Character Conversion: Uppercase to Lowercase

To convert an uppercase letter to lowercase, we note that ASCII codes for the uppercase letters ‘A’ to ‘Z’ form a sequence from 65 to 90.

The corresponding lowercase letters ‘a‘ to ‘z’ have codes in sequence from 97 to 122.

We say that ASCII codes form a **collating sequence** and we use this fact to sort textual information into alphabetical order.

To convert from an uppercase character to its lowercase equivalent, we add 32 to the ASCII code of the uppercase letter to obtain the ASCII code of the lowercase equivalent.

To convert from lowercase to uppercase, we subtract 32 from the ASCII code of the lowercase letter to obtain the ASCII code of the corresponding uppercase letter.

The number 32 is obtained by subtracting the ASCII code for ‘A’ from the ASCII code for ‘a’
(i.e. ‘A’ - ‘a’ = 97 - 65 = 32).

Example 3.19: Write a program to prompt the user to enter an uppercase letter, read the letter entered and display the corresponding lowercase letter. The program should then convert the letter to its to lowercase equivalent and display it, on a new line.

```
; char.asm: character conversion: uppercase to
lowercase

        .model small
        .stack 100h

CR      equ     13d
LF      equ     10d

        .data
msg1        db  'Enter an uppercase letter: $'
result      db  CR, LF, 'The lowercase equivalent is:
$'

        .code

; main program
start:
        mov  ax, @data
        mov  ds, ax

        mov  dx, offset msg1
        call puts          ; prompt for uppercase letter
        call getc          ; read uppercase letter
        mov  bl, al        ; save character in bl

        add  bl, 32d       ; convert to lowercase

        mov  dx, offset result
        call puts          ; display result message
        mov  dl, bl
        call putc          ; display lowercase letter

        mov ax, 4c00h
        int 21h            ; return to ms-dos
```

```
; user defined subprograms

puts:               ; display a string terminated by $
                    ; dx contains address of string
       mov ah, 9h
       int 21h      ; output string
       ret

putc:               ; display character in dl
       mov ah, 2h
       int 21h
       ret

getc:               ; read character into al
       mov ah, 1h
       int 21h
       ret

       end start
```

Executing this program produces as output:

```
Enter an uppercase letter: G
The lowercase equivalent is: g
```

The string `result` is defined to begin with the Return and Line-feed characters so that it will be displayed on a new line. An alternative would have been to include the two characters at the end of the string `msg1`, before the `'$'` character, e.g.

```
msg1   db   'Enter an uppercase letter: ',CR, LF, '$'
```

After displaying `msg1`, as defined above, the next item to be displayed will appear on a new line.

Exercises

3.11 Modify the above program to convert a lowercase letter to its uppercase equivalent.

3.12 Write a program to convert a single digit number such as 5 to its character equivalent '5' and display the character.

I/O Subprogram Consistency

We have now written three I/O subprograms: `putc`, `getc` and `puts`.

One difficulty with these subprograms is that they use different registers for parameters based on the requirements of the MS-DOS I/O subprograms.

This means that we have to be careful to remember which register (`al`, `dl`, `dx`) to use to pass parameters.

A more consistent approach would be to use the same register for passing the parameters to all the I/O subprograms, for example the `ax` register could be used.

Since we cannot change the way MS-DOS operates, we can do this by modifying our subprograms. We will use `al` to contain the character to be displayed by `putc` and `ax` to contain the address of the string to be displayed by `puts`. The `getc` subprogram returns the character entered in `al` and so does not have to be changed.

Example 3.20: Revised versions of `puts` and `putc`:

```
puts:           ; display a string terminated by $
                ; ax contains address of string

       mov  dx, ax   ; copy address to dx for ms-dos
       mov  ah, 9h
       int  21h      ; call ms-dos to output string
       ret

putc:                ; display character in al
       mov  dl, al   ; copy al to dl for ms-dos
       mov  ah, 2h
       int  21h
       ret
```

Example 3.21: To illustrate the use of the new definitions of `putc` and `puts`, we rewrite the Program 3.19, which converts an uppercase letter to its lowercase equivalent:

```
; char2.asm: character conversion: uppercase to
lowercase

        .model small
        .stack 100h

CR      equ      13d
LF      equ      10d

        .data

msg1        db   'Enter an uppercase letter: $'
result db   CR, LF, 'The lowercase equivalent is: $'

        .code
; main program
start:
        mov  ax, @data
        mov  ds, ax

        mov  ax, offset msg1
        call puts
        call getc          ; read uppercase letter
        mov  bl, al        ; save character in bl

        add  bl, 32d       ; convert to lowercase

        mov  ax, offset result
        call puts          ; display result message
        mov  al, bl
        call putc          ; display lowercase letter

        mov ax, 4c00h
        int 21h            ; return to ms-dos
```

```
; user defined subprograms

puts:           ; display a string terminated by $
                ; ax contains address of string
       mov dx, ax
       mov ah, 9h
       int 21h ; call ms-dos to output string
       ret

putc:                   ; display character in al
       mov dl, al
       mov ah, 2h
       int 21h
       ret

getc:                   ; read character into al
       mov ah, 1h
       int 21h
       ret

       end start
```

3.4.1 Saving Registers

There is one disadvantage in using the above method of implementing `putc` and `puts`.

We now use two registers where formerly we only used one register to achieve the desired result. This reduces the number of registers available for storing other information.

Another important point also arises. In the `puts` subprogram, for example, the `dx` register is modified. I

f we were using this register in a program before the call to `puts` then the information stored in `dx` would be lost, unless we saved it before calling `puts`.

This can cause subtle but serious errors, in programs, that are difficult to detect. The following code fragment illustrates the problem:

```
mov  dx, 12        ; dx = 12

mov  ax, offset msg1   ; display message msg1

call puts          ; dx gets modified
add  dx, 2         ; dx will NOT contain 14
```

It may be much later in the execution of the program before this error manifests itself. Beginners make this type of error quite frequently in assembly language programs.

When a program behaves strangely, it is usually a good debugging technique to check for this type of situation, i.e. check that subprograms do not modify registers which you are using for other purposes.

This is a general problem with all subprograms that change the values of registers. All of our subprograms carrying out I/O change the value of the `ah` register. Thus, if we are using the `ah` register before calling a subprogram, we must save it before the subprogram is called.

In addition, the MS-DOS subprogram invoked using the `int` instruction may also change a register's value. For example, subprogram number 2h (used by `getc`) does this. It modifies the `al` register to return the value entered at the keyboard. The MS-DOS subprogram may also change other register values and you must be careful to check for this when using such subprograms.

There is a straightforward solution to this problem. We can and should write our subprograms so that before modifying any registers they first save the values of those registers. Then, before returning from a subprogram, we restore the registers to their original values.

(In the case of `getc`, however, we would not save the value of the `al` register because we want `getc` to read a value into that register.)

The **stack** is typically used to save and restore the values of registers used in subprograms.

The stack is an area of memory (RAM) where we can temporarily store items. We say that we "push the item onto the stack" to save it.

To get the item back from the stack, we "pop the item from the stack".

The 8086 provides **push** and **pop** instructions for storing and retrieving items from the stack. See Chapter 2 for details.

Example 3.22: We now rewrite the `getc`, `putc` and `puts` subprograms to save the values of registers and restore them appropriately. The following versions of `getc`, `putc` and `puts` are therefore safer in the sense that registers do not get changed without the programmer realising it.

```
puts:           ; display a string terminated by $
                ; dx contains address of string
       push ax  ; save ax
       push bx  ; save bx
       push cx  ; save cx
       push dx  ; save dx

       mov  dx, ax
       mov  ah, 9h
       int  21h ; call ms-dos to output string

       pop  dx  ; restore dx
       pop  cx  ; restore cx
       pop  bx  ; restore bx
       pop  ax  ; restore ax
       ret

putc:              ; display character in al
       push ax  ; save ax
       push bx  ; save bx
       push cx  ; save cx
       push dx  ; save dx

       mov  dl, al
       mov  ah, 2h
       int  21h

       pop  dx  ; restore dx
       pop  cx  ; restore cx
       pop  bx  ; restore bx
       pop  ax  ; restore ax
       ret
```

```
getc:                ; read character into al
      push bx   ; save bx
      push cx   ; save cx
      push dx   ; save dx

      mov ah, 1h
      int 21h

      pop  dx   ; restore dx
      pop  cx   ; restore cx
      pop  bx   ; restore bx

      ret
```

Note that we pop values from the **stack in the reverse order to the way we pushed them on**, due to the ***last-in-first-out (LIFO)*** nature of stack operations.

From now on, when we refer to `getc`, `putc` and `puts` in these notes, the definitions above are those intended.

Note: It is **vital**, when using the stack in subprograms, **to pop off all items pushed on the stack in the subprogram before returning from the subprogram**.

Failure to do so leaves an item on the stack which will be used by the `ret` instruction as the return address. This will cause your program to behave weirdly to say the least! If you are lucky, it will crash! Otherwise, it may continue to execute from any point in the program, producing baffling results.

The point is worth repeating: *when using the stack in a subprogram, be sure to remove all items pushed on, before returning from the subprogram.*

3.5 Control Flow: Jump Instructions

3.5.1 Unconditional Jump Instruction

The 8086 unconditional `jmp` instruction causes control flow (i.e. which instruction is next executed) to transfer to the point indicated by the label given in the `jmp` instruction.

Example 3.23: This example illustrates the use of the `jmp` instruction to implement an **endless** loop – not something you would noramlly wish to do!

```
again:
        call getc          ; read a character
        call putc          ; display character
        jmp again          ; jump to again
```

This is an example of a **backward** jump as control is transferred to an earlier place in the program.

The code fragment causes the instructions between the label `again` and the `jmp` instruction to be repeated endlessly.

You may place a label at any point in your program and the label can be on the same line as an instruction e.g.

```
again: call getc     ; read a character
```

The above program will execute forever unless you halt it with an interrupt, e.g. by pressing ctrl/c or by switching off the machine.

Example 3.24: The following code fragment illustrates a forward jump, as control is transferred to a later place in the program:

```
            call getc          ; read a character
            call putc          ; display the character
            jmp finish         ; jump to label finish

            <do other things>; Never gets done !!!

  finish:
            mov  ax, 4c00h
            int  21h
```

In this case the code between `jmp` instruction and the label `finish` will not be executed because the `jmp` causes control to skip over it.

3.5.2 Conditional Jump Instructions

The 8086 provides a number of conditional jump instructions (e.g. `je`, `jne`, `ja`). These instructions will only cause a transfer of control if some condition is satisfied.

For example, when an arithmetic operation such as add or subtract is carried out, the CPU sets or clears a flag (Z-flag) in the **status** register to record if the result of the operation was zero, or another flag if the result was negative and so on.

If the Z-flag has value 1, it means that the result of the last instruction which affected the Z-flag was 0.

If the Z-flag has value 0, it means that the result of the last instruction which affected the Z-flag was not 0.

By testing these flags, either individually or a combination of them, the conditional jump instructions can handle the various conditions (==, !=, <, >, <=, >=) that arise when comparing values. In addition, there are conditional jump instructions to test for conditions such as the occurrence of **overflow** or a change of sign.

The conditional jump instructions are sometimes called **jump-on-condition** instructions. They test the values of the flags in the status register.

(The value of the `cx` register is used by some of them). One conditional jump is the **`jz`** instruction which jumps to another location in a program just like the `jmp` instruction except that it only causes a jump if the Z-flag is set to 1**, i.e.** if the result of the last instruction was 0. (The `jz` instruction may be understood as standing for 'jump on condition zero' or 'jump on zero').

Example 3.25: Using the `jz` instruction.

```
            mov  ax, 2          ; ax = 2
            sub  ax, bx         ; ax = 2 - bx
            jz   nextl          ; jump if (ax-bx) == 0
            inc  ax             ; ax = ax + 1
nextl:
            inc  bx

The above is equivalent to:

            ax = 2;
            if ( ax != bx )
            {
                ax = ax + 1 ;
            }

            bx = bx + 1 ;
```

In this example, the Z-flag will be set (to 1) only if `bx` contains 2. If it does, then the `jz` instruction will cause the jump to take place as the test of the Z-flag yields true.

We are effectively comparing `ax` with `bx` and jumping if they are equal.

The 8086 provides the `cmp` instruction for such comparisons It works exactly like the `sub` instruction except that the operands are not affected, i.e. it subtracts the source operand from the destination but **discards** the result leaving the destination operand unchanged. However, it does modify the status register. All the flags that would be set or reset by `sub` are set or reset by `cmp`. So, if you wish to compare two values it makes more sense to use the `cmp` instruction.

Example 3.26: The above example could be rewritten using `cmp`:

```
      mov  ax, 2    ; ax becomes 2
      cmp  ax, bx   ; set flags according to (ax - bx)
      jz   equals   ; jump if (ax == bx)
      inc  ax       ; executed only if bx != ax
equals:
      inc  bx
```

Note: The `cmp` compares the **destination** operand with the **source** operand. The order is obviously important because for example, an instruction such as `jng dest, source` will cause a branch only if `dest` <= `source` .

Most jump-on-condition instructions have more than one name, for example the `jz` (jump on zero) instruction is also called **`je`** (jump on equal). Thus the above code could be written:

```
cmp  ax, bx
je   equals        ; jump if ax == bx
```

This name for the instruction makes the code more readable in a situation where we are testing two values for equality.

The jump-on-condition instructions may be used to jump forwards (as in the above example) or backwards and thus implement loops.

There are **sixteen** jump-on-condition instructions which test whether flags or combinations of flags are set or cleared.

However, rather than concentrating on the flag settings, it is easier to understand them in terms of comparing numbers (signed and unsigned separately) as equal, not equal, less than, greater than, greater than or equal and less than or equal.

Table 3.1 lists the jump-on-condition instructions. It gives the alternative names for those that have them.

Name(s)	Jump if	Flags tested
je / jz	equal/zero	zf = 1
jne / jnz	not equal/not zero	zf = 0
Operating with Unsigned Numbers		
ja / jnbe	above/not below or equal	(cf or zf) = 0
jae / jnb	above or equal/not below	cf = 0
jb / jnae / jc	below/not above or equal/carry	cf = 1
jbe / jna	below or equal/not above	(cf or zf) = 1
Operating with Signed Numbers		
jg / jnle	greater/not less than nor equal	zf=0 and sf = of
jge / jnl	greater or equal/not less	sf = of
jl / jnge	less /not greater nor equal	sf <> of
jle / jng	less or equal/not greater	(zf=1) or (sf!=of)
jo	overflow	of = 1
jno	not overflow	of = 0
jp / jpe	parity/parity even	pf = 1
jnp / jpo	no parity/odd parity	pf = 0
js	sign	sf = 1
jns	no sign	sf = 0

Table 3.1: Conditional jump instructions

Notes:
• `cf`, `of`, `zf`, `pf` and `sf` are the carry, overflow, zero, parity and sign flags of the flags (status) register.

• (`cf` or `zf`) = 1 means that the jump is made if either `cf` or `zf` is set to 1.

· In the above instructions, the letter `a` can be taken to mean above and the letter `b` to mean below. Instructions using these letters (e.g. `ja`, `jb` etc.) operate on **unsigned** numbers.

The letter `g` can be taken to mean greater than and the letter `l` to mean less than. Instructions using these letters (e.g. `jg`, `jl` etc.) operate on **signed** numbers.

It is the **programmer's responsibility** to use the correct instruction depending on whether signed or unsigned numbers are being manipulated.

There are also four jump instructions involving the `cx` register: `jcxz`, `loop`, `loope`, `loopne`. For example, the `jcxz` instruction causes a jump if the contents of the `cx` register is zero.

3.5.3 Implementation of `if-then` control structure

The general form of the **`if-then`** control structure in C is:

```
if (condition )
{
     /* action statements */
}
<rest of program>
```

It consists of a condition to be evaluated and an action to be performed if the condition yields true.

Example 3.27:

C version

```
if ( i == 10 )
{
     i = i + 5 ;
     j = j + 5 ;
}
/* Rest of program */
```

There are two ways of writing this in assembly language. One method tests if the condition `(i == 10)` is true. It branches to carry out the action if the condition is true. If the condition is false, there is a second unconditional branch to the next part of the program. This is written as:

8086 version 1:

```
          cmp  i, 10
          je   label1   ; if i == 10 goto label1
          jmp  rest     ; otherwise goto rest
label1:       add i, 5
          add  j, 5
rest:                           ; rest of program
```

The second method tests if the condition `(i != 10)` is true, branching to the code to carry out the rest of the program if this is the case. If this is not the case, then the action instructions are executed:

8086 version 2:

```
          cmp  i, 10
          jne  rest     ; if i != 10 goto rest
          add  i, 5     ; otherwise do action part
          add  j, 5
rest:                   ; rest of program
```

The second method only requires a single branch instruction and is to be preferred.

So, in general, to implement an if-then construct in assembly language, **we test the inverse of the condition that would be used in the high level language form of the construct**, as in version 2 above.

3.5.4 Implementation of `if-then-else` control structure

The general form of this control structure in C is:

```
if ( condition )
{
    /* action1 statements     */
}
else
{
    /* action2 statements     */
}
```

Example 3.28: Write a code fragment to read a character entered by the user and compare it to the character 'A'. Display an appropriate message if the user enters an 'A'. This code fragment is the basis of a guessing game program.

C version:

```
          printf("Guessing game: Enter a letter (A
to Z): ");
          c = getchar() ;
          if ( c == 'A' )
              printf("You guessed correctly !! ");
          else
              printf("Sorry incorrect guess ") ;
```

8086 version:

```
                mov ax, offset prompt ; prompt user
                call puts
                call getc             ; read character

                cmp al, 'A'       ; compare it to 'A'
                jne is_not_an_a       ; jump if not 'A'
                mov ax, offset yes_msg        ;  if
action
                call puts    ; display correct guess

                jmp end_else      ; skip else action
is_not_an_A:                      ; else action
                mov ax, offset no_msg
                call puts         ; display wrong guess

end_else:
```

If the value read is the letter `'A'`, then the `jne` will not be executed, `yes_msg` will be displayed and control transferred to `end_else`. If the value entered is not an `'A'`, then the `jne` is executed and control is transferred to `is_not_an_A`.

Example 3.29: The complete program to play a guessing game based on the above code fragment is:

```
; guess.asm: Guessing game program.
;User is asked to guess which letter the program
'knows'
; Author:  Joe Carthy
; Date:    March 1994

        .model small
        .stack 100h

CR      equ      13d
LF      equ      10d

        .data
prompt db   "Guessing game: Enter a letter (A to Z):
$"
yes_msg    db   CR, LF,"You guessed correctly !! $"
no_msg db   CR, LF,"Sorry incorrect guess $"

        .code
start:
        mov  ax, @data
        mov  ds, ax
        mov  ax, offset prompt
        call puts               ; prompt for input

        call getc               ; read character
        cmp  al, 'A'
        jne  is_not_an_a           ; if (al != 'A') skip
action
        mov  ax, offset yes_msg    ; if action
        call puts               ; display correct guess
        jmp  end_else1          ; skip else action
is_not_an_A:                    ; else action
        mov  ax, offset no_msg
        call puts               ; display wrong guess

end_else1:
```

```
finish:    mov ax, 4c00h
       int 21h

; User defined subprograms
; < puts getc defined here>

       end start
```

Note: In this program we use the label `end_else1` to indicate the end of the `if-then-else` construct.

It is important, if you use this construct a number of times in a program, to use different labels each time the construct is used. So a label such as `end_else2` could be used for the second occurrence of the construct although it is to be preferred if a more meaningful label such as `is_not_an_A` is used.

Example 3.30: Modify Program 3.19, which converts an uppercase letter to lowercase, to test that an uppercase letter was actually entered. To test if a letter is uppercase, we need to test if its ASCII code is in the range 65 to 90 (‘A’ to ‘Z’). In C such a test could be written as:

```
if ( c >= ‘A‘ && c <= ‘Z‘ )
  /* it is uppercase letter */
```

The opposite condition, i.e. to test if the letter is not uppercase may be written as:

```
if ( c < ‘A‘ || c > ‘Z‘ )
  /* it is not uppercase letter */
```

The variable `c` contains the ASCII code of the character entered. It is being compared with the ASCII codes of ‘A’ and ‘Z’.

The notation `&&` used in the first condition, reads as `AND`, in other words if the value of c is greater than or equal to ‘A’ `AND` it is less than or equal to ‘Z’, then `c` contains an uppercase letter.

The notation `||` used in the second condition reads as `OR`, in other words, if the value of `c` is less than ‘A’ `OR` if it is greater than ‘Z’, it cannot be an uppercase letter. We use the first condition in the 8086 program below.

C version:

```
main()     /* char.c: convert letter to lowercase */
{
  char c;

  printf("\nEnter an uppercase letter: ");
  c = getchar();
  if ( c >= 'A' && c <= 'Z' )
  {
      c = c + ( 'a' - 'A' ) ;   /* convert to
lowercase */
      printf("\nThe lowercase equivalent is: %c ",
c);
  }
  else
      printf("\nNot an uppercase letter %c ", c );
}
```

8086 version:

```
; char3.asm: character conversion: uppercase to
lowercase
; Author:  Joe Carthy
; Date:    March 1994

  .model small
  .stack 100h

CR     equ      13d
LF     equ      10d

  .data

msg1       db  CR, LF,'Enter an uppercase letter:
$'
result db  CR, LF,'The lowercase equivalent is: $'
bad_msg    db  CR, LF,'Not an uppercase letter: $'

       .code                ; main program
```

```
start:
      mov  ax, @data
      mov  ds, ax

      mov  ax, offset msg1
      call puts
      call getc          ; read uppercase letter
      mov  bl, al        ; save character in bl
      cmp  bl, 'A'
      jl   invalid       ; if bl < 'A' goto invalid
      cmp  bl, 'Z'       ; if bl > 'Z' goto invalid
      jg   invalid
                         ; otherwise its valid
      add  bl, 32d       ; convert to lowercase

      mov  ax, offset result
      call puts          ; display result message
      mov  al, bl
      call putc          ; display lowercase letter
      jmp finish

invalid:
      mov  ax, offset bad_msg    ; not uppercase
      call puts              ; display bad_msg
      mov  al, bl
      call putc              ; display character
entered

finish:
      mov ax, 4c00h
      int 21h            ; return to ms-dos

; subprograms getc, putc and puts should be defined
here

      end start
```

This program produces as output, assuming the digit 8 is entered:

```
Enter an uppercase letter: 8
Not an uppercase letter:   8
```

It produces as output, assuming the letter Y is entered:

```
Enter an uppercase letter:   Y
The lowercase equivalent is: y
```

Exercises

3.13 Write a program to read a digit and display an error message if a non-digit character is entered.

3.14 In the code fragments below where will execution continue from when `<jump-on-condition>` is replaced by (a) `je lab1` ; (b) `jg lab1`; (c) `jle lab1`; (d) `jz lab1`

```
(i)   mov ax, 10h
      cmp ax, 9h
      <jump-on-condition>
      ; rest of program
      ..........
      ..........
lab1:

(ii)  mov cx, 0h
      cmp cx, 0d
      <jump-on-condition>
      ; rest of program
      ..........
      ..........
lab1:
```

3.15 Write programs to test that a character read from the keyboard and transfer control to label `ok_here`, if the character is:

(i) a valid lowercase letter (‘a’ <= character <= ‘z’)

(ii) either an uppercase or lowercase letter (‘A’ <= character <= ‘Z’ OR ‘a’ <= character <= ‘z’)

(iii) is not a lowercase letter, i.e. character < ‘a’ or character > ‘z’.

The programs should display appropriate messages to prompt for input and indicate whether the character satisfied the relevant test.

3.5.5 Loops

We have already seen how loops could be implemented using the `jmp` instruction to jump backwards in a program. However, we noted that since `jmp` is an unconditional jump, it gives rise to infinite loops. The solution is to use jump-on-condition instructions. For example, a `while` loop to display the '*' character 60 times may be implemented as in Example 3.31.

Example 3.31: Display a line of 60 stars.

C version:

```
count = 1 ;
while ( count <= 60 )
{
    putchar('*') ;
    count = count + 1 ;
}
```

8086 version:

```
            mov cx, 1d          ; cx = 1
            mov al, '*'         ; al = '*'

disp_char:
            cmp cx, 60d
            jnle end_disp       ; if cx > 60 goto end_disp
                call putc       ; display '*'
                inc cx          ; cx = cx + 1
            jmp disp_char           ; repeat loop
test

end_disp:
```

The instruction jnle (jump if not less than or equals) may also be written as jg (jump if greater than). We use a similar technique to that used in the implementation of an if-then construct in that we test the inverse of the condition used in the C code fragment(count <= 60). This allows us write clearer code in assembly language.

Example 3.32: Write a code fragment to display the characters from ‘a’ to ‘z’ on the screen using the knowledge that the ASCII codes form a **collating sequence**. This means that the code for ‘b’ is one greater than the code for ‘a’ and the code for ‘c’ is one greater than that for ‘b’ and so on.

C version:

```
c = ‘a‘ ;    /* c = 97 (ASCII for ‘a‘)
while ( c <= ‘z‘ )
{
    putchar( c );
    c = c + 1 ;
}
```

8086 version:

```
            mov  al, ‘a’
startloop:
            cmp  al, ‘z’
            jnle endloop      ; while al <= ‘z’
                 call putc     ; display character
                 inc  al       ; al = al + 1
            jmp  startloop        ; repeat test
endloop:
```

This program produces as output

```
abcdefghijklmnopqrstuvwxyz
```

In the last two examples, we specified how many times the loop action was to be carried out (such a loop is called a **deterministic** loop).

We frequently encounter cases when we do not know how many times the loop will be executed. For example, at each iteration we may ask the user if the loop action is to be repeated and the loop continues to execute or is terminated on the basis of the user's response.

Example 3.33: Program 3.19 reads an uppercase letter, converts it to lowercase and displays the lowercase equivalent. We now modify it, so that the user may repeat this process as often as desired. The user is asked to enter `'y'` to carry out the operation, after each iteration.

C version:

```
main()
{
   char c, reply;

   reply = 'y';

   while ( reply == 'y' )
   {
        printf("\nEnter an uppercase letter: ");
        c = getchar();
        c = c + ( 'a' - 'A' ) ;       /* convert to lowercase
*/
        printf("\nThe lowercase equivalent is: %c ", c);
        printf("\nEnter y to continue: ");
        reply = getchar();
   }
}
```

8086 version:

```
; char4.asm: character conversion: upper to lowercase
  .model small
  .stack 100h
CR     equ      13d
LF     equ      10d
  .data
reply      db   'y'
msg0       db   CR, LF, 'Enter y to continue: $'
msg1       db   CR, LF, 'Enter an uppercase letter: $'
result db   CR, LF, 'The lowercase equivalent is: $'
  .code
;                    main program
start:
       mov  ax, @data
       mov  ds, ax
readloop:
       cmp  reply, 'y'         ; while (reply == 'y')
       jne  finish             ; do loop body

       mov  ax, offset msg1
       call puts           ; prompt for letter
       call getc           ; read character
       mov  bl, al         ; save character in bl
       add  bl, 32d        ; convert to lowercase

       mov  ax, offset result
       call puts           ; display result message
       mov  al, bl
       call putc           ; display lowercase letter

       mov  ax, offset msg0
       call puts           ; prompt to continue
       call getc           ; read reply
       mov  reply, al          ; save character in reply
       jmp  readloop       ; repeat loop test

finish:
       mov ax, 4c00h
       int 21h             ; return to ms-dos
; user defined subprograms should be defined here
       end start
```

Executing this program produces as output, assuming the user enters the characters C, y, X and n:

```
Enter an uppercase letter: C
The lowercase equivalent is: c
Enter y to continue: y
Enter an uppercase letter: X
The lowercase equivalent is: x
Enter y to continue: n
```

Exercises

3.16 Modify the program in Example 3.33 to test that the letter entered is a valid uppercase letter. If it isn't an uppercase letter a suitable error message should be displayed and the program should continue executing for as long as the user wishes.

3.17 Modify the guessing game program (Program 3.29) to allow the user three guesses, terminating if any guess is correct.

3.18 Modify the guessing game program to allow users guess as many or as few times as they wish, terminating if any guess is correct.

3.19 Modify the guessing game program to loop until a correct guess is made.

3.5.6 Counting Loops

Counting loops, where we know in advance how many times to repeat the loop body, occur frequently in programming and as a result most high-level languages have a special construct called a **for-loop** to implement them.

In Program 3.31, to display the '*' character 60 times, we counted upwards from 1 to 60, testing each time around the loop to see if we have reached 60. In assembly language programming, it is common to count downwards, e.g. from 60 to 0.

Because this type of situation occurs frequently in programming, it can be implemented by using the **`loop`** instruction.

The `loop` instruction combines testing of `cx` with zero and the decrementing of `cx` in a single instruction, i.e. the loop instruction decrements `cx` by 1 and tests if `cx` equals zero.

It causes a jump if `cx` does not equal 0. It can only be used in conjunction with the `cx` register (known as the **count** register), i.e. the `cx` register is initialised with the number of times the loop is to be repeated. Program 3.31 can be rewritten to use the loop instruction as follows:

Example 3.36: Using `loop` instruction.

```
        mov  al, '*'        ; al = '*'
        mov  cx, 60d        ; cx = 60    ; loop count

disp_char:
```

```
      call putc          ; display '*'
      loop disp_char     ; cx = cx - 1, if (cx != 0)
goto disp_char
```

Here, `cx` is initialised to 60, the number of iterations required. The instruction `loop disp_char` first decrements `cx` and then tests if `cx` is not equal to 0, branching to `disp_char` only if `cx` does not equal 0.

General format for using `loop` instruction:

```
  mov cx, count     ; count = # of times to repeat
loop
start_loop:         ; use any label name

        <loop body>       ; while cx > 0
                    ; repeat loop body
instructions
      loop start_loop
```

To use the `loop` instruction, simply store the number of iterations required in the `cx` register and construct a loop body as outlined above. The last instruction of the loop body is the `loop` instruction.

Note 1: The loop body will always be executed **at least once**, since the `loop` instruction tests the value of `cx` after executing the loop body.

Note 2: What happens if `cx` is initialised to 0? The `loop` instruction decrements `cx` before testing the condition (`cx != 0`).

Thus we continue around the loop, with `cx` becoming more negative. We will repeat the loop body `65,536` times.

Why ?

The reason is because we keep subtracting `1` from `cx` until we reach `0`. Eventually, by making `cx` more negative, the largest negative number that `cx` can contain is reached. Since `cx` is 16-bit register, we know from Appendix 2, that this number is `-32768d`, which is the 16-bit number `1000 0000 0000 0000`.

Subtracting `1` from this yields the 16-bit number `0111 1111 1111 1111` or 32767d.

We can subtract `1` from this number `32767` times before reaching 0, which terminates the `loop` instruction. Thus the total number of iterations is `32768 + 32767 + 1` which equals `65,535 + 1` (the extra `1` is because `cx` started at `0` and was decremented to `-1` before the test).