INTRODUCTION TO THE PIC18 MICROCONTROLLER

PIC Microcontroller: An Introduction to Software & Hardware Interfacing

Han-Way Huang, Thomson Delmar Learning, 2005

Chung-Ping Young

楊中平

Networked Embedded Applications and Technologies Lab

Department of Computer Science and Information Engineering

National Cheng Kung University, TAIWAN

Components of an Assembly Program

- Assembler directives
- Assembly language instructions
- Comments

Elements of an Assembly Language Statement

- Label
- Mnemonics
- Operands
- Comment

Label Field

- Must start from column 1 and followed by a tab, a space, a colon (:), or the end of a line.

- Must start with an alphabetic character or underscore (_).

- May contain alphanumeric characters, underscores and question marks (?).

- May contain up to 32 characters and is case-sensitive by default.

wait	btfss	sum,7	; **wait** is a label
_again	decf	loop_cnt,F	; **_again** is a label

Mnemonic Field

- Can be an assembly instruction mnemonic or assembly directive
- Must begin in column two or greater
- Must be separated from the label by a colon, one or more spaces or tabs

```
        addlw  0x10        ; addlw is the mnemonic field
loop    incf   0x30,W,A    ; incf is a mnemonic
false   equ    0           ; equ is the mnemonic field
```

The Operand Field

- The operand (s) follows the instruction mnemonic.
- Provides the operands for an instruction or arguments for an assembler directive.
- Must be separated from the mnemonic field by one or more spaces or tabs.
- Multiple operands are separated by commas.

	movff	0x30,0x400	; "**0x30,0x400**" is the operand field
	decf	loop_cnt,F	; label **loop_cnt** is the operand
true	equ	1	; '1' is the argument for **equ**

Comment field

- Is optional
- A comment starts with a semicolon.
- All characters to the right of the semicolon are ignored by the assembler
- Comments provide documentation to the instruction or assembler directives
- A comment may explain the function of a single statement or the function of a group of instructions

```
too_high    decf    mean,F,A        ; prepare to search in the lower half
```

"too_high" is a label

"decf" is a mnemonic

"mean,F,A" is the operand field

"; prepare to search in the lower half" is a comment

Assembler Directives

- Control directives
- Data directives
- Listing directives
- Macro directives
- Object file directives

Control Directives

if <expr> ; directives for conditional assembly
else
endif

Example.

```
if version == 100
      movlw  D'10'
      movwf  io1,A
else
      movlw  D'26'
      movwf  io2,A
endif
```

end ; indicates the end of the program

[<label>] code [<ROM address>]

- Declares the beginning of a section of program code.
- If no label is specified, the section is named "**.code**".
- The starting address of the section is either included in the directive or assigned at link time if not specified in the directive.

```
reset   code    0x00
        goto    start
```

#define <name> [<string>] ; defines a text substitution string

```
#define     loop_cnt        30
#define     sum3(x,y,z)     (x + y + z)
#define     seed            103
```

#undefine <label> ; deletes a substitution string

#include "<include_file>" **(**or **#include <include_file>)**

#include "lcd_util.asm" ; include the **lcd_util.asm** file from current directory

#include <p18F8680.inc> ; include the file **p18F8680.inc** from the installation
; directory of **mplab**.

radix <default_radix>

- sets the default radix for data expression
- the default radix values are: hex, dec, or oct

radix dec ; set default radix to decimal

while <expr>

endw

- The lines between **while** and **endw** are assembled as long as **<expr>** is true.

Data Directives

db	**<expr>,…,<expr>**		; define 1 or multiple byte values
db	**“text_string”**		; define a string
dw	**<expr>,…,<expr>**		**;** define 1 or multiple word constants
dw	**“text_string”**		; define a string
dt	<expr>, …, <expr>		; generates a series of retlw instructions
<label>	**set**	**<expr>**	; assign a value (<expr>) to **label**
<label>	**equ**	**<expr>**	; defines a constant

Data Directives Examples

```
led_pat  db   0x30,0x80,0x6D,9x40,0x79,0x20,0x33,0x10,0x5B,0x08
msg1     db   “Please enter your choice (1/2):”,0
array    dw   0x1234,0x2300,0x40,0x33
msg2     dw   “The humidity is “,0
results  dt   1,2,3,4,5
sum_hi   set  0x01
sum_lo   set  0x00
TH       equ  200
TL       equ  30
```

What is a macro?

- A group of instructions that are grouped together and assigned a name
- One or multiple arguments can be input to a macro
- By entering the macro name, the same group of instructions can be duplicated in any place of the program.
- User program is made more readable by using macros
- User becomes more productive by saving the text entering time

Macro Directives

macro
endm
exitm

Macro Definition Examples

```
eeritual    macro                   ; macro name is eeritual
            movlw   0x55            ; instruction 1
            movwf   EECON2          ; instruction 2
            movlw   0xAA            ; instruction 3
            movwf   EECON2          ; instruction 4
            endm
```

Macro Call Example

```
            eeritual                ; this macro call causes the
                                    ; assembler to insert
                                    ; instruction 1 … instruction 4
```

More Macro Examples

```
sum_of_3   macro    arg1, arg2, arg3          ; WREG ← [arg1]+[arg2]+[arg3]
           movf     arg1,W,A
           addwf    arg2,W,A
           addwf    arg3,W,A
           endm

           sum_of_3 0x01, 0x02, 0x03          ; WREG ← [0x01] + [0x02] + [0x03]
```

Object File Directives

banksel <label>

- generate the instruction sequence to set active data bank to the one where <label> is located
- <label> must have been defined before the banksel directive is invoked.

```
bigq    set      0x300
        …
        …
        banksel  bigq        ; this directive will cause the assembler to
                             ; insert the instruction movlb 0x03
```

Object File Directives (continues)

[<label>] org <expr>

- sets the program origin for subsequent code at the address defined in <*expr*>.
- <label> will be assigned the value of <expr>.

```
reset     org     0x00
          goto    start
          …
start     …
led_pat   org     0x1000                ; led_pat has the value of 0x1000
          db      0x7E,0x30,0x6D,0x79,0x33,0x5B,0x5F,0x70,0x7F,0x7B
```

Object File Directives (continued)

processor <processor_type>

- Sets the processor type

```
processor   p18F8680     ; set processor type to PIC18F8680
```

Program Development Procedure

- Problem definition
- Algorithm development using pseudo code or flowchart
- Converting algorithm into assembly instruction sequence
- Testing program using normal data, marginal data, and erroneous data

Algorithm Representation

Step 1

…

Step 2

…

Step 3

…

Flowchart Symbols

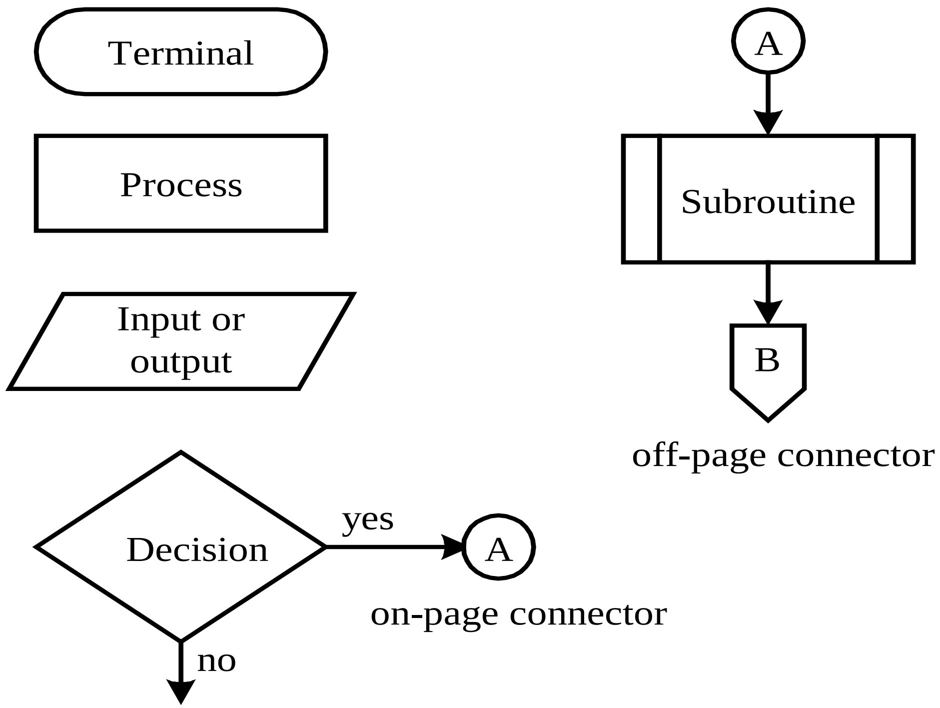


Figure 2.1 Flowchart symbols used in this book

Assembly Program Template

```
        org     0x0000      ; program starting address after power on reset
        goto    start
        org     0x08
        …                   ; high-priority interrupt service routine
        org     0x18
        …                   ; low-priority interrupt service routine
start   …
        …                   ; your program
        end
```

Program Template Before Interrupts Have Been Covered

```
        org     0x0000   ; program starting address after power on reset
        goto    start
        org     0x08
        retfie           ; high-priority interrupt service routine
        org     0x18
        retfie           ; low-priority interrupt service routine
start   …
        …                ; your program
        end
```

Case Issue

- The PIC18 instructions can be written in either uppercase or lowercase.
- MPASM allows the user to include “p18Fxxxx.inc” file to provide register definitions for the specific processor.
- All special function registers and bits are defined in uppercase.
- The convention followed in this text is: using **lowercase** for instructions and directives, using **uppercase** for special function registers.

Byte Order Issue

- This issue concerns how bytes are stored for multi-byte numbers.
- The **big-endian** method stores the most significant byte at the lowest address and stores the least significant byte in the highest address.
- The **little-endian** method stores the most significant byte of the number at the highest address and stores the least significant byte of the number in the lowest address.
- The 32-bit number 0x12345678 will stored as follows with two methods:

	Big-Endian Method				Little-Endian Method				
address	P	P+1	P+2	P+3	P	P+1	P+2	P+3	
value	12	34	56	78	78	56	34	12	(in hex)

Figure 02_t1 Byte order example

Programs for Simple Arithmetic Operations

Example 2.4 Write a program that adds the three numbers stored in data registers at 0x20, 0x30, and 0x40 and places the sum in data register at 0x50.

Solution:

Algorithm:

Step 1
Load the number stored at 0x20 into the WREG register.
Step 2
Add the number stored at 0x30 and the number in the WREG register and leave the sum in the WREG register.
Step 3
Add the number stored at 0x40 and the number in the WREG register and leave the sum in the WREG register.
Step 4
Store the contents of the WREG register in the memory location at 0x50.

The program that implements this algorithm is as follows:

```
        #include <p18F8720.inc>   ; can be other processor
        org     0x00
        goto    start
        org     0x08
        retfie
        org     0x18
        retfie
start   movf    0x20,W,A          ; WREG ← [0x20]
        addwf   0x30,W,A          ; WREG ← [0x20] + [0x30]
        addwf   0x40,W,A          ; WREG ← [0x20] + [0x30] + [0x40]
        movwf   0x50,A            ; 0x50 ← sum (in WREG)
        end
```

Example 2.5 Write a program to add two 24-bit numbers stored at 0x10~0x12 and 0x13~0x15 and leave the sum at 0x20..0x22.

Solution:

```
        #include <p18F8720.inc>
        org     0x00
        goto    start
        org     0x08
        retfie
        org     0x18
        retfie
start   movf    0x10,W,A        ; WREG ← [0x10]
        addwf   0x13,W,A        ; WREG ← [0x13] + [0x10]
        movwf   0x20,A          ; 0x20 ← [0x10] + [0x13]
        movf    0x11,W,A        ; WREG ← [0x11]
        addwfc  0x14,W,A        ; WREG ← [0x11] + [0x14] + C flag
        movwf   0x21,A          ; 0x21 ← [WREG]
        movf    0x12,W,A        ; WREG ← [0x12]
        addwfc  0x15,W,A        ; WREG ← [0x12] + [0x15] + C flag
        movwf   0x22,A          ; 0x22 ← [WREG]
        end
```

Example 2.6 Write a program to subtract 5 from memory locations 0x10 to 0x13.

Solution:

Algorithm:

Step 1. Place 5 in the WREG register.

Step 2. Subtract WREG from the memory location 0x10 and leave the difference in the memory location 0x10.

Step 3. Subtract WREG from the memory location 0x11 and leave the difference in the memory location 0x11.

Step 4. Subtract WREG from the memory location 0x12 and leave the difference in the memory location 0x12.

Step 5. Subtract WREG from the memory location 0x13 and leave the difference in the memory location 0x13.

The Program for Example 2.6

```
        #include <p18F8720.inc>
        org     0x00
        goto    start
        org     0x08
        retfie
        org     0x18
        retfie
start   movlw   0x05            ; WREG ← 0x05
        subwf   0x10,F,A        ; 0x10 ← [0x10] – 0x05
        subwf   0x11,F,A        ; 0x11 ← [0x11] – 0x05
        subwf   0x12,F,A        ; 0x12 ← [0x12] – 0x05
        subwf   0x13,F,A        ; 0x13 ← [0x13] – 0x05
        end
```

Example 2.7 Write a program that subtracts the number stored at 0x20..0x23 from the number stored at 0x10..0x13 and leaves the difference at 0x30..0x33.

Solution:

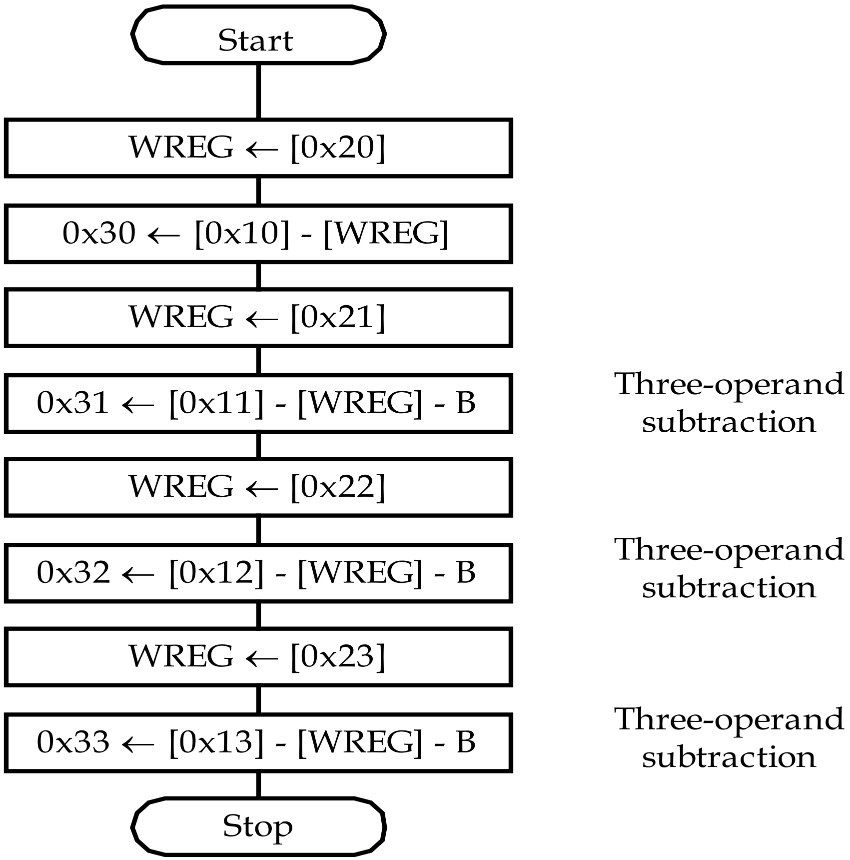


Figure 2.2 Logic flow of Example 2.7

The **program** for **Example 2.7**

```
            #include  <p18F8720.inc>
            org       0x00
            goto      start
            org       0x08
            retfie
            org       0x18
            retfie
start       movf      0x20, W, A       ; 0x30 ← [0x10] – [0x20]
            subwf     0x10, W, A       ;       “
            movwf     0x30, A          ;       “
            movf      0x21, W, A       ; 0x31 ← [0x11] – [0x21]
            subwfb    0x11, W, A       ;       “
            movwf     0x31, A          ;       “
            movf      0x22, W, A       ; 0x32 ← [0x12] – [0x22]
            subwfb    0x12, W, A       ;       “
            movwf     0x32, A          ;       “
            movf      0x23, W, A       ; 0x33 ← [0x13] – [0x23]
            subwfb    0x13, W, A       ;       “
            movwf     0x33, A          ;       “
            end
```

Binary Coded Decimal (BCD) Addition

- Decimal digits are encoded using 4 bits
- Two decimal digits are packed into a byte in memory
- After each addition, one needs to use the **daw** instruction to adjust and correct the result.

Let data register 0x24 and 0x25 holds BCD numbers, the following instruction sequence adds these two BCD numbers and saves the sum in 0x30

```
movf    0x24,W,A
addwf   0x25,W,A
daw
movwf   0x30,A
```

Example 2.9 Write an instruction sequence that adds the decimal numbers stored at 0x10...0x13 and 0x14...0x17 and stores the sum in 0x20..0x23.

Solution:

```
                #include   <p18F8720.inc>
                …
start           movf       0x10,W      ; add the least significant byte
                addwf      0x14,W      ;          “
                daw                    ; adjust for valid BCD
                movwf      0x20        ; save in the destination
                movf       0x11        ; add the second to least significant byte
                addwfc     0x15,W      ;          “
                daw                    ;          “
                movwf      0x21        ;          “
                movf       0x12        ; add the second to most significant byte
                addwfc     0x16        ;          “
                daw                    ;          “
                movwf      0x22        ;          “
                movf       0x13        ; add the most significant byte
                addwfc     0x17        ;          “
                daw                    ;          “
                movwf      0x23        ;          “
                end
```

Multiplication

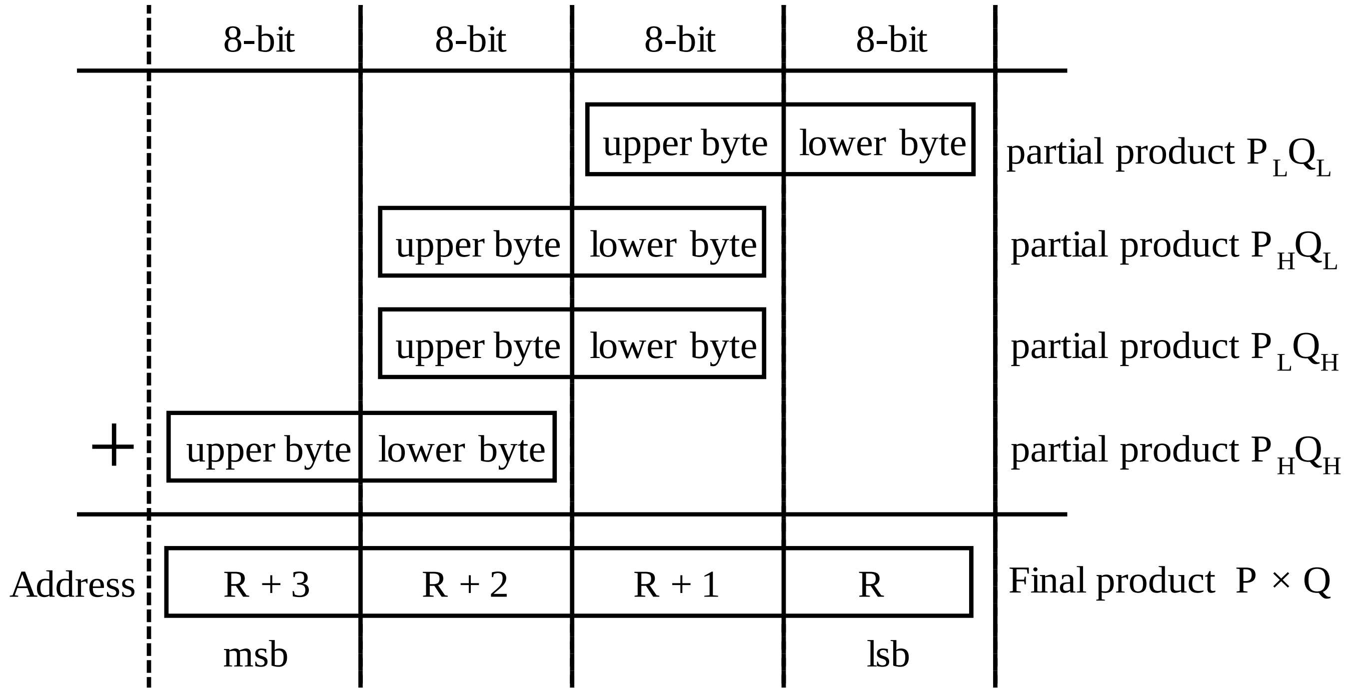
- PIC18 has two instructions for 8-bit multiplication: **mulwf f** and **mullw k**.
- The products are stored in the **PRODH:PRODL** register pair.
- The multiplication of numbers larger than 8 bits must be synthesized.
- The following instruction sequence performs 8-bit multiplication operation:

```
movf    0x10,W,A
mulwf   0x11,A
movff   PRODH,0x21      ; upper byte of the product
movff   PRODL,0x20      ; lower byte of the product
```

- To perform multiplication operation on numbers longer than 8 bits, the operand must be broken down into 8-bit chunks. Multiple 8-bit multiplications are performed and the resultant partial products are aligned properly and added together.
- Two 16-bit numbers P and Q can be broken down into as follows:

$P = P_H P_L$
$Q = Q_H Q_L$

Adding the Partial Products



Note: msb stands for most significant byte and lsb stands for least significant byte

Figure 2.4 16-bit by 16-bit multiplication

Instruction sequence to multiply two numbers that are stored at N:N+1 and M:M+1:

```
movwf   N,A           ;       "
movf    M+1,W,A
mulwf   N+1,A         ; compute MH × NH
movff   PRODL,PR+2
movff   PRODH,PR+3
movf    M,W,A         ; compute ML × NL
mulwf   N,A
movff   PRODL,PR
movff   PRODH,PR+1
movf    M,W,A
mulwf   N+1,A         ; compute ML × NH
movf    PRODL,W,A     ; add ML × NH to PR
addwf   PR+1,F,A      ;       "
movf    PRODH,W,A     ;       "
addwfc  PR+2,F,A      ;       "
movlw   0             ;       "
addwfc  PR+3,F,A      ; add carry
movf    M+1,W,A
mulwf   N,A           ; compute MH × NL
movf    PRODL,W,A     ; add MH × NL to PR
```

```
addwf    PR+1,F,A        ;          "
movf     PRODH,W,A       ;          "
addwfc   PR+2,F,A        ;          "
movlw    0               ;          "
addwfc   PR+3,F,A        ; add carry
nop
end
```

Program Loops

- Enable the microcontroller to perform repetitive operations.
- A loop may be executed a finite number of times or infinite number of times.

Program Loop Construct

1. Do S forever

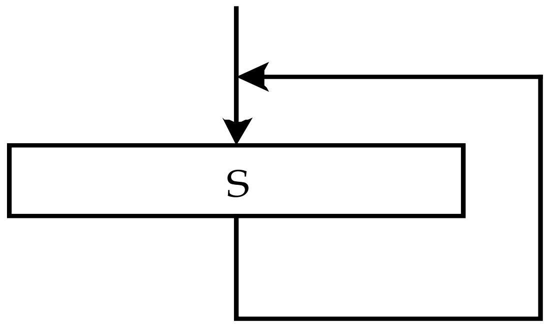
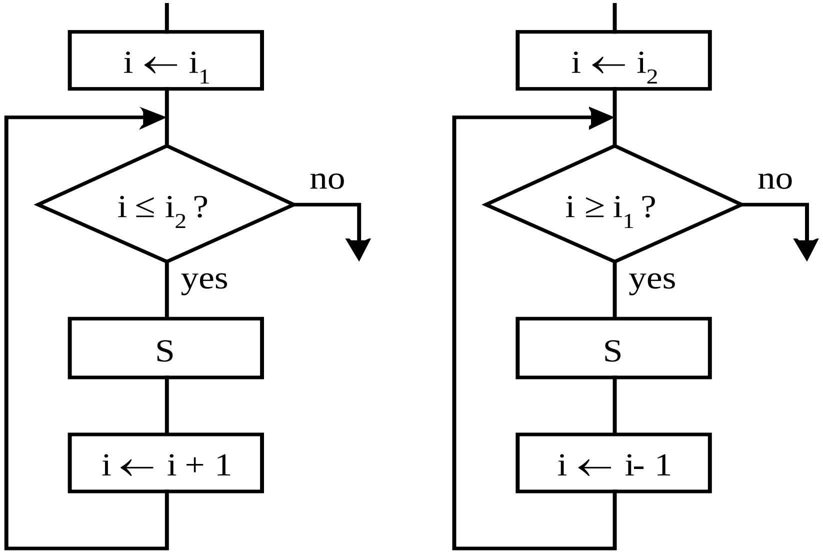


Figure 2.5 An infinite loop

2. for i = n1 to n2 Do S or **for i = n2 downto n1 do S**



(a) For $i = i_1$ to i_2 Do S (b) For $i = i_2$ downto i_1 Do S

Figure 2.6 A **For-loop** looping construct

3. while C do S

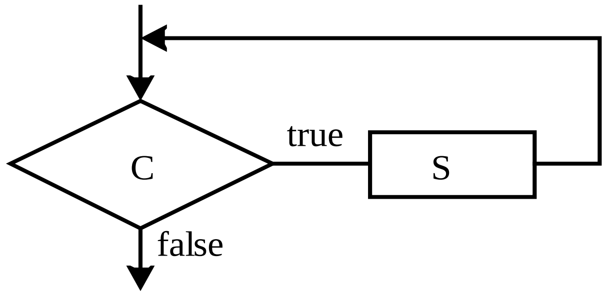


Figure 2.7 The **While ... Do** looping construct

4. repeat S until C

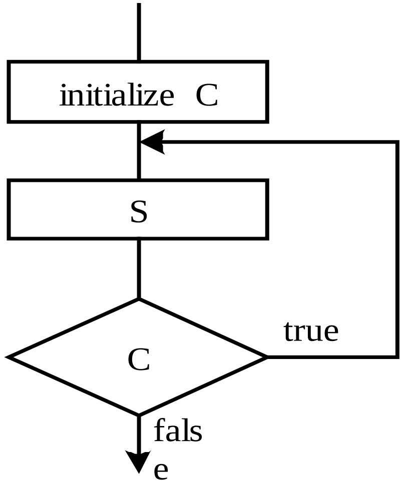


Figure 2.8 The **Repeat ... Until** looping construct

Changing the Program Counter

- Microcontroller executes instruction sequentially in normal condition.
- PIC18 has a 21-bit program counter (PC) which is divided into three registers: PCL, PCH, and PCU.
- PCL can be accessed directly. However, PCH and PCU are not directly accessible.
- One can accessed the values of PCH and PCU indirectly by accessing the PCLATH and PCLATU.
- Reading the PCL will cause the values of PCH and PCU to be copied into the PCLATH and PCLATU.
- Writing the PCL will cause the values of PCLATCH and PCLATU to be written into the PCH and PCU.
- In normal program execution, the PC value is incremented by either 2 or 4.
- To implement a program loop, the processor needs to change the PC value by a value other than 2 or 4.

Instructions for Changing Program Counter

BRA n: jump to the instruction with address equals to PC+2+n

B_{CC} n: jump to the instruction with address equals to PC+2+n if the condition code CC is true.

CC can be any one of the following:

C: C flag is set to 1

N: N flag is set to 1 which indicates that the previous operation result was negative

NN: N flag is 0 which indicates non-negative condition

NOV: V flag is 0 which indicates there is no overflow condition

NZ: Z flag is 0 which indicates the previous operation result was not zero

OV: V flag is 1 which indicates the previous operation caused an overflow

Z: Z flag is 1 which indicates the previous operation result was zero

goto n: jump to address represented by **n**

The destination of a **branch** or **goto** instruction is normally specified by a label.

Instructions for Changing Program Counter (continued)

cpfseq	f,a	; compare register f with WREG, skip if equal
cpfsgt	f,a	; compare register f with WREG, skip if equal
cpfslt	f,a	; compare register f with WREG, skip if less than
decfsz	f,d,a	; decrement f, skip if 0
dcfsnz	f,d,a	; decrement f, skip if not 0
incfsz	f,d,a	; increment f, skip if 0
infsnz	f,d,a	; increment f, skip if not 0
tstfsz	f,a	; test f, skip if 0
btfsc	f,b,a	; test bit b of register f, skip if 0
btfss	f,b,a	; test bit b of register f, skip if 1

Instructions for changing register value by 1

incf	f,d,a
decf	f,d,a

Examples of Program loops that execute n times

Example 1

```
i_cnt    equ      PRODL    ; use PRODL as loop count
         clrf     i_cnt,A
i_loop   …
         …                 ; i_cnt is incremented in the loop
         movlw    n
         cpfseq   i_cnt,A  ; compare i_cnt with WREG and skip if equal
         goto     i_loop   ; executed when i_cnt ≠ loop limit
```

Example 2

```
n        equ      20          ; n has the value of 20
lp_cnt   set      0x10        ; assign file register 0x10 to lp_cnt
         …
         movlw    n
         movwf    lp_cnt      ; prepare to repeat the loop for n times
loop     …                    ; program loop
         …                    ;       “
         decfsz   lp_cnt,F,A  ; decrement lp_cnt and skip if equal to 0
         goto     loop        ; executed if lp_cnt ≠ 0
```

Example 2.12 Write a program to compute 1 + 2 + 3 + … + n and save the sum at 0x00 and 0x01.

Solution:

1. Program logic

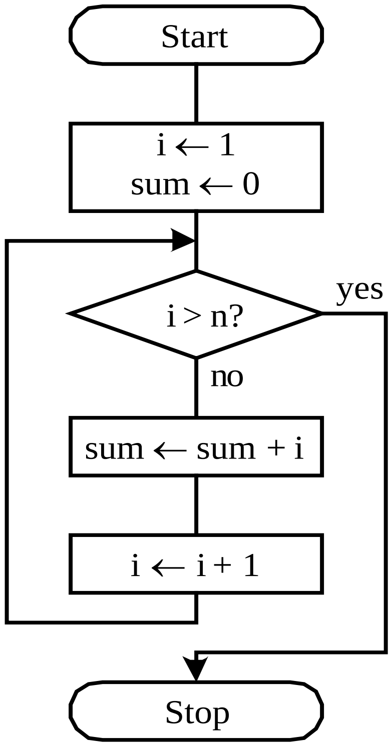


Figure 2.12 Flowchar for computing 1+2+...+n

Program of Example 2.12 (in **for i = n1 to n2** construct)

```
        #include <p18F8720.inc>
        radix   dec
n       equ     D'50'
sum_hi  set     0x01      ; high byte of sum
sum_lo  set     0x00      ; low byte of sum
i       set     0x02      ; loop index i
        org     0x00      ; reset vector
        goto    start
        org     0x08
        retfie
        org     0x18
        retfie
start   clrf    sum_hi,A  ; initialize sum to 0
        clrf    sum_lo,A  ;          "
        clrf    i,A       ; initialize i to 0
        incf    i,F,A     ; i starts from 1
sum_lp  movlw   n         ; place n in WREG
        cpfsgt  i,A       ; compare i with n and skip if i > n
        bra     add_lp    ; perform addition when i ≤ 50
        bra     exit_sum  ; it is done when i > 50
```

```
add_lp    movf     i,W,A        ; place i in WREG
          addwf    sum_lo,F,A   ; add i to sum_lo
          movlw    0
          addwfc   sum_hi,F,A   ; add carry to sum_hi
          incf     i,F,A        ; increment loop index i by 1
          bra      sum_lp
exit_sum  nop
          bra      exit_sum
          end
```

Example 2.13
Write a program to find the largest element stored in the array that is stored in data memory locations from 0x10 to 0x5F.

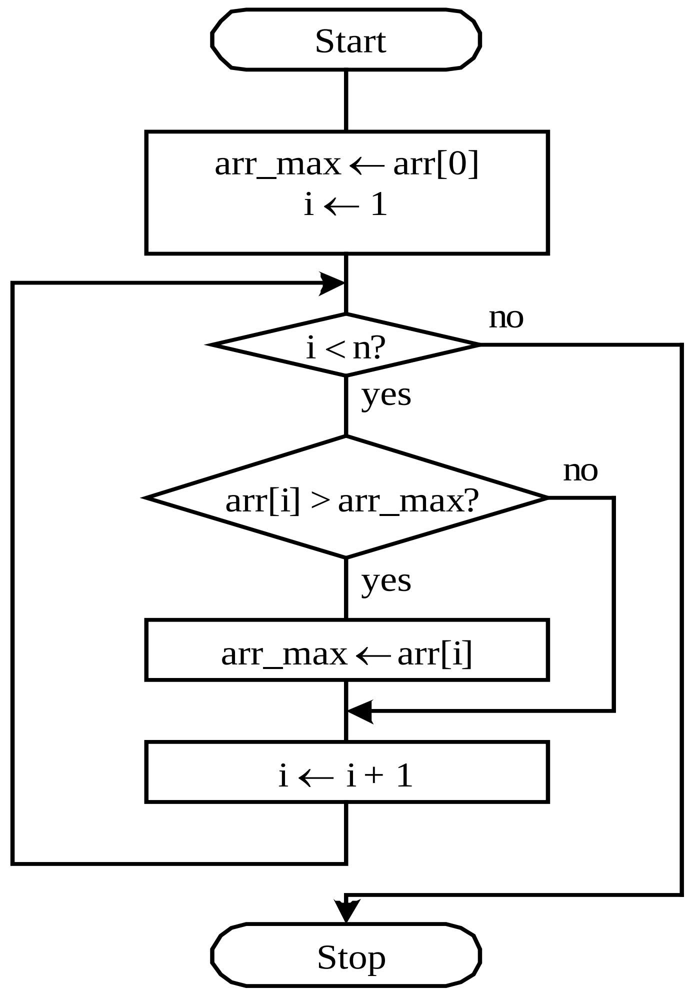


Figure 2.13 Flowchart for finding the maximum array element

Program for Example 2.13

```
arr_max   equ       0x00
i         equ       0x01
n         equ       D'80'              ; the array count
          #include  <p18F8720.inc>
          org       0x00
          goto      start
          org       0x08
          retfie
          org       0x18
          retfie
start     movff     0x10,arr_max       ; set arr[0] as the initial array max
          lfsr      FSR0,0x11          ; place address of arr[1] in FSR0
          clrf      i,A                ; initialize loop count i to 0
again     movlw     n-1                ; number of comparisons to be made
; the next instruction implements the condition C (i = n)
          cpfslt    i,A                ; skip if i < n-1
          bra       done               ; all comparisons have been done
; the following 7 instructions update the array max
          movf      POSTINC0,W
```

```
            cpfsgt    arr_max,A        ; is arr_max > arr[i]?
            bra       replace          ; no
            bra       next_i       ; yes
replace     movwf     arr_max,A    ; update the array max
next_i      incf      i,F,A
            goto      again
done        nop
            end
```

Reading and Writing Data in Program Memory

- PIC18 provides TBLRD and TBLWT instructions for accessing data in program memory.
- The operations of reading data from and writing data into program memory are shown in Figure 2.14 and 2.15.

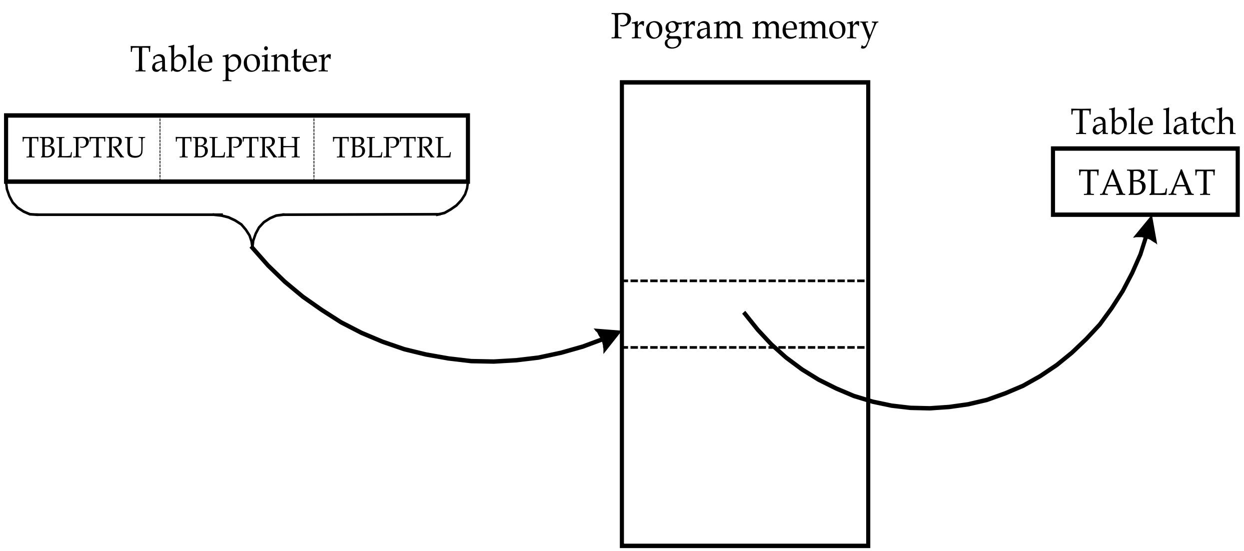


Figure 2.14 Table read operation (Redraw with permission of Microchip)

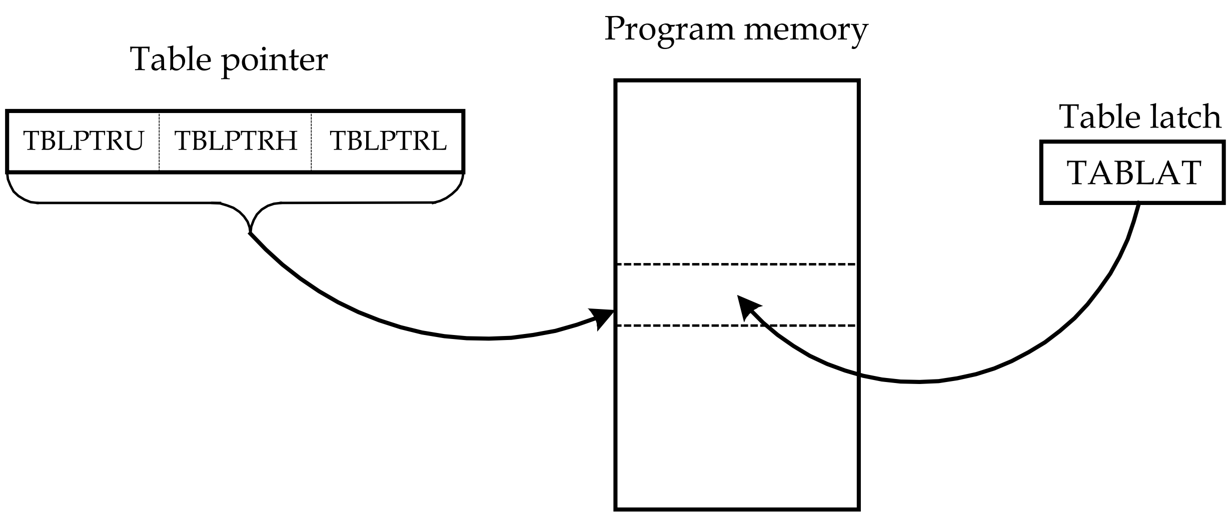


Figure 2.15 Table write operation (redraw with permission of Microchip)

The table pointer consists of three registers:

- TBLPTRU (6 bits)
- TBLPTRH (8 bits)
- TBLPTRL (8 bits)

Versions of table read and table write instructions

Table 2.11 PIC18 MCU table read and write instructions

Mnemonic, operator	Description	16-bit instruction word	Status affected
TBLRD*	Table read	0000 0000 0000 1000	none
TBLRD*+	Table read with post-increment	0000 0000 0000 1001	none
TBLRD*-	Table read with post-decrement	0000 0000 0000 1010	none
TBLRD+*	Table read with pre-increment	0000 0000 0000 1011	none
TBLWT*	Table write	0000 0000 0000 1100	none
TBLWT*+	Table write with post-increment	0000 0000 0000 1101	none
TBLWT*-	Table write with post-decrement	0000 0000 0000 1110	none
TBLWT+*	Table write with pre-increment	0000 0000 0000 1111	none

Reading the program memory location **prog_loc** involves two steps:

Step 1. Place the address of **prog_loc** in TBLPTR registers

```
movlw    upper prog_loc
movwf    TBLPTRU,A
movlw    high prog_loc
movwf    TBLPTRH,A
movlw    low prog_loc
movwf    TBLPTRL,A
```

Step 2. Perform a TBLRD instruction.

```
tblrd
```

The TBLPTR registers can be incremented or decremented before or after the read or write operations as shown in Table 2.11.

Logic Instructions

Table 2.12 PIC18 MCU logic instructions

Mnemonic, operator	Description
andwf f,d,a	AND WREG with f
comf f,d,a	Complement f
iorwf f,d,a	Inclusive OR WREG with f
negf f,a	Negate f
xorwf f,d,a	Exclusive OR WREG with f
andlw k	AND literal with WREG
iolw k	Inclusive OR literal with WREG
xorlw k	Exclusive OR literal with WREG

Applications of Logic Instructions

1. Set a few bits in a byte
2. Clear certain bits in a byte
3. Toggle certain bits in a byte

To **set bits** 7, 6, and 0 of PORTA to 1

```
movlw    B'11000001'
iorwf    PORTA,F,A
```

To **clear bits** 4, 2, and 1 of PORTB to 0

```
movlw    B'11101001
andwf    PORTB,F,A
```

To **toggle bits** odd bits of PORTC

```
movlw    B'10101010'
xorwf    PORTC
```

Example 2.16 Write a program to find out the number of elements in an array of 8-bit elements that are a multiple of 8. The array is in the program memory.

Solution:

1. A number must have the lowest 3 bits equal to 0 to be a multiple of 8
2. Use the **Repeat S until C** looping construct

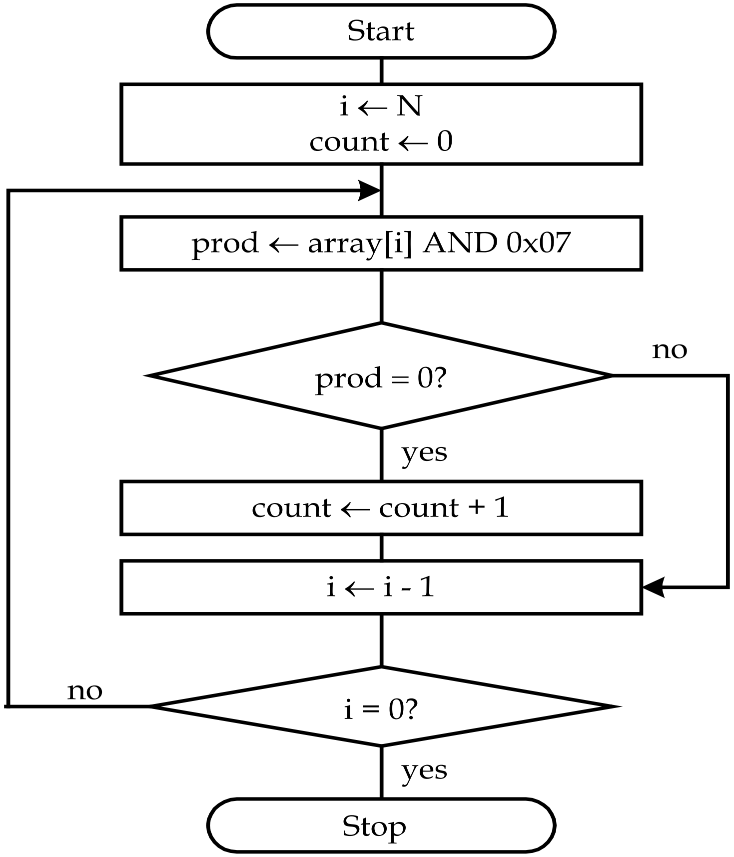


Figure 2.16 Flowchart for Example 2.16

```
        #include <p18F8720.inc>
ilimit  equ     0x20            ; loop index limit
count   set     0x00
ii      set     0x01            ; loop index
mask    equ     0x07            ; used to masked upper five bits
        org     0x00
        goto    start
        …                       ; interrupt service routines
start   clrf    count,A
        movlw   ilimit
        movwf   ii              ; initialize ii to ilimit
        movlw   upper array
        movwf   TBLPTRU,A
        movlw   high array
        movwf   TBLPTRH,A
        movlw   low array
        movwf   TBLPTRL,A
        movlw   mask
i_loop  tblrd*+                 ; read an array element into TABLAT
        andwf   TABLAT,F,A
        bnz     next            ; branch if not a multiple of 8
```

```
            incf      count,F,A         ; is a multiple of 8
next        decfsz    ii,F,A            ; decrement loop count
            bra       i_loop
            nop
array       db        0x00,0x01,0x30,0x03,0x04,0x05,0x06,0x07,0x08,0x09
            db        0x0A,0x0B,0x0C,0x0D,0x0E,0x0F,0x10,0x11,0x12,0x13
            db        0x14,0x15,0x16,0x17,0x18,0x19,0x1A,0x1B,0x1C,0x1D
            db        0x1E,0x1F
            end
```

Using Program Loops to Create Time Delays

- The PIC18 uses a crystal oscillator or a RC circuit to generate the clock signal needed to control its operation.
- The instruction execution time is measured by using the instruction cycle clock.
- One instruction cycle is equal to four times the crystal oscillator clock period.
- Select an appropriate instruction that will take a multiple of 10 or 20 instruction cycles to execute.
- A desirable time delay is created by repeating the chosen instruction sequence for certain number of times.

A Macro to Repeat An Instruction for Certain Number of Times

```
dup_nop    macro      kk             ; duplicate the nop instruction kk times
           variable   i
i = 0
           while i < kk
           nop                       ; takes 1 instruction cycle time
i += 1
           endw
           endm
```

To create 0.5 ms time delay with 40 MHz crystal oscillator

```
           radix      dec
loop_cnt   equ        PRODL
           movlw      250
           movlw      loop_cnt,A
again      dup_nop    17             ; insert 17 nop instructions
           decfsz     loop_cnt,F,A   ; 1 instruction cycle
           bra        again          ; 2 instruction cycles
```

Example 2.18 Write a program to create a time delay of 100 ms for the demo board that uses a 40 MHz crystal oscillator to operate.

Solution: Repeat the previous instruction sequence for 200 times can create a 100 ms time delay.

```
        radix    dec
lp_cnt1 equ      0x21
lp_cnt2 equ      0x22
        movlw    200
        movwf    lp_cnt1,A
loop1   movlw    250
        movwf    lp_cnt2,A
loop2   dup_nop  17          ; 17 instruction cycles
        decfsz   lp_cnt2,F,A ; 1 instruction cycle (2 when [lp_cnt1] = 0)
        bra      loop2       ; 2 instruction cycles
        decfsz   lp_cnt1,F,A
        bra      loop1
```

Rotate Instructions

rlcf f, d, a ; rotate left **f** through carry

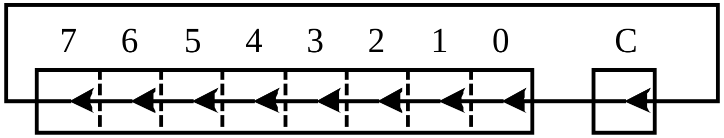


Figure 2.17 Operation performed by the **rlcf f,d,a** instruction

rlncf f, d, a ; rotate left **f** (not trough carry)

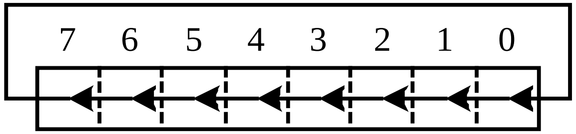


Figure 2.18 Operation performed by the **rlncf f,d,a** instruction

rrcf **f, d, a** ; rotate right **f** through carry

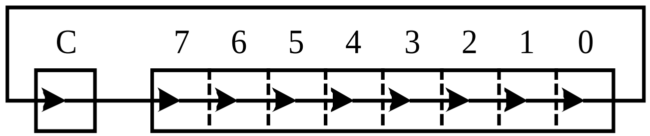


Figure 2.19 Operation performed by the **rrcf f,d,a** instruction

rrncf **f, d, a** ; rotate right f (not through carry)

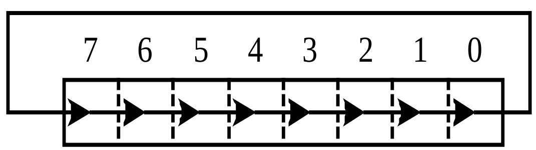
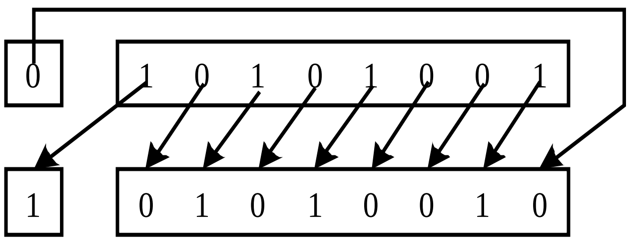


Figure 2.20 Operation performed by the **rrncf f,d,a** instruction

Example 2.19 Compute the new values of the data register 0x10 and the C flag after the execution of the **rlcf 0x10,F,A** instruction. [0x10] = 0xA9, C = 1

Solution:



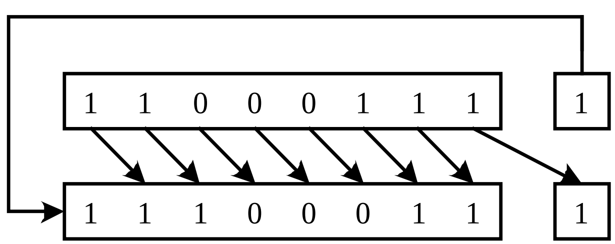
The result is

Original value	New value
[0x10] = 1010 1001 C = 0	[0x10] = 01010010 C = 1

Figure 2.21 Operation of the RLCF 0x10,F,A instruction

Example 2.20 Compute the new values of the data register 0x10 and the C flag after the execution of the **rrcf 0x10,F,A** instruction. [0x10] = 0xC7, C = 1

Solution:



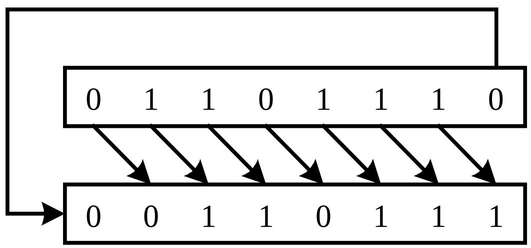
The result is

Original value	New value
[0x10] = 1100 0111 C = 1	[0x10] = 1110 0011 C = 1

Figure 2.22 Operation of the RRCF 0x10,F,A instruction

Example 2.21 Compute the new values of the data memory location 0x10 after the execution of the **rrncf 0x10,F,A** instruction and the **rlncf 0x10,F,A** instruction, respectively. [0x10] = 0x6E

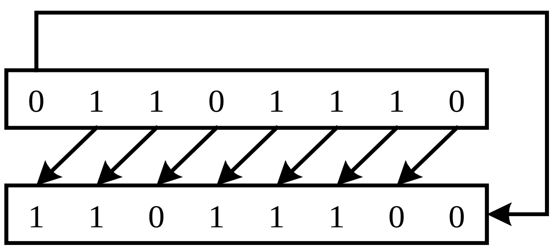
Solution:



The result is

original value	new value
[0x10] = 0110 1110	[0x10] = 0011 0111

Figure 2.23 Operation performed by the rrncf 0x10, F, A instruction



The result is

Before	After
[0x10] = 0110 1110	[0x10] = 1101 1100

Figure 2.24 Operation performed by the **rlncf 0x10, F, A** instruction

Bit Operation Instructions

```
bcf   f, b, a     ; clear bit b of register f
bsf   f, b, a     ; set bit b of register f
btg   f, b, a     ; toggle bit b of register f
```

Examples

1. bcf STATUS,C,A ; clear the C flag of the STATUS register
2. bsf sign,0,A ; set the bit 0 of register **sign** to 1
3. btg sign,0,A ; toggle bit 0 of register **sign** (0 to 1 or 1 to 0)

Perform Multiplication by Shift Left Operations

Multiply the 3-byte number store at 0x00…0x02 by 8

```
        movlw   0x03            ; set loop count to 3
loop    bcf     STATUS, C, A    ; clear the C flag
        rlcf    0x00, F, A      ; shift left one place
        rlcf    0x01, F, A      ;       “
        rlcf    0x02, F, A      ;       “
        decfsz  WREG,W,A        ; have we shifted left three places yet?
        goto    loop            ; not yet, continue
```

Perform Division by Shifting to the Right

Divide the 3-byte number stored at 0x10…0x12

```
        movlw   0x04            ; set loop count to 4
loop    bcf     STATUS, C, A    ; shift the number to the right 1 place
        rrcf    0x12, F, A      ;    "
        rrcf    0x11, F, A      ;    "
        rrcf    0x10, F, A      ;    "
        decfsz  WREG,W,A        ; have we shifted right four places yet?
        goto    loop            ; not yet, continue
```