Program to read up to 100 ASCII characters, sum up any integers, change upper to lower case and store in msg.
Note that numbers are in the range $30 - $39 and the number can be found by setting all but the 4 least significant bits to 0.
Note that upper case letters are in the range $41 to $5A and lower case differ by having bit 6 set to 1 (where bit 1 is the least sig. bit). One can get the lower case character by "or-ing" the upper case character bit-by-bit with $20, or simply adding $20 since the bit is not set for the upper case character (no chance of "carry").
Combined solution to exercises 1-3 of M68000 Programming Notes, 6/10/99
-D. Pellett 3/18/04 (beta version)

```assembly
xref getchar, strout, decin, decout, newline, stop

start:  lea msg1, a0 ; ask for number of characters to read (100 max)
        move.w #50, d0
        jsr strout
        jsr decin ; get the number (in d0)
        jsr newline
        cmp.w #100, d0 ; see if it fits
        ble fits ; it fits - branch
        move.w #100, nchar ; too big - set nchar = 100
        lea msg2, a0 ; send message to input 100 max
fits:   move.w d0, nchar ; enter number in nchar
        lea msg3, a0 ; send message to input characters
        move.w #26, d0
        jsr strout
        jsr newline
        jmp readem

readem: lea msg, a0 ; put address of msg in a1
        move.w nchar, d1 ; number of bytes to read
        clr d2 ; clear the register for the sum
        jmp enter ; enter loop at end

loop:  jsr getchar ; getchar puts the character in d0
        jsr addch ; subroutine to test and add integers, change case
        move.b d0, (a1)+ ; move the character to memory
enter: dbra d1, loop ; subtract 1 from d1 and see if done

; now output the information
        lea msg4, a0
        move.w #37, d0 ; output info message
        jsr strout
        jsr newline
        lea msg, a0 ; set up for outputting character string
        move.w nchar, d0
        jsr strout ; output the string
        jsr newline
        lea msg5, a0 ; output sum message
        move.w #31, d0
        jsr strout
        move.w d2, d0 ; output the sum
        jsr decout
        jsr newline
        jsr stop ; end of program
```
; Subroutine addch tests ASCII character in d0
; If it is a number, adds it to the sum in d2
; If it is an upper case letter, changes it to lower case
; Lower case letter or number is returned in d0

addch:  and.b #$7F, d0          ; mask off parity bit of character
        cmp.b #$30, d0          ; see if it is less than $30
        blt skip               ; if so, skip
        cmp.b #$39, d0          ; see if it is greater than $39
        bgt skip               ; if so, skip
        move.w d0, d3          ; is a number - put in d3 (don't change d0)
        and.w #$000F, d3       ; get the number by masking off upper bits
        add.w d3, d2           ; add to sum in d2
        rts                    ; return from subroutine

skip:   cmp.b #$41, d0          ; see if it is less than $41 (A)
        blt skip1              ; if so, skip1
        cmp.b #$5A, d0          ; see if it is greater than $5A (Z)
        bgt skip1              ; if so, skip1
        add.w #$20, d0          ; upper case letter - convert to lower

skip1:  rts

data

msg1:  db  'Enter the number of characters to read (100 max): '  
msg2:  db  'Your entry was too big. Enter 100 characters: '        
msg3:  db  'Now enter the characters: '                           
msg4:  db  'Here is the string (all lower case): '               
msg5:  db  'The sum of numbers entered is: '                     
msg:   dw  100                                                    ; set aside 100 bytes of storage
nchar: dw  20                                                    ; number of characters to read
        even
        end
Programming problem solution:

(a) 1, 26, 27, 28, 29.

(b) In Macintosh programming, A5 is used as a pointer to the data area for the code. That is, it contains the address of the beginning of the data area. References to the data use indirect addressing relative to A5. Thus, if A5 were changed, incorrect addresses would be used and the program would fail.

(c) Again indirect addressing is used, this time relative to the program counter.

(d) Where you see the #',s: 18, 19, 21, 23.

(e) The subroutine return address would not be placed on the stack so the return from the subroutine would not work.

(f) The stack pointer contains the address of the last type used in the stack. When the subroutine is called, the return address (4 bytes) is placed on the stack so the number is reduced by 4 (the stack grows downward): $69C774.

(g) Actually, there is a special instruction for comparing with immediate data, which this line uses. So the assembler program will fix this detail and insert cmpilb.

(h) The cmpil instruction (see above) must have the immediate data as the first argument, so the arguments must not be reversed.

(i) $37$ is a number so this routine moves off all but the 4 lower bits of dx, leaving $7$.

(j) $37 = 3 \times 16 + 7 = 55$.

*Notation: $69C774$ is expressed in hexadecimal (base 16). This is indicated by the $.$
Diagrams from Horowitz and Hill show connections of an input register and an output register to an M68008 microprocessor.

(a) Make a timing diagram showing the clock signal on the 74LS574 octal D latch when the output register is accessed with its valid address during a write cycle. Also show the output of the address decoder. Fill these two signals in on the copy of the write cycle timing diagram which shows R/W, the D signal, VALID ADDRESS on the address lines and VALID DATA on the data lines (lumped together as usual). (i.e., use Figure below).

(b) Explain why the output register can have its tri-state output lines enabled continuously without causing conflicts with the computer data bus.

(b) The output register reads data from the data bus at the time indicated above. Its output connects to the peripheral, not to the data bus, so the output can remain enabled continuously (unless the peripheral device prefers otherwise).