3.3 I/O port

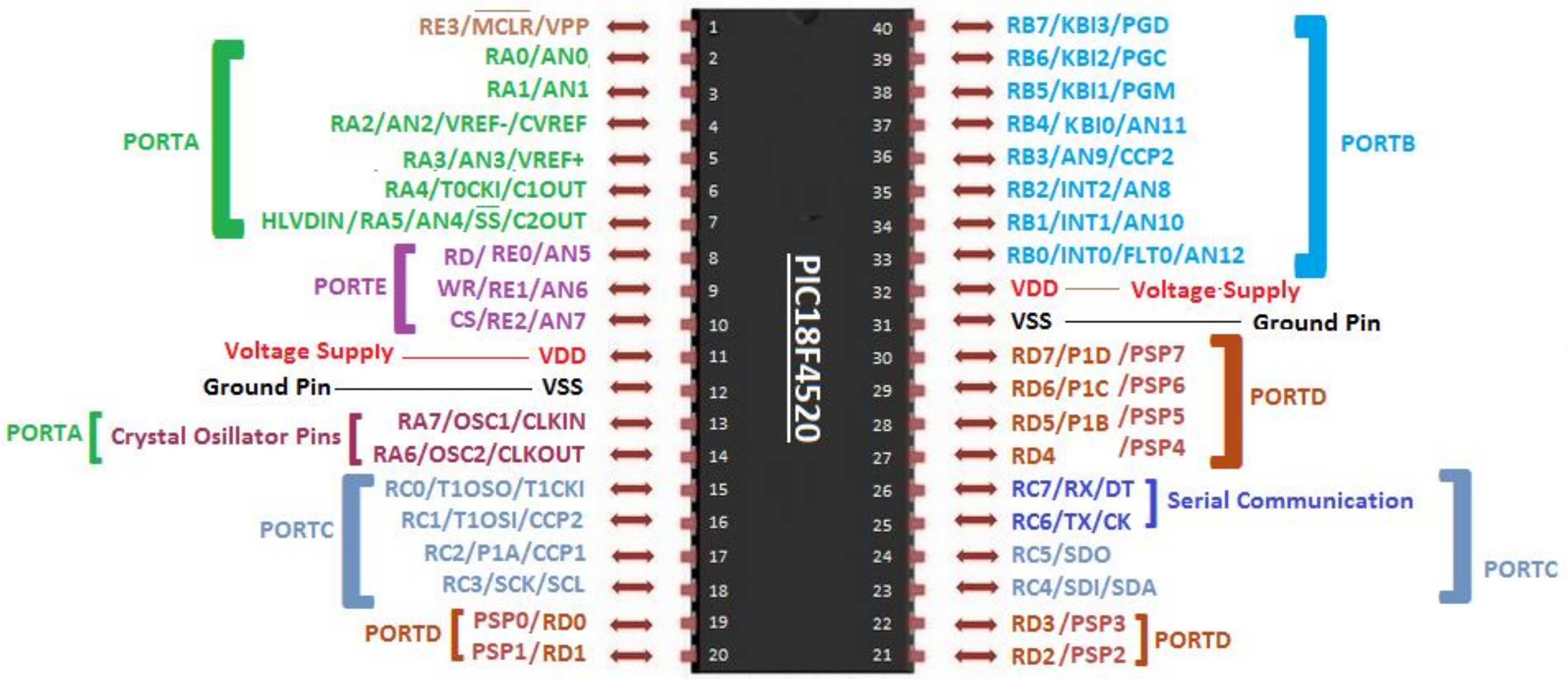
- I/O port programming
- I/O bit manipulation programming

3.3.1 I/O port programming

- microcontroller has many ports for I/O operations
- I/O ports are used to interact, monitor and control peripherals
- the number of ports in the PIC18 family depends on the number of pins on the chip
- PIC18F4520 has five ports: Ports A-E

Pins	**18**	**28**	**40**	**64**	**80**
Chip	**PIC18F1220**	**PIC18F2220**	**PIC18F458**	**PIC18F6525**	**PIC18F8525**
Port A	√	√	√	√	√
Port B	√	√	√	√	√
Port C		√	√	√	√
Port D			√	√	√
Port E			√	√	√
Port F				√	√
Port G				√	√
Port H				√	√
Port J				√	√
Port K					√
Port L					√

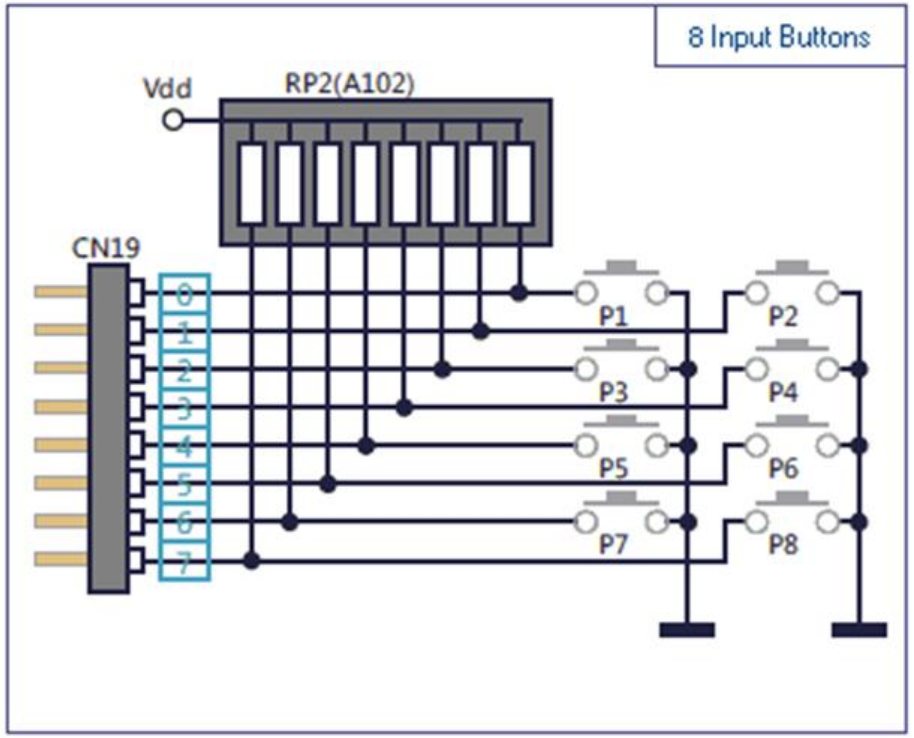
40 pin PDIP

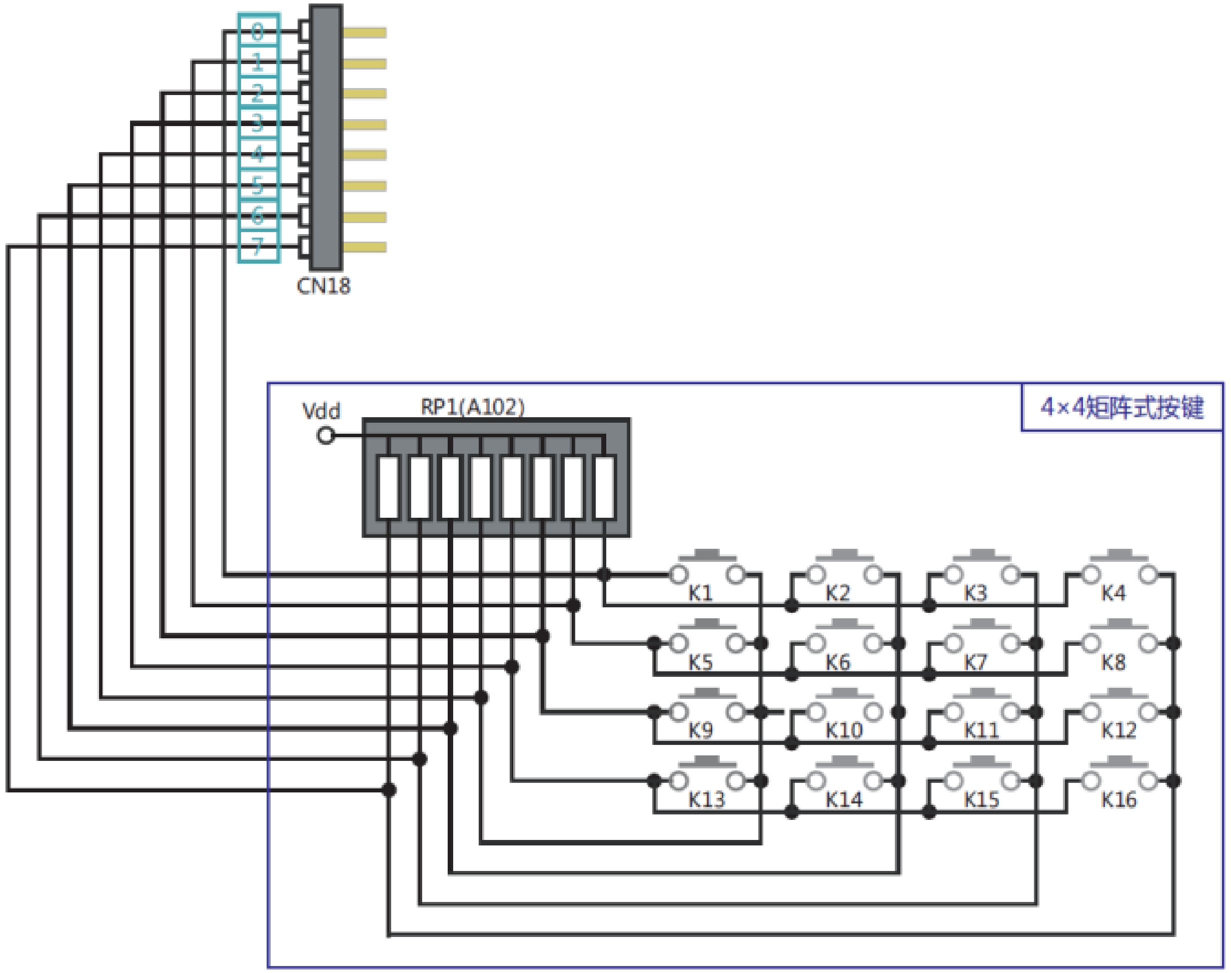


PIC18F4520
RE3/MCLR/VPP 1
PORTA
RA0/AN0 2
RA1/AN1 3
RA2/AN2/VREF-/CVREF 4
RA3/AN3/VREF+ 5
RA4/T0CKI/C1OUT 6
HLVDIN/RA5/AN4/SS/C2OUT 7
PORTE
RD/ RE0/AN5 8
WR/RE1/AN6 9
CS/RE2/AN7 10
Voltage Supply VDD 11
Ground Pin VSS 12
PORTA Crystal Osillator Pins
RA7/OSC1/CLKIN 13
RA6/OSC2/CLKOUT 14
PORTC
RC0/T1OSO/T1CKI 15
RC1/T1OSI/CCP2 16
RC2/P1A/CCP1 17
RC3/SCK/SCL 18
PORTD
PSP0/RD0 19
PSP1/RD1 20
40 RB7/KBI3/PGD
39 RB6/KBI2/PGC
38 RB5/KBI1/PGM
37 RB4/ KBI0/AN11
36 RB3/AN9/CCP2
35 RB2/INT2/AN8
34 RB1/INT1/AN10
33 RB0/INT0/FLT0/AN12
PORTB
32 VDD Voltage Supply
31 VSS Ground Pin
30 RD7/P1D /PSP7
29 RD6/P1C /PSP6
28 RD5/P1B /PSP5
27 RD4 /PSP4
PORTD
26 RC7/RX/DT
25 RC6/TX/CK
Serial Communication
24 RC5/SDO
23 RC4/SDI/SDA
PORTC
22 RD3/PSP3
21 RD2/PSP2
PORTD

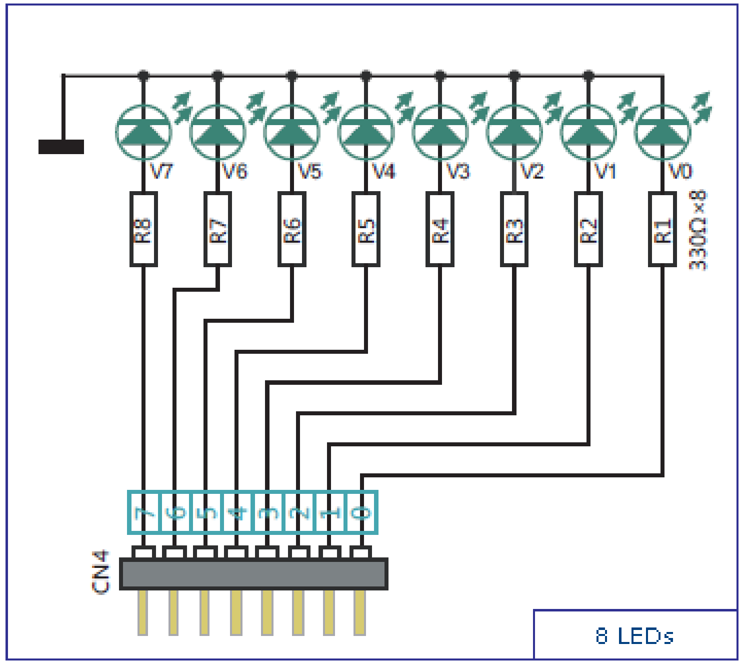
- PIC18F4520 sets aside 35 pins for the five I/O ports
- other pins are V_{DD} (*voltage supply*), V_{SS} (*ground*), and MCLR (*reset*)
- different ports have different number of pins
  - Ports A, B, C, D: 8 pins
  - Port E: 3 pins
- to use any one port as input or output, it must be programmed
- I/O port may have other functions, e.g. ADC, timers, etc.

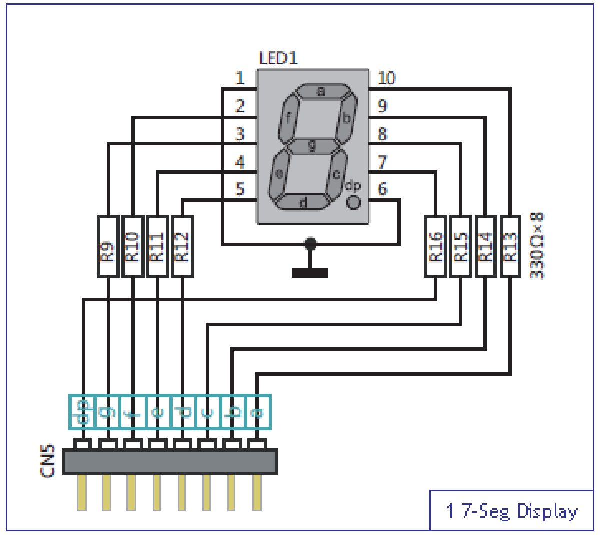
Peripherals - input

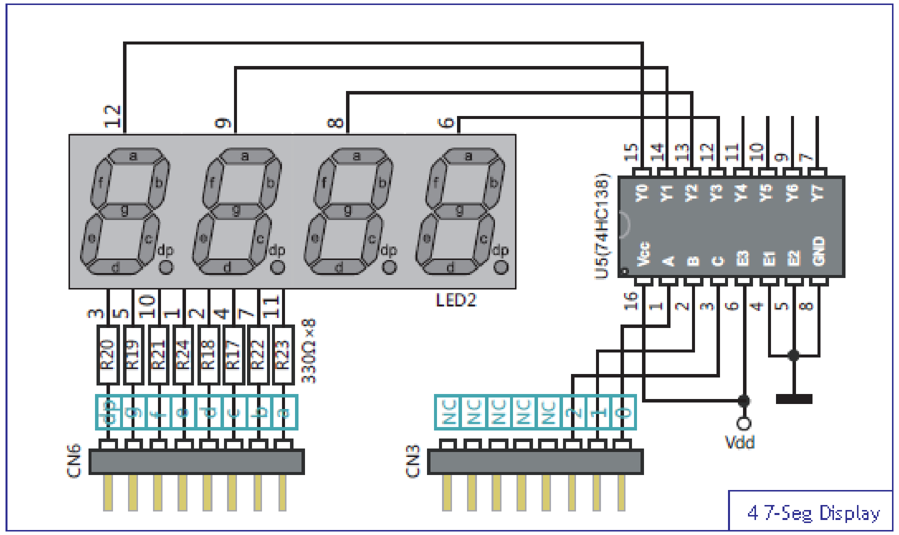




Peripherals - output







- each I/O port has three SFRs associated with it

  PORTx – indicates the voltage levels on the pins of the device

  TRISx – data direction register

  LATx – stands for latch

- e.g. Port B has PORTB, TRISB and LATB

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

1-

- each of the Ports A-E can be used for input and output

- TRISx SFR is used for designating the direction of a port
  - 0 for output (e.g. controlling LED state)
  - 1 for input (e.g. scan input button)

- to output data to Port C
  - Write 0s to TRISC SFR
  - Write data to PORTC SFR

- to input data from Port D
  - Write 1s to TRISD SFR
  - Read data from PORTD SFR

- output the hex value 0x26 to Port C

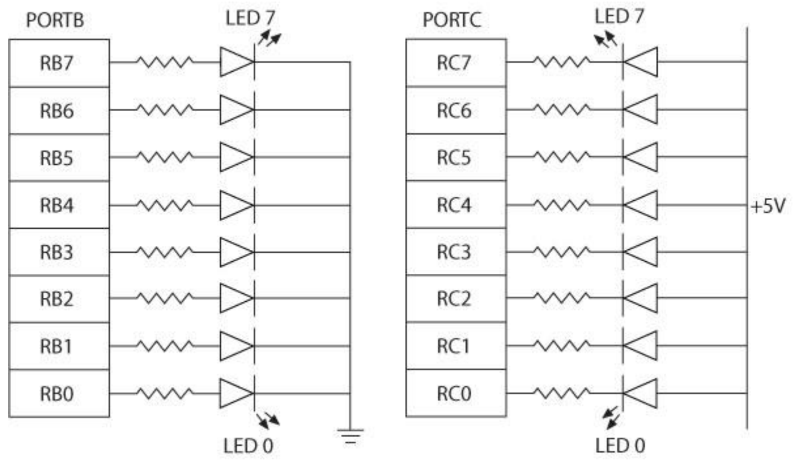
```
clrf TRISC
```
set to 0
```
movlw 0x26
movwf PORTC
```

- read the current value of Port D into WREG

```
setf TRISD
```
set to 1
```
movf PORTD, W
```

Interfacing with LED

- two ways of connecting LEDs to I/O port:
  - Common Cathode: LED cathodes are grounded and logic 1 from the I/O port pin turns on the LED
  - Common Anode: LED anodes are connected to the power supply and logic 0 from the I/O port pin turns on the LED



Common Cathode
Active high

Common Anode
Active low

Example

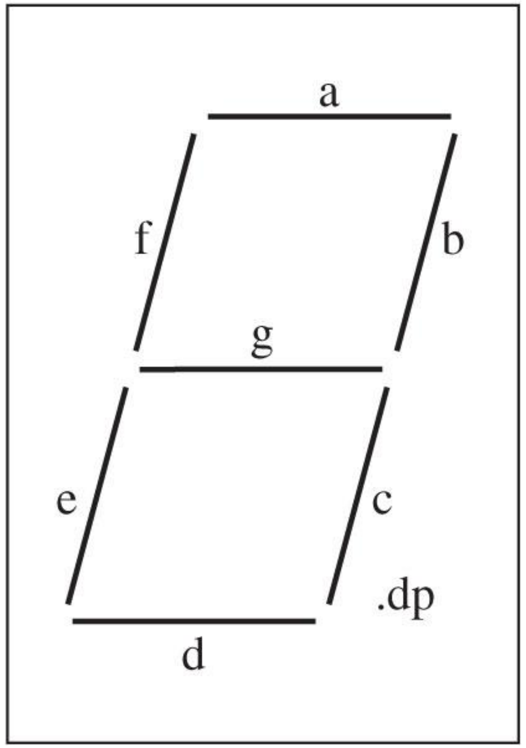
Connect PORTC to 8 LEDs. Turn on alternate LEDs and toggle PORT C forever with some time delay.

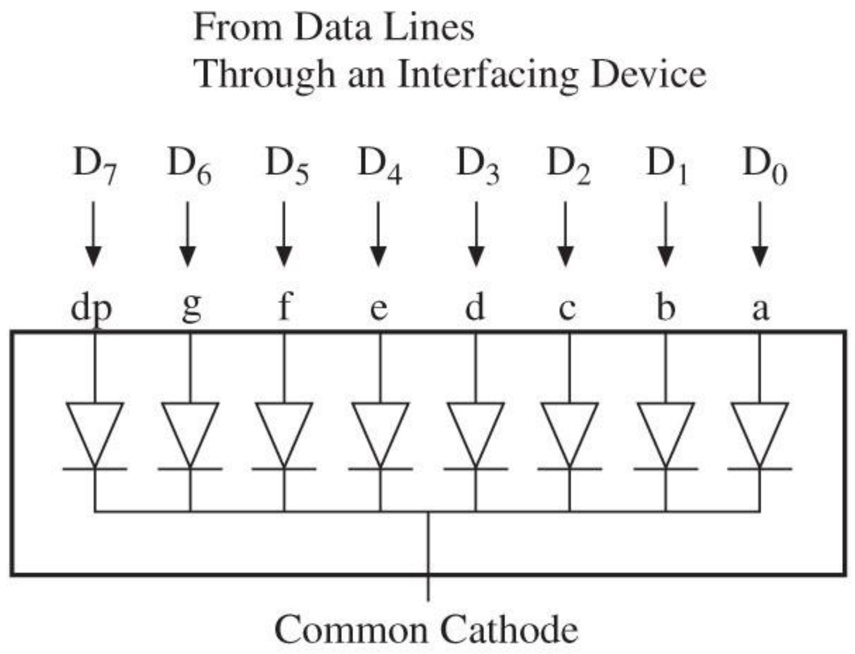
```
        movlw   00          ;Load W register with 0
        movwf   TRISC       ;Set up PORTC as output
Loop:   movlw   0x55        ;Byte 55H to turn on alternate LEDs
        movwf   PORTC       ;Turn on LEDs
        call    Delay       ;Delay some time
        comf    PORTC,F     ;Toggle PORTC
        call    Delay       ;Delay some time
        goto    Loop        ;Repeat forever

Delay:                      ;Delay function
        ….
        return
```

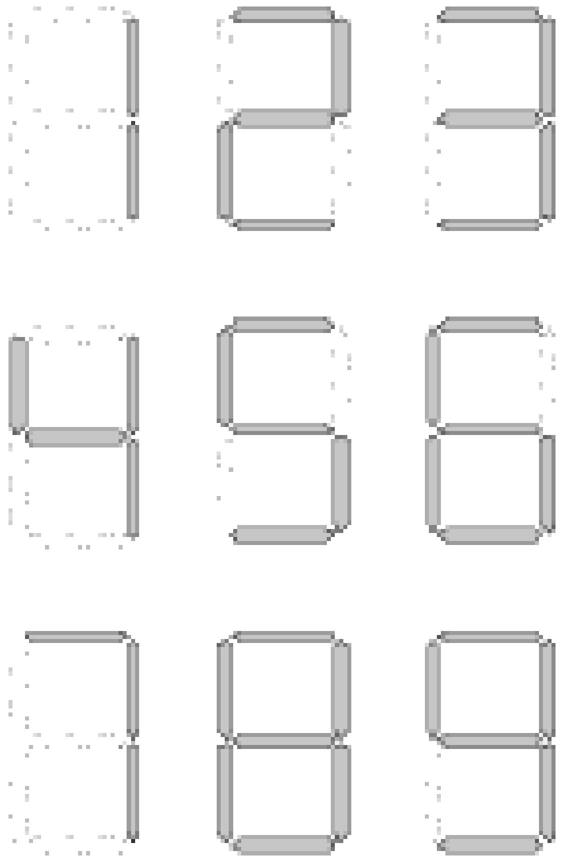
Interfacing with 7-segment LED

- often used to display digits (e.g. BCD numbers 0 through 9) and a few alphabets
- a group of eight LEDs physically mounted in the shape of the number eight plus a decimal point
- each LED is called a segment and labeled as ‘a’ through ‘g’.





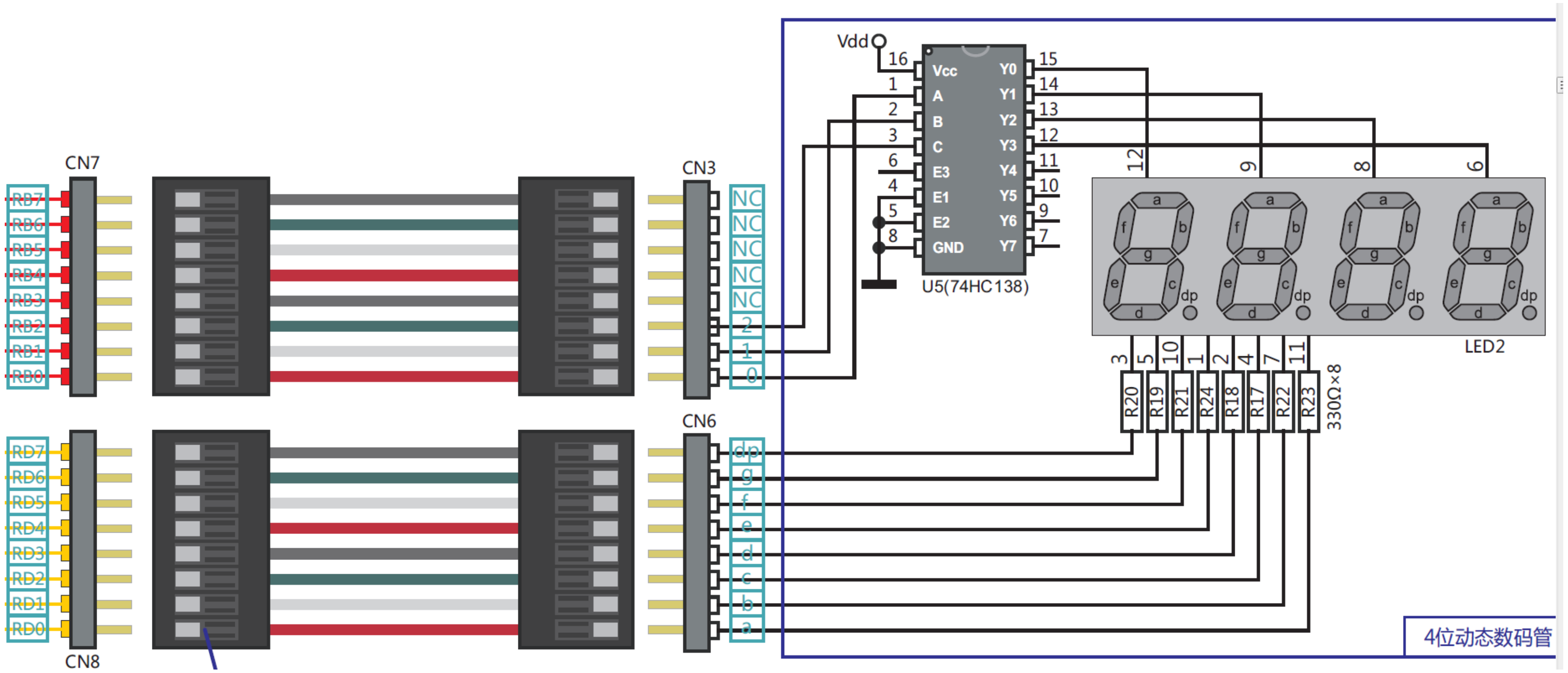
- in a common cathode 7-segment LED, all cathodes are connected together to ground and the anodes are connected to data lines
- logic 1 turns on a segment
- e.g. to display digit 1, all segments except b and c should be off
- byte 00000110 = 06H will display digit 1



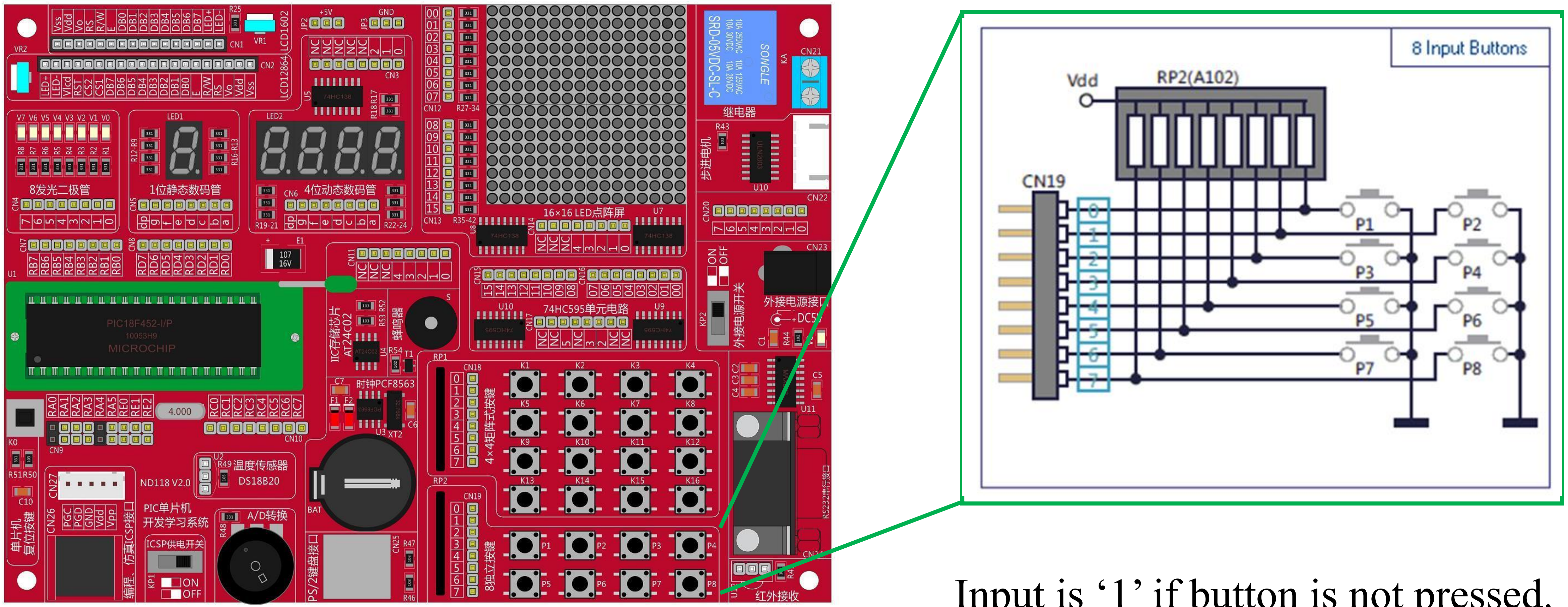
Digit Shown	Illuminated Segment (1 = illumination) a	b	c	d	e	f	g
0	1	1	1	1	1	1	0
1	0	1	1	0	0	0	0
2	1	1	0	1	1	0	1
3	1	1	1	1	0	0	1
4	0	1	1	0	0	1	1
5	1	0	1	1	0	1	1
6	1	0	1	1	1	1	1
7	1	1	1	0	0	0	0
8	1	1	1	1	1	1	1
9	1	1	1	1	0	1	1

Interfacing with 4-digit 7-segment LED

- decoder selects the position where digit pattern is displayed
- use time multiplexing (*short delay time*) if we need to display all four digits



8-button input



Input is ‘1’ if button is not pressed.
Input is ‘0’ if button is pressed.

Scan 8-button input

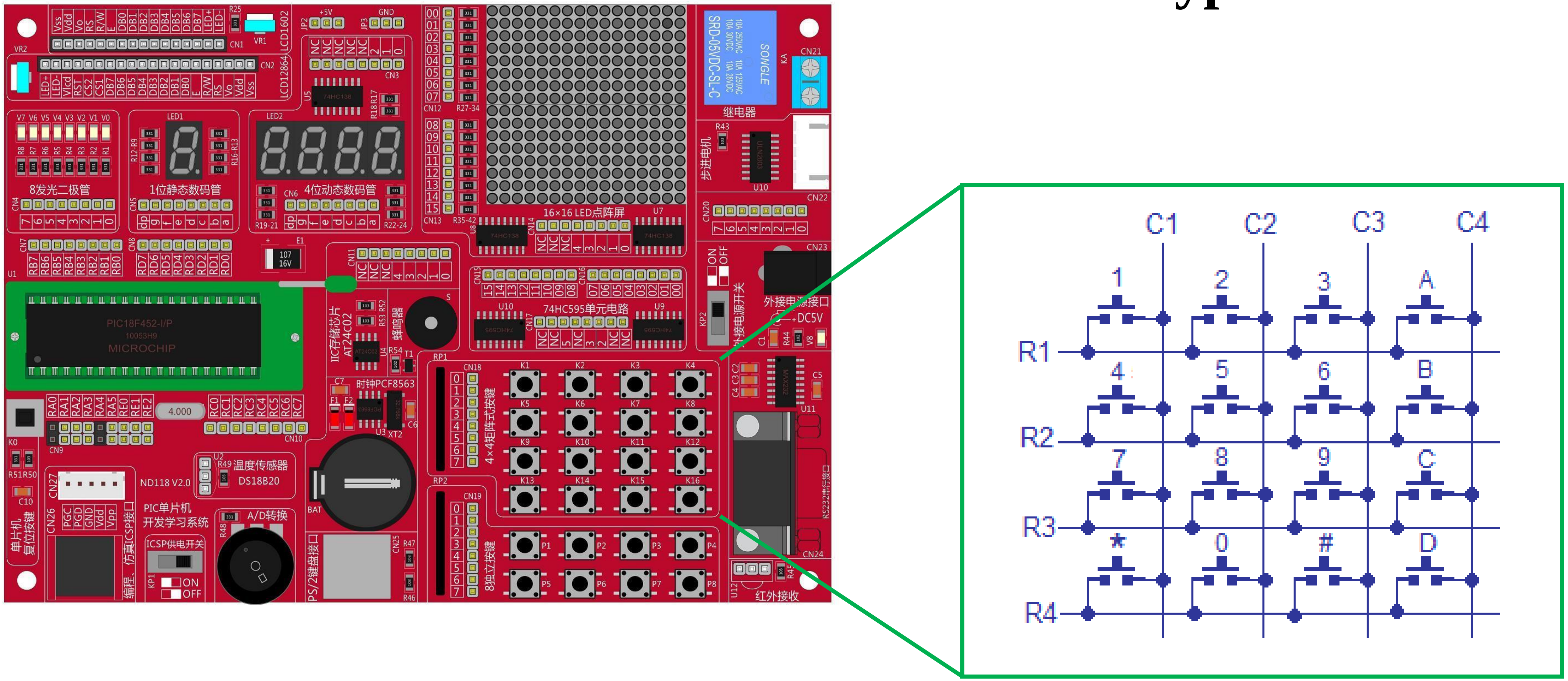
Scan buttons P1 to P8 sequentially.

If ‘0’ is detected, return the corresponding button number.

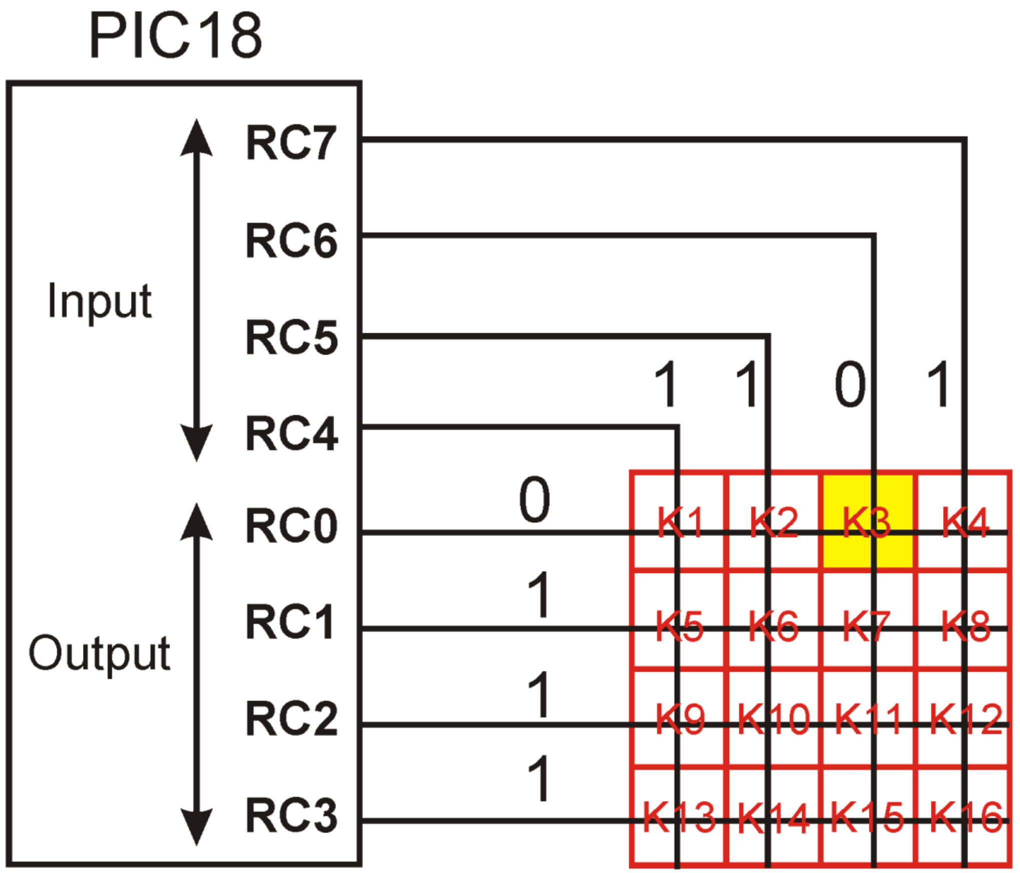
If ‘0’ is not detected after scanning all buttons, return ‘9’.

If P1 is pressed, 0xFE will return when (movf PORTX, w)

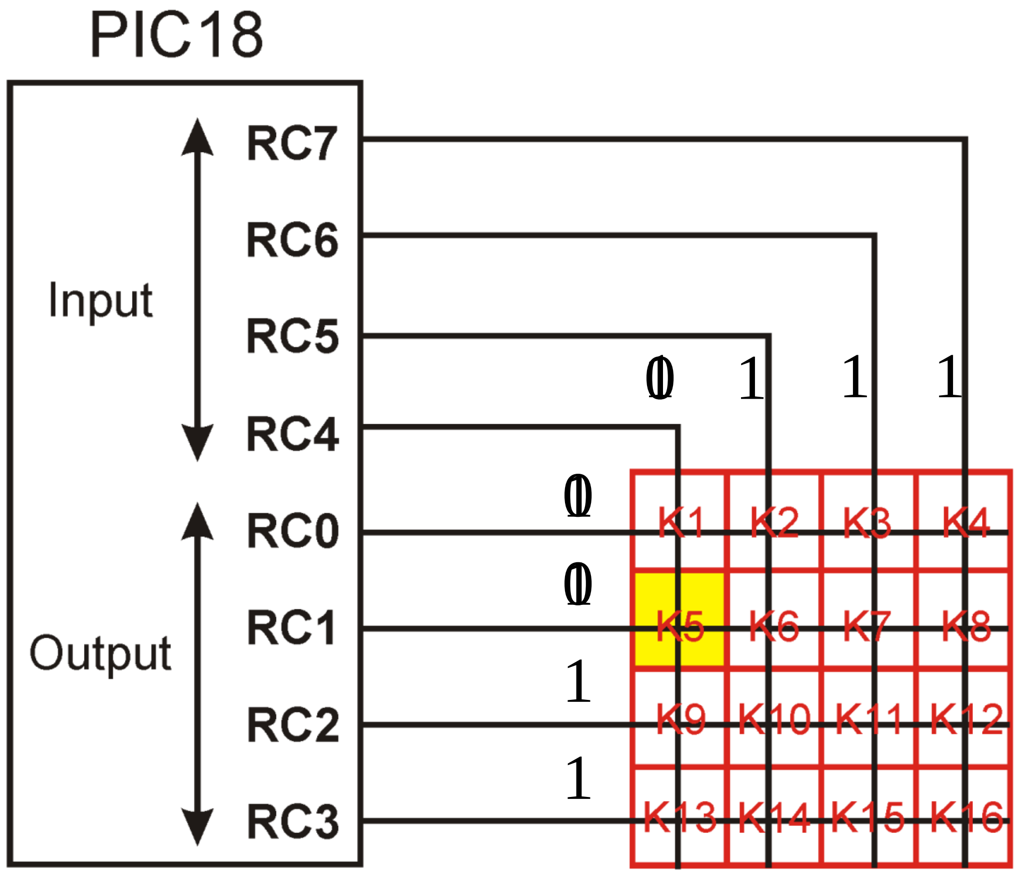
4 x 4 Keypad



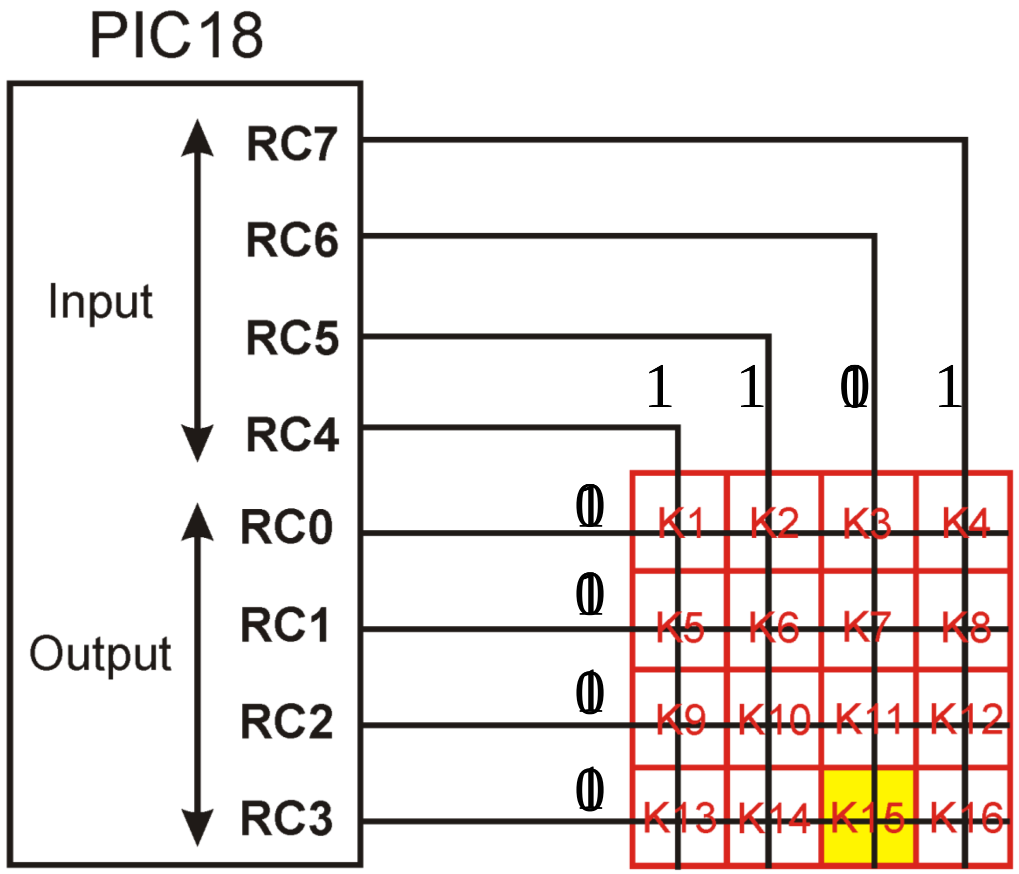
Interfacing with 4 x 4 Keypad



When K5 is pressed



When K15 is pressed



Scan 4 x 4 keypad

Scan keys K1 to K16 sequentially.

If key is pressed, return the corresponding key number.

If key is not pressed after scanning all keys, return ‘0’.

- Port A
  - A6 and A7 are not available if we use crystal oscillator
  - alternate function – ADC
  - we do not use Port A for simple I/O functions

- Port B
  - alternate function – interrupt

- Port C
  - alternate function – serial communication

- Port E
  - 3 additional analog inputs

- read-after-write I/O operation

```
        clrf    TRISB       output
        setf    TRISC       input
Loop:   movf    PORTC,W
        nop
        movwf   PORTB
        bra     Loop
```

```
        clrf    TRISB
        setf    TRISC
Loop:   movff   PORTC,PORTB
        bra     Loop
```

- upon reset, all ports have value FFH on their TRIS register (*set to input*)

3.3.2 I/O bit manipulation programming

- need to access individual bits of the port instead of the entire 8 bits
- PIC18 provides instructions that alter individual bits without altering the rest of the bits in that port
- common bit-oriented instructions:

Instructions	**Function**
bsf fileReg, bit	**Bit Set fileReg**
bcf fileReg, bit	**Bit Clear fileReg**
btg fileReg, bit	**Bit Toggle fileReg**
btfsc fileReg, bit	**Bit test fileReg, skip if clear**
btfss fileReg, bit	**Bit test fileReg, skip if set**

A	B	C	D	E
RA0	RB0	RC0	RD0	RE0
RA1	RB1	RC1	RD1	RE1
RA2	RB2	RC2	RD2	RE2
RA3	RB3	RC3	RD3	
RA4	RB4	RC4	RD4	
RA5	RB5	RC5	RD5	
	RB6	RC6	RD6	
	RB7	RC7	RD7	

bsf fileReg, bit

- set High a single bit of a given fileReg

- bit is the desired bit number (0 to 7)

  e.g. `bsf PortB, 5` sets bit 5 of Port B to be 1

  direction of portb -> output

Example

Connect 8 LEDs to Port C. Write a program to turn on each LED one by one with time delay.

```
        clrf    TRISC       ;make PORTC an output port
Loop:   bsf     PORTC,0     ;turn on RC0
        call    Delay       ;delay some time
        bsf     PORTC,1     ;turn on RC1
        call    Delay       ;delay some time
        bsf     PORTC,2     ;turn on RC2
        call    Delay       ;delay some time
        bsf     PORTC,3     ;turn on RC3
        call    Delay       ;delay some time
        …
        goto    Loop        ;repeat forever

Delay:                      ;delay function
        ….
        return
```

bcf fileReg, bit

- clear a single bit of a given fileReg

  e.g. `bcf PortB, 5` sets bit 5 of Port B to be 0

Example

Reverse bit 2 of Port C continuously.

one bit only

```
        bcf     TRISC, 2        ;make RC2 an output pin
Loop:   bsf     PORTC,2         ;set RC2 high
        call    Delay           ;delay some time
        bcf     PORTC,2         ;set RC2 low
        call    Delay           ;delay some time
        goto    Loop            ;repeat forever


Delay:                          ;delay function
        ....
        return
```

btg fileReg, bit

- toggle a single bit of a given fileReg

  e.g. `btg PortB, 5` toggles bit 5 of Port B

  (i.e., sets it to 1 if the current value is 0 & vice versa)

Example

Reverse bit 2 of Port C continuously.

```
        bcf     TRISC, 2        ;make RC2 an output pin
Loop:   btg     PORTC,2         ;toggle RC2
        call    Delay           ;delay some time
        goto    Loop            ;repeat forever

Delay:                          ;delay function
        ....
        return
```

Checking the state of an input port pin

- btfsc (bit test fileReg, skip if clear) and btfss (bit test fileReg, skip if set)
- they are used to make branching decision based on the status of a given bit (*conditional skipping*)
- e.g. btfsc PORTD, 2 skips the next instruction if bit 2 of Port D equals 0
- e.g. btfss PORTD, 2 skips the next instruction if bit 2 of Port D equals 1

Example

Write a program to perform the following:

(a) keep monitoring RB2 until it becomes high

(b) when RB2 becomes high, write value 45H to Port C

```
        bsf     TRISB, 2        ; set RB2 as input
        clrf    TRISC           ; set Port C as output
        movlw   0x45            ; WREG = 45H
Again:  btfss   PORTB, 2        ; test RB2 for high
        bra     Again     = 1   ; keep checking if low
        movwf   PORTC           ; write 45H to Port C
```

Example

Write a program to check RB2.
If RB2 = 0, send the letter ‘N’ to Port D.
If RB2 = 1, send the letter ‘Y’ to Port D.

```
          bsf       TRISB, 2     ; set RB2 as input
          clrf      TRISD        ; set Port D as output
Again:    btfss     PORTB, 2     ; test RB2
          bra       Over         ; RB2 = 0
                             = 1
          movlw     A'Y'
          movwf     PORTD        ; write 'Y' to Port D
          bra       Again
Over:     movlw     A'N'
          movwf     PORTD        ; write 'N' to Port D
          bra       Again
```

Example

Rewrite the previous program using btfsc.

```
         bsf       TRISB, 2    ; set RB2 as input
         clrf      TRISD       ; set Port D as output
Again:   btfsc     PORTB, 2    ; test RB2
         bra       Over        ; RB2 = 1
         movlw     A'N'
         movwf     PORTD       ; write 'N' to Port D
         bra       Again
Over:    movlw     A'Y'
         movwf     PORTD       ; write 'Y' to Port D
         bra       Again
```

Summary

- programming entire I/O ports
- programming individual I/O port pins

3.4 arithmetic and logic instructions

- arithmetic instructions
- signed number concepts
- logic instructions
- BCD and ASCII conversion

3.4.1 arithmetic instructions

- addition, subtraction, multiplication, division
- unsigned numbers – all bits are used to represent data
- 8-bit data 00 to FFH (0 to 255 decimal)

Addition

```
ADDLW K                     ;WREG = WREG + K
```

could change flag bits of status register

```
                  e.g. result
ADDWF fileReg,D,A           ;result = WREG + fileReg
```

addition of individual bytes

must involve WREG because memory-to-memory arithmetic operations are not allowed

```
ADDWFC fileReg,D,A          ;result = WREG + fileReg + carry
```

addition of 16-bit numbers – multi-byte addition

Example:

```
MOVLW  0xF5
ADDLW  0x0B
WREG  =  00
C=1  Z=1  DC=1
```

100

5+B=10

Example:

Assume that RAM locations 40-43 have the following hex values.

```
40=(7D)   41=(EB)   42=(C5)   43=(5B)
```

Write a program to find the sum of the values. Location 6 contains the low byte and location 7 contains the high byte of the sum.

Solution:

```
L_BYTE      EQU 0x6          ; low byte of sum
H_BYTE      EQU 0x7          ; high byte of sum
            MOVLW 0
            MOVWF H_BYTE
            ADDWF 0x40, W    ; WREG = 7D, C = 0   add ram1
            BNC   N_1
            INCF  H_BYTE, F  ; H_BYTE = 0
N_1         ADDWF 0x41, W    ; WREG = 68, C = 1   add ram2
            BNC   N_2
            INCF  H_BYTE, F  ; H_BYTE = 1
N_2         ADDWF 0x42, W    ; WREG = 2D, C = 1   add ram3
            BNC   N_3
            INCF  H_BYTE, F  ; H_BYTE = 2
N_3         ADDWF 0x43, W    ; WREG = 88, C = 0   add ram4
            BNC   N_4
            INCF  H_BYTE, F  ; H_BYTE = 2
N_4         MOVWF L_BYTE     ; L_BYTE = 88
```

Example

Write a program to add two 16-bit numbers. The numbers are

$\overset{03\ 02}{\text{3CE7}}$H and $\overset{01\ 00}{\text{3B8D}}$H

Assume location 00 contains 8D, location 01 contains 3B, location 02 contains E7, location 03 contains 3C.

```
MOVF    0x00,W
ADDWF       0x02, F   RESULT = 00 + 02
MOVF        0x01,W
ADDWFC    0x03, F   RESULT = 01 + 03 + C
```

Location 2 contains low byte of the sum. Location 3 contains high byte of the sum.

BCD Number System

- Binary Coded Decimal : binary representation of decimal digits 0 - 9
- two terms for BCD number :
  - Unpacked BCD
  - Packed BCD

Digit	*BCD*
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

- unpacked BCD - lower 4 bits represent a BCD number, upper 4 bits are zero
  - 0000 0101 is unpacked BCD for 5
  - 0000 1001 is unpacked BCD for 9
- packed BCD - a byte has two BCD numbers
  - 0101 1001 is packed BCD for 59
  - it is twice as efficient in storing data
- when we get two BCD numbers, we want to add them directly in form of packed BCD
- however, ADDLW, ADDWF, ADDWFC are just for binary number!

Addition of Packed BCD Numbers

- to calculate $17_{10}+18_{10}=35_{10}$

```
MOVLW 17H
ADDLW 18H
```

- if the resulting low order byte > 9, i.e. there is a carry from bit 3 to bit 4, the programmer must add 6 to the low digital
- then we get the correct packed BCD sum (35H)
- same adjustment is needed for high order byte

$$\begin{array}{rcc} & 1 & 7 \\ + & 1 & 8 \\ \hline & 2 & F \end{array}$$

$$\begin{array}{lrcc} & & 2 & F \\ \longrightarrow & + & 0 & 6 \\ \cline{2-4} & & 3 & 5 \end{array}$$

DAW (Decimal Adjust WREG)

- designed to correct BCD addition problem
- only work with an operand in WREG
- add 6 to the lower 4 bits if DC = 1
- add 6 to the upper 4 bits if C = 1

```
MOVLW   0x47
ADDLW   0x55
DAW
```

```
      4 7
    + 5 5
    -----
      9 C
      A 2    adjust lower 4 bits
  = 1 0 2    adjust upper 4 bits
```

Example

Assume that RAM locations 40-43 have the following packed BCD values.

40=(34) 41=(21) 42=(99) 43=(88)

Write a program to find the sum of the values. The result must be in packed BCD.

Solution:

```
L_BYTE          EQU     0x6
H_BYTE          EQU     0x7
                MOVLW           0
                MOVWF   H_BYTE
                ADDWF           0x40, W
                DAW
                BNC             N_1
                INCF            H_BYTE, F
N_1             ADDWF           0x41, W
                DAW
                BNC             N_2
                INCF            H_BYTE, F
N_2             ADDWF           0x42, W
                DAW
                BNC             N_3
                INCF            H_BYTE, F
N_3             ADDWF           0x43, W
                DAW
                BNC             N_4
                INCF            H_BYTE, F
N_4             MOVWF           L_BYTE
```

Subtraction

- separate subtracter circuitry is cumbersome
- PIC uses adder circuitry to perform subtraction

  minuend – subtrahend = minuend + (-subtrahend)

- PIC performs the 2’s complement of the subtrahend
- adds it to minuend
- uses C flag as borrow, borrow = $\overline{C}$
- result is positive if N = 0 or C = 1
- result is negative if N = 1 or C = 0 (*the result is left in 2’s complement*)

- there are four subtraction instructions:

```
SUBLW k              ; WREG = k – WREG
SUBWF f, d, a        ; destination = fileReg – WREG
SUBWFB f, d, a       ; destination = fileReg – WREG – borrow
SUBFWB f, d, a       ; destination = WREG – fileReg – borrow
```

Example

```
Examine
      MOVLW 0x23
      SUBLW 0x3F



Solution: 3F - 23
K=3F=0011 1111
W=23=0010 0011→2's complement→ 1101 1101
0011 1111 + 1101 1101 = 1 0001 1100
C=1 Bit7=N=0
```

Example

Write a program to subtract 4C-6E.

```
MYREG EQU 0x20
      MOVLW 0x4C
      MOVWF MYREG
      MOVLW 0x6E
      SUBWF MYREG, W       ; 4C-6E=DE, N=1
```

0100 1100 - 0110 1110 = 0100 1100 + 1001 0010 = 1101 1110

= 0

Note that we can check N flag (C flag) to determine if the result is negative or not.

Example

Write a program to subtract two 16-bit numbers (2762H-1296H).

76

Assume location 6 = 62, location 7 = 27.

```
    MOVLW  0x96
    SUBWF  0x6, F ;F=F-W=62-96=CCH, C=0,N=1
    MOVLW  0x12
    SUBWFB  0x7,F ;F=F-W-b=27-12-1=14H
```

= 0

Note that we can check N flag (C flag) to determine if the result is negative or not.

+ve 14CC

Multiplication

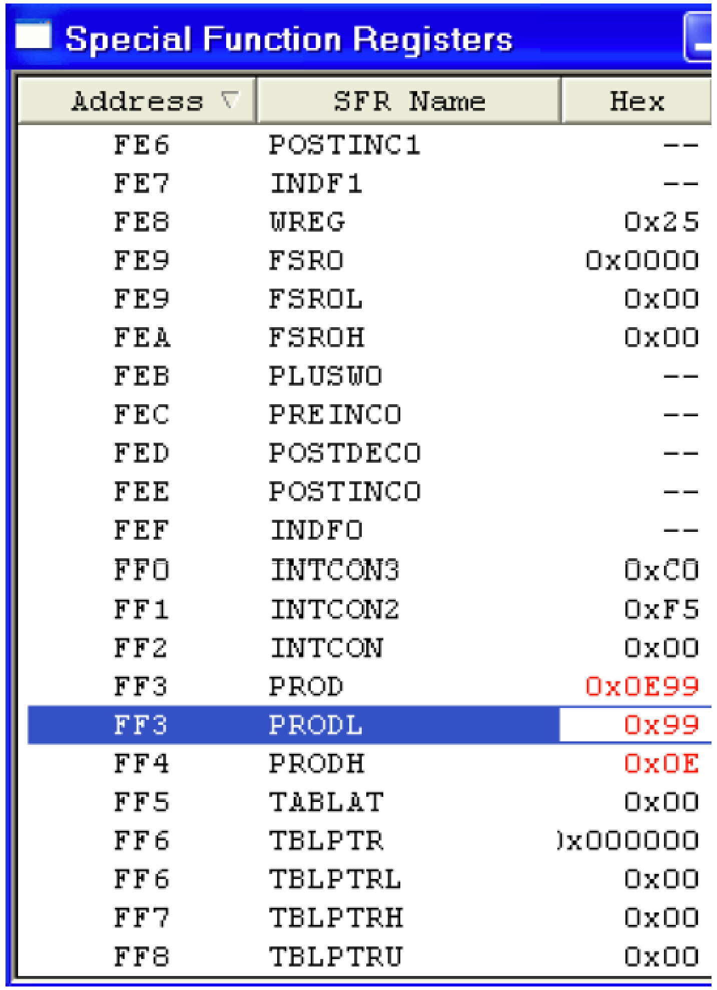
- PIC supports byte-by-byte multiplication
- one operand must be in WREG
- second operand must be a literal
- after multiplication, the result is stored in PRODH (high byte) and PRODL (low byte)

MULLW k ; PROD = WREG x k

Example

```
MOVLW  0x25
MULLW  0x65
```

PRODH = 0EH, PRODL = 99H



Special Function Registers

Address	SFR Name	Hex
FE6	POSTINC1	--
FE7	INDF1	--
FE8	WREG	0x25
FE9	FSR0	0x0000
FE9	FSR0L	0x00
FEA	FSR0H	0x00
FEB	PLUSW0	--
FEC	PREINC0	--
FED	POSTDEC0	--
FEE	POSTINC0	--
FEF	INDF0	--
FF0	INTCON3	0xC0
FF1	INTCON2	0xF5
FF2	INTCON	0x00
FF3	PROD	0x0E99
FF3	PRODL	0x99
FF4	PRODH	0x0E
FF5	TABLAT	0x00
FF6	TBLPTR	)x000000
FF6	TBLPTRL	0x00
FF7	TBLPTRH	0x00
FF8	TBLPTRU	0x00

Division

- no single instruction for the division of byte/byte numbers
- need to write a program

i. numerator is placed in a fileReg
ii. denominator is subtracted repeatedly
iii. quotient is the number of times we subtracted
iv. remainder in fileReg

Example

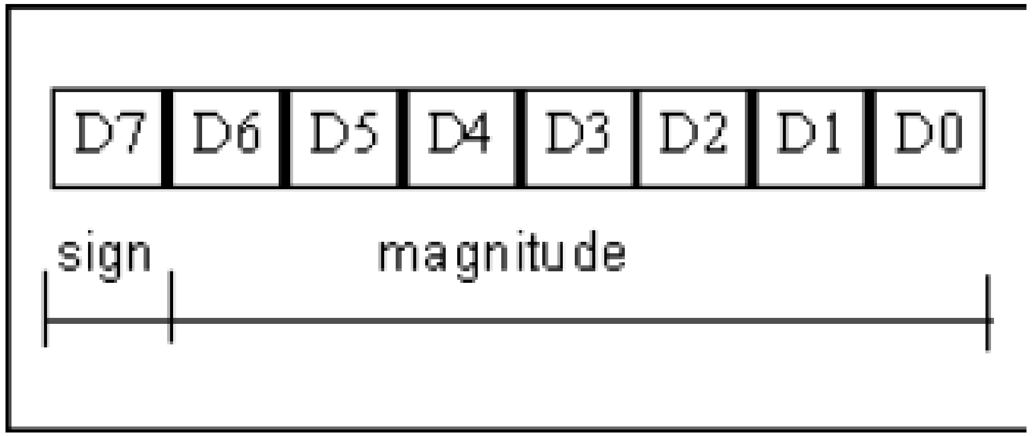
Convert the hexadecimal number FDH stored in location 0x15H, into decimal. Save the digits in locations 0x22,0x23,0x24.

```
NUME        EQU    0x15          ;location for numerator
QU          EQU    0x20          ;location for quotient
RMND_L      EQU    0X22          ;low
RMND_M      EQU    0x23          ;middle
RMND_H      EQU    0x24          ;high
MYNUM       EQU    0xFD
MYDEN       EQU    D’10’
                   ORG     0H
                   MOVLW   MYNUM
                   MOVWF   NUME      ; = FD
                   MOVLW   MYDEN
                   CLRF    QU
D_1                INCF    QU,F
                   SUBWF   NUME,F
                   BC      D_1
```

```
        ADDWF   NUME,F              ;correction for sub num to -ve value
        DECF    QU, F
        MOVFF   NUME, RMND_L        ;save first digit
        MOVFF   QU, NUME
        CLRF    QU
D_2     INCF    QU,F
        SUBWF   NUME,F
        BC      D_2
        ADDWF   NUME,F              ;correction for sub num to -ve value
        DECF    QU, F
        MOVFF   NUME, RMND_M        ;save second digit
        MOVFF   QU, RMND_H          ;save third digit
```

3.4.2 signed number concepts

- MSB is set aside for the sign (0 is +ve, 1 is -ve)
- the rest, 7 bits, are used for the magnitude
- to convert any 7-bit positive number to negative, use the 2's complement
- you have 128 negative numbers and 127 positive numbers



Example

Show how the PIC18 would represent –128.

Solution:

Observe the following steps.

```
1.  1000 0000  128 in 8-bit binary 80H
2.  0111 1111  invert each bit
3.  1000 0000  add 1 (which becomes 80 in hex)
```

Therefore -128 = 80H, the signed number representation in 2’s complement for -128.

Bit7 = N = 1

Overflow problem

When two signed numbers are added and the number of bits required to represent the sum exceeds the number of bits in the two numbers, an overflow result is indicated by an incorrect sign bit.

An overflow can occur only when both numbers are positive or negative, e.g. 125 + 58

```
            01111101
+           00111010
--------------------------
            10110111
```

From the binary calculation, +ve + +ve = -ve => impossible => Overflow.

Check sign bit

- the CPU understands only 0s and 1s and ignores the human convention of positive and negative numbers
- the overflow flag (OV) is designed to indicate an overflow of the operations for the signed numbers
- 8-bit signed number ranges **–128 to 127** in decimal
- When is an overflow?
    - If the result of an operation on signed numbers is too large for the 8-bit register, an overflow has occurred and the programmer must be notified
- OV flag is set to 1
    - If there is carry from bit 6 to bit 7 but no carry out of bit 7 (C = 0)
    - If there is carry out of bit 7 (C = 1) but no carry from bit 6 to bit 7

Example

If there is carry from bit 6 to bit 7 but no carry out of bit 7 (C = 0)

Examine the following code and analyze the result.

```
        MOVLW  +D'96'
        ADDLW  +D'70'
```

Solution:

```
     96   0110 0000
+    70   0100 0110
  -----   ---------
+ 166     1010 0110  (N=1, C=0, OV=1)
```

The signed value in WREG=A6H=-90 is wrong. **Programmers must check it (check OV) by themselves**.

The above checking is done by computer. Actually, the CPU can check the sign bit. See if the sign bit agrees with our expectation or not.

Example

If there is carry out of bit 7 (C = 1) but no carry from bit 6 to bit 7

Observe the following, noting the role of the OV flag.

```
MOVLW -D'128'
ADDLW -D'2'
```

Solution:

```
  -128      1000 0000
+   -2      1111 1110
--------    ---------
- 130       0111 1110   (N=0, C=1, OV=1)
```

The sign bit is wrong so the hardware sets OV=1.

Example

Examine the following, noting the role of OV.

```
    MOVLW +D'7'
    ADDLW +D'18'
```

Solution:

```
       7     0000 0111
  +   18     0001 0010
  --------   ---------
      25     0001 1001  (N=0,  C=0,  OV=0)
```

+ve + +ve , the resulting sign bit is 0 (+ve), the sign bit is correct.

The hardware resets OV=0.

3.4.3 logic instructions

Widely used instructions (*affect only Z and N flags*):

`ANDLW k` k AND w

`ANDWF FileReg, d` w AND FileReg

often used to mask certain bits of an operand (set to 0)

`IORLW k`

`IORWF FileReg, d`

often used to set certain bits of an operand to 1

`XORLW k`

`XORWF FileReg, d`

often used to check if two registers have the same value, or toggle the bits of an operand

1 xor 1 -> 0
0 xor 0 -> 0
same -> 0

xor all 1 -> 1's com

COMF FileReg,d

takes the 1's complement of a file register

affect only Z and N flags

NEGF FileReg

takes the 2's complement of a file register

affect all flags

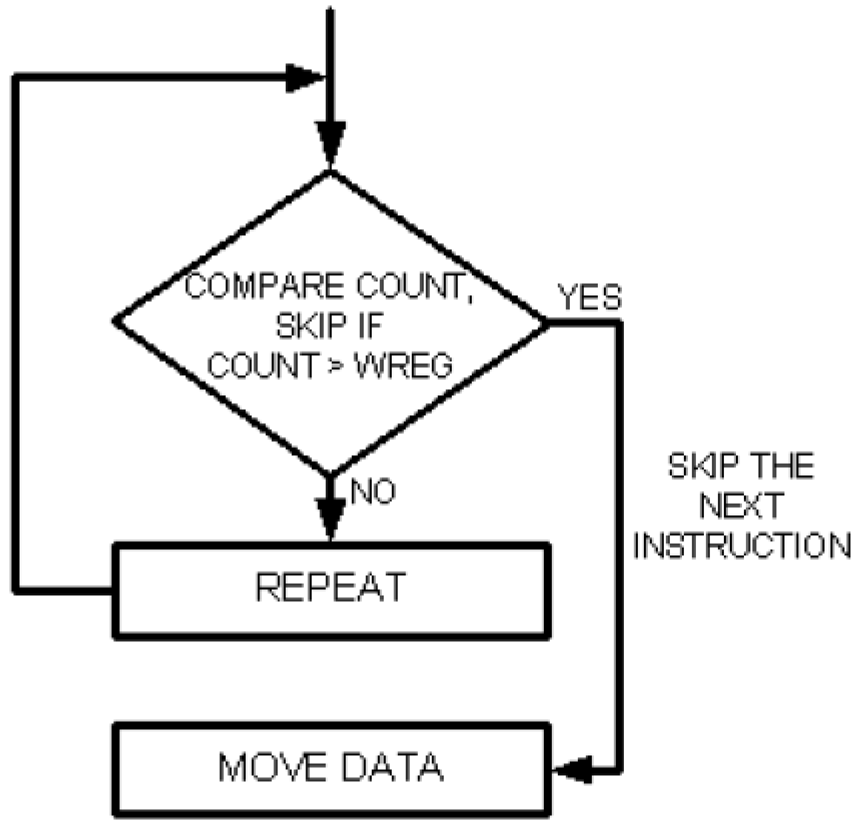
Example

```
    MYREG EQU 0x10
    MOVLW 0x85
    MOVWF MYREG
    NEGF MYREG
```

- these instructions take 1 (*falling through*) or 2 (*skip*) machine cycles
- compare fileReg with WREG
- subtraction is carried out, but operands are not changed

`CPFSGT fileReg`	**compare fileReg with WREG, skip if greater than**	fileReg > WREG
`CPFSEQ fileReg`	**compare fileReg with WREG, skip if equal**	fileReg = WREG
`CPFSLT fileReg`	**compare fileReg with WREG, skip if less than**	fileReg < WREG

Flowchart for CPFSGT


COMPARE COUNT, SKIP IF COUNT > WREG
YES
NO
SKIP THE NEXT INSTRUCTION
REPEAT
MOVE DATA

INSTRUCTIONS

```
AGAIN      CPFSGT COUNT

           GOTO AGAIN

           MOVWF PORTB
```

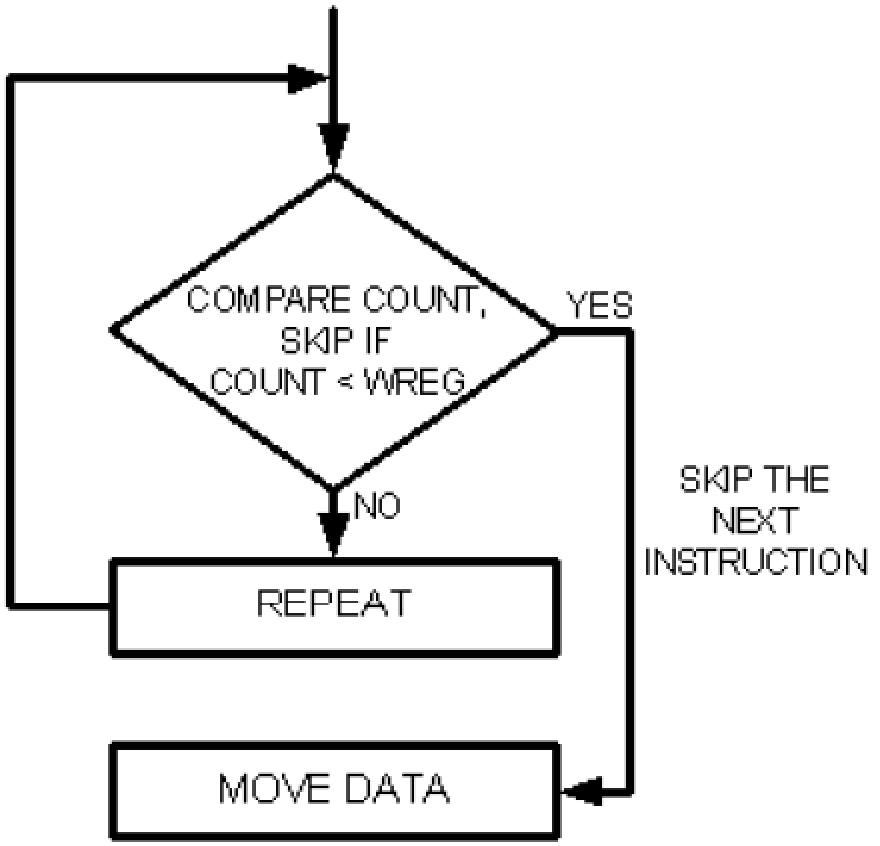
Example

Write a program to find the greater of the two values 27 and 54, and place it in fileReg 0x20.

```
Val_1       EQU   d'27'
Val_2       EQU   d'54'
Greater     EQU   0x20

movlw       Val_1
movwf       Greater
movlw       Val_2
cpfsgt      Greater
movwf       Greater   no skip
```

Flowchart for CPFSLT


COMPARE COUNT, SKIP IF COUNT < WREG
YES
NO
SKIP THE NEXT INSTRUCTION
REPEAT
MOVE DATA

INSTRUCTIONS

```
AGAIN      CPFSLT COUNT

           GOTO AGAIN

           MOVWF PORTB
```

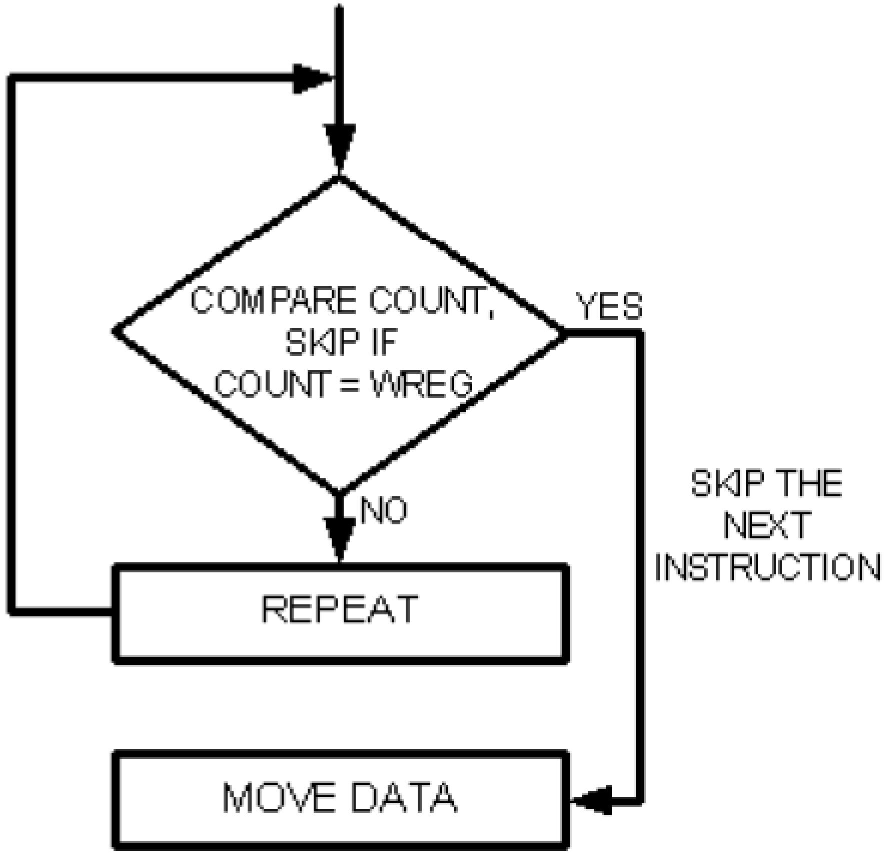
Example

Write a program to find the smaller of the two values 27 and 54, and place it in fileReg 0x20.

```
Val_1       EQU   d’27’
Val_2       EQU   d’54’
Smaller     EQU   0x20

movlw       Val_2
movwf       Smaller
movlw       Val_1
cpfslt      Smaller
movwf       Smaller
```

Flowchart for CPFSEQ


COMPARE COUNT, SKIP IF COUNT = WREG
YES
NO
SKIP THE NEXT INSTRUCTION
REPEAT
MOVE DATA

INSTRUCTIONS

```
AGAIN      CPFSEQ COUNT

           GOTO AGAIN

           MOVWF PORTB
```

Example

Write code to determine if data on PORTB contains the value 99H. If so, write letter ‘y’ to PORTC. Otherwise, write letter ‘N’ to PORTC.

```
        CLRF      TRISC
        MOVLW     A'N'
        MOVWF     PORTC
        SETF      TRISB
        MOVLW     0x99
        CPFSEQ    PORTB
        BRA       OVER
  yes
        MOVLW     A'y'
        MOVWF     PORTC
OVER:   ...  ...  ...
```

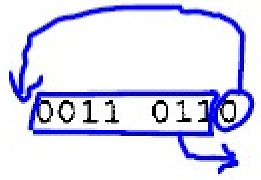
Rotate instruction

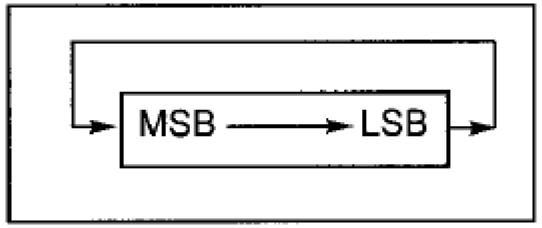
- rotate fileReg **Right** or **Left** (no Carry)

  RRNCF fileReg, d

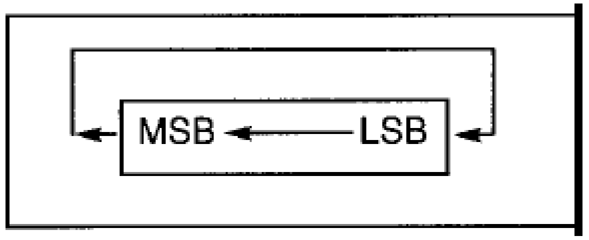
  RLNCF fileReg, d
- affect the N and Z flag bits

rotate right





rotate left



```
MREG EQU 0x20
        MOVLW  0x36               ;WREG = 0011 0110
        MOVWF  MYREG
        RRNCF  MYREG,F            ;MYREG = 0001 1011
        RRNCF  MYREG,F            ;MYREG = 1000 1101
        RRNCF  MYREG,F            ;MYREG = 1100 0110
        RRNCF  MYREG,F            ;MYREG = 0110 0011
```

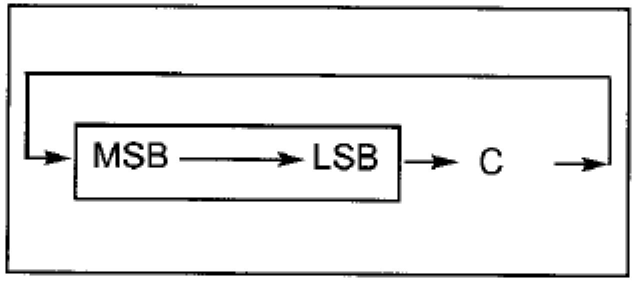
```
MREG EQU 0x20
        MOVLW  0x72               ;WREG = 0111 0010
        MOVWF  MYREG
        RLNCF  MYREG,F            ;MYREG = 1110 0100
        RLNCF  MYREG,F            ;MYREG = 1100 1001
```

- rotate fileReg **R**ight or **L**eft through Carry flag

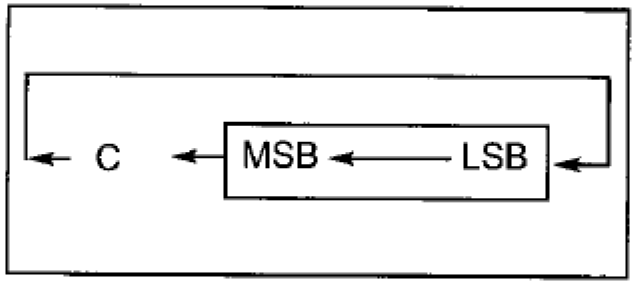
  RRCF fileReg, d

  RLCF fileReg, d
- affect the C, N and Z flag bits

rotate right



rotate left



```
MREG  EQU  0x20
      BCF     STATUS,C     ;make C = 0 (carry is D0
      MOVLW   0x26         ;WREG = 0010 0110
      MOVWF   MYREG
      RRCF    MYREG,F      ;MYREG = 0001 0011 C = 0
      RRCF    MYREG,F      ;MYREG = 0000 1001 C = 1
      RRCF    MYREG,F      ;MYREG = 1000 0100 C = 1
```

```
MREG  EQU  0x20
      BSF     STATUS,C     ;make C = 1 (carry is D0
      MOVLW   0x15         ;WREG = 0001 0101
      MOVWF   MYREG
      RLCF    MYREG,F      ;MYREG = 0010 1011 C = 0
      RLCF    MYREG,F      ;MYREG = 0101 0110 C = 0
      RLCF    MYREG,F      ;MYREG = 1010 1100 C = 0
      RLCF    MYREG,F      ;MYREG = 0101 1000 C = 1
```

Example

Write code to find the number of 1's in a given number 97H.

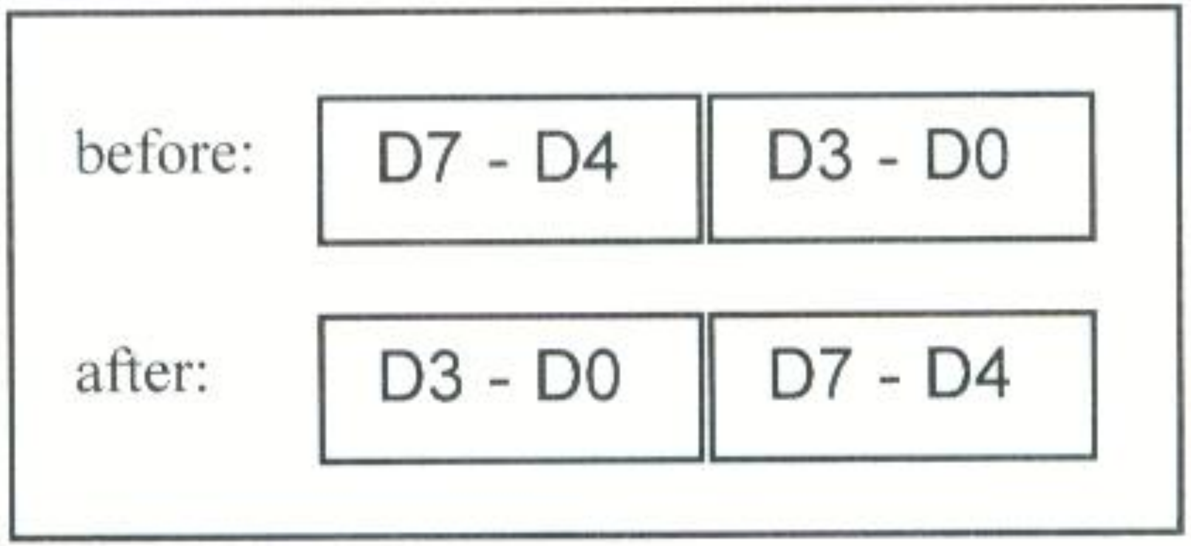
```
R1              EQU 0x20; fileReg for number of 1's
COUNT           EQU 0x21; fileReg for counter
VALREG          EQU 0x22; fileReg for the byte

                BCF     STATUS, C; C=0
                CLRF    R1
                MOVLW   0x8
                MOVWF   COUNT   ; load the counter
                MOVLW   0x97
                MOVWF   VALREG   ; load the byte
AGAIN           RLCF    VALREG, F
                BNC     NEXT
                INCF    R1
NEXT            DECF    COUNT, F
                BNZ     AGAIN
```

SWAP

```
SWAPF fileReg, d
```

swap the lower 4 bits and the higher 4 bits of fileReg



	7	6	5	4	3	2	1	0
A	1	1	1	1	0	0	0	0

	7	6	5	4	3	2	1	0
A	0	0	0	0	1	1	1	1

What method can you use if there is no SWAPF instruction?

do RRNCF/RLNCF 4 times

Example

Re-write the following program if there is no swapf instruction.

```
MyReg        EQU 0x20
             MOVLW 0x72
             MOVWF MyReg
             SWAPF MyReg,F
```

3.4.4 BCD and ASCII conversion

Key	ASCII (hex)	Binary	BCD (unpacked)
0	30	011 0000	0000 0000
1	31	011 0001	0000 0001
2	32	011 0010	0000 0010
3	33	011 0011	0000 0011
4	34	011 0100	0000 0100
5	35	011 0101	0000 0101
6	36	011 0110	0000 0110
7	37	011 0111	0000 0111
8	38	011 1000	0000 1000
9	39	011 1001	0000 1001

Packed BCD and ASCII Conversion

Packed BCD to ASCII

Packed BCD	**unpacked BCD**	**ASCII**
29H	02H & 09H	32H & 39H
0010 1001	0000 0010 & 0000 1001	0011 0010 & 0011 1001

Packed BCD to ASCII

key	**ASCII**	**unpacked BCD**	**packed BCD**
"4"	34H	0000 0100	0100 0111
"7"	37H	0000 0111	

Example

Assume that register WREG has packed BCD. Write a program to convert it to two ASCII numbers and place them in file register locations 6 and 7.

e.g. 10100101 and 11110000 -> 10100000

```
BCD_VAL     EQU 0x29
L_ASC       EQU 0x06
H_ASC       EQU 0x07

            MOVLW   BCD_VAL       ;load the BCD value
            ANDLW   0x0F          ;mask the upper 4 bits
            IORLW   0x30          ;make it an ASCII, W=39H
            MOVWF   L_ASC         ;save it
            MOVLW   BCD_VAL       ;load the BCD value
            ANDLW   0xF0          ;mask the lower 4 bits
            SWAPF   WREG, W       ;swap
            IORLW   0x30          ;make it an ASCII, W=32H
            MOVWF   H_ASC         ;save it
```

Example

Write a program to convert two digit ASCII numbers (47) to a packed BCD.

```
MYBCD       EQU 0x29

            MOVLW A'4'
            ANDLW 0x0F          ;mask the upper 4 bits
            MOVWF MYBCD
            SWAPF MYBCD, F
            MOVLW A'7'
            ANDLW 0x0F
            IORWF MYBCD, F
```

Summary

- unsigned number arithmetic operations
- signed number arithmetic operations
- logic, compare and rotation instructions
- BCD and ASCII conversion