Chapter 7
Assembly Language Programming for PIC18F: Part 2

Chien-Chung Ho (何建忠)

PIC18F Jump/Branch instructions

- There is one unconditional JUMP such as GOTO k instruction, where ‘k’ is an address. Hence, the GOTO instruction uses “direct” or “absolute” addressing mode.
- There is also an unconditional branch such as BRA d instruction, where ‘d’ is a signed 11-bit offset (range -1024 decimal to +1023 decimal with 0 being positive). Hence, this instruction uses “relative” addressing mode.

PIC18F Jump/Branch instructions

- There are eight conditional branch conditions. They use “relative” addressing mode.
- Consider Bcc d instruction where ‘d’ is an 8-bit signed offset (range -128 decimal to +127 decimal with 0 being positive). Note that cc (condition code) in Bcc can be replaced by eight conditions providing eight instructions: BC, BNC, BZ, BNZ, BN, BNN, BOV, and BNOV. It should be mentioned that these instructions are applicable to signed numbers.

PIC18F Jump/Branch instructions

TABLE 7.1 PIC18F Jump/Branch instructions

Instruction	*Operation*
GOTO k	Unconditionally jumps to an address defined by the k. Uses direct or absolute mode.
Bcc d	If the condition cc is true, then (PC+2) + 2 x d → PC. The PC value is current instruction location plus 2. Displacement d is an 8-bit signed number.
BRA d	Branch always to (PC+2) + 2 x d, where PC value is current instruction location plus 2. d is a signed 11-bit number. This is an unconditional branch instruction with relative mode.
All instructions in the above except GOTO and BRA are executed in one cycle; GOTO and BRA are executed in two cycles. The size of each instruction except GOTO is one work; the size of GOTO is two words.	

PIC18F Jump/Branch instructions

- GOTO k instruction unconditionally jumps to a 21-bit address. (20-bit ‘k’) is loaded into the PC (bit 1 through bit 20) with the least significant bit (bit 0 of the PC) as 0. This will make the target address an even number.

PIC18F Jump/Branch instructions

- Bcc d instruction will branch if the condition cc is true, then the program. The 2's complement number '2 x d' is added to the PC. Since the PC will be incremented to fetch the next instruction, the new address will be PC + 2 + 2d. This instruction is then a two-cycle instruction. If the condition is false, then the next instruction is executed. Note that displacement 'd' is an 8-bit signed number.

PIC18F Jump/Branch instructions

- There are 8 conditions such as BC (Branch if Carry = 1), BNC (Branch if no Carry), BZ (Branch if result equals to zero, i.e., Z = 1), and BNZ (Branch if not equal, i.e., Z = 0), BN (Branch if negative i.e. N = 1), BNN (Branch if not negative i.e. N = 0), BOV (Branch if Overflow, i.e. OV = 1), BNOV (Branch if no Overflow, i.e. OV = 0).

PIC18F Jump/Branch instructions

```
                                    1:              #INCLUDE<P18F4321.INC>
                                    2:              ORG         0x00
0000  0E02  MOVLW  0x2          3:      BACK    MOVLW       0x02
0002  0802  SUBLW  0x2          4:              SUBLW       0x02
0004  E001  BZ     0x8          5:              BZ          DOWN
0006  0E04  MOVLW  0x4          6:              MOVLW       0x04
0008  0804  SUBLW  0x4          7:      DOWN    SUBLW       0x04
000A  E0FA  BZ  0               8:              BZ          BACK
000C  0003  SLEEP               9:              SLEEP
```

PIC18F Jump/Branch instructions

- The first branch instruction, BZ DOWN (line 5) at address 0x0004, has a machine code 0xE001. Upon execution of the instruction BZ (branch if Z-flag = 1), the PIC18F branches to label DOWN if Z = 1.
- The machine code 0xE001 means that the op-code for BZ is 0xE0 and the relative 8-bit signed offset value is 0x01 (+1). This is a positive value indicating a forward branch.

PIC18F Jump/Branch instructions

- Note that while executing BZ DOWN at address 0x0004, the PC points to address 0x0006 since the program counter is incremented by 2. This means that the program counter contains 0x0008. The offset 0x01 is multiplied by 2 and added to address 0x0006 to find the target branch address where the program will jump if Z = 1. The branch address can be calculated as follows:

```
  0x0006 =  0000 0000  0000 0110
+ 0x0002 =  0000 0000  0000 0010  (0x01 is multiplied by 2 , and sign-extended to 16 bits)
-----------------------------------------
            0000 0000  0000  1000 = 0x0008
```

PIC18F Jump/Branch instructions

- Consider the second branch instruction, BZ BACK (line 8). Upon execution of BZ BACK, the PIC18F branches to label BACK if Z = 1; otherwise, the PIC18F executes the next instruction. The machine code of this instruction at address 0x000A is 0xE0FA, where 0xE0 is the op-code and 0xFA is the signed 8-bit offset value. The offset is represented as an 8-bit two's complement number. Since 0xFA is a negative number (-6), this is a backward jump.

PIC18F Jump/Branch instructions

- Note that while executing BZ BACK at address 0x000A, the PC points to address 0x000C since the program counter is incremented by 2. This means that the program counter contains 0x000C. The offset -6 is multiplied by 2, and then added to 0x000C to find the address value where the program will branch if Z = 1. The branch address is calculated as follows:

```
  0x000C = 0000 0000 0000 1100
+0xFFF4 = 1111 1111 1111 0100   (0xFA is multiplied by 2, and then sign-extended
                                 to 16 bits)
-----------------------------------------------
        1 0000 0000 0000 0000 = 0x0000
        ↗
Ignore final carry
```

PIC18F Jump/Branch instructions

- In order to add a 16-bit signed number with an 8-bit signed number, the 8-bit signed number must first be sign-extended to 16 bits. The two 16-bit numbers can then be added. Any carry resulting from the addition must be discarded.

PIC18F Jump/Branch instructions

- BRA d The BRA (Branch Always) instruction uses the relative addressing mode. The BRA d instruction unconditionally branches to (PC + 2 + 2 x d) where offset ‘d’ is a signed 11-bit number specifying a byte range from -1024 decimal to + 1023 decimal with 0 being positive.
- Notice that all conditional branch instructions such as “BZ d” also uses relative mode but uses ‘d’ as a signed 8-bit offset.

PIC18F Test, Compare, and Skip instr.

TABLE 7.2 PIC18F Test, Compare, and Skip instructions

Instruction	*Operation*
BTFSC F, b, a	Bit Test File register, Skip if Clear. If bit 'b' in register 'F' is '0', then the next instruction is skipped. If bit 'b' is '1', then the next instruction is executed.
BTFSS F, b, a	Bit Test File, Skip if Set. If bit 'b' in register 'F' is '1', then the next instruction is skipped. If bit 'b' is '0', then the next instruction is executed.
CPFSEQ F, a	Compare F with W, Skip if F = W. Compares the contents of data memory location 'F' to the contents of W by performing an unsigned subtraction. If 'F' = W, then the next instruction is skipped; else, the next instruction is executed.
CPFSGT F, a	Compare F with W, Skip if F > W. Compares the contents of data memory location 'F' to the contents of the W by performing an unsigned subtraction. If the contents of 'F' are greater than the contents of W, then the next instruction is skipped; else, the next instruction is executed.
CPFSLT F, a	Compare F with W, Skip if f < W. Compares the contents of data memory location 'F' to the contents of W by performing an unsigned subtraction. If the contents of 'F' are less than the contents of W, then the next instruction is skipped; else, the next instruction is executed.

PIC18F Test, Compare, and Skip instr.

DECFSNZ F, d, a	Decrement F, Skip if Not 0. The contents of register 'F' are decremented by 1. If the result is not '0', then the next instruction is skipped; else, the next instruction is executed.
DECFSZ F, d, a	Decrement F, Skip if 0. The contents of register 'F' are decremented by 1. If the result is '0' ,then the next instruction is skipped; else, the next instruction is executed.
INCFSNZ F, d, a	Increment F, Skip if Not 0. The contents of register 'F' are incremented by 1. If the result is not '0', then the next instruction is skipped; else, the next instruction is executed.
INCFSZ F, d, a	Increment F, Skip if 0. The contents of register 'F' are incremented by 1. If the result is '0', then the next instruction is skipped; else, the next instruction is executed.
TSTFSZ F, a	Test F, Skip if 0. If 'F' = 0, then the next instruction is skipped; else, the next instruction is executed.

- All instructions in the above are executed in one to two cycles. No flags are affected. The size of each instruction is one word.
- a = 0 means that the data register is located in the access bank while a = 1 means that the contents of BSR specify the address of the bank.
- For destination: d = 0 means that the destination is WREG while d = 1 means that the destination is file register.

PIC18F Test, Compare, and Skip instr.

- BTFSC F, b, a (BTFSC 0x40, 5) instruction tests the specified bit ‘b’ in the file register ‘F’, and skips the next instruction if the bit ‘b’ is 0. On the other hand, if bit ‘b’ is 1, the PIC18F executes the next instruction.
- Hence, the GOTO or BRA instruction is typically used after the BTFSC instruction. The BTFSC instruction is useful for conditional (polled) I/O.

PIC18F Test, Compare, and Skip instr.

- Consider BTFSC 0x40, 5.
- Prior to execution of BTFSC 0x40, 5: [0x40] = F1H.
- After execution of BTFSC 0x40, 5: Since bit 5 of [0x40] is 1, the BTFSC executes the next instruction.

PIC18F Test, Compare, and Skip instr.

- The BTFSC instruction can be used to write the PIC18F assembly language instruction sequence for the following C segment:

```
if (x<0)
        y++ ;
else
        y--;
```

The PIC18F assembly language program can be written as follows:

```
        BTFSC   X, 7        ; Check sign bit (bit 7) of [X]. If  negative, increment [Y]
        BRA     NEG         ; If  [X] is positive, decrement [Y]
        DECF    Y           ; Increment [Y] if  [X] is negative
        BRA     NEXT
NEG     INCF    Y           ; Decrement [Y] if [X] is positive
NEXT            ----------  ;  Next instruction
```

PIC18F Test, Compare, and Skip instr.

- BTFSS F, b, a (BTFSS 0x70, 0) instruction tests the specified bit 'b' in the file register 'F', and skips the next instruction if the bit 'b' is '1'. On the other hand, if bit 'b' is '0', the PIC18F executes the next instruction. Like the BTFSC instruction, the GOTO or BRA instruction is typically used after the BTFSS instruction. The BTFSS instruction can be used for conditional (polled) I/O. This topic is discussed later.

  Next, as an example, consider BTFSS 0x70, 0.

  Prior to execution of BTFSS 0x70, 0 : [0x70] = 0xF1.

  After execution of BTFSS 0x70, 0: Since bit 0 of [0x70] is 1, the BTFSS skips the next instruction.

PIC18F Test, Compare, and Skip instr.

The same example for BTFSC, described in the last section, can be used to illustrate the BTFSS instruction. The assembly language program is written using the BTFSS instruction for the following C segment:

```
if (x<0)
y++ ;
else
y--;
```

The PIC18F assembly language instruction sequence using the BTFSS is provided below:

```
        BTFSS   X, 7      ; check sign bit (bit 7) of [X]. If  negative, increment [Y]
        BRA     POS       ; if  [X] is positive, decrement [Y]
        INCF    Y         ; increment [Y] if  [X] is negative
        BRA     NEXT
POS     DECF    Y         ; decrement [Y] if [X] is positive
NEXT    ---------         ; Next instruction
```

PIC18F Test, Compare, and Skip instr.

- PIC18F COMPARE instructions The CPFSEQ F, a instruction compares [F] with [WREG] by performing an unsigned subtraction, (both numbers are considered unsigned, skips the next instruction if [F] = [WREG]; if [F] ≠ [WREG], then the PIC18F executes the following instruction. The CPFSGT F, a, on the other hand, compares [F] with [WREG] by performing an unsigned subtraction, skips the next instruction if [F] > [WREG]; if [F] ≤ [WREG], then the PIC18F executes the next instruction. The CPFSLT F, a instruction compares [F] with [WREG] by performing an unsigned subtraction, skips the next instruction if [F] < [WREG]; if [F] ≥ [WREG], then the PIC18F executes the next instruction. Note that in all three cases, [F] and [WREG] are considered as 8-bit unsigned (positive) numbers. The GOTO or BRA instruction is typically used after each of these COMPARE instructions. These instructions do not provide any result of subtraction and also, they do not affect any status flags.

PIC18F Test, Compare, and Skip instr.

- Suppose it is desired to find the number of matches for an 8-bit unsigned number in data register 0x80 with a data array (stored from low to high memory) of 50 bytes in memory pointed to by 0x50. Assume that data are already stored in memory. The following instruction sequence with CPFSEQ can be used :

PIC18F Test, Compare, and Skip instr.

```
        CLRF    0x40        ; Clear  0x40 to 0. Register 0x40 will  hold
                            ;  the  number of matches
        MOVLW   D'50'       ;  Move 50 into WREG
        MOVWF   0x20        ;  Initialize 0x20 with the array count, 50
        LFSR    0, 0x50     ; Initialize indirect pointer FSR0 with 0x50
        MOVF    0x80, W     ;  Move [0x80] to WREG
BACK    CPFSEQ  POSTINC0    ; Compare the number to be matched with [WREG]
        BRA     NOMATCH
        INCF    0x40        ; If there is a match, increment [0x40] by 1
NOMATCH DECF    0x20        ; Decrement [0x20] by 1
        BNZ     BACK        ; Go to BACK if Z is not 0
        ----------          ; Next instruction
```

Note that, in the above, CPFSEQ rather than SUBWF is used. This is because we are not interested in the subtraction result. Rather, we are interested in the number of matches. If SUBWF is used, one needs to load the number to be matched or the data byte from the array after each SUBWF; the subtraction result would erase the data Hence, CPFSEQ instead of SUBWF is ideal for the above example.

PIC18F Test, Compare, and Skip instr.

- DECFSNZ F, d, a (DECFSNZ 0x30, W or DECFSNZ 0x30, F) instruction decrements [F] by 1, and If the result is not ‘0’, skips the instruction; else, the next instruction is executed. The DECFSZ decrements [F] by 1, and if the result is ‘0’, it skips the instruction; else, the next instruction is executed.
- Both instructions can be used to execute a certain loop ‘n’ times, where ‘n’ is an 8-bit number. This is another way of executing a loop without using the conditional branch instructions.

PIC18F Test, Compare, and Skip instr.

- For example, in order to execute a loop to obtain the 8-bit SUM (10 x A) by repeated addition, assume that the 8-bit unsigned number, ‘A’, is stored in register 0x70, and the SUM will be stored in 0x50. The following PIC18F instruction sequence using DECFSNZ will accomplish this:

PIC18F Test, Compare, and Skip instr.

```
        CLRF     0x50      ; Clear register 0x50 to 0 for SUM
        MOVLW    D'10'     ; Move 10 to WREG
        MOVWF    0x60      ; Initialize register 0x60 with 10
        MOVF     0x70,W    ; Move 'A' into WREG
REPEAT  ADDWF    0x50, F   ; Add 'A' 10 times, store result in 0x50
        DECFSNZ  0x60, F   ; Decrement counter, skip if reg 0x60 not 0
        GOTO     NEXT
        GOTO     REPEAT    ; Repeat addition until counter 0x60 is 0
NEXT    ----------         ; Next instruction
```

Using the DECFSZ , the above program to compute (10 x A) can be written as follows:

```
        CLRF     0x50      ; Clear register 0x50 to 0 for SUM
        MOVLW    D'10'     ; Move 10 to WREG
        MOVWF    0x60      ; Initialize register 0x60 with 10
        MOVF     0x70,W    ; Move 'A' into WREG
REPEAT  ADDWF    0x50, F   ; Add 'A' 10 times, store result in 0x50
        DECFSZ   0x60, F   ; Decrement counter, skip if 0
        GOTO     REPEAT    ; Repeat addition until counter 0x60 is 0
        ---------          ; Next instruction
```

PIC18F Test, Compare, and Skip instr.

- INCFSNZ F, d, a (INCFSNZ or INCFSZ 0x40, W or 0x40, F) increments the contents of register ‘F’ by one, and skips the next instruction if the result is not 0; else, the next instruction is executed.
- The INCFSZ F, d, a increments the contents of register ‘F’ by one, and skips the next instruction if the result is 0; else, the next instruction is executed.

PIC18F Test, Compare, and Skip instr.

- Both instructions can be used to execute a certain loop ‘n’ times, where ‘n’ is an 8-bit number. This is another way of executing a loop without using the conditional branch instructions.
- Consider the same examples using the DECFSNZ and DECFSZ instructions. As before, a loop will be executed to obtain the 8-bit SUM (10 x A) by repeated addition, assuming that the 8-bit unsigned number, ‘A’, is stored in register 0x70, and the SUM will be stored in 0x50.

PIC18F Test, Compare, and Skip instr.

The following PIC18F instruction sequence using INCFSNZ will accomplish this:

```
        CLRF    0x50      ; Clear register 0x50 to 0 for SUM
        MOVLW   0xF6      ; Move -10 to WREG
        MOVWF   0x60      ; Initialize register 0x60 with -10
        MOVF    0x70,W    ; Move 'A' into WREG
REPEAT  ADDWF   0x50, F   ; Add 'A' 10 times, store result in 0x50
        INCFSNZ 0x60, F   ; Increment counter, skip if reg 0x60 not 0
        GOTO    NEXT
        GOTO    REPEAT    ; Repeat addition until counter 0x60 is 0
NEXT    ----------        ; Next instruction
```

PIC18F Test, Compare, and Skip instr.

Using the INCFSZ , the above program to compute (10 x A) can be written as follows:

```
CLRF    0x50      ; Clear register 0x50 to 0 for SUM
MOVLW   0xF6      ; Move -10 to WREG
MOVWF   0x60      ; Initialize register 0x60 with -10
MOVF    0x70,W    ; Move 'A' into WREG
REPEAT  ADDWF   0x50, F   ; Add 'A' 10 times, store result in 0x50
        INCFSZ  0x60, F   ; Increment counter, skip if counter is 0
        GOTO    REPEAT    ; Repeat addition until counter 0x60 is 0
        --------          ; Next instruction
```

PIC18F Test, Compare, and Skip instr.

- TSTFSZ F, a (TSTFSZ 0x60) checks if [F] = 0, and skips the next instruction if it is zero (Z = 1); otherwise (Z = 0), the next instruction is executed.
- The TSTFSZ instruction can be used to check the contents of a register for 0 without using the conditional branch instruction. For example, a typical decrementing counter can be implemented using the conditional branch instruction such as BNZ as follows:

PIC18F Test, Compare, and Skip instr.

```
COUNTER   EQU     0x40
          MOVLW   D'50'       ; Initialize loop counter with 50
          MOVWF   COUNTER
LOOP      DECF    COUNTER     ; Decrement COUNTER by 1
          BNZ     LOOP
                              ; Branch  if [COUNTER]  is not 0
          -------             ; Next instruction
```

The above loop can be implemented using the TSTFSZ instruction as follows:

```
COUNTER   EQU     0x40
          MOVLW   D'50'       ; Initialize loop counter with 50
          MOVWF   COUNTER
LOOP      DECF    COUNTER     ; Decrement COUNTER by 1
          TSTFSZ  COUNTER     ; Test COUNTER  for 0 and if  not 0,
          GOTO    LOOP        ; go to LOOP. If [COUNTER] is  0, skip.
          -------             ; Next instruction
```

PIC18F Test, Compare, and Skip instr.

Example 7.1 Write a PIC18F assembly language program at 0x100 to subtract two 32-bit numbers as follows:

```
          [0x43] [0x42] [0x41] [0x40]
MINUS [0x75] [0x74] [0x73] [0x72]
      ------------------------------------
          [0x43] [0x42] [0x41] [0x40]
```

Assume data registers 0x40 through 0x43 contain the first 32-bit number while data registers 0x72 through 0x75 contain the second 32-bit number. Also, assume that data are already loaded into the data registers. Store the 32-bit result in data registers 0x40 (lowest byte) through 0x43 (highest byte). Use a loop. This is Example 6.12 using a loop.

PIC18F Test, Compare, and Skip instr.

Solution

```
        INCLUDE <P18F4321.INC>
        ORG     0x100
        MOVLW   4               ; Move WREG with 4
        MOVWF   0x50            ; Initialize 0x50 with loop count (4)
        LFSR 0, 0x0072          ; Initialize pointer FSR0 with 0x0072
        LFSR 1, 0x0040          ; Initialize pointer FSR1 with 0x0040
        ADDLW   0x00            ; Clear carry flag
START   MOVF    POSTINC0, W     ; Move byte into WREG and update pointer
        SUBWFB  POSTINC1, F     ; Subtract [WREG] and carry from byte,
                                ; store result in data register
        DECFSZ  0x50, F         ; Decrement counter 0x50 by 1 and skip if [0x50] = 0
        BNZ     START           ; Branch to START if Z is 0
        SLEEP                   ; Halt
        END
```

Note: In the above, DECFSZ is used instead of DECF. This is because DECF affects the carry flag while DECFSZ does not.

PIC18F Table Read/Write instructions

- The PIC18F program memory is 16 bits wide, while the PIC18F data memory space is 8 bits wide. Programs are stored in program memory with the data register contents defined using the assembler's DB directive. In order to execute a program requiring data, the data bytes stored in program memory using DB directive must be transferred to the specified data registers in data memory. Since the sizes of program memory and data memory are different, it would be difficult to accomplish this data transfer.

PIC18F Table Read/Write instructions

- Four table read and four table write instructions facilitate transferring data between these two memory spaces through an 8-bit TABLAT (register called Table latch), and a 21-bit TBLPTR (pointer register called the Table pointer).
- The TBLPTR includes three registers, namely TABLPTRU (bits 20 through 16), TBLPTRH (bits 15 through 8), and TBLPTRL (bits 7 through 0).

PIC18F Table Read/Write instructions

- Two operations to move bytes between the program memory and the data memory:
  - Table Read (TBLRD)
  - Table Write (TBLWT)
- Table read operation retrieves data from program memory and places it into the data memory. The Table read operation is accomplished with four addressing modes (register indirect, postincrement, postdecrement, and predecrement).

PIC18F Table Read/Write instructions

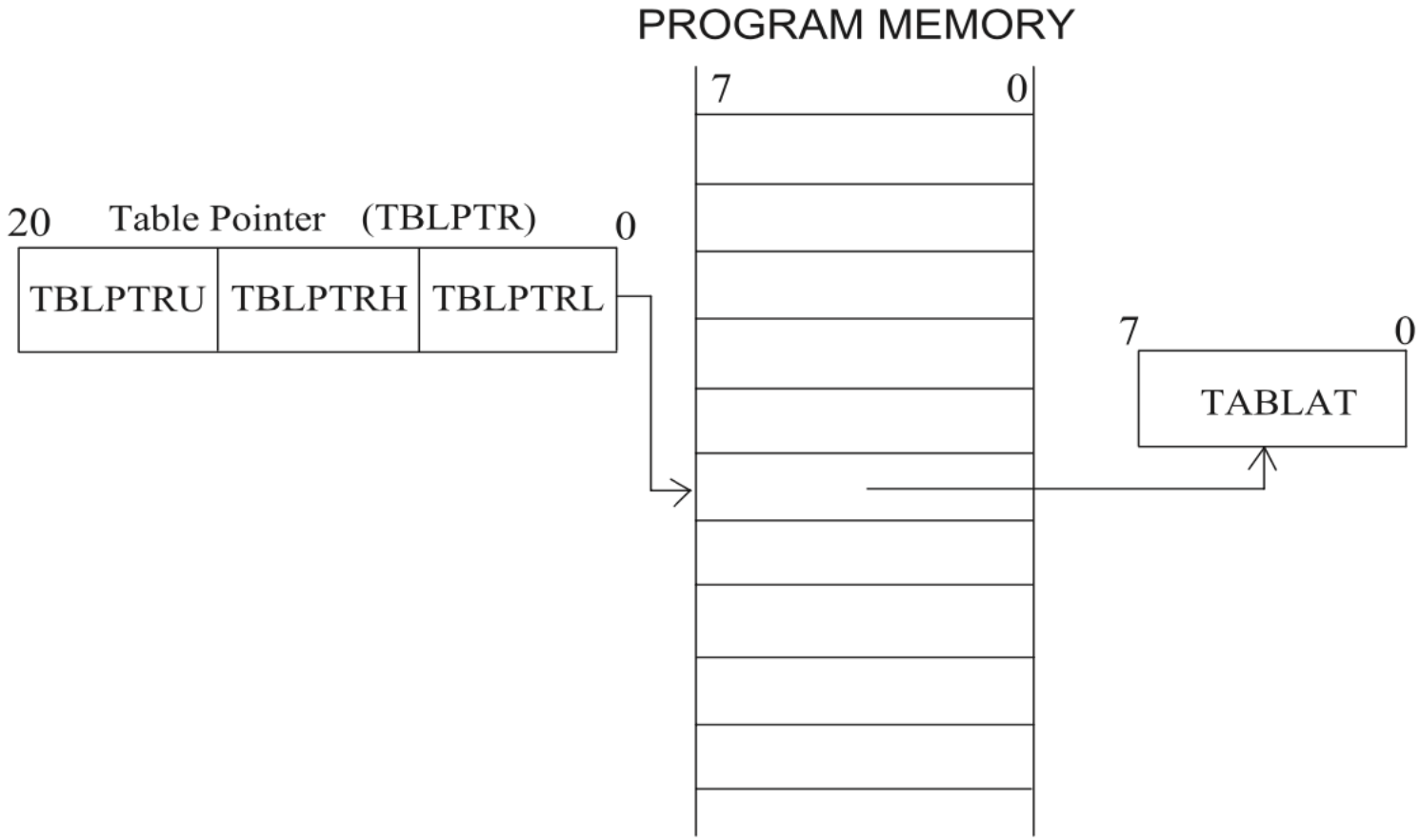


FIGURE 7.1 Table Read Operation (Instruction TBLRD*)

PIC18F Table Read/Write instructions

- Table write operation stores data from the data memory space into holding registers in program memory.
- Table operations work with data bytes. A table block containing data, rather than program instructions, is not required to be word aligned. Therefore, a table block can start and end at any byte address.

PIC18F Table Read/Write instructions

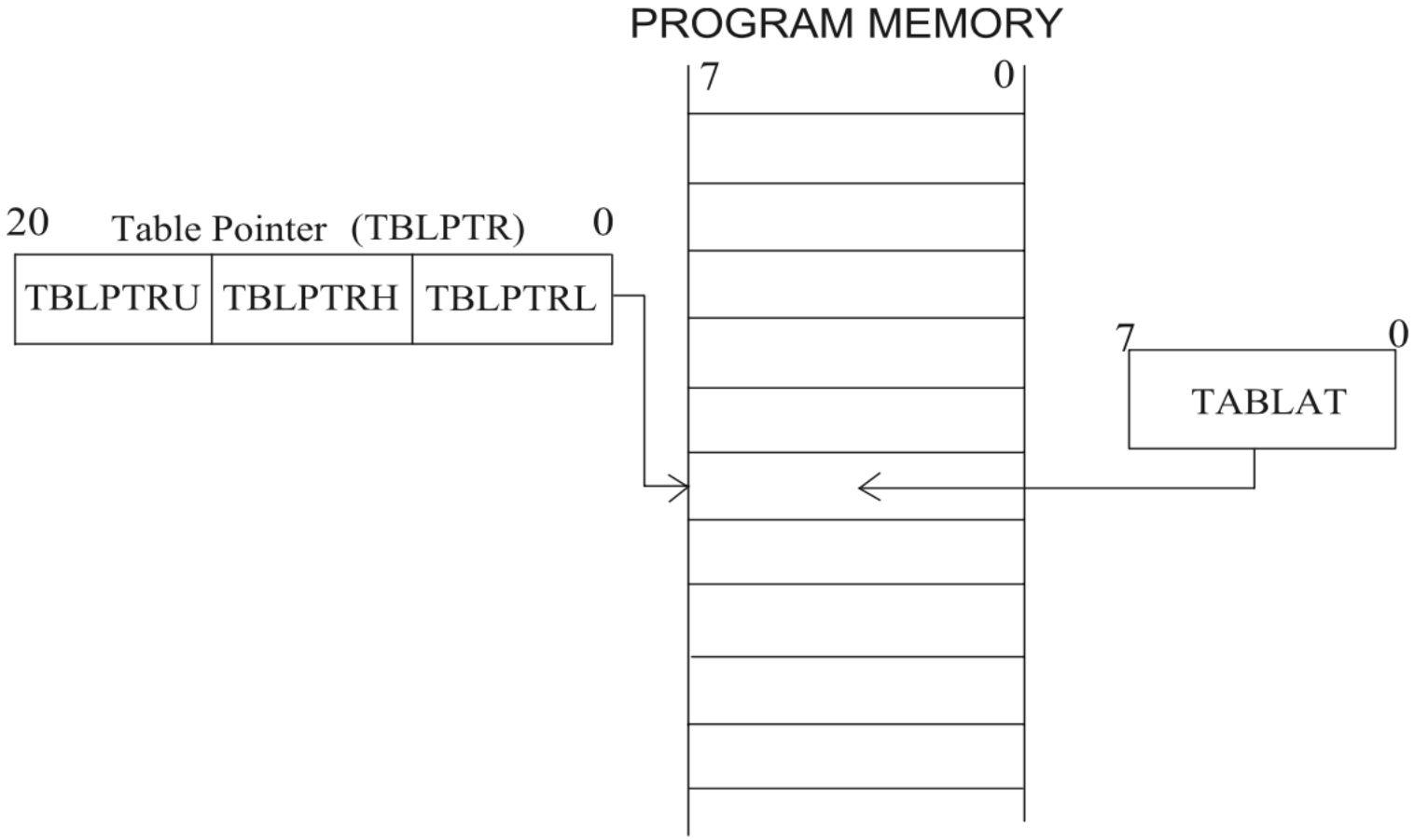


FIGURE 7.2 Table Write Operation (Instruction TBLWT*)

TABLE 7.3 PIC18F Table Read/Write instructions

TBLRD*	Move 8-bit data from program memory addressed by 21-bit TBLPTR into the 8-bit register TABLAT.
TABLRD*+	Move 8-bit data from program memory addressed by 21-bit TBLPTR into the 8-bit register TABLAT, and then increment TBLPTR by 1.
TBLRD*-	Move 8-bit data from program memory addressed by 21-bit TBLPTR into the 8-bit register TABLAT, and then decrement TBLPTR by 1.
TBLRD+*	Increment TBLPTR by 1 and then move 8-bit data from program memory addressed by 21-bit TBLPTR into the 8-bit register TABLAT. .
TPLWT*	Move 8-bit data from 8-bit register TABLAT into program memory addressed by 21-bit TBLPTR.
TBLWT*+	Move 8-bit data from 8-bit register TABLAT into program memory addressed by 21-bit TBLPTR , and then increment TBLPTR by 1.
TBLWT*-	Move 8-bit data from 8-bit register TABLAT into program memory addressed by 21-bit TBLPTR , and then decrement TBLPTR by 1.
TBLWT+*	Increment TBLPTR by 1 and then move 8-bit data from 8-bit register TABLAT into program memory addressed by 21-bit TBLPTR .

- All TBLRD and TBLWT instructions are executed in two cycles.
- The size of each instruction is one word.

PIC18F Table Read/Write instructions

- Consider TBLRD* instruction.

  Prior to execution of TBLRD*, [TBLPTR] = 0x02318, [TABLAT] = 0x24, and [0x002318] = 0xF2.

  After execution of TBLRD*, [TABLAT] = 0xF2, [TBLPTR] = 0x002318 (unchanged), and [0x002318] = 0xF2 (unchanged).

- Consider TBLRD*+ instruction.

  Prior to execution of TBLRD* +, [TBLPTR] = 0x002318, [TABLAT] = 0x24, and [0x002318] = 0xF2.

  After execution of TBLRD*+, [TABLAT] = 0xF2, [TBLPTR] = 0x002319, and [0x002318] = 0xF2 (unchanged).

PIC18F Table Read/Write instructions

- Consider TBLRD*- instruction.

  Prior to execution of TBLRD*-: [TBLPTR] = 0x000052, [TABLAT] = 0x24, [0x000052] = 0xF2
  After execution of TBLRD*-: [TABLAT] = 0xF2, [TBLPTR] = 0x000051, [0x000052] = 0xF2 (unchanged)
- Consider TBLRD+* instruction.

  Prior to execution of TBLRD+*: [TBLPTR] = 0x0030, [TABLAT] = 0x24, [0x000031] = 0xF2
  After execution of TBLRD+*: [TABLAT] = 0xF2, [TBLPTR] = 0x000031, [0x000031] = 0xF2 (unchanged)

PIC18F Table Read/Write instructions

- Consider TBLWT* instruction.

  Prior to execution of TBLWT*: [TBLPTR] = 0x000120, [TABLAT] = 0x24, [0x000120] = 0x14
  After execution of TBLWT*: [0x000120] = 0x24, [TABLAT] = 0x24 (unchanged), [TBLPTR] = 0x000120 (unchanged)

- Consider TBLWT*+ instruction.

  Prior to execution of TBLWT*+: [TBLPTR] = 0x000030, [TABLAT] = 0x1F, [0x000030] = 0x02
  After execution of TBLWT*+: [0x000030] = 0x1F, [TBLPTR] = 0x000031, [TABLAT] = 0x1F (unchanged)

PIC18F Table Read/Write instructions

- Consider TBLWT*- instruction.

  Prior to execution of TBLWT*-: [TBLPTR] = 0x000318, [TABLAT] = 0x24, [0x000318] = 0xF2
  After execution of TBLWT*-: [0x000318] = 0x24, [TBLPTR] = 0x000317, [TABLAT] = 0x24 (unchanged)

- Consider TBLWT+* instruction.

  Prior to execution of TBLWT+*: [TBLPTR] = 0x000118, [TABLAT] = 0x24, [0x000119] = 0xF2
  After execution of TBLWT+*: [0x000119] = 0x24, [TBLPTR] = 0x000119, [TABLAT] = 0x24 (unchanged)

PIC18F Table Read/Write instructions

- Example 7.2. Write a PIC18F assembly language program at address 0x100 to move the ASCII codes (30H through 39H) for BCD numbers 0 through 9 from program memory starting at address 0x200 (30H at address 0x200, 31H at 0x201, and so on) into data memory starting at address 0x40 (30H to be stored at address 0x40, 31H at 0x41, and so on).

PIC18F Table Read/Write instructions

Solution

```
            INCLUDE  <P18F4321.INC>
            ORG      0x100              ; #1 Starting address of program
COUNTER     EQU      0x20
            MOVLW    UPPER ADDR         ; #2 Move upper 5 bits (00H) of address
            MOVWF    TBLPTRU            ; #3 to TBLPTRU
            MOVLW    HIGH ADDR          ; #4 Move bits 15-8 (02H) of address
            MOVWF    TBLPTRH            ; #5 to  TBLPTRH
            MOVLW    LOW ADDR           ; #6 Move bits 7-0 (00H) of address
            MOVWF    TBLPTRL            ; #7 to TBLPTRL
            LFSR     0, 0x40            ; #8 Initialize FSR0 to 0x40 to be used as
                                        ; destination pointer in data memory
            MOVLW    D'10'              ; #9 Initialize COUNTER with 10
            MOVWF    COUNTER            ; #10 Move [WREG] into COUNTER
LOOP        TBLRD*+                     ; #11 Read data from program memory into
                                        ; TABLAT, increment TBLPTR by 1
            MOVF     TABLAT, W          ; #12 Move [TABLAT] into WREG
            MOVWF    POSTINC0           ; #13 Move W into data memory pointed to
                                        ; by FSR0, and then increment FSR0 by 1
            DECF     COUNTER, F         ; #14 Decrement COUNTER BY 1
            BNZ      LOOP               ; #15 Branch  if  Z = 0, else Stop
FINISH      BRA      FINISH             ; Stop
            ORG      0x200              ; Store ASCII codes in program memory
ADDR        DB       30H, 31H, 32H, 33H, 34H, 35H, 36H, 37H, 38H, 39H
            END
```

PIC18F Table Read/Write instructions

- In the above program, the # sign along with the line number is placed before the comment in order to identify the specific line for explanation. Note that the directive DB used with the label ADDR at the end of the program stores the ASCII codes for BCD starting address 0x200 in the program memory.

PIC18F Table Read/Write instructions

- Line #1 specifies the starting address of the program at 0x000200. Note that the programmer does not have to define the address, 0xFF5 for TABLAT. This is a predefined address (Special function register) by Microchip. The MPLAB assembler determines this internally.

PIC18F Table Read/Write instructions

- The TBLPTR is divided into three registers as TBLPTRU (predefined address 0xFF8), TBLPTRH (predefined address 0xFF7), and TBLPTRL (predefined address 0xFF6). The MPLAB assembler identifies TBLPTRU as UPPER, TBLPTRH as HIGH, and TBLPTRL as LOW.
- Line #’s 2 through 7 initialize TBLPTR with the 21-bit address 0x000200.

PIC18F Table Read/Write instructions

- Line #8 initializes data memory pointer FSR0 to 0x40.
- Line #’s 9 and 10 initialize COUNTER with 10.
- Line #11 reads a byte from program memory addressed by TBLPTR into TABLAT, and then increments TBLPTR by 1.
- Line #12 moves the contents of TABLAT into WREG.
- Line #13 moves [WREG] into the destination data memory address pointed to by FSR0, and then increments FSR0 by 1.

PIC18F Table Read/Write instructions

- Line #14 decrements [COUNTER] by 1.
- BNZ at line #15 checks the Z flag, and if Z = 0, the LOOP executed 10 times, and thus, the ASCII numbers 30H through 39H are transferred from program memory to data memory.

PIC18F Subroutine instructions

- Table 7.4 lists PIC18F subroutine instructions. These include PUSH/POP and subroutine CALL/RETURN instructions. The subroutine instructions automatically use the “hardware stack” implemented by the manufacturer. The programmer can create “software stack” if needed for storing local variables.

PIC18F Subroutine instructions

TABLE 7.4 PIC18F Subroutine instructions

Instruction	*Operation*
CALL k, s	Call the subroutine at address k within the two Megabytes of program memory. First, return address (PC + 4) is pushed onto the return stack. If 's' = 1, the WREG, STATUS and BSR registers are also pushed into their respective shadow registers (internal to the CPU), WS, STATUSS and BSRS. If 's' = 0, these registers are unaffected (default). Then, the value 'k' is loaded into PC.
POP	Discards top of stack pointed to by SP and decrements STKPTR by 1.
PUSH	PUSHes or writes the PC onto the stack, and increments STKPTR by 1.
RETFIE	Return from interrupt. Stack is popped and Top-of-Stack (TOS) is loaded into the PC. Interrupts are enabled by setting either the high or low priority global interrupt enable bit. This instruction is normally used at the end of an interrupt service routine.
RETLW k	WREG is loaded with the eight-bit literal 'k'.The program counter is loaded from the top of the hardware stack (the return address).
RETURN s	Return from subroutine. The stack is popped and the top of the stack (TOS) is loaded into the program counter. If 's'= 1, the contents of the shadow registers, WS, STATUSS and BSRS, are loaded into their corresponding registers, WREG, STATUS and BSR. If 's' = 0, these registers are not affected (default).

PIC18F Subroutine instructions

- CALL and RETURN instructions are executed in two cycles.
- POP and PUSH are executed in one cycle.
- The size of each instruction except CALL is one word; the size of CALL instruction is two words.

PIC18F Subroutine instructions

- POP instruction reads (pops) the TOS (Top Of Stack) value from the return stack and discards it; [STKPTR] is decremented by 1. The TOS value becomes the previous value that was pushed onto the return stack.

  As an example, consider the POP instruction with numerical data in the following:
  Prior to execution of the POP: [STKPTR] = 0x15, [0x15] = TOS (Top Of Stack) = 0x000080, Stack (1 level down), [0x14] = 0x000150
  After execution of the POP: [STKPTR] = 0x14, [0x14] = 0x000150
  Note that previous TOS (0x000080) is discarded, although 0x000080 is physically the content of 0x15. and previous Stack (1 level down) is the current TOS.
  Figures 7.3 (a) and (b) depict this.

PIC18F Subroutine instructions

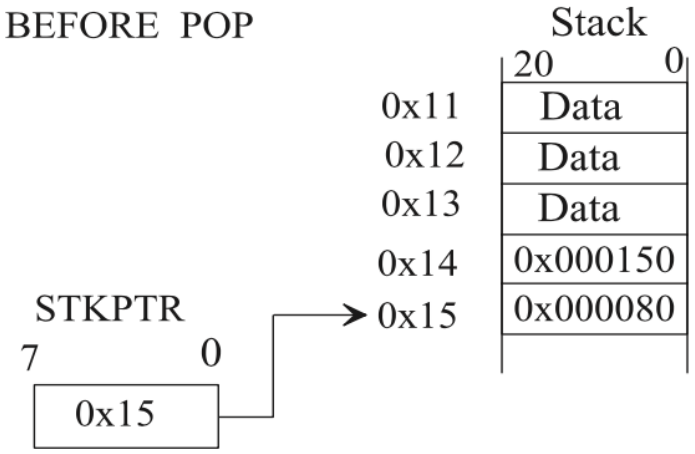


FIGURE 7.3 (a) PIC18F Hardware Stack with arbitrary data before execution of POP instruction

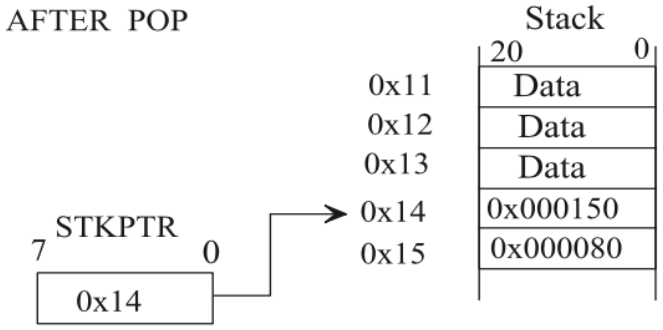


FIGURE 7. 3 (b) PIC18F Hardware Stack with arbitrary data after execution of POP instruction; address 0x15 is assumed to be free

PIC18F Subroutine instructions

- PUSH writes (pushes) PC+2 onto the top of the return stack; [STKPTR] is incremented by 1. The previous TOS value is pushed down on the stack.

  As an example, consider the PUSH instruction with numerical data in the following: Prior to execution of PUSH: [STKPTR] = 0x14, [0x14] = TOS (Top Of Stack) = 0x00007C, [PC+ 2] = 0x0000A4; that is the PUSH instruction is stored at address 0x0000A2.
  After execution of the PUSH: [STKPTR] = 0x15, [0x15] = TOS = 0x0000A4, Previous TOS (one level down), [0x14] = 0x00007C. Figures 7.4 (a) and (b) depict this.

PIC18F Subroutine instructions

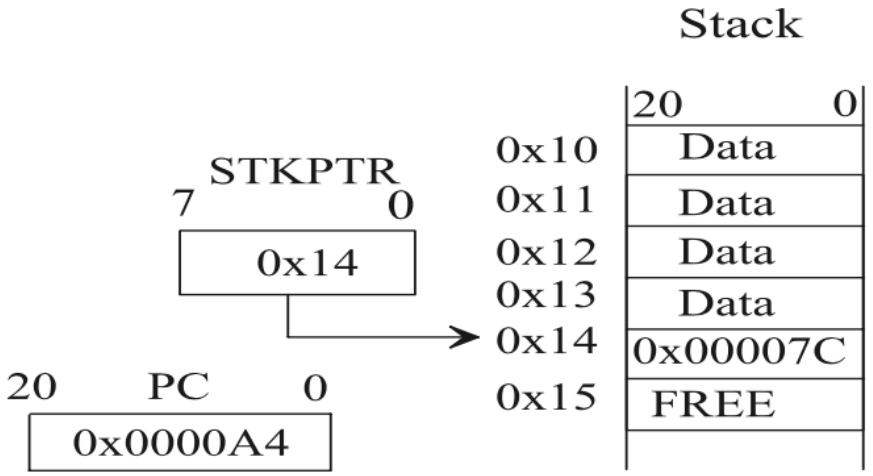


FIGURE 7.4 (a) PIC18F Hardware Stack with arbitrary data before execution of PUSH instruction

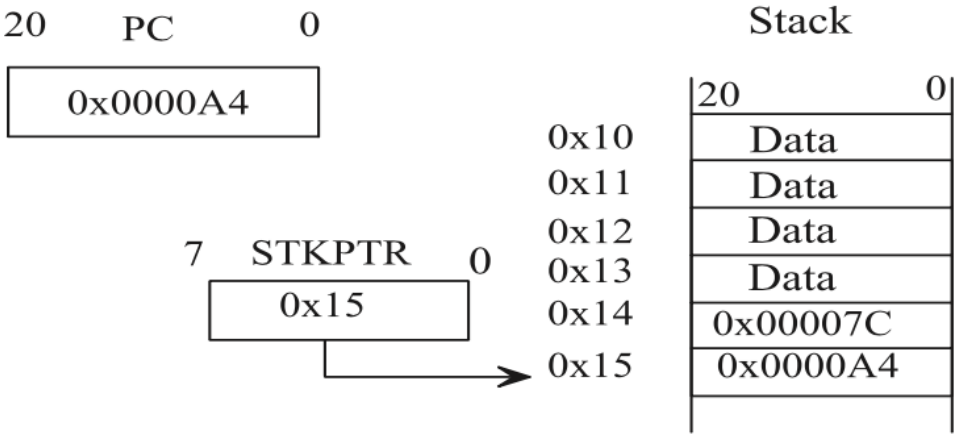


FIGURE 7.4 (b) PIC18F Hardware Stack with arbitrary data after execution of PUSH instruction

PIC18F Subroutine instructions

- The "CALL k, s" with s = 0 (or CALL k) instruction is the simplest way of CALLing a subroutine; s = 0 is the default case. As an example, the CALL START instruction automatically pushes the current contents of the PC onto the stack, and loads PC with the label called START. Note that address START contains the starting address of the subroutine. The "RETURN s" instruction with s = 0 (or RETURN since s = 0 is default) pops the return address (PC pushed onto the stack by the CALL START instruction) from TOS, and loads PC with this address. Thus, control is returned to the main program, and program execution continues with the instruction next to the CALL START.

PIC18F Subroutine instructions

Main Program			*Subroutine*		
	—		SUB	—	; First instruction of subroutine
	—			—	
	—			—	
START	CALL	SUB		—	
	—			—	
	—			—	
	—			RETURN	; Last instruction of subroutine

Here, the CALL SUB instruction in the main program calls the subroutine SUB. In response to the CALL instruction, the PIC18F pushes the current PC contents (START in this case) onto the stack and loads the starting address SUB of the subroutine into PC. After the subroutine is executed, the RETURN instruction at the end of the subroutine pops the address START from the stack into PC, and program control is then returned to the main program. Note that in the PIC18F MPLAB assembler, the starting address of the main program must be at a lower address than the starting address of the subaroutine.

PIC18F System Control instructions

TABLE 7.5 PIC18F System control instructions

Instruction	*Operation*
CLRWDT	Clears watchdog timer to 0.
RESET	Resets all registers and flags to their 'hardware reset" values. The hardware RESET is performed upon activation of the PIC18F input pin. The RESET instruction provides software reset.
SLEEP	The PIC18F is put into sleep mode with the oscillator stopped.
NOP	No Operation

- All instructions in the above are exected in one cycle.
- The size of each instruction is one word.

PIC18F Hardware vs. Software stack

- The PIC18F stack is a group of thirty one 21-bit registers to hold memory addresses. This stack (also called the “hardware stack”) is neither part of data memory nor program memory. Note that the size of the stack (21-bit) is the same as the size of the PC (21-bit). The SP (Stack Pointer) is 5-bit wide in order to address 31 registers.
- In the PIC18F, after a POP (stack read), the SP is decremented by one while the SP is incremented by one after a PUSH (stack write). Also, the SP points to the last used address. Although with a 5-bit SP, 32 registers are available, the PIC18F stack provides 31 registers with addresses 00001_2 through 11111_2.

PIC18F Hardware vs. Software stack

- These 31 registers are typically used to store return addresses after execution of the subroutine CALL instructions. Sometimes it may be necessary to save local variables before executing a subroutine CALL instruction, and the hardware stack may not be adequate. In that case, the user can create a "software stack" using one or more of the three File Select Registers (FSRs) as the SP along with the registers in data memory.
- The programmer can implement a software stack using the POP instruction, and hence, will be able to properly manage the return stack. The PUSH instruction allows implementing a software stack by modifying TOS and then pushing it onto the return stack.

PIC18F Hardware vs. Software stack

- The PIC18F uses one of the FSR's as the software stack pointer, and supports software stack with the register indirect postincrement and predecrement addressing modes. In addition to the software stack pointers (FSRn), any bank of data registers can be used for the software stack. Subroutine CALLs, and interrupts automatically use the hardware stack pointer (STKPTR).

PIC18F Hardware vs. Software stack

- The PIC18F accesses the system stack from the top for operations such as subroutine calls or interrupts. This means that stack operations such as subroutine calls or interrupts access the hardware stack automatically from HIGH to LOW memory.
- The low five bits of the STKPTR is used as the stack pointer for the hardware stack. The STKPTR can be initialized using PIC18F MOVE instructions. For example, in order to load 0x14 into the STKPTR, the following instruction sequence can be used:

```
MOVLW    0x14     ; Load 0x14 into WREG
MOVWF    STKPTR   ; Load [WREG] into STKPTR
```

PIC18F Hardware vs. Software stack

- The STKPTR is incremented by 1 after a push and decremented by one after a pop. Suppose that a PIC18F CALL instruction such as CALL 0x000200 located at [PC] = 0x000100 is executed; then, after execution of the subroutine call, the PIC18F will push the current contents of PC (0x000102) onto the hardware stack, and then load PC with 0x000200. The RETURN instruction at the end of the subroutine will pop 0x000102 from the hardware stack into the PC and return control to the main program.

PIC18F Hardware vs. Software stack

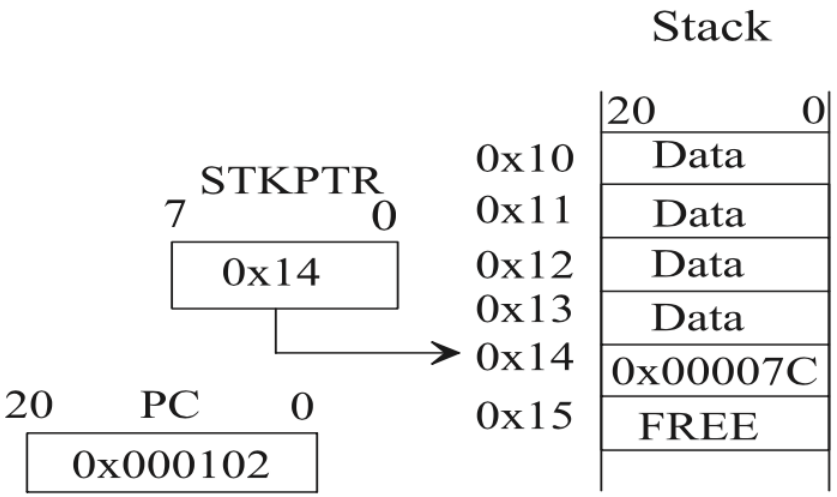


FIGURE 7. 5 (a) PIC18F Hardware Stack with arbitrary data before execution of CALL 0x000200 instruction

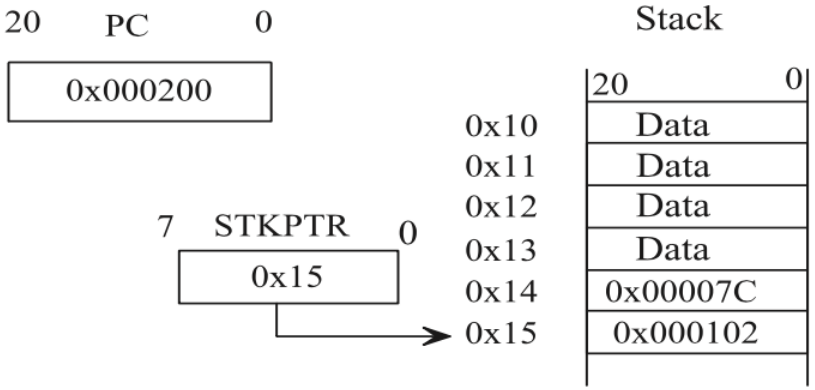


FIGURE 7. 5 (b) PIC18F Hardware Stack with arbitrary data after execution of CALL 0x000200 instruction

PIC18F Hardware vs. Software stack

- In the PIC18F, software stack can be created using appropriate addressing modes. Typical PIC18F memory instructions such as the MOVFF instruction can be used to access the stack.
- Also, by using one of the three FSRn (FSR0–FSR2) as software stack pointers, stacks can be filled from either HIGH to LOW memory or vice versa:

PIC18F Hardware vs. Software stack

- 1. Filling a stack from HIGH to LOW memory (Top of the stack) is implemented with postdecrement mode for the push and preincrement mode for pop.
- 2. Filling a stack from LOW to HIGH (Bottom of the stack) memory is implemented with preincrement for the push and postdecrement for pop.

PIC18F Hardware vs. Software stack

- The programmer can create a software stack growing from HIGH to LOW memory addresses using FSRn as the stack pointer. In this case, the stack is accessed from the top. To push the contents of a data register onto the software stack, MOVFF instruction with appropriate addressing modes can be used. For example, to push contents of a data register 0x30 using FSR0 as the stack pointer, the following PIC18F instruction sequence can be used:

```
LFSR    0, 0x0070          ; initialize FSR0 with 0x70 to be used as the SP
MOVFF   0x30, POSTDEC0     ; Push [0x30] to stack, decrement SP (FSR0) by 1
```

PIC18F Hardware vs. Software stack

- This is shown in Figures 7.6(a) and (b). Figure 7.6(a) shows the software stack with arbitrary data prior to execution of the above instructions. Figure 7.6(b) shows the software stack with arbitrary data after execution of the above instructions. Note that the stack pointer, FSR0 in this case, is decremented by 1 after PUSH. Hence, the stack grows from HIGH to LOW addresses.

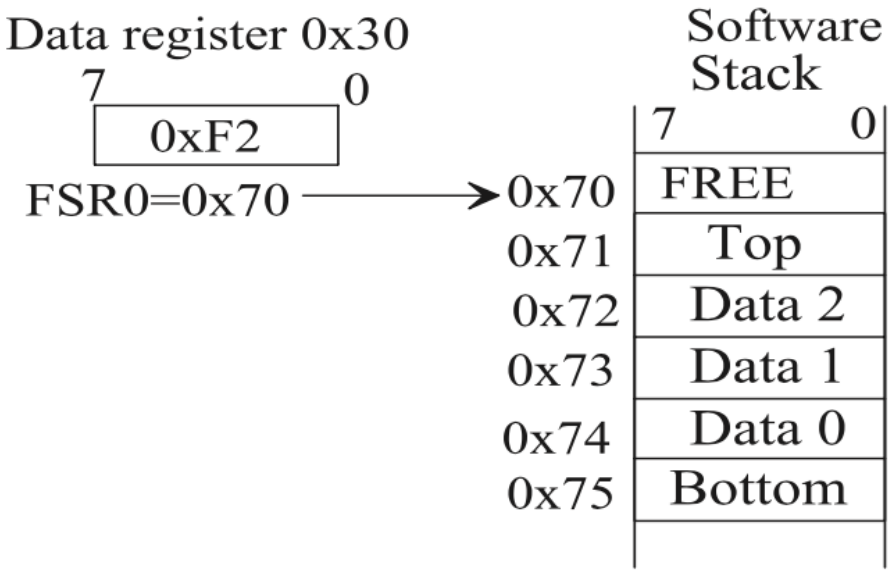


FIGURE 7.6 (a) PIC18F software stack with arbitrary data accessing stack from the top

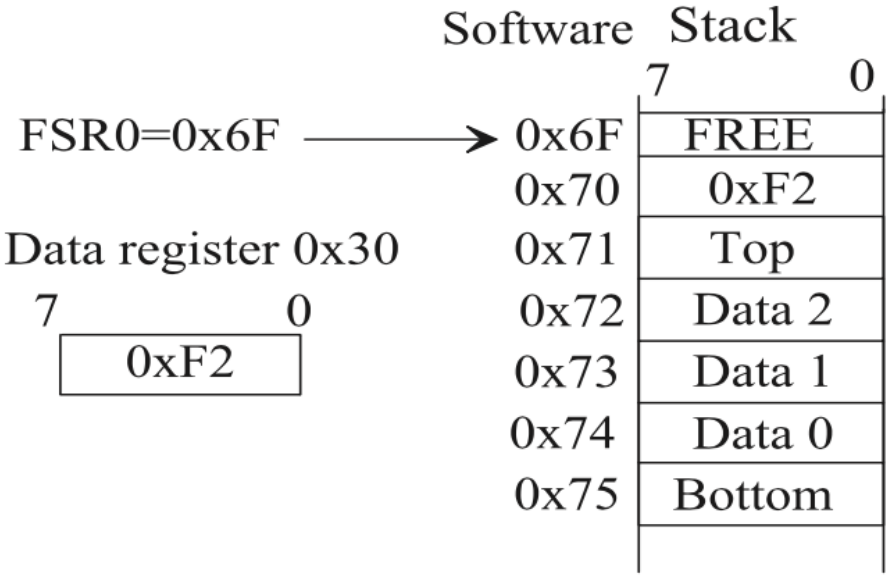


FIGURE 7.6 (b) PIC18F software stack with arbitrary data accessing stack from the top

PIC18F Hardware vs. Software stack

- The 8-bit data 0xF2 can be popped from the stack into another data register 0x20, for example, using the MOVFF PREINC0, 0x20 instruction. Note that the stack pointer, FSR0 in this case, is incremented by 1 before POP.
- Next, consider the stack growing from LOW to HIGH memory addresses in which the programmer also utilizes FSRn as the stack pointer. In this case, the stack is accessed from the bottom.

PIC18F Hardware vs. Software stack

- To push the 8-bit contents of a data register onto the software stack, MOVFF instruction with appropriate addressing modes can be used. For example, to push contents of a data register 0x20 using FSR1 as the stack pointer, the following PIC18F instruction sequence can be used:

```
LFSR    1, 0x0053          ; initialize FSR1 to 0x53 to be used as the SP
MOVFF 0x20, PREINC0  ; Increment SP (FSR1) by 1, Push [0x20] to stack.
```

PIC18F Hardware vs. Software stack

- Figure 7.7(a) shows the software stack with arbitrary data prior to execution of the above instructions. Figure 7.7(b) shows the software stack after execution. Note that the stack pointer, FSR1 in this case, is incremented by 1 after PUSH.
- The 8-bit data 0x17 can be popped from the stack into another data register 0x26, for example, using the MOVFF POSTDEC1, 0x26 instruction. Note that the stack pointer, FSR1 in this case, is decremented by 1 after POP.

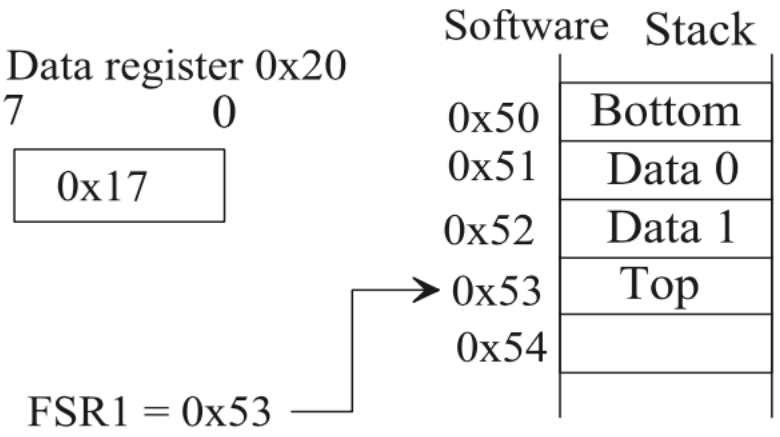


FIGURE 7.7 (a) PIC18F software stack with arbitrary data growing from HIGH to LOW memory before PUSH

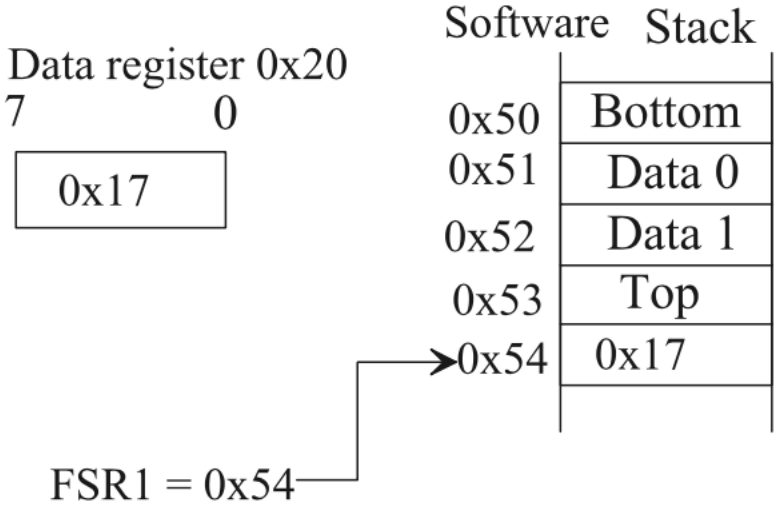


FIGURE 7.7 (b) PIC18F software stack with arbitrary data growing from HIGH to LOW memory after PUSH

PIC18F Hardware vs. Software stack

Example 7.3 Write a PIC18F subroutine at address 0x100 to compute $Y = \sum_{i=1}^{N} X_i^2$. Assume the X_i's are 8-bit unsigned integers and $N = 4$. The numbers are stored in consecutive locations. Assume data register 0x40 points to the first element of the array for X_i's. The array elements are stored from LOW to HIGH memory addresses. The subroutine will store the 16-bit result (*Y*) in data memory registers 0x21 (high byte) and 0x20 (low byte), Also, write the main program at address 0x50 that will initialize STKPTR to 0x05, FSR0 to 0x0040, call the subroutine, compute (*Y*/4) by discarding the remainder, and then stop. Verify the correct operation of the programs using the MPLAB. Show screen shots as necessary.

PIC18F Hardware vs. Software stack

```
        INCLUDE  <P18F4321.INC>
        ORG      0x50            ; Starting address of the main program
; LOAD FOUR ARBITRARILY CHOSEN DATA  INTO DATA MEM ADDR . 0x40 TO 0x43
        MOVLW    0x7E            ; Move 0x7E into WREG
        MOVWF    0x40            ; Move 0x7E into file register 0x40
        MOVLW    0x08            ; Move 0x08 into WREG
        MOVWF    0x41            ; Move 0x08 into file register 0x41
        MOVLW    0x23            ; Move 0x23 into WREG
        MOVWF    0x42            ; Move 0x23 into file register 0x42
        MOVLW    0x30            ; Move 0x30 into WREG
        MOVWF    0x43            ; Move 0x43 into file register 0x40
; INITIALIZE STKPTR, CALL SUBROUTINE, AND DIVIDE BY 4 BY RIGHT SHIFT TWICE
        MOVLW    0x05            ; Move 0x05 into WREG
        MOVWF     STKPTR         ; Load 0x05 into STKPTR
        LFSR     0,0x0040        ; Load File Select Register 0 with register 0x0040
        CALL     SQR             ; Call the function SQR
        BCF      STATUS, C       ; Clear the carry flag
        RRCF     0x21,F          ; Rotate right, or divide by 2
        BCF      STATUS, C       ; Clear the carry flag
        RRCF     0x20,F          ; Rotate right, or divide by 2
FINISH  GOTO     FINISH          ; Halt
```

PIC18F Hardware vs. Software stack

```
        ORG     0x100           ; Starting address of the subroutine
SQR     MOVLW   0x00
        MOVWF   0x21            ; Clear register 0x21
        MOVWF   0x20            ; Clear register 0x20
        MOVLW   0x04
        MOVWF   0x60            ; Move 0x04 into register 0x60
BACK    MOVFF   INDF0, 0x50     ; Move the value addressed by FSR0 into
                                ; register 0x50. 0x50 is used as a holding register in
                                ; data memory Should not be confused with the
                                ; starting address 0x50 of the main program which
                                ; is in program memory of the PIC18F
        MOVF    POSTINC0, W     ; Move value addressed by FSR0  into WREG, and
                                ; then auto increment FSR0 by 1
        MULWF   0x50            ; Multiply WREG by 0x50, or X squared
        MOVF    PRODL, W        ; Move low byte of answer to WREG
        ADDWF   0x20, F         ; Sum with value in 0x20
        MOVF    PRODH, W        ; Move high byte of product to WREG
        ADDWFC  0x21, F         ; Sum with carry with value in 0x21
        DECFSZ  0x60, F         ; Decrement register 0x60 by one, skip next step if 0
        GOTO    BACK            ; Start over
        RETURN                  ; Return to main code
        END
```

Multiplication and Division algorithms

- An unsigned binary number is always positive. An 8-bit unsigned binary integer represents all numbers from 00_{16} through FF_{16} (0_{10} through 255_{10}).
- A signed binary number, on the other hand, includes both positive and negative numbers. It is represented in the microcontroller in two's-complement form.

Multiplication and Division algorithms

- The most significant bit (MSB) of a signed number represents the sign of the number. For example, bit 7 of an 8-bit number represents the signs of the respective numbers. A “0” at the MSB represents a positive number; a “1” at the MSB represents a negative number.
- Note that the 8-bit binary number 11111111 is 255_{10} when represented as an unsigned number. On the other hand, 11111111_2 is -1_{10} when represented as a signed number.

Multiplication and Division algorithms

- The PIC18F includes only unsigned multiplication instruction. The PIC18F instruction set does not provide any instructions for signed multiplication, unsigned and signed division instructions. These algorithms are covered in details in Chapter 4.

Signed Multiplication algorithm

- Signed multiplication can be performed using various algorithms. A simple algorithm follows. Assume that M (multiplicand) and Q (multiplier) are in two's-complement form. Assume that M_n and Q_n are the most significant bits (sign bits) of M and Q, respectively.

Signed Multiplication algorithm

1. If $M_n = 1$, compute the two's complement of M.; else, keep M unchanged.
2. If $Q_n = 1$, compute the two's complement of Q; else, keep Q unchanged.
3. Multiply the n - 1 bits of the multiplier and the multiplicand using unsigned multiplication.
4. The sign of the result, $S_n = M_n \oplus Q_n$.
5. If $S_n = 1$, compute the two's-complement of the result obtained in step 3; else, keep result unchanged.

Signed Multiplication algorithm

- Assume that M and Q are two's-complement numbers. Suppose that $M = 1100_2$ and $Q = 0111_2$. Because $M_n = 1$, take the two's-complement of $M = 0100_2$; because $Q_n = 0$, do not change Q. Multiply 0111_2 and 0100_2 using the unsigned multiplication method discussed before. The product is 00011100_2. The sign of the product $S_n = M_n \oplus Q_n = 1 \oplus 0 = 1$. Hence, take the two's-complement of the product 00011100_2 to obtain 11100100_2, which is the final answer: -28_{10}.

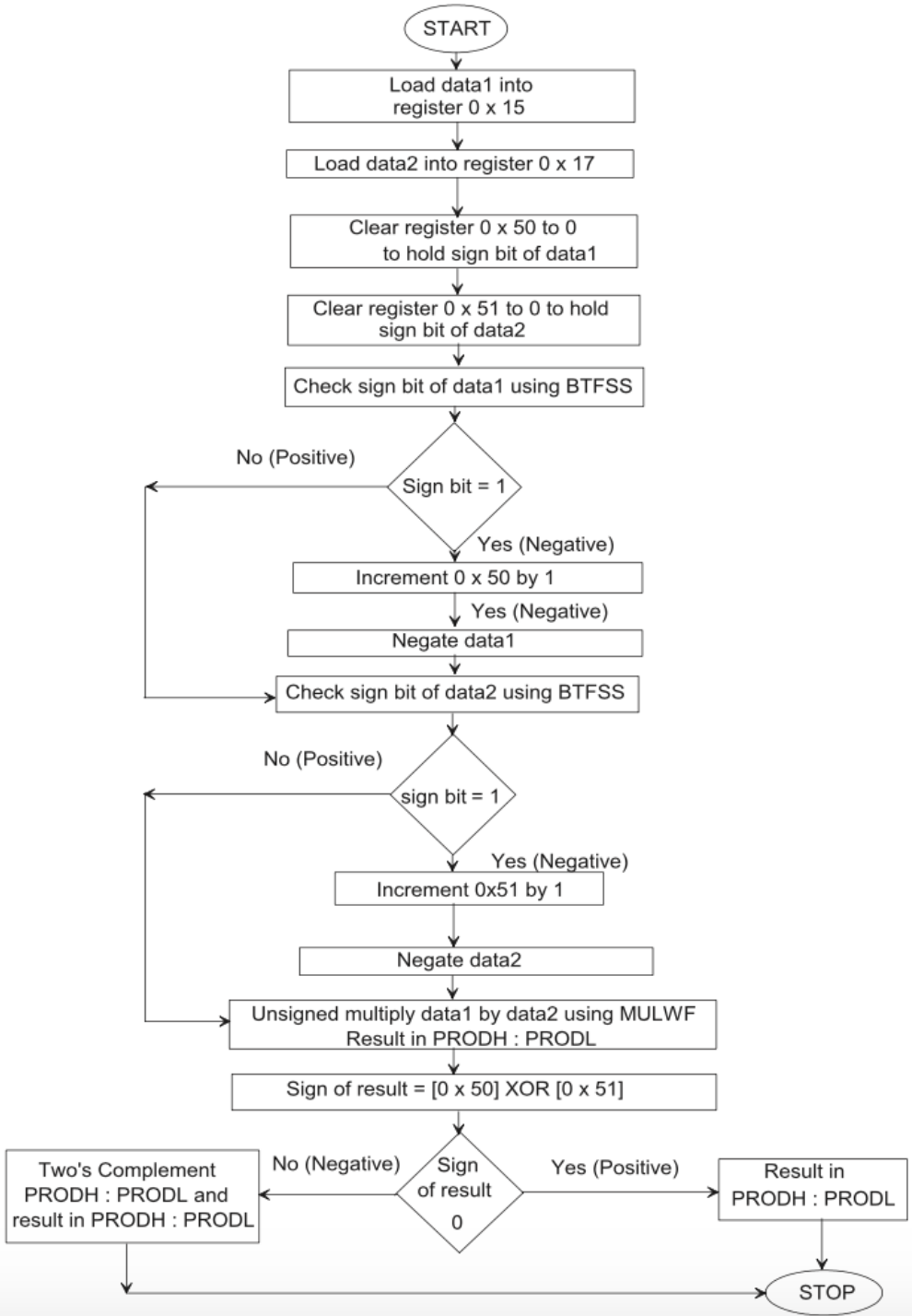
Signed Multiplication algorithm

Example 7.4 Using the signed multiplication algorithm just described, it is desired to multiply two 8-bit signed numbers stored in data registers 0x15 and 0x17. Save the 16-bit result in PRODH:PRODL.

(a) Flowchart the problem.

(b) Convert the flowchart to PIC18F assembly language program starting at address 0x100.

Signed Multiplication algorithm



Signed Multiplication algorithm

```
        INCLUDE <P18F4321.INC>
        ORG     0x100
MULT1   EQU     0x15
MULT2   EQU     0x17
SIGN1   EQU     0X50
SIGN2   EQU     0X51
        MOVLW   0xFE        ; Load first 8-bit data (-2) in WREG
        MOVWF   MULT1       ; Save in MULT1
        MOVLW   0xFC        ; Load 2nd 8-bit data (-4) in WREG
        MOVWF   MULT2       ; Save in MULT2
        CLRF    SIGN1       ; Clear [SIGN1] to 0
```

Signed Multiplication algorithm

```
                CLRF    SIGN2       ; Clear [SIGN2] to 0
; STEPS 1 AND 2 OF THE ALGORITHM OF  SECTION 7.7.1
                BTFSS   MULT1, 7    ; Check sign bit 7  for 1 for 1st #
                BRA     NEG         ; If sign = 0, branch to check sign of 2nd #
                INCF    SIGN1       ; Increment [SIGN1] if  sign of 1st# = 1
                NEGF    MULT1       ; and take 2's complement of [MULT1]
        NEG     BTFSS   MULT2, 7    ; Check sign bit 7  for 1 for 2nd #
                BRA     POSMUL      ; If both sign= 0, branch for unsigned mul
                INCF    SIGN2       ; Increment [SIGN2] if  sign of 2nd # = 1
                NEGF    MULT2       ; and take 2's complement of [MULT2]
; STEP 3 OF THE ALGORITHM OF  SECTION 7.7.1
        POSMUL  MOVF    MULT1, W    ; Move [MULT1] to WREG
                MULWF   MULT2       ; Unsigned product in PRODH:PRODL
                MOVF    SIGN1, W    ; Move [SIGN1] to WREG
                XORWF   SIGN2       ; Compute sign of the result
                BTFSS   SIGN2, 0    ; If sign of result is 0, result in
                BRA     FINISH      ; PRODH:PRODL and Stop
                COMF    PRODL       ; For negative result, take comp of PROD
```

Signed Multiplication algorithm

```
; STEPS  4  AND 5  OF THE ALGORITHM OF  SECTION 7.7.1
                COMF     PRODH     ; Take 2's complement of PRODL
                MOVLW    1
                ADDWF    PRODL
                MOVLW    0
                ADDWFC PRODH, F ; Result in PRODH:PRODL in 2's comp
        FINISH  SLEEP
                END
```

Unsigned Division algorithm

The 8-bit by 8-bit unsigned division can be performed using the repeated subtraction algorithm. For example, consider dividing 7_{10} by 3_{10} as follows:

Dividend	Divisor	Subtraction Result	Counter
7_{10}	3_{10}	$7 - 3 = 4$	1
		$4 - 3 = 1$	$1 + 1 = 2$

Quotient = counter value = 2

Remainder = subtraction result = 1

Unsigned Division algorithm

- 1. Compare the dividend with the divisor for equality.
- 2. If equal, increment the counter by 1, and then perform (dividend - divisor). Store the subtraction result in the data register holding the dividend. If not equal, go to step 3.
- 3. Compare if dividend > divisor. If greater, increment the counter by 1 and then perform (dividend - divisor). Store the subtraction result in the data register holding the dividend.

Unsigned Division algorithm

- 4. Go to Step 1, and repeat steps 1 through 3 until the subtraction result is less than or equal to 0.
- 5. When the subtraction result in dividend is less than or equal to the divisor, go to halt. The counter will contain the *quotient* (number of times divisor can be subtracted until subtraction result is less than the divisor). The data register holding the dividend will contain the *remainder* (result of subtraction).

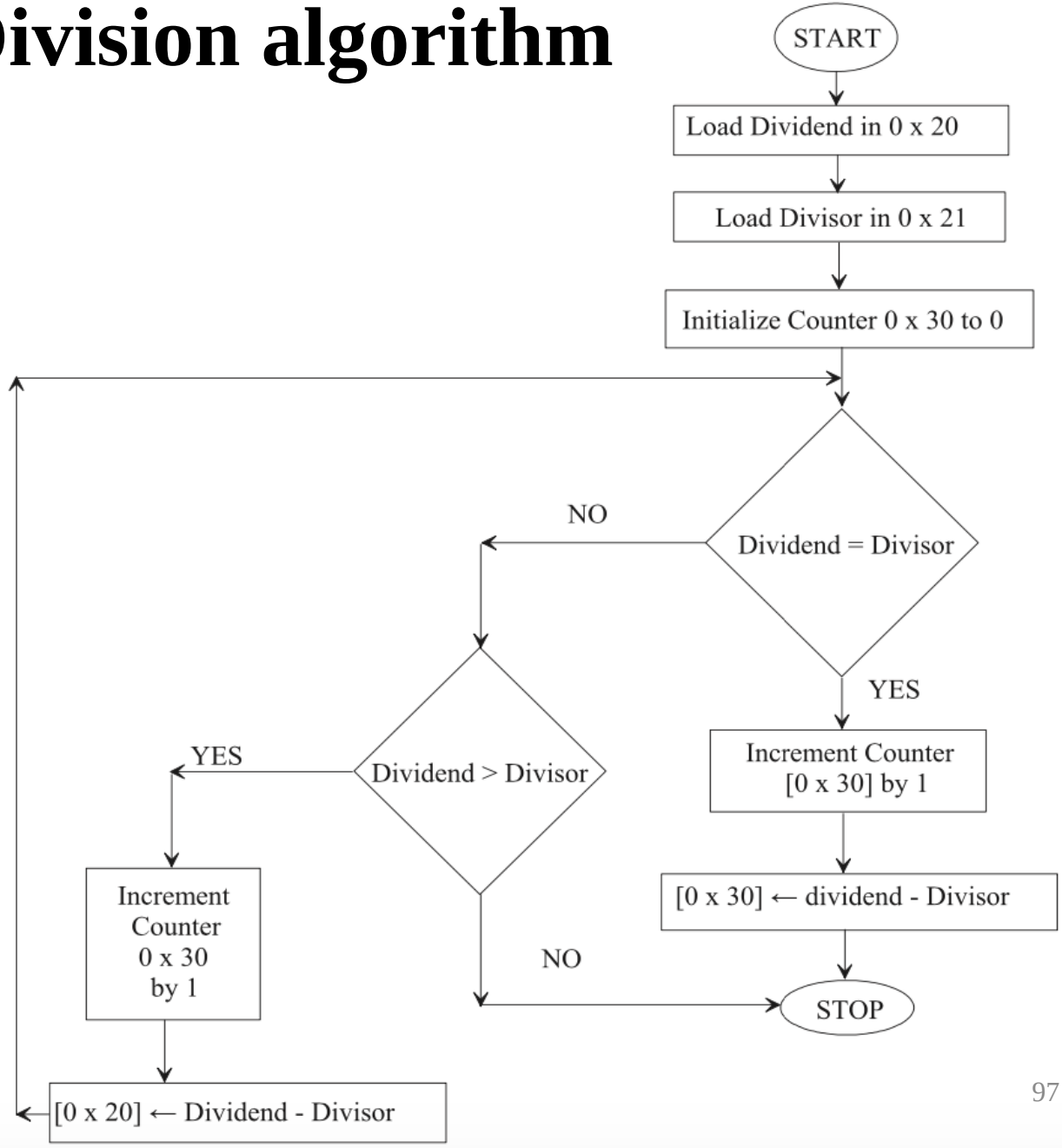
Unsigned Division algorithm

Example 7.5 Using the unsigned division algorithm just described, it is desired to divide an 8-bit unsigned number (dividend) stored in data register 0x20 by another 8-bit unsigned number (divisor) stored in data register 0x30. Save the 16-bit result in data registers 0x71 (remainder high byte) and 0x70 (quotient in low byte).

(a) Flowchart the problem.

(b) Convert the flowchart to PIC18F assembly language program starting at address 0x100.

Unsigned Division algorithm



```
(b)         INCLUDE <P18F4321.INC>
            ORG     0x100
DIVIDEND    EQU     0x20
DIVISOR     EQU     0x21
COUNTER     EQU     0x30
            MOVLW   16              ;  Dividend in WREG
            MOVWF   DIVIDEND        ; Store dividend in 0x20
            MOVLW   4               ; Divisor in WREG
            MOVWF   DIVISOR         ; Store divisor in 0x21
            CLRF    COUNTER         ; Clear Counter to 0
; STEPS 1 AND 2 OF THE ALGORITHM OF SECTION 7.7.2
BACK        CPFSEQ  DIVIDEND        ; If  dividend equals divisor, skip next instr.
            BRA     RESULT          ; If not equal, branch to RESULT
            INCF    COUNTER, F      ; Increment  [0x20] by 1
            SUBWF   DIVIDEND, F     ; Subtract divisor from dividend, result in 0x20
            BRA     FOREVER         ; Go to Halt
; STEPS 3 , 4 AND 5 OF THE ALGORITHM OF SECTION 7.7.2
RESULT      CPFSGT  DIVIDEND        ; If  dividend greater than divisor, skip next inst.
            BRA     FOREVER         ; Quotient in 0x30, Remainder in 0x20, halt
            INCF    COUNTER, F      ; Increment [0x20] by 1
            SUBWF   DIVIDEND, F     ; Subtract divisor from dividend, result in 0x20
            BRA     BACK            ; Repeat
FOREVER     GOTO    FOREVER         ; Halt
            END
```

Signed Division algorithm

- The 8-bit by 8-bit signed division algorithm uses the equation for division: Dividend = Quotient x Divisor + Remainder
- A simple algorithm follows. Assume that DV (Dividend) and DR (Divisor) are in two's-complement form. For the first case, perform unsigned division using repeated subtraction of the magnitudes without the sign bits. The sign bit of the quotient is determined as $DV_n \oplus DR_n$.

Signed Division algorithm

- 1. If $DV_n = 1$, compute the two's complement of DV, else keep DV unchanged.
- 2. If $DR_n = 1$, compute the two's complement of DR, else keep DR unchanged.
- 3. Divide the *n* - 1 bits of the dividend by the divisor using unsigned division algorithm (repeated subtraction).
- 4. The sign of the Quotient, $Q_n = DV_n \oplus DR_n$. The sign of the remainder is the same as the sign of the dividend unless the remainder is zero.
- 5. If $Q_n = 1$, compute the two's-complement of the quotient obtained in step 3, else keep the quotient unchanged.

Signed Division algorithm

- Example 7.6. Write a PIC18F assembly language program at address 0x100 to divide an 8-bit signed number (dividend) in register 0x30 by another 8-bit signed number (divisor) in register 0x40.

Signed Division algorithm

```
            INCLUDE <P18F4321.INC>
            ORG     0x100
COUNTER     EQU     0x20
DIVIDEND    EQU     0x30
DIVISOR     EQU     0x40
SIGN1       EQU     0X50
SIGN2       EQU     0X51
            MOVLW   4               ; Load 8-bit data (+4) in WREG
            MOVWF   DIVIDEND        ; Save in DIVIDEND
            MOVLW   0xFE            ; Load 8-bit data (-2) in WREG
            MOVWF   DIVISOR         ; Save in DIVISOR
            CLRF    SIGN1           ; Clear [SIGN1] to 0
            CLRF    SIGN2           ; Clear [SIGN2] to 0
; STEPS 1 AND 2 OF THE ALGORITHM OF SECTION 7.7.3
            BTFSS   DIVIDEND, 7     ; Check sign bit 7  for 1 for 1st #
            BRA     NEG             ; If sign= 0, branch to check sign of 2nd#
            INCF    SIGN1           ; Increment [SIGN1] if  sign of 1st# = 1
            NEGF    DIVIDEND        ; take 2's complement of [DIVIDEND]
NEG         BTFSS   DIVISOR, 7      ; Check sign bit 7  for 1 for [DIVISOR]
            BRA     POSDIV          ; If both signs = 0, branch for unsigned division
            INCF    SIGN2           ; Increment [SIGN2] if  sign of 2nd# = 1
            NEGF    DIVISOR         ; and take 2's complement of [DIVISOR]
```

```
; STEP 3 OF THE ALGORITHM OF  SECTION 7.7.3
POSDIV    MOVF    DIVISOR, W     ; Load divisor into WREG
          CLRF    COUNTER        ; Clear Counter to 0
BACK      CPFSEQ  DIVIDEND       ; If  dividend equals divisor, skip next instr.
          BRA     RESULT1        ; If not equal, branch to RESULT
           INCF   COUNTER, F     ; Increment  [0x20] by 1
          SUBWF   DIVIDEND, F    ; Subtract divisor from dividend, result in 0x30
RESULT1   CPFSGT  DIVIDEND       ; If  dividend greater than divisor, skip next inst.
          BRA     RESULT         ; Quotient in 0x20, Remainder in 0x30
          INCF    COUNTER, F     ; Increment [0x20] by 1
          SUBWF   DIVIDEND, F    ; Subtract divisor from dividend, result in 0x30
          BRA     BACK           ; Repeat
; STEPS  4 AND 5 OF THE ALGORITHM OF  SECTION 7.7.3
RESULT    MOVF    SIGN1, W       ; Move [SIGN1] to WREG
          XORWF   SIGN2          ; Compute sign of the result
          BTFSS   SIGN2, 0       ; If sign of the quotient is 0, result in
          BRA     FINISH         ; 0x70 and Stop
          NEGF    0x20           ; For negative result, take comp of [0x20]
          BTFSS   SIGN1, 0       ; Check sign of  DIVIDEND
          BRA     FINISH         ; If plus, positive remainder in 0x30
          NEGF    0x30           ; If negative, negate remainder in 0x30
FINISH    SLEEP
          END
```

Advanced Programming Examples

Example 7.7 Write a subroutine in PIC18F assembly language program at address 0x200 to find the n^{th} number (for example, n = 0 to 6) of the Fibonacci sequence. The subroutine will obtain the desired Fibonacci number using a lookup table stored starting at an address 0x150 in the program memory. Also, write the main program at address 0x100 which will transfer Fibonacci array from program memory stored at address starting at 0x200 to data memory stored starting at address 0x40, initialize STKPTR to 0x15, store a number (0 to 6) in WREG, initialize data pointer FSR1 to 0x40, call the sbroutine, and stop. The Fibonacci sequence for n = 0 to 6 is provided below:

n	Fib(n)
0	1
1	1
2	2
3	3
4	5
5	8
6	13

```
; MAIN  PROGRAM
                    INCLUDE  <P18F4321.INC>
                    ORG      0x100          ; Starting address of  the main program
        COUNTER     EQU      0x20
;FIBONACCI ARRAY TRANSFER FROM PROGRAM MEMORY TO DATA MEMORY
                    MOVLW    UPPER ADDR; Move upper 5 bits (00H) of address
                    MOVWF    TBLPTRU      ;  to TBLPTRU
                    MOVLW    HIGH ADDR    ; Move bits 15-8 (01H) of address
                    MOVWF    TBLPTRH      ; to TBLPTRH
                    MOVLW    LOW ADDR     ; Move bits 7-0 (50H) of address
                    MOVWF    TBLPTRL      ; to TBLPTRL
                    LFSR     0, 0x40      ; Initialize FSR0 to 0x40 to be used as
                                          ; destination pointer in data memory
                    MOVLW    D'7'         ; Initialize COUNTER with 7
                    MOVWF    COUNTER      ; Move [WREG] into COUNTER
        LOOP        TBLRD*+               ; Read data from program memory into
                                          ; TABLAT, increment TBLPTR by 1
                    MOVF     TABLAT, W    ; Move [TABLAT] into WREG
                    MOVWF    POSTINC0     ; Move W into data memory addressed
                                          ; by FSR0, and then increment FSR0 by 1
                    DECF     COUNTER, F   ; Decrement COUNTER BY 1
                    BNZ      LOOP         ; Branch  if  Z = 0
; INITIALIZE STKPTR, LOAD n, INITIALIZE DATA POINTER, CALL SUBROUTINE
                    MOVLW    0x15         ; Initialize STKPTR to 0x15
                    MOVWF    STKPTR
```

Advanced Programming Examples

```
                MOVLW   4               ; Move n into WREG
                LFSR    1, 0x40         ; Load 0x40 into FSR0 to be used as pointer
                CALL    FIBNUM
FINISH          BRA     FINISH
; READ  THE FIBONACCI  NUMBER FOR n FROM  DATA MEMORY INTO 'W'
; USING MOVF WITH  INDEXED  ADDRESSING MODE
; SUBROUTINE
                ORG     0x200
      FIBNUM    MOVF    PLUSW1, W       ; Result in WREG
                RETURN                  ; Return to FINISH in main
                ORG     0x150
      ADDR      DB      1, 1, 2, 3, 5, 8, D'13'  ; Fibonacci numbers
                END
```

Advanced Programming Examples

Example 7.8 Without using a lookup table, and the MOVF instruction with indexed addressing mode in the subroutine as in Example 7.7, write a subroutine in PIC18F assembly language at address 0x150 to find the n^{th} number (0 to 6) of the Fibonacci sequence. The subroutine will return the desired Fibonacci number in WREG based on 'n' stored by the main program. Also, write the main program at address 0x100 which will store the n^{th} number (0 to 6) in WREG, call the subroutine, and stop. The Fibonacci sequence for n = 0 to 6 is provided below:

n	Fib(n)
0	1
1	1
2	2
3	3
4	5
5	8
6	13

Advanced Programming Examples

This program can be written using the RETLW instruction which is ideal for returning the desired value using an operation alternate to using a table lookup with indexed addressing mode shown in Example 7.6. Note that the RETLW k loads the 8-bit immediate data k into WREG, and returns to the main program by loading the Program Counter with the address from the top of the hardware stack. The assembly language program is provided below:

```
        INCLUDE  <PIC18F4321.INC>
        ORG      0x100          ; Main program
        MOVLW    0x10           ; Initialize STKPTR with  0x10
        MOVWF    STKPTR
        MOVLW    6              ; Load n into WREG
        CALL     FIBNUM         ; Call subroutine FIBNUM to find Fibonacci #
HERE    BRA      HERE           ; Halt
        ORG      0x150          ; Subroutine
FIBNUM  MULLW    2              ; PRODH:PRODL  2 x n, offset of RETLW table.
                                ; 'n' is  multiplied by 2 since the  instruction size is
                                ; word
```

Advanced Programming Examples

```
MOVF    PRODL, W  ; Save low order 8 bits of the product in WREG
ADDWF   PCL       ; PCL = PCL + 2 x n
                  ; Fibonacci number table follows
RETLW   0x01
RETLW   0x01
RETLW   0x02
RETLW   0x03
RETLW   0x05
RETLW   0x08
RETLW   0x0D      ; 13 in decimal
END               ; WREG is loaded with the 8-bit literal k, and PC is
                  ; loaded with the return address from the top of the
                  ; stack
```

Advanced Programming Examples

Example 7.9 Write a PIC18F assembly language program at address 0x200 to add two 16-bit numbers (N1 and N2), each containing two ASCII digits. The first 16-bit number (N1) is stored in two consecutive locations (from LOW to HIGH) in data memory with the low byte addressed by data reg 0x40, and the high byte addressed by data reg 0x41. The second 16-bit number (N2) is also stored in two consecutive locations (from LOW to HIGH) in data memory with the low byte addressed by data reg 0x50, and the high byte addressed by data reg 0x51. Store the packed BCD result in WREG.

Advanced Programming Examples

Note that ASCII codes for decimal numbers 0 through 9 are 30H through 39H (See chapter 1).

Numerical Example: Assume [N1] = 3439H and [N2] = 3231H. The procedure for adding the two 16-bit ASCII numbers (N1 and N2) will be as follows:

1. Convert N1 and N2 to unpacked BCD numbers by masking off the high four bits using ANDWF instruction. This means that N1 = 0409H and N2 = 0201H.
2. Logically shift the high byte of N1 four times to the left so that the high byte will be converted from 04H to 40H. This is equivalent to swapping the low four bits (nibble) with the high four bits (nibble) using the SWAPF instruction. Logically OR this with the low byte of N1. Hence, N1 will be converted from unpacked BCD (0409H) to packed BCD 49H. Similarly, convert N2 from unpacked BCD (0201H) to packed BCD (21H).

Advanced Programming Examples

3. Add (binary addition) the two packed BCD numbers (49H, 12H) using ADDWF instruction to obtain the following result:

First packed BCD byte = 49H = 0100 1001
Second packed BCD byte = 21H = 0010 0001

Result after binary addition 0110 1010 (6AH)

4. Perform BCD correction on the binary result 0110 1010
Add 6 using DAW instruction 0110

0111 0000 = 70H
(Correct packed BCD result)

```
            INCLUDE  <P18F4321.INC>
            ORG      0x200
COUNTER     EQU      0x45
            MOVLW    0x39          ; #1 LOAD LOW BYTE OF N1 INTO 0x40
            MOVWF    0x40
            MOVLW    0x34          ; #2 LOAD HIGH BYTE OF N1 INTO 0x41
            MOVWF    0x41
            MOVLW    0x31          ; #3 LOAD LOW BYTE OF N2 INTO 0x50
            MOVWF    0x50
            MOVLW    0x32          ; #4 LOAD HIGH BYTE OF  N2 INTO 0x51
            MOVWF    0x51
            MOVLW    2             ; #5 INITIALIZE  COUNTER
            MOVWF    COUNTER
            LFSR     0, 0x40       ; #6  INITIALIZE FSR0 TO 0x40
            LFSR     1, 0x50       ; #7 INITIALIZE FSR1 TO 0x50
; STEP1: CONVERT N1 AND N2 TO UNPACKED BCD.
START       MOVLW    0x0F
            ANDWF    POSTINC0, F   ; #8 CONVERT  N1 TO UNPACKED BCD
            ANDWF    POSTINC1, F   ; #9 CONVERT  N2 TO UNPACKED BCD
            DECF     COUNTER, F    ; #10 DECREMENT  COUNTER BY 1
            BNZ      START         ; #11 BRANCH  IF NOT  ZERO
; UNPACKED BCD RESULT 0x41 (N1 HIGH BYTE), 0x40 (N1 LOW BYTE), 0x51
; (N2 HIGH BYTE), 0x50 (N2 LOW  BYTE)
```

Advanced Programming Examples

```
; STEP2: CONVERT N1 AND N2 FROM  UNPACKED TO PACKED BCD
            SWAPF     0x41, W       ; #12 SWAP LOW NIBBLE OF 0x41 WITH
                                    ; HIGH NIBBLE AND STORE  IN WREG
            IORWF     0x40, F       ; #13 OR [W] WITH [0x40], PACKED BCD N1
            SWAPF     0x51, W       ; #14 SWAP LOW NIBBLE OF 0x51 WITH
                                    ; HIGH NIBBLE AND STORE IN WREG
            IORWF     0x50, W       ; #15 OR [WREG] WITH [0x50], PACKED BCD
                                    ; N2 IN WREG
; STEP3: PERFORM BINARY ADDITION BETWEEN N1 (PACKED BCD) WITH
; N2 (PACKED BCD) AND STORE RESULT IN WREG
            ADDWF     0x40, W       ; #16  BINARY RESULT IN WREG
; STEP4: ADJUST (BCD CORRECTION) [WREG] TO CONTAIN CORRECT PACKED BCD
            DAW                     ; #17 ADJUST THE RESULT TO CONTAIN
                                    ; CORRECT PACKED BCD
FINISH      BRA       FINISH        ; HALT
            END
```

Advanced Programming Examples

- Line #’s 1 through 4 initialize N1 and N2 so that [0x40] = 39H, [0x41] = 34H, [0x50] = 31H, [0x51] = 32H.
- Line #5 initializes COUNTER with loop count of 2 for converting the numbers from ASCII to unpacked BCD.
- Line #’s 6 and 7 initialize FSR0 and FSR1 with 0x40 and 0x50 respectively.
- Line #’s 8 through 11 convert the two bytes of ASCII codes in 0x41 (high byte) and 0x40 (low byte) into unpacked BCD in 0x41 (high byte) and 0x40 (low byte). Also, Line #’s 8 through 11 convert the ASCII numbers, N1 and N2, into their corresponding unpacked BCD bytes.

Advanced Programming Examples

- Line #’s 12 through 15 convert the unpacked BCD numbers (N1 and N2) into packed BCD bytes. This is done by swapping high unpacked bytes of N1 and N2 , and then ORing with the corresponding low unpacked bytes.
- Line #16 performs binary addition of the two packed BCD bytes (N1 and N2), and stores the binary result in WREG.
- The DAW instruction at Line #17 adjusts the contents of WREG to provide the correct packed BCD result.

PIC18F Delay Routine

- Typical PIC18F software delay routines can be written by loading a “counter” with a value equivalent to the desired delay time, and then decrementing the “counter” in a loop using typically MOVE, DECREMENT, and conditional BRANCH instructions. For example, the following PIC18F instruction sequence can be used for a delay loop:

```
        MOVLW   COUNT
        MOVWF   0x20
DELAY   DECF    0X20, F
        BNZ     DELAY
```

PIC18F Delay Routine

- Note that DECF in the above decrements the register 0x20 by one, and if Z =0, the program branches to DELAY and if Z = 1, the PIC18F executes the next instruction. The initial loop counter value of "COUNT" can be calculated using the machine cycles required to execute the following PIC18F instructions:

MOVLW	(1 cycle)
MOVWF	(1 cycle)
DECF	(1 cycle)
BNZ	(2/1 cycles)

PIC18F Delay Routine

- Note that the BNZ instruction requires two different execution times. BNZ requires 2 cycles when the PIC18F branches if Z = 0. However, the PIC18F goes to the next instruction and does not branch when Z = 1. This means that the DELAY loop will require 2 cycles for “(COUNT-1)” times, and the last iteration will take 1 cycle. The desired delay time can be obtained by loading register 0x20 with the appropriate COUNT value.

PIC18F Delay Routine

Assuming a 1-MHz default crystal frequency, the PIC18F's clock period will be 1μs. Note that the PIC18F divides the crystal frequency by 4. This is equivalent to multiplying the clock period by 4. Hence, each instruction cycle will be 4 microseconds. For a 100 microsecond delay, total cycles $= \frac{100\ micro\,\text{sec}}{4\ micro\,\text{sec}} = 25$. The BNZ in the loop will require 2 cycles for (COUNT - 1) times when Z = 0 and the last iteration will take 1 cycle when no branch is taken (Z = 1). Thus, the total cycles including the MOVLW = Cycles for (MOVLW + MOVWF + (BNZ for Z = 0 + DCF) x (COUNT - 1)) + BNZ (Z = 1) = 1 + 1 + 3 × (COUNT - 1) + 1 = 25. Hence, COUNT = 8.3. Therefore, register 0x20 should be loaded with an integer value of 8 for an approximate delay of 100 microseconds.

MOVLW	(1 cycle)
MOVWF	(1 cycle)
DECF	(1 cycle)
BNZ	(2/1 cycles)

PIC18F Delay Routine

Now, in order to obtain delay of one millisecond, the above DELAY loop of 100 microseconds can be used with an external counter. Counter value $= \frac{1\ milli\,\text{sec}}{100\ micro\,\text{secs}} = 10.$

The following instruction sequence will provide an approximate delay of one millisecond:

```
        MOVLW   D'10'
        MOVWF   0x30      ; Initialize counter 0x30 for one millisecond delay
BACK    MOVLW   8
        MOVWF   0x20      ; Initialize counter 0x20 for 100 microsecond delay
DELAY   DECF    0X20, F   ; 100 microsec delay
        BNZ     DELAY
        DECF    0X30, F
        BNZ     BACK
```

PIC18F Delay Routine

- The above instruction sequence will provide an approximate delay of one millisecond. Note that execution times of certain instructions such as “MOVLW D’10’”, “MOVWF 0x30”, “DECF 0x30, F”, and “BNZ BACK” are discarded since their execution times are negligible compared to one millisecond.

PIC18F Delay Routine

Example 7. 10 Assume 1 MHz PIC18F. Consider the following subroutine:

```
DELAY   MOVLW   D'100'
        MOVWF   0x20
LOOP    DECFSZ  0x20, F
        BRA     LOOP
        RETURN
```

(a) Calculate the time delay provided by the above subroutine.

(b) Calculate the counter value to be loaded into data register 0x20 for 1 ms delay.

PIC18F Delay Routine

(a) Each instruction in the above subroutine is executed in one cycle except for the DECFSZ instruction. DECFSZ is executed in one cycle if it does not skip, and two cycles if it skips (See Appendix D for instruction cycles).

Hence, total instruction cycles = Cycle for MOVLW + Cycle for MOVWF + (100-1) (Cycles for DECFSZ if it does not skip and BRA instructions) + (Cycle for DECFSZ if it skips) + Cycle for RETURN

= 1 + 1 + 99 (1 + 1) + 2 + 1

= 203

Since for the PIC18F, one instruction cycle = 4 clock cycles, total delay = (203) x 4 = 812 clock cycles. Also, for 1 MHz clock, each clock cycle is 1 μsec. Hence, total time delay = 812 μsec.

PIC18F Delay Routine

(b) Let n be the counter value. Hence, $(2 + 2 \times (n-1) + 3) \times 4 = 1000\ \mu sec$. Note that 1 ms = 1000 μsec. Therefore, n = 123.5. Therefore, data register 0x20 should be loaded with 124 for an approximate delay of 1 ms.