Chapter 4: Advanced Assembly Programming

- A program consists of two parts: ***data structures*** and ***algorithms***
- Algorithms deal with systematic methods for information processing (see ch. 2)
- Data structures deal with how information is organized, such as
  - **stacks**: a first-in-last-out (FILO) data structure
  - **arrays**: an ordered set of elements of the same type
  - **strings**: a sequence of characters

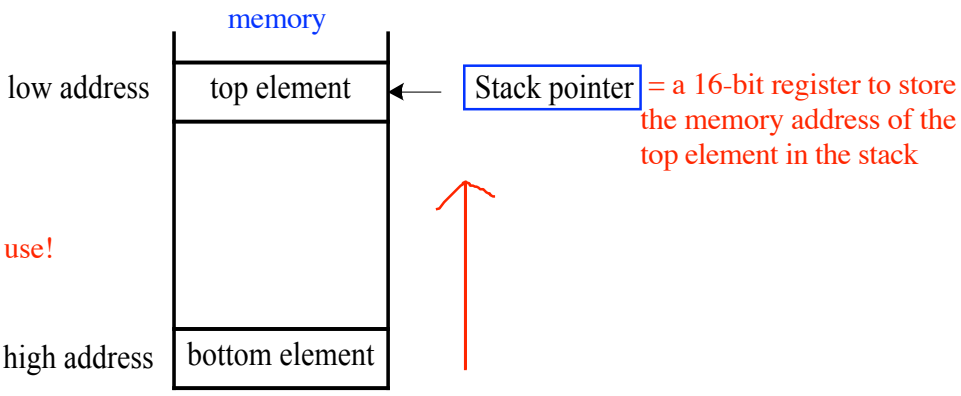
Stack

- A data structure with a top element and a bottom element
- Elements can only be added (so-called ***push***) or deleted (so-called ***pull***) from the top of the stack
- A reserved RAM area in main memory when programs agree to perform only push and pull operations
- 68HC12 has a ***16-bit stack pointer*** (SP) that points to the memory location (in byte) of the top element
- Grows from ***higher*** address towards ***lower*** address
- Must check the stack ***overflow*** and ***underflow*** (or pulled too many times) to make sure that the program won't crash
- 2 step process:
  - **Push** writes data from a register to the top of the stack after decrementing the SP
  - Pull loads data from the top of the stack to a register before incrementing the SP

PUSH = 1. up SP
2. write to stack

PULL = 1. read from stack
2. down SP

up/down SP depends on the register in use!



- Instructions have equivalent store and instructions combined with predecrement and postincrement indexed addressing modes

Table 4.1 68HC12 push and pull instructions and their equivalent load and store instructions

pre-decrement

Mnemonic	Function	Equivalent instruction
psha	push A into the stack	staa 1, -SP
pshb	push B into the stack	stab 1, -SP
pshc	push CCR into the stack	none
pshd	push D into stack	std 2, -SP
pshx	push X into the stack	stx 2, -SP
pshy	push Y into the stack	sty 2, -SP
pula	pull A from the stack	ldaa 1, SP+
pulb	pull B from the stack	ldab 1, SP+
pulc	pull CCR from the stack	none
puld	pull D from the stack	ldd 2, SP+
pulx	pull X from the stack	ldx 2, SP+
puly	pull Y from the stack	ldy 2, SP+

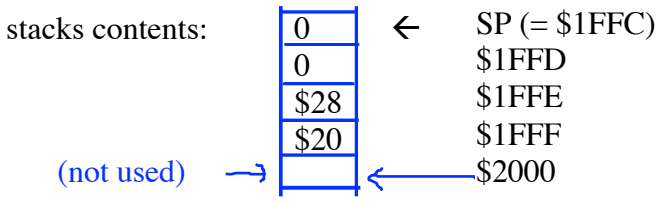
8 bits use 1 address

16 bits user 2 addresses

post-increment

Example: What would be the contents of the stack after the execution of the following instruction sequence?

```
SP <-- $2000        LDS     #$2000    ; initializes the SP to memory location $2000
A <-- 0010 0000     LDAA    #$20      ; load the 8-bit value $20 to A
stack <-- (A)       STAA    1,-SP     ; push A into stack (= PSHA), one byte
B <-- 0010 1000     LDAB    #40       ; load the 8-bit value 40 (hex $28) to B
stack <-- (B)       STAB    1,-SP     ; push B into stack (= PSHB), one byte
                    LDX     #0        ; load the 16-bit value 0 to X
                    STX     2,-SP     ; push X into stack (= PSHX), two bytes
```



- Main use of the stack is saving the return address for a subroutine call.
- Need to set up the stack pointer's memory location before using the stack.
- HCS12's external SRAM has the address space $1000~$3BFF open for the user program and the stack. Since the stack grows from higher towards lower addresses, a good initial setting for the SP would be **$2000**, while $2000 is used as starting address of the user program.

Indexable Data Structures

dc.b = define constant byte
dc.w = define constant word = 2 bytes
rmb = reserve memory space

- Vectors (1-D) and matrices (2-D) are indexable data structures.
- First element of a vector is associated with the index **i = 0**.
- Directives **dc.b**, **db**, and **fcb** are used to define arrays of 8-bit elements.
- Directives **dc.w**, **dw**, and **fdb** are used to define arrays of 16-bit elements.
- Directives **ds.b**, **ds**, and **rmb** are used to reserve memory space for arrays of 8-bit elements.
- Directives **ds.w** and **rmw** are used to reserve memory space for arrays of 16-bit elements.

```
        org     $1000           ; first element of vec_x starta at $1000
        dc.b    11,12,13,14,15  ; array of five 8-bit numbers
```

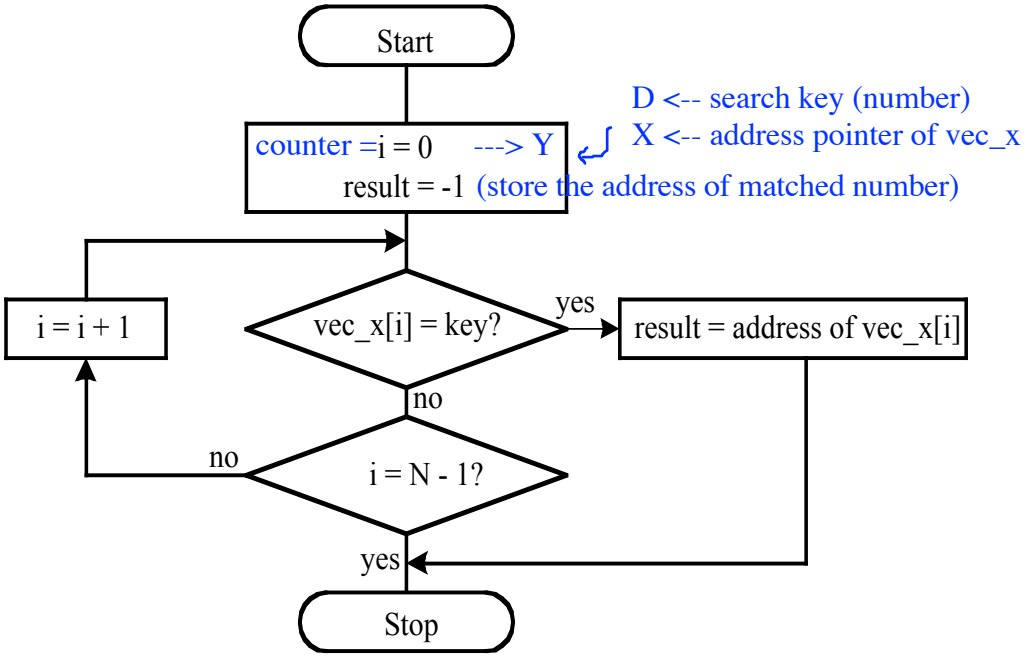
t	
$1000	0000 1011
$1001	0000 1100
$1002	0000 1101
$1003	0000 1110
$1004	0000 1111

Example: **(Search a number in an array)** Write a program to find out if the array ***vec_x*** contains a value stored in ***key***. The array has N 16-bit elements and is not sorted.

To implement the sequence search algorithm, we will

- Use the double accumulator D to hold the search key.
- Use the index register X as a pointer to the array.
- Use the index register Y to hold the loop counter.
- Compare the search key with every array element.

Why cannot use A and B?



```
            N           equ     30          ; array size
constants   notfound    equ     -1          ; flag for key not found
            key         equ     190         ; define the searching key = 190

                        org     $1000       ; memory location for "result"
            result      rmw     1           ; reserve 16 bits for "result" @ $1000 & $1001

                        org     $2000       ; starting address of the program
                        ldy     #N          ; set up loop counter in Y
watch out the use of #  ldd     #notfound   ;
                        std     result      ; initialize the search result to -1
                        ldd     #key        ;
                        ldx     #vec_x      ; place the starting address of vec_x in X
            loop        cpd     2,X+        ; compare the key D with array element [X],
                                            ; then increment X by 2        [Why?]
                                            ; (post-increment indexed addressing)
                        beq     found       ; branch to "found" if equal
                        dbne    Y,loop      ; decrement Y, then branch to "loop" if ≠ 0
                        bra     done        ; if = 0, branch to "done"
            found       dex                 ; need to restore X to point to the address of
                        dex                 ; the matched element
why need them?          stx     result      ;
            done        swi                 ; return to the DBug12 monitor

            vec_x       dw      13,15,320,980,42,86,130,319,430,4, 90,20,18,55,30,51,37
                        dw      89,550,39, 78,1023,897,930,880,810,650,710,300,190
                        end
```

; the number 190 is located at $4058.
; "result" is located at $1000, which contains the address ($4058) of the matched key 190.
; type "md 1000" to see the content.

```
cpd   2,X+   =   cpd   X
                 inx
                 inx
```

The above is a sequential search:

- search an array element-by-element sequentially
- inefficient (e.g., what if the matching number is the last element)

Binary Search

- If an array (e.g., a lookup table) needs to be searched frequently, then a sequential search is very inefficient.
- A better approach is to **sort** the array first and then use the binary search algorithm.

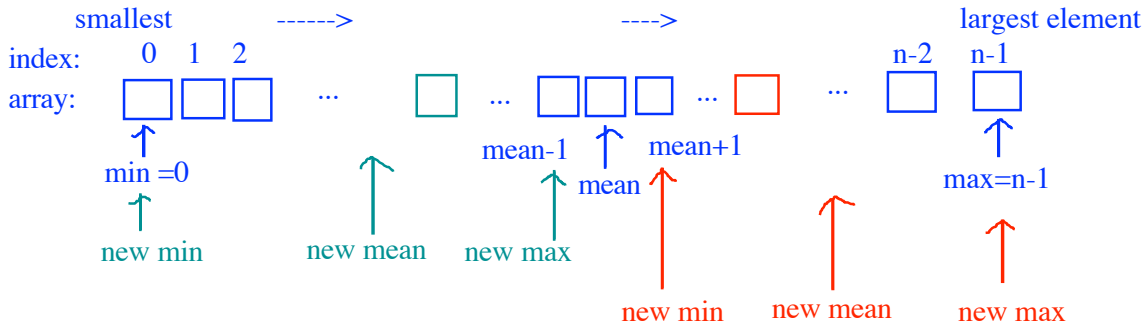
- Assumptions:
  - Requires a sorted array
  - The sorted array (in ascending order) has *n* elements and is stored at memory locations starting at label ***arr***.
  - Let ***max*** and ***min*** represent the highest and lowest range of array **indices** to be searched. [Norte: they are not address or element values.]
  - Let ***mean*** represent the average of *max* and *min*. mean = (max+min)/2

- The binary search algorithm divides the array into 3 portions:
  1. The portion of the array with indices ranging from *mean*+1 to *max*.
  2. The element with index equal to *mean* (i.e., middle element).
  3. The portion of the array with indices ranging from *min* to *mean*-1.

- The algorithm then compare the key with the middle element:
  - If the key equals the middle element, then stop.
  - If the key is larger than the middle element, then the key can be found only in the portion of the array with larger indices. The search will be continued in the upper half.
  - If the key is smaller than the middle element, then the key can be found only in the portion of the array with smaller indices. The search will be continued in the lower half.

key	> = <	array[mean]

Step 1: Initialize variables *max* to *n*-1 and *min* to 0.
Step 2: If max < min, then stop. No element matches the key.
Step 3: Let *mean* = (*max* + *min*)/2.
Step 4: If *key* = *arr*[*mean*], then key is found in the array, exit.
Step 5: If *key* < *arr*[*mean*], then set *max* to *mean*-1 and go to step 2.
Step 6: If *key* > *arr*[*mean*], then set *min* to *mean*+1 and go to step 2.



Example: Write a sequence of instructions to implement the binary search algorithm. Use an array of N 8-bit elements. Store a "1" as the result if key is found in the array.

```
n       equ   30          ; array size
key     equ   69          ; define the search key = 69
        org   $1000       ; reserves memory space for variables
max     rmb   1           ; reserve one byte for maximum index
min     rmb   1           ; reserve one byte for minimum index
mean    rmb   1           ; reserve one byte for mean index
result  rmb   1           ; for search result (1=found; 0=no match)

        org   $2000       ; program start address
        clra              ; clear A
        staa  min         ; initialize min to 0
        staa  result      ; initialize result to 0 (no match)
        ldaa  #n-1        ; watch out "#"
        staa  max         ; initialize max to n-1
        ldx   #arr        ; use X as the pointer to the array (start address)
loop    ldab  min         ; copy min to B
        cmpb  max         ; compute (B) – (max)
        lbhi  notfound    ; if min > max, long branch to "notfound"
        addb  max         ; max+min  (compute mean)
        lsrb              ; divide by 2 (shift right)
        stab  mean        ; save mean
        ldaa  b,x         ; get a copy of the element arr[mean]
        cmpa  #key        ; compute arr[mean]– key
        beq   found       ; if =0, branch to "found"
        bhi   search_lo   ; if >0, arr[mean] is on RHS of the key
        ldaa  mean        ; if <0, arr[mean] is on LHS of the key
        inca              ; replace min by mean+1
        staa  min         ; but max stays the same
        bra   loop
search_lo ldaa mean       ; replace max by mean-1
        deca              ; but min stays the same
        staa  max         ;
        bra   loop        ;
found   ldaa  #1          ; set "result"=1
        staa  result
notfound swi
arr     db    1,3,6,9,11,20,30,45,48,60
        db    61,63,64,65,67,69,72,74,76,79
        db    80,83,85,88,90,110,113,114,120,123
        end
```

B=index of mean

wacth out "#"

A contains the element with the index = sum of B+X

A <-- arr[B+X]

step 4:

step 5:

step 6:

if arr[mean] > key

no

yes

; $0F is found at location $1002, which is the memory location of "mean".
; $0F corresponds to the 15th element in the "array", array[15]=69=the search key.

Strings

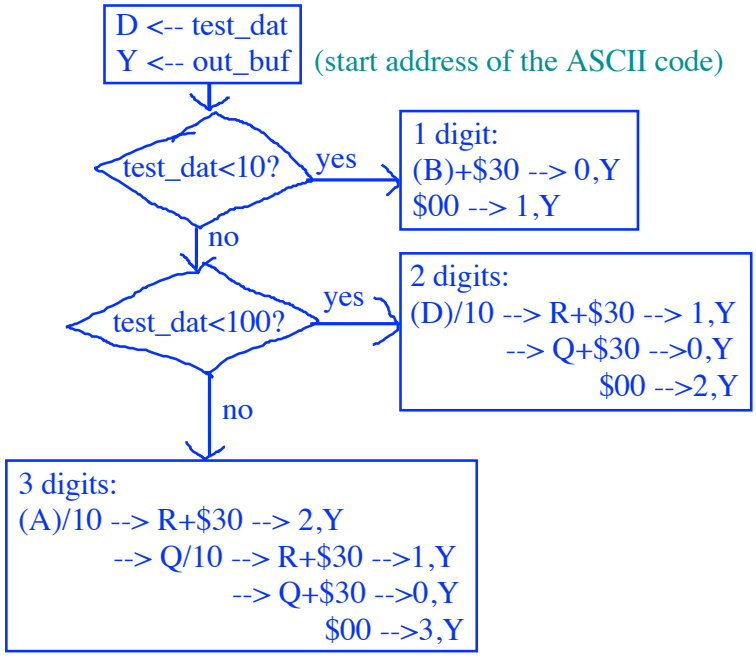
- A sequence of characters terminated by a NULL (ASCII code **$00**) or other special character such as EOT (ASCII code 4).
- Heavily used in input and output operations. Example: Display text/numbers on a computer screen.
- All characters are stored as ASCII codes in microcontroller memory.
- A number (stored in binary format in memory) must be converted to ASCII code before being displayed on the computer screen.
- To convert a binary number into an ASCII string, we divide the binary number by 10 repeatedly until the quotient is zero. The remainder gives the corresponding BCD digit. For each remainder, the hex number $30 is added to get the ASCII character of the digit.

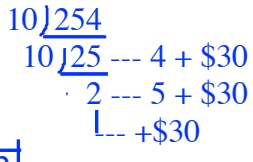
Binary to BCD ASCII Conversion

Example: Convert the unsigned 8-bit number in accumulator A into three BCD digits terminated by a NULL character. Each digit is represented in the ASCII form.

Since an 8-bit number can be from 0 to 255, we need to use

1. 4 bytes to hold the four ASCII codes of the three BCD digits and the NULL character.
2. Repeat the division-by-10 + $30 method.





0,Y	$32
1,Y	$35
2,Y	$34
3,Y	$00

X=divisor=#10
D=the number
Y=result address pointer

What not to useA or B?

8

0,Y	$38
1,Y	$00

19

0,Y	$31
1,Y	$39
2,Y	$00

```
test_dat    equ   254          ; the 8-bit number  (Note: this is a constant)
            org   $1000        ; below is a variable
out_buf     rmb   4            ; the ASCII code of 3 BCD digits + 1 NULL

            org   $2000        ; starting address of the program
            ldaa  #test_dat    ; load the test data  (A=254)
            ldy   #out_buf     ; load the address of the converted result (Y=$1000)
            tab                ; transfer from A into B          [Why this step?]
; check to see if the number has only one digit
            cmpb  #9           ; compute (B) - 9
            bhi   chk_99       ; if >0, branch to consider 2 digits
            addb  #$30         ; if ≤0, convert the digit into ASCII code
            stab  0,y          ; save the code and increment the pointer
            clr   1,y          ; terminated the string with NULL
            jmp   done
; check to see if the number has two digits
chk_99      clra               ; clear A → D=A:B=$00:B
            cmpb  #99          ; is the number greater than 99?
            bhi   three_dig    ; if yes, the string has three digits
            ldx   #10          ; if not, perform divide-by-10
            idiv               ; (D)/(X) → quotient X & remainder D=A:B
            addb  #$30         ; convert the lower digit to ASCII
            stab  1,y          ; store the unit digit
            xgdx               ; exchange the contents of X and D
            addb  #$30         ;
            stab  0,y          ; save the tens digit
            clr   2,y          ; terminated the string with NULL
            bra   done
;
three_dig   ldx   #10          ;
            idiv
            addb  #$30
            stab  2,y          ; save the unit digit
            xgdx               ; swap the quotient to D
            ldx   #10
            idiv
            addb  #$30
            stab  1,y          ; save the tens digit
            xgdx               ; swap the hundreds digit to B
            addb  #$30
            stab  0,y          ; save the ASCII code of the hundreds digit
            clr   3,y          ; terminate the string with NULL

done        swi
            end
```

watch "#" for constant & variable

unit digit

tens digit

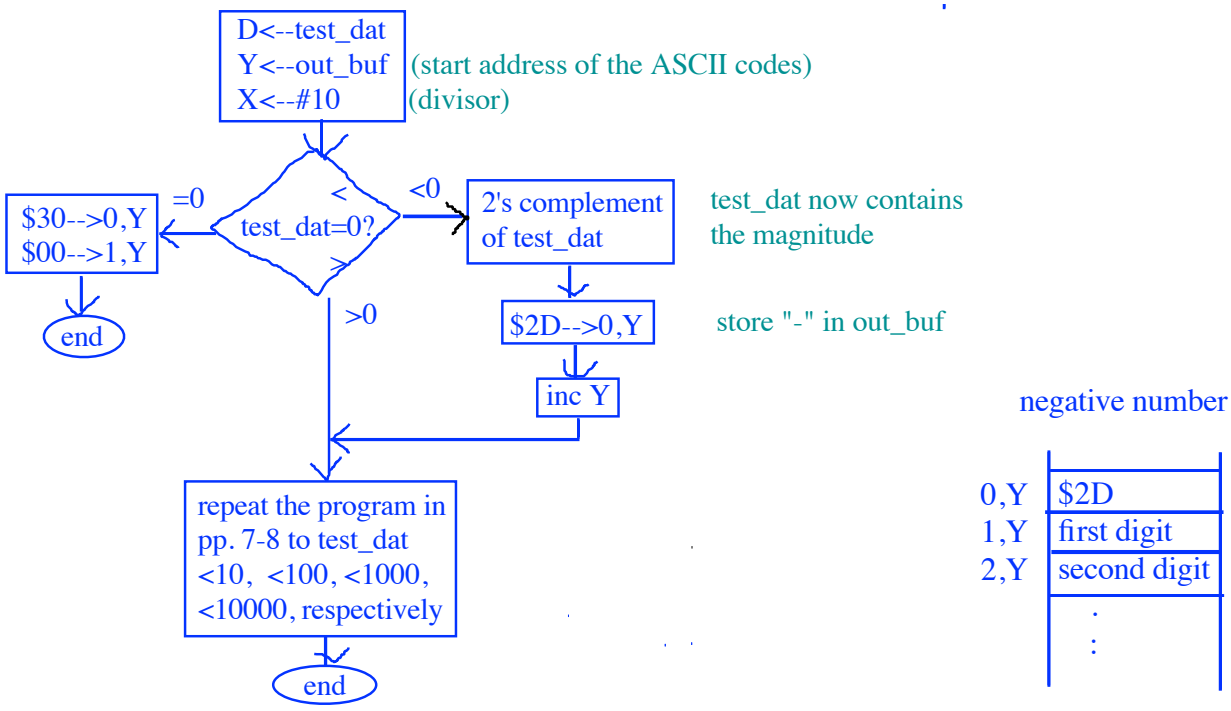
unit digit

tens digit

hundreds digit

Example: Convert the 16-bit signed integer stored in D into a string of BCD digits.

- A signed 16-bit integer is in the range of -32768 to +32767.
- A negative number is stored in signed-2's complement form.
- A minus character is needed for a negative number, plus changing 2's complement to magnitude.
- A NULL character is needed to terminate the converted BCD string.
- Up to **7 bytes** are needed to hold the converted digits in BCD format.
- Repeat the division-by-10 + $30 method.




```
test_dat    equ   -4300      ; this is a constant
            org   $1000      ; below are variables
out_buf     rmb   7          ; storage of the resulted BCD digits
temp        rmb   2          ; temporary storage

            org   $2000
            ldd   #test_dat  ; load the test data                (D = -4300)
            ldy   #out_buf   ; Y as the pointer to the output buffer (Y=$1000)
            cpd   #0         ; compare D with 0
            lbpl  chk_9      ; if plus, no complement and jump to p. 10
            lbeq  zero       ; is the given number a 0?   (If yes, jump to p. 12)
; the following three instructions compute the magnitude of the given number in D
            coma             ; perform 1's complement to A
            comb             ; and B → 1's complement of D
            addd  #1         ; 1's complement +1 = 2's
            std   temp       ; store the magnitude into "temp"
```

watch "#" for constant & variable

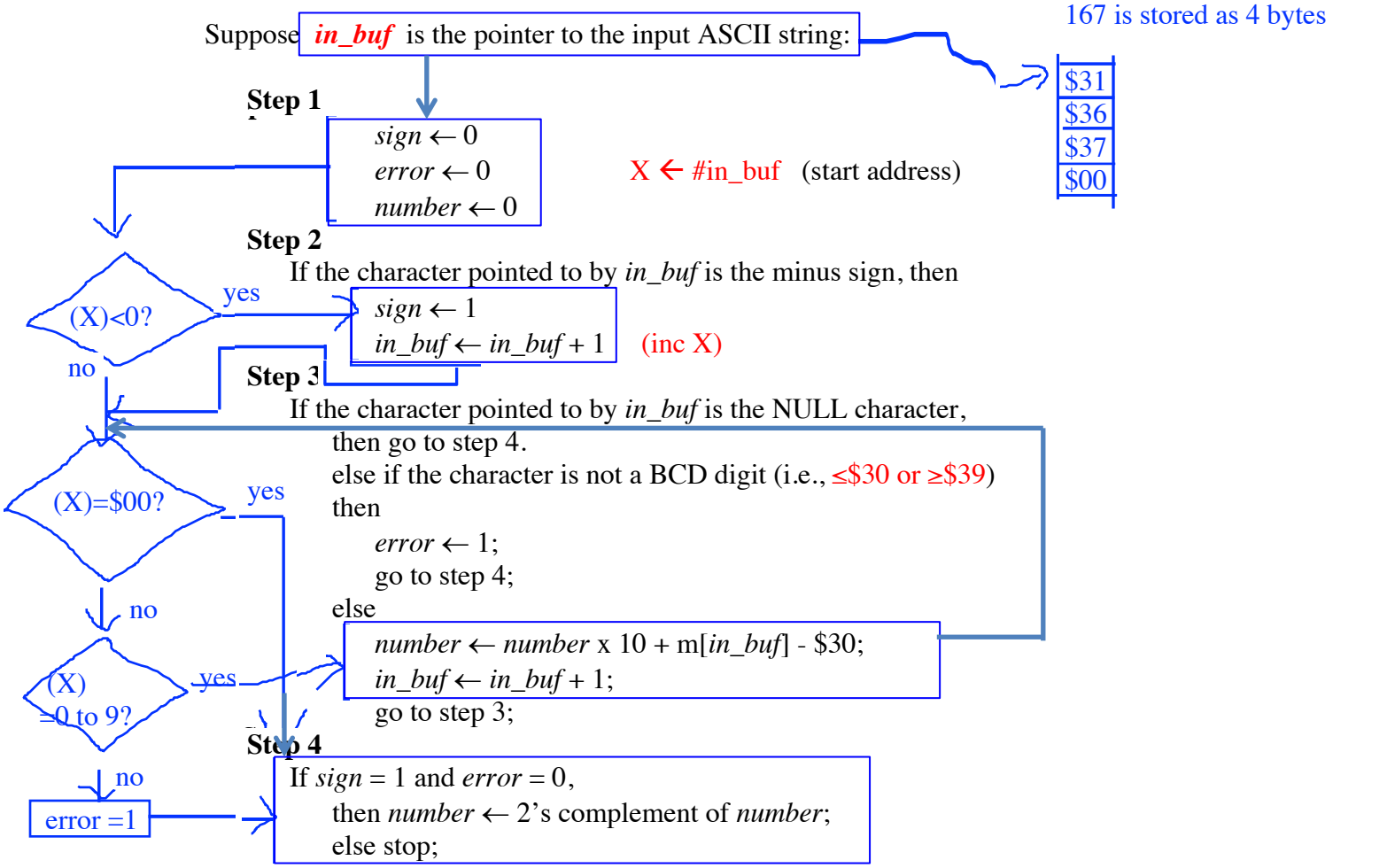
2's complement is converted bak to magnitude

```
        ldaa    #$2D        ; place a negative sign ($2D=ASCII of -)
        staa    0,y         ; store the sign to Y=$1000
        iny                 ; set storage address for the next converted digit
        ldd     temp        ; move "temp" back to D
chk_9   cpd     #9          ; compute (D)-9
        lbhi    chk_99      ; if >0, then long branch to consider 2 digits
        addb    #$30        ; convert the digit to ASCII
        stab    0,y         ;
        clr     1,y         ; terminate the string with NULL
        lbra    done        ; long branch
chk_99  cpd     #99         ; does the number have only two digits?
        lbhi    chk_999     ; branch if the number has more than two digits
        ldx     #10
        idiv
        addb    #$30        ; convert the ones digit to BCD ASCII
        stab    1,y         ; save the ones digit
        xgdx
        addb    #$30
        stab    0,y         ; save the tens digit
        clr     2,y         ; add a NULL character
        lbra    done
chk_999 cpd     #999        ; does the number have only three digits?
        lbhi    chk_9999    ; branch if the number has more than three digits
        ldx     #10
        idiv
        addb    #$30        ; convert the ones digit
        stab    2,y         ; save the ones digit
        xgdx
        ldx     #10
        idiv
        addb    #$30
        stab    1,y         ; save the tens digit
        xgdx
        addb    #$30        ; convert the hundreds digit
        stab    0,y
        clr     3,y         ; add a NULL character
        lbra    done
chk_9999 cpd    #9999       ; does the number have only four digits?
        lbhi    five_digit  ; branch if the number has five digits
        ldx     #10
        idiv
        addb    #$30
        stab    3,y         ; save the ones digit
        xgdx
        ldx     #10
        idiv
```

```
            addb    #$30
            stab    2,y
            xgdx
            ldx     #10
            idiv
            addb    #$30
            stab    1,y
            xgdx
            addb    #$30
            stab    0,y
            clr     4,y         ; add a NULL character
            lbra    done
-------------------------------------------------------------------
five_digit  ldx     #10
            idiv
            addb    #$30
            stab    4,y         ; save the ones digit
            xgdx
            ldx     #10
            idiv
            addb    #$30
            stab    3,y         ; save the tens digit
            xgdx
            ldx     #10
            idiv
            addb    #$30
            stab    2,y         ; save the hundreds digit
            xgdx
            ldx     #10
            idiv
            addb    #$30
            stab    2,y         ; save the hundreds digit
            xgdx
            ldx     #10
            idiv
            addb    #$30
            stab    1,y         ; save the thousands digit
            xgdx
            addb    #$30
            stab    0,y         ; save the ten-thousands digit
            clr     5,y         ; add a NULL character
done        swi
zero        ldaa    #$30
            staa    0,y
            clr     1,y
            swi
            end
```

BCD ASCII to Binary Conversion

- When a number is entered from a keyboard, it is often in the form of a NULL-terminated ASCII string.
- Need to convert from the ASCII string that represents a BCD number to a binary number (in signed-2's complement form) for memory storage.

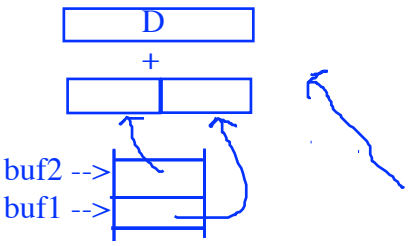


Example: Convert a BCD ASCII string (in the range of $-2^{15} \sim 2^{15}-1$ for 2 bytes) to a binary number. Store the result in D.

```
                minus       equ     $2D         ; ASCII code of minus sign (a constant)
                            org     $1000
4 bytes         in_buf      fcc     "167"       ; input a string of characters to be converted
                            fcb     0           ; followed by the NULL character
                out_buf     rmb     2           ; store the converted value of the characters
                buf2        rmb     1           ; temporary storage holds upper byte
                buf1        rmb     1           ; holds the lower byte of the converted number
                sign        rmb     1           ; holds the sign of the number
                error       rmb     1           ; indicates the occurrence of illegal character
```

167 = $00A7 will be stored in $1004 & $1005

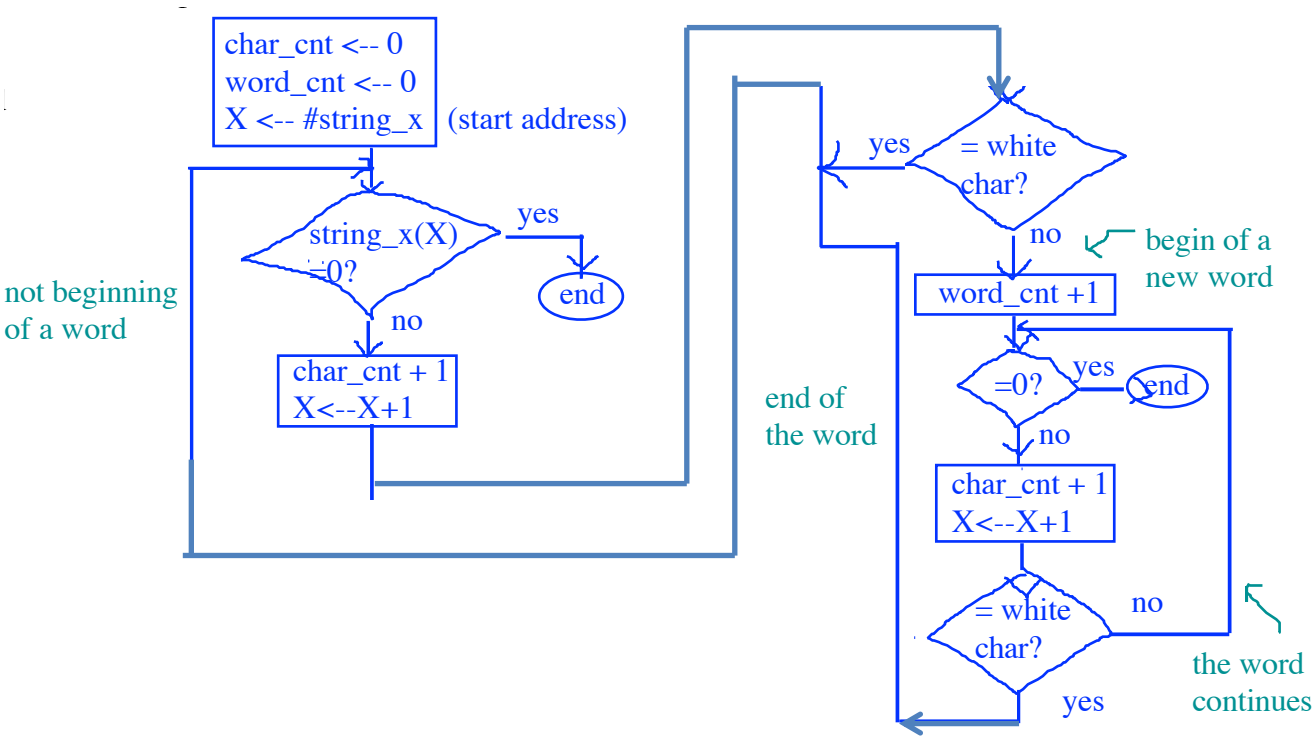
```
                org     $2000
                clr     sign
                clr     error
                clr     out_buf
                clr     out_buf+1
                clr     buf2
                ldx     #in_buf     ; X points to the start address of digits    (X=$1000)
                ldaa    0,x         ; load the ASCII of the first character       (A=$31)
                cmpa    #minus      ; is the first character a minus sign?
                bne     continue    ; if not equal, branch
                inc     sign        ; set the sign to 1
                inx                 ; X points to the address of 2nd character
--------------------------------------------------------------------------------
continue        ldaa    1,x+        ; load X into A; afterwards, increment X by 1
                lbeq    done        ; is the current character a NULL character?
                                    ; if =0, end of the string is reached   (0=$00=NULL)
                cmpa    #$30        ; compute (A) - $30
check if the ASCII                  ;
character is a number               ;
                lblo    in_error    ; if <0, error is detected and jumps to in_error
                cmpa    #$39        ;
                lbhi    in_error    ; if >39, error is detected and jumps to in_error
                suba    #$30        ; convert to the BCD digit value
                staa    buf1        ; save the digit value temporarily     buf1 ← (A)-#30
                ldd     out_buf     ;
                ldy     #10         ;
                emul                ; perform 16-bit x16-bit          Y:D ← DxY
                addd    buf2        ; add buf1 value (16 bits=buf2:buf1) & buf2=$00
                std     out_buf     ; above steps give      out_buf ← out_bufx10 + buf1
                bra     continue
--------------------------------------------------------------------------------
in_error        ldaa    #1          ; set error flag = 1
                staa    error
                swi
done            ldaa    sign        ; check to see if the original number is negative
                beq     positive
                ldd     out_buf     ; if negative, change the 2 bytes to 2's complement
                coma                ;       "
                comb                ;       "
                addd    #1          ;       "
                std     out_buf
positive        swi
                end
```




```
; md $1000     $31  $36  $37  00  00  A7 … 167
; $00 = NULL;    $00 A7 = 167 in hex
```

Character and Word Counting

- A string is terminated by the NULL character.
- Every non-NULL character (includes white space and punctuation) causes the character count to be incremented by one.
- A new word is surrounded by white space or punctuation.
- When a new word is identified, it must be scanned through the remaining nonwhite characters before the next word can be identified.



white characters

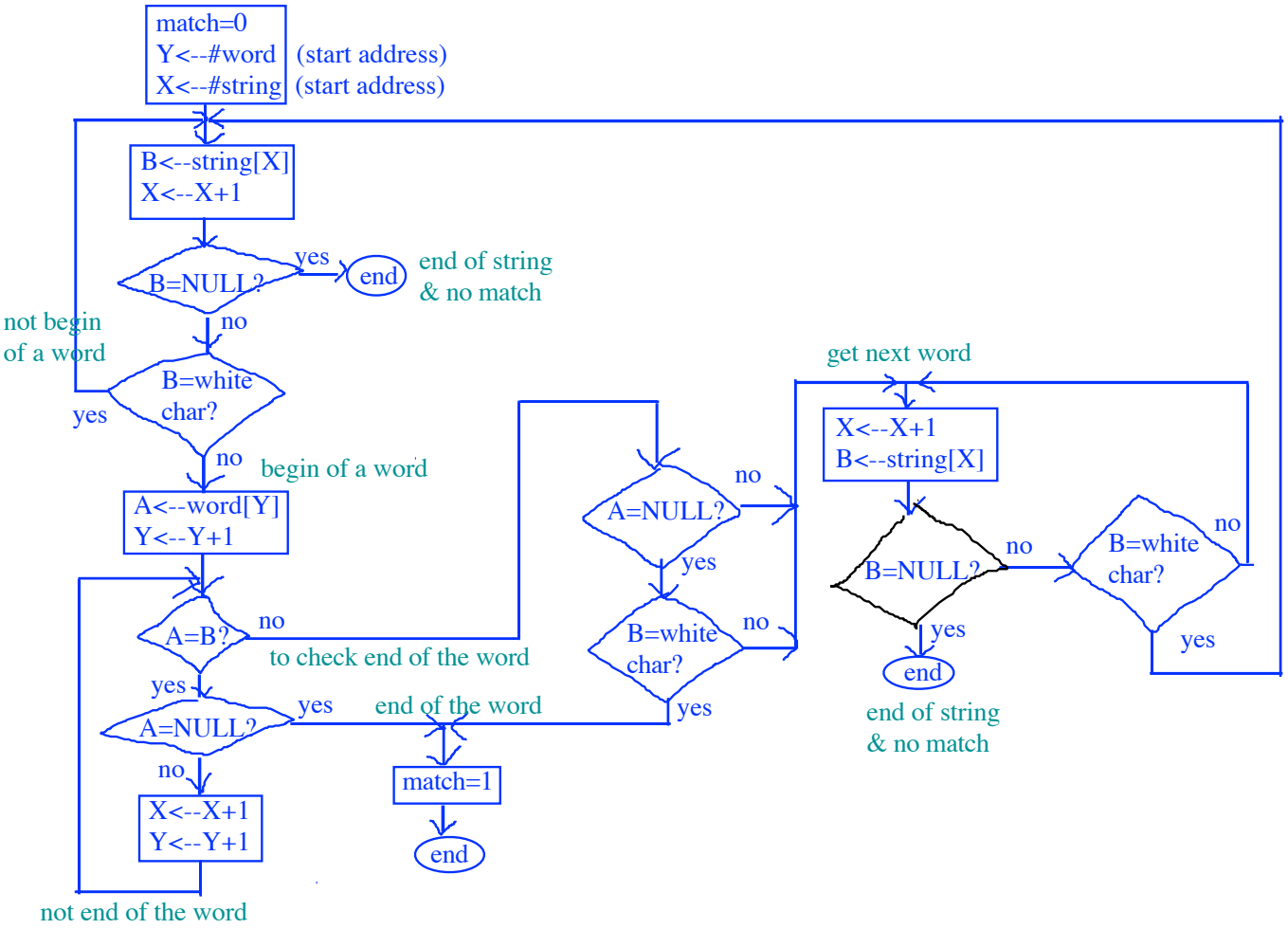
```
tab         equ     $09         ; ASCII table of special characters; tab key
sp          equ     $20         ; space key
cr          equ     $0D         ; enter key
lf          equ     $0A         ; line feed
            org     $1000
char_cnt    rmb     1           : character counter
word_cnt    rmb     1           ; word counter
string_x    fcc     "this is a strange test string to count chars and words."
            fcb     0           ; Null character for string termination

            org     $2000
            ldx     #string_x   ; load start address of the string   (X=$1002)
            clr     char_cnt
            clr     word_cnt
```

```
ldab   0,X
inx
string_lp       ldab    1,x+            ; get one character and move pointer to next char
                lbeq    done            ; is this the end of the string? (=$00=NULL?)
                inc     char_cnt
; the following 8 instructions skip white space characters between words
                cmpb    #sp
                beq     string_lp
                cmpb    #tab
                beq     string_lp
                cmpb    #cr
                beq     string_lp
                cmpb    #lf
                beq     string_lp
; a non-white character is the start of a new word
                inc     word_cnt
wd_loop         ldab    1,x+            ; get one character and move pointer to next char
                beq     done
                inc     char_cnt
; the following 8 instructions check the end of a word
                cmpb    #sp
                lbeq    string_lp
                cmpb    #tab
                lbeq    string_lp
                cmpb    #cr
                lbeq    string_lp
                cmpb    #lf
                lbeq    string_lp
                bra     wd_loop
done            swi
                end


; md $1000     37  0B  74  68  69
; number of characters = $37 (=55, includes white space and period)
; number of words = $0B (=11), $74=t; $68=h; …
```

Word Matching

- Searching process starts from the beginning of the given string, and looks for the next new word from the given string and compare it with the given word.
- The comparison is performed character-by-character until two words are found to be either matched or unmatched.
- If not equal, then the remaining characters of the word in the string are skipped.
- The comparison must be performed one character beyond the last character of the given word if the comparison is matched, and there are 3 possible outcomes:

1. the matched word is not the last word in the string. Comparison of the last characters will yield "not equal."
2. the matched word is the last word of the given string, but comparison of the last characters will again give the result "not equal."
3. the matched word is the last word of the given string, and it is followed by the NULL character. The comparison result for the last character is "equal."




```
tab          equ   $09
sp           equ   $20
cr           equ   $0D
lf           equ   $0A
period       equ   $2E
comma        equ   $2C
semicolon    equ   $3B
exclamation  equ   $21
null         equ   $0
```

; ASCII table of special characters

match up to here

"I am a testing program to perform a test."+NULL
↑ --->
X = string pointer

"test"+NULL
↑ --->
Y = word pointer

```
                org     $1000
match           rmb     1               ; reserve 1 byte for result

                org     $2000
                clr     match
                ldx     #string_x       ; load X the start address of the string
loop            ldab    1,x+            ; copy the first character of the string into B,
                                        ; then increment X by 1 to point to next character
; following 10 instructions skip white spaces to look for next word
                tstb                    ; B contains a characer in the string
                beq     done            ; if (B)=NULL, the end of string, jump to "done"
                cmpb    #sp             ; if not, test (B)= white characters
                beq     loop            ; skip white characters
                cmpb    #tab
                beq     loop
                cmpb    #cr
                beq     loop
                cmpb    #lf
                beq     loop
; the first nonwhite character is the beginning of a new word to be compared
                ldy     #word_x         ; load Y the start address of the matching word
                ldaa    1,y+            ; A contains a character in the matching word
                                        ; increment Y by 1 to point to next character
next_ch         cba                     ; compute (A)-(B)
                bne     end_of_wd       ; if not equal, jump to "end of word" to check if
                                        ; the end of word is reached special characters
                cmpa    #null           ; check if the end of the matching word is reached
                beq     matched         ; if yes, then there is a match
                ldaa    1,y+            ; if not, get next character from the matching word
                ldab    1,x+            ; get the next character from the string
                bra     next_ch         ; repeat the process
; the following 10 instructions check to see if the end of the given word is reached
end_of_wd       cmpa    #null           ; if the special character is not NULL,
                bne     next_wd         ; then not the end of string
                cmpb    #cr             ; matched in other special characters
                beq     matched
                cmpb    #lf             ; a word is surrounded by white space
                beq     matched         ; or punctuation
                cmpb    #tab
                beq     matched
                cmpb    #sp
                beq     matched
                cmpb    #period
                beq     matched
                cmpb    #comma
                beq     matched
```

check next characters

check if B contains a white character

if yes, word is matched

```
            cmpb    semicolon
            beq     matched
            cmpb    exclamation
            beq     matched

next_wd     ldab    1,x+        ; Is the next character of the string NULL?
            beq     done        ; if yes, the end of string is reached
            cmpb    #cr         ; if not, is it a special character?
            lbeq    loop        ; if yes, repeat the matching process
            cmpb    #lf
            lbeq    loop
            cmpb    #tab
            lbeq    loop
            cmpb    #sp
            lbeq    loop
            bra     next_wd
matched     ldab    #1
            stab    match
done        swi

string_x    fcc     "This string contains certain number of words to be matched."
            fcb     0
word_x      fcc     "word"
            fcb     0
            end
```

String Insertion

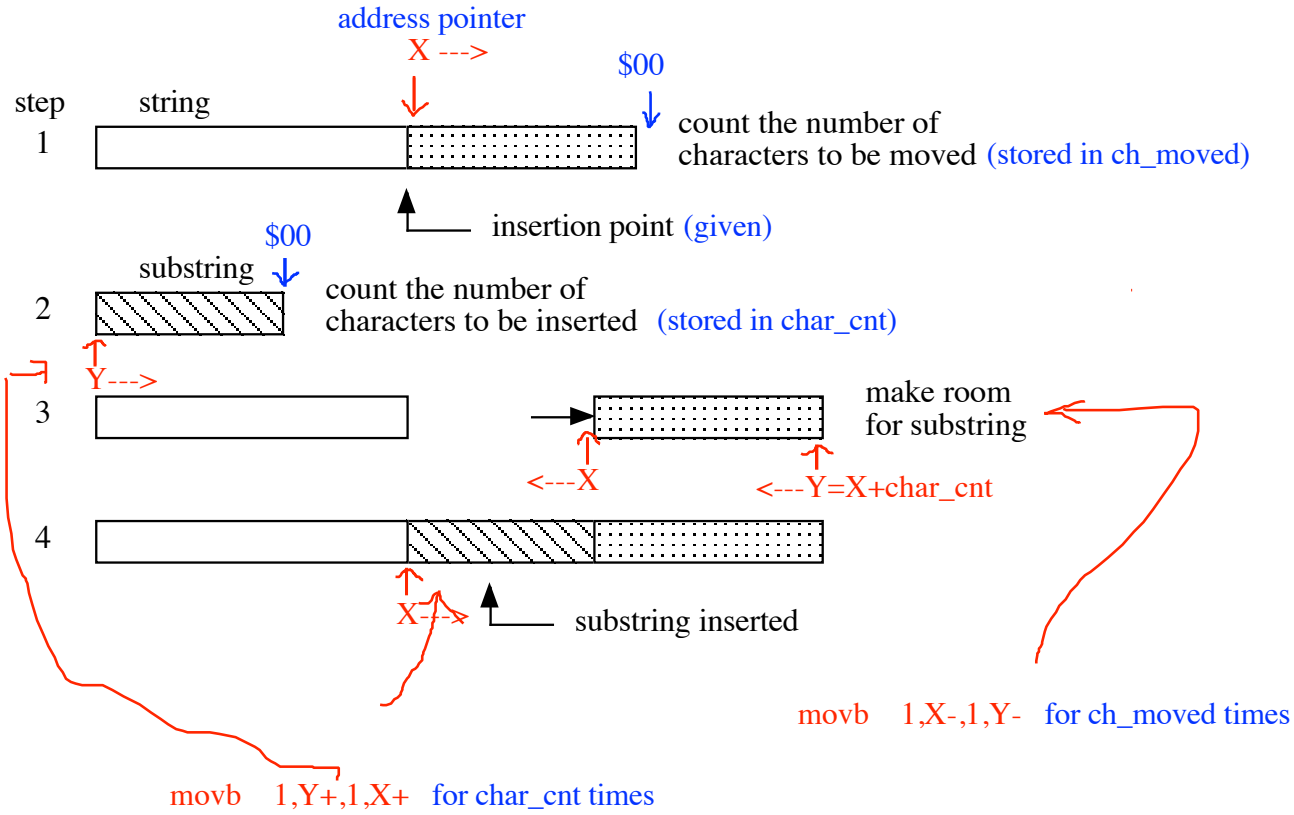
- The address pointers to the string and the substring to be inserted are given.
- All of the characters starting from the inserting point until the end of the string must be moved by the distance equal to the length of the substring.

Step 1: Count the number of characters that need to be moved. The NULL character is included because the resultant new string must be terminated by a NULL character.

Step 2: Count the number of characters in the substring to be inserted, excluding the NULL character.

Step 3: Move the characters in the string starting from the insertion point until the end of string (move the last character first).

Step 4: Insert the substring (the first character of the substring is inserted first).

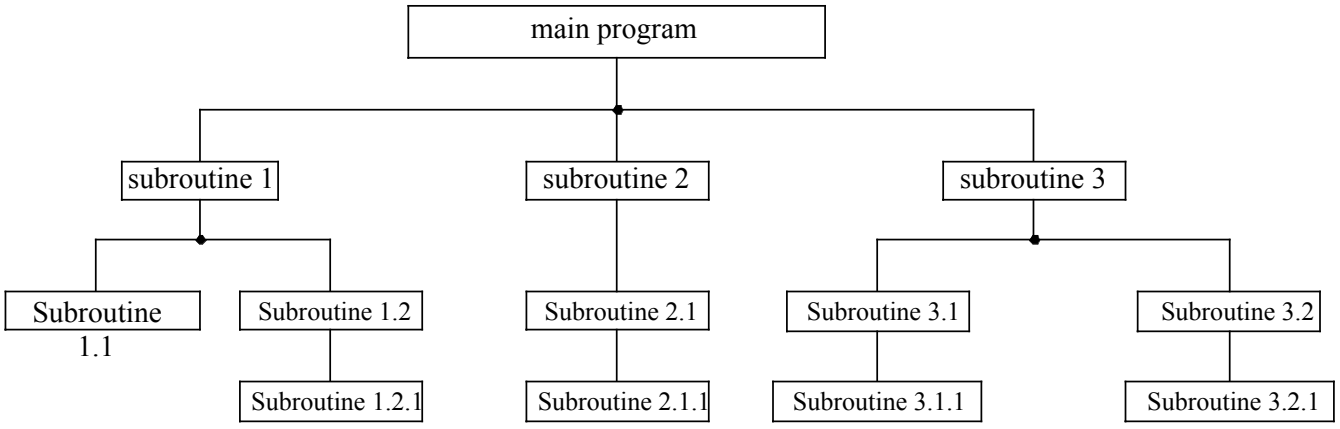



```
                org     $1000
ch_moved        rmb     1               ; reserve 1 byte for substring counter
char_cnt        rmb     1               ; reserve 1 byte for character to be moved
sub_strg        fcc     "the first and most famous "
                fcb     0               ; null character to terminate string
string_x        fcc     "Yellowstone is national park."
                fcb     0               ; null character to terminate string
offset          equ     15              ; position of insertion point in string
ins_pos         equ     string_x+offset ; memory location of insertion point

                org     $2000
; the next 7 instructions count the number of characters to be moved
                ldaa    #1              ; initialize the counter to count the number
                staa    ch_moved        ; of characters after the insertion point
                ldx     #ins_pos        ; use X to point to the insertion point
cnt_moved       ldaa    1,x+            ; load A with a string character
                beq     cnt_chars       ; Is it a NULL? If yes, counting is completed
                inc     ch_moved        ; if not, continue to count the number of
why needs dex?  bra     cnt_moved       ; characters after the insertion point
cnt_chars       dex                     ; subtract 1 from X so it points to NULL
                ldy     #sub_strg       ; use Y as a pointer to the substring address
                clr     char_cnt        ;
```

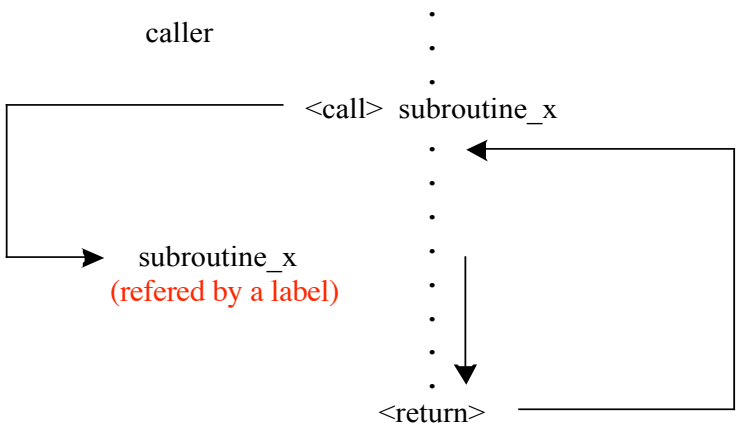
```
; the following 3 instructions count the move distance
char_loop   ldab   1,y+              ;
            beq    mov_loop          ; is it a NULL? If yes, counting is completed
            inc    char_cnt          ; store the number of characters in substring
            bra    char_loop         ;
-------------------------------------------------------------------------------
mov_loop    tfr    x,y               ; make a copy of X to Y
            ldab   char_cnt
            aby                      ; compute the copy destination: Y+B → Y
            ldab   ch_moved          ; place the # of characters to be moved in B

; two following instructions copy the contents of the address pointed to by X to the
; location of Y. Afterwards, X and Y are decremented by 1. (In the beginning, X points
; to the end of the string and Y points to the end of the inserted string.)
again       movb   1,x-,1,y-         ; make room for insertion
            dbne   b,again           ;
            ldx    #ins_pos          ; X points to the insert point of string
            ldy    #sub_strg         ; Y points to the substring
            ldab   char_cnt          ; B has the # of characters of substring
insert_lp   movb   1,y+,1,x+         ; copy (Y) to X; then increment X and Y
            dbne   b,insert_lp       ; decrement B. If B≠0, repeat.
            swi
            end
```

Subroutines

- Good program design is based on the concept of modularity---the partitioning of a large program into subroutines.
- Simplifies the design of a complex program by the ***divide-and-conquer*** approach.
- Sequences of instructions that can be called from various places in the program.
- Allow the same operations to be performed with different parameters.



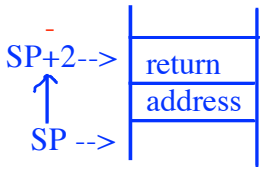
- Program execution must return to the point immediately after the subroutine call (this address is called ***return address***) when the subroutine completes its computation.
- Subroutine call instruction will save the return address in the ***system stack***.
- When completing the task, a return instruction in the subroutine will retrieves the return address from the system stack and transfer CPU control to it.



BSR ***<rel>*** ; branch to subroutine with address offset given by optional ***rel***

- Use the relative addressing mode to specify the subroutine to be called.
- Relative address is specified by using **a label** and the assembler will figure out the relative offset and place it in the program memory.
- When executed, the stack pointer (SP) is decremented by 2, the return address is saved in the stack, the offset in the instruction is added to the current PC value, and finally instruction execution is continued from there.

```
BSR    bubble        ; bubble is the label containing the offset
```

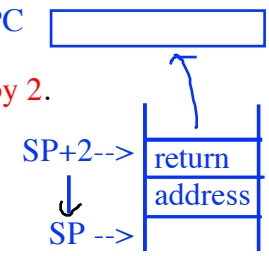


JSR ***<opr>*** ; jump to subroutine with the given address ***opr***

- More general than BSR; use direct, extended, indexed, and indexed indirect addressing modes to specify the subroutine to be called.
- The subroutine can be located anywhere within 64 Kbyte.
- First saves the return address in the stack and then jump to execute the subroutine.

```
JSR    $FF           ; call the subroutine located at $FF
JSR    sq_root       ; call the subroutine sq_root
JSR    0,X           ; call the subroutine pointed to by X
```

RTS ; return from subroutine



- Loads PC with a 16-bit value pulled from the stack and increments the SP by 2.
- Program execution continues at the address restored from the stack.

CALL ***<opr>*** ; to be used in expanded memory

- Designed to work with expanded memory (larger than 64 Kbytes) supported by some 68HC12 members, which treat the 16-Kbyte memory space from $8000 to $BFFF as a program memory window.
- See page 129 of the textbook for detail.

RTC ; return from CALL

- Terminates subroutines in expanded memory invoked by CALL.

Issues in Subroutine Calls

Caller = the program unit that makes the subroutine call
Callee = the subroutine

1. ***Parameter passing*****:** The caller usually wants the subroutine to perform computations using the parameters passed to it.

from main to subroutine

- Use CPU registers – convenient when there are only a few parameters to be passed.
- Use the stack – the stack must be cleaned up after the computation is completed. This can be done by either the caller or the callee.
- Use global memory – accessible to both the caller and callee.

2. ***Returning results*****:**

from subroutine to main

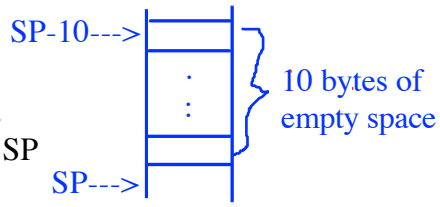
- Use CPU registers.
- Use the stack – the caller created a hole in which the result will be placed.
- Use global memory.

3. ***Local variable allocation*****:** A subroutine may need memory locations (pointed to by SP) to hold temporary variables (i.e., local variables) and results.

in subroutine right after entry

- Allocated by the callee in the stack so that they are not accessible to any other program units.
- **LEAS** is the most efficient method of local variable allocation. It loads the SP with an effective address specified by the program.

```
leas  -10,sp     ; allocate 10 bytes in the stack for local variables
                 ; subtracts 10 from SP and puts the difference to SP
```



4. ***Local variable deallocation***

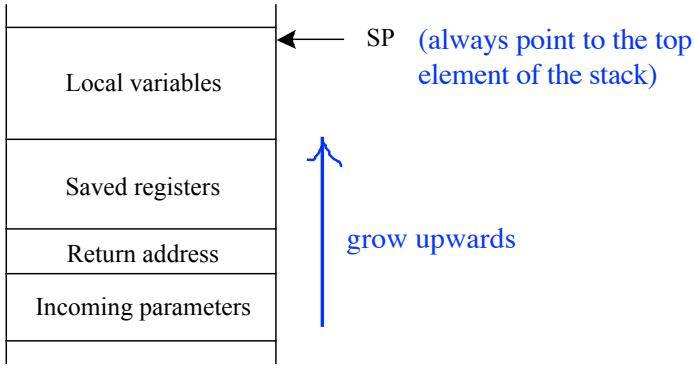
in subroutine right before exit

- The stack space allocated to local variables must be deallocated by the subroutine before the subroutine returns to the caller.

```
leas   10,sp     ; deallocate 10 bytes from the stack before returning to the caller
                 ; adds 10 to SP and puts the sum to SP
```

Stack Frame

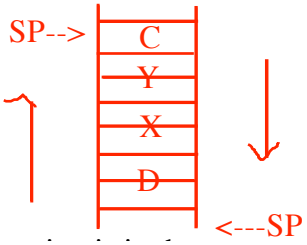
- Stack is heavily used during a subroutine call.
  - the caller passes parameters to the callee.
  - the callee saves registers and allocates local variable in the stack.
- Stack frame is the region in the stack that holds incoming parameters, subroutine return address, local variables, and saved registers.
- Stack frame is also called the activation record of the subroutine.



If a subroutine saves registers when it is entered, it must restore them in the reverse order (i.e., first-in-last-out, FILO) before returning to the caller.

Example: Right after entering the subroutine, the following registers are saved in following order:

```
pshd             ; from regiters to stack
pshx             ; watch out 1 vs. 2 bytes
pshy
pshc
```

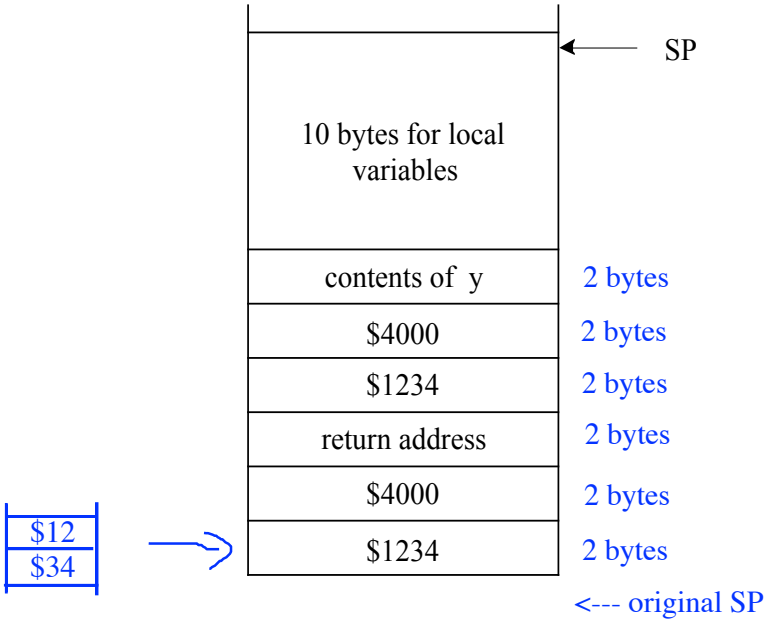


Before leaving the subroutine, restoration is in the reverse order:

```
        pulc            ; from stack back to registers
        puly
        pulx
        puld
```

Example: Draw the stack frame for the following program segment after LEAS –10,SP is executed:

```
                ldd     #$1234          ; pushes a 16-bit word from D into the stack
                pshd                    ;         "
                ldx     #$2000          ; pushes a 16-bit word from X into the stack
                pshx                    ;         "
                jsr     sub_xyz         ; return address is stored into the stack
                …
sub_xyz         pshd                    ; the subroutine pushes the contents of D, X, Y
                pshx                    ; into the stack
                pshy                    ;         “
                leas    -10,sp          ; allocate 10 bytes in the stack
                …
```



Examples of Subroutines

Finding the **Greatest Common Divisor** (GCD) of positive integers m and n.

Step 1: If m = n, then
gcd ← m;
return.

GCD(3,3)=3

Step 2 if n < m, then swap m and n.

GCD(9,3)=GCD(3,9)=3
make sure the first number is smaller

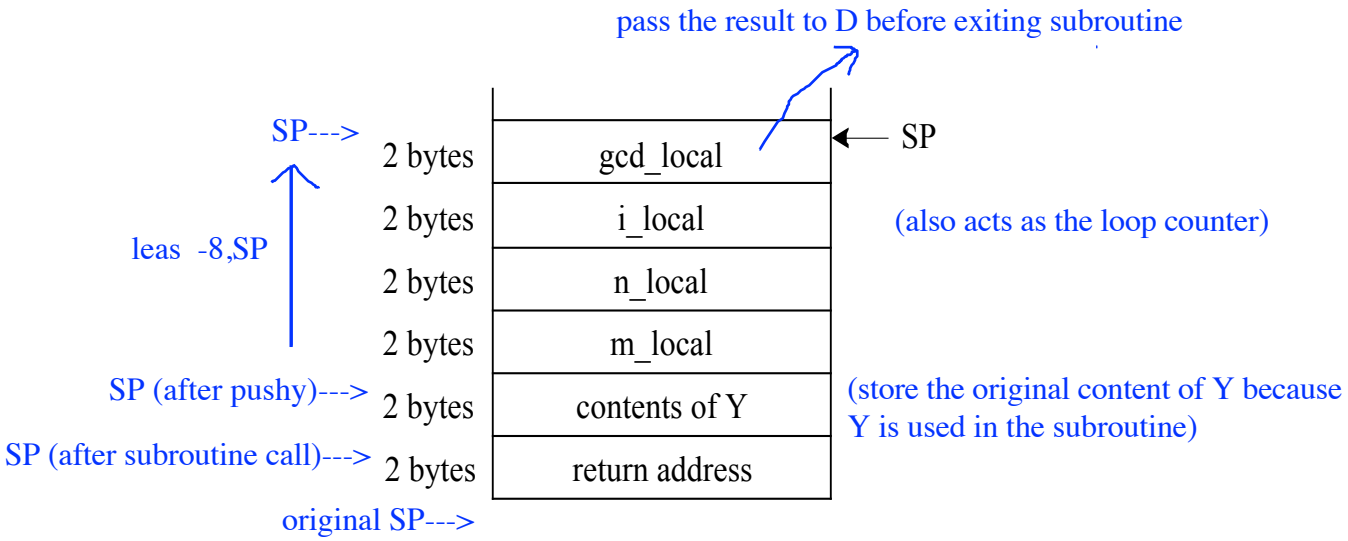
Step 3 if m = 1, then
gcd ← 1;
return.

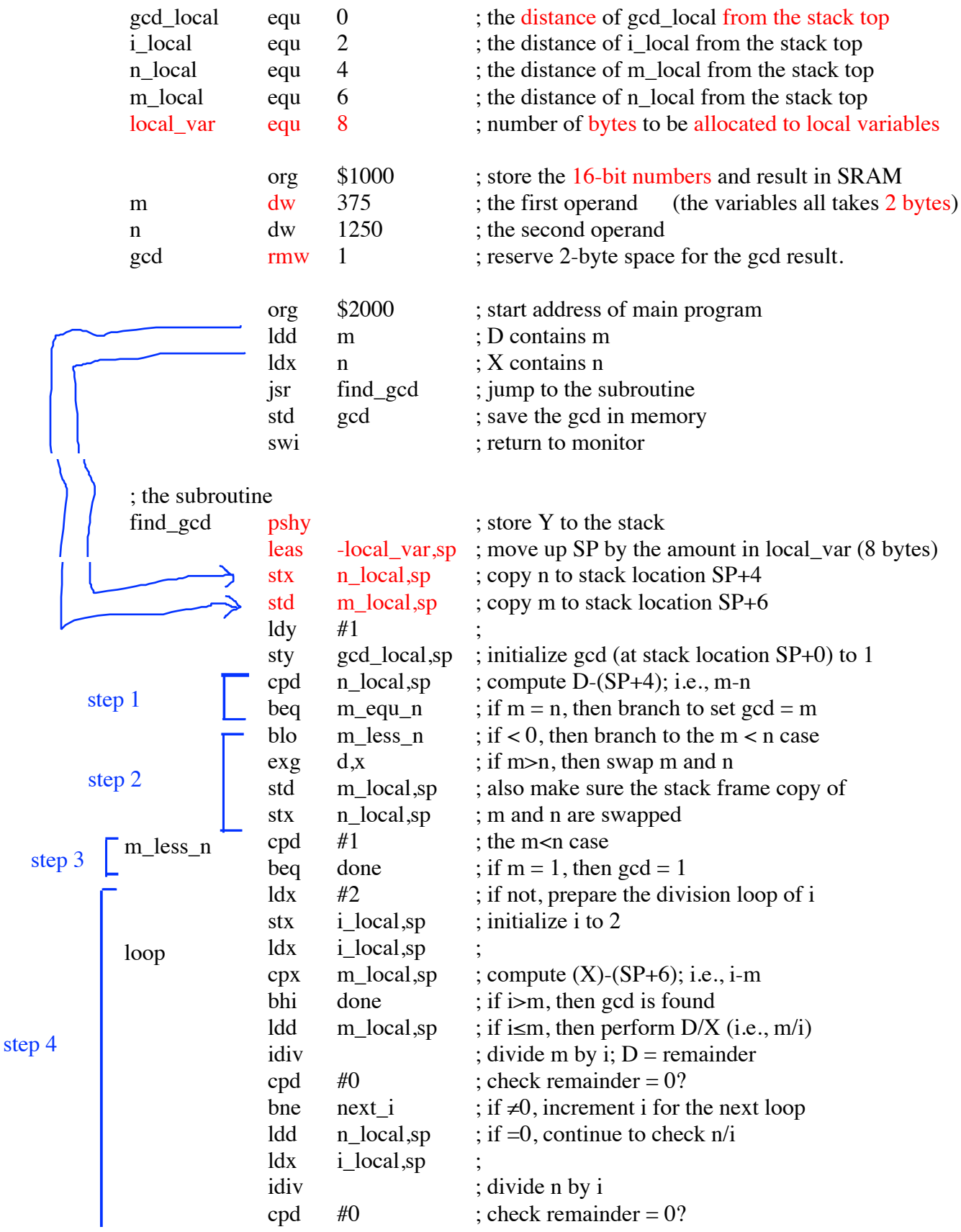
GCD(1,n)=1

Step 4
For i = 2 to m do
if remainder(m/i) = 0 and remainder(n/i) = 0, then
gcd ← i;
return.

Assume two 16-bit unsigned integers m and n are stored in X and D.

- Subroutine parameters are passed in X and D, as global variables.
- Subroutine local variables:
  1. **gcd_local** = temporary gcd.
  2. **i_local** = a value to test divide m and n.
  3. **n_local** = hold the incoming n value.
  4. **m_local** = hold the incoming m value.




gcd_local equ 0 ; the distance of gcd_local from the stack top
i_local equ 2 ; the distance of i_local from the stack top
n_local equ 4 ; the distance of m_local from the stack top
m_local equ 6 ; the distance of n_local from the stack top
local_var equ 8 ; number of bytes to be allocated to local variables
org $1000 ; store the 16-bit numbers and result in SRAM
m dw 375 ; the first operand (the variables all takes 2 bytes)
n dw 1250 ; the second operand
gcd rmw 1 ; reserve 2-byte space for the gcd result.
org $2000 ; start address of main program
ldd m ; D contains m
ldx n ; X contains n
jsr find_gcd ; jump to the subroutine
std gcd ; save the gcd in memory
swi ; return to monitor
; the subroutine
find_gcd pshy ; store Y to the stack
leas -local_var,sp ; move up SP by the amount in local_var (8 bytes)
stx n_local,sp ; copy n to stack location SP+4
std m_local,sp ; copy m to stack location SP+6
ldy #1 ;
sty gcd_local,sp ; initialize gcd (at stack location SP+0) to 1
step 1
cpd n_local,sp ; compute D-(SP+4); i.e., m-n
beq m_equ_n ; if m = n, then branch to set gcd = m
blo m_less_n ; if < 0, then branch to the m < n case
step 2
exg d,x ; if m>n, then swap m and n
std m_local,sp ; also make sure the stack frame copy of
stx n_local,sp ; m and n are swapped
step 3
m_less_n cpd #1 ; the m<n case
beq done ; if m = 1, then gcd = 1
ldx #2 ; if not, prepare the division loop of i
stx i_local,sp ; initialize i to 2
loop ldx i_local,sp ;
cpx m_local,sp ; compute (X)-(SP+6); i.e., i-m
bhi done ; if i>m, then gcd is found
step 4
ldd m_local,sp ; if i≤m, then perform D/X (i.e., m/i)
idiv ; divide m by i; D = remainder
cpd #0 ; check remainder = 0?
bne next_i ; if ≠0, increment i for the next loop
ldd n_local,sp ; if =0, continue to check n/i
ldx i_local,sp ;
idiv ; divide n by i
cpd #0 ; check remainder = 0?

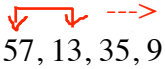
```
                bne     next_i          ; if ≠0, increment i for the next loop
                ldd     i_local,sp      ; if =0, i is the gcd
                std     gcd_local,sp    ; set i as the current gcd
next_i          ldx     i_local,sp      ; increment i for the next loop
                inx                     ;
                stx     i_local,sp      ; increment i
                jmp     loop            ; repeat the loop
m_equ_n         ldd     m_local,sp      ; pass the gcd to D
                bra     exit
done            ldd     gcd_local,sp    ; pass the gcd to D
exit            leas    local_var,sp    ; deallocate local variables
                puly                    ; restore Y
                rts                     ; return to main program

                end
```

Bubble sort

- Sorting is useful for improving the searching speed when an array or a file needs to be searched many times.
- Bubble sort is a simple widely known sorting method.
- Basic idea is to go through the array sequentially several times.
  - Each iteration consists of comparing each element in the array with its successor (i.e., compare x[i] with x[i+1]) and interchanging the two elements if they are not in proper order. (pairwise compare & swap)
  - The element x[n-i] will be in its proper position after iteration i.
  - Array of n elements requires no more than n-1 iterations.
  - Since elements in positions greater than or equal to n-i are already in proper position after iteration i, they need not be considered in succeeding iterations. Thus, in iteration i, n-i comparisons are made.
  - Each number slowly bubbles up to its proper position.

requires 2 loops

pairwise compare and swap



Example: Sort the array 157, 13, 35, 9, 98, 810, 120, 54, 40, 30 in ascending order.

inner loop

After the 1st iteration, the array becomes 13, 35, 9, 98, 157, 120, 54, 10, 30, 810. n-1 compares
The largest element (in this case 810) is in its proper position within the array.

outer loop

After the 2nd iteration, the array becomes 13, 9, 35, 98, 120, 54, 10, 30, 157, 810. n-2 compares
The second largest element (in this case 157) is in its proper position within the array.

After the 3rd iteration, the array becomes 13, 9, 35, 98, 54, 10, 30, 120, 157, 810. n-3 compares

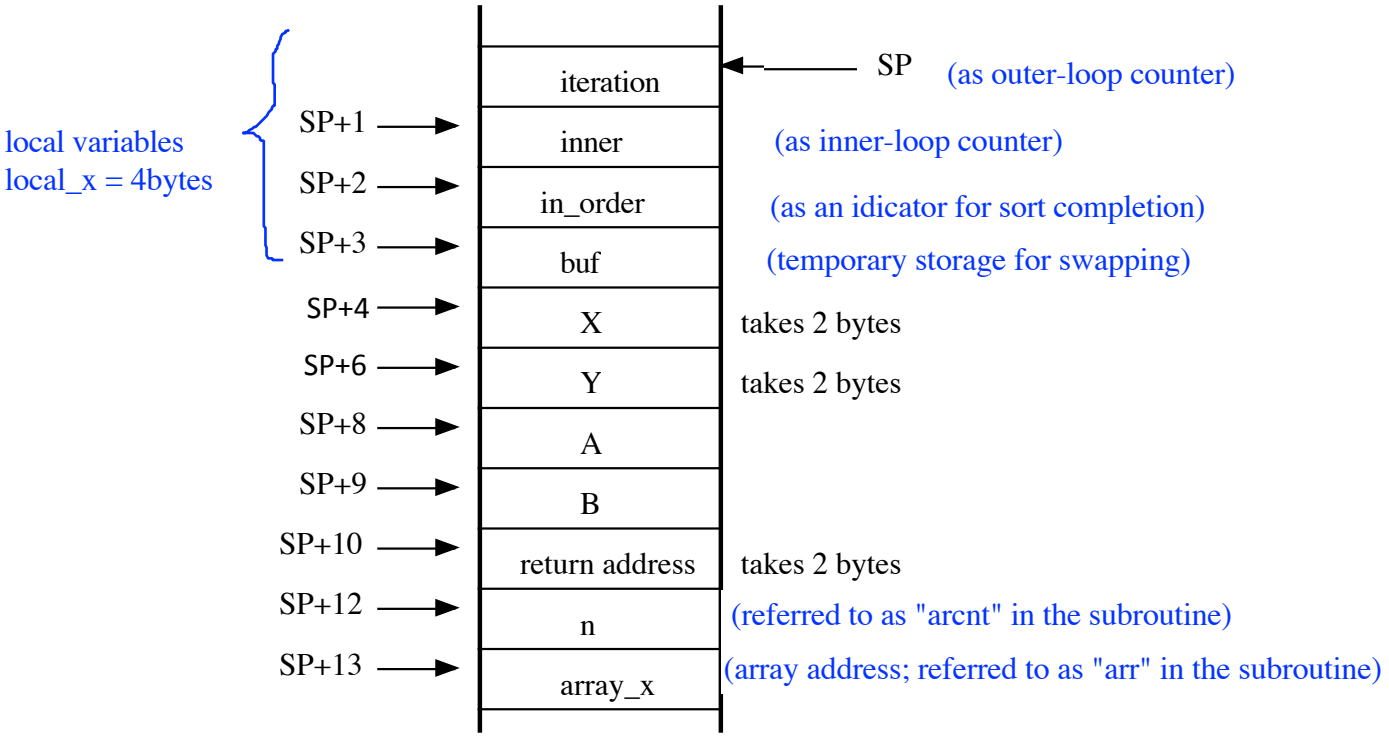
…

After the 9th iteration, the array becomes 9, 10, 13, 30, 35, 54, 98, 120, 157, 810. n-9 compares

only needs at most n-1=9 iterartins

Example: Use an array that consists of N 8-bit unsigned integers for testing a bubble-sort subroutine.

- Pass the base address of the array and the array count in the stack.
- Four bytes are needed for local variables:
- Buf: buffer space for swapping adjacent elements.
- In_order: flag to indicate whether the array is in order after an iteration.
- Inner: loop count for each iteration.
- Iteration: number of iterations remains to be performed.




```
array start address  arr         equ   13          ; distance of the variable arr from stack top
number of elements   arcnt       equ   12          ; distance of the variable arcnt from stack top
                     buf         equ   3           ; distance of local variable buf from stack top
                     in_order    equ   2           ; distance of local variable in_order from stack top
                     inner       equ   1           ; distance of local variable inner from stack top
                     iteration   equ   0           ; distance of local variable iteration from stack top
                     true        equ   1
                     false       equ   0
                     n           equ   30          ; array count
                     localx      equ   4           ; number of bytes used by local variables

                     org   $1000
array_x              db    3,29,10,98,54,9,100,104,200,92,87,48,27,22,71
                     db    1,62,67,83,89,101,190,187,167,134,121,20,31,34,54
```

```
                    org     $2000
                    lds     #$2000      ; initialize SP (can be ignored by default)
                    ldx     #array_x    ; X has the start address of the array
                    pshx                ;
                    ldaa    #n          ; A has the address of n
                    psha
                    jsr     bubble      ;
why needs this? --> leas    3,sp        ; deallocate space used by outgoing parameters
                    swi                 ; break to D-Bug12 monitor

        bubble      pshd
                    pshy
                    pshx
                    leas    -localx,sp  ; allocate space for local variables, move SP up 4
setup n-1 iteration ldaa    arcnt,sp    ; compute the number of iterations to be performed
A <--- n            deca                ; A ← n-1
                    staa    iteration,sp; number of iterations = n-1=29
        ploop       ldaa    #true       ; set array in_order flag to true before any iteration
                    staa    in_order,sp ;       "
                    ldx     arr,sp      ; use X as the array address pointer (SP+13)
                    ldaa    iteration,sp; use A as the loop counter          [A=29 initially]
                    staa    inner,sp    ; initialize inner loop count for each iteration
        cloop       ldaa    0,x         ; compare two adjacent elements
                    cmpa    1,x         ; compute arr[x]-arr[x+1]
                    bls     looptest    ; if <0, then continue on next array element
                    staa    buf,sp      ; if ≥ 0, then swap two adjacent elements
                    ldaa    1,x         ;       "
                    staa    0,x         ;       "
                    ldaa    buf,sp      ;       "
                    staa    1,x         ;       "
                    ldaa    #false      ; set the in-order flag to false
                    staa    in_order,sp ;       "
        looptest    inx                 ; compare next array element by increment X
                    dec     inner,sp    ; decrement the inner loop counter by 1
                    bne     cloop       ; if counter ≠0, repeat the inner loop
                    tst     in_order,sp ; if =0, test array in_order flag after each iteration
                    bne     done        ; if in_order=true; then the array is sorted
                    dec     iteration,sp; if not, decrement the loop counter
                    bne     ploop       ; repeat the outer loop
        done        leas    local,sp    ; deallocate local variables
                    pulx
                    puly
                    puld
                    rts

                    end
```

Finding the Square Root

One of the methods for finding the square root of an integer q is based on

$$\sum_{i=0}^{n-1} i = \frac{n(n-1)}{2}$$

The equation can be transformed into

$$q = n^2 = \sum_{i=0}^{n-1} (2i+1)$$

Run a loop with i as the loop counter to perform the summation:

if sum > q, then i+1 = square root of q.

Suppose we want to compute the square root of q, and n is the integer value that is closet to the true square root. One of the following three relationships is satisfied: $n^2 < q$, $n^2 = q$, and $n^2 > q$.

Example: Implement the square root algorithm on a 32-bit (or 4-byte) unsigned integer. The parameter q is pushed onto the stack and the square root of q (= 4 bytes) is returned in D (= 2 bytes).

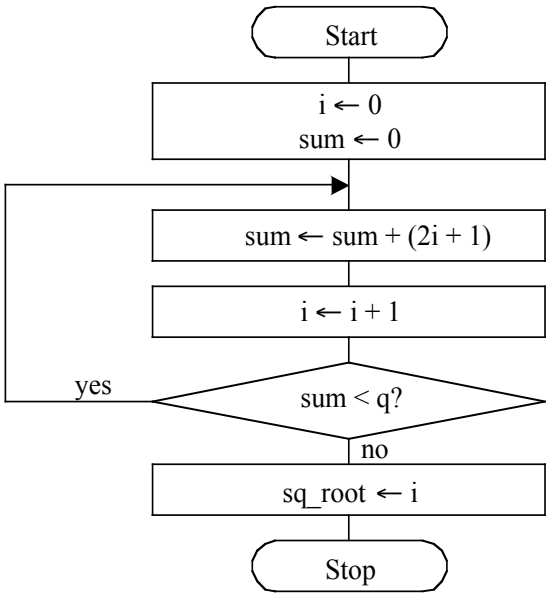
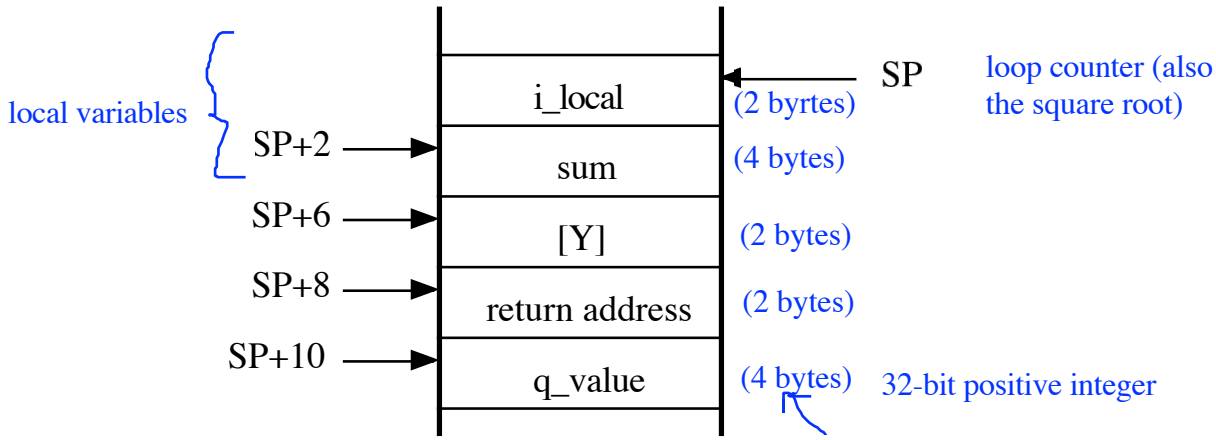


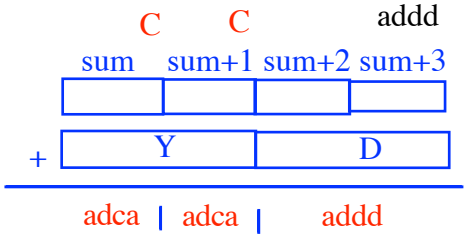
Figure 4.16 Algorithm for finding the square root of integer q.

- This algorithm will find the exact integer square root if the given number has one.
- If the given number does not have an exact integer square root, then the number returned by subroutine may not be the closet approximation.
- The algorithm will stop only when sum > q.
- The last i value may not be as close to the real square root as it the value i-1.
- This can be fixed easily by comparing $i^2 - q\_val$ and $q\_val - (i-1)^2$.




```
q_hi        equ   $000F        ; upper word of q          [q has 4 bytes]
q_lo        equ   $4240        ; lower word of q
i_local     equ   0            ; distance of local variable i from the stack top
sum         equ   2            ; distance of local variable sum from the stack top
q_val       equ   10           ; distance of incoming q from the stack top
localx      equ   6            ; number of bytes allocated to local variables

            org   $1000
sq_root     rmb   2            ; to hold the square root of q

            org   $2000
            ldd   #q_lo        ; push q to the stack (q-value the bottom of stack)
            pshd
            ldd   #q_hi
            pshd
            jsr   find_sq_root ; jump to the subroutine
            std   sq_root      ; the square root is found
            leas  4,sp         ; deallocate 4 bytes from the stack
            swi

find_sq_root:                  ; subroutine starts here
            pshy               ; save y in the stack
            leas  -localx,sp   ; allocate local variables (move SP up by 6)
            ldd   #0           ; initialize local variable i to 0
            std   i_local,sp   ;          "
            std   sum,sp       ; initialize local variable sum to 0
            std   sum+2,sp     ; sum has 4 bytes
loop        ldd   i_local,sp         ; load i to D
            ldy   #2                 ; load 2 to Y
            emul                     ; compute 2i (i.e., Y:D=DxY)
            addd  sum+2,sp           ; add 2i to sum
```

compute "sum+2i+1"



why needs Y-->D?

```
                std     sum+2,sp        ; the following instructions
                tfr     y,d             ; are needed because sum
                adcb    sum+1,sp        ; has 4 bytes (i.e., two 16-bit
                stab    sum+1,sp        ; additions)
                adca    sum,sp          ;
                staa    sum,sp          ;
```

compute "sum+2i+1"

```
                ldaa    #1              ; add 1 to sum
                adda    sum+3,sp        ; the following instructions
                staa    sum+3,sp        ; are needed because sum
                ldaa    sum+2,sp        ; has 4 bytes.
                adca    #0              ; (need to propagate carry to the
                staa    sum+2,sp        ; most significant byte of sum)
                ldaa    sum+1,sp
                adca    #0
                staa    sum+1,sp
                ldaa    sum,sp
                adca    #0
                staa    sum,sp
                ldd     i_local,sp      ; increment i by 1 for next iteration
                addd    #1
                std     i_local,sp
; compare sum to q_val by performing subtraction (need consider borrow)
                ldd     sum+2,sp        ; load lowest 2 bytes of sum to D
                subd    q_val+2,sp      ; (D)-q → D
                ldaa    sum+1,sp        ; load the second highest byte of sum to D
                sbca    q_val+1,sp      ; (A)-q-borrow → A
                ldaa    sum,sp          ; load the highest byte of sum to D
                sbca    q_val,sp        ; gives the C flag a 1 (borrow) or 0
                lblo    loop            ; if C=1 (i.e., sum<q), perform next iteration
                ldd     i_local,sp      ; place sq_root in D before return

exit            leas    localx,sp       ; deallocate space used by local variables
                puly
                rts

                end
```

End of Chapter 4