Esercitazioni PIC Midrange.

Inserto_c: paging.

Analogamente ai banchi per la memoria RAM, anche la memoria programma può essere divisa in pagine. Vediamo cosa comporta questo.

Memoria RAM e memoria programma.

Ci può essere una certa confusione tra la memoria programma e la memoria RAM.
Ricordiamo che i PIC sono in architettura Harvard, il che significa una separazione totale tra il bus dati, che accede alla RAM attraverso le istruzioni, e il bus istruzioni, che accede alla memoria programma attraverso i meccanismi del Program Counter.

I due bus non solo sono separati, ma hanno anche diversa dimensione:

Famiglia	Bus istruzioni	Bus dati
Baseline	12 bit	8 bit
Midrange	14 bit	8 bit
Enhanced Midrange	14 bit	8 bit
PIC18F	16 bit	8 bit

In questo senso, parlare di byte (= 8 bit) per la memoria RAM è perfettamente corretto, ma è in uso semplificare parlando di byte anche per la memoria programma, anche se ampia 14 bit e non 8.

Nelle dimensioni obbligate del bus istruzioni devono essere contenuti gli opcpodes, ovvero la codifica binaria dell'istruzione, che comprende sia il codice specifico dell'azione da compiere, sia l'oggetto o la destinazione.

Ad esempio, nel caso di una scrittura di un bit, **`bcf`** o **`bsf`**, nell'opcode deve essere contenuta sia l'operazione da eseguire che il bit da modificare e l'indirizzo in cui modificarlo, con i problemi di banking descritti nell'inserto precedente.
Attenzione che in questo caso stiamo parlando di indirizzi del bus dati, diretti alla memoria RAM.

Se consideriamo il caso di un salto **`goto`** o di una chiamata di subroutine **`call`**, l'opcode sarà composto similmente dal codice dell'azione da eseguire e dall'indirizzo di destinazione.
Attenzione che in questo caso si tratta di un indirizzo nella memoria programma: le due istruzioni deviano l'esecuzione da una posizione ad un'altra nella memoria programma.
Per fare questo modificano il contenuto del Program Counter.

Vogliamo ancora una volta ricordare il meccanismo del ***Program Counter.***
Questo registro punta l'istruzione che dovrà essere eseguita ed è incrementato automaticamente dai meccanismi dell'unità centrale.
In particolare, viene forzato a zero all'accensione e ai reset e questo fa si che la prima istruzione da

eseguire sia quella posta all'indirizzo 0000h della memoria programma.

Seguendo il clock, i meccanismi di indirizzamento prelevano l'istruzione, la eseguono e da questa traggono anche l'indicazione dell'incremento da applicare al Program Counter per il prelievo dell' istruzione successiva. Nel casso di istruzioni "comuni", come **`nop, bcf, movlw`**, ecc, si tratterà di una unità: il PC sarà incrementato di 1 e punterà all'indirizzo 0001h, dove il ciclo si ripeterà.

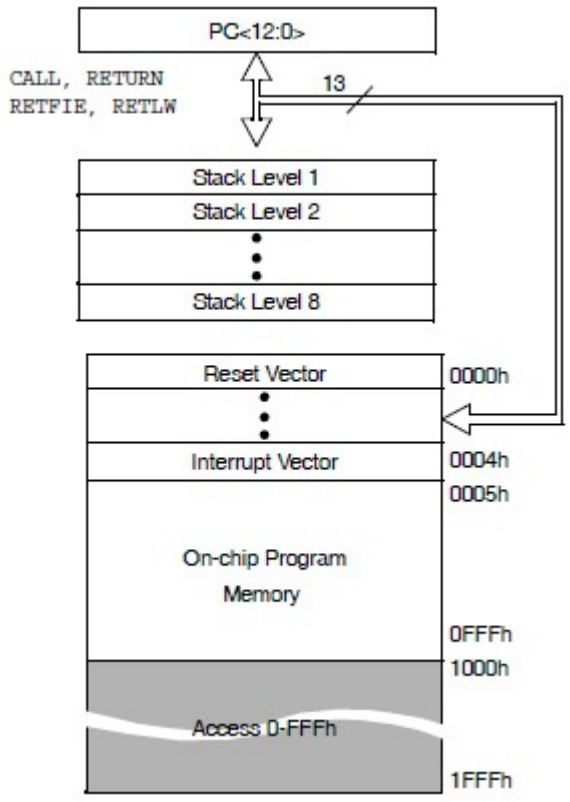
Dunque, il valore contenuto nel PC viene modificato dal flusso delle istruzioni in modo automatico:

- al **reset viene forzato a 0**
- all'**interrupt viene forzato a 4**
- l'esecuzione di una istruzione generica lo incrementa di una unità, puntando alla successiva
- una istruzione di salto condizionato, se eseguita, aumenta di due, saltando l'istruzione seguente. Queste istruzioni modificano il PC solo di una quantità fissa (1 o 2).
- una chiamata a subroutine o un salto non condizionato ne modifica il valore sostituendolo con l'indirizzo di destinazione in oggetto. Queste istruzioni modificano il contenuto del PC di un valore diverso da una unità.

Queste funzioni, compreso il salvataggio dell'indirizzo di ritorno nello Stack, sono del tutto automatici e trasparenti per l'utente.

Un particolare aspetto hanno le istruzioni di rientro (**`retlw, return, retfie`**): anche queste modificano il contenuto del ***PC***, ma lo fanno prelevando dal top dello Stack l'indirizzo completo a 13 bit per cui possono raggiungere qualsiasi posizione nella memoria programma.

Dal punto di vista grafico possiamo rappresentare così la situazione:



Stack (a 8 livelli) e PC (a 13 bit) sono al di fuori della mappa di memoria programma.

Essa, invece, va, nell'esempio, da 0000h (il vettore di reset) fino ad una estensione massima di 0FFFh, ovvero 4k (equivalenti a 2 pagine da 2k ciascuna).
Ovviamente si potranno avere chip con una, due o quattro pagine a seconda del modello.

Osserviamo che esistono solo due locazioni la cui funzione è prefissata, una per il vettore di reset e una per quello di interrupt.

Come per i banchi, l'ampiezza dell'istruzione di 14 bit limita lo spazio indirizzabile nella RAM, così, anche nel caso delle istruzioni che muovono il PC, la loro ampiezza limita l'area indirizzabile a soli 2k.

Questo giustifica la divisione della memoria programma in pagine da 2k ciascuna; all'interno di queste, l'indirizzo fornito con l'opcode sarà sufficiente.

Come passare da una pagina ad un'altra?

Se consideriamo una istruzione come **goto target** : la sua azione è quella di sostituire il contenuto del PC (che punterebbe alla linea successiva) con l'indirizzo **target**.
I 14bit dell'ampiezza dell'opcode sono nella forma:

10 1kkk kkkk kkkk

dove **101** è il codice dell'operazione **goto** e gli 11 bit rimanenti **k** sono l'indirizzo della memoria programma da raggiungere.
Quindi, 2^{11}=2048 sono le locazioni raggiungibili;ovvero, delle 8192 locazioni possibili, solamente le prime 2048 saranno indirizzate direttamente attraverso la codifica diretta dell'istruzione.
Consideriamo un caso qualunque, dove abbiamo definito:

```
target   equ 0x02FF
target1  equ 0x12FF
```

ovvero, in forma binaria:

Locazione	Indirizzo
target	0 0010 1111 1111
target1	1 0010 1111 1111

Si tratta di indirizzi a 13 bit (ampiezza massima dello stack e del bus relativo). In azzurro sono evidenziati gli 11 bit fornibili dall'opocode, che sono identici: la differenza tra i due è costituita dai successivi bit 12 e 13.
Ma, se utilizziamo queste label per un salto non condizionato:

```
goto target      ; indirizzo nei primi 2k
goto target1     ; indirizzo negli ultimi 2k
```

entrambe le righe rimanderanno il ***PC*** alla locazione **02FFh**, dato che nell'opcode possono essere contenuti solamente i primi 11 bit (bassi) della destinazione:

Locazione	Codifica
goto target	10 1010 1111 1111
goto target1	10 1010 1111 1111

Evidenziate in rosso le cifre di codifica del **goto** e in azzurro i bit dell'indirizzo che l'opcode può contenere.
Più piccolo è il numero di bit del core, meno ce ne saranno disponibili per l'indirizzo: nei core a 12 e 14 bit non ce ne sono abbastanza per poter indirizzare locazioni oltre una certa ampiezza, che è denominata pagina.

Famiglia	Core	Dimensione pagina
Baseline	12-bit	512
Midrange	14-bit	2048
PIC18F	16-bit	8192

Esattamente come per i banchi, anche nel caso della memoria programma, la limitata estensione

dell'opcode impedisce di fornire indirizzi che coprano oltre 2k; per superare questo limite occorre utilizzare un meccanismo che fornisca i due bit necessari per l'area da 8k.

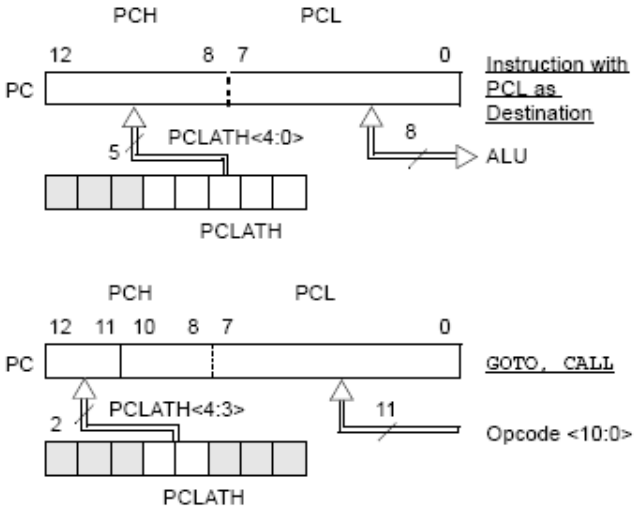
Soffermiamoci ancora un momento sul ***Program Counter***, che è la chiave del problema delle pagine e vediamo come è articolato nei Midrange.

Il ***Program Counter (PC)***, di per se, non è mappato in memoria e non è direttamente raggiungibile, ma dispone di una organizzazione che ne permette comunque un certo controllo.
E' composto praticamente da due registri:

- il primo è il **`PCL`**. Questo registro è mappato in memoria RAM e fa parte degli SFR.
  E', quindi, accessibile in lettura e scrittura e contiene i primi 8 bit bassi del Program Counter.
  E' allocato in modo da poter essere accessibile da qualsiasi banco (mirrored); nella pratica si trova all'indirizzo 0x02, 0x82, 0x102 e 0x182 dell'area RAM, anche se noi lo utilizzeremo esclusivamente attraverso la sua label, che è appunto **`PCL`**.
- I 5 bit più alti del ***PC*** non sono accessibili direttamente, in quanto non fanno parte della mappa di memoria. Possono, però, essere scritti indirettamente attraverso il registro **`PCLATH`**. Anche questo è un SFR nell' area RAM con le stesse caratteristiche di accesso del precedente.

Va osservato che il **`PCLATH`** non il byte alto di del PC, ma un elemento esterno che viene sommato al PC.

Da un punto di vista grafico possiamo rappresentare la situazione così:



L'indirizzo reale del salto o della chiamata è formato artificialmente dalla composizione di due elementi: gli 8 bit della parte bassa del PC, denominati **`PCL`**, a cui si aggiungono i primi 5 bit del registro **`PCLATH`**.

All' automatismo di incremento del PC, **solo ed esclusivamente per queste istruzioni**, si sostituisce la combinazione **`PCLATH+PCL`**.

Nell' esempio in figura, gli 11 bit dell' indirizzo di destinazione sono forniti dall'opcode (core a 14 bit) ed i rimanenti sono derivati dai bit 4:3 del registro **`PCLATH`**.

Ne risulta che lo switch per la manipolazione delle pagine è costituito dai bit **`PCLATH<4:3>`**.

E' della massima importanza sapere che il contenuto del **`PCLATH`** viene modificato solo dal reset o POR e/o da una scrittura diretta da parte del programma e che, una volta scritto con un determinato valore, questo permane fino a che non viene riscritto da una istruzione o cancellato dal reset.

Può non essere immediato da comprendere, ma proviamo a riassumere:

- a seguito di una istruzione che non sia **`goto`** o **`call`**, il PC viene incrementato automaticamente di una unità, puntando alla locazione successiva

Ad esempio, osserviamo una sequenza di istruzioni posta a fine pagina 0:

Istruzione	PC	Pagina
`movlw data`	0FFC	; pagina 0
`movwf target`	0FFD	
`movwf target1`	0FFE	
`addwf limit,w`	0FFF	
`nop`	1000	; passaggio in pagina 1
`movwf target2`	1001	

Il PC punta all'indirizzo della prima istruzione (0FFC).
L'esecuzione incrementa il PC a 0FFD e così via, per l' esecuzione delle istruzioni successive.
Quando il PC arriva a 0FFF, l'esecuzione dell' istruzione relativa lo incrementa a 1000: il passaggio alla pagina 1 è del tutto automatico.

PC incrementa da 0 al massimo numero contenibile (il che copre per intero gli 8k massimi disponibili) e, nel caso di istruzioni diverse da quelle di salto, non coinvolge in alcun modo altri registri.
Una volta raggiunto il massimo indirizzo disponibile, si ha un azzeramento del contatore del PC, con un ritorno all'indirizzo 0000h (wrap around).

Quanto sopra per quello che riguarda le istruzioni "ordinarie", tra cui sono compresi anche i ***test-&-skip*** come **btfss**, **btfsc**, **decfsz**, ecc. Nel caso di queste ultime, può, a seconda del risultato del test, non essere eseguita la linea immediatamente seguente, ma si passa a quella successiva (skip). Parrebbe che il PC debba essere aggiornato in modo diverso da quanto visto, ma, in pratica, non è così. Infatti, si tratta di istruzioni che impiegano due cicli nel caso in cui sia necessario lo skip, inserendo un **`nop`**; ne deriva che il PC viene incrementato sempre dallo stesso meccanismo rigidamente automatico di +1, ma due volte.

- Se, però, eseguiamo un **`goto`** o **`call`**, entra in azione il meccansimo del **`PCLATH`**, che viene in questo caso sommato al **`PCL`** per determinare l'indirizzo complessivo di destinazione e superare quindi il limite di 2k della pagina.

Dunque, se i bit di switch dei banchi sono forniti da **`RP1:0`** dello **`STATUS`**, i bit di switch delle pagine sono forniti attraverso la manipolazione del registro **`PCLATH`** e, in particolare, dai suoi bit 3 e 4, **`PCLATH <4: 3>`** bit, secondo questa logica:

PCLATH<4:3>	Pagina	Indirizzi
00	0	0000h - 07FFh
01	1	0800h - 0FFFh
10	2	1000h - 17FFh
11	3	1800h - 1FFFh

Quindi, per poter indirizzare correttamente un salto o una chiamata a subroutine sarà necessario prima dell'esecuzione di queste istruzioni, aver manipolato gli switches delle pagine in modo da puntare quella giusta.

Come per i banchi, se il programma supera l'ampiezza della pagina, occorre tenere sotto stretto controllo la situazione delle pagine.
Possiamo schematizzare le situazioni diverse che si creano a seconda dell'istruzione eseguita; ecco le 4 possibili situazioni:

1. La situazione 1 mostra come il PC è modificato da una scrittura del PCL **(PCLATH<4: 0> -> PCH)**. 8 bit sono forniti dall'ALU che ha decodificato l'istruzione e i rimanenti derivano dal PCLATH.
2. La situazione 2 mostra come il PC viene caricato durante una istruzione GOTO **(PCLATH<4:3> -> PCH).** Qui l'istruzione fornisce 11 bit e solo due dipendono dal PCLATH
3. La situazione 3 mostra come il PC viene caricato durante una CALL **(PCLATH <4: 3> -> PCH).** anche qui l'istruzione fornisce 11 bit e solo due dipendono dal PCLATH, ma, in aggiunta, il PC è caricato nella parte superiore dello Stack (TOS - Top of Stack).
4. La situazione 4 mostra l'azione delle istruzioni di ritorno, che prelevano dallo stack l'intero indirizzo

Ancora un esempio, per un **goto**:

- l'istruzione **goto label** porta con se 11 bit bassi della locazione **label**
- a questi si sommano come bit 12:11 i bit 4:3 del registro PCLATH
- il salto ha destinazione l'indirizzo risultante dalla somma, indirizzo che viene forzato nel PC

Istruzione	PC	Pagina
`movlw data`	`004C`	`; pagina 0`
`goto label`	`004D`	`; PC=PCLATH+label`
label = 0x100 PCLATH<4:3>=00 PC = 00000100		
`label ....`	`0100`	

Questo consente di saltare senza modificare il **PCLATH** all' interno della stessa pagina.

Istruzione	PC	Pagina
`movlw data`	`004C`	`; pagina 0`
`goto label`	`004D`	`; PC=PCLATH+label`
label = 0x100 PCLATH>4:3>=00 PC = 00000100		
`label goto label2`	`0100`	
label = 0x15A PCLATH<4:3>=00 PC = 0000015A		
`label2 ....`	`015A`	

Supponiamo, però, che il **PCLATH**, dal reset, non sia mai stato modificato; il suo contenuto è 0. Vogliamo saltare alla locazione **104Ch**; ecco cosa succede:

Istruzione	PC	Pagina
movlw data	004C	; pagina 0
goto label	004D	; PC=PCLATH+label
label = 0x104C PCLATH<4:3>=00 PC = 00004C		
movlw data	004C	

In questo caso il programma è bloccato in un loop chiuso!

Se l'indirizzo in oggetto supera la dimensione di 11 bit, come in questo caso (in giallo), quelli in eccesso NON possono essere contenuti dall'opcode (evidenziati in azzurro):

Bit	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Indirizzo	0	1	0				4				C			
opcode	0	1	0	0	0	0	0	1	0	0	1	1	0	0
PCLATH	0	0												
Indirizzo reale	0	0	0	0	0	0	0	1	0	0	1	1	0	0
	0		0				4				C			

Dato che il **PCLATH** contiene 0 e l'opcode non può contenere più di 11 bit, l'indirizzo realmente raggiunto dal salto non sarà 104Ch, ma 004Ch. Quindi:

- fino a che il programma si svolge senza salti o chiamate, il contatore che guida il succedersi delle istruzioni è automatico (PC) e procede su tutta l' ampiezza della memoria programma
- se il programma contiene salti o chiamate, queste sono indirizzate correttamente, ma solo all' interno della pagina corrente

Per poter raggiungere l'indirizzo voluto, sarà necessario modificare il **PCLATH**:

Istruzione	PC	Pagina
movlw data	004C	; pagina 0
pagesel label	004D	; aggiorna PCLATH
goto label	004E	; PC=PCLATH+label
label = 0x104C PCLATH<4:3>=01 PC = 0104C		
label	104C	

Ora, la somma del **PCLATH** + l'indirizzo fornito dall'opcode conduce il salto alla giusta destinazione:

Bit	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Indirizzo	0	1	0				4				C			
opcode	0	1	0	0	0	0	0	1	0	0	1	1	0	0
PCLATH	0	1												
Indirizzo reale	0	1	0	0	0	0	0	1	0	0	1	1	0	0
	1		0				4				C			

Da notare che:

mowf PCL,W

ha come risultato che il **PCL** viene copiato in W e che, contemporaneamente il **PCH** viene copiato in **PCLATH**.

PAGESEL.

Osserviamo che negli esempi abbiamo escluso a priori una manovra diretta su bit del **PCLATH** per sostituirla con uno pseudo opcode fornito da MPASM: **pagesel**

Come **banksel** per i banchi, **pagesel** modifica i bit necessari del **PCLATH** in relazione alla label che ha per oggetto.

L' uso di **pagesel** è importante dato che, se in generale si esemplifica la movimentazione del PCLATH con i suoi bit 3 e 4, nella forma classica:

```
; passaggio in pagina 1
     bsf   PCLATH,3
     bcf   PCLATH,4
```

pagesel viene compilato con i giusti valori in relazione alla label che ha come oggetto.

Questo permette di prescindere dalla conoscenza di dove sia posizionata la label, cosa comune nel caso di sorgenti rilocabili.

Un'altra ragione per impiegare lo pseudo opcode è che il sistema di bit **NON è valido per tutti i chip**. Così, per gli Enhanced Midrange, con un set di istruzioni esteso, **pagesel** è implementato con l'opcode **movlp**, che modifica il **PCLATH** con una unica istruzione.

Ultimo, ma non certo meno importante, evita l'impiego di valori assoluti che rende non portabile il codice.
Di conseguenza, va abbondato l'uso di modifica a bit singoli in favore del **pagesel**.

Modifichiamo il PCLATH.

Va ricordato come fatto essenziale che PCLATH non viene mai modificato automaticamente se non dal POR o dai reset.

Il suo valore iniziale al reset è 0, in modo da indirizzare la pagina 0 dove si trova il vettore di reset, Poi, **si potrà variarne il valore solamente scrivendolo da programma** e, u**na volta modificato, il suo valore resterà invariato fino ad una nuova modifica o ad un reset** (dove viene riportato a 0).

Dunque, è essenziale, come mostrato dall'inizio, distinguere i meccanismi automatici di incremento del PC dalla modifica volontaria del **PCL+PCLATH**.

Osserviamo le seguenti istruzioni:

```
 0045          movlw   data
 0046          pagesel RdData    ; punta pagina della routine
 0047          call    RdData    ; salto alla subroutine
 0048          movwf   target    ; rientro ed esecuzione linea
 0049          goto    WrData    ; salto nella stessa pagina
...
 0200 WrData   movwf   Txbuf     ; indirizzo in pagina 0
...
...
 1000 RdData   movwf  buff1      ; routine in pagina 1
....
```

All'atto pratico, `call RdData` verrà eseguita correttamente, dato che abbiamo puntato la pagina 1 con `pagesel RdData`, ma la successiva `goto WrData` fallirà completamente, inviando l' esecuzione all' indirizzo 1200h!

Questo accade perchè abbiamo ancora il PCLATH che punta in pagina 1: dato che non lo abbiamo modificato, qualsiasi salto si troverà con l'indirizzo formato da PCL + il PCLATH attuale ! Occorre una correzione, ovvero riportare il PCLATH alla pagina della routine chiamata:

```
 0045              movlw   data
 0046              pagesel RdData     ; punta pagina della routine
 0047              call    RdData     ; esegue la routine
 0048              movwf   target     ; rientro ed esecuzione linea
 0049              pagesel WrData     ; posiziona il PCLATH per il goto
 004A              goto    WrData     ; salta
...
 0200   WrData     movwf   Txbuf      ; indirizzo in pagina 0
...
...
 1000   RdData     movwf   buff1      ; routine in pagina 1
....
```

Poichè **WrData** si trova nella stessa pagina di esecuzione, avremmo potuto scrivere anche:

```
 0049            pagesel $       ; PCLATH alla pagina corrente
 004A            goto    WrData  ; salto
...
 0200   WrData   movwf   Txbuf   ; indirizzo in pagina 0
```

con il medesimo effetto: **pagesel $** dice semplicemente al **PCLATH** di posizionarsi sulla pagina corrente.

Tutto quanto finora detto per il **goto**, vale ugualmente per il **call** di una subroutine.

Ricordiamo che l'azione sul PC è la medesima: entrambe le istruzioni deviano l'esecuzione ad una diversa area di memoria programma.

La differenza consiste nel fatto che, per il **goto**, la deviazione è definitiva.
Per **call**, invece, al salto corrisponde il salvataggio nel TOS dell'indirizzo seguente la chiamata (indirizzo di ritorno); **call** deve essere completata da una istruzione di rientro (**retlw** o **return**) che preleva dallo stack l'indirizzo completo a 13 bit per il rientro.
Per il resto, anche la codifica di **call** può supportare solo 11 bit di indirizzo e richiede la manipolazione del **PCLATH** per superare questo limite.

```
 0045            movlw   data
 0046            pagesel RdData      ; punta pagina della routine
 0047            call    RdData      ; esegue la routine
 0048            movwf   target      ; rientro ed esecuzione linea
 0049            pagesel $           ; posiziona il PCLATH alla pagina
corrente
 004A            call    WrData      ; chiama la routine
...
 0200   WrData   movwf   Txbuf       ; in pagina 0
...
...
 1000   RdData   movwf   buff1       ; routine in pagina 1
....
```

In mancanza del **pagesel $** , il **call WrData** si troverebbe a puntare a 1200h.

Va posta attenzione alla differenza tra il ritorno da una subroutine e la necessità del **pagesel $**

Ricordiamo ancora una volta che le istruzioni di ritorno non necessitano di manipolazione del **PCLATH** dato che caricano il PC con l'intero indirizzo prelevato dallo Stack.
Questo garantisce il ritorno all'istruzione successiva alla chiamata senza il bisogno di altre azioni.

Però, rientrati al flusso principale, il ***PC*** è modificato, ma il **PCLATH** no: resta identico a quanto è stato scritto nell'ultima modifica.

La cosa può parere intricata, ma basta riflettere un istante; ad esempio:

- siamo in pagina 0 dobbiamo chiamare una routine in pagina 3: portiamo i bit **PCLATH<4:3>** a 11
- chiamiamo la subroutine: l'indirizzo dell'opcode si sommerà al PCLATH, determinando il giusto indirizzo nella corretta pagina

- al termine della routine, l'istruzione di rientro modifica il PC, ma lascia invariato il **PCLATH**: torniamo al flusso principale, ma sempre con il **PCLATH** che punta alla pagina 3

Se dobbiamo chiamare o saltare in altre pagine, occorrerà modificare il **PCLATH**.

Nel problema del paging e della gestione del **PCLATH** vengono coinvolte anche le tabelle retlw o goto che sono chiamate con **call** o **goto**..

Quando non è necessario tenere conto del paging?

Essenzialmente, quando utilizziamo un chip dotato di una sola pagina di memoria.

Questo non toglie, però, che si debba comunque essere a conoscenza del problema ed è una ragione per evitare l'uso di chip minimali (e obsoleti) come il troppo famoso 16F84. Avendo una sola pagina di memoria programma, non consente all'utente di comprendere il problema: quando si troverà davanti a programmi che superano la pagina e chip con più pagine, avrà serie difficoltà.

Inoltre, possiamo non considerare il problema delle pagine quando siamo certi che il nostro programma ha una ampiezza tale da essere contenuto nella pagina 0.

Nelle esercitazioni facciamo, però, uso degli switch di pagina anche quando essi chiaramente non servono, date le dimensioni minimali dei programmi, ad esempio forzando il posizionamento di subroutines in pagine diverse dalla 0.

Per contro, si dovrà fare uso dello switch di pagina quando si utilizza una compilazione modulare/rilocabile, dato che , a priori, non possiamo dire dove il Linker posizionerà i vari elementi.

Strategie di paging.

Da quanto detto, è evidente che il programmatore di PIC che non adotti una qualche strategia per superare il problema del paging, si troverà a dover usare un buona parte della sua attenzione per evitare errori nel trattamento di salti e chiamate.

E' evidente che, così come un errore di banco indirizza un registro non desiderato, un errore nella paginazione manda in esecuzione parti di programma non dovute, con il risultato di far spendere un tempo, anche molto consistente, per individuare la causa.

Consigli pragmatici, per evitare questo, possono essere:

- Dato che il problema del paging ha importanza critica per l' Assembly, per programmi che si svolgono su più pagine una possibile via di uscita è quella di usare linguaggi superiori, nei quali sono i compilatori a farsi carico del problema. Questa soluzione, però, non comporta una maggiore efficienza del codice, nè riduce le dimensioni (anzi, è più vero il contrario, sopratutto con basse ottimizzazioni).

- Un'altra soluzione è quella di adottare i PIC18F, che sono esenti dal paging e, per quanto riguarda i registri SFR, anche dal banking, che resta necessario solo per la RAM dati.
  L'unico limite può essere dato dal fatto che, attualmente, non ci sono PIC18F in package più piccoli dei 18 pin ed il loro costo è leggermente superiore a quello dei Midrange.
  In ogni caso, questa è la soluzione che consigliamo a chi vuole realizzare programma Assembly di una certa dimensione senza impazzire sul paging e minimizzare il banking, che, purtroppo, sono due punti critici e cordialmnte antipatici nei PIC e ne riducono l'efficienza e la maneggiabilità.

Però, usando Assembly per i Midrange si possono prospettare alcune soluzioni meno drastiche.
La principale è la più semplice, e praticamente unica, nel caso di rilocabili: usare massicciamente **`pagesel`** e **`banksel`**.

L'aggiunta delle istruzioni di cambio pagine e banchi, anche se appesantiscono il codice, lo fanno per micro o nano secondi, il che non comporta in generale alcuna inefficienza del codice stesso.
Per contro, garantiscono la sicurezza di aver sempre indirizzato pagine e banchi corretti.
Questa è la via più rigorosa: la compilazione di una serie di files rilocabili non permette sempre di conoscere a priori dove saranno allocate le varie sezioni; di conseguenza, se il programma supera la pagina, occorre aver introdotto quanto necessario per garantire un paging funzionale.

Questa è la soluzione che useremo nelle esercitazioni.

A queste considerazioni possiamo aggiungerne altre di carattere generale:

1. nella scrittura di moduli rilocabili, è opportuno che il modulo non travalichi il limite della pagina. Questo si può ottenere con la creazione di sezioni di linker specifiche per ogni pagina.
   La direttiva **`CODE`** dice a MPASM per iniziare una nuova sezione di programma; la direttiva **`UDATA`** inizia una nuova sezione di dati nel linker. Quando si esegue il linker, si crea un file .map in cui è riportato dove il compilatore ha allocato ogni sezione del programma, compreso il banco per la RAM dati.
2. mantenere il **`PCLATH`** sulla pagina corrente. Questo consente a chiamate e salti all'interno di un modulo di non richiedere la manipolazione del registro. Esso va sempre impostato prima della chiamata o del salto e riportato alla pagina corrente al ritorno dalla chiamata
3. Possiamo anche ricorrere a strutture in cui, ad esempio, le subroutine sono tutte raccolte in una pagina e così pure le tabelle. Questo consente a priori di sapere dove esse stanno e usare il paging solo dove necessario. Si tratta, però di situazioni più laboriose e più a rischio. Solamente nel raro caso di strutture fortemente critiche sui tempi di esecuzione sarà necessaria una cura specifica nell'allocare i vari componenti dell'azione in una medesima pagina per evitare la manipolazione di **`PCLATH`**.

Esistono anche altre possibilità, attraverso l'impiego di macro specifiche. Nel WEB sono recuperabili diverse pagine a riguardo.
La funzione di queste macro è quello di far stabilire dall'Assembler, durante la compilazione, la necessità o meno di inserire le manipolazioni di **`PCLATH`**. Lo scopo è quello di migliorare la prestazione, rendendo nel contempo più facile da leggere i sorgenti.
Una forma classica e ben nota sono le macro **fcall**, **pcall**, ecc, che troviamo descritte a http://www.piclist.com/techref/microchip/pages.htm.

Un problema di queste strutture è quello di non poter con facilità creare moduli rilocabili, il che è, invece, necessario quando il programma assume dimensioni non minimali.

LCALL e LGOTO

Il MacroAssembler MPASM mete a disposizione due macro per il trattamento delle chiamate e dei salti al di fuori della pagina. La prima è **lcall** (long call):

```
lcall label
```

equivale a :

```
pagesel label
call    label
```

Occorre, dove necessario, ripristinare “manualmente” il **PCLATH**:

```
pagesel label
call    label
pagesel $
```

Esiste la stessa cosa per il salto: **lgoto** (long goto) che è sempre macro. Quindi:

```
lgoto label
```

equivale a :

```
pagesel label
goto    label
```

Ugualmente,dove necessario, occorre ripristinare “manualmente” il **PCLATH**.

RETLW Table e paging.

Nei capitoli precedenti abbiamo dettagliato il problema del paging per quanto riguarda le conseguenze delle due istruzioni che generano salti: **goto** e **call**.

Qui risulta indispensabile modificare manualmente il ***PC*** (in effetti, il **PCLATH**) per arrivare alla giusta destinazione indirizzata da queste istruzioni. E' abbastanza semplice comprendere che la limitazione dell'ampiezza della pagina, dipendente dalla lunghezza in bit degli opcodes, rende necessaria una correzione esterna quando si supera la pagina corrente.
Ricordiamo ancora che il ***PC***, nei Midrange, è di 13bit, ma che l'istruzione comprendono solo 11bit di indirizzo: i due mancanti sono forniti da **PCLATH<4:3>** e sono gestibili con semplicità dal **pagesel**.

Non parrebbe esserci altro motivo di manipolare il **PC**, anche se **PCL** e **PCLATH** , essendo mappati in memoria, sono accessibili alle normali istruzioni che trattano l'area RAM (**movwf**, **movf**, **andwf**, **addwf**, ecc).

E, in effetti, la manipolazione del ***PC*** è una azione che richiede una precisa valutazione e, all'atto pratico, non ha molte ragioni per essere effettuata, al di là dell'aggiustamento della pagina nei salti. Ricordiamo anche che i Midrange non hanno istruzioni per trattare lo stack, né i meccanismi di "sorveglianza" nel caso di overflow o underflow che hanno, ad esempio, i **PIC18F**.
Quindi, correntemente, non sembra esserci ragione di agire su **PCL** e **PCLATH** al di fuori dei casi indicati.

Però, c'è una situazione che è comune e che nasconde un tranello micidiale. Si tratta di questo: nei Midrange, per accedere ai dati nella memoria programma, è necessario eseguire la lettura di una **tabella retlw**.

La tabella tipica consiste in una serie di istruzioni **retlw K** dove K è il valore numerico a 8 bit che si vuole in uscita dalla tabella.
La prima istruzione di ingresso nella tabella calcola l'offset utilizzando **addwf PCL,f** , dato che l'indice della tabella è solitamente in WREG, ed è seguita dalle righe **retlw**:

```
Table:
     addwf  PCL,f          ; punta PC
     retlw  ...
```

addwf PCL,f aggiunge l'indice al **PCL** e la modifica del **PC** fa si che l'istruzione successiva da eseguire sia la riga **retlw** puntata dall'indice.

Come sappiamo, il PC nei Midrange è ampio 13bit: gli 8 bit bassi (**PCL**) sono mappati in RAM e sono direttamente leggibili e scrivibili. I restanti 5 bit alti non sono accessibili direttamente e possono solo essere scritti attraverso il registro **PCLATH**, che è R/W (solo i primi 5 bit (bit 4:0) di questo registro sono implementati).

L'accesso alla tabella si effettua con l'istruzione **call**., in modo che la riga **retlw** indirizzata provochi il rientro al punto del programma successivo alla chiamata.

Se la tabella si trova in una pagina diversa da quella chiamante, l'uso del **pagesel** permette di raggiungerla correttamente:

```
pagesel  Table
call     Table
```

Parrebbe che tutto sia a posto, ma c'è un problema.

Se la chiamata alla tabella e la tabella stessa stiano nella medesima pagina, non serve neppure il **pagesel**.. Nei Midrange la pagina è ampia 2048 istruzioni (4 volte quella dei Baseline) e programmi che superano questa ampiezza non sono di immediata scrittura.

Supponiamo che la situazione sia questa:

```
  PC         istruzione
0059      movlw  3        ; indice per tabella
005A      call  table     ; chiama la tabella
005B      .....
.....
0150      Table:
0150      addwf  PCL,f    ; PCL=PCL+W
0151      retlw  K1
0152      retlw  K2
0153      retlw  K3
0154      ....
```

Carichiamo in W l'indice della riga della tabella che ci interessa raggiungere. Chiamiamo la tabella come subroutine.

Nella tabella, sommiamo al **PCL** l'indice, in modo da non portarsi all'istruzione seguente, ma a quella puntata dall'indice.

Così, se **W=3**, abbiamo al momento della somma **PCL=0x150** e dopo la somma **PCL=0x150+3=0x153**.

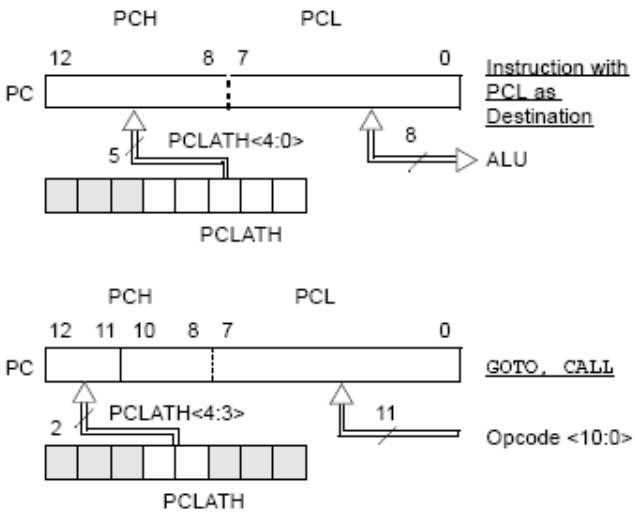
Quindi il passo successivo eseguirà non la riga alla locazione **0x151**, ma quella alla **0x153**.

Se compiliamo e proviamo l'esempio, otteniamo che, dopo la modifica del **PCL**, l'istruzione eseguita non sarà quella nella posizione **0x153**, ma quella alla locazione **0x53** !!

Il **PC** porterà ad un salto in una locazione che nulla ha a che fare con la tabella che stiamo interrogando e il flusso logico del programma sarà completamente alterato con le evidenti conseguenze.

Questa situazione è dovuta ad una particolarità poco evidente: si rende necessario osservare che qualsiasi istruzione, diversa da un salto (**goto** e **call**) e che modifica il registro **PCL**, cancella anche il 9° bit (bit 8) del ***PC***.

Questo è dovuto alla struttura intrinseca dei Midrange e che possiamo rivedere nello schema che abbiamo già riportato e al quale diamo ora una lettura più approfondita.



Qualsiasi istruzione che modifica **PCL** inserisce i bit **4:0** di **PCLATH** come bit **12:8** di **PCH**.

In questo caso occorre modificare manualmente il **PCLATH**.

Mentre nel caso di **call** e **goto**, invece, sono solo i bit **4:3** di **PCLATH** a sostituire i bit **12:11** di **PCH**.

I bit **10:0** sono portati direttamente nella codifica dell'istruzione che permette di indirizzare una pagina da **2048** locazioni (**2^11 = 2048**).

Se la destinazione è al di fuori della pagina, l'operazione di adattamento del **PCLATH** è eseguita dal **pagesel**.
Occorre anche ricordare che il registro **PCLATH** è azzerato al reset e che viene modificato soloo ed esclusivamente con istruzioni; inoltre, una volta scritto con un valore, questo non cambia fino ad una successiva riscrittura o ad un reset .

Così, se chiamiamo una **retlw table** che sia pure posizionata nella pagina dell'istruzione chiamante, ma oltre le prime 2^8 locazioni, l'istruzione **addwf PCL,f** , che punta la riga della tabella da eseguire, copia **PCLATH4:0** nel **PCH** col risultato che, invece della riga **retlw** stabilita dall'indice, punta ad una locazione errata!

Nell'esempio numerico riportato sopra, se il **PCLATH** non è stato scritto dopo il reset, il suo contenuto è 0, abbiamo che **PCL**, a 8bit, non contiene **0x153**, ma solo gli 8bit bassi, ovvero **0x53**.
L'indirizzo complessivo del prossimo passo del ***PC***, dato da **PCLATH+PCL**, sarà

PCLATH	PCL
0	53
00000	01010011

Quindi si eseguirà non la 0x153, ma l'istruzione a 0x53.

Se avessimo la tabella in pagina 1, il **pagesel** della chiamata avrebbe modificato **PCLATH4:3**

```
 PC          istruzione
0059       movlw   3          ; indice per tabella
005A       pagesel table      ; PCLATH4:3 = 01  PCLATH4:0 = 01000
           call    table      ; chiama la tabella
.....
; pagina 1 (0x0800-0x0FFF)
0950       Table:
0950       addwf   PCL,f    ; PCL=PCL+W
0951       retlw   K1
0952       retlw   K2
0953       retlw   K3
```

Il **PCLATH** è stato modificato per la prima volta dopo il reset e ora contiene 01000; abbiamo sempre che **PCL**, a 8bit, non contiene **0x953**, ma solo gli 8bit bassi, ovvero **0x53**.
L'indirizzo complessivo del prossimo passo del ***PC***, dato da **PCLATH+PCL**, sarà

PCLATH	PCL
8	53
01000	01010011

L'istruzione in esecuzione si troverà a 0x0853 invece che a 0x0953

Nel caso in cui il **PCLATH** fosse stato modificato dopo il reset scrivendo anche i bit **PCLATH2:0**, gli indirizzi di destinazione dei salti saranno dipendenti da queste scritture.
Infatti, ricordiamo, il contenuto del **PCLATH** viene modificato solo con istruzioni o azzerato al reset.

Se, nell'esempio qui sopra, la tabella fosse posta all'indirizzo **0x850**, la situazione diventerebbe questa:

PCLATH	PCL
8	53
01000	01010011

L'istruzione in esecuzione si troverà a 0x0853 come previsto!

Altrettanto, nel primo esempio, se la tabella iniziasse a, ad esempio **0xA0**, si avrebbe:

PCLATH	PCL
0	A3
00000	10100011

L'istruzione in esecuzione si troverà a 0x00A3 come previsto!

Ovvero:

La tabella, così come scritta, funziona correttamente solo se è posizionata entro le prime 256 (FFh) locazioni della pagina.

Se la tabella è posizionata diversamente, occorre intervenire.

Una prima soluzione è piazzare le tabelle all'interno dello spazio limitato. La cosa è possibile solamente dove si ha a che fare con poche e brevi tabelle ed è comunque una restrizione rischiosa.

Una seconda soluzione è quella di aggiungere istruzioni per modificare il **PCLATH** in funzione del

posizionamento della tabella nella memoria programma.

Ad esempio:

```
Table:
    movwf   index           ; salva indice in W
    movlw   HIGH(TblStrt)   ; indirizzo alto della tabella
    movwf   PCLATH          ; al PCLATH
    movf    index,w         ; recupera indice della tabella
    addlw   TblStrt         ; sommato all'indirizzo basso
    skpnc                   ; se non c'è carry, salta
     incf   PCLATH,f        ; altrimenti aggiusta PCLATH
    movwf   PCL             ; copia in PCL
TblStrt:
    retlw   ...
```

Questa manipolazione del **PCLATH** permette di piazzare la tabella in qualsiasi punto della memoria programma.

NOTE RIASSUNTIVE.

Nei PIC, l'intero tema dell'indirizzamento nella memoria programma è piuttosto contorto.
Ci sono alcuni casi da considerare in modo particolare:

- La memoria programma nei Midrange può essere ampia fino a un massimo di 8192word (indirizzi da 0x0000 a 0x1FFF), pari a 4 pagine da 2048word ciascuna, anche se la maggior parte dei dispositivi ne contengono di meno.

- Questo spazio è gestito da un bus indirizzi a 13 bit ed è qui che cominciano i problemi.
  Il limite è dato dal fatto che la lunghezza di un opcode (14bit) non può ospitare 13bit di indirizzo, dato che deve contenere anche il codice dell'istruzione.
  Ed è ancora più difficile ricavare 13 bit di informazioni di indirizzo dai registri di calcolo della ALU che sono a 8bit.

- Per contro, lo stack (che è completamente inaccessibile al programmatore dei Midrange) è di 13bit, cosicchè **`return`**, **`retlw`** e **`retfie`** possono prelevare l'intero indirizzo di ritorno.

- Ma **`call`** e **`goto`** portano nella codifica solo un indirizzo di destinazione parziale a 11bit dei 13 bit necessari per raggiungere qualsiasi punto della memoria programma da 8192word.
  I 2 bit di indirizzo più significativi sono presi dal registro **`PCLATH<4:3>`**, che deve già fare riferimento alla pagina (2048word) corretta quando una di queste istruzioni viene eseguita.
  In caso contrario il salto andrà ad una destinazione errata.

- Ogni altra istruzione che modifica **`PCL`** , modifica solo gli 8bit di questo registro, ovvero gli 8bit bassi dell'indirizzo a 13bit.. I restanti 5 bit sono presi da **PCLATH<4:0>**.
  Quando realizzano tabelle di look-up, **`PCLATH`** deve fare riferimento alla corretta area da 256 parole dove si trovano i **`retlw`**, prima di modificare PCL.

- Negli schemi visti prima, va notato che:
  - **`pagesel`** agisce solo modificando i bit **`PCLATH<4:3>`** e lasciando immutati gli altri
  - mentre ogni istruzione che sia diretta a modificare **`PCL`** non modifica il **`PCLATH`** e questo può comportare un ***PC*** non corretto.

Alcune conseguenze pratiche di cui sopra comprendono:

- **`PCLATH`** non ha alcun effetto sul "normale" flusso del programma da un'istruzione a quello successivo. Quindi, solitamente, non viene preso in considerazioni.
  L'avanzamento del ***PC*** da una istruzione alla successiva dipende dagli automatismi della CPU e non comporta alcuna modifica di **`PCLATH`**: questo registro viene cambiato solo dal reset, che lo azzera, o da una istruzione esplicita che lo scriva.

- In chip con meno di 2K (2048 – una pagina) di memoria di programma o in chip con più pagine, ma di cui ne sia utilizzata una sola, **`call`** e **`goto`** funzionano sempre correttamente e non c'è alcuna preoccupazione per il paging.
  Questa situazione è comune a piccoli programmi hobbistici o di istruzione e tende a

nascondere la situazione che si viene a creare se si supera la pagina 0.

• In chip dove sono usate più pagine di memoria programma, la cosa più sicura, principalmente per compilazioni rilocabili, è inserire la direttiva **pagesel** prima delle istruzioni **call** e **goto**:

```
pagesel   target
call      target
pagesel   target1
goto      target1
```

Questo "aggiusta" automaticamente il **PCLATH<4:3>** e consente di ottenere il salto alla giusta destinazione, ovunque sia posta in memoria.
Però, da non scordare mai:
- il valore scritto nel **PCLATH**, una volta modificato dal **pagesel** (o dalla sequenza di istruzioni equivalenti) resta fissato fino alla prossima modifica
- il **pagesel** modifica solo **PCLATH<4:3>**, ma non tocca **PCLATH<2:0>**

• Quanto detto sopra comporta che **retfie**, che funziona su tutti i processori, risente di questo grave problema: se, per qualche ragione, avete modificato il **PCLATH** durante la routine dell'interrupt, un successivo **goto** o **call** , dopo il **retfie**, potrebbe non funzionare correttamente (ricordiamo ancora una volta che il contenuto di **PCLATH**, una volta scritto, si modifica solo con un'altra scrittura).
Per evitare questo problema, è opportuno salvare il registro **PCLATH** nella all'ingresso dell'interrupt e ripristinarlo all'uscita.

• Altrettanto per **return** e **retlw** : se **PCLATH** è stato modificato nella subroutine, il prossimo **goto** o **call** dopo il ritorno potrebbe non funzionare correttamente.
Per evitare questo problema, inserire una direttiva **pagesel** riferita alla pagina corrente dopo il ritorno dalla chiamata subroutine:

```
pagesel target
call    target
pagesel $
```

• Nel caso delle **tabelle retlw** occorre considerare che la modifica del **PCL** effettuata dal sommargli l'indice:

```
addwf   PCL,f
```

ha lo spiacevole effetto di azzerare il bit 0 di **PCLATH**, ovvero il nono bit dell'indirizzo della tabella. Se la tabella è al di fuori dei primi FF indirizzi della pagina, ne risulta che si finisce per puntare ad una locazione errata, con le relative disastrose conseguenze.
Occorrerà manipolare il **PCLATH** per avere il giusto schema di bit che indirizzano la locazione corretta, come visto sopra.

RIFERIMENTI.

- DS33023a Microchip
- AN556 Microchip