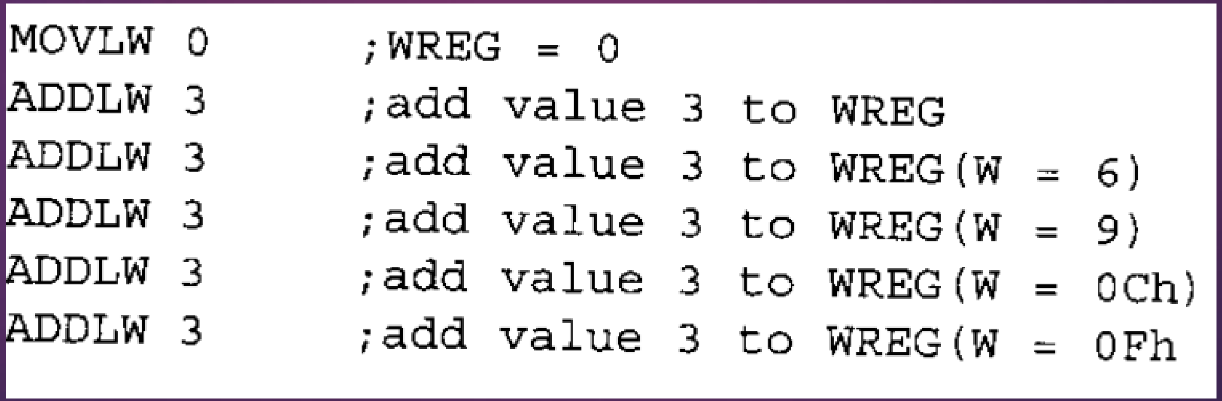
Microprocessor Interfacing & Programming

LECTURE 9&10

Branch Intsructions and Looping

- **Looping in PIC:** Repeating a sequence of instructions or an operation a certain number of times is called **loop.**
- In PIC, several ways to repeat an operation many times.

Drawback???



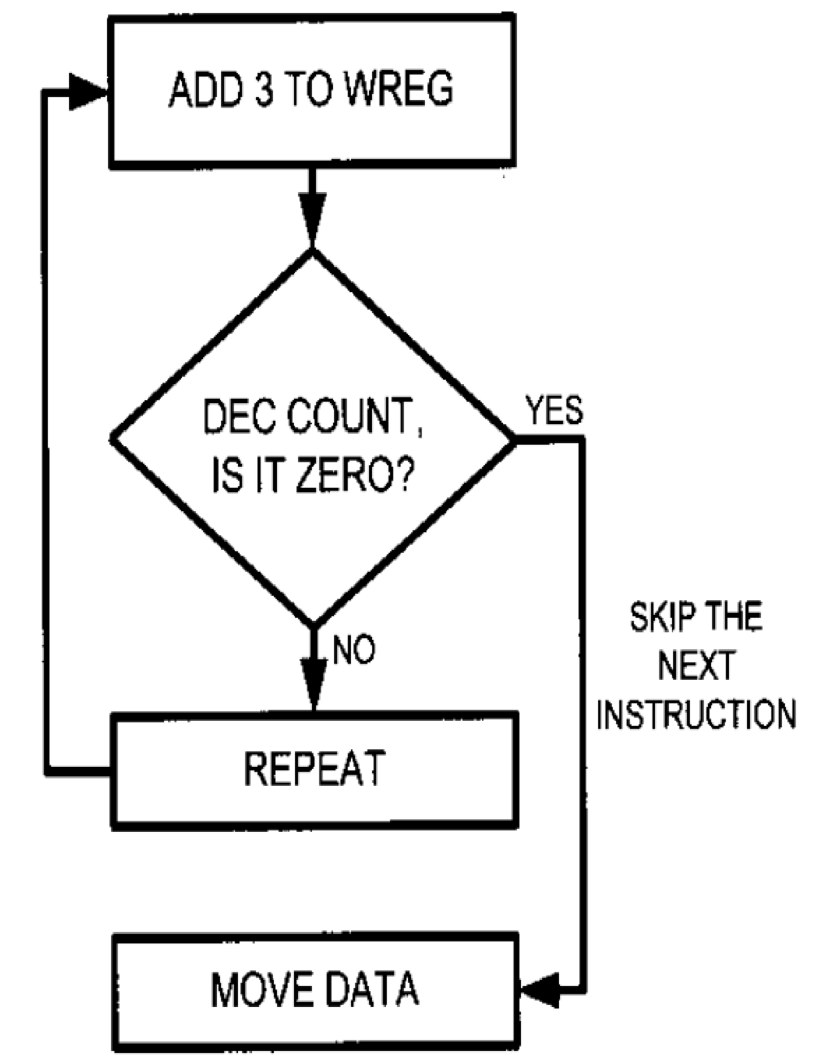
```
MOVLW  0        ;WREG = 0
ADDLW  3        ;add value 3 to WREG
ADDLW  3        ;add value 3 to WREG(W = 6)
ADDLW  3        ;add value 3 to WREG(W = 9)
ADDLW  3        ;add value 3 to WREG(W = 0Ch)
ADDLW  3        ;add value 3 to WREG(W = 0Fh
```

Ways to do looping in PIC

- 2 methods for looping.
  - DECFSZ
  - BNZ
- **DECFSZ instruction and looping:**
- It is decrement fileReg skip zero, widely used, supported across all PIC families.
- **Format:** DECFSZ fileReg, d

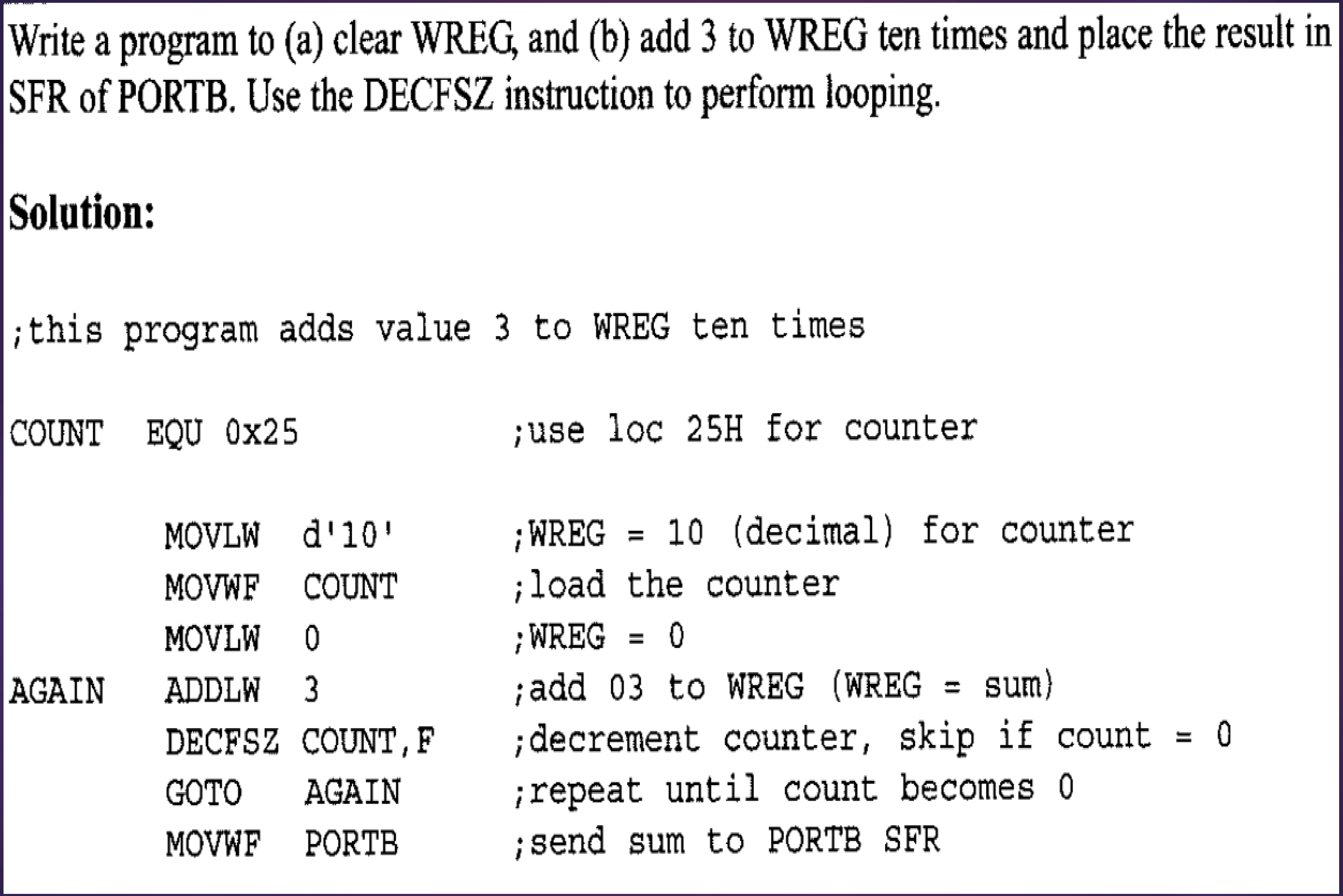
By placing, the "GOTO target" instruction right below it, we can create a loop.

The target address of the "GOTO target" instruction is the beginning of the loop.



INSTRUCTIONS

```
AGAIN     ADDLW 3

          DECFSZ COUNT

          GOTO AGAIN

          MOVWF PORTB
```



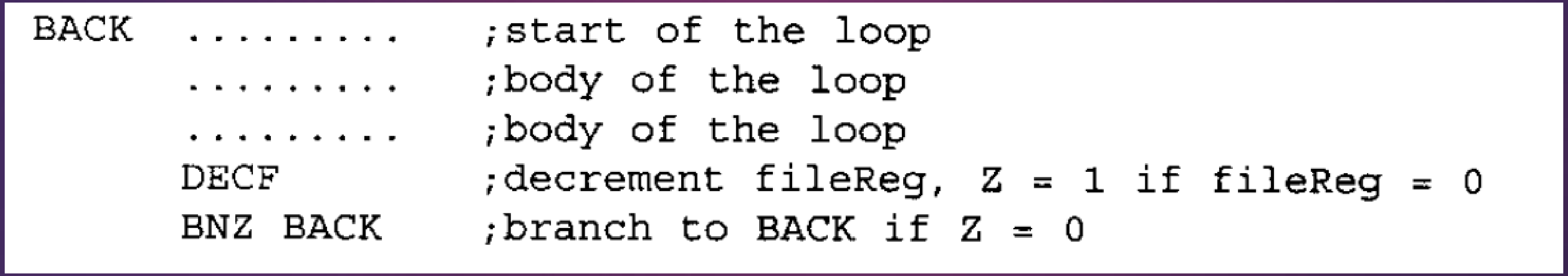
Write a program to (a) clear WREG, and (b) add 3 to WREG ten times and place the result in SFR of PORTB. Use the DECFSZ instruction to perform looping.

Solution:

```
;this program adds value 3 to WREG ten times

COUNT   EQU 0x25              ;use loc 25H for counter

        MOVLW   d'10'         ;WREG = 10 (decimal) for counter
        MOVWF   COUNT         ;load the counter
        MOVLW   0             ;WREG = 0
AGAIN   ADDLW   3             ;add 03 to WREG (WREG = sum)
        DECFSZ  COUNT,F       ;decrement counter, skip if count = 0
        GOTO    AGAIN         ;repeat until count becomes 0
        MOVWF   PORTB         ;send sum to PORTB SFR
```

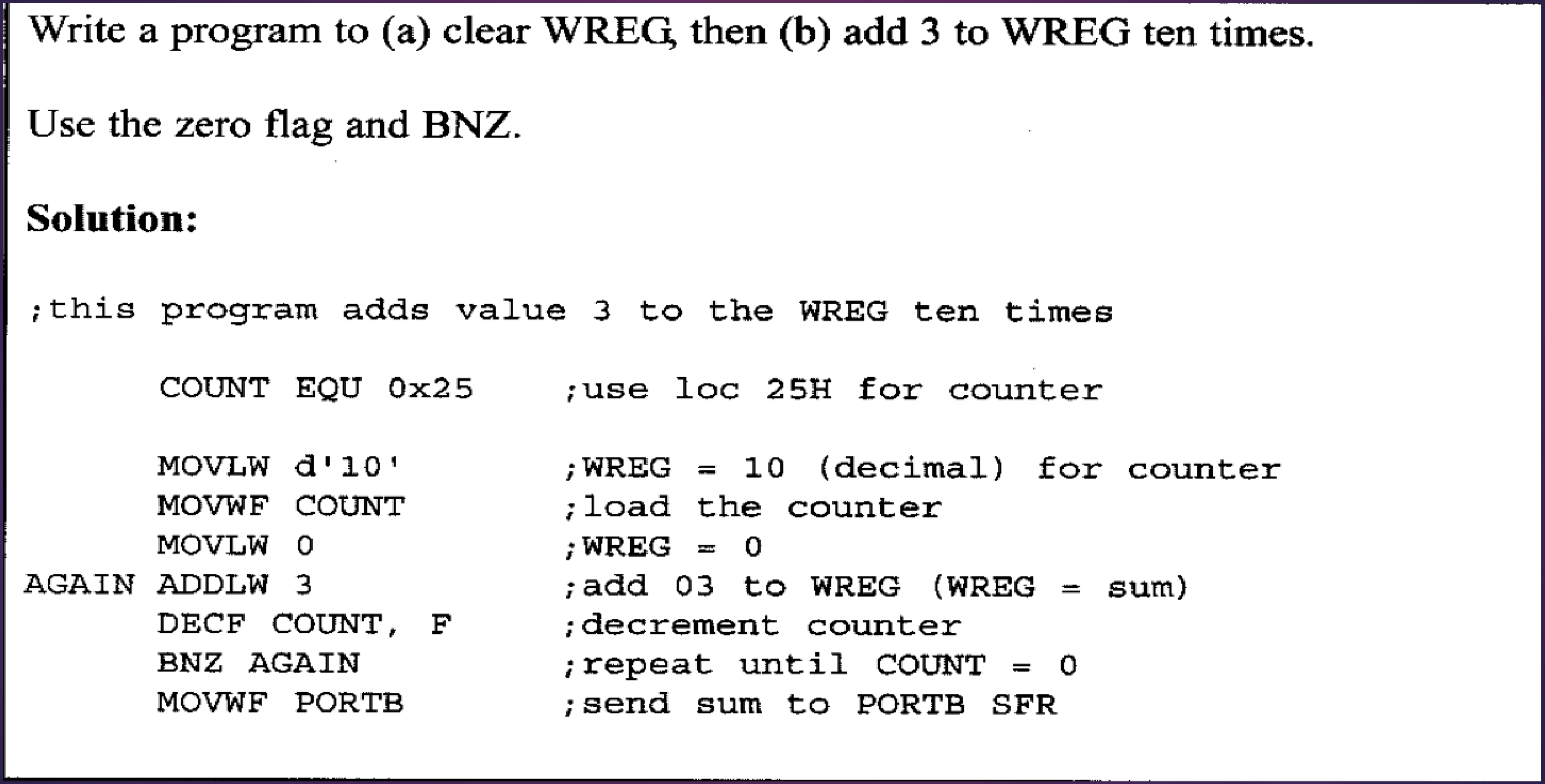
BNZ for looping

- This instruction BNZ (branch if not zero) is supported by PIC18 family and not earlier families such as PIC16 or PIC12.
- It uses zero flag in the status register.



```
BACK   .........    ;start of the loop
       .........    ;body of the loop
       .........    ;body of the loop
       DECF         ;decrement fileReg, Z = 1 if fileReg = 0
       BNZ  BACK    ;branch to BACK if Z = 0
```

- Notice that the BNZ instruction refers to the Z flag of the status register affected by the previous instruction, DECF.

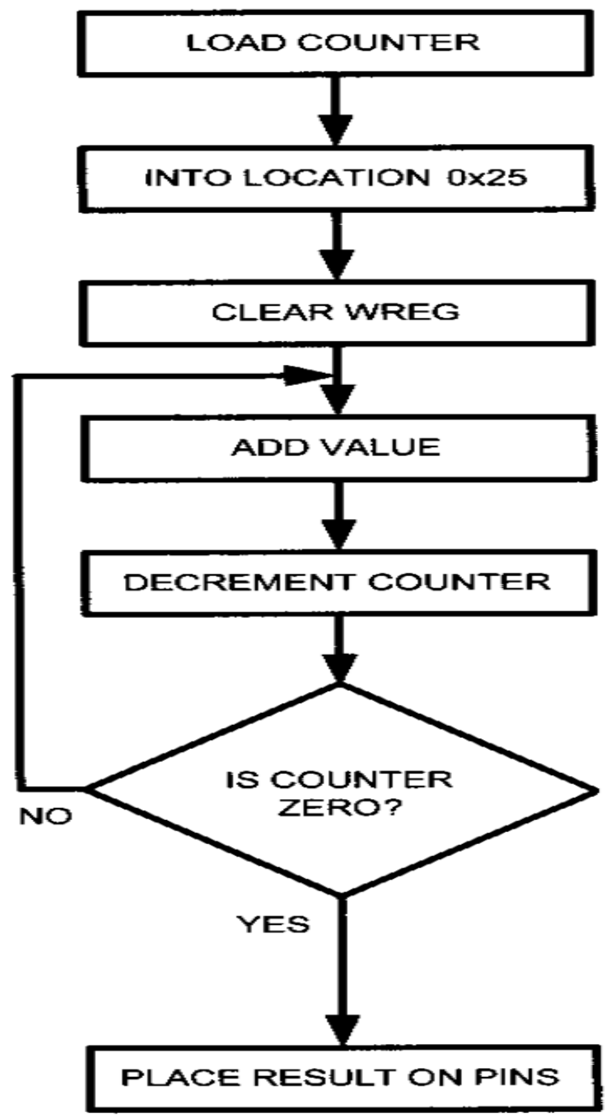


Write a program to (a) clear WREG, then (b) add 3 to WREG ten times.

Use the zero flag and BNZ.

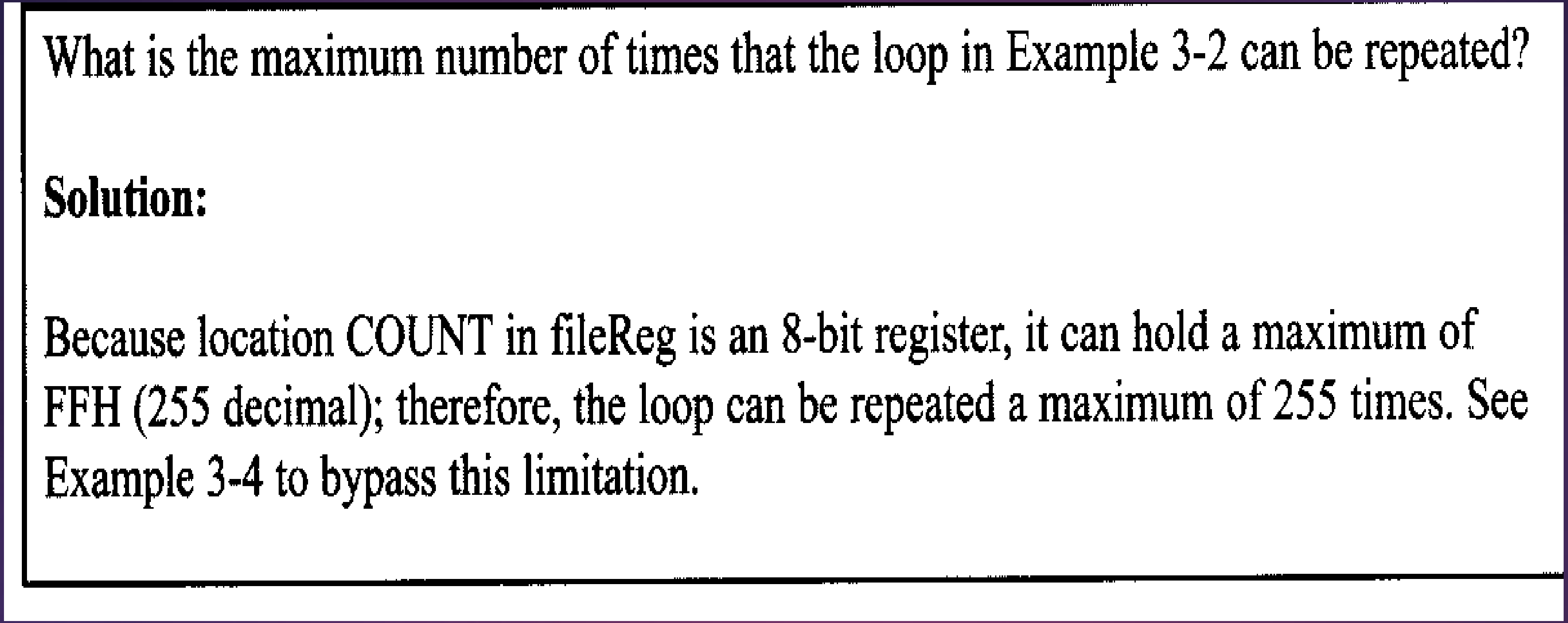
Solution:

```
;this program adds value 3 to the WREG ten times

      COUNT EQU 0x25     ;use loc 25H for counter

      MOVLW d'10'        ;WREG = 10 (decimal) for counter
      MOVWF COUNT        ;load the counter
      MOVLW 0            ;WREG = 0
AGAIN ADDLW 3            ;add 03 to WREG (WREG = sum)
      DECF COUNT, F      ;decrement counter
      BNZ AGAIN          ;repeat until COUNT = 0
      MOVWF PORTB        ;send sum to PORTB SFR
```



INSTRUCTIONS

```
        MOVLW D'10'

        MOVWF COUNT

        MOVLW 0

AGAIN   ADDLW 3

        DECF COUNT, F

        BNZ AGAIN

        MOVWF PORTB
```



What is the maximum number of times that the loop in Example 3-2 can be repeated?

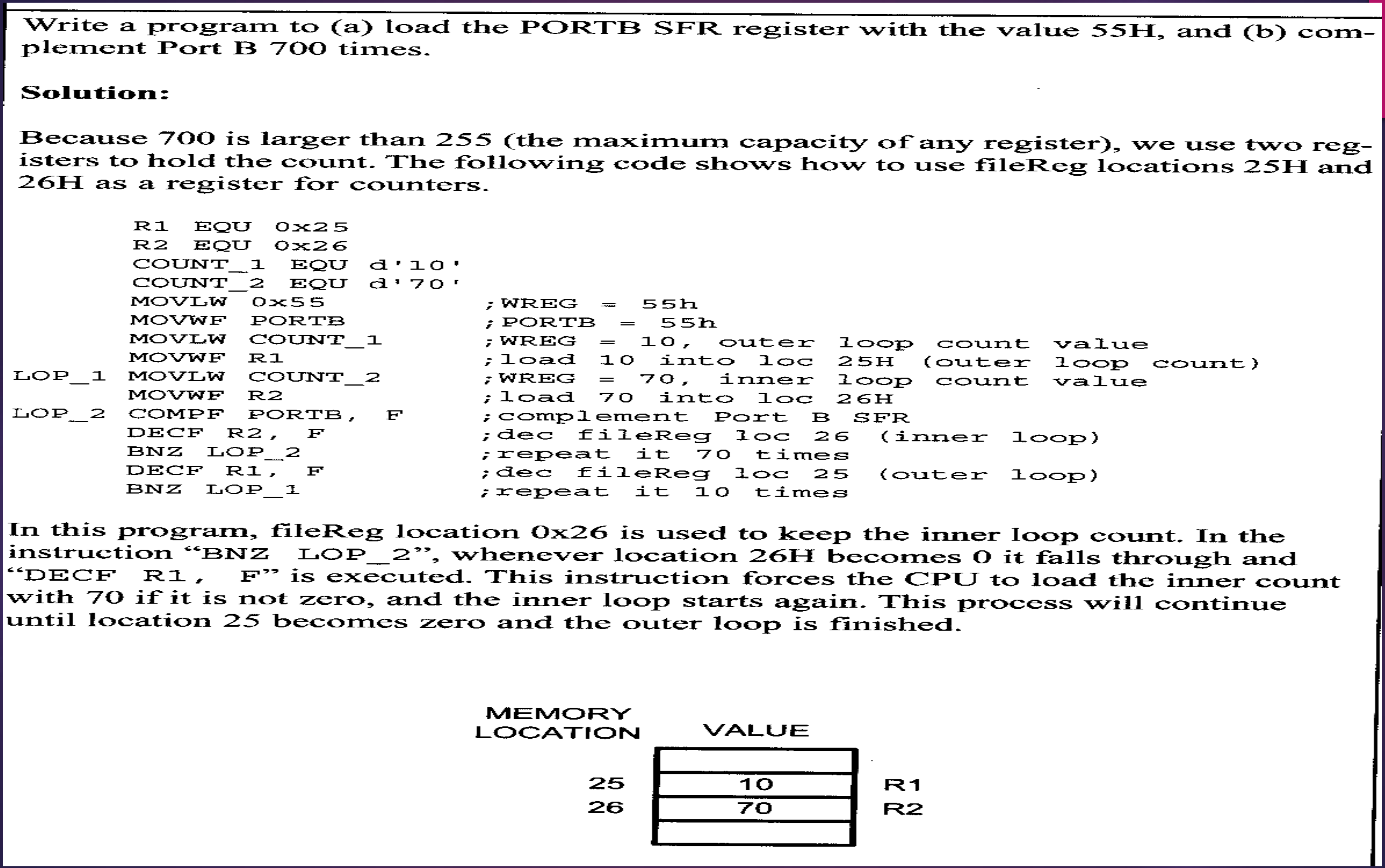
Solution:

Because location COUNT in fileReg is an 8-bit register, it can hold a maximum of FFH (255 decimal); therefore, the loop can be repeated a maximum of 255 times. See Example 3-4 to bypass this limitation.

If we want the loop to be repeated more than 255 times then???

Loop inside a Loop

- To repeat an action more times than 255, we use loop inside a loop, which is called a **nested loop**.
- In nested loop, we use two registers to hold the count.



Write a program to (a) load the PORTB SFR register with the value 55H, and (b) complement Port B 700 times.

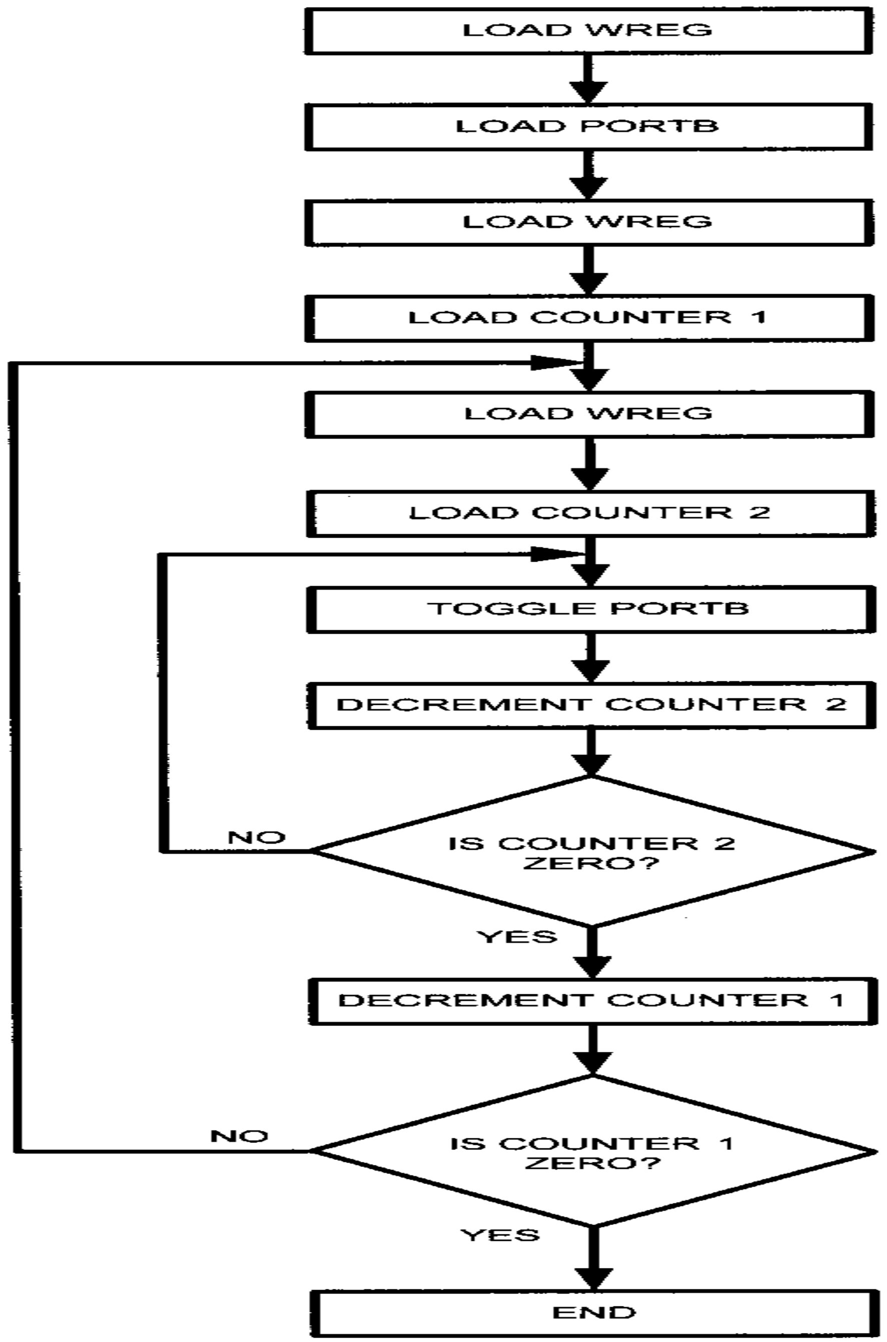
Solution:

Because 700 is larger than 255 (the maximum capacity of any register), we use two registers to hold the count. The following code shows how to use fileReg locations 25H and 26H as a register for counters.

```
        R1  EQU  0x25
        R2  EQU  0x26
        COUNT_1  EQU  d'10'
        COUNT_2  EQU  d'70'
        MOVLW  0x55             ;WREG = 55h
        MOVWF  PORTB            ;PORTB = 55h
        MOVLW  COUNT_1          ;WREG = 10, outer loop count value
        MOVWF  R1               ;load 10 into loc 25H (outer loop count)
LOP_1   MOVLW  COUNT_2          ;WREG = 70, inner loop count value
        MOVWF  R2               ;load 70 into loc 26H
LOP_2   COMPF  PORTB,  F        ;complement Port B SFR
        DECF  R2,  F            ;dec fileReg loc 26 (inner loop)
        BNZ  LOP_2              ;repeat it 70 times
        DECF  R1,  F            ;dec fileReg loc 25 (outer loop)
        BNZ  LOP_1              ;repeat it 10 times
```

In this program, fileReg location 0x26 is used to keep the inner loop count. In the instruction "BNZ LOP_2", whenever location 26H becomes 0 it falls through and "DECF R1, F" is executed. This instruction forces the CPU to load the inner count with 70 if it is not zero, and the inner loop starts again. This process will continue until location 25 becomes zero and the outer loop is finished.

MEMORY LOCATION	VALUE	
25	10	R1
26	70	R2

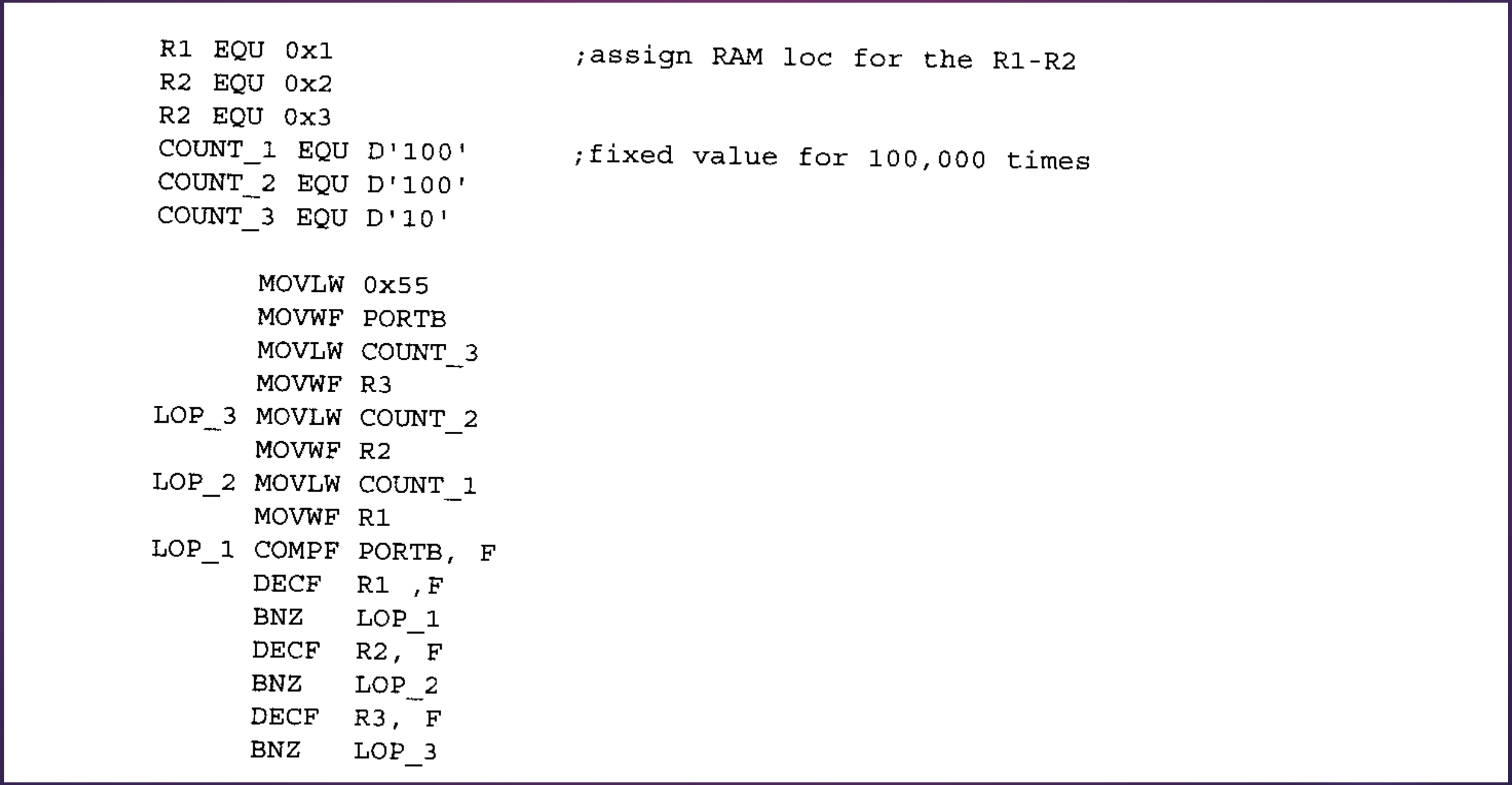


INSTRUCTIONS

```
        MOVLW 0x55
        MOVWF PORTB
        MOVLW COUNT_1
        MOVWF R1
LOP_1   MOVLW COUNT_2
        MOVWF R1
LOP_2   COMPF PORTB, F
        DECF R2, F
        BNZ LOP_2
        DECF R1, F
        BNZ LOP_1
```

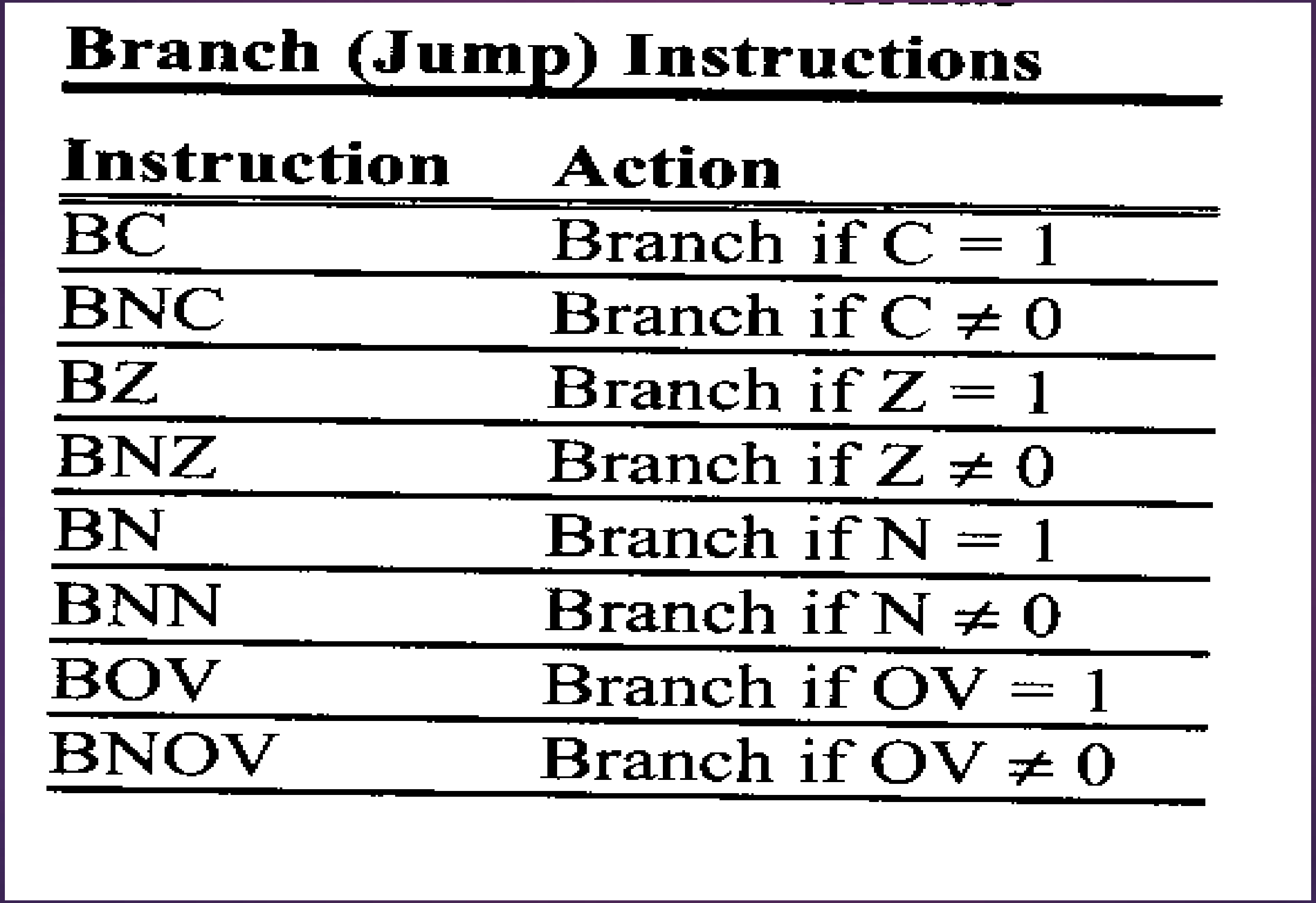
Looping 100,000 times

- As two registers give us a maximum value of 65025 (255 x 255), we can use three registers to get up more iterations.



```
R1  EQU  0x1                        ;assign RAM loc for the R1-R2
R2  EQU  0x2
R2  EQU  0x3
COUNT_1  EQU  D'100'                ;fixed value for 100,000 times
COUNT_2  EQU  D'100'
COUNT_3  EQU  D'10'

         MOVLW  0x55
         MOVWF  PORTB
         MOVLW  COUNT_3
         MOVWF  R3
LOP_3    MOVLW  COUNT_2
         MOVWF  R2
LOP_2    MOVLW  COUNT_1
         MOVWF  R1
LOP_1    COMPF  PORTB,  F
         DECF   R1  ,F
         BNZ    LOP_1
         DECF   R2,  F
         BNZ    LOP_2
         DECF   R3,  F
         BNZ    LOP_3
```

Other Conditional Jumps

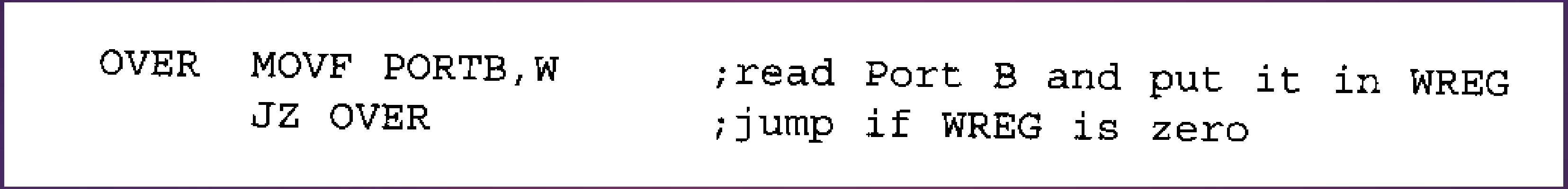


Branch (Jump) Instructions

Instruction	Action
BC	Branch if C = 1
BNC	Branch if C ≠ 0
BZ	Branch if Z = 1
BNZ	Branch if Z ≠ 0
BN	Branch if N = 1
BNN	Branch if N ≠ 0
BOV	Branch if OV = 1
BNOV	Branch if OV ≠ 0

BZ (Branch if Z=1)

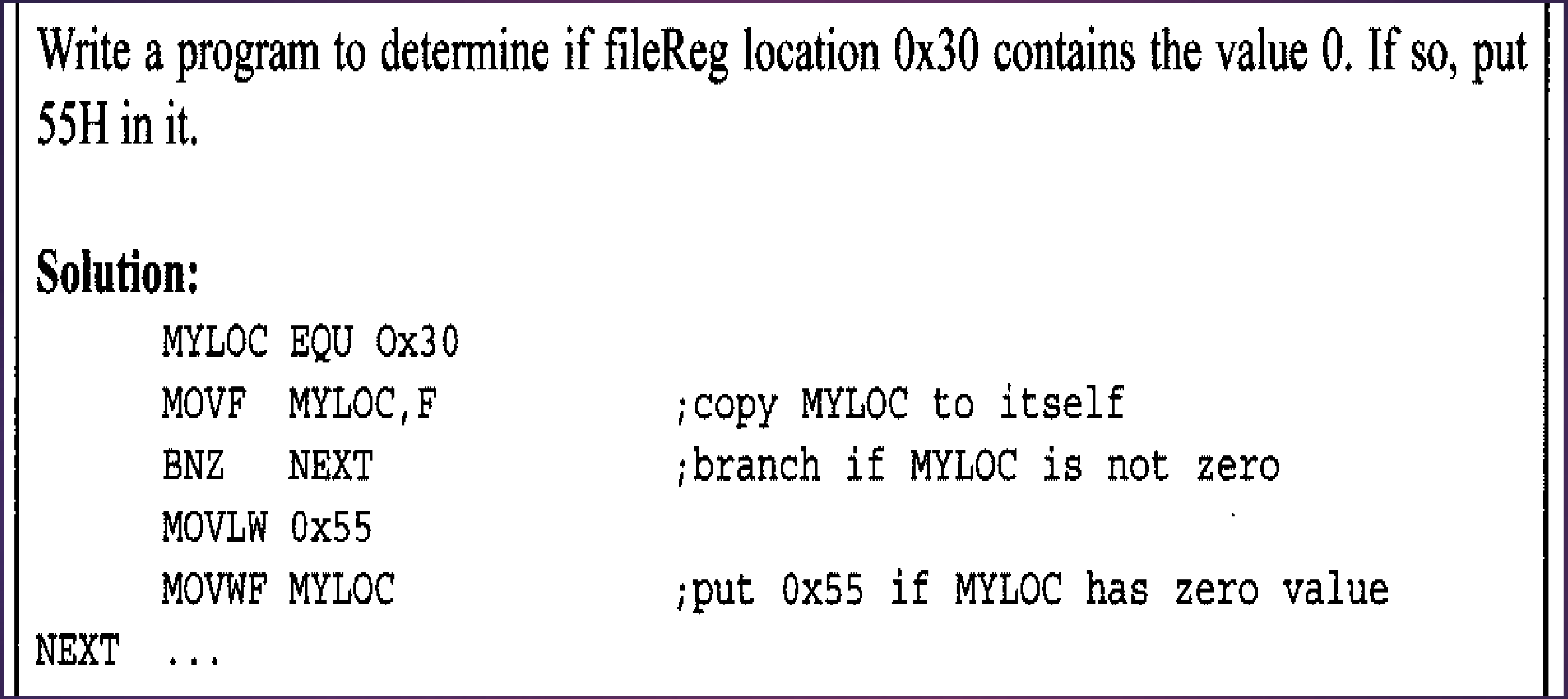
- In this instruction, the Z flag is poled. If it is high, it jumps to the target addres.



```
OVER   MOVF  PORTB,W          ;read Port B and put it in WREG
       JZ  OVER               ;jump if WREG is zero
```

NOTE:

1. BZ instruction can be used to see whether any fileReg or WREG is zero.
2. We don't have to perform an arithmetic operation such as decrement to use BZ instruction.



Write a program to determine if fileReg location 0x30 contains the value 0. If so, put 55H in it.

Solution:

```
      MYLOC EQU 0x30
      MOVF  MYLOC,F          ;copy MYLOC to itself
      BNZ   NEXT             ;branch if MYLOC is not zero
      MOVLW 0x55
      MOVWF MYLOC            ;put 0x55 if MYLOC has zero value
NEXT  ...
```

Why to copy itself??

- To update the STATUS flags (Z, N, etc.)
- The instruction does not change the value in the register.
- But it refreshes the Zero flag (Z) based on that register's contents.
- This allows you to test if the register is 0 or non-zero.
- If you used MOVF fileReg, W, you'd lose the old contents of WREG.
- By using MOVF fileReg, F, you leave WREG untouched and only update flags.

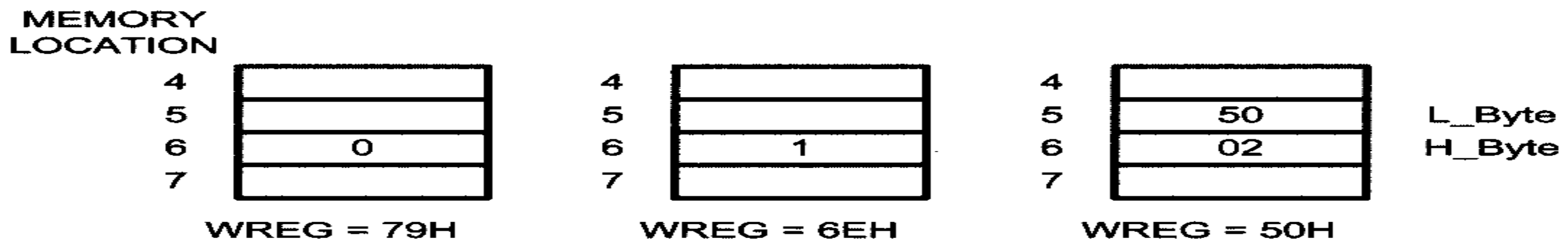
BNC (branch if no carry, if C=0)

- This instruction uses the C flag in status register to make the decision whether to jump.

Find the sum of the values 79H, F5H, and E2H. Put the sum in fileReg locations 5 (low byte) and 6 (high byte).

Solution:

```
L_Byte  EQU  0x5                 ;assign RAM loc 5 to L_byte of sum
H_Byte  EQU  0x6                 ;assign RAM loc 6 to H_byte of sum

       ORG  0h
       MOVLW  0x0                ;clear WREG(WREG = 0)
       MOVWF  H_Byte             ;H_Byte = 0
       ADDLW  0x79               ;WREG = 0 + 79H = 79H, C = 0
       BNC    N_1                ;if C = 0, add next number
       INCF   H_Byte,F           ;C = 1, increment (now H_Byte = 0)
N_1    ADDLW  0xF5               ;WREG = 79 + F5 = 6E and C = 1
       BNC    N_2                ;branch if CY = 0
       INCF   H_Byte,F           ;C = 1, increment (now H_Byte = 1)
N_2    ADDLW  0xE2               ;WREG = 6E + E2 = 50 and C = 1
       BNC    OVER               ;branch if C = 0
       INCF   H_Byte,F           ;C = 1, increment (now H_Byte = 2)
OVER   MOVWF  L_Byte             ;now L_Byte = 50H, and H_Byte = 02
       END
```



Unconditional branch Instruction

- The unconditional branch is a jump in which control is tranferred unconditionally to the target location.
- In PIC18, we have two unconditional branches:

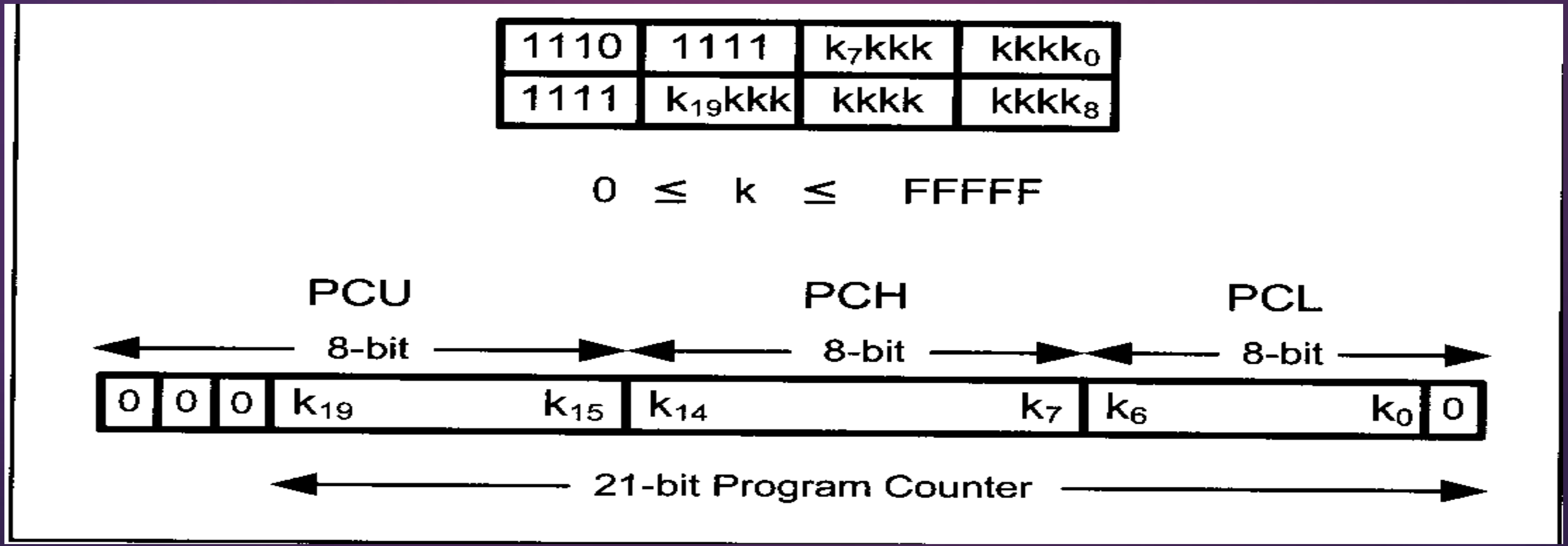
1. GOTO (go to)
2. BRA (branch)

Deciding which one to use depends on the target address.

NOTE: All conditional branches are short jumps, meaning that the target address must be within 256 bytes of the contents of the program cpunter.

GOTO

- GOTO is a **long jump** that can go to any memory location in the 2M address space of the PIC18.
- It is a 4 byte (32 bit) instruction in which 12 bits are used for the opcode, and the other 20 bits represent the 20-bit address of the target location.
- 20 bit address allow a jump to 1M of memory locations from 00000 to 1FFFFH, instead of 2M.

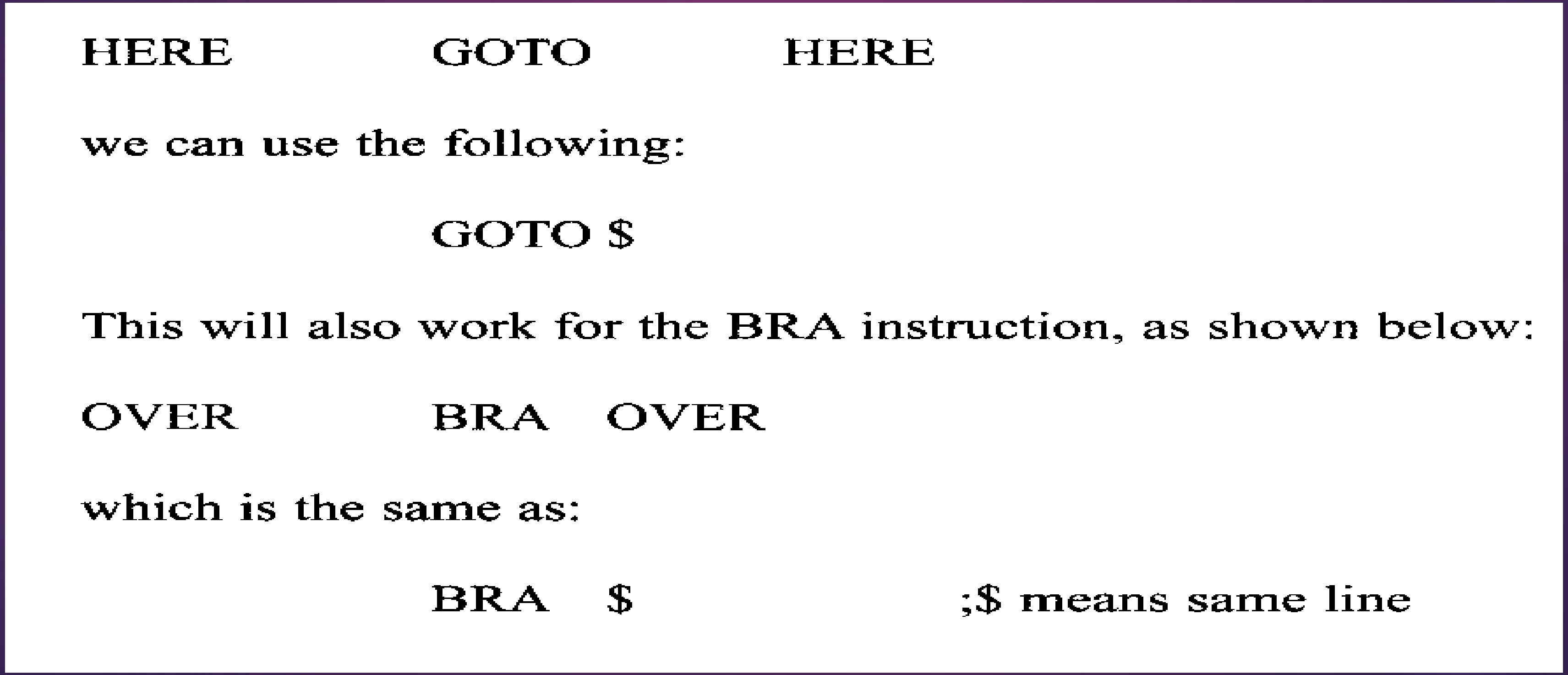


BRA (branch)

- Remember that the PC in PIC18 is 21 bit, giving a ROM address space of 2M, not all PIC18 family members have that much on-chip program ROM.
- For this reason, there is also a BRA instruction, which is 2 byte instruction as opposed to the 4 byte GOTO instruction.
- This can save some bytes of memory in many applications where ROM memory space is in short supply.
- In 2 byte (16 bits) instruction, the first 5 bits are the opcode and the rest 11 bits is the address of the the target location.
- The relative address range is 000-FFFH.

GOTO to itself using $ sign

- We can use the GOTO (jump) to itself in order to keep the microcontroller busy.



```
HERE        GOTO        HERE
```

we can use the following:

```
            GOTO $
```

This will also work for the BRA instruction, as shown below:

```
OVER        BRA   OVER
```

which is the same as:

```
            BRA   $                ;$ means same line
```

Call Instructions

- Another control transfer instruction is the CALL instruction, which is used to call a subroutine.
- Subroutines are often used to perform tasks that need to be performed frequently.
- This makes a program more structured in addition to saving memory space.
- In PIC18, there are two instructions for call:

1. CALL (long call)
2. RCALL (relative call)

CALL

- It is 4 byte instruction, 12 bits are used for the opcode and the other 20 bits, A1-A21, are used for the address of the target subroutine.
- Just as GOTO instruction, it can go to anywhere within 00000-1FFFFH for PIC18.
- To make sure that the PIC knows where to come back to after execution of the called subroutine, the microcontroller automatically saves on the stack the address of the instruction immediately below the CALL.
- When a subroutine is called, control is tranferred to that subroutine, and the processor saves the PC of the next on the stack and begines to fetch the instructions from the new location.

CALL and RETURN instructions

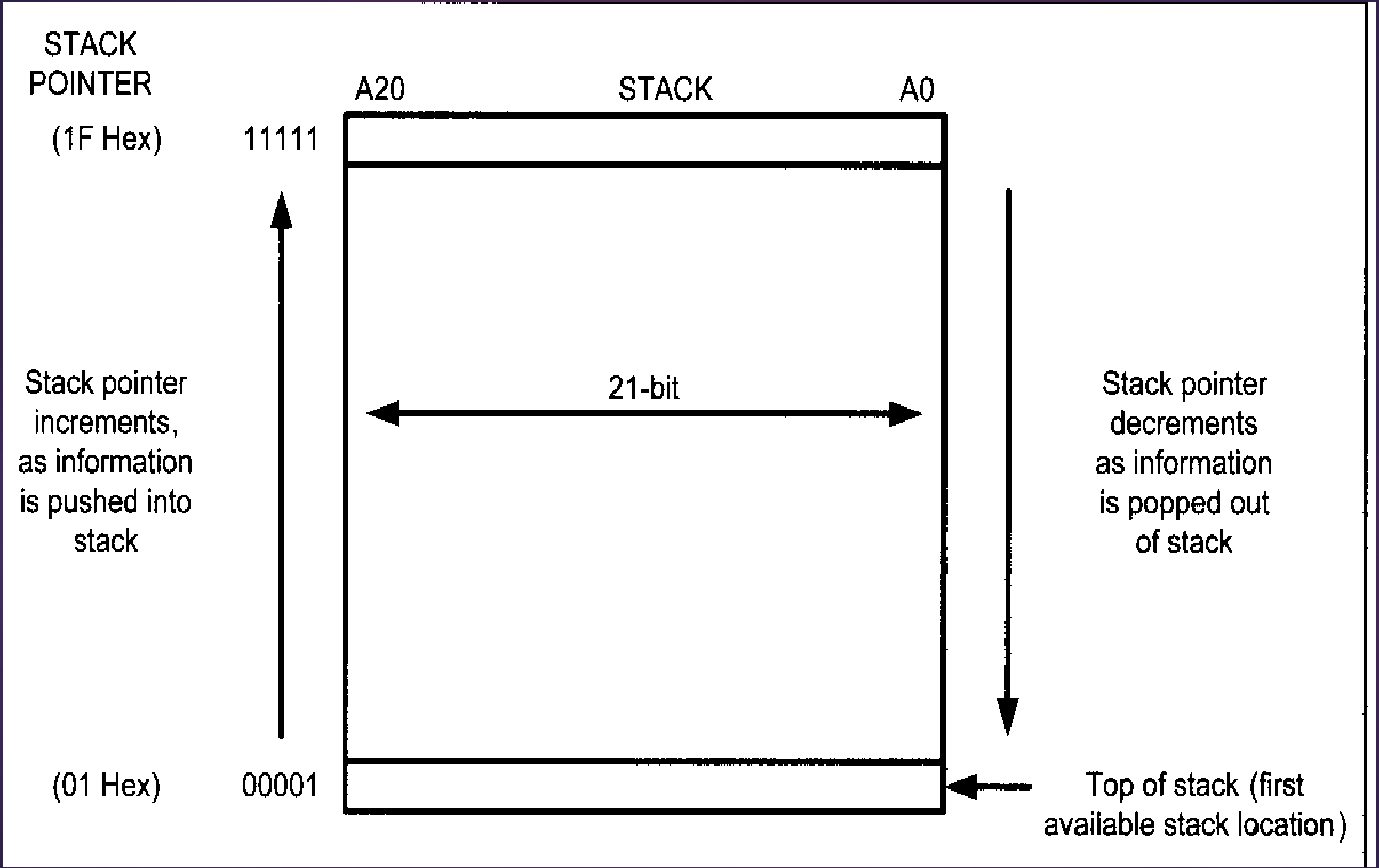
- After finishing execution of the subroutine, the instruction RETURN transfers control back to the caller.
- Every subroutine needs RETURN as the last instruction.
- Use GOTO for unconditional jumps.
- Use CALL when you need to execute a subroutine and later return to where you left off.

Stack and stack pointer in PIC18

- The stack is read/write memory (RAM) used by the processor to store some very critical information temporarily.
- This information usually is an address, but it could be data as well.
- The stack in PIC18 is 21-bit wide because the PC is 21-bit.
- Just like PC, it can take values of 00000 to 1FFFFFH.
- If the stack is RAM, there must be a register inside the processor to point to it.
- The register used to access the stack is called the SP (stack pointer) register.

Stack Pointer

- PIC18 has a 5-bit stack pointer, which can take values of 00 to1FH.
- That gives us a total of 32 locations where each location is 21 bits wide.
- When the PIC18 is powered on, SP register contains value 0.
- The storing of processor information such as the PC on the stack is called **PUSH.**
- Loading the contents of the stack back into the processor register is called a **POP.**

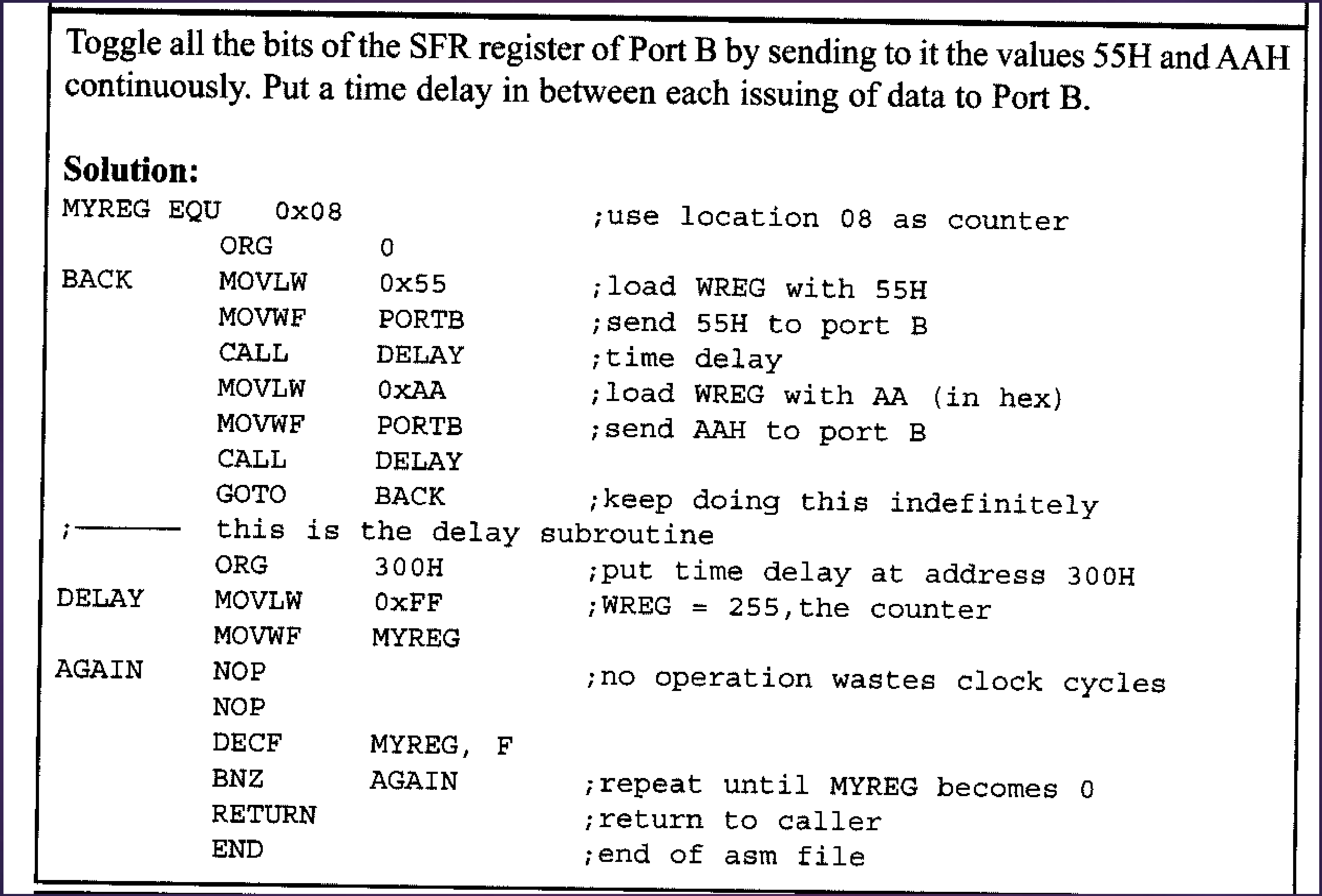

STACK POINTER
A20
STACK
A0
(1F Hex)
11111
21-bit
Stack pointer increments, as information is pushed into stack
Stack pointer decrements as information is popped out of stack
(01 Hex)
00001
Top of stack (first available stack location)

Pushing and Popping the Stack

- In PIC, the SP is pointing to the last used location of the stack.
- The last used location of the stack is called Top of the Stack (TOS).
- As data is pushed onto the stack, SP incremented.
- Popping is the reverse process of Pushing.
- When the RETURN instruction at the end of subroutine is executed, the top location of the stack is copied back to the PC and the SP is decremented once.

Stack is a LIFO (Last In First Out) memory

- In PIC, processor uses the stack to save the address of the instruction just below the CALL instruction.
- This is how processor knows where to resume when it returns from the called subroutine.
- Refer to the next example to examine the contents of stack and stack pointer.



Toggle all the bits of the SFR register of Port B by sending to it the values 55H and AAH continuously. Put a time delay in between each issuing of data to Port B.

Solution:

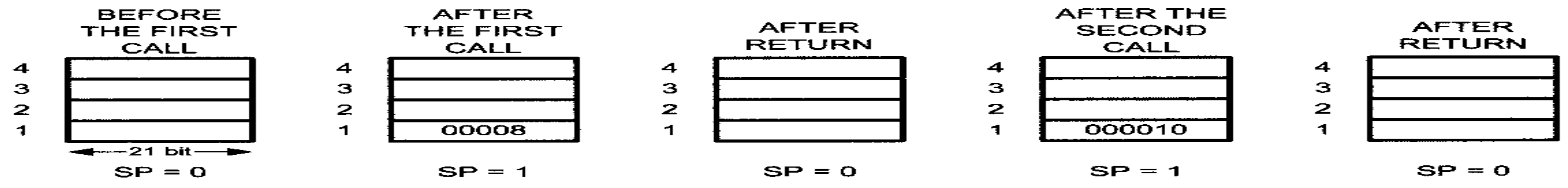
```
MYREG EQU   0x08                  ;use location 08 as counter
        ORG       0
BACK    MOVLW     0x55            ;load WREG with 55H
        MOVWF     PORTB           ;send 55H to port B
        CALL      DELAY           ;time delay
        MOVLW     0xAA            ;load WREG with AA (in hex)
        MOVWF     PORTB           ;send AAH to port B
        CALL      DELAY
        GOTO      BACK            ;keep doing this indefinitely
;------ this is the delay subroutine
        ORG       300H            ;put time delay at address 300H
DELAY   MOVLW     0xFF            ;WREG = 255,the counter
        MOVWF     MYREG
AGAIN   NOP                       ;no operation wastes clock cycles
        NOP
        DECF      MYREG, F
        BNZ       AGAIN           ;repeat until MYREG becomes 0
        RETURN                    ;return to caller
        END                       ;end of asm file
```

Analyze the stack for the CALL instructions in the following program.

Solution:

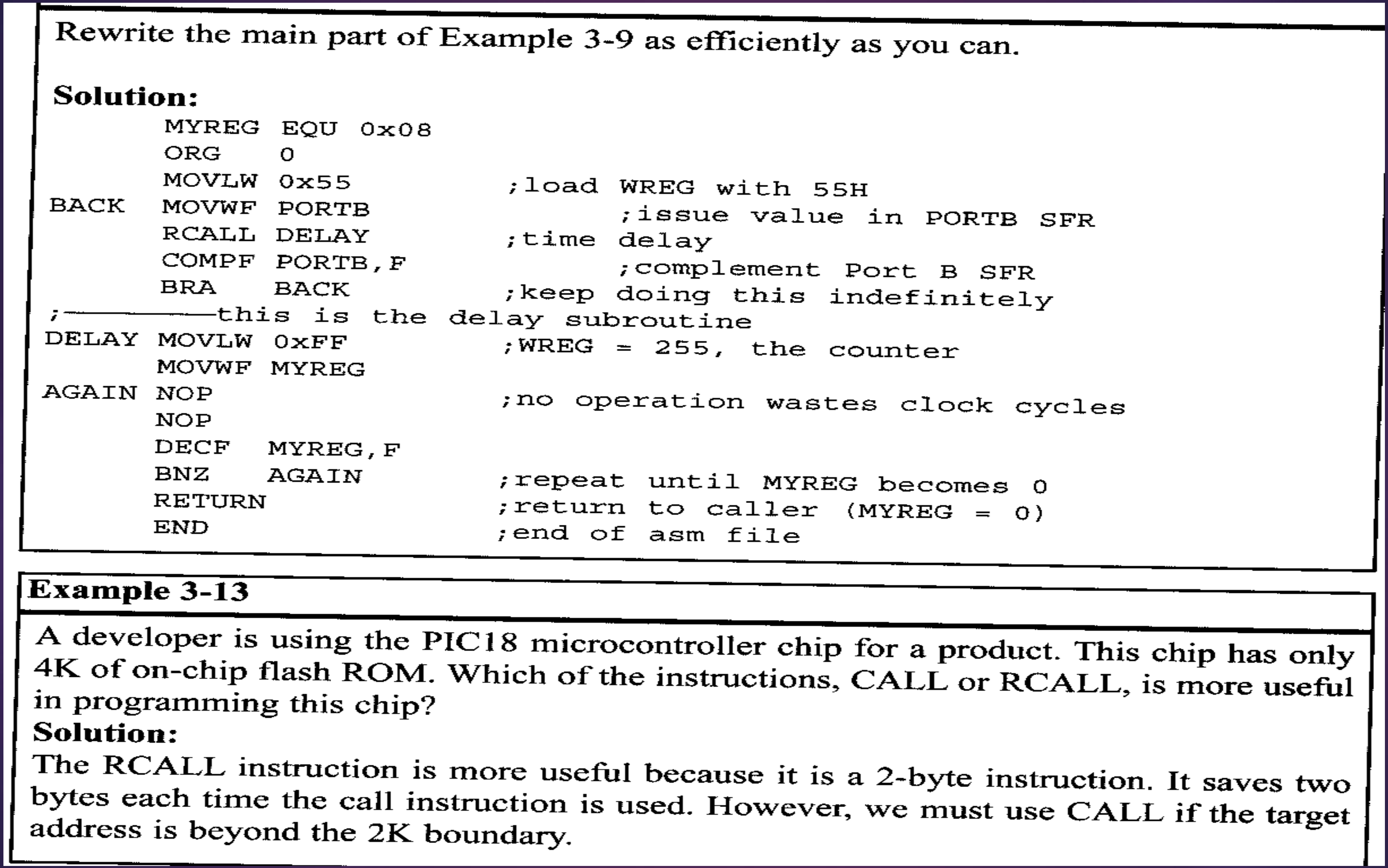
When the first CALL is executed, the address of the instruction "MOVLW 0xAA" is saved (pushed) on the stack. The last instruction of the called subroutine must be a RETURN instruction, which directs the CPU to pop the contents of the top location of the stack into the PC and resume executing at address 000007. The diagrams show the stack frame after the CALL and RETURN instructions.

```
LOC   OBJECT CODE       LINE SOURCE TEXT
  VALUE

                       00001 #DEFINE PORTB 0xF81
  00000008             00002 MYREG EQU 0x08         ;use location 08 as counter
                       00003
                       00004
  000000               00005            ORG    0
000000 0E55            00006  BACK      MOVLW  0x55   ;load WREG with 55H
000002 6E81            00007            MOVWF  PORTB  ;send 55H to port B
000004 EC80 F001       00008            CALL   DELAY  ;time delay
000008 0EAA            00009            MOVLW  0xAA   ;load WREG with AA (in hex)
00000A 6E81            00010            MOVWF  PORTB  ;send AAH to port B
00000C EC80 F001       00011            CALL   DELAY
000010 EF00 F000       00012            GOTO   BACK   ;keep doing this indefinitely
                       00013
                       00014 ;―――――――― this is the delay subroutine
                       00015
  000300               00016            ORG    300H   ;put delay at address 300H
000300 0EFF            00017  DELAY     MOVLW  0xFF   ;WREG = 255,the counter
000302 6E08            00018            MOVWF  MYREG
000304 0000            00019  AGAIN     NOP           ;no op wastes clock cycles
000306 0000            00020            NOP
000308 0608            00021            DECF   MYREG, F
00030A E1FC            00022            BNZ    AGAIN  ;repeat until MYREG becomes 0
00030C 0012            00023            RETURN        ;return to caller
                       00024            END           ;end of asm file
```



RCALL

- It is a 2-byte instruction. Thus the target address of the subroutine must be wihtin 2K because only 11 bits of the 2 bytes are used for the address.
- The only difference between CALL and RCALL is that the target address for CALL can be anywhere within the 2M address space of the PIC18 while the traget address of RCALL must be within 2K range.



Rewrite the main part of Example 3-9 as efficiently as you can.

Solution:

```
        MYREG  EQU  0x08
        ORG    0
        MOVLW  0x55              ;load WREG with 55H
BACK    MOVWF  PORTB                   ;issue value in PORTB SFR
        RCALL  DELAY             ;time delay
        COMPF  PORTB,F                 ;complement Port B SFR
        BRA    BACK              ;keep doing this indefinitely
;————————this is the delay subroutine
DELAY   MOVLW  0xFF              ;WREG = 255, the counter
        MOVWF  MYREG
AGAIN   NOP                      ;no operation wastes clock cycles
        NOP
        DECF   MYREG,F
        BNZ    AGAIN             ;repeat until MYREG becomes 0
        RETURN                   ;return to caller (MYREG = 0)
        END                      ;end of asm file
```

Example 3-13

A developer is using the PIC18 microcontroller chip for a product. This chip has only 4K of on-chip flash ROM. Which of the instructions, CALL or RCALL, is more useful in programming this chip?

Solution:

The RCALL instruction is more useful because it is a 2-byte instruction. It saves two bytes each time the call instruction is used. However, we must use CALL if the target address is beyond the 2K boundary.