
A introduction to x86 assembly

Written by Gavin Estey

email: gavin@senator.demon.co.uk

This document is version:

0.9 Beta 2/19/95

This will hopefully be the final version of this unless somebody finds some mistakes in it. I have spent over 620 minutes creating this document (not including writing or testing the code or the original text document) so I hope you will find it useful.

This version is still in testing which means that I cannot be sure that all the code will compile and work correctly. I have done my best to make sure that all the information contained in this document is correct. If any mistakes are found please could you notify me be email.

If someone who has a computer that doesn't have a VGA card/monitor could you please compile the code on P29 that checks what graphics the computer can handle and tell me if it returned DX as 0.

This document was written by Gavin Estey. No part of this can be reproduced and sold in any commercial product without my written permission (email me for more information). I am not responsible for any damage caused in anyway from this document.

If you enjoy this document then a donation would be gratefully received. No payment is necessary but none will turned down. If any mistakes are noticed then I would like to be notified. If you would like any other help or have suggestions for a later version of this document.

Firstly you will need a suitable assembler to compile your programs. All examples have been tested with two compilers: A86 and Turbo Assembler® ver 4. A86 is a very good shareware assembler capable of producing code up to 80286. This can be found on your local simtel mirror (try [ftp.demon.co.uk](ftp://ftp.demon.co.uk) or [ftp.cdrom.com](ftp://ftp.cdrom.com)) under simtel/msdos/asmutil called A86V322.ZIP.

Firstly I am going to talk about SEGMENTS and OFFSETS. This (for me anyway) is probably the hardest part of assembly to understand.

Segments and offsets:

The original designers of the 8088 decided that nobody will every need to use more than one megabyte of memory. So they built the chip so it couldn't access above that. The problem is to access a whole megabyte 20 bits are needed (one bit being either a one or a zero). Registers only have 16 bits and they didn't want to use two because that would be 32 bits and they thought that this would be too much for anyone. They decided to do the addressing with two registers but not 32 bits. Is this confusing? Blame the designers of the 8088.

OFFSET = SEGMENT * 16

SEGMENT = OFFSET / 16 (the lower 4 bits are lost)

SEGMENT * 16	0010010000010000 - ---	range (0 to 65535) * 16
OFFSET	-- - -0100100000100010	range (0 to 65535)
20 bit address	00101000100100100010	range 0 to 1048575 (1 MEG)
	===== DS =====	
	===== SI =====	

(note DS and SI overlap). This is how DS:SI is used to make a 20 bit address. The segment is in DS and the offset is in SI.

Segment registers are: CS, DS, ES, SS. On the 386+ there are also FS & GS

Offset registers are: BX, DI, SI, BP, SP, IP. In 386+ protected mode, ANY general register (not a segment register) can be used as an Offset register. (Except IP, which you can't access.)

Registers:

AX, BX, CX and DX are general purpose registers. On a 386+ they are 32 bits; EAX, EBX, ECX and EDX. The all can be split up into high and low parts, each being 8 bits.

EAX	32 bits
AX	16 bits
AH	8 bits
AL	8 bits

This means that you can use AH and AL to store different numbers and treat them as separate registers for some tasks.

BX (BH/BL): same as AX

CX (CH/CL): Same as AX (used for loops)

DX (DH/DL): Same as AX (used for multiplication/division)

DI and SI are index registers and can be used as offset registers.

SP: Stack Pointer. Does just that. Points to the current position in the stack. **Don't** alter unless you REALLY know what you are doing or want to crash your computer.

The Stack:

This is an area of memory that is like a stack of plates. The last one you put on is the first one that you take off (LOFO). If another piece of data is put on the stack it grows downwards.

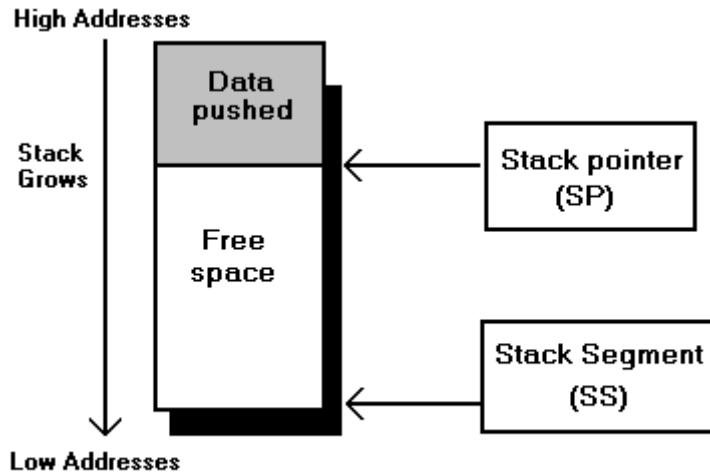


Diagram 1: This shows how the stack is organised

PUSH and POP:

Push puts a value onto the stack and pop takes it back off. Here's some code. (you can't compile this - yet.)

```
push cx          ;put cx on the stack
push ax          ;put ax on the stack
.
.
.
.               ;this is going to pop them back in
.               ;the wrong so they are reversed
pop cx           ;put value from stack into cx
pop ax           ;put value from stack into ax
```

This means that cx is equal to what ax is and ax is equal to what cx was.

Some instructions:

MOV: this moves a value from one place to another.

Syntax:

MOV **destination, source**

for example:

MOV ax,10	;moves an immediate value into ax
MOV bx,cx	;moves value from cx into bx

INT: calls a DOS or BIOS function, mainly to do something you would rather not write a function for e.g. change video mode, open a file etc.

Syntax:

INT **interrupt number**

Note: an h after a number denotes that that number is in hex.

INT 21h	;Calls DOS standard interrupt 21h
INT 10h	;Calls the Video BIOS interrupt

All interrupts need a value to specify what subroutine to use. This is usually in AH. To print a message on the screen all you need to do is this:

MOV ah,9	;subroutine number 9
INT 21h	;call the interrupt

But first you have to specify what to print. This function needs DS:DX to be a far pointer to where the string is. The string has to be terminated with a dollar sign (\$).

This example shows how it works:

```
MyMessage db    "This is a message!$"
.
.
.
mov dx,OFFSET MyMessage
mov ax,SEG MyMessage
mov ds,ax
mov ah,9
int 21h
```

DB declares a piece of data. This is the same as in BASIC:

```
LET MyMessage$ = "This is a message!"
```

DB declares a byte, DW declares a word and DD declares a Dword.

This is your first assembly program. Cut this in and then assemble it. If you are using A86, type:

```
A86 name of file
```

If you are using Turbo Assembler then type:

```
tasm name of file
```

```
tlink name of file minus extention
```

With tlink you can make a .COM file by using a /t switch.

```
;THIS IS A SIMPLE PROGRAM TO DISPLAY A MESSAGE ON  
THE ;SCREEN. SHOULD OUTPUT This is a message! TO SCREEN  
AND ;RETURN WITH NO ERRORLEVEL. THIS CAN'T BE MADE INTO A  
.COM FILE WITH TLINK AS IT GENERATES AN ERROR.
```

```
.MODEL SMALL ;Ignore the first two lines  
              ;I'll explain what this means later.  
.STACK        ;allocate a stack  
.CODE         ;a directive to start code segment  
  
START:        ;generally a good name to use as an  
              ;entry point  
  
JMP BEGIN     ;jump to BEGIN: (like goto)  
  
MyMessage db "This is a message!$"   
  
BEGIN:        ;code starts here  
  
mov dx,OFFSET MyMessage ;put the offset of message  
                        ;in DX  
mov ax,SEG MyMessage    ;put the segment that the  
                        ;message is in in AX  
mov ds,ax               ;copy AX into DS  
mov ah,9                ;move 9 into ah to call  
int 21h                 ;function 9 of interrupt  
                        ;21h - display a string to  
                        ;screen.  
  
MOV ax,4c00h ;move the value of 4c00 hex into AX  
INT 21h      ;call interrupt 21h, subroutine  
4C ;which returns control to dos, otherwise it would  
of ;crashed. The 00 on the end means what errorlevel  
it ;would return to DOS. This can be checked in a batch  
file  
  
END START     ;end here
```

Some instructions that you need to know:

This is just a list of some basic assembly instructions that are very important and are used often.

ADD Add the contents of one number to another

Syntax:

```
ADD operand1, operand2
```

This adds operand2 to operand1. The answer is stored in operand1. Immediate data cannot be used as operand1 but can be used as operand2.

SUB Subtract one number from another

Syntax:

```
SUB operand1, operand2
```

This subtracts operand2 from operand1. Immediate data cannot be used as operand1 but can be used as operand2.

MUL Multiplies two unsigned integers (always positive)

IMUL Multiplies two signed integers (either positive or negative)

Syntax:

```
MUL register or variable  
IMUL register or variable
```

This multiplies the register given by the number in AL or AX depending on the size of the operand. The answer is given in AX. If the answer is bigger than 16 bits then the answer is in DX:AX (the high 16 bits in DX and the low 16 bits in AX).

On a 386, 486 or Pentium the EAX register can be used and the answer is stored in EDX:EAX.

DIV Divides two unsigned integers(always positive)

IDIV Divides two signed integers (either positive or negative)

Syntax:

```
DIV register or variable  
IDIV register or variable
```

This works in the same way as MUL and IMUL by dividing the number in AX by the register or variable given. The answer is stored in two places. AL stores the answer and the remainder is in AH. If the operand is a 16 bit register then the number in DX:AX is divided by the operand and the answer is stored in AX and remainder in DX.

The way we entered the address of the message we wanted to print was a bit cumbersome. It took three lines and it isn't the easiest thing to remember

```
mov dx,OFFSET MyMessage
mov ax,SEG MyMessage
mov ds,ax
```

We can replace all this with just one line. This makes the code easier to read and it easier to remember. This only works if the data is only in the data segment i.e. small memory model.

```
lea dx,MyMessage
or mov dx,OFFSET MyMessage
```

Using lea is slightly slower and results in code which is larger.

LEA means Load Effective Address.

Syntax:

LEA **destination,source**

Destination can be any 16 bit register and the source must be a memory operand (bit of data in memory). It puts the offset address of the source in the destination.

Keyboard input:

We are going to use interrupt 16h, function 00h to read the keyboard. This gets a key from the keyboard buffer. If there isn't one, it waits until there is. It returns the SCAN code in AH and the ASCII translation in AL.

```
MOV ah,00h      ;function 00h of
INT 16h         ;interrupt 16h
```

All we need to worry about for now is the ascii value which is in al.

Printing a character:

The problem is that we have the key that has been pressed in ah. How do we display it? We can't use function 9h because for that we need to have already defined the string which has to end with a dollar sign. This is what we do instead:

```
;after calling function 00h of interrupt 16h
MOV dl,al      ;move al (ascii code) into dl
MOV ah,02h     ;function 02h of interrupt 21h
INT 21h        ;call interrupt 21h
```

If you want to save the value of ah then push it before and pop it afterwards.

Control flow:

Firstly, the most basic command:

```
JMP label
```

This is the same as GOTO in basic.

```
JMP ALabel
.
.
ALabel:
    ;code to do something
```

What do we do if we want to compare something. We have just got a key from the user but we want to do something with it. Lets print something out if it is equal to something else. How do we do that? Its easy. We use the jump on condition commands. Here is a list of them:

Jump on condition commands:

JA	jumps if the first number was above the second number
JAE	same as above, but will also jump if they are equal
JB	jumps if the first number was below the second
JBE	same as above, but will also jump if they are equal
JNA	jumps if the first number was NOT above (same as JBE)
JNAE	jumps if the first number was NOT above or the same as (same as JB)
JNB	jumps if the first number was NOT below (same as JAE)
JNBE	jumps if the first number was NOT below or the same as (same as JA)
JZ	jumps if the two numbers were equal
JE	same as JZ, just a different name
JNZ	jumps if the two numbers are NOT equal
JNE	same as above

[NOTE: There are quite a few more but these are the most useful. If you want the full list then get a good assembly book]

They are very easy to use.

Syntax:

```
CMP register containing value, a value
jump command destination
```

An example of this is:

```
cmp al, 'Y'        ;compare the value in al with Y
je ItsYES           ;if it is equal then jump to ItsYES
```

The following program is an example of how to use control and input and output.

```
.MODEL    SMALL
.STACK                                ;define a stack
.CODE
Start:                                ;a good place to start.

    lea dx,StartUpMessage            ;display a message on the
                                    ;screen
    mov ah,9                         ;using function 09h
    int 21h                          ;of interrupt 21h
    lea dx,Instructions              ;display a message on the
                                    ;screen
    mov ah,9                         ;using function 09h
    int 21h                          ;of interrupt 21h

    mov ah,00h                      ;function 00h of
    int 16h                          ;interrupt 16h gets a
                                    ;character from user
    mov bl,al                        ;save bl
    mov dl,al                        ;move ascii value of key
                                    ;pressed to dl

    mov ah,02h                      ;function 02h of
    int 21h                          ;interrupt 21h displays a
                                    ;character to screen
    cmp bl,'Y'                      ;is al=Y?
    je Thanks                       ;if yes then goto Thanks
    cmp bl,'y'                      ;is al=y?
    je Thanks                       ;if yes then goto Thanks
    jmp TheEnd

Thanks:
    lea dx,ThanksMsg                ;display message
    mov ah,9                        ;using function 9
    int 21h                         ;of interrupt 21h

TheEnd:
    lea dx,GoodBye                  ;print goodbye message
    mov ah,9                        ;using function 9
    int 21h                         ;of interrupt 21h
    mov AX,4C00h                    ;terminate program and
                                    ;return to DOS using
    INT 21h                         ;interrupt 21h function 4CH

.DATA
;0Dh,0Ah adds a enter at the beginning

StartUpMessage DB "A Simple Input Program$"
Instructions    DB 0Dh,0Ah,"Just press a Y to continue...$"
ThanksMsg      DB 0Dh,0Ah,"Thanks for pressing Y!$"
GoodBye        DB 0Dh,0Ah,"Have a nice day!$"
```

END

Procedures:

Assembly, like C and Pascal can have procedures. These are very useful for series of commands that have to be repeated often.

This is how a procedure is defined:

```
PROC AProcedure
    .
    .           ;some code to do something
    .
    RET         ;if this is not here then your computer
                ;will crash
ENDP AProcedure
```

You can specify how you want the procedure to be called by adding a FAR or a NEAR after the procedure name. Otherwise it defaults to the memory model you are using. For now you are better off not doing this until you become more experienced. I usually add a NEAR in as compilers can't decide between a near and a far very well. This means if the jump needs to be far the compiler will warn you and you can change it.

This is a program which uses a procedure.

```
;a simple program with a procedure that prints a
message ;onto the screen. (Use /t switch with tlink).
Should ;display Hello There! on the screen.

.MODEL    TINY
.CODE
ORG       100h

MAIN      PROC
    JMP Start           ;skip the data
HI        DB "Hello There!$" ;define a message
Start:    ;a good place to start.
    Call Display_Hi     ;Call the procedure
    MOV AX,4C00h        ;terminate program and return
                        ;to DOS using
    INT 21h             ;interrupt 21h function 4Ch

Display_Hi PROC         ;Defines start of procedure
    lea dx,HI           ;put offset of message into DX
    mov ah,9            ;function 9 of
    int 21h             ;interrupt 21h
    RET                ;THIS HAS TO BE HERE
Display_Hi ENDP         ;Defines end of procedure

Main      ENDP
```

END MAIN

Memory Models:

We have been using the `.MODEL` directive to specify what type of memory model we use, but what does this mean?

Syntax:

```
.MODEL MemoryModel
```

Where MemoryModel can be SMALL, COMPACT, MEDIUM, LARGE, HUGE, TINY OR FLAT.

Tiny:

This means that there is only one segment for both code and data. This type of program can be a `.COM` file.

Small:

This means that by default all code is placed in one physical segment and likewise all data declared in the data segment is also placed in one physical segment. This means that all procedures and variables are addressed as NEAR by pointing at offsets only.

Compact:

This means that by default all elements of code (procedures) are placed in one physical segment but each element of data can be placed in its own physical segment. This means that data elements are addressed by pointing at both at the segment and offset addresses. Code elements (procedures) are NEAR and variables are FAR.

Medium:

This is the opposite to compact. Data elements are near and procedures are FAR.

Large:

This means that both procedures and variables are FAR. You have to point at both the segment and offset addresses.

Flat:

This isn't used much as it is for 32 bit unsegmented memory space. For this you need a dos extender. This is what you would have to use if you were writing a program to interface with a C/C++ program that used a dos extender such as DOS4GW or PharLap.

Macros:

(All code examples given are for macros in Turbo Assembler. For A86 either see the documentation or look in the A86 macro example later in this document).

Macros are very useful for doing something that is done often but for which a procedure can't be used. Macros are substituted when the program is compiled to the code which they contain.

This is the syntax for defining a macro:

```
Name_of_macro  macro
;
; a sequence of instructions
;
endm
```

These two examples are for macros that take away the boring job of pushing and popping certain registers:

```
SaveRegs macro
    pop ax
    pop bx
    pop cx
    pop dx
endm
RestoreRegs macro
    pop dx
    pop cx
    pop bx
    pop ax
endm
```

Note that the registers are popped in the reverse order to they were popped.

To use a macro in your program you just use the name of the macro as an ordinary macro instruction:

```
SaveRegs
; some other instructions
RestoreRegs
```

This example shows how you can use a macro to save typing in. This macro simply prints out a variable to the screen.

The only problems with macros is that if you overuse them it leads to your program getting bigger and bigger and that you have problems with multiple definition of labels and variables. The correct way to solve this problem is to use the LOCAL directive for declaring names inside macros.

Syntax:

```
LOCAL name
```

Where "name" is the name of a local variable or label.

If you have comments in a macro everytime you use that macro the comments will be added again into your source code. This means that it will become unesescarily long. The way to get round this is to define comments with a ;; instead of a ;. This example illustrates this.

```
;a normal comment  
;;a comment in a macro to save space
```

Macros with parameters:

Another useful property of macros is that they can have parameters. The number of parameters is only restricted by the length of the line.

Syntax:

```
Name_of_Macro macro par1,par2,par3  
;  
;commands go here  
endm
```

This is an example that adds the first and second parameters and puts the result in the third:

```
AddMacro macro num1,num2,result  
    push ax          ;save ax from being destroyed  
    mov ax,num1      ;put num1 into ax  
    add ax,num2       ;add num2 to it  
    mov result,ax     ;move answer into result  
    pop ax            ;restore ax  
endm
```

On the next page there is an example of some a useful macro to exit to dos with a specified . There are two versions of this program because both A86 and Turbo Assembler handle macros differently.

```
;this is a simple program which does nothing but use  
a ;macro to exit to dos. This will only work with tasm  
and ;should be compiled like this: tasm /m2 macro.asm  
;because tasm needs more than one pass to work out  
what ;to do with macros. It does cause a warning but this  
can ;be ignored.
```

```
.MODEL small  
.STACK  
.CODE          ;start the code segment
```

```
Start:          ;a good a place as any to start this
```

```
;now lets go back to dos with another macro  
    BackToDOS 0      ;the errorlevel will be 0
```

```
BackToDOS macro errorlevel
```

```
;;this is a macro to exit to dos with a  
specified ;;errorlevel given. No test is done to make  
sure that a  
;;procedure is actually passed or it is within range.
```

```
    mov ah,4Ch          ;;terminate program and return  
                        ;;to DOS using  
    mov al,errorlevel   ;;put errorlevel into al  
    int 21h             ;;interrupt 21h function 4Ch  
  
    endm                ;;end macro  
end                      ;end program
```

Procedures that pass parameters:

Procedures can be made even more useful if they can be made to pass parameters to each other. There are three ways of doing this and I will cover all three methods:

- In registers,
- In memory,
- In stack.

In registers:

Advantages: Easy to do and fast.

Disadvantages: There is not many registers.

This is very easy to do, all you have to do is to move the parameters into registers before calling the procedure. This example adds two numbers together and then divides by the third it then returns the answer in dx.

```
push ax          ;save value of ax
push bx          ;save value of bx
push cx          ;save value of cx
mov ax,10        ;first parameter is 10
mov bx,20        ;second parameter is 20
mov cx,3         ;third parameter is 3
Call ChangeNumbers ;call procedure
pop cx           ;restore cx
pop bx           ;restore bx
pop ax           ;restore dx
.....
ChangeNumbers PROC ;Defines start of procedure
    add ax,bx      ;adds number in bx to ax
    div cx         ;divides ax by cx
    mov dx,ax      ;return answer in dx
    ret
ChangeNumbers ENDP ;defines end of procedure
```

How to use a debugger:

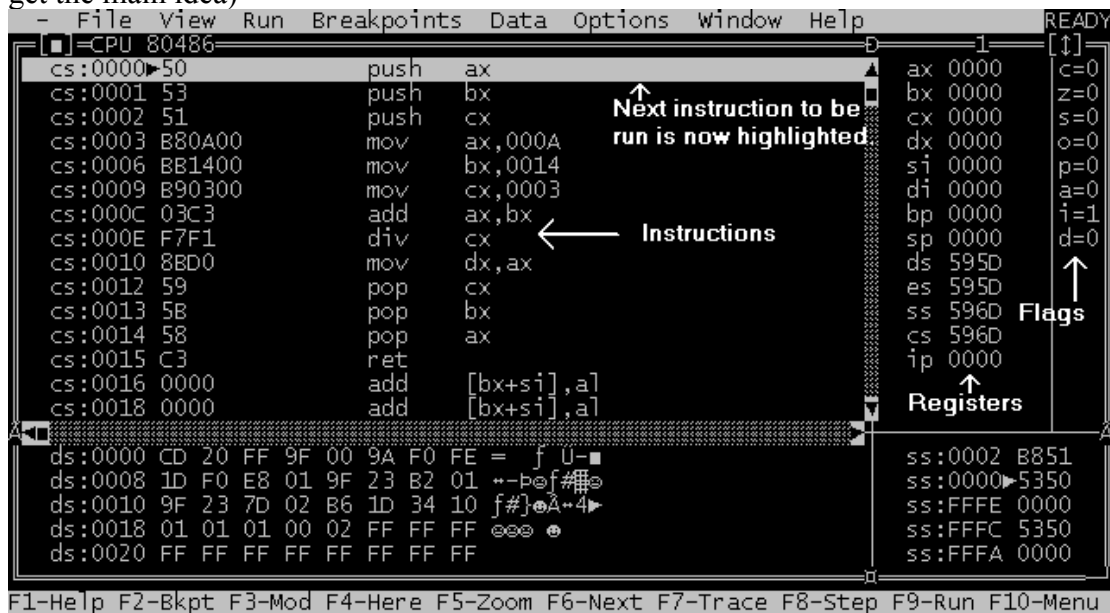
This is a good time to use a debugger to find out what your program is actually doing. I am going to demonstrate how to use Turbo Debugger to check if this program is working properly. First we need to compile this program to either a .EXE or .COM file. Then type:

```
td name of file
```

Turbo Debugger then loads. You can see the instructions that make up your programs, for example the first few lines of this program is shown as:

cs:0000	50	push	ax
cs:0001	53	push	bx
cs:0002	51	push	cx

(This might be slightly different than is shown on your screen but hopefully you will get the main idea)



This diagram shows what the Turbo Debugger® screen looks like

The numbers that are moved into the registers are different that the ones that we typed in because they are represented in their hex form (base 16) as this is the easiest base to convert to and from binary and that it is easier to understand than binary also.

At the left of this display there is a box showing the contents of the registers. At this time all the main registers are empty. Now press **F7** this means that the first line of the program is run. As the first line pushed the ax register into the stack, you can see that the stack pointer (SP) has changed. Press **F7** until the line which contains mov ax,000A is highlighted. Now press it again. Now if you look at the box which contains the contents of the registers you can see that AX contains A. Press it again and BX now contains 14, press it again and CX contains 3. Now if you press **F7** again you can see that AX now contains 1E which is A+14. Press it again and now AX contains A again, A being 1E divided by 3 ($30/3 = 10$). Press **F7** again and you will see that DX now also contains A. Press it three more times and you can see that CX,BX and AX are all set back to their original values of zero.

Passing through memory:**Advantages:** Easy to do and can use more parameters**Disadvantages:** Can be slower

To pass parameters through memory all you need to do is copy them to a variable which is stored in memory. You can use a variable in the same way that you can use a register but commands with registers are a lot faster. This table shows the timing for the MOV command with registers and then variables and then the amount of clock cycles (the speed - smaller faster) it takes to do them.

KEY: reg8 means an 8 bit register eg AL
 mem8 means an 8 bit variable declared with DB
 reg16 means an 16 bit register eg AX
 mem16 means an 16 bit variable declared with DW
 imm8 means an immediate byte eg MOV al,8
 imm16 means an immediate word eg MOV ax,8

Instruction	486	386	286	86
MOV reg/mem8,reg8	1	2/2	2/3	2/9
MOV reg,mem16,reg16	1	2/2	2/3	2/9
MOV reg8,reg/mem8	1	2/4	2/5	2/8
MOV reg16,reg/mem16	1	2/4	2/5	2/8
MOV reg8,imm8	1	2	2	4
MOV reg16,imm16	1	2	2	4
MOV reg/mem8,imm8	1	2/2	2/3	4/10
MOV reg/mem16,imm16	1	2/2	2/3	4/10

These timings are taken from the 'Borland® Turbo Assembler® Quick Reference'

This shows that on the 8086 using variables in memory can make the instruction four times as slow. This means that you should avoid using too many variables unnecessarily. On the 486 it doesn't matter as both instructions take the same amount of time.

The method actually used is nearly identical to passing parameters in registers. This example is just another version of the example given for passing in registers.

```
FirstParam db 0 ;these are all variables to store
SecondParam db 0 ;the parameters for the program
ThirdParam db 0
Answer db 0
.....
    mov FirstParam,10 ;first parameter is 10
    mov SecondParam,20 ;second parameter is 20
    mov ThirdParam,3 ;third parameter is 3
    Call ChangeNumbers
.....
ChangeNumbers PROC ;Defines start of procedure
    push ax ;save ax
    push bx ;save bx
    mov ax,FirstParam ;copy FirstParam into ax
    mov bx,SecondParam ;copy SecondParam into bx
    add ax,bx ;adds number in bx to ax
    mov bx,ThirdParam ;copy ThirdParam into bx
    div bx ;divides ax by bx
    mov Answer,ax ;return answer in Answer
    pop bx ;restore bx
    pop ax ;restore ax
    ret
ChangeNumbers ENDP ;defines end of procedure
```

This way may seem more complicated but it is not really suited for small numbers of this type of parameters. It is much more useful when dealing with strings or large numbers of big values.

Passing through Stack:

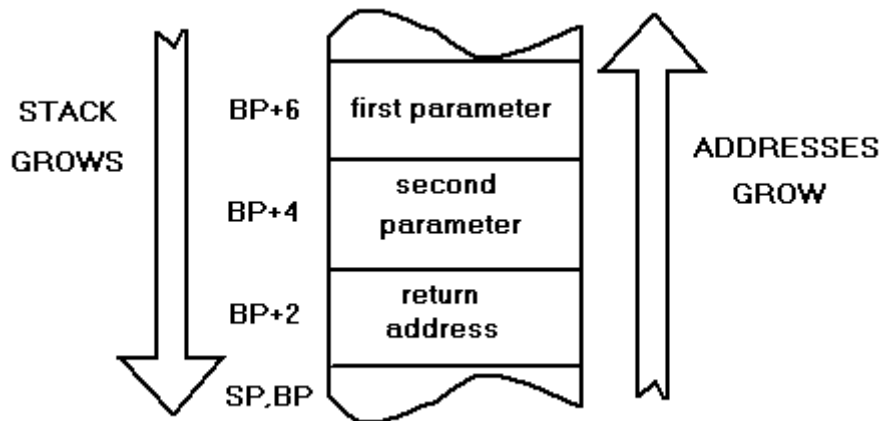
This is the most powerful and flexible method of passing parameters. This example shows the ChangeNumbers procedure that has been rewritten to pass its parameters through the stack.

```

    mov ax,10      ;first parameter is 10
    mov bx,20      ;second parameter is 20
    mov cx,3       ;third parameter is 3
    push ax        ;put first parameter on stack
    push bx        ;put second parameter on stack
    push cx        ;put third parameter on stack
    Call ChangeNumbers
.....
ChangeNumbers PROC ;Defines start of procedure
    push bp
    mov bp,sp
    mov ax,[bp+8]  ;get the parameters from bp
    mov bx,[bp+6]  ;remember that first it is last out
    mov cx,[bp+4]  ;so number is larger
    add ax,bx      ;adds number in bx to ax
    div cx         ;divides ax by cx
    mov dx,ax      ;return answer in dx
    ret
ChangeNumbers ENDP ;defines end of procedure

```

This diagram shows the contents of the stack for a program with two parameters:



To get a parameter from the stack all you need to do is work out where it is. The last parameter is at BP+2 and then the next and BP+4.

Files and how to used them:

Files can be opened, read and written to. DOS has some ways of doing this which save us the trouble of writing our own routines. Yes, more interrupts. Here is a list of helpful functions of interrupt 21h that we are going to need to use for our simple file viewer.

Function 3Dh: open file

Opens an existing file for reading, writing or appending on the specified drive and filename.

INPUT:

AH = 3Dh

AL = access mode

bits 0-2 000 = read only

001 = write only

010 = read/write

bits 4-6 Sharing mode (DOS 3+)

000 = compatibility mode

001 = deny all (only current program can access file)

010 = deny write (other programs can only read it)

011 = deny read (other programs can only write to it)

100 = deny none

DS:DX = segment:offset of ASCIIZ pathname

OUTPUT:

CF = 0 function is succesful

AX = handle

CF = 1 error has occured

AX = error code

01h missing file sharing software

02h file not found

03h path not found or file does not exist

04h no handle available

05h access denied

0Ch access mode not permitted

What does ASCIIZ mean? An ASCIIZ string is a ASCII string with a zero on the end (instead of a dollar sign).

Important: *Remember to save the file handle it is needed for later.*

How to save the file handle:

It is important to save the file handle because this is needed to do anything with the file. Well how is this done? There are two methods, which is better?

- Copy the file handle into another register and don't use that register.
 - Copy it to a variable in memory.

The disadvantages with the first method is that you will have to remember not to use the register you saved it in and it wastes a register that can be used for something more useful. We are going to use the second. This is how it is done:

```
FileHandle DW 0      ;use this for saving the file
                    ;handle

MOV FileHandle,ax     ;save the file handle (same as
                    ;FileHandle=ax)
```

Function 3Eh: close file

Closes a file that has been opened.

INPUT:

AX = 3Eh
BX = file handle

OUTPUT:

CF = 0 function is successful
AX = destroyed
CF = 1 function not successful
AX = error code - 06h file not opened or unauthorized handle.

Important: *Don't call this function with a zero handle because that will close the standard input (the keyboard) and you won't be able to enter anything.*

Function 3Fh: read file/device

Reads bytes from a file or device to a buffer.

INPUT:

AH = 3Fh
BX = handle
CX = number of bytes to be read
DS:DX = segment:offset of a buffer

OUTPUT:

CF = 0 function is successful
AX = number of bytes read
CF = 1 an error has occurred
 05h access denied
 06h illegal handle or file not opened

If $CF = 0$ and $AX = 0$ then the file pointer was already at the end of the file and no more can be read. If $CF = 0$ and AX is smaller than CX then only part was read because the end of the file was reached or an error occurred.

This function can also be used to get input from the keyboard. Use a handle of 0, and it stops reading after the first carriage return, or once a specified number of characters have been read. This is a good and easy method to use to only let the user enter a certain amount of characters.

Note: *If you are using A86 this will cause an error. Change @data to data to make it work.*

```
.MODEL small
.STACK
.CODE
    mov ax,@data      ;base jaddress of data
    mov ds,ax         ;segment

    lea dx,FileName   ;put address of filename in dx
    mov al,2          ;access mode - read and write
    mov ah,3Dh        ;function 3Dh -open a file
    int 21h           ;call DOS service
    mov Handle,ax     ;save file handle for later
    jc ErrorOpening

    mov dx,offset Buffer ;address of buffer in dx
    mov bx,Handle      ;handle in bx
    mov cx,100         ;amount of bytes to be read
    mov ah,3Fh         ;function 3Fh - read from file
    int 21h           ;call dos service
    jc ErrorReading

    mov bx,Handle      ;put file handle in bx
    mov ah,3Eh         ;function 3Eh - close a file
    int 21h           ;call dos service

    mov di,OFFSET Buffer+101 ;Where to put the "$"
    mov byte ptr [di],"$"   ;Put it at es:di
    mov ah,9            ;write a string to the
    mov dx,OFFSET Buffer   ;screen using function 9h
    int 21h             ;of interrupt 21h

    mov AX,4C00h        ;terminate program and return to
                        ;DOS using
    INT 21h            ;interrupt 21h function 4CH

ErrorOpening:
    mov dx,offset OpenError ;display an error
    mov ah,09h          ;using function 09h
    int 21h            ;call dos service
```

--

```

        mov ax,4C01h ;end program with an errorlevel of 1
        int 21h

ErrorReading:
        mov dx,offset ReadError ;display an error
        mov ah,09h             ;using function 09h
        int 21h                 ;call dos service

        mov ax,4C02h ;end program with an errorlevel of 2
        int 21h

.DATA

Handle      DW ? ;variable to store file handle
FileName     DB "C:\test.txt",0 ;file to be opened

OpenError    DB "An error has occurred(opening)!$"
ReadError    DB "An error has occurred(reading)!$"

Buffer       DB 101 dup (?) ;buffer to store data from
                                   ;file one bigger for $

END

```

Function: 3Ch: Create file

Creates a new empty file on a specified drive with a specified pathname.

INPUT:

AH = 3Ch

CX = file attribute

bit 0 = 1 read-only file

bit 1 = 1 hidden file

bit 2 = 1 system file

bit 3 = 1 volume (ignored)

bit 4 = 1 reserved (0) - directory

bit 5 = 1 archive bit

bits 6-15 reserved (0)

DS:DX = segment:offset of ASCIIZ pathname

OUTPUT:

CF = 0 function is successful

AX = handle

CF = 1 an error has occurred

03h path not found

04h no available handle

05h access denied

Important: *If a file of the same name exists then it will be lost. Make sure that there is no file of the same name. This can be done with the function below.*

Function 4Eh: find first matching file

Searches for the first file that matches the filename given.

INPUT:

AH = 4Eh

CX = file attribute mask (bits can be combined)

bit 0 = 1 read only

bit 1 = 1 hidden

bit 2 = 1 system

bit 3 = 1 volume label

bit 4 = 1 directory

bit 5 = 1 archive

bit 6-15 reserved

DS:DX = segment:offset of ASCIIZ pathname

OUTPUT:

CF = 0 function is successful

[DTA] Disk Transfer Area = FindFirst data block

Example of checking if file exists:

```
File      DB "C:\file.txt",0    ;name of file that we want

          LEA dx,File            ;address of filename
          MOV cx,3Fh             ;file mask 3Fh - any file
          MOV ah,4Eh             ;function 4Eh - find first file
          INT 21h                ;call dos service
          JC NoFile
          ;print message saying file exists
NoFile:
          ;continue with creating file
```

This example program creates a file and then writes to it.

```
.MODEL SMALL
.STACK
.CODE

    mov ax,@data      ;base jaddress of data
    mov ds,ax         ;segment

    mov dx,offset StartMessage ;display the starting
                                ;message

    mov ah,09h        ;using function 09h
    int 21h           ;call dos service

    mov dx,offset FileName ;put offset of filename in dx
    xor cx,cx         ;clear cx - make ordinary file
    mov ah,3Ch         ;function 3Ch - create a file
    int 21h           ;call DOS service
    jc Error          ;jump if there is an error

    mov dx,offset FileName ;put offset of filename in dx
    mov al,2          ;access mode -read and write
    mov ah,3Dh         ;function 3Dh - open the file
    int 21h           ;call dos service
    jc Error          ;jump if there is an error
    mov Handle,ax      ;save value of handle

    mov dx,offset WriteMe ;address of information to
                                ;write to the file

    mov bx,Handle      ;file handle for file
    mov cx,38          ;38 bytes to be written
    mov ah,40h         ;function 40h write to file
    int 21h           ;call dos service
    jc error           ;jump if there is an error
    cmp ax,cx          ;was all the data written? Does
                                ;ax=cx?
    jne error         ;if it wasn't then there was an error

    mov bx,Handle      ;put file handle in bx
    mov ah,3Eh         ;function 3Eh - close a file
    int 21h           ;call dos service

    mov dx,offset EndMessage ;display the final message
                                ;on the screen

    mov ah,09h        ;using function 09h
    int 21h           ;call dos service

ReturnToDOS:
    mov AX,4C00h       ;terminate program and return to DOS
    int 21h           ;using interrupt 21h function 4CH
```

```
Error:
    mov dx,offset ErrorMessage ;display an error message
                                ;on the screen
    mov ah,09h                ;using function 09h
    int 21h                   ;call dos service
    jmp ReturnToDOS ;lets end this now

.DATA
StartMessage    DB "This program creates a file called",
                  "NEW.TXT in the C: directory.$"
EndMessage      DB 0Ah,0Dh,"File create OK, look at",
                  "file to be sure.$"
Handle          DW ? ;variable to store file handle
ErrorMessage    DB "An error has occurred!$"
WriteMe         DB "HELLO, THIS IS A TEST, HAS IT",
                  "WORKED?",0 ;ASCIIIZ
FileName        DB "C:\new.txt",0
END
```

How to find out the DOS version:

In many programs it is necessary to find out what the DOS version is. This could be because you are using a DOS function that needs the revision to be over a certain level.

Firstly this method simply finds out what the version is.

```
    mov  ah,30h      ;function 30h - get MS-DOS version
    int  21h         ;call DOS function
```

This function returns the major version number in AL and the minor version number in AH. For example if it was version 4.01, AL would be 4 and AH would be 01. The problem is that if on DOS 5 and higher SETVER can change the version that is returned. The way to get round this is to use this method.

```
    mov  ah,33h      ;function 33h - actual DOS version
    mov  al,06h      ;subfunction 06h
    int  21h         ;call interrupt 21h
```

This will only work on DOS version 5 and above so you need to check using the former method. This will return the actual version of DOS even if SETVER has changed the version. This returns the major version in BL and the minor version in BH.

Fast string print:

We have been using a DOS service, function 9 of interrupt 21h to print a string on the screen. This isn't too fast nor does it allow us to use different colours or position the text. There is another way to print a string to the screen - direct to memory. This is harder as you have to set up everything manually but it has a lot of benefits mainly speed.

```

TextAttribute db 7 ;contains the character attribute
                ;default is grey on black

.....
FastTextPrint PROC
.286           ;need this for shift instructions. Take out if
less
                ;than 286
;=====
;INPUT: AH - Row
;        AL - Column
;        CX - Length of string
;        DS:DX - The string
;        TextAttribute - the colour of the text
;OUTPUT: none
;=====
        push ax bx cx dx bp si di es ;save registers
        mov bl,ah                    ;move row into bl
        xor bh,bh                    ;clear bh
        shl bx,5                     ;shift bx 5 places to the left
        mov si,bx                    ;move bx into si
        shl bx,2                     ;shift bx 2 places to the left
        add bx,si                    ;add si to bx
        xor ah,ah                    ;clear ah
        shl ax,1                     ;shift ax 1 place to the left
        add bx,ax                    ;add ax onto bx
        mov ax,0b800h                ;ax contains text video memory
        mov es,ax                    ;move ax into es
        mov si,dx                    ;mov dx into si
FastTextPrintLoop:
        mov ah,ds:[si]               ;put the char at ds[si] into ah
        mov es:[bx],ah               ;move the char in ah to es[bx]
        inc si                       ;increment si (si+1)
        inc bx                       ;increment bx (bx+1)
        mov ah,TextAttribute         ;put the attribute into ah
        mov es:[bx],ah               ;put ah into es position at bx
        inc bx                       ;increment bx (bx+1)
        loop FastTextPrintLoop       ;loop CX times
        pop es di si bp dx cx bx ax ;restore registers
        ret                          ;return
FastTextPrint ENDP

```

Explanation of new terms in this procedure:

In this procedure there was several things that you have not come across before. Firstly the lines:

```
push ax bx cx dx bp si di es ;save registers
pop es di si bp dx cx bx ax ;restore registers
```

This is just an easier way of pushing and popping more than one register. When TASM (or A86) compiles these lines it translates it into separate pushes and pops. This way just saves you time typing and makes it easier to understand.

Note: To make these lines compile in A86 you need to put commas (,) in between the registers.

This line might cause difficulty to you at first but they are quite easy to understand.

```
mov ah,ds:[si] ;put the char at ds[si] into ah
```

What this does is to move the number stored in DS at the location stored in SI into AH. It is easier to think of DS being like an array in this command. It is the same as this line in C.

```
ah = ds[si];
```

Shifts:

There are four different ways of shifting numbers either left or right one binary position.

SHL Unsigned multiple by two

SHR Unsigned divide by two

SAR Signed divide by two

SAL same as SHL

The syntax for all four is the same.

Syntax:

SHL operand1,operand2

Note: The 8086 cannot have the value of operand2 other than 1. 286/386 cannot have operand2 higher than 31.

Using shifts is a lot faster than using MUL's and DIV's.

Loops:

Using Loop is a better way of making a loop then using JMP's. You place the amount of times you want it to loop in the CX register and every time it reaches the loop statement it decrements CX (CX-1) and then does a short jump to the label indicated. A short jump means that it can only 128 bytes before or 127 bytes after the LOOP instruction.

Syntax:

Loop **Label**

Using graphics in mode 13h:

Mode 13h is only available on VGA, MCGA cards and above. The reason that I am talking about this card is that it is very easy to use for graphics because of how the memory is arranged.

First check that mode 13h is possible:

It would be polite to tell the user if his computer cannot support mode 13h instead of just crashing his computer without warning. This is how it is done.

```
CheckMode13h:
;Returns: DX=0 Not supported, DX=1 supported
    mov ax,1A00h        ;Request video info for VGA
    int 10h             ;Get Display Combination Code
    cmp al,1Ah          ;Is VGA or MCGA present?
    je Mode13hSupported ;mode 13h is supported
    xor dx,dx            ;mode 13h isn't supported dx=0
Mode13hSupported:
    mov dx,1            ;return mode13h supported
```

Just use this to check if mode 13h is supported at the beginning of your program to make sure that you can go into that mode.

Note: I have not tested this on a computer that doesn't have VGA as I don't have any. In theory this should work but you should test this on computers that don't have VGA and see if it works this out.

Setting the Video Mode:

It is very simple to set the mode. This is how it is done.

```
mov ax,13h      ;set mode 13h
int 10h         ;call bios service
```

Once we are in mode 13h and have finished what we are doing we need to we need to set it to the video mode that it was in previously. This is done in two stages. Firstly we need to save the video mode and then reset it to that mode.

```
VideoMode db ?
....
mov ah,0Fh      ;function 0Fh - get current mode
int 10h         ;Bios video service call
mov VideoMode,al ;save current mode

;program code here

mov al,VideoMode ;set previous video mode
xor ah,ah        ;clear ah - set mode
int 10h         ;call bios service
mov ax,4C00h     ;exit to dos
int 21h         ;call dos function
```

Now that we can get into mode 13h lets do something. Firstly lets put some pixels on the screen.

Function 0Ch - Write Graphics Pixel

Makes a color dot on the screen at the specified graphics coordinates.

INPUT:

AH = 0Ch
AL = Color of the dot
CX = Screen column (x coordinate)
DX = Screen row (y coordinate)

OUTPUT:

Nothing except pixel on screen.

Note: This function performs exclusive OR (XOR) with the new color value and the current context of the pixel of bit 7 of AL is set.

```
mov ah,0Ch      ;function 0Ch
mov al,7        ;color 7
mov cx,160      ;x position -160
mov dx,100      ;y position -100
int 10h         ;call bios service
```

This example puts a pixel into the middle of the screen in a the color grey. The problem with this method is that calling interrupts is really slow and should be avoided in speed critical areas. With pixel plotting if you wanted to display a picture the size of the screen you would have to call this procedure 64,000 times (320 x 200).

Some optimizations:

This method isn't too fast and we could make it a lot faster. How? By writing direct to video memory. This is done quite easily.

The VGA memory starts at 0A000h. To work out where each pixel goes you use this simple formula:

$\text{Location} = 0A000h + \text{Xposition} + (\text{Yposition} \times 320)$

Location is the memory location which we want to put the pixel.

This procedure is quite a fast way to put a pixel onto the screen. Thanks go to Denthor of Asphyxia as I based this on his code.

```
PutPixel PROC
.286          ;enable 286 instructions for shifts remove if
              ;you have less than an 286.
;=====
;INPUT:      BX=X postion
;            DX=Y position
;            CL=colour
;OUTPUT:     None
;=====
;this can be optimized by not pushing ax if you
;don't ;need to save it. For A86 change push ds es ax to
;push ;ds,es,ax and do the same thing with pop.
    push ds es ax          ;save ds,es and ax
    mov ax,0A000h          ;ax contains address of video
    mov es,ax              ;es contains address of video
    mov di,bx              ;move x position into di
    mov bx,dx              ;mov y postion into bx
    shl dx,8               ;shift dx 8 places to the left
    shl bx,6               ;shift bx 6 places to the left
    add dx,bx              ;add dx and bx together
    add di,bx              ;add di and bx together
    mov al,cl              ;put colour in al
    stosb                  ;transfer to video memory
    pop ax es ds           ;restore ds,es and ax
    ret
PutPixel ENDP
```

Thank you for reading. I hope that you have learnt something from this. If you need any more help then email me.

Gavin Estey 19/2/95

