ECE 291

Computer Engineering II

Fall 2001 Laboratory Notes

The ECE 291 Documentation Project

Department of Electrical and Computer Engineering

University of Illinois at Urbana-Champaign

Edited by

Peter L. B. Johnson

Fall 2001 Laboratory Notes

by The ECE 291 Documentation Project, Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign

Edited by Peter L. B. Johnson

Published 2001

Revision History

Revision 1 1992 Revised by: TM
Manual reconstructed
Revision 2 1994
Complete revision of manual format and content
Revision 3 1995
Additional material added: instruction summary, string instructions, Mode 13h VGA, Ethernet, C programming, PCX files
Revision 4 1997 Revised by: JWL
Revised for new lab computers, software, resources
Revision 5 2001 Revised by: PLBJ
Complete revision for NASM, new class organization, lab computers, and tools. Converted to DocBook format.

Table of Contents

List of Tables

List of Figures

List of Examples

Preface

1 Acknowledgments

The contents of this manual are a compilation of various references, guides, and tutorials taken from many sources and written by several independent authors. Thanks and appreciation to the following individuals for the use of their material in this manual:

Professor Gernot Metze	Michael Thiems	Jason Gallicchio
Professor Michael C. Loui	Doug Stirrett	Julian Hall
Professor W. Kent Fuchs	Allan M. Krol	Simon Tantham
Professor John W. Lockwood	Brandon Long	Frank Kotler
Tom Maciukenas	Jeff Stahl	Stephen Silver
Eric S. Meidel	Joseph Gebis	
David Eprim Pearah	Peter Johnson	

Indeed this list is not extensive, as some of the material comes from authors unknown at the time of printing of this manual. It is hoped that those individuals who have not been mentioned but have contributed will feel equally appreciated as those listed above.

Chapter 1
Introduction to the Course

Welcome to ECE 291, Computer Engineering II. This chapter will acquaint you with the various aspects of this course, from details on grading to instructions on how to demo machine problems.

1.1 Machine Problems

1.1.1 Requirements and Grading Procedures

Point Totals

Your machine problems will be graded according to the following formula:

$$\text{MP Grade} = \text{Functionality Points} - \text{Score Modifiers} \pm \text{Penalty/Bonus}$$

Functionality Points

You are expected to provide an original, fully functional solution to all machine problems. Due to the complexity of MP2 and later machine problems, you will occasionally be supplied with additional library subroutines which implement all or some portion of the assignment. These subroutines are to assist you in writing and debugging your code, but are not intended to be handed in as part of your solution. If your solution requires the use of one or more library subroutines, you will be penalized accordingly for using them. The penalty for each library subroutine will be indicated in the assignment. There is no penalty for the use of the subroutines `kbdin`, `kbdine`, `dspout`, `dspmsg`, `dosxit`, `ascbin`, and `binasc` (all present in LIB291). We may announce additions to this list of free subroutines.

Score Modifiers

Points may be subtracted from the functionality score based on your submitted source code. The graders look for the following items in the source code:

- Comments
- Technique and Style
- I/O Specifications
- Modularity

Penalty/Bonus

Bonus points are awarded for turning in a machine problem early. As the lab contains far fewer machines than there are students in the class, long waits to use a computer or demo a machine problem can be avoided by starting and finishing assigned machine problems early. To encourage this, extra points are awarded for each working day (Mon-Fri) the assignment is early. The amount of extra points awarded will be shown on the online grading sheet. Likewise, penalty points are subtracted for every working day an assignment is late.

Comments

There are two main kinds of comments we are looking for, *line comments* and *procedure comments*. Good line comments supply additional information or context for a given assembler instruction. They explain something that might not be obvious, rather than merely echoing the instruction itself.

Procedure comments, also known as *header comments*, are required for each and every procedure in your program. Foremost, they describe what the procedure is supposed to do. They should also describe which registers and/or memory locations hold the procedure's input values, which registers and/or memory locations hold the output values when the procedure exits, and which registers are changed as part of the normal execution of the procedure. Excellent examples of proper commenting style are available in your laboratory notebook. Take advantage of them.

Style

Good style implies a number of practices have been followed in your program: (a) code is not repeated, (b) commonly used sequences are separated into independent subroutines, and (c) the execution path of your program does not look like "spaghetti."

Generally, if you have a significant series of instructions repeated in your program, you probably need to rethink how you approached the problem. Look for a way to consolidate the instructions so that you don't have to repeat them; this almost always reveals a solution that is easier to implement and debug.

If you just can not consolidate the instructions, consider making them a separate, independent subroutine. Then you just `call` the subroutine from multiple places in your program.

Finally, any procedure (this includes `main`) should probably resemble the flowchart of the same task. Execution starts at the top, and flows generally downward. If you have lots of branches every which way, you might want to make it easier for human readers and graders (*hint!*). Some advanced microprocessors rely on certain code patterns for optimum performance, and if you follow good programming style, your programs will be more efficient (employers like this!).

I/O Specifications

You will be given specifications for some of the procedures you write for later machine problems. You must follow these specifications, even though you will write the entire program by yourself. These specifications will include which registers pass values into the procedure, which registers hold values on exit from the procedure, and which registers must remain unmodified.

Modularity

Modularity and style are closely related. A modular program will use subroutines in the obvious places; see above. A modular program will also use loops instead of in-line code, and tables instead of a long series of compare and branch instructions. All these practices will make your program much easier to write and debug, and will make your life simpler if you ever need to modify your program. A modular program rarely encounters the problem of a branch-out-of-range; a modular procedure usually fits on the screen, and certainly fits completely on a printed page. If your procedures are longer than one page, think carefully about what you wrote.

1.1.2 Demonstration

When you hand in machine problems, you are responsible for meeting the following conditions:

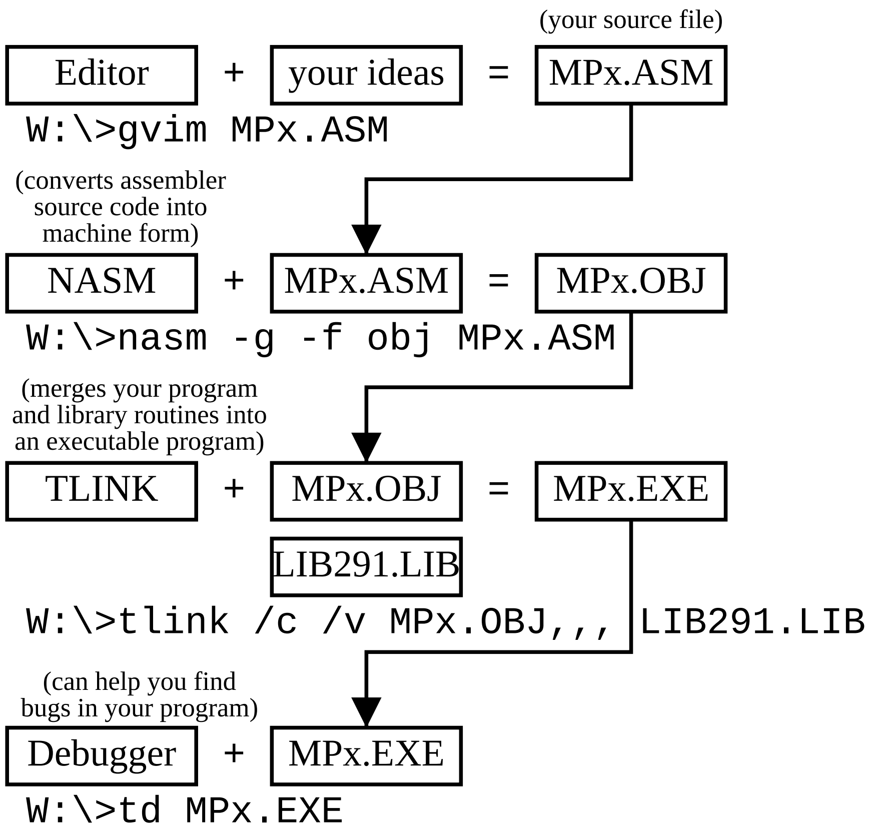
- You must demonstrate your correctly working program to a TA or instructor in the lab, and be able to answer reasonable questions about how you did it.
- You must provide an electronic listing of the `.ASM` file for your program.
- The TA or instructor must be able to copy your `.ASM` file to the official Web-based hand-in, assemble and link them with our copies of `NASM` and `TLINK`, and get the same program.

The lab computers have Intel Pentium III processors running at 1 Ghz. If you own or have access to other PC compatible computers, you will probably be able to do a lot of the lab work outside of the lab. However, your program must eventually be demonstrated in the lab, and there are often subtle differences between PC compatible computer models. If you do your work outside of the lab, make sure you allow yourself enough time to test and modify your program so it runs in the lab. Also remember that the staff will be unable to answer questions about computers outside the lab.

The same warning applies to your choice of editor, debugging tools, and so forth. You may use whatever PC tools you feel most comfortable with, but your program has to run in the lab using our hardware and software. Please note there may be questions on the homework and/or exams which refer to the tools in the lab; you will be expected to be somewhat familiar with them, even if you choose other tools for most of your work.

1.1.3 The Big Picture

Creation of an assembly language program requires several steps, from the initial ideas to the finished executable. The following flow chart illustrates the development of an assembly-language program.


(your source file)
Editor
+
your ideas
=
MPx.ASM
W:\>gvim MPx.ASM
(converts assembler source code into machine form)
NASM
+
MPx.ASM
=
MPx.OBJ
W:\>nasm -g -f obj MPx.ASM
(merges your program and library routines into an executable program)
TLINK
+
MPx.OBJ
=
MPx.EXE
LIB291.LIB
W:\>tlink /c /v MPx.OBJ,,, LIB291.LIB
(can help you find bugs in your program)
Debugger
+
MPx.EXE
W:\>td MPx.EXE

1.2 WWW Page

ECE 291 also has a WWW home page. This page contains several FAQ files relevant to ECE 291, lists relevant newsgroups (including our newsgroup uiuc.class.ece291), contains ECE 291 humor, course specific information, as well as other programming resources. The homeworks and MPs are released and graded online. It also contains useful information on the staff. The URL address for the ECE 291 home page is:

http://courses.ece.uiuc.edu/ece291/

You may access this page using any web browser, including Internet Explorer, Netscape, and Mozilla.

Internet Explorer is available on the machines in the lab.

Chapter 2
Using the PC

Before a program can be successfully written, one must have a good understanding of the functions and operations of the PC in general. This chapter will describe the operation of the PC as it applies to the creation of programs for this course.

2.1 Microsoft Windows / DOS

Microsoft Windows is the program that controls other running programs, supervises and executes the I/O operations (receiving characters from the keyboard or displaying characters on the screen, for instance), and manages the files on the diskettes. When the computer is first powered up (“boot” or “bootstrap”), you commence executing in the Windows system. By means of a dialogue of commands between the system and the user, conducted via the keyboard, mouse, and display, a program may be edited, assembled, and run. You may also create, rename, and delete files on disks or diskettes.

Microsoft DOS was the first Disk Operating System used for IBM and IBM-compatible PCs. Unlike Windows, it ran in text mode and only one program could be open at a time. Commands were entered at the DOS prompt using a keyboard. An extended DOS-compatible prompt is still available under Windows, and it will be used extensively in ECE 291 for assembling and running the MPs written in class. In Windows 2000, this DOS-compatible prompt is sometimes called the Command Prompt.

2.1.1 Entering Commands

DOS displays a prompt to indicate that it is waiting for you to enter a command. The prompt shows the current disk drive and directory name. For example,

```
W:\MyFiles>_
```

The underscore (“_”), called the cursor, indicates the location on the screen that text typed on the keyboard will appear. For example, the following command may now be entered:

```
W:\MyFiles>gvim example.asm
```

This command tells DOS to edit the file EXAMPLE.ASM using the VI editor. VI is the command name, and the filename EXAMPLE.ASM is an *argument* to the command. (Not all DOS commands take arguments).

2.1.2 Files and Filenames

DOS manages the files stored on your diskettes and the hard disk. This includes *system files* as well as your own program files. You can manage your files using the same types of functions you use when editing text—add, delete, etc. As your disks fill up, it's necessary to delete the older files that are no longer needed to free up space. As the network drives accessible from the lab machines have many gigabytes of space available, it should not be necessary to delete any class-related files.

Each file is identified by a file specification *`filespec`*, which consists of these four parts:

`filespec` = [*`disk drive name`*:][*`path`*]*`filename`*[.*`extension`*]

For example:

```
W:\MyFiles\example.asm
```

The *disk drive name* (W:) gives the name of the disk drive containing the file. This can be a hard drive, a network drive, or a diskette drive. If you do not supply a disk drive name, the current one (shown in the DOS prompt) is used. For much of your work, the disk drive need not be specified. The *path* is a series of directory names which gives the specific directory where your file is contained. For example, if the path was "`\MyFiles\MP0\`", that would mean the directory `MyFiles` contains the directory `MP0`, which in turn contains the file. You may decide not to use directories at all, in which case the path can be omitted. However, if you are using directories and you omit the path, DOS will use the current path (shown in the DOS prompt). The filename is the identifier for the file, and the extension gives the type of the file. While Windows allows the filename and extension to be any length, some older DOS programs only allow a filename to be a maximum of eight characters in length, and an extension to be a maximum of three characters in length. The *Glossary* lists some standard extensions. In many contexts (e.g., DOS commands for program assembly, linking, or execution), the extension need not be specified if it is a standard extension.

2.1.3 Wildcards

Wildcards are special characters which can be used in a file specification to simplify the specification of several related files. DOS allows the use of two special characters "*" and "?" in place of specific characters as illustrated below:

`W:*.COM`

any file on Disk W having extension "COM". Hence, "*" stands for any *sequence of characters*.

`W:ABC.?OM`

any file on Disk A with name "ABC" and extension ending in "OM". Hence, "?" stands for any *single character*.

These can be used in combinations, as in:

`W:A??C*.*`

any file whose filename starts with an A, then two more characters, then a C.

The "?" character may be used anywhere in the filename and extension, but the "*" character causes any characters following it *in the same field* to be ignored (i.e.,"`A:AUT*C.BAT`" is equivalent to "`A:AUT*.BAT`").

2.1.4 Useful DOS Commands

Following is a brief list of some useful DOS commands and their meanings. In some instances, examples are given in order to clarify their usage. [..] indicates optional elements in the command.

`type` {*`filespec`*}

Display the named file on the display.

`type` {*`filespec`*} | more

Cause output to pause at page breaks. Any command can be followed by `| more` to cause its output to pause at page breaks. However, `more` cannot be used as a command itself.

`dir` [*`filespec`*]

List the files in the current directory matching *`filespec`* (which may contain wildcards as described in Section 2.1.3), or all the files if *`filespec`* is not specified.

`cd` {*`path`*}

Change the current directory to *`path`*.

`mkdir` {*`name`*}

Create a new directory in the current directory called *`name`*.

`rmdir` {*`name`*}

Remove the named directory (must be empty).

`del` {*`filespec`*}

Delete the files matching *`filespec`* (may contain wildcards).

`copy` {*`fromfile`*} {*`tofile`*}

Copy *`fromfile`* to make *`tofile`*. If the *`filename.ext`* portion of *`tofile`* is not specified, the new copy has the same name as the old one.

`ren` {*`fromfile`*} {*`tofile`*}

Rename a file from *`fromfile`* to *`tofile`*.

`[`*`drive`*`:][`*`path`*`]`*`filename`*

Execute the named file (assumed to be in assembled and linked form).

2.1.5 Batch files

DOS *batch files* are a very useful way of automating common actions. A batch file is a file with an extension of ".BAT". It contains DOS commands and can be executed just like an .EXE file (by typing the filename without the extension). As a simple example, create a file called GO.BAT which contains the following two lines:

```
nasm -g -f obj mp0.asm
tlink /c /v mp0, , , lib291
```

Now type **`GO`** at the DOS prompt. DOS will execute the batch file, assembling and linking your program in the process. You can also specify input arguments to your batch files. An argument is represented within a batch file by the term "%*n*", where *n* is a decimal digit. "%1" represents the first argument, "%2" represents the second, and so on. Using this feature, you could customize the batch file so it takes the MP number as an argument:

```
nasm -g -f obj mp%1.asm
tlink /c /v mp%1, , , lib291
```

This batch file could be called with **`GO 0`** to assemble and link MP0, or **`GO 1`** to assemble and link MP1, and so on. Arguments can be any string of text, not just numbers, so you could also input the library name with the following batch file:

```
nasm -g -f obj mp%1.asm
tlink /c /v mp%1, , , %2
```

This could be called with **`GO 1 LIB291`**, for example.

A better way to automate the assembling and linking is by using a tool called Make that conditionally runs programs based on the last modified date of the source and output files. All of the machine problems in ECE 291 will be distributed with a Makefile to make the build process easier.

2.2 Assembling and Linking Files

A program may be assembled and linked using the following command-line statements:

`nasm` [-g] {-f *`obj`*} [-o *`outfile`*] {*`filename`*} [-l *`listfile`*]

Assemble the named file.

- The `-g` option enables debugging output (so the original source code is visible in the debugger).
- The `-f` option specifies the output object format (obj is standard for DOS programs).
- The `-o` option specifies the output filename. This is optional, as NASM can infer the output filename from the object format and source filename.
- The `-l` option specifies the list output filename. If this option is not specified, a list file will not be written.

`tlink` [/c] [/v] {*`objfile`* ...} , [*`exefile`*] , [*`mapfile`*] , [*`libfile`* ...]

Link together several object files and libraries into a single executable.

- The `/c` option makes the link case-sensitive. As NASM is case-sensitive, this should normally be enabled.
- The `/v` option includes debugging information in the executable.
- *`objfile`* is an input obj file.
- *`exefile`* optionally specifies the filename of the output executable. If not specified, it defaults to the first object file listed with an exe extension.
- *`mapfile`* optionally specifies the filename of a map file. If not specified, a map file is not written.
- *`libfile`* is an (optional) library file.

Chapter 3
Text Editors

Text editors are specialized programs which allow you to create or edit a file by typing in the contents using the keyboard (and sometimes the mouse). Most text editors have many special features which allow text to be created more easily and quickly using functions such as copying, deleting, and moving blocks of text in the file. There are numerous text editors available on the lab machines—of which two of the most frequently used are described in the following sections.

3.1 VIM

The VIM text editor on the PCs is an improved version of the editor “vi”, one of the standard text editors on UNIX systems. VIM has many of the features that you should expect a programmer’s editor to have: Unlimited undo, syntax coloring, split windows, visual selection, a graphical user interface (with menus, mouse control, scrollbars, text selection, and the like), and much more. To edit a file using the GUI version of VIM, type:

`gvim` [*`file`*...]

3.1.1 Entering Commands and Text

VIM is a mode-oriented editor. Initially, when VIM starts, it is in *command* mode. In command mode, every key typed is interpreted as a command, such as a command to delete a character or to move the cursor.

From command mode, pressing **i** (or **a**, or another insert command) puts VIM into *insert mode*. In insert mode, each character typed is inserted into the file. Pressing the backspace key deletes the previous character. From insert mode, pressing the **ESC** key puts VIM back into command mode. It’s also possible to use the cursor keys to move the cursor around in insert mode. If you do not know which mode VIM is in, simply press the **ESC** key to make sure VIM enters the command mode. To exit VIM (and save the current file), type **`:wq`**.

3.1.2 Summary of Commands

Following is a summary of the commands VIM accepts while in command mode.

Moving Around the File

Pressing one of the arrow keys moves the cursor one position in the direction of the arrow. Pressing the **PgUp** key moves backward in the file by a full screen; pressing the **PgDn** key moves forward in the file by a full screen.

`h`	— Move cursor LEFT
`j`	— Move cursor DOWN
`k`	— Move cursor UP
`l`	— Move cursor RIGHT
`b`	— Move cursor back one word
`w`	— Move cursor forward one word
`0`	— Move cursor to beginning of current line
`$`	— Move cursor to end of current line
Enter	— Move cursor to beginning of next line
`^`	— Move cursor to beginning of current line
`H`	— Move cursor to top line of screen
`M`	— Move cursor to middle line of screen
`L`	— Move cursor to last line of screen
CTRL-d	— Move down (forward) half a screen
CTRL-u	— Move up (backward) half a screen
CTRL-f	— Move forward a full screen
CTRL-b	— Move backward a full screen
`{`	— Move to previous paragraph
`}`	— Move to next paragraph
`1G`	— Move to beginning of file
`G`	— Move to end of file
***`xx`*`G`**	— Go to the *`xx`*th line of the file

Inserting Text

`i`	— Enter insert mode to insert text before cursor
`I`	— Enter insert mode to insert text at beginning of line
`a`	— Enter insert mode to append text after the cursor
`A`	— Enter insert mode to append text at end of line
`o`	— Open a line below the current line, and enter insert mode
`O`	— Open a line above the current line, and enter insert mode

Changing Text

`J`	— Join next line to current line
`r`	— Replace one character with a single character
`R`	— Replace string of text (overstrike)
`x`	— Delete one character
`dw`	— Delete one word forward
`db`	— Delete one word backward
`dd`	— Delete current line
`12dd`	— Delete 12 lines
`D`	— Delete from cursor to end of line

Copying Text

`yy`	— Yank the current line to a buffer (does not delete the line)
`12yy`	— Yank 12 lines (including current line) to the buffer
`p`	— Put lines from buffer below current line
`P`	— Put lines from buffer above current line
`:r` ***`filename`***	— Read lines from file *`filename`* and insert at cursor position

Searching for (and Replacing) Text

`/ABC`	— Search for next occurrence of “ABC”
`?ABC`	— Search for previous occurrence of “ABC”
`n`	— Repeat last search
`:%s/`*`regexp`*`/ABC/g`	— Replace all text matching the regular expression *`regexp`* with ABC

Undoing (and Redoing) Commands

`u`	— Undo last command
CTRL-r	— Redo last command

Saving Files and Exiting VIM

`:w`	— Save the current file without leaving (do this periodically)
`:wq`	— Replace old file and exit VIM
`ZZ`	— Shorthand for **`:wq`**

`:q`	— Exit VIM. If the file has been modified and not subsequently saved, VIM will prompt to save the file.
`:q!`	— Exit VIM without replacing the old file

3.2 Emacs

No discussion of VI or VIM is complete without also mentioning Emacs. VI and Emacs are the two most popular editors on the UNIX platform, and many heated debates have been held over which one is “best”. Choose whichever one you feel most comfortable with, as being comfortable with the editor you use will help you code more efficiently.

Emacs is a text editor and more. At its core is a Lisp derivative with extensions to support text editing, called elisp. Just like VIM, it has syntax coloring, split windows, visual selection, a graphical user interface (with menus, mouse control, scrollbars, text selection, and the like), and much more. There are also a large number of extensions which add other functionality (most of which aren’t installed on the lab machines). To edit a file with Emacs, type:

```
emacs [file...]
```

3.2.1 Entering Commands and Text

Unlike VIM, Emacs is not a mode-oriented editor. Insert mode is always active, and commands are triggered only by combinations of modifier keys (such as **CTRL** and **ALT**) and other keys. Sometimes a series of key combinations need to be entered to perform an action: for example, **CTRL-x** followed by **CTRL-c** exits the editor (and prompts for modified files to be saved).

3.2.2 Summary of Commands

Following is a summary of the commands Emacs accepts. To conserve space in this summary, **CTRL** is designated by “C”, **ALT** is designated by “A”, and **META** is designated by “M”.

Moving Around the File

Pressing one of the arrow keys moves the cursor one position in the direction of the arrow. Pressing the **PgUp** key moves backward in the file by a full screen; pressing the **PgDn** key moves forward in the file by a full screen.

`M-b`	— Move cursor back one word
`M-f`	— Move cursor forward one word
`C-a`	— Move cursor to beginning of current line
`C-e`	— Move cursor to end of current line

`M-<`	— Move to beginning of file
`M->`	— Move to end of file
`M-x goto-line`	— Prompt for a line number and move to it

Erasing Text

`J`	— Join next line to current line
`DEL`	— Delete one character backward
`C-d`	— Delete one character forward
`M-d`	— Delete one word forward
`M-DEL`	— Delete one word backward
`C-a C-k C-k`	— Delete current line
`C-k`	— Delete from cursor to end of line

Searching for (and Replacing) Text

`C-s ABC`	— Search for next occurrence of “ABC”
`C-r ABC`	— Search for previous occurrence of “ABC”
C-M-% *regexp* ABC	— Replace all text matching the regular expression *regexp* with ABC

Undoing Commands

`C-x u`	— Undo last command

Saving Files and Exiting Emacs

`C-x C-s`	— Save the current file without leaving (do this periodically)
`C-x C-s C-x C-c`	— Replace old file and exit Emacs
`C-x C-c`	— Exit Emacs. If the file has been modified and not subsequently saved, Emacs will prompt to save the file.

3.3 Other Editors

There are additional editors available for use on the PCs in the lab. However, none of them are quite as advanced or full-featured as VIM or Emacs. We recommend using either VIM or Emacs for writing code.

3.3.1 Notepad

Notepad is the text editor included with Windows. It has copy and paste and search functions, but no syntax highlighting or other advanced features.

3.3.2 Visual Studio Editor

The Visual Studio development suite has a built-in editor that can be used to edit assembly language programs. However, although it does preform limited syntax highlighting, it’s written for MASM syntax, not the NASM syntax we use in ECE 291.

Chapter 4
Debugging Tools

What does a debugging program do and how is it useful? These questions are better explained by example, but to give a quick answer, a *debugging program* allows you to monitor control of a program during program execution. What does that mean? Well, let s give a quick example. Using the debug program (hereafter we will refer to it as the "debugger") you can tell the program to execute until a specific statement is reached; upon reaching this statement the debugger allows you to look at and change values of different variables in the program and also the registers used by the PC (of course the debugger will allow you to do much more than that). The usefulness of this ability to watch your program unfold in great detail will become apparent later on.

Before showing how to use the debugger, a word of encouragement should be offered. In our experience with assembly language and programmers, we have found two types of programmers who use a debugger: first of all, those who just naturally like this kind of programming tool and pick it up easily, and second, those who have done a considerable amount of programming and who have in the process been forced to learn how to use a debugger, and much to their surprise have found it to be extremely useful. We have tried to write the following debugger "tutorial" to create a third category of people who use the debugger—those who have not had much experience programming but have had a good description of how to use a debugger. We hope that you will take the time to learn how to use it, in order to save time debugging your programs later on. And one more word of encouragement: at first, like most people, you may shy away from using the debugger. It may seem too complicated and time-consuming to run part of the program and then to go into memory to try to figure out what state the computer is in (the values of the registers and variables), and whether it is in the state you want it to be in. But trying to fix your program without the debugger is much like trying to fix a car without being able to look under the hood! It will take you a few minutes to learn how the debugger works, but used properly, it will save you hours in the lab. So don t be tentative about "looking inside" the computer while executing your program; it is not so complicated as you might think. If you have never used a debugger before, it will be useful to go through a simple program using the debugger before tackling larger programs. The first machine problem in ECE 291, MP0, will help you learn how to use the debugger.

There are four basic steps in debugging:

1. *Discovering the Bug*. Is there one? It is not always obvious that you have a bug. (Why?)
2. *Isolating the Bug*. Where is it? Locate the part(s) of the code that is causing the bug.
3. *Finding the Bug*. What exactly is wrong with the buggy code fragment?
4. *Fixing the Bug*. How should the buggy fragment be rewritten?

The debugger gives you the tools to help in all these steps. E.g., you can run the code in a step-by-step fashion or until some *breakpoint* you set in the code. In between these execution steps, you can examine memory and CPU state (variables, registers, flags, and stack).

4.1 Turbo Debugger (TD)

Turbo Debugger is a window-oriented mouse-driven debugging tool. To start Turbo Debugger (oftentimes referred to as simply TD), type the following at the DOS prompt:

```
td[progname[.exe]]
```

4.1.1 Mechanics of Using TD

Overview

The main TD window shows a standard (pull-down) menu bar at the top. The menu bar lists the following menus: File, Edit, View, Run, Breakpoints, Data, Options, Windows, Help. The status bar at the bottom of the TD window contains helpful information about the current context. For instance, you often see a list of all the functions associated with the function keys **F1** to **F10**. For instance, **F9** is the “Run” command. That means these functions are available in the current context.

Windows

Within the Main TD window we can have a variety of (sub)windows. At any moment, several of these windows may be active. Each active windows is numbered (1,2,3, etc.) and this number is displayed on the upper right hand corner of window. Below, we will go into some of these windows and discuss how to manage them. Among the active windows, one of them is the current top window. The top window has a little green rectangle at its upper left corner. You can close this window by clicking on this little green rectangle. The function key **F6** (called “Next”) steps through the active windows, allowing each of them to be “top window” in turn.

Online Help

We already mentioned the Help (pulldown) menu. But there are more “immediate” or context-sensitive help available:

Status Bar

We noted that status bar usually shows the list of functions associated with the 10 function keys. But if you hold down the **Alt** key, the status bar will show the functions associated with **Alt**+Function Keys.

E.g., **ALT-F5** is the “User Screen” Function. Try this! This is useful if you need to see any output from your executed code. To get back from the User Screen, press any key.

If you hold down the **Ctrl** key, you will see the **Ctrl**+key functions.

E.g., **CTRL-I** allows you to inspect the variable that the cursor is currently pointed at.

F1 key

This key opens a help window containing information about the current top window, with further subtopics to choose from.

SpeedMenu

This can be invoked with a right mouse click at any time. In most windows, it will display a list of options suitable for that window.

4.1.2 Some Pulldown Menus

View

The types of windows available are listed under the View menu. See Section 4.1.3 for descriptions of the various types of windows available.

Window

The Window menu helps you manage the various windows. This menu is divided into two halves (separated by a horizontal line): the top half contains commands such as Zoom (**F5**), Next (**F6**), Next Pane (**Tab**), etc. The bottom half is a list of the currently active windows.

Run

Windows are for watching. But for action, you need to execute your code. For this, the Run menu provides several modes of execution:

- Run (**F9**), i.e., until program terminates (or until the next breakpoint, see Section 4.1.4).
- Trace Into (**F7**), i.e., single stepping, one instruction at a time. What constitutes a single step depends on which the current “top window”. If the top window is the Module window or if you use **F7**, then a single line of source code is executed. If the top window is the CPU window or if you use **ALT-F7**, then a single machine instruction is executed. If the current source line contains a function call, TD traces into the function (assuming it was compiled with debug information). Some machine instructions, however, cause multiple instructions to be executed include: `CALL`, `INT`, `LOOP`, etc.
- Step Over (**F8**). This is like “Trace Into” except that when the instruction pointer is at a function call, then the entire function is executed and you are next placed at the statement following the function call.
- Animate. Similar to run until terminate, except it pauses between machine instructions to allow you to catch what is happening.
- Restart. Moves the instruction pointer back to the first instruction.

4.1.3 TD Windows

The Regs Window

The Regs window displays the contents of all the processor registers as well as the CPU flags. It is possible to view either the 16-bit registers or the 32-bit registers. Various options can be accessed by using the SpeedMenu (see Section 4.1.1.3). As the program is being stepped through, register and flag changes are highlighted in the window.

The Dump Window

This is a hex display of an area in memory. The leftmost path of each line shows the starting address of the line (e.g., `DS:0000`). In the default display format (byte format), you see 8 bytes per line, and to the right of these 8 bytes are their representation in the IBM extended character set (which is an extension of the ASCII set). You can use the Goto command in the SpeedMenu to examine variables (e.g., Goto `Input`, assuming you have defined the variable “Input”).

The Module Window

The Module window displays the program source code if debugging information is available. The **F8** key steps through each line of code. The **F7** key also steps through each line of code, but unlike **F8**, it also *traces* into procedure calls. **F2** sets a *breakpoint* in the code at the line where the cursor is at (see Section 4.1.2 for more information on stepping through the program and Section 4.1.4 for more information on breakpoints).

The CPU Window

The CPU window combines the Module, Dump, and Regs windows into a single window. It’s also less powerful and harder to use than the three separate windows, so it’s almost always better to use the specific windows rather than the combined CPU window.

4.1.4 Breakpoints

Breakpoints are a device to cause the computer to take specific actions at specific points in its execution. The user can define a breakpoint by specifying three pieces of information:

- The *location* where the breakpoint is set.
- The *condition* which allows the breakpoint to activate.
- The *action* that takes place when the breakpoint is activated.

The simplest kind of breakpoint is one that (a) is associated with a specific instruction in the program, which (b) is always activated (condition is “always true”) and (c) pauses the program execution (action is “break”). These are called simple breakpoints. It is the default assumed by TD, and it should suffice for our purposes. Using this, you run the program at full speed until specific instructions, at which points you can examine the state of the memory and CPU.

How do you set simple breakpoints? Well, you only need to specify an instruction in the program. The simplest is to do this from within the Module window, or from the Code pane in the CPU window:

- First place the cursor at an executable line of code where a breakpoint is desired. (How do you tell if a line is executable?) You then left the 2 leftmost columns of line. Instead of left, you can also use **F2** (see the status line).

  Note: If the line already has a breakpoint, then this action removes that breakpoint. Hence this is also called the *toggle* action.

- If you use the Breakpoint⟶At menu option, you can also place a simple breakpoint at the current cursor position. However, since this has a pop-up Breakpoint option dialog box, you can also specify more complex types of breakpoints.

Breakpoint *addresses* must be entered when you use the keyboard to enter breakpoints. (You can see this in the Breakpoint option dialog box). These are the kinds of addresses you can specify:

- *`#number`* — for instance, #68 specifies a breakpoint in line 68 of your source code.

  Note: If you have several program modules, you need to preface the line number with the module name. E.g. #mp0#68 refers to line 68 in the mp0.asm module.

- Symbolic names—for instance, labels can be specified. If you have have a label called “repeat”, you can use that as an address.

How do you see all the current breakpoints? In the Breakpoints window, which can be activated using the View⟶Breakpoints menu option. This window has two panes: the left pane lists all the current breakpoints, the right pane gives details about the breakpoint that is currently highlighted. The SpeedMenu from the left pane has options to add or delete breakpoints, etc.

4.2 The Case of the Speckled Bug

By Mike Haney.

“You know, my dear Watson,” he said, pausing to produce another billow of aromatic smoke, “a computer program can be debugged using the same methods that are applied to solving a mystery.”

“You mean *observation* and *deduction*, don’t you Holmes?”

“Precisely, Watson. When a program stops unexpectedly or prints out unanticipated messages, the programmer can safely assume that the program has met with foul play. In other words, it has a bug.”

“But how does one debug a large program? So many things could go wrong. The task of righting all of them seems insurmountable.”

“It would be impossible indeed if you tried to solve all the problems at once. No, my friend, one should attack the matter bit by bit. Try to identify the little problems and solve them. Do not search for ‘the magic solution’ that solves everything. Debugging means careful work. We would do well to remember the French philosopher Voltaire:

> ‘Le programme ne le raccommode pas,’

which means, loosely, ‘PROGRAMS DO NOT DEBUG THEMSELVES.’”

“Did Voltaire really say that?”

“No, but he would have if he had thought of it. But nonetheless, when a program dies, the programmer has immediately before him or her the single greatest clue to the problem: *the characters on the screen*. Sometimes the program itself will print out messages of significance to alert the programmer to errors.”

“But that requires the programmer to *think ahead* and include some *diagnostic messages in the program*, doesn’t it Holmes?”

“Of course, Watson. But when no such messages are available, one can still learn quite a bit from the messages (or lack of them) from DOS.”

“Is that all there is to debugging?”

“Most certainly not. In some cases, the cause of the error can be found by inspection, such as a typographical error or the use of the wrong addressing mode. But more often, one must use a debugger to determine the circumstances that led to the error in order to understand the problem.”

“The circumstances, Holmes?”

“Yes, Watson. In particular, the *contents of the registers* tell a great deal about what the program has been doing. When subroutines or interrupts are used, one register is extremely helpful.”

“You mean the *stack pointer*.”

“Exactly. But it is not the contents of the stack pointer itself that is of so great importance, but rather the *contents of the stack* around the offset specified by the stack pointer. There is an old Hungarian saying:

> ‘Kerek egy kis uveg konyakot,’

which means ‘The stack contains a history of your program,’ or ‘Bring me a small flask of brandy;’ I can never remember which. By examining the contents of the stack it is possible to locate previous register contents (saved by `PUSH`) and return offsets. These return offsets form a path through the program that can be followed.”

“There may not be many return offsets on the stack. How is one to follow the ‘flow’ of the program up to the error?”

“The BREAKPOINT and TRACE operations hold the key to that problem. The judicious choice of breakpoints permits the programmer to isolate a region in which the error may be located. When this suspicious region is isolated, it is possible to single step through the code and carefully *observe the changes in the registers and memory variables*.”

“Can you be more explicit?” asked Watson, still perplexed.

“There is no method that works all the time, but there are several heuristics,” replied Holmes. “In the CONTRAST method, the programmer should compare what the *value should be* with what the *value is*. Also, the INPUT/OUTPUT method can help. Examine the variables upon starting a block of code (the input), and examine them again upon coming out of the block (the output). The programmer must then ask the following two questions:

1. Is the input correct? If not, a problem lies before this block.
2. Does the output correspond to the input? If not, a problem lies within this block.

If both are true, then the problem lies further on. Using the debugger’s memory window, the programmer can even force execution of a block of code with specific values in memory variables. There is no mechanical process that will automatically solve every problem. The programmer must *think*. Remember:

> ‘Mind is like parachute: works best when open.’”

“Confucius?”

“No, Charlie Chan. But the point is that the programmer must think about the program in order to debug it.”

“But after one has located the problem, it is so time consuming to re-edit and re-assemble and re-link the program. Surely there must be a better way to make small changes.”

“There is, Watson. With the debugger one can perform a limited amount of editing of the machine code by assembling new instructions in place. One can even replace unwanted code with `NOP`s (null operations). After these small changes, the programmer can continue to run the program in the debugger. Since these changes are not saved, however, the programmer must still edit the source code to make the changes permanent.”

“Won’t this take a lot of time?”

“You will have all the time you need, if you *start early*.”

Chapter 5
Assembly Language

This chapter describes some important aspects of assembly language.

5.1 Conditional Branching and Flags

5.1.1 The Processor Flags Register

The following diagram shows the location of the various flags in the processor status register.

Bit Number:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Flags:					OF	DF	IF	TF	SF	ZF		AF		PF		CF

`OF` — Overflow Flag
`DF` — Direction Flag
`IF` — Interrupt-Enable Flag
`TF` — Trap Flag
`SF` — Sign Flag
`ZF` — Zero Flag
`AF` — Auxiliary Carry Flag
`PF` — Parity Flag
`CF` — Carry Flag

5.1.2 Conditional Jumps

The following table lists the most common jump instructions and the tests they perform:

Instruction	**Jump Condition**	**Test**
`JE`	Jump if Equal	`ZF`=1
`JNE`	Jump if Not Equal	`ZF`=0
`JG`	Jump if Greater	(`ZF`=0) AND (`SF`=`OF`)
`JGE`	Jump if Greater or Equal	`SF`=`OF`
`JL`	Jump if Less	`SF`≠`OF`
`JLE`	Jump if Less or Equal	(`ZF`=1) OR (`SF`≠`OF`)

The following conditional branches are similar to the above but involve comparisons which treat the operands as unsigned integers:

Instruction	Jump Condition	Test
`JA`	Jump if Above	(`CF`=0) AND (`ZF`=0)
`JAE`	Jump if Above or Equal	`CF`=0
`JB`	Jump if Below	`CF`=1
`JBE`	Jump if Below or Equal	(`CF`=1) OR (`ZF`=1)

Finally, the branches below specifically test flags:

Instruction	Jump Condition	Test
`JO`	Jump on Overflow	`OF`=1
`JNO`	Jump on No Overflow	`OF`=0
`JC`	Jump on Carry	`CF`=1
`JNC`	Jump on No Carry	`CF`=0
`JS`	Jump on Sign (Negative)	`SF`=1
`JNS`	Jump on No Sign (Positive)	`SF`=0
`JZ`	Jump if Zero (same as JE)	`ZF`=1
`JNZ`	Jump if Not Zero	`ZF`=0

5.1.3 Meanings of the `OF`, `CF`, `SF`, and `ZF` Flags

The following table describes the meanings of the four flags used in conditional branching:

`OF` (Overflow)

1 — result is outside signed-number range
0 — otherwise

`CF` (Carry)

Carry out of (borrow into) high-order bit.
1 — result is outside unsigned-number range
0 — otherwise

`SF` (Sign)

High-order bit of result.
1 — negative signed number.
0 — positive signed number.

`ZF` (Zero)

1 — result = 0
0 — otherwise

5.2 Variations on Loops

A loop is a programming building block which allows you to repeat certain instructions until some predefined condition holds (or until a condition is no longer met, which is logically equivalent). Many loops simply repeat for a predefined number of iterations, but others are more complicated. Every processor architecture has instructions specifically designed to facilitate loop control. We treat here various methods for writing loops on the x86 processor family.

The writing of loop code is most easily shown by example; here we use a simple task of clearing a block of memory. The C version of this would be the following for loop:

```
for(i=0; i<100; i++)
    list[i] = 0;
```

Assume in the following that the memory block has been defined elsewhere with first byte address, `ListBegin`, and (last byte + 1) address, `ListEnd` (note this means that the last location to be cleared is the one before `ListEnd`), e.g.:

```
ListBegin       resb    100     ; reserve 100 bytes
ListEnd         equ     $       ; define as last-byte-address+1
```

In these examples, `BX` is used as a pointer into the memory block.

5.2.1 Standard Loops

Here is an example of the standard version of a loop, similar to the C version:

```
clrmem1:
        mov     bx, ListBegin   ; loop setup
.loop:
        mov     byte [bx], 0    ; loop action
        inc     bx              ; advance
        cmp     bx, ListEnd     ; termination test
        jb      .loop1          ; recycle in loop
```

This short and fast version illustrates the 4 elements of a loop: 1) setup; 2) loop action; 3) loop advance; and 4) termination test. As written, this version has the disadvantage that it always executes the loop action at least once. This comes about because the end test is performed after the loop action; hence there will be one loop action done even for an empty list (`ListBegin` = `ListEnd`).

A safer version is:

```
clrmem2:
        mov     bx, ListBegin   ; loop setup
.loop:
        cmp     bx, ListEnd     ; termination test
        jae     .next
        mov     byte [bx], 0    ; loop action
        inc     bx              ; advance
        jmp     short .loop     ; recycle
.next:  ...
```

Here, at the cost of one more instruction, the loop will work properly when zero iterations are called for. To speed up the loop itself, one can use the structure of the first example, but enter into the loop differently, i.e.,

```
clrmem3:
        mov     bx, ListBegin   ; loop setup
        jmp     .lptest         ; check for termination first
.loop:
        mov     byte [bx], 0    ; loop action
        inc     bx              ; advance
.lptest:
        cmp     bx, ListEnd     ; termination test
        jb      .loop           ; recycle in loop
```

5.2.2 Indexed Loops

Use of indexed addressing creates a shorter loop sequence:

```
clrmem4:
        mov     bx, ListEnd-ListBegin-1 ; BX = # bytes - 1
.loop:
        mov     byte [ListBegin+bx], 0  ; loop action
        dec     bx                      ; advance (dec here)
        jg      .loop                   ; (arithmetic) termination test
```

Note that now the block is cleared in backwards order, i.e., so that `ListBegin` is cleared last. The arithmetic termination test works here so long as the memory block to be cleared is less than (2^{15}) bytes long—i.e., so long as (`ListEnd-ListBegin`) is positive.

5.2.3 The `LOOP` instruction

The `LOOP` *`label`* instruction is useful when the number of iterations can be determined before the execution of the loop begins. The `LOOP` instruction decrements `CX` by 1 and, if the result is not zero, jumps to *`label`*. This results in the following form for our example task:

```
clrmem5:
        mov     cx, ListEnd-ListBegin   ; CX = # bytes
        xor     bx, bx                  ; index counts up in BX (from 0)
.loop:
        mov     byte [ListBegin+bx], 0  ; loop action
        inc     bx                      ; advance index
        loop    .loop                   ; dec cx and jump if cx not 0
```

Note: On modern processors, the two instruction sequence

```
dec     cx
jnz     .loop
```

is faster than `loop .loop`

This loop could be even shorter if it were also possible to index through `CX` rather than `BX`, but alas this is not so in the 16-bit instruction set (in the 32-bit instruction set, it's possible to index using `ECX`). Note that with a loop offset advance of 1 only, the MOV instruction must be a byte move. There are also variations on the LOOP instruction available for testing zero results from the loop action in addition to counting in `CX`: see Section A.99 for further information on LOOPZ and LOOPNZ.

In addition to the examples shown, there are many other address stepping and testing forms, the usefulness of which depends on special operand situations. The string instructions (see Section 5.5) also provide specialized operations (move, compare, scan, load, and store) on memory blocks of words or bytes.

5.3 Modular Programming

5.3.1 Structured Design and Modular Programming

Programs written in assembly language are inherently more difficult to understand than those written in a high-level language like Pascal or C. A thought that can be expressed in one Pascal statement is spread over ten or so statements in assembly language, and a Pascal construct like IF-THEN-ELSE or WHILE-ENDWHILE has to be simulated using comparisons and jumps. Techniques such as dividing the program into logical modules, commenting, and the use of blank lines and indentations to visually tie together blocks of statements that logically belong together, tend to make a program written in any language more readable but are especially helpful in assembly-language programming.

Sets of equates, macros, structure definitions, blocks of assembly language statements, and complete sub-procedures pertaining to a particular task or device can often become tools useful in other programs. The use of such tools, provided they are tested, validated, and properly documented, tends to make program development easier and faster.

There are two ways to keep such tools in separate files and combine them with the current program as needed: EQUates, MACROs, and STRUCture definitions are kept in an .ASM file and copied into another .ASM program with the `%include` directive; complete subprocedures are best assembled separately and kept in an .OBJ file from which the linker (TLINK) can extract the required subprocedures and combine them with the program into one .EXE file—that is in fact a linker's main *raison d'etre*.

5.3.2 The `%include` directive

An .ASM (or other) file consisting of EQUates, MACROs, and/or STRUCture definitions may be inserted into another assembly language program with the directive

```
%include [path]filename.ext
```

The entire file specified is inserted immediately after the `%include` directive and is assembled together with the rest of the program. Note that the file to be `%included` may consist of any collection of statements acceptable to NASM. If the file contains macro definitions it must of course be `%included` before any of the macros are invoked.

`%include` is particularly well suited for a library of EQUates, MACRO, and/or STRUCture definitions, since they add assembly code to the program only when they are invoked; for a file of subroutines, on the other hand, `%include` lengthens the program by adding to it the assembly code of all subroutines in the file, even those never invoked by the program, and slows down the assembly. A library of subroutines should instead be supplied as a file of separately assembled subroutines (although an include file may be supplied to define the routines as `EXTERN`).

5.3.3 Argument Transmission

The division of a programming task into several logical modules, or subprocedures, where each module's task and interface is defined carefully before detailed programming is started, makes it possible to identify subtasks for which existing subprocedures can be used, leads to faster program development, and makes it easier to test, validate, and verify the program.

The interface specification refers primarily to the calling convention or protocol, i.e., the method that is used to pass arguments to a subprocedure and pass results back to the calling program; restricting all interactions between modules to the well-defined interfaces makes it possible to use separately-developed modules from a tool kit, avoids "side effects," and minimizes incompatibility problems. In high-level languages the parameter passing conventions are pre-defined; in assembly language programs several different methods may be used but some preferred protocols are usually established to simplify the problem.

Arguments may be passed to a subprocedure, and results may be passed back to the calling program, in several different ways. If the subprocedure is recursive, the only practical way to pass arguments if via the stack; if the subprocedure is not recursive arguments may be passed via registers, via the stack, or via global variables (or a combination of these approaches). An argument may be passed "by value," i.e., as a signed or unsigned integer, ASCII character, or other code, or "by reference," i.e., as an address pointing to a variable, list, table, array, or structure. Commonly, argument passing by value is unidirectional, i.e., the values may be used and modified by the subprocedure with the changes not visible to the calling program. Argument passing by reference, however, is bidirectional, since the *contents* of the addresses given as arguments may be modified by the subprocedure, thus implicitly passing results back to the calling program.

An arbitrarily large number of arguments may be pushed onto the stack before the subprocedure is called; these arguments must be removed from the stack again upon return from the subprocedure. (Passing arguments via the stack is the only practical method for recursive subprocedures).

Registers are commonly used when few arguments are passed. (Note that one address argument is sufficient to point to an entire array, parameter list, etc).

Passing arguments via global variables is the least-general method, since specific global variables must be associated with specific subprocedures—the same effect could be achieved by passing the variables by reference.

Results computed by the subprocedure may be passed back to the calling program in a similar way, but note that results for arguments passed by reference are passed back implicitly.

5.3.4 Rules for NASM Procedures that Call External Procedures

It is frequently convenient to use procedures that have been assembled separately, such as library routines, and let the linker combine the separately assembled program with the library routines. Procedures in separately assembled modules are considered “external” to each other. The effective address of a name that is defined in an external procedure cannot be computed by NASM; it will have to be filled in later by the linker. However, such names have to be identified to NASM with the directive

```
EXTERN  name[, ...]
```

where `name` is the symbol defined in the external module.

Similarly, NASM must be able to tell the linker all names in a module that can be referenced by external procedures. This is done with the directive

```
GLOBAL  name[, ...]
```

Note that the names appearing in the EXTERN directive of this module are listed in the PUBLIC directive of the external module, and vice versa.

Lastly, the code segment in this module should have the same name as the code segment in the external procedure (which cannot be changed in the case of library procedures) and have the “combine-type” PUBLIC so that the two logical code segments are combined into one physical segment.

5.3.5 Creating a Separately Assembled Module

The module containing the main program is, strictly speaking, a separately assembled module. It differs from other separately assembled modules, however, in the following points:

- It is most likely the module in which the stack is declared.
- It is most likely the module containing the starting point of the program (`..start`).

Other separately assembled modules, whether parts of the program or library routines, are written following the rules given above but:

- Need not have a stack specified—if a stack is specified it will be concatenated with the stack specified in preceding modules
- Should *not* define a `..start` label.

5.3.6 Linking Separately Assembled Modules

TLINK is designed to link separately assembled procedures as well as extract the library routines used by the program from a library file. The general form for invoking TLINK is:

```
tlink [/c] [/v] {objfile ...} , [exefile] , [mapfile] , [libfile ...]
```

Where *`objfile`* (there may be more than one) are the separately assembled modules of the program and *`libfile`*s are the files to be searched for library routines. See Section 2.2 for a more detailed description of the TLINK options.

5.4 Programming Style

Your style of writing assembly language programs is almost as important as your accuracy. Good habits in layout, selection of symbolic names, and appropriate and illuminating comments help you to program correctly and easily. Good programming habits also make your programs much easier for your TA to read and grade. Here are two examples of the same NASM program to illustrate good and poor programming styles:

```
; MP99
;  Sharon Sharp
;  02-23-2001
;
;   This program copies the word contents of list1 to list2

;     Enter with:
;       SI = offset of List1
;       DI = offset of List2
;
;     Assumes:
;       Both lists are in DS
;       List end marker = 0
;
SEGMENT code

        GLOBAL LstCpy

LstCpy
        push    ax              ; AX used for temp
.lp:
        mov     ax, [si]        ; fetch word and test for zero
        test    ax, ax
        jz      .exit
        mov     [di], ax
        add     si, 2           ; advance and recycle
        add     di, 2
        jmp     .lp
.exit:
        pop     ax
        ret
```

Here the symbols help to define the program structure and the variables. The comments are inserted only where relevant and give functional rather than trivial instruction explanations. The open-spacing and orderly structure of the program also suggests the various parts. In addition, the listing to be handed in has a useful title.

The following program is a different kettle of fish:

```
segment       code
cc
 PUSH ax
    global cc
A:  mov ax,[si]
        CMP      AX,0
```

```
   je   b       ;
 mov    [di],ax;          move ax to list
  add si,2;     incr si
        ADD     di,2
 jmp a
b:              pop ax
 ret
```

This is a fairly extreme example of muddling up a program. The variables are hard to identify; the comments are thoroughly mixed with the instructions and only explain instructions not their function. Amazingly enough, NASM will still assemble this example to a working program, but it is almost incomprehensible to humans, including your teaching assistants.

Here is a complete program written to illustrate good programming style:

```
; Example Program
;  M. C. Loui
;  29 June 1986
;
; Translated to NASM
;  Peter L. B. Johnson
;  2001
;
; This boring program illustrates good programming style. The user
; types a list of names separated by carriage returns; if the user
; presses ESC before the return, then that name is canceled. The
; program uses the extremely inefficient bubble sort algorithm to
; sort the names into lexicographic order and prints them out.

        CR      equ     0Dh
        LF      equ     0Ah
        BEL     equ     07h
        ESCKEY  equ     1Bh
        MAXNUM  equ     20      ; Maximum number of names
        MAXLEN  equ     80      ; Maximum name length

        EXTERN  kbdine, dspout, dspmsg, dosxit

SEGMENT stkseg STACK            ; **** STACK SEGMENT ****
        resb    64*8

SEGMENT code                    ; **** CODE SEGMENT ****

Array   times MAXNUM*MAXLEN db ’$’      ; Array of names
NumNam  dw      0               ; Number of names input
HdMsg   db      ’Type up to 20 names separated by carriage returns’,’$’

..start:
        mov     ax, cs
        mov     ds, ax          ; Initialize DS register

        mov     dx, HdMsg
        call    dspmsg
        call    InNames         ; Input the names
        call    Sort            ; Sort them
        call    OutNames        ; Output the names
        call    dosxit

;
; Subroutine InNames
; Reads in list of up to 20 names from the user and stores them in Array.
;      Output: NumNam - Number of names typed in
;      Calls:  GetOne
;
InNames
        push    bx
        mov     word [NumNam], 0        ; Initialize NumNam
        mov     bx, Array               ; BX points to next name
.lp:
        call    GetOne                  ; Input one name
        cmp     byte [GetStat], 0       ; Done if user typed only
        je      .done                   ;  a carriage return
        add     bx, MAXLEN              ; Point to next name
        inc     word [NumNam]
        cmp     word [NumNam], MAXNUM   ; Continue only if
        jl      .lp                     ;  NumNam < MAXNUM
.done:
        pop     bx
        ret

;
; Subroutine GetOne
; Reads in one name of up to MAXLEN characters into Array.
;      Input:  BX - Offset of name in Array
;      Output: GETSTAT - Status of call
;                      = 1 if normal return
;                      = 0 if user typed only carriage return
;      Calls:  dspmsg, kbdine, dspout
;
GetStat resb    1
Prompt  db      CR,LF,’:$’
GetOne
        push    si
        push    dx
        push    ax
.lp1:
        xor     si, si
        mov     dx, Prompt              ; Prompt user for name
        call    dspmsg
.lp2:
        call    kbdine                  ; Get next character into AL
        cmp     al, CR
        je      .cr
        cmp     al, ESCKEY
        je      .esc
        mov     byte [bx+si], al        ; Store character
        inc     si
        jmp     short .lp2
.esc:
        mov     dl, BEL                 ; If user typed ESC
        call    dspout                  ;  then ring bell
        jmp     short .lp1              ;  and restart this name
.cr:
        mov     byte [GetStat], 1       ; If user typed only CR
        test    si, si
        jnz     .exit
        mov     byte [GetStat], 0       ;  then set GetStat to 0
.exit:
        pop     ax
        pop     dx
        pop     si
        ret

;
; Subroutine Sort
; Sorts Array using the bubble sort algorithm.
; WARNING: This algorithm is extremely inefficient.
;       Input:  NumNam - Number of names in Array
;       Calls:  CmpNam, XChgNam
;
I       resw    1                       ; Index into Array
J       resw    1                       ; Index into Array
Len     resw    1                       ; Length of each name
Sort
        push    ax
        push    dx
        push    si
        push    di

        mov     word [Len], MAXLEN
        mov     ax, [NumNam]            ; Initialize I
        dec     ax                      ;  to NumNam - 1
        mov     [I], ax                 ; for I=N-1 down to 1 do
.loopi:
        cmp     word [I], 1
        jl      .exit
        mov     word [J], 1             ;  for J=1 to I do
.loopj:
        mov     ax, [J]
        cmp     ax, [I]                 ;  (cannot cmp [J],[I])
        jg      .deci
        dec     ax                      ;   compute offset of
        mul     word [Len]              ;    Jth name
        mov     si, ax                  ;   SI = offset of Jth name
        add     di, [Len]               ;   DI = offset of J+1st name
        call    CmpNam
        cmp     byte [Result], 0        ;   if Jth name > J+1st name
        jle     .incj
        call    XChgNam                 ;    then exchange them
```

```
.incj:
        inc     word [J]
        jmp     short .loopj            ;  end for
.deci:
        dec     word [I]
        jmp     short .loopi            ; end for
.exit:
        pop     di
        pop     si
        pop     dx
        pop     ax
        ret

;
; Subroutine CmpNam
; Compares two names and determines which is lexicographically larger.
;       Inputs:  SI, DI - pointers to two names
;       Outputs: Result = -1 if name at SI is before name at DI
;                       = 0  if name at SI equals name at DI
;                       = 1  if name at SI is after name at DI
;
Result  resb    1
CmpNam
        push    ax
        push    cx
        push    si
        push    di

        mov     cx, MAXLEN              ; Length of names
.lp:
        mov     al, [si]                ; Compare next bytes
        cmp     al, [di]
        ja      .a
        jb      .b
        inc     si
        inc     di
        loop    .lp
        mov     byte [Result], 0        ; If all bytes equal, then names
        jmp     short .exit             ;  equal.
.a:
        mov     byte [Result], 1        ; Here if name at SI before name at
        jmp     short .exit             ;  DI.
.b:
        mov     byte [Result], -1       ; Here if name at SI after name at
.exit:                                  ;  DI.
        pop     di
        pop     si
        pop     cx
        pop     ax
        ret

;
; Subroutine XChgNam
; Exchanges the names pointed to by SI and DI.
;       Inputs:  SI, DI - pointers to two names
;
XChgNam
        push    ax
        push    cx
        push    si
        push    di

        mov     cx, MAXLEN
.lp:
        mov     al, [si]                ; Exchange bytes pointed to
        xchg    al, [di]                ;  by SI and DI
        xchg    al, [si]
        inc     si
        inc     di
        loop    .lp

        pop     di
        pop     si
        pop     cx
        pop     ax
        ret

;
; Subroutine OutNames
; Prints out names from Array
;       Inputs:  NumNam - Number of names in Array
;       Calls:   dspmsg
;
CRLF    db      CR, LF, '$'
OutNames
        push    bx
        push    cx
        push    dx

        mov     cx, [NumNam]            ; Number of names to output
        mov     bx, Array               ; BX holds offset of next name
.lp:
        mov     dx, bx
        call    dspmsg
        mov     dx, CRLF                ; CR and LF to separate
        call    dspmsg                  ;  adjacant names
        add     bx, MAXLEN
        loop    .lp

        pop     dx
        pop     cx
        pop     bx
        ret
```

5.5 String Instructions

String operations are one of the best ways to apply something to a range of memory locations. Typical string operations are copying from one range to another and filling a range of memory locations with a specified value. Some string operations are:

`rep movsd` — copy one DWORD from one string to another
`rep stosd` — set the DWORD to the value in `EAX`

The last letter "D" of `MOVSD` or `STOSD` signifies DWORD. WORDs and BYTEs can be specified by using "W" or "B," respectively.

To use a string operation, 5 steps should be taken:

1. Set the source segment and offset
2. Set the destination segment and offset
3. Specify the direction (usually forward) of processing
4. Specify the number of units (DWORDs, WORDs, BYTEs) to apply the operation to
5. Specify the operation

Use `DS` to point to the source segment:

```
        mov     ax, ScratchSeg   ; from a defined segment
        mov     ds, ax
```

Use `ES` to point to the destination segment:

```
        mov     ax, 0A000h       ; graphics segment
        mov     es, ax
```

Specify the direction:

```
        cld             ; set direction flag forward
```

Specify the source offset:

```
        mov     esi, ScratchPad  ; set the source offset
```

Specify the destination offset:

```
        xor     edi, edi         ; set to 0
```

Specify the number of times to repeat the operation:

```
        mov     ecx, 16000
```

Specify the operation:

```
        rep movsd
```

Chapter 6 NASM

Originally written by Julian Hall and Simon Tantham.

6.1 Layout of a NASM Source Line

Like most assemblers, each NASM source line contains (unless it is a macro, a preprocessor directive or an assembler directive: see Section 6.10) some combination of the four fields

```
label:  instruction operands        ; comment
```

As usual, most of these fields are optional; the presence or absence of any combination of a label, an instruction and a comment is allowed. Of course, the operand field is either required or forbidden by the presence and nature of the instruction field.

NASM places no restrictions on white space within a line: labels may have white space before them, or instructions may have no space before them, or anything. The colon after a label is also optional. Note that this means that if you intend to code `lodsb` alone on a line, and type `lodab` by accident, then that's still a valid source line which does nothing but define a label. Running NASM with the command-line option `-w+orphan-labels` will cause it to warn you if you define a label alone on a line without a trailing colon.

Valid characters in labels are letters, numbers, `_`, `$`, `#`, `@`, `~`, `.`, and `?`. The only characters which may be used as the *first* character of an identifier are letters, `.` (with special meaning: see Section 6.8), `_` and `?`. An identifier may also be prefixed with a `$` to indicate that it is intended to be read as an identifier and not a reserved word; thus, if some other module you are linking with defines a symbol called `EAX`, you can refer to `$eax` in NASM code to distinguish the symbol from the register.

The instruction field may contain any machine instruction: Pentium and P6 instructions, FPU instructions, MMX instructions and even undocumented instructions are all supported. The instruction may be prefixed by `LOCK`, `REP`, `REPE`/`REPZ` or `REPNE`/`REPNZ`, in the usual way. Explicit address-size and operand-size prefixes `A16`, `A32`, `O16` and `O32` are provided—. You can also use the name of a segment register as an instruction prefix: coding `es mov [bx],ax` is equivalent to coding `mov [es:bx],ax`. We recommend the latter syntax, since it is consistent with other syntactic features of the language, but for instructions such as `LODSB`, which has no operands and yet can require a segment override, there is no clean syntactic way to proceed apart from `es lodsb`.

An instruction is not required to use a prefix: prefixes such as `CS`, `A32`, `LOCK` or `REPE` can appear on a line by themselves, and NASM will just generate the prefix bytes.

In addition to actual machine instructions, NASM also supports a number of pseudo-instructions, described in Section 6.2.

Instruction operands may take a number of forms: they can be registers, described simply by the register name (e.g. `AX`, `BP`, `EBX`, `CR0`: NASM does not use the `gas`-style syntax in which register names must be prefixed by a `%` sign), or they can be effective addresses (see Section 6.3), constants (Section 6.4) or expressions (Section 6.5).

For floating-point instructions, NASM accepts a wide range of syntaxes: you can use two-operand forms like MASM supports, or you can use NASM's native single-operand forms in most cases. Details of all forms of each supported instruction are given in Appendix A. For example, you can code:

```
fadd    st1             ; this sets st0 := st0 + st1
fadd    st0, st1        ; so does this

fadd    st1, st0        ; this sets st1 := st1 + st0
fadd    to st1          ; so does this
```

Almost any floating-point instruction that references memory must use one of the prefixes `DWORD`, `QWORD` or `TWORD` to indicate what size of memory operand it refers to.

6.2 Pseudo-Instructions

Pseudo-instructions are things which, though not real x86 machine instructions, are used in the instruction field anyway because that's the most convenient place to put them. The current pseudo-instructions are `DB`, `DW`, `DD`, `DQ` and `DT`, their uninitialized counterparts `RESB`, `RESW`, `RESD`, `RESQ` and `REST`, the `INCBIN` command, the `EQU` command, and the `TIMES` prefix.

6.2.1 `DB` and Friends: Declaring Initialized Data

`DB`, `DW`, `DD`, `DQ` and `DT` are used to declare initialized data in the output file. They can be invoked in a wide range of ways:

```
db      0x55                ; just the byte 0x55
db      0x55,0x56,0x57      ; three bytes in succession
db      ’a’,0x55            ; character constants are OK
db      ’hello’,13,10,’$’   ; so are string constants
dw      0x1234              ; 0x34 0x12
dw      ’a’                 ; 0x41 0x00 (it’s just a number)
dw      ’ab’                ; 0x41 0x42 (character constant)
dw      ’abc’               ; 0x41 0x42 0x43 0x00 (string)
dd      0x12345678          ; 0x78 0x56 0x34 0x12
dd      1.234567e20         ; floating-point constant
dq      1.234567e20         ; double-precision float
dt      1.234567e20         ; extended-precision float
```

`DQ` and `DT` do not accept numeric constants or string constants as operands.

6.2.2 `RESB` and friends: Declaring Uninitialized Data

`RESB`, `RESW`, `RESD`, `RESQ` and `REST` are designed to be used in the BSS section of a module: they declare *uninitialised* storage space. Each takes a single operand, which is the number of bytes, words, doublewords or whatever to reserve. NASM does not support the MASM/TASM syntax of reserving uninitialised space by writing `DW ?` or similar things: this is what it does instead. The operand to a `RESB`-type pseudo-instruction is a *critical expression*: see Section 6.7.

For example:

```
buffer:         resb    64      ; reserve 64 bytes
wordvar:        resw    1       ; reserve a word
realarray       resq    10      ; array of ten reals
```

6.2.3 `INCBIN`: Including External Binary Files

`INCBIN` includes a binary file verbatim into the output file. This can be handy for (for example) including graphics and sound data directly into a game executable file. However, it is recommended to use this for only *small* pieces of data. It can be called in one of these three ways:

```
        incbin "file.dat"         ; include the whole file
        incbin "file.dat",1024    ; skip the first 1024 bytes
        incbin "file.dat",1024,512 ; skip the first 1024, and
                                   ; actually include at most 512
```

6.2.4 `EQU`: Defining Constants

`EQU` defines a symbol to a given constant value: when `EQU` is used, the source line must contain a label. The action of `EQU` is to define the given label name to the value of its (only) operand. This definition is absolute, and cannot change later. So, for example,

```
message db ’hello, world’
msglen  equ $-message
```

defines `msglen` to be the constant 12. `msglen` may not then be redefined later. This is not a preprocessor definition either: the value of `msglen` is evaluated *once*, using the value of `$` (see Section 6.5 for an explanation of `$`) at the point of definition, rather than being evaluated wherever it is referenced and using the value of `$` at the point of reference. Note that the operand to an `EQU` is also a critical expression (Section 6.7).

6.2.5 `TIMES`: Repeating Instructions or Data

The `TIMES` prefix causes the instruction to be assembled multiple times. This is partly present as NASM’s equivalent of the `DUP` syntax supported by MASM-compatible assemblers, in that you can code

```
zerobuf:        times 64 db 0
```

or similar things; but TIMES is more versatile than that. The argument to TIMES is not just a numeric constant, but a numeric *expression*, so you can do things like

```
buffer: db 'hello, world'
        times 64-$+buffer db ' '
```

which will store exactly enough spaces to make the total length of buffer up to 64. Finally, TIMES can be applied to ordinary instructions, so you can code trivial unrolled loops in it:

```
        times 100 movsb
```

Note that there is no effective difference between `times 100 resb 1` and `resb 100`, except that the latter will be assembled about 100 times faster due to the internal structure of the assembler.

The operand to TIMES, like that of EQU and those of RESB and friends, is a critical expression (Section 6.7).

Note also that TIMES can't be applied to macros: the reason for this is that TIMES is processed after the macro phase, which allows the argument to TIMES to contain expressions such as `64-$+buffer` as above. To repeat more than one line of code, or a complex macro, use the preprocessor `%rep` directive.

6.3 Effective Addresses

An effective address is any operand to an instruction which references memory. Effective addresses, in NASM, have a very simple syntax: they consist of an expression evaluating to the desired address, enclosed in square brackets. For example:

```
wordvar dw 123
        mov ax,[wordvar]
        mov ax,[wordvar+1]
        mov ax,[es:wordvar+bx]
```

Anything not conforming to this simple system is not a valid memory reference in NASM, for example `es:wordvar[BX]`.

More complicated effective addresses, such as those involving more than one register, work in exactly the same way:

```
        mov eax,[ebx*2+ecx+offset]
        mov ax,[bp+di+8]
```

NASM is capable of doing algebra on these effective addresses, so that things which don't necessarily *look* legal are perfectly all right:

```
        mov eax,[ebx*5]         ; assembles as [ebx*4+ebx]
        mov eax,[label1*2-label2] ; ie [label1+(label1-label2)]
```

Some forms of effective address have more than one assembled form; in most such cases NASM will generate the smallest form it can. For example, there are distinct assembled forms for the 32-bit effective addresses `[EAX*2+0]` and `[EAX+EAX]`, and NASM will generally generate the latter on the grounds that the former requires four bytes to store a zero offset.

NASM has a hinting mechanism which will cause `[EAX+EBX]` and `[EBX+EAX]` to generate different opcodes; this is occasionally useful because `[ESI+EBP]` and `[EBP+ESI]` have different default segment registers.

However, you can force NASM to generate an effective address in a particular form by the use of the keywords `BYTE`, `WORD`, `DWORD` and `NOSPLIT`. If you need `[EAX+3]` to be assembled using a double-word offset field instead of the one byte NASM will normally generate, you can code `[dword EAX+3]`. Similarly, you can force NASM to use a byte offset for a small value which it hasn't seen on the first pass (see Section 6.7 for an example of such a code fragment) by using `[byte EAX+offset]`. As special cases, `[byte EAX]` will code `[EAX+0]` with a byte offset of zero, and `[dword EAX]` will code it with a double-word offset of zero. The normal form, `[EAX]`, will be coded with no offset field.

Similarly, NASM will split `[EAX*2]` into `[EAX+EAX]` because that allows the offset field to be absent and space to be saved; in fact, it will also split `[EAX*2+offset]` into `[EAX+EAX+offset]`. You can combat this behaviour by the use of the `NOSPLIT` keyword: `[nosplit EAX*2]` will force `[EAX*2+0]` to be generated literally.

6.4 Constants

NASM understands four different types of constant: numeric, character, string and floating-point.

6.4.1 Numeric Constants

A numeric constant is simply a number. NASM allows you to specify numbers in a variety of number bases, in a variety of ways: you can suffix `H`, `Q` and `B` for hex, octal and binary, or you can prefix `0x` for hex in the style of C, or you can prefix `$` for hex in the style of Borland Pascal. Note, though, that the `$` prefix does double duty as a prefix on identifiers (see Section 6.1), so a hex number prefixed with a `$` sign must have a digit after the `$` rather than a letter.

Some examples:

```
mov ax,100              ; decimal
mov ax,0a2h             ; hex
mov ax,$0a2             ; hex again: the 0 is required
mov ax,0xa2             ; hex yet again
mov ax,777q             ; octal
mov ax,10010011b        ; binary
```

6.4.2 Character Constants

A character constant consists of up to four characters enclosed in either single or double quotes. The type of quote makes no difference to NASM, except of course that surrounding the constant with single quotes allows double quotes to appear within it and vice versa.

A character constant with more than one character will be arranged with little-endian order in mind: if you code

```
mov eax,'abcd'
```

then the constant generated is not `0x61626364`, but `0x64636261`, so that if you were then to store the value into memory, it would read `abcd` rather than `dcba`. This is also the sense of character constants understood by the Pentium's `CPUID` instruction (see Section A.22).

6.4.3 String Constants

String constants are only acceptable to some pseudo-instructions, namely the `DB` family and `INCBIN`.

A string constant looks like a character constant, only longer. It is treated as a concatenation of maximum-size character constants for the conditions. So the following are equivalent:

```
db 'hello'              ; string constant
db 'h','e','l','l','o'  ; equivalent character constants
```

And the following are also equivalent:

```
dd 'ninechars'          ; doubleword string constant
dd 'nine','char','s'    ; becomes three doublewords
db 'ninechars',0,0,0    ; and really looks like this
```

Note that when used as an operand to `db`, a constant like `'ab'` is treated as a string constant despite being short enough to be a character constant, because otherwise `db 'ab'` would have the same effect as `db 'a'`, which would be silly. Similarly, three-character or four-character constants are treated as strings when they are operands to `dw`.

6.4.4 Floating-Point Constants

Floating-point constants are acceptable only as arguments to `DD`, `DQ` and `DT`. They are expressed in the traditional form: digits, then a period, then optionally more digits, then optionally an `E` followed by an exponent. The period is mandatory, so that NASM can distinguish between `dd 1`, which declares an integer constant, and `dd 1.0` which declares a floating-point constant.

Some examples:

```
dd 1.2                  ; an easy one
dq 1.e10                ; 10,000,000,000
dq 1.e+10               ; synonymous with 1.e10
dq 1.e-10               ; 0.000 000 000 1
dt 3.141592653589793238462 ; pi
```

NASM cannot do compile-time arithmetic on floating-point constants. This is because NASM is designed to be portable - although it always generates code to run on x86 processors, the assembler itself can run on any system with an ANSI C compiler. Therefore, the assembler cannot guarantee the presence of a floating-point unit capable of handling the Intel number formats, and so for NASM to be able to do floating arithmetic it would have to include its own complete set of floating-point routines, which would significantly increase the size of the assembler for very little benefit.

6.5 Expressions

Expressions in NASM are similar in syntax to those in C.

NASM does not guarantee the size of the integers used to evaluate expressions at compile time: since NASM can compile and run on 64-bit systems quite happily, don't assume that expressions are evaluated in 32-bit registers and so try to make deliberate use of integer overflow. It might not always work. The only thing NASM will guarantee is what's guaranteed by ANSI C: you always have *at least* 32 bits to work in.

NASM supports two special tokens in expressions, allowing calculations to involve the current assembly position: the `$` and `$$` tokens. `$` evaluates to the assembly position at the beginning of the line containing the expression; so you can code an infinite loop using `JMP $`. `$$` evaluates to the beginning of the current section; so you can tell how far into the section you are by using `($-$$)`.

The arithmetic operators provided by NASM are listed here, in increasing order of precedence.

6.5.1 `|`: Bitwise OR Operator

The `|` operator gives a bitwise OR, exactly as performed by the `OR` machine instruction. Bitwise OR is the lowest-priority arithmetic operator supported by NASM.

6.5.2 `^`: Bitwise XOR Operator

`^` provides the bitwise XOR operation.

6.5.3 `&`: Bitwise AND Operator

`&` provides the bitwise AND operation.

6.5.4 << and >>: Bit Shift Operators

<< gives a bit-shift to the left, just as it does in C. So `5<<3` evaluates to 5 times 8, or 40. >> gives a bit-shift to the right; in NASM, such a shift is *always* unsigned, so that the bits shifted in from the left-hand end are filled with zero rather than a sign-extension of the previous highest bit.

6.5.5 `+` and `-`: Addition and Subtraction Operators

The `+` and `-` operators do perfectly ordinary addition and subtraction.

6.5.6 `*`, `/`, `//`, `%` and `%%`: Multiplication and Division

`*` is the multiplication operator. `/` and `//` are both division operators: `/` is unsigned division and `//` is signed division. Similarly, `%` and `%%` provide unsigned and signed modulo operators respectively.

NASM, like ANSI C, provides no guarantees about the sensible operation of the signed modulo operator.

Since the `%` character is used extensively by the macro preprocessor, you should ensure that both the signed and unsigned modulo operators are followed by white space wherever they appear.

6.5.7 Unary Operators: `+`, `-`, `~` and `SEG`

The highest-priority operators in NASM's expression grammar are those which only apply to one argument. `-` negates its operand, `+` does nothing (it's provided for symmetry with `-`), `~` computes the one's complement of its operand, and `SEG` provides the segment address of its operand (explained in more detail in Section 6.6).

6.6 `SEG` and `WRT`

When writing large 16-bit programs, which must be split into multiple segments, it is often necessary to be able to refer to the segment part of the address of a symbol. NASM supports the `SEG` operator to perform this function.

The `SEG` operator returns the *preferred* segment base of a symbol, defined as the segment base relative to which the offset of the symbol makes sense. So the code

```
mov ax, seg symbol
mov es, ax
mov bx, symbol
```

will load `ES:BX` with a valid pointer to the symbol `symbol`.

Things can be more complex than this: since 16-bit segments and groups may overlap, you might occasionally want to refer to some symbol using a different segment base from the preferred one. NASM lets you do this, by the use of the `WRT` (With Reference To) keyword. So you can do things like

```
mov ax, weird_seg       ; weird_seg is a segment base
mov es, ax
mov bx, symbol wrt weird_seg
```

to load `ES:BX` with a different, but functionally equivalent, pointer to the symbol `symbol`.

NASM supports far (inter-segment) calls and jumps by means of the syntax `call segment:offset`, where `segment` and `offset` both represent immediate values. So to call a far procedure, you could code either of

```
call (seg procedure):procedure
call weird_seg:(procedure wrt weird_seg)
```

(The parentheses are included for clarity, to show the intended parsing of the above instructions. They are not necessary in practice.)

NASM supports the syntax `call far procedure` as a synonym for the first of the above usages. `JMP` works identically to `CALL` in these examples.

To declare a far pointer to a data item in a data segment, you must code

```
        dw symbol, seg symbol
```

NASM supports no convenient synonym for this, though you can always invent one using the macro processor.

6.7 Critical Expressions

A limitation of NASM is that it is a two-pass assembler; unlike TASM and others, it will always do exactly two assembly passes. Therefore it is unable to cope with source files that are complex enough to require three or more passes.

The first pass is used to determine the size of all the assembled code and data, so that the second pass, when generating all the code, knows all the symbol addresses the code refers to. So one thing NASM can't handle is code whose size depends on the value of a symbol declared after the code in question. For example,

```
        times (label-$) db 0
label:  db 'Where am I?'
```

The argument to `TIMES` in this case could equally legally evaluate to anything at all; NASM will reject this example because it cannot tell the size of the `TIMES` line when it first sees it. It will just as firmly reject the slightly paradoxical code

```
        times (label-$+1) db 0
label:  db 'NOW where am I?'
```

in which *any* value for the `TIMES` argument is by definition wrong!

NASM rejects these examples by means of a concept called a *critical expression*, which is defined to be an expression whose value is required to be computable in the first pass, and which must therefore depend only on symbols defined before it. The argument to the `TIMES` prefix is a critical expression; for the same reason, the arguments to the `RESB` family of pseudo-instructions are also critical expressions.

Critical expressions can crop up in other contexts as well: consider the following code.

```
        mov ax, symbol1
symbol1 equ symbol2
symbol2:
```

On the first pass, NASM cannot determine the value of `symbol1`, because `symbol1` is defined to be equal to `symbol2` which NASM hasn't seen yet. On the second pass, therefore, when it encounters the line `mov ax,symbol1`, it is unable to generate the code for it because it still doesn't know the value of `symbol1`. On the next line, it would see the `EQU` again and be able to determine the value of `symbol1`, but by then it would be too late.

NASM avoids this problem by defining the right-hand side of an `EQU` statement to be a critical expression, so the definition of `symbol1` would be rejected in the first pass.

There is a related issue involving forward references: consider this code fragment.

```
        mov eax, [ebx+offset]
offset  equ 10
```

NASM, on pass one, must calculate the size of the instruction `mov eax,[ebx+offset]` without knowing the value of `offset`. It has no way of knowing that `offset` is small enough to fit into a one-byte offset field and that it could therefore get away with generating a shorter form of the effective-address encoding; for all it knows, in pass one, `offset` could be a symbol in the code segment, and it might need the full four-byte form. So it is forced to compute the size of the instruction to accommodate a four-byte address part. In pass two, having made this decision, it is now forced to honour it and keep the instruction large, so the code generated in this case is not as small as it could have been. This problem can be solved by defining `offset` before using it, or by forcing byte size in the effective address by coding `[byte ebx+offset]`.

6.8 Local Labels

NASM gives special treatment to symbols beginning with a period. A label beginning with a single period is treated as a *local* label, which means that it is associated with the previous non-local label. So, for example:

```
label1  ; some code
.loop   ; some more code
        jne .loop
        ret
label2  ; some code
.loop   ; some more code
        jne .loop
        ret
```

In the above code fragment, each `JNE` instruction jumps to the line immediately before it, because the two definitions of `.loop` are kept separate by virtue of each being associated with the previous non-local label.

NASM goes one step further, in allowing access to local labels from other parts of the code. This is achieved by means of *defining* a local label in terms of the previous non-local label: the first definition of `.loop` above is really defining a symbol called `label1.loop`, and the second defines a symbol called `label2.loop`. So, if you really needed to, you could write

```
label3  ; some more code
        ; and some more
        jmp label1.loop
```

Sometimes it is useful - in a macro, for instance - to be able to define a label which can be referenced from anywhere but which doesn't interfere with the normal local-label mechanism. Such a label can't be non-local because it would interfere with subsequent definitions of, and references to, local labels; and it can't be local because the macro that defined it wouldn't know the label's full name. NASM therefore introduces a third type of label, which is probably only useful in macro definitions: if a label begins with the special prefix `..@`, then it does nothing to the local label mechanism. So you could code

```
label1: ; a non-local label
.local: ; this is really label1.local
..@foo: ; this is a special symbol
label2: ; another non-local label
.local: ; this is really label2.local
        jmp ..@foo              ; this will jump three lines up
```

NASM has the capacity to define other special symbols beginning with a double period: for example, `..start` is used to specify the entry point in the `obj` output format.

6.9 Standard Macros

NASM defines a set of standard macros, which are already defined when it starts to process any source file. If you really need a program to be assembled with no pre-defined macros, you can use the `%clear` directive to empty the preprocessor of everything.

Most user-level assembler directives (see Section 6.10) are implemented as macros which invoke primitive directives; these are described in Section 6.10. The rest of the standard macro set is described here.

6.9.1 `__NASM_MAJOR__` and `__NASM_MINOR__`: NASM Version

The single-line macros `__NASM_MAJOR__` and `__NASM_MINOR__` expand to the major and minor parts of the version number of NASM being used. So, under NASM 0.96 for example, `__NASM_MAJOR__` would be defined to be 0 and `__NASM_MINOR__` would be defined as 96.

6.9.2 `__FILE__` and `__LINE__`: File Name and Line Number

Like the C preprocessor, NASM allows the user to find out the file name and line number containing the current instruction. The macro `__FILE__` expands to a string constant giving the name of the current input file (which may change through the course of assembly if `%include` directives are used), and `__LINE__` expands to a numeric constant giving the current line number in the input file.

These macros could be used, for example, to communicate debugging information to a macro, since invoking `__LINE__` inside a macro definition (either single-line or multi-line) will return the line number of the macro *call*, rather than *definition*. So to determine where in a piece of code a crash is occurring, for example, one could write a routine `stillhere`, which is passed a line number in `EAX` and outputs something like “line 155: still here”. You could then write a macro

```
%macro notdeadyet 0
        push    eax
        mov     eax, __LINE__
        call    stillhere
        pop     eax
%endmacro
```

and then pepper your code with calls to `notdeadyet` until you find the crash point.

6.9.3 `STRUC` and `ENDSTRUC`: Declaring Structure Data Types

The core of NASM contains no intrinsic means of defining data structures; instead, the preprocessor is sufficiently powerful that data structures can be implemented as a set of macros. The macros `STRUC` and `ENDSTRUC` are used to define a structure data type.

`STRUC` takes one parameter, which is the name of the data type. This name is defined as a symbol with the value zero, and also has the suffix `_size` appended to it and is then defined as an `EQU` giving the size of the structure. Once `STRUC` has been issued, you are defining the structure, and should define fields using the `RESB` family of pseudo-instructions, and then invoke `ENDSTRUC` to finish the definition.

For example, to define a structure called `mytype` containing a longword, a word, a byte and a string of bytes, you might code

```
        struc   mytype
mt_long:        resd 1
mt_word:        resw 1
mt_byte:        resb 1
mt_str:         resb 32
        endstruc
```

The above code defines six symbols: `mt_long` as 0 (the offset from the beginning of a `mytype` structure to the longword field), `mt_word` as 4, `mt_byte` as 6, `mt_str` as 7, `mytype_size` as 39, and `mytype` itself as zero.

The reason why the structure type name is defined at zero is a side effect of allowing structures to work with the local label mechanism: if your structure members tend to have the same names in more than one structure, you can define the above structure like this:

```
        struc   mytype
.long:  resd 1
.word:  resw 1
.byte:  resb 1
.str:   resb 32
        endstruc
```

This defines the offsets to the structure fields as `mytype.long`, `mytype.word`, `mytype.byte` and `mytype.str`.

NASM, since it has no *intrinsic* structure support, does not support any form of period notation to refer to the elements of a structure once you have one (except the above local-label notation), so code such as `mov ax,[mystruc.mt_word]` is not valid. `mt_word` is a constant just like any other constant, so the correct syntax is `mov ax,[mystruc+mt_word]` or `mov ax,[mystruc+mytype.word]`.

6.9.4 `ISTRUC`, `AT` and `IEND`: Declaring Instances of Structures

Having defined a structure type, the next thing you typically want to do is to declare instances of that structure in your data segment. NASM provides an easy way to do this in the `ISTRUC` mechanism. To declare a structure of type `mytype` in a program, you code something like this:

```
mystruc:        istruc  mytype
        at mt_long, dd 123456
        at mt_word, dw 1024
        at mt_byte, db ’x’
        at mt_str,  db ’hello, world’, 13, 10, 0
                iend
```

The function of the `AT` macro is to make use of the `TIMES` prefix to advance the assembly position to the correct point for the specified structure field, and then to declare the specified data. Therefore the structure fields must be declared in the same order as they were specified in the structure definition.

If the data to go in a structure field requires more than one source line to specify, the remaining source lines can easily come after the `AT` line. For example:

```
at mt_str, db 123,134,145,156,167,178,189
db 190,100,0
```

Depending on personal taste, you can also omit the code part of the `AT` line completely, and start the structure field on the next line:

```
at mt_str
db ’hello, world’
db 13,10,0
```

6.9.5 `ALIGN` and `ALIGNB`: Data Alignment

The `ALIGN` and `ALIGNB` macros provides a convenient way to align code or data on a word, longword, paragraph or other boundary. The syntax of the `ALIGN` and `ALIGNB` macros is

```
align 4                 ; align on 4-byte boundary
align 16                ; align on 16-byte boundary
align 8,db 0            ; pad with 0s rather than NOPs
align 4,resb 1          ; align to 4 in the BSS
alignb 4                ; equivalent to previous line
```

Both macros require their first argument to be a power of two; they both compute the number of additional bytes required to bring the length of the current section up to a multiple of that power of two, and then apply the `TIMES` prefix to their second argument to perform the alignment.

If the second argument is not specified, the default for `ALIGN` is `NOP`, and the default for `ALIGNB` is `RESB 1`. So if the second argument is specified, the two macros are equivalent. Normally, you can just use `ALIGN` in code and data sections and `ALIGNB` in BSS sections, and never need the second argument except for special purposes.

`ALIGN` and `ALIGNB`, being simple macros, perform no error checking: they cannot warn you if their first argument fails to be a power of two, or if their second argument generates more than one byte of code. In each of these cases they will silently do the wrong thing.

`ALIGNB` (or `ALIGN` with a second argument of `RESB 1`) can be used within structure definitions:

```
        struc   mytype2
mt_byte:        resb 1
                alignb 2
mt_word:        resw 1
                alignb 4
mt_long:        resd 1
mt_str:         resb 32
        endstruc
```

This will ensure that the structure members are sensibly aligned relative to the base of the structure.

A final caveat: `ALIGN` and `ALIGNB` work relative to the beginning of the *section*, not the beginning of the address space in the final executable. Aligning to a 16-byte boundary when the section you're in is only guaranteed to be aligned to a 4-byte boundary, for example, is a waste of effort. Again, NASM does not check that the section's alignment characteristics are sensible for the use of `ALIGN` or `ALIGNB`.

6.10 Assembler Directives

NASM's directives come in two types: *user-level* directives and *primitive* directives. Typically, each directive has a user-level form and a primitive form. In almost all cases, we recommend that users use the user-level forms of the directives, which are implemented as macros which call the primitive forms.

Primitive directives are enclosed in square brackets; user-level directives are not.

In addition to the universal directives described in this chapter, each object file format can optionally supply extra directives in order to control particular features of that file format. These *format-specific* directives are documented along with the formats that implement them, in the NASM Manual.

6.10.1 `BITS`: Specifying Target Processor Mode

The `BITS` directive specifies whether NASM should generate code designed to run on a processor operating in 16-bit mode, or code designed to run on a processor operating in 32-bit mode. The syntax is `BITS 16` or `BITS 32`.

In most cases, you should not need to use `BITS` explicitly. The `aout`, `coff`, `elf` and `win32` object formats, which are designed for use in 32-bit operating systems, all cause NASM to select 32-bit mode by default. The `obj` object format allows you to specify each segment you define as either `USE16` or `USE32`, and NASM will set its operating mode accordingly, so the use of the `BITS` directive is once again unnecessary.

The most likely reason for using the `BITS` directive is to write 32-bit code in a flat binary file; this is because the `bin` output format defaults to 16-bit mode in anticipation of it being used most frequently to write DOS `.COM` programs, DOS `.SYS` device drivers and boot loader software.

You do *not* need to specify `BITS 32` merely in order to use 32-bit instructions in a 16-bit DOS program; if you do, the assembler will generate incorrect code because it will be writing code targeted at a 32-bit platform, to be run on a 16-bit one.

When NASM is in `BITS 16` state, instructions which use 32-bit data are prefixed with an 0x66 byte, and those referring to 32-bit addresses have an 0x67 prefix. In `BITS 32` state, the reverse is true: 32-bit instructions require no prefixes, whereas instructions using 16-bit data need an 0x66 and those working in 16-bit addresses need an 0x67.

The `BITS` directive has an exactly equivalent primitive form, `[BITS 16]` and `[BITS 32]`. The user-level form is a macro which has no function other than to call the primitive form.

6.10.2 `SECTION` or `SEGMENT`: Changing and Defining Sections

The `SECTION` directive (`SEGMENT` is an exactly equivalent synonym) changes which section of the output file the code you write will be assembled into. In some object file formats, the number and names of sections are fixed; in others, the user may make up as many as they wish. Hence `SECTION` may sometimes give an error message, or may define a new section, if you try to switch to a section that does not (yet) exist.

The Unix object formats, and the `bin` object format, all support the standardised section names `.text`, `.data` and `.bss` for the code, data and uninitialised-data sections. The `obj` format, by contrast, does not recognise these section names as being special, and indeed will strip off the leading period of any section name that has one.

The `__SECT__` Macro

The `SECTION` directive is unusual in that its user-level form functions differently from its primitive form. The primitive form, `[SECTION xyz]`, simply switches the current target section to the one given. The user-level form, `SECTION xyz`, however, first defines the single-line macro `__SECT__` to be the primitive `[SECTION]` directive which it is about to issue, and then issues it. So the user-level directive

```
        SECTION .text
```

expands to the two lines

```
%define __SECT__ [SECTION .text]
        [SECTION .text]
```

Users may find it useful to make use of this in their own macros. For example, the `writefile` macro defined in the NASM Manual can be usefully rewritten in the following more sophisticated form:

```
%macro writefile 2+
        [section .data]
%%str:  db %2
%%endstr:
        __SECT__
        mov dx,%%str
        mov cx,%%endstr-%%str
        mov bx,%1
        mov ah,0x40
        int 0x21
%endmacro
```

This form of the macro, once passed a string to output, first switches temporarily to the data section of the file, using the primitive form of the `SECTION` directive so as not to modify `__SECT__`. It then declares its string in the data section, and then invokes `__SECT__` to switch back to *whichever* section the user was previously working in. It thus avoids the need, in the previous version of the macro, to include a `JMP` instruction to jump over the data, and also does not fail if, in a complicated `OBJ` format module, the user could potentially be assembling the code in any of several separate code sections.

6.10.3 `ABSOLUTE`: Defining Absolute Labels

The `ABSOLUTE` directive can be thought of as an alternative form of `SECTION`: it causes the subsequent code to be directed at no physical section, but at the hypothetical section starting at the given absolute address. The only instructions you can use in this mode are the `RESB` family.

`ABSOLUTE` is used as follows:

```
        ABSOLUTE 0x1A
kbuf_chr        resw 1
kbuf_free       resw 1
kbuf            resw 16
```

This example describes a section of the PC BIOS data area, at segment address 0x40: the above code defines `kbuf_chr` to be 0x1A, `kbuf_free` to be 0x1C, and `kbuf` to be 0x1E.

The user-level form of `ABSOLUTE`, like that of `SECTION`, redefines the `__SECT__` macro when it is invoked.

`STRUC` and `ENDSTRUC` are defined as macros which use `ABSOLUTE` (and also `__SECT__`).

`ABSOLUTE` doesn't have to take an absolute constant as an argument: it can take an expression (actually, a critical expression: see Section 6.7) and it can be a value in a segment. For example, a TSR can re-use its setup code as run-time BSS like this:

```
        org 100h                ; it's a .COM program
        jmp setup               ; setup code comes last
        ; the resident part of the TSR goes here
setup:  ; now write the code that installs the TSR here
        absolute setup
runtimevar1 resw 1
runtimevar2 resd 20
tsr_end:
```

This defines some variables "on top of" the setup code, so that after the setup has finished running, the space it took up can be re-used as data storage for the running TSR. The symbol "tsr_end" can be used to calculate the total size of the part of the TSR that needs to be made resident.

6.10.4 `EXTERN`: Importing Symbols from Other Modules

`EXTERN` is similar to the MASM directive `EXTRN` and the C keyword `extern`: it is used to declare a symbol which is not defined anywhere in the module being assembled, but is assumed to be defined in some other module and needs to be referred to by this one. Not every object-file format can support external variables: the `bin` format cannot.

The `EXTERN` directive takes as many arguments as you like. Each argument is the name of a symbol:

```
        extern _printf
        extern _sscanf, _fscanf
```

Some object-file formats provide extra features to the `EXTERN` directive. In all cases, the extra features are used by suffixing a colon to the symbol name followed by object-format specific text. For example, the `obj` format allows you to declare that the default segment base of an external should be the group `dgroup` by means of the directive

```
        extern _variable:wrt dgroup
```

The primitive form of `EXTERN` differs from the user-level form only in that it can take only one argument at a time: the support for multiple arguments is implemented at the preprocessor level.

You can declare the same variable as `EXTERN` more than once: NASM will quietly ignore the second and later redeclarations. You can't declare a variable as `EXTERN` as well as something else, though.

6.10.5 `GLOBAL`: Exporting Symbols to Other Modules

`GLOBAL` is the other end of `EXTERN`: if one module declares a symbol as `EXTERN` and refers to it, then in order to prevent linker errors, some other module must actually *define* the symbol and declare it as `GLOBAL`. Some assemblers use the name `PUBLIC` for this purpose.

The `GLOBAL` directive applying to a symbol must appear *before* the definition of the symbol.

`GLOBAL` uses the same syntax as `EXTERN`, except that it must refer to symbols which *are* defined in the same module as the `GLOBAL` directive. For example:

```
        global _main
_main:  ; some code
```

`GLOBAL`, like `EXTERN`, allows object formats to define private extensions by means of a colon. The `elf` object format, for example, lets you specify whether global data items are functions or data:

```
        global hashlookup:function, hashtable:data
```

Like `EXTERN`, the primitive form of `GLOBAL` differs from the user-level form only in that it can take only one argument at a time.

6.10.6 `COMMON`: Defining Common Data Areas

The `COMMON` directive is used to declare *common variables*. A common variable is much like a global variable declared in the uninitialised data section, so that

```
        common intvar 4
```

is similar in function to

```
        global intvar
        section .bss
intvar  resd 1
```

The difference is that if more than one module defines the same common variable, then at link time those variables will be *merged*, and references to `intvar` in all modules will point at the same piece of memory.

Like `GLOBAL` and `EXTERN`, `COMMON` supports object-format specific extensions. For example, the `obj` format allows common variables to be NEAR or FAR, and the `elf` format allows you to specify the alignment requirements of a common variable:

```
        common commvar 4:near   ; works in OBJ
        common intarray 100:4   ; works in ELF: 4 byte aligned
```

Once again, like `EXTERN` and `GLOBAL`, the primitive form of `COMMON` differs from the user-level form only in that it can take only one argument at a time.

Chapter 7
C Programming

7.1 Introduction to the C Programming Language

C is a *high-level language*, meaning its statements are oftentimes far-removed from the final machine encoding of the program. C statements provide a more intuitive means of communicating a programmer's wishes to the machine, in contrast to assembly—a *low-level language*, where the programmer must spell out to the machine even the most minute detail of a program. This chapter will serve as a brief introduction to the C programming language; a primer for those who know very little or none, and a reference for those who are familiar. However, this is not meant as an extensive introduction, as there are numerous books devoted to the C programming language. This chapter also attempts to illustrate the similarities between assembly and C, as well as demonstrating techniques of mixing the two languages together.

There are two forms of the C calling convention described in this chapter: 16-bit and 32-bit. As the only C compiler available on the lab machines is the 32-bit GNU C Compiler, this chapter will mainly focus on the 32-bit C calling convention. However, Section 7.3 contains a brief discussion of the 16-bit C calling convention in comparison to the 32-bit calling convention described in this chapter.

7.1.1 GNU C Compiler

Most of this discussion assumes the use of the GNU C compiler version 2.95, which is similar to compilers offered from other major sources, but does contain some unique features and syntaxes.

Just as the NASM assembler generates machine code from assembly statements, the GNU C compiler generates machine code from C statements. To compile a program using GNU C, use the following command at the DOS prompt:

```
gcc [-c] {filename}
```

Compiling with the `-c` option creates an object file only, which must be linked using the `gcc` utility without the `-c` option to the appropriate libraries or other object files to create the final executable. This will be necessary when creating mixed-language programs such as those with both assembly and C.

As the GNU C compiler generates 32-bit machine code, the assembly code linked with its object files needs to use 32-bit registers and instructions. The final executable will be a 32-bit protected mode program (see the protected mode tutorial for more details on protected mode and how it's used in ECE 291).

7.2 Mixing Assembly and C

Often it is a good idea to link assembly language programs or routines with high-level programs which may contain resources unavailable to you through direct assembly programming—such as using C's built in graphics library functions or string-processing functions. Conversely, it is often necessary to include short assembly routines in a compiled high-level program to take advantage of the speed of machine language.

All high-level languages have specific calling conventions which allow one language to communicate to the other; i.e., to send variables, values, etc. The assembly-language program that is written in conjunction with the high-level language must also reflect these conventions if the two are to be successfully integrated. Usually high-level languages pass parameters to subroutines by utilizing the stack. This is also the case for C.

7.2.1 Using Assembly Procedures in C Functions

Procedure Setup

In order to ensure that the assembly language procedure and the C program will combine and be compatible, the following steps should be followed:

- Declare the procedure label global by using the `GLOBAL` directive. In addition, also declare global any data that will be used.
- Use the `EXTERN` directive to declare global data and procedures as external. It is best to place the `EXTERN` statement outside the segment definitions and to place near data inside the data segment.
- Follow the C naming conventions—i.e., precede all names (both procedures and data) with underscores.

Stack Setup

Whenever entering a procedure, it is necessary to set up a stack frame on which to pass parameters. Of course, if the procedure doesn't use the stack, then it is not necessary. To accomplish the stack setup, include the following code in the procedure:

```
push    ebp
mov     ebp, esp
```

`EBP` allows us to use this pointer as an index into the stack, and should not be altered throughout the procedure unless caution is taken. Each parameter passed to the procedure can now be accessed as an offset from `EBP`. This is commonly known as a "standard stack frame."

Preserving Registers

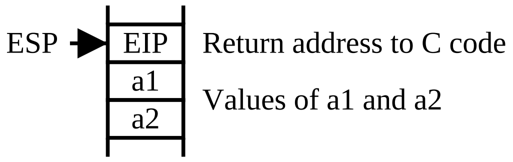
It is necessary that the procedure preserve the contents of the registers `ESI`, `EDI`, `EBP`, and all segment registers. If these registers are corrupted, it is possible that the computer will produce errors when returning to the calling C program.

Passing Parameters in C to the Procedure

C passes arguments to procedures on the stack. For example, consider the following statements from a C main program:

```
           |
extern int Sum();
           |
int a1, a2, x;
           |
x = Sum(a1, a2);
```

When C executes the function call to `Sum`, it pushes the input arguments onto the stack in *reverse* order, then executes a call to `Sum`. Upon entering `Sum`, the stack would contain the following:



Since a1 and a2 are declared as int variables, each takes up one word on the stack. The above method of passing input arguments is called *passing by value*. The code for `Sum`, which outputs the sum of the input arguments via register `EAX`, might look like the following:

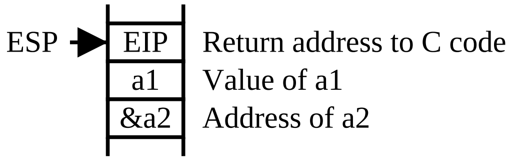
```
_Sum
        push    ebp             ; create stack frame
        mov     ebp, esp
        mov     eax, [ebp+8]    ; grab the first argument
        mov     ecx, [ebp+12]   ; grab the second argument
        add     eax, ecx        ; sum the arguments
        pop     ebp             ; restore the base pointer
        ret
```

It is interesting to note several things. First, the assembly code returns the value of the result to the C program through `EAX` implicitly. Second, a simple `RET` statement is all that is necessary when returning from the procedure. This is due to the fact that C takes care of removing the passed parameters from the stack.

Unfortunately, passing by value has the drawback that we can only return one output value. What if `Sum` must output several values, or if `Sum` must modify one of the input variables? To accomplish this, we must pass arguments by reference. In this method of argument transmission, the addresses of the arguments are passed, not their values. The address may be just an offset, or both an offset and a segment. For example, suppose `Sum` wishes to modify a2 directly—perhaps storing the result in a2 such that a2 = a1 + a2. The following function call from C could be used:

```
Sum(a1, &a2);
```

The first argument is still passed by value (i.e., only its value is placed on the stack), but the second argument is passed by reference (its *address* is placed on the stack). The “&” prefix means “address of.” We say that &a2 is a “pointer” to the variable a2. Using the above statement, the stack would contain the following upon entering `Sum`:



Note that the address of a2 is pushed on the stack, not its value. With this information, `Sum` can access the variable a2 directly. (Hint: use an index register to hold the offset, then use a memory access to access the variable).

Returning a Value from the Procedure

Assembly can return values to the C calling program using only the `EAX` register. If the returned value is only four bytes or less, the result is returned in register `EAX`. If the item is larger than four bytes, a pointer is returned in `EAX` which points to the item. Here is a short table of the C variable types and how they are returned by the assembly code:

Data Type	**Register**
char	`AL`
short	`AX`
int, long, pointer (*)	`EAX`

Allocating Local Data Space on the Stack

Temporary storage space for local variables or data can be created by decreasing the contents of `ESP` just after setting up a stack frame at the beginning of the procedure. It is important to restore the stack space at the end of the procedure. The following code fragment illustrates the basic idea:

```
push    ebp             ; Save caller’s stack frame
mov     ebp, esp        ; Establish new stack frame
sub     esp, 4          ; Allocate local data space of
                        ;  4 bytes
push    esi             ; Save critical registers
push    edi
...
pop     edi             ; Restore critical registers
pop     esi
mov     esp, ebp        ; Restore the stack
pop     ebp             ; Restore the frame
ret                     ; Return to caller
```

7.2.2 Using C Functions in Assembly Procedures

In most cases, calling C library routines or functions from an assembly program is more complex than calling assembly programs from C. An example of how to call the `printf` library function from within an assembly program is shown next, followed by comments on how it actually works.

```
global  _main

extern  _printf

section .data

text    db      "291 is the best!", 10, 0
strformat db    "%s", 0

section .code

_main
        push    dword text
        push    dword strformat
        call    _printf
        add     esp, 8
        ret
```

Notice that the procedure is declared global, and its name must be `_main`, which is the starting point of all C code.

Since C pushes its arguments onto the stack in reverse order, the offset of the string is pushed first, followed by the offset of the format string. The C function can then be called, but care must be taken to restore the stack once it has completed.

When linking the assembly code, include the standard C library (or the library containing the functions you use) in the link. For a more detailed (and perhaps more accurate) description of the procedures involved in calling C functions, refer to another text on the subject.

7.3 16-bit C Programming

While the lab machines do not have a 16-bit C compiler, the 16-bit C calling convention is still very useful when writing 16-bit assembly programs for passing arguments via the stack. While passing arguments in registers may appear to be significantly easier at first glance, as the complexity and number of function arguments increases, the stack is a much better choice for passing arguments. Using the stack also makes procedures easier to write as more registers are free for general use.

Most of the above discussion on the 32-bit C calling convention can be easily translated into 16-bits by using 16-bit registers instead of 32-bit and adjusting offsets accordingly. For example, `SP` and `BP` should be used instead of `ESP` and `EBP` respectively in Section 7.2.1.2 and the first and second arguments in Section 7.2.1.4 are at `[BP+4]` and `[BP+6]`.

However, in a multi-segment 16-bit program, addresses and references are still 32-bit: a 16-bit segment and a 16-bit offset. When multiple code segments are used, `FAR CALL` must be used instead of `CALL`, which offsets stack parameters by an additional 2 bytes because of `CS` being pushed on the stack (also, `RETF` must be used instead of `RET`).

While 8-bit and 16-bit values are still returned in `AL` and `AX` in the 16-bit C calling convention, 32-bit values must be split into two 16-bit chunks. The high-order portion is returned in `DX` and the low-order portion is returned in `AX`.

Chapter 8
Libraries

This chapter describes the use of libraries in developing programs, as well as the details of the LIB291 library used in ECE 291.

8.1 The LIB291 Library of Subroutines

A number of useful routines reside in the LIB291 library file. You are encouraged to use these routines whenever convenient to save programming time. At the end of this section is a detailed explanation of each routine. Following that, the code for two of the routines (`binasc` and `ascbin`) has been included to show you how they work.

`kbdin`

Read in one ASCII character from the keyboard.

`kbdine`

Read in one ASCII character from the keyboard and echo it to the screen.

`dspout`

Type out on the display screen one ASCII character.

`dspmsg`

Type out on the display screen a byte string of ASCII characters.

`dosxit`

Exit from your program back to DOS.

`ascbin`

Convert an ASCII string to an equivalent binary integer.

`binasc`

Convert a binary integer to an equivalent ASCII string.

Note: For some of the Machine Problems, special library files will be provided. These will contain major parts of the Machine Problems which you can use to help develop your code. If you need to use these libraries in order to demonstrate a working program you may; however, there will be penalties for doing so.

8.1.1 Segment Assumptions

Note Carefully: LIB291 is written with the assumption of a single code/data segment which is `GLOBAL` and named `CSEG`. The subroutines also assume that registers `DS`, `SS`, and `SP` have been properly set up before they are called. It is the user's responsibility to set up the code/data segment as `CSEG` and to properly establish `DS` at `CSEG` and to have a usable stack segment pointed to by `SS` and `SP` upon entry.

8.1.2 Using the Library Subroutines

To use the subroutines available in the library file, LIB291, you must do two things:

1. Declare the library subroutine(s) you wish to use as `EXTERN` in your own program:

   ```
   EXTERN dosxit
   ```

2. Include the file name LIB291 in your `TLINK` command:

The `TLINK` program will then search through the file LIB291.LIB and bring in the appropriate subroutines called for by your program.

8.1.3 LIB291 Subroutine Descriptions

`kbdin`

This routine awaits a single character typed in from the keyboard. The ASCII code for that character is returned in register `AL`. Note: `kbdin` does not echo the character back to the display screen.

Exits with:

`AL` = the ASCII code of the character typed in.

`kbdine`

This subroutine is the same as `kbdin` except that in addition it echoes the received character onto the display screen before returning to the calling program.

Exits with:

`AL` = the ASCII code of the character typed in.

dspout

This routine types out on the display screen the ASCII-coded character in `DL`. A typewriter-like format is followed, i.e., successive characters are typed along a line until the end.

Call with:

`DL` = ASCII code of character to be typed on the display.

dspmsg

This routine prints a string of ASCII-coded characters on the display screen. The string must be terminated by an ASCII dollar sign ("$"). The starting offset of the string is to be given as an input in `DX`.

Call with:

`DX` = Offset address of first byte of the ASCII string to be typed.

Example 8-1. Use of `dspmsg` to type out the string `tstmsg` on the display

```
tstmsg  db      'TEST_MSG$'     ; (byte string in data
                                ; segment)
         |
        mov  dx, tstmsg
        call dspmsg             ; type it out
         |
```

dosxit

This routine should be called as the last executed statement to clean up and return control to the DOS system, e.g.,

Example 8-2. Use of `dosxit`

```
         |
done:   call dosxit
```

`dosxit` has no arguments.

ascbin

This routine scans a string of characters (ASCII-coded digits) in successive bytes of memory, and generates a 16 bit binary integer which is the value of that string.

Call with:

`BX` = Offset address of first character of the ASCII string.

Exits with:

`AX` = Signed 16-bit integer having value of the ASCII string.

`BX` = Offset address of the first non-convertible character in the string (e.g., any ASCII character not a decimal digit).

`DL` = Status word giving result of the conversion:
0 if no conversion errors
1 if string had no valid digits
2 if string had too many digits
3 if overflow
4 if underflow (value too negative)

`binasc`

This routine converts a 16 bit binary integer into a string of decimal characters (ASCII-coded digits), writing them as a byte string into memory. Following conversion, the character string may be moved from memory to the display.

Call with:

`AX` = The 16-bit, signed integer to be converted.

`BX` = Starting offset address for a 7-byte buffer to hold the byte string generated.

Exits with:

`BX` = The offset address of the first non-blank character of the string (this may be a minus sign, if the input number was negative). The string will be right-justified within the 7-byte buffer (padded with blanks to the left), and will have a "$" delimiter character after the last digit.

`CL` = Number of non-blank characters generated in the string (including the sign if given). Hence, `CL` = 3 for the number -78, and `CL` = 2 for the number 78.

8.2 Code Examples of `binasc` and `ascbin`

Example 8-3. `binasc` code example

```
; binasc
; By M. C. Loui, 31 Dec 1991
; Converted to NASM by Peter Johnson, Nov 2000

        GLOBAL binasc
```

```
SEGMENT code

; Subroutine binasc
; Converts from binary to ASCII string ending in '$'
; Inputs:  AX = 16-bit signed integer to be converted
;          BX = Starting offset for a 7-byte buffer to hold the result
; Outputs: BX = Offset of first nonblank character of the string
;               (may be a minus sign)
;          CL = Number of nonblank characters generated

saveax  dw      0                       ; To save original input number AX
ten     dw      10

binasc:
        push    si                      ; Save registers
        push    dx
        mov     [cs:saveax], ax

        mov     cl, 0                   ; Initialize count to 0
        mov     si, 5                   ; For SI from 5 downto 0 do

.lp0:
        mov     byte [bx+si], ' '       ;   Initialize output string
        dec     si
        jge     .lp0                    ; End for
        mov     byte [bx+6], '$'        ; End of string

        add     bx, 5                   ; Set BX to point to last char
        cmp     ax, 0                   ; If AX < 0 then
        jge     .lp1
        neg     ax                      ;   Replace by absolute value

.lp1:
        mov     dx, 0                   ; Prepare for double word division
        div     word [cs:ten]           ; Divide by 10
        add     dl, 30h                 ; Convert remainder to ASCII
        mov     [bx], dl
        inc     cl                      ; Another digit

        cmp     ax, 0                   ; AX has quotient
        je      .check
        dec     bx
        jmp     .lp1

        ; Check for negative number
.check:
        cmp     word [cs:saveax], 0
        jge     .done
        dec     bx                      ; If negative, then add minus sign
        mov     byte [bx], '-'
        add     cl, 1

.done:
        pop     dx                      ; Restore registers
        pop     si
        mov     ax, [cs:saveax]
        ret
```

Example 8-4. `ascbin` code example

```
; ascbin
; By M. C. Loui, 31 Dec 1991
; Converted to NASM by Peter Johnson, Nov 2000

        GLOBAL ascbin

SEGMENT code

; Subroutine ascbin
; Converts from ASCII string to binary
; Inputs:  BX = Starting offset of first char of ASCII string
; Outputs: AX = Signed 16-bit number having value of ASCII string
;          BX = Offset of first non-convertible character
;          DL = Status of this call
;               0 if no conversion errors
;               1 if string had no valid digits
;               2 if string had too many digits
;               3 if overflow
;               4 if underflow (too negative)
;
; Revised: 12/17/92 by Tom Maciukenas (ECE291 TA)

ten     dw      10
minus   db      0                       ; 1 if input is negative
digits  db      0                       ; Counts number of digits
status  db      0

ascbin:
        push    si                      ; Save registers
        push    dx

        ; Initialize
        mov     ax, 0
        mov     byte [cs:minus], 0      ; Assume nonnegative
        mov     byte [cs:digits], 0

        ; Skip leading spaces
.spaces:
        cmp byte [bx], ' '
        jne     .signs
        inc     bx
        jmp     .spaces

        ; Check for leading '+' or '-'
.signs:
        cmp     byte [bx], '+'          ; If '+', skip it
        je      .incbx
        cmp     byte [bx], '-'          ; If '-', set minus flag
        jne     .scan                   ;   else scan the number
        mov     byte [cs:minus], 1      ; Remember minus sign

.incbx:
        inc     bx

        ; Scan string
.scan:
        mov     dl, [bx]                ; Check for valid digit
        cmp     dl, '0'
        jb      .end1
        cmp     dl, '9'
        ja      .end1
        jmp     .case                   ; Valid digit - process it

.end1:
        cmp     byte [cs:digits], 0     ; Check for no digits
        jz      .error1
        jmp     .endok

.case:
        inc     byte [cs:digits]        ; Keep track of number of digits
        cmp     byte [cs:digits], 5     ; Check for too many digits
        jg      .error2
        mov     dh, 0                   ; Convert ASCII digit to number
        sub     dx, '0'
        mov     si, dx
        imul    word [cs:ten]           ; Multiply AX by 10
        jo      .error34
        add     ax, si                  ; At this point DX = 0, ..., or 9
        jo      .error34
        inc     bx                      ; Go to next digit
        jmp     .scan

.endok:
        mov     byte [cs:status], 0     ; Normal end
        cmp     byte [cs:minus], 1      ; Negate if necessary
        jne     .done
        neg     ax
        jmp     .done

.error1:
        mov     byte [cs:status], 1
        jmp     .done

.error2:
        mov     byte [cs:status], 2
        jmp     .done

.error34:
        cmp     byte [cs:minus], 1      ; Overflow or underflow?
        je      .ck216
        mov     byte [cs:status], 3
        jmp     .done

        ; Check for -2^16 before declaring underflow (error type 4)
.ck216:
        cmp     ax, 8000h               ; -2^16
        jne     .error4
        cmp     byte [bx+1], ’0’        ; Check that next char
        jb      .ok216                  ;  is not a digit
        cmp     byte [bx+1], ’9’
        ja      .ok216
        jmp     .error4

.ok216:
        mov     byte [cs:status], 0
        jmp     .done

.error4:
        mov     byte [cs:status], 4

.done:
        pop     dx                      ; Restore registers
        pop     si
        mov     dl, [cs:status]
        ret
```

Chapter 9

Data Structures

Data used in computer programs needs to be organized for efficient storage and retrieval. Thus, a critical design decision in the development of any program is the selection of an appropriate data structure and its implementation. This chapter describes arrays, queues, linked lists, binary trees, and hash tables and their implementations on the PC. For further study of these data structures as well as more complex structures, consult the references.

A *data structure* is a collection of data *records* along with a mechanism for insertion, retrieval, and deletion of records. Each record consists of several *fields*, including the information fields that contain the actual data. Other fields of a record may contain *links*, which hold addresses of other records.

9.1 Arrays

The array is a fundamental data structure. Most programming languages provide for arrays as primitive structures. In a *linear array*, each record is associated with a single integer called its *subscript* or *index*. The records in a linear array X of n records are customarily denoted:

$$X(0), X(1), \cdots, X(n-1)$$

For example, the array `Names` given below contains 6 records: `Alice`, `Bobby`, `Cindy`, `David`, `Ellen`, and `Frank`.

`Names(0)`	`Alice`
`Names(1)`	`Bobby`
`Names(2)`	`Cindy`
`Names(3)`	`David`
`Names(4)`	`Ellen`
`Names(5)`	`Frank`

In a d-dimensional array, each record is associated with a vector of d integer subscripts. For example, the 2-dimensional array `Alias` given below has the same set of records as `Names`, but they are organized differently:

i =	**0**	**1**	**2**
`Alias(0,i)`	`Alice`	`Cindy`	`Ellen`

i =	**0**	**1**	**2**
`Alias(1,i)`	`Bobby`	`David`	`Frank`

`Alias` is called a 2 x 3 array, for it has 2 rows and 3 columns.

On the PC, the records of an array are stored in contiguous words to facilitate the computation of the address corresponding to the subscripts. First consider a linear array `Y` with n records, each of which contains b bytes of information. We reserve $n*b$ bytes of memory space for `Y` with the pseudo-op `resb`:

```
Y       resb    n*b
```

To access this array, an address calculation has to be performed. Let y be the offset of the first byte of `Y`. With b bytes per record in `Y`, the address of the first byte of `Y(i)` is:

$$(y + i * b)$$

Suppose that each record of array `Y` comprises one word (b=2). Then to copy `Y(i)` into `AX`, we use the following:

```
        mov     bx, <value of i>
        sal     bx, 1           ; bx = 2*i
        mov     ax, [Y+bx]      ; fetch entry
```

A similar, but more complex calculation must be made for d-dimensional arrays. First one decides how to allocate memory for the records. The Fortran convention is to iterate the leftmost subscript first. Thus the elements of the array `Alias` above would be stored in successive memory words in the following order (memory addresses increase downwards):

```
  Alias(0,0)
  Alias(1,0)
  Alias(0,1)
  Alias(1,1)
  Alias(0,2)
  Alias(1,2)
```

For example, assume that each record of an r x s array `Z` with r rows and s columns has b bytes and that `Z` is defined by:

```
Z       resb    r*s*b
```

If z is the offset of `Z`, then the offset of the first byte of record `Z(i,j)` is given by:

$$z + b * (i + j * r)$$

9.2 Queues

A queue is a list of records in which records are inserted at one end of the list (the *tail* or *rear* of the list), and records are extracted and deleted from the other end (the *head* or *front* of the list). A queue has the

First-In-First-Out (*FIFO*) property: records are removed from the list in the same order as they arrive. An insertion of a record is said to *enqueue* it; similarly, deletion *dequeues* a record.

For example, assume that the names `Alice`, `Bobby`, `Cindy`, and `David` are enqueued in this order onto an initially empty queue:

(head/front)	`Alice`
	`Bobby`
	`Cindy`
(tail/rear)	`David`

If a dequeue operation were performed, then `Alice` would be deleted; if `Ellen` were then enqueued, the queue would become:

	`Alice`
(head/front)	`Bobby`
	`Cindy`
	`David`
(tail/rear)	`Ellen`

If implemented as a linear array, a queue would crawl through memory as records were added and deleted. Even if the queue never held more than two records at any time, the queue might exhaust all the memory allocated for it!

The *circular* queue offers a simple solution to this problem. A circular queue is simply a queue consisting of a finite number of memory locations together with pointers, *`Rear`* and *`Front`*, that point to the last character enqueued and dequeued and "wrap around" from the last to the first memory location assigned to the queue.

For example, suppose the message `CIRCULAR QUEUES!` has been enqueued onto a queue of length `QLen` = 20 bytes that was initially empty, and that the first 3 characters have been dequeued. The 20 memory locations assigned to the queue would then contain:

At this point, additional characters may be dequeued until the queue is empty, and additional characters may be enqueued until the queue is full. The empty/full conditions may be deduced from the relative positions of the `Front` and `Rear` pointers: since `Front=Rear` may mean either empty or full, `Rear` is usually allowed to advance only to just before `Front`; then `Front=Rear` means empty, `Rear=Front-1` (mod `QLen`) means full. The maximum usable queue length is `QLen`-1 when this method is used. Alternately, the number of characters currently in the queue can be monitored - as in the example below.

If several identical queues are used, the queue parameters can be specified conveniently in a `STRUC`ture:

```
QLen    equ     512     ; queue length

STRUC   QSpec
.Front  resw    1       ; index to last char. dequeued
.Rear   resw    1       ; index to last char. enqueued
.QBeg   resw    1       ; pointer to first byte of queue
.Count  resw    1       ; # of bytes currently in queue
.NMsgs  resw    1       ; # of messages pending
```

```
....
ENDSTRUC
```

Two queues (e.g., a Transmit Queue `TQ` and a Receive Queue `RQ`) are then specified as follows:

```
TQBeg   resb    QLen    ; TQ Space
RQBeg   resb    QLen    ; RQ Space
TQ      istruc QSpec
        at .Front, dw -1
        at .Rear, dw -1
        at .QBeg, dw TQBeg
        at .Count, dw 0
        at .NMsgs, dw 0
        ....
        iend
RQ      istruc QSpec
        at .Front, dw -1
        at .Rear, dw -1
        at .QBeg, dw RQBeg
        at .Count, dw 0
        at .NMsgs, dw 0
        ....
        iend
```

The use of the `QSpec` structure simplifies references to the queue parameters: e.g., `TQ+QSpec.Front` refers to the front pointer of the Transmit Queue, and `RQ+QSpec.NMsgs` refers to the number of messages pending in the Receive Queue.

Outlines of subroutines to enqueue and dequeue a character, using `QLen` and the `QSpec` defined above, are shown below:

Subroutine to *enqueue* the character in `AL` at the rear of the queue pointed to by `DI`:

1. If queue is full, set error flag and return immediately.
2. If not,
    a. advance `word [DI+QSpec.Rear]` (modulo `QLen`)
    b. adjust `word [DI+QSpec.Count]`
    c. reset error flag
    d. enqueue the character in `AL` at `byte [word [DI+QSpec.Rear]]`
    e. return

Subroutine to *dequeue* into `AL` the character at the front of the queue pointed to by `DI`:

1. If queue is empty, set error flag and return immediately.
2. If not,
    a. advance `word [DI+QSpec.Front]` (modulo `QLen`)

b. adjust `word [DI+QSpec.Count]`

c. reset error flag

d. dequeue the character at `byte [word [DI+QSpec.Front]]` into `AL`

e. return

9.3 Linked Lists

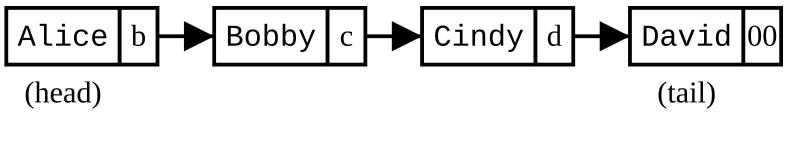
Suppose that we wish to maintain a list of records sorted according to the value of an information field in the records. We may wish to insert a new record at the appropriate point in the list or to delete some record from the list. What data structure should we use? Stacks and queues are inappropriate because they allow records to be inserted or deleted only at the ends of the list. To insert a new record into the middle of a linear array requires a slow, awkward relocation of all records between the insertion point and the end of the array.

An appropriate solution to this problem is the *linked list*, which, at some cost in memory space, permits lists to be constructed and modified easily. In a linked list, each record contains a link field which holds the address of the next record in the list. The sequencing from one record of the list to the next thus involves accessing the link field of each record, rather than stepping along in linear address space. In this fashion, insertions and deletions of records involve merely resetting of links. Because records may now be located anywhere in the memory space allocated for storage, linked lists are appropriate whenever dynamic storage allocation is needed. Linked lists are not needed for storage of static data (e.g., tables of constants) nor in cases where data arrives in orderly fashion (e.g., data buffers).

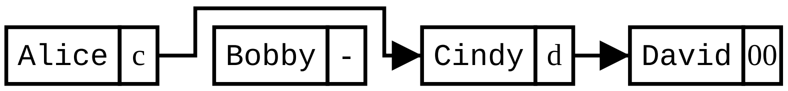
To illustrate, consider the list of names given below: `Alice` is at offset a, `Bobby` is at b, `Cindy` is at c, and `David` is at d. Each cell now has two fields: info and link:

Address	**Info**	**Link**
a:	`Alice`	b
b:	`Bobby`	c
c:	`Cindy`	d
d:	`David`	00

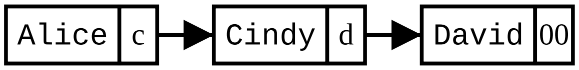
The link field in the last record, `David`, has a special value ‘00’ to mark the end of the list. We draw this list with an arrow from each link field to the record whose address is stored in this link.



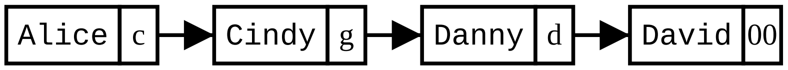
To delete the record `Bobby`, simply change the link field in the record `Alice`: (the space used for `Bobby` is actually saved in another list; see below)



The list now has only three records:



To insert a new record `Danny` at address g between `Cindy` and `David`, merely set the link of `Cindy` to the address of `Danny` and the link of `Danny` to the address of `David`:



On the PC each link field holds a one-word offset in the Data Segment. Thus if the information fields occupy *b* bytes, then the length of each record is *b*+2 bytes. Assume that the link field is located *b* bytes from the beginning of the record; let us define the constant `LINK`:

```
LINK    equ     b
```

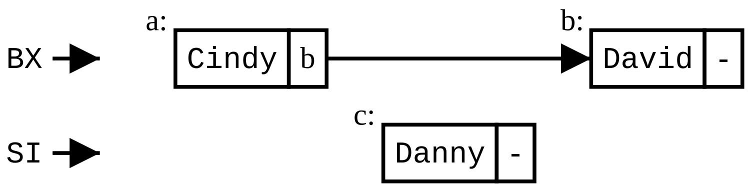
Suppose `BX` holds the offset of (the first byte of) a record in a linked list. Then `[BX+LINK]` specifies the link field of this record. To change `BX` to point to the next record in the list:

```
        mov     bx, [bx+LINK]
```

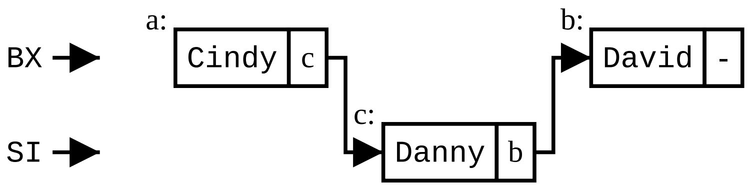
To insert a record whose offset is in `SI` immediately following the record whose offset is in `BX`, we do the following:

```
        mov     ax, [bx+LINK]   ; Copy link
        mov     [bx+LINK], si
        mov     [si+LINK], ax
```

Before:



After:



When a record is deleted from a linked list, it is prudent to recover the memory allocated to the record, rather than simply disconnecting all links to it. We can organize a linked list of unused records, called the Free Storage List (FSL). To allocate space for a new record, we take a record from the FSL, set up the information fields, and link the new record into the data structure. Let the word variable `FSLPtr` hold the offset of the record at the head of the FSL. Deletions and insertions on the FSL always occur at its head; since the FSL has the LIFO property, it is really a stack.

Suppose we wish to delete a record R from a linked list. Assume that `BX` contains the offset of the record immediately preceding R in the list. To delete R, we unlink it from the list and link it onto the head of the FSL:

```
        mov     si, [bx+LINK]   ; SI = offset of R
        mov     ax, [si+LINK]   ; Delete R from list
        mov     [bx+LINK], ax
        mov     dx, [FSLPtr]    ; Insert R onto FSL
        mov     [FSLPtr], si    ;   as the new head record
        mov     [si+LINK], dx
```

To allocate space for a new record, we simply unlink the head record of the FSL:

```
        mov     bx, [FSLPtr]    ; BX = offset of new record
        mov     ax, [bx+LINK]   ; Update FSLPTR
        mov     [FSLPtr], ax
```

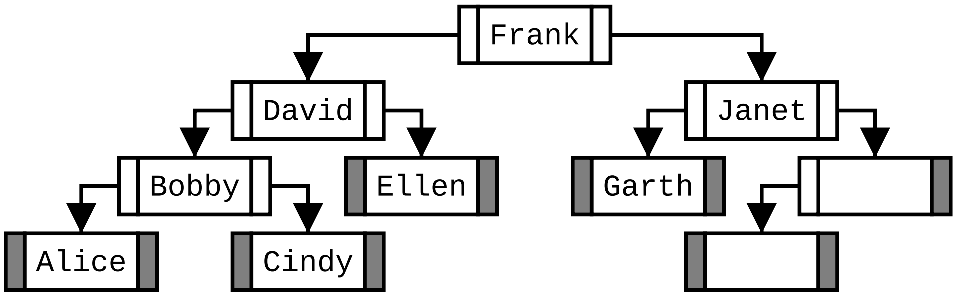
The use of linked lists arose early (1962) in the development of artificial intelligence research. The linked list is a fundamental data structure of the LISP language, which is heavily used for artificial intelligence programming. Many variations on the idea of linked cells have been subsequently introduced. For example, a doubly linked list has both forward and backward links to facilitate searching in the list. The binary trees discussed below also use more than one link per record.

9.4 Binary Trees

To locate a record with a particular information field in a linked list, one starts at the head of the list and traces through successive records, one at a time. This process can take a long time, even with sorted lists. to reduce the time, we would like to avoid inspecting many records. The *binary tree* data structure permits faster searches than linked lists, but requires more space because it has more link fields in each record.

In a binary tree, each record is stored in a *node*. For each node X, at most one node Y is the *left child* of X and at most one node Z is the *right child* of X. In other words, any node X may have 0, 1, or 2 children. X is the common *parent* of nodes Y and Z. There is one node in the tree with no parent; it is called the *root* of the tree.

In the following tree, each record has a link that points to its left child and a link that points to its right child. (For some applications the implementation of the binary tree could include a pointer from each node to its parent.) `Bobby` is the left child of `David`, and `Ellen` is the right child of `David`. `David` is the parent of both `Bobby` and `Ellen`. `Frank` is the root of the tree.



Notice the arrangement of names in this tree. All names in the left subtree of any node are lexicographically less than the name at that node; that is, they would occur earlier in an alphabetic sort. All names in the right subtree are lexicographically greater. For instance, the left subtree of `Frank` is the tree rooted at `David`, and all names in this subtree are lexicographically less than `Frank`. Thus the location of a record in the tree expresses its relationship to other records.

This arrangement of names permits rapid insertion of new records. Starting at the root of the tree, compare the new name with the name at the current node. If the new name is "smaller," then proceed to the left child; if the new name is "larger," then proceed to the right child. For example, to insert `Harry` into the tree, inspect `Frank`, `Janet`, and `Garth` in that order and then make `Harry` the right child of `Garth`. In similar fashion one can search the tree for a specific name.

If a tree with n records is well balanced, then the maximum number of records inspected during an insertion is approximately $\log_2 n$. In contrast, for linear lists this maximum number would be n. The references describe techniques for keeping trees well balanced after both insertions and deletions.

9.5 Hash Tables

In their quest for ever faster searching methods, several IBM engineers and scientists discovered hashing in the 1950's. The idea is to transform the information itself into a subscript in a linear array. Let the array that holds the data be T(0), ..., T(n-1). The array T is called a *hash table*. A *hash function* h transforms the information x into an integer h(x) such that:

$$0 \leq h(x) \leq n-1$$

The information x is then stored at T(h(x)), together with any additional information fields associated with x. If the record T(h(x)) is already in use, then a *collision* occurs, and x must be stored elsewhere. A good hashing scheme minimizes the frequency of collisions by scattering information into random locations in the hash table. The choice of hash functions and the resolution of collisions are discussed below.

The choice of n affects the performance of a hashing scheme. If n is much larger than the number of records to be stored, then collisions are infrequent, but space is wasted.

9.5.1 Hash Functions

Perhaps the simplest hash function maps x, interpreted as a positive integer, to its remainder when divided by n:

$$h(x) = x \bmod n$$

If x consists of more than one word, then one can compress x into one word by forming the exclusive-or of the words x_1, ..., x_k that constitute x:

$$h(x) = (x_1 \text{ XOR } \cdots \text{ XOR } x_k) \bmod n$$

This method of compression generally works well in practice, for every bit of x participates in the computation. For example, suppose the names `Alice`, `Bobby`, `Cindy`, `David`, and `Ellen` are inserted into a hash table with n = 7. On the PC with this compression method we obtain the following table `Names`:

`Names(0)`	
`Names(1)`	`Bobby`
`Names(2)`	`Ellen`
`Names(3)`	`Cindy`
`Names(4)`	`Alice`
`Names(5)`	
`Names(6)`	`David`

9.5.2 Collision Resolution

The two fundamental collision resolution schemes are *linear probing* and *separate chaining*. Many other collision resolution schemes have been devised. They are described in the references.

In linear probing, when a collision occurs during insertion at T(i), one probes sequentially among T(i+1), T(i+1), ..., T(n-1), T(0), T(1), etc. For example, in the hash table `Names` above, an attempt to insert `Frank` causes a collision with `David` at `Names(6)`. With linear probing, `Frank` would be stored at `Names(0)`. An attempt to insert `Garth` into `Names` causes a collision with `Ellen` at `Names(2)`. With linear probing, since `Names(3)` and `Names(4)` are already occupied, `Garth` must be stored in `Names(5)`. Although linear probing permits very rapid insertions, deletions are difficult.

In separate chaining, each table entry T(i) is a pointer to the head of a separate linked list. All records that are mapped to the same subscript are stored in the same list. With separate chaining both insertions and deletions are straightforward. In essence, separate chaining partitions a list into n pieces. Thus an insertion into a hash table with separate chaining could be n times faster than insertion into a sorted linked list.

References

A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1982.

D. E. Knuth. *The Art of Computer Programming*. Vol. 1 and vol. 3, Addison-Wesley, 1973.

E. M. Reingold and W. J. Hansen. *Data Structures*. Little, Brown, 1983.

H. R. Lewis and L. Denenberg. *Data Structures and their Algorithms*. Harper Collins, 1991.

Chapter 10
I/O Devices

10.1 Keyboard

10.1.1 Keyboard Interface Hardware

The keyboard unit contains an Intel 8048 microcontroller that is programmed to scan the keyboard for key presses and releases (each counts as an individual keystroke), debounce the keystrokes, implement the “typematic” (hold-to-repeat) feature, maintain a 16-keystroke buffer, and transmit each keystroke serially to the PC’s system unit. There are two bidirectional data lines in the cable connecting the keyboard unit to the system unit: KBD DATA carries either the serial keystroke data from the keyboard or a “clear-and-reenable” handshaking signal from the system unit; KBD CLK carries either the baud rate clock from the keyboard unit or a “clock disable” control signal from the system unit. (Note that when the keyboard clock is disabled, the keyboard does not even respond to **CTRL-ALT-DEL**.

Keystroke data are transmitted serially at 10,000 baud over the KBD DATA line together with the baud rate clock on the KBD CLK line. Each character transmitted consists of 2 start bits and 8 data bits; there are no stop bits. The line is 0 (low) when idle, the start bits are 1 (high). The data bits are transmitted LSB first; bits 0-6 are the “scan code” which uniquely identifies the key by its position on the keyboard, bit 7 (MSB) is 0 for key press and 1 for key release. Holding a key down for more than half a second invokes the “typematic” action: key press scan codes are sent repeatedly at the rate of 10 per second without intervening key release scan codes, until the key is released.

In the system unit the character received on the KBD DATA line is reconverted to parallel format, gated into port `60h`, and a interrupt is sent on `IRQ1` to the Interrupt Controller. Since the baud rate clock is transmitted along with the data, the circuit needed to deserialize the data is significantly simpler than a UART; it is essentially a serial-in, parallel-out shift register. The Interrupt Controller triggers interrupt 9 for `IRQ1`.

The interrupt 9 handler must send an End-of-Interrupt signal to the Interrupt Controller, and on the original IBM PC also needs to acknowledge the reception of the character by sending a clear-and-reenable handshaking signal to the keyboard unit over the KBD DATA line. This is done by setting bit 7 of port `61h` to 1 and back to 0. (The other bits of port `61h` should be left unchanged since they control other functions. E.g., bits 0 and 1 enable the built-in speaker, and bit 6 disables the keyboard clock.) On more recent machines, this acknowledgment is unnecessary but not harmful.

The use of scan codes together with the key press/release information makes it easy to assign arbitrary meanings to the keys, e.g., to convert the standard QWERTY keyboard layout to the Dvorak layout; to discriminate between different keys having the same labels such as the number keys in the main keyboard and the numeric keypad, the left and right **SHIFT** keys, etc.; to handle special key combinations for “hot key” applications; and to identify the sequence in which certain keys have been pressed and released. For normal typing and character entry, on the other hand, keystrokes should simply be converted to ASCII codes; this is done by the default interrupt 9 handler in the system BIOS, described below.

10.1.2 Keyboard Interrupt 9 Handler

KBD_INT, the name given to the default BIOS Keyboard Interrupt 9 Handler, reads the scan code from port `60h`, sends the clear-and-reenable handshaking signal to the keyboard unit, processes the scan code, sends an End-of-Interrupt signal to the Interrupt Controller (code `20h` to port `20h`), and returns from the interrupt.

The scan code processing performed in the KBD_INT routine in BIOS consists of the following tasks:

1. Intercept the following special key combinations:
   - **CTRL**-**ALT**-**DEL** (invokes a system reset)
   - **CTRL**-**BREAK** (invokes interrupt `1Bh`. By default an immediate `IRET` is performed unless the user has installed an interrupt `1Bh` handler)
   - **CTRL**-**NUM LOCK** (enters a SUSPEND state, i.e. waits in a loop within KBD_INT until any key other than **NUM LOCK** is pressed)
   - **SHIFT**-**PRTSC** (invokes interrupt `5`)

2. Maintain a record of the state of the **SHIFT**, **CTRL**, **ALT**, **CAPS LOCK**, **NUM LOCK**, **SCROLL LOCK**, and **INSERT** keys. This invokes monitoring key presses as well as releases, and suppressing the typematic action of the LOCK and **INSERT** keys to get toggle action.
3. Convert the scan code for any other key press (and for **INSERT**) into a two-byte “extended ASCII” code representation and store it in a 16-word circular “type-ahead” buffer KB_BUFFER (or sound a beep if the buffer is full)
4. “Compose” an ASCII code for digits typed on the numeric keypad while **ALT** is held down. The code is the number (modulo 256).

Shift Status Bytes

Task 2 maintains two “shift mode status” bytes, located at `0040:0017h` and `0040:0018h` in IBM-compatible BIOS, indicating the following:

Table 10-1. Meaning of Shift Status Byte at `0040:0017h`

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
INSERT state active	**CAPSLK** state active	**NUMLK** state active	**SCRLK** state active	**ALT** key down	**CTRL** key down	**LSHIFT** key down	**RSHIFT** key down

Table 10-2. Meaning of Shift Status Byte at `0040:0018h`

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
INSERT key down	**CAPSLK** key down	**NUMLK** key down	**SCRLK** key down	**SUSPEND** state active	0	0	0

KB_BUFFER, the type-ahead buffer maintained by Task 3, is independent of the 16-keystroke buffer maintained within the keyboard unit. For every keypress other than shift mode keys (but including **INSERT**) an entry is made in KB_BUFFER, modified as appropriate by the status of the various shift mode keys. Note that **CAPS LOCK** affects only the letter keys, **NUM LOCK** only the number keys in the numeric keypad, and that the effect of either key is reversed when **SHIFT** is also pressed. Note further that when **ALT**, **CTRL**, and **SHIFT** are pressed in combination (other than the **CTRL**-**ALT**-**DEL** “system reset” case) their precedence is **ALT** first, **CTRL** second, **SHIFT** last.

User-defined Interrupt 9 Handlers

Writing your own interrupt 9 keyboard handler is relatively straightforward since the hardware does most of the work for you. If you use your own interrupt 9 handler, none of the above functions will happen, but you can check for things the normal handler won’t check for.

Your interrupt handler will be a normal interrupt service routine. The only special requirement is that it acknowledges reception of the keyboard event by toggling bit 7 of port 61h to 1 and back to 0. The other bits of port 61h must not be modified, since they control other hardware. This is only required for full original IBM PC compatibility. The following code example is a skeleton of an interrupt 9 handler:

Example 10-1. Interrupt 9 (Keyboard) Handler

```
KbdInt
        push    ax              ; Save registers
        push    ds              ;
        mov     ax, cs          ; Make sure DS = CS
        mov     ds, ax          ;
        in      al, 60h         ; Get scan code
                ¦               ;
                ¦               ; Process event
                ¦               ;
        in      al, 61h         ; Send acknowledgment without
        or      al, 10000000b   ;   modifying the other bits.
        out     61h, al         ;
        and     al, 01111111b   ;
        out     61h, al         ;
        mov     al, 20h         ; Send End-of-Interrupt signal
        out     20h, al         ;
        pop     ds              ; Restore registers
        pop     ax              ;
        iret                    ; End of handler
```

The procedure for installing an interrupt 9 handler is exactly the same as that for installing an interrupt 1Ch timer interrupt routine, except that the interrupt 9 vector is located at address `0000:0024` (segment address `0000`, offset `0024h`). Remember to save the old vector and restore it before your program exits.

10.1.3 Library Procedures for Keyboard Input

Keyboard routines in LIB291

`kbdine`

This routine waits for a character to become available, echoes the character to the display, and then returns with the character stored in `AL`.

`kbdin`

This routine is the same as KBDINE except the character is not echoed.

One thing to note about the above routines is that they will wait indefinitely until a key is typed. If this is not desired, one should check to see if a key is available before calling `kbdine` or `kbdin`. For this you call interrupt 16h with `AH`=1, which returns with the zero flag `ZF`=1 if no character has been typed, or `ZF`=0 if a character is available. An example of how to use this interrupt is shown below:

Example 10-2. Using Interrupt 16h with `AH`=1

```
mov     ah, 1   ; Get the status of keyboard
                ;  buffer
int     16h     ; ZF=1 if buffer is empty
jz      .nochar ; Do something else if buffer is
                ;  empty
call    kbdine  ; Get the character
        ¦
        ¦
```

10.1.4 BIOS Function Calls

Several useful BIOS routines for the keyboard can be accessed by using `INT 16h` with `AH`=0, 1, or 2.

Subfunction 0 (i.e. `AH`=0) waits, if necessary, until an entry is present in the BIOS type-ahead buffer KB_BUFFER, then removes the entry from KB_BUFFER into register `AX`. The scan code (or second code) is in `AH` and the ASCII code (or 00h) is in `AL`.

Subfunction 1 returns with the entry at the tail of KB_BUFFER copied into `AX` (but not removed from the buffer) and the zero flag set/reset if the buffer is/isn't empty. I.e., if `ZF`=1, a new entry is not available, and the entry copied into `AX` was typed 16 entries ago; if `ZF`=0, `AX` contains the new entry (subfunction 0 must be used to actually remove the entry from KB_BUFFER). Subfunction 1 may be used to "preview" a character such as **CTRL**-**C** before it is acted upon.

Subfunction 2 copies the first "shift status" byte (see Table 10-1) into `AL`.

Subfunction 0 and 1 recognize **CTRL**-**BREAK** and invoke interrupt 1Bh. The default interrupt 1Bh handler in BIOS is an `IRET`. A user-written interrupt 1Bh handler may be installed, but care must be taken to issue End-of-Interrupt commands for all hardware interrupts that happen to be in service when **CTRL**-**BREAK** was pressed, and to reset those hardware devices.

Here is a short table summarizing the `INT 16h` functions above:

`INT 16h` Function	Description

AH=0	Checks (and waits) for a keypress. Removes it from buffer.
	AL ← ASCII code
	AH ← scan (or second) code

AH=1	Checks (but doesn't wait) for a keypress. Leaves it in buffer. ZF ← 0 if an entry is present. 1 if the buffer is empty. If an entry is present, it is copied (not removed) just as above.
AH=2	Copies the first shift status byte to AL.

10.1.5 DOS Function Calls

In addition to the BIOS calls, there are several DOS function calls which also provide keyboard services. The calls are performed using `INT 21h` with AH=01h, 06h, 07h, 0Ah, 0Bh, and 0Ch. These DOS function calls use the BIOS calls described above. Some functions recognize **CTRL-BREAK** and **SHIFT-PRTSC**; note that **CTRL-BREAK** from a DOS function invokes interrupt 23h.

DOS functions 01h, 07h, or 08h wait (if necessary), read a new buffer entry, and return the ASCII code in register AL; an ASCII code of 00h indicates that a second DOS function call is needed to get the "second code" into AL. Function 01h also echoes the character to the display, functions 07h and 08h don't. Functions 01h and 08h recognize **CTRL-BREAK** and **SHIFT-PRTSC**; function 07h ignores them.

DOS function 06h, when used for input by setting DL = -1, is similar to function 07h but does not wait for an entry: if no entry is ready the function returns with the zero flag set, otherwise the zero flag is reset and the ASCII code is returned in AL.

DOS function 0Ah allows input of an entire input string. The characters are echoed to the display as they are entered; the string is terminated with **ENTER**. Keys requiring a "second code" are ignored. If the allowed maximum length of the string (including the terminating **ENTER**) is N characters, an (N+2)-byte input buffer pointed to by `DS:DX` must be set up, with byte 0 set to N. When the complete string has been input, byte 1 of the buffer will be set to the actual character count (*not* including the terminating **ENTER**); the characters of the string, including the **ENTER**, start in byte 2. A beep sounds if the string is too long, i.e., if the Nth character is not **ENTER**.

DOS function 0Bh returns 0/-1 in register AL if KB_BUFFER is/isn't empty. This function recognizes **CTRL-BREAK**.

DOS function 0Ch, with AL = 01h, 06h, 07h, 08h, or 0Ah, first clears KB_BUFFER and then invokes the DOS function specified in AL

Here is a short table summarizing the `INT 21h` functions above:

`INT 21h` Function	Description

`INT 21h` Function	Description
`AH`=01h `AH`=07h `AH`=08h	Checks (and waits) for a keypress. Removes it from buffer. `AL` ← ASCII code Function 01h also echoes the character.
`AH`=06h (`DL`=-1)	Checks (but doesn't wait) for a keypress. Leaves it in buffer. `ZF` ← 0 if an entry is present. 1 if the buffer is empty. If an entry is present, it is copied (not removed) just as above.
`AH`=0Ah	Inputs an entire string. Characters are echoed as they are typed. ENTER terminates the entry. (See above for more information).
`AH`=0Bh	`AL` ← 0 if the buffer is empty, -1 if it isn't empty.
`AH`=0Ch	With `AL`=01h,06h,07h,08h,0Ah, performs the same functions as above (function is in `AL`) but first clears the buffer.

10.1.6 Extended ASCII Codes

Scan Codes

The KB_BUFFER entries are "extended ASCII" codes consisting of two bytes. For letter, number, and punctuation keys (combined with the status of **SHIFT**, **CAPS LOCK**, **NUM LOCK**, or **CTRL**) the extended ASCII code consists of the key's scan code in the high byte (with MSB = 0 for a key press) and the corresponding ASCII code in the low byte. Scan codes for the standard and the AT keyboards ae shown below. Key releases are indicated by the scan code MSB = 1 and the low 7 bits set to the scan code.

Table 10-3. Scan Codes Assigned to Keys on the Standard or AT Keyboard

Key	Scan Code
ESCape	1
1 2 3 4 5 6 7 8 9 0 - =	2-13
Backspace	14
Tab	15
Q W E R T Y U I O P [ ]	16-27
ENTER	28
CTRL	29
A S D F G H J K L ; ' '	30-41
SHIFT (left)	42

Key	Scan Code
\	43
Z X C V B N M , . /	44-53
SHIFT (right)	54
Print Screen	55
ALT (left)	56
SPACEBAR	57
CAPS LOCK	58
F1 to F10	59-68
NUM LOCK	69
SCROLL LOCK	70
7 8 9 (Numeric keypad)	71-73
gray -	74
4 5 6 (Numeric keypad)	75-77
gray +	78
1 2 3 (Numeric keypad)	79-81
0 (Numeric keypad)	82
DELETE	83
SYSTEM REQUEST	84

ASCII codes composed on the numeric keypad with **ALT** held down return a 00h scancode.

Second Codes

Keys which have no standard ASCII representation (**F1**, **PageUp**, **Insert**, etc.) are stored with an ASCII code of 00h in the low byte and a "second code" (usually, but not always, the scan code) in the high byte, as shown in Table 10-4. Key combinations not shown are ignored.

Table 10-4. "Second Codes" for ASCII Code 00h Key Combinations

Key	Scan Code
NUL character	3
SHIFT-TAB	15
ALT-Q,W,E,R,T,Y,U,I,O,P	16-25
ALT-A,S,D,F,G,H,J,K,L	30-38
ALT-Z,X,C,V,B,N,M	44-50
F1 to F10	59-68
HOME	71
UP ARROW	72
PAGE UP	73
LEFT ARROW	75
RIGHT ARROW	77

Key	Scan Code
END	79
DOWN ARROW	80
PAGE DOWN	81
INSERT	82
DELETE	83
SHIFT-F1 to F10	84-93
CTRL-F1 to F10	94-103
ALT-F1 to F10	104-113
CTRL-PRINT SCREEN	114
CTRL-LEFT ARROW	115
CTRL-RIGHT ARROW	116
CTRL-END	117
CTRL-PAGE DOWN	118
CTRL-HOME	119
ALT-1,2,3,4,5,6,7,8,9,0,-,=	120-131
CTRL-PAGE UP	132
F11, F12	133, 134

10.1.7 Applications

Monitoring How Long a Key is Pressed

When the keyboard is to be used for real-time control of a simple sound synthesizer, or the motors in a robot, etc., it may be necessary to monitor not only key presses but also key releases, possibly for several keys at once. Of course, the KBD_INT interrupt 9 handler in BIOS does exactly that for the shift keys, but not for other, arbitrary keys. A substitute interrupt 9 handler would differ from KBD_INT only in the way the scan codes sent from the keyboard unit are processed. It would most likely maintain a “status word” in which individual bits are set or reset according to whether the corresponding keys are pressed or released. This status word can then be monitored by the main program.

“Hot Keys”

A *hot key* is a key combination that activates a resident program, temporarily suspends whatever application program is running, performs a specific task, and then returns control to the application program. A hot key thus acts like an interrupt, as in **SHIFT-PRTSC**. E.g., a hot key combination might be used to display the time in the right hand upper corner of the screen for a few seconds.

Key combinations that are unlikely to be used normally are candidates for this purpose, particularly key combinations that are ignored by KBD_INT, such as **ALT** and **gray +** (key #78). A solution which is simple, elegant, and does not interfere with the normal operation of KBD_INT, is to write a “preprocessor” for interrupt 9 which intercepts and processes **ALT** and key #78 scan codes, ignores all others, and then exits to the original KBD_INT routine to send the acknowledge signal to the keyboard, process the scan codes, send E-o-I to the Interrupt Controller, and return from the interrupt. The preprocessor either maintains a flag bit which is set/reset when **ALT** is pressed/released, or tests the KBD_INT Shift Status byte at `0040:0017h`, and invokes the desired hot-key action when the **ALT** mode is active *and* key #78 is pressed.

Shown below is the outline of a hot-key routine that displays the current time on the screen for a few seconds whenever **ALT** & key #78 is pressed. Because hot key routines, like interrupts, may be invoked at any time (in particular, during execution of a DOS function), and DOS is not a reentrant operating system, DOS functions cannot be used in hot-key routines. Hence, BIOS call 1Ah is used to get the Time-of-Day value, rather that the much simpler DOS function 2Ch.

Display subroutine

1. Get Time-of-Day (BIOS call 1Ah) in `CX:DX`
2. Convert to HH:MM:SS format
3. Save contents of upper right hand corner of screen
4. Display the current time there
5. Delay about 2 seconds
6. Restore original screen contents
7. Return

Preprocessor (invoked when an interrupt 9 occurs)

1. Enable interrupts
2. Save working registers
3. Maintain/check **ALT** mode flag
4. If key #78 and **ALT**-mode, call Display subroutine
5. Restore working registers
6. Exit to old type-9 interrupt vector

The program to install the hot-key routine must “chain” interrupt 9 and then exit to DOS, but leave the preprocessor and all routines required for the hot-key action resident. The old interrupt 9 vector should be saved in a doubleword so the preprocessor can exit to it with a JMP dword [oldvect] before the interrupt 9 vector is set to point to the preprocessor. DOS function 31h is used to “terminate-but-stay-resident” by setting `DX` to the number of 16-byte paragraphs to be kept resident (including 16 paragraphs for PSP, the Program Segment Prefix), and `AL` to an exit code that can be examined by batch commands.

10.2 Mouse

The mouse is controlled using the mouse functions at interrupt 33h. There are many functions, but this section will only cover the basic set required to get things going. For more information, see the references on the web page.

In order to use the mouse, you must first call Function 0000h (Reset Driver and Read Status). This initializes the drivers and hardware. The mouse cursor will initially be hidden, so you must use Function 0001h (Show Mouse Cursor) to make it visible. From then on, just call Function 0003h (Return Position and Button Status) to get the position and button status whenever your program needs it. Make sure you hide the mouse cursor before your program exits.

10.2.1 Mouse Interrupts (INT 33h)

Function 0000h: RESET DRIVER and READ STATUS

This function initializes the hardware and software so the mouse is ready to be used. The mouse will initially be hidden.

Inputs

AX = 0000h

Outputs

AX = Status
0000h : Error. Hardware/software not installed.
FFFFh : OK. Hardware/software installed.

BX = Number of buttons
FFFFh : Two buttons.
0000h : Other than two buttons.
0003h : Three buttons.

Function 0001h: SHOW MOUSE CURSOR

This function makes the mouse cursor visible on the screen. If you are programming text or graphics by writing directly to the video memory, you should hide the mouse cursor before doing so to stop the mouse from leaving graphic junk on the screen.

Inputs

`AX` = 0001h

Outputs

(None)

Function 0002h: HIDE MOUSE CURSOR

This function makes the mouse cursor invisible. Multiple calls to this function require multiple calls to Function 0001h (Show Mouse Cursor) before the mouse cursor will appear again, because the mouse driver keeps a count of the number of times the mouse has been hidden.

Inputs

`AX` = 0002h

Outputs

(None)

Function 0003h: RETURN POSITION AND BUTTON STATUS

This function returns the current mouse cursor position and button status. Position is measured in pixels, with the origin (0,0) at the upper left corner of the screen. In text mode, each character is assumed by the mouse driver to correspond to eight pixels horizontally and eight pixels vertically. Thus, to get the row and column position of the mouse cursor in text mode, divide the values in `CX` and `DX` by eight.

Inputs

`AX` = 0003h

Outputs

`BX` = Button status (1 = corresponding button pressed)

Bit 0 : Left mouse button.

Bit 1 : Right mouse button.
Bit 2 : Middle mouse button (if present).
Bits 3-15 : Cleared to 0.

`CX` = Pixel column position.
`DX` = Pixel row position.

Function 0004h: POSITION MOUSE CURSOR

This function will position the mouse cursor on the screen. As in function 0003h, position is measured in pixels, with the origin (0,0) at the upper left corner of the screen. See the description of function 0003h for more information.

Inputs

`AX` = 0004h
`CX` = Column position
`DX` = Row position

Outputs

(None)

Function 0007h: DEFINE HORIZONTAL CURSOR RANGE

This function will limit the horizontal position of the mouse cursor to a defined section on the screen. The column positions are given in pixels.

Inputs

`AX` = 0007h
`CX` = Leftmost column boundary
`DX` = Rightmost column boundary

Outputs

(None)

Function 0008h: DEFINE VERTICAL CURSOR RANGE

This function will limit the vertical position of the mouse cursor to a defined section on the screen. The row positions are given in pixels.

Inputs

`AX` = 0008h
`CX` = Upper row boundary
`DX` = Lower row boundary

Outputs

(None)

10.2.2 Changing the Mouse Cursor

When the mouse cursor is enabled, the actual appearance of the cursor is dependent on the current video mode. If the video mode is text, then the cursor defaults to a character-sized block of color. If the video mode is a graphics mode, then the cursor appears as an arrow. Oftentimes you will want to change the appearance of the mouse cursor to better facilitate the application in which you are using it. For example, if the application is a paint program, you may want to the mouse cursor to appear as a paint brush perhaps. In a video game, you may want the mouse cursor to appear as a crosshair for targeting enemy space ships.

There are two basic ways in which this can be accomplished. First, you could manually create your own mouse “cursor” by simply reading the position of the mouse and manually drawing and erasing whatever graphic image you desire. However, you would necessarily have to be sure to restore the contents of the screen under which the mouse cursor is moving as the mouse is repositioned.

The second method is to use functions 0009h and 000Ah of `INT 33h` to redefine the appearance of the mouse cursor. This method is more attractive because you do not need to concern yourself with restoring the screen contents under the mouse cursor as it moves this would be done automatically as it normally is. Function 0009h is used more often than 000Ah, so it is the only one discussed below.

Function 0009h:

This function will redefine the appearance of the mouse cursor when the screen is in a *graphics* mode.

Inputs

`AX` = 0009h
`BX` = Column of cursor hot spot in bitmap (-16 to 16)
`CX` = Row of cursor hot spot in bitmap (-16 to 16)
`ES:DX` = Pointer to cursor bitmap

Outputs

(None)

The *hot spot* is a term given to the pixel location within the mouse cursor image whose coordinate on the screen is the same as the position of the mouse cursor. Essentially, this hot spot allows us to know where the entire image is located on the screen relative to the mouse position (returned in function 0003h, for instance). Initially, the hot spot is in the upper-left corner of the default mouse cursor (the arrow).

The *cursor bitmap* can be a 16x16 pixel image which is defined in memory as follows:

Offset	Size	Description
00h	16 words	Screen Mask
20h	16 words	Cursor Mask

Each word defines the sixteen pixels of a row, with the rightmost pixel being the least significant bit. The image is defined beginning with the top row of pixels in the image.

The image is formed on the screen by first ANDing the pixels on the screen with the Screen Mask image, then XORing the pixels on the screen with the Cursor Mask image.

10.3 8253 Timer Chip

The 8253 Timer contains 3 independent channels. Each channel consists of a 16-bit downcounter with a CLOCK input, a GATE input for enabling/triggering the count, and a counter output (OUT), a 16-bit COUNT register for holding the count value, and a CONTROL register for controlling the operation of the counter and the loading/reading of the COUNT register. Each channel may count in one of six modes (interrupt on terminal count, hardware retriggerable one-shot, rate generator, square wave generator, software triggered strobe, and hardware triggered strobe) and may count in BCD or binary. The output is formed by copying the contents of the channel's COUNT register to the channel's counter and starting the downcount. Depending on the mode selected, the GATE input may act as an enable input, or as a trigger to start the downcount; similarly, the downcounter may automatically reload the COUNT and repeat, or require a reload/retrigger (one-shot operation). Consult the Intel 8253-5 Programmable Interval Timer data sheet for more details.

The CONTROL register of a channel is loaded by writing a control byte to I/O port 43h. The interpretation of the control byte is shown in Table 10-5; note that bits 7 and 6 determine which channel is affected.

Table 10-5. Interpretation of the Timer Control Byte

Bits 7,6:	Channel ID (11 is illegal)

Bits 5,4:	Read/load mode for two-byte count value: 00 — latch count for reading 01 — read/load high byte only 10 — read/load low byte only 11 — read/load low byte then high byte
Bits 3,2,1:	Count mode selection (000 to 101)
Bit 0:	0/1: Count in binary/BCD

The 16-bit COUNT registers of channels 0, 1, and 2 are located at I/O ports 40h, 41h, and 43h, resp. Each COUNT register must be loaded according to the mode selected in the CONTROL byte for that channel; single-byte loads leave the other byte 0. The COUNT register may be read “on the fly” by latching the current count from the downcounter into the COUNT register while the downcounter continues counting.

In the PC all three channels use a 1.19318 MHz signal as clock input. GATE0 and GATE1 are permanently tied to 1, so the outputs of Channels 0 and 1 are continuous. The channels are programmed during the BIOS power-up initialization sequence as follows:

The CONTROL byte for Channel 0 is 00110110b Channel 0, 2-byte count value, mode 3 (continuous symmetrical square wave), count in binary. The COUNT value for Channel 0 is 0000h, i.e., 65536 counts, so the frequency of OUT0 is 1.1931817 Mhz/65536 $\approx$ 18.2 Hz. Channel 0’s output is connected to the IRQ0 Interrupt Request line of the 8259 Interrupt Controller; hence an interrupt 08h will occur at a 18.2 Hz rate, or once every 55 msec. The interrupt 08h handler maintains the PC’s time-of-day clock and performs other internal timing functions. To simplify the use of the timer interrupt for user applications (and to minimize interactions with the internal timing functions), the interrupt 08h handler issues a software interrupt 1Ch which is vectored during initialization to the “default interrupt handler” (an `IRET`).

The CONTROL byte for Channel 1 is 01010100b Channel 1, 1-byte (LSB) count value, mode 2 (rate generator), count in binary. The COUNT value for Channel 1 is (00)12h = 18, so the frequency of OUT1 is 1.1931817 Mhz/18 $\approx$ 66 kHz. Channel 1 controls the refresh timing of the memory.

The CONTROL byte for Channel 2 is 10110110b Channel 2, 2-byte count value, mode 3 (symmetrical square wave, continuous provided OUT2 = 1), count in binary. The COUNT value for Channel 2 is 0533h = 1331, so the frequency of OUT2 is 1.1931817 MHz/1331 $\approx$ 896 Hz. Channel 2 is used to produce a beep from the built-in speaker. More details on controlling the speaker are given below.

10.4 Internal Speaker

10.4.1 The Speaker Interface

The PC has an internal speaker which is capable of generating beeps of different frequencies. You control the speaker by providing a frequency number which determines the pitch of the beep, then turning the speaker on for the duration of the beep.

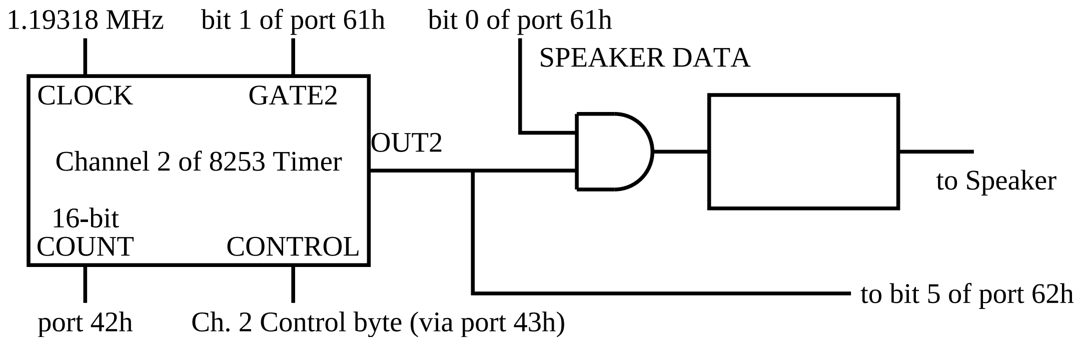
The frequency number you provide is actually a counter value. The PC uses it to determine how long to wait between sending pulses to the speaker. A smaller frequency number will cause the pulses to be sent quicker, resulting in a higher pitch. The PC uses a base rate of 1,193,180 Hz (this frequency is generated by an oscillator chip). The frequency number tells the PC how many of these cycles to wait before sending another pulse. Thus, you can calculate the frequency number required to generate a specific frequency by the following formula:

frequency number = 1,193,180 / frequency

The frequency number is a word value, so it can take values between 0 and 65,535 inclusive. This means you can generate any frequency between 18.21 Hz (frequency number = 65,535) and 1,193,180 Hz (frequency number = 1).

Figure 10-1 is a diagram of the hardware for driving the built-in speaker. OUT2 is the output of Channel 2 of the 8253-5 timer chip, GATE2 (= bit 1 of port 61h) is the enable/trigger control for the Channel 2 counter, and SPEAKER DATA (= bit 0 of port 61h) is a line that may be used independently to modulate the output waveform, e.g., to control the speaker volume.

Figure 10-1. Built-in speaker hardware



The count and load modes selected for Channel 2 during BIOS initialization are probably the best to use for tone production. In Mode 3, the counter output is a continuous symmetrical square wave as long as the GATE line of the channel is enabled; the other modes either produce outputs that are too asymmetrical or require retriggering for each count cycle.

The frequency count is loaded into the Channel 2 COUNT register at I/O port 42h. GATE2 (bit 1 of I/O port 61h) must be set to 1 to get an output on OUT2; the SPEAKER DATA line (bit 0 of I/O port 61h) must also be set to 1 to produce a tone. Note that the remaining bits of port 61h must not be changed since they control RAM enable, keyboard clock, etc. To silence the speaker, bits 1 or 0 of port 61h are set to 0 (without disturbing the remaining bits of port 61h).

10.4.2 Generating Sounds

You can communicate with the speaker controller using IN and OUT instructions. The following lists the steps in generating a beep:

1. Send the value 182 to port 43h. This sets up the speaker.
2. Send the frequency number to port 42h. Since this is an 8-bit port, you must use two `OUT` instructions to do this. Send the least significant byte first, then the most significant byte.

3. To start the beep, bits 1 and 0 of port 61h must be set to 1. Since the other bits of port 61h have other uses, they must not be modified. Therefore, you must use an `IN` instruction first to get the value from the port, then do an OR to set the two bits, then use an `OUT` instruction to send the new value to the port.
4. Pause for the duration of the beep.
5. Turn off the beep by resetting bits 1 and 0 of port 61h to 0. Remember that since the other bits of this port must not be modified, you must read the value, set just bits 1 and 0 to 0, then output the new value.

The following code fragment generates a beep with a frequency of 261.63 Hz (middle C on a piano keyboard) and a duration of approximately one second:

```
        mov     al, 182         ; Prepare the speaker for the
        out     43h, al         ;  note.
        mov     ax, 4560        ; Frequency number (in decimal)
                                ;  for middle C.
        out     42h, al         ; Output low byte.
        mov     al, ah          ; Output high byte.
        out     42h, al
        in      al, 61h         ; Turn on note (get value from
                                ;  port 61h).
        or      al, 00000011b   ; Set bits 1 and 0.
        out     61h, al         ; Send new value.
        mov     bx, 25          ; Pause for duration of note.
.pause1:
        mov     cx, 65535
.pause2:
        dec     cx
        jne     .pause2
        dec     bx
        jne     .pause1
        in      al, 61h         ; Turn off note (get value from
                                ;  port 61h).
        and     al, 11111100b   ; Reset bits 1 and 0.
        out     61h, al         ; Send new value.
```

Another way to control the length of beeps is to use the timer interrupt. This gives you better control over the duration of the note and it also allows your program to perform other tasks while the note is playing.

10.4.3 Frequency Table

The following table lists frequencies and frequency numbers for the three octaves around middle C on a piano keyboard.

Note	Frequency	Frequency #
C	130.81	9121
C#	138.59	8609
D	146.83	8126

Note	**Frequency**	**Frequency #**
D#	155.56	7670
E	164.81	7239
F	174.61	6833
F#	185.00	6449
G	196.00	6087
G#	207.65	5746
A	220.00	5423
A#	233.08	5119
B	246.94	4831
Middle C	261.63	4560
C#	277.18	4304
D	293.66	4063
D#	311.13	3834
E	329.63	3619
F	349.23	3416
F#	369.99	3224
G	391.00	3043
G#	415.30	2873
A	440.00	2711
A#	466.16	2559
B	493.88	2415
C	523.25	2280
C#	554.37	2152
D	587.33	2031
D#	622.25	1917
E	659.26	1809
F	698.46	1715
F#	739.99	1612
G	783.99	1521
G#	830.61	1436
A	880.00	1355
A#	923.33	1292
B	987.77	1207
C	1046.50	1140

Chapter 11
Graphics

This chapter describes the various common graphics modes and how to use them in assembly language, as well as how to implement various algorithms for drawing lines and other images on the graphics screen.

11.1 Text Mode

11.1.1 Displaying Text on the Screen

INT 10h Functions

One way to display text on the screen quickly is to use the BIOS interrupt 10h functions. See the INT 10h function list elsewhere for a complete description of these functions. A brief list of the more useful functions is given here:

Function 0	Set Video Mode
Function 2	Set Cursor Position
Function 6	Scroll Active Page Up
Function 9	Write Attribute/character at Current Cursor Position

Library Routines

In addition to these interrupts, the following subroutines for displaying characters are available through the LIB291 library file:

`dspout`

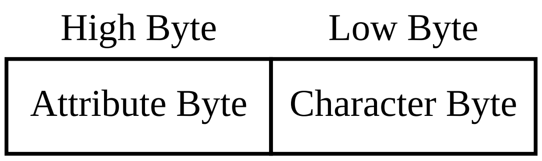
This subroutine prints the character found in `DL` to the screen, at the current cursor position. The character must be in ASCII. The contents of `DL` are preserved upon return. The cursor is advanced after the write.

`dspmsg`

This subroutine will print to the screen, starting at the current cursor position, a string of ASCII characters. `DX` must contain an offset from `DS` to the location of the first character to be printed. In addition, the string must end with an ASCII dollar sign ($), and hence may not contain one. (The dollar sign is the text delimiter). The contents of `DX` are preserved upon return.

Direct Writes to Video Memory

A third method for displaying text involves accessing the video memory directly. The contents of the text screen are stored in memory beginning at address B8000h. Each character on the text display comprises one word in memory, the meaning of which is shown in the following figure:



The attribute byte gives information about the color of that particular character on the screen (discussed in the following section). The character byte is simply the 8-bit ASCII code for the character at that position.

The screen is divided into rows and columns, with the upper-left character position usually referred to as row 0 and column 0. The first row of the screen is stored first beginning with the first column (row 0, column 0), then the next row, and so on. Thus, an 80x25 text screen requires

$$80 \text{ columns} * 25 \text{ rows} * 2 \text{ bytes per row} = 4000 \text{ bytes}$$

to store the entire screen.

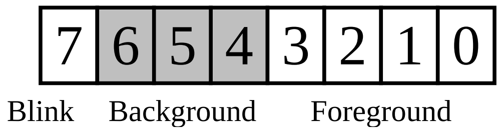
The following code fragment illustrates an example of how to access the video memory. This code fragment will change the top row of characters to yellow “A”s on a blue background. Note the use of segment register `ES` to access the memory at absolute address B8000h.

```
        mov     bx, 0           ; Begin pointer at row 0 and column 0
        mov     ax, 0B800h      ; Set up ES to point to video segment
        mov     es, ax
        mov     cx, 80          ; Counter for number of columns in top row
.lp:
        mov     [es:bx], 1E41h  ; Yellow ’A’ on blue background
        add     bx, 2           ; Update pointer to next character position
        loop    .lp             ; Do 80 characters
```

11.1.2 Attributes

The Attribute Byte

The attribute byte specifies the colors for the character and its background. The table below describes the format for the attribute byte. You can write a character and its attribute to the display using Interrupt 10h, Function 09h (see Section 11.3.8), or by writing directly to video memory. The structure of the attribute is shown in the following figure:



Blink	1 = blink on, 0 = off.
Background	Selects a color from the first eight colors in the palette.
Foreground	Selects one of the 16 colors in the palette.

Color Palette

0 — Black	8 — Dark Gray
1 — Blue	9 — Light Blue
2 — Green	10 — Light Green
3 — Cyan	11 — Light Cyan
4 — Red	12 — Light Red
5 — Magenta	13 — Light Magenta
6 — Brown	14 — Yellow
7 — Light Gray	15 — White

After power up the attributes for the entire screen are set to (00000111b), the attribute for a light gray character and a black background, with no blinking.

11.2 VGA Mode 13h Graphics

11.2.1 Overview

Mode 13h is a standard VGA graphics mode. While it may seem somewhat limited in resolution and colors when compared to SVGA modes, it is supported on nearly every computer available today. There are 320 pixels over and 200 pixels down the screen, each of which may have one of 256 colors. Mode 13h is unique from other higher-resolution, but only 16-color, VGA graphics modes because the memory for the video screen is arranged linearly, where one byte corresponds to one pixel on the screen. As a result, Mode 13h is very easy to program in since it's not necessary to deal with bit planes.

The first step to making a Mode 13h program is to call a BIOS interrupt to enter it. Do this by:

```
mov     ax, 0013h
int     10h
```

To return back to text at the end of the program, use:

```
        mov     ax, 0003h
        int     10h
```

Once in Mode 13h, each pixel can be set to any color from a palette of 256 colors. There are two methods to do this by—using BIOS interrupts or direct writes. Direct writes are the best choice here due to the simplicity of Mode 13h (the bit planes used in other VGA modes can make BIOS interrupts the best choice). Since each pixel directly corresponds to one byte, just set the byte to the color (0 - 255). Use the following formula to determine the address and move a byte into it:

$$\text{offset} = 320 * Y * X$$

The following code fragment illustrates how to set a pixel:

```
; Before drawing graphics
        mov     ax, 0A000h      ; Set ES to graphics screen segment
        mov     es, ax
                |
            (set X and Y and DL to the color)
                |
        imul    di, [Y], 320    ; Multiply Y by 320 and store in DI
        add     di, [X]         ; Add the X coordinate
        mov     [es:di], dl     ; Set the pixel to the color specified
                                ;  by DL
```

This method can be much faster than BIOS, because for most things, only a few IMUL instructions should be needed and the majority of pixels can be set by moving relative to the original address. For example, to move left or right, simply subtract or add 1 from the address, and to move up and down, subtract or add 320, respectively. Use this method along with string operations to set entire ranges, draw straight horizontal and vertical lines, and draw boxes. For more information about string operations, refer to Section 5.5.

11.2.2 Using Extra Segments

One of the most aggravating thing about creating animated images is the flicker and delays seen by the user while the image is being drawn. Because of Mode 13h's memory scheme and usage, it is possible to draw the graphics on another segment in memory instead of the graphics screen, and when an individual frame is finished being created, to copy that segment directly onto the VGA screen memory. The easiest and fastest way to do this is to use string operations.

To create a blank segment in the program that graphics instructions can be "redirected" to, do the following:

```
SEGMENT ScratchSeg
ScratchPad      resb 64000
```

This should be included at the beginning of the program along with other segment definitions. Then, instead of setting `ES` to point to the VGA memory segment, have `ES` point to the ScratchSeg segment when plotting pixels.

Once an entire frame has been drawn on the scratch segment, it is very simple to copy it to the VGA screen:

```
mov     ax, ScratchSeg  ; from scratch segment
mov     ds, ax
mov     ax, 0A000h      ; to graphics screen
mov     es, ax
cld                     ; set direction flag forward
mov     esi, ScratchPad ; set the source offset
xor     edi, edi        ; dest. offset will always be 0
mov     ecx, 320*200/4  ; 320*200, copying doublewords
rep movsd
```

This will issue a `MOV` command 16,000 times, each time moving a DWORD (4 pixels) from the ScratchSeg segment to the VGA screen.

> **Note:** Each segment declared in this fashion occupies 64k = 64000 bytes of memory. Real mode only allows 640k of total memory, so only declare at most 2 or 3 scratch segments.

11.3 Interrupt 10h Video Reference

To use one of these funtions, first place the function number in `AH`, then set the other input registers, then call the function with INT 10h. Registers not used by the function as inputs or outputs are not affected.

> **Note:** The text functions work in all modes, including graphics modes.

11.3.1 Function 00h: Set Video Mode

Sets the video mode. This function will clear the screen unless bit 7 of the `AL` register is set.

Inputs

`AH` = 00h
`AL` = Video Mode (add 128 to not clear the screen). This is an very incomplete list of modes; see another interrupt reference for full details of all modes supported.
 01h: 40x25 Text, 16 colors, 8 pages
 03h: 80x25 Text, 16 colors, 8 pages
 13h: 320x200 Graphics, 256 colors, 1 page

Outputs

(None)

11.3.2 Function 01h: Define Cursor Appearance

Sets the starting and ending lines of the screen cursor, and can also be used to make the cursor invisible. In some modes, the characters are not exactly 8 lines high in these cases the graphics hardware will adapt the input values to the current character size.

Inputs

`AH` = 01h
`CH` = Starting line of cursor (0-7). Add 20h to make the cursor invisible.
`CL` = Ending line of cursor (0-7).

Outputs

(None)

11.3.3 Function 02h: Set Cursor Position

Moves the cursor to the specified position on the screen. The video hardware maintains a separate cursor for each display page, and the cursor will move only if the page number given in `BH` is the current display page. Giving a position that is off the screen will cause the cursor to disappear from the screen.

Inputs

`AH` = 02h
`BH` = Display page (valid only in text modes—use 00h for graphics modes).
`DH` = Row (00h is top row).
`DL` = Column (00h is leftmost column).

Outputs

(None)

11.3.4 Function 05h: Set Current Display Page

Sets the display page which will appear on the screen.

Inputs

`AH` = 05h
`AL` = Display page (the range of valid values depends on the current video mode—see *Function 00h*)

Outputs

(None)

11.3.5 Function 06h: Scroll Text Lines Up

Scrolls part or all of the current display page up by one or more text lines. This function can also be used to clear part or all of the screen.

Inputs

`AH` = 06h
`AL` = Number of lines to be scrolled up (`AL` = 00h will clear the window).
`BH` = Color attribute for blank lines. In text mode, this corresponds to the attribute byte. In VGA graphics modes, this is the color number to which all the pixels in the blank lines will be set.
`CH` = Top row of window to be scrolled up.
`CL` = Leftmost column of window.
`DH` = Bottom row of window.
`DL` = Rightmost column of window.

Outputs

(None)

11.3.6 Function 07h: Scroll Text Lines Down

Inputs

`AH` = 07h
`AL` = Number of lines to be scrolled down (`AL` = 00h will clear the window).
`BH`, `CH`, `CL`, `DH`, `DL` : Same as *Function 06h*.

Outputs

(None)

11.3.7 Function 08h: Read Character and Attribute at Current Cursor Position

Inputs

`AH` = 08h
`BH` = Display page.

Outputs

`AL` = ASCII code of character. If the current video mode is a graphics mode and no match is found, 00h is returned.
`AH` = Color attribute.

11.3.8 Function 09h: Write Character and Attribute at Current Cursor Position

Writes the specified character with the specified color attribute at the current cursor position in the specified display page. The cursor is NOT moved to the next screen position. Special control codes are not recognized, and are printed as normal ASCII characters (e.g., writing a carriage return will not cause the cursor to move to the beginning of the line).

Inputs

`AH` = 09h
`AL` = ASCII code.
`CX` = Repeat factor. The character will be written this many times. In graphics mode, all the characters must fit on the same screen line.
`BH` = Display page.
`BL` = Color attribute. In text mode, this corresponds to the attribute byte. In graphics mode, this is the foreground color (the background color will be 0). In graphics mode, if bit 7 is set, the character will be XORed with the current bitmap.

Outputs

(None)

11.3.9 Function 0Ah: Write Character Only at Current Cursor Position

This function is identical to *Function 09h* except that in text modes, the attribute byte currently in video memory is not modified (`BL` is ignored). In graphics modes, this function is exactly identical to Function 09h. See the comments for Function 09h.

Inputs

`AH` = 0Ah
`AL`, `CX`, `BH`, `BL` : Same as *Function 09h*

Outputs

(None)

11.3.10 Function 0Ch: Write Graphics Pixel

Sets the color number of the specified pixel in graphics mode. Valid in all graphics modes.

Inputs

`AH` = 0Ch
`BH` = Display page.
`DX` = Screen line (0 is top).
`CX` = Screen column (0 is leftmost).
`AL` = Color number.

Outputs

(None)

11.3.11 Function 0Dh: Read Graphics Pixel

Returns the color number of the specified pixel in graphics mode. Valid in all graphics modes.

Inputs

`AH` = 0Dh
`BH` = Display page.
`DX` = Screen line (0 is top).
`CX` = Screen column (0 is leftmost).

Outputs

`AL` = Color number.

11.3.12 Function 0Eh: Write Character

Writes the specified character to the current cursor position in the current display page. In text modes, the attribute byte is not modified. The cursor is moved to the next screen position, and the screen is scrolled up if necessary. Special ASCII characters, like carriage return and backspace, are interpreted as control codes and will modify the cursor position accordingly.

Inputs

`AH` = 0Eh
`AL` = ASCII code.
`BL` = Foreground color (valid only in graphics modes).

Outputs

(None)

11.3.13 Function 10h: VGA Color Functions

Color in VGA modes is quite complicated because it is based on a palette of colors. There are 256 DAC (Digital-to-Analog Converter) registers. Each of these 18-bit registers contains an RGB (Red, Green, Blue) color value. These registers define the basic color palette used in 256-color modes. register. Each palette register corresponds to one color number in video memory. For example, if the bitmap contains color number 17 for a certain pixel, then DAC register #17 determines what color will be generated for the pixel.

It is possible to set the DAC registers by using either Interrupt 10h functions (listed below) or by directly accessing the VGA card using port I/O. See Example 11-1 for code that sets the palette using port I/O. For more information on VGA graphics, see Section 11.2.

Subfunction 00h: Set Border (Overscan) Color

Sets the color of the border around the screen.

Inputs

`AX` = 1001h
`BH` = DAC register number (0-255).

Outputs

(None)

Subfunction 10h: Set Individual DAC Register

Sets the RGB (Red, Green, Blue) values for one of the DAC (Digital-to-Analog Converter) registers.

Inputs

`AX` = 1010h
`BX` = DAC register number (0-255).
`DH` = Red value (0-63).

`CH` = Green value (0-63).
`CL` = Blue value (0-63).

Outputs

(None)

11.4 PCX Graphic File Format

The PCX format is a relatively simple format that provides a minimum of compression using Run Length Encoding (RLE). RLE means that the file can be read from start to finish in one pass and encoded or decoded without any holistic information (i.e., in order to figure out what the next encoded byte is, you only have to know what preceded it, not anything after it.) The PCX format is especially useful for 320x200x256 VGA mode 13h (where each pixel is stored as a byte). The PCX format was originally used by PC Paintbrush.

11.4.1 RLE Encoding

The following discussion assumes 320x200x256 VGA mode 13h, as described in Section 11.2

Two types of bytes are stored in the data image portion of a PCX file. One type is a *length*, and the other is *color*. A length byte is specified by the two upper bits being set. This limits the length specified by a length byte to 64. The other type is a color byte, and specifies a value for the byte from the palette table (the palette holds the actual RGB values of the color, and the color byte is an index into this table). This is the same method used in mode 13h. The first byte from the data is read. If the two upper bits are set, then it is a length byte, and the next byte is the color which will be replicated as many times as stated by the length byte, from left to right on the screen, ending at the end of a line (see BYTES_PER_LINE below). If the two bits are not set, then it is a color byte, and it goes onto the screen in the next location (left to right) as is.

Note: Any color greater than or equal to 192 cannot be stored as a single color byte, and must be a given a length first. For instance, if you have a single byte of color 192, then it must be represented by two bytes of 193 (length byte of 1) and 192 (color byte 192).

11.4.2 PCX File Format

The PCX file itself contains two parts—the first part is called the *header*, which contains information about the image; the second part is the *image data*, which contains actual image data and color information. Rather than explain each field of the header in detail, a structure is shown below which gives a brief glance at the purpose of each field.

```
STRUC PCX_Header
.Manufacturer   resb    1       ; should always be 0Ah
.Version        resb    1       ; ❶
.Encoding       resb    1       ; should always be 01h
.BitsPerPixel   resb    1       ; ❷
.XMin           resw    1       ; image width = XMax-XMin
.YMin           resw    1       ; image height = YMax-YMin
.XMax           resw    1
.YMax           resw    1
.VertDPI        resw    1       ; ❸
.Palette        resb    48      ; ❹
.Reserved       resb    1
.ColorPlanes    resb    1       ; ❺
.BytesPerLine   resw    1       ; ❻
.PaletteType    resw    1
.HScrSize       resw    1       ; only supported by
.VScrSize       resw    1       ; PC Paintbrush IV or higher
.Filler         resb    56
ENDSTRUC
```

❶ PCX version number. It corresponds to the following PC Paintbrush versions and/or features:
0 — Version 2.5
2 — Version 2.8, palette included
3 — Version 2.8, use default palette
5 — Version 3.0 or better

❷ Number of bits of color used for each pixel.
1 — Monochrome
4 — 16 colors
8 — 256 colors
24 — 16.7 million colors

❸ Vertical resolution, in DPI (dots per inch).

❹ If 16 colors or less, contains the color palette.

❺ Number of color planes:
4 — 16 colors
3 — 24 bit color (16.7 million colors)

❻ Number of bytes per line (the width of the image in bytes). For 320x200, 256-color images, this is 320 bytes per line.

In a PCX file containing 16 colors of less, the palette is contained in the .Palette section of the header. In a PCX file containing 256 colors, the palette is at the end of the file, and takes up the last 768 bytes (256 * 3 bytes per color RGB). If the last 768 bytes is a palette, there is a padding byte preceding it in the file (whose value is 12).

Example 11-1. Displaying a PCX File

```
EXTERN  kbdin, dosxit           ; LIB291 functions

SEGMENT ScratchSeg
ScratchPad      resb 65535

SEGMENT stkseg STACK
```

```
        resb    64*8
stacktop:
        resb    0

SEGMENT code

PCX1    db      ’my_pcx1.pcx’, 0        ; Filenames
PCX2    db      ’my_pcx2.pcx’, 0        ; (Must end with 0 byte)

..start:
        mov     ax, cs          ; Set up data and stack segments
        mov     ds, ax
        mov     ax, stkseg
        mov     ss, ax
        mov     sp, stacktop

MAIN:
        ; Sets up mode 13h and clears screen
        mov     ax, 0013h
        int     10h

        mov     dx, pcx1        ; Filename to display
        call    ShowPCX         ; Display PCX file to screen

        ; Wait for keypress
        call    kbdin

        ; Go back to text mode
        mov     ax, 0003h
        int     10h

        ; Return to DOS
        call    dosxit

;---------------------------------------------------
; ShowPCX procedure by Brandon Long,
;   modified by Eric Meidel and Nathan Jachimiec,
;   converted to NASM, cleaned up, and better commented by Peter Johnson
; Inputs: DX has the offset of PCX filename to show.
; Output: PCX file displayed (all registers unchanged)
; Notes:  Assumes PCX file is 320x200x256.
;         Uses ScratchSeg for temporary storage.
;         The PCX file must be in the same directory as this executable.
;---------------------------------------------------
ShowPCX
        push    ax              ; Save registers
        push    bx
        push    cx
        push    si
        push    di
        push    es

        mov     ax, 3D00h
        int     21h             ; Open file
        jc      .error          ; Exit if open failed

        mov     bx, ax          ; File handle
        mov     cx, 65535       ; Number of bytes to read
        mov     ax, ScratchSeg  ; DS:DX -> buffer for data
        mov     ds, ax
        mov     dx, ScratchPad
        mov     si, dx
        mov     ah, 3Fh
        int     21h             ; Read from file

        mov     ax, 0A000h      ; Start writing to upper-left corner
        mov     es, ax          ; of graphics display
        xor     di, di

        add     si, 128         ; Skip header information

        xor     ch, ch          ; Clear high part of CX for string copies

.nextbyte:
        mov     cl, [si]        ; Get next byte
        cmp     cl, 0C0h        ; Is it a length byte?
        jb      .normal         ;  No, just copy it
        and     cl, 3Fh         ; Strip upper two bits from length byte
        inc     si              ; Advance to next byte - color byte
        lodsb                   ; Get color byte into AL from [SI]
        rep stosb               ; Store to [ES:DI] and inc DI, CX times
        jmp     short .tst

.normal:
        movsb                   ; Copy color value from [SI] to [ES:DI]

.tst:
        cmp     di, 320*200     ; End of file? (written 320x200 bytes)
        jb      .nextbyte

        mov     cl, [si]
        cmp     cl, 0Ch         ; Palette available?
        jne     .close

        ; Set palette using port I/O
        mov     dx, 3C8h
        mov     al, 0
        out     dx, al
        inc     dx              ; Port 3C9h
        mov     cx, 256*3       ; Copy 256 entries, 3 bytes (RGB) apiece
        inc     si              ; Skip past padding byte

.palette:
        lodsb
        shr     al, 1           ; PCX stores color values as 0-255
        shr     al, 1           ;  but VGA DAC is only 0-63
        out     dx, al
        dec     cx
        jnz     .palette

.close:
        mov     ah, 3Eh
        int     21h             ; Close file

.error:
        pop     es              ; Restore registers
        pop     di
        pop     si
        pop     cx
        pop     bx
        pop     ax
        ret
```

Chapter 12
Serial Communication

12.1 Serial Data Communication

Within a processor character data are generally transferred, stored, or processed as bytes, i.e., as 8 bits in parallel. Parallel transmission of character data to a peripheral device requires a bundle of lines that includes one line for each bit of the character, a strobe or clock line, and possibly some other control and status signals. For long distances, parallel transmission becomes unattractive due to the increased cost of the multiple wires as well as the increased difficulty of avoiding crosstalk and skew between the bundled signals. The alternative, serial transmission, requires only one line over which the bits that represent a character are sent one after the other. Serial transmission is also used for data communication via the commercial voice-grade telephone network, where the serial bit stream is used to modulate an analog tone frequency signal appropriate for transmission over the telephone network.

Data communication is usually *bidirectional*, alternately in either direction (*half-duplex*) or simultaneously in both directions (*full-duplex*). Half-duplex operation permits the use of the same line (and of the same telephone line signals) for transmission as well as reception; the special character EOT (End-of-Transmission) is used to trigger the turn-around. Full-duplex operations requires separate transmit and receive lines (and two distinct sets of telephone line signals).

Figure 12-1. Typical Connection between and another Device via the Telephone Network

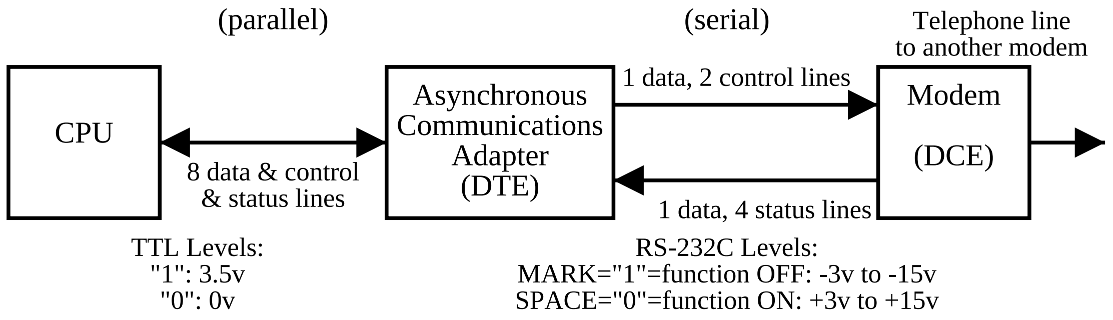

(parallel)
(serial)
Telephone line to another modem
CPU
8 data & control & status lines
Asynchronous Communications Adapter (DTE)
1 data, 2 control lines
1 data, 4 status lines
Modem (DCE)
TTL Levels: "1": 3.5v "0": 0v
RS-232C Levels: MARK="1"=function OFF: -3v to -15v SPACE="0"=function ON: +3v to +15v

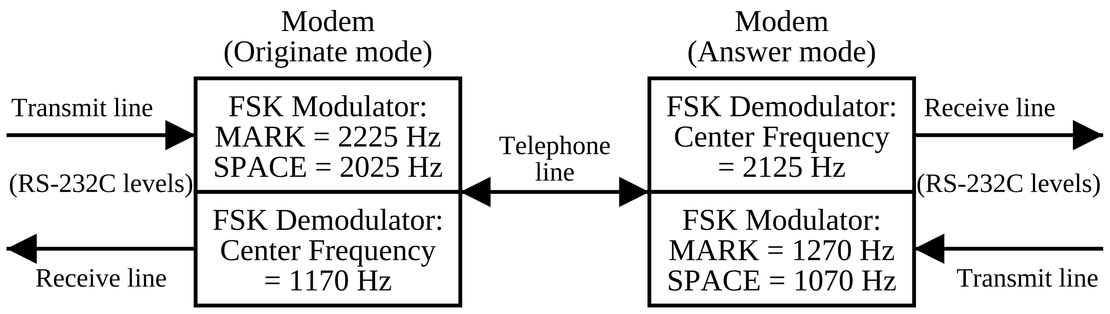
Figure 12-1 shows one side of a full-duplex connection between a CPU and another device via the telephone network. The Asynchronous Communication Adapter, located on a card that plugs into an I/O slot of the PC, performs the parallel-to-serial conversion of transmitted data and the serial-to-parallel conversion of received data; the “Modem” (Modulator/Demodulator), also called “Data Set,” performs the conversion between discrete voltage levels and analog tone signal representations, and vice versa. The interface between the Asynchronous Communications Adapter and the modem should follow the standards set by the Electronic Industries Association (EIA), e.g., EIA RS-232C, or the similar international standards set by the Comite Consultatif International Telephonique et Telegraphique (CCITT), e.g., CCITT V.24.

12.2 Modems, Bauds, and Bits per Second

A voice-grade telephone line has a useful frequency range of 300-3000 Hz but modems typically use tones restricted to the range 300-2400 Hz, primarily to avoid a 2600 Hz signaling tone that causes call disconnect. Various modulation schemes are used to convert the representation of information from the RS-232C discrete voltage levels to amplitude, phase, or frequency shift keyed analog signals, and vice versa for demodulation; each analog symbol may represent one or more bits. The number of symbols per second sent over a communication line called BAUD (after J.M.E. Baudot, 1845-1903, a French inventor who studied telegraph codes.)

Simple modulation techniques carry one bit per symbol. For example, in a "type 103" 300 baud modem each bit is translated to one of two tone frequencies using FSK (Frequency Shift Keying). Two sets of frequencies are used to provide full-duplex operation; each set is used for either transmit or receive, depending on whether the modem originated or answered the call. Details are given in Figure 12-2.

Figure 12-2. 300 Baud Asynchronous Full-Duplex U.S. Frequency Assignments



Complex modulation schemes such as CCITT V.22bis carry 4 bits per symbol, using a combination of amplitude and phase shift keying, to achieve a data transfer rate of 2400 bits/sec. This rate is usually called "2400 baud" in reference to the 2400 levels/sec on the RS-232C line; the symbol rate on the telephone line is 600 baud.

12.3 Interface Standards

One of the earliest standards for interfacing digital devices and modems is the EIA RS-232 standard, called "Interface between Data Terminal Equipment and Data Circuit- Terminating Equipment Employing Serial Binary Interface." RS-232C is the latest version (CCITT V.24 is virtually identical). It lists the electrical and mechanical interface characteristics, describes the function of signals, and lists subsets of signals for specific interface types. A computer, printer, etc., is Data Terminal Equipment (DTE), a modem or data set is Data Circuit-Terminating (or Communication) Equipment (DCE). As its name indicates the standard is intended for DTE-DCE connections.

Table 12-1 shows the most commonly used RS-232C signals and their pin numbers on the standard 25-pin D-shell connector. Signal names are given with DTE as reference. A male D-shell connector is used on DTE, a female one on DCE; a straight female-male cable connects DTE to DCE. The standard defines a total of 21 signals, including "secondary" signals and signals that allow data rate selection. Most applications use a subset of these signals; some use a 9-pin D-shell connector instead of the 25-pin connector.

Table 12-2 shows the RS-232C electrical specifications. The voltage levels specified for the RS-232C driver outputs provide zero crossing and better noise immunity than the levels used in standard TTL or MOS technologies but require either power supply voltages (usually +12v/-12v) that are not available, or needed, in the rest of the DTE circuitry, or the use of chips that derive a negative supply voltage on-chip from the standard +5v supply, e.g., the MAXIM MAX232, a 3 driver/3 receiver chip.

Table 12-1. Most Commonly Used RS-232C Signals

25-Pin # [9-Pin]	Signal Name	Source
1	Protective (Earth) Ground	
7 [5]	Signal Ground	
2 [3]	Transmitted Data (TxD)	DTE
4 [7]	Request to Send (RTS)	
20 [4]	Data Terminal Ready (DTR)	
3 [2]	Received Data (RxD)	DCE
5 [8]	Clear to Send (CTS)	
6 [6]	Data Set Ready (DSR)	
22 [9]	Ring Indicator (RI)	
8 [1]	Received Line Signal Detect / Carrier Detect (RLSD/CD)	

Table 12-2. RS-232C Electrical Specifications

Mode of operation	single-ended (unbalanced)
Cable length	50 feet max.
Data rate	20 kb/s max.
Driver output	+5v to +15v for "0", -5v to -15v for "1"
Voltage applied to driver output	$\pm$25v max.
Driver load	3 kΩ to 7 kΩ
Output slew rate	30 v/μs max.
Receiver input range	$\pm$15v
Receiver sensitivity	$\pm$3v
Receiver input resistance	3 kΩ to 7 kΩ

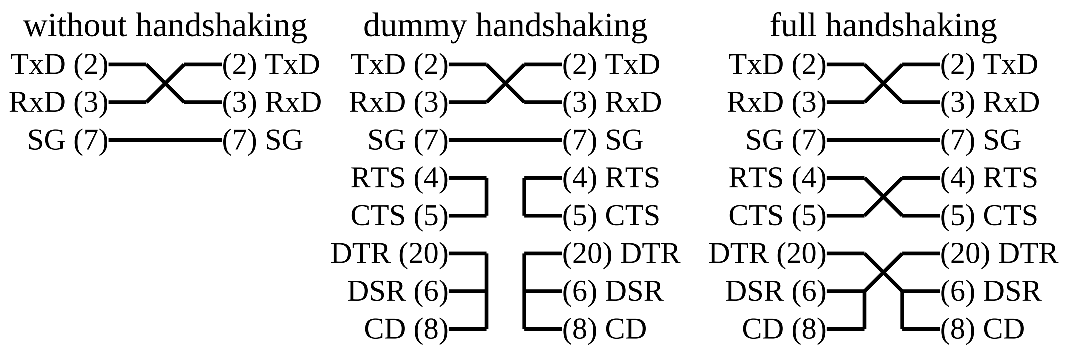
Neither the 50' maximum cable length nor the maximum data rate of the RS-232C standard should be a serious limitation in the DTE-DCE application for which the standard is written the DTE and the modem are usually located near each other, and reliable communication over the switched telephone network at more than about 2400 bit/sec is very difficult at present. Surprisingly, the RS-232C specifications do not recognize the usefulness of such a standard in applications other than local DTE-DCE connections, nor do they specify any distance vs. rate tradeoffs; in practice, twisted pair cable can be used successfully up to 3000’ at rates up to 1200 bits/sec, and up to 250’ at up to 9600 bits/sec.

The major difficulty with using RS-232C over long distances is that the Signal Ground is usually connected to earth at both ends so that ground current through the cable causes offsets in the voltages sensed. Newer standards for single-ended systems, e.g. EIA RS-423A, specify a common return path for all signals and ground the return path only at the transmitter; this standard also relates maximum data rate and maximum distance: 100 kbit/sec at 30’, 10 kbit/sec at 300’, or 1 kbit/sec at 4000’. Even higher data rates over longer distances are possible with systems using differential signal transmission: e.g., EIA RS-422A specifies 10 Mbit/sec at 40’, 1 Mbit/sec at 400’, or 100 kbit/sec at 4000’.

12.4 Connections, Compatibility, and Null Modems

RS-232C interfaces are frequently used in applications for which the standard was not originally intended, specifically DTE-DTE connections. Computers, terminals, printers, plotters, and other DTEs often have serial interfaces labeled “RS-232C compatible.” Generally this means that the signals that are implemented do not violate the standard but that not all of the standard’s signals are implemented, that the device will therefore not interface properly with a modem, and is in fact designed for direct DTE-DTE connection. E.g., data can be sent from a computer to a serial printer using only the Transmitted Data and Signal Ground lines, if the software takes care of the delays needed to let the printer perform carriage return, form feed, etc.; if the printer uses the XON/XOFF protocol the Received Data line is also needed so that the printer can send the XON/XOFF characters back to the computer, and if it uses a busy/wait protocol a handshaking line (typically Data Set Ready) is needed instead. In any case, the Transmitted Data line from the computer (pin 2 of its DTE connector) must be wired to the Received Data line of the printer (pin 3 of its DTE connector), and other lines may have to be similarly crossed to imitate the use of modems. Examples of such “null-modem” cables are shown in Figure 12-3; the typical null-modem cable has female connectors at each end.

Figure 12-3. Typical Null-Modem Cables



The printer of other DTE device may deviate from the standard even further by using a female D-shell connector wired in such a way that a straight-through extension cable rather than a null-modem cable is used for the DTE-DTE connection. Also, a 9-pin connector may be used instead of the standard 25-pin connector. Furthermore, exactly which signals are used for handshaking depends on the device as well as the software used to drive it. Thus, the direct connection of two “RS-232C compatible” DTE devices may require some experimentation and preparation of a cable specific to that application. An arsenal of “breakout boxes” (preferably with bicolor LEDs showing which lines are active), null-modem boxes, 25-to-9 pin converters, and male-male and female-female “gender changers” may make this task easier.

Deviations from the standard may also occur with respect to signal levels. TTL inverters are sometimes used rather than RS-232C line drivers or receivers which may require a negative voltage supply. There are several potential problems with this approach:

- The 0v output of a TTL driver may not be recognized by an RS-232C line receiver as a valid input. Line receivers generally use hysteresis to improve noise immunity, i.e., as long as the input level stays between the lower and upper thresholds the output will maintain the previous value; these thresholds usually are adjustable via a control pin. E.g., the SN75154 Quadruple Line Receiver may operate in either the “normal” mode (control pin connected to +5v) with a -1.1v to +2.2v hysteresis, or the “fail-safe” mode (control pin open) with a +1.4v to 2.2v hysteresis. Thus, a grounded or open input is valid in the fail-safe mode, but not in the normal mode. (75154s are used in the IBM Asynchronous Communication Adapter, set for normal mode.)
- TTL circuits used as drivers may not tolerate line-to-line shorts which RS-232C drivers are designed to withstand.
- TTL circuits used as receivers may not tolerate the ±25v input voltage range which RS-232C receivers are designed to withstand.

12.5 Parallel/Serial Conversion

The serial data communication scheme used here is called asynchronous because the time between transmitted characters is not fixed, and the transmitting and receiving devices are not synchronized to the same clock, although the individual bits of each character are transmitted at a known baud rate. The line is held at the MARK level when idle; for each character, the receiver must recognize when the character starts and synchronize itself to the transmitter to read the individual bits of the character. This is accomplished by sending each character in a “frame” consisting of a START BIT (a SPACE for one bit period), the bits of the character (least significant bit first), and a least one STOP BIT (a MARK for a least one bit period).

Characters are represented by from 5 to 8 information bits, with 8 bits most common. The 8 bits may represent the extended-ASCII codes, or the standard ASCII codes followed by a parity bit to allow the detection of single-bit errors. The STOP bit is essentially an enforced IDLE before the next START bit; it gives the receiver time to complete processing of the received character (e.g., compute and check the parity of the received character) and allows for slight differences between the transmit and receive clocks.

Common baud rates for 8-bit (or 7-bit and parity) characters with 1 start and 1 stop bit are 300, 1200, 2400, 4800, 9600, and 19200 baud, corresponding to 30, 120, etc. characters per second (since the start and stop bits must be included in the bit count); old-fashioned mechanical teletypes ran at 75 baud using characters with 1 start, 5 (Baudot code) data, and 1.5 stop bits, or at 110 baud using characters with 1 start, 8 data, and 2 stop bits, i.e., at 10 characters/sec.

Receivers typically use an internal clock that is 16 times the baud rate. Reception of a character starts when the receiver detects a 1-to-0 (IDLE-to-START) transition. The receiver then waits for 8 block periods (.5 bit period) and tests the line again: if the signal is now 1 this is considered a false start and the receiver goes back to looking for another 1-to-0 transition; if the signal is still 0 it was a valid start and the remaining bits will be sensed every 16 block periods (1 bit period) thereafter. This approach synchronizes the receiver to the transmitter to within 1/16 of a bit period at the beginning of each character, and tends to place the times at which the signal is sampled at the middle of each bit period, thus maximizing the tolerance for differences between the receiver's and transmitter's internal clocks. The receiver will indicate a "framing error" if the signal is not at the 1 level at the middle of the stop bit period. The next START transition may occur immediately thereafter.

12.6 Serial Data Communication using BIOS calls

The PC supports up to four Communications Adapters, COM1-COM4, identified by `DX`=0, 1, 2, or 3 respectively. BIOS interrupt 14h calls with `DX`=0, 1, 2, or 3 and `AH`=0, 1, 2, or 3 may be used to initialize the adapter to the character format and the baud rate given in `AL`, to transmit the character in `AL`, to put the received character into `AL`, or to read the modem status into `AL`. In all cases, the port status is returned in `AH`. BIOS interrupt 14h calls with `AH`=4 or 5 provide extended initialization and modem port control.

BIOS call 14h with `AH`=0 is used to select standard character formats and baud rates by setting `AL` to BBBPPSLL, where:

BBB = 000, 001, 010, 011, 100, 101, 110, 111 for 110, 150, 300, 600, 1200, 2400, 4800, 9600 baud

PP = x0 for no parity, 01 for odd parity, and 11 for even parity

S = 0 for 1 stop bit, 1 for 2 (1.5 if 5 info. bits) stop bits

LL = 00 for 5, 01 for 6, 10 for 7, 11 for 8 info. bits

In addition, all interrupts from the Adapter are disabled, the port status is returned in `AH` and the modem status in `AL`, according to Table 12-3 and Table 12-4 below.

BIOS call 14h with `AH`=1 waits for Transmit Holding Register Empty (THRE) and transmits the character in `AL`; it returns with the port status (Table 12-3) in `AH`, with bit 7 set if a timeout occurred.

BIOS call 14h with `AH`=2 waits for Received Data Available (RDA) and returns with the received character in `AL` and the port status (Table 12-3) in `AH`, with bit 7 set if a timeout occurred.

BIOS call 14h with `AH`=3 returns the port status (Table 12-3) in `AH` and the modem status (Table 12-4) in `AL`.

BIOS call 14h with `AH`=4 can be used as an alternative to `AH`=0 to select higher baud rates or more specific serial port parity. Various settings are in `BH`, `BL`, `CH`, and `CL`.

`BH` = 0 for no parity, 1 for odd parity, 2 for even parity, 3 for "stick" odd parity, and 4 for "stick" even parity

`BL` = 0 for 1 stop bit, 1 for 2 (1.5 if 5 info. bits) stop bits

`CH` = 0 for 5, 1 for 6, 2 for 7, 3 for 8 info. bits

`CL` = 00h to 08h for 110, 150, 300, 600, 1200, 2400, 4800, 9600, and 19200 baud if ComShare is not installed. If ComShare is installed, 00h to 0Bh map to 19200, 38400, 300, 14400, 1200, 2400, 28800, 9600, 19200, 38400, 57600, and 115200 baud.

Note that the BIOS calls with `AH`=1 or 2 are not fast enough for sustained operation at 1200 baud or more. Note also that BIOS calls cannot be used to control or sense the modem control/device handshaking signals, or to use interrupts.

Table 12-3. BIOS Serial Port Status, Returned in `AH`

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
TIMEOUT; function failed	Transmit Shift Reg. Empty (TSRE)	Transmit Holding Reg. Empty (THRE)	Break Detected	Framing Error	Parity Error	Overrun Error	Received Data Available (RDA)

Table 12-4. BIOS Modem Port Status, Returned in `AL`

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Received Line Signal Detect (RLSD)	Ring Indicator (RI)	Data Set Ready (DSR)	Clear to Send (CTS)	Change in RLSD	Trailing Edge RI	Change in DSR	Change in CTS

12.7 Serial Data Communication using IN and OUT

A more detailed knowledge of the Asynchronous Communication Adapter is needed touse it directly via OUT and IN instructions. A simplified logic diagram of the Adapter is shown in Figure 12-4.

The National Semiconductor INS8250 chip is an Asynchronous Communication Element (ACE), also called a Universal Asynchronous Receiver-Transmitter (UART), a “smart peripheral” chip that can be programmed to perform full-duplex conversion of character data between parallel and serial formats at various baud rates, for different character formats, including the insertion and removal of start, stop, and parity bits, control of modem functions, and monitoring of modem status signals. Machines today use a different UART chip than the INS8250, but it is compatible with the 8250, so the following discussions are still valid.

In the PC, I/O ports 3F8h to 3FEh and interrupt request line IRQ4 are assigned to the primary (COM1) adapter, I/O ports 2F8h to 2FEh and interrupt request line IRQ3 to the secondary (COM2) adapter; a jumper on the Adapter card is used to configure it as COM1 or COM2. COM3 and COM4 share IRQ4 and IRQ3 with COM1 and COM2, but use different I/O ports.

The internal registers of the 8250 ACE that are accessible via I/O port addresses are shown in Table 12-5. All registers are 8 bits wide. The TRANSMIT HOLDING and RECEIVE BUFFER Registers are padded with 0s on the left for characters with fewer than 8 information bits.

Figure 12-4. Simplified logic diagram of the Asynchronous Communications Adapter

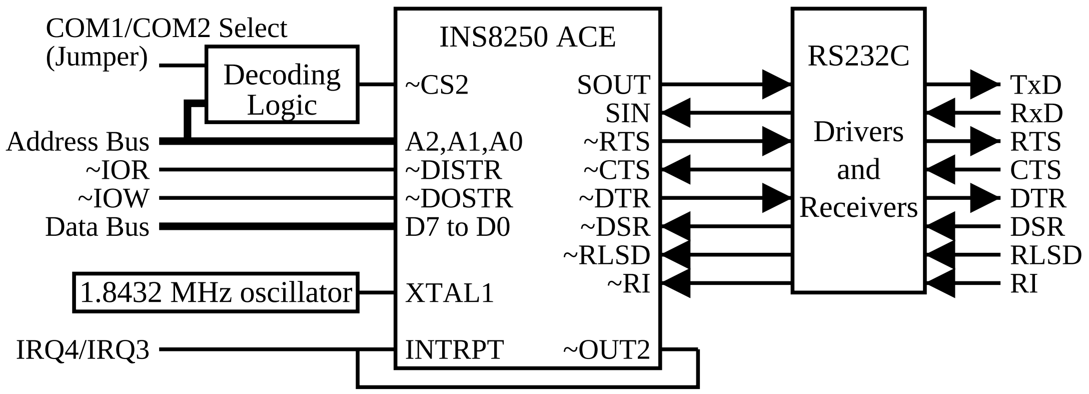


Table 12-5. Accessible registers in the IN8250 ACE

Register	**Address**
Divisor Latch (Low Byte)	ComBase+0 and DLAB=1
Divisor Latch (High Byte)	ComBase+1 and DLAB=1
Transmit Holding Register	ComBase+0 and DLAB=0 and `OUT`
Receive Buffer Register	ComBase+0 and DLAB=0 and `IN`
Interrupt Enable Register	ComBase+1 and DLAB=0
Interrupt ID Register	ComBase+2
Line Control Register	ComBase+3
Modem Control Register	ComBase+4
Line Status Register	ComBase+5
Modem Status Register	ComBase+6
DLAB=bit 7 of Line Control Register	
ComBase=port 03F8h for COM1, port 02F8h for COM2	

12.7.1 Selecting Baud Rate and Character Format

Before the 8250 ACE can be used, it must be programmed with baud rate, character format, and interrupt source selections. For standard baud rates and character formats and polled operation the BIOS initialization function described in a previous section is preferable. Otherwise, the 8250 ACE must be programmed by setting the control registers shown in Section 12.7.5.

The baud rate is selected by writing a two-byte divisor value into the DIVISOR LATCHES. An internal clock signal equal to 16 times the baud rate is obtained by dividing this divisor into the oscillator frequency. Several divisor values are shown below for an oscillator frequency of 1.8432 MHz:

Desired Baud Rate	**Decimal Divisor**	**Hexadecimal Divisor**
50	2304	0900
300	384	0180

Desired Baud Rate	**Decimal Divisor**	**Hexadecimal Divisor**
1200	96	0060
9600	12	000C

The Divisor Latches are accessed by first setting DLAB, the Divisor Latch Access Bit (bit 7 of the Line Control Register); after the divisor bytes are loaded, DLAB must be cleared for normal register addressing. E.g., to select 50 baud, write 80h to the Line Control Register to set DLAB, write 09h to the high-byte Divisor Latch, and 00h to the low-byte Divisor Latch. If the character format is programmed next, DLAB can be cleared then.

The character format is selected by programming the LINE CONTROL Register. E.g., to specify character having 7 information bits, a parity bit forced to 0, and 1 stop bit, write 00111010 to the Line Control Register.

12.7.2 Modem Control and Device Handshaking

The RS-232C interface signals Data Terminal Ready (DTR) and Request to Send (RTS) may be controlled via bits 0 and 1 of the MODEM CONTROL Register; Clear to Send (CTS), Data Set Ready (DSR), Ring Indicator (RI), and Received Line Signal Detect (RLSD)—also Carrier Detect (CD)—may be sensed as bits 4, 5, 6, and 7 of the MODEM STATUS register, and changes in CTS, DSR, RI, and RLSD since the last time the Modem Status Register was read as bits 0, 1, 2, and 3.

12.7.3 Transmit and Receive in the 8250 ACE

The 8250's transmitter uses a Transmit Shift Register (TSR), not accessible to the programmer, together with the Transmit Holding Register (THR) for double-buffered operation: a new character may be written into the THR while the previous character is being shifted out of the TSR on the Serial Out (SOUT) line. The character in THR (and the computed parity bit, if used) is automatically moved to TSR as soon as TSR is empty and serial transmission of the information bits preceded by a start bit and followed by the selected number of stop bits is started. The THR Empty (THRE) Flag indicates when THR can be loaded again; the TSR Empty (TSRE) Flag indicates similarly that TSR is empty (and that a character sent to THR would be immediately moved to TSR, so that a second character may be sent to THR without checking for THRE). The THRE and TSRE flags are found in the LINE STATUS Register; they are set initially by Master Reset. THRE is cleared when a character is loaded into THR.

The 8250's receiver similarly uses a Receive Shift Register (RSR), not accessible to the programmer, together with the Receive Buffer Register (RBR) for double-buffered operation: a character is held in RBR while the bits of the next character frame are being shifted into RSR on the Serial In (SIN) line. When the complete frame has been received, the start and stop bits are deleted, the parity check, if used, is computed, the character is automatically moved from RSR to RBR, and the Received Data Available (RDA) Flag is set. Also the Parity Error Flag is set if the parity check was used but failed, the Framing Error Flag is set if the line was not at the MARK level when the first stop bit was expected, and the Overrun Error Flag is set if the previous character in RBR had not been removed and was overwritten when the new character was moved into RBR from RSR. In addition, the Break Detected Flag is set if the line was in the SPACE condition for more than a character frame time ("long space"). These five flags are found in the LINE STATUS Register; they are cleared initially by Master Reset. RDA is also cleared whenever a character is read from RBR.

12.7.4 Input/Output without Interrupts

If no interrupts are used the INTERRUPT ENABLE Register should be cleared (initialization via BIOS does that) and OUT2 (bit 3 of the MODEM CONTROL Register) should be cleared. The polling routine should check THRE (bit 5 of the Line Status Register) and RDA (bit 0 of the Line Status Register); a new character can be sent to the Transmit Holding Register when THRE = 1; a new character can be read from the Receive Buffer Register when RDA = 1.

12.7.5 Bit Interpretation of Control and Status Registers

Table 12-6. Serial Interrupt Enable Register (@ ComBase+1, with DLAB=0)

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	0	0	0	Enable Modem Status Change (bits 0-3) Interrupt	Enable Receive Line Status (bits 1-4) Interrupt	Enable Transmit Holding Register Empty (THRE) Interrupt	Enable Received Data Available (RDA) Interrupt

Table 12-7. Serial Line Control Register (@ ComBase+3)

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bits 1, 0
Divisor Latch Access Bit (DLAB)	Set Break SOUT → 0; Long Space	Stick Parity (force parity to ~bit 4)	Even parity select	Parity Enable	# Stop bits: 0: 1 stop bit 1: 2 stop bits (1.5 if 3 info)	# Info bits: 00: 5 info bits 01: 6 info bits 10: 7 info bits 11: 8 info bits

Table 12-8. Modem Control Register (@ ComBase+4)

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	0	0	Loop (for diagnos-tics)	OUT2 (enables interrupts)	OUT1 (n.c.)	Request to Send (RTS)	Data Terminal Ready (DTR)

Table 12-9. Serial Line Status Register (@ ComBase+5)

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	TSR Empty	THR Empty	Break Detected	Framing Error	Parity Error	Overrun Error	Received Data Available (RDA)
		Write to THR	Read Line Status Register				Read RBR

Table 12-10. Modem Status Register (@ ComBase+6)

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Received Line Signal Detected (RLSD)	Ring Indicator (RI)	Data Set Ready (DSR)	Clear to Send (CTS)	Change in RLSD	Trailing Edge RI	Change in DSR	Change in CTS

12.7.6 Interrupt-Driven Input/Output

The following steps are necessary to allow interrupt-driven operation of the 8250 ACE (the discussion assumes the use of COM1):

1. Select the 8250 ACE interrupt sources by programming the INTERRUPT ENABLE Register
2. Set OUT2 in the MODEM CONTROL Register so that the interrupt signal from the 8250 ACE is passed to the IRQ4 Interrupt Request Line
3. Unmask IRQ4 at the 8259 Programmable Interrupt Controller by clearing bit 4 of the 8259's Mask Register (at port address 21h)
4. Enable interrupts in the CPU, 8250 ACE interrupts will cause an interrupt 0Ch in the CPU.

The interrupt 0Ch service routine can determine the source of the interrupt by examining the INTERRUPT ID Register and take the appropriate action. The interrupt condition in the 8250 ACE is typically reset automatically, as indicated in the description of the Interrupt ID Register. However, the 8259 Interrupt Controller must also be reset, by sending an End-of-Interrupt command byte (20h) to port 20h before returning form the interrupt service routine.

Unfortunately, the behavior of some versions of the UART is less than ideal. The following anomalies were described in an obscure application note:

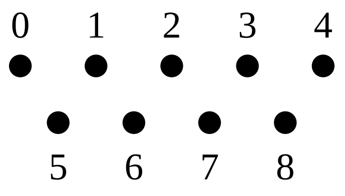
- In the 8250-B, used in many 8088-based PCs:
  - Enabling THRE interrupts by writing a "1" to bit 1 of the Interrupt Enable Register triggers a THRE interrupt even when the Transmit Holding Register (THR) is not empty. Thus, any character that happens to be waiting in THR will be lost. The recommended software fix for this anomaly is to enable THRE interrupts only when THR is empty, i.e., only when the THRE flag is true.
  - A random character may occasionally be transmitted at power-on. The recommended fix for the receiver is to discard any character that may be waiting in the Receive Buffer Register at initialization.

  - If the UART is never disabled, the Modem Status and Line Status Registers are never updated, the current error status indications cannot be read, and the character in THR will be transmitted repeatedly.
  - There are miscellaneous timing problems.

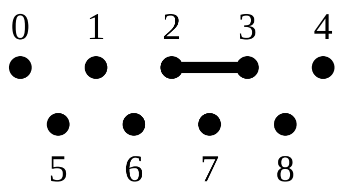
- In the 8250A (for 8086 CPUs) and the 16450 (for 80286 and later CPUs), the anomalies listed above have been eliminated, but a new anomaly has appeared:
  - A pending THRE interrupt may be lost if a high-priority (RDA, or Receive Line Status) interrupt occurs before the THRE interrupt is serviced. The following software fix is suggested: Before leaving the interrupt service routine for the high-priority interrupt, either disable and then re-enable THRE interrupts, or check the THRE flag and, if true, service the THRE condition immediately.

12.8 Creating a Null-Modem

Often during testing of software which utilizes the serial port, it is useful to have your machine "talk to itself." In other words, the receive and send lines on the serial port are in some fashion connected to one another. This means that whatever your computer sends out will be immediately received again—although for all intents and purposes, your computer does not know that the received data originated from itself. One quick way to accomplish this task is to use a so-called turnaround plug on the serial port (the shop often carries these under the part # DE 9S). When viewed from the front, the plug's pins will look like the following:



Also shown are the numbers assigned to each of the nine pins. Pins 2 and 3 must be connected together (a little solder and a short piece of wire will do the trick) as follows:



This is the most rudimentary form of a null-modem. It may not operate correctly for some applications which utilize more of the pins, but will work for any course MP's assigned.

Chapter 13

Parallel Communication

13.1 Printer Adapter Hardware

The parallel interface provides the signals and hardware to transfer data one character at a time (8 bits in parallel) between the CPU and a device, usually a printer. The signals consist of 8 data lines and 9 handshaking (status and control) lines. The hardware (latches and buffers for the data, status and control signals, and logic to connect them to the internal data bus and to address them as I/O ports via the address bus) is located on a printer adapter card that plugs into an I/O slot. The signals to and from the printer, shown in Table 13-1, are available on a 25-pin female D-shell connector on the printer adapter. Up to 3 printer adapters may be installed; the addresses available for the data, status, and control ports of an adapter are shown in Table 13-2. BIOS determines (during the restart initialization) which addresses have printer adapters installed; `DX` = 0 to 2 is then used within BIOS to refer to the printer adapters that were found to be present.

Table 13-1. Signals on 25-pin Printer Connector (pin numbers)

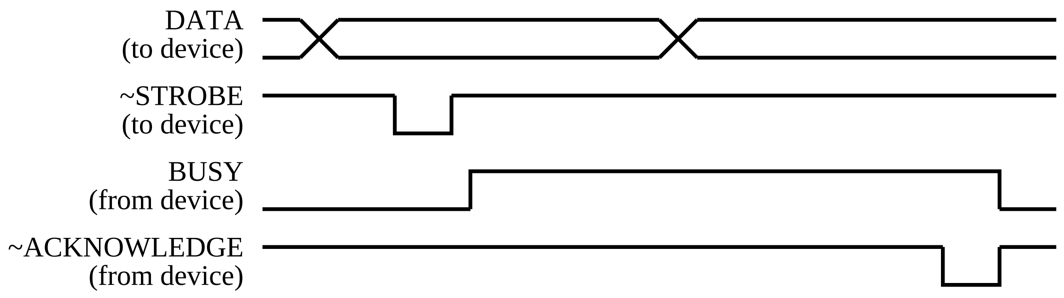
Data Signals [to device]		Status Signals [from device]		Control Signals [to device]	
Data 0-7	(2-9)	~Acknowledge	(10)	~Strobe	(1)
Ground	(18-25)	Busy	(11)	~Auto Feed	(14)
		Out of Paper	(12)	~Initialize	(16)
		Selected	(13)	~Select	(17)
		~Error	(15)		

Table 13-2. I/O Port Addresses (in hexadecimal) for Printer Adapter Buffer Registers

	Data Port	Status Port	Control Port
Monochrome Display / Printer Adapter	03BC	03BD	03BE
Primary Adapter	0378	0379	037A
Secondary Adapter	0278	0279	027A

The printer is assumed to use the "Centronics" protocol, shown in Figure 13-1. The printer sets BUSY high while it is processing a character; BUSY may also be high because the printer is disconnected, off-line, or in an error state.

Figure 13-1. Timing Diagram for the Centronics Protocol



In the *polled* mode of printing, the character bits are put on the DATA lines, BUSY is tested repeatedly until it is found to be low, then the ~STROBE pulse is sent. The printer sets BUSY high when the character data have been latched and sets it low again when the character has been processed. (The Centronics protocol specifies that the DATA lines be stable from at least 500 ns before to at least 500 ns after the ~STROBE pulse, and the ~STROBE pulse be at least 500 ns long. These times may of course be shortened for a specific printer, at the risk of loss of generality.) Programs using the polled mode should include a “timeout” counter to guard against a permanent BUSY condition. BIOS calls and DOS functions use this mode for printing.

In the *interrupt-driven* mode of printing, the positive-going edge of the ~ACKNOWLEDGE signal is used to cause an interrupt 0Fh via the IRQ7 line to the Interrupt Controller; the Interrupt Handler can send a new character to the printer whenever it is invoked, since ACKNOWLEDGE indicates that the previous character has been processed. The DOS command `PRINT` uses this mode to spool and print files.

13.2 BIOS and DOS Function Calls

BIOS call INT 17h has three subfunctions, selected with AH set to 0-2. Subfunctions assume `DX` = printer number (0-2); they return with `AH` = printer status byte (see below). The subfunctions are:

`AH` = 0: Print character specified in `AL`. If BUSY does not go low within about 16 seconds, a “timeout” is declared and BIOS returns with bit 0 of the statues byte set (and the character in `AL` is lost).

`AH` = 1: Initialize (set ~SELECT low, pulse ~INITIALIZE low, set ~AUTO FEED high, and disable IRQ7 interrupts from ~ACKNOWLEDGE). IBM- or EPSON-compatible printers respond to the INITIALIZE pulse by performing a carriage return and establishing the current line as top-of-page.

`AH` = 2: Get printer status byte into `AH`. The meaning (some of the signals have been inverted on the way from the connector) is:

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
~Busy	Ack	Out of Paper	Selected	I/O Error	*unused*	*unused*	Timeout

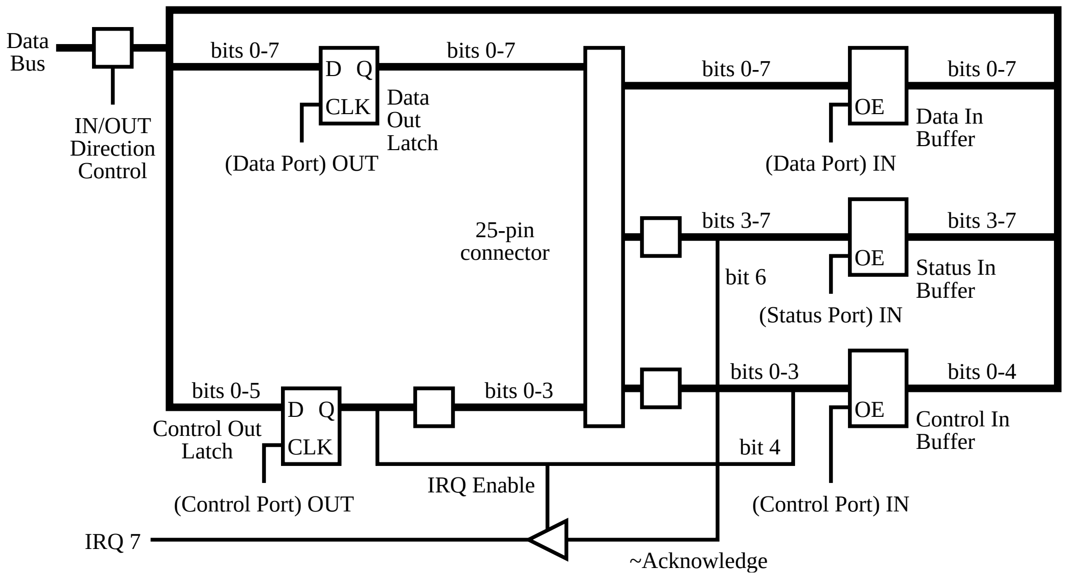
DOS function call INT 21h with `AH` = 5 prints the contents of `DL` interpreted as an ASCII character.

13.3 Machine Language Control of Printer Adapter

BIOS calls or DOS functions are most likely adequate for the control of a printer in the polled mode. Direct machine-language control is necessary, however, in situations such as the following:

- Interrupt-driven operation of the printer.
- Polled operation with features which, e.g., permit taking the printer offline for adding paper without incurring the automatic 16-second timeout and the consequent loss of a character.
- READING parallel data INTO the CPU from a device.

Figure 13-2. Simplified Logic Diagram of the Printer Adapter



Machine-language control uses `OUT` instructions to transfer a byte from register `AL` of the CPU to the adapter's Out latches, or `IN` instructions to transfer data from the In buffers to register `AL`. A simplified logic diagram is shown in Figure 13-2. Machine language control is straightforward but several peculiarities of the adapter logic must be noted:

- Data is sent to the printer by `OUT`putting a byte from register `AL` to the Data Out Latch. This latch has tri-state outputs which are always enabled. Thus, the printer's Data lines CANNOT be used for input; the Data In Buffer can only be used to read back to data in the Data Out Latch.
- Status signals from the printer are `IN`put into register `AL` via the Status In Buffer (bits 3-7 only). Table 13-3 shows the meaning assigned to the various bits. Note that some signals are complemented between the connector and the In Buffer. (The status byte returned by BIOS call 17h register `AH` has a timeout indicator added in bit 0 and, for some reason, has bits 3 and 6 complemented by software.)

Table 13-3. Register `AL` Bit Assignments for Printer Status Signals

Pin 11	Pin 10	Pin 12	Pin 13	Pin 15	

Pin 11	**Pin 10**	**Pin 12**	**Pin 13**	**Pin 15**			
Busy	~Ack	Out of Paper	Selected	~I/O Error			
Bit 7	**Bit 6**	**Bit 5**	**Bit 4**	**Bit 3**	**Bit 2**	**Bit 1**	**Bit 0**
~Busy	~Ack	Out of Paper	Selected	~I/O Error	n.c.	n.c.	n.c.

- Control signals are sent to the printer by `OUT`putting register `AL` (bits 0-5 only) to the Control Out Latch. Table 13-4 shows the meaning assigned to the various bits. Note that some signals are complemented between the latch output and the connector. Bit 4 (IRQ ENABLE) is not available at the connector; it is used to enable the ORing of the ACKNOWLEDGE status signal to the IRQ7 input to the 8259 Interrupt Controller (which will cause an interrupt 0Fh if not masked). Bit 5 is latched but not used further (see, however, the following section).

The control signals at the connector may be read back into register `AL` (bits 0-4 only) via the Control In Buffer. There are corresponding complementations between the connector and the In buffer (and bit 4 is simply a copy of the Out latch bit 4) so that Table 13-4 can be used in reverse for input. Note that since the Control Out Latch outputs are buffered, the control signal pins corresponding to control bits 0-3 *can* be used for input.

Table 13-4. Register `AL` Bit Assignments for Printer Control Signals

Bit 7	**Bit 6**	**Bit 5**	**Bit 4**	**Bit 3**	**Bit 2**	**Bit 1**	**Bit 0**
n.c.	n.c.	n.c.	IRQ Enable	Select	~Initialize	Auto Feed	Strobe
				Pin 17	**Pin 16**	**Pin 14**	**Pin 1**
				~Select	~Initialize	~Autofeed	~Strobe

To illustrate machine language control, a program fragment corresponding to subfunction 0 ("print the character in register `AL`") of BIOS call 17h is shown below:

```
        sti             ; allow higher-priority interrupt
        ...
        mov     si, dx  ; printer number
        mov     bl, [PrintTimeout+si]
                ; load timeout parameter byte (=10 for PC)
        shl     si, 1
        mov     dx, [PrinterBase+si]
                ; Data port address of printer in DX
        ...
        OR      AH, AH
        JZ      .B2
        ...
.B2:                    ; subfunction 0
        push    ax
        out     dx, al  ; send character to Data Out latch
        inc     dx      ; point to Status port
.B3:                    ; loop while BUSY until timeout
```

```
        sub     cx, cx  ; outer loop
.B3_1:                  ; inner loop
        in      al, dx  ; read printer status
        test    al, 80h ; test the BUSY status bit
        jnz     .B4     ; not busy
        loop    .B3_1   ; busy: repeat inner loop
        dec     bl      ; decr. outer loop counter
        jnz     .B3     ; repeat outer loop
        or      ah, 1   ; set timeout flag
        and     ah, 0F9h; clear unused bits
        jmp     ...     ; go to return with error flag set
        ...
.B4:                    ; NOT BUSY: send ~STROBE
        inc     dx      ; point to Control port
        mov     al, 0Dh ; set bit 0 (=STROBE) high - also sets
                        ;   IRQ ENABLE low, SELECT high,
                        ;   ~INITIALIZE high, and AUTO FEED low
        out     dx, al  ; send character to Control port
        mov     al, 0Ch ; set STROBE low again
        out     dx, al
        pop     ax
        ...             ; go to read the status into AH
        iret
```

13.4 Use of the Printer Adapter for Data Input

If 5 bits are sufficient for input, BIOS call 17h, subfunction 2, can be used to read the printer status lines into bits 3-7 of register AH—see Section 13.2. Note that the signals appearing in bits 7, 6, and 3 will have been complemented either by the printer adapter hardware or BIOS software, bits 2 and 1 will be low, and bit 0 will be set to the timeout flag value.

Either the 5 status lines or the 4 control lines can be read into register AL with machine language instructions—see Table 13-3 and Table 13-4 for the bit assignments and signal complementations. The control lines can safely be used for input since the Control Out Latch outputs are buffered properly. Unconnected lines will float high.

If 8-bit parallel input via the printer DATA lines is desired the adapter hardware must be modified so that the ~OE (OUTPUT ENABLE) input to the Data Out Latch is under program control. Bit 5 of the Control Out Latch can be used for this purpose; in fact, since bit 5 of the control byte is held low by the present version of BIOS this modification would not interfere with normal BIOS use. An improved adapter design would allow bit 5 of the Control Out Latch to be read back via the Control In Buffer.

Appendix A.
Intel x86 Instruction Reference

Originally written by Julian Hall and Simon Tantham.

This appendix provides a incomplete list of the machine instructions which NASM will assemble, and a short description of the function of each one. SSE, SSE2, 3DNow!, Cyrix MMX, and some undocumented instructions are not included in this list due to space concerns in the lab manual. See the NASM manual for a complete list of all the instructions NASM will assemble.

It is not intended to be exhaustive documentation on the fine details of the instructions' function, such as which exceptions they can trigger: for such documentation, you should go to Intel's Web site, http://www.intel.com/.

Instead, this appendix is intended primarily to provide documentation on the way the instructions may be used within NASM. For example, looking up `LOOP` will tell you that NASM allows `CX` or `ECX` to be specified as an optional second argument to the `LOOP` instruction, to enforce which of the two possible counter registers should be used if the default is not the one desired.

The instructions are not quite listed in alphabetical order, since groups of instructions with similar functions are lumped together in the same entry. Most of them don't move very far from their alphabetic position because of this.

A.1 Key to Operand Specifications

The instruction descriptions in this appendix specify their operands using the following notation:

Registers

`reg8` denotes an 8-bit general purpose register, `reg16` denotes a 16-bit general purpose register, and `reg32` a 32-bit one. `fpureg` denotes one of the eight FPU stack registers, `mmxreg` denotes one of the eight 64-bit MMX registers, and `segreg` denotes a segment register. In addition, some registers (such as `AL`, `DX` or `ECX`) may be specified explicitly.

Immediate operands

`imm` denotes a generic immediate operand. `imm8`, `imm16` and `imm32` are used when the operand is intended to be a specific size. For some of these instructions, NASM needs an explicit specifier: for example, `ADD ESP,16` could be interpreted as either `ADD r/m32,imm32` or `ADD r/m32,imm8`. NASM chooses the former by default, and so you must specify `ADD ESP,BYTE 16` for the latter.

Memory references

`mem` denotes a generic memory reference; `mem8`, `mem16`, `mem32`, `mem64` and `mem80` are used when the operand needs to be a specific size. Again, a specifier is needed in some cases: `DEC [address]` is ambiguous and will be rejected by NASM. You must specify `DEC BYTE [address]`, `DEC WORD [address]` or `DEC DWORD [address]` instead.

Restricted memory references

One form of the `MOV` instruction allows a memory address to be specified *without* allowing the normal range of register combinations and effective address processing. This is denoted by `memoffs8`, `memoffs16` and `memoffs32`.

Register or memory choices

Many instructions can accept either a register *or* a memory reference as an operand. `r/m8` is a shorthand for `reg8/mem8`; similarly `r/m16` and `r/m32`. `r/m64` is MMX-related, and is a shorthand for `mmxreg/mem64`.

A.2 Key to Opcode Descriptions

This appendix also provides the opcodes which NASM will generate for each form of each instruction. The opcodes are listed in the following way:

- A hex number, such as `3F`, indicates a fixed byte containing that number.
- A hex number followed by `+r`, such as `C8+r`, indicates that one of the operands to the instruction is a register, and the 'register value' of that register should be added to the hex number to produce the generated byte. For example, `EDX` has register value 2, so the code `C8+r`, when the register operand is `EDX`, generates the hex byte `CA`. Register values for specific registers are given in Section A.2.1.
- A hex number followed by `+cc`, such as `40+cc`, indicates that the instruction name has a condition code suffix, and the numeric representation of the condition code should be added to the hex number to produce the generated byte. For example, the code `40+cc`, when the instruction contains the `NE` condition, generates the hex byte `45`. Condition codes and their numeric representations are given in Section A.2.2.
- A slash followed by a digit, such as `/2`, indicates that one of the operands to the instruction is a memory address or register (denoted `mem` or `r/m`, with an optional size). This is to be encoded as an effective address, with a ModR/M byte, an optional SIB byte, and an optional displacement, and the spare (register) field of the ModR/M byte should be the digit given (which will be from 0 to 7, so it fits in three bits). The encoding of effective addresses is given in Section A.2.3.
- The code `/r` combines the above two: it indicates that one of the operands is a memory address or `r/m`, and another is a register, and that an effective address should be generated with the spare (register) field in the ModR/M byte being equal to the "register value" of the register operand. The encoding of effective addresses is given in Section A.2.3; register values are given in Section A.2.1.
- The codes `ib`, `iw` and `id` indicate that one of the operands to the instruction is an immediate value, and that this is to be encoded as a byte, little-endian word or little-endian doubleword respectively.
- The codes `rb`, `rw` and `rd` indicate that one of the operands to the instruction is an immediate value, and that the *difference* between this value and the address of the end of the instruction is to be encoded as a byte, word or doubleword respectively. Where the form `rw/rd` appears, it indicates that either `rw` or `rd` should be used according to whether assembly is being performed in `BITS 16` or `BITS 32` state respectively.

- The codes `ow` and `od` indicate that one of the operands to the instruction is a reference to the contents of a memory address specified as an immediate value: this encoding is used in some forms of the `MOV` instruction in place of the standard effective-address mechanism. The displacement is encoded as a word or doubleword. Again, `ow/od` denotes that `ow` or `od` should be chosen according to the `BITS` setting.
- The codes `o16` and `o32` indicate that the given form of the instruction should be assembled with operand size 16 or 32 bits. In other words, `o16` indicates a `66` prefix in `BITS 32` state, but generates no code in `BITS 16` state; and `o32` indicates a `66` prefix in `BITS 16` state but generates nothing in `BITS 32`.
- The codes `a16` and `a32`, similarly to `o16` and `o32`, indicate the address size of the given form of the instruction. Where this does not match the `BITS` setting, a `67` prefix is required.

A.2.1 Register Values

Where an instruction requires a register value, it is already implicit in the encoding of the rest of the instruction what type of register is intended: an 8-bit general-purpose register, a segment register, a debug register, an MMX register, or whatever. Therefore there is no problem with registers of different types sharing an encoding value.

The encodings for the various classes of register are:

8-bit general registers

`AL` is 0, `CL` is 1, `DL` is 2, `BL` is 3, `AH` is 4, `CH` is 5, `DH` is 6, and `BH` is 7.

16-bit general registers

`AX` is 0, `CX` is 1, `DX` is 2, `BX` is 3, `SP` is 4, `BP` is 5, `SI` is 6, and `DI` is 7.

32-bit general registers

`EAX` is 0, `ECX` is 1, `EDX` is 2, `EBX` is 3, `ESP` is 4, `EBP` is 5, `ESI` is 6, and `EDI` is 7.

Segment registers

`ES` is 0, `CS` is 1, `SS` is 2, `DS` is 3, `FS` is 4, and `GS` is 5.

Floating-point registers

`ST0` is 0, `ST1` is 1, `ST2` is 2, `ST3` is 3, `ST4` is 4, `ST5` is 5, `ST6` is 6, and `ST7` is 7.

64-bit MMX registers

`MM0` is 0, `MM1` is 1, `MM2` is 2, `MM3` is 3, `MM4` is 4, `MM5` is 5, `MM6` is 6, and `MM7` is 7.

Control registers

`CR0` is 0, `CR2` is 2, `CR3` is 3, and `CR4` is 4.

Debug registers

`DR0` is 0, `DR1` is 1, `DR2` is 2, `DR3` is 3, `DR6` is 6, and `DR7` is 7.

Test registers

`TR3` is 3, `TR4` is 4, `TR5` is 5, `TR6` is 6, and `TR7` is 7.

(Note that wherever a register name contains a number, that number is also the register value for that register.)

A.2.2 Condition Codes

The available condition codes are given here, along with their numeric representations as part of opcodes. Many of these condition codes have synonyms, so several will be listed at a time.

In the following descriptions, the word “either,” when applied to two possible trigger conditions, is used to mean “either or both”. If “either but not both” is meant, the phrase “exactly one of” is used.

- `O` is 0 (trigger if the overflow flag is set); `NO` is 1.
- `B`, `C` and `NAE` are 2 (trigger if the carry flag is set); `AE`, `NB` and `NC` are 3.
- `E` and `Z` are 4 (trigger if the zero flag is set); `NE` and `NZ` are 5.
- `BE` and `NA` are 6 (trigger if either of the carry or zero flags is set); `A` and `NBE` are 7.
- `S` is 8 (trigger if the sign flag is set); `NS` is 9.
- `P` and `PE` are 10 (trigger if the parity flag is set); `NP` and `PO` are 11.
- `L` and `NGE` are 12 (trigger if exactly one of the sign and overflow flags is set); `GE` and `NL` are 13.
- `LE` and `NG` are 14 (trigger if either the zero flag is set, or exactly one of the sign and overflow flags is set); `G` and `NLE` are 15.

Note that in all cases, the sense of a condition code may be reversed by changing the low bit of the numeric representation.

A.2.3 Effective Address Encoding: ModR/M and SIB

An effective address is encoded in up to three parts: a ModR/M byte, an optional SIB byte, and an optional byte, word or doubleword displacement field.

The ModR/M byte consists of three fields: the `mod` field, ranging from 0 to 3, in the upper two bits of the byte, the `r/m` field, ranging from 0 to 7, in the lower three bits, and the spare (register) field in the middle (bit 3 to bit 5). The spare field is not relevant to the effective address being encoded, and either contains an extension to the instruction opcode or the register value of another operand.

The ModR/M system can be used to encode a direct register reference rather than a memory access. This is always done by setting the `mod` field to 3 and the `r/m` field to the register value of the register in question (it must be a general-purpose register, and the size of the register must already be implicit in the encoding of the rest of the instruction). In this case, the SIB byte and displacement field are both absent.

In 16-bit addressing mode (either `BITS 16` with no `67` prefix, or `BITS 32` with a `67` prefix), the SIB byte is never used. The general rules for `mod` and `r/m` (there is an exception, given below) are:

- The mod field gives the length of the displacement field: 0 means no displacement, 1 means one byte, and 2 means two bytes.
- The r/m field encodes the combination of registers to be added to the displacement to give the accessed address: 0 means BX+SI, 1 means BX+DI, 2 means BP+SI, 3 means BP+DI, 4 means SI only, 5 means DI only, 6 means BP only, and 7 means BX only.

However, there is a special case:

- If mod is 0 and r/m is 6, the effective address encoded is not [BP] as the above rules would suggest, but instead [disp16]: the displacement field is present and is two bytes long, and no registers are added to the displacement.

Therefore the effective address [BP] cannot be encoded as efficiently as [BX]; so if you code [BP] in a program, NASM adds a notional 8-bit zero displacement, and sets mod to 1, r/m to 6, and the one-byte displacement field to 0.

In 32-bit addressing mode (either BITS 16 with a 67 prefix, or BITS 32 with no 67 prefix) the general rules (again, there are exceptions) for mod and r/m are:

- The mod field gives the length of the displacement field: 0 means no displacement, 1 means one byte, and 2 means four bytes.
- If only one register is to be added to the displacement, and it is not ESP, the r/m field gives its register value, and the SIB byte is absent. If the r/m field is 4 (which would encode ESP), the SIB byte is present and gives the combination and scaling of registers to be added to the displacement.

If the SIB byte is present, it describes the combination of registers (an optional base register, and an optional index register scaled by multiplication by 1, 2, 4 or 8) to be added to the displacement. The SIB byte is divided into the scale field, in the top two bits, the index field in the next three, and the base field in the bottom three. The general rules are:

- The base field encodes the register value of the base register.
- The index field encodes the register value of the index register, unless it is 4, in which case no index register is used (so ESP cannot be used as an index register).
- The scale field encodes the multiplier by which the index register is scaled before adding it to the base and displacement: 0 encodes a multiplier of 1, 1 encodes 2, 2 encodes 4 and 3 encodes 8.

The exceptions to the 32-bit encoding rules are:

- If mod is 0 and r/m is 5, the effective address encoded is not [EBP] as the above rules would suggest, but instead [disp32]: the displacement field is present and is four bytes long, and no registers are added to the displacement.
- If mod is 0, r/m is 4 (meaning the SIB byte is present) and base is 4, the effective address encoded is not [EBP+index] as the above rules would suggest, but instead [disp32+index]: the displacement field is present and is four bytes long, and there is no base register (but the index register is still processed in the normal way).

A.3 Key to Instruction Flags

Given along with each instruction in this appendix is a set of flags, denoting the type of the instruction. The types are as follows:

- `8086`, `186`, `286`, `386`, `486`, `PENT` and `P6` denote the lowest processor type that supports the instruction. Most instructions run on all processors above the given type; those that do not are documented. The Pentium II contains no additional instructions beyond the P6 (Pentium Pro); from the point of view of its instruction set, it can be thought of as a P6 with MMX capability.
- `FPU` indicates that the instruction is a floating-point one, and will only run on machines with a coprocessor (automatically including 486DX, Pentium and above).
- `MMX` indicates that the instruction is an MMX one, and will run on MMX-capable Pentium processors and the Pentium II.
- `PRIV` indicates that the instruction is a protected-mode management instruction. Many of these may only be used in protected mode, or only at privilege level zero.
- `UNDOC` indicates that the instruction is an undocumented one, and not part of the official Intel Architecture; it may or may not be supported on any given machine.

A.4 `AAA`, `AAS`, `AAM`, `AAD`: ASCII Adjustments

```
AAA                           ; 37                [8086]

AAS                           ; 3F                [8086]

AAD                           ; D5 0A             [8086]
AAD imm                       ; D5 ib             [8086]

AAM                           ; D4 0A             [8086]
AAM imm                       ; D4 ib             [8086]
```

These instructions are used in conjunction with the add, subtract, multiply and divide instructions to perform binary-coded decimal arithmetic in *unpacked* (one BCD digit per byte - easy to translate to and from ASCII, hence the instruction names) form. There are also packed BCD instructions `DAA` and `DAS`: see Section A.23.

`AAA` should be used after a one-byte `ADD` instruction whose destination was the `AL` register: by means of examining the value in the low nibble of `AL` and also the auxiliary carry flag `AF`, it determines whether the addition has overflowed, and adjusts it (and sets the carry flag) if so. You can add long BCD strings together by doing `ADD`/`AAA` on the low digits, then doing `ADC`/`AAA` on each subsequent digit.

`AAS` works similarly to `AAA`, but is for use after `SUB` instructions rather than `ADD`.

`AAM` is for use after you have multiplied two decimal digits together and left the result in `AL`: it divides `AL` by ten and stores the quotient in `AH`, leaving the remainder in `AL`. The divisor 10 can be changed by specifying an operand to the instruction: a particularly handy use of this is `AAM 16`, causing the two nibbles in `AL` to be separated into `AH` and `AL`.

`AAD` performs the inverse operation to `AAM`: it multiplies `AH` by ten, adds it to `AL`, and sets `AH` to zero. Again, the multiplier 10 can be changed.

A.5 `ADC`: Add with Carry

```
ADC r/m8,reg8                 ; 10 /r                [8086]
ADC r/m16,reg16               ; o16 11 /r            [8086]
ADC r/m32,reg32               ; o32 11 /r            [386]

ADC reg8,r/m8                 ; 12 /r                [8086]
ADC reg16,r/m16               ; o16 13 /r            [8086]
ADC reg32,r/m32               ; o32 13 /r            [386]

ADC r/m8,imm8                 ; 80 /2 ib             [8086]
ADC r/m16,imm16               ; o16 81 /2 iw         [8086]
ADC r/m32,imm32               ; o32 81 /2 id         [386]

ADC r/m16,imm8                ; o16 83 /2 ib         [8086]
ADC r/m32,imm8                ; o32 83 /2 ib         [386]

ADC AL,imm8                   ; 14 ib                [8086]
ADC AX,imm16                  ; o16 15 iw            [8086]
ADC EAX,imm32                 ; o32 15 id            [386]
```

`ADC` performs integer addition: it adds its two operands together, plus the value of the carry flag, and leaves the result in its destination (first) operand. The flags are set according to the result of the operation: in particular, the carry flag is affected and can be used by a subsequent `ADC` instruction.

In the forms with an 8-bit immediate second operand and a longer first operand, the second operand is considered to be signed, and is sign-extended to the length of the first operand. In these cases, the `BYTE` qualifier is necessary to force NASM to generate this form of the instruction.

To add two numbers without also adding the contents of the carry flag, use `ADD` (Section A.6).

A.6 `ADD`: Add Integers

```
ADD r/m8,reg8                 ; 00 /r                [8086]
ADD r/m16,reg16               ; o16 01 /r            [8086]
ADD r/m32,reg32               ; o32 01 /r            [386]

ADD reg8,r/m8                 ; 02 /r                [8086]
ADD reg16,r/m16               ; o16 03 /r            [8086]
ADD reg32,r/m32               ; o32 03 /r            [386]

ADD r/m8,imm8                 ; 80 /0 ib             [8086]
ADD r/m16,imm16               ; o16 81 /0 iw         [8086]
ADD r/m32,imm32               ; o32 81 /0 id         [386]

ADD r/m16,imm8                ; o16 83 /0 ib         [8086]
ADD r/m32,imm8                ; o32 83 /0 ib          [386]

ADD AL,imm8                   ; 04 ib                 [8086]
ADD AX,imm16                  ; o16 05 iw             [8086]
ADD EAX,imm32                 ; o32 05 id             [386]
```

`ADD` performs integer addition: it adds its two operands together, and leaves the result in its destination (first) operand. The flags are set according to the result of the operation: in particular, the carry flag is affected and can be used by a subsequent `ADC` instruction (Section A.5).

In the forms with an 8-bit immediate second operand and a longer first operand, the second operand is considered to be signed, and is sign-extended to the length of the first operand. In these cases, the `BYTE` qualifier is necessary to force NASM to generate this form of the instruction.

A.7 `AND`: Bitwise AND

```
AND r/m8,reg8                 ; 20 /r                 [8086]
AND r/m16,reg16               ; o16 21 /r             [8086]
AND r/m32,reg32               ; o32 21 /r             [386]

AND reg8,r/m8                 ; 22 /r                 [8086]
AND reg16,r/m16               ; o16 23 /r             [8086]
AND reg32,r/m32               ; o32 23 /r             [386]

AND r/m8,imm8                 ; 80 /4 ib              [8086]
AND r/m16,imm16               ; o16 81 /4 iw          [8086]
AND r/m32,imm32               ; o32 81 /4 id          [386]

AND r/m16,imm8                ; o16 83 /4 ib          [8086]
AND r/m32,imm8                ; o32 83 /4 ib          [386]

AND AL,imm8                   ; 24 ib                 [8086]
AND AX,imm16                  ; o16 25 iw             [8086]
AND EAX,imm32                 ; o32 25 id             [386]
```

`AND` performs a bitwise AND operation between its two operands (i.e. each bit of the result is 1 if and only if the corresponding bits of the two inputs were both 1), and stores the result in the destination (first) operand.

In the forms with an 8-bit immediate second operand and a longer first operand, the second operand is considered to be signed, and is sign-extended to the length of the first operand. In these cases, the `BYTE` qualifier is necessary to force NASM to generate this form of the instruction.

The MMX instruction `PAND` (see Section A.115) performs the same operation on the 64-bit MMX registers.

A.8 `ARPL`: Adjust RPL Field of Selector

```
ARPL r/m16,reg16              ; 63 /r                 [286,PRIV]
```

ARPL expects its two word operands to be segment selectors. It adjusts the RPL (requested privilege level - stored in the bottom two bits of the selector) field of the destination (first) operand to ensure that it is no less (i.e. no more privileged than) the RPL field of the source operand. The zero flag is set if and only if a change had to be made.

A.9 BOUND: Check Array Index against Bounds

```
BOUND reg16,mem               ; o16 62 /r           [186]
BOUND reg32,mem               ; o32 62 /r           [386]
```

BOUND expects its second operand to point to an area of memory containing two signed values of the same size as its first operand (i.e. two words for the 16-bit form; two doublewords for the 32-bit form). It performs two signed comparisons: if the value in the register passed as its first operand is less than the first of the in-memory values, or is greater than or equal to the second, it throws a BR exception. Otherwise, it does nothing.

A.10 BSF, BSR: Bit Scan

```
BSF reg16,r/m16               ; o16 0F BC /r        [386]
BSF reg32,r/m32               ; o32 0F BC /r        [386]

BSR reg16,r/m16               ; o16 0F BD /r        [386]
BSR reg32,r/m32               ; o32 0F BD /r        [386]
```

BSF searches for a set bit in its source (second) operand, starting from the bottom, and if it finds one, stores the index in its destination (first) operand. If no set bit is found, the contents of the destination operand are undefined.

BSR performs the same function, but searches from the top instead, so it finds the most significant set bit.

Bit indices are from 0 (least significant) to 15 or 31 (most significant).

A.11 BSWAP: Byte Swap

```
BSWAP reg32                   ; o32 0F C8+r         [486]
```

BSWAP swaps the order of the four bytes of a 32-bit register: bits 0-7 exchange places with bits 24-31, and bits 8-15 swap with bits 16-23. There is no explicit 16-bit equivalent: to byte-swap AX, BX, CX or DX, XCHG can be used (Section A.157).

A.12 BT, BTC, BTR, BTS: Bit Test

```
BT r/m16,reg16                ; o16 0F A3 /r        [386]
BT r/m32,reg32                ; o32 0F A3 /r        [386]
BT r/m16,imm8                 ; o16 0F BA /4 ib      [386]
BT r/m32,imm8                 ; o32 0F BA /4 ib      [386]

BTC r/m16,reg16               ; o16 0F BB /r         [386]
BTC r/m32,reg32               ; o32 0F BB /r         [386]
BTC r/m16,imm8                ; o16 0F BA /7 ib      [386]
BTC r/m32,imm8                ; o32 0F BA /7 ib      [386]

BTR r/m16,reg16               ; o16 0F B3 /r         [386]
BTR r/m32,reg32               ; o32 0F B3 /r         [386]
BTR r/m16,imm8                ; o16 0F BA /6 ib      [386]
BTR r/m32,imm8                ; o32 0F BA /6 ib      [386]

BTS r/m16,reg16               ; o16 0F AB /r         [386]
BTS r/m32,reg32               ; o32 0F AB /r         [386]
BTS r/m16,imm                 ; o16 0F BA /5 ib      [386]
BTS r/m32,imm                 ; o32 0F BA /5 ib      [386]
```

These instructions all test one bit of their first operand, whose index is given by the second operand, and store the value of that bit into the carry flag. Bit indices are from 0 (least significant) to 15 or 31 (most significant).

In addition to storing the original value of the bit into the carry flag, `BTR` also resets (clears) the bit in the operand itself. `BTS` sets the bit, and `BTC` complements the bit. `BT` does not modify its operands.

The bit offset should be no greater than the size of the operand.

A.13 `CALL`: Call Subroutine

```
CALL imm                      ; E8 rw/rd             [8086]
CALL imm:imm16                ; o16 9A iw iw         [8086]
CALL imm:imm32                ; o32 9A id iw         [386]
CALL FAR mem16                ; o16 FF /3            [8086]
CALL FAR mem32                ; o32 FF /3            [386]
CALL r/m16                    ; o16 FF /2            [8086]
CALL r/m32                    ; o32 FF /2            [386]
```

`CALL` calls a subroutine, by means of pushing the current instruction pointer (`IP`) and optionally `CS` as well on the stack, and then jumping to a given address.

`CS` is pushed as well as `IP` if and only if the call is a far call, i.e. a destination segment address is specified in the instruction. The forms involving two colon-separated arguments are far calls; so are the `CALL FAR mem` forms.

You can choose between the two immediate far call forms (`CALL imm:imm`) by the use of the `WORD` and `DWORD` keywords: `CALL WORD 0x1234:0x5678`) or `CALL DWORD 0x1234:0x56789abc`.

The `CALL FAR mem` forms execute a far call by loading the destination address out of memory. The address loaded consists of 16 or 32 bits of offset (depending on the operand size), and 16 bits of segment. The operand size may be overridden using `CALL WORD FAR mem` or `CALL DWORD FAR mem`.

The `CALL r/m` forms execute a near call (within the same segment), loading the destination address out of memory or out of a register. The keyword `NEAR` may be specified, for clarity, in these forms, but is not necessary. Again, operand size can be overridden using `CALL WORD mem` or `CALL DWORD mem`.

As a convenience, NASM does not require you to call a far procedure symbol by coding the cumbersome `CALL SEG routine:routine`, but instead allows the easier synonym `CALL FAR routine`.

The `CALL r/m` forms given above are near calls; NASM will accept the `NEAR` keyword (e.g. `CALL NEAR [address]`), even though it is not strictly necessary.

A.14 `CBW`, `CWD`, `CDQ`, `CWDE`: Sign Extensions

```
CBW                           ; o16 98                [8086]
CWD                           ; o16 99                [8086]
CDQ                           ; o32 99                [386]
CWDE                          ; o32 98                [386]
```

All these instructions sign-extend a short value into a longer one, by replicating the top bit of the original value to fill the extended one.

`CBW` extends `AL` into `AX` by repeating the top bit of `AL` in every bit of `AH`. `CWD` extends `AX` into `DX:AX` by repeating the top bit of `AX` throughout `DX`. `CWDE` extends `AX` into `EAX`, and `CDQ` extends `EAX` into `EDX:EAX`.

A.15 `CLC`, `CLD`, `CLI`, `CLTS`: Clear Flags

```
CLC                           ; F8                    [8086]
CLD                           ; FC                    [8086]
CLI                           ; FA                    [8086]
CLTS                          ; 0F 06                 [286,PRIV]
```

These instructions clear various flags. `CLC` clears the carry flag; `CLD` clears the direction flag; `CLI` clears the interrupt flag (thus disabling interrupts); and `CLTS` clears the task-switched (`TS`) flag in `CR0`.

To set the carry, direction, or interrupt flags, use the `STC`, `STD` and `STI` instructions (Section A.147). To invert the carry flag, use `CMC` (Section A.16).

A.16 `CMC`: Complement Carry Flag

```
CMC                           ; F5                    [8086]
```

`CMC` changes the value of the carry flag: if it was 0, it sets it to 1, and vice versa.

A.17 `CMOVcc`: Conditional Move

```
CMOVcc reg16,r/m16            ; o16 0F 40+cc /r       [P6]
CMOVcc reg32,r/m32            ; o32 0F 40+cc /r      [P6]
```

`CMOV` moves its source (second) operand into its destination (first) operand if the given condition code is satisfied; otherwise it does nothing.

For a list of condition codes, see Section A.2.2.

Although the `CMOV` instructions are flagged `P6` above, they may not be supported by all Pentium Pro processors; the `CPUID` instruction (Section A.22) will return a bit which indicates whether conditional moves are supported.

A.18 `CMP`: Compare Integers

```
CMP r/m8,reg8                 ; 38 /r                [8086]
CMP r/m16,reg16               ; o16 39 /r            [8086]
CMP r/m32,reg32               ; o32 39 /r            [386]

CMP reg8,r/m8                 ; 3A /r                [8086]
CMP reg16,r/m16               ; o16 3B /r            [8086]
CMP reg32,r/m32               ; o32 3B /r            [386]

CMP r/m8,imm8                 ; 80 /0 ib             [8086]
CMP r/m16,imm16               ; o16 81 /0 iw         [8086]
CMP r/m32,imm32               ; o32 81 /0 id         [386]

CMP r/m16,imm8                ; o16 83 /0 ib         [8086]
CMP r/m32,imm8                ; o32 83 /0 ib         [386]

CMP AL,imm8                   ; 3C ib                [8086]
CMP AX,imm16                  ; o16 3D iw            [8086]
CMP EAX,imm32                 ; o32 3D id            [386]
```

`CMP` performs a ‘mental’ subtraction of its second operand from its first operand, and affects the flags as if the subtraction had taken place, but does not store the result of the subtraction anywhere.

In the forms with an 8-bit immediate second operand and a longer first operand, the second operand is considered to be signed, and is sign-extended to the length of the first operand. In these cases, the `BYTE` qualifier is necessary to force NASM to generate this form of the instruction.

A.19 `CMPSB`, `CMPSW`, `CMPSD`: Compare Strings

```
CMPSB                         ; A6                   [8086]
CMPSW                         ; o16 A7               [8086]
CMPSD                         ; o32 A7               [386]
```

`CMPSB` compares the byte at `[DS:SI]` or `[DS:ESI]` with the byte at `[ES:DI]` or `[ES:EDI]`, and sets the flags accordingly. It then increments or decrements (depending on the direction flag: increments if the flag is clear, decrements if it is set) `SI` and `DI` (or `ESI` and `EDI`).

The registers used are `SI` and `DI` if the address size is 16 bits, and `ESI` and `EDI` if it is 32 bits. If you need to use an address size not equal to the current `BITS` setting, you can use an explicit `a16` or `a32` prefix.

The segment register used to load from `[SI]` or `[ESI]` can be overridden by using a segment register name as a prefix (for example, `es cmpsb`). The use of `ES` for the load from `[DI]` or `[EDI]` cannot be overridden.

`CMPSW` and `CMPSD` work in the same way, but they compare a word or a doubleword instead of a byte, and increment or decrement the addressing registers by 2 or 4 instead of 1.

The `REPE` and `REPNE` prefixes (equivalently, `REPZ` and `REPNZ`) may be used to repeat the instruction up to `CX` (or `ECX` - again, the address size chooses which) times until the first unequal or equal byte is found.

A.20 `CMPXCHG`: Compare and Exchange

```
CMPXCHG r/m8,reg8              ; 0F B0 /r             [PENT]
CMPXCHG r/m16,reg16            ; o16 0F B1 /r         [PENT]
CMPXCHG r/m32,reg32            ; o32 0F B1 /r         [PENT]
```

`CMPXCHG` compares its destination (first) operand to the value in `AL`, `AX` or `EAX` (depending on the size of the instruction). If they are equal, it copies its source (second) operand into the destination and sets the zero flag. Otherwise, it clears the zero flag and leaves the destination alone.

`CMPXCHG` is intended to be used for atomic operations in multitasking or multiprocessor environments. To safely update a value in shared memory, for example, you might load the value into `EAX`, load the updated value into `EBX`, and then execute the instruction `lock cmpxchg [value],ebx`. If `value` has not changed since being loaded, it is updated with your desired new value, and the zero flag is set to let you know it has worked. (The `LOCK` prefix prevents another processor doing anything in the middle of this operation: it guarantees atomicity.) However, if another processor has modified the value in between your load and your attempted store, the store does not happen, and you are notified of the failure by a cleared zero flag, so you can go round and try again.

A.21 `CMPXCHG8B`: Compare and Exchange Eight Bytes

```
CMPXCHG8B mem                  ; 0F C7 /1             [PENT]
```

This is a larger and more unwieldy version of `CMPXCHG`: it compares the 64-bit (eight-byte) value stored at `[mem]` with the value in `EDX:EAX`. If they are equal, it sets the zero flag and stores `ECX:EBX` into the memory area. If they are unequal, it clears the zero flag and leaves the memory area untouched.

A.22 `CPUID`: Get CPU Identification Code

```
CPUID                          ; 0F A2                [PENT]
```

`CPUID` returns various information about the processor it is being executed on. It fills the four registers `EAX`, `EBX`, `ECX` and `EDX` with information, which varies depending on the input contents of `EAX`.

`CPUID` also acts as a barrier to serialise instruction execution: executing the `CPUID` instruction guarantees that all the effects (memory modification, flag modification, register modification) of previous instructions have been completed before the next instruction gets fetched.

The information returned is as follows:

- If `EAX` is zero on input, `EAX` on output holds the maximum acceptable input value of `EAX`, and `EBX:EDX:ECX` contain the string `"GenuineIntel"` (or not, if you have a clone processor). That is to say, `EBX` contains `"Genu"` (in NASM's own sense of character constants, described in \k{chrconst}), `EDX` contains `"ineI"` and `ECX` contains `"ntel"`.
- If `EAX` is one on input, `EAX` on output contains version information about the processor, and `EDX` contains a set of feature flags, showing the presence and absence of various features. For example, bit 8 is set if the `CMPXCHG8B` instruction (Section A.21) is supported, bit 15 is set if the conditional move instructions (Section A.17 and Section A.34) are supported, and bit 23 is set if MMX instructions are supported.
- If `EAX` is two on input, `EAX`, `EBX`, `ECX` and `EDX` all contain information about caches and TLBs (Translation Lookahead Buffers).

For more information on the data returned from `CPUID`, see the documentation on Intel's web site.

A.23 DAA, DAS: Decimal Adjustments

```
DAA                           ; 27                  [8086]
DAS                           ; 2F                  [8086]
```

These instructions are used in conjunction with the add and subtract instructions to perform binary-coded decimal arithmetic in *packed* (one BCD digit per nibble) form. For the unpacked equivalents, see Section A.4.

`DAA` should be used after a one-byte `ADD` instruction whose destination was the `AL` register: by means of examining the value in the `AL` and also the auxiliary carry flag `AF`, it determines whether either digit of the addition has overflowed, and adjusts it (and sets the carry and auxiliary-carry flags) if so. You can add long BCD strings together by doing `ADD`/`DAA` on the low two digits, then doing `ADC`/`DAA` on each subsequent pair of digits.

`DAS` works similarly to `DAA`, but is for use after `SUB` instructions rather than `ADD`.

A.24 DEC: Decrement Integer

```
DEC reg16                     ; o16 48+r            [8086]
DEC reg32                     ; o32 48+r            [386]
DEC r/m8                      ; FE /1               [8086]
DEC r/m16                     ; o16 FF /1           [8086]
DEC r/m32                     ; o32 FF /1           [386]
```

`DEC` subtracts 1 from its operand. It does *not* affect the carry flag: to affect the carry flag, use `SUB something,1` (see Section A.150). See also `INC` (Section A.80).

A.25 DIV: Unsigned Integer Divide

```
DIV r/m8                      ; F6 /6                [8086]
DIV r/m16                     ; o16 F7 /6            [8086]
DIV r/m32                     ; o32 F7 /6            [386]
```

DIV performs unsigned integer division. The explicit operand provided is the divisor; the dividend and destination operands are implicit, in the following way:

- For DIV r/m8, AX is divided by the given operand; the quotient is stored in AL and the remainder in AH.
- For DIV r/m16, DX:AX is divided by the given operand; the quotient is stored in AX and the remainder in DX.
- For DIV r/m32, EDX:EAX is divided by the given operand; the quotient is stored in EAX and the remainder in EDX.

Signed integer division is performed by the IDIV instruction: see Section A.77.

A.26 EMMS: Empty MMX State

```
EMMS                          ; 0F 77                [PENT,MMX]
```

EMMS sets the FPU tag word (marking which floating-point registers are available) to all ones, meaning all registers are available for the FPU to use. It should be used after executing MMX instructions and before executing any subsequent floating-point operations.

A.27 ENTER: Create Stack Frame

```
ENTER imm,imm                 ; C8 iw ib             [186]
```

ENTER constructs a stack frame for a high-level language procedure call. The first operand (the iw in the opcode definition above refers to the first operand) gives the amount of stack space to allocate for local variables; the second (the ib above) gives the nesting level of the procedure (for languages like Pascal, with nested procedures).

The function of ENTER, with a nesting level of zero, is equivalent to

```
push    ebp             ; or push bp          in 16 bits
mov     ebp, esp        ; or mov bp, sp       in 16 bits
sub     esp, operand1   ; or sub sp, operand1 in 16 bits
```

This creates a stack frame with the procedure parameters accessible upwards from EBP, and local variables accessible downwards from EBP.

With a nesting level of one, the stack frame created is 4 (or 2) bytes bigger, and the value of the final frame pointer EBP is accessible in memory at [EBP-4].

This allows `ENTER`, when called with a nesting level of two, to look at the stack frame described by the *previous* value of `EBP`, find the frame pointer at offset -4 from that, and push it along with its new frame pointer, so that when a level-two procedure is called from within a level-one procedure, `[EBP-4]` holds the frame pointer of the most recent level-one procedure call and `[EBP-8]` holds that of the most recent level-two call. And so on, for nesting levels up to 31.

Stack frames created by `ENTER` can be destroyed by the `LEAVE` instruction: see Section A.95.

A.28 `F2XM1`: Calculate 2**X-1

```
F2XM1                          ; D9 F0              [8086,FPU]
```

`F2XM1` raises 2 to the power of `ST0`, subtracts one, and stores the result back into `ST0`. The initial contents of `ST0` must be a number in the range -1 to +1.

A.29 `FABS`: Floating-Point Absolute Value

```
FABS                           ; D9 E1              [8086,FPU]
```

`FABS` computes the absolute value of `ST0`, storing the result back in `ST0`.

A.30 `FADD`, `FADDP`: Floating-Point Addition

```
FADD mem32                     ; D8 /0              [8086,FPU]
FADD mem64                     ; DC /0              [8086,FPU]

FADD fpureg                    ; D8 C0+r            [8086,FPU]
FADD ST0,fpureg                ; D8 C0+r            [8086,FPU]

FADD TO fpureg                 ; DC C0+r            [8086,FPU]
FADD fpureg,ST0                ; DC C0+r            [8086,FPU]

FADDP fpureg                   ; DE C0+r            [8086,FPU]
FADDP fpureg,ST0               ; DE C0+r            [8086,FPU]
```

`FADD`, given one operand, adds the operand to `ST0` and stores the result back in `ST0`. If the operand has the `TO` modifier, the result is stored in the register given rather than in `ST0`.

`FADDP` performs the same function as `FADD TO`, but pops the register stack after storing the result.

The given two-operand forms are synonyms for the one-operand forms.

A.31 `FBLD`, `FBSTP`: BCD Floating-Point Load and Store

```
FBLD mem80                     ; DF /4              [8086,FPU]
```

```
FBSTP mem80                  ; DF /6               [8086,FPU]
```

FBLD loads an 80-bit (ten-byte) packed binary-coded decimal number from the given memory address, converts it to a real, and pushes it on the register stack. FBSTP stores the value of ST0, in packed BCD, at the given address and then pops the register stack.

A.32 FCHS: Floating-Point Change Sign

```
FCHS                         ; D9 E0               [8086,FPU]
```

FCHS negates the number in ST0: negative numbers become positive, and vice versa.

A.33 FCLEX, FNCLEX: Clear Floating-Point Exceptions

```
FCLEX                        ; 9B DB E2            [8086,FPU]
FNCLEX                       ; DB E2               [8086,FPU]
```

FCLEX clears any floating-point exceptions which may be pending. FNCLEX does the same thing but doesn't wait for previous floating-point operations (including the *handling* of pending exceptions) to finish first.

A.34 FCMOVcc: Floating-Point Conditional Move

```
FCMOVB fpureg                ; DA C0+r             [P6,FPU]
FCMOVB ST0,fpureg            ; DA C0+r             [P6,FPU]

FCMOVBE fpureg               ; DA D0+r             [P6,FPU]
FCMOVBE ST0,fpureg           ; DA D0+r             [P6,FPU]

FCMOVE fpureg                ; DA C8+r             [P6,FPU]
FCMOVE ST0,fpureg            ; DA C8+r             [P6,FPU]

FCMOVNB fpureg               ; DB C0+r             [P6,FPU]
FCMOVNB ST0,fpureg           ; DB C0+r             [P6,FPU]

FCMOVNBE fpureg              ; DB D0+r             [P6,FPU]
FCMOVNBE ST0,fpureg          ; DB D0+r             [P6,FPU]

FCMOVNE fpureg               ; DB C8+r             [P6,FPU]
FCMOVNE ST0,fpureg           ; DB C8+r             [P6,FPU]

FCMOVNU fpureg               ; DB D8+r             [P6,FPU]
FCMOVNU ST0,fpureg           ; DB D8+r             [P6,FPU]

FCMOVU fpureg                ; DA D8+r             [P6,FPU]
FCMOVU ST0,fpureg            ; DA D8+r             [P6,FPU]
```

The FCMOV instructions perform conditional move operations: each of them moves the contents of the given register into ST0 if its condition is satisfied, and does nothing if not.

The conditions are not the same as the standard condition codes used with conditional jump instructions. The conditions B, BE, NB, NBE, E and NE are exactly as normal, but none of the other standard ones are supported. Instead, the condition U and its counterpart NU are provided; the U condition is satisfied if the last two floating-point numbers compared were *unordered*, i.e. they were not equal but neither one could be said to be greater than the other, for example if they were NaNs. (The flag state which signals this is the setting of the parity flag: so the U condition is notionally equivalent to PE, and NU is equivalent to PO.)

The FCMOV conditions test the main processor's status flags, not the FPU status flags, so using FCMOV directly after FCOM will not work. Instead, you should either use FCOMI which writes directly to the main CPU flags word, or use FSTSW to extract the FPU flags.

Although the FCMOV instructions are flagged P6 above, they may not be supported by all Pentium Pro processors; the CPUID instruction (Section A.22) will return a bit which indicates whether conditional moves are supported.

A.35 FCOM, FCOMP, FCOMPP, FCOMI, FCOMIP: Floating-Point Compare

```
FCOM mem32                    ; D8 /2              [8086,FPU]
FCOM mem64                    ; DC /2              [8086,FPU]
FCOM fpureg                   ; D8 D0+r            [8086,FPU]
FCOM ST0,fpureg               ; D8 D0+r            [8086,FPU]

FCOMP mem32                   ; D8 /3              [8086,FPU]
FCOMP mem64                   ; DC /3              [8086,FPU]
FCOMP fpureg                  ; D8 D8+r            [8086,FPU]
FCOMP ST0,fpureg              ; D8 D8+r            [8086,FPU]

FCOMPP                        ; DE D9              [8086,FPU]

FCOMI fpureg                  ; DB F0+r            [P6,FPU]
FCOMI ST0,fpureg              ; DB F0+r            [P6,FPU]

FCOMIP fpureg                 ; DF F0+r            [P6,FPU]
FCOMIP ST0,fpureg             ; DF F0+r            [P6,FPU]
```

FCOM compares ST0 with the given operand, and sets the FPU flags accordingly. ST0 is treated as the left-hand side of the comparison, so that the carry flag is set (for a "less-than" result) if ST0 is less than the given operand.

FCOMP does the same as FCOM, but pops the register stack afterwards. FCOMPP compares ST0 with ST1 and then pops the register stack twice.

FCOMI and FCOMIP work like the corresponding forms of FCOM and FCOMP, but write their results directly to the CPU flags register rather than the FPU status word, so they can be immediately followed by conditional jump or conditional move instructions.

The `FCOM` instructions differ from the `FUCOM` instructions (Section A.69) only in the way they handle quiet NaNs: `FUCOM` will handle them silently and set the condition code flags to an “unordered” result, whereas `FCOM` will generate an exception.

A.36 `FCOS`: Cosine

```
FCOS                          ; D9 FF                [386,FPU]
```

`FCOS` computes the cosine of `ST0` (in radians), and stores the result in `ST0`. See also `FSINCOS` (Section A.61).

A.37 `FDECSTP`: Decrement Floating-Point Stack Pointer

```
FDECSTP                       ; D9 F6                [8086,FPU]
```

`FDECSTP` decrements the ‘top’ field in the floating-point status word. This has the effect of rotating the FPU register stack by one, as if the contents of `ST7` had been pushed on the stack. See also `FINCSTP` (Section A.46).

A.38 `FxDISI`, `FxENI`: Disable and Enable Floating-Point Interrupts

```
FDISI                         ; 9B DB E1             [8086,FPU]
FNDISI                        ; DB E1                [8086,FPU]

FENI                          ; 9B DB E0             [8086,FPU]
FNENI                         ; DB E0                [8086,FPU]
```

`FDISI` and `FENI` disable and enable floating-point interrupts. These instructions are only meaningful on original 8087 processors: the 287 and above treat them as no-operation instructions.

`FNDISI` and `FNENI` do the same thing as `FDISI` and `FENI` respectively, but without waiting for the floating-point processor to finish what it was doing first.

A.39 `FDIV`, `FDIVP`, `FDIVR`, `FDIVRP`: Floating-Point Division

```
FDIV mem32                    ; D8 /6                [8086,FPU]
FDIV mem64                    ; DC /6                [8086,FPU]

FDIV fpureg                   ; D8 F0+r              [8086,FPU]
FDIV ST0,fpureg               ; D8 F0+r              [8086,FPU]

FDIV TO fpureg                ; DC F8+r              [8086,FPU]
FDIV fpureg,ST0              ; DC F8+r              [8086,FPU]

FDIVR mem32                  ; D8 /0                [8086,FPU]
FDIVR mem64                  ; DC /0                [8086,FPU]

FDIVR fpureg                 ; D8 F8+r              [8086,FPU]
FDIVR ST0,fpureg             ; D8 F8+r              [8086,FPU]

FDIVR TO fpureg              ; DC F0+r              [8086,FPU]
FDIVR fpureg,ST0             ; DC F0+r              [8086,FPU]

FDIVP fpureg                 ; DE F8+r              [8086,FPU]
FDIVP fpureg,ST0             ; DE F8+r              [8086,FPU]

FDIVRP fpureg                ; DE F0+r              [8086,FPU]
FDIVRP fpureg,ST0            ; DE F0+r              [8086,FPU]
```

`FDIV` divides `ST0` by the given operand and stores the result back in `ST0`, unless the `TO` qualifier is given, in which case it divides the given operand by `ST0` and stores the result in the operand.

`FDIVR` does the same thing, but does the division the other way up: so if `TO` is not given, it divides the given operand by `ST0` and stores the result in `ST0`, whereas if `TO` is given it divides `ST0` by its operand and stores the result in the operand.

`FDIVP` operates like `FDIV TO`, but pops the register stack once it has finished. `FDIVRP` operates like `FDIVR TO`, but pops the register stack once it has finished.

A.40 `FFREE`: Flag Floating-Point Register as Unused

```
FFREE fpureg                 ; DD C0+r              [8086,FPU]
```

`FFREE` marks the given register as being empty.

A.41 `FIADD`: Floating-Point/Integer Addition

```
FIADD mem16                  ; DE /0                [8086,FPU]
FIADD mem32                  ; DA /0                [8086,FPU]
```

`FIADD` adds the 16-bit or 32-bit integer stored in the given memory location to `ST0`, storing the result in `ST0`.

A.42 `FICOM`, `FICOMP`: Floating-Point/Integer Compare

```
FICOM mem16                  ; DE /2                [8086,FPU]
FICOM mem32                  ; DA /2                [8086,FPU]

FICOMP mem16                 ; DE /3                [8086,FPU]
FICOMP mem32                  ; DA /3                [8086,FPU]
```

FICOM compares ST0 with the 16-bit or 32-bit integer stored in the given memory location, and sets the FPU flags accordingly. FICOMP does the same, but pops the register stack afterwards.

A.43 FIDIV, FIDIVR: Floating-Point/Integer Division

```
FIDIV mem16                   ; DE /6                [8086,FPU]
FIDIV mem32                   ; DA /6                [8086,FPU]

FIDIVR mem16                  ; DE /0                [8086,FPU]
FIDIVR mem32                  ; DA /0                [8086,FPU]
```

FIDIV divides ST0 by the 16-bit or 32-bit integer stored in the given memory location, and stores the result in ST0. FIDIVR does the division the other way up: it divides the integer by ST0, but still stores the result in ST0.

A.44 FILD, FIST, FISTP: Floating-Point/Integer Conversion

```
FILD mem16                    ; DF /0                [8086,FPU]
FILD mem32                    ; DB /0                [8086,FPU]
FILD mem64                    ; DF /5                [8086,FPU]

FIST mem16                    ; DF /2                [8086,FPU]
FIST mem32                    ; DB /2                [8086,FPU]

FISTP mem16                   ; DF /3                [8086,FPU]
FISTP mem32                   ; DB /3                [8086,FPU]
FISTP mem64                   ; DF /0                [8086,FPU]
```

FILD loads an integer out of a memory location, converts it to a real, and pushes it on the FPU register stack. FIST converts ST0 to an integer and stores that in memory; FISTP does the same as FIST, but pops the register stack afterwards.

A.45 FIMUL: Floating-Point/Integer Multiplication

```
FIMUL mem16                   ; DE /1                [8086,FPU]
FIMUL mem32                   ; DA /1                [8086,FPU]
```

FIMUL multiplies ST0 by the 16-bit or 32-bit integer stored in the given memory location, and stores the result in ST0.

A.46 `FINCSTP`: Increment Floating-Point Stack Pointer

```
FINCSTP                       ; D9 F7                [8086,FPU]
```

`FINCSTP` increments the ‘top’ field in the floating-point status word. This has the effect of rotating the FPU register stack by one, as if the register stack had been popped; however, unlike the popping of the stack performed by many FPU instructions, it does not flag the new `ST7` (previously `ST0`) as empty. See also `FDECSTP` (Section A.37).

A.47 `FINIT`, `FNINIT`: Initialise Floating-Point Unit

```
FINIT                         ; 9B DB E3             [8086,FPU]
FNINIT                        ; DB E3                [8086,FPU]
```

`FINIT` initialises the FPU to its default state. It flags all registers as empty, though it does not actually change their values. `FNINIT` does the same, without first waiting for pending exceptions to clear.

A.48 `FISUB`: Floating-Point/Integer Subtraction

```
FISUB mem16                   ; DE /4                [8086,FPU]
FISUB mem32                   ; DA /4                [8086,FPU]

FISUBR mem16                  ; DE /5                [8086,FPU]
FISUBR mem32                  ; DA /5                [8086,FPU]
```

`FISUB` subtracts the 16-bit or 32-bit integer stored in the given memory location from `ST0`, and stores the result in `ST0`. `FISUBR` does the subtraction the other way round, i.e. it subtracts `ST0` from the given integer, but still stores the result in `ST0`.

A.49 `FLD`: Floating-Point Load

```
FLD mem32                     ; D9 /0                [8086,FPU]
FLD mem64                     ; DD /0                [8086,FPU]
FLD mem80                     ; DB /5                [8086,FPU]
FLD fpureg                    ; D9 C0+r              [8086,FPU]
```

`FLD` loads a floating-point value out of the given register or memory location, and pushes it on the FPU register stack.

A.50 `FLDxx`: Floating-Point Load Constants

```
FLD1                          ; D9 E8                [8086,FPU]
FLDL2E                        ; D9 EA                [8086,FPU]
FLDL2T                        ; D9 E9                [8086,FPU]
```

```
FLDLG2                        ; D9 EC               [8086,FPU]
FLDLN2                        ; D9 ED               [8086,FPU]
FLDPI                         ; D9 EB               [8086,FPU]
FLDZ                          ; D9 EE               [8086,FPU]
```

These instructions push specific standard constants on the FPU register stack. `FLD1` pushes the value 1; `FLDL2E` pushes the base-2 logarithm of e; `FLDL2T` pushes the base-2 log of 10; `FLDLG2` pushes the base-10 log of 2; `FLDLN2` pushes the base-e log of 2; `FLDPI` pushes pi; and `FLDZ` pushes zero.

A.51 `FLDCW`: Load Floating-Point Control Word

```
FLDCW mem16                   ; D9 /5               [8086,FPU]
```

`FLDCW` loads a 16-bit value out of memory and stores it into the FPU control word (governing things like the rounding mode, the precision, and the exception masks). See also `FSTCW` (Section A.64).

A.52 `FLDENV`: Load Floating-Point Environment

```
FLDENV mem                    ; D9 /4               [8086,FPU]
```

`FLDENV` loads the FPU operating environment (control word, status word, tag word, instruction pointer, data pointer and last opcode) from memory. The memory area is 14 or 28 bytes long, depending on the CPU mode at the time. See also `FSTENV` (Section A.65).

A.53 `FMUL`, `FMULP`: Floating-Point Multiply

```
FMUL mem32                    ; D8 /1               [8086,FPU]
FMUL mem64                    ; DC /1               [8086,FPU]

FMUL fpureg                   ; D8 C8+r             [8086,FPU]
FMUL ST0,fpureg               ; D8 C8+r             [8086,FPU]

FMUL TO fpureg                ; DC C8+r             [8086,FPU]
FMUL fpureg,ST0               ; DC C8+r             [8086,FPU]

FMULP fpureg                  ; DE C8+r             [8086,FPU]
FMULP fpureg,ST0              ; DE C8+r             [8086,FPU]
```

`FMUL` multiplies `ST0` by the given operand, and stores the result in `ST0`, unless the `TO` qualifier is used in which case it stores the result in the operand. `FMULP` performs the same operation as `FMUL TO`, and then pops the register stack.

A.54 `FNOP`: Floating-Point No Operation

```
FNOP                          ; D9 D0                [8086,FPU]
```

`FNOP` does nothing.

A.55 `FPATAN`, `FPTAN`: Arctangent and Tangent

```
FPATAN                        ; D9 F3                [8086,FPU]
FPTAN                         ; D9 F2                [8086,FPU]
```

`FPATAN` computes the arctangent, in radians, of the result of dividing `ST1` by `ST0`, stores the result in `ST1`, and pops the register stack. It works like the C `atan2` function, in that changing the sign of both `ST0` and `ST1` changes the output value by pi (so it performs true rectangular-to-polar coordinate conversion, with `ST1` being the Y coordinate and `ST0` being the X coordinate, not merely an arctangent).

`FPTAN` computes the tangent of the value in `ST0` (in radians), and stores the result back into `ST0`.

A.56 `FPREM`, `FPREM1`: Floating-Point Partial Remainder

```
FPREM                         ; D9 F8                [8086,FPU]
FPREM1                        ; D9 F5                [386,FPU]
```

These instructions both produce the remainder obtained by dividing `ST0` by `ST1`. This is calculated, notionally, by dividing `ST0` by `ST1`, rounding the result to an integer, multiplying by `ST1` again, and computing the value which would need to be added back on to the result to get back to the original value in `ST0`.

The two instructions differ in the way the notional round-to-integer operation is performed. `FPREM` does it by rounding towards zero, so that the remainder it returns always has the same sign as the original value in `ST0`; `FPREM1` does it by rounding to the nearest integer, so that the remainder always has at most half the magnitude of `ST1`.

Both instructions calculate *partial* remainders, meaning that they may not manage to provide the final result, but might leave intermediate results in `ST0` instead. If this happens, they will set the C2 flag in the FPU status word; therefore, to calculate a remainder, you should repeatedly execute `FPREM` or `FPREM1` until C2 becomes clear.

A.57 `FRNDINT`: Floating-Point Round to Integer

```
FRNDINT                       ; D9 FC                [8086,FPU]
```

`FRNDINT` rounds the contents of `ST0` to an integer, according to the current rounding mode set in the FPU control word, and stores the result back in `ST0`.

A.58 FSAVE, FRSTOR: Save/Restore Floating-Point State

```
FSAVE mem                     ; 9B DD /6              [8086,FPU]
FNSAVE mem                    ; DD /6                 [8086,FPU]

FRSTOR mem                    ; DD /4                 [8086,FPU]
```

`FSAVE` saves the entire floating-point unit state, including all the information saved by `FSTENV` (Section A.65) plus the contents of all the registers, to a 94 or 108 byte area of memory (depending on the CPU mode). `FRSTOR` restores the floating-point state from the same area of memory.

`FNSAVE` does the same as `FSAVE`, without first waiting for pending floating-point exceptions to clear.

A.59 FSCALE: Scale Floating-Point Value by Power of Two

```
FSCALE                        ; D9 FD                 [8086,FPU]
```

`FSCALE` scales a number by a power of two: it rounds `ST1` towards zero to obtain an integer, then multiplies `ST0` by two to the power of that integer, and stores the result in `ST0`.

A.60 FSETPM: Set Protected Mode

```
FSETPM                        ; DB E4                 [286,FPU]
```

This instruction initalises protected mode on the 287 floating-point coprocessor. It is only meaningful on that processor: the 387 and above treat the instruction as a no-operation.

A.61 FSIN, FSINCOS: Sine and Cosine

```
FSIN                          ; D9 FE                 [386,FPU]
FSINCOS                       ; D9 FB                 [386,FPU]
```

`FSIN` calculates the sine of `ST0` (in radians) and stores the result in `ST0`. `FSINCOS` does the same, but then pushes the cosine of the same value on the register stack, so that the sine ends up in `ST1` and the cosine in `ST0`. `FSINCOS` is faster than executing `FSIN` and `FCOS` (see Section A.36) in succession.

A.62 FSQRT: Floating-Point Square Root

```
FSQRT                         ; D9 FA                 [8086,FPU]
```

`FSQRT` calculates the square root of `ST0` and stores the result in `ST0`.

A.63 `FST`, `FSTP`: Floating-Point Store

```
FST mem32                     ; D9 /2                  [8086,FPU]
FST mem64                     ; DD /2                  [8086,FPU]
FST fpureg                    ; DD D0+r                [8086,FPU]

FSTP mem32                    ; D9 /3                  [8086,FPU]
FSTP mem64                    ; DD /3                  [8086,FPU]
FSTP mem80                    ; DB /0                  [8086,FPU]
FSTP fpureg                   ; DD D8+r                [8086,FPU]
```

`FST` stores the value in `ST0` into the given memory location or other FPU register. `FSTP` does the same, but then pops the register stack.

A.64 `FSTCW`: Store Floating-Point Control Word

```
FSTCW mem16                   ; 9B D9 /0               [8086,FPU]
FNSTCW mem16                  ; D9 /0                  [8086,FPU]
```

`FSTCW` stores the FPU control word (governing things like the rounding mode, the precision, and the exception masks) into a 2-byte memory area. See also `FLDCW` (Section A.51).

`FNSTCW` does the same thing as `FSTCW`, without first waiting for pending floating-point exceptions to clear.

A.65 `FSTENV`: Store Floating-Point Environment

```
FSTENV mem                    ; 9B D9 /6               [8086,FPU]
FNSTENV mem                   ; D9 /6                  [8086,FPU]
```

`FSTENV` stores the FPU operating environment (control word, status word, tag word, instruction pointer, data pointer and last opcode) into memory. The memory area is 14 or 28 bytes long, depending on the CPU mode at the time. See also `FLDENV` (Section A.52).

`FNSTENV` does the same thing as `FSTENV`, without first waiting for pending floating-point exceptions to clear.

A.66 `FSTSW`: Store Floating-Point Status Word

```
FSTSW mem16                   ; 9B DD /0               [8086,FPU]
FSTSW AX                      ; 9B DF E0               [286,FPU]

FNSTSW mem16                  ; DD /0                  [8086,FPU]
FNSTSW AX                     ; DF E0                  [286,FPU]
```

`FSTSW` stores the FPU status word into `AX` or into a 2-byte memory area.

`FNSTSW` does the same thing as `FSTSW`, without first waiting for pending floating-point exceptions to clear.

A.67 `FSUB`, `FSUBP`, `FSUBR`, `FSUBRP`: Floating-Point Subtract

```
FSUB mem32                    ; D8 /4                [8086,FPU]
FSUB mem64                    ; DC /4                [8086,FPU]

FSUB fpureg                   ; D8 E0+r              [8086,FPU]
FSUB ST0,fpureg               ; D8 E0+r              [8086,FPU]

FSUB TO fpureg                ; DC E8+r              [8086,FPU]
FSUB fpureg,ST0               ; DC E8+r              [8086,FPU]

FSUBR mem32                   ; D8 /5                [8086,FPU]
FSUBR mem64                   ; DC /5                [8086,FPU]

FSUBR fpureg                  ; D8 E8+r              [8086,FPU]
FSUBR ST0,fpureg              ; D8 E8+r              [8086,FPU]

FSUBR TO fpureg               ; DC E0+r              [8086,FPU]
FSUBR fpureg,ST0              ; DC E0+r              [8086,FPU]

FSUBP fpureg                  ; DE E8+r              [8086,FPU]
FSUBP fpureg,ST0              ; DE E8+r              [8086,FPU]

FSUBRP fpureg                 ; DE E0+r              [8086,FPU]
FSUBRP fpureg,ST0             ; DE E0+r              [8086,FPU]
```

`FSUB` subtracts the given operand from `ST0` and stores the result back in `ST0`, unless the `TO` qualifier is given, in which case it subtracts `ST0` from the given operand and stores the result in the operand.

`FSUBR` does the same thing, but does the subtraction the other way up: so if `TO` is not given, it subtracts `ST0` from the given operand and stores the result in `ST0`, whereas if `TO` is given it subtracts its operand from `ST0` and stores the result in the operand.

`FSUBP` operates like `FSUB TO`, but pops the register stack once it has finished. `FSUBRP` operates like `FSUBR TO`, but pops the register stack once it has finished.

A.68 `FTST`: Test `ST0` Against Zero

```
FTST                          ; D9 E4                [8086,FPU]
```

`FTST` compares `ST0` with zero and sets the FPU flags accordingly. `ST0` is treated as the left-hand side of the comparison, so that a “less-than” result is generated if `ST0` is negative.

A.69 `FUCOMxx`: Floating-Point Unordered Compare

```
FUCOM fpureg                  ; DD E0+r              [386,FPU]
FUCOM ST0,fpureg              ; DD E0+r              [386,FPU]

FUCOMP fpureg                 ; DD E8+r              [386,FPU]
FUCOMP ST0,fpureg             ; DD E8+r              [386,FPU]

FUCOMPP                       ; DA E9                [386,FPU]

FUCOMI fpureg                 ; DB E8+r              [P6,FPU]
FUCOMI ST0,fpureg             ; DB E8+r              [P6,FPU]

FUCOMIP fpureg                ; DF E8+r              [P6,FPU]
FUCOMIP ST0,fpureg            ; DF E8+r              [P6,FPU]
```

`FUCOM` compares `ST0` with the given operand, and sets the FPU flags accordingly. `ST0` is treated as the left-hand side of the comparison, so that the carry flag is set (for a "less-than" result) if `ST0` is less than the given operand.

`FUCOMP` does the same as `FUCOM`, but pops the register stack afterwards. `FUCOMPP` compares `ST0` with `ST1` and then pops the register stack twice.

`FUCOMI` and `FUCOMIP` work like the corresponding forms of `FUCOM` and `FUCOMP`, but write their results directly to the CPU flags register rather than the FPU status word, so they can be immediately followed by conditional jump or conditional move instructions.

The `FUCOM` instructions differ from the `FCOM` instructions (Section A.35) only in the way they handle quiet NaNs: `FUCOM` will handle them silently and set the condition code flags to an "unordered" result, whereas `FCOM` will generate an exception.

A.70 `FXAM`: Examine Class of Value in `ST0`

```
FXAM                          ; D9 E5                [8086,FPU]
```

`FXAM` sets the FPU flags C3, C2 and C0 depending on the type of value stored in `ST0`: 000 (respectively) for an unsupported format, 001 for a NaN, 010 for a normal finite number, 011 for an infinity, 100 for a zero, 101 for an empty register, and 110 for a denormal. It also sets the C1 flag to the sign of the number.

A.71 `FXCH`: Floating-Point Exchange

```
FXCH                          ; D9 C9                [8086,FPU]
FXCH fpureg                   ; D9 C8+r              [8086,FPU]
FXCH fpureg,ST0               ; D9 C8+r              [8086,FPU]
FXCH ST0,fpureg               ; D9 C8+r              [8086,FPU]
```

`FXCH` exchanges `ST0` with a given FPU register. The no-operand form exchanges `ST0` with `ST1`.

A.72 `FXRSTOR`: Restore FP and MMXTM State and Streaming SIMD Extension State

```
FXRSTOR memory           ; 0F,AE,/1              [P6,SSE,FPU]
```

The `FXRSTOR` instruction reloads the FP and MMXTM technology state, and the Streaming SIMD Extension state (environment and registers), from the 512-byte memory area defined by memory. This data should have been written by a previous FXSAVE.

A.73 `FXSAVE`: Store FP and MMXTM State and Streaming SIMD

```
FXSAVE memory           ; 0F,AE,/0         [P6,SSE,FPU]
```

`FXSAVE`The FXSAVE instruction writes the current FP and MMXTM technology state, and Streaming SIMD Extension state (environment and registers), to the specified 512-byte destination defined by memory. It does this without checking for pending unmasked floating-point exceptions (similar to the operation of FNSAVE). Unlike the FSAVE/FNSAVE instructions, the processor retains the contents of the FP and MMXTM technology state and Streaming SIMD Extension state in the processor after the state has been saved. This instruction has been optimized to maximize floating-point save performance.

A.74 `FXTRACT`: Extract Exponent and Significand

```
FXTRACT                       ; D9 F4                [8086,FPU]
```

`FXTRACT` separates the number in `ST0` into its exponent and significand (mantissa), stores the exponent back into `ST0`, and then pushes the significand on the register stack (so that the significand ends up in `ST0`, and the exponent in `ST1`).

A.75 `FYL2X`, `FYL2XP1`: Compute Y times Log2(X) or Log2(X+1)

```
FYL2X                         ; D9 F1                [8086,FPU]
FYL2XP1                       ; D9 F9                [8086,FPU]
```

`FYL2X` multiplies `ST1` by the base-2 logarithm of `ST0`, stores the result in `ST1`, and pops the register stack (so that the result ends up in `ST0`). `ST0` must be non-zero and positive.

`FYL2XP1` works the same way, but replacing the base-2 log of `ST0` with that of `ST0` plus one. This time, `ST0` must have magnitude no greater than 1 minus half the square root of two.

A.76 `HLT`: Halt Processor

```
HLT                           ; F4                   [8086]
```

`HLT` puts the processor into a halted state, where it will perform no more operations until restarted by an interrupt or a reset.

A.77 `IDIV`: Signed Integer Divide

```
IDIV r/m8                     ; F6 /7                 [8086]
IDIV r/m16                    ; o16 F7 /7             [8086]
IDIV r/m32                    ; o32 F7 /7             [386]
```

`IDIV` performs signed integer division. The explicit operand provided is the divisor; the dividend and destination operands are implicit, in the following way:

- For `IDIV r/m8`, `AX` is divided by the given operand; the quotient is stored in `AL` and the remainder in `AH`.
- For `IDIV r/m16`, `DX:AX` is divided by the given operand; the quotient is stored in `AX` and the remainder in `DX`.
- For `IDIV r/m32`, `EDX:EAX` is divided by the given operand; the quotient is stored in `EAX` and the remainder in `EDX`.

Unsigned integer division is performed by the `DIV` instruction: see Section A.25.

A.78 `IMUL`: Signed Integer Multiply

```
IMUL r/m8                     ; F6 /5                 [8086]
IMUL r/m16                    ; o16 F7 /5             [8086]
IMUL r/m32                    ; o32 F7 /5             [386]

IMUL reg16,r/m16              ; o16 0F AF /r          [386]
IMUL reg32,r/m32              ; o32 0F AF /r          [386]

IMUL reg16,imm8               ; o16 6B /r ib          [286]
IMUL reg16,imm16              ; o16 69 /r iw          [286]
IMUL reg32,imm8               ; o32 6B /r ib          [386]
IMUL reg32,imm32              ; o32 69 /r id          [386]

IMUL reg16,r/m16,imm8         ; o16 6B /r ib          [286]
IMUL reg16,r/m16,imm16        ; o16 69 /r iw          [286]
IMUL reg32,r/m32,imm8         ; o32 6B /r ib          [386]
IMUL reg32,r/m32,imm32        ; o32 69 /r id          [386]
```

`IMUL` performs signed integer multiplication. For the single-operand form, the other operand and destination are implicit, in the following way:

- For `IMUL r/m8`, `AL` is multiplied by the given operand; the product is stored in `AX`.
- For `IMUL r/m16`, `AX` is multiplied by the given operand; the product is stored in `DX:AX`.
- For `IMUL r/m32`, `EAX` is multiplied by the given operand; the product is stored in `EDX:EAX`.

The two-operand form multiplies its two operands and stores the result in the destination (first) operand. The three-operand form multiplies its last two operands and stores the result in the first operand.

The two-operand form is in fact a shorthand for the three-operand form, as can be seen by examining the opcode descriptions: in the two-operand form, the code `/r` takes both its register and `r/m` parts from the same operand (the first one).

In the forms with an 8-bit immediate operand and another longer source operand, the immediate operand is considered to be signed, and is sign-extended to the length of the other source operand. In these cases, the `BYTE` qualifier is necessary to force NASM to generate this form of the instruction.

Unsigned integer multiplication is performed by the `MUL` instruction: see Section A.107.

A.79 `IN`: Input from I/O Port

```
IN AL,imm8                    ; E4 ib               [8086]
IN AX,imm8                    ; o16 E5 ib           [8086]
IN EAX,imm8                   ; o32 E5 ib           [386]
IN AL,DX                      ; EC                  [8086]
IN AX,DX                      ; o16 ED              [8086]
IN EAX,DX                     ; o32 ED              [386]
```

`IN` reads a byte, word or doubleword from the specified I/O port, and stores it in the given destination register. The port number may be specified as an immediate value if it is between 0 and 255, and otherwise must be stored in `DX`. See also `OUT` (Section A.111).

A.80 `INC`: Increment Integer

```
INC reg16                     ; o16 40+r            [8086]
INC reg32                     ; o32 40+r            [386]
INC r/m8                      ; FE /0               [8086]
INC r/m16                     ; o16 FF /0           [8086]
INC r/m32                     ; o32 FF /0           [386]
```

`INC` adds 1 to its operand. It does *not* affect the carry flag: to affect the carry flag, use `ADD something,1` (see Section A.6). See also `DEC` (Section A.24).

A.81 `INSB`, `INSW`, `INSD`: Input String from I/O Port

```
INSB                          ; 6C                  [186]
INSW                          ; o16 6D              [186]
INSD                          ; o32 6D              [386]
```

`INSB` inputs a byte from the I/O port specified in `DX` and stores it at `[ES:DI]` or `[ES:EDI]`. It then increments or decrements (depending on the direction flag: increments if the flag is clear, decrements if it is set) `DI` or `EDI`.

The register used is `DI` if the address size is 16 bits, and `EDI` if it is 32 bits. If you need to use an address size not equal to the current `BITS` setting, you can use an explicit `a16` or `a32` prefix.

Segment override prefixes have no effect for this instruction: the use of `ES` for the load from `[DI]` or `[EDI]` cannot be overridden.

`INSW` and `INSD` work in the same way, but they input a word or a doubleword instead of a byte, and increment or decrement the addressing register by 2 or 4 instead of 1.

The `REP` prefix may be used to repeat the instruction `CX` (or `ECX` - again, the address size chooses which) times.

See also `OUTSB`, `OUTSW` and `OUTSD` (Section A.112).

A.82 `INT`: Software Interrupt

```
INT imm8                      ; CD ib                [8086]
```

`INT` causes a software interrupt through a specified vector number from 0 to 255.

The code generated by the `INT` instruction is always two bytes long: although there are short forms for some `INT` instructions, NASM does not generate them when it sees the `INT` mnemonic. In order to generate single-byte breakpoint instructions, use the `INT3` or `INT1` instructions (see Section A.83) instead.

A.83 `INT3`, `INT1`, `ICEBP`, `INT01`: Breakpoints

```
INT1                          ; F1                   [P6]
ICEBP                         ; F1                   [P6]
INT01                         ; F1                   [P6]

INT3                          ; CC                   [8086]
```

`INT1` and `INT3` are short one-byte forms of the instructions `INT 1` and `INT 3` (see Section A.82). They perform a similar function to their longer counterparts, but take up less code space. They are used as breakpoints by debuggers.

`INT1`, and its alternative synonyms `INT01` and `ICEBP`, is an instruction used by in-circuit emulators (ICEs). It is present, though not documented, on some processors down to the 286, but is only documented for the Pentium Pro. `INT3` is the instruction normally used as a breakpoint by debuggers.

`INT3` is not precisely equivalent to `INT 3`: the short form, since it is designed to be used as a breakpoint, bypasses the normal IOPL checks in virtual-8086 mode, and also does not go through interrupt redirection.

A.84 `INTO`: Interrupt if Overflow

```
INTO                          ; CE                   [8086]
```

`INTO` performs an `INT 4` software interrupt (see Section A.82) if and only if the overflow flag is set.

A.85 INVD: Invalidate Internal Caches

```
INVD                          ; 0F 08                [486]
```

INVD invalidates and empties the processor's internal caches, and causes the processor to instruct external caches to do the same. It does not write the contents of the caches back to memory first: any modified data held in the caches will be lost. To write the data back first, use WBINVD (Section A.154).

A.86 INVLPG: Invalidate TLB Entry

```
INVLPG mem                    ; 0F 01 /0             [486]
```

INVLPG invalidates the translation lookahead buffer (TLB) entry associated with the supplied memory address.

A.87 IRET, IRETW, IRETD: Return from Interrupt

```
IRET                          ; CF                   [8086]
IRETW                         ; o16 CF               [8086]
IRETD                         ; o32 CF               [386]
```

IRET returns from an interrupt (hardware or software) by means of popping IP (or EIP), CS, and the flags off the stack and then continuing execution from the new CS:IP.

IRETW pops IP, CS and the flags as 2 bytes each, taking 6 bytes off the stack in total. IRETD pops EIP as 4 bytes, pops a further 4 bytes of which the top two are discarded and the bottom two go into CS, and pops the flags as 4 bytes as well, taking 12 bytes off the stack.

IRET is a shorthand for either IRETW or IRETD, depending on the default BITS setting at the time.

A.88 JCXZ, JECXZ: Jump if CX/ECX Zero

```
JCXZ imm                      ; a16 E3 rb            [8086]
JECXZ imm                     ; a32 E3 rb            [386]
```

JCXZ performs a short jump (with maximum range 128 bytes) if and only if the contents of the CX register is 0. JECXZ does the same thing, but with ECX.

A.89 JMP: Jump

```
JMP imm                       ; E9 rw/rd             [8086]
JMP SHORT imm                 ; EB rb                [8086]
JMP imm:imm16                 ; o16 EA iw iw         [8086]
JMP imm:imm32                 ; o32 EA id iw         [386]
JMP FAR mem                   ; o16 FF /5            [8086]
JMP FAR mem                     ; o32 FF /5            [386]
JMP r/m16                       ; o16 FF /4            [8086]
JMP r/m32                       ; o32 FF /4            [386]
```

`JMP` jumps to a given address. The address may be specified as an absolute segment and offset, or as a relative jump within the current segment.

`JMP SHORT imm` has a maximum range of 128 bytes, since the displacement is specified as only 8 bits, but takes up less code space. NASM does not choose when to generate `JMP SHORT` for you: you must explicitly code `SHORT` every time you want a short jump.

You can choose between the two immediate far jump forms (`JMP imm:imm`) by the use of the `WORD` and `DWORD` keywords: `JMP WORD 0x1234:0x5678`) or `JMP DWORD 0x1234:0x56789abc`.

The `JMP FAR mem` forms execute a far jump by loading the destination address out of memory. The address loaded consists of 16 or 32 bits of offset (depending on the operand size), and 16 bits of segment. The operand size may be overridden using `JMP WORD FAR mem` or `JMP DWORD FAR mem`.

The `JMP r/m` forms execute a near jump (within the same segment), loading the destination address out of memory or out of a register. The keyword `NEAR` may be specified, for clarity, in these forms, but is not necessary. Again, operand size can be overridden using `JMP WORD mem` or `JMP DWORD mem`.

As a convenience, NASM does not require you to jump to a far symbol by coding the cumbersome `JMP SEG routine:routine`, but instead allows the easier synonym `JMP FAR routine`.

The `CALL r/m` forms given above are near calls; NASM will accept the `NEAR` keyword (e.g. `CALL NEAR [address]`), even though it is not strictly necessary.

A.90 `Jcc`: Conditional Branch

```
Jcc imm                         ; 70+cc rb             [8086]
Jcc NEAR imm                    ; 0F 80+cc rw/rd       [386]
```

The conditional jump instructions execute a near (same segment) jump if and only if their conditions are satisfied. For example, `JNZ` jumps only if the zero flag is not set.

The ordinary form of the instructions has only a 128-byte range; the `NEAR` form is a 386 extension to the instruction set, and can span the full size of a segment. NASM will not override your choice of jump instruction: if you want `Jcc NEAR`, you have to use the `NEAR` keyword.

The `SHORT` keyword is allowed on the first form of the instruction, for clarity, but is not necessary.

A.91 `LAHF`: Load AH from Flags

```
LAHF                            ; 9F                   [8086]
```

`LAHF` sets the `AH` register according to the contents of the low byte of the flags word. See also `SAHF` (Section A.137).

A.92 `LAR`: Load Access Rights

```
LAR reg16,r/m16                 ; o16 0F 02 /r          [286,PRIV]
LAR reg32,r/m32                 ; o32 0F 02 /r          [286,PRIV]
```

`LAR` takes the segment selector specified by its source (second) operand, finds the corresponding segment descriptor in the GDT or LDT, and loads the access-rights byte of the descriptor into its destination (first) operand.

A.93 `LDS`, `LES`, `LFS`, `LGS`, `LSS`: Load Far Pointer

```
LDS reg16,mem                   ; o16 C5 /r             [8086]
LDS reg32,mem                   ; o32 C5 /r             [8086]

LES reg16,mem                   ; o16 C4 /r             [8086]
LES reg32,mem                   ; o32 C4 /r             [8086]

LFS reg16,mem                   ; o16 0F B4 /r          [386]
LFS reg32,mem                   ; o32 0F B4 /r          [386]

LGS reg16,mem                   ; o16 0F B5 /r          [386]
LGS reg32,mem                   ; o32 0F B5 /r          [386]

LSS reg16,mem                   ; o16 0F B2 /r          [386]
LSS reg32,mem                   ; o32 0F B2 /r          [386]
```

These instructions load an entire far pointer (16 or 32 bits of offset, plus 16 bits of segment) out of memory in one go. `LDS`, for example, loads 16 or 32 bits from the given memory address into the given register (depending on the size of the register), then loads the *next* 16 bits from memory into `DS`. `LES`, `LFS`, `LGS` and `LSS` work in the same way but use the other segment registers.

A.94 `LEA`: Load Effective Address

```
LEA reg16,mem                   ; o16 8D /r             [8086]
LEA reg32,mem                   ; o32 8D /r             [8086]
```

`LEA`, despite its syntax, does not access memory. It calculates the effective address specified by its second operand as if it were going to load or store data from it, but instead it stores the calculated address into the register specified by its first operand. This can be used to perform quite complex calculations (e.g. `LEA EAX,[EBX+ECX*4+100]`) in one instruction.

`LEA`, despite being a purely arithmetic instruction which accesses no memory, still requires square brackets around its second operand, as if it were a memory reference.

A.95 `LEAVE`: Destroy Stack Frame

```
LEAVE                         ; C9                   [186]
```

`LEAVE` destroys a stack frame of the form created by the `ENTER` instruction (see Section A.27). It is functionally equivalent to `MOV ESP,EBP` followed by `POP EBP` (or `MOV SP,BP` followed by `POP BP` in 16-bit mode).

A.96 `LGDT`, `LIDT`, `LLDT`: Load Descriptor Tables

```
LGDT mem                      ; 0F 01 /2             [286,PRIV]
LIDT mem                      ; 0F 01 /3             [286,PRIV]
LLDT r/m16                    ; 0F 00 /2             [286,PRIV]
```

`LGDT` and `LIDT` both take a 6-byte memory area as an operand: they load a 32-bit linear address and a 16-bit size limit from that area (in the opposite order) into the GDTR (global descriptor table register) or IDTR (interrupt descriptor table register). These are the only instructions which directly use *linear* addresses, rather than segment/offset pairs.

`LLDT` takes a segment selector as an operand. The processor looks up that selector in the GDT and stores the limit and base address given there into the LDTR (local descriptor table register).

See also `SGDT`, `SIDT` and `SLDT` (Section A.143).

A.97 `LMSW`: Load/Store Machine Status Word

```
LMSW r/m16                    ; 0F 01 /6             [286,PRIV]
```

`LMSW` loads the bottom four bits of the source operand into the bottom four bits of the `CR0` control register (or the Machine Status Word, on 286 processors). See also `SMSW` (Section A.146).

A.98 `LODSB`, `LODSW`, `LODSD`: Load from String

```
LODSB                         ; AC                   [8086]
LODSW                         ; o16 AD               [8086]
LODSD                         ; o32 AD               [386]
```

`LODSB` loads a byte from `[DS:SI]` or `[DS:ESI]` into `AL`. It then increments or decrements (depending on the direction flag: increments if the flag is clear, decrements if it is set) `SI` or `ESI`.

The register used is `SI` if the address size is 16 bits, and `ESI` if it is 32 bits. If you need to use an address size not equal to the current `BITS` setting, you can use an explicit `a16` or `a32` prefix.

The segment register used to load from `[SI]` or `[ESI]` can be overridden by using a segment register name as a prefix (for example, `es lodsb`).

`LODSW` and `LODSD` work in the same way, but they load a word or a doubleword instead of a byte, and increment or decrement the addressing registers by 2 or 4 instead of 1.

A.99 `LOOP`, `LOOPE`, `LOOPZ`, `LOOPNE`, `LOOPNZ`: Loop with Counter

```
LOOP imm                      ; E2 rb               [8086]
LOOP imm,CX                   ; a16 E2 rb           [8086]
LOOP imm,ECX                  ; a32 E2 rb           [386]

LOOPE imm                     ; E1 rb               [8086]
LOOPE imm,CX                  ; a16 E1 rb           [8086]
LOOPE imm,ECX                 ; a32 E1 rb           [386]
LOOPZ imm                     ; E1 rb               [8086]
LOOPZ imm,CX                  ; a16 E1 rb           [8086]
LOOPZ imm,ECX                 ; a32 E1 rb           [386]

LOOPNE imm                    ; E0 rb               [8086]
LOOPNE imm,CX                 ; a16 E0 rb           [8086]
LOOPNE imm,ECX                ; a32 E0 rb           [386]
LOOPNZ imm                    ; E0 rb               [8086]
LOOPNZ imm,CX                 ; a16 E0 rb           [8086]
LOOPNZ imm,ECX                ; a32 E0 rb           [386]
```

`LOOP` decrements its counter register (either `CX` or `ECX`—if one is not specified explicitly, the `BITS` setting dictates which is used) by one, and if the counter does not become zero as a result of this operation, it jumps to the given label. The jump has a range of 128 bytes.

`LOOPE` (or its synonym `LOOPZ`) adds the additional condition that it only jumps if the counter is nonzero *and* the zero flag is set. Similarly, `LOOPNE` (and `LOOPNZ`) jumps only if the counter is nonzero and the zero flag is clear.

A.100 `LSL`: Load Segment Limit

```
LSL reg16,r/m16               ; o16 0F 03 /r        [286,PRIV]
LSL reg32,r/m32               ; o32 0F 03 /r        [286,PRIV]
```

`LSL` is given a segment selector in its source (second) operand; it computes the segment limit value by loading the segment limit field from the associated segment descriptor in the GDT or LDT. (This involves shifting left by 12 bits if the segment limit is page-granular, and not if it is byte-granular; so you end up with a byte limit in either case.) The segment limit obtained is then loaded into the destination (first) operand.

A.101 `LTR`: Load Task Register

```
LTR r/m16                     ; 0F 00 /3            [286,PRIV]
```

`LTR` looks up the segment base and limit in the GDT or LDT descriptor specified by the segment selector given as its operand, and loads them into the Task Register.

A.102 `MOV`: Move Data

```
MOV r/m8,reg8                 ; 88 /r               [8086]
MOV r/m16,reg16               ; o16 89 /r           [8086]
MOV r/m32,reg32               ; o32 89 /r           [386]
MOV reg8,r/m8                 ; 8A /r               [8086]
MOV reg16,r/m16               ; o16 8B /r           [8086]
MOV reg32,r/m32               ; o32 8B /r           [386]

MOV reg8,imm8                 ; B0+r ib             [8086]
MOV reg16,imm16               ; o16 B8+r iw         [8086]
MOV reg32,imm32               ; o32 B8+r id         [386]
MOV r/m8,imm8                 ; C6 /0 ib            [8086]
MOV r/m16,imm16               ; o16 C7 /0 iw        [8086]
MOV r/m32,imm32               ; o32 C7 /0 id        [386]

MOV AL,memoffs8               ; A0 ow/od            [8086]
MOV AX,memoffs16              ; o16 A1 ow/od        [8086]
MOV EAX,memoffs32             ; o32 A1 ow/od        [386]
MOV memoffs8,AL               ; A2 ow/od            [8086]
MOV memoffs16,AX              ; o16 A3 ow/od        [8086]
MOV memoffs32,EAX             ; o32 A3 ow/od        [386]

MOV r/m16,segreg              ; o16 8C /r           [8086]
MOV r/m32,segreg              ; o32 8C /r           [386]
MOV segreg,r/m16              ; o16 8E /r           [8086]
MOV segreg,r/m32              ; o32 8E /r           [386]

MOV reg32,CR0/2/3/4           ; 0F 20 /r            [386]
MOV reg32,DR0/1/2/3/6/7       ; 0F 21 /r            [386]
MOV reg32,TR3/4/5/6/7         ; 0F 24 /r            [386]
MOV CR0/2/3/4,reg32           ; 0F 22 /r            [386]
MOV DR0/1/2/3/6/7,reg32       ; 0F 23 /r            [386]
MOV TR3/4/5/6/7,reg32         ; 0F 26 /r            [386]
```

`MOV` copies the contents of its source (second) operand into its destination (first) operand.

In all forms of the `MOV` instruction, the two operands are the same size, except for moving between a segment register and an `r/m32` operand. These instructions are treated exactly like the corresponding 16-bit equivalent (so that, for example, `MOV DS,EAX` functions identically to `MOV DS,AX` but saves a prefix when in 32-bit mode), except that when a segment register is moved into a 32-bit destination, the top two bytes of the result are undefined.

`MOV` may not use `CS` as a destination.

`CR4` is only a supported register on the Pentium and above.

A.103 `MOVD`: Move Doubleword to/from MMX Register

```
MOVD mmxreg,r/m32             ; 0F 6E /r            [PENT,MMX]
MOVD r/m32,mmxreg             ; 0F 7E /r            [PENT,MMX]
```

`MOVD` copies 32 bits from its source (second) operand into its destination (first) operand. When the destination is a 64-bit MMX register, the top 32 bits are set to zero.

A.104 `MOVQ`: Move Quadword to/from MMX Register

```
MOVQ mmxreg,r/m64             ; 0F 6F /r             [PENT,MMX]
MOVQ r/m64,mmxreg             ; 0F 7F /r             [PENT,MMX]
```

`MOVQ` copies 64 bits from its source (second) operand into its destination (first) operand.

A.105 `MOVSB`, `MOVSW`, `MOVSD`: Move String

```
MOVSB                         ; A4                   [8086]
MOVSW                         ; o16 A5               [8086]
MOVSD                         ; o32 A5               [386]
```

`MOVSB` copies the byte at `[ES:DI]` or `[ES:EDI]` to `[DS:SI]` or `[DS:ESI]`. It then increments or decrements (depending on the direction flag: increments if the flag is clear, decrements if it is set) `SI` and `DI` (or `ESI` and `EDI`).

The registers used are `SI` and `DI` if the address size is 16 bits, and `ESI` and `EDI` if it is 32 bits. If you need to use an address size not equal to the current `BITS` setting, you can use an explicit `a16` or `a32` prefix.

The segment register used to load from `[SI]` or `[ESI]` can be overridden by using a segment register name as a prefix (for example, `es movsb`). The use of `ES` for the store to `[DI]` or `[EDI]` cannot be overridden.

`MOVSW` and `MOVSD` work in the same way, but they copy a word or a doubleword instead of a byte, and increment or decrement the addressing registers by 2 or 4 instead of 1.

The `REP` prefix may be used to repeat the instruction `CX` (or `ECX` - again, the address size chooses which) times.

A.106 `MOVSX`, `MOVZX`: Move Data with Sign or Zero Extend

```
MOVSX reg16,r/m8              ; o16 0F BE /r         [386]
MOVSX reg32,r/m8              ; o32 0F BE /r         [386]
MOVSX reg32,r/m16             ; o32 0F BF /r         [386]

MOVZX reg16,r/m8              ; o16 0F B6 /r         [386]
MOVZX reg32,r/m8              ; o32 0F B6 /r         [386]
MOVZX reg32,r/m16             ; o32 0F B7 /r         [386]
```

`MOVSX` sign-extends its source (second) operand to the length of its destination (first) operand, and copies the result into the destination operand. `MOVZX` does the same, but zero-extends rather than sign-extending.

A.107 `MUL`: Unsigned Integer Multiply

```
MUL r/m8                      ; F6 /4                  [8086]
MUL r/m16                     ; o16 F7 /4              [8086]
MUL r/m32                     ; o32 F7 /4              [386]
```

`MUL` performs unsigned integer multiplication. The other operand to the multiplication, and the destination operand, are implicit, in the following way:

- For `MUL r/m8`, `AL` is multiplied by the given operand; the product is stored in `AX`.
- For `MUL r/m16`, `AX` is multiplied by the given operand; the product is stored in `DX:AX`.
- For `MUL r/m32`, `EAX` is multiplied by the given operand; the product is stored in `EDX:EAX`.

Signed integer multiplication is performed by the `IMUL` instruction: see Section A.78.

A.108 `NEG`, `NOT`: Two's and One's Complement

```
NEG r/m8                      ; F6 /3                  [8086]
NEG r/m16                     ; o16 F7 /3              [8086]
NEG r/m32                     ; o32 F7 /3              [386]

NOT r/m8                      ; F6 /2                  [8086]
NOT r/m16                     ; o16 F7 /2              [8086]
NOT r/m32                     ; o32 F7 /2              [386]
```

`NEG` replaces the contents of its operand by the two's complement negation (invert all the bits and then add one) of the original value. `NOT`, similarly, performs one's complement (inverts all the bits).

A.109 `NOP`: No Operation

```
NOP                           ; 90                     [8086]
```

`NOP` performs no operation. Its opcode is the same as that generated by `XCHG AX,AX` or `XCHG EAX,EAX` (depending on the processor mode; see Section A.157).

A.110 `OR`: Bitwise OR

```
OR r/m8,reg8                  ; 08 /r                  [8086]
OR r/m16,reg16                ; o16 09 /r              [8086]
OR r/m32,reg32                ; o32 09 /r              [386]

OR reg8,r/m8                  ; 0A /r                  [8086]
OR reg16,r/m16                ; o16 0B /r              [8086]
OR reg32,r/m32                ; o32 0B /r              [386]
OR r/m8,imm8                  ; 80 /1 ib             [8086]
OR r/m16,imm16                ; o16 81 /1 iw         [8086]
OR r/m32,imm32                ; o32 81 /1 id         [386]

OR r/m16,imm8                 ; o16 83 /1 ib         [8086]
OR r/m32,imm8                 ; o32 83 /1 ib         [386]

OR AL,imm8                    ; 0C ib                [8086]
OR AX,imm16                   ; o16 0D iw            [8086]
OR EAX,imm32                  ; o32 0D id            [386]
```

`OR` performs a bitwise OR operation between its two operands (i.e. each bit of the result is 1 if and only if at least one of the corresponding bits of the two inputs was 1), and stores the result in the destination (first) operand.

In the forms with an 8-bit immediate second operand and a longer first operand, the second operand is considered to be signed, and is sign-extended to the length of the first operand. In these cases, the `BYTE` qualifier is necessary to force NASM to generate this form of the instruction.

The MMX instruction `POR` (see Section A.122) performs the same operation on the 64-bit MMX registers.

A.111 `OUT`: Output Data to I/O Port

```
OUT imm8,AL                   ; E6 ib                [8086]
OUT imm8,AX                   ; o16 E7 ib            [8086]
OUT imm8,EAX                  ; o32 E7 ib            [386]
OUT DX,AL                     ; EE                   [8086]
OUT DX,AX                     ; o16 EF               [8086]
OUT DX,EAX                    ; o32 EF               [386]
```

`OUT` writes the contents of the given source register to the specified I/O port. The port number may be specified as an immediate value if it is between 0 and 255, and otherwise must be stored in `DX`. See also `IN` (Section A.79).

A.112 `OUTSB`, `OUTSW`, `OUTSD`: Output String to I/O Port

```
OUTSB                         ; 6E                   [186]
OUTSW                         ; o16 6F               [186]
OUTSD                         ; o32 6F               [386]
```

`OUTSB` loads a byte from `[DS:SI]` or `[DS:ESI]` and writes it to the I/O port specified in `DX`. It then increments or decrements (depending on the direction flag: increments if the flag is clear, decrements if it is set) `SI` or `ESI`.

The register used is `SI` if the address size is 16 bits, and `ESI` if it is 32 bits. If you need to use an address size not equal to the current `BITS` setting, you can use an explicit `a16` or `a32` prefix.

The segment register used to load from `[SI]` or `[ESI]` can be overridden by using a segment register name as a prefix (for example, `es outsb`).

OUTSW and OUTSD work in the same way, but they output a word or a doubleword instead of a byte, and increment or decrement the addressing registers by 2 or 4 instead of 1.

The REP prefix may be used to repeat the instruction CX (or ECX - again, the address size chooses which) times.

A.113 PACKSSDW, PACKSSWB, PACKUSWB: Pack Data

```
PACKSSDW mmxreg,r/m64         ; 0F 6B /r             [PENT,MMX]
PACKSSWB mmxreg,r/m64         ; 0F 63 /r             [PENT,MMX]
PACKUSWB mmxreg,r/m64         ; 0F 67 /r             [PENT,MMX]
```

All these instructions start by forming a notional 128-bit word by placing the source (second) operand on the left of the destination (first) operand. PACKSSDW then splits this 128-bit word into four doublewords, converts each to a word, and loads them side by side into the destination register; PACKSSWB and PACKUSWB both split the 128-bit word into eight words, converts each to a byte, and loads *those* side by side into the destination register.

PACKSSDW and PACKSSWB perform signed saturation when reducing the length of numbers: if the number is too large to fit into the reduced space, they replace it by the largest signed number (7FFFh or 7Fh) that *will* fit, and if it is too small then they replace it by the smallest signed number (8000h or 80h) that will fit. PACKUSWB performs unsigned saturation: it treats its input as unsigned, and replaces it by the largest unsigned number that will fit.

A.114 PADDxx: MMX Packed Addition

```
PADDB mmxreg,r/m64            ; 0F FC /r             [PENT,MMX]
PADDW mmxreg,r/m64            ; 0F FD /r             [PENT,MMX]
PADDD mmxreg,r/m64            ; 0F FE /r             [PENT,MMX]

PADDSB mmxreg,r/m64           ; 0F EC /r             [PENT,MMX]
PADDSW mmxreg,r/m64           ; 0F ED /r             [PENT,MMX]

PADDUSB mmxreg,r/m64          ; 0F DC /r             [PENT,MMX]
PADDUSW mmxreg,r/m64          ; 0F DD /r             [PENT,MMX]
```

PADDxx all perform packed addition between their two 64-bit operands, storing the result in the destination (first) operand. The PADDxB forms treat the 64-bit operands as vectors of eight bytes, and add each byte individually; PADDxW treat the operands as vectors of four words; and PADDD treats its operands as vectors of two doublewords.

PADDSB and PADDSW perform signed saturation on the sum of each pair of bytes or words: if the result of an addition is too large or too small to fit into a signed byte or word result, it is clipped (saturated) to the largest or smallest value which *will* fit. PADDUSB and PADDUSW similarly perform unsigned saturation, clipping to 0FFh or 0FFFFh if the result is larger than that.

A.115 `PAND`, `PANDN`: MMX Bitwise AND and AND-NOT

```
PAND mmxreg,r/m64             ; 0F DB /r              [PENT,MMX]
PANDN mmxreg,r/m64            ; 0F DF /r              [PENT,MMX]
```

`PAND` performs a bitwise AND operation between its two operands (i.e. each bit of the result is 1 if and only if the corresponding bits of the two inputs were both 1), and stores the result in the destination (first) operand.

`PANDN` performs the same operation, but performs a one's complement operation on the destination (first) operand first.

A.116 `PCMPxx`: MMX Packed Comparison

```
PCMPEQB mmxreg,r/m64          ; 0F 74 /r              [PENT,MMX]
PCMPEQW mmxreg,r/m64          ; 0F 75 /r              [PENT,MMX]
PCMPEQD mmxreg,r/m64          ; 0F 76 /r              [PENT,MMX]

PCMPGTB mmxreg,r/m64          ; 0F 64 /r              [PENT,MMX]
PCMPGTW mmxreg,r/m64          ; 0F 65 /r              [PENT,MMX]
PCMPGTD mmxreg,r/m64          ; 0F 66 /r              [PENT,MMX]
```

The `PCMPxx` instructions all treat their operands as vectors of bytes, words, or doublewords; corresponding elements of the source and destination are compared, and the corresponding element of the destination (first) operand is set to all zeros or all ones depending on the result of the comparison.

`PCMPxxB` treats the operands as vectors of eight bytes, `PCMPxxW` treats them as vectors of four words, and `PCMPxxD` as two doublewords.

`PCMPEQx` sets the corresponding element of the destination operand to all ones if the two elements compared are equal; `PCMPGTx` sets the destination element to all ones if the element of the first (destination) operand is greater (treated as a signed integer) than that of the second (source) operand.

A.117 `PMADDWD`: MMX Packed Multiply and Add

```
PMADDWD mmxreg,r/m64          ; 0F F5 /r              [PENT,MMX]
```

`PMADDWD` treats its two inputs as vectors of four signed words. It multiplies corresponding elements of the two operands, giving four signed doubleword results. The top two of these are added and placed in the top 32 bits of the destination (first) operand; the bottom two are added and placed in the bottom 32 bits.

A.118 `PMULHW`, `PMULLW`: MMX Packed Multiply

```
PMULHW mmxreg,r/m64           ; 0F E5 /r              [PENT,MMX]
PMULLW mmxreg,r/m64           ; 0F D5 /r              [PENT,MMX]
```

`PMULxW` treats its two inputs as vectors of four signed words. It multiplies corresponding elements of the two operands, giving four signed doubleword results.

`PMULHW` then stores the top 16 bits of each doubleword in the destination (first) operand; `PMULLW` stores the bottom 16 bits of each doubleword in the destination operand.

A.119 `POP`: Pop Data from Stack

```
POP reg16                     ; o16 58+r               [8086]
POP reg32                     ; o32 58+r               [386]

POP r/m16                     ; o16 8F /0              [8086]
POP r/m32                     ; o32 8F /0              [386]

POP DS                        ; 1F                     [8086]
POP ES                        ; 07                     [8086]
POP SS                        ; 17                     [8086]
POP FS                        ; 0F A1                  [386]
POP GS                        ; 0F A9                  [386]
```

`POP` loads a value from the stack (from `[SS:SP]` or `[SS:ESP]`) and then increments the stack pointer.

The address-size attribute of the instruction determines whether `SP` or `ESP` is used as the stack pointer: to deliberately override the default given by the `BITS` setting, you can use an `a16` or `a32` prefix.

The operand-size attribute of the instruction determines whether the stack pointer is incremented by 2 or 4: this means that segment register pops in `BITS 32` mode will pop 4 bytes off the stack and discard the upper two of them. If you need to override that, you can use an `o16` or `o32` prefix.

The above opcode listings give two forms for general-purpose register pop instructions: for example, `POP BX` has the two forms `5B` and `8F C3`. NASM will always generate the shorter form when given `POP BX`.

A.120 `POPAx`: Pop All General-Purpose Registers

```
POPA                          ; 61                     [186]
POPAW                         ; o16 61                 [186]
POPAD                         ; o32 61                 [386]
```

`POPAW` pops a word from the stack into each of, successively, `DI`, `SI`, `BP`, nothing (it discards a word from the stack which was a placeholder for `SP`), `BX`, `DX`, `CX` and `AX`. It is intended to reverse the operation of `PUSHAW` (see Section A.127), but it ignores the value for `SP` that was pushed on the stack by `PUSHAW`.

`POPAD` pops twice as much data, and places the results in `EDI`, `ESI`, `EBP`, nothing (placeholder for `ESP`), `EBX`, `EDX`, `ECX` and `EAX`. It reverses the operation of `PUSHAD`.

`POPA` is an alias mnemonic for either `POPAW` or `POPAD`, depending on the current `BITS` setting.

Note that the registers are popped in reverse order of their numeric values in opcodes (see Section A.2.1).

A.121 POPFx: Pop Flags Register

```
POPF                          ; 9D                    [186]
POPFW                         ; o16 9D                [186]
POPFD                         ; o32 9D                [386]
```

POPFW pops a word from the stack and stores it in the bottom 16 bits of the flags register (or the whole flags register, on processors below a 386). POPFD pops a doubleword and stores it in the entire flags register.

POPF is an alias mnemonic for either POPFW or POPFD, depending on the current BITS setting.

See also PUSHF (Section A.128).

A.122 POR: MMX Bitwise OR

```
POR mmxreg,r/m64              ; 0F EB /r              [PENT,MMX]
```

POR performs a bitwise OR operation between its two operands (i.e. each bit of the result is 1 if and only if at least one of the corresponding bits of the two inputs was 1), and stores the result in the destination (first) operand.

A.123 PSLLx, PSRLx, PSRAx: MMX Bit Shifts

```
PSLLW mmxreg,r/m64            ; 0F F1 /r              [PENT,MMX]
PSLLW mmxreg,imm8             ; 0F 71 /6 ib           [PENT,MMX]

PSLLD mmxreg,r/m64            ; 0F F2 /r              [PENT,MMX]
PSLLD mmxreg,imm8             ; 0F 72 /6 ib           [PENT,MMX]

PSLLQ mmxreg,r/m64            ; 0F F3 /r              [PENT,MMX]
PSLLQ mmxreg,imm8             ; 0F 73 /6 ib           [PENT,MMX]

PSRAW mmxreg,r/m64            ; 0F E1 /r              [PENT,MMX]
PSRAW mmxreg,imm8             ; 0F 71 /4 ib           [PENT,MMX]

PSRAD mmxreg,r/m64            ; 0F E2 /r              [PENT,MMX]
PSRAD mmxreg,imm8             ; 0F 72 /4 ib           [PENT,MMX]

PSRLW mmxreg,r/m64            ; 0F D1 /r              [PENT,MMX]
PSRLW mmxreg,imm8             ; 0F 71 /2 ib           [PENT,MMX]

PSRLD mmxreg,r/m64            ; 0F D2 /r              [PENT,MMX]
PSRLD mmxreg,imm8             ; 0F 72 /2 ib           [PENT,MMX]

PSRLQ mmxreg,r/m64            ; 0F D3 /r              [PENT,MMX]
PSRLQ mmxreg,imm8             ; 0F 73 /2 ib           [PENT,MMX]
```

`PSxxQ` perform simple bit shifts on the 64-bit MMX registers: the destination (first) operand is shifted left or right by the number of bits given in the source (second) operand, and the vacated bits are filled in with zeros (for a logical shift) or copies of the original sign bit (for an arithmetic right shift).

`PSxxW` and `PSxxD` perform packed bit shifts: the destination operand is treated as a vector of four words or two doublewords, and each element is shifted individually, so bits shifted out of one element do not interfere with empty bits coming into the next.

`PSLLx` and `PSRLx` perform logical shifts: the vacated bits at one end of the shifted number are filled with zeros. `PSRAx` performs an arithmetic right shift: the vacated bits at the top of the shifted number are filled with copies of the original top (sign) bit.

A.124 `PSUBxx`: MMX Packed Subtraction

```
PSUBB mmxreg,r/m64            ; 0F F8 /r             [PENT,MMX]
PSUBW mmxreg,r/m64            ; 0F F9 /r             [PENT,MMX]
PSUBD mmxreg,r/m64            ; 0F FA /r             [PENT,MMX]

PSUBSB mmxreg,r/m64           ; 0F E8 /r             [PENT,MMX]
PSUBSW mmxreg,r/m64           ; 0F E9 /r             [PENT,MMX]

PSUBUSB mmxreg,r/m64          ; 0F D8 /r             [PENT,MMX]
PSUBUSW mmxreg,r/m64          ; 0F D9 /r             [PENT,MMX]
```

`PSUBxx` all perform packed subtraction between their two 64-bit operands, storing the result in the destination (first) operand. The `PSUBxB` forms treat the 64-bit operands as vectors of eight bytes, and subtract each byte individually; `PSUBxW` treat the operands as vectors of four words; and `PSUBD` treats its operands as vectors of two doublewords.

In all cases, the elements of the operand on the right are subtracted from the corresponding elements of the operand on the left, not the other way round.

`PSUBSB` and `PSUBSW` perform signed saturation on the sum of each pair of bytes or words: if the result of a subtraction is too large or too small to fit into a signed byte or word result, it is clipped (saturated) to the largest or smallest value which *will* fit. `PSUBUSB` and `PSUBUSW` similarly perform unsigned saturation, clipping to `0FFh` or `0FFFFh` if the result is larger than that.

A.125 `PUNPCKxxx`: Unpack Data

```
PUNPCKHBW mmxreg,r/m64        ; 0F 68 /r             [PENT,MMX]
PUNPCKHWD mmxreg,r/m64        ; 0F 69 /r             [PENT,MMX]
PUNPCKHDQ mmxreg,r/m64        ; 0F 6A /r             [PENT,MMX]

PUNPCKLBW mmxreg,r/m64        ; 0F 60 /r             [PENT,MMX]
PUNPCKLWD mmxreg,r/m64        ; 0F 61 /r             [PENT,MMX]
PUNPCKLDQ mmxreg,r/m64        ; 0F 62 /r             [PENT,MMX]
```

PUNPCKxx all treat their operands as vectors, and produce a new vector generated by interleaving elements from the two inputs. The PUNPCKHxx instructions start by throwing away the bottom half of each input operand, and the PUNPCKLxx instructions throw away the top half.

The remaining elements, totalling 64 bits, are then interleaved into the destination, alternating elements from the second (source) operand and the first (destination) operand: so the leftmost element in the result always comes from the second operand, and the rightmost from the destination.

PUNPCKxBW works a byte at a time, PUNPCKxWD a word at a time, and PUNPCKxDQ a doubleword at a time.

So, for example, if the first operand held 0x7A6A5A4A3A2A1A0A and the second held 0x7B6B5B4B3B2B1B0B, then:

PUNPCKHBW would return 0x7B7A6B6A5B5A4B4A.
PUNPCKHWD would return 0x7B6B7A6A5B4B5A4A.
PUNPCKHDQ would return 0x7B6B5B4B7A6A5A4A.
PUNPCKLBW would return 0x3B3A2B2A1B1A0B0A.
PUNPCKLWD would return 0x3B2B3A2A1B0B1A0A.
PUNPCKLDQ would return 0x3B2B1B0B3A2A1A0A.

A.126 PUSH: Push Data on Stack

```
PUSH reg16                    ; o16 50+r              [8086]
PUSH reg32                    ; o32 50+r              [386]

PUSH r/m16                    ; o16 FF /6             [8086]
PUSH r/m32                    ; o32 FF /6             [386]

PUSH CS                       ; 0E                    [8086]
PUSH DS                       ; 1E                    [8086]
PUSH ES                       ; 06                    [8086]
PUSH SS                       ; 16                    [8086]
PUSH FS                       ; 0F A0                 [386]
PUSH GS                       ; 0F A8                 [386]

PUSH imm8                     ; 6A ib                 [286]
PUSH imm16                    ; o16 68 iw             [286]
PUSH imm32                    ; o32 68 id             [386]
```

PUSH decrements the stack pointer (SP or ESP) by 2 or 4, and then stores the given value at [SS:SP] or [SS:ESP].

The address-size attribute of the instruction determines whether SP or ESP is used as the stack pointer: to deliberately override the default given by the BITS setting, you can use an a16 or a32 prefix.

The operand-size attribute of the instruction determines whether the stack pointer is decremented by 2 or 4: this means that segment register pushes in BITS 32 mode will push 4 bytes on the stack, of which the upper two are undefined. If you need to override that, you can use an o16 or o32 prefix.

The above opcode listings give two forms for general-purpose register push instructions: for example, PUSH BX has the two forms 53 and FF F3. NASM will always generate the shorter form when given PUSH BX.

The instruction `PUSH SP` may be used to distinguish an 8086 from later processors: on an 8086, the value of `SP` stored is the value it has *after* the push instruction, whereas on later processors it is the value *before* the push instruction.

A.127 `PUSHAx`: Push All General-Purpose Registers

```
PUSHA                        ; 60                    [186]
PUSHAD                       ; o32 60                [386]
PUSHAW                       ; o16 60                [186]
```

`PUSHAW` pushes, in succession, `AX`, `CX`, `DX`, `BX`, `SP`, `BP`, `SI` and `DI` on the stack, decrementing the stack pointer by a total of 16.

`PUSHAD` pushes, in succession, `EAX`, `ECX`, `EDX`, `EBX`, `ESP`, `EBP`, `ESI` and `EDI` on the stack, decrementing the stack pointer by a total of 32.

In both cases, the value of `SP` or `ESP` pushed is its *original* value, as it had before the instruction was executed.

`PUSHA` is an alias mnemonic for either `PUSHAW` or `PUSHAD`, depending on the current `BITS` setting.

Note that the registers are pushed in order of their numeric values in opcodes (see Section A.2.1).

See also `POPA` (Section A.120).

A.128 `PUSHFx`: Push Flags Register

```
PUSHF                        ; 9C                    [186]
PUSHFD                       ; o32 9C                [386]
PUSHFW                       ; o16 9C                [186]
```

`PUSHFW` pops a word from the stack and stores it in the bottom 16 bits of the flags register (or the whole flags register, on processors below a 386). `PUSHFD` pops a doubleword and stores it in the entire flags register.

`PUSHF` is an alias mnemonic for either `PUSHFW` or `PUSHFD`, depending on the current `BITS` setting.

See also `POPF` (Section A.121).

A.129 `PXOR`: MMX Bitwise XOR

```
PXOR mmxreg,r/m64            ; 0F EF /r              [PENT,MMX]
```

`PXOR` performs a bitwise XOR operation between its two operands (i.e. each bit of the result is 1 if and only if exactly one of the corresponding bits of the two inputs was 1), and stores the result in the destination (first) operand.

A.130 RCL, RCR: Bitwise Rotate through Carry Bit

```
RCL r/m8,1                    ; D0 /2                [8086]
RCL r/m8,CL                   ; D2 /2                [8086]
RCL r/m8,imm8                 ; C0 /2 ib             [286]
RCL r/m16,1                   ; o16 D1 /2            [8086]
RCL r/m16,CL                  ; o16 D3 /2            [8086]
RCL r/m16,imm8                ; o16 C1 /2 ib         [286]
RCL r/m32,1                   ; o32 D1 /2            [386]
RCL r/m32,CL                  ; o32 D3 /2            [386]
RCL r/m32,imm8                ; o32 C1 /2 ib         [386]

RCR r/m8,1                    ; D0 /3                [8086]
RCR r/m8,CL                   ; D2 /3                [8086]
RCR r/m8,imm8                 ; C0 /3 ib             [286]
RCR r/m16,1                   ; o16 D1 /3            [8086]
RCR r/m16,CL                  ; o16 D3 /3            [8086]
RCR r/m16,imm8                ; o16 C1 /3 ib         [286]
RCR r/m32,1                   ; o32 D1 /3            [386]
RCR r/m32,CL                  ; o32 D3 /3            [386]
RCR r/m32,imm8                ; o32 C1 /3 ib         [386]
```

`RCL` and `RCR` perform a 9-bit, 17-bit or 33-bit bitwise rotation operation, involving the given source/destination (first) operand and the carry bit. Thus, for example, in the operation `RCR AL,1`, a 9-bit rotation is performed in which `AL` is shifted left by 1, the top bit of `AL` moves into the carry flag, and the original value of the carry flag is placed in the low bit of `AL`.

The number of bits to rotate by is given by the second operand. Only the bottom five bits of the rotation count are considered by processors above the 8086.

You can force the longer (286 and upwards, beginning with a `C1` byte) form of `RCL foo,1` by using a `BYTE` prefix: `RCL foo,BYTE 1`. Similarly with `RCR`.

A.131 RDMSR: Read Model-Specific Registers

```
RDMSR                         ; 0F 32                [PENT]
```

`RDMSR` reads the processor Model-Specific Register (MSR) whose index is stored in `ECX`, and stores the result in `EDX:EAX`. See also `WRMSR` (Section A.155).

A.132 RDPMC: Read Performance-Monitoring Counters

```
RDPMC                         ; 0F 33                [P6]
```

`RDPMC` reads the processor performance-monitoring counter whose index is stored in `ECX`, and stores the result in `EDX:EAX`.

A.133 `RDTSC`: Read Time-Stamp Counter

```
RDTSC                         ; 0F 31                [PENT]
```

`RDTSC` reads the processor's time-stamp counter into `EDX:EAX`.

A.134 `RET`, `RETF`, `RETN`: Return from Procedure Call

```
RET                           ; C3                   [8086]
RET imm16                     ; C2 iw                [8086]

RETF                          ; CB                   [8086]
RETF imm16                    ; CA iw                [8086]

RETN                          ; C3                   [8086]
RETN imm16                    ; C2 iw                [8086]
```

`RET`, and its exact synonym `RETN`, pop `IP` or `EIP` from the stack and transfer control to the new address. Optionally, if a numeric second operand is provided, they increment the stack pointer by a further `imm16` bytes after popping the return address.

`RETF` executes a far return: after popping `IP`/`EIP`, it then pops `CS`, and *then* increments the stack pointer by the optional argument if present.

A.135 `ROL`, `ROR`: Bitwise Rotate

```
ROL r/m8,1                    ; D0 /0                [8086]
ROL r/m8,CL                   ; D2 /0                [8086]
ROL r/m8,imm8                 ; C0 /0 ib             [286]
ROL r/m16,1                   ; o16 D1 /0            [8086]
ROL r/m16,CL                  ; o16 D3 /0            [8086]
ROL r/m16,imm8                ; o16 C1 /0 ib         [286]
ROL r/m32,1                   ; o32 D1 /0            [386]
ROL r/m32,CL                  ; o32 D3 /0            [386]
ROL r/m32,imm8                ; o32 C1 /0 ib         [386]

ROR r/m8,1                    ; D0 /1                [8086]
ROR r/m8,CL                   ; D2 /1                [8086]
ROR r/m8,imm8                 ; C0 /1 ib             [286]
ROR r/m16,1                   ; o16 D1 /1            [8086]
ROR r/m16,CL                  ; o16 D3 /1            [8086]
ROR r/m16,imm8                ; o16 C1 /1 ib         [286]
ROR r/m32,1                   ; o32 D1 /1            [386]
ROR r/m32,CL                  ; o32 D3 /1            [386]
ROR r/m32,imm8                ; o32 C1 /1 ib         [386]
```

`ROL` and `ROR` perform a bitwise rotation operation on the given source/destination (first) operand. Thus, for example, in the operation `ROR AL,1`, an 8-bit rotation is performed in which `AL` is shifted left by 1 and the original top bit of `AL` moves round into the low bit.

The number of bits to rotate by is given by the second operand. Only the bottom 3, 4 or 5 bits (depending on the source operand size) of the rotation count are considered by processors above the 8086.

You can force the longer (286 and upwards, beginning with a `C1` byte) form of `ROL foo,1` by using a `BYTE` prefix: `ROL foo,BYTE 1`. Similarly with `ROR`.

A.136 `RSM`: Resume from System-Management Mode

```
RSM                           ; 0F AA                 [PENT]
```

`RSM` returns the processor to its normal operating mode when it was in System-Management Mode.

A.137 `SAHF`: Store AH to Flags

```
SAHF                          ; 9E                    [8086]
```

`SAHF` sets the low byte of the flags word according to the contents of the `AH` register. See also `LAHF` (Section A.91).

A.138 `SAL`, `SAR`: Bitwise Arithmetic Shifts

```
SAL r/m8,1                    ; D0 /4                 [8086]
SAL r/m8,CL                   ; D2 /4                 [8086]
SAL r/m8,imm8                 ; C0 /4 ib              [286]
SAL r/m16,1                   ; o16 D1 /4             [8086]
SAL r/m16,CL                  ; o16 D3 /4             [8086]
SAL r/m16,imm8                ; o16 C1 /4 ib          [286]
SAL r/m32,1                   ; o32 D1 /4             [386]
SAL r/m32,CL                  ; o32 D3 /4             [386]
SAL r/m32,imm8                ; o32 C1 /4 ib          [386]

SAR r/m8,1                    ; D0 /0                 [8086]
SAR r/m8,CL                   ; D2 /0                 [8086]
SAR r/m8,imm8                 ; C0 /0 ib              [286]
SAR r/m16,1                   ; o16 D1 /0             [8086]
SAR r/m16,CL                  ; o16 D3 /0             [8086]
SAR r/m16,imm8                ; o16 C1 /0 ib          [286]
SAR r/m32,1                   ; o32 D1 /0             [386]
SAR r/m32,CL                  ; o32 D3 /0             [386]
SAR r/m32,imm8                ; o32 C1 /0 ib          [386]
```

`SAL` and `SAR` perform an arithmetic shift operation on the given source/destination (first) operand. The vacated bits are filled with zero for `SAL`, and with copies of the original high bit of the source operand for `SAR`.

`SAL` is a synonym for `SHL` (see Section A.144). NASM will assemble either one to the same code, but NDISASM will always disassemble that code as `SHL`.

The number of bits to shift by is given by the second operand. Only the bottom 3, 4 or 5 bits (depending on the source operand size) of the shift count are considered by processors above the 8086.

You can force the longer (286 and upwards, beginning with a `C1` byte) form of `SAL foo,1` by using a `BYTE` prefix: `SAL foo,BYTE 1`. Similarly with `SAR`.

A.139 `SALC`: Set AL from Carry Flag

```
SALC                             ; D6                   [8086,UNDOC]
```

`SALC` is an early undocumented instruction similar in concept to `SETcc` (Section A.142). Its function is to set `AL` to zero if the carry flag is clear, or to `0xFF` if it is set.

A.140 `SBB`: Subtract with Borrow

```
SBB r/m8,reg8                    ; 18 /r                [8086]
SBB r/m16,reg16                  ; o16 19 /r            [8086]
SBB r/m32,reg32                  ; o32 19 /r            [386]

SBB reg8,r/m8                    ; 1A /r                [8086]
SBB reg16,r/m16                  ; o16 1B /r            [8086]
SBB reg32,r/m32                  ; o32 1B /r            [386]

SBB r/m8,imm8                    ; 80 /3 ib             [8086]
SBB r/m16,imm16                  ; o16 81 /3 iw         [8086]
SBB r/m32,imm32                  ; o32 81 /3 id         [386]

SBB r/m16,imm8                   ; o16 83 /3 ib         [8086]
SBB r/m32,imm8                   ; o32 83 /3 ib         [8086]

SBB AL,imm8                      ; 1C ib                [8086]
SBB AX,imm16                     ; o16 1D iw            [8086]
SBB EAX,imm32                    ; o32 1D id            [386]
```

`SBB` performs integer subtraction: it subtracts its second operand, plus the value of the carry flag, from its first, and leaves the result in its destination (first) operand. The flags are set according to the result of the operation: in particular, the carry flag is affected and can be used by a subsequent `SBB` instruction.

In the forms with an 8-bit immediate second operand and a longer first operand, the second operand is considered to be signed, and is sign-extended to the length of the first operand. In these cases, the `BYTE` qualifier is necessary to force NASM to generate this form of the instruction.

To subtract one number from another without also subtracting the contents of the carry flag, use `SUB` (Section A.150).

A.141 `SCASB`, `SCASW`, `SCASD`: Scan String

```
SCASB                         ; AE                  [8086]
SCASW                         ; o16 AF              [8086]
SCASD                         ; o32 AF              [386]
```

`SCASB` compares the byte in `AL` with the byte at `[ES:DI]` or `[ES:EDI]`, and sets the flags accordingly. It then increments or decrements (depending on the direction flag: increments if the flag is clear, decrements if it is set) `DI` (or `EDI`).

The register used is `DI` if the address size is 16 bits, and `EDI` if it is 32 bits. If you need to use an address size not equal to the current `BITS` setting, you can use an explicit `a16` or `a32` prefix.

Segment override prefixes have no effect for this instruction: the use of `ES` for the load from `[DI]` or `[EDI]` cannot be overridden.

`SCASW` and `SCASD` work in the same way, but they compare a word to `AX` or a doubleword to `EAX` instead of a byte to `AL`, and increment or decrement the addressing registers by 2 or 4 instead of 1.

The `REPE` and `REPNE` prefixes (equivalently, `REPZ` and `REPNZ`) may be used to repeat the instruction up to `CX` (or `ECX` - again, the address size chooses which) times until the first unequal or equal byte is found.

A.142 `SETcc`: Set Register from Condition

```
SETcc r/m8                    ; 0F 90+cc /2         [386]
```

`SETcc` sets the given 8-bit operand to zero if its condition is not satisfied, and to 1 if it is.

A.143 `SGDT`, `SIDT`, `SLDT`: Store Descriptor Table Pointers

```
SGDT mem                      ; 0F 01 /0            [286,PRIV]
SIDT mem                      ; 0F 01 /1            [286,PRIV]
SLDT r/m16                    ; 0F 00 /0            [286,PRIV]
```

`SGDT` and `SIDT` both take a 6-byte memory area as an operand: they store the contents of the GDTR (global descriptor table register) or IDTR (interrupt descriptor table register) into that area as a 32-bit linear address and a 16-bit size limit from that area (in that order). These are the only instructions which directly use *linear* addresses, rather than segment/offset pairs.

`SLDT` stores the segment selector corresponding to the LDT (local descriptor table) into the given operand.

See also `LGDT`, `LIDT` and `LLDT` (Section A.96).

A.144 `SHL`, `SHR`: Bitwise Logical Shifts

```
SHL r/m8,1                    ; D0 /4               [8086]
SHL r/m8,CL                   ; D2 /4               [8086]
SHL r/m8,imm8                 ; C0 /4 ib            [286]
SHL r/m16,1                     ; o16 D1 /4              [8086]
SHL r/m16,CL                    ; o16 D3 /4              [8086]
SHL r/m16,imm8                  ; o16 C1 /4 ib           [286]
SHL r/m32,1                     ; o32 D1 /4              [386]
SHL r/m32,CL                    ; o32 D3 /4              [386]
SHL r/m32,imm8                  ; o32 C1 /4 ib           [386]

SHR r/m8,1                      ; D0 /5                  [8086]
SHR r/m8,CL                     ; D2 /5                  [8086]
SHR r/m8,imm8                   ; C0 /5 ib               [286]
SHR r/m16,1                     ; o16 D1 /5              [8086]
SHR r/m16,CL                    ; o16 D3 /5              [8086]
SHR r/m16,imm8                  ; o16 C1 /5 ib           [286]
SHR r/m32,1                     ; o32 D1 /5              [386]
SHR r/m32,CL                    ; o32 D3 /5              [386]
SHR r/m32,imm8                  ; o32 C1 /5 ib           [386]
```

`SHL` and `SHR` perform a logical shift operation on the given source/destination (first) operand. The vacated bits are filled with zero.

A synonym for `SHL` is `SAL` (see Section A.138). NASM will assemble either one to the same code.

The number of bits to shift by is given by the second operand. Only the bottom 3, 4 or 5 bits (depending on the source operand size) of the shift count are considered by processors above the 8086.

You can force the longer (286 and upwards, beginning with a `C1` byte) form of `SHL foo,1` by using a `BYTE` prefix: `SHL foo,BYTE 1`. Similarly with `SHR`.

A.145 `SHLD`, `SHRD`: Bitwise Double-Precision Shifts

```
SHLD r/m16,reg16,imm8           ; o16 0F A4 /r ib        [386]
SHLD r/m16,reg32,imm8           ; o32 0F A4 /r ib        [386]
SHLD r/m16,reg16,CL             ; o16 0F A5 /r           [386]
SHLD r/m16,reg32,CL             ; o32 0F A5 /r           [386]

SHRD r/m16,reg16,imm8           ; o16 0F AC /r ib        [386]
SHRD r/m32,reg32,imm8           ; o32 0F AC /r ib        [386]
SHRD r/m16,reg16,CL             ; o16 0F AD /r           [386]
SHRD r/m32,reg32,CL             ; o32 0F AD /r           [386]
```

`SHLD` performs a double-precision left shift. It notionally places its second operand to the right of its first, then shifts the entire bit string thus generated to the left by a number of bits specified in the third operand. It then updates only the *first* operand according to the result of this. The second operand is not modified.

`SHRD` performs the corresponding right shift: it notionally places the second operand to the *left* of the first, shifts the whole bit string right, and updates only the first operand.

For example, if `EAX` holds `0x01234567` and `EBX` holds `0x89ABCDEF`, then the instruction `SHLD EAX,EBX,4` would update `EAX` to hold `0x12345678`. Under the same conditions, `SHRD EAX,EBX,4` would update `EAX` to hold `0xF0123456`.

The number of bits to shift by is given by the third operand. Only the bottom 5 bits of the shift count are considered.

A.146 `SMSW`: Store Machine Status Word

```
SMSW r/m16                    ; 0F 01 /4             [286,PRIV]
```

`SMSW` stores the bottom half of the `CR0` control register (or the Machine Status Word, on 286 processors) into the destination operand. See also `LMSW` (Section A.97).

A.147 `STC`, `STD`, `STI`: Set Flags

```
STC                           ; F9                   [8086]
STD                           ; FD                   [8086]
STI                           ; FB                   [8086]
```

These instructions set various flags. `STC` sets the carry flag; `STD` sets the direction flag; and `STI` sets the interrupt flag (thus enabling interrupts).

To clear the carry, direction, or interrupt flags, use the `CLC`, `CLD` and `CLI` instructions (Section A.15). To invert the carry flag, use `CMC` (Section A.16).

A.148 `STOSB`, `STOSW`, `STOSD`: Store Byte to String

```
STOSB                         ; AA                   [8086]
STOSW                         ; o16 AB               [8086]
STOSD                         ; o32 AB               [386]
```

`STOSB` stores the byte in `AL` at `[ES:DI]` or `[ES:EDI]`, and sets the flags accordingly. It then increments or decrements (depending on the direction flag: increments if the flag is clear, decrements if it is set) `DI` (or `EDI`).

The register used is `DI` if the address size is 16 bits, and `EDI` if it is 32 bits. If you need to use an address size not equal to the current `BITS` setting, you can use an explicit `a16` or `a32` prefix.

Segment override prefixes have no effect for this instruction: the use of `ES` for the store to `[DI]` or `[EDI]` cannot be overridden.

`STOSW` and `STOSD` work in the same way, but they store the word in `AX` or the doubleword in `EAX` instead of the byte in `AL`, and increment or decrement the addressing registers by 2 or 4 instead of 1.

The `REP` prefix may be used to repeat the instruction `CX` (or `ECX` - again, the address size chooses which) times.

A.149 `STR`: Store Task Register

```
STR r/m16                     ; 0F 00 /1             [286,PRIV]
```

`STR` stores the segment selector corresponding to the contents of the Task Register into its operand.

A.150 SUB: Subtract Integers

```
SUB r/m8,reg8                  ; 28 /r                  [8086]
SUB r/m16,reg16                ; o16 29 /r              [8086]
SUB r/m32,reg32                ; o32 29 /r              [386]

SUB reg8,r/m8                  ; 2A /r                  [8086]
SUB reg16,r/m16                ; o16 2B /r              [8086]
SUB reg32,r/m32                ; o32 2B /r              [386]

SUB r/m8,imm8                  ; 80 /5 ib               [8086]
SUB r/m16,imm16                ; o16 81 /5 iw           [8086]
SUB r/m32,imm32                ; o32 81 /5 id           [386]

SUB r/m16,imm8                 ; o16 83 /5 ib           [8086]
SUB r/m32,imm8                 ; o32 83 /5 ib           [386]

SUB AL,imm8                    ; 2C ib                  [8086]
SUB AX,imm16                   ; o16 2D iw              [8086]
SUB EAX,imm32                  ; o32 2D id              [386]
```

`SUB` performs integer subtraction: it subtracts its second operand from its first, and leaves the result in its destination (first) operand. The flags are set according to the result of the operation: in particular, the carry flag is affected and can be used by a subsequent `SBB` instruction (Section A.140).

In the forms with an 8-bit immediate second operand and a longer first operand, the second operand is considered to be signed, and is sign-extended to the length of the first operand. In these cases, the `BYTE` qualifier is necessary to force NASM to generate this form of the instruction.

A.151 TEST: Test Bits (notional bitwise AND)

```
TEST r/m8,reg8                 ; 84 /r                  [8086]
TEST r/m16,reg16               ; o16 85 /r              [8086]
TEST r/m32,reg32               ; o32 85 /r              [386]

TEST r/m8,imm8                 ; F6 /7 ib               [8086]
TEST r/m16,imm16               ; o16 F7 /7 iw           [8086]
TEST r/m32,imm32               ; o32 F7 /7 id           [386]

TEST AL,imm8                   ; A8 ib                  [8086]
TEST AX,imm16                  ; o16 A9 iw              [8086]
TEST EAX,imm32                 ; o32 A9 id              [386]
```

`TEST` performs a “mental” bitwise AND of its two operands, and affects the flags as if the operation had taken place, but does not store the result of the operation anywhere.

A.152 `VERR`, `VERW`: Verify Segment Readability/Writability

```
VERR r/m16                    ; 0F 00 /4             [286,PRIV]

VERW r/m16                    ; 0F 00 /5             [286,PRIV]
```

`VERR` sets the zero flag if the segment specified by the selector in its operand can be read from at the current privilege level. `VERW` sets the zero flag if the segment can be written.

A.153 `WAIT`: Wait for Floating-Point Processor

```
WAIT                          ; 9B                   [8086]
```

`WAIT`, on 8086 systems with a separate 8087 FPU, waits for the FPU to have finished any operation it is engaged in before continuing main processor operations, so that (for example) an FPU store to main memory can be guaranteed to have completed before the CPU tries to read the result back out.

On higher processors, `WAIT` is unnecessary for this purpose, and it has the alternative purpose of ensuring that any pending unmasked FPU exceptions have happened before execution continues.

A.154 `WBINVD`: Write Back and Invalidate Cache

```
WBINVD                        ; 0F 09                [486]
```

`WBINVD` invalidates and empties the processor's internal caches, and causes the processor to instruct external caches to do the same. It writes the contents of the caches back to memory first, so no data is lost. To flush the caches quickly without bothering to write the data back first, use `INVD` (Section A.85).

A.155 `WRMSR`: Write Model-Specific Registers

```
WRMSR                         ; 0F 30                [PENT]
```

`WRMSR` writes the value in `EDX:EAX` to the processor Model-Specific Register (MSR) whose index is stored in `ECX`. See also `RDMSR` (Section A.131).

A.156 `XADD`: Exchange and Add

```
XADD r/m8,reg8                ; 0F C0 /r             [486]
XADD r/m16,reg16              ; o16 0F C1 /r         [486]
XADD r/m32,reg32              ; o32 0F C1 /r         [486]
```

`XADD` exchanges the values in its two operands, and then adds them together and writes the result into the destination (first) operand. This instruction can be used with a `LOCK` prefix for multi-processor synchronisation purposes.

A.157 XCHG: Exchange

```
XCHG reg8,r/m8                ; 86 /r                 [8086]
XCHG reg16,r/m8               ; o16 87 /r             [8086]
XCHG reg32,r/m32              ; o32 87 /r             [386]

XCHG r/m8,reg8                ; 86 /r                 [8086]
XCHG r/m16,reg16              ; o16 87 /r             [8086]
XCHG r/m32,reg32              ; o32 87 /r             [386]

XCHG AX,reg16                 ; o16 90+r              [8086]
XCHG EAX,reg32                ; o32 90+r              [386]
XCHG reg16,AX                 ; o16 90+r              [8086]
XCHG reg32,EAX                ; o32 90+r              [386]
```

`XCHG` exchanges the values in its two operands. It can be used with a `LOCK` prefix for purposes of multi-processor synchronisation.

`XCHG AX,AX` or `XCHG EAX,EAX` (depending on the `BITS` setting) generates the opcode `90h`, and so is a synonym for `NOP` (Section A.109).

A.158 XLATB: Translate Byte in Lookup Table

```
XLATB                         ; D7                    [8086]
```

`XLATB` adds the value in `AL`, treated as an unsigned byte, to `BX` or `EBX`, and loads the byte from the resulting address (in the segment specified by `DS`) back into `AL`.

The base register used is `BX` if the address size is 16 bits, and `EBX` if it is 32 bits. If you need to use an address size not equal to the current `BITS` setting, you can use an explicit `a16` or `a32` prefix.

The segment register used to load from `[BX+AL]` or `[EBX+AL]` can be overridden by using a segment register name as a prefix (for example, `es xlatb`).

A.159 XOR: Bitwise Exclusive OR

```
XOR r/m8,reg8                 ; 30 /r                 [8086]
XOR r/m16,reg16               ; o16 31 /r             [8086]
XOR r/m32,reg32               ; o32 31 /r             [386]

XOR reg8,r/m8                 ; 32 /r                 [8086]
XOR reg16,r/m16               ; o16 33 /r             [8086]
XOR reg32,r/m32               ; o32 33 /r             [386]

XOR r/m8,imm8                 ; 80 /6 ib              [8086]
XOR r/m16,imm16               ; o16 81 /6 iw          [8086]
XOR r/m32,imm32               ; o32 81 /6 id          [386]

XOR r/m16,imm8                ; o16 83 /6 ib          [8086]
XOR r/m32,imm8                ; o32 83 /6 ib          [386]
```

```
    XOR AL,imm8                    ; 34 ib              [8086]
    XOR AX,imm16                   ; o16 35 iw          [8086]
    XOR EAX,imm32                  ; o32 35 id          [386]
```

`XOR` performs a bitwise XOR operation between its two operands (i.e. each bit of the result is 1 if and only if exactly one of the corresponding bits of the two inputs was 1), and stores the result in the destination (first) operand.

In the forms with an 8-bit immediate second operand and a longer first operand, the second operand is considered to be signed, and is sign-extended to the length of the first operand. In these cases, the `BYTE` qualifier is necessary to force NASM to generate this form of the instruction.

The MMX instruction `PXOR` (see Section A.129) performs the same operation on the 64-bit MMX registers.

Appendix B.
ASCII Code Tables

B.1 Character Set Quick Reference (Hex)

To use this reference, form the two-digit hex code by finding the most significant digit in the horizontal row at the top, and the least significant digit in the vertical column to the side.

	0	1	2	3	4	5	6	7
0	Blank (Null)	►	Blank (Space)	0	@	P	′	p
1	☺	◄	!	1	A	Q	a	q
2	☻	↕	"	2	B	R	b	r
3	♥	‼	#	3	C	S	c	s
4	♦	¶	$	4	D	T	d	t
5	♣	§	%	5	E	U	e	u
6	♠	▬	&	6	F	V	f	v
7	•	↨	’	7	G	W	g	w
8	◘	↑	(	8	H	X	h	x
9	○	↓	)	9	I	Y	i	y
A	◙	→	*	:	J	Z	j	z
B	♂	←	+	;	K	[	k	{
C	♀	∟	,	<	L	\	l	|
D	♪	↔	-	=	M	]	m	}
E	♫	▲	.	>	N	^	n	~
F	☼	▼	/	?	O	_	o	Δ

	8	9	A	B	C	D	E	F
0	Ç	É	á	░	└	╨	α	≡
1	ü	æ	í	▒	┴	╤	β	±
2	é	Æ	ó	▓	┬	╥	Γ	≥
3	â	ô	ú	│	├	╙	π	≤
4	ä	ö	ñ	┤	─	╘	Σ	⌠
5	à	ò	Ñ	╡	┼	╒	σ	⌡
6	å	û	ª	╢	╞	╓	µ	÷
7	ç	ù	º	╖	╟	╫	τ	≈
8	ê	ÿ	¿	╕	╚	╪	Φ	°
9	ë	Ö	⌐	╣	╔	┘	Θ	•
A	è	Ü	¬	║	╩	┌	Ω	·
B	ï	¢	½	╗	╦	█	δ	√
C	î	£	¼	╝	╠	▄	∞	ⁿ
D	ì	¥	¡	╜	═	▌	∅	²
E	Ä	₧	«	╛	╬	▐	∈	■
F	Å	ƒ	»	┐	╧	▀	∩	Blank ("FF")

B.2 ASCII Codes in Hex, Octal, and Decimal

Char		Hex	Oct	Dec
NUL	^@	00	000	0
SOH	^A	01	001	1
STX	^B	02	002	2
ETX	^C	03	003	3
EOT	^D	04	004	4
ENQ	^E	05	005	5
ACK	^F	06	006	6
BEL	^G	07	007	7
BS	^H	08	010	8
TAB	^I	09	011	9
LF	^J	0A	012	10
VT	^K	0B	013	11
FF	^L	0C	014	12
CR	^M	0D	015	13
SO	^N	0E	016	14
SI	^O	0F	017	15
DLE	^P	10	020	16
DC1	^Q	11	021	17
DC2	^R	12	022	18
DC3	^S	13	023	19
DC4	^T	14	024	20
NAK	^U	15	025	21
SYN	^V	16	026	22
ETB	^W	17	027	23
CAN	^X	18	030	24
EM	^Y	19	031	25
SUB	^Z	1A	032	26
ESC	^[	1B	033	27
FS left		1C	034	28
GS right		1D	035	29
RS up		1E	036	30
US down		1F	037	31
Space		20	040	32
!		21	041	33
"		22	042	34
#		23	043	35
$		24	044	36
%		25	045	37

Char	**Hex**	**Oct**	**Dec**
&	26	046	38
’ apostr.	27	047	39
(	28	050	40
)	29	051	41
*	2A	052	42
+	2B	053	43
, comma	2C	054	44
- dash	2D	055	45
. period	2E	056	46
/	2F	057	47
0	30	060	48
1	31	061	49
2	32	062	50
3	33	063	51
4	34	064	52
5	35	065	53
6	36	066	54
7	37	067	55
8	38	070	56
9	39	071	57
:	3A	072	58
;	3B	073	59
$<$	3C	074	60
=	3D	075	61
$>$	3E	076	62
?	3F	077	63
@	40	100	64
A	41	101	65
B	42	102	66
C	43	103	67
D	44	104	68
E	45	105	69
F	46	106	70
G	47	107	71
H	48	110	72
I	49	111	73
J	4A	112	74
K	4B	113	75
L	4C	114	76
M	4D	115	77

Char	**Hex**	**Oct**	**Dec**
N	4E	116	78
O	4F	117	79
P	50	120	80
Q	51	121	81
R	52	122	82
S	53	123	83
T	54	124	84
U	55	125	85
V	56	126	86
W	57	127	87
X	58	130	88
Y	59	131	89
Z	5A	132	90
[	5B	133	91
\	5C	134	92
]	5D	135	93
^ caret	5E	136	94
_ underln	5F	137	95
‘	60	140	96
a	61	141	97
b	62	142	98
c	63	143	99
d	64	144	100
e	65	145	101
f	66	146	102
g	67	147	103
h	68	150	104
i	69	151	105
j	6A	152	106
k	6B	153	107
l	6C	154	108
m	6D	155	109
n	6E	156	110
o	6F	157	111
p	70	160	112
q	71	161	113
r	72	162	114
s	73	163	115
t	74	164	116
u	75	165	117

Char	**Hex**	**Oct**	**Dec**
v	76	166	118
w	77	167	119
x	78	170	120
y	79	171	121
z	7A	172	122
{	7B	173	123
| pipe	7C	174	124
}	7D	175	125
~ tilde	7E	176	126
DELete	7F	177	127

Glossary

Several words and phrases will be used often in ECE 291 and throughout this manual. You should learn their meanings. Only partial definitions are given here. A complete discussion of some words would require several pages of explanation.

A

American Standard Code for Information Interchange

This is the set of 8-bit codes (including a parity bit) assigned to all the keys on the keyboard, including upper/lower case, punctuation and other symbols, and special communication control characters. Within your program, characters received from the keyboard or sent to the display are encoded in ASCII, even numerical digits such as 5, whose ASCII code is 35 (hex). Note, however, that the direct signals between the PC and the keyboard or the display are generally encoded in non-ASCII forms special to their use. See the ASCII tables for a complete listing of ASCII codes.

Assembly

The process of translating an assembly language program into machine code so it can be executed by the PC. This is done by an assembler (in our case, the Netwide Assembler, NASM). The file generated by this process is called an object file, and has the extension `.OBJ`.

Assembly Language

The language used for writing programs for the PCs in ECE 291. Assembly language is a low level language, just one step up from the computer's native language, machine code.

D

Directory

A logical grouping of files on a disk for the purpose of organizing files. Each directory can contain files and/or other directories, so a hierarchy of files and directories can be created.

Diskette

Usually a 3.5 inch removable media. A double-sided, high density diskette can hold about 1.44 MB of information in magnetic patterns on the surface of the disk. (MB stands for MegaByte, which is approximately 1 million bytes). Each PC in the lab has one 3.5 inch disk drive. Historical note: These used to be called floppy disks, because of the earlier 5¼ inch diskettes, which had floppy paper casings instead of hard plastic.

E

Editing

The process of creating a file or changing the contents of a file. There are several text editors available in the lab, including `gvim`, `EMACS`, and `EDIT`.

Execution Time

The time at which the linked-together code is actually being executed by the computer (as opposed to assembly time, when the program is being translated into machine code).

F

File

A unit of information stored on a disk. Each file has a specification of the form *`filename.extension`*, where the *`filename`* identifies the file, and the 3-letter *`extension`* identifies the file type. Some standard extensions are:

.ASM	assembler source file
.BAT	DOS batch file
.COM	executable program file
.EXE	executable program file
.LIB	library file
.LST	list file
.OBJ	object file

L

Listing File

An optional output file from the assembly process that shows how the assembly language program has been translated into object code. A listing file has the extension `.LST`.

Linking

The process of combining (i.e., linking together) the object code modules (object files) of previously assembled program elements into a complete, executable program. The elements linked together might consist of your program and library subroutines. The final executable file has the extension `.EXE` or `.COM`; it can be loaded into the memory of the computer and executed.

O

Object File

The file produced by an assembler in the translation of a source file. The object file contains the binary encoding of the instructions and information about global symbols. It must be combined with other object code modules to form executable code. Object files have the extension `.OBJ`.

Operating System

The control program that supervises and controlsthe operations of the computer system. The operating system in the lab is Windows 2000, but most of what we'll be using in ECE 291 is a precursor to Windows called DOS (Disk Operating System) that is emulated by Windows 2000 in the Command Prompt.

P

Printer

A device which prints text and/or graphics on paper. You can use the printers in the lab to produce hard copies of your programs and other files.

Prompt

A message displayed at the beginning of a line by a program to request a response from the user. DOS prompts for commands with the current disk drive and directory name followed by an angle bracket (e.g., `D:\MYFILES`⟩).

R

Run Time

See: Execution Time

S

Source File

A file which contains a program written in assembly language. It is an input to the assembler. An assembly language source file usually has the extension `.ASM`.

Index