Esercitazioni PIC Midrange.

Inserto a: banking.

Vediamo alcune informazioni essenziali che riguardano il problema dei banchi nei PIC Midrange. Si tratta essenzialmente di macro predisposte e che agiscono principalmente attorno ai test sui flag dello **STATUS.**

Banchi (e pagine).

Vogliamo focalizzare l' attenzione su alcune delle principali particolarità della famiglia Midrange e di come essa si distanzi dai Baseline, fornendo maggiori risorse, ma anche qualche complicazione in più.

In primo luogo, dobbiamo obbligatoriamente introdurre i concetti di:

- **banco** per la memoria dati e gli SFR
- **pagina** per la memoria programma.

Se pure qualche cenno ne è stato fatto a riguardo dei Baseline, bisogna dire che, in generale, questi sono chip minimali in cui la quantità di memoria, sia programma che RAM, è abbastanza contenuta e spesso non supera la quantità che può essere contenuta in un banco o in una pagina; di conseguenza, il problema non si presenta nella maggioranza dei casi di programmi non troppo complessi.

Diversa è la tendenza dei Midrange, che iniziano a disporre di molta più memoria e periferiche e questo, a causa della struttura RISC/Harvard dei PIC, è fonte di qualche complessità.

Consideriamo qui i banchi.

Nella struttura RISC impiegata nei PIC, il codice dell'istruzione (opcode) contiene sia la codifica dell'istruzione vera e propria, sia l'indirizzo a cui è diretta o il valore da movimentare.

Per i Midrange, l'ampiezza di una istruzione è di 14 bit.

Per chiarire ulteriormente, possiamo dividere le istruzioni in alcuni gruppi, a seconda di cosa fanno. Abbiamo quindi:

- **istruzioni dirette ad azioni sull'area di memoria RAM**. Una istruzione tipica contienel'azione da effettuare e dove effettuarla. Ad es.
  `decf cntr, f` ha lo scopo di decrementare il contenuto della locazione di RAM (file) che corrisponde alla label `cntr.` I 14bit dell'ampiezza dell'opcode sono codificati nella forma:

  `00 0011 dfff ffff`

dove **000011** è il codice dell'operazione **decf**, **d** è la destinazione (in questo caso **f**, ovvero il file stesso) e i 7 bit rimanenti **f** sono l'indirizzo della locazione RAM su cui agire. Quindi, solo 7 bit possono essere usati per l'indirizzo, e $2^{\wedge 7}$=128 sono le locazioni nella memoria RAM raggiungibili direttamente dall'istruzione

- **istruzioni di movimentazione nell'area di memoria programma**. Si tratta di istruzioni che consentono salti o il ritorno da subroutine ed interruzioni. Ad es.:
  goto target sostituisce il contenuto del PC (che punterebbe alla linea successiva) con l'indirizzo target. I 14bit dell'ampiezza dell'opcode sono nella forma:

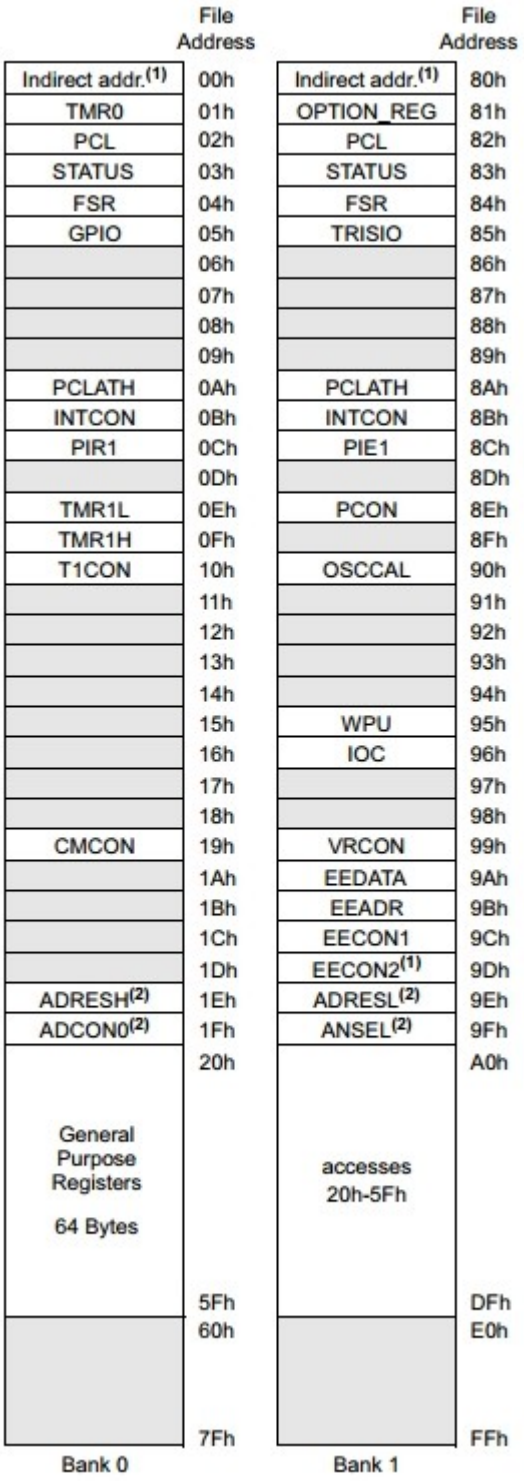
  10 1kkk kkkk kkkk

  dove **101** è il codice dell'operazione **goto** e gli 11 bit rimanenti **k** sono l'indirizzo della memoria programma da raggiungere. Quindi, $2^{\wedge 11}$=2048 sono le locazioni raggiungibili nella memoria programma.

Lasciamo da parte per ora i problemi relativi alle istruzioni di salto (**goto**, **call**) e ritorno (**return**, **retlw**, **retfie**) e concentriamoci su quelli relativi alla memoria RAM.
Dato che solo 7 bit dell'opcode possono contenere indirizzo, l'estensione massima di questo può coprire solo 128 locazioni (2^7=128); all'interno di questo limite, l'opcode fornisce tutte le informazioni necessarie per essere eseguito.

Però è comune nei Midrange avere una area RAM anche molto superiore a 128 bytes: in questo caso l'istruzione non è più in grado da sola di raggiungere tutta l'area e si richiede un meccanismo ausiliario di supporto.

Vediamo, ad esempio, la mappa della memoria RAM di un chip generico, ad esempio il 12F629:



1. - la quantità di registri obbliga a suddividere la memoria RAM in banchi. Qui ne abbiamo due: **banco 0** e **banco 1**. In altri chip ci potranno essere fino a 4 banchi.

2. - è possibile che alcune locazioni (in grigio nell' immagine) non siano implementate; questa è una caratteristica che troviamo in tutte le mappe di memoria ed è dovuta a ragioni di progetto. Per conoscenza, scrivendo qualcosa nelle aree "grigie" non si ha alcun risultato ed accedendo in lettura a queste aree, tipicamente si legge 0.

3. - alcuni registri di particolare importanza si trovano "mirrored" su tutte le pagine sono accessibili senza problemi, qualunque sia il banco in cui ci si trova. Quindi, ci troviamo con **`STATUS,FSR , PCL , PCLATH , FSR`** e **`TMR0`** che non richiedono di commutare il banco: questa situazione è presente in tutti i PIC ed è molto comoda, dato questi registri sono impiegati intensivamente nei programmi. L' accesso mirrored consente di evitare la perdita di tempo delle istruzione di commutazione dei banchi.

4.- oltre agli SFR, l'area RAM comprende anche la RAM dati vera e propria, che qui si estende per una ampiezza di 64 bytes ed è accessibile comunque da entrambi i banchi. Ovvero, scrivere all' indirizzo 20h o A0h è la medesima cosa. E quanto scritto in 5Fh sarà leggibile a DFh. Si parla di RAM condivisa (*shared RAM*).

In questi chip abbiamo anche un'area "grigia" di RAM dati, quindi non implementata, e che si comporta come indicato prima.

Notiamo che il primo banco (banco 0) va dall'indirizzo 00h a 7Fh, ovvero è ampio 128 bit; altrettanto il banco 1, che va da 80h a FFh.

Se il banco è ampio 128 bit, questa area può essere indirizzata interamente attraverso il codice dell'istruzione, ma questo nn è sufficiente a passare in aree adiacenti.
Come abbiamo osservato dalla tabella precedente, un byte in un banco ha come corrispondente un altro byte nel banco successivo.
L'indirizzo dei due, che sono contenuti in una area complessiva di 256 byte, si differenzia per il bit più alto dell'indirizzo, l'ottavo bit. Infatti $2^8=256$.

Così, in relazione ad un determinato registro, ad esempio **GPIO,** ci troviamo in questa situazione:

SFR	Banco	Indirizzo
GPIO	0	**05h 00000101b**
TRISIO	1	**85h 10000101b**

Sia **GPIO** che **TRISIO** hanno i primi 7 bit dell'indirizzo identici (5 = 00000101), ma si differenziano
per l'ottavo bit.
Siccome nell'opcode ne abbiamo solo 7, l'istruzione **clrf GPIO** e **clrf TRISIO** sono codificate in modo identico, con la stessa sequenza di cifre binarie:

Istruzione	Opcode
clrf GPIO	00 0001 1000 0101
clrf TRISIO	00 0001 1000 0101

Però, **GPIO** è in banco 0 (bit 8 = 0) e **TRISIO** è in banco 1 (bit 8 = 1).
Occorre che sia implementato un meccanismo che permette di gestire l'ottavo bit a seconda del banco che si vuole raggiungere.
Questo meccanismo è definito come switch di banco ed è costituito da uno (O due) bit presenti nello

STATUS — STATUS REGISTER (ADDRESS: 03h OR 83h)

Reserved	Reserved	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

Si tratta del bit **RP0** (e **RP1**) che nei Baseline era denominato **PA0 .**

Questo bit viene aggiunto automaticamente all'indirizzo contenuto nell'opcode e ne diventa l'ottavo bit. Nell'opcode possiamo identificare l'indirizzo relativo nel banco, mentre l'ottavo bit informa il processore di quale banco deve essere indirizzato.
L'ottavo bit sarà a 0 per gli indirizzi da 00h a 7Fh.

00h -> 00000000
7Fh -> 01111111

Il banco 1 comprende gli indirizzi da 80h a FFh. Anch'essi richiedono 7 bit di indirizzamento, ma a patto che il bit 8 sia a 1:

80h -> 10000000
FFh -> 11111111

Quindi il passaggio da un set di indirizzi all'altro si può effettuare variando il valore di un ottavo bit, che è lo switch dei banchi**.**
Dunque, agisco sullo switch di banco, modificandolo, prima di effettuare l'istruzione voluta. Ne consegue questa tabella:

RP0	Banco
0	0
1	1

Quindi, in istruzioni:

```
bcf   STATUS,RP0    ; bit ottavo = 0 punta al banco 0
clrf  GPIO          ; azzero i latch del port
```

e

```
bsf   STATUS,RP0    ; bit ottavo = 1 punta al banco 1
clrf  TRISIO        ; tutti i bit utili come uscite
```

Queste istruzioni vanno aggiunte ogni volta che sia necessario accedere ad un qualsiasi registro in un banco diverso da quello in cui ci si trova.
E...

...una nota importante.

 **Per default all' accensione, RP0 = 0, ovvero il banco 0 è quello che viene puntato al POR.**

Vuol dire che questo banco sarà accessibile senza alcuna manovra supplementare e quindi è quello più "significativo".
Questo comporta che, se, **dopo il reset**, ad inizio sorgente, agissimo semplicemente con:

```
clrf TRISIO
```

otterremmo, sorpresa sgradita, di aver azzerato il registro **`GPIO`** ! , dato che il banco puntato non è il primo, ma il secondo. Per poter accedere al banco 1, dove si trova **`TRISIO`** occorre aggiungere all'indirizzo di destinazione l'ottavo bit a 1.

Un'altra Nota MOLTO importante.

Per capire cosa comporta lo switch di banco, occorre tenere presente che:

 **Gli switch dei banchi sono modificati in modo automatico solamente al reset/POR, dove sono riportati a 0.**
In ogni altro caso, durante l'esecuzione del programma, per modificarli dovrò utilizzare istruzioni.

Quindi:

 **Il bit di switch dei banchi, una volta settati ad un valore, restano a quel valore fino a che il programma non li modifica o viene generato un reset che li azzera.**

Se siamo passati al banco 1 per agire su **TRISIO**, abbiamo dovuto modificare il valore contenuto in **RP0.**
Questo valore NON sarà modificato che da un'altra istruzione o dal reset.
Di conseguenza, una volta passati in un banco, accediamo alla RAM di quel banco e solo a quella fino a che non modifichiamo lo switch del banco.
Così, volendo poi scrivere **GPIO** , occorrerà ritornare al banco 0 con una nuova scrittura che azzeri il bit **RP0** !

```
bsf   STATUS,RP0 ; bit ottavo = 1 punta al banco 1
clrf  TRISIO     ; tutti i bit utili come uscite
bcf   STATUS,RP0 ; passa al banco 0
clrf  GPIO       ; azzero i latch del port
```

Altrettanto importante:

 **Non ci sono meccanismi automatici nell' Assembler per tener conto del banco in cui ci si trova e questo vuol dire che si rende necessario, da parte del programmatore, una sorveglianza della situazione dei banchi, che è a suo carico.**

Per contro, MPASM genera un **warning** , per l'esattezza il **Message 302**,

```
Message[302]: Register in operand not in bank 0. Ensure that bank bits are
correct.
```

che avvisa l'utente ogni volta si esegue una azione su un registro che non si trova nel banco 0.

 **Ricordiamo ancora una volta che i Message o Warning NON impediscono la compilazione del sorgente.**

Il loro scopo è quello di attirare l'attenzione del programmatore sui punti del testo sorgente che potrebbero dare origine ad un eseguibile che non fa quello che ci si aspetta

In questo senso, i **Message** e la finestra di ***Output*** , durante la compilazione, sono elementi importanti, che il programmatore deve tenere nell'opportuna considerazione.

In ogni caso, l'emissione di warning da parte di MPASM è controllata dalle opzioni di assemblaggio, per cui posso abilitare o meno questa opzione, ammettendo o abolendo i messaggi relativi, con la tipica riga di comando **errorlevel**. Ad esempio:

```
errorlevel -302
```

disabilita la stampa dei messaggi [302].

 **In relazione a quanto detto, non finiremo mai di sconsigliare nel modo più assoluto** l'abolizione dei messaggi di avviso, la tipica **`errorlevel -302`** che infesta troppi esempi sul Web.
Non fatelo: vi togliete la possibilità di disporre di un elemento che facilita il debug.
Se proprio non potete sopportare questi messaggi nel listato, aboliteli solamente DOPO che il programma funziona in modo corretto.

Una valutazione del banco in cui ci si trova, da programma, richiederebbe di analizzare lo STATUS per conoscere il valore di **`RP0`** (e un eventuale **`RP1`**), ma questo non nè utile, nè necessario, dato che una programmazione modulare prevederà un uso intensivo dello pseudo opcode **`banksel`**.

Banksel.

Una ulteriore considerazione, però, è necessaria in riferimento alla necessità **di eliminare ogni riferimento ad assoluti.** Può essere pratico disporre di macro per agire sullo switch di banco:

```
BANK0 macro
    bcf STATUS,RP0 ; passa al banco 0
      endm
BANK1 macro
    bsf STATUS,RP0 ; passa al banco 1
      endm
```

che sono auto esplicative. Quindi:

```
     BANK1           ; passa al banco 1
    clrf  TRISIO     ; tutti i bit utili come uscite
```

Un problema non secondario, però, riguarda il posizionamento degli SFR nei banchi. Per agire correttamente su un determinato SFR dobbiamo sapere dove si trova !
Anche usando **`macro`** si deve comunque sapere in quale banco si trova il registro; se per **`TRIS`** l'informazione può essere tenuta a memoria, potreste dire, senza guardare la mappa di memoria, in che banco si trova **`ANSEL`**?

Dunque è più pratico ricorre ad altre soluzioni. In particolare, è importante sapere che:

 **Usando il Macro Assembler MPASM non occorre scrivere alcuna macro per lo switch dei banchi, dato che esiste lo pseudo opcode `banksel`.**

Importante: **banksel** può avere come oggetto non solo il valore assoluto del banco, ma, molto più

praticamente, la label che indica il registro da raggiungere.
Quindi potrò scrivere:

```
banksel   ANSEL
 movlw    b'00001111'
 movf     ANSEL
```

Qui, come regola aurea su cui abbiamo insistito particolarmente, facciamo ricorso ai **simboli al posto degli assoluti** e all'automatismo offerto da **`banksel` che determina, in base alla label indicata, il banco di appartenenza e comanda di conseguenza lo switch nel modo corretto.**
Non ci occorre sapere a quale banco appartenga **`ANSEL`**: ci penserà l'Assemblatore a definire il giusto valore dello switch di banco.

Questo diventa fondamentale quando abbiamo a che fare con chip più complessi, dove i banchi salgono a 4 ed occorre agire su due switch, **`RP0`** e **`RP1`** che definiscono i bit 6 e 7 dell' indirizzo.
Se sfruttiamo quanto offerto dal MacroAssembler, risparmiamo tempo, fatica e riduciamo gli errori.

Quindi, per favore:

niente macro inutili per commutare i banchi. Non servono.

Anche se le trovate ad ogni piè sospinto negli esempi presenti nel WEB: **NON SERVONO A NULLA**, dato che **`banksel`** svolge la stessa funzione in modo molto più efficace.

Anche se molti esempi sul WEB usano simili macro, non seguite questa abitudine.

Vedremo la questione ancora più volte durante le varie esercitazioni.

`banksel` consente anche di superare il problema del banking in chip con più banchi (fino a 32), dove abbiamo la necessità di usare un numero di bit di switch maggiore; il meccanismo (Enhanced Midrange) è analogo, ma i bit non si troveranno più nello STATUS, bensì un registro specifico.
Sarà compito del compilatore assegnare allo pseudo opcode **`banksel`** la giusta azione in relazione al chip per cui si sta compilando.

Più banchi?

Se osserviamo la mappa di memoria di un chip del genere del 16F690, troviamo ben 4 banchi RAM

	File Address		File Address		File Address		File Address
Indirect addr. (1)	00h	Indirect addr. (1)	80h	Indirect addr. (1)	100h	Indirect addr. (1)	180h
TMR0	01h	OPTION_REG	81h	TMR0	101h	OPTION_REG	181h
PCL	02h	PCL	82h	PCL	102h	PCL	182h
STATUS	03h	STATUS	83h	STATUS	103h	STATUS	183h
FSR	04h	FSR	84h	FSR	104h	FSR	184h
PORTA	05h	TRISA	85h	PORTA	105h	TRISA	185h
PORTB	06h	TRISB	86h	PORTB	106h	TRISB	186h
PORTC	07h	TRISC	87h	PORTC	107h	TRISC	187h
	08h		88h		108h		188h
	09h		89h		109h		189h
PCLATH	0Ah	PCLATH	8Ah	PCLATH	10Ah	PCLATH	18Ah
INTCON	0Bh	INTCON	8Bh	INTCON	10Bh	INTCON	18Bh
PIR1	0Ch	PIE1	8Ch	EEDAT	10Ch	EECON1	18Ch
PIR2	0Dh	PIE2	8Dh	EEADR	10Dh	EECON2(1)	18Dh
TMR1L	0Eh	PCON	8Eh	EEDATH	10Eh		18Eh
TMR1H	0Fh	OSCCON	8Fh	EEADRH	10Fh		18Fh
T1CON	10h	OSCTUNE	90h		110h		190h
TMR2	11h		91h		111h		191h
T2CON	12h	PR2	92h		112h		192h
	13h		93h		113h		193h
	14h		94h		114h		194h
CCPR1L	15h	WPUA	95h	WPUB	115h		195h
CCPR1H	16h	IOCA	96h	IOCB	116h		196h
CCP1CON	17h	WDTCON	97h		117h		197h
	18h		98h	VRCON	118h		198h
	19h		99h	CM1CON0	119h		199h
	1Ah		9Ah	CM2CON0	11Ah		19Ah
	1Bh		9Bh	CM2CON1	11Bh		19Bh
PWM1CON	1Ch		9Ch		11Ch		19Ch
ECCPAS	1Dh		9Dh		11Dh	PSTRCON	19Dh
ADRESH	1Eh	ADRESL	9Eh	ANSEL	11Eh	SRCON	19Eh
ADCON0	1Fh	ADCON1	9Fh	ANSELH	11Fh		19Fh

I banchi sono sempre da 128 byte ciascuno, ma la loro estensione complessiva è 512 bytes e quindi occorrono due bit addizionali, l'ottavo e il nono bit d indirizzo (2^9=512).

Ed in effetti lo STATUS contiene questi due bit: si tratta del bit **RP0**e **RP1**

STATUS – STATUS REGISTER (ADDRESS: 03h, 83h, 103h OR 183h)

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(1)	C(1)
bit 7							bit 0

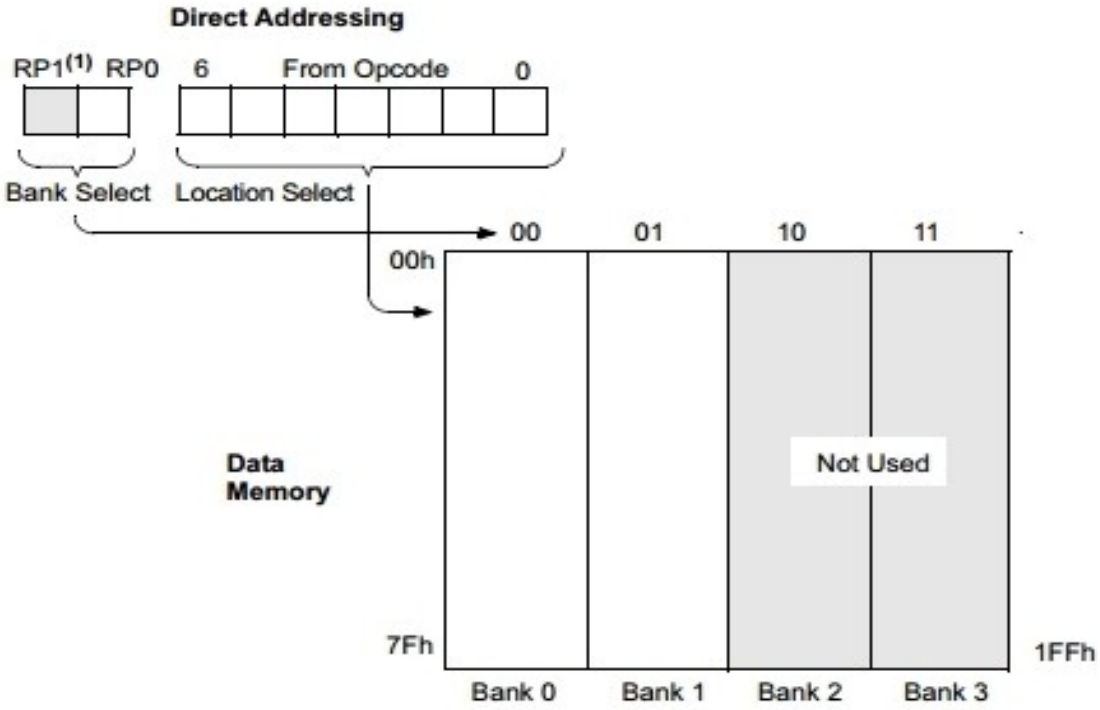
Osserviamo che, rispetto allo STATUS di 12F629, il bit **RP0** si trova nella stessa posizione, mentre il bit **RP1** occupa la posizione a fianco che era non implementata

La funzione dei due bit è intuitiva:

RP1	RP0	Banco
0	0	0
0	1	1
1	0	2
1	1	3

Possiamo rappresentare graficamente quanto detto: vediamo in forma grafica l'azione in un chip dotato di 4 banchi, che il massimo per i Midrange, nella modalità detta indirizzamento diretto, ovvero effettuato usando il contenuto dell'opcode e degli switches di banco.

Da un punto di vista grafico:



Se i banchi sono solo due **`RP1`** non è implementato.

Riassumendo: per raggiungere una locazione in RAM

- 7 bit di indirizzo sono forniti dall'opcode.
- **A questi si aggiungono sempre, in modo automatico e trasparente per l'utente, come bit più significativi `RP1:0` dello `STATUS`.**

Nel caso visto per il 12F629/675, i banchi sono solo due e **`RP1`** è sempre a 0, per cui il solo **`RP0`** determina il banco. In 16F690 i banchi sono 4 e sono presenti sia **`RP1`** che **`RP0`**.

Dobbiamo sempre ricordare che, all'accensione, il banco0 è quello di default (**`RP1:0=00`**), l'inizializzazione di un port all' avvio richiede un cambio di banco:

```
clrf GPIO         ; azzera GPIO banco 0
bsf  STATUS, RP0  ; accedi al banco 1
clrf TRISIO       ; pin come uscite
bcf  STATUS, RP0  ; torna in banco 0
```

E' evidente che, sbagliando il banco, si andrà ad agire su un registro piuttosto che su un altro, con i prevedibili risultati disastrosi sul funzionamento di quanto collegato al microcontroller.

Per curiosità, ci si potrebbe chiedere perchè il **TRIS**, che è collegato al **GPIO** si trovi su un banco diverso.

Questo è dovuto alla filosofia di progetto precedentemente indicata: è normale che durante l'esecuzione di un programma si vari più volte lo stato di un pin in uscita, mentre è meno comune che se ne inverta la direzione durante l'esecuzione; solitamente la direzione viene impostata all'inizio. Questo vuol dire che l'azione sul registro di direzione è "meno importante" di quella sul **GPIO** e quindi **TRIS** viene "esiliato" in banco 1.

Banking.

Nei Midrange il problema della divisione della RAM in banchi diventa essenziale e che, se viene trascurato, impedisce il funzionamento del programma.

Nei Baseline, in generale, gli SFR principali sono mappati in modo uguale in tutti i banchi e sono accessibili da qualsiasi punto del programma senza particolari attenzioni (solamente quelli relativi alla gestione della EEPROM interna, che può essere considerata non di uso comune, hanno un accesso diverso dal banco 0),. Quindi, l'utente difficilmente si trova di fronte alla necessità di confrontarsi con il sistema dei banchi.

Usando, poi, chip in cui il problema dei banchi non esiste (es. 10F2xx, 12F509), dato che hanno solo il banco 0, risulta che il principiante non è intralciato dal problema; però, per contro, finisce per trascurarlo del tutto. La facilità di uso è un vantaggio al momento, ma comporta che, quando si affronteranno chip più complessi, mancherà la comprensione dell'importanza che ha il problema e si troverà inizialmente in difficoltà.

Così, affrontando i Midrange, dove la divisione in banchi dell'area di memoria RAM è un fatto comune, si può indurre in gravi errori che impediscono il funzionamento del programma. Siccome il problema è più in relazione con la struttura hardware dei PIC piuttosto che con la logica del programma, ne risulta che spesso la ricerca di problemi causati da una errata gestione dei banchi possono essere causa di sostanziose perdite di tempo e di indubbio stress.

Osservando la mappa di memoria RAM vediamo che per i Midrange esistono, come per tutti i PIC, registri accessibili da tutti i banchi, ma si tratta degli SFR fondamentali di gestione degli algoritmi (**STATUS, PCL, PCLATH, FSR, INTCON**); si tratta di elementi chiave della programmazione e come tali è utile averli sempre disponibili, senza commutare i banchi, ottenendo una maggiore rapidità ed efficienza di esecuzione del codice.

Ma per tutti gli altri, compresi quelli della semplice gestione degli I/O digitali, come abbiamo visto sopra **occorre passare da un banco ad un altro!**

Quanto detto vale ovviamente per tutto il contenuto dell'area di memoria RAM, compresa la RAM dati, che risulta accessibile in diversa misura sui banchi. Anche qui, se trattiamo solo programmi minimali o con un impegno di RAM dati tale da essere contenuta nel solo banco 0, il problema non si pone. Se, però, la richiesta di RAM dati aumenta oltre la capacità di un banco, ecco che la commutazione dei banchi diventa una necessità.

Ancora qualcosa sui banchi.

Se torniamo ad ad osservare la mappa di memoria RAM e in particolare gli SFR

	File Address		File Address		File Address		File Address
Indirect addr. (1)	00h	Indirect addr. (1)	80h	Indirect addr. (1)	100h	Indirect addr. (1)	180h
TMR0	01h	OPTION_REG	81h	TMR0	101h	OPTION_REG	181h
PCL	02h	PCL	82h	PCL	102h	PCL	182h
STATUS	03h	STATUS	83h	STATUS	103h	STATUS	183h
FSR	04h	FSR	84h	FSR	104h	FSR	184h
PORTA	05h	TRISA	85h	PORTA	105h	TRISA	185h
PORTB	06h	TRISB	86h	PORTB	106h	TRISB	186h
PORTC	07h	TRISC	87h	PORTC	107h	TRISC	187h
	08h		88h		108h		188h
	09h		89h		109h		189h
PCLATH	0Ah	PCLATH	8Ah	PCLATH	10Ah	PCLATH	18Ah
INTCON	0Bh	INTCON	8Bh	INTCON	10Bh	INTCON	18Bh
PIR1	0Ch	PIE1	8Ch	EEDAT	10Ch	EECON1	18Ch
PIR2	0Dh	PIE2	8Dh	EEADR	10Dh	EECON2(1)	18Dh
TMR1L	0Eh	PCON	8Eh	EEDATH	10Eh		18Eh
TMR1H	0Fh	OSCCON	8Fh	EEADRH	10Fh		18Fh
T1CON	10h	OSCTUNE	90h		110h		190h
TMR2	11h		91h		111h		191h
T2CON	12h	PR2	92h		112h		192h
	13h		93h		113h		193h
	14h		94h		114h		194h
CCPR1L	15h	WPUA	95h	WPUB	115h		195h
CCPR1H	16h	IOCA	96h	IOCB	116h		196h
CCP1CON	17h	WDTCON	97h		117h		197h
	18h		98h	VRCON	118h		198h
	19h		99h	CM1CON0	119h		199h
	1Ah		9Ah	CM2CON0	11Ah		19Ah
	1Bh		9Bh	CM2CON1	11Bh		19Bh
PWM1CON	1Ch		9Ch		11Ch		19Ch
ECCPAS	1Dh		9Dh		11Dh	PSTRCON	19Dh
ADRESH	1Eh	ADRESL	9Eh	ANSEL	11Eh	SRCON	19Eh
ADCON0	1Fh	ADCON1	9Fh	ANSELH	11Fh		19Fh

notiamo alcune caratteristiche:

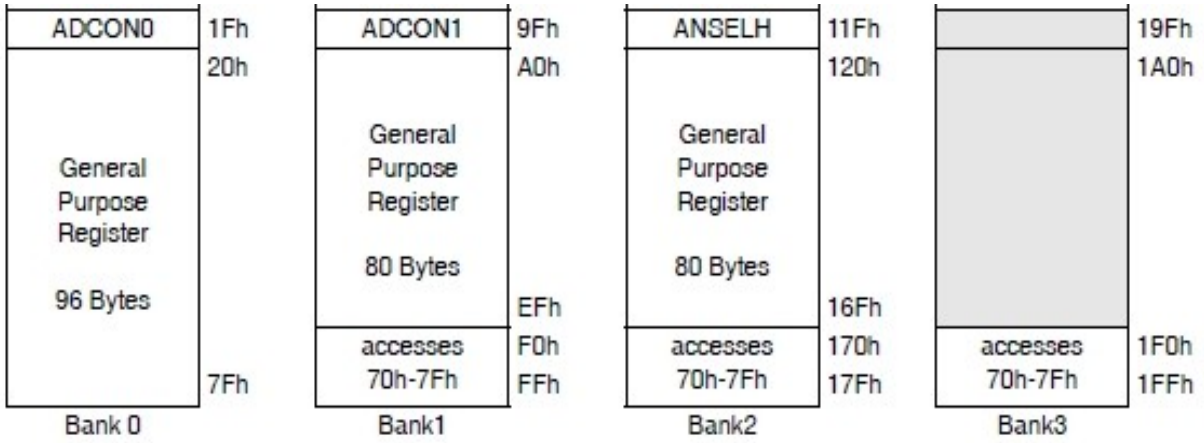
1. come detto in precedenza, alcuni SFR sono mappati in tutti i banchi. Questo vuol dire che, in qualunque banco ci si trovi (= qualunque sia il valore di RP1:0), essi saranno accessibili

direttamente. Come detto, si tratta degli SFR di gestione generale.

2. Gli altri SFR, che sono utilizzati per il controllo delle periferiche, sono disponibili solamente in un banco preciso. Questo vuol dire che sarà necessario manovrare RP1:0 per raggiungere il giusto banco

3. gli SFR non sono "impaccati" in uno spazio compatto, ma sono alternati a aree non implementate (in grigio nella figura). Questa caratteristica è comune a vari chip: la disposizione apparentemente non ottimale è dovuta al fatto che gli spazi liberi, in altri modelli, sono riempiti da SFR di periferiche qui non presenti. La cosa si giustifica facilmente con la filosofia di progetto che cerca di avere quanto più possibilmente uniformi le strutture dei componenti di una famiglia, in modo da riservare il minimo di sorprese all'utilizzatore.

E' anche interessante osservare la parte della mappa relativa alla RAM dati:



Notiamo che è disponibile RAM dati sui primi tre banchi in maniera uguale e solamente parziale sul banco 3. Ma, principalmente, dobbiamo notare che la RAM dati è divisa in due componenti:

- **RAM condivisa** (*shared RAM*), che, come per alcuni degli SFR, è accessibile in modo indipendente dalla situazione degli switches di banco. Nella figura, essa va da 70h a 7Fh nel banco 0, ma è ripetuta identicamente negli altri tre:

Indirizzo	Banco
70h-7Fh	0
F0h-FFh	1
170h-17Fh	2
1F0h-1FFh	3

Questo vuol dire che scrivendo, ad esempio, la locazione 7Eh, ritroveremo lo stesso valore in FEh, 17Eh e 1FEh e viceversa.

Questa area di RAM condivisa ha lo scopo di supportare dati che devono essere accessibili con immediatezza, senza commutare banchi, ad esempio quelli di salvataggio del contesto nelle gestioni delle interruzioni (come è spiegato nell'inserto relativo all'interrupt).
In generale, è una parte minoritaria della RAM dati e va usata con attenzione per evitare

sovrapposizioni di scritture.

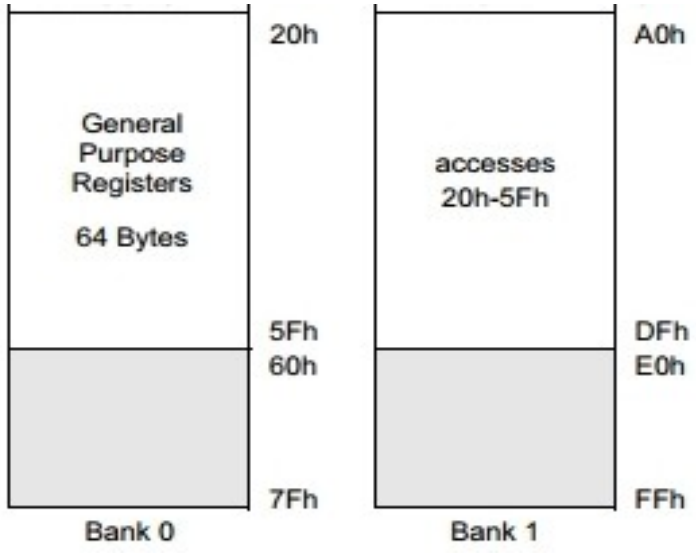
- Il resto della RAM dati è "***banked***", ovvero è accessibile solo in uno specifico banco. Per cui, se devo scrivere la cella 20h, dovrò essere in banco 0 (**`RP1:0 = 00`**), mentre per scrivere la cella 120h dovrò essere in banco 2 (**`RP1:0 = 10`**).
-

Si tratta della maggior parte della RAM dati.
Solitamente si utilizza lo statement **`CBLOCK/ENDC`** o **`EQU`** per puntare l'indirizzo iniziale di una assegnazione, ma il **`Macro Assembler MPASM`** fornisce anche diverse modalità di allocazione della RAM dati per compilazioni modulari, dove, a priori, non conosciamo l'area in cui il **Linker** posizionerà le label.

Abbiamo così lo statement **`UDATA`** in relazione ad aree RAM generica e **`UDATA_SHR`** specificamente previsto per le aree di RAM shared, che allocano le risorse di RAM dati in funzione delle disponibilità del chip usato.

In questo modo il programmatore può decidere dove posizionare la memoria richiesta dalle varie sezioni del programma, ma non gli è richiesta una strenua attenzione all'indirizzamento, che viene risolto in modo automatico.

Ovviamente esistono eccezioni nella struttura dell'area RAM. Ad esempio, essa diventa particolare per 12F629/675, la cui situazione è la seguente:



I chip dispongono di soli due banchi, avendo una quantità di SFR limitata e anche la RAM dati è limitata, ma **è configurata in modo da essere tutta RAM shared.**

Ovvero, sarà necessario operare sugli switches di banco per quanto riguarda gli SFR, ma non sarà necessario per la RAM dati. Questa scelta facilità l'utilizzatore, ma penalizza per contro la quantità di RAM disponibile.
In particolare, notiamo che lo spazio tra 60h e 7Fh (che è mirrored tra E0h e FFh) non è implementata.

Dato che ogni chip dispone di risorse diverse e la sua mappa di memoria può essere differente da quelle esemplificate, la cosa diventa importante sia nella valutazione del chip da impiegare a seconda delle necessità del programma, sia nella necessità o meno di usare switch di banco.

E' da notare che una procedura che sia generalizzabile al massimo, prevederà comunque le azioni sui banchi; nel caso in cui esse non siano necessarie per le particolari caratteristiche del chip usato, sono presenti nel MacroAssembler MPASM meccanismi che aboliscono le operazioni dove la situazione delle risorse le renda inutili, dandone notizia con un messaggio, ad esempio

```
Message[312] Page or Bank selection not needed for this device. No code
generated
```

Una Nota importante relativa a banksel.

banksel , nonostante appaia come una sola pseudo-istruzione, in effetti è una macro che viene compilata come uno o due `bcf/bsf STATUS,RP0` a seconda delle caratteristiche del chip impiegato

Nel caso di un chip con due banchi sarà sostituito da una istruzione **`bcf/bsf STATUS,RP0.`**

Però, un chip con 4 banchi richiede 2 switches e il banksel sarà compilato come una coppia di istruzioni:

```
bcf/bsf   STATUS,RP0
 bcf/bsf STATUS,RP1
```

Quindi è da evitare la manovra:

```
banksel     RAM0
btsfc       testpoint, tst5
 banksel    RAM1
next ...
```

Ricordiamo che **il salto condizionato btsfc/s riguarda LA SOLA linea successiva.**

Questo va bene per chip con due banchi, per cui il banksel origina una sola linea.

Nel momento in cui abbiamo in uso un chip con 4 banchi, il `banksel` sarà compilato come DUE righe e il salto non porterebbe a `next`, ma al secondo `bcf/bsf` con le evidenti conseguenze.

Occorre, dunque, prestare attenzione.

Alcuni appunti su cose da emendare.

In molti esempi che trovate nel mare magnum del WEB, sono presenti elementi impropri che andrebbero evitati.
Sul problema dei comandi **errorlevel** abbiamo già detto sopra.

Un altro caso è quello di trovare nel sorgente una istruzione di selezione del banco 0 subito dopo il reset. Ad esempio:

```
ORG     0               ; vettore reset
bcf     STATUS,RP0
```

Questo non ha alcuna senso, né utilità in quanto, al POR, è proprio il banco 0 ad essere indirizzato.

Si è sentita l'obiezione:"meglio mettere l'istruzione perchè non si sa mai...", **cosa che non ha alcun senso**.

Se il microcontroller funziona regolarmente, i default al POR e al reset saranno automaticamente eseguiti. Quindi, ribadirli nel programma aggiungendo istruzioni non serve a nulla; microsecondi buttati.

Solo se il chip fosse difettoso ci sarebbe la possibilità di un funzionamento anomalo, ma in questo caso è inutile aggiungere istruzioni perchè non sarebbe possibile determinare a priori quali saranno i problemi espressi dal componente mal funzionante.

Anche nel caso in cui ci sia la possibilità di ritrovarsi con reset eseguiti parzialmente a causa di variazioni anomale della tensione di alimentazione, la soluzione consisterà nell'eliminare alla sorgente il problema, non certo quella di applicare "toppe" a casaccio nel firmware.

In questo senso, si deve considerare che i Midrange incorporano due meccanismi di protezione, **PoWer On Timer** e **Brown Out Detector** proprio per evitare qualsiasi problema nella fase di POR.

Per ultimo, dell'impiego dissennato di assoluti al posto di label abbiamo già trattato ampiamente nella parte
relativa ai Baseline, a cui rimandiamo. Qui ricordiamo solo che l'impiego di valori assoluti nell' indirizzamento è da bandire “assolutamente”, in quanto possibile fonte di non pochi problemi, tra cui quello immediato di rendere illeggibile il sorgente.

Altre considerazioni verrano fatte durante lo svolgimento delle esercitazioni.

Bancofobia.

Si rileva quanto mai sovente negli utenti non professionali la tendenza a voler minimizzare a tutti i costi l'hardware attorno al PIC, arrivando ad eliminare le resistenze in serie ai LED. Questa è una mistificazione della funzione specifica dei microcontroller.

Certamente questo componente racchiude in se le numerose funzioni di un sistema a microprocessore: un solo chip contiene CPU, RAM, ROM, clock, sistemi di gestione dei bus, I/O, timer, ecc. Non occorre altro.

Ma questo non vuol dire che il microcontroller debba poter funzionare senza componenti esterni. Se, esternamente, le regole dell'elettronica richiedono componenti addizionali per interfacciarsi con carichi e ingressi, questi non possono essere aboliti tout court per un malcompreso “obbligo” di minimizzazione.

La stessa cosa capita per il software: abolire quanto possibile istruzioni “inutili” sembra l'obbligo morale di tanti. Sotto la mannaia dei giacobini a caccia di istruzioni “inutili” cadono quelle di selezione dei banchi, a seguito di ragionamenti anche corretti.
Ad esempio, se siamo in banco 0, in un chip con 4 banchi, per passare al banco 1 basta che modifichi il solo **`RP0`**. Così, se sono in banco 3, per passare al 2 basta che modifichi **`RP1`**. E' “uno spreco” quello che fa **`banksel`**, scrivendo comunque entrambi i bit.

Se il ragionamento è corretto, non lo sono le sue motivazioni.
Innanzitutto il microcontroller esegue le istruzioni in tempi minimi: a 4MHz di clock, una istruzione impegna tipicamente 1us: solamente in una ridottissima percentuale di algoritmi, legati a processi esterni particolari, ci si trova a dover valutare i tempi a questa (e anche a maggiore) definizione.
Aggiungere qua e là 1 o 2us è del tutto trascurabile.

Inoltre, nel 99% delle applicazioni, la maggior parte del tempo il micro la passa in attese ed avere “risparmiato” 1us non apporta alcun beneficio.
Se non, forse purtroppo, la soddisfazione personale di aver abolito una riga “inutile”.

E non esiste neppure un vantaggio in termini di energia nell'eliminare alcune istruzioni; in questo senso quello che conta sono ben altre azioni.

Per contro, l'eliminazione degli switch di banco inutili richiede da parte del programmatore la costante e sicura conoscenza del banco in cui si sta lavorando, dato che l'Assembler non può avere meccanismi di correzione a riguardo.

Ne risulta che, fino a quando si ha a che fare con programmi semplici, la cosa, anche se richiede un lavoro extra, è fattibile. Ma quando il programma diventa appena un poco complicato, con ampio uso di RAM, l'assillo dello stato dei banchi diventa un peso che, se trascurato, porta al mancato funzionamento del programma.

E, dato che la questione dei banchi non è un problema di software, ma una necessità dovuta all'organizzazione fisica del chip, ecco che ci si trova davanti a notevoli difficoltà nel ricercare le cause del mancato funzionamento: la logica del programma è perfetta, ma non funziona. Non funziona perchè si è dimenticato di scambiare un banco ed invece di agire su un registro, agisco su

un altro. Questo dal listato non è evidente. Spesso, solo entrando con una emulazione diretta, lunga e noiosa, si riesce a capire dove sta l'errore.
Soprattutto quando, con poca coscienza e conoscenza, si inizia il sorgente con l'eliminazione del **`Message[302]`**, stupidaggine purtroppo così comune negli esempi disponibili.

Comunque, stiamo osservando che la forma modulare del sorgente, se è un carico "inutile" per piccoli programmi, diventa l'unico modo serio per realizzarne di grandi senza impiegare tempi che forse per un hobbista all'inizio della sua carriera sono accettabili, ma non lo sono per chi ha raggiunto una certa abilità, né tanto meno per chi deve realizzare programmi per lavoro, dove il time-to-market stabilisce i costi. Solo l'uso costante di **`banksel`** consente di realizzare programmi che non sono affetti dal problema dei banchi; e, anche se si aggiungono all'esecuzione alcuni microsecondi, non è certo un problema.

Qualora i tempi diventino critici, allora si potrà agire con la ricerca di algoritmi ottimizzati. Altrimenti, volendo ottenere con il minimo rischio sorgenti funzionanti, meglio concentrarsi sulla logica del programma e cancellare dalla mente completamente il problema dei banchi.

Peraltro, è questo che fanno i linguaggi ad alto livello come il C o il BASIC. In questi il problema dei banchi non si pone in quanto i compilatori provvedono a "farcire" adeguatamente l'eseguibile con il costante comando degli switches dei banchi; il tutto al di fuori dell'azione del programmatore, che non deve pensare a questo aspetto poco simpatico dei PIC.

Quindi, a nostro parere vale la regola aurea per cui il miglior programma è quello che funziona come deve.
Una "bellezza e eleganza intrinseca" del codice sorgente non riguardano l'eseguibile (anche se la mancanza di forma nel sorgente è per lo meno causa di difficoltà nel debug e nella manutenzione del software). Per chi avesse dubbi, basta ricordare la gara annuale che si svolge tra programmatori di C per realizzare le logiche più contorte e meno intuitive (il C permette queste cose), pur avendo programmi perfettamente funzionanti.

Quindi la conclusione è:

- se il programma è minimale, l'impegno di RAM sarà altrettanto minimale. Solitamente si utilizzerà una versione assoluta per il sorgente e, di conseguenza, la gestione dei banchi potrà essere seguita "manualmente", se si desidera. Sempre avendo presente il rischio di trovarsi con un programma non funzionante per misteriosi motivi, che poi si riducono ad aver dimenticato di cambiare banco...

- se il programma assume una certa complessità, come capita nelle applicazioni professionali, non ha alcun senso spendere tempo e fatica per correre dietro alla situazione dei banchi, cosa che non comporta alcun miglioramento del funzionamento del programma, mentre distrae il programmatore dal seguire la logica del suo programma.

In particolare, se si ha a che fare per lavoro con l'Assembly, la via delle compilazioni modulari è l'unica che permetta di raggiungere risultati anche di elevata complessità. Qui, la gestione dei banchi deve essere esclusa dalle priorità del lavoro e l'uso di **banksel** diventa un obbligo.

In conclusione, anche se siete solo hobbisti, è più sensato seguire vie che conducano al risultato con il giusto sforzo, senza aggiungere carchi inutili dovuti ad una mal compresa esigenza di ridurre tempi e byte. Poi, ovviamente, se si vuole menare vanto di avere realizzato comandi di LED senza

resistenze in serie o programmi in cui si sono “guadagnati” 10 byte su 1500 e risparmiato 2us su un ciclo di esecuzione di 2s, è una scelta personale. Basta che si sappia cosa esattamente si sta facendo.

Un breve riassunto.

1. I PIC sono realizzati in architettura Harvard, ovvero con il bus dati e il bus programma separati. Ne deriva che la memoria dati (RAM) e quella programma sono due aree diverse. Per i Midrange il bus dati è ampio 8 bit e quello istruzione è ampio 14 bit.

2. L'area della memoria RAM contiene sia la RAM dati vera e propria, sia i registri di controllo delle varie funzioni (SFR).

3. Alcune istruzioni sono dirette a leggere o scrivere l'area RAM. Queste istruzioni dispongono di un numero limitato di bit per definire l'indirizzo su cui operare. Per i Midrange si tratta di un massimo di 7bit con i quali è possibile indirizzare fino a 128 locazioni.

4. Nei Midrange la quantità di SFR e di memoria dati disponibile è tale da necessitare di un'area maggiore di 128 bit. Il costruttore organizza questa area in blocchi da 128 bit, detti banchi. Sono disponibili, a seconda del chip, 2 o 4 banchi per la memoria RAM.

5. Per poter raggiungere gli indirizzi al di fuori dei 128 bit gestiti direttamente dall'istruzione, è implementato un meccanismo che aggiunge a questo indirizzo uno o due bit, come settimo e ottavo; questo consente di indirizzare correttamente locazioni in tutti i banchi.

6. I bit di questo meccanismo sono **`RP0`** (e **`RP1,`** se ci sono 4 banchi), detti switches dei banchi e si trovano nel registro **`STATUS.`** La loro logica è questa:

RP1	RP0	Banco
0	0	0
0	1	1
1	0	2
1	1	3

7. Per passare da un banco ad un altro occorrerà modificare questi bit. La cosa è facilmente realizzabile con lo pseudo opcode **`banksel`** fornito dal MacroAssembler MPASM.

Vanno ricordate le seguenti regole:

- **I bit di switch dei banchi sono SEMPRE aggiunti all'indirizzo dell'opcode, in modo automatico e trasparente per l'utente.**

- **Al reset questi bit sono posti a 0, per cui è puntato il banco 0. Per passare a qualsiasi altro banco occorre modificare gli switch.**

- **I bit si modificano con una semplice istruzione bcf/bsf o, meglio, con il banksel. Una volta modificati, restano in questa condizione fino ad una successiva modifica o reset.**