

LECTURE NOTES FOR CS-5330: IBM PC ASSEMBLY LANGUAGE  
UNIVERSITY OF TEXAS AT DALLAS

## TABLE OF CONTENTS

## Lecture 1

General references.

Introduction to the Course.

What is Assembly Language?

Advantages and disadvantages. Comparison to higher-level languages.

IBM PC System Architecture.

CPU. Memory. Peripherals. Architecture.

**ASSIGNMENT #1.**

Read Chapter 0.

The Process of Assembly.

The editor, assembler, linker, and debugger.

DOS and Simple File Operations.

Powering up the computer. The prompt. The default drive. Files. The directory. DIR. ERASE. RENAME. TYPE. COPY. Executable files. Function keys (F3, Ctrl-Alt-Del).

EDLIN and Editing.

List (L). Line numbers (".#" and "#"). Insert (I). F1, F2x, DEL, F4x, ESC, INS. Delete (D). Search (S). Replace (R). Quit (Q). End (E).

**ASSIGNMENT (continued).**

Practice EDLIN.

## Lecture 2

Comments.

Booting. Floppy disks. Formatting disks. Getting hardcopy.

Review.

Intel 8088 CPU Registers.

Registers. General purpose registers. Accumulator. Instruction Pointer. Stack Pointer. Flag register.

**ASSIGNMENT #2.**

Read and do problems for chapter 1.

Memory Usage.

Megabytes and kilobytes. Words and doublewords. Variables. DB, DW, and DD.

The MOV Instruction.

A Simple Program Fragment.

Play Misty for Me.

DEBUG.

Debugging. Absolute addresses vs. symbolic addresses. Quit (Q). Assemble (A). Unassemble (U). Enter (E). Dump (D). Go (G). Register (R).

## Handout

Quick Reference: MS-DOS, EDLIN, and DEBUG.

## Lecture 3

Comments.Review.More DEBUG Commands.

How DEBUG stores programs. Name (N). Load (L). Write (W).

Arithmetic Instructions.

ADD, SUB. Timing. INC, DEC. Carry Flag. ADC, SBB.

INTEGER\*8 addition sample program.

Flags.

PF, AF, ZF, SF, OF.

Jumps and Conditional Jumps.

JMP. Labels. JC, JNC, JZ, JNZ, ... JP, JPE. JCXZ. A

simulated Pascal for-loop, programmed using DEBUG.

**ASSIGNMENT #3.**

Write two simple assembler programs in DEBUG.

## Handout

Sample INTEGER\*8 Addition Program.

## Lecture 4

Comments.

BYTE PTR, WORD PTR.

Review.**ASSIGNMENT #4.**

Read portions of chapter 3.

Addressing Modes.

Immediate, register, direct, register indirect modes.

OFFSET. Index registers (SI, DI). Base registers (BX, BP).

INTEGER\*8 addition example. Base relative and direct indexed

modes. INTEGER\*8 addition example. The CLC and LOOP

instructions. Base indexed mode.

ASCII.

Character representation. Control characters (bell, backspace, tab, line feed, form feed, carriage return, escape). Use of a character data type in assembly language.

"0"-"9". "A"-"Z". "a"-"z". Example program to convert two ASCII hexadecimal digits in DH and DL to a byte in AL.

## Lecture 5

Review.System Calls.

MS-DOS interrupt 21H. Character input (functions 1 and 8).

Character output (functions 2 and 5). CMP. Carriage returns

as opposed to carriage-return/line-feeds. Displaying

messages (function 9).

Segments.

Segment:offsets. Segment registers. Default segments.

Pseudo-ops and the Format of .ASM Programs.

SEGMENT, ENDS, ASSUME, EQU.

**ASSIGNMENT #5.**

Write two "typewriter" programs using MASM and LINK.

Running the Assembler and the Linker.Logical Instructions.

NOT, AND, OR, XOR, TEST.

## Handout

Template Program.

Assignment #5.

## Lecture 6

Comments.

Grading policy.

Review.The Shift and Rotate Instructions.

Shifting. Shift counts. SHR and SAR. SHL and SAL. ROL and ROR. RCL and RCR. Sample program to multiply by 10.

The Jump Instructions (continued).

FAR, NEAR, and SHORT jumps. Jumps used with CMP: JA, JAE, JB, ..., JNE, JG, JGE, JL, ....

Stack Operations, PROCs, and CALLs.

CALL, RET, PROC, ENDP, PUSH, POP. 2 rules for stack use. Sample procedure to convert to upper case.

**ASSIGNMENT #6.**

Write a procedure to display a 2-digit hexadecimal number given a byte, and write a "file dump" program using it.

## Handout

Simple "Typewriter" Program Using Procedures.

Assignment #6.

## Lecture 7

Comments.Review.The Stack.

Queues. Stacks. "The" stack. How PUSH, POP, CALL, and RET work. How our template program works. Recursion. A sample recursive procedure to compute N!. The MUL instruction.

Accessing Files.

Generally used "file" operations: read filename from keyboard; open or create a file; seek a given record in the file; read or write the record; close the file. Pseudo-code version of the file-dump program.

DOS Functions for Files.

String input (function 0AH); string buffer. File open (function 3DH); file access codes; ASCIZ file name termination; file handles; DOS error reporting. Read record (function 3FH); sequential file access; byte count. File close (function 3EH).

## Lecture 8

Comments.

Reserved words. A third rule for stack use.

Review.More About DOS, From the User's Standpoint.

Wildcards (ambiguous file names). "File" names for devices: CON, PRN, LPTn, AUX, COMn, NUL. I/O redirection.

More DOS File Functions.

Create file (function 3CH); file attribute word. Seek record (function 42H); random file access; record numbers; file access method. Sample programs to: seek an absolute position in the file, move to the end of the file, and backspace in the file. Write record (function 3FH). Predefined file handles: "standard" input, output, error, auxiliary, and printer devices. Writing a record to the standard output as an alternative to function 9. Delete file (function 41H). Rename file (function 56H).

**ASSIGNMENT #7.**

Read in chapter 2. Write a program to convert a WordStar style text file to a standard ASCII text file.

Macros.

Uses for macros. Arguments. MACRO, ENDM. A sample macro to display a character.

## Handout

List of "Reserved" Words in MASM.  
DOS Function Reference Sheet (INT 21H).  
Assignment #7.

## Lecture 9

Comments.

=.

Review.More About Macros.

Sample string display macro. Sample NEAR conditional JC macro. LOCAL. Sample "generic" NEAR conditional jump macro. &. "Libraries" of macros. INCLUDE. IF1, ENDIF. Sample macros for file operations: CLOSE, WRITE, READ. Macros make programs simpler to understand.

The IF Pseudo-ops.

Conditional assembly. IF, ENDIF, IFE, ELSE.

String Instructions.

LODSB, LODSW, STOSB, STOSW, MOVSB, MOVSW, CMPSB, CMPSW, SCASB, SCASW. Auto-increment and auto-decrement. Source strings (DS:SI) and destination strings (ES:DI). CLD, STD. "Reverse" nature of CMPSB, CMPSW, SCASB, and SCASW. Auto-repeat: REP, REPE, REPNE.

## Lecture 10

Comments.

Backups.

Review.**ASSIGNMENT #8.**

Read in chapter 3 and do problems. Read chapter 5.

Separate Assembly of Procedures.

PUBLIC. Template for individually assembled procedures.

Sample procedure to display a decimal number given a binary number in AL. EXTRN. Linking.

More About Keyboard Input.

Unfiltered keyboard input (DOS function 7). Keyboard status (DOS function 0BH). Extended IBM PC keyboard codes. An improved keyboard reading macro. IFNB and IFB. Omitting arguments in macros.

The ANSI Driver.

Special features of the video display. A universal software interface. Installing the ANSI driver. Escape sequences.

Macros for the ANSI driver. Sample "typewriter" program using some features of the ANSI driver.

## Handout

Reference Sheet for the ANSI Driver and its Macros.

## Lecture 11

Comments.

"Mnemonics". Order of macro arguments. Angle brackets in macro and conditional assembly arguments.

Review.**ASSIGNMENT #9.** MID-TERM PROJECT.

Write a program that takes a text file as input, sorts the lines into alphabetical order, and creates a sorted file as output.

Hand-compilation of Pascal Procedures.

Advantages. "Faking" Pascal constructs: procedures, arguments of procedures. RET n. Use of EQU to give stack variables names. Local variables. For-loops. Helpful macros: memory-to-memory MOV and CMP. Disadvantages. Sample program to average an integer array. While-do. Passing arrays as arguments.

## Handout

Assignment #9.

Comparison of Some Internal Sorting Methods.

## Lecture 12.

Comments.Review.More Thoughts on Hand-compilation.

Full-blown macro approach to hand-compilation. Macros: RETURN, BEGIN, VAR, FOR, ENDFOR, DOWNT, ENDDOWN, REPEAT, UNTIL, WHILE, DO, ENDDO, MIF, MTHEN, MELSE, MENDIF. Sample hand-compilation of Euclid's algorithm. The IRP pseudo-op. Sample hand-compilation of Bubble Sort.

## Handout

Pascal Source Code for Various Sorting Algorithms.

## Lecture 13

Comments.

Shortening the macro library to speed compilation. LEA:  
alternative to OFFSET; unusual addition operation.

Review.**ASSIGNMENT #10.**

Read chapters 4 and 9.

Miscellaneous Topic 1: String Comparisons.

Format of strings required by midterm project. Algorithm for string comparison. LAHF and SAHF. Macro for string comparison. Tricks: OR for checking if zero; XOR for setting to zero. XCHG.

Miscellaneous Topic 2: The System Librarian, LIB.

Libraries of assembled procedures. Fuller explanation of LINK syntax. LIB operations: +, -, \*. The advantage of using libraries.

Miscellaneous Topic 3: Jump Tables and Call Tables.

Jump tables. Uses. Sample program to process control characters.

Miscellaneous Topic 4: The Program Segment Prefix (PSP).

The PSP. Useful data in the PSP. Memory size. Passing parameters to the program from the DOS command line. Sample program to display the passed parameters. Switches. Sample program to locate switches.

## Lecture 14

Comments.

Ideas for final projects.

Review.Overview of the 8087 Numeric Coprocessor.

The need for and advantages of floating-point hardware. Speed and new data types of the 8087.

Programmer's Model of the 8087.

The 8087 as an extension to the 8088. DW, DD, DQ, and DT for defining 8087 data types in memory. 8087 registers: designations; internal data representation; used as a stack.

Some Simple 8087 Instructions: Loading and Storing.

Using FLD, FSTP, FILD, FISTP, FBLD, and FBSTP with memory variables. Automatic type conversions. Sample program to convert 8087 PACKED DECIMAL to an ASCII decimal display. FWAIT; the 8087 as an independent coprocessor; parallel processing. Using FLD and FSTP with 8087 registers. FST and FIST.

## Handout

Some Ideas for Final Projects.  
8087 Reference Sheet 1.

## Lecture 15

Comments.

Coming to terms with "Phase errors".

Review.The .8087 Pseudo-op and the FINIT Instruction.Less Primitive 8087 Instructions: The Four Operations.

Fop, FopP, FopR, FIop, FIopR. Sample expression calculation. The 8087 can calculate more quickly than it can load and store. Sample application: evaluating a real polynomial of a real argument by Horner's rule; evaluating a real polynomial of a complex argument by a modified Horner's rule. FXCH.

The 8087 Status Word.

Exceptions: invalid operation, denormalized operand, zero divide, overflow, underflow, precision. Sticky bits. FSTSW. FCLEX. Special numbers: unnormal, NAN, normal, infinity, zero, empty, denormal, indefinite. Masked responses. Condition codes. FXAM. Sample program to check the special number type.

Some Very Simple 8087 Instructions.

FABS, FCHS, FNOP (and NOP), FRNDINT, FSQRT. The speed of FSQRT. FLDLG2, FLDLN2, FLDL2E, FLDL2T, FLDPI, FLDZ, FLD1. Sample program to compute the solution of a quadratic equation, with error checking.

## Handout

8087 Numeric Coprocessor Reference Sheet 2.

Final Project Option #1: Some Database Functions.

## Lecture 16

Comments.Review.8087 Comparison Instructions.

FCOM, FCOMP, FCOMPP, FICOM, FICOMP, FTST. The condition code after a comparison. Putting the condition code into the 8088 flag register. A sample macro to test the results of an 8087 comparison. A sample program to sort three real numbers.

**ASSIGNMENT #11.**

Read chapter 6.

The IBM PC BIOS.

"Layers" of the operating system. BIOS.

Interrupts.

Interrupt vectors. Software interrupts. Hardware interrupts. IRET.

Useful BIOS Interrupts.

Print screen (INT 5H). Equipment check (INT 11H). Serial I/O (INT 14H). Keyboard break (INT 1BH). Timer tick (INT 1CH). Graphics character table (INT 1FH). Video I/O (INT 10H):

Display modes of the computer. Video display hardware. Pixels. Set size of cursor (function AH=1). Pages. Change cursor position and page (function AH=2). Change active page (function AH=5). Sample "typewriter" program with paging. Scrolling (functions AH=6 and AH=7).

## Lecture 17

Comments.

Review.

**ASSIGNMENT #12.**

Read chapter 7.

Simple Computer Graphics.

Primitive operations: DRAW\_DOT and setting graphics mode. A sample macro to draw a horizontal line. Non-horizontal lines. Pseudo-code for a non-horizontal line drawer. A sample program which puts some text and graphics on the screen.

Implementing DRAW DOT Simple-mindedly.

Using BIOS INT 10H, function AH=12. Execution time overhead.

Video Memory.

Relating the screen image to video memory. The attribute byte in text mode: blinking, colors, etc. The layout of video memory in text mode. Direct manipulation of video memory in text mode.

The 320x200 Color Graphics Mode.

The layout of video memory in 320x200 color graphics mode. DRAW\_DOT using direct video memory manipulations. XORing. Colors. The background color. The palette. Using BIOS INT 10H, function AH=11, for setting the background color and the palette.

## Handout

Final Project Option #2: Infinite Precision Arithmetic.

## Lecture 18

Review.

The 640x200 Graphics Mode.

The layout of video memory in 640x200 mode. DRAW\_DOT in this mode. Selection of the pixel color with the background color.

User-definable Character Sets.

The required steps to define and use new characters. Setting up a character table. Changing interrupt vector 1FH to point to it, using DOS function 25H. A sample program to define the greek character pi, and fill the screen with it.

Direct Programming of the CRT Controller Chip.

I/O ports. IN and OUT. Use of the IBM PC I/O address space. The 6845 CRT controller and related I/O ports. Accessing the 18 internal registers of the 6845. Scrolling by individual character. Making the cursor invisible. Horizontal and vertical retrace. Advantages of performing video I/O during retrace. A program to detect retrace.

Introduction to the Serial Port Hardware.

Serial interfaces. The 8250 UART. Port assignment for serial I/O. Data input and output. Checking for I/O status and errors using the line status port. A dumb terminal program.

Hardware References.

## Handout

TI PC Video I/O BIOS Interrupt and Video Memory Programming.

## Lecture 19

Comments.

Review.

Initializing the Serial I/O Protocol.

Using the line-control register to define port assignments.  
Selecting the baud rate. Selecting other parameters.  
Sending a break character.

Hardware Handshaking for Serial I/O.

The need for handshaking. RTS, CTS, DSR, DTR, DCD, and RI.  
Reading these signals from the modem status port. Setting  
these signals from the modem control port. Adding  
handshaking to the dumb terminal program. Overall interrupt  
enable. Loop-back. Software handshaking.

Interrupt-Driven Serial I/O.

The need for interrupt-driven I/O. The interruptable  
conditions of the UART. Enabling the interrupts. Processing  
the interrupts. A primitive interrupt handler. Use of  
circular queues for input and output buffering. A sample  
circular queue implementation.

**ASSIGNMENT #13.**

Read chapter 8.

Other Topics: Sound.

## Lecture 20

Comments.

Review.

What Else?

The LDS and LES Instructions.

Interfacing to Higher-Level Languages.

How interfacing differs from language to language. Microsoft  
FORTRAN. Microsoft Pascal. Compiled and interpreted BASIC.  
Turbo Pascal. A sample dot-product procedure (using the  
8087) for Microsoft FORTRAN, Microsoft Pascal, and Turbo  
Pascal. Benchmark.

Final Words.

Final Final Words.

When the final is due. Burkey's law.

University of Texas at Dallas  
COURSE NOTES FOR CS-5330  
IBM PC ASSEMBLY LANGUAGE

**CLASS 1**

INSTRUCTOR: Ronald S. Burkey  
OFFICE: Founders 1.420  
TELEPHONE: 690-2867  
OFFICE HOURS:

TEACHING ASSISTANT: James Loughheed  
OFFICE:  
TELEPHONE: 690-2156  
OFFICE HOURS:

ASSIGNMENTS: All assignments are due the Monday after they  
are given.

TEXTBOOK:  
L. J. Scanlon, *IBM PC & XT Assembly Language. A Guide for Programmers.*

## ADDITIONAL REFERENCES:

General Reference

D. J. Bradley, *Assembly Language Programming for the IBM Personal Computer.*

MS-DOS System Interrupts

D. N. Jump, *Programmers's Guide to MS-DOS for the IBM PC.*

8087 Numeric Coprocessor Chip

R. Starz, *8087 Applications and Programming for the IBM PC and Other PCs.*

8088 CPU Instruction Set

Intel, *iAPX 88 Book*, or *The 8086 Family User's Manual*, or *iAPX 86/88, 186/188 User's Manual*, or any other manufacturer's literature on the 8086, 8088, 80186, 80188, 80286, or 80288 microprocessors.

Introduction

This course deals with assembly language programming on the IBM PC. Assembly language is very difficult, at least at first, for most people. Because of this, it is very important to do as much actual programming as possible. There are, however, a number of topics that must be discussed first. For instance, before beginning to actually learn any assembly language, we must first learn something about the hardware structure of the IBM PC, the internal operation of the PC's CPU, the use of the IBM PC programs that enable us to create other assembly language programs, and even something about the IBM PC operating system. In order to begin programming as soon as possible, some of these subjects will be only lightly skimmed at first.

We will be doing a lot of programming, so it is very helpful for you to have easy access to a PC. There are PCs available on campus for your use, but it will certainly be easier if you have a machine dedicated for your own use. Although the book and the lectures will

deal solely with the IBM PC (or IBM XT), you may use any "PC-compatible" computer to do your work. For example, I use a Zenith Z-150 computer, and there are many TI Professional computers on campus (and available in the bookstore for a huge discount). For our purposes, a "PC-compatible" computer will be any computer that runs the MS-DOS operating system. These include, for example, the Tandy 1000, Tandy 1200, Tandy 2000, Compaq, Columbia, DG/One, IBM AT, etc. One restriction you should keep in mind, however, is that I will need to see some of your programs in operation -- thus, either your program must run on a "normal" IBM PC or XT (or on a TI Professional), or else you must arrange for a computer of the proper type to be available on occasion to run your programs. Also, I will only be able to provide you with system-specific information for the TI Professional and close IBM PC or XT compatibles. For any other computers, you will have to get any system-specific information on your own.

TI Professional computers and IBM PCs are available in JO 4.918 and JO 4.920 respectively. To use these facilities, you must register in JO 4.920. Introductory seminars on the fundamentals of using these computers will be held in JO 5.504 at 10:00 am on Friday, May 31, and at 10:00 am on Saturday, June 1. These classes last approximately one hour. We will also cover some of these basics in class, but I strongly recommend that you attend the seminars if you are not already familiar with PC-compatible computers.

We will explicitly discuss (and have available for your use) only the "standard" IBM PC programs (written by the software firm Microsoft), namely: the text editor program EDLIN, the assembler ASM (or MASM), and the linker LINK. These programs, particularly EDLIN, are rather bad and you are encouraged to use any alternative functionally equivalent programs. For example, two alternative text editor programs are WordStar (in "non-document" mode) and the built-in editor of Turbo Pascal. An alternative assembler is the "Turbo Assembler" from Speedware. I do not recommend buying these programs, but (if available to you) you may find them somewhat more satisfactory than the Microsoft alternatives. WordStar is apparently available in the computer labs.

The overwhelming majority of IBM PCs and compatibles use the MS-DOS (or "PC-DOS") operating system, so in this course we will not deal with any other PC operating system (Pick, Unix, CPM-86, etc.). Any version of MS-DOS is acceptable, with version 2.0 (or higher) recommended.

#### What is Assembly Language?

"High"-level languages such as BASIC, FORTRAN, Pascal, Lisp, APL, etc. are designed to ease the strain of programming by providing the user with a set of somewhat sophisticated operations that are easily accessed. In the pantheon of high-level languages, FORTRAN is rather low since the most sophisticated features it provides are the ability to do complex arithmetic and to call functions and subroutines. On the other hand, Lisp and APL are somewhat higher since they provide the ability to operate on entire (complicated) data structures.

In these examples, several other distinctions between higher-level and lower-level languages may be perceived. For one thing, FORTRAN is

more flexible in a certain sense than (for example) APL. Using a high-level language is convenient if the sophisticated features provided are those you need, but almost impossible if not. FORTRAN users often exploit knowledge of how the language is implemented to accomplish things that are apparently impossible, such as addressing character strings as integer or logical variables, treating multiply dimensioned arrays as one-dimensional arrays, treating complex numbers as pairs of real numbers, etc. Such tricks are impossible in APL since the user can never understand the APL implementation or exploit his knowledge if he did. APL users have their own "bag of tricks", but these usually involve just an understanding of the proper use of APL operators rather than a determined effort to subvert their use (as in FORTRAN).

Another aspect of high-level languages is that they tend to be efficient only in their narrow range of special use. For example, Lisp may be wonderful if you are doing recursion or playing with list-structures, and APL is wonderful if you are doing number crunching on arrays, but for simple general-purpose items like do-loops they are laughably inefficient compared to FORTRAN or C.

The lesson we derive is this: a very low-level language might be very flexible and efficient (in terms of speed and memory use), but might be very difficult to program in since no sophisticated operations are provided and since the programmer must understand in detail the operation of the computer. The fact that only primitive operations are available means that the *source code* for such a low-level program tends to be *much longer* than the source code for an equivalent higher-level program, even though the final compiled code may be much shorter. One would only want to use such a language if speed and memory consumption were very critical concerns, and only for the most critical parts of a program. If a program spends 90% of its time executing 10% of its code, that 10% might profitably be converted to a very low-level language.

Assembly language is essentially the lowest possible level of language. It is almost a one-to-one representation of the actual instructions understood by the microprocessor, but in a somewhat human-understandable form. Its relationship to high-level languages is exactly as described above.

Some of these points are illustrated by a result apparently discovered by Lipow (M. Lipow, "Number of Faults per Line of Code," *IEEE Trans. on Software Engr.*, vol. 5, SE-8, no. 4, July 1982, as quoted by M. Franklin in *Using the IBM PC: Organization and Assembly Language Programming*). The number of programming faults  $N$  per line of source code is empirically given by

$$N/P = A + B \ln P + C (\ln P)^2 ,$$

where  $P$  is the number of executable lines of source code and  $A$ ,  $B$ , and  $C$  are constants varying with the programming language used. Typical values are  $A=0.001184$ ,  $B=0.0009749$ , and  $C=0.00001855$  for assembly language, and  $A=0.005171$ ,  $B=0.002455$ , and  $C=0.00004638$  for higher-level languages. Thus, *per line* assembly language is more fault-free than a high-level language. However, each line of assembly language performs a much simpler task than a line of, say, Lisp. Thus *many more* lines of

assembly language are need for any given algorithm and the number of faults per program is therefore dramatically larger.

In microprocessor assembly languages (including IBM PC assembly language), the following features are typically built-in: the ability to read the values stored at various "memory locations", the ability to write a new value into a memory location, the ability to do integer arithmetic of limited precision (add, subtract, multiply, divide), the ability to do logical operations (or, and, not, xor), and the ability to "jump" to programs stored at various locations in the computer's memory.

Not included are the ability to directly perform floating-point arithmetic (with microprocessors prior to the iAPX88 used in the IBM PC, even *integer* multiplication and division were not included), the ability to perform graphics, and the ability to access files. All of these functions must either be performed indirectly by (assembly language) software, or else must be performed by optional add-on hardware.

Let us consider a simple comparison of high-level and assembly language programs. Of course, at present we aren't capable of understanding in detail what the assembly program does, but at least it will give us the flavor of assembly programming. Here is a segment of FORTRAN code to average together the N numbers stored in the array X(I):

```

      INTEGER*2 I,X(N)
      INTEGER*4 AVG
      .
      .
      .
C AVERAGE THE ARRAY X, STORING THE RESULT AS AVG:
      AVG=0
      DO 10 I=1,N
10    AVG=AVG+X(I)
      AVG=AVG/N
      .
      .
      .

```

Here, on the other hand, is part of an IBM PC assembly program to do the same thing. As above, we will assume that the values to be averaged are INTEGER\*2, since floating-point arithmetic isn't available, but we will use INTEGER\*4 to store the intermediate results. In the following program, n, avg, and x are names of memory locations used to store data:

```

      mov cx,n           ; cx is used as the loop
                        ; counter. It starts at N and
                        ; counts down to zero.
      mov dx,0          ; the dx register stores the
                        ; two most significant bytes of
                        ; the running sum
      mov ax,0          ; use ax to store the least
                        ; significant bytes
      mov si,offset x   ; use the si register to point

```

```

; to the currently accessed
; element X(I), starting with
; I=0
addloop:
    add ax,word ptr [si] ; add X(I) to the two least
                        ; significant bytes of AVG
    adc dx,0             ; add the "carry" into the two
                        ; most significant bytes of AVG
    add si,2            ; move si to point to X(I+1)
    loop addloop        ; decrement cx and loop again
                        ; if not zero
    div n                ; divides AVG by N
    mov avg,ax          ; save the result as AVG

```

In this example, the assembly program had 10 executable instructions (as opposed to 4 for the FORTRAN program), even though it only performed integer arithmetic. Moreover, writing it required intimate knowledge of how the variables *x*, *n*, and *avg* were stored in memory. Also, the FORTRAN program was somewhat self-documenting in that it was obvious what the program did (even to someone not familiar with FORTRAN). The assembly program, however, was not really understandable (even to the initiated), and required extensive comments. The assembly language version is much faster than the FORTRAN version. The (Microsoft) FORTRAN version executes (on an IBM PC) in about 2.2 seconds with  $n=32000$ , while the assembly version needs about 0.4 seconds.

#### IBM PC System Architecture

I have already referred to IBM PC's microprocessor and to its memory. Let us consider in more detail what these components are and how they fit together to form a computer such as the IBM PC.

The *microprocessor* or CPU is probably the part we really think of as the computer. It has the job of reading "instructions" from the computer's memory and executing them. These instructions are the type mentioned earlier -- they access memory, do arithmetic and logical operations, and perform a few other services as well. In the IBM PC, the CPU consists of a single "chip", called the 8088 or iAPX88, designed by a company called Intel. Various other "compatible" computers may use any of several other similar CPUs: the 8086 (iAPX86), 80188 (iAPX188), 80186 (iAPX186), 80288 (iAPX288), or 80286 (iAPX286). In theory, all assembly language programs written for an 8088 microprocessor should run on all of these other processors as well (though not vice-versa). This does not mean, however, that such programs will necessarily work correctly (even though they run). This will become clearer in a moment.

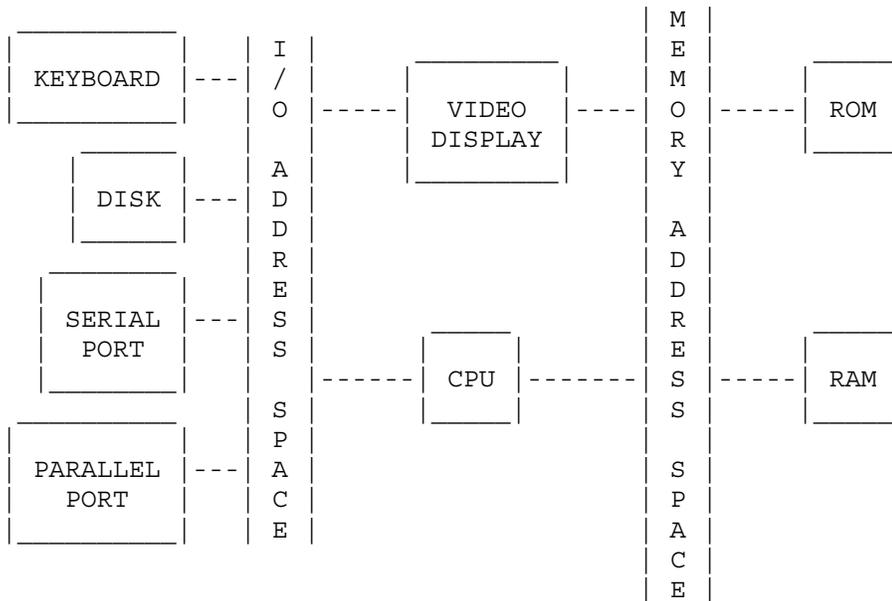
The *memory* of the computer is used to "store" instructions (programs) or data. To the computer, the memory appears as a sequence of locations (or *addresses*). At each address is stored a byte -- i.e., an integer number with a value between 0 and 255. There are two types of memory in an IBM PC. These are the ROM -- or Read Only Memory -- and the RAM -- the Random Access Memory. For a ROM, as its name implies, the stored byte may only be *read* by the CPU. For a RAM memory location, the stored byte may be both read and written (i.e., changed). RAMs also differ from ROMs in that they are "volatile"; when the

computer is turned off, all information stored in RAM is lost. By contrast, the data in a ROM never changes. Both types of memory are "random access" in the sense that the byte at any address may be accessed at any time. In the IBM PC, there are just over one million possible memory addresses. However, not all of these addresses actually refer to RAM or ROMs. Typically, the highest addresses are ROM locations. The lowest 640K ("K" means "1024 bytes") may have RAM memory attached (although 256K is a more usual amount of RAM). Most of the remaining "address space" is unused.

The video display also typically uses a small amount of the remaining address space. This will be discussed much later.

In the IBM PC, there is a 64K address space, known as I/O space, which is completely separate from the one- megabyte memory address-space mentioned earlier. Most I/O occurs by using the I/O space. Each I/O device actually installed in the computer occupies an address, or series of addresses, in I/O space. These devices include the video display controller, the keyboard, the floppy disk controllers, the hard disk controller, the clock, the serial ports (for modem communications, printers, plotters, or other devices), and the parallel port (for the printer).

This system architecture is illustrated in the following figure:



**ASSIGNMENT:** Read and do the problems for Chapter 0 in the book.

Binary Arithmetic

(See the book.)

The Process of Assembly

The process of creating working assembly language programs involves a number of steps, which I will describe in a general way. Later, we will see how to carry out these steps in detail.

Assembly language is a compiled language, in the sense that assembly language source-code must first be created with a text-editor program, and then the source-code must be compiled. Assembly language compilers are universally call "assemblers".

Five types of auxiliary programs are commonly used in 8088 assembly language programming. First, as mentioned above, is the text-editor, which is used to type in assembly language source code and then to edit it when errors are discovered. Second is the assembler. The assembler "assembles" the source code, creating "object" code in the process. The object code is neither executable nor human-readable. The third program is the linker. The linker combines object code modules created by the assembler or by various high-level compilers. For example, if we wrote a program yesterday to convert hexadecimal numbers to decimal, and we write a program today to convert decimal numbers to hexadecimal, then we may want to write a third program tomorrow which, when linked with these, reads two decimal numbers from the keyboard, converts them to hexadecimal, adds them, and writes the back to the screen in decimal. The fourth program, the loader, is actually built in to the operating system and is never explicitly executed. The loader takes the "relocatable" code created by the linker, "loads" it into memory at the lowest available location, and runs it.

The fifth program, the debugger, provides an environment for running and testing assembly language programs. With the debugger, a program may be slowly executed in a controlled way so that errors can be more easily located and corrected. The debugger also incorporates its own simple editor and assembler, so simple assembly language techniques can be typed in and tested "on the fly".

#### DOS and Simple File Operations

The operating system used on the IBM Personal Computer is known as MS-DOS, or PC-DOS, or simply as DOS. DOS provides the environment in which programs (including assembly language programs) run. (We will see later how to access the functions provided by DOS from within assembly programs.)

DOS also provides a set of helpful utility functions which must be understood in order to successfully program under DOS.

When the computer is turned on, the first thing it does is to execute a self-test program. (This can take quite a long time for an IBM PC, and somewhat less for various compatible computers.) After all testable components (particularly the memory) are tested, the computer "boots up" DOS and prompts the user for input. On a system with a hard disk this may happen automatically; if the system has no hard disk (only floppy disks), a floppy disk containing the operating system must be in the appropriate disk drive. (This is usually the leftmost drive.) What prompt is displayed depends a lot on the configuration of the computer and the "boot disk". Typically, the user is prompted to enter the date and the time. Do not skip this.

When the computer displays the prompt "A>" (or "B>", or "C>", etc.), it is in "command mode", waiting for the user to type in a

legitimate DOS command. DOS's job is to manage the IBM PC file system, and most commands somehow manipulate files. Each disk drive attached to the computer is given a letter, starting with "A" for the first floppy drive, "B" for the second, etc. The hard disks, if any, are given letters after the floppies. For example, in a system with two floppies and a hard disk, the hard disk is drive "C".

At any given time, one disk drive is designated as the "default drive". Which drive is the default is indicated by the command prompt. For example, if the input prompt is "B>", then the default drive is "B". The default drive can be changed by typing the new default drive letter followed by a colon (and a carriage return). Thus, "A:<cr>" would change the default drive to "A". Most DOS commands automatically work on the default drive.

The data on a DOS disk is stored in "files", each of which has a name and various other attributes. In DOS, a filename consists basically of 8 (or less) alphanumeric characters. There is also, optionally, a 3 character "extension", which is separated from the name by a period. Here are examples of valid filenames in DOS:

```

FOO
FOO.BAR
TABBY.CAT
12345678.910
FILTHY5.
```

Typically, the 3 character extension is used to designate the *type* of file. Thus, a Pascal source filename always ends in ".PAS", a FORTRAN source filename always ends in ".FOR", an assembler source filename always ends in ".ASM", and executable machine code always ends in ".COM" or ".EXE".

The directory of all files on the default disk can be obtained by using the DOS command "DIR" (the quotation marks are not typed on the computer). Here is a sample directory:

```

Directory of  E:\editing

PCWSMSG S OVR      30976  11-12-84   2:52p
PCSOVLY1 OVR      43264   5-28-85  12:28p
MAILMRGE OVR      13568   3-28-84   2:40p
MERGPRIN OVR       7936   4-24-85   3:43p
WSMSG S  OVR      26624   4-24-85   3:43p
WSOVLY1  OVR      27392   4-24-85   3:43p
SINGULAR TXT      62976   5-02-85   3:01p
5330     CPM      18441   5-27-85   4:19p
SINGULAR BAK      62976   5-02-85   3:00p
TEMP           0       5-28-85   1:00p
5330     BAK      22912   5-28-85  11:50a
5330     TXT      25984   5-28-85  12:59p
      14 File(s)  2207744 bytes free
```

This particular directory contains 14 files, leaving about 2 megabytes of room on the disk for new files. One of the files, called "SINGULAR.TXT" contains 62976 bytes, and was last updated on May 2 at 3:01 in the afternoon. The directory of a drive other than the default

can be obtained by explicitly specifying the drive. Thus, "DIR B:" gives the directory of the "B" drive.

Files can be deleted with the "ERASE fn" command (where "fn" is meant to represent any filename). Thus, "ERASE TEMP" would erase the file named "TEMP" on the default drive, while "ERASE B:TEMP" would erase a file on drive "B".

A file can be renamed with the "RENAME oldname newname" command. Thus, "RENAME TEMP FOO.BAR" would rename TEMP on the default drive as FOO.BAR.

Source files may be displayed on the CRT with the command "TYPE fn". Thus "TYPE SINGULAR.TXT" would display the contents of SINGULAR.TXT on the default drive.

Files may be duplicated or copied from one drive to another by using the "COPY oldfile newfile" command. For example, in order to copy the file named "SPIFFY" on drive A: to drive B: (renaming it "SPOFFY" in the process) we could say "COPY A:SPIFFY B:SPOFFY". If we wanted the new file (on drive "B") to retain the name "SPIFFY", we could use the simpler syntax "COPY A:SPIFFY B:". If we were actually logged onto drive "B" (i.e., if "B" was the default drive), then we could employ the still simpler syntax "COPY A:SPIFFY".

If you type a command which DOS does not recognize as being one of its built-in commands (like those above), it will search the default disk's directory for an executable file of that name. For example, if you typed "EDLIN<cr>", DOS would look for a file named either "EDLIN.EXE" or (in this case) "EDLIN.COM". Having found EDLIN.COM (which is a very poor text editor program used, among other things, for creating assembly language source files), it would then load the program into memory and execute it. From that point on, EDLIN, rather than DOS, would be in control of the computer. Any prompts seen after this are EDLIN prompts and must be responded to with EDLIN commands rather than DOS commands.

DOS also has certain built-in editing functions that you can use if you make a mistake in typing in a command. Using the backspace or back-arrow key erases the last character you typed in. Pressing the escape key erases the entire line you have typed. Pressing the F3 function key automatically repeats the last command you typed in (although you still have to press the carriage return to make it start). There are also several other editing keys that you will become familiar with later.

In dire extremity the computer may be "re-booted" by simultaneously pressing the control, alternate, and delete keys. [Generally, you only need to do this if you have crashed your program. However, on an IBM PC, it is all-too- easy to crash DOS at the same time. If this happens, ctrl- alt-del won't work, and you'll have to turn the computer completely off to re-boot.]

EDLIN and Editing

As mentioned before, the text editor program used for creating assembly language source code is EDLIN.COM. We will discuss this editor (even though it is very bad) because it is the standard editor provided with MS-DOS. [However, anyone lucky enough to have a copy of any other editor is welcome to forget that EDLIN even exists. The only requirement on the editor is that it be capable of producing standard ASCII text files. In particular, WordStar is suitable for this in "non-document" mode. In "document" mode, however, the files WordStar creates are non-standard.]

EDLIN is a "line editor" similar to (and probably derived from) the primitive CP/M line editor ED. In order to edit a file with EDLIN, use the command "EDLIN fn", where "fn" is the filename. The file need not exist in advance (that is, you can create a file from scratch). Having done this, you find yourself in EDLIN command-mode, from which you can type in EDLIN (but not DOS) commands.

EDLIN is rather simple to use. Every line of text in your file is given a number by EDLIN, and you specify these numbers to indicate which lines a particular EDLIN command applies to. For example, suppose you are editing a file called TEST.ASM which contains 100 lines. The EDLIN command to "list" lines of text is "L", and to specify that you wanted to list lines 5 through 24 you would use the command "5,24L". If you did this, EDLIN would respond with something like

```
5: (line 5)
6: (line 6)
.
.
.
24: (line 24)
```

where "(line 5)" represents the text of line 5, etc. Line numbers can run from 1 to 65536, and you can also use the special symbols "." and "#" as line numbers. "." is the same as the line-number of the "current line", and "#" is the highest possible line number. Thus, for example, ".,#L" would list all text from the "current" line to the end of the file.

The command "I" allows you to insert new lines into the text. For instance, "5I" would let you start inserting lines *prior to* the present line 5. EDLIN presents you with a prompt and allows you to type in new lines (just as in DOS command mode) until you finally enter a line of the form F6<cr>. EDLIN also renumbers all of the lines in your file. (The line-numbers used by EDLIN are for editing purposes only and don't actually appear in your file.)

Simply typing a line-number by itself (and a carriage return) allows you to edit just that line. The present form of the line is displayed on the screen, and on the immediately following line you are given a prompt to input the new form of the line. Usually, the present form is nearly correct and you simply want to make a few changes (although if you wanted to you could type in an entirely different line). You do this by using the DOS built-in editing functions. I

mentioned a few of these functions earlier. Here is a more complete list:

F1 or right-arrow: The present character is ok. Move one character to the right.

F2x: The characters up to x are ok. Move over to x.

F3: The entire line is ok (or, at least, any changes are near the end of the line). Move to the end of the line.

left-arrow: Move backward one character.

DEL: Delete the present character.

F4x: Delete the characters up to x.

ESC: Oops! Forget the changes I've made.

INS: Go into "insert mode". In insert mode, you can insert as many characters as you want. To get out of "insert mode", press <INS> again.

Another EDLIN command is the "D", or "delete lines" command. For example, you can delete lines 5 through 24 by using the command "5,24D".

You may search your file for a certain string with the "S" command. For instance, "5,24Sstring" would search lines 5 through 24 for the string "string".

The "R" command can replace a given string by any other string. For example, "5,24RoldstringF6newstring" will replace all occurrences of "oldstring" in lines 5 through 24 by "newstring". You can also put a question mark in front of the "R", and EDLIN will prompt you to see if the replacement is all right.

"Q" quits EDLIN without saving any of the changes you have made. "E" also quits, but saves your work. In general, DOS editors (including EDLIN) do not erase your original copy of the text when they save the new copy. Rather, they rename the old copy to have the same name, but an extension of ".BAK". Thus, you can generally recover a prior version of your file if you make some ghastly error.

**ASSIGNMENT:** Use EDLIN, or any other convenient text editor (suitable for creating source programs), to write a few paragraphs explaining your reason for wanting to learn 8086 assembly language. This description should occupy between one and two single-spaced pages when printed. If your reasons don't occupy this much space, fill the remainder of the space with text from page 178 of the textbook. Use the features of the editor program to correct any typing errors you make.

Questionnaire for CS 5330: IBM PC Assembly Language

- 1) Your name: .
- 2) Your field of study: .  
Graduate Undergraduate
- 3) Your level of programming experience:  
Novice  
Not too bad  
Expert/hacker/professional.
- 4) List any computer languages you feel proficient in:
  
- 5) Have you ever used any type of assembly language?  
No Yes
- 6) Do you own, or have easy access to, any type of IBM-PC compatible computer? (That is, any computer which runs the MS-DOS or PC-DOS operating system.)  
No Yes  
If "yes", what type of computer is it?  
Are you very familiar with this computer?  
Yes No
- 7) What is your reason for taking this course? (Or: What are you hoping to learn from this course?)

University of Texas at Dallas  
COURSE NOTES FOR CS-5330  
IBM PC ASSEMBLY LANGUAGE

**CLASS 2**Some Comments

-1. Beyond the textbook, you are not, NOT, **NOT** required to buy any books or software for this course. In particular, you are not required (or even advised) to buy the products called "Turbo Pascal" or "Turbo Assembler". If you have access to either of these products, on the other hand, Turbo Pascal has a much better text editor than EDLIN, and Turbo Assembler (which I have never used) is reputed to be a better assembler than MASM.

0. The lecture notes are available in WS-compatible format on floppy disk.

1. Several of the surveys stated that the reason for taking the course was "wanted to become computer literate", or similar comments. A graduate-level assembly-language course is not a good place to accomplish this. A certain amount of programming experience is necessary to grasp what's going on. (I have to not only teach assembly language, but the use of the IBM PC as well. I cannot do all of this and additionally teach introductory programming.) I suggest a Pascal course for complete beginners. You should be in this course only if you know *some* programming language on *some* computer fairly well.

2. Office hours: Monday through Thursday, 3:00-7:00.

3. T.A.: Please direct any questions outside of class to me and not to the T.A.

4. In order to use the IBM PC, you must first be able to turn it on and "boot" it up. This involves several things. First, you must have a disk containing the operating system. This disk is known as a "DOS disk" or "system disk". You should not ever store files on this disk. It is used only to hold the operating system and certain utility programs such as the line editor EDLIN.COM. To boot up the system, put the system disk in drive "A" and *then* turn the computer on and close the door of the disk-drive. Similarly, when you are done with the computer, open the door of the disk drive and *then* turn the computer off.

5. Since all of our work will involve creating and manipulating disk files, and since disk files are stored on floppy disks, you will need to acquire some floppy disks. In theory, you need "soft-sectored, double-sided, double-density" disks, but in practice any soft-sectored 5 1/4 inch disks will work. The bookstore sells these for about \$3 apiece. If you are willing to buy them in lots of ten (you will really need more than one disk anyway), you can get them much cheaper. Sabet Electronics on Floyd Rd. sells boxes of ten for \$12, while the Micro Store on N. Central sells (I believe) boxes of ten for \$20. Unless you are a fool who wants to be parted from his money, do not buy disks in Computerland or similar stores.

6. Formatting floppy disks. Before floppy disks can be used to store files, they must be "formatted". To format a new disk, put the *system disk* in drive "A" and the *new disk* in drive "B". Use the command "A:FORMAT B:/V". Check carefully to make sure you don't accidentally format the system disk.
7. Getting printouts. If the computer you are using has a printer attached, you can get printouts in several ways. First, an exact picture of the screen (a "screen dump") can be obtained by using the "print screen" button on the keyboard. Second, the DOS "TYPE" command (which prints a text file on the screen) can be used along the "printer echo" feature of DOS. "Printer echo" can be initiated (and terminated) by pressing ctrl-P (that is, by simultaneously pressing the "ctrl" key and the "P" key). In printer-echo mode, everything that is displayed on the screen is simultaneously sent to the printer. Thus, TYPEing a text file would also result in it being printed. Third, in MS-DOS, i/o devices are treated in some ways like disk files. The printer is usually given the "file" name "LPT1", and you can print a text file by COPYing it to the "file" LPT1. For example, the file TEXT.TXT could be printed by using the command "COPY TEXT.TXT LPT1".

#### Review of the Last Class

Recall that in the last class we basically did three things.

First, we had some general comments on the nature of assembly language and on the IBM-PC. The most important points made were these: That assembly language programs are very small and fast at runtime, but that their source-code is very long and takes a long time to write. That assembly language is very primitive in the sense that most operations are performed by means of integer and logical arithmetic and by instructions that move data around in memory. That assembly language is a *compiled* (or *assembled*) language and that to create a working assembly language program we must run the text editor (usually EDLIN) to create the source code, then run the assembler (usually MASM) to assemble the source code down to object code, and finally to run the linker (LINK) to produce executable code.

Second, since using EDLIN, MASM, and LINK involves creating and manipulating a number of files on the disk, we had to learn something about the MS-DOS operating system. We learned how to use the commands

|        |                                     |
|--------|-------------------------------------|
| DIR    | list the disk's directory           |
| RENAME | rename a file                       |
| ERASE  | erase a file                        |
| TYPE   | display a source file on the screen |
| COPY   | copy a file.                        |

Also, we learned something about DOS's built-in editing commands. DOS always remembers the last command typed and allows you to re-use the command (possibly editing it first). These editing keys are also used extensively in EDLIN. Here are the editing keys we covered:

|                          |  |
|--------------------------|--|
| Backspace or left-arrow: | unrecall a character.                    |
| ESC:                     | unrecall entire line.                    |
| F1 or right-arrow:       | recall one more character.               |
| F2x:                     | recall all characters up to character x. |

F3: recall rest of line.  
 DEL: skip a character.  
 F4x: skip to character x.  
 INS: toggle in/out of "insert" mode.

For example, if the buffer contained "COPY A:FOO.BAR B:", then the with the following sequence of editing commands we would see screen displays:

```
A>_
                                     (F3)
A>COPY A:FOO.BAR B:_
                                     (twelve backspaces)
A>COPY _
                                     (INS C: INS)
A>COPY C:_
                                     (two F1s)
A>COPY C:A:_
                                     (F2R)
A>COPY C:A:FOO.BA_
                                     (eight backspaces)
A>COPY C:_
                                     (F4<space> INS NIFTY.COM INS)
A>COPY C:NIFTY.COM_
                                     (F3)
A>COPY C:NIFTY.COM B:_
```

We also discussed the text editor program EDLIN. We discussed the commands

|                     |                    |
|---------------------|--------------------|
| <i>line1,line2D</i> | delete lines       |
| <i>line</i>         | edit line          |
| E                   | end and save       |
| <i>lineI</i>        | insert line        |
| <i>line1,line2L</i> | list lines         |
| Q                   | end and don't save |

A line number consists of a number from 1 to 65536, or else the symbols "." (meaning the current line) or "#" (meaning the line after the last line in the file).

### Intel 8088 CPU Registers

Generally (though not always) when we program in a high-level language we think in terms of the following types of constructs:

|            |   |
|------------|---|
| CONSTANTS  | numerical, string, or some other quantities whose unchanging actual values are known when the program written |
| VARIABLES  | quantities (whose initial values may or may not be known) whose values change as the program executes         |
| PROCEDURES | functions or subroutines which may or may not have arguments and may or may not return answers                |

None of these items has any real direct equivalent in terms of assembly language. Each, in practice, is a combination of several assembly language features.

In assembly language, on the other hand, much thought goes into the use of the computer's memory (considered as a sequence of bytes or words) and the CPU's *registers*. A register is like a memory location in that it can store a byte (or word) value. [These register sizes apply to CPUs like the 8088, 8086, 8080, Z80, etc. The 68000 CPU has all 4-byte registers. The Z8000 CPU has registers that can be grouped in various ways to contain anything from one byte to 8 bytes. Some TI microprocessors have no registers at all.] However, a register has no address in the computer's memory. Registers are not a part of the computer's memory, but are built into the CPU itself.

Registers are so important in assembly language programming (on microcomputers) for various reasons. First, the variety of instructions using registers tends to be greater than that for operating on values stored at memory locations. Second, these instructions tend to be shorter (i.e., take up less room to store in memory). Third, register-oriented instructions operate faster than memory-oriented instructions since the computer hardware can access a register much faster than a memory location.

The 8086-family of microprocessors have a number of registers, *all* of which are partially or totally dedicated to some specific type of use. Here is a list of the registers and their uses. Do not worry if their uses do not seem clear yet. For the present, it suffices for us that the italicized registers are so specialized that they can *only* be used for their special purpose, while the registers in normal type can often be used just like 16-bit (word) memory locations:

|      |   |
|------|---|
| AX   | The accumulator                         |
| BX   | The pointer register                    |
| CX   | The loop counter                        |
| DX   | Used for multiplication and division    |
| SI   | The "source" string index register      |
| DI   | The "destination" string index register |
| BP   | Used for passing arguments on the stack |
| SP   | <i>The stack pointer</i>                |
| IP   | <i>The instruction pointer</i>          |
| CS   | <i>The "code segment" register</i>      |
| DS   | The "data segment" register             |
| SS   | <i>The "stack segment" register</i>     |
| ES   | <i>The "extra segment" register</i>     |
| FLAG | <i>The flag register</i>                |

The first seven registers might reasonably be called "general purpose" registers since they can be used rather flexibly to manipulate word values until (or unless) their special functions are needed. AX, BX, CX, and DX are more flexible than the others in that they may be used either as word registers (containing 16-bit values) or as pairs of byte registers (containing 8-bit values). The byte-sized registers gotten this way are known as AL, BL, CL, DL, AH, BH, CH, and DH. For example, AL contains the less significant byte of AX, while AH contains the more significant byte.

Several of these special register types are common among microprocessors:

The accumulator is often a special register which is designated to contain the results of certain arithmetic operations. Many instructions execute faster when operating on the accumulator than they do when operating on other registers, which are in turn faster than operations on memory variables. The 8088 has the 8-bit accumulator AL and the 16-bit accumulator AX.

The instruction pointer (or program counter) is a register controlling the execution of programs. Recall that both programs and data are stored in the computer's memory. Most program code is stored in memory in such a way that sequentially executed instructions are actually stored sequentially in memory. The IP (instruction pointer) register contains the address of the next instruction to be executed. For every instruction fetched from memory, the IP is automatically incremented by the number of bytes in the instruction.

The stack pointer (SP) contains the address of the next memory location to be added to the stack. We will discuss stacks later.

The flag register contains a number of bit-sized "flags" describing the status and configuration of the CPU. Its main use is in controlling conditional execution of parts of a program.

**ASSIGNMENT:** Read and do the problems for Chapter 1 in the book.

### Memory Usage

As mentioned before, the 8088 microprocessor can address up to 1 megabyte of memory. "Mega" is a prefix that means "million", so we would expect that a megabyte of memory would represent one million memory locations, each capable of storing one byte. This is almost correct, except that in computing "mega" refers to  $2^{20}=1,048,576$ . (Similarly "kilo", which normally means "thousand", means  $2^{10}=1024$  in computing.) Thus, the 8088 address space contains 1,048,576 locations, each (in theory -- i.e., if the appropriate hardware memory devices are actually installed in the system) capable of storing one byte. Thus, memory addresses range from 0 to 1048575 in decimal, or 0 to FFFFF in hexadecimal.

Memory locations can also be ganged in pairs to store words (16-bit or two-byte values) or in quadruplets to store doublewords (4-byte values). There are additional groupings that we will encounter when studying the *8087 numeric coprocessor* chip which can be optionally installed in the IBM PC. Unlike many processors, which require word and doubleword values to be stored at even addresses, the 8088 allows any size of variable to be stored at any address. However, when 8088 programs are run on the software-compatible 8086 microprocessor (or the 80186 or 80286) word and doubleword variables stored at odd addresses can result in a loss of efficiency (i.e., execution speed).

Note that although the address space of the 8088 is one megabyte in size, we will assume for the present that it is only 64K in size. This is due to limitations of the 8088 processor which we will learn

about in much greater detail later. Thus, for the present, we will assume that all addresses are 16 bits (words) in size, and that they vary from 0 to FFFF (hex).

In assembler, memory space for variables is usually allocated with the DB, DW, and DD operators. These operators associate a *type* (i.e., byte, word, or doubleword) with each variable, and (optionally) assigns a *name* and an initial value as well. The operators are not *instructions* for the CPU to execute; rather they inform the assembler program (MASM) that you intend to symbolically refer to certain memory locations in certain ways. After the source code is assembled, the object code contains only references to real memory addresses and no trace of the original symbolic names exists.

Here is a sample of how variables are declared in 8088 assembly language. Suppose that we want to create a byte-sized variable called "FOO" with the initial value 27 (decimal), a word-sized variable called "BAR" with the initial value of 3E1 (hexadecimal), and a doubleword variable called "REAL\_FAT\_RAT" whose initial value we don't care about. Here is one way of doing this: in our assembly language program we include the lines

```
foo db 27           ; by default all numbers are decimal
bar dw 3e1h        ; appending an "h" means hexadecimal
real_fat_rat dd ?  ; "?" means "don't care about the value"
```

A variable name thus refers to the value *stored* at a particular (and unchanging) address. While the value of the variable may change (that is, the value stored at the memory location), the address of the variable never changes.

### The MOV Instruction

Recall that in the first class we saw a simple example of an assembly language program for averaging together the elements of an integer array. We didn't understand too well at that time what the program did, but one thing that may have been apparent was that out of the ten instructions in the program, five were of the form "mov something,something". The "MOV" instruction is the most important (i.e., most frequently used) instruction in 8088 assembly language and hence is the first 8088 instruction we will learn about. The MOV instruction has the format

MOV *destination,source*

and allows you to MOVE data into or out of registers or memory locations. "Destination" specifies where the data is moved *to* and "source" specifies where the data is moved *from*.

In general, the destination can be either a register or a memory location. The source can also be either a register or a memory location, except that *memory-to-memory* transfers are not allowed. The source can, in addition, be simply a numeric value. Thus, for example, we could load a register with the *value* 5, or we could load it with the value from the *address* 5. While the MOV instruction modifies that value stored in the destination, the source is not changed in any way.

Registers are referred to by the register names listed above (AX, BX, etc.), while memory variables may be referred to by their symbolic names. (Memory locations may also be referred to in a number of other complicated ways, but we will discuss these later.) Here are some examples of MOV instructions, using the variables FOO, BAR, and REAL\_FAT\_RAT defined earlier:

```

mov ax,bar      ; load the word-size register ax with
                ; the word value stored at location bar.
mov dl,foo      ; load the byte-size register dl with
                ; the byte value stored at location foo.
mov bx,ax       ; load the word-size register bx with
                ; the byte value in ax.
mov bl,ch       ; load the byte-size register bl with
                ; the byte value in ch.
mov bar,si      ; store the value in the word-size
                ; register si at the memory location
                ; labelled "bar".
mov foo,dh      ; store the byte value in the register
                ; dh at memory location foo.
mov ax,5        ; store the word 5 in the ax register.
mov al,5        ; store the byte 5 in the al register.
mov bar,5       ; store the word 5 at location bar.
mov foo,5       ; store the byte 5 at location foo.

```

Notice that the size of the source and destination (i.e., byte or word) must match in register-to-register, memory-to-register, or register-to-memory transfers. In *immediate* transfers (in which a constant value is directly stored into the destination), the constant value must be consistent with the size of the destination. In the example above, 5 could be either a byte or a word value; if, however, we had used the value 3172, this could only represent a word (not a byte), so

```

mov al,3172
mov foo,3172

```

are illegal.

#### A Simple Sample Program Fragment

Let us now consider our first fully understandable fragment of 8088 programming. Suppose that we have 4 word-sized values stored in the variables MY, NAME, IS, NOBODY, (storing, say, the initial values 4, 5, 6, and 32) and that we want to move these values to the variables PLAY, MISTY, FOR, ME. In FORTRAN this would be easy. We would have something like this:

```

INTEGER*2 MY,NAME,IS,NOBODY,PLAY,MISTY,FOR,ME
DATA MY,NAME,IS,NOBODY/4,5,6,32/
.
.
.
PLAY=MY
MISTY=NAME
FOR=IS
ME=NOBODY
.
.
.

```

In assembler, on the other hand, we are slightly hampered by the lack of a memory-to-memory version of the MOV instruction. In order to overcome this problem we will do the memory-to-memory move in two steps: first, we will move the value from memory to one of the general-purpose registers of the CPU (say, AX), and then move the value from the register back into memory. Our program fragment might look like this:

```

; destination variables
play    db ?
misty   db ?
for     db ?
me      db ?
; source variables
my      db 4
name    db 5
is      db 6
nobody  db 32

.
.
.

mov ax,my      ; PLAY=MY
mov play,ax
mov ax,name   ; MISTY=NAME
mov misty,ax
mov ax,is     ; FOR=IS
mov for,ax
mov ax,nobody ; ME=NOBODY
mov me,ax

```

This program actually works, as we will see below.

#### DEBUG

Our programs will mainly be developed as described so many times earlier, using EDLIN to create the source code, MASM to assemble it, and LINK to link it. Our very first programs, however, will be written instead using a program called DEBUG. The reason for this is that with DEBUG we can concentrate our thoughts purely on assembly language; with MASM, on the other hand, (as we will see later) we need to do a number of things that have nothing to do with assembly language in order to get our programs to work.

DEBUG is the system "debugger". It has its own built-in editor and primitive assembler, and its code does not need to be linked. Debug also has facilities for modifying memory locations and for examining memory locations. Because of this, we can develop and run simple programs entirely inside of the DEBUG program, much more conveniently than we can with the EDLIN-MASM-LINK cycle.

DEBUG cannot be used to conveniently develop larger programs, however, because (among other things) it does not allow the use of symbolic labels. With DEBUG, one must literally know the memory addresses of all data items. In DEBUG, an (immediate) value is distinguished from the value stored at an address in that an address is enclosed in square brackets. For example, DEBUG accepts

```
mov ax,200
```

as meaning "load ax with the value 200", while it accepts

```
mov ax,[200]
```

as meaning "load ax with the value at address 200". DEBUG also differs from the macro assembler (MASM) in that the default base for numbers is hexadecimal rather than decimal. Thus the "200" used above is really hexadecimal 200, or 512 decimal.

In the "My Name is Nobody" example used earlier we might (if we were using DEBUG) define (in our own minds) the byte variables MY, NAME, IS, NOBODY, PLAY, MISTY, FOR, and ME to reside at memory locations 200 (hex) through 207. In that case, our program fragment would become

```
mov ax,[200] ; PLAY=MY
mov [204],ax
mov ax,[201] ; MISTY=NAME
mov [205],ax
mov ax,[202] ; FOR=IS
mov [206],ax
mov ax,[203] ; ME=NOBODY
mov [207],ax
```

although the comments would be deleted since the debugger is incapable of remembering them.

Let's see how we might run such a program from within DEBUG. First, DEBUG may be run by typing "DEBUG<cr>" from MS-DOS command mode. After DEBUG loads from the disk and begins executing, you will see the prompt "-". This means that you must now enter DEBUG commands rather than DOS commands.

The most important DEBUG command is "Q", or "quit". Entering "Q<cr>" (without the quotes, of course) at the "-" prompt returns you to DOS command mode.

Our little program may be entered with the "A" or "assemble" command. Let "nnnn" represent any address. If you type "Annnn<cr>" at the "-" prompt, DEBUG will enter an input mode in which you can type in

assembly language instructions. These instructions are then immediately assembled into a program, with the first instruction you type beginning at address nnnn. For reasons we will learn about later, many programs begin at location 100 (hex). Therefore, we can get our program into the computer by typing "A100<cr>" and then entering in the lines of our program one by one. Entering a blank line terminates this process. That is, when the program is entirely entered, you just type an extra carriage return to get out of input mode.

We can check that the program is actually in the computer at address 100 with the "U" or "unassemble" command. Entering "Unnnn<cr>" at the "-" prompt unassembles the program. That is, it looks at the bytes stored in memory beginning at location nnnn and *deduces* the assembler instructions you must have typed in. For example, "U100<cr>" would verify our program.

Here is what it looks like to use DEBUG as described above, with our input underlined and the computer's output in normal type:

```

-a100
48EE:0100 mov ax, [200]
48EE:0103 mov [204], ax
48EE:0106 mov ax, [201]
48EE:0109 mov [205], ax
48EE:010C mov ax, [202]
48EE:010F mov [206], ax
48EE:0112 mov ax, [203]
48EE:0115 mov [207], ax
48EE:0118
-u100
48EE:0100 A10002      MOV      AX, [0200]
48EE:0103 A30402      MOV      [0204], AX
48EE:0106 A10102      MOV      AX, [0201]
48EE:0109 A30502      MOV      [0205], AX
48EE:010C A10202      MOV      AX, [0202]
48EE:010F A30602      MOV      [0206], AX
48EE:0112 A10302      MOV      AX, [0203]
48EE:0115 A30702      MOV      [0207], AX
48EE:0118 48          DEC      AX
48EE:0119 023C      ADD      BH, [SI]
48EE:011B 17          POP      SS
48EE:011C 7301      JNB     011F
48EE:011E C3          RET
48EE:011F E87600     CALL    0198

```

The "48EE:" should be ignored for the present. It has to do with the fact that the address space is 1M in size but that we are assuming it to be 64K in size. (If you try this, you will probably get a different number than 48EE.) The 4-digit hexadecimal numbers following the "48EE:" represent the addresses we are using. Notice that in the unassemble the program continues (beginning at address 118) past the point at which we stopped typing in instructions. This is because the DEBUG program does not know or care where your program begins or ends. It is perfectly willing to interpret any garbage hanging around in memory (such as fragments of previously run programs) as valid instructions. Moreover, if you are not careful, it is perfectly willing to execute them. If you don't want to see these extra garbage

instructions, the "U" command is willing to have you specify an ending address as well as a beginning address for the unassembled. Thus, in this case, "U100,118<cr>" would list just the instructions we typed in.

Another inconvenient aspect of DEBUG is that if you make any mistakes entering the program, it is likely that you will have to type in the entire program again, from the point of the mistake. For example, if we had accidentally entered the instruction "mov ax,dx" at address 106, we would have to use the command "A106<cr>" to re-enter all of the statements from "mov ax,[201]" on. The reason we could not simply correct the single bad statement is that different 8088 instructions assemble down to strings of bytes of different lengths. Thus, unless the correct and incorrect statements *assemble* to the same length, either the correct instruction will overlap the succeeding instructions in memory or else will leave a garbage-filled gap between itself and the next instruction. We will see ways later of partially getting around this fact in some simple cases, but there is no generally applicable fix other than re-typing the program.

In any case, before we can run this program we must initialize the variables MY, NAME, IS, and NOBODY (which is to say, the values stored at memory locations 200 through 203). This can be done with the "E" or "enter" instruction. This command allows you to enter hexadecimal values rather than 8088 instructions into memory locations. If you type "Ennn<cr>", DEBUG will go into an input mode in which you can sequentially enter values into the memory locations beginning at address nnnn. You can enter as many bytes (in hexadecimal) as you wish, with the bytes separated by spaces. A carriage return terminates input mode. As you enter the new values of the bytes, DEBUG thoughtfully displays the old values for you, on the grounds that you might not want to change them. If you want to leave a particular byte unchanged, you can simply hit the space bar (without entering a hex value) to go to the next memory location. In our example, to initialize locations 200-203 with 4, 5, 6, and 32 (20 in hexadecimal), we could do this:

```
-e200
419F:0200  77.4   20.5   64.6   69.20
```

Here, the "419F:" is to be ignored as above, and 77, 20, 64, and 69 are the original values stored at addresses 200-203.

[It is also possible to use the DB and DW operators to define these initial values: i.e.,

```
-a200
419F:200  db  4
419F:201  db  5
419F:202  db  6
419F:203  db 20
```

However, this involves some tricky aspects that we will understand better later, so it is best to avoid this method (for now) when using DEBUG.]

We can check to see if this worked as we expected by using the "D" or "display" command. Typing "Dnnnn<cr>" at the "-" prompt displays in hexadecimal the values at the bytes beginning at address nnnn. Another

form of this command is "Dnnnn,mmmm<cr>", which displays all bytes between addresses nnnn and mmmm, inclusive. For example, using this command we might now see:

```
-d200,207
419F:0200  04 05 06 20 72 65 63 74  ... rect
```

Here, the "04" through "74" are the hexadecimal forms of the bytes at addresses 200 through 207. The "." through "t" on the right-hand side are the "ASCII" forms of these bytes. If you don't know what this means, don't worry; we'll learn more about it later.

At this point, we are actually ready to run our program. This is done with the "G" or "go" command. Typing "G=nnnn,mmmm<cr>" runs the program beginning at address nnnn, stopping when it should execute the instruction stored at address mmmm. In our case, for example, the program begins at address 100, and the first instruction we *don't* want to be executed is at 118. Thus we can run the program with the command "G=100,118<cr>". Since the object of the program is to modify the memory locations 204 through 207, we can check to see if the program worked by examining those memory locations to see if they hold the correct values:

```
-d200,207
419F:0200  04 05 06 20 04 05 06 20  ... ..
```

Apparently the program works!

DEBUG has many other commands, but for now we will learn about just one other command, the "R" or "register" command. "R" may be used either to examine or to modify the 8088's registers. For example, type "R<cr>" displays the contents of all registers, as well as the value on top of the stack and the hexadecimal values and unassembled instruction at the address stored in the instruction pointer (IP) register:

```
-r
AX=0000 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=4410 ES=4410 SS=4410 CS=4410 IP=0100 NV UP DI PL NZ NA PO NC
4410:0100 005B9F  ADD      [BP+DI-61],BL      SS:FF9F=E8
```

In order to modify a register value, we type "Rrn<cr>", where "rn" is the name of a register (AX, BX, etc.). Then debug prompts for a new value to store in the register. For instance, to store 4567 (hex) in the CX register (which, from the display above is presently set to 0), we do:

```
-rcx
CX 0000
:4567
```

Such assignments can, of course, be easily checked with "R<cr>".

QUICK REFERENCE: MS-DOS, EDLIN, AND DEBUG**MS-DOS: FUNCTION KEYS**

|     |                           |     |                                |
|-----|---------------------------|-----|--------------------------------|
| ->  | move right                | F3  | move to end of line            |
| <-  | move left                 | ESC | move to beginning              |
| F2x | move right to character x | INS | toggle into/out of insert mode |
| DEL | delete character          |     |                                |
| F4x | delete to character x     | F5  | create new template            |

**MS-DOS: COMMANDS**

d: Change default drive to drive d:.

CLS Clear the screen.

COPY source [destination] [/V] Copy the source file to the specified destination. "Verify" the copy if "/V" is present. The destination can be a file or a disk drive. If the destination is absent, copy to default drive.

DEBUG Run the system debugging utility.

DIR [d:] Display the directory of drive d: (or default drive if absent).

EDLIN destination Edit the specified file.

ERASE destination Erase the specified file.

DISKCOPY s: d: [/V] Copy the entire disk in drive s: to drive d: (This erases the disk in drive d:.)

FORMAT d: /V Format (erase) the disk in drive d:.

LINK Run the system linker.

MASM Run the Macro Assembler.

RENAME source destination Renames the source file using the destination name.

TYPE source Display the contents of the specified file.

**NOTE:** Above, bracketed quantities are optional. *Source* and *destination* are filenames (unless specified), and *s:* and *d:* are disk-drive names.]

**THE TEXT EDITOR, EDLIN**

start,endD Delete the specified range of lines.

line Edit the specified line.

E End the editing session and save the text.

lineI Enter insert mode just prior to the specified line.

start,endL List the specified range of lines on screen.

Q End the editing session but do not save the text.

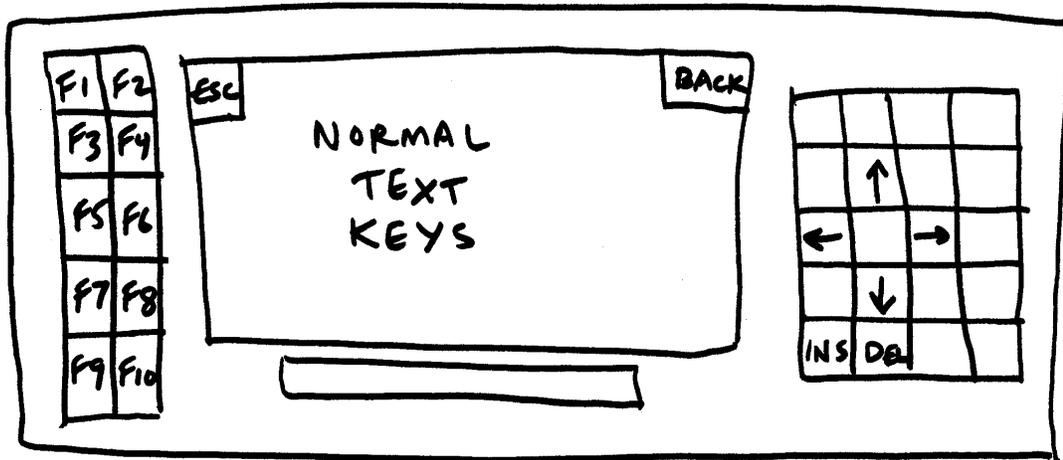
start,end[?]Rstring1F6string2 Replace (with prompting if ? is present) all occurrences of string1 with string2 in the specified range of lines.

start,endSstring Search the specified range of lines for an occurrence of the string.

**NOTE:** Above, *start*, *end*, and *line* refer to line-numbers. *String1*, *string2*, and *string* refer to strings. A "line number" can be either an actual number or the symbols "." (the current line) or "#" (the end of the text).

**DEBUG, THE SYSTEM DEBUGGING UTILITY**

Astart Input assembly language at the specified address.  
Dstart,end "Dump" (in hex) the specified address range.  
Estart Input (in hex) byte values at the specified address.  
G=start,end Execute the program in this address range.  
Q Quit and return to DOS.  
R[register] Display the values of all registers or else  
modify the specified register.  
Ustart,end Unassemble the specified locations in memory.  
**NOTE:** Above, *start* and *end* refer to addresses.

MS - DOS EDITING KEYS

IBM-PC KEYBOARD

|                |                                   |
|----------------|-----------------------------------|
| F1 OR →        | MOVE RIGHT                        |
| ← OR BACKSPACE | MOVE LEFT                         |
| DEL            | DELETE A CHARACTER                |
| INS            | TOGGLE IN/OUT OF<br>"INSERT" MODE |
| F3             | GOTO END OF LINE                  |
| ESC            | GOTO BEGINNING OF LINE            |
| F2x            | MOVE RIGHT TO CHARACTER x         |
| F4x            | DELETE TO CHARACTER x             |

## SAMPLE MS-DOS EDITING

SUPPOSE: LAST COMMAND TYPED WAS  
COPY A:FOO.BAR B:

-----  
(MS-DOS PRESENTS PROMPT)

A>\_

(USE F3 FUNCTION KEY)

A>COPY A:FOO.BAR B:\_

(BACKSPACE 12 TIMES)

A>COPY \_

(INS C: INS)

A>COPY C:\_

(2 → 's)

A>COPY C:A:\_

(F2R)

A>COPY C:A:FOO.BA\_

(BACKSPACE 8 TIMES)

A>COPY C:\_

(F4/ INS NIFTY.COM INS)

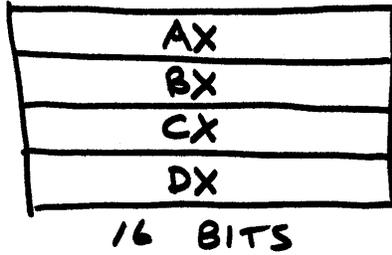
A>COPY C:NIFTY.COM\_

(F3)

A>COPY C:NIFTY.COM B:\_

REGISTER SET OF 8088 CPU

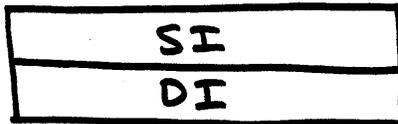
GENERAL REGISTERS :



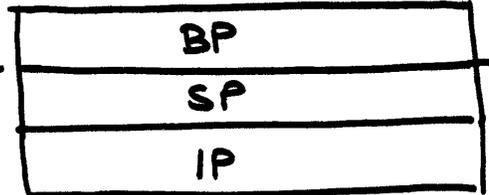
OR



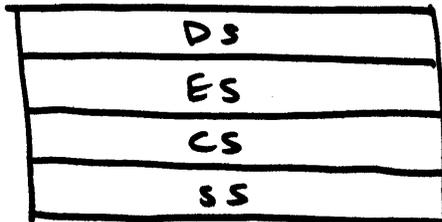
INDEX REGISTERS :



"POINTER" REGISTERS :



"SEGMENT" REGISTERS :



"FLAG" REGISTER :



"DEFINE DATA" OPERATORS

```

FOO DB 27 ; DEFINE BYTE VARIABLE
          ; NAMED "FOO", INITIALLY 27
BAR DW 3E1H ; WORD VARIABLE BAR,
          ; INITIALLY 3E1 (HEX)
REAL_FAT_RAT DD ? ; DOUBLEWORD VARIABLE,
          ; UNINITIALIZED .

```

THE MOV INSTRUCTION

```

MOV AX, BAR ; LOAD 16-BIT REGISTER AX WITH
              ; THE VALUE OF THE WORD VAR. BAR
MOV DL, FOO ; LOAD A BYTE FROM FOO TO DL
MOV BX, AX ; COPY BX REGISTER INTO AX
MOV BL, CH ; " CH " " BL
MOV BAR, SI ; SAVE SI REGISTER INTO VAR. BAR
MOV FOO, DH ; " DH " " " " FOO
MOV AX, 5 ; LOAD AX WITH THE VALUE 5
MOV AL, 5 ; " AL " " " " "
MOV BAR, 5 ; LET THE VARIABLE BAR BE SET TO 5
MOV FOO, 5 ; " " " " FOO " " " "

```

## OUR FIRST PROGRAM

### FORTRAN VERSION:

```

INTEGER*2 MY, NAME, IS, NOBODY
INTEGER*2 PLAY, MISTY, FOR, ME

```

⋮

```

PLAY = MY
MISTY = NAME
FOR = IS
ME = NOBODY

```

⋮

### ASSEMBLER:

; DESTINATION VARIABLES

```

PLAY    DB  ?
MISTY   DB  ?
FOR     DB  ?
ME      DB  ?

```

; SOURCE VARIABLES

```

MY      DB  4
NAME    DB  5
IS      DB  6
NOBODY  DB  32

```

⋮

```

MOV AX, MY ; PLAY = MY
MOV PLAY, AX
MOV AX, NAME ; MISTY = NAME
MOV MISTY, AX
MOV AX, IS ; FOR = IS
MOV FOR, AX
MOV AX, NOBODY ; ME = NOBODY
MOV ME, AX

```

⋮

RUNNING OUR PROGRAM WITH DEBUG

ENTERING THE PROGRAM

```
-a100
48EE:0100 mov ax, [200]
48EE:0103 mov [204], ax
48EE:0106 mov ax, [201]
48EE:0109 mov [205], ax
48EE:010C mov ax, [202]
48EE:010F mov [206], ax
48EE:0112 mov ax, [203]
48EE:0115 mov [207], ax
48EE:0118
```

CHECKING THE PROGRAM

```
-u100
48EE:0100 A10002      MOV      AX, [0200]
48EE:0103 A30402      MOV      [0204], AX
48EE:0106 A10102      MOV      AX, [0201]
48EE:0109 A30502      MOV      [0205], AX
48EE:010C A10202      MOV      AX, [0202]
48EE:010F A30602      MOV      [0206], AX
48EE:0112 A10302      MOV      AX, [0203]
48EE:0115 A30702      MOV      [0207], AX
48EE:0118 48          DEC      AX
48EE:0119 023C        ADD      BH, [SI]
48EE:011B 17          POP      SS
48EE:011C 7301        JNB     011F
48EE:011E C3          RET
48EE:011F E87600        CALL    0198
```

ENTERING THE DATA

```
-e200
419F:0200 77.4 20.5 64.6 69.20
```

CHECKING THE DATA

```
-d200,207
419F:0200 04 05 06 20 72 65 63 74 ... rect
```

RUNNING THE PROGRAM

```
-G=100,118
```

CHECKING THE RESULTS

```
-d200,207
419F:0200 04 05 06 20 04 05 06 20 ... ..
```

University of Texas at Dallas  
COURSE NOTES FOR CS-5330  
IBM PC ASSEMBLY LANGUAGE

**CLASS 3**Comments

1. On closer examination, the office hours 12-7 do seem a little extravagant. Therefore, my office hours will officially be Monday-Thursday, 3:00-7:00. However, I am usually here from noon (unless I go to lunch) and you are free to come any time you want to.

2. I'd like to comment on a couple of the points brought out in the "essays" I received as a result of the first assignment. First, Manning Grinnan tells us that the text editor called "PC-Write" is rather good *and* can be legally copied. *If* this is true, you can obtain a copy of PC-Write from him and use it rather than EDLIN (if you want to). Second, I mentioned last time that the 8088 family of microprocessors was introduced in 1978 and hence is rather ancient as far as the microcomputing world is concerned. This has led some of you to conclude that these processors are on their way out and that there would consequently be few job opportunities for 8088 assembly language programmers. Simple arithmetic, however, shows the opposite. At the present time, there is an installed base of nearly 2 million IBM PCs and XTs. This does not include the IBM ATs, or compatibles like Compaq or TI Professionals. It is likely that IBM will discontinue this line in the next year or two, but the 2 million already-installed PCs will not disappear for many years. Moreover, the machines which are expected to replace the PCs are machines like the IBM AT. The AT, though much faster than a PC, is compatible at the assembly-language level. Thus, there is likely to be a demand for IBM PC assembly language programmers for a number of years.

Review

In the last class we had our first real excursion into assembly language programming. In order to write and run a simple program, we:

Learned about the registers of the 8088. These registers, AX, BX, CX, DX, SI, DI, BP, etc., are all 16 bits in size, except that AX-DX can be considered to be 8 byte-sized registers called AL-DL and AH-DH. All of the registers have specialized functions, but AX-DX are commonly used for most data manipulations. AX is the 16-bit *accumulator* and AL is the 8-bit accumulator.

We learned about how to define "variables" in assembly language using the DB, DW, and DD operators. These operators allow us to symbolically refer to a variable by means of a *name*, and they give the variables *types* -- i.e., byte, word, or doubleword. They are also optionally capable of giving variables initial values. If we use the DEBUG program, however, which has its own built-in assembler, we cannot use such symbolic names. We must instead know the specific numerical address at which a variable is stored in memory.

We learned about the MOV instruction of the 8088 microprocessor. The instruction

MOV *destination, source*

copies the data specified by the source operand into the location specified by the destination operand. The destination and source can be either registers or memory locations, except that *no memory-to-memory* transfers are allowed. Moreover, the source operand can (in addition to registers and memory locations) be an *immediate* value.

We then wrote a short program which defined a number of byte-sized variables MY, NAME, IS, NOBODY, PLAY, MISTY, FOR, and ME. The first four of these were initialized to some known values (4, 5, 6, and 32) and the program copied these values to the second set of four variables.

We learned how to use the utility program DEBUG to run this program. In DEBUG, our program could be used just as written except that we had to use actual addresses for the variables rather than symbolic names. We stored our variables at locations 200 (hex) through 207. In DEBUG, we had to use expressions like [200], [201], etc. rather than MY, NAME, etc. We discussed the DEBUG commands

|                     |  |
|---------------------|--|
| <i>Aaddress</i>     | Input assembly language at the specified address.  |
| <i>Ustart, end</i>  | Display the assembly instructions stored in the specified address range.   |
| <i>Eaddress</i>     | Enter hexadecimal values to be stored beginning at the specified address.  |
| <i>Dstart, end</i>  | Display a hexadecimal "dump" of the bytes stored in the indicated range.   |
| <i>G=start, end</i> | Execute the program beginning at the start address. The end address is the address of the first instruction <i>not</i> executed. |
| <i>R[register]</i>  | Display the contents of (and optionally change) the contents of the registers.   |

#### More DEBUG Instructions

The DEBUG program works essentially with the *assembled* (i.e., executable) forms of programs. Even though you can type in assembly language instructions at the keyboard and get listings of the assembly language instructions stored in memory, DEBUG does not store or remember your source code. For example, if you enter the assembly language instruction

MOV AX, [200]

then DEBUG will instantly assemble this (producing the bytes A1, 00, and 02, incidentally) and store these bytes in memory. The original string of characters, "MOV AX, [200]", is completely lost. For the "U" DEBUG command, which lists the assembly language code at a given address, DEBUG actually looks at the bytes stored in memory (the A1, 00, and 02) and *deduces* the assembly language instructions that must have been typed in.

It is possible with DEBUG to store your program on the disk or to read from disk a program which is already there. However, since DEBUG really only recognizes executable code, it can only read or write files of executable code. It cannot, for example, read (and assemble) assembly language source code, nor can it save your program as a text file containing assembly language. [Actually, there are some MS-DOS tricks that can enable you to do these things, but they are not worth the effort.]

Saving a file with DEBUG is a slightly complex procedure. Loading a file from disk is somewhat easier. The first step in both is to inform DEBUG of the name of the file. This is done with the "N", or "name", command. This command has the syntax

*Nfilename*

Almost all files containing executable code have one of the following filename extensions: ".COM", ".EXE", or ".BIN". A program can be loaded into memory with the "L" command, which has the syntax

*L[address]*

The address at which the program is to be loaded can be (optionally) specified. If not, the program is loaded at address 100 (hex). In all file operations with DEBUG, the BX and CX registers are used together as a doubleword variable containing the number of bytes in the program. CX is the least significant word and BX (usually zero) is the most significant word. Therefore, after the file is loaded, DEBUG thoughtfully fixes these registers to contain a byte count. In any case, once the file is loaded, you may now examine or modify it using any of the DEBUG operations.

In order to save a file, as above you must use the "N" command to define the filename. (However, if you first loaded the file as described above, you don't need to use the "N" command again unless you want to change to a different filename.) You must also load the BX and CX registers with the byte count to be saved. This is done with the "R" command. For example, if your program was 200 (hex) bytes long, you would use "RBX" to load BX with zero and "RCX" to load CX with 200. Finally, the file can be saved with the "W" or "write" command. The syntax for this is

*W[address]*

where, as before, the starting address of the code to be saved is by default 100 but can optionally be explicitly entered.

### Arithmetic Instructions

It is gratifying that our little program worked (though full of errors), but unfortunately it doesn't do anything very interesting. Let us therefore learn about some more 8088 instructions. In this section, we will learn about some of the integer arithmetic operations, namely addition and subtraction. Integer multiplication and division instructions are also available, but we will discuss them later.

First we will consider the ADD and SUB instructions. These arithmetic instructions are quite similar to the MOV instruction in some ways. Like the MOV instruction, the general form of the add and subtract instructions is

*mnemonic destination,source*

In assembly language, a "mnemonic" is the symbolic name for a CPU instruction. For example, "MOV" is the mnemonic for the MOV instruction. For addition and subtraction the mnemonics are "ADD" and "SUB". Thus, an add instruction looks like

*ADD destination,source*

and a subtract instruction looks like

*SUB destination,source*

The ADD (SUB) instruction adds (subtracts) the value of the source operand to (from) the value of the destination operand, and stores the result in the destination. As with the MOV instruction, the source and the destination operands can be any combination of registers or memory variables, except that no memory-to-memory operations are allowed. Also, the source operand can be an immediate value rather than a register or memory variable. Also, as before, the sizes of the source and destination operands must match -- that is, they must both be bytes or both be words.

Here are some samples of valid additions and subtractions:

```
add dx,dx      ; add the DX register to itself.
add cx,5       ; add the value 5 to the cx reg.
add si,di      ; add the di register to si reg.
add bl,cl      ; add cl reg. to bl. reg.
add foo,5      ; add the value 5 to the
                ; variable foo.
add foo,al     ; add contents of al to foo.
sub bar,5      ; subtract word value 5 from bar
sub bar,3e1h   ; subtract 3e1h from variable bar
sub al,foo     ; subtract value of var. foo from al
sub si,ax      ; subtract contents of ax from si
```

Combinations such as

```
add cl,3e1h
add cl,bx
sub foo,cx
```

are illegal since the source and destination operands have incompatible sizes. (In each case listed, the destination is byte-sized and the source is word-sized.)

Recall our earlier sample program to copy the variables MY, NAME, IS, and NOBODY to the variables PLAY, MISTY, FOR, and ME. If, instead, we wanted to do the equivalent of

```

PLAY=MY+1
MISTY=NAME-1
FOR=IS+1
ME=NOBODY-1

```

we could modify our program in several ways to accomplish this. For example, we could simply copy the variables just as before, and then add or subtract one from the appropriate memory locations:

```

mov ax,my          ; PLAY=MY
mov play,ax
mov ax,name       ; MISTY=NAME
mov misty,ax
mov ax,is         ; FOR=IS
mov for,ax
mov ax,nobody    ; ME=NOBODY
mov me,ax
add play,1       ; PLAY=PLAY+1
sub misty,1     ; MISTY=MISTY-1
[or: add misty,-1]
add for,1       ; FOR=FOR+1
sub me,1       ; ME=ME-1

```

Another possibility is that since each value must pass through the intermediate register AX in this program, we could perform the additions or subtractions "on the fly":

```

mov ax,my          ; get MY into ax,
add ax,1          ; increment it,
mov play,ax       ; and save it as PLAY.
mov ax,name       ; similarly for MISTY=NAME-1
sub ax,1
mov misty,ax
mov ax,is         ; similarly for FOR=IS+1
add ax,1
mov for,ax
mov ax,nobody    ; similarly for ME=NOBODY-1
sub ax,1
mov me,ax

```

The latter approach is superior in that the assembled code is both smaller and executes faster than the former. We will find out later how to calculate such things, but for now we will note that

| <u>Instruction</u>    | <u>Clock Cycles</u> | <u>Assembled Size</u> |
|-----------------------|---------------------|-----------------------|
| MOV AX, memory        | 14                  | 3                     |
| MOV memory, AX        | 14                  | 3                     |
| ADD memory, immediate | 31                  | 6                     |
| SUB memory, immediate | 31                  | 6                     |
| ADD AX, immediate     | 4                   | 3                     |
| SUB AX, immediate     | 4                   | 3                     |

where a "clock cycle" is a unit of time approximately equal to 210 nanoseconds. A nanosecond is one billionth of a second. Thus the first program fragment executes in 236 clock cycles (about 50 microseconds) and uses 48 bytes when assembled, while the second

executes in 128 clock cycles (about 26 microseconds) and uses 36 bytes of memory.

Indeed, we can improve this slightly by using the "increment" and "decrement" instructions. These instructions have the form

```
INC destination
```

and

```
DEC destination
```

where the destination operand is a byte or word register or memory variable. INC adds one to the destination, while DEC subtracts one. Thus these instructions are functionally equivalent to

```
ADD destination,1
```

and

```
SUB destination,1
```

However, the increment and decrement instructions are actually smaller and execute more quickly. Thus, the following fragment is superior to both of the above:

```
mov ax,my          ; get MY into ax,
inc ax            ; increment it,
mov play,ax       ; and save it as PLAY.
mov ax,name       ; similarly for MISTY=NAME-1
dec ax
mov misty,ax
mov ax,is         ; similarly for FOR=IS+1
inc ax
mov for,ax
mov ax,nobody     ; similarly for ME=NOBODY-1
dec ax
mov me,ax
```

Actually, I was exaggerating slightly when I stated that the increment and decrement instructions were functionally equivalent to the instructions that immediately subtract or add one. There is, in fact, one difference between (for example) `ADD destination,1` and `INC destination`. This difference is the effect on the carry flag. We will discuss the so-called "flags" in the next section; for now, just think of the carry flag as a bit-sized register located somewhere in the CPU. As it happens, `ADD` and `SUB` instructions may modify the carry flag, but `INC` and `DEC` instructions do not.

What is the carry flag used for?

It can happen in integer addition that the result of an addition is too big for the destination address to hold. For example, if we have the word-values 8000H and 8001H stored in some word-sized source or destination the sum, 10001H, is 17 bits long and hence too large to store in the destination. It is easy to show that this in fact represents the worst possible case: that is, the result of a 16-bit

addition can never be more than 17 bits long. Similarly, the result of an 8-bit addition can never be more than 9 bits long. Thus, in practice, what the CPU does is to go ahead and store the 16 *least* significant bits of the result at the destination (or 8 least significant bits for a byte operation), and then store the significant 17th (9th) bit as the one-bit *carry flag*. This is very similar to what happens in pencil-and-paper addition. In adding (for instance) the numbers 7 and 8 on paper, we would "store" the least significant digit of the sum, 5, as part of the result, and would "carry" the most significant digit, 1.

Similarly, in subtraction with pencil and paper, we sometimes need to subtract a larger digit from a smaller digit. To do this, we "borrow" from the next higher digit. This "borrowed" digit is always zero or one, just like a carried digit.

In fact, the carry flag is used to store both carries and borrows in integer addition and subtraction. We will illustrate this point with 8-bit addition. Suppose that the registers AL, BL, and CL contain the (decimal) numbers 200, 195, and 25, respectively:

```
MOV AL,200
MOV BL,195
MOV CL,25
```

In the addition

```
ADD AL,BL
```

we would have the result 395, which is too big to store in the AL register. Thus, the carry flag would be "set" to one, and the result would be truncated to 8 bits: i.e., AL would contain 139. On the other hand, in the addition

```
ADD AL,CL
```

the result, 225 (<256) is byte sized, so we would find that AL contains 225 and the carry flag is "cleared" to zero. In the subtraction

```
SUB AL,BL
```

we are subtracting a smaller number from a bigger number, so AL register contains the result, 5, and the carry flag (which stores the "borrow") is cleared. In the subtraction

```
SUB BL,AL
```

however, we are subtracting a larger number from a smaller one, so the carry flag is set (indicating that a borrow has occurred). The result finally stored in BL is gotten by subtracting AL from BL *plus* 256, or 251. This is, of course, very similar to the way borrowing works with pencil and paper.

The carry flag is often used to check for *overflow* in integer operations. Another common use is in multiple-precision arithmetic. Since the 8088 has instructions for manipulating only byte (INTEGER\*1) and word (INTEGER\*2) values, it is necessary to write *programs* to deal

with arithmetic of higher precision: for example, doubleword (INTEGER\*4) arithmetic. To see the problems involved, let's see what we have to do to perform word arithmetic using only byte operations. For example, let's suppose we want to perform the equivalent of the operation

```
MOV AX,3E1H
MOV BX,736H
ADD AX,BX
```

using only byte additions. We *cannot* simply replace the "ADD AX,BX" instruction with the instructions

```
ADD AL,BL
ADD AH,BH
```

This is evident from the fact that this gives us the wrong answer:  $3E1+736=B27$ , but the latter instruction sequence gives us A27 instead. What has happened is that the addition of the least significant bytes (ADD AL,BL) overflowed and generated a carry. As in pencil-and-paper addition, where the carry is added in when we go to the next column, we needed to add the carry to the sum AH+BH. Thus, since we did not do this, the most significant byte of our result was one too small. We would have had similar problems in performing a subtraction.

Fortunately, we can take care of this problem by using the "add with carry" and "subtract with borrow" instructions of the 8088. These have the syntax

```
ADC destination,source
```

and

```
SBB destination,source
```

and are exactly like ADD and SUB except that ADC automatically adds in the carry left over from previous operations, and SBB automatically subtracts the borrow. In the example we are using, if we replaced "ADD AX,BX" by

```
ADD AL,BL
ADC AH,BH
```

we would find that our result was now correct since the carry is taken account of. This idea can be extended to higher precision arithmetic: The least significant bytes (or words) are ADDED, and all higher bytes (words) are ADCed. Similarly, in subtraction, the least significant bytes (words) are SUBed, and all higher bytes (words) are SBBed.

Let us write a short sample program illustrating this by performing an INTEGER\*8 addition on two variables stored in memory, placing the result in a third variable. There is no "define data" operator for declaring INTEGER\*8 variables, so we will declare our variables using 4 DW operators for each variable. Suppose that our variables are called "A","B", and "C". Our data declaration might look something like this:

```

; variable A
A   DW ?      ; least significant word of A
     DW ?      ; next more significant word
     DW ?      ; etc.
     DW ?
; variable B
B   DW ?,?,?,?
; variable C: the result of adding A and B
C   DW 4 DUP (?)

```

Several points are of interest here. One is that we notice in the declaration of A that the DW operator does not actually *require* a name; it is possible to use the DW (and DB and DD) operator to define variables that have no name. In the declaration of B we notice that we can use DW to define a number of word values in just one line of assembly code, so long as the individual word variables do not need to have names. In the declaration of C, we see that we can go even further (if we like) and, instead of entering 4 unknown words as "? , ? , ? , ? ", we can instruct the assembler to DUPLICATE the unknown value 4 times. This is very convenient if we happened to have (say) 1000 unknown (or known, but identical) values rather than just 4 of them. Another interesting point is that even though our variables are INTEGER\*8 in our minds, to the assembler they are only INTEGER\*2. Evidently, the interpretation of data structures in memory is more a personal option than a necessity of the CPU. Traditionally, the 808x family of processors stores data in memory with the least significant byte first, so we have adhered to tradition by putting the least significant word first.

The program to add A and B, giving C, could look something like this:

```

; program to add two INTEGER*8 values A and B to give C.
; First, add the least significant words:
    mov ax,A   ; get the least significant word of A
    add ax,B   ; add the least significant word of B
    mov C,ax   ; store the in the least sig. byte of C.
; Next, add the more significant words:
    mov ax,A+2 ; get the word at addr. A+2
    adc ax,B+2 ; add (with carry) the next word of B
    mov C+2,ax ; save it.
; Similarly for the next two bytes:
    mov ax,A+4
    adc ax,B+4
    mov C+4,ax
    mov ax,A+6
    adc ax,B+6
    mov C+6,ax

```

The only novel feature here is the use of expressions like B+4 to refer to the word four bytes past B. In general, such expressions retain the attributes of the symbols appearing in them: Thus, since B is a word variable, B+4 is also a word variable, but at a slightly different address. Note carefully that B+4 is an operation performed by the *assembler*, and its only effect is to calculate a new address; the instructions above *do not* add 2 or 4 or 6 to the values in AX, A, B, or C. If we were using DEBUG, of course, such expressions would be

meaningless since DEBUG doesn't recognize symbolic expressions. In DEBUG, we might define A to start at address 200, B at 208, and C at 210, so the expressions "A", "A+2", "A+4", and "A+6" would be replaced by things like "[200]", "[202]", "[204]", and "[206]".

### Flags

The carry flag is actually just one bit in the "flag" register of the 8088 CPU. There are a number of other flags in the flag register. Here is a complete list of the flags and other bits in the register (some of these won't be meaningful at the moment, but don't let that worry you):

#### FLAGS SET BY CPU OPERATIONS

- CF Carry Flag, which we've already encountered.
- PF Parity Flag. The parity flag is *set* if the number of non-zero bits in the result is even, and is *cleared* if odd.
- AF Half-carry Flag. Used in BCD arithmetic.
- ZF Zero Flag. Set to zero if the result is zero.
- SF Sign Flag. Equal to the highest bit in the result. In unsigned arithmetic this is not meaningful, but in signed arithmetic, the sign flag is set if and only if the result is negative.
- OF Overflow Flag. This flag is exactly like the carry flag, except that it is used for signed, rather than unsigned, arithmetic.

#### FLAGS SET BY THE USER

- TF Trace Flag. When this is set, it is possible in theory (i.e., with the proper software) to *single-step* the processor.
- IF Interrupt-enable Flag. We will discuss this flag further later.
- DF Direction Flag. Used in string operations and controls whether strings grow upwards in memory or downwards. Also discussed later.

For the present, only the flags in the first group are understandable and useful to us. Of those, all but the Parity Flag and Half Carry are constantly used.

The Overflow Flag deserves some further discussion. The Overflow Flag is the signed-arithmetic version of the Carry flag. The Carry Flag is used in unsigned arithmetic, and is set when unsigned arithmetic overflows. For example, in byte addition CF will be set if the result is greater than 255 or less than 0. For unsigned arithmetic, however, bytes represent values between -128 and +127, so results outside of this range will result in OF being set. (And similarly for word values.) For example, subtracting 3-5 would result in a carry (or actually a borrow) since the result, -2, is not in the range 0..255, but no overflow since -2 *is* in the range -128..127. Similarly, 100+100 would give an overflow but no carry.

The Overflow flag is used *only* to detect integer overflows, and *is not used* for multiple-precision signed arithmetic.

### Jumps and Conditional Jumps

The instructions we have considered so far are limited in that they allow only *linear* code: that is, code in which the instructions are executed sequentially according to their order in the PC's memory. However, for real programming we need to have a way of transferring control from one program location to another. We need to be able to choose which part of the computer's memory contains the program to be executed.

The control is accomplished with "jump" instructions. Jump instructions have the syntax

*mnemonic address*

The mnemonic here can be a number of different things, but for the moment, we will assume that it is "JMP". A JMP instruction "jumps" from the present location in memory (as indicated by the instruction pointer register IP) to the specified address in memory. In essence, JMP simply stores the given address in the IP register.

In DEBUG, the address operand is, of course, simply a number. For example, if we executed the instruction

```
JMP 121
```

then the very next instruction executed would be the instruction located at address 121 (hex). For the Macro Assembler (and for clarity), however, addresses are indicated by symbolic labels. A label is just like a variable name, except that it is followed by a colon. For example, in the program fragment

```

.
.
.
JMP FOOBAR
ADD AX,21
FOOBAR:
INC AX
.
.
.
```

"FOOBAR" is a label. The program also illustrates how labels are actually used to represent addresses in JMP instructions. Note that labels are actually quite different from variable names, in that labels represent *addresses*, while variables represent the *contents* of addresses.

JMP performs an *unconditional* jump: it always goes to the specified address, regardless of any special conditions that may obtain. There are also a series of *conditional* jump instructions which perform a jump only if some special condition *is* met. These instructions all have the general syntax given above, but their

mnemonics differ. With one exception, all conditional jumps occur only if some particular configuration of flag bits is set. Recall that flags are set or cleared depending on the results of the most recent arithmetic operations. Thus, conditional jumps are used to execute different segments of code if different arithmetic conditions obtain.

While we will eventually discuss all of the conditional jump instructions, here is a list of those conditional jumps that are pertinent to what we already know:

| Flag     | Jump<br>if set    | Jump<br>if not set |
|----------|-------------------|--------------------|
| Carry    | JC <i>address</i> | JNC <i>address</i> |
| Zero     | JZ <i>address</i> | JNZ <i>address</i> |
| Overflow | JO <i>address</i> | JNO <i>address</i> |
| Sign     | JS <i>address</i> | JNS <i>address</i> |
| Parity   | JP <i>address</i> | JNP <i>address</i> |

In addition, the instruction

*JCXZ address*

jumps to the specified address if the CX register is zero.

JMPs and conditional jumps differ in another way. JMPs may be used to transfer control to any address in memory. Conditional jumps, on the other hand, are *relative* jumps. A relative jump is one that, loosely speaking, must be within 128 bytes of the current address.

Let us see how these things work by writing a short sample program. Let us write a program which multiplies the AX register by ten, storing the result in the BX register. This program will perform repeated additions in order to perform the multiplication. A Pascal version of the program might appear like this:

```
var ax,bx,cx:integer;
begin
  bx:=0;
  for cx:=10 downto 1 do bx:=bx+ax
end;
```

We have used **downto** in the **for**-loop rather than **to** because in assembler loops which count *down* are much easier to implement than loops which count *up*.

Here is an assembler version of the program:

```
mov bx,0 ; the BX register holds the running sum
mov cx,10 ; loop ten times
again:
  jxcz done ; if cx has reached zero, stop
  add bx,ax ; BX:=BX+AX
  dec cx ; decrement the loop counter
  jmp again
done:
```

This is not a particularly good implementation, but it does illustrate how the JCXZ instruction is used. Here is a somewhat better job:

```

    mov bx,0 ; the BX register holds the running sum
    mov cl,10 ; this time, use CL as the loop counter
again:
    add bx,ax ; bx:=bx+ax
    dec cl ; decrement the loop counter
    jnz again ; repeat only if the loop counter isn't zero

```

Actually, neither version would be used in practice, since there is a built-in "loop" instruction that combines several of the instructions we have used here.

In DEBUG, the latter version would look like this:

```

-A100
4410:0100 MOV BX,0
4410:0103 MOV CL,10
4410:0105 ADD BX,AX
4410:0107 DEC CL
4410:0109 JNZ 105
4410:010B
-U100,10A
4410:0100 BB0000      MOV      BX,0000
4410:0103 B110      MOV      CL,10
4410:0105 01C3     ADD      BX,AX
4410:0107 FEC9     DEC      CL
4410:0109 75FA     JNZ     0105

```

**ASSIGNMENT:** Using DEBUG, write two simple assembler programs. The first program, which should begin at the address 0100 (hex) must read the byte at 0200, add 10, and store the result at 0201. Use DEBUG's D(ump) command to verify that the program operates correctly, and use the computer's "print screen" button to print the dump on the printer. The second program should subtract one from the number at location 0200 until reaching zero, then should store the number 65 at location 0202. Again, print a dump to demonstrate that this program worked. List both programs (not necessarily simultaneously) on the screen with DEBUG's U(nassemble) command, and print the screen to get a hardcopy listing of these programs.

```

; This is a sample program to perform multi-byte arithmetic.
; It adds the INTEGER*8 variables A and B, producing the
; result C.

```

```

; All of the lines in italics represent things that
; would be required in our program if we were using the
; Macro Assembler. As long as we are using DEBUG, however
; we can forget such stuff and just concentrate on the
; underlined material.

```

```

data      segment
a         dw      4 dup (?)      ; first operand
b         dw      4 dup (?)      ; second operand
c         dw      4 dup (?)      ; destination for result
data      ends

stacksegment  stack
             dw      100 dup (?)  ; alloc 100 words for stack
stackends

code        segment
             assume ds:data,cs:code

; main routine
beginproc   far
             push    ds           ; prepare return address
             mov     ax,0
             push    ax
             mov     ax,data      ; initialize ds register
             mov     ds,ax
             mov     ax,stack     ; initialize stack
             mov     ss,ax

             mov     ax,a         ; first, add the least significant
             add    ax,b         ; words of the two operands and
             mov    c,ax         ; store the result in c.

             mov     ax,a+2      ; do the same with the next bytes,
             adc    ax,b+2      ; but add in the carry as well.
             mov    c+2,ax

             mov     ax,a+4      ; etc.
             adc    ax,b+4
             mov    c+4,ax

             mov     ax,a+6      ; etc.
             adc    ax,b+6
             mov    c+6,ax

             ret                ; end of the program. go to DOS or DEBUG
beginendp

code       ends

end begin

```

ADD and SUB instruction

ADD destination , source

SUB destination , source

EXAMPLES :

[ USING      FOO DB ?  
              BAR DW ?           ]

ADD DX,DX ; ADD DX TO ITSELF  
 ADD CX,5 ; ~~ADD~~ ADD 5 TO CX  
 ADD SI,DI ; ADD DI REG. TO SI REG.  
 ADD BL,CL ; BYTE REG. ADDITION  
 ADD FOO,5 ; ADD 5 TO VAR. FOO  
 ADD FOO,AL ; ADD CONTENTS OF AL TO FOO

SUB BAR,5 ; SUBTRACT 5 FROM BAR

SUB BAR,3E1H

SUB AL,FOO ; SUBTRACT VAR. FOO  
   ; FROM AL

SUB SI,AX ; SUBTRACT WORD REGISTERS

NOT ALLOWED :

SUB FOO,FOO ; MEMORY TO MEMORY  
 ADD CL,3E1H } INCOMPATIBLE  
 ADD CL,BX    } SIZES .  
 SUB FOO,CX    }

SAMPLE PROGRAMFORTRAN:

```

PLAY = MY + 1
MISTY = NAME - 1

```

(PLAY, MISTY, MY, and NAME are INTEGER\*2)

ASSEMBLY:

(PLAY, MISTY, MY, and NAME are all defined as DW ? )

; ASSEMBLY LANGUAGE, VERSION 1

```

MOV AX, MY      }
MOV PLAY, AX    } ; PLAY = MY

```

```

MOV AX, NAME    }
MOV MISTY, AX   } ; MISTY = NAME

```

```

ADD PLAY, 1     ; PLAY = PLAY + 1

```

```

SUB MISTY, 1    ; MISTY = MISTY - 1

```

[Cor: ADD MISTY, -1 ]

; ASSEMBLY LANGUAGE, VERSION 2

```

MOV AX, MY      }
ADD AX, 1       } ; PLAY = MY + 1
MOV PLAY, AX    }

```

```

MOV AX, NAME    }
SUB AX, 1       } ; MISTY = NAME - 1
MOV MISTY, AXAX }

```

INC and DEC instructions

INC destination ; destination  $\leftarrow$  destination + 1

DEC destination ; destination  $\leftarrow$  destination - 1

EXAMPLES:      INC FOO  
                   INC AX  
                   etc.

TIMING AND MEMORY USE:

| <u>INSTRUCTION</u>                              | <u>CLOCK CYCLES</u> | <u>ASSEMBLED SIZE</u> |
|---|---------------------|-----------------------|
| MOV AX, memory                                  | 14                  | 3 BYTES               |
| MOV memory, AX                                  | 14                  | 3                     |
| ADD <del>AX</del> <sup>memory</sup> , immediate | 31                  | 6                     |
| SUB memory, immediate                           | 31                  | 6                     |
| ADD AX, immediate                               | 4                   | 3                     |
| SUB AX, immediate                               | 4                   | 3                     |
| INC AX  | 2                   | 1                     |
| DEC AX  | 2                   | 1                     |

[CLOCK CYCLE = 210 nanoseconds]

SAMPLE PROGRAMS :

VERSION 1: ~~12~~<sup>24</sup> BYTES, 118 CLOCKS

VERSION 2: 18 BYTES, 64 CLOCKS

VERSION 3 (USING "INC" AND "DEC" INSTEAD OF "ADD" AND "SUB"): 14 BYTES, 60 CLOCKS

FLAGS

|    |            |    |          |
|----|------------|----|----------|
| CF | Carry      | ZF | Zero     |
| PF | Parity     | SF | Sign     |
| AF | Half-carry | OF | Overflow |

JUMPS

JMP address ; Unconditional jump  
 JCXZ address ; Jump if CX = 0

| <u>Flag</u> | <u>Jump if set</u> | <u>Jump if not set</u> |
|-------------|--------------------|------------------------|
| Carry       | JC address         | JNC address            |
| Zero        | JZ address         | JNZ address            |
| Overflow    | JO address         | JNO address            |
| Sign        | JS address         | JNS address            |
| Parity      | JP address         | JNP address            |

SAMPLE PROGRAM

```

var ax, bx, cx : integer;
begin
  bx := 0;
  for cx := 10 downto 1 do bx := bx + ax
end;
-----
MOV BX, 0 ; THE BX REGISTER HOLDS RUNNING SUM
MOV CX, 10 ; CX IS LOOP VARIABLE
AGAIN:
JNZ DONE ; QUIT IF LOOP COUNTER ZERO
ADD BX, AX ; UPDATE THE SUM
DEC CX ; UPDATE LOOP COUNTER
JMP AGAIN
DONE:

```

University of Texas at Dallas  
COURSE NOTES FOR CS-5330  
IBM PC ASSEMBLY LANGUAGE

**CLASS 4**Comments

1. As mentioned last time, the word processing program called "PC-Write" can be obtained from Manning Grinnan by supplying a disk to him. This program can be used in place of EDLIN. PC-Write is distributed as "Shareware". Shareware can be copied freely and given away, but if you like the program the author requests a contribution (in this case, \$75). From the published descriptions of this program, it appears to be rather good.

2. Some of you have noticed that if, in DEBUG, you type a perfectly legal instruction of the form

```
mnemonic destination,source
```

you get an error message. For example,

```
MOV [202],65
```

gives an error. This happens because DEBUG is unable to determine whether this is a *byte* instruction or a *word* instruction. That is, DEBUG cannot tell if you are moving the byte value 65 to the byte-variable [202], or if you are moving the word value 65 to the word-variable [202]. We will also find similar cases later when using the Macro Assembler. In cases like

```
MOV AL,65
```

or

```
MOV [202],AL
```

DEBUG can deduce that 65 and [202] must refer to byte-values since AL is a byte register. Thus, we sometimes need a way to tell DEBUG that a quantity is a word or a byte. This can be done by appending the phrase "BYTE PTR" or "WORD PTR" to "[202]". For example, by typing

```
MOV BYTE PTR [202],65
```

in the byte case, and

```
MOV WORD PTR [202],65
```

in the word case, our error messages would disappear. "PTR" stands for "pointer", so these additional phrases mean that [202] "points" to a byte value or to a word value. In the Macro Assembler, "WORD PTR" and "BYTE PTR" can be used for this, but they can also be used to *override* a variable type. For example, if FOO is defined by

```
FOO DB ?
BAR DW ?
```

then the instruction

```
MOV AX,FOO
```

is illegal since AX is a word register and FOO is a byte variable. However, the assembler would allow

```
MOV AX,WORD PTR FOO
```

This instruction would load the value of FOO into the AL register, and the least significant byte of BAR into AH.

3. Somehow I appear to have managed to give the impression in the last class that the instruction

```
JCXZ address
```

is a *generic* instruction which works for all registers: i.e., that there are instructions like

```
JALZ address
JSIZ address
```

etc. This is not true, however. There is *only* a "JCXZ" instruction, plus other conditional jumps which depend on flag values.

4. The DEBUG program *does not* accept symbolic references to addresses. That is, you *cannot* use variable names and labels in DEBUG. In DEBUG, you must always explicitly (i.e., numerically) use the addresses you want. On the other hand, when we get to use the Macro Assembler, we will see that it usually accepts *only* symbolic addresses and that it is difficult to use actual numerical addresses with it.

#### Review

In the last class, we began learning about the arithmetic instructions of the 8088 CPU. We learned how to add and subtract bytes or words using the ADD and SUB instructions. These instructions have the syntax

```
mnemonic destination,source
```

which was identical to that of the MOV instruction. We learned that there were "increment" and "decrement" instructions

```
INC destination
```

and

```
DEC destination
```

which acted like

```
ADD destination,1
```

and

```
SUB destination,1
```

The latter instructions differed from the former only in that the *carry flag* was treated differently, and in that they required less time to execute and less memory to store.

We learned that the arithmetic operations affect the various flags of the CPU -- these being

```
CF   The Carry Flag
ZF   The Zero Flag
SF   The Sign Flag
OF   The Overflow Flag
PF   The Parity Flag
AF   The Half-carry Flag
```

The carry flag was particularly useful in multiple precision arithmetic. In adding (or subtracting) byte values, we found that it was actually possible to produce a 9-bit result, while adding word values could produce a 17-bit result. This extra bit is stored in the CPU as the carry flag. Multiple-precision arithmetic routines could take advantage of this by using the instructions

```
ADC destination,source
```

(or "ADd with Carry") and

```
SBB destination,source
```

(or "SuBtract with Borrow"). For example, we can subtract the AX register from the BX register with the word-subtraction instruction

```
SUB BX,AX
```

or with the byte instructions

```
SUB BL,AL
SBB BH,AH
```

Another use for the CPU flags was that they could be tested by the *conditional jump* instructions. An *unconditional jump* instruction was of the form

```
JMP address
```

and transferred program control unconditionally to the specified address. For example, in DEBUG, the instruction

```
JMP 200
```

would immediately start executing the instructions at address 200 (hex). This was a little trickier symbolically since we had to define symbolic names to memory locations. Labels had the same

characteristics as variable names, except that they are followed by a colon. For example, we could have in our program

```

      .
      .
      .
      JMP TEST_LABEL
      .
      .
      .
TEST_LABEL:
      .
      .
      .
    
```

Conditional jumps, on the other hand, jump to the specified address only if the appropriate flags are set. For example,

```
JC TEST_LABEL
```

would jump to location TEST-LABEL only if the carry flag is set to one. Here are the various conditional jumps we saw:

|          | Jump              | Jump               |
|----------|-------------------|--------------------|
| Flag     | if set            | if not set         |
| Carry    | JC <i>address</i> | JNC <i>address</i> |
| Zero     | JZ <i>address</i> | JNZ <i>address</i> |
| Sign     | JS <i>address</i> | JNS <i>address</i> |
| Overflow | JO <i>address</i> | JNO <i>address</i> |
| Parity   | JP <i>address</i> | JNP <i>address</i> |

We also saw that the conditional jump

```
JCXZ address
```

does not depend on any flag values, but will jump if the CX register is zero.

**ASSIGNMENT:** Read sections 3.1-3.3 of the textbook.

Addressing Modes

So far, we have talked about "sources" of data and "destinations" for data, and source and destination operands have always been of one of the following three types:

- IMMEDIATE* values were simply actual values, such as 5 or 200.
- REGISTERS* were the CPU registers, and were referred to by their names, like AX, BX, CX, DX, etc.
- DIRECT* addresses (though we did not refer to them by this name), were the *addresses* at which values were stored. These were referred to by their symbolic names in the Macro Assembler and by addresses in square brackets (like "[200]") in DEBUG.

These three ways of specifying sources and destinations are called "addressing modes" of the CPU. There are actually several more addressing modes, which we will discuss now. In general, each of these new addressing modes can be used in any situation where we can use direct addressing. With new addressing modes, we can add flexibility to the way memory is accessed.

*REGISTER INDIRECT.* In register indirect addressing, the *address* of the memory value is stored in a register, and we access the variable by giving the name of the register rather than the name of the variable. Let us consider an example of this. Suppose that we have the data declaration statements

```
FOO DB 100
BAR DW 3E1H
SAM DW 35
JANE DW 75
```

and that the *address* at which the variable BAR is stored is placed into the BX register of the CPU. If want to load the value of the variable BAR into the AX register, we can use either direct addressing,

```
MOV AX, BAR
```

or we can use register indirect addressing as follows

```
MOV AX, [BX]
```

In executing the first instruction, the CPU merely goes to the specified location and reads the value stored there (putting the result in AX). In the second instruction, the CPU reads the BX register, *interprets* that value as an address, and fetches the value stored at that address. (The second instruction, by the way, executes 1 clock cycle more quickly than the first, even though it may appear at first sight that the CPU is doing more work.)

This may be a little clearer if we think of using the DEBUG program. Suppose that the variable BAR is stored at address 200 (hex). Then the first instruction is just

```
MOV AX, [200]
```

To use the second instruction, however, the BX register must have been set up to contain the address of BAR -- say with the instruction

```
MOV BX, 200
```

Thus,

```
MOV BX, 200
MOV AX, [BX]
```

accomplishes the same thing as

```
MOV AX, [200]
```

except for the effect on the BX register.

It is a little more difficult to accomplish the operation "MOV BX,200" (that is, loading the address of BAR into BX) symbolically (if we are using the Macro Assembler), since we haven't learned yet how to compute the addresses of variables. That is, we have not yet learned how to store the *address* (as opposed to the value) of a variable into a register. This is accomplished with the "OFFSET" operator. The expression "OFFSET BAR" is the address of the BAR variable. Thus, the instruction

```
MOV BX,OFFSET BAR
```

loads the address of BAR into BX.

Here are some examples of register indirect addressing

```
MOV CX,[BX]           ; move the word value stored at the
                      ; address contained in BX to the CX
                      ; register.
ADD AL,[DI]           ; add (to AL) the value of the byte
                      ; stored at the address contained in
                      ; the DI register.
SUB [SI],DX           ; subtract the dx register's contents
                      ; from the word-value stored at the
                      ; address contained in the SI reg.
INC BYTE PTR [BX]     ; increment the byte stored
                      ; at the address pointed to by BX.
INC WORD PTR [BX]     ; ditto, but word value instead.
ADC WORD PTR [SI],65  ; add the immediate value 65
                      ; to the word value at the address
                      ; pointed to by SI.
```

Notice the use of "WORD PTR" and "BYTE PTR" in some of these instructions. These must be used in the cases shown because there is no *implicit* information in the instruction to allow the assembler to deduce whether an expression like "[BX]" refers to a byte stored in memory, or to a word.

In fact, *only* the BX, SI, and DI registers can be used to point to addresses in this way. (The BP register can also be used, but we will avoid using it until we discuss segmented memory.) Thus, instructions like

```
MOV [AX],DL
ADD AX,[DX]
```

are illegal.

Let us write a short program using register indirect addressing. Recall that in the last class we wrote a short program that performed INTEGER\*8 addition, adding the variables A and B to get C. A, B, and C were declared as

```

; EACH VARIABLE CONSISTS OF 4 WORDS.  THE LEAST SIGNIFICANT
; WORD IS STORED FIRST IN MEMORY, AND THE MOST SIGNIFICANT
; WORD IS LAST.
A    DW    4 DUP (?)
B    DW    4 DUP (?)
C    DW    4 DUP (?)

```

and the words making up each of these variables were arranged in order of increasing significance (i.e., least significant word first). Our previous program went something like this:

```

    MOV AX,A          ; ADD LEAST SIGNIFICANT WORDS.
    ADD AX,B
    MOV C,AX

    MOV AX,A+2        ; ADD (WITH CARRY) NEXT WORD
    ADC AX,B+2
    MOV C+2,AX

```

etc. (for next two words)

With register indirect addressing, we could write the program something like this, using the SI register to point to the words in A, DI to point to B, and BX to point to C:

```

; FIRST, SET UP POINTER REGISTERS
    MOV SI,OFFSET A    ; USE SI REGISTER TO POINT TO THE
                       ; WORDS IN A.
    MOV DI,OFFSET B    ; USE DI FOR B.
    MOV BX,OFFSET C    ; USE BX TO POINT TO C.

; NOW ADD LEAST SIGNIFICANT WORDS
    MOV AX,[SI]        ; GET LEAST SIG. WORD OF A
    ADD AX,[DI]        ; ADD TO LEAST SIG. WORD OF B
    MOV [BX],AX       ; STORE AT C

; NOW UPDATE POINTERS TO POINT TO NEXT WORDS OF A, B, AND C
    INC SI             ; WE HAVE TO INCREMENT SI TWICE, SINCE
    INC SI             ; ADD SI,2 WOULD DESTROY THE CARRY FLAG
    INC DI
    INC DI
    INC BX
    INC BX

; NOW ADD (WITH CARRY) THE NEXT MORE SIGNIFICANT WORDS:
    MOV AX,[SI]
    ADC AX,[DI]
    MOV [BX],AX

; NOW UPDATE POINTERS:
    etc.

```

Considering that the latter program contains a lot more code, is a lot more difficult to understand, and (for all we know) isn't any faster, why would we ever try to do this calculation using register indirect addressing? The reason is that the latter program can easily be fixed to work for any size integers, not just INTEGER\*8. This, in turn, can

be done because the latter program can be easily rewritten as a loop. The code used in the latter program is the *same* for each word added, except that the very first word uses an ADD and the other words use ADC. This is no problem, however, since we could simply use ADC each time and add an instruction to our program that clears the carry flag initially. Here is the way our program looks, written as a loop:

```
; FIRST, SET UP POINTER REGISTERS
    MOV SI,OFFSET A      ; USE SI REGISTER TO POINT TO THE
                        ; WORDS IN A.
    MOV DI,OFFSET B      ; USE DI FOR B.
    MOV BX,OFFSET C      ; USE BX TO POINT TO C.
    MOV CX,4             ; USE CX AS A LOOP COUNTER AND
                        ; LOOP ON 4 WORDS.
    CLC                  ; CLEAR THE CARRY FLAG

; NOW, LOOP

AGAIN:
    MOV AX,[SI]          ; FETCH ONE OF A'S WORDS.
    ADC AX,[DI]          ; ADD TO WORD OF B.
    MOV [BX],AX          ; STORE IN C.

; UPDATE THE POINTER REGISTERS TO THE NEXT WORDS IN A,B,C
    INC SI               ; WE HAVE TO INCREMENT SI TWICE, SINCE
    INC SI               ; ADD SI,2 WOULD DESTROY THE CARRY FLAG
    INC DI
    INC DI
    INC BX
    INC BX

    DEC CX               ; DECREMENT LOOP COUNTER
    JNZ AGAIN           ; IF LOOP COUNTER NOT ZERO YET, ; LOOP AGAIN
```

In this program, we can add integers of any precision just by putting the appropriate word-count in CX at the beginning of the program.

*BASE RELATIVE* and *DIRECT INDEXED* addressing do not actually differ from each other in practice (though possibly they differ from a logical standpoint), so we will not distinguish between them. In these addressing modes, the effective address of the data in memory is computed using one of the registers BX, DI, or SI (or BP) as above, but with an additional numerical offset added to the address. This is useful for a variety of applications, including addressing elements of one-dimensional arrays. The assembler and debugger (syntactically) accept a variety of forms for this addressing mode. For example, to load into AX the word 2 bytes *past* the address pointed to by the SI register, DEBUG will accept any of the following forms:

```
MOV AX,2[SI]
MOV AX,[SI]2
MOV AX,2+[SI]
MOV AX,[SI+2]
MOV AX,[SI]+2
```

However, two forms are most commonly encountered; these are

```
[base register + offset]
address[index register]
```

BX and BP are known as the *base registers*, while SI and DI are known as the *index registers*. Both "offset" and "address" represent numerical offsets, but the interpretations are different. In accessing the elements of a one-dimensional array, the first thing that appears in the forms above (i.e., either "base register" or "address") is usually interpreted to be the starting address of the array. The second part of the address (i.e., "offset" or "index register") is usually taken to be the relative position of the element in the array. For example, we think of FOO[SI] as representing the SI-th element of the array FOO, but we think of [BX+3] as representing the third element of the array pointed to by BX. (Assuming byte-sized array elements, of course).

In these addressing modes, if you use the name of a *variable* as the numeric offset, the assembler is clever enough to know that you are referring to the *address* of the variable rather than to its value. Moreover, the value actually referred to in memory is assumed to be of the same type as the variable whose name was used. That is FOO and FOO[SI] are both byte values in memory. For example, the operation

```
MOV AL,FOO
```

which is performed in register indirect addressing by

```
MOV BX,OFFSET FOO
MOV AX,[BX]
```

can be performed in direct indexed addressing by

```
MOV SI,0
MOV AX,FOO[SI]
```

This addressing mode is somewhat more convenient for our INTEGER\*8 addition program since with it we can get away with using less registers. Using direct indexed addressing our program might look like this:

```
; FIRST, SET UP POINTER REGISTER AND COUNTER
MOV SI,0          ; USE SI AS ARRAY INDEX
MOV CX,4          ; USE CX AS A LOOP COUNTER AND
                  ; LOOP ON 4 WORDS.
CLC               ; CLEAR THE CARRY FLAG

; NOW, LOOP

AGAIN:
MOV AX,A[SI]      ; FETCH ONE OF A'S WORDS.
ADC AX,B[SI]      ; ADD TO WORD OF B.
MOV C[SI],AX      ; STORE IN C.

INC SI           ; NEXT ELEMENT OF THE ARRAYS
INC SI

LOOP AGAIN
```

One new feature of this program is the use of the "LOOP" instruction. The instruction

```
LOOP address
```

is equivalent to the instructions

```
DEC CX
JCXZ address
```

and is very commonly used to control loops. It is slightly interesting to compare this program to the previous version, which used register indirect addressing. This version is very simple compared to the previous version, in terms of length and clarity of the source code. In terms of execution speed, register indirect operations are in general 4 clock cycles faster than direct indexed operations. However, by using direct indexed addressing we have been able to skip 2 pointer updates, each taking 4 clock cycles. Thus, each iteration of the loop requires 4 clock cycles (less than one microsecond) more in the second version of the program. Of course, the second version required less initialization of registers and, taking the entire program into account (not just the loop), the second version executes 8 clock cycles (about 1.6 microseconds) slower than the first.

*BASE INDEXED* addressing is like direct indexed addressing, except that the contents of *two* registers are combined with a numerical displacement to get the address of the data. In base indexed addressing, BX (or BP) is combined with either SI or DI. That is, base indexed addressing combines a base register and an index register and an offset. This addressing mode is useful for accessing the elements of two-dimensional arrays. As in direct indexed addressing, the assembler accepts a wide range of syntax: For example, each of the following loads AX with the word two bytes past the address consisting of the sum of the BX and DI registers:

```
MOV AX, [BX+2+DI]
MOV AX, [DI+BX+2]
MOV AX, [BX+2] [DI]
MOV AX, [BX] [DI+2]
MOV AX, 2 [BX] [DI]
```

For instance, if the registers BX and DI contain 3E1H and 47H, respectively, this instruction would load AX with the word at address 3E1H+47H+2H=42AH. Here are some other examples of valid base index addressing:

```
ADD AL, 2 [BX] [DI] ; ADD A BYTE TO AL
SUB 2 [BX] [DI], AX ; SUBTRACT AX FROM A WORD
INC WORD PTR 2 [BX] [DI] ; INCREMENT A WORD
DEC BYTE PTR 2 [BX] [DI] ; DECREMENT A BYTE
MOV BYTE PTR 2 [BX] [DI], 65 ; SAVE THE IMMEDIATE
; BYTE 65
MOV FOO [BX] [DI], 65 ; DITTO
```

As before, in some cases the assembler cannot figure out the size of the desired memory value (i.e., byte or word), so the specifiers "WORD PTR" and "BYTE PTR" are appended. In the final example, the variable

FOO is a byte value, so the assembler assumes that the immediate value is a byte also.

## ASCII

Up to now, we have thought of the values stored in various memory bytes as representing numbers -- i.e., unsigned bytes with values from 0 to 255 or signed bytes with values from -128 to 127 -- or as *parts* of other numbers (for example, the individual bytes of a word). However, the computer can also be used to manipulate *character* data, such as the text manipulated by the EDLIN program. Thus, a fourth interpretation of a byte is that it represents a character of text.

Characters are represented as bytes using ASCII, or the *American Standard Code for Information Interchange*. This is not related to the IBM mainframe EBCDIC character code standard. ASCII is almost universally used throughout computing and is summarized (in the form used on the IBM PC) on page 296 of the text. Let us briefly discuss ASCII.

The byte values 0-1F (hex) typically represent control codes that are used to pass information to I/O devices such as printers or terminals. The byte values 1-26 (decimal) are passed to the computer when you type a control character at the keyboard (that is, when you hold down the "ctrl" key and press one of the letters "A"- "Z"). Several of these codes also have their own dedicated keys: 8 is the backspace, 9 is the tab, and 13 is the carriage return. Of the control characters that *do not* have their own dedicated keys, the most interesting are: 7 (the "bell"), 10 (the "line feed"), and 12 (the "form feed"). Another useful byte value, 27 (decimal), is called the "escape" character and is created by pressing the "ESC" key. Often, programs (as well as I/O devices) use these codes to activate special functions. We will study later how these various codes are used. Under some circumstances, the values mentioned above are used to perform the functions described, while under other circumstances the IBM PC treats these codes as "graphics characters" that may or may not be available on other computers.

The code 20 (hex) represents a blank space, while the codes 30-39 (hex) represent the digits "0"- "9". The upper case letters "A"- "Z" are represented by the ASCII codes 41-5A (hex), while the lower-case letters are represented by the codes 61-7A. Most of remaining codes less than 80 (hex) are punctuation characters.

ASCII does not define the values of the codes above (or equal to) 80H. On the IBM PC, however, they are used to represent various graphics and special characters. Programs using these special characters cannot be counted upon to run on computers other than a legitimate, pedigreed IBM PC.

If we use the macro assembler (rather than DEBUG), literal characters, enclosed in quotes, can be used any place a byte constant can be used. For example, instead of

```
FOO DB 41H ; 41H is the ASCII code for capital A
```

we could have

```
FOO DB 'A'
```

Instead of

```
SUB AL,41H
```

we could have

```
SUB AL,'A'
```

Such substitutions are more meaningful than might be imagined. In the above example, if AL already contained one of the characters "A"- "Z", the subtraction performed there would convert the character to a number from 0 to 25.

As another example, consider a program to convert a two-digit hexadecimal number stored in DX in ASCII form into a byte stored in AL. We will suppose that the DH register contains the ASCII code of a character "0"- "9" or "A"- "F" representing the most significant digit, while the DL register contains the least significant digit. Since we haven't learned many of the 8088 instructions it would be convenient to use, this program won't be the best one we could possibly write.

```
; FIRST, LET'S PUT SOME CHARACTERS IN DX, SO THE PROGRAM
; WILL HAVE SOME DATA TO WORK ON:
    MOV DH,'7'
    MOV DL,'F'
; THAT IS, LET'S CONVERT THE NUMBER 7F (HEX) .

; THE PROGRAM.
; FIRST, WORK ON THE UPPER DIGIT:
    MOV AL,DH      ; GET THE DIGIT
    SUB AL,'A'     ; IS THE DIGIT BIGGER OR EQUAL TO "A"?
    JC NUMERIC_1  ; IF "0"- "9", JUMP TO ANOTHER ROUTINE
    ADD AL,10     ; IS "A"- "F", SO CONVERT TO 10-15
    JMP CONTINUE_1
NUMERIC_1:
    MOV AL,DH      ; GET THE DIGIT AGAIN
    SUB AL,'0'     ; CONVERT TO 0-9
; AT THIS POINT, AL CONTAINS A VALUE 0-15
CONTINUE_1:
    ADD AL,AL      ; DOUBLE AL
    ADD AL,AL      ; QUADRUPLE IT
    ADD AL,AL      ; OCTUPLE IT
    ADD AL,AL      ; NOW AL CONTAINS 16*UPPER DIGIT OF HEX
```

```
; NOW WORK ON SECOND DIGIT
  MOV AH,DL      ; GET THE DIGIT
  SUB AH,'A'     ; IS THE DIGIT BIGGER OR EQUAL TO "A"?
  JC NUMERIC_2   ; IF "0"- "9", JUMP TO ANOTHER ROUTINE
  ADD AH,10      ; IS "A"- "F", SO CONVERT TO 10-15
  JMP CONTINUE_2
NUMERIC_2:
  MOV AH,DL      ; GET THE DIGIT AGAIN
  SUB AH,'0'     ; CONVERT TO 0-9
; AT THIS POINT, AH CONTAINS A VALUE 0-15
CONTINUE_2:
  ADD AL,AH      ; ADD LOWER HEX DIGIT TO 16*UPPER
```

Upon running this program, the string "7F" stored in DX should be converted to the byte 7FH stored in AL.

ADDRESSING MODES OF THE 8088

**IMMEDIATE** — THE VALUE TO BE USED IS SPECIFIED.

EXAMPLE: MOV AX, 5

**DIRECT** — THE ADDRESS OF THE VALUE IS SPECIFIED.

EXAMPLES: MOV AX, BAR (MASM)  
MOV AX, [5] (DEBUG)

**REGISTER** — THE REGISTER CONTAINING THE VALUE IS SPECIFIED.

EXAMPLE: MOV AX, BX

---

**REGISTER INDIRECT** — THE REGISTER CONTAINING THE ADDRESS OF THE VALUE IS SPECIFIED.

EXAMPLE: MOV AX, [BX]

**DIRECT INDEXED** — A REGISTER AND NUMBER WHICH, COMBINED, GIVE THE ADDRESS OF THE VALUE ARE SPECIFIED.

EXAMPLES: MOV AX, BAR[SI] (MASM)  
MOV AX, 5[SI] (DEBUG)

**BASE INDEXED** — TWO REGISTERS AND A NUMBER, WHICH COMBINE TO AN ADDRESS, ARE SPECIFIED.

EXAMPLES: MOV AX, BAR[BX+SI] (MASM)  
MOV AX, 5[BX+SI] (DEBUG)

; This is the second version of the INTEGER\*8 addition  
; program, using register indirect addressing.

```

data    segment
a       dw      4 dup (?)      ; first operand
b       dw      4 dup (?)      ; second operand
c       dw      4 dup (?)      ; result
data    ends

stack   segment stack
        dw      100 dup (?)    ; space for stack
stack   ends

code    segment
        assume  cs:code,ds:data

begin   proc    far
        push   ds              ; prepare return address
        mov    ax,0
        push   ax
        mov    ax,data ; prepare DS register
        mov    ds,ax
        mov    ax,stack       ; prepare SS register
        mov    ss,ax

; first, set up the various pointer registers
        mov    si,offset a     ; si ==> a
        mov    di,offset b     ; di ==> b
        mov    bx,offset c     ; bx ==> c
        mov    cx,4           ; use cx to count words in loop
        cld                   ; clear the carry flag

; now, do loop on the 4 words in a, b, and c
again:  mov    ax,[si]         ; get a word from a
        adc    ax,[di]         ; add to word from b
        mov    [bx],ax        ; and save the result in c
; update the pointers to point to the next word
        inc    bx             ; INC twice instead of ADD BX,2
        inc    bx             ; since this won't affect the carry
        inc    si             ; flag.
        inc    si
        inc    di
        inc    di
; loop again, if not at end:
        dec    cx             ; update loop counter
        jnz   again

        ret
begin   endp

code    ends

        end    begin

```

; This is the third version of the INTEGER\*8 addition  
; program, using direct indexed addressing.

```

Data      segment
a         dw      4 dup (?)      ; first operand
b         dw      4 dup (?)      ; second operand
c         dw      4 dup (?)      ; result
data     ends

stack    segment stack
        dw      100 dup (?)      ; space for stack
stackends

code     segment
        assume  cs:code,ds:data

beginproc      far
        push    ds                ; prepare return address
        mov     ax,0
        push    ax
        mov     ax,data           ; prepare DS register
        mov     ds,ax
        mov     ax,stack         ; prepare SS register
        mov     ss,ax

; first, set up the various pointer registers
        mov     si,0              ; word index
        mov     cx,4              ; cx is the loop counter
        cld                       ; clear the carry flag

; now, loop:
again:    mov     ax,a[si]         ; get word of a
        adc     ax,b[si]         ; add word of b
        mov     c[si],ax         ; store in c
        inc     si                ; INC twice instead of ADD SI,2 to
        inc     si                ; avoid changing carry flag.
        loop   again             ; decrement cx, loop if not zero

        ret

beginendp

code     ends

        end      begin

```

**ASCII :**  
AMERICAN STANDARD CODE FOR INFORMATION INTERCHANGE

ASCII — A WAY OF INTERPRETING  
 CHARACTERS AS BYTES (AND VICE-VERSA)

ASCII CODES (BYTES) ↔ CHARACTERS

SOME COMMONLY USED ASCII CODES :

|                | <u>BYTE</u>                                     | <u>CHARACTER</u>  |
|----------------|---|---|
|                | 1-26 (DECIMAL)                                  | { "CONTROL CHARACTERS"<br>{ ctrl-A THROUGH ctrl-Z<br>(NOT PRINTABLE)  |
| FOR<br>EXAMPLE | {<br>7<br>8<br>9<br>10<br>12<br>13              | BELL<br>BACKSPACE<br>TAB<br>LINE FEED<br>FORM FEED<br>CARRIAGE RETURN |
|                | 27<br>32<br>30H - 39H<br>41H - 5AH<br>61H - 7AH | ESCAPE<br>SPACE<br>"0" - "9"<br>"A" - "Z"<br>"a" - "z"                |
|                | 80H - FFH                                       | { UNDEFINED BY<br>STANDARD — USED<br>FOR GRAPHICS, etc.               |

```

; PROGRAM TO CONVERT THE 2-DIGIT HEXADECIMAL NUMBER WHOSE
; ASCII CODES ARE STORED IN THE DX REGISTER (DL=LEAST
; SIGNIFICANT DIGIT, DH=MOST SIGNIFICANT DIGIT) INTO A
; BYTE VALUE IN THE AL REGISTER. THIS PROGRAM IS NOT
; WRITTEN VERY WELL BECAUSE WE HAVEN'T LEARNED ALL OF THE
; INSTRUCTIONS NEEDED TO DO IT WELL.

; FIRST, LET'S PUT SOME CHARACTERS IN DX, SO THE PROGRAM
; WILL HAVE SOME DATA TO WORK ON:
    MOV DH, '7'
    MOV DL, 'F'
; THAT IS, LET'S CONVERT THE NUMBER 7F (HEX) .

; THE PROGRAM.

; FIRST, WORK ON THE UPPER DIGIT:
; IF THE DIGIT IS "0"- "9", WE MUST CONVERT IT TO THE
; BYTE 0-9, ELSE IF THE DIGIT IS "A"- "F" WE MUST CONVERT
; IT TO THE BYTE 10-15. THEN, SINCE THIS IS THE MOST
; SIGNIFICANT DIGIT, IT MUST BE MULTIPLIED BY 16 AND STORED.
    MOV AL, DH      ; GET THE DIGIT
    SUB AL, 'A'     ; IS THE DIGIT BIGGER OR EQUAL TO "A"?
    JC NUMERIC_1   ; IF "0"- "9", JUMP TO ANOTHER ROUTINE
    ADD AL, 10     ; IS "A"- "F", SO CONVERT TO 10-15
    JMP CONTINUE_1

; THIS PART IS EXECUTED IF THE DIGIT IS "0"- "9"
NUMERIC_1:
    MOV AL, DH      ; GET THE DIGIT AGAIN
    SUB AL, '0'     ; CONVERT TO 0-9

; AT THIS POINT, AL CONTAINS A VALUE 0-15
CONTINUE_1:
    ADD AL, AL      ; DOUBLE AL
    ADD AL, AL      ; QUADRUPLE IT
    ADD AL, AL      ; OCTUPLE IT
    ADD AL, AL      ; NOW AL CONTAINS 16*UPPER DIGIT OF HEX

; NOW WORK ON THE LOWER DIGIT USING SAME TYPE OF ALGORITHM
    MOV AH, DL      ; GET THE DIGIT
    SUB AH, 'A'     ; IS THE DIGIT BIGGER OR EQUAL TO "A"?
    JC NUMERIC_2   ; IF "0"- "9", JUMP TO ANOTHER ROUTINE
    ADD AH, 10     ; IS "A"- "F", SO CONVERT TO 10-15
    JMP CONTINUE_2

; EXECUTE THIS ONLY IF LOWER DIGIT IS "0"- "9"
NUMERIC_2:
    MOV AH, DL      ; GET THE DIGIT AGAIN
    SUB AH, '0'     ; CONVERT TO 0-9

; AT THIS POINT, AH CONTAINS A VALUE 0-15
CONTINUE_2:
    ADD AL, AH      ; ADD LOWER HEX DIGIT TO 16*UPPER

```

University of Texas at Dallas  
 COURSE NOTES FOR CS-5330  
 IBM PC ASSEMBLY LANGUAGE

## CLASS 5

Review

We learned about several specifying phrases that can be used in source or destination operands to modify their use. The "WORD PTR" specifier changes the *type* of a variable to WORD. For example, in an instruction like

```
INC WORD PTR FOO
```

the variable FOO, which we normally declare to be a BYTE, is used as if it was a WORD. Similarly, "BYTE PTR" is used to force a variable to be of type BYTE in a given instruction. The phrase "OFFSET" specifies that the *address* of the variable should be used, rather than its value. For example,

```
MOV BX,OFFSET FOO
```

loads the BX register with the address of the variable FOO, rather than with its value (which would not be allowed anyway since FOO is a BYTE).

Most of the previous class was spent discussing the remaining addressing modes of the 8088 microprocessor. Up until that class we had dealt exclusively with the *immediate* addressing mode, the *direct* addressing mode, and the *register* addressing mode.

The new addressing modes learned were the *register indirect mode*, the *direct indexed mode* (which is divided in the book into the two modes called *direct indexed* and *base relative*), and the *base indexed* mode. Here is a summary of all of the addressing modes:

|                   |   |
|-------------------|---|
| IMMEDIATE         | the instruction specifies the actual <u>value</u> to be used in the operation.  |
| DIRECT            | the instruction specifies the <u>address of the value</u> to be used.   |
| REGISTER          | the instruction specifies the <u>register containing the value</u> to be used.  |
| REGISTER INDIRECT | the instruction specifies the <u>register containing the address of the value</u> to be used.   |
| DIRECT INDEXED    | the instruction specifies a <u>register</u> and a <u>number</u> which, when combined, give the <u>address of the value</u> to be used.    |
| BASE INDEXED      | the instruction specifies <u>two registers</u> and a <u>number</u> which, when combined, give the <u>address of the value</u> to be used. |

We found essentially, that each of these modes may be used in any situation where any other mode can be used. There are some exceptions, of course. An immediate value cannot be the destination operand of an instruction, for example.

We also discussed the ASCII correspondence between bytes and characters. We found that each character corresponds to some byte value (the ASCII code of the character), but that not all ASCII codes correspond to printable characters. Some ASCII codes, for example, correspond to non-printable "control characters".

Finally, we learned (in passing) two new instructions. These were the instructions

```
CLC          ; clear carry flag
LOOP address ; same as DEC CX/JCXZ address
```

### System Calls

So far, we have concentrated mostly on 8088 assembly language itself, with very little reference to the actual operating system being used. That is, while we have used some programs (DEBUG, EDLIN) which are available only for the MS-DOS operating system, the programs we have written for ourselves have been "generic" in nature and could run regardless of the operating system used.

However, as we have been discovering, the types of operations which the microprocessor can perform are very primitive in nature -- data movement, integer arithmetic, etc. There is no built-in way of reading the keyboard, writing to the crt, accessing disk-files, and so forth. These facilities are provided instead by the operating system, and the way they are provided differs from operating system to operating system. In this class, we will study the I/O features provided by the MS-DOS operating system, since this is the DOS used by the overwhelming majority of IBM PCs and compatibles. Thus, the programs we write using the MS-DOS functions should still run on the TI Professionals and on other compatibles. Later in the semester, we will see how to do some kinds of I/O directly through the hardware, bypassing the operating system entirely. When we finally reach that point, we will discover some incompatibilities between (for example) IBM PCs and TIs. However, that is still in the future.

Most DOS functions are performed by executing the instruction

```
INT 21H    ; call DOS
```

DOS provides on the order of 40 different services that can be activated in this way. You select between the various services (which are summarized -- with some misprints -- on pages 218-223 of the text) by placing a number in the AH register describing the desired function. You may also need to place additional arguments in other registers. Some services return values in registers or in memory.

**(NOTE:** anybody familiar with the CP/M operating system will notice that the first 30 or 35 functions mentioned on pp. 218-223 are very similar to the CP/M "BDOS" calls. This happens because MS-DOS is designed to emulate CP/M and, in particular, to allow automated

translation of CP/M assembly language source code to MS-DOS assembly language source code.)

For now, almost all of these DOS services do not concern us. We will at first just use a few of the simplest functions, like writing to the screen or reading from the keyboard.

The simplest DOS service is function number 1, which reads a keyboard character. When a key is pressed on the computer's keyboard, the ASCII code of the selected character is "read" by the operating system and is presented to the user program when DOS function number 1 is executed. Here is how to use DOS function 1:

```
MOV  AH,1      ; SELECT FUNCTION NUMBER 1
INT  21H      ; CALL MS-DOS
```

When this program fragment is executed, DOS takes control of the computer away from your program and simply waits for a key to be pressed at the keyboard. The ASCII code for this character is returned in the AL register. For example, if the character "A" was typed, we would find the ASCII code 41H in the AL register. Function 1 displays the character on the screen as it is typed (it *echoes* the character). In some cases this isn't desirable, and we would prefer to just read the keyboard character without any screen echo. This is accomplished with DOS function 8, which is otherwise just like function 1:

```
MOV  AH,8      ; PREPARE TO READ KBD WITHOUT ECHO
INT  21H      ; CALL MS-DOS
```

Another commonly used DOS service is function number 2, which displays a character on the screen. This is used much like function number 1, except that before using it the character to be displayed must be put into the DL register. For example, to print an "A" we would

```
MOV  DL,'A'    ; PREPARE THE CHARACTER TO BE PRINTED
MOV  AH,2      ; SELECT DOS FUNCTION NUMBER 2
INT  21H      ; CALL MS-DOS
```

A similar facility is provided by DOS function number 5. With DOS function 5, the character in DL is printed on the printer rather than displayed on the screen.

Here is a simple program which uses some of these functions: it is a very simple "typewriter", that allows you to type at the keyboard and see what you type displayed on the screen. As an added convenience, we will also arrange things so that you can exit the program by pressing the ESC key:

AGAIN:

```
MOV  AH,1      ; FIRST, GET (AND ECHO) A CHARACTER
                    ; USING FUNCTION 1
INT  21H      ; CALL DOS
CMP  AL,27    ; IS THE CHARACTER <ESC>?
JNZ  AGAIN    ; IF YES, QUIT. IF NOT, GET ANOTHER.
```

A similar program using DOS function 8 instead of function 1 would need to explicitly display the character since function 8 does not echo:

AGAIN:

```

MOV  AH,8           ; GET KBD CHARACTER USING DOS FUNC. 8
INT  21H           ; CALL MS-DOS
CMP  AL,27         ; <ESC> CHARACTER?
JZ   DONE          ; IF YES, THEN QUIT
MOV  DL,AL         ; IF NOT, PREPARE TO DISPLAY IT
MOV  AH,2          ; SELECT THE SCREEN-DISPLAY FUNCTION
INT  21H           ; CALL DOS
JMP  AGAIN         ; AND REPEAT ...

```

DONE:

One interesting point about this program is the use of the "CMP" or "compare" instruction. CMP has a syntax identical to the SUB instruction, of which it is a variation. CMP sets or resets the status flags (CF, ZF, SF, etc.) just as if a subtraction has been performed, but does not modify the destination operand in any way. In this case, AL contains the keyboard character which we want to test against the ASCII "escape" code, 27. These two are equal if and only if we get zero when subtracting them -- that is, if and only if the Zero Flag is set after a subtraction. However, a subtraction would change the value of AL, which we don't want. Thus, CMP is exactly what we need to perform this test since it sets the flags properly but does not change AL.

In using the above program, it is interesting to note that when you press the carriage return key, the cursor goes to the beginning of the line, but does not advance to the next line. The latter is the function of the "line feed". Every line which is printed or displayed in MS-DOS is terminated with a carriage return, followed by a line feed -- i.e., the ASCII codes 13 and 10. This effect can also be seen when we use DOS function 9, the "display string" function. Unlike function 2, which can just display one character at a time, function 9 can display an entire string of characters with just one invocation. In order to use function 9, we must first store the starting address of the string in the DX register. The string is simply a sequence of ASCII codes stored someplace in memory. The string is terminated with a "\$" character. A typical use for function 9 might be something like this:

```

MESSAGE DB 'Hello, I am a sample program!',13,10,'$'
      .
      .
      .
MOV  DX,OFFSET MESSAGE ; LOAD UP ADDRESS OF STRING
MOV  AH,9
INT  21H

```

As you might expect, executing this results in the message "Hello, I am a sample program!" appearing on the CRT. Several features of this tiny program are worth commenting on. First, we see once again the ability of the DB and DW operators to store *more than one* byte or word. In this case, the DB statement is equivalent to

```

MESSAGE DB 'H'
         DB 'e'
         DB 'l'
         .
         .
         .
         DB '$'

```

Second, we observe that the carriage return and line feed codes 13 and 10 have been used to move the cursor to a new line after displaying the string. Otherwise, the cursor would simply have sat foolishly at the end of the message on the screen. Third, we see once again the use of the "OFFSET" operator to give the address of a variable rather than the value of a variable.

Of the many other DOS functions available, the most important ones are probably those that allow the assembly program to read or write disk files. We will discuss these DOS functions later when we are in a better position to use them.

### Segments

As we know, the address space of the 8088 microprocessor is 1 megabyte in size. However, this address space is not *continuous*: rather, it is divided into *segments* of size 64K. Each segment, on the other hand, does have continuous addresses, ranging from 0 to 65535.

Segments cannot begin at arbitrary places in memory; they can only begin at addresses that are multiples of 16. Therefore, when the addresses of segments are referred to, they are always divided by 16. For example, segment 0 begins at address 0, segment 1 begins at address 16, segment 2 begins at address 32, etc. Thus, we can have segment numbers ranging from 0 to 65535. Both segment numbers and addresses within segments are word values requiring 16 bits for their specification.

Locations in memory are seldom specified by their actual addresses. They are almost always specified by giving their segment number and their address within the segment. For example, rather than discussing the memory location 8000H, we might discuss the location "8000:0000". The meaning of something like "8000:0000" is that we are referring to address 0 *within* the segment 8000H. In an expression like this, the number before the colon is the "segment" and the number after it is called the "offset". This explains the use of the OFFSET operator to get the addresses of variables in assembly language instructions like

```
MOV SI,OFFSET FOO
```

These *segment:offset* pairs are not uniquely defined. That is, while we can uniquely determine a memory location once we are given a *segment:offset* pair (by means of the formula  $address = 16 * segment + offset$ ), we cannot determine the *segment* and the *offset* if we are simply given the address. In fact, *many different segment:offset* pairs describe the same address. In the example above, all of the following refer to address 8000H:

8000:0000, 7FFF:0010, 7FFE:0020, etc.

Since the *segment* and the *offset* are word values, they may be specified by the CPU's 16 bit registers rather than by actual numbers. For example, we might (intellectually, at least) have a *segment:offset* pair like 8000:DX or SI:DI. Actually, these particular combinations are not allowed. In fact, for most purposes, the CPU does not even allow you to specify combinations like 8000:0000.

In the 8088 microprocessor, *segments* almost always have to be specified by one of the so-called *segment registers* CS, DS, ES, or SS, while offsets are specified by one of the memory addressing modes (i.e., any addressing mode except immediate or register). Thus, we typically see *segment:offset* specifications like

```
DS:FOO
ES:[SI]
CS:5
etc.
```

In accessing data, the 8088 combines the data address specified in the instruction with a segment register, thus getting the full address of the data. The 8088 has default segment register choices which it uses in certain types of memory accesses. For the instruction types we have learned about, the rule is this:

The DS segment register is always used by default *unless* the BP base register is used in indirect addressing.

If the BP register is used in indirect addressing, the SS segment register is the default. Thus, for example, we have the following equivalent forms of instructions:

```
MOV AX,BAR                MOV AX,DS:BAR
MOV [SI],AL              MOV DS:[SI],AL
ADC WORD PTR [DI],65     MOV WORD PTR DS:[DI],65
MOV DL,5[BP]            MOV DL,SS:5[BP]
MOV DL,5[BP][SI]       MOV DL,SS:5[BP][SI]
```

We see that data for programs usually resides in the *Data Segment* (i.e., the segment pointed to by the DS register), or in the *Stack Segment* (pointed to by SS). In most cases, but not all, these default segment register choices can be overridden by explicitly specifying a new segment register:

```
MOV AX,ES:BAR
MOV ES:[SI],AL
ADC WORD PTR ES:[DI],65
MOV DL,ES:5[BP]
MOV DL,ES:5[BP][SI]
```

all access data stored in the *Extra Segment*.

Some defaults cannot be overridden. All program code is contained in the *Code Segment*, pointed to by the CS register. All stack operations, which we discuss later, access the Stack Segment, pointed

to by SS. String operations use the Data Segment as their sources and the Extra Segment as their destinations. For flexible programming, therefore, our programs must *change* the values of the segment registers -- i.e., the locations of the segments. For the simple programs we will write at first, though, the segment registers can simply be set properly at the beginning of the program and then be forgotten.

We have been so successful in using the DEBUG program to run our own programs partially because DEBUG allows us to forget these details about segments. DEBUG automatically assigns all of the segment registers CS, DS, ES, and SS so that they (initially) contain the same values. Because of this, we can write our programs correctly just by assuming that the address space of the CPU is 64K in size and that the *offsets* of the addresses are actually the entire address. We will see shortly that the situation is somewhat more complex when we begin using the Macro Assembler.

#### Pseudo-Ops and the Format of .ASM Programs

Programming with DEBUG is easy since we just type in our program and our variables, and can execute our program immediately. With the Macro Assembler, however, which we now begin to use, there are many additional programming details which must be taken care of before we can even assemble our program. These details are really only incidental since they are much the same for every program and have very little to do with the algorithm employed or the function performed. Indeed, for most of our programs, we can simply employ a "pattern" or "template" program which takes care of all these details, but which has places where we can insert our own variables or source code.

(See handout.)

Most of the template is devoted to proper allocation of memory segments. For example, the lines of the program containing the words SEGMENT or ENDS tell the assembler where memory segments begin and end. The parts of the program bounded by the lines

```
DATA SEGMENT
DATA ENDS
```

tell the assembler where the segment *called* "DATA" begins and ends. The word "DATA" in this case is simply a name and does not imply that this is the *Data Segment*, pointed to by the DS register. As it happens, for clarity we have chosen DATA to be the Data Segment in this particular sample program -- but it doesn't have to be that way in general. Assuming, however, that the segment called "DATA" is the Data Segment, the ASSUME pseudo-op is used to inform the assembler of this fact. In our program, the assembler is told to assume that "DATA" is the name of the Data Segment and "CODE" is the name of the Code Segment. This information helps the assembler compute the addresses (actually, the offsets) of variables and labels, but the ASSUME pseudo-op does not actually assign the proper values to the segment registers. Thus, we must supply additional code to perform this housekeeping detail. The EQU pseudo-op is used to define "constants". This is useful if there is some value which is constant throughout the program, but which we may wish to change at some later time. For example, in our template, the "size of the stack" is defined by a constant called

"STACKSIZE". The size of the stack never changes in the program, but may be decreased or increased by the programmer the next time the program is assembled: therefore, it is usefully defined to be a constant.

We will not discuss these housekeeping details further at this time. Let it suffice for the present that our template program can be used to build up real programs, whether or not we understand why. As we become more familiar with assembly language, and in particular with memory allocation, the necessity of some of these "features" of the assembler will become clear to us (or, at any rate, clearer than they are now).

**ASSIGNMENT:**

1. Read Chapter 2 *through* page 38, plus section 2.7.
2. Write a program using EDLIN and run it using the Macro Assembler. You can use the "template" program and simply insert your own code and variables in the spaces provided. This program should:
  - a) Read the keyboard using a DOS function.
  - b) Exit from the program if the escape key is pressed.
  - c) Otherwise, convert the character to upper case.
  - d) Display the upper case character on the screen using a DOS function.
  - e) Go back to step "a".

You will turn in a *disk* (no paper!) containing the source code (call it "PROG3.ASM") and the executable code (PROG3.EXE) of the program. Write your name on the disk-label with a *felt-tipped* pen, and embed your name in the program (in a comment). I will run it to make sure that it works. (And you had better not *crash* my system, either!) Please use only the instructions we have discussed in class or that have been in the assigned reading.

3. Write a second program which is similar but more complex. Again, turn in the ASM file (PROG4.ASM) and the EXE file (PROG4.EXE), on the same disk as PROG3. This program should:
  - a) Using a DOS function (or a routine of your own devising), display a sign-on message (such as, "CS-5330 Program 4, by John Doe").
  - b) Input lines of text, as follows:
    - i) Display a prompt at the beginning of the line
    - ii) Input a character from the keyboard using a DOS function.
    - iii) If the character is an <ESC>, go to step "c".
    - iv) If the character is *printable* (' ' through ASCII code 7EH), display the character on the screen by using a DOS function and put it in a *text buffer* in memory.
    - v) If the character is a backspace, display it and remove the previous character from the text buffer. However, do not backspace past the physical beginning of the line.
    - vi) If the character is a carriage return, display a carriage return *and* a line feed, putting both of these characters in the buffer. Then, go to the beginning of step "b".
    - vii) Ignore any other non-printable characters.

- viii) Go back to step "ii".
- c) Print the entire buffer on the printer.
- d) Return to DOS.

In addition to this, the program must check for *buffer overflow*: that is, it must not allow any characters to be stored in the buffer once it is full. Therefore, you must provide some appropriate error routine to handle this case. Because this is our first long program, I have posted a sample solution to the problem on the door of my office and you are free to look at this if you get stuck. **NOTE:** I wrote this program myself (I didn't get it out of any book), so if any *substantial* portion of my code appears in the middle of your program I'll know (and I'll make you do it over again)!

#### Running the Assembler and the Linker

The current reading assignment in the textbook contains descriptions of how to run the linker and the assembler, so it would not do for me to spend too much time on this. However, to a certain extent I can cut through the mumbo-jumbo in the book and summarize the use of these programs as follows:

Suppose that you have written an assembly language source file, and that this file is called "SOURCE.ASM" and is stored on drive B:. In order to assemble it, you must put a disk containing the assembler (in our case, the Macro Assembler -- MASM -- rather than the small assembler as mentioned in the book) and type

```
B>A:MASM SOURCE;
```

Notice that you do not need to type the ".ASM" extension of your source filename since this is the default for the assembler. The semi-colon at the end of the line means: just skip all of the prompts which the assembler would otherwise give and which are shown in the book. This syntax will not create a "listing file" (as described in the book), so if you want to do that you'd better follow the book's instructions.

After the assembler runs, assuming that there were no errors, your B: disk will contain, among other things, a new file called "SOURCE.OBJ". The file must now be "linked" using the linker program. To do this, put a disk containing the linker program in drive A: and type

```
B>A:LINK SOURCE;
```

The linker will then execute and, if there are no errors, produce a new file on the B: disk called "SOURCE.EXE". The program may now be executed by typing

```
B>SOURCE
```

or may be debugged with DEBUG (as described in section 2.7 of the text) by typing

```
B>A:DEBUG SOURCE.EXE
```

As should be clear, you will save yourself a lot of time flipping disks if you make just one disk containing DEBUG, MASM, LINK, and EDLIN and use this disk in the A: drive.

### Logical Operations

For a little relaxation, let us discuss the simple "logical" operation instructions supplied by the 8088 CPU. We have already seen how to perform simple integer arithmetic using the 8088, and logical arithmetic turns out to be very similar.

The most primitive instruction is the

*NOT destination*

instruction, which takes the one's complement (or logical negation) of the destination operand. The destination can be either a byte or a word, whereas we know that logical operations really apply to two-valued variables. How, then, can "logical" operations be performed on bytes, which have 256 possible values? The answer is that many of the 8088's logical operations work "bitwise". Each bit of the destination operand is taken to be a two-valued logical operator and is acted on separately. As a simple example with the NOT instruction, if we had the code

```
MOV  AL,0
NOT  AL
```

we would find that the resulting AL value would have all of its bits set (as opposed to the initial value, with all bits clear). The value 0 is typically used in many programming languages to represent the logical value .FALSE., so the result of this negation should be the value which programming languages typically use to mean .TRUE. In fact, for our example, the result is a signed value of -1 (or, unsigned 255) which is, indeed, typical.

There are four bitwise logical instructions with the syntax

*mnemonic destination,source*

The AND instruction performs a bitwise logical "and" of the source operand to the destination operand. The most common use of this instruction is probably to "mask out" certain bits of the destination. This happens because any bit anded with a set bit is unchanged, but any bit anded with a clear bit is cleared. For example,

```
AND  AL,0FH
```

would reset the four highest bits of AL, but leave the four lowest bits unchanged.

The OR instruction performs a bitwise logical "or" of the source to the destination operands. This instruction is commonly used to set selected bits of the destination, since bits or-ed with 0 are unchanged, but bits or-ed with 1 are set. For example,

```
OR   AL,0F0H
```

would set the four highest bits of AL, leaving the lowest four bits unchanged.

The XOR instruction performs a bitwise logical "exclusive or" of the source to the destination. This instruction is commonly used to reverse selected bits of the destination, since bits xor-ed with 0 are unchanged, but bits xor-ed with 1 are "flipped". For example,

```
XOR  AL,0F0H
```

would negate the four highest bits of AL, leaving the lowest four bits unchanged.

The TEST instruction bears the same relation to AND that the CMP instruction bears to SUB. Namely, it sets all of the flags just as AND would, but it does not actually modify the destination operand in any way. In general, the only flags of use in any of these bitwise operations are PF (the parity flag) and ZF (the zero flag).

**ASSIGNMENT:**

1. Read Chapter 2 *through* page 38, plus section 2.7.
2. Write a "typewriter" program using EDLIN and run it using the Macro Assembler. You can use the "template" program and simply insert your own code and variables in the spaces provided. This program should:
  - a) Read the keyboard using a DOS function.
  - b) Exit from the program if the escape key is pressed.
  - c) Otherwise, convert the character to upper case.
  - d) Display the upper case character on the screen using a DOS function.
  - e) Go back to step "a".

You will turn in a *disk* (no paper!) containing the source code (call it "PROG3.ASM") and the executable code (PROG3.EXE) of the program. Write your name on the disk-label with a *felt-tipped* pen, and embed your name in the program (in a comment). I will run it to make sure that it works. (And you had better not *crash* my system, either!) Please use only the assembly language features we have discussed in class or that have been in the assigned reading.

3. Write a second program which is similar but more complex. Again, turn in the ASM file (PROG4.ASM) and the EXE file (PROG4.EXE), on the same disk as PROG3. This program should:
  - a) Using a DOS function (or a routine of your own devising), display a sign-on message (such as, "CS-5330 Program 4, by John Doe").
  - b) Input lines of text, as follows:
    - i) Display a prompt at the beginning of the line
    - ii) Input a character from the keyboard using a DOS function.
    - iii) If the character is an <ESC>, go to step "c".
    - iv) If the character is *printable* (' ' through ASCII code 7EH), display the character on the screen by using a DOS function and put it in a *text buffer* in memory.
    - v) If the character is a backspace, display it and remove the previous character from the text buffer. However, do not backspace past the physical beginning of the line.
    - vi) If the character is a carriage return, display a carriage return *and* a line feed, putting both of these characters in the buffer. Then, go to the beginning of step "b".

- vii) Ignore any other non-printable characters.
- viii) Go back to step "ii".
- c) Print the entire buffer on the printer.
- d) Return to DOS.

In addition to this, the program must check for *buffer overflow*: *that is, it must not allow any characters to be stored in the buffer once it is full. Therefore, you must provide some appropriate error routine to handle this case. Because this is our first long program, I have posted a sample solution to the problem on the door of my office and you are free to look at this if you get stuck. NOTE: I wrote this program myself (I didn't get it out of any book), so if any substantial portion of my code appears in the middle of your program I'll know (and I'll make you do it over again)!*

```

; This is a "pattern" for writing assembly language programs for use
; with the Macro Assembler. A simple program should look exactly like
; this, except that the comment "VARIABLES GO HERE!" should be replaced
; by your DB and DW operators (i.e., your variable declarations), and ;
; the comment "CODE GOES HERE!" should be replaced by your executable
; code. This is not the only "pattern" which will work, but it does
; work, which is something. The parts in BOLDFACE are the parts of
; the program that really need to be typed in (as opposed to comments).
; The parts in ITALICS are words (or values) which I have chosen for
; convenience and which could be replaced by anything else (so long as
; the change is made consistently throughout the program). All words
; not in italics must be exactly as shown.

```

```

; The following defines the system stack, for storing temporary
; variables and return addresses.

```

```

stacksize equ 100 ; size of the stack in words
stack segment stack
    dw stacksize dup (?)
stack ends

```

```

; The following defines the data area of the program. All variables
; are stored in this region.

```

```

data segment
;    VARIABLES GO HERE!
data ends

```

```

; All of the rest defines the code area of the program.

```

```

code segment
    assume cs:code,ds:data

start proc far
    mov ax,stack ; set up stack
    mov ss,ax
    mov sp,stacksize
    push ds ; standard return address setup
    mov ax,0
    push ax
    mov ax,data ; set up data segment
    mov ds,ax

```

```

;    CODE GOES HERE!

```

```

    ret
start endp

```

```

code ends

```

```

    end start

```

## SYSTEM INTERRUPTS

ACTIVATE DESIRED MS-DOS SERVICES BY PLACING THE "FUNCTION NUMBER" IN AH, AND "INTERRUPTING" THE SYSTEM:

```
MOV AH, desired service number
INT 21H ; "INTERRUPT"
```

### SOME SIMPLE SERVICES:

FUNCTION 1: "READ" THE KEYBOARD, PUTTING AN ASCII CODE IN THE AL REGISTER:

```
MOV AH, 1 ; PREPARE TO READ KBD
INT 21H ; DO IT!
```

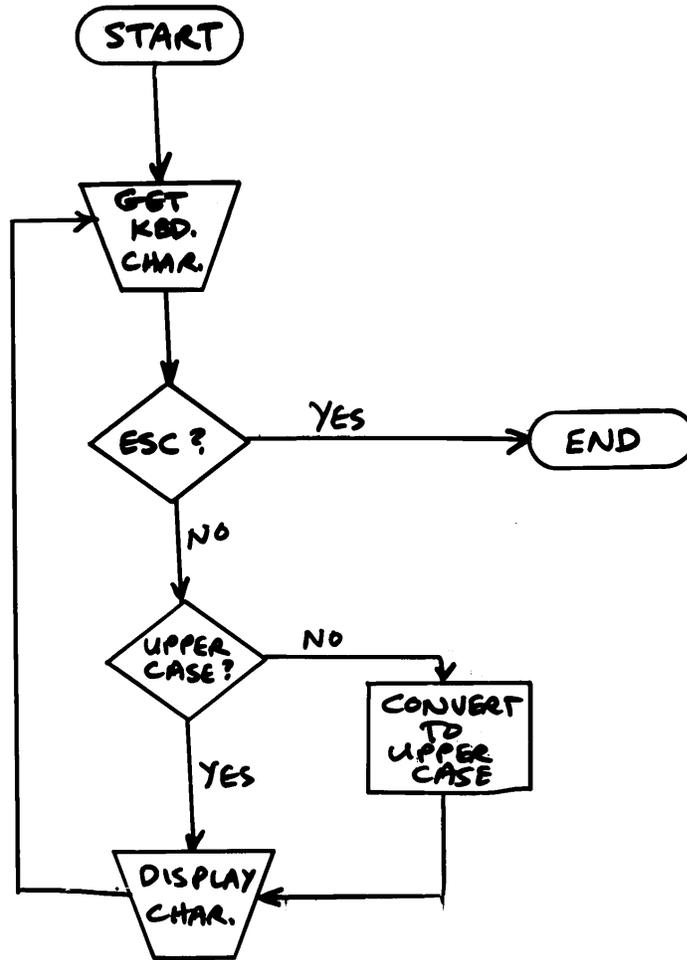
FUNCTION 2: DISPLAY ON THE SCREEN THE CHARACTER CORRESPONDING TO THE ASCII CODE IN DL:

```
MOV DL, ascii code
MOV AH, 2 ; PREPARE TO DISPLAY DL
INT 21H ; DO IT.
```

FUNCTION 5: PRINT (ON THE PRINTER) THE CHARACTER CORRESPONDING TO DL:

```
MOV DL, ascii code
MOV AH, 5 ; PREPARE TO PRINT
INT 21H ; DO IT.
```

# SIMPLE "TYPEWRITER" PROGRAM



University of Texas at Dallas  
COURSE NOTES FOR CS-5330  
IBM PC ASSEMBLY LANGUAGE

**CLASS 6**Comments

1) Grading policy. Since the last homework assignment was rather long and frightening to some of you, I'd like to comment on how grades will be assigned. In the first place, the goal of the course is for you to learn how to program in 8088 assembly language, and to learn (at least in an elementary way) how to exploit the features of the MS-DOS operating system and the IBM PC (or compatible) hardware. The homework assignments are *not* intended to test this ability. Rather, assembly language programming can be effectively learned only through practice, and the homework is intended to supply the necessary practice. This is why I am willing to freely discuss the programs with you and to post solutions on my office door. These things are all directed towards the goal of giving you familiarity with the language *and* of pointing out to you your misconceptions. The fact that the homework is not "graded" does not indicate that it is not important -- in fact, the homework is very important, since if you do not do it you will not learn the things I will eventually grade you on. Successive homework assignments will involve more and more newly learned features of assembly language and the IBM PC but will not become significantly longer.

2) You will be graded on two programming projects -- a "mid-term" project and a "final" project. You will be given several weeks to complete each of the projects (probably two weeks for the mid-term and three for the final). You are guaranteed an "A" for the course if you produce a working final project that meets the assigned specifications. If you are unable to do this, the midterm and the *quality* of the final project will be taken into account. In some cases, the homework programs may also be considered, but this will certainly not be true in general. Nevertheless, a *working program* is the standard of success in this course. We are not here to learn "armchair programming".

Review

In the previous class we finally reached the threshold of writing programs that actually do something.

First, we learned about some of the MS-DOS functions that are activated with the instruction "INT 21H". These MS-DOS functions are used to obtain input and output in our programs. We learned about 5 different MS-DOS functions:

| <u>FUNCTION</u>                           | <u>AH</u> | <u>INPUT</u> | <u>OUTPUT</u> |
|---|-----------|--------------|---------------|
| Get keyboard char.<br>(with screen echo). | 1         | --           | AL=character  |
| Display character<br>on screen.           | 2         | DL=character | --            |
| Print character<br>on printer.            | 5         | DL=character | --            |
| Get keyboard char.<br>(no screen echo).   | 8         | --           | AL=character  |
| Display string on<br>the screen.          | 9         | DX => string | --            |

Characters are represented by their ASCII codes, while a string is a sequence of ASCII codes in memory, terminated by a "\$" character.

We discussed memory *segments*. We found that program code is always taken from the *code segment*, which is pointed to by the code segment register CS. Stack operations (which we haven't discussed yet) always take place in the *stack segment*, pointed to by the stack segment register SS. String operations (which we also haven't discussed) take their source operands from the data segment (pointed to by DS) and use the extra segment (pointed to by ES) as their destination.

In accessing the data for a program, however, there is a certain amount of flexibility. By default, all data is in the data segment (pointed to by DS) unless an addressing mode is used in which the BP base register indirectly contributes to the address. (E.g., MOV AL,[BP].) In this case, the stack segment (SS) is used by default. However, these defaults can be explicitly overridden by specifying a different segment register in the instruction. (E.g., MOV AL,ES:[BP].) Here, then, is a summary of how the memory segments and segment registers are related:

| <u>TYPE OF MEMORY REFERENCE</u> | <u>DEFAULT SEGMENT BASE</u> | <u>ALTERNATE SEGMENT BASE</u> |
|---------------------------------|-----------------------------|-------------------------------|
| Instruction fetch.              | CS                          | NONE                          |
| Stack operation.                | SS                          | NONE                          |
| String source.                  | DS                          | CS, ES, SS                    |
| String destination.             | ES                          | NONE                          |
| BP used as base register.       | SS                          | CS, DS, ES                    |
| Other. (General data.)          | DS                          | CS, ES, SS                    |

In using the DEBUG program, all segment registers are initially set to the same value (and remain that way unless changed by the program), so quite often we do not need to worry about such things.

We learned something about the format which source programs need to have before they can be properly assembled and run. A "template" program was handed out. This template could be used as a guide for writing real programs; all we had to do was to insert our variable

declarations into a certain slot in the program, and insert our source code into another designated slot.

We learned how to use the Macro Assembler and the linker to create an executable program from our assembly language source-code. This was very easy. If our source code was called *source.ASM*, then to assemble the program we used the command

```
B>A:MASM source;
```

Note the use of the semi-colon at the end of the line. To then "link" the program we typed

```
B>A:LINK source;
```

Note that in neither case did we use the filename extension (initially ".ASM"). Finally, the program can be executed with just

```
B>source
```

or can be loaded and tested with DEBUG by typing

```
B>A:DEBUG source.EXE
```

Finally, we learned several new 8088 instructions.

```
CMP destination,source
```

was identical to the SUB command, except that the result of the subtraction was thrown away rather than stored in the destination operand. Thus, the net effect was to set the status flags as if a SUB instruction had been executed. This instruction is useful for comparisons of many kinds. We also saw several instructions for performing logical arithmetic. The instruction

```
NOT destination
```

took the logical one's complement (or negation) of the destination operand. Similarly, the instructions

```
AND destination,source
OR  destination,source
XOR destination,source
```

respectively ANDed, ORed, or XORed the source operand to the destination operand. The instruction

```
TEST destination,source
```

bears the same relationship to AND that CMP does to SUB. That is, it ANDs the source operand to the destination, but throws away the result rather than saving it in the destination, thus setting the flags as if an AND has been executed.

The Shift and Rotate Operations

The remaining logical instructions perform the "rotate" and "shift" operations. The best way to describe these operations is in terms of examples. Let us suppose that the AL register contains the binary value 00110011B and consider what a "shift" operation would do to the value in AL.

A *logical right shift* of the AL register moves all of the bits of our value to the right. The bit which is furthest to the right already is shifted completely out of our value (and is stored in the carry flag), while a new bit of zero is introduced on the left. Thus, our value becomes 00011001B, with the carry flag set to 1. Shifting again, we get 00001100B with CF=1. Shifting again, we get 00000110B with CF=0, etc. The logical right shift has a number of uses, of which the most important may be unsigned division by 2 (or a power of two). That is, a logical right shift is the same as integer division (of an unsigned number) by two. Moreover, the "remainder" of the division (which is always zero or one) is stored in the carry flag. The "division" nature of the shift is easily seen in the above example: in decimal notation, our original value is 51, and the successive shifts give 25, 12, and 6. Pictorially, this operation is represented as follows:

```
0 --> value --> CF
```

meaning that a zero bit enters the value from the left, and the rightmost bit is placed into CF.

The syntax of the logical right shift instruction is

```
SHR destination,count
```

where the destination operand specifies the byte or word value to be shifted (in any addressing mode except immediate), and the count operand specifies *how many* places the value is to be shifted. The count operand can have **ONLY** two forms: it can be the number 1, or it can be the register CL:

```
SHR destination,1
SHR destination,CL
```

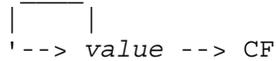
For the count operand *no* other numbers are allowed, and no other registers or addressing modes. This syntax is also used for all of the other rotate and shift operations. In our simple example, the code sequence

```
MOV AL,00110011B ; OUR SAMPLE NUMBER.
SHR AL,1         ; DIVIDE BY 2.
SHR AL,1         ; DIVIDE BY 2.
SHR AL,1         ; DIVIDE BY 2.
```

would result in AL containing 6, as would the sequence

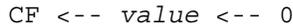
```
MOV AL,00110011B ; OUR SAMPLE NUMBER.
MOV CL,3         ; DIVIDE BY
SHR AL,CL        ; 2^3=8.
```

The *arithmetic right shift* instruction, SAR, acts identically except that instead of shifting in a zero bit at the left, the most significant bit is simply duplicated:

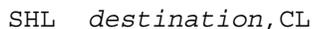


This instruction is useful for integer division by two of a *signed* value since, it preserves the sign of the result. For the example we used above, the initial value was positive, so this instruction acts exactly like SHR. Let us therefore consider a negative example. In fact, let us choose as our initial value -51 (11001101B), the two's complement of our previous example. Successive shifts give us 11100110B (-26 with CF=1), 11110011B (-13), 11111001B (-7 with CF=1), 11111100B (-4 with CF=1), 11111110B (-2), 11111111B (-1), and an endless series of 11111111B (-1 with CF=1). While these are not exactly what we normally would think of as being the results of these divisions, they are clearly closely related to the more normal conception.

The *logical left shift* instruction, SHL (or SAL, which is absolutely equivalent), is similar to the instructions discussed so far, except that the shift is to the left rather than to the right:

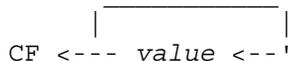


This instruction essentially performs integer multiplication of the destination operand by two, storing the carry (naturally enough) in the Carry Flag. In our example, starting with 00110011B (51), successive left shifts by one place give us 01100110B (102), 11001100B (204), 10011000B (148, with CF=1), etc. If the value is shifted by just one place, the Carry Flag may, of course, be used to detect an overflow in the result. With an instruction like



where CL contains a value greater than one, the Carry Flag cannot be used for this purpose since it contains merely the overflow bit from the *final* shift, which may be zero even if many overflows occurred on earlier shifts. However, such overflows can be detected with the Overflow Flag, which is set if any *signed* overflow has occurred.

Next, consider the *rotate* instructions. There are two rotate instructions: ROL (or rotate left) and ROR (rotate right). These instructions are like the SHL and SHR instructions, except that instead of shifting in a zero at one side, they shift in the value which would otherwise be shifted out into the Carry Flag. Pictorially, ROL does the following:



That is, the most significant bit is stored in the carry flag and shifted into the lowest bit. In our oft-used example, 00110011B would

become successively 01100110B, 11001100B, 10011001B, etc. Similarly, the ROR instruction does something like this:

```

|-----|
|---> value ---> CF

```

Clearly, for either of these operations, if we successively rotated 8 times (for bytes, or 16 times for words), we would get back the value we started with (except for the carry flag).

The *rotate through carry* instructions RCL and RCR differ from ROL and ROR in that they, respectively, perform the functions

```

|-----|
|-- CF <-- value <--

```

```

|-----|
|---> value --> CF --

```

Thus, as in the SHR and SHL instructions, the destination and the Carry Flag together form an effective 9 bit (for byte destination, or 17 bit for word destination) register, which is rotated as a unit.

As an example of the use of some of these instructions, let us write a short program fragment which multiplies the AX register by 10. As we know, the 8088 microprocessor has a built-in multiplication instruction which can perform this operation. Since we have not yet encountered this instruction, we will use the SHL instruction instead. Recall that the SHL instruction basically multiplies the destination operand by  $2^{\text{count}}$ , where the count operand is either 1 or CL. We can therefore compute  $10 \cdot AX$  by using instead the formula  $10 \cdot AX = 8 \cdot AX + 2 \cdot AX$ , and using SHL to compute the intermediate results. Since we can only conveniently test the SHL result for overflow of *signed* arithmetic, we will assume that AX is a signed value. Our program will terminate with the Overflow Flag set if an overflow occurs, and OF clear if no overflow occurs:

```

; Multiply AX by 10
    shl ax,1      ; first, multiply AX by 2.
    jo  done     ; if this results in an error, quit.
    mov bx,ax    ; otherwise, save 2*AX as BX.
    mov cl,2     ; now, multiply AX (=2*original AX)
    shl ax,cl    ; by 4.
    jo  done     ; again, quit on error.
    add ax,bx    ; otherwise, add the intermediate
                ; results to get 10*AX.
done:

```

This program is not especially efficiently written, but (even so) it is much faster than a program using the integer multiplication instruction to multiply AX by 10. From Appendix C of the text, we find that the execution time of an integer (word) multiply varies from 118 to 133 clock cycles. However, adding the execution times of the individual

instructions of our program, we get an execution time (assuming no overflow) of  $2+4+2+4+16+4+3=35$  clock cycles.

Another simple use (this time, of the SHR instruction), is to divide a byte into *nibbles*. A nibble consists of 4 bits, while a byte consists of 8 bits, so each byte consists of two nibbles. A nibble has a value between 0 and 15, and hence can be expressed in terms of a single hexadecimal digit. To get the least significant nibble of a byte (say, AL) is very easy, since we simply use the AND instruction:

```
AND  AL,0FH      ; get less significant nibble of AL.
```

To get the more significant nibble, however, requires us to divide by 16 -- that is, to shift right by four bits:

```
; Get the more significant nibble of AL:
MOV  CL,4        ; shift right by 4 bit positions.
SHR  AL,CL
```

When, eventually, we begin to directly program the computer hardware, rather than doing our I/O through the operating system, we will see that there are many other cases in which information is encoded into bit strings less than 8 (or 16) bits long. These smaller bit strings are sometimes "packed" into bytes, and are accessed through shift operations.

Just as ADDs and ADCs or SUBs and SBBs can be combined to perform multiple-precision integer addition and subtraction, shifts and rotate-through-carries can be combined to perform multiple-precision integer multiplication and division. We will see more of this later.

### Jump Instructions, Continued

So far, we have discussed the unconditional jump instruction JMP, and the conditional jump instructions JCXZ, JC, JNC, JZ, JNZ, etc. These instructions all had the syntax

*mnemonic address*

Here, the address operand was taken to be any *label* (in MASM) or any specific number (in DEBUG).

Our conception of the jump instructions must now be modified somewhat.

In the first place, there are actually three varieties of unconditional jump instructions, which we might refer to as "JMP FAR", "JMP NEAR", and "JMP SHORT". The JMP NEAR case is the default, being equivalent to JMP as we have used it so far. With JMP NEAR, the new address is assumed to be within the current code segment, so we only need to specify the new *offset* in the address rather than a new *segment* as well.

For JMP FAR, on the other hand, the new address may be anywhere in memory. Thus, a new value for the CS (code segment) register and the IP (instruction pointer) register must be specified somehow by the

instruction. We will not discuss JMP FAR further at this point since we have no immediate application for it.

JMP SHORT is a form that may be used if the new address is within -128..127 of the current address. It has the advantage of requiring both less memory and less execution time than JMP NEAR, and hence should be used whenever possible. Indeed, every JMP we have used so far in our examples or homework programs could (and should) be replaced by JMP SHORTs. The syntax of this instruction is

```
JMP  SHORT address
```

For example,

```
JMP  SHORT boston_celtics
      .
      .
      .
boston_celtics:
```

might be a reasonable replacement for

```
JMP  boston_celtics
      .
      .
      .
boston_celtics:
```

after the recent NBA playoffs. It is perfectly safe to use JMP SHORTS even if you don't actually know how far you are jumping, since if the jump is more than -128..127 bytes, the assembler will simply give you an error message to this effect.

There are several additional conditional jump instructions which we have not discussed yet. These conditional jump instructions are useful if we need to test whether one number is greater than or less than another number. The first step in determining such relationships is often to use the "CMP *destination,source*" instruction. This instruction sets (or resets) all of the flags in such a way that we can determine all of the traditional numerical relations between the destination operand and the source. There are six traditional numerical relationships: equal, not-equal, greater, less, greater-or-equal, and less-or-equal. The following table summarizes all of the conditional jump instructions relevant to testing these conditions:

| <u>Jump on the condition:</u>         | <u>Unsigned</u> | <u>Signed</u> |
|---------------------------------------|-----------------|---------------|
| <i>destination</i> .GT. <i>source</i> | JA              | JG            |
| <i>destination</i> .EQ. <i>source</i> | JE              | JE            |
| <i>destination</i> .NE. <i>source</i> | JNE             | JNE           |
| <i>destination</i> .LT. <i>source</i> | JB              | JL            |
| <i>destination</i> .LE. <i>source</i> | JBE             | JLE           |
| <i>destination</i> .GE. <i>source</i> | JAE             | JGE           |

Thus, if we executed

```
CMP  AL,80H
```

then

```
    JA    address
```

would jump if AL, considered as an unsigned number, was greater than 80H (128). Similarly,

```
    JC    address
```

would jump if AL was greater than 80H considered as a *signed* number -- i.e., greater than -128. Some of these instructions are actually the same as other instructions we have encountered earlier under different names. For example, JE is absolutely the same instruction as JZ. These two different names are provided by the assembler as a convenience to you (to make your code clearer). JE is typically used when testing for "equality", while JZ is used when testing for "zero". Other of the jumps have no equivalent among the single-flag conditional jumps and actually test *combinations* of flags to determine if a jump should be taken.

Although we have not mentioned it before, *all of the conditional jumps are SHORT jumps*. Thus, instructions like JC, JNC, JZ, JNZ, etc., are incapable of jumping to a label more than -128 or 127 bytes away. This can become a problem in a complicated program, and you will undoubtedly encounter assembler error messages complaining about your conditional jumps before long. The only reasonable cure for this problem is to avoid writing "spaghetti" code with a complex structure of jumps, and to write instead structured code with "calls" to routines that perform well defined functions. We will discuss such subroutines next.

### Stack Operations, Procs, and Calls

Although we have been able to write some programs (the homework) that jump around to different parts of the program in some intricate pattern, there is a flaw in our ability to control program execution. That flaw is that we have not yet seen any way to execute a fragment of code and then *return* to where we were. The ability to do this, however, is crucial if we are to write programs that we can understand.

This can be made clearer with an example. In our last homework assignment, there was a program that we can schematically outline as follows:

again:

```
    get_a_keyboard_character_into_AL
    cmp  al,27                ; escape key?
    je   done                ; if so, quit.
    convert_AL_to_upper_case
    display_AL_on_screen
    jmp  again
```

done:

In actuality, the lines "get\_a\_keyboard\_character\_into\_AL", etc. were implemented by code sequences that ranged from fairly trivial to slightly more complicated. If we could write our programs in a way that was more similar to the pseudo-code shown here, however, two

advantages would result. One is clearly that our programs would be easier to understand. The other is that we could easily re-use sequences of code. In this example, there is only one place in the program where we convert a character to upper case -- but in other programs it could easily happen that an "upper case" function was required at many points in the program, and we would have to type in the same code over and over again.

We can do something very similar to the pseudo-code if the concept of a *procedure* is introduced. A procedure is a portion of a program which executes when it is *called*, and then *returns* to the address immediately following the instruction that called it. If, for example, we introduced procedures called "GETCHR" [for "get character (from keyboard)"], "PUTCHR" [for "put character (to screen)"], and "UPCASE" (for "convert to upper case"), we could write the program as

again:

```

    call getchr      ; get a keyboard character
    cmp  al,27      ; escape key?
    je   done       ; if so, quit.
    call upcase     ; convert to upper case
    call putchr     ; display the character
    jmp  again

```

done:

using the new instruction "CALL", which has the same syntax as a "JMP" instruction. This code fragment, which is clearly much easier to understand than the homework solution we wrote earlier, would actually work, if we went ahead and wrote the source-code for the procedures GETCHR, PUTCHR, and UPCASE.

Just as we had a "pattern" or "template" for writing programs, we can have a pattern (albeit a very simple one) for writing procedures. Here is a "template" for a procedure:

```

name proc
;   PUT CODE FOR PROCEDURE HERE!
    ret
name endp

```

The *name* of the procedure is selected by the programmer -- in this case, it would presumably be "getchr", "putchr", or "upcase". Although procedures may be *nested* (we'll talk about this later), for now we'll assume that all procedures are defined within the code segment, but do not overlap. (Discuss handout.)

One thing that procedures *do not* automatically do (but which it would be nice to have) would be for all registers which are not actually used to pass arguments to and from a procedure to be preserved. For example, when we use the DOS functions, the registers not actually used remain the same after the INT 21H instruction as before it. In our routine UPCASE we also find the registers preserved, except for the AL register (which contains the value output by the procedure). However, the GETCHR and PUTCHR procedures do not have this property -- GETCHR modifies the AH register, while PUTCHR modifies the AH register and the DL register.

We will see in a moment that with the 8088 microprocessor it is only convenient to preserve 16 bit registers, and not 8 bit registers. Thus, since GETCHR uses the 16 bit register AX to return a result, we will "forgive" it for changing AH (even though only AL actually contains the result). In this extended sense, GETCHR also preserves registers. UPCASE, however, is now guilty of screwing up two 16 bit registers, AX and DX, rather than two merely 8 bit registers, AL and DL.

We can arrange for UPCASE to preserve registers by introducing the PUSH and POP instructions. Both of these instructions have the syntax

*mnemonic register*

The instruction

*PUSH register*

"saves" the value of the register, while

*POP register*

restores it. We will see in a moment how these instructions (and CALLs and RETurns) work. The most important aspect of these instructions, however, is that there are two simple rules governing their use:

- 1) **For every PUSH there must be an eventual POP (before any RET can be executed), and**
- 2) **Registers are popped in reverse order of the way they were pushed. Thus, PUSH AX/PUSH BX should be reversed with POP BX/POP AX.**

To see how these instructions work in practice, let us make the procedure UPCASE preserve the AX and DX registers. To do this, we must save (PUSH) the values of the registers before they are modified, and restore (POP) them before exiting the procedure with RET. Our new routine might look something like this:

```
; DISPLAY THE AL CHARACTER ON THE SCREEN.
PUTCHR PROC
    PUSH AX          ; SAVE AX REGISTER.
    PUSH DX          ; SAVE DX REGISTER.
    MOV  AH,2        ; PREPARE TO DISPLAY THE CHARACTER.
    MOV  DL,AL
    INT  21H
    POP  DX          ; RESTORE THE DX REGISTER.
    POP  AX          ; RESTORE THE AX REGISTER.
    RET
PUTCHR ENDP
```

In the next class, we will learn about the *stack* -- i.e., about how the PUSH, POP, CALL, and RET instructions actually do what they do -- and about MS-DOS functions for accessing disk files.

**ASSIGNMENT:**

1. Do the problems for Chapter 2. Read Chapter 3.4 *through* section 3.7. This chapter need not be thoroughly understood at this point since you will gain competence in the material of Chapter 3 through practice rather than through study.
2. Write the assembly-language source code for a *procedure* (that is, code beginning with the "PROC" pseudo-op, ending with the "ENDP" pseudo-op, called with a "CALL" instruction, and finished with a "RET" instruction) whose specifications are as follows:

The routine should convert a byte-value in AL to an ASCII string in DX. For example, if the input to the routine is 7FH (in the AL register), then the output should be the character '7' in the DH register and 'F' in the DL register. A logical way of writing this procedure (to my mind) would be something like this in pseudo-code:

```
get the most significant nibble (4 bits) of AL
convert it to a hex digit
store the hex digit in DH
get the least significant nibble of AL
convert it to a hex digit
store the hex digit in DL
```

Clearly, for this, it would be helpful to write *yet another* procedure to convert a nibble (0-15) to a hex digit ('0'-'9', 'A'-'F').

Test this routine. (For example, you can test it using DEBUG.) You need not explicitly turn in the subroutine since it will be used (and therefore embedded in) the next problem:

- 3) Write a "file dump" *program* (called PROG5.ASM) which does the following:
  - a) Displays a prompt on the screen and inputs a string from the keyboard. This string represents a filename. You can use DOS functions 9 and 10 for this.
  - b) Uses a DOS function (3DH) to open (for reading) the specified file. If the file does not exist, displays an error message and quits.

c) Reads (DOS function 3FH) the file and (using the procedure you wrote earlier) produces a hexadecimal and ASCII dump similar to the D(ump) facility in DEBUG. Each line of the display should consist of: a relative byte number, 16 bytes displayed in hexadecimal, and 16 characters. For non-printable characters, a dot should be displayed. For example, here is a short sample dump of a text file:

```
0000  20 20 20 20 20 20 20 20 20 54 68 69 73 20 69 73 20          This is
0010  61 20 73 61 6D 70 6C 65 20 66 69 6C 65 20 63 6F  a sample file co
0020  6E 74 61 69 6E 69 6E 67 20 74 65 78 74 20 74 6F  ntaining text to
0030  20 64 75 6D 70 2E 0D 0A 54 68 69 73 20 69 73 20  dump...This is
0040  74 68 65 20 73 65 63 6F 6E 64 20 6C 69 6E 65 20  the second line
0050  6F 66 20 74 68 65 20 66 69 6C 65 2E 0D 0A 0D 0A  of the file.....
```

d) Closes the files (DOS function 3EH).

e) Quits.

; This is the simple "typewriter" program using procedures.

```
stack    segment stack
         dw      1000 dup (?)
stack    ends

data     segment
data     ends

code     segment
         assume  cs:code,ds:data
start    proc    far
         push    ds                ; standard return address setup
         mov     ax,0
         push    ax
         mov     ax,data          ; set up data segment
         mov     ds,ax

         mov     ah,5
         mov     al,1
         int     10h
         mov     ah,3
         mov     dx,0
         mov     bh,1
         int     10h

again:   call    getchr           ; get a keyboard character.
         cmp     al,27           ; escape?
         je      done           ; if done, quit.
         call    upcase         ; convert to upper case.
         call    putchr         ; display the character on the screen.
         cmp     al,13          ; carriage return?
         jne    again          ; if not, then loop.
         mov     al,10          ; display a line feed.
         call    putchr
         jmp     short again

done:    mov     ah,5
         mov     al,0
         int     10h
         mov     ah,3
         mov     dx,0
         mov     bh,0
         int     10h

         ret
start    endp
```

```
; Here is a procedure to get a keyboard character into AL.
getchr  proc
        mov     ah,8      ; use a DOS function.
        int     21H
        ret
getchr  endp

; Here is a procedure to display AL on the screen.
putchr  proc
        mov     ah,2      ; use a DOS function
        mov     dl,al
        int     21H
        ret
putchr  endp

; Here is a procedure for converting AL to upper case.
upcase  proc
        cmp     al,'a'    ; below 'a'?
        jb     already_upper ; if yes, do nothing.
        cmp     al,'z'    ; above 'z'?
        ja     already_upper ; if yes, do nothing.
        add     al,'A'-'a' ; convert to upper case
already_upper:
        ret
upcase  endp

code    ends
        end     start
```

## GRADING POLICY

- ① THE GOAL OF THIS COURSE IS FOR YOU TO LEARN TO PROGRAM IN IBM PC (COMPATIBLE) ASSEMBLY LANGUAGE. THEREFORE, YOUR GRADE WILL FINALLY BE BASED ON WHETHER YOU CAN WRITE WORKING PROGRAMS, AND (TO A MUCH LESSER EXTENT) ON THE QUALITY OF THE PROGRAMS.
- ② THE HOMEWORK IS NOT INTENDED TO TEST YOU. IT IS NOT "GRADED", AS SUCH. ASSEMBLY LANGUAGE IS LEARNED THROUGH PRACTICE. THE HOMEWORK PROGRAMS ARE INTENDED TO GIVE YOU THIS PRACTICE, SO THAT YOU CAN LEARN FROM YOUR MISTAKES. IT IS, THEREFORE, VERY IMPORTANT TO DO THE HOMEWORK PROGRAMS.
- ③ YOU WILL BE GRADED ON TWO "PROJECTS". THE MID-TERM PROJECT WILL TAKE 2 WEEKS TO COMPLETE, AND THE FINAL PROJECT 3 WEEKS. A FINAL PROGRAM THAT WORKS AND MEETS THE ASSIGNED SPECIFICATIONS INSURES AN "A" IN THIS COURSE. A FINAL PROGRAM OF LESS QUALITY, HOWEVER, WILL BE PARTIALLY COMPENSATED BY A GOOD MIDTERM (AND POSSIBLY BY THE HOMEWORK PROGRAMS).

SIMPLE MS-DOS FUNCTIONS

| <u>FUNCTION</u>                    | <u>AH</u> | <u>INPUT</u> | <u>OUTPUT</u> |
|------------------------------------|-----------|--------------|---------------|
| GET KEYBOARD CHARACTER (WITH ECHO) | 1         | —            | AL=CHARACTER  |
| DISPLAY CHARACTER ON SCREEN        | 2         | DL=CHARACTER | —             |
| PRINT CHARACTER ON PRINTER         | 5         | DL=CHARACTER | —             |
| GET KEYBOARD CHARACTER (NO ECHO)   | 8         | —            | AL=CHARACTER  |
| DISPLAY STRING ON SCREEN           | 9         | DX ⇒ STRING  | —             |

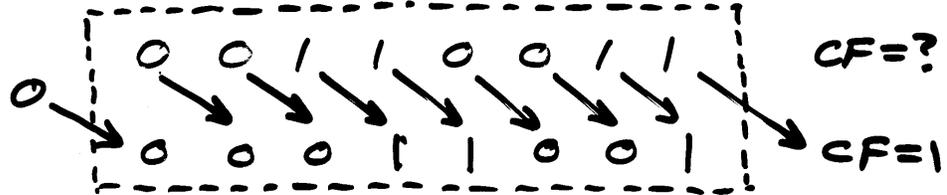
MEMORY SEGMENTS

| <u>TYPE OF MEMORY REFERENCE</u> | <u>DEFAULT SEGMENT</u> | <u>ALTERNATE SEGMENT</u> |
|---------------------------------|------------------------|--------------------------|
| INSTRUCTION FETCH               | CS                     | NONE                     |
| STACK OPERATION                 | SS                     | NONE                     |
| STRING SOURCE                   | DS                     | CS, ES, SS               |
| STRING DESTINATION              | ES                     | NONE                     |
| BP AS BASE REGISTER             | SS                     | CS, DS, ES               |
| OTHER (GENERAL DATA)            | DS                     | CS, ES, SS               |

# SHIFT

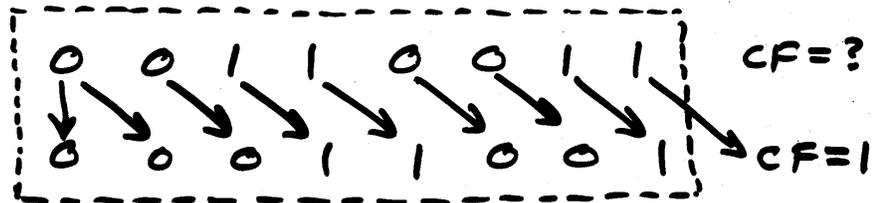
"Logical" right shift SHR :

0 → value → CF



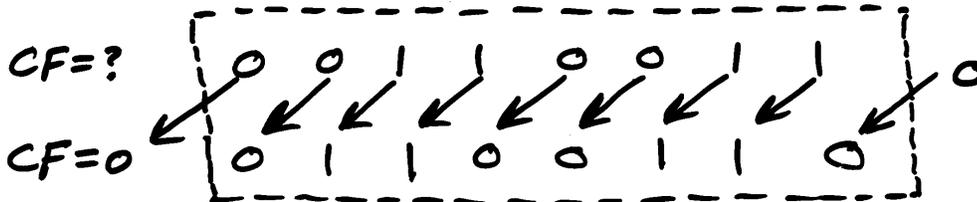
"Arithmetic" right shift SAR :

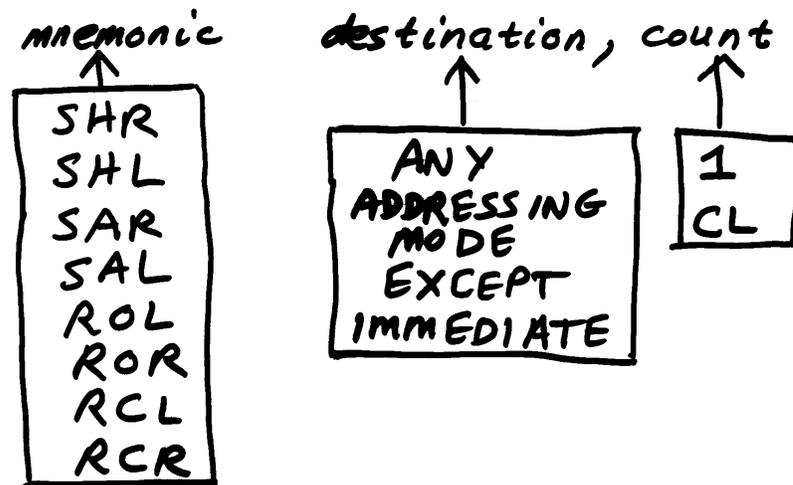
Value → CF



"Logical" or "Arithmetic" left shift SHL, SAL :

CF ← value ← 0



SYNTAX OF SHIFT (OR ROTATE)EXAMPLES:

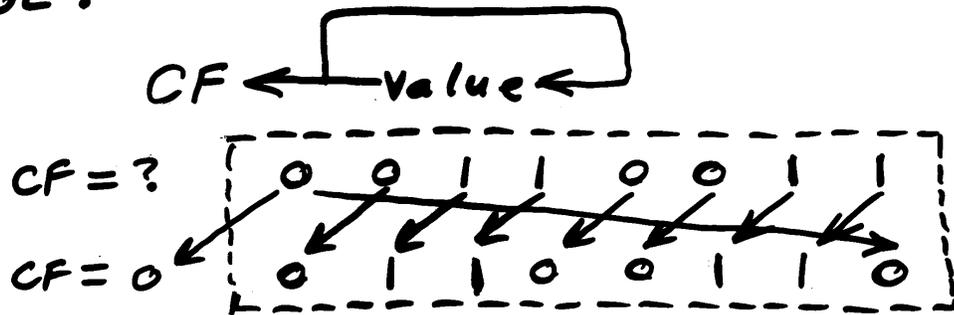
```
SHR AL,1 ; UNSIGNED AL/2
SAR DX,1 ; SIGNED DX/2
SHL Foo,1 ; Foo ← Foo*2
```

```
[ MOV CL,4 ]
```

```
SHR AL,CL ; UNSIGNED AL/16
SAR DX,CL ; SIGNED DX/16
SHL Foo,CL ; Foo ← Foo*16
```

ROTATE

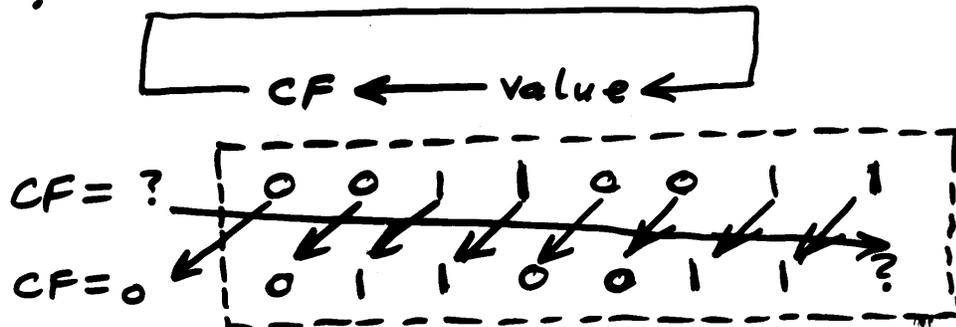
ROL :



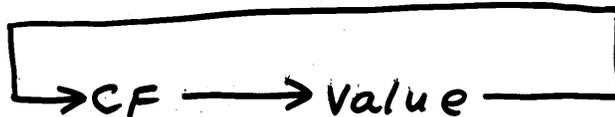
ROR :



RCL :



RCR :



## SAMPLE LOGICAL PROGRAMS

---

① SUPPOSE AL = 7EH

```

; GET "HIGH NIBBLE" OF AL :
  MOV CL, 4      ; DIVIDE
  SHR AL, CL    ; BY 16 .

```

Now, AL = 07H .

---

② SUPPOSE AL = 7EH

```

; GET "LOW NIBBLE" OF AL :
  AND AL, 0FH

```

Now, AL = 0EH

---

③ ; MULTIPLY AX BY 10 ,  
; USING  $10 * AX \equiv 8 * AX + 2 * AX$ .

```

  SHL AX, 1      ; AX ← 2 * AX.
  JO  DONE      ; QUIT IF OVERFLOW.
  MOV BX, AX     ; SAVE 2 * AX.
  MOV CL, 2      ; PREPARE TO
  SHL AX, CL     ; MULTIPLY BY 4 .
  JO  DONE      ; QUIT IF OVERFLOW.
  ADD AX, BX     ; FINALLY, 10 * AX .
DONE:

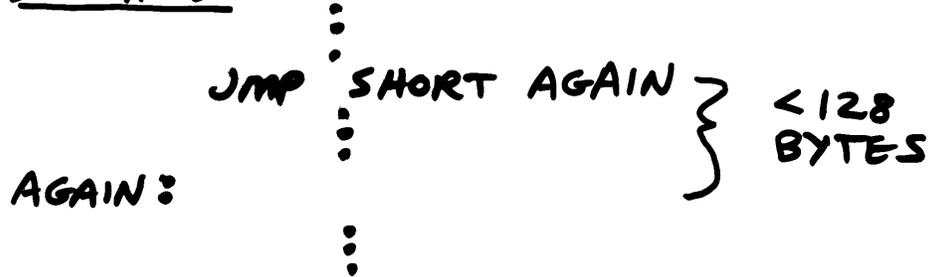
```

JUMP "SHORT"

JMP SHORT address

USE IF address IS WITHIN -128 .. 127 OF CURRENT ADDRESS .

EXAMPLE:



NEW CONDITIONAL JUMPS

mnemonic address

[USE AFTER CMP destination, source]

| <u>JUMP IF</u>       | <u>UNSIGNED</u> | <u>SIGNED</u> |
|----------------------|-----------------|---------------|
| destination > source | JA              | JG            |
| destination = source | JE              | JE            |
| destination ≠ source | JNE             | JNE           |
| destination < source | JB              | JL            |
| destination ≤ source | JBE             | JLE           |
| destination ≥ source | JAE             | JGE           |

PSEUDO-CODE FOR "TYPEWRITER" PROGRAM

AGAIN:

```

Get_a_keyboard_character_into_AL
CMP AL, 27           ; ESCAPE KEY?
JE  DONE            ; QUIT IF SO.
Convert_AL_to_upper_case
display_AL_on_screen
JMP SHORT AGAIN

```

DONE:

"TYPEWRITER" WITH PROCEDURE CALLS

```

AGAIN: CALL GETCHAR      ; GET KBD CHAR.
        CMP AL, 27       ; ESCAPE KEY?
        JE  DONE        ; QUIT IF SO.
        CALL UPCASE     ; CONVERT TO UPPER.
        CALL PUTCHEX    ; DISPLAY IT
        JMP SHORT AGAIN

```

"TEMPLATE" FOR PROCEDURES

name PROC

; PUT CODE HERE!

RET

name ENDP

University of Texas at Dallas  
 COURSE NOTES FOR CS-5330  
 IBM PC ASSEMBLY LANGUAGE

## CLASS 7

Comments

1. Although all of the programs I have looked at have worked (more or less), I have made some comments in some of your programs. If you want to read these comments, look for files on your disk with names like PROG3.AS\$ or PROG4.AS\$ (or anything that ends in a '\$'). These are simply your source programs, with comments added. The comments begin with "\*\*\*\*RSB\*\*\*\*" and so are easy to find. Of course, if I felt that no comments were required, there are no such files on your disk. In general, I was very pleased with the programs I saw.

2. Some general comments are in order, however. First, although the sequence line-feed/carriage-return is in theory (and mostly in practice) equivalent to carriage-return/line-feed, I have occasionally seen instances in which a program or a hardware device had problems dealing with lf/cr. Consequently, it is safer to stick to the cr/lf sequence.

3. Many of you have been using instructions like

```
CMP  AL,41H      ; CHECK FOR CAPITAL "A"
MOV  DL,32       ; MOVE SPACE INTO DL
etc.
```

There is nothing wrong with this, except that it makes the programs harder to understand. It is better to use self-evident code like

```
CMP  AL,'A'
MOV  DL,' '
```

4. On several different occasions, I have given you different stories about whether MS-DOS function calls preserve the CPU's registers. To reiterate, while I have never seen a statement in print to the effect that MS-DOS preserves registers, it has always been my experience that MS-DOS function calls do not affect the registers (other than those used to pass arguments). For that reason, I told you to assume that registers are preserved by MS-DOS calls. It has come to my attention, however, that some calls change the registers. In particular, the "print string" function (number 9) modifies the AX register even though it does not output any values in this register. On my computer, using function number 9 always results in having a dollar sign, "\$" in the AL register. (This fact screws up at least one of your programs, when run on my computer). For this reason,

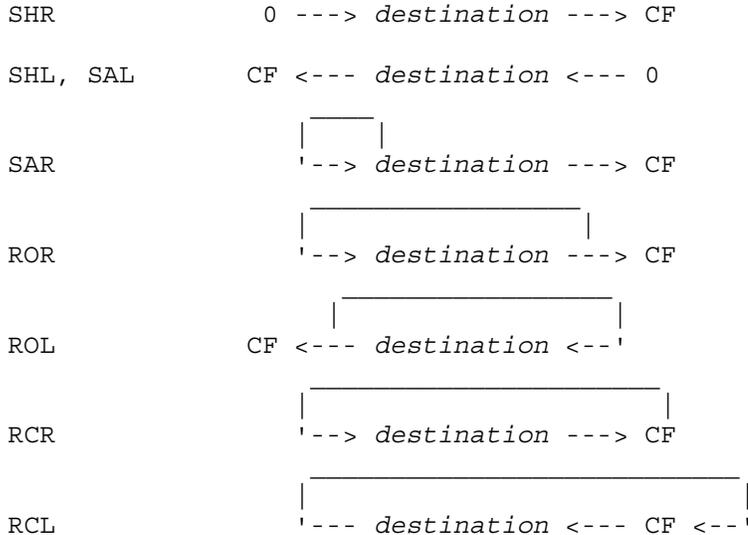
**from now on, we will assume that MS-DOS does not  
 preserve registers.**

This will hold even if you test the functions you are using and determine that they preserve registers. The reason for this restriction is that while the DOS functions preserve certain registers

on *your* machine, they might not behave this way on a different machine or under different circumstances.

Review

In the last class, we finished up the discussion of all logical instructions available on the 8088 microprocessor. This included the *logical shift* instructions SHR and SHL, the *arithmetic shift* instructions SAR and SAL (=SHL), the *rotate* instructions ROR and ROL, and the *rotate through carry instructions* RCR and RCL. *Pictorially*, we could represent the behavior of these instructions as follows:



All of these instructions have the syntax

*mnemonic destination, count*

where the destination operand can be a word or byte in any addressing mode (except immediate) and the count operand gives the number of bit positions by which the value is to be shifted or rotated. The count operand is restricted to being the number 1 or the CL register. The most important use of these instructions is probably for fast integer multiplication or division by two.

We continued with our discussion of the various "jump" instructions provided by the 8088 microprocessor. We found that the "JMP" instruction which we had already used rather freely has several variations. The most important variation, other than the "plain vanilla" JMP is the "JMP SHORT". The JMP SHORT instruction executes as fast as JMP, but requires one less byte in memory. It can be used only when the destination address is within 128 bytes of the current address.

We found that all conditional jump instructions are SHORT jumps, which can result in some problems with more complicated programs (in particular, with "spaghetti code"). Several new conditional jump instructions were discussed. These new conditional jumps are all used in conjunction with the CMP instruction and correspond to testing the

conditions EQ, NE, LT, GE, GT , and LE. For each such condition, there are separate conditional jumps for signed and for unsigned arithmetic.

One way to avoid spaghetti code and, at the same time, make our code easier to write, debug, and understand, is to make some use of subroutines to modularize our code. To this end, we discussed procedures. Procedures are defined using the PROC pseudo-op. Procedures are pieces of code that execute some pre-defined function when they are "CALled", and then "RETurn" to immediately begin executing the instruction after the CALL. Procedures have their own *names* and act much like procedures in higher-level languages.

We also discussed the instructions

```
PUSH 16-bit-register
POP  16-bit-register
```

which, respectively, save the contents of a register or restore the contents of a register. These instructions are quite frequently used for temporary storage of values, particularly with procedure calls, so that a procedure can preserve all of the registers it does not actually use for returning output values. However, only word registers can be pushed or popped, so instructions like PUSH AL are not allowed.

The CALL, RET, PUSH, and POP instructions all operate by using the *stack*, which we will now discuss.

### The Stack

In our "typewriter" program homework assignment, we encountered the use of a "text buffer" for temporarily storing information. In using our buffer, we employed a *pointer* (usually a register) which picked out some location in the buffer as being the "current" location -- i.e., the next position at which a byte could be stored (or, in some of your programs, the last position at which information was stored previously). Every time a new item was added to the buffer, the pointer was incremented (or decremented, depending on the program) to point to the next position, and so forth. When the program was finally finished, the characters were removed from the buffer one-by-one and printed on the printer.

This type of data structure is known as a *queue*. In a queue, data is added to the *end* of the buffer, but it is removed from the *beginning* of the buffer. A queue is also called a *FIFO list* -- a list of data items for which the *First* items Input are also the *First* items Output.

There is a similar type of data structure, called a *stack*, which is somewhat more important in microcomputing. A stack is just like a queue, except that the *last* item added to the stack is the *first* removed. The 8088 microprocessor has several built-in registers and instructions which make using a stack much simpler than, say, using a queue. Effective use of the 8088 microprocessor is impossible without the stack.

In an 8088 program, there is typically one stack used, known as *the stack*. The stack is used to restore RETurn addresses for use with CALL instructions, and to store the values of registers saved with the

PUSH instruction. Arguments for procedures are also often passed on the stack. Some high-level languages store "local" variables for subroutines on the stack, as opposed to "global" variables stored in the data segment. In any case, it is clear that the stack may be used constantly in many programs.

The parameters of the stack (its size and position) are arbitrary, and are decided by your program, which often initializes the stack at the beginning and then essentially forgets about it. The stack resides in its own memory segment, the *Stack Segment*, which is pointed to by the stack segment register, SS. The maximum size of the stack is arbitrary, except that it must be less than 64K. The 8088 microprocessor itself never checks at any time whether a "stack overflow" has occurred, unlike our clever homework program. The programmer, therefore, is responsible for reserving more space on the stack than will ever be used. The stack has its own dedicated "pointer register", SP, to pick out the current position of the "top" of the stack. In actuality, the stack grows downward in memory, so the top of the stack is always at the lowest memory location used.

Here's how the stack is actually used. In the first place, the entries in the 8088's stack are word values. It is not possible to store bytes in the stack, except insofar as they are parts of words. When we PUSH a register, the 8088 decrements the stack pointer (SP) twice, and stores the value of the register at SS:SP. Thus, in the 8088, the stack pointer always points at the most recently used address on the stack. Conversely, when the register is POPped, the value at the top of the stack is stored in the register and then SP is incremented twice. The stack is a *Last In First Out* data structure. This inverse behavior of the PUSH and POP instructions explains the rule we mentioned in the last class that **registers must be popped in reverse order of the way they were pushed**. Now that we understand more about how the stack works, of course, we see that this rule is not strictly accurate -- rather, it is just good advice. If we POP the registers in a different order, it simply means that we will have permuted the values stored in the registers. For example, suppose that we would like to take the value currently in the AX register and store it in the BX register, while simultaneously storing BX in CX and CX in AX. A program for that might look like this:

```
; PROGRAM TO CYCLICALLY PERMUTE THE AX, BX, AND CX ; REGISTERS:  FIRST,
PUSH ALL OF THE REGISTERS.
    PUSH AX
    PUSH BX
    PUSH CX
; NOW, POP THE REGISTERS IN THE PROPER ORDER:
    POP  AX  ; AX = ORIGINAL CX
    POP  CX  ; CX = ORIGINAL BX
    POP  BX  ; BX = ORIGINAL AX
```

Similarly, when you CALL a procedure, the 8088 PUSHes the return address onto the stack and then performs a JMP to the procedure. When you RETURN from the procedure, the 8088 POPs the return address off of the stack and JMPs to the return address. There are actually two variations of the CALL and RET instructions, corresponding to the JMP NEAR and JMP FAR instructions. (There is no analog to the JMP SHORT.) For a procedure *within* the segment, only the offset word of the return

address needs to be pushed onto the stack. To call a procedure in a *different* segment, however, both the segment word and the offset word must be pushed. The fact that both PUSHes and CALLs store values on the stack explains the other rule for using PUSHes and POPs: namely, that **for each PUSH there must be a POP, before any RETs occur.** This is a good rule of thumb, although we see now that is not strictly accurate. For example, in some cases, we might want to store the RETURN address on the stack using PUSHes rather than a CALL. This actually happens with our "template" program. The operating system runs our programs by executing a CALL FAR to our main procedures. The first act of the main procedure in our template program, however, is to PUSH two values onto the stack:

```
; SET UP RETURN ADDRESS FOR TEMPLATE PROGRAM:
    PUSH DS    ; SEGMENT OF RETURN ADDRESS WITHIN DOS.
    MOV  AX,0
    PUSH AX    ; OFFSET OF RETURN ADDRESS WITHIN DOS.
    .
    .
    .
; RETURN TO DOS
    RET  FAR
```

This works because *when the program begins executing* (i.e., before DS has been modified by the program) DS:0 is the return address within DOS for the program. By default, RETURN instructions in procedures declared with "PROC" are RET NEARs, while RETURNS in PROC FARs (like our template program) are RET FAR, so we did not explicitly have to say RET FAR in the sample program fragment.

Although it was not mentioned in the previous class, the PUSH and POP instructions can actually be used with any addressing mode, not just with registers. Two restrictions apply: you (apparently) cannot push an immediate value onto the stack, and you can only push words (not bytes) onto the stack. Thus, a more accurate syntax for PUSH and POP would be

```
PUSH source
POP  source
```

For instance, we could use instructions like

```
PUSH BAR      ; PUSH THE VALUE OF BAR ONTO THE STACK.
POP  BAR[SI]  ; STORE TOP-OF-STACK AS BAR[SI].
```

where BAR is our standard word variable.

Let us consider a short sample program using the stack. We will write a *recursive* procedure for computing the value of  $N!=1*2*\dots*N$ . In Pascal, such a program might look like this:

```
function factorial(n:integer):integer;
begin
    if n=0 then factorial:=1
    else factorial:=n*factorial(n-1)
end;
```

This function is recursive, in that it CALLs itself. In order to compute factorial(n), the computer must first compute factorial(n-1), etc. However, the computer does not know this: it thinks that it can simply directly compute factorial(n). When it discovers that it must first compute factorial(n-1), however, it is halfway through the function; so it PUSHes the *environment* (the local variables called "n" and "factorial", the return address, and various variables used internally by Pascal) onto the stack and begins executing factorial(n-1). Of course, during the computation of factorial(n-1), it must again store the environment on the stack and begin to compute factorial(n-2), etc. Thus, the ability to perform such a calculation recursively is very dependent on the stack.

In assembler, the program might go something like this:

```
; PROCEDURE TO COMPUTE N!. WE WILL PASS BOTH INPUT AND
; OUTPUT IN THE AX REGISTER. IF INTEGER OVERFLOW OCCURS,
; THE OVERFLOW FLAG WILL BE SET.
FACTORIAL PROC
    CMP  AX,0      ; N=0?
    JNE  NOT_ZERO ; IF NOT, THEN CONTINUE.
    MOV  AX,1      ; OTHERWISE, RETURN 0!=1.
    JMP  DONE
; THE RECURSIVE STEP. AT THIS POINT, WE KNOW N<>0.
; COMPUTE N! AS N*(N-1)!
NOT_ZERO:
    PUSH AX      ; STORE N, SO IT WON'T BE MESSED UP
    DEC  AX      ; WHEN (N-1)! IS COMPUTED.
    CALL FACTORIAL ; COMPUTE (N-1)! AND STORE IN AX.
    POP  BX      ; GET BACK N, BUT IN BX.
    JO   DONE    ; QUIT ON ERROR.
    MUL  BX      ; MULTIPLY AX BY BX, .
; EXIT POINT OF THE PROCEDURE.
DONE:
    RET
FACTORIAL ENDP
```

Aside from the recursion, the only novel feature of this program is the use of the MULtiple instruction, which performs an unsigned multiply of its operand with the accumulator. We will discuss the multiply and divide instructions more fully later. As it happens, 8! is 40,320 so that 9! overflows (leaving OF set).

### Accessing Files

Like the simple character I/O functions considered earlier, all file accesses, in which data is input from files or output to files, are managed by DOS. However, since files operations are intrinsically more complex than character operations, the DOS interface with your program is also somewhat more complex.

In order to access disk files, there is a sequence of DOS calls which must be executed more-or-less in order. Similar sequences of instructions are also required in many high-level languages, so let us discuss the necessary steps before going on to find out about the DOS functions provided to carry out these steps.

First, we need some way to *input* a filename from the keyboard. (This step is not always strictly necessary, since we might "hard-code" a fixed filename into our programs. However, we'll assume that we're being flexible and therefore will input our filenames at run-time.) A filename is just a string, so what we would really like is some kind of whole-string input function, just as we already have a whole-string display function.

Second, we need some way to associate the filename, which is a string in memory, with an actual file on the disk. There are several possibilities here. If the file already exists and we simply wish to read or rewrite parts of it, we want to *open* the file. A file-open operation is equivalent to the **reset** operation in Pascal. If the file does not exist or if we want to empty the file and then write to it, this is a *create* operation. *Create* corresponds to the Pascal command **rewrite**.

Third, we want to move to the proper position in the file for the particular data we are interested in. For example, to read or write the entire file, we would start at the beginning. To add to the end of the file, we would move to the end. For "random access", we would want to move to an arbitrary point in the file. We might call such movement within the file a *seek* operation, since it seeks the proper location for I/O.

Fourth, we want to actually perform the desired *read* or *write* operations. These are equivalent to the Pascal statements **read** and **write** (or **get** and **put**, depending on how bad the particular version of Pascal). We can perform as many *seeks*, *reads*, and *writes* as we wish before going on the fifth step:

Fifth, and finally, when we are done we want to *close* the file. File operations use up various MS-DOS resources, such as memory used for buffers, so closing the file represents a helpful (and, in fact, required) bookkeeping chore. This is comparable, not surprisingly, to the Pascal **close** operation.

To see how all of these steps actually fit together in a real program, let's consider a pseudo-code version of the current homework assignment, which "dumps" a file, in hexadecimal:

```

; Program to do a "file dump".
    Prompt_user_to_input_filename_for_dump
    Read_filename_from_keyboard
; This file must already exist if we're to read it, so we
; must open it rather than create it:
    Open_the_file
    If error (file doesn't exist), then JMP ERROR

; Main loop of program:
AGAIN:
    Read_16_byte_record_from_file
    If error (end of file, or disk read error), JMP QUIT
    Display_the_16_bytes_in_hex
    JMP AGAIN

QUIT:

```

```

        Close_the_file
        JMP DONE

; Error exit for nonexistent file
ERROR:
        Print_error_message

; Exit point of program:
DONE:

```

Now let's see how some of these steps correspond to MS-DOS functions that we can really call.

### DOS Functions for Files

For each of the various operations we have described, there is a DOS function to handle it. For now, we will only describe the functions that are really relevant to the homework program assignment, with other functions described next time.

DOS function 10 (0AH) is the *buffered input, or string input* function. It allows the input of an entire string at one time (as opposed to the character-by-character input we have used already). This function is very convenient in that it allows the use of all the built-in editing functions of MS-DOS, like the backspace, escape, function keys, INS, DEL, etc. In order to use function 10, it is necessary to first set up a buffer area in the data segment, much like the buffer we used in our typewriter program. Assuming that we want to enter up to N characters in our string, the buffer must be N+2 characters long, since the first two characters have a special meaning. The first byte, on input, must contain a value corresponding to the *maximum* number of characters in the string (i.e., N). The second byte is meaningful only on output (i.e., after function 10 has been executed) and contains a count of the number of characters actually entered. This number can be less than the maximum, since input can be terminated either when the maximum number of characters has been entered, or when the carriage return key is pressed. Here is a sample use of buffered input:

```

BUFSIZE    EQU 32                ; LET THE FILENAME BE A MAX. 32 CHARS.
BUFFER     DB     BUFSIZE
BUFLLEN    DB     ?
BUFSTR     DB     BUFSIZE DUP (?)
.
.
.
MOV  AH,10    ; FUNCTION 10.
MOV  DX,OFFSET BUFFER
INT  21H

```

The definition of the buffer may seem slightly tricky at first, but actually it's easy to understand. The buffer as a whole begins at BUFFER which, as mentioned before, is also a byte variable whose value is the maximum number of characters to be entered. We have gone out of our way to give the second byte of the buffer a name (BUFLLEN) since the number of actual characters entered will eventually be stored there and

we will probably need to access it. Of course, we have also reserved the necessary space to store the actual keystrokes beginning at BUFSTR.

Suppose that we used this function to enter a filename: say, "B:MYFILE.TST". After the INT 21H instruction had executed (and we had typed in this name), we would find that BUFFER contained 32 (as we had assigned), BUFLLEN contained 12 (the actual number of characters in "B:MYFILE.TST"), and beginning at BUFSTR the characters "B", ":", ..., "T", followed by various "garbage" characters. We will find below that the DOS function for opening a file requires the filename to be terminated with a zero byte, just as the string print function requires a "\$" terminator. Thus, it is prudent to go ahead and execute the instructions

```
MOV  BL,BUFLLEN      ; GET THE NUMBER OF CHARS. IN BUF.
MOV  BH,0            ; INTO BX.
MOV  BUFSTR[BX],0   ; TERMINATE THE STRING WITH A ZERO.
```

which ensure this type of termination.

DOS function 3DH is used to *open* an existing file, and is very easy to use. Into the DX register we must put the address of the filename, terminated by zero. In our example, the actual filename (unlike the buffer used for DOS function 10) begins at BUFSTR. In the AL register goes the "file access code". The file access code tells what kind of access you intend to perform. The choices for AL are:

```
0    File is read-only.
1    File is write-only.
2    File is read/write.
```

In our case, we only intend to read the file, so we might choose a file access code of 0. Thus, to open the file, our DOS call might look like this:

```
MOV  AH,3DH         ; CHOOSE DOS FILE-OPEN FUNCTION.
MOV  AL,0           ; MAKE FILE READ-ONLY.
MOV  DX,OFFSET BUFSTR ; SPECIFY FILENAME.
INT  21H
JC   ERROR         ; IF ERROR, GO TO ERROR ROUTINE.
MOV  HANDLE,AX     ; SAVE THE FILE HANDLE.
```

Unlike the simple character-I/O functions we used earlier, which always worked, it is possible for file functions to *fail* and to report an *error condition* when they are finished executing. File functions have failed if the *Carry Flag* is set on return. If the carry flag is set, the AX register contains an *error code* describing the type of error. (For our immediate purposes, the *type* of error is of no consequence, so we will not discuss this further now.) If no error has occurred, the AX register contains the *file handle*. To the human, files on the disk are designated by their *names*. For most file operations (other than open and create), however, DOS instead *numbers* the files with word values called *file handles*, and you must use file handles to specify to DOS *which* file is being operated on. There can be several files open at one time, and it would be inconvenient for DOS to have to fool with the actual filename every time a file function is requested; thus the file handle is used as a convenient means of reference. You should

immediately save the file handle in a word variable so you don't lose it.

DOS function 3FH is the *file read* function. It is capable of reading a number of bytes specified by the program (but <64K) into a buffer starting at any given position in memory. The buffer need only be as long as the length of the "records" that are to be read in. The "record length" is also loaded into the CX register, while the address of the buffer is loaded into the DX register. For our program, for example, part of our data segment might look like this:

```
REC_LEN EQU 16 ; RECORD LENGTH IS 16.
HANDLE DW ? ; STORE THE FILE HANDLE HERE.
FILE_BUF DB REC_LEN DUP (?) ; USE 16-BYTE RECORDS.
```

where FILE\_BUF is our chosen buffer. A read operation might look like this:

```
MOV AH,3FH ; SELECT DOS READ FUNCTION.
MOV BX,HANDLE ; SPECIFY WHICH FILE TO USE.
MOV CX,REC_LEN ; 16-BYTE RECORDS.
MOV DX,OFFSET FILE_BUF ; SPECIFY BUFFER LOCATION.
INT 21H
JC ERROR ; IF ERROR, EXIT.
```

Notice that we have not used any *seek* operation as described earlier. If no seek operations are performed, the file is by default accessed *sequentially* -- i.e., beginning at the beginning and progressing through the file until the end is reached. On executing this code fragment, we find the first 16 bytes of the file in our buffer. On executing it a second time, we find the second 16 bytes, etc. The file-read operation is very clever, in the sense that if there are less bytes left in the file than you have specified in CX, it will simply read a shorter record. If no error condition exists, DOS will return in the AX register the actual number of bytes read from the file. Normally, of course, the AX register will contain a 16 in our example. However, for the *last* record in the file, AX may be any number from 1 to 16. If AX is ever returned as zero -- i.e., no bytes read from file -- then the end of the file has been reached. That is, the end of file is detected by checking for AX=0.

DOS function 3EH is the *file close* operation and is the simplest operation of all:

```
MOV AH,3EH ; CLOSE THE FILE
MOV BX,HANDLE ; SPECIFIED BY BX
INT 21H
```

The operation should always be performed before ending your program.

MS-DOS FUNCTION CALLS~~PRESERVE ALL REGISTERS~~

CANNOT BE ASSUMED  
TO PRESERVE REGISTERS  
(EVEN IF YOU TEST THEM)!

EXAMPLE: IF YOU NEED TO PRESERVE THE  
AL REGISTER WHEN CALLING DOS FUNCTION  
NUMBER 9:

```
PUSH AX
MOV AH,9
MOV DX,OFFSET STRING
INT 21H
POP AX
```

SUMMARY OF SHIFT AND ROTATE

SYNTAX:

mnemonic destination, count  
1 or CL

IN PICTURES:

SHR 0 → destination → CF  
SHL, SAL CF ← destination ← 0

SAR 

ROR 

ROL CF ← 

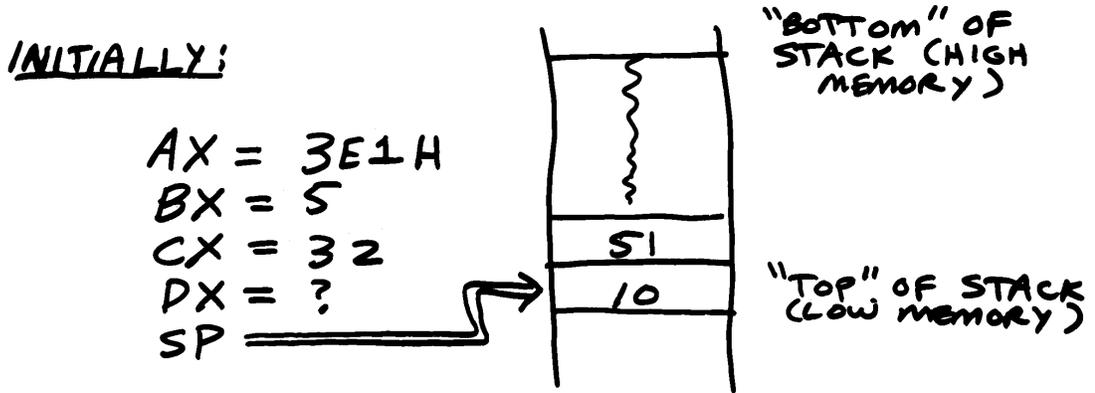
RCR 

RCL 

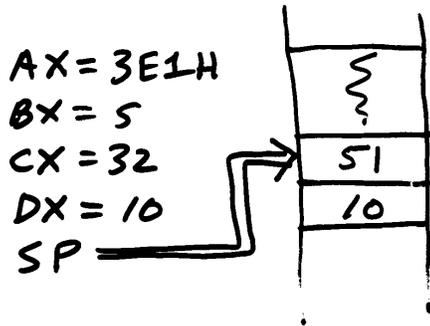
FLAGS:

OF (overflow flag)  
SET IF SIGNED OVERFLOW OCCURS.

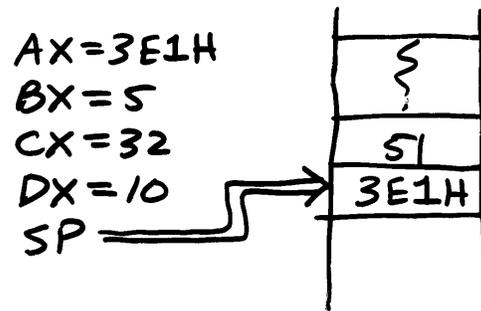
## EXAMPLE OF STACK USE



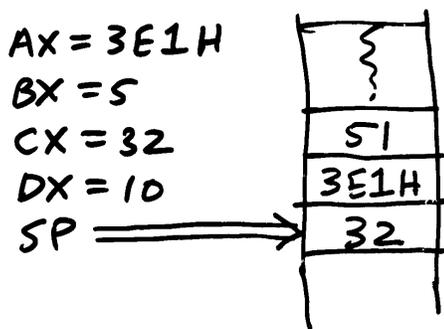
### ① POP DX



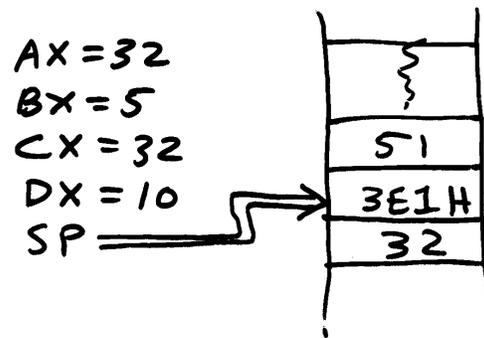
### ② PUSH AX



### ③ PUSH CX



### ④ POP AX



A RECURSIVE PROCEDURE

```

{ PASCAL VERSION: }
FUNCTION factorial (n: INTEGER): INTEGER;
BEGIN
  IF n = 0 THEN factorial := 1
  ELSE factorial := n * factorial(n-1)
END;

```

---

```

; ASSEMBLER VERSION: COMPUTE N!. PASS
; BOTH INPUT (N) AND OUTPUT (N!) IN AX.
; SET THE OVERFLOW FLAG IF UNSIGNED
; INTEGER ARITHMETIC OVERFLOWS.
FACTORIAL PROC
  CMP AX, 0                ; N = 0 ?
  JNE NOT_ZERO            ; IF NOT, CONTINUE.
  MOV AX, 1                ; OTHERWISE, 0! = 1.
  JMP SHORT DONE
; THIS IS THE RECURSIVE STEP. AT THIS
; POINT, WE KNOW N ≠ 0, SO WE MUST
; COMPUTE N! AS N * (N-1)!
NOT_ZERO:
  PUSH AX                  ; SAVE N.
  DEC AX                   ; COMPUTE N-1.
  CALL FACTORIAL           ; COMPUTE (N-1)!.
  POP BX                   ; GET BACK N.
  JO DONE                  ; QUIT ON OVERFLOW.
  MUL BX                   ; N! = N * (N-1)!.
; EXIT POINT:
DONE: RET
FACTORIAL ENDP

```

## FILE OPERATIONS

① INPUT FILENAME FROM KEYBOARD.

② OPEN EXISTING FILE

OR

CREATE NEW FILE.

③ SEEK THE PROPER POSITION IN THE FILE.

④ READ A RECORD FROM THE FILE

OR

WRITE A RECORD TO THE FILE .

⑤ CLOSE THE FILE .

↑  
REPEAT AS  
NEEDED

PSEUDO-CODE FOR HOMEWORK PROGRAM

```

; Program to do a "file dump".
    Prompt_user_to_input_filename_for_dump
    Read_filename_from_keyboard
; This file must already exist if we're to read it, so we
; must open it rather than create it:
    Open_the_file
    If error (file doesn't exist), then JMP DONE

; Main loop of program:
AGAIN:
    Read_16_byte_record_from_file
    If error (end of file, or disk read error), JMP QUIT
    Display_the_16_bytes_in_hex
    JMP SHORT AGAIN

; Normal exit at end of file
QUIT:
    Close_the_file

; Exit point of program:
DONE:

```

---

DOS FUNCTION 10: BUFFERED KEYBOARD INPUT

```

; Function 10 needs a buffer to store the input string in.
; Example:
BUFSIZE EQU 32          ; LET THE FILENAME BE A MAX. 32 CHARS.
BUFFER DB      BUFSIZE
BUFLLEN DB      ?
BUFSTR DB      BUFSIZE DUP (?)

; Sample use of DOS function 10.
    MOV  AH,10          ; FUNCTION 10.
    MOV  DX,OFFSET BUFFER
    INT  21H

; To use the input string from function 10 as a filename
; in file operations, the string
; must be terminated with a zero (just as the strings for
; function 9 are terminated with dollar signs).
    MOV  BL,BUFLLEN      ; GET THE NUMBER OF CHARS. IN BUF.
    MOV  BH,0            ; INTO BX.
    MOV  BUFSTR[BX],0    ; TERMINATE THE STRING WITH A ZERO.

```

DOS FUNCTION 3FH: READ A RECORD FROM THE FILE

; To read a "record", you need to know the "record length",  
; in bytes. The record length is not a characteristic of  
; the file -- it is chosen by the programmer and can be  
; different for every record, if desired.

REC\_LEN EQU 16

; You also need a place in memory to put the record:

FILE\_BUF DB REC\_LEN DUP (?)

; Sample use of function 3FH:

```
MOV AH,3FH          ; SELECT DOS READ FUNCTION.
MOV BX,HANDLE       ; SPECIFY WHICH FILE TO USE.
MOV CX,REC_LEN      ; 16-BYTE RECORDS.
MOV DX,OFFSET FILE_BUF ; SPECIFY BUFFER LOCATION.
INT 21H
JC  ERROR          ; IF ERROR, EXIT.
```

; On return, AX contains the *actual* number of bytes read  
; (usually equal to REC\_LEN, but possibly less).

---

DOS FUNCTION 3EH: CLOSE A FILE

```
MOV AH,3EH          ; CLOSE THE FILE
MOV BX,HANDLE       ; SPECIFIED BY BX
INT 21H
```

PSEUDO-CODE FOR 16-BYTE RECORD "DUMP"

```

; DUMP 16 BYTES FROM FILE BUFFER:
  DISP_HEX (HIGH BYTE OF BYTE_COUNT)
  DISP_HEX (LOW BYTE OF BYTE_COUNT)
  PUTCHR ( ' ' )
; LOOP ON HEX DISPLAY OF THE 16 BYTES:
  MOV CX, 16
  MOV SI, 0
HEX_LOOP:
  DISP_HEX ( FIL_BUF[SI] )
  PUTCHR ( ' ' )
  INC SI
  LOOP HEX_LOOP
  PUTCHR ( ' ' )
; NOW LOOP ON ASCII DISPLAY OF 16 BYTES:
  MOV CX, 16
  MOV SI, 0
ASCII_LOOP:
  IF FIL_BUF[SI] PRINTABLE
    THEN PUTCHR (FIL_BUF[SI])
    ELSE PUTCHR ( '.' )
  INC SI
  LOOP ASCII_LOOP
; END OF LINE:
  PUTCHR (CARRIAGE RETURN)
  PUTCHR (LINE FEED)

```

University of Texas at Dallas  
COURSE NOTES FOR CS-5330  
IBM PC ASSEMBLY LANGUAGE

**CLASS 8**Comments

1. In the previous class, I mentioned how to find (on your disk) the comments I have appended to your programs. Unfortunately, I didn't quite say what I meant, and I probably left *all* of you with the impression that I had no comments on your programs. You can determine *if* there are comments by checking for files with names ending in "\$". However, the comments are not *in* those files. **The comments are in your .ASM files (if a .AS\$ file appears on the disk), or in a file with the extension .PCW (if a .PC\$ file appears).** Sorry about that.

2. Some of you have also been victims of another oversight of mine. When you are writing an assembler program, you cannot just use any old name for variables or labels (or procedures or segments, etc.). The macro assembler has a set of **reserved words** which you are not allowed to use. (See handout.) Unfortunately, the assembler is so dumb that it may not give you an error message if you use a reserved word. Instead, it may give you dozens of errors like: "Invalid operand", "Extra characters in field", "End of file encountered", "Open segments", etc. So far, variables called "RECORD" have been a particular problem. Do not use these reserved words as names.

3. We have already discussed (several times) the two rules for proper stack use: namely, that data is popped from the stack in reverse order of the way it is pushed, and that all pushed data must be popped prior to a return. We would do well make a third rule of thumb explicit. The third rule is this: **never JMP out of a procedure.** When a procedure is called, the return address is pushed onto the stack. This return address is popped only when the RET instruction at the end of the procedure is executed. Therefore, if we get out of the procedure with a JMP rather than a RET, this return address will never be popped off of the stack, leaving a very confused program. [If you are clever, of course, you can explicitly remove the return address off of the stack with a POP instruction and then use a JMP. This is better avoided, however, since it destroys the "block structure" of your program. That is, it is like re-introducing spaghetti code in spite of using procedures to avoid it.] In most cases, a jump out of a procedure can be avoided by using *flags* to indicate error conditions on return from the procedure.

Review

In the previous class, we covered two topics.

The first topic covered was the operation and function of the *stack*, and the deeper meaning of the PUSH, POP, CALL, and RET instructions. We found that the stack is a data structure stored in the memory of the computer. Data is stored at the *top of the stack* when a PUSH is performed, and is removed from the top of the stack when a POP is performed. For this reason, data is POPped from the stack in reverse order of the way it is PUSHed onto the stack. The "top" of the

stack is actually at a lower address in memory than the "bottom" of the stack, since the stack begins in high memory and grows downward. There is a dedicated CPU register, the SP register, which points to the current top-of-stack, and the stack occupies its own dedicated segment (pointed to by the SS register). Your program selects the *size* of the stack (i.e., the amount of memory *reserved* for the stack) and the location of the stack (the value of SS and the initial value of SP). However, if your program does not take care of the details of initializing the stack, the operating system assigns a default stack of limited size for your program to use.

*Return addresses* of procedures are also stored on the stack. The CALL instruction, by which procedures are activated, is actually equivalent to the sequence of instructions

```
PUSH IP    ; Push the current address, as indicated by
           ; the Instruction Pointer register (this is
           ; not actually a legal instruction).
JMP  procedure
```

while the RET instruction, which returns from a procedure, is equivalent to

```
POP  IP    ; restore the old current address -- i.e.,
           ; jump to the address given by the top of
           ; the stack.
```

(Of course, the JMP instruction itself is equivalent to

```
MOV  IP, address of procedure      )
```

This explains why all PUSHed data must typically be POPped before RETurning from a procedure -- otherwise, the PUSHed data which has not been POPped from the top of the stack will be interpreted by RET as a return address.

We also discussed some of the DOS functions relevant to file operations. Speaking in general, we found that the following file operations need to be provided by DOS:

```
Input a string from the keyboard.
Open a file.
Create a file.
Seek a location in the file.
Read a record from the file.
Write a record to the file.
Close the file.
```

Of these, we discussed the *input string*, the *open*, the *read record*, and the *close* DOS functions.

DOS function 10 (0AH) is the buffered string input function. It is capable of reading entire strings from the keyboard, allowing use of all of the MS-DOS editing keys.

DOS function 3DH is the *file open* functions. It takes as input a *filename* (which is a string, terminated by a zero byte) and returns a

word-value called the *file handle* by which all of the other DOS functions refer to the file. That is, the once a file is opened, it is referred to by *number* rather than by name. This number is called the "file handle" and should be saved as a variable, or else it will be lost and the program won't be able to access the file again. Function 3DH is also used to specify the *file access code* -- i.e., whether the file is read-only, write-only, or read/write.

DOS function 3FH is the *read record* function. It reads a record -- i.e., a certain number of bytes -- from the file and stores the record in memory. In order to use this function, you must specify the *file handle* (that is, which file is being read), the *record size* (that is, how many bytes to read), and the address in memory at which the record is to be stored. If there are less bytes left in the file than indicated by the record size, just the number of bytes remaining in the file will be read.

DOS function 3EH is the *file close* operation. It is the easiest function to use, since it is only necessary to specify the file handle of the file to be closed.

Later in the lecture, we will discuss more of the available DOS file functions.

#### More About DOS, From the User's Standpoint

So far, from the user's standpoint, we have discussed only the crudest aspects of DOS. For example, we have discussed the format of filenames, we have discussed concepts like "default disk drive", and we have discussed some of the simplest and most useful DOS commands, such as DIR, COPY, ERASE, RENAME, TYPE, and CLS. Now, however, I'd like to briefly mention some other features of the operating system.

*WILDCARD FILENAMES.* Many of the DOS commands accept not only explicit filenames, such as

```
ERASE B:MYFILE.ASM
```

but also *wildcard file specifications* or *ambiguous filenames*. An *ambiguous filename*, rather than explicitly naming a certain file (like B:MYFILE.ASM) may match a number of different filenames. An *ambiguous filename* is one which contains either or both of the "wildcard" characters "\*" and "?". The "?" character matches any single character, while the "\*" matches a succeeding string of characters. Here are some examples of ambiguous filename use with the DOS commands:

|                        |  |
|------------------------|--|
| DIR *.TXT              | Display the directory just for all files with names like <i>something.TXT</i>  |
| ERASE *.*              | Erase every file (i.e., all files with names like <i>something.something</i> ) |
| COPY A:PROG?.* B:      | Copy files like PROG3.ASM, PROG4.asm, PROG3.BAK, etc. from drive A to drive B  |
| RENAME PROG?.* MYPG?.* | Rename files like PROG3.ASM to MYPROG3.ASM                                     |

ERASE PROG\*.\*                   Erase files like PROG3.ASM  
and PROG3.BAK, as well as  
PROG454.ASM, PROGABCD.FOO,  
etc.

Wildcards are not available with every command, but they can often be timesavers. Wildcards can also be used with some DOS functions, though not with any we have discussed so far.

*DEVICES AS "FILES"*. There are two essentially different classes of sources of input or destinations for output on computers. The two classes are the *files* and the *I/O devices*. I/O devices are, of course, things like the keyboard, the screen, the printer, the plotter, and so forth. There are many cases, however, in which we would like to ignore the distinctions between these classes and simply deal with "sources of input" or "destinations for output" in a unified way, without worrying about the nature of the source or the destination. For this reason, for each I/O device MS-DOS designates a certain "filename" by which the device can be referred to. These names can be used just like filenames in any DOS command for which they would be meaningful, and can be used with the DOS file functions. We'll see how to do this in a moment. Among the defined device names are these:

|      |   |
|------|---|
| CON  | the <i>console</i> . This is the keyboard on input and the screen on output.  |
| PRN  | the <i>printer</i> . This can only be used for output.  |
| LPTn | where n is 1, 2, or 3. There can actually be up to three printers, and they can be specifically addressed with these device names. Usually, however, there is only one printer, LPT1, which is also called PRN. |
| AUX  | the <i>serial port</i> . Used for both input and output.  |
| COMn | where n is 1 or 2. There can actually be up to two serial ports, and COM1 is alternately called AUX.  |
| NUL  | the <i>null device</i> or <i>bit bucket</i> . Output to this device is thrown away.   |

Many other *device drivers* can be *installed* in the computer and the devices accessed thereby can be referred to by means of their own particular "filenames".

Here are some sample uses of devices as "files". Rather than displaying a file on the screen with

A>>TYPE *filename*

we could "copy" the file to the screen with

A>>COPY *filename* CON

or to the printer with

A>>COPY *filename* PRN

Similarly, we could create a short file by typing the input for it at the keyboard with

```
A><COPY CON filename
```

This also works with the DOS file functions. For example, we could use the DOS *open file* function to open the file with name "CON" and use the *read record* and *write record* functions to read from the keyboard or write to the screen. This works with all of the DOS file functions discussed so far, but won't work with some that we will discuss later. For example, we will see that there is a DOS function to *erase a file* from the disk -- however, it is meaningless to erase a device name.

*I/O REDIRECTION.* Many programs take their input from the keyboard and write their output to the screen. With such programs it is easy to *redirect* the I/O to other files or devices. To redirect input so that it comes from a file or a device rather than from the keyboard, we append "<name" (where *name* is the name of the file or the device) when inputting the name of the program from the DOS command line. When redirecting output to a file or device, we append ">name". As a simple example, suppose that you wanted to print the directory of your disk on the printer: you might say

```
A>DIR >PRN
```

To output the directory instead to a file called "MYDIR.TXT", you might say

```
A>DIR >MYDIR.TXT
```

There are also DOS functions that allow redirection of any I/O (not just the console or the screen), but we will not discuss these for some time.

#### More DOS File Functions

The previously alluded-to DOS file functions which we have not yet discussed (of those that we *intend* to discuss at present) are the *create file*, *seek*, and *write record* functions. For future reference, we will also discuss the DOS *delete file* function and *rename file* function.

DOS function 3CH *creates* a file. Creation of a file is similar to opening the file, except that the file need not already exist. The create-file operation results in an entirely new file being added to the disk's directory, and then being opened for use. If the file already exists, attempting to use the create-file function results in *emptying* the file and then opening it for use. Thus, the create-file function always results in an empty file being opened. Use of the *create* function is similar to the use of the *open* function, except that as input a *file attribute* parameter is passed in CX rather than a *file access code* in AL. The *file attribute* parameter pertains to various features of MS-DOS that we have not discussed, so we will simply always set it to zero, indicating that none of the special available attributes is selected. Here is a sample use of *create*:

```

; Here is a hard-coded filename. We could, of course, input
; a filename from the keyboard using DOS function 10:
FILENAME DB 'B:TEST.TST',0
; Storage location for the file handle:
HANDLE   DW    ?
        .
        .
        .
; Sample use of DOS function 3CH:
MOV  AH,3CH
MOV  DX,OFFSET FILENAME ; select the filename.
MOV  CX,0                ; set no special file attributes.
INT  21H
JC   ERROR              ; if error, go to error routine.
MOV  HANDLE,AX          ; save the returned file handle.

```

As usual, the function returns with the *Carry Flag* set if an error occurs, and with the *file handle* in AX if no error occurs. The handle, as usual, should be saved for future use.

DOS function 42H is the *seek* function. Whenever you access files with MS-DOS, MS-DOS maintains a variable, the *file pointer*, which indicates the current byte of the file being accessed. When a file is opened or created, this pointer is automatically set to "point" at the very first byte of the file. When you read or write a record to the file, the pointer is incremented to point at the next record. This is fine (and very convenient) if you are accessing the file *sequentially* -- i.e., a record at a time, in the same order as they are stored in the file -- but it is not so convenient if you want to access records *randomly* -- i.e., in some other order. The DOS *seek* function, however, allows you to set the file pointer to any convenient value and therefore to access the records in any order you please. MS-DOS files are not limited to a size of 64K bytes, so a word-size file pointer would not be big enough for general purposes. MS-DOS therefore employs a *doubleword* file pointer. Consequently, when you use the *seek* function, the new file pointer must be specified by using *two* CPU word-size registers in conjunction. These pairs of registers *do not* specify a segment:offset -- rather, *they specify a full (signed) INTEGER\*4 file pointer*. In the case of DOS function 42H, the desired new file-pointer is passed in the CX:DX register pair, with CX containing the most significant word of the pointer and DX containing the least significant word. Actually, the *seek* function is more flexible than this. It is capable of interpreting CX:DX as either the new value of the file pointer, or as an offset from the current position or from the end of the file. The interpretation is controlled by the input value of the AL register:

```

AL=0      Use CX:DX as the new file pointer.
AL=1      Add CX:DX to the current file pointer to get
           the new file pointer.
AL=2      Move CX:DX bytes past the end of the file.

```

The *seek* function also returns the true value of the new file pointer (that is, not an offset as in AL=1 or AL=2) in the DX:AX register pair.

```

; Sample uses of DOS function 42H:

; Move to position 100 in the currently open file:
MOV  AH,42H          ; select seek function.
MOV  BX,HANDLE      ; select the file used.
MOV  CX,0            ; most significant word of
                    ; doubleword value 100.
MOV  DX,100         ; least significant word of
                    ; doubleword value 100.
MOV  AL,0           ; use CX:DX as the new pointer.
INT  21H
JC   ERROR          ; if error, then go elsewhere.

; Move to end of file -- that is, select an offset of
; zero from the end of the file:
MOV  AH,42H          ; select seek function.
MOV  BX,HANDLE      ; select the file used.
MOV  CX,0            ; most significant word of zero.
MOV  DX,0           ; least significant word of zero.
MOV  AL,2           ; select "offset from end of file"
                    ; interpretation of CX:DX.

INT  21H
JC   ERROR          ; error exit.

; Backspace: Move backwards one position in the file --
; i.e., use an offset of -1 from the current file pointer:
MOV  AH,42H          ; select seek function.
MOV  BX,HANDLE      ; select the file used.
MOV  CX,-1          ; most significant word of -1.
MOV  DX,-1          ; least significant word of -1.
MOV  AL,1           ; select "offset from current
                    ; pointer" interpretation of CX:DX.

INT  21H
JC   ERROR          ; error exit.

```

The only possibly tricky point is in the calculation of the most- and least-significant words which are to be stored in CX and DX. For example, if CX:DX=-1 then CX=-1 and DX=-1 (not CX=0 and DX=-1).

DOS function 40H is the *write record* function. Its calling sequence is identical to that of function 3FH, the *read record* function. Here is a sample use of the *write record function*:

```

; Write a record of length 128 bytes from the buffer called
; "BUFFER" to the file:
MOV  AH,40H          ; select write-record function.
MOV  CX,128          ; write a record of 128 bytes.
MOV  DX,OFFSET BUFFER ; specify where the data is.
MOV  BX,HANDLE      ; specify the file involved.
INT  21H
JC   ERROR          ; error exit.

```

As with the *read record* function, on return the AX register contains a count of the number of bytes actually written. This may be less than the number specified by the CX register if the disk has become full during the write operation. This is, of course, an error condition, even though it is not signalled by the Carry Flag being set.

There are a number of "file handles" permanently assigned by the operating system to *I/O devices* rather than to disk files. If desired, the read-record and write-record DOS functions can be used with these dedicated file handles (rather than with file handles obtained from *open* and *create* operations) to obtain I/O on various peripheral devices. I/O devices *do not* have to be explicitly "opened" and "closed" as files do. (Although, as we have already seen, there is also a series of "filenames" that could be used to "open" devices and get file handles for them.) Here are the permanently assigned file handles:

- 0            *Standard input device*, normally the keyboard.
- 1            *Standard output device*, normally the screen.
- 2            *Standard error output device* (the screen).
- 3            *Standard auxiliary device* (the serial port). This handle may be used for either input or output.
- 4            *Standard printer device*.

For example, instead of using the normal "print string" function (number 9) of DOS, we could output a string to the "standard output device" as follows:

```
; A string:
STRING DB 'This is a test string',13,10

; Alternate print-string function.
MOV  AH,40H      ; select record-output function.
MOV  BX,1        ; handle of standard output device.
MOV  DX,OFFSET STRING
MOV  CX,23       ; the length of the string.
INT  21H
```

This technique has the advantage (over function 9) that we do not need to terminate our strings with "\$", but we do have to know the length of the string. Actually, although we have specified explicitly here that there are 23 bytes in the string, there is a slightly tricky way that we can get the assembler itself to figure out the length of the string for us. Suppose that instead of the data declaration shown above we have the following:

```
; Two strings:
STRING1 DB 'This is the first test string',13,10
STRING2 DB 'This is the second test string',13,10
DUMMY  DB ?     ; a dummy declaration, not actually used.
```

Now, when we use a word like "STRING1", the assembler knows that in reality this represents an address, but also that STRING1 has the "attribute" of being a byte variable. Thus when we have an instruction like

```
MOV  AL,STRING1
```

the assembler knows that since STRING1 is a byte variable we must mean to use the *byte value stored at* STRING1 rather than the address of STRING1. Also, however, recall that we are able to perform limited arithmetic using STRING1, such as

```
MOV AL,STRING1+1
```

which means: load AL using the value of the byte *after* the STRING1. That is, STRING1+1 is interpreted as still being a byte variable, but at a slightly different address. If, however, we did something like

```
MOV AL,STRING2-STRING1
```

the assembler does something different. It subtracts the addresses of the two variables, as we might expect, but it also *cancel*s out the "byte variable" attribute. That is, STRING2-STRING1 is a *number* rather than a byte variable. In fact, it is a number exactly equal to the length of STRING1. Because of this fact, if we are tricky, we could write code sequences like this:

```
; Display STRING1 on screen and STRING2 on printer.
; First, STRING1:
MOV AH,40H           ; select record-output function.
MOV BX,1             ; handle of standard output device.
MOV DX,OFFSET STRING1
MOV CX,STRING2-STRING1 ; the length of the string.
INT 21H
; Next, STRING2:
MOV AH,40H           ; select record-output function.
MOV BX,2             ; handle of standard printer.
MOV DX,OFFSET STRING2
MOV CX,DUMMY-STRING2 ; the length of the string.
INT 21H
```

DOS function 41H is used to *delete* files from the directory of the disk. The file to be deleted *must not be open*, and is specified by giving the address of the filename, terminated by a zero. Example:

```
; Erase the file B:FOO.BAR
FILENAME DB 'B:FOO.BAR',0
.
.
.
MOV AH,41H           ; select delete file function.
MOV DX,OFFSET FILENAME ; select the file.
INT 21H
JC ERROR             ; as usual, error exit.
```

DOS function 56H, the *rename file* function, is also rather easy to use. Of course, the arguments of this function are the file's old name (which is stored in the DX register), and the file's new name (which is stored in the DI register). There is one slightly tricky aspect, in that the old name of the file is in the Data Segment (pointed to by DS), while the new name is in the Extra Segment (pointed to by ES). As an example, to change the name of B:FOO.BAR to PIFFLE.BAK, we might have the following:

```

; Old name of file:
OLDNAME DB 'B:FOO.BAR',0
; New name of file:
NEWNAME DB 'PIFFLE.BAK',0
.
.
.
; First, fiddle with the segment registers since our
; new name for the file is in the Data Segment:
    PUSH ES          ; save old ES.
    PUSH DS          ; make ES
    POP ES           ; equal to DS.
; Now, actually do the rename:
    MOV AH,56H       ; select DOS rename file function.
    MOV DX,OFFSET OLDNAME
    MOV DI,OFFSET NEWNAME
    INT 21H
; Now fix up the ES register:
    POP ES

```

There are, of course, many other DOS functions, of which we will discuss several in later lectures.

#### **ASSIGNMENT:**

1. Read in the textbook, beginning with the section titled "Macro Pseudo-Ops" in the middle of p. 39, and continuing until the *end* of section 2.6.
2. In your previous homework programs, turn all of your code which is suitable into procedures which can be individually assembled into linkable .OBJ files. In particular, you should do this *at least* for your code to capitalize a byte and for your procedure to display the hexadecimal equivalent of a byte.
3. In your previous homework programs, turn all of your code which is suitable into macros. For example, you might have macros to display a character on the screen, display a message on the screen, open and close files, etc. You might also make macros for reading and writing records. (You need not make any macros for code already in a procedure, however, unless you want to.)
4. Using these procedures and macros (almost *no* additional programming should be necessary) write a program which converts WordStar text files to ASCII text files. WordStar text differs from regular ASCII text in three ways: First, even though the ASCII standard defines only the codes from 0-127, Wordstar also uses the codes from 128-255. It does this by *setting* the most significant bit of some of the normal ASCII codes to one. Thus, we must *clear* the most significant bit of every character. Second, WordStar embeds *printing control characters* in the text. Printing control characters are codes ASCII 0-31. Thus, these bytes (not including 8, 9, 10, 12, and 13 -- the backspace, tab, line feed, form feed, and carriage return) should be "filtered" out of the text. Third, Wordstar embeds "dot commands" in the text, also to control printing. A *dot command* is a line of text which begins with a ".". Dot commands should therefore be filtered out as well. A program to do all of this goes something like this:

- a) Prompt the user to input two filenames -- one of which is an input file (containing text in WordStar format) and the other of which is an output file (containing text in ASCII format).
- b) *Open* the input file and *create* the output file.
- c) For each byte in the input file do the following:
  - i. Read the byte.
  - ii. Clear the high bit of the byte.
  - iii. If the byte is a control character (other than the exceptions listed above) go to step i.
  - iv. If the byte is the first byte of the line (i.e., the first byte after a line feed) and is a ".", then ignore the rest of the input line (until a line feed is encountered).
  - v. Write the byte to the output file.
- d) Close both files.

### Macros

As an alternative to writing "spaghetti code", we have found that our programs can be "modularized" -- and hence be made easier to write, debug, and understand -- by writing "procedures", which can be called again and again, even though the code for the procedure appears just once in the program.

Procedures do have some drawbacks, however. For one thing, there is a certain amount of overhead (in terms of execution time) involved in calling a procedure. According to Appendix C of the text, the CALL and RET instructions together (as we have been using them) require 43 clock cycles (or nearly 10 microseconds) to execute. For many purposes, this amount of time is totally negligible. For others, such as an operation carried out inside a loop which we want to iterate many times, very quickly, a "wasted" 10 microseconds is dramatically more important.

Another drawback which is probably more important for us, is that even assembler code consisting mostly of CALLs to procedures is not really that understandable. It is much better than straight spaghetti code, of course, but that isn't saying much.

Let us consider a very simple example of the latter point. Suppose that in writing our programs we get tired of continually typing in the sequence of instructions

```
MOV  DL,character
MOV  AH,2
INT  21H
```

to display a character on the screen and decide that it would be preferable to have a procedure called "PUTCHR" to display characters on the screen. Actually, we have already seen the PUTCHR procedure in the past; it was designed to accept a character in the AL register and display it:

```
MOV AL,character
CALL PUTCHR
```

Now, it can hardly escape our notice that the latter sequence is scarcely shorter or more understandable than the former. Apparently, modularity isn't that useful when we get down to instruction sequences that are too short. It would be better if we could somehow specify both the instruction sequence we want *and* the arguments for it in the same line. For example, if we could say something like

```
PUTCHR character
```

it would clearly be much better.

Actually, there is a way to do this, and it involves something called "macros". First, let's see how to define and use macros, and then let's discuss how macros differ from procedures. A macro which works as we suggested above can be defined as follows:

```
; A macro to display a character on the screen:
PUTCHR MACRO CHARACTER
    MOV DL,CHARACTER ; get ready to display the char.
    MOV AH,2         ; with DOS function 2.
    INT 21H
ENDM
```

The syntax of use of a macro is exactly as suggested above. To display a carriage-return/line-feed sequence we could just do:

```
PUTCHR 13 ; display a carriage return.
PUTCHR 10 ; display a line feed.
```

However, this isn't good only for "immediate" values: we could also do things like

```
PUTCHR AL ; display the char. in AL.
PUTCHR FOO ; display the char. in the var. FOO.
PUTCHR FOO[SI] ; display the SI-th char. in array FOO.
```

The definition of a macro differs in several ways from that of a procedure, of which the most obvious is that the PROC and ENDP pseudo-ops are replaced by MACRO and ENDM pseudo-ops. Another feature of macros is that they can have *arguments*. The string "CHARACTER" appearing twice in the macro definition is an argument of the macro. In use, the string "CHARACTER" is simply replaced by whatever appears after the word "PUTCHR". Thus, in

```
PUTCHR 13
```

the string "13" replaces each occurrence of "CHARACTER" in the body of the macro. That is, during assembly, "PUTCHR 13" is simply replaced by

```
MOV DL,13 ; get ready to display the char.
MOV AH,2 ; with DOS function 2.
INT 21H
```

Macros can also have two or more arguments, separated by commas.

In reality, the significant difference between macros and procedures is the lack of a "RET" instruction in the macro definition, and the lack of a "CALL" when the macro is actually used. Macros differ from procedures in that they are assembled "inline". That is, every place in the program that the macro is used, the assembler copies in all of the macro's code. If we use the macro called "PUTCHR" 17 times in the program, the three instructions which constitute PUTCHR will also appear 17 times in the program. The code of a procedure, on the other hand appears just once. Thus, macros do not have the execution-time overhead of procedures (no 43 clock cycles for a CALL and a RET), but they do have a memory overhead since their code is copied many times. In practice, *short* code sequences are often turned into macros, while *long* code sequences are often turned into procedures.

In the next lecture, we will see many more examples of macros and the pseudo-ops which help in defining macros.

## List of "Reserved" Words in MASM

Strictly speaking, these words are not reserved, but simply (often) misinterpreted by the assembler if used as names of variables or as labels. The assembler will usually not give you any good indication of your error and, indeed, can give you dozens of inexplicable error messages like "Open segments", "Invalid operand", "Extra characters in field", "End of file encountered", etc., if you use the reserved words as variable names or labels. This list given below is not necessarily complete.

All 8088 instruction mnemonics

|       |       |        |        |       |      |       |       |
|-------|-------|--------|--------|-------|------|-------|-------|
| AAA   | AAD   | AAM    | AAS    | ADC   | ADD  | AND   | CALL  |
| CBW   | CLC   | CLD    | CLI    | CMC   | CMP  | CMPS  | CMPSB |
| CMPSW | CWD   | DAA    | DAS    | DEC   | DIV  | ESC   | HLT   |
| IDIV  | IMUL  | IN     | INC    | INT3  | INT  | INTO  | IRET  |
| JA    | JAE   | JB     | JBE    | JC    | JCXZ | JE    | JG    |
| JGE   | JL    | JLE    | JMP    | JNA   | JNAE | JNB   | JNBE  |
| JNC   | JNE   | JNG    | JNGE   | JNL   | JNLE | JNO   | JNP   |
| JNS   | JNZ   | JO     | JP     | JPE   | JPO  | JS    | JZ    |
| LAHF  | LDS   | LEA    | LES    | LOCK  | LODS | LODSB | LODSW |
| LOOP  | LOOPE | LOOPNE | LOOPNZ | LOOPZ | MOV  | MOVS  | MOVSB |
| MOVSW | MUL   | NEG    | NOP    | NOT   | OR   | OUT   | POP   |
| POPF  | PUSH  | PUSHF  | RCL    | RCR   | REPE | REPNE | REPNZ |
| REPZ  | RET   | ROL    | ROR    | SAHF  | SAL  | SAR   | SBB   |
| SCAS  | SCASB | SCASW  | SHL    | SHR   | STC  | STD   | STI   |
| STOS  | STOSB | STOSW  | SUB    | TEST  | WAIT | XCHG  | XLAT  |
| XOR   |       |        |        |       |      |       |       |

All 8087 instruction mnemonics

|         |        |         |        |         |         |        |
|---------|--------|---------|--------|---------|---------|--------|
| F2XM1   | FABS   | FADD    | FADDP  | FBLD    | FBSTP   | FCHS   |
| FCLEX   | FCOM   | FCOMP   | FCOMPP | FDECSTP | FDISI   | FDIV   |
| FDIVP   | FDIVR  | FDIVRP  | FENI   | FFREE   | FIADD   | FICOM  |
| FICOMP  | FIDIV  | FIDIVR  | FILD   | FIMUL   | FINCSTP | FINIT  |
| FIST    | FISTP  | FISUB   | FISUBR | FLD     | FLD1    | FLDCW  |
| FLDENV  | FLDL2E | FLDL2T  | FLDLG2 | FLDLN2  | FLDPI   | FLDZ   |
| FMUL    | FMULP  | FNCLEX  | FNDISI | FNENI   | FNINIT  | FNOP   |
| FNSAVE  | FNSTCW | FNSTENV | FNSTSW | FPATAN  | FPREM   | FPTAN  |
| FRNDINT | FRSTOR | FSAVE   | FSCALE | FSQRT   | FST     | FSTCW  |
| FSTENV  | FSTP   | FSTSW   | FSUB   | FSUBP   | FSUBR   | FSUBRP |
| FTST    | FWAIT  | FXAM    | FXCH   | FFREE   | FXTRACT | FYL2X  |
| FYL2PI  |        |         |        |         |         |        |

Pseudo-Ops

|        |         |       |      |        |         |         |
|--------|---------|-------|------|--------|---------|---------|
| ASSUME | COMMENT | DB    | DD   | DQ     | DT      | DW      |
| ELSE   | END     | ENDIF | ENDP | ENDS   | EQU     | EVEN    |
| EXTRN  | GROUP   | IF    | IF1  | IF2    | IFB     | IFDEF   |
| IFDIF  | IFE     | IFIDN | IFNB | IFNDEF | INCLUDE | LABEL   |
| NAME   | ORG     | PAGE  | PROC | PUBLIC | RECORD  | SEGMENT |
| STRUC  | SUBTTL  | TITLE |      |        |         |         |

Operators (NOT in alphabetical order)

|        |      |      |        |     |       |      |
|--------|------|------|--------|-----|-------|------|
| MOD    | SHL  | SHR  | AND    | OR  | XOR   | NOT  |
| EQ     | NE   | LT   | GT     | LE  | GE    | SEG  |
| OFFSET | TYPE | SIZE | LENGTH | PTR | SHORT | THIS |
| HIGH   | LOW  |      |        |     |       |      |

Registers (NOT in alphabetical order)

|    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| AH | BH | CH | DH | AL | BL | CL | DL | AX | BX | CX | DX | SI | DI |
| BP | SP | CS | DS | ES | SS |    |    |    |    |    |    |    |    |

## DOS FUNCTION REFERENCE SHEET (INT 21H)

**NOTE:** *filenames* are specified in ASCIZ format. They are ASCII character strings, terminated with a zero byte. All file functions (3CH and above) return with CF set on error.

**AH=1H** **Input one character from the keyboard (with echo).**

OUTPUT: AL=ASCII code of the character.

**AH=2H** **Display one character on the screen.**

INPUT: DL=ASCII code of the character.

**AH=5H** **Print one character on the printer.**

INPUT: DL=ASCII code of the character.

**AH=8H** **Input one character from the keyboard (no echo).**

OUTPUT: AL=ASCII code of the character.

**AH=9H** **Display a character string on the screen.**

INPUT: DS:DX=*segment:offset* of the start of the string (which is terminated with "\$").

**AH=0AH** **Input a character string from the keyboard.**

INPUT: DS:DX=*segment:offset* of a buffer. If *N* is the maximum allowable length of the input string, then the buffer must be *N*+2 bytes long and the first byte must contain the value *N*.

OUTPUT: Location *N*+1 of the buffer contains the actual number of input characters, and the string is in the buffer beginning at location *N*+1.

**AH=3CH** **Create a new file (or clear an old one).**

INPUT: DS:DX=*segment:offset* of the ASCIZ filename.  
CX=*attribute word* of the new file (normally 0).

OUTPUT: AX=file handle.

**AH=3DH** **Open an existing file.**

INPUT: DS:DX=*segment:offset* of the ASCIZ filename.  
AL=*file access code* (0=read-only, 1=write-only, 2=read/write).

OUTPUT: AX=file handle.

**AH=3EH** **Close an open file.**

INPUT: BX=file handle.

**AH=3FH** **Read a record from a file (or a device).**

INPUT: BX=file handle. CX=record size. DS:DX=start of the buffer into which data will be read.

OUTPUT: AX=number of bytes actually read into buffer.

**AH=40H** **Write a record to a file (or a device).**

INPUT: Same as with AH=3FH.

OUTPUT: AX=number of bytes actually written to disk.

**AH=41H** **Delete a file from the disk.**

INPUT: DS:DX=*segment:offset* of ASCIZ filename.

**AH=42H** **Seek a given location in a file.**

INPUT: BX=file handle. CX:DX=doubleword offset in bytes. AL=interpretation of offset (0=from beginning of file, 1=from current position, 2=from end of file).

OUTPUT: DX:AX=doubleword pointer after move.

**AH=56H** **Rename a disk-file.**

INPUT: DS:DX=*segment:offset* of the old ASCIZ filename. ES:DI=*segment:offset* of the new ASCIZ filename.

**ASSIGNMENT:**

1. Read in the textbook, beginning with the section titled "Macro Pseudo-Ops" in the middle of p. 39, and continuing until the *end* of section 2.6.
2. In your previous homework programs, turn all of your code which is suitable into procedures which can be individually assembled into linkable .OBJ files. In particular, you should do this *at least* for your code to capitalize a byte and for your procedure to display the hexadecimal equivalent of a byte.
3. In your previous homework programs, turn all of your code which is suitable into macros. For example, you might have macros to display a character on the screen, display a message on the screen, open and close files, etc. You might also make macros for reading and writing records. (You need not make any macros for code already in a procedure, however, unless you want to.)
4. Using these procedures and macros (almost *no* additional programming should be necessary) write a program which converts WordStar text files to ASCII text files. WordStar text differs from regular ASCII text in three ways: First, even though the ASCII standard defines only the codes from 0-127, Wordstar also uses the codes from 128-255. It does this by *setting* the most significant bit of some of the normal ASCII codes to one. Thus, we must *clear* the most significant bit of every character. Second, WordStar embeds *printing control characters* in the text. Printing control characters are codes ASCII 0-31. Thus, these bytes (not including 8, 9, 10, 12, and 13 -- the backspace, tab, line feed, form feed, and carriage return) should be "filtered" out of the text. Third, Wordstar embeds "dot commands" in the text, also to control printing. A *dot command* is a line of text which begins with a ".". Dot commands should therefore be filtered out as well. A program to do all of this goes something like this:
  - a) Prompt the user to input two filenames -- one of which is an input file (containing text in WordStar format) and the other of which is an output file (containing text in ASCII format).
  - b) *Open* the input file and *create* the output file.
  - c) For each byte in the input file do the following:
    - i. Read the byte.
    - ii. Clear the high bit of the byte.
    - iii. If the byte is a control character (other than the exceptions listed above) go to step i.
    - iv. If the byte is the first byte of the line (i.e., the first byte after a line feed) and is a ".", then ignore the rest of the input line (until a line feed is encountered).
    - v. Write the byte to the output file.
  - d) Close both files.

## AMBIGUOUS FILE NAMES

### "WILDCARD" CHARACTERS :

? MATCHES ANY CHARACTER .  
 \* MATCHES ANY STRING .

### EXAMPLES :

DIR \*.TXT — DISPLAY A DIRECTORY  
 OF ALL FILES WITH NAMES LIKE  
 anything.TXT .

ERASE \*.\* — ERASE ALL FILES .

COPY A:PROG?.\* B: — COPY ALL  
 FILES WITH NAMES LIKE PROG3.ASM,  
 PROG4.ASM, PROG3.BAK, etc.  
 FROM DRIVE A: TO DRIVE B: .

RENAME PROG?.\* MYPG?.\* —  
 RENAME FILES LIKE PROG3.ASM  
 TO MYPG3.ASM .

ERASE PROG\*.\* — ERASE FILES  
 LIKE PROG3.ASM, PROG3.BAK,  
 PROG454.ASM, 'PROGABCD.FOO', etc. .

"FILE NAMES" OF DEVICES

CON The "console" — the screen on output, the keyboard on input.

PRN The printer.

LPT1 } Printers 1, 2, and 3 .  
LPT2 } (LPT1 = PRN)  
LPT3 }

AUX The serial port.

COM1 } Serial ports 1 and 2 .  
COM2 } (COM1 = AUX)

NUL The "bit bucket" — throws away any output.

EXAMPLES:

- A> COPY filename CON displays the file on the screen.
- A> COPY filename PRN prints the file on the printer.
- A> COPY CON filename creates the file, reading lines from the keyboard.

SAMPLE DOS FILE FUNCTION USAGECREATE FILE TEST.TST :

```

FILENAME DB 'B:TEST.TST', 0
NEWNAME  DB 'PIFFLE.BAK', 0
HANDLE   DW ?
        :
        :
        :
MOV AH, 3CH ; DOS FUNCTION 3CH
MOV DX, OFFSET FILENAME
MOV CX, 0 ; FILE ATTRIBUTES OFF.
INT 21H ; CALL DOS.
JC ERROR ; ERROR EXIT.
MOV HANDLE, AX

```

DELETE FILE TEST.TST :

```

MOV AH, 41H ; DOS FUNCTION 41H
MOV DX, OFFSET FILENAME
INT 21H ; CALL DOS.
JC ERROR ; ERROR EXIT.

```

RENAME FILE TEST.TST TO PIFFLE.BAK :

```

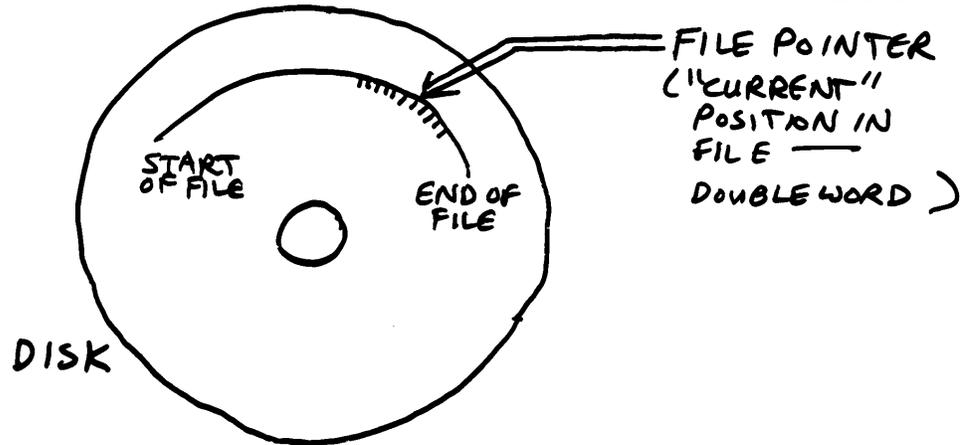
PUSH ES ; SAVE ES
PUSH DS } ; ES ← DS
POP ES }
MOV AH, 56H ; DOS FUNCTION 56H
MOV DX, FILENAME ; OLD NAME.
MOV DI, NEWNAME ; NEW NAME.
INT 21H ; CALL DOS.
POP ES ; RESTORE ES.

```

MORE SAMPLE DOS USAGE

WRITE RECORD — WORKS SAME WAY AS  
READ-RECORD FUNCTION .

SEEK :



THE SEEK FUNCTION MODIFIES THE FILE-  
 POINTER, USING :

- CX:DX — DOUBLEWORD (SIGNED 32-BIT INTEGER)  
 "OFFSET".
- AL — "METHOD OF MOVEMENT":
  - AL=0 : CX:DX = NEW FILE POINTER.
  - AL=1 : CX:DX = OFFSET FROM CURRENT FILE POINTER .
  - AL=2 : CX:DX = OFFSET FROM END OF FILE .

## PREDEFINED FILE-HANDLES

- 0 STANDARD INPUT (THE KEYBOARD).
- 1 STANDARD OUTPUT (THE SCREEN).
- 2 STANDARD ERROR OUTPUT (SCREEN).
- 3 STANDARD AUXILIARY DEVICE (THE SERIAL PORT FOR BOTH INPUT AND OUTPUT).
- 4 STANDARD PRINTER

### SAMPLE USE:

```
; "PRINT STRING" FUNCTION (ALTERNATIVE TO DOS 9).
STRING DB 'THIS IS THE STRING TO PRINT', 13, 10
      :
```

```
MOV AH, 40H ; DOS WRITE RECORD FUNCTION
MOV BX, 1 ; HANDLE 1 = SCREEN.
MOV DX, OFFSET STRING
MOV CX, 29 ; "RECORD" LENGTH =
           ; = STRING LENGTH.
INT 21H
```

MACROS

```

name MACRO argument list
    :
;    CODE
    :
    ENDM

```

EXAMPLE:

```

; MACRO TO DISPLAY A CHARACTER ON THE SCREEN.

```

```

PUTCHR MACRO CHARACTER
    MOV     DL, CHARACTER
    MOV     AH, 2
    INT     21H
    ENDM
    :

```

```

PUTCHR 'A'      ; DISPLAY AN "A"
PUTCHR 13      ; DISPLAY A (CR)
PUTCHR 10      ; DISPLAY LINE FEED
PUTCHR VAR[SI] ; DISPLAY SIXTH
                ; CHARACTER OF THE
                ; BYTE-ARRAY VAR

```

University of Texas at Dallas  
 COURSE NOTES FOR CS-5330  
 IBM PC ASSEMBLY LANGUAGE

## CLASS 9

Comments

1. On the use of = in place of EQU. Because it's not clear to me from the homework problems if many of you actually understood the distinction between these two, let me say a couple of words about it. EQU gives a constant value a symbolic name, so that wherever the constant might be used in the source program, the symbolic name can be used instead. For example, the statement

```
MYCONSTANT EQU 5
```

means that instead of using the constant 5 in the source program we can instead use the name MYCONSTANT. Once this EQU is used, the value of MYCONSTANT is fixed: it will always be replaced by 5, wherever the assembler finds it. The = pseudo-op is used in much the same way,

```
MYCONSTANT = 5
```

except that the value of MYCONSTANT is not fixed in a certain sense. The value of MYCONSTANT can be changed by some pseudo-ops and, in particular, by later = pseudo-ops. All such changes in the value of MYCONSTANT occur at assembly time, since this is when the assembler interprets the pseudo-ops. Thus, it does not really make sense to talk about the *program* being able to change the value of MYCONSTANT, since the program can only affect quantities that exist at run-time. This distinction is not trivial, since misunderstanding it can result in a program that does not work. For one thing, the = pseudo-ops take effect in the order in which the assembler encounters them, and *not* in the order in which the program would encounter them as it was executing. For example, if we had

```
MYCONSTANT = 5
      MOV  CX,15
AGAIN:
MYCONSTANT = MYCONSTANT+1
      LOOP AGAIN
```

this would result in MYCONSTANT having the value 6 (since the assembler only encounters 2 '='s) rather than 20.

2. Please **DO NOT** ever (on purpose) turn in a program that will crash. At the very least, if you feel compelled to do this, *remove the .EXE file from the disk*, so that I can't unwittingly run the program. If your program crashes my computer it puts me in quite a bad mood. (Of course, accidents do happen, and may be forgiven, but I don't have to like it.)

3. In the homework assignment, cross out part 2: "convert your code into procedures". We won't get to this today. However, you still have to do parts 1, 3, and 4.

4. In the previous lecture I was asked where macros must be defined in the program and I thoughtlessly answered "anywhere". In fact, macros should be defined before they are used. Thus, the beginning of the program is a good place for them.

#### Review

In the previous class we discussed various features of MS-DOS that we had skipped past previously. One DOS feature discussed was ambiguous filenames. We discovered that there are two "wildcard" characters which can often be included in filenames to indicate that some DOS command is to work on an entire set of files rather than on just one explicitly-named file. The legal wildcard characters are the "?", which matches any single character, and the "\*", which matches any string. Thus, for example,

```
DIR *.TXT
```

would display a directory of just the .TXT files on the disk.

Another new DOS feature was the accessing of devices by means of their "filenames". Each I/O device attached to the system has a "name" (or possibly several names) like a file has a filename. More interesting, devices can be used as sources or destinations for many DOS commands that we have previously only seen used on files. For example, we could use the DOS "COPY" command to copy a file to a file, or a file to a device, a device to a file, or a device to a device. Some of the defined device names are CON (the screen and keyboard), PRN (the printer), and NUL (the "bit bucket"). These names can be used not only with DOS commands, but also with DOS file functions. The DOS "open file" function, for example, can be used to prepare a file for I/O, or to prepare a device for I/O, and the DOS "read record" and "write record" functions can subsequently actually perform this I/O.

I/O redirection was the final DOS feature discussed. With I/O redirection, output intended for one file (or device) can be redirected to another, or input intended to come from a file (or device) can be redirected to another. So far (and for the immediate future) only redirection of output intended for the screen, or input from the keyboard is considered. With most programs that send output to the screen, the output can be redirected to a file (or device) by appending ">filename" to the DOS command line. Thus,

```
DIR >MYFILE
```

stores the directory display that would normally appear on the screen instead in the file MYFILE. Similarly, programs that normally take input from the keyboard, like EDLIN or DEBUG, can instead take input from a file by appending "<filename" to the DOS command line.

We also discussed further DOS file functions. Five functions were discussed: 3CH, the *create file* function; 42H, the *seek* function; 40H, the *write record* function; 41H, the *delete file* function; and 56H, the *rename file* function. Because the handout reference sheet discusses all of these functions, there is no need to elaborate on them further. I would like to emphasize once again, however, that you should never delete an open file.

We found that as well as built-in filenames for devices, DOS also has a set of predefined file handles for devices. The predefined file handles can be used in any situation in which a real file handle is used, except that there is never any need to *open*, *create*, or *close* the "files" associated with these handles. The predefined handles are: 0 (the *standard input device*), 1 (the *standard output device*), 2 (the *standard error device*), 3 (the *standard auxiliary device*), and 4 (the *standard printer*).

Finally, we began discussing *macros*. Like a *procedure*, a macro is a sequence of instructions grouped together and given a name. The sequence of instructions forming a macro is executed by giving the name of the macro, much like a procedure is executed, except that a macro is not *CALLed*, nor does a macro end with a *RETurn*. The reason macros are not *called* is that they are executed *inline* -- that is, wherever the assembler finds a use of the macro, it simply sticks all of the source code for the macro into the program at that point. Here is an example of a definition of a macro to get a character from the keyboard and store it in AL:

```
GETCHR    MACRO
          MOV  AH,8
          INT  21h
          ENDM
```

Actually using the macro in an assembler program would look something like this:

```
.
.
.
AGAIN:
    GETCHR                ; get a keyboard character.
    CMP  AL,27           ; ESC?
.
.
.
```

When this code is assembled, the assembler "expands" the macro, giving an actual code sequence of

```
.
.
.
AGAIN:
    MOV  AH,8
    INT  21h
    CMP  AL,27           ; ESC?
.
.
.
```

Macros have the advantage over procedures that they execute more quickly, but the disadvantage that they use more memory. They have, however, one other very crucial advantage -- they can take arguments.

Here is an example of a macro to display a character on the screen. This macro takes an argument, namely the character to be displayed:

```
PUTCHR    MACRO    CHARACTER
          MOV      DL,CHARACTER
          MOV      AH,2
          INT      21h
          ENDM
```

When the macro is "expanded", every occurrence of the variable (CHARACTER) is replaced by the actual argument. Thus,

```
PUTCHR    'a'
```

expands to

```
MOV      DL,'a'
MOV      AH,2
INT      21H
```

Indeed, a more reasonable definition of the GETCHR macro would be

```
GETCHR    MACRO    CHARACTER
          MOV      AH,8
          INT      21H
          MOV      CHARACTER,AL
          ENDM
```

since we would then be able to specify a destination for the character other than AL. The ability of macros to accept arguments is extremely important, since it is the key to writing assembly language programs that are partially understandable. Therefore, we will now spend a lot of time discussing macros.

### More About Macros

The best way of seeing the value and use of macros is probably by example. Therefore, let's see some more examples of macros.

*PRINT STRING.* Like the PUTCHR macro, which displays a character, we might introduce a DISPLAY macro, which displays a string. Such a macro might go something like this:

```
DISPLAY    MACRO    MESSAGE
          MOV      AH,9
          MOV      DX,OFFSET MESSAGE
          INT      21H
          ENDM
```

In use, we could write, for example,

```
SIGN_ON    DB    'THIS IS A TEST',13,10,'$'
          .
          .
          .
          DISPLAY    SIGN_ON
```

which would then display the indicated message.

"NEAR" *CONDITIONAL JUMPS*. As we have discovered, all of the conditional jump instructions which we must use so frequently are *SHORT* jumps. That is, their destination addresses must be within 128 bytes of the address at which the jump is executed. In many cases this is all right, but when we start getting "relative jump out of range" errors, we usually must deal with them very clumsily. This difficulty can be avoided by defining macros that *act like* conditional jumps, but make *NEAR* jumps rather than *SHORT* jumps. Let us explicitly consider just the JC instruction -- all other conditional jumps being very similar. A JC instruction looks like

```
JC    address
```

where the address operand is a nearby label. We will define a macro JMPC whose use

```
JMPC address
```

is almost identical to that of JC, but results in a NEAR jump. (I say "almost" identical, since it is actually more flexible in ways we will discuss in later lectures.) We would like the JMPC macro to expand something like this:

```
JNC NO_CARRY
JMP  address
NO_CARRY:
```

There is a slight problem with this in that if we use the JMPC macro several times in our program, the label NO\_CARRY will appear many times and the assembler will give us error messages to the effect that the label NO\_CARRY is "multiply defined". Fortunately, MASM provides us with a pseudo-op to overcome this difficulty. The LOCAL pseudo-op specifies that a certain symbol is "local" to the macro. That is, that the symbol has a meaning only inside of the macro, and can be re-used as desired in the rest of the program. The general syntax of the LOCAL pseudo-op is

```
LOCAL    symbol list
```

and the LOCAL statement must appear as the *first line* of the macro definition (after MACRO itself). For instance, in the macro

```
HELLO    MACRO    MESSAGE, CHARACTER
          LOCAL   SAM, JANET, MORNING
          .
          .
          .
          ENDM
```

the symbols SAM, JANET, and MORNING (as well as the arguments MESSAGE and CHARACTER) are all local to the macro HELLO. In our example, the JMPC macro would therefore be defined as

```
; a "NEAR" conditional jump-on-carry macro:
JMPC MACRO ADDRESS
```

```

LOCAL NO_JUMP
JNC   NO_JUMP
JMP   ADDRESS
NO_JUMP:
ENDM

```

In practice, one could also define such near-jump macros for all of the other commonly used conditional jumps and use them in place of a true conditional jump whenever necessary. Thus, there would be a lot of macros like JMPc, JMPnc, JMPz, JMPcxz, etc.

A different approach to simply defining a JMPccc macro for each Jccc instruction is to try and define a single macro that handles all possible cases. This might be desirable since there are 31 different conditional jump instructions, and we don't want to define 31 different JMPccc macros. We can cover all possible cases with just one macro by employing the & operator. The & operator is used in macros to concatenate or join together strings. Consider the following macro:

```

; "Universal" conditional jump macro:
JP   MACRO      JP_TYPE, ADDRESS
LOCAL      NO_JUMP, YES_JUMP
J&JP_TYPE YES_JUMP
JMP       SHORT NO_JUMP
YES_JUMP:
JMP       ADDRESS
NO_JUMP:
ENDM

```

Here are some sample invocations of JP:

```

JP   C,AGAIN      ; jump, if CF=1, to label AGAIN.
JP   NZ,AGAIN     ; jump, if ZF=0, to AGAIN.
JP   CXZ,AGAIN    ; jump, if CX=0, to AGAIN.
JP   AE,AGAIN     ; jump on unsigned >= to AGAIN.

```

To see how this macro works, consider what happens when "JP C,AGAIN" is expanded. In this case, JP\_TYPE is the string "C", so J&JP\_TYPE creates the string "JC", and the macro expands as

```

JC   YES_JUMP
JMP  NO_JUMP
YES_JUMP:
JMP  AGAIN
NO_JUMP:

```

The JP macro is not quite so efficient as the JMPccc macros, since it generates one additional JMP instruction. However, it might be more convenient to use. Of course, in practice, we might never encounter a JNO instruction which jumps more than 128 bytes (indeed, we may never encounter a JNO at all), so is probably wiser to use JP to handle this case than to invent a JMPNO instruction. Similar comments apply to some of the other Jccc's as well.

One thing that makes macros useful is that they can be re-used. They do not have to be re-typed for every new program. The reason for this is that the macro assembler has the capability during assembly of

"reading in" or "including" source code files other than the one explicitly being compiled. To see how this works, let us suppose for the sake of argument that we have a file on our disk (called "MACROS.LIB") which contains nothing but the definitions of various macros: for example, it might contain the definitions of JMPC, JMPZ, JMPO, JMPS, JMPNC, JMPNZ, JMPNO, and JMPNS, and nothing else (that is, no data segment, no code segment, no procedures, no "END" statement, etc.). For the sake of argument we will also suppose that we are using MASM to assemble a completely different file, called "PROG6.ASM". For the programs we have written so far, the assembler simply reads in the source file (PROG6.ASM), assembles it, and writes the assembled program (PROG6.OBJ) to disk. Suppose, however, that PROG6.ASM contained the line

```
INCLUDE  MACROS.LIB
```

The INCLUDE pseudo-op specifies that an additional source-code file (in this case, the file is MACROS.LIB, containing macro definitions) should now be assembled *before* continuing with the assembly of PROG6.ASM. That is, we can effectively use such separately written source code as if it were actually contained in our own program. If the file contains macros, we can effectively use the macros, even though we have not defined them in our program but have defined them at some previous time. The bottom line, therefore, is that every time you think of a useful macro definition you should go and add it to MACROS.LIB. If you do this, you can use the macro in any future program without further thought. Indeed, in a very reasonable sense, we can think of macro definitions as permanent programmer-defined *extensions* to the instruction set of the microprocessor. With this interpretation in mind we will simply begin to use PUTCHR, JMPC, JMPNC, etc., and any other macros we define *as if* they were actual instructions of the processor. This is not to say that every macro you define should be added to the library. Many macros are defined on the spur of the moment as a time-saving aid just for a particular program, and would not be at all useful for any other program ever written. However, all generally useful macros should be put into the library.

[Actually, for reasons explained in the text, and which there is no reason to go into now, you should use the statements

```
IF1
  INCLUDE MACROS.LIB
ENDIF
```

to include the macro file, rather than just INCLUDE by itself. You should simply put these three lines at the beginning of your template program.]

*SOME FILE-USE MACROS.* Macros are ideal for simplifying the use of many DOS functions (such as file functions), because most of the code required for using a DOS function simply sets up the various parameters needed. Indeed, we might define macros for all of the commonly used file functions: *open*, *create*, *seek*, *read*, *write*, and *close*. For demonstration purposes, we will consider just the read, write, and close functions, with open, create, and seek being left as exercises for the student. [This remark is only half-facitious, since any time

spent now on these macros will surely be saved when later programs are written.]

The close-file operation is the most obvious, since only one argument -- the file handle -- is required:

```
; Macro to close a file:
CLOSE    MACRO    HANDLE
          MOV      AH,3EH          ; DOS CLOSE-FILE FUNCTION.
          MOV      BX,HANDLE      ; SELECT THE FILE TO CLOSE.
          INT      21H
          ENDM
```

The read-record and write-record functions are hardly less obvious, except that we must also provide arguments for a memory buffer for the record, the length of the record, and any error exits. Here is how we might write the write-record macro:

```
; Macro to write a record to a file:
WRITE    MACRO    HANDLE,BUFFER,RECLEN,ERROR_EXIT
          MOV      AH,40H          ; DOS WRITE-RECORD FUNCTION.
          MOV      BX,HANDLE      ; SELECT THE FILE TO WRITE.
          MOV      CX,RECLEN      ; SELECT RECORD LENGTH.
          MOV      DX,OFFSET BUFFER ; BUFFER POSITION.
          INT      21H
          JMPC     ERROR_EXIT     ; EXIT ON WRITE-ERROR.
          CMP      AX,RECLEN      ; CHECK # OF BYTES WRITTEN.
          JMPNE   ERROR_EXIT     ; EXIT IF TOO FEW BYTES.
          ENDM
```

Note the use of the JMPcc macros to provide error exits. The read-record macro differs only in that we are likely to want *separate* exits for disk error than for the end-of-file:

```
; Macro to read a record from a file:
READ     MACRO    HANDLE,BUFFER,RECLEN,END_FILE,ERROR_EXIT
          MOV      AH,3FH          ; DOS READ-RECORD FUNCTION.
          MOV      BX,HANDLE      ; SELECT THE FILE TO READ.
          MOV      CX,RECLEN      ; SELECT RECORD LENGTH.
          MOV      DX,OFFSET BUFFER ; BUFFER POSITION.
          INT      21H
          JMPC     ERROR_EXIT     ; EXIT ON READ-ERROR.
          CMP      AX,0           ; END OF FILE?
          JMPE    END_FILE       ; IF YES, EXIT.
          ENDM
```

One thing that is not obvious until we begin to use these macros is just how simple some parts of our program begin to appear. Let us take a simple example, in which we want to write a program to copy one file to another, much like the DOS COPY command. Such a program would appear as below, using the macros we have already defined. Although it is a rather inefficient way of proceeding in MS-DOS, we will use records of length 1 (that is, we will copy the file byte by byte) and hence will call our file buffer "CHARACTER":

```

; Program to copy one file to another, byte-by-byte:

; Data:
input      dw  ?      ; storage for handle of input file.
output     dw  ?      ; storage for handle of output file.
character  db  ?      ; buffer for file I/O.
error_message db 7, 'Disk I/O error.',13,10,'$'
          .
          .
          .
; CODE TO OPEN THE INPUT FILE AND CREATE THE OUTPUT FILE.
          .
          .
          .
again:     read      input,character,1,quit,error
          write     output,character,1,error
          jmp      again

error:     display   error_message
quit:     close     input
          close     output

```

Once again, it is apparent that programs written with macros can be much easier to understand than programs in which every instruction is explicitly written out.

#### The IF Pseudo-ops

With the programs we have written so far, all of the code in the program is assembled when MASM is run, with the result that a .OBJ object file is created. There are instances, however, when we would prefer that some of our code is *not* assembled, or is assembled under some conditions and not others.

For one thing, if our program does not work as expected, we might want to embed instructions in our program that are assembled only so long as we are in the *debugging* phase of writing the program. When the program is finally totally debugged, as it stands now, we have to go back through the program and remove (or put semi-colons in front of) all such debugging statements. It would be better if the assembler was simply smart enough, somehow, to not assemble such code even though it is actually in the program.

Another case of some interest is if we are writing a program that we expect to run on several different computers. These different computers may have slightly different features or require the program to be written in a slightly different way. Thus, it would be nice to have the assembler actually assemble the program slightly differently in these two cases rather than have to maintain several separate versions of the same program.

These things can be done with several different pseudo-ops, which we will collectively call the "IF" pseudo-ops. With the IF pseudo-ops we can specify that blocks of code are to be assembled under some conditions (as evaluated at assembly time) and not under others. IF pseudo-ops are *not* instructions of the CPU and *do not* affect program control at run-time.

There are a number of different IF pseudo-ops, although we will explicitly discuss only the ones called "IF" and "IFE". The general syntax of "IF" is:

```

IF    numerical expression
      .
      .
      .
;    CODE!
      .
      .
      .
ENDIF

```

The code between IF and ENDIF is assembled only if the numerical expression operand evaluates to something other than zero. As a simple example, for debugging purposes we might include the following in our program:

```

DEBUGGING EQU 1
      .
      .
      .
      IF DEBUGGING
      PUTCHR '*'
      ENDIF

```

As long as DEBUGGING is not set to zero, the PUTCHR macro is assembled and (at runtime) displays an "\*" on the screen. Once we are no longer debugging, we can set DEBUGGING to zero before assembling the program and no such code will be assembled.

The IFE pseudo-op operated identically, except that the code between itself and ENDIF is assembled only if the numerical expression is zero. As an example of this, we might write a macro which is a variation of the PUTCHR macro in that it sends output either to the screen or to the printer depending on whether (at assembly time) one of its arguments is zero or 1:

```

DO_CHAR MACRO CHARACTER, DEVICE
      IFE DEVICE          ; IF DEVICE=0, USE SCREEN.
      MOV AH,2
      ENDIF
      IF DEVICE          ; IF DEVICE=1, USE PRINTER.
      MOV AH,5
      ENDIF
      MOV DL,CHARACTER
      INT 21H
      ENDM

```

Thus, to send an "A" to both the screen and to the printer, we might include lines like this in our code:

```

DO_CHAR 'A',0
DO_CHAR 'A',1

```

There is a second form of the IF-clause (or IFE) with a syntax like this:

```

    IF  numerical expression
      .
      .
      .
;   CODE!
      .
      .
      .
    ELSE
      .
      .
      .
;   CODE!
      .
      .
      .
    ENDIF

```

which assembles the code between IF and ELSE if the expression is non-zero and assembles the code between ELSE and ENDIF if the expression is zero. Thus our DO\_CHAR macro could have been slightly simplified as

```

DO_CHAR  MACRO  CHARACTER, DEVICE
          IFE  DEVICE          ; IF DEVICE=0, USE SCREEN.
          MOV  AH,2
          ELSE          ; IF DEVICE=1, USE PRINTER.
          MOV  AH,5
          ENDIF
          MOV  DL,CHARACTER
          INT  21H
          ENDM

```

### String Instructions

For recreational purposes, we will now completely change the subject and discuss some of the built-in string-manipulation features of the 8088 microprocessor.

The 8088 can manipulate strings of either byte-values or word-values, and these strings can be up to 64K in length. There are five built-in general types of functions:

1) The *load* instructions LODSB and LODSW load an element of a string into the accumulator. LODSB loads a byte from a byte string into AL, while LODSW loads a word from a word string into AX.

2) The *store* instructions STOSB and STOSW store the value in the accumulator as an element of a string.

3) The *move* instructions MOVSB and MOVSW copy the elements of one string to another string.

4) The *compare* instructions CMPSB and CMPSW compare the elements of two strings.

5) The *scan* instructions SCASB and SCASW scan a string for a value equal to the accumulator.

There are two key features of the string instructions which make them different from the otherwise similar normal MOV and CMP instructions. The first feature is that appropriate pointer registers are automatically incremented or decremented for string operations. We have seen before that in order to access the elements of a character or word array, we would normally use the BX, SI, or DI registers to "point" at the proper place in the string. Then, after whatever processing we desire is finished, the pointer register must be explicitly updated to point to the next string element. With the string instructions, however, this updating is automatic. The second key feature is that all of the string instructions can be *automatically repeated* for a certain number of iterations or until a certain condition of the ZF flag holds. The features together allow for some very simple, fast, and powerful string manipulations.

Except for those instructions that implicitly use the accumulator as source or destination for string elements, source strings are pointed to by the DS:SI *segment:offset* register-pair, and destination strings are pointed to by the ES:DI register-pair. Since all strings are of length less than 64K, the DS and ES registers can never change in a string operation; the SI (Source Index) and DI (Destination Index) registers are, however, automatically updated (i.e., incremented or decremented by 1 or 2) to point to the next string element. As an example of how this works, let us consider two strings, *both* in the data segment, called "SOURCE" and "DESTINATION":

```
SOURCE      DB  'THIS IS A TEST STRING'
DESTINATION  DB  '                '
```

In order to set up DS:SI to point to the source string, we need merely do this:

```
MOV  SI,OFFSET SOURCE
```

while to set up ES:DI we need:

```
PUSH DS          ; SET ES TO EQUAL DS.
POP  ES
MOV  DI,OFFSET DESTINATION
```

To specify whether SI and DI are automatically incremented or automatically decremented, we must set (or reset) a flag in the CPU. This flag, the *Direction Flag*, or DF, specifies the "direction" (i.e., positive or negative) in which string operations move. String instructions are *auto-incrementing* (which we want, having given the starting addresses rather than ending addresses of the strings) if DF is cleared by the instruction

```
CLD          ; set to auto-increment.
```

The STD instruction, on the other hand, sets *auto-decrement* mode. We can *move* one byte from the source string to the destination with the instruction

```
MOVSB    ; move one byte.
```

Thus, this instruction would copy the "T" from SOURCE to DESTINATION. Similarly, we can move one word from source to destination with

```
MOVSW    ; move one word.
```

Since we have used the CLD instruction to set auto-increment mode, the former instruction increments both SI and DI by one, so that they point to the next byte of the string. The latter instruction increments both SI and DI by two, so that they point to the next word of the string.

The CMPSB and CMPSW string comparison instructions are very similar. They perform a CMP on the current string elements (bytes or words) and then update SI and DI to point to the next elements of their respective strings. There is one tricky point, however. The book states, apparently correctly, that Intel has foolishly implemented these instructions differently from the CMP instruction. The flags are indeed set *as if* a subtraction has been performed. However, the hypothetical subtraction is

```
source-destination
```

rather than destination-source, as in CMP. Usually this is of no consequence since only checks for equality are typically performed, but it is a nuisance.

The LODSB and LODSW instructions load the accumulator with an element from the string source. The STOSB and STOSW instructions store the value from the accumulator into the destination string. The SCASB and SCASW instructions COMPare the value of the accumulator with the current element from the destination string. Like the CMPSB and CMPSW instructions, the comparison is backwards:

```
accumulator-destination.
```

All of these instructions, of course, either auto-increment or auto-decrement SI and/or DI by the appropriate amounts.

The real power of the string instructions is felt only when the auto-repeat feature is used. Any of the string instructions may be repeated a fixed number of times, with or without an additional check of the ZF flag as a termination condition. In order to automatically repeat a string instruction, we need to use a *repeat prefix* with the instruction. There are three repeat prefixes, REP, REPE (or REPZ), and REPNE (or REPNZ). These prefixes repeat the specified instruction the number of times indicated by the CX register. REPE and REPNE additionally check the value of the ZF flag at the end of each iteration and terminate if the flag is set (REPE) or not set (REPNE). In the case of our sample strings, SOURCE is 21 characters long, and the instructions

```
MOV  CX,21
REP  MOVSB
```

would copy all of SOURCE into DESTINATION. *Filling* a block of memory with a given byte (or word) is also easy. For example, to fill DESTINATION with "A" rather than " " we could do

```
MOV  CX,21
MOV  AL,'A'
REP  STOSB
```

The instructions SCASB and SCASW, in conjunction with REPE, are used to *scan* a string for a given value. For instance,

```
MOV  CX,21
MOV  AL,' '
```

```
REPE SCASB
```

would halt when it located the first byte of DESTINATION which is a space (in this case, the first character). On the other hand,

```
MOV  CX,21
MOV  AL,' '
```

```
REPNE SCASB
```

would search for the first non-space (in this case, it would halt at the end of the string, since there is no non-space character).

The final major use of these operations is in comparing two strings for equality (or inequality). The instructions

```
MOV  CX,21
REPNE CMPSB
```

would halt at the first byte of the strings which didn't match. Thus, if the instruction halts at the *end* of the strings, they must be the same.

We will discuss the use of these string operations further in the next lecture when we have an actual use for them.

CONDITIONAL "NEAR" JUMPS

```

; JUMP "NEAR" ON CARRY:
; JMP C      MACRO  ADDRESS
;             LOCAL NO_JUMP
;             JNC   NO_JUMP
;             JMP   ADDRESS ; JUMP IF CARRY
NO_JUMP:
        ENDM

```

---

```

; "UNIVERSAL" CONDITIONAL JUMP:
; JP      MACRO  JP_TYPE, ADDRESS
;             LOCAL NO_JUMP, YES_JUMP
;             J & JP_TYPE YES_JUMP
;             JMP   SHORT NO_JUMP
YES_JUMP:
        JMP   ADDRESS
NO_JUMP:
        ENDM

```

EXAMPLES:

```

JP  C, AGAIN ; SAME AS "JMP C, AGAIN"
JP  NZ, AGAIN ; JUMP TO AGAIN IF ZF=0.
JP  CXZ, AGAIN ; " " " " CX=0.
      ⋮

```

AGAIN:

```

; Macro to close a file:
CLOSE    MACRO    HANDLE
          MOV      AH,3EH          ; DOS CLOSE-FILE FUNCTION.
          MOV      BX,HANDLE      ; SELECT THE FILE TO CLOSE.
          INT      21H
          ENDM

; Note the use of the JMPccc macros to provide error exits below:
; Macro to write a record to a file:
WRITE    MACRO    HANDLE,BUFFER,RECLen,ERROR_EXIT
          MOV      AH,40H          ; DOS WRITE-RECORD FUNCTION.
          MOV      BX,HANDLE      ; SELECT THE FILE TO WRITE.
          MOV      CX,RECLen     ; SELECT RECORD LENGTH.
          MOV      DX,OFFSET BUFFER ; BUFFER POSITION.
          INT      21H
          JMP      ERROR_EXIT     ; EXIT ON WRITE-ERROR.
          CMP      AX,RECLen     ; CHECK # OF BYTES WRITTEN.
          JMPNE   ERROR_EXIT     ; EXIT IF TOO FEW BYTES.
          ENDM

; Macro to read a record from a file:
READ     MACRO    HANDLE,BUFFER,RECLen,END_FILE,ERROR_EXIT
          MOV      AH,3FH          ; DOS READ-RECORD FUNCTION.
          MOV      BX,HANDLE      ; SELECT THE FILE TO READ.
          MOV      CX,RECLen     ; SELECT RECORD LENGTH.
          MOV      DX,OFFSET BUFFER ; BUFFER POSITION.
          INT      21H
          JMP      ERROR_EXIT     ; EXIT ON READ-ERROR.
          CMP      AX,0           ; END OF FILE?
          JMPE    END_FILE       ; IF YES, EXIT.
          ENDM

; Program to copy one file to another, byte-by-byte:

; Data:
input    dw    ?                ; storage for handle of input file.
output   dw    ?                ; storage for handle of output file.
character db  ?                ; buffer for file I/O.
error_message db 7,'Disk I/O error.',13,10,'$'
.
.
.
; CODE TO OPEN THE INPUT FILE AND CREATE THE OUTPUT FILE.
.
.
.
again:   read    input,character,1,quit,error
          write   output,character,1,error
          jmp     again

error:   display error_message
quit:    close   input
          close   output

```

## STRING CONVENTIONS

SOURCES OR DESTINATIONS : CAN BE

ACCUMULATOR AL or AX

or

DS: SI  $\Rightarrow$  SOURCE STRING  
["SI" stands for Source Index]

or

ES: DI  $\Rightarrow$  DESTINATION STRING  
["DI" stands for Destination Index]

---

DIRECTION OF STRING OPERATIONS :

DF = Direction Flag of CPU

CLD instruction clears DF  
Sets AUTO-INCREMENT

STD instruction sets DF  
Sets AUTO-DECREMENT

University of Texas at Dallas  
COURSE NOTES FOR CS-5330  
IBM PC ASSEMBLY LANGUAGE

## CLASS 10

Comments

1. On backup copies of your programs. Those of you who are not overly familiar with microcomputers (and some who are) are probably not aware just how unreliable and risky it is to keep *just one* copy of valuable programs. Neither the computer hardware nor the diskette medium itself is reliable enough for this practice to succeed in the long run. Moreover, you yourselves are not immune from human error. To protect yourself in both cases, you should make backup copies of your work. [Case history, for anyone who doesn't believe me: Two class members this past weekend found that files on their disks had been destroyed by the computer. In one case, this included not only all of the previous homework assignments, but the macro library as well.] Imagine what it would be like to have the computer destroy a program you had been working on for three weeks (such as the final project), including a macro library you had been building for seven weeks! The solution for this problem is to make backup copies of your programs on a *separate* disk. Indeed, you might consider keeping *two* backups (and an "original"), using all three disks in rotation, so that you always have a backup of both the "current" version and of the previous version of your program. Backup copies can be made in several ways, including the DOS COPY command and the DOS program BACKUP. In either case, it is a good idea to put a "copy protect" tab on your master disk before making a backup, to prevent careless errors on your part (such as copying the backup onto the original rather than vice-versa).

Review

In the last class, we continued our discussion of macros by presenting a number of examples of macros that could actually be of use in practice. We introduced a macro with syntax

```
DISPLAY  message
```

where *message* is the name of a string-variable defined with DB, which could display a string on the screen. We introduced macros of the form

```
JMPccc  address
```

where *ccc* represents a "condition" (like *C*, *NC*, *Z*, *NZ*, *CXZ*, etc., as used in conditional jump instructions). These macros are functionally equivalent to the conditional jump instructions *Jccc*, except that they are NEAR rather than SHORT jumps. Since it would be painful to define *JMPccc* macros for each of the 31 *Jccc* instructions, we also introduced a "universal" conditional NEAR jump macro,

```
JP  ccc, address
```

We also saw several macros that are useful in DOS file operations. We had

```

CLOSE      handle
WRITE     handle,buffer,reclen,error_exit
READ      handle,buffer,reclen,end_file,error_exit

```

which were used for their respective DOS functions.

In order to define some of these macros, we introduced new pseudo-ops. We saw the

```

LOCAL      symbol list

```

macro, which defined a list of symbols that were *local* to the macro. We saw the *concatenation operator &*, which *joined* two strings together. We also saw the IF, IFE, ELSE, and ENDIF pseudo-ops, which were used like

```

IF[E]      expression
           .
           .
           .
;   code.
           .
           .
           .
[ELSE]
           .
           .
           .
ENDIF

```

The assembler assembles the code following IF only if the *expression* operand is non-zero (and the code following IFE only if *expression* is zero). If ELSE is present, the code between ELSE and ENDIF is assembled only if the condition fails (*expression*=0 for IF, *expression*<>0 for IFE).

The most significant fact about macros, other than the fact that they allow a syntax similar to that found in higher-level languages, is that they can be re-used for many programs. Typically, we store the macros in a special file (called, say, MACRO.LIB), and we *include* the macros in any program we write by putting the lines

```

IF1
  INCLUDE MACRO.LIB
ENDIF

```

at the beginning of our source-code files -- indeed, in our template program, if we wish. In this way, as time goes on, it is as if we gradually add new, sophisticated instructions to the processor.

We also briefly discussed the *string* instructions available on the 8088 microprocessor. While we did not go into this in tremendous detail (it is included in the next reading assignment), we did find out the following things:

Strings consist of sequences of bytes or words, stored in the Data Segment or in the Extra Segment. All string operations operate

implicitly on the accumulator (AL or AX) or on the *source string* (pointed to by DS:SI) or on the *destination string* (pointed to by ES:DI). Since these operands are implicit in the instructions, no string instruction requires any arguments.

String operations either auto-increment (move upwards in memory) or auto-decrement (move downwards in memory). The prevailing direction is controlled by the DF flag in the CPU. The instruction CLD sets auto-increment mode, while the instruction STD sets auto-decrement mode.

There are five basic types of string instructions: The *load* instruction loads the accumulator from the source string; the *store* instruction stores the accumulator to the destination string; the *scan* instruction compares the accumulator to the destination string; the *move* instruction moves one element from the source string to the destination string; and the *compare* instruction compares one element of the source and destination strings. The *scan* and *compare* instructions are irritating, in that these subtract the destination from the source, rather than the source from the destination as in CMP. Each of the instructions works on just *one* string element, but they auto-increment or auto-decrement SI and/or DI in preparation for the next string operation.

However, there are ways to automatically repeat each string operation. A string instruction is automatically repeated with the REP, REPE, or REPNE *prefixes*. If a repeat prefix is used, the designated string operation is repeated the number of times specified by the CX register. However, for the REPE prefix, this is done only as long as the Zero Flag is set. With the REPNE prefix, the string instruction is repeated only while the Zero Flag is *not* set.

The basic uses of the string instructions are these: block moves, block fills, checking a string for a given character, and comparing two strings.

#### **ASSIGNMENT:**

1. Read section 3.8 and do the problems for chapter 3. Read chapter 5.
2. In your previous homework programs, adapt all suitable code into procedures which you can *individually* assemble. This includes at *least* your code for capitalizing the character in AL and for displaying the hexadecimal characters corresponding to a byte.
3. Finish up any pending homework assignments, since next week we will start on the mid-term project.

#### Separate Assembly of Procedures

We have already seen how our programming tasks are simplified by storing macros in a special "library" file for re-use by new programs. It would be nice if we could do something similar for procedures, since they are usually even more complex and perform even more sophisticated tasks than macros.

In fact, we obviously could reuse procedures in the same way as we re-use macros -- by "including" them in our programs with the INCLUDE macro. However, there is a much better way. The problem with including them is that the source code may be large and may take the assembler a long time to assemble. If we could somehow use the already-assembled forms of the procedures, it would be much better since object code is much more compact than source code, and would therefore take less time and memory to utilize.

In fact, we can do exactly this: We can separately compile our procedures and our "main" program and simply "link" them together with the LINK program. In this way, we not only are able to have instant access to all of our old procedures, but we are also able to cut down on our assembly time.

Here is a "template" for a file containing a procedure to be individually assembled:

```

PUBLIC      name
CODE SEGMENT
  ASSUME    CS:CODE
name PROC   FAR
  .
  .
  .
;  CODE!
  .
  .
  .
  RET
name ENDP
CODE ENDS
END

```

Several features of this definition are worth discussion. For one thing, the pseudo-op PUBLIC is used to declare to the assembler that the name of the procedure (in this case, *name*) is *global* -- that is, that it is meaningful even after assembly of this file. Normally, names used in the source code have meaning only at assembly time and completely disappear after that. In real terms, the .OBJ file created by the assembler contains no references to these names. The PUBLIC pseudo-op, however, allows names to be embedded in the .OBJ file in such a way that the linker can link the procedure with another separately assembled procedure that calls it.

Another interesting feature is that there is no data segment and the ASSUME pseudo-op contains no "DS:DATA" part. While it is possible, with some effort, to include things like this, it is not a good practice to allow separately assembled procedures to directly access (that is, to access by name) items in the data segment. Generally, separately assembled procedures should access data *indirectly*, through registers and through pointers contained in registers. Otherwise, you would have to make sure that every program which used this procedure was set up in a certain way, to contain the proper names. We will see in the next lecture how arguments are typically passed to procedures *via* the stack.

A third feature is that, like a main program, we have declared our procedure to be FAR, rather than NEAR. While it is possible to separately assemble NEAR procedures, there are problems involved which are similar to those mentioned above in regard to data segments. FAR procedures have the disadvantage that there is slightly more memory and time overhead in CALLs and RETs, but the advantage that (being in a separate code segment) they do not subtract from the allotted 64K code of the main program.

A final point is that we end the procedure with "END" rather than with "END name". The *name* part of an END statement specifies to the assembler the address at which the *main program* should start executing. For a separately assembled procedure, there is no such relevant starting address (since it is not a main program), so the starting address is omitted.

Here is an example of a procedure which can be separately assembled:

```

        public  disp_dec
code    segment
        assume  cs:code
; This is a procedure to convert a binary number in the AL
; register to decimal ASCII characters and send them to
; the console. A binary-to-decimal conversion works by
; dividing by ten (saving the remainders), until the number
; being divided is zero. The remainders (which are all
; 0-9) can be directly converted into decimal digits ('0'-
; '9'). The only problem is that the digits are calculated
; in reverse order (least significant to most significant)
; and must therefore be stored rather than immediately
; displayed as they are calculated. The stack is ideal for
; this temporary storage. We push a "fence" value (decimal
; 10) onto the stack first, so that when the digits ('0'-
; '9') are eventually popped and displayed we will know
; when to quit.
disp_dec proc far
        mov     bx,10          ; push a "fence" onto the stack.
        push   bx
dec_loop:
; first, divide AX (=AL) by BL (=10).
        mov     ah,0          ; prepare for a division.
        div    bl            ; divide al by 10.
; result of division is AL, and the remainder (0-9) is AH.
        add     ah,'0'        ; convert remainder to ASCII.
        mov     dl,ah         ; prepare to push it.
        push   dx
        cmp     al,0          ; all converted?
        jnz    dec_loop
; now, pop and display all of the digits, quitting when
; the "fence" value of 10 (decimal) is reached.
dec_disp:
        pop     dx            ; get a digit.
        cmp     dl,10         ; fence?
        jz     dec_done      ; if so, quit.
        mov     ah,2          ; display the digit.
        int    21H

```

```

        jmp     short dec_disp
dec_done:
        ret
disp_dec endp
code    ends
end

```

Aside from the fact that this *does* illustrate a procedure that can be individually assembled, the only point of interest about it is the use of the stack for temporary storage and the use of a "fence" value to indicate completion rather than a count for looping on the computed digits. We will, however, have a use for this routine, as we will discover later in the lecture.

Next, now that we know how to *define* procedures for separate assembly, we must find out how to *use* separately assembled procedures. We can freely use separately assembled procedures in our programs, except that we must take the step of declaring to the assembler that we are doing so. This is accomplished with the EXTRN (for: EXTeRNally defined symbol) pseudo-op. The EXTRN pseudo-op has a syntax like this:

```
EXTRN    list of symbol:type
```

Each externally defined symbol (along with its *type*) must appear in an EXTRN statement before it is used. The *type* of the symbol specifies whether the symbol is the name of a byte or word variable, or whether it is a near or far procedure. To use our sample procedure, we might put the line

```
EXTRN    DISP_DEC:FAR
```

near the beginning of our main program, before any CALLs to DISP\_DEC have actually appeared in the source-code. Thus,

```
EXTRN    DISP_DEC:FAR
.
.
.
MOV  AL,145
CALL DISP_DEC
```

would display the string "145" on the screen, even though DISP\_DEC was not otherwise defined in the program.

One other thing is necessary, however, and that is that the way we link our program must be changed. Suppose that our main program is called PROGRAM (with .ASM for source code and .OBJ for assembled code), and that DISP\_DEC is contained in a file called DECIMAL (.ASM or .OBJ). Normally, we would link PROGRAM with a DOS command like this:

```
LINK PROGRAM;
```

which would create PROGRAM.EXE. To link in DECIMAL.OBJ as well, we would have to modify this to

```
LINK PROGRAM+DECIMAL;
```

which would create PROGRAM.EXE. Note that this is not the same as "LINK DECIMAL+PROGRAM;", which would try to create a file called DECIMAL.EXE.

For now, we will deal with individually assembled procedures in just this way. In later lectures, we will see that it is possible combine all of our individual .OBJ files into more compact "libraries" of procedures, and thereby avoid the clutter and confusion of having so many files on our disk.

### More About Keyboard Input

Let's extend by a little our knowledge of how to get keyboard input.

In all of our previous work, we used DOS functions 1, 8, and 10 to read the keyboard for us. One feature of these functions which is sometimes useful is that they check for various control keys like CTRL-C, CTRL-S, CTRL-P, etc. to be entered, in which case some special actions are taken. For example, CTRL-C exits the program. In many cases, however, this is not what we want. We may want to use these particular keys to perform some other function, or we may not want the user to be able to break out of the program with CTRL-C. Fortunately, there is another DOS function, *function 7*, which takes care of part of this problem. Function 7 is exactly like function 8, except that it reads these special keys just mentioned as if they were like any other key, and performs no special checks. Thus, we could use function 7 anywhere we now use function 8, and simply not worry about losing control of the computer if the user presses a CTRL-C.

Another flaw in the way we have dealt with the keyboard is that whenever we want a keyboard character DOS takes control away from the program and just waits for a key to be pressed. This is fine if we know that nothing else of value can be accomplished by the program until it receives some input. It may be, however, that we don't want to wait for a key to be pressed -- rather we might want to have the program go on and do something else in the meantime, just checking the keyboard occasionally to see if a character is ready for processing. For this, DOS has provide *function number 0BH*, the keyboard status function. *The keyboard status function simply returns a value indicating whether or not a character is ready at the keyboard. Here is a sample use:*

```
MOV  AH,0BH           ; SELECT KEYBOARD STATUS FUNCTION.
INT  21H
CMP  AL,0             ; KEY PRESSED?
JNZ  YES_KEY_PRESSED
```

The output values of this function are indicated by the the above code. If there is a character ready, AL contains 0FFH on return, whereas if no character is ready, AL contains 0. Unfortunately, this function checks for CTRL-C and exits from the program if it is pressed.

The standard ASCII character set does not contain many of the keys that actually appear on the IBM PC's keyboard. Therefore, pressing these keys cannot result in the return of a standard ASCII code. Examples of such keys are the function keys F1-F10 and the arrow keys.

On the IBM PC, pressing these keys results instead in *extended ASCII codes* being returned by DOS functions 1, 7, and 8. An extended ASCII code consists of *two bytes* rather than one. Since each DOS call results in just *one byte* being returned in the AL register, we need to call DOS *twice* to retrieve an extended code. We know to do this since the first character returned is always a zero. The most useful extended codes are outlined on p. 230 of the text.

Let's see an illustration of this. One thing the table on p. 230 tells us is that the extended code for a down-arrow is 50H. If the down-arrow key was being pressed on the keyboard, a call to DOS using function 8 would return a zero, and a second call would return a 50H (a "P"). We would know that this was a down-arrow rather than a "P" simply because a real "P" would not be preceded by a zero. For example, code that can distinguish entry of a down-arrow key from that of a real "P" might look like this:

```

mov  ah,8          ; read the keyboard.
int  21H
cmp  al,0          ; if al=0, an extended code follows.
jnz  real_ASCII   ; otherwise, it's a real ASCII char.
mov  ah,8          ; since al was zero, we must go ahead
                        ; and read the extended code.
int  21H

```

Of course, since we would probably like to read the keyboard using a macro, this jumping around to different addresses is rather inconvenient. As an alternative, we might return the normal ASCII characters as values 0-127, and return extended codes as values 128-255. Here is a macro for that:

```

; A better keyboard-reading macro. It returns extended
; key codes as values 128-255, rather than requiring a
; second use of getchr after a returned value of 0 has been
; detected.
getchr macro character
    local real_ascii
    mov     ah,7          ; read the keyboard.
    int     21h
    cmp     al,0          ; extended character?
    jnz     real_ascii   ; if not, then okay.
    mov     ah,7          ; if so, get the
    int     21h          ; extended code.
    or     al,80H        ; convert to 128-255.
real_ascii:
; if the argument of the macro is blank, just leave the
; character in AL.
    ifnb   <character>
        mov     character,al
    endif
endm

```

Other than the extended characters, the only real novel point about this macro is the use of the IFNB conditional assembly pseudo-op. This pseudo-op works just like IF and IFE, except that instead of testing a numerical condition, the code between IFNB and ENDIF (or ELSE, if present) is assembled if the *string argument* (which is enclosed in

angular brackets as delimiters) is *not blank*. There is a similar IFB pseudo-op which assembles only if the string argument *is blank*. Thus, for example, "GETCHR" would get a character into AL, while "GETCHR BL" would get a character into BL. (This macro also incorporates the "improvement" of using DOS function 7 rather than function 8.) For example, if we used the statement

```
getchr
```

then on pressing a "P" we would find that AL contained 50H (the ASCII code for "P"); on pressing a down-arrow, we would find that AL=0D0H (=80H+50H).

### The ANSI Driver

Now that we have seen how to get some additional use out of the keyboard, it would also be nice to get some additional control over the screen. For example, it would be nice to be able to easily clear the screen from within our programs, or to move the cursor to any part of the screen we desire. We might want to *highlight* some of our characters (displaying them more brightly than the surrounding characters), or underline them, or make them blink, or display them in reverse video, or modify the color of the characters. While the IBM PC does not have an unlimited repertoire of display modes, many of these things can be easily accomplished. Which of these features is available on a given IBM PC (or PC clone) depends on the *video hardware* with which the machine is equipped. If we use a standard software interface, however, as discussed below, we can for the most part ignore such hardware details.

This universal software interface is the so-called ANSI device driver, which is contained in a file called ANSI.SYS. While this software cannot overcome any hardware limitations of the computer (for example, it cannot display color with a *monochrome adapter* or underlining with a *color adapter*), it nevertheless allows us to write our programs *as if* all features are available. If the features we elect to use (such as underlining) don't happen to be available, our programs will still work -- they just won't underline anything. Some features, such as cursor movement, are controlled by software (rather than hardware) and are always available.

In order to use the features of the ANSI driver, the driver must be "installed" in the computer. This is done *automatically* when the computer is turned on or is reset, but *only* if certain conditions are satisfied. In order to insure that ANSI.SYS is automatically installed, or to determine if ANSI.SYS *is* installed, you must examine a file called CONFIG.SYS. There are two cases. If there *is* no file CONFIG.SYS, you should create such a file using EDLIN; CONFIG.SYS should contain the single line

```
DEVICE=ANSI.SYS
```

On the other hand, if CONFIG.SYS already exists, you should examine it with the command

```
TYPE CONFIG.SYS
```

If CONFIG.SYS contains the line mentioned above, then ANSI.SYS is already installed. Otherwise, you should add this line to CONFIG.SYS with EDLIN. In either case, if you have modified CONFIG.SYS you should reset the computer by simultaneously holding down *ctrl-alt-del*. This information applies only to IBM PCs and close clones; on some machines, the ANSI driver may be always available.

We will assume from now on that ANSI.SYS is installed.

As mentioned earlier, the ANSI driver allows you to control the screen and the cursor in several new ways. It does this by means of *escape sequences* output to the screen. An escape sequence is a string of characters, the first of which is the escape character (ASCII code 27 decimal). In the case of the ANSI driver, the second character in the escape sequence is always a left-square-bracket, "[". The left-bracket is followed by (optional) numerical parameters, with the string finally being terminated by an alphabetic character that indicates which function is to be performed. This can hopefully be made clearer by an example:

```
; Example of how to use the ANSI driver to clear the screen
; and then to move the cursor down 15 lines:
clear_screen db      27, '[2J$'
move_down    db      27, '[15B$'
.
.
.
    display  clear_screen
    display  move_down
```

Here, "displaying" the string `clear_screen` causes the screen to clear, and "displaying" the string `move_down` causes the cursor to move down 15 lines. Both strings begin with `<ESC>` and "[", are followed by one numerical parameter, and end with an alphabetic character. (The dollar signs are, of course, used only by the `display` macro to terminate the strings and have nothing to do with the ANSI driver.) The "J" terminating alphabetic character indicates that an erasure function is desired, while the "B" character indicates a "move cursor down" operation. The numerical parameter "2" indicates a full-screen erasure, while the numerical parameter "15" indicates that the cursor is to move down 15 lines. Notice that the numerical parameters are actually given by their ASCII decimal equivalents. In many cases, numerical parameters are optional, defaulting to one (or some other obvious value). Thus, if the "15" had been omitted in `move_down`, the cursor would only move down by one line.

(See handout for full list of supported commands.)

Unfortunately, this ANSI driver does not implement many of the useful features in the ANSI standard. However, it does implement several commands we can get a lot of use out of. Because each command has its own mnemonic, it is helpful for us to access the ANSI commands through macros rather than through the various escape sequences. I have created a file of macros whose use is outlined in the handout. This list is posted on my door, and the macro library (ANSI.LIB) is also available to anyone who wants to bring a disk down to my office. Let's see how some of these macros might work.

The easiest macros are, of course, for those ANSI functions with no arguments, like ED and EL. The macro for ED looks like this:

```
; Erase entire display:
ed      macro
        esc_bracket
        putchr '2'
        putchr 'J'
        endm
```

where we have used both the previously defined macro PUTCHR a new macro ESC\_BRACKET

```
; This macro does the ESC[ part:
esc_bracket macro
        putchr 27
        putchr '['
        endm
```

which starts the escape sequence by sending the escape left-bracket sequence to the ANSI driver.

Macros for functions with one argument are somewhat more complex, since several new factors need to be taken into account. One is that the argument is optional and may be omitted. This is not a problem, since we have already seen how arguments can be omitted from macros if we test them with the IFNB conditional assembly pseudo-op. The second difficulty is slightly more serious, in that for a macro like

```
CUU AL
```

in which the number of rows to move upwards is given by the AL register, we actually want to send the ASCII decimal equivalent (which could be up to three characters) to the ANSI driver. This is where the DISP\_DEC procedure defined earlier comes in, since it has exactly this function. Therefore, we include the following in ANSI.LIB

```
        extrn  disp_dec:far
; This macro is used to ease use of the decimal display
; procedure. Sample:
;      DECIMAL                ; DISPLAY DECIMAL NUMBER IN AL.
;      DECIMAL 46              ; DISPLAY 46.
;      DECIMAL FOO[SI]        ; DISPLAY DECIMAL NUMBER FOO[SI].
decimal macro  number
        ifnb  <number>
            mov    al,number
        endif
        call   disp_dec
        endm
```

to let us easily use DISP\_DEC. We also find it convenient to have a macro like:

```

; This macro does just the initial one-argument prefix.
; First it sends the ESC[, then it sends the ASCII decimal
; representation of the argument (if present). That is,
; ONE_ARG sends the sequence ESC[#.
one_arg macro    argument
    esc_bracket
    ifnb    <argument>
        decimal argument
    endif
endm

```

With tools like these, construction of, say, a CUU macro is very easy. In fact, we have

```

; This moves the cursor up.  Examples:
;     cuu    2                ; move up by two lines.
;     cuu                    ; move up by one line.
cuu    macro    distance
    one_arg    distance
    putchr    'A'
endm

```

```

; This moves the cursor down.  Examples:
;     cud    2                ; move down two lines.
;     cud                    ; move down one line.
cud    macro    distance
    one_arg    distance
    putchr    'B'
endm

```

etc. Two-argument functions are turned into macros in much the same way.

From now on, let us assume that all of the macros described in the handout exist and can be used freely. What might a program using these macros look like? For the purposes of demonstration, here is a short sample program:

```

; Program to demonstrate the use of the ANSI driver and
; macros. This program will:
; 1. Erase the screen.
; 2. Move to the bottom of the screen, and print in
;    high-intensity mode "This is the status line.
;    Status= TEST", where "TEST" is blinking.
; 3. Move to the top of the screen and input characters
;    in "typewriter mode" (and low intensity) until a
;    ctrl-C is pressed. Since we now use DOS function
;    7 in our GETCHR macro, we must explicitly test for
;    this.
ED                ; erase the screen.
CUP 25,1          ; move to bottom line.
SGR 0             ; first, turn off all attributes.
SGR 1             ; then, turn on high intensity.
DISPLAY MSG1     ; "This is the status line. Status= ".
SGR 5             ; turn on blinking.
DISPLAY MSG2     ; "TEST".
SGR 0             ; turn off all attributes.

```

```
        CUP 1,1          ; home the cursor.
AGAIN:  GETCHR           ; get a keyboard character.
        CMP AL,3         ; ctrl-C?
        JE  DONE
        PUTCHR AL        ; display the character.
        JMP SHORT AGAIN
DONE:
```

While no work will be assigned which explicitly requires the use (or mastery) of the ANSI driver, these commands are available for use and are rather fun to play with.

**REFERENCE SHEET FOR THE ANSI DRIVER AND ITS MACROS**

Note 1. The ANSI screen driver must be installed by putting the line  
*DEVICE=ANSI.SYS*

into a file called *CONFIG.SYS* on the boot disk. To use the ANSI macros once the driver is installed, the assembly language source file must contain the line

*INCLUDE ANSI.LIB*

and the object file must be linked to the file *DECIMAL.OBJ*.

Note 2. Below, # stands for any decimal number.

| <u>ANSI<br/>MNEMONIC</u> | <u>ESCAPE<br/>SEQUENCE</u>                 | <u>MACRO USE</u>   | <u>DESCRIPTION OF ACTION</u>  |
|--------------------------|--|--|---|
| CUP                      | ESC [#;#H<br>ESC [#H<br>ESC [;#H<br>ESC [H | CUP <i>row,column</i><br>CUP <i>row</i><br>CUP <i>,column</i><br>CUP | Move cursor to specified row and column. Default is 1 for either omitted parameter. |
| CUU                      | ESC [#A<br>ESC [A                          | CUU <i>distance</i><br>CUU   | Move cursor specified distance up. Default=1.                                       |
| CUD                      | ESC [#B<br>ESC [B                          | CUD <i>distance</i><br>CUD   | Same as CUU, but down.  |
| CUF                      | ESC [#C<br>ESC [C                          | CUF <i>distance</i><br>CUF   | Same, but to the right instead of up or down.                                       |
| CUB                      | ESC [#D<br>ESC [D                          | CUB <i>distance</i><br>CUB   | Same, but to the left.  |
| HVP                      | ESC [#;#f<br>ESC [#f<br>ESC [;#f<br>ESC [f | HVP <i>row,column</i><br>HVP <i>row</i><br>HVP <i>,column</i><br>HVP | Same as CUP.  |
| SCP                      | ESC [s                                     | SCP  | "Save" cursor position.   |
| RCP                      | ESC [u                                     | RCP  | "Recall" cursor position.   |
| ED                       | ESC [2J                                    | ED   | Erase the screen.   |
| EL                       | ESC [K                                     | EL   | Erase to end of line.   |
| SGR                      | ESC [#m                                    | SGR <i>attribute</i>   | Turn on specified char. attribute. (Note 3.)  |
| SM                       | ESC [=#h                                   | SM <i>mode</i>   | Set specified screen mode. (See note 4.)  |
| RM                       | ESC [=#l                                   | RM <i>mode</i>   | Reset specified mode.   |

Note 3. On an IBM PC, the allowed character attributes are: 0=all attributes off, 1=high intensity, 4=underscore, 5=blinking, 7=reverse video, 8=invisible, 30-37=foreground color (black, red, green, yellow, blue, magenta, cyan, white), and 40-47=background color.

Note 4. On an IBM PC, the allowed "screen modes" are 0=40x25 b&w, 1=40x25 color, 2=80x25 b&w, 3=80x25 color, 4=320x200 color, 5=320x200 b&w, 6=640x200 b&w, 7=line wrap.

```

        PUBLIC      name
CODE SEGMENT
        ASSUME     CS:CODE
name PROC          FAR
        ...
;   CODE!
        ...
        RET
name ENDP
CODE ENDS
        END
-----
        public     disp_dec
code    segment
        assume     cs:code
; This is a procedure to convert a binary number in the AL register ;
to decimal ASCII characters and send them to the console. A ; binary-
to-decimal conversion works by dividing by ten (saving the ;
remainders), until the number being divided is zero. The remainders ;
(which are all 0-9) can be directly converted into decimal digits ;
('0'- '9'). The only problem is that the digits are calculated in ;
reverse order (least significant to most significant) and must ;
therefore be stored rather than immediately displayed as they are ;
calculated. The stack is ideal for this temporary storage. We push ;
a "fence" value (decimal 10) onto the stack first, so that when the ;
digits ('0'- '9') are eventually popped and displayed we will know ;
when to quit.
disp_dec proc far
        mov        bx,10          ; push a "fence" onto the stack.
        push       bx
dec_loop:
; first, divide AX (=AL) by BL (=10).
        mov        ah,0          ; prepare for a division.
        div        bl           ; divide al by 10.
; result of division is AL, and the remainder (0-9) is AH.
        add        ah,'0'       ; convert remainder to ASCII.
        mov        dl,ah        ; prepare to push it.
        push       dx
        cmp        al,0         ; all converted?
        jnz        dec_loop
; now, pop and display all of the digits, quitting when
; the "fence" value of 10 (decimal) is reached.
dec_disp:
        pop        dx           ; get a digit.
        cmp        dl,10        ; fence?
        jz         dec_done     ; if so, quit.
        mov        ah,2         ; display the digit.
        int        21H
        jmp        short dec_disp
dec_done:
        ret
disp_dec endp
code    ends
        end

```

```
; A better keyboard-reading macro.  It returns extended
; key codes as values 128-255, rather than requiring a
; second use of getchr after a returned value of 0 has been
; detected.
```

```
getchr    macro    character
           local   real_ascii
           mov     ah,7          ; read the keyboard.
           int     21h
           cmp     al,0          ; extended character?
           jnz    real_ascii    ; if not, then okay.
           mov     ah,7          ; if so, get the
           int     21h          ; extended code.
           or     al,80H        ; convert to 128-255.
```

```
real_ascii:
; if the argument of the macro is blank, just leave the
; character in AL.
           ifnb   <character>
           mov     character,al
           endif
           endm
```

```
-----
; Example of how to use the ANSI driver to clear the screen
; and then to move the cursor down 15 lines:
```

```
clear_screen db    27,'[2J$'
move_down    db    27,'[15B$'
           .
           .
           .
           display clear_screen
           display move_down
```

```
-----
; Erase entire display:
```

```
ed        macro
           esc_bracket
           putchr '2'
           putchr 'J'
           endm
```

```
-----
; This macro does the ESC[ part:
```

```
esc_bracket macro
           putchr 27
           putchr '['
           endm
```

```

        extrn  disp_dec:far
; This macro is used to ease use of the decimal display
; procedure.  Sample:
;   DECIMAL          ; DISPLAY DECIMAL NUMBER IN AL.
;   DECIMAL 46       ; DISPLAY 46.
;   DECIMAL FOO[SI]  ; DISPLAY DECIMAL NUMBER FOO[SI].

```

```

decimal macro  number
    ifnb  <number>
        mov  al,number
    endif
    call  disp_dec
endm

```

-----

```

; This macro does just the initial one-argument prefix.
; First it sends the ESC[, then it sends the ASCII decimal
; representation of the argument (if present).  That is,
; ONE_ARG sends the sequence ESC[#].

```

```

one_arg macro  argument
    esc_bracket
    ifnb  <argument>
        decimal argument
    endif
endm

```

-----

```

; This moves the cursor up.  Examples:
;   cuu  2          ; move up by two lines.
;   cuu          ; move up by one line.

```

```

cuu  macro  distance
    one_arg  distance
    putchr  'A'
endm

```

```

; This moves the cursor down.  Examples:
;   cud  2          ; move down two lines.
;   cud          ; move down one line.

```

```

cud  macro  distance
    one_arg  distance
    putchr  'B'
endm

```

```
; Program to demonstrate the use of the ANSI driver and
; macros. This program will:
;   1. Erase the screen.
;   2. Move to the bottom of the screen, and print in
;      high-intensity mode "This is the status line.
;      Status= TEST", where "TEST" is blinking.
;   3. Move to the top of the screen and input characters
;      in "typewriter mode" (and low intensity) until a
;      ctrl-C is pressed. Since we now use DOS function
;      7 in our GETCHR macro, we must explicitly test for
;      this.

; Variables:
MSG1 DB 'This is the status line. Status=$'
MSG2 DB 'TEST$'

.
.
.

; Code:
    ED                ; erase the screen.
    CUP 25,1          ; move to bottom line.
    SGR 0             ; first, turn off all attributes.
    SGR 1             ; then, turn on high intensity.
    DISPLAY MSG1      ; "This is the status line. Status= ".
    SGR 5             ; turn on blinking.
    DISPLAY MSG2      ; "TEST".
    SGR 0             ; turn off all attributes.
    CUP 1,1           ; home the cursor.

AGAIN:
    GETCHR            ; get a keyboard character.
    CMP AL,3          ; ctrl-C?
    JE DONE
    PUTCHR AL         ; display the character.
    JMP SHORT AGAIN

DONE:
```

LECTURE NOTES FOR CS-5330: IBM PC ASSEMBLY LANGUAGE  
UNIVERSITY OF TEXAS AT DALLAS

## CLASS 11

Comments

1. Although the *names* of assembly-language instructions are called *mnemonics*, the word "mnemonic" itself has nothing to do with assembly language. A mnemonic is simply an easy-to-remember name for something. Thus, 8088 instructions and ANSI cursor motion commands can each be named with "mnemonics", and these two types of mnemonics do not have to be related in any way.

2. As several of you have discovered, the arguments of a macro must always appear in the correct order. I'm sorry that I didn't make this clear, however it should be obvious on reflection. When a macro is expanded, the assembler simply makes a straight substitution of whatever you type for the arguments. After all, you haven't given the assembler the slightest bit of information to allow it to do anything else. The order of the arguments is the only thing the assembler has to go on.

3. When I mentioned in the last class that I would not be teaching this course again, I seem to have given some of you the impression that I didn't *like* the course. This isn't true; actually, it's just that I am very unlikely to be *chosen* by the Computer Science department to teach the course again. In fact I would like to teach it again, but I probably won't be given the option.

4. In discussing the extended keyboard codes last time, I forgot to mention that this is an area where incompatibility between IBM PCs and PC clones can appear. There is no guarantee that an IBM clone has the same special function keys, returns the same extended codes, or even that it uses extended codes at all! This is not to say, however, that our new, improved GETCHR macro (which can detect extended keyboard codes as well as ASCII codes) cannot be used with such clones. It is just that there isn't that much advantage in using it if there are no extended codes to detect.

5. Some things I neglected to say about angle brackets ">" and "<". In the IFB and IFNB conditional pseudo-ops, which (respectively) assemble the code that follows if the argument is blank or non-blank, the argument is enclosed in angle brackets:

```
IFB <argument>
IFNB <argument>
```

In fact, the angle brackets act as delimiters in several circumstances. It turns out, for example, that you are *apparently* not allowed to include spaces in the arguments of macros: statments like

```
PUTCHR    BYTE PTR [BX]
```

are treated as if they were just "PUTCHR BYTE". Obviously, this won't do. However, as far as I can tell, you can use angle bracket

delimiters to group the words "BYTE", "PTR", and "[BX]" together in spite of the intervening spaces, so that

```
PUTCHR    <BYTE PTR [BX]>
```

would do just what we want. I believe that the angle brackets are used just like parentheses in normal arithmetic to "group" together different items. However, this is a rather large extrapolation of the uses of angle brackets mentioned in the Macro Assembler manual.

### Review

In the previous lecture, we began by discussing separate assembly of procedures. We found that individual procedures could be separately assembled, and eventually linked together to form a complete program. We discussed only the simplest case, of a file containing a single FAR procedure and no data. We found that there were basically three things we had to do to make individual assembly work. First, in the file containing the procedure we had to include the line

```
PUBLIC    name
```

(where *name* is the name of the procedure) in order to declare that the name of the procedure was available globally -- i.e., not just in this file. Second, we needed to include the line

```
EXTRN    name:FAR
```

in the file calling the procedure. Otherwise, the assembler (not knowing that the name was *externally defined*) would merely think that it was *undefined*. Finally, after all parts of the program were assembled, we needed to change the way we linked the program so that *all* files needed were linked together.

We also discussed improving our keyboard-input abilities. Three points were brought out. First, if we want to avoid the normal ctrl-C checking done by DOS function 8 (and 1 and 10), we can use instead DOS function 7, which is equivalent except that no checking for special characters is done. Second, if we merely want to check whether a keyboard character is available (rather than actually have DOS take control and wait for one) we can use DOS function 11 (0BH). This "keyboard status" function returns a byte indicating whether or not a character is ready, but it doesn't actually return the character. The value 255 is returned in AL if a character is ready, and a zero is returned if not. Third, we discussed the *extended character codes* of the IBM PC. These are two-byte (rather than single-byte ASCII) codes for the non-ASCII characters on the IBM PC's keyboard. An extended code is fetched using two DOS "get character" function calls. The second call is needed only if the first one returns a zero byte; otherwise, the byte returned by the first call is a legitimate ASCII character as we have been assuming so far. Taking all of these facts into account, we introduced a new-and-improved get-character macro GETCHR which returned 0-127 for a real ASCII character and 128-255 for an extended code.

We also discussed some improvements in our screen-output capabilities. These improvements were effected by using the ANSI.SYS

installable device driver. ANSI.SYS needed to be "installed" by including the line

```
DEVICE=ANSI.SYS
```

in the file CONFIG.SYS (and rebooting, the very first time). After this, however, the driver would be automatically installed whenever the computer is turned on. Once installed, the ANSI driver provides a uniform software interface for screen control among PCs and PC-clones. Of course, not every hardware feature is present on every machine, so the driver just ignores it if you try to select a feature that doesn't exist. Features included in the driver are: screen erasure, positioning the cursor, and turning (or off) various attributes such as blinking, underlining, reverse video, etc. These features are controlled by means of "escape sequences" output to the screen. However, for us it is more convenient to control everything by means of macros. I have provided a special file of macros for this purpose. Since the available features are well described by the handout given in the previous lecture, there is no need to go into these things further.

### Mid-term Project

Without further ado, let me state that the mid-term project is a sorting program. The program to be written will do the following: It will read a text file from the disk, it will sort the text read in this way so that the lines are in alphabetical order, and then it will write the lines of text (in alphabetical order) to a new file. Such a program could be used in many ways. It could, for example, sort a file of mailing addresses so that the names are in alphabetical order. For example, a file reading

```
Moon, Alfred           77 Wacker Rd., Chicago, IL
Feynman, Richard      California Institute of Technology
Burkey, Ronald S.     U. of Texas at Dallas
Reagan, Ronald        White House, Washington, D.C.
```

could be sorted to

```
Burkey, Ronald S.     U. of Texas at Dallas
Feynman, Richard      California Institute of Technology
Moon, Alfred          77 Wacker Rd., Chicago, IL
Reagan, Ronald        White House, Washington, D.C.
```

Actually, there is a program called SORT provided with MS-DOS that does this. It takes lines from the standard input device, sorts them, and sends them to the standard output device. Normally this program is used with the I/O redirection feature so that it can get input from a file and send the output to another file. For example, in the case above, if the original file is called "UNSORTED.TXT", then

```
SORT <UNSORTED.TXT >SORTED.TXT
```

would create a file called "SORTED.TXT" which contains the sorted list. There is, however, a real need for a better sorting program, since SORT is unbelievably slow.

(Discussion of the mid-term assignment handout.)

Sorting Algorithms (or: Is the Bubble Sort a Communist Conspiracy?)

(Discussion of the sorting-method handout.)

Hand Compilation of Pascal Procedures

Generally speaking, algorithms are not developed in assembly language. As you have all found out, programming in assembler is difficult and confusing enough by itself. With the additional burden of having to debug the algorithm as well, program development slows to practically zero.

One alternative to trying to directly writing an algorithm in assembler is "hand compilation" or translation of programs that have been written and already debugged in another computer language. There are two advantages in this. One is, of course, that you get a working assembler program in a much shorter time. Usually, you will want to go through the assembler program afterwards to clean it up a bit and make it more efficient. Nevertheless, hand compilation is a quick route to a working program. The second advantage is that by trying to systematically convert a higher-level language to assembler you can gain a lot of insight into the operation of programs compiled with a true compiler. In our case, we will learn to hand-compile a subset of the Pascal language, since many of you are learning that language now, and since the source-code for our sorting procedures is in Pascal.

We will discuss and learn how to fake a number of Pascal constructs, including: integer variables, arrays of integers, arguments of procedures, local variables of procedures, FOR-DOs, REPEAT-WHILEs, WHILE-DOs, and various other things that occur to us along the way.

First, let us consider the simplest possible Pascal procedure: the empty procedure

```
procedure myproc;
begin
end;
```

This is quite simple to translate into assembler. We will assume that all Pascal procedures correspond to FAR PROCs, and it should be no surprise to anybody that myproc translates to

```
myproc    proc        far
           ret
myproc    endp
```

Of course, whether or not a real Pascal (or other language) compiler would turn MYPROC into a FAR procedure (as opposed to a NEAR procedure) depends entirely on the compiler. Microsoft Pascal (or FORTRAN) would create a FAR procedure, but Turbo Pascal would create a NEAR procedure.

Let's see a more difficult one. How about this:

```

procedure myproc(c:char);
begin
  write(c)
end;

```

This procedure has two new features. It has an *argument*, and it has an executable statement inside the block. Of course, it is clear that the "write(c)" must translate to something like "PUTCHR c" -- however, we don't know how Pascal manages to pass the value of *c* to the procedure. It turns out that Pascal compilers generally arrange for arguments to be passed on the *stack*, so in converting myproc to assembler we would probably want to provide a macro like

```

myproc    macro    c
           mov     al,c      ; put the value of c into al.
           push   ax       ; and push it onto the stack.
           call   myproc_procedure
           endm

```

Then, anyplace we would have "myproc(c)" in Pascal, we could simply use "myproc c" in assembler. Again, however, whether a real Pascal compiler would actually use the stack exactly like this varies from compiler to compiler. Typically, a *value* parameter might be passed on the stack like this. On the other hand, a *variable* parameter would not be passed on the stack -- rather, its *address* would be passed. With FORTRAN, all parameters are variable parameters, so only the addresses would be placed on the stack. With some compilers, data is assumed to be on the data segment, so only a word address appears on the stack; with others, data can be anywhere in memory, so a doubleword address is put on the stack. For us, *integers* will always be passed on the stack, but when we use arrays we will pass addresses so we don't have to put the entire array on the stack.

This is fine, but how does myproc\_procedure (which we haven't defined yet) manage to get the argument *c* off of the stack? It cannot easily POP it off, since it would first have to POP off its own return address! The solution is to use the BP register to index into the stack. As we have mentioned earlier, the BP register can be used to indirectly address memory just like BX, SI, and DI, with the exception that BP defaults to the *stack segment*, while the others default to the *data segment*. This still leaves us with a dilemma, however, since if the argument *c* is not POPped, then it will still be on the stack after the procedure RETURNS, and then must be explicitly POPped (presumably by our macro). [That is, we are apparently breaking our rule that everything pushed onto the stack must be popped off.] Actually, this is not a problem. There is a form of the RET instruction that we have not encountered before, in which we can specify as an argument a number of bytes to be popped after the return address is popped:

```

RET  number_of_bytes

```

This instruction would remove both the return address and the specified number of bytes from the stack. With this new RET instruction, we can write our procedure as follows:

```

; Procedure called by the myproc macro:
myproc_procedure proc    far
    push    bp            ; push bp, since we'll use it.
    mov     bp,sp        ; at this point [bp] gives the last
                        ; thing pushed on the stack (which is
                        ; its old value), [bp+2] and [bp+4]
                        ; are the two words of the return
                        ; address, and [bp+6] is the (word)
                        ; argument pushed on the stack.
    putchr [bp+6]        ; display the character.
    pop     bp            ; restore bp before quitting.
    ret     2             ; return and pop the argument.
myproc_procedure endp

```

While this isn't pretty, we have to at least admit that it will probably work. Actually, our procedure can be made to look a little better if we use a property of the EQU pseudo-op which we haven't exploited before. Consider the following use of EQU:

```
FOO EQU -10000000
```

It does not matter to EQU that there is no such number as -10000000 in 8088 assembly language; if EQU is not able to readily interpret its argument (-10000000) as a byte or a word, it simply treats it as a text string. Then, wherever it finds the string "FOO" in the program, it will replace it with "-10000000". At that point, of course, the assembler itself is likely to discover an error -- however, this doesn't matter to EQU, which performed its function (namely, simple text substitution) perfectly. This principle can be extended farther: EQU doesn't even necessarily need a *number*. We could have something like

```
FOO EQU BYTE PTR [BP+6]
```

if we wanted. This *does not* load FOO with the value at [BP+6]; rather, it replaces the string "FOO" with the string "BYTE PTR [BP+6]". Actually, this is very convenient for us. Instead of the above form of myproc\_procedure, we could write

```

; Procedure called by the myproc macro:
myproc_procedure proc    far
    push    bp            ; push bp, since we'll use it.
    mov     bp,sp
c    equ    byte ptr [bp+6]

    putchr c              ; display the character.

    pop     bp            ; restore bp before quitting.
    ret     2             ; return and pop the argument.
myproc_procedure endp

```

Here, there is absolutely no doubt that "putchr c" corresponds to "write(c)" in Pascal. Of course, for a tiny program like this, using the EQU as we have is a bit of overkill. For a more complex procedure, however, this trick is indispensable, since it means that our code can be understood and debugged much more easily.



```

; Exit point of the procedure.
done:    pop        bp
         add        sp,2          ; get rid of i
         ret        4           ; pop the arguments.
myproc_procedure endp

```

As usual, a real compiler might behave somewhat differently. For example, it might push BP onto the stack and define the local variables below BP (rather than define the local variables and then push BP). For FORTRAN, the local variables might not even be on the stack since FORTRAN subroutines cannot be recursive and therefore do not need this kind of storage maneuver.

There are several points worth noting about the above translation. First, aside from the tricky stuff with BP, SP, and EQU at the beginning and end, the translation of the procedure itself was quite straightforward. Indeed, there was almost nothing to think about. The most difficult thing we had to manage was the memory-to-memory move required by the Pascal statement `i:=n`. Apparently, we will always treat integer variables as word-size memory variables, and ignore the *registers* of the CPU as much as possible. Therefore, the process of hand-translation continually involves us in memory-to-memory operations for which we have no 8088 instructions. We could save ourselves a lot of trouble if at the outset we simply define some macros that fake memory-to-memory instructions. For example, we might have

```

; Memory to memory MOV operation:

```

```

move macro    var1,var2
    mov       ax,var2
    mov       var1,ax
endm

```

```

; Memory to memroy CMP operation:

```

```

compare macro var1,var2
    mov       ax,var2
    cmp       var1,ax
endm

```

Indeed, with the regularity of these macros it is almost impossible not to notice that we can define a memory-to-memory macro that "fakes" *all* memory to memory instructions. Such a macro is similar in concept to the generic NEAR conditional jump macro JP. What we would like is a macro called, say, MEMORY, which is used like this:

```

MEMORY      ADD,VAR1,VAR2      ; add VAR2 to VAR1.
MEMORY      CMP,VAR1,VAR2     ; compare VAR2 to VAR1.
MEMORY      MOV,VAR1,VAR2     ; move VAR2 to VAR1.
etc.

```

In fact, such a macro is very easy to define:

```

; Memory to memory instruction macro:

```

```

memory      macro    mnemonic,var1,var2
    mov       ax,var2
    mnemonic  var1,ax
endm

```

With this macro we are willing to dare even more: a for-to-do loop instead of a for-downto-do loop:

```
for i:=1 to N do write(c)
```

The middle part of our program would simply change to

```

////////////////////////////////////
      mov      i,1          ; prepare for the loop.
for_loop:
      memory  cmp,i,n      ; done?
      ja      done        ; if yes, then exit from loop.
      putchr  c           ; display the character.
      inc     i           ; and iterate.
      jmp     for_loop
////////////////////////////////////

```

A second point to consider, though, is that however easy we find it to make this translation, it is likely that in some cases we will want to check out our translated program to see about removing some of the inefficiencies our straightforward translation may have introduced. In this case, for example, we could stick with our original downto loop and to eliminate the *i* variable altogether in favor of *CX* as a counter. In that way we could use the built-in LOOP command as well. In many cases, however, the straightforward translation will be good enough in spite of the inefficiencies, or with just a small part of the program needing optimization. In the example we have been looking at, the few microseconds of inefficient code we have introduced are insignificant next to the hundreds of microseconds required by PUTCHR to display the character, so it would be silly to replace *i* with *CX* as suggested a moment ago. In general, only code inside a loop that must execute many times, very quickly, should be so optimized.

Let's look at some more Pascal constructs. Consider the problem of averaging together all of the elements of an integer array, stopping when the number -666 (minus the "number of the beast") is discovered:

```

type integer_array=array[1..M] of integer;

procedure average(  a      :integer_array;
                   var result :integer);
var i:integer;
begin
  i:=1;
  result:=0;
  while a[i]<>-666 do
  begin
    result:=result+a[i];
    i:=i+1
  end;
  result:=result div (i-1)
end;

```

Here, we see three new features. One, the WHILE-DO differs so little from the FOR-DO that there is no reason to discuss it. Two, the fact that RESULT is a "variable parameter" means that its value should be returned to the main program. A real compiler would simply pass the

address of RESULT as an argument to AVERAGE rather than passing RESULT's actual value. For us this would be rather inconvenient since we couldn't use our EQU trick any more. Therefore, we will do the following: We will pass the value of RESULT on the stack, but with the proviso that it should *not* be removed from the stack when average returns. Therefore, we should push this argument on the stack *first* and only do a RET 2 rather than a RET 4 at the end. The trickiest part is what to do about the array a[i]. Clearly, we want to pass the *offset* of the array on the stack as an argument. As usual, we want to do with with a macro, so that we could simply substitute "average a,r" wherever the original Pascal has "average(a,r)":

```
average macro    array, result
                mov     ax,result           ; pass result first.
                push   ax
                mov     ax,offset array     ; pass the array next.
                push   ax
                call   average_procedure
                pop    ax                   ; and return the result.
                mov     result,ax
                endm
```

The real question is what to do about the array once we get inside average\_procedure. There is no good answer to this that I know of (that is, no answer that ends up looking much like Pascal). We could have a macro that computes the address of an array element given the offset of the array and the index, and stores this address in (say) SI. For example, something like this:

```
element equ      word ptr [si]

; Compute address of a[i]:
index macro      array,i
                mov     si,i               ; get index 1,2,3,...
                dec     si                 ; convert to 0,1,2,...
                shl     si,1               ; convert to 0,2,4,...
                add     si,array           ; add to base address.
                endm
```

With this, we could simply use the word ELEMENT wherever we might be inclined to use a[i], so long as we had first executed "INDEX A,I". Here is how average\_procedure might look in assembler if we employ this approach:

```
average_procedure proc far
                sub     sp,2               ; move past local variable i.
                push   bp
                mov     bp,sp
; The arguments and local variables:
i               equ     word ptr [bp+2]
; bp+4 and bp+6 point to the return address.
a               equ     word ptr [bp+8]
result          equ     word ptr [bp+10]
; Also define this shorthand for [bx+si], which addresses a[]:
element         equ     word ptr [si]
```

```

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
        mov     i,1           ; i:=1;
        mov     result,0      ; result:=0;
while:
        index   a,i           ; compute SI for a[i]=a_elt.
        cmp     element,-666  ; a[i]<>-666?
        je      done_while   ; if equal, then exit while.
        memory  add,result,element ; result:=result+a[i].
        inc     i
        jmp     while
done_while:
        mov     ax,result     ; get ready to divide result
        mov     dx,0          ; by i.
        dec     i             ; i:=i-1.
        div     i
        mov     result,ax     ; and save result.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; Exit point of the procedure.
        pop     bp
        add     sp,2          ; move past local variable i.
        ret     2             ; POP just one argument.
average_procedure endp

```

As before, there is certainly room for improvement in this translation. For example, instead of executing "INDEX A,I" every time through the loop, it would be faster to just do "MOV SI,A" before the loop begins and to add two to SI every time through the loop. Whether such changes are worthwhile depends entirely on the application. The point, of course, is that you have to have a working program before you can decide whether to optimize it! Hand-compilation gives a quick way of getting the initial working program.

## MID-TERM PROJECT ASSIGNMENT FOR CS-5330

You are to write a sorting program. This program will read a text file from the disk, sort it so that the lines of text are in ascending alphabetic order, and then write the sorted file to disk. As a concrete example, suppose we have a file UNSORTED.TXT containing the lines

```
Moon, Alfred      77 Wacker Rd., Chicago, IL
Feynman, Richard California Institute of Technology
Burkey, Ronald S. U. of Texas at Dallas
Reagan, Ronald   White House, Washington, D.C.
```

which (we might suppose) is a mailing list. From this file, we want to create a file of the sorted lines (say, SORTED.TXT):

```
Burkey, Ronald S. U. of Texas at Dallas
Feynman, Richard California Institute of Technology
Moon, Alfred      77 Wacker Rd., Chicago, IL
Reagan, Ronald   White House, Washington, D.C.
```

Your program, which will be called MID-TERM.ASM, will be used to sort these files in the following way:

```
MID-TERM <UNSORTED.TXT >SORTED.TXT
```

That is, rather than explicitly opening (or creating) and closing the files, you will use the predefined input and output handles (the "standard input" and "standard output"), and will specify the appropriate files by means of I/O redirection. There is a built-in program in MS-DOS (called SORT.EXE rather than MID-TERM.EXE) which works in exactly this way, which you can experiment with. You will read the entire file into memory with a single read operation before sorting. If the file is too big to fit into the provided space, you will provide an appropriate error exit and message. All messages displayed by your program must go to the "standard error" predefined handle; if you fail to do this, the message will be redirected and stored in the output file.

You will not rearrange the text in memory (since this is a rather time-consuming process). Instead, you will maintain an array of pointers to the lines of text and will rearrange the pointers rather than the text lines themselves. Therefore, three distinct tasks remain after reading the text into memory. First, you must process the text to create the array of pointers; we will call this the PROCESS STEP. Second, you must sort; we will call this the SORT STEP. Third, you must write the sorted text to disk. Once the format of data storage in memory is understood, the latter step should be no difficulty (being quite similar to earlier assignments) and should require no further discussion. The PROCESS and SORT STEPS will be further described individually, however.

You should not attempt to write and debug the entire program at once. At first, you should completely leave out the SORT STEP. This intermediate program (without sorting) will be easy to test, since it will simply be a file-copy program.

The PROCESS STEP. When we get to the SORT STEP, we will find it convenient to have previously calculated (and stored in memory) the length of each text line -- rather than simply to have the lines terminated with CR/LFs as they are when read in from the disk. For convenience, we will impose the condition that all lines contain between 0 and 255 characters. Lines longer than 255 characters will be truncated. This means that a byte variable suffices to hold the line length. We will insist that this byte count be stored in the byte immediately preceding the line. Denoting the line length by "<COUNT>", the PROCESS step consists of converting a line like

```
This is the forest primeval<CR><LF>
```

to

```
<COUNT>This is the forest primeval
```

Also, a pointer to this string (i.e., a word whose value is the offset of <COUNT>) must be added to the end of the pointer array. In the example shown, <COUNT> is 27 since the line (exclusive of the CR/LF) contains 27 characters. This can, perhaps, be made clearer by considering a fictitious example in which a data structure such as the one we are considering has been set up by means of DBs and DWs rather than by calculation:

```
; Sample data structure of strings pointed to by a pointer array.
; The strings:
STRING2  DB  27,"No it's not, it's Cleveland"
... (Anything) ...
STRING1  DB  27,"This is the forest primeval"
... (Anything) ...
STRING3  DB  19,"Somebody's confused"
.
.
.
; The pointer array:
POINTERS DW  OFFSET STRING1
          DW  OFFSET STRING2
          DW  OFFSET STRING3
.
.
.
```

Fortunately, no movement of the text is needed to set up this data structure. There is *at least* one usable byte for the count in front of every text line (except the very first line) since there must be a carriage return (which we don't need in our data structure) at the end of the preceding text line. Since we have pointers to the strings, we don't care about the exact placement of the strings in memory, nor about the relative positions of the strings, nor about garbage between the strings (such as carriage return characters).

The pointer array is much like the buffer we used in our simple typewriter program (except that it holds words rather than bytes), so you must provide an appropriate error message and exit in case there are more lines of text in the file than you have provided for in your array. I would suggest that (throughout the PROCESS STEP) BX be mainly

used to address the pointer array and SI/DI to address the text, since SI and DI are heavily used by the built-in string operations.

The SORT STEP. Sorting will be handled as follows. You will define a macro

```
SORT NUM_LINES, POINTERS
```

which PUSHes the word NUM\_LINES onto the stack, then PUSHes OFFSET POINTERS onto the stack, and then CALLs the externally defined (i.e., separately assembled) FAR procedure SORT\_ARRAY. NUM\_LINES and OFFSET POINTERS, in fact, constitute all of the information needed by the sorting procedure, if NUM\_LINES gives the number of lines and POINTERS gives pointers to those lines. These quantities are arguments of SORT\_ARRAY and must be removed from the stack by SORT\_ARRAY.

You are free to use any sorting algorithm (except the Bubble Sort) in writing the routine SORT\_ARRAY, just so long as your routine obeys the calling conventions mentioned above. (This means that the main programs and sorting procedures of everyone in the class will be interchangeable, in the sense that any of you could link your main programs to the sorting routine of another student in the class, or to my sorting procedures. After the mid-term project has been turned in, we will have a contest between all of the sorting routines and distribute copies of the fastest to everyone in the class -- if the author doesn't object.) I would be particularly interested in seeing a Quicksort implemented. However, if you are not adventurous enough (or foolhardy enough) to work so independently, see the next paragraph.

PRETESTED SORTING ALGORITHMS. I can give you the Pascal source code for your choice of the following algorithms: *Selection Sort*, *Insertion Sort*, *Heapsort*, and *Shellsort*. Of these, *Insertion Sort* will be the easiest to handle and *Heapsort* the most difficult. The Pascal source code can be straightforwardly "hand compiled" to assembly language as described in class. Unlike the procedures required by our project, however, these algorithms all assume that the given array contains integers to be sorted (rather than pointers to strings to be sorted). Do not let this worry you. You should hand-compile and test your chosen procedure (by actually sorting integer arrays). Only convert the procedure to string form after this testing is done. The conversion to string form is actually quite easy. The only necessary changes are to certain comparison operations (which are indicated by comments in the Pascal code). These comparisons are always of the form

```
COMPARE      variable1,variable1
```

where the operands are integer memory-variables and the COMPARE macro is as described in class. You simply need to write a macro (called, say, CMP\_S) which is used exactly like COMPARE, but assumes that the values of *variable1* and *variable2* are the offsets of the strings to be compared rather than integers to be themselves compared. Replacing COMPARE with CMP\_S in the selected places mentioned before completely converts any integer sorting routine to a string-sorting routine.

If the comments above on converting Pascal integer-sorting routines to Assembler string-sorting routines don't seem clear to you, a sample bubble-sort conversion will be posted on my office door.

Unlike the homework assignments, no sample solution of the entire problem will be posted. However, various hints will be posted (such as the sample Bubble Sort mentioned above) and, as usual, I will be willing to discuss the problem (and your bugs) with you.

## SOME INTERNAL SORTING METHODS

The information below is extracted from volume 3, *Sorting and Searching*, of D. E. Knuth's "The Art of Computer Programming" series.  $N$  represents the number of records sorted. The timing and program size (=programming difficulty) figures are based on programs written in the "MIXAL" language, which is the assembly language for a fictitious computer (invented by Knuth) called "MIX". However, the information should be representative of what we would find in 8088 assembler as well. The list below includes algorithms we will use for the mid-term project, but does not (in any sense) exhaust the known sorting algorithms. The methods are arranged roughly in decreasing order of execution time when sorting a randomly ordered list. Average and maximum running times are theoretically derived. The  $N=16$  and  $N=1000$  running times are empirical averages.

| Method                                       | Program Size | Running Time (in MIX clock cycles) |             |      |         |
|--|--------------|------------------------------------|-------------|------|---------|
|  |              | Average                            | Maximum     | N=16 | N=1000  |
| Exchange Selection<br>(Bubble Sort)          | --           | $5.75N^2$                          | $7.5N^2$    | --   | --      |
| Straight Selection<br>(Selection Sort)       | 15           | $2.5N^2+3N\ln N$                   | $3.25N^2$   | 853  | 2525287 |
| Straight Insertion<br>(Insertion Sort)       | 12           | $2N^2+9N$                          | $4N^2$      | 494  | 1985574 |
| Heapsort                                     | 30           | $23.08N\ln N+0.2N$                 | $<26N\ln N$ | 1068 | 159714  |
| Diminishing<br>Increment Sort<br>(Shellsort) | 21           | $15N^{1.25}$                       | $cN^{1.5?}$ | 567  | 137502  |
| Partition-<br>Exchange Sort<br>(Quicksort)   | 63           | $11.67N\ln N-1.74N$                | $>2N^2$     | 470  | 81486   |
| Median-of-3<br>Quicksort                     | 100          | $10.63N\ln N+2.11N$                | $>N^2$      | 487  | 74574   |

**NOTES:**

1. The *Bubble Sort* is the most widely taught sorting algorithm. It has the distinction of being the worst sorting algorithm ever invented. The best thing that can be said about it is that it is only slightly more difficult to program than the *Insertion Sort*. Help stamp out this vicious bubble-sort menace today! (It goes almost without saying that this is the algorithm chosen by our textbook.)
2. The *Insertion Sort* is the easiest algorithm to program, and is quite good for small lists or for lists that are nearly in order to begin with.
3. The *Heapsort* is the simplest sorting method (and the only one on our list) guaranteed (even in the worst case) to run in a time proportional to  $N \ln N$ .
4. The *Shellsort* is the best bargain in time/programming-effort tradeoff, but the running time has been derived empirically rather than by analysis -- there is no guarantee of good performance.
5. The *Quicksort* is the fastest (on the average) and possibly the most widely used sorting algorithm. However, it has a terrible worst-case running time (the worst case being an already-sorted file) and is rather tricky to program. Moreover, being recursive, it can require up to  $N$  additional words on the stack in the worst-case.

PASCAL SOURCE: DISPLAY N COPIES OF THE CHARACTER C

```

procedure myproc(n:integer; c:char);
var i:integer;
begin
  for i:=n downto 1 do write(c)
end;

```

ASSEMBLY SOURCE FOR SAME FUNCTION

```

; Macro to CALL the procedure which mimics the Pascal procedure.
; Any place the Pascal call "MYPROC(N,C)" would be used, we use
; the assembly statement "MYPROC N,C".

```

```

myproc    macro        n,c
          mov          ax,n                ; push the first argument.
          push        ax
          mov          al,c                ; push the second argument.
          push        ax
          call        myproc_procedure
          endm

; Procedure to mimic the Pascal procedure. Called only through the
; macro MYPROC.

myproc_procedure proc    far
          sub          sp,2                ; move the stack pointer past i.
          push        bp                    ; save the old value of bp.
          mov         bp,sp
; Use the EQU trick to set up variables:
i          equ        word ptr [bp+2]
; [bp+4] and [bp+6] give the return address.
c          equ        byte ptr [bp+8]
n          equ        word ptr [bp+10]

; The actual algorithm:
          mov          ax,n                ; get ready for the for-loop by
          mov          i,ax                ; doing i:=n
for_loop:
          cmp         i,1                  ; i down to 1 yet?
          jb          done                 ; if yes, quit.
          putchr     c                      ; display the character.
          dec         i                      ; continue the count down.
          jmp         for_loop

; Exit point of the procedure.
done:     pop         bp
          add         sp,2                ; get rid of i
          ret         4                    ; pop the arguments.
myproc_procedure endp

```

**PASCAL CODE**

```
{ Pascal code to average elements of an integer array until -666. }
procedure average(a:array[1..M] of integer;    var result:integer);
var i:integer;
begin
  i:=1; result:=0;
  while a[i]<>-666 do
    begin result:=result+a[i]; i:=i+1 end;
  result:=result div (i-1)
end;
```

**ASSEMBLER CODE**

```
; Macro used any place Pascal would use the procedure AVERAGE.
average macro    array, result
  push          result                ; pass result first.
  mov           ax,offset array        ; pass the array next.
  push         ax
  call          average_procedure
  pop           result                ; and return the result.
  endm

; Compute address of a[i]. ARRAY is supposed to be an integer
; variable containing the address of a[1].
index macro     array,i
  mov           si,i                  ; get index 1,2,3,...
  dec           si                    ; convert to 0,1,2,...
  shl           si,1                  ; convert to 0,2,4,...
  add           si,array              ; add to base address.
  endm

; The actual procedure which mimics Pascal:
average_procedure proc far
  sub           sp,2                  ; move past local variable i.
  push         bp
  mov          bp,sp
; The arguments and local variables:
i equ          word ptr [bp+2]
a equ          word ptr [bp+8]
result equ     word ptr [bp+10]
element equ    word ptr [si]
;//////////////////////////////////////
  mov          i,1                    ; i:=1;
  mov          result,0                ; result:=0;
while: index   a,i                    ; compute SI for a[i]=a_elt.
  cmp         element,-666            ; a[i]<>-666?
  je          done_while              ; if equal, then exit while.
  memory     add,result,element       ; result:=result+a[i].
  inc        i
  jmp        while
done_while:
  mov         ax,result                ; get ready to divide result
  mov         dx,0                    ; by i.
  dec         i                        ; i:=i-1.
  div         i
  mov         result,ax                ; and save result.
;//////////////////////////////////////
  pop         bp                      ; Exit point of the procedure.
  add         sp,2                    ; move past local variable i.
  ret         2                        ; POP just one argument.
average_procedure endp
```

LECTURE NOTES FOR CS-5330: IBM PC ASSEMBLY LANGUAGE  
UNIVERSITY OF TEXAS AT DALLAS

## CLASS 12

Comments

1. As far as the mid-term is concerned, let me repeat my policy on sorting algorithms. You can use any algorithm you like (except the Bubble Sort), or you can use one of the Pascal algorithms I am handing out at this instant. You can either write the algorithm directly in assembler, or you can use the translation techniques I have discussed and will continue to discuss today. Using my translation techniques would probably be the easiest thing to do.

Review

In the previous lecture, we briefly discussed sorting algorithms, but the bulk of the lecture was spent discussing "hand-compilation" or translation of higher-level languages to assembler. The sample higher-level language used was Pascal, although it could really have been FORTRAN, C, or any other (similar) compiled language.

What we basically found was that, except for tricky program headers, variable declaration, and program termination, we were able to rather straightforwardly translate our procedures. Each Pascal procedure was converted in two parts: first, we wrote an actual assembler procedure which was functionally the same as the Pascal procedure; second, we wrote a macro to make the calling sequence of the assembler procedure very similar to that of the Pascal. For example, a Pascal procedure call

```
sort(a,b)
```

would be turned into a macro with a syntax like

```
sort a,b
```

which would in turn call a procedure (say, SORT\_ARRAY) that does the actual work. For us, each Pascal procedure corresponds to a FAR PROC, and all arguments to the FAR PROC are passed on the stack. Thus, the covering macro (in this case, SORT) basically does nothing more than PUSH the arguments (or POP them if they contain return values).

Not only are arguments to procedures passed on the stack, but they remain there throughout the execution of the procedure. Moreover, local variables of the procedure are also stored on the stack. (Not to mention the return address of the procedure and any other variables pushed onto the stack.) Because of all of this, we needed a better way to address the stack than we were using previously -- i.e., better than pushing and popping. The answer to this was that the BP register was ideal for addressing the stack since it acts just like the BX, SI, and DI registers, except that its default segment was the stack segment rather than the data segment.

A typical beginning for a translation of a Pascal procedure was therefore something like this:

```

        PUBLIC      name
CODE SEGMENT
        ASSUME     CS:CODE
name PROC        FAR
        SUB       SP,2*NUMBER_OF_LOCAL_VARIABLES
        PUSH     BP
        MOV      BP,SP

```

where the "SUB SP,..." instruction is used to make room on the stack for the local variables. A typical ending for a translated procedure was

```

        POP      BP
        ADD     SP,2*NUMBER_OF_LOCAL_VARIABLES
        RET     2*NUMBER_OF_ARGUMENTS
name ENDP
CODE ENDS
        END

```

where the RET statement pops all of the arguments off of the stack, as well as removing the return address of the procedure. With this kind of stack usage, [BP] refers to the old value of BP, which was pushed on the stack, [BP+2] refers to the first local variable, [BP+4] refers to the second local variable, ... , [BP+N] refers to the last local variable, [BP+N+2] refers to one word of the return address, [BP+N+4] refers to the second word of the return address, [BP+N+6] refers to the last-pushed argument, etc. As a concrete example, for a procedure with the two local variables I and J, and the argument K, [BP+2] would be I, [BP+4] would be J, and [BP+10] would be K. A trick we can use to simplify the work for ourselves is to actually define these relationships to hold, using EQU pseudo-ops:

```

I      EQU  WORD PTR [BP+2]
J      EQU  WORD PTR [BP+4]
K      EQU  WORD PTR [BP+10]

```

While all of this is rather confusing, we found that by setting up our procedures like this, we could write various macros that made our assembler programs look very much like the original Pascal. Clearly, since all of the variables correspond to memory variables rather than register variables, our translation work is simplified by macros that mimic memory-to-memory instructions (which are not present in the 8088 instruction set). The macros

```

MOVE      VAR1,VAR2          ; MEMORY-TO-MEMORY MOV.
COMPARE   VAR1,VAR2          ; MEMORY-TO-MEMORY CMP.
MEMORY    MNEMONIC,VAR1,VAR2 ; "GENERIC" MEMORY-TO-MEMORY INST.

```

were very helpful.

#### More Thoughts on Hand-Compilation

With all our efforts in the last class directed towards simplifying macros for the body of our program, we found that we were able to write rather simple programs -- with extremely tricky and confusing preliminary and terminating code appended to it. In the array-averaging example we did last time, the actual translation of the

Pascal algorithm required 14 lines of code (one of which was simply a label), while the "supporting" assembly code defining the procedure and setting up the BP register and the variables, etc. was 17 lines (not including comments or an INCLUDE statement). This is not what we would ordinarily think of as being a very desirable situation (except that it's better than 45 lines of algorithm and 17 lines of supporting code).

Fortunately, there is nothing to keep us from inventing algorithms to do away with the complex supporting code. To see how this might be done, let's start with the code that we have to use to terminate our assembler routine. This code looks something like this:

```

        POP        BP
        ADD        SP,2*NUMBER_OF_LOCAL_VARIABLES
        RET        2*NUMBER_OF_ARGUMENTS
name ENDP
CODE ENDS
        END

```

If we introduce a macro like

```

RETURN MACRO  NAME,NUMVARS,NUMARGS
        POP        BP
        ADD        SP,2*NUMVARS
        RET        2*NUMARGS
NAME ENDP
CODE ENDS
        END
        ENDM

```

then we can simply terminate our separately assembled routine with the line

```

        RETURN    PROCNAME,NUMVARS,NUMARGS

```

assuming that this was the only procedure in the file.

Similar comments apply if we try to invent a macro to begin our program and a macro to define our variables. Such macros are slightly tricky to write (and there's no particular reason to discuss them in detail), so let's just assume from now on that there is a macro

```

        BEGIN    PROCNAME,NUMVARS

```

which can be used at the beginning of the procedure to eliminate the PUBLIC, SEGMENT, ASSUME, and PROC statements, as well as the initialization of BP and SP. The definition of such a macro is posted on my office door. Similarly, let's suppose that there is a macro

```

        VAR        VARNAME [,TYPE]

```

which defines a local variable or a procedure argument, and replaces the EQU statement we have been using so far. In this case, the TYPE argument is either BYTE or WORD (with the default being WORD). Such a macro must clearly maintain an internal counter so that successive uses of the macro will define variables at successive locations on the

stack. (For convenience, the BEGIN macro initializes this counter, so that BEGIN must be used before VAR.) We also need a way to skip past the return address, which is also on the stack. Using VAR without an argument skips past one word on the stack, but defines no variable, so

```
VAR
VAR
```

skips past the return address.

Let's see how these concepts can be applied to an actual procedure. An example similar to those used in the previous class is:

```
procedure myproc(n:integer; c:char);
var i:integer;
begin
  for i:=1 to n do write(c)
end;
```

Except for an INCLUDE statement to include the macro library, here are the entire contents of a file implementing this routine in assembler:

```

BEGIN      MYPROC_PROC,1    ; DEFINE MYPROC_PROC WITH 1 LOCAL VAR.
VAR        I                ; DEFINE LOCAL VARIABLE.
VAR        ;                ; SKIP ONE WORD OF RETURN ADDRESS.
VAR        ;                ; SKIP OTHER WORD OF RETURN ADDRESS.
VAR        C,BYTE          ; DEFINE THE LAST-PUSHED ARGUMENT.
VAR        N                ; DEFINE THE FIRST-PUSHED ARGUMENT.
MOV        I,1              ; PREPARE FOR THE LOOP.
FOR: COMPARE I,N            ; END OF LOOP?
JA         DONE             ; IF YES, QUIT.
PUTCHR    C                 ; DISPLAY THE CHARACTER.
INC       I                 ; GO THROUGH THE LOOP AGAIN.
JMP       FOR
DONE:RETURN MYPROC_PROC,1,2 ; RETURN, POPPING 1 LOCAL VAR., 2 ARGS.
```

which, we have to admit, is almost lucid compared to our normal assembler programs.

Of, course the program would be even better if we had macros that would make the for-loop more obvious (not that it's not obvious now). For instance, to help out with a loop like "for counter:=start to finish" we could define

```
FOR  MACRO    COUNTER,START,FINISH,LABEL
      MOVE    COUNTER,START
LABEL&TOP:
      COMPARE  COUNTER,FINISH
      JA      LABEL&BOTTOM
      ENDM
```

and

```

ENDFOR MACRO   COUNTER, LABEL
    INC        COUNTER
    JMP        LABEL&TOP
LABEL&BOTTOM:
    ENDM

```

with which our MYPROC\_PROC would look like this:

```

BEGIN        MYPROC_PROC, 1 ; DEFINE MYPROC_PROC WITH 1 LOCAL VAR.
VAR          I              ; DEFINE LOCAL VARIABLE.
VAR          ;              ; SKIP ONE WORD OF RETURN ADDRESS.
VAR          ;              ; SKIP OTHER WORD OF RETURN ADDRESS.
VAR          C, BYTE        ; DEFINE THE LAST-PUSHED ARGUMENT.
VAR          N              ; DEFINE THE FIRST-PUSHED ARGUMENT.
FOR          I, 1, N, MAIN
    PUTCHR   C              ; DISPLAY THE CHARACTER.
ENDFOR      I, MAIN
RETURN      MYPROC_PROC, 1, 2 ; RETURN, POPPING 1 LOCAL VAR., 2 ARGS.

```

which looks rather swell. The purpose of the LABEL argument of the macro is, of course, that the assembler has no immediately obvious way of matching a particular FOR to a particular ENDFOR (if, for example, we had nested for-loops), so the additional argument helps out in that respect. Similarly, for for-downto loops we can define the macros

```

DOWNTO      COUNTER, START, FINISH, LABEL
ENDDOWN     COUNTER, LABEL

```

Actually, the LABEL argument can be entirely eliminated by maintaining an internal "stack" in the assembler. This can be done, though not easily and obviously, so we won't go into details.

With REPEAT/UNTIL or WHILE-DO or IF-THEN, we don't have it so easy, since these can depend on complicated conditions that we can't summarize quite so nicely in a macro argument. Nevertheless, we can still make our code look a little prettier (if not simpler) by defining macros for them. For instance,

```

REPEAT MACRO LABEL
LABEL&TOP:
    ENDM

```

and

```

UNTIL MACRO   CCC, LABEL
    LOCAL     DONE
    J&CCC     DONE
    JMP       LABEL&TOP
DONE:
    ENDM

```

In this case, the local label DONE takes the place of LABEL&BOTTOM and the loop continues until the specified condition CCC applies. (CCC could be anything like Z, E, NZ, NE, etc.) As before, of course, in practice a stack could be used to automatically assign LABEL and we wouldn't need any such argument. We'll see in a moment how the REPEAT and UNTIL macros can be used in practice.

Similarly, for WHILE/DO we might define

```

WHILE MACRO      LABEL
LABEL&TOP:
    ENDM

DO   MACRO      CCC, LABEL
    LOCAL      CONTINUE
    J&CCC      CONTINUE
    JMP        LABEL&BOTTOM
CONTINUE:
    ENDM

ENDDO MACRO LABEL
    JMP        LABEL&TOP
LABEL&BOTTOM:
    ENDM

```

With our usual comments about not using LABEL, in practice, these would be used like

```

WHILE
... (code to evaluate the condition) ...
DO   ccc
... (body of the loop) ...
ENDDO

```

So long as the condition was satisfied, the code between DO and ENDDO would continue to be executed.

So as not to belabor the point, let me just say that the same kind of thing can be done to simulate the IF/THEN/ELSE structure of Pascal in assembler. The macros involved are MIF, MTHEN, MELSE, and MENDIF. Notice the M's in front of the names. These are there because the assembler already has pseudo-ops called IF, ELSE, and ENDIF, and we need to avoid confusion. These macros would be used like this: To simulate, for example, the Pascal command "if i=1 then begin ... end else begin ... end" we would do something like

```

MIF
COMPARE I,1
MTHEN
...
MELSE
...
MENDIF

```

As I mentioned before, some of the macros I have just shown you can be greatly improved through tricky programming to eliminate many extraneous arguments. This tricky programming is not really for a novice, particularly as it involves some things in the assembler that we have not discussed yet. However, these macros would be very useful for us, so I am willing to provide them on disk to anybody who wants them. Also, they are posted on the door of my office and (if you are crazy) you might enjoy trying to figure out how they work.

**FROM NOW ON, I AM GOING TO UNCONDITIONALLY ASSUME THAT THE FOLLOWING MACROS ARE AVAILABLE** (with their argument structures reflecting the removal of some extraneous arguments):

```

BEGIN      procname, number_of_local_variables
VAR        varname [, vartype]
RETURN     number_of_arguments
FOR        counter, start, finish
ENDFOR     counter [, step]
DOWNTO    counter, start, finish
ENDDOWN    counter [, step]
REPEAT
UNTIL      condition
WHILE
DO         condition
ENDDO
MIF
MTHEN     condition
MELSE
MENDIF

```

where, as before, *condition* represents Z, NZ, C, NC, etc. The macro MELSE is *not* optional. *vartype* and *step* are optional arguments with *step* defaulting to 1 and *vartype* defaulting to WORD.

With all of that settled, Let's look at some further examples of hand compilation, but with some more reasonable algorithms.

*EUCLID'S ALGORITHM.* Euclid's algorithm is a way of finding the greatest common divisor of two integer numbers. Recall that the greatest common divisor of a number is the largest number that evenly divides both of the two given numbers. For example, the greatest common divisor (or GCD) of 12 and 30 is 6.

Euclid's algorithm is often considered the oldest true algorithm. In Pascal, here is Euclid's algorithm:

```

{ M and N are the given numbers, and RESULT is the GCD on output. }
procedure gcd(n,m:integer; var result:integer);
var r:integer;
begin
  repeat
    r:=m mod n;  m:=m div n;
    if r<>0 then begin m:=n; n:=r  end
  until r=0;
  result:=n
end;

```

Obviously, we would be able to write a little clearer program if we had a division macro, but since we have already introduced so many new macros, let's just see what this program would look like with what we have already.

First, we recall that no CODE, PUBLIC, SEGMENT, etc. statements are needed. We simply begin the procedure (which we'll call GCD\_PROC) with a BEGIN statement:

```
BEGIN      GCD_PROC,1
```

where the 1 indicates one local variable. To define the variables, we will suppose that RESULT was the argument first PUSHed onto the stack. The order of M and N is irrelevant since the program uses these two variables symmetrically. Thus, the local variables and arguments are defined by

```
VAR      R
VAR
VAR
VAR      M
VAR      N
VAR      RESULT
```

Notice the use of VAR with no argument to indicate the position of the return address on the stack.

The main part of the program consists of a repeat-until and a variable assignment:

```
REPEAT
...
CMP      R,0
UNTIL    E
MOV      RESULT,N
```

while the program must end with

```
RETURN   2
```

since there are three arguments but we are only popping two because the value of RESULT must be returned to the calling program. It only remains to fill in the "..." between REPEAT and UNTIL. This part of the program comes down to

```
...(some fiddling division instructions)...
MIF
  CMP      R,0
MTHEN    NE
  MOVE     M,N
  MOVE     N,R
MELSE
MENDIF
```

Here, we have no code for the ELSE part, but the MELSE macro is required. Therefore, we leave in the MELSE and just don't put any code between MELSE and MENDIF.

Putting all of this together in a program we get the following masterpiece:

```
BEGIN GCD_PROC,1
IRP X,<R,,M,N,RESULT>
VAR X
ENDM
```

```

REPEAT
    MOV AX,M           ; PREPARE TO DIVIDE.
    MOV DX,0
    DIV N
    MOV R,DX          ; STORE THE REMAINDER IN R.
    MOV M,AX          ; STORE THE QUOTIENT IN M.
    MIF
        CMP R,0
    MTHEN NE
        MOVE M,N
        MOVE N,R
    MELSE
    MENDIF
    CMP R,0
UNTIL E
MOVE RESULT,N
RETURN 2

```

There are two things of interest here. One is the use of the IRP pseudo-op. The IRP pseudo-op, which has the syntax

```

IRP  DUMMY,<VALUE1, VALUE2, ... >
... (CODE) ...
ENDM

```

repeats the macro body that follows (until ENDM is reached) once for each value in the list of values enclosed in angular brackets. Each time, the DUMMY argument is replaced by the value. In our case, the IRP ... ENDM expands to

```

VAR  R
VAR
VAR
VAR  M
VAR  N
VAR  RESULT

```

and therefore saves a few lines on the page. However, the latter form may be clearer.

More interesting, because of our extensive use of macros, our program no longer looks like assembly language. Indeed, the way the program has been formatted makes it look much more like Pascal! In a way, with all of our macros, we have managed to turn the assembler into a crude compiler.

Of course, to actually go ahead and use this procedure, we would want to write a controlling macro, say

```
GCD  M,N,RESULT
```

to call GCD\_PROC, but to avoid overdoing it, we'll leave this as an exercise for the student.

*BUBBLE SORT.* As mentioned, the one sorting algorithm that you are not allowed to use for the mid-term is the Bubble Sort (which is discussed in the book). Therefore, it is the one algorithm we can

safely discuss in class. However, I urge you never to use a Bubble Sort in practice.

While the Bubble Sorting program given in the book (p. 174) is not terribly complex, it is also by no means easy to understand. From our standpoint, of course, it also has the flaw that it uses actual data (in the data segment) and would need additional instructions added to interface with the calling program in the way we demand. Let's see how a translation of a Pascal program might look if we extensively use macros. Let's use the following Bubble Sort program:

```
_procedure_ sort(a:_array of integer_; n:_integer_);
_var_ j,t:_integer_;
_begin
  _repeat
    t:=a[1];
    _for_ j:=2 _to_ N _do
      _if_ a[j-1]>a[j] _then
        _begin_ t:=a[j-1]; a[j-1]:=a[j]; a[j]:=t _end_
    _until_ t=a[1]
  _end_;
```

Of course, to effectively write this program, we must recall the scheme we worked out to deal with array elements. We decided that the easiest thing to do was to use a macro to put the address of the array element into some register (say SI), and then to access the array element with a name like

```
element equ word ptr [si]
```

Of course, if ELEMENT accesses (say) a[i], then it is easy to see that ELEMENT-2 accesses a[i-1], ELEMENT+2 accesses a[i+1], etc. This approach is seen in the following translated program:

```

begin sort_array,2                ; Sort_array has two local
vars.                               ;
irp x,<j,t,,a,n>                   ; Define local variables j,t
    var x                           ; and arguments a,n.
endm
element equ word ptr [si]          ; Used to address a[i].
; The ADDRESS macro used below was called INDEX in class.
repeat
    address a,1                     ; Compute address of a[1].
    move t,element                  ; t:=a[1].
    for j,2,n
        mif
            address a,j             ; Compute address of a[j].
            compare element-2,element ; If a[j-1]>a[j]
        mthen a                     ; then:
            move t,element-2        ; t:=a[j-1].
            move element-2,element  ; a[j-1]:=a[j].
            move element,t          ; a[j]:=t.
        melse
            mendif
    endfor j
    address a,1                     ; Compute address of a[1].
    compare t,element               ; Repeat until
until e                             ; t=a[1].
return 2                             ; return and pop two args.

```

Notice that in this program *not one* 8088 instruction appears explicitly.

## PASCAL SOURCE CODE FOR VARIOUS SORTING ALGORITHMS

The following procedures are all taken from R. Sedgewick's book, *Algorithms*, and sort arrays of integers. The address of an array and the number of elements of the array are passed to the routines as arguments. If you want, you can translate ("hand-compile") these routines into assembler for the mid-term project. In each routine, the underlined comparison should be translated into a COMPARE macro. In order to convert the assembler versions of the routines to string-sorts rather than integer-sorts, it is merely necessary to change those particular uses of COMPARE into CMP\_S, where CMP\_S is the string-comparison macro discussed in class. A similar translation of a Bubble Sort routine is posted on the door of my office.

```

=====
{ Selection Sort procedure. }
procedure sort(a:array of integer; n:integer);
var i,k,min,t:integer;
begin
  for i:=1 to n do
    begin
      min:=i;
      for k:=i+1 to n do if a[k]<a[min] then min:=k;
      t:=a[min]; a[min]:=a[i]; a[i]:=t
    end
  end;
=====

```

```

=====
{ Insertion Sort procedure. }
procedure sort(a:array of integer; n:integer);
var v,i,k:integer;
begin
  for i:=2 to n do
    begin
      v:=a[i]; k:=i;
      while a[k-1]>v do begin a[k]:=a[k-1]; k:=k-1 end;
      a[k]:=v
    end
  end;
=====

```

```
=====
{ Heapsort procedures. Unlike all of our other sort routines,
  the heapsort must call an auxiliary routine, DOWNHEAP. }
```

```
procedure downheap(n,m:integer; a:array of integer);
label 0;
var i,k,r:integer;
begin
  r:=a[m];
  while m<=n div 2 do
  begin
    k:=m+m;
    if k<n then if a[k]<a[k+1] then k:=k+1;
    if r>=a[k] then goto 0;
    a[m]:=a[k]; m:=k
  end;
  0: a[m]:=r
end;
```

```
procedure sort(a:array of integer; n:integer);
var i,k,t:integer;
begin
  i:=n;
  for k:=n div 2 downto 1 do downheap(i,k,a);
  repeat
    t:=a[1]; a[1]:=a[i]; a[i]:=t;
    i:=i-1; downheap(i,1,a)
  until i<=1
end;
```

```
=====
{ Shellsort procedure. }
procedure sort(a:array of integer; n:integer);
label 0;
var i,k,h,v:integer;
begin
  h:=1; repeat h:=3*h+1 until h>n;
  repeat
    h:=h div 3;
    for i:=h+1 to n do
    begin
      v:=a[i]; k:=i;
      while a[k-h]>v do
      begin
        a[k]:=a[k-h]; k:=k-h;
        if k<=h then goto 0
      end;
      0:a[k]:=v
    end;
  until h=1
end;
```

SOME PASCAL-LIKE MACROS

BEGIN procname, num-local-variables

VAR varname [, vartype]

RETURN num-of-arguments

FOR counter, start, finish

ENDFOR counter [, step]

DOWNTO counter, start, finish

ENDDOWN counter [, step]

REPEAT

UNTIL condition

WHILE

DO condition

ENDDO

MIF

MTHEN condition

MELSE

MENDIF

## BUBBLE SORT

PASCAL CODE

```

procedure sort(a:array of integer; n:integer);
var j,t:integer;
begin
  repeat
    t:=a[1];
    for j:=2 to N do
      if a[j-1]>a[j] then
        begin t:=a[j-1]; a[j-1]:=a[j]; a[j]:=t end
    until t=a[1]
end;

```

ASSEMBLER CODE, WITH MACROS

```

begin sort_array,2           ; Sort_array has two local vars.
irp x,<j,t,,a,n>           ; Define local variables j,t
  var x                     ; and arguments a,n.
endm
element equ word ptr [si]   ; Used to address a[i].
; The ADDRESS macro used below was called INDEX in class.
repeat
  address a,1               ; Compute address of a[1].
  move t,element           ; t:=a[1].
  for j,2,n
    mif
      address a,j           ; Compute address of a[j].
      compare element-2,element ; If a[j-1]>a[j]
    mthen a                 ; then:
      move t,element-2     ; t:=a[j-1].
      move element-2,element ; a[j-1]:=a[j].
      move element,t      ; a[j]:=t.
    melse
      mendif
  endfor j
  address a,1               ; Compute address of a[1].
  compare t,element        ; Repeat until
until e                    ; t=a[1].
return 2                  ; return and pop two args.

```

LECTURE NOTES FOR CS-5330: IBM PC ASSEMBLY LANGUAGE  
UNIVERSITY OF TEXAS AT DALLAS

## CLASS 13

Comments

1. During this week, I would like to discuss the programming of the 8087 numeric coprocessor chip, as well as various other short, miscellaneous topics. This will conclude the computer-independent part of the course. That is, this will finish the discussion of everything we expect to apply indiscriminately to all IBM PCs and PC clones.

2. Next week, I will begin discussing use of the so-called "BIOS" of the computer and direct programming of the hardware -- the video display, the beeper, and the serial ports. The validity of this information can vary from one computer to the next. Indeed, it can vary depending on the way the computer is equipped. In class, I will explicitly discuss only a relatively standard configuration of an IBM PC or a close clone. I will *attempt*, in addition, to supply in handouts similar information for TI-PCs. (I say "attempt" because I am simply not very familiar with these machines.) Unfortunately, for machines outside those mentioned I cannot begin to come up with the necessary information. However, learning how (for example) graphics programming can be done on an IBM PC might be valuable in figuring out how to do it on some similar computers.

3. Those of you who have been religiously keeping up your macro library (or who have acquired *my* macro library) have probably noticed the main disadvantages of extensively using macros -- namely, they add a lot to your disk space and they add a lot to the assembly time. Let's consider, for example, my macro library, as used when assembling the sample bubble sort program written in the last class. My macro library is presently about 30K, including comments. When the bubble sort program assembles (using the macro library) it takes 31 seconds and gives a message "36120 bytes free". If, on the other hand, I strip all unnecessary text from the macro library (including comments, multiple spaces, and leading spaces), the macro library is about 6K, and an assembly of the bubble sort takes 22.5 seconds with "43796 bytes free". Clearly, it is better from the standpoint of program debugging to have a small macro library, while from the standpoint of documentation and maintainability it is better to have a large, well commented, and pretty-printed macro library. The answer is that you should keep *two* copies of the macro library. You should keep a nice (large) copy with a lot of comments and nice punctuation on an "archive" disk which you don't use for program development. You should keep a "lean" version without any unnecessary text on your program development disk. For any of you who want to do this, I have a program which can be used to automatically strip away comments and so forth, creating the "lean" version of the library from the fully commented version. Whenever you want to modify the macro library you should edit the "full" library, and then (once the modifications are tested) create a new "lean" version with the stripping program.

4. On the homework.

- a) If you intend to simply copy the answers out of the book, *don't bother*. I don't object to you doing this, and it won't

count against you. However, don't waste *my* time by making me look at it.

- b) On BX and BP. The *only* allowed addressing modes using *two* registers use the BX+SI, BX+DI, BP+SI, or BP+DI registers. There are no other allowed combinations. However, if a valid combination is used, they can be used like [BX+SI], [BX][SI], etc.
- c) On LEA. The big difference between instructions like "MOV AX, OFFSET FOO" and "LEA AX,FOO" is that the first calculates the number to be stored in AX at assembly time and the second calculates it at run time. This means that LEA is more flexible. For example, it can be used to compute addresses of data items like "LEA AX,FOO[SI]" and "LEA AX,FOO[SI][BX]". Indeed, it is not necessary for AX to even represent an address. An instruction like "LEA AX,[BX+SI+5]" could be viewed as an *addition* instruction in which two registers and an immediate value are added to give the value of a third register!
- d) I found most of your problems 9 very irritating. The program in the book, which most of you imitated, ran like this:

```

                TEST AX,0FFFFH
                JZ  NORM
                MOV  CX,15
NEXT_BIT:      JS  NORM
                SHL  AX,1
                LOOP NEXT_BIT
NORM:

```

This is not a good program. The worst thing about this program is the LOOP, which is totally unnecessary. (Since we have previously checked to make sure that AX is not zero, we know that the JS instruction will *always* terminate the loop. It is not necessary to maintain a count for this.) Here is a better program:

```

                TEST AX,AX
                JLE NORM
NEXT_BIT:      SHL  AX,1
                JNS  NEXT_BIT
NORM:

```

### Review

In the previous lecture, we concluded our discussion of the use of macros in hand-compilation of Pascal (or other types of programs) into assembly language. We found that, in addition to the memory-memory macros introduced earlier, it was also possible to define macros that automated translation of many other language features.

We introduced a BEGIN macro to automate writing program headers. We saw a RETURN macro that automated the termination of programs. We saw the VAR macro that automatically declared variables for us. These

particular macros were used only in separately assembled subroutines and not in the "main" program.

For putting "structure" into our programs, we found that it was possible to have macros which simulated the FOR/TO/DO, FOR/DOWNT/DO, REPEAT/UNTIL, WHILE/DO, and IF/THEN/ELSE structures in Pascal.

Taking all of these things together (including any other simplifying macros that take our fancy), we saw that it is often possible to write entire assembler procedures that don't explicitly contain a single label or 8088 instruction. In a sense, the definition of all of these macros creates a simple Pascal-like compiler inside the Macro Assembler, which we can use to write structured programs. Of course, such complete automation introduces inefficiencies in many operations, so it is often important to analyze the resulting code afterward to see if some kind of optimization is necessary (for small portions of the code).

**ASSIGNMENT:** Read chapters 4 and 9 in the book.

#### Miscellaneous Topic 1: String Comparisons

Now let's consider another necessary component of the sorting project: namely, string comparisons. We have already decided that a "string" is to be represented (in our project) by a data structure like

```
STRLEN    DB    8
STRING    DB    "A String"
```

and that the kind of information about the string we are likely to have handy is OFFSET STRLEN. (This, for example, is the information available in the "pointer array" in our mid-term project.) This type of data structure is actually how some compilers represent strings. For instance, Turbo Pascal strings are stored in memory this way.

It is likely that we would want to compare our strings with a macro like

```
CMP_S     <OFFSET STRING1>,<OFFSET STRING2>
```

This macro should return with the flags set exactly as if an integer comparison had been done: the conditional jumps JB, JBE, JE, etc. should work properly if STRING1<STRING2, etc. The greater-than and less-than operations should be based on alphabetical order. That is, we want STRING1 to be less than STRING2 if it would appear before STRING2 in a dictionary. However, for our own convenience (because of ASCII ordering), we will assume that lower-case letters are distinct from upper-case letters and that (for example) 'A'<'a'.

Actually, there are two cases of interest:

- 1) If the corresponding characters of the two strings are the same, then the strings are equal if they are of the same length, or else the one with the shorter length is smaller.
- 2) If the corresponding characters are *not* all the same, the first non-matching character determines which string is smaller.

For example, in the strings

```
"I am me"
"I am me also"
"I am not"
```

the first one is "less" than the second since it is shorter (but otherwise the same). The first and second are both less than the third since the first non-matching character of the first two ("m") is "smaller" than the first non-matching character of the third ("n").

Let us consider a macro to do the job of string comparison for us. Since the 8088 has some built-in string instructions, it is likely that we'd want to use them rather than simulate them with our own instructions. Recall that the source string for an 8088 string instruction is always in the data segment and that the destination string is always in the extra segment. We will suppose, for convenience, that ES=DS, so no problem arises from this. Also, we will suppose that the CLD instruction has been used previously, so that string operations are auto-incrementing. (If you do not assign ES as DS and use CLD, the macro will operate erratically and give strange results.)

The pseudo-code for such a macro might go something like this:

```
{ Note that because of the way we've defined our strings, STRING[0]
  is the length of the string, STRING[1] is the first character, and
  so forth. }
for i:=1 to min(string1[0],string2[0]) do
  if string1[i]<>string2[i] then goto done;
{ If this point has been reached, then all of the characters match and
  the only difference can be in the string lengths. }
compare(string1[0],string2[0]);
done:
```

Here it is assumed that both the COMPARE operation and the <> operation set the flags properly, so that when DONE is reached all the work of the comparison has been done. In actual practice, the MIN operation hypothesized above must already compare the string lengths, so we can simply store this information and re-use it later if necessary, rather than do the comparison again. This involves two new instructions, LAHF and SAHF. LAHF saves the values of the various arithmetic flags (Z, C, P, S, O, and H) as a byte in the AH register. Similarly, SAHF restores the flag values using AH. As indicated, these instructions are good for temporarily saving the flags, while in the meantime using instructions that would mess up the flags if they had not been saved.

```
; String comparison macro:
;   SOURCE operand = address of source string.
;   DESTINATION operand = address of destination string.
cmp_s    macro    destination, source
          local   done, length_ok, cmp_lengths
; Set up SI and DI to the source and destination strings:
          mov     di,destination
          mov     si,source
```

```

; Perform the MIN operation from the pseudo-code. Set CL to be the ;
smaller of the two string lengths:
    mov     cl,[di]          ; get destination length.
    cmp     cl,[si]         ; compare to source length.
    lahf                    ; save flags in AH register.
    jb     length_ok       ; if CL=smaller length, then ok.
    mov     cl,[si]         ; otherwise, load CL with smaller.
length_ok:
; Since I in the for-loop must start at one, eliminate the I=0 case. ;
If the length is zero, don't do the comparison, or else we'd loop
; through 64K bytes!
    or     cl,cl           ; CL=0?
    jz     cmp_lengths     ; If yes, compare lengths.
    xor    ch,ch           ; Otherwise, make CX=length.
; Exchange SI and DI because of the goofy "reverse" way the string
; comparison instructions work.
    xchg   si,di
    inc    si              ; move SI past length to string.
    inc    di              ; same for DI.
; Now, compare them:
    repe   cmpsb
    jne    short done     ; if a non-match was found, quit.
; Otherwise, the strings completely matched, so we must use the value
; of the length comparison made earlier as our result:
cmp_lengths:
    sahf                    ; restore flags from AH register.
; Exit point
done:
    endm

```

This macro contains a number of features we haven't seen before. For instance, there are a couple of tricks involved. One trick is the use of "OR CL,CL" to check whether CL is zero. This has no effect on CL, but sets the ZF flag if CL is zero. Similarly, "XOR CH,CH" is used to zero out CH. These instructions are useful in some situations since they execute in 3 clock cycles, while the more obvious "CMP CL,0" and "MOV CH,0" execute in 4. We also saw the XCHG instruction, which we have not discussed before. The XCHG instruction is exactly like the MOV instruction except that it *exchanges* the values of its two arguments rather than simply moving one to the other.

Recall our earlier definition of the strings "This is the forest primeval", "No it's not, it's Cleveland", and "Someone's confused", with the values OFFSET STRING1, OFFSET STRING2, OFFSET STRING3, ... being stored in the pointer array. A statement in our program like

```
CMP_S    <OFFSET STRING1>,<OFFSET STRING2>
```

would allow the jumps JA or JAE to be taken, but not JE or JB, since "This is ..." is greater than "No it's not ...". The angle brackets are necessary here since the arguments of the macro contain spaces.

On the other hand, if BX contained 2, then POINTERS[BX] is the address of STRING2 and POINTERS[BX+2] is the address of STRING3, so

```
CMP_S    POINTERS[BX],POINTERS[BX+2]
```

would jump for a JB or JBE, but not for JE, JA, etc., since "No it's not ..." is less than "Someone's confused".

This is very convenient for *converting* an integer-sorting routine to a string-sorting routine. In an integer-sorting routine, all *decisions* are (or could be) made by statements like

```
COMPARE   POINTERS [SI], POINTERS [DI]
```

or

```
COMPARE   V, POINTERS [SI]
```

where V has been previously defined by

```
MOVE      V, POINTERS [SI]
```

Consequently, if POINTERS is interpreted instead as a word-array of pointers to strings rather than as an array of integers, the comparisons will work properly if COMPARE is replaced by CMP\_S. With a little thought (and a listing of a sorting algorithm in front of you), it is easy to see that such replacement of certain selected COMPAREs is the *only* necessary change in the sorting algorithm.

Example: In the Bubble-sort algorithm discussed in the previous class, if you replace the line "COMPARE ELEMENT-2,ELEMENT" by "CMP\_S ELEMENT-2,ELEMENT", we get a string-sorting routine which should work in the mid-term project (if you have otherwise kept to the program specifications)! Of course, you are not allowed to turn in a Bubble Sort for the mid-term project -- but you might want to try it out if you're having a little difficulty (just to assure yourself that this problem is solvable).

#### Miscellaneous Topic 2: The System Librarian, LIB

We have already seen how convenient (and easy) it can be to modularize our program by dividing it up into procedures (each possibly being controlled by a macro). By deciding on a uniform interface between procedures -- such as deciding that most procedures are FAR, and that all parameters should be passed on the stack in a certain way -- it is possible to actually write procedures at different times, store them in different files, and individually assemble them whenever we like. Indeed, individual procedures may even be written by different programmers, as long as each agrees to conform to the standard interface. The LINK program then takes care of putting all of these routines together into a single executable (.EXE) file.

As usual, however, there is a disadvantage. The disadvantage is that each procedure then requires its own .ASM file and its own .OBJ file on the disk. The .ASM file, of course, contains the source code, and the .OBJ file contains the linkable object code. With all of these .ASM files and .OBJ files floating around, the disk rapidly becomes either full, or at least so cluttered that it's difficult to read the directory. Worse, at link-time (i.e., when LINK is being run) you have to remember the name of each file containing a procedure you've used and type it in!

Of course, the .ASM files probably shouldn't be on your working disk after they have been debugged: They should be safely stored on an "archive" disk where you won't accidentally screw them up. (This same comment holds for *everything* that you get working, such as your homework programs, both in .ASM and .EXE form.) However, this doesn't solve the problem of all the .OBJ files. For example, you already have (at least) object files for: a hexadecimal display routine, a lower-to-upper-case translation routine, a decimal display routine (for those of you using the ANSI screen driver). Soon you will have object files for several sorting routines (at least an integer sort and a string sort), plus files you will have to create in conjunction with the final project. Each of these routines should be on the disk with the linker. Clearly, this could quickly get out of hand.

One alternative to storing all of the object files separately on the disk and explicitly linking these files together is combining the object files together into a "library". That is, the .ASM files would still be assembled separately, creating individual .OBJ files. But these .OBJ files would be processed further (creating a "library") and then be erased. Rather than seeing a large number of .OBJ files on the disk, we would then see just the library. Just as the linker is able to link together various files to create an executable program, it is able to "search" a library for unknown symbols. It does not link in every routine in the library -- it just links the routines it needs (and which are in the library).

For the sake of argument, suppose that we have written a program (called MAIN) that uses the external routines SORT, UPCASE, and HEXDISP (which, respectively, sort, convert to upper case, and display a hexadecimal number). Assuming that each program is represented by a .OBJ file, we would normally link our program by typing

```
A><u>LINK MAIN+SORT+UPCASE+HEXDISP;
```

Now, if we had left out the ";" above, the linker would have gone ahead to prompt us for various other items: the name of the .EXE file (which defaults to be the same as the .ASM file), the name of a .MAP file (which is of no concern to us at present), and the names of libraries to be searched. The responses to all of these prompts can also be given on the command line (if we want) by separating the responses by commas. For example, to create a .EXE file called TEMP.EXE, a map file called TEMP.MAP, using a library called LIBRARY.LIB, we could say

```
A><u>LINK MAIN+SORT+UPCASE+HEXDISP,TEMP,TEMP,LIBRARY
```

If we don't want to answer one of the prompts, we can simply put nothing between the commas, and the default response will be filled in. For example, in the case above, it is more likely that we want the program to be called MAIN.EXE and that SORT, UPCASE, and HEXDISP are already in LIBRARY.LIB. In that case, we could type

```
A><u>LINK MAIN,,,LIBRARY
```

By default, library files have the extension .LIB, and we can have as many of them searched as we want (by giving their names, separated by plus signs). For example, we could have searched both LIBRARY1 and LIBRARY2 (in that order) with

A>LINK MAIN,,,LIBRARY1+LIBRARY2

This is all fine, except that we don't know how to get all of these object files into libraries.

\*\*\*\*\* WARNING \*\*\*\*\*  
 IT IS UNBELIEVABLY EASY TO SCREW UP A LIBRARY IF YOU DON'T KNOW  
 WHAT YOU'RE DOING. THEREFORE, ALWAYS MAKE A BACKUP COPY FIRST!  
 \*\*\*\*\*

Object files are put into libraries using the utility program LIB.EXE which is (sometimes) distributed with MS-DOS or MASM. (For some odd reason LIB is not uniformly available. However, I have seen it on some of your disks so I know that you can get a copy if you would like to use it.) LIB allows you to do three things. It lets you:

- + Add a "module" to the library. (A module is essentially the same as an object file.) This also creates the library file if it did not already exist.
- Delete a module from a library.
- \* Extract a module from a library. That is, to create an object file which is the same as the module in the library. The library file itself is unaffected by this.

When using LIB, each function is designated by the indicated symbol (+ - \*). The general syntax of LIB is

*LIB commandstring[,listfile]*

The command string is a list of libraries and object files, separated by the function symbols mentioned above, and the listing file is a file which contains a report on the contents of the library after these operations are performed. The easiest way to illustrate how these functions work is to actually see some examples.

First, let's suppose that we have a library (or *don't* have a library) called LIBRARY.LIB to which we want to add a module represented by the file UPCASE.OBJ. Here is a LIB command to do this:

```
LIB LIBRARY+UPCASE;
```

(The meaning of the semicolon is that no listing file is produced.) Assuming that this operation was successful (which you shouldn't until you have tested it), the UPCASE module is now in the library and UPCASE.OBJ can (if you like) be erased from the disk. Similarly, to delete the UPCASE module from LIBRARY.LIB, we would say

```
LIB LIBRARY-UPCASE;
```

This does not create a copy of UPCASE.OBJ on the disk -- it simply eliminates UPCASE from the library. Consequently, if you have no other copy of UPCASE.OBJ, you could be in trouble. On the other hand, *extracting* UPCASE from the LIBRARY with

```
LIB LIBRARY*UPCASE;
```

would create UPCASE.OBJ. It would also leave the library unchanged, so that you now have two copies of UPCASE: one on the disk and one in the library.

You can also combine several commands in one line. For example,

```
LIB LIBRARY-UPCASE+UPCASE*HEXDISP-HEXDISP;
```

would: a) remove the existing copy of UPCASE (probably an old version) from the library; b) add a new copy of UPCASE (probably an updated version) to the library from the disk; c) create a HEXDISP.OBJ file from a module in the library; and finally d) get rid of the HEXDISP module from the library (but leaving the copy on disk). It is obvious, but note anyway that the commands "+", "-", and "\*" are not related to arithmetic commands; in particular, \* *does not* have a higher priority than "+" and is *not* automatically executed first. The commands are simply executed in the order they are encountered, from left to right.

As mentioned, by filling in the name of the listing file, you can also get a report on the contents of the library. This is probably a good thing to do after a complicated operation like that given above. For example, to see the report on the screen we would have replaced the ";" by a ",CON". Of course, not having done that we could still run LIB again to get the report:

```
LIB LIBRARY,CON
```

Another (possibly unobvious) advantage of using libraries rather than object files is that they can save a lot of space. Although using the "DIR" command in DOS gives one the impression that space for files is allocated by the byte (with no wasted space), in truth space for files is allocated in terms of blocks of a fixed size. For example, a typical block size for a floppy disk is 1K, while a typical blocksize for a hard disk is 4K. This means that any file, no matter how small, must occupy at least 1K. For example, thirty .OBJ files must occupy at least 30K on a floppy (120K on a hard disk). Now, as it happens, .OBJ files for individually assembled procedures are typically very small (say, only a few hundred bytes long). Therefore, storing them as separate object files probably wastes 75-80% of the disk spaced use for that purpose on a floppy (and much more on a hard disk). In a *library*, on the other hand, all of this excess space is removed. Thus, a library containing the thirty modules alluded to earlier might occupy on 5 or 6K of disk space, a clear improvement.

### Miscellaneous Topic 3: Jump Tables and Call Tables

There is an assembly-language programming technique which is so widespread and useful that it is a shame we have not had occasion to make use of it in the assigned homework. This is the technique of the "jump table", and we will discuss it now.

In the homework we have seen already a number of instances of the following programming situation. We have some value (say, for concreteness, a character stored in the AL register) which we would like to test. On the basis of the value of the character, we would

like to perform some particular action from a list of many possible actions. For instance, if the character is a backspace we would like to backspace, if it is a tab we would like to tab, etc. A typical way of programming this has been something like this:

```

CMP  AL,8           ; BACKSPACE?
JE   BACKSPACE     ; IF YES, GO ELSEWHERE.
CMP  AL,9           ; TAB?
JE   TAB           ; IF YES, GO ELSEWHERE.
CMP  AL,10          ; LINE FEED?
JE   LF            ; etc.

```

There is nothing particularly wrong with this (at least in the case shown) since we only have to check a small number of alternatives. One potential problem, however, is that it takes longer to check for the alternatives near the bottom of the list than it does to check for the alternatives near the top. If we had to perform these checks for all 255 characters, or for 1000 extended characters, the penalty in execution time might be too large. Another problem is simply that the code looks bad. It is rather spaghetti-like and difficult to follow in some cases. If, for example, we had written instead

```

CMP  AL,8           ; BACKSPACE?
JNE  NOBACKSPACE   ; IF NOT, CONTINUE.
... (CODE FOR BS)... ; OTHERWISE, PROCESS THE BACKSPACE.
NOBACKSPACE:
CMP  AL,9           ; TAB?
JNE  NOTAB         ; IF NOT, CONTINUE.
... (CODE FOR TAB)... ; OTHERWISE, PROCESS THE TAB.
NOTAB:

```

(as, indeed, we are likely to) the program is almost impossible to understand.

A better alternative would be to have a built-in CASE construct as in Pascal. For instance, if we could write

```

CASE AL OF
8:BACKSPACE
9:TAB
10:LINE_FEED
etc.

```

we could follow the code much more easily. Fortunately, if the values to be checked are nearly in arithmetic progression (as above with 8, 9, 10, ...) we can do something like this with the "jump table" technique.

A jump table is nothing more than a linear array of words or doublewords representing addresses in the program. For example,

```

JP_TAB    DW    BACKSPACE
          DW    TAB
          DW    LINE_FEED

```

would be a (rather short) jump-table if there actually were labels in the program like BACKSPACE, TAB, and LINE\_FEED. (Note: No OFFSET operator is needed.) In this case, we have a NEAR jump-table -- a

table of word addresses within the segment. If we used instead the DD operator (rather than DW) we would have a FAR jump-table -- a table of doubleword addresses that could be anywhere in memory.

Jump tables are useful because the addresses used in JMP or CALL instructions can be specified directly or indirectly as well as immediately. (Recall that these are all *addressing modes* of the 8088.) Normally, we say something like

```
JMP AGAIN
.
.
.
AGAIN:
```

This is essentially the *immediate* addressing mode, since the address (which is simply a number) is specified in the instruction itself. We can, however, be a little trickier. We could, for example, say

```
JMP JP_TAB
```

Since JP\_TAB is not itself a label, and is instead a variable, the assembler will interpret its *value* (rather than its address) as the location to jump to. In this case, since the value of JP\_TAB is BACKSPACE, the program will jump to BACKSPACE. This is, of course, the direct addressing mode.

Jump-tables really have their use, however, in indexed addressing modes. For example, we could have instructions like

```
JMP JP_TAB[SI]
```

which would use the value at the SI-th position of the jump-table as the address. For example, if SI is 0 this instruction would jump to BACKSPACE; if SI is 2 it would jump to TAB; etc.

This fact has many potential uses. One common use is if we write a program which is to be used by other programs, but not to be *linked* to them. For example, we might want to write a program which we want to load into memory to be used at various times by various programs. Or, we might want to "call" the program from BASIC. (Since BASIC is interpreted rather than compiled, we could not LINK our program to BASIC). There is no problem with this if the program simply performs one well-defined function. The problem with such an idea comes if the program performs *several* different functions, each with its own entry point. If this is the case, we have no easy way of determining the entry points for the individual functions (except the very first, which could be placed at the beginning). One way out of this is to put a *jump table* (containing the entry points) at the beginning of the program, where we could easily read it from BASIC. Alternately, a parameter selecting the desired function could be passed as an argument and the program itself could access the jump-table. This is the technique used, for example, by DOS interrupt 21H.

Another common use is in processing commands typed at the keyboard. For example, a typewriter program written using this

technique might look something like this: First, in the data segment we might have a jump-table like

```
CTRL DW BELL           ; PROCESS BELL CHARACTER, ASCII 7.
      DW BACKSPACE
      DW TAB
      DW LINEFEED
      DW AGAIN         ; DO THIS FOR ASCII CHARACTER 11.
      DW FORMFEED
      DW CARRIAGERETURN
```

for processing the ASCII control characters 7-13. If we assume that only these control characters need special processing and all others would just be ignored, our program itself might be:

```
AGAIN:
      GETCHR           ; GET A CHARACTER INTO AL.
      CMP AL,127      ; TOO BIG TO BE PRINTABLE?
      JAE AGAIN       ; IF YES, DO NOTHING.
      CMP AL,' '      ; PRINTABLE (IF BIGGER THAN ' ', SINCE <127)?
      JB CONTROL      ; IF NOT, MUST BE CONTROL CHARACTER.
      PUTCHR          ; DISPLAY THE CHARACTER.
      JMP SHORT AGAIN ; LOOP.
; At this point, the character is known to be a control character.
CONTROL:
      CMP AL,7        ; LESS THAN 7?
      JB AGAIN       ; IF YES, DO NOTHING.
      CMP AL,13       ; GREATER THAN 13?
      JA AGAIN       ; IF YES, DO NOTHING.
; At this point, the character is known to be in the jump-table.
; We need to convert the byte in AL, which is 7,8,...,13 to a number
; in SI which is 0, 2, 4, ..., 12:
      SUB AL,7        ; FIRST, CONVERT TO 0,1,2,...6.
      SHL AL,1        ; NOW, CONVERT TO 0,2,...12.
      MOV AH,0        ; STORE IN AX RATHER THAN AL.
      MOV SI,AX       ; STORE IN SI RATHER THAN AX.
      JMP CTRL[SI]    ; JUMP TO THE PROPER SECTION OF CODE.
;
; As a first pass, make most of the functions trivial (and not really
; necessary).
;
BELL:
      PUTCHR 7        ; sound the bell.
      JMP AGAIN
;
BACKSPACE:
      PUTCHR 8        ; backspace.
      JMP AGAIN
;
TAB:
      PUTCHR 9        ; tab.
      JMP AGAIN
;
LINEFEED:
      PUTCHR 10       ; line feed.
      JMP AGAIN
;
```

```

FORMFEED:
    ED                ; ANSI erase display command.
    JMP AGAIN
;
CARRIAGERETURN:
    PUTCHR 13        ; do both a carriage return and a line feed.
    PUTCHR 10
    JMP AGAIN

```

Another use for the jump-table is if the table contains addresses of *procedures*. In this case, we would use a command like

```
CALL JP_TAB[SI]
```

One might think that this would use a "call table" rather than a "jump table". However, only the term "jump table" is in common use.

#### Miscellaneous Topic 4: The Program Segment Prefix (PSP)

We have written several programs which performed I/O on files, and so far we have seen two distinct ways of specifying filenames for our programs. First, of course, we saw that it was possible to simply have our program prompt the user to enter the name from the keyboard as the program runs. Second, we have seen that filenames can be specified on the DOS command line by using I/O redirection. Of course, even though we have never learned to do it, we know from experience that it is possible to control our programs quite flexibly from the DOS command line. For example, earlier in the lecture we saw that very complex LIB commands like

```
LIB MAIN+PACKED,,,LIBRARY1+LIBRARY2
```

can be handled entirely from the command line without any prompting by the program.

Such tricks (as well as others we won't discuss) are performed using something called the "Program Segment Prefix" (or *PSP*). The PSP is a 256 byte "header" that is attached to every program loaded by DOS. The PSP helps to "interface" your program to DOS. For example, the various manipulations we do at the beginning of the program to store the return address basically just save the location of the PSP. Although we define our own stack, if no stack is defined it defaults to being in the PSP. In DOS 1.x (as opposed to 2.x or above, which we are using), most disk operations used the PSP by default. The fact of most interest to us is that there is a buffer in the PSP which is used to pass commands from DOS to our program.

Here are some of the items in the PSP. Some of these things we will discuss here, and others are included for those of you who know more about the IBM PC or who will continue programming on the IBM PC once this course is finished. The latter items are starred.

|   | <u>Address</u> | <u>Data Type</u> | <u>Description of Data</u>   |
|---|----------------|------------------|--|
| * | 0H             | WORD             | An INT 20H instruction. This is actually the address returned to by DOS when the RET at the end of your program is executed.                                   |
|   | 2H             | WORD             | Total length of memory in <i>paragraphs</i> . (A paragraph is 16 bytes.) This count includes the memory already used (for example, by DOS or by your program). |
| * | 2CH            | WORD             | Segment number of the "environment". The environment contains ASCIZ strings defined by the DOS SET command (and other commands).                               |
| * | 5CH            | 16 BYTES         | First FCB created from DOS command line.   |
| * | 6CH            | 16 BYTES         | Second FCB created from DOS command.   |
|   | 80H            | BYTE             | Length of the DOS command line.  |
|   | 81H            | 127 BYTES        | The DOS command line itself, <i>except</i> for the name of the program being executed (which has been removed).  |

Only two items here are really understandable to us, given our present knowledge. First, the length of the computer's memory (in 16-byte paragraphs) is given at offset=2 in the PSP. Second, the DOS command line -- i.e., the line you typed in response to the DOS A> prompt -- is given beginning at offset=80H. In order to access these items, we merely need to know where in memory the PSP is located.

When your program begins executing, both the ES and DS registers point to the PSP. (Indeed, in some circumstances, even CS and SS point to it.) Of course, one of the first actions in our template program is to modify DS so that it points instead to our own Data Segment. However, ES is still pointing at the PSP, so we can address the PSP items mentioned above using ES. Here, for example, is how we can read the memory size into AX:

```
MOV AX,ES:2
```

Of course, to convert this into bytes we would need to multiply by 16, while to convert to K we would need to divide by 64.

Of somewhat more interest is the DOS command buffer at 80H in the PSP. This buffer holds the last DOS command line typed (except for the program name, which has been removed) and consequently can be used to pass parameters or names of files to the program. If, for example, we typed a DOS command line of "TEST HELLO, STRANGER" (where TEST is the name of our program), then when TEST executed it would find that the string beginning at 81H in the PSP read " HELLO, STRANGER" (notice the leading space) and that the count at 80H would be 16. Here is a short program that does nothing more than displays the command buffer:

```

PUSH DS          ; SAVE DS.
PUSH ES          ; PUT ES INTO DS SINCE THE DOS DISPLAY
POP DS           ; FUNCTION REQUIRES THE MESSAGE TO BE IN THE
                  ; DATA SEGMENT.
MOV AH,40H       ; DOS WRITE-RECORD FUNCTION
MOV BX,1         ; WRITE TO STANDARD OUTPUT HANDLE.
MOV CL,DS:80H    ; GET THE BYTE COUNT INTO CX.
MOV CH,0
MOV DX,81H       ; PUT OFFSET OF BUFFER INTO DX.
INT 21H
POP DS           ; RECALL DS.

```

The only thing here which is possibly slightly tricky is that we have addressed the length byte as DS:80H. The "DS:" would appear to be a default segment override, even though the byte is actually in the data segment. Actually, the "DS:" does not override the segment -- rather it simply specifies that 80H is the address of a variable rather than an immediate value.

In MS-DOS, only two types of information are typically passed on the command line. These are:

- 1) Names of files; and
- 2) "Switches".

A switch is a parameter which modifies slightly the operation of the program. For example, we can format a disk with

```
FORMAT B:
```

but we can format and *verify* the disk with

```
FORMAT B:/V
```

Here, the "/V" is a switch. Normally, as here, switches are signified by a preceding "/". For your own programs, you can, of course, define any switches you like (or none). Here is a short sequence of instructions to check the entered command for the switch character "/":

```

; CHECK FOR SWITCH CHARACTER USING 8088 STRING INSTRUCTIONS. WE
; DO NOT HAVE TO SET ES=DS AS ABOVE SINCE WE USE THE "DESTINATION
; STRING", WHICH IS ALWAYS ADDRESSED BY ES:DI.
    CLD                ; SET STRING FUNCTIONS TO INCREMENT.
    MOV DI,81H         ; STARTING OFFSET OF STRING.
    MOV CL,ES:80H     ; GET LENGTH OF STRING INTO
    MOV CH,0          ; CX.
    JCXZ NOSWITCH     ; IF LENGTH IS ZERO, THEN NO SWITCH.
    MOV AL,'/'        ; CHECK FOR "/"
    REPNE SCASB       ; USING THE "SCAN" FUNCTION.
    JNE NOSWITCH      ; IF NOT FOUND, THEN NO SWITCH.
YESSWITCH:           ; OTHERWISE,...

```

PSEUDO-CODE STRING COMPARISON

```
{ Note that because of the way we've defined our strings, STRING[0]
  is the length of the string, STRING[1] is the first character, and
  so forth. We assume that the <> character comparison and the
  COMPARE byte comparison set the flags properly. }
for i:=1 to min(string1[0],string2[0]) do
  if string1[i]<>string2[i] then goto done;
{ If this point has been reached, then all of the characters match and
  the only difference can be in the string lengths. }
compare(string1[0],string2[0]);
done:
```

STRING COMPARISON MACRO

```
; String comparison macro:
;   SOURCE operand = address of source string.
;   DESTINATION operand = address of destination string.
; (It might also be nice to PUSH SI and DI. )
cmp_s    macro    destination, source
          local    done, length_ok, cmp_lengths
; Set up SI and DI to the source and destination strings:
          mov     di,destination
          mov     si,source
; Perform the MIN operation from the pseudo-code. Set CL to be the ;
smaller of the two string lengths:
          mov     cl,[di]           ; get destination length.
          cmp     cl,[si]          ; compare to source length.
          lahf                    ; save flags in AH register.
          jb     length_ok        ; if CL=smaller length, then ok.
          mov     cl,[si]          ; otherwise, load CL with smaller.
length_ok:
; Since I in the for-loop must start at one, eliminate the I=0 case. ;
If the length is zero, don't do the comparison, or else we'd loop
; through 64K bytes!
          or     cl,cl             ; CL=0?
          jz     cmp_lengths      ; If yes, compare lengths.
          xor     ch,ch           ; Otherwise, make CX=length.
; Exchange SI and DI because of the goofy "reverse" way the string
; comparison instructions work.
          xchg   si,di
          inc    si               ; move SI past length to string.
          inc    di               ; same for DI.
; Now, compare them:
          repe   cmpsb
          jne   short done       ; if a non-match was found, quit.
; Otherwise, the strings completely matched, so we must use the value
; of the length comparison made earlier as our result:
cmp_lengths:
          sahf                    ; restore flags from AH register.
; Exit point
done:
          endm
```

SAMPLE STRING COMPARISONS:

```
STRING1 DB 19,"HELLO, I'M A STRING"
STRING2 DB 15,"I'M ANOTHER ONE"
        CMP_S    <OFFSET STRING1>,<OFFSET STRING2>
```

## LIB COMMANDS

- + ADD A MODULE TO THE LIBRARY.
- DELETE A MODULE FROM THE LIBRARY.
- \* EXTRACT A MODULE FROM THE LIBRARY.

### EXAMPLES :

```
LIB LIBRARY +UPCASE;  
LIB LIBRARY -UPCASE;  
LIB LIBRARY *UPCASE;  
LIB LIBRARY-UPCASE+UPCASE;  
LIB LIBRARY, CON
```

Sample Use of Jump Tables

; A simple "typewriter" program, using a jump table to process the  
; special control characters listed below:

```

CTRL DW BELL           ; PROCESS BELL CHARACTER, ASCII 7.
      DW BACKSPACE
      DW TAB
      DW LINEFEED
      DW AGAIN         ; DO THIS FOR ASCII CHARACTER 11.
      DW FORMFEED
      DW CARRIAGERETURN

AGAIN:
      GETCHR           ; GET A CHARACTER INTO AL.
      CMP AL,127      ; TOO BIG TO BE PRINTABLE?
      JAE AGAIN       ; IF YES, DO NOTHING.
      CMP AL,' '      ; PRINTABLE (IF BIGGER THAN ' ', SINCE <127)?
      JB CONTROL      ; IF NOT, MUST BE CONTROL CHARACTER.
      PUTCHR          ; DISPLAY THE CHARACTER.
      JMP SHORT AGAIN ; LOOP.
; At this point, the character is known to be a control character.
CONTROL:
      CMP AL,7        ; LESS THAN 7?
      JB AGAIN        ; IF YES, DO NOTHING.
      CMP AL,13       ; GREATER THAN 13?
      JA AGAIN        ; IF YES, DO NOTHING.
; At this point, the character is known to be in the jump-table.
; We need to convert the byte in AL, which is 7,8,...,13 to a number
; in SI which is 0, 2, 4, ..., 12:
      SUB AL,7        ; FIRST, CONVERT TO 0,1,2,...6.
      SHL AL,1        ; NOW, CONVERT TO 0,2,...12.
      MOV AH,0        ; STORE IN AX RATHER THAN AL.
      MOV SI,AX       ; STORE IN SI RATHER THAN AX.
; If jump table is in code segment, use "JMP CS:CTRL[SI]" below.
      JMP CTRL[SI]   ; JUMP TO THE PROPER SECTION OF CODE.
;
; The code jumped to by the jump table:
;
BELL:
      PUTCHR 7        ; sound the bell.
      JMP AGAIN
      .
      .
      .
;
FORMFEED:
      ED              ; ANSI erase display command.
      JMP AGAIN
;
CARRIAGERETURN:
      PUTCHR 13       ; do both a carriage return and a line feed.
      PUTCHR 10
      JMP AGAIN

```

PARTIAL LAYOUT OF THE PROGRAM SEGMENT PREFIX (PSP)

| <u>Address</u> | <u>Data Type</u> | <u>Description of Data</u>   |
|----------------|------------------|--|
| * 0H           | WORD             | An INT 20H instruction. This is actually the address returned to by DOS when the RET at the end of your program is executed.                                   |
| 2H             | WORD             | Total length of memory in <i>paragraphs</i> . (A paragraph is 16 bytes.) This count includes the memory already used (for example, by DOS or by your program). |
| * 2CH          | WORD             | Segment number of the "environment". The environment contains ASCII strings defined by the DOS SET command (and other commands).                               |
| * 5CH          | 16 BYTES         | First FCB created from DOS command line.   |
| * 6CH          | 16 BYTES         | Second FCB created from DOS command.   |
| 80H            | BYTE             | Length of the DOS command line.  |
| 81H            | 127 BYTES        | The DOS command line itself, <i>except</i> for the name of the program being executed (which has been removed).  |

PROGRAM TO DISPLAY THE PARAMETERS PASSED TO THE PROGRAM

```

PUSH DS          ; SAVE DS.
PUSH ES          ; PUT ES INTO DS SINCE THE DOS DISPLAY
POP  DS          ; FUNCTION REQUIRES THE MESSAGE TO BE IN THE
                  ; DATA SEGMENT.
MOV  AH,40H      ; DOS WRITE-RECORD FUNCTION
MOV  BX,1        ; WRITE TO STANDARD OUTPUT HANDLE.
MOV  CL,DS:80H   ; GET THE BYTE COUNT INTO CX.
MOV  CH,0
MOV  DX,81H      ; PUT OFFSET OF BUFFER INTO DX.
INT  21H
POP  DS          ; RECALL DS.

```

PROGRAM TO CHECK FOR "SWITCH" CHARACTERS

```

; CHECK FOR SWITCH CHARACTER USING 8088 STRING INSTRUCTIONS. WE
; DO NOT HAVE TO SET ES=DS AS ABOVE SINCE WE USE THE "DESTINATION
; STRING", WHICH IS ALWAYS ADDRESSED BY ES:DI.
CLD              ; SET STRING FUNCTIONS TO INCREMENT.
MOV  DI,81H      ; STARTING OFFSET OF STRING.
MOV  CL,ES:80H   ; GET LENGTH OF STRING INTO
MOV  CH,0        ; CX.
JCXZ NOSWITCH    ; IF LENGTH IS ZERO, THEN NO SWITCH.
MOV  AL,'/'      ; CHECK FOR "/"
REPNE SCASB      ; USING THE "SCAN" FUNCTION.
JNE  NOSWITCH    ; IF NOT FOUND, THEN NO SWITCH.
YESSWITCH:       ; OTHERWISE,...

```

LECTURE NOTES FOR CS-5330: IBM PC ASSEMBLY LANGUAGE  
UNIVERSITY OF TEXAS AT DALLAS

## CLASS 14

Comments

1. Although I made a big fuss a few weeks ago over somebody crashing my system with a program, I have done the same thing. The SHORTLIB program which I mentioned to you last week apparently does not run on a TI, even though it runs on an IBM PC. Sorry about that.

2. As you know, we are scheduled to begin the final project next Monday. Some of you have ideas for projects you would like to do. I would like all of you to think about it and to come up with some ideas. Don't worry if your idea might involve something we haven't discussed in class. I will endeavor to get you any information you need for the project (or, at least, to help you get such information). Don't worry if the project seems too hard or too easy (I will decide whether it's too hard or too easy). Remember that you have three weeks to work on it, so it won't be trivial by any standard. Cooperative projects are okay -- i.e., if you have a project which can be modularized for several of you to work on, it might be an acceptable idea. Here are some reasonably fertile areas for projects:

- a) Computer graphics. Although we have not yet discussed graphics on the PC, we will begin to do so next week, and so information is no problem.
- b) Additional DOS utilities. We have already developed a "file dump" program and a Wordstar-to-ASCII conversion utility. There is a lot of room for improvement in this area.
- c) Additional database-type utilities. We are in the process of developing a sorting program. Additional database type utilities like file encryptors, data compression programs, etc., would be interesting.
- d) "Infinite precision" arithmetic routines. An "infinite precision" arithmetic scheme does not use numbers of fixed sized -- i.e., byte or word. Rather, the size is adjustable. Thus, the product of a word times a byte is 3 bytes. Or a product of a 50-byte number times a 75-byte number is 125 bytes.
- e) Mathematically oriented programs. For example, programs to locate prime numbers rapidly.
- f) Number crunching. These are projects involving the 8087 numeric coprocessor, which we will begin discussing in a moment.
- g) "Background" features. Although we have not discussed interrupts in class, there are many interesting things to be done with interrupt-controlled features of the computer (like real-time clocks), as opposed to programs as such.

- h) "Device drivers". Again, something we have not discussed in class, although possibly the most significant feature of MS-DOS. An installable device driver is a way of introducing new device names (like CON, PRN, etc.) into the system. These "devices" have an almost unlimited range of possible functions, and have the tremendous advantage that they can be accessed in a uniform way (via their "filename") from any high-level or low-level language.
- i) etc.

If you have no inclination to move out into these territories, I will have several generic projects available, which I will assign at random. Remember, a project you select could be much easier than one that I select.

### Review

In the previous class, we discussed several miscellaneous items.

We discussed the format used to store strings in memory in the mid-term project. We also discussed comparison of strings stored in this format, and saw a macro that would do the job.

We discussed the system librarian program, LIB. LIB is used combine assembled "object modules" (i.e., .OBJ files) into libraries. With such libraries, we could then eliminate the .OBJ files from our disks. This had the advantage of uncluttering our directories and freeing up a lot of disk space. The library of object modules could also be used with LINK in place of all the individual .OBJ files.

We discussed "jump tables" and their uses. A jump table is a word (or doubleword) array of addresses. Because the JMP and CALL instructions are capable of using any addressing mode to specify the location to jump to (not just immediate mode as we have always done before), the program can pick out (from the table) a particular address to jump to, using an indexed addressing mode. For example, with the jump-table

```
TABLE    DW    BACKSPACE
         DW    TAB
         DW    LINEFEED
```

the instruction

```
        JMP  TABLE[SI]
BACKSPACE:
TAB:
LINEFEED:
```

would jump to any of the labels BACKSPACE, TAB, or LINEFEED, depending on whether SI is 0, 2, or 4. Similarly, if the entries in the table are names of procedures rather than names of labels, we could use a

```
        CALL TABLE[SI]
```

to CALL the selected procedure.

Finally, we discussed the Program Segment Prefix (PSP). The PSP is a 256 (100H) header that is appended to the beginning of every program we run. It helps the program interface to DOS. In particular, it allows DOS to pass various items of information to the program. The most interesting item from our standpoint is that a buffer in the PSP contains the "parameters" specified when the program name was typed in at the DOS prompt. The "parameters" are simply everything on the command line except the prompt itself and the program name. Thus, in the command

```
A>MASM PROG7,,PROG7;
```

the "parameters" are contained in the string " PROG7,,PROG7;". When MASM executes, it finds this string in a buffer beginning at offset 81H in the PSP. A count of the bytes in the parameter string is at offset 80H in the PSP. The PSP itself is located at ES:0 when the program begins executing, so the count and the buffer are (respectively) at ES:80H and ES:81H.

#### Overview of the 8087 Numeric Coprocessor

The greatest lack in microcomputers in general (these days) is the lack of a built-in floating-point arithmetic instruction set. We have seen so far how it is possible to use the 8088 to perform reasonably flexible integer arithmetic on words or on bytes. Unfortunately, these data types do not by any stretch of the imagination begin to cover the data types used in business or industry. For these applications (and for technical applications) we also need integer arithmetic of much higher precision, and floating point arithmetic. Word arithmetic deals with numbers in the range 0 to 65535 (or -32768 to 32767), and this is clearly inadequate (for instance) if we are writing software to compute the earnings of a ten-million dollar corporation.

Higher precision arithmetic can, of course, be provided by the software. You have seen in chapter 4 how to perform doubleword arithmetic (including multiplication and square roots), which has twice the precision of ordinary word-arithmetic. Doublewords express values in the range of approximately -2 billion to +2 billion. Still, this wouldn't be sufficient for the finances of companies like IBM, AT&T, or the U.S. government, and still less so for the staggeringly large quantities found in science. Even worse, software arithmetic is tremendously slow in comparison to arithmetic performed by hardware. A word-multiply by the 8088 hardware (using the built-in MUL instruction) takes less than 30 microseconds, whereas the software doubleword multiply procedure given on p. 153 in the book takes about 200 microseconds. For higher precision than this, or for floating point arithmetic, the situation becomes much worse still. The manufacturer's literature (from Intel) for the 8087 chip (which we will discuss shortly) makes the following estimates of the time involved in single precision floating point arithmetic when: a) the 8087 chip is used (i.e., the arithmetic is done in hardware); and b) 8086 software is used. (The 8086 being a somewhat faster version of the 8088.) All times are in microseconds:

| <u>INSTRUCTION</u> | <u>8087</u> | <u>8086</u> |
|--------------------|-------------|-------------|
| Multiplication     | 19          | 1600        |
| Addition           | 17          | 1600        |
| Division           | 39          | 3200        |
| Comparison         | 9           | 1300        |
| Load               | 9           | 1700        |
| Store              | 18          | 1200        |
| Square Root        | 36          | 19600       |
| Tangent            | 90          | 13000       |
| Exponentiation     | 100         | 17100       |

While this table reflects a lot of optimism in the 8087 column and a lot of poor software in the 8086 column, it does get across a true point: Namely that if speed is crucial, then the 8088 microprocessor cannot be regarded (in comparison to devices performing arithmetic in hardware) as an effective tool for performing high-precision arithmetic -- the software is simply too slow. Even the simplest software floating point operations require over a millisecond (1/1000 second) to perform.

Although you have read Chapter 4, which covers software for multiple-precision arithmetic, we will not cover this material in class, nor do any problems reflecting the material. The reason for this is not merely that such software is bad -- it is that it is no longer cost-effective for programmers to write arithmetic software for 8088-based computers. Today, it is possible (and cost-effective) for all arithmetic to be done in the computer hardware. This situation has come about because of a chip manufactured by Intel -- the 8087 numeric coprocessor extension chip. The 8087 can perform fast, accurate arithmetic on many data types, is easy to program, can be plugged into an IBM PC in about 5 minutes, and costs less than \$100 (retail). Also, it is almost the sole advantage (aside from larger memory) of the IBM PC over earlier microcomputers. To go further, it is the sole advantage of the IBM PC over 16-bit microcomputers based on other CPUs than the 8086 or 8088. We will discuss the 8087 in much more detail in the remainder of the lecture, but the table of execution times we saw earlier gives some indication of the chip's abilities. As Richard Starz, the author of the standard book on the 8087 puts it, the 8087 "turns hours into minutes", and he is not far off in this assessment. (It is interesting to note that the 8087 can actually perform a double-precision floating point multiplication *faster* than the 8088 can perform a word-integer multiplication in hardware).

As indicated, the 8087 is capable of performing all of the standard arithmetic functions (+, -, \*, /) as well as many "transcendental" functions like the square root, logarithm, exponential, and some trigonometric functions. It also utilizes many new data types. Here is a list of 8087 data types (the numbers in the "range" column being only approximate):

| <u>DATA TYPE</u> | <u>BITS</u> | <u>DIGITS</u> | <u>RANGE</u>                 |
|------------------|-------------|---------------|------------------------------|
| Word Integer     | 16          | 4             | -32768 to 32767              |
| "Short" Integer  | 32          | 9             | -2 billion to +2 billion     |
| "Long" Integer   | 64          | 18            | -9E18 to 9E18                |
| Short Real       | 32          | 6             | 1E-37 to 1E38                |
| Long Real        | 64          | 15            | 1E-307 to 1E308              |
| Temporary Real   | 80          | 19            | 1E-4932 to 1E4932            |
| Packed Decimal   | 80          | 18            | 18 decimal digits and a sign |

Only the word data-type is present on the 8088, so the addition of these types is a tremendous extension. In point of fact, the 8087 (which does nothing more than arithmetic) is a much more complex device than the "general purpose" 8088 microprocessor that it extends.

Unfortunately, (as I understand it) very few of the computers in the micro lab are equipped with an 8087, so we cannot have any assigned work using it. However, it would be fun for a few of you to have a final project involving the 8087. In the 8087 discussions that follow we will assume that an 8087 has been added to the computer, even though this does not reflect the general situation in the micro labs.

#### Programmer's Model of the 8087

As far as the IBM PC is concerned, the 8087 is not a *device* -- i.e., it is not addressed through I/O ports as are various hardware devices we will study later. Rather, it is an *extension* of the CPU. When an 8087 is added to the system, it seems to the programmer as if the 8088 had simply been upgraded to include many new registers, new instructions, and new data types. Early versions (say 1.0) of the Macro Assembler did not support these new instructions or data types. Fortunately, MASM version 3.x supports all 8087 instructions and data types; DEBUG also supports the 8087 to a certain extent.

Let us first discuss how the assembler deals with all of these new data types. After that, we will discuss some of the new instructions.

First, a trivial case. As you might expect, a variable of the WORD-INTEGGER type is simply defined with DW statement, as always. The 8087 word data type is compatible with the 8088 word data type and need not be discussed further.

On the other hand, SHORT-INTEGGER and SHORT-REAL (single precision) variables, which are both four bytes long, are defined using the DD ("define doubleword") statement. You may recall that DD is also used to define variables whose values are segment:offset pairs of words. In fact, DD is rather flexible and is able to set up all three of these distinct (though all 4-byte) data types. It does this by examining the value and seeing what type it must be. For example, consider the following uses of the DD statement:

AGAIN:

```

...
FOO    DD    100.0    ; DEFINE A SHORT REAL.
BAR    DD    100      ; DEFINE A SHORT INTEGER.
REAL   DD    AGAIN   ; DEFINE A SEGMENT:OFFSET.
FAT    DD    3.1E17   ; DEFINE A SHORT REAL.

```

The DD operator is able to deduce that the first and last assignments are real since they contain decimal points. BAR must be an integer since it is a number with no decimal point. REAL must be a segment:offset since its value is a label, which can only represent an address. In this case, the word at REAL will contain the offset of AGAIN, while the word at REAL+2 will contain the segment. Notice that the bytes making up the variables FOO and BAR are totally different, even though the value represented is the same.

The LONG-INTEGERS and LONG-REAL (double precision) data types each occupy 8 bytes and are defined by the DQ ("define quadword") operator. We have not mentioned DQ previously, but it is very similar to the DD and DW operators. Here are some obvious examples:

```
FOO      DQ      100.0      ; DEFINE A LONG REAL.
BAR      DQ      100        ; DEFINE A LONG INTEGER.
```

Finally, the TEMPORARY-REAL and PACKED-DECIMAL (or BCD) types are defined with the DT ("define tenbyte") operator. The TEMPORARY-REAL is just a floating point form with slightly extended precision and a vastly extended range of exponents. The PACKED-DECIMAL type is reminiscent of the integer types (it consists of 18 decimal digits plus a sign), but it has a different memory representation. The PACKED-DECIMAL type consists of ten bytes; one byte is the "sign" byte (it is either zero or 80H), and all of the other bytes represent two BCD digits each. In any case, here are some examples of how to use the DT operator:

```
FOO      DT      100.0      ; DEFINE A TEMPORARY REAL.
BAR      DT      100        ; DEFINE A PACKED DECIMAL.
```

None of these data types is understood by the 8088 alone. However, as mentioned, adding an 8087 to the system also adds many instructions to the instruction set. These new instructions are able to deal with the new data types. We will discuss some of these new instructions in the next section.

Before we can understand any new instructions, however, we have to understand the 8087 register set -- i.e., the new registers added to the CPU by adding an 8087 to the system. All of the new instructions use the new registers (none of them use the *old* 8088 registers), and many of them also use memory variables of the new data types.

The 8087 has 8 80-bit registers. These huge (by 8088 standards) registers are each capable of holding a number in the largest (in terms of bytes) and most precise data type -- the TEMPORARY-REAL type. Note that any data type can be converted to TEMPORARY-REAL without loss of precision. The 8087 takes advantage of this fact to internally store all numbers in TEMPORARY-REAL form. Thus, internally, all 8087 operations deal only with the TEMPORARY-REAL form, and there is no need for the explicit type-conversion operations so common in higher level languages. All type-conversions to and from TEMPORARY-REAL and the other data types occur when data is loaded into the 8087 registers from memory, or stored into memory from the 8087 registers.

The 8087 registers form a stack, much like the 8088's stack (but with TEMPORARY REAL values rather than word values). The registers are

designated as ST(0) [or just ST], ST(1), ST(2), ..., ST(7). ST(0) is the register at the top of the stack, while ST(7) is the register at the bottom of the stack. If new data is loaded into the 8087, the new data is put at the top of the stack and becomes register ST(0). The old ST(0) register becomes ST(1), the old ST(1) becomes ST(2), etc. Finally, the old value in ST(7) just disappears. Actually, the register which was previously addressed as ST(7) is now addressed as ST(0) and the new data simply overwrites the old ST(7) data. This way of working should be very familiar to anyone used to the FORTH programming language, or to Hewlett-Packard calculators.

Some Simple 8087 Instructions: Loading and Storing

**NOTE:** in what follows we will discuss how various instructions are used, but we will not in general discuss the timing of these instructions. This is meant to complement the information in the book. On pp. 280-282 of the text is a table which gives the timing of every instruction, but unfortunately neglects telling how to use a single one of them!

In this section, we discuss the 8087 instructions FLD, FILD, FBLD, FSTP, FISTP, and FBSTP. The first three instructions are analogous to 8088 PUSH instructions, except that data is pushed onto the 8087 register stack rather than the 8088 stack. Similarly, the latter three instructions are similar to the 8088 POP instructions, in that they pop data from the 8087 register stack.

The command for loading new (REAL, not INTEGER or PACKED) data into the 8087 is the FLD command. All 8087 commands begin with "F" (probably to indicate "Floating point"). The general syntax of FLD is

FLD *source*

where the source operand is any 8088 addressing mode except register or immediate. Suppose we had the following data and program:

```
FOO1      DD      100.0          ; SINGLE PRECISION 100.
FOO1.5    DD      ?              ; AN UNUSED DOUBLEWORD.
FOO2      DQ      100.0          ; DOUBLE PRECISION 100.
FOO3      DT      100.0          ; TEMPORARY REAL 100.
.
.
.
FLD      foo          ; 8087 LOAD INSTRUCTION.
```

where *foo* represents any of FOO1, FOO2, FOO3. The FLD instruction would read either FOO1, FOO2, or FOO3 from memory and *push* the value onto the 8087's stack of registers. In this case, each of the FOOs has the value 100, so this instruction would result in ST(0) being 100 and in all of the other ST(i) being pushed downward on the stack. As mentioned earlier, all data is internally stored in the 8087 in TEMPORARY REAL format, so the results of "FLD FOO1", "FLD FOO2", and "FLD FOO3" are identical. FLD automatically performs the type conversion from SHORT REAL or LONG REAL to TEMPORARY REAL as the data is loaded.

Similarly, the 8087 register stack can be *popped* (with the result being stored in memory) by the FSTP instruction. The general syntax is

FSTP *destination*

where, again, the destination can be any 8088 addressing mode except register. For instance, having already used the FLD instruction to make ST(0) equal to 100, we could then store the 100 at FOO1.5 by saying

FSTP FOO1.5

On the other hand, we could also use one of the other addressing modes. For example, we could assign SI to be 4 (recall that a SHORT REAL variable is 4 bytes long) and perform a

FSTP FOO1[SI]

Either of these instructions would result in the value 100 being popped from the 8087 register stack (and converted to SHORT REAL format), with all of the other ST(i) moving upward. Indeed, the register stack would be identical to the way it was before the original FLD instruction, except that the original value of ST(7) (i.e., the "bottom" of the stack) would have been destroyed by the FLD.

Another way of looking at these operations is by drawing a picture of the stack after the operations are performed. If we designate quantities with definite (but unknown) values with x0, x1, etc., then the stack behaves like this:

| <u>Instruction</u>  | <u>ST(0)</u> | <u>ST(1)</u> | <u>ST(2)</u> | ... | <u>ST(7)</u> |
|---------------------|--------------|--------------|--------------|-----|--------------|
| (initial condition) | x0           | x1           | x2           |     | x7           |
| FLD <i>foo</i>      | 100          | x0           | x1           |     | x6           |
| FSTP FOO1.5         | x0           | x1           | x2           |     | ?            |

The final question mark in the table indicates that the value is not the original x7, which has been destroyed. Of course, the effect of these instructions on the memory variables has not been shown, but all are unchanged except that FOO1.5 is overwritten with 100 in the final step.

The FLD and FSTP instructions load and store only REAL values. INTEGER and PACKED (or BCD) values have their own load and store operations. The INTEGER load operation is FILD. It can be used to push any INTEGER data type onto the register stack:

```
FOO1      DW  100      ; WORD INTEGER 100.
FOO1.5    DW  ?        ; UNKNOWN WORD INTEGER.
FOO2      DD  100      ; SHORT INTEGER 100.
FOO3      DQ  100      ; LONG INTEGER 100.
.
.
.
FILD foo      ; PUSH ANY FOOn ONTO REGISTER STACK.
FISTP FOO1.5  ; STORE TOP OF STACK (100) INTO FOO1.5.
```

There is no need to discuss this further, since there is almost no difference from the FLD and FSTP instructions described earlier. As always, the 100 pushed onto the stack by FILD is in the TEMPORARY REAL format, so the 8087 must convert the data type for both of these instructions.

The BCD (PACKED) data type is pushed and popped with the FBLD and FBSTP instructions:

```
FOO1      DT   100      ; PACKED 100.
FOO2      DT   ?       ; UNKNOWN VALUE.
.
.
.
          FBLD FOO1      ; PUSH 100 ONTO THE REGISTER STACK.
          MOV  SI,10
          FBSTP FOO1[SI] ; POP AND STORE AT FOO2.
```

Here I have used an indexed addressing mode just to emphasize again that any addressing mode can be used other than 8088 register mode or immediate mode.

Even though we have not yet learned how to use the 8087 to perform any arithmetic, these load and store instructions are by themselves useful, by virtue of the fact that they automatically convert from one data type to another. In general, with the 8088 we perform most of our arithmetic in binary (word) form, and converting back and forth from the decimal representation is rather inconvenient and slow, particularly for doubleword or higher precision integers. By providing a PACKED (BINARY CODED DECIMAL) form which is much closer to the normal decimal representation, the 8087 does most of our conversion work for us (and more quickly, too). In a binary coded decimal (BCD) representation, each byte of the number represents two decimal digits. Each decimal digit (0-9) occupies one nibble of the byte. For example, the BCD representation of 45 (decimal) is 45H. The BCD representation of 100 (decimal) is 0100H. The BCD representation of 65536 (decimal) is 065536H.

Clearly, converting a BCD number to decimal (ASCII) is very simple: We simply separate off the nibbles of each byte, and convert each nibble to a decimal digit by adding '0'. In the case of the 8087, the PACKED (or BCD) data type consists of ten bytes. The first nine bytes are BCD bytes in increasing order of significance. The last byte is the "sign" byte. It is zero for a positive number and 80H for a negative number. Thus, in memory, the number -123456789 would be represented in PACKED DECIMAL form as the bytes

```
89H, 67H, 45H, 23H, 01H, 00H, 00H, 00H, 00H, 80H
```

Of course, several niceties suggest themselves if we actually wanted to display the number: we would want to strip off any leading zeros, but we would always want to print one '0' if the result actually is zero. For instance (for the number being discussed), we would want to display "-123456789" rather than "-000000000123456789".

In any case, the first step of the display procedure would have to be to convert the number to be displayed into BCD form. This is made

trivial by the 8087's automatic type-conversion feature. For instance, to convert the WORD INTEGER FOO to the PACKED DECIMAL BAR, we would simply use the instructions

```

        FILD      FOO          ; CONVERT TO TEMPORARY REAL.
        FBSTP    BAR          ; CONVERT TO PACKED DECIMAL.

```

Here is a simple 8088 procedure to complete the display process and show the number on the screen, assuming that the argument passed on the stack is the *address* of the buffer containing the BCD representation of the number:

```

; PUTDEC is assumed to be a macro which converts its argument (a
; number from 0 to 9) to a character from '0' to '9' and displays it.
; With no argument, it uses AL.
begin disp_packed,2
irp x,<i,flag,,bcd>
    var x
endm
digit equ byte ptr [si]          ; used to access the bytes.

fwait                            ; wait for the 8087 to finish.
mov flag,0                       ; while 0, there could be leading 0s.
mov si,bcd                       ; get address of the bcd number.
add si,9                         ; point si at last byte of number.
mif
    test digit,80H              ; minus sign?
mthen nz
    putchr '-'                 ; if so, display it.
melse
mendif
mov i,10
repeat                            ; first, need to get rid of all
    dec si                    ; leading zeros. We begin by
    mif                      ; counting down from the most
        cmp i,1              ; significant byte until we find
    mthen be                 ; a byte which is non-zero. (This
        putchr '0'          ; byte could still have an upper
        exit                ; nibble of zero, but it's a start
    melse                   ; anyhow.) If we count down all the
        dec i                ; way to the last digit and they're
        cmp digit,0         ; all zero, simply display a '0'
    mendif                  ; and quit.
until ne
repeat                            ; Now that we've reached a non-zero
    mif                      ; byte, split it up into nibbles
        mov al,digit        ; and convert to ASCII by adding
        mov cl,4           ; '0'. Note that if FLAG is still
        shr al,cl          ; zero and the upper nibble is zero
    mthen nz                 ; then it's a leading zero and should
        putdec              ; be suppressed.
    melse
    mif
        cmp flag,0
    mthen nz
        putdec 0
    melse

```

```

    mendif
mendif
mov flag,1           ; In any case, set FLAG=1 so no more
mov al,digit        ; digits are suppressed.
and al,0fh          ; Do lower nibble now.
putdec
dec si
dec i
until z
return 1

```

There is nothing particularly novel about this program, except that it contains a new 8087 instruction FWAIT, which we have not seen before. To explain FWAIT we have to understand a little more about how the 8087 operates.

While installation of the 8087 appears to the programmer (in many situations) merely to have added some registers to the 8088 and to have upgraded its instruction set, there are other situations in which it becomes apparent that the 8087 really is a separate processor from the 8088. The key point to understand is that many 8087 operations take a very long time (relative to most 8088 instructions), and it would be wasteful to simply keep the 8088 idle while this processing occurs. Instead, the 8088 and 8087 actually execute *simultaneously*, with each working on its own assigned task. The 8087 executes all of the 8087 instructions, and the 8088 executes all of the 8088 instructions. Let's consider a simple example of this. In the program

```

    FLD  FOO
    MOV  CX,10
AGN: LOOP AGN

```

the 8087 does not simply passively wait for the 8088 to finish executing the FLD command. Instead, it goes ahead and begins executing the indicated loop. The FLD instruction requires about 50 clock cycles to execute (for a SHORT REAL FOO), while the 8088 MOV instruction indicated takes about 4 clocks and the LOOP takes about 17. Thus, *the 8088 is already on its third iteration of the loop by the time the 8087 finishes executing the FLD*. This being the case, it is important that the 8087 and 8088 do not simultaneously try to access the same bytes of memory. Otherwise, one processor could modify a word before the other is finished reading it, and chaos could ensue. (Of course, if both processors are simply *reading* the bytes, there is no problem.) One way to ensure this synchronization between the processors is to use the FWAIT instruction. The FWAIT instruction tells the 8088 to wait until the 8087 is finished with its current instruction before continuing. In the program we have been discussing, in which a PACKED DECIMAL number is converted to ASCII and displayed, the FWAIT is used for exactly this. Assuming that the PACKED DECIMAL number was created by an 8087 FBSTP instruction in the first place, the FWAIT instruction ensures that the FBSTP instruction has terminated before attempting any conversion.

Taking another simple example, if we wanted to write a new value into FOO immediately after using FLD to load FOO into the 8087, we could not simply say something like this:

```
FLD  FOO
MOV  WORD PTR FOO,AX
```

Instead, we would have to ensure that the FLD had finished with FOO before continuing:

```
FLD  FOO
FWAIT
MOV  WORD PTR FOO,AX
```

This synchronization is not a problem if we stick exclusively to 8087 instructions. For example, we could say

```
FLD  FOO
FSTP FOO
```

without fear of problems. The assembler *automatically* encodes an FWAIT in front of every 8087 instruction, so we ourselves do not have to worry about it.

The fact that the 8088 and 8087 execute simultaneously and independently is sometimes of profound importance. Such co-processing allows the 8088 to perform rather complex processing with no runtime penalty! Since the program has to wait for the 8087 anyway, there is no reason why it can't do some useful processing as well in the meantime. We will possibly see some examples of this later.

Although (for simplicity's sake) we did not mention it before, the FLD and FSTP commands (*not* the INTEGER or PACKED equivalents) can also use 8087 registers as sources or destinations of data. In particular, "FLD ST(0)" makes a copy of ST(0) and then pushes it onto the stack -- i.e., it duplicates the top of the stack. For example,

| Instruction         | ST(0) | ST(1) | ST(2) | ST(3) | ... | ST(7) |
|---------------------|-------|-------|-------|-------|-----|-------|
| (initial condition) | 100   | 200   | 300   | 400   |     | 800   |
| FLD ST(1)           | 200   | 100   | 200   | 300   |     | 700   |
| FLD ST(0)           | 200   | 200   | 100   | 200   |     | 600   |
| FSTP ST(2)          | 200   | 200   | 200   | 300   |     | ?     |

Notice in the last example that the top of the stack (200) was copied to ST(2) (which was 100 to begin with), and then the top of stack was popped. The old ST(1) (which happened to be 200) thus became the top of stack, ST(2) (which we forced to be 200) became ST(1), and ST(3) (which happened to be 200) became ST(2). ST(4) through ST(6) (which are not shown) are 400, 500, and 600. The instruction "FSTP ST(0)" can be used to simply pop the stack and throw away the data, since it performs a meaningless operation [copies ST(0) to itself] and then pops the stack.

There are two other 8087 "store" operations which I haven't mentioned yet. One is the FST instruction. It is a "store" instruction alone, and not a "store and pop". Many 8087 instructions have two versions, one which pops the stack after completion and one which doesn't. The version of the instruction which pops is always designated by a trailing "P". Thus, "FST" doesn't pop, "FSTP" does. FST is much like FSTP, except for the popping, so we can use it to store *several* copies of the same data in memory:

```

FOO1      DD      ?
FOO2      DQ      ?
          FST      foo
          FST      foo[SI]

```

There is one more difference, however, and that is that FSTP does not work with TEMPORARY REAL values in memory. (Thus, no FOO with a DT.) In point of fact, no 8087 instruction (other than those we've mentioned) can deal with a TEMPORARY REAL value in memory. *Only the FLD and FSTP instructions can use a TEMPORARY REAL value in memory.* All other REAL 8087 instructions use only the SHORT and LONG REAL types. Similarly, there is an FIST instruction, which is like the FISTP (INTEGER store) instruction, except that it does not pop the stack and that it uses only WORD INTEGER and SHORT INTEGER data in memory. It cannot use LONG INTEGER data in memory. *Only the FILD and FISTP instructions can use a LONG INTEGER value in memory.* All other INTEGER 8087 instructions use only WORD INTEGER and SHORT INTEGER types. There is no non-popping equivalent of the FBSTP instruction for PACKED DECIMAL. *Only the FBLD and FBSTP instructions can use PACKED DECIMAL values in memory.*

Let's summarize all of this in an easy to understand form:

| <u>DATA TYPE IN MEMORY</u> | <u>INSTRUCTIONS USING IT</u> |
|----------------------------|------------------------------|
| WORD INTEGER               | ALL INTEGER INSTRUCTIONS     |
| SHORT INTEGER              | ALL INTEGER INSTRUCTIONS     |
| LONG INTEGER               | ONLY FILD AND FISTP          |
| SHORT REAL                 | ALL REAL INSTRUCTIONS        |
| LONG REAL                  | ALL REAL INSTRUCTIONS        |
| TEMPORARY REAL             | ONLY FLD AND FSTP            |
| PACKED DECIMAL             | ONLY FBLD AND FBSTP          |

Once inside the 8087, however (as mentioned several times before), all data becomes TEMPORARY REAL.

SOME IDEAS FOR FINAL PROJECTS

As you know, we are scheduled to begin the final project next Monday. Some of you have ideas for projects you would like to do. I would like all of you to think about it and to come up with some ideas. Don't worry if your idea might involve something we haven't discussed in class. I will endeavor to get you any information you need for the project (or, at least, to help you get such information). Don't worry if the project seems too hard or too easy (I will decide whether it's too hard or too easy). Remember that you have three weeks to work on it, so it won't be trivial by any standard. Cooperative projects are okay -- i.e., if you have a project which can be modularized for several of you to work on, it might be an acceptable idea. Here are some reasonably fertile areas for projects:

- a) Computer graphics. Although we have not yet discussed graphics on the PC, we will begin to do so next week, and so getting information (for a TI or an IBM) is no problem.
- b) Additional DOS utilities. We have already developed a "file dump" program and a Wordstar-to-ASCII conversion utility. There is a lot of room for useful utilities.
- c) Additional database-type utilities. We are in the process of developing a sorting program. Additional database utilities like file encryptors, data compression programs, etc., would be interesting.
- d) "Infinite precision" arithmetic routines. An "infinite precision" arithmetic scheme does not use numbers of fixed sizes -- i.e., byte or word. Rather, the size is adjustable. Thus, the product of a word times a byte is 3 bytes. Or: the product of a 50-byte number times a 75-byte number is 125 bytes.
- e) Mathematically oriented programs. For example, programs to locate prime numbers rapidly.
- f) Number crunching. These are projects involving the 8087 numeric coprocessor, which we will begin discussing in a moment.
- g) "Background" features. Although we have not discussed interrupts in class, there are many interesting things to be done with interrupt-controlled features of the computer (like real-time clocks), as opposed to programs as such.
- h) "Device drivers". Again, something we have not discussed in class, although possibly the most significant feature of MS-DOS. An installable device driver is a way of introducing new device names (like CON, PRN, etc.) into the system. These "devices" have an almost unlimited range of possible functions, and have the tremendous advantage that they can be accessed in a uniform way (via their "filename") from any high-level or low-level language.

If you have no inclination to move out into these territories, I will have several generic projects available, which I will assign at random. Remember, a project you select could be much easier than one that I select for you (particularly if your work has been good).

8087 REFERENCE SHEET 1

*Hardware floating point (8087) vs. software floating point (8086).  
Execution time is in microseconds. (REF: Intel literature.)*

| <u>INSTRUCTION</u> | <u>8087</u> | <u>8086</u> |
|--------------------|-------------|-------------|
| Multiplication     | 19          | 1600        |
| Addition           | 17          | 1600        |
| Division           | 39          | 3200        |
| Comparison         | 9           | 1300        |
| Load               | 9           | 1700        |
| Store              | 18          | 1200        |
| Square Root        | 36          | 19600       |
| Tangent            | 90          | 13000       |
| Exponentiation     | 100         | 17100       |

*8087 data types. The 8088 has only byte and word integer. (Ranges are only approximate.)*

| <u>DATA TYPE</u> | <u>BITS</u> | <u>DIGITS</u> | <u>RANGE</u>                 |
|------------------|-------------|---------------|------------------------------|
| Word Integer     | 16          | 4             | -32768 to 32767              |
| Short Integer    | 32          | 9             | -2 billion to +2 billion     |
| Long Integer     | 64          | 18            | -9E18 to 9E18                |
| Short Real       | 32          | 6             | 1E-37 to 1E38                |
| Long Real        | 64          | 15            | 1E-307 to 1E308              |
| Temporary Real   | 80          | 19            | 1E-4932 to 1E4932            |
| Packed Decimal   | 80          | 18            | 18 decimal digits and a sign |

*Use of 8087 memory variables by 8087 instructions:*

| <u>DATA TYPE IN MEMORY</u> | <u>INSTRUCTIONS USING IT</u> |
|----------------------------|------------------------------|
| WORD INTEGER               | ALL INTEGER INSTRUCTIONS     |
| SHORT INTEGER              | ALL INTEGER INSTRUCTIONS     |
| LONG INTEGER               | ONLY FILD AND FISTP          |
| SHORT REAL                 | ALL REAL INSTRUCTIONS        |
| LONG REAL                  | ALL REAL INSTRUCTIONS        |
| TEMPORARY REAL             | ONLY FLD AND FSTP            |
| PACKED DECIMAL             | ONLY FBLD AND FBSTP          |

*Load operations:*

|      |                 |  |
|------|-----------------|--|
| FILD | <i>variable</i> | ; push INTEGER <i>variable</i> onto stack. |
| FLD  | <i>variable</i> | ; push REAL <i>variable</i> onto stack.    |
| FLD  | ST( <i>i</i> )  | ; push copy of ST( <i>i</i> ) onto stack.  |
| FBLD | <i>variable</i> | ; push PACKED <i>variable</i> onto stack.  |

*Store operations:*

|       |                 |  |
|-------|-----------------|--|
| FISTP | <i>variable</i> | ; pop INTEGER <i>variable</i> from stack.  |
| FSTP  | <i>variable</i> | ; pop REAL <i>variable</i> from stack.     |
| FSTP  | ST( <i>i</i> )  | ; copy ST to ST( <i>i</i> ) and pop stack. |
| FBSTP | <i>variable</i> | ; pop PACKED <i>variable</i> from stack.   |

*There are also FIST and FST instructions which store but do not pop.*

The four arithmetic operations. op=ADD, SUB, MUL, or DIV:

Perform an operation:

```

FIop      variable      ; ST <- ST op INTEGER variable
Fop       variable      ; ST <- ST op REAL variable
Fop       ST,ST(i)      ; ST <- ST op ST(i)
Fop       ST(i),ST      ; ST(i) <- ST(i) op ST
FopP      ST(i),ST      ; ST(i) <- ST(i) op ST and pop ST

```

If op=SUB or DIV, then there are also FIopR, FopR, and FopRP instructions, which are the same as those above except that the order of the operation is reversed. For example:

```

FopR      ST,ST(i)      ; ST <- ST(i) op ST

```

LECTURE NOTES FOR CS-5330: IBM PC ASSEMBLY LANGUAGE  
UNIVERSITY OF TEXAS AT DALLAS

## CLASS 15

Comments

0. Unfortunately, there is a bug in the Pascal sorting algorithms I have handed out, even though I thought I had thoroughly tested them. The Heapsort algorithm contains several typos. If you are *not* using the Heapsort (as given by the handout), make a notation on your handout that the algorithm is incorrect. If you are using it, then come to me to for the correct version.

0.5. Even worse, there is a bug in the insertion sort! I forgot to tell you that for the insertion sort to work, the zeroeth pointer in the pointer array (counting the pointer to the first line as the *first* pointer) must point to a *sentinel* line which is smaller than all of the text lines -- i.e., is of zero length. The entire problem can be fixed by putting the following two lines immediately prior to the DW instruction defining the pointer array:

```
ZERO DB 0      ; sentinel string of zero length.
      DW ZERO   ; pointer to the sentinel.
      ... (DW defining pointer array) ...
```

1. "Phase Error Between Passes". When using the assembler, many of you have seen the error message "phase error between passes", often accompanied by a large number of other errors. Usually, the "phase error" and most of the other messages constitute just *one* error, and not the apparently large number indicated by the assembler. To understand this, we have to understand something about the way the assembler works. The assembler is a "two pass" assembler. This means that the assembler does not simply read your source file, glance at it, and produce the correct object code. Rather, it has to process the source code *twice* to produce the correct object code. On the first pass (i.e., the first time through the code), the assembler computes various things like the length of each instruction and the location of each label. On the second pass through the code, it is then able to fill in many things that were unknown on the first pass. For example, in a program like

```
JMP  AGAIN
.
.
.
AGAIN:
```

the location of the label AGAIN is not known to the assembler when the instruction "JMP AGAIN" is reached on the first pass. Therefore, on the second pass, the assembler must supply the proper address for the JMP. Now, there are a number of ways to confuse the assembler so that some numbers calculated during the first pass are not equal to numbers computed during the second pass. Such a discrepancy is called a "phase error between passes". Here is an example of a phase error:

```
IF1
  MOV AX,5
ENDIF
```

AGAIN:

Since the instruction "MOV AX,5" is included only on the first pass, the position of all following labels is different during the second pass than it is during the first pass. Thus, when AGAIN is reached, a "phase error" message appears. There is no error message at the IF1/MOV/ENDIF, since the phase error has not yet been detected at that point, even though it is the IF1/MOV/ENDIF at fault. Thus, lesson 1 about phase errors is:

- 1) The phase error probably does not occur at the point indicated by the assembler. (However, it probably occurs before the next preceding label, or else the phase error would have been detected earlier.)

There is more, however. Once a phase error has occurred, the assembler becomes confused about a number of things, particularly about labels (since the locations have changed from pass 1 to pass 2). Therefore, it begins to generate many other errors at apparently fine instructions. Thus, lesson 2 about phase errors is:

- 2) If a phase error occurs, don't believe any of the other error messages until you fix the phase error. Rather, fix the phase error and re-assemble. Most of the other error messages are probably fictitious.

### Review

In the previous class, we began discussing the 8087 numeric coprocessor extension to the 8088 CPU. The 8087 is capable of performing fast hardware-based arithmetic on a number of new datatypes not present with the 8088 alone. In many ways, adding an 8087 to the system is like adding many new registers and instructions to the 8088.

The new datatypes are the SHORT INTEGER, which is 4 bytes long and defined with the DD operator, the LONG INTEGER, which is 8 bytes long and defined with the DQ operator, the PACKED DECIMAL, which is a 10 byte BCD form defined by the DT operator, the SHORT REAL (or single-precision floating point), which is 4 bytes long and defined with DD, the LONG REAL (or double-precision), which is 8 bytes long and defined with DQ, and the TEMPORARY REAL, which is 10 bytes long and defined with DT. In general, just a few instructions deal with LONG INTEGER, PACKED DECIMAL, and TEMPORARY REAL data items in memory. Most REAL instructions use only SHORT and LONG REALS in memory, while most INTEGER instructions use only WORD and SHORT INTEGERS in memory. Internally, however, all data actually stored in the 8087 is in TEMPORARY REAL format, and all data loaded by the 8087 are converted to this form.

The 8087 has 8 80-bit (TEMPORARY REAL) registers. These registers are arranged in a stack with the top of stack being ST(0) [or just ST], the next being ST(1), ..., down to ST(7). Three instructions are used to push data onto the register stack. The instruction

*FBLD source*

pushes the value of a PACKED DECIMAL memory variable onto the stack. The instruction

*FILD source*

pushes the value of a WORD, SHORT, or LONG INTEGER memory variable onto the stack. The instruction

*FLD source*

pushes the value of a SHORT, LONG, or TEMPORARY REAL memory variable or the value of a (TEMPORARY REAL) register onto the stack. All necessary type conversions to TEMPORARY REAL are taken care of by these instructions. Similarly, "*FBSTP destination*", "*FISTP destination*", and "*FSTP destination*" pop the stack, convert the number to the proper type, and store it at the indicated destination. Only these six instructions use LONG INTEGER, TEMPORARY REAL, and PACKED DECIMAL memory variables. All others use only WORD or SHORT INTEGER, or SHORT or LONG REAL memory variables, or (TEMPORARY REAL) registers.

We also discussed the instructions FST and FIST, which are like FSTP and FISTP except that they store the result *without* popping (and their data types are restricted as mentioned above).

The final topic discussed in the previous lecture was "synchronization" of the 8088 and 8087 processors. Although it often appears to the programmer as if the two processors simply constitute a single "souped-up" processor, there are times when it becomes apparent that the processors are actually running independently and simultaneously. This happens when the two processors are simultaneously trying to read and write the same memory locations. When this happens, the data in memory can be corrupted and the program can misbehave. The cure for this is the FWAIT instruction. The FWAIT instruction forces the 8088 to wait until the 8087 has completed executing its current instruction before continuing. This instruction is necessary only under the conditions mentioned; if the processors are accessing distinct memory locations (or are merely reading the locations rather than writing to them), no special effort need be made to maintain synchronization. This independent processing by the 8087 and 8088 is actually an advantage in many situations since it allows the 8088 to perform rather complex functions (with no runtime penalty) while the 8087 is crunching numbers.

The .8087 Pseudo-op and the FINIT instruction

In order to use 8087 instructions in your program, you must use the pseudo-op ".8087" before using any 8087 instructions. That is, you should use the line

```
.8087
```

near the beginning of your program. Another instruction which might be used near the beginning of your main program (not your procedures!) is the

FINIT

instruction. FINIT initializes the 8087 processor. Of course, you do not want to do this more than once, and particularly not *during* a calculation.

#### Less Primitive 8087 instructions: The Four Operations

Of course, the reason we are using the 8087 in the first place is probably that we'd like to do some arithmetic. For example, we'd probably like to do some addition, multiplication, subtraction, and division. That is the subject of this section.

Except as specified below there is generally no difference (except for the results) in using any of the four operations. That is, the syntax of the instructions does not vary among the operations. We will discuss a generic operation *op*, which you can imagine to be any of: ADD, SUB, MUL, or DIV. Here is a list of the 8087 instructions pertaining to the four arithmetic operations, leaving out the various operands of the instructions:

```

Fop
FopP
FopR           (Only for subtraction and division)
FIop
FIopR         (Only for subtraction and division)

```

(For example, here is a list of all addition operations: FADD, FADDP, and FIADD.)

*Fop* is the most commonly used operation. A *Fop* instruction has the syntax

```

Fop      variable    ; Add a REAL variable to ST(0).
Fop      ST,ST(i)    ; Add ST(i) to ST(0).
Fop      ST(i),ST    ; Add the ST(0) to ST(i).

```

As you might suppose from the discussion earlier, the memory variable here can only be of the SHORT REAL or LONG REAL types. For a memory variable which is WORD INTEGER or SHORT INTEGER, we must use instead the *FIop* instruction, which has the syntax

```

FIop      variable    ; Add an INTEGER variable to ST(0).

```

The other operand combinations we saw with *Fop* would make no sense for *FIop* since the operands would have to be 8087 registers (which always contain TEMPORARY REAL values and not integers). Except for TEMPORARY REALS in registers, the TEMPORARY REAL, PACKED DECIMAL, and LONG INTEGER data types cannot be used with any *Fop*-type instruction.

The use of these instructions is very straightforward, and surprises seldom crop up. For example, here is a simple program to calculate VAR0=VAR1\*VAR2+VAR3, where VAR1 is LONG REAL, VAR2 is SHORT INTEGER, VAR3 is SHORT REAL, and VAR0 is intended to be PACKED DECIMAL:

```

VAR0      DT          ?      ; SPACE FOR RESULT.
VAR1      DQ          7.0    ; MAKE VAR1 A LONG REAL 7.

```

```

VAR2      DD      12      ; MAKE VAR2 A SHORT INTEGER 12.
VAR3      DD      22.0    ; MAKE VAR3 A SHORT REAL 22.0.
...
FLD       VAR1      ; PUSH VAR1 ONTO STACK.
FIMUL     VAR2      ; MULTIPLY BY VAR2.
FADD      VAR3      ; ADD VAR3.
FBSTP     VAR0      ; STORE AS VAR0 AND POP.
    
```

Here we have used two of the load and store operations seen earlier, as well as an *Fop* (with *op=ADD*) and an *FIop* (with *op=MUL*). Other than the novelty of the new instructions, there is nothing here to startle the mind.

The instruction *FopP* is similar to *Fop* and *FIop*, except that the stack is popped after the operation is performed. Obviously, this doesn't make any sense if the destination of the result was the stack top *ST(0)* (or just *ST*, for short). Therefore, the only operand combination allowed is reflected by the following syntax:

```

FopP      ST(i),ST
    
```

The *FopP* form is very useful when the calculation is finished (or, at least, some step is finished), and intermediate results need to be disposed of. Consider, for example, a program to square a number on top of the register stack:

```

FMUL      ST,ST(0)
    
```

Here there are no intermediate results to dispose of. Suppose, however, that we wanted to compute the cube. We would have to do something like

```

FLD       ST(0)      ; DUPLICATE TOP OF STACK.
FMUL      ST(1),ST   ; MULTIPLY SECOND ON STACK BY TOP.
FMULP     ST(1),ST   ; DO IT AGAIN, AND DISPOSE OF THE TOP.
    
```

If, say, the number 2 is at the top of the stack, the stack would look like this during the calculation:

| <u>INSTRUCTION</u>  | <u>ST(0)</u> | <u>ST(1)</u> |
|---------------------|--------------|--------------|
| (initial condition) | 2            | ?            |
| FLD ST(0)           | 2            | 2            |
| FMUL ST(1),ST       | 2            | 4            |
| FMULP ST(1),ST      | 8            | ?            |

The process of taking the cube forced us to make an extra copy of the initial value, and this extra copy had to be disposed of in the end. Without a "popping" form of *FMULP* this could have been quite difficult.

You might wonder why the extra copy of the initial value was not simply stored in a memory variable. The answer to this is that because of the type conversions performed by the 8087 whenever a value is loaded from memory or stored to memory, the loading and storing process is rather slow. As an example of this, it takes about 90 clock cycles to add or multiply two 8087 registers, but it takes 50 clock cycles to load a register from memory to begin with, or 90 cycles to save the value in memory from the register. *That is, it can take longer to load*

the 8087 registers for an operation and to store the result than to do the operation itself! From this fact you can easily imagine that there is a great emphasis placed on arranging your calculations in such a way that as few 8087 memory accesses as possible are needed. Indeed, many high-level languages which use the 8087 are needlessly slow because they are unable to deal with this fact. Compilers generally feel that the place for variables is in memory, so they are continually storing and reloading intermediate results rather than just keeping them in 8087 registers. (Many compilers have other problems that make number crunching inefficient. Some compilers, for example, rather than simply using 8087 instructions "inline", will call procedures which in turn use the 8087 instructions. In this case, the execution time overhead of the calling sequence for these procedures can be as large as the time to execute the 8087 function!) An assembly-language programmer can usually beat even the fastest compiler, increasing execution speed by a factor (say) of three, if many 8087 operations are involved. To sum all this up in a few words:

**AS MUCH AS POSSIBLE, KEEP ALL INTERMEDIATE RESULTS IN 8087 REGISTERS, RATHER THAN IN MEMORY.**

The final elementary arithmetic operations,

```
FopR      variable
FIopR     variable
```

exist only for  $op=SUB$  and  $op=DIV$ . These instructions are just like  $Fop$  and  $FIop$  except that they perform the operation in reverse order. For example, the instruction "FSUB FOO" would subtract the variable FOO from  $ST(0)$  and store the result at  $ST(0)$ . On the other hand "FSUBR FOO" would subtract  $ST(0)$  from FOO and store the result at  $ST(0)$ .

The instructions will probably be easier to understand in the context of a program. Let us consider the problem of evaluating a polynomial  $u(z)$  of degree  $N$ . A polynomial function, of course, is a function of the form

$$u(z) = u_0 + u_1 z + \dots + u_N z^N .$$

We will assume that  $z$  is a single precision real, and that the coefficients  $u_k$  are contained in an array  $u$  of single precision reals, with  $u_0$  coming first and  $u_N$  coming last. [Note, by the way, that choosing to work in single precision is merely a matter of convenience. Since all calculations are performed in TEMPORARY REAL format, there is essentially no penalty in terms of running time for using double precision or TEMPORARY REAL types for these quantities. Of course, it takes more memory to store higher precision variables than to store lower precision ones.] For example, to evaluate

$$r = u(z) = 5 + 4 z + 3 z^2 + 2 z^3 + z^4$$

at  $z=10.0$  we might have the data

```
N    EQU    4                ; DEGREE OF POLYNOMIAL IS 5.
U    DD    5.0, 4.0, 3.0, 2.0, 1.0 ; COEFFICIENTS OF THE POLYNOMIAL.
Z    DD    10.0              ; VALUE OF THE VARIABLE.
R    DD    ?                 ; THE ANSWER.
```

A reasonably efficient way of evaluating such polynomials is with *Horner's rule*

$$r = u(z) = 5 + z*(4 + z*(3 + z*(2 + z*(1))))$$

which is especially well suited for us (with the 8087) since it allows us to keep both the intermediate results of the calculation and the value of  $z$  in 8087 registers. Here is a simple 8087 program for computing the value of the polynomial using Horner's rule:

```
; INITIALIZE TO USE HORNER'S RULE.
    FLD  Z                ; GET Z INTO THE 8087.
    MOV  SI,4*N          ; USE SI TO INDEX THE COEFFICIENT ARRAY, AND
                        ; START WITH U[N] .
    FLD  U[SI]           ; GET U[N] .
    MOV  CX,N            ; LOOP N TIMES.
; EACH TIME THROUGH THE LOOP, MULTIPLY BY Z AND ADD COEFFICIENT.
; THROUGHOUT THE LOOP, ST(1) IS Z AND ST(0) IS THE RUNNING TOTAL.
AGAIN:
    FMUL ST,ST(1)        ; MULTIPLY RUNNING TOTAL BY Z.
    SUB  SI,4            ; MOVE TO NEXT COEFFICIENT.
    FADD U[SI]           ; ADD THE COEFFICIENT TO THE RUNNING SUM.
    LOOP AGAIN
; SAVE THE RESULT:
    FSTP R               ; SAVE AND POP.
    FSTP ST(0)          ; ALSO, POP Z FROM THE 8087.
```

This program, while not the absolute best program that could be written, is efficient for the 8087. This means that it has the following two properties:

- 1) Transfer of data between the 8087 and memory is kept to a minimum. [In this particular example, each data item is loaded into the 8087 just once, and the only thing stored to memory is the final result.]
- 2) Since the 8088 instructions can execute simultaneously with the preceding 8087 instruction, they should be interspersed with 8087 instructions in such a way that they require the minimum execution time. [In our example, the 8088 instructions take zero execution time.]

The program also has the nice property that the final instructions are arranged in such a way that no FWAIT is needed to ensure that the result is actually in memory before any successive 8088 instructions try to use it.

[Any of you who are fond of a particular high-level language might be interested in benchmarking this program against a high-level equivalent. The average execution time of our assembler program is

$$216 + 211*N \quad \text{clock cycles}$$

for comparison. Thus, for a tenth degree polynomial, our routine would take about 2326 cycles, or roughly 500 microseconds.]

To take a slightly more intricate example, let us consider the case in which the variable  $z$  is complex, but in which the  $u$ 's are still real. Recall that a complex number is a number of the form

$$a + b i$$

where  $a$  and  $b$  are real, but where  $i$  is the square-root of  $-1$ . In normal arithmetic,  $-1$  has no square root, so the number  $i$  is sometimes called an "imaginary" number. For our problem, we will represent  $Z$  by  $X+iY$ . Arithmetic with complex numbers obeys the following rules:

$$\begin{aligned}(a + b i) + (c + d i) &= (a + c) + (b + d) i \\(a + b i) * (c + d i) &= (a*c - b*d) + (a*d + b*c) i\end{aligned}$$

Thus, a complex addition corresponds to two real additions, and a complex multiplication corresponds to 4 real multiplications and two real additions. One high-level language, FORTRAN, has a built-in COMPLEX data type, which both explains and is explained by FORTRAN's overwhelming use in some technical fields (such as physics, my own field). In most other languages, we have to "fake" complex operations by means of real operations. For example, we might define a complex number to be an array of 2 reals, and we might write procedures to perform various operations of complex arithmetic (like the  $+$  and  $*$  operations mentioned above).

In FORTRAN, we would do little more to convert a polynomial-evaluation procedure to allow complex arguments than to change the data types of  $U$ ,  $Z$ , and  $R$  from REAL to COMPLEX. In other languages, we would probably "fake it" by replacing the multiplication step and addition step with the kind of expressions given above. Thus, the entire polynomial evaluation would take  $N$  complex additions and  $N$  complex multiplications, which corresponds to  $4N$  real multiplications and  $4N$  real additions. So long as we are going out of our way to program the problem in assembler, however, we might as well spend a little extra effort to find out if this is really the best algorithm for the job.

In vol. 2 of Knuth, Seminumerical Algorithms, section 4.6.4, a somewhat better algorithm is given. In pseudo-code, this algorithm can be expressed as follows:

```
T := 2X
S := X2 + Y2
A := UN
B := UN-1
FOR I := 2 TO N DO
  BEGIN
    C := B + T * A
    B := UN-I - S * A
    A := C
  END
REAL PART OF R := X * A + B
IMAGINARY PART OF R := Y * A
```

Once again, this algorithm is pretty good for the 8087 since we can keep  $X$ ,  $Y$ ,  $T$ ,  $S$ ,  $A$ ,  $B$ , and  $C$  in the 8087 registers. However, the programming is somewhat trickier than in the previous example since the

addresses of these various quantities on the stack aren't the same all the time. Here is an 8087 program embodying this algorithm:

```

; Note that a complex number is stored as two real numbers in
; consecutive locations.
N EQU 4 ; POLYNOMIAL OF DEGREE 4.
U DD 5.0, 4.0, 3.0, 2.0, 1.0 ; THE COEFFICIENTS.
Z DD 10.0, 1.0 ; LET Z=10+i.
R DD ?, ? ; THE (COMPLEX) RESULT.

; ST0 ST1 ST2 ST3 ST4 ST5 ST6 ST7
FLD Z ; X
FLD ST(0) ; X X
FADD ST,ST(1) ; T X
FLD Z+4 ; Y T X
FLD ST(0) ; Y Y T X
FMUL ST,ST(1) ; Y*Y Y T X
FLD ST(3) ; X Y*Y Y T X
FMUL ST,ST(4) ; X*X Y*Y Y T X
FADDP ST(1),ST ; S Y T X
MOV SI,4*N
FLD U[SI-4] ; B S Y T X
FLD U[SI] ; A B S Y T X
SUB SI,8
MOV CX,N-1
AGAIN:
FLD ST(4) ; T A B S Y T X
FMUL ST,ST(1) ; A*T A B S Y T X
FADDP ST(2),ST ; A B+A*T S Y T X
FMUL ST,ST(2) ; A*S B+A*T S Y T X
FSUBR U[SI] ; NEW_B NEW_A S Y T X
SUB SI,4
FXCH ST(1) ; NEW_A NEW_B S Y T X
LOOP AGAIN
FMUL ST(5),ST ; A B S Y T A*X
FMULP ST(3),ST ; B S A*Y T A*X
FADDP ST(4),ST ; S A*Y T B+A*X
FSTP ST(0) ; A*Y T B+A*X
FSTP R+4 ; T B+A*X
FSTP ST(0) ; B+A*X
FSTP R ; empty stack

```

As usual, this code is reasonably straightforward, at least in the sense of embodying the algorithm. We do see, however, one new 8087 instruction -- namely

```
FXCH ST(k)
```

which exchanges ST(0) and ST(k). We also see that an 8087 program can be greatly clarified by appending as comments a "snapshot" of the stack at every step.

[On the average, this code executes in a time

$$1219 + 512*(N-1) \text{ clock cycles.}$$

In an actual test of this code against Microsoft FORTRAN (which used a complexification of Horner's rule rather than this algorithm), I found that the FORTRAN code for computing a 19-th degree polynomial (10000 iterations) executed in 191.1 seconds (with a .EXE file 31562 bytes long), while the assembly version took 26.7 seconds (with a 1338 byte .EXE file). The assembly version was therefore 8 times as fast. However, if (in fairness) we take into account the fact that our algorithm used only half as many floating point operations, we must conclude that the assembly version was really only 4 times as fast as FORTRAN. It is interesting to note that the average execution time of our assembly code above in computing a 19-th degree polynomial is 2.3 milliseconds -- not that much different than the time required for *software* to compute *one* floating-point addition or multiplication. ]

#### The 8087 Status Word

With the 8088, it is often possible for instructions to generate various error conditions (or other types of conditions), on the basis of which the program must take some kind of action. To handle this we have various "flags" in the CPU's "flag register", namely ZF, SF, CF, PF, OF, and HF. Conditional jump instructions can test the values of these flags.

With the 8087, the situation is less convenient. Since the 8087 cannot access the 8088 registers directly, there is no direct reflection (in the 8088 flag register) of conditions in the 8087. Instead, we must perform some additional operations to handle such information. The 8087 has its own "status word" (similar to the 8088's flag word), but we have to go out of our way to get this word stored into memory where it can be examined by the 8088.

The 8087 status word contains somewhat more information than the 8088 flag word. For our purposes, we will limit ourselves to a consideration of two kinds of information in the status word. Six of the bits in the status word describe *exceptions* which have occurred in the calculation, and four of the remaining bits describe the kind of number which the calculation has produced. Let us first discuss the exceptions.

An 8087 "exception" is an indication that an error has occurred in the calculation. The six possible exceptions are:

| <u>BIT</u> | <u>FLAG</u> | <u>DESCRIPTION</u>  |
|------------|-------------|---|
| 0          | IE          | invalid operand exception. This exception can occur in many ways and usually indicates an invalid operand. For example, if an invalid exception would occur in taking the square root of a negative number. |
| 1          | DE          | denormalized operand exception. We will discuss "denormal" numbers in a moment.   |
| 2          | ZE          | zero-divide exception is caused by a division by zero.  |
| 3          | OE          | overflow exception happens when the result of an operation is greater than the largest number allowed by the TEMPORARY REAL format -- i.e., greater than 1.0E+4932 or less than - 1.0E4932.                 |
| 4          | UE          | underflow exception happens when the result of an operation is smaller in magnitude than the smallest non-zero TEMPORARY REAL number -- i.e., less than 1.0E-4932.  |
| 5          | PE          | precision exception. Occurs when precision is lost in the operation. For example, rounding the number 1.234567E-54 to an integer is bound to lose all significant figures of the operand.                   |

These flags represent (respectively) bits 0-5 of the 8087 status word. When these bits are zero, no exception has occurred. When they are one, an exception *has* occurred.

The exception flags are referred to in the Intel literature as "sticky bits". They are "sticky" in the sense that once they are turned on they remain on. This is convenient because it means that rather than continually checking the exception flags after every 8087 operation, we can simply wait until the end of a sequence of calculations and then check the flags. In order to check the exception flags (and the rest of the status word as well), we must use the FSTSW instruction to save the status word in memory, where it can be examined by the 8088. Here is an example showing how to do that (simultaneously checking for overflow):

```
status_word dw ?
.
.
.
FSTSW     status_word     ; get the 8087 status word.
FWAIT
TEST      status_word,8   ; mask in just the OE bit.
JNZ       overflow       ; if set, go to overflow handler.
FCLEX
```

The final instruction in this little program is the 8087 instruction to "clear" the exception flags. In general, since the exception bits are sticky, this is the only way to clear them short of resetting the 8087. Whether or not the exception flags are explicitly checked by the program, the 8087 internally performs an error-correction routine to try and recover from the error condition. The error-correction

routines are known as *masked responses*, and in order to understand them we have to understand some more about how the 8087 represents numbers.

First, we need to understand about the condition codes. The condition codes are the other four bits in the status word that we mentioned earlier. The condition codes serve two purposes. First, they are set as the result of comparison operations (which we have not yet discussed), so we must test them to determine (say) if one number is less than another. Second, they are used to give the *type* of number resulting from an operation. By "type" I do not mean "data type" in the sense of being a WORD INTEGER, SHORT INTEGER, ..., TEMPORARY REAL. Since all 8087 data is internally represented as TEMPORARY REAL, this would be useless. Rather, the 8087 has ways of representing several types of "unusual" numbers. Here are the special types of numbers recognized by the 8087 (the numbers at the beginnings of the rows are explained in a minute):

|     |            |  |
|-----|------------|--|
| 0   | UNNORMAL   | See denormal.  |
| 1   | NAN        | Stands for "Not A Number". A NAN is a bit pattern which, though of the proper length, does not represent any valid number. That is, for the 8087, not all bit patterns are used as numbers. NANs are useful since they can sometimes be used to detect uninitialized variables.  |
| 2   | NORMAL     | Any valid TEMPORARY REAL number other than 0.  |
| 3   | INFINITY   | The 8087 has a representation for infinity, and can perform arithmetic on it.  |
| 4   | ZERO       | (Does this need explanation?)  |
| 5,7 | EMPTY      | This refers to a number in an empty register -- i.e., a register which has not been FLDED.   |
| 6   | DENORMAL   | The 8087 does not <i>suddenly</i> underflow when numbers become too small -- rather, it <i>gradually</i> underflows. Normally, representations of floating point numbers demand "normalization". That is, that the leading bit of the number be one. The 8087, however, allows the leading bits of its numbers to be zero. Thus, at the cost of some lost precision, it can represent numbers smaller than 1.0E-4932. Such an unnormalized number is called <i>denormal</i> . A denormal number is called <i>unnormal</i> after it has been used in a calculation. |
|     | INDEFINITE | Represents a "don't know" answer. In each data type, a special value is used to represent an indefinite answer. For the TEMPORARY REAL format, the indefinite number is a special NAN.   |

With these special types of numbers in hand, we can now understand how the 8087 processes (i.e., recovers from) exceptions. While we cannot go into every possible case here, here is a rule-of-thumb guide to the typical responses for the indicated exceptions:

IE The 8087 returns an indefinite number.

DE No response. This is not an error as such.

ZE Return a properly signed infinity.

OE Return a properly signed infinity.

UE Denormalize (unless this results in zero, in which case return a zero).

PE No special action.

We can explicitly check the type of number returned by an operation by examining the condition code bits of the 8087 status word. Bits 8-10 of the status word are the condition code bits  $C_0$ - $C_2$ , while bit 14 is condition code bit  $C_3$ .  $C_1$  represents the sign of the number. If  $C_1=0$ , then the number is positive; if  $=1$ , then the number is negative.  $C_0$ ,  $C_2$ ,  $C_3$  together form a 3-bit number which can be interpreted according to the table of special number types above -- i.e., 0=unnormal, 1=NAN, etc. Normally, however, prior to checking these bits we must use the FXAM instruction to ensure that the bits are set properly. FXAM "examines" ST(0) to determine its type.

To illustrate these ideas, here is a sample program (from the Intel literature) which uses the jump-table technique to test the condition codes and determine the type of the number:

```
JTABLE DW UNNORM,NAN,NORM,INFINITY,ZERO,EMPTY,DENORM,EMPTY
STATUS DW ?

        FXAM                ; SET CONDITION CODE BITS PROPERLY.
        FSTSW STATUS        ; GET THE STATUS WORD.
        FWAIT               ; WAIT FOR IT.
; MUST DO A LOT OF SHIFTS TO GET ALL OF THE BITS IN THE PROPER
; POSITION (i.e, TO GET C0, C2, AND C3 INTO BITS 0, 1, AND 2).
        MOV AL,BYTE PTR STATUS+1    ; GET HIGH BYTE OF STATUS.
        MOV BL,AL                ; SAVE C0
        AND BL,1                  ; IN BL.
        SHR AL,1                  ; DIVIDE BY 2 TO GET C1.
        MOV BH,AL                ; SAVE C1
        AND BH,2                  ; IN BH.
        SHR AL,1                  ; DIVIDE BY 8 TO
        SHR AL,1                  ; GET C2 INTO
        SHR AL,1                  ; PROPER POSITION.
        AND AL,4                  ; MASK OUT EVERYTHING ELSE THAN C2.
        OR AL,BH                  ; GET BACK BH.
        OR AL,BL                  ; GET BACK BL.
; NOW HAVE 0-7 IN AL.  NEED 0-14 IN SI:
        MOV AH,0
        MOV SI,AX
        ADD SI,AX
        JMP JTABLE[SI]

;
UNNORM:
NAN:
```

NORMAL:  
 INFINITY:  
 ZERO:  
 EMPTY:  
 DENORM:

While this does not finish the discussion of the condition code -- in particular, we have not discussed the *comparison* instructions (FCOM, FCOMP, FCOMPP, FICOM, FTST), which are analogous to the 8088 CMP instruction and also set the condition code -- we will not consider the condition code further at this point. We will discuss comparison instructions in a later lecture.

### Some Very Simple 8087 Instructions

In the remainder of this lecture, I would simply like to tell you in capsule form about some additional 8087 instructions that are both useful and even easier to use than the instructions discussed so far. None of the instructions to be discussed has any operands:

The FABS instruction is used to take absolute value of the top of the stack: ST <-- |ABS(ST)|.

The FCHS instruction multiplies the top of stack by -1:  
 ST <-- -ST.

The FNOP instruction performs no operation. It simply kills time. There is a similar 8088 instruction NOP, which we have not had occasion to use up to this point.

The FRNDINT instruction rounds the top of stack to an integer. ST <-- ROUND(ST). (This is not to say that the *data type* of the top of stack is changed to INTEGER -- the data type of a register is always TEMPORARY REAL.)

The FSQRT instruction replaces the top of stack by its square root. This instruction is noteworthy in that it is probably the fastest 8087 instruction in relation to the complexity of the function performed. Here is a benchmark given by Starz in 8087: Applications and Programming for the IBM PC and Other PCs:

| <u>PROGRAM/MACHINE</u>        | <u>TIME (SECONDS) FOR 5000<br/>SQUARE ROOTS</u> |
|-------------------------------|---|
| 8087 routine                  | 0.35  |
| Apple II+ BASIC (interpreter) | 130.0   |
| DEC 2060 BASIC (compiler)     | 0.40  |
| VAX 780 FORTRAN (compiler)    | 0.20  |
| IBM 3081 BASIC (interpreter)  | 0.26  |

Not one of these comparisons is really fair. For one thing, it is unfair to compare a language interpreter against a compiler, since interpreters cannot help being slow. Also, it is slightly unfair to compare even a compiled program against an assembler program since, as we have seen, an assembly language programmer can easily beat the fastest compiler when it comes to 8087 code. (On the other hand, the ease of writing such an 8087 program speaks in favor of such a comparison). The most interesting (and fairest) comparison is probably

to VAX FORTRAN. As far as square roots are concerned, it seems to imply that the 8087 is about half of a VAX (and at somewhat less than half the cost).

Several 8087 instructions are used to FLD (i.e., PUSH) predefined constants onto the 8087 register stack (below, "e" designates the base of natural logarithms, 2.7182818...):

|        |   |
|--------|---|
| FLDLG2 | push the logarithm (base 10) of 2 onto the stack. |
| FLDLN2 | push the logarithm (base e) of 2 onto the stack.  |
| FLDL2E | push the logarithm (base 2) of e onto the stack.  |
| FLDL2T | push the logarithm (base 2) of 10 onto the stack. |
| FLDPI  | push pi (3.14159265...) onto the stack.           |
| FLDZ   | push zero onto the stack.                         |
| FLD1   | push one onto the stack.                          |

As a simple example using some of the concepts we have discussed, let us compute the roots,  $x$ , of the quadratic equation

$$0 = x^2 + a x + b$$

where  $a$  and  $b$  are real numbers. The roots of the equation are given by the formulas

$$x_1 = -(a/2) + [ (a/2)^2 - b ]^{1/2}$$

$$x_2 = \quad " \quad - \quad "$$

Normally, special programming techniques are needed to get the required accuracy out of these formulas in various cases. We will (for brevity) ignore this fact and simply compute a straightforward solution -- *with* checks of the exceptions, however:

```

A   DD   ?           ; DON'T CARE ABOUT THE VALUES OF THE COEFFICIENTS.
B   DD   ?
X1  DD   ?           ; PLACE FOR THE RESULTS.
X2  DD   ?
STATUS DW ?         ; PLACE FOR THE 8087 STATUS WORD.

      FCLEX          ; GET RID OF ANY PREVIOUS EXCEPTION FLAGS.
      ;             ST0          ST1          ST2          ST3          ST4
      FLD  A          ;             A
      FLD1          ;             1          A
      FADD ST,ST(0)  ;             2          A
      FDIVP ST(1),ST ;             A/2
      FCHS          ;             -A/2
      FLD ST(0)      ;             -A/2          -A/2
      FMUL ST,ST(1)  ;             (A/2)2          -A/2
      FSUB B          ;             (A/2)2-B          -A/2
      FSQRT          ;             SQRT          -A/2
      FLD ST(0)      ;             SQRT          SQRT          -A/2
      FADD ST,ST(2)  ;             X1          SQRT          -A/2
      FSTP X1        ;             SQRT          -A/2
      FSUBP ST(1),ST ;             X2
      FSTP X2        ;
      FSTSW STATUS   ; GET STATUS WORD INTO MEMORY.
; THE ERRORS WE HAVE TO LOOK FOR ARE:
; IE  INVALID -- THIS OCCURS IF ROOTS ARE COMPLEX.
; OE  OVERFLOW -- THIS OCCURS IF A WAS TOO BIG.
; PE  PRECISION -- THIS OCCURS IF ROOTS ARE CLOSE TO ZERO OR A/2.
; OF THE OTHER EXCEPTIONS, ZE (ZERODIVIDE) CANNOT OCCUR AND DE
; (DENORMAL) AND UE (UNDERFLOW) ARE NOT OF INTEREST TO US.
      FWAIT
      TEST STATUS,1  ; IE?
      JNZ  INVALID
      TEST STATUS,8  ; OE?
      JNZ  OVERFLOW
      TEST STATUS,32 ; PE?
      JNZ  PRECISION

```

OK:

This concludes the lectures that will be spent exclusively discussing the 8087. However, I will continue to talk about the 8087 periodically throughout the remainder of the semester. You might be interested to know that of the 107 instructions and variations on "8087 reference sheet 2" (which covers all of the 8087 instructions), we have so far discussed 70. 9 of the remaining instructions are comparison instructions and will be discussed in the next lecture.

8087 NUMERIC COPROCESSOR REFERENCE SHEET 2

|         |                                | TERMS USED |                               |     |                    |
|---------|--------------------------------|------------|-------------------------------|-----|--------------------|
| ST(k)   | k-th element of 8087 stack     | long       | LONG INTEGER variable         | log | logarithm, base 10 |
| real    | SHORT or LONG REAL variable    | packed     | PACKED DECIMAL variable       | ln  | logarithm, base e  |
| integer | WORD or SHORT INTEGER variable | {pop}      | ST is popped after operation  | lg  | logarithm, base 2  |
| temp    | TEMPORARY REAL variable        | {nowait}   | use no FWAIT before operation |     |                    |

|   |    |              |   |    |           | STATUS WORD BITS |    |                |     |    |                   |
|---|----|--------------|---|----|-----------|------------------|----|----------------|-----|----|-------------------|
| 0 | IE | invalid      | 3 | OE | overflow  | 8                | C0 | condition code | B-D | ST | stack top pointer |
| 1 | DE | denormalized | 4 | UE | underflow | 9                | C1 | condition code | E   | C3 | condition code    |
| 2 | ZE | zero-divide  | 5 | PE | precision | A                | C2 | condition code | F   | B  | busy              |

INTERPRETATION OF THE CONDITION CODE BITS

C3, C2, and C0 together make a 3-bit number (with C3 the most significant bit). Interpretation of the 3-bit code:

After a comparison (FCOM, FCOMP, FCOMPP, FICOM, FICOMP, FTST):

0= ST>source (ST>0 for FTST), 1= ST<source (ST<0 for FTST), 4= ST=source (ST=0 for FTST), 7= don't know.

After FXAM: 0=unnormal, 1=not-a-number(NAN), 2=normal, 3=infinity, 4=zero, 5=empty, 6=denormal, 7=empty.

[C1 is the sign of the number (0=positive, 1=negative).]

The instructions with comments beginning in "-" rather than ";" require special knowledge to be used effectively. The execution times listed are in clock cycles. The times are typical rather than guaranteed. When memory variables are used, the effective address calculation time (EA) must also be added. The times shown are for WORD INTEGER or SHORT REAL variables. For SHORT INTEGER or LONG REAL, add about 5 clock cycles.

| INSTRUCTION | PURPOSE                         | TIME | EXCEPTIONS | INSTRUCTION | PURPOSE                                | TIME | EXCEPTIONS |
|-------------|---------------------------------|------|------------|-------------|--|------|------------|
| FABS        | ; ST= ST                        | 14   | I          | FLDL2T      | ; PUSH lg(10)                          | 19   | I          |
| FADD        | ; FADDP ST(1),ST                | 85   | IDOUP      | FLDPI       | ; PUSH pi                              | 19   | I          |
| FADD        | ST,ST(k) ; ST=ST+ST(k)          | 85   | IDOUP      | FLDZ        | ; PUSH zero                            | 14   | I          |
| FADD        | ST(k),ST ; ST(k)=ST+ST(k)       | 85   | IDOUP      | FLD1        | ; PUSH one                             | 18   | I          |
| FADD        | real ; ST=ST+real               | 105  | IDOUP      | FMUL        | ; FMULP ST(1),ST                       | 138  | IDOUP      |
| FADDP       | ST(k),ST ; ST(k)=ST+ST(k) {pop} | 90   | IDOUP      | FMUL        | ST,ST(k) ; ST=ST*ST(k)                 | 138  | IDOUP      |
| FBLD        | packed ; PUSH packed            | 300  | I          | FMUL        | ST(k),ST ; ST(k)=ST(k)*ST              | 138  | IDOUP      |
| FBSTP       | packed ; POP packed             | 530  | I          | FMUL        | real ; ST=ST*real                      | 118  | IDOUP      |
| FCHS        | ; ST=-ST                        | 15   | I          | FMULP       | ST(k),ST ; ST(k)=ST(k)*ST {pop}        | 142  | IDOUP      |
| FCLEX       | ; clear exceptions              | 5    | none       | FNCLEX      | ; {nowait} clear exceptions            | 5    | none       |
| FCOM        | ; CMP ST,ST(1)                  | 45   | ID         | FNDISI      | - {nowait} disable interrupts          | 5    | none       |
| FCOM        | ST(k) ; CMP ST,ST(k)            | 45   | ID         | FNENI       | - {nowait} enable interrupts           | 5    | none       |
| FCOM        | real ; CMP ST,real              | 65   | ID         | FNINIT      | ; {nowait} initialize 8087             | 5    | none       |
| FCOMP       | ; CMP ST,ST(1) {pop}            | 47   | ID         | FNOP        | ; no operation                         | 13   | none       |
| FCOMP       | ST(k) ; CMP ST,ST(k) {pop}      | 47   | ID         | FNSAVE      | 94 bytes - {nowait} save all registers | 208  | none       |
| FCOMP       | real ; CMP ST,real {pop}        | 68   | ID         | FNSTCW      | 2 bytes - {nowait} save control word   | 15   | none       |
| FCOMPP      | ; CMP ST,ST(1) {pop} {pop}      | 50   | ID         | FNSTENV     | 14 bytes - {nowait} save environment   | 45   | none       |
| FDECSTP     | ; move stack down by 1          | 9    | none       | FNSTSW      | 2 bytes ; {nowait} save status word    | 15   | none       |
| FDISI       | - disable interrupts            | 5    | none       | FPATAN      | - ST="partial" ARCTAN(ST)              | 650  | UP         |
| FDIV        | ; FDIVP ST(1),ST                | 198  | IDZOUN     | FPREM       | - ST=ST MOD ST(1)                      | 125  | IDU        |

|         |                                 |     |        |          |                                  |     |       |
|---------|---------------------------------|-----|--------|----------|----------------------------------|-----|-------|
| FDIV    | ST,ST(k) ; ST=ST/ST(k)          | 198 | IDZOUN | FPTAN    | - ST="partial" TAN(ST)           | 450 | IP    |
| FDIV    | ST(k),ST ; ST(k)=ST(k)/ST       | 198 | IDZOUN | FRNDINT  | ; ST=ROUND(ST)                   | 45  | IP    |
| FDIV    | real ; ST=ST/real               | 220 | IDZOUN | FRSTOR   | 94 bytes - restore all registers | 208 | none  |
| FDIVP   | ST(k),ST ; ST(k)=ST(k)/ST {pop} | 202 | IDZOUN | FSAVE    | 94 bytes - save all registers    | 208 | none  |
| FDIVR   | ; FDIVRP ST(1),ST               | 199 | IDZOUN | FSCALE   | - ST=ST SHL ST(1)                | 35  | IOU   |
| FDIVR   | ST,ST(k) ; ST=ST(k)/ST          | 199 | IDZOUN | FSQRT    | ; ST=SQRT(ST)                    | 183 | IDP   |
| FDIVR   | ST(k),ST ; ST(k)=ST/ST(k)       | 199 | IDZOUN | FST      | ST(k) ; ST(k)=ST                 | 18  | IOUP  |
| FDIVR   | real ; ST=real/ST               | 221 | IDZOUN | FST      | real ; real=ST                   | 87  | IOUP  |
| FDIVRP  | ST(k),ST ; ST(k)=ST/ST(k) {pop} | 203 | IDZOUN | FSTCW    | 2 bytes - save control word      | 15  | none  |
| FENI    | - enable interrupts             | 5   | none   | FSTENV   | 14 bytes - save environment      | 45  | none  |
| FFREE   | ST(k) ; tag ST(k) as empty      | 11  | none   | FSTP     | ST(k) ; ST(k)=ST {pop}           | 20  | IOUP  |
| FIADD   | integer ; ST=ST+integer         | 120 | IDOP   | FSTP     | real ; real=ST {pop}             | 89  | IOUP  |
| FICOM   | integer ; CMP ST, integer       | 80  | ID     | FSTP     | temp ; temp=ST {pop}             | 55  | IOUP  |
| FICOMP  | integer ; CMP ST, integer {pop} | 82  | ID     | FSTSW    | 2 bytes ; save status word       | 15  | none  |
| FIDIV   | integer ; ST=ST/integer         | 230 | IDZOUN | FSUB     | ST,ST(k) ; FSUBP ST(1),ST        | 85  | IDOUP |
| FIDIVR  | integer ; ST=integer/ST         | 230 | IDZOUN | FSUB     | ST(k),ST ; ST(k)=ST(k)-ST        | 85  | IDOUP |
| FILD    | integer ; PUSH integer          | 50  | I      | FSUB     | real ; ST=ST-real                | 105 | IDOUP |
| FILD    | long ; PUSH long                | 64  | I      | FSUBP    | ST(k),ST ; ST(k)=ST(k)-ST {pop}  | 90  | IDOUP |
| FIMUL   | integer ; ST=ST*integer         | 130 | IDOP   | FSUBR    | ; FSUBRP ST(1),ST                | 87  | IDOUP |
| FINCSTP | ; move stack up by 1            | 9   | none   | FSUBR    | ST,ST(k) ; ST=ST(k)-ST           | 87  | IDOUP |
| FINIT   | ; initialize 8087               | 5   | none   | FSUBR    | ST(k),ST ; ST(k)=ST-ST(k)        | 87  | IDOUP |
| FIST    | integer ; integer=ST            | 86  | IP     | FSUBR    | real ; ST=real-ST                | 105 | IDOUP |
| FISTP   | integer ; POP integer           | 88  | IP     | FSUBRP   | ST(k),ST ; ST(k)=ST-ST(k) {pop}  | 90  | IDOUP |
| FISTP   | long ; POP long                 | 100 | IP     | FTST     | ; CMP ST,0                       | 42  | ID    |
| FISUB   | integer ; ST=ST-integer         | 120 | IDOP   | FWAIT    | ; wait for 8087                  | >3  | none  |
| FISUBR  | integer ; ST=integer-ST         | 120 | IDOP   | FXAM     | ; report nature of ST            | 17  | none  |
| FLD     | ST(k) ; PUSH ST(k)              | 20  | ID     | FXCH     | ; exchange ST and ST(1)          | 12  | I     |
| FLD     | real ; PUSH real                | 43  | ID     | FXCH     | ST(k) ; exchange ST and ST(k)    | 12  | I     |
| FLD     | temp ; PUSH temp                | 57  | ID     | FXTRACT  | - separate exponent and          | 50  | I     |
| FLDCW   | 2 bytes - load control word     | 10  | none   |          | - significand of ST              |     |       |
| FLDENV  | 14 bytes - load environment     | 40  | none   | FYXL2X   | - ST(1)=ST(1)*lg(ST) {pop}       | 950 | P     |
| FLDLG2  | ; PUSH log(2)                   | 21  | I      | FYXL2XP1 | - ST(1)=ST(1)*lg(ST+1) {pop}     | 850 | P     |
| FLDLN2  | ; PUSH ln(2)                    | 20  | I      | F2XM1    | - ST=(2 to the ST)-1             | 500 | UP    |
| FLDL2E  | ; PUSH lg(e)                    | 18  | I      |          |                                  |     |       |

FINAL PROJECT #1: *SOME DATABASE FUNCTIONS*

This project is, in a sense, a continuation of the mid-term project. The mid-term project *sorted* a text file. This project allows us to quickly insert, delete, or locate elements in the sorted file. This project actually consists of writing several short programs. Nobody has to write *all* of the programs. Anyone doing this project must write at least three of the programs, however. Here are the required programs:

1. INDEX.ASM -- *index* a text file. (Anybody familiar with DBASE II will have seen a similar function before.) The INDEX program takes as input a text file, which may (or may not) have been previously sorted. This text file is assumed to be larger than the available memory of the computer. As output, INDEX sets up a file of *pointers* to the individual lines of the text file. This is very similar to the way the sorting program worked, except that it is not necessary to maintain the length of each text line as a separate count. (Thus, the given text file is unmodified by this program.) Since the text file can be longer than 64K, the pointers to the lines must be *doublewords* rather than words. The output file thus consists of 4-byte records, each of which represents a doubleword pointer into the text file. The only exception will be the first 4-byte record, which will contain a doubleword count of the *number* of text-lines. Suppose that our text file was

```
This is a sample text file<cr><lf>
which is to be used with the indexing function <cr><lf>
for the final project.<cr><lf>
```

Since there are three lines of text, the output index file's first record would be a doubleword with the value 3, indicating that there are three text lines. This would be followed by three records containing the doublewords 0, 28, and 77, which are the relative positions of the three lines in the file. That is, the output index-file would contain the bytes (in hex)

```
03 00 00 00    00 00 00 00    1C 00 00 00    4D 00 00 00
```

(Recall that the bytes of an integer number are stored least-significant-byte to most-significant-byte.) As a convention, we will suppose that the pointer being negative -- i.e., bit 31 being one -- marks the line as "deleted" from the textfile. This convention is used by the programs that follow. Of course, this fact does not affect the operation of the INDEX program since by definition all lines in the original text file are "present" and not "deleted". The index file is useful in dealing with a large (sorted) file, since it allows us to add or delete records to the file much more quickly than by simply rearranging the file (at least, if the records average much more than 4 bytes in length). The syntax of the INDEX program will be

```
INDEX textfile indexfile
```

and you should not assume that the entire array of doubleword pointers can fit into memory at once. Notice that neither the input nor output files are given by I/O redirection, and the PSP must be explicitly read to determine the names of the files.

2. DSPLINES.ASM -- *display* a range of text-lines, using the pointers in an index file. Here is the syntax of the DSPLINES program:

```
DSPLINES textfile indexfile [/Ln:m] [/D]
```

Here, neither the index file nor the text file is given by I/O redirection. (If desired, the output can be redirected to a file rather than to the console.) The name of the text file is passed in the PSP, and you must check the parameter buffer for the name, and explicitly open the file. The "/Ln:m" gives the range of lines to display. Thus, if we used the switch /2:3, the program should get the second line pointer from the index file, go to the indicated position in the text file, and display the line; then it should do the same thing for the third pointer in the index file. Because of our convention of "deleting" lines

by setting the high bit of the pointer, the high bit must be zeroed before doing this. We assume that the lines are "numbered" 1, 2, 3, etc., with the zeroeth position in the index file being occupied by the line count. In order to move to the proper position in the text file, the DOS seek function is used. You should also allow the following defaults for the switches -- "/Ln:" means to list from line n to the end of file; "/L:m" means to list from line 1 to line m; "/L" or "/L:" (or the switch being omitted altogether) means to list the entire file. If n:m expresses a range of lines greater than that contained in the file, then just those lines in the file should be displayed. If the high bit (bit 31) of the pointer to a line is set, then instead of displaying the line (which has, by convention, been deleted from the file), the message "DELETED LINE" should be displayed. However, if the /D switch is present, even the deleted lines should be displayed (rather than the "DELETED LINE" message).

3. BINSERCH.ASM -- perform a binary search of the text file (assuming that the order of the records in the index file reflects a completely sorted file). The binary search is discussed in Chapter 5 of the text, and it should be reasonably clear how to extend it to the case we are considering. The "table" being searched here is essentially the index file (containing the pointers) rather than the table of integers as in the book. Note that before any pointer in the index file can be used, its high bit (bit 31) must be cleared. The syntax for BINSERCH is

```
BINSERCH textfile indexfile <stringfile
```

Here, the "string file" is a file containing strings to be searched for. If the I/O redirection is omitted, these will simply be typed in at the keyboard. In either case, neither the text file nor the index file is given by redirection, and each must be explicitly gotten from the parameter buffer in the PSP. Output may be redirected, if desired by the user. The output is as follows: for each input string, there will be an output message, with one message per output line. The format of the messages is as follows: if the string is found, the message

```
STRING FOUND AT LINE n
```

is printed. If the string is found but has been deleted (high bit of the pointer is one), the message

```
STRING FOUND AT LINE n, BUT IS DELETED
```

If not, the message,

```
STRING WOULD BE BETWEEN LINES n AND m
```

is printed. Here, "m" is intended to be one greater than "n". For example, searching the file

```
GOODBYE  
HELLO  
LENS  
ZOOM
```

for "ZOOM" would result in "STRING FOUND AT LINE 4". Searching for "HAL" would result in "STRING WOULD BE BETWEEN LINES 1 AND 2", while searching for "GANDER" would give "STRING WOULD BE BETWEEN LINES 0 AND 1" (even though there is no line 0). If the string is found, the line number displayed must be the number of the *first* line in the file which matches the string. (If there are several lines which are the same, a binary search can result in finding any of the lines, not necessarily the first).

4. INSERT.ASM -- insert a line of text into the textfile. Again, this function assumes a sorted file (at least, that the pointers give the proper sorted order). This does not actually insert anything into the text file. Rather, it puts the line at the end of the text file and inserts a pointer into the pointer file (at the proper position) for the line. Here is the syntax of INSERT:

```
INSERT textfile indexfile <stringfile [/Ln]
```

This program actually provides two distinct functions. If the /Ln is not present, it inserts each string in the stringfile into its proper position in the text file by manipulating the pointers as suggested. It must somehow search the textfile to find the proper locations. If the /Ln is present, the entire stringfile is simply inserted into the textfile prior to line n, regardless of whether this would maintain the order.

5. DELETE.ASM -- delete lines from the text file. The textfile is not actually modified. As with INSERT, the index file is manipulated. The syntax of DELETE is

```
DELETE textfile indexfile [<stringfile] [/Ln:m] [/U]
```

If /Ln:m is present (with the defaults and conventions mentioned above), then the indicated lines are deleted. This is done by setting the high bits (bit 31) of the appropriate pointers in the index file. If /Ln:m is not present, the indicated string file is used; the textfile is searched for the lines in the stringfile, and those lines are deleted by the method just mentioned. Any lines not found in the file are, of course, not deleted. If the /U switch is set, the operation is unchanged except that the lines are "undeleted" from the text file. That is, the high bits of the pointers are set to zero rather than to one.

6. CONDENSE.ASM -- condense the text file. This puts the text file into the order indicated by the index file. Syntax:

```
CONDENSE textfile indexfile newtextfile
```

The new text file need not be indexed by this procedure. If re-indexing is desired, INDEX can be run.

7. SEQSERCH.ASM -- sequentially search the text file for the given strings. The syntax and characteristics of this program are similar to BINSERCH, except that a different searching method is used. In this case, however, the file is not assumed to be sorted. The exact syntax is

```
SEQSERCH textfile indexfile <stringfile [/Ln:m]
```

thus we allow a specific range of lines to be searched, if desired.

8. MERGE.ASM -- merge two textfile/indexfile pairs. The syntax is

```
MERGE textfile1 indexfile1 textfile2 indexfile2
```

This operation results in files 2 being unchanged, but files 1 being enlarged to contain all initial files. The initial files are assumed sorted. Textfile 2 is simply appended to the end of textfile 1 by this operation, but indexfile 2 is shuffled into indexfile 1 in such a way that the output file is properly sorted.

SOME HINTS

It is clear that there is a lot of overlapping functionality in these programs. Having the following things available would help a lot:

A data item

PARAM\_LOC DW *next location to be used in PSP parameter buffer (initially 81H)*

Macros

```
NEXT_FILE location      ; Using PARAM_LOC, get next filename from the parameter buffer: i.e., put a zero at the end of
                        ; it and store its location.
GET_RANGE n,m           ; Using PARAM_LOC, search PSP parameter buffer for /Ln:m and save the values. (Also, deal with
                        ; the defaults.)
BIN_SERCH indhandle,txthandle,addressofstring ; Search file for string, returning with Z set if found, and with
                        ; DX:CX containing the line-number (or the preceding line-number, if not found).
```

**EVALUATION OF A REAL POLYNOMIAL  
USING HORNER'S RULE**

A POLYNOMIAL

$$r = u(z) = 5 + 4z + 3z^2 + 2z^3 + z^4$$

$$= 5 + z*(4 + z*(3 + z*(2 + z*(1))))$$

DATA FOR THE PROGRAM

```

N   EQU 4                      ; DEGREE OF POLYNOMIAL IS 5.
U   DD  5.0, 4.0, 3.0, 2.0, 1.0 ; COEFFICIENTS OF THE POLYNOMIAL.
Z   DD  10.0                    ; VALUE OF THE VARIABLE.
R   DD  ?                       ; THE ANSWER.

```

THE PROGRAM

```

; INITIALIZE TO USE HORNER'S RULE.
   FLD  Z                        ; GET Z INTO THE 8087.
   MOV  SI,4*N                  ; USE SI TO INDEX THE COEFFICIENT ARRAY, AND
                               ; START WITH U[N] .
   FLD  U[SI]                   ; GET U[N] .
   MOV  CX,N                    ; LOOP N TIMES.
; EACH TIME THROUGH THE LOOP, MULTIPLY BY Z AND ADD COEFFICIENT.
; THROUGHOUT THE LOOP, ST(1) IS Z AND ST(0) IS THE RUNNING TOTAL.
AGAIN:
   FMUL ST,ST(1)                ; MULTIPLY RUNNING TOTAL BY Z.
   SUB  SI,4                    ; MOVE TO NEXT COEFFICIENT.
   FADD U[SI]                   ; ADD THE COEFFICIENT TO THE RUNNING SUM.
   LOOP AGAIN
; SAVE THE RESULT:
   FSTP R                       ; SAVE AND POP.
   FSTP ST(0)                  ; ALSO, POP Z FROM THE 8087.

```

EVALUATION OF A REAL POLYNOMIAL  
AT A COMPLEX ARGUMENT

THE ALGORITHM

```
T := 2X
S := X2 + Y2
A := UN
B := UN-1
FOR I := 2 TO N DO
BEGIN
  C := B + T * A
  B := UN-I - S * A
  A := C
END
REAL PART OF R := X * A + B
IMAGINARY PART OF R := Y * A
```

THE PROGRAM

```
; Note that a complex number is stored as two real numbers in
; consecutive locations.
N EQU 4 ; POLYNOMIAL OF DEGREE 4.
U DD 5.0, 4.0, 3.0, 2.0, 1.0 ; THE COEFFICIENTS.
Z DD 10.0, 1.0 ; LET Z=10+i.
R DD ?, ? ; THE (COMPLEX) RESULT.

; ST0 ST1 ST2 ST3 ST4 ST5 ST6 ST7
FLD Z ; X
FLD ST(0) ; X X
FADD ST,ST(1) ; T X
FLD Z+4 ; Y T X
FLD ST(0) ; Y Y T X
FMUL ST,ST(1) ; Y*Y Y T X
FLD ST(3) ; X Y*Y Y T X
FMUL ST,ST(4) ; X*X Y*Y Y T X
FADDP ST(1),ST ; S Y T X
MOV SI,4*N
FLD U[SI-4] ; B S Y T X
FLD U[SI] ; A B S Y T X
SUB SI,8
MOV CX,N-1
AGAIN:
FLD ST(4) ; T A B S Y T X
FMUL ST,ST(1) ; A*T A B S Y T X
FADDP ST(2),ST ; A B+A*T S Y T X
FMUL ST,ST(2) ; A*S B+A*T S Y T X
FSUBR U[SI] ; NEW_B NEW_A S Y T X
SUB SI,4
FXCH ST(1) ; NEW_A NEW_B S Y T X
LOOP AGAIN
FMUL ST(5),ST ; A B S Y T A*X
FMULP ST(3),ST ; B S A*Y T A*X
FADDP ST(4),ST ; S A*Y T B+A*X
FSTP ST(0) ; A*Y T B+A*X
FSTP R+4 ; T B+A*X
FSTP ST(0) ; B+A*X
FSTP R ; empty stack
```

*EVALUATION OF THE CONDITION CODE*

```

JTABLE DW UNNORM,NAN,NORM,INFINITY,ZERO,EMPTY,DENORM,EMPTY
STATUS DW ?

        FXAM                ; SET CONDITION CODE BITS PROPERLY.
        FSTSW STATUS        ; GET THE STATUS WORD.
        FWAIT               ; WAIT FOR IT.
; MUST DO A LOT OF SHIFTS TO GET ALL OF THE BITS IN THE PROPER
; POSITION (i.e, TO GET C0, C2, AND C3 INTO BITS 0, 1, AND 2).
        MOV AL,BYTE PTR STATUS+1 ; GET HIGH BYTE OF STATUS.
        MOV BL,AL           ; SAVE C0
        AND BL,1            ; IN BL.
        SHR AL,1            ; DIVIDE BY 2 TO GET C1.
        MOV BH,AL          ; SAVE C1
        AND BH,2            ; IN BH.
        SHR AL,1            ; DIVIDE BY 8 TO
        SHR AL,1            ; GET C2 INTO
        SHR AL,1            ; PROPER POSITION.
        AND AL,4            ; MASK OUT EVERYTHING ELSE THAN C2.
        OR AL,BH            ; GET BACK BH.
        OR AL,BL            ; GET BACK BL.
; NOW HAVE 0-7 IN AL.  NEED 0-14 IN SI:
        MOV AH,0
        MOV SI,AX
        ADD SI,AX
        JMP JTABLE[SI]

;
UNNORM:
NAN:
NORMAL:
INFINITY:
ZERO:
EMPTY:
DENORM:

```

## A SLEAZY SOLUTION OF THE QUADRATIC EQUATION

$$0 = X^2 + A X + B$$

```

A   DD   ?           ; DON'T CARE ABOUT THE VALUES OF THE COEFFICIENTS.
B   DD   ?
X1  DD   ?           ; PLACE FOR THE RESULTS.
X2  DD   ?
STATUS DW ?         ; PLACE FOR THE 8087 STATUS WORD.

      FCLEX           ; GET RID OF ANY PREVIOUS EXCEPTION FLAGS.
      ; ST0          ST1          ST2          ST3          ST4
      FLD  A           ;      A
      FLD1           ;      1          A
      FADD ST,ST(0)   ;      2          A
      FDIVP ST(1),ST ;      A/2
      FCHS           ;      -A/2
      FLD ST(0)       ;      -A/2      -A/2
      FMUL ST,ST(1)   ;      (A/2)2    -A/2
      FSUB B          ;      (A/2)2-B    -A/2
      FSQRT           ;      SQRT      -A/2
      FLD ST(0)       ;      SQRT      SQRT      -A/2
      FADD ST,ST(2)   ;      X1        SQRT      -A/2
      FSTP X1         ;      SQRT      -A/2
      FSUBP ST(1),ST ;      X2
      FSTP X2         ;
      FSTSW STATUS    ; GET STATUS WORD INTO MEMORY.
; THE ERRORS WE HAVE TO LOOK FOR ARE:
; IE  INVALID -- THIS OCCURS IF ROOTS ARE COMPLEX.
; OE  OVERFLOW -- THIS OCCURS IF A WAS TOO BIG.
; PE  PRECISION -- THIS OCCURS IF ROOTS ARE CLOSE TO ZERO OR A/2.
; OF THE OTHER EXCEPTIONS, ZE (ZERODIVIDE) CANNOT OCCUR AND DE
; (DENORMAL) AND UE (UNDERFLOW) ARE NOT OF INTEREST TO US.
      FWAIT
      TEST STATUS,1   ; IE?
      JNZ  INVALID
      TEST STATUS,8   ; OE?
      JNZ  OVERFLOW
      TEST STATUS,32  ; PE?
      JNZ  PRECISION

```

OK:

LECTURE NOTES FOR CS-5330: IBM PC ASSEMBLY LANGUAGE  
UNIVERSITY OF TEXAS AT DALLAS

## CLASS 16

Comments

1. In the previous lecture I gave a handout describing the specifications for a final project that involved manipulations of a data base. This project consisted of 7 or 8 programs, of which I specified that each person would write 3. *Please* do not (like a bunch of dummies) all choose the 3 easiest programs. The programs together provide many useful database functions, but individually the programs are of no use. That is, if you had a set of all of the programs, you could use them to do something! If you don't, you can't. Therefore, we need to make sure that every program is done by at least one person. Also, you *must* stick to the specifications of the problem! Otherwise, your programs could not be used with somebody else's programs -- i.e., they cannot be used at all. All of the final projects (that are generally useful) and the best of the midterms will be distributed to the entire class. Among the projects I have heard about so far are:

- Sorting of files
- Enciphering and deciphering of files
- Database manipulation
- Fourier transforms
- A parsing utility
- Macros for high resolution graphics
- Device drivers for:
  - Interrupt-driven serial I/O
  - User-definable character sets
- Factoring of numbers
- etc.

Once again, I have not yet had the chance to prepare an alternate final project assignment, but I hope to in the next couple of days.

2. Also, I have not looked at any of the midterm projects that have been handed in, so be patient.

Review

In the previous class we discussed most of the 8087 numeric coprocessor instruction set. We discussed the pseudo-op

```
.8087
```

which informs the assembler that 8087 instructions are to be used. We also discussed the instructions

```
FINIT    -- initialize the 8087 coprocessor.
Fop      -- perform the operation op=ADD, SUB, MUL, or DIV.
FopP     -- same as Fop, but pop the stack.
FopR     -- same as Fop, but in reverse order.
FIop     -- same as Fop, but integer memory variable.
FIopR    -- same as FIop, but in reverse order.
FXCH     -- exchange registers.
```

```

FCLEX      -- clear exception flags in the status word.
FSTSW/FNSTSW -- save the status word in memory.
FXAM       -- set the condition code in the status word.
FABS       -- take absolute value.
FCHS       -- change sign.
FNOP       -- do nothing.
FRNDINT    -- round to integer.
FSQRT      -- take square root.
FLDLG2     -- load log(2), base 10.
FLDLN2     -- load ln(2).
FLDL2E     -- load log(e), base 2.
FLDL2T     -- load log(10), base 2.
FLDPI      -- load pi.
FLDZ       -- load zero.
FLD1       -- load one.

```

The status word, we found, was a word register in the 8087 that contained the exception flags -- which are like the 8088 flags in that they indicate various processing errors -- and also contained the condition code bits -- which are also like the 8088 flags in that they indicate the type of value being processed.

The six exception flags -- bits 0 through 5 of the status word -- indicated the following conditions: IE (invalid operand), DE (denormal number), ZE (zero divide), OE (overflow), UE (underflow), and PE (precision loss). These flags are "sticky bits". Once set they remain set until the FCLEX (clear exception) instruction is executed.

The condition code bits  $C_0$ - $C_3$  -- bits 8,9,10, and 14 of the status word -- serve two purposes. First, they are used to indicate the result of comparison operations (discussed today). Second, they are used with the FXAM (examine ST) instruction to indicate if the stack top is a "special" number.  $C_1$  is used to indicate the *sign*, while  $C_0$ ,  $C_2$ , and  $C_3$  together form a 3-bit number (with value 0-7) indicating that the number is

```

0    unnormal
1    NAN
2    normal
3    infinity
4    zero
5,7  empty
6    denormal

```

Of these, only the NAN, denormal, and unnormal numbers do not have an obvious meaning. A NAN (not-a-number) is an illegal bit pattern not corresponding to any specific floating-point number. A denormal number is a number with an extended exponent range (beyond the TEMPORARY REAL limit of 1E-4932), but with a compensating reduced precision. An unnormal number is the result of a calculation on a denormal number.

### Comparison Instructions

Just as the 8088 has a CMP instruction which sets the CPU flags in such a way as to indicate the relationship (<, >, =, etc.) between two numbers, the 8087 has various comparison instructions that serve the

same purpose (but for 8087 data types). Here is a list of the comparison instructions provided:

```
FCOM
FCOMP
FCOMPP
FICOM
FICOMP
FTST
```

We will not exhaustively consider every possible syntactical form of these, since they are all outlined in the handout of the previous lecture. Basically, each of these instructions compares *something* (i.e., some memory variable or some explicit or implicit register or value) to the top of the stack, ST. Here is how the condition code bits are set on the basis of a comparison:

| <u>C0</u> | <u>C2</u> | <u>C3</u> | <u>CONDITION DETECTED</u>        |
|-----------|-----------|-----------|----------------------------------|
| 0         | 0         | 0         | ST > <i>compared value</i>       |
| 0         | 0         | 1         | ST < <i>compared value</i>       |
| 1         | 0         | 0         | ST = <i>compared value</i>       |
| 1         | 1         | 1         | don't know (for example, ST=NAN) |

Notice that, unlike the 8088 CMP instruction, there is no meaningful relation between these flags and the flags we would see using FSUB followed by FXAM. (The 8088 CMP instruction, on the other hand, is almost the same as the 8088 SUB instruction.)

The only difference between the various comparison instructions lies in which value is specified to be compared to ST, and in the number of stack elements popped after the operation. The instructions with one trailing "P" pop one stack element, and the instruction FCOMPP pops the top two elements.

The instructions FCOM, FCOMP, and FCOMPP are used to compare ST to a real variable in memory, or to 8087 registers. FCOMPP is more restricted than the other two: it always compares just ST and ST(1). (Any other operands for FCOMPP wouldn't make sense since only ST and ST(1) can be popped by the instruction.)

The instructions FICOM and FICOMP compare ST to an integer variable in memory. (An instruction like FICOMPP wouldn't make any sense since ST(1) must be a TEMPORARY REAL and not an integer.)

Finally, the instruction FTST compares ST to the value 0.0. That is, the condition codes afterward reflect whether ST was positive, negative, zero, or NAN.

Fortunately, the condition codes have been defined in such a way that if we can manage to get the more significant byte of the status word into the 8088's flag register, then we can use the 8088 conditional jump instructions in a very natural way. Of course, to understand this assumes that we understand how the bits of the 8088 flag register are arranged and that we recall the relationship between the various conditional jumps and the settings of the 8088 flags. The 8088 flag register is arranged like this: bit 0=CF, bit 2=PF, bit

4=AF, bit 6=ZF, and bit 7=SF. Thus, the following code fragment would put C<sub>0</sub> into CF, C<sub>2</sub> into PF, and C<sub>3</sub> into ZF:

```
FSTSW     STATUS           ; GET STATUS WORD.
FWAIT
MOV       AH,BYTE PTR STATUS+1
SAHF                                           ; STORE UPPER BYTE IN FLAGS REGISTER.
```

Looking at the condition code table given earlier, the parity flag (PF) will be set only for the NAN condition. Thus, executing

```
JP        NAN
```

would leave us with only the cases

| ZF | CF | CONDITION  |
|----|----|------------|
| 0  | 0  | ST > value |
| 0  | 1  | ST < value |
| 1  | 0  | ST = value |

These are exactly the conditions used with the normal 8088 JA, JAE, JB, JBE, JE, JNE instructions. (Incidentally, since we normally redefine the symbol "JP" as a macro rather than an instruction, the above JP won't work. Fortunately, there is an alternate name for the jump-on-parity instruction, namely "JPE".) For example, if we used the instruction

```
JBE address
```

the jump would be taken only if ST is less than or equal to the tested value. For ease of use, all of this can be combined in the following macro:

```
; USED AFTER AN 8087 COMPARISON OPERATION. THE ALLOWED CONDITIONS
; ARE A, AE, B, BE, E, NE. THE ARGUMENTS ARE OPTIONAL. IF NANADDRESS
; IS MISSING, NO NAN EXIT IS TAKEN. IF ADDRESS (AND CCC) ARE MISSING,
; ONLY THE NAN EXIT IS USED.
FJP MACRO      CCC,ADDRESS,NANADDRESS
LOCAL        STATUS,CONTINUE,DONE
FSTSW       CS:STATUS           ; GET STATUS WORD INTO MEMORY.
FWAIT
MOV         AH,BYTE PTR CS:STATUS+1 ; LOAD UPPER BYTE INTO AH
SAHF
JNP        CONTINUE           ; IF NOT A NAN, CONTINUE.
IFNB      <NANADDRESS>
JMP        NANADDRESS        ; IF A NANE, JUMP TO NAN
ELSE
JMP        DONE
ENDIF
STATUS DW    ?                ; STORAGE FOR STATUS WORD.
CONTINUE:
IFNB      <ADDRESS>
J&CCC     ADDRESS            ; JUMP ON SPECIFIED CONDITION.
ENDIF
DONE:
ENDM
```

To see how we might use the comparison operations, let us suppose that we want to write a program fragment to read three numbers A, B, and C from memory, rearrange them in ascending order, and write them back into memory. Here is an algorithm to do that:

```

;
; ST0      ST1      ST2
FLD  A      ; GET A INTO 8087.      A
FLD  B      ; GET B INTO 8087.      B      A
FCOM                ; COMPARE A AND B.
FJP  AE,OK1  ; IF B>=A, OKAY.
FXCH                ; OTHERWISE, SWAP.      MAX(A,B)  MIN(A,B)
; AT THIS POINT, WE KNOW THAT ST0>=ST1.
OK1: FLD  C      ; GET C INTO 8087.      C      MAX(A,B)  MIN(A,B)
FCOM                ; COMPARE C TO MAX.
FJP  AE,OK2  ; IF C>=ST, THEN DONE.
; AT THIS POINT, WE KNOW THAT EITHER C MUST BE SWAPPED WITH ST1,
; OR ELSE C (ST), ST1, AND ST2 MUST BE ROTATED:
FCOM ST,ST(2)  ; COMPARE C AND MIN(A,B).
FJP  AE,SWP   ; IF GREATER, JUST SWAP C AND ST1.
; NEED TO ROTATE ST0, ST1, AND ST2:
FSTP ST(3),ST ;
JMP  OK2
; NEED TO SWAP ST0 AND ST1:
SWP: FXCH                ;
; DONE. AT THIS POINT, ST0>=ST1>=ST2:
OK2: FSTP C      ;
FSTP B      ;
FSTP A      ;
;
; empty

```

As mentioned before, our systematic discussion of the 8087 is now at an end (though we will return to discussing the 8087 occasionally from now on), so we will proceed to the next topic: The IBM PC BIOS.

**ASSIGNMENT:** Read Chapter 6 in the text.

### The IBM PC BIOS

MS-DOS, like many other operating systems, is built up in several "layers". Lower levels of the operating system are able to handle very simple and primitive operations, like displaying a character on the screen, while higher levels are able to do more sophisticated operations like managing the file system. In general, every sophisticated operation handled by the higher levels of the operating system is made up of a number of primitive operations handled by the lower levels. In particular, higher levels in the operating system do not directly perform I/O of any kind. They often appear to do I/O, but in reality they are just calling the lower levels of the operating system, which then do all of the real work.

For our purposes, we will speak of DOS as having just two such layers. The lower level is known as the "BIOS" (Basic I/O System), while the higher level is known (confusingly) as "DOS". The reason for making such distinctions has to do with compatibility of programs when run on various different computers. We have already seen how I/O operations can be performed by using DOS interrupt 21H -- programs using DOS interrupts for I/O will run on any MS-DOS based computer (such as an IBM PC, a TI PC, etc.) because it is the *essential job* of

DOS to provide a uniform interface between the user and the computer. MS-DOS itself, however, is ignorant of all hardware details of the computer. DOS does not understand about different brands of printers, different display cards, different keyboards, etc. DOS simply employs the BIOS to perform all primitive I/O functions and takes for granted that the BIOS understands all of the hardware details.

In general, then, MS-DOS itself is identical on all computers (except that one might have a newer or older version), but the BIOS differs on every machine, except close clones.

Just as our programs communicate with DOS by means of the "DOS interrupt" 21H (or other DOS interrupts we haven't discussed), our programs can communicate directly with BIOS by using BIOS interrupts. Many of the functions provided by BIOS interrupts are similar to those provided by some DOS functions, and for these DOS simply passes any function requests it gets directly to BIOS. This means that direct BIOS calls operate somewhat more quickly than the corresponding DOS calls, since the overhead of a DOS call is avoided. However, it is wiser (in my opinion) to avoid the practice of calling BIOS directly (except when absolutely unavoidable) for the sake of compatibility.

BIOS interrupts provide two types of functions. First, it provides standard functions required by MS-DOS, with a calling sequence that is likely to be the same for every computer. There is no point using such functions since DOS itself provides a uniform interface to them. Second, BIOS provides functions which aid in using special hardware features of the machine and are therefore likely to differ in all machines that are not close clones (since the hardware features are likely to be different).

To take the simplest example of such a hardware-related BIOS call, the IBM PC has a BIOS interrupt for performing I/O on a cassette tape. (Cassette tapes used to be used for mass storage of programs and data before disk-drives became inexpensive. Of course, this practice was obsolete before the IBM PC was introduced, so it is unclear what the purpose of providing such I/O was.) However, (it is probable that) no clone of the IBM PC has a cassette port, and no such BIOS interrupt is generally available on other machines.

Thus, the situation we have is this: **a) generally, BIOS functions should be used only when they access a special (non-common) hardware feature of the machine; and b) therefore, these BIOS calls won't work on other machines and therefore should not be used.** Or, to summarize, BIOS calls should never be used.

In the very likely event that nobody will take this argument seriously except me, let us now discuss some of the available BIOS functions. The textbook describes many of these functions (including some useless ones) in great detail. I will adopt a similar practice, except that my chosen set of useful BIOS interrupts will be different. Before going into this, however, it would be useful to discuss the use of "interrupts" in the IBM PC and clones.

Interrupts

We have seen that it can be very useful to divide our programs up into separate *procedures*, each with some well-defined functionality, and to build up our programs by combining these building-block procedures, rather than to continually be rewriting the same algorithms into our programs. The operating system can be thought of as a set of such building-block utilities -- utilities to read and write characters, to open and close files, to input and output strings, etc. However, unlike our procedures, the DOS procedures cannot be *linked* to our programs; rather, they are always in memory. To call DOS procedures we must seemingly know their actual addresses and not merely their names.

Actually, MS-DOS avoids this problem by using the ability of the 8088 microprocessor to employ *software interrupts*. The software interrupt system is very much like a big jump-table, accessing all of the DOS procedures. At the beginning of memory (i.e., at address 0:0) there is a table of *interrupt vectors*. An interrupt vector is a doubleword pointer (segment:offset) to a piece of code very much like a procedure. As far as the program is concerned, however, these "procedures" are accessed by interrupt-number rather than by address. Interrupt numbers run from 0 to 255, so the first  $256*4=1024$  bytes of memory are used to store interrupt vectors.

This will, perhaps, be clearer with examples. The first four positions in memory (i.e., interrupt vector 0) constitute the segment:offset of a "procedure" that we will refer to as `DIVIDE_BY_ZERO`. The next four positions (interrupt vector 1) point to a procedure `SINGLE_STEP`. The next four (interrupt vector 2) point to `NON_MASKABLE`, etc. Now, if we had the desire to call the procedure `DIVIDE_BY_ZERO`, we could simply do so with a `CALL` instruction if we happened to know its address. (Actually we couldn't, since `DIVIDE_BY_ZERO` is not actually a procedure -- we are simply pretending that it is. For reasons discussed below, we would first have to use the `PUSHF` instruction to push the flag register onto the stack, and then we would have to execute a `FAR CALL` to the procedure.) Since this address is stored at location zero, we could (with a few instructions) retrieve the address and call the procedure. Similarly, with jump-table type manipulations we could call either `SINGLE_STEP` or `NON_MASKABLE`.

This is too much work, since with the software interrupt system we can each procedure with just one instruction:

```
INT  interrupt_number
```

calls the indicated procedure in the interrupt vector table. Thus, `INT 0` would call `DIVIDE_BY_ZERO`, `INT 1` would call `SINGLE_STEP`, ..., `INT 21H` would call procedure 33 (whose address is stored at  $4*21H=84H$ ), etc.

Actually, we have been oversimplifying. The interrupt vectors are used not just by *software interrupts*, but by *hardware interrupts* as well. Various I/O devices attached to the computer are capable of "interrupting" the CPU by sending it an electrical signal when attention is desired. The CPU then responds by calling the appropriate procedure from its interrupt vector table. This is one of the reasons

why you can type on the computer's keyboard and not lose any characters, even if the computer is busy -- pressing a key on the keyboard sends an interrupt to the 8088, which then stops whatever it is doing and processes the interrupt (the key press) immediately. Since hardware interrupts can occur asynchronously -- that is, at any time -- it is important for the interrupt procedures to ensure that all registers (especially the flag register) are the same when they return as when they were called. For this reason, hardware interrupts and software interrupts (using INT) *automatically* push the 8088 flag register onto the stack, as well as performing a FAR CALL to the interrupt routine. There is also a special form of the RET instruction to pop the flag register on return. This special instruction is the IRET instruction -- **all interrupt-processing routines must be terminated with IRET rather than with RET**. This is what I mean by saying that interrupt-processing routines, with addresses stored in the interrupt vector table, are not true procedures (at least, as we have used the term in the past). On the other hand, the interrupt processing routines are almost certainly "procedures" in the sense that they were assembled with PROC and ENDP directives.

Interrupt-driven I/O is much more efficient than *polled* I/O, in which the operating system periodically checks to see if any device wants to be service. Prior to the IBM PC, microcomputers typically did not provide interrupt-driven I/O (presumably because it was a little more trouble to do). Therefore, this is one of the few areas in which IBM deserves praise for providing quality rather than sleaze, as far as the IBM PC is concerned.

#### Useful BIOS Interrupts

The following are the IBM PC BIOS interrupts in which we will be interested (I am told that some of them do not work on the TI PC, but I have been unable as yet to come up with the appropriate TI information):

| <u>INTERRUPT</u> | <u>FUNCTION</u>          |
|------------------|--------------------------|
| 5H               | Print screen             |
| 10H              | Video I/O                |
| 11H              | Equipment check          |
| 14H              | Serial I/O               |
| 1BH              | Keyboard break           |
| 1CH              | Timer tick               |
| 1FH              | Graphics character table |

Most of the other BIOS interrupts mentioned in the book should never be used (for the reasons mentioned earlier). Indeed, most of the interrupts we will discuss have only a limited use (or are used only when you are trying to program very cute and clever things).

Let us discuss these interrupts in numerical order.

INTERRUPT 5H prints the screen on the printer. It is equivalent to pressing the print-screen button on the keyboard and is, in fact, executed when the print-screen button is pushed. The reading assignment discusses how to use this feature in your program, so we won't go into it further here. This interrupt is interesting in another respect, in that it is one of the few interrupt-vectors you can

profitably change. Suppose that you have written a program which performs some function that you would like to *always* have available at the touch of a key, but that you don't want to build in to all of your programs (or into WordStar, the Macro Assembler, etc.). For instance, you might want to press a key and have the time of day automatically displayed in the upper right-hand corner of the screen. One way you could provide such a function is this: If you could arrange for your program to be kept always in memory, then you could go to the position in the interrupt vector table which stores the address of interrupt routine 5 (i.e., address  $4*5=20$ ), and you could *change* the address to be that of your program. Then, whenever the print-screen button is pressed, your program would be executed rather than the print-screen function. Actually, it is slightly more complicated than this since there are requirements of such interrupt routines which we haven't discussed, nor have we discussed how to keep programs in memory rather than having to continually reload them from disk. These things can all be done, however, and I am willing to tell you how to do them (in private).

For the moment, let us skip interrupt 10H since it is the most complex of the interrupts. We will come back to it in a moment.

INTERRUPT 11H the "equipment check" function. This *does not* check that the equipment is functioning correctly; rather, it returns a word which gives an inventory of some of the devices which are *supposed* to be attached to the PC. (Actually, it reads the values of various switches which were supposed to have been correctly set by the owner of the computer to indicate what peripheral devices are available for the program to use.) This information is useful if the program is intended to run on several different computers that may be equipped differently. Of course, so far all of our programs would have run on any MS-DOS based machine, so this information would not have been of use. The textbook gives the format of this inventory word, so we will not talk about it further, except to say that the basic use of the equipment check function is that the following questions can be answered from it:

- 1) Is there a printer attached?
- 2) Is there a serial port attached?
- 3) Is an IBM compatible monochrome or color display card used (and, if so, which one)?

These questions are important if you use some of the BIOS functions mentioned below or if you try to directly program the hardware.

INTERRUPT 14H performs serial I/O using the RS-232 interface. We have not talked about serial I/O (though we may do so later), so this will not be meaningful to many of us. For those to whom serial I/O is no mystery, let me say the following. It is almost impossible to perform serial I/O using MS-DOS or BIOS functions. It is almost always necessary to directly program the serial I/O hardware (the Universal Asynchronous Receiver-Transmitter, or "UART"). Therefore, interrupt 14H is almost useless for I/O. However, interrupt 14H also has an initialization function, as well as providing I/O functions, and can profitably be used to *initialize* the serial hardware. In order to initialize the serial interface, we must: a) load AH with 0 (to select the initialization function rather than the I/O functions); b) load AL

with a byte indicating the desired parameters of the serial I/O protocol; and c) INT 14H. Here is the meaning of the various bits that must be assigned via the AL register:

| <u>BITS</u> | <u>PARAMETER</u> | <u>MEANING OF THE VALUES</u>                                      |
|-------------|------------------|---|
| 7,6,5       | baud             | 0=110 baud, 1=150, 2=300, 3=600<br>4=1200, 5=2400, 6=4800, 7=9600 |
| 4,3         | parity           | 0=ignore, 1=odd, 3=even   |
| 2           | stop bits        | 0=1 stop, 1=2 stops   |
| 1,0         | word length      | 2=7 bits, 3=8 bits  |

If, for example, we wanted to use 9600 baud, no parity, 1 stop bit, and 8 bit words, we would use the code

```
MOV  AL,11100011B
MOV  AH,0
INT  14H
```

where, for clarity, the fields in AL are successively in italics and in normal type.

INTERRUPT 1CH is used when the ctrl-break key is pressed. Thus, like interrupt 5, the interrupt vector for this function could (in some circumstances) be profitably changed to point to a routine of yours rather than the ctrl-break routine selected by DOS.

INTERRUPT 1DH is the timer tick interrupt. As mentioned earlier, the interrupt routines are executed not only when an INT instruction (a software interrupt) is executed, but when a *hardware* interrupt occurs - that is, when certain electrical signals are sent to the CPU by peripheral devices. One type of hardware interrupt, generated by a "clock" chip, causes the CPU to update a count in memory. Since these interrupts occur regularly (18.2 times per second), this count can be interpreted as a "time of day". Now, when the time-of-day interrupt occurs, an interrupt 1DH is also executed. The 1DH interrupt actually does nothing, but the interrupt vector could be changed to point to one of your own routines. Thus, if you can think of anything you'd like to do 18.2 times per second, this interrupt could profitably be used to do it. The most common use for this interrupt would be to provide a real-time clock which continually displays the correct time (at least, accurate to within 1/18.2 seconds) on the screen.

INTERRUPT 1FH is not actually an executable interrupt. rather, the interrupt-vector table simply uses this position to store a useful address. That useful address is the address of a user-definable character set. Normally, interrupt vector 1FH (that is, the value stored at 4\*1FH=7CH in memory) has the value 0:0, which is used to indicate that no user-definable characters are used. If, instead, this is changed to some other value, the value is interpreted as the address of a table of character shapes. The user-definable characters are the codes 128-255 (with the shapes of the normal ASCII characters 0-127 being fixed). The user-defined characters are seen only if the computer is in "graphics mode" (we will discuss what this means in a

moment), and are irrelevant to the normally used 25x80 text mode of the display. We will discuss the form of the character definition table when we learn (in a future lecture) how the IBM PC video display is directly accessed by memory manipulations.

INTERRUPT 10H is the most useful BIOS interrupt to the programmer. Interrupt 10H is used to provide the standard character output function (you should not use it for this, however), as well as to provide some additional control over the video display. Many of the features which this interrupt is typically used to control are also controlled by the ANSI driver. The ANSI driver should be used in preference to interrupt 10H whenever possible, so I will only talk about interrupt 10H functions *not* provided by the ANSI driver.

Before doing this, however, we should talk a little about the various video modes of the IBM PC. In some modes, the PC is capable of displaying only alphanumeric characters. In others, it displays graphics and characters. The graphics modes are more flexible, but also operate more slowly.

The most common modes are the 80x25 black&white and the 80x25 color text modes. The "80x25" refers to the fact that the display consists of 80 columns and 25 rows. Whether or not color is available depends on whether the computer has a monochrome or a color monitor (i.e., CRT or screen) and on how the PC is equipped internally. Normally, an IBM PC is equipped internally with either a "monochrome display adapter" card or with a "color adapter card". (There are also other possibilities, such as a "Hercules graphics card", which we won't discuss.) Color adapter cards differ from monochrome adapter cards not only in that they display color, but also in that they allow the use of high-resolution graphics. With a monochrome card, only the 80x25 black&white (or 40x25) display mode can be used. With a color adapter, there are two typically used high-resolution graphics modes as well: the 320x200 color graphics mode and the 640x200 b&w graphics mode.

The picture you see displayed by the computer is actually composed of a large number of small dots called "pixels". In black&white mode, each pixel is either black (invisible) or white (visible). In color mode, each pixel can be one of several colors. When alphanumeric (ASCII) text is displayed, the computer itself actually decides which pixels are visible and which are invisible, on the basis of its knowledge of the shapes of the characters. When graphics are displayed, however, each pixel on the screen is individually set or cleared by the program. Thus, pictures of arbitrary complexity can be drawn, up to the "resolution" of the display mode. In the 320x200 color mode, the computer resolves the displays into 320 columns of pixels and 200 rows of pixels. Each pixel can be one of four colors, from a "palette" of colors selected by the program. In 640x200 mode, there are instead 640 columns of pixels, so the resolution is twice as great in the horizontal direction. On the other hand, each pixel can be only black or white. The computer is also smart enough to allow you to continue to use alphanumeric characters in "graphics" mode.

Like DOS interrupt 21H, the BIOS video I/O interrupt 10H allows a selection of various functions, based on the value of the AH register. Function AH=0 is the "select video mode" function, and selects from

among the video modes mentioned above (and others). This function is not to be used, since the ANSI driver provides this feature as well.

Function AH=1 is probably the most useful function. It allows you to set the size of the cursor. The cursor consists of a series of horizontal lines. For the monochrome card, these lines are numbered 0 to 13 (from top to bottom), while for the color card they are numbered 0 to 7 (from top to bottom). Normally the cursor consists of lines 11 and 12 in monochrome and lines 6 and 7 in color. You can, however, change this default. The arguments for this function are: the top cursor line number in CH, and the bottom cursor line number in CL. Thus,

```
MOV  AH,1          ; SELECT CURSOR SIZE.
MOV  CH,0          ; TOP LINE IS TOP.
MOV  CH,7          ; BOTTOM LINE IS BOTTOM.
INT  10H          ; CALL BIOS.
```

would change the cursor to a full-size block with a color adapter, and to half of a block with a monochrome adapter.

Function AH=2 is used to change the cursor position. Normally, this is of no interest since the ANSI driver allows us to do the same thing with the CUP command. However, if we are using several video "pages" this becomes important. The monochrome adapter allows no extra pages; the color adapter, on the other hand allows up to four video pages. What is a "page"? We can think of the video display as a book, in which at any one time we can only see one page. There are, however, other pages, and if we knew how to change the pages we could see what was being displayed there. Normally, what is displayed is page 0. With the color adapter there are also pages 1, 2, and 3. Function AH=2 does not allow us to *change* the pages, but it does allow us to move the *cursor* to a new page. (That is, since the cursor would then be on a page not displayed, we wouldn't see it any more.) Function AH=5 is used to change the active display page. Thus, functions AH=2 and AH=5 are used in conjunction with each other. (To see the exact arguments of these functions, look on p. 206 of the text.) The importance of this fact is that if we have some complicated display on the screen, then we can switch to another video page, display something else, and then return to our original page without having to redraw the complicated display. This is useful for many things, such as help screens and so forth. Here is a short "typewriter" program that illustrates the use of a paged display: it switches to page 1, does I/O for a while, and then flips back to an undisturbed page 0:

```

; CHANGE TO PAGE 1.
    MOV  AH,5           ; PAGE DISPLAY FUNCTION.
    MOV  AL,1           ; PAGE 1
    INT  10H           ; CALL BIOS.
; MOVE CURSOR TO PAGE 1.
    MOV  AH,2           ; MOVE CURSOR FUNCTION.
    MOV  DH,0           ; ROW 0.
    MOV  DL,0           ; COLUMN 0.
    MOV  BH,1           ; PAGE 1.
; TYPEWRITER PART.
AGAIN:
    GETCHR              ; GET KEYBOARD CHARACTER.
    CMP  AL,27          ; ESCAPE?
    JE   DONE           ; QUIT IF SO.
    PUTCHR              ; DISPLAY ON THE SCREEN.
    JMP  SHORT AGAIN
; CHANGE BACK TO PAGE 0.
    MOV  AH,5           ; PAGE DISPLAY FUNCTION.
    MOV  AL,0           ; PAGE 0
    INT  10H           ; CALL BIOS.
; MOVE CURSOR TO PAGE 0.
    MOV  AH,2           ; MOVE CURSOR FUNCTION.
    MOV  DH,0           ; ROW 0.
    MOV  DL,0           ; COLUMN 0.
    MOV  BH,0           ; PAGE 0.

```

In theory, the row and column selected by the AH=2 function are based on (0,0) being the upper left-hand corner of the screen. In practice, (0,0) seems to be the last position used before the page was changed! Thus, the sample program shown ends up at the original cursor position on the original page. (And, if it is executed again, the cursor for the typewriter part will start at the same position it left off at before.)

Though there is no need to say more about them than the book says already, functions AH=6 and AH=7 ("scroll active page up" and "scroll active page down") are also rather useful. They allow any rectangular "window" in the active display page to be scrolled up or down by any number of lines.

In general, there are no other useful BIOS interrupt 10H functions. There are several character output functions whose basic virtues seem to be that they allow various "attributes" (blinking, reverse video, background color, etc.) of the character to be set. Of course, these functions are provided by the ANSI driver, so they are of no special interest to us. Though not generally useful, in the next lecture we will discuss several of the graphics functions provided in interrupt 10H.

MACRO TO CONDITIONALLY JUMP FOR AN 8087 CONDITION

```

; USED AFTER AN 8087 COMPARISON OPERATION.  THE ALLOWED CONDITIONS
; ARE A, AE, B, BE, E, NE.  THE ARGUMENTS ARE OPTIONAL.  IF NANADDRESS
; IS MISSING, NO NAN EXIT IS TAKEN.  IF ADDRESS (AND CCC) ARE MISSING,
; ONLY THE NAN EXIT IS USED.
FJP MACRO      CCC,ADDRESS,NANADDRESS
  LOCAL      STATUS,CONTINUE,DONE
  FSTSW      CS:STATUS          ; GET STATUS WORD INTO MEMORY.
  FWAIT      ; WAIT FOR IT.
  MOV        AH,BYTE PTR CS:STATUS+1 ; LOAD UPPER BYTE INTO AH
  SAHF      ; AND FROM THERE TO FLAGS REG.
  JNP        CONTINUE          ; IF NOT A NAN, CONTINUE.
  IFNB      <NANADDRESS>      ; IF A NANE, JUMP TO NAN
    JMP      NANADDRESS        ; PROCESSOR.
  ELSE
    JMP      DONE
  ENDIF
STATUS DW    ?                ; STORAGE FOR STATUS WORD.
CONTINUE:
  IFNB      <ADDRESS>
    J&CCC    ADDRESS          ; JUMP ON SPECIFIED CONDITION.
  ENDIF
DONE:
  ENDM

```

CODE TO ARRANGE THREE FLOATING POINT NUMBERS IN ASCENDING ORDER

```

A   DD   ?
B   DD   ?
C   DD   ?

;
; ST0      ST1      ST2
FLD  A      ; GET A INTO 8087.      A
FLD  B      ; GET B INTO 8087.      B      A
FCOM ; COMPARE A AND B.
FJP  AE,OK1 ; IF B>=A, OKAY.
FXCH ; OTHERWISE, SWAP.      MAX(A,B)  MIN(A,B)
; AT THIS POINT, WE KNOW THAT ST0>=ST1.
OK1: FLD  C      ; GET C INTO 8087.      C      MAX(A,B)  MIN(A,B)
FCOM ; COMPARE C TO MAX.
FJP  AE,OK2 ; IF C>=ST, THEN DONE.
; AT THIS POINT, WE KNOW THAT EITHER C MUST BE SWAPPED WITH ST1,
; OR ELSE C (ST), ST1, AND ST2 MUST BE ROTATED:
FCOM ST,ST(2) ; COMPARE C AND MIN(A,B).
FJP  AE,SWP  ; IF GREATER, JUST SWAP C AND ST1.
; NEED TO ROTATE ST0, ST1, AND ST2:
FSTP ST(3),ST ;      MAX(A,B)  MIN(A,B)      C
JMP  OK2
; NEED TO SWAP ST0 AND ST1:
SWP: FXCH ;      MAX(A,B)      C      MIN(A,B)
; DONE.  AT THIS POINT, ST0>=ST1>=ST2:
OK2: FSTP C      ;      MID(A,B,C)  MIN(A,B,C)
      FSTP B      ;      MIN(A,B,C)
      FSTP A      ;      empty

```

A TYPEWRITER PROGRAM WITH PAGING

```
; CHANGE TO PAGE 1.
    MOV AH,5           ; PAGE DISPLAY FUNCTION.
    MOV AL,1           ; PAGE 1
    INT 10H           ; CALL BIOS.
; MOVE CURSOR TO PAGE 1.
    MOV AH,2           ; MOVE CURSOR FUNCTION.
    MOV DH,0           ; ROW 0.
    MOV DL,0           ; COLUMN 0.
    MOV BH,1           ; PAGE 1.
; TYPEWRITER PART.
AGAIN:
    GETCHR             ; GET KEYBOARD CHARACTER.
    CMP AL,27         ; ESCAPE?
    JE DONE           ; QUIT IF SO.
    PUTCHR            ; DISPLAY ON THE SCREEN.
    JMP SHORT AGAIN
; CHANGE BACK TO PAGE 0.
    MOV AH,5           ; PAGE DISPLAY FUNCTION.
    MOV AL,0           ; PAGE 0
    INT 10H           ; CALL BIOS.
; MOVE CURSOR TO PAGE 0.
    MOV AH,2           ; MOVE CURSOR FUNCTION.
    MOV DH,0           ; ROW 0.
    MOV DL,0           ; COLUMN 0.
    MOV BH,0           ; PAGE 0.
```

LECTURE NOTES FOR CS-5330: IBM PC ASSEMBLY LANGUAGE  
UNIVERSITY OF TEXAS AT DALLAS

## CLASS 17

Comments

1. I have prepared an alternate final project for those of you who are not keen on the database manipulation project. The new project involves infinite precision arithmetic. I hope it interests you, because my imagination has been completely depleted. If you don't like this one, you'd better think of something else quite rapidly.

2. Several of you have noticed that the book by Jump, *Programmer's Guide to MS-DOS for the IBM PC*, is quite useful, but also quite unavailable. I have recently found another book, *The MS-DOS Handbook* by King, which is very similar to Jump's book, but with a concentration on somewhat different areas. Also, there is a new book by Peter Norton which could be useful; its title is similar to Jump's title. Both of the latter books are available in the Doubleday bookstore at Northpark shopping center.

Review

In the previous class, we completed our systematic discussion of the 8087 coprocessor and began discussing the IBM PC BIOS calls.

We discussed the 8087 comparison instructions,

FCOM  
FCOMP  
FCOMPP  
FICOM  
FICOMP  
FTST

which were somewhat similar to the 8088 CMP instruction. These instructions set various condition code bits (in the 8087 status word) as a result of the comparison. The status word then had to be stored in memory with the FSTSW instruction so that it could be examined by means of 8088 instructions. Fortunately, Intel arranged the condition code bits of the status word in such a way that they could be loaded into the 8088 flag register and used more-or-less directly and straightforwardly for conditional jumps.

We began discussing the IBM PC BIOS. The BIOS, or Basic I/O System is a "lower level" of the operating system which handles all of the hardware-related details of I/O. MS-DOS itself is above such stuff and really knows almost nothing about the hardware. Thus, DOS calls should be generally used for compatibility from computer to computer, but BIOS calls are sometimes unavoidable when special features of the hardware are to be used.

Programs interface with DOS and with BIOS via *software interrupts*. Software interrupts (the INT instruction) and hardware interrupts use the *interrupt-vector table* at the beginning of memory as a jump-table giving the addresses of various DOS and BIOS functions. There are 256

possible interrupts (0-255), each with a doubleword address, so the interrupt vector table is  $256*4=1024$  bytes long.

A number of interrupts are used to access DOS. The most common DOS interrupt is 21H, which we have used quite often. BIOS also has a number of interrupts assigned to it. In the previous lecture, we discussed 7 BIOS interrupts:

| <u>INTERRUPT</u> | <u>FUNCTION</u>                  |
|------------------|----------------------------------|
| 5H               | Print screen                     |
| 10H              | Video I/O control                |
| 11H              | Equipment check                  |
| 14H              | Serial I/O control               |
| 1BH              | Keyboard break function          |
| 1CH              | Timer tick                       |
| 1FH              | Point to definable character set |

There is no reason to go into these again, except for interrupt 10H, which is rather important.

Of the many functions provided by interrupt 10H, we have discussed so far only function AH=1 (which is used to select the cursor size), function AH=2 (which is used to select the cursor position), and function AH=5 (which is used to select the video "page" being displayed). In general, the ANSI CUP function should be used in preference to the AH=2 function unless the paged feature of the display is used. The video display is like a book, for which only one page can be seen at a time. Functions AH=2 and AH=5 together provide a way of switching from page to page.

We also discussed the most common video hardware with which the IBM PC can be equipped. The "monochrome card" allows only the display of text in black and white, with no paged display. The "color adapter" card, on the other hand, allows both text and graphics, in color, and with a paged display in text mode. There are also various other display cards, which we did not discuss; nor did we discuss the available hardware on other computers, such as the TI PC.

**ASSIGNMENT:** Read chapter 7 of the textbook. (This chapter is remarkable, in that it is one of the few chapters on graphics in *any* book on the IBM PC that never gets around to discussing anything about graphics!)

### Simple Computer Graphics

Since the computer display consists of a large number of closely spaced dots (pixels), the most primitive possible graphics operation is to light up one of these dots. Moreover, every less primitive graphics operation, such as drawing lines, circles, etc., can be construed as a sequence of dot operations. Clearly, then, the first thing we should learn how to do in computer graphics is to light up a pixel.

(This comment applies to the IBM PC color adapter and to most other types of graphics displays on microcomputers -- however, it is not universally true. With certain hardware, it is more reasonable to consider *line drawing* as the primitive operation. For the computers at

our disposal, this is not a real concern. As mentioned before, of course, with the IBM monochrome adapter, no graphics can be done.)

Unfortunately, there is no uniform way to do this on all computers. We will see, however, that we can still develop a uniform *software interface* for the graphics on each machine. To do this, we need a workable model of the display that we can use for every computer.

We will suppose, as mentioned, that the display is divided into a rectangular grid of pixels, each one of which can be either "on" (i.e., visible) or "off" (invisible). The number of *rows* of pixels on the screen will be called NUM\_ROWS, while the number of *columns* of pixels will be called NUM\_COLS. The rows are numbered from 0 to NUM\_ROWS-1, with row zero being at the top of the screen. The columns are numbered from 0 to NUM\_COLS-1, with zero being at the left of the screen. This model corresponds closely to the actual situation on the IBM PC and many other PCs. (For instance, on the TI PC, with three-plane graphics, NUM\_ROWS=300 and NUM\_COLS=720.) Our primitive graphics operation is embodied in a macro

```
DRAW_DOT ROW,COLUMN
```

which turns on the dot at the indicated row and column. Although we have not yet discussed how such a macro might be written, let us assume that such a macro exists on every computer -- i.e., on all IBM PCs, all TI PCs, etc. If this is the case, it would be logical for us to write all of our graphics procedures in terms of this hardware-dependent macro (and the constants NUM\_ROWS and NUM\_COLS) alone. We could then re-assemble our programs on any machine and they would work without change to the source code.

Note that the choice of DRAW\_DOT as a macro rather than a procedure is not arbitrary. When we do computer graphics, we typically turn on thousands or tens of thousands of dots. If the dot-drawing process has a lot of overhead associated with it -- for example, the time required to call a procedure -- the drawing speed of our program visibly slows down. With a macro, there is no time-overhead since there is no CALL, and the graphics can be drawn much more quickly.

There are several steps usually necessary in doing our computer graphics. First, when the computer is turned on, it is in "text" mode rather than "graphics" mode. That is, it can display text, but not graphics. To display graphics, we must first "turn on" graphics mode. This is done with the SM ANSI driver command. (Recall the ANSI driver handout.) This macro has the syntax

```
SM mode
```

where the mode operand specifies the desired mode of the display. The choices are

| <u>MODE</u> | <u>DESCRIPTION</u>    |
|-------------|-----------------------|
| 0           | 40x25 black and white |
| 1           | 40x25 color           |
| 2           | 80x25 black and white |
| 3           | 80x25 color           |

```

4          320x200 color
5          320x200 black and white
6          640x200 black and white

```

The first four modes are text modes (with only modes 2 and 3 being in common use), while the last three modes are graphics modes. (The ANSI driver also allows a MODE=7. This turns on line wrap -- typing past the end of the line automatically continues at the beginning of the next line. The ANSI command RM 7 turns off line wrap. Of course, this is not at all relevant to graphics.) Thus, the first thing we normally do in our graphics program is to use the ANSI command

```
SM 4      or      SM 5      or      SM 6
```

Once in one of these graphics modes, the screen corresponds to our model of it, with NUM\_ROWS=200 and NUM\_COLS=320 or 640.

Next, of course, we would typically use a sequence of DRAW\_DOT commands (or procedures using DRAW\_DOT commands) to turn on the correct arrangement of pixels.

Finally, after giving the viewer a chance to look at the display, we would finally return to text mode with, for example,

```
SM 3
```

Note that the SM command automatically clears the screen for us.

As the simplest example of an application for DRAW\_DOT, let us consider a macro to draw a horizontal line. A horizontal line is, of course, confined to some particular row (called, say, "ROW") of the display. To draw such a line, we would only need to set every pixel in ROW between the starting and ending columns:

```

; MACRO TO DRAW A HORIZONTAL LINE.  BY ASSUMPTION, COL1<COL2.
DRAW_HORIZONTAL MACRO ROW,COL1,COL2
    LOCAL AGAIN
    MOV  SI,COL1          ; USE SI TO COUNT COLUMNS.
AGAIN:  DRAW_DOT ROW,SI   ; SET THE PIXEL.
        INC  SI          ; NEXT COLUMN.
        CMP  SI,COL2     ; DONE?
        JBE  AGAIN
    ENDM

```

Since the only access to the graphics hardware in this macro is through the macro DRAW\_DOT, this macro is totally hardware independent.

Drawing horizontal (and vertical) lines is rather easy, as we have just seen. Drawing a general line between two arbitrarily chosen points on the screen is somewhat trickier. Indeed, at first glance, it may not even be entirely obvious what we mean by a "line" between two points. If by a "line" we mean the mathematically precise definition (the infinitely thin shortest path between two points) there might not even be any pixels on the screen (other than the endpoints) which are on the line. Clearly, however, this is not what we mean -- we mean to turn on all pixels which, in some sense, are *close* to being on the line, so that to the eye we have a good approximation of a straight

line. There are actually a number of ways of approaching this problem, with differing results. We will choose a rather simple approach.

To see how we might go about drawing such a "line", let us consider first the case in which we know that the line is nearly horizontal. "Nearly horizontal" will mean simply that the line is spread across more columns of pixels than rows of pixels. We might draw the line as follows: Within each column between the given endpoints of the line, turn on the pixel *vertically closest* to the mathematically correct line.

This approach is conceptually simple, but has the flaw that we do not immediately see any good way to calculate the indicated pixels. We obviously do not care to calculate the distance of each pixel in the column from the mathematically perfect line. Indeed, even if we could narrow the choice of pixels down to just a couple, calculating the distance of the pixel from the line requires several floating-point operations -- and since we are probably turning on tens of thousands of pixels, all of these calculations take a long time.

Fortunately, there is a way of faking all of these floating point operations with just a couple of word additions on the 8088, although how the method works requires a little thought to understand. To illustrate the algorithm, let us suppose that we want to draw a line between ROW1, COL1 and ROW2, COL2, where ROW2>ROW1 and COL2>COL1. Here is the algorithm in pseudo-code:

```

DX := COL2-COL1
DY := ROW2-ROW1
ROW := ROW1
K := DX/2
FOR COL := COL1 TO COL2 DO
BEGIN
    DRAW_DOT ROW, COL
    K := K+DY
    IF K>=DX THEN
    BEGIN
        K := K-DX
        ROW := ROW+1
    END
END
END

```

The key feature of this algorithm is, of course, the running count K. ROW is incremented every time K surpasses a multiple of DX. However, there are DX passes through the loop, and K is incremented by DY each time, so there are a total of  $(DX*DY)/DX = DY$  times that ROW is incremented. This is just what we want.

In practice, this algorithm becomes slightly more complex since the assumptions ROW2>ROW1, COL2>COL1, and DX>DY occur only about 1/8 of the time. All of the other possible cases differ only slightly from this one, however, and it is relatively easy to write a procedure that handles all possible cases. Such a procedure is featured on the door of my office. To avoid wasting class time, I will simply suppose (whenever the necessity arises) that there is a machine-independent macro

```
DRAW_LINE ROW1, COL1, ROW2, COL2
```

which draws a line between the points (ROW1,COL1) and (ROW2,COL2), and depends only on the hardware-dependent macro DRAW\_DOT.

To see how all of these things fit together, let us write a short program that does the following: First, it gets into graphics mode, and with DRAW\_LINE puts a box around the screen. Second, it goes to the middle of the screen and displays the message "Hello, bunky!" (recalling that text can be displayed in graphics mode, but not vice-versa). Third (and finally), it waits for the user to type any key, then it gets back into text mode and quits:

```
N    EQU  NUM_ROWS-1
M    EQU  NUM_COLS-1
MSG  DB   "Hello bunky!",13,10,'$'
    ...
SM   6
DRAW_LINE 0,0,0,M      ; LINE AT TOP OF SCREEN.
DRAW_LINE 0,M,N,M      ; LINE AT RIGHT.
DRAW_LINE N,M,N,0     ; LINE AT BOTTOM.
DRAW_LINE N,0,0,0     ; LINE AT LEFT.
CUP 13,15              ; GOTO CHARACTER POSITION 13,15.
DISPLAY  MSG
GETCHR
SM   3
```

### Implementing DRAW\_DOT Simple-mindedly

On an IBM PC, we can implement DRAW\_DOT very simply (and shoddily) using one of the remaining undiscussed functions of BIOS interrupt 10H. The relevant function is AH=12, as mentioned in the textbook.

Function AH=12 of interrupt 10H turns on a selected pixel. Basically, all we need to do is to select the column number, the row number, and the *color* of the pixel. Colors are only available in 320x200 color graphics mode. In 640x200 graphics mode, all graphics are black and white. We will therefore give DRAW\_DOT a third argument, COLOR -- if present, this argument will actually specify the color of the dot. If absent, the color white will be assumed. Similarly, DRAW\_LINE will be given a fifth argument (COLOR) which works the same way. The actual choices for COLOR will be discussed later, and until then the argument will simply be omitted as above. Here, then, is a simple-minded implementation of DRAW\_DOT using BIOS interrupt 10H:

```
draw_dot macro row,column,color
    mov  dx,row          ; select pixel row for BIOS function.
    mov  cx,column      ; select pixel column.
    ifb  <color>
        mov  al,3        ; if color not selected, use white.
    else
        mov  al,color    ; otherwise, use the selected color.
    endif
    mov  ah,12          ; use function AH=12.
    int  10H           ; call BIOS.
endm
```

The reason this implementation is so poor is the tremendous amount of time overhead involved. Recall that the INT instruction is like a CALL except that it performs some additional functions (therefore taking longer to do so). Also, once BIOS takes control, it has its own internal jump-table to select from among the functions specified by AH. In all, the overhead associated with using this DRAW\_DOT macro is several times longer than the time required to actually set the pixel. Indeed, if we run the sample program given in the last section, the operation is so slow that we can actually see the lines being drawn.

The only way around this problem is for our macro to directly access the hardware rather than to go through an intermediary like BIOS. We will begin discussing this in the next section. (Actually, even going directly to the hardware does not totally fix the speed problem, and in the end we would probably want to introduce *both* a hardware-dependent DRAW\_DOT and a hardware-dependent DRAW\_LINE. For now, however, we will not pursue this path.)

Another remaining BIOS 10H *graphics* function is also slightly interesting. Function AH=13 is the inverse of AH=12: It *reads* a dot rather than writing a dot. That is, it can examine a given pixel position and determine whether the pixel is on or off. There is another, similar, function AH=8, which reads a *character* position on the screen and returns the value of the character.

### Video Memory

These reading functions are possible because in an IBM PC the video display behaves very much like a section of memory -- indeed, in some sense the video display *is* a section of memory. This is easiest to think about (at first) in the case of a text display mode, for the monochrome display card. The monochrome display card actually contains 4K of memory, beginning at location B000:0000 in the PC. This memory can be accessed by the 8088 just like any other memory in the computer; however, it serves a special purpose. This memory, the *video memory*, specifies the image on the display. Now, for a display of 80 columns by 25 rows of text,  $80 \times 25 = 2000$  bytes would seemingly be necessary for specifying all of the information in the display. Actually, each character position on the screen is controlled by *two* bytes rather than just one. One of the bytes is the ASCII representation of the character, as we might have expected. The other, the *attribute byte*, specifies what attributes the character has: is it blinking?, is it underlined?, etc. Thus,  $2 \times 2000 = 4000$  bytes are really necessary to give all of the information in the display. This 4000 bytes agrees nicely with the fact that 4K of memory is available on the monochrome card itself.

Similarly, the IBM color adapter card has 16K memory on board, and this memory begins at address B800:0000 in the PC. Since only 4000 bytes are needed for a screenful of text, this leaves room on the card to store the remaining (normally unused) three display pages discussed in the previous lecture.

The form of the attribute byte is as follows

| <u>BIT</u> | <u>DESCRIPTION</u>             |
|------------|--------------------------------|
| 0          | foreground bit B               |
| 1          | foreground bit G               |
| 2          | foreground bit R               |
| 3          | I -- intensity (0=low, 1=high) |
| 4          | background bit B               |
| 5          | background bit G               |
| 6          | background bit R               |
| 7          | BL -- blink (0=no, 1=yes)      |

The bits R, G, and B go together to specify either a property of the character itself (a "foreground" property) or a property of the character's background field. As you may surmise from the "RGB", these bits are really intended to represent colors; the monochrome card has only black and white as the allowed colors. Here is a table of the allowed colors, when the I=1 (i.e., high intensity) and I=0 (low intensity):

| <u>RGB</u> | <u>COLOR (high intensity)</u> | <u>COLOR (low intensity)</u> |
|------------|-------------------------------|------------------------------|
| 0          | dark gray                     | black                        |
| 1          | light blue                    | blue                         |
| 2          | light green                   | green                        |
| 3          | light cyan                    | cyan                         |
| 4          | light red                     | red                          |
| 5          | light magenta                 | magenta                      |
| 6          | yellow                        | brown                        |
| 7          | white                         | light gray                   |

Thus, for the color adapter, a foreground RGB of 3 and a background of 2 would result in a cyan (whatever that is) character on a green background. For the monochrome card, only RGBs of 0 and 7 are allowed, except that a foreground RGB of 1 can be used to display an underlined character.

This information is really useful only in text mode and (of course) the ANSI driver (command SGR) can be used to set all of the mentioned attributes in any desired way. Nevertheless, we can now see how video memory is laid out. The active display page consists of 2000 consecutive words, of which the first (less significant) byte is the character, and the second (more significant) byte is the attribute. If we were to directly manipulate this memory with our programs, we could change the characters and/or attributes anywhere on the screen without going through BIOS. Here, for example, is a program which sticks an "A" in the middle of the screen:

```

MOV  AX,0B800H          ; PREPARE DISPLAY AS EXTRA SEGMENT.
MOV  ES,AX
MOV  ES:2000,"A"       ; PUT AN "A" AT POSITION 2000.

```

This little program assumes a color adapter; for a monochrome adapter, ES should have been set to 0B000H. To set the attribute byte rather than the character itself, we would have used address ES:2001.

I won't describe such screen manipulations in text mode any further because, as I have tried to get across, I don't approve of them except in the most performance-critical applications. Direct screen manipulations are very often not compatible from machine to machine.

Unfortunately, as mentioned earlier, because of speed considerations, direct screen manipulations are often unavoidable in graphics applications.

In the graphics modes, the memory requirements change and the video memory is layed out quite differently. There are three cases. In 640x200 mode,  $640 \times 200 = 128000$  bits or 16000 bytes are needed to contain the image, corresponding to the 16K memory on the color adapter. (Thus, no extra display pages are available.) In 320x200 black and white mode, just half of this (8000 bytes) is required. In 320x200 color mode, however, each pixel can have 4 different colors; each color requires two bits to specify it; so  $8000 \times 2 = 16000$  bytes are again required. For the sake of brevity, we will cover just the 640x200 mode and the 320x200 color mode.

#### The 320x200 Color Graphics Mode

To represent a row of pixels in 320x200 color graphics mode, we need  $(320 \text{ columns}) \times (2 \text{ color bits}) = 640 \text{ bits} = 80 \text{ bytes}$ , just the same as is needed to represent a row of characters. Indeed, each row of pixels *is*, in fact, *represented by 80 consecutive bytes in video memory*. Unfortunately, for some strange reason, the *even* numbered rows are in video memory at addresses B800:0000-1FFF, while the *odd* numbered rows are in video memory at addresses B800:2000-3FFF. Thus, row 0 begins at address B800:0000, row 1 begins at B800:2000, row 2 begins at B800:0050 (remembering that 80 decimal is 50H), etc.

Things are a little better if we stay within a single row. Since each pixel needs two bits to specify its color, each byte in video memory is divided into 4 2-bit hunks, with each hunk representing a pixel. The two most significant bits are the first pixel (moving from left to right on the screen), the two next most significant bits are the second pixel, etc. The two bits themselves represent one of four colors, but rather than explain the representation of colors now, let us simply assume that two zero bits give black and two one bits give white.

As an example, if the memory at B800:0000 looked like 11000011B, 11110011B, 00110011B, etc., then the pixels in the first row of the screen would be on, off, off, on, on, on, off, on, off, on, off, on, etc. Here is an example DRAW\_DOT macro taking account of this arrangement, but ignoring the color parameter:

```
; MACRO TO SET A PIXEL IN 320X200 MODE. THE FIRST STEP IS TO LOCATE
; WHICH BYTE IN VIDEO MEMORY CONTAINS THE PIXEL. THE SECOND STEP IS
; TO FIND OUT WHICH BITS IN THE BYTE REPRESENT THE PIXEL. THE THIRD
; STEP IS, OF COURSE, TO SET THE PIXEL.
DRAW_DOT MACRO ROW,COLUMN
    LOCAL EVEN
    PUSH ES
    MOV AX,0B800H
    MOV ES,AX
; FIRST, COMPUTE THE OFFSET OF THE BEGINNING OF THE ROW IN VIDEO MEM.
    MOV BX,0          ; ES:BX IS BEGINNING OF VIDEO MEMORY.
    MOV AX,ROW
    SHR AX,1          ; GET EVEN/ODD BIT INTO CARRY.
    JNC EVEN          ; ROW IS EVEN, SO OKAY.
```

```

                MOV  BX,2000H          ; BASE OF ODD ROWS IN VID. MEM.
EVEN:          MOV  CX,80              ; MULTIPLY ROW BY 80.
                MUL  CX
                ADD  BX,AX             ; NOW ES:BX POINTS TO ROW BEGINNING.
; COMPUTE WHICH BYTE IN THE ROW:
                MOV  AX,COLUMN
                SHR  AX,1               ; GET RID OF THE COLOR.
                SHR  AX,1
                ADD  BX,AX             ; ADD ROW-OFFSET TO COL-OFFSET.
; SECOND, COMPUTE NEW COLOR:
                MOV  AX,11000000B      ; SHIFT THIS RIGHT BY THE RIGHT
                MOV  CX,COLUMN         ; NUMBER OF BITS.
                AND  CX,011B
                SHL  CX,1
                SHR  AX,CL
; THIRD, FIX UP THE BYTE IN VIDEO MEMORY:
                OR   ES:[BX],AX        ; SET THE (NOW PROPERLY POSITIONED)
                POP  ES                ; BITS IN VIDEO MEMORY.
                ENDM

```

(The complexity of this scheme explains the comment made earlier that a much faster DRAW\_LINE routine could be written if it could directly access video memory rather than go through this macro. For example, the MUL step could be avoided. Actually, we could conveniently do this multiplication using only shift instructions, but we will skip that for now.)

There are several variations of this scheme which are of interest. One is this: instead of *setting* the bits representing the pixel by ORing them, we could XOR them instead. What is the meaning of this? Recall how the OR and XOR operations work:

| <u>X</u> | <u>Y</u> | <u>X OR Y</u> | <u>X XOR Y</u> |
|----------|----------|---------------|----------------|
| 0        | 0        | 0             | 0              |
| 0        | 1        | 1             | 1              |
| 1        | 0        | 1             | 1              |
| 1        | 1        | 1             | 0              |

We can summarize this by saying that ORing or XORing anything with zero leaves it unchanged. ORing anything with one sets it to one. XORing anything with one complements it. This means, first, that (like the OR operation) a XOR operation would affect *only* the bits representing the pixel. Second, since *most* of the screen is generally blank in graphics applications, the XOR operation would *usually* set the pixel bits (just as OR does). Third, and this is the interesting part, since the XOR operation is inherently reversible, we can *erase* any dots or lines we draw just by redrawing them! I will make this clear with an example. Suppose that we want to set the pixel represented by bits 2 and 3 of the byte 00000000B in video memory. If we XOR this byte with 00001100B (i.e., if we use DRAW\_DOT written with XOR instead of with OR), the byte in video memory becomes 00001100B. If we now repeat this operation (XORing with 00001100B by using DRAW\_DOT), then a glance at the truth table for XOR given above shows that the byte in video memory returns to its original form 00000000B.

XORing is not, however, the only way to erase dots and lines from the display.

Another way to erase dots from the screen would be to allow the use of colors other than white in the DRAW\_DOT macro above. The color white for the dots is hard-coded into the routine by the selection of AX=1100000B just under the comment "SECOND, COMPUTE NEW COLOR". If, instead of ORing with 11B, we provided some means of *setting* the appropriate bits in video memory to 00, 01, or 10, then we could use the other three available colors instead. (Setting the bits to 00 involves simply ANDing them with zero. Setting them to 01B or 10B involves first clearing them by ANDing with 00 and then ORing them with 01B or 10B.) One of these colors, 00, is black -- thus we could erase dots by selecting black as the color of the dots. The difference between this and the XORing scheme is that XORing returns the color of the dot to its *original* color whereas setting the bits to zero simply makes them black regardless of the original color.

Actually, in calling the colors selected by 00B and 11B "black" and "white" I have been guilty of an oversimplification. Since only two bits are available to specify the color of the pixel, only four colors may be used on the screen at any one time. However, there is considerable latitude in selecting *which* four colors those are. For example, every pixel set to 00B will be the same color wherever it appears on the screen -- but that color need not be *black*. Pixels set to 00B are painted the so-called "background color". Recall that we discussed background colors earlier, when covering the attribute byte. There, we found out that the I (intensity), R, G, and B bits together allowed us to select one of 16 colors for the background. These colors are numbered 0-15 as computed using IRGB. (That is, the first column of our earlier table gives the colors 8-15, and the second column give the colors 0-7.) The same situation holds in 320x200 graphics mode: any IRGB color may be selected as the background color (pixel=00B).

Selection of the other three pixel colors is more limited. There are, in fact, only two choices: "palette 0" and "palette 1":

| <u>PALETTE</u> | <u>01B</u> | <u>10B</u> | <u>11B</u> |
|----------------|------------|------------|------------|
| 0              | green      | red        | yellow     |
| 1              | cyan       | magenta    | white      |

(Before we forget it, recall that the BIOS INT 10H function AH=12, which is used to turn on a pixel, had an argument for selecting the color of the dot. At that time, we did not understand colors, so we didn't discuss this argument further. Since only four colors are allowed, the argument must have a value of 1-3 to select the colors described in the palette table above, and a value of 0 to select the background color. In addition, if 128 is added to the value, then the color is XORed with the pixel.)

The background color and the palette can be chosen by using another BIOS INT 10H function, the AH=11 (0BH) function. This function has two subfunctions. These subfunctions are chosen by putting a number into the BH register. If BH=0, we can select the background color by specifying its code (IRGB=0-15) in the BL register. If BH=1, we can select the palette with BL. Here are two examples from Jump's book on DOS:

```
; set background color to light red.
  mov  ah,11          ; function 11.
  mov  bh,0          ; subfunction 0.
  mov  bl,12         ; 12=light red.
  int  10H           ; BIOS video I/O interrupt.

; set palette to 0.
  mov  ah,11          ; function 11.
  mov  bh,1          ; subfunction 1.
  mov  bl,0          ; palette=0.
  int  10H           ; BIOS video I/O interrupt.
```

In the next lecture, we will discuss the 640x200 graphics mode, as well as direct programming of the video hardware.

## FINAL PROJECT #2: INFINITE PRECISION ARITHMETIC

Our normal 8088 arithmetic instructions take arguments of byte or word precision and return answers of a similar precision. Such arithmetic operations have the problem that they can only deal with integers of limited precision and that the operations can "overflow", giving incorrect answers. Infinite precision arithmetic, on the other hand, gets around this problem by allowing the size (in terms of number of bytes) of numbers to vary. To do this, we will assume that all numbers consist of an integral number of words, and the size (in words) of the number must be stored along with the binary representation of the number. For example, in normal binary notation the number 3fcH would be represented by the bytes

```
FC 03
```

(recalling that numbers are stored in memory with the least significant byte first and the most significant byte last). In infinite precision arithmetic, however, a word count would also have to be stored *along with* the representation of the number:

```
01 00 fc 03
```

The leading word whose value is 1 indicates that the number consists of one word. The number 3fc14569H would be expressed similarly as

```
02 00 69 45 c1 3f
```

The word count appended to the beginning of the number is only a word itself, so our arithmetic cannot actually be *infinitely* precise. Indeed, we knew this already from the fact that the memory (and speed) of the computer is limited. Rather, numbers can contain only up to 32K words, or something over 100,000 decimal digits. The 32K limit comes not from the fact that the word count is itself a word, but from the fact that a memory segment can hold only up to 64K bytes or 32K words.

Note that all infinite precision numbers will represent *signed* numbers (as opposed to unsigned numbers).

Rules for manipulating infinite precision numbers (that is, for doing the four elementary operations of +, -, \*, and /) can easily be adapted from our earlier discussions and from the material in chapter 4. The only difference is that a certain amount of memory management must also go on, since the sizes of results of the operations may vary. To handle this, we will store all infinite precision numbers in buffers with the following format (which is slightly augmented from the representation suggested above):

|              |  |
|--------------|--|
| word 0       | contains N, the <i>maximum</i> number of words that can be held by the buffer.               |
| word 1       | the <i>actual</i> number of words held by the buffer.  |
| words 2..N+1 | the buffer itself, the first part of which contains the binary representation of the number. |

This buffer form is very reminiscent of the buffers used by DOS function 10 to read strings from the keyboard. The basic rules for memory management under the four operations are these: if N and M are the *actual* number of words occupied by the two operands (X and Y, respectively), and if L is the maximum-size parameter of the result buffer (with the result being called Z), then the minimum allowable value of L is given by

| OPERATION      | L                         |
|----------------|---------------------------|
| Z=X+Y or Z=X-Y | 1+max{N,M}                |
| Z=X*Y          | N+M                       |
| Z=X/Y          | 0 if M>N, otherwise N-M+1 |
| Z=X mod Y      | min{N,M}                  |
| Z=sqrt(X)      | (N+1) div 2               |

The allocation of these buffers will be the responsibility of a main program which we ourselves will not write. We will write procedures (in our normal way, with FAR PUBLIC PROCs and arguments on the stack) which use such pre-allocated buffers, but which will do the appropriate error checking (according to the above table) to ensure buffers of the correct size.

Your assignment, should you choose to accept it, is to do any three of the following six numbered options:

- 1) Write procedures ADD\_INF and SUB\_INF which (respectively) perform infinite-precision addition and subtraction on two operand buffers, giving a third operand buffer as the result. Calling sequence for Z=X+Y or Z=X-Y: PUSH OFFSET X, PUSH OFFSET Y, PUSH OFFSET Z, CALL ADD\_INF or SUB\_INF. The "offsets" referred to are the offsets of the buffers.
- 2) Write a procedure MUL\_INF to do infinite precision multiplication. The calling sequence is the same as above.
- 3) Write a procedure DIV\_INF to do infinite precision division. This procedure will return both the quotient and remainder of the division. To divide X by Y, giving a quotient Q and remainder R, the calling sequence is PUSH OFFSET X, PUSH OFFSET Y, PUSH OFFSET Q, PUSH OFFSET R, CALL DIV\_INF. If the actual value of the quotient is not desired (that is, if only the remainder is of interest), then the maximum-length parameter of Q should be set to -1 on input. Similarly, if R is not desired, then its maximum-length parameter should be set to -1 on input.
- 4) Write a procedure INP\_INF to read an infinite precision number from the standard input device. That is, to read the ASCII decimal digits and to convert them to number in infinite precision format. The calling sequence (to get input X) is PUSH OFFSET X, CALL INP\_INF.
- 5) Write a procedure OUT\_INF to display the ASCII decimal form of an infinite precision number on the standard output device. The calling sequence (to display X) is PUSH OFFSET X, PUSH OFFSET Y, CALL OUT\_INF. The "Y" here is a buffer at least as large as X which is used as working storage so that OUT\_INF does not need to destroy X as it works.
- 6) Write a procedure SQR\_INF to take the square root of an infinite precision number. The calling sequence is PUSH OFFSET X, PUSH OFFSET Z, CALL SQR\_INF. It is also possible to write a procedure which, like DIV\_INF, returns both a square root and a "remainder" (or, at least, an indication of whether the result was exact or approximate). However, you need not do this.

All of the above (except 5) should (on output) adjust the actual-length parameter of the output buffer(s) to reflect the actual result-size. The only exception to this is if an error occurs. Since the actual-length parameter can vary only from 0 to 32K, the routines can report errors by returning a negative actual length. Here are the errors to be reported:

| <u>ERROR CODE</u> | <u>ERROR DESCRIPTION</u>  |
|-------------------|---|
| -1                | Bad buffer size. This indicates that according to the size rules for results mentioned above, there is not enough room in the output buffer for the result. (Be careful not to report this error for a -1 size specification in DIV_INF.) |
| -2                | Divide by zero error.   |
| -3                | Negative argument of square root.   |
| -4                | Illegal input in INP_INF -- the ASCII characters input do not represent a number. (Note that this is distinct from error -1, which can also occur for INP_INF.)   |
| -5                | Disk full error on OUT_INF.   |

As an example of how to use these routines, let us consider code to multiply 10,000 by 1,000,000,000. Since the first of these numbers (X) is one-word, and the second (Y) is two words, the result is three words and we could have something like this:

```

X   DW  1,1           ; PARAMETERS FOR X BUFFER.
    DW 10000          ; VALUE FOR X BUFFER.
Y   DW  2,2           ; PARAMETERS FOR Y BUFFER.
    DD 1000000000     ; VALUE FOR Y BUFFER.
Z   DW  3,?,3 DUP (?) ; RESULT BUFFER.
...
MOV AX,OFFSET X      ; PUSH FIRST ARGUMENT.
PUSH AX
MOV AX,OFFSET Y      ; PUSH SECOND ARGUMENT.
PUSH AX
MOV AX,OFFSET Z      ; PUSH LOCATION OF RESULT.
PUSH AX
CALL MUL_INF
    
```

Of course, all of these utilities would in practice be controlled by macros. In any case, after this call we would find that Z contained the result (100 trillion).

As with the database project, these functions together form an integrated whole, so we want every function to be attempted by at least one person. The only functions which are (in a sense) dispensable are INP\_INF (since the assembler pseudo-ops DW, DD, and DQ can take its place to a certain extent) and SQR\_INF, which may not be used much.

| <u>MODE</u> | <u>DESCRIPTION</u>      |
|-------------|-------------------------|
| 0           | 40x25 black and white   |
| 1           | 40x25 color             |
| 2           | 80x25 black and white   |
| 3           | 80x25 color             |
| 4           | 320x200 color           |
| 5           | 320x200 black and white |
| 6           | 640x200 black and white |

-----

```
; MACRO TO DRAW A HORIZONTAL LINE.  BY ASSUMPTION, COL1<COL2.
DRAW_HORIZONTAL MACRO ROW, COL1, COL2
```

```
    LOCAL AGAIN
    MOV SI, COL1          ; USE SI TO COUNT COLUMNS.
AGAIN:  DRAW_DOT ROW, SI  ; SET THE PIXEL.
        INC SI           ; NEXT COLUMN.
        CMP SI, COL2     ; DONE?
        JBE AGAIN
    ENDM
```

-----

```
DX := COL2-COL1
DY := ROW2-ROW1
ROW := ROW1
K := DX/2
FOR COL := COL1 TO COL2 DO
BEGIN
    DRAW_DOT ROW, COL
    K := K+DY
    IF K>=DX THEN
    BEGIN
        K := K-DX
        ROW := ROW+1
    END
END
```

-----

```
N EQU NUM_ROWS-1
M EQU NUM_COLS-1
MSG DB "Hello bunky!", 13, 10, '$'
    ...
SM 6
DRAW_LINE 0, 0, 0, M ; LINE AT TOP OF SCREEN.
DRAW_LINE 0, M, N, M ; LINE AT RIGHT.
DRAW_LINE N, M, N, 0 ; LINE AT BOTTOM.
DRAW_LINE N, 0, 0, 0 ; LINE AT LEFT.
CUP 13, 15 ; GOTO CHARACTER POSITION 13, 15.
DISPLAY MSG
GETCHR
SM 3
```

```

draw_dot macro row,column,color
    mov  dx,row          ; select pixel row for BIOS function.
    mov  cx,column      ; select pixel column.
    ifb  <color>
        mov  al,3        ; if color not selected, use white.
    else
        mov  al,color    ; otherwise, use the selected color.
    endif
    mov  ah,12          ; use function AH=12.
    int  10H           ; call BIOS.
endm

```

---

| <u>BIT</u> | <u>DESCRIPTION</u>             |
|------------|--------------------------------|
| 0          | foreground bit B               |
| 1          | foreground bit G               |
| 2          | foreground bit R               |
| 3          | I -- intensity (0=low, 1=high) |
| 4          | background bit B               |
| 5          | background bit G               |
| 6          | background bit R               |
| 7          | BL -- blink (0=no, 1=yes)      |

---

| <u>RGB</u> | <u>COLOR (high intensity)</u> | <u>COLOR (low intensity)</u> |
|------------|-------------------------------|------------------------------|
| 0          | dark gray                     | black                        |
| 1          | light blue                    | blue                         |
| 2          | light green                   | green                        |
| 3          | light cyan                    | cyan                         |
| 4          | light red                     | red                          |
| 5          | light magenta                 | magenta                      |
| 6          | yellow                        | brown                        |
| 7          | white                         | light gray                   |

DRAW\_DOT MACRO

```

; MACRO TO SET A PIXEL IN 320X200 MODE.  THE FIRST STEP IS TO LOCATE
; WHICH BYTE IN VIDEO MEMORY CONTAINS THE PIXEL.  THE SECOND STEP IS
; TO FIND OUT WHICH BITS IN THE BYTE REPRESENT THE PIXEL.  THE THIRD
; STEP IS, OF COURSE, TO SET THE PIXEL.
DRAW_DOT MACRO ROW,COLUMN
    LOCAL EVEN
    PUSH ES
    MOV AX,0B800H
    MOV ES,AX
; FIRST, COMPUTE THE OFFSET OF THE BEGINNING OF THE ROW IN VIDEO MEM.
    MOV BX,0                ; ES:BX IS BEGINNING OF VIDEO MEMORY.
    MOV AX,ROW
    SHR AX,1                ; GET EVEN/ODD BIT INTO CARRY.
    JNC EVEN                ; ROW IS EVEN, SO OKAY.
    MOV BX,2000H            ; BASE OF ODD ROWS IN VID. MEM.
EVEN:    MOV CX,80           ; MULTIPLY ROW BY 80.
    MUL CX
    ADD BX,AX                ; NOW ES:BX POINTS TO ROW BEGINNING.
; COMPUTE WHICH BYTE IN THE ROW:
    MOV AX,COLUMN
    SHR AX,1                ; GET RID OF THE COLOR.
    SHR AX,1
    ADD BX,AX                ; ADD ROW-OFFSET TO COL-OFFSET.
; SECOND, COMPUTE NEW COLOR:
    MOV AX,11000000B        ; SHIFT THIS RIGHT BY THE RIGHT
    MOV CX,COLUMN            ; NUMBER OF BITS.
    AND CX,011B
    SHL CX,1
    SHR AX,CL
; THIRD, FIX UP THE BYTE IN VIDEO MEMORY:
    OR ES:[BX],AX           ; SET THE (NOW PROPERLY POSITIONED)
    POP ES                  ; BITS IN VIDEO MEMORY.
    ENDM

```

---

| <u>PALETTE</u> | <u>01B</u> | <u>10B</u> | <u>11B</u> |
|----------------|------------|------------|------------|
| 0              | green      | red        | yellow     |
| 1              | cyan       | magenta    | white      |

LECTURE NOTES FOR CS-5330: IBM PC ASSEMBLY LANGUAGE  
UNIVERSITY OF TEXAS AT DALLAS

## CLASS 18

CommentsReview

In the previous lecture we began discussing high-resolution graphics (or, at least, what passes for high-resolution graphics) on the IBM PC using the standard color adapter board.

We discovered that the computer's display consists of a rectangular grid of dots known as "pixels". Normally, the computer is in "text" mode, capable of displaying 25 rows of 80 characters each (80x25). To do high-resolution graphics, however, we must put the computer into "graphics" mode -- typically 640x200 (black and white) or 320x200 (4 color) mode. These modes are selected, respectively, by the ANSI screen driver commands "SM 3", "SM 4", and "SM 6". Once in graphics mode, all computer graphics operations can be reduced to repeated applications of the primitive operation of setting a pixel to a certain color (such as black or white). Therefore, we introduced a macro

```
DRAW_DOT      ROW,COLUMN [,COLOR]
```

to perform this operation. (The COLOR argument, if omitted, defaults to the lightest available color.) The dot-setting operation is the only hardware-dependent operation we need, and all other graphics operations can then be built out of DRAW\_DOTs in a hardware-independent way. For example, we saw the hardware-independent macros

```
DRAW_HORIZONTAL  ROW,COL1,COL2
DRAW_LINE        ROW1,COL1,ROW2,COL2 [,COLOR]
```

which, respectively, drew lines from the points (ROW,COL1) to (ROW,COL2) and from (ROW1,COL1) to (ROW2,COL2).

We also discussed two implementations of the DRAW\_DOT macro. The first and simplest implementation simply involved a call to the BIOS interrupt 10H. One of the 10H functions, the AH=12 function, allows us to set an arbitrary pixel to a chosen color, which is exactly what we need. Unfortunately, there is so much overhead (in terms of execution time) to execute a BIOS function that the DRAW\_LINE macro might execute unacceptably slowly in many applications.

To overcome this speed problem, we needed a second and more complex implementation. This second DRAW\_DOT macro needed to directly access the video display, rather than go through BIOS. This we were able to do by accessing the computer's "video memory". All information in the image on the screen comes (more or less directly) from a region of the computer's memory known as the *video memory*. For the IBM monochrome display card, which has 4K of memory on board at address B000:0000, only text can be displayed. 2000 characters can be shown on the screen (80\*25=2000), and each character has an "attribute byte"

giving its intensity, whether it is blinking, etc. Thus, the text alone uses up the entire video memory for the monochrome board.

The IBM color adapter board has 16K at address B800:0000, and so can contain 4 text pages. In graphics mode, however, both the 640x200 mode ( $=640 \times 200 = 128000$  bits) and the 320x200 color ( $=320 \times 200 \times 2 = 128000$  bits) use up all available memory on the graphics board. We have not yet discussed the 640x200 mode. In the 320x200 mode, we explored the relationship between the pixels and the video memory. Basically, each row of pixels is specified by 80 consecutive bytes of video memory. The even numbered rows of pixels are at offsets 0-1FFFH, while the odd numbered rows are at offsets 2000H-3FFFH. Each byte specifies the states of 4 pixels. Bits 6,7 are the leftmost pixel, bits 4,5 are the next pixel, bits 2,3 are the next after that, and bits 0,1 are the rightmost pixel.

The two bits for each pixel represent its *color*. If the bits are 00, the color is the "background color". The other three combinations are one of three colors from the currently selected "palette". The background color and the palette can be selected by the programs, but they hold throughout the entire screen -- i.e., only four colors can be displayed at any time, but there is a certain latitude in selecting the particular four colors to be displayed. There are 16 possible choices for the background color and two possible choices for the palette, but it does not seem necessary to repeat all of the detailed options at this point. The background color and the palette are selected using the AH=11 function of INT 10H.

#### The 640x200 Graphics Mode

Dot-addressable graphics in the 640x200 mode is very similar to that in the 320x200 color mode. In fact, the only difference is that instead of each byte in video memory representing 4 pixels of 4 possible colors, the bytes represent 8 pixels of 2 possible colors (i.e., visible and invisible). Thus, for example, bit 7 of the byte at address B800:0000 is the dot in the upper left-hand corner of the screen. Bit 6 is the dot to the right of that, etc. Only minor modifications therefore need to be made to the DRAW\_DOT macro developed in the previous lecture:

```
; MACRO TO SET A PIXEL IN 640X200 MODE. THE FIRST STEP IS TO LOCATE
; WHICH BYTE IN VIDEO MEMORY CONTAINS THE PIXEL. THE SECOND STEP IS
; TO FIND OUT WHICH BITS IN THE BYTE REPRESENT THE PIXEL. THE THIRD
; STEP IS, OF COURSE, TO SET THE PIXEL.
DRAW_DOT MACRO ROW,COLUMN
    LOCAL EVEN
    PUSH ES
    MOV AX,0B800H
    MOV ES,AX
; FIRST, COMPUTE THE OFFSET OF THE BEGINNING OF THE ROW IN VIDEO MEM.
    MOV BX,0          ; ES:BX IS BEGINNING OF VIDEO MEMORY.
    MOV AX,ROW
    SHR AX,1          ; GET EVEN/ODD BIT INTO CARRY.
    JNC EVEN          ; ROW IS EVEN, SO OKAY.
    MOV BX,2000H      ; BASE OF ODD ROWS IN VID. MEM.
EVEN:    MOV CX,80     ; MULTIPLY ROW BY 80.
    MUL CX
```

```

        ADD  BX,AX          ; NOW ES:BX POINTS TO ROW BEGINNING.
; COMPUTE WHICH BYTE IN THE ROW:
        MOV  AX,COLUMN
        SHR  AX,1          ; 8 BITS PER BYTE.
        SHR  AX,1
; *****
        SHR  AX,1
        ADD  BX,AX        ; ADD ROW-OFFSET TO COL-OFFSET.
; SECOND, COMPUTE NEW COLOR:
        MOV  AX,1000000B  ; SHIFT THIS RIGHT BY THE RIGHT
        MOV  CX,COLUMN    ; NUMBER OF BITS.
        AND  CX,0111B
; *****
        SHR  AX,CL
; THIRD, FIX UP THE BYTE IN VIDEO MEMORY:
        OR   ES:[BX],AX   ; SET THE (NOW PROPERLY POSITIONED)
        POP  ES          ; BITS IN VIDEO MEMORY.
        ENDM

```

In this macro, the only differences from our 320x200 macro are in three of the lines in the area delimited by rows of asterisks. Apparently, although the documentation on this is rather sleazy, the background color in 640x200 mode is *always* black. However, the foreground color (the color of the dots) can be selected to be any of the possible 16 colors. To do this, we use BIOS (as in the previous lecture) and pretend that we are setting the *background* color. That is, we use BIOS INT 10H, function AH=11, subfunction BH=0.

Fortunately, we need not continue and develop more algorithms for dot-addressable graphics, since one of our class members is developing (as a final project) a set of macros for graphics programming.

#### User-definable Character Sets

I alluded earlier to the fact that with the IBM PC some of the displayable characters can be *programmed* so that they assume a different shape. The standard ASCII characters cannot be changed in this way, but the characters in the range 128-255 can be so altered. Unfortunately, these re-programmed characters can only be seen in graphics mode and not in text mode (where we would probably want them). Nevertheless, understanding how 640x200 graphics works, we are now in a position to understand the programming of the characters.

Here are the various steps involved in using the programmable character set:

- 1) Set up the table of character definitions.
- 2) Set the address at interrupt vector 1FH to point to the new table.
- 3) Put the computer in graphics mode.
- 4) Use the characters (codes 128-255).

Of course, the normal ASCII characters (0-127) are still available regardless of whatever changes we make.

Let us take these steps in order. First, how do we set up a table of character definitions? To define the shape of the character requires 8 contiguous bytes in memory. There are 128 user-definable characters, so (in all) the shape table has 128\*8=1024 bytes. Each of the bytes of a character definition sets up one row of pixels of the character. Thus, each character consists of 8 rows (8 bytes) of 8 columns (8 bits per byte). As with 640x200 graphics, the leftmost pixels of the character correspond to bit 7, the next pixel to bit 6, etc. To take an example from Jump, here is a definition of the greek letter *pi*:

|                         | <u>7</u> | <u>6</u> | <u>5</u> | <u>BITS</u> |          |          |          |          |       |
|-------------------------|----------|----------|----------|-------------|----------|----------|----------|----------|-------|
|                         |          |          |          | <u>4</u>    | <u>3</u> | <u>2</u> | <u>1</u> | <u>0</u> |       |
| <u>0</u>                | 0        | 0        | 0        | 0           | 0        | 0        | 0        | 0        | = 00H |
| <u>1</u>                | 0        | 1        | 1        | 1           | 1        | 1        | 1        | 0        | = 7EH |
| <u>2</u>                | 1        | 0        | 1        | 0           | 0        | 1        | 0        | 0        | = A4H |
| <u>3</u>                | 0        | 0        | 1        | 0           | 0        | 1        | 0        | 0        | = 24H |
| <u>BYTE</u><br><u>4</u> | 0        | 0        | 1        | 0           | 0        | 1        | 0        | 0        | = 24H |
| <u>5</u>                | 0        | 0        | 1        | 0           | 0        | 1        | 0        | 0        | = 24H |
| <u>6</u>                | 0        | 0        | 1        | 0           | 0        | 1        | 0        | 0        | = 24H |
| <u>7</u>                | 0        | 0        | 0        | 0           | 0        | 0        | 0        | 0        | = 00H |

Thus, assuming that we had *only* this special character to define, our character definition table might look like:

```
PI  DB  00H,7EH,0A4H,024H,024H,024H,024H,00H
```

(Of course, in practice, we would define many more characters than just this, or else it's not worth the trouble.)

The next step, changing the interrupt vector table so that it contains the address of the new character shape table, is best accomplished using a DOS function. DOS function 25H sets a given interrupt vector to a desired value. To use it, we simply set up DS:DX to contain the new address to be used as the interrupt vector, and load AL with the interrupt number. In our case, the new table is probably already in the data segment, so all we need to do is

```
; CHANGE INTERRUPT VECTOR 1FH:
  MOV  AH,25H          ; DOS SET-INTERRUPT FUNCTION.
  MOV  DX,OFFSET PI   ; NEW CHARACTER TABLE ADDRESS.
  MOV  AL,1FH         ; CHANGE INTERRUPT VECTOR 1FH.
  INT  21H
```

Here is a short program that combines all of the steps: it sets up a new character table containing *pi*, then goes into graphics mode, fills the screen with *pis*, waits for a character to be typed at the

keyboard, returns to text mode, resets the character table interrupt vector to the default (0:0), and quits:

```

I    DW    ?
PI   DB    00H,7EH,0A4H,024H,024H,024H,024H,00H
    ...
; CHANGE INTERRUPT VECTOR 1FH:
    MOV    AH,25H                ; DOS SET-INTERRUPT FUNCTION.
    MOV    DX,OFFSET PI          ; NEW CHARACTER TABLE ADDRESS.
    MOV    AL,1FH                ; CHANGE INTERRUPT VECTOR 1FH.
    INT    21H
; GRAPHICS MODE.
    SM    6
; PRINT 2000 PIs:
    FOR    I,1,2000
        PUTCHR 128
    ENDFOR I
    GETCHR
; TEXT MODE.
    SM    3
; CHANGE INTERRUPT VECTOR 1FH:
    MOV    AH,25H                ; DOS SET-INTERRUPT FUNCTION.
    PUSH   DS
    MOV    DX,0                  ; NEW CHARACTER TABLE ADDRESS (0:0).
    MOV    DS,DX
    MOV    AL,1FH                ; CHANGE INTERRUPT VECTOR 1FH.
    INT    21H
    POP    DS

```

#### Direct Programming of the CRT Controller Chip

As it turns out, there are additional (though relatively minor) features of the video hardware which you cannot access even with BIOS-- but which you can access by directly programming the hardware.

As was mentioned in the very first lecture, the 8088 CPU actually has *two* address spaces. One address space contains the memory, and is used by almost all of the 8088 instructions. The other address space contains the "I/O ports" and is addressed by only a few instructions -- in particular, the IN and OUT instructions. The I/O address space is only 64K in size but, like the memory space, contains both bytes and words. (However, for the IBM PC, usually only *byte* I/O is performed.) Typically, I/O devices are interfaced with the computer in such a way that the CPU can communicate with them by writing information to certain I/O ports and reading information from other ports. This is not universally true -- the video memory is a counter-example -- but it is the normal case.

Here is the syntax for the IN and OUT instructions:

```

IN    accumulator,port        ; GET A BYTE OR WORD FROM THE
                                ; PORT SPECIFIED.
OUT   port,accumulator        ; WRITE A BYTE OR WORD TO THE
                                ; SPECIFIED PORT.

```

Here, the accumulator is either the AL register (for byte values) or the AX register (for word values). The port operand gives the address

of the I/O port. This is slightly tricky. The port operand must be either an *immediate* value in the range 0-255, or else it must be the DX register. If the port operand is the DX register, then the DX register holds the *address* of the port. Here are some examples:

```
; WRITE THE BYTE VALUE 77 TO I/O PORT 44:
    MOV  AL,77
    OUT  44,AL
; WRITE THE BYTE VALUE 88 TO I/O PORT 3E1H:
    MOV  AL,88
    MOV  DX,3E1H
    OUT  DX,AL
; READ WORD VALUE FROM I/O PORT 44:
    IN   AX,44
; READ WORD VALUE FROM I/O PORT 3E1H:
    MOV  DX,3E1H
    IN   AX,DX
```

As mentioned, however, almost all I/O ports *actually* used with the IBM PC are byte ports, so we would always use AL rather than AX with our IN and OUT instructions. It is a little inconvenient to continually use the accumulator and DX registers like this, but we can compensate a little by introducing some macros to take care of the work:

```
; MACRO TO WRITE A BYTE VALUE TO A PORT:
OUT_PORT MACRO      ADDRESS,VALUE
    MOV             DX,ADDRESS      ; ADDRESS PORT WITH DX.
    MOV             AL,VALUE
    OUT             DX,AL
    ENDM
```

Here is a partial list of some of the I/O ports typically used in an IBM PC, taken mostly from Sargent and Shoemaker, *The IBM Personal Computer from the Inside Out*. All addresses are in hex:

| <u>Port Addresses</u> | <u>Used by</u>                           |
|-----------------------|--|
| 0-1F                  | 8237 4-channel DMA controller            |
| 20-3F                 | 8259 8-channel interrupt controller      |
| 40-5F                 | 8253 3-channel counter/timer circuit     |
| 60-7F                 | 8255 24-line parallel I/O interface      |
| 80-9F                 | DMA 64K page register                    |
| A0-BF                 | NMI mask bit latch                       |
| C0-C7                 | PCjr sound generator                     |
| C8-EF                 | Reserved                                 |
| F0-FF                 | PCjr floppy diskette interface           |
| 100-1FF               | Not usable                               |
| 200-20F               | Game I/O adapter                         |
| 210-217               | Expansion unit                           |
| 220-24F               | Reserved                                 |
| 250-277               | Not used                                 |
| 278-27F               | Second parallel printer interface (LPT2) |
| 280-2EF               | Not used                                 |
| 2F0-2F7               | Reserved                                 |
| 2F8-2FF               | Second serial interface (COM2)           |
| 300-31F               | Prototype card                           |
| 320-32F               | Hard disk                                |
| 330-377               | Not used                                 |

|         |   |
|---------|---|
| 378-37F | First parallel printer interface (LPT1) |
| 380-38C | SDLC or second binary synchronous port  |
| 390-39F | Not used                                |
| 3A0-3A9 | Primary binary synchronous interface    |
| 3B0-3BF | Monochrome display                      |
| 3C0-3CF | Reserved                                |
| 3D0-3DF | Color/graphics display adaptor          |
| 3E0-3EF | Reserved                                |
| 3F0-3F7 | 5-1/4" floppy disk drive controller     |
| 3F8-3FF | First serial interface (COM1)           |

There are many common additions to the PC that use additional ports. For example, the clock/calendar on the AST 6-Pak Plus card uses ports 2C0-2DF, which are marked "not used" in the table above.

The video display, as you may have surmised from what was said above, is accessed partly through memory and partly through I/O ports. Naturally, the video memory itself is entirely contained in memory. However, there is a hardware device -- the 6845 CRT controller chip (by Motorola, presumably) -- which can be addressed via I/O ports instead. The 6845 takes care of most of the electronic and software dirty work of running the display screen, leaving the PC with the relatively simple job of merely interfacing with the 6845 rather than controlling the screen itself. There are also a few other ports which, while not connected to the 6845 itself, help to control some features of the video display.

Although we will not discuss all of these, for the sake of reference here is a list of the ports used for video I/O:

| <u>Video Port</u> | <u>Description</u>                    |
|-------------------|---------------------------------------|
| 3D4               | Register selection port for the 6845. |
| 3D5               | 6845 registers.                       |
| 3D8               | Display mode control.                 |
| 3D9               | Color select.                         |
| 3DA               | Status/Mode I/O                       |
| 3DB               | Light pen latch CLEAR                 |
| 3DC               | Light pen latch PRESET                |

These port-address assignments hold for the color adapter card. For the monochrome card, the relevant ports are 3BnH rather than 3DnH. I will assume, without warning from now on, that the color adapter card is being used. Many of the BIOS 10H functions do little more than write to these I/O ports, or read from them. Nevertheless, there are a few features of the video display that BIOS does not tap. We will confine our attention to the two 6845 ports, 3D0H and 3D1H, and to the status port 3DBH.

The 6845 video controller chip has 19 internal registers, all of which can be accessed by the PC. One of these registers, the *address register*, is directly available as port 3D4H. All of the other registers are located at port 3D5H. In order to use any 6845 register *other* than the address register, it is first necessary to put the its number (0-17) into the address register, and *then* to access it with IN and OUT instructions to port 3D5H. This will become clearer in a moment with a programming example.

The 18 6845 registers other than the address register are referred to as R0-R17. Of these, R0-R9 have to do mostly with the electrical and timing relationships of various parts of the video signal and should be left alone. Of the remaining registers, only R10-R15 have any utility beyond that already provided by BIOS.

R12-R15 provide a more flexible means of controlling the display page than is given with BIOS. With BIOS, a display page always begins at a 4K boundary in video memory, so there are exactly 4 available pages. Using R12-R15, however, we can begin a display page at any address in video memory. This is very useful not only for paging, but for scrolling as well. Normally, we are in page 0 of the display, which begins at offset 0 in video memory. If we changed the page to begin at offset 80 (note that the "addresses" used by the 6845 ignores the existence of the attribute byte -- thus, the 8088 thinks the page begins at address 160, but the 6845 thinks it begins at 80), then the we would have the effect that the screen scrolled upward by one line. However, this scrolling is *extremely* fast. More interestingly, if we changed the page to begin at offset 1, the display would scroll right by one character.

Registers R12-R13 hold the more significant byte and the less significant byte of the address of the beginning of the active display page. Here is how we would program the "scroll right" function mentioned above:

```
; MACRO TO SEND A BYTE VALUE TO A GIVEN 6845 REGISTER.
OUT_6845 MACRO REGISTER,VALUE
    OUT_PORT 3D4H,REGISTER ; FIRST, SELECT DESIRED REGISTER.
    OUT_PORT 3D5H,VALUE ; THEN DO THE OUTPUT.
ENDM

; CODE TO PROGRAM THE 6845 PAGE SELECT REGISTERS WITH AN ADDRESS OF
; 1. R12=MORE SIGNIFICANT BYTE=0, AND R12=LESS SIGNIFICANT BYTE=1.
; THE ADDRESS WHICH THE 6845 THINKS OF AS 1 IS 2 TO THE 8088.
    OUT_6845 12,0 ; R12=0.
    OUT_6845 13,1 ; R13=1.
```

Registers R14-R15 give the high and low bytes of the cursor address, but otherwise they differ from R12-R13 only in that they can be *read* as well as *written*. To move the cursor to the beginning of the page we have just selected, we would do something like:

```
; MOVE CURSOR TO VIDEO-MEMORY LOCATION 1:
    OUT_6845 14,0
    OUT_6845 15,1
```

Registers R10-R11 control the cursor size, but give us more control than the corresponding BIOS function. With these registers we can change the cursor shape, but we can also completely eliminate the cursor. R10 gives the starting cursor line, and R11 gives the ending line. Recall that these lines are numbered from 0-14. The lowest 5 bits (0-4) of the registers are devoted to specifying the starting and ending lines, while bits 5 and 6 of R10 are used to control the blink-rate:

| <u>BIT 5</u> | <u>BIT 6</u> | <u>CURSOR</u> |
|--------------|--------------|---------------|
| 0            | 0            | not blinking  |
| 0            | 1            | not displayed |
| 1            | 0            | fast blink    |
| 1            | 1            | slow blink    |

For a color display, the cursor normally occupies lines 6 and 7. Thus, to keep the cursor the same shape as it normally is, but to turn off displaying it, we might do this:

```
OUT_6845 10,0101010B
```

To make the cursor a non-blinking block, we could do

```
OUT_6845 10,0000000B
```

In these examples, the blink rate bits are italicized (bit 7 is omitted), while the starting line bits are in normal type. On many computers (including the IBM PC), the external hardware overrides the blink-speed control (or non-blink control), so that only options 00 and 01 are valid, with the cursor *always* blinking if displayed. Turning the cursor off can be very useful in using full-screen displays in which various parts of the screen must be updated quickly, since we don't have to watch the cursor annoyingly flit across the screen.

The other generally useful I/O port is the status port 3DAH. To understand the value of the information available at this port, we must understand something about the image displayed on the screen. In the first place, recall that the image is controlled by the contents of video memory, and that video memory is accessible to both the 8088 and to the 6845. Electrically, it is impossible for both devices to access the memory *simultaneously*, yet by the rules of chance this is bound to happen occasionally since the 8088 and 6845 operate independently and simultaneously. When this happens, the 8088 accesses the video memory and the 6845 is simply denied access. This means that there are sometimes brief intervals of time in which the information displayed at some spot on the screen is incorrect (since the 6845 has been denied access to video memory). This can result in an annoying flicker of the display while the 8088 is directly accessing video memory. (For instance, when the IBM color adapter scrolls the display, it completely rewrites video memory; to avoid the flicker just mentioned, the adapter turns the entire display *off*, rewrites the screen, and then turns the display on again. This, of course, is hardly less annoying than any flicker would be.)

However, there are also long intervals of time in which the 6845 is not accessing the video memory at all. These intervals are the horizontal and vertical retrace times. We can think of the display as being drawn by a tiny pixel-sized cursor. This cursor starts at the upper left-hand side of the screen. First, it moves rightward, drawing the first row of pixels. Then, it moves downward and draws the second line. When it finally gets to the bottom of the screen, it then moves back to the top of the screen and starts over. The time during which it is moving from the end of one row to the beginning of the next (and is therefore not reading any pixel values from memory) is the horizontal retrace time. The vertical retrace time, of course, occurs while the cursor is moving from the bottom of the screen to the top.

Since the 6845 is not accessing the video memory during these times, if the 8088 could confine its use of the video memory to just these times, then there would be no flicker.

Port 3DAH is an input port which, among other things, can be used to determine if a horizontal or vertical retrace is in progress:

```

BIT 0          is one if either type of retrace is in progress.
BIT 3          is one if a vertical retrace is in progress.

```

The point of distinguishing between the two types of retraces is that the vertical retrace takes much longer to complete (about 1.25 milliseconds) than the horizontal retrace (about 10 microseconds) and therefore more memory accesses can occur during it. Here is a sample program which branches, depending on whether a retrace is occurring:

```

MOV  DX,3DAH   ; GET THE STATUS BYTE.
IN   AL,DX
TEST AL,100B   ; VERTICAL RETRACE?
JNZ  VERTICAL
TEST AL,1      ; HORIZONTAL RETRACE?
JNZ  HORIZONTAL
NORETRACE:

```

This concludes our in-class discussion of the video hardware.

#### Introduction to the Serial Port Hardware

If you are not interested in serial communications (or know nothing about it) you can close your ears at this point.

The IBM PC can be equipped with 0, 1, or 2 *serial ports* (or more under some circumstances), which are used to perform serial I/O. The serial ports implement the so-called RS-232 standard and can be used to connect the IBM PC to a large number of external devices like plotters, some kinds of printers, digitizers, modems, terminals, etc. Unfortunately, in most instances, using the IBM PC serial ports requires direct programming of the hardware, since the BIOS is really not up to the job of generally handling serial I/O.

In the IBM PC, most of the dirty work for serial I/O is handled by a device known as a *UART* -- for Universal Asynchronous Receiver Transmitter. The particular UART used in an IBM PC is the 8250 single-chip UART manufactured by Intel. Like the video display hardware, the serial I/O hardware has a selection of I/O ports in the 8088's I/O address space assigned to it. By the way, we have a slight conflict in terminology here. Addresses in the 8088's I/O space are referred to as "ports"; however, an RS-232 interface is also referred to as a serial "port". Hopefully this will not be too confusing. On an IBM PC, it is typical to call the first RS-232 interface "COM1", and the second "COM2". We will adopt this practice. Here are the relevant ports:

| <u>I/O Port</u> | <u>Used for</u> | <u>Description</u>  |
|-----------------|-----------------|---|
| 3F8H            | output          | a) transmitter holding register<br>b) baud-rate divisor (LSB) |
| 3F8H            | input           | receiver data register  |
| 3F9H            | output          | a) interrupt-enable register<br>b) baud-rate divisor (MSB)    |
| 3FAH            | input           | interrupt-identification reg.                                 |
| 3FBH            | output          | line-control register   |
| 3FCH            | output          | modem-control register  |
| 3FDH            | input           | line-status register  |
| 3FEH            | input           | modem-status register   |

These are the port addresses for COM1, while the port addresses for COM2 are the same except they are of the form 2FnH rather than 3FnH.

For today, we will ignore the apparent complexities of serial I/O and concentrate on just the simplest aspects. Almost all of the ports in the table above are used for initializing the serial interface. For the moment, let us suppose that the serial interface has been correctly initialized at some previous time, and that we simply want to do some I/O.

The ports involved in simple I/O are 3F8H and 3FDH. On output, register 3F8H is used to output data through the serial interface to some external device. In input, it is used to read data sent by the external device to the serial interface. In order to determine if the UART is ready to send more data, or to determine if the UART contains data that should be read, the status register 3FDH is used. The status register is used not only for this, but also to indicate various transmission or reception errors. The bits read from the status register are interpreted as follows:

| <u>BIT</u> | <u>DESCRIPTION</u>   |
|------------|--|
| 0          | =1 if an input byte is ready to read.  |
| 1          | =1 if you didn't manage to read the character before it was overwritten by the next one. |
| 2          | =1 if there is a parity error.   |
| 3          | =1 if there is a baud rate error.  |
| 4          | =1 if a break character has been received.   |
| 5          | =1 if free to send another character.  |
| 6          | =1 if all data has been completely transmitted.  |

Thus, to send a character out through the serial interface, we would do something like

```
OUT_PORT 3F8H, character
```

Similarly, to receive a character, we would do something like

```
; MACRO TO GET A VALUE FROM A PORT:
IN_PORT  MACRO  VALUE, PORT
          MOV    DX, PORT          ; SET UP PORT ADDRESS.
          IN     AL, DX            ; GET THE BYTE.
          MOV    VALUE, AL        ; AND SAVE IT.
          ENDM
          ...
IN_PORT  character, 3F8H
```

However, we normally wouldn't do these things without checking the line-status first to determine if the UART was ready for these operations. Thus, we would probably want macros like

```
; MACRO TO DETERMINE IF THE UART IS READY TO SEND A CHARACTER: JUMP
; TO THE SPECIFIED ADDRESS IF THE UART IS BUSY.
OUT_STAT MACRO NOTREADY
    IN_PORT AL,3FDH ; GET THE STATUS BYTE.
    TEST AL,32 ; TRANSMITTER HOLDING REG. EMPTY?
    JZ NOTREADY ; IF NO, JUMP
ENDM

; MACRO TO DETERMINE IF THE UART HAS VALID INPUT TO BE READ. IF NOT,
; JUMP TO THE SPECIFIED ADDRESS.
IN_STAT MACRO NOTREADY
    IN_PORT AL,3FDH ; GET THE STATUS BYTE.
    TEST AL,1 ; DATA READY?
    JZ NOTREADY ; IF NO, JUMP
ENDM
```

As a sample application, let's put all of these ingredients together to form a "dumb terminal" program. A dumb terminal is a device that accepts input from either the keyboard or the RS-232 interface. Both types of input are displayed on the screen, but input from the keyboard is sent out over the RS-232 as well. Here, in pseudo-code, is the algorithm for a dumb terminal:

```
REPEAT
    IF CHARACTER READY AT KEYBOARD THEN
        BEGIN
            GET THE CHARACTER FROM THE KEYBOARD
            DISPLAY IT ON THE SCREEN
            WAIT UNTIL RS-232 IS NOT BUSY
            SEND IT OUT OVER THE RS-232
        END
    IF CHARACTER READY AT RS-232 THEN
        BEGIN
            GET THE CHARACTER FROM THE RS-232
            DISPLAY IT ON THE SCREEN
        END
    END
FOREVER
```

Given the macros we have already developed (and recalling that DOS function 0BH can be used to check if a keyboard character is ready), this program is very easy to write:

```
; KEYBOARD INPUT PART.
KEYBOARD: MOV AH,0BH ; CHECK KEYBOARD STATUS
            INT 21H ; BY CALLING DOS.
            OR AL,AL ; AL=0?
            JZ RS232 ; IF YES, THEN NO CHARACTER IS READY.
            GETCHR CL ; GET A KEYBOARD CHARACTER.
            PUTCHR CL ; DISPLAY IT ON THE SCREEN.
NOTREADY: OUT_STAT NOTREADY ; WAIT UNTIL RS-232 IS READY TO XMIT.
            OUT_PORT 3F8H,CL ; SEND CHARACTER TO RS-232.
```

```
; RS-232 INPUT PART.
RS232:  IN_STAT KEYBOARD      ; IF NO RS232 CHARACTER, GO BACK TO KBD.
        IN_PORT CL,3F8H      ; GET THE RS232 CHARACTER.
        PUTCHR CL            ; DISPLAY IT.
        JMP  KEYBOARD
```

In practice, however, there is much more to setting up the serial port than I have indicated, as we will see in the next lecture.

#### REFERENCES:

A very good reference for hardware related topics such as those we have been discussing is

*8088 Assembler Language Programming: The IBM PC*, by David Willen and Jeffrey Krantz.

This book covers most of the material we have discussed today, but is particularly good on the subject of serial I/O. Another reasonably good general reference with a hardware bias is

*The IBM Personal Computer from the Inside Out*, by Murray Sargent and Richard Shoemaker.

For those with a fascination for the hardware alone,

*Interfacing to the IBM Personal Computer*, by Lewis Eggebrecht is an interesting reference, particularly if you are designing or prototyping cards to be used in an IBM PC (or clone).

*This information was given to me by J. Midgley. I have not tried out this information, nor do I know its original source.*

TI PC BIOS VIDEO I/O INTERRUPT 49H

| <u>FUNCTION</u> | <u>DESCRIPTION</u>   |
|-----------------|--|
| AH= 1H          | Set cursor type. CX=0bb0ssss0000eeee, where bb=blink mode, ssss=start line, and eeee=end line.   |
| AH= 2H          | Set cursor position DH=column (0-79) and DL=row (0-24).  |
| AH= 3H          | Read cursor status. Returns CX and DX (as above?).   |
| AH= 6H          | Scroll or move text block. DX=upper left hand corner of the block.<br>BX=destination for move. CH=width. CL=length. AL=0 for move. AL<>0 for copy.       |
| AH= 8H          | Get character at cursor. AL=character. AH=attribute.   |
| AH= 9H          | Write character and attribute at cursor. AL=character, BL=attribute.<br>CX=number of copies of the character to make.                                    |
| AH=0AH          | Write character with same attribute as last time. AL=character. CX=number of copies.   |
| AH=0EH          | Write character and advance cursor (executing control characters).<br>AL=character.  |
| AH=10H          | Write block at cursor. DX:BX(?)=address of block. CX=length of block.<br>AL=attribute.   |
| AH=11H          | Write block at cursor using attribute latch. DX:BX(?)=address of block.<br>CX=length of block.   |
| AH=12H          | Change attribute of entire screen to AL.   |
| AH=13H          | Clear screen and home cursor.  |
| AH=14H          | Clear graphics screen.   |
| AH=15H          | Set status region to begin at line CX. CX must be greater than the cursor line. Disable if CX=0.   |
| AH=16H          | Set attribute latch to BL=abureGRB. The attributes are a=alt set, b=blink, u=underline, r=reverse video, e=enable, GRB=(presumably) the color.           |
| AH=17H          | Get offset of first character (on screen?) into DX. The segment is always 0DE00H.  |
| AH=18H          | Print string pointed to by BX. The first byte of the string must be the character count. Advance the cursor and execute control characters as necessary. |

TI VIDEO MEMORY PROGRAMMING (3-PLANE GRAPHICS)

The 720x300 pixels on the screen are each controlled by a three bit number. The MSB is in a memory segment starting at C800:0000, the next is in a memory segment starting at D000:0000, and the LSB is in a memory segment starting at C000:0000. Three eight-bit registers map this number to the possible combinations of the three colors that can appear on the screen. The green latch is at DF02:0000, the red latch is at DF03:0000, and the blue latch is at DF01:0000. For example, if you wished the number 7 to correspond to yellow on the screen, you would set bit 7 in the green and red latches and clear bit 7 in the blue latch. Or, if you wished a blue background, you would set bit 0 in the blue latch (and, of course, clear bit 0 in the others). The standard setting of the latches is DF02:0000=F0, DF03:0000=CC, and DF01:0000=AA, because this makes each of the bits in the control number correspond to Green, Red, and Blue, respectively.

Just as AH=13H INT 49H clears the alpha screen, AH=14H clears the graphics screen. Or, you may make graphics vanish by clearing the three latches. If not cleared or disabled, graphics and alphanumeric will appear simultaneously.

The three planes of video memory are divided into words. Each word controls 16 pixels, with the most significant bit being the leftmost pixel, and the least significant bit the rightmost pixel. The first row of pixels is controlled by the words at offsets 0, 2, ... ,88 (decimal), the second row of pixels by 92, 94, ... , 180 (decimal), etc. Thus, each row takes 92 bytes, of which 90 are used, giving 90\*8=720 pixels per row. The words of the very last row on the screen are at offsets 27508, 27510, up to 27596 (decimal).

LECTURE NOTES FOR CS-5330: IBM PC ASSEMBLY LANGUAGE  
UNIVERSITY OF TEXAS AT DALLAS

## CLASS 19

CommentsReview

In the previous lecture we completed our discussion of dot-addressable graphics and began discussing hardware-oriented programming.

We discussed the 640x200 black and white graphics display mode, finding that it differed very little from the 320x200 color mode.

We discussed user-programmable character sets. Recall that while the standard ASCII characters simply have fixed appearances, the characters 128-255 have shapes defined by a "shape table" that can be altered by our programs. The steps necessary in defining these characters were: a) set up a new shape table; b) change interrupt vector 1FH to point to the new table; c) get into graphics display mode (the definable characters only appear in graphics mode); and d) use the characters. The shape table was very much like the 640x200 video memory layout. Each character consists of 8 bytes (rows) of 8 bits (columns) each. Changing the pointer to the shape table is also easy since there is a DOS function (function 25H) to handle it.

We discussed the IN and OUT instructions, which are used to access the 8088's "I/O address space" as opposed to its "memory address space". In general, most hardware devices are interfaced to the PC by assigning them I/O addresses (*ports*) through which they can communicate with the PC. We also saw a "map" of the I/O address space giving many of the port assignments. Recall that the IN and OUT instructions have the syntax

```
IN   accumulator,port
OUT  port,accumulator
```

where, of course, the accumulator is either AL (the usual case in the IBM PC) or AX. The port operand is either an immediate value in the range 0-255 or else is the DX register (which *contains* the port address 0-64K). Because of the necessity of continually using AL and DX in these instructions, we also introduced macros

```
IN_PORT  destination,port
OUT_PORT  port,source
```

in which the port, destination, and source could be given by any of the usual addressing modes.

We discussed direct programming of the video hardware. The video hardware consists of a 6845 CRT controller chip and various supporting devices. Programming the video hardware consists of programming the 6845's 19 internal registers, and programming various auxiliary registers outside of the 6845. Basically, directly programming the 6845 provides little capability not already inherent in BIOS, except

that we gain the ability to turn off the cursor, and the ability to scroll the screen by individual characters rather than by line. There was, however, a quite useful auxiliary register, the status register 3DAH (3BAH for the monochrome adapter), that allowed us to tell when horizontal or vertical retrace was occurring, thus helping us to avoid priority clashes between the 8088 and 6845 and preventing nasty screen flicker.

We also began discussing how to program serial I/O. The serial I/O hardware consists mainly of an 8250 UART (Universal Asynchronous Receiver Transmitter), which also does most of the dirty work for us. Most of the programming work is in initializing the 8250, which we ignored in the previous class. We found, however, that the actual I/O -- i.e. sending bytes out through the UART, or receiving bytes from the UART -- was quite easy, and we wrote a dumb terminal program to illustrate that fact.

#### Initializing the Serial I/O Protocol

Recall that the following port assignments are used for the serial I/O hardware:

| <u>I/O Port</u> | <u>Used for</u> | <u>Description</u>  |
|-----------------|-----------------|---|
| 3F8H            | output          | a) transmitter holding register<br>b) baud-rate divisor (LSB) |
| 3F8H            | input           | receiver data register  |
| 3F9H            | output          | a) interrupt-enable register<br>b) baud-rate divisor (MSB)    |
| 3FAH            | input           | interrupt-identification reg.                                 |
| 3FBH            | output          | line-control register   |
| 3FCH            | output          | modem-control register  |
| 3FDH            | input           | line-status register  |
| 3FEH            | input           | modem-status register   |

These are the port addresses for COM1 (the "primary" serial port), while the port addresses for COM2 (the "secondary" serial port) are the same except they are of the form 2FnH rather than 3FnH.

Notice that two of the ports, including 3F8H (which we used so extensively in the previous class), apparently each have two distinct functions. These distinct functions are distinguished (much as 6845 registers are selected) by the value written to another I/O port -- the *line-control register* 3FBH. If the value 128 is written to the line-control register, then functions "b" (baud-rate definition) are selected. If bit 7 of the line-control register is zero, functions "a" are selected.

The "baud rate" (or simply *baud* if we are being picky) is the speed (in bits/second) at which data is transmitted. To the data bits are appended various other bits, mainly for error-checking purposes. A reasonable rule of thumb is that the baud rate must be divided by 10 to get the transmission rate in bytes/second. The highest baud rate available on the PC is 9600 baud, or about 1000 bytes/second. Not all baud rates are used. Some typical baud rates are: 300, 1200, 2400, 9600, and 19200 (though not with a PC). Normally, the baud rate is set only during initialization and is then forgotten. Here are the allowed

values for the "baud rate divisor" stored in ports 3F8H and 3F9H when functions "b" are selected:

| <u>BAUD RATE</u> | <u>DIVISOR (IN HEX)</u> |
|------------------|-------------------------|
| 50               | 900                     |
| 75               | 600                     |
| 110              | 417                     |
| 134.5            | 359                     |
| 150              | 300                     |
| 300              | 180                     |
| 600              | C0                      |
| 1200             | 60                      |
| 1800             | 40                      |
| 2000             | 3A                      |
| 2400             | 30                      |
| 3600             | 20                      |
| 4800             | 18                      |
| 7200             | 10                      |
| 9600             | C                       |

There are many more choices here than allowed by BIOS (though most are useless), and the same holds true of many other serial I/O parameters. As a simple example of how to set the baud rate, let us suppose that we want to use 100 baud. The divisor is 417H, so we have to send the least significant byte (17H) to one port and the most significant byte (4H) to another.

```

OUT_PORT  3FBH,128      ; SELECT FUNCTIONS "B".
OUT_PORT  3F8H,17H     ; LSB OF BAUD DIVISOR.
OUT_PORT  3F9H,4H      ; MSB OF BAUD DIVISOR.
OUT_PORT  3FBH,00000111B ; EXPLAINED BELOW.

```

(Alternately, we could just put 417H into AX and use a *word* OUT instruction rather than a byte OUT; this is one of the few word-I/O ports in the IBM PC.) Of course, typically, we would also reset bit 7 of port 3FBH to zero -- otherwise, we couldn't use the primary functions of these ports: we couldn't use port 3F8H for inputting or outputting data. There is a problem with this in that (other than bit 7) we don't know what value port 3FBH is supposed to have. (And we can't use an IN instruction to find out, since the port is read-only.) The program fragment above uses a very common setting for the line-control register.

The meaning of the bits in the line-control register 3FBH is as follows:

| <u>BIT</u> | <u>MEANING</u>   |
|------------|--|
| 1,0        | 00 = 5 BIT DATA<br>01 = 6 BIT DATA<br>10 = 7 BIT DATA<br>11 = 8 BIT DATA |
| 2          | 0 = 1 STOP BIT<br>1 = 2 STOP BITS (1.5 FOR 5 BIT WORDS)                  |
| 3          | 0 = NO PARITY<br>1 = PARITY  |
| 4          | 0 = ODD PARITY<br>1 = EVEN PARITY  |
| 5          | 0 = DISABLED (USE THIS ONE!)<br>1 = STICK PARITY                         |
| 6          | 0 = DISABLED (NORMALLY)<br>1 = SEND BREAK CHARACTER                      |
| 7          | SELECTS BETWEEN FUNCTIONS A AND B  |

One common serial I/O protocol uses 8-bit words, two stops, and no parity which, from an examination of the table above, would result in the line-control register being set to 00000111B. This is, by no coincidence, the value we have used to reset the line-control register with in the example above. The protocol mentioned is often found on computer bulletin boards. Note, however, that several other protocols are common. In general, the correct protocol will have to be researched individually for every device (including remote mainframes) connected to the PC. Mainframes often use protocols featuring 7-bit data "words", since they transmit only standard ASCII characters, and these require only 7 bits for their specification.

In general, because most of the serial port features controlled by the baud-rate divisor and line-control I/O ports (but *not* implemented by BIOS) are not useful, there is little reason to directly program these I/O ports in your programs. The only common exception is if you want to send a "break" character to the remote device (which in this case is presumably a mainframe). For this, you must use the line-control I/O port:

```
; PROGRAM TO SEND A BREAK CHARACTER IF WE ARE USING 8-BIT WORDS,
; TWO STOPS, AND NO PARITY:
    OUT_PORT 3FBH,01000111B    ; SEND BREAK.
    MOV      CX,DELAYCOUNT    ; PREPARE TO DELAY A CERTAIN TIME.
WAIT: LOOP   WAIT
    OUT_PORT 3FBH,00000111B    ; STOP THE BREAK.
```

Of the other I/O ports that we haven't discussed, two are used to allow the UART to perform interrupt-driven I/O, while the other three are used for "handshaking" purposes.

#### Hardware Handshaking for Serial I/O

The ideas we have discussed so far are all right for mere transmission and reception of data by means of a serial interface, but there are many conditions that have not been considered. Let us consider, for example, the case in which the IBM PC is being used as a terminal in order to communicate with a remote mainframe over the telephone lines. The remote computer could be ten feet away, or it could be 3000 miles away, and there are many circumstances that could

ruin the connection between the two. This is not especially serious if a human being is constantly supervising the operation (for instance, if you are sitting at the IBM PC and watching its screen), but if the computers are operating unattended it would be best if the PC could have some better indication of how things are going.

This problem is approached by adding various status signals (i.e., extra wires) to the serial interface. These extra signals are set or checked by the PC and by the remote device. Some of the signals are basically relevant only to devices which are directly connected (with an RS-232 cable) to the computer. Others are relevant if the PC and the remote device are connected via a modem over a telephone line. (That is, if the PC is connected via RS-232 to a modem, and the modem connected to the telephone system. Of course, the modem is itself a "remote device" which is directly connected to the computer, so the direct-connect signals could still be relevant.)

The extra wires collectively allow hardware "handshaking". "Handshaking" is the term for what we have been discussing -- it is the passing of control and status signals between devices, as well as mere data. Hardware handshaking differs from software handshaking in that the control and status information is sent via wires rather than, for example, through funny data bytes (such as ASCII control characters).

Here are the extra RS-232 signals:

| <u>NAME</u> | <u>I OR O</u> | <u>RELEVANCE</u> | <u>DESCRIPTION</u>   |
|-------------|---------------|------------------|--|
| RTS         | OUTPUT        | DIRECT           | REQUEST-TO-SEND -- INFORMS THE REMOTE DEVICE THAT THE PC WANTS TO SEND DATA.                         |
| CTS         | INPUT         | DIRECT           | CLEAR-TO-SEND -- TELLS THE PC THAT THE REMOTE DEVICE WILL ACCEPT DATA.                               |
| DSR         | INPUT         | DIRECT           | DATA-SET-READY -- TELLS THE PC THAT THE REMOTE DEVICE IS ACTUALLY POWERED UP AND CONNECTED.          |
| DTR         | OUTPUT        | DIRECT           | DATA-TERMINAL-READY -- TELLS THE REMOTE DEVICE THAT THE PC IS POWERED UP AND CONNECTED.              |
| DCD         | INPUT         | MODEM            | DATA-CARRIER-DETECT -- TELLS THE PC THAT THE MODEM IS RECEIVING A SIGNAL OVER THE TELEPHONE LINE.    |
| RI          | INPUT         | MODEM            | RING-INDICATOR -- TELLS THE PC THAT THE MODEM IS GETTING A RINGING SIGNAL FROM THE REMOTE TELEPHONE. |

The four input signals can all be read from the modem status port 3FEH:

| <u>BIT</u> | <u>MEANING</u>                            |
|------------|---|
| 0          | =1 if the CTS signal has <i>changed</i> . |
| 1          | =1 if the DSR signal has <i>changed</i> . |
| 2          | =1 if the RI signal has <i>changed</i> .  |
| 3          | =1 if the DCD signal has <i>changed</i> . |
| 4          | =1 if the CTS signal is <i>set</i> .      |
| 5          | =1 if the DSR signal is <i>set</i> .      |
| 6          | =1 if the RI signal is <i>set</i> .       |
| 7          | =1 if the DCD signal is <i>set</i> .      |

(It can happen, by the way, that some remote devices use the *inverses* of these signals -- or use strange combinations of the signals -- to indicate status conditions we are discussing or for some other kind of handshaking. In general, whether this is so must be determined individually for every device connected to the computer.)

The two output signals of the PC (DTR and RTS) can be set or reset by the program. Changing these signals, and various other quantities, is accomplished by changing the modem-control register, at output port 3FCH. Bits 0 and 1 of this port are, respectively, used to set DTR and RTS. As with the status signals discussed in the previous paragraph, some remote devices have odd interpretations for these signals. In principle, since DTR is used simply to indicate that the PC is present, it should be so initialized and then forgotten. RTS, on the other hand is used to indicate that the PC *wants* to receive data, and may therefore change quite often.

Another factor which makes the job of properly programming the handshaking is that some remote devices require funny cables that permute the control signals. However, in theory, here is how the relevant signals of the computer and the remote device should be connected:

| <u>COMPUTER</u>        | <u>DEVICE</u>                 |
|------------------------|-------------------------------|
| TXD (transmitted data) | -----> RXD (received data)    |
| RXD (received data)    | <----- TXD (transmitted data) |
| RTS                    | -----> CTS                    |
| CTS                    | <----- RTS                    |
| DSR                    | <----- DTR                    |
| DTR                    | -----> DSR                    |

where the arrows indicate the direction of information flow.

Programming the handshaking will probably be more understandable with an example. Let us consider the dumb terminal program discussed in the last class, and assume that the PC is directly connected to the remote terminal (with RTS and CTS properly connected) rather than communicating through the telephone. Under what circumstances would we require any kind of handshaking in that program? Well, the remote computer might occasionally become busy (changing the CTS being received by the PC), so we might want to light up a little "busy" message on the screen. Similarly, the PC itself might want data input to stop for a short time. How could this happen? Typically, since it is maddening to sit in front of a computer screen *waiting* for the display to be drawn, we always use as high a baud rate as possible. This is fine for the transmission and reception of ordinary printable characters since the computer is fast enough to display the character before the next character arrives. Some control characters, however, take a relatively long time to "display": for instance, a carriage-return/line-feed might cause the display to scroll. At 9600 baud, a new character is being input every 1/1000 second; if (say) it takes 10/1000 second to scroll the screen, up to ten characters could be lost. Thus, having received a carriage return character from the serial port, we might want to manipulate RTS to cause any further input to stop until the screen is finished scrolling.

Here is the modified pseudocode for our dumb-terminal program:

```

REPEAT
  IF CTS SAYS REMOTE COMPUTER IS BUSY THEN
  BEGIN
    SET RTS.
    DISPLAY "BUSY" AT SOME DESIGNATED "STATUS" SCREEN LOCATION.
    REPEAT (NOTHING) UNTIL CTS SAYS REMOTE COMPUTER IS NOT BUSY.
    DISPLAY "    " AT THE "STATUS" LOCATION.
    CLEAR RTS.
  END.
  IF CHARACTER READY AT KEYBOARD THEN
  BEGIN
    GET THE CHARACTER FROM THE KEYBOARD.
    DISPLAY IT ON THE SCREEN.
    IF THE CHARACTER IS CARRIAGE RETURN THEN DISPLAY LINE-FEED.
    WAIT UNTIL THE UART IS NOT BUSY.
    SEND CHARACTER TO THE UART.
  END.
  IF CHARACTER READY AT RS-232 THEN
  BEGIN
    GET THE CHARACTER FROM THE UART.
    IF NOT A LINE FEED THEN DISPLAY IT
    ELSE BEGIN
      SET RTS.
      DISPLAY THE LINE FEED.
      CLEAR RTS.
    END.
  END.
FOREVER

```

We have also manipulated RTS to print the "BUSY" message, since all of that moving around on the screen could take a long time. This program could send up to one character to the remote computer after it goes "BUSY", but before the transition is detected. This is okay; since communications is not synchronized (and occurs at a finite speed), the remote computer has to be prepared for this possibility.

This notion of hardware "handshaking" -- i.e., of connected devices sending not only data but also status signals to each other -- is very important and comes up for any kind of interface. For example, the IBM PC's "parallel port", which is used to connect printers to the PC but which we have no reason to discuss (since BIOS is just as good), employs the same ideas.

Before we leave the topic of handshaking, there are two other bits of the modem control register that should be mentioned. Bit 3 is relevant to interrupt-driven I/O as discussed below. Bit 3 must be one for interrupt-driven I/O and can be zero otherwise. Bit 4 is the "loop back" bit. If bit 4 is set, the UART internally connects its input to its output, so that programs can sometimes be tested using this feature even if a remote device is not attached. If, for instance, we set bit 4 and then used our simple dumb terminal program from the previous lecture, every key typed at the keyboard would be echoed *twice* to the screen. The first echo comes from the fact that all keyboard characters are automatically echoed to the screen by the program; the second from the fact that the keyboard character is sent out over the serial interface, but the serial output is looped back to the input, so

the character is received from the serial port and displayed a second time by the program.

It should also be noted that in many cases hardware handshaking is not possible for the simple reason that the connected device is physically too far away and it would be expensive to use enough wires to allow hardware handshaking. In this case, software handshaking is often used. In software handshaking, "control characters" sent over the input and output wires are used to convey the meanings normally conveyed by the status wires. One common scheme is to use the ASCII control characters 0-31, which have standardized meanings in this regard. There are too many such control characters to discuss, but two are noteworthy. Ctrl-S (or "XOFF"), which we often use to pause the display in DOS, is typically used as a signal to the remote device to indicate that it should stop sending data for a while. Ctrl-Q (or "XON") is often used *after* a ctrl-S to tell the remote device to start sending data again.

#### Interrupt-driven Serial I/O

Recall that in previous lectures I mentioned that in the IBM PC it is possible to have interrupt-driven input and output, and that interrupt-driven I/O is used for the sake of efficiency. For interrupt-driven *input* to the computer, the peripheral device sends an interrupt signal to the computer whenever it has data it would like to give to the computer. This is useful if the input can occur *asynchronously* -- i.e., at random times that cannot be predicted by the computer in advance. Thus, the computer user can strike a key at the keyboard at any time and still expect it to be received. When a keystroke is received by the computer, the character is stored in a buffer until such time as the program should call for it. There is no reason why other input cannot be handled like this. For example, there is no reason why serial input from the RS-232 port cannot be buffered by an interrupt routine. This would prevent the computer from either losing input or else spending all of its time monitoring the peripheral device for input.

For interrupt-driven *output*, the computer buffers all of its output data. Then, whenever the peripheral device isn't busy, it interrupts the CPU, and the interrupt routine removes characters from the buffer (the output queue), giving them to the peripheral device. This prevents the computer (which operates very quickly) from wasting a lot of time waiting for the peripheral device (which may be very slow) to finish outputting the data. As an example, print spooler programs work like this. A print spooler allows the computer to "print" text as fast as it can (thinking that the output is all going to the printer); instead, the text is buffered, with the printer periodically interrupting the computer for new characters. After all of the text is in the buffer, the computer can go ahead with the next program -- while the printer is still printing the previous document. These ideas also hold for the serial interface which (even at 9600 baud) is slow by computer standards.

It should therefore come as no surprise that the serial interface can be set to interrupt the 8088 under a variety of conditions. The interrupts are controlled by the interrupt-enable register (output port 3F9H) and the interrupt-identification register (input port 3FAH).

There are four different conditions under which the 8250 (the UART) can be set to interrupt the CPU. The conditions are these: 1) the UART has an input character ready for the CPU ; 2) the UART is ready to get another output character from the CPU; 3) the UART has detected a reception error or an input "break" character; and 4) the UART has detected some change in the input CTS, DTR, DCD, or RI handshaking signals. Each of these kinds of interrupts can be selectively enabled or disabled by programming the interrupt-enable register at port 3F9H. The interrupts are enabled by, respectively, setting bits 0-3 of the interrupt-enable register. Any bit left at zero disables that particular type of interrupt. (Also, as mentioned earlier, bit 3 of the modem-control register must be 1 for any of these interrupts to have effect.)

If any interrupt signal from COM1 is received by the CPU, interrupt vector 12 (0CH) is used to activate an interrupt processing routine. To determine *what type* of interrupt it was, the interrupt-identification register at 3FAH must be read. Bit 0 of this register is 0 if an interrupt is pending, and is 1 if no interrupt is pending. This information is used to determine if several interrupts might have been received simultaneously. Thus, after the interrupt-service routine has done its job, it should then check the interrupt-identification register again to determine if it needs to service another interrupt. Bits 2 and 1 indicate the interrupt type:

| <u>TYPE</u>          | <u>BITS 2,1</u> | <u>PRIORITY</u> | <u>ACTION TO RESET</u> |
|----------------------|-----------------|-----------------|------------------------|
| ERROR OR BREAK       | 1 1             | FIRST           | READ LINE-STATUS       |
| RECEIVED DATA        | 1 0             | SECOND          | READ RECEIVED DATA     |
| TRANSMITTER READY    | 0 1             | THIRD           | OUTPUT A CHARACTER     |
| MODEM STATUS CHANGED | 0 0             | FOURTH          | READ THE MODEM STATUS  |

If several interrupt conditions occur simultaneously, only one of them will be reported, and they will be reported in the order indicated under "priority". An interrupt condition is "cleared" by the indicated action, after which any remaining interrupt conditions should be processed.

Let's see what a typical interrupt service routine for interrupt vector 12 might be like:

```

; INTERRUPT SERVICE ROUTINE FOR COM1. NOTE THAT ALL REGISTERS MUST
; BE PRESERVED AND THERE MUST BE AN IRET AT THE END:
        PUSH        AX
AGAIN:   IN_PORT    AL,3FAH        ; GET INTERRUPT TYPE.
        TEST        AL,1          ; INTERRUPT PENDING?
        JNZ         DONE          ; IF NO, THEN QUIT.
        AND         AL,0110B      ; GET JUST INTERRUPT ID.
        JZ          MODEM_STAT    ; MODEM STATUS CHANGED.
        CMP         AL,0110B      ; CHARACTER ERROR?
        JE          CHAR_ERROR    ;
        CMP         AL,0010B      ; XMITTER READY?
        JE          XMIT_READY    ;
        JMP         CHAR_READY    ; CHARACTER IS READY.

DONE:   POP         AX
        IRET
    
```

```

; CODE FOR THE VARIOUS TYPES OF INTERRUPTS:
MODEM_STAT:
CHAR_ERROR:
XMIT_READY:
CHAR_READY:
        JMP         AGAIN

```

Just what code goes at the final four labels depends on the application. If we were using interrupt-driven input, we would maintain a "circular queue" of received characters. Recall that a "queue" is a data structure in which new data is added at one end and old data is removed at the other. In our case, the interrupt service routine 12 would put the characters into the queue, while the executing program would remove them whenever it wants. However, a straight queue would be less than worthless since as data is added to it it would creep its way, wormlike, up through memory. Obviously, this wouldn't be too good. A circular queue, on the other hand, is similar to a straight queue except that it has a fixed place in memory. If we try to move past the end of it, we go to the beginning of it. (That's why it's circular.) In essence, all addresses in the queue are "modulo" the queue size. Such modular arithmetic is especially easy (on a binary computer) if the queue size is a power of two, which we will require. To implement a circular queue we need a buffer to contain the queue, and we need two pointers into the queue, indicating the next position to add a character and the next position to remove a character. An error condition can occur if many more characters are added to the queue than are removed, and the character-addition pointer catches up with the character-deletion pointer. We will arrange to detect such an error if the addition-pointer is one less than the deletion-pointer. With these ideas in mind, here is how we might implement the CHAR\_READY function above:

```

; CODE TO SET UP A QUEUE OF INPUT CHARACTERS.  FOR MY ALGORITHM, THE
; BUFFER SIZE MUST BE A POWER OF TWO.
BUFSIZE    EQU    1024                ; MAKE THE BUFFER 1024 CHARS. LONG.
BUFFER     DB    BUFSIZE DUP (?)      ; THE QUEUE.
PUTPTR     DW    0                    ; NEXT POSITION TO ADD CHARACTER.
GETPTR     DW    0                    ; NEXT POSITION TO REMOVE CHAR.
ERROR      DB    0                    ; BECOMES ONE IF BUFFER OVERFLOWS.
; NOTE THAT PUTPTR IS UPDATED BY THIS ROUTINE, BUT GETPTR IS UPDATED
; ONLY WHEN A CHARACTER IS REMOVED FROM THE QUEUE -- WHICH THIS
; PROGRAM NEVER DOES.
CHAR_READY:
        PUSH     SI
        MOV     SI,PUTPTR              ; SET UP A POINTER TO THE BUFFER.
        IN_PORT BUFFER[SI],3F8H
        INC     SI                    ; MOVE POINTER.
        AND     SI,BUFSIZE-1          ; BUFFER IS CIRCULAR.  0 FOLLOWS
                                                ; BUFSIZE-1.
        MOV     PUTPTR,SI              ; SAVE NEW VALUE OF POINTER.
        CMP     SI,GETPTR              ; BUFFER OVERFLOW?
        JZ     OVERFLOW
        POP     SI
        JMP     AGAIN
; THIS CODE IS EXECUTED ONLY FOR A BUFFER OVERFLOW.
OVERFLOW:

```

```

MOV     ERROR,0FFH      ; ERROR CODE.
MOV     SI,GETPTR       ; MOVE GETPTR SO THAT PUTPTR
INC     SI              ; CAN'T GET PAST IT.
AND     SI,BUFSIZE-1   ; MAKE IT CIRCULAR.
MOV     GETPTR,SI      ; SAVE NEW GETPTR.
POP     SI
JMP     AGAIN

```

Of course, a routine to get a character from the buffer is extremely simple to write. Ignoring the possibility that an overflow error might have occurred (as indicated by ERROR):

```

; GET A CHARACTER FROM THE BUFFER INTO AL:
MOV     SI,GETPTR       ; WHERE TO GET THE CHARACTER FROM.
CMP     SI,PUTPTR       ; ANY CHARACTERS?
JZ     NO_CHARS        ; IF NOT, GO DO SOMETHING ELSE.
MOV     AL,BUFFER[SI]  ; GET THE CHARACTER.
INC     SI              ; UPDATE THE POINTER.
AND     SI,BUFSIZE-1   ; MAKE IT CIRCULAR.
MOV     GETPTR,SI

```

At this point, we will leave the topic of serial I/O, even though we have barely begun to cover it. We have spent so much time on it for several reasons. First, along with number crunching and high-resolution graphics, the deficiencies in built-in serial I/O software probably prompt more actual assembly language programming than any other area. Second, information about serial I/O is relatively difficult to come by and few people appear to understand the topic. Third, we were forced to consider several topics of great importance (such as handshaking and interrupt-driven I/O) which we might otherwise have neglected.

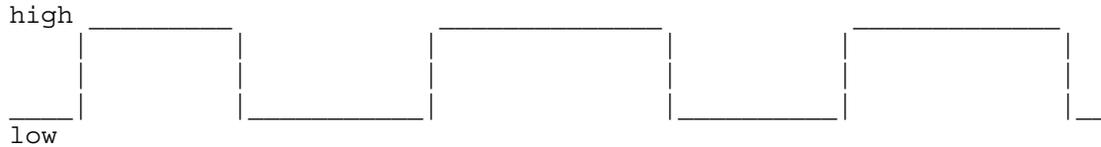
**ASSIGNMENT:** Read chapter 8 in the text.

Other Topics: Sound

While the IBM PC cannot be regarded as a music machine, it is capable of making a range of tones of selectable durations. There is no way (however complex) of controlling the volume of the sound, and there is no reasonable way of controlling the quality of the sound -- i.e., which harmonics are included in the tone.

Sound is a wave-phenomenon. Pictorially, we represent a sound as a wiggly line. The *height* of the wiggles represents the *volume* of the sound. The *number* of wiggles per unit distance along the wiggly line is the *pitch* or *frequency* of the sound. What distinguishes two sounds of the same pitch is the shape of the wiggles. Thus, a violin sounds different from a trumpet, even if the two are playing notes of essentially the same pitch. As mentioned, with the PC there is no way to change the height (volume) of the wiggles, and there is no way to change the shape of the wiggles.

The PC tries to produce sound as a "square wave". That is, instead of smooth oscillations like gentle waves on water, it produces sound like this:



It turns out that, physically, the speaker which actually ends up producing the sound is not capable of reproducing a square wave, so that the speaker ends up rounding off the sharp corners of the wave. Nevertheless, your program really has control over only two aspects of the sound -- how long the sound wave remains in the "high" part of the cycle, and how long it remains in the "low" part of the cycle. Together, these two characteristics allow you to control the pitch and, minimally, the quality of the sound.

The PC hardware allows you to either: a) select the frequency of the sound and just go ahead and play it while your program does other things; or b) control the high/low durations of each sound-cycle individually. The latter is more flexible, but (given the ease of use of the former option, and the poor quality of the PC as a sound instrument) the former option is probably preferable in most circumstances.

Since this topic is covered rather well (and concisely) by the reading assignment, and since no interesting new principles are illustrated in it, I do not see any particular reason to go beyond these remarks. Please feel free to have fun with it on your own.

LECTURE NOTES FOR CS-5330: IBM PC ASSEMBLY LANGUAGE  
UNIVERSITY OF TEXAS AT DALLAS

## CLASS 20

Comments

1. For anybody who is interested, I have prepared a slide comparing the results of the mid-term sorting competition. The slide compares *just* the speed of the sorting procedures and not the I/O. Included on the slide is everyone who turned in a routine compatible enough that I could force it into the form I had specified in the problem handout. The procedures are labeled by the last four digits of the social security number of the author, except my procedures (which are all adapted from Sedgewick's book *Algorithms*) and the SORT procedure provided with DOS. These results are in accordance with what could have been predicted by examination of the handout on sorting methods. Quicksort is the clear winner, and Bubble Sort the clear loser. DOS trails even Bubble Sort by a factor of three. Presumably, DOS actually uses a Bubble Sort, and the factor of three is due to the fact that DOS compares *capitalized* strings rather than strings in ASCII order.

Review

Most of the previous class was taken up by a cursory examination of serial I/O techniques.

We learned that several of the I/O ports used for serial I/O actually have two distinct functions. These functions are selected by writing either a 0 or a 1 to bit 7 of the *line-control* register at port 3FBH. Normally, bit 7 is 0, but for initialization bit 7 can be set to 1 -- in which case ports 3F8H and 3F9H are used to control the *baud* rate of the serial interface. Many different baud rates (including all of the usual ones) from 50 to 9600 baud are available.

The line-control register is also used to initialize various other parameters of the serial interface: the word-length, the number of stop bits, the parity, etc. Also, bit 6 of the line-control register is used to send a "break" character.

We discussed the handshaking signals available on the RS-232 interface -- these include RTS, CTS, DSR, DTR, DCD, and RI. We discussed both the normal interpretation of these signals, as well as how to program them. CTS, DSR, RI, and DCD are status (input) signals that can be read from the *modem-status port* 3FEH. The modem-status port tells both the state of the signal and whether it has *changed* since last read. DTR and RTS are control (output) signals which can be set or reset by writing to the *modem-control register* 3FCH. In addition, this register can also be used to "loop back" the UART's output to its input.

We discussed how the RTS and CTS signals are used to stop communications when the computer or the remote device are busy. We also saw the pseudo-code for an extension of our dumb-terminal program to include such handshaking.

We discussed interrupt-driven serial I/O. The UART can interrupt the CPU under four different conditions: transmitter empty, receiver full, reception error, or change in the status signals CTS, DSR, RI, and DCD. Whether or not any of these conditions *actually* interrupts the 8088 depends on how four bits are set in the *interrupt-enable register* at port 3F9H. If an interrupt occurs, it is always the same interrupt 12 (0CH). For the interrupt service routine to determine which condition caused the interrupt, the *interrupt-ID register* at port 3FAH is examined.

Interrupt-driven I/O can be used to eliminate CPU waiting for either slow input devices or slow output devices (with fast and slow being relative to the speed of the 8088). In either case, a "circular queue" is set up to hold the data. For interrupt-driven input, the interrupt service routine puts bytes in the queue, and the program removes them when needed. For interrupt-driven output, the program puts the data in the queue, and the interrupt service routine removes them whenever it is idle.

We also briefly discussed the sound-producing possibilities of the PC. We found that the PC basically produces "square wave" sound and that our only choice is the frequency (pitch) of the sound. However, the treatment of this topic was mainly left to the reading assignment.

#### What Else?

Since this is the last lecture, it is probably reasonable to mention some of the topics that we have *not* covered so far.

1. First, it should be recognized that even the topics we have spent some time on have been covered only sketchily. A number of 8088 instructions and a number of the features of the 8087 have not been covered. Many advanced (and elementary) features of MS-DOS have not even been mentioned. All of the hardware topics discussed have merely been skimmed.
2. We have had the chance only to discuss a little about printing hardcopy. In many ways, programming printers is as intricate a task as programming screen displays. Indeed, many printers are intelligent (they contain respectable processors) and have *many more* features than CRT displays. At the very least, most modern dot-matrix printers have dot-addressable graphics and programmable character sets. Unfortunately, the range of printers commonly used with IBM PCs is so great that even the sleaziest discussion of them is inappropriate. The "standard" IBM printer is related to the MX-80 printer manufactured by Epson, but even here there are so many variations that I am at a loss to cover any reasonable subset of the subject.
3. We have not at all discussed direct disk operations. Indeed, there are *several layers* of disk operations that we haven't discussed. We have discussed only MS-DOS disk operations at the file level. There are both higher levels and lower levels than this. MS-DOS *groups* files together into *directories* (and *sub-directories*) which can be manipulated to a certain extent. This is the higher level we haven't discussed. At the lower levels, we have not at all discussed *how* MS-DOS actually stores files on disks. Physically, disks have concentric "tracks" or "cylinders", which are divided into "sectors". Information

on the disk is stored in the sectors -- this includes files, directories, the operating system, etc. MS-DOS has a reasonably complicated scheme for calculating how sectors are grouped together to form files. We have not discussed the DOS or BIOS functions for reading or writing chosen sectors on the disk, and have limited ourselves to accessing entire files. Programming at this level is necessary to "fix" a disk which has somehow been messed up. Nor have we discussed a level even lower than this: directly programming the disk I/O hardware rather than relying on BIOS. Such programming is necessary if, for example, we are interested in copy-protection schemes.

4. We have not discussed installable device drivers. Installable device drivers are possibly the most significant aspect of MS-DOS. Recall that not only files but also *i/o devices* have "filenames" by which they can be accessed. An installable device driver sets up a new device name (of your choice) with a function of your choice, which can then be used in a totally uniform way under any programming language and many utilities. This concept has far wider application than it might seem. It applies not only to real I/O devices for which there is actually a chunk of hardware, but for "virtual" devices like print spoolers and RAM-disks. Indeed, it can be applied to almost any function performed by the computer. As an example, consider the ANSI installable screen driver. In many programs, in many languages, one would like to be able to move the cursor around on the screen at will. However, the capabilities differ from programming language to programming language. Because of the ANSI driver there is a uniform interface and level of competence across all programming environments. Let's consider a more severe (real life) example. Suppose that we were to add a "speech card" to the PC, so that the PC could produce intelligible English speech. Consider a very likely case: to get the card to work (so we can play with it), we try to control it using a BASIC program. That is, we write a "device driver" routine in BASIC. This is great, because we are now in the fortunate position of being able to add speech to all of our BASIC programs. Now, on the other hand, suppose that tomorrow we wanted to make the device speak from Pascal instead of from BASIC. What could we do? Well, we could completely rewrite our BASIC "device driver" in Pascal -- more likely, we would just skip it altogether and use BASIC. Of course, the latter course isn't possible if we are simultaneously trying to use *two* devices, one controlled by BASIC and one controlled by Pascal. Or, if we were trying to give it to a friend who loved Pascal and hated BASIC. Or, if we were doing this on the job and it needed to be in Pascal for a good reason. Or, if we suddenly discovered the thing was so great that we could sell it and make money .... If, instead, we had been clever enough to write an installable device driver (called, say, VOICE) to control the speech board, then we could immediately make speech from either BASIC or Pascal, merely by accessing the "file" VOICE.

5. We have not discussed *interfacing* our assembly language programs to higher-level languages -- for instance, how to write an assembly-language SUBROUTINE for use in a FORTRAN program.

6. And so forth ....

Clearly, we have only a limited amount of time. Therefore, today I would just like to discuss a little about interfacing with higher-level languages (item 5). We will relegate the rest of the stuff to a pile marked "might have been", and think nostalgically about it.

#### The LDS and LES Instructions

Two 8088 instructions which we will get a lot of use out of in a moment are the LDS and LES instructions. These instructions load a *doubleword pointer* into a pair of CPU registers. The syntax of these instructions is

```
mnemonic register,source
```

Here, of course, the mnemonic is either LDS or LES. The register operand is any of: AX-DX, SI, DI, BP, or SP, although only SI, DI, and BX are of real use. The source operand is a *doubleword* variable in memory.

The value of these instructions is that if the source operand contains a segment:offset pair, then the offset will be loaded into the specified general register, while the segment will be loaded into the DS register (for LDS) or the ES register (for LES). Thus, these instructions can offer a quick way to set up (for example) DS:SI, DS:DI, ..., ES:BX to point at some desired variable in memory.

As an example, let us consider a case somewhat like our sorting project, in which we maintain an array of *doubleword* pointers to strings rather than *word* pointers:

```
STRING1  DB  14,"I AM A STRING"
STRING2  DB  19,"I AM ANOTHER STRING"
STRING3  DB   4,"ETC."
...
POINTERS DD  STRING1
          DD  STRING2
          DD  STRING3
          ...
```

Hence, at POINTERS is the doubleword address of string 1, at POINTERS+4 is the doubleword address of string 2, etc. To load ES:DI with the address of string 2, we could do something like

```
LES  DI,POINTERS+4
```

or something like

```
MOV  SI,4
LES  DI,POINTERS[SI]
```

Thus, to set up ES:DI and DS:SI for a comparison of string 2 and string 3, we might do something like

```
MOV  BX,4
LES  DI,POINTERS[BX]
MOV  BX,8
LDS  SI,POINTERS[BX]
```

Note that we could *not* load DS:SI and *then* ES:DI since POINTERS is in the data segment and by the time LES is executed the DS register would have been destroyed and the POINTERS array would therefore be located incorrectly. Below, we will see applications in which the doubleword addresses are variables on the stack, so this comment would not apply.

### Interfacing to Higher-level Languages

\*\*\*\*\*  
**NOTE:** The information in this section is abstracted from the following sources:

Microsoft FORTRAN Compiler User's Guide, Chapter 9, by Microsoft.

Turbo Pascal 3.0 Reference Manual, pp. 210-211, pp. 216-219, and pp. 221-226 by Borland International.

The IBM Personal Computer from the Inside Out, section 4-5, by Sargent and Shoemaker. (Discusses compiled and interpreted BASIC and Microsoft FORTRAN and Pascal.)

Organization and Assembly Language Programming, Chapter 11, by Franklin. (Discusses BASICA and Microsoft FORTRAN and Pascal.)

Inside the IBM PC, Appendix 3, by Norton. (Discusses Microsoft Pascal.)

8087: Applications and Programming for the IBM PC and Other PCs, Chapter 8, by Starz. (Discusses compiled and interpreted BASIC.)

I have not personally used any of this information except for that pertaining to FORTRAN and Turbo Pascal, so I cannot vouch for its general correctness or completeness.

\*\*\*\*\*

In general, our macros BEGIN, VAR, and RETURN implement sort of a generalized high-level language interface, although they won't work without modification for most specific languages. Real languages usually differ from our interface specification, though the exact way they differ varies from language to language. Real interfaces can differ from ours in basically three ways:

- 1) Procedures might be NEAR (we always make them FAR).
- 2) We have typically passed the *values* of byte or word variables and the *addresses* of all other variables (such as arrays) as arguments to procedures. Some languages choose differently between "value arguments" and "variable arguments".
- 3) For "variable arguments" of procedures (i.e., arguments whose addresses are passed), we have always used the *offsets* as the addresses. Some languages require both *segments* and *offsets*.

These possibilities can mostly be taken care of by fixing up the BEGIN, VAR, and RETURN macros for whatever language is being used, plus some changes in the way we use the macros. Before seeing any explicit

examples of how to do this, let's see what various languages require in terms of interfacing. First there are some (apparently) universal requirements: all languages require that BP and the segment registers (ES, DS, SS) be *preserved* by the assembly language routine. Also, the assembly language routine must *POP* all arguments from the stack on return (except where explicitly stated below). Apparently, it is also usual to declare the code segment as

```
CODE      SEGMENT  'CODE'
```

although I not yet seen any statements to the effect that this is necessary.

We will discuss the specific requirements of the following languages: Microsoft FORTRAN and Pascal, Turbo Pascal, BASICA (but only briefly), and Compiled BASIC. Unfortunately, I have no information on the interfacing requirements of any C compiler. Indeed, I do not pretend exhaustive coverage even of the few languages discussed. Later, we will see some programming examples of the information given below:

MICROSOFT FORTRAN. All FORTRAN CALLs are FAR, as ours are. All arguments, however, are *variable* parameters -- they are the *addresses* of the values rather than the values themselves. Furthermore, they are all doubleword addresses. Thus, after a statement like CALL SORT(A,N), the stack would look like

```
(BOTTOM OF STACK)
...
SEGMENT OF A
OFFSET OF A
SEGMENT OF N
OFFSET OF N
SEGMENT OF RETURN ADDRESS
SP => OFFSET OF RETURN ADDRESS
```

With an arrangement like this, the LDS and LES instructions are very useful in loading the addresses of the arguments into registers.

**Functions:** FUNCTIONS as well as SUBROUTINES can be written in assembly language. If the returned value is a *word* (INTEGER\*2 or LOGICAL), it is returned in the AX register. If the returned value is INTEGER\*4 or LOGICAL\*4, it is returned in the DX:AX register pair. Any REAL or COMPLEX result is returned in a temporary (*not* TEMPORARY REAL) variable created by the compiler. The address of this temporary variable is pushed onto the stack after all of the parameters. (In the above example, if SORT was a REAL function, the address of the result would be on the stack between OFFSET N and the segment of the return address.)

**Declaration:** No special declaration of the assembly language procedure is required in the FORTRAN program. (I.e., the EXTERNAL declaration is not required.)

**Data Types:** a) The CHARACTER\*n type is simply an array of n bytes; b) the REAL type is the IEEE format (same as 8087) for FORTRAN versions 3.0 and later, but is the Microsoft format for earlier versions. The Microsoft real number format is not compatible with the 8087.

**Segments:** If you want to use local variables in the data segment instead of the stack, you *must* declare the data segment as follows:

```
DATA      SEGMENT   PUBLIC 'DATA'
;      ... (DATA GOES HERE) ...
DATA      ENDS
DGROUP   GROUP     DATA
          ASSUME    DS:DGROUP
```

MICROSOFT PASCAL. These procedures are also FAR. The way arguments are passed is more flexible than in FORTRAN, however. Value parameters -- i.e., those without the "VAR" specifier in the PROCEDURE -- are passed by value, and variable parameters are passed by address. Variable parameters using VAR have only the offsets of the addresses on the stack, while those using the alternate form VARS have both the segment and the offset on the stack.

**Functions:** Pascal FUNCTIONS as well as PROCEDURES can be written in assembler. Single byte results (BYTE or SINT) are returned in the AL register and word results (WORD or INTEGER) are returned in AX. I do not know how other data types are passed, but I assume it is by a mechanism similar to FORTRAN's, since the two compilers are so similar.

**Declaration:** Pascal requires each assembly language procedure to be explicitly declared in the main program, using the reserved word EXTERNAL. For example, for our sorting project, we would have to put the line

```
PROCEDURE SORT_ARRAY(N:INTEGER; VAR A: ...); EXTERNAL;
```

(Apparently, EXTERN can also be used rather than EXTERNAL.) The type specification after A has been left blank since it depends how A has been declared. This particular declaration actually works immediately with our sorting procedure since it calls for an integer passed by value and an array passed by offset. On the other hand, using "VARS N:INTEGER; VARS A" would produce a CALL identical to FORTRAN's.

**Data Types:** The LSTRING data type consists of a one-byte count of the number of characters in the string, followed by the string itself. LSTRING parameters (arguments) are passed to the assembly language program in a funny way. First, the *maximum* allowed length of the string is pushed onto the stack, then the address of the string (assuming a VAR or VARS is used) is pushed. Thus, a VAR LSTRING parameter takes 4 bytes on the stack rather than two.

BASICA AND COMPILED BASIC. Through sheer chance, BASIC's calling conventions agree perfectly with ours. That is, our BEGIN, VAR, and RETURN macros work perfectly as is. Assembly language routines for interpreted BASIC should be *dynamically relocatable*, which is to say that it should be possible to put them in any segment (though not necessarily any offset in the segment). The easiest way to achieve this is to avoid having any separate data, stack, or extra segment for the routine. On the other hand, another thing to keep in mind is that BASIC only guarantees a stack of 8 words when your assembler routine is run (although apparently it is often larger). Thus, it may be necessary to set

up your own stack, which is a direct contradiction of the advice given above.

**Declaration:** In compiled BASIC, no special declaration of the assembler routine is required; the routine is called with a CALL statement. In interpreted BASIC, however, a number of additional steps are necessary. Since interpreted BASIC is not LINKed, the assembler routine must be both assembled and LINKed separately. Also, special techniques must be employed to get the assembler routine *into* memory for BASIC to access, as well as to get BASIC to recognize the routine. I have seen at least three separate methods for doing this, each of which is too complicated for someone who has not used them to usefully describe. I recommend consulting the IBM BASIC manual, or the references I have given above, if you want to interface to interpreted BASIC.

**Data Types:** Strings are handled differently in compiled and interpreted BASIC. In interpreted BASIC, passing a string parameter to an assembler routine causes the *address* of a 3-byte *string descriptor* to be placed on the stack. The string descriptor, in turn, consists of a one-byte character count and a two-byte address (pointing to the actual string). In compiled BASIC, the situation is the same except that the string descriptor consists of a 2-byte character count and a 2-byte address. (What fun, eh?)

TURBO PASCAL. Even though Turbo Pascal does not use LINK, it is still easy to interface it to assembler programs. All assembler procedures are NEAR (rather than FAR, as we have always found so far), and either the *values* of parameters or the *addresses* are passed, depending on whether the VAR declaration is used. If VAR is used, doubleword addresses are passed on the stack. (Thus, VAR is equivalent to the Microsoft Pascal VARS.)

**Functions:** Pascal FUNCTIONS as well as PROCEDURES can be written in assembler. Function calls push an additional parameter onto the stack to hold the result of the function; however, BOOLEAN, BYTE, INTEGER, and pointer results are actually returned in registers and this extra parameter must be POPped by the routine. Thus, an integer function SORT(VAR I:INTEGER; J:INTEGER) would set up the stack like

```

                (BOTTOM OF STACK)
                ...
                VALUE OF SORT RESULT
                SEGMENT OF I
                OFFSET OF I
                VALUE OF J
SP =>  OFFSET OF RETURN ADDRESS

```

As mentioned, the function value parameter on the stack is used to return a result only if the data type is bigger than a word. INTEGER results are returned in AX. BYTES (and CHARs) are returned in AL with AH=0. BOOLEANs return with the ZF flag set if false and ZF cleared if true. Pointer values are returned in DX:AX.

**Declaration:** Since the assembler routines are not LINKed, the name of the routine need not be declared PUBLIC (though it doesn't hurt to do so). The assembler routine must be assembled, linked, and then turned into a "binary" file by the program EXE2BIN.

Assuming, for example, that we had started with a program named "SORT.ASM", this would finally result in a file "SORT.BIN". The assembler routine must be declared (assuming the example used above) in the Pascal calling program as

```
FUNCTION SORT(VAR I:INTEGER; J:INTEGER):INTEGER; EXTERNAL 'SORT.BIN';
```

This differs from the Microsoft Pascal declaration in that the name of the file is also included. This is done because the compiler will load SORT.BIN at *compile time* and therefore needs to be told where to find it. There is a disadvantage to this in that there is no way of knowing where in the code segment the assembler routine will be loaded. (Remember, it is a NEAR routine, so the CS register does not point directly to it when the procedure is called.) Therefore, the assembler routine must be completely relocatable. In such a routine, therefore, you *must not use CALLs* (since there is no way to know the location of called routines), or use NEAR jumps (since there is no way to know the offsets to jump to), or use your own data, stack, or extra segments (since you couldn't know where they are in memory). This is not a real hardship in many cases since you can still use conditional jumps, LOOPS, and SHORT jumps. However, it does make jump-tables rather difficult to implement. Also, if you are using many of our macros, it necessitates changing them so that they use only the normal conditional jumps rather than the "improved" JP form.

**Data Types:** a) the format of REALs depends on the version of Turbo Pascal being used. Currently, there is a version which does software floating point (using a 6-byte format), a version which uses 8087 double-precision format, and a version using a 10-byte BCD format (not compatible with any 8087 format). b) Strings are stored with the first byte being a character count and all remaining bytes constituting the string itself. c) A pointer is a doubleword segment:offset pair.

**Segments:** As mentioned, you should not set up your own segments. The DS register points to a segment containing all of the global variables of the Pascal program. The first variable declared is put at the top of the segment. The second is immediately below it, the third is below that, etc. However, unless you can determine the offset of the top of the segment (which is less than 0FFFFH unless 64K of variables are defined), this information is not a lot of use.

In presenting programming examples for the information given above, I will concentrate on Microsoft FORTRAN and Turbo Pascal since these are what I am familiar with. (Also, given the low price and quality of Turbo Pascal, it is irrational to program in any other traditional higher-level language -- but we won't get into that.) As noted above, however, it is possible to declare a Microsoft Pascal procedure in such a way that a call identical to that of Microsoft FORTRAN is produced, so any FORTRAN-callable SUBROUTINE produced can be called from Microsoft Pascal as well (so long as compatible data types are used).

As an example, let us suppose that we are working on a machine equipped with an 8087 coprocessor and we would like to speed up our number-crunching a little. In particular, we would like to have an assembler routine to compute the *dot-product* of two vectors. Recall

that the dot-product is given by multiplying the corresponding element of the two vector and then adding up all of the products. In FORTRAN, a routine to do this would look like this:

```

        SUBROUTINE DOTPRO (ARRAY1,ARRAY2,N,RESULT)
        DOUBLE PRECISION RESULT,ARRAY1(N),ARRAY2(N)
        RESULT=0.0D0
        DO 10 I=1,N
10      RESULT=RESULT+ARRAY1(I)*ARRAY2(I)
        RETURN
        END

```

We have used DOUBLE PRECISION here because FORTRAN double precision is identical to Turbo Pascal's REAL data type if an 8087 is being used, and we would like to minimize our work.

Unfortunately, it is impossible to write a single assembler routine which can work for *both* Microsoft FORTRAN and Turbo Pascal (unless we get very tricky) since FORTRAN SUBROUTINES are FAR PROCs and Turbo PROCEDURES are NEAR PROCs. However, we can still write an assembler routine which requires only minimal changes to work with either language. We can put a constant into our assembler routine (using EQU) to indicate which language interface is desired, and we can use conditional assembly pseudo-ops in conjunction with this constant to assemble slightly different code in the two cases. Of course, the more compatible the language interfaces, the more similar the assembled code. [This is a very desirable way to write our procedure since it quite often happens that the mere *existence* of a useful routine that interfaces to a certain language forces us to stick to that language forever -- no matter how horrible and obsolete it becomes. Witness (or witless) the vast commercial, government, and scientific commitment to COBOL and FORTRAN. While these languages (no doubt) have some minor interesting features, the main argument in favor of their use is that there is such a large base of installed software.]

The first step is to choose compatible data types. For example, in our sample FORTRAN procedure (which we will use as the basis for our assembler routine), we have chosen to use REAL\*8 since we know that it is available in both FORTRAN and Turbo. The second step is to make the argument passing compatible. This can be done by declaring all of the Turbo Pascal arguments to be *variable* parameters -- since variable parameters are passed by pushing the segment:offset of the variable's address onto the stack, just as FORTRAN does. Moreover, this is especially easy in Turbo (as opposed to standard Pascal), since there is a "typeless" variable parameter declaration in which we don't even need to specify the variable types:

```
procedure dotpro(var array1,array2,n,result); external 'dotpro.bin';
```

All this being done, our sole problem is the NEAR vs. FAR PROC problem.

Here, then, is an assembler routine taking these ideas into account. It can be interfaced to FORTRAN if the constant LANGUAGE is changed to 2, and it can be interfaced to Turbo (with the external declaration shown above) if LANGUAGE is 0:

```

        .8087
        public  dotpro
code    segment 'code'
        assume  cs:code

; Turbo Pascal if the following is zero, FORTRAN if two.
language equ    0          ; 0 or 2.
        if     language
dotpro  proc    near          ; Turbo Pascal.
        else
dotpro  proc    far          ; Microsoft FORTRAN.
        endif
result  equ     dword ptr [bp+4+language]
n       equ     dword ptr [bp+8+language]
array2  equ     dword ptr [bp+12+language]
array1  equ     dword ptr [bp+16+language]

        push   bp
        mov    bp,sp
        push   ds
        push   es
; Get N into CX:
        lds   bx,n           ; now DS:BX points to N.
        mov   cx,[bx]       ; now CX contains N.
; Initialize 8087:
        fldz                   ; load zero into 8087.
; Now, prepare DS:SI to point to array1 and ES:DI to point to array2.
        lds   si,array1
        les   di,array2
; The main loop:
again:  fld   qword ptr [si]   ; get element of array1.
        add   si,8           ; update pointer.
        fmul  qword ptr es:[di] ; multiply by element of array2.
        add   di,8           ; update pointer.
        fadd                   ; add to running sum and pop.
        loop  again          ; repeat until done.
; Store result:
        lds   bx,result       ; DS:BX => result.
        fstp  qword ptr [bx]  ; store it.
; wait to make sure:
        fwait
        pop   es
        pop   ds
        pop   bp
        ret   16
dotpro  endp
code    ends
        end

```

In actual tests, this routine performs quite well. I have used it in both FORTRAN and Pascal, and have compared it to equivalent routines written in those languages. Here is benchmark of the three languages, all using the 8087 coprocessor:

| <u>Time to Take the Dot-Product of Two REAL*8 4000-vectors</u> |              |
|--|--------------|
| <u>LANGUAGE</u>  | <u>TIME</u>  |
| assembler  | 0.26 seconds |
| Microsoft FORTRAN  | 0.61 seconds |
| Turbo Pascal   | 2.06 seconds |

Clearly, there is some potential for interfacing 8087 number-crunching routines to "slow" languages like Microsoft FORTRAN and Turbo Pascal. In this test, which involves 8000 floating point operations, the 8087 runs at about 0.03 Mflops (1 Mflop= 1 million floating point operations per second) in the assembler routine, very close to its theoretical maximum throughput (sometimes quoted as 0.05 Mflops).

Final Words

This is the end of the course, and I hope you have gotten something out of it, even if it has been necessary to do a lot of work and cover a lot of details. Unfortunately, as I have stated so often, so much of learning assembly language is just practicing and learning from mistakes, that it is necessary to do a lot of programming. Also, since most of the reason for doing assembly language at all on the IBM PC is simply because we want to get at the low-level features of the machine, we needed to cover those low-level features. Unfortunately, the information needed to do many of these things is spread across many different books (almost none of it being in our own textbook), so I felt that it was necessary to present a lot of information, even though we barely scratched the surface of most topics. However, as you are probably aware, full mastery of every topic is not needed to get a lot of use out of the machine.

Final Final Words

The final projects must be turned in by 8:00 pm., Wednesday, the 14th of August (1985). Please make some effort to insure that the program works and is bug-free.

\*\*\*\*\*  
**BURKEY'S LAW:**           A program which has not been tested is a program which does not work.  
**COROLLARY:**            Most people do not test their programs.  
 \*\*\*\*\*

STUDENT AND INSTRUCTOR RESULTS  
 SORTING TIMES (PLUS OR MINUS 0.06 SECONDS) FOR VARIOUS  
 ALGORITHMS AND FILES OF VARIOUS SIZES

| <u>STUDENT ID</u> | <u>SORTING ALGORITHM</u> | <u>50 LINES<br/>1K BYTES</u> | <u>400 LINES<br/>13K BYTES</u> | <u>850 LINES<br/>40K BYTES</u> |
|-------------------|--------------------------|------------------------------|--------------------------------|--------------------------------|
| RSB               | QUICK SORT               | 0.05                         | 0.39                           | 0.77                           |
| 1392              | SHELL SORT               | 0.05                         | 0.60                           | 1.32                           |
| RSB               | SHELL SORT               | 0.06                         | 0.66                           | 1.38                           |
| 8012              | HEAP SORT                | 0.00                         | 0.66                           | 1.31                           |
| RSB               | HEAP SORT                | 0.05                         | 0.71                           | 1.26                           |
| 4931              | SHELL SORT               | 0.05                         | 0.82                           | 1.71                           |
| 7758              | INSERTION SORT           | 0.05                         | 4.40                           | 14.39                          |
| 8868              | INSERTION SORT           | 0.05                         | 4.50                           | 14.94                          |
| 7003              | INSERTION SORT           | 0.06                         | 4.50                           | 15.00                          |
| 2939              | INSERTION SORT           | 0.05                         | 4.51                           | 14.94                          |
| 6192              | INSERTION SORT           | 0.05                         | 4.61                           | 15.54                          |
| 2377              | INSERTION SORT           | 0.05                         | 4.62                           | 15.55                          |
| 1173              | INSERTION SORT           | 0.05                         | 4.67                           | 15.55                          |
| RSB               | INSERTION SORT           | 0.05                         | 4.99                           | 16.58                          |
| 8642              | INSERTION SORT           | 0.05                         | 5.05                           | 17.25                          |
| RSB               | SELECTION SORT           | 0.17                         | 9.00                           | 34.55                          |
| 8540              | SELECTION SORT           | 0.22                         | 11.37                          | 45.48                          |
| RSB               | BUBBLE SORT              | 0.22                         | 21.25                          | 67.67                          |
| DOS               | SPOOKY, ISN'T IT?        | 1.00                         | 32.00                          | 211.00                         |

TESTING THE ASSEMBLER DOT-PRODUCT ROUTINE

Two results are computed by each program. *R* is the dot-product as given by the assembler routine and *S* is the dot-product computed by the high-level language.

```
{ Pascal version: set LANGUAGE=0 in DOTPRO.ASM, then use MASM, LINK,
  and EXE2BIN to create DOTPRO.BIN. Declaration of the routine: }
procedure dotpro(var array1,array2,n,result); external 'dotpro.bin';

{ ===== Main program: ===== }
const max=4000;
var r,s:real; i,n:integer; v1,v2:array[1..max] of real;
begin
{ ===== First, fill the arrays with known data: ===== }
  for i:=1 to max do begin v1[i]:=i; v2[i]:=i+1 end;
{ ===== Compute first in Pascal to make sure of the answer: ===== }
  writeln('Now computing in Pascal. ');
  s:=0;
  for i:=1 to max do s:=s+v1[i]*v2[i];
{ ===== Next, do the computation in assembler: ===== }
  writeln('Now computing it in assembler. ');
  n:=max;
  dotpro(v1,v2,n,r);
{ ===== Finally, compare the two results: ===== }
  writeln('Pascal gives ',s);
  writeln('Assembler gives ',r)
end.
```

**C FORTRAN VERSION: SET LANGUAGE=2 IN DOTPRO.ASM AND THEN ASSEMBLE  
C TO GET DOTPRO.OBJ. NO DECLARATION OF THE ASSEMBLER ROUTINE NEEDED.**

```
C ===== MAIN PROGRAM: =====
$NOFLOATCALLS
$STORAGE:2
$LARGE V1,V2
      PARAMETER (MAX=4000)
      IMPLICIT REAL*8 (P-Z)
      DIMENSION V1 (MAX), V2 (MAX)
C ===== FIRST, FILL THE ARRAYS WITH KNOWN DATA: =====
      DO 10 I=1,MAX
          V1 (I)=I
10          V2 (I)=I+1
C === FIRST, COMPUTE IN FORTRAN TO BE SURE OF THE RESULT: =====
      WRITE (*,*) 'NOW COMPUTING IN FORTRAN'
      S=0.0D0
      DO 20 I=1,MAX
20          S=S+V1 (I)*V2 (I)
C ===== NEXT, COMPUTE IT IN ASSEMBLER: =====
      WRITE (*,*) 'NOW COMPUTING IN ASSEMBLER'
      N=MAX
      CALL DOTPRO (V1,V2,N,R)
C ===== FINALLY, COMPARE THE TWO RESULTS: =====
      WRITE (*,*) 'FORTRAN GIVES ',S
      WRITE (*,*) 'ASSEMBLER GIVES ',R
      STOP
      END
```

