ADVANCED ASSEMBLY PROGRAMMING

PIC Microcontroller: An Introduction to Software & Hardware Interfacing

Han-Way Huang

Thomson Delmar Learning, 2005

Chung-Ping Young
楊中平

Networked Embedded Applications and Technologies Lab

Department of Computer Science and Information Engineering
National Cheng Kung University, TAIWAN

Signed Arithmetic

- Some quantities can be positive or negative.
- A negative number is represented by the 2's complement of its magnitude.
- The subtract and add instructions can handle negative number properly when they are represented in 2's complement format.

Modulo Math

- All computers uses limited number of bits to represent numbers.
- Adding 2^8 to a number in an 8-bit computer does not affect the result.
- Similarly, adding 2^{16} and 2^{32} to a number in a 16-bit and 32-bit computer do not affect the result.

Procedure for Signed 8-bit Multiplication

Step 1
Multiply two operands (op1 $\times$ op2) disregarding their signs.
Step 2
If op1 is negative, then subtract op2 from the upper byte of the product.
Step 3
if op2 is negative, then subtract op1 from the upper byte of the product.

Procedure for Signed 16-bit Multiplication

Step 1
Multiply two operands (op1 $\times$ op2) disregarding their signs.
Step 2
If op1 is negative, then subtract op2 from the upper 16 bits of the product.
Step 3
if op2 is negative, then subtract op1 from the upper 16 bits of the product.

Example 4.1 Write a program to compute the product of two 8-bit signed numbers represented by data memory locations op1 and op2 and leave the product in file registers PRODH and PRODL.

Solution:

```
        #include <p18F8720.inc>
op1     set     0x00    ; the first operand
op2     set     0x01    ; the second operand
num1    equ     0x87    ; test number 1
num2    equ     0x98    ; test number 2
        org     0x00    ; reset vector
        goto    start
        org     0x08    ; high priority interrupt service routine
        retfie
        org     0x18    ; low priority interrupt service routine
        retfie
start   movlw   num1    ; set up op1 for testing purpose
        movwf   op1,A   ;           "
        movlw   num2    ; set up op2 for testing purpose
        movwf   op2,A   ;           "
        movf    op1,W,A
```

```
        mulwf    op2,A
        btfsc    op2,7,A           ; test the sign bit of the second
operand
        subwf    PRODH,F,A         ; if negative, eliminate extra term
        movf     op2,W,A
        btfsc    op1,7,A           ; test the sign bit of the first operand
        subwf    PRODH,F,A         ; if negative, eliminate extra term
        nop
        end
```

Example 4.2 Write a program to compute the product of two 16-bit signed integers and leave the product in memory locations represented by **result** ... **result+3**.

Solution:

```
        #include <p18F8720.inc>
arg1_hi set      0x00              ; high byte of the first argument
arg1_lo set      0x01              ; low byte of the first argument
arg2_hi set      0x02              ; high byte of the second argument
arg2_lo set      0x03              ; low byte of the second argument
result  set      0x04              ; location to hold the product
num1_hi equ      0x83              ; test number 1
num1_lo equ      0x45              ;         "
num2_hi equ      0x81              ; test number 2
num2_lo equ      0x47              ;         "
```

```
        org     0x00            ; reset vector
        goto    start
        …                       ; interrupt service routines
start   movlw   num1_hi         ; set up test numbers
        movwf   arg1_hi         ;       "
        movlw   num1_lo         ;       "
        movwf   arg1_lo         ;       "
        movlw   num2_hi         ;       "
        movwf   arg2_hi         ;       "
        movlw   num2_lo         ;       "
        movwf   arg2_lo         ;       "
        movf    arg1_lo,W
        mulwf   arg2_lo         ; compute arg1_lo × arg2_lo
        movff   PRODL,result
        movff   PRODH,result+1
        movf    arg1_hi,W
        mulwf   arg2_hi         ; compute arg1_hi × arg2_hi
        movff   PRODL,result+2
        movff   PRODH,result+3
        movf    arg1_lo,W
        mulwf   arg2_hi         ; compute arg1_lo × arg2_hi
        movf    PRODL,W
        addwf   result+1,F
```

```
          movf    PRODH,W
          addwfc  result+2,F
          clrf    WREG
          addwfc  result+3,F      ; add carry to the most significant byte
          movf    arg1_hi,W
          mulwf   arg2_lo         ; compute arg1_hi × arg2_lo
          movf    PRODL,W
          addwf   result+1,F
          movf    PRODH,W
          addwfc  result+2,F
          clrf    WREG            ; add carry to the most significant byte
          addwfc  result+3,F      ;          "
          btfss   arg2_hi,7       ; check the sign of arg2
          goto    sign_arg1
          movf    arg1_lo,W
          subwf   result+2,F      ; delete extra term
          movf    arg1_hi,W       ;          ;          "
          subwfb  result+3,F      ;          "
sign_arg1 btfss   arg1_hi,7       ; check the sign of arg1
          goto    more            ; continue to perform other operation
          movf    arg2_lo,W
          subwf   result+2,F      ; delete extra term
```

```
            movf      arg2_hi,W,A
            subwfb    result+3,F,A
more        nop
            end
```

Unsigned Divide Operation

- Performed by repeated subtraction using the circuit shown in Figure 4.1.
- Registers N, R, and Q are initialized to the **divisor**, **0,** and the **dividend**.
- R and Q will hold the remainder and quotient at the end of the division operation.

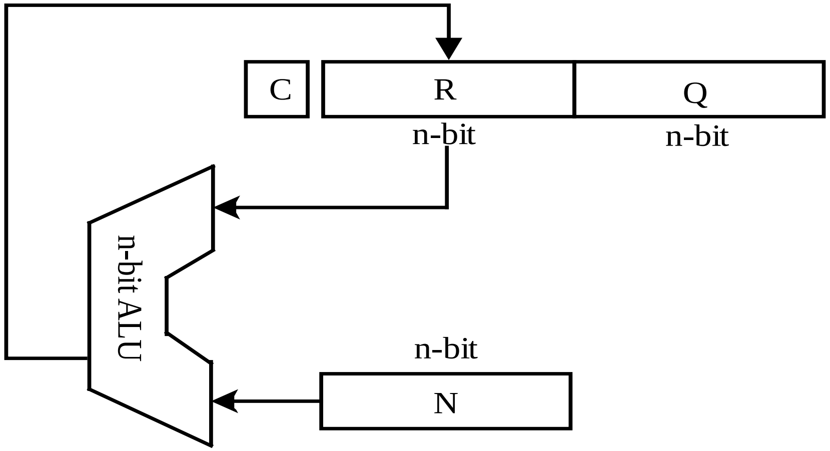

Figure 4.1 Division hardware

Algorithm for Unsigned Division

Repeat the following steps **n** times

Step 1
Shift the register pair (R, Q) one place to the left.
Step 2
Subtract the contents of N from R and put the result back to R if the result is nonnegative.
Step 3
If the result of Step 2 is negative, then set the least significant bit of Q to 0. Otherwise, set the least significant bit of Q to 1.

Example 4.4 Write a program to divide an unsigned 16-bit number into another unsigned 16-bit number.

```
        #include <p18F8720.inc>
        radix   dec         ; set default radix to decimal
lp_cnt  set     0x00        ; loop count
temp    set     0x01        ; temporary storage
dsr     set     0x04        ; divisor
quo     set     0x06        ; quotient
rem     set     0x08        ; remainder
dd_h    equ     0x68        ; high byte of dividend test number
dd_l    equ     0x20        ; low byte of dividend test number
dr_h    equ     0x01        ; high byte of divisor test number
dr_l    equ     0x48        ; low byte of divisor test number
        org     0x00
        goto    start
        org     0x08
        retfie
        org     0x18
        retfie
start   movlw   dd_h        ; initialize Q register in Figure 4.1
        movwf   quo+1,A     ;           "
        movlw   dd_l        ;           "
        movwf   quo,A       ;           "
        movlw   dr_h        ; initialize N register in Figure 4.1
        movwf   dsr+1,A     ;           "
```

```
        movlw   dr_l            ;       “
        movwf   dsr,A           ;       “
        clrf    rem,A           ; initialize R register in Figure 4.1 to 0
        clrf    rem+1,A         ;       “
        movlw   16
        movwf   lp_cnt,A        ; initialize loop count to 16
loop    bcf     STATUS,C,A      ; clear the C flag
        rlcf    quo,F,A         ; rotate (R, Q) pair to the left one place
        rlcf    quo+1,F,A       ;       "
        rlcf    rem,F,A         ;       "
        rlcf    rem+1,F,A       ;       "
        movf    dsr,W,A
        subwf   rem,W,A
        movwf   temp,A          ; save the low byte of the difference
        movf    dsr+1,W,A
        subwfb  rem+1,W,A
        btfss   STATUS,C        ; skip if carry is 1
        goto    less
        bsf     quo,0,A         ; set the quotient bit to 1
        movwf   rem+1,A         ; place the difference in R register
        movff   temp,rem        ;       "
```

```
        goto    next
less    bcf     quo,0,A         ; set the quotient bit to 0
next    decfsz  lp_cnt,F,A      ; decrement the loop count and skip is zero
        goto    loop
        nop
        end
```

Signed Division

- When dividend and divisor differs in sign, the sign of the quotient is minus.
- The sign of the remainder agrees with the dividend.

Example 4.5 Write a PIC18 assembly program to perform the 8-bit signed divide operation and leave the quotient and remainder in registers labeled as **quo** and **rem**, respectively.

```
            #include <p18F8720.inc>
            radix     dec
sign        set       0x00
dvd         set       0x01      ; dividend
dsr         set       0x02      ; divisor
quo         set       0x03      ; quotient
rem         set       0x04      ; remainder
lp_cnt      set       0x05      ; loop count
dd          equ       0x82      ; test number for dividend
dr          equ       0xf5      ; test number for divisor
            org       0x00
            goto      start
            org       0x08
            retfie
            org       0x18
            retfie
start       bcf       sign,2,A  ; initialize the sign of quotient to positive
            bcf       sign,1,A  ; initialize the sign of dividend to positive
            bcf       sign,0,A  ; initialize the sign of divisor to positive
            movlw     dd
            movwf     dvd,A
            movlw     dr
            movwf     dsr,A
```

```
        btfss   dvd,7,A         ; check the sign of dividend
        goto    second
        btg     sign,2          ; change the sign of quotient
        bsf     sign,1          ; record the sign bit of the dividend
        negf    dvd,A           ; compute the magnitude of dividend
second  btfss   dsr,7,A         ; check the sign of the divisor
        goto    do_it
        btg     sign,2,A        ; change the sign of quotient
        bsf     sign,0,A        ; set the sign of the divisor
        negf    dsr,A           ; compute the magnitude of divisor
do_it   movf    dvd,W,A
        movwf   quo,A
        clrf    rem,A           ; initialize R register in Figure 4.1
        movlw   8
        movwf   lp_cnt,A        ; initialize loop count to 8
loop    bcf     STATUS,C,A      ; clear the C flag
        rlcf    quo,F,A         ; rotate (R, Q) pair to the left one place
        rlcf    rem,F,A         ;       "
        movf    dsr,W,A
        subwf   rem,W,A         ; subtract and leave the difference in WREG
        btfss   STATUS,C,A      ; skip if carry is 1
```

```
            goto    negative
            bsf     quo,0,A         ; set the least significant bit of Q1 to 1
            movwf   rem,A           ; place the difference in R1
            goto    next
negative    bcf     quo,0,A         ; set the quotient bit to 0
next        decfsz  lp_cnt,F,A      ; decrement the loop count and skip if zero
            goto    loop
            btfss   sign,2,A        ; skip if sign of quotient is negative
            goto    check_re        ;       "
            negf    quo,A
check_re    btfss   sign,1,A        ; skip if dividend is negative
            goto    ok_skip         ;       "
            negf    rem,A
ok_skip     nop
            end
```

The Stack

- A first-in-last-out data structure.
- Has a stack pointer that points to the top byte or the byte above its top byte.
- A block of RAM of adequate size is required.
- The most common operations performed on the stack are **push** and **pop**.
- The push operation grows the stack while the pop operation shrinks the stack.
- A stack is often used to hold return address for subroutine calls or interrupt handling and temporary storage.

PIC18 Software Stack

- PIC18 does not have a hardware stack (no dedicated stack pointer).

- One of the FSR registers can be used as the stack pointer. By convention, FSR1 is used as the stack pointer by the PIC18 C compiler.

- The PIC18 stack is shown in Figure 4.2

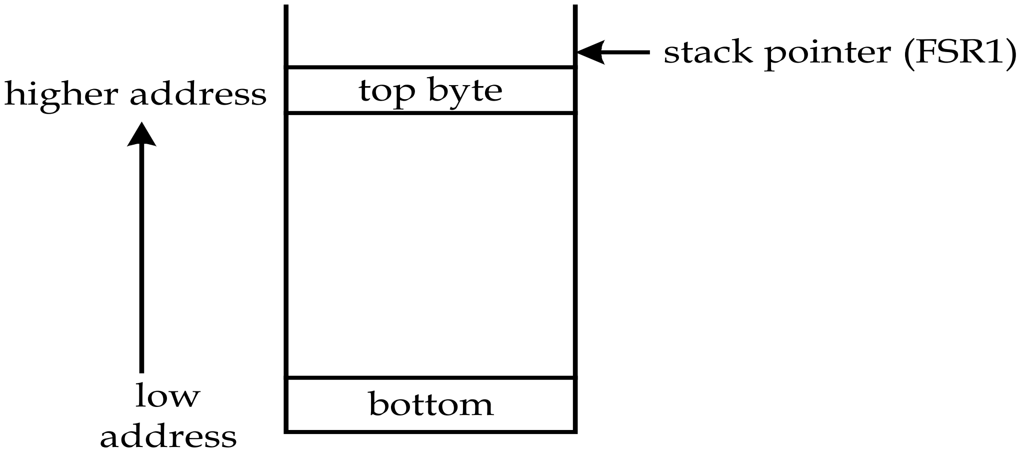

Figure 4.2 The PIC18 Software Stack

Stack Macros

- **Push** and **pop** operations are often performed during a subroutine call.
- The user often needs to allocate and deallocate stack space during a subroutine call.
- Allocation and deallocation of stack space are performed during subroutine calls.

```
pushr   macro   arg                 ; macro to push the arg register
        movff   arg,POSTINC1
        endm

popr    macro   arg                 ; arg is a file register
        movff   POSTDEC1,arg        ; decrement the stack pointer
        movff   INDF1,arg           ; pop off a byte from the stack onto arg
        endm

push_dat
        macro   dat                 ; this macro push the value dat onto the
stack
        movlw   dat
        pushr   WREG
        endm
```

Stack Macros (continued)

```
alloc_stk    macro   n            ; this macro allocates n bytes in stack
             movlw   n
             addwf   FSR1L,F,A
             movlw   0x00
             addwfc  FSR1H,F,A
             endm

dealloc_stk          macro        n     ; this macro deallocates n bytes from stack
             movlw   n
             subwf   FSR1L,F,A
             movlw   0
             subwfb  FSR1H,F,A    ; subtract borrow from high byte of FSR1
             endm
```

Examples of Stack Macros Applications

1. Push PRODL into the software stack

   pushr PRODL

2. Pop PRODL from the software stack

   popr PRODL

3. Push the value 0x20 onto the software stack

   push_dat 0x20

4. Allocate 10 bytes of stack space

   alloc_stk 10

5. Deallocate 10 bytes from the stack

   dealloc_stk 10

Subroutines

- A sequence of instructions that can be called from many different places in a program.
- Parameters can be passed to the subroutine so that it can perform operations on different variables.
- Allows the user to use divide-and-conquer strategy to solve complicated problems.
- Using subroutines can in some cases reduce the code size significantly.

Structured Programming methodology

- Start with a main program that outlines the logical flow of the algorithm
- Assign program details to subroutines
- A subroutine may also call other subroutines

PIC18 Mechanisms to Support Subroutine Calls

- A 31-slot **return address stack** for saving and restoring return address

- A **return address stack pointer** points to the top of the return address stack.

- The **push** and **pull** instructions for pushing values onto the return address stack and pulling values into program counter from the return address stack

- A one-layer deep **fast register stack** for fast saving and restoring the STATUS, WREG, and BSR registers during interrupts and subroutine calls

- The **rcall n** and **call k[,s]** instructions for making subroutine calls.

- The **return [s]** instruction for returning from the subroutine

- The **retlw n** instruction for table lookup

- The **s** bit option allows one to save WREG, STATUS, and BSR in the fast register stack when making subroutine calls and restore them before returning from the subroutine

The CALL n[,s] Instruction

- A 2-cycle, 32-bit instruction
- **n** is a 20-bit value that specifies the word address of the subroutine.
- The **s** bit allows the PIC18 to save WREG, STATUS, BSR in the fast register stack.

The RCALL n instruction

- **n** is an 11-bit signed integer that specifies the distance of the subroutine to be called.
- Allows the PIC18 to call a subroutine that is 1-KW away
- The address of the subroutine to be called is at PC+2+2n.
- A 16-bit, two-cycle instruction

The return [s] instruction

- Return from subroutine
- When **s** is 1, the contents of the fast return stack will be popped off to WREG, STATUS, and BSR registers.
- The top entry of the return address stack will be popped off and loaded into the PC.

The retlw k instruction

- The WREG register is loaded with k.
- The top entry of the return address stack will be popped off and loaded into the PC.

Examples of Subroutine Calls and Returns

```
        call    sub_x, FAST     ; save BSR, STATUS, and WREG in fast
        …                       ; register stack
        …
sub_x   …
        …
        return  FAST            ; restore values saved in fast register stack
```

Table Lookup Instruction: retlw

```
        movlw   i               ; prepare to lookup the ith entry
        call    look_up
        …
        …
look_up addwf   PCL             ; jump to one of the following entries
        retlw   k0
        retlw   k1
        …
        retlw   kn              ; end of table
```

Issues Related to Subroutine Calls

- **Parameter passing.**
  1. Use general-purpose registers
  2. Use the stack
  3. Use global memory (same as method 1 for PIC18)
- **Result returning**
  1. Use general-purpose registers
  2. Use the stack
  3. Use global memory (same as method 1 for PIC18)
- **Allocation and deallocation of local variables**
  1. Local variables are allocated when the subroutine is entered.
  2. Local variables are deallocated before returning from the subroutine

The Stack Frame

- Stack frame is a data structure that holds the incoming parameters, saved registers, and local variables for the subroutine.
- A frame pointer is beneficial for managing the stack frame.
- The **stack pointer** may change during the lifetime of a subroutine, which sometimes makes the access of stack element tricky.
- The **frame pointer** points to a fixed location in the stack and does not change during the lifetime of the subroutine.
- By convention, the FSR2 register is used as the frame pointer.
- The PIC18 stack frame is shown in Figure 4.5.

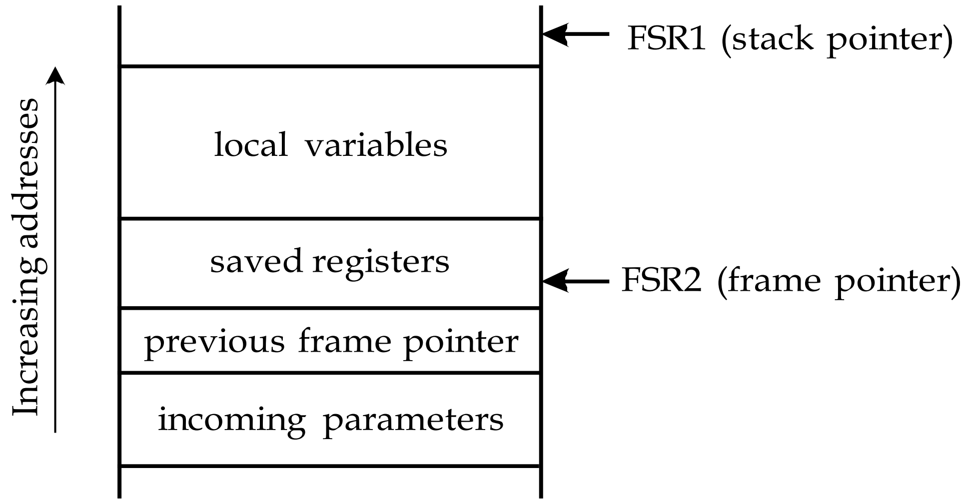

Figure 4.5 PIC18 Stack frame

Functioning of the Stack Frame

- Upon entry to a subroutine, the value of FSR2 is pushed onto the stack and the value of FSR1 is copied into FSR2.
- Some registers are pushed onto the stack.
- Local variables are allocated in the stack.
- Incoming parameters have negative offsets relative to the FSR2 register.
- Local variables have positive offsets relative to FSR2.

A stack frame example

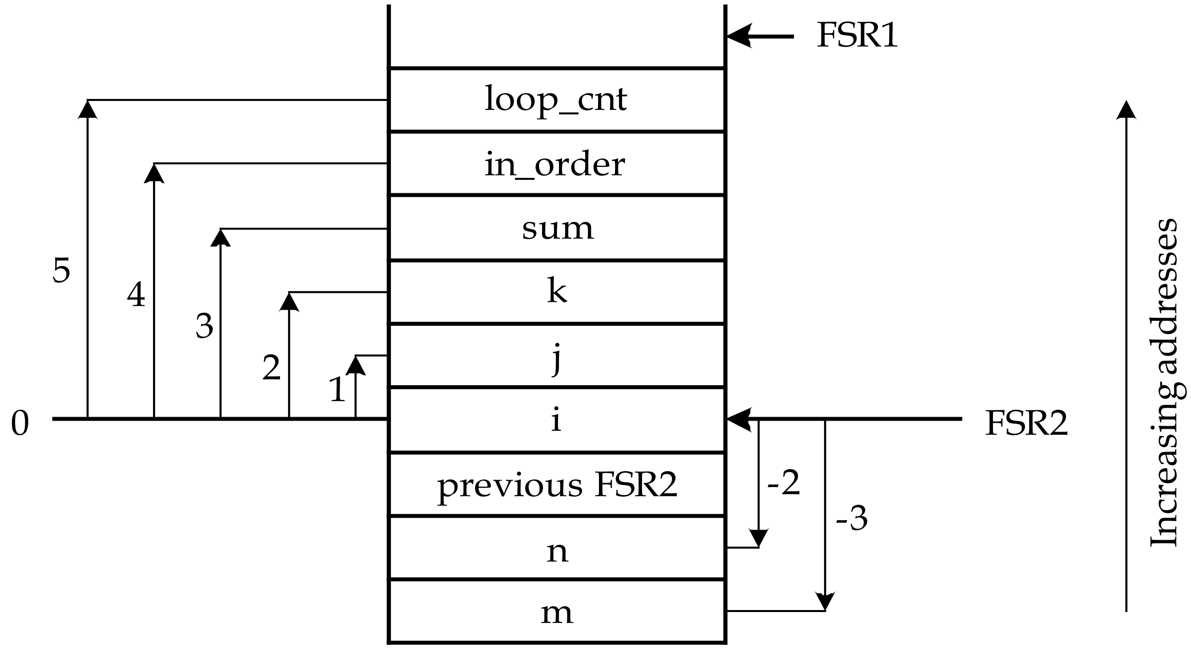

Figure 4.6 Stack frame and variable offsets

Accessing Locations in the Stack Frame

- Each stack frame slot has an offset relative to the slot pointed to by FSR2
- Assign a name to each stack frame slot by using the **equ** directive
- Execute the **movf PLUSW2,W** instruction

```
loop_cnt  equu   5     ; offset of loop_cnt from frame pointer
in_order  equ    4     ; offset of in_order from frame pointer
sum       equ    3     ; offset of sum from frame pointer
k         equ    2     ; offset of k from frame pointer
j         equ    1     ; offset of j from frame pointer
i         equ    0     ; offset of i from frame pointer
n         equ    -2    ; offset of n from frame pointer
m         equ    -3    ; offset of m from frame pointer
; loading  sum into WREG
          movlw  sum
          movf   PLUSW2,W
; loading m into WREG
          movlw  m
          movf   PLUSW2,W
```

Example 4.9 Write a subroutine to find the maximum element of an array of 8-bit elements in program memory. Pass the array count and starting address in the software stack.

Solution:

The caller of this subroutine will

1. Push the low part of the array starting address onto the stack.
2. Push the high part of the array starting address onto the stack.
3. Push the upper part of the array starting address onto the stack.
4. Push the array count onto the stack.

The subroutine will

1. Push FSR2L and FSR2H onto the stack
2. Copy FSR1 to FSR2
3. Push TBLPTRL, TBLPTRH, and TBLPTRU onto the stack
4. Allocate two bytes in the stack for local variables (one byte for **lp_cnt**, and one byte for **ar_max**

These operations need planning and thinking.

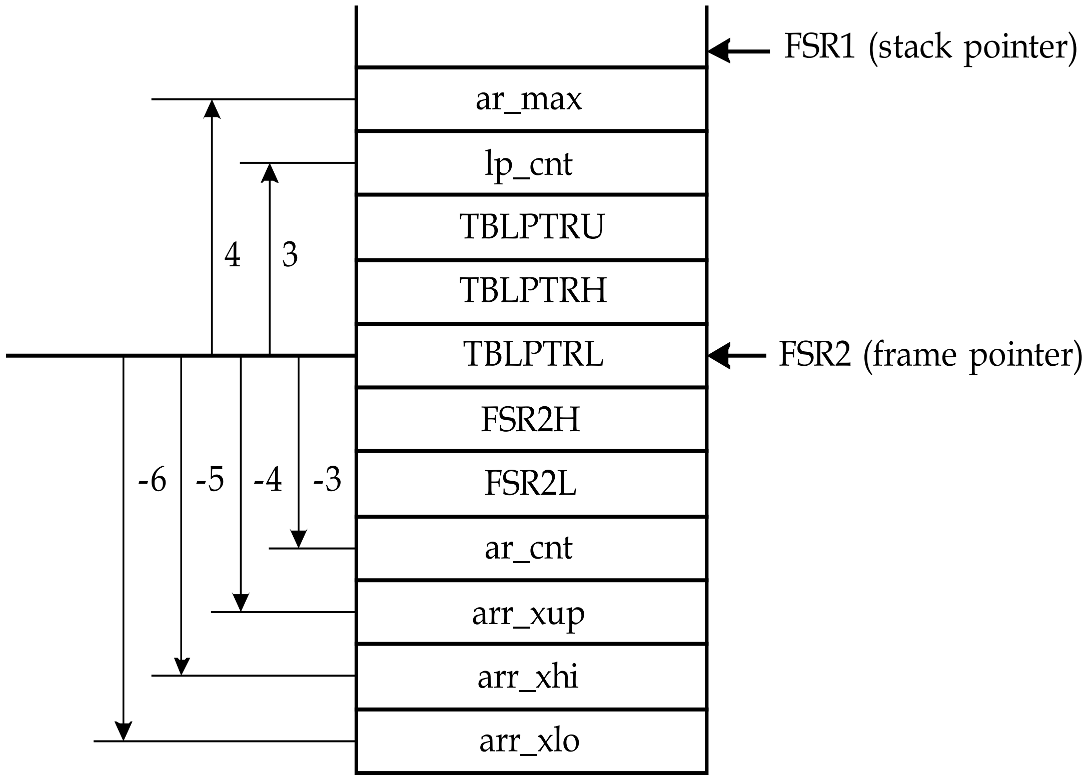

Figure 4.7 Stack frame for Example 4.9

```
          #include <p18F8720.inc>
          radix    dec
acnt      equ      32             ; array count
lp_cnt    equ      3              ; stack frame offset for lp_cnt
ar_max    equ      4              ; stack frame offset for array max
ar_cnt    equ      -3             ; stack frame offset for array count
arr_xup   equ      -4             ; stack frame offset for array base
arr_xhi   equ      -5             ;       "
arr_xlo   equ      -6             ;       "
array_max set      0x00           ; register to hold array max
          org      0x00           ; reset vector
          goto     start
          org      0x08
          retfie
          org      0x18
          retfie
start     lfsr     FSR1,0xE00     ; set up software stack pointer
          movlw    low arr_x      ; pass array base in the stack
          pushr    WREG           ;       "
          movlw    high arr_x     ;       "
          pushr    WREG           ;       "
          movlw    upper arr_x    ;       "
```

```
            pushr       WREG                ;           "
            movlw       acnt                ; pass array count in stack
            pushr       WREG                ;           "
            call        find_amax,FAST      ; call the subroutine
            movff       PRODL,array_max               ; save the result
            dealloc_stk             4                 ; clean up the allocated stack
space
            nop
here        bra         here
; *********************************************************************
; This subroutine finds the maximum element of an 8-bit array and returns it in
PRODL.
; *********************************************************************
find_max    pushr       FSR2L               ; save caller's frame pointer
            pushr       FSR2H               ;           "
            movff       FSR1L,FSR2L                   ; set up new frame pointer
            movff       FSR1H,FSR2H         ;           "
            pushr       TBLPTRL             ; save table pointer in stack
            pushr       TBLPTRH             ;           "
            pushr       TBLPTRU             ;           "
            alloc_stk   2                   ; allocate 2 bytes for local variables
            movlw       ar_cnt
            movff       PLUSW2,PRODL        ; copy array count into PRODL
```

```
        decf      PRODL,F,A
        movlw     lp_cnt
        movff     PRODL,PLUSW2     ; initialize lp_cnt to arcnt-1
        movlw     arr_xlo          ; place array pointer in TBLPTR
        movff     PLUSW2,TBLPTRL   ;          "
        movlw     arr_xhi          ;          "
        movff     PLUSW2,TBLPTRH   ;          "
        movlw     arr_xup          ;          "
        movff     PLUSW2,TBLPTRU   ;          "
        tblrd*+                    ; assign arr_x[0] as the initial array max
        movlw     ar_max           ;          "
        movff     TABLAT,PLUSW2    ;          "
cmp_lp  movlw     lp_cnt
        tstfsz    PLUSW2
        goto      next
        goto      done
next    movlw     lp_cnt
        decf      PLUSW2,F         ; decrement the loop index
        tblrd*+                    ; read in the next array element
        movlw     ar_max
        movf      PLUSW2,W,A
        cpfsgt    TABLAT
```

```
        movlw       ar_max
        movff       TABLAT,PLUSW2    ; update the current array max
        goto        cmp_lp
done    movlw       ar_max
        movff       PLUSW2,PRODL
        dealloc_stk                  2; deallocate local variables
        popr        TBLPTRU
        popr        TBLPTRH
        popr        TBLPTRL
        popr        FSR2H
        popr        FSR2L
        return      FAST
; define an array of 32 8-bit elements.
arr_x   db          1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16
        db          17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32
        end
```

Binary Search

- Searching an array or a file is a common operation.
- An efficient search algorithm can be applied to a sorted array or file.
- Binary search is a very efficient search algorithm.
- Divide the sorted array into three parts: upper half, middle element, and lower half.
- Use three array indices to guide the search of the array: min, mean, and max.
- The index **min** refers to the lowest array index for searching.
- The index **max** refers to the highest array index for searching.
- The index **mean** is the average of **min** and **max**.

Binary Search Algorithm

Step 1
Use *max* and *min* as the upper and lower indices of the array for searching and initialize *max* and *min* to **n – 1** and **0**, respectively.
Step 2
If *max* < *min*, then stop. No element matches the key.
Step 3
Let *mean* **=** (min + max)/2.
Step 4
If key **=** array_y[mean], then key is found; stop.
Step 5
If key **<** array_y[mean], then set *max* to **mean –1** and go to Step 2.
Step 6
If key **>** array_y[mean], then set *min* to **mean +1** and go to Step 2.

Example 4.10 Write a subroutine to implement the binary search algorithm. The starting address of the array, array count, and search key are passed in the stack. A value of 1 will be returned in PRODL if the key is found in the array. Otherwise, a value of 0 will be returned.

Solution:

- This subroutine saves the table pointer in the stack and allocates four bytes for local variables.

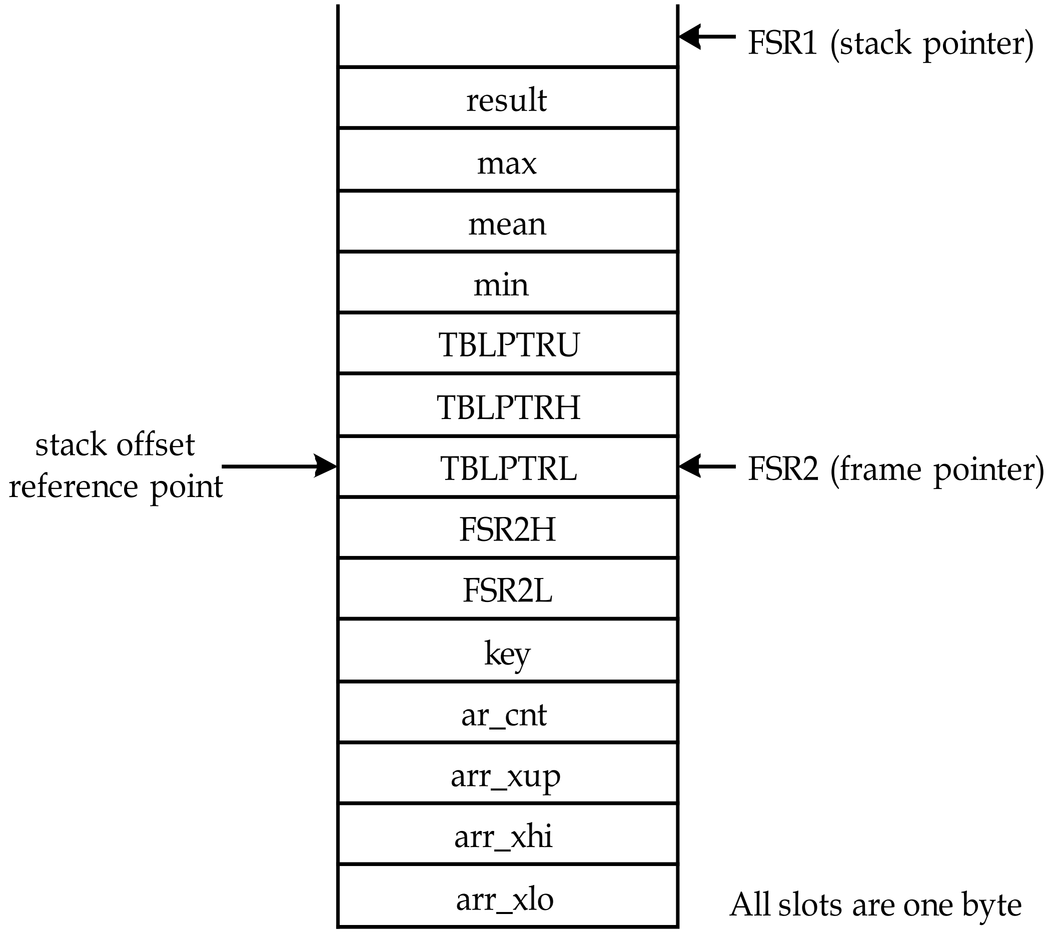

Figure 4.8 Stack frame of example 4.10

Binary Search Subroutine and Test Program

```
            #include <p18F8720.inc>
            radix    dec
; -----
; include stack macros pushr, push_dat, popr, alloc_stk, and dealloc_stk here
; -----
acnt        equ      32              ; array count
local_var   equ      4               ; number of bytes for local variables
skey        equ      7               ; key for search
min         equ      3               ; offset for lower bound of search range index
mean        equ      4               ; offset for middle element index
max         equ      5               ; offset for upper bound of search range index
result      equ      6               ; offset for search result
key         equ      -3              ; offset for search key
ar_cnt      equ      -4              ; stack frame offset for array count
arr_xup     equ      -5              ; stack frame offset for array base
arr_xhi     equ      -6              ;     "
arr_xlo     equ      -7              ;     "
find_it     set      0x00            ; register to hold array max
            org      0x00            ; reset vector
            goto     start
            org      0x08
            retfie
```

```
        org         0x18
        retfie
start   lfsr        FSR1,0xE00      ; set up software stack pointer
        movlw       low arr_x       ; pass array base in the stack
        pushr       WREG            ;           "
        movlw       high arr_x      ;           "
        pushr       WREG            ;           "
        movlw       upper arr_x     ;           "
        pushr       WREG            ;           "
        movlw       acnt            ; pass array count in stack
        pushr       WREG            ;           "
        movlw       skey            ; pass key for search
        pushr       WREG
        call        bin_search,FAST ; call the subroutine
        movff       PRODL,find_it   ; save the result
        dealloc_stk                 5           ; clean up the stack space used by paran
        nop
here    bra         here

bin_search          pushr           FSR2L       ; save callers frame pointer
        pushr       FSR2H           ;           "
        movff       FSR1L,FSR2L     ; set up frame pointer
        movff       FSR1H,FSR2H     ;           "
```

```
        pushr      TBLPTRL          ; save table pointer in stack
        pushr      TBLPTRH          ; "
        pushr      TBLPTRU          ; "
        alloc_stk  local_var        ; allocate space for local variables
        movlw      ar_cnt           ; max ← (ar_cnt – 1)
        movff      PLUSW2,PRODL     ; "
        decf       PRODL,F,A        ; "
        movlw      max              ; "
        movff      PRODL,PLUSW2     ;  "
        movlw      min              ; min ← 0
        clrf       PLUSW2,A         ;          "
        movlw      result
        clrf       PLUSW2           ; set the search result to not found
bloop   movlw      max
        movff      PLUSW2,PRODL     ; place max index in PRODL
        movlw      min
        movff      PLUSW2,WREG      ; place min index in W
        cpfslt     PRODL            ; is max < min?
        goto       do_it            ; max >= min and continue
        goto       done             ; no match is found in the array
do_it   addwf      PRODL,F,A        ; compute (min+max), this may set C flag
        bcf        STATUS,C         ; compute (min+max)/2 and place it
```

```
        rrcf      PRODL,F,A        ; in PRODL
        movlw     mean
        movff     PRODL,PLUSW2     ; place mean index in the stack frame slot
; compare array[mean] with the search key
        movlw     arr_xlo          ; place array base in TBLPTR
        movff     PLUSW2,TBLPTRL   ;        "
        movlw     arr_xhi          ;        "
        movff     PLUSW2,TBLPTRH   ;        "
        movlw     arr_xup          ;        "
        movff     PLUSW2,TBLPTRU   ;        "
; add mean to TBLPTR to compute the address of arr_x[mean]
        movf      PRODL,W,A        ; add lower 8 bits of mean to
        addwf     TBLPTRL,F,A      ; TBLPTR
        movlw     0                ;        "
        addwfc    TBLPTRH,F,A      ;        "
        addwfc    TBLPTRU,F,A      ;        "
        tblrd*                     ; read array[mean] into TABLAT
; compare key with array[mean]
        movlw     key
        movf      PLUSW2,W,A       ; place key in WREG
        cpfseq    TABLAT,A
        goto      not_equal
```

```
            movlw       result
            incf        PLUSW2,F       ; set search result to "found"
            goto        done
not_equal   cpfslt      TABLAT,A
            goto        go_low
            movlw       mean
            movff       PLUSW2,PRODL
            incf        PRODL,F,A
            movlw       min
            movff       PRODL,PLUSW2   ; set min to mean+1 to
            goto        bloop          ; search upper half
go_low      movlw       mean
            movff       PLUSW2,PRODL
            decf        PRODL,F,A
            movlw       max
            movff       PRODL,PLUSW2   ; set max to mean - 1 and
            goto        bloop          ; search lower half
done        movlw       result
            movff       PLUSW2,PRODL   ; place the result in PRODL before return
            dealloc_stk                local_var  ; deallocate local variables
            popr        TBLPTRU
            popr        TBLPTRH
```

```
        popr    TBLPTRL
        popr    FSR2H
        popr    FSR2L
        return  FAST
; define an array of 32 8-bit elements for testing purpose.  The element arr_x[i] is
; located at arr_x[0] + i
arr_x   db      1,2,3,4,5,6,7,8,9,10
        db      11,12,13,14,15,16,17,18,19,20
        db      21,22,23,24,25,26,27,28,29,30
        db      31,32
        end
```

Example 4.11 Convert the unsigned 16-bit division program into a subroutine. The caller of this subroutine will push the dividend and divisor into the stack and make a hole in the stack for this subroutine to return the remainder and the quotient.
Solution:

The caller of this subroutine will do the following
- Push the dividend onto the stack
- Make a hole of two bytes to hold the remainder
- Push the divisor onto the stack

The **divu16** subroutine will do the following
- Push the FSR2 onto the stack
- Copy FSR1 to FSR2 (set up new frame pointer)
- Save PRODL in the stack
- Allocate space for local variables

The stack frame of this subroutine is shown in Figure 4.9.

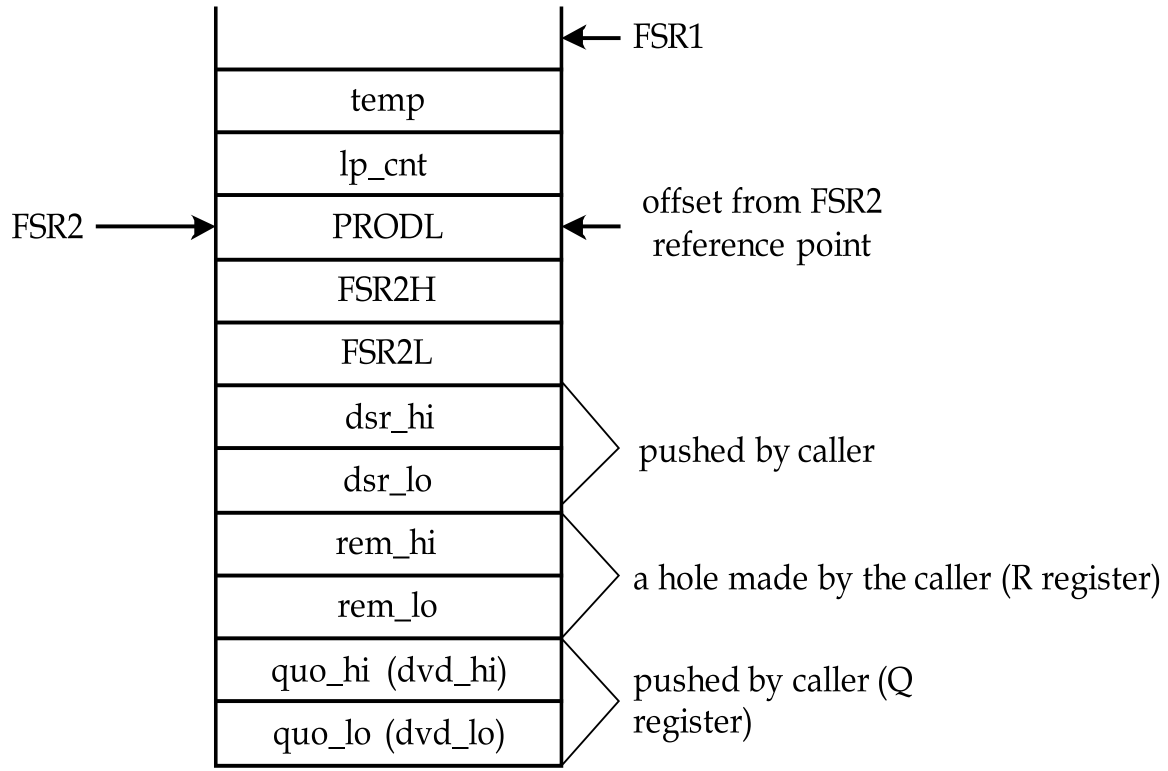

Figure 4.9 Stack frame of example 4.11

```
            #include <p18F8720.inc>
            radix       dec
; ---
; include macros pushr, popr, alloc_stk, and dealloc_stk here
; ---
loc_var     equu        2           ; local variable size
lp_cnt      equ         1           ; loop count
temp        equ         2           ; temporary storage
quo_hi      equ         -7          ; offset for quotient and dividend
quo_lo      equ         -8          ;       "
rem_hi      equ         -5          ; offset for remainder
rem_lo      equ         -6          ;       "
dsr_hi      equ         -3          ; offset for divisor
dsr_lo      equ         -4          ;       "
dd_h        equ         0xEC        ; high byte of dividend test number
dd_l        equ         0x46        ; low byte of "
dr_h        equ         0x00        ; high byte of divisor test number
dr_l        equ         0x87        ; low byte of "
quo         set         0x00        ; memory space to hold the quotient
rem         set         0x02        ; memory space to hold the remainder
            org         0x00
            goto        start
```

```
div16u  pushr      FSR2L            ; save the previous frame pointer
        pushr      FSR2H            ;   "
        movff      FSR1L,FSR2L      ; set up frame pointer
        movff      FSR1H,FSR2H      ;   "
        pushr      PRODL            ; save PRODL in stack
        alloc_stk  loc_var          ; allocate space for local variables
        movlw      rem_hi           ; clear R register to 0
        clrf       PLUSW2           ;   "
        movlw      rem_lo           ;   "
        clrf       PLUSW2           ;   "
        movlw      16               ; initialize loop count to 16
        movwf      PRODL            ;   "
        movlw      lp_cnt           ;   "
        movff      PRODL,PLUSW2;   "
loop    bcf        STATUS,C,A       ; clear the C flag
        movlw      quo_lo           ; rotate (R, Q) pair to the left one place
        rlcf       PLUSW2,F         ;   "
        movlw      quo_hi           ;   "
        rlcf       PLUSW2,F         ;   "
        movlw      rem_lo           ;   "
        rlcf       PLUSW2,F         ;   "
        movlw      rem_hi           ;   "
```

```
rlcf    PLUSW2,F        ; "
movlw   rem_lo          ; get the low byte of the remainder in
movff   PLUSW2,PRODL    ; PRODL
movlw   dsr_lo          ; get the low  byte of the divisor in WREG
movf    PLUSW2,W        ;               "
subwf   PRODL,F,A
movlw   temp
movff   PRODL,PLUSW2    ; save the low byte of difference at temp
movlw   rem_hi
movff   PLUSW2,PRODL
movlw   dsr_hi
movf    PLUSW2,W
subwfb  PRODL,F         ; subtract the high byte
btfss   STATUS,C        ; skip if carry is 1
goto    less
movlw   quo_lo
bsf     PLUSW2,0        ; set the quotient bit to 1
movlw   rem_hi
movff   PRODL,PLUSW2    ; place the difference in the R register
movlw   temp            ;               "
movff   PLUSW2,PRODL    ; "
movlw   rem_lo          ; "
```

```
            movff       PRODL,PLUSW2  ;          "
            goto        next
less        movlw       quo_lo
            bcf         PLUSW2,0        ; set the quotient bit to 0
next        movlw       lp_cnt
            decfsz      PLUSW2,F        ; decrement the loop count and skip if zero
            goto        loop
            dealloc_stk loc_var         ; deallocate local variables
            popr        PRODL
            popr        FSR2H
            popr        FSR2L
            return      FAST
            end
```

String Processing

- A string is a sequence of characters terminated by a NULL character.
- A character is encoded in ASCII code.
- Strings are needed in input and output.
- A number must be converted to ASCII code before it can be output.
- The decimal number 10805 will be converted to 0x31 0x30 0x38 0x30 0x35
- A binary number can be converted to its equivalent decimal string by performing repeated division by 10.
- The remainder of the first division is the least significant digit, the last division will separate the most significant digit and second most significant digit.

Example 4.12 Write a subroutine to convert a 16-bit signed number to an ASCII string
that represents its equivalent decimal value.

Solution: Let **p**, **rem**, **quo**, and **sign** represent the integer to be converted, remainder after the divide-by-10 operation, the quotient after the divide-by-10 operation, and the sign of **p**, respectively. The algorithm for converting a binary number into a decimal string is as follows:

Step 1
If p = 0, then store $30 and $00 in the buffer and stop.

Step 2
If p < 0, then set sign to 1 and compute p's 2's complement. Otherwise, set sign to 0.

Step 3
Divide p by 10 and leave the remainder and the quotient in **rem** and **quo**, respectively.

Step 4
Add $30 to the remainder and save it in the next available space in the buffer.

Step 5
If quo ≠ 0 then p ← quo and go to Step 3.

Step 6
Push the characters in the buffer into the stack.

Step 7
If sign is 1, then store the ASCII code of the minus sign as the first character in the buffer.

Step 8
Pop out all the decimal digits from the stack into the buffer.

The stack frame of this subroutine is in Figure 4.10.

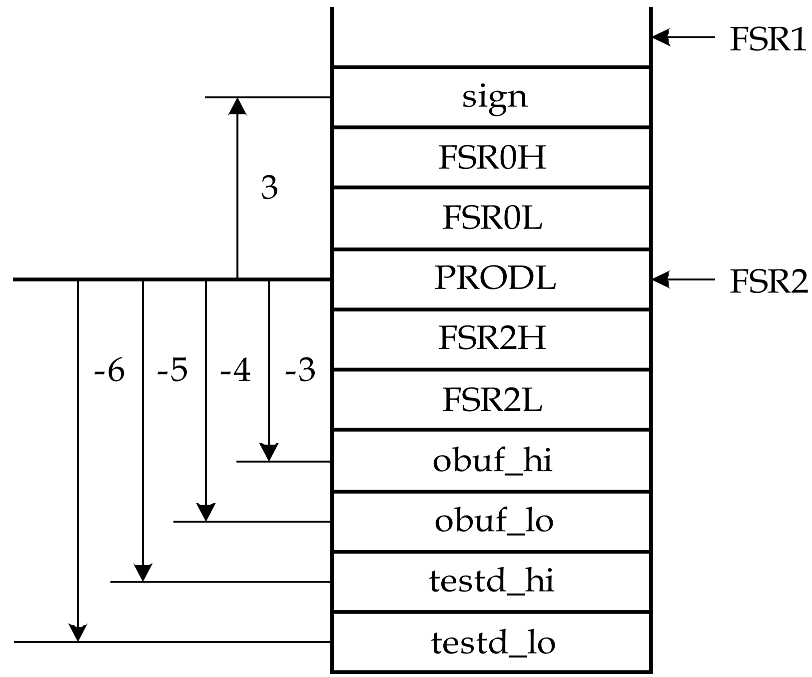

Figure 4.10 Stack frame for example 4.12

```
            #include <p18F8720.inc>
            radix     dec                 ; set default radix to decimal
; ---
; include macro definitions of pushr, popr, push_dat, alloc_stk, and dealloc_stk here
; ---
tstd_hi     equu      0x88                ; test data high byte
tstd_lo     equ       0x89                ; test data low byte
sign        equ       3                   ; stack offset for sign in bin2dec
testd_lo    equ       -6                  ; stack offset for test data in bin2dec
testd_hi    equ       -5                  ;       "
obuf_lo     equ       -4                  ; stack offset for conversion buffer
obuf_hi     equ       -3                  ;       "
minus       equ       0x2D                ; ASCII code of minus sign
loc_varc    equ       1                   ; local variable size of bin2dec routine
obuf        set       0x0100
            org       0x00
            goto      start
```

The test program is as follows:

```
            org         0x08            ; high-priority interrupt vector
            retfie
            org         0x18            ; low-priority interrupt vector
            retfie
start       lfsr        FSR1,0xE00      ; set up stack pointer
            push_dat    tstd_lo         ; pass test data in stack
            push_dat    tstd_hi         ;          "
            push_dat    low obuf        ;          "
            push_dat    high obuf       ;          "
            call        bin2dec,FAST    ; call bin2dec subroutine
            dealloc_stk                 4          ; clean up stack space
con_loop                nop
            bra         con_loop
```

```
bin2dec  pushr      FSR2L            ; save previous frame pointer
         pushr      FSR2H            ;   "
         movff      FSR1L,FSR2L      ; set up new frame pointer
         movff      FSR1H,FSR2H      ;                  "
         pushr      PRODL
         pushr      FSR0L            ; save FSR0 in stack
         pushr      FSR0H            ;   "
         alloc_stk  loc_varc         ; allocate local variable space
; set FSR0 as the pointer to the buffer that holds the conversion result
         movlw      obuf_lo
         movff      PLUSW2,FSR0L
         movlw      obuf_hi
         movff      PLUSW2,FSR0H
; initialize the sign of the number to be converted to positive
         movlw      sign             ;   "
         clrf       PLUSW2           ;   "
; test to find out if the number is negative. If yes, compute its
; two's complement.
         movlw      testd_hi
         btfss      PLUSW2,7         ; compute the magnitude when negative
         goto       tst_zero         ; no need to complement when positive
         comf       PLUSW2,F         ; complement the upper byte when negative
```

```
        clrf    PRODL,A         ; add carry to testd_hi (carry may be 0)
        movlw   testd_lo        ; compute the two's complement of data
        negf    PLUSW2          ; to be converted
        movlw   testd_hi        ;           "
        movf    PLUSW2,W        ; "
        addwfc  PRODL,F         ; "
        movlw   testd_hi        ; "
        movff   PRODL,PLUSW2    ; "
; change sign to 1 to indicate minus
        movlw   sign            ;           "
        incf    PLUSW2,F        ;           "
        bra     normal
; check if the number to be tested is zero
tst_zero movlw  testd_hi        ; test high byte
        tstfsz  PLUSW2          ;           "
        bra     normal
        movlw   testd_lo        ; test low byte
        tstfsz  PLUSW2          ; "
        bra     normal
        movlw   0x30
        movwf   POSTINC0
        clrf    INDF0           ; terminate the buffer with NULL character
```

```
            goto       done
; normal repeated divide-by-10 loop starts here
normal      movlw      testd_lo            ; pass the dividend
            movf       PLUSW2,W            ;          "
            pushr      WREG                ;          "
            movlw      testd_hi            ;          "
            movf       PLUSW2,W            ;          "
            pushr      WREG                ;          "
div_lp      alloc_stk  2                   ; make room for remainder
            push_dat   10                  ; push 10 into the stack as divisor
            push_dat   0                   ;          "
            call       div16u              ; call subroutine to perform division
            dealloc_stk                    3          ; deallocate the stack space (3 bytes rem
            popr       WREG                ; pop off low byte of remainder
            addlw      0x30                ; convert the remainder into ASCII code
            movwf      POSTINC0            ; and save it in buffer
; is quotient equal to 0?
            movlw      -1
            tstfsz     PLUSW1
            goto       div_lp
            movlw      -2
            tstfsz     PLUSW1
```

```
            goto        div_lp
            clrf        INDF0                   ; terminate the buffer with NULL
            dealloc_stk                         2           ; clean up the stack
; set FSR0 to point to the start of the buffer to reverse the string
            movlw       obuf_lo                 ;           "
            movff       PLUSW2,FSR0L            ;           "
            movlw       obuf_hi                 ;           "
            movff       PLUSW2,FSR0H            ;           "
            push_dat    0                       ; push a NULL character onto the stack
push_lp     movf        POSTINC0,W
            bz          to_pop                  ; is this the NULL character
            pushr       WREG
            goto        push_lp
; reverse the converted string
to_pop      movlw       obuf_lo                 ; set FSR0 to point to the start of buffer
            movff       PLUSW2,FSR0L            ;           "
            movlw       obuf_hi                 ;           "
            movff       PLUSW2,FSR0H            ;           "
            movlw       sign
            movf        PLUSW2,W                ; is the converted data negative?
            bz          pop_loop                ;           "
; add a minus sign if the number is negative
```

```
         movlw   minus
         movwf   POSTINC0
pop_loop popr    WREG            ; reverse string loop
         movwf   POSTINC0
         tstfsz  WREG,A          ; reach the end of string?
         goto    pop_loop
done     popr    WREG            ; get rid of sign
         popr    FSR0H
         popr    FSR0L
         popr    PRODL
         popr    FSR2H
         popr    FSR2L
         return  FAST
```

Example 4.13 Write a subroutine that computes the product of two unsigned 16-bit integers. The numbers to be multiplied and the pointer to the buffer for storing the product are passed in the stack.

Solution: The subroutine will save all the used registers in the stack and allocates four bytes for local variables to hold the product. The stack frame is shown in Figure 4.11.

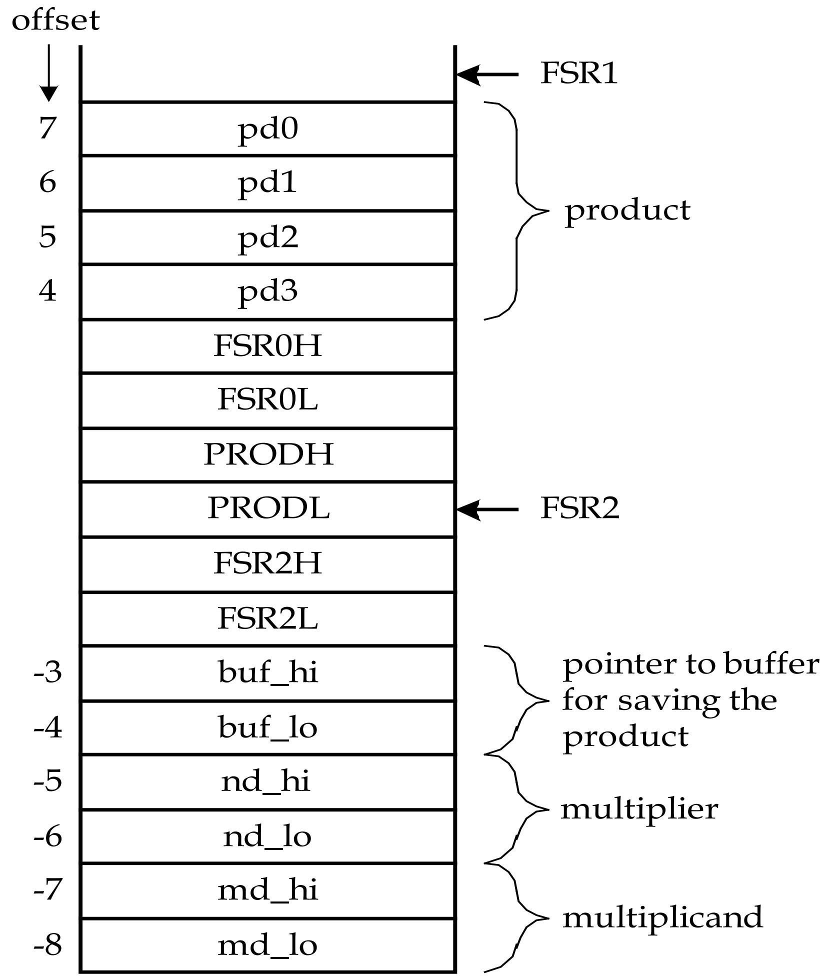

Figure 4.11 Stack frame for Example 4.13

```
mul_16u pushr     FSR2L            ; save previous frame pointer
        pushr     FSR2H            ;   "
        movff     FSR1L,FSR2L      ; set up new frame pointer
        movff     FSR1H,FSR2H      ;   "
        pushr     PRODL            ; save PRODL in stack
        pushr     PRODH            ; save PRODH in stack
        pushr     FSR0L            ; save FSR0 in stack
        pushr     FSR0H            ;   "
        alloc_stk loc_varm
; compute md_lo × nd_lo and place in pd1..pd0
        movlw     md_lo
        movff     PLUSW2,PRODL     ; place md_lo in PRDOL
        movlw     nd_lo
        movf      PLUSW2,W         ; place nd_lo in WREG
        mulwf     PRODL            ; compute nd_lo * md_lo
        movlw     pd0
        movff     PRODL,PLUSW2
        movlw     pd1
        movff     PRODH,PLUSW2
; compute md_hi × nd_hi and place the product in pd3..pd2
        movlw     md_hi
        movff     PLUSW2,PRODL     ; place md_hi in PRODL
```

```
        movlw   nd_hi
        movf    PLUSW2,W          ; place nd_hi in WREG
        mulwf   PRODL             ; compute nd_hi * md_hi
        movlw   pd2
        movff   PRODL,PLUSW2
        movlw   pd3
        movff   PRODH,PLUSW2
; compute md_lo × nd_hi
        movlw   md_lo
        movff   PLUSW2,PRODL
        movlw   nd_hi
        movf    PLUSW2,W
        mulwf   PRODL
        movlw   pd1               ; add to pd1
        movf    PLUSW2,W          ;         "
        addwf   PRODL,F           ;         "
        movlw   pd1               ;         "
        movff   PRODL,PLUSW2      ;         "
        movlw   pd2
        movf    PLUSW2,W
        addwfc  PRODH,F
        movlw   pd2
```

```
        movff    PRODH,PLUSW2    ;
        clrf     PRODL
        movlw    pd3
        movf     PLUSW2,W
        addwfc   PRODL,F
        movlw    pd3
        movff    PRODL,PLUSW2
; compute md_hi × nd_lo
        movlw    md_hi
        movff    PLUSW2,PRODL
        movlw    nd_lo
        movf     PLUSW2,W
        mulwf    PRODL
        movlw    pd1             ; add to pd1
        movf     PLUSW2,W        ;          "
        addwf    PRODL,F         ;          "
        movlw    pd1             ;          "
        movff    PRODL,PLUSW2    ;          "
        movlw    pd2             ; add to pd2
        movf     PLUSW2,W        ;          "
        addwfc   PRODH,F         ;          "
        movlw    pd2             ;          "
```

```
        movff   PRODH,PLUSW2    ;           “
        clrf    PRODL           ; add carry to most significant byte
        movlw   pd3             ;           “
        movf    PLUSW2,W        ;           “
        addwfc  PRODL,F         ;           “
        movlw   pd3             ;           “
        movff   PRODL,PLUSW2    ;           “
; use FSR0 as a pointer to the buffer that holds the product
        movlw   buf_lo
        movff   PLUSW2,FSR0L
        movlw   buf_hi
        movff   PLUSW2,FSR0H
; save product in the buffer
        popr    WREG
        movwf   POSTINC0
        popr    WREG
        movwf   POSTINC0
        popr    WREG
        movwf   POSTINC0
        popr    WREG
        movwf   POSTINC0
        popr    FSR0H
```

```
popr     FSR0L
popr     PRODH
popr     PRODL
popr     FSR2H
popr     FSR2L
return   FAST
end
```

Input Data Conversion Issue

- A number entered from the keyboard is encoded in ASCII code.
- Each input character represents a decimal digit.
- The input number can be positive or negative.
- A negative number is entered using signed magnitude format.
- Illegal characters must be checked.
- The following algorithm uses three variables to facilitate the conversion:
  - **sign:** keeps track of the sign of the number
  - **error:** indicates whether the input string has any error
  - **number:** stores the converted number

ASCII String to Binary Conversion Algorithm

Step 1

sign ← 0
error ← 0
number ← 0

Step 2

If the character pointed to by *in_ptr* is the minus sign, then

sign ← 1
in_ptr ← in_ptr + 1

Step 3

If the character pointed to by *in_ptr* is the NULL character, then go to Step 4.
else if the character is not a BCD digit (i.e., m[in_ptr] > $39 or m[in_ptr] < $30), then

error ← 1;
go to Step 4;

else

number ← number × 10 + m[in_ptr] - $30;
in_ptr ← in_ptr + 1;
go to Step 3;

Step 4

If sign = 1 and error = 0, then

number ← two's complement of number;

else stop;

Example 4.14 Write a subroutine that converts a decimal digit string into the binary number that it represents. The string is stored in the data memory and its starting address is passed to this function in the FSR0 register. The string may represent a positive or a negative number and is no longer than 7 bytes. Thus, the converted binary number can be accommodated in two bytes. The converted result will be returned in PRODL and PRODH. Do not check for illegal characters.

Solution:

- This subroutine needs to call **mul_16u** subroutine to perform multiplication.
- This subroutine passes a pointer to a buffer to hold the product.
- The buffer is reserved in the stack frame.

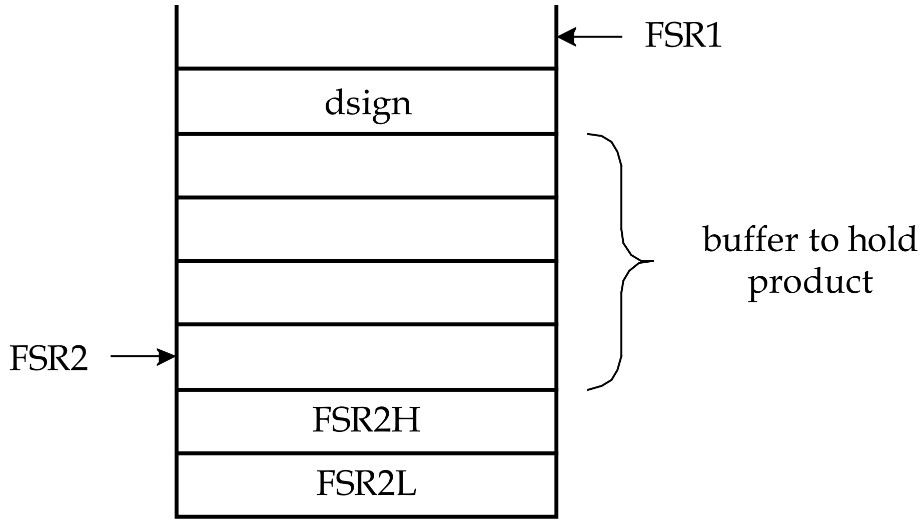

Figure 4.12 Stack frame for Example 4.14

The subroutine is as follows:

```
loc_vard   equ    5       ; size of local variables for d2b
dsign      equ    4       ; offset of dsign from FSR2
dec_strg   set    0x10    ; address of the buffer for the decimal string
```

```
dec2bin  pushr      FSR2L           ; save the current frame pointer
         pushr      FSR2H           ;   "
         movff      FSR1L,FSR2L     ; set up new frame pointer
         movff      FSR1H,FSR2H     ;   "
         alloc_stk  loc_vard        ; allocate 5 bytes for local variables
         movlw      0
         clrf       PLUSW2          ; clear the buffer that holds the product
         incf       WREG,W          ;   "
         clrf       PLUSW2          ;   "
         incf       WREG,W          ;   "
         clrf       PLUSW2          ;   "
         incf       WREG,W          ;   "
         clrf       PLUSW2          ;   "
         movlw      dsign           ; initialize sign to positive
         clrf       PLUSW2          ;   “
         movff      INDF0,PRODL     ; get a character from string
         movlw      0x2D            ; place minus sign in WREG
         cpfseq     PRODL           ; is the first character a minus sign?
         goto       con_lp
         movlw      dsign
         incf       PLUSW2          ; set sign to minus
         movf       POSTINC0,W      ; move pointer to next character
```

```
con_lp    movff       POSTINC0,PRODL
          tstfsz      PRODL,A            ; reach the NULL character?
          goto        normal             ; not yet
          goto        done               ; yes, reach NULL character
normal    movlw       0x30
          subwf       PRODL,F,A          ; convert to digit value
          movlw       0                  ; push multiplicand
          movf        PLUSW2,W           ;         "
          pushr       WREG               ;         "
          movlw       1                  ;         "
          movf        PLUSW2,W           ;         "
          pushr       WREG               ;         "
          push_dat    10                 ; push multiplier 10 into stack
          push_dat    0                  ;         "
          pushr       FSR2L              ; push buffer pointer
          pushr       FSR2H              ;         "
          call        mul_16U            ; multiply the current result by 10
          dealloc_stk 6                  ; clean up stack space
          movf        PRODL,W
          addwf       INDF2,F            ; add the converted digit to the product
          clrf        PRODL,A            ;         "
          movlw       1
```

```
        movf        PLUSW2,W         ; place the upper byte of the product in WREG
        addwfc      PRODL,F          ;          "
        movlw       1                ;          "
        movff       PRODL,PLUSW2     ;          "
        goto        con_lp
; check the sign of the converted binary number before return
done    dealloc_stk                  3          ; move down the stack pointer
        popr        PRODH            ; place upper byte of product in PRODH
        popr        PRODL            ; place lower byte of product in PRODL
        movlw       dsign            ; check the sign
        tstfsz      PLUSW2           ;          "
        goto        negate
        goto        getback
negate  negf        PRODL            ; find 2's complement of the result
        comf        PRODH            ;          "
        movlw       0                ;          "
        addwfc      PRODH,F          ;          "
getback popr        FSR2H
        popr        FSR2L
        return
; ----
; include MUL16U and its associated equ directives here.
; ----
```

Bubble Sort

- Sorting makes searching operation easy to perform.
- Bubble sort is simple, widely known, but inefficient.
- Go through the array or file several iterations with each iteration placing one element in the right position.
- The main operation is to compare each array element x[i] with its immediate successor x[i+1] and swap them if they are not in proper order.
- For an array of **n** elements, **n – 1** comparisons are performed in the first iteration.
- As more and more iterations are performed, more and more elements are in their right positions. Fewer comparisons are needed.
- If there is no swapping done during an iteration, then the array is already in order and comparisons can be terminated.

Example 4.15 Write a subroutine to implement the bubble sort algorithm and a sequence of instructions along with a set of test data for testing this subroutine. Use an array in data memory that consists of n 8-bit unsigned integers for testing purpose.

Solution: The bubble sort subroutine needs three local variables:

- **in_order:** a flag indicating whether the array is in order after an iteration
- **inner:** number of comparisons remained to be performed in an iteration
- **iteration:** number of iterations remained to be performed

The caller of bubble sort subroutine pushed the array base address and array count in the stack.

The following program declares an array of **n** 8-bit elements in program memory and copies it to the data memory for testing purpose.

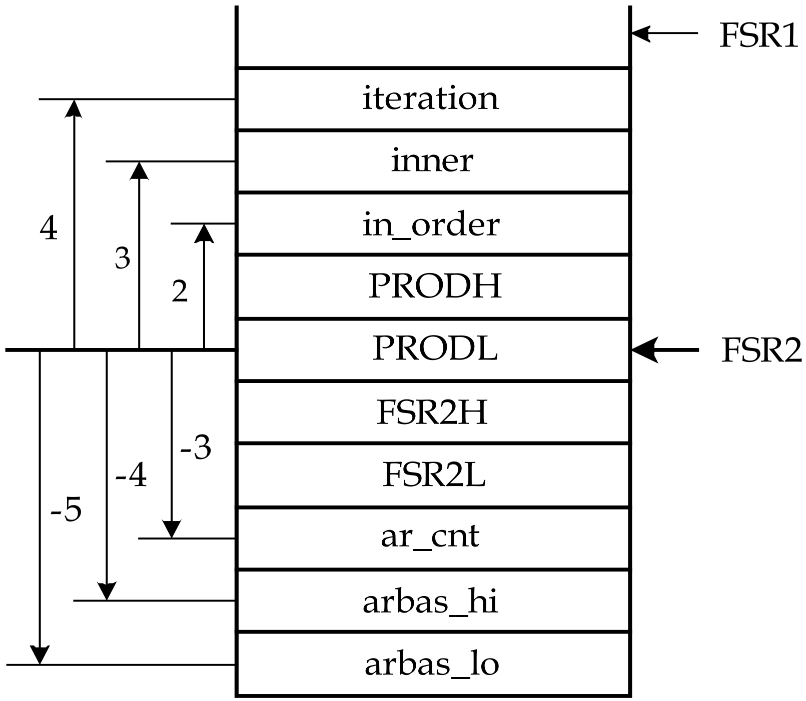

Figure 4.14 Stack frame of Example 4.15

```
            #include  <p18F8720.inc>
            radix     dec
; ----
; include macro definitions pushr, push_dat, popr, alloc_stk, and dealloc_stk here
; ----
NN          equ       30                   ; array count
loc_varb    equ       3                    ; number of bytes of local variables
arbas       equ       0x0100               ; array base address
lp_cnt      set       0                    ; use data memory location 0 for loop count
in_order    equ       2                    ; offset of local variable from frame pointer
inner       equ       3                    ; "
iteration   equ       4                    ; "
ar_cnt      equ       -3                   ; "
arbas_hi    equ       -4                   ; "
arbas_lo    equ       -5                   ; "
            org       0x00
            goto      start
            org       0x08                 ; high-priority interrupt vector
            retfie
            org       0x18                 ; low-priority interrupt vector
            retfie
```

```
start    lfsr    FSR1,0xE00        ; set up stack pointer
         movlw   high arbas        ; use FSR0 as the pointer to the array
         movwf   FSR0H,A           ; to be sorted
         movlw   low arbas         ;          "
         movwf   FSR0L,A           ;          "
         movlw   upper xarr        ; set up table pointer to point to
         movwf   TBLPTRU,A         ; the array in program memory
         movlw   high xarr         ;          "
         movwf   TBLPTRH,A         ;          "
         movlw   low xarr          ;          "
         movwf   TBLPTRL,A         ;          "
         movlw   NN
         movwf   lp_cnt
; ************************************************************
; copy an array from program memory to data memory to test the bubble
; sort subroutine.
; ************************************************************
copy_lp  tblrd*+                   ; read a byte into the table latch
         movff   TABLAT,POSTINC0 ; copy one byte
         decfsz  lp_cnt,F,A
         bra     copy_lp
         movlw   low arbas         ; push array base address onto stack
```

```
            pushr     WREG              ;          "
            movlw     high arbas        ;          "
            pushr     WREG              ;          "
            push_dat  NN                ; push array count
            call      bubble,FAST       ; call the bubble sort function
forever     nop
            bra       forever

bubble      pushr     FSR2L             ; save the current frame pointer
            pushr     FSR2H             ;          "
            movff     FSR1L,FSR2L       ; set up new frame pointer
            movff     FSR1H,FSR2H       ;          "
            pushr     PRODL             ; save PRODL in stack
            pushr     PRODH             ; save PRODH in stack
            alloc_stk loc_varb
            movlw     ar_cnt
            movff     PLUSW2,PRODL
            decf      PRODL,F,A
            movlw     iteration         ; set iteration to array count – 1
            movff     PRODL,PLUSW2      ;          “
```

```
ploop   movlw   in_order
        clrf    PLUSW2          ; set in_order flag to true (0)
        movlw   iteration
        movff   PLUSW2,PRODL
        movlw   inner
        movff   PRODL,PLUSW2    ; initialize inner loop count
        movlw   arbas_lo        ; use FSR0 as the array pointer
        movff   PLUSW2,FSR0L    ;          "
        movlw   arbas_hi        ;          "
        movff   PLUSW2,FSR0H    ;          "
cloop   movff   INDF0,PRODL     ; place arr[i] in PRODL
        movlw   1
        movf    PLUSW0,W        ; place arr[i+1] in WREG
        cpfsgt  PRODL,A
        goto    looptst
        movwf   PRODH           ; swap arr[i] with arr[i+1]
        movlw   1               ;          "
        movff   PRODL,PLUSW0    ;          "
        movff   PRODH,INDF0     ;          "
        movlw   in_order        ; set the in_order flag to false (1)
        incf    PLUSW2,F        ;          "
looptst movf    POSTINC0,W      ; increment array pointer
```

```
            movlw       inner           ; decrement inner loop count and skip if it
            decfsz      PLUSW2,F        ; has been decremented to zero
            goto        cloop
            movlw       in_order
            tstfsz      PLUSW2          ; is the array in order?
            goto        nexti           ; not yet
            goto        done            ; yes, return.
nexti       movlw       iteration
            decfsz      PLUSW2,F
            goto        ploop
done        dealloc_stk loc_varb
            popr        PRODH
            popr        PRODL
            popr        FSR2H
            popr        FSR2L
            return      FAST
xarr        db          0x56,0x1F,0x01,0x08,0x11,0x47,0x21,0x20,0x30,0x07
            db          0x19,0x18,0x17,0x16,0x15,0x14,0x13,0x12,0x29,0x28
            db          0x27,0x26,0x25,0x24,0x23,0x22,0x06,0x05,0x04,0x03
            end
```

Square Root Computation

- A simple but efficient way to compute the approximate integral square root of an integer is successive approximation method.
- The square root of an **2n**-bit number would be **n**-bit.
- Let **SAR** be an n-bit number, **Q** be a 2n-bit number of which the square root is to be computed.

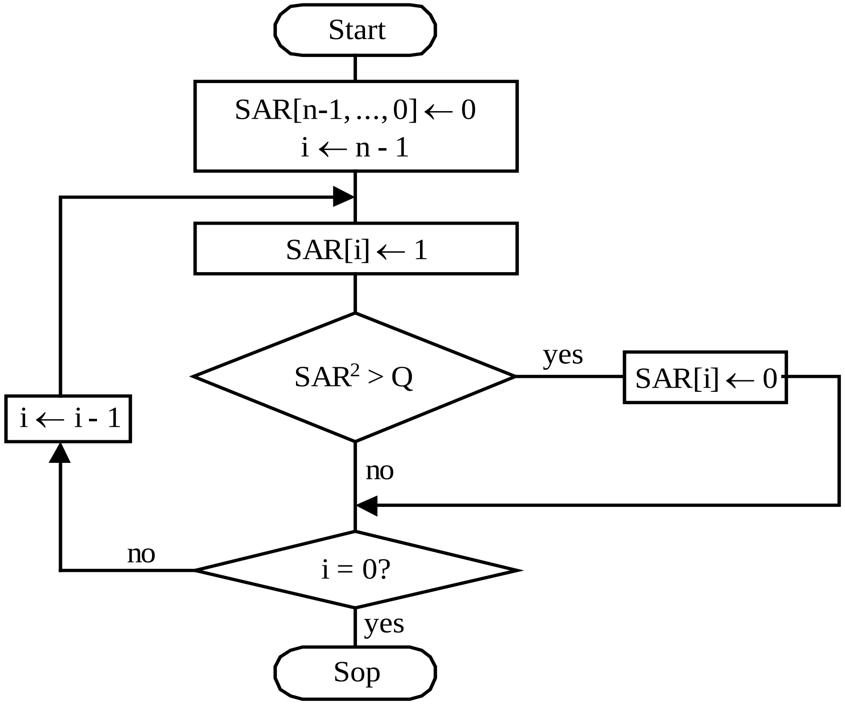

Figure 4.15 Successive approximation method for computing the square root of a 2n-bit number Q.

- The algorithm in Figure 4.15 computes an approximate integer square root smaller than or equal to the actual one. The better approximation could be [SAR] or [SAR]+1.
- The better approximate square root can be found by comparing **Q – [SAR*SAR]** and **([SAR]+1)*([SAR]+1)**.

Example 4.16 Write an assembly routine to implement the successive approximation method for computing the square root of a 16-bit integer. The 16-bit integer will be pushed into the stack. The square root is returned in PRODL.

Solution: The subroutine has three local variables:

1. sar: Successive approximation register
2. mask: A value used to set the **ith** bit
3. lp_cnt: loop count

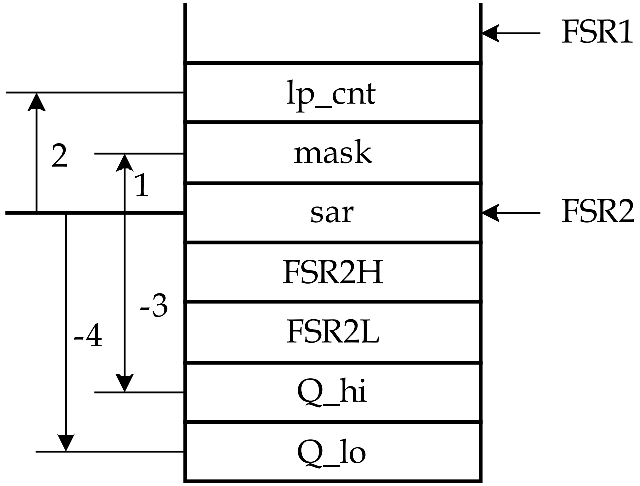

Figure 4.16 Stack frame for Example 4.16

```
            #include <p18F8720.inc>
            radix       dec
; ----
; include macro definitions pushr, push_dat, popr, alloc_stk, and dealloc_stk here
; ----
testd_hi    equu        0x41                ; data to be tested
testd_lo    equ         0x2E                ;   "
loc_vars    equ         3                   ; number of bytes for local variable
Q_hi        equ         -3                  ; offsets of Q from frame pointer
Q_lo        equ         -4                  ;   "
sar                     equ                 0  ;          "
mask        equ         1                   ;   "
lp_cnt      equ         2                   ;   "
sq_root     set         0x00                ; memory location to save square root
            …                               ; insert testing program here
find_sqr    pushr       FSR2L
            pushr       FSR2H
            movff       FSR1L,FSR2L
            movff       FSR1H,FSR2H
            alloc_stkloc_vars
            movlw       8                   ; initialize loop count
            movwf       PRODL,A             ;   "
```

```
        movlw   lp_cnt          ;        "
        movff   PRODL,PLUSW2    ;        "
        movlw   0x80            ; start with the mask 0x80
        movwf   PRODL           ;        "
        movlw   mask            ;        "
        movff   PRODL,PLUSW2    ;        "
        clrf    INDF2           ; initialize SAR to 0
loop    movlw   mask
        movf    PLUSW2,W        ; place mask in WREG
        iorwf   INDF2,W         ; set the ith bit of SAR
        mulwf   WREG,A          ; compute SAR*SAR
        movlw   Q_lo            ; compute SAR*SAR - Q
        movf    PLUSW2,W        ;        "
        subwf   PRODL,F         ;        "
        movlw   Q_hi            ;        "
        movf    PLUSW2,W        ;        "
        subwfb  PRODH,F         ;        "
        btfsc   STATUS,C        ; skip if Q > SAR*SAR
        goto    nextbit
        movlw   mask            ; the guess is right, so
        movf    PLUSW2,W        ; set bit i of SAR
        iorwf   INDF2,F         ;        "
```

```
nextbit movlw   lp_cnt          ; decrement the loop count
        decf    PLUSW2,F        ;           "
        bz      done            ; it is done if lp_cnt = 0
        bcf     STATUS,C,A      ; clear the C flag
        movlw   mask
        rrcf    PLUSW2,F        ; shift the mask to the right
        goto    loop
; ***************************************************************
; Before return, find out if SAR*SAR or (SAR+1)**2 is closer to Q and
; return [SAR] or [SAR]+1 accordingly.
; ***************************************************************
done    movf    INDF2,W         ; get the estimated square root
        mulwf   WREG,A          ; compute SAR*SAR
        movlw   Q_lo            ; compute SAR*SAR - Q (is < 0)
        movf    PLUSW2,W        ;           "
        subwf   PRODL,F,A       ;           "
        movlw   Q_hi            ;           "
        movf    PLUSW2,W        ;           "
        subwfb  PRODH,F,A       ;           "
        comf    PRODH,F,A       ; compute the magnitude of
        negf    PRODL,A         ; |SAR*SAR - Q|
        movlw   0               ;           "
```

```
addwfc   PRODH,F,A       ;         "
movlw    lp_cnt          ; copy |SAR*SAR - Q| into lp_cnt
movff    PRODH,PLUSW2    ; and mask
movlw    mask            ;         "
movff    PRODL,PLUSW2    ;         "
movf     INDF2,W         ; compute SAR+1
incf     WREG,W,A        ;         "
mulwf    WREG,A          ; compute (SAR+1)**2
movlw    Q_lo            ; compute (SAR+1)**2 - Q
movf     PLUSW2,W        ; and leave the difference in
subwf    PRODL,F,A       ; PRODH:PRODL
movlw    Q_hi            ;         "
movf     PLUSW2,W        ;         "
subwfb   PRODH,F,A       ;         "
movlw    mask            ; compare [Q-(SAR*SAR)] with
movf     PLUSW2,W        ; (SAR+1)**2 - Q
subwf    PRODL,F,A       ;         "
movlw    lp_cnt          ;         "
movf     PLUSW2,W        ;         "
subwfb   PRODH,F,A       ;         "
btfsc    STATUS,C        ;         "
goto     sel_SAR         ;         "
```

```
        incf        INDF2,F      ; increment SAR by 1
sel_SAR dealloc_stk 2            ; remove lp_cnt and mask from stack
        popr        PRODL        ; place SAR in PRODL
        popr        FSR2H
        popr        FSR2L
        return
        end
```

Prime Number Test

- An integer is prime if it cannot be divided by any of the prime numbers smaller or equal to its square root.
- A less efficient method is to divide the given number by all the integers from 2 to its approximate integral square root.

Example 4.17 Write a subroutine to test whether a 16-bit unsigned integer is a prime number. The integer to be tested is pushed into the stack and the test result is returned in the WREG register. If the number is prime, the subroutine returns a one. Otherwise, it returns a zero.

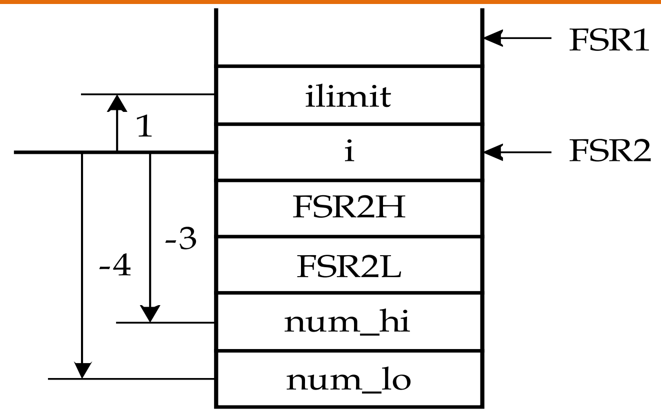

Figure 4.17 Stack frame for example 4.17

```
            #include   <p18F8720.inc>
            radix      dec
; ----
; include macro definitions pushr, push_dat, popr, alloc_stk, and dealloc_stk here
; ----
loc_varp    equu       2                ; local variable size for prime_test
ilimit      equ        1                ; offset of ilimit from FSR2
num_hi      equ        -3               ; offset of test number high byte from FSR2
num_lo      equ        -4               ; offset of test number low byte from FSR2
```

```
prime_tst       pushr       FSR2L
        pushr       FSR2H
        movff       FSR1L,FSR2L     ; set up frame pointer
        movff       FSR1H,FSR2H     ;           "
        alloc_stk   loc_varp        ; allocate local variables
        movlw       num_lo          ; get a copy of the low byte
        movf        PLUSW2,W        ; of the incoming number
        pushr       WREG
        movlw       num_hi          ; get a copy of the high byte
        movf        PLUSW2,W        ;           "
        pushr       WREG
        call        find_sqr        ; compute the upper limit for prime test
        dealloc_stk                 2       ; clean up the pushed data from stack
        movlw       ilimit          ; set up the upper limit for prime
        movff       PRODL,PLUSW2    ; test
        movlw       2
        movwf       INDF2           ; initialize i to 2
loop    movlw       ilimit          ; place the ilimit in WREG
        movf        PLUSW2,W        ;           "
        cpfsgt      INDF2           ; is i > ilimit?
        goto        inloop
        goto        isprime         ; it is prime if i > ilimit
```

```
inloop      movlw       num_lo       ; push the number to be test for prime
            movf        PLUSW2,W     ; onto the stack
            pushr       WREG         ;        "
            movlw       num_hi       ;        "
            movf        PLUSW2,W     ;        "
            pushr       WREG         ;        "
            alloc_stk   2            ; allocate two bytes for R register
            movf        INDF2,W      ; get the value of i
            pushr       WREG         ; push i into the stack
            push_dat    0            ; push 0 as the upper byte of divisor
            call        div16u
            dealloc_stk 3            ; remove the pushed top three bytes
            movlw       -1           ; the low byte of the remainder is one
            tstfsz      PLUSW1       ; below the byte pointed to by FSR1
            goto        nexti        ; remainder is not zero, test next i
            goto        not_prime
nexti       incf        INDF2,F
            dealloc_stk 3            ; deallocate the pushed remaining 3 bytes
            goto        loop
not_primedealloc_stk 2+loc_varp      ; deallocate the pushed remaining 3 bytes
                                     ; and two bytes of local variables
            popr        FSR2H
            popr        FSR2L
```

```
        movlw   0                       ; return a 0 to indicate non_prime
        return
isprime dealloc_stk                     loc_varp  ; deallocate local variables
        popr    FSR2H
        popr    FSR2L
        movlw   1
        return
; ----
; include subroutine find_sqr and its associated equ directives here
; ----
; ----
; include subroutine div16u and its associated equ directives here
; ----
        end
```