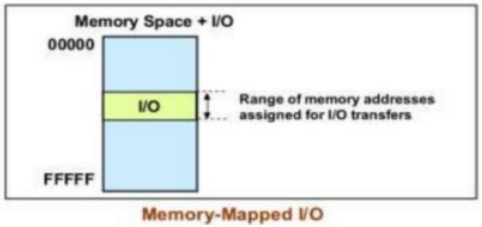
Memory Mapped I/O vs. Isolated I/O

- **Memory Mapped I/O (MOTOROLA):**
  1. **Any instruction that reads or writes memory can read/write I/O Port**
  2. **Address specifies which module (input, output, RAM, ROM), will communicate with the processor**
  3. **Example:**

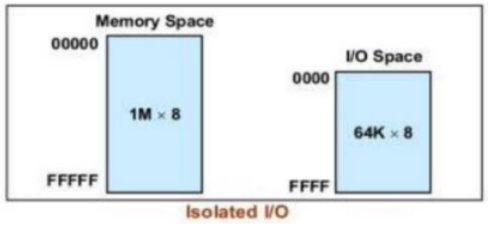
```
LDAA $200A   ;load value from
             ;RAM address $200A
STAA $0024   ;store value to
             ;Port H address $0024
```

- **Isolated I/O (INTEL):**
  1. **The control bus signals that activate the I/O are separate from those that activate the memory device.**
  2. **These systems have a separate address space.**
  3. **Separate instructions are used to access I/O and Memory.**
  4. **Example:**

```
IN AL,$10     ;read from port $10
MOV $200A,AL  ;store value to
              ;RAM address $200A
```



Memory-Mapped I/O



Isolated I/O

RISC vs. CISC

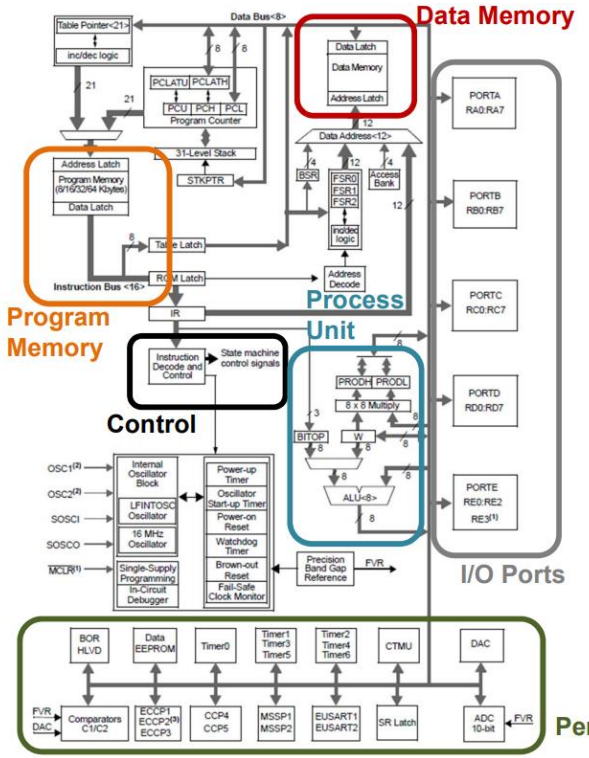
RISC	CISC
Simple instruction set	Complex instruction set
Regular and fixed instruction format	Irregular instruction format
Wired instructions, take only one or a few clock cycles to execute.	Complex microcoded instructions, take many clock cycles to execute.
Simple address modes	Complex address modes
Separated data and program memory	Combined data and program memory
Most operations are register to register	Most operations can be register to memory
Take shorter time to design and debug	Take longer time to design and debug
Provide large number of CPU registers	Provide smaller number of CPU registers
Simple operations, longer programs EFFICIENCY?	Complex operations, shorter programs EFFICIENCY?

Memory mapped I/O

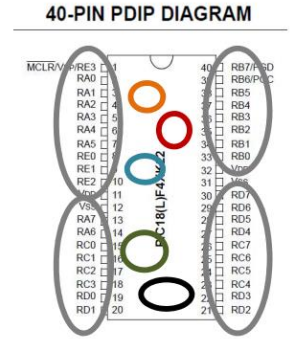
Address	Name	Address	Name	Address	Name	Address	Name
FFFh	TOSU	FDFh	INDF2(1)	FBFh	CCPR1H	F9Fh	IPR1
FFEh	TOSH	FDEh	POSTINC2(1)	FBEh	CCPR1L	F9Eh	PIR1
FFDh	TOSL	FDDh	POSTDEC2(1)	FBDh	CCP1CON	F9Dh	PIE1
FFCh	STKPTR	FDCh	PREINC2(1)	FBCh	CCPR2H	F9Ch	—(2)
FFBh	PCLATU	FDBh	PLUSW2(1)	FBBh	CCPR2L	F9Bh	OSCTUNE
FFAh	PCLATH	FDAh	FSR2H	FBAh	CCP2CON	F9Ah	—(2)
FF9h	PCL	FD9h	FSR2L	FB9h	—(2)	F99h	—(2)
FF8h	TBLPTRU	FD8h	STATUS	FB8h	BAUDCON	F98h	—(2)
FF7h	TBLPTRH	FD7h	TMR0H	FB7h	PWM1CON(3)	F97h	—(2)
FF6h	TBLPTRL	FD6h	TMR0L	FB6h	ECCP1AS(3)	F96h	TRISE(3)
FF5h	TABLAT	FD5h	T0CON	FB5h	CVRCON	F95h	TRISD(3)
FF4h	PRODH	FD4h	—(2)	FB4h	CMCON	F94h	TRISC
FF3h	PRODL	FD3h	OSCCON	FB3h	TMR3H	F93h	TRISB
FF2h	INTCON	FD2h	HLVDCON	FB2h	TMR3L	F92h	TRISA
FF1h	INTCON2	FD1h	WDTCON	FB1h	T3CON	F91h	—(2)
FF0h	INTCON3	FD0h	RCON	FB0h	SPBRGH	F90h	—(2)
FEFh	INDF0(1)	FCFh	TMR1H	FAFh	SPBRG	F8Fh	—(2)
FEEh	POSTINC0(1)	FCEh	TMR1L	FAEh	RCREG	F8Eh	—(2)
FEDh	POSTDEC0(1)	FCDh	T1CON	FADh	TXREG	F8Dh	LATE(3)
FECh	PREINC0(1)	FCCh	TMR2	FACh	TXSTA	F8Ch	LATD(3)
FEBh	PLUSW0(1)	FCBh	PR2	FABh	RCSTA	F8Bh	LATC
FEAh	FSR0H	FCAh	T2CON	FAAh	—(2)	F8Ah	LATB
FE9h	FSR0L	FC9h	SSPBUF	FA9h	EEADR	F89h	LATA
FE8h	WREG	FC8h	SSPADD	FA8h	EEDATA	F88h	—(2)
FE7h	INDF1(1)	FC7h	SSPSTAT	FA7h	EECON2(1)	F87h	—(2)
FE6h	POSTINC1(1)	FC6h	SSPCON1	FA6h	EECON1	F86h	—(2)
FE5h	POSTDEC1(1)	FC5h	SSPCON2	FA5h	—(2)	F85h	—(2)
FE4h	PREINC1(1)	FC4h	ADRESH	FA4h	—(2)	F84h	PORTE(3)
FE3h	PLUSW1(1)	FC3h	ADRESL	FA3h	—(2)	F83h	PORTD(3)
FE2h	FSR1H	FC2h	ADCON0	FA2h	IPR2	F82h	PORTC
FE1h	FSR1L	FC1h	ADCON1	FA1h	PIR2	F81h	PORTB
FE0h	BSR	FC0h	ADCON2	FA0h	PIE2	F80h	PORTA

ports

PICs use a memory mapped solution

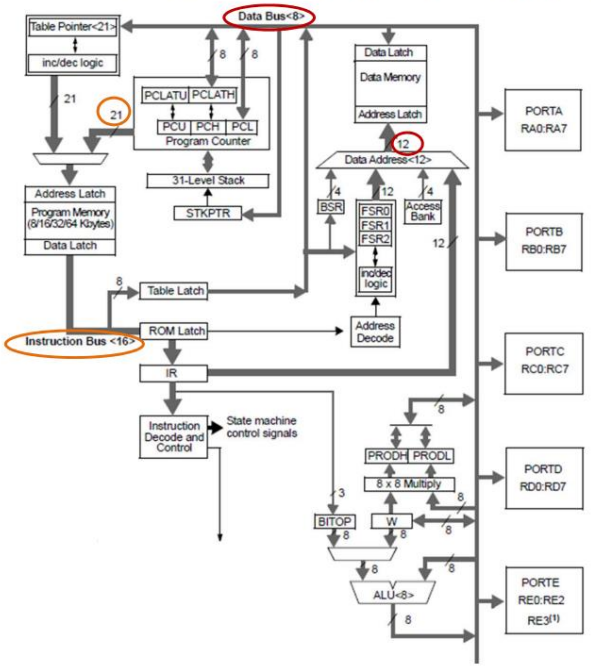


PIC18 Architecture 18F45K22



System on chip

PIC18 Architecture. Data/Program buses



Program Memory Bus:

21 bit address bus
16 bit wide instruction

(32KB out of 2MB max)

Data Memory Bus:

12 bit address bus
8 bit wide data

(1536B out of 4096B max)

Types of semiconductor memory

Main classification

- **Volatile:** loses data when power is turned off
  - Typical volatile: **Random-Access Memory (RAM)**. It's a Read/Write memory.
- **Non-volatile:** keeps stored data when power is turned off
  - Typical non-volatile: **Read-Only Memory (ROM)**. Can only be read but not written by the processor

Types of RAM:

- **Dynamic random-access memory (DRAM):** need periodic refresh
- **Static random-access memory (SRAM):** no periodic refresh is required

Types of ROM:

- **Mask-programmed read-only memory (MROM):** programmed when being manufactured
- **Programmable read-only memory (PROM):** can be programmed by the end user

Separation of Data Memory and Program Memory

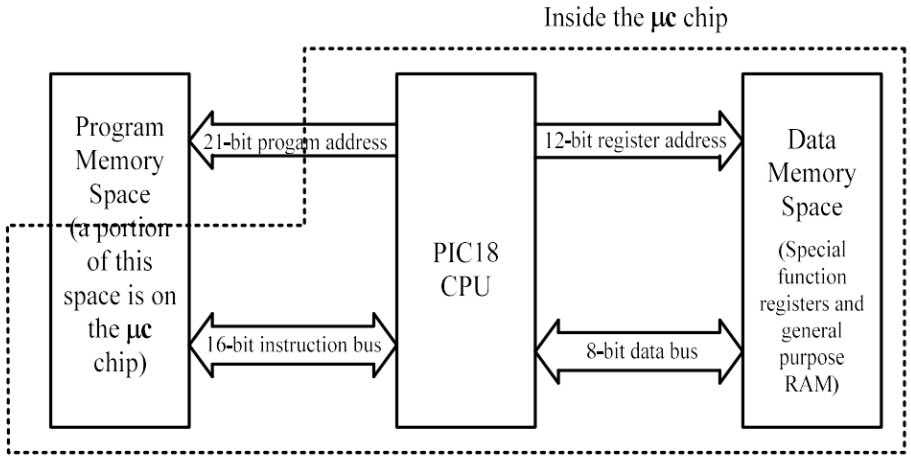
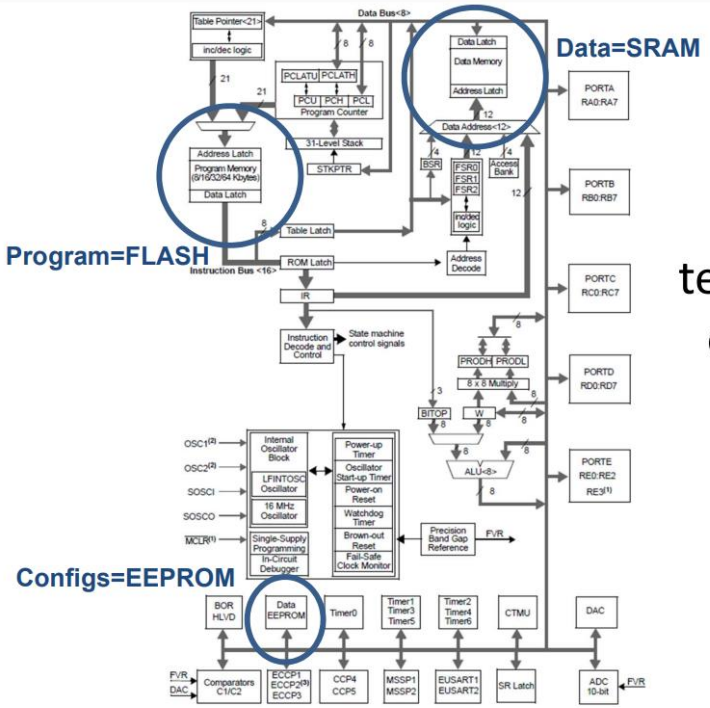


Figure 1.3 The PIC18 memory spaces

Types of semiconductor memory

Types of ROM:

- **Erasable programmable ROM (EPROM)**
  - electrically programmable many times
  - erased by ultraviolet light (through a window)
  - erasable in bulk (whole chip in one erasure operation)
- **Electrically erasable programmable ROM (EEPROM)**
  - electrically programmable many times
  - electrically erasable many times
  - can be erased one location, one row, or whole chip in one operation
- **Flash memory**
  - electrically programmable many times
  - electrically erasable many times
  - can only be erased in bulk (either a block or the whole chip)



Memory technologies on PIC18F

Program Memory Organization

- The program counter (PC) is 21-bit long, which enables the user program to access up to 2 MB of program memory.

- The PIC18 has a 31-entry return address stack to hold the return address for subroutine call.

- After power-on, the PIC18 starts to execute instructions from address 0.

- The location at address 0x08 is reserved for high-priority interrupt service routine.

- The location at address 0x18 is reserved for low-priority interrupt service routine.

- Up to 32KB of program memory is inside the MCU chip. (Accessing a location beyond the upper boundary of the physically implemented memory will return all '0's)

2- Sumar el contingut de @000h amb @001h i guardar a @002h

```
MOVLB  0          //BSR = 0
MOVF   00, w, A   //w <- 000h
ADDWF  01, w, A   //w<- 000h+001h
MOVWF  02,A       //002h <- w
```

3- Sumar 312h amb 107h guardar a 203h

```
MOVLB  3          //BSR=3
MOVF   12h, w, B  //w<- @312
MOVLB  1          //BSR = 1
ADDWF  7h, w, B   //w <- @312 + @107
MOVLB  2          //BSR=2
MOVWF  03, B      //@203 <- w (@312+@107)
```

1- POSAR A @001 L'ABS(@000)

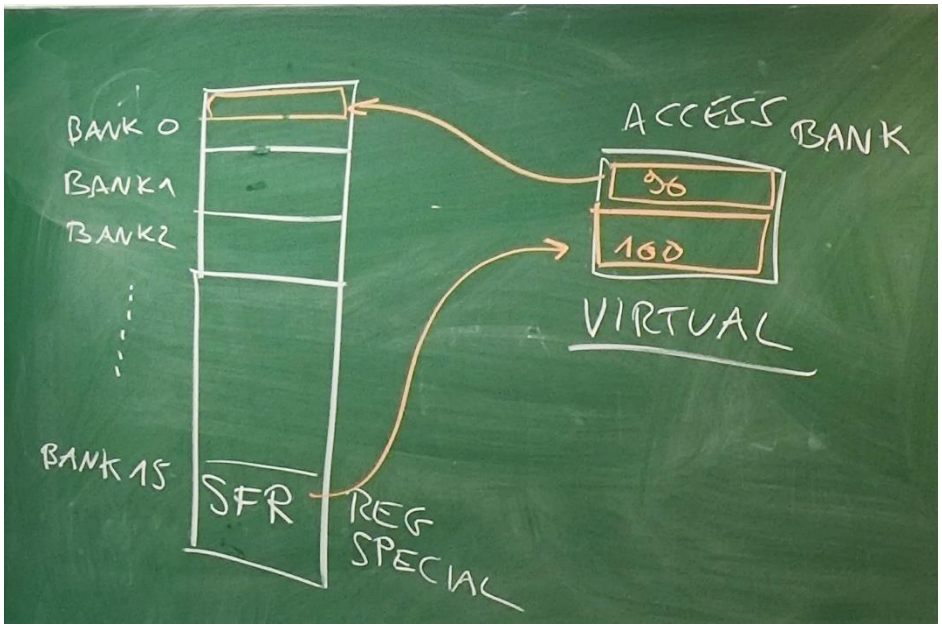
```
MOVF   01,W, A    //W <- @001
BNN    final
COMF   WREG       //W = -W
final:
MOVWF  00, A      //000<- W
```

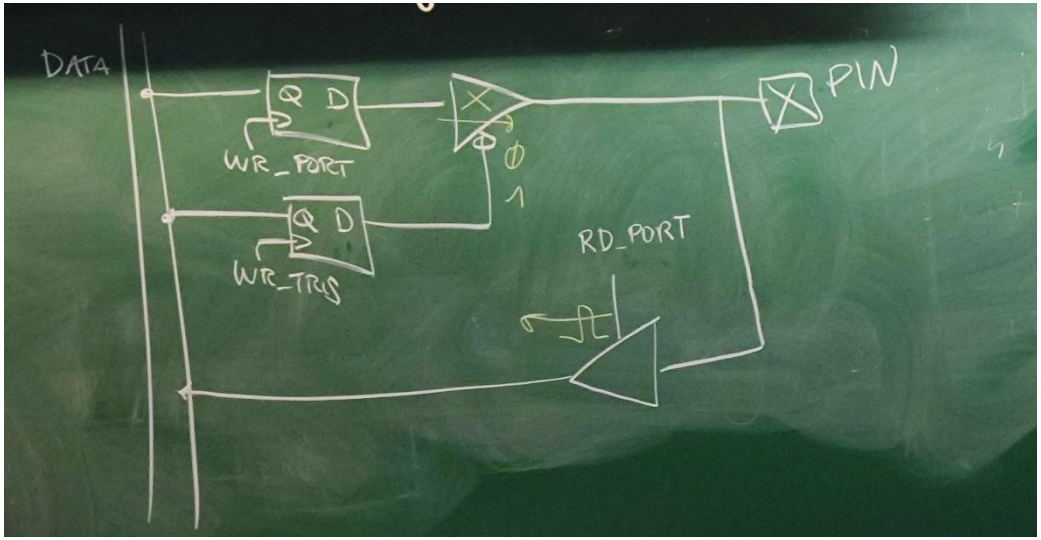
NECF = nega f

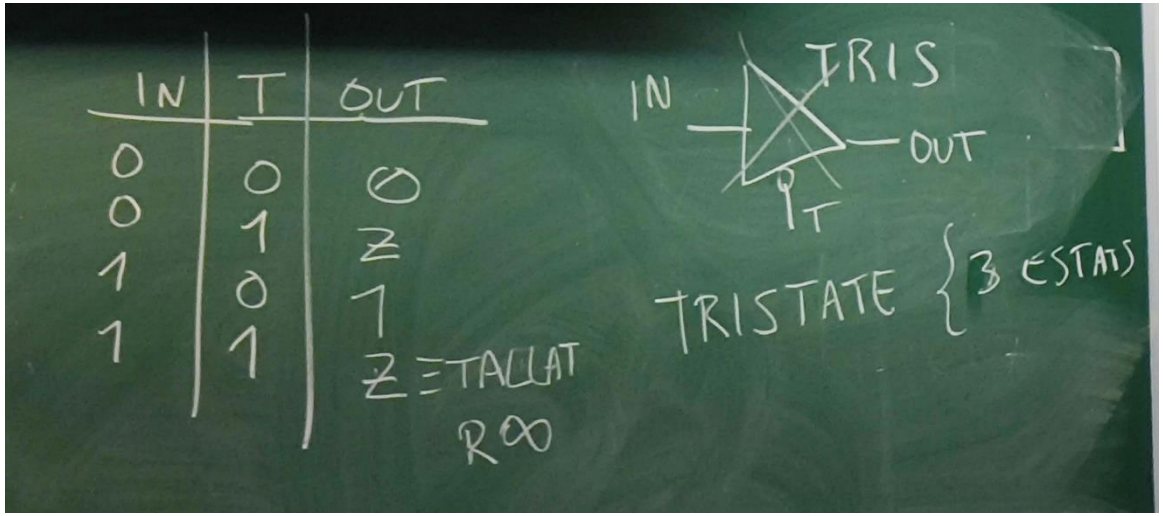
COMF = complementa f

A = es fa servir nomes 96 bytes del bloc 0 (mode Acces Bank)

B = es fan servir tots els banks







PROGRAMA QUE ESPERA UN '1' AL BIT 2 DEL PORT B

//**TRISB**, bit 2 <- 1, ENTRADA

//**ANSELB**, bit 2 <- 0, DIGITAL

//LECTURA, PORT B, bit 2

BSF TRISB,2,A //BIT SET FILE

BCF ANSELB,2,A //BIT CLEAR FILE

Bucle:

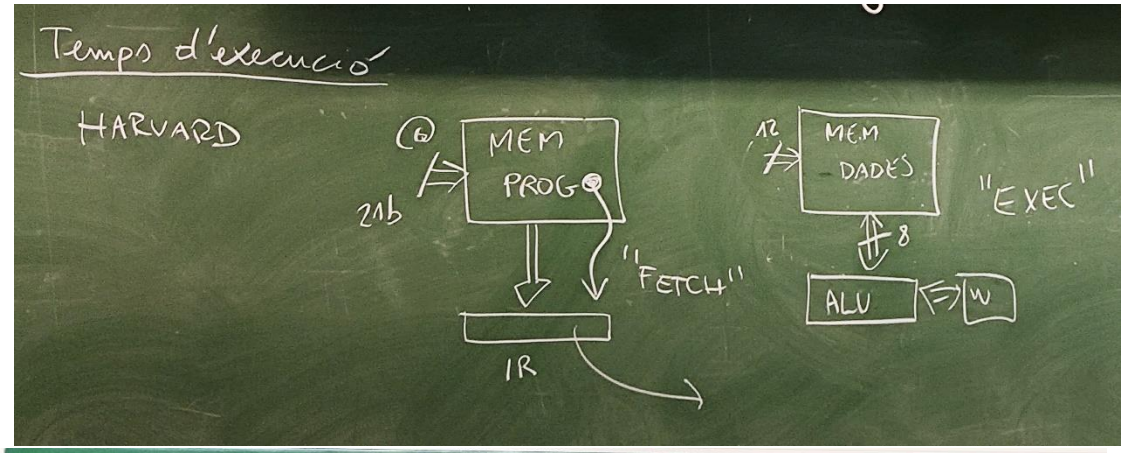
BTFSC PORTB,2,A //BIT TEST FILE SKIP CLEAR

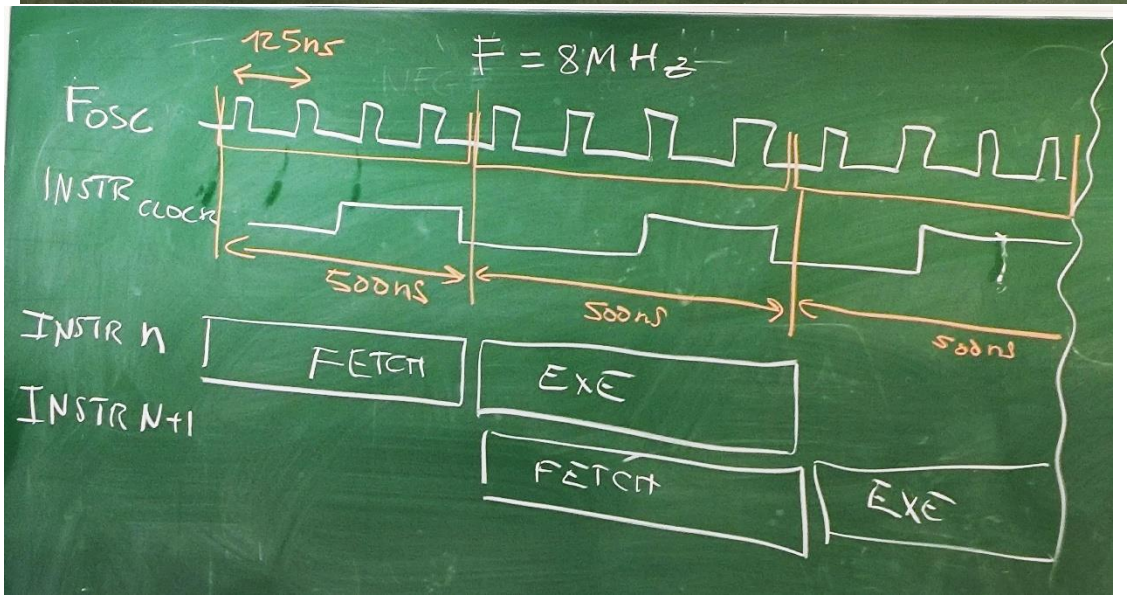
BRA Bucle

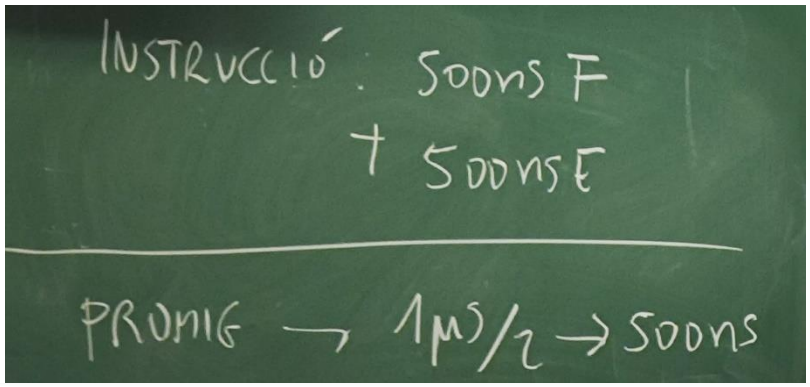
PROGRAMA QUE POSA 33d A LA POSICIO 006h

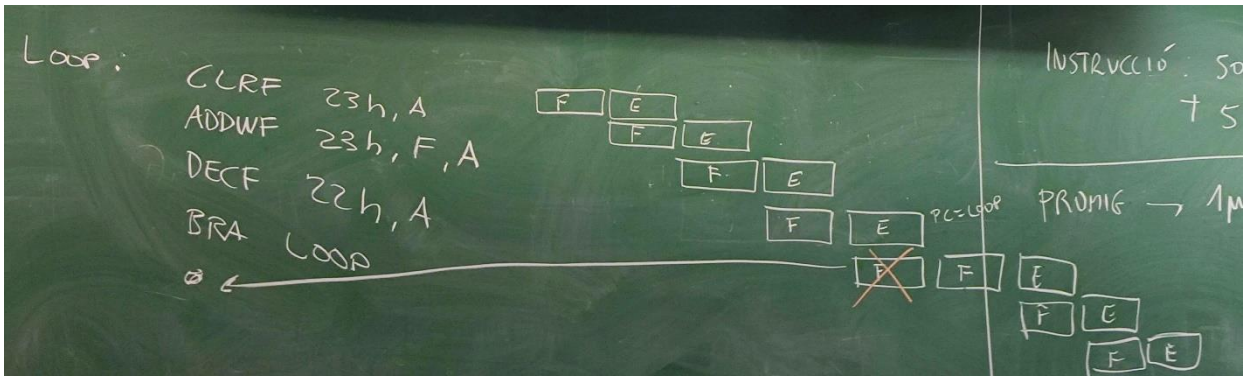
MOVLW 33

MOVWF **06h**, A









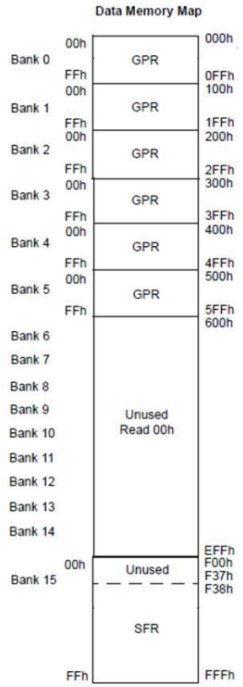
2-Word instructions

- The standard PIC18 instruction set has four two-word instructions:
  CALL, MOVFF, GOTO and LFSR.
- In all cases, the second word of the instructions always has '1111' as its four Most Significant bits
- If the instruction is executed in proper sequence, immediately after the first word, the data in the second word is accessed and used by the instruction sequence. If the first word is skipped for some reason and the second word is executed by itself, a NOP is executed instead

CASE 1:		
Object Code	**Source Code**	
0110 0110 0000 0000	TSTFSZ REG1	; is RAM location 0?
1100 0001 0010 0011	MOVFF REG1, REG2	; Yes, skip this word
1111 0100 0101 0110		; Execute this word as a NOP
0010 0100 0000 0000	ADDWF REG3	; continue code
CASE 2:		
Object Code	**Source Code**	
0110 0110 0000 0000	TSTFSZ REG1	; is RAM location 0?
1100 0001 0010 0011	MOVFF REG1, REG2	; No, execute this word
1111 0100 0101 0110		; 2nd word of instruction
0010 0100 0000 0000	ADDWF REG3	; continue code

Banked RAM

- Most instructions use 8 bits to specify a data register (**f** field).
- Eight bits can specify only 256 registers. This limitation forces the PIC18 to **divide data registers into banks**.
- PIC18F45K22 implements **seven banks:**
  - 6 GPR-based for a total of 6*256B = 1,5KB
  - 1 SFR-based (200B implemented out of 256 possible)



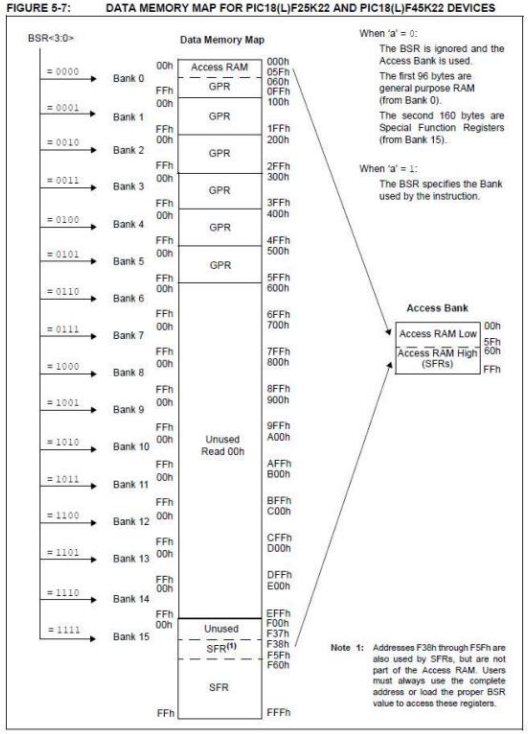
PIC18 Data Memory

- Implemented in SRAM and consists of **general-purpose registers** (GPR) and **special-function registers** (SFR). Both are referred to as data registers.
- A PIC18 MCU may have up to 4096 bytes of data memory.
- General-purpose registers (GPR) are used to hold dynamic data.
- Special-function registers (SFR) are used to control the operation of peripheral functions.

Banked RAM

- Only one bank is active at a time. When operating on a data register in a different bank, bank switching is needed.
  The active bank is specified by the **BSR register** (Bank Select Register)
- Bank switching incurs overhead and may cause program errors.
- **Access bank** is created to minimize the problems of bank switching.
- Access bank consists of the lowest **96 bytes in general-purpose registers** and the highest **160 bytes of special function registers**.
- When operands are in the access bank, no bank switching is needed.

FIGURE 5-7: DATA MEMORY MAP FOR PIC18(L)F25K22 AND PIC18(L)F45K22 DEVICES

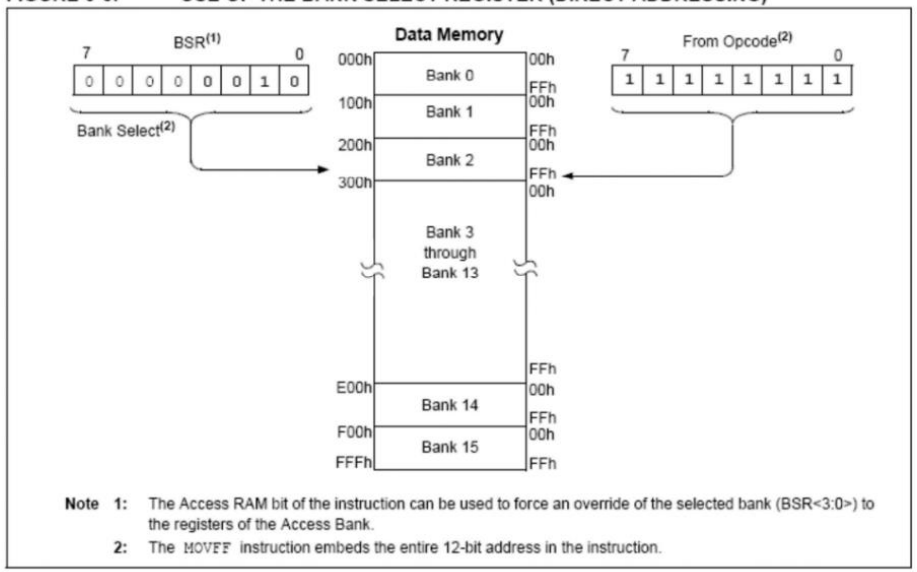


RAM is divided into 256B Banks.

Two addressing modes: Banked (BSR) and Access

Bank Select Register

FIGURE 5-6: USE OF THE BANK SELECT REGISTER (DIRECT ADDRESSING)



(The BSR can be loaded directly by using the MOVLB instruction)

Selecting Access Bank or BSR

- To indicate use of Access Bank or BSR we use the 'a' parameter in many instructions

`movwf f,a`

When 'a' = 0:

The BSR is ignored and the Access Bank is used.

The first 96 bytes are general purpose RAM (from Bank 0). The remaining 160 bytes are Special Function Registers (from Bank 15).

When 'a' = 1:

The BSR specifies the bank used by the instruction

Examples of the Use of Access Bank

- The following examples are instructions with 3 parameters, e.g.: **addwf f,d,a**

- Arguments 'd' and 'a' can be 0 or 1. For the sake of understanding, in the slides we'll use the following labels:
  - In 'd' (destination): **F** (1) indicates file, **W** (0) indicates WREG register
  - In 'a' (bank addressing): **A** (0) indicates Access Bank, **BANKED** (1) indicates BSR usage

```
addwf 0x20,F,A        ;add WREG with the data register at 0x20 in access bank
                      ;and store the sum at 0x20 in access bank.

subwf 0x30,F,BANKED   ;subtract the value of WREG from the data register
                      ;0x30 in the bank specified by the current contents
                      ;of the BSR register. The difference is stored in
                      ;data register 0x30 in the same bank.

addwf 0x40,W,A        ;add the WREG register with data register at 0x40 in
                      ;access bank and leaves the sum in WREG.
```

Status register

REGISTER 5-2: STATUS REGISTER

U-0	U-0	U-0	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x
—	—	—	N	OV	Z	DC(1)	C(2)
bit 7							bit 0

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'	
-n = Value at POR	'1' = Bit is set	'0' = Bit is cleared	x = Bit is unknown

bit 7-5 **Unimplemented:** Read as '0'

bit 4 **N:** Negative bit
This bit is used for signed arithmetic (2's complement). It indicates whether the result was negative (ALU MSB = 1).
1 = Result was negative
0 = Result was positive

bit 3 **OV:** Overflow bit
This bit is used for signed arithmetic (2's complement). It indicates an overflow of the 7-bit magnitude which causes the sign bit (bit 7 of the result) to change state.
1 = Overflow occurred for signed arithmetic (in this arithmetic operation)
0 = No overflow occurred

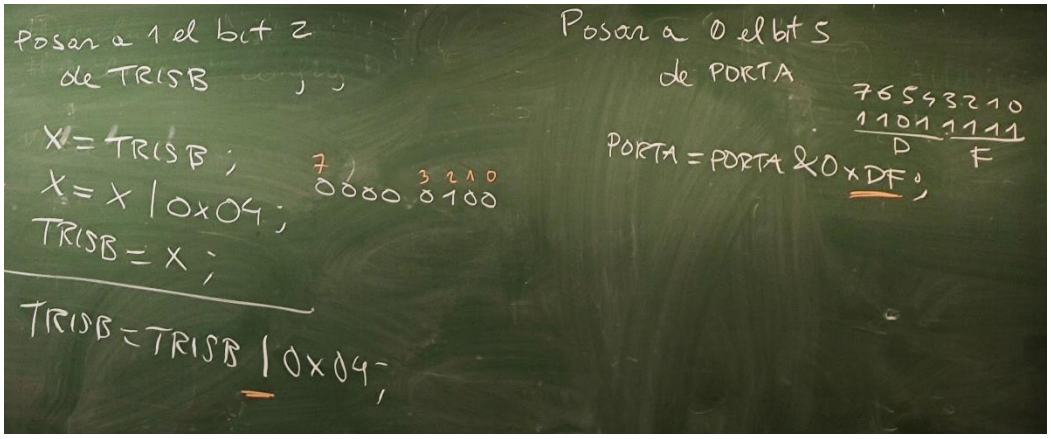
bit 2 **Z:** Zero bit
1 = The result of an arithmetic or logic operation is zero
0 = The result of an arithmetic or logic operation is not zero

bit 1 **DC:** Digit Carry/Borrow bit(1)
For ADDWF, ADDLW, SUBLW and SUBWF instructions:
1 = A carry-out from the 4th low-order bit of the result occurred
0 = No carry-out from the 4th low-order bit of the result

bit 0 **C:** Carry/Borrow bit(2)
For ADDWF, ADDLW, SUBLW and SUBWF instructions:
1 = A carry-out from the Most Significant bit of the result occurred
0 = No carry-out from the Most Significant bit of the result occurred

& -> **AND BIT A BIT**

&& -> **AND BOOLEANA**



PIC18 Addressing Modes

4 addressing modes:

- **Inherent**: don't need any argument

```
sleep, reset
```

- **Literal**: require an explicit argument in the opcode

```
movlw 0x30     ; load 0x30 into WREG
```

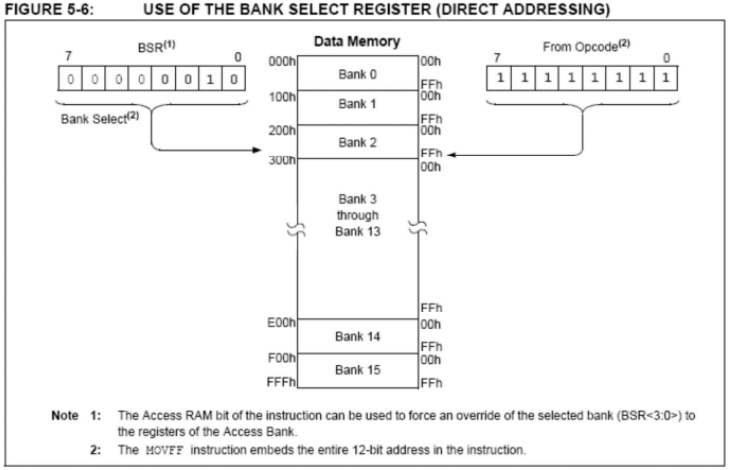
- **Direct**: specifies source and/or destination address

```
movwf 0x20     ; the value 0x20 is
               ; register direct mode
```

- **Indirect**: Access a location without giving a fixed address in the instruction. Use FSR register as pointers.

Direct addressing mode

FIGURE 5-6: USE OF THE BANK SELECT REGISTER (DIRECT ADDRESSING)



Note 1: The Access RAM bit of the instruction can be used to force an override of the selected bank (BSR<3:0>) to the registers of the Access Bank.
2: The MOVFF instruction embeds the entire 12-bit address in the instruction.

ADDWF f, d, a

The destination of the operation's results is determined by the destination bit 'd'.
'd' = 1, the results are stored back in the source register, overwriting its original contents.
'd' = 0, the results are stored in the W register.

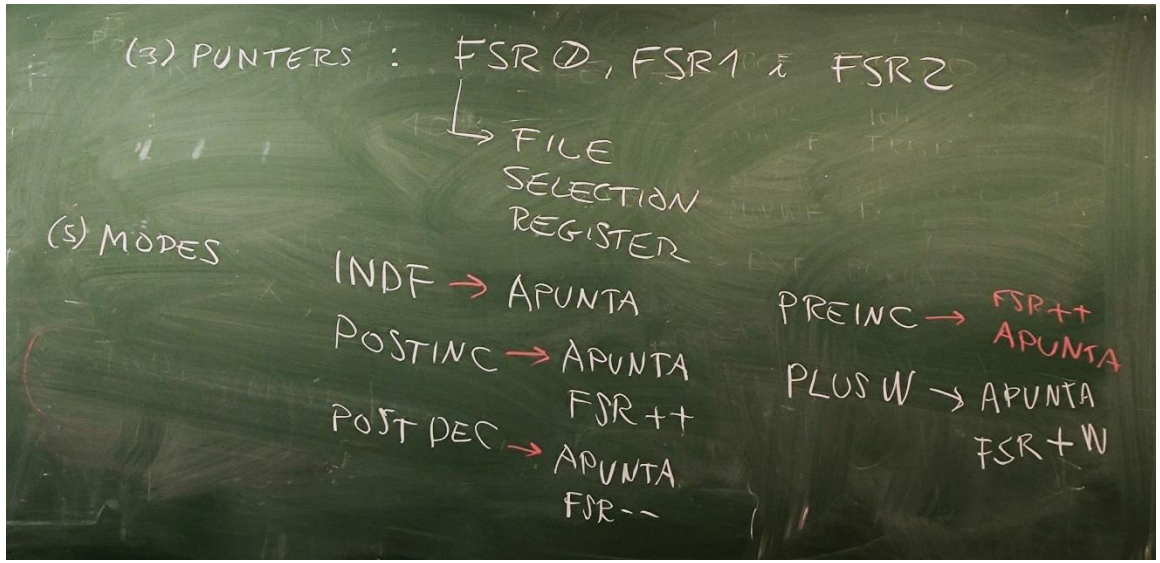
Instructions without the 'd' argument have a destination that is implicit in the instruction.

Indirect addressing mode

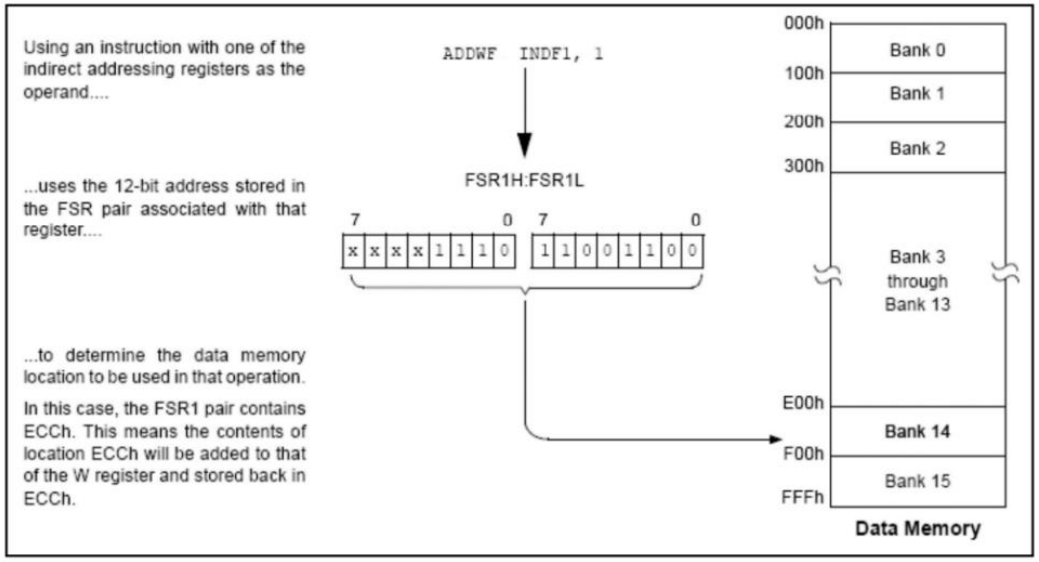
- File Select Registers (FSRx) are used as pointers to the actual data register.
- The registers for Indirect Addressing are also implemented with Indirect File Operands that permit automatic manipulation of the pointer value with auto-incrementing, auto-decrementing or offsetting with another value.
- 5 Modes: INDF POSTDEC POSTINC PREINC PLUSW

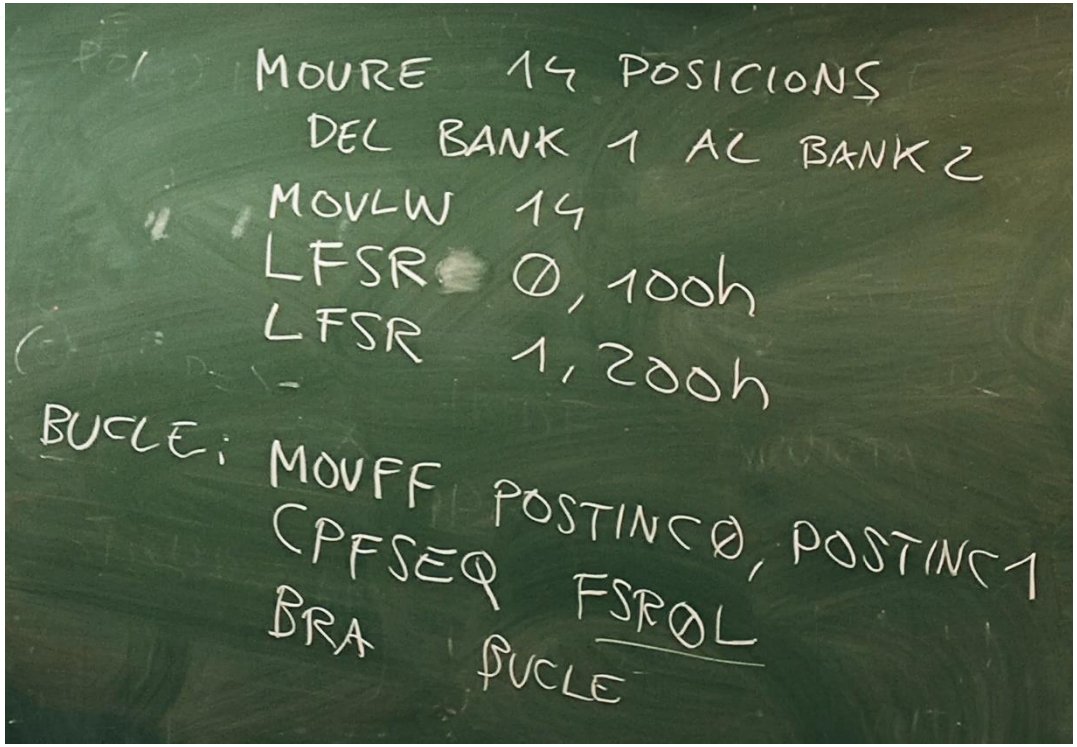
EXAMPLE 5-5: HOW TO CLEAR RAM (BANK 1) USING INDIRECT ADDRESSING

```
        LFSR    FSR0, 100h ;
NEXT    CLRF    POSTINC0    ; Clear indirect
                            ; register then
                            ; inc FSR pointer
        BTFSS   FSR0H, 1    ; All done with
                            ; Bank1?
        BRA     NEXT        ; NO, clear next
CONTINUE                    ; YES, continue
```



Indirect addressing mode





Assembler directives. Examples

#define <name> [<string>] #define PORTA 80
This directive defines a text substitution string. Whenever <name> is encountered in the assembly code, <string> will be substituted.

#include <include_file> #include <p18f2525.inc>
This directive includes additional source file. The specified file is read in as source code. The effect is the same as if the entire text of the included file were inserted into the file at the location of the include statement.

[<label>] org <expr> Reset ORG 0000h
This directive sets the program origin for subsequent code at the address defined in <expr>.

Programs. Begin & end

START

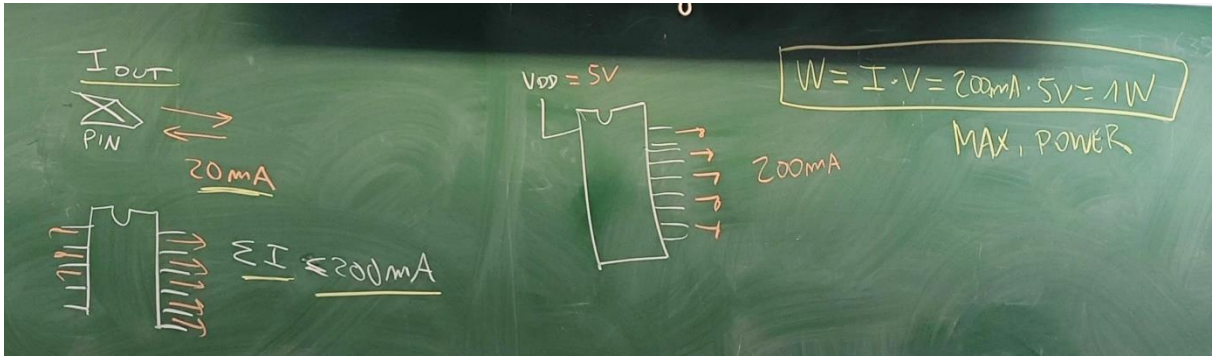
We fix the location of instructions in memory by the directive ORG

```
org 0x0000h
bra main
```

STOP

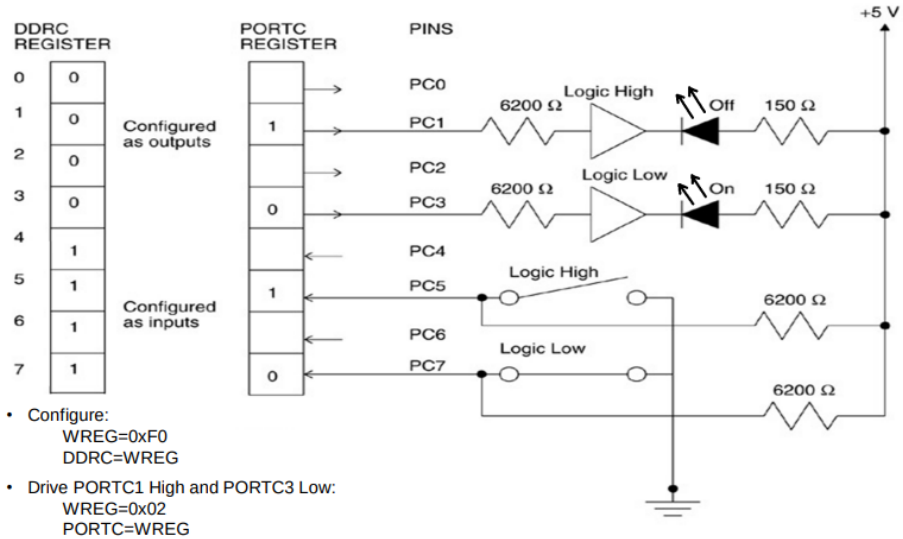
Program can NOT wander around any memory location.
Execution must be limited to program lines written by user.

```
loop
 bra loop
end
```



Need of I/O ports

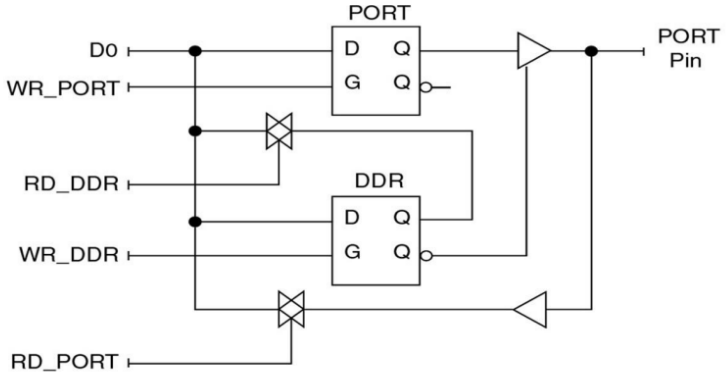
- Several applications require interfacing an MCU with Light Emitting Diodes, Switches, Liquid Crystal Display, Seven Segment Displays.
- I/O ports therefore have to be programmed to handle input/output signals.



- Configure:
  WREG=0xF0
  DDRC=WREG
- Drive PORTC1 High and PORTC3 Low:
  WREG=0x02
  PORTC=WREG
- Read Inputs (use High Nibble only)
  WREG=PORTC

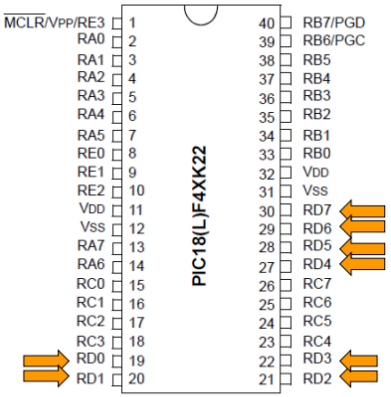
General Purpose I/O: Bidirectional

- Most GPIO pins on the MCU can be programmed for use in either direction.
- Two registers: the data register PORT and data direction register DDR.
  - The DDR determines the direction of the port pin.
    - if the DDR = 0 then the port is an output and
    - if the DDR = 1 the data register output is disabled and the port pin is placed in high impedance state.



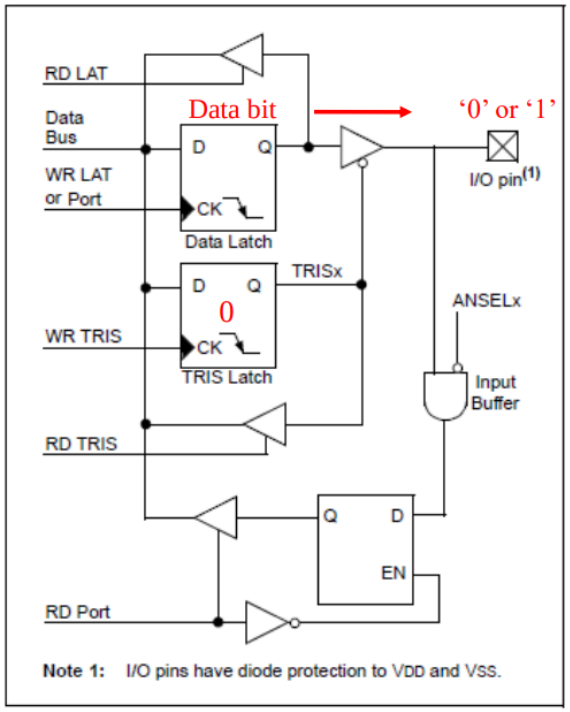
Overview of the PIC18 Parallel I/O Ports

- I/O pins are often grouped into **ports**.
- A port consists of up to 8 pins, a data direction register (**TRISx**), a latch register (**LATx**) , and a data register (**PORTx**); where x=A,B,C,D,…etc.
- Data to be output is written into the latch, which in turn drives the output pins.
- An I/O port is often **multiplexed** with one or more peripheral functions.
- When a peripheral function is enabled, the I/O pins cannot be used for general purpose I/O.
- A PIC18 may have as many as 10 I/O ports.



Example: PORTD pins D{0..7}

PIC18 I/O port operation

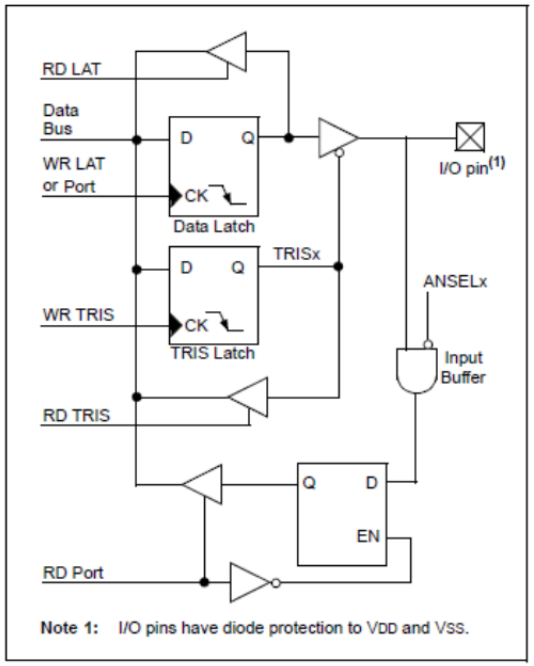


Output

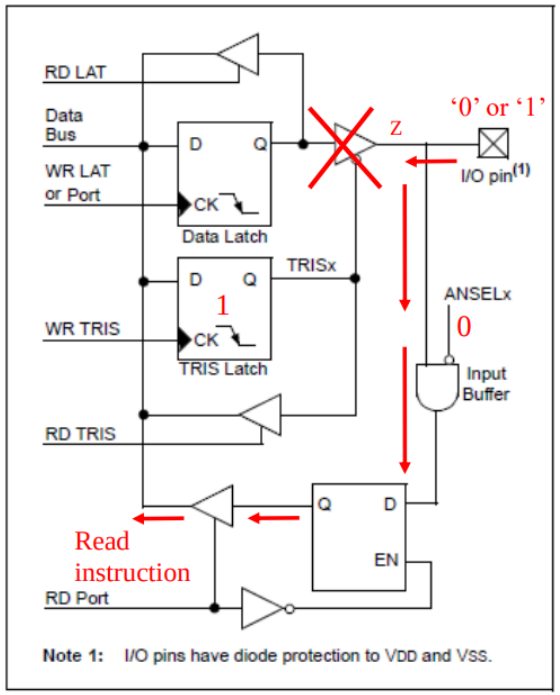
PIC18 I/O port operation

- **TRISx register (data direction register. 1 Input, 0 Output)**
- **PORTx register (reads the levels on the pins of the device)**
- **LATx register (output latch)**
- **ANSELx register (analog input control.** 0 Digital / 1 Analog**)**

x = {A, B, C, D, E}



PIC18 I/O port operation



Input

PORTA

- **PORTA is an 8-bit wide, bidirectional port.**
- **Pins RA6 and RA7 are multiplexed with the main oscillator pins.**
- **The operation of pins RA<3:0> and RA5 as analog is selected by setting the ANSELA<5, 3:0> bits in the ANSELA register which is the default setting after a Power-on Reset.**

Note: **On a Power-on Reset, RA5 and RA<3:0> are configured as analog inputs and read as '0'. RA4 is configured as a digital input.**

PORTB

- **PORTB is an 8-bit wide, bidirectional port.**
- **All pins may be used as Digital or Analog Pins**
- **The pins RB<2:0> may be used for external interrupts (Int0, Int1 and Int2).**
- **Four of the PORTB pins (RB<7:4>) are individually configurable as interrupt-on-change pins**

Note: **On a Power-on Reset, RB<5:0> are configured as analog inputs by default and read as '0'; RB<7:6> are configured as digital inputs.**

When the PBADEN Configuration bit is set to '0', RB<5:0> will alternatively be configured as digital inputs on POR.

PORTC

- **PORTC is an 8-bit wide, bidirectional port.**
- **RC2..RC7 may be used as Analog Pins. On a Power-on Reset, these pins are configured as analog inputs.**

PORTD

- **PORTD is an 8-bit wide, bidirectional port.**
- **All the pins may be used as Analog Pins. On a Power-on Reset, these pins are configured as analog inputs.**

PORTE

- **PORTE is a 4-bit wide, bidirectional port.**
- **RE0..RE2 may be used as Analog Pins. On a Power-on Reset, these pins are configured as analog inputs.**
- **RE3 = MASTER CLEAR. On a Power-on Reset, RE3 is enabled as a digital input only if Master Clear functionality is disabled.**

Example: C programming

```
#include <p18f45k22.h>
#include "config.h"

void main()
{
    ANSELA = 0xC0;    //A5-A0:DIGITAL, Pin A6 and A7? OSC or IO
    ANSELB = 0x00;    // B DIGITAL
    TRISA = 0xFF;     // PORTA INPUT
    TRISB = 0x00;     // PORTB OUTPUT
    PORTB = 0x00;

    while (1) {
        PORTB = PORTA;
    };
}
```

Example: Initializing PORTA

```
MOVLB 0xF          ; Set BSR for banked SFRs
CLRF PORTA,1       ; Initialize PORTA by
                   ; clearing output
                   ; data latches
CLRF LATA,1        ; Alternate method
                   ; to clear output
                   ; data latches
MOVLW C0h          ; Configure I/O
MOVWF ANSELA,1     ; for digital inputs
MOVLW 0CFh         ; Value used to
                   ; initialize data
                   ; direction
MOVWF TRISA,1      ; Set RA<3:0> as inputs
                   ; RA<5:4> as outputs. A6 and A7 are OSC!!
```

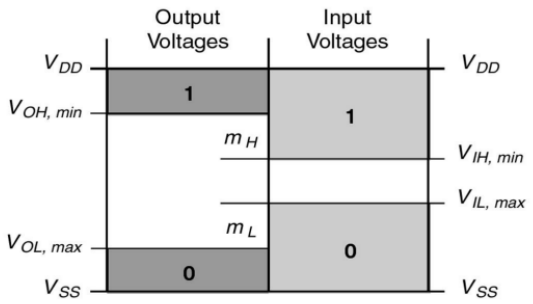
Interfacing: Voltage Parameters

- **Like any digital device, before we can connect something to an input or output, we need to know the specification for the interface parameter.**
- **The first parameter to consider are the input and output voltage levels and corresponding noise margins.**

VDD and VSS are supply voltages. The output voltage parameters are V_{OL} and V_{OH}.

The input voltage parameters are V_{IL} and V_{IH}.

For a digital system to work correctly, the output high voltage always must be between $V_{IH,min}$ and VDD.



Noise Margin?

Interfacing: Current Parameters

- **The interface current parameters are the output currents, I_{OH} and I_{OL}, and the input leakage current I_{IN}.**
  1. **I_{OH} is the current *flowing out* of a *high output* (sourced)**
  2. **I_{OL} is the current *flowing in* to a *low output* (sunk)**
  3. **I_{IN} is the leakage current that flows into or out of an input pin.**

- **These currents are used to determine the *static fanout of a device*, that is, the number of inputs that can be connected to one output while preserving the required voltage margins.**
  1. **Static fanout for a low output is: $n_L = | I_{OL,max} | / | I_{In} |$**
  2. **Static fanout for a high output is: $n_H = | I_{OH,max} | / | I_{In} |$**
  3. **$n = \min [n_H, n_L]$**

Absolute maximum ratings

Ambient temperature under bias	-40°C to +125°C
Storage temperature	-65°C to +150°C
Voltage on any pin with respect to VSS (except VDD, and $\overline{MCLR}$)	-0.3V to (VDD + 0.3V)
Total power dissipation **(Note 1)**	1.0W
Maximum current out of VSS pin (-40°C to +85°C)	300 mA
Maximum current out of VSS pin (+85°C to +125°C)	125 mA
Maximum current into VDD pin (-40°C to +85°C)	200 mA
Maximum current into VDD pin (+85°C to +125°C)	85 mA
Input clamp current, IIK (VI < 0 or VI > VDD)	±20 mA
Output clamp current, IOK (VO < 0 or VO > VDD)	±20 mA
Maximum output current sunk by any I/O pin	25 mA
Maximum output current sourced by any I/O pin	25 mA
Maximum current sunk by all ports (-40°C to +85°C)	200 mA
Maximum current sunk by all ports (+85°C to +125°C)	110 mA
Maximum current sourced by all ports (-40°C to +85°C)	185 mA
Maximum current sourced by all ports (+85°C to +125°C)	70 mA

Maximum I_{OH} or **I_{OL}** for the PIC18F45K22 is +/- 25mA.

This Maximum ratings give the value that if exceeded may destroy the part.

DC Characteristics. Supply Voltage

27.1 DC Characteristics: Supply Voltage, PIC18(L)F2X/4XK22

PIC18(L)F2X/4XK22				Standard Operating Conditions (unless otherwise stated) Operating temperature -40°C ≤ TA ≤ +125°C				
Param No.	Symbol	Characteristic		Min	Typ	Max	Units	Conditions
D001	VDD	**Supply Voltage**	PIC18LF2X/4XK22	1.8	—	3.6	V	
			PIC18F2X/4XK22	2.3	—	5.5	V	
D002	VDR	**RAM Data Retention Voltage**(1)		1.5	—	—	V	
D003	VPOR	VDD **Start Voltage** to ensure internal Power-on Reset signal		—	—	0.7	V	See section on Power-on Reset for details
D004	SVDD	VDD **Rise Rate** to ensure internal Power-on Reset signal		0.05	—	—	V/ms	See section on Power-on Reset for details
D005	VBOR	**Brown-out Reset Voltage**						
		BORV<1:0> = 11(2)		1.75	1.9	2.05	V	
		BORV<1:0> = 10		2.05	2.2	2.35	V	
		BORV<1:0> = 01		2.35	2.5	2.65	V	
		BORV<1:0> = 00(3)		2.65	2.85	3.05	V	

Note 1: This is the limit to which VDD can be lowered in Sleep mode, or during a device Reset, without losing RAM data.

2: On PIC18(L)F2X/4XK22 devices with BOR enabled, operation is supported until a BOR occurs. This is valid although VDD may be below the minimum rated supply voltage.

3: With BOR enabled, full-speed operation (FOSC = 64 MHz or 48 MHz) is supported until a BOR occurs. This is valid although VDD may be below the minimum voltage for this frequency.

Input characteristics

27.8 DC Characteristics: Input/Output Characteristics, PIC18(L)F2X/4XK22

DC CHARACTERISTICS			Standard Operating Conditions (unless otherwise stated) Operating temperature -40°C ≤ TA ≤ +125°C				
Param No.	Symbol	Characteristic	Min	Typ†	Max	Units	Conditions
	VIL	**Input Low Voltage**					
		I/O PORT:					
D140		with TTL buffer	—	—	0.8	V	4.5V ≤ VDD ≤ 5.5V
D140A			—	—	0.15 VDD	V	1.8V ≤ VDD ≤ 4.5V
D141		with Schmitt Trigger buffer	—	—	0.2 VDD	V	2.0V ≤ VDD ≤ 5.5V
		with I²C levels	—	—	0.3 VDD	V	
		with SMBus levels	—	—	0.8	V	2.7V ≤ VDD ≤ 5.5V
D142		MCLR, OSC1 (RC mode)(1)	—	—	0.2 VDD	V	
D142A		OSC1 (HS mode)	—	—	0.3 VDD	V	
	VIH	**Input High Voltage**					
		I/O ports:		—	—		
D147		with TTL buffer	2.0	—	—	V	4.5V ≤ VDD ≤ 5.5V
D147A			0.25 VDD + 0.8	—	—	V	1.8V ≤ VDD ≤ 4.5V
D148		with Schmitt Trigger buffer	0.8 VDD	—	—	V	2.0V ≤ VDD ≤ 5.5V
		with I²C levels	0.7 VDD	—	—	V	
		with SMBus levels	2.1	—	—	V	2.7V ≤ VDD ≤ 5.5V
D149		MCLR	0.8 VDD	—	—	V	
D150A		OSC1 (HS mode)	0.7 VDD	—	—	V	
D150B		OSC1 (RC mode)(1)	0.9 VDD	—	—	V	
	IIL	Input Leakage I/O and MCLR(2),(3)					VSS ≤ VPIN ≤ VDD, Pin at high-impedance
D155		I/O ports and MCLR	— — — —	0.1 0.7 4 35	50 100 200 1000	nA nA nA nA	≤ +25°C(4) +60°C +85°C +125°C

Interpretation: if the power supply (VDD) is between 4,5V and 5,5V then $V_{IL\,MAX}$ = 0,8V

$V_{IL\,MAX}$ = 0,8V (voltage input low maximum) means that any voltage under 0,8V put on an input pin will always be read as a logical '0'

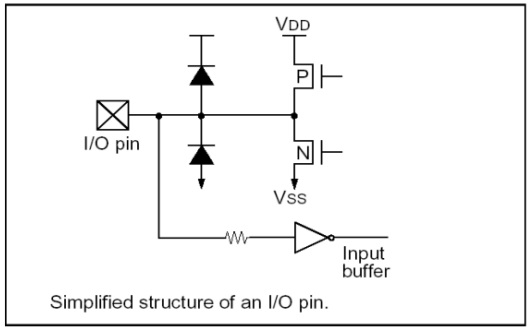
Output characteristics

27.8 DC Characteristics: Input/Output Characteristics, PIC18(L)F2X/4XK22 (Continued)

DC CHARACTERISTICS			Standard Operating Conditions (unless otherwise stated) Operating temperature -40°C ≤ TA ≤ +125°C				
Param No.	Symbol	Characteristic	Min	Typ†	Max	Units	Conditions
D159	VOL	**Output Low Voltage** I/O ports	—	—	0.6	V	IOL = 8 mA, VDD = 5V IOL = 6 mA, VDD = 3.3V IOL = 1.8 mA, VDD = 1.8V
D161	VOH	**Output High Voltage(3)** I/O ports	VDD - 0.7	—	—	V	IOH = 3.5 mA, VDD = 5V IOH = 3 mA, VDD = 3.3V IOH = 1 mA, VDD = 1.8V

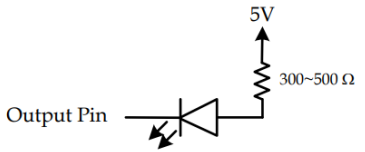
$V_{OL\,MAX}$ = 0,6V (voltage output low maximum) means that even in the worst conditions, a logical '0' will always be output as a voltage less than 0,6V

Protection diodes

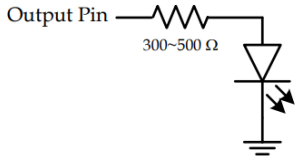


Simplified structure of an I/O pin.

Interfacing with LEDs



(a) low voltage on output pin lights LED



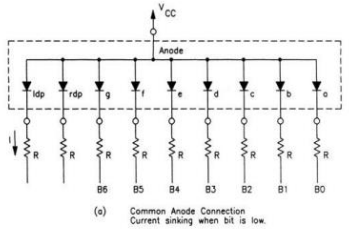
(b) high voltage on output pin lights LED

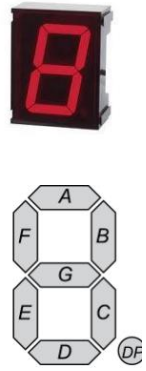
A LED emits light when current flows through it in the positive direction i.e. when the voltage on the anode side is made higher than the voltage on the cathode side.

The forward voltage across the LED is typically about 1.5 to 2 Volts.
We need to add a resistor to limit the current.

Seven Segment Displays

- Common Anode:
  1. All anodes are tied in common.
  2. Segment will be lit whenever a low voltage is applied.
- Common Cathode:
  1. all cathodes are tied in common.
  2. Segment will be lit whenever a high voltage is applied.
- Current limiting resistors must be included or else you might damage display.





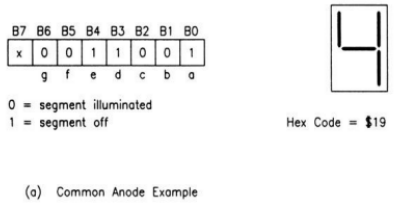
Connecting a LED to an Output Port

- Using a 5-volt supply and assuming that the LED has a 2.0 V drop across it, what resistor value will limit the current to 10mA?
- Answer:
  - 5V = 2.0V + I_{Rx} x Rx
  - Setting I_{Rx} to 10mA the resistor Rx is solved to be 300 Ohm.

Seven Segment Displays: Examples

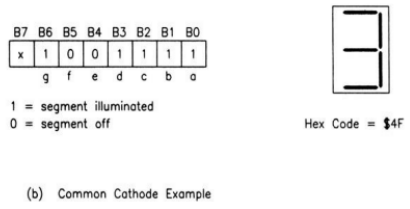
- **Depending on the type of display used a different hex code is generated by the MCU.**

1.Common Anode: sending a "0" will illuminate the segment.



(a) Common Anode Example

2.Common Cathode: sending a "1" will illuminate the segment.



(b) Common Cathode Example

Interfacing with DIP Switches

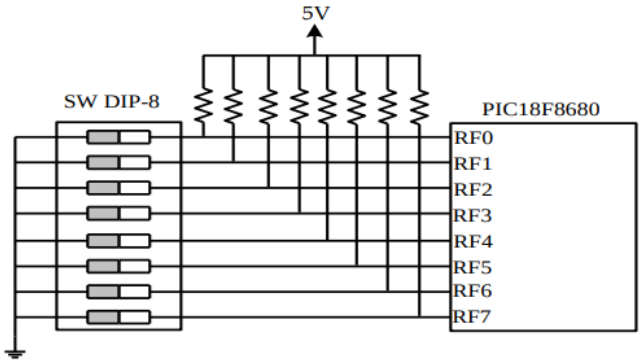


Figure 7.24b Connecting a set of eight DIP switches to port F of the PIC18F8680

Reading a byte from the DIP switches to WREG

```
movlw   0xFF    ; configure port F for input
movwf   TRISF   ;       "
movf    PORTF,W         ; read portF
```

Interfacing with Keypad

Types of Key Switches

1. Membrane: A plastic or rubber membrane presses one conductor onto another.
2. Capacitive: Two parallel plates. Pressing the plates changes the distance between the plates and changes the capacitance.
3. Hall effect: The motion of the magnetic flux lines of a permanent magnet perpendicular to a crystal is detected as voltage appearing between the two faces of the crystal.
4. Mechanical: Two metal contacts are brought together to complete an electrical circuit.

Mechanical Keypads and Keyboard

- **Low cost and strength of construction**
- **Most popular**
- **Pressing the key switch generates a series of pulses instead of a single clean output**
- **Human being cannot press and release the key switch 20 ms**
- **A debouncing process is required for correct operation**

Keypad Input Program

Keypad Input Program **Consists of Three Parts**

1. **Keypad scanning**
2. **Key switch debouncing**
3. **Table lookup**

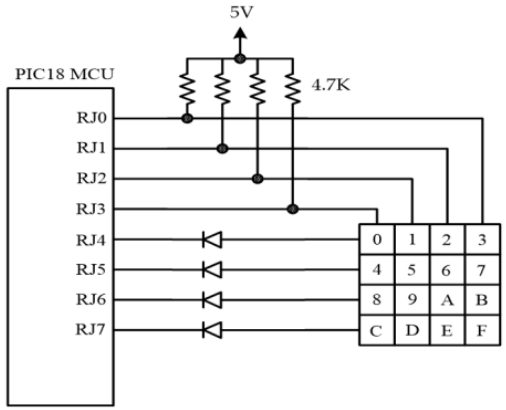


Figure 7.26b Sixteen-key keypad connected to PIC18 (used in all SSE demo boards)

Keypad Scanning

- **Performed to detect which key is being pressed**
- **Performed row by row or column by column**
- **The rows and columns of a keypad are simply conductors**

Example of "row-by-row" scan:

- **The row to be scanned is pulled to low. Other rows are pulled high.**
- **When a key is pressed, the corresponding row and column are shorted together and is detected low.**
- **The diodes in Figure 7.26b provide protection of accidental simultaneous press of multiple keys.**

Keypad Debouncing

- **When a key is pressed, the voltage of the key switch falls and rises a few times within a period of about 5 ms as a contact bounces.**
- **A debouncer will recognize that the switch is closed after the voltage is low for about 10 ms and will recognize that the switch is open after the voltage is high for about 10 ms.**
- **Both hardware and software debouncing solutions are available.**
- **The simplest and most popular software solution is** wait-and-see. **The program simply waits for 10 ms after detecting a key switch is closed and re-examines the same key.**
- **Three hardware debouncing techniques are shown in Figure 7.27.**

ASCII Code Lookup

- **The keypad input subroutine returns the ASCII code to the caller.**
- **This step can be embedded in the debouncing procedure.**

Keypad Debouncing

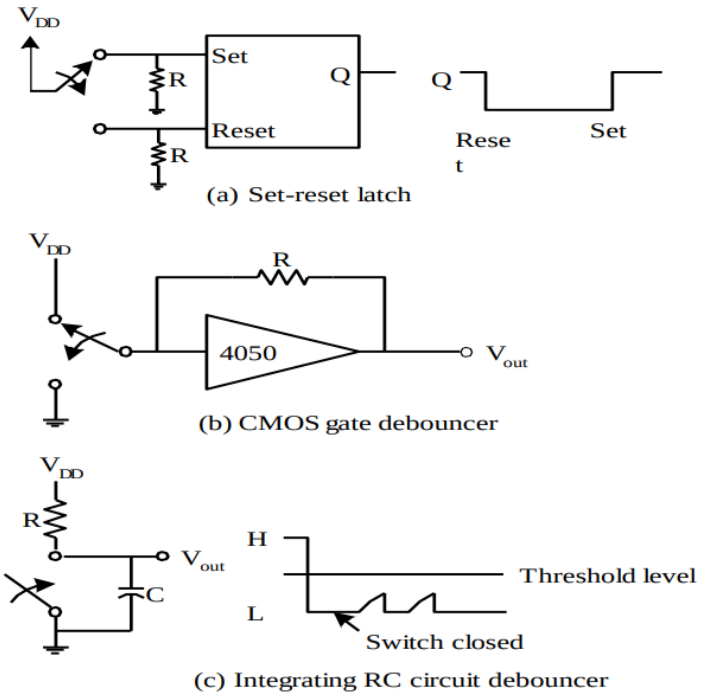


Figure 7.27 Hardware debouncing techniques

```
#include      <p18F8680.h>
#define  keypad_dir  TRISJ
#define  keypad      PORTJ
void wait_10ms(void);
…
unsigned char get_key (void)
{
 keypad_dir = 0x0F;        /* configure RJ7..RJ4 for output, RJ3..RJ0 for input */
 keypad |= 0xF0;           /* start with no row being scanned (row pins to 1) */
 while (1) {
    keypad &= 0xEF;              /* set RJ4 to low to scan the first row */
    if (!(keypadbits.RJ3)) {     /* read the first column */
        wait_10ms( );
        if (!(keypadbits.RJ3))
            return 0x30;  /* return the ASCII code of 0 */
    }
    if (!(keypadbits.RJ2)) {     /* read the second column */
        wait_10ms( );
        if (!(keypadbits.RJ2))
            return 0x31;         /* return the ASCII code of 1 */
…
}
```

Keypad debouncing

"row-by-row" scanning

Processor - I/O speed mismatch

- 1GHz microprocessor can execute 1 billion load or store instructions per second, or 4,000,000 KB/s data rate
  - I/O devices data rates range from 0.01 KB/s to 30,000 KB/s
- Input: device may not be ready to send data as fast as the processor loads it
  - Also, might be waiting for human to act
- Output: device may not be ready to accept data as fast as processor stores it

Synchronization: Mechanisms

Blind Cycle: Software simply waits for a fixed amount of time and assumes the I/O will complete after that fixed delay.

Usage?

Where I/O speed is short and predictable

Gadfly (busy waiting, polling): is a software loop that checks the I/O status waiting for done status.

Usage?

When real time response is not important (CPU can wait)

Interrupts: uses hardware to cause special software execution i.e. input device will cause interrupt when it has new data!

Usage?

When real time response is crucial

Periodic Polling: Uses a clock interrupt to periodically check the I/O Status (i.e. The MCU or CPU will check the status)

Usage?

In situations that require interrupts but the I/O device does not support requests

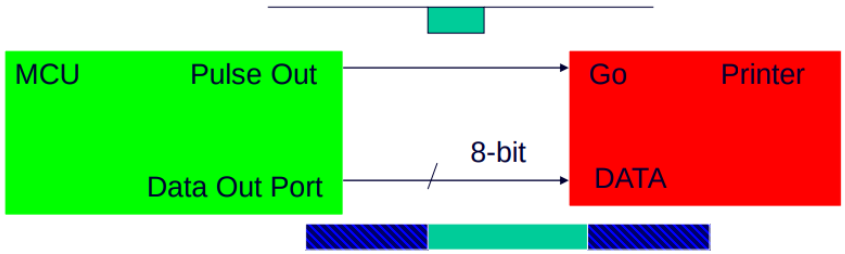
DMA: Transfer data directly to/from memory or I/O without CPU intervention.

Usage?

In situations where Bandwidth and latency are important.

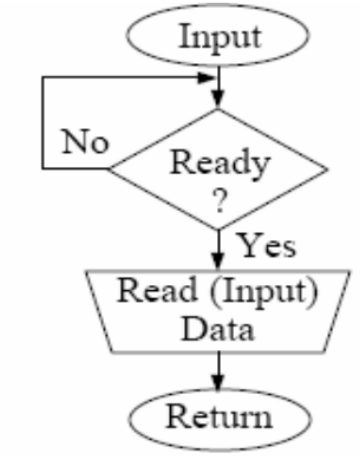
Blind Cycle Synchronization: Example

Printer can put 10 characters per second
With Blind Cycle there is no printer status signal from printer!
A simple software interface would be to output a character then wait 100 ms for it to finish.
Advantage?
Simple
Disadvantage?
If output rate is variable, then time delay is wasted.

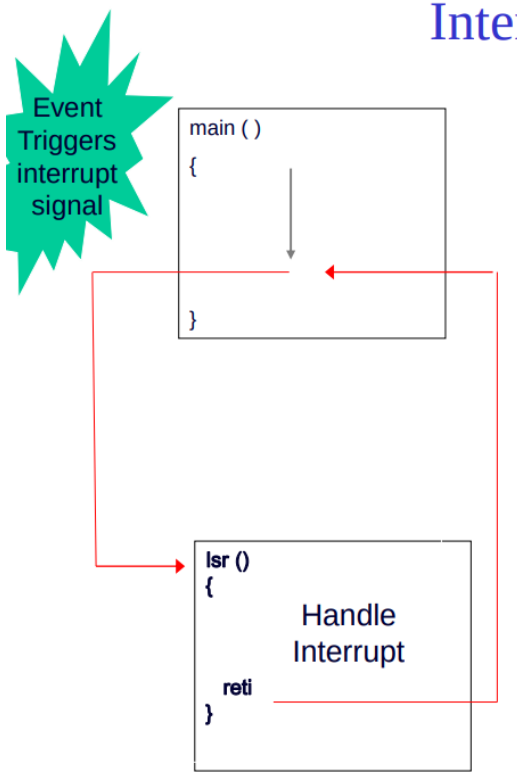


Gadfly (Busy Waiting)

- Software checks a status bit in the I/O device and loops back until device is ready.
- The **Gadfly loop must precede the data transfer for an input** device.



Interrupts



- Several steps have to be completed by the processor to return to the instruction following the instruction that was interrupted.

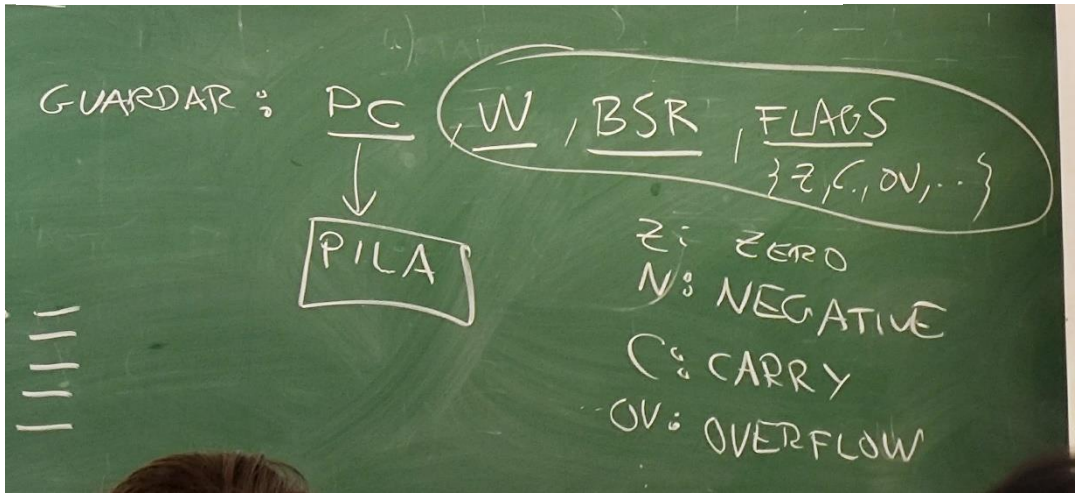
Support for interrupts

- Save the PC for return
  - But where?
- Save other special registers (Status…)
  - But where?
- Where to go when interrupt occurs?
  - MCU defines location. Ex: 0x08

- Determine cause of interrupt?
  - MCU has Cause Register, some bit field gives cause of exception

Some parameters

- *Priority*: Which interrupt will be attended first ?
- *Masking*: Can an interrupt be ignored ?
- *Latency*: How long does it take to attend the interrupt ?
- *Service time*: How long takes the interrupt to be served ?



Interrupts

- An I/O interrupt is like overflow exceptions except:
  - An I/O interrupt is "asynchronous"
  - More information needs to be conveyed

- An I/O interrupt is asynchronous with respect to instruction execution:
  - I/O interrupt is not associated with any instruction, but it can happen in the middle of any given instruction
  - I/O interrupt does not prevent any instruction from completion

Interrupts, traps & exceptions

- Exception: signal marking that something "out of the ordinary" has happened and needs to be handled

- Interrupt: asynchronous exception

- Trap: synchronous exception

Interrupts: sources

Interrupts from on-chip resources.
examples: Serial Interface Module, timer overflow, ADC, …

External Interrupts.
Examples: Input ports, IRQ line…

Software Interrupts.

Exceptions.
Examples: Opcode Trap, stack overflow, divide by zero…

PIC18 registers related to interrupts

These registers enable/disable the interrupts, set the priority of the interrupts, and record the status of each interrupt source.

- RCON → Reset Control Register
- INTCON → Interruption Control Registers
- INTCON2
- INTCON3

- PIR1, PIR2, PIR3, PIR4 and PIR5 → Peripheral IF's
- PIE1, PIE2, PIE3, PIE4 and PIE5 → Peripheral IE's
- IPR1, IPR2, IPR3, IPR4 and IPR5 → Peripheral IP's

Each interrupt source has three bits to control its operation:
- An **Interruption flag** (IF) bit
- An **Interruption enable** (IE) bit
- An **Interruption priority** (IP) bit

PIC18 Interrupt Operation

- All interrupts are divided into **core group** and **peripheral group**.
  The following interrupts are in the **core group**:
  1. INT0…INT2 pin interrupts
  2. TMR0 overflow interrupt
  3. PORTB input pin (RB7…RB4) change interrupts
- The interrupt sources in the **core group** can be enabled by setting the GIE bit and the corresponding enable bit of the interrupt source.
  Ex: To enable TMR0 interrupt, one must set both the GIE and the TMR0IE bits to 1.
- The interrupts in the peripheral group can be enabled by setting the GIE, PEIE, and the associated interrupt enable bits.
  Ex: To enable A/D interrupt, one needs to set the GIE, PEIE, and the ADIE bits
- In order to identify the cause of interrupt, one need to check each individual interrupt flag bit.
- When an interrupt is responded to, the GIE bit is cleared to disable further interrupt, the return address is pushed onto return address stack and the PC is loaded with the interrupt vector.
- Interrupt flags must be cleared in the interrupt service routine to avoid reiterative interrupts.

Resum

TABLE 9-1: REGISTERS ASSOCIATED WITH INTERRUPTS

Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Register on Page
ANSELB	—	—	ANSB5	ANSB4	ANSB3	ANSB2	ANSB1	ANSB0	150
INTCON	GIE/GIEH	PEIE/GIEL	TMR0IE	INT0IE	RBIE	TMR0IF	INT0IF	RBIF	109
INTCON2	$\overline{RBPU}$	INTEDG0	INTEDG1	INTEDG2	—	TMR0IP	—	RBIP	110
INTCON3	INT2IP	INT1IP	—	INT2IE	INT1IE	—	INT2IF	INT1IF	111
IOCB	IOCB7	IOCB6	IOCB5	IOCB4	—	—	—	—	153
IPR1	—	ADIP	RC1IP	TX1IP	SSP1IP	CCP1IP	TMR2IP	TMR1IP	121
IPR2	OSCFIP	C1IP	C2IP	EEIP	BCL1IP	HLVDIP	TMR3IP	CCP2IP	122
IPR3	SSP2IP	BCL2IP	RC2IP	TX2IP	CTMUIP	TMR5GIP	TMR3GIP	TMR1GIP	123
IPR4	—	—	—	—	—	CCP5IP	CCP4IP	CCP3IP	124
IPR5	—	—	—	—	—	TMR6IP	TMR5IP	TMR4IP	124
PIE1	—	ADIE	RC1IE	TX1IE	SSP1IE	CCP1IE	TMR2IE	TMR1IE	117
PIE2	OSCFIE	C1IE	C2IE	EEIE	BCL1IE	HLVDIE	TMR3IE	CCP2IE	118
PIE3	SSP2IE	BCL2IE	RC2IE	TX2IE	CTMUIE	TMR5GIE	TMR3GIE	TMR1GIE	119
PIE4	—	—	—	—	—	CCP5IE	CCP4IE	CCP3IE	120
PIE5	—	—	—	—	—	TMR6IE	TMR5IE	TMR4IE	120
PIR1	—	ADIF	RC1IF	TX1IF	SSP1IF	CCP1IF	TMR2IF	TMR1IF	112
PIR2	OSCFIF	C1IF	C2IF	EEIF	BCL1IF	HLVDIF	TMR3IF	CCP2IF	113
PIR3	SSP2IF	BCL2IF	RC2IF	TX2IF	CTMUIF	TMR5GIF	TMR3GIF	TMR1GIF	114
PIR4	—	—	—	—	—	CCP5IF	CCP4IF	CCP3IF	115
PIR5	—	—	—	—	—	TMR6IF	TMR5IF	TMR4IF	116
PORTB	RB7	RB6	RB5	RB4	RB3	RB2	RB1	RB0	148
RCON	IPEN	SBOREN	—	$\overline{RI}$	$\overline{TO}$	$\overline{PD}$	$\overline{POR}$	$\overline{BOR}$	56

Legend: — = unimplemented locations, read as '0'. Shaded bits are not used for Interrupts.

Interrupt sources. Examples

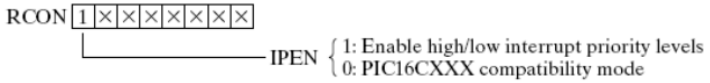
INT0…INT2 Pin Interrupts
- All INT pins interrupt are edge-triggered.
- The edge-select bits are contained in the INTCON2 register.
- When an edge-select bit is set to 1, the corresponding INT pin interrupts on the rising edge.

Port B Pins Input Change Interrupt
- An input change on pins RB7…RB4 sets the flag bit RBIF (INTCON<0>).
- If the RBIE bit is set, then the setting of the RBIF bit causes an interrupt.
- In order to use this interrupt, the RB7…RB4 pins must be configured for input.

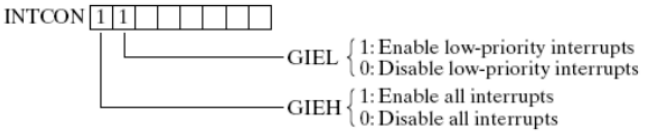
TMR0 Overflow Interrupt
- The Timer0 module contains a 16-bit timer TMR0.
- Timer0 can operate in either 8-bit or 16-bit mode.
- When TMR0 rolls over from 0xFF to 0x00 or from 0xFFFF to 0x0000, it may trigger an interrupt.

Low-priority interrupt structure

- The interrupt priority scheme can be enabled or disabled using IPEN bit
- IPEN = 1 ==> enable priority levels
  IPEN = 0 ==> single interrupt level (all interrupts are high priority)



(a) Initialization for two levels of interrupt priority

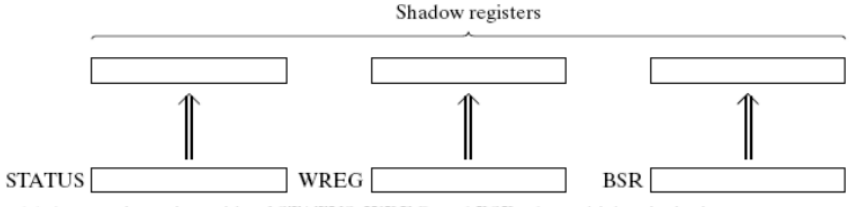


(b) Global interrupt enable bits

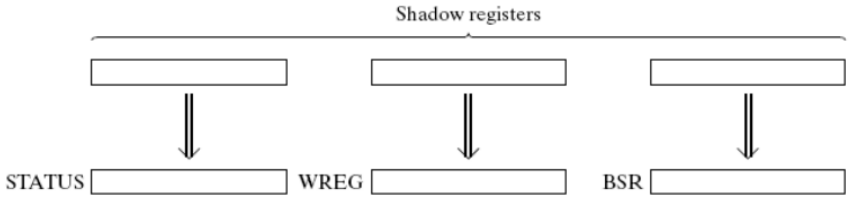
- GIEL is automatically cleared when a low-priority interrupt occurs
- *retfie* sets again the GIEL bit, and pops the PC from stack
- Each interrupt source has associated with it a priority bit
- The default state of all priority bits is 'high' at power-on reset

High-priority interrupt structure

- High-priority interrupts are able to suspend the execution of low-priority ones
- When a high-priority interrupt occurs, STATUS, WREG and BSR registers are automatically copied to shadow registers
- retfie FAST restores the shadow registers, the PC, and sets GIEH bit
- Never use retfie FAST with low-priority interrupts

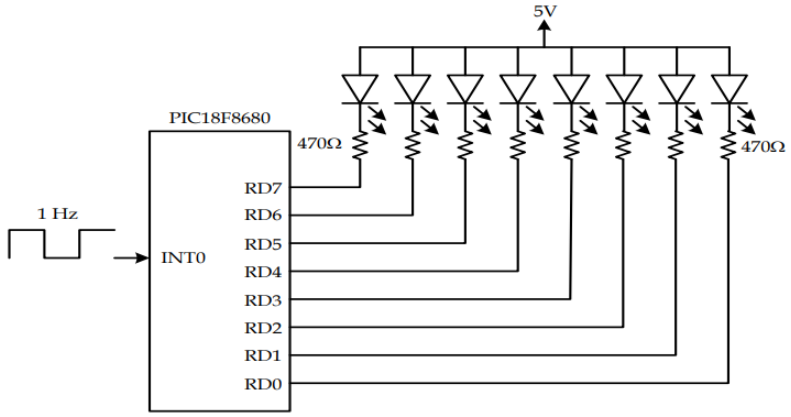


(a) Automatic setting aside of **STATUS**, **WREG**, and **BSR** when a high-priority interrupt occurs



(b) Automatic restoration of **STATUS**, **WREG**, and **BSR** in response to the "**retfie FAST**" instruction.

Example



- By enabling the INT0 interrupt, the INT0 pin will generate an interrupt to the CPU once every second.
- The interrupt service routine simply increments a memory counter, outputs it to the Port D pins, and returns.
- A LED is turned on when its driving Port D pin outputs a low.

Interrupt Programming Template in C

```
void interrupt tc_int(void)
{
    if (TMR0IE && TMR0IF)
    {
            TMR0IF=0;
            tick_count++;
            return;
    }
// process other interrupt sources here, if required
}
```

```
void interrupt low_priority tc_clr(void)
{
        if (TMR1IE && TMR1IF)
        {
                TMR1IF=0;
                tick_count = 0;
                return;
        }
// process any other low priority sources here, if required

}
```

Example -C language version-

```
#include <xc.h>

unsigned char count;

void interrupt service_routine_HP (void)
{
      if (INTCONbits.INT0IF && INTCONbits.TMR0IE)
      {
            INTCONbits.INT0IF=0;
            count--;
            PORTD = count;
      }
// process other interrupt sources here, if required
}

void interrupt low_priority service_routine_LP(void)
{
// Nothing to do for LP INT's in this example
}
```

```
void main(void)
 {
  // Init PIC
  TRISD = 0x00;        // Configure PORTD for output
  count = 0xFF;        // turn off all LEDs initially
  PORTD = count;       //          "
  RCONbits.IPEN = 1;  // enable priority interrupt
  INTCON = 0x90;       // enable GIEH and INT0 interrupt

   while (1);   // Main loop, is an idle loop
                //  that waits for interruptions
 }
```

Context Saving During Interrupts

When an interrupt occurs, the PIC18 MCU saves WREG, BSR, and STATUS in the fast register stack.

- The user can use the **retfie fast** instruction to retrieve these registers before returning from the interrupt service routine.

- One can save additional registers in the stack if the interrupt service needs to modify these registers. These registers must be restored before returning from the service routine.

- In C language, one can add a **save** clause to the **#pragma** statement to inform the C compiler to generate appropriate instructions for saving additional registers.
  #pragma interrupt high_ISR save = reg_1,…, reg_n
  #pragma interrupt low_ISR save = reg1,…, reg_n

- A whole section of data can be saved by the following statements:
  #pragma interrupt high_ISR save = section("section name")
  #pragma interrupt low_ISR save = section("section name")

Saving context in RAM

If retfie fast is not used

EXAMPLE 9-1: SAVING STATUS, WREG AND BSR REGISTERS IN RAM

```
MOVWF    W_TEMP                  ; W_TEMP is in virtual bank
MOVFF    STATUS, STATUS_TEMP     ; STATUS_TEMP located anywhere
MOVFF    BSR, BSR_TEMP           ; BSR_TMEP located anywhere
;
; USER ISR CODE
;
MOVFF    BSR_TEMP, BSR           ; Restore BSR
MOVF     W_TEMP, W               ; Restore WREG
MOVFF    STATUS_TEMP, STATUS     ; Restore STATUS
```

Resets

- Resets can establish the initial values for the CPU registers, flip-flops, and control registers of the interface chips so that the computer can function properly.
- The reset service routine will be executed after the CPU leaves the reset state.
- The reset service routine has a fixed starting address and is stored in the read only memory.
- The PIC18 can differentiate the following types of reset:
  1. Power-on reset (POR)
  2. MCLR pin reset during normal operation
  3. MCLR pin reset during sleep mode
  4. Watchdog timer (WDT) reset (during normal operation)
  5. Programmable brown-out reset (BOR)
  6. RESET instruction
  7. Stack full reset
  8. Stack underflow reset

Resets vs. Interrupts

Reset/interrupts: What are the similarities and differences?

- Similarities:
  1. Both are extraordinary events that cause the MPU to deviate from fetch & decode (i.e., asynchronous)
  2. Both cause the MPU to copy a special address to its Program Counter and execute a different program (ISR).

- Differences:
  1. Resets cause MPU to abort its normal fetch and execute then prepares it to start from scratch
  2. Interrupts cause MPU to abort its normal fetch and execute but returns the MPU to the instruction following the one that was interrupted.

```
Void main(void) {

TRISD = 0xFC;

PORTD = 0x00;

ANSELD= 0x00;

ANSELB = 0x00;

TRISB= 0x06;

IPEN = 1;

INT1IP = 1;

INT2IP = 0;

INT1IE = 1;

INT2IE = 1;

INT1IF = 0;

INT2IF = 0;

GIEL = 1;

GIEH = 1;

WHILE (1) ;

}

Void interrupt_low RSI_low();

IF ( INT2IF == 1 && INT2IE ==1 ) PORTD = 0x00;  INT2IF = 0;

Void interrupt RSI_High();

IF (INT1F == 1 && INT1E == 1) PORTD= 0x01; INT1F  = 0;
```

```
#include <xc.h>
#include "config.h"
void configPIC(){
ANSELA=0x00; // All pins as digital
ANSELB= 0x00;
ANSELC=0x00;
ANSELD=0x00;
ANSELE=0x00;
TRISA=0x00;
TRISB= 0xFD; TRISC=0x00;
TRISD=0x00;
TRISE=0x00;
}
void main(void){
configPIC();
LATC = 0;
int apretat2 = 0;
int apretat3 = 0;
while(1) {
   if(PORTBbits.RB0 == 1) PORTBbits.RB1 = 0;
   else PORTBbits.RB1 = 1;
   if (PORTBbits.RB2 == 1 && apretat2 == 0) {
      LATC++;
      apretat2 = 1;
   }
   else if (PORTBbits.RB2 == 0 && apretat2 == 1) apretat2 = 0;
   if (apretat3 == 1 && PORTBbits.RB3 == 0) {
      LATC = 0;
      apretat3 = 0;
   }
   else if (PORTBbits.RB3 == 1 && apretat3 == 0) apretat3 = 1;
}
}
```

```
PGM   code
Start
      movlw B'00000001'; W=0
      ; Configurar PORTs
      movlb 0xF
      clrf PORTB        ; Netegem i inicialitzem el PORTB
      movlw 0F0h        ; Valor per inicialitzar el ANSELB
      movwf ANSELB      ; Inicialitzar el ANSELB
      movlw 0F1h        ; Valor per inicialitzar el TRISB
      movwf TRISB       ; Inicialitzar el TRISB

Loop
      ; Llegir RB0 i actualitzar RB1
      btfsc PORTB,0,A   ;Miro si RB0==0
      bcf LATB,1
      btfss PORTB,0,A   ;Miro si RB0==1
      bsf LATB,1

      ; Posar el valor a RB2 i esperar uns 40 cicles
      btfss LATB,2,A
      goto RB2es0
RB2es1
      btfsc LATB,2,A    ;Miro si RB2==0
      bcf LATB,2
      goto delay_40cicles
RB2es0
      btfss LATB,2,A    ;Miro si RB2==1
      bsf LATB,2

delay_40cicles
      nop
      nop
      nop
      nop
```

```c
#include <xc.h>
#define _XTAL_FREQ 8000000
#include "config.h"
void configPIC(){
ANSELA=0x00; // All pins as digital
ANSELB= 0x00;
ANSELC=0x00;
ANSELD=0x00;
ANSELE=0x00;
TRISA=0x00;
TRISB= 0xFD; TRISC=0x00;
TRISD=0x00;
TRISE=0x00;
}
void main(void){
configPIC();
LATC = 0x01;
int apretat2 = 0;
int apretat3 = 0;
int apretat4 = 0;
while(1) {
   if(PORTBbits.RB0 == 1) PORTBbits.RB1 = 0;
   else PORTBbits.RB1 = 1;
   if (PORTBbits.RB2 == 1 && apretat2 == 0) {
      __delay_ms(25);
      LATC++;
      apretat2 = 1;
   }
   else if (PORTBbits.RB2 == 0 && apretat2 == 1) apretat2 = 0;
   if (apretat3 == 1 && PORTBbits.RB3 == 0) {
      __delay_ms(25);
      LATC = 0;
      apretat3 = 0;
   }
   else if (PORTBbits.RB3 == 1 && apretat3 == 0) apretat3 = 1;
   if (PORTBbits.RB4 == 1 && apretat4 == 0) {
      __delay_ms(25);
      if (LATC == 0x80) LATC = 0x01;
      else LATC = LATC*2;
      apretat4 = 1;
  }
  else if (PORTBbits.RB4 == 0 && apretat4 == 1) apretat4 = 0;
}
}
```

```c
#include <xc.h>
#include <stdlib.h>
#include "config.h"
#define _XTAL_FREQ 8000000

int numeros[10] = {0x3F, 0x06, 0x5B, 0x4F, 0x66, 0x6D, 0x7D, 0x07, 0x7F, 0x6F};
int apretat[4] = {0x00, 0x00, 0x00, 0x00}; //Indica si RC4, RC5, RC6 i RC7 estan apretats o no
int entrada[4] = {0x00, 0x00, 0x00, 0x00}; //Numeros pel display

void mostrar_num() {
   int display = 0x08;
   for (int i = 0; i < 4; i++) {
      LATD = 0x00;
      LATA = display;
      LATD = numeros[entrada[i]];
      __delay_ms(2);
      display = display >> 1;
   }
}

void main(void)
 {
   ANSELC = 0x00;
   TRISC = 0xFF;
   ANSELA = 0x00;
   TRISA = 0x00;
   ANSELD = 0x00;
   TRISD = 0x00;
   int activat = 1;
   while (1) {
      if (PORTCbits.RC7 == 1 && apretat[3] == 0) {
         int x = rand();
         entrada[3] = x%10;
         entrada[2] = (x/10)%10;
         apretat[3] = 1;
      }
      if (PORTCbits.RC7 == 0 && apretat[3] == 1) apretat[3] = 0;
      if (PORTCbits.RC5 == 1 && apretat[1] == 0) {
         int x = rand();
         entrada[1] = x%10;
         entrada[0] = (x/10)%10;
         apretat[1] = 1;
      }
      if (PORTCbits.RC5 == 0 && apretat[1] == 1) apretat[1] = 0;
      if (PORTCbits.RC6 == 0 && apretat[2] == 1) {
         apretat[2] = 0;
         int x, y;
         x = (entrada[0]*10) + entrada[1];
         y = (entrada[2]*10) + entrada[3];
         x = x*y;
         entrada[3] = x%10;
         x = x/10;
         entrada[2] = x%10;
         x = x/10;
         entrada[1] = x%10;
         x = x/10;
         entrada[0] = x%10;
      }
      if (PORTCbits.RC6 == 1 && apretat[2] == 0) apretat[2] = 1;
      if (PORTCbits.RC4 == 0 && apretat[0] == 1 && activat == 1) {
         LATA = 0x00;
         apretat[0] = 0;
         activat = 0;
      }
      if (PORTCbits.RC4 == 0 && apretat[0] == 1 && activat == 0) {
         activat = 1;
         apretat[0] = 0;
      }
      if (PORTCbits.RC4 == 1 && apretat[0] == 0) apretat[0] = 1;
      if (activat) mostrar_num();
```

```c
#include <xc.h>
#include <stdlib.h>
#include "config.h"
#define _XTAL_FREQ 8000000

int numeros[10] = {0x3F, 0x06, 0x5B, 0x4F, 0x66, 0x6D, 0x7D, 0x07, 0x7F, 0x6F};
int apretat[3] = {0x00, 0x00, 0x00}; //Indica si RB0, RB1 i RB2 estan apretats o no
int entrada[4] = {0, 0, 0, 0}; //Numeros pel display
int contador = 1;
int num = 0;

void mostrar_num() {
   int display = 0x08;
   for (int i = 0; i < 4; i++) {
      LATD = 0x00;
      LATA = display;
      LATD = numeros[entrada[i]];
      __delay_ms(2);
      display = display >> 1;
   }
}

void interrupt RSI_high() {
   if (INT0IE == 1 && INT0IF == 1) {
      LATD = 0x00;
      num = rand()%10000;
      entrada[3] = num%10;
      entrada[2] = (num/10)%10;
      entrada[1] = (num/100)%10;
      entrada[0] = (num/1000)%10;
      mostrar_num();
      INT1IE = 1;
      INT2IE = 1;
      INT0IF = 0;
   }
   else if (INT1IE == 1 && INT1IF == 1) {
      LATD = 0x00;
      num = num + contador;
      if (num > 9999) num = 9999;
      entrada[3] = num%10;
      entrada[2] = (num/10)%10;
      entrada[1] = (num/100)%10;
      entrada[0] = (num/1000)%10;
      mostrar_num();
      INT1IF = 0;
   }
   else if (INT2IE == 1 && INT2IF == 1) {
      if (contador == 1000) contador = 1;
      else contador = contador*10;
      INT2IF = 0;
   }
}

void main(void)
 {
   ANSELA = 0x00;
   TRISA = 0x00;
   ANSELD = 0x00;
   TRISD = 0x00;
   TRISB = 0x07;
   ANSELB = 0x00;
   IPEN = 1;
   INTEDG0 = 1;
   INTEDG1 = 1;
   INTEDG2 = 0;
   INT0IE = 1;
   INT1IE = 0;
   INT2IE = 0;
   INT0IF = 0;
   INT1IF = 0;
   INT2IF = 0;
   GIEH = 1;
   mostrar_num();
   while (1) mostrar_num();
```

```c
#include <xc.h>
#include <stdlib.h>
#include "config.h"
#define _XTAL_FREQ 8000000

int numeros[10] = {0x3F, 0x06, 0x5B, 0x4F, 0x66, 0x6D, 0x7D, 0x07, 0x7F, 0x6F};
int apretat[4] = {0x00, 0x00, 0x00, 0x00}; //Indica si RC4, RC5, RC6 i RC7 estan apretats o no
int entrada[4] = {0x00, 0x00, 0x00, 0x00}; //Numeros pel display

unsigned int const ASCII[96] = { 0x00, /* (space) */ 0x86, /* ! */ 0x22, /* " */ 0x7E, /* # */ 0x6D,
/* $ */ 0xD2, /* % */ 0x46, /* & */ 0x20, /* ' */ 0x29, /* ( */ 0x0B, /* ) */ 0x21, /* * */ 0x70,
/* + */ 0x10, /* , */ 0x40, /* - */ 0x80, /* . */ 0x52, /* / */ 0x3F, /* 0 */ 0x06, /* 1 */ 0x5B,
/* 2 */ 0x4F, /* 3 */ 0x66, /* 4 */ 0x6D, /* 5 */ 0x7D, /* 6 */ 0x07, /* 7 */ 0x7F, /* 8 */ 0x6F,
/* 9 */ 0x09, /* : */ 0x0D, /* ; */ 0x61, /* < */ 0x48, /* = */ 0x43, /* > */ 0xD3, /* ? */ 0x5F,
/* @ */ 0x77, /* A */ 0x7C, /* B */ 0x39, /* C */ 0x5E, /* D */ 0x79, /* E */ 0x71, /* F */ 0x3D,
/* G */ 0x76, /* H */ 0x30, /* I */ 0x1E, /* J */ 0x75, /* K */ 0x38, /* L */ 0x15, /* M */ 0x37,
/* N */ 0x3F, /* O */ 0x73, /* P */ 0x6B, /* Q */ 0x33, /* R */ 0x6D, /* S */ 0x78, /* T */ 0x3E,
/* U */ 0x3E, /* V */ 0x2A, /* W */ 0x76, /* X */ 0x6E, /* Y */ 0x5B, /* Z */ 0x39, /* [ */ 0x64,
/* \ */ 0x0F, /* ] */ 0x23, /* ^ */ 0x08, /* _ */ 0x02, /* ` */ 0x5F, /* a */ 0x7C, /* b */ 0x58,
/* c */ 0x5E, /* d */ 0x7B, /* e */ 0x71, /* f */ 0x6F, /* g */ 0x74, /* h */ 0x10, /* i */ 0x0C,
/* j */ 0x75, /* k */ 0x30, /* l */ 0x14, /* m */ 0x54, /* n */ 0x5C, /* o */ 0x73, /* p */ 0x67,
/* q */ 0x50, /* r */ 0x6D, /* s */ 0x78, /* t */ 0x1C, /* u */ 0x1C, /* v */ 0x14, /* w */ 0x76,
 /* x */ 0x6E, /* y */ 0x5B, /* z */ 0x46, /* { */ 0x30, /* | */ 0x70, /* } */ 0x01, /* ~ */ 0x00 /* (del) */};

int CAFE[]= {0x39,0x77,0x71,0x79,0x00};
int pointer = 0;
int contador_izq = 0;
int contador_der = 0;
int mov_ant = 0; //0 = se movia a la izquierda, 1 = se movia a la derecha
int mov_izq = 0;
int mov_der = 0;
int parar = 1;

void mover_izq() {
     pointer = pointer %5;
     ++contador_izq;
     int port_a = 0x01;
     for(int i = 3; i >= 0; --i) {
        PORTD = 0x00;
        PORTA = port_a;
        PORTD = CAFE[(pointer + i) % 5];
        __delay_ms(3);
        port_a = port_a << 1;
     }

     if (contador_izq == 41) {
     ++pointer;
     contador_izq = 0;
     }
  }

void mover_der() {
   pointer = pointer %5;
   ++contador_der;
   int port_a = 0x08;
   for(int i = 0; i <= 3; ++i) {
      PORTD = 0x00;
      PORTA = port_a;
      PORTD = CAFE[(pointer + i) % 5];
      __delay_ms(3);
      port_a = port_a >> 1;
   }

   if (contador_der == 41) {
   --pointer;
   if (pointer < 0) pointer = 4;
   contador_der = 0;
   }
}

void parar_mov() {
     pointer = pointer %5;
     int port_a = 0x08;
     for(int i = 0; i <= 3; i++) {
        PORTD = 0x00;
        PORTA = port_a;
        PORTD = CAFE[(pointer + i) % 5];
        __delay_ms(3);
        port_a = port_a >> 1;
     }
}

void interrupt RSI() {
if (INT0IE == 1 && INT0IF == 1) {
     for (int i = 0; i < 15; i++) parar_mov();
     if (parar == 0) {
        parar = 1;
        if (mov_izq == 1) {
           mov_ant = 0;
           mov_izq = 0;
        }
        else {
           mov_ant = 1;
           mov_der = 0;
        }
     }
     else {
        parar = 0;
        if (mov_ant == 0) mov_izq = 1;
        else mov_der = 1;
     }
     INT0IF = 0;
  }
  else if (INT1IE == 1 && INT1IF == 1) {
     mov_der = 0;
     mov_izq = 1;
     INT1IF = 0;
  }
  else if (INT2IE == 1 && INT2IF == 1) {
     mov_izq = 0;
     mov_der = 1;
     INT2IF = 0;
  }
}

void main(void)
 {
   ANSELA = 0x00;
   TRISA = 0x00;
   ANSELD = 0x00;
   TRISD = 0x00;
   TRISB = 0x07;
   ANSELB = 0x00;
   IPEN = 1;
   INTEDG0 = 1;
   INTEDG1 = 1;
   INTEDG2 = 1;
   INT0IE = 1;
   INT1IE = 1;
   INT2IE = 1;
   INT0IF = 0;
   INT1IF = 0;
   INT2IF = 0;
   GIEH = 1;
   while (1) {
      if (parar == 0) {
         if (mov_izq == 1) mover_izq();
         else mover_der();
      }
      if (parar == 1 || (mov_izq == 0 && mov_der == 0)) parar_mov();
   }
 }
```

Què és un computador?

Un computador consta de hardware i software; El hardware es basa en quatre tipus de components:

- Processador: Responsable de realitzar totes les operacions computacionals i la coordinació de l'ús dels recursos de l'ordinador. Un sistema computacional pot constar d'un o més processadors. Un processador pot realitzar tasques de propòsit general o de propòsit específic, com renderització gràfica, impressió...
- Dispositius d'entrada: Un computador està dissenyat per a executar programes que manipules certes dades. Els dispositius d'entrada són necessaris per a entrar al programa i entrar-hi les dades per a ser processades. Teclats, ratolins, scanners, sensors...
- Dispositius de sortida: Els resultats finals han de ser mostrats en una pantalla o impresos en paper per tal que l'usuari els pugui veure (LEDs, etc). Aquests són els dispositius de sortida. - Dispositius de memòria: Perquè els programes puguin ser executats i les dades puguin ser processades s'han de guardar en memòria perquè el processador pugui accedir-hi.

El processador

El processador, també anomenat CPU està constituït al menys d'aquests tres components:

- Registres: Un registre és un lloc d'emmagatzematge dins la CPU. S'utilitza per a contenir dades o adreces de memòria durant l'execució d'una instrucció. Com que els registres estan molt aprop de la CPU, permet un accés molt ràpid durant l'execució del programa. El nombre de registres varia molt entre processadors.
- ALU (Arithmetic Logic Unit): La ALU realitza totes les computacions numèriques i lògiques. La ALU rep dades de la memòria, fa les operacions i si és necessari reescriu les dades a la memòria de nou.
- Unitat de Control: La unitat de control conté la lògica de les instruccions del hardware. L'unitat de control descodifica i monitoritza les execucions de les instruccions. També actua com un àrbitre ja que varies porcions del computador necessiten els recursos de la CPU. Les activitats de la CPU estan sincronitzades amb el rellotge del sistema. La unitat de control també conté un registre anomenat program counter (PC) que manté un seguiment de les adreces de les següents instruccions a ser executades.

El microprocessador

Un microprocessador és un processador integrat en un sol circuit. Un microcomputador és un computador que utilitza un microprocessador com a CPU. La velocitat de rellotge ha anat millorant de manera increïble amb els anys, però la velocitat d'accés a memòria ha millorat molt poc; això provoca que el processador pot fer operacions aritmètiques en un cicle de rellotge però necessita forces cicles per accedir a les dades de memòria. Aquesta diferència fa que la rapidesa del rellotge sigui inútil per a millorar el rendiment.

La solució al problema, és afegir una petita memòria d'alta velocitat a la CPU; l'anomenada Memòria Cache. La CPU pot accedir a dades de la seva pròpia cache amb tan sols un o dos cicles perquè està molt aprop de la ALU.

Els microcontroladors

Un microcontrolador (MCU) es un computador implementat en un sol circuit de gran escala integrat. Amés d'aquests components que estàn compresos en un microprocessador, un MCU també conté alguns dels següents elements:

- Memòria
- Temporitzadors, comptadors d'events, capturadors d'entrades, de sortides, comparadors de sortides, etc...
- PWM
- Conversos Analògic-Digital
- Conversos Digital-Analògic
- Controlador de memòria d'accés directe

LA MEMÒRIA

Els programes i les dades estan emmagatzemats en memòria. Es classifica en dos tipus Random Access Memory (RAM) i Read-Only Memory (ROM).

RANDOM-ACESS MEMORY (RAM)
La memòria RAM, és **volàtil**; és a dir, no pot retenir les dades amb l'absència d'energia (electricitat). A la RAM també se l'anomena Memòria de Lectura/Escriptura, perquè **permet que el processador llegeixi i escrigui en la memòria**. Tant l'escriptura com la lectura suposen la mateixa quantitat de temps. Sempre que el dispositiu tingui energia, el microprocessador pot escriure en una localització de la RAM i llegir de nou els mateixos continguts quan sigui necessari. Llegir la memòria és un procés no-destructiu; en canvi, quan el micro escriu dades a memòria, les dades antigues són substituïdes, i per tant destruïdes.

Existeixen dos tipus de RAM: **Static RAM** (SRAM) i **Dynamic RAM** (DRAM). La **SRAM** utilitza de quatre a sis transistors per a guardar un bit d'informació; sempre que el dispositiu tingui energia, la informació de la SRAM no es veurà afectada (borrada). La **RAM dinàmica (DRAM)**, utilitza un transistor i un capacitor per a guardar un bit d'informació. La informació guardada en el capacitor (en forma de càrrega elèctrica) es perdrà amb el temps, així que és necessari una operació d'actualització per a mantenir els continguts d'aquesta. La RAM, s'utilitza principalment per a guardar programes dinàmics i dades; un ordinador sovint vol córrer diferents programes a la vegada, i aquests programes normalment utilitzen diferents sets de dades. Els programes i les dades, doncs, han de carregar-se des de un disc dur o memòries secundaries cap a la RAM, és per això que se'ls hi diu dinàmiques.

READ-ONLY MEMORY (ROM)
La memòria ROM no és volàtil, si es treu l'energia de la ROM i després es torna a aplicar les dades originals encara seran allà. Com implica el seu nom, la ROM només pot ser llegida; tot i que això no es del tot cert. La majoria de ROM requereixen algoritmes especials i voltatge per a escriure dades al xip. Sense l'algoritme i el voltatge específics qualsevol intent de escriure-hi no tindrà èxit. Avui en dia hi ha molts diferents tipus de ROM:

- Masked-programmed read-only memory:
  - Programada durant la fabricació.
- Programable read-only memory
  - Rom que pot ser programada per a l'usuari final utilitzant un PROM (Programador de ROM)
  - Un cop programada no pot ser canviada.
- Erasable programable read-only memory
  - Pot ser borrada amb llum ultraviolat
  - Després pot ser reescrita
- Electrically erasable programable read-only memory
  - Pot ser borrada i reprogramada amb electricitat.
- Flash Memory
  - Va ser creada per a incorporar els vantatges i treure les desavantatges de EPROM i EEPROM. La memòria flash pot ser borrada i reprogramada en el sistema sense un programa dedicat o un programador.

El software del Computador

Els programes es coneixen com a software. Un programa es una sèrie d'instruccions que el computador pot executar, es guarda a la memòria del computador en forma de nombres binaris: Instruccions Màquina. La llargada de una instrucció màquina d'un computador pot ser fixe o variable; si fixem la mida de les instruccions, fa que descodificar-les sigui molt més simple i per tant podem simplificar el disseny del processador; però té un gran desavantatge, la llargària del programa pot ser molt més llarga a causa de la ineficiència de la codificació de les instruccions. La gran majoria d'instruccions són de 16 bits, però n'hi ha quatre que són de 32 bits.

LLENGUATGE ASSEMBLADOR
Desenvolupar programes en llenguatge màquina (binari) és molt difícil; El llenguatge assemblador es va inventar per a simplificar la feina de programar. Els mnemònics són representacions en no-alt nivell del llenguatge màquina, per exemple en el PIC18:

Decf fp_cnt, F, A significa "decrementa la variable lp_cnt localitzada al access bank 1 unitat".

Llenguatges com C, C++ o Java són llenguatges d'alt nivell, ja que s'acosten al anglès parlat i sovint una sola línia representa moltes instruccions en assemblador.

Vista general del PIC18

ORGANITZACIÓ DE MEMÒRIA
La memòria consisteix en una seqüència de ubicacions directament adreçables. Una ubicació de memòria es referida com una unitat d'informació; en el PIC18, una ubicació de memòria conté 8 bits d'informació, és a dir, 1 Byte; la meitat (4bits) són 1 nibble. Una ubicació de memòria pot ser utilitzada per a guardar dades, instruccions, estats dels perifèrics, etc. Una unitat d'informació té dos components: La seva *adreça* i el seu *contingut*.

ADREÇA -> -> CONTINGUTS
Cada ubicació de memòria, té una adreça que ha de ser proporcionada abans de que els seus continguts puguin ser accedits. La CPU es comunica amb la memòria primer identificant l'adreça i llavors passant-la per el bus d'adreçes; similar al fet que un carter necessita l'adreça per a donar la carta.
Per tal de diferenciar els continguts d'un registre amb l'adreça d'un registre o una ubicació de memòria, utilitzarem les següents notacions:

- **[adreça de registre]**: Es refereix als continguts d'un registre; [WREG] es refereix als continguts del registre WREG; [0x20] es refereix als continguts del registre de propòsit general a l'adreça 0x20.
- **Adreça**: Es refereix al registre o ubicació de memòria; 0x10 es refereix al registre de funció especial 0x10.

SEPARACIÓ DE LA MEMÒIRA DE DADES I LA MEMÒRIA DE PROGRAMES

El PIC18, assigna diferents espais de memòria a les dades i als programes i destina busos separats entre ells per què puguin accedir a la vegada. El PIC18 té un PC de 21 bits dividit en tres registres: PCU, PCH i PCL. Entre ells, el PCL és accessible directament per l'usuari. El PCH i el PCL tenen 8 bits mentre que el PCU té 5 bits.

MEMÒRIA DE DADES
La memòria de dades del PIC18 està implementada com a SRAM, cada ubicació de memòria de dades és referida com a registre o registre de fitxers. El PIC18 suporta 4096 bytes de dades en memòria. Es requereixen 12 bits d'adreça per a seleccionar un dels registres.

A causa de la mida limitada de les instruccions del PIC18 (16 bits la majoria), només 8 bits de l'instrucció queden lliures per a especificar el registre que operar. Com a resultat, els dissenyadors van dividir els 4096 bytes de la memòria de dades del PIC18 en 16 bancs (Banks). Només un sol banc de 256 registres pot estar actiu a l'hora. Quatre bits addicionals es posen en un registre especial anomenat BSR (Bank Select Register) per a seleccionar el banc actiu; l'usuari ha de canviar els continguts del BSR per a canviar el banc actiu.

Hi ha dos tipus de registres: Registres de propòsit general (GPRs) i registres de funcions especials (SFRs). Els GPRs s'utilitzen per a guardar dades dinàmiques quan la CPU del PIC18 està executant un programa; els SFRs, són utilitzats per la CPU i els perifèrics per a controlar les ordres del MCU; els registres estan implementats com a SRAM.

Els SFRs estan assignats de la més alta adreça cap avall, mentre que els GPRs comencen del l'adreça 0 cap a dalt.

[memòria de dades eeprom ---------- 1.5.3 Han-Way Huang // per llògica es pot deduïr -> Electrically eraseable programable rom]

ORGANITZACIÓ DE MEMÒIRA DE PROGRAMA

El PIC18, com ja hem dit, té un PC de 21 bits i en conseqüència és capaç de adreçar els 2MB d'espai de memòria de programa. Si accedim una ubicació inexistent de memòria, llegirem tot 0s.

La MCU del PIC18 té 31 entrades d'adreces de retorn (un stack) per a guardar adreces de retorn de crides a subrutines i interrupcions de processos. Aquesta pila no és part de l'espai de la memòria de programa.

6. Interrupcións

Una interrupció és un mecanisme proveït per un microprocessador o un sistema per a sincronitzar operacións d'E/S, manejar condicións d'errors i events d'emergència, coordinar l'ús de recursos compartits...

1 - CONCEPTES BÀSICS.

Què es una interrupció?

Una interrupció es un event que requereix que la CPU abandoni l'execució normal del programa i executi un servei relacionat amb l'event. Una interrupció pot ser generada internament (dins el xip) o externament (fora del xip). Una **interrupció externa** és generada quan els cirucits externs provoquen una senyal d'interrupció a la CPU. Una **interrupció interna** pot ser generada per el propi circuit del hardware del xip o per errors de software. En alguns microcontroladors, temporitzadors, funcions d'E/S i la CPU estàn incorporades en el mateix xip, i aquests subsistemes poden generar interrucions a la CPU. També poden ser causades per accións abnormals de l'execució del programa: *codis d'operació il·legals, overflows, divisións per zero...* són les anomenades **interrupcións de software**. Els termes *traps* o *excepcións* també són utilitzats per a referir-se a les interrupcións de software.

```
Exemple d'interrupció (analogia):
Quan estàs sentat al teu escriptori llegint un llibre, i de cop sona el telèfon; segurament actuaràs de la següent manera:

1.  Recordar el nombre de pàgina o posar un punt de llibre a la pàgina que estàs llegint.
2. Agafar el telèfon i dir "Hola, etc etc..."
3. Escoltar la veu del telèfon per saber si la veu és familiar o has de preguntar qui és.
4. Parlar amb la persona.
5. Penjar el telèfon quan acabes de parlar.
6. Obrir el llibre de nou i tornar on ho havies deixat.
```

Amb aquest exemple podem explicar algunes coses molt similars a com un microprocessador actua amb una interrupció:

1. Com a estudiant, tu passes la major part del temps estudiant. Contestar una trucada, només passa ocasionalment. De manera similar, el microprocessador està normalment executant programes la major part del temps. `Les interrupcións només forcen al mircroprocessador a parar l'execució de l'aplicació momentàniament i realitzar accións necessàries`.

2. Abans de agafar el telèfon, acabes de llegir la frase i llavors poses un punt de llibre per a recordar la pàgina que estaves llegint per acabar-la desrés de parlar per telèfon. La majoria dels microprocessadors necessiten identificar la causa de l'interrupció que estàn executant i `guardar l'adreça de la següent instrucció de memòria` (normalment a la pila) per tal de poder continuar amb el programa quan acabi l'interrupció.

3. Saps qui és la persona que et truca escoltant la seva veu a través del telèfon, o preguntat-li algunes coses per tal de saber qui és. De manera similar el micro `necessita identificar la causa de l'interrupció` abans de poder fer les accións apropiades. Això està dissenyat en el hardware del micro.

4. Després d'identificar la persona que t'ha trucat, comences una conversa amb aquella persona amb els temes de conversa adequats. De manera similar `el micro actuarà de la manera apropiada per a la font de l'interrupció`

5. Quan acabes la conversa per telèfon, penjes, obres el llibre per la pàgina on havies deixat el punt i continues amb la lectura. Similarment, després de fer les accións necessàries i apropiades per l'interrupció, el micro tornarà a la següent instrucció on ha tingut lloc l'interrupció. Això és gràcies a que l'adreça on s'havia de continuar havia estat guardada a la memòria (stack).

Emmascarament d'interrupcións

Depenent de la situació i l'aplicació, algunes interrupcións podrien ser no desitjades o necessàries i s'haurien de prevenir d'interrompre la CPU. La majoria dels microprocessadors i microcontroladors tenen l'opció d'ignorar aquestes interrupcións. Aquests tipus d'interrupcións s'anomenen **interrupcións emmascarables**. Hi ha alguns tipus d'interrupcións que no poden ser ignorats per la CPU i per les quals s'han de pendre mesures immediatament (**interrupcións no emmascarables**). Un programa pot demanar a la CPU ignorar o fer cas a les interrupcións emmascarables a través de posar a 1 o 0 el "*enable bit*". Una interrupció es marca com a **pendent** quan està activa però encara no ha estat tractada per la CPU. Una *interrupció pendent* pot ser o no ser tractada per la CPU, depenen de l'*enable bit* de l'interrupció.

Prioritat d'interrupcións

Pot ser que en un moment donat, la CPU rebi més d'una interrupcións pendents a la vegada. La CPU ha de **decidir quina interrupció ha de tractar primer**. La solució és prioritzar totes les fonts d'interrupcións. Una interrupció amb prioritat més alta sempre rebrà tractament que les de prioritat més baixa.

Vector d'interrupcións

Per a proveïr servei a una interrupció, el processador ha de saber l'adreça d'inici de la rutina de tractament. Aquesta adreça s'anomena **vector d'interrupcións**.

El pic18, només té dos vectors d'interrupció per a tractar totes les interrupcións. Aquestes interrupcións estàn classificades en alta i baixa prioritat.

2.INTERRUPCIÓNS EN EL PIC18

El PIC18 té les següents fonts d'interrupcións:

- INT pin (INT0 ... INT3)
- Canvis de pins del PortB (qualsevol dels quatre ports superiors de PORTB)
- Perifèrics del xip

Registres relacionats amb interrupcións

Hi ha fins a 13 registres per a controlar les operacións d'interrupció. Aquests són:

- RCON
- INTCONx (x = 2, 3; l'1 no porta x)
- PIR1, PIR2 i PIR3
- PIE1, PIE2, i PIE3
- IPR1, IPR2 i IPR3

Cada font d'interrupcións te tres bits per a controlar les seves operacións, aquests són:

- *Flag bit*: indica si un event d'interrupció ha passat.
- *Enable bit*: habilita o deshabilita una interrupció.
- *Priority bit*: Selecciona si és alta o baixa prioritat. Només funciona quan l'esquema de prioritats està activat.

Registre RCON

Aquest registre té un bit (IPEN) per a activar l'esquema de prioritats. Els altres bits s'utilitzen per a indicar *causes de reset*.

Registres INTCON - Interrupt Control Registers

Aquests registres conenen bits d'*enable, priority i flag* per pins INT externs, pins PORTB canviats i interrupcións de Timer 0 (TMR0) per overflow.

El pin INT0 no té un *priority bit* per a seleccionar la seva prioritat. De fet, aquest pin sempre està en **alta** prioritat, ja que apareix en els dos circuits d'alta i baixa prioritat.

`Quan l'esquema de prioritats està actiu, l'usuari ha de posar el git GIEH (bit 7) per tal de permetre les interrupcións de baixa prioritat. Posar el bit GIEL sense el bit GIEH no activarà les interrupcións de baixa prioritat.`

BIT 7 de INTCON anomenat **GIE/GIEH** (Global interrupt enable bit)
Si IPEN (Bit7 del RCON) = 0

- 0 desactiva totes les interrupcións
- 1 les activa

Si IPEN (Bit 7 del RCON) = 1

- 0: Desactiva totes les interrupcións.
- 1: Activa totes les interrupcións **d'alta prioritat**.

BIT 6 de INTCON anomenat **PEIE/GIEL** (Peripherial interrupt enable bit) Si IPEN (Bit 7 del RCON) = 0

- 0: Desactiva totes les interrupcións de perifèrics.
- 1: Activa totes les interrupcións de perifèrics.

Si IPEN (Bit 7 del RCON) = 1

- 0: Desactiva totes les interrupcións **de baixa prioritat**
- 1: Activa totes les interrupcións **de baixa prioritat**

Registres PIR1 . . . PIR3

Aquests registres contenen els bits de *flag* individuals per a les interrupcións perifèriques. Aquests *flag bits* permeten a la rutina de tractament identificar la causa de l'interrupció. Segons el model de PIC18, els bits poden varirar.

Registres PIE1 . . . PIE3

Contenen bits *enable* individuals per a les interrupcións perifèriques. `quan el bit IPEN (Bit 7 del RCON) és 1, el bit PEIE ha de ser posat a 1 per a permetre qualsevol d'aquestes interrupcións perifèriques`.

Registres IPR1 . . . IPR3 - *Interrupt Priority Registers*

Contenen els bits de prioritat individuals per a les interrupcións perifèriques. Aquests registres només etenn efecte quan el esquema de prioritats d'interrupcións està activat (IPEN = 1). Activant la prioritat d'interrupcións i posant el bit de prioritat associat, l'usuari pot posar qualsevol interrupció perifèrica a alta o baixa prioritat.