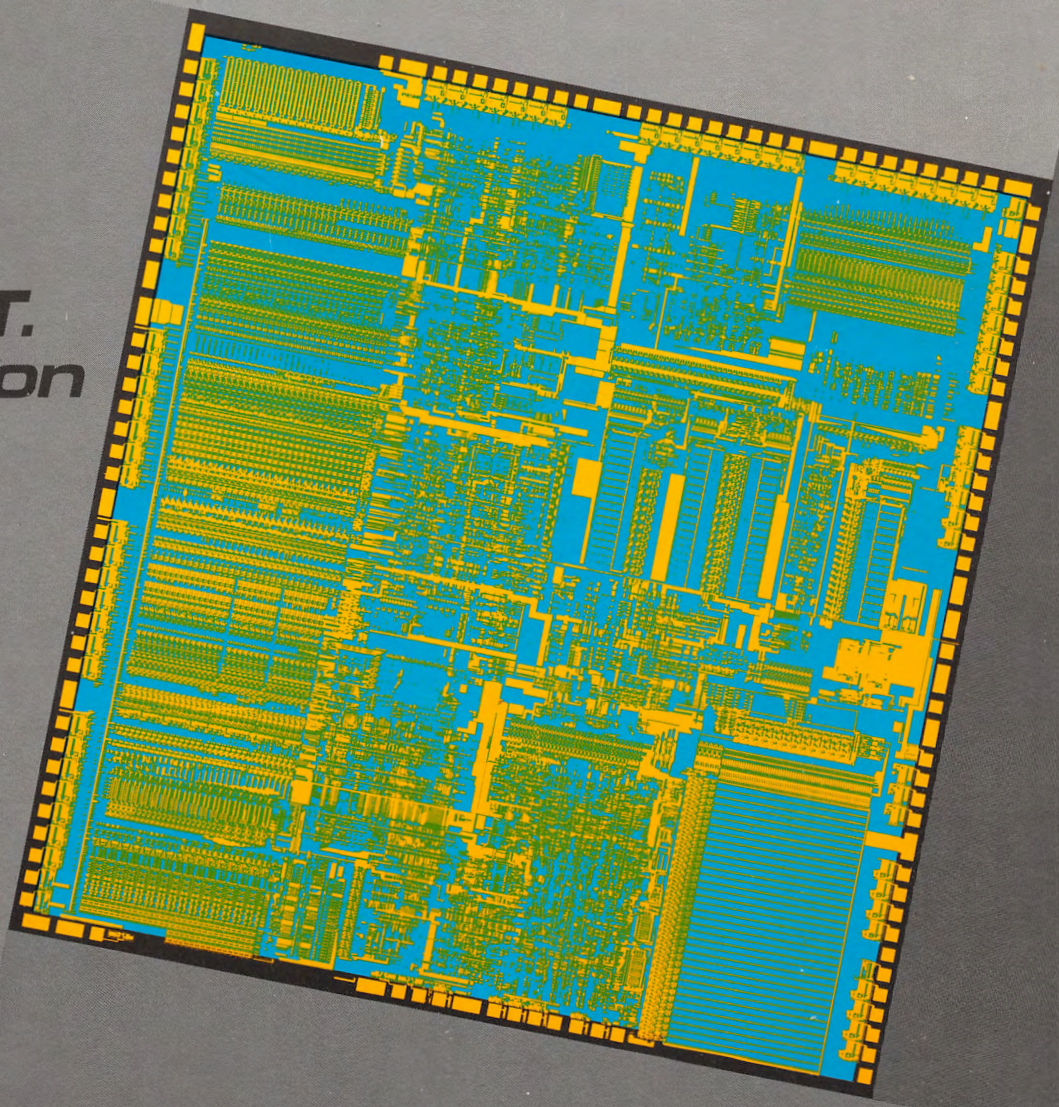
Programming the Intel 80386

Bud E. Smith

Mark T. Johnson



SCOTT, FORESMAN AND COMPANY COMPUTER BOOKS

PROGRAMMING THE INTEL 80386

PROGRAMMING THE INTEL 80386

Bud E. Smith
Mark T. Johnson

Scott, Foresman and Company
Glenview, Illinois London

ISBN 0-673-18568-0

Copyright © 1987 Bud E. Smith.

All Rights Reserved.

Printed in the United States of America.

Library of Congress Cataloging-in-Publication Data

Smith, Bud E.
Programming the Intel 80386.

Includes index.
1. Intel 80386 (Microprocessor)—Programming.
I. Johnson, Mark T. II. Title.
QA76.8.I2928S65 1987 005.265 86-29862
ISBN 0-673-18568-0

1 2 3 4 5 6 MVN 91 90 89 88 87 86

NOTICE OF LIABILITY

The information in this book is distributed on an "As Is" basis, without warranty. Neither the author nor Scott, Foresman and Company shall have any liability, loss, or damage caused or alleged to be caused directly or indirectly by the programs contained herein. This includes, but is not limited to, interruption of service, loss of data, loss of business or anticipatory profits, or consequential damages from the use of the programs.

Scott, Foresman Professional Publishing Group books are available for bulk sales at quantity discounts. For information, please contact Marketing Manager, Professional Books, Professional Publishing Group, Scott, Foresman and Company, 1900 East Lake Avenue, Glenview, IL 60025.

Dedication

Dedicated to all those who are right now burning the midnight oil trying to make something "insanely great" happen with a pile of quirky transistors in silicon, and to their long-suffering friends and families.

Acknowledgments

Our first debt is to Jonathon Shiell of Carlton G. Amdahl Associates, Inc., in Santa Clara, California. He has put many hours into suggestions, comments, and criticisms. He also contributed notes for what became Chapter 7, and gave us early access to his excellent series of articles for *Byte* magazine and the research behind them. Beyond this, he has also marshalled an impressive array of friends and confreres in Silicon Valley who have interrupted important and pressure-filled work to contribute. The level of advice would normally be expensive to get; the level of energy and committment given to turning a pedestrian work into something to be proud of are priceless.

Ken Williams, Vice President for Research and Development at Softguard Systems, was the single main source in keeping us relentlessly up-to-date with the latest developments. Many others at Softguard have also provided assistance at a time when all of them needed R & R, not more 386-oriented work.

Dick Balluf of DB Micro offered a software developer's viewpoint of what's needed in a book of this nature.

Andrew J. Piziali looked past the technical details to help correct our all-too-frequent abuse of that most important programming language, English.

Bill Gladstone and Tershia D'Elgin of Waterside Productions put the wheels back on when they were about to fall off, introduced the principals to one another and kept on top of schedules and financial commitments.

Richard Swadley at Scott, Foresman had patience with a project that went slowly for all too long.

Bob Caldwell of Centaurus Software and Roger Ogden gave hardware support for the writing process.

Contents

Introduction

Welcome to *Programming the Intel 80386*, the most detailed yet easy to understand book to date on the capabilities of the Intel 80386 microprocessor. In this book, we hope to reach several audiences:

The experienced 8086/8088 programmer who wants or needs to understand the capabilities of the 80386. This book delves into the low-level and operating system considerations which give necessary background for designing and implementing serious programming projects.

The programmer experienced with other microprocessors (68000, 6502, Z80, etc.) who wants to step into the 8086 family. This book not only gives detailed coverage of the instruction set, it explains how addressing, protection, and other important features of the 80386 work. It does not assume prior experience with previous Intel microprocessors.

The high-level language programmer who wants to learn assembly language. This book includes an introduction to numbering systems and other basic topics. This reader, however, will also need programming examples, either in book form or from a fellow programmer, to understand how assembly language is used.

The person with good computer knowledge who wants to be an intelligent computer and software buyer. Besides describing the processor itself, this book talks about different memory arrangements and possible operating system capabilities. Someone buying an 80386-based computer would do well to know what the processor can handle as background for judging the products that use it.

This book is not designed to be a complete source for those who are writing systems programs like operating systems, device drivers, etc; no single book can do that. Reading this book would be an excellent first step, however, for writing these complex programs. This book is not for the person new to programming, although the first chapter and parts of other chapters are accessible to the general reader.

Given the variety of material presented here, you may want to read it out of order, or skim some parts as you go through it the first time; it's hard to imagine anyone reading the instruction set in Chapter 4 straight through. Even if you are normally very careful with your books, this might be a good one to write in. Put question marks next to new subjects you want to explore more, and check off sections as you read them. For anyone who's not already an expert there will be much here to learn and to research further later on.

HIGHLIGHTS OF THE 80386

This book looks at the 80386 on its own terms, not just as a derivative of the 8086/88 and the 80286 (we will refer to the 8086 and the 8088 together as simply "the 8086" unless otherwise specified). In fact, much of the text deals with ideas that have come to the microprocessor world from minicomputers and mainframes. Capabilities like paging, virtual memory, and multitasking are discussed at some length. These are features which are made available by operating systems; the experienced programmer of IBM PC-type computers will be eager to know what advantages the 80386 offers the applications programmer. Each of these advantages is discussed at length, with terms defined where needed. Here's a short list of important new capabilities available in Real Mode, based on the experience of a friend who's writing systems software for the new processor:

SPEED. Simply by running at a clock speed of 16 MHz, the 80386 is twice as fast as most 80286s. Lower clock counts and new instructions give a further 50% speed increase. Thus, machines built around the new processor run about three times as fast as an IBM PC AT. The use of cache memories and other memory optimizations can increase performance even more.

32-BIT REGISTERS. Larger registers make it very easy to handle large numbers and other data items efficiently. They also make it possible to do more work with register variables and cut down memory accesses.

NEW INSTRUCTIONS. Among the most important of the many new instructions offered by the 80386 are the Bit Test instructions, which allow testing and turning on and off of individual bits.

EXTRA SEGMENT REGISTERS. The new FS and GS registers are offered as segment registers for additional data segments. They can also

be used as general registers (up to a point), and are very useful in either capacity.

NEW DEBUG REGISTERS. The 80386 has registers which make it easy to put in code or even data breakpoints, and to single-step a program. These registers can be used either from within a program or by a debugger.

SCALED INDEXING. Memory addresses which use indexing can now be scaled by a factor of 2, 4, or 8 bytes, making certain data structures very easy and fast to access.

IMMEDIATE SHIFTS AND ROTATES. This capability, which first appeared on the 80286 and continues with the 80386, makes many applications problems easier to solve. The number of bits by which to shift or rotate a quantity can now be given as an immediate value in the code itself, avoiding extra register usage.

FLAT MEMORY MODEL. The old 64 Kb limit on segment size is now gone. The new limit is 4 Gb, making it easy for even a very large program to have just one code segment, one stack segment, and one or more data segments, each with huge data structures not broken into segments. However, unlike the advantages listed above, this one is only available in Protected Mode.

OVERVIEW

This book is somewhat unusual for a work about microprocessors. The 80386 has many capabilities similar to those of a minicomputer or even (in some cases) a mainframe. As an example, the 80386 can execute up to 4 million instructions per second (mips). It was only ten years ago that the first mainframe to attain a speed of 1 mips became available. Thus all the work that was done for optimizing those computers—support hardware, efficient and capable operating systems, and more—applies directly to the 80386. The designers at Intel have been very conscious of this previous work, building capabilities like paging and virtual memory support right into the chip for the first time in any microprocessor. In order to describe the capabilities of the new chip, much space is devoted to paging, segmentation, and other advanced topics not usually covered in such a "chip book."

This book is called *Programming the Intel 80386,* and it includes a complete description of the instruction set as used for non-systems pro-

grams, with a summary of the different classes of instructions and examples of how each works. However, it does not include large-scale programming examples or a systematic treatment of optimization techniques. There is so much ground to cover in describing the processor itself and the operations it can perform that large application examples are left for future books, including one from this pair of authors. These future programming-examples books will generally assume that you've read a book like this one as preparation for plunging into the details of actual applications.

Because there are not yet an operating system and computer which really stretch the power of the 80386, successive topics within a chapter may seem somewhat unrelated. This is because the system software and large applications which will relate them, tying the capabilities together into new and powerful programs, are not yet written. The information in this book will give the programmer the vocabulary and insight he or she will need in order to design and implement the programs which will be demanded in the next few years. As you use an 80386-based system with a suitable operating system, the capabilities described here will come alive for you and be reflected in the programs you write.

This book is not only broad in its coverage of important topics, it is also deep. As authors we have been surprised to find out how little most programmers know about how a microprocessor really works. For instance, when we say an instruction takes two clocks to execute, how does this relate to what's going on inside the chip itself? Chapter 7 explains the operations of the 80386 and the units which make it up. It describes techniques for memory access and follows a short programming example as it is executed. We hope that even those who mostly program some other microprocessor will find this chapter interesting, and that it will give programmers a better understanding of what they're actually causing to happen as they do their work.

CONTEXT

A book like this aims for a certain timelessness; it should be as useful ten years after it was published as when it first came out. However, this work is being published early enough in the life cycle of the 80386 that some background should be given. In early 1987, as this is published, the 80386 itself is in version B0. It is available in 12.5 MHz and 16 MHz

versions, but the 16 MHz version seems to be the choice of system developers.

The processor is being produced using a 1.5 micron feature size, which means that the highest resolution possible in placing circuit elements is 1.5 microns. It's believed that within one year Intel will come out with an 80386 made with a 1 micron feature size. This new version should allow clock speeds of 20 or even 25 MHz, but won't cause any changes in the instruction set or in the number of clocks needed for each instruction.

There are currently three types of 80386-based systems: 1) the Compaq 386, built around a proprietary design by Compaq; 2) other announced computers built around an 80386 motherboard (processor plus support chips) made by Intel; and 3) Turbo cards for the IBM PC AT and compatibles, which offer an 80386 and a cache memory and which plug into the socket normally occupied by an 80286.

As this is written there are no true operating systems for the 80386, only a bootstrapping program from Microsoft which allows the Compaq 386 to run 8086-based operating systems. The bootstrapper supports an extended memory specification to allow access to memory beyond the 640 Kb limit of MS-DOS 2.x and 3.x. Also, there are only a couple of assemblers available, and these are known to be "buggy."

Our contacts at companies working on system software for the 80386 tell us that it works as expected, with the exception of some obscure bugs which will not be noticed by applications programmers. This book does not depend on any one operating system, version of an assembler, or computer system. However, the mnemonics (instruction names like MUL and ADD) used by your assembler might be slightly different from the ones we use here.

Future revisions of this book, which will be produced as demand warrants, will note any changes in the 80386. We would be interested in hearing your likes and gripes about this book, along with the things you would like to see included in it or in other works. In particular, we are planning a book of programming examples which will be a companion to this volume, and would appreciate any suggestions for that book. Such letters can be sent to the authors care of Scott, Foresman; simply put both authors' names and the name of this book on the envelope.

CHAPTER 1

Assembly Language Basics

ASSEMBLY LANGUAGE AND THE ASSEMBLER
COMPUTER NUMBERS
BINARY MATH
BINARY MATH APPLICATIONS

There is a tremendous range of experience among programmers using the 80386. Some are systems programmers with extensive experience in operating systems, device drivers, or large assembly language programs. Others are experienced in one or more high-level languages like BASIC and PASCAL, but have used assembly language very little, perhaps only as in-line code within a high-level program. This chapter will help ensure that each reader is exposed to the most important concepts used in assembly language programming.

First we'll take a look at an 80386-style assembly language statement as an example of what an assembler does and what each statement causes the machine to do. Following this we'll discuss the different types of numbers used in assembly-language programming—binary, BCD, hexadecimal, and more. Most programmers will know most of these terms, but the basics of number types and operations will be assumed later in the book, so an overview is given here. If you have programmed

Intel chips in assembler before, you may want to skim this chapter rather than read it. On the other hand, if much of what follows is new to you, further study might be indicated before plunging into 80386 programming. This chapter will be a useful review for the large number of programmers with some, but not extensive, assembler experience.

ASSEMBLY LANGUAGE AND THE ASSEMBLER

The following instruction is a typical 80386 assembly language statement:

```
MOV AX, BX
```

The first word is MOV, an assembly language command, like a direct order from the programmer to the computer. So what gets moved? AX and BX are registers, storage locations in the 80386 itself. The number stored in the BX register is moved into location AX. After the MOV both AX and BX will contain the same value.

Each non-blank line in an assembly language program contains either an "instruction" or a "directive." The most important pieces of an instruction are a command and up to three operands. The command performs some operation, generally using the data in its operands. Other parts of an instruction and a directive are described below.

MOV AX, BX, for instance, has a command and two operands. Most two-operand instructions are of the form:

```
COMMAND Destination, Source
```

Arithmetic instructions with the commands ADD and DIV, for example, look like this. The instruction takes its second operand, performs some operation on it using the first operand (like adding the two together), then stores the result in the first operand. Whatever was in the first operand is overwritten by the result.

Many instructions have implied operands. For instance, the instruction CLC means "clear the C bit," that is, put the Carry bit to 0. The

command itself is "clear;" the operand (the Carry bit) is implied by the instruction. An instruction's operands can be individual bits in the chip's own registers, bytes, words or doublewords in registers, or any of these in the computer's main memory. Chapter 2 will cover more about the overall design of the computer system.

Normally a computer executes instructions in the order in which they appear in the program. However, we need to be able to change this sometimes—in order to make the program execute a loop, for example. To make this happen we might use these instructions:

```
            JCC LoopTop
            :
            :
LoopTop :   MOV BX, AX   ; top of the loop
            :
            :
            JZ LoopTop
```

The word "LoopTop" is a label for the statement MOV BX, AX. The symbol ";" on the same line indicates a comment, which follows immediately. The dots on some lines indicate statements that we're not interested in for now.

The JCC (Jump if Carry Clear) instruction checks the Carry bit (more on this later) and, if a 0 is in it, passes control to the MOV statement labeled "LoopTop." The JZ (Jump if Zero) instruction does the same thing, but checks the Zero bit to decide whether to jump to the label.

An assembly language statement can take the form:

```
Label :   COMMAND   Operand(s)   ; comment
```

The label is optional, and so is the comment; the number of operands needed varies with the command used. The only part of a statement that is always required is the command itself; if a label is alone on a line it's assumed to go with the command on the following line.

Instead of going on with examples of various 80386 instructions here, we refer the interested reader to Chapter 3, which gives a brief functional description of each type of 80386 instruction. Beyond the actual instruc-

tions and operands, there are several other elements in assembly language, and these are explained below.

Instruction Prefixes

Prefixes (reserved words used in an 80386 statement just before the instruction itself) cause some instructions to act differently than they would otherwise. Two useful prefixes are LOCK, which helps the 80386 grab sole control of a piece of RAM when several processors are sharing the same memory, and REP and its variants, which cause an instruction to be repeated. Prefixes are mentioned here because they further enlarge what we can find on any single line of an assembler program:

```
Label :  PREFIX  COMMAND  Operand(s)  ; comment
```

For example:

```
OneLoop :  REP  MOVS  Dest, Source  ; moves bytes
```

This moves a given number of bytes (the number is found in one of the chip's registers) from one location in RAM to another.

Assembler Directives

The commands discussed above are all instructions, which the assembler translates into machine language commands to be executed by the microprocessor. The assembler determines exactly which form of a given command to use in order to get the effect the programmer wants.

There are times when we need to tell the assembler in advance what it is we plan to do. For instance, if our data items in a program are all word-sized (16-bit quantities), we don't want the computer to move doublewords (32-bit quantities) around. Assembler directives look a lot like regular instructions, but aren't translated directly into machine language commands which tell the processor what to do next. Instead, they tell the assembler how to interpret the instructions and directives

which follow them, point out where a program starts and stops, and serve many other functions.

For example, almost every program has variables in it. A directive to tell the assembler a variable's name has the form:

```
Name    Directive   Initial Value   ; comment
```

For example:

```
MyAge   DB          29              ; directive w/fake data
        DB          0               ; 2nd byte, initial value 0
```

This tells the assembler that whenever the variable MyAge is referred to in the program following, a byte value is being referred to and the byte's initial value is 29. The byte reserved on the next line could be referred to as MyAge + 1, meaning "the byte after the byte with MyAge in it." As shown by these two directives, the name at the start of a directive line is optional.

Another important directive is EQU, which tells the assembler to give a name to a number. For instance, the following directive tells the assembler to use the number 62 wherever the word Retire is found in the program:

```
Retire  EQU  62
```

Notice that the DB directive tells the assembler to put a certain value (29) in a given byte in memory, while the EQU directive tells it to remember to replace the word "Retire" with the value 62 as it is converting the program to machine language. In a typical use of the EQU directive, if the retirement age needed by the program changes, we can change the line to read:

```
Retire  EQU  65
```

When the program is reassembled the number 65 will be used wherever the word Retire is found.

More information about directives can be found in the documentation for the assembler you're using, and this should be studied carefully. Among other functions, directives determine how programs are arranged in memory, and they can help give a program some of the structure of a high-level-language program.

Assembler Arithmetic

In both instructions and directives the assembler will evaluate expressions for us. That is, if it finds an expression (like RETIRE − 3) where it expects a single number, it will do the math required and place the result in the machine language program. For instance, if we've used an EQU instruction to tell the assembler that Retire is equal to 65, then the following command will move Retire + 5, or (65 + 5 =) 70 into AX:

```
MOV AX, Retire + 5
```

However, the work of deciding what "Retire + 5" equals is done when the program is assembled, so assembler math can only use numbers that have been specified before a given statement is assembled. Exact descriptions of math done by your assembler will be found in the appropriate documentation.

What Assembly Language Does

You might ask, "What's the difference between assembly language and a high-level language?" The structure of assembly statements is a little unusual, but even the brief examples seen so far demonstrate loops, transfers of control, conditional statements, and variables, all elements of high-level languages.

The most important feature of assembly language is that every instruction is translated by the assembler into one and only one machine language instruction. The machine language code for a MOV between two registers is slightly different than that for a MOV between two locations in RAM, but in both cases each assembly instruction translates

into one instruction in machine language. In a high-level language each statement can be translated into any number of machine language instructions (sometimes just one, but often five or more depending on the statement and its context).

Another important feature of assembly language is that the programmer can directly name the microprocessor registers and exact memory locations to be used for storing and operating on data. When a BASIC programmer says LET A = B, he or she has no control over where the numbers end up. Writing MOV AX, BX in assembler, however, lets the programmer know exactly what he or she is doing.

The different forms taken by a MOV instruction depend on exactly what type of operands it has, and are rarely of concern to the programmer. The important thing is the overall effect, that a value is copied from one location to another. This book will give both the overall effect and the details of how it's achieved, because at those times when it becomes important to know an instruction's format the information needs to be quickly available and understandable.

It is often said that an assembly language program can do everything a high level language can do, and additional things besides. This is true from the computer's point of view, but not necessarily the programmer's. The sheer amount of detail that the assembler programmer must be concerned with—exactly where does this number go, will this loop always terminate—can cause the programmer to shy away from complicated control and data structures. Even a poorly written assembler program will generally execute faster than a well-written high-level-language program, but only the most carefully written assembler code approaches a high-level-language in clarity (to humans) and ease of maintenance.

Assembling an 80386 Instruction

Let's look at Listing 1-1 to see exactly what the assembler does with a very small piece of assembly language code.

The assembler will translate the code in Listing 1-1 into machine language for the 80386 to execute.

Line 1, the EQU statement, tells the assembler to substitute the number 8 wherever it sees the word "GenRegs."

Line 2, the DB statement, tells the assembler that a given byte in memory will be named RegsUsed, and will start out with the value 0.

Line 3 tells the processor to compare the value in RegsUsed, which may have been changed by the lines above it, to the number 8, which is

```
GenRegs   EQU  8   ; 1. number of 80386 general registers
RegsUsed  DB   0   ; 2. number of registers in use
DoMore      :      ; code that stores data in general registers
            :      ; and increments RegsUsed when a register
                   ; is used
            :
            CMP  RegsUsed,GenRegs   ; 3. all registers in use?
            JNE  DoMore             ; 4. if not, continue
```

Listing 1-1. Assembly Process Example

represented by GenRegs. In hexadecimal (discussed later) the machine language produced by the assembler is:

```
80 3E ?? 08
```

This means compare (80) a byte at a specified location in memory (3E ??) to an immediate value (08, as defined by the EQU above). The Zero flag will be set to 1 if the two numbers are equal.

The trickiest part of this is the displacement (??), which is determined by the addressing mode used to find the byte in memory. This is discussed further in Chapter 4.

Line 4 tells the processor to continue executing at the label DoMore if the CMP on the line above showed that the two numbers were equal. The machine language is:

```
75 XX
```

Here 75 indicates a "short jump," to a location within about +/− 127 bytes if the condition of "not equal" (the zero flag is 0) is met. XX is the number of bytes to the location, which would be the number of bytes between the instruction labeled DoMore and this JNE instruction.

Listing 1-2 summarizes the assembly process by showing the assembly code next to the resulting machine-language output.

Assembly Language			Machine Language
GenRegs	EQU	8	
RegsUsed	DB	0	0
DoMore	:		
	:		
	:		
	CMP	RegsUsed,GenRegs	80 3E ?? 08
	JNE	DoMore	75 XX

Listing 1-2. Assembly Process in Hexadecimal

This book gives the numeric equivalent of each 80386 instruction and addressing mode, while an assembler listing typically gives the machine-language output resulting from assembling a given program. Once you grasp the basic concept of how an instruction is translated to machine language, further study will let you learn as much as needed about the machine language code produced by your program.

COMPUTER NUMBERS

Representing What's in the Computer

In order to talk about what's going on inside the computer and command it to do exactly what we want, several different numbering systems are used. The most important is binary, the language used by the computer itself.

The insides of the computer can be seen as a series of toggle switches that can be set to either of two positions, 0 or 1, Off or On, False or True. Each switch is called a bit. To describe whether a series of bits is on or off, we write down their values in sequence (1001 or 10110 or 10001001001); each represents a series of bits in the computer, some of which are on (1) and some of which are off (0).

In order to organize all these bit values, we usually look at them in groups of 8, called bytes. The last binary number above would thus be written as 00000100 01001001, so we could talk about the value in the first byte or in the second byte.

We can describe exactly what's stored at each location in the computer, but writing a byte value (01101001 for instance) takes a lot of

space, so we use hexadecimal notation, which sums up in one symbol what's contained in a group or four bits. A symbol is assigned to each of the possible combinations of bits, as in Listing 1-3.

A single byte, then, can be represented by two hex digits, one for each four bits in the byte; 11000001 becomes C1, and 01001001 becomes 49. Since we can't tell at first glance whether 49 represents a decimal value or a hex value, we write it followed by an H; 49H means 49, hex. Although hex notation is prominently used in programming, it is really only a shorthand way to write out binary values.

Octal notation is commonly used in programming some computers. This notation looks at only three bits at a time, and translates each group of three bits into an octal digit, 0–7. Each character identifies one of eight patterns (not one of sixteen as with hex), and writing out the value in a byte takes three characters rather than two. Octal notation is not used in this book.

Many times we are interested in the number represented by a pattern of bits. To translate a binary pattern into a decimal number we start with the rightmost position in the binary number (the units position). A 1 in the units position has the value one. The position next to it is the

Bit Pattern	Decimal Equivalent	Hex Digit
Pos'n value : 8421		
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	10	A
1011	11	B
1100	12	C
1101	13	D
1110	14	E
1111	15	F

Listing 1-3. Binary, decimal, and hex numbers

twos position; a 1 in the twos position has the value two. The next position is the fours position, and so on using a greater power of two for each position. Looked at in this way, every binary number can be translated into decimal by translating each position into a decimal number and adding the results. For instance, 1110 read from right to left means "0 ones, 1 two, 1 four, and 1 eight." Translating this into decimal is simple: $0 + 2 + 4 + 8 = 14$.

Another way to look at the same thing is to think of the bit positions strictly as successive powers of 2. This is the system used in talking about bit positions in a byte.

Binary number:	1	0	1	0	1	0	1	0
Power of 2:	7	6	5	4	3	2	1	0

The "0th" (or rightmost) bit position has the value 2^0, the next position over has value 2^1, and so on up through 2^7, or 128. So "there's a 0 in bit position 4" means that the fifth position from the right has a 0 in it.

Representing Numbers

There are problems with representing numbers in pure binary form. For one thing, we've only talked about representing integers; no fractions or numbers with decimal points have been discussed. Also, most people don't work in binary, so numbers have to be converted back and forth on their way in and out of the computer.

One way to simplify number representation is to make the computer use base 10, as most people do. This is done using BCD (Binary Coded Decimal) notation. The 80386 includes hardware support for BCD arithmetic in a series of "ASCII Adjust" instructions that use the AL register to do BCD math two digits at a time.

In "packed BCD" four bits represent one digit, 0-9. As the figure above shows, four bits can hold sixteen different patterns (as in the hex digits 0-9 plus A-F), yet we only need ten of them (the decimal digits 0-9), so each BCD digit is wasting some of the available storage capability. An even worse problem comes when we try to do math with these numbers, because the computer automatically treats them as binary numbers, producing results that make no sense for BCD digits. Although we'll go

into binary math in more detail below, let's add two BCD digits to illustrate this problem.

Binary math: 1000 (8 BCD) + 0100 (4 BCD) = 1100 (? BCD)

The sum of the two numbers is 12, but by the rules of BCD, 12 is too large a number to fit in a single, 4-bit BCD digit. What we want is for the computer to do a carry for us whenever the result of adding two BCD digits results in a number above 9 (which no longer fits in a single digit):

BCD math: 1000 (8 BCD) + 0100 (4 BCD) = 0001 0010 (12 BCD)

The ASCII Adjust instructions automatically note when an addition or other arithmetic operation has caused a discrepancy between binary rules and BCD rules, and make the needed adjustments. Note that two packed BCD digits fit in a single byte, so with BCD notation a single byte can express any number from 00 through 99; in binary the same byte can express any number from 0 through 255. BCD causes a loss of storage capability.

If enough RAM is available to make up for the wasted space caused by BCD digits, and if speed is not of the essence, BCD math is very effective for results that need to be in a human-understandable format. To store large numbers we can just use more BCD digits as needed, and to handle numbers with decimal points we simply agree on a convenient format that tells us where to place the decimal point for each number. BCD works well for dollars-and-cents applications, where it's unusual to find a number of much more than 15 digits and precision all the way down to the cents columns is very important.

However, for large numbers (as used in science and other applications) we need a format that stores very large numbers in a fixed number of bytes and doesn't just keep adding to the amount of storage needed as numbers get larger. The answer here is to use a floating-point format. Almost all floating-point math is done through prewritten software packages, and you'll need to learn the rules of any such package you wish to use.

Floating-point numbers are an agreed-on format for storing large numbers in a limited number of bits. For instance, a 32-bit floating-point number might have this format:

First bit: sign bit of fraction
Next 8 bits: exponent
Next 23 bits: fraction

It is assumed that the number we're representing is in the form:

$$(+/- \ 1.\text{fraction}) \times (2^{+/- \ \text{exponent}})$$

This is much like the scientific notation you might have learned in school. Since the "1" at the start of "1.fraction" and the "2" as the base for the exponent are always the same, they're not included in the floating-point number. Also, a trick involving the largest number that can fit in 8 bits allows the sign of the exponent to be deduced, so it's not explicitly stored in the number either.

The advantage of floating point numbers is that any number between about $1/2^{128}$ and $1*(2^{127})$ can be represented in a mere 32 bits, so we can express (if sometimes inexactly) numbers as small as about 1 over a 1 followed by 38 zeros, and as large as about 1 followed by 37 zeros. However, the floating-point representation is not precise since the fraction part of the number is 23 bits, not 128. Thus a floating-point number is usually a rounded number. Precision can be improved greatly by using 64 or 80 bits instead of 32, but precision loss remains in many cases. The adjustments needed to make a binary-based computer do floating-point arithmetic are so complicated that such math is almost always done through calls to prewritten software packages or math coprocessors.

Translating Large Binary Numbers

When working with computers we end up talking a lot about large binary numbers. The 80386, for example, handles data in 32-bit chunks, and the largest number we can express in 32 bits is:

11111111 11111111 11111111 11111111

Reading right to left, this is:

1 + 2 + 4 + . . . + 1,073,741,824 + 2,147,483,648 =
4,294,967,295 (4 billion bytes, or 4 Gb).

This number is also the number of bytes of RAM that the 80386 can address directly. When you read that the 80386 can directly address 4 gigabytes, you can see that it does so by using a 32-bit–long number to point to the byte that is being addressed.

For dealing with these large numbers and for doing math with smaller numbers, a programmer needs to have a feel for what a given power of 2 translates to in decimal. Listing 1-4 shows the number of bytes associated with the powers of 2 from 2^0 up through 2^{31}.

These numbers are very handy when talking about the capabilities of the 80386. A good way to deal with the really large numbers, like 2^{24} or 2^{32}, is to note that 2^{10} is just over a thousand (1 Kb), 2^{20} is just over a million (1 Mb), and 2^{30} is just over a billion (1 Gb). Powers in between fall into the same kind of progression (1, 2, 4, 8, 16, 32, 64, 128, 256, 512) that is found between 2^1 and 2^9. So 2^{16}, for example, is 64 (because of the 6) Kb (because of the 10).

BINARY MATH

The next few pages contain a reasonably complete description of binary math in very compact form, because many readers will already know some or all of the material. If any of the sections below are new to you, you'll save a lot of programming and debugging time in the future by working out the examples given and then constructing a few of your own until the principles involved are understood.

We can do binary math by simply following a set of rules, using the decimal values of the numbers only as a check. In adding two binary digits, we can run into four possible combinations:

0 + 0 = 0
0 + 1 = 1

$1 + 0 = 1$
$1 + 1 = 0$, carry of 1

We can add any two binary numbers just by repeatedly applying these rules for one pair of bits at a time, starting with the two rightmost digits

Number of bytes	Decimal equivalent	Name
2^0	1	
2^1	2	
2^2	4	
2^3	8	
2^4	16	
2^5	32	
2^6	64	
2^7	128	
2^8	256	
2^9	512	
2^{10}	1,024	1 kilobyte (kilo = thousand)
2^{11}	2,048	2 Kb
2^{12}	4,096	4 Kb
2^{13}	8,192	8 Kb
2^{14}	16,384	16 Kb
2^{15}	32,768	32 Kb
2^{16}	65,536	64 Kb
2^{17}	131,072	128 Kb
2^{18}	262,144	256 Kb
2^{19}	524,288	512 Kb
2^{20}	1,048,576	1 megabyte (mega = million)
2^{21}	2,097,152	2 Mb
2^{22}	4,194,304	4 Mb
2^{23}	8,388,608	8 Mb
2^{24}	16,777,216	16 Mb
2^{25}	33,554,432	32 Mb
2^{26}	67,108,864	64 Mb
2^{27}	134,217,728	128 Mb
2^{28}	268,435,456	256 Mb
2^{29}	536,870,912	512 Mb
2^{30}	1,073,741,824	1 gigabyte (giga = billion)
2^{31}	2,147,483,648	2 Gb
2^{32}	4,294,967,296	4 Gb

Listing 1-4. Powers of 2 and Bytes

and working left (just as with two decimal numbers). The only complication is the carry generated when we add two 1's; the trick is to make sure to add the two digits in the operands first, yielding a one-bit result, then add the carry to this result to determine the final value (and whether or not there's a further carry). Here's an example:

```
 1100110
  110111
---------
10011101
```

This is the equivalent of adding (0 + 2 + 4 + 0 + 0 + 32 + 64 =) 102 and (1 + 2 + 4 + 0 + 16 + 32 =) 55, with a result of (1 + 0 + 4 + 8 + 16 + 128 =) 157, as expected.

Subtracting Binary Numbers

Subtraction is less intuitively obvious than addition. The four possibilities when subtracting one binary digit from another are:

0 − 0 = 0
0 − 1 = 1, borrow 1
1 − 0 = 1
1 − 1 = 0

We can subtract by repeatedly applying these rules to two binary numbers, starting at the rightmost end, just as in addition. Again, subtract the two bits in the operands first, then subtract any borrow from the result. When the result of subtracting two digits is a 1 and a borrow is applied, the result becomes 0; when the result of subtracting two digits is a 0 and a borrow is applied, the result is 1 and a further borrow is generated.

```
 1100110
  110001
---------
  110101
```

Here's how an internal dialogue of someone doing this subtraction in his or her head might go:

"Ones position:	0 − 1	= 1, borrow 1.
Twos position:	1 − 0	= 1, subtract borrow, result is 0.
Fours position:	1 − 0	= 1.
Eights position:	0 − 0	= 0.
Sixteens position:	0 − 1	= 1, borrow 1.
Thirty-twos position:	1 − 1	= 0, subtract borrow, result is 1 with a borrow of 1.
Sixty-fours position:	1 − 0	= 1, subtract borrow, result is 0."

This is the equivalent of subtracting 49 from 102, with the result (32 + 16 + 0 + 4 + 0 + 1 =) 53, as expected. This process is complicated enough that people often do binary subtraction by converting both operands to decimal, doing the subtraction, then converting the result to binary again. Another technique is to convert the second operand to its opposite (negate it), then add the two numbers together, using the addition technique. This is the technique used inside the 80386 and other processors in its family.

Negative Binary Numbers

So far we've only dealt with positive numbers, and we've used as many bits to represent a number as it needed; the first bit is always 1, because it's useless to write leading zeros. In most systems which can handle positive and negative numbers, however, a leading 1 always indicates a negative number. In order to express both kinds of numbers, we agree in advance that all numbers will be (for example) eight bits long, and that leading zeros will be written out, as in 00000100 (the binary representation for 4 in decimal).

Several different ways exist to describe negative numbers. The simplest is to put an additional bit in front of each number to represent the sign of the number, a 0 bit for positive, a 1 bit for negative, with the bits after the first one representing the number's size. This is called "sign-magnitude representation," meaning "a sign bit in front, plus several bits to represent the magnitude." However, the addition and subtraction

rules described above, which are the simplest possible, don't work for sign-magnitude numbers unless adjustments are made. Another problem is that there are two equivalent ways to represent zero: 00000000 (or 0), and 10000000 (or "negative zero"); both mean the same thing. The floating-point representation described above is a modified form of sign-magnitude representation; it's compact but difficult for doing arithmetic.

Inside the 80386, as in most computers, negative numbers are represented by a method called "two's complement" notation. We use this notation because the same simple math rules apply to both positive and negative two's complement numbers. Positive numbers in two's complement are exactly the same as in regular binary notation: 4 decimal is 00000100 binary; 11 decimal is 00001011 binary. Zero is 00000000.

To form a negative number we construct a positive number of the same magnitude and then flip (i.e., negate or invert) every bit, which yields "one's complement" notation, then add one to this result. Here are two examples of forming negative two's complement numbers:

Start with decimal number:	−4	−11
Write out positive number in binary:	00000100	00001011
Negate each bit (one's complement):	11111011	11110100
Add 1 (two's complement):	11111100	11110101

There are a couple of things worth noting. Each negative number in two's complement starts with 1, so the lead bit serves as a sign bit; the negative numbers work fine for the computer but are hard for a person to convert to decimal. The simplest way to do this for negative two's complement numbers is to find the absolute value of them by reversing the process above: negate each bit and then add 1 to get the positive component (magnitude or absolute value) of the negative number. The resulting positive binary number is then easy to convert to decimal.

Here are two examples of arithmetic with two's complement numbers:

```
  11001000 (−56)        11101111 (−17)
+ 01101001 (105)      − 00000011 ( +3)
  00110001 (+49)        11101100 (−20)
```

Both follow the same arithmetic rules as the positive (i.e., unsigned) numbers we dealt with before. In the rest of this book a two's complement number will also be called a signed number, meaning a binary number in which the high-order (leftmost) bit serves as a sign bit.

BINARY MATH APPLICATIONS

Overflow and Carry

When the 80386 performs a math operation (like an add or subtract) it uses a special piece of hardware called an adder, which handles only a couple of different operations, but performs them very quickly. The 80386 can perform up to 8 million simple additions in a second.

However, sometimes even a simple operation like addition can get complicated. When we add two unsigned numbers the result can be too large to fit in the number of bits available. For example, 10000001 (129 decimal) + 01111111 (127 decimal) = 1 00000000 (256 decimal), which doesn't fit in eight bits. The adder will return a result of 00000000, which is not what we expect.

To communicate that a problem has occurred, the adder sets an extra bit inside the microprocessor (called the "carry flag"). This flag is set (to 1) if the result of an addition doesn't fit in the number of bits available for the result, and is cleared (to 0) otherwise.

In a signed number only the low-order (leftmost) seven bits contain a number's value; the high-order bit is reserved for the sign. The equivalent of a carry (magnitude too large to fit the representation) occurs when adding two seven-bit numbers gives a result that won't fit in seven bits. For example, 01111100 (124 decimal) + 00001111 (15 decimal) = 10001011 (−11 decimal), which is not right at all. This forced changing of the high-order bit, caused by the sum of the remaining bits being too large, is called an overflow. There's a flag in the 80386 (called the "overflow" flag) that is set any time the result in the first seven bits changes the eighth bit, and cleared otherwise. The overflow can be ignored when we're dealing with unsigned numbers. To summarize, a carry is caused by a carry out of the leftmost bit, and an overflow is caused by a forced change of the high-order bit.

The exact conditions under which each flag is set can be important, and there are other flags in the chip that depend on the results of

arithmetic, as do carry and overflow. These are discussed in detail in Chapter 2.

Sign Extension

So far all our examples of binary math have dealt with numbers that can be represented in an eight-bit byte. The 80386 also uses words (two bytes) and doublewords (four bytes) in its hardware. Often, we need to convert a byte to a word or doubleword (called a dword).

This is simple for positive numbers; just add zeroes to the left end of the number. The byte 00010011 (19 decimal) becomes the word 00000000 00010011 (19 decimal), for instance. Negative numbers in two's complement representation are a little more complicated. If we add zeroes to the left end of a negative number, it changes completely; 11101101 (−19 decimal) becomes 00000000 11101101 (237 decimal). Luckily, there is a simple rule called sign extension that fits smaller operands into larger ones while preserving both magnitude and sign. Just take the leftmost bit of the smaller data type (i.e., byte or word) and repeat it in the extra bits of the larger data type (i.e., word or dword). The byte 00010011 (19 decimal) still becomes 00000000 00010011 (19 decimal), but the byte 11101101 (−19 decimal) becomes 11111111 11101101 (−19 decimal) as a word. The 80386 allows you to either zero extend (the added bits become zeroes) or sign extend (the added bits are copies of the leftmost bit) when moving a byte into a word or a doubleword, or moving a word into a doubleword.

Binary Logic

Some words we use somewhat loosely in English (such as "and," "or," and "not") have precise meanings in computer math as logical operators on numbers. Listing 1-5 shows what different logical operators do when applied to different combinations of bits.

One way to remember these rules is to think of the 0's as representing false English statements (like "all hair is green") and the 1's as representing true statements (like "the earth is round"). Substitute these statements for the 0's and 1's on the left hand side of the equations in Listing 1-5 to make sentences. The equation is true if the resulting sentence is also true. For example, 0 AND 0 becomes "all hair is green AND the earth is round;" this is false because all hair isn't green.

NOT 0 = 1 NOT 1 = 0
(result is opposite of operand)

0 AND 0 = 0 0 AND 1 = 0 1 AND 0 = 0 1 AND 1 = 1
(result is 1, or True, if and only if both operands are True)

0 OR 0 = 0 0 OR 1 = 1 1 OR 0 = 1 1 OR 1 = 1
(result is 1, or True, if either operand is True)

0 XOR 0 = 0 0 XOR 1 = 1 1 XOR 0 = 1 1 XOR 1 = 0
(result is 1, or True, if one and only one operand is True)

Listing 1-5. Binary Logic

This trick doesn't work as well for XOR, which is short for "eXclusive OR" and means "either one, but not both." As we start using non-English constructions like XOR and construct statements that combine two or more operators in one statement (NOT 0 OR 1), it becomes easier to simply memorize and apply rules like those in Listing 1-5 than to construct increasingly convoluted sentences as examples.

We can apply these logic rules to bytes as well as bits. Simply line the bytes up as if you were going to add them, then use the rules above on each pair of bits in turn. Since there is no carry, it doesn't matter whether the comparisons start with the two leftmost bits or the two rightmost bits. Listing 1-6 shows the results of applying the NOT, AND, OR, and XOR operators to byte operands.

These operators work the same way on operands of any number of bits as long as both operands are of the same length.

```
NOT          11001010 AND   11001010 OR    11001010 XOR
01010110     01010110       01010110       01010110
10101001     01000010       11011110       10011100
```

Listing 1-6. Examples of NOT, AND, OR, XOR

CHAPTER 2

80386 Applications Architecture

In this chapter we talk about the 80386's architecture, or the way its facilities look to the programmer trying to make the chip work. We start with a brief description of the history of the 80386, and an equally brief discussion of the functions of a typical microprocessor. This chapter concentrates on the facilities used by the applications programmer and describes each of the chip's registers, including those generally used only by operating systems. Also covered are interrupts and exceptions.

A program running in the 80386's Virtual 8086 Mode works almost exactly the same as one running in Real Mode, and both are well suited for bringing 8086 programs and operating systems, and 80286 Real Mode programs, onto the 80386.

Programs running in Protected Mode have access to all the facilities in Real Mode (as described here) plus those available in Protected Mode. An understanding of the material here is necessary for using any of the modes and capabilities of the 80386.

We start with some background on the evolution leading up to the Intel 8086 family and the 80386. This gives some insight into why the 80386 works the way it does and what the future may hold. Next, we'll talk about what a microprocessor is and what parts make it up. Following this is the bulk of the chapter, which is about the 80386's internal organization, how it looks at memory, and the data types and addressing modes it uses in talking to memory. This chapter covers all registers and addressing modes except the ones only accessible through Protected Mode. Some of the material here, though, can be skimmed by the experienced 8086 and/or 80286 programmer, since much of the 80386 design is an extension of these earlier microprocessors.

INTEL'S AND OTHER EARLY MICROPROCESSORS

The 4004 from Intel (the first general-purpose microprocessor) appeared in 1971. Like the later examples of its type, it could execute any program made up of the right commands, bring data in, operate on it, and send data back out. All this was done by a concatenation of electrical components about the size of a quarter. The 4004 operated on four bits of data at a time, with each four bits encoding a single decimal digit, as in BCD math (Chapter 1). The 4004 was originally designed for use in calculators. It contained the equivalent of about 1,000 transistors and performed 8,000 operations in a second, easily fast enough for calculator use.

Within the next couple of years Intel introduced two new microprocessors, the 8008 (basically a 4004 which processed eight bits at a time) and the 8080 (the first chip powerful enough to run a small computer). The 8080 (which is still popular for process control and other applications) includes facilities for performing decimal and 16-bit math, making subroutine calls, and addressing memory up to 64 Kb. The 8080 uses an 8-bit data path and a 16-bit addressing bus, meaning it can handle numbers and addresses up to 64 K (2 to the 16th) easily.

A group of Intel engineers left their company to form Zilog (just as Intel itself was formed by ex-Fairchild Semiconductor people) and in

1976 introduced the popular Zilog Z80. This did everything the 8080 does and then some, adding additional instructions and registers. However, so many 8080-based computers existed by this time that most programs were still written to run unchanged on both the Z80 and the less powerful 8080, maximizing market share but sacrificing performance. Computers which run the once-popular CP/M operating system tend to have some version of the 8080 or Z80.

Other companies weren't oblivious to the progress and sales made by these new chips. In 1974 Motorola introduced the 6800, which was as powerful as the competition and added new ways to address data, speeding program development and execution. In 1975 some ex-Motorola engineers produced the first processor in the 6500 series, which was similar to but cheaper than the 6800. The MOS Technologies 6502 and its descendants are found in the Apple II series of computers.

In 1980 Motorola came out with the 68000, a 16/32 bit chip which mostly handled 16-bit values but had some ability to process 32 bits internally and used 32-bit addressing to directly address up to 4 billion bytes of memory. The 68000 could handle about 800,000 operations per second. Its descendants include the 68010, the 68020, and the brand new 68030; each of the chips in this family powers both microcomputers and minicomputers. The 68000 series is currently the main competition in sales and capability to the 8086 family.

In the decade between the 4004 and the 68000, microprocessors improved dramatically in most respects; about 70 times as many transistors could be found on a chip, the largest available word size had climbed from 4 bits to at least 16 bits, and the speed of operation was increased over 100 times. Although some physical limitations of chip construction were approached, the size of the market has increased enough to inspire a similar rate of improvement in this decade, as reflected by the 80386. What the future holds is anyone's guess.

THE 8086 FAMILY

In 1978 Intel introduced the 8086, followed a year later by the 8088. The 8086 is a 16-bit chip (in the data sizes it processes internally and in the size of words its data bus brings in from memory). However, the address size used is 20 bits, so 1 megabyte (2 to the 20th power) can be addressed directly, but only in 64 Kb chunks.

The 8088 works just like the 8086, but its data bus is only 8 bits wide. When the 8088 needs a 16-bit quantity it must get it in two 8-bit chunks, slowing operations. When processing data internally or working with byte-sized data, the 8088 is just as fast as the 8086 and has the advantage of being able to work with inexpensive 8-bit memory chips and other peripherals designed for older microprocessors. It is irritating to the experienced 8088 programmer to find repeated references to the "8086" or the "basic architecture" in technical documents and books. All this means is that the 8086 is the model which must be emulated by other members of its family to preserve compatability. The 8088 is identical to the 8086 in every way but the size of its data bus. Any reference to the 8086 mean "the 8086 and the 8088" unless specified otherwise.

In discussing the 80386's operations in Real Mode we will largely be talking about the capabilities of an 8086. The instruction set is somewhat extended on the 80386; several new registers have been added, as has 32-bit capability. However, the 8086 programmer has only a short learning curve to traverse before learning to program the 80386 in Real Mode.

MICROPROCESSOR BASICS

Every computer has at its heart a central processing unit with the computer's arithmetic and logic control circuitry. A microprocessor is a central processing unit (CPU) fitted onto one (or at most a few) silicon chips. The small size of the microprocessor is less important than its low cost, which makes possible low computer prices and therefore the recent explosion in small computers.

From a programmer's point of view the basic workings of a microprocessor are the same as the basic workings of any other CPU. The two necessary parts of a CPU are an arithmetic/logic unit (ALU) that performs arithmetic operations, and a control unit (CU) that brings data to the ALU and otherwise directs the moving of data within the computer. In order to speed up operations most CPUs contain registers, which are quickly accessible storage locations. Some of the registers are used mostly by programmers, while others are reserved for use by the control unit.

One of the two remaining parts of a computer is the memory, which holds programs and data. There are two types of memory we're concerned with on the 80386: random access memory (RAM), which the

programmer can both write data to and read data from, and read-only memory (ROM), which can only be read. RAM generally loses its contents when power is turned off or the computer is rebooted, while ROM keeps the same contents at all times. The other parts of a computer in which we are interested are its I/O ports, through which data is sent and received between the CPU and the outside world. Figure 2-1 depicts the relationship between these different parts of the computer.

The distinction between memory and I/O ports is becoming blurred because much I/O is memory-mapped; the contents of locations in RAM control what is output (memory-mapped video displays) or reflect what is input (memory-mapped keyboard input). In these cases I/O is handled just by reading and writing values in memory put there automatically by an input device or transferred automatically to an output device. The only other use of I/O by many programs is to write to and read from a disk. This is largely accomplished by calls to the operating system, which accesses code in ROM to make the transfers.

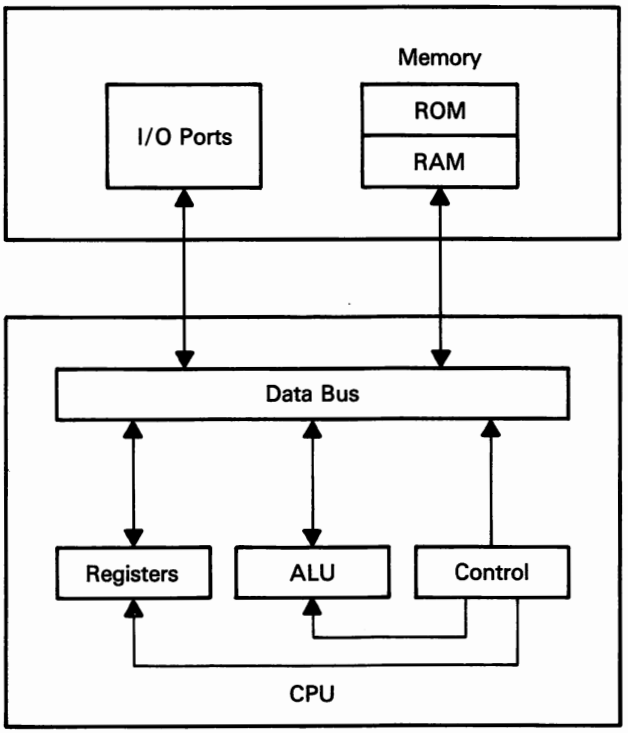


Figure 2-1. The CPU plus I/O and memory

Data movement inside the computer is done at the command of the control unit on the CPU, and goes through the data bus (which is like a parallel port between the CPU and the memory and I/O ports). The computer's registers and ALU are each connected directly to one end of the data bus, while memory and I/O are connected to the other. The control unit directs the placing of data on (and reading of data from) the bus by the CPU.

When a computer is referred to as a 16-bit or 32-bit computer, this refers to the number of bits which fit in its internal registers. Almost as important as the register size is the width of the data bus. When the 8088 is referred to as an 8/16–bit computer, this means it has an 8–bit data bus and a 16–bit internal register size. The 80386 is a "true 32-bit" computer, meaning that its registers and data bus are all 32 bits wide. The 80386 also has a 32-bit address bus, making address calculation much easier, because a single register can hold a complete address.

The 80386 is highly integrated; it has its CPU (including ALU and registers) and memory management unit (discussed in detail below) all on a single chip. This is a real achievement for a 32-bit processor. Main memory (RAM and ROM) and I/O ports are not on the microprocessor chip.

There is much we haven't discussed here: the timing of the different signals that control data movement and other chip functions, how the 80386 communicates with memory, and how a computer is physically organized into circuit boards, buses, and so on. Many of the hardware details are covered in Chapter 7. Communications with memory are covered below; the other chips found in a typical 80386-based computer aren't discussed here, because they change as new, more highly integrated support chips become available.

THE FETCH-DECODE-EXECUTE CYCLE AND THE 80386

Most computers in current use execute one instruction at a time. A regular order of events is repeated for each instruction: an instruction is fetched from memory, the instruction is decoded (translated from 0s and 1s into the microcode used inside the CPU itself), and then the microcode is executed. This is called the fetch-decode-execute cycle.

There are a couple of additional considerations on most computers. During fetching a register called the Program Counter is automatically

incremented. The Program Counter tells the computer where to look for its next instruction. As the counter is incremented by the same amount for each fetch, instructions are executed in the order in which they're stored in memory. This only changes when a "jump" instruction or a "call" to a subroutine forces a brand new value into the Program Counter, causing the program to get its next instruction from some new location.

The other consideration is that many instructions have operands that are stored in memory when the instruction is brought into the CPU. For instance, an ADD instruction might add two numbers, one of which is in memory. The value in memory must be brought into the ALU so it can be operated on. Thus, the fetch-decode-execute cycle comes to look more like a fetch-increment program counter-decode-get operand from memory-execute cycle. Instructions with an operand in memory can start taking a long time to complete.

The 80386 gets around this complication by using a technique called "pipelining." While one instruction is being fetched another is being decoded and a third is being executed. Five or six instructions are typically in one or another of these stages at any given time. Pipelining is discussed in great detail in Chapter 7; the important thing to understand for now is that most of the time the 80386 finishes executing one instruction and then immediately starts executing a new one that has already been fetched and decoded.

The 80386 Processor

The 80386 processor is divided up into functional pieces called "units." The only one the programmer has direct control over is the Execution Unit, which contains the chip's onboard storage registers and arithmetic hardware, plus the controller that actually causes instructions to execute. Besides the chip the most important element of the computer is main memory, usually in the form of RAM. In the rest of this chapter we're going to talk a little bit about the 80386's ALU, and a lot about its registers and communications with memory. The areas talked about the most are the ones that an applications programmer has the most control over.

The 80386's Execution Unit includes a fast adder, which works with 32-bit values, and a 64-bit barrel shifter, which can shift or rotate a 32-bit operand by up to 31 bits in either direction. The important thing to know for programming purposes is that adds and shifts are among the fastest instructions, especially if the operands are in registers.

The 80386's Applications Register Set

The 80386 has many registers; most are accessible to applications programmers, a few are used only by systems programs. The applications register set includes the general registers, the segment registers, the flags register, and the instruction pointer. These are discussed below.

"Base register set" is another term used to describe some of the registers found on the 80386. The base register set is a core group of registers that are found on each and every member of the 8086 family. These include the lower 16 bits of the applications registers along with the machine status word (covered in Chapter 6).

The 80386 has eight general registers. The registers are each 32 bits wide. The full 32-bit registers have names starting with E (for Extended): EAX, EDX, ECX, EBX, EBP, ESI, EDI, and ESP (Figures 2-2 and 2-3). The lower 16 bits of each of these registers can be addressed using the same

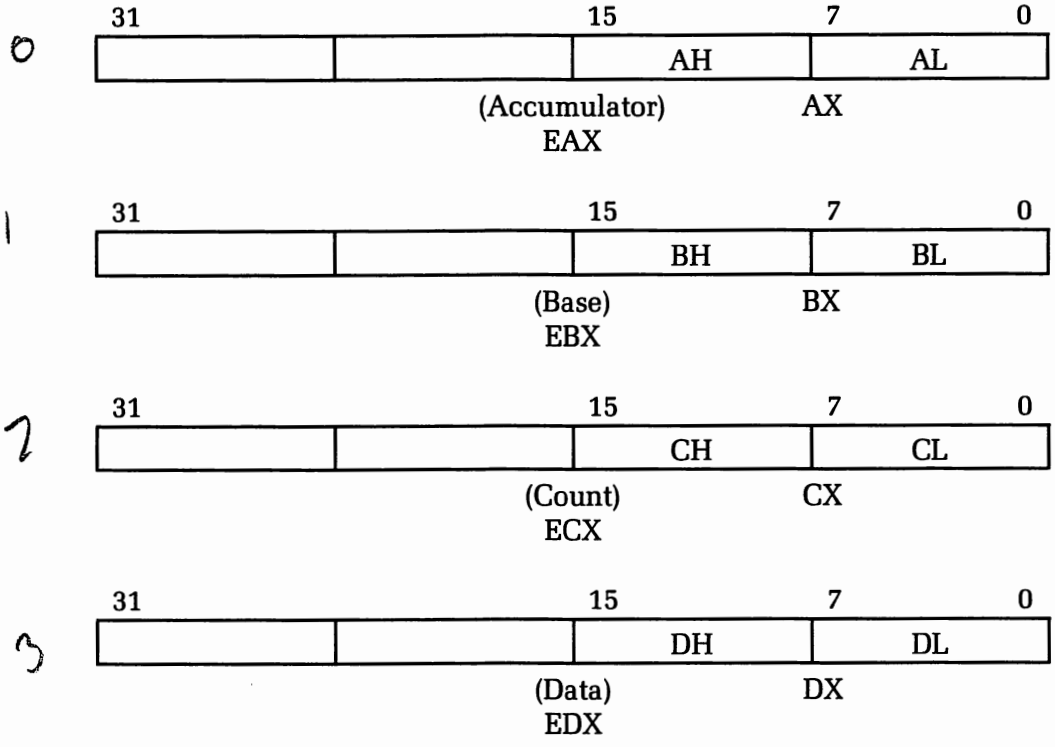


Figure 2-2. 80386 Data Registers

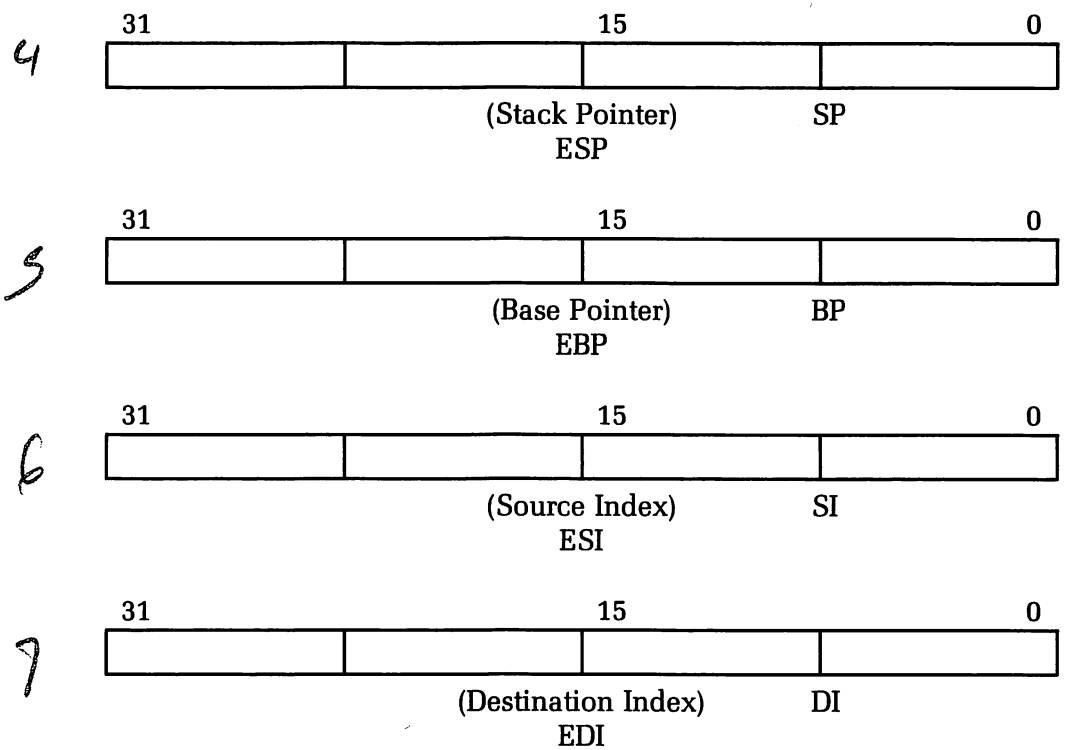


Figure 2-3. 80386 Pointer and Index Registers

names as on earlier 8086-family chips: AX, DX, CX, BX, BP, SI, DI, and SP (the same names without the E). Finally, the first four 16-bit registers can each be addressed pairs of byte-sized registers: AH and AL name the high (bits 8 through 15) and low (bits 0 through 8) halves of AX; the same pattern is used for DH and DL, CH and CL, and BH and BL, each naming one of the two bytes in the corresponding 16-bit register.

All of the registers are doubleword addressable (names starting with E, for Extended); they are accessible as full-sized 32-bit registers. For a register to be "dword addressable" means that a single command (for example, MOV EAX, 1) affects a whole doubleword (or dword). The lower half of each register is word addressable (accessible as a 16-bit register). A dword-sized register that is "word addressable" can have values put in its lower word without affecting the upper 16 bits of the dword. When we wish to refer to a register in either its 32-bit or 16-bit form (whichever is more convenient for the programmer) we put the E in the name in

parentheses: (E)AX means "EAX or AX, whichever is needed." The first four registers are "byte addressable" in their first two bytes.

Despite the fact that these are general registers, each has specific uses. Operations like ADD allow the programmer to name any two registers as operands. Operations like PUSH, for instance, assume that the location in the stack which the operand will be pushed to is pointed at by SP. Because of this type of assumed use, each register is reserved for certain purposes when needed:

(E)AX, or the Accumulator register, is used for BCD math.
(E)BX, or the Base register, is used as a base for address calculations.
(E)CX, or the Count register, is used as a counter for string operations.
(E)DX, or the Data register, holds data for any of several different kinds of operations.
(E)SP, or the Stack Pointer, has the current offset of the top of the stack.
(E)BP, or the Base Pointer, can point to the base of a data area.
(E)SI and (E)DI, or the Source and Destination Index, are used when moving strings to point to the source string and the destination string.

Further details on the uses of each register are given in chapter 4.

The segment registers on the 80386 are used to name the starting points in memory of different pieces of code and data (Figure 2-4). There are six segment registers, each with a different purpose. CS is the starting address of a program's code, DS of its data, and SS of its stack. ES, FS and GS are all extra segments for additional data structures; of these only ES is used specifically by certain instructions.

An applications program often doesn't need to modify the segment registers, which can be completely controlled by the operating system.

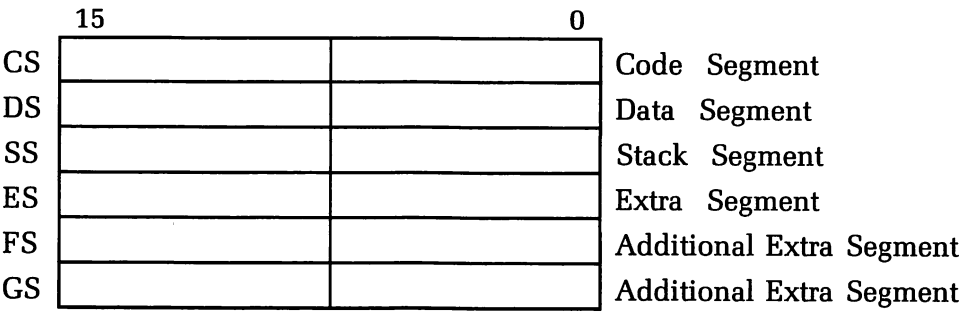


Figure 2-4. The Segment Registers

The contents of the segment registers are combined with other registers to tell the program where to get its next instruction (CS plus the Instruction Pointer), where the top of the stack is (SS plus ESP, the Stack Pointer), and so on. The exact way the two registers are combined varies depending on what mode your program is operating in, as explained below.

The final two registers of interest are EFlags, which controls some operations and indicates the current status of the 80386, and the Instruction Pointer, which is combined with the CS (Code Segment) register to point to the next instruction to be executed. Only some of the bits in EFlags (each of which has a different meaning) can be directly set by the applications programmer, and the Instruction Pointer can only be changed as a side effect of operations like jumps and calls.

A BIT-BY-BIT LOOK AT EFLAGS

EFlags is the 80386's 32-bit-wide flags register. Applications deal only with the lower 16 bits of this, which are collectively called the Flags register. The bits in EFlags reflect the status of the 80386 and control the way some operations are performed.

There are three different types of flags used here: Systems flags (which reflect the current state of the machine as a whole and which are more often used by operating systems than by applications programs), Status flags (which reflect the state of a particular program), and a Control flag (which directly affects how a few instructions operate). Figure 2-5 shows the EFlags register and the Flags register (the EFlags' lower 16 bits).

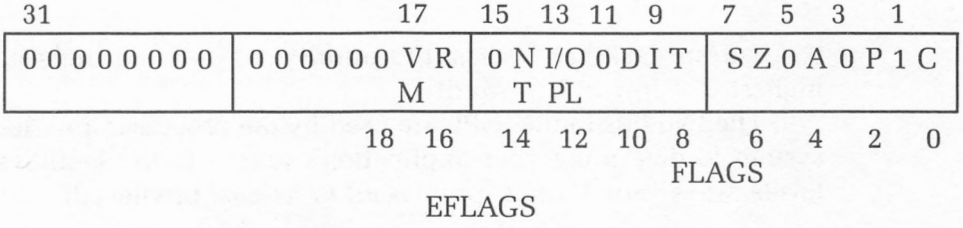


Figure 2-5. EFlags and Flags

Reserved for Intel

Some bits have been reserved for Intel; bits 31 through 18, the first 14 bits of EFlags, are always 0. Bits 15, 5, and 3 are also always 0, and bit 1 is always 1. These bit positions are reserved for Intel use. If you use these bits and other "reserved for Intel" bits your programs will probably work on current versions of the 80386, but may run into trouble on later Intel chips.

Systems Flags

VM (VIRTUAL MODE) FLAG, BIT 17

0 = Protected Mode, 1 = Virtual 8086 Mode

The Virtual Mode flag indicates whether your program is running in Virtual 8086 Mode (discussed in Chapter 5). You generally won't be able to examine this bit or the next one while in Real Mode.

R (RESUME) FLAG, BIT 16

0 = no fault, 1 = debug fault

The Resume flag turns off debugging temporarily when a program resumes just after a debugging exception.

NT (NESTED TASK) FLAG, BIT 14

0 = current task not nested, 1 = current task is nested

NT, for Nested Task, indicates whether the current task is running "beneath" some other task; it affects how the IRET instruction operates.

IOPL (I/O PRIVILEGE LEVEL) FLAG, BITS 13 AND 12

0 = current task has highest I/O priority, 1 = second highest, 2 = third highest, 3 = lowest I/O priority

The two bits in the IOPL are used by the processor and the operating system to determine your application's access to I/O facilities. Allowed levels range from 0 (most privileged) to 3 (least privileged).

I (INTERRUPT) FLAG, BIT 9

0 = external interrupts disabled, 1 = enabled

The Interrupt flag controls whether or not the CPU will respond to or ignore external interrupts. Exceptions (caused directly by the running of a program) and nonmaskable external interrupts are unaffected by this flag. Programs which will run on multitasking systems, as many 80386 programs will, should modify this flag as infrequently as possible.

T (TRAP) FLAG, BIT 8

0 = no trap, 1 = interrupt after each instruction

The Trap flag causes an exception to be generated by every instruction; this is used for single-stepping instructions when debugging.

Status Flags

Status flags (used directly by applications programs) are set or cleared by some 80386 instructions, especially arithmetic ones.

Once conditioned the flag can be examined by the programmer to help determine what the program will do next. Specific instructions cause the flags to be changed in specific ways, so while each flag has a general function, the exact meaning of the flag's status depends on the instruction that has executed most recently.

O (OVERFLOW) FLAG, BIT 11

0 = no overflow, 1 = overflow occurred

The overflow flag is set (put to 1) when the result of some arithmetic operation is too large (needs too many bits) to fit in the result; if the result fits, the 0 flag is cleared.

S (SIGN) FLAG, BIT 7

0 = high bit is 0, 1 = high bit is 1

The Sign flag has the same value as the high bit of an instruction's result. When using signed numbers this high bit indicates the sign of the destination operand: 1 if negative, 0 if positive.

Z (ZERO) FLAG, BIT 6

0 = last result not 0; 1 = last result was 0

Zero is set to 1 (True) if the result of an operation is zero, and is set to 0 (False) if the result is nonzero. Thus the Z flag is only 0 when the result is nonzero.

A (ADJUST OR AUXILIARY CARRY) FLAG, BIT 4

0 = no internal carry, 1 = internal carry

This bit is called the Adjust flag (or Auxiliary Carry flag). It indicates whether an "internal carry" has occurred. If a BCD add or subtract causes a carry or borrow from the fourth bit of the operand into the fifth bit, the A flag is set; otherwise it is cleared.

P (PARITY) FLAG, BIT 2

0 = low byte even parity, 1 = low byte odd parity

The setting of the Parity flag depends on the low-order eight bits of a result. If 0, 2, 4, 6, or all 8 bits in a byte are set to 1, the Parity flag is cleared and parity is even. If the low byte has an odd number of bits set to 1 (1, 3, 5, or 7 bits), the P flag is set and parity in the byte is odd.

C (CARRY) FLAG, BIT 0

0 = no carry from high bit, 1 = carry

This flag indicates whether an addition or subtraction has caused a carry or borrow from the high bit of the destination into what would be the next higher bit if there were one.

Control Flags

Control flags affect string instructions only.

D (DIRECTION) FLAG, BIT 10

0 = auto-increment string instructions, 1 = auto-decrement

The Direction flag controls the "direction" of string operations. When the D flag is cleared these operations process strings from low memory up towards high memory. When this flag is set strings are processed from high to low memory.

FLAT AND SEGMENTED MEMORY

On the 8086 addresses are formed by combining two values: a 16-bit long segment selector and a 16-bit long offset. To calculate an address the segment selector is shifted left four places and the two numbers are added, yielding a 20-bit address (See Figure 2-6.)

Programs and data areas are each made up of pieces called segments. For example, at the start of a major piece of the program CS (the Code Segment) is loaded with a value which is the starting location of the program's code. The Instruction Pointer is set to 0. As successive instructions are executed the value in the Instruction Pointer is incremented to point to new instructions in succession. JMP and CALL instructions cause a new value to be loaded into the Instruction Pointer, but the Code Segment is unchanged. Because the value in the 16-bit Instruction Pointer is perforce limited to a maximum of $2^{16}-1$, the maximum size of a program segment is 2^{16} bytes, or 64 Kb. In fact, the maximum size of any segment under this addressing scheme is 64 Kb. A program or data area can continue over several segments, but the 20-bit address size limits memory to 2^{20} bytes, or 1 Mb. Programmers using DOS are further limited by that operating system's inability to address more than 640 Kb.

These segment and memory-size limitations are a severe problem for many 8086 programs and systems. A data structure, for instance, can be no larger than 64 Kb without stretching across two or more segments. Any program that accesses such a large structure must constantly check

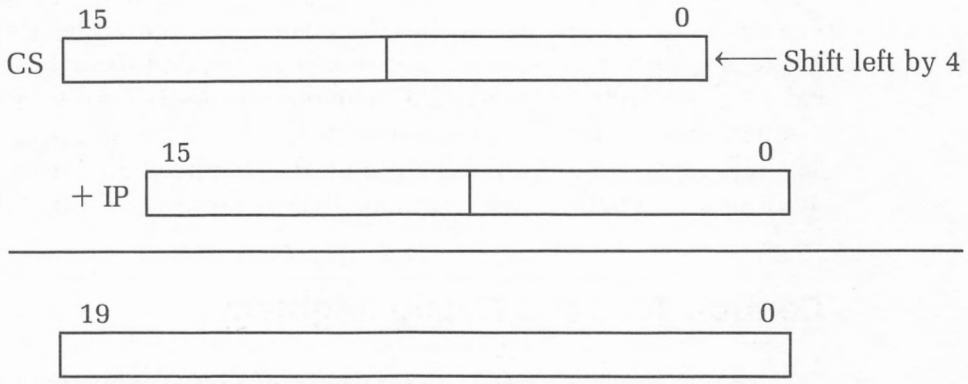


Figure 2-6. Adding Selector and Offset to Get 20-bit Address

whether it is approaching a segment boundary and switch to a new segment at the appropriate time. This checking and switching greatly slows access to these large structures. Updating a video screen, for example, must be done very quickly to prevent visible flickering. A 1,024 by 1,024 pixel screen might need one byte to describe each pixel (on, off, blinking, and brightness are among the possible parameters for even a noncolor screen). This means that 1 Mb of RAM is needed to support the screen, far more than the 64 Kb limit of one data segment; this is the same as the amount of memory an 8086 can address at all, leaving no room in memory for programs or data!

Addresses for data are calculated by using the DS (or sometimes the ES) register and an offset; addresses for stack operations are calculated with the SS register and SP (the Stack Pointer). In each case the maximum segment size is 64 Kb.

The 80286 and the 80386 both have Real Modes, in which they operate this same way. In Protected Mode, however, things are different. On the 80386 we can take advantage of the larger 32-bit registers to implement a flat memory model.

In a flat memory model there are no segments (or if you prefer there is one large segment that holds everything). Memory is treated as one big, unbroken expanse. This is the memory model used by the 68000 and other popular microprocessors. To implement it on the 80386 we just set all the segment registers to 0. All the registers that are used as offsets when we calculate addresses (the Instruction Pointer (EIP), the Stack Pointer (ESP), and the other general registers) are 32 bits wide. Thus the most memory we can address is 2^{32} bytes, or 4 Gb; this is 4 billion bytes, or over 4,000 times more than can be addressed by the 8086.

If we wish we can also use a segmented memory model in 80386 Protected Mode, just as on the 8086. However, the address calculation method is different. Segment registers aren't added directly to offsets to calculate an address. Instead the segment register is used as a selector or pointer into a list of "segment descriptors." The descriptor contains several pieces of information about the segment, including the base address and length of the segment. This is described in detail in Chapter 5.

Control, Test, and Debug Registers

These registers aren't generally used by applications programs. However, they are important to understand because they are used to

support the operating system, coprocessors, debuggers, and other parts of the environment your programs are developed and run in.

There are four Control Registers that can only be accessed by variants of the MOV instruction. For example, MOV EAX, CRO will load EAX with the contents of CRO, the first control register; MOV CR3, EBX will load CR3, the last control register, with the contents of EBX. These variants of MOV can be used only at privilege level 0. CRO contains several flags of interest.

PG (PAGING ENABLE) FLAG, BIT 31

0 = no paging, 1 = paging on

When this flag is on the processor uses the paging tables, which are used for Virtual Memory and other purposes to determine what address to use. When it's off the paging tables are unused.

ET (EXTENSION TYPE) FLAG, BIT 4

0 = 16-bit (80287) coprocessor, 1 = 32-bit (80387) coprocessor

The setting of this flag tells the 80386 which type of coprocessor is available, a 16-bit 80287 or a 32-bit 80387. The 80386 uses a 16-bit protocol in the first case, a 32-bit protocol in the second.

TS (TASK SWITCHED) FLAG, BIT 3

0 = no task switch, 1 = task switched

When this flag is on, a task switch has just occurred. This flag affects coprocessor and other instructions.

EM (EMULATION) FLAG, BIT 2

0 = no emulation of coprocessor, 1 = emulate coprocessor

The ESC command is generally used to transfer control to the numeric coprocessor. If EM is set when ESC is executed an exception is generated to allow an exception handler to emulate the numeric coprocessor.

MP (MATH PRESENT) FLAG, BIT 1

0 = no coprocessor, 1 = coprocessor present

The 80386 tests this flag when executing a WAIT instruction. If set the TS flag is tested; if that is also set exception 7 is generated, which should cause the coprocessor to be made available.

PE (PROTECTION ENABLE) FLAG, BIT 0

0 = Real Mode, 1 = Protected Mode (includes Virtual 8086 Mode)

The setting of this flag indicates whether the processor is in Real Mode or Protected Mode. Note that Virtual 8086 Mode is a subset of Protected Mode.

All 32 bits of the second control register (CR1) are currently reserved by Intel. CR2 is used when paging is on; when a page fault occurs (typically because a needed page is not in memory) the linear address that triggered the fault is stored here. The upper 20 bits of CR3 are also used for paging; they hold the base address of the paging directory. The lower 12 bits of CR3 are undefined.

The debug registers (Figure 2-7) are a vital element in the advanced debugging capabilities of the 80386. The registers themselves are described briefly below. The Resume and Trap bits in the EFlags register, also used in debugging, are described above.

The four Debug Address Registers (DR0-DR3) contain addresses of breakpoints. The addresses can be either real addresses or indexes to the page tables, depending on whether or not paging is enabled. Since different tasks can use different paging tables, a bit in DR7 (see below) tells whether the addresses in DR0-DR3 apply to all tasks or to the current task only. The addresses are actual addresses and apply to the current task when the processor is in Real Mode, since paging and multitasking are both unavailable. Thus applications programs can use the debug registers directly.

Depending on the settings of the flags in DR7, the four addresses in DR0-DR3 can cause a break in execution when the data in them is executed, overwritten, or either read or written.

Registers DR4 and DR5 are reserved by Intel. Register DR6 is the Debug Status Register, and contains several bits of interest. The low-order bits, B0 through B3, indicate which set of conditions caused a break. Likewise BD, BS, and BT (bits 13, 14, and 15) indicate conditions in the debug registers, whether a single-step exception occurred, and whether the new TSS invoked by a task switch has its T bit set, causing a break on the attempt to switch to the task.

Register DR7, the Debug Control Register, helps turn debugging

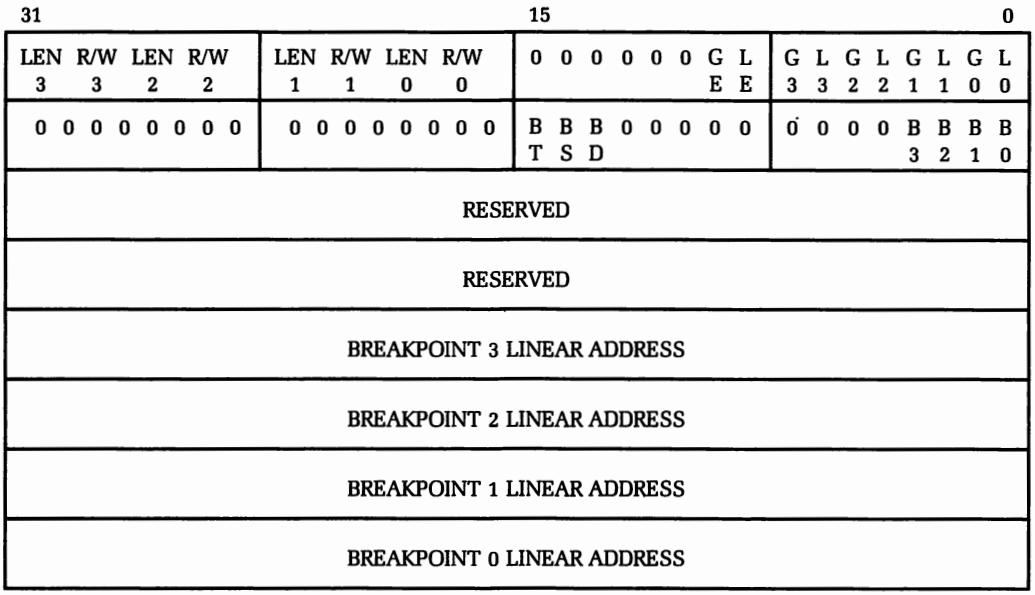


Figure 2-7. Debug Registers

features on and off. The fields LENO-LEN3 specify the length of the data item to be monitored at the address specified in D0-D3. R/W 0-3 tell under which conditions a given Debug Address Register will cause a break, on instruction execution only, on data writes, or on either reads or writes. If the value in one of the four R/W fields is 0 (break on instruction execution) the corresponding LEN field should also be 0 (no length specified).

The four bits G0 through G3 specify for each register (D0-D3) whether it is globally enabled (accessible by all tasks). L0 through L3 specify whether each of the address registers is locally enabled; that is, accessible specifically by the current task. L0-L3 are changed at each switch to a new task, but are overridden by a 1 in the corresponding flag (G0-G3).

Because of the pipelining feature of the 80386 the processor may cause a breakpoint by prefetching an instruction. Since several instructions in a row may be in the chip's pipeline, the offending instruction may even be one that would not normally have been executed. If LE or GE is set, prefetching is turned off, slowing execution but ensuring that only the instruction which is currently being executed can cause a break. LE is cleared at a task switch, but GE is not.

The two test registers are accessed by variants of the MOV instruction. They allow data to be written to and read from the Translation Lookaside Buffer, which is used to speed paging. They are of potential help in writing test programs for the hardware and even in writing optimization routines for paging. They are probably not of much use to the typical applications or systems programmer, however, so they won't be explained in more detail here.

Memory Management Registers

There are four registers that describe data structures used by an operating system to manage memory. They will be described briefly here and mentioned again elsewhere in the book. The important thing to know is that the 80386 can be running several tasks at once and that each task is made up of several segments; each segment has a descriptor that gives its size and other important information. Multitasking is described in Chapter 5.

IDTR The Interrupt Descriptor Table Register points to the table of entry points for interrupt handlers.

GDTR The Global Descriptor Table Register contains the descriptors of segments that are available to any of the (possibly many) tasks running on the computer.

LDTR The Local Descriptor Task Register contains the descriptors of segments that are available specifically to the currently running task.

TR The Task Register holds a copy of the descriptor of the current task and describes where the descriptor itself is stored in memory.

Data Types

The basic data types of the 80386 are those found on the chip itself—bits (the smallest unit of data for any computer), bytes (as used in AL, AH, and other byte-sized registers), 16-bit words (as in AX, BX and other 16-bit registers), and dwords (the size of all the physical registers, EAX, ESP, etc.). This can be confusing because a computer's "word size" is typically the size of its data bus and/or internal registers, and the 80386

is 32 bits throughout. However, a 16-bit quantity is still called a word for consistent usage throughout the 8086 family. Other data types are derived from these basic ones, either by interpreting the data in one of the basic types or by combining the basic types to form larger units.

When we deal with memory the 80386 actually reads and writes dwords (four-byte chunks), but particular bytes can be updated independently of the dword they're in. To the programmer memory is made up of a long series of bytes, each with its own address.

Below is a list of the data types supported by the 80386. Much of this list will be a review to the experienced assembly language programmer, but it's important that these terms be completely understood.

The bit is the smallest unit and can have a value of 0 or 1. When examining bits in the EFlags register these values can be thought of as False (0) and True (1).

The byte is the basic organizational unit, consisting of eight bits. It can be seen in several different ways:

1) An unsigned number from 0 to 255. Every bit from the lowest (bit 0) to the highest (bit 7) represents a successive power of 2. These values can also represent ASCII characters, with a letter, numeric digit, or special character assigned to each of the values from 0 to 255.
2) A signed number from −128 to +127. The high-order bit (bit 7) represents the sign of the number, a high-order 1 represents a negative number, and the remaining bits represent the magnitude of the number. Two's complement notation is used to determine the bit pattern that represents negative numbers (see Chapter 1). It's important to note that the bit pattern 11111111 represents either the unsigned number 255 or the signed number −128, depending only on how it's interpreted.
3) A BCD digit. This is just like an unsigned number but with values restricted to the range 0 through 9. A single byte can also contain two packed BCD digits, each taking up four bits (a nibble). Commands found on all 8086-family processors support packed BCD arithmetic.

The 16-bit word can, of course, contain two bytes, each of which may have any of the formats above. Looked at as a single quantity the word can be interpreted in two ways:

1) A 16-bit unsigned number ranging in value from 0 to 65,535 (64 K). Again, every bit represents a power of two; the bits are numbered from 0 (lowest) to 15 (highest).
2) A signed number, with bit 15 treated as a sign bit, from −32,768 to 32,767. If bit 15 is 1 the number is negative; if bit 15 is 0 the number is positive.

A 32-bit dword, besides containing any of the data types above, has several possible types of values. These values have extra importance because the 80386 has 32-bit registers, a 32-bit data bus, and a 32-bit address bus. Thus the 32-bit data types represent the largest numbers and addresses the 80386 can handle at top speed. They have two possible interpretations:

1) A 32-bit unsigned number ranging in value from 0 to $2^{32} - 1$ (about 4 billion). This many bytes is called 4 gigabytes, and is the most memory the 80386 can address directly. 32-bit unsigned numbers are also called near pointers, as they can point to any location within a given segment.
2) A 32-bit signed number, with bit 31 treated as a sign bit, ranging in value from 2^{32} to $2^{32} - 1$, a range between about −2 billion and +2 billion. Again, a high bit value of 1 indicates a negative number.

There are several data types made up by repeating one of the basic data types:

1) The bit field is a series of contiguous bits (each next to the others) within a dword. The bit field starts anywhere within a dword, but doesn't extend past the end of the dword it starts in. The maximum length of a bit field is anywhere from 1 to 32 bits, as determined by its starting point in the dword. The EFlags register is made up of a series of bit fields.
2) The term "string" is used in a general sense to mean a contiguous series of bits, bytes, words, or dwords. A bit string may contain up to $2^{32} - 1$ bits; other strings may contain up to $2^{32} - 1$ bytes. A character string is a string of bytes each containing an ASCII character.

The 8087 uses two additional large data types not normally found in 80386 programming: a 64-bit (8-byte) quadword (qword) and an 80-bit long quantity called a tbyte.

Addressing Modes

Some instructions in an assembly-language program operate on data that is contained in the instruction itself (immediate data), or on data that is in one of the processor's registers, and thus do not access memory. The addressing mode in which data from a register is used directly is called Register Operand Mode; when data is contained in an instruction itself it is called Immediate Operand Mode. The command MOV AX, 7FH gets its destination operand from Register Operand Mode and its source (second) operand by using Immediate Operand Mode.

When a computer program reads data from memory or writes data to memory, it must tell the computer what location in memory to use. The simplest assembly-language statements name the location to be used directly. For instance, ADD AX, ANADDRESS refers to a previously specified location in memory, ANADDRESS.

Effective programming requires that there be several different ways to name a memory location. There are four possible elements within an assembly language statement that are combined to determine the address to be used. This calculated or "effective address" is combined with the appropriate segment register to determine the address that is actually used. These four elements are:

1) Base. This is the contents of one of the general purpose registers. It is used as a starting point; other elements are added to it to form the effective address. The beginning location of an array might be placed in a register for use in this way.
2) Displacement. The displacement is the address of a location in memory. The displacement can be 8, 16, or 32 bits long.
3) Index. As with based addressing, the contents of a general purpose register are used to produce an effective address. When 16-bit operands are in force SI and DI are used for indexing; with 32-bit operands any register but ESP can be used.
4) Scaled. If the index is a 32-bit quantity it may also be multiplied by 2, 4, or 8. This is useful in accessing arrays with elements of fixed size.

An effective address is calculated by combining the elements listed above in a straightforward way:

EA = Base Register + (Index Register * Scale factor) + Displacement

The formula is simplified when not all elements are used. If there is no index, for instance, the EA is just Base + Displacement. Here are some examples of how each mode is used and how the different modes can be combined:

1) Direct addressing includes both register and immediate operands; no memory access is needed.
2) Memory uses a displacement only. The displacement is typically indicated by a label; the distance from the current instruction to the label is calculated and used as the offset to use in accessing memory.
3) Indirect uses the contents of a register as an address. The register being used for indirection is placed in brackets. For example, MOV AX, BX will move the contents of BX into AX. MOV AX, [BX] will move the contents of the memory location indicated by BX's contents into AX.
4) Based addressing allows a constant to be added to the value in a register and the resulting sum used as the effective address. The register + displacement expression is placed in brackets. MOV AX, [BX + 4] will take the contents of BX, add 4 to it, and then use the result as the effective address. The quantity at this address will be moved into AX.

   The remaining modes all use indexing. The index is placed in brackets after the base it will be added to; only indexes can be scaled (multiplied by 2, 4, or 8).
5) Indexed addressing adds a direct address and an index to produce the effective address. ADD ECX, TABLE[SI] will add SI to the value of TABLE to calculate an effective address. The number at the effective address will be added to ECX.
6) Index combines a base in a register and an index in a register. MOV ECX, [EDX] [EAX] will add the contents of EAX to the contents of EDX and use the result to calculate an effective address. When 32-bit operands are used indexes can also be scaled.
7) Scaled index mode allows the index to be multiplied by 2, 4, or 8 before it is used; ADD ECX, TABLE[ESI * 8] will multiply the value in ESI by 8, add it to the address represented by TABLE, and use the result as an effective address. This is especially effective if the data items being used are 8 bytes long.

Besides the modes listed here, any combination of bases, indexes (whether scaled or not), and displacements can be used. Address calculation is done while the processor is doing other things, so even the most complex address takes no extra time to calculate. There is one exception; if a base, an index, and a displacement are all present, the instruction as a whole will take one extra clock to execute.

Interrupts and Exceptions

An interrupt is an alteration in the normal flow of program execution. Interrupt handling capabilities are built into the 80386; the first thing the processor does when executing an instruction is check for interrupts. The interrupt (which can come from outside the program or be caused by executing an instruction) causes a table of interrupts to be accessed; the table points to a routine that serves as an "interrupt handler," presumably doing whatever is necessary to resolve the interrupt. There are three types of interrupts on the 80386. The first is called an "exception" and is caused by the execution of an instruction. The INT instruction, for example, actually causes an exception when executed. The handling of interrupts varies among different operating systems; A list of real mode exceptions is provided in Table 2-1.

There are four types of exceptions: aborts, traps, faults, and programmed exceptions (also called software interrupts). An abort is the most serious; it is an exception that doesn't allow the instruction that caused it to be identified, nor does it allow restarting the program that triggered it. This is the case when some element of the computer system behaves unexpectedly, meaning that not only is the current instruction causing problems, but previous results may have been incorrect. Hardware errors (not I/O device errors) and inconsistent values in system tables are often reported with aborts.

A trap is an exception reported just after the offending instruction has finished executing. A fault, on the other hand, is reported just before an instruction begins to execute or during execution of the instruction.

A software interrupt is caused by an instruction designed to cause an exception some or all of the time. INT 3, INT n, INTO, and BOUND are the instructions designed to sometimes or always cause exceptions.

Hardware interrupts are of two main types: maskable interrupts (which can be recognized or not depending on the setting of the I flag) and nonmaskable interrupts (which must always be recognized). Interrupts are recognized by the 80386 at the very beginning of executing each

Number	Description	Instruction
0	Divide error	DIV, IDIV
1	Debug exception	Any
2	Non-maskable interrupt	
3	Breakpoint	INT 3
4	Overflow	INTO
5	Array boundary check	BOUND
6	Invalid opcode	Undefined opcode (includes LOCK with wrong instruction)
7	Coprocessor not available	ESC, WAIT
8	Interrupt vector too large for table	INT
9	Reserved	
10	Invalid TSS	Any task switch
11	Segment not present	Many
12	Stack boundary crossed (Stack access, offset less than 0 or greater than 64 Kb)	PUSH, POP, PUSHF, POPF, PUSHA, POPA
13	General protection (Data or code access, offset > 64 Kb, or instruction length greater than 15 bytes)	Many
14	Page fault (page not present)	Many
15	Reserved	
16	Coprocessor error	ESC, WAIT
17-31	Reserved	
32-255	Available for maskable interrupts	

TABLE 2-1. Interrupt ID Numbers and Descriptions

instruction; a non-masked hardware interrupt will cause an interrupt handler to be called with no intervention from the program, regardless of what the program is doing at the time. Maskable interrupts are signalled via the 80386's INTR pin; non-maskable interrupts are signalled via the NMI pin, although these interrupts actually are masked during handling of a previous non-maskable interrupt.

As Table 2-1 shows, all non-maskable interrupts are assigned identifier 2, but maskable interrupts can use any identifier from 32 through

255. These numbers are assigned using an external interrupt controller (for example, an Intel 8259A Programmable Interrupt Controller). Each of these can handle up to eight interrupts, and one controller can have another controller as one of its inputs. This is known as using "cascaded" interrupt controllers. The arrangement of the controllers is transparent to the programmer, who knows an interrupt only by its number. Interrupts of all types are little different on the 80386 than on previous members of the 8086 family. One of the big differences is the problems that can be caused by one task altering the I flag in a multitasking system, as described in Chapter 6.

Particular interrupts are used for important functions like coprocessor handling and protection. We will discuss interrupts as they arise in connection with other subjects, but will not cover their use in detail.

CHAPTER 3

Guide to the Instructions

This chapter is meant to be a road map to the standard instructions of the 80386. As such it serves two purposes. First it will serve to help those readers who are unfamiliar with the 8086 family of processors. Its

second purpose is as a reference to the instruction set by function (Chapter 4 presents the instructions in alphabetical order).

The instructions are grouped according to function. Each instruction is described briefly and common usages are given. Most of the functional groups include an example that demonstrates the operation of several of the instructions from the group.

Central to the understanding of the 80386 instructions is the concept of operands. Simply put, an operand is the data value that the instruction "operates" on. Although each instruction has its own operand structure, there are many common features that can be easily summarized.

An operand is normally classified as either a source or a destination operand (depending on whether the instruction gets data from it or stores data into it). In general a source operand may be a register, a memory location, or an immediate value. Destination operands may be only registers or memory locations. It is unusual for an instruction to accept memory locations as both source and destination operands.

Usually the operands may be of any size (byte, word, or dword). For most instructions with multiple operands all operands are the same size.

For a full description of the function and operands for each instruction, see Chapter 4.

DATA MOVEMENT INSTRUCTIONS

This group of instructions performs the essential task of moving data from place to place within the computer system. Data can be moved between two registers, between a register and memory, and between a register or memory and the stack.

MOV The MOV (move) instruction moves a single data item from one place to another.

XCHG XCHG (exchange) is used to swap the contents of two registers or swap the contents of a register with that of a memory location. This operation is often used for synchronization of multiple processes because it can not be interrupted by another device using the data bus.

PUSH The PUSH instruction copies its source operand onto the top of the stack. PUSH is most often used to place parameters on the stack

before calling a procedure. The instruction is also useful for temporarily storing something on the stack.

POP The POP instruction removes the top entry from the stack and moves it to the destination operand. The instruction is used to restore values saved on the stack by a PUSH instruction.

PUSHA and PUSHAD The PUSHA and PUSHAD (push all) instructions are used to place all eight general registers on the top of the stack. The difference is that PUSHA pushes the 16-bit registers and PUSHAD (PUSH All Doubleword) pushes the 32-bit registers. These instructions are very useful in the preamble of a procedure to save the state of the registers for the caller.

POPA and POPAD The POPA and POPAD (pop all) instructions are the complements of the PUSHA and PUSHAD instructions. Their function is to restore all eight general registers to the values they had before the corresponding push all. Again, the POPA operates on the 16-bit registers and the POPAD operates on the 32-bit registers.

The example below shows an easy and a hard way to swap the contents of two registers. Notice how the stack is used in Case 1 to avoid the use of a memory location.

State Before

EAX	00000117
EBX	00002F3E

Case 1

```
PUSH    EAX       ; Save contents of EAX on the stack.
MOV     EAX,EBX   ; Get EBX contents into EAX.
POP     EBX       ; Get old EAX contents into EBX.
```

Case 2

```
XCHG    EAX,EBX   ; Swap contents of EAX and EBX.
```

State After (Both Cases)

EAX	00002F3E
EBX	00000117

STANDARD ARITHMETIC INSTRUCTIONS

These instructions are used to perform arithmetic on signed and unsigned integers. This is the most common type of arithmetic required in programs, so this group is a cornerstone of the instruction set.

ADD The ADD instruction is used to add two operands, placing the result in the first (destination) operand.

SUB The SUB (subtract) instruction is used to subtract one operand from another. The destination is replaced with the old value of the destination minus the value of the source.

INC The INC (increment) instruction performs just like an ADD with an immediate operand of one, except that the carry flag is unaffected. This instruction is most commonly used in loops to add one to the value of an index.

DEC The DEC (decrement) instruction performs just like a SUB with an immediate operand of one, except that the carry flag is unaffected. This instruction is most often used in loops to subtract one from the value of an index or loop counter.

MUL The MUL (unsigned integer multiply) instruction is the simpler of the 80386's two multiply instructions; it takes only one operand, the source. The other two operands of the multiply are implied by the size of the source operand.

IMUL The other multiply instruction is IMUL (signed integer multiply). It is a much more versatile and complex instruction. The instruction has four basic forms, based on the number and type of operands. There are forms with one, two, and three operands.

DIV The DIV (unsigned division) instruction performs an unsigned divide. For a division there are really four operands: the dividend, the

divisor, the quotient, and the remainder. Only the location of the divisor is specified. All the other operands are located implicitly, based on the size of the divisor.

IDIV The IDIV (integer division) instruction performs identically to the DIV instruction except that IDIV performs a signed divide.

NEG The NEG (two's complement negate) instruction is used to change the sign of its single operand. The operand may be either a register or memory location.

CMP The CMP (compare) instruction is identical to the SUB instruction in all respects except one: it does not store the result. The instruction is used to compare two numbers in preparation for one of the conditional jump instructions (see below).

ADC The only difference between the ADC (add with carry) instruction and the ADD instruction is that the former adds the value of the carry bit into the sum. This feature makes it useful for multiple precision arithmetic.

SBB There is only one difference between the SBB (subtract with borrow) instruction and the SUB instruction: SBB subtracts the value of the carry bit from the difference. This feature makes it useful for multiple precision arithmetic.

To appreciate the difference between ADD and ADC consider the following example:

State Before

EAX	FFFFFFF0	MEMLOC	000027DA
EDX	00002F3E	MEMLOC+4	00000117

Case 1

```
ADD   EAX,MEMLOC      ; Add first pair of DWORDS.
ADD   EDX,MEMLOC+4    ; Add second pair of DWORDS.
```

State After (Case 1)

EAX	000027CA	MEMLOC	000027DA
EDX	00003055	MEMLOC+4	00000117

Case 2

```
ADD   EAX,MEMLOC      ; Add least significant DWORD.
ADC   EDX,MEMLOC+4    ; Add most significant DWORD.
```

State After (Case 2)

EAX	000027CA	MEMLOC	000027DA
EDX	00003056	MEMLOC+4	00000117

The following example illustrates the difference between the signed and unsigned multiply instructions:

State Before

AL	84
DL	12

Case 1

```
MUL   AL,DL   ; 132*18 = 2376.
```

State After (Case 1)

AX	0948

The value in AX is 948H or 2376 decimal.

Case 2

```
IMUL   AL,DL   ; −124*18 = − 2232.
```

State After (Case 2)

AX	F748

The following example illustrates the difference between the signed and unsigned divide instructions:

State Before

AX	04BF
DL	9A

Case 1

```
DIV   AL,DL   ; 1215/154 = 7, Remainder 137.
```

State After (Case 1)

AX	8907
AH	89
AL	07

AL contains the quotient; AH contains 89H or 137 decimal, the remainder.

Case 2

```
IDIV   AL,DL   ; 1215/−102 = −11, Remainder 93.
```

State After (Case 2)

AX	5DF5
AH	5D
AL	F5

AL contains F5H or −11 decimal, the value of the quotient; AH contains 5DH or 93 decimal, the remainder.

DATA CONVERSION INSTRUCTIONS

This group of instructions is used to convert one type of data into another. Most of the instructions deal with signed numbers, but there is one that helps with unsigned data.

MOVSX The MOVSX (move with sign extension) instruction moves a source operand into a larger destination operand by extending the sign bit of the source throughout the upper part of the destination.

MOVZX MOVZX (move with zero extension) is similar to MOVSX except that the upper part of the destination is set to zero.

CBW The CBW (convert byte to word) instruction is like a limited form of MOVSX. It can only deal with the AL register as the source and the AX register as the destination. The AH register is filled with the sign bit from AL, converting the signed byte in AL into a signed word in AX.

CWDE A similar instruction is CWDE (convert word to dword extended). This instruction converts the signed word in AX into a signed dword in EAX by extending the sign bit of AX throughout the upper half of EAX.

CWD The CWD (convert word to dword) instruction has the same goal as CWDE but behaves differently. It is also meant to convert a signed word to a signed dword, but the result is placed into two registers (unlike CWDE). The DX register is filled with the sign bit from AX.

CDQ The last of the conversion instructions is CDQ (convert dword to qword). This instruction fills the EDX register with the sign bit of the EAX register.

The following example illustrates the instructions in this group:

State Before

EAX	FACEFFF0
EBX	5A5A5A5A
ECX	ACACACAC
EDX	12345678

MEMLOC	FF3E
MEMLOC2	07DB

Instructions

```
MOVSX   EAX,MEMLOC    ; Move (sign extend) from MEMLOC.
MOVZX   EBX,MEMLOC    ; Move (zero extend) from MEMLOC.
MOVSX   ECX,MEMLOC2   ; Move (sign extend) from MEMLOC2.
MOVZX   EDX,MEMLOC2   ; Move (zero extend) from MEMLOC2.
```

State After

EAX	FFFFFF3E
EBX	0000FF3E
ECX	000007DB
EDX	000007DB

MEMLOC	FF3E
MEMLOC2	07DB

DECIMAL ARITHMETIC INSTRUCTIONS

The 80386 does not directly provide support for BCD arithmetic, but the decimal adjust instructions (when used in conjunction with the normal arithmetic instructions) do provide this support. The decimal adjust instructions come in two basic types; The ASCII adjust instructions handle one BCD digit per byte, and the decimal adjust instructions handle BCD digits packed two per byte.

AAA, AAS, AAM, and AAD There is one ASCII adjust instruction for each of the four basic arithmetic operations. Three of these are used after the normal arithmetic instruction to adjust the result so that it still contains a valid BCD digit. They are AAA (ASCII adjust after addition), AAS (ASCII adjust after subtraction), and AAM (ASCII adjust after multiplication). The other instruction, AAD (ASCII adjust before division), is performed before the division to prepare the operands for division.

DAA and DAS There are only two decimal adjust instructions, one for addition and one for subtraction. They are used in the same way as their ASCII adjust counterparts.

The following example uses the instruction AAA to illustrate the use of the ASCII adjust instructions:

State Before

AX | 0009 |

Instructions

```
ADD   AL,8   ; 9 + 8 = 11 H (17).
AAA          ; Adjust to BCD format.
```

State After

AX | 0107 |

LOGICAL INSTRUCTIONS

This group of instructions is used for two purposes: it provides the operations used on Boolean (sometimes called logical) values, and also provides ways to access and manipulate bit fields within bytes, words, or dwords.

AND The AND instruction performs a logical "and" function on its two operands. This instruction is useful for clearing a bit field to zero.

OR The OR instruction performs a logical "or" function. The common usage with bit fields is to set the value of the field by first using AND to clear the field and then using OR to store the desired value.

NOT The NOT instruction (one's complement negation) takes only a single operand. This instruction reverses all the bits in its operand.

TEST The TEST (logical compare) instruction performs identically to AND except that the result is not stored. This instruction is useful for testing the value of a bit field for zero (or non-zero).

XOR The XOR instruction provides the logical "exclusive or" function. The main value of the instruction with bit fields is to complement only the bits in a given field.

SETxx The SETxx instructions are used to remember the result of some comparison. The "xx" is replaced by one of a large set of possible comparison conditions. If the comparison is true then the destination operand is set to one. The destination is set to zero if the comparison is false.

The following example shows the effects of the four major logical instructions using identical operands. If you have difficulty understanding the results of these operations, try converting the hexadecimal numbers to binary. This exercise will give you a better appreciation for the operations involved.

State Before

AX	AAAA

MEMLOC1	C3C3
MEMLOC2	C3C3
MEMLOC3	C3C3

Instructions

```
AND   MEMLOC1,AX
OR    MEMLOC2,AX
XOR   MEMLOC3,AX
NOT   AX
```

State After

AX	5555

MEMLOC1	8282
MEMLOC2	EBEB
MEMLOC3	6969

SHIFT AND ROTATE INSTRUCTIONS

These instructions provide a way to move bits within one of the standard data types. The main usefulness of these instructions lies in two areas; they can be used to speed multiply and divide operations on integers, and can also be used in the implementation of bit fields.

SHR and SHL The SHR (shift right) and SHL (shift left) instructions perform logical shifts. They shift zeros into one end of the operand so that data bits "fall off" the other end.

SAR and SAL The two arithmetic shift instructions are SAR (right) and SAL (left). The left arithmetic shift is identical to a left logical shift. The SAR, however, shifts in a copy of the sign bit on the left. This feature allows for division of signed numbers by powers of 2.

ROR and ROL The rotate instructions do not cause any data bits to be shifted out of the operand. Instead, the ROR and ROL instructions perform a circular shift. In a circular shift any bits shifted out one end of an operand are shifted back in the other end.

RCR and RCL The next pair of shift instructions are RCR and RCL, the rotate through carry shifts. In these instructions a bit shifted out one end of the operand is placed into the carry flag, and the old value of

the carry flag is shifted into the other end of the operand. The most common use of these instructions is for multiple precision shifts (of all three kinds).

SHRD and SHLD The last of the shift instructions are the double precision shifts SHRD and SHLD. These are also useful for multiple precision shifts, but work in a different way than the rotate through carry instructions. These instructions have three operands: source, destination, and shift count. The destination is shifted and bits "fall off" one end, but the vacated bits are filled with bits from the source. The source is unchanged.

The following example illustrates the difference between the normal rotate instructions and the rotate through carry instructions:

State Before

AX	6699
FLAGS	0202

Case 1

```
ROR   AX,1   ; Sets overflow, carry unchanged.
```

State After (Case 1)

AX	B34C
FLAGS	0A02

Case 2

```
RCR   AX,1   ; Clears overflow, sets carry.
```

State After (Case 2)

AX	334C
FLAGS	0A03

BIT INSTRUCTIONS

This instruction group is very useful for manipulating single bits. It includes all the required operations for setting and testing individual bits.

BT, BTS, BTR, and BTC The BT (bit test) instruction simply places the value of a specified bit into the carry flag. Three other instructions perform the same function but also allow you to store a value in the addressed bit. BTS sets the bit to one; BTR resets the bit to zero, and BTC complements the bit.

BSF and BSR The two bit scan instructions, BSF (bit scan forward) and BSR (bit scan reverse), find the first 1 bit in an operand from either the least significant end (BSF) or the most significant end (BSR). Either instruction will be useful in implementing bit maps. BSR can also be used to compute base two logarithms.

The following example shows one of the bit scan and one of the bit test instructions (note that the 22 in the BTC instruction is a decimal number):

State Before

EAX	004037BF
EBX	00000000
ECX	00000000
FLAGS	0242

Instructions

```
BSR   EBX,EAX   ; EBX = bit number of most significant one.
BTC   EAX,22    ; Change bit 22 from one to zero.
BSR   ECX,EAX   ; EBX = bit number of most significant one.
```

State After

EAX	000037BF
EBX	00000016
ECX	0000000D
FLAGS	0203

FLAG CONTROL INSTRUCTIONS

This group of instructions allows the programmer to monitor and control the 80386 flags. There are two sub-groups; one contains instructions that address an individual flag, and the other contains instructions that address the flags as a whole.

There are seven instructions that reference an individual flag. The CLD and STD instructions clear and set the flag which controls the direction of the string operations. The CLI and STI instructions clear and set the interrupts enabled flag. The CLC and STC instructions clear and set the carry flag. In addition, the CMC instruction is also provided to complement the value of the carry flag.

The LAHF instruction loads the least significant byte of the flags into the AH register. The SAHF instruction, on the other hand, stores the data from the AH register into the low byte of the flags register. There are four instructions that move flags to and from the stack. The PUSHF and PUSHFD instructions push the FLAGS and EFLAGS registers (respectively) onto the stack. The corresponding pop operations are performed by POPF and POPFD.

The following example demonstrates both the usage of the stack to save the flags and one of the flag setting instructions.

State Before

EAX	000037BF
FLAGS	0A93

Instructions

```
PUSHF           ; Save current flags on the stack.
SUB     EAX,EAX ; Clear OF, SF, AF, CF; set ZF and PF.
POPF            ; Get old flags back from the stack.
STD             ; Set the direction flag.
```

State After

EAX	00000000
FLAGS	0E93

STRING INSTRUCTIONS

It is often necessary to deal with large chunks of data at a time. The instructions in the String Instruction group are helpful in doing just that. Many common loops that would otherwise require several instructions can be implemented in just a single string instruction.

Each of the string instructions normally performs a single operation. The operation can be a move, a compare, a load, a store, or a scan. The (E)SI register is used to point to the source operand, and the (E)DI register is used to point to the destination operand. Both these registers are updated to point to the next string element after the instruction. The

instructions are most useful, however, when used with one of the repeat prefixes.

REP, REPE, REPZ, REPNE, and REPNZ The repeat prefixes allow the string instruction that follows to be executed a controlled number of times. The REP prefix causes the following string instruction to be executed the number of times in the ECX register. The REPE and REPZ prefixes are synonyms. They cause the instruction to repeat until either the count is exhausted or the instruction ends with the Z flag set to zero. The other two repeats (REPNE and REPNZ) work just like REPE except that the loops they are used in exit when the Z flag is set to one.

MOVS The MOVS instruction simply moves a string of data from one spot in memory to another. Care must be taken to achieve the desired result when the two strings overlap. The direction flag is useful for this situation.

CMPS The CMPS instruction is used to compare two strings. The REPE or REPNE prefix is used to terminate the comparison at the point of interest.

STOS The STOS instruction is useful for filling a string with a constant value. It transfers the contents of the appropriate part of the EAX register into each element of the string.

SCAS The SCAS instruction is similar to STOS in the way that CMPS is similar to MOVS. SCAS compares each element of the string with the contents of the suitable part of the EAX register. Normally REPE or REPNE are used with this instruction.

LODS The LODS instruction is atypical of this group in that it is not usually used with a repeat prefix. It merely loads the next element of the string into some part of the EAX register.

The following example uses the MOVS instruction to illustrate the general principles of the string instruction group:

Assumptions

STRING 1 is at offset 100H in segment 151H.
STRING 20 is at offset 105H in segment 151H.
The D (direction) flag is set to 0 (forward).

State Before

ESI	00000200
EDI	00000300
DS	0151
ES	011C

STRING1	A1	A2	A3	A4	A5
STRING2	FF	FF	FF	FF	FF

Instructions

```
LEA       SI,STRING1   ; Prepare the source address.
LES       DI,STRING2   ; Prepare the destination.
MOV       ECX,5        ; Get the count for the string move.
REP MOVSB              ; Move the byte string.
```

State After

ESI	00000105
EDI	0000010A
DS	0151
ES	0151

STRING1	A1	A2	A3	A4	A5
STRING2	A1	A2	A3	A4	A5

FLOW CONTROL INSTRUCTIONS

It would be impossible to write any useful programs without some way to change the normally sequential flow of instruction execution. The instructions in this group provide the needed tools for controlling the flow of program execution.

JMP The most basic transfer of control is the unconditional jump (JMP) instruction. Most commonly the programmer supplies a label after

the JMP that indicates which instruction is to be executed next. The assembler then generates the correct form of JMP based on its knowledge of the location of the named statement.

Jxx The next type of control transfer is the conditional jump, the Jxx instructions (where "xx" is replaced by a code for the particular condition to be tested). See Chapter 4 for an explanation of all of these codes. Generally the programmer does a comparison (or any instruction which sets some flags) and then issues a conditional jump causing the program to change locations based on a bit in the FLAGS register.

CALL The CALL instruction is used to implement procedures (sometimes called subroutines, subprograms, or functions). It transfers control in a manner similar to JMP. However, just before the transfer, information is placed on the stack so that the CALLed procedure can cause instruction execution to resume at the instruction following the CALL. The RET (return) instruction is used to perform this action.

LOOP, LOOPE, LOOPZ, LOOPNE, and LOOPNZ The loop instructions help in the construction of many program loops. The LOOP instruction is used to execute a loop the number of times specified by the ECX register. The LOOPE and LOOPZ instructions have the additional constraint that the loop will terminate when the zero flag is zero. The LOOPNE and LOOPNZ instructions are similar to LOOPE and LOOPZ except that a zero flag of one will cause the loop to terminate.

INT, INTO, IRET, and IRETD The final form of control transfer is the interrupt. These will be only briefly treated here, as they are covered in more detail in chapter 5. The INT instruction causes a system interrupt procedure to be activated. The INTO instruction causes interrupt procedure 4 to be activated if the overflow flag is set. Interrupt procedures are normally provided by the operating system. The interrupt procedure will return to the calling program with one of the return from interrupt instructions (IRET or IRETD).

The flow control instruction example deviates from the form of our other examples. This one consists of a program fragment written in a higher-level language and the assembler code that might be generated by the compiler. Note that the high-level code is intentionally *not* the best possible code to sum the odd numbers between 1 and 10.

The high level code:

```
SUM = 0;
DO INDEX = 1 TO 10;
  IF (INDEX AND 1) = 1 THEN SUM = SUM + INDEX;
END;
```

The generated assembly code.

```
        MOV   BX,0          ; Set SUM to zero.
        MOV   AX,1          ; Initial value of INDEX.
        MOV   ECX,10        ; Repeat count for the loop.
LOOP_START:
        TEST  AX,1          ; Test the low order bit.
        JZ    SKIPIT        ; Jump if number is even.

        ADD   BX,AX         ; Sum if number is odd.
SKIP_IT:
        INC   AX            ; Increment number.
        LOOP  LOOP_START    ; Loop until done.
        MOV   SUM,BX        ; Store the result.
```

HIGH-LEVEL LANGUAGE SUPPORT INSTRUCTIONS

The instructions in this group are normally generated by high-level language compilers, not assembly language programmers. They make the compiler writer's job easier and provide fast hardware support for some common operations.

BOUND The BOUND instruction is meant for use in checking array bounds. It takes a value, a lower limit, and an upper limit as operands. If the value does not lie within the limits an interrupt occurs.

ENTER and LEAVE The ENTER and LEAVE instructions are complementary instructions meant to lessen the set-up time at the beginning of a procedure. ENTER sets up the stack at the beginning

of a procedure to ease access to arguments, local variables, and variables in related procedures. LEAVE restores the stack in preparation for the RET instruction.

PROCESSOR CONTROL INSTRUCTIONS

The instructions in this group control the activity of the central processor. The most common usage is to help interface with other processors (like an 80287 floating point coprocessor) in the system.

ESC The ESC prefix informs the 80386 that the instruction that follows is not meant for the 80386 at all, but rather must be handled by a coprocessor.

WAIT The WAIT instruction causes the processor to stop executing instructions. The processor will not resume until the bus signal BUSY is inactive. This operation is used to wait for a coprocessor to finish the computation of a needed result.

LOCK The LOCK prefix causes the system bus to be dedicated to the 80386 for the duration of the prefixed instruction. This capability is used to prevent bus contention at critical times in multi-processor systems. NOTE: the LOCK prefix in the 80386 has a number of restrictions. Please check the full instruction definition in Chapter 4 if you are porting a program that uses LOCK from an older processor to the 80386.

NOP The no-operation instruction, NOP, is an instruction that does nothing. It is therefore not commonly used. It does, however, have some usefulness in debugging.

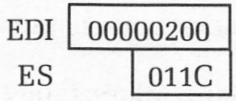
HLT The final instruction in this group is HLT, the halt instruction. This instruction causes the processor to stop all activity until reset. HLT is normally only used as a drastic measure.

ADDRESS MANIPULATION INSTRUCTIONS

In this group are the instructions used for loading address pointers. Although there are six instructions in the group, they really are of only two types. The first type consists of the single instruction LEA. LEA loads a register with the offset of a specified memory location. It is useful for

loading an index register. The second type is the full pointer load type. These instructions load both a segment register and an index register. The segment register is loaded with the segment selector of a memory location, and the index register is loaded with the offset of the memory location. The instruction mnemonics are of the form Lxx, where "xx" is the name of one of the segment registers.

State Before



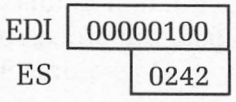
Case 1

```
MOV   ES,MEMLOC    ; Load selector into ES.
LEA   EDI,MEMLOC   ; Load offset into EDI.
```

Case 2

```
LES   EDI,MEMLOC   ; Load selector into ES, offset into EDI.
```

State After (Both Cases)



THE TRANSLATION INSTRUCTION

XLAT The final instruction does not fit any of the other groups, so it gets a group of its own. The XLAT instruction performs a table look

up translation. That is, it assumes that AL contains a byte index into a table pointed to by (E)BX. The byte in AL is replaced by the table entry that it points to. This instruction is useful in character code translation applications, and in command parsing and other character classification applications.

SUMMARY

This chapter has been a brief summary of the 80386 instruction set. It should also prove useful as a reference when you know what you need to do but are unsure of which instruction to use. You should now be prepared for chapter 4, which consists of a more detailed look at each of the instructions.

CHAPTER 4

The 80386 Instruction Set

So far we've tried to take a gentle approach to some intimidatingly technical subjects. With its high-level capabilities, the 80386 has the most advanced ideas in computing burned into its silicon; yet, because it is compatible with earlier Intel efforts, it also embodies a history of microcomputing. Programming the chip to its capabilities therefore requires drawing on a large body of knowledge, some of which is summarized in the preceding pages. Having a firm grasp of this general knowledge enables the programmer to act as an architect, designing a program or set of programs that will fit the user's needs while getting the most out of the hardware resources at hand.

A large amount of programming time is spent grinding out solutions to problems after the design work is done—get the data into the chip, shift

or multiply or add it, then store or print it. In this effort the chip's instruction set is the programmer's toolbox. Knowing when to use a rubber mallet (perhaps a Shift instruction) and when to use a 5-pound sledge (like the Multiply instruction) can make a big difference in how the final product looks and works.

This chapter is a list of tools. The instruction set is the closest most of us get to the actual inner workings of the chip. A working knowledge of factors like timing and at least a general understanding of instruction sizes is important to anyone wanting to be a competent assembler programmer. Working assembler programs can be written by those who know only a small subset of the instruction set and are aware that assembler programs run faster than high-level language programs, but don't know why. This approach may be sufficient when writing small program loops in assembler to be combined with large high-level language programs. However, those who want to go beyond this minimal approach and write medium or large programs that get the most out of the machine would do well to spend some time with this chapter, trying previously unused instructions in small programs and putting some thought into how to write efficient code.

The efficiency comes from knowing how quickly each of a program's routines will execute. For a very simple example, let's say you're converting an 8088 program to the 80386 and optimizing it where possible. The 8088 program at one point has to multiply a number by 10, and does so by shifting the number left by three bits (same as multiplying by 8) and then adding the original number to the result twice (this equals multiplying by 10). At 14 clocks this multistep process is far more efficient than using the 8088's MUL instruction (which takes 70 or more clocks to execute) *if* all the numbers are in registers. On the 80386 the same multistep process takes 9 clocks, but using an IMUL instruction would require only 10 clocks. Given the simplicity and directness of using a single instruction, the IMUL is a better choice.

Figure 4-1 shows the fact that the 80386's instruction set is "flat" in terms of execution time; that is, the slower and the faster instructions aren't too far apart in speed. Of course it also illustrates the fact that the 80386 is much faster than the 8088. On the 80386 IDIV (or Integer DIVision), for instance, only takes 6 times as long as a CMP (CoMPare). On the 8088, by contrast, an IDIV takes 19 times as long as the CMP. On the older chip a good way to speed up your program was to replace a slow instruction with a bunch of fast instructions (for example, doing three ADDs instead of MULtiplying by 3). Another good trick was to use

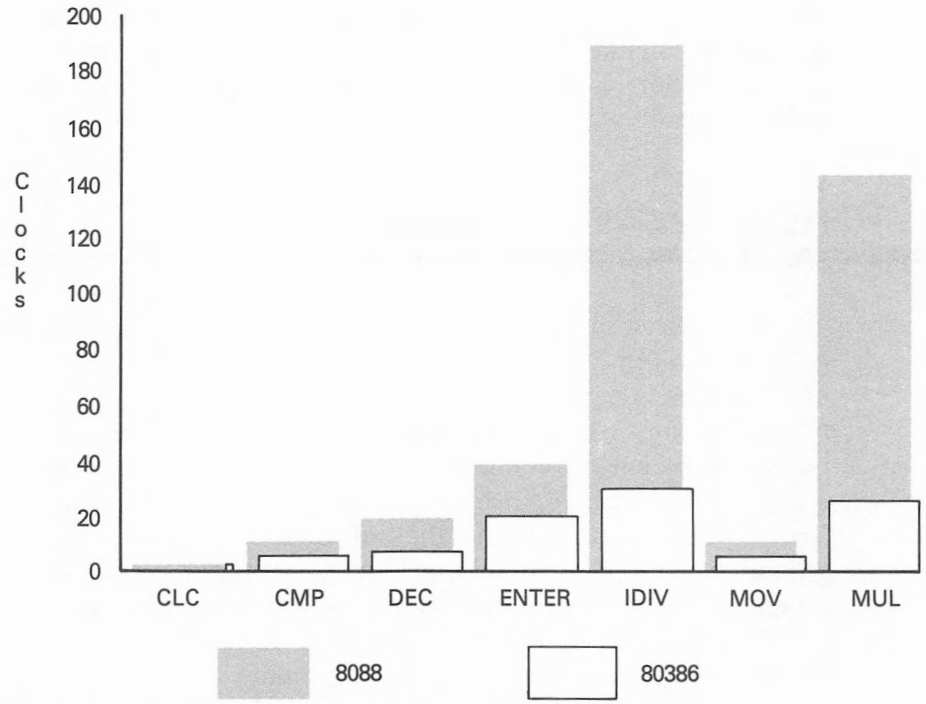


Figure 4-1. Some Instruction Execution Times, 8088 vs. 80386

a CMP or two before something slow like an IDIV. The result of the CMP might help you avoid doing the IDIV. Of course, this took more programmer time and resulted in lengthy, hard-to-follow code. On the 80386, however, it's not worth avoiding something like an IDIV, since executing even a few other instructions will take just as long. In general, the 80386 will reward using the most logically appropriate instruction and using the entire instruction set, not just a few of the faster instructions.

This brings us to the other benefit of knowing the information given in this chapter—compactness. Well-written assembler programs are compact for several reasons. They always use the best instruction for the job, based on a detailed knowledge of the instruction set and the time needed to execute each instruction. Also, the actual size of an instruction is a factor. An instruction that adds two registers is smaller than an instruction that adds a register and an immediate value, because the immediate value takes up space in the object code.

A diligent programmer will use the information in this chapter as a guide and then test time- or size-critical programs to see how long they

run and how big they are. If you're writing a subroutine that will be used by others, including timing and size information with the routine's documentation will make your program much more usable by later programmers.

HOW ASSEMBLY LANGUAGE WORKS

Your assembly language program is not really the lowest possible level of programming. First, the assembler translates your program into the "formats" described below. The instruction POPA, for instance, is translated into a byte containing 61 hex. Other instructions are longer because they need to include memory addresses or register designations.

These assembled instructions make up "object code," which is in "machine language." Using the information in this chapter you can predict exactly what machine language code a given instruction will produce once it's assembled. Compilers produce machine language object code too, but it tends to be less efficient than assembler-based code because compilers aren't as smart as humans.

Simpler chips have very simple assembly languages and assembler programs. The 6502, for instance, has only two general registers. Very specific instructions like TXY (transfer the value in the X register to the Y register) are used and each instruction has one machine language counterpart (or perhaps a few, each using a different addressing mode). The 80386, on the other hand, has many registers and many addressing modes. To transfer any register to another (or to memory, or to transfer from memory to the register) we use the MOV instruction followed by the registers we wish to move information from and to. This means that the MOV instruction has many machine language counterparts, depending on the type of move (register to register in this case) and which addressing modes are used. Different versions have different formats and lengths. The important difference between assembly language and a high-level language is that every instruction in assembler translates into a single machine language instruction, although which one is used depends on such things as which register is used. In high-level language most single instructions translate into several machine language instructions.

Just how integral, how "built-in" is the assembly language of a processor? The secret's in the "microcode" included with most modern processors. Microcode is a "program" running in the chip that decodes

the object code presented to it and performs the desired functions. Microcode is a tool of the hardware designers that allows them to simplify and modularize their chip design to make it easier to work with.

WHAT IS AN INSTRUCTION'S FORMAT?

An instruction's format is just the translation of the instruction into 0's and 1's. For much assembly language programming there's no need to be concerned about formats. However, when trying to read hex dumps knowledge of instruction formats is necessary, and the only way to patch an existing program is by using knowledge of the formats to change the instructions themselves.

Most 80386 instructions have many different formats because they're really many different instructions. To the chip, adding a number from memory into one in a register is much more complicated than adding two values in registers. The prefetcher, for instance, has to try to steal bus cycles to go get the operand from memory while some other command is executing, which can affect prefetch efficiency. The various 80386 assemblers are very powerful and keep the programmer a healthy distance from the bit-level operations of the chip. This is usually a good thing. To better understand the formats used by each instruction we'll look at the elements a machine language has to have to make a computer get things done.

First there is the instruction itself, often called the opcode (short for operation code). Are we asking the computer to move information? To add two numbers together? To transfer program execution to a new location? The opcode tells the computer what we're going to do. In several cases the opcode is all that the computer needs to get the needed task done. For instance, the instruction STC (SeT the Carry bit) requires only an opcode since its operation is implicit in the instruction itself. The instruction format is very simple: 0F9H. At one byte this is as small as any instruction on the 80386, and executes in a mere two clocks.

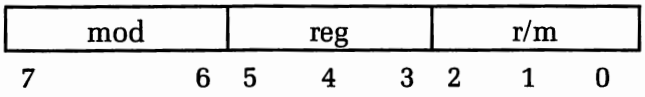
Besides the opcode the computer needs to know where to find its operands—the parameters or arguments needed to complete the instruction. There are two techniques used by 80386 instructions to encode this information. The first is simple, the second more complex.

Some instructions use bits in the opcode to specify a register operand. The PUSH command (which puts a number on the stack) has

one operand. If the number being PUSHed is a register, a number telling which register to push is included in the opcode: 50H pushes the AX register, while 51H pushes the CX register. The last three bits tell which register to push. Again, these instructions are one byte long and take two clocks to execute.

Instructions that have more than one operand or an operand in memory use the more complicated scheme to specify their operand location. For these instructions there is an another byte following the opcode byte that is called the ModRM byte. This byte is formatted as follows:

ModRM Byte



The three fields in the ModRM byte each play a different role in specifying the operands of the instruction.

The "r/m" field occupies the least significant three bits of the byte. It specifies either one of the eight general registers or one of 24 addressing modes, depending on the value of the "mod" field.

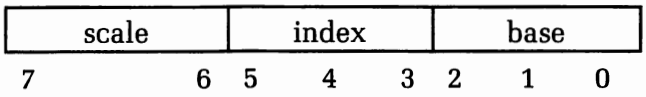
The "reg" field is contained in the next three bits in the byte. For instructions with two operands, it specifies a register number. For single operand instructions this field acts as an extension of the opcode, further specifying the exact nature of the instruction.

The most significant two bits of the byte are occupied by the "mod" field. Of its four possible values, one indicates that the "r/m" field contains a register number. The other three values are used to select one of three groups of eight addressing modes.

Now to return to our example of the PUSH instruction. If the number being pushed is in RAM, we need to tell the computer where to find the number that it's going to push. In this case, the instruction goes like this: 0FFH mod 6 r/m. The instruction itself is the first byte (all 1's), plus three bits in the middle of the second byte (110). The "mod r/m" bits (two for mode, three for register/memory) tell the addressing mode and where in memory to get the number from. Then, the actual address in RAM to get the number from is given. With all the work that goes into decoding this instruction, it's surprising that the instruction executes in only 5 clocks—on the 8088 it takes more than 16 clocks.

Some of the addressing modes mentioned above require even more information to successfully locate the desired operand. When a ModRM byte specifies an addressing mode that requires this information, it is followed by another byte. This extra byte is called the SIB byte for Scale, Index, and Base. Its format is:

SIB Byte



As in the ModRM byte, the three fields in the SIB byte each specify a different item of information about the addressing mode.

The "base" field is contained in the least significant three bits of the byte. It specifies the register that will be used as the base register.

The index register is specified in the "index" field, which occupies the next three bits in the SIB byte.

The "scale" field resides in the most significant two bits of the byte. It gives the scale factor used for indexed addressing modes.

Speaking of software evolution, an experienced programmer can see many evidences of change over time just by looking at the instruction formats for the 80386. For instance, the new Bit Test instructions (which test and change individual bits) might be expected to be as short as possible. After all, these instructions will be much-used and they perform straightforward functions. However, these instructions actually all start with the same first byte (0FH) and it takes another byte to specify which instruction is being used, plus more bytes for address and immediate data. Compared to similar instructions the extra byte seems to make these instructions take an extra cycle to execute.

Precise directions for decoding the information given in the format section included with each instruction are given just before the start of the listings of the instructions themselves.

TIMING INFORMATION

The timing data given with the instructions ranges from very precise to surprisingly vague, considering that we're dealing with assembly

language (which gives us the most precise control over the machine that we can hope to get). Of course, instructions like CLC (Clear the Carry bit), which always do the same thing in exactly the same way, always take the same amount of time (two clocks in this case). Instructions that can use either registers or memory as operands vary tremendously in the amount of time they take to execute. In a way, such an instruction is actually several different instructions depending on where its operands are coming from. Thinking of it as a single instruction is a powerful tool for making programming easier but distances us from what's actually happening inside the chip. It takes a fair amount of work to adjust your thinking down to the machine level when trying to calculate how long an instruction will take to execute. The information given for each instruction gives the various forms an instruction can take and timing information for each.

Also complicating matters is the fact that the timing information given below isn't precise; most programs take about 5 percent longer to execute than they should based on the timing information given below. The details of this are interesting and depend directly on the way the 80386 is designed, but if you don't often have cause to time your programs or write timing-critical loops you might wish to skip the explanation below.

The reason the chip runs slow is that the times given in this chapter assume that pipelining is always in effect. Pipelining, as described previously, is the ability to fetch one instruction, decode another, and execute a third all at the same time. This is aided by the existence of queues, which hold several fetched instructions awaiting decoding and several more decoded instructions awaiting execution. The timing information given below assumes that pipelining is in full effect at all times. In actuality, however, some instructions take less time to execute than they do to fetch and decode. If several of these instructions occur in a row the queues slowly empty as the instruction unit charges through the simple code. Once the queues empty, the instruction unit has to wait for at least part of the fetching and decoding time until it can do its own thing. When this happens the chip slows down (See Figure 4-2). Any instruction that transfers program control (such as JMP, CALL, RET, etc.) will also empty out the queues since the fetcher doesn't know where the next instruction is coming from until it's needed.

The upshot of this problem (which is not a "bug" in the 80386, but actually just a necessary exception to its usual full-speed, pipelined execution) is that the times given below aren't exact. If the instruction

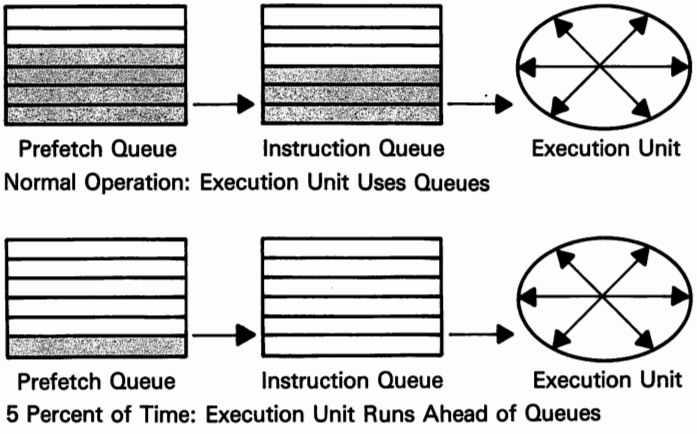


Figure 4-2. Queues and Execution Speed

queue happens to be empty when the execution unit is ready, the next instruction will take longer than usual to execute. How much longer depends on how long it takes to get a decoded instruction into the queue. On average the occasional breakdown of pipelining adds a little more than 5 percent to the time it takes a given program to execute. If your system has a cache memory this percentage will be less, as fetching will execute even more quickly than usual. Of course, there's not really such a thing as an "average" program, so you'll have to compare your own programs to the clock counts given below if you wish to see what effect all this is having on your own programs.

Knowing exact timing information is unlikely to be necessary for most programs most of the time. The "average" programmer (again, there's really no such thing) can get by just fine by adding 5 percent to the calculated execution time for a given program. It's almost never worthwhile to try and cut down this overhead by reordering instructions.

Writing a program that needs to execute in a precise amount of time can range from frustrating (for a simple timing loop) to impossible (if your program has to handle highly variable input data). Adding to this frustration is the variation in times a given instruction can take to execute, depending mostly on where it gets its operands. If your program should take 129 cycles to execute and it's taking 142, is the problem an extreme example of pipelining breakdown or did you make a mistake in adding up the cycle counts given below?

Really extinguishing any hopes of precisely timing some programs is the difference in execution speeds between Real Mode and Protected Mode. The programmer doesn't necessarily know when coding what mode the program will be run in, so he or she can't predict program execution time if some of the instructions run at different speeds in different modes. Luckily the speeds are only different when loading pointers (LDS, LES, LFS, LGS, and LSS) and making CALLs, JMPs, and RETs across segments (the conditional jumps like JNE don't go across segments). Few programs make these types of jumps in timing-sensitive sections, or make enough of these jumps that the added execution time in Protected Mode is a problem.

THE 80386 AND OTHER iAPX 86 CHIPS

The 80386 programmer who is converting programs from earlier iAPX family chips, or whose programs may later be converted to one of the earlier chips, needs to know which instructions work on which microprocessors. Even the 80386-only programmer will want to adapt sections of code written for the 8086/88, saving himself much work in the process. The earlier chapters on the iAPX 86 family included some information about which instructions were new with each new chip.

The instruction descriptions below tell in detail which instructions (and even which modes of a given instruction) were first seen with which chip. Every new iAPX 86's instruction set is a superset of the preceding chips; which allows upward compatability of programs. In some cases the chips are assembly language compatible but not object code compatible. In these cases "all you need to do" to make a program run is to recompile its source code on the new machine. The reason for the quotes around "all you need to do" is that it's easy to get the source for your own programs, but try getting a copy of the source code of Lotus 1-2-3 so you can recompile it!

Few new instructions have been added with each new chip and operating mode. About nine out of ten of the 80386's instructions are unchanged from the 8086/88 (except that when run on the 80386 they can handle 32-bit operands). Of the remaining one tenth, some have had new addressing modes or protection considerations added, and a few are brand new.

Figure 4-3 shows the relationship between the various pieces of the 80386's instruction set. Most of the instructions come unchanged from the 8086/88. In fact, these instructions are called the Basic Instruction Set of the iAPX 86 family. Several new instructions of the type used in everyday programming were added for the 80186/188, and that chip's operation is considerably faster than its predecessor's.

The 80286 had no new instructions of the type programmers use every day, but it did have new operating system type instructions for memory management and program protection. Very few programmers have learned to use these new instructions because few operating systems or applications programs have taken any advantage of the 80286's capabilities. Indeed, the 640 Kb address space limit (which MS-DOS imposes on the IBM PC) still hasn't been lifted as the first 80386-based machines become available. A whole cottage industry of standards committees and programming tips and tricks grew up among those seeking to get beyond this limit.

The 80286 was an odd man out in other ways as well. For instance, its onboard clock wasn't duplicated on the 80386. The 80386, on the other hand, is a direct descendant of the 8086 which incorporates the lessons learned with the 80286. It has several new everyday-type instructions (as represented by the Extended Instruction Set in 80386 Real Mode) mostly dealing with bit manipulation (something Intel's chips had always fallen short in compared to the competition). It also integrates 32-bit registers and data paths completely throughout the system while still making byte and word manipulation easy.

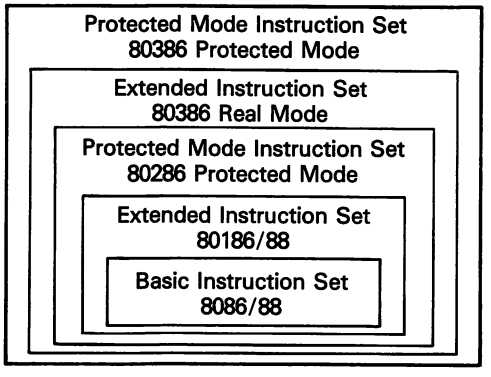


Figure 4-3. The 80386 Instruction Set and Other iAPX 86 Instructions

Finally, the 80386 has a complete set of memory management, protection, virtual memory, and debugging instructions accessible mostly through Protected Mode (as represented by the System Control Instruction Set, 80386 Protected Mode).

THE INSTRUCTIONS

The remainder of this chapter is a list of the instructions available in 80386 assembler language, with instruction formats, clock counts, and other information given for each instruction. The instructions are arranged alphabetically except when extremely similar or identical instructions can be grouped on a single page. This grouping makes reading through the descriptions easier.

Perhaps the most important sections for someone concerned about the instruction formats are the pages just before the instructions start. These include all the references needed to understand the abbreviations in the formatting information.

Figure 4-4 presents a list of logical functions and their effects on bit patterns. Several instructions implement one of these functions directly, and many rely on these definitions in their descriptions.

The instruction descriptions have a consistent format throughout. There are several sections that appear for each instruction. The abbreviations and formats used are described below.

The first line of each instruction's description contains three separate pieces of information. On the left is the instruction mnemonic normally used by the assembler. In the middle is a descriptive title for the instruction. On the right is the first iAPX processor on which the instruction appeared.

The first major section contains the instructions' various opcodes,

P	Q	P and Q	P or Q	P xor Q	Not P	Not Q
1	1	1	1	0	0	0
1	0	0	1	1	0	1
0	1	0	1	1	1	0
0	0	0	0	0	1	1

Figure 4-4. Logical Operations on Nibbles P (1100) and Q (1010)

the corresponding assembler format, and the execution time for the instruction in clocks. Any of the string instructions that can be repeated have two timing columns, one for with and one for without a REP prefix. Certain other instructions (JMP and CALL, for instance) have an additional column to clarify the type of the instruction used with that opcode.

Values that may be used in the opcode column and their meanings are:

hh Two hexadecimal digits specify a byte containing exactly that value.

[r] A standard mod r/m byte (see above). By implication this may be followed by an SIB byte (also above) and/or a memory offset.

[n] Where "n" is digit 0 through 7, this is a mod r/m byte with the register field set to the digit. In this case the digit is really an extension of the opcode.

ib An immediate value of byte length.

iw An immediate value of word length.

id An immediate value of dword length.

db An 8-bit signed value to be added to (E)IP to obtain the target for a jump or call type instruction.

dw A 16-bit signed value to be added to (E)IP to obtain the target for a jump or call type instruction.

d A displacement (offset) of the operand in memory within a specific segment. May be 16 or 32 bits.

dd A 32-bit signed value to be added to EIP to obtain the target for a jump or call type instruction.

pd A 32-bit pointer. The first 16 bits are a segment selector and the second 16 bits are the offset within that segment.

pp A 48-bit pointer. The first 16 bits are a segment selector and the next 32 bits are the offset within that segment.

The format column uses many of the same codes as the opcode column to represent instruction operands. These codes have the same meaning in both columns. This column also sometimes uses specific register names or explicit numbers to represent the operands. In addition the following codes are also used in the format column:

r/mb The operand is the contents of the byte register or memory location specified by the mod r/m byte.

r/mw	The operand is the contents of the word register or memory location specified by the mod r/m byte.
r/md	The operand is the contents of the dword register or memory location specified by the mod r/m byte.
rb	The operand is the contents of the byte register specified by the mod r/m byte.
rw	The operand is the contents of the word register specified by the mod r/m byte.
rd	The operand is the contents of the dword register specified by the mod r/m byte.
mb	The operand is the contents of the byte at the memory location specified by the mod r/m byte.
mw	The operand is the contents of the word at the memory location specified by the mod r/m byte.
md	The operand is the contents of the dword at the memory location specified by the mod r/m byte.
mw:w	The operand is a pointer at the memory location specified by the mod r/m byte. The pointer contains 16 bits of segment selector and 16 bits of segment offset.
mw:d	The operand is a pointer at the memory location specified by the mod r/m byte. The pointer contains 16 bits of segment selector and 32 bits of segment offset.

The clocks column contains the instruction execution time in machine cycles (or clocks). If there are two values separated by a "/" then the first is the time required for register operands and the second is the time required for memory operands. Some of the entries contain formulas which are normally explained just below the table. However, the transfer-of-control instructions usually contain an entry of the form "7+m." The "m" in this formula refers to the number of components in the next instruction executed. A component is defined as one of the following: each prefix byte, each opcode byte, any mod r/m byte, any SIB byte, any displacement value, or any immediate value.

The next section describes the flags affected by the instruction. A blank under a flag name means no change. A "0" indicates that the flag is cleared to zero. A "1" shows that the flag is set to one. A "U" is used when the instruction leaves the flag in an undefined state. An "S" shows that the flag is set or cleared depending on the result of the instruction.

The next section gives the pseudocode, a semi-formal description of

the operation of the instruction. This an algorithmic description borrowing forms from higher-level languages.

The function performed by the instruction is described in detail in the operation section. Any clarification of timings, opcodes, or operands is also given here.

All exceptions that can be generated by the instruction are listed in the exceptions section. The modes column uses a single letter to represent the processor mode for each exception. A "P" stands for Protected Mode, an "R" represents Real Mode, and a "V" is used for Virtual 8086 Mode.

The user notes section contains any extra information about the instruction. Common usage, things to beware of, and compatibility issues are some of the things often found in this section.

Finally, each instruction has an example to illustrate its usage. Please note that these examples are not always samples of good coding practice. They are designed to demonstrate what a particular instruction does.

AAA ASCII Adjust AL for Add 8086

Opcode	Format	Clocks
37	AAA	4

Flags

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0			U				U	U	0	S	0	U	1	S

Pseudocode

```
If (lower nibble of AL > 9) OR (Aux Carry = 1) THEN
  increment AL by 6
  set upper nibble of AL to 0
  increment AH by 1
  set Carry Flag and Aux Carry Flag
ELSE
  reset Carry Flag and Aux Carry Flag
END IF
```

Operation

AAA changes AL to an unpacked decimal number.

Exceptions

None

User Notes

This instruction should be used after adding BCD digits. It converts the results of the addition to BCD. Most programmers won't use this instruction much. BCD arithmetic is explained in Chapter 1.

AAA itself only handles binary to BCD conversion. To convert AL to ASCII use OR AL,48.

Example

```
MOV   AX,8   ; Loads a BCD 8 into AL and clears AH.
ADD   AL,6   ; Adds a BCD 6 to AL giving 14 (decimal).
AAA          ; AX contains 104 (hex) or 14 BCD, both CF and AF
             ; are 1.
```

AAD ASCII Adjust AL Before Division 8086

Opcode	Format	Clocks
D5 0A	AAD	19

Flags

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0			U				S	S	0	U	0	S	1	U

Pseudocode

Set AL to (AL + (10 × AH))
Set AH to 0

Operation

AAD converts a number from unpacked BCD to binary. The greatest possible value of an unpacked BCD value in a 16-bit register is 99_{10} (a 9 in each byte of the register), which is 63_{16}. This easily fits in AL, so AH is set to 0 after the conversion.

Exceptions

None

User Notes

This instruction should be used before dividing unpacked BCD digits. Most programmers won't use this instruction much. BCD arithmetic is explained in Chapter 1.

The instruction itself only handles BCD to binary conversion. To convert AL from ASCII to BCD use AND AL,15.

Example

```
MOV   AX,405H   ; Loads dividend (a BCD 45) into AX.
MOV   BL,3      ; Loads divisor (a BCD 3) into BL.
AAD             ; Converts BCD into binary, result is 45 (decimal).
IDIV  BL        ; Does the division, result is 15 (decimal).
AAM             ; AX contains 105 (hex) or 15 BCD.
```

AAM ASCII Adjust AL after Multiply 8086

Opcode	Format	Clocks
D4 0A	AAM	17

Flags

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0			U				S	S	0	U	0	S	1	U

Pseudocode

```
Divide AL by 10
Set AH to quotient      (Tens digit of result)
Set AL to remainder     (Ones digit of result)
```

Operation

Any number less than 100_{10} will fit easily in the AL register, which can hold any number up to 128_{10}. AAM converts a number less than 100 which is in AL to an unpacked BCD number in AX.

Exceptions

None

User Notes

This instruction should be used after multiplying unpacked BCD digits. The result of the MUL is always less than ($9 \times 9 =$) 81, so the assumption that

the number in AAL is less than 100 works out fine. Most programmers won't use this instruction much. BCD arithmetic is explained in Chapter 1.

The instruction itself only handles BCD to binary conversion. To convert AL from ASCII to BCD use AND AL,15.

Example

See example for AAD.

AAS ASCII Adjust AL after Subtract 8086

Opcode	Format	Clocks
3F	AAS	4

Flags

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0			U				U	U	0	S	0	U	1	S

Pseudocode

```
IF (lower nibble of AL > 9) OR (Aux Carry = 1) THEN
  decrement AL by 6
  set upper nibble of AL to 0
  decrement AH by 1
  set Carry and Aux Carry Flags
ELSE
  reset Carry Flag and Aux Carry Flag
END IF
```

Operation

AAS changes AL to an unpacked decimal number and adjusts it, using the Aux Carry Flag to decide how much work is needed. The Aux Carry or the size of AL's lower nibble indicates whether a decimal carry was needed.

Exceptions

None

User Notes

This instruction should be used after subtracting BCD digits, and converts the results of the subtraction from binary to BCD (taking the effects of the subtraction into account). Most programmers won't use this instruction much. BCD arithmetic is explained in Chapter 1.

AAS itself only handles binary to BCD conversion. To convert AL to ASCII use OR AL,48.

Example

```
MOV   AX,205H   ; Loads a BCD 25 into AX.
SUB   AL,8      ; Subtracts a BCD 8 from AL giving 0FD (hex).
AAS             ; AX contains 107 (hex) or 17 BCD, both CF and
                ; AF are 1.
```

ADC Add With Carry 8086

Opcode	Format	Clocks
14 ib	ADC AL,ib	2
15 iw	ADC AX,iw	2
15 id	ADC EAX,id	2
80 [2] ib	ADC r/mb,ib	2/7
81 [2] iw	ADC r/mw,iw	2/7
81 [2] id	ADC r/md,id	2/7
83 [2] ib	ADC r/mw,ib	2/7
83 [2] ib	ADC r/md,ib	2/7
10 [r]	ADC r/mb,rb	2/7
11 [r]	ADC r/mw,rw	2/7
11 [r]	ADC r/md,rd	2/7
12 [r]	ADC rb,r/mb	2/6
13 [r]	ADC rw,r/mw	2/6
13 [r]	ADC rd,r/md	2/6

adcb
adcw
adcl
adcb
adcw
adcl

Flags

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0			S				S	S	0	S	0	S	1	S

Pseudocode

IF (source operand has fewer bits than destination) THEN
  sign-extend source operand
END IF
Add source operand to destination, place result in destination operand
Add CF to destination, place result in destination operand

Operation

ADC adds two numbers whose values are given as operands of the instruction, plus the value in the Carry bit. The first operand is overwritten by

the result, while the second operand is unchanged. A good translation of an ADC instruction into English might be "Add operand number 2 into operand number 1, then add 1 more if needed."

Exceptions	Modes	Reasons
#GP(0)	P	Result in nonwritable segment
#GP(0)	P	Illegal memory effective address in CS, DS, ES, FS, or GS segments
#SS(0)	P	Illegal address in SS segment
#PF(fc)	P V	Page fault
INT(13)	R V	Some part of operand is outside of address range 0 to 0FFFFH

User Notes

The ADC instruction is generally used when doing multibyte, multiword, or multidword additions, to let the Carry bit propagate as needed through the series of additions.

Understanding how the flags work for ADC can be very important, especially since the result of one of the flags after an addition is often used to decide whether to make a jump, or even as a parameter in a subroutine call. The ADD (below) works just like the ADC but ignores the value in the Carry bit. Anyone modifying an existing program should look at all ADC's following any changes to make sure the Carry bit is still set as the original programmer had assumed it would be.

Example

```
MOV   AX,956   ; Loads a 956 (3BC hex) into AX.
MOV   BX,373   ; Loads a 373 (175 hex) into BX.
ADD   AL,BL    ; Adds 0BC (hex) and 75 (hex) giving 31 (hex) with
               ; CF set.
ADC   AH,BH    ; Adds 3 and 1 giving 5 (because CF was set).
               ; AX now contains 1329 (531 hex), the sum of 956
               ; and 373.
```

ADD Add 8086

Opcode	Format	Clocks
04 ib	ADD AL,ib	2
05 iw	ADD AX,iw	2
05 id	ADD EAX,id	2
80 [0] ib	ADD r/mb,ib	2/7
81 [0] iw	ADD r/mw,iw	2/7
81 [0] id	ADD r/md,id	2/7
83 [0] ib	ADD r/mw,ib	2/7
83 [0] ib	ADD r/md,ib	2/7
00 [r]	ADD r/mb,rb	2/7
01 [r]	ADD r/mw,rw	2/7
01 [r]	ADD r/md,rd	2/7
02 [r]	ADD rb,r/mb	2/6
03 [r]	ADD rw,r/mw	2/6
03 [r]	ADD rd,r/md	2/6

Flags

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0			S				S	S	0	S	0	S	1	S

Pseudocode

IF (source operand has fewer bits than destination operand) THEN
  sign-extend source operand
END IF
Add source operand to destination, place result in destination operand

Operation

ADD simply adds two numbers whose values are given as operands of the instruction. The first operand is overwritten by the result, while the

second operand is unchanged. A good translation of an ADD instruction into English might be "Add operand number 2 into operand number 1."

Exceptions	Modes	Reasons
#GP(0)	P	Result in nonwritable segment
#GP(0)	P	Illegal memory effective address in CS, DS, ES, FS, or GS segments
#SS(0)	P	Illegal address in SS segment
#PF(fc)	P V	Page fault
INT(13)	R V	Some part of operand is outside of address range 0 to 0FFFFH

User Notes

Understanding how the flags work for ADD can be very important, especially since the result of the Carry flag after an addition is often used to decide whether to make a jump, or even as a parameter in a subroutine call. The ADC (above) even incorporates a carry from (perhaps) a previous ADD into a subsequent add. Many fascinating and obscure programming tricks can be executed using the flags; some of these were used to get around the lack of bit-test instructions on Intel chips before the 80386. Some programmers will find themselves unraveling this arcana while translating programs to the 80386, while others will reinvent it in translating back down to the earlier chips.

The ADD instruction should be used when possible in preference to the ADC to avoid complications caused by the often unpredictable state of the Carry bit.

Example

```
MOV   AX,956   ; Loads a 956 (3BC hex) into AX.
MOV   BX,373   ; Loads a 373 (175 hex) into BX.
ADD   AX,BX    ; AX now contains 1329 (531 hex), the sum of 956
               ; and 373.
```

AND Logical And 8086

Opcode	Format	Clocks
24 ib	AND AL,ib	2
25 iw	AND AX,iw	2
25 id	AND EAX,id	2
80 [4] ib	AND r/mb,ib	2/7
81 [4] iw	AND r/mw,iw	2/7
81 [4] id	AND r/md,id	2/7
20 [r]	AND r/mb,rb	2/7
21 [r]	AND r/mw,rw	2/7
21 [r]	AND r/md,rd	2/7
22 [r]	AND rb,r/mb	2/6
23 [r]	AND rw,r/mw	2/6
23 [r]	AND rd,r/md	2/6

Flags

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0			0				S	S	0		0	S	1	0

Pseudocode

```
REPEAT
  IF a bit in the destination operand is 1 and the corresponding bit in
    the source operand is 1 THEN
    leave the bit in the destination operand at 1
  ELSE
    clear the bit in the destination operand to 0
  END IF
UNTIL all bits in destination operand are checked
```

Operation

AND carries out a Boolean or "logical" AND on its two operands and leaves the result in the destination operand. This operation is depicted in Figure 4-4, which gives "truth tables" for all logical operations. A 1 can be regarded as T or True, while a 0 corresponds to F or False. A logical AND takes two bits and calculates a result using this rule: If both input bits are 1, then the output bit is 1, else the output bit is 0. The instruction AND simply does the same operation on all the bits in each of two operands; the leftmost bit in one operand is compared to the leftmost bit in the other, then the two bits one position to the right are compared, until all the bit pairs have been compared.

The use of a logical AND is much like the use of "and" in English; if you say, "The cat is black and white," then you are incorrect if the cat is neither black nor white, black but not white, or white but not black.

AND can only be used with two operands of the same size (same number of bits); otherwise the comparison would be meaningless for some of the bits in the longer operand.

Exceptions	Modes	Reasons
#GP(0)	P	Result in nonwritable segment
#GP(0)	P	Illegal memory effective address in CS, DS, ES, FS, or GS segments
#SS(0)	P	Illegal address in SS segment
#PF(fc)	P V	Page fault
INT(13)	R V	Some part of operand is outside of address range 0 to 0FFFFH

User Notes

AND is often used to clear certain bits or to do nibble arithmetic (as in BCD arithmetic); to clear the high nibble of a byte in AL, just use AND AL, 15. It is also used to implement bit tests and graphics on earlier iAPX 86's, but these functions can be implemented with generally more efficient bit-test commands on the 80386.

Example

```
MOV   AX,5963H    ; Loads a hex number into AX.
MOV   BX,6CA5H    ; Loads a hex number into BX.
AND   AX,BX       ; AX now contains 4821 hex.
```

BOUND Check Value in Range 80186

Opcode	Format	Clocks
62 [r]	BOUND rw,mw	10*
62 [r]	BOUND rw,md	10*

*Number of clocks is for when Interrupt 5 is not taken

Flags

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0									0		0		1	

Pseudocode

```
IF (first operand < word pointed to by second operand) OR
   (first operand > word after word pointed to by second operand)
     THEN
  INT 5
END IF
```

Operation

BOUND is used to check whether a value is outside or inside an acceptable range. If the value is inside the range then no action is taken. Otherwise an Interrupt 5 is triggered. This instruction is intended to be generated by high-level language compilers to check if an array index has gotten too high or too low. If it has the index is no longer within the bounds of the array.

The first operand, a register, holds the array index. The second operand is a pointer to a location in memory. At this location are two words, the lower and upper indices of the array. These numbers are the minimum and

maximum values of the index, not the lowest and highest points in memory occupied by the array.

If the index is less than the first word or greater than the second, an interrupt type 5 is generated. The BOUND instruction sets no flags and gives no indication of which bound check failed.

Exceptions	Modes	Reasons
#GP(0)	P	Illegal memory effective address in CS, DS, ES, FS, or GS segments
#SS(0)	P	Illegal address in SS segment
#PF(fc)	PV	Page fault
#UD	P	Second operand specifies a register, not memory
INT(5)	P R V	Bound test failed
INT(13)	R V	Some part of operand is outside of address range 0 to 0FFFFH

User Notes

The BOUND instruction is very useful for reassuring yourself that your program isn't looking off the end of an array. However, the actual implementation of the instruction is a bit troublesome. It might seem easier to set the Overflow Flag if the bound check failed, and perhaps also set the Sign Flag to indicate whether the index had gone too low or too high.

Instead, BOUND simply jumps to an interrupt when the check fails. Therefore, the interrupt must be set up in advance. It can handle the error completely or it can simply set a flag to indicate that an error has occurred, return control to the program, and let the main program code jump on the flag to an appropriate error handler.

Example

```
MOV     WORD PTR BND, 0          ; Sets lower bound to zero.
MOV     WORD PTR BND+2, 99       ; Sets upper bound to 99.
MOV     AX,100                   ; Loads value to check into
                                 ; AX.
BOUND   AX,BND                   ; Causes an Interrupt 5.
```

BSF Bit Scan Forward 80386

Opcode	Format	Clocks
0F BC [r]	BSF rw,r/mw	10+3n*
0F BC [r]	BSF rd,r/md	10+3n*

*n is number of zero bits skipped

Flags

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0								S	0		0		1	

Pseudocode

```
IF 2nd operand is zero THEN
  Set ZF
  Set 1st operand to an undefined value
ELSE
  Clear ZF
  Select low-order bit {bit 0} of 2nd operand
  DO WHILE selected bit = 0 {not executed if bit 0 = 1}
    Select next higher bit
  ENDDO
  Copy index of selected bit into 1st operand {index is 0-15, or 0-31}
END IF
```

Operation

BSF finds the first 1 bit in its second operand, a word or dword in a register or memory. It starts looking at bit 0 and stops when it finds a 1 bit. The index or bit position of this 1 bit is then placed in the second operand.

Exceptions	Modes	Reasons
#GP(0)	P	Illegal memory effective address in CS, DS, ES, FS, or GS segments
#SS(0)	P	Illegal address in SS segment
#PF(fc)	P V	Page fault
INT(13)	R V	Some part of operand is outside of address range 0 to 0FFFFH

User Notes

Many applications are made easier with the new bit manipulation instructions (for instance setting up allocation tables, wherein a bit's setting represents whether a piece of RAM or a sector on disk is in use). The BSF command allows the first nonzero bit (perhaps the first allocated sector on disk) to be quickly found. For example, a simple loop might find the first nonzero word or dword in a sector allocation table, then use BSF to find the first nonzero bit.

Example

```
MOV   BX,3CD0H   ; Loads value to scan into BX.
BSF   AX,BX      ; Sets AX to 4 and clears ZF.
```

BSR Bit Scan Reverse 80386

Opcode	Format	Clocks
0F BD [r]	BSR rw,r/mw	10+3n*
0F BD [r]	BSR rd,r/md	10+3n*

*n is number of zero bits skipped

Flags

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0								S	0		0		1	

Pseudocode

```
IF 2nd operand is zero THEN
  Set ZF
  Set 1st operand to an undefined value
ELSE
  Clear ZF
  Select high-order bit {bit 15 or 31} of 2nd operand
  DO WHILE selected bit = 0 {not executed if bit 0 = 1}
    Select next lower bit
  ENDDO
  Copy index of selected bit into 1st operand {index is 0-15, or 0-31}
END IF
```

Operation

BSR finds the first 1 bit in its second operand, a word or dword in a register or memory. It starts looking at the high bit and stops when it finds a 1 bit. The index or bit position of this 1 bit is then placed in the second operand.

Exceptions	Modes	Reasons
#GP(0)	P	Illegal memory effective address in CS, DS, ES, FS, or GS segments
#SS(0)	P	Illegal address in SS segment
#PF(fc)	P V	Page fault
INT(13)	R V	Some part of operand is outside of address range 0 to 0FFFFH

User Notes

Many applications are made easier with the new bit manipulation instructions (for instance setting up allocation tables, wherein a bit's setting represents whether a piece of RAM or a sector on disk is in use). The BSR command allows the first nonzero bit (perhaps the first allocated sector on disk) to be quickly found. For example, a simple loop might find the first nonzero word or dword in a sector allocation table, then use BSR to find the first nonzero bit.

Example

```
MOV   BX,3CD0H   ; Loads value to scan into BX.
BSR   AX,BX      ; Sets AX to 13 and clears ZF.
```

BT Bit Test 80386

Opcode	Format	Clocks
0F A3 [r]	BT r/mw,rw	3/12
0F A3 [r]	BT r/md,rd	3/12
0F BA [4] ib	BT r/mw,ib	3/6
0F BA [4] ib	BT r/md,ib	3/6

Flags

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0									0		0		1	S

Pseudocode

Use the 1st operand to locate a register or a memory address.
Use the 2nd operand to select a single bit (bit 0 is the low order bit).
Set the Carry Flag to the selected bit.

Operation

Bit Test allows the programmer to select any bit in memory and put it into the Carry Flag. The first operand is a register or memory address. The second operand is a bit number. It can be any unsigned integer up to 8 bits wide, but only the low order 4 or 5 bits are used depending on the size of the first operand. The Carry Flag is then set to this bit. The bit in memory is left unchanged.

Exceptions	Modes	Reasons
#GP(0)	P	Result in nonwritable segment
#GP(0)	P	Illegal memory effective address in CS, DS, ES, FS, or GS segments

#SS(0)	P	Illegal address in SS segment
#PF(fc)	P V	Page fault
INT(13)	R V	Some part of operand is outside of address range 0 to 0FFFFH

User Notes

Many applications are made easier with the new bit manipulation instructions (for instance setting up allocation tables, wherein a bit's setting represents whether a piece of RAM or a sector on disk is in use). The BT command allows quick access to any bit in the table. A more dramatic example to the average user is the improvement in updating large video displays made possible by the large segment size and quick bit access of the 80386.

Example

```
MOV   AX,3CD0H   ; Loads value to scan into AX.
BT    AX,10      ; Sets CF.
```

BTC Bit Test and Complement 80386

Opcode	Format	Clocks
0F BB [r]	BTC r/mw,rw	6/13
0F BB [r]	BTC r/md,rd	6/13
0F BA [7] ib	BTC r/mw,ib	6/8
0F BA [7] ib	BTC r/md,ib	6/8

Flags

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0									0		0		1	S

Pseudocode

Use the 1st operand to locate a register or a memory address.
Use the 2nd operand to select a single bit (bit 0 is the low order bit).
Set the Carry Flag to the selected bit.
Invert the selected bit {If it's 0 make it 1, if 1 make it 0}.

Operation

Bit Test and Complement has two functions. The first is to allow the programmer to select any bit in memory and invert (or reverse) it in memory, so the bit now has the opposite value it had before the command. The second is to save the old bit value in the Carry Flag.

BTC's first operand is a memory address or register. The second operand is a bit number. It can be any unsigned integer up to 8 bits wide, but only the low order 4 or 5 bits are used depending on the size of the first operand. The Carry Flag is then set to this bit. The bit in memory is inverted (set to 0 if it had been 1 and vice versa).

Exceptions	Modes	Reasons
#GP(0)	P	Result in nonwritable segment
#GP(0)	P	Illegal memory effective address in CS, DS, ES, FS, or GS segments
#SS(0)	P	Illegal address in SS segment
#PF(fc)	P V	Page fault
INT(13)	R V	Some part of operand is outside of address range 0 to 0FFFFH

User Notes

Many applications are made easier with the new bit manipulation instructions (for instance setting up allocation tables, wherein a bit's setting represents whether a piece of RAM or a sector on disk is in use). The BTC command allows any bit in the table to be quickly reversed. One dramatic example is the improvement in updating large video displays made possible by the large segment size and quick bit access of the 80386.

Example

```
MOV   AX,3CD0H   ; Loads value to scan into AX.
BTC   AX,2       ; Clears CF, sets AX to 3CD4 (hex).
```

BTR Bit Test and Reset 80386

Opcode	Format	Clocks
0F B3 [r]	BTR r/mw,rw	6/13
0F B3 [r]	BTR r/md,rd	6/13
0F BA [6] ib	BTR r/mw,ib	6/8
0F BA [6] ib	BTR r/md,ib	6/8

Flags

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0									0		0		1	S

Pseudocode

Use the 1st operand to locate a register or memory address.
Use the 2nd operand to select a single bit (bit 0 is the low order bit).
Set the Carry Flag to the selected bit.
Reset the selected bit (*Make it 0*).

Operation

Bit Test and Reset has two functions. The first is to allow the programmer to select any bit in memory and reset it (put in 0) in memory. The second is to save the old bit value in the Carry Flag.

BTR's first operand is a memory address or register. The second operand is a bit number. It can be any unsigned integer up to 8 bits wide, but only the low order 4 or 5 bits are used depending on the size of the first operand. The Carry Flag is then set to this bit. The bit in memory is reset (cleared, or made equal to 0).

Exceptions	Modes	Reasons
#GP(0)	P	Result in nonwritable segment
#GP(0)	P	Illegal memory effective address in CS, DS, ES, FS, or GS segments
#SS(0)	P	Illegal address in SS segment
#PF(fc)	P V	Page fault
INT(13)	R V	Some part of operand is outside of address range 0 to 0FFFFH

User Notes

Many applications are made easier with the new bit manipulation instructions (for instance setting up allocation tables, wherein a bit's setting represents whether a piece of RAM or a sector on disk is in use). The BTR command allows any bit in the table to be quickly reset (thus releasing a sector for instance). A more dramatic example to the average user is the improvement in updating large video displays made possible by the large segment size and quick bit access of the 80386.

Example

```
MOV   AX,3CD0H   ; Loads value to scan into AX.
BTR   AX,7       ; Sets CF, sets AX to 3C50 (hex).
```

BTS Bit Test and Set 80386

Opcode	Format	Clocks
0F AB [r]	BTS r/mw,rw	6/13
0F AB [r]	BTS r/md,rd	6/13
0F BA [5] ib	BTS r/mw,ib	6/8
0F BA [5] ib	BTS r/md,ib	6/8

Flags

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0									0		0		1	S

Pseudocode

Use the 1st operand to locate a register or a memory address.
Use the 2nd operand to select a single bit (bit 0 is the low order bit).
Set the Carry flag to the selected bit.
Set the selected bit (*Make it 1*).

Operation

Bit Test and Set has two functions. The first is to allow the programmer to select any bit in memory and set it (put in 1) in memory. The second is to save the old bit value in the Carry Flag.

BTS's first operand is a memory address or register. The second operand is a bit number. It can be any unsigned integer up to 8 bits wide, but only the low order 4 or 5 bits are used depending on the size of the first operand. The Carry Flag is then set to this bit. The bit in memory is set (made 1).

Exceptions	Modes	Reasons
#GP(0)	P	Result in nonwritable segment
#GP(0)	P	Illegal memory effective address in CS, DS, ES, FS, or GS segments
#SS(0)	P	Illegal address in SS segment
#PF(fc)	P V	Page fault
INT(13)	R V	Some part of operand is outside of address range 0 to 0FFFFH

User Notes

Many applications are made easier with the new bit manipulation instructions (for instance setting up allocation tables, wherein a bit's setting represents whether a piece of RAM or a sector on disk is in use). The BTS command allows any bit in the table to be quickly set (thus marking a sector as allocated, for instance). A more dramatic example to the average user is the improvement in updating large video displays made possible by the large segment size and quick bit access of the 80386.

Example

```
MOV   AX,3CD0H   ; Loads value to scan into AX.
BTS   AX,2       ; Clears CF, sets AX to 3CD4 (hex).
```

CALL Call Procedure 8086

Opcode	Format	Type	Clocks
E8 dw	CALL dw	Near, direct	7+m
E8 dd	CALL dd	Near, direct	7+m
FF [2]	CALL r/mw	Near, indirect	7+m/10+m
FF [2]	CALL r/md	Near, indirect	7+m/10+m
9A pd	CALL pd	Far, direct	17+m,*
9A pp	CALL pp	Far, direct	17+m,*
FF [3]	CALL mw:w	Far, indirect	22+m,*
FF [3]	CALL mw:d	Far, indirect	22+m,*

*These instructions have varying functions and timings in Protected Mode (see Chapter 5).

Flags

Normally CALL affects no flags. However, when a task switch is made in Protected Mode all flags are changed to the new task's saved flags.

Pseudocode

```
IF inter-segment CALL THEN
  PUSH CS (Code Segment) onto stack
  Set CS to segment selector of operand
END IF
PUSH IP (Instruction Pointer) onto stack
Set IP to offset part of operand
```

Operation

The CALL instruction is used to transfer control to another part of the program in such a way that control can be returned to the current point at a

later time. This facility is used to implement the procedures, functions, and subroutines of higher level languages. It is also extremely useful in assembly programs.

There are several types of CALL instructions, based on two different pairs of attributes. A call may be either near or far, and either direct or indirect. Therefore there are four different type of calls to consider.

The difference between a near and a far call is simply whether the called procedure is in the same segment as the caller (near) or in a different segment (far). For a far call the 80386 must place both the CS and IP registers on the stack before placing the new values into them. For a near call only IP is saved on the stack and changed.

The distinction between direct and indirect calls depends on how the programmer specifies the address of the called procedure. In a direct call the address is placed directly into the instruction. In an indirect call the programmer supplies a pointer to a register or memory location containing the address of the called procedure.

A near direct call contains a displacement to be added to IP, rather than an offset to be stored there as in the far direct call. A near indirect call may point to either a register or a memory location that contains the actual segment and offset for the called procedure. A far indirect call may not use a register, only memory (since full pointers can be 48 bits long).

The basic procedure is the same in Protected Mode, but inter-segment calls are much more complicated, since CALL may specify an operating system routine or even another task. In either case memory protection must be checked (see Chapter 5).

Exceptions	Modes	Reasons
#NP	P	Target code segment not present
#TS	P	Task switch required
#GP(0)	P	Illegal memory effective address in CS, DS, ES, FS, or GS segments
#SS(0)	P	Illegal address in SS segment
#PF(fc)	P V	Page fault
INT(13)	R V	Some part of operand is outside of address range 0 to 0FFFFH

User Notes

Most programmers use CALLS frequently to help get the most mileage out of a given piece of code. The interesting problem is how to pass parameters to and from the CALLed routine. On the later members of the iAPX 86 family, instructions like PUSHA help in preserving register values, while ENTER helps implement multiply nested routines. The subjects of when to use routines versus copies of pieces of code, and when to have the same routine copied into each segment versus using intersegment calls, are too complex to cover here, but are worth study as a way to optimize the efficiency of programs—especially the large programs supported easily by the 80386.

Example

```
CALL   SUBROUTINE
```

CBW Convert Byte to Word 8086
CWDE Convert Word to Dword 80386

Opcode	Format	Clocks
98	CBW	3
98	CWDE	3

Flags

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0									0		0		1	

Pseudocode

```
IF operand size = 16 bits THEN (* CBW *)
   Set each bit of AH to same value as highest order bit of AL.
ELSE (* CWDE *)
   Set high order 16 bits of EAX to same value as highest order bit of AX.
ENDIF
```

Operation

Converting a byte to a word is done by "sign-extending" the byte. This means that the byte's highest-order bit (its leftmost or most significant bit) is "propagated" or copied into every bit of the word not already occupied by the byte. For instance, sign-extending 1111 0000 into a word yields 1111 1111 1111 0000, while sign-extending 0101 0101 yields 0000 0000 0101 0101. Converting a word to a dword works in the same way.

Although it's not intuitively obvious why this works, it turns out that sign-extending a byte yields a word with the same value as the byte. Sign-extending just preserves the byte's value when it is placed into a word.

Exceptions

None

User Notes

CBW and CWDE are necessary instructions for the many times when a low-precision number needs to be converted for use in higher-precision math.

Example

```
MOV   AL,0FCH   ; Loads a −4 into AL.
CBW             ; Sets AX to FFFC (hex) or a word-length −4.
```

CLC Clear Carry Flag 8086

Opcode	Format	Clocks
F8	CLC	2

Flags

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0									0		0		1	0

Pseudocode

Clear the Carry Flag to 0.

Operation

CLC simply sets the Carry Flag to zero.

Exceptions

None

User Notes

Some other microprocessors have only an ADC instruction and no ADD instruction. On these machines using CLC or its equivalent is always necessary. Normally this isn't necessary on the iAPX 86s because there's an ADD instruction that ignores the carry when executing and conditions it when finished executing. However, there are two ways to write loops that add large numbers by repeatedly adding byte-sized pieces of them. One

way is to start with an ADD and then use a loop with ADC. The other way is to just use a loop with ADC; this results in a smaller program but requires that CLC be used before the start of the loop.

Also, unlike some other processors, the iAPX 86s require that C be *cleared* before a SBB (SuBtract with Borrow) that is not preceded by a flag-conditioning SUB. Other microprocessors may require that the carry type flag be *set* before an SBB-type operation.

Example

```
CLC   ; Clears CF.
```

CLD Clear Direction Flag 8086

Opcode	Format	Clocks
FC	CLD	2

Flags

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0				0					0		0		1	

Pseudocode

Clear the Direction Flag to 0.

Operation

CLD simply sets the Direction Flag to zero.

Exceptions

None

User Notes

The Direction Flag controls the direction of string operations. When D is cleared the index registers SI and/or DI are *incremented* after each repeat of a string operation. This is the "normal" direction, and is useful when each character in the string is stored in successively higher-numbered locations in memory and the string is being processed first

character first. It's also useful when the string is stored starting in high memory and heading toward low and is being processed last character first.

Example

```
CLD   ; Clears DF.
```

CLI Clear Interrupts Enabled Flag 8086

Opcode	Format	Clocks
FA	CLI	3

Flags

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0					0				0		0		1	

Pseudocode

Clear the Interrupt Flag to 0

Operation

CLI sets the Interrupt Flag to zero, preventing further interrupts until it is again set to one. However, there are some complications in 80286 and 80386 Protected Mode. CLI can fail if the current privilege level of the program executing the CLI is larger (less privileged) than the I/O privilege level bits in the Flags register.

Exceptions	Modes	Reasons
#GP(0)	P	Current privilege is greater than IOPL

User Notes

CLI is a vital instruction for time-sensitive pieces of code that can't be allowed the indeterminate length stops in processing that occur when

interrupts are enabled. CLI also disables the keyboard (assuming it's interrupt-based and not polled), which is useful for ensuring that users do not attempt to interrupt the program.

Although the privilege restrictions on CLI are burdensome, they're necessary. If an operating system is timesharing between several programs it must be able to interrupt each in turn to let the others execute. If your operating system allows access to Protected Mode, it's important to understand how it handles privilege levels before attempting to control interrupts from within your program.

Example

```
CLI   ; Clears IF.
```

CMC Complement Carry Flag 8086

Opcode	Format	Clocks
F5	CMC	2

Flags

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0									0		0		1	S

Pseudocode

```
IF (Carry flag = 0) THEN
  set the Carry flag
ELSE
  clear the Carry flag
END IF
```

Operation

CMC complements (negates, reverses, flips) the Carry Flag. After CMC the Carry Flag will contain the bit value opposite of the one it had before.

Exceptions

None

User Notes

This instruction is useful when using Carry as data storage or a parameter within a program. If you already know the state of the Carry Flag don't use CMC; use CLC or STC instead. This practice will make your programs more maintainable.

Example

```
CMC   ; Changes the sense of CF.
```

CMP Compare 8086

Opcode	Format	Clocks
3C ib	CMP AL,ib	2
3D iw	CMP AX,iw	2
3D iw	CMP EAX,id	2
80 [7] id	CMP r/mb,ib	2/5
81 [7] iw	CMP r/mw,iw	2/5
81 [7] id	CMP r/md,id	2/5
83 [7] ib	CMP r/mw,iw	2/5
83 [7] ib	CMP r/md,id	2/5
38 [r]	CMP r/mb,rb	2/5
39 [r]	CMP r/mw,rw	2/5
39 [r]	CMP r/md,rd	2/5
3A [r]	CMP rb,r/mb	2/6
3B [r]	CMP rw,r/mw	2/6
3B [r]	CMP rd,r/md	2/6

Flags

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0			S				S	S	0	S	0	S	1	S

Pseudocode

Subtract second operand from first, but don't store result in either
Condition flags based on result of subtraction

Operation

CMP subtracts its second operand from its first, but doesn't store the result in the destination operand or anywhere else. Instead, the only output

is that the applicable flags are set or reset ("conditioned" is often used to mean "set or reset") according to the result of the subtraction. SUB is explained below.

Exceptions	Modes	Reasons
#GP(0)	P	Illegal memory effective address in CS, DS, ES, FS, or GS segments
#SS(0)	P	Illegal address in SS segment
#PF(fc)	P V	Page fault
INT(13)	R V	Some part of operand is outside of address range 0 to 0FFFFH

User Notes

CMP is often used as a bit-test operator to force the Flags bits to tell you about what bit pattern is in the first operand. It's also used more generally to help decide whether one number is greater than another; in either case, a conditional jump can use the information gained to help the program decide what to do next.

Example

```
MOV   AX,956   ; Loads a 956 (3BC hex) into AX.
MOV   BX,373   ; Loads a 373 (175 hex) into BX.
CMP   AX,BX    ; Clears CF, AF, ZF, SF, and OF; sets PF.
```

CMPS Compare String 8086

Opcode	Format	Clocks Single	Clocks Repeated
A6	CMPSB	10	5+9*N
A7	CMPSW	10	5+9*N
A7	CMPSD	10	5+9*N

The "N" in the "Clocks Repeated" column stands for the number of repetitions actually executed.

Flags

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0			S				S	S	0	S	0	S	1	S

Pseudocode

```
Subtract ES:[(E)DI] from DS:[(E)SI], but don't store result
Condition flags based on result of subtraction
IF DF = 0 THEN
  Add size of operands (in bytes) to (E)SI and (E)DI
ELSE
  Subtract size of operands (in bytes) from (E)SI and (E)DI
END IF
```

Operation

CMPS subtracts its destination operand from its source operand, but doesn't store the result in the destination operand or anywhere else. Instead, the only output is that the applicable flags are set or reset ("conditioned") according to the result of the subtraction.

This differs from CMP in that CMPS's operands are not specified in the instruction. The first one is pointed to by the Source Index [(E)SI] (which normally points to the Data Segment unless a prefix override has been used) and the second is pointed to by the Destination Index [(E)DI] (which always must point to the Extra Segment for all string operations).

All of this could be done with the CMP, but CMPS also changes the values in SI and DI. Normally the Direction flag is 0 and SI and DI are incremented, but if an STD has been executed SI and DI are decremented. The increment and decrement are useful in automatically setting up the pointers for a repetition of the same instruction, and the REPE and REPNE prefixes are generally used with CMPS.

Exceptions	Modes	Reasons
#GP(0)	P	Illegal memory effective address in CS, DS, ES, FS, or GS segments
#SS(0)	P	Illegal address in SS segment
#PF(fc)	P V	Page fault
INT(13)	R V	Some part of operand is outside of address range 0 to 0FFFFH

User Notes

CMPS is generally used to compare two strings for equality. In this case the CX register is initialized to the length of the strings (since CX is checked and decremented by the REP-type prefixes) and SI and DI are initialized to the first character in each of the strings (which may or may not be a length byte). Then REPE CMPS does a continuing multiple operation (compare the two bytes, increment [or decrement if D = 1] the pointers, decrement CX) until the CMP shows two operands to be not equal (Z flag $\neq$ 0) or CX is decremented to 0. Commonly a JNZ instruction will then be used to jump away if the strings aren't equal; otherwise code for equal strings is executed.

Use REPNE CMPS for this operation, and the loop will fail when CX is zero or two *equal* characters are found. This tests for two strings that are different in every position.

Example

```
STR1  DD        1,2,3,4,5
STR2  DD        1,2,4,5,6
:               :
CLD                     ; Ensure direction is forward.
LDS       ESI,STR1      ; Set up source of compare, DS:[ESI].
LES       EDI,STR2      ; Set up destination of compare,
                        ; ES:[EDI].
MOV       ECX,5         ; Set up repeat count for compare.
REPE CMPSD              ; Executes 3 times, ends with ECX=2,
                        ; DS:[ESI] points at 4 in
                        ; STR1, and ES:[EDI] points at 5 in
                        ; STR2.
```

CWD Convert Word to Dword 8086
CDQ Convert Dword to Qword 80386

Opcode	Format	Clocks
99	CWD	2
99	CDQ	2

Flags

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0									0		0		1	

Pseudocode

```
IF operand size = 16 bits THEN (* CWD *)
   Set each bit of DX to same value as highest order bit of AX.
ELSE (* CDQ *)
   Set each bit of EDX to same value as highest order bit of EAX.
END IF
```

Operation

Converting a word to a doubleword is done by "sign-extending" the word. This means that the word's highest-order bit (its leftmost or most significant bit) is "propagated" or copied into every bit of a second word. For instance, sign extending 1111 1111 1111 0000 yields 1111 1111 1111 1111 1111 1111 1111 0000, while sign extending 0000 0000 0101 0101 yields . . . you get the idea. There is an explanation of sign extension under CBW in this

chapter; the important point is that CWD yields a doubleword with the same value as the original word. CDQ works the same way for longer conversions.

Exceptions

None

User Notes

CWD (or CDQ) is a necessary instruction for the many times when a word-length number is needed for higher-precision math. Since CWD has no effect on positive numbers, user's often fail to use it for routines that only handle positive input and then run into problems later when negative input is encountered. Since it only takes two clocks to execute, using this instruction in a conversion loop is efficient even when it's not initially expected that input will have negative values.

On the 80386 this instruction is not needed much, a dword fits in a single register of the 80386, while CWD spreads it over two registers. CWD is important when converting programs between the 80386 and earlier iAPX 86s.

Example

```
MOV  AX,0FFFCH  ; Loads a −4 into AX.
CWD             ; Sets DX to FFFF (hex) making DX:AX a
                ; dword-length −4.
```

DAA — Decimal Adjust AL After Addition — 8086

Opcode	Format	Clocks
27	DAA	4

Flags

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0			S				S	S	0	S	0	S	1	S

Pseudocode

```
IF (lower nibble of AL > 9) OR (Aux Carry = 1) THEN
  increment AL by 6
  set Aux Carry Flag
ELSE
  reset Aux Carry Flag
END IF
IF (AL > 9FH) OR (Carry = 1) THEN
  set AL to AL + 60H
  set Carry flag
ELSE
  clear Carry flag
END IF
```

Operation

DAA adjusts AL after a two packed BCD digit ADD. This instruction is needed because the smallest operands ADD uses are byte sized, while packed BCD stores two digits in a byte; so there's no easy way to add BCD

digits one at a time. DAA fixes each digit to the result you'd expect from adding two two-digit decimal numbers, including conditioning the Carry Flag appropriately for a decimal carry.

For instance, the number 25_{10} fits in a single packed BCD byte: 0010 0101 (25H). Adding this byte to itself yields this result: 0100 1010 (4AH). Without adjustment this result can't be interpreted as a packed-BCD byte. It does make sense, however, if you consider that ADD treats the operands as binary bytes and gives a binary result. DAA increments AL by 6, giving the result 0101 0000 (50H), which in packed-BCD is 50_{10}, the correct result.

Exceptions

None

User Notes

DAA is often used as part of a loop adding multi-precision BCD numbers: ADD a one-byte pair of digits, use DAA to adjust the result, ADC the next one-byte pair of digits, use DAA, and continue using ADC and DAA to the limits of the precision in use.

DAA itself only handles binary to BCD conversion.

Example

```
MOV   AX,18H   ; Loads a packed BCD 18 into AL and clears AH.
ADD   AL,6     ; Adds a packed BCD 6 to AL giving 1E (hex).
DAA            ; AX contains 24 (hex) or 24 packed BCD, CF is 0
               ; and AF is 1.
```

DAS Decimal Adjust AL After Subtraction 8086

Opcode	Format	Clocks
2F	DAS	4

Flags

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0			S				S	S	0	S	0	S	1	S

Pseudocode

```
IF (lower nibble of AL > 9) OR (Aux Carry = 1) THEN
  decrement AL by 6
  set Aux Carry Flag
ELSE
  reset Aux Carry Flag
END IF
IF (AL > 9FH) OR (Carry = 1) THEN
  set AL to AL − 60H
  set Carry flag
ELSE
  clear Carry flag
END IF
```

Operation

DAS adjusts AL after a two packed BCD digit SUB. This instruction is needed because the smallest operands SUB uses are byte sized, while packed BCD stores two digits in a byte, so there's no easy way to subtract

BCD digits one at a time. DAS fixes each digit to the result you'd expect from subtracting two two-digit decimal numbers, including conditioning the Carry Flag appropriately for a decimal carry.

For instance, the number 25_{10} fits in a single packed BCD byte: 0010 0101 (25H). Subtracting the BCD byte for 17_{10}, which is 0001 0111 (17H), yields this result: 0000 1110 (0EH), which can't be interpreted successfully as a packed BCD byte. It does make sense, however, if you consider that SUB treats the operands as binary bytes and gives a binary result. DAS decrements AL by 6, which leaves the result as 0000 1000 (08H), which is 08_{10}, the correct result.

Exceptions

None

User Notes

DAS is often used as part of a loop subtracting multi-precision BCD numbers: SUB a one-byte pair of digits, use DAS to adjust the result, SBB the next one-byte pair of digits, use DAS, and continue using SBB and DAS to the limits of the precision you're using.

DAS itself only handles binary to BCD conversion.

Example

```
MOV   AX,16H   ; Loads a packed BCD 16 into AL and clears AH.
SUB   AL,8     ; Subtracts a packed BCD 8 from AL giving 0D
               ; (hex).
DAS            ; AX contains 8 or 8 packed BCD, CF is 0 and AF
               ; is 1.
```

DEC Decrement 8086

Opcode	Format	Clocks
FE [1]	DEC r/mb	2/6
FF [1]	DEC r/mw	2/6
FF [1]	DEC r/md	2/6
48+rw	DEC rw	2
48+rd	DEC rd	2

Flags

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0			S				S	S	0	S	0	S	1	

Pseudocode

Set operand to operand − 1.

Operation

DEC subtracts 1 from its single operand and stores the result back in the operand.

Exceptions	Modes	Reasons
#GP(0)	P	Result in nonwritable segment
#GP(0)	P	Illegal memory effective address in CS, DS, ES, FS, or GS segments
#SS(0)	P	Illegal address in SS segment
#PF(fc)	P V	Page fault
INT(13)	R V	Some part of operand is outside of address range 0 to 0FFFFH

User Notes

Although DEC seems to be a much simpler command than SUB, it takes about the same amount of time to execute. Also, DEC doesn't set the Carry Flag, so in many cases it's preferable to use SUB with a second operand of 1. It's only worthwhile to use two DECs to replace a SUB with a second operand of 2 when memory is *very* tight.

Example

```
MOV   AX,956    ; Loads a 956 (3BC hex) into AX.
DEC   AX        ; AX now contains 955.
```

DIV Unsigned Divide 8086

Opcode	Format	Clocks
F6 [6]	DIV AL,r/mb	14/17
F7 [6]	DIV AX,r/mw	22/25
F7 [6]	DIV EAX,r/md	38/41

Flags

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0									0		0		1	

Pseudocode

```
IF opcode is F6 THEN (* dividend is a word *)
  divide AX by unsigned byte in operand
  store quotient in AL
  store remainder in AH
ELSE IF operand size is 16 THEN (* dividend is a dword *)
  divide DX:AX by unsigned word in operand
  store quotient in AX
  store remainder in DX
ELSE (* dividend is a qword *)
  divide EDX:EAX by unsigned dword in operand
  store quotient in EAX
  store remainder in EDX
END IF
```

Operation

Both the dividend and the divisor are treated as unsigned integers. The answer is given as a quotient part in one register and a remainder part in another, both of which are unsigned integers.

Exceptions	Modes	Reasons
#GP(0)	P	Result in nonwritable segment
#GP(0)	P	Illegal memory effective address in CS, DS, ES, FS, or GS segments
#SS(0)	P	Illegal address in SS segment
#PF(fc)	P V	Page fault
INT(0)	P R V	Result too large for destination or divisor is zero
INT(13)	R V	Some part of operand is outside of address range 0 to 0FFFFH

User Notes

DIV takes as long as several SUBS to execute; it's one of the few commands on the 80386 which is worth avoiding if possible when execution speed is critical. Interrupts result if the quotient is too big or the divisor is zero, so when code space is not at a premium it's worth checking for division by zero (go to an error-handling routine), division by one (jump around the division after adjusting the divisor as needed), and even sometimes division by two or four (use shifts to do the division) before actually executing the DIV. Too large a quotient will result in interrupt 0 and can lead to a call to an error-handling routine or more robust division routine.

DIV is used for unsigned division; this means that the dividend and divisor are always treated as positive numbers, ignoring the convention that a 1 in the high bit denotes a negative number. This allows positive numbers of greater magnitude to be handled directly by DIV. It's often worthwhile to write a division routine which uses repeated DIVs or IDIVs to handle division with greater precision than is possible with a single DIV or IDIV instruction.

Example

```
MOV   AX,956   ; Loads a 956 (3BC hex) into AX.
MOV   BX,300   ; Loads a 300 (12C hex) into BX.
DIV   AX,BX    ; AL now contains 3 and AH contains 56.
```

ENTER — Make Stack Frame for Procedure — 80186

Opcode	Format	Clocks
C8 iw 00	ENTER iw,0	10
C8 iw 01	ENTER iw,1	12
C8 iw ib	ENTER iw,ib	15+4*n

The "n" in the "Clocks" column stands for the level number (first operand) minus one.

Flags

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0									0		0		1	

Pseudocode

```
Push (E)BP
Save a copy of (E)SP
IF second operand is greater than 0 THEN
  FOR 1 TO second operand minus 1 DO
    Decrement (E)BP by 2 (or 4)
    Push the word pointed to by (E)BP
  ENDDO
  Push the word pointed to by the saved copy of (E)SP
END IF
Set (E)BP to the saved copy of (E)SP
Set (E)SP to (E)SP minus first operand
```

Operation

ENTER is used to implement procedure calls, and is a command expressly designed for implementing high-level languages. Basically, four

items are put onto the stack with each call: the procedure's arguments, the return address, a group of "frame pointers," and the local variables used by the procedure. All this information together is called a "stack frame." The "frame pointers" are pointers to the stack frames of the procedure calls that lead to the current procedure being called.

Of the four components of the stack frame, ENTER handles the last two very conveniently. The procedure parameters are pushed explicitly by the programmer before the CALL instruction (which pushes the return address). ENTER pushes the frame pointers (if any) and allocates room on the stack for the local variables.

There are two operands for ENTER: the first is the number of bytes of local variable storage to reserve, and the second is the level of the current procedure being entered. Level controls how many frame pointers are pushed on the stack (i.e., a level zero ENTER only allocates local variable storage). The LEAVE instruction restores the stack to its state before the ENTER was executed.

The steps executed are: Push the current BP onto the stack, replace the BP with the SP, push all the frame pointers (unless the level operand is zero), and decrement the SP by the number of bytes of storage you wish to save for local variables. These steps can be implemented by other assembly-language commands; the advantage of ENTER is that it executes very quickly considering everything it does, and it provides a standard way for compiler writers (who are less interested in maximum efficiency than in predictability) to implement procedure calls.

Most assembly-language programs will either use ENTER with level zero or not at all. The command probably does more than most assembly-language programs need. Levels greater than zero were created mostly for compiler writers.

Exceptions	Modes	Reasons
#SS(O)	P	(E)SP has exceeded the stack limit

User Notes

ENTER at levels above 0 becomes complicated, because pointers to each of the previous stack frames must be resaved at every new procedure call. This is done so that each new procedure can treat as global all the

variables in the procedure that called it, in the one that called the caller, etc. This kind of limited access to variables is basic to several of the newer programming languages.

ENTER is normally the first instruction in a procedure. The first parameter is the number of bytes of local variables to be saved; they are then accessed using BP as an index register. The old BP value is at displacement zero and the frame pointers (if any) follow. Next are the local variables. The second parameter is the current level. At the end of the procedure the LEAVE command will restore the stack pointer, wiping out access to the local variables.

Example

```
SUBROUTINE:
    ENTER   12,3    ; SUBROUTINE has 3 local variables (dwords)
                    ; and is at nesting level 3.
    :       :       :
    LEAVE           ; Removes current stack frame from stack.
    RET     8       ; SUBROUTINE has 2 parameters (dwords).
```

HLT Halt 8086

Opcode	Format	Clocks
F4	HLT	5

Flags

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0									0		0		1	

Pseudocode

```
WHILE no enabled external interrupt or RESET DO
END WHILE
```

Operation

HLT stops the 80386 from executing any further commands until an external interrupt or RESET is received. HLT is basically used for controlling programs that must interact directly with the outside world (other CPU's, which would reactivate the halted microprocessor with a RESET, or external devices, which would reactivate it with a hardware interrupt).

Exceptions	Modes	Reasons
#GP(0)	P	Current privilege level is not zero

User Notes

Single-CPU computers that aren't used for device control don't use HLT very often. Even when it is normally used it can be replaced with an infinite

loop that waits for an interrupt. This allows more flexibility than simply shutting down the processor, but can cost a little bit of time (while some or all of the wait loop executes) when the interrupt is received.

Example

```
HLT   ; Processor stops.
```

IDIV Signed Divide 8086

Opcode	Format	Clocks
F6 [7]	IDIV AL,r/mb	19/22
F7 [7]	IDIV AX,r/mw	27/30
F7 [7]	IDIV EAX,r/md	43/46

Flags

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0									0		0		1	

Pseudocode

```
IF opcode is F6 THEN (* dividend is a word *)
  divide AX by signed byte in operand
  store quotient in AL
  store remainder in AH
ELSE IF operand size is 16 THEN (* dividend is a dword *)
  divide DX:AX by signed word in operand
  store quotient in AX
  store remainder in DX
ELSE (* dividend is a qword *)
  divide EDX:EAX by signed dword in operand
  store quotient in EAX
  store remainder in EDX
END IF
```

Operation

Both the dividend and the divisor are treated as signed integers. The answer is given as a quotient part in one register and a remainder part in another, both of which are signed integers.

Exceptions	Modes	Reasons
#GP(0)	P	Result in nonwritable segment
#GP(0)	P	Illegal memory effective address in CS, DS, ES, FS, or GS segments
#SS(0)	P	Illegal address in SS segment
#PF(fc)	P V	Page fault
INT(0)	P R V	Result too large for destination or divisor is zero
INT(13)	R V	Some part of operand is outside of address range 0 to 0FFFFH

User Notes

IDIV takes a couple of clocks longer than DIV to execute. Like DIV, it's one of the few commands on the 80386 that is worth avoiding if possible when execution speed is critical. Also, interrupts result if the quotient is too big or the divisor is zero, so when code space is not at a premium it's worth checking for division by zero (go to an error-handling routine), division by one (jump around the division after adjusting the divisor as needed), and even sometimes division by two or four (use rotates to do the division) before actually executing the DIV. Too large a quotient will result in interrupt 0 and can lead to a call to an error-handling routine or more robust division routine.

IDIV is used for signed division; this means that the dividend and divisor are treated as either positive or negative numbers, depending on whether a number's high bit is 1 (making the number negative). IDIV handles both positive or negative numbers, but doesn't handle positive integers as large as those handled by DIV. It's often worthwhile to write a division routine that uses repeated DIVs or IDIVs to handle division with greater precision than is possible with a single DIV or IDIV instruction.

Example

```
MOV   AX,956     ; Loads a 956 (3BC hex) into AX.
MOV   BX,-300    ; Loads a -300 (FED4 hex) into BX.
IDIV  AX,BX      ; AL now contains -3 (FFFD hex) and AH
                 ; contains 56.
```

IMUL Signed Multiply 8086

Opcode	Format	Clocks
F6 [5]	IMUL r/mb	9-14/12-17
F7 [5]	IMUL r/mw	9-22/12-25
F7 [5]	IMUL r/md	9-38/12-41
0F AF [r]	IMUL rw,r/mw	9-22/12-25
0F AF [r]	IMUL rd,r/md	9-38/12-41
6B [r] ib	IMUL rw,r/mw,ib	9-14/12-17
6B [r] ib	IMUL rd,r/md,ib	9-14/12-17
6B [r] ib	IMUL rw,ib	9-14/12-17
6B [r] ib	IMUL rd,ib	9-14/12-17
69 [r] iw	IMUL rw,r/mw,iw	9-22/12-25
69 [r] id	IMUL rd,r/md,id	9-38/12-41
69 [r] iw	IMUL rw,iw	9-22/12-25
69 [r] id	IMUL rd,id	9-38/12-41

Flags

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0			S				U	U	0	U	0	U	1	S

Pseudocode

```
IF one operand form THEN
  IF operand size is byte THEN
    Set AX to the product of AL and the operand
  ELSE IF operand size is word THEN
    Set DX:AX to the product of AX and the operand
  ELSE (* operand size is dword *)
    Set EDX:EAX to the product of EAX and the operand
  END IF
ELSE IF two operand form THEN
  Set the first operand to the product of the first and second operands
ELSE (* three operand form *)
  Set the first operand to the product of the second and third operands
END IF
```

Operation

All operands are treated as signed numbers. All results can be treated as signed numbers. The maximum size of the result of a multiplication of two n-bit numbers is a 2-n bit number. Therefore the multiple operand forms of this instruction indicate the loss of the high-order bits of the result when overflow is set. The single operand forms do not have the same problem since the results produced are twice the size of the input operands.

Exceptions	Modes	Reasons
#GP(0)	P	Illegal memory effective address in CS, DS, ES, FS, or GS segments
#SS(0)	P	Illegal address in SS segment
#PF(fc)	P V	Page fault
INT(13)	R V	Some part of operand is outside of address range 0 to 0FFFFH

User Notes

The timing information for this instruction is given in ranges because the time required for a multiply depends on the size of the multiplier. The more significant bits, the longer the operation takes. The 80386 takes advantage of this fact with an early-out multiplication algorithm. The rightmost operand in all forms of the IMUL instruction is called the optimizing multiplier ("m" in the formula below). The actual number of clocks required for a multiply can be calculated with the following formula:

```
IF m = 0 THEN clocks = 9
ELSE clocks = max( log2(|m|), 3) + 6
```

Example

```
MOV   AL,40   ; Loads a 40 into AL.
IMUL  10      ; AX now contains 400, note that overflow gets set.
```

IN Input from Port 8086

Opcode	Format	Clocks
E4 ib	IN AL,ib	5
E5 ib	IN AX,ib	5
E5 ib	IN EAX,ib	5
EC	IN AL,DX	6
ED	IN AX,DX	6
ED	IN EAX,DX	6

Flags

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0									0		0		1	

Pseudocode

```
IF second operand is immediate THEN
  Zero extend second operand to 16 bits to form input port address
ELSE
  Input port address is contents of DX
END IF
IF first operand size is byte THEN
  Move the byte from the input port to AL
ELSE IF first operand size is word THEN
  Move the word from the input port to AX
ELSE (* the first operand size is dword *)
  Move the dword from the input port to EAX
END IF
```

Operation

The IN instruction is used to obtain a single byte, word, or dword from a peripheral device port. A port number may be any number from 0 to 65,535 ($2^{16}-1$). Normally a device has several ports assigned, some for commands, some for status, and some for data. Device control is accomplished by sending information to the command ports and getting information from the status ports. Input is done by getting data from a data port.

Port numbers 00F8H through 00FFH are reserved by Intel and shouldn't be used.

Exceptions	Modes	Reasons
#GP(0)	P	Current privilege is higher than IOPL
#GP(0)	V	Some of the corresponding permission bits in TSS equal 1

User Notes

Most programs get input through calls to an operating system, and thus don't use the IN instruction, but the IN instruction is indispensable for those writing device drivers or for anyone who must deal directly with a device.

When only a few bytes of input are needed the IN instruction should be used. If the input device can't provide data at a high rate of speed the IN instruction can be put in a loop, with NOPs or a counting loop included to slow down the transfer of data.

Example

```
MOV   DX,20     ; Sets up port number for IN instruction.
IN    EAX,DX    ; Input a dword to EAX.
```

INC Increment 8086

Opcode	Format	Clocks
FE [0]	INC r/mb	2/6
FF [0]	INC r/mw	2/6
FF [0]	INC r/md	2/6
40+rw	INC rw	2
40+rd	INC rd	2

Flags

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0			S				S	S	0	S	0	S	1	

Pseudocode

Set operand to operand + 1

Operation

INC adds 1 to its single operand and stores the result back in the operand.

Exceptions	Modes	Reasons
#GP(0)	P	Result in nonwritable segment
#GP(0)	P	Illegal memory effective address in CS, DS, ES, FS, or GS segments
#SS(0)	P	Illegal address in SS segment
#PF(fc)	P V	Page fault
INT(13)	R V	Some part of operand is outside of address range 0 to 0FFFFH

User Notes

Although the INC command seems to be a much simpler command than ADD, it takes about the same amount of time to execute. Also, INC doesn't set the Carry Flag, so in many cases it's preferable to use ADD with a second operand of 1. It's only worthwhile to use two INCs to replace an ADD with a second operand of 2 when memory is *very* tight.

Example

```
MOV   AX,956   ; Loads a 956 (3BC hex) into AX.
INC   AX       ; AX now contains 957.
```

INS Input String from Port 80186

Opcode	Format	Clocks Single	Clocks Repeated
6C	INSB	8	6+6*N
6D	INSW	8	6+6*N
6D	INSD	8	6+6*N

The "N" in the "Clocks Repeated" column stands for the number in the (E)CX register.

Flags

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0									0		0		1	

Pseudocode

```
IF operand size is byte THEN
  Move the byte from the input port named in DX to AL
ELSE IF operand size is word THEN
  Move the word from the input port named in DX to AX
ELSE (* operand size is dword *)
  Move the dword from the input port named in DX to EAX
END IF
IF DF = 0 THEN
  Add size of operand (in bytes) to (E)DI
ELSE
  Subtract size of operand (in bytes) from (E)DI
END IF
```

Operation

The INS instruction, like IN, is used to obtain a single byte, word, or dword from a peripheral device port. A port number may be any number from 0 to 65,535 ($2^{16}-1$). Normally a device has several ports assigned, some for commands, some for status, and some for data. Device control is accomplished by sending information to the command ports and getting information from the status ports. Input is done by getting data from a data port.

There are several distinctions between IN and INS. In INS the port number is always in DX. In INS the destination of the data is pointed to by ES:[(E)DI], and no segment override is permitted. Finally, the INS instruction is designed to be used with the REP prefix. That is, at the end of the instructions (E)DI is incremented or decremented (depending on the Direction Flag) by the operand size.

Port numbers 00F8H through 00FFH are reserved by Intel and shouldn't be used.

Exceptions	Modes	Reasons
#GP(0)	P	Current privilege is higher than IOPL
#GP(0)	P	Illegal memory effective address in CS, DS, ES, FS, or GS segments
#SS(0)	P	Illegal address in SS segment
#PF(fc)	P V	Page fault
INT(13)	R V	Some part of operand is outside of address range 0 to 0FFFFH
#GP(0)	V	Some of the corresponding permission bits in TSS equal 1

User Notes

Most programs obtain input through calls to an operating system, and thus don't use the INS instruction, but it is indispensable for those writing device drivers or for anyone who must deal directly with a device.

When only a few bytes of input are needed the IN instruction should be used. If the input device can't provide data at a high rate of speed, the INS

instruction can be put in a LOOP, with NOPs or a counting loop included to slow down the transfer of data.

Example

```
CLD                 ; Ensure direction is forward.
LES     EDI,INSTR   ; Set up destination of input, ES:[EDI].
MOV     ECX,5       ; Set up repeat count for INS.
MOV     DX,40       ; Set up input port number for INS.
REP INSB            ; Get 5 bytes of data from input port 40.
```

INT Call to Interrupt Procedure 8086

Opcode	Format	Clocks
CC	INT 3	33*
CD ib	INT ib	37*
CE	INTO	3,35*

*These instructions have varying functions and timings in Protected Mode (see Chapter 5). INTO takes 3 clocks when the interrupt is not taken and 35 clocks when the interrupt is taken.

Flags

Normally INT affects no flags. However, in Protected Mode all flags are changed to the new task's saved flags when a task switch is made.

Pseudocode

```
IF not INTO or OF flag set THEN
  Push (E)Flags register onto stack
  Push CS (Code Segment) register onto stack
  Push (E)IP (Instruction Pointer) onto stack
  Disable external interrupts (clear Interrupt Flag IF)
  Move segment and offset from interrupt vector table into CS:(E)IP
END IF
```

Operation

An interrupt is much like a call to a far procedure, with the flags automatically saved on the stack and interrupts automatically disabled. The location to jump to is not given by the INT command itself; instead the interrupt number is used to look up an entry in a table of interrupts, which gives the address of the interrupt routine.

When executed in Real mode INT first pushes the Flags register onto the stack, then it pushes the code segment register onto the stack. The same goes

for the Instruction Pointer. The Interrupt Flag IF is then cleared, disabling external (hardware) interrupts. Finally, the new code segment and offset are copied from the interrupt vector table into the CS register, and the Instruction Pointer is set to the start of the interrupt routine. When the interrupt routine is finished the registers will be restored and execution will proceed from the command after the INT.

The basic procedure is the same in Protected Mode, but several checks of privilege levels and sufficient stack sizes have to be passed or protection errors occur (see Chapter 5).

INTO is like INT except that the number of the interrupt routine to be used is assumed to be 4, and the INT is only executed if the Overflow Flag is set. INTO is generally used just after an arithmetic operation.

Exceptions	Modes	Reasons
#NP	P	Target code segment not present
#TS	P	Task switch required
#GP	P	Illegal CS, DS, ES, FS, or GS segment
#SS	P	Illegal SS segment
*****	R	80386 shut down due to insufficient stack space
#GP(0)	V	Emulates the interrupt operation if IOPL is less than 3

User Notes

Generally the interrupt vector table is set up by the operating system, and interrupts are used largely for operating system calls. Your operating system manual should give instructions for writing your own interrupts.

INTO replaces a series of instructions like: JNO OverOne, INT 4, OverOne; . . . A series of instructions like this will serve to construct an INTO "workalike" for other flags and interrupt numbers.

Example

```
MOV   AL,100   ; Loads a number into AL.
MUL   AL,10    ; This multiply will cause OF to be set.
INTO           ; Interrupt 4 will be taken.
```

IRET Interrupt Return 8086

Opcode	Format	Clocks
CF	IRET	22*

*This instruction has varying functions and timings in Protected Mode (see Chapter 5).

Flags

IRET restores all flags to the saved flags on the stack.

Pseudocode

Pop (E)IP (Instruction Pointer) from stack.
Pop CS (Code Segment) register from stack.
Pop (E)Flags register from stack.

Operation

An IRET is just like a RET from a far procedure except that the Flags register is restored from the stack. Since the flags are restored from their state before the (presumed) INT or INTO, the Interrupt Flag is also restored to its previous state so the disabling of interrupts that occurred when the INT was executed is now overridden by the previous state of the Interrupt Flag.

The basic procedure is the same in Protected Mode, but many checks of privilege levels and sufficient stack sizes have to be passed or protection errors occur (see Chapter 5).

Exceptions	Modes	Reasons
#NP	P	Target code segment not present
#TS	P	Task switch required

#GP	P	Illegal CS, DS, ES, FS, or GS segment
#SS	P	Illegal SS segment
INT(13)	R	Part of operand being popped lies beyond 0FFFFH
#GP(0)	V	Emulates the interrupt operation if IOPL is less than 3

User Notes

Generally the interrupt vector table is set up by the operating system, and interrupts are used largely for operating system calls. The IRET instructions for these system calls are at the end of the calls, and are not seen by the programmer. Your operating system manual should give instructions for writing your own interrupts.

Example

```
IRET    ; Return from interrupt.
```

Jcc Jump if Condition is Met 8086

Opcode	Format	Jump Condition	Clocks
77 db	JA db	Above (CF=0 and ZF=0)	7+m,3
0F 87 dw	JA dw	Above (CF=0 and ZF=0)	7+m,3
0F 87 dd	JA dd	Above (CF=0 and ZF=0)	7+m,3
73 db	JAE db	Above or equal (CF=0)	7+m,3
0F 83 dw	JAE dw	Above or equal (CF=0)	7+m,3
0F 83 dd	JAE dd	Above or equal (CF=0)	7+m,3
72 db	JB db	Below (CF=1)	7+m,3
0F 82 dw	JB dw	Below (CF=1)	7+m,3
0F 82 dd	JB dd	Below (CF=1)	7+m,3
76 db	JBE db	Below or equal (CF=1 or ZF=1)	7+m,3
0F 86 dw	JBE dw	Below or equal (CF=1 or ZF=1)	7+m,3
0F 86 dd	JBE dd	Below or equal (CF=1 or ZF=1)	7+m,3
72 db	JC db	Carry (CF=1)	7+m,3
0F 82 dw	JC dw	Carry (CF=1)	7+m,3
0F 82 dd	JC dd	Carry (CF=1)	7+m,3
E3 db	JCXZ db	CX register is zero	7+m,3
74 db	JE db	Equal (ZF=1)	7+m,3
0F 84 dw	JE dw	Equal (ZF=1)	7+m,3
0F 84 dd	JE dd	Equal (ZF=1)	7+m,3
E3 db	JECXZ db	ECX register is zero	7+m,3
7F db	JG db	Greater (ZF=0 and SF=OF)	7+m,3
0F 8F dw	JG dw	Greater (ZF=0 and SF=OF)	7+m,3
0F 8F dd	JG dd	Greater (ZF=0 and SF=OF)	7+m,3
7D db	JGE db	Greater or equal (SF=OF)	7+m,3
0F 8D dw	JGE dw	Greater or equal (SF=OF)	7+m,3
0F 8D dd	JGE dd	Greater or equal (SF=OF)	7+m,3
7C db	JL db	Less (SF<>OF)	7+m,3
0F 8C dw	JL dw	Less (SF<>OF)	7+m,3
0F 8C dd	JL dd	Less (SF<>OF)	7+m,3
7E db	JLE db	Less or equal (ZF=1 or SF<>OF)	7+m,3
0F 8E dw	JLE dw	Less or equal (ZF=1 or SF<>OF)	7+m,3
0F 8E dd	JLE dd	Less or equal (ZF=1 or SF<>OF)	7+m,3
76 db	JNA db	Not above (CF=1 or ZF=1)	7+m,3
0F 86 dw	JNA dw	Not above (CF=1 or ZF=1)	7+m,3

0F 86 dd	JNA dd	Not above (CF=1 or ZF=1)	7+m,3
72 db	JNAE db	Not above or equal (CF=1)	7+m,3
0F 82 dw	JNAE dw	Not above or equal (CF=1)	7+m,3
0F 82 dd	JNAE dd	Not above or equal (CF=1)	7+m,3
73 db	JNB db	Not below (CF=0)	7+m,3
0F 83 dw	JNB dw	Not below (CF=0)	7+m,3
0F 83 dd	JNB dd	Not below (CF=0)	7+m,3
77 db	JNBE db	Not below or equal (CF=0 and ZF=0)	7+m,3
0F 87 dw	JNBE dw	Not below or equal (CF=0 and ZF=0)	7+m,3
0F 87 dd	JNBE dd	Not below or equal (CF=0 and ZF=0)	7+m,3
73 db	JNC db	Not carry (CF=0)	7+m,3
0F 83 dw	JNC dw	Not carry (CF=0)	7+m,3
0F 83 dd	JNC dd	Not carry (CF=0)	7+m,3
75 db	JNE db	Not equal (ZF=0)	7+m,3
0F 85 dw	JNE dw	Not equal (ZF=0)	7+m,3
0F 85 dd	JNE dd	Not equal (ZF=0)	7+m,3
7E db	JNG db	Not greater (ZF=1 or SF<>OF)	7+m,3
0F 8E dw	JNG dw	Not greater (ZF=1 or SF<>OF)	7+m,3
0F 8E dd	JNG dd	Not greater (ZF=1 or SF<>OF)	7+m,3
7C db	JNGE db	Not greater or equal (SF<>OF)	7+m,3
0F 8C dw	JNGE dw	Not greater or equal (SF<>OF)	7+m,3
0F 8C dd	JNGE dd	Not greater or equal (SF<>OF)	7+m,3
7D db	JNL db	Not less (SF=OF)	7+m,3
0F 8D dw	JNL dw	Not less (SF=OF)	7+m,3
0F 8D dd	JNL dd	Not less (SF=OF)	7+m,3
7F db	JNLE db	Not less or equal (ZF=0 and SF=OF)	7+m,3
0F 8F dw	JNLE dw	Not less or equal (ZF=0 and SF=OF)	7+m,3
0F 8F dd	JNLE dd	Not less or equal (ZF=0 and SF=OF)	7+m,3
71 db	JNO db	Not overflow (OF=0)	7+m,3
0F 81 dw	JNO dw	Not overflow (OF=0)	7+m,3
0F 81 dd	JNO dd	Not overflow (OF=0)	7+m,3
7B db	JNP db	Not parity (PF=0)	7+m,3
0F 8B dw	JNP dw	Not parity (PF=0)	7+m,3
0F 8B dd	JNP dd	Not parity (PF=0)	7+m,3
79 db	JNS db	Not sign (SF=0)	7+m,3
0F 89 dw	JNS dw	Not sign (SF=0)	7+m,3
0F 89 dd	JNS dd	Not sign (SF=0)	7+m,3
75 db	JNZ db	Not zero (ZF=0)	7+m,3
0F 85 dw	JNZ dw	Not zero (ZF=0)	7+m,3
0F 85 dd	JNZ dd	Not zero (ZF=0)	7+m,3

70 db	JO db	Overflow (OF=1)	7+m,3
0F 80 dw	JO dw	Overflow (OF=1)	7+m,3
0F 80 dd	JO dd	Overflow (OF=1)	7+m,3
7A db	JP db	Parity (PF=1)	7+m,3
0F 8A dw	JP dw	Parity (PF=1)	7+m,3
0F 8A dd	JP dd	Parity (PF=1)	7+m,3
7A db	JPE db	Parity even (PF=1)	7+m,3
0F 8A dw	JPE dw	Parity even (PF=1)	7+m,3
0F 8A dd	JPE dd	Parity even (PF=1)	7+m,3
7B db	JPO db	Parity odd (PF=0)	7+m,3
0F 8B dw	JPO dw	Parity odd (PF=0)	7+m,3
0F 8B dd	JPO dd	Parity odd (PF=0)	7+m,3
78 db	JS db	Sign (SF=1)	7+m,3
0F 88 dw	JS dw	Sign (SF=1)	7+m,3
0F 88 dd	JS dd	Sign (SF=1)	7+m,3
74 db	JZ db	Zero (ZF=1)	7+m,3
0F 84 dw	JZ dw	Zero (ZF=1)	7+m,3
0F 84 dd	JZ dd	Zero (ZF=1)	7+m,3

Flags

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0									0		0		1	

Pseudocode

```
IF condition is met THEN
   Set Instruction Pointer (IP) to IP + sign-extended displacement
END IF
```

Operation

Jump instructions allow for non-sequential flow of program execution. The jumps covered here are conditional in that they either jump or don't jump depending on the state of the processor. Unconditional jumps, covered

below, always jump. Jumps may be further classified according to how the destination address is computed.

There are three types of jumps on the 80386—short, near, and far. Short jumps are relative jumps to an address close to the jump instruction (within the same code segment and within −128 to +127 bytes of the next instruction). Near jumps are also relative, but have a much greater range (anywhere in the same code segment). Far jumps, on the other hand, are absolute jumps to a particular address, which can be in any code segment. In the 80386 conditional jumps may be only short or near. In addition, the JCXZ and JECXZ instructions are only short jumps.

Most of the conditional jumps test the state of one or more of the flag bits. If the condition (listed in the above table) is met then the jump is taken; otherwise execution continues with the next instruction in sequence. The JCXZ and JECXZ instructions do not test any of the flags, however. Instead they test the value in the CX (or ECX) register.

In the timings above the first figure is for when the jump is taken and the second is for when the condition is not true and the jump is not taken. All jump instructions slow down the 80386 when the jump is taken, because the instruction queue is cleared and must be refilled. The value "m" in the clock count for the jumps is the number of "components" in the instruction at the target of the jump. Each prefix byte, opcode byte, mod r/m byte, or SIB byte is counted as a single component. Any displacement or immediate value also counts as one component each.

Exceptions	Modes	Reasons
#GP(0)	P	Jump target is beyond the limits of the code segment

User Notes

Many of the conditional jumps are meant for use with unsigned numbers, while others are meant for comparisons of signed numbers. The way to tell the difference is that the use of "above" and "below" indicates unsigned comparisons. The use of "greater" and "less" refers to the use of signed numbers.

A close examination of the instruction table for conditional jumps reveals that there are often several mnemonics for the same opcode (and

therefore conditional test). The reason for this redundancy is that the same state of the flags can mean different things based on the context of the jump. For example, a conditional jump often appears after a CMP or SUB has been executed. The conditional jump then compares the two operands of the previous instruction, and a JE (jump equal) might be appropriate. On the other hand, right after a DEC instruction the same jump with the JZ (jump zero) mnemonic could be used to terminate a loop after a count reached zero. We recommend (as a matter of good programming style) choosing jump mnemonics carefully so that they indicate the meaning of your comparisons.

The JCXZ and JECXZ jumps are "short" jumps; the location being jumped to must be within 128 bytes before or 127 bytes after the end of the jump instruction, and within the same segment. However, there are no direct opposites to these commands, so to reach a far location when CX becomes zero JCXZ's operand will need to be the label of a JMP command, probably outside the loop JCXZ is within.

Instructions like REPE and LOOPNE cause looping to occur until either the Zero Flag is set or cleared, or CX is zero. A JCXZ instruction just after the repeat or loop will cause a jump in those cases when the termination occurred due to CX reaching zero, rather than the Zero Flag reaching the needed setting. This allows different types of looping terminations to be handled differently.

Example

```
        MOV   ECX,5      ; Set up operand for compare.
        CMP   ECX,7      ; Compare the 5 with the 7.
        JLE   TARGET     ; The jump will be taken, 10 clocks
                         ; required.
        :     :          :
TARGET: AND   AL,7       ; 1 byte opcode, 1 mod r/m byte,
                         ; 1 immediate = 3 components.
```

JMP Unconditional Jump 8086

Opcode	Format	Type	Clocks
EB db	JMP db	Short, direct	7+m
E9 dw	JMP dw	Near, direct	7+m
E9 dd	JMP dd	Near, direct	7+m
FF [4]	JMP r/mw	Near, indirect	7+m/10+m
FF [4]	JMP r/md	Near, indirect	7+m/10+m
EA pd	JMP pd	Far, direct	17+m,*
EA pp	JMP pp	Far, direct	17+m,*
FF [5]	JMP mw:w	Far, indirect	22+m,*
FF [5]	JMP mw:d	Far, indirect	22+m,*

*These instructions have varying functions and timings in Protected Mode (see chapter 5).

Flags

Normally JMP affects no flags. However, in Protected Mode all flags are changed to the old task's saved flags when a task switch is made.

Pseudocode

```
IF intersegment JMP THEN
  Set CS to segment selector of operand
END IF
Set IP to offset part of operand
```

Operation

The function of JMP is to provide an unconditional transfer of control. It corresponds to the GOTO of higher-level languages, and differs from the conditional jump (Jcc) in that the transfer is always made and

the target of the jump may be in another segment. All jump instructions slow down the 80386 because the instruction queue is cleared and must be refilled after the jump is taken.

There are five distinct types of unconditional jumps, differing in how far away from the current instruction the jump target may be and in the technique used in specifying the target address.

The short jump (also a direct jump) specifies the target with an immediate displacement following the jump opcode. The displacement is simply added to IP to produce the new execution address. Since the displacement is a byte, the target must lie within the same code segment and be within −128 and +127 bytes from the instruction following the jump.

A near direct jump is similar to the short jump in that the target address is specified by an immediate displacement following the jump opcode. This displacement is also added directly to IP. Note that at the time of the add IP points at the instruction following the jump. The displacement is either a word or dword, depending on the size of the current code segment. The jump can therefore reach anywhere in the current code segment.

With a near indirect jump the target address is specified indirectly. It is still restricted to the same code segment as the jump instruction, but specifies either a register or memory location containing the offset within the current code segment of the target instruction. The offset is either a word or dword, depending on the size of the current code segment.

A far direct jump permits the target of the jump to be in a different code segment. The jump instruction contains an immediate operand that is a pointer to the target instruction. The pointer contains a word segment selector (which is loaded into CS) and either a word or dword offset (which is loaded into (E)IP). The size of the offset depends on the size of the target code segment.

A far indirect jump is similar to a near indirect jump in that the instruction contains a pointer to the actual target address. However, the target address is a full pointer rather than a simple offset, and therefore may not be in a register. The target pointer contains a segment selector (which is loaded into CS) and either a word or dword offset (which is loaded into (E)IP). The size of the offset depends on the size of the target code segment.

The basic procedure is the same in Protected Mode, but intersegment jumps are much more complicated since the jump may specify an operating system routine or even another task. In either case memory protection must be checked (see Chapter 5).

Exceptions	Modes	Reasons
#NP	P	Target code segment not present
#TS	P	Task switch required
#GP(0)	P	Illegal memory effective address in CS, DS, ES, FS, or GS segments
#SS(0)	P	Illegal address in SS segment
#PF(fc)	P V	Page fault
INT(13)	R V	Some part of operand is outside of address range 0 to 0FFFFH

User Notes

On the 80386 a short jump (distance given by a byte) and a near jump (distance given by a word) in Real Mode both take only 7 clocks to execute. The only advantage of a short jump is that its instruction is a total of 1 byte shorter. Thus, no execution speed is saved by rearranging your code to make sure most jumps are short, as long as they're in the same segment.

Example

```
        JMP     TARGET  ; The jump will take 10 clocks.
        :       :       :
TARGET: AND     AX,7    ; 1 byte opcode, 1 mod r/m byte,
                        ; 1 immediate = 3 components.
```

LAHF Load Flags Into AH 8086

Opcode	Format	Clocks
9F	LAHF	2

Flags

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0									0		0		1	

Pseudocode

Set AH to low byte of Flags register.

Operation

LAHF is a quick way to load all the flags stored in the low byte of the Flags register into AH, where they can be changed, examined, or stored as needed. This instruction has been provided for compatibility with the 8080/85, and is rarely used in iAPX 86 family programs (which would use POPF instead).

The bits in AH after the transfer are: SF ZF x AF x PF x CF, with "x" meaning "undefined."

Exceptions

None

User Notes

Except for programs converted from (or due to be converted to) the 8080/85, this instruction is normally not used. It could be used to check the status of all the low-byte flags at once (by comparing AH to a bit pattern, for instance), or to implement obscure conditional jumps that depend on flag combinations not covered by JE, JNA, etc. However, this is made more difficult by the fact that several of the bits are undefined.

To get around this, LAHF could be implemented; then AND AH,D5H would ensure that the undefined bits were cleared. Finally, one or more AND's followed by JE's or JNE's would compare the bit pattern in AH to various predefined patterns, and jump based on the result.

Example

```
LAHF                    ; Loads the flags into AH.
AND     AH,11H          ; Mask out the AF and CF bits.
JNZ     SOMEWHERE       ; Simulates a "jump on AF or CF set"
                        ; instruction.
```

LEA Load Effective Address Offset 8086

Opcode	Format	Clocks
8D [r]	LEA r16,m16	2
8D [r]	LEA r32,m16	2
8D [r]	LEA r16,m32	2
8D [r]	LEA r32,m32	2

Flags

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0									0		0		1	

Pseudocode

```
IF register size is 16 THEN
  IF memory size is 16 THEN
    Set register to offset part of effective address
  ELSE (* memory size is 32 *)
    Set register to low order 16 bits of offset part of effective address
  END IF
ELSE (* register size is 32 *)
  IF memory size is 16 THEN
    Set register to offset part of effective address, zero extended to 32 bits
  ELSE (* memory size is 32 *)
    Set register to offset part of effective address
  END IF
END IF
```

Operation

LEA is like a MOV from memory to a register, but the offset part of the address is MOVed, not the contents of RAM at the address. The memory size referred to in the pseudocode is determined by the USE attribute of the segment containing the memory address.

Exceptions	Modes	Reasons
#UD	P	Second operand is a register
INT(6)	R V	Second operand is a register

User Notes

Sometimes LEA and MOV can be used interchangeably. For instance, LEA AX, STRUCTURE and MOV AX, OFFSET STRUCTURE have the same effect. However, LEA allows the use of any addressing mode for the second operand, as long as it results in a memory reference. Thus LEA AX,STRUCTURE [BX] [D1] allows the address of a doubly-indexed pointer to be moved directly into AX; this couldn't be done directly with a MOV.

Another interesting aspect of the LEA instruction is its capability to provide a very fast but somewhat limited integer multiply instruction. By using scaled index addressing mode, multiplications by 2, 4, and 8 can be accomplished. With based scaled index addressing mode, LEA will perform multiplications by 3, 5, or 9. Note that these all require only 2 clocks, far faster than either multiply or shift instructions.

Example

```
MOV   BX,11           ; Get an 11 into BX to be multiplied.
LEA   AX,[BX][BX*4]   ; Loads AX with BX*5, in this case 55.
```

LEAVE — Remove Procedure Stack Frame — 80186

Opcode	Format	Clocks
C9	LEAVE	4

Flags

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0									0		0		1	

Pseudocode

Set (E)SP to (E)BP.
POP old frame pointer into (E)BP.

Operation

LEAVE is used to implement procedure calls, and is a command expressly designed for implementing high-level languages. It resets the stack pointer to exclude the procedure's local variables and pops from the stack a "frame pointer." These actions prepare the stack for the RET instruction that should immediately follow.

Exceptions	Modes	Reasons
#SS(0)	P	BP points to a location outside the current stack segment
INT(13)	R V	Some part of the operand lies outside address space 0 to 0FFFFH

User Notes

LEAVE is much simpler than ENTER because ENTER does most of the work. It sets BP to point at the correct place so that all LEAVE needs to do is put BP into the stack pointer and then pop the old BP value.

LEAVE should be the last instruction before RET in a procedure in which ENTER is the first instruction. The LEAVE instruction changes the stack pointer so the local variables that had been on the stack are removed and the frame pointer (BP) is set ready for the procedure that called this one.

Example

```
SUBROUTINE:
            ENTER12,3   ; SUBROUTINE has 3 local variables
                        ; (dwords) and is at nesting level 3.
            :     :     :
            LEAVE       ; Removes current stack frame from
                        ; stack.
            RET   8     ; SUBROUTINE has 2 parameters
                        ; (dwords).
```

LOCK Assert BUS LOCK Signal Prefix 8086

Opcode	Format	Clocks
F0	LOCK	0

Flags

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0									0		0		1	

Pseudocode

Set BUS LOCK signal for duration of the following instruction.

Operation

The LOCK prefix is used in multiple-processor systems to prevent contention for shared memory at certain critical times. It asserts a special bus signal, called LOCK, which prevents any other bus device from accessing the bus during the time the signal is asserted. The signal is asserted during the entire execution of the instruction that follows the LOCK prefix.

On the 80386 only certain instructions may be used with LOCK. The use of any other instruction will cause an undefined opcode trap to occur. The valid instruction/operand combinations are:

BT, BTS, BTR, BTC	mem, reg/imm
ADD, OR, ADC, SBB, AND, SUB, XOR	mem, reg/imm
XCHG	reg, mem
XCHG	mem, reg
NOT, NEG, INC, DEC	mem

Note that all these instructions require a value to be read from memory, modified in some way, and stored back in the same memory location. The XCHG instruction is always locked, even if the prefix is not present.

Exceptions	Modes	Reasons
#GP(0)	P	Current privilege level is higher than the I/O privilege level
#UD	P V	Instruction following LOCK is not listed in the table above
INT(6)	R	Instruction following LOCK is not listed in the table above

User Notes

The LOCKed instruction can generate any additional exception, just as if it had been executed "normally."

For an example of why this prefix might be needed, consider the following situation. A two-processor system contains a shared memory location used as an "event" counter. When either processor detects an "event" it must increment this shared memory location. The program in each processor will use an INC instruction without the LOCK prefix. Suppose that four events have already been detected and both processors detect an event at about the same time. If both processors attempt to execute their respective INC instructions too close together, the following disaster happens.

The first processor reads the value (4) from shared memory and begins the increment process internally. While the internal increment is happening in processor one, processor two gets the bus and uses it to read the same value (4) from the same memory location. While processor two is internally incrementing this value the first processor stores the result of its increment (5) back into memory. Finally, the second processor stores the result of its increment (also 5) back into the shared memory location. What has happened is that the counter has missed an "event." The LOCK prefix would have prevented this bug.

You may say to yourself that the above scenario is quite unlikely and that is just the reason that this kind of bug is so difficult to find. A system with this bug in it could run correctly for weeks or months and then suddenly start

"crashing" mysteriously because a situation that provoked the bug became common.

Locked access may not be guaranteed if the other processor executes an instruction with any one or more of the following conditions:

- The LOCK prefix is not used.
- The instruction is not in the list above.
- A memory operand is specified that does not exactly overlap.

Previous Intel processors allowed a more liberal use of the LOCK prefix. Therefore, take care when converting any programs from these earlier systems that contain a LOCK.

Example

```
LOCK BTR    FLAGWORD,AVAILBIT   ; Clear avail bit to indicate
                                ; we have the shared resource.
JNC         NODICE              ; Jump if someone else
                                ; already had it allocated.
```

LODS Load String 8086

Opcode	Format	Clocks Single	Clocks Repeated
AC	LODSB	5	*
AD	LODSW	5	*
AD	LODSD	5	*

*LODS is the one string instruction which cannot use any of the repeat prefixes.

Flags

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0									0		0		1	

Pseudocode

```
Determine size of operand
IF operand size is 8 THEN
  Move byte from address DS:[(E)SI] to AL
ELSE IF operand size is 16 THEN
  Move word from address DS:[(E)SI] to AX
ELSE (* operand size is 32 *)
  Move dword from address DS:[(E)SI] to EAX
IF DF = 0 THEN
ENDIF
  Add size of operand (in bytes) to (E)SI
ELSE
  Subtract size of operand (in bytes) from (E)SI
END IF
```

Operation

LODS is just like a MOV from memory to AL or AX, but it also automatically adjusts SI after the move. If the Direction Flag is 0 SI is incremented;

otherwise it's decremented. If a byte was moved the index is adjusted by 1; if a word, by 2; and if a dword, by 4.

Exceptions	Modes	Reasons
#GP(0)	P	Illegal memory effective address in CS, DS, ES, FS, or GS segments
#SS(0)	P	Illegal address in SS segment
#PF(fc)	P V	Page fault
INT(13)	R V	Some part of operand is outside of address range 0 to 0FFFFH

User Notes

LODS can't be used with any of the REP prefixes; REP would only have the effect of repeatedly overwriting the AL or AX register with each of the bytes or words pointed to by SI. However, LODS is often used in a repeat loop looking for a given character in a string: LODS loads AL, which is then compared to the needed character, and this is repeated until the character is found.

LODS and STOS can be used together to transfer a string from DS to ES, with any needed conditional tests or changes put in between LODS and STOS. If a string of known length is to be moved unchanged, then MOVS will do the same thing faster.

Example

```
;       The following copies a string from one location in memory to
;       another. It copies until either a maximum number of characters
;       have been moved or a zero character has been moved.
;       These zero-terminated strings are just like "C" language strings.
;
        CLD                     ; Ensure direction is forward.
        LDS     ESI,SSTR        ; Get pointer to source string,
                                ; DS:[ESI].
        LES     EDI,DSTR        ; Get pointer to destination
                                ; string, ES:[EDI].
        MOV     ECX,MAXSTR      ; Set up repeat count for
                                ; LOOP.
```

```
COPYL:
        LODSB                   ; Get next character from
                                ; source string.
        STOSB                   ; Store into destination string.
        TEST    AL,AL           ; Will set Zero flag only on
                                ; zero character.
        LOOPNZ  COPYL           ; Loop until MAXSTR bytes or
                                ; a zero byte have been moved.
```

LOOPcc — Loop Control with CX Counter — 8086

Opcode	Format	Jump Condition	Clocks
E2 db	LOOP db	(E)CX <> 0	11+m
E1 db	LOOPE db	(E)CX <> 0 and ZF = 1	11+m
E1 db	LOOPZ db	(E)CX <> 0 and ZF = 1	11+m
E0 db	LOOPNE db	(E)CX <> 0 and ZF = 0	11+m
E0 db	LOOPNZ db	(E)CX <> 0 and ZF = 0	11+m

Flags

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0									0		0		1	

Pseudocode

```
Decrement (E)CX (* No flags are changed *)
Determine size of operand
IF condition is met THEN
   Set Instruction Pointer (IP) to IP + sign-extended displacement
END IF
```

Operation

LOOP decrements CX (or ECX), then checks to see that it's not zero. The Zero Flag may also be checked. If the register is not zero and any optional ZF condition is met, a short jump is made to the label given as an operand after the LOOPcc. The assembler translates the label into a byte offset, which can range from 128 bytes before the instruction to 127 bytes after. This offset is added to the current address to determine where execution will proceed next.

Exceptions	Modes	Reasons
#GP(0)	P	Jump target is beyond the limits of the code segment

User Notes

LOOP allows a FOR-type loop to be implemented directly in assembly language. As with a FOR loop, the value in the CX register shouldn't be altered by any of the commands within the loop or the loop could easily go on forever. The loop count is treated as an unsigned integer.

A common source of confusion arises from the name LOOPZ. Just remember that the instruction loops while the last result to set the flags is zero and (E)CX zero.

Example

See LODS (above) for a good example of LOOPcc.

Lxx Load Full Pointer 8086

Opcode	Format	Clocks
C5 [r]	LDS rw,mw:w	7,22*
C5 [r]	LDS rd,mw:d	7,22*
C4 [r]	LES rw,mw:w	7,22*
C4 [r]	LES rd,mw:d	7,22*
0F B2 [r]	LSS rw,mw:w	7,22*
0F B2 [r]	LSS rd,mw:d	7,22*
0F B4 [r]	LFS rw,mw:w	7,22*
0F B4 [r]	LFS rd,mw:d	7,22*
0F B5 [r]	LGS rw,mw:w	7,22*
0F B5 [r]	LGS rd,mw:d	7,22*

*These instructions require extra clocks in Protected Mode due to extra processing needs (see Chapter 5).

Flags

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0									0		0		1	

Pseudocode

Set the segment register to the segment part of the second operand.
Set the general register to the offset part of the second operand.

Operation

These instructions are used to set up two registers at one time. Both a segment register and one of the general registers are loaded. The segment register is loaded with the 16-bit segment selector of the operand. The

general register is loaded with either a word or dword offset of the operand within its segment. The size loaded depends on the size attribute of the specified segment.

This basically simple operation is somewhat more complicated in Protected Mode. The main problem here is that any change in one of these registers means that the processor will now be trying to reference a whole different piece of memory, which probably has different protection levels than the current one. If virtual memory techniques are in use the segment may even be out on disk.

Exceptions	Modes	Reasons
#UD	P	Source operand is a register
#GP(0)	P	Illegal memory effective address in CS, DS, ES, FS, or GS segments
#GP(0)	P	Null selector loaded to SS
#SS(0)	P	Illegal address in SS segment
#PF(fc)	P V	Page fault
INT(13)	R V	Some part of operand is outside of address range 0 to 0FFFFH
INT(6)	R V	Source operand is a register

User Notes

The most common use of these instructions is to set up one of the data segment registers and an index register in preparation for accessing a new block of memory.

Example

```
LES   DI, STRUCTURE   ; Loads segment of STRUCTURE into ES and
                      ; offset of STRUCTURE into DI.
```

MOV Move Data 8086

Opcode	Format	Clocks
B0+r ib	MOV rb,ib	2
B8+r iw	MOV rw,iw	2
B8+r id	MOV rd,id	2
C6 [0] ib	MOV r/mb,ib	2
C7 [0] iw	MOV r/mw,iw	2
C7 [0] id	MOV r/md,id	2
A0 d	MOV AL,db	4
A1 d	MOV AX,dw	4
A1 d	MOV EAX,dd	4
A2 d	MOV db,AL	2
A3 d	MOV dw,AX	2
A3 d	MOV dd,EAX	2
88 [r]	MOV r/mb,rb	2
89 [r]	MOV r/mw,rw	2
89 [r]	MOV r/md,rd	2
8A [r]	MOV rb,r/mb	2/4
8B [r]	MOV rw,r/mw	2/4
8B [r]	MOV rd,r/md	2/4
8E [0]	MOV ES,r/mw	2/5*
8E [1]	MOV CS,r/mw	2/5*
8E [2]	MOV SS,r/mw	2/5*
8E [3]	MOV DS,r/mw	2/5*
8C [0]	MOV r/mw,ES	2
8C [1]	MOV r/mw,CS	2
8C [2]	MOV r/mw,SS	2
8C [3]	MOV r/mw,DS	2

*These instructions require 18/19 clocks in Protected Mode due to extra processing needs (see Chapter 5).

Flags

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0									0		0		1	

Pseudocode

Set first operand to second operand.

Operation

MOV is the same as the assignment operator in a high-level language; it sets the first operand equal to the second. Unlike other assembly languages, the destination operand is given first, followed by the source operand.

Although this seems very simple, there are two additional complications encountered in using the MOV instruction. The first is when a MOV SS is executed. This automatically inhibits all interrupts until after the next instruction is executed. The implied expectation is that the next instruction will be MOV SP or some other instruction that results in the SS:SP combination being restored to a meaningful value.

The second complication is encountered when moving into any segment register in Protected Mode. The main problem here is that any change in one of these registers means that the processor will now be trying to reference a whole different piece of memory, which probably has different protection levels than the current one.

Exceptions	Modes	Reasons
#GP, #SS, #NP	P	A segment register is being loaded
#GP(0)	P	Result in nonwritable segment
#GP(0)	P	Illegal memory effective address in CS, DS, ES, FS, or GS segments
#SS(0)	P	Illegal address in SS segment
#PF(fc)	P V	Page fault
INT(13)	R V	Some part of operand is outside of address range 0 to 0FFFFH

User Notes

The most important thing to know about using MOV is knowing when to pull some less-used but better-suited instruction from your bag of tricks. For instance, MOVS is often better for moving strings, while IN and OUT are needed if your source or destination is a port.

Example

```
MOV   AX,ES   ; Copies the contents of ES into AX.
```

MOVxX — Move with Sign/ Zero Extension — 8086

Opcode	Format	Clocks
0F BE [r]	MOVSX rw,r/mb	3/6
0F BE [r]	MOVSX rd,r/mb	3/6
0F BF [r]	MOVSX rd,r/mw	3/6
0F B6 [r]	MOVZX rw,r/mb	3/6
0F B6 [r]	MOVZX rd,r/mb	3/6
0F B7 [r]	MOVZX rd,r/mw	3/6

Flags

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0									0		0		1	

Pseudocode

Extend the source operand (by either zeros or the sign bit) to the length of the destination.
Store the result into the destination operand.

Operation

MOVSX and MOVZX handle the problem of a destination operand that has more bits in it than the source (i.e., the target is a word, the source a byte). SX stands for "sign extension" and means that the high bit of the source is copied into the additional bits available in the target. ZX stands for "zero extension," and the target's available bits are filled with zeros.

Exceptions	Modes	Reasons
#GP(0)	P	Illegal memory effective address in CS, DS, ES, FS, or GS segments
#SS(0)	P	Illegal address in SS segment
#PF(fc)	P V	Page fault
INT(13)	R V	Some part of operand is outside of address range 0 to 0FFFFH

User Notes

Because these instructions only allow one of the general registers as a destination, they are most useful in setting up registers for further computations.

Example

```
MOVSX   AX,92H   ; Sets AX to 0FF92 (hex).
```

MOVS Move String 8086

Opcode	Format	Clocks Single	Clocks Repeated
A4	MOVSB	7	5+4*N
~~A7~~ A5	MOVSW	7	5+4*N
A7	MOVSD	7	5+4*N

*The "N" in the "Clocks Repeated" column stands for the number of repetitions (from (E)CX).

Flags

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0									0		0		1	

Pseudocode

```
Determine size of operand
IF operand size is 8 THEN
  Move byte from address DS:[(E)SI] to ES:[(E)DI]
ELSE IF operand size is 16 THEN
  Move word from address DS:[(E)SI] to ES:[(E)DI]
ELSE (* operand size is 32 *)
  Move dword from address DS:[(E)SI] to ES:[(E)DI]
END IF
IF DF = 0 THEN
  Add size of operand (in bytes) to (E)SI and to (E)DI
ELSE
  Subtract size of operand (in bytes) from (E)SI and to (E)DI
END IF
```

Operation

MOVS is just like a regular MOV from the place in memory pointed to by the source index to the place in the extra segment pointed to by the destination index. However, it also automatically adjusts both indices after the move. If the Direction Flag is 0 SI and DI are incremented; otherwise they are decremented. If a byte was moved the indices are adjusted by 1; if a word, by 2; and if a dword, by 4.

Exceptions	Modes	Reasons
#GP(0)	P	Result in nonwritable segment
#GP(0)	P	Illegal memory effective address in CS, DS, ES, FS, or GS segments
#SS(0)	P	Illegal address in SS segment
#PF(fc)	P V	Page fault
INT(13)	R V	Some part of operand is outside of address range 0 to 0FFFFH

User Notes

MOVS can't be used with REPE or REPZ since it doesn't condition any flags. However, the REP prefix works as expected, moving CX bytes or words from the source to the destination. This is the usual way of using MOVS.

REP works well when you can load CX with the length of the string you're moving, but if the transfer has to stop when a given byte is found some other method has to be used. One method is to use LODS, do the test, use STOS, and then repeat with a LOOP instruction.

Example

```
CLD                     ; Ensure direction is forward.
LDS       ESI,STR1      ; Set up source pointer, DS:[ESI].
LES       EDI,STR2      ; Set up destination pointer, ES:[EDI].
MOV       ECX,5         ; Set up repeat count for move.
REPE MOVSD              ; Executes 5 times, copies from STR1 to
                        ; STR2.
```

MUL Unsigned Multiply 8086

Opcode	Format	Clocks
F6 [4]	MUL r/mb	9-14/12-17
F7 [4]	MUL r/mw	9-22/12-25
F7 [4]	MUL r/md	9-38/12-41

Flags

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0			S				U	U	0	U	0	U	1	S

Pseudocode

```
IF operand size is byte THEN
   Set AX to the product of AL and the operand
ELSE IF operand size is word THEN
   Set DX:AX to the product of AX and the operand
ELSE (* operand size is dword *)
   Set EDX:EAX to the product of EAX and the operand
END IF
```

Operation

All operands are treated as unsigned numbers. All results can be treated as unsigned numbers. The maximum size of the result of a multiplication of two n-bit numbers is a 2n-bit number. Therefore the results produced are twice the size of the input operands.

Exceptions	Modes	Reasons
#GP(0)	P	Illegal memory effective address in CS, DS, ES, FS, or GS segments
#SS(0)	P	Illegal address in SS segment
#PF(fc)	P V	Page fault
INT(13)	R V	Some part of operand is outside of address range 0 to 0FFFFH

User Notes

The timing information for this instruction is given in ranges because the time required for a multiply depends on the size of the multiplier. The more significant bits, the longer the operation takes. The 80386 takes advantage of this fact with an early-out multiplication algorithm. The specified operand in the MUL instruction is called the optimizing multiplier ("m" in the formula below). The actual number of clocks required for a multiply can be calculated with the following formula:

IF m = 0 THEN clocks = 9
ELSE clocks = max(log2(|m|), 3) + 6

Example

```
MOV   AL,128   ; Loads a 128 (80 hex) into AL.
MOV   BL,10    ; Loads a 10 (0A hex) into AL.
IMUL  BL       ; AX now contains 1280 (500 hex), note that over-
               ; flow gets set.
```

NEG Two's Complement Negation 8086

Opcode	Format	Clocks
F6 [3]	NEG r/mb	2/6
F7 [3]	NEG r/mw	2/6
F7 [3]	NEG r/md	2/6

Flags

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0			S				S	S	0		0	S	1	S

Pseudocode

```
Subtract operand from 0
Place result in operand
IF operand is zero THEN
  clear carry flag to 0
ELSE
  set carry flag to 1
END IF
```

Operation

NEG does a two's complement negation of its single operand. If the operand is 65 before the negation, it will be −65 after, and vice versa. This is done by taking the two's complement of the operand. Every bit in the operand is reversed; 1's are changed to 0's and 0's to 1's, and the result is then incremented by 1. This gives the negation of the original operand.

Another way of looking at NEG's operation is to say that the operand is subtracted from 0 and the result is placed in the operand. This operation is easier to understand, but doesn't make clear what's happening on the bit level.

Exceptions	Modes	Reasons
#GP(0)	P	Result in nonwritable segment
#GP(0)	P	Illegal memory effective address in CS, DS, ES, FS, or GS segments
#SS(0)	P	Illegal address in SS segment
#PF(fc)	P V	Page fault
INT(13)	R V	Some part of operand is outside of address range 0 to 0FFFFH

User Notes

There are many cases in which it is useful to negate a number, and NEG works for all of them. The one's complement of a word can be obtained with NOT.

Example

```
MOV   AX,579BH   ; Loads a value into AX.
NEG   AX         ; Sets AX to 0A865 hex.
```

NOP No Operation 8086

Opcode	Format	Clocks
90	NOP	3

Flags

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0									0		0		1	

Pseudocode

Do nothing for 3 clocks.

Operation

The NOP instruction takes up one byte of code space and when executed takes three clock cycles. No registers, flags, or contents of memory are changed. The NOP instruction is an alias for the instruction: "XCHG AX,AX."

Exceptions

None

User Notes

NOP does nothing, which is surprisingly often a worthwhile thing to do. It's used to fill space when debugging on the fly, or when patching an assembled program on disk. The assembler also puts NOPs in some of the

code it outputs. If it can't predict in advance the number of bytes a given instruction will take up, it allocates the maximum amount of space that could be needed. If not all the space is used the difference is filled with NOPs.

Someone writing a program for a system with super-critical timing or space limitations could write a filter to take in an assembled program and output the same program with all NOPs removed. The program would also have to adjust all jump-type instructions to account for the removed bytes.

Example

```
NOP   ; Does nothing.
```

NOT One's Complement Negation 8086

Opcode	Format	Clocks
F6 [2]	NOT r/mb	2/6
F7 [2]	NOT r/mw	2/6
F7 [2]	NOT r/md	2/6

Flags

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0									0		0		1	

Pseudocode

```
REPEAT
  Reverse a bit in the operand
UNTIL all bits in operand are reversed
```

Operation

NOT carries out a Boolean or "logical" NOT on its single operand and leaves the result in the operand. This operation is depicted in Figure 4-4, which gives "truth tables" for all logical operations. A 1 can be regarded as T or True, while a 0 corresponds to F or False. A logical NOT takes a single bit as operand and reverses it: if the bit is 0 it's changed it to 1; if it's a 1 it's changed to 0.

A logical NOT simply changes a statement to its opposite or negative, so applying NOT to an operand is also called "negating" the operand. In English this is like inserting the word 'not' into a statement: "The dog is black" becomes "The dog is NOT black."

The result of applying NOT to a number is called the "one's complement" of the number, and arithmetic can be done using one's complements.

Adding 1 to this result gives the "two's complement," which the iAPX 86s use to do math.

Exceptions	Modes	Reasons
#GP(0)	P	Result in nonwritable segment
#GP(0)	P	Illegal memory effective address in CS, DS, ES, FS, or GS segments
#SS(0)	P	Illegal address in SS segment
#PF(fc)	P V	Page fault
INT(13)	R V	Some part of operand is outside of address range 0 to 0FFFFH

User Notes

NOT is used to do bit tests and comparisons; many of these can also be done by using the bit test instructions which are new on the 80386 (a register-to-immediate bit test takes 3 cycles versus 2 for a similar NOT). The NOT is only more efficient if no other supporting instructions are needed to set up or decipher the comparison results.

Another problem with NOT (for many applications) is that it sets no flags.

Example

```
MOV   AX,579BH   ; Loads a value into AX.
NOT   AX         ; Sets AX to 0A864 hex.
```

OR — Or — 8086

Opcode	Format	Clocks
0C ib	OR AL,ib	2
0D iw	OR AX,iw	2
0D id	OR EAX,id	2
80 [1] ib	OR r/mb,ib	2/7
81 [1] iw	OR r/mw,iw	2/7
81 [1] id	OR r/md,id	2/7
08 [r]	OR r/mb,rb	2/7
09 [r]	OR r/mw,rw	2/7
09 [r]	OR r/md,rd	2/7
0A [r]	OR rb,r/mb	2/7
0B [r]	OR rw,r/mw	2/7
0B [r]	OR rd,r/md	2/7

Flags

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0			0				S	S	0		0	S	1	0

Pseudocode

```
REPEAT
  IF a bit in the destination operand is 0 and the corresponding bit in
       the source operand is 0 THEN
    leave the bit in the destination operand at 0
  ELSE
    set the bit in the destination operand to 1
  END IF
UNTIL all bits in destination operand are checked
```

Operation

OR carries out a Boolean or "logical" OR on its two operands and leaves the result in the leftmost operand. This operation is depicted in Figure 4-4, which gives "truth tables" for all logical operations. A 1 can be regarded as T or True, while a 0 corresponds to F or False. A logical 'OR' takes two bits and calculates a result using this rule: if either input bit is 1 or both input bits are 1, then the output bit is 1; otherwise the output bit is 0. The instruction OR simply does the same operation on all the bits in each of two operands; the leftmost bit in one operand is compared to the leftmost bit in the other, then the two bits one position to the right are compared, until all the bit pairs have been compared.

The logical OR is also called an "inclusive OR" since it "includes" the case where both statements are true. This is much like the use of "or" in English; if I say, "It's going to rain or snow tomorrow" then I'm proved right by rain, snow, or both.

OR can only be used with two operands of the same size (same number of bits); otherwise the operation would be meaningless for some of the bits in the longer operand.

Exceptions	Modes	Reasons
#GP(0)	P	Result in nonwritable segment
#GP(0)	P	Illegal memory effective address in CS, DS, ES, FS, or GS segments
#SS(0)	P	Illegal address in SS segment
#PF(fc)	P V	Page fault
INT(13)	R V	Some part of operand is outside of address range 0 to 0FFFFH

User Notes

OR is used to do bit setting; often this can also be done by using the bit-test instructions that are new on the 80386. A register to immediate bit test takes 3 cycles versus 2 for a similar OR, so the OR is only more efficient if no other supporting instructions are needed to set up for the bit setting.

Example

```
MOV   AX,5963H   ; Loads a hex number into AX.
MOV   BX,6CA5H   ; Loads a hex number into BX.
OR    AX,BX      ; AX now contains 7DE7 hex.
```

OUT Output to Port 8086

Opcode	Format	Clocks
E6 ib	OUT ib,AL	3
E7 ib	OUT ib,AX	3
E7 ib	OUT ib,EAX	3
EE	OUT DX,AL	4
EF	OUT DX,AX	4
EF	OUT DX,EAX	4

Flags

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0									0		0		1	

Pseudocode

```
IF first operand is immediate THEN
  Zero extend first operand to 16 bits to form output port address
ELSE
  Output port address is contents of DX
END IF
IF second operand size is byte THEN
  Move the byte in AL to the output port
ELSE IF second operand size is word THEN
  Move the word in AX to the output port
ELSE (* second operand size is dword *)
  Move the dword in EAX to the output port
END IF
```

Operation

The OUT instruction is used to send a single byte, word, or dword to a peripheral device port. A port number may be any number from 0 to 65,535

($2^{16}-1$). Normally a device has several ports assigned, some for commands, some for status, and some for data. Device control is accomplished by sending information to the command ports and getting information from the status ports. Output is produced by sending data to a data port.

Port numbers 00F8H through 00FFH are reserved by Intel and shouldn't be used.

Exceptions	Modes	Reasons
#GP(0)	P	Current privilege is higher than IOPL
#GP(0)	V	Some of the corresponding permission bits in TSS equal 1

User Notes

Most programs send output through calls to an operating system, and thus don't use the OUT instruction. Even those that bypass the operating system often use MOVs to write to video RAM and get information out. However, the OUT instruction is indispensable for those writing device drivers or for anyone who must deal directly with a device.

When only a few bytes of output are needed the OUT instruction should be used. If the output device can't accept data at a high rate of speed the OUT instruction can be put in a loop, with NOPs or a counting loop included to slow down the transfer of data.

Example

```
MOV   AL,20H   ; Loads an ASCII space into AL.
OUT   30,AL    ; Output it to port 30.
```

OUTS Output String to Port 80186

Opcode	Format	Clocks Single	Clocks Repeated
6E	OUTSB	7	5+5*N
6F	OUTSW	7	5+5*N
6F	OUTSD	7	5+5*N

The "N" in the "Clocks Repeated" column stands for the number in the (E)CX register.

Flags

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0									0		0		1	

Pseudocode

```
IF operand size is byte THEN
  Move the byte in AL to the output port named in DX
ELSE IF operand size is word THEN
  Move the word in AX to the output port named in DX
ELSE (* operand size is dword *)
  Move the dword in EAX to the output port named in DX
END IF
IF DF = 0 THEN
  Add size of operand (in bytes) to (E)SI
ELSE
  Subtract size of operand (in bytes) from (E)SI
END IF
```

Operation

The OUTS instruction, like OUT, is used to send a single byte, word, or dword to a peripheral device port. A port number may be any number from 0 to 65,535 ($2^{16}-1$). Normally a device has several ports assigned, some for commands, some for status, and some for data. Device control is accomplished by sending information to the command ports and getting information from the status ports. Output is produced by sending data to a data port.

There are several distinctions between OUT and OUTS. In OUTS the port number is always in DX, and the source of the data is pointed to by DS:[(E)SI] unless a segment override is used. Finally, the OUTS instruction is designed to be used with the REP prefix. That is, at the end of the instructions (E)SI is incremented or decremented (depending on the Direction Flag) by the operand size.

Port numbers 00F8H through 00FFH are reserved by Intel and shouldn't be used.

Exceptions	Modes	Reasons
#GP(0)	P	Current privilege is higher than IOPL
#GP(0)	P	Illegal memory effective address in CS, DS, ES, FS, or GS segments
#SS(0)	P	Illegal address in SS segment
#PF(fc)	P V	Page fault
INT(13)	R V	Some part of operand is outside of address range 0 to 0FFFFH
#GP(0)	V	Some of the corresponding permission bits in TSS equal 1

User Notes

Most programs send output through calls to an operating system, and thus don't use the OUTS instruction. Even those that bypass the operating system often use MOVs to write to video RAM and get information out. However, the OUTS instruction is indispensable for those writing device drivers or for anyone who must deal directly with a device.

If the output device can't accept data at a high rate of speed the OUTS instruction can be put in a loop, with NOPs or a counting loop included to slow down the transfer of data.

Example

```
CLD                     ; Ensure direction is foward.
LDS     ESI,OUTSTR      ; Set up source of output, DS:[ESI].
MOV     ECX,5           ; Set up repeat count for OUTS.
MOV     DX,40           ; Set up output port number for OUTS.
REP OUTSB               ; Send 5 bytes of data to output port 40.
```

POP Pop Stack to Operand 8086

Opcode	Format	Clocks
8F [0]	POP mw	5
8F [0]	POP md	5
58+rw	POP rw	4
58+rd	POP rd	4
1F	POP DS	7,*
07	POP ES	7,*
17	POP SS	7,*
0F A1	POP FS	7,*
0F A9	POP GS	7,*

*These instructions require 21 clocks in Protected Mode due to extra processing needs (see Chapter 5).

Flags

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0									0		0		1	

Pseudocode

```
IF operand size is word THEN
  Move the word at SS:[(E)SP] to the destination word
  Add 2 to (E)SP
ELSE (* operand size is dword *)
  Move the dword at SS:[(E)SP] to the destination dword
  Add 4 to (E)SP
END IF
```

Operation

In general the POP instruction moves the word on the top of the stack into the operand given by the instruction, leaving the top of the stack pointing to a different word.

In particular, POP copies the word pointed to by SS:SP into the operand. Then SP is set to SP + 2 (or 4). Since the stack starts at address SS and grows downward, incrementing SP has the effect of making the stack smaller. The word that SP now points to is the word above the popped word in RAM, but is regarded as below the POPped word on the stack.

Although this is a little confusing, through sheer necessity most programmers become familiar with accessing the stack. There are two additional complications encountered in using the POP instruction. The first is that when a POP SS is executed in Real Mode this automatically inhibits all interrupts until after the next instruction is executed. The implied expectation is that the next instruction will be POP SP or some other instruction which results in the SS:SP combination being restored to a meaningful value.

The second complication is encountered when POPping into any segment register in Protected Mode. The main problem here is that any change in one of these registers means that the processor will now be trying to reference a whole different piece of memory which probably has different protection levels than the current one. Chapter 5 explains the protection checks which must be cleared to POP into a segment register when in Protected Mode.

Exceptions	Modes	Reasons
#GP, #SS, #NP	P	A segment register is being loaded
#GP(0)	P	Result in nonwritable segment
#GP(0)	P	Illegal memory effective address in CS, DS, ES, FS, or GS segments
#SS(0)	P	Illegal address in SS segment
#PF(fc)	P V	Page fault
INT(13)	R V	Some part of operand is outside of address range 0 to 0FFFFH

User Notes

The most important thing to know about using the stack is that within a given subroutine the number of PUSHes and the number of POPs must match. Mismatches here cause major problems at execution time. The well-behaved ones result in your program blowing up spectacularly and immediately; the subtle ones don't manifest themselves until your program has been sold to the public for a year, at which time it's very expensive to fix.

Several programming tricks concern the stack. One is to go back and reaccess words you've already popped, figuring they'll still be in the same place in RAM as they were before. This often comes about when large data structures have been stashed on the stack instead of placed at a well-defined or specially protected place in memory. Another is to place strange values on the stack and then immediately execute a RET or IRET, knowing that execution will then transfer to the new value. Many of these tricks will fail to work on newer iAPX 86s as the word size increases and protection restrictions increase. Some of the rest will be rendered troublesome as successive generations of programmers (a generation being about two years) try to modify existing programs, innocently assuming that the stack is being handled in a normal way.

Example

See PUSH (below) for an example using POP.

POPA Pop All General Registers 80186

Opcode	Format	Clocks
61	POPA	24
61	POPAD	24

Flags

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0									0		0		1	

Pseudocode

```
(* Refer to POP instruction for clarification of POP operation *)
IF operand size is word THEN
  POP DI
  POP SI
  POP BP
  POP BX (* Discard SP value from stack *)
  POP BX
  POP DX
  POP CX
  POP AX
ELSE (* operand size is dword *)
  POP EDI
  POP ESI
  POP EBP
  POP EBX (* Discard ESP value from stack *)
  POP EBX
  POP EDX
  POP ECX
  POP EAX
END IF
```

Operation

POPA pops the eight words on top of the stack into the general registers (E)DI, (E)SI, (E)BP, (E)SP, (E)BX, (E)DX, (E)CX, and (E)AX (in that order). Note, however, that the value POPped for (E)SP is not stored there, but discarded. The new value of (E)SP is as if eight pop instructions were executed.

Exceptions	Modes	Reasons
#SS(0)	P	Illegal address in SS segment
#PF(fc)	P V	Page fault
INT(13)	R V	Some part of operand is outside of address range 0 to 0FFFFH

User Notes

This instruction replaces eight separate POP instructions, and executes in 8 fewer clocks while taking only 1/8 the space in code. The main advantage of POPA, though, is the conceptual simplicity of simply getting all the registers loaded from the stack at once. Many hard-to-find bugs have been caused by subroutines that PUSHed some registers at the beginning and then POPped them back in the wrong order at the end.

If code space is at a premium, POPA (preceded in almost all cases by its counterpart PUSHA) should be used even if only two or three registers actually need to be saved on the stack. If execution time is at a premium (as it increasingly will be in multiuser environments), remember that the break-even point is six registers PUSHed and later POPped.

POPA makes a lot of sense when used to implement high-level languages whose unpredictable levels of nesting often require that all registers be saved and restored for each subroutine call.

Knowing the order in which POPA loads the registers makes possible several tricky ways of getting new values into the registers while mostly bypassing protection mechanisms. While this can be fast and efficient, it can also lead to obscure and troublesome bugs, especially in Protected Mode.

Example

See PUSHA (below) for an example using POPA.

POPF Pop Flags 8086

Opcode	Format	Clocks
9D	POPF	5
9D	POPFD	5

Flags

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0	S	* *	S	S	*	S	S	S	0	S	0	S	1	S

*IOPL is altered only at privilege level 0; IF is altered only when privilege is less than or equal to IOPL.

Pseudocode

```
(* Refer to POP instruction for clarification of POP operation *)
IF operand size is word THEN
  POP into FLAGS
ELSE (* operand size is dword *)
  POP into EFLAGS
END IF
```

Operation

The POPF instruction copies the word pointed to by SS:[(E)SP] into the flags register and then increments SP by 2 (or 4). Since the stack starts at address SS and grows downward, incrementing SP has the effect of making the stack smaller.

The flags copied into the register are, in order from most significant (bit 15) to least significant (bit 0): x, nested task, I/O privilege level (2 bits), overflow, direction, interrupts enabled, trap, sign, zero, x, auxiliary carry, x, parity, x, and carry ("x" for undefined). Note that the VM (virtual memory) and

RF (resume) flags are not altered by this instruction. In addition, the IOPL and IF flags may not be altered unless the current task has sufficient privilege. In this last case no exception is generated.

Exceptions	Modes	Reasons
#SS(0)	P	Illegal address in SS segment
INT(13)	R	Some part of operand is outside of address range 0 to 0FFFFH
#GP(0)	V	Used to emulate the instruction
#GP(0)	V	IOPL is less than 3

User Notes

The POPF instruction, like the POPA instruction, is basically designed to help implement high-level language compilers. With its counterpart PUSHF, POPF enables all the flags to be saved and restored *en masse* so the compiler doesn't have to "think" about which ones need to be saved and which don't at each subroutine call. PUSHF and POPF are almost always executed either just before (PUSHF) and just after (POPF) a subroutine call, or at the start (PUSHF) and end (POPF) of a subroutine.

Of course POPF can be used by human assembly-language programmers as well. In addition to its use with PUSHF it can also be used to quickly load the entire flags register with an appropriate bit pattern; just PUSH the bit pattern onto the stack and then POPF it into the flags register. This is fast and a little dangerous, since it changes not only the conditional flags like Zero and Carry, but also more sensitive ones such as I/O Privilege Level and Nested Task.

Example

See PUSHF (below) for an example using POPF.

PUSH Push Operand onto Stack 8086

Opcode	Format	Clocks
FF [6]	PUSH mw	5
FF [6]	PUSH md	5
50+r	PUSH rw	2
50+r	PUSH rd	2
6A ib	PUSH ib	2
68 iw	PUSH iw	2
68 id	PUSH id	2
0E	PUSH CS	2
1E	PUSH DS	2
06	PUSH ES	2
16	PUSH SS	2
0F A0	PUSH FS	2
0F A8	PUSH GS	2

Flags

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0									0		0		1	

Pseudocode

```
IF operand size is word THEN
   Subtract 2 from (E)SP
   Move the source word to SS:[(E)SP]
ELSE (* operand size is dword *)
   Subtract 4 from (E)SP
   Move the source dword to SS:[(E)SP]
END IF
```

Operation

In general the PUSH instruction puts its operand onto the top of the stack, changing the stack pointer so that the new value is the new top of the stack.

In particular, PUSH changes SP to SP − 2 (or 4). Since the stack starts at address SS and grows downward, decrementing SP has the effect of making the stack larger. The word that SP now points to is the word below (at a lower RAM address than) the last word pushed, but it is regarded as "above" that word on the stack. The contents of PUSH's operand are now copied to the new top of the stack.

Although this is a little confusing, through sheer necessity most programmers become familiar with accessing the stack. There are two additional complications encountered in using the PUSH instruction. The first occurs when a PUSH SP is executed. On early iAPX 86s this causes the chain of events one would normally expect from looking at the pseudocode above: SP is decremented by 2, and then its value is pushed onto the stack. However, on the iAPX 286 and 386 the value pushed is SP *before* the decrement, which is the effect programmers are usually trying to achieve when they push SP.

The second is almost always ignorable: if (E)SP is 1 when PUSH is executed, the 80386 will shut down due to lack of stack space. It is unlikely that this problem would ever confront you.

Exceptions	Modes	Reasons
#GP(0)	P	Illegal memory effective address in CS, DS, ES, FS, or GS segments
#SS(0)	P	Illegal address in SS segment
#PF(fc)	P V	Page fault
*****	R V	System shut down due to lack of stack space

User Notes

The most important thing to know about using the stack is that within a given subroutine the number of PUSHes and the number of POPs must match. Mismatches here cause major problems at execution time. The well-behaved ones result in your program blowing up spectacularly and immediately; the subtle ones don't manifest themselves until your program has been sold to the public for a year, at which time it's very expensive to fix.

Several programming tricks concern the stack. One is to PUSH several words and then change the stack pointer so they won't be interfered with as subroutines call each other. When the data is no longer needed, restore SP to its previous value. This often comes about when large data structures are stashed on the stack instead of placed at a well-defined or specially protected place in memory. Another is to place strange values on the stack and then immediately execute a RET or IRET, knowing that execution will then transfer to the new value. Many of these tricks will fail to work on newer iAPX 86s as the word size increases and protection restrictions increase. Some of the rest will be rendered troublesome as successive generations of programmers (a generation being about two years) try to modify existing programs, innocently assuming that the stack is being handled in a normal way.

Example

```
PUSH   EAX              ; Registers are full, make one
                        ; available.
IMUL   EAX,MEMLOC,10    ; Multiply a memory value
                        ; by 10 and
MOV    MEMLOC,EAX       ; store it back where it came
                        ; from.
POP    EAX              ; Restore old value of EAX.
```

PUSHA Push All General Registers 80186

Opcode	Format	Clocks
60	PUSHA	18
60	PUSHAD	18

Flags

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0									0		0		1	

Pseudocode

```
(* Refer to PUSH instruction for clarification of PUSH operation *)
IF operand size is word THEN
   Save value of SP in an internal register
   PUSH AX
   PUSH CX
   PUSH DX
   PUSH BX
   PUSH saved SP value
   PUSH BP
   PUSH SI
   PUSH DI
ELSE (* operand size is dword *)
   Save value of ESP in an internal register
   PUSH EAX
   PUSH ECX
   PUSH EDX
   PUSH EBX
   PUSH saved ESP value
   PUSH EBP
```

```
    PUSH ESI
    PUSH EDI
END IF
```

Operation

PUSHA pushes the eight words on top of the stack from the general registers (E)AX, (E)CX, (E)DX, (E)BX, (E)SP, (E)BP, (E)SI, and (E)DI (in that order). Note, however, that the value PUSHed for (E)SP is the value before the instruction began to execute. The new value of (E)SP is as if eight PUSH instructions were executed.

If SP is 1, 3, or 5 before PUSHA is executed, the 80386 will shut down without executing it. If SP is an odd number between 7 and 15, exception 13 will occur.

Exceptions	Modes	Reasons
#SS(0)	P	Illegal address in SS segment
#PF(fc)	P V	Page fault
*****	R V	System shut down due to lack of stack space
INT(13)	R V	Some part of operand is outside of address range 0 to 0FFFFH

User Notes

This instruction replaces eight separate PUSH instructions and executes in only 2 extra clocks, while taking only 1/8 the space in code. The main advantage of PUSHA, though, is the conceptual simplicity of simply getting all the registers put on the stack at once. Many hard-to-find bugs have been caused by subroutines that PUSHed some registers at the beginning and then POPped them back in the wrong order at the end.

If code space is at a premium PUSHA (followed in almost all cases by its counterpart POPA) should be used even if only two or three registers actually need to be saved on the stack. If execution time is at a premium (as it increasingly will be in multiuser environments) PUSHA gives no real advantage; POPA, however, gives a slight advantage when at least six of the registers need to be PUSHed and later POPped.

PUSHA makes a lot of sense when used to implement high-level languages whose unpredictable levels of nesting often require that all registers be saved and restored for each subroutine call.

Example

```
SUBROUTINE:
        PUSHA       ; Save all registers on the stack.
        :       :   :
        POPA        ; Restore all registers from the stack.
        RET         ; Return from subroutine.
```

PUSHF Push Flags 8086

Opcode	Format	Clocks
9C	PUSHF	4
9C	PUSHFD	4

Flags

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0									0		0		1	

Pseudocode

```
(* Refer to PUSH instruction for clarification of PUSH operation *)
IF operand size is word THEN
  PUSH from FLAGS
ELSE (* operand size is dword *)
  PUSH from EFLAGS
END IF
```

Operation

The PUSHF instruction sets (E)SP to (E)SP − 2 (or 4) and then copies the (E)FLAGS register into the word pointed to by SS:[(E)SP]. Since the stack starts at address SS and grows downward, decrementing (E)SP has the effect of making the stack larger.

The flags copied onto the stack are, in order from most significant (bit 15) to least significant (bit 0): x, nested task, I/O privilege level (2 bits), overflow, direction, interrupts enabled, trap, sign, zero, x, auxiliary carry, x, parity, x, and carry ("x" for undefined).

The 80386 will shut down in Real Mode if SP = 1, due to lack of stack space.

Exceptions	Modes	Reasons
#SS(0)	P	Illegal address in SS segment
*****	R	System shut down due to lack of stack space
#GP(0)	V	Used to emulate the instruction
#GP(0)	V	IOPL is less than 3

User Notes

The PUSHF instruction, like the PUSHA instruction, is basically designed to help implement high-level language compilers. With its counterpart POPF, PUSHF enables all the flags to be saved and restored *en masse* so the compiler doesn't have to "think" about which ones need to be saved and which don't at each subroutine call. PUSHF and POPF are almost always executed either just before (PUSHF) and just after (POPF) a subroutine call or at the start (PUSHF) and end (POPF) of a subroutine.

Of course, PUSHF can be used by human assembly-language programmers as well. In addition to its use with POPF, it can also be used to quickly get the entire flags register onto the stack, and then perhaps to another register. From here, it can be ANDed with 7FD5H to make sure the undefined bits are clear, then Compared with any of a number of bit patterns. JE and JNZ, for example, could then be the final step in implementing homemade conditional Jumps.

Example

```
PUSHF                   ; Save the flags on the stack.
OR      SS:[SP],800H    ; Set the overflow flag.
POPF                    ; Restore modified flags from the stack.
```

REPcc — Repeat While Condition is Met (prefix) — 8086

Opcode	Format	Repeat Condition	String Instructions	Clocks
F2 *	REP *	(E)CX > 0	INS, MOVS, OUTS, STOS	*
F3 *	REPE *	(E)CX > 0 and ZF = 1	CMPS, SCAS	*
F2 *	REPNE *	(E)CX > 0 and ZF = 0	CMPS, SCAS	*
F2 *	REPNZ *	(E)CX > 0 and ZF = 0	CMPS, SCAS	*
F3 *	REPZ *	(E)CX > 0 and ZF = 1	CMPS, SCAS	*

*See description of individual string instructions.

Flags

The repeat prefixes do not affect the flags, but some of the individual string instructions do. See the individual descriptions for full details on flags affected for each instruction.

Pseudocode

```
IF (CX ≠ 0) THEN (* if CX is initially zero, the loop is not executed *)
  REPEAT
    Respond to any pending interrupts
    Perform the string operation which REPcc is a prefix to
    Decrement CX by 1 (* No flags are conditioned *)
  UNTIL repeat condition is not met (* if ZF is checked it is after
    execution of the instruction *)
END IF
```

Operation

The REP group of prefixes is used only with the string instructions as listed in the table above. When one of these prefixes is used (E)CX is compared to 0. If it is zero the prefixed string instruction is not executed. If it

is not zero the string instruction is repeatedly executed until either (E)CX becomes zero or (for REPE, REPNE, REPNZ, and REPZ) the Zero Flag no longer has the correct value.

At the start of each loop iteration, interrupts are handled. The string instruction is then executed normally. Next CX is decremented without affecting any flags. This loop repeats as long as the repeat condition (see above table for each prefix) still holds.

Note that the loop is not executed at all if CX is initially zero, but is executed at least once no matter what the state of the Zero Flag.

The string instructions listed are specially designed to work with the REP type prefixes. They automatically adjust the SI and DI pointers during execution. If the Direction Flag is 0 the pointers are incremented by 1 for each byte moved; if the Direction Flag is 1 the pointers are decremented by 1 for each byte moved.

Exceptions

The repeat prefixes do not generate exceptions, but some of the individual string instructions do. See the individual descriptions for full details on exceptions for each instruction.

User Notes

There are three advantages to using REP and related prefixes. The first is that they are compact; a single line can hold the equivalent of perhaps six non-string instructions. The second is that execution is very fast; the checking that controls the looping is faster by 10 or more clocks than if it were handled by other instructions. This speed saving adds up with every repetition of the loop. The third advantage is the conceptual ease of using REPcc. What is in most cases a single thought in the programmer's mind (move *this* string to *there*) translates into a single line in the program, a rare situation in assembler.

There is one thing to beware of when using these prefixes. Since they do so much in a single line, they are a likely hiding place for bugs. For instance, not initializing the CX register properly will result in a string of up to 65,535 characters being loaded, moved, or output, so the CX register may need a range check before the repeated instruction is executed.

Another thing needs to be checked when using REP INS or REP OUTS. Not all input and output ports can receive or send characters fast enough to keep up with REP loops. If this is the case a slower loop will need to be substituted. This is especially worth considering when some of your users will be running your program on a different machine than you're testing on.

The JCXZ and zero-flag jumps (JZ/JE and JNZ/JNE) can be used just after the REPcc to distinguish between loops that stop because of the CX register reaching 0 and loops that stop because of the Zero Flag.

Example

```
LDS     SI,SRC_STR      ; Set up source pointer for string move.
LES     DI,DEST_STR     ; Set up destination pointer for string
                        ; move.
MOV     ECX,STRLEN      ; Number of bytes to move.
REP MOVSB               ; Move SRC_STR to DEST_STR.
```

RET Return from CALL 8086

Opcode	Format	Type	Clocks
C3	RET	Near	10+m
CB	RET	Far	18+m,*
C2 iw	RET iw	Near, pop parameters	10+m
CA iw	RET iw	Far, pop parameters	18+m,*

*These instructions have varying functions and timings in Protected Mode (see Chapter 5).

Flags

Normally RET affects no flags. However, when a task switch is made in protected mode, all flags are changed to the old task's saved flags.

Pseudocode

```
POP IP (Instruction Pointer) from stack
IF far return THEN
  POP CS (Code Segment) from stack
END IF
IF byte count given THEN
  Pop that many bytes from the stack
END IF
```

Operation

A RET from a CALL in the same code segment as the subroutine (near return) is a simple operation. The instruction pointer (IP) is POPped from the top of the stack. If the RET is from a CALL in another segment (far return) the code segment (CS) is also POPped. In either case execution then resumes from the address made up of CS:IP.

As an aid to passing parameters on the stack, an alternate form of RET lets the programmer specify the number of bytes of parameters that exist on

the stack. The processor will then remove them from the stack before returning control to the calling procedure.

The basic procedure is the same in Protected Mode, but inter-segment returns are much more complicated, since the CALL may have specified an operating system routine or even another task. In either case memory protection must be checked (see Chapter 5).

Exceptions	Modes	Reasons
#NP	P	Target code segment not present
#TS	P	Task switch required
#GP(0)	P	Illegal memory effective address in CS, DS, ES, FS, or GS segments
#SS(0)	P	Illegal address in SS segment
#PF(fc)	P V	Page fault
INT(13)	R V	Some part of operand is outside of address range 0 to 0FFFFH

User Notes

Some of the most popular programming tricks are executed by putting various values on the stack and then executing a RET. Self-contained Case statements can be built this way, for instance. As memory-management on the 86 family gets more complicated, however, these tricks will become increasingly dangerous. For example, on the 80386 the segment size is not fixed, so a RET to an address within the 64 Kb segment range is not necessarily in the same segment anymore. Another problem is that virtual memory is now available, meaning that the segment you're returning to may not even be in memory when you start to return to it. Any tricks which fool the chip may also fool the operating system, with unknown results.

Example

```
RET  8  ; Pops 8 bytes of parameters off the stack.
```

Rxx Rotate 8086

Opcode	Format	Clocks
D0 [2]	RCL r/mb,1	9/10
D2 [2]	RCL r/mb,CL	9/10
C0 [2] ib	RCL r/mb,ib	9/10
D1 [2]	RCL r/mw,1	9/10
D3 [2]	RCL r/mw,CL	9/10
C1 [2] ib	RCL r/mw,ib	9/10
D1 [2]	RCL r/md,1	9/10
D3 [2]	RCL r/md,CL	9/10
C1 [2] ib	RCL r/md,ib	9/10
D0 [3]	RCR r/mb,1	9/10
D2 [3]	RCR r/mb,CL	9/10
C0 [3] ib	RCR r/mb,ib	9/10
D1 [3]	RCR r/mw,1	9/10
D3 [3]	RCR r/mw,CL	9/10
C1 [3] ib	RCR r/mw,ib	9/10
D1 [3]	RCR r/md,1	9/10
D3 [3]	RCR r/md,CL	9/10
C1 [3] ib	RCR r/md,ib	9/10
D0 [0]	ROL r/mb,1	3/7
D2 [0]	ROL r/mb,CL	3/7
C0 [0] ib	ROL r/mb,ib	3/7
D1 [0]	ROL r/mw,1	3/7
D3 [0]	ROL r/mw,CL	3/7
C1 [0] ib	ROL r/mw,ib	3/7
D1 [0]	ROL r/md,1	3/7
D3 [0]	ROL r/md,CL	3/7
C1 [0] ib	ROL r/md,ib	3/7
D0 [1]	ROR r/mb,1	3/7
D2 [1]	ROR r/mb,CL	3/7
C0 [1] ib	ROR r/mb,ib	3/7
D1 [1]	ROR r/mw,1	3/7
D3 [1]	ROR r/mw,CL	3/7
C1 [1] ib	ROR r/mw,ib	3/7

D1 [1]	ROR r/md,1	3/7
D3 [1]	ROR r/md,CL	3/7
C1 [1] ib	ROR r/md,ib	3/7

Flags

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0			*						0		0		1	S

*OF is set only on single bit rotates. Multi-bit rotates leave OF in an undefined state.

Pseudocode

```
Place first operand in an internal register
DO second operand TIMES
  IF rotate direction is left THEN
    Save high order bit
  ELSE
    Save low order bit
  END IF
  Shift one bit in the rotate direction
  IF CF is involved THEN
    IF rotate direction is left THEN
      Place CF into low order bit
    ELSE
      Place CF into high order bit
    END IF
  ELSE
    IF rotate direction is left THEN
      Place saved bit into low order bit
    ELSE
      Place saved bit into high order bit
    END IF
  END IF
  Place saved bit into CF
```

```
ENDDO
IF second operand is 1 THEN
   IF rotate direction is left THEN
      IF high order bit ≠ CF THEN
         SET OF to 1
      ELSE
         Clear OF to 0
      END IF
   ELSE
      IF high order bit ≠ next to high order bit THEN
         Set OF to 1
      ELSE
         Clear OF to 0
      END IF
   END IF
END IF
Store internal register into first operand
```

Operation

Rotate instructions are like shifts in that the bit pattern in the first operand is moved either to the left or right by the number of places in the second operand. The difference is that the bits rotated out of one end of the operand are not lost as in a shift. These bits are brought back in at the opposite end of the operand.

The second and third letters in the instruction mnemonics control different aspects of the rotation. The third letter controls the rotate direction, "L" for left and "R" for right. The second letter controls how the Carry Flag is involved. If the second letter is "O" (plain rotates) then the Carry Flag simply contains the last bit that was moved from one end of the operand to the other. If the second letter of the instruction mnemonic is "C" (rotates through carry) then the Carry Flag is actually treated as part of the operand to be rotated. In this case a bit rotated out of one end of an operand is placed into the Carry Flag, but first the old value of the Carry Flag is moved into the vacated bit position at the opposite end of the operand.

The second operand can be either an immediate number or the contents of the CL register. However, only rotate counts of 31 or less are allowed. If the count is greater than 31 only the lowest five bits are used. In addition,

there is a special short form of the instruction for the special case of a rotate count of 1.

Exceptions	Modes	Reasons
#GP(0)	P	Result in nonwritable segment
#GP(0)	P	Illegal memory effective address in CS, DS, ES, FS, or GS segments
#SS(0)	P	Illegal address in SS segment
#PF(fc)	P V	Page fault
INT(13)	R V	Some part of operand is outside of address range 0 to 0FFFFH

User Notes

One use of rotates on older processors was to speed special cases of multi-precision multiplies and divides. The 80386 with its greater word size and fast multiply and divide instructions obviates the need for much of this trickery.

Another use for rotates is to get bits into the Carry Flag, where a JC or JNC can branch on the bit's value. The bit test instructions (new on the 80386) allow this test to be done more directly.

Example

```
MOV   EAX,0CADE4956H    ; Loads a value into EAX.
STC                     ; Ensure that the carry flag is set.
RCL   EAX,3             ; Sets EAX to 56F24AB7 and CF to 0.
```

SAHF Store AH Into Flags 8086

Opcode	Format	Clocks
9E	SAHF	3

Flags

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0							S	S	0	S	0	S	1	S

Pseudocode

Move AH value into low byte of Flags register.

Operation

SAHF is a quick way to set all the flags in the low byte of the Flags register to the bit pattern in AH after they have been set, reset, examined, or saved as needed. This instruction has been provided for compatability with the 8080/85, and is rarely used by iAPX 86 family programs, which would use POPF instead.

The bits taken from AH are: SF ZF x AF x PF x CF, with "x" meaning undefined.

Exceptions

None

User Notes

Except for programs converted from (or due to be converted to) the 8080/85 this instruction is normally not used. It could be used to set the status of all the low-byte flags at once (by loading AH with a bit pattern, for instance).

Example

```
LAHF        ; Loads the flags into AH.
OR    AH,4  ; Sets image of parity flag in AH.
SAHF        ; Simulates a "set parity flag" instruction.
```

Sxx Shift 8086

Opcode	Format	Clocks
D0 [4]	SAL r/mb,1	3/7
D2 [4]	SAL r/mb,CL	3/7
C0 [4] ib	SAL r/mb,ib	3/7
D1 [4]	SAL r/mw,1	3/7
D3 [4]	SAL r/mw,CL	3/7
C1 [4] ib	SAL r/mw,ib	3/7
D1 [4]	SAL r/md,1	3/7
D3 [4]	SAL r/md,CL	3/7
C1 [4] ib	SAL r/md,ib	3/7
D0 [7]	SAR r/mb,1	3/7
D2 [7]	SAR r/mb,CL	3/7
C0 [7] ib	SAR r/mb,ib	3/7
D1 [7]	SAR r/mw,1	3/7
D3 [7]	SAR r/mw,CL	3/7
C1 [7] ib	SAR r/mw,ib	3/7
D1 [7]	SAR r/md,1	3/7
D3 [7]	SAR r/md,CL	3/7
C1 [7] ib	SAR r/md,ib	3/7
D0 [4]	SHL r/mb,1	3/7
D2 [4]	SHL r/mb,CL	3/7
C0 [4] ib	SHL r/mb,ib	3/7
D1 [4]	SHL r/mw,1	3/7
D3 [4]	SHL r/mw,CL	3/7
C1 [4] ib	SHL r/mw,ib	3/7
D1 [4]	SHL r/md,1	3/7
D3 [4]	SHL r/md,CL	3/7
C1 [4] ib	SHL r/md,ib	3/7
D0 [5]	SHR r/mb,1	3/7
D2 [5]	SHR r/mb,CL	3/7
C0 [5] ib	SHR r/mb,ib	3/7
D1 [5]	SHR r/mw,1	3/7
D3 [5]	SHR r/mw,CL	3/7
C1 [5] ib	SHR r/mw,ib	3/7
D1 [5]	SHR r/md,1	3/7

D3 [5]	SHR r/md,CL	3/7
C1 [5] ib	SHR r/md,ib	3/7

Flags

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0			*				S	S	0		0	S	1	S

*OF is set only on single bit shifts. Multi-bit shifts leave OF in an undefined state.

Pseudocode

```
Place first operand in an internal register
DO second operand TIMES
  IF shift direction is left THEN
    Place high order bit into CF
  ELSE
    Place low order bit into CF
  END IF
  Shift one bit in the shift direction
  IF shift direction is left THEN
    Place 0 into low order bit
  ELSE IF instruction is SHR THEN
    Place 0 into high order bit
  ELSE (* instruction is SAR *)
    Place old high order bit into high order bit
  END IF
END DO
IF second operand is 1 THEN
  IF shift direction is left THEN
    IF high order bit ≠ CF THEN
      Set OF to 1
    ELSE
      Clear OF to 0
    END IF
  ELSE IF instruction is SHR THEN
    Clear OF to 0
```

```
    ELSE (* instruction is SAR *)
      Set OF to high order bit
    END IF
  END IF
  Store internal register into first operand
```

Operation

Shift instructions move the pattern of bits in the first operand either to the left or right by the number of places in the second operand. This action will cause some bits to disappear from one end of the operand and cause the same number of vacated bits to be filled at the opposite end of the operand. The manner of filling depends on the instruction used.

The second and third letters in the instruction mnemonics control different aspects of the shift. The third letter controls the shift direction, "L" for left and "R" for right. The second letter controls whether we have an arithmetic (second letter "A") or logical (second letter "H") shift. A logical shift always fills vacated bit positions with zeros. An SAL instruction fills vacated positions (on the right) with zeros. On the other hand, an SAR instruction fills vacated positions (on the left) with copies of the value of the sign bit before the shift.

The second operand can be either an immediate number or the contents of the CL register. However, only shift counts of 31 or less are allowed. If the count is greater than 31 only the lowest five bits are used. In addition there is a special short form of the instruction for the special case of a shift count of 1.

Exceptions	Modes	Reasons
#GP(0)	P	Result in nonwritable segment
#GP(0)	P	Illegal memory effective address in CS, DS, ES, FS, or GS segments
#SS(0)	P	Illegal address in SS segment
#PF(fc)	P V	Page fault
INT(13)	R V	Some part of operand is outside of address range 0 to 0FFFFH

User Notes

One use of shifts on older processors was to speed special cases of single precision multiplies and divides. The 80386 with its fast multiply and divide instructions obviates the need for much of this trickery. If you need to use these instructions for that purpose, remember to use the arithmetic forms for signed numbers and the logical forms for unsigned numbers.

Another use for shifts is to get bits into the Carry Flag, where a JC or JNC can branch on the bit's value. The bit test instructions (new on the 80386) allow this test to be done more directly.

Example

```
MOV   EAX,0CADE4956H   ; Loads a value into EAX.
SAR   EAX,3            ; Sets EAX to 0F95BC92AH and CF
                       ; to 0.
```

SHxD Shift Double 80386

Opcode	Format	Clocks
0F A4 [r] ib	SHLD r/mw,rw,ib	3/7
0F A4 [r] ib	SHLD r/md,rd,ib	3/7
0F A5 [r]	SHLD r/mw,rw,CL	3/7
0F A5 [r]	SHLD r/md,rd,CL	3/7
0F AC [r] ib	SHRD r/mw,rw,ib	3/7
0F AC [r] ib	SHRD r/md,rd,ib	3/7
0F AD [r]	SHRD r/mw,rw,CL	3/7
0F AD [r]	SHRD r/md,rd,CL	3/7

Flags

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0			U				S	S	0	U	0	S	1	S

Pseudocode

```
Place first operand in an internal register IR1
Place second operand in an internal register IR2
DO third operand TIMES
  IF shift direction is left THEN
    Place high order bit of IR2 into CF
  ELSE
    Place low order bit of IR2 into CF
  END IF
  Shift IR2 one bit in the shift direction
  Shift IR1 one bit in the shift direction
  IF shift direction is left THEN
    Place CF into low order bit of IR1
  ELSE
    Place CF into high order bit of IR1
```

```
  END IF
END DO
Store IR1 into first operand
```

Operation

The shift double instructions move the pattern of bits in the first operand either to the left or right by the number of places in the third operand. This action will cause some bits to disappear from one end of the operand and cause the same number of vacated bits to be filled at the opposite end of the operand. The vacated bits are filled from the second operand as if it too had been shifted by the same number of bits and the bits shifted off one end were shifted into the first operand. Note that the second operand is unchanged by this instruction.

The third letter in the instruction mnemonic controls the shift direction, "L" for left and "R" for right. The effect of the instruction is as if operands one and two were taken as one double-sized number and shifted. The new value of operand one is then stored, but operand two is not stored. For SHLD operand one is on the left and operand two is on the right. The opposite is true for SHRD.

The third operand can be either an immediate number or the contents of the CL register. However, only shift counts of 31 or less are allowed. If the count is greater than 31 only the lowest five bits are used. If the operand size is 16 bits (word) and the shift count is greater than 15, the instruction sets its first operand and all the flags to an undefined state.

Exceptions	Modes	Reasons
#GP(0)	P	Result in nonwritable segment
#GP(0)	P	Illegal memory effective address in CS, DS, ES, FS, or GS segments
#SS(0)	P	Illegal address in SS segment
#PF(fc)	P V	Page fault
INT(13)	R V	Some part of operand is outside of address range 0 to 0FFFFH

User Notes

The shift double instructions behave similarly to a logical shift (see Sxx for SHL and SHR) of the first operand by the third operand. The difference is that the bits vacated by the shift are replaced by bits from the opposite end of the second operand instead of with zeros.

These new shift instructions are ideal for implementing a multiple precision shift with a loop. Any type of shift or rotate can be handled just by minor modifications to the loop structure.

Example

```
;;;;;;;;;;;;;;
;
;  BLOCK is the offset in the current data segment (pointed to by DS) of
;  an array of LENGTH dwords that is to be treated as one long bit
;  string and rotated left by 10 bits.
;
;;;;;;;;;;;;;;
    MOV     EDX,BLOCK               ; Save value of first dword for
                                    ; after the loop.
    MOV     ESI,0                   ; Set up index register to point to
                                    ; first dword.
    MOV     ECX,LENGTH              ; Get number of dwords in
                                    ; BLOCK and
    DEC     ECX                     ; subtract one for our loop count.
DELOOP:
    MOV     EAX,BLOCK+4[ESI*4]      ; Load next dword from the array
                                    ; at BLOCK.
    SHLD    BLOCK[ESI*4],EAX,10     ; Shift current dword, filling from
                                    ; next.
    INC     ESI                     ; Increment index for next time
                                    ; through the loop.
    LOOP    DELOOP                  ; Loop until all dwords are done
                                    ; except the last.
    SHLD    BLOCK[ESI*4],EDX,10     ; Shift last dword, filling from
                                    ; original value of first.
```

SBB Subtract With Borrow 8086

Opcode	Format	Clocks
1C ib	SBB AL,ib	2
1D iw	SBB AX,iw	2
1D id	SBB EAX,id	2
80 [3] ib	SBB r/mb,ib	2/7
81 [3] iw	SBB r/mw,iw	2/7
81 [3] id	SBB r/md,id	2/7
83 [3] ib	SBB r/mw,ib	2/7
83 [3] ib	SBB r/md,ib	2/7
18 [r]	SBB r/mb,rb	2/7
19 [r]	SBB r/mw,rw	2/7
19 [r]	SBB r/md,rd	2/7
1A [r]	SBB rb,r/mb	2/6
1B [r]	SBB rw,r/mw	2/6
1B [r]	SBB rd,r/md	2/6

Flags

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0			S				S	S	0	S	0	S	1	S

Pseudocode

IF (source operand has fewer bits than destination) THEN
  sign-extend source operand
END IF
Subtract source operand from destination.
Subtract CF from destination, place result in destination operand.

Operation

SBB subtracts the second operand plus the Carry Flag from the first operand. The first operand is overwritten by the result, while the second operand is unchanged. A good translation of an ADC instruction into English might be, "Subtract operand number 2 from operand number 1, then subtract 1 more if needed."

Exceptions	Modes	Reasons
#GP(0)	P	Result in nonwritable segment
#GP(0)	P	Illegal memory effective address in CS, DS, ES, FS, or GS segments
#SS(0)	P	Illegal address in SS segment
#PF(fc)	P V	Page fault
INT(13)	R V	Some part of operand is outside of address range 0 to 0FFFFH

User Notes

The SBB instruction is generally used when doing multibyte, multiword, or multidword subtractions, to let the carry bit automatically propagate as needed through the series of differences.

Understanding how the flags work for ADC can be very important, especially since the result of one of the flags after an addition is often used to decide whether to make a jump, or even as a parameter in a subroutine call. SUB (below) works just like SBB but ignores the value in the carry bit. Anyone modifying an existing program should look at all SBBs just following a change to make sure the carry bit is still set as the original programmer had assumed it would be.

Example

```
MOV   AX,1329   ; Loads a 1329 (531 hex) into AX.
MOV   BX,373    ; Loads a 373 (175 hex) into BX.
SUB   AL,BL     ; Subtracts 75 (hex) from 31 (hex) giving 0BC
                ; (hex) with CF set.
```

```
SBB     AH,BH   ; Subtracts 1 from 5 giving 3 (because CF was set).
                ; AX now contains 956 (3BC hex) the difference
                ; between 1329 and 373.
```

SCAS Scan String 8086

Opcode	Format	Clocks Single	Clocks Repeated
AE	SCASB	7	5+8*N
AF	SCASW	7	5+8*N
AF	SCASD	7	5+8*N

The "N" in the "Clocks Repeated" column stands for the number of repetitions actually executed.

Flags

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0			S				S	S	0	S	0	S	1	S

Pseudocode

```
IF operand size is 8 bits THEN
   Subtract ES:[(E)DI] from AL, but don't store result
ELSE IF operand size is 16 bits THEN
   Subtract ES:[(E)DI] from AX, but don't store result
ELSE (* operand size is 32 bits *)
   Subtract ES:[(E)DI] from EAX, but don't store result
END IF
Condition flags based on result of subtraction
IF DF = 0 THEN
   Add size of operands (in bytes) to (E)DI
ELSE
   Subtract size of operands (in bytes) from (E)DI
END IF
```

Operation

Like CMPS, SCAS subtracts one number from another but doesn't store the result. Instead, the only output is that the applicable flags are set or reset ("conditioned") according to the result of the subtraction.

The difference is that SCAS uses the A register and the contents of ES:[DI], subtracting the latter from the former. Finally, DI is adjusted. Normally the Direction Flag is 0, and DI is incremented; but if an STD has been executed, DI is decremented. The increment and decrement are useful in automatically setting up the pointers for a repetition of the same instruction, and the REPE and REPNE prefixes are generally used with SCAS.

Exceptions	Modes	Reasons
#GP(0)	P	Illegal memory effective address in CS, DS, ES, FS, or GS segments
#SS(0)	P	Illegal address in SS segment
#PF(fc)	P V	Page fault
INT(13)	R V	Some part of operand is outside of address range 0 to 0FFFFH

User Notes

SCAS is generally used to compare each element of a string to a preset value (the one stored in the A register). The A register is initialized and SCAS compares it to the values in the string starting at ES:[DI]. By using REPNZ (or REPZ) with SCAS, the comparison can continue until a character not equal to (or equal to) the value in A is found.

Example

```
STR1    DD          1,2,3,4,5
        :           :
        CLD                     ; Ensures direction is forward.
        MOV         AL,3        ; Sets up source of scan, DS:[ESI].
        LES         EDI,STR1    ; Sets up destination of scan, ES:[EDI].
        MOV         ECX,5       ; Sets up repeat count for scan.
        REPE SCASB              ; Executes 3 times, ends with ECX=2,
                                ; ES:[EDI] points at 4.
```

SETcc Set Byte on Condition 80386

Opcode	Format	Set Condition	Clocks
0F 97	SETA r/mb	Above (CF=0 and ZF=0)	4/5
0F 93	SETAE r/mb	Above or equal (CF=0)	4/5
0F 92	SETB r/mb	Below (CF=1)	4/5
0F 96	SETBE r/mb	Below or equal (CF=1 or ZF=1)	4/5
0F 92	SETC r/mb	Carry (CF=1)	4/5
0F 94	SETE r/mb	Equal (ZF=1)	4/5
0F 9F	SETG r/mb	Greater (ZF=0 and SF=OF)	4/5
0F 9D	SETGE r/mb	Greater or equal (SF=OF)	4/5
0F 9C	SETL r/mb	Less (SF<>OF)	4/5
0F 9E	SETLE r/mb	Less or equal (ZF=1 or SF<>OF)	4/5
0F 96	SETNA r/mb	Not above (CF=1 or ZF=1)	4/5
0F 92	SETNAE r/mb	Not above or equal (CF=1)	4/5
0F 93	SETNB r/mb	Not below (CF=0)	4/5
0F 97	SETNBE r/mb	Not below or equal (CF=0 and ZF=0)	4/5
0F 93	SETNC r/mb	Not carry (CF=0)	4/5
0F 95	SETNE r/mb	Not equal (ZF=0)	4/5
0F 9E	SETNG r/mb	Not greater (ZF=1 or SF<>OF)	4/5
0F 9C	SETNGE r/mb	Not greater or equal (SF<>OF)	4/5
0F 9D	SETNL r/mb	Not less (SF=OF)	4/5
0F 9F	SETNLE r/mb	Not less or equal (ZF=0 and SF=OF)	4/5
0F 91	SETNO r/mb	Not overflow (OF=0)	4/5
0F 9B	SETNP r/mb	Not parity (PF=0)	4/5
0F 99	SETNS r/mb	Not sign (SF=0)	4/5
0F 95	SETNZ r/mb	Not zero (ZF=0)	4/5
0F 90	SETO r/mb	Overflow (OF=1)	4/5
0F 9A	SETP r/mb	Parity (PF=1)	4/5
0F 9A	SETPE r/mb	Parity even (PF=1)	4/5
0F 9B	SETPO r/mb	Parity odd (PF=0)	4/5
0F 98	SETS r/mb	Sign (SF=1)	4/5
0F 94	SETZ r/mb	Zero (ZF=1)	4/5

Flags

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0									0		0		1	

Pseudocode

```
IF set condition is met THEN
  Set operand byte to 1
ELSE
  Clear operand byte to 0
END IF
```

Operation

These instructions are used to save a current condition for later use. They test the state of one or more of the flag bits. If the condition (listed in the above table) is met then the destination byte is set to one; otherwise the destination byte is cleared to zero. Note that these instructions are similar to the conditional jumps (Jcc) except that no transfer of control occurs.

Exceptions	Modes	Reasons
#GP(0)	P	Result in nonwritable segment
#GP(0)	P	Illegal memory effective address in CS, DS, ES, FS, or GS segments
#SS(0)	P	Illegal address in SS segment
#PF(fc)	P V	Page fault
INT(13)	R V	Some part of operand is outside of address range 0 to 0FFFFH

User Notes

Many of the sets are meant for use with unsigned numbers, while others are meant for comparisons of signed numbers. The way to tell the difference

is that the use of "above" and "below" indicates unsigned comparisons; "greater" and "less" refer to the use of signed numbers.

A close examination of the instruction table for conditional sets reveals that there are often several mnemonics for the same opcode (and therefore conditional test). The reason for this redundancy is that the same state of the flags can mean different things based on the context of the instruction. For example, a conditional set often appears after a CMP or SUB has been executed. The set then compares the two operands of the previous instruction and a SETE (set equal) might be appropriate. On the other hand, right after a DEC instruction the same opcode with the SETZ (set zero) mnemonic could be used to check if the count has reached zero. We recommend choosing your set mnemonics carefully so that they indicate the meaning of your comparisons.

The SETcc instructions are often used for setting up tables based on the results of several comparisons or computations. This is particularly useful in compiler writing and other applications that require handling nested levels of conditions.

Another good use for these instructions is for the implementation of Boolean (or logical) variables.

Example

```
MOV     ECX,5   ; Sets up operand for compare.
CMP     ECX,7   ; Compares the 5 with the 7.
SETLE   AL      ; AL will be set to 1.
```

STC Set Carry Flag 8086

Opcode	Format	Clocks
F9	STC	2

Flags

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0									0		0		1	1

Pseudocode

Set the Carry Flag to 1.

Operation

STC simply sets the Carry Flag to 1.

Exceptions

None

User Notes

STC has several uses. Perhaps the most prevalent is when the Carry Flag is being used for storing data within a program or even passing parameters between programs, in which case STC and CLC together condition the flag as needed. Another is when doing arithmetic operations; STC followed by

ADC or SBB simulates a carry or borrow by an earlier instruction. Still another is when doing logical operations such as rotates; an STC followed by a rotate is a useful way to change the value in a register.

Example

```
STC   ; Sets CF.
```

STD Set Direction Flag 8086

Opcode	Format	Clocks
FD	STD	2

Flags

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0				1					0		0		1	

Pseudocode

Set the Direction Flag to 1.

Operation

STD simply sets the Direction Flag to 1.

Exceptions

None

User Notes

The Direction Flag controls the direction of string operations. When DF is set the index registers SI and/or DI are *decremented* after each repeat of a string operation. This is the "reverse" direction, and is useful when each

character in the string is stored in successively lower numbered locations in memory and the string is being processed first character first. It's also useful when the string is stored starting in low memory and heading toward high and is being processed last character first.

Example

```
STD   ; Sets DF.
```

STI Set Interrupts Enabled Flag 8086

Opcode	Format	Clocks
FB	STI	3

Flags

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0					1				0		0		1	

Pseudocode

Set the Interrupt Flag to 1.

Operation

STI sets the Interrupt Flag to 1, permitting interrupts after the next instruction if it does not clear the Interrupt Flag. However, there are some complications in 80286 and 80386 Protected Mode. CLI can fail if the current privilege level of the program executing the STI is larger (less privileged) than the I/O Privilege Level bits in the Flags register.

Exceptions	Modes	Reasons
#GP(0)	P	Current privilege is greater than IOPL

User Notes

STI is used to allow interrupts after they've been turned off by a CLI instruction, or at the start of a program to ensure that interrupts are enabled.

Although the privilege restrictions on STI are burdensome, they're necessary; if an operating system is timesharing between several programs it must be able to protect them from interrupts when needed. If your operating system allows access to Protected Mode it's important to understand how it handles privilege levels before attempting to control interrupts from within your program.

Example

```
STI   ; Sets IF.
```

STOS Store String 8086

Opcode	Format	Clocks Single	Clocks Repeated
AA	STOSB	4	5+5*N
AB	STOSW	4	5+5*N
AB	STOSD	4	5+5*N

The "N" in the "Clocks Repeated" column stands for the number of repetitions (from (E)CX).

Flags

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0									0		0		1	

Pseudocode

```
IF operand size is 8 bits THEN
   Store byte from AL into byte at ES:[(E)DI]
ELSE IF operand size is 16 bits THEN
   Store word from AX into word at ES:[(E)DI]
ELSE (* operand size is 32 bits *)
   Store dword from EAX into dword at ES:[(E)DI]
END IF
IF DF = 0 THEN
   Add size of operands (in bytes) to (E)DI
ELSE
   Subtract size of operands (in bytes) from (E)DI
END IF
```

Operation

STOS is just like a MOV from AL or AX to the place in memory pointed to by ES:DI, but it also automatically adjusts DI after the move. If the Direction Flag is 0 DI is incremented; otherwise it's decremented. If a byte was moved the index is adjusted by 1; if a word, by 2; if a dword, by 4.

Exceptions	Modes	Reasons
#GP(0)	P	Result in a nonwritable segment
#GP(0)	P	Illegal memory effective address in CS, DS, ES, FS, or GS segments
#SS(0)	P	Illegal address in SS segment
#PF(fc)	P V	Page fault
INT(13)	R V	Some part of operand is outside of address range 0 to 0FFFFH

User Notes

STOS can't be used with REPE or REPZ since it doesn't condition any flags; using the REP prefix will cause the character in AL to be copied into successive positions in a string. This is a good way to fill a string with a given character.

LODS and STOS can be used together to transfer a string from DS to ES, with any needed conditional tests or changes put in between LODS and STOS. If a string of known length is to be moved unchanged, then MOVS will do the same thing faster.

Example

```
CLD                     ; Ensures direction is forward.
XOR     EAX,EAX         ; Sets EAX to zero.
LES     EDI,BIGARRAY    ; Sets up destination of store, ES:[EDI].
MOV     ECX,1000        ; Sets up repeat count for store.
REP STOSD               ; Fills BIGARRAY with 1000 dword zeros.
```

SUB Subtract 8086

Opcode	Format	Clocks
2C ib	SUB AL,ib	2
2D iw	SUB AX,iw	2
2D id	SUB EAX,id	2
80 [5] ib	SUB r/mb,ib	2/7
81 [5] iw	SUB r/mw,iw	2/7
81 [5] id	SUB r/md,id	2/7
83 [5] ib	SUB r/mw,ib	2/7
83 [5] ib	SUB r/md,ib	2/7
28 [r]	SUB r/mb,rb	2/7
29 [r]	SUB r/mw,rw	2/7
29 [r]	SUB r/md,rd	2/7
2A [r]	SUB rb,r/mb	2/6
2B [r]	SUB rw,r/mw	2/6
2B [r]	SUB rd,r/md	2/6

Flags

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0			S				S	S	0	S	0	S	1	S

Pseudocode

IF (source operand has fewer bits than destination operand) THEN
  sign-extend source operand
END IF
Subtract source operand from destination, place result in destination operand.

Operation

SUB subtracts the second operand from the first operand. The first operand is overwritten by the result, while the second operand is unchanged. A good translation of a SUB instruction into English might be, "Subtract operand number 2 from operand number 1."

Exceptions	Modes	Reasons
#GP(0)	P	Result in a nonwritable segment
#GP(0)	P	Illegal memory effective address in CS, DS, ES, FS, or GS segments
#SS(0)	P	Illegal address in SS segment
#PF(fc)	P V	Page fault
INT(13)	R V	Some part of operand is outside of address range 0 to 0FFFFH

User Notes

The SUB instruction is used when doing single subtraction operations, or one SUB can be done before a series of SBB instructions to do multibyte, multiword, or multidword subtractions (to let the Carry bit automatically propagate as needed through the series of differences).

Understanding how the flags work for SUB can be very important, especially since the result of one of the flags after a subtraction is often used to decide whether to make a jump, or even as a parameter in a subroutine call. SBB (above) works just like SUB but uses the value in the Carry bit.

Example

```
MOV   AX,1329   ; Loads a 1329 (531 hex) into AX.
MOV   BX,373    ; Loads a 373 (175 hex) into BX.
SUB   AX,BX     ; AX now contains 956 (3BC hex) the difference
                ; between 1329 and 373.
```

TEST Logical Compare 8086

Opcode	Format	Clocks
A8 ib	TEST AL,ib	2
A9 iw	TEST AX,iw	2
A9 id	TEST EAX,id	2
F6 [4] ib	TEST r/mb,ib	2/5
F7 [4] iw	TEST r/mw,iw	2/5
F7 [4] id	TEST r/md,id	2/5
84 [r]	TEST r/mb,rb	2/5
85 [r]	TEST r/mw,rw	2/5
85 [r]	TEST r/md,rd	2/5
84 [r]	TEST rb,r/mb	2/5
85 [r]	TEST rw,r/mw	2/5
85 [r]	TEST rd,r/md	2/5

Flags

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0			0				S	S	0		0	S	1	0

Pseudocode

```
REPEAT
  IF a bit in the destination operand is 1 and the corresponding bit in
    the source operand is 1 THEN
    set the bit in the result (kept internally in the 386) to 1
  ELSE
    clear the bit in the result (kept internally in the 386) to 0
  END IF
UNTIL all bits in the operands are checked
Set the flags based on the internal result
```

Operation

TEST carries out a Boolean or "logical" AND on its two operands, but does not store the result anywhere. Instead, the only output is that the applicable flags are set or reset ("conditioned") according to the AND's result. AND is explained in detail above.

TEST can only be used with two operands of the same size (same number of bits); otherwise the comparison would be meaningless for some of the bits in the longer operand.

Exceptions	Modes	Reasons
#GP(0)	P	Illegal memory effective address in CS, DS, ES, FS, or GS segments
#SS(0)	P	Illegal address in SS segment
#PF(fc)	P V	Page fault
INT(13)	R V	Some part of operand is outside of address range 0 to 0FFFFH

User Notes

TEST is often used to test whether a single bit is set. This is done by comparing the number in question with an immediate value that has a single bit set. If the bit is set the Zero Flag will have a 1 in it after the test; otherwise it will have a 0. The new 80386 bit-test commands can do the same comparison more directly but slightly more slowly (3 cycles vs. 2).

Example

```
MOV   AX,9563H     ; Loads a hex number into AX.
TEST  AX,0C6A5H    ; Sets flags: SF=1, ZF=0, PF=1.
```

WAIT Wait for Coprocessor 8086

Opcode	Format	Clocks
9B	WAIT	6*

*This is the minimum value, applicable if the BUSY signal is already inactive.

Flags

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0									0		0		1	

Pseudocode

```
WHILE the BUSY pin is active DO
ENDDO
```

Operation

The WAIT instruction is used to synchronize the 80386 with an 80287 or 80387 numeric coprocessor. These coprocessors are used to speed certain numeric computations, particularly floating point operations. The coprocessors work in parallel with the 80386. When the program on the 80386 needs the results from one of these computations it must wait until the coprocessor is finished. While the coprocessor is working it keeps an active signal on the 80386's BUSY pin; when it finishes it makes that signal inactive. The WAIT instruction simply does nothing until it detects that the BUSY signal is inactive. Then execution proceeds to the next instruction. The 80386 program is thus assured that the desired result is ready for use.

Exceptions	Modes	Reasons
#NM	P R V	The task switched flag in the machine status word is set
#MF	P R V	The ERROR# input pin is asserted (unmasked numeric error detected)

User Notes

Instructions for the numeric coprocessors are not covered in this book, but we have included WAIT because it controls the 80386 and not the coprocessor.

Example

```
WAIT  ; Waits for numeric coprocessor to finish.
```

XCHG Exchange 8086

Opcode	Format	Clocks
90+r	XCHG AX,rw	3
90+r	XCHG rw,AX	3
90+r	XCHG EAX,rd	3
90+r	XCHG rd,EAX	3
86 [r]	XCHG rb,r/mb	3/5
86 [r]	XCHG r/mb,rb	3/5
87 [r]	XCHG rw,r/mw	3/5
87 [r]	XCHG r/mw,rw	3/5
87 [r]	XCHG rd,r/md	3/5
87 [r]	XCHG r/md,rd	3/5

Flags

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0									0		0		1	

Pseudocode

Move the value of the destination operand to an internal register.
Replace the destination operand with the value of the source operand.
Replace the source operand with the value of the destination operand (from the internal register).

Operation

XCHG swaps the contents of its two operands in a single instruction. Without XCHG the same operation would require three MOV instructions (and an extra register for temporary storage) or a PUSH, MOV, POP sequence. In addition, the bus LOCK signal is asserted during this instruction

regardless of whether the LOCK prefix was used or not. This means that the data transfers cannot be interrupted by any other device using the system bus.

Exceptions	Modes	Reasons
#GP(0)	P	At least one of the operands is in a nonwritable segment
#GP(0)	P	Illegal memory effective address in CS, DS, ES, FS, or GS segments
#SS(0)	P	Illegal address in SS segment
#PF(fc)	P V	Page fault
INT(13)	R V	Some part of operand is outside of address range 0 to 0FFFFH

User Notes

Perhaps the most common use of XCHG is to provide a means of interprocess synchronization. Because the XCHG is always locked, each of two independent processes (perhaps on different processors) can use it to access a variable in memory that they share without worrying about contention from the other. For example, suppose two tasks need to have access to the same output buffer. A word can be dedicated as a flag to indicate the buffer's availability. A zero could indicate the buffer was available and a one could indicate that one of the tasks was using the buffer. Before storing any data in the buffer each task must first load one into a register, XCHG that register with the Buffer Flag, and test the register for zero. If the register is zero then the task may use the buffer safely, knowing the Buffer Flag contains a one and the other task will not attempt to use it. If the XCHG were not locked then the situation could arise where both tasks "thought" they had safe access to the buffer. Finally, when the task finishes its use of the buffer it must store a zero back into the Buffer Flag.

Most sort algorithms require that two elements of the array be exchanged when they are found to be out of order. With XCHG the number of instructions required to accomplish this can be reduced from four to three. An additional benefit is that only one register is required instead of two. In fact, at the expense of two clocks, no registers need to be modified. The example below illustrates this technique.

Example

```
XCHG  AX,DATA1  ; AX now contains DATA1 and vice versa.
XCHG  AX,DATA2  ; DATA2 now contains original DATA1, AX has
                ; original DATA2.
XCHG  AX,DATA1  ; AX has its original value, DATA1 contains origi-
                ; nal DATA2.
```

XLAT Translate String 8086

Opcode	Format	Clocks
D7	XLAT	5

Flags

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0									0		0		1	

Pseudocode

Move the byte at DS:[(E)BX + unsigned AL] to AL.

Operation

XLAT is used to change a table index into a table value. It is only useful for byte-valued tables of 256 bytes or less in length. However, this is a very common usage. XLAT expects the table's base address to be in (E)BX and the table offset to be in AL. The instruction then stores the byte at that table offset into AL.

Exceptions	Modes	Reasons
#GP(0)	P	Illegal memory effective address in CS, DS, ES, FS, or GS segments
#SS(0)	P	Illegal address in SS segment
#PF(fc)	P V	Page fault
INT(13)	R V	Some part of operand is outside of address range 0 to 0FFFFH

User Notes

The most commonly cited use for XLAT is the conversion of one character code to another. The programmer builds a 256-byte table containing the values of the target character code, and the XLAT instruction then provides a quick and easy way to do the conversion.

Perhaps a more commonly useful example of XLAT is what we call a "character classification application." This technique can be used by compilers, assemblers, or any other program that has input consisting of character strings that must be interpreted. An XLAT table is built containing character classification codes. For instance: all upper case letters get code 1, all lower case letters get code 2, all digits get code three, etc. An XLAT instruction and a few simple compares and conditional jumps can easily determine the type of character and perform the desired processing. As an alternative to the compares, a jump table could be used.

Example

```
TABLE25  DB      0,25,50,75,100,125,150,175,200,225,250
                                    ; multiples of 25.
         :       :
         CMP     AL,10              ; Our table only handles
                                    ; multiples up to 10.
         JA      TOOBIG             ; So, let's handle it in some other
                                    ; way.
         LDS     BX,TABLE25         ; Sets up table address for XLAT.
         XLAT                       ; Fast multiply by 25 for small
                                    ; unsigned numbers.
```

XOR Exclusive Or 8086

Opcode	Format	Clocks
34 ib	XOR AL,ib	2
35 iw	XOR AX,iw	2
35 id	XOR EAX,id	2
80 [6] ib	XOR r/mb,ib	2/7
81 [6] iw	XOR r/mw,iw	2/7
81 [6] id	XOR r/md,id	2/7
30 [r]	XOR r/mb,rb	2/7
31 [r]	XOR r/mw,rw	2/7
31 [r]	XOR r/md,rd	2/7
32 [r]	XOR rb,r/mb	2/6
33 [r]	XOR rw,r/mw	2/6
33 [r]	XOR rd,r/md	2/6

Flags

	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
0			0				S	S	0		0	S	1	0

Pseudocode

```
REPEAT
  IF a bit in the destination operand is the same as the corresponding
  bit in the source operand THEN
    clear the bit in the destination operand to 0
  ELSE
    set the bit in the destination operand to 1
  END IF
UNTIL all bits in the destination operand are checked
```

Operation

XOR carries out a Boolean or "logical" XOR on its two operands and leaves the result in the leftmost operand. This operation is depicted in Figure 4-4, which gives "truth tables" for all logical operations. A 1 can be regarded as T or True, while a 0 corresponds to F or False. A logical XOR takes two bits and calculates a result using this rule: if one and only one input bit is 1, then the output bit is 1; otherwise the output bit is 0. The instruction XOR simply does this same operation on all the bits in each of two operands; the leftmost bit in one operand is compared to the leftmost bit in the other, then the two bits one position to the right are compared, until all the bit pairs have been compared.

A logical XOR is called an "eXclusive OR" since it "excludes" the case where both statements are true. The closest English equivalent to an XOR is to say, "She's rich or thin, but not both." In this case you have two chances to be incorrect, if the person is neither rich nor thin, or the person is both rich and thin.

XOR can only be used with two operands of the same size (same number of bits); otherwise the comparison would be meaningless for some of the bits in the longer operand.

Exceptions	Modes	Reasons
#GP(0)	P	Result in a nonwritable segment
#GP(0)	P	Illegal memory effective address in CS, DS, ES, FS, or GS segments
#SS(0)	P	Illegal address in SS segment
#PF(fc)	P V	Page fault
INT(13)	R V	Some part of operand is outside of address range 0 to 0FFFFH

User Notes

XOR is used to do bit tests and comparisons; many of these can also be done by using the bit-test instructions that are new on the 80386. A register-to-immediate bit test takes 3 cycles versus 2 for a similar XOR, so the XOR is only more efficient if no other supporting instructions are needed to set up or decipher the comparison results.

Example

```
MOV   AX,5963H    ; Loads a hex number into AX.
XOR   AX,6CA5H    ; AX now contains 35C6 hex, SF=0, ZF=0, PF=1.
```

CHAPTER 5

Protected Mode

MULTITASKING
SEGMENTATION
PAGING
VIRTUAL MEMORY

The applications programmer faces a challenge in writing programs for the 80386. On the one hand, 8086-style programs can run directly on the new processor, so the experienced programmer need not learn anything new to get some use out of the chip. However, there is a bewildering array of capabilities available that are new to the majority of programmers who haven't used 80286 or 80386 Protected Mode. These capabilities will affect the environment in which even the simplest programs will be running.

This chapter describes the workings of such features as tasks, segments, and pages. It will not instantly make the general reader into a systems programmer, but it should promote an understanding of the capabilities and limitations of 80386-based systems and give the Real Mode programmer a running start on writing capable 80386 programs.

MULTITASKING

The word "task" is often tossed about rather casually to describe almost anything a computer might do. Yet it's very important to understand both the idea of a task on any computer and some of the details of the 80386 implementation. It's important because this processor gives the operating systems designer the power to allow the multitasking of several different operations at once, while still looking at each application as running on a simple, single-function computer.

Imagine two different processors sharing a common memory area but running two completely different programs. Call the processors A and B. Now imagine that we wanted to suddenly swap the program which was running on A with the program which was running on B. What would need to be changed?

The state of an 80386-based machine at any point in time can be pretty well described by simply listing the contents of its registers. They point out which instruction is being executed and what data in memory is being accessed, and contain other data the processor is using. Thus we would swap the contents of the registers in each chip. Processor A would get the segment registers, the instruction pointer, and the general register values from processor B; processor B would get the same from processor A. Include the Flags register and we've pretty well got it; the two processors have now swapped jobs.

This interchange is much like a "task switch" on the 80386. Since there's only one processor, the first program controls the entire system for a while; then the program is told to "abandon ship." It saves the values of all its registers in an area of memory called a Task State Segment (TSS). This contains copies of all the registers, as well as other information described below. The registers are reloaded from the TSS of the program which is taking over. When execution proceeds a new program is in control, and the old program is waiting to be called again when needed.

This viewpoint helps us understand exactly what multitasking is on the 80386. Multitasking doesn't necessarily mean multiple users; indeed, a single program like a word processor can start up a task to handle a chore like print buffering and close down the task when the print is done, with the "child" task invisible to the user. A single-user multitasking system can have one operating system that controls several programs running at once as separate tasks; the user can assign priorities to them and switch among them at will.

A task can also be an instance of running a given program and its

operating system, controlled by a "hypervisor" or control program that handles task switches, priority levels, etc. Virtual 8086 Mode is implemented this way (see Chapter 6). The hypervisor can have 80386-based applications programs run directly under it, and can host multiple 8086-style applications and operating systems with the programs never "knowing" they're being swapped in and out.

Multiuser systems can be implemented by letting each user run as a separate task (in any of the forms listed above). The processor switches among the different tasks so fast that each user thinks he or she has sole control of the computer. It's also possible to let each of the users have multitasking at his or her terminal, but this can lead to processor overload. The number of tasks that can run at once without obvious strain depends on the type of hard disk, memory, and other peripherals with which the processor must interact, as well as the timing requirements of each task.

80386 Support for Multitasking

Several data structures, stored in a standard form for quick handling, are recognized by the 80386 and a special register, the Task Register (TR), points to the current task. Using these elements the 80386 can switch from one task to another in 268 clocks (about 17 microseconds on a 16 MHz system). If no actual work was done between switches, this would allow nearly 60,000 task switches per second (although operating system overhead adds to this figure), which means that many task switches can be made per second while still allowing plenty of time for actually executing programs.

The software structures are listed below.

1. There is a special kind of segment called a Task State Segment (TSS), which is always at least 26 dwords long. The TSS has room for copies of all the 80386's registers, including EFlags and EIP, the instruction pointer. It also contains a 16-bit pointer to the previous TSS, which is kept in case of a return to that TSS. This is useful when a subordinate task (like an exception handler) has been called by another task and must return control to the caller when finished.
2. There is also a special Task State Segment descriptor. All segments have descriptors that give needed information about the task, including its location, size, and privilege level. For a TSS descriptor the LIMIT or length must be at least 103 bytes (104 if

an I/O permission map is used; a pointer to it is stored in the last word of the TSS). Longer LIMITs allow bigger TSS's to be used, with the user defining the values in the additional bytes. Also worth noting is that TSS's aren't reentrant; if a task is busy (as indicated by the Busy bit in the TSS descriptor), it can't be started again.

3. Finally, there is a Task Gate Descriptor. The task gate is a short data structure that allows access to a TSS (other gates allow access to other data structures). Figure 5-1 shows the format of the Task Gate Descriptor. The DPL is the privilege level of the task gate. The procedure that accesses the task gate must operate at a privilege level as low or lower than that of the task gate. If the access is allowed, the selector allows the task state segment to be accessed without further protection checking. This means that a procedure that couldn't normally access a given TSS can reach it via a task gate. Unlike TSS's, the Task Gate Descriptor can be in a local data table and visible to some procedures (including interrupt and exception handlers) but not others. This allows flexible access to a task, which can have several task gates, each accessible to different procedures and each with a different privilege level, but all sharing the same Selector field.

Besides the software structures, there is also a register dedicated to multitasking, the TR register. The TR register is like an iceberg—most of it is invisible. The upper 16 bits are the visible part; they contain a selector that points to the current TSS descriptor. The hidden part holds a 16-bit TSS base and a 16-bit TSS limit. The TSS acts as an on-processor cache so these fields can be accessed quickly when the current TSS is referenced by a program.

A task switch occurs when a JMP or CALL refers to a TSS descriptor or a task gate, when an interrupt or exception points to a task gate, or when the current task executes an IRET and the NT (Nested Task) flag is

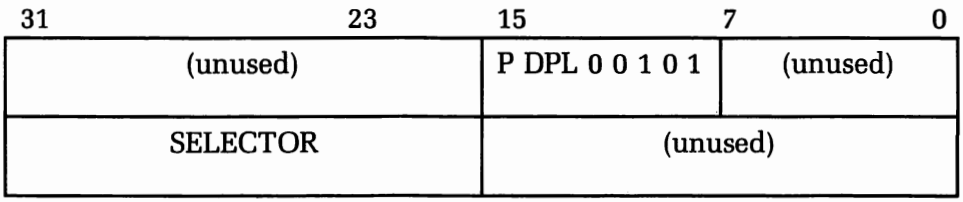


Figure 5-1. Task Gate Descriptor

set. In this last case the pointer to the previous Task State Segment stored in the current TSS is used to quickly bring back the calling task.

Several checks are made for privilege level and for the presence in memory of the current TSS (if it's paged out an exception occurs). If the checks are passed the current task's registers are saved in the current TSS. The TR is then loaded with the selector of the new TSS, and the registers are loaded with the information in this new TSS. Finally, the TS (Task Switched) bit in the Machine Status Word is set to alert coprocessors that a change has occurred.

SEGMENTATION

Segments are logically separate pieces of memory, each with its own size, level of protection, and other characteristics, but the segments can overlap or even be exactly the same area of memory (this is called aliasing). The 80386 accesses six segments at a time, each pointed to by a 16-bit register. The registers CD, SS, and DS specify the starting addresses for Code, Stack, and/or Data Segments. The registers ES, FS, and GS specify starting addresses for three Extra segments. Each of these segments has its own segment descriptor that contains protection and size information.

On the 8086/88 segments are limited to 64 Kb each. This means that most programs that run on it need separate segments for code, data, and stack, and medium and large programs have to access several segments for code, data, and stack during execution. Large data structures are broken up into parts by segment boundaries.

The 80386 allows segments to be much larger, 4 Gb each. This allows a great deal of flexibility in managing memory. However, the structure of memory is largely determined by the operating system. An 8086 operating system (or any 80386 Real Mode or Virtual 8086 Mode program) running on the 80386 will impose exactly the same limits as found on the earlier chip. An 80386 OS may impose restrictions for purposes of compatibility with existing code or to simplify the work it has to do. The processor's full capabilities are discussed; how many of these an application has access to depends on the operating system.

Before we plunge into the details of segmentation, it's worth noting that the 80386 can support a totally unsegmented and unprotected memory model. By setting all the segment registers to 0 and giving each segment a limit of 4 Gb and a protection level of 3 (no protection), the

instruction pointer, stack base, index pointer, and data pointers can all refer to the same 4 Gb area, allowing them to be intermixed at will. This is not necessarily good for all applications, but it allows UNIX-type operating systems (which generally use a flat memory model) to be ported directly to 80386-based systems.

Linear Address Creation

Most applications programs use a variety of addressing modes to access memory. These modes affect the final value of the offset used in address calculations; this offset is called the Effective Address of an operand in memory. The effective address is combined with the appropriate segment base register to create a linear address. If paging isn't on, the linear address is used to directly access memory.

In Real Mode the appropriate segment register is simply multiplied by 16 (shifted left by 4 positions) and then added to the lower 16 bits of the effective address to create a 20-bit address, just like on the 8086 or the 80286 in its Real Mode. The same technique is used for tasks running in Virtual 8086 mode. In Protected Mode linear addresses can be up to 32 bits long and access up to 4 Gb of memory.

The segment registers are actually 64 bits long, but only the upper 16 bits are visible to programs. The 64-bit entry contains all the needed protection and other information about the segment, and is called a segment descriptor. It includes the actual base of the segment, which is a quantity up to 32 bits wide that addresses 4 Gb of memory (the linear address space). The linear address is calculated by adding the segment base to the effective address, with no shifting.

The registers on the processor itself contain segment descriptors for the segments that are in use by the currently executing task. However, an 80386 program may have many segments, and the processor itself may be running several tasks at once. The segment descriptors stored in the registers are actually only easy-to-access copies of the segment descriptors stored in memory.

Segment Descriptors and Tables

The format of a segment descriptor is shown in Figure 5-2. The fields within the descriptor are interpreted slightly differently for different types of descriptors; the one described here is for code and data segments.

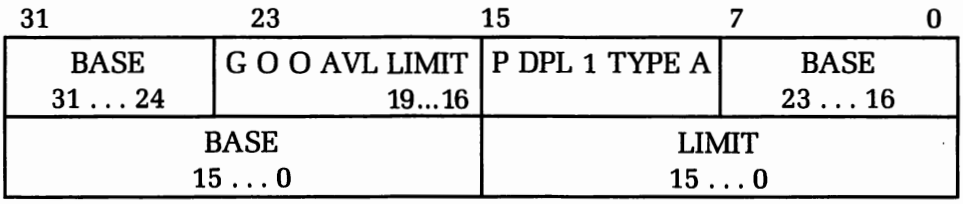


Figure 5-2. Segment Descriptor

Other types of segments are similar. Individual fields are listed below.

1. BASE. The base of the segment is a 32-bit-long field stored in three separate pieces within the segment descriptor. The pieces are concatenated to form a single 32–bit quantity.
2. LIMIT. The limit is a 20–bit–long quantity stored in two separate pieces within the segment descriptor. The two pieces are concatenated to form a single 20–bit quantity. Such a quantity can normally only be used to address 1 Mb of memory, which would limit segments to 1 Mb in length if not for the G bit.
3. GRANULARITY (G). When this bit is zero the limit is indeed a maximum of 1 Mb, but if the G bit is one the limit is multiplied by 4 K (the size of a page). The limit now says not that the segment can only be a given number of bytes long, but that it can only be a given number of *pages* long. This means that the limit can only be checked to within the nearest page of the actual end of the segment.
4. AVL. This bit determines whether the descriptor is available for use by the operating system.
5. PRESENT (P). This bit indicates whether the segment is actually in memory. The bit is 1 and the segment is unavailable if segment-based virtual memory is in use and the segment is swapped out or in a space not mapped by the paging unit.
6. DESCRIPTOR PRIVILEGE LEVEL (DPL). These two bits indicate the privilege level needed to access the descriptor, from 0 (highest) to 3.
7. TYPE. This 4-bit field is used differently by different types of descriptors. It can specify, for instance, that a segment is executable, readable, and/or writeable. For operating system segments this field can specify a type of descriptor like LDT, TSS, or Gate, the latter two of which have subtypes of their own.
8. ACCESSED (A). This bit is set if the descriptor is loaded into a segment register.

Descriptors are created and maintained by the operating system and other systems software like compilers and linkers; all of them must work together to keep the descriptors updated correctly.

All descriptors are kept in memory-based tables. These tables can be up to 8,192 (2^{13}) descriptors long. There is a single Global Descriptor Table (GDT) containing descriptors of segments that can be accessed by any task with a high enough privilege level to get at the segment, either directly or via a gate. Usually operating system code is globally available. Each task also has a Local Descriptor Table (LDT), which describes the segments accessible only by that task; this would usually include a task's code and data. However, two tasks can share part or all of an LDT. For example, a word processing program that creates a separate task to serve as a print controller can let the new task access the code and data needed for its job.

One of the interesting special topics that arises in connection with segment descriptors is aliasing. When two or more descriptors name parts of the linear address space that partially or completely overlap, they are aliases of one another. A descriptor for a code segment, for instance, might specify that the code is readable but not writeable. An alias segment descriptor for the code segment might specify that the same area was writeable, and the alias could then be used (either deliberately or accidentally) to modify or overwrite the code, possibly causing problems. Also, the operating system must keep track of the aliases; if the operating system wants to delete a segment, for instance, it must delete or modify all the descriptors that allow access to that part of memory.

PAGING

The paging capability of the 80386 is one of the most interesting and novel aspects of the processor, yet it's completely a tool of the operating system. Whereas the applications programmer at least can see part of the segment registers that control segment addressing, paging is invisible to the programmer. It's also new with the 80386, so there's no base of understanding for the 8086 or 80286 programmer to build on. Yet paging is very much worth understanding. It will probably be the main tool used to access large amounts of actual memory, and it is vital to the operation of Virtual Mode, which will be the basis of both the multitasking systems

and the giant applications programs of the future. It's easy to learn enough about paging to discuss and work around its implications intelligently without actually writing operating system code. The information in this section and below, plus information about your particular operating system, may be enough for many programmers.

An 80386 page is a 4 Kb-sized piece of memory (some other computers use other page sizes). A page may start at any point in memory, but for convenience's sake pages are usually placed at addresses 4 Kb apart (page frames). The addresses of page frames are 0, 4 Kb, 8 Kb, 16 Kb, etc. Any data item that starts at one of these addresses is said to be "aligned on a page boundary." Addressing a page frame in the 4 Gb linear address space is made easier because only a 20-bit address is needed; the last 12 of the total 32 bits in the address are all zeros. In virtual memory pages are swapped between disk and memory as needed. The 4 Kb sections on disk that hold pages are called page slots.

An 80386 system doesn't have to use paging at all, and an 80386 in Real Mode can't use it. A bit in the Machine Status Word, inaccessible to Real Mode and applications programs, allows paging to be on or off. However, paging is an almost irresistible tool for the systems programmer, since (for one thing) it provides an easy way to assign almost any actual address to replace the linear address calculated by a program. Thus, most 80386 operating systems will do at least some paging.

Memory can also be only paged, not segmented, by working around segmentation with the trick of putting everything in one large segment. Paging can then be used for protection and privilege control. Segments have their advantages too, especially for protection of all code or pieces of data. A common memory model will probably use segments for protection based on segments and paging for memory remapping and virtual memory. Paging will be invisible to the applications programmer, but it will affect the memory locations that a program resides in and accesses.

Physical Address Creation

The Physical Address is the address that comes out of the paging unit. When paging is not on (or even in many cases when it is on) the Linear Address that goes into the paging unit is the same as the physical address that comes out. However, while the process of translation is complicated and not always used, it's worth understanding as it can affect every aspect of the system.

A physical address has three parts: a 10-bit DIR entry, a 10-bit PAGE entry, and a 12-bit OFFSET (see Figure 5-3). These three fields are used to generate a physical address in the following way:

1. The 32-bit CR3 register holds the address of the current Page Directory. The low 12 bits of CR3 are always 0 because the Page Directory always starts on a page boundary. The 10-bit DIR value from the Linear Address points to the needed Page Directory Entry.
2. The high 20 bits of the Page Directory Entry point to a Page Table; 12 low-order zeros are tacked on to make a full physical address for the page-aligned table. The 10-bit PAGE value from the middle of the Linear Address points to the needed Page Table Entry.
3. The high 20 bits of the Page Table Entry point to a page frame; 12 low-order zeros are tacked on to make a full linear address for the page frame. The 12-bit OFFSET value from the low-order bits of the Linear Address points to the needed location in memory.

Each page directory has up to 1024 entries, allowing that many page tables. Each page table can point to as many as 1024 pages, each 4 Kb long. This means that a single directory can address 1024 * 1024 * 4096 bytes, or 4 Gb (the entire physical address space of the 80386).

The Page Directory Entries (PDEs) and Page Table Entries (PTEs) are almost identical. The page table can be thought of as a "page descriptor table," and the Page Table Entry as a "page descriptor," if this helps keep them straight. Figure 5-3 shows an entry like that found in either table. A bit-by-bit breakdown is listed below.

1. PAGING ADDRESS (Bits 31 . . . 12). For a PDE this address points to a Page Table; for a PTE it points to a page. The lower 12 bits of the address are always 0's, since both Page Tables and pages are aligned on 4096-byte boundaries.

Bit 31	12	11 . . . 9	8	7	6	5	4	3	2	1	0
Paging Address		OS Res	0	0	D	A	0	0	U/S	R/W	P

Figure 5-3. Page Directory Entry/Page Table Entry

2. OS RESERVED (Bits 11 . . 9). These bits are available for operating system use. A typical function would be keeping statistics for virtual memory and page swapping, like the number of times a page had been accessed in a given period.
3. D (Bit 6). This is the Dirty bit for page table entries, which is set automatically when the page is written to.
4. A (Bit 5). This is the Accessed bit for page directory and page table entries, which is set automatically when the page is read or written to.
5. U/S (Bit 2). This is the User/Supervisor bit. If it's set then programs with protection level 3 (lowest level) are allowed access.
6. R/W (Bit 1). If U/S is 1 (user access is allowed) then a zero in this bit means only read accesses are allowed; a 1 means write accesses are also allowed.

   These bits are meaningful for PDEs and PTEs. An individual page is protected according to the most restrictive pair of bits from its PDE or PTE. If the U/S bit is 0 the page may be neither read or written to. If U/S is 1 and R/W is 0 the page may be read only; if both bits are 1 (in both entries, since the most restrictive applies), the page can be read or written to.
7. P (Bit 0). This is the Present bit that indicates whether the PDE or PTE points to a page that is presently in memory. If this bit is 1 the bit fields are as defined above. If it's 0 the needed page is out on disk and the remaining 31 bits can be the location of the page slot out on disk that has the needed page.

While the Page Table Entry contains a lot of useful information about a page, it's very hard to get to; two memory accesses are required just to get the address of each page. The Translation Lookaside Buffer (TLB) in the Paging Unit contains the 32 most recently used PTEs. The TLB often has the needed PTE, eliminating the need to actually look at the memory-based tables over 95% of the time. The workings of the TLB are explained in detail in Chapter 7.

VIRTUAL MEMORY

Virtual memory lets a programmer work without having to worry about how big memory actually is. In virtual memory systems the hard

disk in effect becomes the main memory, and RAM ("real memory") is a holding area for the code and data currently in use by the processor. See Figure 5-4 for an illustration of the relationship between these types of memory.

The number of bits in the largest address that a processor can construct determines the size of its virtual address space. On advanced processors like the 80386 there are two different address sizes: a large address determines the virtual address space or amount of memory the programmer can use. A smaller address determines the real address space, the amount of real memory that the processor can support. On the 80386 the address information in Paging Tables allows a virtual address space of 64 Tb (over 64 trillion bytes). This is the same as 64,000 1 gigabyte CD ROMs. (If one copy of every book ever printed were stored on CD ROM, fewer than 64,000 disks would be needed.)

The real address space can be accessed directly with a 32-bit address, with or without paging. This address size allows up to 4 Gb of real memory to be accessed (over 4,000 times more than a 1 Mb system has). However, these limits are beyond the reach of current hardware. In practice real addresses are used to access RAM and virtual addresses are

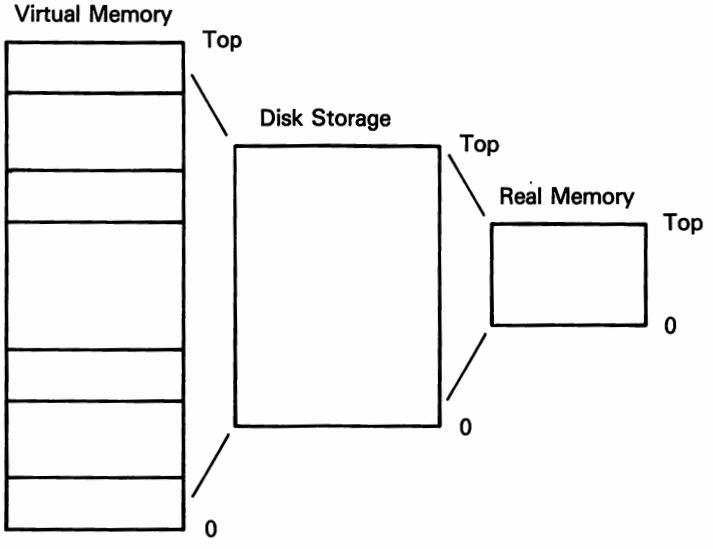


Figure 5-4. Relation of Virtual Memory to Disk Storage to Real Memory

used to access a virtual space no bigger than a hard disk. The virtual addresses won't use even the full 4 Gb of space allowed by a 32-bit address. The larger virtual address spaces will be used for networked systems and communications at first.

A simple form of virtual memory is already in use by large programs. In this technique a core part of a program stays in memory at all times. The rest of the program is divided up into pieces of about 64 Kb each. As the user accesses different parts of the program, these 64-Kb chunks are brought in from disk as needed. Each chunk is called an "overlay." However, dividing a program into overlays and managing the swapping of them in and out of memory are big chores for the programmer.

In virtual memory every program, data file, etc. is seen as a series of 4 Kb "pages." The real address space (RAM) is divided into a 4 Kb-wide "page frames," and the virtual address space (hard disk) is divided into 4 Kb-wide "page slots." When needed by the user's access of different parts of a program, different pages are brought in from their page slots into page frames in real memory, where they can be accessed. When memory is full and another page is needed, an existing page frame is overwritten (if it hasn't been changed since coming into memory) or copied out to its page slot on disk before being overwritten (if it has been changed). All this takes place out of sight of the programmer; the programmer simply accesses code and data at different virtual addresses and the processor and operating system cooperate to bring the needed page into memory when it's actually needed.

How the 80386 Supports Virtual Memory

The following features are important in the 80386 support of virtual memory:

1. The ability to access large address spaces.
2. The ability to translate virtual addresses into real addresses for accessing real (physical or RAM) memory.
3. The ability to cause an exception (instead of simply halting) if a needed piece of code or data is not currently in memory (a "page fault").
4. Instructions that can be restarted if a page fault prevents the instruction from completing.

This is a lot for a processor to do, but much of the work is still left to the operating system. When a page fault occurs the operating system must get the needed page from disk and bring it into memory. If memory is full it's the operating system that must write an existing page to disk, if needed, and then overwrite it with the needed page.

There are several important issues surrounding virtual memory which will directly affect the applications programmer. One is how much real memory is needed to support a given amount of virtual memory. For instance, will a system with 1 Mb of available RAM (memory above the amount needed for video support and the operating system) run three programs at once, each of which normally uses 640 Kb of RAM? A lot of research into this kind of question has been done on mainframes, but the results are not directly applicable to the different operating systems and kinds of demands made on microcomputer-based systems.

A preliminary estimate is that a computer can safely support about twice as much virtual memory as the amount of real memory available. Thus our three 640 Kb programs would probably run comfortably on a system with 1 Mb of memory free to support them. The important issue is the size of the "working set" of each program. If each program accesses a large number of pages repeatedly (has a large working set) they may each make the operating system go to disk frequently for more pages, causing "page thrashing." Repeatedly recalculating large spreadsheets while doing a large compile or sort at the same time might be one way to cause page thrashing.

CHAPTER 6

V8086 Mode and More

VIRTUAL 8086 MODE DEFINED
VIRTUAL MACHINES
MORE ON VIRTUAL MODES
PROCESSORS AND MODES COMPARED
OPERATING SYSTEM CONSIDERATIONS

The preceding chapters described the 80386's Real Mode and Protected Mode. This chapter builds on the preceding ones to describe Virtual 8086 (V8086) Mode, which allows a task (for instance, an applications program) to run as if it was in Real Mode, while the operating system has access to all the capabilities of Protected Mode. This chapter also compares the 80386 in each of its modes with the 8086, describing the differences between the processors and how they affect software. Finally, we discuss how these factors affect what is possible in an operating system. This section should help the reader make an informed choice when selecting an OS and get the most out of the one chosen.

This discussion also explains how the 80386 can run plain-vanilla PC programs and full-featured 32-bit tasks concurrently; why the 80286 didn't run so many of the earlier programs, even in Real Mode; and why the first (and least expensive) operating systems to become available are

likely to tap only part of the processor's power. It ties together what's gone before, ending with a look at future possibilities.

VIRTUAL 8086 MODE DEFINED

There is over $5 billion worth of software already in use for 8086/88 and 80286-based computers. This existing software (programs and operating systems) can be run unchanged in 80386 Real Mode. In Real Mode only one operating system and one program are running at a time, allowing for compatability with PC and AT-style programs but using little of the power of the 80386 besides its sheer speed. To the operating system and the programmer the 80386 seems much like a fast 8086 with additional instructions and registers and the same 1 Mb limit on memory that the 8086 has.

The alternative to Real Mode is Protected Mode, which offers access to all the capabilities of an 80386-based computer: multitasking, paging, virtual memory, etc. The computer as a whole can only be in Real Mode or in Protected Mode. However, while the computer is in Protected Mode, an individual task (which can be a single program with or without a copy of its operating system) can run in Virtual 8086 Mode, which gives an 8086-style environment (the same as an 80386 Real Mode environment) to a task. Meanwhile, the 80386's operating system and other programs have access to everything the computer can do in Protected Mode.

To talk about V8086 mode and its implications we have to review and extend some vocabulary. A task is any independently running program; to be precise, a task is a program that has an 80386 Task State Segment or an equivalent. The TSS stores register contents and other information, allowing the 80386's operating system to halt and restart the task at any time.

Luckily for the user's existing software library, but unfortunately for the cause of clarity, an 8086 operating system like DOS 2.x or 3.x can run as a task. While the user is running this task he or she can start up, run, and exit nearly any program that runs on either an 8086 or 80286, like Turbo Pascal® or Lotus 1-2-3®. Thus we have an operating system like DOS 2.x running as a task under the 80386's own operating system, which handles paging, virtual memory, and priority among other tasks that are running at the same time. To avoid confusion we will at some points call the 80386's own operating system a hypervisor (to differentiate it from another operating system running under it as a task). The hypervisor

allows the user to set priorities and switch among several different tasks at once.

VIRTUAL MACHINES

In virtual memory a program is allowed to be very large and/or have large data structures, and act as if the program and its data are all in memory at the same time. The operating system is responsible for allocating memory space to the program and for bringing code and data in as needed in cases where everything can't really fit in memory at once.

A "virtual machine" extends the concept of virtual memory to the entire computer, including I/O devices like the keyboard and monitor, the register set, etc. The program runs just as if it had an entire machine (in this case an 8086-based computer) to itself, thus the name Virtual 8086 Mode. When the program makes calls on system resources (like a write to the monitor), the 80386's hypervisor may let the call perform its action or it may trap the call and perform the write itself, or not do the write at all, before returning control to the running program.

In particular, let's suppose an 8086 assembler program wants to write "Hello" on the screen. It dutifully calls DOS 3.1 with an INT and DOS executes an I/O write to the screen. However, DOS 3.1 is running as a V8086 task, which has privilege level 3. The hypervisor has put a privilege level of 0 on the I/O area, so the DOS write causes a protection exception. The interrupt vector calls the hypervisor by causing a task switch to the hypervisor's TSS, which must decide what the V8086 task is trying to do, emulate it (by doing the actual screen write, in some other way, or by ignoring the attempt), and then return control to the V8086 DOS task. As more of this type of intercepting and decision-making goes on, the 8086 assembler program is increasingly slowed when it or DOS tries to perform I/O or otherwise use system resources. The same process applies to V8086 copies of CP/M 86 or 8086-type UNIX lookalikes.

We said above that the I/O area had privilege level 0, but addresses are being constructed 8086-style (1 Mb limit), so segment-based protection (which uses full 80386 addresses) won't work. The 8086-style address is sent to the paging unit where it can be mapped to any page, which may be in memory or out on disk (Virtual Memory). Thus the operating system can enforce page-based protection whenever a V8086 task is running, and can support virtual memory for several V8086 tasks at once by using the paging unit and page tables.

This same type of protection checking and simulation can take place any time the 80386 is running multiple tasks, not just when one of the tasks is 8086-style. How does V8086 Mode "fool" the program which is running into thinking that it's running on a 16-bit 8086 instead of a 32-bit 80386?

Bit 17 of the EFlags register is called "VM" (Virtual Mode). When a Virtual 8086 task starts running, the VM bit is set. This bit tells the 80386 to generate addresses and handle segments as it would in Real Mode, so addresses are generated by shifting the appropriate segment register left four positions, adding the appropriate 16-bit offset, then using the result as a memory address. The resulting 20-bit number allows only 1 Mb of memory to be addressed. The VM bit also makes only the 16-bit part of a 32-bit register (like EFlags) accessible to the current task. Note that the VM flag in bit 17 is not in the lower 16 bits of the register and not accessible to a task running while VM is on.

Also, while the V8086 bit is set certain privileged opcodes cause exceptions. Only the instructions and addressing modes found on the 8086 are allowed to execute.

This also helps explain how the 80386's Real Mode functions. When the Protection Enable bit in the Machine Status Word is clear, the same 16-bit addressing and opcode restrictions apply as in Virtual 8086 Mode. In both cases the current task operates as if running on an 8086. The difference is that the Machine Status Word controls the whole machine, putting it in Real Mode, while EFlags controls only the current task; the current task acts like it's in Real Mode but a task switch back to the hypervisor turns the V8086 flag off, allowing the privileged opcodes and addressing modes to be used again.

MORE ON VIRTUAL MODES

The discussion below gives some background on the history and current status of virtual modes. This will be interesting to many but does not apply directly to V8086 mode; this section can be skipped.

The idea of virtual machines was pioneered by IBM on its largest machines, the 370 series. The VM operating system currently used on all these machines was developed inside IBM as a research project in the late 1960's. There are two parts to VM: CP and CMS. CP (Control Program) serves the same function as an 80386 hypervisor, allocating

resources to various tasks, trapping calls to I/O and other resources, and then emulating or ignoring them. Individual programs run under a single-user, single-tasking operating system called CMS, Cambridge (now Conversational) Monitor System. CMS is like the copies of DOS we've been talking about; it seems to have control of the computer, but actually causes CP to take over when it tries to access most parts of the computer itself.

An interesting aspect of CP is that it can run another copy of CP as a task under itself. Neither copy "knows" whether it actually has control of the computer; it simply acts as if it does, making use of the full resources of the system, and if there is another CP "above" it controlling the computer that CP will trap the calls of the "lower" one.

This is not true of the 80386. It can run V8086 tasks in which the 8086 environment is duplicated, and all 8086 programs and operating systems will run under it. An 80286 Real Mode (8086-type) application will also run under it, but 80286 and 80386 operating systems cannot be run as tasks under an 80386 hypervisor. It is thus said that the 80386 can "virtualize" the 8086, but not the 80286 or itself. This is because of problems with certain instructions, including PUSHF and POPF, which push and pop all flags.

This means that the 80386 can run 8086-style operating systems, but can't run 80286 versions of DOS or XENIX, for instance, as guest tasks. These programs can run as the sole operating system for the 80386 if set up properly, but the processor will then emulate an 80286 completely, and that machine doesn't have the capabilities (paging, multitasking, etc.) of the 80386. Thus these operating systems are only stopgaps until true 80386-based operating systems can be developed.

As long as the 80386 is running in Real Mode, it is only acting as a fast 8086. When it's running an 80286 operating system, it's acting as a fast 80286. But when it's running a V8086 task the operating system and tasks running concurrently with the V8086 task have access to the full power of the 80386. The V8086 task doesn't need all of these powers, it simply "wants to pretend" it's got full control of an 8086. However, the V8086 task may run slower on the 80386 than it did before. If multitasking is occurring the V8086 task will be slowed by not running all of the time. Even if the V8086 task is all that's running, many calls to DOS may result in task switches to the hypervisor, which then must decide what's happened, emulate the DOS request or not, and then return control to DOS. Many lines of code may execute between the original call to DOS and a return to the guest application, slowing the perceived speed of the

application. The exact effects depend on which DOS calls are trapped by the hypervisor (causing extra processing) and which are allowed to proceed.

When a hypervisor is supporting multiple V8086 tasks it must use paging. This is because each task will produce addresses in the same range: 0 to 1 Mb. Paging can be used to remap most of these addresses to physical addresses beyond 1 Mb. It can also be used to help the various tasks share the operating system and/or hypervisor code. Finally, paging protects the I/O areas from direct access by the V8086 task.

As an example of the power of paging, operating systems that allow some degree of multitasking on the 8086 (like Concurrent PC-DOS from Digital Research) have problems with programs that are "ill-behaved." The best example of poor behavior is updating the video display. A well-behaved program will call the operating system to do a screen write, but this is noticeably slow. Ill-behaved programs write directly to the memory locations that the screen is mapped to, producing fast results. The problem comes when the user wants to run two programs at once, one in "foreground" (the program the user is interacting with) and the other in "background" (running invisible to the user). The foreground program has control of the video monitor; the operating system intercepts any attempt by the background program to write to the monitor. But if the background program bypasses the operating system and writes directly to video memory, the user suddenly sees results from the background program appearing on his foreground screen, which is running some other program!

With the 80386 a multitasking operating system simply uses the paging tables to remap the background task's accesses to video memory. The accesses now update some other area of memory; the actual video memory is controlled by the operating system. This allows the operating system to completely control what appears on the screen, while the ill-behaved program runs very efficiently when allowed to have control of the monitor.

PROCESSORS AND MODES COMPARED

There are two basic types of programs (including operating systems) which run on processors from the 8086 family. The first are Real Mode-style programs, including 8086/88 programs and operating systems,

80286 Real Mode programs, 80386 Real Mode programs, and V8086 tasks. The second are Protected Mode programs, which include 80286 and 80386 operating systems and applications programs that run in Protected Mode on one of the two processors (rare as of this writing). We won't concern ourselves here with compatability between the 80286 Protected Mode programs or operating systems, and we'll also leave aside the 80386's capability to run 80286 programs directly or concurrently with its own programs. Those bringing such programs to the 80386 environment may be directly in contact with Intel for help.

Here we will concentrate on the various types of Real Mode programs and the similarities and differences among them. This information will help in understanding the different processors and in bringing applications from the 8086 and 80286 Real Mode to the 80386, to be run as either V8086 tasks (very simple) or Protected Mode tasks (possibly simple depending on the 80386 OS).

8086 Programs vs. 80386 Real Mode and Virtual Mode Programs

One of the great advantages of the 80386 is that in Real Mode it can run 8086 programs and operating systems almost unchanged. It can also run these same programs and OSs as V8086 tasks, so they can run unchanged while a hypervisor allows paging, multitasking, and other capabilities to be used. In order to take advantage of these capabilities, however, the programmer needs to know exactly what differences there are between programs for the 8086 and programs for the 80386.

Because Real Mode and V8086 Mode are so similar, we will treat them as the same when we compare them to the 8086. The next section explains the few differences between them and how a V8086 task so closely simulates a Real Mode program in full control of the computer.

There are two types of differences between V8086/80386 Real Mode programs and 8086 programs. The first is improvements: new registers and instructions, for example. The second is surprises: differences that can cause an 8086 program to act differently when run on an 80386. Some of the improvements can also become surprises; for instance, increased speed can cause some programs to work differently. Being optimists, we'll cover the improvements first.

The first improvement is speed. An 8086 Real Mode program executes six to ten times faster when run on the 80386. About half of the speed increase is due simply to the processor running at 16 MHz instead

of, for example, about 5 MHz. Programs that often use certain much-speeded instructions (like IMULs) can expect even greater performance improvements. This one change makes a noticeable difference to the user for most programs. If the 80386 is accessing a reasonably fast hard disk, this pleasant effect is increased. The speed increase is free to the programmer; no changes need to be made in the program. Also, the program can be taken as is and run on an 8086-based system at any time. Also free is the ability of an 8086 program to be multitasked with other programs running concurrently on the 80386.

Other performance improvements in Real Mode require changes to the program itself. These changes will make the program smaller, faster, and/or more capable; they will also make it incompatible with the 8086. Before enhancing an application (the desired changes may not require a complete rewrite), the programmer must consider the vast number of computers that can no longer run a program when it uses the new features available in 80386 Real Mode. One good idea is to use comments to point out the new 80386-only code, and to keep any removed code (either as comments within the program or as part of a program library with the old version). That warning given, the new features are explored below.

New Architecture

The 80386 is architecturally different from the 8086 in that it has two new segment registers: FS and GS. These new registers serve the same function as ES, pointing to the base of Extra data segments. However, ES is used as the implicit destination of certain string operations. The FS and GS registers are not implied by any operation, and thus are free for use in any way the programmer wishes. 8086 programs can be optimized just by keeping much-used memory variables in FS and GS for faster access. Segment overrides also allow these two registers to be used as advertised, as base registers for addressing new data segments.

Real Mode programs can also access all 32 bits of the 80386's general registers simply by using the name of the larger register in an instruction: MOV EAX, EBX. The assembler will generate an operand size prefix that will cause the processor to access the entire 32-bit register (accessing only the lower 16 bits is the default in Real Mode). (This requires some care, since the upper 16 bits can't be assumed to be zero if they haven't been initialized.) Many programs will find this addition of 32-bit operands one of the most significant improvements on the new

processor, since it allows much larger numbers and other data items to be used without slowing processing.

Other new registers are also available. The debug registers can be used from directly within a program or from an external debugger (to greatly speed program debugging). We won't cover their use here, but they're worth further investigation. Control registers aren't accessible to applications programs (except perhaps through gates or calls provided by the operating system), and test registers won't be generally used. In fact, since test registers aren't a standard part of the 80386's architecture, future versions of the 80386 and future processors in the same family may not even have them.

New and Changed Instructions from 80188, 80186, 80286

Several new instructions were introduced with the simultaneous introduction of the 80186/188 and 80286. The 186/188 was an improved version of the 8086/88 with a few new instructions and lower clock counts, especially for complex instructions like MUL. It's been used mostly for coprocessor boards and process control in automobiles and other applications. The 80286 is, of course, the processor at the heart of the IBM PC AT and compatibles. In Real Mode it has the same instruction set as the 80186/188; in Protected Mode it has protection and other features similar to the 80386, but based on a 16-bit architecture.

The 80386 Real Mode has all the instructions of its predecessors; for the benefit of those experienced with the 8086 and for those interested in compatability between the members of the 8086 family we'll describe instructions that were new with the 80186/188 and 80286 Real Mode. Full descriptions are given in Chapter 4.

PUSHA AND POPA

These instructions cause all the general registers to be pushed onto or popped from the stack, in this order: AX, CX, DX, BX, SP (its value before the push), BP, SI and DI. The PUSHAD and POPAD versions, new on the 80386, push and pop the full 32-bit registers, while the PUSHA and POPA push only the lower 16 bits. On the 80386, these instructions take about the same amount of time as eight separate pushes or pops.

INS AND OUTS

These are block move instructions that move a byte, word, or dword, and then automatically increment or decrement SI, which is the pointer to the memory block being read or written.

ENTER AND LEAVE

These instructions cause a "stack frame" to be pushed onto or popped from the stack. The stack frame contains a count of the nesting level of the current procedure, a count of the number of bytes of storage used for parameter passing by the procedure, and the parameter bytes themselves. These instructions are favorites with compiler writers for structured high-level languages.

BOUND

BOUND compares a signed array index stored in a register to two memory locations, the lower and upper bounds of the index. On the 80386 these numbers can be 16 or 32 bits long. If the value in the register is not within bounds an exception 5 occurs.

IMMEDIATE OPERANDS FOR EXISTING INSTRUCTIONS

An immediate value can be pushed, used for multiplication, or used as a count for shifts and rotates. This can decrease register usage by putting operands in the program code itself, and decrease instruction counts by avoiding the step of loading a register with the value that now is given as an immediate operand.

New Instructions for 80386 Real Mode

The 80386 has many new instructions available in Real Mode (and also, of course, in Protected Mode, which has some other new instructions of its own). Also, some instructions act differently because of the availability of 32-bit operands. Several new instructions are the result of additions to the architecture changes.

LSS, LFS, LGS

The new Load Pointer instructions allow full use of the SS, FS, and GS registers. As with the other load instructions, a segment register and a specified other register are loaded with values from a specified location in memory. The LFS and LGS are needed to load the two new registers; LSS has been added to ease the use of multiple stack segments.

MOV

The MOV instruction can be used to move data to and from debug, control, and test registers. These new registers are now accessed by the MOV instruction.

MOVSX, MOVZX

With the availability of dword registers a problem arises. When moving a byte or word into a dword, what value should the added bit positions be filled with? MOVZX (MOV with zero-extend) causes the extra bits to be loaded with 0's; MOVSX (MOV with sign-extend) causes the extra bits to be loaded with the high bit of the source. MOVZX causes an unsigned number to keep the same value in its 32-bit version; MOVSX does the same for signed numbers. These operations also work for moving a byte into a word.

OTHER INSTRUCTIONS

Other new instructions are of new types, giving capabilities not available on the 8086 family until now. To the degree a program uses these instructions it becomes increasingly more capable than (and increasingly incompatible with) the earlier members of the 8086 family.

1. Long-distance Conditional Jumps. Instructions like JS (Jump if Sign) are limited on earlier members of the 8086 family to a range from 128 bytes before to 127 bytes after the current instruction. On the 80386 these "short jumps" are still most efficient (the instruction is smaller, but conditional "near jumps" (effectively to anywhere within the current segment) are also possible.

2. Single-bit Instructions. Four entirely new instructions are BT, BTC, BTR, and BTS. These are all Bit Test instructions that store an indicated bit in the Carry Flag. The differences in the instructions reflect what happens to the indicated bit; it can be left alone, complemented (reversed), reset (put to 0), or set (put to 1).
3. Bit Scan Instructions. BSF and BSR are new instructions that find the lowest-order (BSF) or highest-order (BSR) 1 bit in a word or dword register operand.
4. Double Shifts. SHLD and SHRD are double-precision shift instructions. They allow a 16- or 32-bit operand to be shifted by any number of bits up to 31, just like a regular shift; the difference is that the bit positions left empty by the shift are filled with bits from a register operand. The second operand, however, is left unchanged.
5. Set Byte on Condition. These commands are just like the conditional jump instructions; most of the conditions that cause a jump can cause a byte to be set. If the condition is met (for instance, in SETS if the sign bit is set) an indicated byte is set to 1; if the condition is not met the byte is zeroed. This is useful for evaluating multiple sets of conditions without the tangled code and slowed execution caused by repeated conditional jumps.

Other Differences Between 8086 and Real Mode 80386 Programs

The differences listed here are the ones that could possibly lead to compatibility problems between an 8086 program and the same program running in 80386 Real Mode. These differences apply to 8086 programs and to the same programs running as V8086 tasks. They're grouped by type in the list below.

GROUP 1: INTRUCTION SPEED IMPROVED

The speed of the 16 MHz version of the 80386 comes from two sources: faster clock speed (three times as fast as for many 8086s) and lower clock counts. The faster clock speed can cause problems for any program that assumes a given group of operations will take at least a given amount of real time. The lower clock counts can mean that delays required by certain I/O devices may no longer be supplied by the same code that used to provide them.

GROUP 2: WRAPAROUNDS

If an 8086 program specifies an address greater than 1 Mb (maximum possible is 1 Mb plus about 64 Kb), the address is truncated so it points somewhere within the first 64 Kb of the address space. On the 80386 the value is used as is. Page remapping can map this high 64 Kb area to the lowest 64 Kb of the address space for compatibility.

The 8086 allows accesses to bytes at an offset beyond 65,535 (code or data) to wrap around to byte 0 of the same segment. Accesses to offsets below zero (data only) wrap around to byte 65,535 of the same segment. The 80386 causes exceptions in both these cases (exception 12 for stack data, exception 13 otherwise).

Also, shifts and rotates are only controlled by the low-order five bits of the shift count. The greatest possible shift or rotate count is thus 31; values greater than this cause a shift by whatever remainder is left when the count is divided by 32.

GROUP 3: PREFIXES

On the 8086 redundant prefixes can be used indefinitely, the LOCK prefix can be used at will, and prefixes before an ESC instruction are ignored when execution resumes after the exception is handled. On the 80386 instructions must be less than or equal to 15 bytes long, which is only a problem if redundant prefixes are used. The LOCK prefix can only be used before certain instructions when they update memory: bit tests, adds and subtracts, increments and decrements, logical instructions, and XCHG when one of the operands is in memory. Finally, prefixes before an ESC instruction are executed, not ignored.

GROUP 4: EXCEPTIONS AND INTERRUPTS

On the 8086 a divide exception returns with CS:IP pointing to the instruction after the DIV. On the 80386 CS:IP points to the DIV instruction itself after the exception. On the 8086 the IDIV instruction causes an exception if its quotient is 80H (bytes) or 8000H (words); the 80386 returns the appropriate quotient.

The exception caused by an instruction executing when the 80386 is in single-step mode has higher priority than an external interrupt, so single-stepping a program prevents single-stepping an external interrupt.

When an NMI (Non-Maskable Interrupt) is received, further NMIs are masked until an IRET is executed by the routine handling the original NMI.

Finally, an undefined 8086 opcode may be defined when executed on the 80386, causing some instruction to execute. If undefined for the 80386, the opcode will cause exception 6. The 80386 has six new exceptions that only arise if a bug is encountered in the 8086 program.

GROUP 5: NUMERIC COPROCESSORS

The increased speed of the 80386 may cause problems when executing with a coprocessor. Where several lines of code might once have allowed sufficient time for the coprocessor to finish its work, the same code executing more quickly on the 80386 may finish executing before the coprocessor is done.

On the 80386 all coprocessor errors are vectored to interrupt 16. Interrupt controllers are no longer used for coprocessor interrupts, so an interrupt handler need no longer deal with a controller.

GROUP 6: PUSHED VALUES

Pushing the stack pointer SP now pushes the value of SP before it's incremented by the PUSH itself; previously the value of SP after the current PUSH was put on the stack. Also, PUSHF for the 8086 always has ones in positions 12 through 15. On the 80386 bit 15 is always zero and bits 12, 13, and 14 have the last value loaded into them; 12 and 13 together are the I/O Privilege Level, and 14 is the Nested Task flag.

OPERATING SYSTEM CONSIDERATIONS

There are many different types and styles of operating systems. The 80386, with its support for a flat memory model as well as other important features, will attract ports of many operating systems from minicomputers. From our point of view, though, the important difference between operating systems is the degree of access they give to the potential power of the 80386. There are many different levels of capability possible.

The simplest is a hypervisor that simply starts up one V8086 task, allowing a single user to run a single copy of an 8086 operating system. This generally will support a single program. Extending this initial model, paging can be used to emulate such features as "above-board" (beyond 640 Kb) memory. It can even allow virtual memory, so the program can use more memory than is actually physically available, the difference being made up by page swapping from a hard disk.

The next step is to allow the running of several V8086 tasks at once. One problem here is that as soon as multitasking is allowed all accesses to instructions that change the Interrupt flag must be trapped to the hypervisor; otherwise programs could hang each other up at any time. This is easy to do but increases processing overhead quite a bit. The operating system can be a single-user multitasking system, a multiuser system with each user single-tasking, or even a multiuser multitasking system. Such a system would either demand that the computer running it have a certain amount of physical memory for each user allowed to sign on, or use virtual memory so a hard disk could take up some of the slack. The individual tasks, however, would still be V8086 tasks that do not use the full power of the 80386.

Finally, this last hypervisor (a multiuser, multitasking system with virtual memory) can be extended to allow 80386 Protected Mode programs to run along with the V8086 tasks. This hypervisor would have to include a full operating system to handle system calls by the PM programs. It might include compatibility with DOS, for instance, so the V8086 tasks running under it could all be applications programs that access the same OS code in the hypervisor, avoiding the need to keep a separate operating system in each V8086 task.

This simplistic summary touches on the major issues from the point of view of making full use of the 80386. A different balance between the need for increased power and the need to keep using existing software will be struck by each owner of an 80386-based system, and this will help decide which operating system(s) to use.

Sample Hypervisors

As an example of the many ways the 80386 can be used, the initial hypervisor developed by Microsoft for the Compaq 80386 does very little. It starts a single Virtual 8086 task and then turns over control to the nearest copy of DOS 2.x or 3.x. The user starts up the copy of DOS and then runs the program as if on an IBM PC. The only addresses that are

page-protected are the ones where extended memory (beyond 640 Kb) would normally be found. When one of these addresses is accessed a page protection fault occurs, and the interrupt vector causes a task switch to the hypervisor. The hypervisor does the needed memory access and then returns control to the executing program. The V8086 task generally should run faster than the same program and operating system running on an 8086 or 8088, because the full speed of the 80386 is used with only a slight amount of extra overhead for accesses to extended memory.

A more capable operating system might allow more than one V8086 task to run at once. The user might be running Lotus 1-2-3 in the foreground while Turbo Pascal compiles a program in the background. These two programs will each get a certain percentage of the processor's time, as specified by the user. The processor will run one program for a few thousandths of a second and then switch to the other one. If there isn't enough RAM for both programs the 80386 and operating system will use virtual memory to keep only the currently used code and data in real memory. At any given time the contents of memory will be those page-sized pieces of code and data that have been used most recently.

Formerly tricky programs now cause little problem. For instance, a game might run only under a particular version of DOS. The user can start up that version of DOS and the game as a new task, separate from other programs, and swap back and forth between them at will. The game can be suspended when not in the foreground, or allowed to keep running. Only the most-used parts of the old version of DOS and the game will be in memory at any one time, the rest residing out on disk until needed.

Despite the support given by the 80386, an operating system that will allow all this is difficult to write. Books have been written on such problems as which pages to have in memory at any given time, and which pages to overwrite first when memory fills up; these are called "page replacement strategies." Debugging such an operating system can be an immense task. Once it's written and made to work, however, the capabilities of the computer running it are multiplied.

CHAPTER 7

The 80386 Processor In Depth

COMPUTER SYSTEM PERFORMANCE
HOW AN 80386 ACCESS MEMORY WORKS
INSIDE THE 80386
RIPLEY'S BELIEVE IT OR NOT
SOME APPLICATIONS PROGRAMMING CONSIDERATIONS

The 80386 processor is more complex than any of its predecessors, but correspondingly more powerful. A programmer who knows how the chip works right down to the hardware level will find it easier to write for and more enjoyable to work with. This chapter starts by looking at how the 80386 interacts with memory and coprocessors. Then we look at the chip itself and its functional parts or "units." Finally, we trace a series of instructions as they are fetched from memory, decoded to a form the 80386 can work with, and executed.

The information in this chapter isn't strictly necessary for getting started with the 80386. However, it's useful background, and issues covered here, such as caching, can make a difference in computer buying decisions. As you learn more about the processor and want to push it closer to its limits, knowing more about how the chip actually operates can really make

a difference. Finally, it's more fun to work with a computer when you have a solid understanding of how it behaves and why.

COMPUTER SYSTEM PERFORMANCE

The next several sections are useful as a buyer's guide for 80386-based computers and add-on cards. They describe several ways in which a computer system can be organized for optimum performance, and discuss some of the limitations of current technology; information that can translate directly into dollars-and-cents buying decisions.

A computer is made up of many parts, each of which sometimes has to wait for one of the others. A computer with floppy disk drives, for instance, may execute a thousand instructions in a thousandth of a second, only to wait half a second or more while data is read from disk.

The same type of interaction occurs between a processor and the memory it accesses, with the processor often having to wait for data or results to be successfully read or written before proceeding. This is why there are general registers on the chip, to save intermediate results of computations so we don't have to read from or store to memory as often. At some point the program does have to access memory, often large chunks of it in sequence, as in string and other operations. With a chip that executes instructions as fast as the 80386 does, speeding communication with memory is especially important; the advantage of having a fast chip is lessened if every memory access causes a long wait.

A controlling factor in discussing the speed of the 80386 is the clock cycle. A chip called a clock generator sends a signal on the 80386 CLK2 pin at regular intervals. A "cycle" of an electrical signal is the time it takes the signal's amplitude to rise, fall, and rise again to its starting point. The 80386 converts every two cycles sent by the clock generator to a single internal cycle, which we call a "clock." For a 16 MHz chip the clock generator sends a cycle every 1/32 millionth of a second; this is converted to an internal signal that oscillates every 1/16 millionth of a second, and so on for other "clock speeds."

Some internal functions of the 80386 take about half a clock to complete. Even the shortest instruction (such as a MOV from one register to another) takes two complete cycles, for reasons we'll examine later.

Figure 7-1 shows CLK2 signals in relation to the 80386 processor clock.

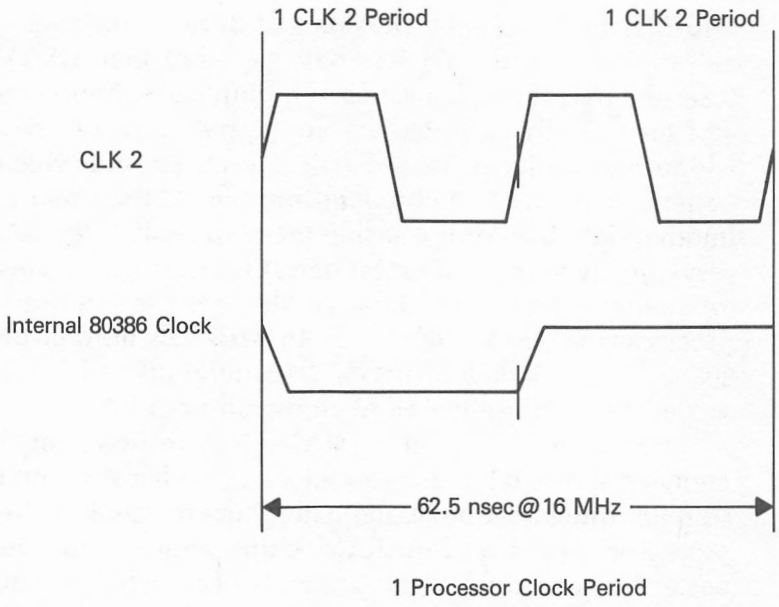


Figure 7-1

HOW AN 80386 MEMORY ACCESS WORKS

Many programmers speak glibly of "accessing memory" and "three-cycle RAM access" without needing to know hardware details. Understanding the power of the 80386, and when that power is useful, requires a little more understanding of what's actually happening in the computer.

When a processor needs to get a byte from memory it first calculates the address at which the byte resides. Then it activates (raises the voltage level of) certain pins, called address pins. The address pins taken as a group are called the address bus. On many computers the same pins are used for address and data, albeit at different and well-defined times. This "time multiplexing" is done on the 8086 and 80286, but not on the 80386, which has 32 address pins and 32 completely independent data pins. The pattern of activated pins alerts the chips that make up memory as to which byte to send to the processor.

The RAM chips at the specified address then read the byte from the chips where it resides and place it on the data bus (all the data pins together). The Dynamic RAM chips (DRAMs) used in most computers experience a loss of power when they place the needed data on the bus, and must be repowered before they can be accessed again. Static RAMs

(SRAMs) are faster than DRAMs and have no repower requirements, but are expensive and hold less data per chip than DRAMs. The data bus takes the byte through a series of multiplexors, transceivers, and receivers and then finally activates the appropriate pins on the 80386 to specify which bits of the requested byte are on (1) and which are off (0). The pattern of 1s and 0s is brought into the 80386 where it is placed on an internal data bus for use inside the chip itself. If the RAM chips respond very quickly to the request for data, the entire process (express an address to memory, place the data on the bus, and return the data to the processor) takes two cycles. A 16 MHz (16 million cycles per second) 80386 system therefore needs 2/16 millionths of a second (.000000125 seconds) to read a byte, word, or dword from RAM.

Notice the caveat "if the RAM chips respond very quickly." When a memory access takes an extra clock cycle because the RAM chips didn't respond quickly enough, the extra clock is called a "wait state." A slow processor can use slow RAM chips without introducing wait states, because a long single cycle is plenty of time for the chip to respond to a request. As faster processors are used (with each clock taking correspondingly less time) the same RAM chip can become a one-wait-state or even a two-wait-state chip when used on a particular system. The 80386 has several tricks that allow relatively slow RAM chips to be used while still avoiding wait states.

One such trick is called address pipelining. The term "pipelining" is usually used to describe the concurrent operation of prefetch, predecode, execution, and memory management units. In the 80386 the actual accessing of code and data from memory is also pipelined. Let's say that we're using slow RAM chips that would normally introduce one wait state in a 16 MHz 80386-based system, on which a single clock takes 62.5 nanoseconds (millionths of a second). A series of three reads, one followed by another, would go something like this:

Clock 1: START. Express an address on the address bus.
Clock 2: Wait while the RAM chips respond (wait state).
Clock 3: Data returned to chip on the data bus; END.
Clock 4: START. Express an address on the address bus.
Clock 5: Wait while the RAM chips respond (wait state).
Clock 6: Data returned to chip on the data bus; END.
Clock 7: START. Express an address on the address bus.
Clock 8: Wait while the RAM chips respond (wait state).
Clock 9: Data returned to chip on the data bus; END.

The 80386, however, can express an address for one read while the

data from a previous read is being returned, so a sequence of three reads takes less time:

Clock 1: START. Express an address on the address bus.
Clock 2: Wait while the RAM chips respond (wait state).
Clock 3: Data returned; END & START. Express next address.
Clock 4: Wait while the RAM chips respond (wait state).
Clock 5: Data returned; END & START. Express next address.
Clock 6: Wait while the RAM chips respond (wait state).
Clock 7: Data returned; END & START. Express next address.

After the first read (which takes three clocks) successive reads take only two clocks each. The process works about the same when writes are involved; a three-cycle write becomes a two-cycle write. In both cases the first memory access after an idle bus clock takes an extra clock, just as the first few instructions after a JMP or CALL (which disrupt pipelining among the 80386's units) take extra clocks.

There are other ways to avoid wait states. A RAM chip that introduces wait states when used at 16 MHz may be just fast enough to avoid them at a slower speed (like 12.5 MHz), but now the rest of the system is slowed down to avoid memory wait states. Address pipelining is optional because it requires precise synchronization between the processor and the rest of the system (which may not always be ready for an address to be expressed before the data is returned and picked up off the data bus by the processor). When 16-bit rather than 32-bit memory chips are used, address pipelining is not available.

Interleaving Memory

Every read or write request causes a brief sequence of events in a DRAM: power up, wait for a request, get request, answer the request (causing a drop in power), and repower up. The processor only has to wait until the request is answered before going on about its business, which might include immediately accessing some other chip (even while the first is still repowering up). If chips in the same bank of DRAM are accessed twice in a row, the following sequence occurs: get request, answer the request (causing a drop in power), get new request, power up (processor waits), answer the request (power drop), and repower up. The extra waiting time while the DRAM bank powers up again can cause a wait state to be added to any memory accesses that try to use the same bank twice in a row.

The simplest form of memory organization has all the chips in one bank (for our purposes a bank is any chunk of memory that can be

resupplied with power while another bank is being accessed). Dividing the memory into multiple banks can save extra wait states caused by back-to-back accesses to the same bank. It's very common for the processor to access memory in sequence; first this address, then the following one, then the one after that. Thus successive addresses should be in different banks. This can be implemented with a two-bank interleaved memory system; each successive dword is in a different bank. This is done by using the lowest bit of the address as a bank selector. Sequential accesses never induce extra wait states by hitting the same bank twice in a row. Random back-to-back accesses have a 50:50 chance of causing delays; introducing more banks would reduce the delays for random accesses without changing the situation for the more common sequential accesses. The additional cost of increasing the number of banks is rendered less worthwhile by the 80386's pipeline and the use of caches (below), which allow some extra wait states to be tolerated without delays in actual processing.

Figure 7-2 shows Interleaved RAM.

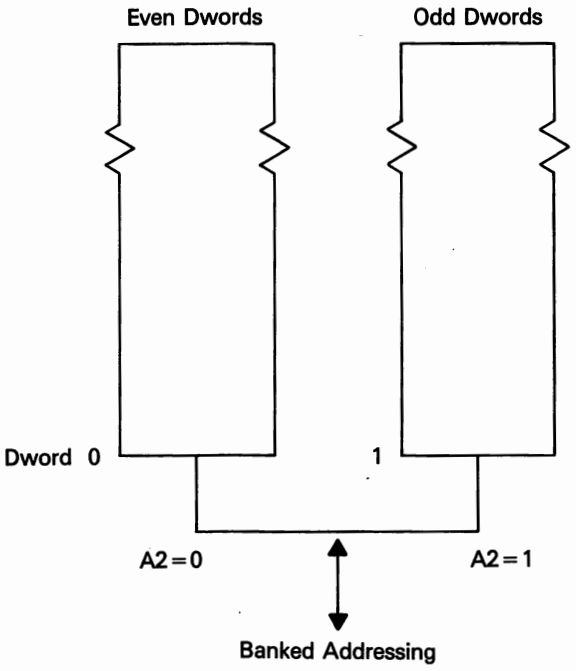


Figure 7-2. Interleaved RAM (2 banks)

Memory Caching

On 80386-based systems caching is the use of a small amount of faster memory to hold frequently-used or soon-to-be-used information. In virtual memory all of RAM can be used as a cache for a hard disk, while on the chip itself the prefetch and decode queues act as caches for information the execution unit would normally have to ask for from memory. The term "caching" is often used to mean memory caching, which adds a group of fast memory chips (generally no wait state SRAMs) to hold a subset of the information held in large amounts of slower (probably 2 wait state DRAMs) main memory. Figure 7-3 shows a block diagram of a cache system.

An 80386 running at 16 MHz can do a memory read in 125 nanoseconds (2 clocks) if the memory responds fast enough. On pipelined accesses the available time rises to 187.5 nanoseconds (3 clocks) because the first clock of a memory access takes place during the last clock of the previous access. Table 7-1 illustrates various RAM chip response times. As of this writing most SRAMs respond within the 125 nanosecond window of a non-pipelined access, introducing no wait states. Expensive DRAMs respond in a window between 125 and 187.5 nanoseconds, introducing one wait state (making the processor wait one clock) on non-pipelined accesses and adding no wait states when pipelining is in effect. Cheap DRAMs respond in 187.5—250 nanoseconds, adding two wait states for non-pipelined or one wait state for pipelined accesses. Non-cache systems made with expensive DRAMs have good performance. In order to build large memories of the size which 80386-based systems can easily handle, however, the cheaper DRAMs are needed, and an added cache of SRAMs will noticeably improve system performance.

A SIMPLE CACHE

When a cache is used accesses to memory become more complicated. On a non-cache (and non-virtual memory) system the processor asks for

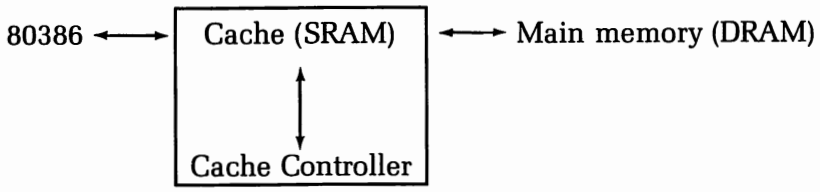


Figure 7-3. Cache Subsystem

Type of Access	Window at 16 MHz	Type of Chip	Number of Wait States
Non-pipelined read	125 nanoseconds	SRAM	0
		Fast DRAM	1
		Slow DRAM	2
Pipelined read (saves 1 clock)	187.5 nanoseconds	SRAM	0
		Fast DRAM	0
		Slow DRAM	1

Table 7-1. RAM chip response time

information and main memory provides it. In a cached system the processor asks for information and the cache controller checks to see if the information is in the cache and provides it if found. Otherwise, the request goes through to main memory, which provides the information. Table 7-2 summarizes the timing involved in these cases.

From the data in the figure we can see that a 50% cache success rate (hit rate) means that half our accesses must work (0 wait states) just to

Action	Clocks Needed
Non-cached Read:	
Express address to memory	1
Wait for RAM chip	2
Data returned	1
Total	4 (2 wait states)
Cached Read (hit):	
Express address to cache	1
Check cache, return data	1
Total	2 (0 wait states)
Cached Read (miss):	
Express address to cache	1
Check cache, no match	1
Express address to memory	1
Wait for RAM chip	2
Data returned	1
Total	6 (4 wait states)

Table 7-2. Cache vs. Non-Cache Accesses

balance out those that don't work (4 wait states) and match the performance of a two wait state non-cached system. Luckily, small caches of 64 Kb or even less can offer hit rates around 90%, justifying the expense of the cache. Table 7-3 shows hit rates for a direct-mapped cache with a 4-byte line size; other choices of mapping and line size improve the hit rate by a few percentage points. The most commonly available SRAM chips support a 64 Kb cache at minimum cost.

The actual size of RAM makes little difference to the performance of a cache. In a given period of time most programs access the same few Kb of code and data repeatedly. For instance, if you're typing a letter in a word processing program, the print control and help parts of the program might go unused for several minutes at a time. As you type, make changes, and then type some more, the same data is accessed over and over. A minority of programs, however, behave poorly from a cache's point of view. For instance, during repeated recalculating of a large (bigger than the cache) spreadsheet the cache can be a liability, but entering data to the spreadsheet is much like the using the word processing program (repeated use of small pieces of code and data).

The next section goes into greater detail about how a cache works on an 80386-based system, and may not be for everyone. Caches can improve the performance of a system, but a computer is just as likely to be I/O-bound as memory-bound (especially as regards disk and video updating), so due attention should be paid to many factors other than caching in designing or purchasing a computer.

CACHE ACCESSES

Several problems and performance issues come up in building a cache. We'll cover them here in just enough detail to give a working vocabulary

Cache Size	Hit Rate	Performance Gain
No cache, all 2-wait-state DRAM	0%	0%
16K	81%	35%
32K	86%	38%
64K	88%	39%
128K	89%	39%
No cache, all 0-wait-state SRAM	100%	47%

Table 7-3. Cache Size and Performance

for the computer buyer and programmer. The discussion here assumes a 64K direct-mapped cache that can access up to 16 Mb of memory.

The cache itself has two levels. The first is made up of a block of tag SRAMs that contain address information. In this direct-mapped cache every 256 addresses map to one location in the cache. The cache contains a copy of the information at one of these 256 locations. Note that 64 K of cache entries × 256 addresses/entry yields 16 Mb addressable. The second level contains data SRAMs that have the actual data copied from DRAM.

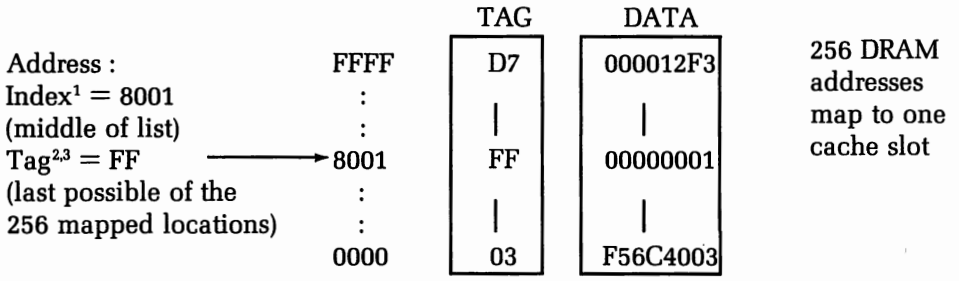
The 32-bit address from the 80386 is divided into three parts. The first 8 bits should be zero, since this cache only covers 16 Mb of memory. An exception might be generated if a non-zero value was found here. Otherwise, the next 8 bits are used as a "tag" field, and the final 16 bits are an "index" to determine which address in DRAM is being accessed.

The tag SRAMs contain a list of all the 4-byte "blocks" that are in the cache itself. The "index" tells us where in the list the information we need might be. The tag field tells us which of the 256 possible DRAM locations for a given cache location is actually in the cache. If the piece of data in the cache is the needed one, a hit has occurred and the data is sent to the processor. Otherwise, a miss is reported and the request is sent to DRAM. Normally the data from DRAM is also copied to the cache so it will be there if the same information is requested again.

The cache shown in Figure 7-4 is 4 bytes wide; each location holds a dword. Wider (8 or even 16 byte) caches can be used, which slows cache updating but improves hit rates. An 8-byte-wide cache has a hit rate several percentage points higher than the 4-byte-wide one, at a considerable cost in complexity.

Also, every location in main memory is mapped to exactly one cache location. If a program repeatedly accesses two locations in turn and both are mapped to the same cache location, a "cache thrash" occurs: check cache for data item 1, not found, get from DRAM, update cache; check cache for data item 2, not found (because just updated with item 1), get from DRAM, update cache; check cache for data item 1, not found (because just updated with item 2), get from DRAM, update cache; and so on as long as the alternating accesses occur.

This particular problem can be avoided by allowing any location in main memory to go to either of two locations as needed. Now the cache controller has to look in two places on every cache access, and when placing a new item in the cache it has to decide which of the two possible locations to use. This is called 2-way associativity, and using it



1. 16-bit Index tells which location of 64 K locations in tag SRAM to check.
2. 8-bit Tag in SRAM tells which of the 256 possible dwords is in the data SRAMs.
3. If requested Tag is same as Tag in SRAM, a hit has occurred. If not, access goes to DRAM and Tag and Data are updated to hold accessed data.

Figure 7-4. Cache Access

improves hit rates by perhaps 1 percent (at a high cost in control logic). Four-way caches are possible, as are fully associative caches (which allow each location in memory to map to any slot in the cache). These can be built at high cost in complexity and low improvement in hit rates for most applications.

Because of the problems introduced by writing to a cache (the cache now has information that main memory doesn't; what if a DMA controller or other processor tries to access stale memory whose correct contents are in the cache) many caches are read-only, sending all writes directly to main memory. The concerns that arise when writing to a cache are called "cache coherency" problems, and are similar to the updating problems (solved by accessed bits and dirty bits and other techniques) faced by paging systems.

Page Mode RAMs

When a dword is read from memory a total of 32 memory chips are accessed, each contributing a single bit to the result. The dword's address is divided up into row and column numbers that tell each chip which bit to set. If the row number is held constant from one access to the next only the column number must be changed to indicate which bit to grab from within each chip.

With some simple comparison logic at the top of the address bus the system can recognize when an address will result in the same row being

accessed (with only the column number changing). Such an access can be made more quickly than usual, and a 1 wait state RAM will give no wait states for this type of access. Depending on the number of rows and columns within, for example, a megabit chip, perhaps 2,048 bits (512 bytes) can be accessed at these higher speeds. The technique is called Page Mode RAMs, and it depends on RAM chips that are organized in such a way as to respond more quickly to these accesses. When it's used the effect is close to the same as having a 512 byte cache, but there's no penalty for misses, just a speeding up of all accesses within the same 512 byte group as was last accessed.

Memory Organization

One of the big problems for 80386-based systems is maintaining compatibility with existing 16-bit computers and software. The earlier computers were 16-bit machines, while the 80386 is 32 bits throughout. Many of the details of programming the 80386 involve choices between 16- and 32-bit operands and addressing. How are these conflicts resolved in the hardware? The 80386 can be connected to either a 16-bit data bus or a 32-bit one. We'll look at each situation; the 32-bit bus first because it's simpler, then the complications (nearly invisible to the programmer) introduced by connecting a 16-bit bus to a 32-bit computer.

To the programmer the 80386's memory is seen as a single sequence of bytes; a word is any two bytes in a row, and a dword is any four bytes in a row. Actually memory is divided into four sections, each one byte wide, so instead of one long stack of bytes we have four short stacks of bytes arranged next to each other, each stack with its own connection to the data bus.

The four stacks together can be seen as a single stack of dwords. Figure 7-5 shows this memory organization and how 8-bit-wide sections of the data bus are connected to each stack of bytes. When the 80386 asks for a dword from memory a byte is taken from each short stack, placed on the appropriate section of the data bus, and then sent to the processor. This is how the 80386 can grab a dword in a single memory access.

This division of memory is reflected in the pins of the 80386. We speak of the 80386 as having a 32-bit address bus, allowing it to access 2^{32} bytes of memory, but a close look at a pinout of the chip shows only 30 address pins, A2-A31. Where are pins A0 and A1?

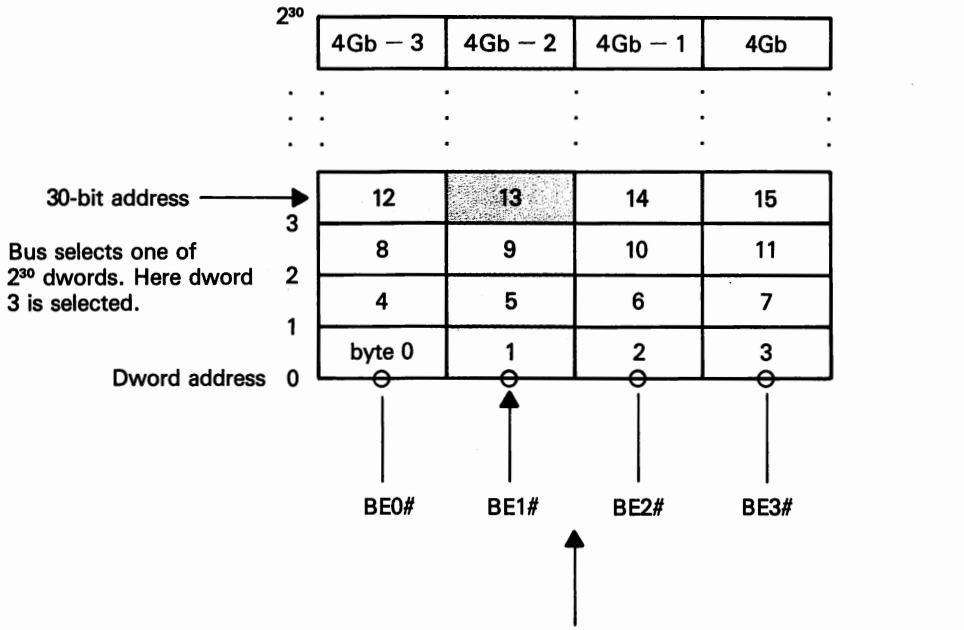


Figure 7-5. A 30-bit address bus and four byte enable lines allow 2^{32} bytes to be addressed. Here the second byte in dword 3 (byte address: 13) is accessed.

The address bus actually indicates which dword to select from memory, not which byte. The address pins allow us to select from among 2^{30} dwords (which at 4 bytes per dword is the same as 2^{32} bytes). We know that the 80386 allows us to select any single byte from memory, as well as any dword, so once a dword is selected how are individual bytes chosen from within it?

One way would be to bring the whole dword into the processor and then allow internal logic to select which byte(s) to use, but instead the byte selection is handled by the bus systems. The 80386 has four byte enable pins, named BE0# through BE3#. While the address pins tell which dword to select, the byte enable pins tell which byte(s) within the dword to choose. The two low bits of a 32-bit 80386 address are used to determine which byte enable pins to activate.

A word which is at an even address (like 0, 2, 4, etc.) is said to be "aligned on a word boundary;" a dword at an address which is a multiple of 4 (like 0, 4, 8, and so on) is "aligned on a dword boundary." Most programmers know that it's important when setting up data structures to

align words on word boundaries and dwords on dword boundaries. A description of the interaction of address lines and byte enables in the address bus shows when and why this is important.

If a non-aligned dword is selected, a single access can't get the whole dword. For instance, if the dword is at byte address 2, half of it is in the first dword of memory and the other half is in the second, so the 80386 automatically makes two accesses. The first gets the upper two bytes, at byte addresses 4 and 5. This is done by activating the address pins which select the second dword and BE0# and BE1# to get the first and second bytes. The second access gets the lower two bytes, at byte addresses 2 and 3 (the address selects the first dword and BE2# and BE3# get that dword's third and fourth bytes). The bytes are automatically arranged in the correct order on the data bus, but getting the dword has still taken two memory accesses. If a dword is aligned on a dword boundary the address pins select the dword and all four byte enable pins are activated at once to grab each of the needed bytes.

Understanding how the 80386 interacts with a 32-bit bus makes it easy to deduce some of the consequences of using a 16-bit bus instead. The 80386 has an input pin, BS16# (Bus Size 16), which is activated by some device on the bus to tell the 80386 when it's accessing 16-bit devices. If the entire data bus is 16 bits then BS16# is asserted for every access.

The processor only uses the lower 16 bits of its data bus when accessing 16-bit memory, and this causes predictable consequences; a word that is aligned on an odd boundary requires two accesses to bring to the processor, one for each byte. A dword takes two accesses if aligned on a word or dword boundary (half brought by each access), and a dword on an odd boundary takes three separate accesses (the first gets a single byte, the second gets the middle two bytes, and the third gets the remaining byte).

Also, address pipelining is turned off during 16-bit accesses, so no extra cycles are available for the memory to respond. A 16-bit data bus hooked up to two-wait-state RAM chips can therefore cause considerable delays in memory-intensive operations, and correct alignment of data structures on word boundaries becomes very important. Prefetching (discussed below) is also affected. A slower or smaller bus is kept busier by the execution unit as it tries to keep up with the memory accesses of the commands in progress. The prefetcher then has fewer cycles to use for bringing in code, at a time when it needs extra cycles to overcome a smaller bus size, increasing the likelihood that the prefetch and decode queues will empty, slowing execution.

These differences boil down to some simple rules for the programmer. All words should be aligned on word boundaries (always important with 16-bit buses) and all dwords should be aligned on dword boundaries (always important with 32-bit buses). Finally, a program that shows good performance on one computer may work noticeably better or worse on another, depending on concrete factors like RAM chip speed and bus sizes, but also on interactions between them that are difficult to predict without actual testing.

DESIGN TRADE-OFFS

In terms of expense and complexity, the design options above boil down to two main techniques for improving memory performance: memory caching and everything else. If all the non-caching techniques are used (interleaving, page mode RAMs, RAMs with only one wait state, a 32-bit data bus, and address pipelining) performance will approach the best possible, which would be zero-wait-state SRAMs for all of memory. If none of these techniques are used the resulting slow computer can be brought up to optimum speed by adding a cache.

The first two 80386-based systems on the market illustrate these design choices. The Compaq 386 doesn't use caching; it uses page mode RAMs to generate a 512-byte effective cache, and it uses RAMs that are one wait state when used with a 16 MHz 80386 (the only kind available at this writing). It also has a special 32-bit bus for fast memory access; the rest of the system (I/O and extra memory) is on a 16-bit bus. Despite the lack of caching, the performance of this well-designed machine's memory system is very good.

The other 80386-based systems available now are an expanding number of 386 Turbo cards. These are small (about 3" by 5") computer cards that have an 80386 and some support chips and are designed to plug into an IBM PC AT or compatible. Circuitry on the card maps the 80386's pins into the card's connector, which mimics the pins of an 80286; just remove the 80286 from your AT, plug in the turbo card, and voila!—an instant 80386 system.

The turbo card makers have no control over what kind of RAMs and memory organization techniques are used on the computers that host them, so the best will feature an SRAM cache like the one described above, which helps the processor avoid accessing regular RAM at all on about 90% of memory reads. One turbo card has a 64 Kb SRAM cache with no real bus; the 80386 pins are wired directly to the cache for

greater performance. An IBM PC AT with this turbo card and a good hard disk will come close to the performance of an 80386-based machine like the Compaq 386 at a much lower price (especially for those who already own an AT or compatible).

Other systems will use different techniques or combinations of techniques to achieve the best possible performance at the lowest attainable cost (and the shortest amount of development time). One factor that will greatly affect future machines is the forthcoming availability of faster versions of the 80386. RAM chips that are one wait state at 16 MHz can become two wait state chips at 20 or 25 MHz, increasing the importance of the techniques discussed here. In fact, a non-cached computer may actually deteriorate in performance when a faster processor is added because of the increase in wait states. This depends very much on timing particulars that will be different for every system.

Adding a coprocessor is also a popular way to speed up program execution. The 80386 can use either an 80287 or an 80387, although the latter isn't available as of this writing. The 80387 should offer perhaps three times the performance of the earlier coprocessor, but until it's available this is a moot point.

I/O INTERFACE

The terms "memory-mapped I/O" and "I/O-mapped I/O" are tossed around easily by programmers, but what do they really mean? In terms of hardware, not as much as you might think. The same control logic, data bus, and address bus are used whether accessing memory or I/O. Of course, an I/O device is different from a memory chip, but a lot of flexibility is possible in talking to the input/output device.

There are two real (non-virtual mode) address spaces on the 80386. The one we've been discussing is the memory address space, which can contain up to 2^{32} bytes. This space can contain I/O devices as well as memory. Some advantages of this are listed below.

1. Addresses are all 32 bits wide.
2. Any instruction that addresses memory can be used. This includes AND, OR, and TEST instructions. Because most instructions can be used, all addressing modes can be used to access I/O devices.
3. I/O spaces can be protected, using either segment or page-based protection.

4. The other way to handle I/O for a given device is to put it in the I/O address space. This is a separate 64 Kb, which is not accessible via normal memory accesses or by most instructions, including data transfer instructions. This separation is enforced by the M/IO pin on the 80386, which tells the address bus whether to use memory or I/O. Only the IN, OUT, INS, and OUTS instructions can be used to access the I/O address space.

The advantages of I/O-mapped I/O are:

1. Addresses can be 16 bits wide, or even smaller, so address decoding is simpler.
2. I/O spaces are inaccessible to most instructions, protecting against most accidental reads or overwriting.
3. The two techniques can be combined by mapping the I/O space onto a 64 Kb section of the memory address space. With this method software that assumes memory-mapped I/O and software that uses IN, OUT, and so on will all work on the same system. The addressing advantages of each method are kept this way, but the protection advantages of both methods are lost; segment-based protection, for instance, is no protection against the OUTS instruction.

INSIDE THE 80386

So far we've talked a lot about the hardware organization of an 80386-based system; now we'll discuss the organization and internal workings of the processor itself. Everything discussed below works the same no matter what the processor is connected to. Understanding these details isn't too important in buying decisions, but it can be of great help in programming better applications in assembler.

Instruction Pipelining Made Easy

The most important technique for speeding access to memory (pipelining) is built into the 80386 from the ground up. Normally a processor executes an instruction in this order:

1. Fetch an instruction from memory.
2. Decode the instruction into a standard form that will be processed further by the execution unit.
3. Execute the instruction itself.

Only one step can be done at a time, so processing might proceed something like this:

Time 1: Fetch instr1.
Time 2: Decode instr1.
Time 3: Execute instr1.
Time 4: Fetch instr2.
Time 5: Decode instr2.
Time 6: Execute instr2.

Pipelining combines these steps so that at any given time several steps are being executed at once. One instruction might be fetched and a second instruction decoded while the third is being executed. In order to do all this simultaneously the processor itself must be divided up into semi-independent units, each of which can perform its own given function while the other units are simultaneously performing theirs. Let's assume that the processor is divided into a prefetcher, a decoder, and an execution unit (executer being too strong a term) and see how several things might be accomplished at once (Table 7-4).

In 6 units of time 4 complete instructions have been prefetched, decoded, and executed, and two instructions have been partially processed (prefetched and perhaps decoded, but not yet executed). The term "prefetching" is used to indicate that the instruction was fetched while other things were occurring, so the rest of the system didn't have to wait

	Prefetcher	Decoder	Execution Unit
Time 1:	Prefetch instr1		
Time 2:	Prefetch instr2	Decode instr1	
Time 3:	Prefetch instr3	Decode instr2	Execute instr1
Time 4:	Prefetch instr4	Decode instr3	Execute instr2
Time 5:	Prefetch instr5	Decode instr4	Execute instr3
Time 6:	Prefetch instr6	Decode instr5	Execute instr4

Table 7-4.

for a fetch step. Without pipelining only two instructions could have been completely processed in the same amount of time, as shown above. In this simple example pipelining has allowed us to do twice as much work in a given time.

Real life is more complicated than this. Prefetching, decoding, and executing don't take equal amounts of time for every instruction; sometimes one instruction might tie up the data bus, forcing another instruction to wait to be prefetched. Also, the prefetcher always grabs instructions in sequence, no matter what the instructions do. Any instruction that causes a change in sequence (a CALL or a JMP that is taken as opposed to one that falls through to the next instruction) or an interrupt will cause the 80386's queues to be emptied. The instruction that is processed immediately after a change in sequence takes longer because the execution unit must wait for it to be prefetched and predecoded before starting the execution step.

The 80386 is divided up into units whose functions can be summarized as follows. The prefetch unit's job is to get instructions from memory onto the chip. It uses the bus during times when the execution unit doesn't need it, grabbing instructions from memory and storing them in a prefetch queue on the processor. Sixteen bytes of instruction code (usually three or four instructions) can be held here. The decode unit then translates the instruction into a standard format that can be used directly by the control and execution units. Three instructions can be held in the decode queue. Part of the decoded instruction is a pointer into the control ROM, which holds the microcode to orchestrate the actions of the 80386's units.

The execution unit then acts under orders from the control unit to actually perform the instruction. The execution unit includes an ALU for arithmetic and the chip's register set. The segment unit calculates addresses and checks them against a segment's protection. The address from the segmentation unit then goes to the paging unit, which (if paging is on) translates it into a page-based physical address; otherwise the address is used as it comes from the segment unit. Finally, the address is sent to the bus interface unit, which controls access to both the address bus and the data bus. The bus interface unit is also used by the prefetcher for its requests and by the paging unit for access to memory-based page translation tables.

There are other microprocessors that pipeline instructions to about the same extent as described above. What makes the 80386 unique is that it does address translation and protection checking for segments and,

surprisingly, for paging on the chip itself. Other processors use a chip called a memory management unit (MMU) for this. Sending an initial address to it and then waiting for a final address to be returned, with protection checking completed, slows down the rest of the system. Also, different computers can have the same processor but different MMU's, resulting in compatibility problems. On the 80386 the MMU is not only onboard, but its operations are pipelined right along with fetching, decoding, and executing, so address calculation takes no extra time; only the actual accessing of memory during execution affects the number of clocks an instruction takes. Some complicated addresses take one extra cycle to calculate, as described in Chapter 2; a small percentage of paged addresses take a few extra cycles, as described below.

80386 Units in Detail

Any discussion of the 80386 includes mention of its ability to save time by pipelining instructions; sometimes a diagram depicting the chip as having separate units is included. Let's take a close look at the chip itself and see what makes it tick. We'll start with an overview of the 80386's units (which may be enough for some readers), then plunge into the units themselves and the detailed makeup of each of them. The foldout map of the chip included in the back of the book will be referred to and explained here.

The last part of this chapter will follow a short series of instructions as they're prefetched, decoded, and executed. Using the detailed look at the chip itself as background, this will demonstrate just how the processor is able to keep so many things going on at once, and how different parts of the process are kept from interfering with each other.

This section makes repeated references to the foldout figure at the back of the book, so you may wish to have the figure out while you read. This figure is more detailed than any Intel or other documentation we've seen, leading to differences between this figure and others which show the 80386's units. An example is that Intel documentation shows the segmentation unit as having a three-way adder; our figure shows two two-way adders (labeled +) in series, which have the final effect of adding three numbers together. Both approaches are accurate, but the more detailed one gives better insight into how the processor actually works.

The foldout figure shows a summary view of the 80386's pinout. The processor actually has 132 pins; 82 of them are named in the figure. The

50 missing ones include 41 power pins that distribute power to the various units, 8 pins that are unused, and a pin called HLDA. The 82 pins accounted for include the 32 data pins and the 34 pins used for addressing. Of the 16 other pins like INTR, M/IO#, and BS16#, most are described elsewhere in this chapter.

The figure also shows the contents of each of the processor's units. Now for the grand tour.

PREFETCH UNIT

The prefetch unit is one of the keys to the speed of the 80386. Its basic purpose is to use otherwise empty bus cycles to bring instructions from memory onto the 80386. A 16-byte code queue holds the instructions until the decoder needs them. The prefetcher labors under several handicaps, including having last priority access to the bus and not knowing when jumps or calls will change instruction sequencing. The prefetcher always grabs instructions in the order in which they appear in memory. In fact, the prefetcher simply grabs code one dword at a time, not caring whether it's bringing in complete instructions or pieces of two instructions with each access. When a prefetched and decoded jump is executed and the jump is taken, the contents of the prefetch and decode queues are cleared out.

When the jump is taken and the queues are empty, processing is slowed down. An instruction is fetched, immediately decoded, and then executed. If there are enough cycles in which the bus isn't busy during decoding and execution, the prefetcher starts filling up its queue again. If the first instruction after a jump is a MUL or some other instruction that takes several clocks to calculate, the prefetcher can catch up quickly. Program loops, for instance, can perform better with a MUL or similar instruction at the top of the loop. Simple instructions like CLC or a register-to-register MOV execute so fast that they may not let the prefetcher catch up, causing further delays if the queue runs dry again.

INSTRUCTION DECODE UNIT

The instruction decode unit also saves time for the processor as a whole. Since it doesn't use the bus it can do its work any time the prefetch queue isn't empty. Each byte in an instruction implies whether or not more bytes follow, telling the decoder whether it has a whole instruction done yet (the decoder can translate about one byte per clock).

The decoder's function is to translate instructions into a form the execution unit can use quickly. A possible format of a decoded instruction is shown in Listing 7-1. The instruction as a whole takes up 112 bits. The first three are yes-or-no bits for whether certain prefixes are present in the assembly language instruction. The next 12 are the address in the control ROM of the microcode which will actually cause the instruction to execute. Twelve bits allow addressing of up to 2^{12} (4096) separate items; there are actually 2560 words of 37 bits each in the control ROM.

The next three bit fields give the numbers of the segment register, base register, and index register used by the instruction in addressing. Two valid bits tell whether the base and index registers are needed by this instruction. A two-bit scale factor allows four scale factors: 1, 2, 4, and 8.

Two large 32-bit wide fields tell the displacement of an operand and give an immediate operand if any. Two more bit fields give the number of two registers that may also serve as operands. Whether the operand is in memory, immediate, or a register, the execution unit can get the information it needs from the decoded instruction. The last 14 bits of the decoded instruction are flags that we won't discuss in detail here. Note that the entire decoded instruction is simply the 80386's translation of the machine language generated by the assembler. If code could be written directly in this 112-bit form, both assembly and decoding could be skipped (at a large cost in human time to write the needed code).

Length	Meaning
1	LOCK prefix present
1	Address Size prefix present
1	Operand Size prefix present
12	Control ROM entry address
3	Segment Register
4	Base Register number, and valid bit
4	Index Register number, and valid bit
2	Scale factor
32	Displacement
32	Immediate operand
3	Operand Register (source).
3	Operand Register (destination)
14	Other flags, modifiers, and so on
112	

Listing 7-1. Sample Definition for the Decoded Instruction Queue Entry.

CONTROL UNIT

This part of the 80386 actually has hooks into each of the other units, directing their actions and interactions. The heart of the control unit is the Control ROM, which contains the actual instructions used inside the 80386. These instructions are 37 bits wide, and there are currently 2,560 of them stored on the ROM. As mentioned above, the microcode instructions are accessed by a 12-bit field in the decoded instruction.

The actual format of a microcode instruction is proprietary to Intel. There are some facts that are generally known, however. Having the control words in a ROM makes it much easier to upgrade and debug the processor as a whole. The ROM is not changed lightly, but it is changed much more easily than anything else about the 80386. Even serious flaws in a processor are often repaired by reprogramming the ROM to work around the flaw rather than by changing the chip itself.

Also, the 37-bit microcode words used for control are fairly narrow. Wide control words tend to contain enough information to direct every unit on the processor at once. The narrow words used here tend to direct one unit at a time. Since the prefetcher and decoder follow fairly straightforward action sequences, it can be assumed that most of the words in the ROM direct the actions of the other units, especially the execution unit.

The control unit is also the only unit besides the bus unit that can talk to the outside world. The pins which connect here go to a Request Prioritizer, which tells the processor when to do what. This is very important since 6 or 7 instructions may be in various stages of processing at any one time and the 80386 may be part of a multiprocessing system in which it's not the only chip which can control the system bus.

The next three units discussed are each very complicated. We take a quick look at them here, concentrating on hardware details. The next section of this chapter, which follows a short sequence of instructions as they execute, describes further how they work and interact with one another. Details of register usage, protection, and paging are covered elsewhere in the book.

TIMING

The details of timing within the 80386 are very complicated, but some basic rules apply. Moving information from one queue to another queue, a queue to a register, or one register to another takes one clock. The information must get to one of these safe places before a new cycle starts

or it could be wiped out by new data placed on the same section of the internal bus. Any move between registers (including 80386 internal registers), for instance, is assumed to take one clock, no matter how many adders, muxes, and so on are between the registers. These single clocks don't necessarily show up in the total execution time of an instruction, because many of them are pipelined with other processes. One of the few things that always takes noticeable clocks is getting operands from memory (four-plus extra clocks) and using a memory operand as a destination. This takes five clocks: four to get the operand and one to start the memory write. The second clock of the write can be pipelined, proceeding while the execution unit starts work on another instruction.

Also worth noting is that different activities within the 80386 can be seen as taking place at half-cycle intervals. This is convenient for discussion, but not always an exact description of timing.

EXECUTION UNIT

The execution unit has several important pieces: the register file (with all the general and control registers) and the math unit (including a shifter, adder, and special logic for multiplication and division). Two inputs come into the execution unit: a line from the bus unit containing instruction operands that come from memory, and a line from the decoder that contains immediate operands and displacements for operands. A multiplexor (labeled MUX) selects one of these two inputs and sends it to a register or temporary register for future use.

Numbers from the register file can be sent straight to the segment unit for address calculations or they can loop through the arithmetic area. Let's say we have an instruction, ADD EAX, MEM_DWORD. The operand MEM_DWORD is brought in by the bus unit and placed in a temporary register (this takes a total of four bus cycles, since operands are gotten from memory during execution). Both operands are then sent to the arithmetic area, (where they are added together) and the result is placed in a special register called AR_OUT. The adding and placement in AR_OUT takes one bus cycle. A second bus cycle is needed to get the result from AR_OUT and place it back in EAX.

The execution unit can't do anything else while the result is moved from AR_OUT to EAX, since the very next instruction might use EAX. If the new value had not yet been moved into the register, the next instruction might use the old value, as the new value circles around the internal bus to be placed in EAX. This problem (an instruction needing a

value that isn't yet back where it's needed) is called interlock. Since the 80386 doesn't have logic to check for this problem, it simply takes two bus cycles for even the simplest instruction (like a MOV from register to register), thus preventing interlock.

One of the toughest tasks for the 80386 is giving better performance than 16-bit processors while dealing with pieces of data that are twice that size. Some of this power is provided in the hardware. For instance, the barrel shifter is 64 bits wide. This allows it to handle a 32-bit shift of a 32-bit operand in a single clock. More power is provided by clever coding. For instance, a MUL instruction takes between 9 and 38 clocks: six clocks, plus one clock for each significant position in one of the multiplicands (minimum three). Multiplying by 3 (highest bit in second bit position) takes (6 + 3 =) 9 clocks. Multiplying by a number with the highest bit in the last (32nd) position takes (6 + 32 =) 38 clocks. A performance penalty is only paid when the full capability of the processor is being used; simple operations proceed quickly. Most instructions (for example, adds) proceed at the same fast speed regardless of the size of the operand.

SEGMENTATION UNIT

The address for reading or writing to memory comes out of the execution unit in two places: a displacement and a scaled index. These two numbers are added and held in a temporary register. If a base is also needed (i.e., the address has a base, an index, and a displacement), the result must be looped back into the adder and added to the base coming from the execution unit. This takes an extra clock whether or not the index is scaled. The result of all this adding is an effective address, which is the offset part of an address after all calculations are made.

The segmentation unit compares the effective address to the length limit for the segment in use, as determined by examining the segment descriptor. All of the segment descriptors in effect at any given time (CS, SS, DS, ES, FS, and GS) are stored on the 80386. Access rights are also compared to the operation that the processor is attempting, and a problem in either area causes a protection exception to be generated. If protection checks are met, the segment base is added to the effective address, yielding the linear address. If paging isn't in effect this is the same as the physical address that will be used to access memory.

PAGING UNIT

The paging unit sometimes does nothing, since paging may not be in use in simpler systems. When it is used, however, it must do a lot of work very quickly if it isn't to slow down the entire processor. This is a real problem because each page has its own protection and addressing information; at 4 Kb per page frame even a relatively small 16 Mb address space contains about 4,000 pages. The needed information for all these pages can't be stored on the processor itself, but instead must be stored in memory. The memory structure comes in two parts: a Page Directory, which tells where each page's information is, and a Page Table, which contains a descriptive entry for each page. Getting a given page takes a read of the Page Directory, which helps us generate an address for a read of the Page Table, which gives the needed information for finding the actual page. The structures involved are described elsewhere. How can we have paging without a ruinous number of memory accesses?

The answer is the same as for memory caching: store the most-used information in a small, fast cache. The page cache is on the 80386 itself in the paging unit, and is called the Page Cache or Translation Lookaside Buffer (TLB). This name comes from the way in which the paging unit "looks aside" into this buffer to check for the needed address translation information before accessing the Page Directory table in memory.

Microprocessor designers have a lot of things to put on a VLSI chip like the 80386, which has about 275,000 transistors. Once things like prefetching, general registers, and paging are handled, only one cache can be provided on-chip. There are three different things we'd like to cache if we could: code, addressing information, and data. The 80386's designers chose to have an addressing cache on-chip (the TLB) in the paging unit.

The TLB has four sets of eight entries each (see Figure 7-6). Each TLB entry has two parts. The first is called VAh, the high 17 bits of the page's Virtual Address. The second part is a copy of the actual 32-bit Page Table Entry, which would normally be read from memory.

This next part is complicated, but read on if you really want to know how the TLB works. Three bits from the middle of the Virtual Address, bits 14-12, give us an offset from 0 through 7. Remember, the TLB has four sets of eight entries each, so the three bits point to one entry from each set.

Each of the four entries is read out of its set at the same time. The high 17 bits of the current Virtual Address are compared to the 17-bit

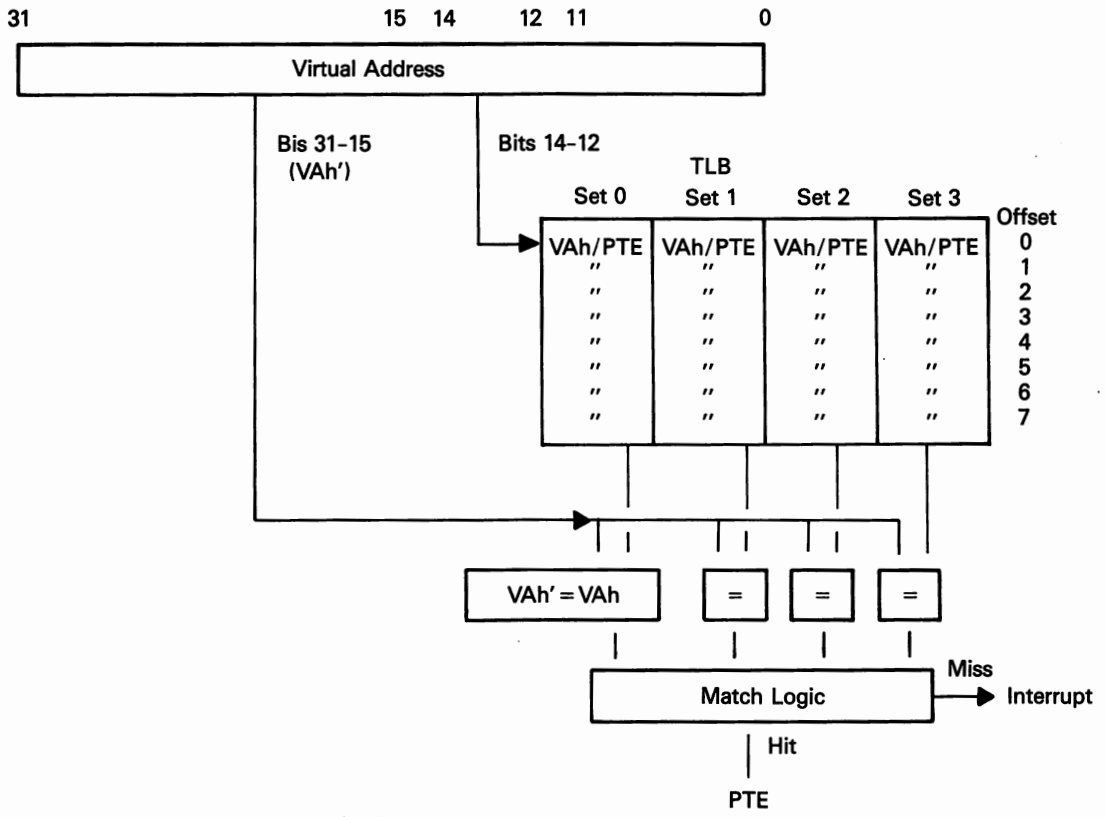


Figure 7-6. Sample Diagram for a 32-entry TLB.

VAh field in each entry. If one of them matches, the Page Table entry attached to the TLB entry is the one we need, and no memory access is needed.

Now, if you're reading this with your calculator at hand hoping to catch a mistake, you might say, "Wait! 17 bits of the Virtual Address isn't enough to determine a page from any other page. 20 bits are needed." However, the 17 bits from the VA combined with the 3 bits used as an index into the TLB make 20 bits. Note that the foldout of the 80386 shows 20 bits going to the Page Cache for use in checking. This trick means that every page in memory is mapped into one of the eight offset levels in the TLB. When copied into the TLB the page must go to the appropriate offset level; which set it goes in is determined by hardware. It doesn't matter which set of the four is used because the entry from every set is compared to VAh every time.

A match is called a page hit. It might seem surprising that 98% of paged memory addresses are page hits, but the 32 entries in the TLB allow (32 entries × 4 Kb/page =) 128 Kb of memory to be addressed via these entries. Since most programs use the same few pages most of the time, this cache is large enough to give us a high number of page hits. Page misses cause the processor to go to memory for the required page information, which takes two separate memory accesses. An applications programmer need do little to increase page hits, since the program may not even be running on a paged system. However, programs that jump around very large data areas (> 128 Kb) might cause more page misses than others. Many of the same considerations are in effect here as with memory caching.

The paging unit yields a physical address. This could have been attained in one of three ways: paging was off so the linear address was used unchanged; a TLB page hit allowed the necessary address to be constructed and checked without extra memory accesses; or a TLB miss caused memory accesses that may or may not have disrupted pipelining and slowed the processor.

BUS UNIT

The bus unit controls all communications between the 80386 and the outside world, except for the pins that go to the control unit and the CLK2 clock signal. Data transfer requests from the execution unit and code fetch requests from the prefetch unit are prioritized (code fetches last) and filled by the bus unit. The bus unit also handles interactions with coprocessors and other chips that can control the system bus.

Instruction Execution and the 80386

This section ties together our newly minted understanding of the inner workings of the 80386 by examining how a pair of instructions operate within the chip. We look at what happens during each 80386 clock cycle as the two instructions move through each unit to completion. This brief example will help improve 80386 programming skills by showing in depth how instructions interact with the processor and each other. Listing 7-2 shows the two instructions that will be executed. The foldout of 80386 internals is also helpful for following the action.

For simplicity we assume that the code is being executed just after a jump instruction, which causes the pipeline of the 80386 (especially the

Address	Contents (Code)	Instruction
00120000	03 D9	ADD EBX, ECX
00120002	64: 2B B4 DA 01234567	SUB ESI, FS: 01234567h [EDX][EBX*8]
0012000A		

Listing 7.2. Example Instructions

prefetch and decode queues) to clear. The prefetcher starts by getting the first dword of the instruction stream (the code at 00120000 and above) and storing the four bytes in the prefetch queue. The prefetcher will keep taking advantage of otherwise unused bus cycles to fill its queue until it's full, and then continue fetching whenever the queue begins to empty. We won't pay much attention to the prefetcher after this, since its work is vital but simple.

CYCLE 1

ADD: The decode unit goes to work as soon as a single byte appears in the prefetch queue. The decode unit grabs the byte (which has the value 03) and decodes it, placing the information in the Decoded Instruction Queue (a decode queue fill pointer points to the first empty entry in the decode queue). The first decoded byte tells the decoder that the instruction is an ADD and that it has a ModRM byte, which indicates that there is more to follow.

CYCLE 2

ADD: The decoder reads in byte 2 from the prefetcher, which is the ADD's ModRM byte, value D9. This byte indicates that the EBX and ECX registers are to be used and that there is no more work to do on this instruction. The decoded instruction is marked READY and left in the queue for execution by the control unit. The queue fill pointer is advanced to point to the next empty entry, as will happen after each successful instruction decode.

CYCLE 3

ADD: No prior instructions are in the pipeline, so this instruction begins executing immediately. The first microword of the instruction is read out of the Control ROM, and the two operands (EBX and ECX) are read out of

the register file in the execution unit onto the displacement and index buses.

In the second half of the cycle the two values are moved into the ALU, with microcode specifying that they be added together. The result is placed in the AR_OUT register.

SUB: The decoder sees that the SUB has an FS segment override prefix, value 64. The queue entry's segment register field will be changed to specify FS instead of the default DS.

CYCLE 4

ADD: Nothing happens in the execution unit during the first part of the cycle. During the second part the contents of AR_OUT are written into EBX in the register file, and flag bits in EFlags are updated to reflect the result of the ADD. This completes the ADD instruction. Note that the ADD instruction is listed in the instruction set as taking two clocks, but it took four here due to the fact that pipelining was inactive and all the queues were empty when this code began executing.

SUB: The decoder translates the SUB instruction byte, value 2B, which indicates a ModRM byte follows.

If this instruction were completely decoded the displacement would be bypassed around the register file and onto the displacement bus, while the index or base (whichever is specified) was read out of the register file and placed on the index bus.

CYCLE 5

All cycles after this are executing **SUB** only.

The decoder translates the ModRM (mode register/memory) byte, value B4, which indicates that a SIB byte follows. Normally this decoding would be pipelined with other instruction execution, but the queue was emptied by the JMP.

CYCLE 6

The decoder translates the SIB (scale factor-index-base) byte, value DA, which indicates that a 4-byte displacement follows.

CYCLE 7

The decoder puts a 4-byte displacement, value 01234567H, directly in the displacement slot of the queue entry for the SUB instruction. The decoding of the SUB instruction is now finished.

CYCLE 8

As described above, the displacement is bypassed around the register file and onto the displacement bus while the index is read out of the register file (EBX) and placed on the index bus.

The end of the cycle sees the index scaled by 8 and added to the displacement, with the result going into a holding register on the segmentation unit.

CYCLE 9

A base register has been specified, in addition to the already accounted-for displacement and index, so the base is read out of the register file (EDX), the previous result comes out of the holding register, and the two are combined in the adder. This is the extra cycle needed when instructions use a base, a displacement, and an index.

Note that if we had listed an instruction after the SUB it would have been prefetched and decoded, and would have reached the execution unit by now.

CYCLE 10

The specified segment register (FS) is read out of the register file and the limit of this segment is compared with the effective address in the holding register and other logic checks to see whether this segment can be read from (Access Rights in the foldout), since the result will be written to a register.

In the second part of the cycle the base address of the segment is read from the segment descriptor and added to the effective address to form the linear address, which is placed in a holding register in the paging unit.

CYCLE 11

The linear address is used to access the TLB (which we'll assume gives us a TLB hit) and the physical address for the data is calculated by merging the upper 20 bits from the TLB entry (which name a page frame) and the lower 12 bits from the linear address (which name a byte within the page).

At the end of the cycle the resulting physical address is latched (placed in a holding register).

CYCLES 12 THROUGH 14

We assume that the bus unit is free and that the memory attached to the system bus has one wait state (is 3-cycle memory). These cycles are then spent retrieving the data from memory. Note that memory accesses that are part of executing an instruction aren't pipelined. At the end of Cycle 14 the data is latched (stored) in the bus unit, where it can be accessed.

CYCLE 15

The data from memory is bypassed to one input of the ALU, the value in ESI is read into the other, and the subtraction is done with the result placed in AR_OUT.

CYCLE 16

For the first half of the cycle nothing happens; in the second half the new value is written from AR_OUT to ESI and the flag bits are conditioned accordingly.

RIPLEY'S BELIEVE IT OR NOT

The example above included a two-byte ADD instruction, which took four cycles to execute, and an eight-byte SUB which took an additional 12 cycles to execute. Listing 7-3 shows examples from among the longest possible meaningful instructions on the 80386. AS, OS, and FS are address-size, operand-size, and segment override prefixes.

```
INSTRUCTION 1:  LOCK: AS: OS: FS: BTS 12345678H [EDI]
                      [EBX*4], 24
CODE:       F0: 67: 66: 64: 0F BA 94 9F 12345678 18

INSTRUCTION 2:  LOCK: AS: OS: FS: ADD 12345678H [EDI] [EBX*4],
                      9ABCDE02H
CODE:       F0: 67: 66: 64: 81 84 9F 12345678 9ABCDE02
```

Listing 7-3. Longest Meaningful Instructions

Instruction 1 is 13 bytes long and takes 10 cycles to decode and 9 cycles to execute. Instruction 2 is 15 bytes long and takes 9 cycles to decode and 8 cycles to execute. Both instructions are noteworthy for taking longer to decode than to execute.

SOME APPLICATIONS PROGRAMMING CONSIDERATIONS

Some simple rules for writing efficient 80386 applications programs come out of the discussion in this and other chapters:

1. *Minimize operating system calls.*
   This is normally a good idea, since a single OS call can cause the execution of thousands of bytes of code, but it's an especially good idea on an 80386-based system, where a program's operating system may be running under a hypervisor that has final control of system resources. In this case, a call to the operating system may cause a further call to the hypervisor, causing longer than usual delays.
2. *Align program modules on page boundaries.*
   In a paged system large program modules that are page-aligned at the start will help prevent page swaps.
3. *Assist locality of reference*
   The principle of locality of reference is important to memory caches, page mode RAMs, the prefetcher, the TLB paging queue, and other aspects of 80386-based systems. Programs with have small, tightly written modules will execute more efficiently because they will tend to take advantage of these built-in speedup techniques.
4. *Put those instructions that are quick to decode and long to execute at the top of a loop.*

A register-to-register MUL is a good example of such an instruction. These instructions will tend to help the 80386 catch up after a jump causes its onboard queues to be emptied.

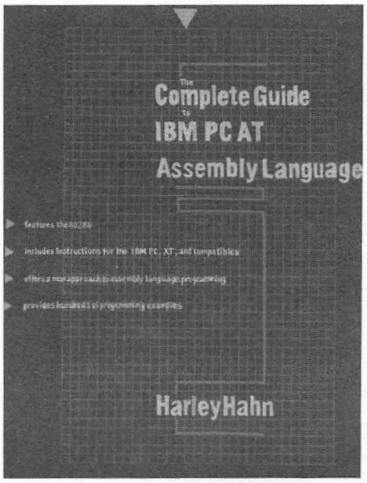
5. *Targets of jumps should be dword-aligned*
   This will prevent the processor, which is already slowed by having its queues emptied, from also having to make extra memory accesses to get its first instruction after the jump.
6. *Memory operands should be aligned on their natural boundaries.*
   Dword operands should be dword aligned, word operands should be word aligned, and the first bit in a series of bit fields should be byte, word, or dword aligned (depending on the number and length of the bit fields involved).
7. *Records and other large data structures should be dword- or even page-aligned.*
   Dword alignment helps speed access to a record. Page alignment should be considered whenever a record or other structure approaches 2 Kb (half a page) in size. This will improve access speed within the structure and improve performance in a page-based Virtual Memory system.

Index

More IBM Books from Scott, Foresman and Company

The Complete Guide to IBM PC AT Assembly Language

By **Harley Hahn**
608 pages
$24.95
18263-0



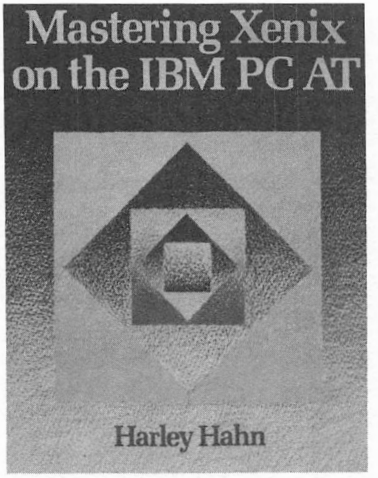
This book is an excellent introduction to programming the 80286 processor and is perfect for either the beginning programmer or the experienced programmer new to assembly language. Both a tutorial and a reference, *The Complete Guide to IBM PC AT Assembly Language* offers a new approach to assembly language programming and includes hundreds of examples, with many specific instructions for the IBM PC and XT, too. In a refreshingly readable style, Hahn succeeds at presenting complex concepts in carefully defined, easy-to-understand terms. This book

- Provides instructions on how to construct, process and run your own programs
- Offers valuable suggestions on defining data, control flow, and string instructions
- Contains numerous practical techniques for working with bits, interrupts, input/output, and much more
- Includes a useful command summary and a comprehensive glossary of technical terms

More IBM Books from Scott, Foresman and Company

Mastering Xenix on the IBM PC AT

By **Harley Hahn**
416 pages
$21.95
18260-6



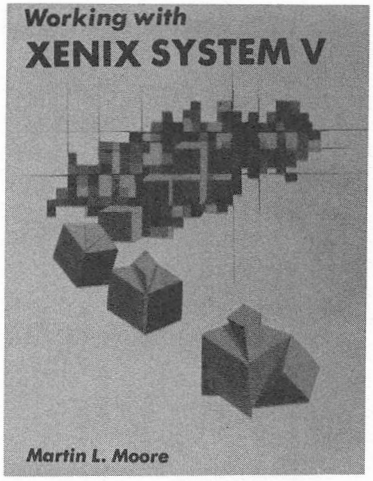
The only book specifically devoted to Xenix on the IBM PC AT, this handbook gives you a thorough introduction to IBM Xenix, one of the most powerful operating systems for small computers. The author starts with the basics and progresses to more advanced Xenix concepts, helping you understand and use the Xenix file system, send and receive messages with the Xenix mail system, program the shell, and more. The second book in our popular series on the PC AT, this book will give you a sound understanding of the complex Xenix operating system. This book

- Introduces computer novices and experienced Unix users to IBM Personal Computer Xenix
- Focuses on the *practical* uses of Xenix, and makes a complex system easy to understand
- Provides a clear overview of the operating system, detailed explanations of important concepts, and a wealth of examples
- Explains how to use the *vi* editor to create and modify your own text files
- Shows you how to work with the major Xenix commands and options
- Includes a detailed glossary and a summary of *vi* commands

More IBM Books from Scott, Foresman and Company

Working With Xenix System V

By **Martin L. Moore**
256 pages
$19.95
18080-8



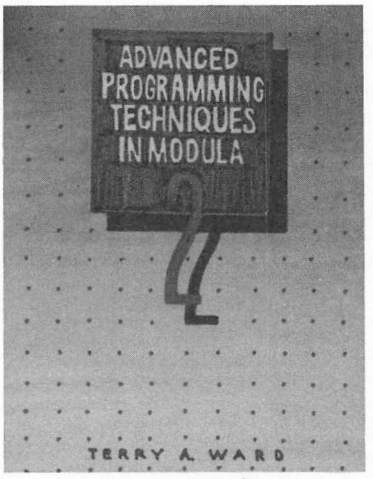
This is the *first* book on the new Xenix System V. Both a tutorial for novice Xenix users and a comprehensive reference guide, *Working With Xenix System V* helps you learn enough about Xenix to use it productively. Written for non-programmers, this book takes the mystery out of the latest version of the Microsoft Xenix operating system. *Since Xenix System V is now compatible with Unix System V, this book will also help you use the Unix operating system on a variety of personal computers.* Learn to use your Xenix- or Unix-based computer more efficiently with this complete user's handbook. This book

- Shows you how to do useful applications with Microsoft Xenix on your IBM PC XT, AT, or compatible
- Leads you through the most common Xenix operations, covering commands you'll use every day
- Includes a thorough guide to the Xenix file structure and the shell
- Explains how to use the *vi* text editor and the *mm* macros for word processing
- Includes a quick-reference command dictionary which describes the most useful Xenix commands in detail
- Provides a handy chart listing the key command differences between Microsoft, Xenix, AT&T Unix System V, and the Berkeley Version 4.2BSD Unix

More IBM Books from Scott, Foresman and Company

Advanced Programming Techniques Modula-2

By **Terry Ward**
400 pages
$21.95
18615-6



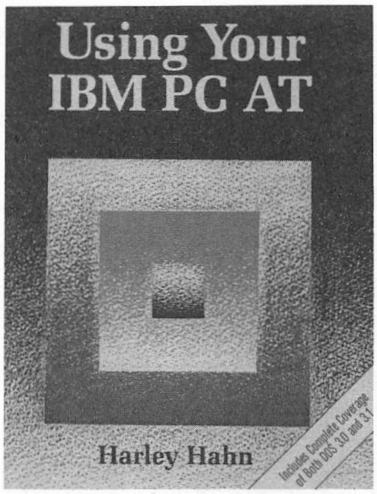
Advanced Programming Techniques in Modula-2 presents a wide variety of tools, tips, and techniques for advanced Modula-2 programmers. Beginning with a brief overview of the language, this book includes many examples of advanced programming projects. This book

- Includes benchmark utilities for testing compiler performance
- Details a set of specialized module libraries for extending Modula-2, including string handling, set operations, and bit and matrix manipulations
- Provides a set of sorting modules using various sorting algorithms
- Features a set of interface modules for use with the reader's programs

More IBM Books from Scott, Foresman and Company

Using Your IBM PC AT

By **Harley Hahn**
351 pages
$19.95
18262-2



This valuable reference guide introduces you to IBM's bestselling new personal computer, the PC AT. Written in an easy-going, nontechnical style, this book is designed to get you using your computer immediately. The author clearly guides you from the basics to advanced features of the AT, with an emphasis on making a complicated system understandable. This book also includes a list of popular software programs that work on the PC AT, many examples, and a variety of information not found in the computer's manual. This book

- Tells you everything you need to know to use your PC AT effectively
- Gives detailed information on the latest versions of DOS—versions 3.0 and 3.1
- Includes a clear overview of PC AT hardware, software, and peripherals
- Offers thorough definitions of important computer terms and concepts
- Provides techniques for managing disks and files
- Offers a helpful summary of DOS commands and error messages
- Explains such advanced features as tree-structure files and the line editor

Order Form

To order, contact your local bookstore or send this form to

Scott, Foresman and Company
Professional Publishing Group
1900 East Lake Avenue
Glenview, IL 60025
(312) 729-3000

In Canada, contact
Macmillan of Canada
164 Commander Blvd.
Agincourt, Ontario
MIS 3C7

Qty	Code #	Title	Price
Total Order			$
State and/or Local Taxes			$
6% of Total before taxes for postage*			$
Total			$

Please check method of payment:

☐ Check/Money Order (Make checks payable to Scott, Foresman and Company)

Amount enclosed $_____

☐ MasterCard ☐ VISA

Credit Card No. _____ Exp. _____

Signature _____

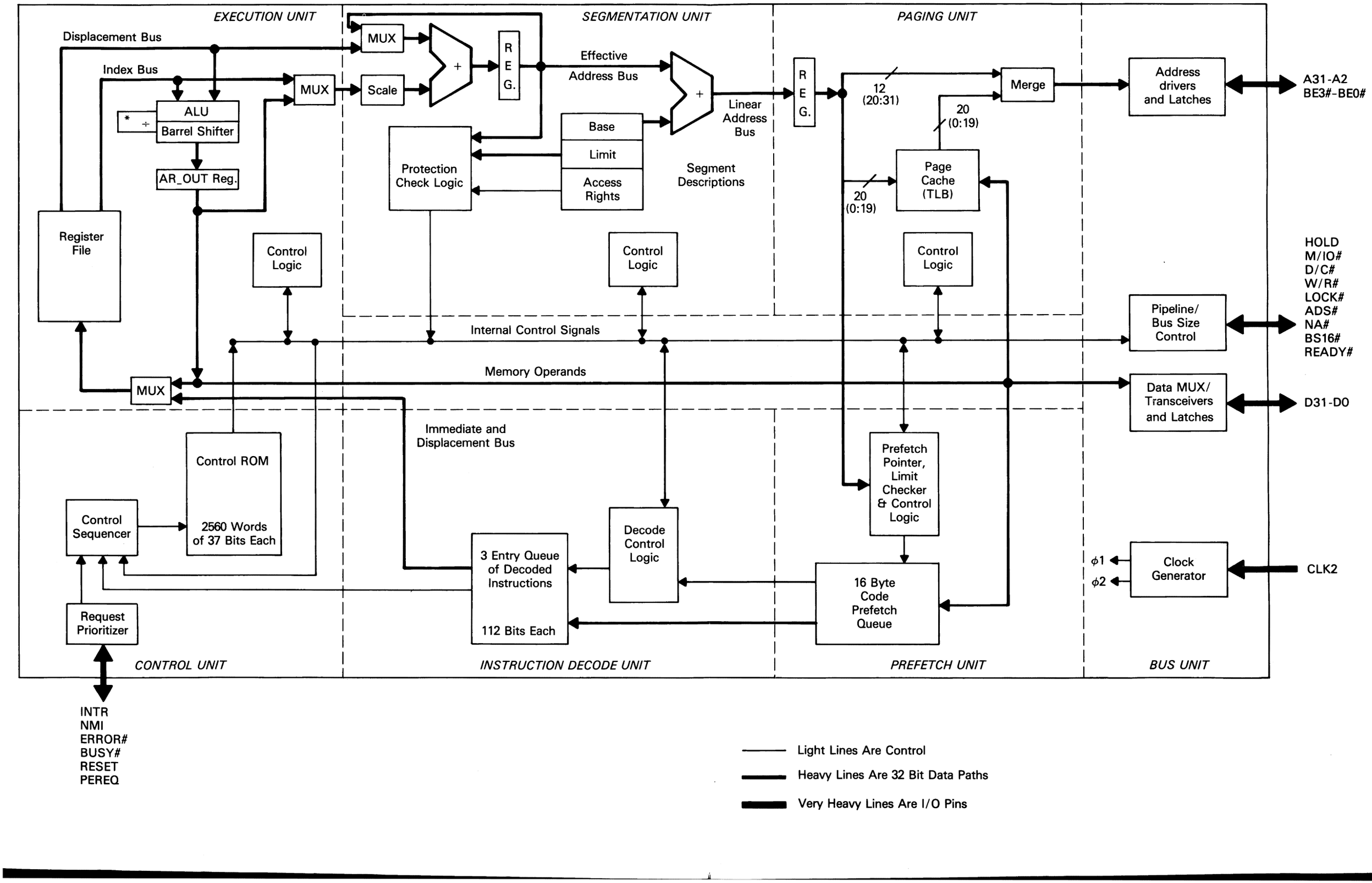
Name (please print) _____

Address _____

City _____ State _____ Zip _____

* If you enclose a check with your order, there is no charge for postage.

Full payment must accompany your order. Prices subject to change without notice.


EXECUTION UNIT
SEGMENTATION UNIT
PAGING UNIT
Displacement Bus
Index Bus
MUX
Scale
+
R E G.
Effective Address Bus
Base
Limit
Access Rights
Segment Descriptions
Linear Address Bus
Protection Check Logic
ALU
Barrel Shifter
AR_OUT Reg.
Register File
Control Logic
Internal Control Signals
Memory Operands
Merge
12 (20:31)
20 (0:19)
Page Cache (TLB)
Address drivers and Latches
A31-A2
BE3#-BE0#
HOLD
M/IO#
D/C#
W/R#
LOCK#
ADS#
NA#
BS16#
READY#
Pipeline/ Bus Size Control
Data MUX/ Transceivers and Latches
D31-D0
Immediate and Displacement Bus
Control ROM
2560 Words of 37 Bits Each
Control Sequencer
Request Prioritizer
3 Entry Queue of Decoded Instructions
112 Bits Each
Decode Control Logic
Prefetch Pointer, Limit Checker & Control Logic
16 Byte Code Prefetch Queue
Clock Generator
φ1
φ2
CLK2
CONTROL UNIT
INSTRUCTION DECODE UNIT
PREFETCH UNIT
BUS UNIT
INTR
NMI
ERROR#
BUSY#
RESET
PEREQ
Light Lines Are Control
Heavy Lines Are 32 Bit Data Paths
Very Heavy Lines Are I/O Pins

>$22.95

Programming the Intel 80386

Written for both novice and experienced programmers, this complete guide offers an in-depth look at the new 80386 processor. Following a brief discussion of assembler and microprocessors, this valuable reference explains timing, system design, processor layout, and efficiency in programming. Special features include

- a fold-out diagram of the chip's pinout
- a complete instruction set, including detailed programmer's notes
- techniques for programming in the protected mode

You'll also find details about the differences between real mode, protected mode, and virtual 8086 mode, and much more.

Full of information not found anywhere else, **Programming the Intel 80386** is just the resource you need to use the powerful new Intel 80386 microprocessor.

Bud E. Smith is a computer programmer, technical writer, and data processing supervisor. He has worked for Cox Cable San Diego and Megahaus, Inc., and has had wide experience as a consultant. He is also a consulting editor to *Exclusive Review*, a monthly newsletter for Apple II and AppleWorks users.

Mark T. Johnson has been programming since 1968. He has been employed in computer centers at both Michigan State University and the University of Queensland. He has also worked with computers at several companies in the United States and Australia, and is currently an independent consultant in computer graphics and electronic publishing.

PROPERTY OF THE MARK WILLIAMS LIBRARY

Mark Williams Company

RESMAN AND COMPANY

ISBN 0-673-18568-0