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
```