

The University of Texas at Arlington

Lecture 8

Addressing, Tables, Banks, Memory



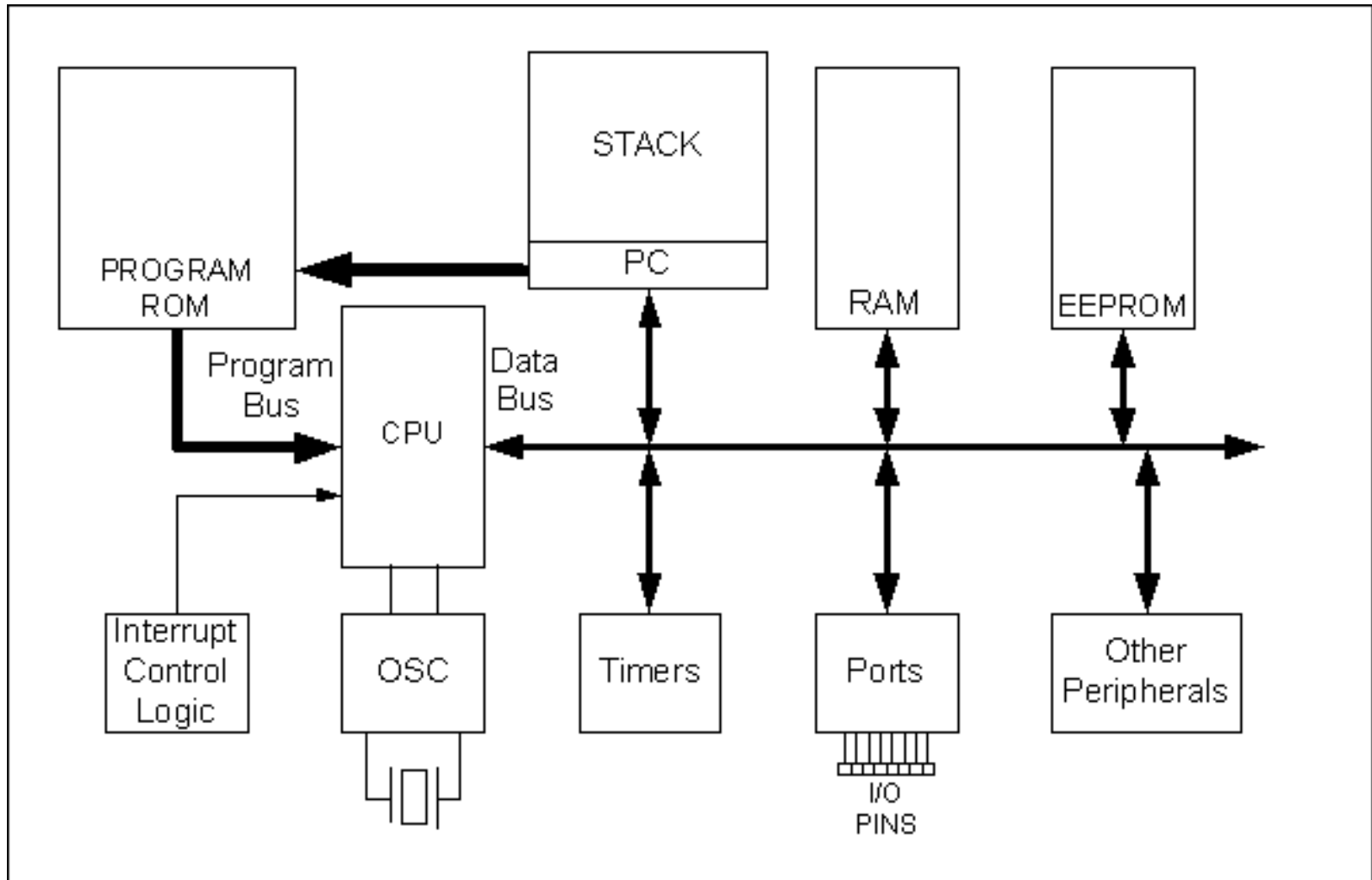
CSE@UTA

CSE 3442/5442

Embedded Systems 1

Based heavily on slides by Dr. Gergely Záruba and Dr. Roger Walker

How Can the CPU Access Data?



How Can the CPU Access Data?

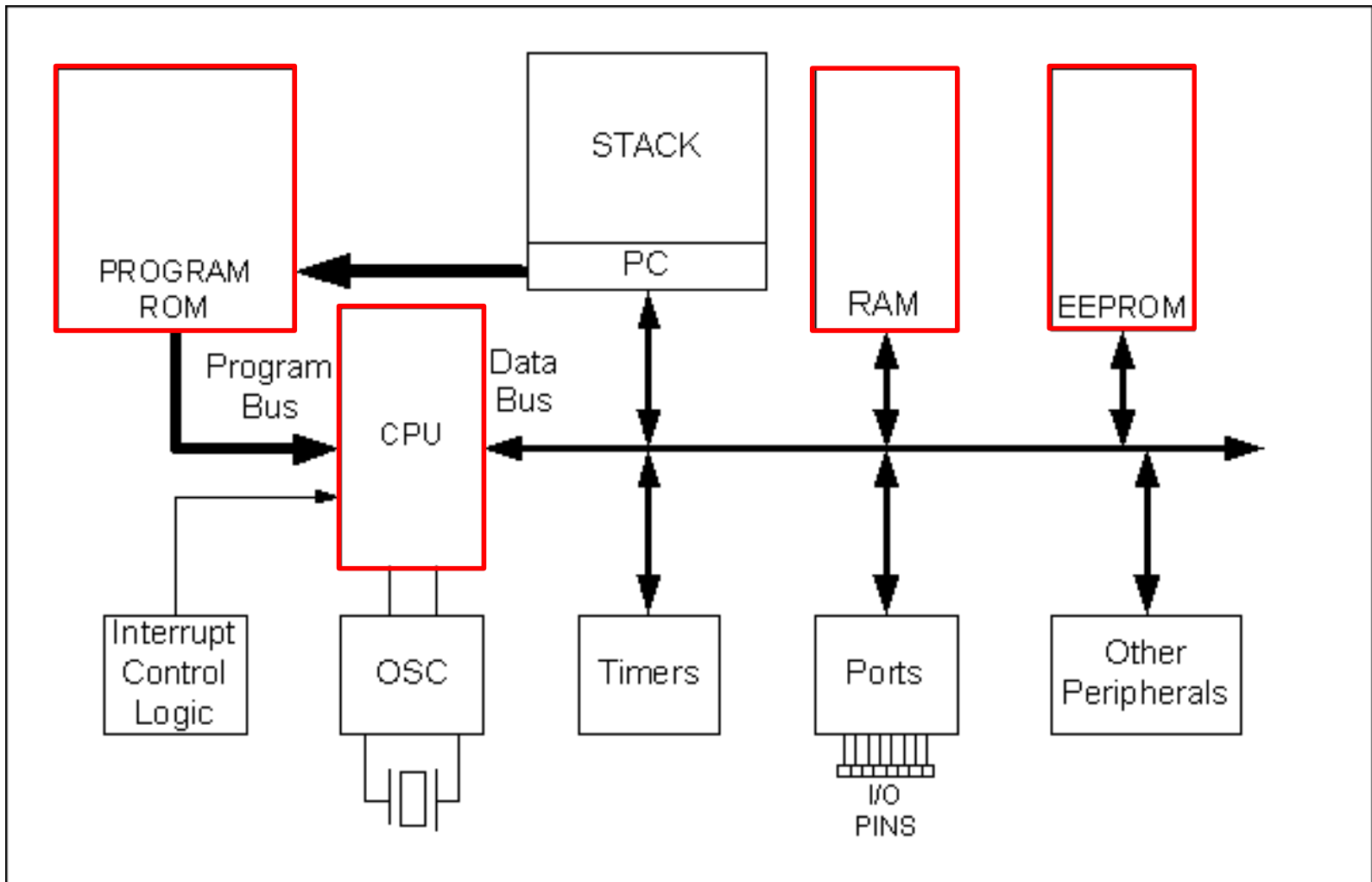
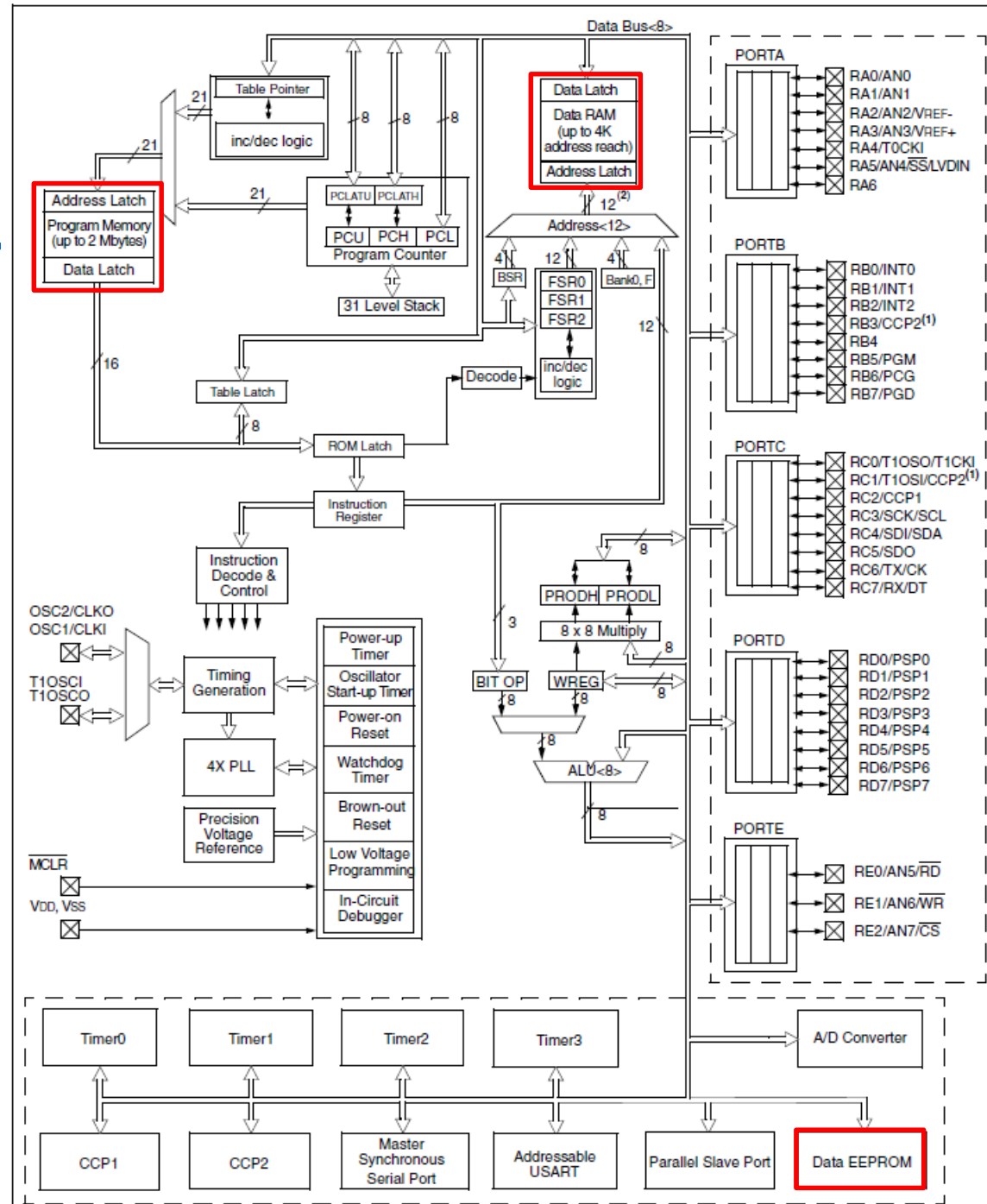
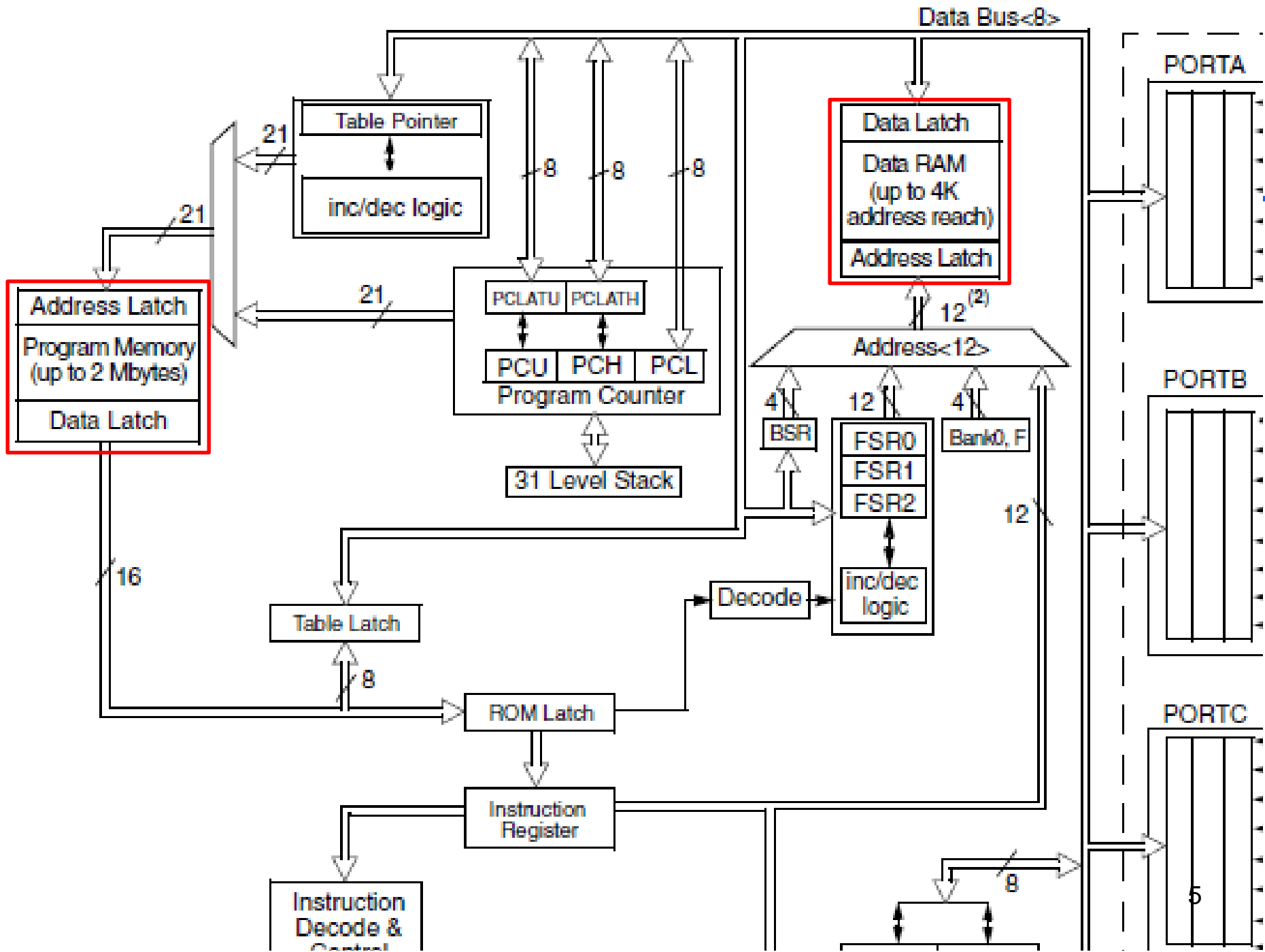


FIGURE 1-2: PIC18F4X2 BLOCK DIAGRAM







Accessing Data

- CPU can grab data...
 - from a register
 - from a memory location
 - provided as an immediate value
- Known as different **Addressing Modes**
- Determined by μ C designers
 - Cannot be altered by the programmer



Addressing Modes

1. Immediate

- Operand part of the instruction (constant K)

2. Direct

- Instruction has the operand of a RAM address and thus can be directly addressed

3. Register Indirect

- Kind of like using pointers to address registers. There are specific SFRs set aside for this.

4. Indexed-ROM

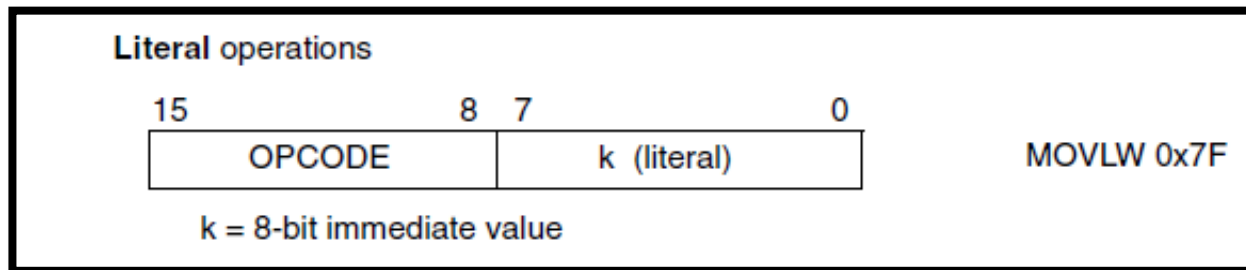
- Constant fixed data stored alongside the program code



Immediate Addressing

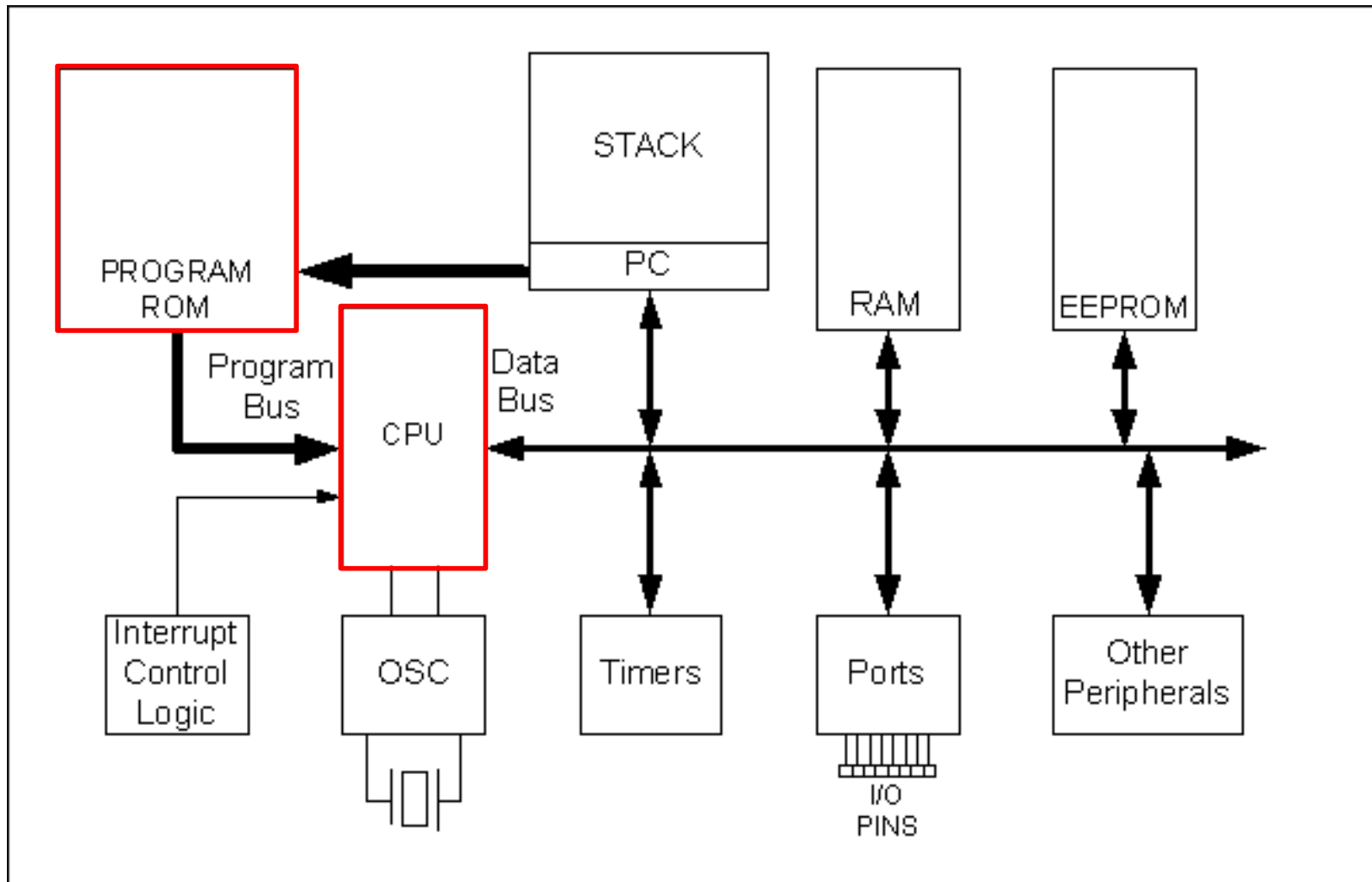
- Operand (data) is part of the instruction, thus ‘immediately’ available when instruction is fetched
 - Immediate Data == “Literal” Data
- Literal Operations

MOVLW 0x7F ; 7FH → WREG

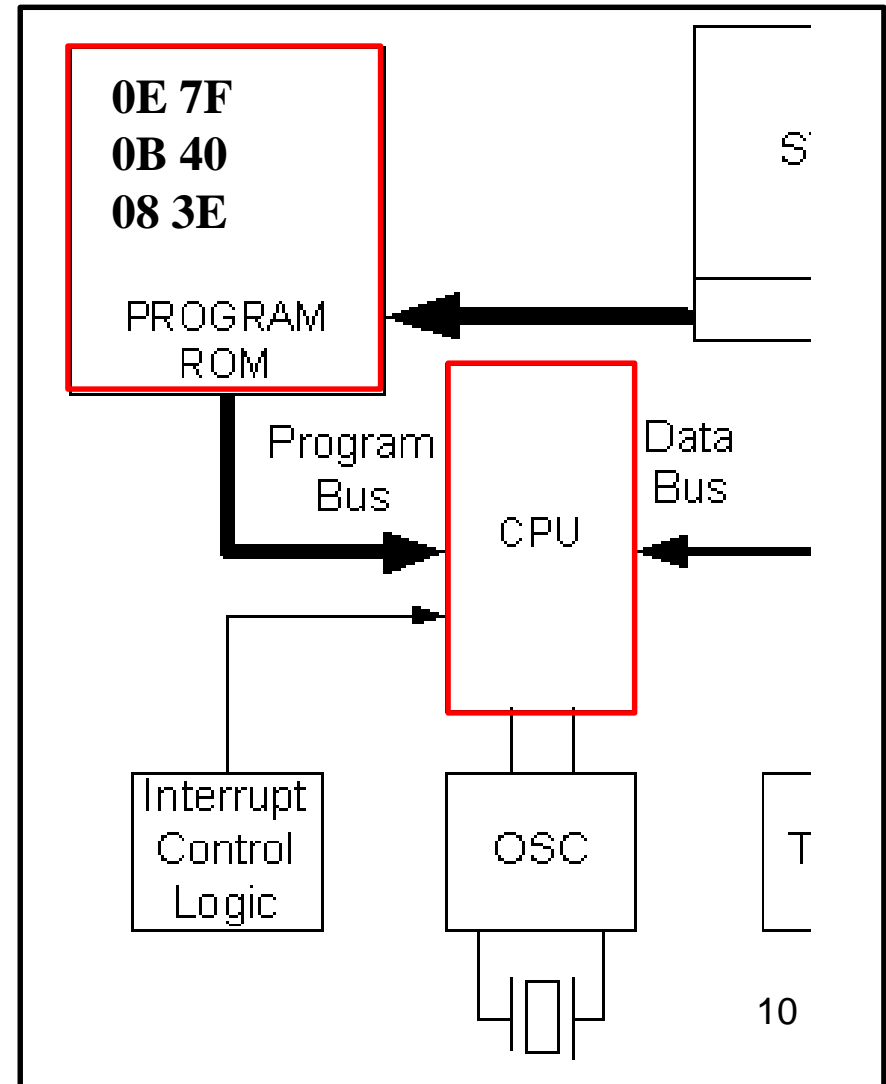
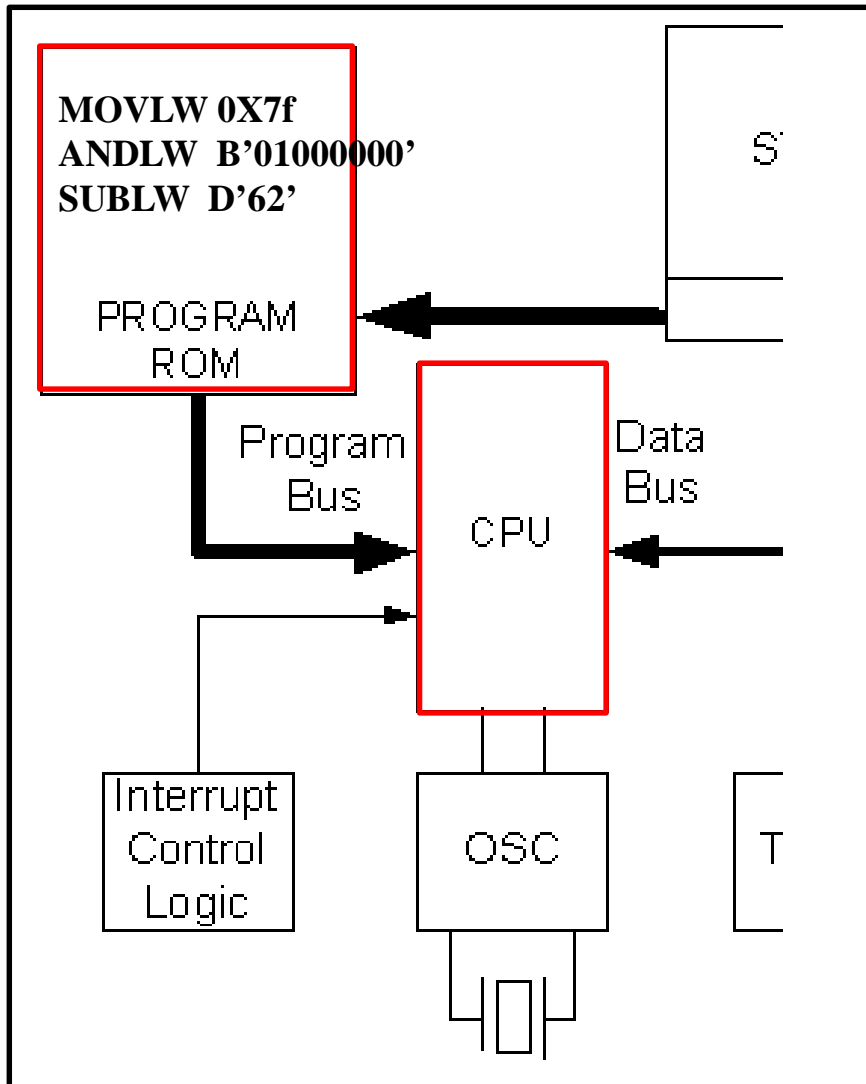


ANDLW B'01000000' ;AND WREG with 40H
SUBLW D'62' ;subtract WREG from 62

Immediate Addressing



Immediate Addressing





Immediate Addressing

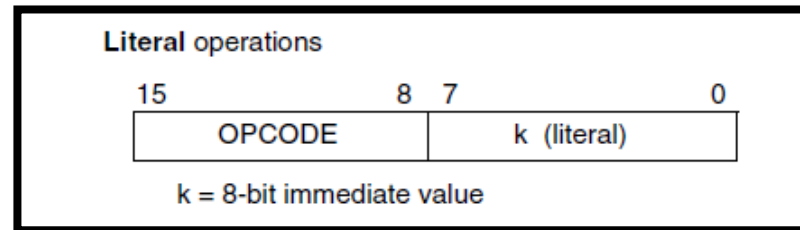
- Operand (data) is part of the instruction, thus ‘immediately’ available when instruction is fetched
 - Immediate Data == “Literal” Data

- Using EQU

COUNT EQU 0x30

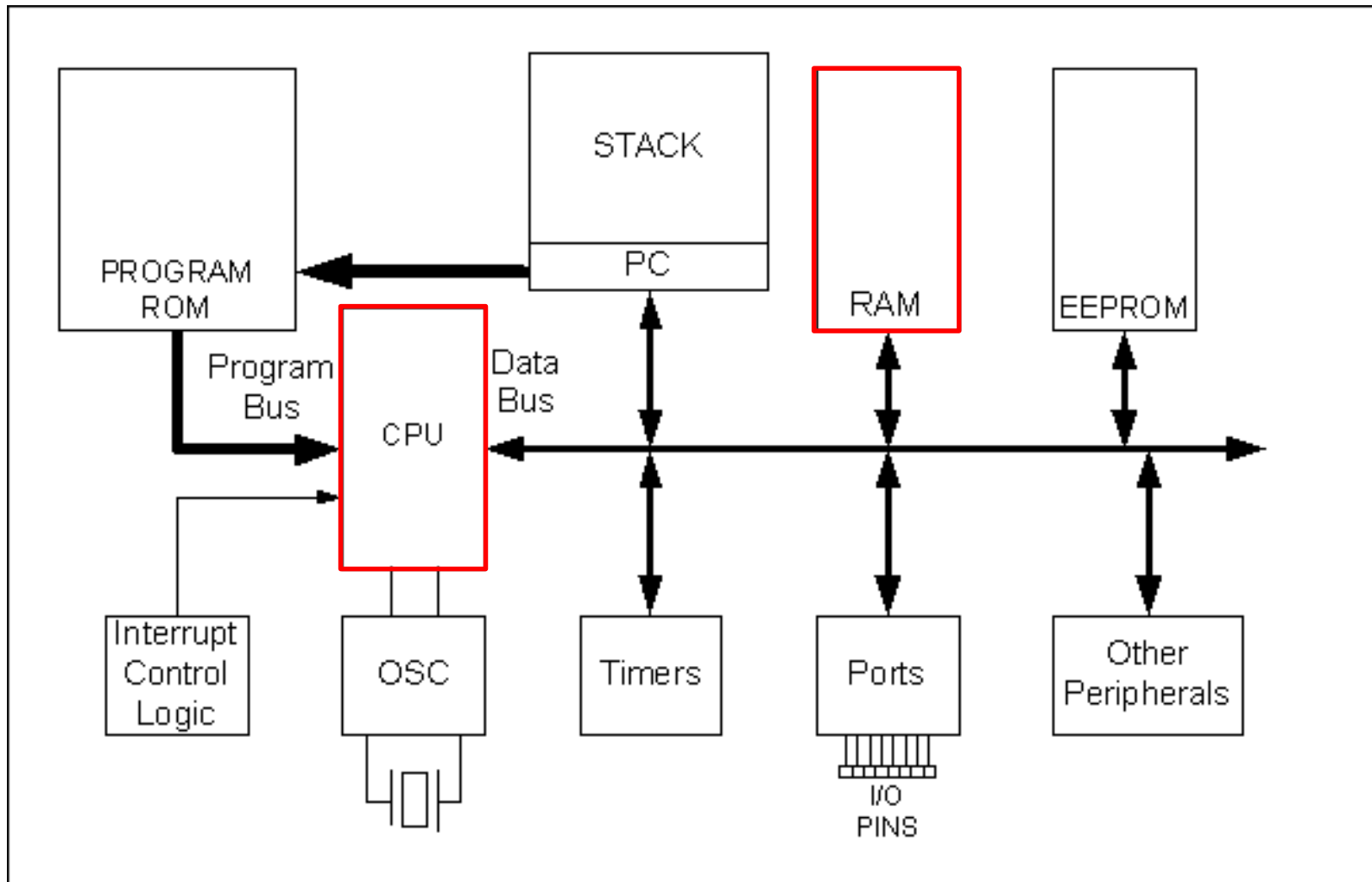
MOVLW COUNT ; 30H → WREG

;Assembler insures 30H placed in the second byte of the instruction



- RAM cannot be immediately addressed (i.e., there is no MOVKF).

Direct Addressing





Direct Addressing

- Operand (data) is obtained from **or to** file register
 - Data is in a RAM location whose address is known and included as part of the instruction

```
MOVLW 0x56      ;WREG = 56H (immediate addressing mode)
MOVWF 0x40      ;copy WREG into fileReg RAM location 40H
MOVFF 0x40,0x50 ;copy data from loc 40H to 50H.
```

- Instead of “literal” data being put in ROM next to Opcode the address of the File Register location is put there
- Note the MOVWF can only access the current bank while the MOVFF instruction can access all of the 4K RAM address space (recall, that File register (RAM) arranged into 16 Banks of 256 bytes).



Bank Switching

- Max 4K of RAM (in PIC18 but not all have max)
- Only 256 bytes are addressable
- RAM is divided into a max of 16 banks
- Default bank's lower 128 bytes are general purpose, while upper 128 are the SFR
- **MOVWF fileReg , A**
 - Until now we have ignored A
 - If A=0 then default bank is used
 - If A=1 then bank selector register is used to determine bank

Bank Addressing

MOVWF	Move W to f				
Syntax:	<code>[label] MOVWF f[,a]</code>				
Operands:	$0 \leq f \leq 255$ $a \in [0,1]$				
Operation:	$(W) \rightarrow f$				
Status Affected:	None				
Encoding:	<table border="1"> <tr> <td>0110</td> <td>111a</td> <td>ffff</td> <td>ffff</td> </tr> </table>	0110	111a	ffff	ffff
0110	111a	ffff	ffff		
Description:	<p>Move data from W to register 'f'. Location 'f' can be anywhere in the 256 byte bank. If 'a' is 0, the Access Bank will be selected, overriding the BSR value. If 'a' = 1, then the bank will be selected as per the BSR value (default).</p>				
Words:	1				
Cycles:	1				

Bank Addressing

- Direct Addressing Instructions take two bytes, one for the operation code and the other for an 8 bit 256 byte **Access Bank** address.

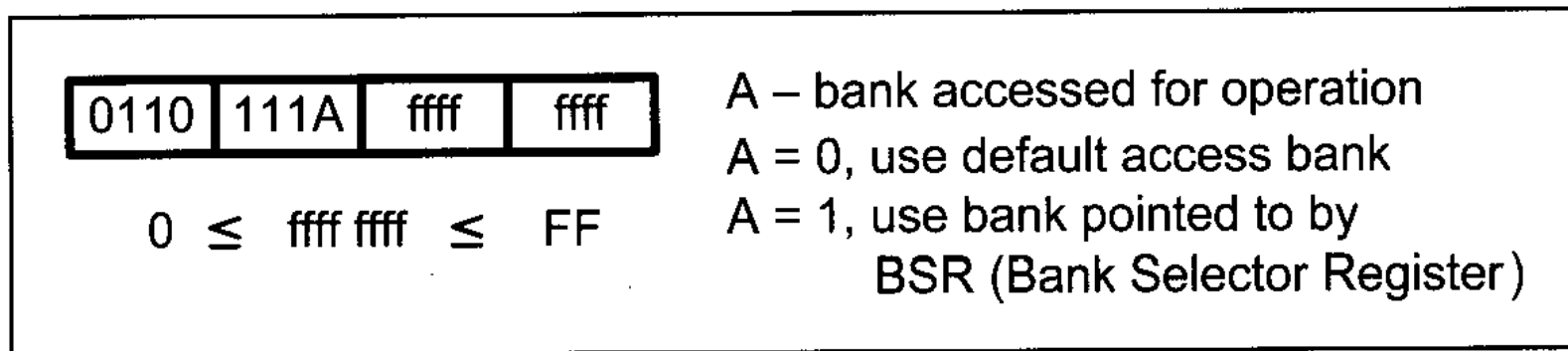
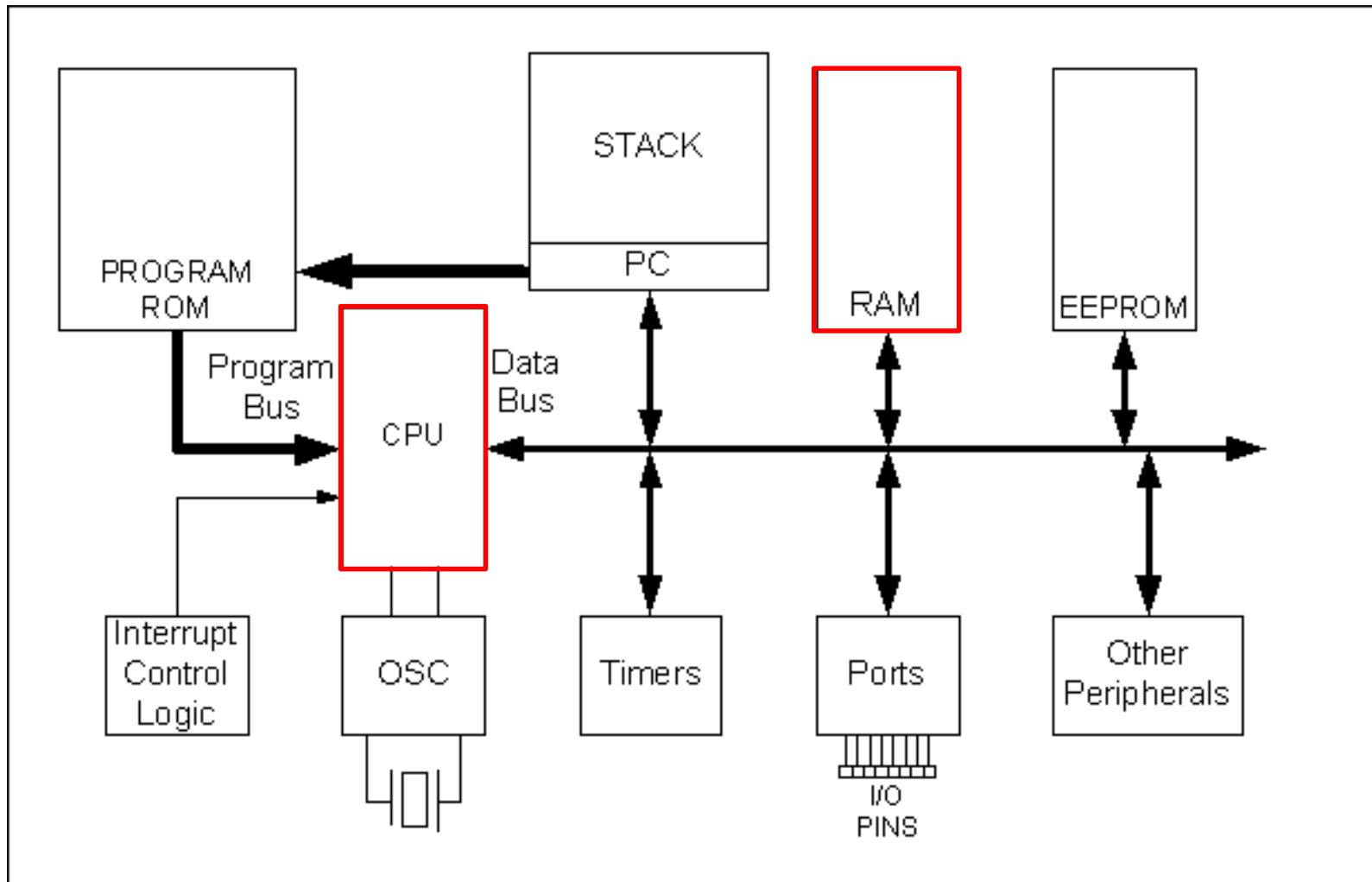


Figure 6-1b. MOVWF Direct Addressing Opcode

- Thus will need way to access the other banks

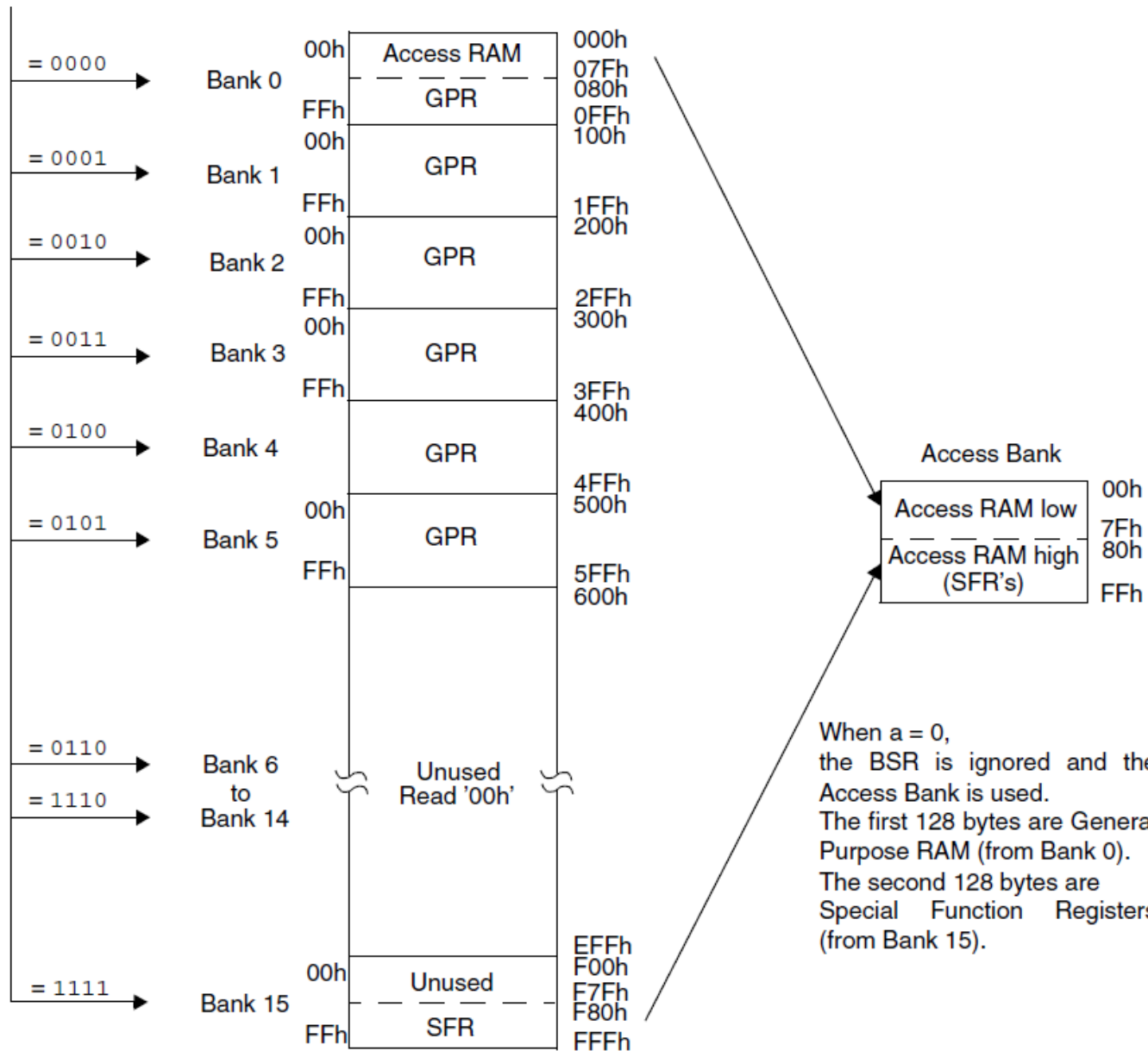
Direct Addressing





BSR<3:0>

Data Memory Map



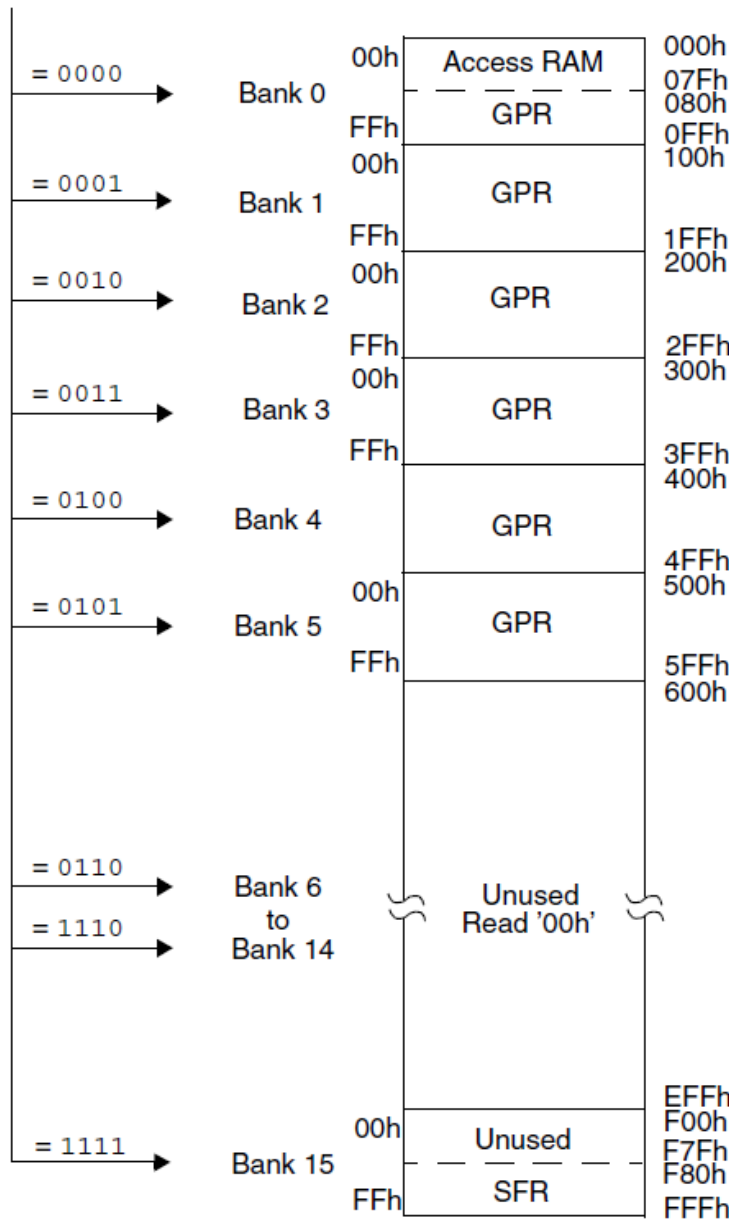
When a = 0, the BSR is ignored and the Access Bank is used. The first 128 bytes are General Purpose RAM (from Bank 0). The second 128 bytes are Special Function Registers (from Bank 15).

When a = 1, the BSR is used to specify the RAM location that the instruction uses.

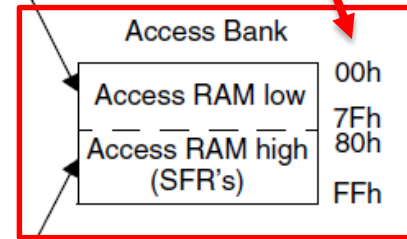


BSR<3:0>

Data Memory Map



Relative Address



When a = 0, the BSR is ignored and the Access Bank is used. The first 128 bytes are General Purpose RAM (from Bank 0). The second 128 bytes are Special Function Registers (from Bank 15).

0xFFF = 4KB

When a = 1, the BSR is used to specify the RAM location that the instruction uses.

Relative Address

Absolute Address

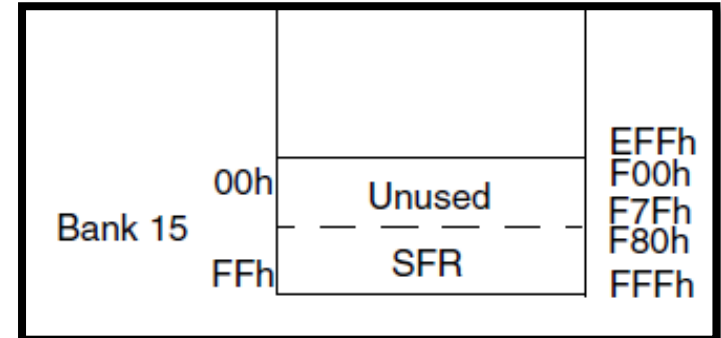


SFRs

Special Function Registers

TABLE 4-1: SPECIAL FUNCTION REGISTER MAP

Address	Name	Address	Name	Address	Name	Address	Name
FFh	TOSU	FDFh	INDF2 ⁽³⁾	FBFh	CCPR1H	F9Fh	IPR1
FFEh	TOSH	FDEh	POSTINC2 ⁽³⁾	FBEh	CCPR1L	F9Eh	PIR1
FFDh	TOSL	FDDh	POSTDEC2 ⁽³⁾	FBDh	CCP1CON	F9Dh	PIE1
FFCh	STKPTR	FDCh	PREINC2 ⁽³⁾	FBCh	CCPR2H	F9Ch	—
FFBh	PCLATU	FDBh	PLUSW2 ⁽³⁾	FBBh	CCPR2L	F9Bh	—
FFAh	PCLATH	FDAh	FSR2H	FBAh	CCP2CON	F9Ah	—
FF9h	PCL	FD9h	FSR2L	FB9h	—	F99h	—
FF8h	TBLPTRU	FD8h	STATUS	FB8h	—	F98h	—
FF7h	TBLPTRH	FD7h	TMR0H	FB7h	—	F97h	—
FF6h	TBLPTRL	FD6h	TMR0L	FB6h	—	F96h	TRISE ⁽²⁾
FF5h	TABLAT	FD5h	T0CON	FB5h	—	F95h	TRISD ⁽²⁾
FF4h	PRODH	FD4h	—	FB4h	—	F94h	TRISC
FF3h	PRODL	FD3h	OSCCON	FB3h	TMR3H	F93h	TRISB
FF2h	INTCON	FD2h	LVDCON	FB2h	TMR3L	F92h	TRISA
FF1h	INTCON2	FD1h	WDTCON	FB1h	T3CON	F91h	—
FF0h	INTCON3	FD0h	RCON	FB0h	—	F90h	—
FEFh	INDF0 ⁽³⁾	FCFh	TMR1H	FAFh	SPBRG	F8Fh	—
FEeh	POSTINC0 ⁽³⁾	FCEh	TMR1L	FAEh	RCREG	F8Eh	—
FEDh	POSTDEC0 ⁽³⁾	FCDh	T1CON	FADh	TXREG	F8Dh	LATE ⁽²⁾
FECh	PREINC0 ⁽³⁾	FCCh	TMR2	FACH	TXSTA	F8Ch	LATD ⁽²⁾
FEbh	PLUSW0 ⁽³⁾	FCBh	PR2	FABh	RCSTA	F8Bh	LATC
FEAh	FSR0H	FCAh	T2CON	FAAh	—	F8Ah	LATB
FE9h	FSR0L	FC9h	SSPBUF	FA9h	EEADR	F89h	LATA
FE8h	WREG	FC8h	SSPADDD	FA8h	EEDATA	F88h	—
FE7h	INDF1 ⁽³⁾	FC7h	SSPSTAT	FA7h	EECON2	F87h	—
FE6h	POSTINC1 ⁽³⁾	FC6h	SSPCON1	FA6h	EECON1	F86h	—
FE5h	POSTDEC1 ⁽³⁾	FC5h	SSPCON2	FA5h	—	F85h	—
FE4h	PREINC1 ⁽³⁾	FC4h	ADRESH	FA4h	—	F84h	PORTE ⁽²⁾
FE3h	PLUSW1 ⁽³⁾	FC3h	ADRESL	FA3h	—	F83h	PORTD ⁽²⁾
FE2h	FSR1H	FC2h	ADCON0	FA2h	IPR2	F82h	PORTC
FE1h	FSR1L	FC1h	ADCON1	FA1h	PIR2	F81h	PORTB
FE0h	BSR	FC0h	—	FA0h	PIE2	F80h	PORTA





Bank Selector Register

- **Bank Selector Register (BSR)** is an 8 bit register in the SFR
 - Only the 4 LSBs are used
- If using the BSR, then bank 0 is a continuous 00F-FFH and bank FH's upper 128 bytes are the SFR (as in the RAM map)
- Default value for BSR is 0
- Thus if need to use other banks:
 - Load BSR with the desired banks number **MOVLB** instruction can be helpful
 - Use A=1 in the instructions
- **INCF MYREG, F, 0** vs. **INCF MYREG, F, 1**



MOVLB

Bank Selector Register

MOVLB **Move literal to low nibble in BSR**

Syntax: [*label*] MOVLB *k*

Operands: $0 \leq k \leq 255$

Operation: $k \rightarrow \text{BSR}$

Status Affected: None

Encoding:

0000	0001	kkkk	kkkk
------	------	------	------

Description: The 8-bit literal 'k' is loaded into the Bank Select Register (BSR).

Words: 1

Cycles: 1



Bank Selector Register Example

MYREG EQU 0x40

```

MOVLB 0x2 ; 0010 → BSR (use bank 2)
MOVLW 0
MOVWF MYREG, 1 ; loc 0x240=0
INCF MYREG, F, 1 ; loc 0x240=1
INCF MYREG, F, 1 ; loc 0x240=2

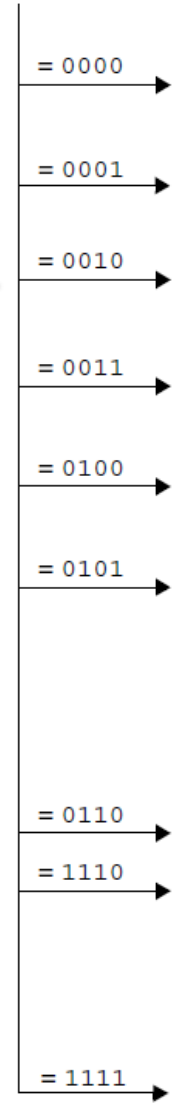
```

```

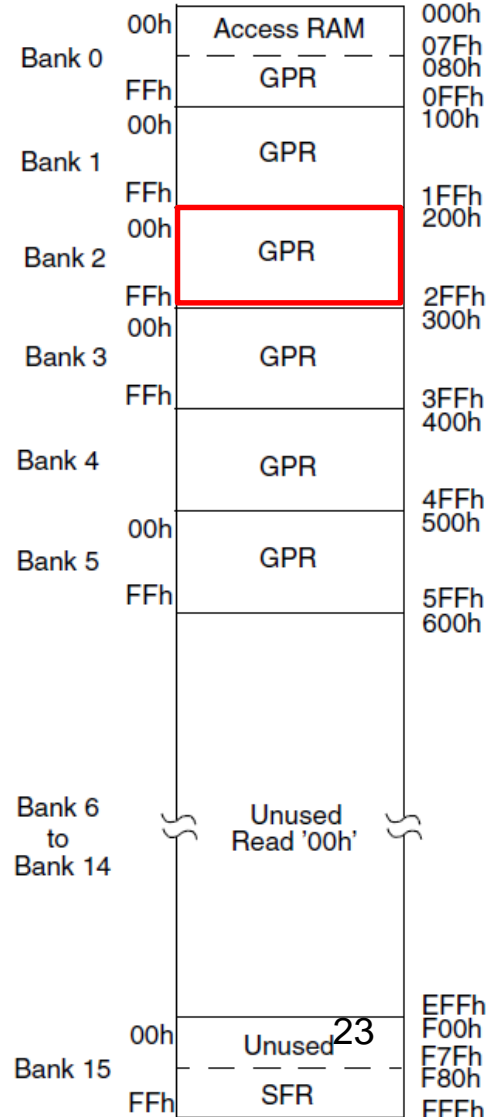
MOVLB 0x2 ; 0010 → BSR (use bank 2)
MOVLW 0
MOVWF 0x40, 1 ; loc 0x240=0
INCF 0x40, F, 1 ; loc 0x240=1
INCF 0x40, F, 1 ; loc 0x240=2

```

BSR<3:0>



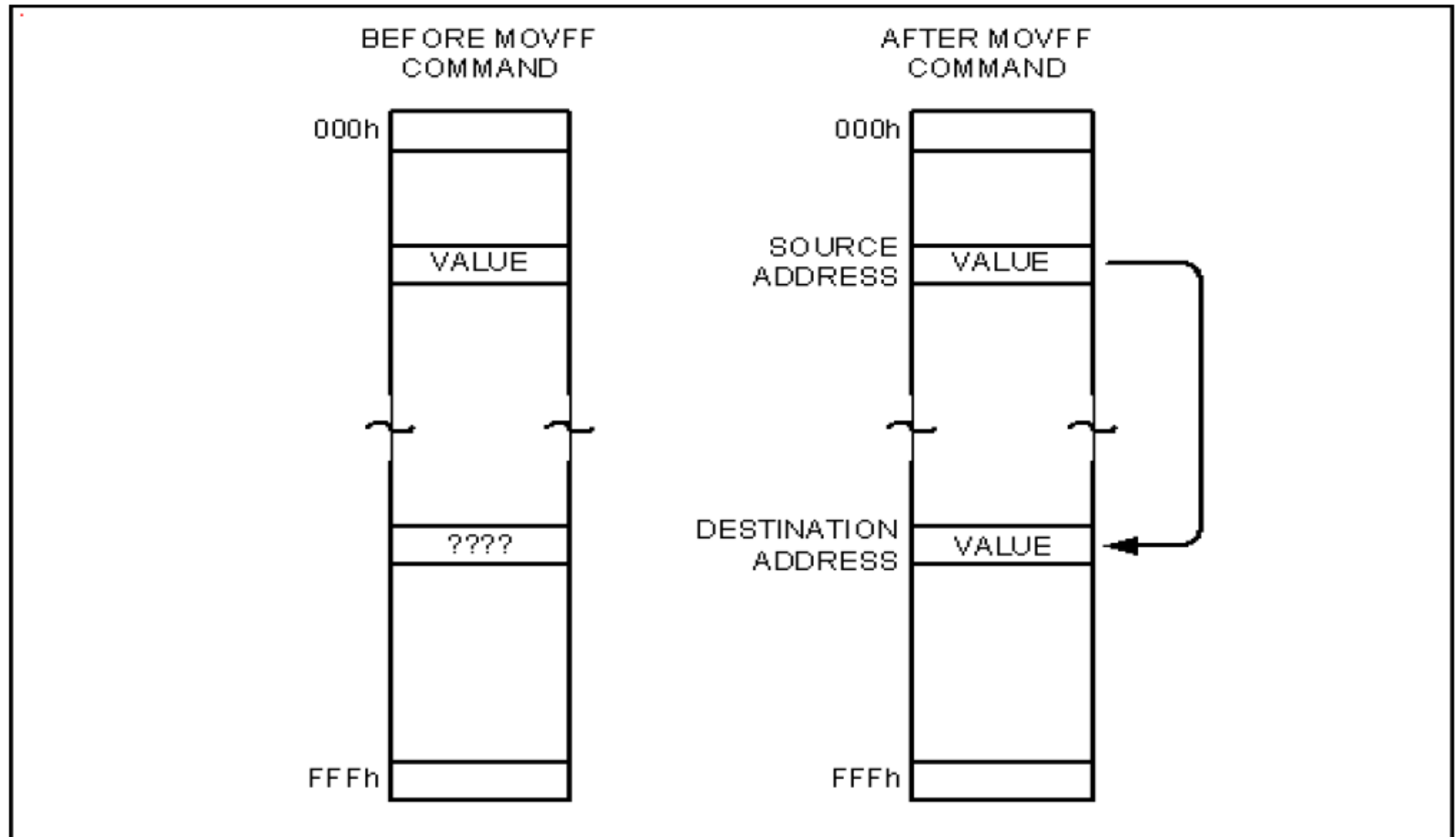
Data Memory Map



MOVFF FileRegS, FileRegD

MOVFF 05H, 09H

MOVFF 09H, LATC





Moving Data Between RAM Registers

- **MOVFF** can move data between any RAM registers without the need for BSR
- This is possible because **MOVFF** is 4 byte instruction (8 bits of opcode, 2×12 bits for address = 32 bits total)
- But no arithmetic can take place without the use of the WREG

MOVFF

MOVFF

Move f to f

Syntax: `[label] MOVFF fs,fd`

Operands: $0 \leq f_s \leq 4095$
 $0 \leq f_d \leq 4095$

Operation: $(f_s) \rightarrow f_d$

Status Affected: None

Encoding:

1st word (source)
 2nd word (destin.)

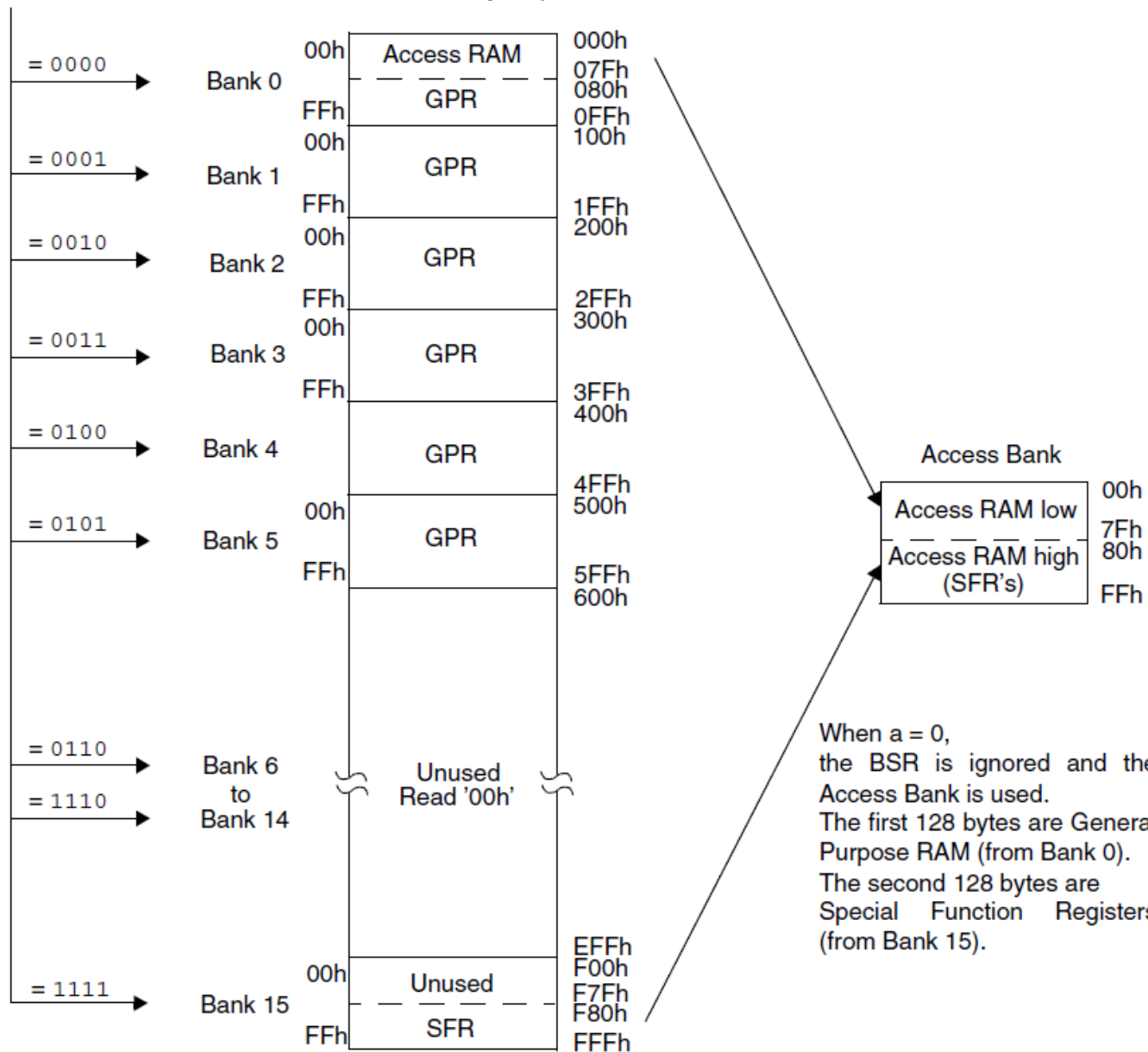
1100	ffff	ffff	ffff _s
1111	ffff	ffff	ffff _d

Description: The contents of source register 'f_s' are moved to destination register 'f_d'. Location of source 'f_s' can be anywhere in the 4096 byte data space (000h to FFFh), and location of destination 'f_d' can also be anywhere from 000h to FFFh.



BSR<3:0>

Data Memory Map



When a = 0, the BSR is ignored and the Access Bank is used. The first 128 bytes are General Purpose RAM (from Bank 0). The second 128 bytes are Special Function Registers (from Bank 15).

When a = 1, the BSR is used to specify the RAM location that the instruction uses.



SFRs

TABLE 4-1: SPECIAL FUNCTION REGISTER MAP

Address	Name	Address	Name	Address	Name	Address	Name
FFh	TOSU	FDFh	INDF2 ⁽³⁾	FBFh	CCPR1H	F9Fh	IPR1
FFEh	TOSH	FDEh	POSTINC2 ⁽³⁾	FBEh	CCPR1L	F9Eh	PIR1
FFDh	TOSL	FDDh	POSTDEC2 ⁽³⁾	FBDh	CCP1CON	F9Dh	PIE1
FFCh	STKPTR	FDCh	PREINC2 ⁽³⁾	FBCh	CCPR2H	F9Ch	—
FFBh	PCLATU	FDBh	PLUSW2 ⁽³⁾	FBBh	CCPR2L	F9Bh	—
FFAh	PCLATH	FDAh	FSR2H	FBAh	CCP2CON	F9Ah	—
FF9h	PCL	FD9h	FSR2L	FB9h	—	F99h	—
FF8h	TBLPTRU	FD8h	STATUS	FB8h	—	F98h	—
FF7h	TBLPTRH	FD7h	TMR0H	FB7h	—	F97h	—
FF6h	TBLPTRL	FD6h	TMR0L	FB6h	—	F96h	TRISE ⁽²⁾
FF5h	TABLAT	FD5h	T0CON	FB5h	—	F95h	TRISD ⁽²⁾
FF4h	PRODH	FD4h	—	FB4h	—	F94h	TRISC
FF3h	PRODL	FD3h	OSCCON	FB3h	TMR3H	F93h	TRISB
FF2h	INTCON	FD2h	LVDCON	FB2h	TMR3L	F92h	TRISA
FF1h	INTCON2	FD1h	WDTCON	FB1h	T3CON	F91h	—
FF0h	INTCON3	FD0h	RCON	FB0h	—	F90h	—
FEFh	INDF0 ⁽³⁾	FCFh	TMR1H	FAFh	SPBRG	F8Fh	—
FEh	POSTINC0 ⁽³⁾	FCEh	TMR1L	FAEh	RCREG	F8Eh	—
FEDh	POSTDEC0 ⁽³⁾	FCDh	T1CON	FADh	TXREG	F8Dh	LATE ⁽²⁾
FECh	PREINC0 ⁽³⁾	FCCh	TMR2	FACH	TXSTA	F8Ch	LATD ⁽²⁾
FEBh	PLUSW0 ⁽³⁾	FCBh	PR2	FABh	RCSTA	F8Bh	LATC
FEAh	FSR0H	FCAh	T2CON	FAAh	—	F8Ah	LATB
FE9h	FSR0L	FC9h	SSPBUF	FA9h	EEADR	F89h	LATA
FE8h	WREG	FC8h	SSPADD	FA8h	EEDATA	F88h	—
FE7h	INDF1 ⁽³⁾	FC7h	SSPSTAT	FA7h	EECON2	F87h	—
FE6h	POSTINC1 ⁽³⁾	FC6h	SSPCON1	FA6h	EECON1	F86h	—
FE5h	POSTDEC1 ⁽³⁾	FC5h	SSPCON2	FA5h	—	F85h	—
FE4h	PREINC1 ⁽³⁾	FC4h	ADRESH	FA4h	—	F84h	PORTE ⁽²⁾
FE3h	PLUSW1 ⁽³⁾	FC3h	ADRESL	FA3h	—	F83h	PORTD ⁽²⁾
FE2h	FSR1H	FC2h	ADCON0	FA2h	IPR2	F82h	PORTC
FE1h	FSR1L	FC1h	ADCON1	FA1h	PIR2	F81h	PORTB
FE0h	BSR	FC0h	—	FA0h	PIE2	F80h	PORTA

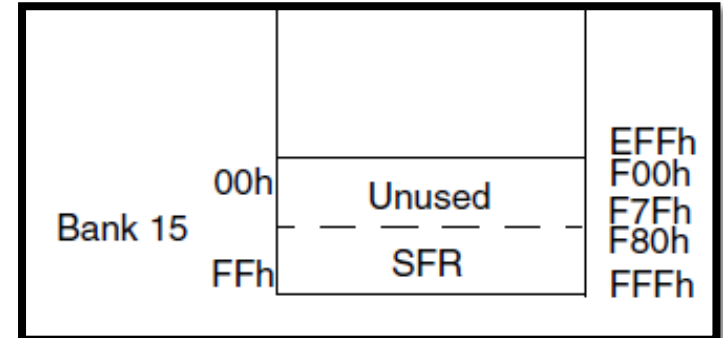


TABLE 4-1: SPECIAL FUNCTION REGISTER MAP



Address	Name	Address	Name	Address	Name	Address	Name
FFh	TOSU	FDh	INDF ⁽³⁾	FBFh	CCPR1H	F9Fh	IPR1
FFEh	TOSH	FDEh	POSTINC ⁽³⁾	FBEh	CCPR1L	F9Eh	PIR1
FFDh	TOSL	FDDh	POSTDEC ⁽³⁾	FBDh	CCP1CON	F9Dh	PIE1
FFCh	STKPTR	FDCh	PREINC ⁽³⁾	FBCh	CCPR2H	F9Ch	—
FFBh	PCLATU	FDBh	PLUSW ⁽³⁾	FBBh	CCPR2L	F9Bh	—
FFAh	PCLATH	FDAh	FSR2H	FBAh	CCP2CON	F9Ah	—
FF9h	PCL	FD9h	FSR2L	FB9h	—	F99h	—
FF8h	TBLPTRU	FD8h	STATUS	FB8h	—	F98h	—
FF7h	TBLPTRH	FD7h	TMR0H	FB7h	—	F97h	—
FF6h	TBLPTRL	FD6h	TMR0L	FB6h	—	F96h	TRISE ⁽²⁾
FF5h	TABLAT	FD5h	T0CON	FB5h	—	F95h	TRISD ⁽²⁾
FF4h	PRODH	FD4h	—	FB4h	—	F94h	TRISC
FF3h	PRODL	FD3h	OSCCON	FB3h	TMR3H	F93h	TRISB
FF2h	INTCON	FD2h	LVDCON	FB2h	TMR3L	F92h	TRISA
FF1h	INTCON2	FD1h	WDTCON	FB1h	T3CON	F91h	—
FF0h	INTCON3	FD0h	RCON	FB0h	—	F90h	—
FEFh	INDF ⁽³⁾	FCFh	TMR1H	FAFh	SPBRG	F8Fh	—
FEEh	POSTINC ⁽³⁾	FCEh	TMR1L	FAEh	RCREG	F8Eh	—
FEDh	POSTDEC ⁽³⁾	CDh	T1CON	FADh	TXREG	F8Dh	LATE ⁽²⁾
FECh	PREINC ⁽³⁾	CCCh	TMR2	FACH	TXSTA	F8Ch	LATD ⁽²⁾
FEBh	PLUSW ⁽³⁾	CBh	PR2	FABh	RCSTA	F8Bh	LATC
FEAh	FSR0H	CAh	T2CON	FAAh	—	F8Ah	LATB
FE9h	FSR0L	FC9h	SSPBUF	FA9h	EEADR	F89h	LATA
FE8h	WREG	FC8h	SSPADD	FA8h	EEDATA	F88h	—
FE7h	INDF ⁽³⁾	FC7h	SSPSTAT	FA7h	EECON2	F87h	—
FE6h	POSTINC ⁽³⁾	FC6h	SSPCON1	FA6h	EECON1	F86h	—
FE5h	POSTDEC ⁽³⁾	FC5h	SSPCON2	FA5h	—	F85h	—
FE4h	PREINC ⁽³⁾	FC4h	ADRESH	FA4h	—	F84h	PORTE ⁽²⁾
FE3h	PLUSW ⁽³⁾	FC3h	ADRESL	FA3h	—	F83h	PORTD ⁽²⁾
FE2h	FSR1H	FC2h	ADCON0	FA2h	IPR2	F82h	PORTC
FE1h	FSR1L	FC1h	ADCON1	FA1h	PIR2	F81h	PORTB
FE0h	BSR	FC0h	—	FA0h	PIE2	F80h	PORTA



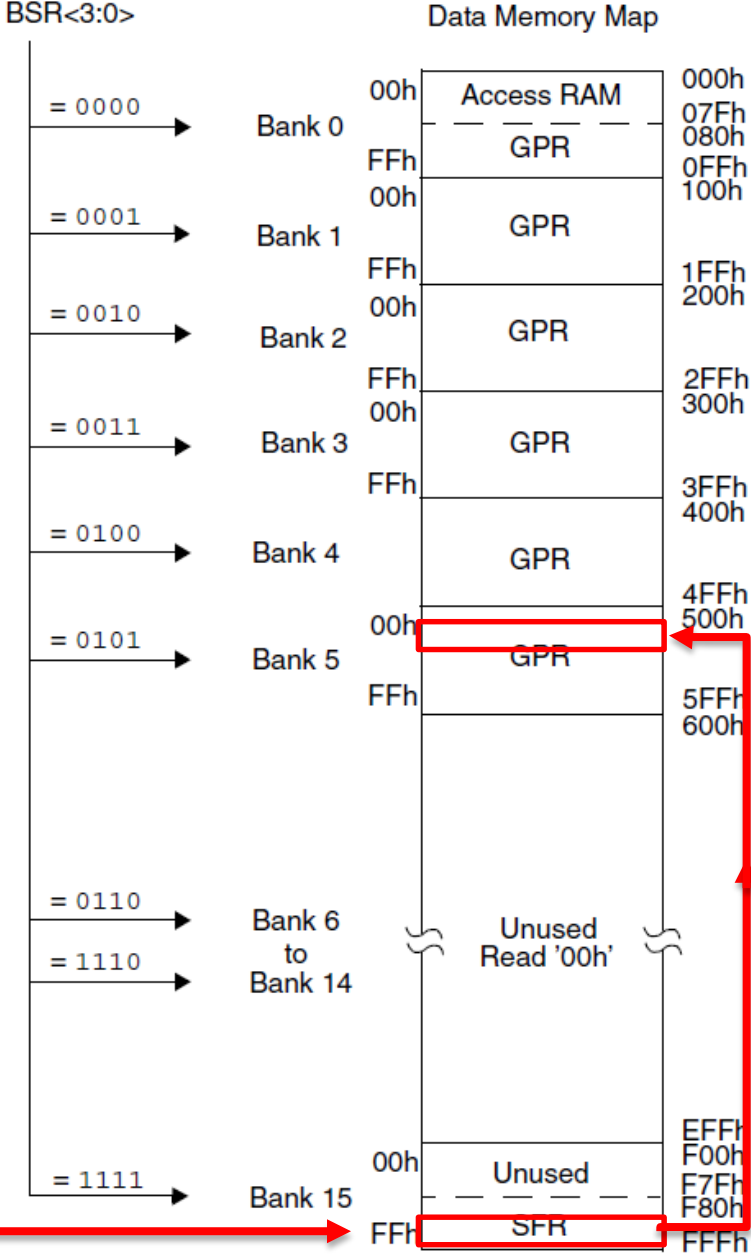
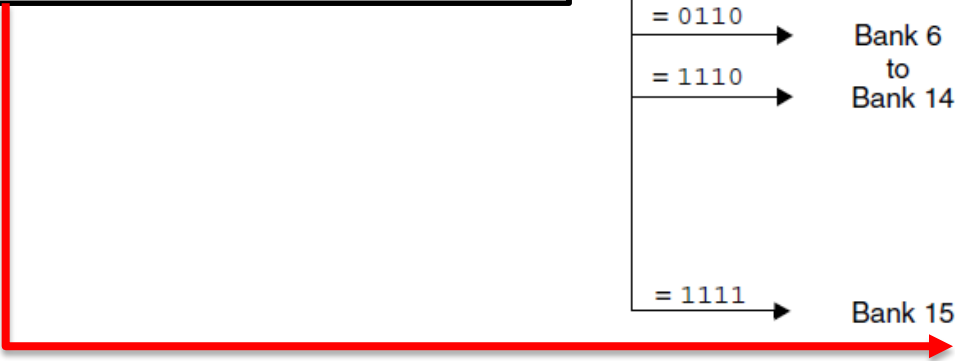
Register Indirect Addressing

- A special register (NOT SFR) is used as a pointer (actually 3)
- FSRs (File Select Register) are 12-bit registers:
 - **FSR0, FSR1, FSR2**
 - Each is represented by two SFRs, e.g., FSR0 has FSR0L and FSR0H
- **LFSR 1, 0x030** ; load 0x30 into FSR 1
- **LFSR 0, 0x130** ; load 0x130 into FSR 0
- The file register that the FSR is pointing to can be then reached in **INDF0, INDF1, and INDF2**, respectively
- **LFSR 0, 0x130**
- **MOVWF INDF0** ; contents of W moved to fileReg 0x130



```

MOVLW 28           ;28 → WREG
LFSR 1, 0x52F      ;load 0x52F into FSR 1
MOVWF INDF1       ;Copy contents of WREG into
                    ;RAM location held in FSR1
  
```





FSR0-2 Registers Used for Register Indirect

- Each FSR0-2 register is 12 bits thus consisting of two one byte file registers.
- The low order 8 bits are in one byte (FSRxL) and the upper 4 bits in the low order bits (or nibble) of the second byte (FSRxH).

FE0h	BSR
FE1h	FSR1L
FE2h	FSR1H
FE3h	PLUSW1 ⁽³⁾
FE4h	PREINC1 ⁽³⁾
FE5h	POSTDEC1 ⁽³⁾
FE6h	POSTINC1 ⁽³⁾
FE7h	INDF1 ⁽³⁾
FE8h	WREG
FE9h	FSR0L
FEAh	FSR0H

FD7h	TMR0H
FD8h	STATUS
FD9h	FSR2L
FDAh	FSR2H
FDBh	PLUSW2 ⁽³⁾
FDC0h	PREINC2 ⁽³⁾



Register Indirect Addressing Advantages

- Can now access data dynamically instead of just statically (Direct Add. Mode)
- We can simply increment the pointer
== incrementing a file register (**INCF f, 1**)
- Effective for sequential data (array/string) operations
- There are instructions for incrementing FSRs and clearing memory locations to which FSR points.

Assume that RAM locations 30–34H have a string of ASCII data, as shown below. Write a program to get each character and send it to Port B one byte at a time. Show the program using:

30 = ('H')

31 = ('E')

32 = ('L')

33 = ('L')

34 = ('O')

Assume that RAM locations 30–34H have a string of ASCII data, as shown below. Write a program to get each character and send it to Port B one byte at a time. Show the program using:

(a) Using direct addressing mode

CLRF	TRISB	;make Port B an output	30 = ('H')
MOVFF	0x30, PORTB	;copy contents of loc 0x30 to PB	31 = ('E')
MOVFF	0x31, PORTB		32 = ('L')
MOVFF	0x32, PORTB		33 = ('L')
MOVFF	0x33, PORTB		34 = ('O')
MOVFF	0x34, PORTB		

Assume that RAM locations 30–34H have a string of ASCII data, as shown below. Write a program to get each character and send it to Port B one byte at a time. Show the program using:

(a) Using direct addressing mode

CLRF	TRISB	;make Port B an output	30 = ('H')
MOVFF	0x30, PORTB	;copy contents of loc 0x30 to PB	31 = ('E')
MOVFF	0x31, PORTB		32 = ('L')
MOVFF	0x32, PORTB		33 = ('L')
MOVFF	0x33, PORTB		34 = ('O')
MOVFF	0x34, PORTB		

(b) Using register indirect mode

	COUNTREG EQU 0x20	;fileReg loc for counter	
	CNTVAL EQU 5	;counter value	
	CLRF TRISB	;make Port B an output (TRSI = 0 = out)	
	MOVLW CNTVAL	;WREG = 5	
	MOVWF COUNTREG	;load the counter, Count = 5	
	LFSR 2, 0x30	;load pointer. FSR2 = 30H, RAM address	
B3	MOVF INDF2, W	;copy RAM loc FSR2 points at to WREG	
	MOVWF PORTB	;copy WREG to PORTB	
	INCF FSR2L	;increment FSR2 to point at next loc	
	DECF COUNTREG, F	;decrement counter	
	BNZ B3	;loop until counter = zero	



Useful Instruction for Work with the FSR Registers

- **INDF_n** after operation, the FSR_n stays the same
 - `CLRF INDF1` ;clears fileReg pointed to by FSR1, FSR1 unchanged
- **POSTINC_n** after operation, the FSR_n is incremented
 - `MOVWF POSTINC2` ;copy WREG to fileReg pointed..., FSR2++
- **PREINC_n** FSR_n is incremented, then operation is performed
 - `ADDWF PREINC0` ;FSR0++, new pointer address added to WREG
- **POSTDEC_n** after operation, the FSR_n is decremented
 - `MOVWF POSTDEC1` ;copy WREG to fileReg pointed..., FSR1--
- **PLUSW_n** after operation on address of (FSR_n + WREG), FSR_n & W unchanged
 - `CLRF PLUSW1` ;clears fileReg pointed to by FSR1+WREG, FSR1 and WREG remain unchanged
 - Note: The auto-increment/decrement affects the entire 12 bits of FSR_n and has no effect on Status register. Thus FSR_n going from FFF to 000 will not be detected by the flags.



FSR Auto-increment

Write a program to clear 16 RAM locations starting at RAM address 60H.

```
COUNTREG EQU 0x10      ;fileReg loc for counter
CNTVAL EQU D'16'      ;counter value
    MOVLW  CNTVAL      ;WREG = 16
    MOVWF  COUNTREG    ;load the counter, Count = 16
    LFSR   1,0x60      ;load pointer. SFR0 = 40H, RAM address
B3   CLRF  POSTINC1    ;clear RAM, increment FSR1 pointer
    DECF  COUNTREG,F  ;decrement counter
    BNZ   B3          ;loop until counter = zero
```



FSR Auto-increment

- Write a program to copy a block of 5 bytes of data from RAM locations starting at 30H to RAM locations starting at 60H.

Solution:

```
COUNTREG EQU 0x10 ;fileReg loc for counter
CNTVAL EQU D'5' ;counter value
MOVLW CNTVAL ;WREG = 10
MOVWF COUNTREG ;load the counter, count = 10
LFSR 0,0x30 ;load pointer. FSR0 = 30H, RAM address
LFSR 1,0x60 ;load pointer. FSR1 = 60H, RAM address
B3 MOVF POSTINC0,W ;copy RAM to WREG and increment FSR0
MOVWF POSTINC1 ;copy WREG to RAM and increment FSR1
DECF COUNTREG,F ;decrement counter
BNZ B3 ;loop until counter = zero
```

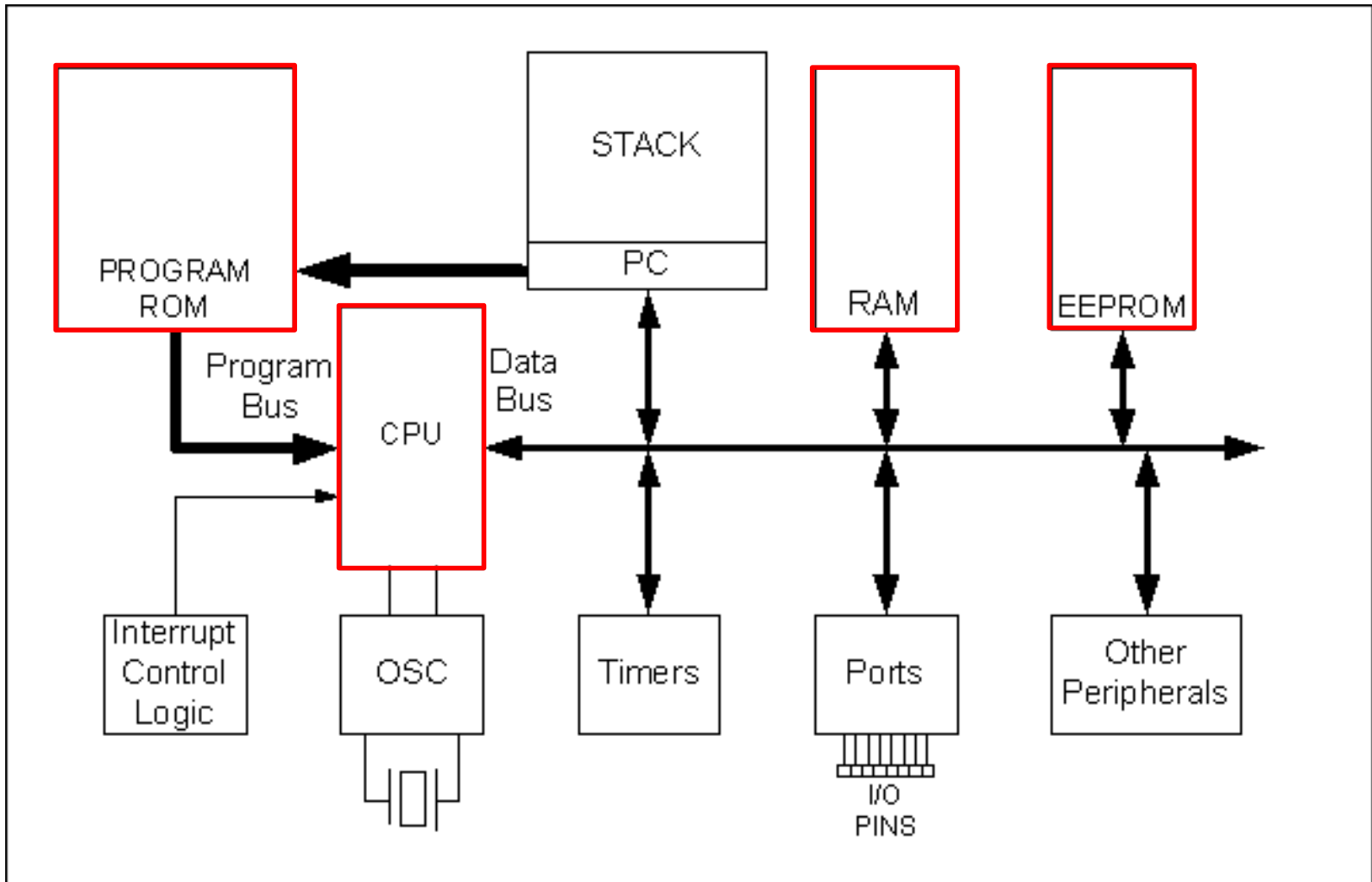
Before we run the above program.

30 = ('H') 31 = ('E') 32 = ('L') 33 = ('L') 34 = ('O')

After the program is run, the addresses 60–64H have the same data as 30–34H.

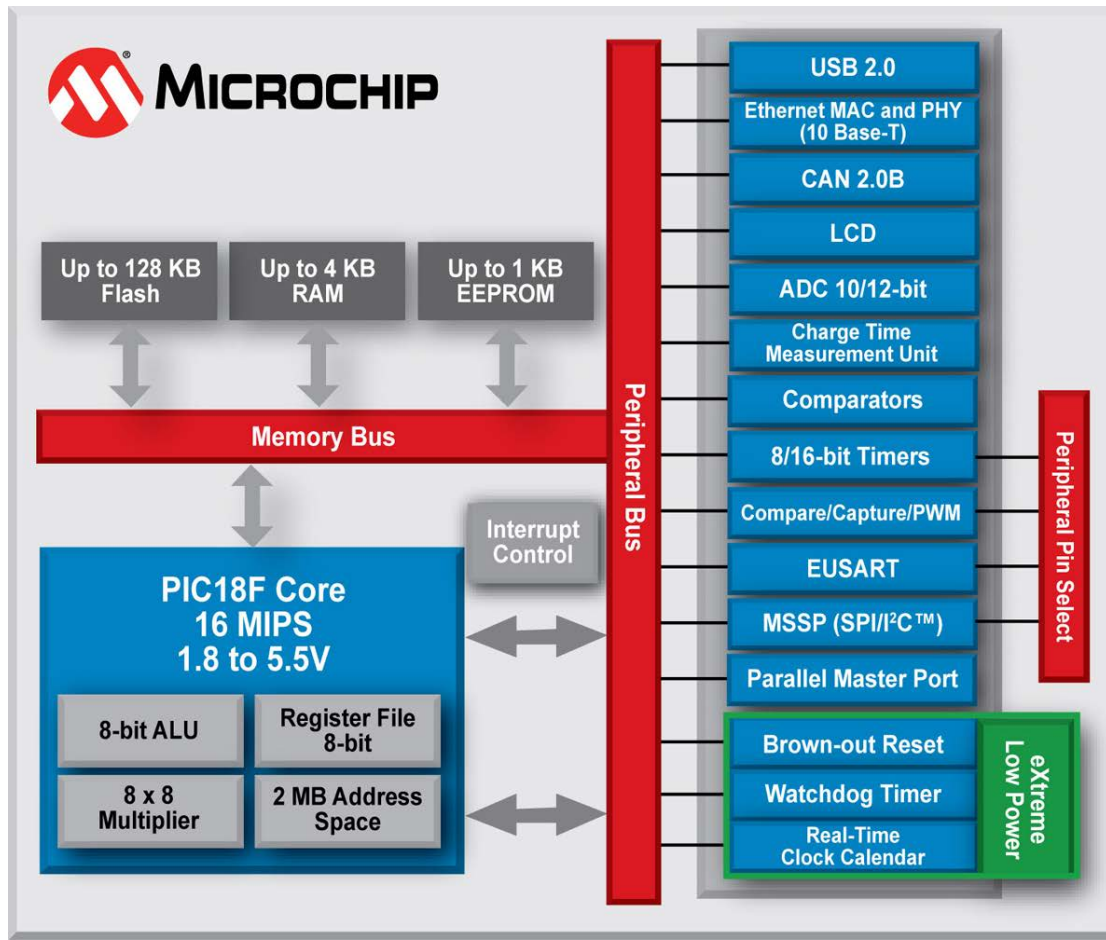
30 = ('H') 31 = ('E') 32 = ('L') 33 = ('L') 34 = ('O')
60 = ('H') 61 = ('E') 62 = ('L') 63 = ('L') 64 = ('O')

How Can the CPU Access Data?



Storing Data in Program Memory (ROM)

- The ROM (program memory) can be used to store constants (e.g., strings) to save available RAM space



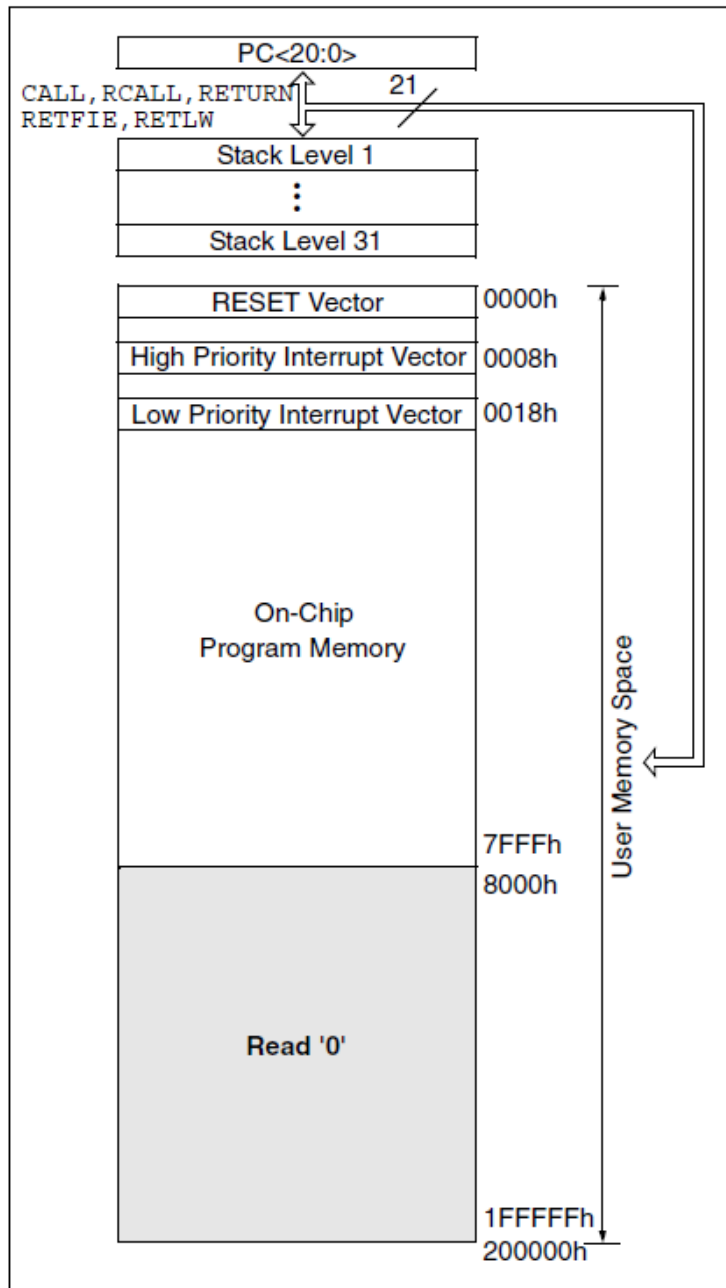


Memory Sizes on the 452

TABLE 1-1: DEVICE FEATURES

Features	PIC18F242	PIC18F252	PIC18F442	PIC18F452
Operating Frequency	DC - 40 MHz	DC - 40 MHz	DC - 40 MHz	DC - 40 MHz
Program Memory (Bytes)	16K	32K	16K	32K
Program Memory (Instructions)	8192	16384	8192	16384
Data Memory (Bytes)	768	1536	768	1536
Data EEPROM Memory (Bytes)	256	256	256	256
Interrupt Sources	17	17	18	18
I/O Ports	Ports A, B, C	Ports A, B, C	Ports A, B, C, D, E	Ports A, B, C, D, E
Timers	4	4	4	4
Capture/Compare/PWM Modules	2	2	2	2
	MSSP.	MSSP.	MSSP.	MSSP.

FIGURE 4-2: PROGRAM MEMORY MAP AND STACK FOR PIC18F452/252



INSTRUCTIONS IN PROGRAM MEMORY

Program Memory			LSB = 1	LSB = 0	Word Address ↓
Byte Locations →					000000h
					000002h
					000004h
					000006h
Instruction 1:	MOVLW	055h	0Fh	55h	000008h
Instruction 2:	GOTO	000006h	EFh	03h	00000Ah
			F0h	00h	00000Ch
Instruction 3:	MOVFF	123h, 456h	C1h	23h	00000Eh
			F4h	56h	000010h
					000012h
					000014h



DB – Define Byte or “Declare Data of One Byte”

- Assembler directive **DB** can be used to store/write bytes in ROM (program mem)
 - 8-bit chunks
 - Fixed data

```
                ORG    500H                ;must be even address
DATA1          DB     D'28'
DATA2          DB     0x39
```

```
                ORG    510H                ;must be even address
DATA3          DB     'H','E','L','L','O','1'
```

```
                ORG    520H                ;must be even address
DATA4          DB     "Hello World"
```



DB – Define Byte or “Declare Data of One Byte”

```
;MY DATA IN ROM
    ORG 500H                ;notice it must be an even address
DATA1 DB D'28'             ;DECIMAL(1C in hex)
DATA2 DB B'00110101'      ;BINARY (35 in hex)
DATA3 DB 0x39              ;HEX

    ORG 510H                ;notice it must be an even address
DATA4 DB 'Y'               ;single ASCII char
DATA5 DB '2','0','0','5'  ;ASCII numbers

    ORG 518H                ;notice it must be an even address
DATA6 DB "Hello ALI"      ;ASCII string
    END
```

DATA1	DATA2	DATA3
500 = (1C)	501 = (35)	502 = (39)

DATA4	DATA5			
510 = (59)	511 = (32)	512 = (30)	513 = (30)	514 = (35)
Y	2	0	0	5

DATA6				
518 = (48)	519 = (65)	51A = (6C)	51B = (6C)	51C = (6F)
H	e	l	l	o
51D = (20)	51E = (41)	51F = (4C)	520 = (49)	
SPACE	A	L	I	



DATA vs. DB

- Can use **DATA** directive for larger values
- **DB**: 0 – 255 (0x00 – 0xFF)
- **DATA**: 0 – 65,535 (0x00 – 0xFFFF)

MPLAB Ex.



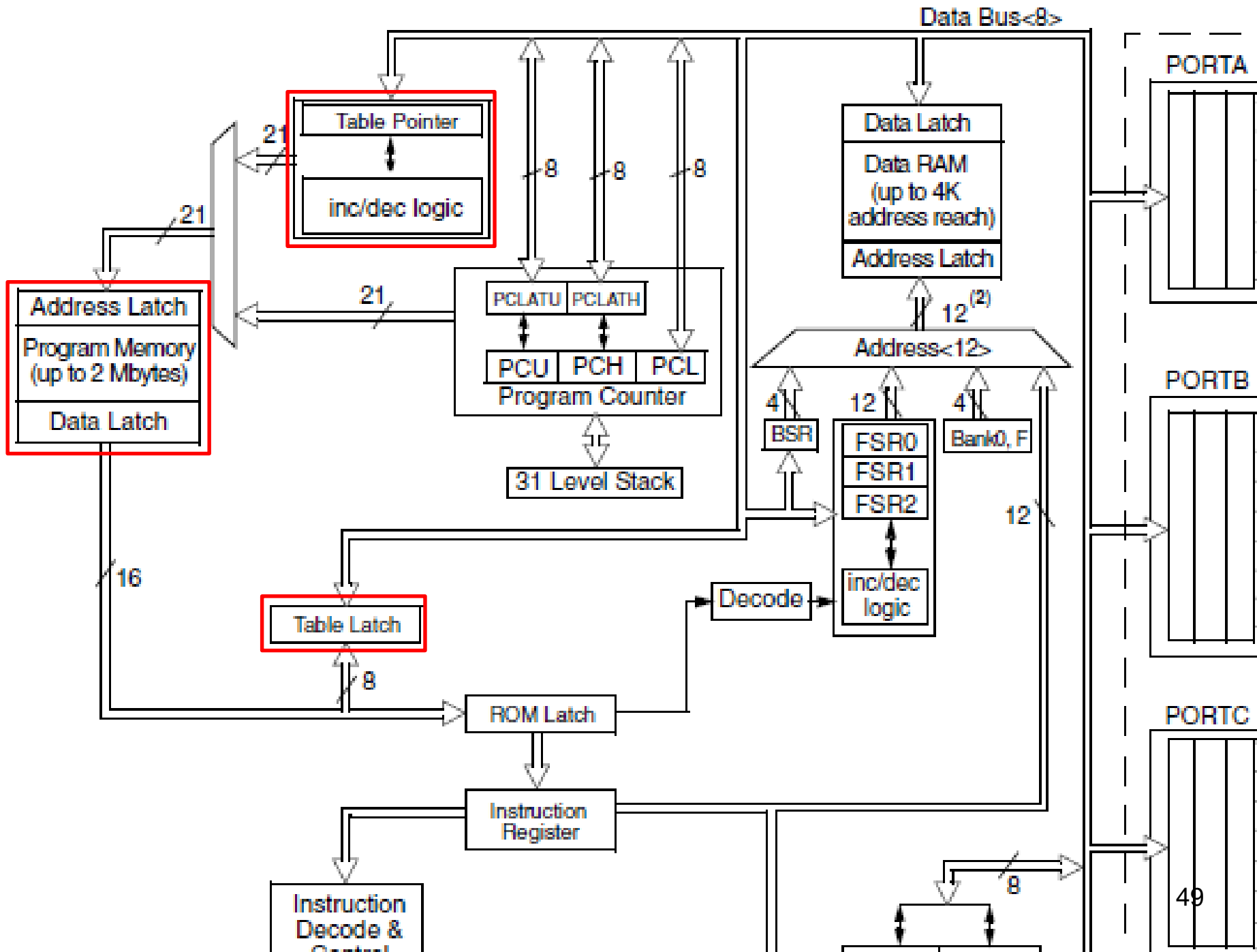
Lookup Tables

- Instead of calculating, sometimes storing lookup tables is more efficient
 - (cosine/sine tables, square tables, etc.)
- Lookup tables can be stored as instructions in the ROM
- **RETLW K** is a return from subroutine command that copies K into WREG as well. This can be used easily for lookup tables.



Reading Data from ROM

- Register indirect ROM addressing, i.e., accessing ROM is done through SFR registers
 - Known as *table processing*
- **TBLPTR** is a 21 bit register pointing to the data accessed in ROM
 - (TBLPTRU, TBLPTRH, TBLPTRL)
- **TBLAT** (table latch) is used to copy/hold the data pointed by TBLPTR, once instructed





Reading Data from ROM (cont'd)

Table 6-3: PIC18 Table Read Instructions

Instruction	Function	Description
TBLRD*	Table read	After read, TBLPTR stays the same
TBLRD*+	Table read with post-inc.	Reads and increments TBLPTR
TBLRD*-	Table read with post-dec.	Reads and decrements TBLPTR
TBLRD+*	Table read with pre-inc.	Increments TBLPTR and then reads

Note: The byte of data is read into the TABLAT_{ch} register from code space pointed to by TBLPTR.

Assuming that program ROM space starting at 250H contains "USA", write a program to send all the characters to Port B one byte at a time.

```

    ORG    0000H           ;burn into ROM starting at 0
    MOVLW 0x50           ;WREG = 50 low-byte addr
    MOVWF TBLPTRL        ;look-up table low-byte addr
    MOVLW 0x02           ;WREG = 2, high-byte addr
    MOVWF TBLPTRH        ;look-up table high-byte addr
    CLRF  TRISB          ;TRISB = 00 (Port B as output)
B7   TBLRD*+             ;bring in next byte and inc TBLPTR
    MOVF  TABLAT,W        ;copy to WREG (Z = 1, if null)
    BZ    EXIT           ;is it null char? exit if yes
    MOVWF PORTB          ;send it to Port B
    BRA  B7              ;continue
EXIT GOTO  EXIT

    ORG    0x250
MYDATA DB "USA",0       ;notice null
    END
```

Write a program to get the x value from Port B and send $x^2 + 2x + 3$ to Port C. Assume PB3–PB0 has the x value of 0–9. Use a look-up table instead of a multiply instruction.

Solution:

```
ORG    0
SETF   TRISB      ;TRISB = FFh (Port B as input)
CLRF   TRISC      ;TRISC = 00 (Port C as output)
B1     MOVF   PORTB,W    ;read x from Port B into WREG
      ANDLW  0x0F      ;mask upper bits
      CALL  XSQR_TABLE  ;get x2 from the look-up table
      MOVWF PORTC      ;copy it to Port C
      BRA   B1         ;continue

XSQR_TABLE
      MULLW 0x2        ;align it for even address
      MOVFF PRODL, WREG ;put it into WREG for indexing
      ADDWF PCL        ;PCL = PCL + WREG
      RETLW D'3'       ;(0)2 + 2(0) + 3 = 3
      RETLW D'6'       ;(1)2 + 2(1) + 3 = 6
      RETLW D'11'      ;(2)2 + 2(2) + 3 = 11
      RETLW D'18'      ;(3)2 + 2(3) + 3 = 18
      RETLW D'27'      ;(4)2 + 2(4) + 3 = 27
      RETLW D'38'      ;(5)2 + 2(5) + 3 = 38
      RETLW D'51'      ;(6)2 + 2(6) + 3 = 51
      RETLW D'66'      ;(7)2 + 2(7) + 3 = 66
      RETLW D'83'      ;(8)2 + 2(8) + 3 = 83
      RETLW D'102'     ;(9)2 + 2(9) + 3 = 102
      END
```

MPLAB Ex.



Macros

- Macro is used for referencing the same group of instructions repeatedly
 - Macro == sequence of instructions
- Thus do not have to repeat/write the instructions each time instruction group are used
 - For useful non-standard operations
- Place/define “above” your main code (ORG 0)
- Macros can call other macros or itself recursively
 - Max 16 nested macro calls



Macros

name	MACRO arg1, ... , argN	;200 character limit
	...	;macro body
	...	;macro body
	ENDM	

E.g.: create a copy literal to file register operation

```
MOVLF    MACRO    k, myReg
          MOVLW    k
          MOVWF    myReg
          ENDM
```

```
ORG 0
```

```
...
```

```
MOVLF 0x55, 0x20
MOVLF 0xFF, PORTB
```



Macros – LOCAL Directive

- Must declare **labels** in the macro's body as LOCAL to prevent conflicts
- LOCAL **must** be used right after macro dir.

The LOCAL directive can be used to declare all names and labels at once as follows:

```
LOCAL      name1, name2, name3
```

or one at a time as:

```
LOCAL      name1
```

```
LOCAL      name2
```

```
LOCAL      name3
```



Macros – LOCAL Directive

```
DELAY_2 MACRO V1, V2, R1, R2
    LOCAL BACK
    LOCAL AGAIN
    MOVLW V2
    MOVWF R2
AGAIN MOVLW V1
    MOVWF R1
BACK  NOP
      NOP
      NOP
      NOP
      DECF  R1, F
      BNZ  BACK
      DECF  R2, F
      BNZ  AGAIN
    ENDM
```




```
;------  
;Program 6-4: toggling Port B using macros  
    #include P18F458.INC  
  
;------sending data to fileReg macro  
MOVLF MACRO K, MYREG  
    MOVLW K  
    MOVWF MYREG  
ENDM  
  
;------time delay macro  
DELAY_1 MACRO V1, TREG  
    LOCAL BACK  
    MOVLW V1  
    MOVWF TREG  
BACK  NOP  
      NOP  
      NOP  
      NOP  
      DECF  TREG, F  
      BNZ  BACK  
ENDM  
  
;------program starts  
    ORG    0  
    CLRF   TRISB           ;Port B as an output  
  
OVER  MOVLF   0x55, PORTB  
      DELAY_1 0x200, 0x10  
      MOVLF   0xAA, PORTB  
      DELAY_1 0x200, 0x10  
      BRA    OVER  
      END  
  
;------end of file
```

MPLAB Ex.



Macro vs. Subroutine

- **Macros**

- Increase overall code size
 - 10-instruction macro called 10 times = 100 total instructions
- Allows in-line arguments in macro call
- No return values

- **Subroutines**

- Fixed code size
- No in-line arguments when calling a subroutine
- Return value is “possible”
 - *retlw (return with literal in WREG)*
- Uses stack space
 - Too many nested calls can cause stack issues



#INCLUDE Directives

- Use INCLUDE directive to reference macros/code defined in other files
- The specified file is read in a source code

4.42.1 Syntax

Preferred:

```
#include include_file  
#include "include_file"  
#include <include_file>
```

Supported:

```
include include_file  
include "include_file"  
include <include_file>
```

```
#include P18F452.inc
```

```
#include <MyMacros.mac>
```

```
#include "C:\Program Files... .h"
```

Assembler will look for file in...

1. current working directory
2. source file directory
3. MPASM assembler executable directory



Modules

- With having the main procedure and subroutines in the same file...
 - If one subroutine fails, all must be rewritten
- Treat each subroutine as its own program
 - Known as “modules”
 - Each a separate file (.o or .asm file)
 - Assembled and tested independently
 - All brought together (linked) to form a single program



Modules Directives

- **EXTERN**
 - Notifies assembler/linker that certain names and variables are not defined in the present module but in another (externally)
- **GLOBAL**
 - Notifies assembler/linker that certain names and variables may be used by other outside (external) modules
- **GLOBAL (public)** allows the assembler and linker to match it with its **EXTERN** counterpart(s)



Module Example

```
-----  
;PROG 6-6: MAIN.ASM - CALCULATING AND TESTING CHECKSUM BYTE  
#include P18F458.INC  
  
RAM_ADDR EQU 40H  
COUNTREG EQU 0x20 ;fileReg loc for counter  
CNTVAL EQU 4 ;counter value  
CNTVAL1 EQU 5 ;counter value  
  
EXTERN CAL_CHKSUM  
EXTERN TEST_CHKSUM  
  
PGM CODE  
;-----main program  
ORG 0  
CALL COPY_DATA ;this subroutine is in this file  
CALL CAL_CHKSUM ;this sub is in external file  
CALL TEST_CHKSUM ;this sub is in external file  
BRA $
```

```
-----  
;PROG 6-6: CALCCSB.ASM - CALCULATING CHECKSUM BYTE  
#include P18F458.inc  
  
RAM_ADDR EQU 40H  
COUNTREG EQU 0x20 ;fileReg loc for counter  
CNTVAL EQU 4 ;counter value  
CNTVAL1 EQU 5 ;counter value  
  
GLOBAL CAL_CHKSUM  
  
PGM CODE ;we use this to inform the linker that  
;the code segment has the name PGM  
  
CAL_CHKSUM  
MOVLW CNTVAL ;WREG = 4  
MOVWF COUNTREG ;load the counter  
LFSR 0,RAM_ADDR ;load pointer. FSR0 = 40H  
CLRF WREG  
C2 ADDWF POSTINC0,W ;add RAM to WREG and increment FSR0  
DEC F WREG ;decrement counter  
BNZ C2 ;loop until counter = zero  
XORLW 0xFF ;1's comp  
ADDLW 1 ;2' compl  
MOVWF POSTINC0  
RETURN  
END
```



Module Example

```
-----  
;PROG 6-6: MAIN.ASM - CALCULATING AND TESTING CHECKSUM BYTE  
#include P18F458.INC  
  
RAM_ADDR EQU 40H  
COUNTREG EQU 0x20 ;fileReg loc for counter  
CNTVAL EQU 4 ;counter value  
CNTVAL1 EQU 5 ;counter value  
  
EXTERN CAL_CHKSUM  
EXTERN TEST_CHKSUM  
  
PGM CODE  
;-----main program  
ORG 0  
CALL COPY_DATA ;this subroutine is in this file  
CALL CAL_CHKSUM ;this sub is in external file  
CALL TEST_CHKSUM ;this sub is in external file  
BRA $
```

```
-----  
;PROG 6-6: CALCCSB.ASM - CALCULATING CHECKSUM BYTE  
#include P18F458.inc  
  
RAM_ADDR EQU 40H  
COUNTREG EQU 0x20 ;fileReg loc for counter  
CNTVAL EQU 4 ;counter value  
CNTVAL1 EQU 5 ;counter value  
  
GLOBAL CAL_CHKSUM  
  
PGM CODE ;we use this to inform the linker that  
;the code segment has the name PGM  
  
CAL_CHKSUM  
MOVLW CNTVAL ;WREG = 4  
MOVWF COUNTREG ;load the counter  
LFSR 0,RAM_ADDR ;load pointer. FSR0 = 40H  
CLRF WREG  
C2 ADDWF POSTINC0,W ;add RAM to WREG and increment FSR0  
DECW COUNTREG,F ;decrement counter  
BNZ C2 ;loop until counter = zero  
XORLW 0xFF ;1's comp  
ADDLW 1 ;2' compl  
MOVWF POSTINC0  
RETURN  
END
```



Data RAM in C

- PIC18 has 4K of RAM (file registers)
- Compiler has chosen variable (data) locations automatically so far
- **NEAR** and **FAR** storage qualifiers

Storage qualifier	RAM
near	In access bank
far	Anywhere in data RAM file register (default)

```
near unsigned char myArray[100];
```

```
...
```

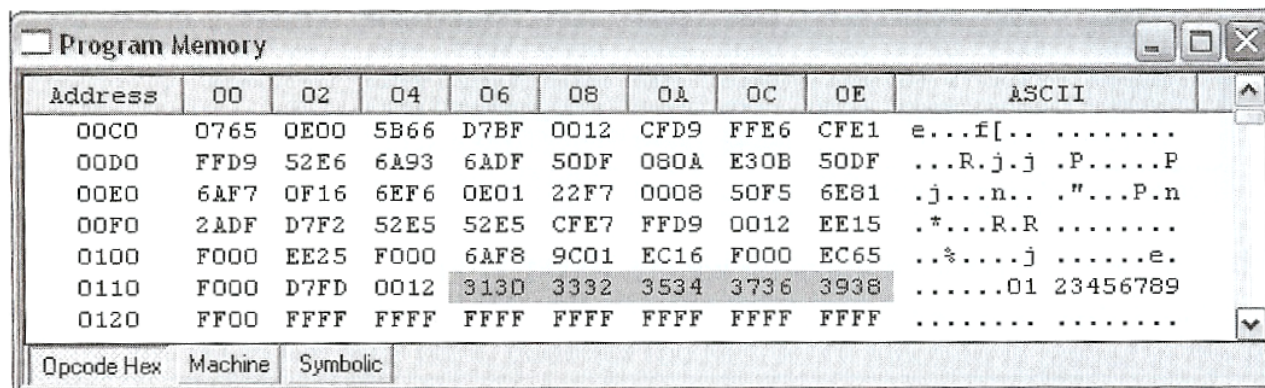
```
far unsigned char myArray[100];
```


Working With Data in ROM Using C

Use the keyword *rom*

```
#include <P18F458.h>
rom const char mynum[]= "0123456789"; //uses program
                                     //ROM space for fixed (constant) data

void main(void)
{
    unsigned char z;
    TRISB = 0;           //make Port B an output
    for(z=0; z<10; z++)
        PORTB=mynum[z];
}
```



Address	00	02	04	06	08	0A	0C	0E	ASCII
00C0	0765	0E00	5B66	D7BF	0012	CFD9	FFE6	CFE1	e...f[.
00D0	FFD9	52E6	6A93	6ADF	50DF	080A	E30B	50DF	...R.j.j .P....P
00E0	6AF7	0F16	6EF6	0E01	22F7	0008	50F5	6E81	.j...n.. ."...P.n
00F0	2ADF	D7F2	52E5	52E5	CFE7	FFD9	0012	EE15	.*...R.R
0100	F000	EE25	F000	6AF8	9C01	EC16	F000	EC65	..~....je.
0110	F000	D7FD	0012	313D	3332	3534	3736	393801 23456789
0120	FF00	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF

Figure 7-14. Fixed Data Placed in Program ROM as Shown in MPLAB

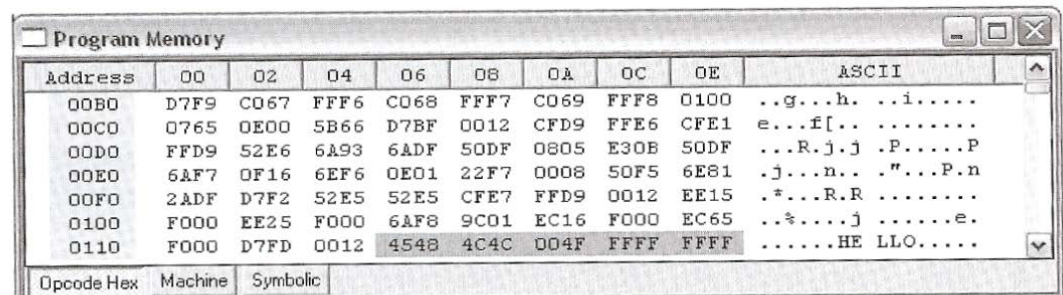
Using Near

- *near* and *far* can be used to control where the data in the ROM should be (in low 64K or anywhere)
- More efficient use of code space

Table 7-6: NEAR and FAR Usage for ROM

Storage qualifier	ROM
near	In program space of 0000–FFFFH (64 kB)
far	In program space of 000000–1FFFFFFH (2 MB)

```
//Program 7-2A
#include <P18F458.h>
    near rom const char mydata[] = "HELLO"; //program ROM data
void main(void)
{
    unsigned char z;
    TRISB = 0; //make Port B an output
    for(z=0;z<5;z++)
        PORTB = mydata[z];
}
```



Address	00	02	04	06	08	0A	0C	0E	ASCII
00B0	D7F9	C067	FFF6	C068	FFF7	C069	FFF8	0100	..g...h. .i.....
00C0	0765	0E00	5B66	D7BF	0012	CFD9	FFE6	CFE1	e...f[.
00D0	FFD9	52E6	6A93	6ADF	50DF	0805	E30B	50DF	...R.j.j .P....P
00E0	6AF7	0F16	6EF6	0E01	22F7	0008	50F5	6E81	.j...n.. ."...P.n
00F0	2ADF	D7F2	52E5	52E5	CFE7	FFD9	0012	EE15	*...R.R
0100	F000	EE25	F000	6AF8	9C01	EC16	F000	EC65	..*...je.
0110	F000	D7FD	0012	4548	4C4C	004F	FFFF	FFFFHE LLO.....

Notice 4 digits for address (0000–FFFF)



Placing ASM Code/Data in C

- Assembly can be directly embedded in C code
 - **#asm** and **#endasm**
 - or **asm()**;

```
unsigned int var;

void main(void)
{
    var = 1;
    #asm          // like this...
        BCF 0,3
        BANKSEL(_var)
        RLF (_var)&07fh
        RLF (_var+1)&07fh
    #endasm

    // do it again the other way...
    asm("BCF 0,3");
    asm("BANKSEL _var");
    asm("RLF (_var)&07fh");
    asm("RLF (_var+1)&07fh");
}
```



#pragma

- In C, can put code or data at exact ROM or RAM address

2.9.1.1 SYNTAX

section-directive:

```
# pragma udata [attribute-list] [section-name [=address]]
| # pragma idata [attribute-list] [section-name [=address]]
| # pragma romdata [overlay] [section-name [=address]]
| # pragma code [overlay] [section-name [=address]]
```



Placing ASM Code in C at Specific Addresses

- To place code at specific ROM address
 - **ORG** (for ASM) **#pragma** (for C compiler)

```
#include <P18F458.h>
#pragma code main = 0x50 //place the main at ROM addr 0x50
void MSDelay(unsigned int);
void main(void)
{
    unsigned char mydata[] = "HELLO";
    unsigned char z;
    TRISB = 0; //make Port B an output
    for(z=0;z<5;z++)
    {
        PORTB = mydata[z];
        MSDelay(250);
    }
}

#pragma code MSDelay = 0x300 //place delay at ROM addr 0x300
void MSDelay(unsigned int itime)
{
    unsigned int i;
    unsigned char j;
    for(i=0;i<itime;i++)
        for(j=0;j<165;j++);
}
```



Placing ASM ROM Data in C at Specific Addresses

- To place data at specific ROM address
 - **ORG** (for ASM) **#pragma** (for C compiler)

```
#include <P18F458.h>
#pragma romdata mydata = 0x200 //place mydata at ROM addr 0x200
    near rom const char mydata[] = "HELLO"; //ROM data
void main(void)
{
    unsigned char z;
    TRISB = 0; //make Port B an output
    for(z=0; z<5; z++)
        PORTB = mydata[z];
}
```

Address	00	02	04	06	08	0A	0C	0E	ASCII
01F0	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF
0200	4548	4C4C	004F	FFFF	FFFF	FFFF	FFFF	FFFF	HELLO...
0210	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF
0220	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF
0230	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF

Opcode Hex Machine Symbolic

Idata Example

```
//Program 7-9 (using idata)
#include <P18F458.h>
#pragma idata mydata = 0x150
unsigned char mydata[] = "HELLO"; //RAM data
void main(void)
{
    unsigned char z;
    TRISB = 0; //make Port B an output
    for(z=0; z<5; z++)
        PORTB = mydata[z];
}
```

We can verify the above concept by simulating the program on the MPLAB and examining the RAM at address 0x150.

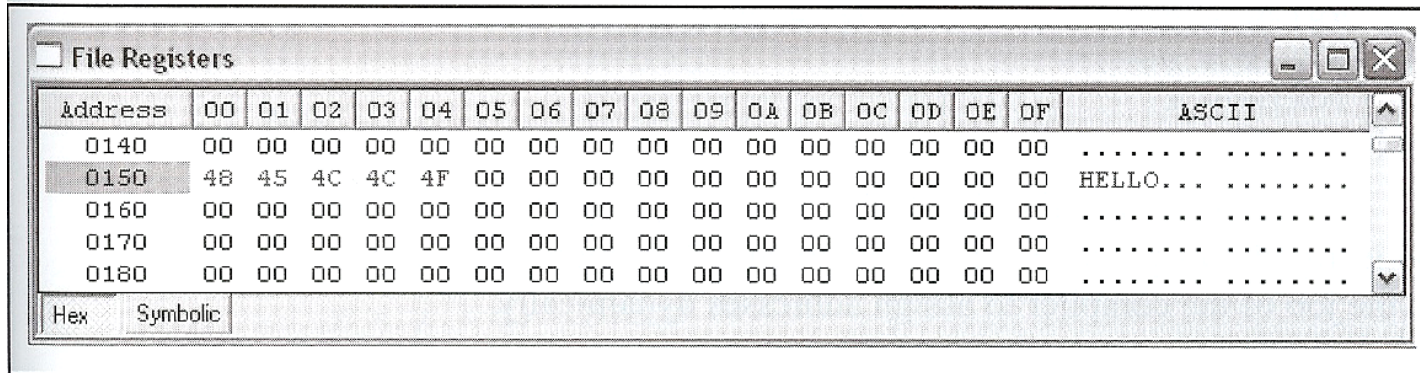


Figure 7-22. Screen Shot for Program 7-9



Overlay Variables

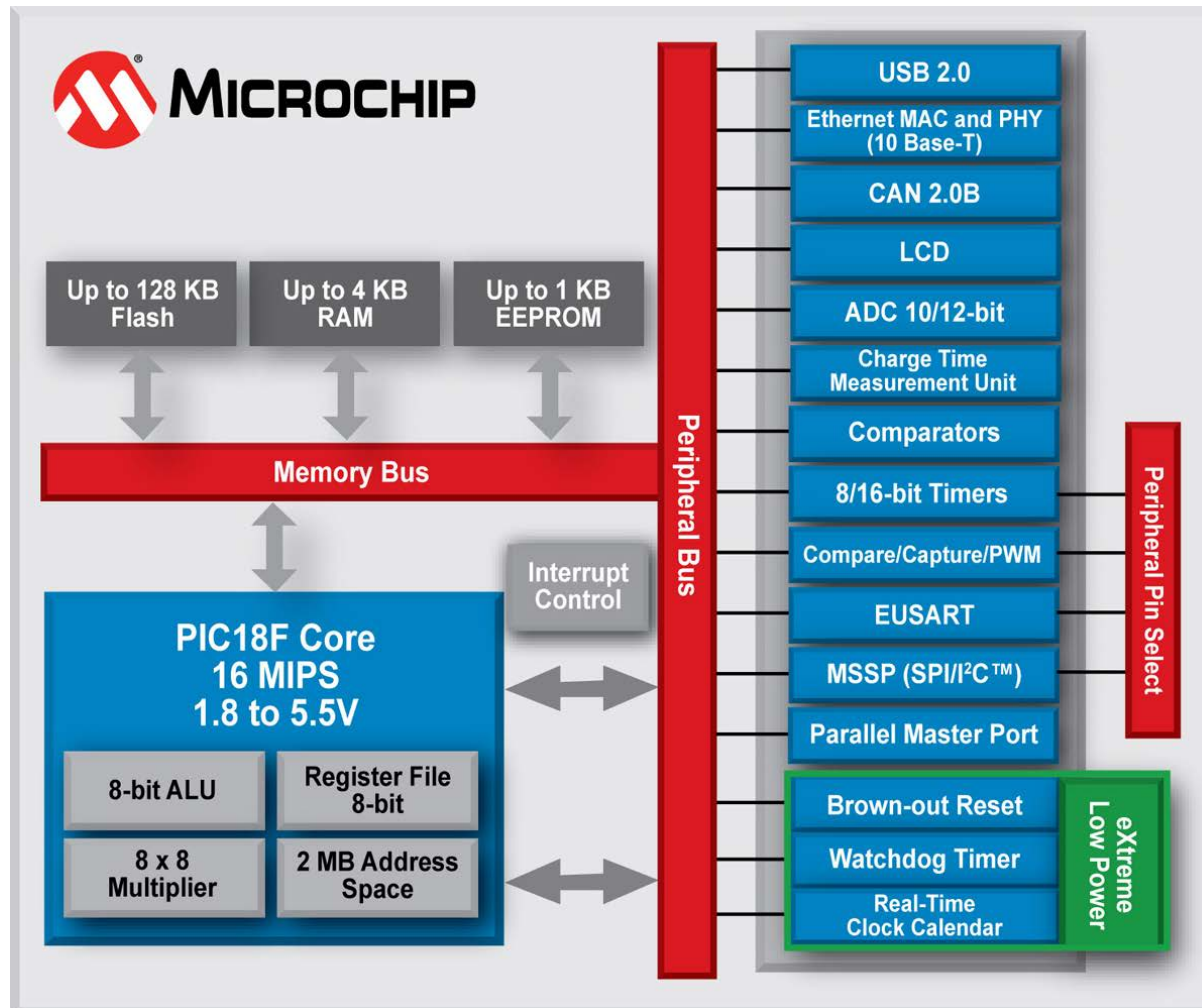
- Two variables can use the same space if they are not used at the same time
- The compiler may decide to use the same physical location for variables `x` and `y` in the following two functions:

```
unsigned char functionA(void)
{
    overlay unsigned int x=0;
    x++;
    return x;
}
```

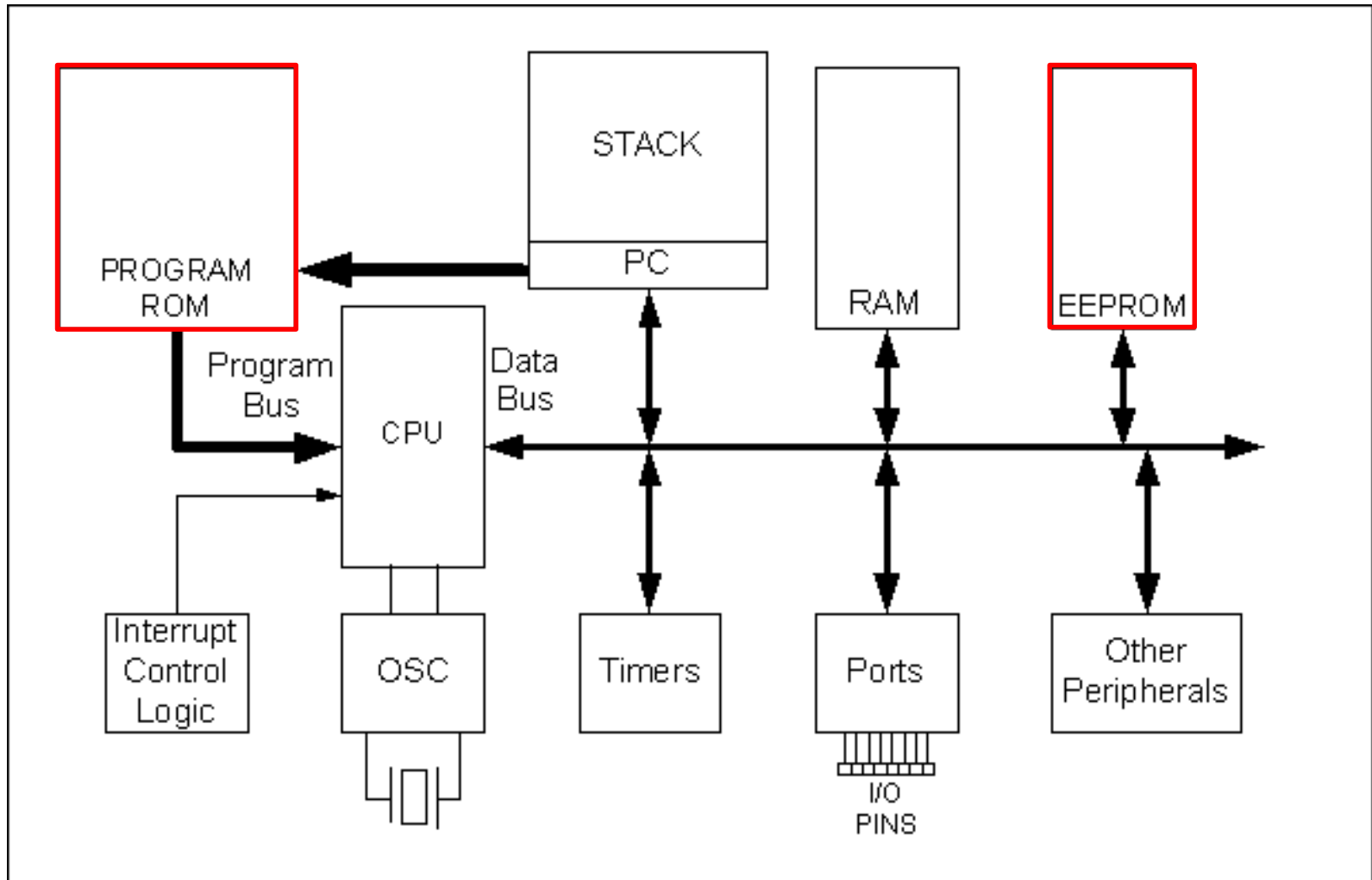
```
unsigned char functionB(void)
{
    overlay unsigned int y=5;
    y--;
    return y;
}
```

- What would happen if functionA called functionB?

Non-Volatile Memory in PICs



Non-Volatile Memory in PICs





Non-Volatile Memory in PICs

- In addition to the volatile (but flip-flop based and thus stable) SRAM, there are two kinds of non-volatile memory integrated into PIC18Fs:
 - **Flash EPROM**, this is the memory in which firmware is uploaded
 - **EEPROM** memory, for storing variables that should not be reset
- Why not only use one kind?
 - **EEPROM** is more expensive, but it can be rewritten byte-by-byte.
 - **Flash EPROM**, before rewriting, requires a block erase (flashing it) and can only be written one block at a time.
- In general, electrically erasable programmable ROM can only be rewritten a limited amount of times before being damaged (~100k times).



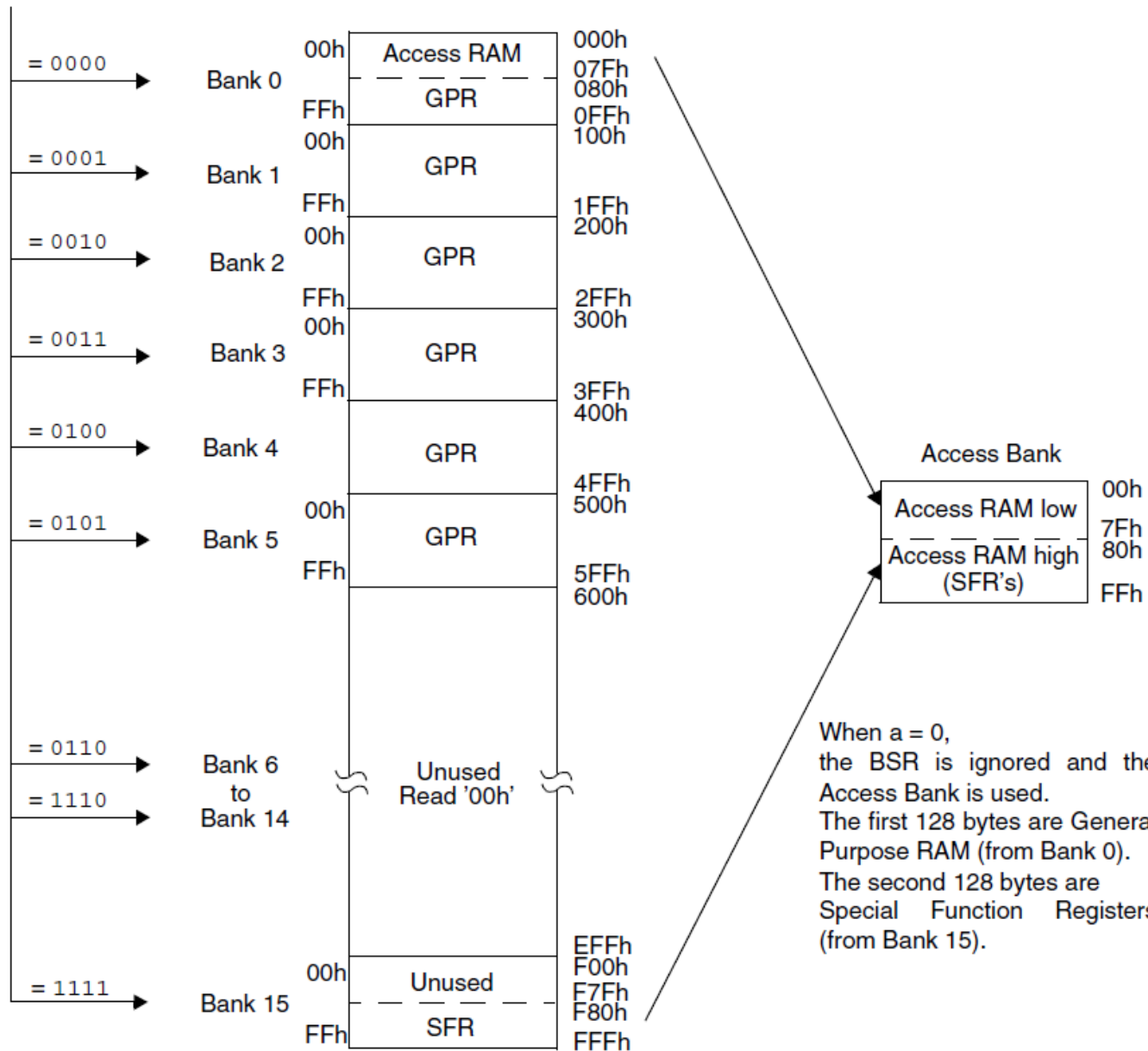
EEPROM

- Four SFRs used for EEPROM read/write
 - **EECON1**
 - Control register for EEPROM access
 - **EECON2**
 - “Dummy” register used for writing sequence
 - **EEDATA**
 - Holds the 8-bit data for writing to or reading from
 - **EEADR**
 - Holds the EEPROM location (address of data)



BSR<3:0>

Data Memory Map



When a = 0, the BSR is ignored and the Access Bank is used. The first 128 bytes are General Purpose RAM (from Bank 0). The second 128 bytes are Special Function Registers (from Bank 15).

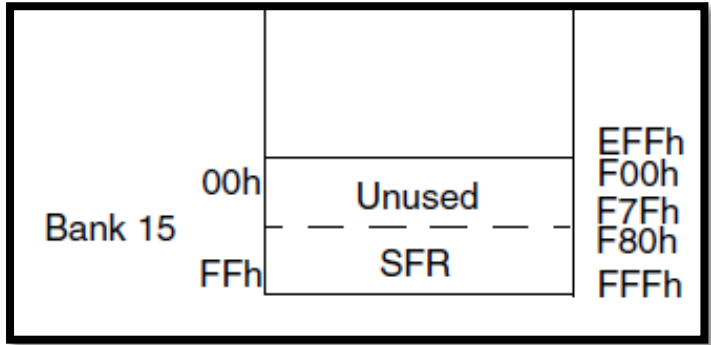
When a = 1, the BSR is used to specify the RAM location that the instruction uses.



SFRs for EEPROM

TABLE 4-1: SPECIAL FUNCTION REGISTER MAP

Address	Name	Address	Name	Address	Name	Address	Name
FFFh	TOSU	FDFh	INDF2 ⁽³⁾	FBFh	CCPR1H	F9Fh	IPR1
FFEh	TOSH	FDEh	POSTINC2 ⁽³⁾	FBEh	CCPR1L	F9Eh	PIR1
FFDh	TOSL	FDDh	POSTDEC2 ⁽³⁾	FBDh	CCP1CON	F9Dh	PIE1
FFCh	STKPTR	FDCh	PREINC2 ⁽³⁾	FBCh	CCPR2H	F9Ch	—
FFBh	PCLATU	FDBh	PLUSW2 ⁽³⁾	FBBh	CCPR2L	F9Bh	—
FFAh	PCLATH	FDAh	FSR2H	FBAh	CCP2CON	F9Ah	—
FF9h	PCL	FD9h	FSR2L	FB9h	—	F99h	—
FF8h	TBLPTRU	FD8h	STATUS	FB8h	—	F98h	—
FF7h	TBLPTRH	FD7h	TMR0H	FB7h	—	F97h	—
FF6h	TBLPTRL	FD6h	TMR0L	FB6h	—	F96h	TRISE ⁽²⁾
FF5h	TABLAT	FD5h	T0CON	FB5h	—	F95h	TRISD ⁽²⁾
FF4h	PRODH	FD4h	—	FB4h	—	F94h	TRISC
FF3h	PRODL	FD3h	OSCCON	FB3h	TMR3H	F93h	TRISB
FF2h	INTCON	FD2h	LVDCON	FB2h	TMR3L	F92h	TRISA
FF1h	INTCON2	FD1h	WDTCON	FB1h	T3CON	F91h	—
FF0h	INTCON3	FD0h	RCON	FB0h	—	F90h	—
FEFh	INDF0 ⁽³⁾	FCFh	TMR1H	FAFh	SPBRG	F8Fh	—
FEeh	POSTINC0 ⁽³⁾	FCEh	TMR1L	FAEh	RCREG	F8Eh	—
FEDh	POSTDEC0 ⁽³⁾	FCDh	T1CON	FADh	TXREG	F8Dh	LATE ⁽²⁾
FECh	PREINC0 ⁽³⁾	FCCh	TMR2	FACH	TXSTA	F8Ch	LATD ⁽²⁾
FEbh	PLUSW0 ⁽³⁾	FCBh	PR2	FABh	RCSTA	F8Bh	LATC
FEAh	FSR0H	FCAh	T2CON	FAAh	—	F8Ah	LATB
FE9h	FSR0L	FC9h	SSPBUF	FA9h	EEADR	F89h	LATA
FE8h	WREG	FC8h	SSPADD	FA8h	EEDATA	F88h	—
FE7h	INDF1 ⁽³⁾	FC7h	SSPSTAT	FA7h	EECON2	F87h	—
FE6h	POSTINC1 ⁽³⁾	FC6h	SSPCON1	FA6h	EECON1	F86h	—
FE5h	POSTDEC1 ⁽³⁾	FC5h	SSPCON2	FA5h	—	F85h	—
FE4h	PREINC1 ⁽³⁾	FC4h	ADRESH	FA4h	—	F84h	PORTE ⁽²⁾
FE3h	PLUSW1 ⁽³⁾	FC3h	ADRESL	FA3h	—	F83h	PORTD ⁽²⁾
FE2h	FSR1H	FC2h	ADCON0	FA2h	IPR2	F82h	PORTC
FE1h	FSR1L	FC1h	ADCON1	FA1h	PIR2	F81h	PORTB
FE0h	BSR	FC0h	—	FA0h	PIE2	F80h	PORTA





REGISTER 6-1: EECON1 REGISTER (ADDRESS FA6h)

R/W-x	R/W-x	U-0	R/W-0	R/W-x	R/W-0	R/S-0	R/S-0
EEPGD	CFGS	—	FREE	WRERR	WREN	WR	RD
bit 7						bit 0	

- bit 7 **EEPGD:** FLASH Program or Data EEPROM Memory Select bit
 1 = Access FLASH Program memory
 0 = Access Data EEPROM memory
- bit 6 **CFGS:** FLASH Program/Data EE or Configuration Select bit
 1 = Access Configuration or Calibration registers
 0 = Access FLASH Program or Data EEPROM memory
- bit 5 **Unimplemented:** Read as '0'
- bit 4 **FREE:** FLASH Row Erase Enable bit
 1 = Erase the program memory row addressed by TBLPTR on the next WR command
 (cleared by completion of erase operation)
 0 = Perform write only
- bit 3 **WRERR:** FLASH Program/Data EE Error Flag bit
 1 = A write operation is prematurely terminated
 (any MCLR or any WDT Reset during self-timed programming in normal operation)
 0 = The write operation completed
Note: When a WRERR occurs, the EEGPD or FREE bits are not cleared. This allows tracing of the error condition.
- bit 2 **WREN:** FLASH Program/Data EE Write Enable bit
 1 = Allows write cycles
 0 = Inhibits write to the EEPROM
- bit 1 **WR:** Write Control bit
 1 = Initiates a data EEPROM erase/write cycle or a program memory erase cycle or write cycle.
 (The operation is self-timed and the bit is cleared by hardware once write is complete. The WR bit can only be set (not cleared) in software.)
 0 = Write cycle to the EEPROM is complete
- bit 0 **RD:** Read Control bit
 1 = Initiates an EEPROM read
 (Read takes one cycle. RD is cleared in hardware. The RD bit can only be set (not cleared) in software. RD bit cannot be set when EEGPD = 1.)
 0 = Does not initiate an EEPROM read



Writing to EEPROM

1. Load **EEADR** with EEPROM location destination
2. Load **EEDATA** with data we want to write
3. Set **EECON1** configurations for writing
 $EEPGD = 0, CFGS = 0, WREN = 1$
4. Write 0x55 to **EECON2**
5. Write 0xAA to **EECON2**
6. Set **EECON1bits.WR = 1**
7. Wait until **WR** bit clears to 0



Reading From EEPROM

1. Load **EEADR** with EEPROM location source
2. Set **EECON1** configurations for reading
EEPGD = 0, CFGS = 0, RD = 1
3. Data is fetched and put into **EEDATA**
4. Copy **EEDATA** to variable/RAM location to use as a normal value



EEPROM Header File

- Can use **EEP.h** for simpler functions calls for EEPROM writing and reading

```
29  * CONSEQUENTIAL DAMAGES, FOR ANY REASON WHATSOEVER.
30  ****
31  #include <pconfig.h>
32
33  /* FUNCTION PROTOTYPES */
34  #if defined (EEP_V1) || defined (EEP_V2) || defined (EEP_V3)
35
36  void Busy_eep ( void );
37  unsigned char Read_b_eep( unsigned int badd );
38  void Write_b_eep( unsigned int badd, unsigned char bdat );
39
40  #endif
41
42
43  #endif
44
```