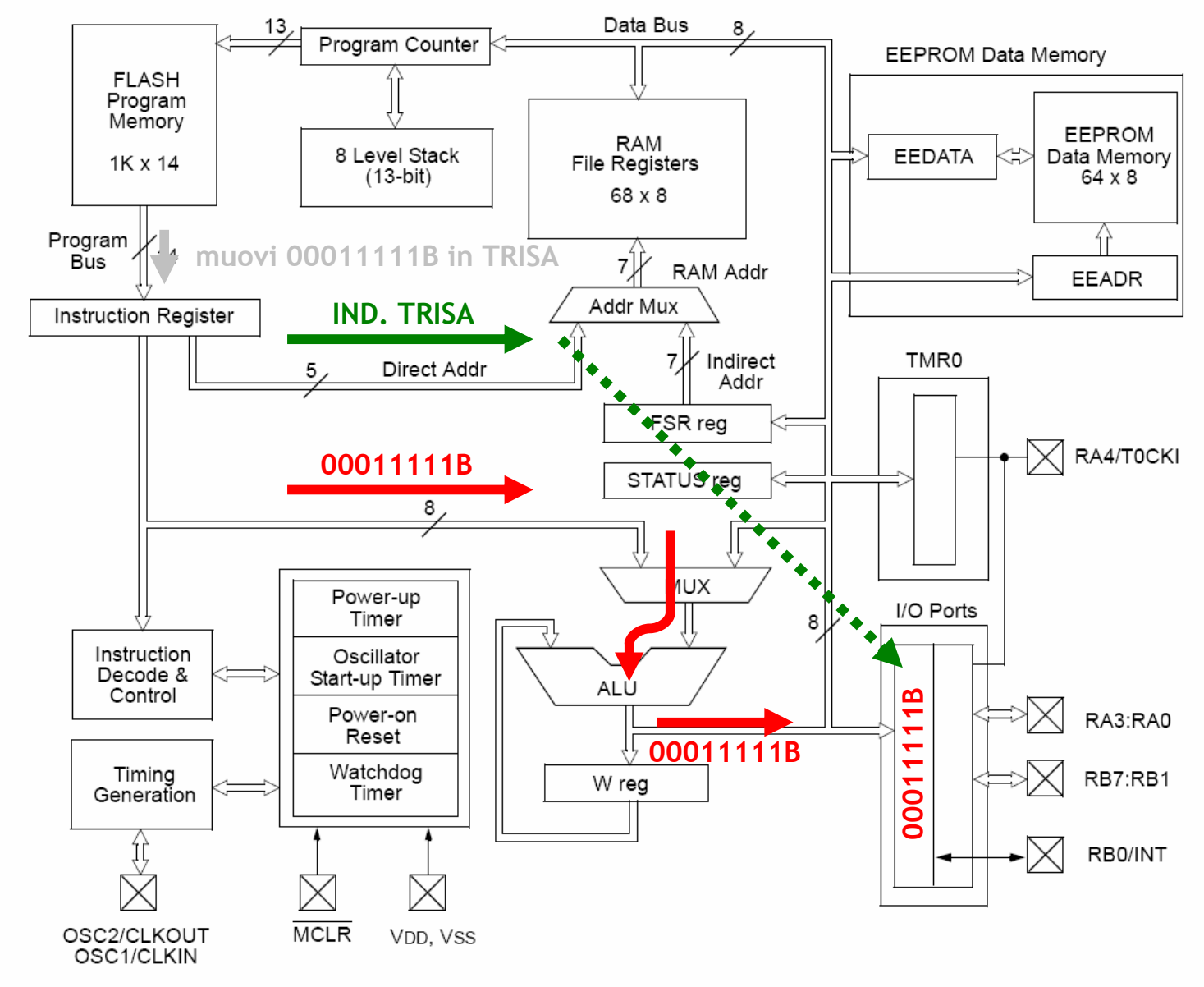
Il dato precedentemente memorizzato in W ripassa attraverso l'ALU e raggiunge quindi il bus per essere memorizzato nel registro TRISA

# PERCHE' PER COPIARE UN DATO IN UN REGISTRO DOBBIAMO PASSARE DA W?




L'istruzione viene caricata dalla memoria all'istruction register

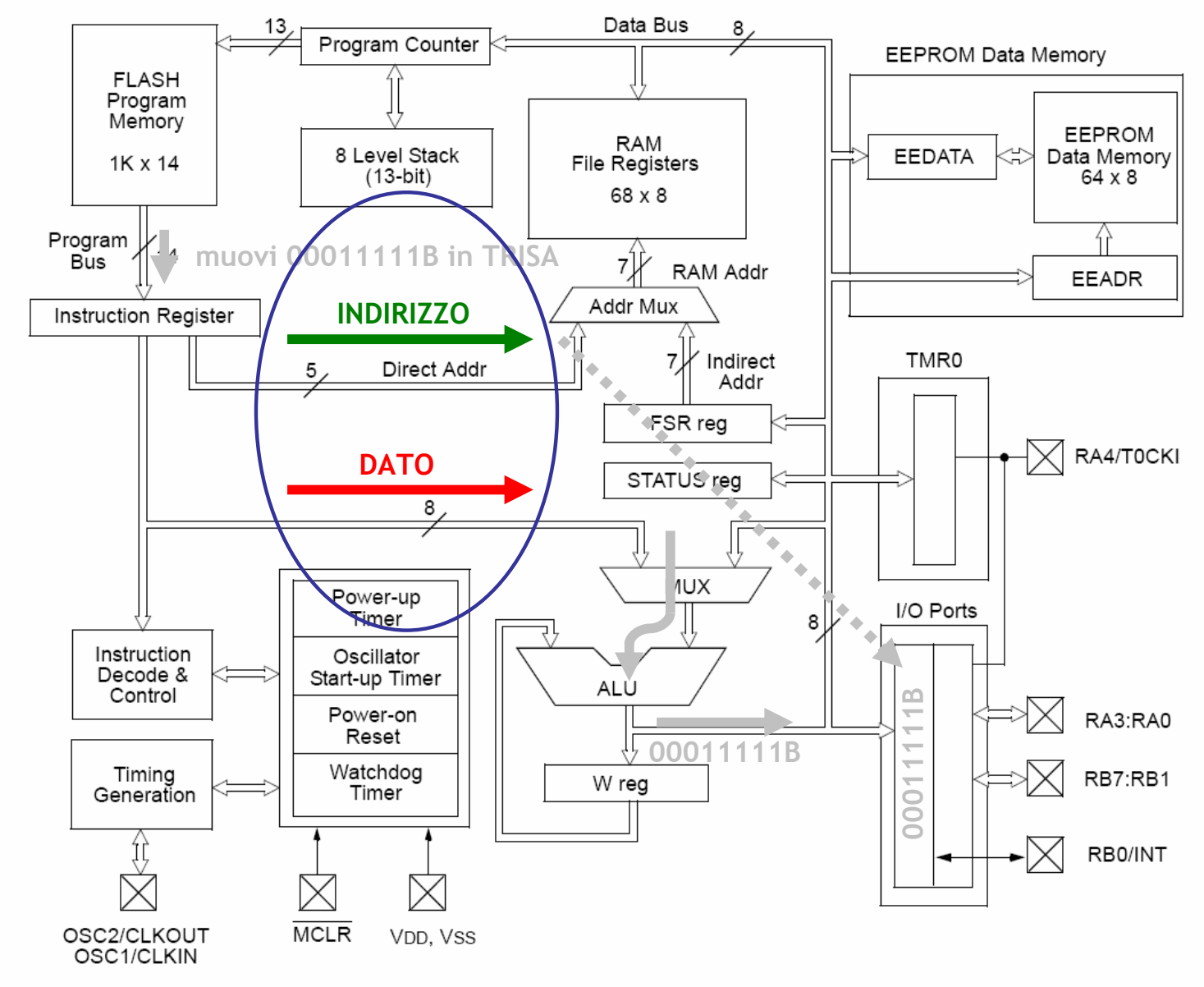
# PERCHE' PER COPIARE UN DATO IN UN REGISTRO DOBBIAMO PASSARE DA W?




L'istruzione viene caricata dalla memoria all'istruction register

E' possibile indirizzare il registro di destinazione e contemporaneamente inviare il dato da memorizzare?

# PERCHE' PER COPIARE UN DATO IN UN REGISTRO DOBBIAMO PASSARE DA W?




L'istruzione viene caricata dalla memoria all'istruction register

E' possibile indirizzare il registro di destinazione e contemporaneamente inviare il dato da memorizzare?

NO, non ci possiamo permettere di mandare dato ed indirizzo contemporaneamente!!

# E' UN PROBLEMA LEGATO ALLA DIMENSIONE DELL'OPCODE

Per i microcontrollori PIC l'opcode è grande 14 bit.

Per eseguire l'operazione di copia di un valore numerico direttamente in un registro servirebbero:

8 bit per il dato da copiare

(lavoriamo con dati a 8 bit)

7 bit per specificare il registro

(il file register è diviso in banchi da 127 celle, quindi si indirizza con 7 bit)

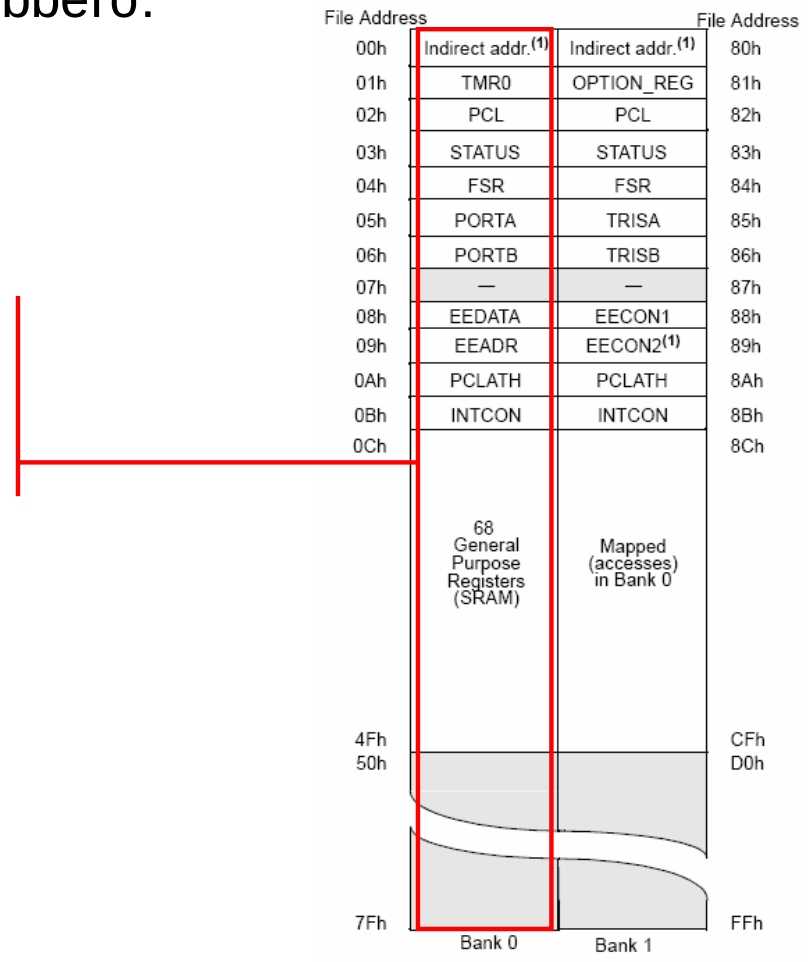
6 bit per indicare l'operazione da utilizzare

(il microcontrollore in esame ha 35 istruzioni, quindi si indica l'istruzione con 6 bit)

---

21 bit

(quindi servono 2 istruzioni per svolgere l'operazione)



# IMPOSTAZIONE DEI REGISTRI

```
;-------- IMPOSTAZIONE DEI REGISTRI ------

        ORG     00H
        bsf     STATUS,RP0

        movlw   00011111B
        movwf   TRISA

        movlw   11111110B
        movwf   TRISB

        bcf     STATUS,RP0

        bsf     PORTB,LED
```

Copia in W il numero 11111110B (B indica che si sta fornendo il numero in formato binario)

Si copia il contenuto di W nel registro TRISB (tutti, tranne il pin 0 a cui collegheremo il LED, sono inpostati come input)

Bit RP0 del registro STATUS = 0 (torniamo nel banco 0 del file register)

Accendo il LED collegato al pin RB0 mettendo a “1” il relativo bit del registro PORTB

# IMPOSTAZIONE DEI REGISTRI

```
;-------- IMPOSTAZIONE DEI REGISTRI ------

        ORG     00H
        bsf     STATUS,RP0

        movlw   00011111B
        movwf   TRISA

        movlw   11111110B
        movwf   TRISB

        bcf     STATUS,RP0

        bsf     PORTB,LED
```

Copia in W il numero 11111110B (B indica che si sta fornendo il numero in formato binario)

Si copia il contenuto di W nel registro TRISB (tutti, tranne il pin 0 a cui collegheremo il LED, sono inpostati come input)

Bit RP0 del registro STATUS = 0 (torniamo nel banco 0 del file register)

Accendo il LED collegato al pin RB0 mettendo a “1” il relativo bit del registro PORTB

# IMPOSTAZIONE DEI REGISTRI

```
;-------- IMPOSTAZIONE DEI REGISTRI ------

        ORG     00H
        bsf     STATUS,RP0

        movlw   00011111B
        movwf   TRISA

        movlw   11111110B
        movwf   TRISB

        bcf     STATUS,RP0

        bsf     PORTB,LED
```

Copia in W il numero 11111110B (B indica che si sta fornendo il numero in formato binario)

Si copia il contenuto di W nel registro TRISB (tutti, tranne il pin 0 a cui collegheremo il LED, sono inpostati come input)

Bit RP0 del registro STATUS = 0 (torniamo nel banco 0 del file register)

Accendo il LED collegato al pin RB0 mettendo a “1” il relativo bit del registro PORTB

# IMPOSTAZIONE DEI REGISTRI

```
;-------- IMPOSTAZIONE DEI REGISTRI ------

        ORG     00H
        bsf     STATUS,RP0

        movlw   00011111B
        movwf   TRISA

        movlw   11111110B
        movwf   TRISB

        bcf     STATUS,RP0

        bsf     PORTB,LED
```

Copia in W il numero 11111110B (B indica che si sta fornendo il numero in formato binario)

Si copia il contenuto di W nel registro TRISB (tutti, tranne il pin 0 a cui collegheremo il LED, sono inpostati come input)

Bit RP0 del registro STATUS = 0 (torniamo nel banco 0 del file register)

Accendo il LED collegato al pin RB0 mettendo a “1” il relativo bit del registro PORTB

# CORPO DEL PROGRAMMA

```
;-------------------- MAIN --------------------

Main

        call    Delay
        btfsc   PORTB,LED
        goto    SetToZero

        bsf     PORTB,LED
        goto    Main

SetToZero

        bcf     PORTB,LED
        goto    Main
```

Etichetta “Main”.
Serve come riferimento per funzioni di tipo Goto

Chiama una routine che genera un ritardo di tempo costante

Btfsc = “Bit Test Flag Skip Clear” Controlla lo stato attuale del pin a cui è collegato il LED. Se il LED è spento (uscita clear) salta la successiva istruzione.

Nel caso in cui il LED sia spento, il LED viene acceso ed infine l’istruzione “goto” rimanda all’inizio della routine.

Nel caso invece in cui il led sia acceso l’istruzione “goto” manda nella subroutine SetToZero dove il LED viene spento. Infine si ritorna anche in questo caso all’inizio del Main.

# CORPO DEL PROGRAMMA

```
;-------------------- MAIN --------------------

Main

        call    Delay
        btfsc   PORTB,LED
        goto    SetToZero

        bsf     PORTB,LED
        goto    Main

SetToZero

        bcf     PORTB,LED
        goto    Main
```

Etichetta "Main".
Serve come riferimento per funzioni di tipo Goto

Chiama una routine che genera un ritardo di tempo costante

Btfsc = "Bit Test Flag Skip Clear" Controlla lo stato attuale del pin a cui è collegato il LED. Se il LED è spento (uscita clear) salta la successiva istruzione.

Nel caso in cui il LED sia spento, il LED viene acceso ed infine l'istruzione "goto" rimanda all'inizio della routine.

Nel caso invece in cui il led sia acceso l'istruzione "goto" manda nella subroutine SetToZero dove il LED viene spento. Infine si ritorna anche in questo caso all'inizio del Main.

# CORPO DEL PROGRAMMA

```
;-------------------- MAIN --------------------

Main

        call    Delay
        btfsc   PORTB,LED
        goto    SetToZero

        bsf     PORTB,LED
        goto    Main

SetToZero

        bcf     PORTB,LED
        goto    Main
```

Etichetta "Main".
Serve come riferimento per funzioni di tipo Goto

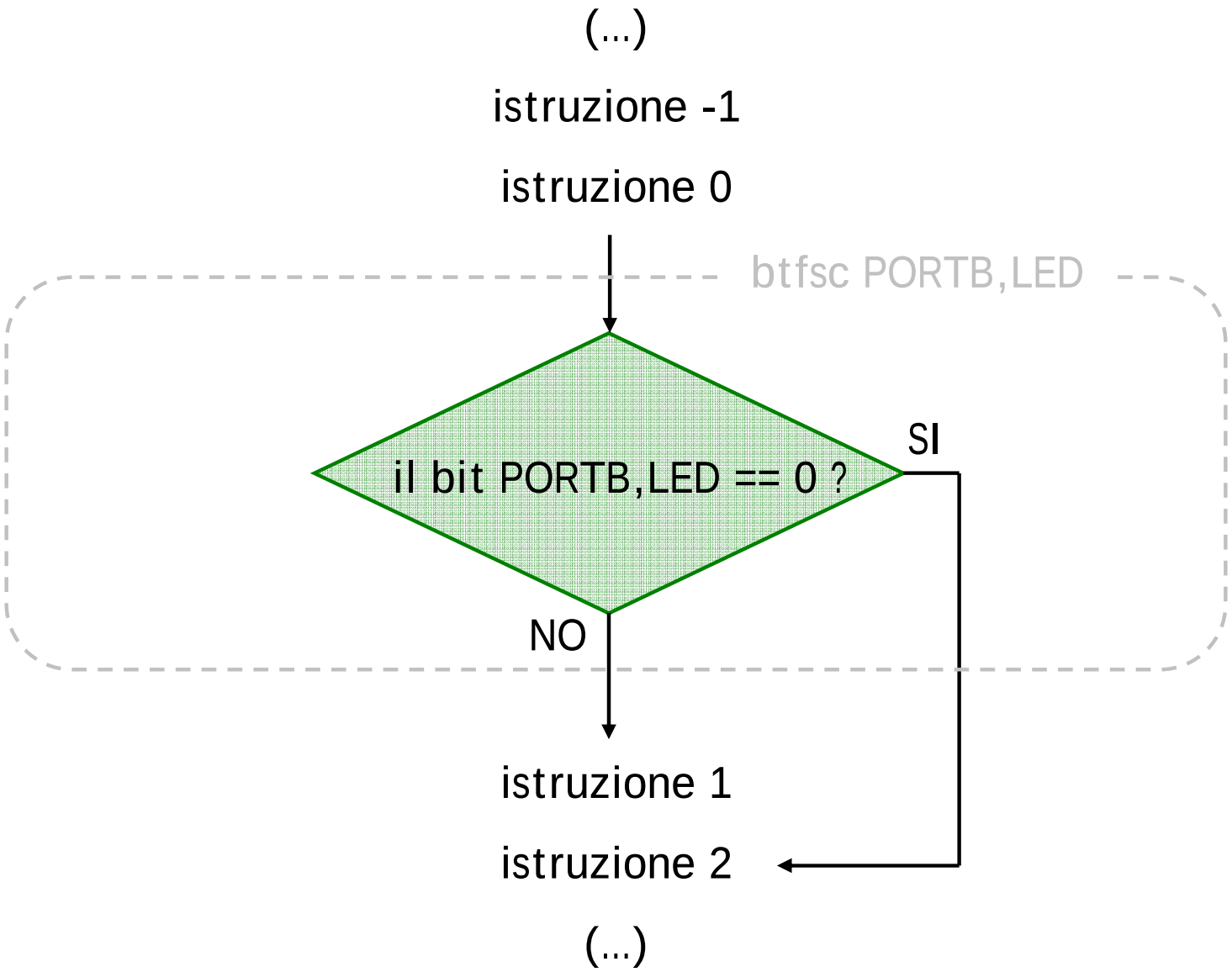
Chiama una routine che genera un ritardo di tempo costante

Btfsc = "Bit Test Flag Skip Clear"
Controlla lo stato attuale del pin a cui è collegato il LED. Se il LED è spento (uscita clear) salta la successiva istruzione.

Nel caso in cui il LED sia spento, il LED viene acceso ed infine l'istruzione "goto" rimanda all'inizio della routine.

Nel caso invece in cui il led sia acceso l'istruzione "goto" manda nella subroutine SetToZero dove il LED viene spento. Infine si ritorna anche in questo caso all'inizio del Main.

# ISTRUZIONE btfsc



# if IN ASSEMBLER

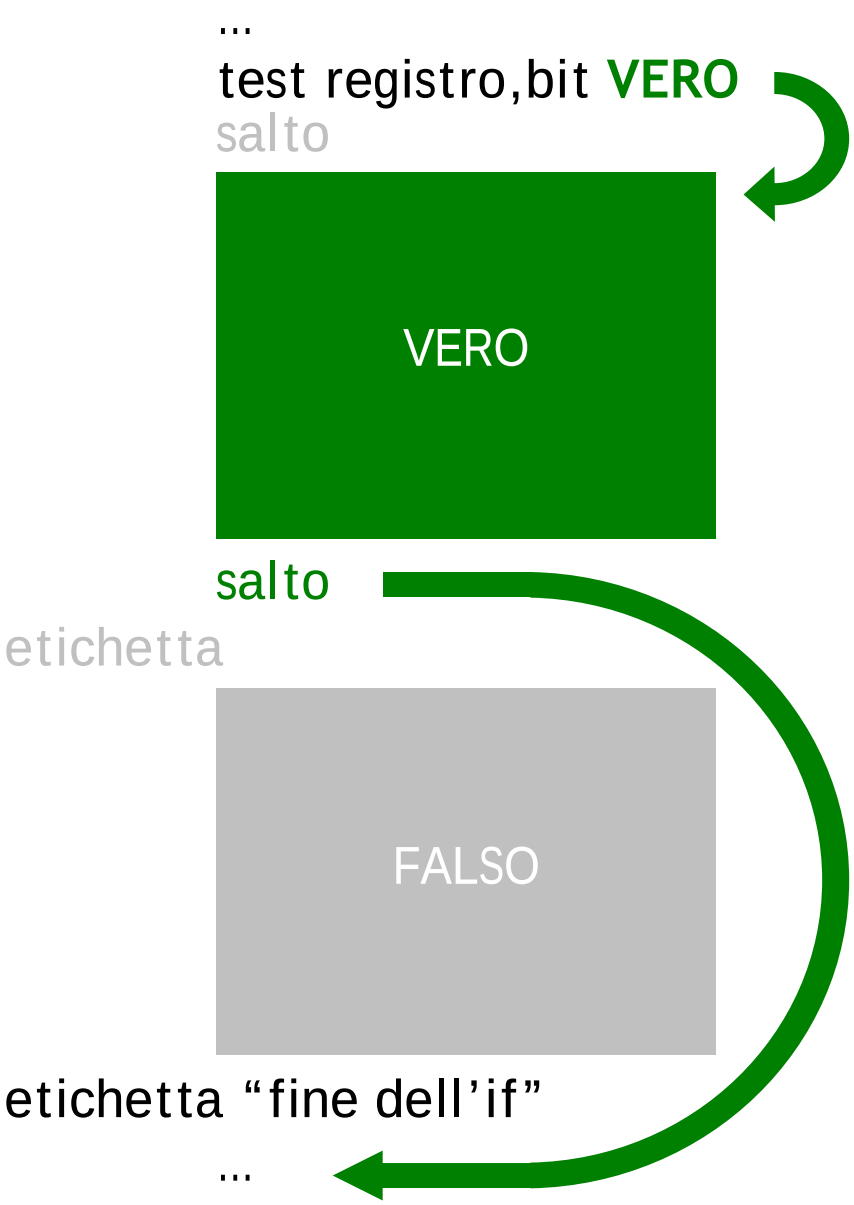

L’istruzione di test in assembler esegue il controllo sul valore di un bit e a seconda del risultato:

- l’esecuzione del firmware prosegue inalterata ma viene saltata l’istruzione successiva al test

- l’esecuzione del firmware prosegue inalterata

E’ necessario quindi organizzare correttamente la struttura del firmware dotandola di opportuni salti al fine di gestire l’istruzione di test

# if IN ASSEMBLER




L’istruzione di test in assembler esegue il controllo sul valore di un bit e a seconda del risultato:

- l’esecuzione del firmware prosegue inalterata ma viene saltata l’istruzione successiva al test
- l’esecuzione del firmware prosegue inalterata

E’ necessario quindi organizzare correttamente la struttura del firmware dotandola di opportuni salti al fine di gestire l’istruzione di test

# if IN ASSEMBLER


L’istruzione di test in assembler esegue il controllo sul valore di un bit e a seconda del risultato:

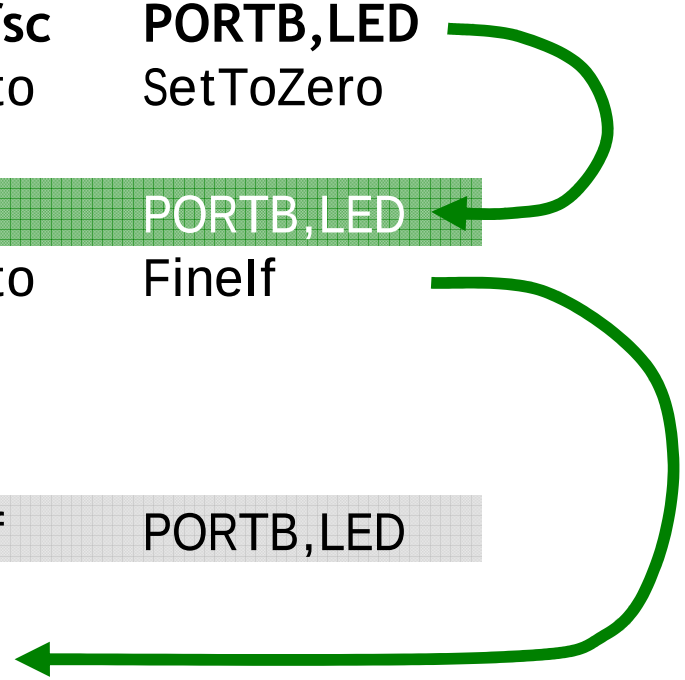
- l’esecuzione del firmware prosegue inalterata ma viene saltata l’istruzione successiva al test

- l’esecuzione del firmware prosegue inalterata

E’ necessario quindi organizzare correttamente la struttura del firmware dotandola di opportuni salti al fine di gestire l’istruzione di test

# CORPO DEL PROGRAMMA

```
;-------------------- MAIN --------------------

Main

        call    Delay
        btfsc   PORTB,LED
        goto    SetToZero

        bsf     PORTB,LED
        goto    FineIf

SetToZero

        bcf     PORTB,LED

FineIf

        goto    Main
```



**ATTENZIONE!!!!**
**Versione alternativa NON ottimizzata e non utilizzata nel seguito.**

# CORPO DEL PROGRAMMA

```
;-------------------- MAIN --------------------

Main

        call    Delay
        btfsc   PORTB,LED
        goto    SetToZero

        bsf     PORTB,LED
        goto    Main

SetToZero

        bcf     PORTB,LED
        goto    Main
```

Etichetta "Main".
Serve come riferimento per funzioni di tipo Goto

Chiama una routine che genera un ritardo di tempo costante

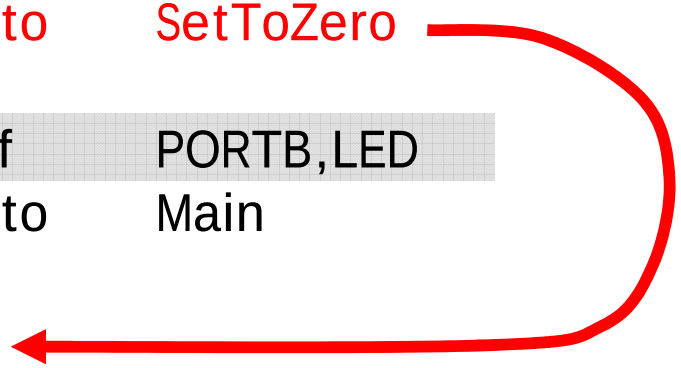
Btfsc = "Bit Test Flag Skip Clear" Controlla lo stato attuale del pin a cui è collegato il LED. Se il LED è spento (uscita clear) salta la successiva istruzione.

Nel caso in cui il LED sia spento, il LED viene acceso ed infine l'istruzione "goto" rimanda all'inizio della routine.

Nel caso invece in cui il led sia acceso l'istruzione "goto" manda nella subroutine SetToZero dove il LED viene spento. Infine si ritorna anche in questo caso all'inizio del Main.

# CORPO DEL PROGRAMMA

```
;-------------------- MAIN --------------------

Main

        call    Delay
        btfsc   PORTB,LED
        goto    SetToZero

        bsf     PORTB,LED
        goto    Main

SetToZero

        bcf     PORTB,LED
        goto    Main
```



Etichetta "Main".
Serve come riferimento per funzioni di tipo Goto

Chiama una routine che genera un ritardo di tempo costante

Btfsc = "Bit Test Flag Skip Clear"
Controlla lo stato attuale del pin a cui è collegato il LED. Se il LED è spento (uscita clear) salta la successiva istruzione.

Nel caso in cui il LED sia spento, il LED viene acceso ed infine l'istruzione "goto" rimanda all'inizio della routine.

Nel caso invece in cui il led sia acceso l'istruzione "goto" manda nella subroutine SetToZero dove il LED viene spento. Infine si ritorna anche in questo caso all'inizio del Main.

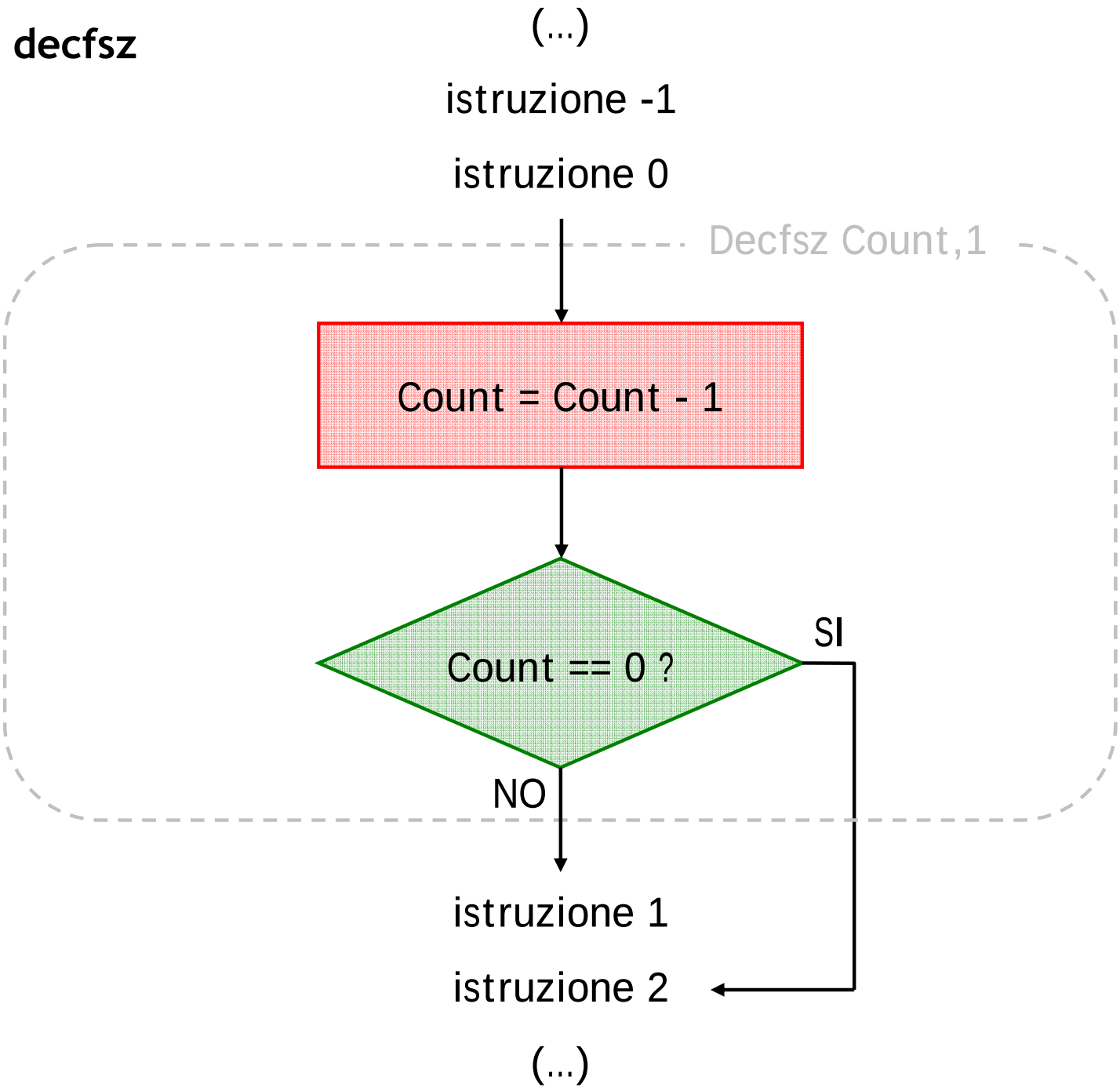
# LA ROUTINE "DELAY"

```
;-------------------- DELAY --------------------

Delay
        clrf    Count
        clrf    Count+1
DelayLoop
        decfsz  Count,1
        goto    DelayLoop
        decfsz  Count+1,1
        goto    DelayLoop
        return

        END
```

Azzera la prima e poi la seconda cella di memoria riservate all'inizio per la variabile "Count".
L'azzeramento della cella, caso particolare di scrittura di dati, può essere eseguito con una solo istruzione dedicata. Il problema dell'opcode non si presenta in quanto non sono necessari gli 8 bit del dato (sempre = 0)

| Count+1 | Count |
|---|---|
| 0 | 0 |

# ISTRUZIONE decfsz



## REGISTER 2-1: STATUS REGISTER (ADDRESS 03h, 83h)

| R/W-0 | R/W-0 | R/W-0 | R-1 | R-1 | R/W-x | R/W-x | R/W-x |
|---|---|---|---|---|---|---|---|
| IRP | RP1 | RP0 | $\overline{TO}$ | $\overline{PD}$ | Z | DC | C |
| bit 7 | | | | | | | bit 0 |

bit 7-6 **Unimplemented:** Maintain as '0'

bit 5 **RP0**: Register Bank Select bits (used for direct addressing)
01 = Bank 1 (80h - FFh)
00 = Bank 0 (00h - 7Fh)

bit 4 **$\overline{TO}$**: Time-out bit
1 = After power-up, CLRWDT instruction, or SLEEP instruction
0 = A WDT time-out occurred

bit 3 **$\overline{PD}$**: Power-down bit
1 = After power-up or by the CLRWDT instruction
0 = By execution of the SLEEP instruction

bit 2 **Z**: Zero bit
1 = The result of an arithmetic or logic operation is zero
0 = The result of an arithmetic or logic operation is not zero

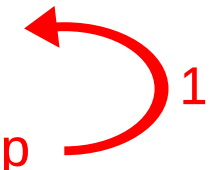
bit 1 **DC**: Digit carry/$\overline{borrow}$ bit (ADDWF, ADDLW, SUBLW, SUBWF instructions) (for $\overline{borrow}$, the polarity is reversed)
1 = A carry-out from the 4th low order bit of the result occurred
0 = No carry-out from the 4th low order bit of the result

bit 0 **C**: Carry/$\overline{borrow}$ bit (ADDWF, ADDLW, SUBLW, SUBWF instructions) (for $\overline{borrow}$, the polarity is reversed)
1 = A carry-out from the Most Significant bit of the result occurred
0 = No carry-out from the Most Significant bit of the result occurred

**Note:** A subtraction is executed by adding the two's complement of the second operand. For rotate (RRF, RLF) instructions, this bit is loaded with either the high or low order bit of the source register.

# LA ROUTINE "DELAY"

```
;-------------------- DELAY --------------------

Delay
        clrf    Count
        clrf    Count+1
DelayLoop
        decfsz  Count,1
        goto    DelayLoop
        decfsz  Count+1,1
        goto    DelayLoop
        return

        END
```

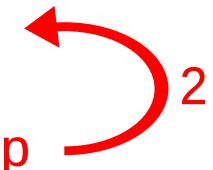



Decrementa di 1 il primo byte di Count, se, dopo il decremento, Count == 0 allora viene saltata la successiva istruzione. Altrimenti riesegue il decremento.

| Count+1 | Count |
|---|---|
| 0 | 255 |

# LA ROUTINE "DELAY"

```
;-------------------- DELAY --------------------

Delay
        clrf    Count
        clrf    Count+1
DelayLoop
        decfsz  Count,1
        goto    DelayLoop
        decfsz  Count+1,1
        goto    DelayLoop
        return

        END
```

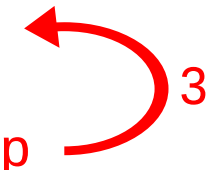



Decrementa di 1 il primo byte di Count, se, dopo il decremento, Count == 0 allora viene saltata la successiva istruzione. Altrimenti riesegue il decremento.

| Count+1 | Count |
|---|---|
| 0 | 254 |

# LA ROUTINE "DELAY"

```
;-------------------- DELAY --------------------

Delay
        clrf    Count
        clrf    Count+1
DelayLoop
        decfsz  Count,1
        goto    DelayLoop
        decfsz  Count+1,1
        goto    DelayLoop
        return

        END
```




Decrementa di 1 il primo byte di Count, se, dopo il decremento, Count == 0 allora viene saltata la successiva istruzione. Altrimenti riesegue il decremento.

| Count+1 | Count |
|---|---|
| 0 | 253 |

# LA ROUTINE "DELAY"

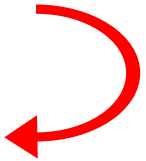
```
;-------------------- DELAY --------------------

Delay
        clrf    Count
        clrf    Count+1
DelayLoop
        decfsz  Count,1
        goto    DelayLoop
        decfsz  Count+1,1
        goto    DelayLoop
        return

        END
```


Decrementa di 1 il primo byte di Count, se, dopo il decremento, Count == 0 allora viene saltata la successiva istruzione. Altrimenti riesegue il decremento.

| Count+1 | Count |
|---|---|
| 0 | 1 |

# LA ROUTINE "DELAY"

```
;-------------------- DELAY --------------------

Delay
        clrf    Count
        clrf    Count+1
DelayLoop
        decfsz  Count,1
        goto    DelayLoop
        decfsz  Count+1,1
        goto    DelayLoop
        return

        END
```

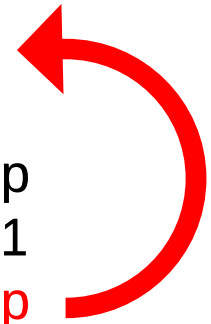


Il secondo byte di Count viene decrementato di 1 ogni volta che il primo byte si azzera. Anche qui, dopo 256 decrementi, si salta l'istruzione "goto".

| Count+1 | Count |
|---|---|
| 255 | 0 |

# LA ROUTINE "DELAY"

```
;------------------- DELAY -------------------

Delay
        clrf    Count
        clrf    Count+1
DelayLoop
        decfsz  Count,1
        goto    DelayLoop
        decfsz  Count+1,1
        goto    DelayLoop
        return

        END
```

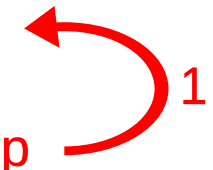


Il secondo byte di Count viene decrementato di 1 ogni volta che il primo byte si azzera. Anche qui, dopo 256 decrementi, si salta l'istruzione "goto".

| Count+1 | Count |
|---|---|
| 255 | 0 |

# LA ROUTINE "DELAY"

```
;-------------------- DELAY --------------------

Delay
        clrf    Count
        clrf    Count+1
DelayLoop
        decfsz  Count,1
        goto    DelayLoop
        decfsz  Count+1,1
        goto    DelayLoop
        return

        END
```

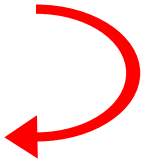



Decrementa di 1 il primo byte di Count, se, dopo il decremento, Count == 0 allora viene saltata la successiva istruzione. Altrimenti riesegue il decremento.

| Count+1 | Count |
|---|---|
| 255 | 255 |

# LA ROUTINE "DELAY"

```
;-------------------- DELAY --------------------

Delay
        clrf    Count
        clrf    Count+1
DelayLoop
        decfsz  Count,1
        goto    DelayLoop
        decfsz  Count+1,1
        goto    DelayLoop
        return

        END
```

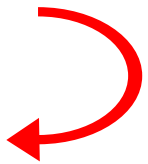


Il secondo byte di Count viene decrementato di 1 ogni volta che il primo byte si azzera. Anche qui, dopo 256 decrementi, si salta l'istruzione "goto".

| Count+1 | Count |
|---|---|
| 1 | 0 |

# LA ROUTINE "DELAY"

```
;-------------------- DELAY --------------------

Delay
        clrf    Count
        clrf    Count+1
DelayLoop
        decfsz  Count,1
        goto    DelayLoop
        decfsz  Count+1,1
        goto    DelayLoop
        return

        END
```



Decrementa di 1 il primo byte di Count, se, dopo il decremento, Count == 0 allora viene saltata la successiva istruzione. Altrimenti riesegue il decremento.

| Count+1 | Count |
|---|---|
| 0 | 0 |

# LA ROUTINE "DELAY"

```
;-------------------- DELAY --------------------

Delay
        clrf    Count
        clrf    Count+1
DelayLoop
        decfsz  Count,1
        goto    DelayLoop
        decfsz  Count+1,1
        goto    DelayLoop
        return

        END
```

Ritorna dalla routine nel Main.

Segna la fine del programma

# LA ROUTINE "DELAY"

```
;-------------------- DELAY --------------------

Delay
        clrf    Count
        clrf    Count+1
DelayLoop
        decfsz  Count,1
        goto    DelayLoop
        decfsz  Count+1,1
        goto    DelayLoop
        return

        END
```

Ritorna dalla routine nel Main.

Segna la fine del programma

# LA ROUTINE DELAY

```
Delay
        clrf    Count
        clrf    Count+1
DelayLoop
        decfsz  Count,1
        goto    DelayLoop
        decfsz  Count+1,1
        goto    DelayLoop
        return
```

La routine Delay genera un ritardo di tempo sfruttando l'esecuzione di cicli di istruzioni e la durata temporale di ognuna delle istruzioni.

VANTAGGI

1. Il ritardo sintetizzato può essere molto preciso

SVANTAGGI

1. La modifica del ritardo sintetizzato non è semplice
2. Il microcontrollore non può eseguire nessun'altra istruzione mentre esegue la ruotine di ritardo (a meno di non utilizzare un interrupt che la interrompa)
3. Questo tipo di approccio non è facilmente utilizzabile con linguaggi ad alto livello

# LA ROUTINE DELAY

## Quanto vale il ritardo sintetizzato?

Leggendo il datasheet del microcontrollore è possibile conoscere la durata di ogni istruzione. In genere ogni istruzione dura 4 volte il periodo di clock (1 ciclo) del micro, ma ci sono alcune eccezioni, come le istruzioni goto, call e decfsz che possono durare 8 periodi di clock. Ipotiziamo un quarzo da 4 MHz.

CLRF dura 1 ciclo, quindi 1 µS

DECFSZ, quando non si verifica la condizione di salto della successiva istruzione dura 1 ciclo.
Quando si verifica il salto dura 2 cicli.

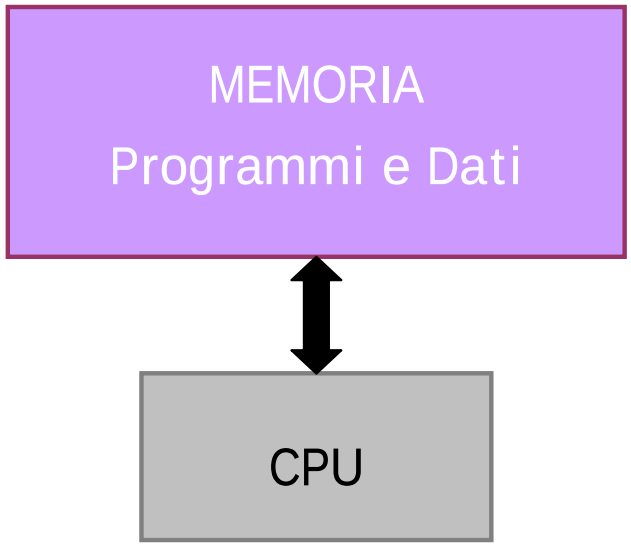
GOTO e RETURN durano sempre 2 cicli

```
Delay
            clrf    Count
            clrf    Count+1
DelayLoop
            decfsz  Count,1
            goto    DelayLoop
            decfsz  Count+1,1
            goto    DelayLoop
            return
```

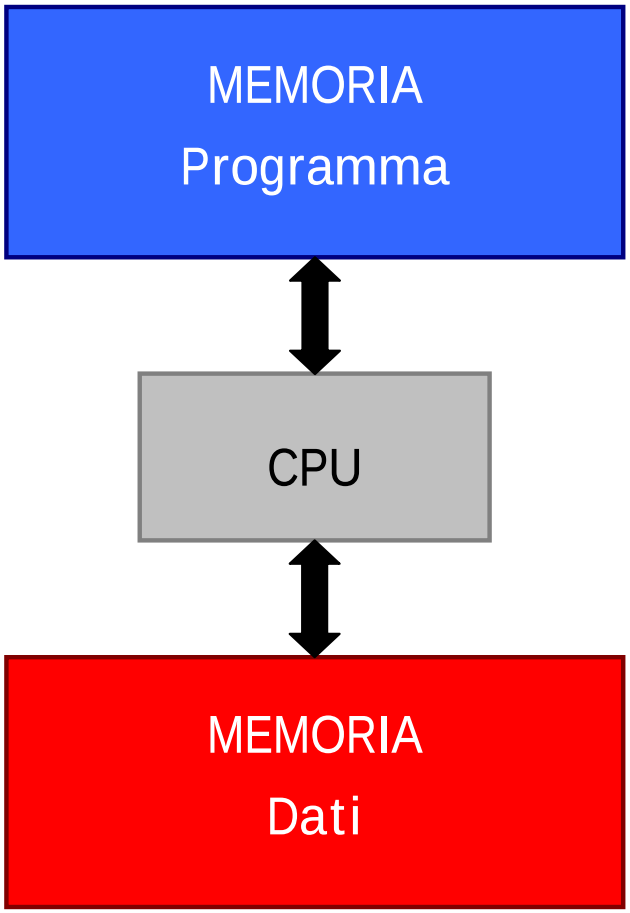
2+[(1+2)*256-1+1+2]*256-1+2 = 197123 µS

Volendo essere più precisi andrebbero anche considerati i tempi necessari alle istruzioni nel Main per chiamare la routine

# PERCHE' ALCUNE ISTRUZIONI POSSONO DURARE 1 O 2 CICLI?



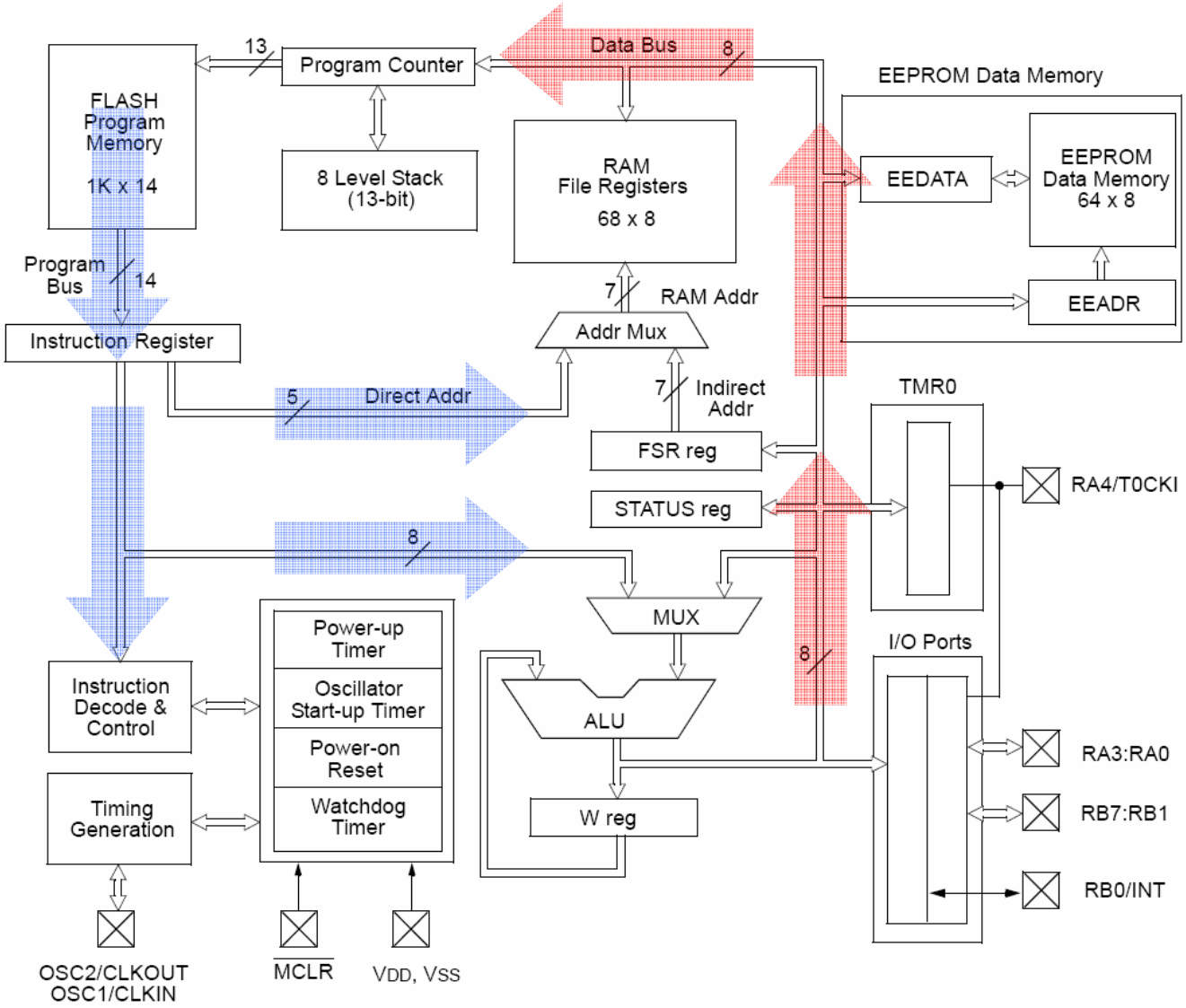

Von-Neumann




Harvard

# PERCHE' ALCUNE ISTRUZIONI POSSONO DURARE 1 O 2 CICLI?

La procedura di esecuzione di una singola istruzione qualunque è divisa in varie fasi e richiede un tempo totale per l'esecuzione pari a **8 PERIODI DI CLOCK**



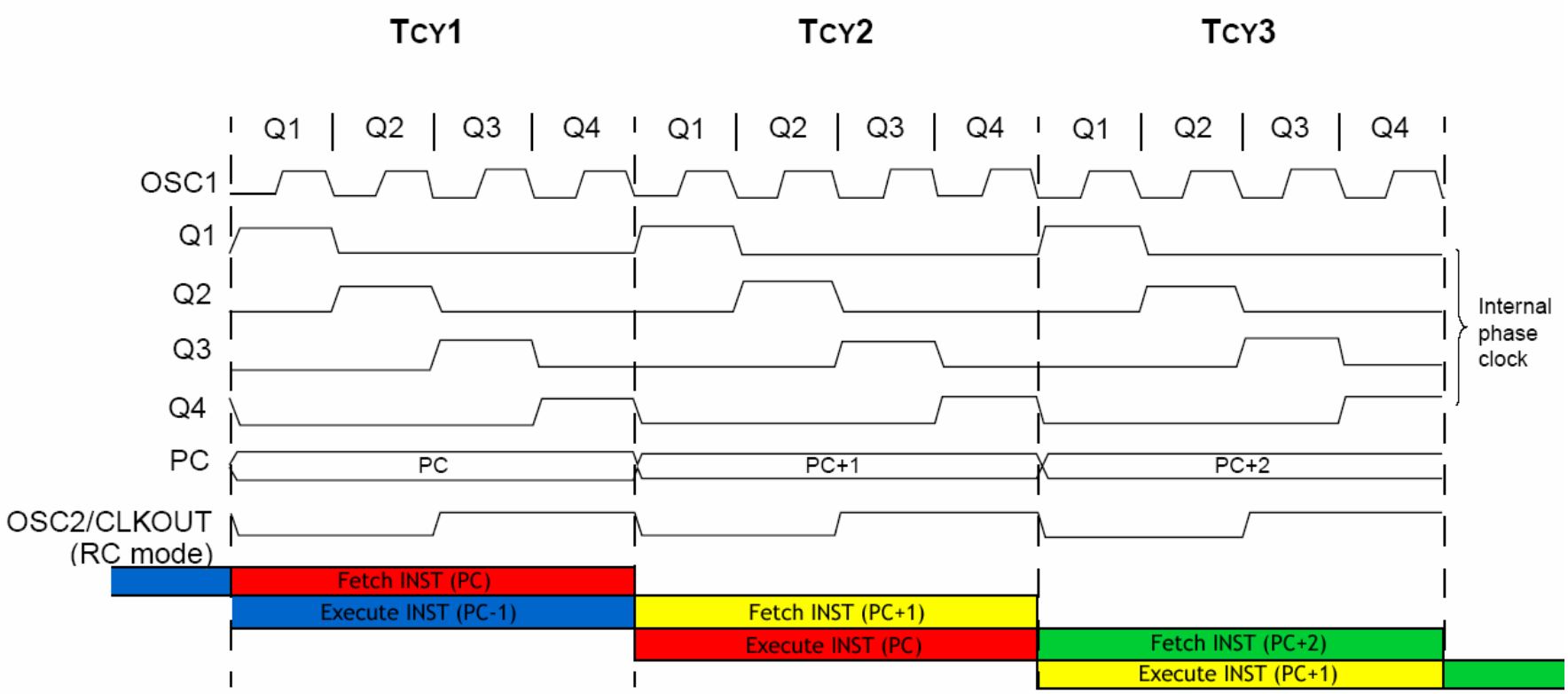

Al fine di velocizzare l'esecuzione delle istruzioni i microcontrollori PIC utilizzano un'architettura **Harvard** al posto della classica **Von-Neumann**. L'architettura Harvard presenta memorie separate per dati e programma che vengono indirizzati tramite due bus distinti.

DATI

PROGRAMMA

Questa struttura permette, **NELLA MAGGIOR PARTE DEI CASI**, di eseguire contemporaneamente L'ESECUZIONE di una istruzione e il FETCH dell'istruzione successiva.

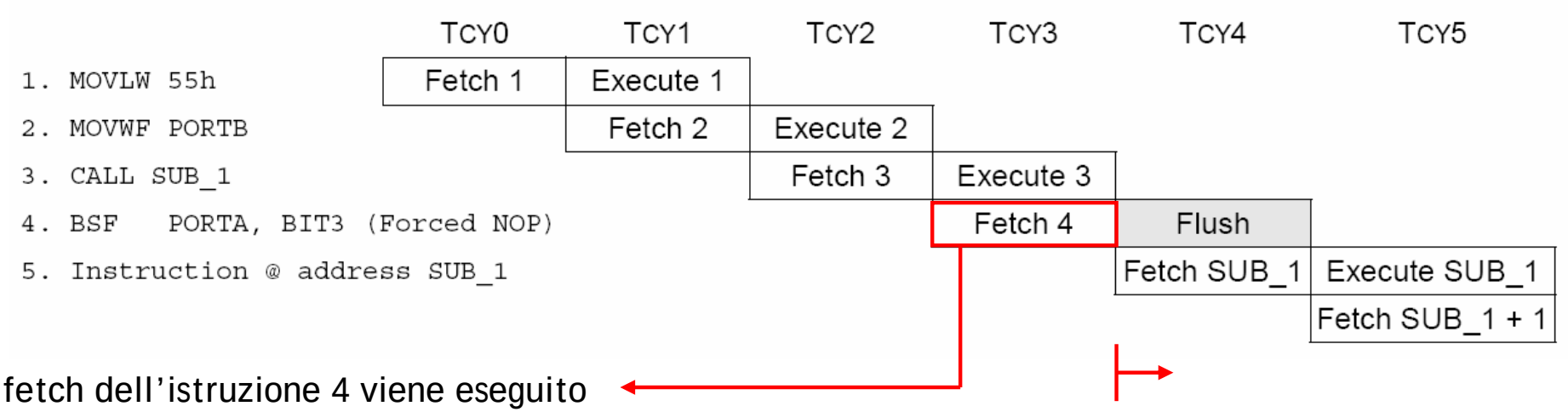
Si tratta di una struttura di tipo PIPELINE a due livelli che permette di ottenere un **RADDOPPIO DEL RATE DI ISTRUZIONI/s DEL MICRO**




TCY è un intervallo di tempo della durata di 4 periodi di clock

**PROBLEMA** la pipeline non funziona nel caso in cui il risultato dell'istruzione deve essere quello di modificare il flusso di esecuzione del programma, ovvero quando l'istruzione modifica il **Program Counter**

Questo è infatti un caso in cui l'istruzione decide quale sarà la prossima istruzione da eseguire e di conseguenza è necessario attendere la completa esecuzione del comando prima di fare nuovamente il fetch (se non si attende la fine della fase di execute del comando non si conosce l'indirizzo dell'istruzione da eseguire!!)




Il fetch dell'istruzione 4 viene eseguito durante l'esecuzione dell'istruzione 3 che è una *Call*. Il risultato del fetch è però inutile dato che la *Call* carica nel PC l'indirizzo di una diversa successiva istruzione. Per questo motivo nel successivo TCY non c'è nessuna istruzione da eseguire ma si deve eseguire il fetch dell'istruzione indicata dalla *Call*

L'esecuzione del firmware prosegue normalmente

Le istruzioni in questa situazione e che quindi durano 2 $T_{CY}$ sono:

| | | | |
|---|---|---|---|
| CALL<br>GOTO | Classiche istruzioni di salto o di chiamata di una routine | RETFIE<br>RETLW<br>RETURN | Sono instruzioni che si utilizzano per tornare nel codice principale da una sub-routine generica o da quella di gestione di interrupt |

Ci sono invece alcune istruzioni la cui durata dipende dall'esito di un test:

| | | | |
|---|---|---|---|
| DECFSZ<br>INCFSZ | Incrementano o decrementano un registro. Viene saltata la successiva istruzione se il risultato dell'operazione è 0 | BTFSC<br>BTFSS | Controllano lo stato di un bit di un registro. Viene saltata la successiva istruzione a seconda del valore del bit letto. |

# IL FIRMWARE COMPLETO

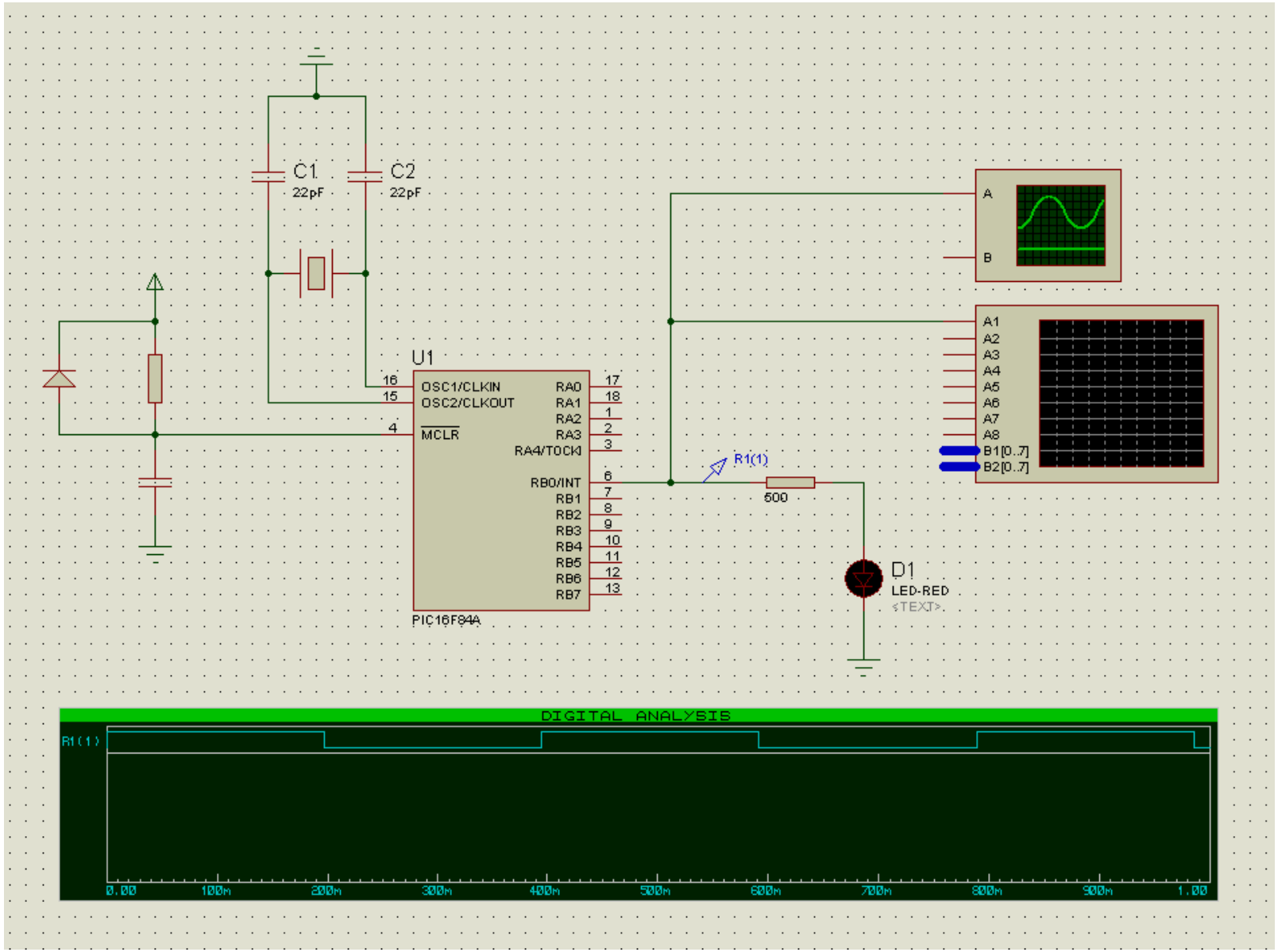
```
        PROCESSOR   16F84A
        RADIX       DEC

        INCLUDE     "P16F84A.INC"

LED     EQU         0

        ORG         0CH
Count   RES 2
```

```
        ORG         00H
        bsf         STATUS,RP0

        movlw       00011111B
        movwf       TRISA

        movlw       11111110B
        movwf       TRISB

        bcf         STATUS,RP0

        bsf         PORTB,LED
```

```
Main
        call        Delay
        btfsc       PORTB,LED
        goto        SetToZero

        bsf         PORTB,LED
        goto        Main
SetToZero
        bcf         PORTB,LED
        goto        Main
```

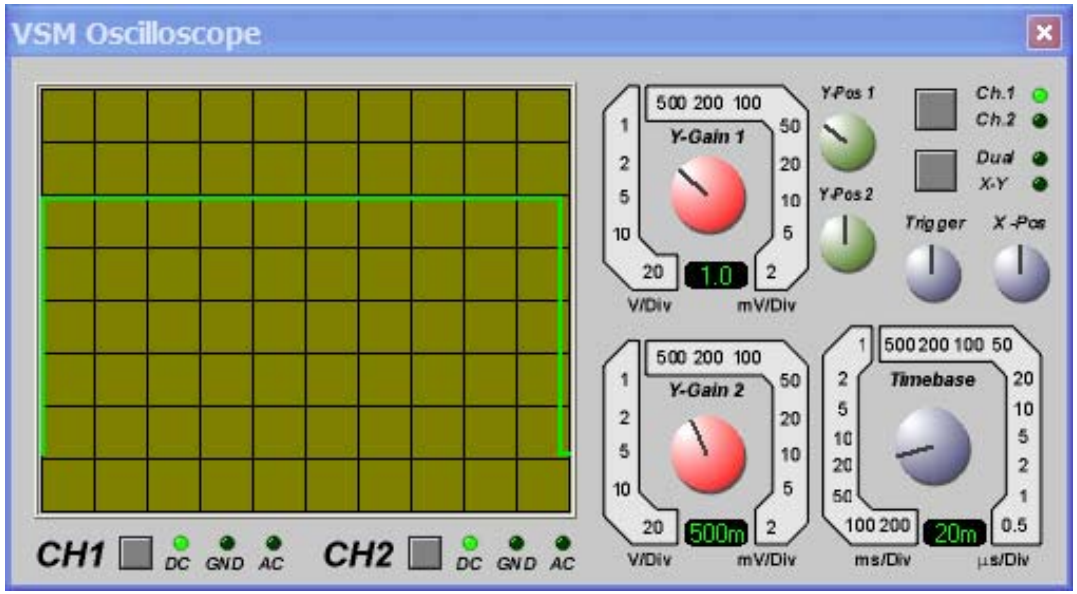
```
Delay
        clrf        Count
        clrf        Count+1
DelayLoop
        decfsz      Count,1
        goto        DelayLoop
        decfsz      Count+1,1
        goto        DelayLoop
        return

        END
```

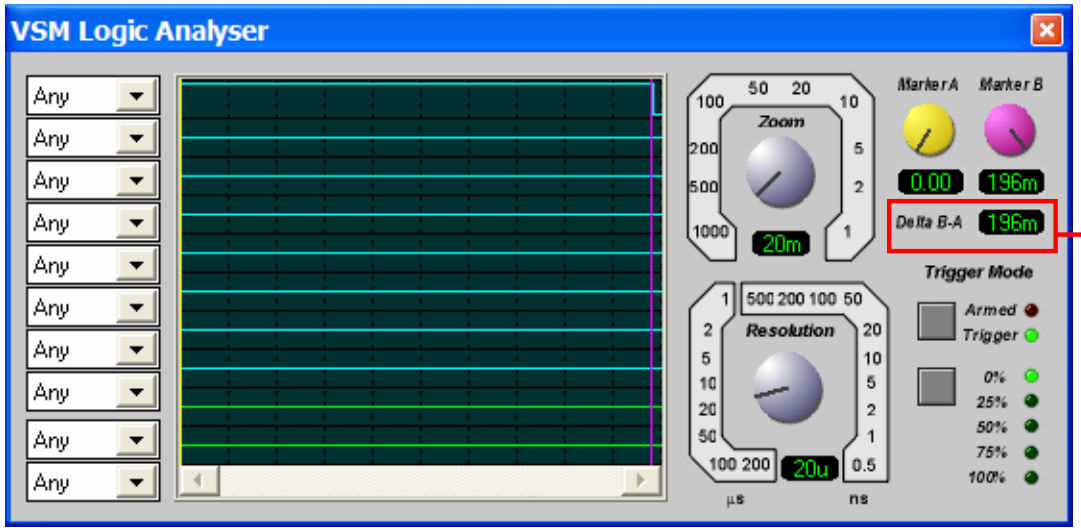
# SIMULAZIONE DEL CIRCUITO




Simulazione del circuito tramite il programma Proteus

# RISULTATI DELLA SIMULAZIONE




Entrambi gli strumenti hanno un base dei tempi impostata su

20 ms/div

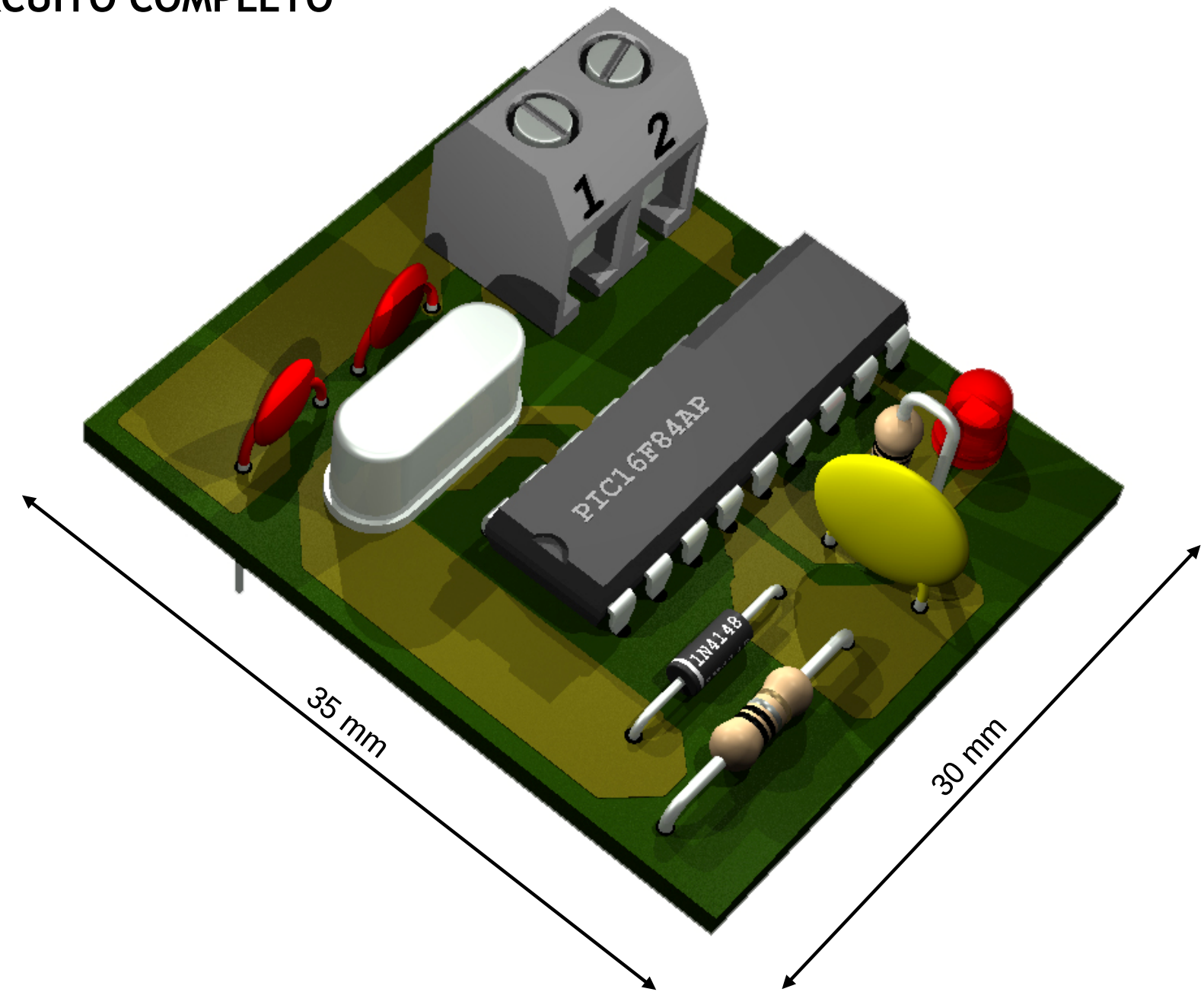
Questo significa che un segnale della durata di 200 ms (poco più del nostro caso) occuperebbe 10 divisioni, cioè l'intero schermo

L'analizzatore logico permette di misurare gli intervalli di tempo tramite l'utilizzo di apposite guide (in figura in viola)

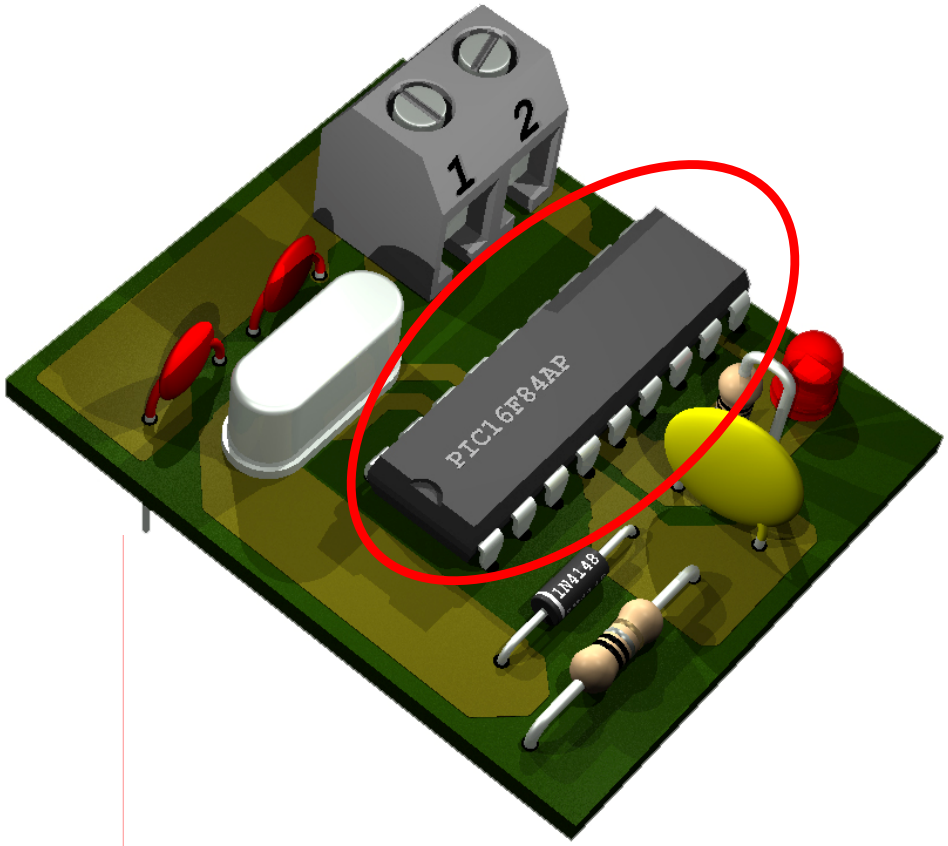
Il risultato della misura viene riportato dallo strumento attraverso appositi display:

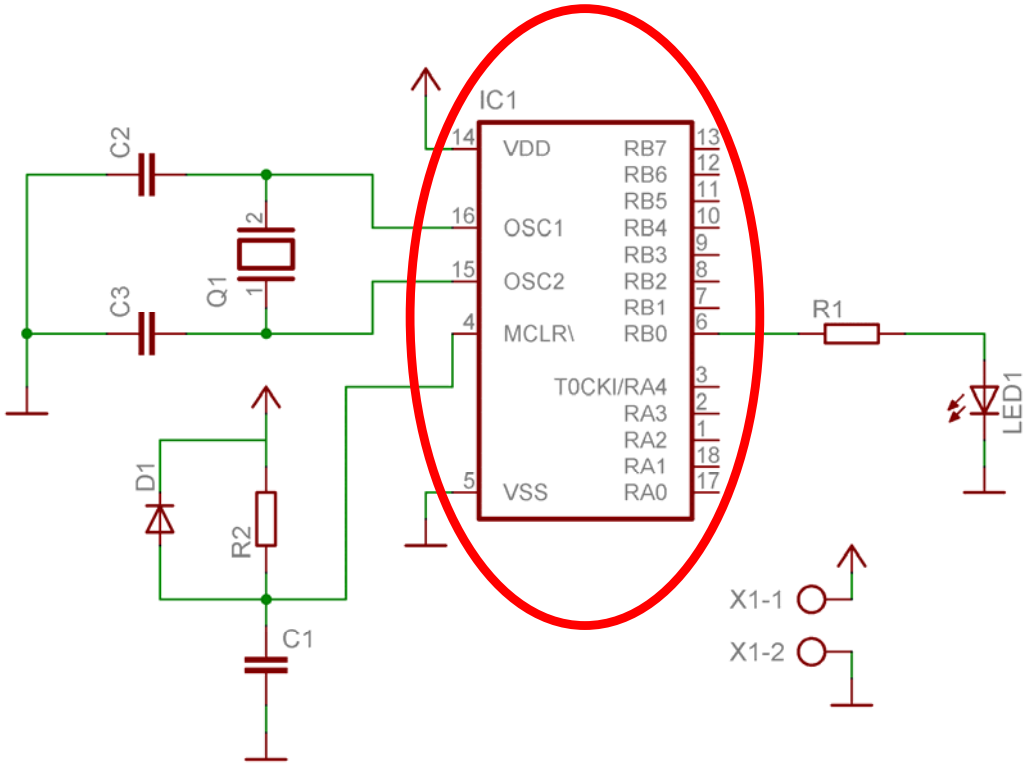
**196 ms!! LA SIMULAZIONE CONFERMA IL CALCOLO ANALITICO**

# IL CIRCUITO COMPLETO



# IL CIRCUITO COMPLETO

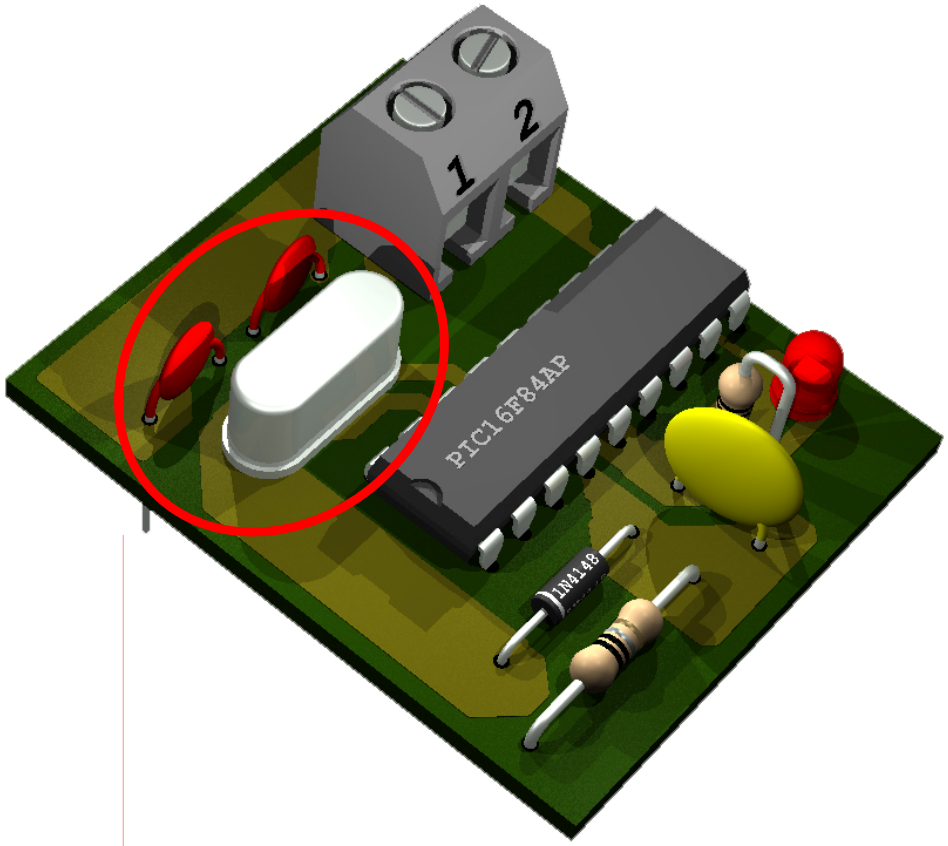


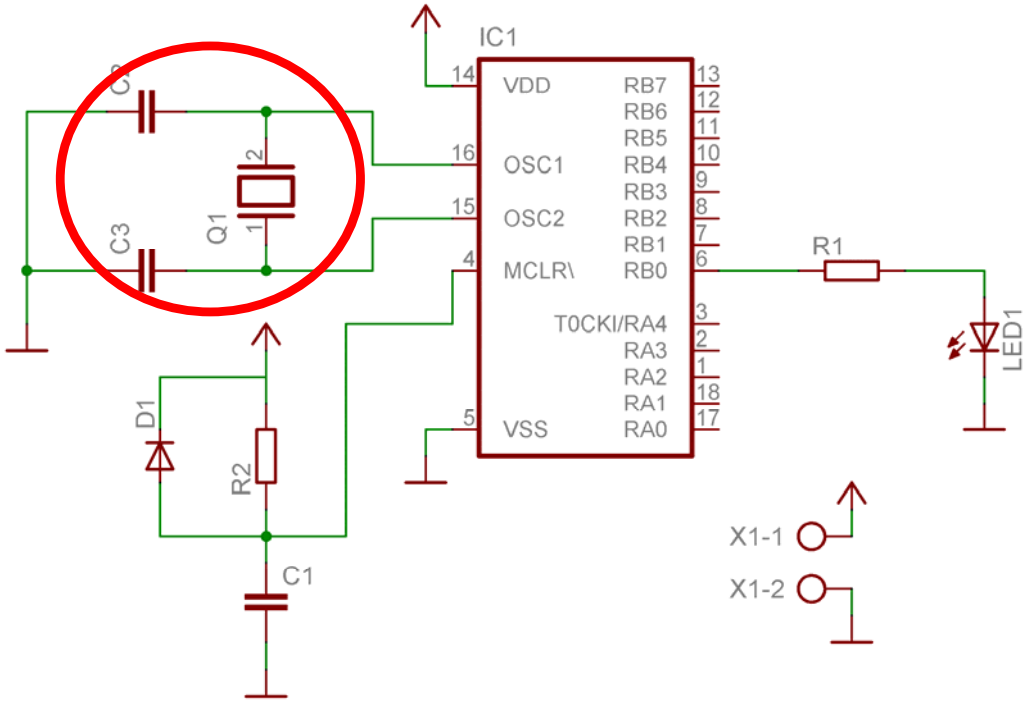




IL MICROCONTROLLORE

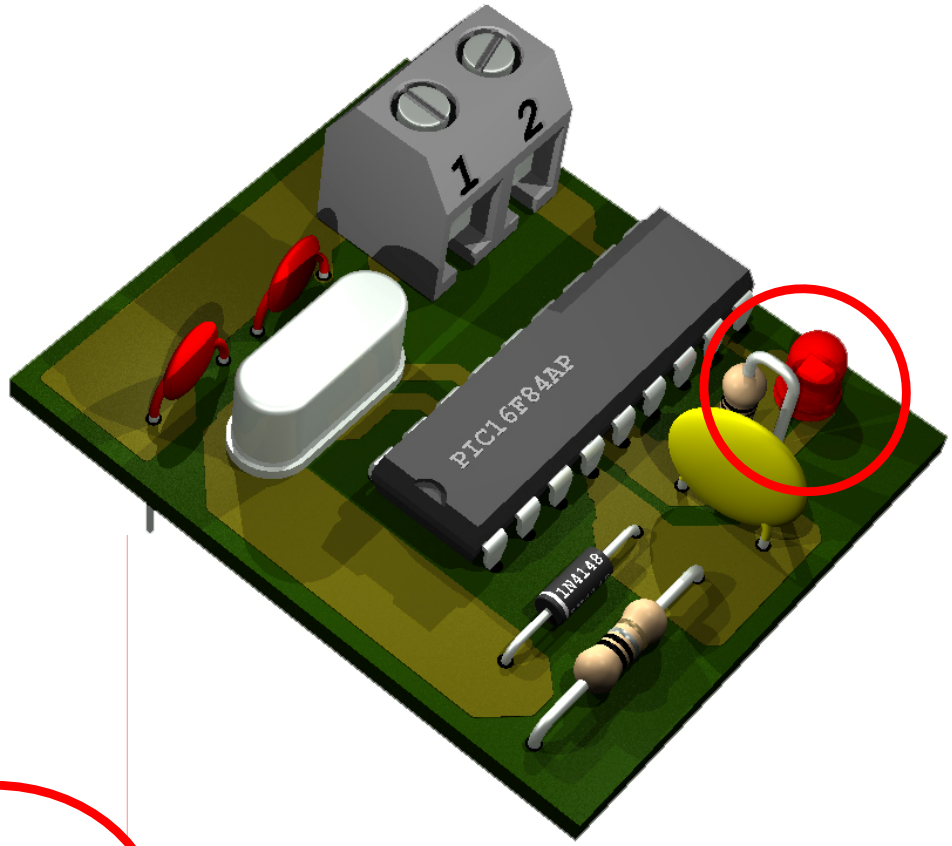
# IL CIRCUITO COMPLETO

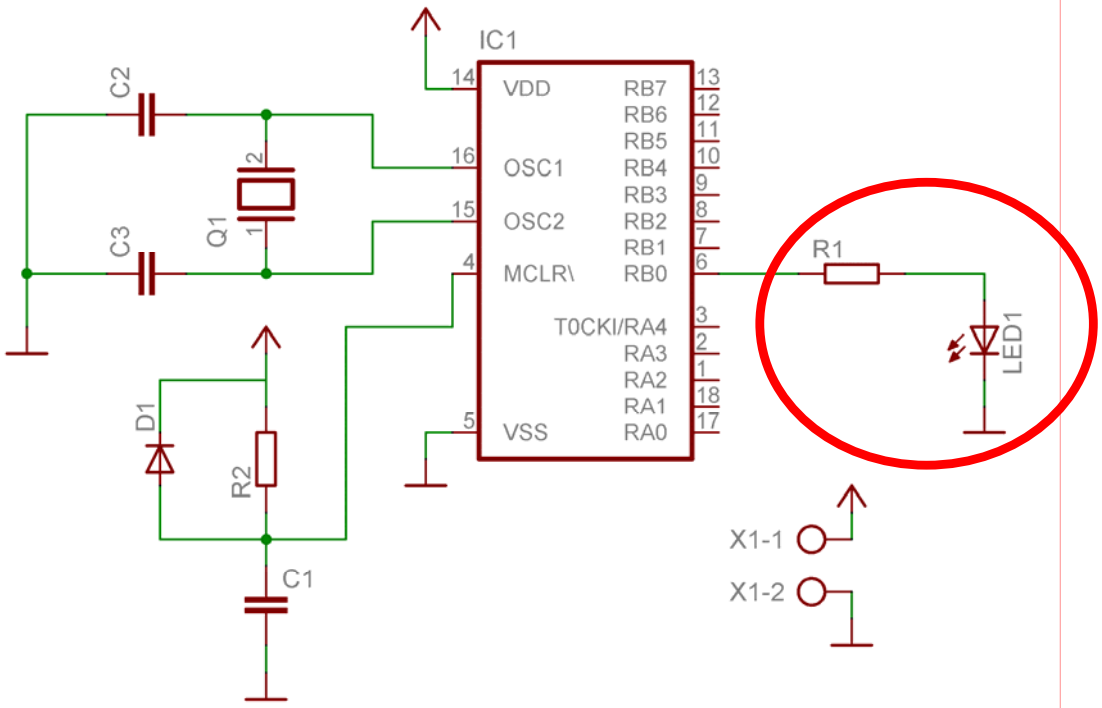
IL QUARZO E I CONDENSATORI PER L'OSCILLATORE INTERNO



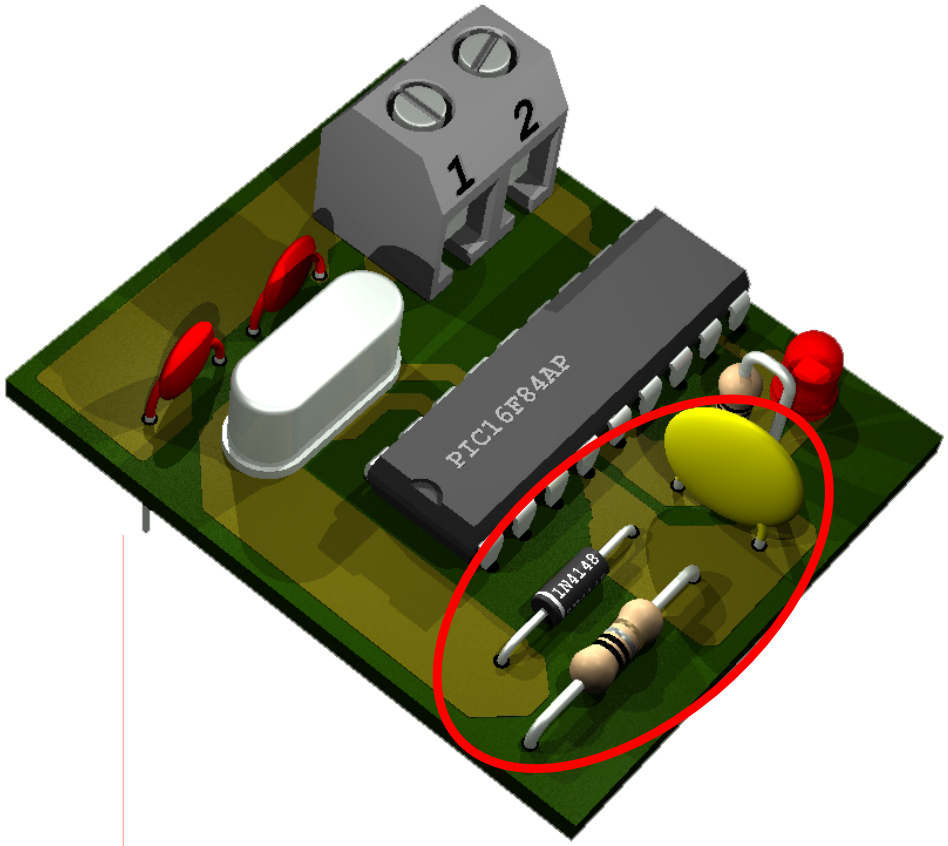
# IL CIRCUITO COMPLETO

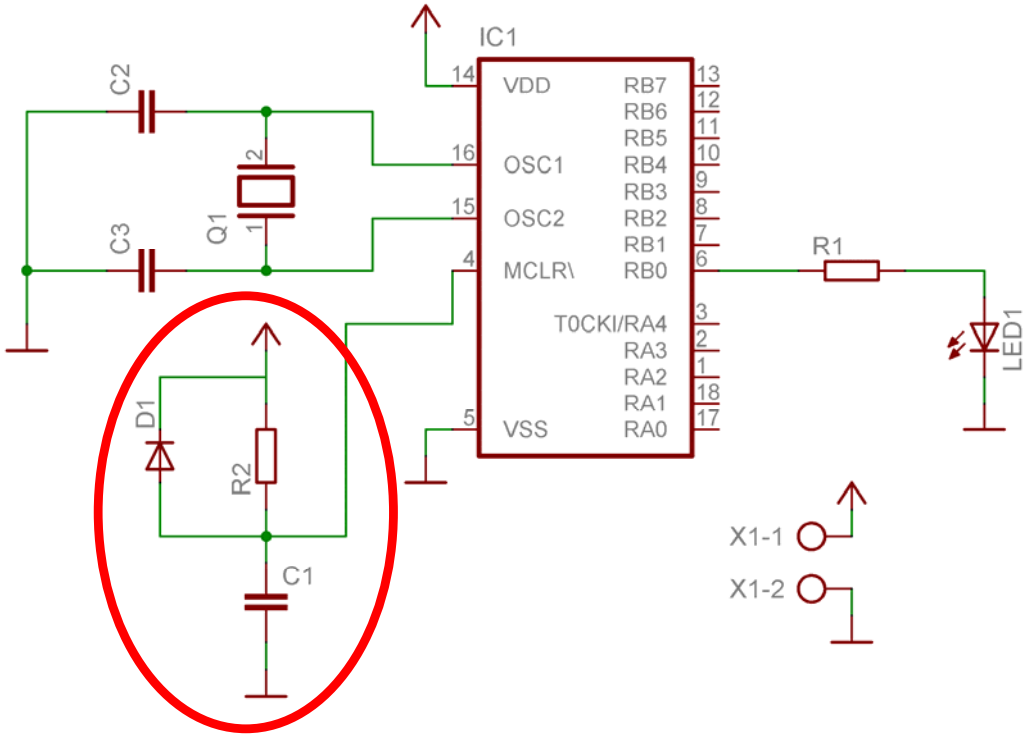



IL LED E LA RESISTENZA DI LIMITAZIONE DELLA CORRENTE

# IL CIRCUITO COMPLETO

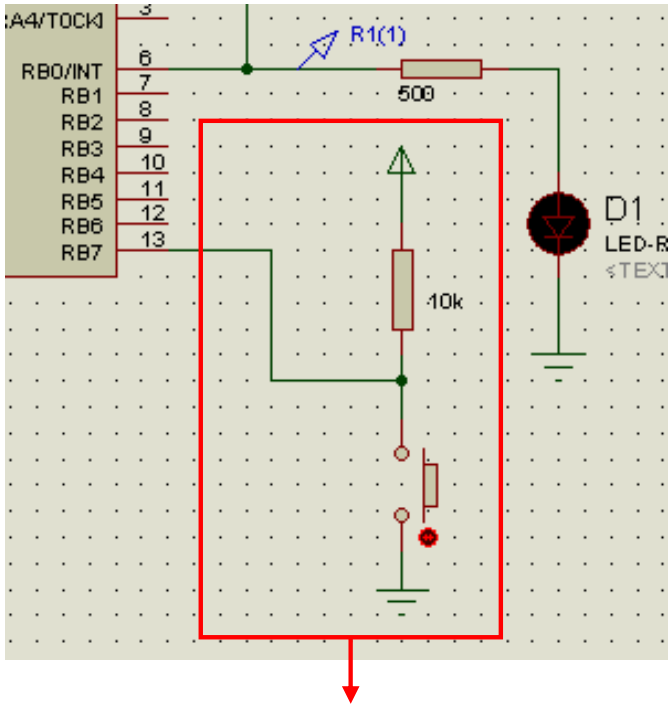



EXTERNAL
POWER-ON
RESET CIRCUIT

# MODIFICA 1 – AGGIUNTA DI UN PULSANTE

Ci proponiamo di modificare leggermente e il firmware e lo schema elettrico aggiungendo un pulsante la cui pressione determini l'inizio del lampeggiamento del LED.

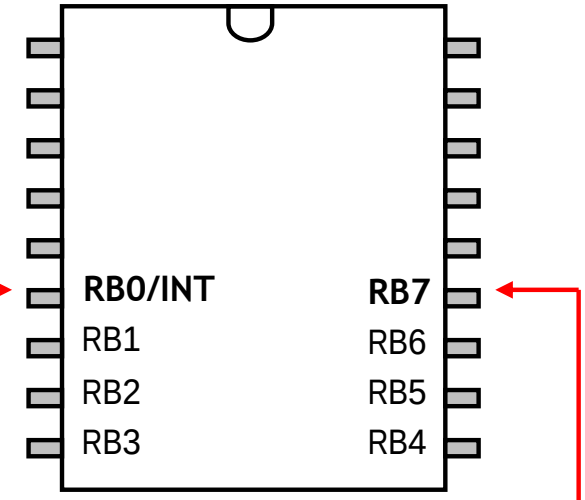



La modifica al circuito consiste semplicemente nel collegamento di un pulsante e resistenza di pull-up al pin RB7

## E IL FIRMWARE?

Naturalmente per poter accorgersi della pressione del pulsante il pin RB7 deve essere configurato come un ingresso. Nel nostro caso questa impostazione era già stata fatta all'inizio del firmware

```
ORG     00H
bsf     STATUS,RP0

movlw   00011111B
movwf   TRISA

movlw   11111110B

movwf   TRISB
```




**1=INPUT 0=OUTPUT**

# MODIFICA 1 – AGGIUNTA DI UN PULSANTE

```
            PROCESSOR       16F84A
            RADIX       DEC

            INCLUDE     "P16F84A.INC"

LED         EQU         0
SWITCH      EQU         7

            ORG         0CH
Count       RES 2

            btfsc       PORTB,SWITCH
            goto        $-1
Main
            call        Delay
            btfsc       PORTB,LED
            goto        SetToZero
```

(Continua…)

Definiamo una costante simbolica per semplificare la programmazione, da ora in poi potremo riferirci al pin della porta B cui è collegato l'interruttore senza dover ricordare quale pin precisamente fosse, basterà scrivere PORTB,SWITCH

Controlliamo lo stato del pin che inizialmente sarà alto per via del pull-up. Quando il pulsante verrà premuto il suo stato andrà basso e verrà saltato il successivo *goto* per proseguire nel resto del programma

Questa istruzione manda il programma all'istruzione che si trova all'indirizzo attuale del program counter (cioè l'attuale posizione) -1. E' un modo comodo per saltare indietro di una istruzione senza dover inserire un'etichetta e appesantire il codice

# MODIFICA 1 – AGGIUNTA DI UN PULSANTE

```
            PROCESSOR       16F84A
            RADIX       DEC

            INCLUDE     "P16F84A.INC"

LED         EQU         0
SWITCH      EQU         7

            ORG         0CH
Count       RES 2

            btfsc       PORTB,SWITCH
            goto        $-1
Main
            call        Delay
            btfsc       PORTB,LED
            goto        SetToZero
```

(Continua…)

Definiamo una costante simbolica per semplificare la programmazione, da ora in poi potremo riferirci al pin della porta B cui è collegato l'interruttore senza dover ricordare quale pin precisamente fosse, basterà scrivere PORTB,SWITCH

Controlliamo lo stato del pin che inizialmente sarà alto per via del pull-up. Quando il pulsante verrà premuto il suo stato andrà basso e verrà saltato il successivo *goto* per proseguire nel resto del programma

Questa istruzione manda il programma all'istruzione che si trova all'indirizzo attuale del program counter (cioè l'attuale posizione) -1. E' un modo comodo per saltare indietro di una istruzione senza dover inserire un'etichetta e appesantire il codice

# MODIFICA 1 – AGGIUNTA DI UN PULSANTE

```
            PROCESSOR       16F84A
            RADIX       DEC

            INCLUDE     "P16F84A.INC"

LED         EQU         0
SWITCH      EQU         7

            ORG         0CH
Count       RES 2

            btfsc       PORTB,SWITCH
            goto        $-1
Main
            call        Delay
            btfsc       PORTB,LED
            goto        SetToZero
```

(Continua…)

Definiamo una costante simbolica per semplificare la programmazione, da ora in poi potremo riferirci al pin della porta B cui è collegato l'interruttore senza dover ricordare quale pin precisamente fosse, basterà scrivere PORTB,SWITCH

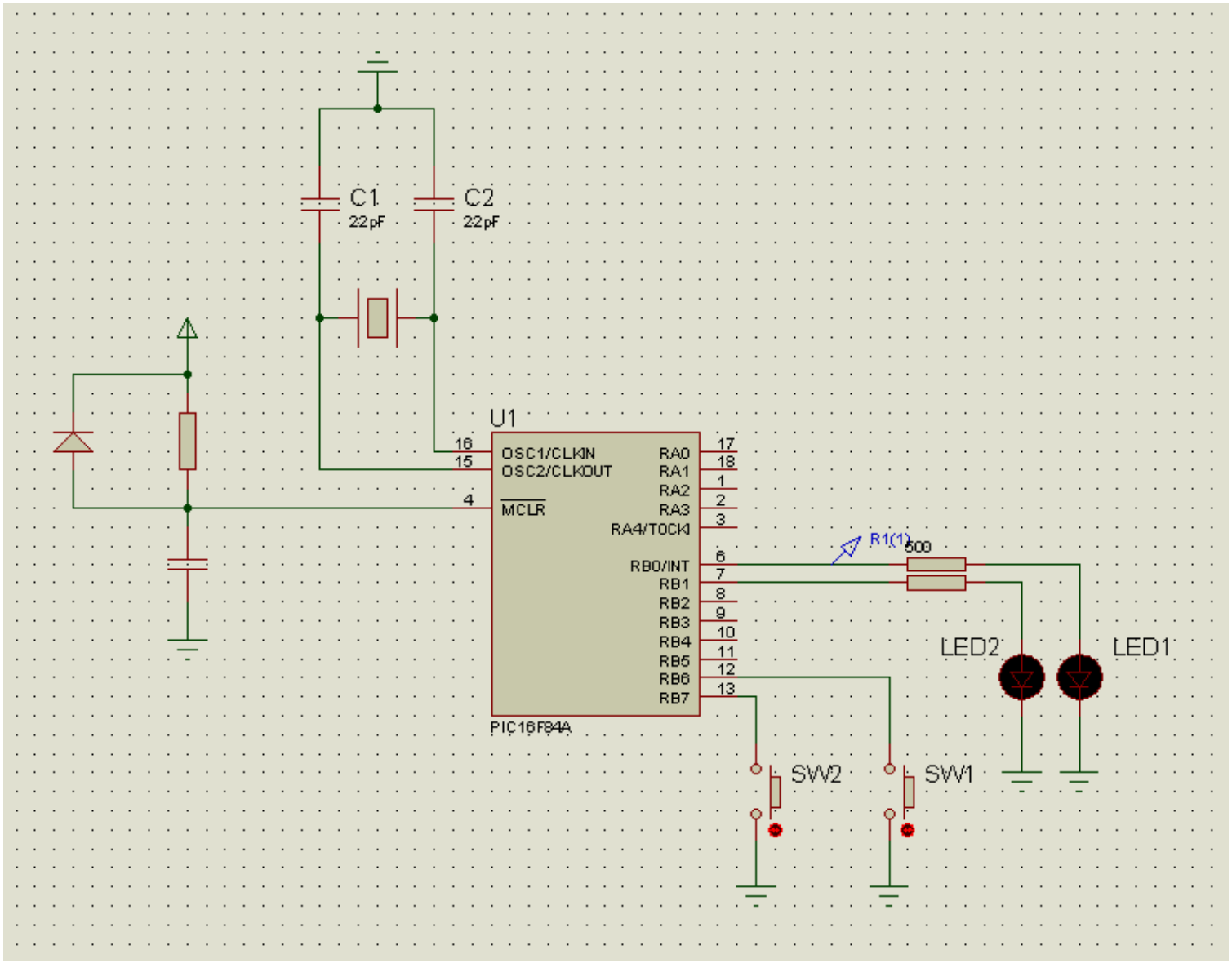
Controlliamo lo stato del pin che inizialmente sarà alto per via del pull-up. Quando il pulsante verrà premuto il suo stato andrà basso e verrà saltato il successivo *goto* per proseguire nel resto del programma

Questa istruzione manda il programma all'istruzione che si trova all'indirizzo attuale del program counter (cioè l'attuale posizione) -1. E' un modo comodo per saltare indietro di una istruzione senza dover inserire un'etichetta e appesantire il codice

# MODIFICA 1 - IL FIRMWARE COMPLETO

```
            PROCESSOR   16F84A
            RADIX       DEC

            INCLUDE     "P16F84A.INC"

LED         EQU         0
SWITCH      EQU         7

            ORG         0CH
Count       RES 2


            ORG         00H
            bsf         STATUS,RP0

            movlw       00011111B
            movwf       TRISA

            movlw       11111110B
            movwf       TRISB

            bcf         STATUS,RP0

            bsf         PORTB,LED

            btfsc       PORTB,SWITCH
            goto        $-1
Main
            call        Delay
            btfsc       PORTB,LED
            goto        SetToZero

            bsf         PORTB,LED
            goto        Main
SetToZero
            bcf         PORTB,LED
            goto        Main

Delay
            clrf        Count
            clrf        Count+1
DelayLoop
            decfsz      Count,1
            goto        DelayLoop
            decfsz      Count+1,1
            goto        DelayLoop
            return

            END
```

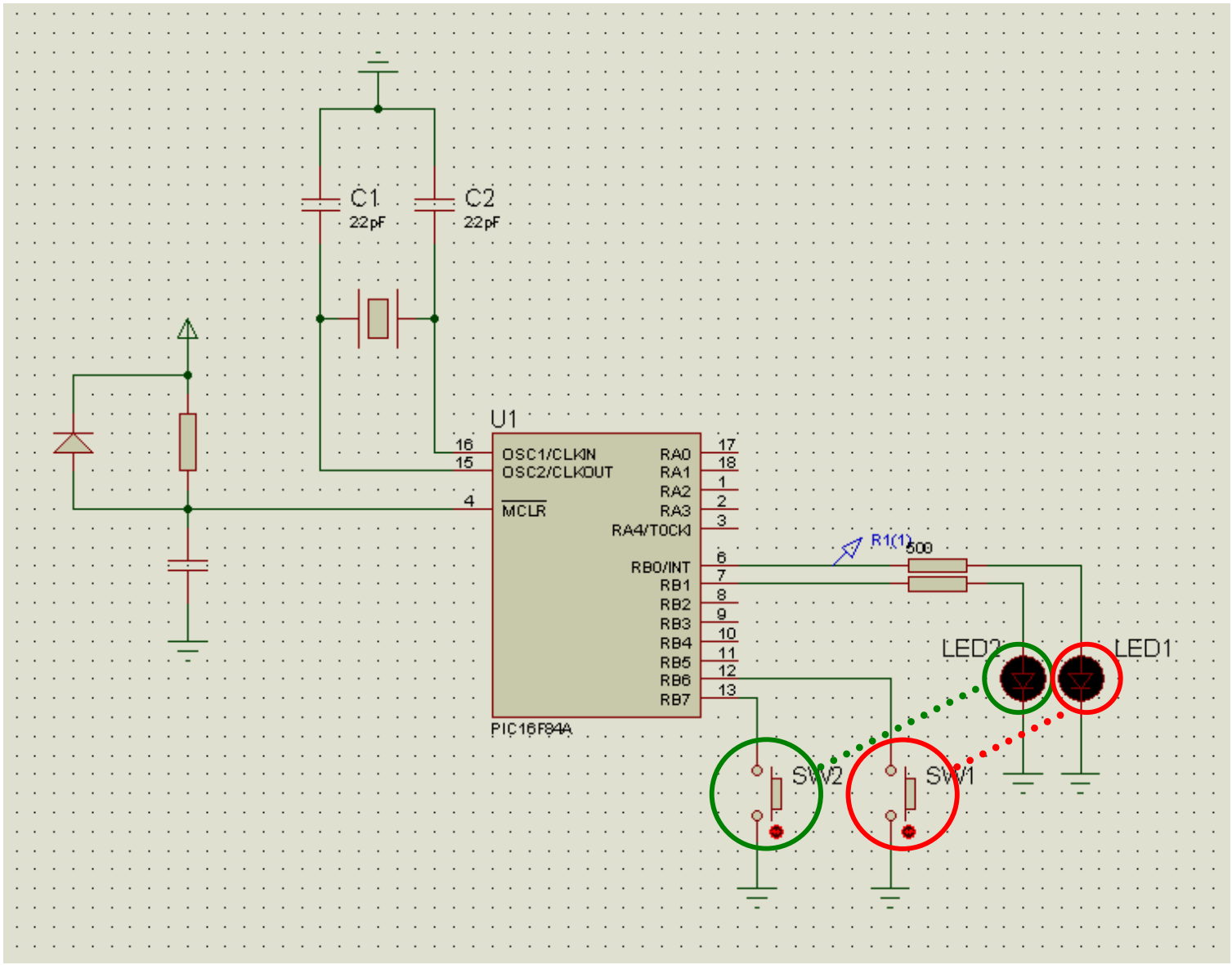
# MODIFICA 2 – UTILIZZO DI INTERRUPT




L'intento di questa seconda modifica è quello di gestire due diodi LED tramite due pulsanti.

Ognuno dei due pulsanti sarà strettamente correlato ad uno dei due LED e la sua funzione sarà quella di avviare o fermare il lampeggiamento del LED stesso; il tutto in maniera indipendente dall'altro LED.

# MODIFICA 2 – UTILIZZO DI INTERRUPT




L'intento di questa seconda modifica è quello di gestire due diodi LED tramite due pulsanti.

Ognuno dei due pulsanti sarà strettamente correlato ad uno dei due LED e la sua funzione sarà quella di avviare o fermare il lampeggiamento del LED stesso; il tutto in maniera indipendente dall'altro LED.

# MODIFICA 2 – UTILIZZO DI INTERRUPT

L'intento di questa seconda modifica è quello di gestire due diodi LED tramite due pulsanti. Ognuno dei due pulsanti sarà strettamente correlato ad uno dei due LED e la sua funzione sarà quella di avviare o fermare il lampeggiamento del LED stesso; il tutto in maniera indipendente dall'altro LED.

## COME ORGANIZZARE IL NUOVO FIRMWARE?

Il problema riguarda soprattutto la ruotine principale *Main* nella quale eseguiamo l'operazione di lampeggiamento dei LED. Nell'esempio iniziale, una volta acceso il circuito, non vi era nessun particolare evento che potesse disturbare l'esecuzione del lampeggiamento.

## ORA LA SITUAZIONE E' MOLTO DIVERSA IN QUANTO E' NECESSARIO:

1 – Verificare per ognuno dei due LED se esso debba o meno lampeggiare

2 – Pilotare correttamente i LED per il lampeggiamento

3 – Gestire gli eventi di pressione dei pulsanti di controllo che abilitano/fermano il lampeggiamento

**TUTTO QUANTO NELLA MANIERA PIU' TRASPARENTE POSSIBILE, SENZA CIOE' CHE IL LAMPEGGIAMENTO DEL LED NON COINVOLTO DALLA PRESSIONE DEL PULSANTE SI MODIFICHI SENSIBILMENTE!!!**

# MODIFICA 2 – UTILIZZO DI INTERRUPT

Analisi di alcuni possibili approcci al problema. Routine *Main*


## PROBLEMA 1

Queste istruzioni vengono eseguite ad ogni loop del *Main*, quindi influiscono SEMPRE sul lampeggiamento (il cui timing era stato così duramente calcolato nel primo esempio).

Volendo potremmo anche tenerne conto…

# MODIFICA 2 – UTILIZZO DI INTERRUPT

Analisi di alcuni possibili approcci al problema. Routine *Main*


**PROBLEMA 2**

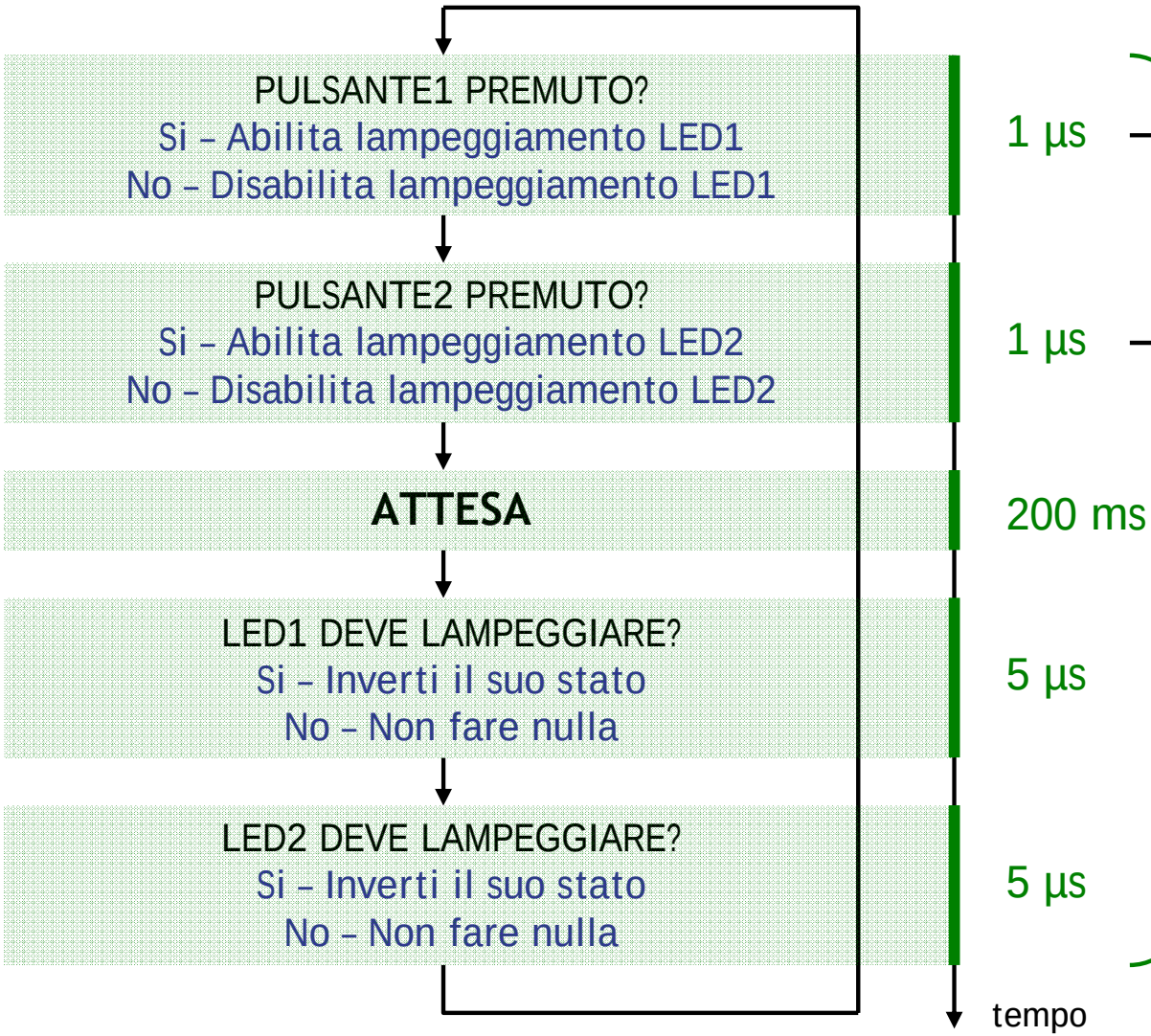

Considerando i riflessi umani il pulsante viene tenuto premuto per circa 0,1 s.

L'attesa che abbiamo sintetizzato dura però 200ms (in genere non si scende molto al di sotto dato che vogliamo VEDERE il lampeggiamento).

Quindi se la pressione non si verifica nell'intervallo di tempo in cui controlliamo essa verrà persa!!!

# MODIFICA 2 – UTILIZZO DI INTERRUPT

Analisi di alcuni possibili approcci al problema. Routine *Main*



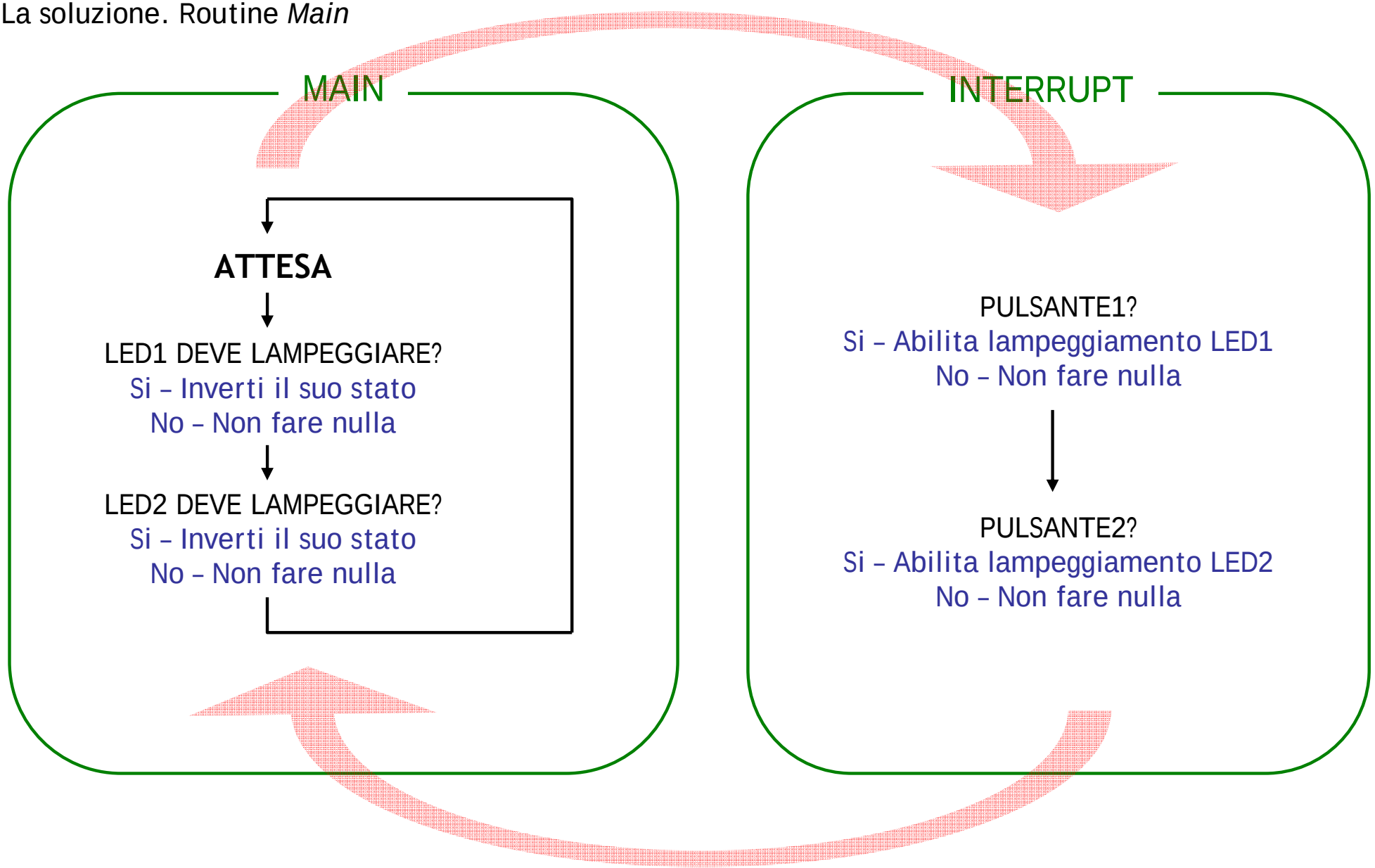

Se tutta la durata del ciclo il pulsante viene letto solo per:

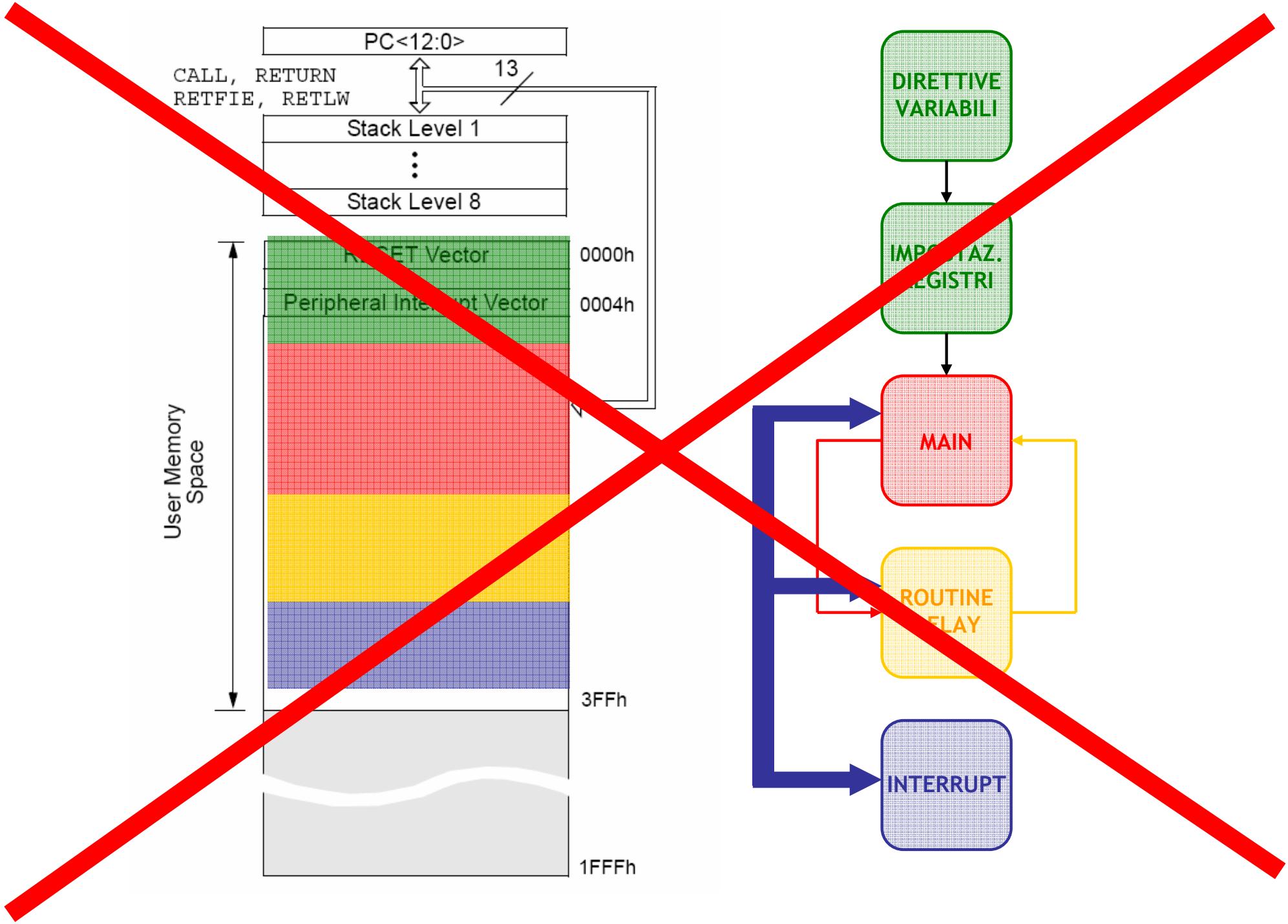
$$\frac{1\ \mu s}{200.012\ \mu s} = 0{,}0005\ \%$$

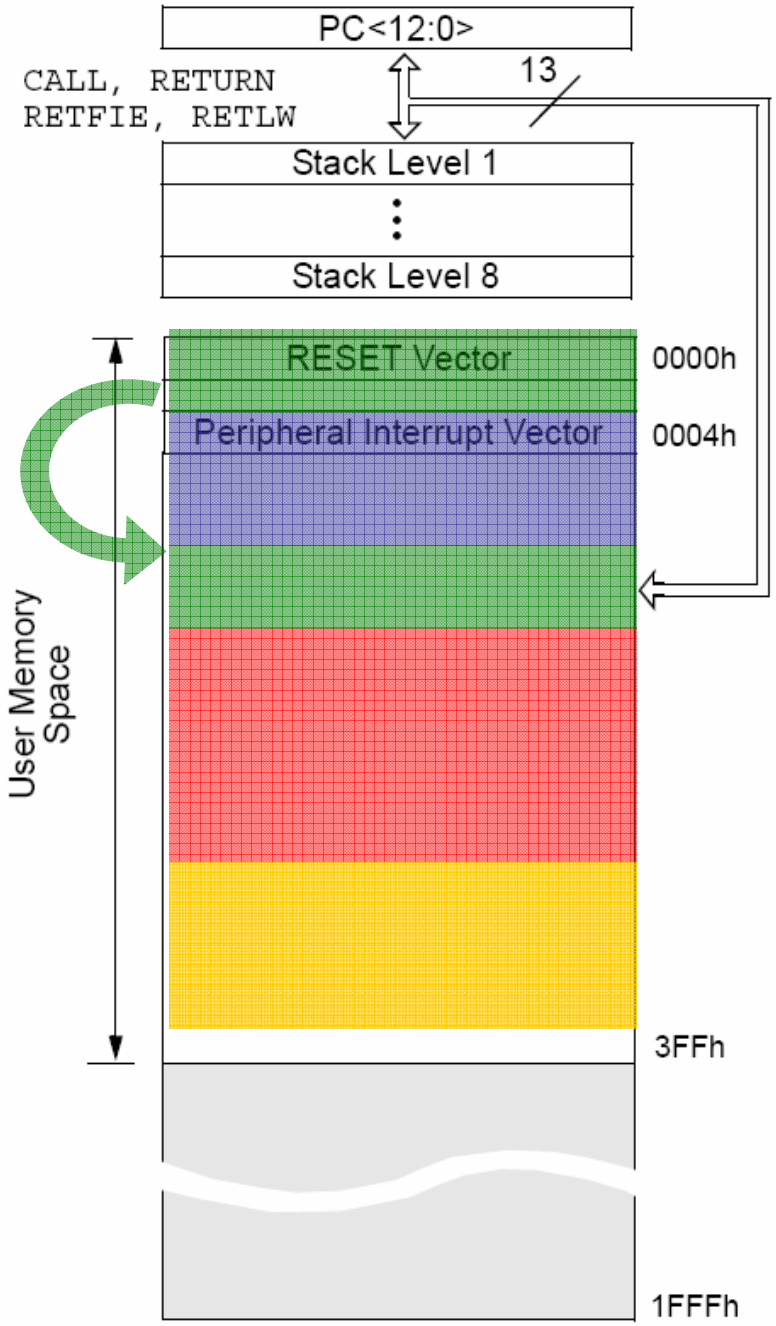
Basterebbe tenere premuto il pulsante più a lungo per avere la certezza che il circuito se ne accorga… però non dovrebbe essere l'utilizzatore ad adattarsi alla macchina, ma il contrario!!!

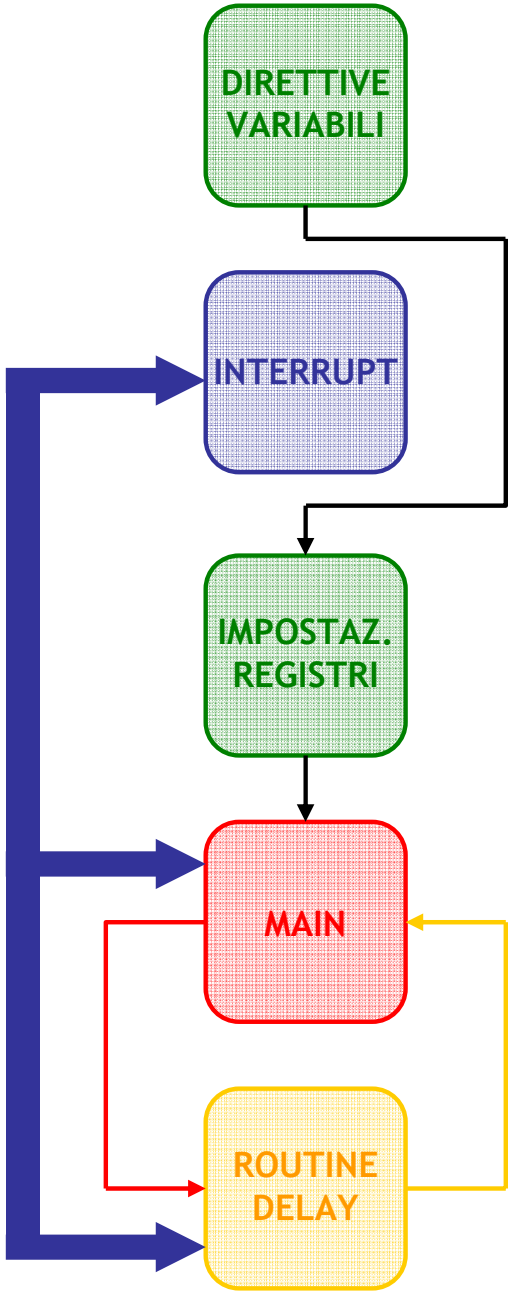
# MODIFICA 2 – UTILIZZO DI INTERRUPT

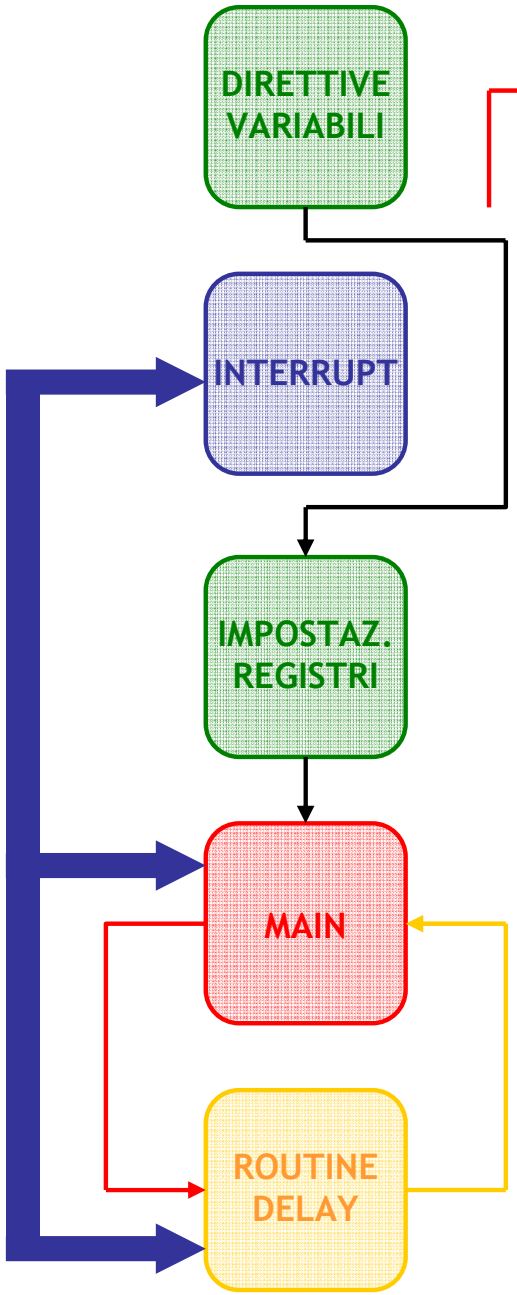
La soluzione. Routine *Main*




PC<12:0>
CALL, RETURN
RETFIE, RETLW
13
Stack Level 1
Stack Level 8
Peripheral Interrupt Vector
0000h
0004h
User Memory Space
3FFh
1FFFh
DIRETTIVE VARIABILI
MAIN
ROUTINE
INTERRUPT


PC<12:0>
CALL, RETURN
RETFIE, RETLW
13
Stack Level 1
Stack Level 8
RESET Vector
0000h
Peripheral Interrupt Vector
0004h
User Memory Space
3FFh
1FFFh

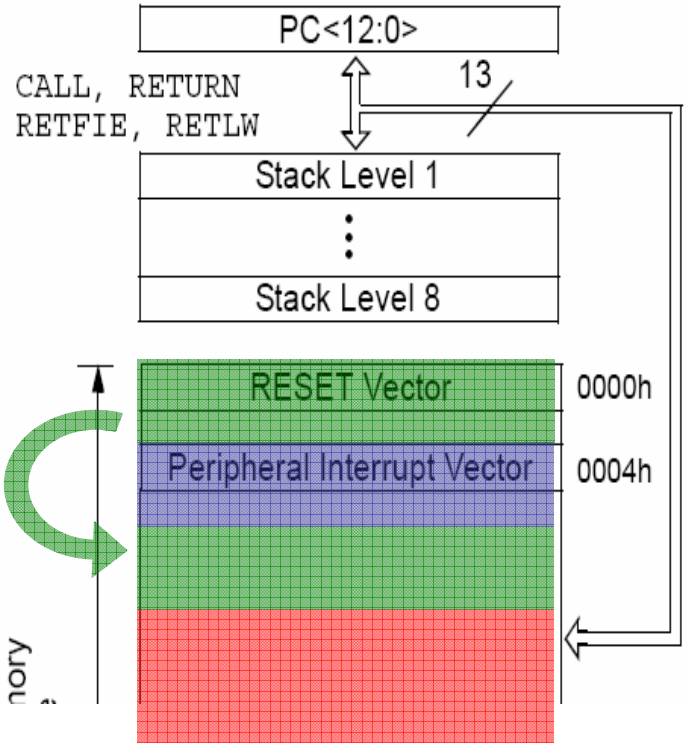

DIRETTIVE VARIABILI
INTERRUPT
IMPOSTAZ. REGISTRI
MAIN
ROUTINE DELAY

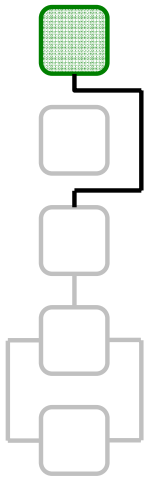



COME MAI IL FIRMWARE NON PUO' ESSERE ESEGUITO IN MANIERA PIU' SEQUENZIALE?

Quando si verifica un interrupt il micro cessa la normale esecuzione delle istruzioni e nel PC viene caricato l'indirizzo 0004h dell'*interrupt vector*

Dato che prima dell'interrupt vector c'è spazio solo per 3 istruzioni tipicamente si inserisce all'indirizzo 0000h un goto che punta all'inizio vero e proprio del programma.





```
        PROCESSOR       16F84A
        RADIX           DEC

        INCLUDE         "P16F84A.INC"

LED1    EQU     0
LED2    EQU     1
SWITCH1 EQU     6
SWITCH2 EQU     7

        ORG     0CH

Count   RES 2
QUALE   RES 1

        ORG     00H
        goto    inizio
```

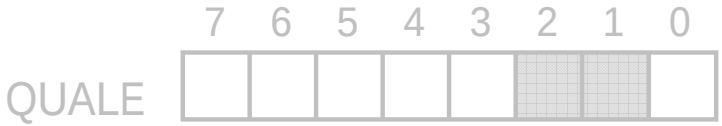
LED1 → RB0
LED2 → RB1

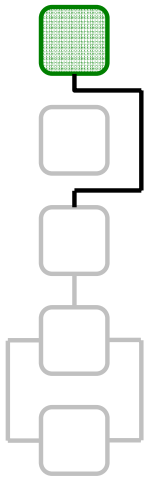
SWITCH1 → RB6
SWITCH2 → RB7

Definiamo la variabile QUALE di un byte che useremo per stabilire se i LED devono o meno lampeggiare.

Associamo il bit 1 al LED1 e il bit 2 al LED2. Il bit 1 viene modificato dallo SWITCH1 mentre il bit 2 dallo SWITCH2

Questa istruzione realizza quello che nello schema a blocchi è il salto in nero
Come mai è necessario?





```
            PROCESSOR       16F84A
            RADIX           DEC

            INCLUDE         "P16F84A.INC"

LED1        EQU     0
LED2        EQU     1
SWITCH1     EQU     6
SWITCH2     EQU     7

            ORG     0CH

Count       RES 2
QUALE       RES 1

            ORG     00H
            goto    inizio
```

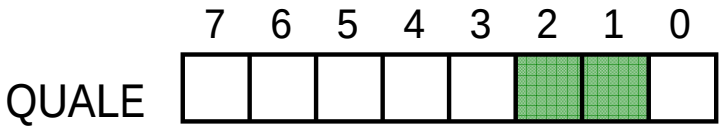
LED1 → RB0
LED2 → RB1

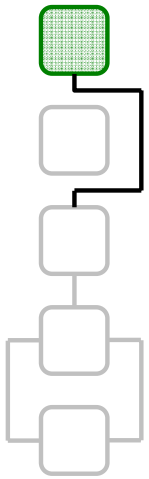
SWITCH1 → RB6
SWITCH2 → RB7

Definiamo la variabile QUALE di un byte che useremo per stabilire se i LED devono o meno lampeggiare.

Associamo il bit 1 al LED1 e il bit 2 al LED2. Il bit 1 viene modificato dallo SWITCH1 mentre il bit 2 dallo SWITCH2

Questa istruzione realizza quello che nello schema a blocchi è il salto in nero
Come mai è necessario?





```
            PROCESSOR       16F84A
            RADIX           DEC

            INCLUDE         "P16F84A.INC"

LED1        EQU     0
LED2        EQU     1
SWITCH1     EQU     6
SWITCH2     EQU     7

            ORG     0CH

Count       RES 2
QUALE       RES 1

            ORG     00H
            goto    inizio
```

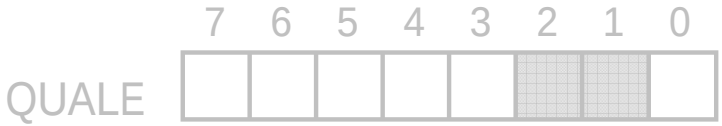
LED1 → RB0
LED2 → RB1

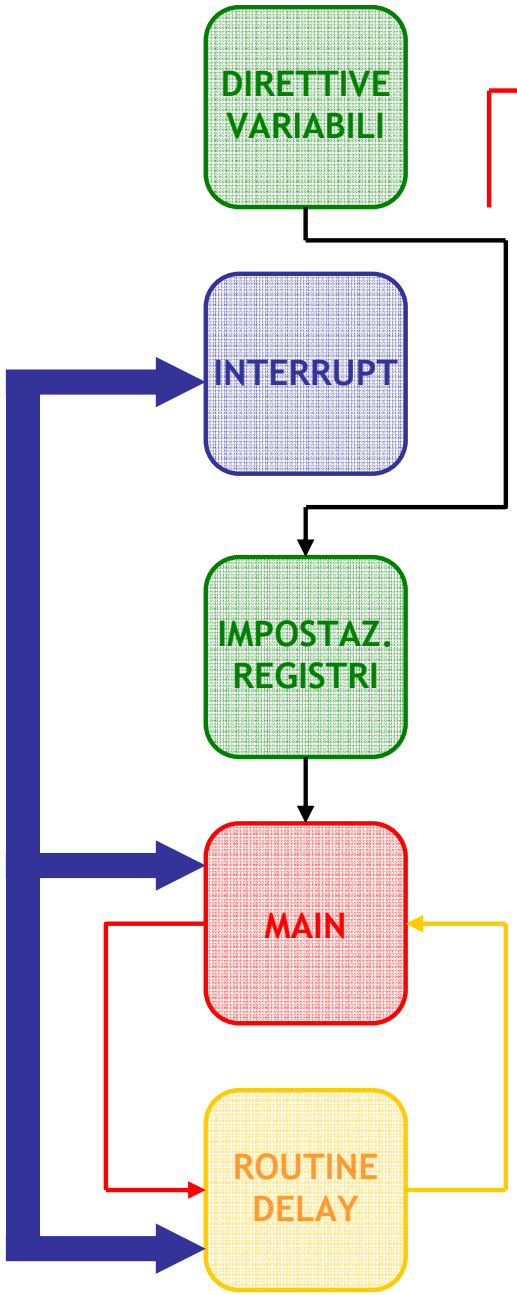
SWITCH1 → RB6
SWITCH2 → RB7

Definiamo la variabile QUALE di un byte che useremo per stabilire se i LED devono o meno lampeggiare.

Associamo il bit 1 al LED1 e il bit 2 al LED2. Il bit 1 viene modificato dallo SWITCH1 mentre il bit 2 dallo SWITCH2

Questa istruzione realizza quello che nello schema a blocchi è il salto in nero

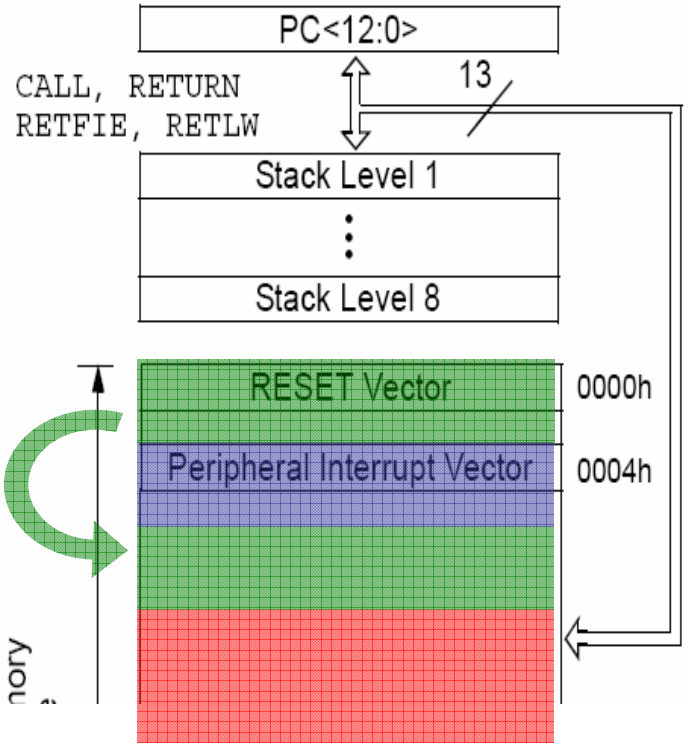


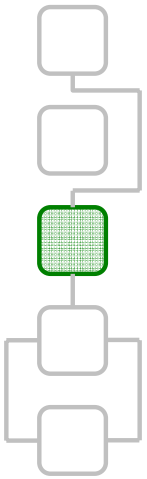



COME MAI IL FIRMWARE NON PUO' ESSERE ESEGUITO IN MANIERA PIU' SEQUENZIALE?

Quando si verifica un interrupt il micro cessa la normale esecuzione delle istruzioni e nel PC viene caricato l'indirizzo 0004h dell'*interrupt vector*

Dato che prima dell'interrupt vector c'è spazio solo per 3 istruzioni tipicamente si inserisce all'indirizzo 0000h un goto che punta all'inizio vero e proprio del programma.

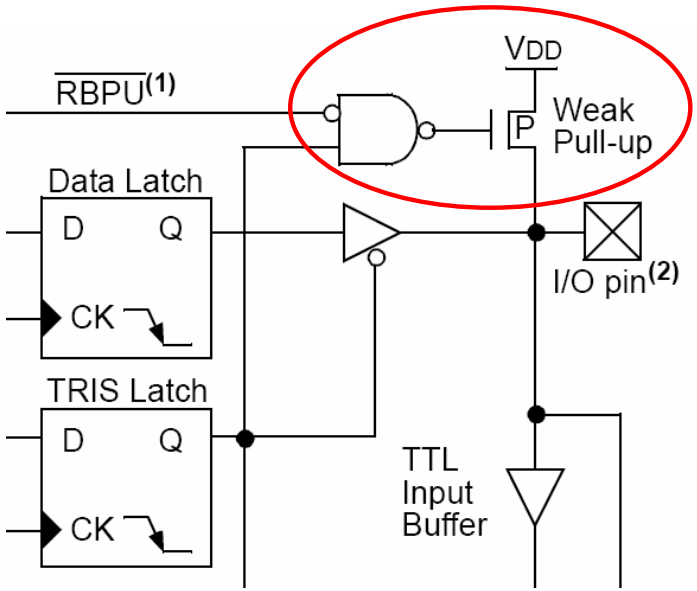




```
inizio
        bsf     STATUS,RP0

        movlw   00011111B
        movwf   TRISA

        movlw   11111100B
        movwf   TRISB

        movlw   01111111B
        andwf   OPTION_REG,F

        movlw   10000000B
        movwf   INTCON

        bcf     STATUS,RP0

        clrf    QUALE
```

→ Ora i LED sono 2, servono due output

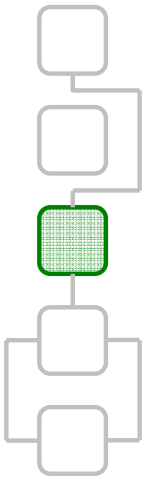
→ Imposta il registro OPTION_REG. Mettendo a zero il bit 7 imposta il pull-up resistivo interno sui pin della porta B (risparmiamo un resistore!!!)



# OPERAZIONI DI MASCHERAMENTO CON OPERATORI LOGICI: AND


A and B = OUT

| A | B | OUT |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |



```
inizio
        bsf     STATUS,RP0

        movlw   00011111B
        movwf   TRISA

        movlw   11111100B
        movwf   TRISB

        movlw   01111111B
        andwf   OPTION_REG,F

        movlw   10000000B
        movwf   INTCON

        bcf     STATUS,RP0

        clrf    QUALE
```

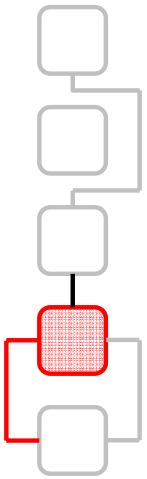
Imposta il registro INTCON.

Per il momento abilitiamo soltanto il Global Interrupt Enable (bit7) ma ancora non abilitiamo l'interrupt specifico (che si abilita ponendo a 1 il bit3; il bit0 è il relativo flag)

Azzero la variabile QUALE in modo che all'inizio i LED non lampeggino

INTCON

| R/W-0 | R/W-0 | R/W-0 | R/W-0 | R/W-0 | R/W-0 | R/W-0 | R/W-x |
|---|---|---|---|---|---|---|---|
| GIE | EEIE | T0IE | INTE | RBIE | T0IF | INTF | RBIF |
| bit 7 | | | | | | | bit 0 |



```
                call    Delay
                bcf     INTCON,0
                bsf     INTCON,3

Main

                call    Delay
                btfss   QUALE,1
                goto    led1_fermo
                movlw   00000001B
                xorwf   PORTB,F

led1_fermo

                btfss   QUALE,2
                goto    Main
                movlw   00000010B
                xorwf   PORTB,F

                goto    Main
```

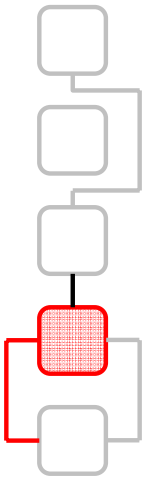
Chiama la routine *Delay* per dare il tempo alle impostazioni di divenire effettive (serve per evitare falsi interventi dell'interrupt alla partenza). Solo successivamente viene attivato l'interrupt e azzerato il relativo flag

Controlla il bit1 della variabile QUALE: se è 1 salta la successiva istruzione e inverte lo stato del LED1

Invece di introdurre un ulteriore test che stabilisca la successiva istruzione da eseguire (*bcf* per azzerare o *bsf* per asserire).

Invertiamo lo stato del bit direttamente sfruttando l'ALU interna tramite l'istruzione *xorwf* che effettua lo XOR tra il byte nel registro W e il contenuto di un registro, salvando il risultato nel registro stesso.

**A XOR 0 = A**
**A XOR 1 = !A**



```
            call    Delay
            bcf     INTCON,0
            bsf     INTCON,3

Main

            call    Delay
            btfss   QUALE,1
            goto    led1_fermo
            movlw   00000001B
            xorwf   PORTB,F

led1_fermo

            btfss   QUALE,2
            goto    Main
            movlw   00000010B
            xorwf   PORTB,F

            goto    Main
```

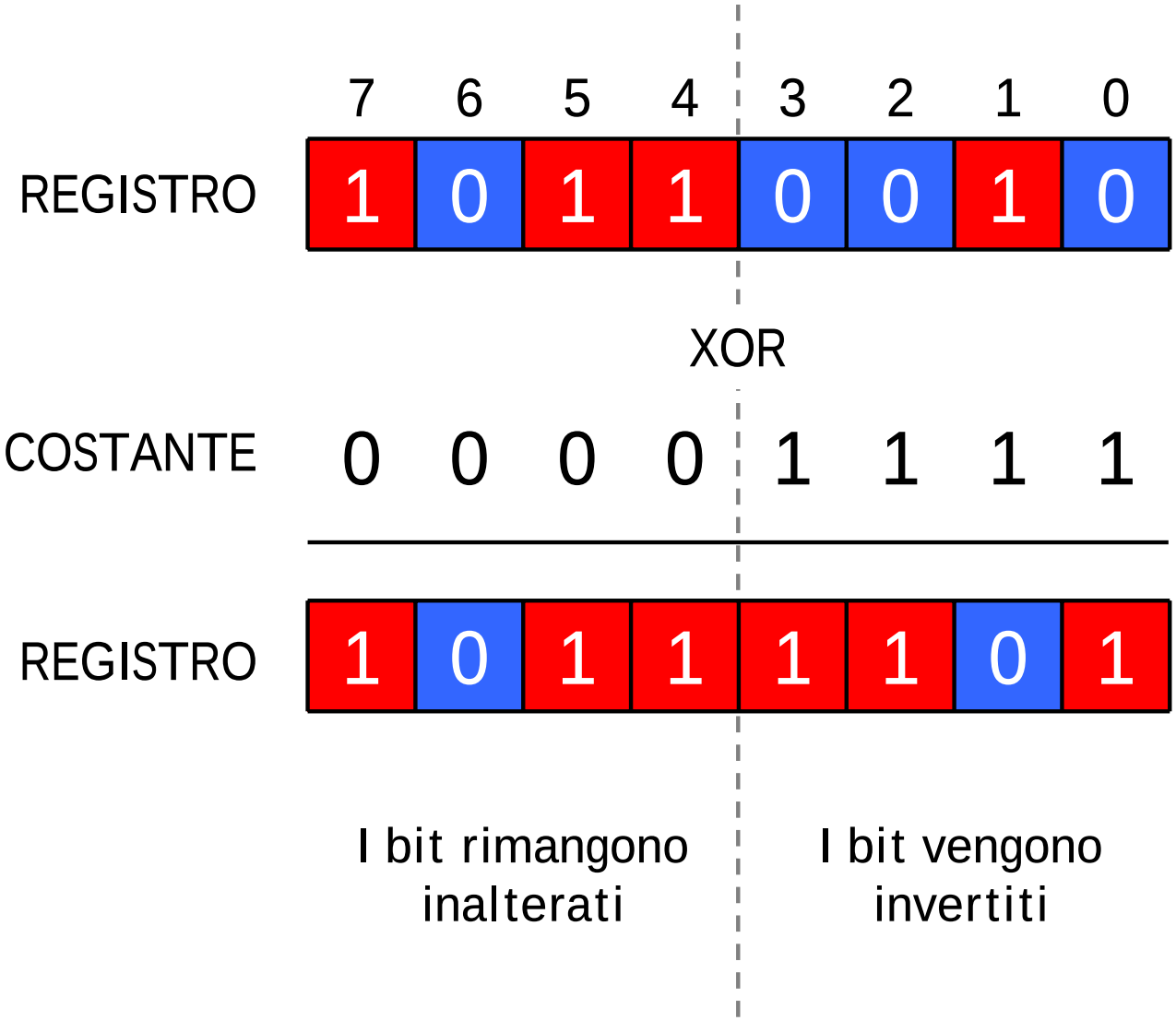
Chiama la routine *Delay* per dare il tempo alle impostazioni di divenire effettive (serve per evitare falsi interventi dell'interrupt alla partenza). Solo successivamente viene attivato l'interrupt e azzerato il relativo flag

Controlla il bit1 della variabile QUALE: se è 1 salta la successiva istruzione e inverte lo stato del LED1

Invece di introdurre un ulteriore test che stabilisca la successiva istruzione da eseguire (*bcf* per azzerare o *bsf* per asserire) invertiamo lo stato del bit sfruttando l'ALU interna tramite l'istruzione *xorwf* che effettua lo XOR tra il byte nel registro W e il contenuto di un registro, salvando il risultato nel registro stesso.

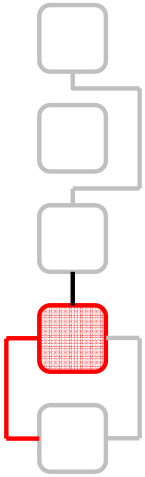
**A XOR 0 = A**
**A XOR 1 = !A**

# OPERAZIONI DI MASCHERAMENTO CON OPERATORI LOGICI: XOR




A xor B = OUT

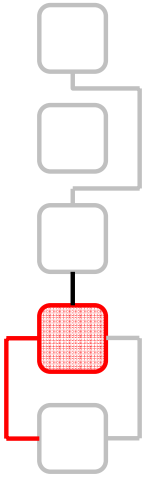
| A | B | OUT |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |



```
            call    Delay
            bcf     INTCON,0
            bsf     INTCON,3

Main

            call    Delay
            btfss   QUALE,1
            goto    led1_fermo
            movlw   00000001B
            xorwf   PORTB,F

led1_fermo

            btfss   QUALE,2
            goto    Main
            movlw   00000010B
            xorwf   PORTB,F

            goto    Main
```

Se invece il bit1 di QUALE dovesse essere 0 salta all'etichetta *led1_fermo*

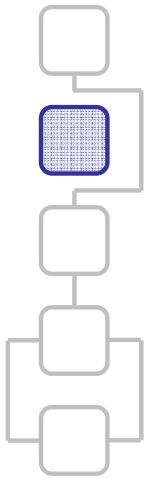
Come prima controlla lo stato del bit2 della variabile QUALE che indica se il LED2 deve lampeggiare. Se il bit2 è 1 allora inverte lo stato del LED2 altrimenti torna all'inizio del *Main*



```
            call    Delay
            bcf     INTCON,0
            bsf     INTCON,3

Main

            call    Delay
            btfss   QUALE,1
            goto    led1_fermo
            movlw   00000001B
            xorwf   PORTB,F

led1_fermo

            btfss   QUALE,2
            goto    Main
            movlw   00000010B
            xorwf   PORTB,F

            goto    Main
```

Se invece il bit1 di QUALE dovesse essere 0 salta all'etichetta *led1_fermo*

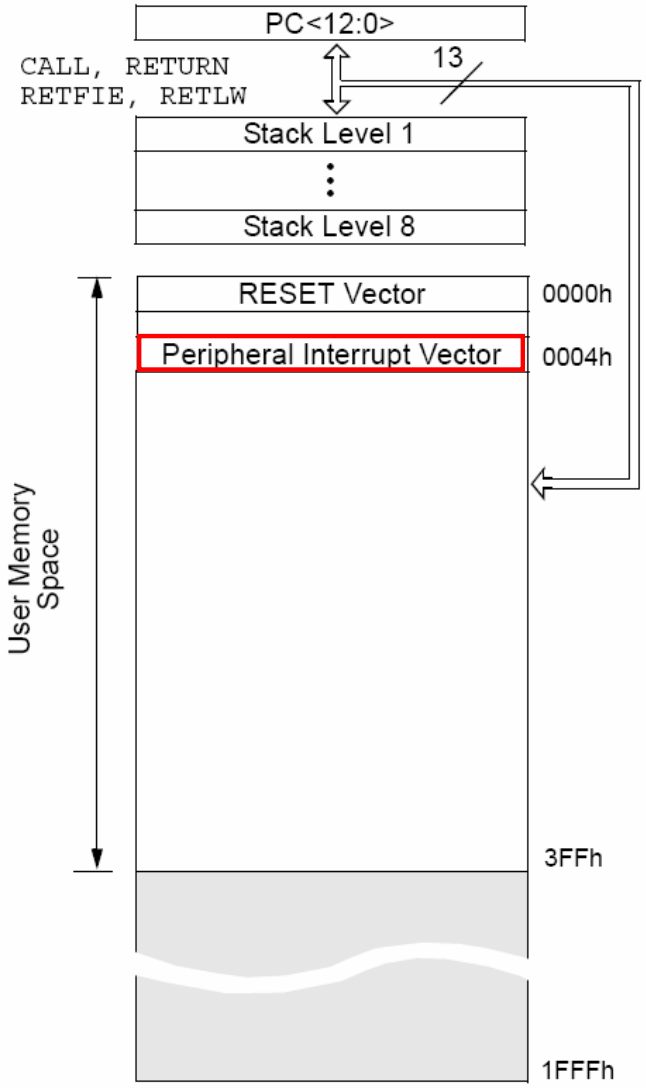
Come prima controlla lo stato del bit2 della variabile QUALE che indica se il LED2 deve lampeggiare. Se il bit2 è 1 allora inverte lo stato del LED2 altrimenti torna all'inizio del *Main*



```
        ORG     04H
```

Indica che le istruzioni seguenti verranno memorizzate a partire dall'indirizzo 0004h, ovvero a partire dall'interrupt vector

```
        "Context Push"

        btfsc   PORTB,SWITCH1
        goto    altro_led
        movlw   00000010B
        xorwf   QUALE,F
        goto    fine_int

altro_led

        btfsc   PORTB,SWITCH2
        goto    fine_int
        movlw   00000100B
        xorwf   QUALE,F

fine_int

        "Context Pop"

        bcf     INTCON,0
        retfie
```



## CONTEXT SAVING

Quando si entra nella routine di interrupt la prima operazione da eseguire, dopo aver disabilitato l'interrupt per evitare nidificazioni, è quella del context saving

Il "salvataggio del contesto" è infatti necessario al fine di garantire che l'interrupt risulti TRASPARENTE al programma principale e non ne pregiudichi il funzionamento.

```
(…)
 ↓
 ↓
movlw   11111100B
 ↓
 ↓
movwf   TRISB
 ↓
 ↓
(…)
```

## CONTEXT SAVING

Quando si entra nella routine di interrupt la prima operazione da eseguire, dopo aver disabilitato l'interrupt per evitare nidificazioni, è quella del context saving

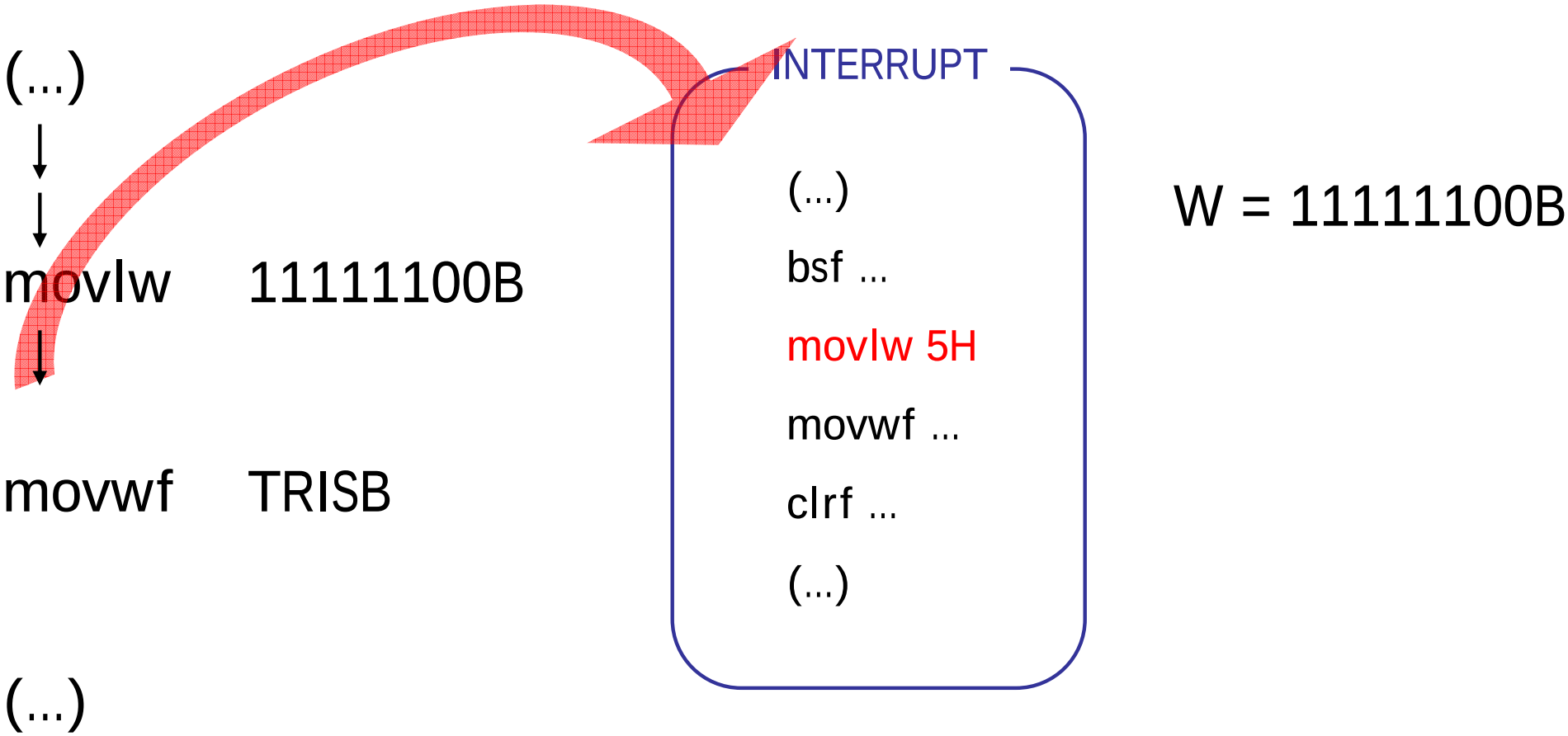
Il "salvataggio del contesto" è infatti necessario al fine di garantire che l'interrupt risulti TRASPARENTE al programma principale e non ne pregiudichi il funzionamento.

```
(…)
 ↓
 ↓
movlw   11111100B
 ↓

movwf   TRISB


(…)
```

## CONTEXT SAVING

Quando si entra nella routine di interrupt la prima operazione da eseguire, dopo aver disabilitato l'interrupt per evitare nidificazioni, è quella del context saving

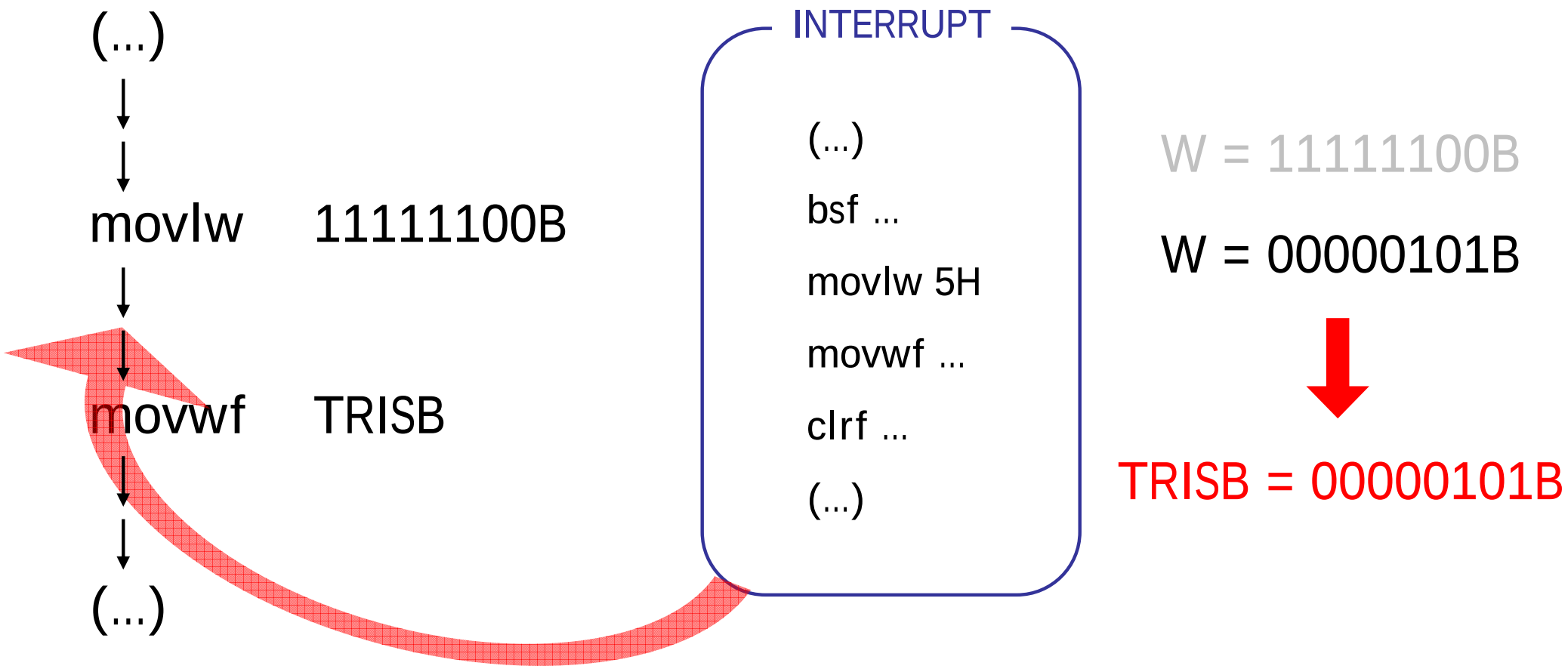
Il "salvataggio del contesto" è infatti necessario al fine di garantire che l'interrupt risulti TRASPARENTE al programma principale e non ne pregiudichi il funzionamento.



## CONTEXT SAVING

Quando si entra nella routine di interrupt la prima operazione da eseguire, dopo aver disabilitato l'interrupt per evitare nidificazioni, è quella del context saving

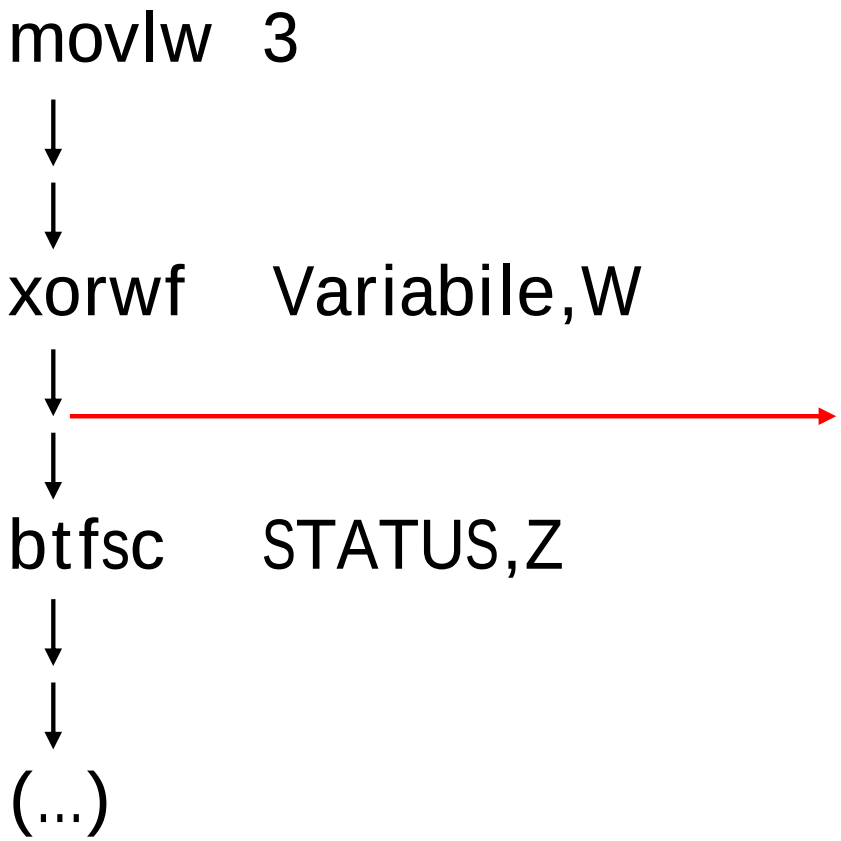
Il "salvataggio del contesto" è infatti necessario al fine di garantire che l'interrupt risulti TRASPARENTE al programma principale e non ne pregiudichi il funzionamento.



## CONTEXT SAVING

Quando si entra nella routine di interrupt la prima operazione da eseguire, dopo aver disabilitato l'interrupt per evitare nidificazioni, è quella del context saving

Il "salvataggio del contesto" è infatti necessario al fine di garantire che l'interrupt risulti TRASPARENTE al programma principale e non ne pregiudichi il funzionamento.




Qui in mezzo non deve accadere nulla ai flag del registro di STATUS (in particolare a Z) altrimenti invalideremmo il test!!!

# CONTEXT SAVING: COME APPLICARLO

Il context saving viene eseguito semplicemente salvando in opportune variabili temporanee create appositamente il valore del registro W e del registro STATUS

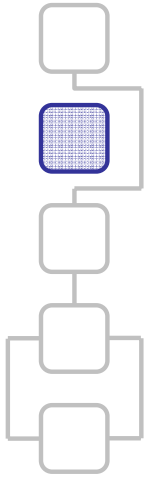
E' necessario però effettuare l'operazione utilizzando solamente istruzioni che non modifichino in alcun modo i flag del registro STATUS: ecco il motivo per cui si utilizza l'istruzione SWAPF invece di una più intuitiva istruzione MOVF

```
ORG     04H

"Context Push"

(...)

"Context Pop"

bcf     INTCON,0
retfie
```

```
movwf   W_TEMP
swapf   STATUS, W
movwf   STATUS_TEMP
```

```
swapf   STATUS_TEMP,W
movwf   STATUS
swapf   W_TEMP, F
swapf   W_TEMP, W
```

Logicamente all'inizio del programma è necessario definire le variabili STATUS_TEMP e W_TEMP



```
        ORG     04H
```

Indica che le istruzioni seguenti verranno memorizzate a partire dall'indirizzo 0004h, ovvero a partire dall'interrupt vector

```
        "Context Push"

        btfsc   PORTB,SWITCH1
        goto    altro_led
        movlw   00000010B
        xorwf   QUALE,F
        goto    fine_int
```

Controlla se è stato premuto il pulsante 1 (verifica che il relativo pin sia a 0). Se Si viene invertito il valore del bit1 della variabile QUALE che dice se il LED1 deve o no lampeggiare

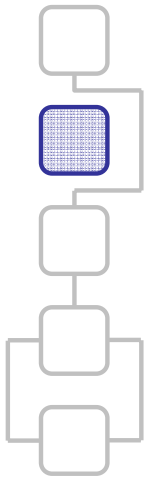
```
altro_led

        btfsc   PORTB,SWITCH2
        goto    fine_int
        movlw   00000100B
        xorwf   QUALE,F
```

Se non è stato premuto il pulsante 1 viene controllato il pulsante 2.

In linea di principio questo secondo controllo non sarebbe necessario, ma viene fatto per maggiore sicurezza.

```
fine_int

        "Context Pop"

        bcf     INTCON,0
        retfie
```

Alla fine della procedura di gestione dell'interrupt viene riabilitato l'interrupt azzerando il flag associato (ogni volta che si verifica l'interrupt è necessario azzerare via firmware il flag per riattivarlo).



```
        ORG     04H
```

Indica che le istruzioni seguenti verranno memorizzate a partire dall'indirizzo 0004h, ovvero a partire dall'interrupt vector

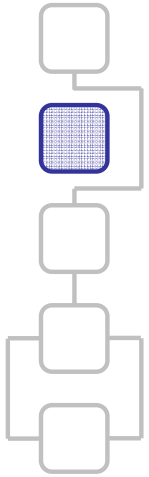
```
        "Context Push"

        btfsc   PORTB,SWITCH1
        goto    altro_led
        movlw   00000010B
        xorwf   QUALE,F
        goto    fine_int

altro_led

        btfsc   PORTB,SWITCH2
        goto    fine_int
        movlw   00000100B
        xorwf   QUALE,F

fine_int

        "Context Pop"

        bcf     INTCON,0
        retfie
```

Controlla se è stato premuto il pulsante 1 (verifica che il relativo pin sia a 0). Se Si viene invertito il valore del bit1 della variabile QUALE che dice se il LED1 deve o no lampeggiare

Se non è stato premuto il pulsante 1 viene controllato il pulsante 2.

In linea di principio questo secondo controllo non sarebbe necessario, ma viene fatto per maggiore sicurezza.

Alla fine della procedura di gestione dell'interrupt viene riabilitato l'interrupt azzerando il flag associato (ogni volta che si verifica l'interrupt è necessario azzerare via firmware il flag per riattivarlo).



```
        ORG     04H
```

Indica che le istruzioni seguenti verranno memorizzate a partire dall'indirizzo 0004h, ovvero a partire dall'interrupt vector

```
        "Context Push"

        btfsc   PORTB,SWITCH1
        goto    altro_led
        movlw   00000010B
        xorwf   QUALE,F
        goto    fine_int
```

Controlla se è stato premuto il pulsante 1 (verifica che il relativo pin sia a 0). Se Si viene invertito il valore del bit1 della variabile QUALE che dice se il LED1 deve o no lampeggiare

```
altro_led

        btfsc   PORTB,SWITCH2
        goto    fine_int
        movlw   00000100B
        xorwf   QUALE,F
```

Se non è stato premuto il pulsante 1 viene controllato il pulsante 2.

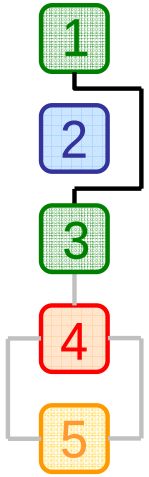
In linea di principio questo secondo controllo non sarebbe necessario, ma viene fatto per maggiore sicurezza.

```
fine_int

        "Context Pop"

        bcf     INTCON,0
        retfie
```

Alla fine della procedura di gestione dell'interrupt viene riabilitato l'interrupt azzerando il flag associato (ogni volta che si verifica l'interrupt è necessario azzerare via firmware il flag per riattivarlo).

```
            PROCESSOR   16F84A
            RADIX       DEC
            INCLUDE     "P16F84A.INC"

LED1        EQU         0
LED2        EQU         1
SWITCH1     EQU         6
SWITCH2     EQU         7

            ORG         0CH
Count       RES 2
QUALE       RES 1

            ORG         00H
            goto        inizio
```

1




```
            ORG         04H

            "Context Push"

            btfsc       PORTB,SWITCH1
            goto        altro_led
            movlw       00000010B
            xorwf       QUALE,F
            goto        fine_int
altro_led
            btfsc       PORTB,SWITCH2
            goto        fine_int
            movlw       00000100B
            xorwf       QUALE,F
fine_int
            "Context Pop"

            bcf         INTCON,0
            retfie
```

2

```
inizio
            bsf         STATUS,RP0

            movlw       00011111B
            movwf       TRISA

            movlw       11111100B
            movwf       TRISB

            movlw       01111111B
            andwf       OPTION_REG

            movlw       10000000B
            movwf       INTCON

            bcf         STATUS,RP0

            clrf        QUALE
```

3

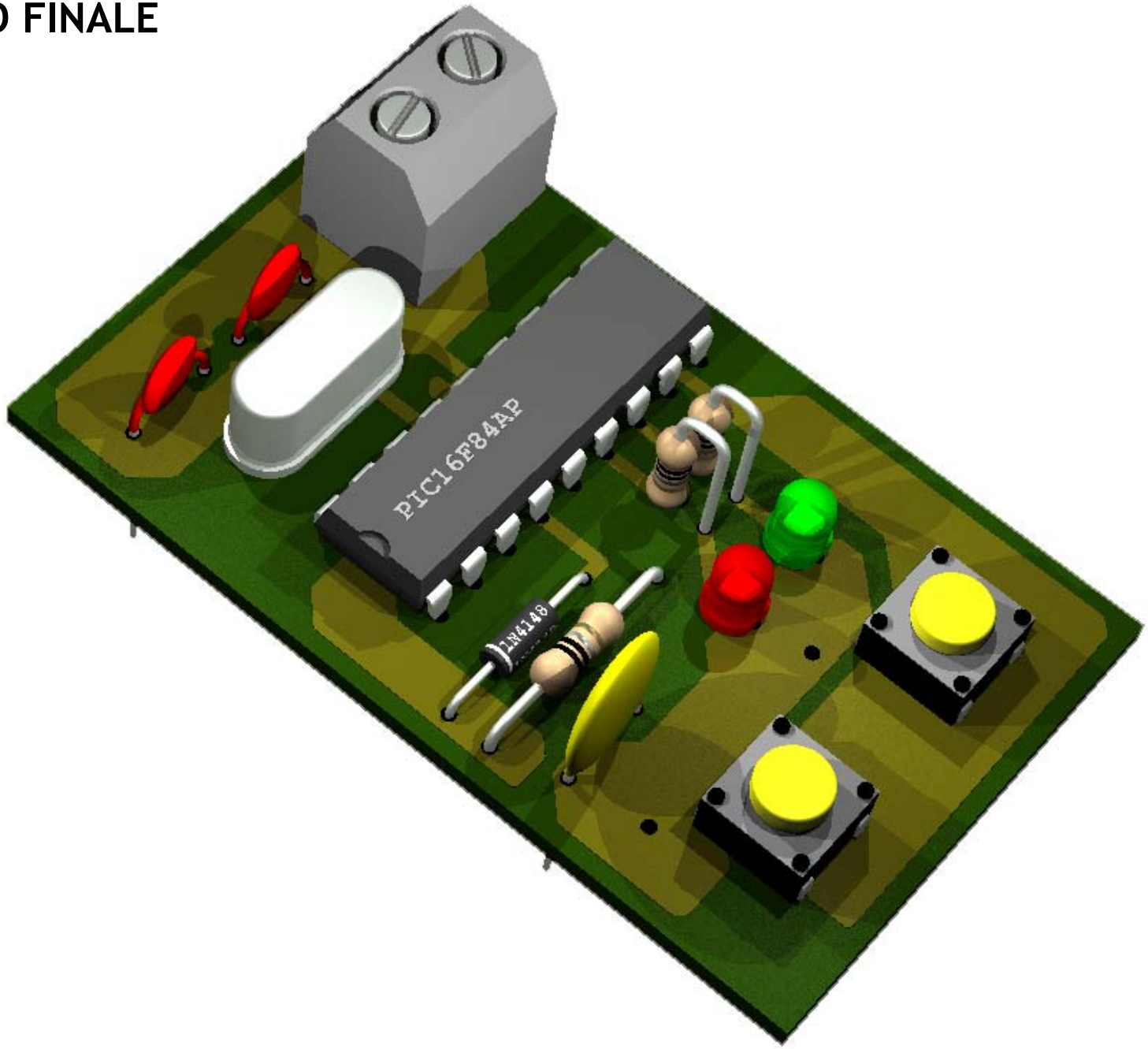
```
            call        Delay
            bcf         INTCON,0
            bsf         INTCON,3

Main
            call        Delay
            btfss       QUALE,1

            goto        led1_fermo
            movlw       00000001B
            xorwf       PORTB,F
led1_fermo
            btfss       QUALE,2

            goto        Main
            movlw       00000010B
            xorwf       PORTB,F
            goto        Main
```

4

```
Delay
            clrf        Count
            clrf        Count+1

DelayLoop
            decfsz      Count,1
            goto        DelayLoop
            decfsz      Count+1,1
            goto        DelayLoop
            return

            END
```

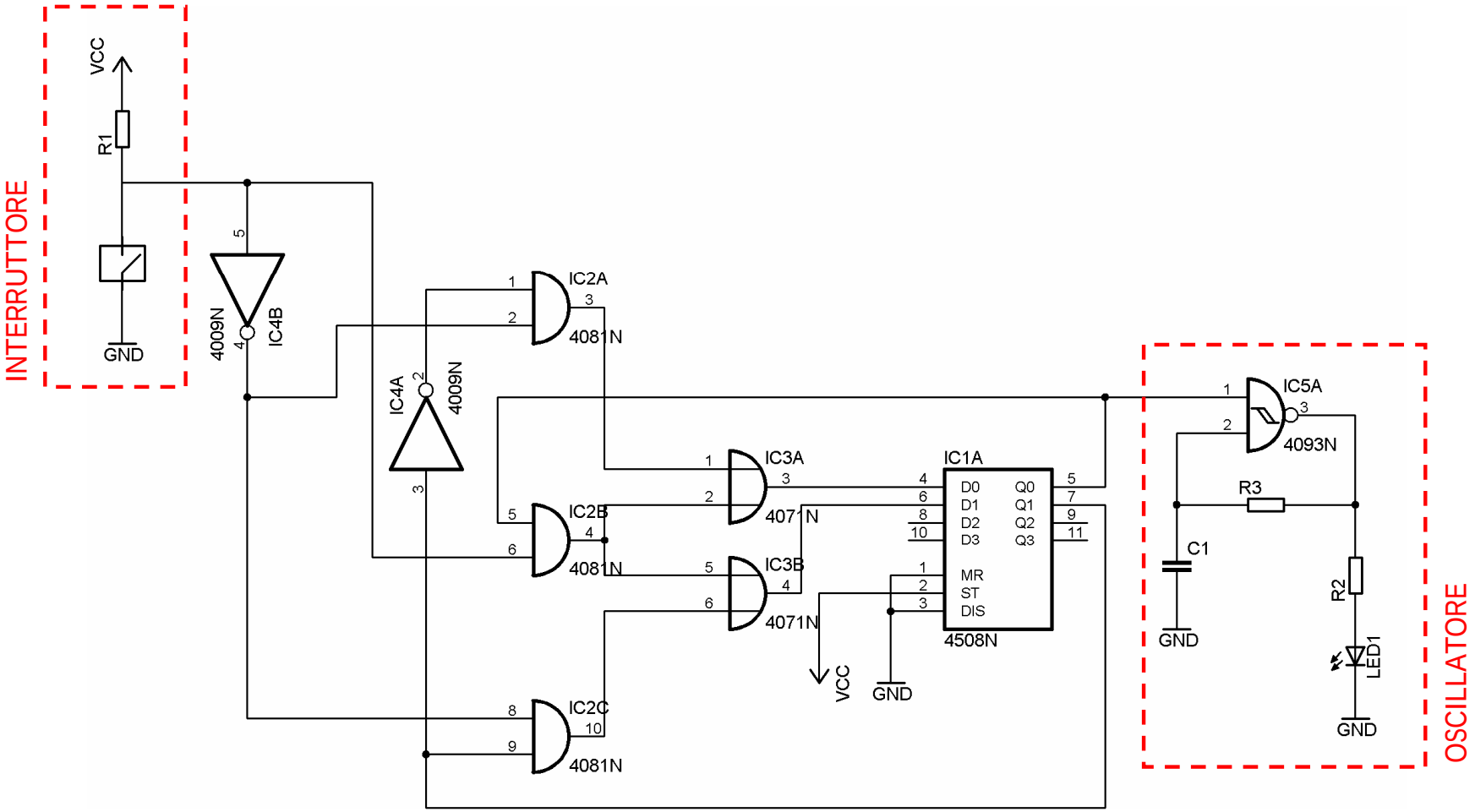
5

# IL CIRCUITO FINALE



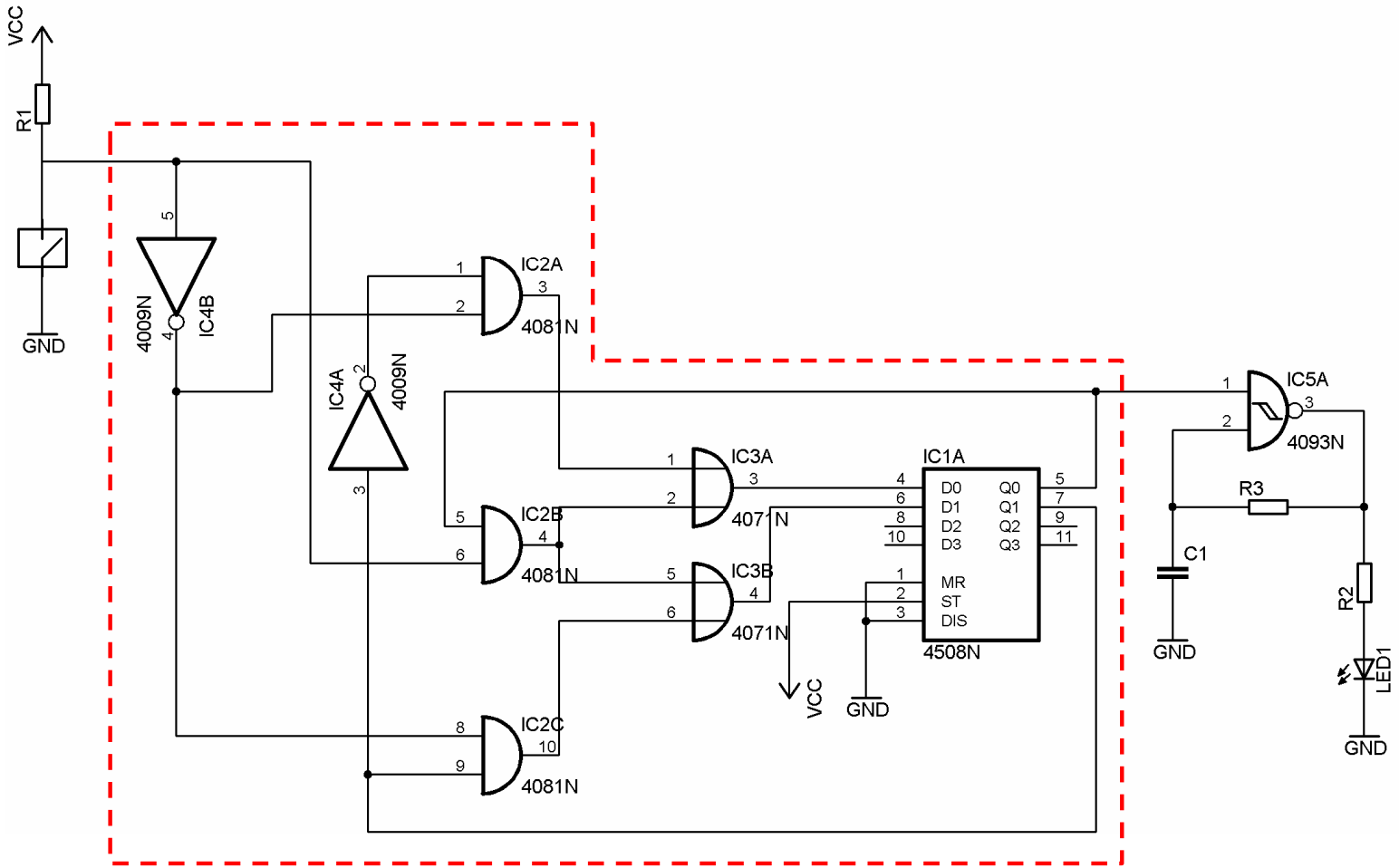
# CONFRONTO CON CIRCUITO A LOGICA DISCRETA

*Si confrontano solo alcune porzioni del circuito per semplicità.*
*Il circuito è semplificato al massimo, al limite dell'ideale.*



# CONFRONTO CON CIRCUITO A LOGICA DISCRETA

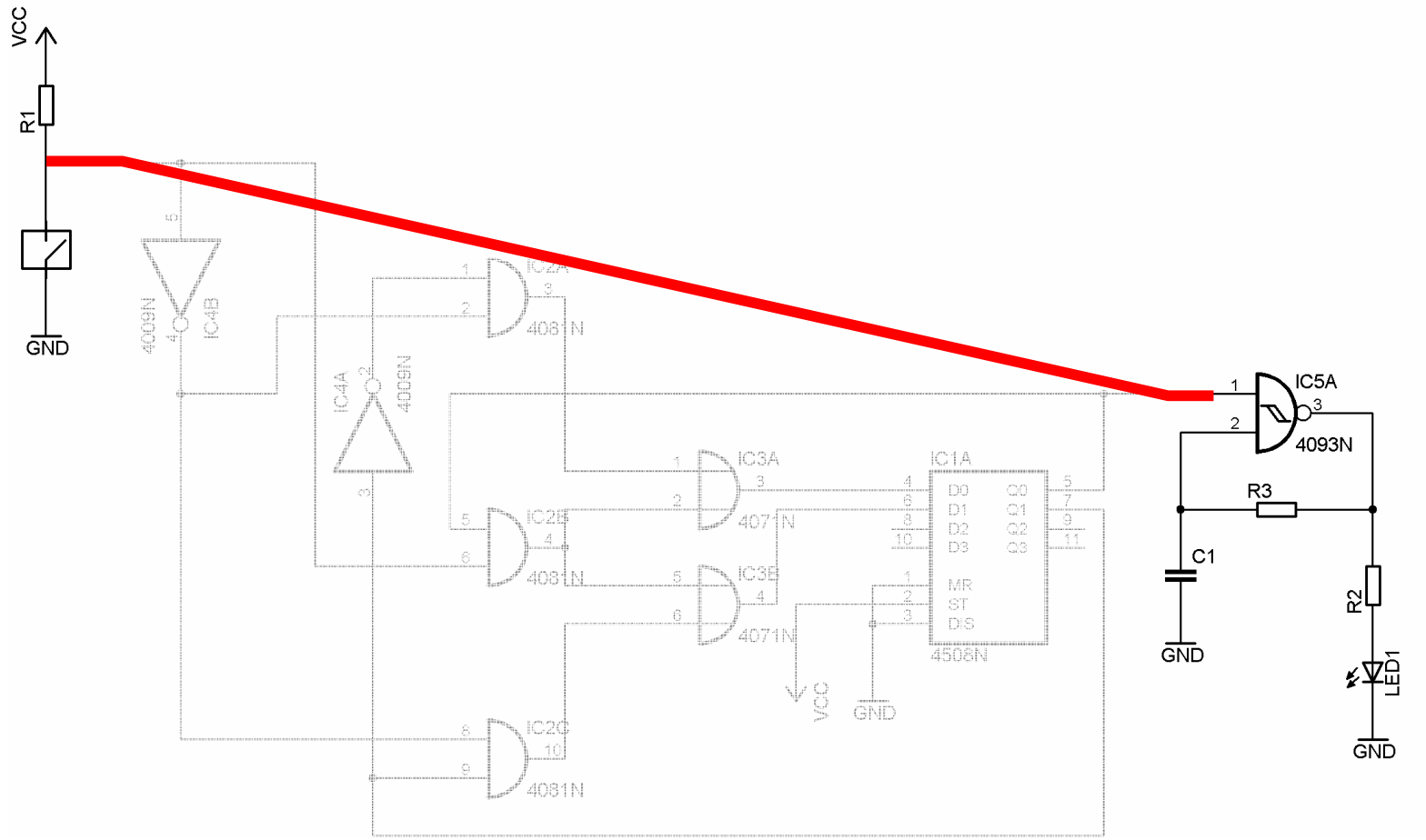
*Si confrontano solo alcune porzioni del circuito per semplicità.*
*Il circuito è semplificato al massimo, al limite dell'ideale.*




A COSA SERVE?

# CONFRONTO CON CIRCUITO A LOGICA DISCRETA

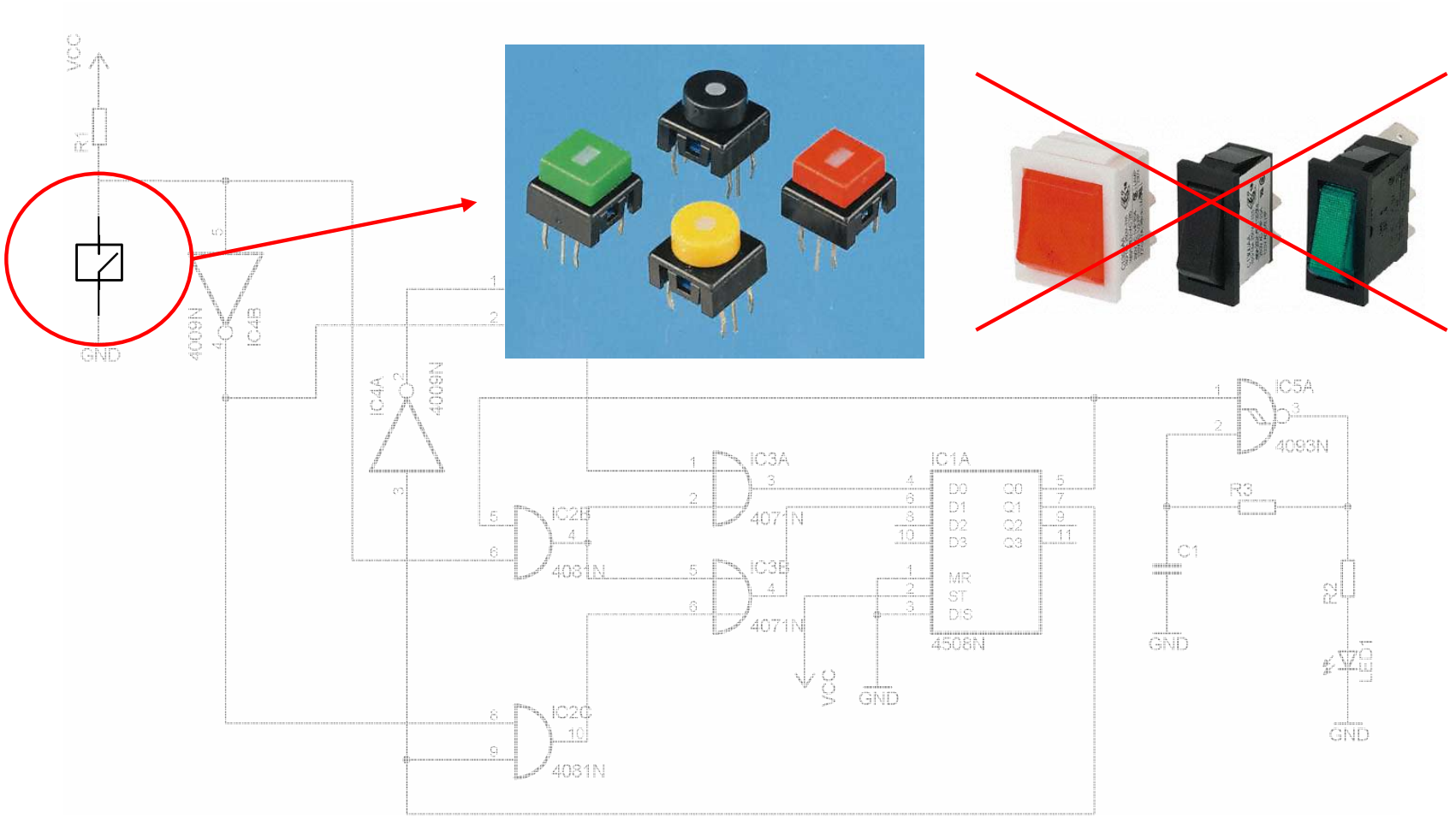
*Si confrontano solo alcune porzioni del circuito per semplicità.*
*Il circuito è semplificato al massimo, al limite dell'ideale.*




NON POTREMMO SEMPLICEMENTE COLLEGARLI COSI'?

# CONFRONTO CON CIRCUITO A LOGICA DISCRETA

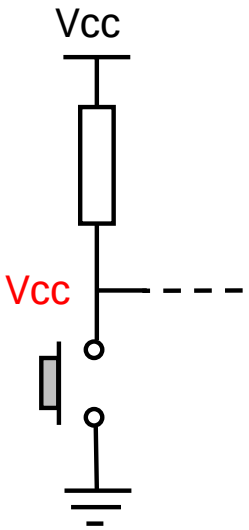
*Si confrontano solo alcune porzioni del circuito per semplicità.*
*Il circuito è semplificato al massimo, al limite dell'ideale.*

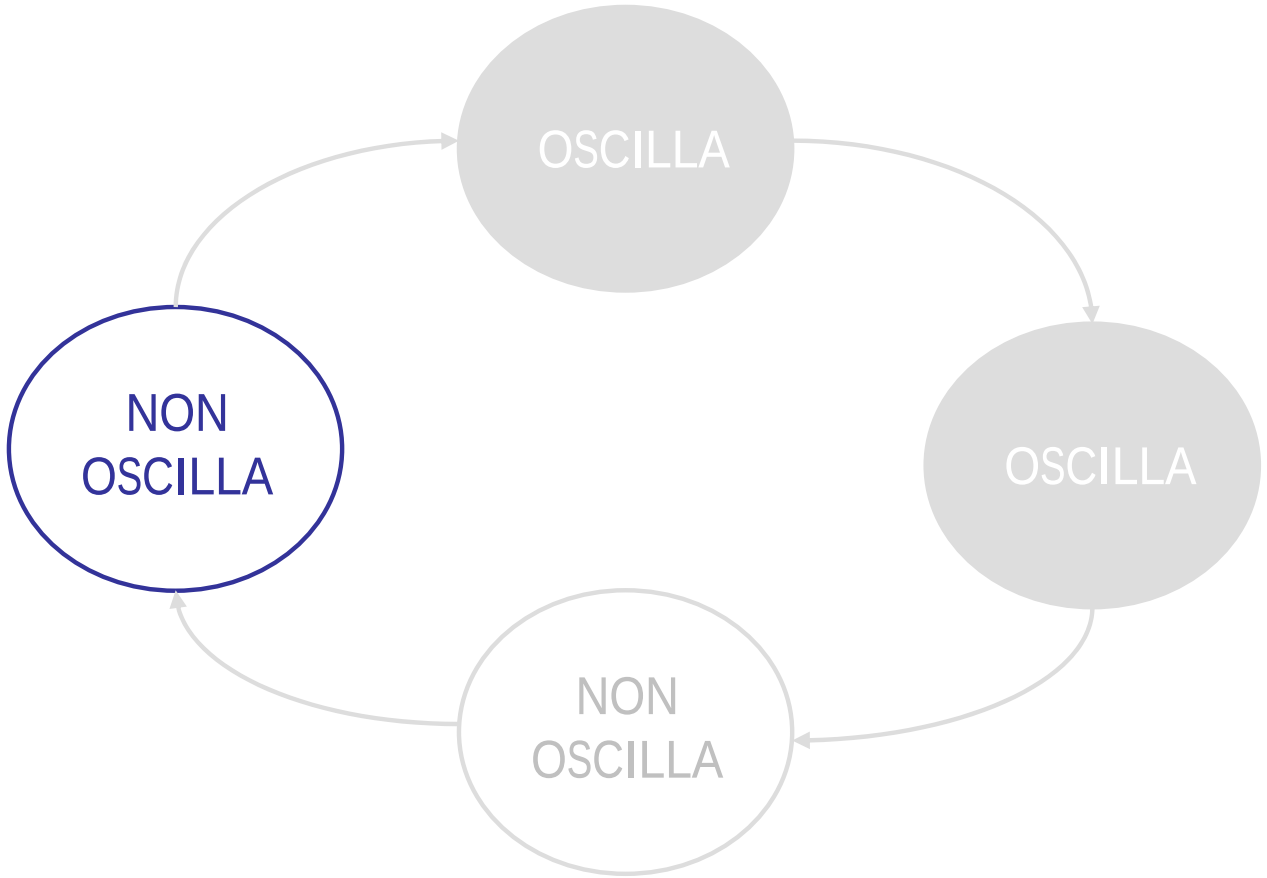



NOI VOGLIAMO UN PULSANTE, NON UN INTERRUTTORE!!!

# CONFRONTO CON CIRCUITO A LOGICA DISCRETA

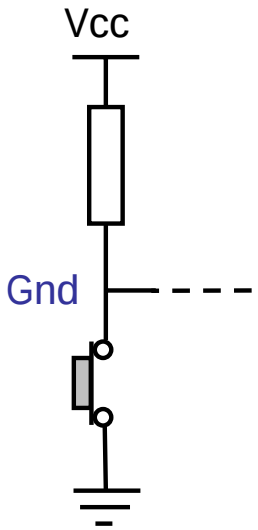
*Diagramma degli stati del circuito di gestione del pulsante*

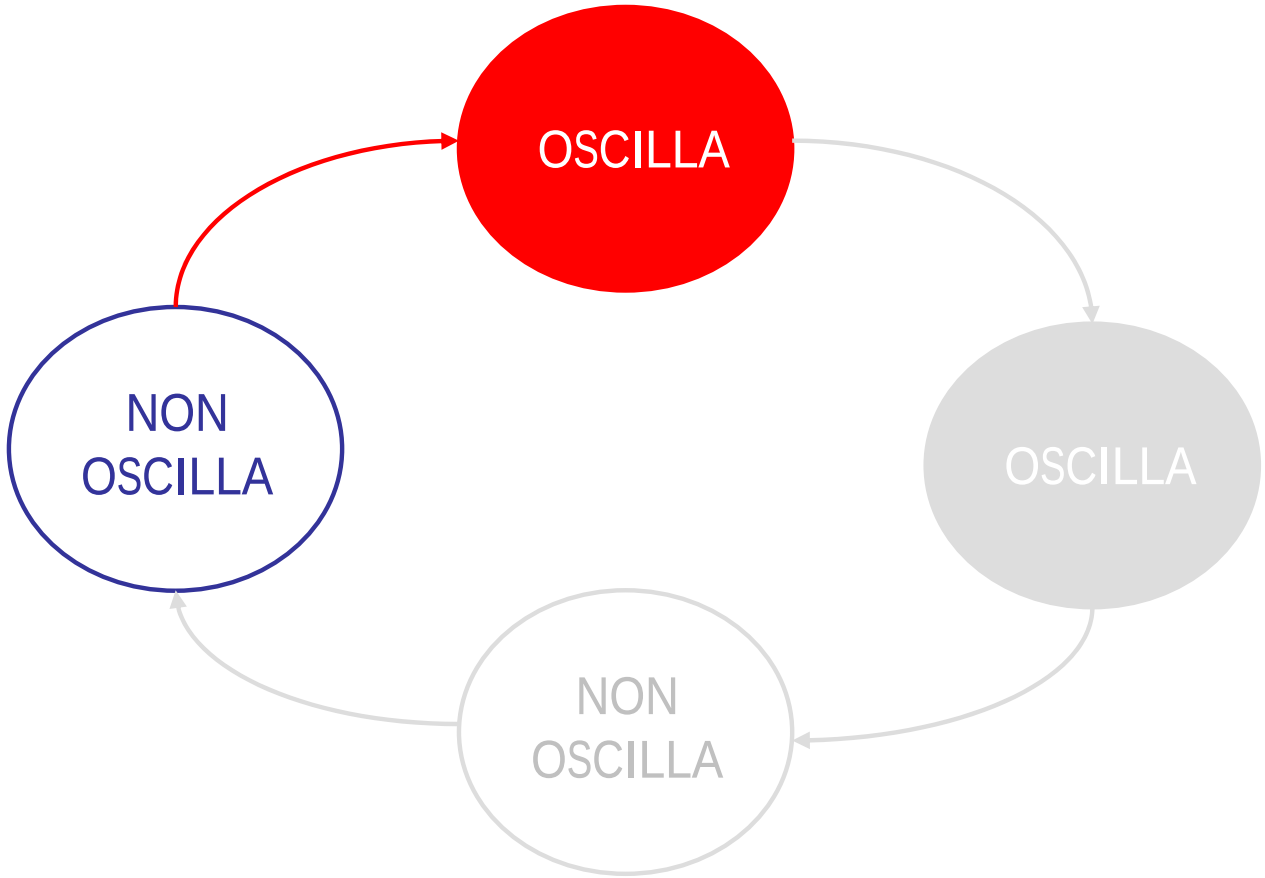


# CONFRONTO CON CIRCUITO A LOGICA DISCRETA

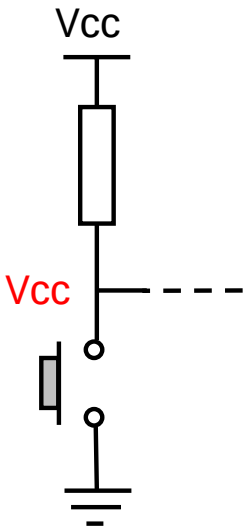
*Diagramma degli stati del circuito di gestione del pulsante*

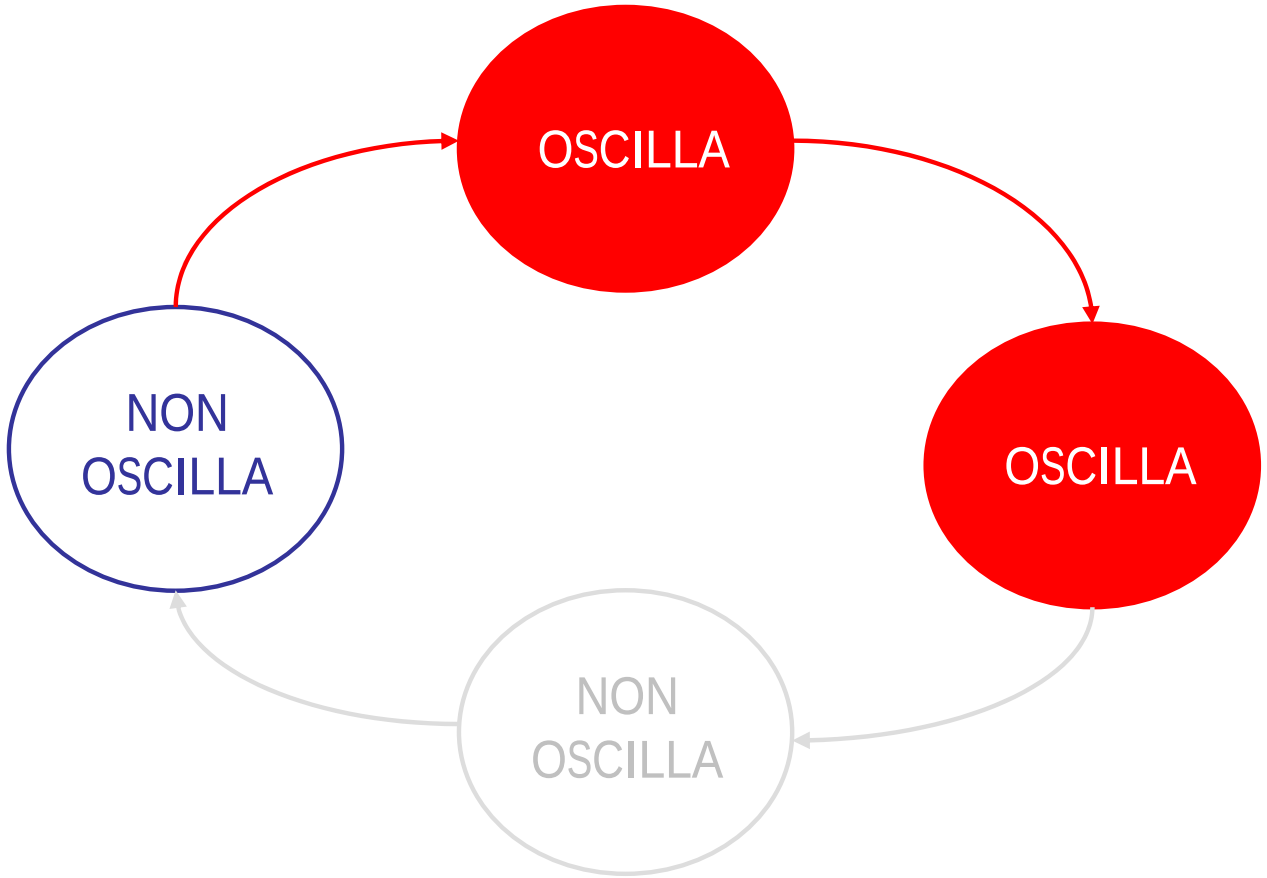


# CONFRONTO CON CIRCUITO A LOGICA DISCRETA

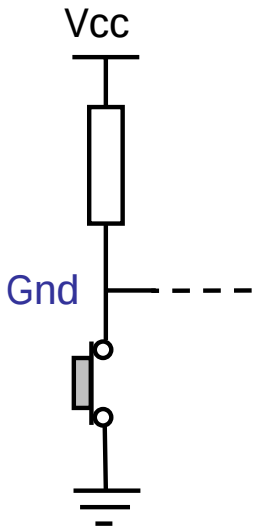
*Diagramma degli stati del circuito di gestione del pulsante*

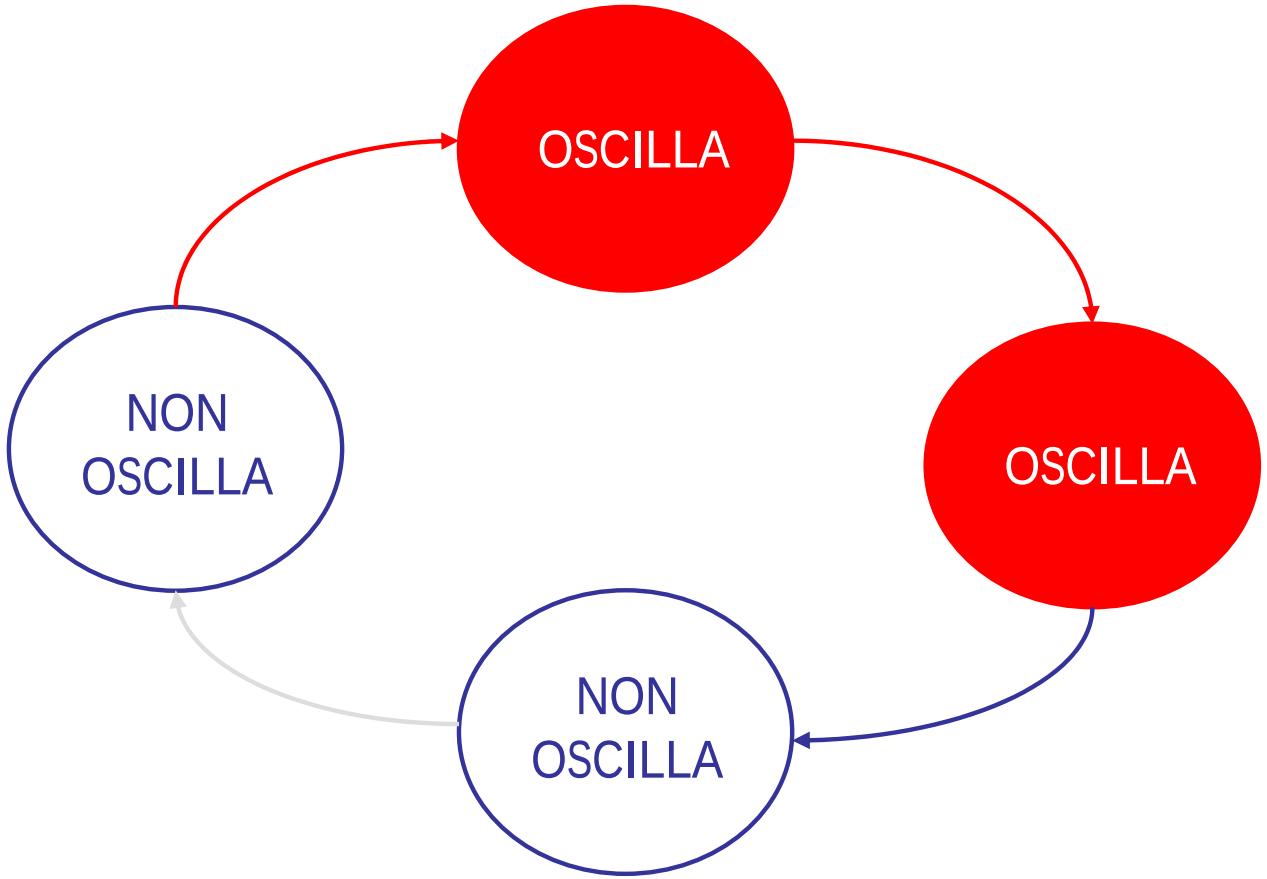


# CONFRONTO CON CIRCUITO A LOGICA DISCRETA

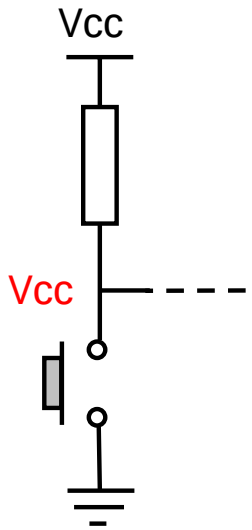
*Diagramma degli stati del circuito di gestione del pulsante*

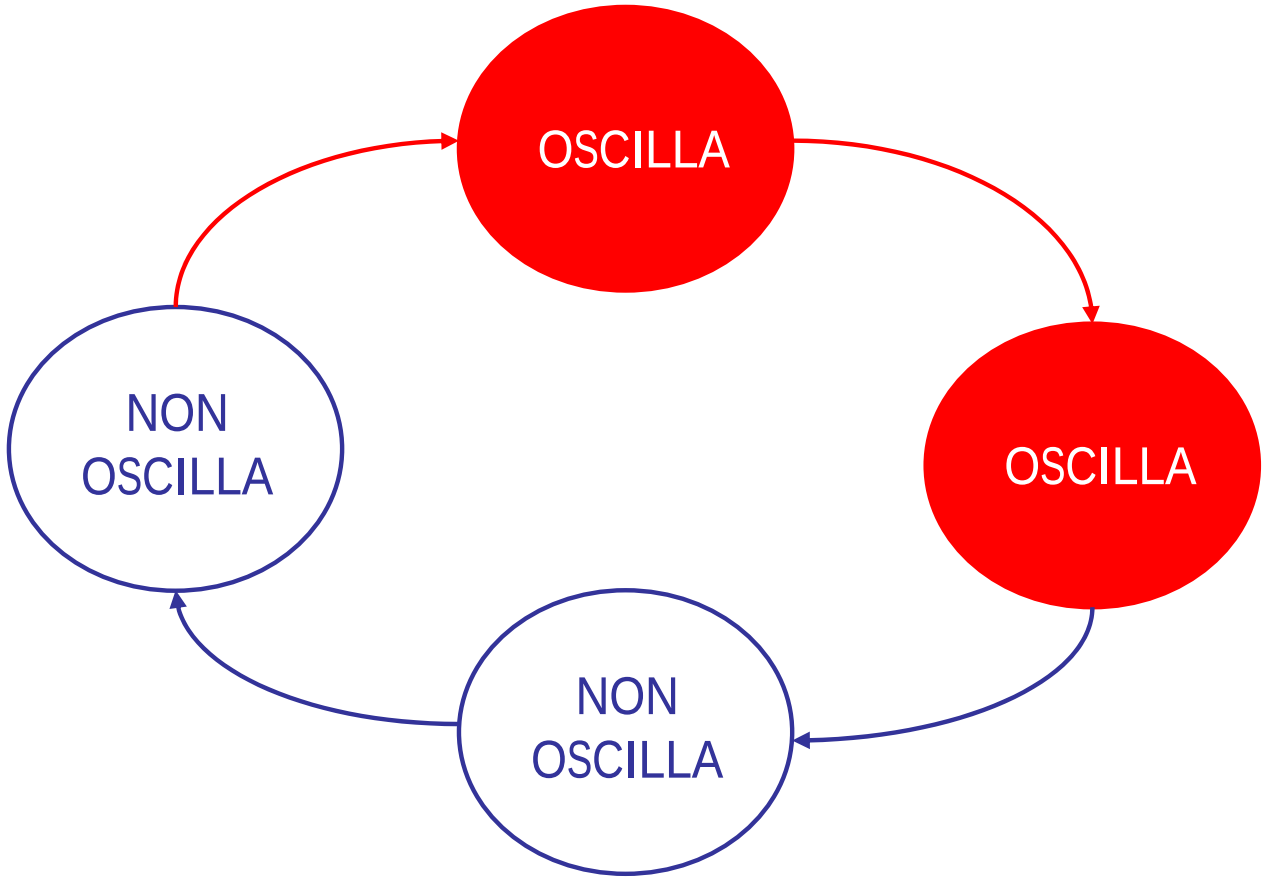


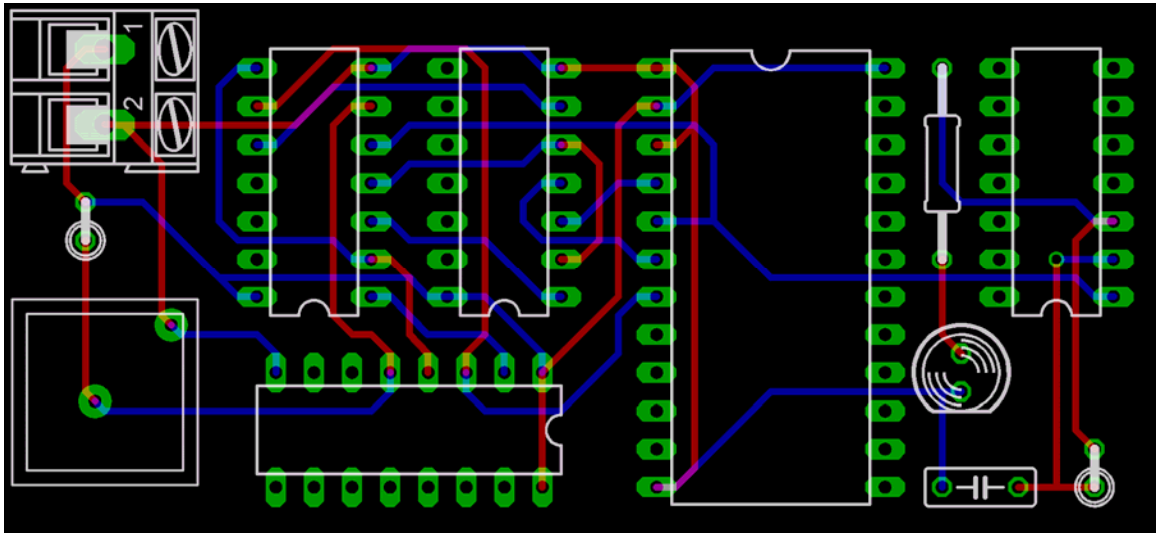
# CONFRONTO CON CIRCUITO A LOGICA DISCRETA

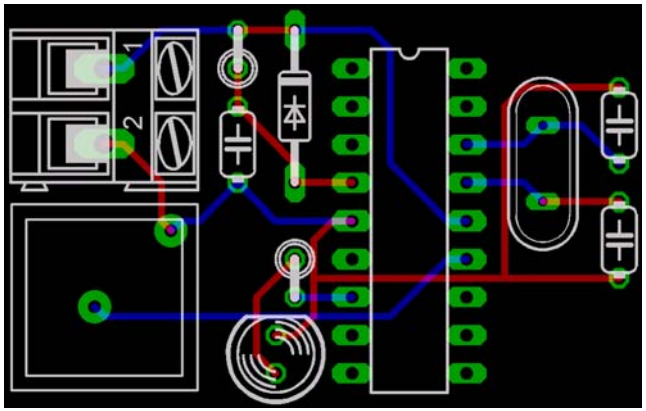
*Diagramma degli stati del circuito di gestione del pulsante*

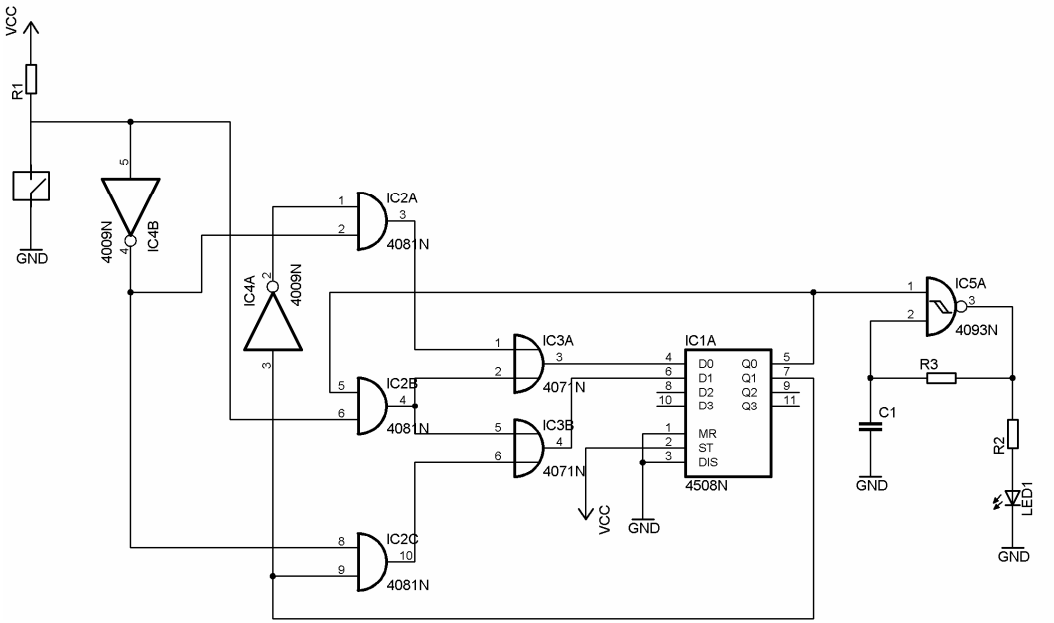


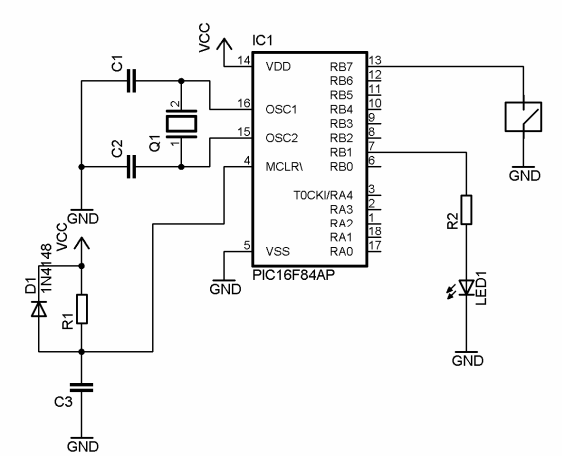



# CONFRONTO CON CIRCUITO A LOGICA DISCRETA

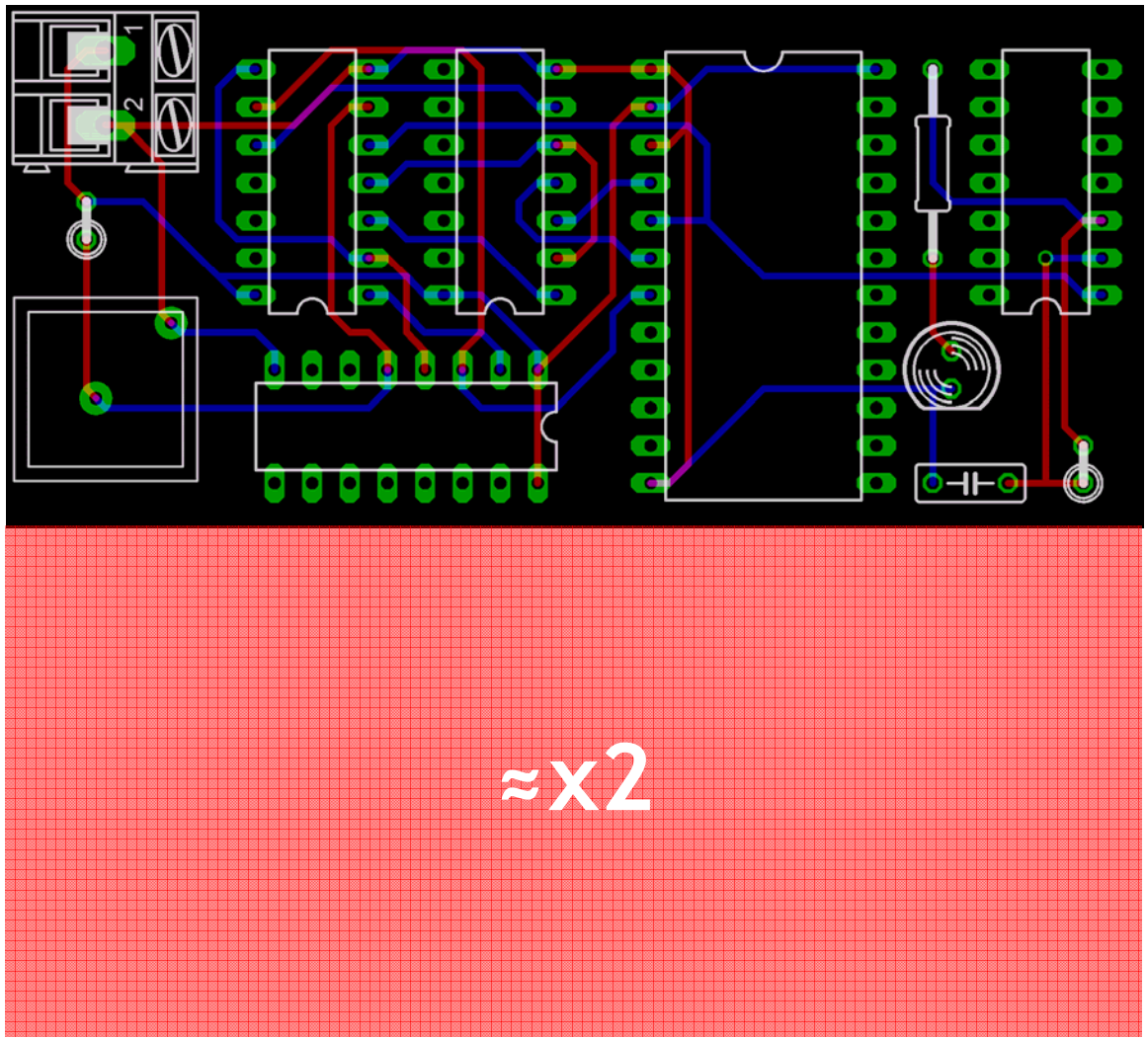


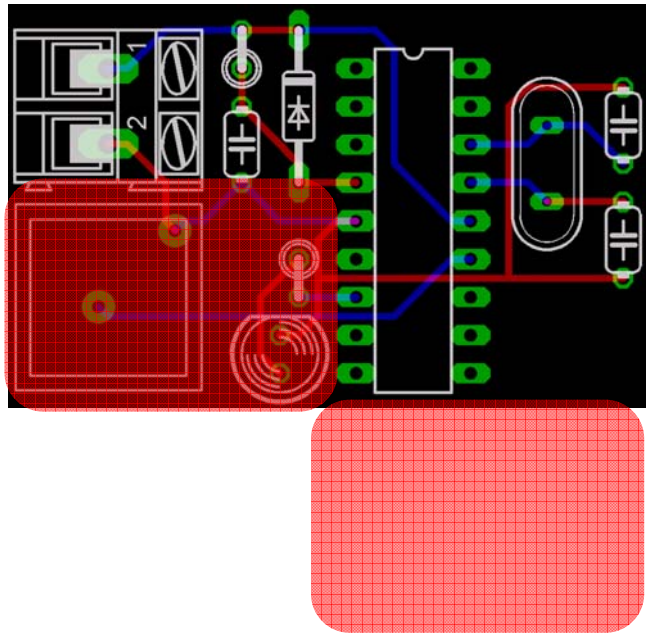




# CONFRONTO CON CIRCUITO A LOGICA DISCRETA

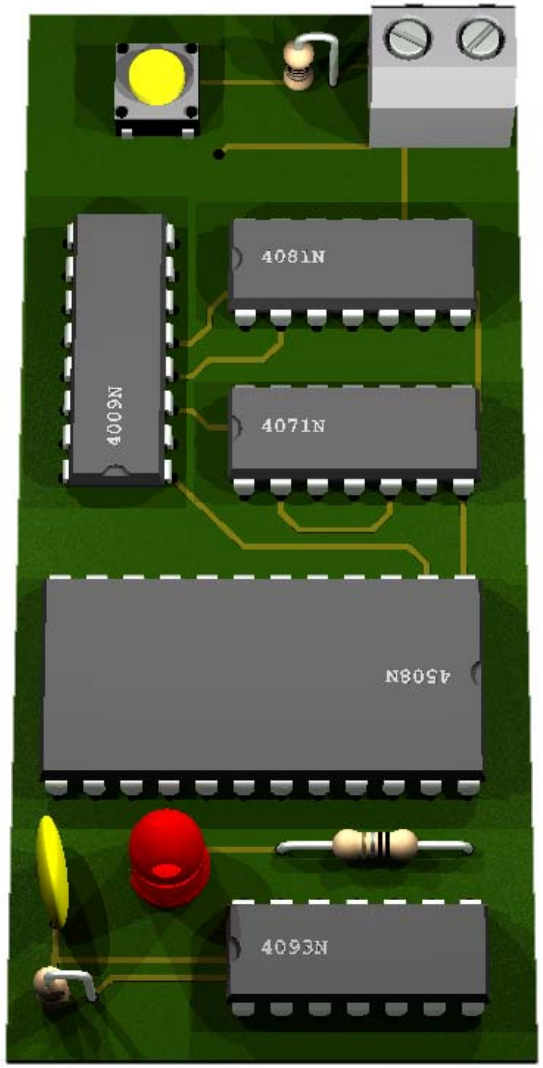


La differenza tra i due circuiti è ancora più evidente se i LED da pilotare aumentano.

Passando da uno a due LED nel caso del circuito con MCU è sufficiente aggiungere un resistore e un pulsante, nel caso del circuito a discreti invece il pcb circa raddoppia!!!

# CONFRONTO CON CIRCUITO A LOGICA DISCRETA




**PER QUALUNQUE INFORMAZIONE, DOMANDA, INSULTO**

fabrizio.guerrieri@ieee.org

**GRAZIE PER L'ATTENZIONE**