Introduction to 8086 Programming

Learning any imperative programming language involves mastering a number of common concepts:

Variables: declaration/definition
Assignment: assigning values to variables
Input/Output: Displaying messages
Displaying variable values
Control flow: if-then
Loops
Subprograms: Definition and Usage

Programming in assembly language involves mastering the same concepts and a few other issues.

Variables
For the moment we will skip details of variable declaration and simply use the 8086 registers as the variables in our programs. Registers have predefined names and do not need to be declared.

We have seen that the 8086 has 14 registers. Initially, we will use four of them – the so called the general purpose registers:

`ax, bx, cx, dx`

These four 16-bit registers can also be treated as eight 8-bit registers: **`ah, al, bh, bl, ch, cl, dh, dl`**

Assignment

In Java, assignment takes the form:

```
x = 42 ;
y = 24;
z = x + y;
```

In assembly language we carry out the same operaion but we use an instruction to denote the assignment operator ("=" in Java).

```
mov    x, 42
mov    y, 24
add    z, x
add    z, y
```

The **mov** instruction carries out assignment in 8086 assembly language.

It which allows us place a number in a register or in a memory location (a variable) i.e. it assigns a value to a register or variable.

Example: **Store the ASCII code for the letter A in register bx.**

A has ASCII code 65D (01000001B, 41H)

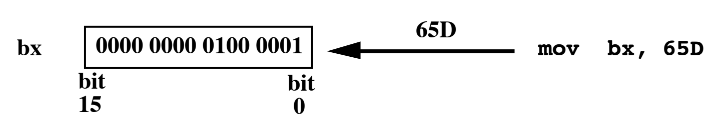
The following **mov** instruction carries out the task:

mov bx, 65d

We could also write it as:

mov bx, 41h
or **mov bx, 01000001b**
or **mov bx, ‘A’**

All of the above are equivalent. They each carry out exactly the same task, namely the binary number representing the ASCII code of A is copied into the bx register.



The effect of executing `mov cx, 65D` is to overwrite the `bx` register with 65D as a 16-bit number

The value is copied into the right-hand side (low-order byte) of the register. The left-hand side will contain all 0’s.

Thus we could also have written it as:

mov bl, 65d
mov bl, ‘A’

Since register bl represents the low-order byte of register bx.

Note: The 8086 Assembler converts a **character constant** i.e. a character in single quotes (e.g. ‘A’) to its ASCII code automatically. This is a very useful feature and it means that you can specify many characters without having to look up their ASCII code. You simply enclose the character in single quotes. You will have to use the

ASCII code for control characters such as carriage return and line feed.

Notation

mov is one of the many 8086 instructions that we will be using. Most assembly language books use uppercase letters to refer to an instruction e.g. MOV.

However, the assembler will also recognise the instruction if it is written in lowercase or in mixed case e.g. Mov. (In fact, the assembler converts all instructions to uppercase).

It is **my personal** preference to use lower case when writing programs. You may write your programs using which ever notation you find convenient, but you should be consistent and stick to one particular style.

More about mov

The **mov** instruction also allows you to copy the contents of one register into another register.

Example:

```
mov     bx, 2
mov     cx, bx
```

The first instruction loads the value 2 into bx where it is stored as a binary number. [a number such as 2 is called an **integer** constant]

The Mov instruction takes two **operands,** representing the ***destination*** where data is to be placed and the ***source*** of that data.

General Form of Mov Instruction

mov ***destination*****,** ***source***

where *destination* must be either a register or memory location and *source* may be a constant, another register or a memory location.

In 8086 assembly language, the source and destination **cannot both** be memory locations in the same instruction.

Note: The comma is essential. It is used to separate the two operands.

A missing comma is a common syntax error.

We will look at manipulating data in memory at a later stage.

More Examples

The following instructions result in registers ax, bx, and cx all having the value 4:

```
mov bx, 4       ; copy number 4 into register bx
mov ax, bx ; copy contents of bx into register ax
mov cx, ax ; copy contents of ax into register cx
```

Comments

Anything that follows semi-colon (**;**) is ignored by the assembler. It is called a **comment**. Comments are used to make your programs readable. You use them to explain what you are doing in English.

It is recommended that you use comments frequently in your programs, not only so that others can understand them, but also for yourself, when you look back at programs you have previously written.

Every programming language has a facility for defining comments.

More 8086 Instructions

add, inc, dec and sub instructions

The 8086 provides a variety of arithmetic instructions. For the moment, we only consider a few of them. To carry out arithmetic such as addition or substraction, you use the appropriate instruction.

In assembly language you can only carry out a single arithmetic operation at a time. This means that if you wish to evaluate an expression such as :

$$z = x + y + w - v$$

You will have to use 3 assembly language instructions – one for each arithmetic operation.

These instruction combine assignment with the arithmetic operation.

Example:

```
mov     ax, 5     ;     load 5 into ax

add     ax, 3     ;     add 3 to the contents of ax,
                  ;     ax now contains 8

inc     ax        ;     add 1 to ax
                  ;     ax now contains 9

dec     ax        ;     subtract 1 from ax
                  ;     ax now contains 8

sub     ax, 6     ;     subtract 4 from ax
                  ;     ax now contains 2
```

The **add** instruction adds the source operand to the destination operand, leaving the result in the destination operand.

The destination operand is always the first operand in 8086 assembly language.

(In M68000 assembly language, it is the other way round i.e. the source operand is always the first operand e.g. `move #10, x`)

The **inc** instruction takes one operand and adds 1 to it. It is provided because of the frequency of adding 1 to an operand in programming.

The **dec** instruction like **inc** takes one operand and subtracts 1 from it. This is also a frequent operation in programming.

The **sub** instruction subtracts the source operand from the destination operand leaving the result in the destination operand.

Some microprocessors do not provide instructions for multiplication or division (e.g. the M6800). With such microprocessors, multiplication and division have to be programmed using repeated additions and subtractions and shift operations (which will be discussed later).

The 8086 provides **mul** and **div** (and others) for multiplication and division.

Ambiguity

Suppose you wish to load the hexadecimal value A (decimal 10) written as **ah** in the register bl.

You might be tempted to write:

```
mov        bl, ah
```

But we have already seen that there is a register called **ah** (the high-order byte of ax) and so the above does not do what we intend. Instead it copies the contents of register ah into bl. In order to avoid ambiguity when writing hexadecimal numbers that begin with a letter we prefix them with 0. Thus we write:

```
mov        bl, 0ah  ; copy hex number ah into bx
```

It is common practice to write decimal numbers with the letter D appended so as to distinguish them from hexadecimal.

The 8086 assembler take all numbers to be decimal numbers unless there is a **B** (binary), **H** (hex) or **O** (octal) appended to them.

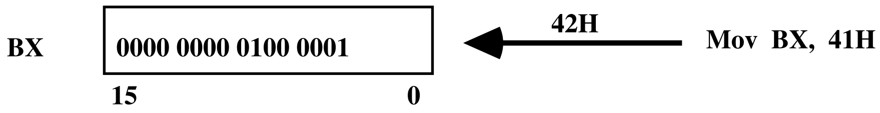
Note:

When data is moved to a register, all 16 bits (or 8 bits) are given a value. The assembler will automatically fill in 0's on the left-hand side.

Example:

mov bx, 42h ; copy 42 hex into bx

42H is 100 0001 in binary. This padded out with nine 0-bits on the left-hand side to fill all 16-bits of the register.



The effect of executing MOV BX, 41H is to overwrite the BX register with 41H in binary.

Exercises:

1) Write instructions to:
Load character ? into register bx
Load space character into register cx
Load 26 (decimal) into register cx
Copy contents of ax to bx and dx

2) What errors are present in the following :

```
mov     ax   3d
mov     23,  ax
mov     cx, ch
move    ax, 1h
add     2, cx
add     3, 6
inc     ax, 2
```

3) Write instructions to evaluate the arithmetic expression $5 + (6-2)$ leaving the result in ax using (a) 1 register, (b) 2 registers, (c) 3 registers

4) Write instructions to evaluate the expressions:

$$a = b + c - d$$

$$z = x + y + w - v + u$$

5) Rewrite the expression in 4) above but using the registers ah, al, bh, bl and so on to represent the variables: a, b, c, z, x, y, w, u, and v.

Input and Output (I/O) in 8086 Assembly Language

Each microprocessor provides instructions for I/O with the devices that are attached to it, e.g. the keyboard and screen.

The 8086 provides the instructions `in` for input and `out` for output. These instructions are quite complicated to use, so we usually use the operating system to do I/O for us instead.

The operating system provides a range of I/O subprograms, in much the same way as there is an extensive library of subprograms available to the C programmer. In C, to perform an I/O operation, we call a subprogram using its name to indicate its operations, e.g. `putchar()`, `printf()`, `getchar()`. In addition we may pass a parameter to the subprogram, for example the character to be displayed by `putchar()` is passed as a parameter e.g. `putchar(c)`.

In assembly language we must have a mechanism to call the operating system to carry out I/O.

In addition we must be able to tell the operating system what kind of I/O operation we wish to carry out, e.g. to read a character from the keyboard, to display a character or string on the screen or to do disk I/O.

Finally, we must have a means of passing parameters to the operating subprogram.

In 8086 assembly language, we do not call operating system subprograms by name, instead, we use a software interrupt mechanism

An interrupt signals the processor to suspend its current activity (i.e. running your program) and to pass control to an interrupt service program (i.e. part of the operating system).

A software interrupt is one generated by a program (as opposed to one generated by hardware).

The 8086 `int` instruction generates a software interrupt.

It uses a single operand which is a number indicating which MS-DOS subprogram is to be invoked.

For I/O and some other operations, the number used is **`21h`**.

Thus, the instruction `int 21h` transfers control to the operating system, to a subprogram that handles I/O operations.

This subprogram handles a variety of I/O operations by calling appropriate subprograms.

This means that you must also specify which I/O operation (e.g. read a character, display a character) you wish to carry out. This is done by placing a specific number in a register. The `ah` register is used to pass this information.

For example, the subprogram to display a character is subprogram number **2h**.

This number must be stored in the `ah` register. We are now in a position to describe character output.

When the I/O operation is finished, the interrupt service program terminates and our program will be resumed at the instruction following `int`.

3.3.1 Character Output

The task here is to display a single character on the screen. There are three elements involved in carrying out this operation using the `int` instruction:

1. We specify the character to be displayed. This is done by storing the character's ASCII code in a specific 8086 register. In this case we use the **`dl`** register, i.e. we use `dl` to pass a parameter to the output subprogram.

2. We specify which of MS-DOS's I/O subprograms we wish to use. The subprogram to display a character is subprogram number **`2h`**. This number is stored in the `ah` register.

3. We request MS-DOS to carry out the I/O operation using the `int` instruction. This means that we **interrupt** our program and transfer control to the MS-DOS subprogram that we have specified using the `ah` register.

Example 1: Write a code fragment to display the character `'a'` on the screen:

C version:

```
      putchar( 'a' ) ;
```

8086 version:

```
  mov dl, 'a'  ; dl = 'a'
  mov ah, 2h   ; character output subprogram
  int 21h      ; call ms-dos output character
```

As you can see, this simple task is quite complicated in assembly language.

3.3.2 Character Input

The task here is to read a single character from the keyboard. There are also three elements involved in performing character input:

1. As for character output, we specify which of MS-DOS's I/O subprograms we wish to use, i.e. the character input from the keyboard subprogram. This is MS-DOS subprogram number **1h**. This number must be stored in the `ah` register.

2. We call MS-DOS to carry out the I/O operation using the `int` instruction as for character output.

3. The MS-DOS subprogram uses the `al` register to store the character it reads from the keyboard.

Example 2: Write a code fragment to read a character from the keyboard:

C version:

```
c = getchar() ;
```

8086 Version:

```
mov ah, 1h  ; keyboard input subprogram
int 21h     ; character input
            ; character is stored in al
mov c, al   ; copy character from al to c
```

The following example combines the two previous ones, by reading a character from the keyboard and displaying it.

Example 3: Reading and displaying a character:

C version:

```
c = getchar() ;
putchar( c ) ;
```

8086 version:

```
mov ah, 1h  ; keyboard input subprogram
int 21h     ; read character into al

mov dl, al  ; copy character to dl

mov ah, 2h  ; character output subprogram
int 21h     ; display character in dl
```

A Complete Program

We are now in a position to write a complete 8086 program. You must use an **editor** to enter the program into a file. The process of using the editor (**editing**) is a basic form of word processing. This skill has no relevance to programming.

We use Microsoft's `MASM` and `LINK` programs for assembling and linking 8086 assembly language programs. `MASM` program files should have names with the **extension** (3 characters after period) `asm`. We will call our first program `prog1.asm`, it displays the letter `'a'` on the screen. (You may use any name you wish. It is a good idea to choose a meaningful file name). Having entered and saved the program using an editor, you must then use the `MASM` and `LINK` commands to translate it to machine code so that it may be executed as follows:

```
C> masm prog1
```

If you have syntax errors, you will get error messages at this point. You then have to edit your program, correct them and repeat the above command, otherwise proceed to the `link` command, pressing Return in response to prompts for file names from `masm` or `link`.

```
C> link prog1
```

To execute the program, simply enter the program name and press the Return key:

```
C> prog1
a
C>
```

Example 4: A complete program to display the letter ‘a‘ on the screen:

```
; prog1.asm: displays the character ‘a’ on the screen
; Author:  Joe Carthy
; Date:    March 1994

       .model small
       .stack 100h

       .code
start:
       mov  dl, ‘a’  ; store ascii code of ‘a’ in dl

       mov  ah, 2h   ; ms-dos character output function
       int  21h      ; displays character in dl register

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

The first three lines of the program are comments to give the name of the file containing the program, explain its purpose, give the name of the author and the date the program was written.

The first two directives, `.model` and `.stack` are concerned with how your program will be stored in memory and how large a stack it requires. The third directive, `.code`, indicates where the program instructions (i.e. the program code) begin.

For the moment, suffice it to say that you need to start all assembly languages programs in a particular format (not necessarily that given above.

Your program must also finish in a particular format, the `end` directive indicates where your program finishes.

In the middle comes the code that you write yourself.

You must also specify where your program starts, i.e. which is the **first** instruction to be executed. This is the purpose of the label, `start`.

(Note: We could use any label, e.g. `begin` in place of `start`).

This same label is also used by the `end` directive. When a program has finished, we return to the operating system.

Like carrying out an I/O operation, this is also accomplished by using the `int` instruction. This time MS-DOS subprogram number `4c00h` is used.

It is the subprogram to terminate a program and return to MS-DOS. Hence, the instructions:

```
mov ax, 4c00h ; Code for return to MS-DOS
int 21H       ; Terminates program
```

terminate a program and return you to MS-DOS.

Time-saving Tip

Since your programs will start and finish using the same format, you can save yourself time entering this code for each program. You create a template program called for example, `template.asm`, which contains the standard code to start and finish your assembly language programs. Then, when you wish to write a new program, you copy this template program to a new file, say for example, `prog2.asm`, as follows (e.g. using the MS-DOS copy command):

```
       C> copy template.asm io2.asm
```

You then edit `prog2.asm` and enter your code in the appropriate place.

Example 3.9: The following template could be used for our first programs:

```
; <filename goes here>.asm:
; Author:
; Date:

       .model small
       .stack 100h

       .code
start:

;      < your code goes here >

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

To write a new program, you enter your code in the appropriate place as indicated above.

Example 3.10: Write a program to read a character from the keyboard and display it on the screen:

```
; prog2.asm: read a character and display it
; Author: Joe Carthy
; Date:   March 1994

      .model small
      .stack 100h

      .code
start:

  mov ah, 1h  ; keyboard input subprogram
  int 21h     ; read character into al

  mov dl, al

  mov ah, 2h  ; display subprogram
  int 21h     ; display character in dl

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

  end start
```

Assuming you enter the letter 'B' at the keyboard when you execute the program, the output will appear as follows:

```
  C> prog2
  BB
```

Rewrite the above program to use a prompt:

```
          C>prog4
          ?B B
```

```
; prog4.asm: prompt user with ?
; Author: Joe Carthy
; Date:   March 1994
      .model small
      .stack 100h
      .code
start:
; display ?
      mov dl, ‘?’     ; copy ? to dl
      mov ah, 2h      ; display subprogram
      int 21h         ; call ms-dos to display ?

; read character from keyboard
      mov ah, 1h      ; keyboard input subprogram
      int 21h         ; read character into al

; save character entered while we display a space
      mov bl, al      ; copy character to bl

; display space character
      mov dl, ‘ ’     ; copy space to dl
      mov ah, 2h      ; display subprogram
      int 21h         ; call ms-dos to display space

; display character read from keyboard
      mov dl, bl      ; copy character entered to dl
      mov ah, 2h      ; display subprogram
      int 21h         ; display character in dl

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

Note: In this example we must save the character entered (we save it in `bl`) so that we can use `ax` for the display subprogram number.

Example 3.12: Modify the previous program so that the character entered, is displayed on the following line giving the effect:

```
C> io4
? x
x
```

In this version, we need to output the Carriage Return and Line-feed characters.

Carriage Return, (ASCII 13D) is the control character to bring the cursor to the start of a line.

Line-feed (ASCII 10D) is the control character that brings the cursor down to the next line on the screen.

(We use the abbreviations CR and LF to refer to Return and Line-feed in comments.)

In C and Java programs we use the newline character '\n' to generate a new line which in effect causes a Carriage Return and Linefeed to be transmitted to your screen.

```
; io4.asm:     prompt user with ?,
; read character and display the CR, LF characters
; followed by the character entered.
; Author:  Joe Carthy
; Date:    March 1994

      .model small
      .stack 100h
      .code
start:
; display ?
      mov dl, ‘?’       ; copy ? to dl
      mov ah, 2h        ; display subprogram
      int 21h           ; display ?

; read character from keyboard
      mov ah, 1h        ; keyboard input subprogram
      int 21h           ; read character into al

; save character while we display a Return and Line-
feed
      mov bl, al        ; save character in bl

;display Return
      mov dl, 13d       ; dl = CR
      mov ah, 2h        ; display subprogram
      int 21h           ; display CR

;display Line-feed
      mov dl, 10d       ; dl = LF
      mov ah, 2h        ; display subprogram
      int 21h           ; display LF

; display character read from keyboard
      mov dl, bl        ; copy character to dl
      mov ah, 2h        ; display subprogram
      int 21h           ; display character in dl

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

Note: Indentation and documentation, as mentioned before, are the responsibility of the programmer. Program 3.13 below is a completely valid way of entering the program presented earlier in Example 3.12:

Example 3.13 without indentation and comments.

```
.model small
.stack 100h
.code
start:
 mov dl,‘?’
 mov ah,2h
 int 21h
 mov ah,1h
 int 21h
 mov bl,al
 mov dl,13d
 mov ah,2h
 int 21h
 mov dl,10d
 mov ah,2h
 int 21h
 mov dl,bl
 mov ah,2h
 int 21h
 mov ax,4c00h
 int 21h
end  start
```

Which program is easier to read and understand ?

String Output

A string is a list of characters treated as a unit. In programming languages we denote a string constant by using quotation marks, e.g. “Enter first number”.

In 8086 assembly language, single or double quotes may be used.

Defining String Variables

The following 3 definitions are equivalent ways of defining a string `“abc“`:

```
version1   db   “abc”             ; string constant
version2   db   ‘a’, ‘b’, ‘c’     ; character constants
version3   db   97, 98, 99        ; ASCII codes
```

The first version uses the method of high level languages and simply encloses the string in quotes. This is the preferred method.

The second version defines a string by specifying a list of the character constants that make up the string.

The third version defines a string by specifying a list of the ASCII codes that make up the string

We may also combine the above methods to define a string as in the following example:

```
  message  db   “Hello world”, 13, 10, ‘$’
```

The string `message` contains 'Hello world' followed by Return (ASCII 13), Line-feed (ASCII 10) and the `'$'` character.

This method is very useful if we wish to include control characters (such as Return) in a string.

We terminate the string with the `'$'` character because there is an MS-DOS subprogram (number `9h`) for displaying strings which expects the string to be terminated by the `'$'` character.

It is important to understand that **`db`** is not an assembly language instruction. It is called a **directive**.

A directive tells the assembler to do something, when translating your program to machine code.

The **`db`** directive tells the assembler to store one or more bytes in a **named memory location**. From the above examples, the named locations are `version1`, `version2`, `version3` and `message`.

These are in effect **string variables**.

In order to display a string we must know where the string begins and ends.

The beginning of string is given by obtaining its address using the `offset` operator.

The end of a string may be found by either knowing in advance the length of the string or by storing a special character at the end of the string which acts as a **sentinel**.

We have already used MS-DOS subprograms for character I/O (number **1h** to read a single character from the keyboard and number **2h** to display a character on the screen.)

String Output

MS-DOS provides subprogram number **9h** to display strings which are terminated by the **`'$'`** character. In order to use it we must:

1 Ensure the string is terminated with the `'$'` character.

2 Specify the string to be displayed by storing its address in the **`dx`** register.

3 Specify the string output subprogram by storing `9h` in `ah.`

4 Use `int 21h` to call MS-DOS to execute subprogram `9h.`

The following code illustrates how the string `'Hello world'`, followed by the Return and Line-feed characters, can be displayed.

Example 3.14: Write a program to display the message `‘Hello world’` followed by Return and Line-feed :

```
; io8.asm: Display the message ‘Hello World’
; Author: Joe Carthy
; Date:   March 1994

      .model small
      .stack 100h

          .data
message       db      ‘Hello World‘, 13, 10, ‘$‘

          .code
start:
          mov ax, @data
          mov ds, ax

; copy address of message to dx
          mov dx, offset message

          mov     ah, 9h       ; string output
          int     21h          ; display string

          mov ax, 4c00h
          int     21h

          end start
```

In this example, we use the **`.data`** directive. This directive is required when memory variables are used in a program.

The instructions

```
mov  ax, @data
mov  ds, ax
```

are concerned with accessing memory variables and must be used with programs that use memory variables. See textbook for further information.

The ***`offset`*** operator allows us to access the address of a variable. In this case, we use it to access the address of `message` and we store this address in the `dx` register.

Subprogram `9h` can access the string `message` (or any string), once it has been passed the starting address of the string.

Exercises

- Write a program to display ‘MS-DOS’ using (a) character output and (b) using string output.

- Write a program to display the message ‘Ding! Ding! Ding!’ and output ASCII code 7 three times. (ASCII code 7 is the Bel character. It causes your machine to beep!).

- Write a program to beep, display ‘?’ as a prompt, read a character and display it on a new line.

Control Flow Instructions: Subprograms

A subprogram allows us to give a **name** to a group of instructions and to use that name when we wish to execute those instructions, instead of having to write the instructions again.

For example, the instructions to display a character could be given the name `putc` (or whatever you choose). Then to display a character you can use the name `putc` which will cause the appropriate instructions to be executed.

This is referred to as **calling** the subprogram. In 8086 assembly language, the instruction `call` is used to invoke a subprogram, so for example, a `putc` subprogram would be called as follows:

```
          call putc     ; Display character in dl
```

The process of giving a group of instructions a name is referred to as **defining** a subprogram. **This is only done once**.

Definition of `putc, getc and puts` subprograms.

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

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

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

The `ret` instruction terminates the subprogram and arranges for execution to resume at the instruction following the `call` instruction.

We usually refer to that part of a program where execution begins as the **main program**.

In practice, programs consist of a main program and a number of subprograms. It is important to note that subprograms make our programs easier to read, write and maintain even if we only use them once in a program.

Note: Subprograms are defined **after** the code to terminate the program, but **before** the `end` directive.

If we placed the subprograms earlier in the code, they would be executed without being called (execution would *fall through into* them). This should **not** be allowed to happen.

The following program illustrates the use of the above subprograms.

```
C> sub
Enter a character: x
You entered: x
```

```
; subs.asm: Prompt user to enter a character
; and display the character entered
; Author: Joe Carthy
; Date:   March 1994

      .model small
      .stack 100h

          .data
prompt        db      'Enter a character: $'
msgout        db      'You entered: $'


          .code
start:
          mov ax, @data
          mov ds, ax

; copy address of message to dx
          mov dx, offset prompt
          call puts       ; display prompt

          call getc   ; read character into al
          mov bl, al  ; save character in bl

;display next message
          mov dx, offset msgout
          call puts       ; display msgout

; display character read from keyboard
      mov dl, bl      ; copy character to dl
      call putc

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

Defining Constants: *Macros*

The **`equ`** directive is used to define constants.

For example if we wish to use the names `CR` and `LF`, to represent the ASCII codes of Carriage Return and Line-feed, we can use this directive to do so.

```
CR   equ 13d
LF   equ 10d
MAX  equ 1000d
MIN  equ 0
```

The assembler, replaces all occurrences of `CR` with the number 13 before the program is translated to machine code. It carries out similar replacements for the other constants.

Essentially, the `equ` directive provides a text substitution facility. One piece of text (`CR`) is replaced by another piece of text (13), in your program. Such a facility is often call a **macro** facility.

We use constants to make our programs easier to read and understand.

Example 3.18: The following program, displays the message ‘Hello World’, and uses the equ directive.

```
; io9.asm: Display the message ‘Hello World’
; Author: Joe Carthy
; Date:   March 1994

      .model small
      .stack 100h
      .data

CR        equ 13d
LF        equ 10d

message   db      ‘Hello World’, CR, LF, ‘$’

      .code
start:
      mov ax, @data
      mov ds, ax

      mov dx, offset message
      call    puts            ; display message

      mov ax, 4c00h
      int     21h

; User defined subprograms

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

      end     start
```

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     equu     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).

Declaring Variables in Assembly Language

As in Java, variables must be declared before they can be used. Unlike Java, we do not specify a variable **type** in the declaration in assembly language. Instead we declare the name and **size** of the variable, i.e. the number of bytes the variable will occupy. We may also specify an initial value.

A **directive** (i.e. a command to the assembler) is used to define variables. In 8086 assembly language, the directive **`db`** defines a byte sized variable; **`dw`** defines a word sized variable (16 bits) and **`dd`** defines a double word (long word, 32 bits) variable.

A Java variable of type **`int`** may be implemented using a size of 16 or 32 bits, i.e. **`dw`** or **`dd`** is used. A Java variable of type `char`, which is used to store a single character, is implemented using the **`db`** directive.

Example:

```
reply           db   'y'
prompt    db    'Enter your favourite colour: ', 0
colour    db    80 dup(?)
i         db    20
k         db    ?
num       dw    4000
large           dd   50000
```

`reply` is defined as a character variable, which is initialised to `'y'`.

`prompt` is defined as a string, terminated by the Null character.

The definition of the variable `colour` demonstrates how to declare an **array** of characters of size 80, which contains undefined values.

The purpose of `dup` is to tell the assembler to duplicate or repeat the data definition directive a specific number of times, in this case `80 dup` specifies that 80 bytes of storage are to be set aside since `dup` is used with the `db` directive.

The `(?)` with the `dup` means that storage allocated by the directive is unitialised or undefined.

`i` and `k` are byte sized variables, where `i` is initialised to `20` and `k` is left undefined.

`num` is a 16-bit variable, initialised to `4000` and the variable `large` is a 32-bit variable, initialised to `15000`.

Indirect Addressing

Given that we have defined a string variable **message** as

```
message db ‘Hello’,0,
```

an important feature is that the characters are **stored in consecutive memory locations**.

If the `‘H’` is in location `1024`, then `‘e’` will be in location `1025`, `‘l’` will be in location `1026` and so on. A technique known as **indirect addressing** may be used to access the elements of the array.

Indirect addressing allows us store the address of a location in a register and use this register to access the value stored at that location.

This means that we can store the address of the string in a register and access the first character of the string via the register. If we increment the register contents by 1, we can access the next character of the string. By continuing to increment the register, we can access each character of the string, in turn, processing it as we see fit.

Figure 1 illustrates how indirect addressing operates, using register `bx` to contain the address of a string `"Hello"` in memory. Here, register `bx` has the value `1024` which is the address of the first character in the string.

Another way of phrasing this is to say that `bx` **points** to the first character in the string.

In 8086 assembly language we denote this by enclosing `bx` in square brackets: **`[bx]`**, which reads as the value **pointed to** by `bx`, i.e. **the contents of the location whose address is stored in the `bx` register**.

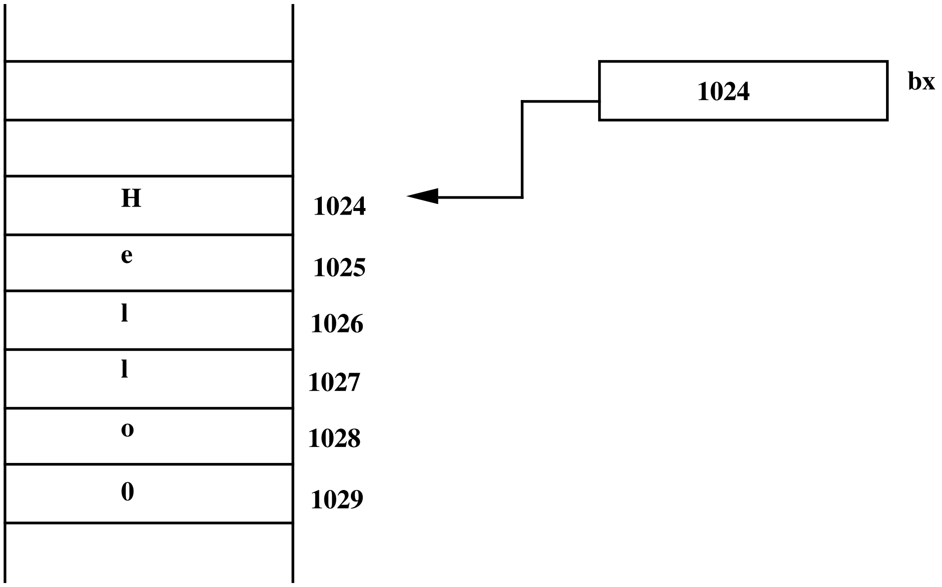


Figure 1: Using the `bx` register for indirect addressing

The first character of the string can be accessed as follows:

```
          cmp  byte ptr [bx], 0    ; is this end of string?
```

This instruction compares the character (indicated by `byte ptr`) pointed to by `bx` with 0.

How do we store the address of the string in `bx` in the first place? The special operator **`offset`** allows us specify the address of a memory variable. For example, the instruction:

```
     mov  bx, offset message
```

will store the **address** of the variable `message` in `bx`. We can then use `bx` to access the variable `message`.

Example: The following code fragment illustrates the use of indirect addressing. It is a loop to count the number of characters in a string terminated by the Null character (ASCII 0). It uses the `cx` register to store the number of characters in the string.

```
message        db        'Hello', 0
.......
........
     mov  cx,   0                 ;   cx   stores   number   of
characters
     mov  bx, offset message  ; store address of message in bx

begin:
          cmp  byte ptr [bx], 0    ; is this end of string?
          je   fin                 ; if yes goto Finished
               inc  cx             ; cx = cx + 1
               inc  bx             ; bx points to next character
          jmp begin
                              ; cx now contains the # of
                              ; characters in message
fin:
```

The label `begin` indicates the beginning of the loop to count the characters. After executing the `mov` instruction, register `bx` contains the address of the first character in the string. We compare this value with `0` and if the value is not `0`, we count it by incrementing `cx`. We then increment `bx` so that it now points to the next character in the string. We repeat this process until we reach the `0` character which terminates the string.

Note: If you omit the 0 character when defining the string, the above program will fail. Why? The reason is that the loop continues to execute, until `bx` points to a memory location containing 0. If 0 has been omitted from the definition of message, then we do not know when, if ever, the loop will terminate. This is the same as an array subscript out of bounds error in a high level language.

The form of indirect addressing described here is called **register indirect addressing** because a register is used store the indirect address.

String I/O

In programming languages such as C, strings are terminated by the `'\0'` character. We adopt the same convention. This method of terminating a string has an advantage over that used for the `puts` subprogram defined earlier, where the '$' character is used to terminate a string. The use of the value 0 to terminate a string means that a string may contain the '$' character which can then be displayed, since '$' cannot be displayed by `puts`.

We use this indirect addressing in the implementation of two subprograms for reading and displaying strings: `get_str` and `put_str`

Example 3.42: Read colour entered by the user and display a suitable message, using `get_str` and `put_str`.

```
; colour.asm: Prompt user to enter a colour and display a message
; Author: Joe Carthy
; Date:   March 1994

          .model small

          .stack 256

CR        equ       13d
LF        equ       10d

; string definitions: note 0 terminator
          .data
msg1      db   ‘Enter your favourite colour: ‘, 0
msg2      db   CR, LF,‘Yuk ! I hate ‘, 0
colour    db   80 dup (0)

          .code
start:
          mov  ax, @data
          mov  ds, ax

          mov  ax, offset msg1
          call put_str              ; display prompt

          mov  ax, offset colour
          call get_str              ; read colour

          mov  ax, offset msg2
          call put_str              ; display msg2

          mov  ax, offset colour
          call put_str              ;  display  colour  entered  by
user

          mov  ax, 4c00h
          int       21h             ; finished, back to dos
```

```
put_str:                 ; display string terminated by 0
                         ; whose address is in ax

          push ax   ; save registers
          push bx
          push cx
          push dx

          mov  bx, ax               ; store address in bx
          mov  al, byte ptr [bx]    ; al = first char in string

put_loop: cmp  al, 0                     ; al == 0 ?
          je   put_fin              ; while al != 0
          call putc                 ; display character
          inc  bx                   ; bx = bx + 1
          mov  al, byte ptr [bx]    ; al = next char in string
          jmp  put_loop             ; repeat loop test
put_fin:
          pop  dx                   ; restore registers
          pop  cx
          pop  bx
          pop  ax
          ret
```

```
get_str:            ; read string terminated by CR into array
                    ; whose address is in ax

               push ax   ; save registers
               push bx
               push cx
               push dx

               mov  bx, ax

               call getc           ; read first character
               mov  byte ptr [bx], al   ; In C: str[i] = al

get_loop:      cmp  al, 13         ; al == CR ?
               je   get_fin        ;while al != CR

               inc  bx             ; bx = bx + 1
               call getc           ; read next character
               mov  byte ptr [bx], al   ; In C: str[i] = al
               jmp get_loop        ; repeat loop test

get_fin:       mov  byte ptr [bx], 0    ; terminate string with 0

               pop  dx             ; restore registers
               pop  cx
               pop  bx
               pop  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

          end start
```

This program produces as output:

```
Enter your favourite colour: yellow
Yuk ! I hate yellow
```

Reading and Displaying Numbers

See Chapter 3 of textbook for implementation details

We use `getn` and `putn` to read and display numbers:

`getn:` reads a number from the keyboard and returns it in the ax register

`putn:` displays the number in the ax register

Example: Write a program to read two numbers, add them and display the result.

```
; calc.asm: Read and sum two numbers. Display result.
; Author: Joe Carthy
; Date:   March 1994

          .model small

          .stack 256

CR        equ       13d
LF        equ       10d

          .data

prompt1   db   ‘Enter first number: ’, 0
prompt2   db   CR, LF,‘Enter second number:’,0
result    db   CR, LF ‘The sum is’, 0

num1      dw   ?
num2      dw   ?
```

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

          mov  ax, offset prompt1
          call put_str                  ; display prompt1
          call getn                ; read first number
          mov  num1, ax

          mov  ax, offset prompt2
          call put_str                  ; display prompt2
          call getn                ; read second number
          mov  num2, ax

          mov  ax, offset result
          call put_str                  ; display result message

          mov  ax, num1            ; ax = num1
          add  ax, num2            ; ax = ax + num2
          call putn                ; display sum

          mov  ax, 4c00h
          int  21h                 ; finished, back to dos


     <definitions of getn, putn, put_str, get_str, getc, putc go
here>

          end start
```

Running the above program produces:

```
Enter first number: 8
Enter second number: 6
The sum is 14
```

More about the Stack

A **stack** is an **area of memory** which is used for storing data on a temporary basis. In a typical computer system the memory is logically partitioned into separate areas. Your program code is stored in one such area, your variables may be in another such area and another area is used for the stack. Figure 2 is a crude illustration of how memory might be allocated to a user program running.

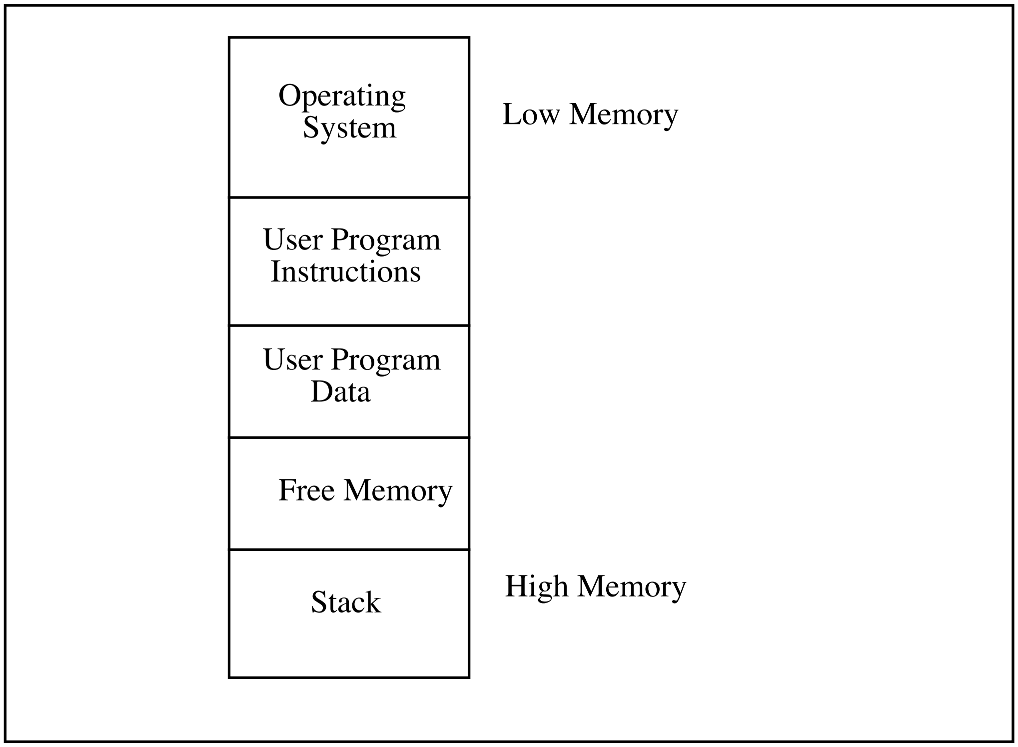


Figure 2: Memory allocation: User programs share memory with the Operating System software

The area of memory with addresses near 0 is called low memory, while high memory refers to the area of memory near the highest address. The area of memory used for your program code is fixed, i.e. once the code is loaded into memory it does not grow or shrink.

The stack on the other hand may require varying amounts of memory. The amount actually required depends on how the program uses the stack. Thus the size of the stack varies during program execution. We can store information on the stack and retrieve it later.

One of the most common uses of the stack is in the implementation of the subprogram facility. This usage is transparent to the programmer, i.e. the programmer does not have to explicitly access the stack. The instructions to call a subprogram and to return from a subprogram automatically access the stack. They do this in order to return to the correct place in your program when the subprogram is finished.

The point in your program where control returns after a subprogram finishes is called the **return address.** The return address of a subprogram is placed on the stack by the `call` instruction. When the subprogram finishes, the `ret` instruction retrieves the return address from the stack and transfers control to that location. The stack may also be used to pass information to subprograms and to return information from subprograms, i.e. as a mechanism for handling high level language parameters.

Conceptually a stack as its name implies is a **stack of data elements**. The size of the elements depends on the processor and for example, may be 1 byte, 2 bytes or 4 bytes. We will ignore this for the moment. We can illustrate a stack as in Figure 3:

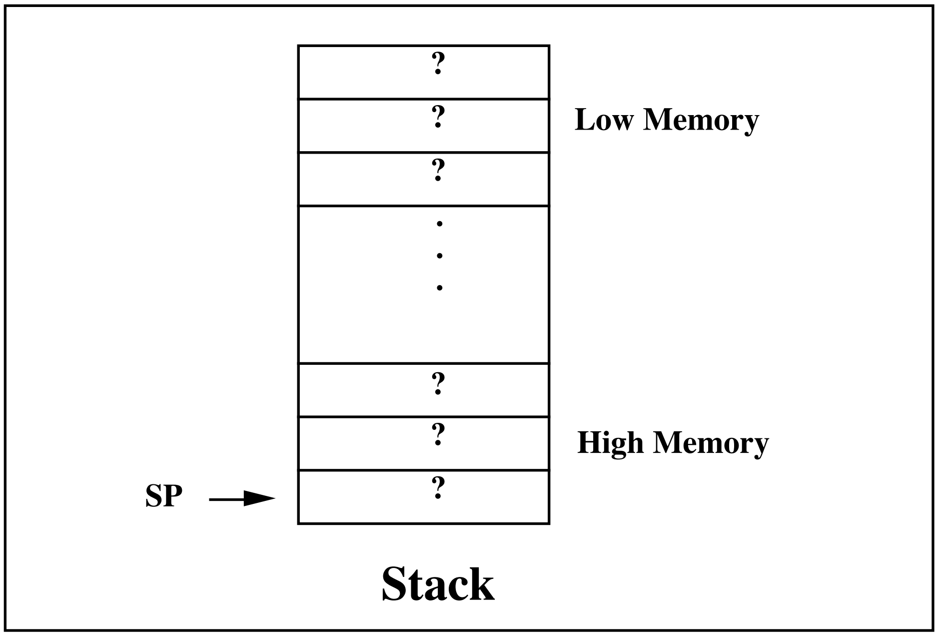


Figure 3: Simple model of the stack

To use the stack, the processor must keep track of where items are stored on it. It does this by using the **stack pointer** (**`sp`**) register.

This is one of the processor's special registers. It points to the **top** of the stack, i.e. its contains the address of the stack memory element containing the value last placed on the stack. When we place an element on the stack, the stack pointer contains the address of that element on the stack. If we place a number of elements on the stack, the stack pointer will always point to the last element we placed on the stack. When retrieving elements from the stack we retrieve them in reverse order. This will become clearer when we write some stack manipulation programs.

There are two basic stack operations which are used to manipulate the stack usually called **push** and **pop**. The 8086 `push` instruction places (pushes) a value on the stack. The stack pointer is left pointing at the value pushed on the stack. For example, if `ax` contains the number `123`, then the following instruction:

```
push ax
```

will cause the value of `ax` to be stored on the stack. In this case the number `123` is stored on the stack and `sp` **points** to the location on the stack where `123` is stored.

The 8086 `pop` instruction is used to retrieve a value previously placed on the stack. The stack pointer is left pointing at the next element on the stack. Thus `pop` conceptually removes the value from the stack. Having stored a value on the stack as above, we can retrieve it by:

```
pop   ax
```

which transfers the data from the top of the stack to `ax`, (or any register) in this case the number `123` is transferred. Information is stored on the stack starting from high memory locations. As we place data on the stack, the stack pointer points to successively lower memory locations. We say that the stack grows downwards. If we assume that the top of the stack is location `1000` (`sp` contains `1000`) then the operation of `push ax` is as follows.

Firstly, `sp` is decremented by the size of the element (2 bytes for the 8086) to be pushed on the stack. Then the value of `ax` is copied to the location pointed to by `sp`, i.e. `998` in this case. If we then assign `bx` the value `212` and carry out a `push bx` operation, `sp` is again decremented by two, giving it the value `996` and `212` is stored at this location on the stack. We now have two values on the stack.

As mentioned earlier, if we now retrieve these values, we encounter the fundamental feature of any stack mechanism. Values are retrieved in **reverse order.** This means that the last item placed on the stack, is the first item to be retrieved. We call such a process a **Last-In-First-Out** process or a **LIFO** process.

So, if we now carry out a `pop ax` operation, `ax` gets as its value `212`, i.e. the last value pushed on the stack.

If we now carry out a `pop bx` operation, `bx` gets as its value `123`, the second last value pushed on the stack.

Hence, the operation of `pop` is to copy a value from the top of the stack, as pointed to by `sp` and to increment `sp` by 2 so that it now points to the previous value on the stack.

We can push the value of any register or memory variable on the stack. We can retrieve a value from the stack and store it in any register or a memory variable.

The above example is illustrated in Figure 4 (steps (1) to (4) correspond to the states of the stack and stack pointer after each instruction).

Note: For the 8086, we can **only push 16-bit items** onto the stack e.g. any register.

The following are ILLEGAL: push al
pop bh

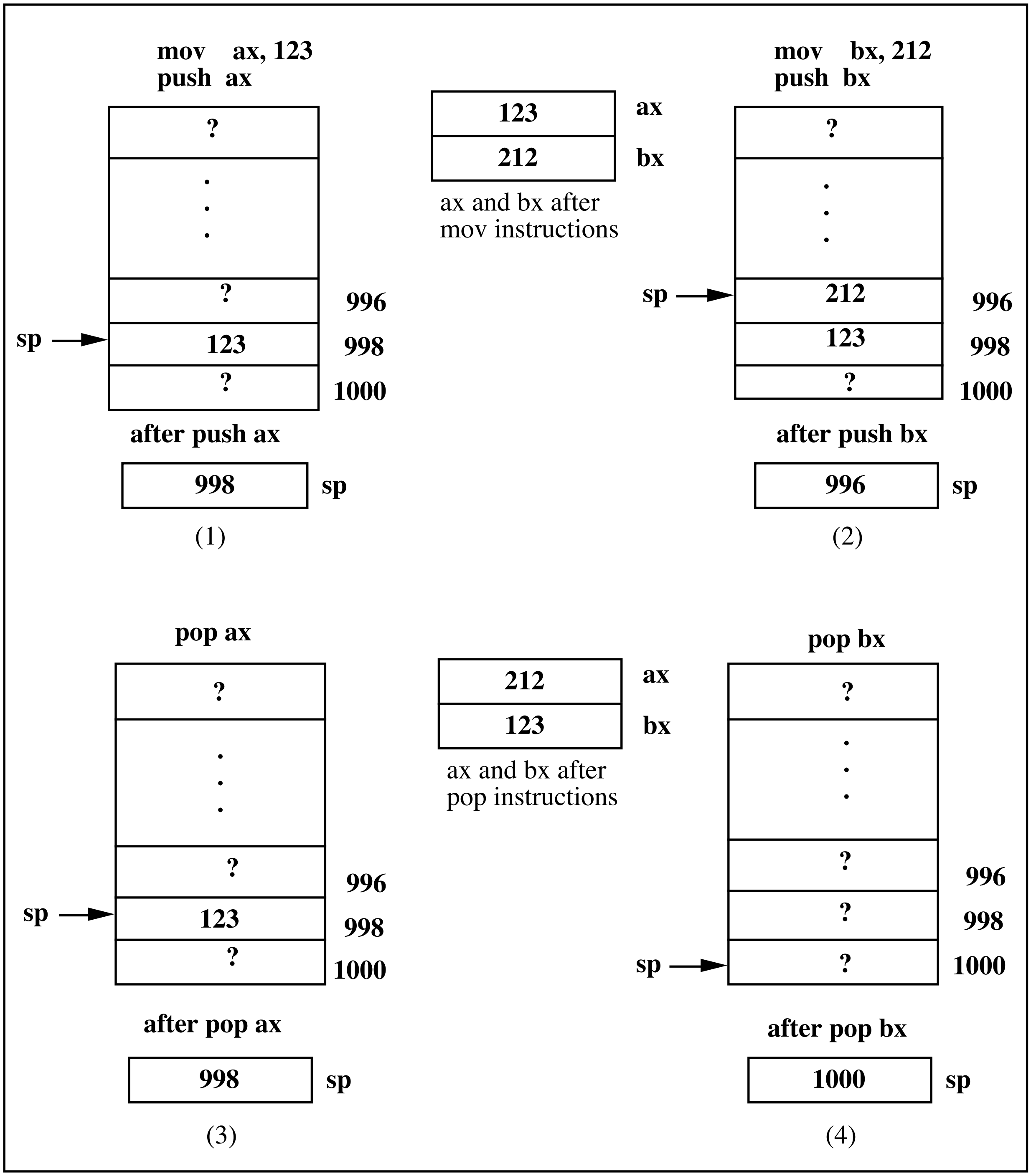


Figure 4: LIFO nature of push and pop

Example: Using the stack, swap the values of the `ax` and `bx` registers, so that `ax` now contains what `bx` contained and `bx` contains what `ax` contained. (This is not the most efficient way to exchange the contents of two variables). To carry out this operation, we need at least one temporary variable:

Version 1**:**

```
push ax            ; Store ax on stack
push bx            ; Store bx on stack
pop  ax            ; Copy last value on stack to ax
pop  bx            ; Copy first value to bx
```

The above solution stores both `ax` and `bx` on the stack and utilises the LIFO nature of the stack to retrieve the values in reverse order, thus swapping them in this example. We really only need to store one of the values on the stack, so the following is a more efficient solution.

Version 2:

```
push ax            ; Store ax on stack
mov  ax, bx        ; Copy bx to ax
pop  bx            ; Copy old ax from stack
```

When using the stack, the number of items pushed on should equal the number of items popped off.

This is vital if the stack is being used inside a subprogram. This is because, when a subprogram is called its return address is pushed on the stack.

If, inside the subprogram, you push something on the stack and do not remove it, the return instruction will retrieve the item you left on the stack instead of the return address. This means that your subprogram cannot return to where it was called from and it will most likely crash (unless you were very clever about what you left on the stack!).

Format of Assembly Language Instructions

The format of assembly language instructions is relatively standard. The **general format** of an instruction is (where square brackets **[]** indicate the **optional** fields) as follows:

```
[Label]   Operation   [Operands]    [; Comment]
```

The instruction may be treated as being composed of four **fields**. All four fields need **not** be present in every instruction, as we have seen from the examples already presented. Unless there is only a comment field, the **operation field** is **always** necessary. The label and the operand fields may or may not be required depending on the operation field.

Example: Examples of instructions with varying numbers of fields.

	Note
`L1:  cmp bx, cx      ; Compare bx with cx`	***all fields present***
`     add ax, 25`	***operation and 2 operands***
`     inc bx`	***operation and 1 operand***
`     ret`	***operation field only***
`; Comment: whatever you wish !!`	***comment field only***

Bit Manipulation

One of the features of assembly language programming is that you can access the individual bits of a byte (word or long word).

You can **set** bits (give them a value of `1`), **clear** them (give them a value of `0`), **complement** them (change `0` to `1` or `1` to `0`), and **test** if they have a particular value.

These operations are essential when writing subprograms to control devices such as printers, plotters and disk drives. Subprograms that control devices are often called **device drivers**. In such subprograms, it is often necessary to set particular bits in a register associated with the device, in order to operate the device. The instructions to operate on bits are called **logical** instructions.

Under normal circumstances programmers rarely need concern themselves with bit operations. In fact most high-level languages do not provide bit manipulation operations. (The C language is a notable exception). Another reason for manipulating bits is to make programs more efficient. By this we usually mean one of two things: the program is smaller in size and so requires less RAM or the program runs faster.

The Logical Instructions: `and, or, xor, not`

As stated above, the logical instructions allow us operate on the bits of an operand. The operand may be a byte (8 bits), a word (16 bits) a long word (32 bits). We will concentrate on byte sized operands, but the instructions operate on word operands in exactly the same fashion.

Clearing Bits: `and` instruction

A bit **`and`** operation compares two bits and sets the result to `0` if either of the bits is `0`.

e.g.

```
1 and 0 returns 0
0 and 1 returns 0
0 and 0 returns 0
1 and 1 returns 1
```

The `and` instruction carries out the `and` operation on all of the bits of the source operand with all of the bits of the destination operand, storing the result in the destination operand (like the arithmetic instructions such as `add` and `sub`).

The operation `0 and x` always results in `0` regardless of the value of `x` (`1` or `0`). This means that we can use the `and` instruction to clear a specified bit or collection of bits in an operand.

If we wish to clear, say bit 5, of an 8-bit operand, we `and` the operand with the value `1101 1111`, i.e. a value with bit `5` set to `0` and all other values set to `1`.

This results in bit `5` of the 8-bit operand being cleared, with the other bits remaining unchanged, since `1 and x` always yields `x`.

(Remember, when referring to a bit number, we count from bit **`0`** upwards.)

Example 4.1: To clear bit `5` of a byte we `and` the byte with `1101 1111`

```
mov  al, 62h        ; al =          0110 0010
and  al, 0dfh       ; and it with   1101 1111
                    ; al is 42h     0100 0010
```

[Note: You can use binary numbers directly in 8086 assembly language, e.g.

```
mov  al, 01100010b
and  al, 11011111b
```

but it is easier to write them using their hexadecimal equivalents.]

The value in the source operand, `0dfh`, in this example, is called a **bit mask**. It specifies the bits in the destination operand that are to be changed. Using the `and` instruction, any bit in the bit mask with value `0` will cause the corresponding bit in the destination operand to be cleared.

In the ASCII codes of the lowercase letters, bit `5` is always `1`. The corresponding ASCII codes of the uppercase letters are identical except that bit `5` is always `0`. Thus to convert a lowercase letter to uppercase we simply need to clear bit `5` (i.e. set bit `5` to `0`). This can be done using the `and` instruction and an appropriate bit mask, i.e. `0dfh`, as shown in the above example. The letter `'b'` has ASCII code `62h`. We could rewrite Example 4.1 above as:

Example B.1: Converting a lowercase letter to its uppercase equivalent:

```
mov  al, 'b'         ; al = 'b'(= 98d or 62h) 0110 0010
and  al, 0dfh        ; mask =                 1101 1111
                     ; al now = 'B'(= 66d or 42h) 0100 0010
```

The bit mask `1101 1111` when used with `and` will always set bit `5` to `0` leaving the remaining bits unchanged as illustrated below:

```
               xxxx xxxx ; destination bits
          and  1101 1111 ; and with mask bits
               xx0x xxxx ;  result  is  that  bit  5  is
cleared
```

If the destination operand contains a lowercase letter, the result will be the corresponding uppercase equivalent. In effect, we have subtracted `32` from the ASCII code of the lowercase letter which was the method we used in Chapter 3 for converting lowercase letters to their uppercase equivalents.

Setting Bits: `or` instruction

A bit `or` operation compares two bits and sets the result to `1` if either bit is set to `1`.

e.g.

```
1 or 0 returns 1
0 or 1 returns 1
1 or 1 returns 1
0 or 0 returns 0
```

The `or` instruction carries out an `or` operation with all of the bits of the source and destination operands and stores the result in the destination operand.

The `or` instruction can be used to set bits to `1` regardless of their current setting since `x or 1` returns `1` regardless of the value of `x` (`0` or `1`).

The bits set using the `or` instruction are said to be **masked in**.

Example: Take the conversion of an uppercase letter to lowercase, the opposite of Example B.1 discussed above. Here, we need to **set** bit `5` of the uppercase letter's ASCII code to `1` so that it becomes lowercase and leave all other bits unchanged. The required mask is `0010 0000` (`20h`). If we store `'A'` in `al` then it can be converted to `'a'` as follows:

```
mov  al, 'A'               ; al = 'A' =   0100 0001
or   al, 20h               ; or with      0010 0000
                           ; gives al = 'a'    0110 0001
```

In effect, we have added `32` to the uppercase ASCII code thus obtaining the lowercase ASCII code.

Before changing the case of a letter, it is important to verify that you have a letter in the variable you are working with.

Exercises

4.1 Specify the instructions and masks would you use to

a) set bits `2`, `3` and `4` of the `ax` register

b) clear bits `4` and `7` of the `bx` register

4.2 How would `al` be affected by the following instructions:

(a) `and al, 00fh`

(b) `and al, 0f0h`

(c) `or al, 00fh`

(d) `or al, 0f0h`

4.3 Write subprograms `todigit` and `tocharacter`, which convert a digit to its equivalent ASCII character code and vice versa.

4.1.3 The `xor` instruction

The `xor` operation compares two bits and sets the result to `1` if the bits are different.

e.g.

```
1 xor 0 returns 1
0 xor 1 returns 1
1 xor 1 returns 0
0 xor 0 returns 0
```

The `xor` instruction carries out the `xor` operation with its operands, storing the result in the destination operand.

The `xor` instruction can be used to **toggle** the value of specific bits (reverse them from their current settings). The bit mask to toggle particular bits should have `1`'s for any bit position you wish to toggle and `0`'s for bits which are to remain unchanged.

Example 4.7: Toggle bits `0`, `1` and `6` of the value in `al` (here `67h`):

```
mov  al, 67h   ; al =         0011 0111
xor  al, 08h   ; xor it with  0100 0011
               ; al is 34h         0111 0100
```

A common use of `xor` is to clear a register, i.e. set all bits to `0`, for example, we can clear register `cx` as follows

```
xor  cx, cx
```

This is because when the identical operands are `xor`ed, each bit cancels itself, producing `0`:

```
0 xor 0 produces 0
1 xor 1 produces 0
```

Thus `abcdefgh xor abcdefgh` produces `00000000` where `abcdefgh` represents some bit pattern. The more obvious way of clearing a register is to use a `mov` instruction as in:

```
mov  cx, 0
```

but this is slower to execute and occupies more memory than the `xor` instruction. This is because bit manipulation instructions, such as `xor`, can be implemented very efficiently in hardware. The `sub` instruction may also be used to clear a register:

```
sub  cx, cx
```

It is also smaller and faster than the `mov` version, but not as fast as the `xor` version. My own preference is to use the clearer version, i.e. the `mov` instruction. However, in practice, assembly language programs are used where efficiency is important and so clearing a register with `xor` is often used.

4.1.4 The `not` instruction

The `not` operation **complements** or **inverts** a bit, i.e.

```
not 1 returns 0
not 0 returns 1
```

The **`not`** instruction inverts **all** of the bits of its operand.

Example 4.8: Complementing the `al` register:

```
mov  al, 33h        ; al =        00110011
not  al             ; al =        11001100
```

Table 1 summarises the results of the logical operations. Such a table is called a **truth table**.

A	B	**not** A	A **and** B	A **or** B	A **xor** B
0	**0**	**1**	**0**	**0**	**0**
0	**1**	**1**	**0**	**1**	**1**
1	**0**	**0**	**0**	**1**	**1**
1	**1**	**0**	**1**	**1**	**0**

Table 4.1: Truth table for logical operators

Efficiency

As noted earlier, the `xor` instruction is often used to clear an operand because of its efficiency. For similar reasons of efficiency, the `or/and` instructions may be used to compare an operand to 0.

Example 4.9: Comparing an operand to 0 using logical instructions:

```
or   cx, cx         ; compares cx with 0
je   label
and  ax, ax         ; compares ax with 0
jg   label2
```

Doing `or/and` operations on identical operands, does not change the destination operand (`x or x` returns `x`; `x and x` returns `x`), but they do set flags in the status register. The `or/and` instructions above have the same effect as the `cmp` instructions used in Example 4.10, but they are faster and smaller instructions (each occupies 2 bytes) than the `cmp` instruction (which occupies 3 bytes).

Shifting and Rotating Bits

We sometimes wish to change the positions of all the bits in a byte, word or long word. The 8086 provides a complete set of instructions for shifting and rotating bits. Bits can be moved right (towards the 0 bit) or left towards the most significant bit. Values shifted off the end of an operand are lost (one may go into the **carry flag**).

Shift instructions move bits a specified number of places to the right or left.

Rotate instructions move bits a specified number of places to the right or left. For each bit rotated, the last bit in the direction of the rotate is moved into the first bit position at the other end of the operand.